

# *Аква в действии*

Раймонд Рестенбург  
Роб Баккер  
Роб Уильямс



Москва, 2018

# *Akka in Action*

RAYMOND ROESTENBURG  
ROB BAKKER  
ROB WILLIAMS



MANNING  
SHELTER ISLAND

Раймонд Рестенбург, Роб Баккер, Роб Уильямс

# Акка в действии

УДК 004.424  
ББК 32.972  
P43

P43 Раймонд Рестенбург, Роб Баккер, Роб Уильямс

Акка в действии / пер. с англ. А. Н. Киселев – М.: ДМК Пресс, 2018. – 522 с.: ил.

**ISBN 978-5-97060-642-1**

В книге рассказывается о фреймворке Akka и описываются его наиболее важные модули. Большое внимание уделено модели программирования с акторами и модулям поддержки акторов, часто используемых при создании конкурентных и распределенных приложений. Продемонстрированы подходы к разработке через тестирование и приемы развертывания и масштабирования отказоустойчивых систем. Во всех примерах книги используется язык программирования Scala.

Издание адресовано разработчикам на Java и Scala, желающим научиться создавать приложения с использованием фреймворка Akka.

УДК 004.424  
ББК 32.972

Original English language edition published by Manning Publications. Copyright © 2017 by Manning Publications. Russian language edition copyright © 2018 by DMK Press. All rights reserved.

Все права защищены. Любая часть этой книги не может быть воспроизведена в какой бы то ни было форме и какими бы то ни было средствами без письменного разрешения владельцев авторских прав.

Материал, изложенный в данной книге, многократно проверен. Но, поскольку вероятность технических ошибок все равно существует, издательство не может гарантировать абсолютную точность и правильность приводимых сведений. В связи с этим издательство не несет ответственности за возможные ошибки, связанные с использованием книги.

ISBN 978-1-61729-101-2 (англ.)  
ISBN 978-5-97060-642-1 (рус.)

Copyright © 2017 by Manning Publications Co.  
© Оформление, перевод на русский язык,  
издание, ДМК Пресс, 2018

# Оглавление

<b>Предисловие</b> .....	<b>11</b>
<b>Благодарности</b> .....	<b>12</b>
<b>О книге</b> .....	<b>13</b>
Кому адресована эта книга .....	13
Содержание книги .....	13
Соглашения об оформлении программного кода .....	15
Требования к программному обеспечению .....	15
Автор в сети.....	15
Об авторах .....	16
Об иллюстрации на обложке.....	16
<b>Глава 1. Введение в Akka</b> .....	<b>18</b>
1.1. Что такое Akka?.....	22
1.2. Актеры: краткий обзор .....	22
1.3. Два подхода к масштабированию: подготовка примера .....	24
1.4. Традиционное масштабирование .....	26
1.4.1. Традиционный подход к масштабированию и хранению: переместить все в базу данных .....	26
1.4.2. Традиционное масштабирование и интерактивная работа: опрос.....	29
1.4.3. Традиционное масштабирование и интерактивная работа: обработка ошибок .....	31
1.5. Масштабирование с Akka.....	32
1.5.1. Подход к масштабированию и хранению с Akka: отправка и прием сообщений .....	33
1.5.2. Масштабирование с Akka и интерактивная работа: отправка сообщений .....	36
1.5.3. Масштабирование с Akka и отказы: асинхронное разделение .....	37
1.5.4. Подход Akka: отправка и получение сообщений .....	37
1.6. Актеры: универсальная модель программирования .....	39
1.6.1. Модель асинхронного выполнения.....	40
1.6.2. Операции с актерами.....	41
1.7. Актеры Akka .....	44
1.7.1 ActorSystem .....	45
1.7.2. ActorRef, почтовый ящик и актер .....	47
1.7.3. Диспетчеры .....	47
1.7.4. Актеры и сеть .....	49
1.8. В заключение .....	49
<b>Глава 2. Подготовка и запуск</b> .....	<b>51</b>
2.1. Клонирование, сборка и интерфейс тестирования .....	52

2.1.1. Сборка с помощью sbt.....	53
2.1.2. Забегая вперед: REST-сервер GoTicks.com.....	54
2.2. Исследование акторов в приложении.....	59
2.2.1. Структура приложения .....	59
2.2.2. Актор, осуществляющий продажу: TicketSeller.....	64
2.2.3. Актор VoxOffice .....	65
2.2.4. Актор RestApi.....	67
2.3. Вперед, в облако .....	70
2.3.1. Создание приложения в облаке Heroku .....	71
2.3.2. Развертывание и запуск в Heroku .....	72
2.4. В заключение .....	73
<b>Глава 3. Разработка с акторами через тестирование.....</b>	<b>75</b>
3.1. Тестирование акторов.....	76
3.2. Односторонние взаимодействия .....	78
3.2.1. Примеры SilentActor .....	79
3.2.2. Пример SendingActor .....	84
3.2.3. Пример SideEffectingActor .....	89
3.3. Двусторонние взаимодействия.....	92
3.4. В заключение .....	93
<b>Глава 4. Отказоустойчивость.....</b>	<b>95</b>
4.1. Что такое отказоустойчивость.....	95
4.1.1. Простые объекты и исключения .....	98
4.1.2. И пусть падает .....	103
4.2. Жизненный цикл актора .....	107
4.2.1. Событие start .....	107
4.2.2. Событие stop.....	108
4.2.3. Событие restart .....	109
4.2.4. Объединяем фрагменты жизненного цикла вместе.....	111
4.2.5. Мониторинг жизненного цикла .....	113
4.3. Супервизор .....	114
4.3.1. Иерархия супервизора .....	114
4.3.2. Предопределенные стратегии .....	117
4.3.3. Собственные стратегии.....	118
4.4. В заключение .....	124
<b>Глава 5. Объекты Future.....</b>	<b>125</b>
5.1. Примеры использования объектов Future .....	126
5.2. Объекты Future не блокируют выполнение потока .....	131
5.2.1. Объекты Promise – это обещания.....	135
5.3. Обработка ошибок в объектах Future .....	138
5.4. Комбинирование объектов Future .....	143
5.5. Объединение объектов Future с акторами .....	152

5.6. В заключение .....	153
<b>Глава 6. Первое распределенное приложение .....</b>	<b>155</b>
6.1. Горизонтальное масштабирование .....	156
6.1.1. Общая терминология .....	156
6.1.2. Причины использования модели распределенного программирования .....	158
6.2. Горизонтальное масштабирование и удаленные взаимодействия ....	159
6.2.1. Реорганизация приложения GoTicks.com .....	161
6.2.2. Удаленные взаимодействия в REPL .....	161
6.2.3. Удаленный поиск.....	167
6.2.4. Удаленное развертывание .....	175
6.2.5. Тестирование с multi-JVM.....	180
6.3. В заключение .....	186
<b>Глава 7. Настройка, журналирование и развертывание .....</b>	<b>188</b>
7.1. Настройка.....	188
7.1.1. Попытка настройки Akka .....	189
7.1.2. Использование значений по умолчанию.....	192
7.1.3. Настройка Akka .....	195
7.1.4. Настройка для нескольких систем.....	196
7.2. Журналирование .....	199
7.2.1. Журналирование в приложении Akka .....	199
7.2.2. Использование журналирования .....	201
7.2.3. Управление журналированием из Akka .....	202
7.3. Развертывание приложений на основе акторов .....	204
7.4. В заключение .....	208
<b>Глава 8. Шаблоны структуризации акторов.....</b>	<b>210</b>
8.1. Конвейеры и фильтры .....	211
8.1.1. Шаблон: конвейеры и фильтры.....	211
8.1.2. Конвейеры и фильтры в Akka .....	212
8.2. Параллельная обработка дроблением с последующим объединением результатов .....	216
8.2.1. Область применения .....	216
8.2.2. Распараллеливание задач в Akka .....	218
8.2.3. Реализация компонента дробления с использованием списка получателей .....	219
8.2.4. Реализация компонента объединения с использованием агрегатора .....	221
8.2.5. Объединение компонентов в реализацию шаблона параллельной обработки дроблением .....	227
8.3. Маршрутизация.....	229
8.4. В заключение .....	234

<b>Глава 9. Маршрутизация сообщений</b> .....	<b>236</b>
9.1. Шаблон «Маршрутизатор».....	237
9.2. Балансировка нагрузки с помощью маршрутизаторов Akka.....	238
9.2.1. Маршрутизатор с пулом .....	242
9.2.2. Маршрутизатор с группой .....	250
9.2.3. Маршрутизатор ConsistentHashing .....	257
9.3. Реализация шаблона маршрутизатора с применением акторов.....	262
9.3.1. Маршрутизация по содержимому.....	262
9.3.2. Маршрутизация на основе состояния.....	263
9.3.3. Реализации маршрутизаторов .....	265
9.4. В заключение .....	266
<b>Глава 10. Каналы обмена сообщениями</b> .....	<b>268</b>
10.1. Типы каналов.....	269
10.1.1. Точка-точка.....	269
10.1.2. Издатель/подписчик .....	270
10.2. Специальные каналы .....	280
10.2.1. DeadLetter.....	281
10.2.2. Гарантированная доставка .....	283
10.3. В заключение .....	289
<b>Глава 11. Конечные автоматы и агенты</b> .....	<b>291</b>
11.1. Использование конечного автомата.....	292
11.1.1. Краткое введение в конечные автоматы .....	292
11.1.2. Создание модели конечного автомата .....	294
11.2. Реализация модели конечного автомата.....	295
11.2.1. Реализация переходов.....	296
11.2.2. Реализация действий при входе в состояния .....	301
11.2.3. Таймеры в конечном автомате.....	305
11.2.4. Завершение конечного автомата .....	308
11.3. Реализация общего состояния с помощью агентов.....	309
11.3.1. Простой доступ к общим данным с помощью агентов .....	310
11.3.2. Ожидание изменения состояния.....	312
11.4. В заключение .....	313
<b>Глава 12. Интеграция с другими системами</b> .....	<b>315</b>
12.1. Конечные точки сообщений .....	315
12.1.1. Нормализатор .....	317
12.1.2. Модель канонических данных.....	319
12.2. Реализация конечных точек с использованием Apache Camel .....	322
12.2.1. Реализация конечной точки-потребителя для приема сообщений из внешней системы .....	323
12.2.2. Реализация конечной точки-производителя для отправки сообщений во внешнюю систему.....	330

12.3. Реализация HTTP-интерфейса .....	335
12.3.1. Пример HTTP-интерфейса.....	336
12.3.2. Реализация конечной точки REST на основе akka-http .....	338
12.4. В заключение .....	344
<b>Глава 13. Потоквые приложения.....</b>	<b>346</b>
13.1. Основы потоковой обработки .....	347
13.1.1. Копирование файлов.....	351
13.1.2. Материализация запускаемых графов .....	355
13.1.3. Обработка событий в потоке .....	360
13.1.4. Обработка ошибок в потоках.....	364
13.1.5. Создание протокола с BidiFlow.....	366
13.2. Потоквая передача данных через HTTP .....	369
13.2.1. Прием потока данных по HTTP .....	370
13.2.2. Возврат потока данных по HTTP .....	372
13.2.3. Согласование контента .....	373
13.3. Ветвление и слияние со специализированным языком описания графов .....	378
13.3.1. Ветвление потоков .....	378
13.3.2. Слияние потоков .....	381
13.4. Посредничество между производителями и потребителями .....	384
13.4.1. Использование буферов.....	385
13.5. Обособление частей графа, действующих с разной скоростью .....	389
13.5.1. Медленный потребитель, накопление событий в блоках .....	389
13.5.2. Быстрый потребитель, дополнительные показатели .....	390
13.6. В заключение .....	391
<b>Глава 14. Кластеры.....</b>	<b>393</b>
14.1. Зачем нужны кластеры? .....	393
14.2. Членство в кластере .....	395
14.2.1. Присоединение к кластеру .....	396
14.2.2. Выход из кластера .....	404
14.3. Обработка заданий в кластере .....	410
14.3.1. Запуск кластера .....	412
14.3.2. Распределение заданий с использованием маршрутизаторов.....	414
14.3.3. Надежная обработка заданий.....	417
14.3.4. Тестирование кластера.....	424
14.4. В заключение .....	428
<b>Глава 15. Хранимые акторы.....</b>	<b>430</b>
15.1. Восстановление состояния с технологией Event Sourcing.....	432
15.1.1. Обновление записей на месте .....	432
15.1.2. Сохранение состояния без изменения.....	433
15.1.3. Event Sourcing для акторов .....	435

15.2. Хранимые акторы .....	436
15.2.1. Хранимый актор .....	437
15.2.2. Тестирование .....	441
15.2.3. Моментальные снимки .....	443
15.2.4. Запрос хранимых событий .....	449
15.2.5. Сериализация .....	451
15.3. Кластер на основе хранимых акторов .....	457
15.3.1. Расширение cluster singleton .....	461
15.3.2. Расширение cluster sharding .....	465
15.4. В заключение .....	470
<b>Глава 16. Советы по повышению производительности .....</b>	<b>471</b>
16.1. Анализ производительности .....	472
16.1.1. Производительность системы .....	472
16.1.2. Показатели производительности .....	474
16.2. Оценка производительности акторов .....	477
16.2.1. Сбор данных в почтовом ящике .....	478
16.2.2. Сбор и обработка данных .....	485
16.3. Улучшение производительности устранением узких мест .....	487
16.4. Настройка диспетчера .....	489
16.4.1. Выявление проблем с пулами потоков .....	489
16.4.2. Использование нескольких экземпляров диспетчеров .....	491
16.4.3. Изменение размера пула потоков статически .....	493
16.4.4. Изменение размера пула потоков динамически .....	496
16.5. Изменение поведения механизма освобождения потоков .....	498
16.5.1. Ограничения механизма освобождения потоков .....	500
16.6. В заключение .....	502
<b>Глава 17. Заглядывая вперед .....</b>	<b>504</b>
17.1. Модуль akka-typed .....	505
17.2. Akka Distributed Data .....	509
17.3. В заключение .....	509
<b>Предметный указатель .....</b>	<b>511</b>

# 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), Бьерну 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 Roostenburg) 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 актерами 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 актерами.

*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 делает 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](http://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 Джеффериса (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 Джеффериса.

# 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. Dobbs 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 имела, 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 процессов 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;

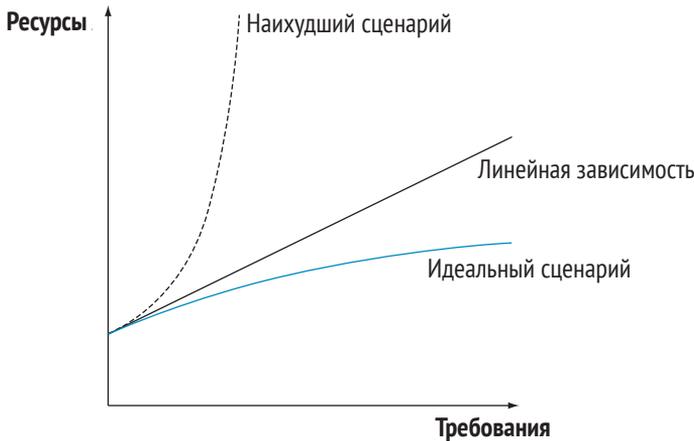


Рис. 1.1. Зависимость требований от ресурсов

- 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.

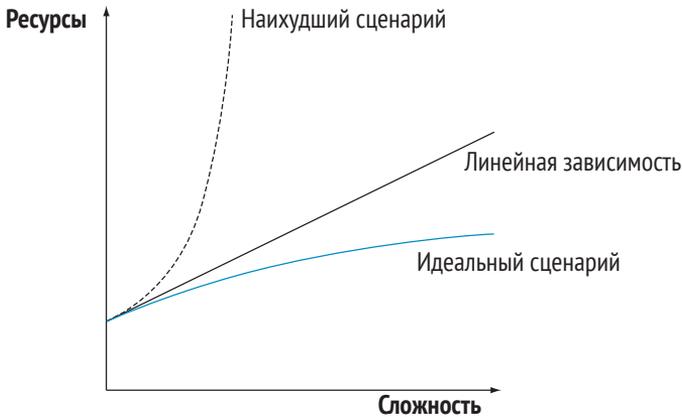


Рис. 1.2. Зависимость сложности от объема требуемых ресурсов

Один из 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 (потoki 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, разворачиваемых 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 Акка с самого начала был вовлечен в создание манифеста и фреймворк Акка – 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 расходом 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 актер – все это можно решить позднее. Этим модель акторов выгодно отличается от непосредственного использования потоков выполнения и сетевых взаимодействий в стиле RPC. Актеры 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 Акка
Масштабирование	Использовать комбинацию потоков выполнения, общего изменяемого состояния в базе данных (операции создания, добавления, изменения, удаления) и RPC-вызовов веб-служб	Актеры 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<sup>1</sup> 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 приспособляется; 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.

<sup>1</sup> [https://ru.wikipedia.org/wiki/Бережливый\\_стартап](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.

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.

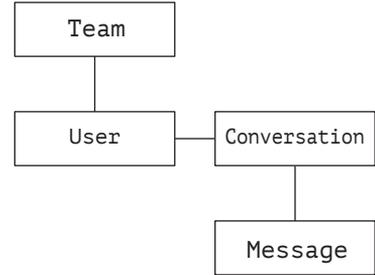


Рис. 1.3. Модель данных

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.

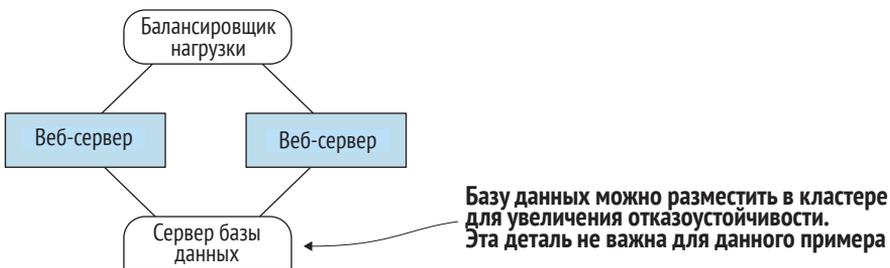


Рис. 1.4. Балансировщик нагрузки

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.

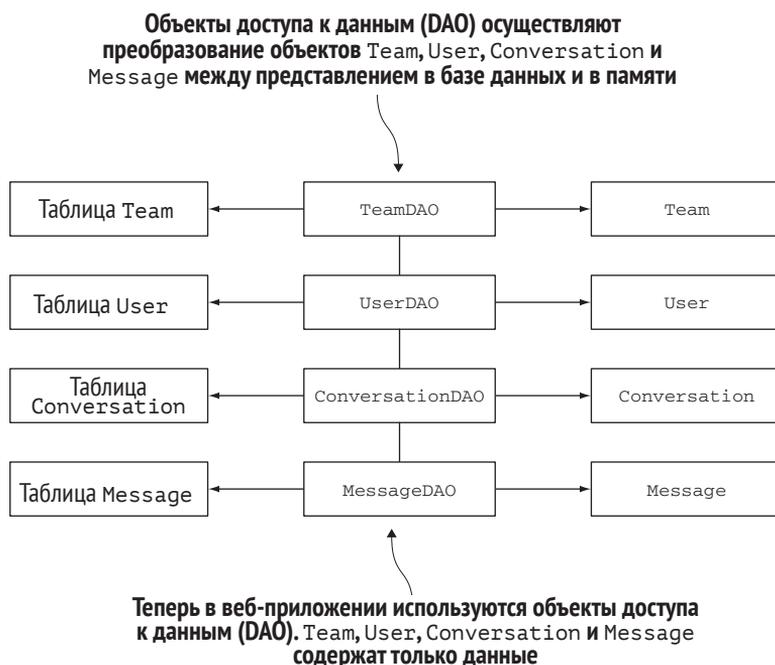


Рис. 1.5. Объекты доступа к данным

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.

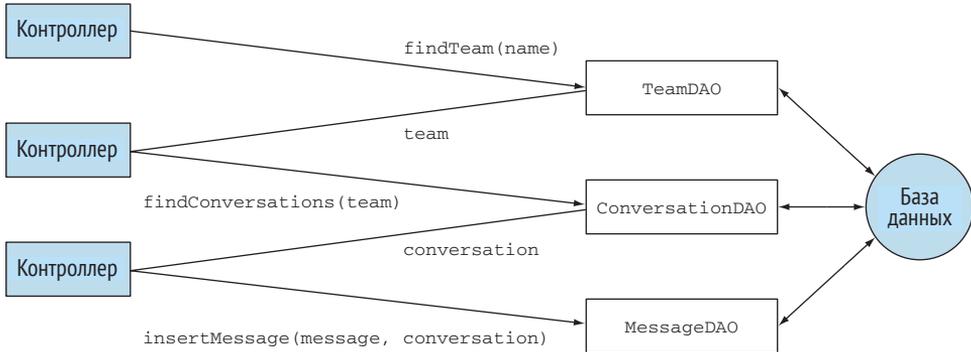


Рис. 1.6. Взаимодействие с базой данных через объекты DAO

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.

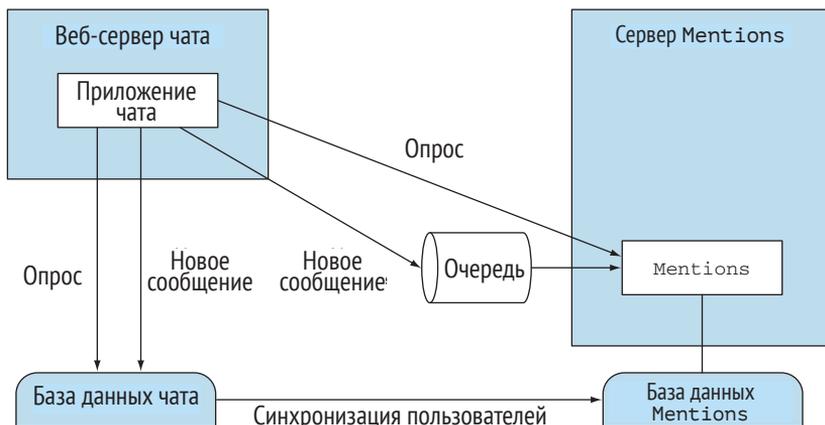


Рис. 1.7. Компонент службы

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.

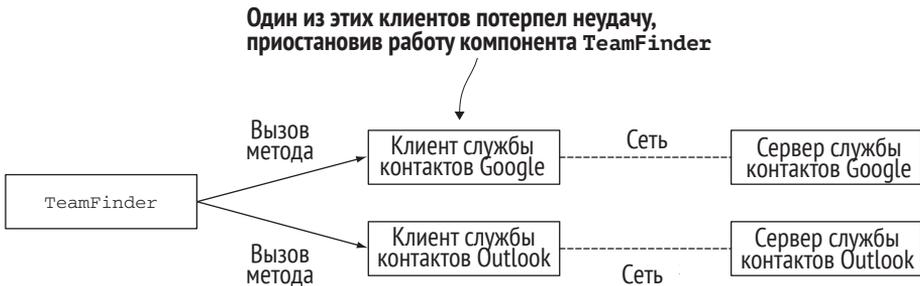


Рис. 1.8. Поиск контактов

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 потребуются? 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.

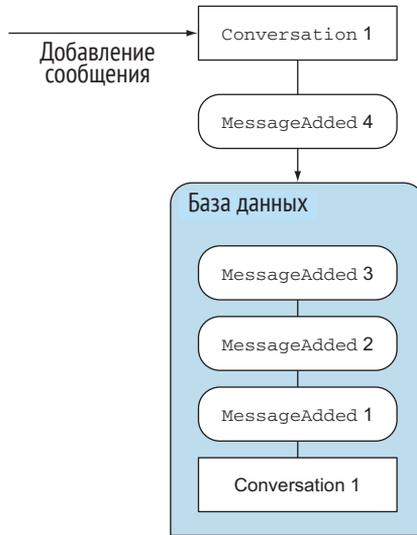


Рис. 1.9. Сохранение диалогов

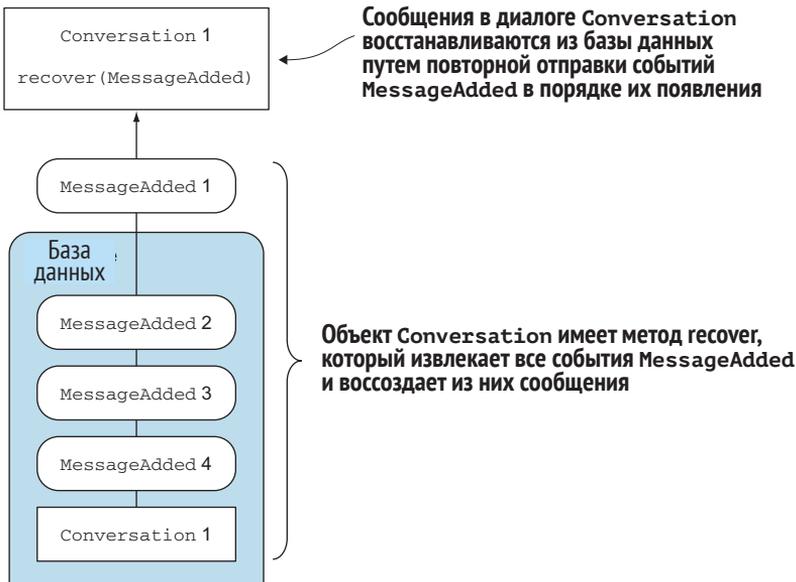


Рис. 1.10. Восстановление диалогов

### 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.

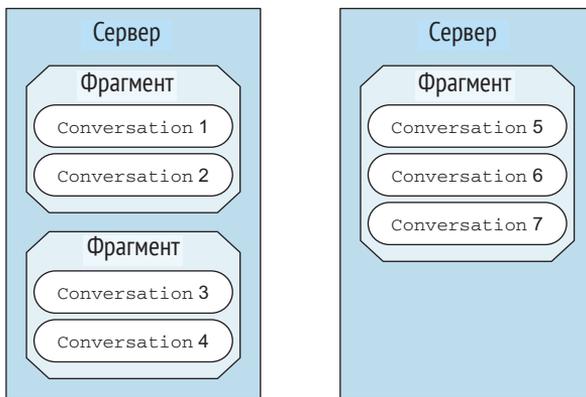


Рис. 1.11. Фрагментирование

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.

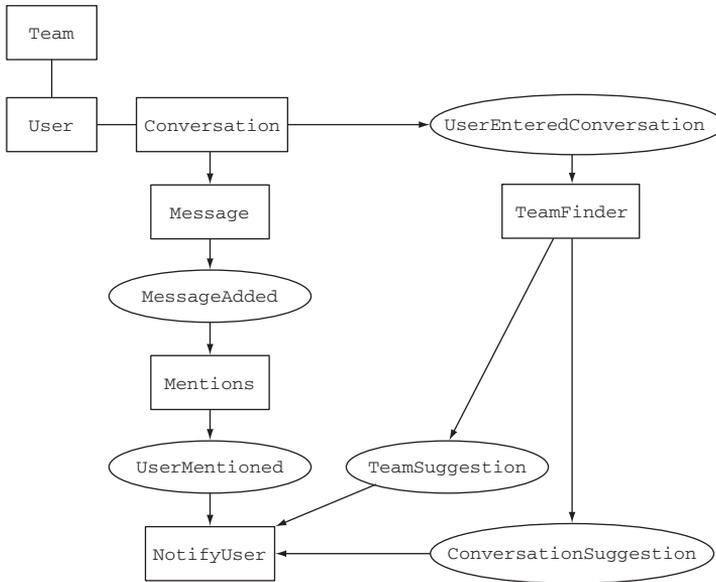


Рис. 1.12. События

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.

### 1.5.3. Масштабирование с Akka и отказы: асинхронное разделение

Желательно, чтобы пользователи могли продолжать диалог даже после того, как компонент Mentions постигнет неудача. То же касается компонента TeamFinder: существующие диалоги должны продолжать работать. Диалоги могут продолжать публиковать события, в то время как подписчики, такие как Mentions и TeamFinder, терпят неудачу или перезапускаются.

Компонент NotifyUser может запоминать подключенные браузеры и посылать им сообщения UserMentioned напрямую, избавляя приложение от необходимости периодически выполнять опрос базы данных.

Такой подход на основе событий обладает несколькими преимуществами.

- Минимизирует прямые зависимости между компонентами. Объект Conversation *не знает о существовании объекта Mentions и никак не заботится о происходящем в ответ на событие*. Диалог может продолжать действовать и после того, как объект Mentions потерпит неудачу.
- Компоненты приложения слабо привязаны ко времени. Совершенно не важно, будет ли объект Mentions получать события чуть позже, намного важнее, что в конце концов он их получит.
- Компоненты не зависят также от местоположения. Объекты Conversation и Mentions могут находиться на разных серверах; события – это просто сообщения, которые с легкостью могут передаваться по сети.

Подход на основе событий решает проблему опроса, характерную для объекта Mentions, а также избавляет от прямой зависимости от компонента TeamFinder. В главе 5, где рассказывается об объектах Future, мы рассмотрим несколько более удачных способов взаимодействий с веб-службами взамен ожиданий последовательных ответов на каждый запрос. Отметим еще раз, что решение этой проблемы можно смоделировать, просто посылая и принимая сообщения.

### 1.5.4. Подход Akka: отправка и получение сообщений

Давайте вспомним, что мы изменили к данному моменту: диалоги теперь являются полноценными объектами в памяти (актерами), хранящими свое внутреннее состояние, восстанавливаются из событий, распределены между серверами, посылают и принимают сообщения.

Взаимодействия между объектами теперь происходят путем обмена сообщениями, а не прямых вызовов методов.

Главное требование – сообщения должны посылаться и приниматься по порядку, по одному каждым актором, если одно событие зависит от другого,

потому что иначе результаты могут получиться самыми непредсказуемыми. Это требует, чтобы диалог Conversation хранил собственные сообщения в секрете от других компонентов. Порядок следования сообщений трудно будет сохранить, если любой другой компонент сможет взаимодействовать с ними.

Не имеет значения, отправляем ли мы сообщение локальному адресату, находящемуся на том же сервере, или удаленному – на другом. Поэтому нам нужна некоторая служба, которая позаботится об отправке сообщений акторам на другом сервере, если потребуется. Нам также потребуется следить за местоположением акторов и предоставлять ссылки на них, чтобы другие серверы могли взаимодействовать с этими акторами. Обо всем этом за нас позаботится Akka, как вы вскоре увидите. В главе 6 мы обсудим основы программирования распределенных приложений с Akka, а в главе 13 познакомимся с кластерными приложениями (то есть с приемами группировки распределенных акторов).

Объект Conversation совершенно не заботит происходящее в компоненте Mentions, но на уровне приложения мы должны знать, когда компонент Mentions не может продолжать работу, чтобы показать пользователям, что соответствующая возможность временно отключена. То есть нам нужен некоторый механизм мониторинга акторов, позволяющий перезапускать их при необходимости. Этот механизм должен поддерживать работу на группе серверов или на одном сервере, локально, то есть он также должен посылать и принимать сообщения. Возможная высокоуровневая структура нашего приложения, с учетом вышесказанного, изображена на рис. 1.13.

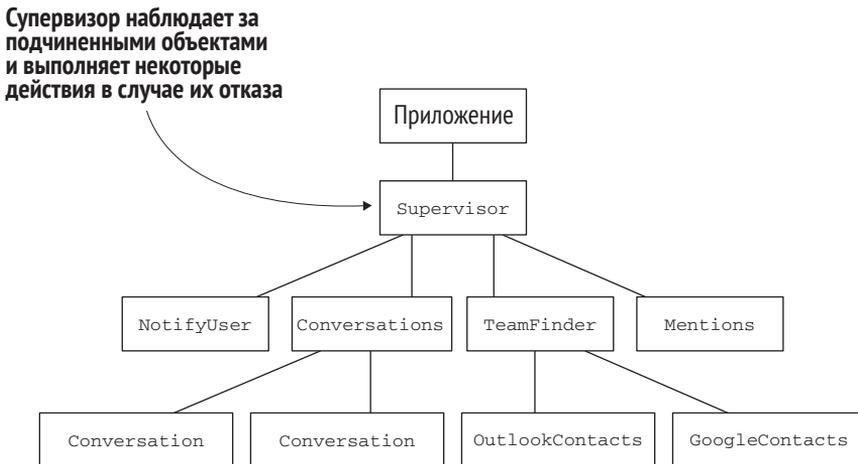


Рис. 1.13. Высокоуровневая структура

Супервизор Supervisor наблюдает за другими компонентами и предпринимает некоторые действия в случае их отказа. Он может, например, решить продолжить работу в случае отказа компонента Mentions или Team-

Finder. Если все объекты, Conversation и NotifyUser, полностью прекратили работу, супервизор может решить выполнить полный перезапуск приложения или вообще остановить его, если сложились такие условия, когда продолжение работы невозможно. Компонент может послать сообщение супервизору, столкнувшись с ошибкой, а супервизор может послать сообщение компоненту, чтобы остановить или перезапустить его. Так выглядит концепция восстановления после ошибок, реализованная в Akka, которая обсуждается в главе 4.

В следующем разделе мы сначала поговорим об акторах как таковых, а потом об акторах Akka.

## 1.6. Акторы: универсальная модель программирования

Большинство современных универсальных языков программирования предполагают последовательное выполнение программ, записанных на них (Scala и Java не являются исключениями). Модель конкурентного программирования требует заполнения разрыва между последовательными определениями и параллельным выполнением.

Если под параллелизмом понимается одновременное выполнение нескольких процессов, то под конкуренцией подразумевается определение процессов, которые *могут* функционировать одновременно или перекрываться во времени, но *не обязательно должны* запускаться одновременно. Конкурентная система не является параллельной по определению. Конкурирующие процессы могут, например, выполняться на одном процессоре, получая определенные кванты времени, следующие друг за другом.

Виртуальная машина Java (JVM) поддерживает стандартную модель конкурентного программирования (см. рис. 1.14), согласно которой, грубо говоря, процессы выражаются методами объектов, выполняющимися в отдельных потоках. Потоки могут выполняться на нескольких процессорах параллельно или на одном процессоре последовательно, используя некоторый механизм квантования времени. Как уже говорилось выше, потоки не могут использоваться для горизонтального масштабирования – только для вертикального.



Рис. 1.14. Модель конкурентного программирования

Необходимая нам модель конкурентного программирования должна поддерживать функционирование на одном или на нескольких процессорах, а также на одном или на нескольких серверах. Модель акторов использует абстракцию отправки/приема сообщений, чтобы не зависеть от количества используемых потоков выполнения или серверов.

### 1.6.1. Модель асинхронного выполнения

Чтобы приложение могло масштабироваться для выполнения на нескольких серверах, модель программирования должна соответствовать важному требованию: она должна действовать *асинхронно*, позволяя компонентам продолжать работу, когда другие не отвечают на вызовы, как в нашем приложении чата (см. рис. 1.15).

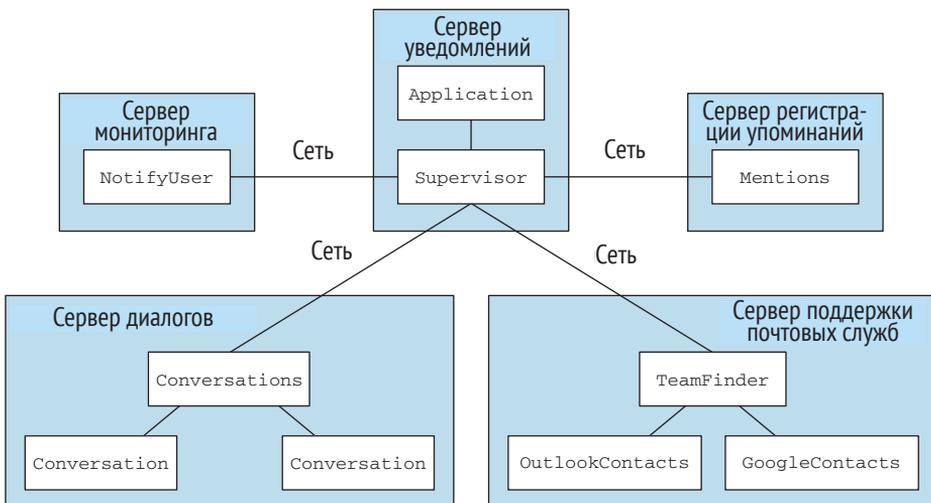


Рис. 1.15. Горизонтальное масштабирование

На рис. 1.15 изображена возможная конфигурация приложения чата, действующего на пяти серверах. Супервизор `Supervisor` отвечает за создание и мониторинг всех остальных компонентов приложения. В данной конфигурации супервизор осуществляет управление, рассылая команды по сети, которая может отключиться, так же как может отключиться любой сервер. Если супервизор будет использовать синхронную связь, ожидая каждого ответа от каждого компонента, мы могли бы попасть в ситуацию, когда один из компонентов перестает откликаться и тем самым блокирует все остальные вызовы. Представьте, что случится, например, если сервер с диалогами начнет перезагрузку, перестав отвечать на сетевые запросы, а супервизору в это время понадобится разослать сообщения всем компонентам.

## 1.6.2. Операции с актерами

Актеры являются основными строительными блоками в модели акторов. Все компоненты в примере приложения являются актерами, как показано на рис. 1.16. Актор – это легковесный процесс, поддерживающий четыре основные операции: создание (create), отправка (send), переход (become) и управление (supervise). Все эти операции выполняются асинхронно.

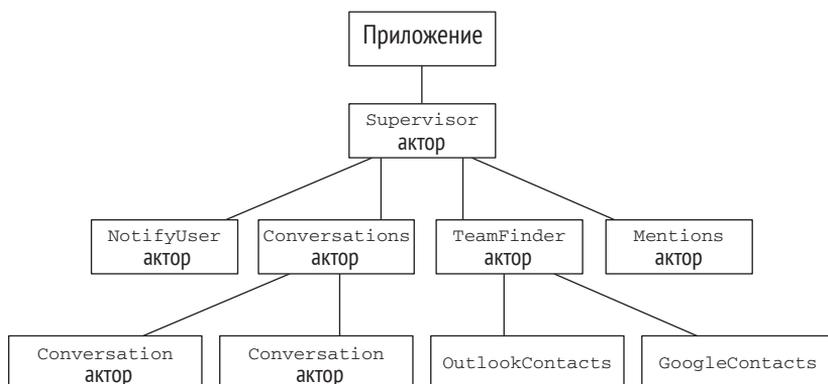


Рис. 1.16. Компоненты

**МОДЕЛЬ АКТОРОВ НЕ НОВА.** Модель акторов не нова и в действительности была разработана довольно давно; сама идея впервые была предложена Карлом Хьюиттом (Carl Hewitt), Питером Бишопом (Peter Bishop) и Ричардом Штайгером (Richard Steiger) еще в 1973 году. Язык Erlang и его библиотеки OTP, разработанные в компании Ericsson в 1986 году, поддерживают модель акторов и использовались для создания масштабируемых систем с жесткими требованиями к доступности. Примером успешного использования языка Erlang является коммутатор AXD 301, который достиг уровня надежности 99.999999%, также известного как *девять девяток*. Реализация модели акторов в Akka имеет пару отличий от реализации в Erlang, но в общем и целом разрабатывалась под влиянием Erlang и унаследовала много идей из этого языка.

### ОТПРАВКА

Актеры могут взаимодействовать с другими актерами, только посылая сообщения. Это поднимает инкапсуляцию на новый уровень. В объектах можно определить, какие методы могут вызываться и какие данные будут доступны извне. Актеры не допускают ни малейшей возможности доступа к их внутреннему состоянию, например к списку сообщений в диалоге. Актеры не могут иметь общего состояния; они не могут, например, ссы-

латься на общий список сообщений в диалоге и параллельно изменять содержимое диалога.

Актор `Conversation` не может просто так вызвать метод любого другого актора, поскольку это может привести к появлению общего изменяемого состояния. Он должен послать сообщение. Отправка сообщения всегда выполняется асинхронно, в стиле *послал и забыл*. Если потребуется точно знать, что актор-адресат получил сообщение, тогда этот актор просто должен послать в ответ некоторое подтверждающее сообщение.

Актор `Conversation` не должен ждать и наблюдать, что происходит с сообщением, отправленным актору `Mentions`; он может просто послать сообщение и продолжить работу, как ни в чем не бывало. Асинхронный обмен сообщениями помогает разорвать тесную связь между компонентами в приложении чата. Это еще одна причина, почему мы использовали очередь сообщений для взаимодействий с объектом `Mentions`, которая теперь больше не нужна.

Сообщения должны быть неизменяемыми, в том смысле, что они не должны допускать возможность изменения после создания. Это делает невозможным изменение одного сообщения двумя акторами по ошибке, что могло бы привести к непредсказуемым результатам.

**ЧТО, КОНТРОЛЬ ТИПОВ НЕ ПОДДЕРЖИВАЕТСЯ?** Актеры могут принимать любые сообщения, и при желании актору можно послать любое сообщение (актор может просто отказаться от обработки некорректного сообщения). Фактически это означает, что контроль типов посылаемых и принимаемых сообщений весьма ограничен. Это может стать неожиданностью для вас, потому что язык `Scala` относится к языкам со статической типизацией и контроль типов на этапе компиляции дает множество преимуществ. Эта гибкость одновременно является и достоинством (как при использовании статических типов организовать обмен данными между разными системами?), и недостатком (меньше информации о типах акторов во время выполнения). Но последнее слово еще не сказано, и разработчики Akka ищут способы определения более безопасных версий акторов, которые могут быть реализованными уже в следующей версии Akka.

Итак, что мы должны сделать, когда пользователь пожелает отредактировать свое сообщение в диалоге `Conversation`? Мы могли бы послать актору диалога сообщение `EditMessage`, содержащее исправленную версию текстового сообщения пользователя, вместо прямого исправления сообщения в общем списке. Актор `Conversation`, получив сообщение `EditMessage`, мог бы заменить существующее сообщение его новой версией.

Неизменяемость становится абсолютной необходимостью, когда дело доходит до конкурентного выполнения, и является еще одним ограниче-

нием, упрощающим жизнь, потому что в механизме остается меньше движущихся частей, которыми требуется управлять.

Порядок следования сообщений в посылающем и принимающем акторах сохраняется. Актор всегда получает сообщения по одному. Представьте, что пользователь отредактировал свое сообщение несколько раз; вполне очевидно, что в конечном итоге пользователь ожидает увидеть результаты последней правки. Порядок сообщений гарантируется только для одного посылающего актора. То есть если одно сообщение в диалоге будут править несколько пользователей, окончательный результат может зависеть от того, как сообщения чередуются во времени.

## СОЗДАНИЕ

Актор может создавать других акторов. На рис. 1.17 показано, как актор Supervisor создает актор Conversations. Как видите, это приводит к автоматическому созданию целой иерархии акторов. Приложение чата сначала создает актор Supervisor, который, в свою очередь, создает всех других акторов в приложении. Актор Conversations восстанавливает все диалоги из журнала, создавая актор Conversation для каждого найденного диалога, который, в свою очередь, восстанавливает свое содержимое из того же журнала.

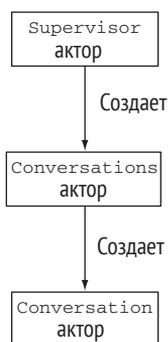


Рис. 1.17.  
Создание

## ПЕРЕХОД

Конечные автоматы – великолепный способ гарантировать, что система будет выполнять только определенные действия, находясь в определенном состоянии.

Актеры принимают сообщения по одному, что является очень удобным свойством для реализации конечных автоматов. Актор может изменять способ обработки входящих сообщений, меняя свое поведение.

Представьте, что пользователи решили закончить диалог. Диалог Conversation запускается в состоянии «открыт» и переходит в состояние «закрыт», получив сообщение CloseConversation. Любое сообщение, посланное диалогу Conversation, находящемуся в состоянии «закрыт», будет проигнорировано. Актор Conversation меняет свое поведение добавления поступающих сообщений на поведение игнорирования всех сообщений.

## УПРАВЛЕНИЕ

Актор должен управлять другими актерами, которые он создал. Супервизор Supervisor в приложении чата может следить за происходящим в главных компонентах, как показано на рис. 1.18.

Супервизор Supervisor решает, что должно произойти, когда какой-то компонент терпит неудачу. Он может, например, решить продолжить ра-

боту приложения после отказа компонента Mentions и актора Notify, потому что они не являются критически важными для системы. Супервизор Supervisor получает специальные сообщения, указывающие на факт ошибки в работе актора и ее причины. Он может решить перезапустить актора или остановить его.

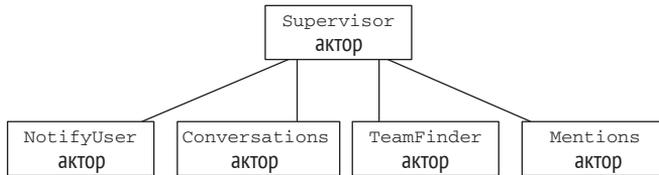


Рис. 1.18. Управление

Роль супервизора может выполнять любой актор, но только в отношении акторов, которых он создал сам. На рис. 1.19 показано, что актор TeamFinder управляет двумя акторами-контактами. В данном случае он мог бы остановить актора OutlookContacts из-за слишком частых ошибок. После этого TeamFinder мог бы продолжить поиск контактов, но только с использованием службы Google.

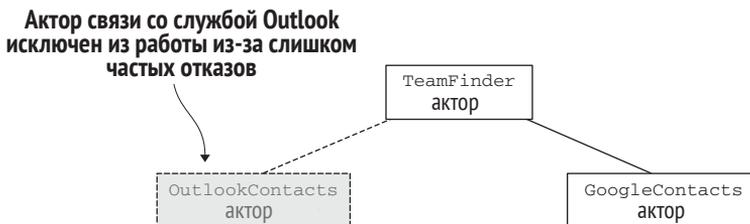


Рис. 1.19. TeamFinder управляет акторами-контактами

Теперь, после знакомства с базовыми операциями акторов, посмотрим, как фреймворк Akka поддерживает акторов и что требуется, чтобы они начали обрабатывать сообщения.

## 1.7. Акторы Akka

До сих пор мы обсуждали модель программирования акторов с концептуальной точки зрения и объясняли причины, почему ее следует использовать. А теперь посмотрим, как фактически фреймворк Akka реализует модель акторов, и на место, где шина встречается с дорогой. Мы рассмотрим, как все детали соединяются в единый механизм и какие компоненты за что отвечают. В следующем разделе мы познакомимся с деталями создания акторов.

### Актеры: независимость по трем осям

На акторы можно взглянуть с другой стороны – как на объекты, независимые друг от друга по трем осям, что обеспечивает широкие возможности масштабирования:

- местоположение;
- время;
- интерфейс.

Независимость по этим трем осям играет важную роль, потому что такая гибкость жизненно необходима для масштабирования. Акторы могут действовать одновременно, если в системе имеется достаточное количество процессоров, или друг за другом. Акторы могут находиться по соседству или далеко друг от друга, и в случае ошибки акторы могут получать сообщения, которые не в состоянии обработать.

- *Местоположение* – актер не дает никаких гарантий и не имеет никаких ожиданий относительно своего местоположения.
- *Время* – актер не дает никаких гарантий и не имеет никаких ожиданий относительно момента, когда завершит свою работу.
- *Интерфейс* – актер не определяет интерфейса. Актер не имеет никаких ожиданий о том, смогут ли понять его сообщения другие акторы. Никакие два актора не имеют общих данных; акторы никогда не ссылаются на и не используют общие данные, которые могут изменяться на месте. Все данные передаются исключительно в сообщениях.

Зависимость компонентов от местоположения, времени и интерфейса – самое большое препятствие для создания приложений, способных восстанавливаться после отказа и масштабироваться по мере необходимости. Система, сконструированная из компонентов, связанных друг с другом по всем трем осям, может существовать только в единой среде выполнения и будет полностью выходить из строя с выходом из строя любого ее компонента.

#### 1.7.1 ActorSystem

Первым делом посмотрим, как создаются акторы. Одни акторы могут создавать другие акторы, но как создается самый первый актер? Взгляните на рис. 1.20.

Первый актер в приложении чата – супервизор Supervisor. Все акторы, изображенные на рис. 1.20, являются частью одного приложения. Как

сделать акторы деталями одного большого механизма, единого целого? Для этого во фреймворке Akka имеется класс `ActorSystem`. Первым делом любое приложение на основе Akka создает систему акторов – экземпляр `ActorSystem`. Этот экземпляр может создавать так называемые акторы верхнего уровня, и обычно с его помощью создается единственный актор верхнего уровня, конструирующий все остальные акторы в приложении. В нашем случае таким единственным актором является супервизор `Supervisor`, осуществляющий мониторинг всех других акторов.

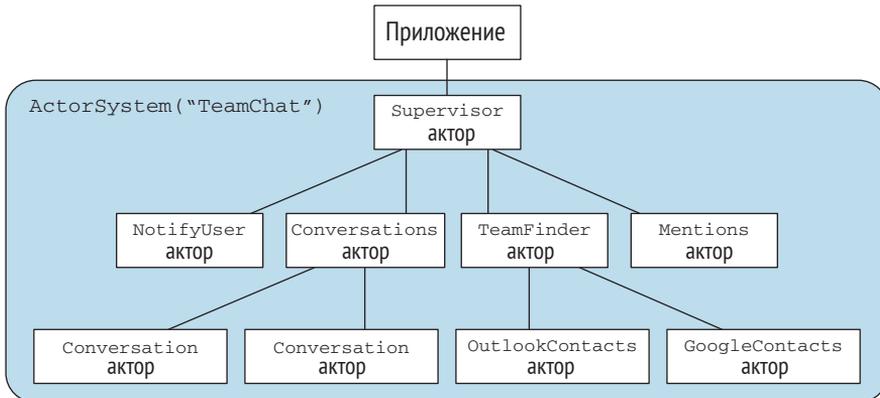


Рис. 1.20. TeamChatActorSystem

Мы уже упоминали, что акторы должны поддерживать такие возможности, как удаленные взаимодействия и сохранение информации в журнале. Объект `ActorSystem` как раз является связующим звеном, обеспечивающим такую поддержку. Большинство возможностей поддерживается *расширениями Akka* – модулями, которые можно подключать к `ActorSystem`. Простым примером может служить поддержка планировщика, периодически посылающего сообщения акторам.

`ActorSystem` возвращает адрес созданного актора верхнего уровня, а не сам актор. Этот адрес называется ссылкой на актор и имеет тип `ActorRef`. Ссылку `ActorRef` можно использовать для отправки сообщений актору. В этом есть определенный смысл, особенно если вспомнить, что актор может находиться на другом сервере.

Иногда возникает необходимость найти актор в системе. Для этого можно воспользоваться объектом `ActorPath`. Иерархию акторов можно сравнить со структурой путей URL. Каждый актор имеет имя. Это имя должно быть уникальным на данном уровне иерархии: два соседних актора не могут иметь одинаковые имена (если вы не укажете имя сами, Akka сгенерирует его автоматически, но вообще всегда лучше самим давать имена акторам). Ссылки на акторы можно получать по путям к ним, абсолютным или относительным.

## 1.7.2. ActorRef, почтовый ящик и актор

Сообщения акторам посылаются по ссылкам ActorRef. Каждый актор имеет почтовый ящик – он действует подобно очереди. Сообщения, которые посылаются по ссылке ActorRef, временно сохраняются в почтовом ящике для последующей обработки, по очереди, в порядке их поступления. На рис. 1.21 показаны отношения между ссылкой ActorRef, почтовым ящиком и актором.

Как именно актор обрабатывает сообщения, описывается в следующем разделе.

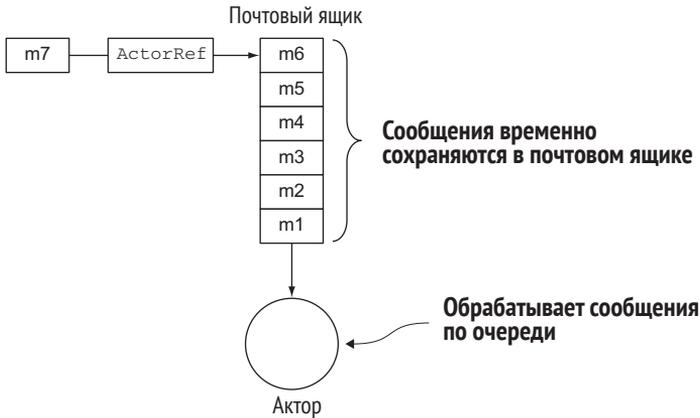


Рис. 1.21. ActorRef, почтовый ящик и актор

## 1.7.3. Диспетчеры

Актеры вызываются диспетчером в некоторый момент. Диспетчер вталкивает сообщения в почтовый ящик актора, как показано на рис. 1.22.

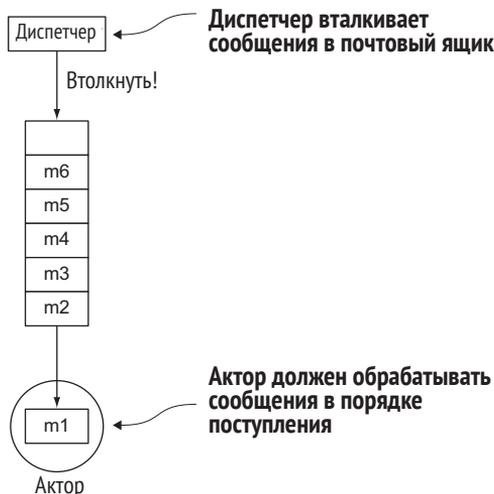
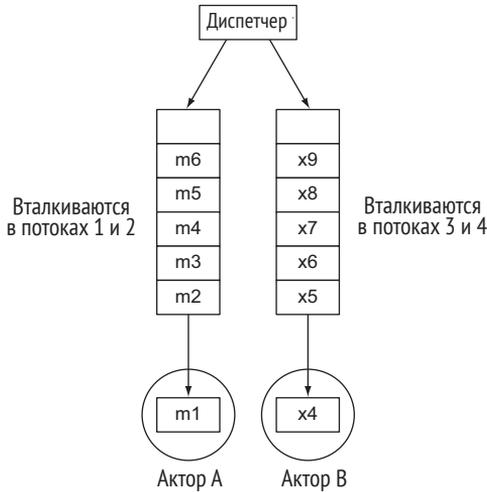


Рис. 1.22. Диспетчер помещает сообщения в почтовый ящик

Тип диспетчера определяется моделью потоков, которая используется для передачи сообщений. Многие акторы могут получать сообщения, поступающие из разных потоков, как показано на рис. 1.23.



**Рис. 1.23.** Диспетчер передает сообщения нескольким акторам

На рис. 1.23 показано, как сообщения от  $m1$  до  $m6$  передаются диспетчером в потоках 1 и 2 и сообщения от  $x4$  до  $x9$  – в потоках 3 и 4. Этот рисунок не означает, что вы можете или должны контролировать потоки, через которые передаются сообщения. Здесь самое важное, что вы можете до некоторой степени настраивать многопоточную модель. Все виды диспетчеров поддерживают некоторую возможность настройки, и вы можете добавить диспетчера в актор, в группу акторов или во всю систему акторов.

То есть, отправляя сообщения актору, в действительности вы передаете их диспетчеру, который рано или поздно поместит сообщение в почтовый ящик актора. Актор, в свою очередь, может оставить сообщение для следующего актора, которое также будет доставлено в некоторый момент.

Акторы оказываются легковесными потому, что действуют поверх диспетчеров; число акторов не обязательно должно быть кратным числу потоков. Акторы Akka занимают намного меньше памяти, чем потоки, например в 1 Гбайт памяти может поместиться порядка 2.7 миллиона акторов. Это намного больше, чем 4096 потоков на 1 Гбайт памяти, то есть у вас больше свободы в создании акторов разных типов, чем в случае с потоками.

На выбор имеется несколько типов диспетчеров, настроенных для конкретных нужд. Возможность настройки диспетчеров и почтовых ящиков открывает широкие возможности управления производительностью. В главе 15 мы дадим несколько простых советов по настройке производительности.

**АД ОБРАТНЫХ ВЫЗОВОВ.** Многие фреймворки реализуют асинхронное программирование посредством механизма обратных вызовов. Если вам довелось использовать такие фреймворки, весьма вероятно, вы сталкивались с ситуацией, которая получила название *ад обратных вызовов*, когда каждый обратный вызов содержит другой обратный вызов, который содержит третий обратный вызов, и т. д.

Сравните это с работой диспетчеров, которые рассылают сообщения по почтовым ящикам в заданном потоке. Актoram не нужно выполнять обратные вызовы внутри обратных вызовов до падения в серную яму. Они могут просто передавать сообщения диспетчеру, который позаботится обо всем остальном.

### 1.7.4. Акторы и сеть

Как акторы Akka взаимодействуют друг с другом посредством сети? Ссылки `ActorRef`, по сути, являются адресами акторов, то есть все, что нужно знать для организации взаимодействий, – это как адреса связаны с акторами. Если инструмент различает локальные и удаленные адреса, вы сможете масштабировать решение, просто настроив разрешение адресов.

В Akka имеется модуль поддержки удаленных взаимодействий (который мы обсудим в главе 6), обеспечивающий желаемую прозрачность. Akka передает сообщения удаленному актору, действующему на удаленной машине, и получает результаты по сети.

Единственное, что меняется при этом, – это порядок разрешения ссылок на удаленные акторы, что достигается простой настройкой, как вы увидите позже. Сам код при этом не изменяется, а это значит, что для перехода от вертикального масштабирования к горизонтальному не требуется менять код.

Гибкость разрешения адресов широко используется в Akka, что мы не раз продемонстрируем в этой книге. Эта гибкость используется удаленными акторами, средствами кластеризации и даже инструментами тестирования.

## 1.8. В заключение

Давайте кратко повторим, что вы узнали в этой главе. Масштабирование традиционно сопряжено с большими сложностями. Сложность, возникающая при масштабировании, быстро выходит из-под контроля. Акторы Akka используют важные архитектурные решения, обеспечивающие значительную гибкость при масштабировании.

Акторы – это модель программирования, поддерживающая масштабирование по вертикали и горизонтали, в которой все крутится вокруг

отправки и приема сообщений. Даже притом, что это не панацея от всех проблем, возможность использовать единую модель программирования снижает сложность масштабирования.

Уникальность фреймворка Akka заключается в простоте конструирования приложений на основе акторов, благодаря чему вы можете сосредоточиться на решении прикладных задач.

На данном этапе у вас должно сложиться понимание, что акторы способны дать вам больше гибкости при невысоком уровне сложности, что существенно упростит масштабирование. Но вам еще многое предстоит узнать, а как известно, дьявол кроется в деталях.

Но сначала мы создадим простой HTTP-сервер на акторах и развернем его в службе PaaS (Platform as a Service – платформа как служба)!

# Глава 2

## Подготовка и запуск

В этой главе:

- извлечение шаблона проекта;
- создание минимального приложения на Akka для размещения в облаке;
- развертывание в Heroku.

Итак, первое звено – *звено сбора данных*, являющееся точкой входа в потоковую систему. На рис. 2.1 показана немного модифицированная архитектурная диаграмма с упором на звено сбора данных.

В этой главе наша цель состоит в том, чтобы показать, как быстро можно создать приложение на основе Akka, которое не только делает что-то нетривиальное, но способно к масштабированию даже в простейшей, начальной реализации. Мы клонируем проект из репозитория [github.com](https://github.com), содержащий пример, и рассмотрим все, что необходимо знать, чтобы начать создавать приложения на основе Akka. Сначала мы рассмотрим зависимости, которые необходимо удовлетворить для минимального приложения, и с помощью инструмента сборки *Lightbend Simple Build Tool (sbt)* создадим единственный JAR-файл, с помощью которого можно запустить приложение. Мы создадим минимальное приложение по продаже билетов и в первой итерации реализуем минимальный набор REST-служб. Мы постараемся максимально упростить его, используя базовые возможности Akka. Наконец, мы покажем, как легко развернуть и запустить это приложение в облаке Heroku. Но самое примечательное – мы очень быстро достигнем желаемого результата!

Одна из самых замечательных особенностей фреймворка Akka заключена в простоте настройки и запуска, а также в его гибкости, в чем вы вскоре убедитесь. Мы оставим в стороне некоторые детали, касающиеся инфраструктуры, и подробнее рассмотрим поддержку HTTP в Akka только в главе 12, и тем не менее в этой главе мы дадим достаточный объем инфор-

мации, чтобы вы могли конструировать REST-интерфейсы любых видов. В следующей главе вы увидите, как все это можно объединить с принципом разработки через тестирование (Test-Driven Development, TDD).

## 2.1. Клонирование, сборка и интерфейс тестирования

Для простоты мы сохранили исходный код приложения в репозитории `github.com`, а также все остальные примеры кода для этой книги. Первое, что вы должны сделать, – клонировать репозиторий в каталог на своем компьютере.

### Листинг 2.1. Клонирование проекта примера

```
git clone https://github.com/RayRoestenburg/akka-in-action.git
```

← Клонирование репозитория Git с полным работающим кодом примера

Эта команда создаст каталог *akka-in-action*, содержащий подкаталог *chapter-up-and-running*, в котором хранится проект примера для данной главы. Мы полагаем, что вы уже знакомы с Git и GitHub. В этой главе мы будем использовать следующие инструменты: `sbt`, Git, Heroku и `httpie` (простой в использовании HTTP-клиент командной строки).

**ПРИМЕЧАНИЕ.** Обратите внимание, что Akka 2.4 требует наличия версии Java 8. Если у вас установлена более ранняя версия `sbt`, обновите ее до версии не ниже 0.13.7. Также нелишним будет установить утилиту `sbt-extras` Пола Филлипса (Paul Phillips, <https://github.com/paulp/sbt-extras>), которая автоматически выясняет, какая версия `sbt` и Scala используется.

Рассмотрим структуру проекта. Проекты `sbt` имеют структуру, напоминающую структуру проектов Maven. Основное отличие в том, что `sbt` позволяет использовать в файлах сборки код на языке Scala. Это делает данный инструмент намного мощнее. Желаящим больше узнать о `sbt` мы рекомендуем книгу Джошуа Суэрет (Joshua Suereth) и Мэттью Фаруэлла (Matthew Farwell) «SBT in Action», выпущенную издательством Manning Publications ([www.manning.com/suereth2/](http://www.manning.com/suereth2/)). Внутри каталога *chapter-up-and-running* весь код для сервера находится в подкаталоге *src/main/scala*; конфигурационные файлы и другие ресурсы – в подкаталоге *src/main/resources*; а тесты – в подкаталоге *src/test/scala*. Проект должен собираться без каких-либо проблем. Запустите следующую команду в каталоге *chapter-up-and-running*:

`sbt assembly` ← Скомпилирует и упакует код в единый файл JAR

Вы должны увидеть, как `sbt` загрузит все необходимые зависимости, выполнит все тесты и, наконец, соберет один большой файл JAR `target/scala-2.11/goticks-assembly-1.0.jar`. После этого вы сможете запустить сервер простой командой, представленной в листинге 2.2.

### Листинг 2.2. Запуск файла JAR

```
java -jar target/scala-2.11/goticks-assembly-1.0.jar
RestApi bound to /0:0:0:0:0:0:0:5000
```

← Приложение запускается как любой другой код на Java

← Вывод в консоль: http-сервер запущен и ожидает соединений на порте с номером 5000

Теперь, убедившись, что проект собирается без ошибок, поговорим о происходящем внутри. В следующем разделе мы начнем с файлов сборки, а затем исследуем ресурсы и фактический код служб.

#### 2.1.1. Сборка с помощью sbt

Рассмотрим сначала файл сборки. В этой главе мы будем использовать простой язык SBT DSL (Domain-Specific Language – предметно-ориентированный язык) файлов сборки, потому что он дает нам все, что необходимо. По мере движения вперед мы будем добавлять все больше зависимостей, но, как вы увидите, в своих будущих проектах вы сможете обойтись без помощи шаблонов или копирования больших фрагментов из файлов сборки других проектов. Если прежде вам не приходилось использовать SBT DSL, обратите внимание, что очень важно оставлять пустые строки между строками с настройками (которые не нужны в режиме полной конфигурации и когда можно писать код на Scala как обычно). Файл сборки находится непосредственно в каталоге *chapter-up-and-running*, под именем *build.sbt*.

### Листинг 2.3. Файл сборки sbt

```
enablePlugins(JavaServerAppPackaging)
name := "goticks"
version := "1.0"
organization := "com.goticks"
libraryDependencies ++= {
  val akkaVersion = "2.4.9"
  Seq(
    "com.typesafe.akka" %% "akka-actor" % akkaVersion,
```

← Требуется для развертывания в Heroku (далее)

← Информация о приложении

← Сообщить sbt об удаленных репозиториях, откуда можно загрузить зависимости

← Используемая версия Akka

← Модуль, требуемый актерам Akka (прежде Lightbend носила название Typesafe; отсюда такие имена пакетов)

```

"com.typesafe.akka" %% "akka-http-core" % akkaVersion,
"com.typesafe.akka" %% "akka-http-experimental" % akkaVersion,
"com.typesafe.akka" %% "akka-http-spray-json-experimental" %
  akkaVersion,
"io.spray"          %% "spray-json"      % "1.3.1",
"com.typesafe.akka" %% "akka-slf4j"     % akkaVersion,
"ch.qos.logback"   % "logback-classic" % "1.1.3",
"com.typesafe.akka" %% "akka-testkit"  % akkaVersion % "test",
"org.scalatest"    %% "scalatest"      % "2.2.0" % "test"
)
}

```

Для тех, кому интересно знать, откуда загружаются библиотеки, заметим, что `sbt` использует несколько предопределенных репозиториев, включая репозиторий `Lightbend`, через который распространяются библиотеки `Akka`, используемые здесь. Для пользователей `Maven` этот файл сборки выглядит очень компактно. Так же как в `Maven`, описав репозитории и зависимости, мы легко сможем получать новейшие версии, просто изменив одно значение.

Каждая зависимость ссылается на артефакт `Maven` в формате *организация % модуль % версия* (пара символов `%%` определяет автоматический выбор правильной версии `Scala` библиотеки). Наиболее важной зависимостью здесь является модуль `akka-actor`. Теперь, подготовив файл сборки, мы можем скомпилировать код, выполнить тесты и собрать файл `JAR`. Запустите следующую команду в каталоге *chapter-up-and-running*.

#### Листинг 2.4. Запуск тестов

```
sbt clean compile test
```

← Удалить целевые файлы; затем скомпилировать и выполнить тесты

Если какие-нибудь зависимости понадобится загрузить, `sbt` сделает это автоматически. Теперь давайте поближе рассмотрим цель, к которой мы идем.

### 2.1.2. Забегая вперед: REST-сервер `GoTicks.com`

Наша служба по продаже билетов позволит клиентам приобретать билеты на любые события, концерты, спортивные соревнования и т. д. Допустим, что мы являемся частью стартапа под названием `GoTicks.com` и в этой первой итерации нам дали задание создать `REST`-сервер, реализующий первую версию службы. Непосредственно сейчас нам нужно, чтобы клиенты получали пронумерованные билеты. После продажи последнего билета на событие сервер должен отвечать `HTTP`-кодом `404 (Not Found)`. Первое, что мы реализуем в `REST API`, – добавление нового события (поскольку все остальные службы требуют наличия события в системе). Новое

событие содержит только название, например «RHCP» (Red Hot Chili Peppers)<sup>1</sup>, и общее число билетов, которое можно продать.

Требования к RestApi перечислены в табл. 2.1.

**Таблица 2.1.** REST API

Описание	Метод HTTP	URL	Тело запроса	Код состояния	Пример ответа
Создать событие	POST	/events/RHCP	{"tickets" : 250}	201 Created	{ "name": "RHCP", "tickets": 250 }
Получить все события	GET	/events	—	200 OK	[{ event : "RHCP", tickets : 249 }, { event : "Radiohead", tickets : 130 }]
Купить билет	POST	/events/RHCP/tickets	{"tickets" : 2 }	201 Created	{ "event" : "RHCP", "entries" : [ { "id" : 1 }, { "id" : 2 } ] }
Отменить событие	DELETE	/events/RHCP	—	200 OK	{ event : "RHCP", tickets : 249 }

Давайте соберем приложение и запустим его под управлением sbt. Перейдите в каталог *chapter-up-and-running* и запустите следующую команду.

**Листинг 2.5.** Запуск приложения локально, под управлением sbt

```
sbt run ← Скомпилировать и запустить приложение
```

```
[info] Running com.goticks.Main
INFO [Slf4jLogger]: Slf4jLogger started
RestApi bound to /0:0:0:0:0:0:0:5000
```

<sup>1</sup> «Красные острые перцы чили» – американская рок-группа, образованная в 1983 году. – Прим. перев.

Подобно большинству инструментов сборки, `sbt` напоминает утилиту `make`: если код нужно скомпилировать, он будет скомпилирован; затем упакован и т. д. В отличие от множества инструментов сборки, `sbt` может также развертывать и запускать приложения локально. Если вы получили сообщение об ошибке, убедитесь, что сервер перед этим не был запущен в другой консоли и никакой другой процесс не использует сетевой порт с номером 5000. Давайте посмотрим, все ли работает, воспользовавшись утилитой командной строки `httpie`<sup>2</sup> для работы с протоколом HTTP. Эта утилита поддерживает формат JSON и автоматически обрабатывает все необходимые заголовки. Сначала посмотрим, можем ли мы создать новое событие с некоторым числом билетов.

### Листинг 2.6. Создание события из командной строки

```
http POST localhost:5000/events/RHCP tickets:=10
HTTP/1.1 201 Created
Connection: keep-alive
Content-Length: 76
Content-Type: text/plain; charset=UTF-8
Date: Mon, 20 Apr 2015 12:13:35 GMT
Proxy-Connection: keep-alive
Server: GoTicks.com REST API

{
  "name": "RHCP",
  "tickets": 10
}
```

← Команда `httpie`, посылающая запрос POST нашему серверу с единственным параметром

← Ответ от сервера («201 Created» означает успех)

Параметр преобразуется в тело запроса в формате JSON. Обратите внимание, что в параметре вместо `=` используется `:=`. Это означает, что значение параметра не является строкой. В данном случае параметр транслируется в `{ "tickets" : 10 }`. Весь следующий блок в листинге 2.6 – это полный HTTP-ответ, полученный утилитой `httpie`. Теперь событие создано. Давайте создадим еще одно:

```
http POST localhost:5000/events/DjMadlib tickets:=15
```

Теперь попробуем выполнить запрос GET. В соответствии с соглашениями REST в ответ на запрос GET, адрес URL которого оканчивается типом сущности, должен вернуть количество экземпляров этой сущности.

### Листинг 2.7. Запрос списка всех событий

```
http GET localhost:5000/events ← Запросить список всех событий
```

<sup>2</sup> Загрузить утилиту `httpie` можно отсюда: <https://github.com/jakubroztocil/httpie>.

```

...
HTTP/1.1 200 OK
Connection: keep-alive
Content-Length: 110
Content-Type: application/json; charset=UTF-8
Date: Mon, 20 Apr 2015 12:18:01 GMT
Proxy-Connection: keep-alive
Server: GoTicks.com REST API

```

← Полный ответ HTTP-сервера  
(200 означает успех)

```

{
  "events": [
    {
      "name": "DjMadlib",
      "tickets": 15
    },
    {
      "name": "RHCP",
      "tickets": 10
    }
  ]
}

```

Обратите внимание, что в ответ сервер вернул все события и количество непроданных билетов. Теперь посмотрим, можно ли купить билет на концерт RHCP.

### Листинг 2.8. Покупка билета на концерт RHCP

```

http POST localhost:5000/events/RHCP/tickets tickets:=2

```

← Команда `httpie`, посылающая запрос POST нашему серверу с единственным параметром

```

HTTP/1.1 201 Created
Connection: keep-alive
Content-Length: 74
Content-Type: application/json; charset=UTF-8
Date: Mon, 20 Apr 2015 12:19:41 GMT
Proxy-Connection: keep-alive
Server: GoTicks.com REST API

```

← Ответ от сервера  
(«201 Created» означает успех)

```

{
  "entries": [
    {
      "id": 1
    },
    {
      "id": 2
    }
  ],
}

```

← Приобретенные нами билеты  
в формате JSON

```
"event": "RHCP"
}
```

В данном случае нераскупленных билетов на данное событие было больше двух; иначе мы получили бы ответ с кодом 404.

Если теперь снова послать запрос GET по адресу URL/events, сервер вернет ответ, как показано в листинге 2.9.

### Листинг 2.9. Ответ на запрос GET после создания двух билетов

```
HTTP/1.1 200 OK
Content-Length: 91
Content-Type: application/json; charset=UTF-8
Date: Mon, 20 Apr 2015 12:19:42 GMT
Server: GoTicks.com REST API
```

```
[
  {
    "event": "DjMadlib",
    "nrOfTickets": 15
  },
  {
    "event": "RHCP",
    "nrOfTickets": 8
  }
]
```

Как и ожидалось, на концерт RHCP осталось 8 непроданных билетов. После покупки всех билетов мы должны получить ответ с кодом 404.

### Листинг 2.10. Результат после продажи всех билетов

```
HTTP/1.1 404 Not Found
Content-Length: 83
Content-Type: text/plain
Date: Tue, 16 Apr 2013 12:42:57 GMT
Server: GoTicks.com REST API
```

← После продажи всех билетов  
сервер ответил кодом 404

```
The requested resource could not be found
but may be available again in the future.
```

На этом мы завершаем исследование всех вызовов, реализованных в REST API. Очевидно, что на данный момент приложение поддерживает базовый набор операций CRUD (Create, Read, Update, Delete – создать, прочитать, изменить, удалить), от создания события до продажи всех билетов. Его нельзя назвать исчерпывающим; например, мы не учитываем события, на которые проданы не все билеты, но которые уже начались и, соответственно, продажа билетов на них должна быть прекращена. В следую-

щем разделе мы посмотрим, как получить результат, который мы только что рассмотрели.

## 2.2. Исследование акторов в приложении

В этом разделе мы рассмотрим внутреннее устройство приложения. Вы можете по ходу объяснений создавать акторы у себя или просто просматривать исходный код, доступный в репозитории `github.com`. Как вы уже знаете, акторы могут выполнять четыре операции; создание, отправку/прием, переход и осуществлять диспетчеризацию. В этом примере мы коснемся только первых двух операций. Сначала рассмотрим общую структуру: как будут выполняться операции разными акторами для поддержки базовых возможностей – создавать события, продавать билеты и завершать события.

### 2.2.1. Структура приложения

Приложение состоит из двух классов акторов. Прежде всего создадим систему акторов, в которой будут находиться все акторы. После этого акторы смогут создавать друг друга. Данная последовательность изображена на рис. 2.1.

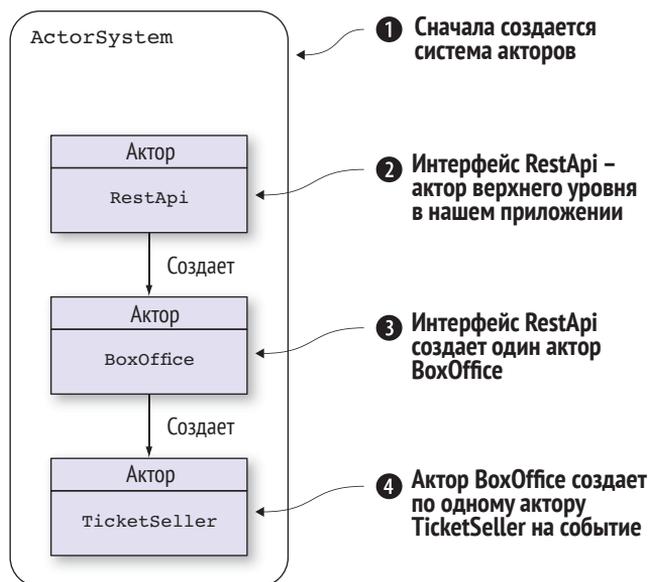


Рис. 2.1. Последовательность создания акторов в ответ на запрос REST

`RestApi` содержит несколько *маршрутов* для обработки HTTP-запросов. Маршруты определяют, как должны обрабатываться HTTP-запросы с применением удобного предметно-ориентированного языка (DSL), поддер-

живаемого модулем akka-http. Мы обсудим маршруты в разделе 2.2.4. По своей сути RestApi – это адаптер для поддержки HTTP: он берет на себя все хлопоты по преобразованию данных в/из формата JSON и производит требуемый HTTP-ответ. Позднее мы посмотрим, как этот актер подключается к HTTP-серверу. Как видите, даже в этом простом примере для обработки запросов порождается несколько акторов, каждый из которых имеет свои конкретные обязанности. Актор TicketSeller следит за продажей билетов на одно конкретное событие. На рис. 2.2 показано, как запрос на создание события пересекает систему акторов (это первая служба в табл. 2.1).

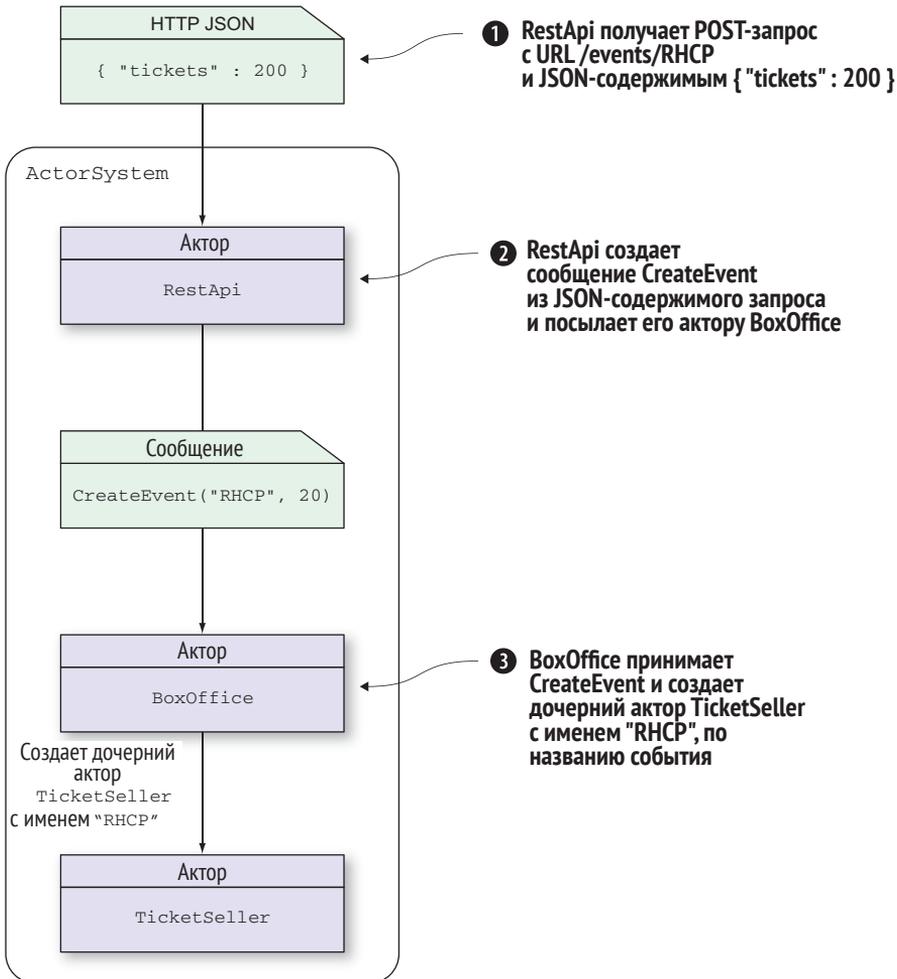


Рис. 2.2. Создание события из содержимого запроса

Вторая служба, которую мы обсудили, позволяет клиенту купить билет (на предварительно созданное событие). На рис. 2.3 показано, что происходит в ответ на запрос покупки билета.

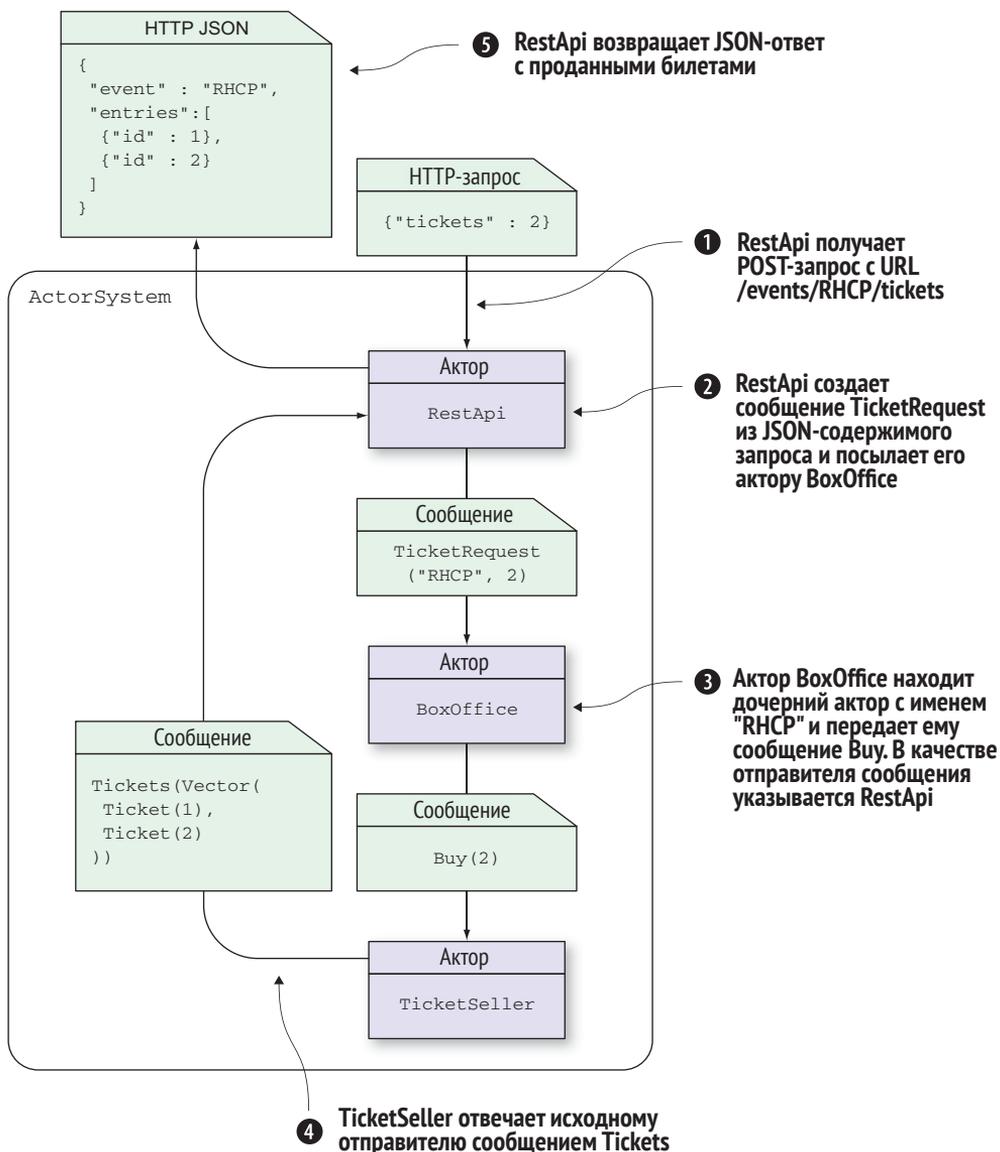


Рис. 2.3. Покупка билета

Отступим на шаг назад и окинем код одним взглядом. Обратите внимание сначала на класс `Main`, который запускает приложение. Объект `Main` – это самое обычное приложение на Scala, которое запускается точно так же, как любое другое приложение на этом языке. Он напоминает класс на Java с методом `main`. Прежде чем погрузиться в код класса `Main`, посмотрим сначала на наиболее важные инструкции – инструкции импорта, которые приводятся в листинге 2.11.

**Листинг 2.11.** Инструкции import для класса Main

```

import akka.actor.{ ActorSystem , Actor, Props }
import akka.event.Logging
import akka.util.Timeout

import akka.http.scaladsl.Http

import akka.http.scaladsl.Http.ServerBinding
import akka.http.scaladsl.server.Directives._
import akka.stream.ActorMaterializer

import com.typesafe.config.{ Config, ConfigFactory }

```

← Код с реализацией акторов находится в пакете akka.actor  
 ← Расширение журналирования  
 ← Для работы с запросами требуется поддержка тайм-аута  
 ← Поддержка протокола HTTP находится в пакете akka.http  
 ← Импорт библиотеки для конфигурирования

В первую очередь класс Main создает ActorSystem, затем RestApi, получает расширение HTTP и связывает с ним маршруты RestApi. Как это делается, будет показано немного позже. Для получения доступа ко многим инструментам поддержки Akka использует так называемые *расширения*, как вы увидите далее в книге; Http и Logging – первые примеры таких расширений.

Никогда не следует жестко «зашивать» в код такие конфигурационные параметры, как имя хоста или номер сетевого порта, поэтому для поддержки гибкой настройки мы используем библиотеку Typesafe Config (подробнее об использовании этой библиотеки рассказывается в главе 7).

В листинге 2.12 показаны основные инструкции, запускающие ActorSystem с расширением http и получающие маршруты RestApi для привязки к HTTP.

**Листинг 2.12.** Запуск HTTP-сервера

```

object Main extends App
  with RequestTimeout {
  val config = ConfigFactory.load()
  val host = config.getString("http.host")
  val port = config.getInt("http.port")

  implicit val system = ActorSystem()
  implicit val ec = system.dispatcher

  val api = new RestApi(system, requestTimeout(config)).routes

  implicit val materializer = ActorMaterializer()
  val bindingFuture: Future[ServerBinding] =
    Http().bindAndHandle(api, host, port)
}

```

← Получить имя хоста и номер порта из конфигурации  
 ← bindAndHandle выполняется асинхронно и требует неявного контекста выполнения ExecutionContext  
 ← RestApi предоставляет поддержку маршрутов HTTP  
 ← Запуск HTTP-сервера с маршрутами RestAPI

Объект `Main` наследует `App`, как любое другое приложение на `Scala`.

Система акторов `ActorSystem` активизируется сразу после создания и запускает пул потоков выполнения.

Метод `Http()` возвращает расширение `HTTP`. Метод `bindAndHandle()` связывает маршруты, объявленные в `RestApi`, с `HTTP`-сервером. Метод `bindAndHandle()` выполняется асинхронно и возвращает объект `Future`. Мы не будем углубляться в эти подробности сейчас и вернемся к ним позже, в главе 5. Метод `Main` не завершается немедленно – `ActorSystem` создает нефоновые (`non-daemon`) потоки выполнения и продолжает выполняться (пока они не завершатся).

В листинге 2.13 для полноты картины показано определение трейта `RequestTimeout`, который позволяет `RestApi` использовать настраиваемые тайм-ауты для запросов в `akka-http`.

**Листинг 2.13.** Трейт `RequestTimeout`

```
trait RequestTimeout {
  import scala.concurrent.duration._
  def requestTimeout(config: Config): Timeout = {
    val t = config.getString("spray.can.server.request-timeout")
    val d = Duration(t)
    FiniteDuration(d.length, d.unit)
  }
}
```

Использует стандартный тайм-аут для запросов из конфигурации сервера `akka-http`

Тайм-аут для запросов извлекается в трейте `RequestTimeout` (не волнуйтесь, если что-то в этом определении вам покажется непонятным; на данном этапе это совершенно не важно).

Сейчас необязательно до конца понимать все детали работы расширения `http`, потому что эти тонкости мы охватим позже.

Акторы в приложении взаимодействуют друг с другом посредством сообщений. Сообщения, которые актор может получать или посылать, объединяются в объекте-компаньоне актора. В листинге 2.14 определяются сообщения, поддерживаемые актором `BoxOffice`.

**Листинг 2.14.** Сообщения для `BoxOffice`

```
case class CreateEvent(name: String, tickets: Int)
case class GetEvent(name: String)
case object GetEvents
case class GetTickets(event: String, tickets: Int)
case class CancelEvent(name: String)
case class Event(name: String, tickets: Int)
```

Сообщение для создания события  
 Сообщение для получения события  
 Сообщение для запроса всех событий  
 Сообщение для получения билетов на событие  
 Сообщение для отмены события  
 Сообщение для описания события



```

case Buy(nrOfTickets) =>
  val entries = tickets.take(nrOfTickets).toVector
  if(entries.size >= nrOfTickets) {
    sender() ! Tickets(event, entries)
    tickets = tickets.drop(nrOfTickets)
  } else sender() ! Tickets(event)
case GetEvent => sender() ! Some(BoxOffice.Event(event, tickets.size))
case Cancel =>
  sender() ! Some(BoxOffice.Event(event, tickets.size))
  self ! PoisonPill
}
}

```

← Извлекает заданное число билетов и списка и отвечает сообщением `Tickets`, содержащим билеты, если имеется достаточное количество непроданных билетов; иначе отвечает пустым сообщением `Tickets`

← В ответ на сообщение `GetEvent` возвращает событие и количество непроданных билетов

Актор `TicketSeller` хранит доступные билеты в неизменяемом списке. Изменяемый список тоже было бы безопасно использовать, потому что он доступен только внутри актора и, соответственно, в каждый момент времени доступен только одному потоку выполнения.

Тем не менее всегда предпочтительнее использовать неизменяемые списки. Вы можете забыть, что он изменяемый, возвращая весь список или его часть другому актору. Например, взгляните на метод `take`, который мы использовали для получения первой пары билетов из списка. Для изменяемого списка (`scala.collection.mutable.ListBuffer`) метод `take` вернет список того же типа (`ListBuffer`), то есть изменяемый.

В следующем разделе мы рассмотрим актор `BoxOffice`.

### 2.2.3. Актор `BoxOffice`

Актор `BoxOffice` должен создать для каждого события дочерний актор `TicketSeller` и делегировать ему продажу билетов на заданное событие. В листинге 2.17 показано, как `BoxOffice` откликается на сообщение `CreateEvent`.

**Листинг 2.17.** Актор `BoxOffice` создает акторы `TicketSellers`

```

def createTicketSeller(name: String) =
  context.actorOf(TicketSeller.props(name), name)
def receive = {
  case CreateEvent(name, tickets) =>
    def create() = {
      val eventTickets = createTicketSeller(name)
      val newTickets = (1 to tickets).map { ticketId =>
        TicketSeller.Ticket(ticketId)
      }.toVector
      eventTickets ! TicketSeller.Add(newTickets)
      sender() ! EventCreated
    }
}

```

← Создает `TicketSeller`, используя контекст, созданный в другом методе, чтобы его легче было переопределить на этапе тестирования

← Локальный метод, который создает актор по продаже билетов, добавляет в него билеты и отвечает сообщением `EventCreated`

```

}
context.child(name).fold(create())(_ => sender() ! EventExists)

```

Создает дочерний актер и отвечает сообщением `EventCreated` или, если событие уже существует, отвечает сообщением `EventExists`

Актер `BoxOffice` создает `TicketSeller` для события, если оно пока отсутствует в системе. Обратите внимание, что для создания акторов вместо системы акторов здесь используется *контекст*; акторы, созданные в контексте другого актора, являются его дочерними актерами и управляются родительским актором (подробнее об этом рассказывается в последующих главах). Актер `BoxOffice` конструирует список нумерованных билетов для события и передает этот список актору `TicketSeller`. Он также отвечает отправителю сообщения `CreateEvent`, что заданное событие создано (в данном случае отправителем является актер `RestApi`). В листинге 2.18 показано, как `BoxOffice` откликается на сообщение `GetTickets`.

#### Листинг 2.18. Покупка билетов

```

case GetTickets(event, tickets) =>
  def notFound() = sender() ! TicketSeller.Tickets(event)
  def buy(child: ActorRef) =
    child.forward(TicketSeller.Buy(tickets))

  context.child(event).fold(notFound())(buy)

```

Отправка пустого сообщения `Tickets`, если запрошенное событие не найдено

Покупка делегируется найденному актору `TicketSeller`

Вызовет `notFound` или `buy` с найденным актором `TicketSeller`

Сообщение `Buy` пересылается актору `TicketSeller`. Такое решение позволяет актору `RestApi` использовать `BoxOffice` как прокси. Ответ от `TicketSeller` возвращается непосредственно `RestApi`.

Следующее сообщение, `GetEvents`, требует более сложной обработки. В данном случае наша задача – опросить все акторы `TicketSeller`, чтобы получить количество непроданных билетов в каждом, и объединить все результаты в список событий. Эта задача особенно интересна тем, что операции выполняются асинхронно, и вместе с тем нам не хотелось бы ждать и препятствовать актору `BoxOffice` обрабатывать другие запросы.

Код в листинге 2.19 использует идею, воплощенную в объектах `Future`, которая подробно будет рассматриваться в главе 5, поэтому если вы захотите пропустить этот пример, можете смело сделать это. Если вы решили разобраться, вам предстоит сложная задача. Взгляните на код!

#### Листинг 2.19. Получение списка событий

```

case GetEvents =>
  import akka.pattern.ask

```

```
import akka.pattern.pipe
```

```
def getEvents = context.children.map { child =>
  self.ask(GetEvent(child.path.name)).mapTo[Option[Event]]
}
```

Определение локального метода  
для опроса всех акторов TicketSeller,  
представляющих события

```
def convertToEvents(f: Future[Iterable[Option[Event]]]) =
  f.map(_._flatten).map(l=> Events(l.toVector))
```

```
pipe(convertToEvents(Future.sequence(getEvents))) to sender()
```

ask возвращает объект Future, который в некоторый момент в будущем получит значение.  
getEvents возвращает Iterable[Future[Option[Event]]]; sequence может превратить это значение  
в Future[Iterable[Option[Event]]]. pipe пошлет значение объекта Future в актор, когда оно будет  
вычислено. В данном случае отправителем сообщения GetEvents является RestApi

Наша цель – опросить все акторы TicketSeller. В ответ на сообщение GetEvent возвращается Option[Event], то есть, отображая все TicketSeller, мы получаем в результате Iterable[Option[Event]]. Этот метод преобразует Iterable[Option[Event]] в Iterable[Event], опуская все пустые результаты Option. Iterable преобразуется в сообщение Events

Прямо сейчас мы не будем углубляться в этот пример, а просто рассмотрим лежащие в его основе идеи. В этом примере метод ask немедленно возвращает объект Future. Этот объект представляет значение, которое будет доступно в некоторый момент в будущем (от англ. *future*, отсюда такое имя объекта). Вместо ожидания результата (события с количеством оставшихся билетов) мы получаем ссылку на объект Future. Мы никогда не читаем значения таких объектов непосредственно, а определяем, что должно произойти, как только это значение станет доступно. Мы можем даже объединить объекты Future в один список и описать, что должно произойти, когда завершатся все асинхронные операции в этом списке.

В конечном итоге после обработки всех запросов этот код посылает сообщение Events обратно отправителю, используя еще один шаблон программирования, вызов метода pipe, который упрощает отправку акторам значений из объектов Future.

Не волнуйтесь, если что-то осталось вам непонятным; впереди нас ждет целая глава, посвященная этой теме. Мы просто попытались заинтересовать вас этой замечательной возможностью – прочитайте главу 5 прямо сейчас, если вам не терпится узнать, как работают эти неблокирующие асинхронные операции.

На этом мы завершаем знакомство с основными особенностями `BoxOfFuture`. В нашем приложении остался еще один актор, который мы рассмотрим в следующем разделе: `RestApi`.

### 2.2.4. Актор RestApi

Актор `RestApi` использует язык маршрутизации HTTP, реализованный в Akka, который мы подробно рассмотрим в главе 12. Интерфейсам служб по мере их роста приходится осуществлять все более сложную маршрути-

зацию запросов. Поскольку мы просто создаем событие и затем продаем билеты на него, наши требования к маршрутизации не особенно велики. Актор `RestApi` определяет несколько классов, которые затем использует для преобразования данных в/из формата JSON (см. листинг 2.20).

**Листинг 2.20.** Сообщения, используемые в `RestApi`

```
case class EventDescription(tickets: Int) {
  require(tickets > 0)
}
case class TicketRequest(tickets: Int) {
  require(tickets > 0)
}
case class Error(message: String)
```

← Сообщение с начальным количеством билетов на событие

← Сообщение с количеством покупаемых билетов

← Сообщение с описанием ошибки

Рассмотрим подробно, как выполняется маршрутизация простого запроса (листинг 2.21). Прежде всего `RestApi` должен обработать POST-запрос на создание события.

**Листинг 2.21.** Определение маршрута для создания события

```
def eventRoute =
  pathPrefix("events" / Segment) { event =>
    pathEndOrSingleSlash {
      post {
        // POST /events/:event
        entity(as[EventDescription]) { ed =>
          onSuccess(createEvent(event, ed.tickets)) {
            BoxOffice.EventCreated(event) => complete(Created, event)
            case BoxOffice.EventExists =>
              val err = Error(s"$event event exists already.")
              complete(BadRequest, err)
          }
        }
      } ~
      get {
        // GET /events/:event
        onSuccess(getEvent(event)) {
          _.fold(complete(NotFound))(e => complete(OK, e))
        }
      } ~
      delete {
        // DELETE /events/:event
        onSuccess(cancelEvent(event)) {
          _.fold(complete(NotFound))(e => complete(OK, e))
        }
      }
    }
  }
```

В случае успеха завершает обработку запроса кодом «201 Created»

Создает событие с помощью метода `createEvent`, который обращается к актору `BoxOffice`

В случае неудачи завершает обработку запроса кодом «400 BadRequest»

```

    }
  }
}

```

Для определения маршрутов используется трейт `BoxOfficeApi`, в котором имеются методы, обертывающие взаимодействия с актором `BoxOffice` так, что код на языке DSL описания маршрутов остается чистым и ясным, как можно видеть в листинге 2.22.

**Листинг 2.22.** Трейт `BoxOfficeAPI`, обертывающий все взаимодействия с актором `BoxOffice`

```

trait BoxOfficeApi {
  import BoxOffice._

  def createBoxOffice(): ActorRef

  implicit def executionContext: ExecutionContext
  implicit def requestTimeout: Timeout

  lazy val boxOffice = createBoxOffice()

  def createEvent(event: String, nrOfTickets: Int) =
    boxOffice.ask(CreateEvent(event, nrOfTickets))
      .mapTo[EventResponse]

  def getEvents() =
    boxOffice.ask(GetEvents).mapTo[Events]

  def getEvent(event: String) =
    boxOffice.ask(GetEvent(event))
      .mapTo[Option[Event]]

  def cancelEvent(event: String) =
    boxOffice.ask(CancelEvent(event))
      .mapTo[Option[Event]]

  def requestTickets(event: String, tickets: Int) =
    boxOffice.ask(GetTickets(event, tickets))
      .mapTo[TicketSeller.Tickets]
}

```

Актор `RestApi` реализует метод `createBoxOffice` для создания дочернего актора `BoxOffice`. В листинге 2.23 демонстрируется фрагмент, управляющий продажей билетов.

Листинг 2.23. Определение маршрута для продажи билетов

```
def ticketsRoute =
  pathPrefix("events" / Segment / "tickets") { event =>
    post {
      pathEndOrSingleSlash {
        // POST /events/:event/tickets
        entity(as[TicketRequest]) { request =>
          onSuccess(requestTickets(event, request.tickets)) { tickets =>
            if(tickets.entries.isEmpty) complete(NotFound)
            else complete(Created, tickets)
          }
        }
      }
    }
  }
}
```

Преобразует данные JSON из запроса в case-класс TicketRequest

В случае успеха завершает обработку запроса кодом «201 Created»

В случае неудачи завершает обработку запроса кодом «400 BadRequest»

Сообщения автоматически преобразуются обратно в формат JSON. Подробнее о том, как это делается, рассказывается в главе 12. Итак, мы познакомились со всеми актерами, участвовавшими в этой первой итерации разработки приложения GoTicks.com. Если вы следовали за нашими объяснениями, а тем более если пытались воспроизвести все это у себя, поздравляем! Вы только что увидели, как на основе акторов Акка создать полностью асинхронное приложение с полноценным интерфейсом REST API. Хотя само приложение довольно тривиально, мы реализовали его так, что все операции в нем выполняются конкурентно, и теперь оно легко масштабируется (благодаря поддержке конкурентного выполнения) и обладает высокой отказоустойчивостью (подробнее об этом ниже). В этом примере также показано, как реализовать асинхронную обработку в синхронной парадигме запрос/ответ мира HTTP. Мы надеемся, вы убедились, что для создания простого приложения достаточно написать всего несколько строк кода. Сравните этот подход с использованием более традиционных инструментов или фреймворков, и вы приятно удивитесь, насколько меньше кода пришлось писать для этого примера. И в качестве вишенки не торта мы покажем вам, что требуется, чтобы развернуть это минимальное приложение в облаке. В следующем разделе мы развернем его в Heroku.com.

## 2.3. Вперед, в облако

Heroku.com – популярный поставщик облачных услуг, поддерживающий приложения на Scala и предоставляющий бесплатные экземпляры, с которыми можно поэкспериментировать. В этом разделе мы покажем вам, как просто развернуть приложение GoTicks.com в облаке Heroku. Мы по-

лагаем, что вы уже установили инструменты Heroku (<https://toolbelt.heroku.com/>). Если нет, обращайтесь за инструкциями по установке на веб-сайт Heroku (<https://devcenter.heroku.com/articles/heroku-command>). Вам также понадобится создать учетную запись на heroku.com. Посетите сайт проекта и создайте учетную запись (кнопка **Sign Up** (Зарегистрироваться) говорит сама за себя). В следующем разделе мы сначала создадим приложение на heroku.com, а затем развернем и запустим его.

### 2.3.1. Создание приложения в облаке Heroku

Сначала войдите на сайт под своей учетной записью и создайте новое Heroku-приложение, которое послужит основой для развертывания нашего приложения GoTicks.com. Для этого выполните следующие команды в каталоге *chapter-up-and-running*.

**Листинг 2.24.** Создание приложения в облаке Heroku

```
heroku login
heroku create

Creating damp-bayou-9575... done,
stack is cedar
http://damp-bayou-9575.herokuapp.com/
|
git@heroku.com:damp-bayou-9575.git
```

Вы должны увидеть нечто похожее на листинг 2.24.

Нам нужно добавить в проект еще кое-что, чтобы окружение Heroku смогло собрать наш код. Первое – файл *project/plugins.sbt*.

**Листинг 2.25.** Файл *plugins.sbt*

```
resolvers += Classpaths.typesafeReleases
addSbtPlugin("com.eed3si9n" % "sbt-assembly" % "0.13.0")
addSbtPlugin("com.typesafe.sbt" % "sbt-native-packager"
  % "1.0.0")
```

← Использовать репозиторий Typesafe Releases

← Использовать sbt-assembly для создания одного большого файла JAR, необходимо для развертывания в Heroku

← Использовать упаковщик для создания сценария, запускающего приложение в Heroku

Этот короткий список инструкций создает один большой файл JAR и сценарий командной оболочки (в случае с Heroku – на языке Bash; Heroku действует под управлением Ubuntu Linux).

Нам также понадобится файл *Procfile* в каталоге *chapter-up-and-running*, который сообщит Heroku, что наше приложение должно выполняться в

*web dyno* – одной из разновидностей процессов в терминологии Heroku. Содержимое *Procfile* показано в листинге 2.26.

### Листинг 2.26. Heroku Procfile

```
web: target/universal/stage/bin/goticks
```

Он подсказывает Heroku, что для запуска приложения требуется выполнить сценарий на языке Bash, который создаст плагин *sbt-native-packager*. Давайте сначала проверим все на локальном компьютере:

```
sbt clean compile stage
heroku local
23:30:11 web.1 | started with pid 19504
23:30:12 web.1 | INFO [Slf4jLogger]: Slf4jLogger started
23:30:12 web.1 | REST interface bound to /0:0:0:0:0:0:0:5000
23:30:12 web.1 | INFO [HttpListener]: Bound to /0.0.0:5000
```

Удалит целевые файлы, выполнит сборку архива без его развертывания

Сообщит Heroku взять архив и запустить его локально

Heroku управляет загрузкой приложения; мы получили PID

Это все, что требуется для подготовки приложения к развертыванию в Heroku. Все необходимое можно подготовить локально, что позволяет провести первое развертывание максимально быстро. После фактического развертывания в Heroku вы увидите, что все последующие развертывания на новых экземплярах в облаке выполняются непосредственно из Git. Собственно развертывание приложения в облаке Heroku описывается в следующем разделе.

## 2.3.2. Развертывание и запуск в Heroku

Мы только что проверили, что можем локально запустить приложение командой *heroku local*. Мы создали новое приложение в Heroku командой *heroku create*. Она также добавляет *git remote* с именем *heroku* в конфигурацию Git. Теперь нам остается только отправить все изменения в репозиторий Git. После этого передать код в Heroku можно командой:

```
git subtree push --prefix chapter-up-and-running heroku master
----> Scala app detected
----> Installing OpenJDK 1.6...
.... // загрузка зависимостей
....
----> Compiled slug size is 43.1MB
----> Launching... done,
v1 http://damp-bayou-9575.herokuapp.com deployed to Heroku
```

Передать в Heroku для развертывания

Так же как прежде, Heroku собирает приложение, на этот раз на удаленном экземпляре

```
To git@heroku.com:damp-bayou-9575.git
* [new branch] master -> master
```

Наконец, команда *push* завершилась успехом: на удаленной машине создана ветвь *master*

Предполагается, что вы отправили все изменения в ветвь `master` и проект находится в корне репозитория `Git`. `Heroku` внедряется в процесс передачи кода в репозиторий `Git` и идентифицирует код как приложение на `Scala`. Загружает все зависимости в облако, компилирует код и запускает приложение. В заключение вы должны увидеть нечто похожее на листинг выше.

### Использование проекта `akka-in-action` из репозитория `GitHub`

Обычно требуется выполнить команду `git push heroku master`, чтобы развернуть приложение в `Heroku`. Если использовать наш проект `akka-in-action` из репозитория `GitHub`, эта команда не будет работать, потому что приложение находится не в корне репозитория `Git`. Чтобы исправить проблему, нужно сообщить `Heroku`, что следует использовать поддерево:

```
git subtree push --prefix chapter-up-and-running heroku master
```

Дополнительные подробности ищите в файле `README.md` в каталоге `chapter-up-and-running`.

Это видно из вывода команды создания приложения; обратите внимание, что `Heroku` определил наше приложение как приложение на `Scala`, поэтому была выполнена установка `OpenJDK`, после чего исходный код был скомпилирован и запущен. Теперь, когда приложение развернуто и выполняется в `Heroku`, можно воспользоваться утилитой `httpie` и протестировать приложение.

#### Листинг 2.27. Тестирование экземпляра `Heroku` с помощью `httpie`

```
http POST damp-bayou-9575.herokuapp.com/events/RHCP tickets:=250
http POST damp-bayou-9575.herokuapp.com/events/RHCP/tickets tickets:=4
```

В ответ на эти команды должны появиться те же результаты, что мы видели выше (листинг 2.10). Поздравляем! Вы только что развернули свое первое приложение `Akka` в `Heroku`! На этом мы завершаем первую итерацию разработки приложения `GoTicks.com`. Теперь приложение, развернутое в `Heroku`, можно вызвать из любой точки.

## 2.4. В заключение

В этой главе вы увидели, как мало требуется для создания полнофункциональной `REST`-службы на основе акторов. Все взаимодействия в ней

выполняются асинхронно. Когда мы протестировали службу с помощью `httpie`, она действовала в полном соответствии с нашими ожиданиями.

Мы даже развернули наше приложение в облаке (Heroku.com)! Надеемся, что вас воодушевил этот короткий пример практического применения Акка. Приложение `GoTicks.com` пока не готово к полноценной эксплуатации. В нем отсутствует возможность хранения билетов в базе данных. Мы развернули его в Heroku, но экземпляры `web-dyno` могут подменяться в любой момент времени, поэтому хранение билетов в памяти – в данном случае не выход. Приложение способно масштабироваться по вертикали, но пока не может масштабироваться на несколько узлов.

Но мы обещаем детально исследовать эти темы в следующих главах, где мы последовательно продолжим доводку системы до рабочего состояния. В следующей главе мы посмотрим, как тестировать системы акторов.

# Глава 3

## Разработка с актерами через тестирование

В этой главе:

- синхронное модульное тестирование акторов;
- асинхронное модульное тестирование акторов;
- модульное тестирование шаблонов обмена сообщениями между актерами.

Забавно вспоминать времена, когда методика TDD (Test-Driven Development – разработка через тестирование) впервые появилась на сцене – основное возражение состояло лишь в том, что создание тестов отнимает слишком много времени и тем самым тормозит разработку. Хотя это мнение редко можно услышать в наши дни, тем не менее существует огромная разница между разными стеками и разными этапами тестирования (такими как модульное или интеграционное тестирование). Почти каждое решение позволяет легко и быстро производить модульное тестирование и проверять компоненты по отдельности. Но когда дело доходит до тестирования взаимодействий, простота и скорость быстро испаряются. Актеры предоставляют очень интересное решение этой проблемы, что объясняется следующими причинами:

- актеры лучше подходят для тестирования, потому что олицетворяют поведение (и почти во всех фреймворках TDD имеется некоторая поддержка BDD – Behavior-Driven Development, или разработки через поведение);
- слишком часто обычные модульные тесты проверяют только интерфейс или, по крайней мере, поддерживают тестирование интерфейсов и поведения по отдельности;

- акторы изначально создавались для обмена сообщениями, что имеет огромное преимущество для тестирования, потому что позволяет легко симитировать любое поведение отправкой сообщения.

Прежде чем начать тестирование (и программирование), возьмем несколько идей из предыдущей главы и покажем, как они выражаются в коде, попутно представив Actor API для создания акторов, а также отправки и приема сообщений. Мы детально рассмотрим работу акторов и перечислим некоторые правила, которые важно соблюдать во избежание проблем. После этого перейдем к реализации некоторых распространенных сценариев, используя подход к разработке через тестирование для создания акторов и немедленно проверяя их работу. На каждом шаге мы сначала будем описывать цель, которую должен достичь код (одно из главных условий TDD). Затем мы напишем спецификацию теста для актора, с которой обычно начинается разработка кода. Потом напишем ровно столько кода, чтобы пройти тестирование, и повторим итерацию. Правила, которые важно соблюдать, чтобы предотвратить возможность появления состояния, общего для нескольких акторов, мы будем давать по мере необходимости, так же как некоторые тонкости работы акторов Akka, имеющие большое значение для разработки через тестирование.

## 3.1. Тестирование акторов

Для начала посмотрим, как осуществляется тестирование отправки и приема сообщений в стиле «отправил и забыл» (односторонний обмен), а затем в стиле запрос/ответ (двусторонний обмен). Мы будем пользоваться фреймворком модульного тестирования *ScalaTest*, который также используется для модульного тестирования Akka. *ScalaTest* – это фреймворк с интерфейсом в стиле *xUnit*. Если вы не знакомы с ним и хотели бы узнать больше, посетите сайт [www.scalatest.org](http://www.scalatest.org). Фреймворк *ScalaTest* разрабатывался с прицелом на удобочитаемость, поэтому код, использующий его, легко читается и позволяет понять суть тестирования даже тем, кто не особенно хорошо знаком с ним. На первый взгляд, тестирование акторов кажется сложнее, чем тестирование обычных объектов, по причинам, перечисленным ниже.

- *Необходимость согласования по времени* – сообщения посылаются асинхронно, поэтому в модульном тесте трудно согласовать момент, когда будет готово к проверке ожидаемое значение.
- *Асинхронность* – акторы способны выполняться параллельно, в разных потоках. Тестирование многопоточного выполнения намного сложнее однопоточного и требует применения примитивов поддержки конкуренции, таких как блокировки, защелки и барьеры для синхронизации результатов из нескольких акторов. Это именно то,

от чего хотелось бы уйти. Неправильное использование только одного барьера может заблокировать тест, что повлечет остановку всего набора тестов.

- *Отсутствие доступа к состоянию* – актер скрывает свое внутреннее состояние и не позволяет обращаться к нему. Актеры доступны только посредством ссылок ActorRef. Вызовы методов и проверка значений свойств акторов – обычное дело при модульном тестировании – невозможны по определению.
- *Взаимодействие/интеграция* – чтобы выполнить интеграционное тестирование нескольких акторов, необходим некоторый механизм «подслушивания» акторов, чтобы убедиться, что сообщения несут в себе ожидаемые значения. Пока неочевидно, как это можно сделать.

К счастью, в Акка имеется модуль akka-testkit. Он содержит множество инструментов, упрощающих тестирование акторов. Модуль akka-testkit поддерживает несколько типов тестов.

- *Однопоточное модульное тестирование* – обычно экземпляр актора недоступен непосредственно. Модуль akka-testkit реализует ссылки типа TestActorRef, открывающие доступ к фактическим экземплярам акторов. С их помощью можно напрямую тестировать экземпляры акторов, вызывая методы, определенные вами, и даже функцию receive в однопоточном окружении, как это обычно делается при тестировании простых объектов.
- *Многopotочное модульное тестирование* – модуль akka-testkit реализует классы TestKit и TestProbe, которые позволяют получать ответы от акторов, исследовать сообщения и настраивать предельное время ожидания конкретных сообщений. TestKit имеет также методы для проверки ожидаемых сообщений. Актеры в этом случае запускаются с использованием обычного диспетчера для многопоточного окружения.
- *Тестирование под управлением нескольких JVM* – Акка также поддерживает инструменты для тестирования в нескольких JVM, которые могут пригодиться для тестирования удаленных систем акторов. Тестирование под управлением нескольких JVM подробно обсуждается в главе 6.

Класс TestKit имеет поле типа TestActorRef, наследующего класс LocalActorRef, устанавливающее диспетчера CallingThreadDispatcher, созданного специально для тестирования. (Он действует не в отдельном потоке, а в потоке вызывающего кода.) Это позволяет использовать все перечисленные выше решения.

В зависимости от предпочтений вы можете использовать любой из перечисленных стилей. Ближе всего к фактическому окружению, в котором предстоит выполняться вашему коду, находится вариант многопоточного тести-

рования, тестирования с использованием класса `TestKit`. Основное свое внимание мы уделим многопоточному подходу, потому что с его помощью можно выявить проблемы, незаметные в однопоточном окружении. (Возможно, вы не удивитесь, если мы скажем, что тоже предпочитаем классическое модульное тестирование, без использования фиктивных объектов.)

Прежде чем начать, необходимо выполнить некоторые подготовительные действия, чтобы не повторяться без необходимости. После создания система акторов запускается и продолжает действовать, пока явно не будет остановлена. Во всех наших тестах нам придется создавать системы акторов и останавливать их. Чтобы упростить тестирование, определим небольшой трейт для использования во всех тестах, гарантирующий автоматическую остановку системы после завершения тестирования.

**Листинг 3.1.** Остановка системы после завершения тестирования

```
import org.scalatest.{ Suite, BeforeAndAfterAll }
import akka.testkit.TestKit

trait StopSystemAfterAll extends BeforeAndAfterAll {
  this: TestKit with Suite =>
  override protected def afterAll() {
    super.afterAll()
    system.shutdown()
  }
}
```

Наследует трейт `BeforeAndAfterAll` из модуля `ScalaTest`

Этот трейт можно использовать только в сочетании с тестами, использующими `TestKit`

Останавливает систему, созданную `TestKit`, по завершении всех тестов

Мы поместили этот файл в каталог `src/test/scala/aia/testdriven`, потому что весь код тестов должен находиться в каталоге `src/test/scala` – корневом каталоге для тестов. Мы будем подмешивать этот трейт в наши тесты, чтобы обеспечить автоматическое завершение системы после выполнения тестов. Класс `TestKit` предоставляет значение `system`, которое может использоваться тестами для создания акторов и всего остального, что должно присутствовать в системе.

В следующих разделах мы будем использовать модуль `akka-testkit` для тестирования некоторых распространенных сценариев использования акторов, в однопоточном и многопоточном окружениях. Существует несколько разных способов взаимодействий акторов друг с другом. Мы рассмотрим разные доступные варианты и протестируем конкретное взаимодействие с помощью модуля `akka-testkit`.

## 3.2. Односторонние взаимодействия

Как вы помните, мы отказались от подхода «вызвать функцию и ждать ответа», поэтому мы преднамеренно использовали односторонние взаимо-

действия. При использовании стиля «послал и забыл» мы не знаем, когда сообщение достигнет актора и достигнет ли вообще. Можно ли в такой ситуации как-то протестировать актора? Хотелось бы иметь возможность послать сообщение актору и затем убедиться, что тот выполнил работу, которую должен был выполнить. Актор, отвечающий на сообщения, должен обработать сообщение, выполнить какие-то действия, например послать сообщение другому актору, сохранить внутреннее состояние, выполнить операции с другим объектом или устройствами ввода/вывода. Если поведение актора не имеет никаких внешних проявлений, мы можем только проверить, что он обработал сообщение без ошибок, и попытаться исследовать состояние актора с помощью `TestActorRef`. Всего имеется три варианта, которые мы рассмотрим.

- `SilentActor` – актор, поведение которого не имеет внешних проявлений; это может быть некоторый промежуточный шаг, выполняемый актором для создания некоторого внутреннего состояния. В этом случае нам требуется убедиться, что актор хотя бы обработал сообщение и не возбудил исключения и завершился, а также проверить изменения во внутреннем состоянии актора.
- `SendingActor` – актор, посылающий сообщение другому актору (или, может быть, нескольким акторам) после обработки полученного сообщения. Мы будем рассматривать актор как «черный ящик» и исследовать сообщение, посланное им в ответ на полученное сообщение.
- `SideEffectingActor` – актор, получающий сообщение и взаимодействующий с обычным объектом некоторым способом. В этом случае после отправки сообщения можно проверить изменения в объекте, произведенные актором.

Мы напишем тест для каждой из этих разновидностей акторов, что поможет нам продемонстрировать средства проверки результатов в тестах.

### 3.2.1. Примеры `SilentActor`

Начнем с `SilentActor`. Так как это наш первый тест, кратко познакомимся с особенностями использования `ScalaTest`.

**Листинг 3.2.** Первый тест для проверки акторов, не имеющих внешних проявлений своего поведения

```
class SilentActor01Test extends TestKit(ActorSystem("testsystem"))
  with WordSpecLike
  with MustMatchers
```

Наследует `TestKit` и создает систему акторов для тестирования

`MustMatchers` реализует удобочитаемые проверки

`WordSpecLike` реализует удобочитаемый предметный язык для тестирования в стиле BDD

```

with StopSystemAfterAll {
  "A Silent Actor" must {
    "change state when it receives a message, single threaded" in {
      // Первоначально пишется тест, терпящий неудачу
      fail("not implemented yet")
    }
    "change state when it receives a message, multi-threaded" in {
      // Первоначально пишется тест, терпящий неудачу
      fail("not implemented yet")
    }
  }
}

```

Гарантирует остановку системы после завершения всех тестов

Тесты пишутся как текстовые определения

Каждый оператор «in» описывает конкретный тест

Этот код образует базовый каркас, необходимый нам для тестирования «немного» актора с поведением, не имеющим внешних проявлений. Мы используем тестирование в стиле WordSpec, потому что он позволяет записывать тесты в виде простых текстовых спецификаций, которые также будут отображаться в момент запуска теста (тесты определяют особенности поведения). В предыдущем примере мы создали спецификацию теста для «немых» акторов с поведением, не имеющим внешних проявлений, которая говорит: «change internal state when it receives a message» (изменяет внутреннее состояние при получении сообщения). Прямо сейчас тест терпит неудачу, поскольку он пока не реализован – как предполагает стиль разработки *красный-зеленый-рефакторинг*, сначала новый тест должен терпеть неудачу (красный), затем реализуется код, удовлетворяющий требованиям теста (зеленый), после этого можно выполнить рефакторинг кода. В листинге 3.3 определяется актер, который ничего не делает, и тест всегда терпит неудачу.

### Листинг 3.3. Первая реализация немногого актора

```

class SilentActor extends Actor {
  def receive = {
    case msg =>
  }
}

```

Принимает любые сообщения; не имеет внутреннего состояния

Чтобы выполнить все тесты сразу, запустите команду `sbt test`. Но также есть возможность выполнить только один тест. Для этого запустите `sbt` в интерактивном режиме и затем выполните команду `testOnly`. В следующем примере мы запускаем тест `aia.testdriven.SilentActor01Test`:

```

sbt
...
> testOnly aia.testdriven.SilentActor01Test

```

Теперь напишем тест, посылающий сообщение немому актору и проверяющий изменение его внутреннего состояния. Для прохождения первого теста нужно дописать актор `SilentActor`, а также определить его *объект-компаньон* (объект, имеющий имя, которое совпадает с именем актора). Объект-компаньон содержит протокол обмена сообщениями; то есть объявляет все сообщения, поддерживаемые актором `SilentActor`, что является отличным способом группировки сообщений, имеющих отношение друг к другу. Листинг 3.4 – это первая попытка.

**Листинг 3.4.** Однопоточное тестирование внутреннего состояния

```
"change internal state when it receives a message, single" in {
  import SilentActor._      ← Импорт сообщений

  val silentActor = TestActorRef[SilentActor]  ← Создание ссылки TestActorRef для
  silentActor ! SilentMessage("whisper")      тестирования в однопоточном окружении
  silentActor.underlyingActor.state must (contain("whisper")) ← Получение актора и
}                                              проверка его состояния
```

Это простейшая версия типичного сценария разработки через тестирование (TDD): вызвать что-то и проверить изменение состояния. Теперь напишем реализацию актора `SilentActor`. В листинге 3.5 приводится первая версия фактической реализации.

**Листинг 3.5.** Реализация `SilentActor`

```
object SilentActor {
  case class SilentMessage(data: String)
  case class GetState(receiver: ActorRef)
}

class SilentActor extends Actor {
  import SilentActor._
  var internalState = Vector[String]()

  def receive = {
    case SilentMessage(data) =>
      internalState = internalState :+ data  ← Состояние хранится в виде вектора; каждое
  }                                          принятое сообщение добавляется в этот вектор

  def state = internalState  ← Метод state возвращает
}                              встроенный вектор
```

← Тип сообщения, который может обрабатывать `SilentActor`

← Объект-компаньон, определяющий поддерживаемые сообщения

Поскольку возвращаемый список является неизменяемым, тест не сможет изменить его и вызвать проблемы при его проверке. Можно совершенно без опаски изменять/устанавливать `internalState`, потому что ак-

тор защищен от параллельного доступа из нескольких потоков. Вообще говоря, хорошей практикой считается использование полей `var` в сочетании с неизменяемыми структурами данных вместо полей `val` с изменяемыми структурами данных. (Это предотвращает случайное совместное использование изменяемого состояния, если вы каким-то образом передаете внутреннее состояние другому актору.)

Теперь рассмотрим многопоточную версию этого теста. Как вы увидите далее, нам также придется немного изменить код актора. Так же как в однопоточной версии нам пришлось добавить метод `state`, чтобы получить возможность протестировать актор, мы должны добавить некоторый код для поддержки тестирования в многопоточном окружении. В листинге 3.6 показано, как осуществляется тестирование.

**Листинг 3.6.** Многопоточное тестирование внутреннего состояния

```
"change internal state when it receives a message, multi" in {
  import SilentActor._

  val silentActor = system.actorOf(Props[SilentActor], "s3")
  silentActor ! SilentMessage("whisper1")
  silentActor ! SilentMessage("whisper2")
  silentActor ! GetState(testActor)
  expectMsg(Vector("whisper1", "whisper2"))
}
```

Объект-компаньон, определяющий поддерживаемые сообщения

Для создания актора используется тестовая система

Сообщение, добавленное в объект-компаньон, для получения состояния

Используется для проверки отправки сообщений экземпляру testActor

Для создания актора `SilentActor` многопоточный тест использует систему акторов `ActorSystem`, являющуюся частью `TestKit`.

Актор всегда создается из объекта `Props`. Объект `Props` описывает порядок создания актора. Самый простой способ создать объект `Props` – создать его с типом актора в его аргументе типа, в данном случае `Props[SilentActor]`. Объект `Props`, созданный таким способом, в конечном счете создаст актора, используя его конструктор по умолчанию.

Поскольку в многопоточном окружении нельзя просто так обратиться к экземпляру актора, нам пришлось придумать другой способ увидеть изменение состояния. Для этого было добавлено сообщение `GetState`, которое получает `ActorRef`. Класс `TestKit` имеет поле `testActor`, которое можно использовать для приема ожидаемых сообщений. Мы добавили метод `GetState`, чтобы дать актору `SilentActor` возможность послать ему свое внутреннее состояние. Теперь мы можем вызвать метод `expectMsg`, который ожидает отправки одного сообщения актору `testActor` и проверяет его; в данном случае это `Vector` со всеми полями данных.

### Настройка тайм-аута для методов expectMsg\*

TestKit имеет несколько версий expectMsg и других методов для проверки сообщений. Все эти методы ждут сообщения в течение определенного времени; если сообщение в заданное время не поступило, они возбуждают исключение. Тайм-аут имеет значение по умолчанию, определяемое параметром akka.test.single-expect-default в конфигурации. Для вычисления фактического времени тайм-аута используется коэффициент расширения (обычно он равен 1, то есть тайм-аут не расширяется). Его назначение – дать средство уравнивания компьютеров, имеющих разную вычислительную мощность. На медленном компьютере мы должны быть готовы ждать дольше (обычно разработчики выполняют тесты на своих быстрых рабочих станциях, а затем сталкиваются с проблемами на медленных серверах непрерывной интеграции). Для каждого компьютера можно настроить свой коэффициент, обеспечивающий успешное прохождение теста (подробнее о настройке рассказывается в главе 7). Кроме того, непосредственно в методе можно установить максимальный тайм-аут, но лучше использовать значения в конфигурации и подставлять значения, необходимые для тестирования, в конфигурации.

Теперь нам осталось только добавить код в немой актер, который обрабатывает сообщение GetState.

#### Листинг 3.7. Реализация SilentActor

```
object SilentActor {
  case class SilentMessage(data: String)
  case class GetState(receiver: ActorRef)
}

class SilentActor extends Actor {
  import SilentActor._
  var internalState = Vector[String]()

  def receive = {
    case SilentMessage(data) =>
      internalState = internalState :+ data
    case GetState(receiver) => receiver ! internalState
  }
}
```

← Сообщение GetState, добавленное ради тестирования

← Внутреннее состояние посылается по ссылке ActorRef в сообщении GetState

Внутреннее состояние посылается обратно в сообщении GetState по ссылке ActorRef, в данном случае актору testActor. Поскольку роль внут-

ренного состояния играет неизменяемый вектор, такая отправка совершенно безопасна. Это верно для обоих типов `SilentActor`: одно- и многопоточного вариантов. Используя этот прием, можно конструировать тесты, знакомые большинству программистов: изменение состояния можно проверить с помощью нескольких инструментов из `TestKit`.

### 3.2.2. Пример `SendingActor`

Обычно актер получает ссылку `ActorRef` с помощью метода `props` для использования на более позднем этапе отправки сообщения. В этом примере мы создадим актер `SendingActor`, который сортирует списки событий и посылает сортированные списки актору-получателю.

**Листинг 3.8.** Тест для актора, посылающего сообщения

```
"A Sending Actor" must {
  "send a message to another actor when it has finished processing" in {
    import SendingActor._
    val props = SendingActor.props(testActor)
    val sendingActor = system.actorOf(props, "sendingActor")

    val size = 1000
    val maxInclusive = 100000

    def randomEvents() = (0 until size).map{ _ =>
      Event(Random.nextInt(maxInclusive))
    }.toVector

    val unsorted = randomEvents()
    val sortEvents = SortEvents(unsorted)
    sendingActor ! sortEvents

    expectMsgPF() {
      case SortedEvents(events) =>
        events.size must be(size)
        unsorted.sortBy(_.id) must be(events)
    }
  }
}
```

← Получатель передается в метод `props`, который создаст `Props`; внутри теста передается `testActor`

← Создание несортированного списка случайных событий

← `testActor` должен получить сортированный список событий

Сообщение `SortEvents` посылается актору `SendingActor`. Оно содержит список событий, который требуется отсортировать. Актер `SendingActor` должен отсортировать события и послать сообщение `SortedEvents` актору-получателю. В тесте вместо настоящего актора, который должен был обработать отсортированный список событий, мы передаем `testActor`, что реализуется очень просто, потому что получатель – это всего лишь ссылка `ActorRef`. Так как сообщение `SortEvents` содержит неупорядоченный

список случайных событий, мы не можем использовать `expectMsg(msg)`; мы не можем заранее предсказать виды и порядок событий в сообщении. В данном случае мы используем метод `expectMsgPF`. Здесь мы сопоставляем сообщение, отправленное актору `testActor`, которое должно быть сообщением `SortedEvents`, содержащим отсортированный вектор событий. Если попробовать выполнить этот тест прямо сейчас, он потерпит неудачу, потому что мы еще не реализовали протокол обмена сообщениями в `SendingActor`. Сделаем это прямо сейчас.

**Листинг 3.9.** Реализация `SendingActor`

```
object SendingActor {
  def props(receiver: ActorRef) =
    Props(new SendingActor(receiver))
  case class Event(id: Long)
  case class SortEvents(unsorted: Vector[Event])
  case class SortedEvents(sorted: Vector[Event])
}

class SendingActor(receiver: ActorRef) extends Actor {
  import SendingActor._
  def receive = {
    case SortEvents(unsorted) =>
      receiver ! SortedEvents(unsorted.sortBy(_.id))
  }
}
```

Оба сообщения - `SortEvents` и `SortedEvents` - несут в себе неизменяемый вектор

- ← Получатель передается через `Props` в конструктор актора `SendingActor`; в тесте мы передаем `testActor`
- ← Сообщение `SortEvents` посылается актору `SendingActor`
- ← Сообщение `SortedEvents` посылается получателю после того, как `SendingActor` отсортирует список

Мы снова создаем объект-компаньон, содержащий протокол обмена сообщениями. Он также имеет метод `props`, создающий объект `Props` для актора. В данном случае актору нужно передать ссылку на актор-получатель, поэтому здесь используется другая разновидность `Props`.

Вызов `Props(arg)` транслируется в вызов метода `Props.apply`, который принимает параметр до востребования (by-name parameter) `creator`. Параметры до востребования вычисляются, только когда на них ссылаются в первый раз, то есть инструкция `new SendingActor(receiver)` выполняется только один раз, когда фреймворку `Akka` потребуется создать его. Создание `Props` в объекте-компаньоне имеет свои преимущества, не позволяя обращаться к внутренним механизмам актора, если вдруг понадобится создать актора из актора. Использование внутренних механизмов актора из `Props` могло бы привести к состоянию гонки или вызвать проблемы с сериализацией, если сам объект `Props` будет помещен в сообщение, пересылаемое по сети. Мы и далее будем использовать этот рекомендованный прием для создания экземпляров `Props`.

Актор `SendingActor` сортирует вектор с помощью метода `sortBy` – он создает сортированную копию вектора, которую можно без всякой опаски передать дальше. Затем получателю посылается сообщение `SortedEvents`. И снова мы используем преимущество неизменяемости `case`-классов и структуры данных `Vector`.

Рассмотрим некоторые распространенные разновидности типа `SendingActor`, перечисленные в табл. 3.1.

**Таблица 3.1.** Виды `SendingActor`

Актер	Описание
<code>MutatingCopyActor</code>	Создает изменяемую копию и посылает ее следующему актору, как описывается в этом разделе
<code>ForwardingActor</code>	Пересылает принятое сообщение, вообще ничего не меняя
<code>TransformingActor</code>	На основе полученного сообщения создает сообщение другого типа
<code>FilteringActor</code>	Часть сообщений пересылает другим актерам, часть сообщений просто отбрасывает
<code>SequencingActor</code>	Создает множество сообщений на основе полученного и посылает их одно за другим другому актору

Все акторы – `MutatingCopyActor`, `ForwardingActor` и `TransformingActor` – можно тестировать, используя один и тот же подход. Им можно передать `testActor` в качестве следующего актора для передачи сообщений и использовать `expectMsg` или `expectMsgPF` для проверки. `FilteringActor` отличается тем, что мы должны знать, какие сообщения он будет передавать дальше, а какие – нет. `SequencingActor` требует аналогичного подхода. Как проверить, что мы получили правильное количество сообщений? Следующий тест отвечает на этот вопрос.

Напишем тест для `FilteringActor`. Актор `FilteringActor` должен фильтровать повторяющиеся события. Он хранит список последних принятых сообщений и проверяет каждое следующее входящее сообщение на наличие повторений. (Это сопоставимо с типичными элементами фреймворков тестирования, основанных на применении фиктивных объектов, которые позволяют проверять вызовы и подсчитывать их количество.)

### Листинг 3.10. Тест для `FilteringActor`

```
"filter out particular messages" in {
  import FilteringActor._
  val props = FilteringActor.props(testActor, 5)
  val filter = system.actorOf(props, "filter-1")
  filter ! Event(1)
  filter ! Event(2)
```

← Послать несколько событий,  
включая повторяющиеся

```

filter ! Event(1)
filter ! Event(3)
filter ! Event(1)
filter ! Event(4)
filter ! Event(5)
filter ! Event(5)
filter ! Event(6)
val eventIds = receiveWhile() {
  case Event(id) if id <= 5 => id
}
eventIds must be(List(1, 2, 3, 4, 5))
expectMsg(Event(6))
}

```

← Принимает сообщения, пока они совпадают с инструкцией case  
 ← Проверяет отсутствие повторяющихся сообщений в результате

Тест использует метод `receiveWhile`, который выбирает сообщения, принятые актором `testActor`, пока инструкция `case` находит совпадения. В тесте `Event(6)` не соответствует шаблону в инструкции `case`, требующему, чтобы числовой идентификатор события был меньше или равен 5, что заставляет его прервать цикл `while`. Метод `receiveWhile` возвращает собранные элементы в том же виде, в каком их вернула частично вычисленная функция, – в виде списка. Теперь напишем актор `FilteringActor`, соответствующий этой спецификации.

### Листинг 3.11. Реализация `FilteringActor`

```

object FilteringActor {
  def props(nextActor: ActorRef, bufferSize: Int) =
    Props(new FilteringActor(nextActor, bufferSize))
  case class Event(id: Long)
}

class FilteringActor(nextActor: ActorRef,
                    bufferSize: Int) extends Actor {
  import FilteringActor._
  var lastMessages = Vector[Event]()
  def receive = {
    case msg: Event =>
      if (!lastMessages.contains(msg)) {
        lastMessages = lastMessages :+ msg
        nextActor ! msg
        if (lastMessages.size > bufferSize) {
          // отбросить более старое
          lastMessages = lastMessages.tail
        }
      }
  }
}

```

← Максимальный размер буфера передается в конструктор  
 ← Вектор для хранения последних сообщений  
 ← Если событие не найдено в буфере, оно пересылается следующему актору  
 ← По заполнении буфера из него удаляется самое старое событие

Актор `FilteringActor` хранит буфер последних полученных сообщений и добавляет в него каждое вновь полученное сообщение, если оно отсутствует в списке. Актору `nextActor` посылаются только сообщения, отсутствовавшие в буфере. При заполнении буфера, когда его размер достигает предельного значения `bufferSize`, из него удаляется самое старое сообщение, чтобы предотвратить разрастание буфера до огромного объема и, возможно, исчерпания доступной памяти.

Метод `receiveWhile` также можно использовать для тестирования актора `SequencingActor`; с его помощью можно убедиться, что последовательность сообщений, вызванных определенным событием, соответствует ожиданиям. Еще два метода, которые могут пригодиться для проверки количества сообщений, – `ignoreMsg` и `expectNoMsg`. Метод `ignoreMsg` принимает частично вычисленную функцию, подобно методу `expectMsgPF`, только вместо проверки сообщения он игнорирует сообщения, соответствующие шаблону. Это удобно, когда, например, интересуют не все сообщения, посланные актору `testActor`, а только некоторые. Метод `expectNoMsg` проверяет отсутствие сообщений в течение некоторого времени, который также можно было бы использовать между посылками повторяющихся сообщений в тесте для `FilteringActor`. В следующем листинге демонстрируется пример теста, использующий метод `expectNoMsg`.

### Листинг 3.12. Еще один тест для `FilteringActor`

```
"filter out particular messages using expectNoMsg" in {
  import FilteringActor._
  val props = FilteringActor.props(testActor, 5)
  val filter = system.actorOf(props, "filter-2")
  filter ! Event(1)
  filter ! Event(2)
  expectMsg(Event(1))
  expectMsg(Event(2))
  filter ! Event(1)
  expectNoMsg
  filter ! Event(3)
  expectMsg(Event(3))
  filter ! Event(1)
  expectNoMsg
  filter ! Event(4)
  filter ! Event(5)
  filter ! Event(5)
  expectMsg(Event(4))
  expectMsg(Event(5))
  expectNoMsg()
}
```

Так как `expectNoMsg` ждет определенное время, проверяя отсутствие сообщений, этот тест выполняется гораздо медленнее.

Как видите, `TestKit` предоставляет актор `testActor`, способный принимать сообщения, которые можно проверить с помощью `expectMsg` и других методов. Класс `TestKit` имеет только один актор `testActor`. А можно ли организовать тестирование актора, посылающего сообщения нескольким акторам? Да, например с помощью класса `TestProbe`. Класс `TestProbe` очень похож на класс `TestKit`, отличаясь только отсутствием необходимости писать свою реализацию, наследуя его. Достаточно создать экземпляр `TestProbe` вызовом конструктора `TestProbe()` – и можно использовать его. `TestProbe` часто будет использоваться в тестах, которые мы еще напишем в этой книге.

### 3.2.3. Пример `SideEffectingActor`

В листинге 3.13 приводится пример очень простого актора `Greeter`, который вводит приветствие на основе полученного сообщения. (Это версия примера «Hello World» на основе актора.)

**Листинг 3.13.** Актор `Greeter`

```
import akka.actor.{ActorLogging, Actor}

case class Greeting(message: String)

class Greeter extends Actor with ActorLogging {
  def receive = {
    case Greeting(message) => log.info("Hello {}!", message)
  }
}
```

Вывод приветствия, включающего полученное сообщение

Актор `Greeter` выполняет единственное действие: он принимает сообщение и выводит его в консоль. `SideEffectingActor` позволяет тестировать ситуации, когда эффект действия невозможно определить непосредственно. Под это определение подходят многие случаи, тем не менее листинг 3.14 в достаточной мере иллюстрирует средства тестирования ожидаемого результата.

**Листинг 3.14.** Тестирование актора `Greeter`

```
import Greeter01Test._

class Greeter01Test extends TestKit(testSystem)
  with WordSpecLike
  with StopSystemAfterAll {

  "The Greeter" must {
```

Использует `testSystem` из объекта `Greeter01Test`

```

"say Hello World! when a Greeting("World") is sent to it" in {
  val dispatcherId = CallingThreadDispatcher.Id
  val props = Props[Greeter].withDispatcher(dispatcherId)
  val greeter = system.actorOf(props)
  EventFilter.info(message = "Hello World!",
    occurrences = 1).intercept {
    greeter ! Greeting("World")
  }
}
}
}
}

object Greeter01Test {
  val testSystem = {
    val config = ConfigFactory.parseString(
      """
        akka.loggers = [akka.testkit.TestEventListener]
      """)
    ActorSystem("testsystem", config)
  }
}
}

```

Однопоточное окружение

Перехватит журналируемое сообщение

Создание системы с подключенным приемником тестовых событий

Актор `Greeter` тестируется путем проверки журналируемых сообщений, которые тот записывает с помощью трейта `ActorLogging`. Модуль `akka-testkit` содержит класс `TestEventListener`, который можно использовать для обработки всех журналируемых событий. `ConfigFactory` может анализировать конфигурацию из строки; в данном случае мы просто переопределяем список обработчиков событий.

Тест выполняется в однопоточном окружении, нам требуется убедиться, что `TestEventListener` записывает событие, когда `Greeter` посылает приветствие «World». Здесь мы использовали объект `EventFilter`, позволяющий фильтровать журналируемые сообщения. В данном случае мы отфильтровываем ожидаемое сообщение, которое должно появиться только однажды. Фильтр применяется, когда выполняется блок кода `intercept`, то есть когда производится отправка сообщения.

Преыдуший пример тестирования актора `SideEffectingActor` показывает, что проверка некоторых взаимодействий может быстро превратиться в сложнейшую задачу. Во многих ситуациях проще адаптировать код для упрощения тестирования. Очевидно, что если передать приемники в тестируемый класс, нам не придется изменять конфигурацию или применять фильтрацию; мы просто получим все сообщения, посылаемые тестируемым актором. В листинге 3.15 приводится адаптированная версия актора `Greeter`, которую можно настроить на отправки сообщения *актору-приемнику*, когда регистрируется очередное приветствие.

**Листинг 3.15.** Упрощение тестирования актора Greeter с помощью приемника

```
object Greeter02 {
  def props(listener: Option[ActorRef] = None) =
    Props(new Greeter02(listener))
}

class Greeter02(listener: Option[ActorRef])
  extends Actor with ActorLogging {
  def receive = {
    case Greeting(who) =>
      val message = "Hello " + who + "!"
      log.info(message)
      listener.foreach(_ ! message)
  }
}
```

Конструктор принимает необязательный приемник; по умолчанию получает значение None

Отправка сообщения приемнику, если определен

Актор `Greeter02` адаптирован так, что принимает аргумент `Option[ActorRef]` со значением по умолчанию `None` в методе `props`. После успешной записи сообщения в журнал он посылает его приемнику `listener`, если был получен непустой аргумент `Option`. При обычном использовании актор не получает ссылку на приемник и ничего никому не посылает. В листинге 3.16 приводится измененная версия теста для этого актора `Greeter02`.

**Листинг 3.16.** Упрощенный тест актора Greeter

```
class Greeter02Test extends TestKit(ActorSystem("testsystem"))
  with WordSpecLike
  with StopSystemAfterAll {

  "The Greeter" must {
    "say Hello World! when a Greeting(\"World\") is sent to it" in {
      val props = Greeter02.props(Some(testActor))
      val greeter = system.actorOf(props, "greeter02-1")
      greeter ! Greeting("World")
      expectMsg("Hello World!")
    }
    "say something else and see what happens" in {
      val props = Greeter02.props(Some(testActor))
      val greeter = system.actorOf(props, "greeter02-2")
      system.eventStream.subscribe(testActor, classOf[UnhandledMessage])
      greeter ! "World"
      expectMsg(UnhandledMessage("World", system.deadLetters, greeter))
    }
  }
}
```

Передача актора `testActor` в роли приемника

Обычная проверка сообщения

Как видите, реализация теста существенно упростилась. Мы просто передаем `Some(testActor)` в конструктор `Greeter02` и проверяем сообщение, посланное актору `testActor`, как обычно.

В следующем разделе мы перейдем к реализации двусторонних взаимодействий и их тестированию.

### 3.3. Двусторонние взаимодействия

Вы уже видели пример двусторонних взаимодействий в многопоточном тесте для актора, реализованного в стиле `SendingActor`, где мы использовали сообщение `GetState` со ссылкой `ActorRef`. Мы просто вызывали оператор `!` с этой ссылкой `ActorRef`, чтобы ответить на запрос `GetState`. Как было показано прежде, метод `tell` принимает неявную ссылку `sender`.

В следующем тесте (листинг 3.17) мы используем трейт `ImplicitSender`. Он передает в неявной ссылке `sender` ссылку на актор из `akka-testkit`.

**Листинг 3.17.** `ImplicitSender`

```
class EchoActorTest extends TestKit(ActorSystem("testsystem"))
  with WordSpecLike
  with ImplicitSender
  with StopSystemAfterAll {
```

← Передает в неявной ссылке `sender` ссылку на актор в классе `TestKit`

Двусторонние взаимодействия легко тестируются в стиле «черного ящика»: на запрос обязательно должен прийти ответ, что легко проверить. В следующем тесте (листинг 3.18) проверяется актор `EchoActor`, который в ответ на любой запрос возвращает то же сообщение.

**Листинг 3.18.** Тестирование актора `EchoActor`

```
"Reply with the same message it receives without ask" in {
  val echo = system.actorOf(Props[EchoActor], "echo2")
  echo ! "some message"
  expectMsg("some message")
}
```

← Обычная проверка сообщения

← Отправка сообщения актору

Мы просто посылаем сообщение, а `EchoActor` возвращает ответ по ссылке, указывающей на актор из `akka-testkit`, которая автоматически была передана как `sender` трейтом `ImplicitSender`. Реализация актора `EchoActor` не содержит ничего особенного, он просто посылает полученное сообщение обратно отправителю, как показано в листинге 3.19.

**Листинг 3.19.** `EchoActor`

```
class EchoActor extends Actor {
  def receive = {
```

```

case msg =>
  sender() ! msg
}
}

```



← Отправка полученного сообщения  
обратно отправителю

Актор `EchoActor` действует в точности, как если бы был задействован шаблон `ask` или метод `tell`; выше показан более предпочтительный способ тестирования двусторонних взаимодействий.

В этом разделе мы кратко рассмотрели идиомы тестирования акторов, предлагаемые классом `TestKit` из фреймворка `Akka`. Все они решают одну и ту же задачу: упрощают разработку модульных тестов, которым требуется доступ к результатам для их проверки. Класс `TestKit` предоставляет методы для тестирования в обоих окружениях – одно- и многопоточном. Мы даже можем немного «схитрить» и в ходе тестирования получить доступ к экземпляру актора. Классификация акторов по способам взаимодействий с другими акторами дает нам шаблоны тестирования, которые были показаны на примере разновидностей `SilentActor`, `SendingActor` и `SideEffectingActor`. Часто самый простой способ протестировать актор – передать ему ссылку на актор `testActor`, который можно использовать для проверки ожидаемых сообщений, посылаемых тестируемым актором. `testActor` можно использовать в роли отправителя в шаблоне запрос/ответ или в роли следующего актора, которому тестируемый актор пересылает полученные сообщения. Наконец, вы видели, что часто имеет смысл подготовить актор для тестирования, особенно если актор «немой», и в этом случае полезно добавить в актор необязательный приемник.

## 3.4. В заключение

Разработка через тестирование (TDD) – это больше, чем просто механизм проверки качества; это особый подход к работе. Фреймворк `Akka` изначально проектировался с прицелом на поддержку TDD. Поскольку основой обычного модульного тестирования является вызов метода и получение ответа, который можно проверить, сопоставив с ожидаемым результатом, мы должны были найти способ принять новое мышление, чтобы двигаться вперед, используя стиль асинхронного программирования, основанный на обмене сообщениями.

Акторы также дают некоторые новые возможности опытным программистам TDD:

- воплощают поведение; тесты фактически проверяют поведение акторов;
- тестирование с применением сообщений выглядит яснее: с сообщениями передается только неизменяемое состояние, что не позволяет тестам повредить тестируемое состояние;

- понимая основы тестирования акторов, можно писать модульные тесты для акторов всех видов.

Эта глава познакомила вас с приемами тестирования, реализованными в Akka, и инструментами, которые предоставляет фреймворк. Неопровержимые доказательства их ценности приводятся в следующих главах, где они используются для быстрой разработки тестируемого рабочего кода.

В следующей главе мы посмотрим, как формируются иерархии акторов и как можно использовать стратегии наблюдения за жизненным циклом для создания отказоустойчивых систем.

# Глава 4

## Отказоустойчивость

В этой главе:

- создание самовосстанавливающихся систем;
- принцип «и пусть падает»;
- жизненный цикл акторов;
- наблюдение за акторами;
- выбор стратегии восстановления после отказа.

Эта глава охватывает инструменты Akka для повышения надежности и отказоустойчивости приложений. Первый раздел описывает принцип «и пусть падает» (let-it-crash), включая средства мониторинга и управления акторами, а также их жизненный цикл. При этом мы, разумеется, рассмотрим некоторые примеры их использования в типичных сценариях отказов.

### 4.1. Что такое отказоустойчивость

Для начала определим, что мы подразумеваем под *отказоустойчивостью* системы и почему следует писать код, предполагающий возможность отказа. В идеальном мире система всегда доступна и может гарантировать успешное выполнение любого действия. Существует только два пути достижения идеала: использовать компоненты, которые никогда не отказывают, или учесть все возможные причины отказов и предусмотреть способы восстановления после них. Однако в большинстве архитектур у нас имеется только механизм, который завершает работу, как только обнаружится необработанная ошибка. Даже если приложение пытается реализовать стратегии восстановления, тестировать их очень сложно, а кроме того, эти стратегии сами добавляют еще один уровень сложности. В процедурном мире каждая попытка сделать что-то требует, чтобы код, получающий результат, проверил его на все возможные виды ошибок. Об-

работка исключений, ставшая основой современных языков, предлагает менее обременительный подход к реализации способов восстановления. Но, хотя исключения позволили писать код, не нуждающийся в проверке ошибок в каждой строке, распространение исключений по обработчикам не привнесло существенных улучшений.

Идея создания системы, свободной от ошибок, выглядит восхитительно, но, к сожалению, создание любой нетривиальной системы с высокой доступностью просто невозможно. Главная причина в том, что значительные части любой нетривиальной системы вам неподконтрольны, и в этих частях могут возникать отказы. Возникает распространенная проблема разделения ответственности: когда в системе имеется большое количество компонентов, совместно используемых разными частями, не всегда ясно, кто должен отвечать за обработку возможных ошибок. Хорошим примером могут служить ошибки сети: она в любой момент может стать недоступной, полностью или частично, и если система должна продолжить работу в этих условиях, вам придется найти некоторый другой способ взаимодействий или, может быть, приостановить взаимодействия на какое-то время. Вы можете зависеть от сторонних служб, которые могут допускать ошибки в работе, терпеть неудачу или периодически быть недоступными. Серверы, на которых действует ваше программное обеспечение, могут оказываться недоступными или даже выходить из строя. Очевидно, что вы не волшебник и не сможете магическим образом заставить сервер восстать из пепла или исправить сломанный диск, чтобы закончить операцию записи на него. Вот почему в джунглях телекоммуникационных систем родился принцип «*и пусть падает*», где отказ техники – не такое уж редкое явление и обеспечение высокой доступности невозможно без плана, учитывающего такие отказы.

Не имея возможности предотвратить все отказы, вы должны предусмотреть стратегию действий, учитывая следующее:

- все рано или поздно ломается. Системы должны быть *отказоустойчивыми* настолько, чтобы оставались доступными и продолжали работу. Ошибки, допускающие возможность восстановления, не должны вызывать катастрофических последствий;
- в некоторых случаях допустима ситуация, когда наиболее важные функции системы остаются доступными, в то время как некоторые другие компоненты терпят ошибки и отключаются от системы, чтобы не мешать ей и не вызывать непредсказуемых результатов;
- в других случаях некоторые компоненты могут быть настолько важными, что для них необходимо создавать активные резервные копии (возможно, действующие на других серверах или использующие другие ресурсы), которые могут включаться в работу по выходу из строя главного компонента и быстро восстанавливать доступность;

- отказы отдельных частей системы не должны вызывать крах всей системы, поэтому нужно предусмотреть способ изоляции отказов, чтобы разобраться с ними потом.

Конечно же, фреймворк Akka не может предложить средства на все случаи жизни. Вам все равно придется реализовать обработку конкретных отказов, но Akka может сделать такую реализацию более чистой и ясной. В табл. 4.1 перечислены возможности, имеющиеся в арсенале Akka, которые помогут вам реализовать отказоустойчивое поведение.

**Таблица 4.1.** Стратегии борьбы с отказами

Стратегия	Описание
Ограничение последствий ошибок или их изоляция	Неисправности должны изолироваться в компонентах, где они возникли, и не вызывать краха всей системы
Структура	Для изоляции отказавшего компонента от остальной системы необходима какая-то структура, обеспечивающая такую изоляцию; система должна определить структуру, в которой можно изолировать активные компоненты
Избыточность	Резервный компонент должен иметь возможность перехватить управление после отказа основного компонента
Замена	Если неисправный компонент можно изолировать, значит, его также можно заменить в структуре. Должные части системы должны сохранить возможность взаимодействовать с компонентом, созданным на замену
Перезагрузка	Если компонент оказывается в неправильном состоянии, нужна возможность вернуть его обратно в начальное правильное состояние. Неправильное состояние может быть причиной отказа, и предсказать все неправильные состояния компонента часто невозможно просто потому, что не все его зависимости подконтрольны вам
Жизненный цикл компонента	Неисправный компонент должен изолироваться, и при невозможности исправить проблему его следует завершить и удалить из системы или инициализировать правильным начальным состоянием. Компонент должен поддерживать определенный жизненный цикл для запуска, перезапуска и завершения компонента
Приостановка	Когда компонент выходит из строя, все обращения к нему должны приостанавливаться до его исправления или замены, чтобы, когда это случится, компонент мог продолжить работу, не потеряв ни бита информации. Вызов, обрабатывавшийся в момент отказа, также не должен исчезать бесследно – он может иметь решающее значение для восстановления, а также содержать информацию, которая может помочь понять причину отказа. Возможно, вам понадобится повторить вызов, если вы уверены, что причина отказа заключена в чем-то другом

Стратегия	Описание
Разделение ответственности	Было бы замечательно, если бы код восстановления после отказа можно было отделить от кода, выполняющего основную работу. Восстановление после отказа является сквозной задачей в процессе нормальной работы. Ясное отделение нормального процесса от процесса восстановления упростит вам задачу. Изменить способ восстановления приложения после ошибки будет намного проще при четком разделении

«Но, минутку, – можете сказать вы, – почему нельзя просто использовать для восстановления старые добрые объекты и исключения?» Обычно исключения используются для отделения последовательности операций и предотвращения попадания в неправильное состояние, а не для восстановления после ошибки в том смысле, как обсуждалось выше. Но давайте посмотрим в следующем разделе, насколько сложно было бы добавить восстановление с использованием исключений и простых объектов.

### 4.1.1. Простые объекты и исключения

Рассмотрим пример многопоточного приложения, которое читает файлы журналов, извлекает из них информацию в объекты и записывает эти объекты в некоторую базу данных. Некоторый процесс следит за появлением новых файлов журналов и каким-то образом сообщает потокам, что требуется обработать новые файлы. На рис. 4.1 приводится схема приложения с выделенной частью, которую мы рассмотрим подробнее («В центре внимания»).

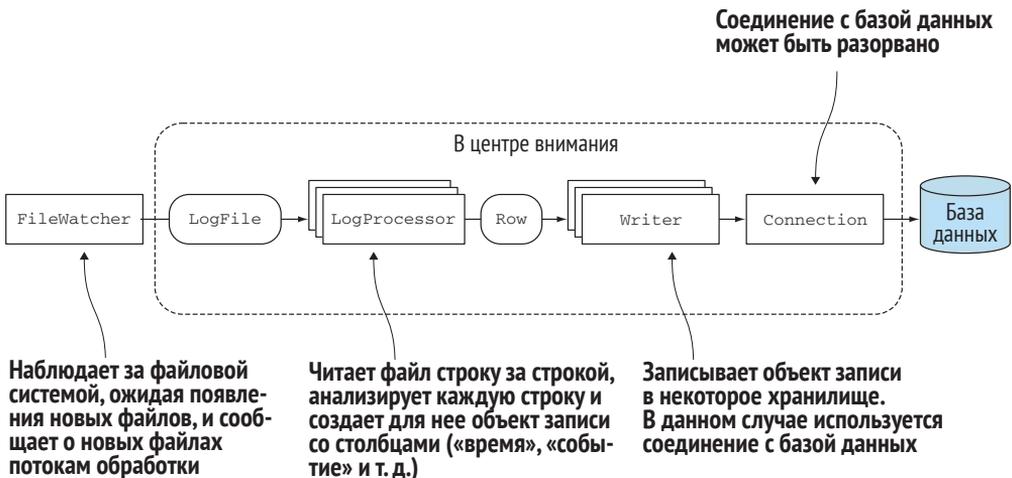


Рис. 4.1. Приложение обработки файлов журналов

На случай разрыва соединения с базой данных мы должны создать новое соединение с другой базой данных и продолжить запись данных. Если со-

единение начинает работать неправильно, мы должны закрыть его, чтобы никакая другая часть приложения не смогла воспользоваться им. В некоторых случаях желательно попытаться переустановить соединение, чтобы восстановить его нормальную работу. Для иллюстрации потенциальных проблем мы будем использовать псевдокод. Рассмотрим ситуацию, когда достаточно установить новое соединение с базой данных, используя стандартный механизм обработки исключений.

Сначала подготовим все объекты, которые будут использоваться в потоках для обработки новых файлов. Также настроим компонент записи в базу данных, использующий объект соединения. На рис. 4.2 показано, как создается компонент, осуществляющий запись. Зависимости, необходимые этому компоненту, передаются через конструктор, как это обычно принято. Параметры для фабрики соединения с базой данных, включая URL, передаются из потока, создающего компонент.

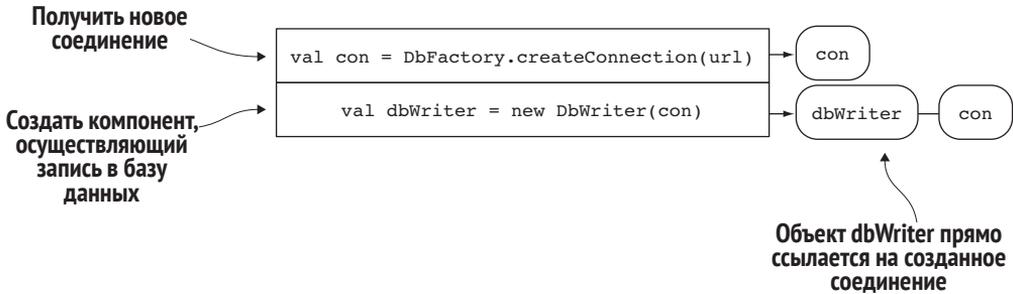


Рис. 4.2. Создание компонента, осуществляющего запись в базу данных

Далее настроим несколько компонентов обработки файлов журналов; каждый получает ссылку на компонент записи, которому он должен передавать готовые объекты записей, как показано на рис. 4.3.

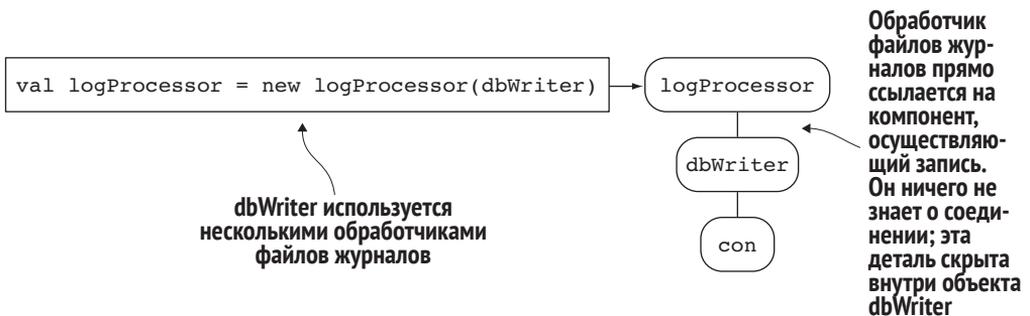


Рис. 4.3. Создание компонента, обрабатывающего файлы журналов

На рис. 4.4 показано, как объекты в этом примере приложения вызывают друг друга.

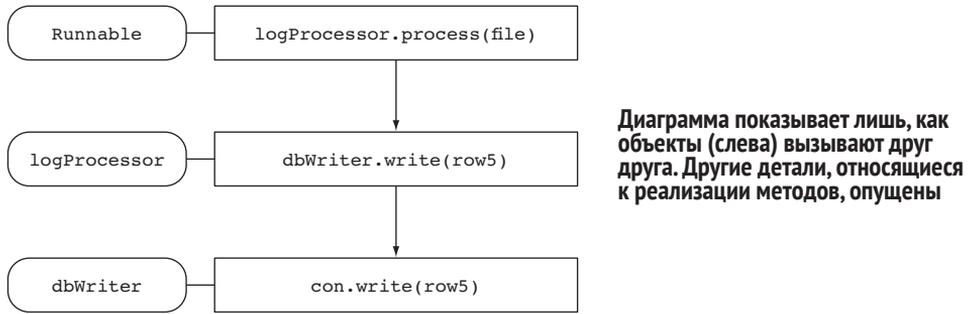


Рис. 4.4. Стек вызовов

Действия, изображенные на рис. 4.4, выполняются в нескольких потоках, одновременно обрабатывающих файлы, найденные компонентом `FileWatcher`. На рис. 4.5 показан стек вызовов в момент возбуждения исключения `DbBrokenConnectionException`, которое подсказывает, что следует переключиться на другое соединение. Детали реализации методов опущены; диаграмма лишь показывает, как объекты вызывают друг друга.

В данном случае недостаточно просто удалить исключение `DbBrokenConnectionException` из стека, нам также нужно заменить нарушенное соединение действующим. Первая проблема, с которой мы сталкиваемся, – сложность добавления кода, восстанавливающего соединение, чтобы он не нарушил дизайн. Кроме того, в этой точке у нас недостаточно информации для воссоздания соединения: мы не знаем, какие строки в файле были уже успешно обработаны, а какие обрабатывались в момент появления исключения.

Если сделать информацию о соединении и обработанных строках доступной всем объектам, это усложнит дизайн и нарушит правила инкапсуляции, инверсии управления и единственной ответственности. (Привет коллегам, сторонникам чистой архитектуры, которые будут просматривать такой код!) Нам требуется всего лишь заменить неисправный компонент. Добавление восстановительного кода непосредственно в обработчик исключения лишь запутает реализацию обработки файлов логикой восстановления соединения с базой данных. Даже если мы найдем удачное место для воссоздания соединения, мы должны быть очень осторожны, чтобы другие потоки не использовали неисправный компонент, пока мы пытаемся заменить его новым, потому что иначе некоторые записи окажутся потерянными.

Кроме того, передача исключений между потоками не является стандартной возможностью; вам придется самим реализовать ее, а это очень непросто. Давайте рассмотрим требования к отказоустойчивости, чтобы понять, имеет ли этот подход хоть какие-то шансы.

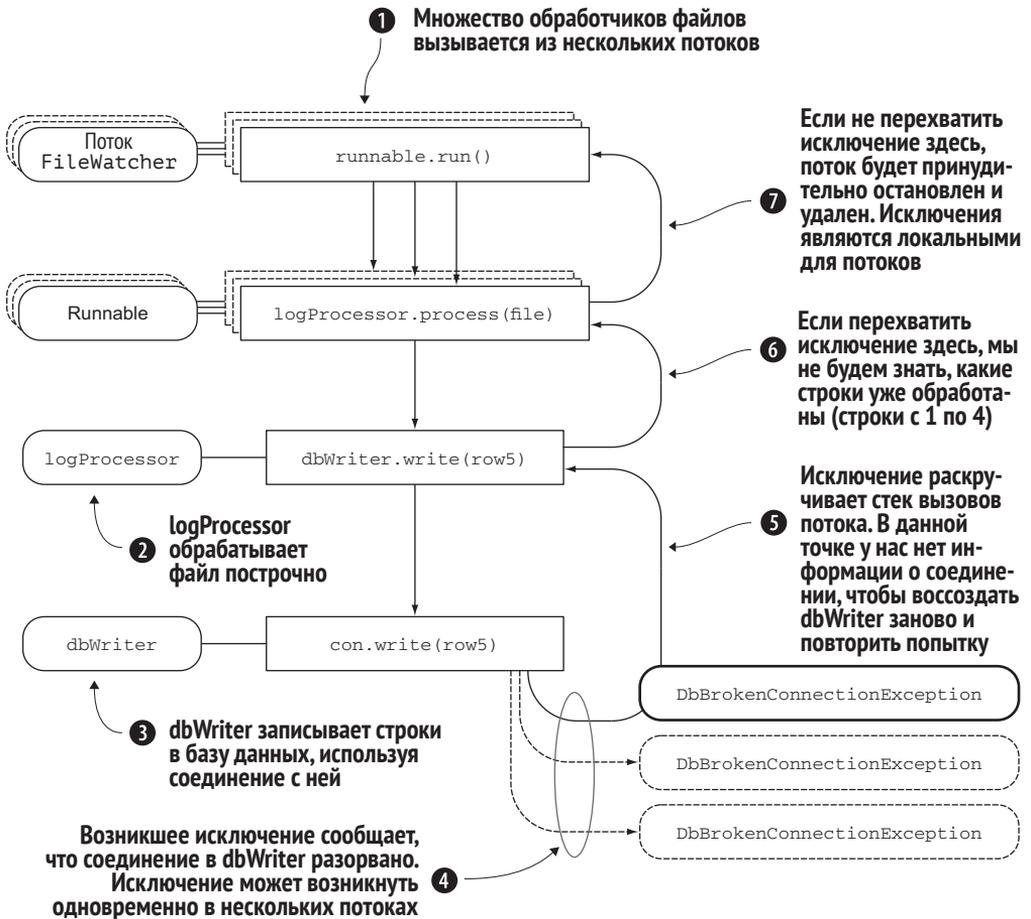


Рис. 4.5. Стек вызовов во время обработки файлов журналов

- *Изоляция ошибки* – изоляция осложняется возможностью появления исключения одновременно в нескольких потоках. Вам придется добавить какой-то механизм блокировки. Удаление неисправного соединения из цепочки объектов – действительно очень сложная задача: приложение придется переписать, чтобы решить ее. Из-за отсутствия стандартного механизма исключения соединения из работы придется реализовать свой механизм, добавив в объекты дополнительный уровень косвенности.
- *Структура* – объекты организованы в простую и понятную цепочку. Одни объекты ссылаются на другие, образуя граф; их очень непросто заменить во время выполнения. Вам придется создать более сложную структуру (добавив некоторый уровень косвенности).
- *Избыточность* – когда возникает исключение, оно начинает раскручивать стек вызовов в обратном направлении. Вы можете пропустить

контекст принятия решения для использования избыточного компонента или потерять контекст, из которого можно получить данные для продолжения работы, как показано в предыдущем примере.

- *Замена* – не существует стратегии по умолчанию для замены объекта в стеке вызовов; вам придется самостоятельно найти решение этой задачи. Существуют некоторые фреймворки внедрения зависимостей, которые обладают некоторыми возможностями для этого, но, если какой-то объект прямо ссылается на старый экземпляр, минуя уровень косвенности, это вызовет серьезные сложности. Если вы намерены заменить объект на месте, вам придется каким-то образом гарантировать работу этого приема в многопоточном окружении.
- *Перезагрузка* – так же как в случае с заменой, автоматическая поддержка возврата объектов в начальное состояние отсутствует, и вам придется самим реализовать ее, добавив уровень косвенности. При этом придется повторно внедрить все необходимые зависимости. Если эти зависимости также потребуют перезагрузки (обработчик файла журнала тоже может возбудить некоторое исключение, допускающее возможность восстановления), ситуация осложнится еще больше.
- *Жизненный цикл компонента* – объект существует в период от момента создания до уничтожения сборщиком мусора. Если вам потребуется какой-то иной механизм, вам придется самостоятельно реализовать его.
- *Приостановка* – входные данные или некоторый контекст становятся недоступными после того, как возникшее исключение начнет свое распространение вверх по стеку. Вам потребуется реализовать свой механизм буферизации для сохранения входящих вызовов до момента исправления ошибки. Если код вызывается из нескольких потоков, вам придется добавить блокировки, предотвращающие одновременное появление нескольких исключений, и найти способ сохранения входных данных для выполнения повторной попытки позже.
- *Разделение ответственности* – код обработки исключений переплетается с кодом обработки файлов, и их невозможно разделить.

Как видите, ситуация выглядит удручающей: добиться отказоустойчивости в такой архитектуре будет очень и очень сложно. Отсутствуют многие функции, необходимые для поддержки отказоустойчивости:

- воссоздание объектов с зависимостями и их замена в структуре приложения являются очень непростой задачей;
- объекты напрямую взаимодействуют друг с другом, поэтому их сложно изолировать;
- код восстановления после ошибки и функциональный код тесно переплетаются друг с другом.

К счастью, имеется более простое решение. Мы уже видели некоторые возможности акторов, помогающие упростить решение подобных проблем. Актors можно повторно (вос)создавать из объектов Props, они являются частью системы акторов и взаимодействуют посредством косвенных ссылок. В следующем разделе мы посмотрим, как акторы избавляют от переплетения функционального кода с кодом восстановления после ошибки и как жизненный цикл акторов делает возможным их приостановку и перезапуск (не вызывая гнева богов конкурентного выполнения) в процессе восстановления после сбоя.

### 4.1.2. И пусть падает

В предыдущем разделе вы узнали, что создание отказоустойчивых приложений с применением простых объектов и обработчиков исключений – очень сложная задача. Давайте теперь посмотрим, как акторы упрощают ее. Что должно произойти, когда актер обрабатывает сообщение и сталкивается с исключением? Мы уже объяснили, почему нежелательно внедрять код восстановления после ошибки в рабочий процесс, поэтому обработка исключения внутри актора, где находится прикладная логика, тоже нам не подходит.

Вместо одного общего потока для выполнения прикладного кода и обработки ошибок акторы Akka предлагают использовать два отдельных потока. Один – для акторов, обрабатывающих обычные сообщения, и другой – для восстановления после ошибок, включающий акторы, которые следят за работой акторов из первого потока. Актors, наблюдающие за другими актерами, называют *супервизорами* (supervisors). На рис. 4.6 показан супервизор, наблюдающий за работой прикладного актора.

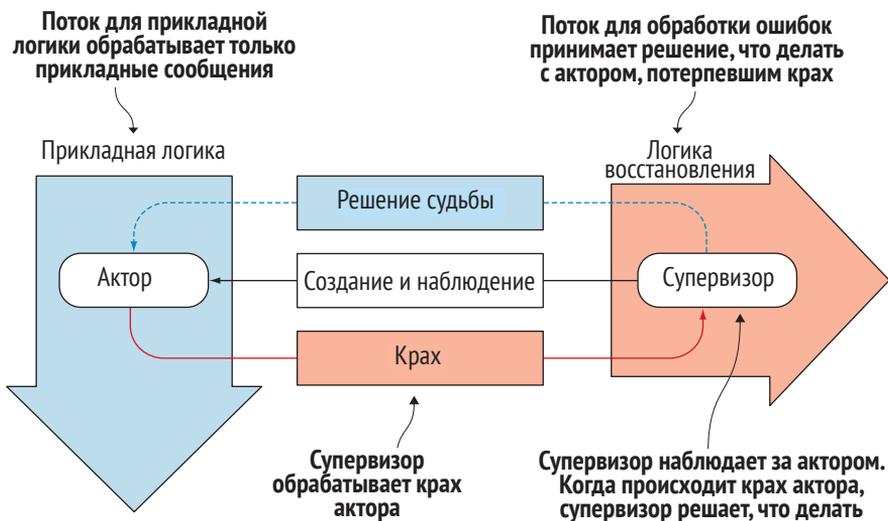


Рис. 4.6. Потoki для прикладного кода и кода восстановления после ошибок

Вместо обработки исключения мы просто позволим актору потерпеть крах. Код обработки сообщений в акторе будет содержать только прикладную логику, он никак не связан с процессом восстановления после ошибки и поэтому будет выглядеть простым и ясным. Почтовый ящик актора, потерпевшего крах, будет приостанавливаться, пока супервизор, действующий в потоке обработки ошибок, не решит, что делать с исключением. Как актер превратить в супервизор? В Akka используется решение, получившее название *родительский контроль*, в том смысле, что любой актер, создающий другие акторы, автоматически становится их супервизором. Супервизор не «перехватывает исключений»; он лишь решает, что должно произойти, когда потерпит крах актер, находящийся под его наблюдением. Супервизор не пытается зафиксировать актер или его состояние. Он лишь определяет выбор и выполнение стратегии восстановления. Супервизору доступны четыре варианта решения, что делать с актором:

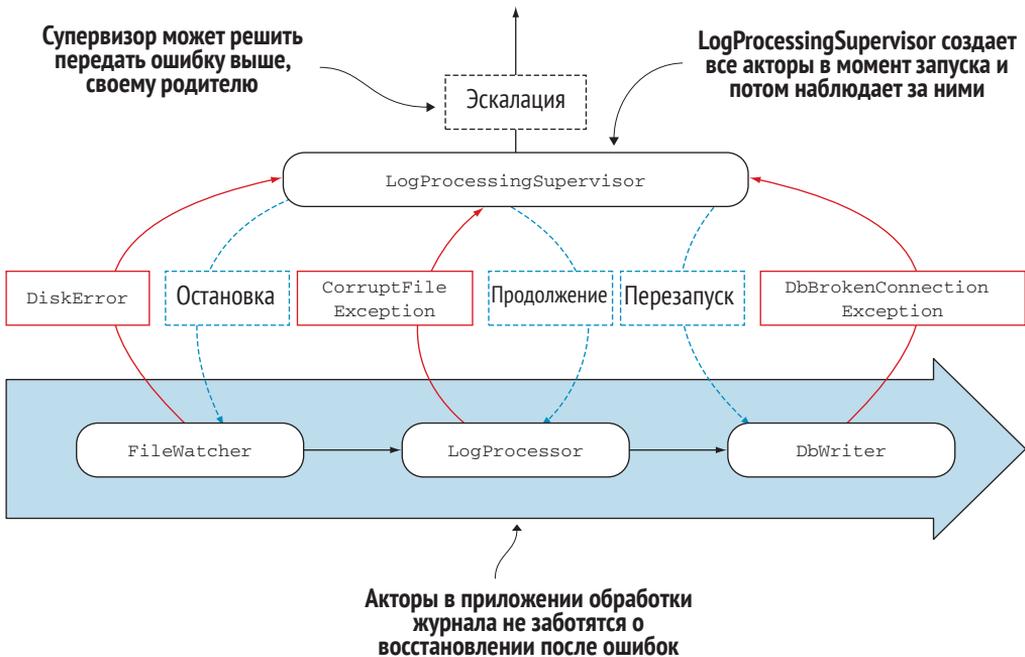
- *перезапуск* – повторно создать актер из соответствующего объекта Props. После перезапуска (перезагрузки) актер сможет продолжить обработку сообщений. Поскольку для взаимодействия с актором все остальные компоненты приложения используют ссылку ActorRef, новый экземпляр актора автоматически будет получать новые сообщения;
- *продолжение* – возобновить работу прежнего экземпляра актора; в этом случае ошибка просто игнорируется;
- *остановка* – завершить актер. В этом случае актер исключается из процесса обработки сообщений;
- *эскалация* – супервизор не знает, что делать с ошибкой, и передает право решения своему родителю, который тоже является супервизором.

На рис. 4.7 приводится пример стратегии, которую мы могли бы выбрать в приложении обработки журналов с помощью акторов. Супервизор принимает одно из возможных действий в ответ на появление ошибки.

На рис. 4.7 показано решение для увеличения отказоустойчивости приложения обработки файлов журналов, по крайней мере для случая разрыва соединения. Когда возникает исключение `DbBrokenConnectionException`, актер `dbWriter` терпит аварию и замещается вновь созданным актором `dbWriter`.

Нам нужно предпринять какие-то специальные шаги, чтобы восстановить сообщение, в ходе обработки которого произошла авария, но о них мы поговорим позже, когда будем обсуждать детали реализации перезапуска. А пока отметим, что в некоторых случаях может быть нежелательно начинать повторную обработку сообщения, так как высока вероятность, что ошибка вызвана им. Примером может служить случай, когда `logProcessor` встречает поврежденный файл: повторная попытка обработать повреж-

денный файл может закончиться так называемым *отравлением почтового ящика* – актер не сможет обработать никакие другие сообщения, потому что сообщение с поврежденным файлом будет вызывать аварию снова и снова. По этой причине в Акка принято решение не возвращать аварийное сообщение в почтовый ящик после перезапуска, но у вас есть возможность поступить иначе, если вы абсолютно уверены, что сообщение не является причиной ошибки, – этот прием мы обсудим позже. Однако если суть работы заключается в обработке десятков тысяч сообщений и одно из них окажется повреждено, поведение по умолчанию обеспечит нормальную обработку всех остальных сообщений; один поврежденный файл не вызовет катастрофических последствий и не будет препятствовать обработке всех последующих файлов.



**Рис. 4.7.** Потоки, прикладной и обработки ошибок, в приложении обработки журналов

На рис. 4.8 показано, как аварийный экземпляр актора `dbWriter` замещается новым экземпляром, когда супервизор решает выполнить перезапуск.

Повторим еще раз, какие преимущества дает подход «и пусть падает».

- *Изоляция ошибок* – супервизор может решить остановить актер и удалить его из системы акторов.
- *Структура* – иерархия ссылок на акторы в системе допускает возможность замены экземпляров акторов, не влияя на работу других акторов.

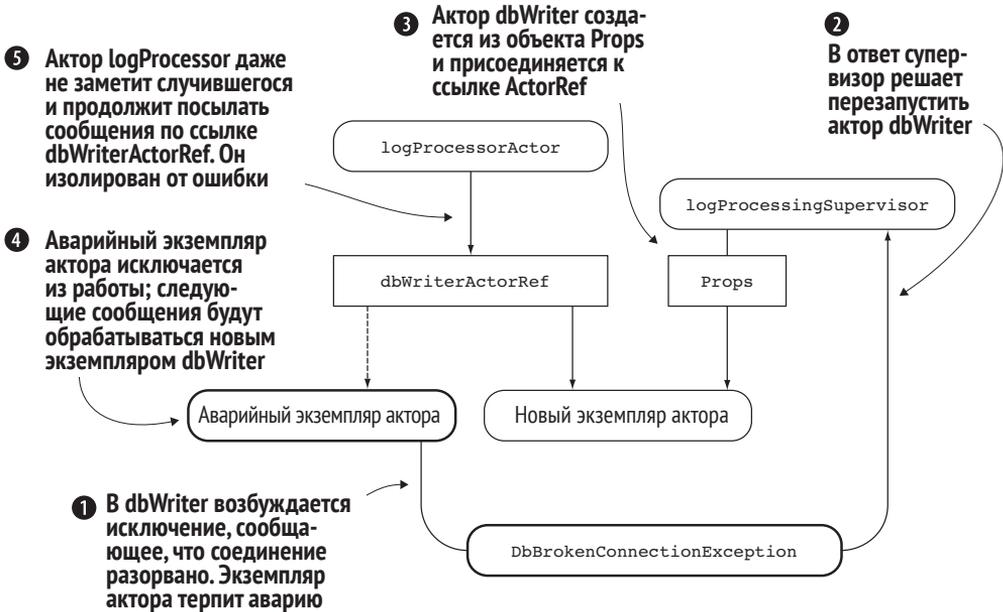


Рис. 4.8. Обработка исключения `DbBrokenConnectionException` с перезапуском

- *Избыточность* – любой актер можно заменить другим актером. В примере с разрывом соединения с базой данных новый актер может попытаться подключиться к другой базе данных. Супервизор может также решить остановить аварийный актер и создать ему на замену актер другого типа. Другой вариант – использовать схему балансировки нагрузки с распределением сообщений между несколькими актерами, о чем рассказывается в главе 9.
- *Замена* – всегда есть возможность создать новый актер из его объекта `Props`. Супервизор может решить заменить аварийный экземпляр актера новым, даже не зная тонкостей, необходимых для создания актера.
- *Перезагрузка* – действие, аналогичное перезапуску.
- *Жизненный цикл компонента* – актер является активным компонентом. Его можно запустить, остановить и перезапустить. В следующем разделе мы подробнее рассмотрим жизненный цикл актеров.
- *Приостановка* – когда актер терпит аварию, работа его почтового ящика приостанавливается, пока супервизор не решит, что делать с актером.
- *Разделение ответственности* – обычная обработка сообщений и наблюдение за работой других актеров никак не связаны между собой. Эти задачи можно реализовать и выполнять независимо друг от друга.

В следующих разделах мы познакомимся с особенностями жизненного цикла акторов и стратегий наблюдения.

## 4.2. Жизненный цикл актора

Теперь вы знаете, что актора можно перезапустить, чтобы восстановить нормальную работу после отказа. Но как исправить состояние актора в момент перезапуска? Чтобы ответить на этот вопрос, нужно поближе познакомиться с жизненным циклом акторов. Фреймворк Akka автоматически запускает актор сразу после его создания. Актор пребывает в состоянии `Started` (запущен), пока не будет остановлен, после чего перейдет в состояние `Terminated` (завершен). После остановки актор не может обрабатывать сообщения и рано или поздно будет утилизирован сборщиком мусора. Пребывающий в состоянии `Started` актор можно перезапустить и сбросить его внутреннее состояние в исходное положение. В предыдущем разделе мы рассмотрели пример замены одного экземпляра актора новым. Перезапуск может происходить столько раз, сколько потребуется. В жизненном цикле актора возникают события трех типов:

- актор создан и запущен – для простоты мы будем называть это событие *start* (пуск);
- по событию *restart* (перезапуск) актор перезапускается;
- по событию *stop* (стоп) актор останавливается.

В трейте `Actor` имеется несколько методов-обработчиков, которые вызываются по событиям, извещающим об изменении состояния актора. Вы можете добавить свои обработчики, которые будут вызываться для воссоздания определенного состояния в новом экземпляре актора, например для обработки сообщения, которое не было обработано до перезапуска, или для освобождения некоторых ресурсов. В следующих разделах рассмотрим эти три события и покажем, как определить свои методы для их обработки. Порядок вызова методов гарантирован, даже притом, что они вызываются фреймворком асинхронно.

### 4.2.1. Событие *start*

Актор создается и автоматически запускается вызовом метода `actorOf`. Акторы верхнего уровня создаются методом `actorOf` системы акторов `ActorSystem`. Родительский актор создает дочерние акторы, используя метод `actorOf` своего контекста `ActorContext`. Этот процесс изображен на рис. 4.9.

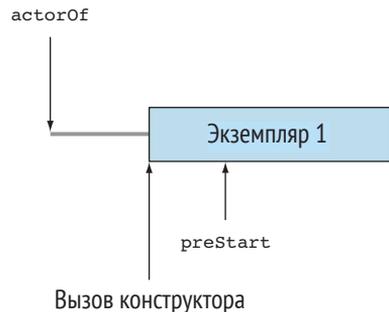


Рис. 4.9. Запуск актора

После создания экземпляра Akka запускает его. Непосредственно перед запуском вызывается метод `preStart` актора. Чтобы воспользоваться этим фактом, нужно переопределить метод `preStart`.

#### Листинг 4.1. Обработчик `preStart`

```
override def preStart(): Unit = {
  println("preStart")      ← Выполнить некоторые действия
}
```

В этом обработчике можно инициализировать состояние актора. Также состояние актора можно инициализировать в его конструкторе.

### 4.2.2. Событие `stop`

Следующее событие, возникающее в жизненном цикле актора, которое мы обсудим, – это событие `stop`. Событие `restart` мы обсудим чуть позже, потому что его обработчик зависит от обработчиков событий `start` и `stop`. Событие `stop` указывает на завершение жизненного цикла актора и возникает только один раз, когда актор останавливается. Остановить актор можно вызовом метода `stop` объектов `ActorSystem` и `ActorContext` или пошлав актору сообщение `PoisonPill`. Этот процесс иллюстрирует рис. 4.10.

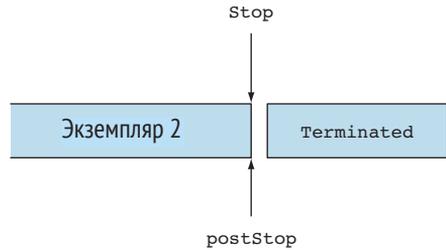


Рис. 4.10. Остановка актора

Непосредственно перед остановкой вызывается метод `postStop` актора. Актор, находящийся в состоянии `Terminated`, не получает новых сообщений для обработки. Метод `postStop` является парным для метода `preStart`.

#### Листинг 4.2. Обработчик `postStop`

```
override def postStop(): Unit = {
  println("postStop")     ← Выполнить некоторые действия
}
```

Обычно этот обработчик реализует действия, обратные действиям в `preStart`, – освобождает ресурсы, зарезервированные в методе `preStart`, и, при необходимости, сохраняет последнее состояние актора в каком-нибудь внешнем хранилище на случай, если оно понадобится потом. Остановленный актор отсоединяется от своей ссылки `ActorRef`, которая затем подключается к `deadLettersActorRef` системы акторов – специальной ссылке `ActorRef`, которая принимает все сообщения, посланные акторам после их остановки.

### 4.2.3. Событие restart

В процессе работы супервизор может решить перезапустить актора. Перезапуск может происходить неоднократно, в зависимости от возникающих ошибок. Это событие сложнее событий `start` и `stop`, потому что при этом происходит замена экземпляра актора. Данный процесс иллюстрирует рис. 4.11.

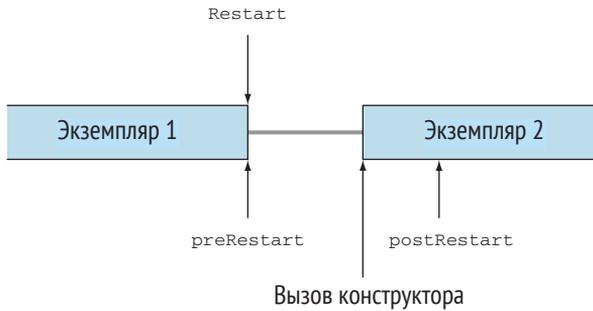


Рис. 4.11. Перезапуск актора

Непосредственно перед перезапуском вызывается метод `preRestart` аварийного экземпляра актора. В этом методе аварийный экземпляр может сохранить свое текущее состояние.

#### Листинг 4.3. Обработчик `preRestart`

```
override def preRestart(reason: Throwable, message:
Option[Any]): Unit = {
  println("preRestart")
  super.preRestart(reason, message)
}
```

Внимание: вызов реализации по умолчанию

Когда ошибка возникает в функции `receive`, в этом параметре передается сообщение, которое актор пытался обработать

Исключение, сгенерированное актором

Будьте внимательны, переопределяя этот обработчик. Реализация метода `preRestart` по умолчанию останавливает все дочерние акторы и затем вызывает обработчик `postStop`. Если забыть вызвать `super.preRestart`, эти действия по умолчанию не будут выполнены. Помните, что акторы (`вос`) создаются из объекта `Props`, который, в свою очередь, вызывает конструктор актора. Актор может создавать дочерние акторы внутри своего конструктора. Если потомков аварийного актора не остановить, в результате при каждом перезапуске родительского актора в системе будут накапливаться экземпляры его дочерних акторов.

Важно отметить, что событие «перезапуск» не останавливает аварийный актор, как это делает вызов метода `stop` (описывался выше). Как вы увидите далее, есть возможность управлять остановкой актора. Когда производится перезапуск, аварийный экземпляр не получает сообщения `PoisonPill` и

не переходит в состояние `Terminated`. Просто создается новый экземпляр, который затем подключается к той же ссылке `ActorRef`, которая использовалась аварийным актором перед ошибкой. Остановленный актор отключается от своей ссылки `ActorRef`, и все его сообщения перенаправляются в `deadLettersActorRef`, как описывалось выше, когда мы обсуждали событие `stop`. Остановленный и перезапускаемый актор объединяет только то, что по умолчанию после отключения от системы для обоих вызывается метод `postStop`.

Метод `preRestart` принимает два аргумента: `reason`, определяющий причину перезапуска, и необязательный аргумент `message` – сообщение, обрабатывавшееся актором в момент аварии. Супервизор может решить, что следует (или можно) сохранить для включения в состояние восстановленного актора. Этого нельзя сделать с помощью локальных переменных, потому что после перезапуска управление передается новому экземпляру. Одно из возможных решений передачи состояния между аварийным и новым экземплярами – послать сообщение актору, которое попадет в его почтовый ящик. (Для этого актор должен послать сообщение по своей собственной ссылке `ActorRef`, которая доступна внутри актора через значение `self`.) Также можно сохранить состояние в каком-нибудь внешнем хранилище, например в базе данных или в файловой системе. Выбор способа полностью зависит от вашей системы и поведения актора.

А теперь вернемся к примеру обработки файлов журналов, где мы решили не терять сообщение `Row` в случае аварии `dbWriter`. Решением в данном случае могла бы стать отправка сообщения `Row` по своей собственной ссылке `ActorRef`, чтобы оно было обработано свежим экземпляром актора. Одна из проблем, которые следует учитывать при таком подходе: когда сообщение посылается обратно в почтовый ящик, порядок сообщений в нем меняется. Отправленное сообщение помещается последним и будет обработано позже других сообщений, уже имеющихся в почтовом ящике. Для актора `dbWriter` это не является проблемой, но помните об этой особенности, применяя данный прием.

После вызова метода `preStart` создается новый экземпляр актора, и, как следствие, объект `Props` выполняет его конструктор. После этого вызывается метод `postRestart` нового экземпляра.

#### Листинг 4.4. Метод `postRestart`

```
override def postRestart(reason: Throwable): Unit = {
  println("postRestart")
  super.postRestart(reason)
}
```

← Исключение, сгенерированное актором

← Внимание: вызов реализации по умолчанию

И снова мы начинаем с предупреждения. Обязательно должна вызываться реализация по умолчанию метода `postRestart`, потому что она вы-

зывает функцию `preStart`. Вызов `super.postRestart` можно опустить, если вы точно не желаете, чтобы вызывался метод `preStart` при перезапуске; в большинстве случаев это не так. По умолчанию в процессе перезапуска вызываются методы `preStart` и `postStop`, они также вызываются по событиям жизненного цикла `start` и `stop`, поэтому код инициализации и освобождения ресурсов имеет смысл добавлять в них, убив двух зайцев одним выстрелом.

В аргументе `reason` точно так же передается причина перезапуска, как в методе `preRestart`. Переопределив метод, можно восстановить любое последнее состояние актора, например используя информацию, сохраненную функцией `preRestart`.

#### 4.2.4. Объединяем фрагменты жизненного цикла вместе

Объединив разные события воедино, получаем полный жизненный цикл актора, как показано на рис. 4.12. В данном случае показано только одно событие `restart`.

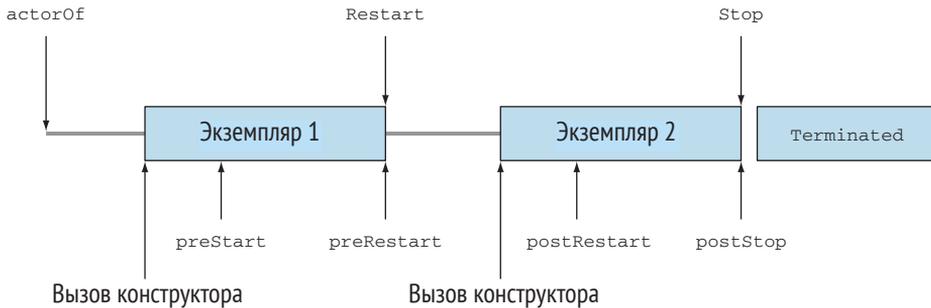


Рис. 4.12. Полный жизненный цикл актора

Добавив в актор все обработчики событий жизненного цикла, можно увидеть, в какой последовательности они возникают.

#### Листинг 4.5. Примеры обработчиков событий жизненного цикла

```
class LifeCycleHooks extends Actor
  with ActorLogging{
  System.out.println("Constructor")

  override def preStart(): Unit = {
    println("preStart")
  }
  override def postStop(): Unit = {
    println("postStop")
  }
  override def preRestart(reason: Throwable, message: Option[Any]): Unit = {
```

```

    println("preRestart")
    super.preRestart (reason, message)
  }
  override def postRestart(reason: Throwable): Unit = {
    println("postRestart")
    super.postRestart(reason)
  }
  def receive = {
    case "restart" =>
      throw new IllegalStateException("force restart")
    case msg: AnyRef =>
      println("Receive")
      sender() ! msg
  }
}

```

В листинге 4.6 приводится тест, генерирующий все три события жизненного цикла. Приостановка сразу после вызова `stop` позволяет увидеть происходящее в `postStop`.

#### Листинг 4.6. Тестирование событий жизненного цикла

```

val testActorRef = system.actorOf(           ← Запуск актора
  Props[LifeCycleHooks], "LifeCycleHooks")
testActorRef ! "restart"                   ← Перезапуск актора
testActorRef.tell("msg", testActor)
expectMsg("msg")
system.stop(testActorRef)                 ← Остановка актора
Thread.sleep(1000)

```

В листинге 4.7 приводится результат тестирования.

#### Листинг 4.7. Вывод из методов обработки событий жизненного цикла

```

Constructor           |→ Событие start
preStart
preRestart force restart
postStop
Constructor           |→ Событие restart
postRestart force restart
preStart
Receive
postStop              ← Событие stop

```

Все акторы проходят через этот жизненный цикл; запускаются, возможно, перезапускаются несколько раз и, наконец, останавливаются. Методы `preStart`, `preRestart`, `postRestart` и `postStop` позволяют актору инициализировать

зировать и сбрасывать состояние, а также управлять его восстановлением после аварии.

### 4.2.5. Мониторинг жизненного цикла

Жизненный цикл актора можно контролировать. Он завершается с остановкой актора. Актор завершается, когда супервизор решит остановить его, вызвав метод `stop` актора или пошлав ему сообщение `PoisonPill`, которое в конечном итоге приводит к вызову метода `stop`. Поскольку реализация по умолчанию метода `preRestart` останавливает все дочерние акторы вызовом их методов `stop`, потомки завершаются в случае перезапуска родителя. Перезапуск аварийного экземпляра актора не завершает его в этом смысле. Аварийный экземпляр удаляется из системы акторов, но не вызовом метода `stop`, ни прямо, ни косвенно. Это связано с тем, что ссылка `ActorRef` продолжит существовать после перезапуска; экземпляр актора не завершается, а замещается новым. `ActorContext` предоставляет метод `watch` для запуска мониторинга события остановки актора и метод `unwatch` для прекращения. Как только актор вызовет метод `watch` по ссылке на актор, он превращается в монитор этой ссылки. Когда контролируемый актор останавливается, он посылает актору-монитору сообщение `Terminated`. Сообщение `Terminated` содержит только ссылку `ActorRef` на остановившийся актор. Тот факт, что аварийный экземпляр актора не останавливается при перезапуске, в том смысле, как это происходит при остановке вызовом метода `stop`, становится понятным, потому что иначе вы получили бы массу сообщений `Terminated` при перезапуске акторов и не смогли бы отличить окончательную остановку актора от перезапуска. В листинге 4.8 демонстрируется актор `DbWatcher`, наблюдающий за жизненным циклом актора `dbWriter`.

**Листинг 4.8.** Наблюдение за жизненным циклом актора `dbWriter`

```
class DbWatcher(dbWriter: ActorRef) extends Actor with ActorLogging {
  context.watch(dbWriter)
  def receive = {
    case Terminated(actorRef) =>
      log.warning("Actor {} terminated", actorRef)
  }
}
```

Управление акторами может осуществляться только их супервизорами, или родителями, а мониторинг – любыми акторами. Если актор имеет доступ к ссылке `ActorRef` на другой актор, он может просто вызвать `context.watch(actorRef)` и после этого получить сообщение `Terminated`, когда указанный актор остановится. Мониторинг и управление могут совмещаться и

образовывать мощные средства управления, как будет показано в следующем разделе.

Мы пока не обсудили, как супервизор в действительности решает судьбу подчиненного актора – остановить его или перезапустить. Это станет темой следующего раздела, где мы подробно рассмотрим вопросы управления. В этом разделе мы впервые посмотрим, как супервизор создает свою иерархию, а затем исследуем стратегии, имеющиеся в распоряжении супервизора.

## 4.3. Супервизор

В этом разделе мы детально рассмотрим работу супервизора. Возьмем за основу пример приложения обработки файлов журналов и покажем разные стратегии управления, доступные супервизору. Основное внимание будет сосредоточено на иерархии супервизора в пути `/user`, который также называется *пространством пользователя* (`user space`). Именно здесь располагаются все прикладные акторы. Сначала мы рассмотрим разные способы определения иерархии супервизоров в приложении, а также достоинства и недостатки каждого. Затем посмотрим, как можно в разных супервизорах использовать разные стратегии.

### 4.3.1. Иерархия супервизора

Иерархия супервизора определяет порядок создания акторов друг другом: каждый актор, создающий другие акторы, является супервизором для созданных им дочерних акторов.

Иерархия остается фиксированной на протяжении всего жизненного цикла дочернего актора. После создания родителем дочерний актор находится под постоянным наблюдением родителя до самой остановки; в Акка нет такого понятия, как усыновление (или удочерение). Единственная возможность для супервизора отказаться от родительских обязанностей – остановить дочерние акторы. Поэтому так важно с самого начала выбрать правильную иерархию акторов в приложении, особенно если вы не планируете останавливать части иерархии для замены их совершенно другими поддеревьями акторов.

Акторы, наиболее подверженные опасностям (для которых наиболее вероятны аварии), должны находиться в иерархии как можно ниже. Ошибки, возникающие глубоко в иерархии, могут быть обработаны большим числом супервизоров, чем ошибки, возникающие ближе к вершине. Когда ошибка возникает на верхнем уровне системы акторов, она может вызвать перезапуск всех акторов верхнего уровня или даже остановить систему акторов.

Рассмотрим иерархию супервизора приложения обработки файлов журналов, как мы обещали в предыдущем разделе. Она изображена на рис. 4.7 в разделе 4.1.2.

В этой конфигурации `LogProcessingSupervisor` создает все акторы, составляющие приложение. Мы связали акторов друг с другом непосредственно, используя ссылки `ActorRef`. Каждому актору известна ссылка `ActorRef` на следующий актор, которому он посылает сообщения. Ссылки `ActorRef` всегда должны оставаться действительными и указывать на следующий экземпляр актора в цепочке. Если экземпляр актора когда-либо будет остановлен, его ссылка `ActorRef` будет указывать на системный актор `deadLetters`, что может нарушить работу приложения. Из-за этого во всех случаях супервизор должен выполнить перезапуск дочернего актора, потому что ссылка `ActorRef` остается действительной и может использоваться повторно.

Преимущество такого подхода заключается в том, что акторы общаются друг с другом непосредственно, а `LogProcessingSupervisor` только создает экземпляры и наблюдает за ними. Недостаток состоит в том, что мы можем выполнить только перезапуск, потому что иначе сообщения будут посылаться в `deadLetters` и теряться там. Кроме того, остановка `FileWatcher` в случае ошибки `DiskError` не вызовет остановки `LogProcessor` или `DbWriter`, потому что они не являются потомками `FileWatcher`. Например, нам может понадобиться остановить `DbWriter` и создать новый экземпляр, чтобы изменить URL базы данных, если благодаря исключению `DbNodeDownException` выяснилось, что текущий узел базы данных остановился. Для создания `DbWriter` в методе `Restart` используется оригинальный объект `Props`, который всегда ссылается на тот же самый URL базы данных. То есть в этой ситуации нам нужно другое решение.

На рис. 4.13 представлен иной подход. В этой версии `LogProcessingSupervisor` не создает всех акторов; `FileWatcher` создает `LogProcessor`, а `LogProcessor` создает `DbWriter`.

Потоки нормальной работы и восстановления после ошибки все еще определяются отдельно, даже притом, что теперь `FileWatcher` и `LogProcessor` создают дочерние акторы и управляют ими, а также обрабатывают обычные сообщения.

Преимущество этого подхода заключается в том, что теперь `LogProcessor` может наблюдать за `DbWriter`, остановить его, когда возникнет исключение `DbNodeException`, и создать новый экземпляр `DbWriter` с альтернативным адресом URL совершенно другого узла базы данных после получения сообщения `Terminated`.

Кроме того, `LogProcessingSupervisor` теперь не должен осуществлять управление всем приложением, он контролирует только работу актора `FileWatcher`. Если бы `LogProcessingSupervisor` наблюдал за обоими акторами – `FileWatcher` и `DbWriter`, ему пришлось бы различать события `Terminated`, посылаемые ими, что ухудшило бы изоляцию кода, где принимаются решения, связанные с обработкой проблем в подчиненных ком-

понентах. В исходном коде примеров на GitHub вы найдете еще несколько примеров стратегий работы супервизоров. В листинге 4.9 показано, как конструируется иерархия, изображенная на рис. 4.13. В следующем разделе мы подробнее рассмотрим супервизоры и используемые ими стратегии.

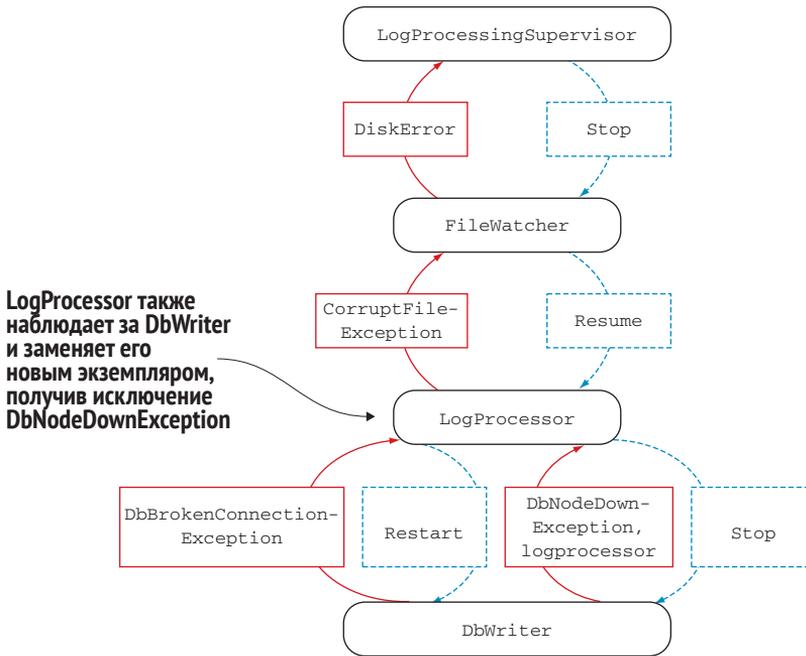


Рис. 4.13. Каждый актер создает дочерние акторы и управляет ими

#### Листинг 4.9. Создание иерархии супервизоров

```
object LogProcessingApp extends App {
  val sources = Vector("file:///source1/", "file:///source2/")
  val system = ActorSystem("logprocessing")
  val databaseUrls = Vector(
    "http://mydatabase1",
    "http://mydatabase2",
    "http://mydatabase3"
  )
  system.actorOf(
    LogProcessingSupervisor.props(sources, databaseUrls),
    LogProcessingSupervisor.name
  )
}
```

Первый URL - начальный; остальные используются в случае исключения DbNodeDownException

В листинге 4.9 можно видеть, как создается приложение обработки файлов журналов. В нем имеется только один актер верхнего уровня, LogPro-

cessingSupervisor, который создается вызовом метода `system.actorOf`. Все остальные акторы создаются на более низких уровнях иерархии. В следующем разделе мы исследуем каждый актор, и вы увидите, как они создают своих потомков.

Теперь, когда вы чуть больше узнали об иерархии супервизоров в приложении, перейдем к следующему разделу, где описываются разные стратегии управления.

### 4.3.2. Предопределенные стратегии

Акторы верхнего уровня в приложении создаются в пути `/user` и управляются *охранником пространства пользователя* (`user guardian`). По умолчанию охранник пространства пользователя перезапускает своих потомков по любому исключению, кроме случаев внутренних исключений, указывающих, что актор потерпел аварию в ходе инициализации, – в этой ситуации аварийный актор просто останавливается. Эта стратегия известна как *стратегия по умолчанию*. Каждый актор имеет свою стратегию супервизора по умолчанию, которую можно переопределить, реализовав метод `supervisorStrategy`. Объект `SupervisorStrategy` реализует две предопределенные стратегии: `defaultStrategy` и `stoppingStrategy`. Как следует из их названий, `defaultStrategy` – это стратегия по умолчанию для всех акторов; если не переопределить стратегию, актор всегда будет использовать стратегию по умолчанию. Стратегия по умолчанию определена в объекте `SupervisorStrategy`, как показано в листинге 4.10.

**Листинг 4.10.** Стратегия супервизора по умолчанию

```
final val defaultStrategy: SupervisorStrategy = {
  def defaultDecider: Decider = {
    case _: ActorInitializationException => Stop
    case _: ActorKilledException => Stop
    case _: Exception => Restart
  }
  OneForOneStrategy()(defaultDecider)
}
```

Выбор директивы зависит от полученного исключения

Stop, Restart, Resume и Escalate – это директивы

Стратегия `OneForOneStrategy` использует `defaultDecider`

Предыдущий код использует стратегию `OneForOneStrategy`, которая пока еще не обсуждалась. Акка позволяет принимать решения о судьбе дочерних акторов двумя способами: все дочерние акторы разделяют общую участь, и ко всем применяется одна стратегия восстановления, или решение относится только к аварийному актору. В некоторых случаях бывает желательно остановить только аварийный дочерний актор. В других может понадобиться остановить все дочерние акторы, если один из них потерпел аварию, например потому, что все они зависят от какого-то одного

ресурса. Если возникло исключение, сообщающее о недоступности общего ресурса, лучшим вариантом будет немедленно остановить все дочерние акторы, не дожидаясь, пока неприятности коснутся каждого из них. Стратегия `OneForOneStrategy` определяет, что все дочерние акторы имеют разную судьбу: решение касается только аварийного потомка. Другой вариант – стратегия `AllForOneStrategy`, которая применяет одно и то же решение ко всем дочерним акторам, даже если аварию потерпел только один из них. Более подробно `OneForOneStrategy` и `AllForOneStrategy` описываются в следующем разделе. В листинге 4.11 показано определение `stoppingStrategy` в объекте `SupervisorStrategy`.

#### Листинг 4.11. Стратегия остановки

```
final val stoppingStrategy: SupervisorStrategy = {
  def stoppingDecider: Decider = {
    case _: Exception => Stop      ← По любому исключению выполняется остановка
  }
  OneForOneStrategy()(stoppingDecider)
}
```

Стратегия `stoppingStrategy` останавливает любой дочерний актор по любому исключению. Эти встроенные стратегии не являются чем-то необычным. Они определяются точно так же, как вы определили бы свою собственную стратегию управления. Итак, что же произойдет, если родитель использует стратегию `stoppingStrategy` и его потомок возбudit исключение, такое как `ThreadDeath` или `OutOfMemoryError`? Любое исключение, не обработанное актором, передается родителю вверх по иерархии. Если фатальное исключение достигнет охранника пространства пользователя, он не сможет обработать его, потому что использует стратегию по умолчанию. В этой ситуации исключение достигнет обработчика в системе акторов и вызовет остановку всей системы. В большинстве случаев считается хорошей практикой не обрабатывать фатальные исключения в супервизорах, а позволить всей системе акторов аккуратно завершить работу, потому что восстановление нормальной работы после фатального сбоя просто невозможно.

### 4.3.3. Собственные стратегии

Каждое приложение должно предусматривать стратегии для всех случаев, требующих повышенной отказоустойчивости. Как вы видели в предыдущих разделах, супервизор может предпринять четыре разных действия, чтобы решить судьбу аварийного актора. Это строительные блоки, которые мы будем использовать. В данном разделе мы вновь вернемся к приложению обработки файлов журналов и определим конкретные стратегии:

- *возобновить* работу потомка, игнорировать ошибки и продолжить обработку сообщений тем же экземпляром актора;
- *перезапустить* потомка, удалить аварийный экземпляр и заменить его новым;
- *остановить* потомка и завершить его навсегда;
- *передать ошибку на уровень выше* и позволить родительскому актору решить, что делать.

Сначала рассмотрим исключения, которые могут возникнуть в приложении обработки файлов журналов. Чтобы не усложнять, определим лишь несколько своих исключений.

**Листинг 4.12.** Исключения в приложении обработки файлов журналов

```
@SerialVersionUID(1L)
class DiskError(msg: String)
  extends Error(msg) with Serializable
```

← Фатальная ошибка, возникающая в случае выхода из строя диска, где хранятся файлы журналов

```
@SerialVersionUID(1L)
class CorruptedFileException(msg: String, val file: File)
  extends Exception(msg) with Serializable
```

← Возникает, когда файл журнала поврежден и не может быть обработан

```
@SerialVersionUID(1L)
class DbNodeDownException(msg: String)
  extends Exception(msg) with Serializable
```

← Возникает, когда обнаруживается авария узла базы данных

Сообщения, которые акторы посылают друг другу, определены в объекте-компаньоне для соответствующего актора.

**Листинг 4.13.** Объект-компаньон LogProcessor

```
object LogProcessor {
  def props(databaseUrls: Vector[String]) =
    Props(new LogProcessor(databaseUrls))
  def name = s"log_processor_${UUID.randomUUID.toString}"
  // представляет новый файл журнала
  case class LogFile(file: File)
```

← Объект Props для создания LogProcessor

← Каждый экземпляр LogProcessor получает уникальное имя

← Файл журнала, получаемый экземпляром LogProcessor от FileWatcher для обработки

```
}
```

Начнем с нижнего конца иерархии и рассмотрим актор, осуществляющий запись в базу данных, который может получить исключение `DbBrokenConnectionException`. После этого исключения экземпляра `dbWriter` следует перезапустить.

**Листинг 4.14.** Актор DbWriter

```
object DbWriter {
  def props(databaseUrl: String) =
```

```

    Props(new DbWriter(databaseUrl))
    def name(databaseUrl: String) =
      s""db-writer-${databaseUrl.split("/").last}""
  }

  case class Line(time: Long, message: String, messageType: String)
}

class DbWriter(databaseUrl: String) extends Actor {
  val connection = new DbCon(databaseUrl)

  import DbWriter._
  def receive = {
    case Line(time, message, messageType) =>
      connection.write(Map('time -> time,
        'message -> message,
        'messageType -> messageType))
  }

  override def postStop(): Unit = {
    connection.close()
  }
}

```

Создание удобочитаемого имени

Строка из файла журнала, проанализированная актором LogProcessor

Попытка записи в соединение может завершиться аварией

Закрывает соединение, если актор потерпел аварию или был остановлен

За DbWriter наблюдает LogProcessor.

#### Листинг 4.15. LogProcessor наблюдает за DbWriter

```

class LogProcessor(databaseUrls: Vector[String])
  extends Actor with ActorLogging with LogParsing {
  require(databaseUrls.nonEmpty)

  val initialDatabaseUrl = databaseUrls.head
  val alternateDatabases = databaseUrls.tail

  override def supervisorStrategy = OneForOneStrategy() {
    case _: DbBrokenConnectionException => Restart
    case _: DbNodeDownException => Stop
  }

  var dbWriter = context.actorOf(
    DbWriter.props(initialDatabaseUrl),
    DbWriter.name(initialDatabaseUrl)
  )
  context.watch(dbWriter)

  import LogProcessor._
  def receive = {

```

Перезапустить, если повторная попытка соединиться может увенчаться успехом

Остановить, если провалились все попытки восстановить соединение

Создать дочерний актор dbWriter и наблюдать за ним

```

case LogFile(file) =>
  val lines: Vector[DbWriter.Line] = parse(file)
  lines.foreach(dbWriter ! _)
case Terminated(_) =>
  if(alternateDatabases.nonEmpty) {
    val newDatabaseUrl = alternateDatabases.head
    alternateDatabases = alternateDatabases.tail
    dbWriter = context.actorOf(
      DbWriter.props(newDatabaseUrl),
      DbWriter.name(newDatabaseUrl)
    )
    context.watch(dbWriter)
  } else {
    log.error("All Db nodes broken, stopping.")
    self ! PoisonPill
  }
}
}

```

← Послать строки в dbWriter

← Если dbWriter завершился, создать новый dbWriter с альтернативным URL и наблюдать за ним

← Все альтернативы опробованы; остановить LogProcessor

В случае разрыва соединения с базой данных из объекта `Props` создается новый экземпляр `DbWriter`, который установит новое соединение в своем конструкторе, выбрав новый URL из `databaseUrl`.

Замена экземпляра `dbWriter` происходит при появлении исключения `DbNodeDownException`. Если все альтернативы исчерпаны, `LogProcessor` останавливается, посылая себе сообщение `PoisonPill`. Строка, обрабатывавшаяся актором в момент появления исключения `DbBrokenConnectionEx-ception`, теряется. Мы рассмотрим решение данной проблемы далее в этом разделе. Выше в иерархии акторов находится актор `LogProcessor`.

Актор `LogProcessor` терпит аварию, когда обнаруживает поврежденный файл. В этом случае не имеет смысла продолжать попытки обработать данный файл; поэтому мы его пропускаем. Актор `FileWatcher` возобновляет работу аварийного актора.

#### Листинг 4.16. FileWatcher наблюдает за LogProcessor

```

class FileWatcher(source: String,
                  databaseUrls: Vector[String])
  extends Actor with ActorLogging with FileWatchingAbilities {
  register(source)
  override def supervisorStrategy = OneForOneStrategy() {
    case _: CorruptedFileException => Resume
  }
  val logProcessor = context.actorOf(
    LogProcessor.props(databaseUrls),

```

← Зарегистрировать исходный URI в API наблюдения за файлами

← Возобновить, если обнаружен поврежденный файл

```

    LogProcessor.name
  )
  context.watch(logProcessor)

import FileWatcher._

def receive = {
  case NewFile(file, _) =>
    logProcessor ! LogProcessor.LogFile(file)
  case SourceAbandoned(uri) if uri == source =>
    log.info(s"$uri abandoned, stopping file watcher.")
    self ! PoisonPill
  case Terminated(`logProcessor`) =>
    log.info(s"Log processor terminated, stopping file watcher.")
    self ! PoisonPill
}

```

Создать LogProcessor и наблюдать за ним

Посылается из API наблюдения за файлами, когда обнаруживается новый файл

FileWatcher останавливает себя после отключения источника, сообщая API наблюдения за файлами, что в данном источнике не нужно больше искать новые файлы

FileWatcher должен остановиться, если LogProcessor остановился из-за исчерпания альтернативных URL баз данных в DbWriter

Мы не будем вдаваться в детали реализации API наблюдения за файлами; гипотетически его предоставляет трейт `FileWatchingAbilities`. Актор `FileWatcher` не выполняет никаких опасных действий и будет продолжать работать, пока API наблюдения за файлами не уведомит его, что данный источник файлов не может дальше использоваться. Супервизор `LogProcessingSupervisor` следит за завершением акторов `FileWatcher` и также обрабатывает ошибку `DiskError`, которая может произойти в любой точке в иерархии. Так как ошибка `DiskError` не определена далее в иерархии, она автоматически будет подниматься вверх, от потомка к родителю. Это фатальная ошибка, поэтому в ответ на нее `FileWatchingSupervisor` решает остановить все акторы в иерархии. Стратегия `AllForOneStrategy` гарантирует остановку всех экземпляров `FileWatcher`, если любой из них потерпит аварию с ошибкой `DiskError`.

#### Листинг 4.17. LogProcessingSupervisor

```

object LogProcessingSupervisor {
  def props(sources: Vector[String], databaseUrls: Vector[String]) =
    Props(new LogProcessingSupervisor(sources, databaseUrls))
  def name = "file-watcher-supervisor"
}

class LogProcessingSupervisor(
  sources: Vector[String],
  databaseUrls: Vector[String]
) extends Actor with ActorLogging {

  var fileWatchers: Vector[ActorRef] = sources.map { source =>

```

```

val fileWatcher = context.actorOf(
  Props(new FileWatcher(source, databaseUrls))
)
context.watch(fileWatcher)
fileWatcher
}

override def supervisorStrategy = AllForOneStrategy() {
  case _: DiskError => Stop
}

def receive = {
  case Terminated(fileWatcher) =>
    fileWatchers = fileWatchers.filterNot(_ == fileWatcher)
    if (fileWatchers.isEmpty) {
      log.info("Shutting down, all file watchers have failed.")
      context.system.terminate()
    }
}
}

```

← Наблюдать за всеми FileWatcher

← Остановить FileWatcher, если случилась ошибка DiskError. LogProcessor и DbWriter, находящиеся ниже в иерархии, также будут остановлены автоматически

← Получено сообщение Terminated для FileWatcher

← После остановки всех экземпляров FileWatcher остановить систему акторов и тем самым завершить приложение

По умолчанию стратегии `OneForOneStrategy` и `AllForOneStrategy` действуют до бесконечности. Обе имеют значения по умолчанию для аргументов `maxNrOfRetries` и `withinTimeRange` конструктора. Но иногда бывает желательно прекратить действие стратегии после некоторого числа попыток или по прошествии определенного времени. В таких случаях просто передайте в этих аргументах требуемые значения. После настройки ограничений, если проблема повторится в течение заданного времени или после предельного количества попыток, ошибка будет передана вверх по иерархии. В листинге 4.18 демонстрируется пример «нетерпеливой» стратегии супервизора базы данных.

**Листинг 4.18.** Стратегия нетерпеливого супервизора базы данных

```

override def supervisorStrategy = OneForOneStrategy(
  maxNrOfRetries = 5,
  withinTimeRange = 60 seconds) {
  case _: DbBrokenConnectionException => Restart
}

```

← Передаст ошибку вверх по иерархии, если проблема повторится в течение 60 секунд или после пяти попыток перезапуска

**ПРИМЕЧАНИЕ.** Важно отметить, что между перезапусками задержка не выполняется; актор будет перезапускаться настолько быстро, насколько это возможно. Если перед перезапуском актора требуется выполнить задержку, используйте специальный актор `BackOffSupervisor`, которому можно передать объект `Props` своего актора. Актор `BackOffSupervisor` создаст экземпляр актора, используя объект `Props`, и будет управлять им как супервизор.

Этот супервизор поддерживает возможность выполнения задержки для предотвращения слишком быстрого перезапуска.

Этот механизм можно использовать для предотвращения непрекращающихся перезапусков актора, не дающих положительных результатов. Обычно он используется вместе с поддержкой наблюдения для реализации стратегии в ответ на остановку наблюдаемого актора; например, чтобы попробовать создать актор снова, спустя какое-то время.

## 4.4. В заключение

Отказоустойчивость является одним из интереснейших аспектов Акка и важнейшим компонентом подхода к организации конкурентного выполнения. Философия «и пусть падает» не призывает игнорировать возможные нарушения в работе и не является панацеей от всех бед. Скорее наоборот: программист должен предусмотреть все требования к восстановлению нормальной работы, но ему в руки даются непревзойденные инструменты, позволяющие реализовать эти требования без необходимости писать тонны кода. В процессе увеличения отказоустойчивости нашего приложения обработки журналов мы выяснили, что:

- иерархия супервизоров позволяет ясно отделить код, отвечающий за восстановление после ошибки;
- благодаря тому что модель акторов основана на обмене сообщениями, сохраняется возможность продолжить работу даже после выхода актора из строя;
- можно возобновить, остановить, перезапустить актор по своему выбору, учитывая конкретные требования в каждом случае;
- можно даже позволить ошибке подняться вверх по иерархии супервизоров.

И снова философия Акка предстала перед нами во всем своем блеске: она помогла воплотить наши потребности в код, но уже структурированным способом, предоставив все необходимые инструменты. В результате у нас получилось реализовать и протестировать довольно сложные требования к отказоустойчивости, не прилагая сколько-нибудь серьезных усилий.

Теперь, зная, как фреймворк Акка может помочь реализовать функциональные возможности конкурентной системы с использованием акторов и обрабатывать ошибки, возникающие в них, вы можете начать создавать мощные, отказоустойчивые приложения. В следующих главах мы создадим несколько разных приложений на основе акторов и посмотрим, как реализовать службы поддержки конфигурации, журналирования и развертывания.

# Глава 5

## Объекты Future

В этой главе:

- использование объектов Future;
- объединение объектов Future;
- восстановление после ошибок внутри объектов Future;
- объединение объектов Future с акторами.

В этой главе мы познакомимся с объектами Future. Это чрезвычайно удобный и простой инструмент объединения асинхронных функций. Фреймворк Akka изначально предоставлял свою реализацию объектов Future. Однако есть ряд других библиотек, также реализующих тип Future, в том числе *Twitter Finagle* и *scalaz*. Доказав свою полезность, пакет *scala.concurrent* прошел процедуру реорганизации Scala Improvement Process (SIP-14), и теперь, начиная с версии Scala 2.10, в него включен тип Future как основа стандартной библиотеки Scala.

Так же как акторы, объекты Future являются важными строительными блоками для организации асинхронного и параллельного выполнения кода. И акторы, и объекты Future – отличные инструменты, которыми можно пользоваться в разных ситуациях. Речь идет о выборе правильного инструмента для разных задач. Сначала, в разделе 5.1 «Примеры использования объектов Future», мы опишем случаи, для которых объекты Future подходят лучше всего, и разберем некоторые примеры. В отличие от акторов, реализующих механизм для создания систем, состоящих из конкурирующих *объектов*, объекты Future предоставляют механизм для конструирования систем из асинхронных *функций*.

Объекты Future позволяют организовать обработку результата функции в текущем потоке, не ожидая, когда она вернет его. Как это реализовать, будет описано в разделе 5.2. Основное внимание мы сосредоточим на изучении примеров использования объектов Future, и не будем глубоко

погружаться в детали абстракции типа `Future[T]`. Объекты `Future` можно свободно *объединять* друг с другом множеством способов. В разделе 5.4 вы узнаете, как можно таким способом конструировать последовательности асинхронных вызовов веб-служб, а в разделе 5.3 – как обрабатывать ошибки.

Вам не обязательно выбирать между объектами `Future` и акторами; они прекрасно уживаются вместе. Фреймворк Akka поддерживает обобщенные шаблоны объединения объектов `Future` с акторами, которые упрощают с обоими механизмами, о чем подробно рассказывается в разделе 5.5.

## 5.1. Примеры использования объектов Future

В предыдущих главах вы многое узнали об акторах.

Чтобы вам было проще понять, когда лучше использовать объекты `Future`, мы представим несколько решений на основе акторов, которые не лишены некоторой сложности. Для этих задач объекты `Future` предлагают намного более простые решения. Акторы отлично подходят для обработки сообщений, сохранения состояния и реагирования соответственно состоянию и принятому сообщению. Они очень устойчивы к ошибкам и способны действовать довольно долго, при условии надлежащей реализации мониторинга и управления ими.

Объекты `Future` – это инструмент, лучше подходящий для случаев, когда предпочтительнее использовать *функции* и нет необходимости хранить какое-либо состояние.

Объект `Future` служит своеобразным хранилищем результата функции (успех или отказ), который будет доступен некогда в будущем (*future*). Это эффективное средство асинхронной обработки результатов, дающее возможность сослаться на результат, который будет доступен рано или поздно. Данную идею поясняет рис. 5.1.

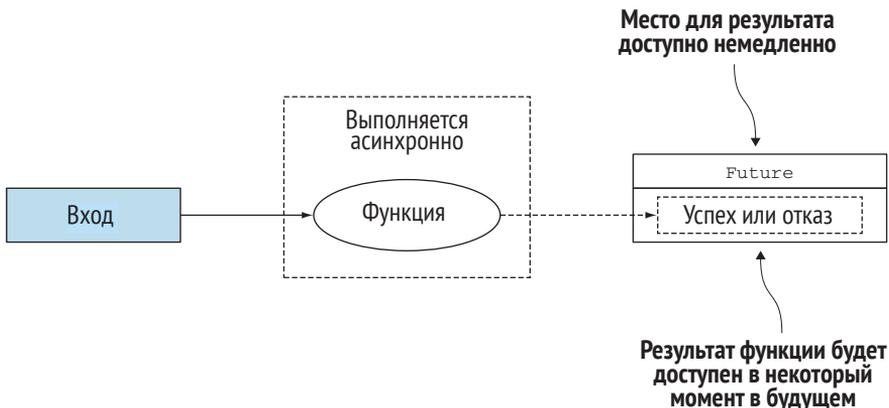


Рис. 5.1. Хранилище для результата функции, выполняемой асинхронно

Объект `Future` – это место для результата, доступное только для чтения. Его нельзя изменить извне. Он будет хранить сам результат в случае успеха или признак отказа сразу по завершении функции. После завершения результат внутри `Future` нельзя изменить, но можно прочитать много раз; он всегда будет одним и тем же. Резервирование места для результата упрощает объединение нескольких функций, выполняющихся асинхронно. Вы можете просто сказать, что делать с ним, как только он появится, как будет показано в следующих разделах. Например, можно послать запрос веб-службе, не блокируя работу текущего потока.

**ЭТО НЕ ПРОСТОЙ ОБЪЕКТ FUTURE ИЗ JAVA.** Знакомые с классом `java.util.concurrent.Future` в Java 7 могли бы по ошибке подумать, что класс `scala.concurrent.Future`, обсуждаемый в этой главе, является лишь оберткой вокруг этого Java-класса. Однако это не так. Класс `java.util.concurrent.Future` требует выполнять опрос и дает возможность получить результат только вызовом блокирующего метода `get`, тогда как класс `Future` в Scala позволяет получать результаты функций без опросов и блокирования, как вы узнаете далее в этой главе. Класс `CompletableFuture<T>`, появившийся в Java 8 (уже после появления `Future[T]` в Scala), уже ближе.

Чтобы лучше понять суть, рассмотрим еще один вариант системы по продаже билетов. Нам хотелось бы создать веб-страницу с дополнительной информацией о мероприятии и его месте проведения. Билет мог бы содержать ссылку на эту страницу, чтобы клиент смог посетить ее, например с помощью своего мобильного устройства. Также у нас может появиться желание показать прогноз погоды в месте проведения, если мероприятие проходит на открытом воздухе, маршрут до места проведения (чтобы клиент мог определить, как лучше поступить – воспользоваться общественным транспортом или поехать на своей машине), места парковок или предложить посетить похожие мероприятия в будущем.

Объекты `Future` особенно удобны для составления конвейеров, в которых одна функция готовит входные данные для следующей, или для параллельного запуска нескольких функций с последующим объединением их результатов. Служба `TicketInfo` будет искать связанную информацию по номеру билета. Любая из вспомогательных служб, возвращающих дополнительную информацию, может отказать в любой момент, и нам не хотелось бы, чтобы эти отказы заблокировали остальные запросы к другим службам во время сбора информации. Но не волнуйтесь, мы начнем с простых примеров. На рис. 5.2 показана конечная цель, к которой мы будем стремиться в этой главе.

Если служба не ответила в течение отведенного времени или попытка связаться с ней потерпела неудачу, соответствующая информация не

должна отображаться. Чтобы построить маршрут к месту проведения мероприятия, мы сначала должны найти его по номеру билета, как показано на рис. 5.3.

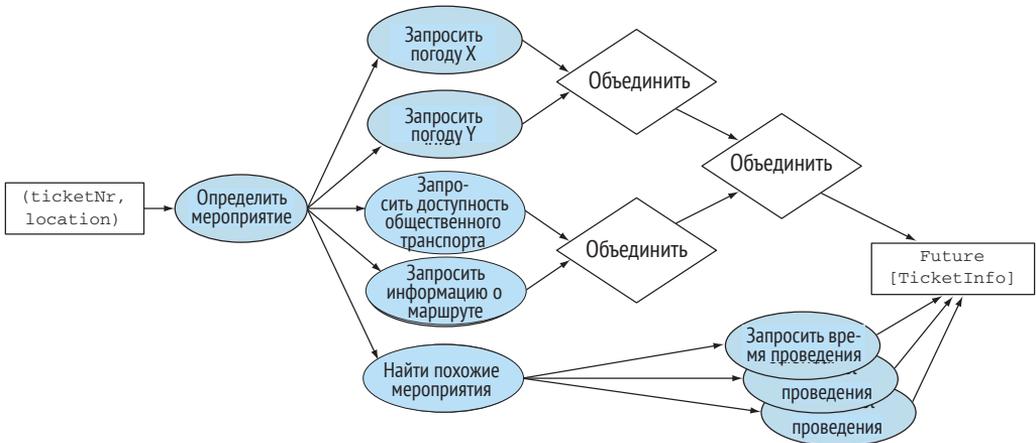


Рис. 5.2. Процесс работы TicketInfoService

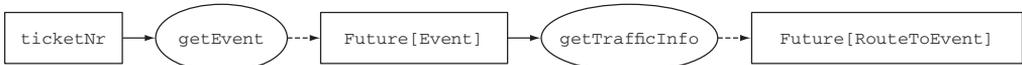


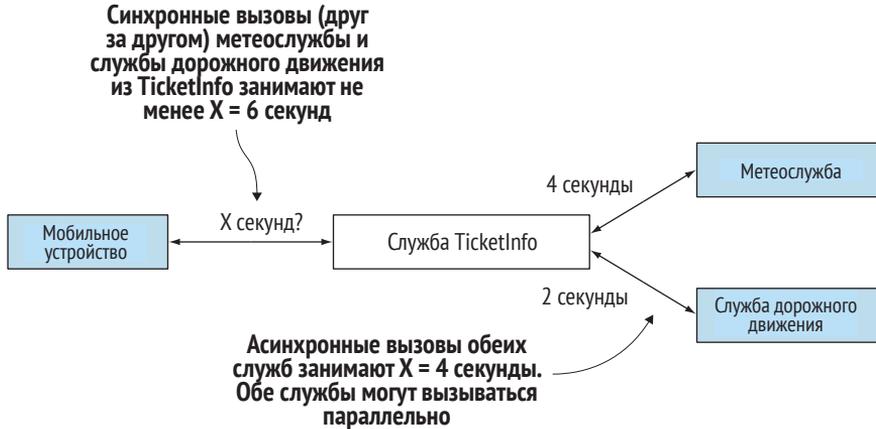
Рис. 5.3. Цепочка асинхронных функций

В данном случае `getEvent` и `getTrafficInfo` – это функции, асинхронно обращающиеся к веб-службам и выполняющиеся друг за другом. Функция `getTrafficInfo` принимает аргумент `Event` и вызывается, когда информация о мероприятии становится доступна в `Future[Event]`. Это сильно отличается от ситуации, когда сначала вызывается метод `getEvent` и затем выполняется цикл проверки получения информации в текущем потоке. Мы просто определяем последовательность действий, согласно которому функция `getTrafficInfo` будет вызвана *рано или поздно*, без необходимости ждать появления информации в потоке выполнения. Функция будет выполнена сразу же, как только это станет возможно. Текущему потоку не придется ждать, пока завершится вызов веб-службы. Отказ от цикла ожидания, как вы понимаете, – это *благо*, потому что потоки выполнения должны заниматься чем-то полезным, а не простаивать в ожидании.

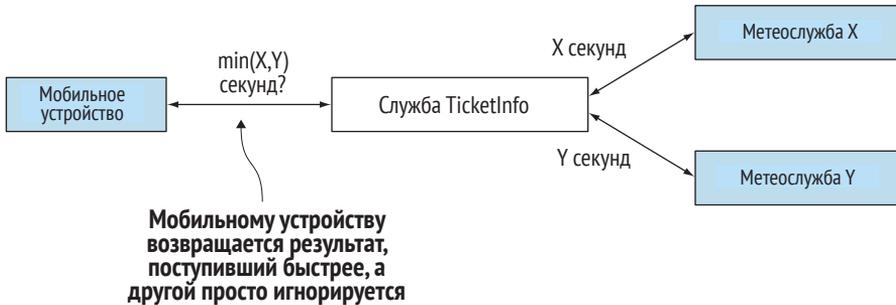
На рис. 5.4 показана простая ситуация, в которой асинхронное обращение к службе является идеальным решением. На нем изображено мобильное устройство, вызывающее службу `TicketInfo`, которая собирает информацию о погоде и маршруте к месту проведения мероприятия из других служб.

Отсутствие необходимости ждать, пока служба дорожного движения вернет ответ, перед вызовом службы погоды уменьшает общее время ожи-

дания ответа на запрос с мобильного устройства. Чем больше служб нужно будет вызвать, тем ощутимее получится эффект от параллельной обработки ответов. На рис. 5.5 показан другой случай. Здесь мы выбираем самый быстрый ответ, посылая запросы двум конкурирующим метеослужбам.



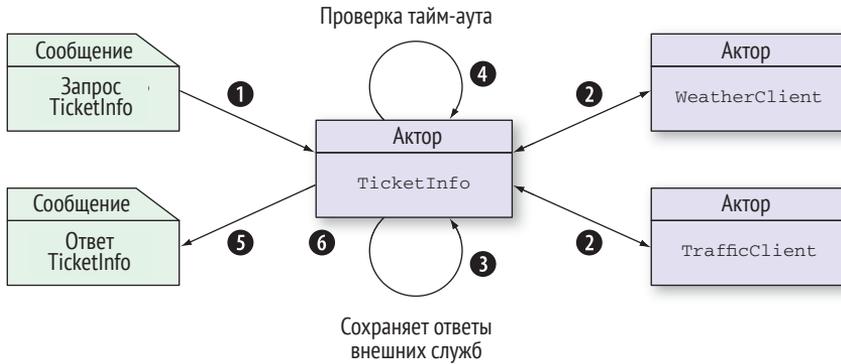
**Рис. 5.4.** Объединение результатов, получаемых синхронно и асинхронно



**Рис. 5.5.** Использование ответа, поступившего быстрее

Представьте, что метеослужба X не работает и не отвечает на запрос. Но это не заставит нас ждать истечения тайм-аута, потому что мы используем ответ от метеослужбы Y, которая работает и отвечает.

Нельзя сказать, что эти сценарии невозможно реализовать с применением акторов. Но придется потрудиться, чтобы воплотить в код такой простой случай. Возьмем для примера объединение результатов, возвращаемых метеослужбой и службой дорожного движения. Нам потребуется создать акторы, определить сообщения и реализовать функции receive в ActorSystem. Нам также придется подумать, как обрабатывать тайм-ауты, когда останавливать акторы, как создавать новые акторы для каждого запроса и как объединять ответы. На рис. 5.6 показано, как могла бы выглядеть реализация на основе акторов.



- ❶ Создать актор TicketInfo и послать запрос.
- ❷ Создать дочерние акторы и послать запросы; связать ответы с запросами.
- ❸ Сохранить ответы дочерних акторов.
- ❹ Послать актору TicketInfo сообщение об истечении тайм-аута, если один из дочерних акторов не ответил вовремя.
- ❺ Вернуть сообщение с собранной информацией, если истек тайм-аут или обе внешние службы ответили вовремя.
- ❻ Остановить актор TicketInfo и дочерние акторы после отправки ответа; только один раз на каждый запрос.

Рис. 5.6. Параллельное выполнение запросов к службам с использованием акторов

Нам понадобятся два отдельных актора, чтобы послать два параллельных запроса метеослужбе и службе дорожного движения. В `TicketInfoActor` придется предусмотреть свой порядок объединения ответов для каждого конкретного случая. Все это слишком сложно для простого вызова двух веб-служб и объединения результатов. Но обратите внимание, что акторы могут оказаться лучшим выбором, когда требуется иметь полный контроль над состоянием или когда необходимо следить за ходом выполнения операции и, возможно, перезапустить ее.

Итак, хотя акторы являются отличным инструментом, они – не единственное средство, позволяющее уйти от блокировок. В данном сценарии инструмент, предназначенный конкретно для объединения результатов функций, мог бы быть проще.

Предыдущий пример относится к случаям, когда объекты `Future` лучше справляются с заданием. Вообще говоря, они оказываются лучшим выбором, когда выполняется хотя бы одно из следующих условий:

- блокировка (ожидание в текущем потоке выполнения) не требуется для обработки результата, возвращаемого функцией;
- в данной точке в коде функция просто вызывается, а обработка результатов выполняется позже, где-то в другом месте;

- требуется вызвать сразу несколько асинхронных функций и объединить их результаты позже;
- требуется вызвать сразу несколько асинхронных функций, и достаточно получить результаты лишь некоторых из них, например тех, которые выполняются быстрее;
- требуется вернуть функцию и результат по умолчанию, если она возбудила исключение;
- требуется объединить несколько функций в конвейер, когда каждая последующая функция зависит от результатов предыдущих.

В следующих разделах мы рассмотрим детали реализации службы `TicketInfo` с использованием объектов `Future`. Начнем с простого асинхронного вызова одной веб-службы.

## 5.2. Объекты Future не блокируют выполнение потока

Создадим службу `TicketInfo`, которая не будет простаивать в ожидании ответа. Для этого попытаемся выполнить два шага, изображенных на рис. 5.7, и получить информацию о маршруте к месту проведения мероприятия.

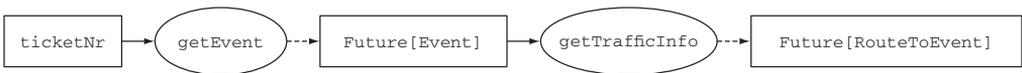


Рис. 5.7. Получение информации о маршруте к месту проведения мероприятия

Первый шаг – получение мероприятия по номеру билета. Синхронный и асинхронный вызовы функций оформляются в потоке выполнения программы совершенно по-разному. В листинге 5.1 показан пример синхронного вызова веб-службы, возвращающей мероприятие по номеру билета.

### Листинг 5.1. Синхронный вызов

```

val request = EventRequest(ticketNr)  ← Создать запрос
val response: EventResponse = callEventService(request)  ← Зabloкирует главный поток до получения ответа
val event: Event = response.event  ← Прочитать код мероприятия
  
```

В листинге 5.1 показаны три строки кода, выполняемые в некотором потоке. Порядок их выполнения прост: вызываемая функция должна вернуть значение в том же потоке. Очевидно, что программа не может продолжить работу в данном потоке, пока запрошенное значение не станет доступно. Выражения в языке `Scala` действуют строго (вычисляются немедленно), поэтому каждая строка в коде «создает законченное значение».

Теперь посмотрим, как можно изменить этот синхронный вызов веб-службы, чтобы сделать его асинхронным. В предыдущем примере `callEventService` выполняет блокирующий вызов веб-службы; она вынуждает поток ждать ответа. Чтобы организовать асинхронный вызов, сначала завернем `callEventService` в блок и вызовем ее в другом потоке, как показано в листинге 5.2.

### Листинг 5.2. Асинхронный вызов

```
val request = EventRequest(ticketNr)  ← Выполнится в потоке X
val futureEvent: Future[Event] = Future {  ← Блок кода будет вызван в
  val response = callEventService(request)  ← другом потоке (в потоке Y)
  response.event  ← Ответ с кодом мероприятия будет
}  ← доступен только в потоке Y
...  ← Ссылку futureEvent можно использовать в потоке X, например передавать
      ← ее в другие функции, но нельзя напрямую прочитать response.event
```

`Future { ... }` – это сокращенная форма вызова метода `apply` объекта `Future` с блоком кода в качестве аргумента, `Future.apply(блок_кода)`. Эта вспомогательная функция позволяет запустить «блок кода» в другом потоке и немедленно вернуть управление (такое возможно, потому что аргумент с блоком кода передается по имени, но об этом чуть позже). Блок кода возвращает значение `Event`, которое вычисляется только однажды.

Для тех, кто только начинает осваивать язык Scala, отмечу, что последнее выражение в блоке автоматически вернет значение. Метод `Future.apply` вернет объект `Future` с типом данных, который возвращает блок кода – в данном случае `Future[Event]`.

Тип ссылки `futureEvent` явно указан в данном примере, но аннотацию типа можно было бы опустить, потому что механизм автоматического определения типов в Scala смог бы определить его без нашей помощи. На протяжении этой главы мы везде будем добавлять аннотации типов, чтобы проще было анализировать примеры.

**АРГУМЕНТЫ МЕТОДА APPLY ОБЪЕКТА FUTURE.** Блок кода в вызове метода `Future.apply` передается как *параметр до востребования*. Параметр до востребования, вычисляется только в момент первого обращения к нему *внутри* функции. В случае с объектами `Future` это происходит в другом потоке. Блок кода в листинге 5.2 обращается к значению `request` уже в другом потоке (в этом примере мы назвали его потоком Y). Подобные ссылки на значения называют *замыканиями на значениях*, в данном случае замыкание происходит на значении `request`, которое играет роль моста между главным и другим потоком, из которого посылается запрос удаленной веб-службе.

Отлично! Теперь веб-служба вызывается в отдельном потоке, и мы можем обработать ответ прямо там. Давайте посмотрим, как можно добавить вызов `callTrafficService`, чтобы получить информацию о маршруте к месту проведения мероприятия (см. листинг 5.3). Для начала выведем полученную информацию в консоль.

**Листинг 5.3.** Обработка результата event

```
futureEvent.foreach { event =>
  val trafficRequest = TrafficRequest(
    destination = event.location,
    arrivalTime = event.time
  )
  val trafficResponse = callTrafficService(trafficRequest)
  println(trafficResponse.route)
}
```

← Асинхронная обработка мероприятия после получения

← Синхронный вызов службы управления дорожным движением с передачей мероприятия в запросе, возвращает TrafficResponse

← Вывод информации о маршруте в консоль

В листинге 5.3 используется метод `foreach` объекта `Future`, который вызывает блок кода с результатом `event`, когда тот станет доступным. Блок кода вызывается, только если вызов `callEventService` завершится успехом.

В данном случае так же предполагается, что `Route` будет использоваться позже, поэтому было бы хорошо иметь возможность вернуть `Future[Route]`. Но метод `foreach` возвращает значение типа `Unit`, поэтому нам придется придумать что-то еще. В листинге 5.4 показано, как проделать этот трюк с помощью метода `map`.

**Листинг 5.4.** Передача результата event по цепочке

```
val futureRoute: Future[Route] = futureEvent.map { event =>
  val trafficRequest = TrafficRequest(
    destination = event.location,
    arrivalTime = event.time
  )
  val trafficResponse = callTrafficService(trafficRequest)
  trafficResponse.route
}
```

← Обрабатывает event и возвращает Future[Route]

← Возвращает значение функции map, которая превратит его в Future[Route]

← Все так же синхронно вызывает функцию callTrafficService, которая возвращает ответ непосредственно

Оба метода, `foreach` и `map`, должны быть вам знакомы по опыту использования библиотеки `scala.collections` и стандартных типов, таких как `Option` и `List`. Концептуально метод `Future.map` имеет много общего, например, с `Option.map`. Как мы знаем, метод `Option.map` вызывает блок кода, если содержит значение, и возвращает новое значение `Option[T]`. Аналогично метод `Future.map` вызывает блок кода, если содержит благополучно вычисленный результат, и возвращает новое значение `Future[T]` – в данном случае `Future[Route]`, потому что последняя строка в блоке кода возвращает

значение `Route`. И снова мы явно определили тип `futureRoute`, чего можно было бы не делать. Код в листинге 5.5 показывает, как можно объединить в цепочку вызовы двух веб-служб.

**Листинг 5.5.** Метод `getRoute`, возвращающий результат `Future[Route]`

```
val request = EventRequest(ticketNr)

val futureRoute: Future[Route] = Future {
  callEventService(request).event
}.map { event =>                                     ← Объединение в цепочку с Future[Event]
  val trafficRequest = TrafficRequest(
    destination = event.location,
    arrivalTime = event.time
  )
  callTrafficService(trafficRequest).route         ← Вернуть маршрут
}
```

Если изменить метод `getEvent` так, чтобы он принимал `ticketNr`, а `getRoute` – аргумент `event`, цепочку с двумя вызовами можно было бы составить, как показано в листинге 5.6. Методы `getEvent` и `getRoute` возвращают `Future[Event]` и `Future[Route]` соответственно.

**Листинг 5.6.** Видоизмененная версия

```
val futureRoute: Future[Route] = getEvent(ticketNr).flatMap { event =>   ←
  getRoute(event)                                                         Здесь следует использовать flatMap; иначе
}                                                                           futureRoute вернет Future[Future[Route]]
```

В листинге 5.6 мы использовали `flatMap` для объединения `getEvent` и `getRoute`. Если бы мы использовали `map`, то в результате получили бы `Future[Future[Route]]`. При использовании `flatMap` вы должны возвращать `Future[T]`. (И снова в этом можно видеть сходство с `Option.flatMap`, например.)

Методы `callEventService` и `callTrafficService` в предыдущих примерах были реализованы как блокирующие, чтобы показать, как выполняется переход от синхронных вызовов к асинхронным. Чтобы действительно получить выгоды от асинхронного стиля программирования, предыдущие методы `getEvent` и `getRoute` следует реализовать с использованием неблокирующего API ввода/вывода и возвращать объекты `Future` непосредственно, чтобы минимизировать количество блокирующихся потоков выполнения. В модуле `akka-http` имеется асинхронный клиент HTTP. В следующих разделах мы будем предполагать, что вызовы веб-служб реализованы с использованием средств из `akka-http`.

До сих пор не указывалось, что для использования объектов `Future` необходимо предоставить доступ к неявному контексту выполнения `Execu-`

tionContext. Если этого не сделать, ваш код просто не будет компилироваться. В листинге 5.7 показано, как импортировать неявное значение для глобального контекста выполнения.

**Листинг 5.7.** Импорт неявного значения для глобального контекста выполнения

```
import scala.concurrent.Implicits.global ← Использовать глобальный контекст ExecutionContext
```

ExecutionContext – это абстракция некоторого пула потоков для выполнения задач. Если вы знакомы с пакетом `java.util.concurrent`, можете считать ExecutionContext некоторым подобием интерфейса `java.util.concurrent.Executor` с расширенными возможностями.

Инструкция `import` в листинге 5.7 помещает *глобальный контекст выполнения* в неявную область видимости и делает его доступным для объектов Future, которые получают возможность запускать блоки кода в некотором потоке.

В разделе 5.5 вы узнаете, что диспетчер системы акторов также можно использовать в роли ExecutionContext, что намного лучше, чем использование глобального контекста, потому что заранее неизвестно, сколько других процессов использует этот глобальный контекст.

В следующем разделе мы познакомимся с типом Promise[T]. При наличии возможности использовать API, возвращающие объекты Future, вы едва ли будете часто обращаться к типу Promise[T]. То есть вы можете сейчас пропустить этот короткий раздел (и вернуться к нему потом) и сразу перейти к разделу, где описывается, как обрабатывать ошибки.

### 5.2.1. Объекты Promise – это обещания

Если объекты Future доступны только для чтения, тогда куда записывается результат? А запись выполняется в объект Promise[T]. Если заглянуть в исходный код реализации по умолчанию объекта Future[T], можно увидеть, что внутренне он состоит из двух частей: части Future, доступной только для чтения, и части Promise. Они – как две стороны одной монеты.

Исходный код с реализацией Promise и Future содержит массу сложностей, связанных с косвенными ссылками, исследованием которых вы можете заняться самостоятельно, если действительно желаете знать все низкоуровневые тонкости.

Самый простой способ узнать, как работает объект Promise, – рассмотреть пример его использования. Объект Promise[T] можно использовать для превращения имеющегося многопоточного API с обратными вызовами в API, возвращающий Future[T]. Рассмотрим небольшой блок кода, посылающий записи в *Apache Kafka*. Если не углубляться в детали, кластер Kafka позволяет сохранять записи в журнал, доступный только для добавления в конец. С целью улучшения масштабируемости и надежности

журнал разбивается на разделы и хранится на нескольких серверах, которые называют *брокерами*. Самое важное для нас в этом примере, что `KafkaProducer` может асинхронно посылать записи брокерам Kafka. Объект `KafkaProducer` имеет метод `send`, принимающий аргумент с функцией обратного вызова. Эта функция будет вызвана точно один раз после благополучной отправки записи в кластер. Листинг 5.8 демонстрирует, как можно использовать `Promise` для обертывания этого метода с обратным вызовом и вернуть объект `Future`.

**Листинг 5.8.** Использование `Promise` для создания API, возвращающего `Future`

```
def sendToKafka(record: ProducerRecord): Future[RecordMetadata] = {
  val promise: Promise[[RecordMetadata]] = Promise[RecordMetadata]()
  val future: Future[RecordMetadata] = promise.future

  val callback = new Callback() {
    def onComplete(metadata: RecordMetadata, e: Exception): Unit = {
      if (e != null) promise.failure(e)
      else promise.success(metadata)
    }
  }

  producer.send(record, callback)
  future
}
```

Создание `Promise` с типом ожидаемого результата, `RecordMetadata`

Запись признака ошибки в `Promise`, если произошла ошибка

Это обратный вызов, посредством которого Kafka сообщает о завершении передачи записи. Вызывается один раз после передачи записи в другом потоке

Запись признака успеха в `Promise`

Выполняет фактическую отправку и передает обратный вызов

Получить ссылку на объект `Future[RecordMetadata]`, который мы сможем вернуть

Вернуть объект `Future` в код, вызвавший метод `sendToKafka`

И снова для ясности в пример добавлены объявления типов. Запись в объект `Promise` можно выполнить только один раз. Вызовы `promise.success(metadata)` и `promise.failure(e)` являются сокращенными версиями вызовов `promise.complete(Success(metadata))` и `promise.complete(Failure(e))` соответственно. Повторная попытка записи в объект `Promise` возбудит исключение `IllegalStateException`.

В этом простом примере нет ничего особенно примечательного, кроме получения ссылки на объект `Future` и записи результата в `Promise`. В более сложных сценариях может потребоваться убедиться в возможности безопасно использовать другие структуры данных в многопоточном контексте. Исходный код с реализацией `Promise` и `Future` может служить отличным справочником для этой цели.

Теперь, когда вы узнали, как можно завернуть обратный вызов в объект `Promise`, углубимся чуть дальше и посмотрим, как фактически действуют `Promise` и `Future`. Если вам это неинтересно, можете смело переходить к

следующему разделу. На рис. 5.8 показано, как метод `Future.apply` создает объект `Promise` и возвращает объект `Future` в потоке X.

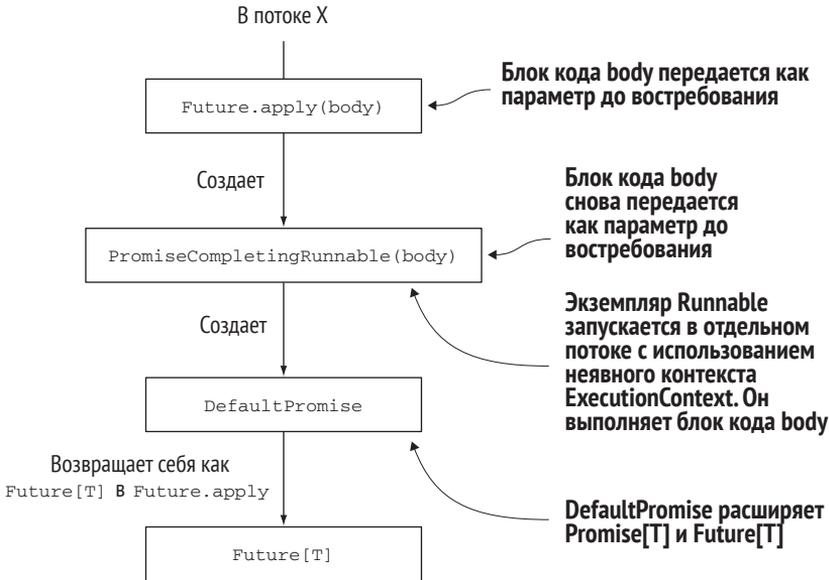


Рис. 5.8. Создание объектов `Promise` и `Future`

Здесь мы опустим некоторые подробности, но на рис. 5.8 видно, что `Future.apply` создает экземпляр подкласса `Runnable`. Экземпляр `Runnable` хранит объект `Promise`, который он сможет использовать, когда будет запущен в другом потоке. Тот же объект `Promise` возвращается как `Future` из `Future.apply`. И снова мы опустим некоторые детали, главное, на что нужно обратить внимание: `DefaultPromise[T]` наследует `Future[T]` и `Promise[T]`, поэтому может действовать «как оба типа сразу».

Здесь важно отметить, что `Runnable` и код, вызвавший `Future.apply`, получают ссылку на одно и то же значение, `DefaultPromise`. `DefaultPromise` поддерживает возможность использования в многопоточной среде, поэтому является потокобезопасным типом. На рис. 5.9 показано, что случится, когда `PromiseCompletingRunnable` будет запущен в другом потоке, который здесь мы назвали потоком Y.

`PromiseCompletingRunnable` выполняет запись в `Promise` в точности, как в примере с Kafka, что приводит к запуску всех зарегистрированных обратных вызовов с конечным результатом в `body`. Обратные вызовы запускаются только один раз, а реализация `Future` и `Promise` гарантирует правильное выполнение с применением низкоуровневых приемов конкурентного программирования. Дальнейшее исследование деталей реализации мы оставляем читателям в качестве самостоятельного упражнения.

Как обещалось, в следующем разделе будет показано, как реализовать восстановление нормальной работы после ошибок.

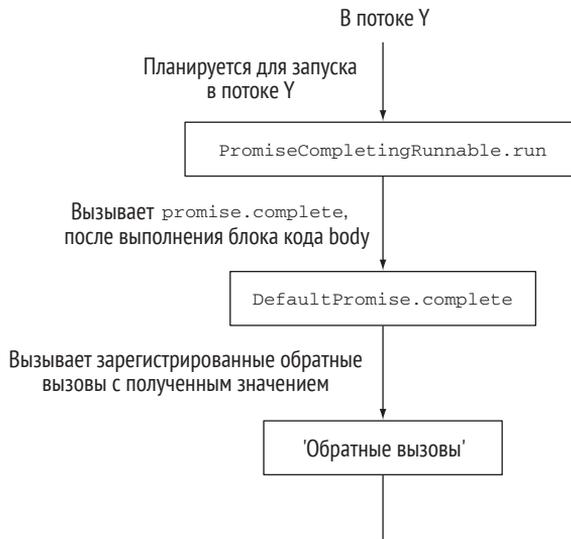


Рис. 5.9. Запись значения в Promise

## 5.3. Обработка ошибок в объектах Future

В предыдущем разделе предполагалось, что вычисления в объекте `Future` всегда завершаются успехом. А теперь давайте посмотрим, что произойдет, если блок кода возбудит исключение `Exception`. Для иллюстрации мы просто возбудим его принудительно. Вызовем метод `foreach` объекта `Future` и выведем результат. Запустите интерактивную оболочку Scala REPL в терминале и следуйте за примером, представленным в листинге 5.9.

### Листинг 5.9. Возбуждение исключения в объекте Future

```

scala> :paste
// Вход в режим вставки (выход комбинацией ctrl-D)

import scala.concurrent._
import ExecutionContext.Implicits.global

val futureFail = Future { throw new Exception("error!") }
futureFail.foreach(value => println(value))

// выход из режима вставки в режим интерпретации.

futureFail: scala.concurrent.Future[Nothing] =
  scala.concurrent.impl.Promise$DefaultPromise@193cd8e1
scala>

```

← Попытка вывести значение сразу после завершения Future

← Ничего не появилось, потому что возникло исключение

Исключение `Exception` возбуждается в другом потоке. Первое, что сразу бросается в глаза, – это отсутствие трассировки стека в консоли, которая появляется, только если исключение возникло в главном потоке REPL. Блок `foreach` не выполнялся. Это объясняется тем, что объект `Future` завершился с ошибочным значением. Получить исключение, ставшее причиной неудачи, можно вызовом метода `onComplete`. Этот метод также принимает блок кода, подобно методам `foreach` и `map`, но уже в виде аргумента `scala.util.Try`. Объект типа `Try` может принимать значения `Success` и `Failure`. В листинге 5.10 представлен сеанс REPL, показывающий, как можно вывести информацию об исключении.

**Листинг 5.10.** Использование метода `onComplete` для обработки успеха или неудачи

```
scala> :paste
// Вход в режим вставки (выход комбинацией ctrl+D)

import scala.util._
import scala.concurrent._
import ExecutionContext.Implicits.global

val futureFail = Future { throw new Exception("error!") }
futureFail.onComplete {
  case Success(value) => println(value)
  case Failure(e) => println(e)
}

// выход из режима вставки в режим интерпретации.

java.lang.Exception: error!
```

← Импортирует определение Try, Success и Failure

← Блок передается как значение типа Try. Try поддерживает сопоставление с образцом, поэтому можно просто передать методу `onComplete` частично определенную функцию, которая производит сопоставление с `Success` и `Failure`

← Вывод сообщения об ошибке в случае нефатального исключения

← Вывод значения в случае успешного завершения

← Пример вывел сообщение об ошибке

Метод `onComplete` позволяет обрабатывать результаты и в случае успеха, и в случае ошибки. Обратите внимание, что в этом примере обратный вызов `onComplete` выполнится, даже если объект `Future` уже завершился, что в данном случае вполне возможно, потому что исключение вызывается в блоке кода немедленно. Это также верно для всех функций, которые регистрируются в объекте `Future`.

**ФАТАЛЬНЫЕ И НЕФАТАЛЬНЫЕ ИСКЛЮЧЕНИЯ.** Объекты `Future` никогда не обрабатывают фатальные исключения. Попытавшись создать объект `Future{ new OutOfMemoryError("arghh") }`, вы увидите, что этот объект вообще не был создан, потому что исключение `OutOfMemoryError` возбуждается немедленно. Внутри логики объектов `Future` используется

специализированный экстрактор `scala.util.control.NonFatal`, и тому есть веская причина. Игнорирование важных фатальных исключений – само по себе ужасная идея. К фатальным относятся исключения: `VirtualMachineError`, `ThreadDeath`, `InterruptedException`, `LinkageError` и `ControlThrowable` (просто загляните в исходный код реализации `scala.util.control.NonFatal`). Большинство из них должны быть вам знакомы; `ControlThrowable` – это маркер для исключений, которые обычно не должны перехватываться.

Метод `onComplete` возвращает значение типа `Unit`, поэтому его нельзя объединить со следующей функцией. Существует также метод `onFailure`, позволяющий обрабатывать исключения. `onFailure` так же возвращает `Unit`, поэтому после него нужно тоже продолжить цепочку функций. В листинге 5.11 приводится пример использования `onFailure`.

#### Листинг 5.11. Использование `onFailure` для обработки исключений

```
futureFail.onFailure {      ← Вызывается в случае ошибки выполнения
  case e => println(e)     ← Соответствует всем нефатальным исключениям
}
```

Мы должны иметь возможность продолжать накапливать информацию в службе `TicketInfo`, даже когда возникает исключение. Служба `TicketInfo` собирает информацию о мероприятии и должна вернуть хотя бы часть информации, если какие-то обращения к сторонним службам потерпели неудачу. На рис. 5.10 показано, как класс `TicketInfo` накапливает информацию о мероприятии по частям.

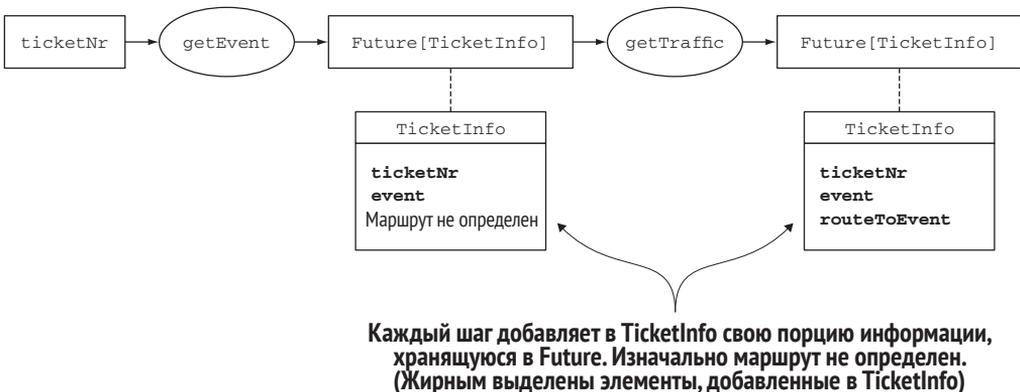


Рис. 5.10. Накопление информации о мероприятии в `TicketInfo`

Методы `getEvent` и `getTraffic` следует изменить так, чтобы они возвращали объект `Future[TicketInfo]`, где последовательно будет накапливаться информация. Класс `TicketInfo` – это простой `case`-класс, содержащий

необязательные поля для хранения результатов, возвращаемых службами. Он показан в листинге 5.12. В следующих разделах мы добавим больше информации в этот класс, например прогноз погоды и сведения о других похожих мероприятиях.

#### Листинг 5.12. case-класс TicketInfo

```
case class TicketInfo(ticketNr:String,
                     event:Option[Event]=None,
                     route:Option[Route]=None)
```

← Вся дополнительная информация для билета с номером ticketNr считается необязательной и по умолчанию отсутствует

Важно отметить, что при работе с объектами Future всегда следует использовать неизменяемые структуры данных. Иначе появится возможность применять в нескольких объектах Future общее изменяемое состояние. В данном случае мы защищены от этого, потому что использовали case-класс и тип Option, которые являются неизменяемыми. Если обращение к службе потерпит неудачу, цепочка выполнения продолжит применять TicketInfo с накопленной до этого информацией. На рис. 5.11 показано, как обрабатывается ошибка вызова GetTraffic.

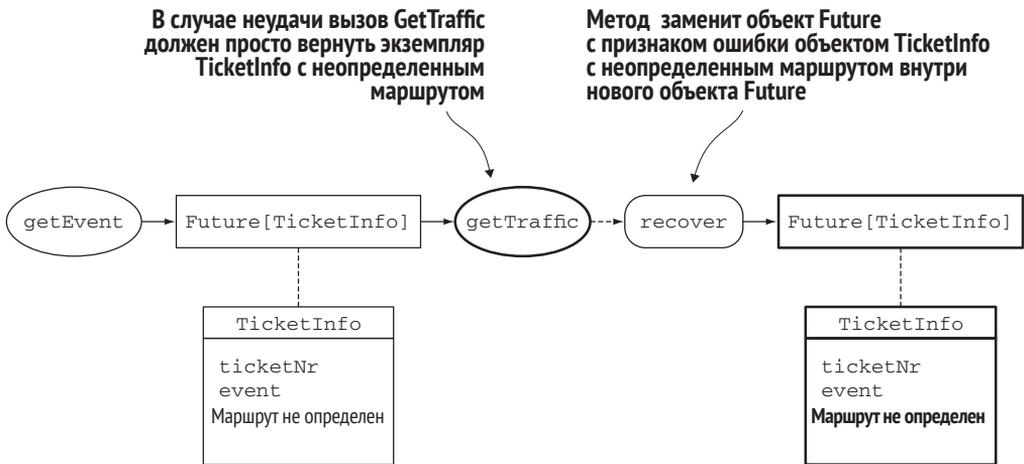


Рис. 5.11. Игнорировать неудачный вызов службы

Для достижения поставленной цели можно использовать метод recover. Он позволяет определить результат в случае появления исключения. В листинге 5.13 показано, как с его помощью вернуть входной объект TicketInfo, если возникнет исключение TrafficServiceException.

#### Листинг 5.13. Использование метода recover для продолжения работы с альтернативным результатом

```
val futureStep1: Future[TicketInfo] = getEvent(ticketNr)
```

← Определяет мероприятие; возвращает Future[TicketInfo]

```

val futureStep2: Future[TicketInfo] = futureStep1.flatMap { ticketInfo =>
  getTraffic(ticketInfo).recover {
    case _: TrafficServiceException => ticketInfo
  }
}

```

Использование flatMap позволяет напрямую вернуть Future[TicketInfo] из блока кода вместо значения TicketInfo

Возврат объекта Future с начальным значением TicketInfo

getTraffic возвращает Future[TicketInfo]

Метод `recover` определен так, что в случае появления исключения `TrafficServiceException` он должен вернуть исходное значение `ticketInfo` в качестве результата для объекта `Future`. Обычно метод `getTraffic` создает копию значения `TicketInfo` с добавленным в него маршрутом. В этом примере к объекту `Future`, возвращаемому методом `getEvent`, мы применили метод `flatMap` вместо `map`. Блок кода, который передается в `map`, должен был бы вернуть значение `TicketInfo`, которое затем будет завернуто в новый объект `Future`. Применение `flatMap` позволяет вернуть `Future[TicketInfo]` непосредственно. Так как `getTraffic` уже возвращает `Future[TicketInfo]`, удобнее использовать `flatMap`.

Существует также похожий метод `recoverWith`. Передаваемый ему блок кода должен вернуть `Future[TicketInfo]` вместо `TicketInfo`. Имейте в виду, что блок кода, переданный в вызов метода `recover`, выполняется после получения ошибки *синхронно*, поэтому желательно, чтобы он был максимально простым и быстрым.

В предыдущем фрагменте кода все еще остается одна проблема. Подумайте, что случится, если неудачу потерпит первый вызов, `getEvent`? Блок кода в вызове `flatMap` не будет вызван, потому что `futureStep1` содержит признак ошибки и не имеет никакого значения, чтобы можно было связать цепочку дальнейших вызовов. `futureStep2` получит ссылку из `futureStep1` на потерпевший неудачу объект `Future`. Чтобы в этом случае вернуть пустой экземпляр `TicketInfo`, содержащий только номер билета `ticketNr`, в первый шаг тоже нужно добавить вызов `recover`, как показано в листинге 5.14.

**Листинг 5.14.** Использование `recover` для возврата пустого `TicketInfo` в случае неудачи `getEvent`

```

val futureStep1: Future[TicketInfo] = getEvent(ticketNr)

val futureStep2: Future[TicketInfo] = futureStep1.flatMap { ticketInfo =>
  getTraffic(ticketInfo).recover {
    case _: TrafficServiceException => ticketInfo
  }
}.recover {
  case e => TicketInfo(ticketNr)
}

```

В случае неудачи в `getEvent` вернуть пустой экземпляр `TicketInfo`, содержащий только `ticketNr`

Блок кода в вызове `flatMap` не будет выполнен, если `futureStep1` потерпит неудачу. В этом случае `flatMap` просто вернет объект `Future` с признаком ошибки. Последний вызов `recover` в листинге 5.14 превратит этот объект `Future` с ошибкой в `Future[TicketInfo]`.

Теперь, когда вы узнали, как восстанавливать нормальную работу после появления ошибок в цепочке объектов `Future`, рассмотрим другие способы комбинирования этих объектов, которые могут пригодиться в реализации службы `TicketInfo`.

## 5.4. Комбинирование объектов Future

В предыдущих разделах вы познакомились с применением методов `map` и `flatMap` для объединения в цепочки вызовов асинхронных функций. В этом разделе мы рассмотрим другие способы комбинирования асинхронных функций с помощью объектов `Future`. Трейт `Future[T]` и объект `Future` оба обладают *методами-комбинаторами*, такими как `flatMap` и `map`, для объединения асинхронных вычислений. Все эти методы, такие как `flatMap` и `map`, можно найти в `Scala Collections API`. Они позволяют конструировать цепочки для преобразования одной неизменяемой коллекции в следующую или пошагового решения задачи. В этом разделе мы лишь в общих чертах рассмотрим возможности комбинирования объектов `Future` в функциональном стиле. Желаям узнать больше о функциональном программировании на `Scala` мы рекомендуем книгу «`Functional Programming in Scala`» Пола Чиусано (Paul Chiusano) и Рунара Бьярнсона (Rúnar Bjarnason) (Manning Publications, 2014).

Для получения дополнительной информации служба `TicketInfo` должна объединить вызовы нескольких веб-служб. Для пошагового добавления информации в `TicketInfo` мы будем использовать методы-комбинаторы, вызывающие функции, которые принимают `TicketInfo` и возвращают `Future[TicketInfo]`. На каждом шаге создается копия case-класса `TicketInfo`, которая передается следующей функции, и постепенно наполняется новой информацией. В листинге 5.15 показаны обновленные и измененные версии `TicketInfo` и других case-классов, используемых в службе.

**Листинг 5.15.** Улучшенный класс `TicketInfo`

```
case class TicketInfo(ticketNr:String,
                     userLocation:Location,
                     event:Option[Event]=None,
                     travelAdvice:Option[TravelAdvice]=None,
                     weather:Option[Weather]=None,
                     suggestions:Seq[Event]=Seq())
```

case-класс `TicketInfo` для хранения рекомендуемого маршрута, доступности общественного транспорта, прогноза погоды и похожих мероприятий

```
case class Event(name:String, location:Location,
```

```

        time:DateTime)

case class Weather(temperature:Int, precipitation:Boolean)

case class RouteByCar(route:String,
                    timeToLeave:DateTime,
                    origin:Location,
                    destination:Location,
                    estimatedDuration:Duration,
                    trafficJamTime:Duration)
                    ← Для простоты маршрут
                      определяется как строка

case class TravelAdvice(routeByCar:Option[RouteByCar]=None,
                       publicTransportAdvice: Option[PublicTransportAdvice]=None)

case class PublicTransportAdvice(advice:String,
                                 timeToLeave:DateTime,
                                 origin:Location, destination:Location,
                                 estimatedDuration:Duration)
                                 ← Для простоты доступность
                                   общественного транспорта
                                   определяется как строка

case class Location(lat:Double, lon:Double)

case class Artist(name:String, calendarUri:String)

```

Все поля объявлены необязательными, кроме номера билета и местоположения пользователя. На каждом шаге в последовательности действий добавляется какая-то дополнительная информация путем копирования аргумента `TicketInfo` и изменения свойств в новом значении `TicketInfo`, которое затем передается следующей функции. Информация, соответствующая шагу, может оставаться пустой, если вызов соответствующей службы потерпел неудачу, как было показано в предыдущем разделе. На рис. 5.12 изображен поток асинхронных вызовов веб-служб и комбинаторов, которые мы сконструируем в этом примере.

Комбинаторы изображены на рис. 5.12 ромбами. Мы подробно рассмотрим каждый из них. Процесс начинается с получения номера билета `ticketNr` и GPS-координат пользователя и завершается возвратом объекта `Future` с результатом `TicketInfo`. В качестве прогноза погоды будет использоваться ответ от одной из метеорологических служб, который придет раньше других. Информация о доступности общественного транспорта и маршруте поездки на личном автомобиле объединяется в `TravelAdvice`. Одновременно выявляются похожие мероприятия и время их проведения. В конечном итоге все промежуточные объекты `Future` объединяются в `Future[TicketInfo]`. Окончательный объект `Future[TicketInfo]` будет также иметь обратный вызов `onComplete`, который завершит обработку HTTP-запроса и отправит ответ клиенту, но мы не будем рассматривать его.

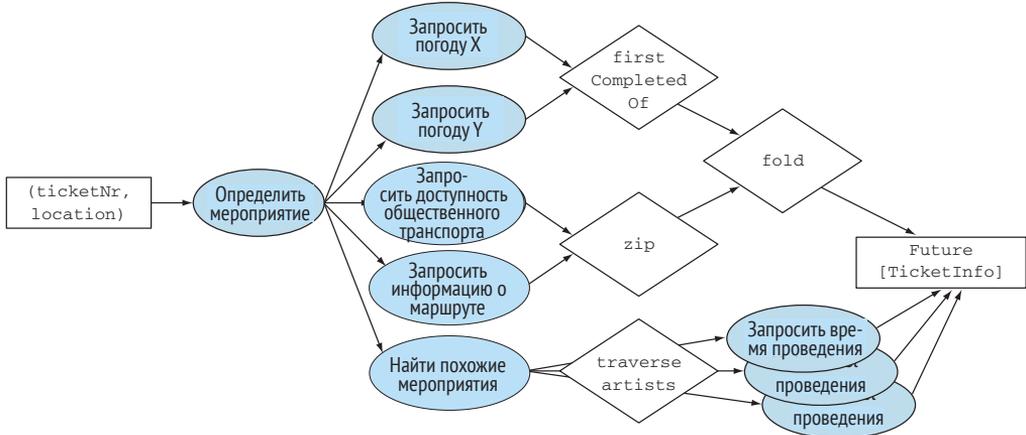


Рис. 5.12. Процесс выполнения службы TicketInfo

Начнем с обращений к метеорологическим службам. Мы должны обратиться сразу к нескольким таким службам и использовать ответ, который придет раньше. На рис. 5.13 изображены комбинаторы, используемые в этом процессе.

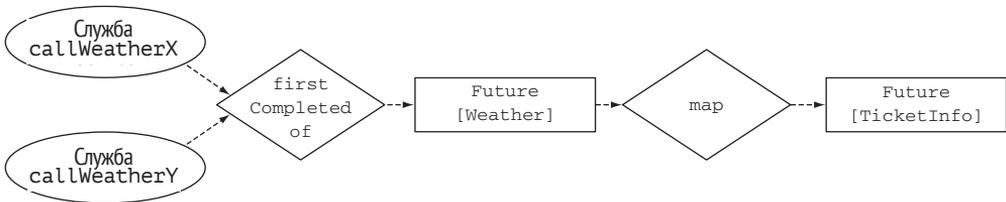


Рис. 5.13. Процесс получения погоды

Обе метеорологические службы возвращают результат `Future[Weather]`, который нужно преобразовать в значение `Future[TicketInfo]` для следующего шага. Если одна из служб не ответит, мы все еще сможем сообщить клиенту прогноз погоды, взяв его из ответа другой службы. В листинге 5.16 показано, как использовать метод `Future.firstCompletedOf` для возврата первого полученного ответа.

**Листинг 5.16.** Использование `firstCompletedOf` для возврата первого полученного ответа

```
def getWeather(ticketInfo: TicketInfo): Future[TicketInfo] = {
```

```
  val futureWeatherX: Future[Option[Weather]] =
```

```
    callWeatherXService(ticketInfo).recover(withNone)
```

```
  val futureWeatherY: Future[Option[Weather]] =
```

```
    callWeatherYService(ticketInfo).recover(withNone)
```

Для восстановления после ошибки вызывается функция `withNone` (здесь не показана). Она просто возвращает значение `None`

```

val futures: List[Future[Option[Weather]]] =
  List(futureWeatherX, futureWeatherY)

val fastestResponse: Future[Option[Weather]] =
  Future.firstCompletedOf(futures)

fastestResponse.map { weatherResponse =>
  ticketInfo.copy(weather = weatherResponse)
}

```

Первый выполнившийся Future[Weather]  
 Копирование ответа метеослужбы в новый ticketInfo. В качестве результата выполнения блока кода в map возвращается копия  
 Блок кода в map преобразует полученное значение Weather в TicketInfo, в результате чего получается Future[TicketInfo]

Для отправки запросов к метеорологическим службам сначала создаются два объекта Future. Функция Future.firstCompletedOf создает новый объект Future из результатов обращений к двум метеорологическим службам. Важно отметить, что firstCompletedOf возвращает первый завершившийся объект Future – успешно или с признаком ошибки. Код в примере 5.16 не сможет добавить прогноз погоды, полученный в результате успешного вызова, если, к примеру, вызов службы weatherX потерпит неудачу раньше, чем служба weatherY успеет вернуть успешный результат. Пока оставим все как есть, предполагая, что неправильно работающая служба действует медленнее правильно работающей службы. Вместо firstCompletedOf можно было бы использовать find. Эта функция принимает несколько объектов Future и функцию-предикат для выбора нужного и возвращает Future[Option[T]]. В листинге 5.17 показано, как можно было бы использовать find для получения первого объекта Future, завершившегося успешно.

#### Листинг 5.17. Использование find для получения первого успешного результата

```

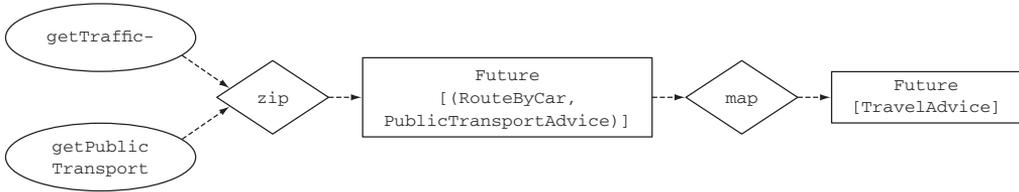
val futures: List[Future[Option[Weather]]] =
  List(futureWeatherX, futureWeatherY)

val fastestSuccessfulResponse: Future[Option[Weather]] =
  Future.find(futures)(maybeWeather => !maybeWeather.isEmpty)
    .map(_.flatten)

```

Поиск первого непустого результата  
 Результат нужно преобразовать в плоский список, потому что find принимает TraversableOnce[Future[T]] и возвращает Future[Option[T]]. В данном примере T – это Option[T]. (Значение futures имеет тип List[Future[Option[Weather]]], а не List[Future[Weather]])

Службы, возвращающие информацию о доступности общественного транспорта и маршруте для поездки на личном автомобиле, должны обрабатываться параллельно, а их результаты, когда они станут доступны, – объединяться в TravelAdvice. На рис. 5.14 изображены комбинаторы, используемые в этом процессе.



**Рис. 5.14.** Подготовка рекомендаций для выбора маршрута

Методы `getTraffic` и `getPublicTransport` возвращают значения разных типов в своих объектах `Future`, `RouteByCar` и `PublicTransportAdvice` соответственно. Эти два значения нужно сначала объединить в кортеж, а затем отобразить его в значение `TravelAdvice`. Класс `TravelAdvice` показан в листинге 5.18.

#### Листинг 5.18. Класс `TravelAdvice`

```

case class TravelAdvice(
  routeByCar: Option[RouteByCar] = None,
  publicTransportAdvice: Option[PublicTransportAdvice] = None
)
  
```

Опираясь на эту информацию, пользователь сможет решить, как он будет добираться до места проведения мероприятия – на своем автомобиле или на общественном транспорте. В листинге 5.19 показано, как для этой цели использовать комбинатор `zip`.

#### Листинг 5.19. Использование `zip` и `map` для объединения описания маршрута и информации о доступности общественного транспорта

```

def getTravelAdvice(info: TicketInfo,
  event: Event): Future[TicketInfo] = {

  val futureR: Future[Option[RouteByCar]] = callTraffic(
    info.userLocation,
    event.location,
    event.time
  ).recover(withNone)

  val futureP: Future[Option[PublicTransportAdvice]] =
    callPublicTransport(info.userLocation,
      event.location,
      event.time
    ).recover(withNone)

  futureR.zip(futureP)
    .map {
      case (routeByCar, publicTransportAdvice) =>
  
```

Объединение `Future[RouteByCar]` и `Future[PublicTransportAdvice]` в `Future[(RouteByCar, PublicTransportAdvice)]`

Преобразование информации о маршруте и доступности общественного транспорта в `Future[TicketInfo]`

```

    val travelAdvice = TravelAdvice
        (routeByCar,
         publicTransportAdvice)
    info.copy(travelAdvice = Some(travelAdvice))
  }
}

```

Предыдущий код сначала объединяет информацию о маршруте и общественном транспорте в новый объект `Future`, содержащий оба результата в виде кортежа. Затем отображает его и возвращает результат в виде объекта `Future[TicketInfo]`, который можно передать дальше по цепочке. Вместо метода `map` можно также использовать *for-генератор*. Иногда этот прием позволяет получить более читаемый код. Листинг 5.20 демонстрирует, как выглядит такая замена; он делает то же самое, что и пара методов `zip` и `map` в листинге 5.19.

**Листинг 5.20.** Использование *for-генератора* для объединения описания маршрута и информации о доступности общественного транспорта

```

for(
  (route, advice) <- futureRoute.zip(futurePublicTransport);
  travelAdvice = TravelAdvice(route, advice)
) yield info.copy(travelAdvice = Some(travelAdvice))

```

for-генератор возвращает `TicketInfo` в виде `Future[TicketInfo]`, подобно тому, как это делает метод `map`

Объект `Future`, созданный методом `zip`, в некоторый момент получит кортеж со значениями `routeByCar` и `publicTransportAdvice` tuple

Незнакомые с *for-генераторами* могут считать их средством выполнения итераций по коллекциям. В случае с объектами `Future` «итерации» выполняются по коллекции, содержащей один или ноль элементов (если в процессе выполнения `Future` возникло исключение).

Следующая часть процесса, которую мы рассмотрим, – сбор информации о похожих мероприятиях. Для этой цели будут использоваться две веб-службы, первая будет возвращать информацию об артистах, работающих в том же жанре, к которому относится текущее мероприятие. А по информации об артистах вторая служба будет возвращать графики проведения мероприятий недалеко от места проведения текущего. В листинге 5.21 показано, как выполняются сбор этой информации и формирование подсказок для передачи пользователю.

**Листинг 5.21.** Использование *for-генератора* для конструирования подсказок

```

def getSuggestions(event: Event): Future[Seq[Event]] = {
  val futureArtists: Future[Seq[Artists]] = callSimilarArtistsService(event)

```

Вернет `Future[Seq[Events]]` со списком запланированных мероприятий для каждого артиста

Вернет `Future[Seq[Artist]]` со списком похожих артистов

```

for(
  artists <- futureArtists
  events <- getPlannedEvents(event, artists)
) yield events
}

```

«artists» вычисляется асинхронно и в конечном итоге получит значение типа Seq[Artist]

for-генератор вернет Seq[Event] в виде Future[Seq[Event]]

«events» вычисляется асинхронно и в конечном итоге получит значение типа Seq[Events] – список запланированных мероприятий для каждого артиста

Предыдущий пример более сложный. Для ясности его реализация разбита на два метода, хотя его можно было бы реализовать как один сплошной блок кода. Метод `getPlannedEvents` выполняется, только когда список артистов будет готов. Он вызывает метод `Future.sequence`, чтобы сконструировать `Future[Seq[Event]]` из `Seq[Future[Event]]`. Иными словами, он объединяет несколько объектов `Future` в один, содержащий список результатов. Код метода `getPlannedEvents` показан в листинге 5.22.

**Листинг 5.22.** Объединение массива объектов `Future` с помощью метода `sequence`

```

def getPlannedEvents(event: Event,
                    artists: Seq[Artist]): Future[Seq[Event]] = {
  val events: Seq[Future[Event]] = artists.map { artist =>
    callArtistCalendarService(artist, event.location)
  }
  Future.sequence(events)
}

```

Возвращает `Future[Seq[Event]]`, список запланированных мероприятий, по одному для каждого артиста

Выполняет обход `Seq[Artists]`. Для каждого артиста вызывает службу получения расписания. Значение «events» имеет тип `Seq[Future[Event]]`

Преобразует `Seq[Future[Event]]` в `Future[Seq[Event]]`. В конечном итоге вернет список мероприятий, когда будут получены результаты всех асинхронных вызовов `callArtistCalendarService`

Метод `sequence` – это упрощенная версия метода `traverse`. В листинге 5.23 показано, как мог бы выглядеть `getPlannedEvent`, если вызов метода `sequence` заменить методом `traverse`.

**Листинг 5.23.** Объединение массива объектов `Future` с помощью метода `traverse`

```

def getPlannedEventsWithTraverse(
  event: Event,
  artists: Seq[Artist]
): Future[Seq[Event]] = {
  Future.traverse(artists) { artist =>
    callArtistCalendarService(artist, event.location)
  }
}
}

```

Метод `traverse` принимает блок кода, который должен вернуть объект `Future`. Это позволяет выполнять обход коллекции и в то же время создавать результаты в `Future`

В реализации с методом `sequence` мы сначала создали последовательность `Seq[Future[Event]]` и затем преобразовали ее в `Future[Seq[Event]]`. В реализации с методом `traverse` мы смогли сделать то же самое без промежуточного шага создания `Seq[Future[Event]]`.

Теперь нам осталось сделать последний шаг: объединить `TicketInfo` с прогнозом погоды и `TicketInfo` с рекомендациями по выбору маршрута. Используем для этого метод `fold`, как показано в листинге 5.24.

**Листинг 5.24.** Еще одно объединение с помощью метода `fold`

```

val ticketInfos = Seq(infoWithTravelAdvice, infoWithWeather)

val infoWithTravelAndWeather: Future[TicketInfo] =
  Future.fold(ticketInfos)(info) {
    (acc, elem) =>
      val (travelAdvice, weather) = (elem.travelAdvice, elem.weather)
      acc.copy(
        travelAdvice = travelAdvice.orElse(acc.travelAdvice),
        weather = weather.orElse(acc.weather)
      )
  }

```

Создаст список с `TicketInfo`, с информацией для выбора маршрута и прогноз погоды

Извлечение необязательных свойств `travelAdvice` и `weather` из `ticketInfo`

Копирует `travelAdvice` или в конструируемый `TicketInfo`. Копия возвращается как следующее значение `acc` для следующего вызова блока кода

`fold` вернет результат выполнения блока кода в аккумуляторе (`acc`). Он передаст в блок кода каждый элемент, в данном случае каждое значение `TicketInfo`

В вызов `fold` передается список и аккумулятор, инициализированный значением `ticketInfo`, содержащим только информацию о мероприятии

Метод `fold` работает в точности как одноименный метод таких структур данных, как `Seq[T]` и `List[T]`, с которым вы наверняка уже знакомы. Он часто используется вместо традиционного цикла `for` для создания некоторой структуры данных путем обхода элементов коллекции. Метод `fold` принимает коллекцию, начальное значение и блок кода, который выполняется для каждого элемента коллекции. Блок принимает два аргумента: значение для накапливаемого состояния и следующий элемент из коллекции. В предыдущем случае в качестве начального значения `TicketInfo` используется значение инициализации. В каждой итерации блок кода возвращает копию `TicketInfo` с вновь добавленной информацией, полученной из списка `ticketInfo`.

Полная реализация процесса показана в листинге 5.25.

**Листинг 5.25.** Полная реализация TicketInfoService

```

def getTicketInfo(ticketNr:String,
                  location:Location):Future[TicketInfo] = {
  val emptyTicketInfo = TicketInfo(ticketNr, location)
  val eventInfo = getEvent(ticketNr, location)
                    .recover(withPrevious(emptyTicketInfo))
  eventInfo.flatMap { info =>

    val infoWithWeather = getWeather(info)

    val infoWithTravelAdvice = info.event.map { event =>
      getTravelAdvice(info, event)
    }.getOrElse(eventInfo)

    val suggestedEvents = info.event.map { event =>
      getSuggestions(event)
    }.getOrElse(Future.successful(Seq()))

    val ticketInfos = Seq(infoWithTravelAdvice, infoWithWeather)

    val infoWithTravelAndWeather = Future.fold(ticketInfos)(info) { (acc, elem) =>
      val (travelAdvice, weather) = (elem.travelAdvice, elem.weather)

      acc.copy(travelAdvice = travelAdvice.orElse (acc.travelAdvice),
              weather = weather.orElse(acc.weather))
    }

    for(info <- infoWithTravelAndWeather;
        suggestions <- suggestedEvents
        ) yield info.copy(suggestions = suggestions)
  }

  // Функции восстановления после ошибок
  type Recovery[T] = PartialFunction[Throwable,T]

  // восстанавливает со значением None
  def withNone[T]:Recovery[Option[T]] = {
    case e => None
  }

  // восстанавливает с пустой последовательностью
  def withEmptySeq[T]:Recovery[Seq[T]] = {
    case e => Seq()
  }

```

← Вызов `getEvent` вернет `Future[TicketInfo]`

← Создаст `TicketInfo` с прогнозом погоды

← Создаст `TicketInfo` с информацией для выбора маршрута

← Получение списка похожих мероприятий

← Добавление прогноза погоды и информации для выбора маршрута в общий `TicketInfo`

← Добавление подсказок

← Методы восстановления после ошибок для использования в `TicketInfoService`

```

}

// восстанавливает с объектом ticketInfo, полученным на предыдущем шаге
def withPrevious(previous:TicketInfo):Recovery[TicketInfo] = {
  case e => previous
}

```

На этом мы завершаем пример реализации `TicketInfoService` с использованием объектов `Future`. Как было показано, объекты `Future` можно комбинировать самыми разными способами, и с помощью существующих методов-комбинаторов их легко можно преобразовывать в последовательности асинхронных вызовов функций. Служба `TicketInfoService` не выполняет ни одного блокирующего вызова. Если вызовы гипотетических веб-служб можно реализовать с использованием асинхронного HTTP-клиента, такого как библиотека *spray-client*, количество потоков выполнения, блокируемых в операциях ввода/вывода, можно свести к минимуму. На момент написания этих строк продолжали появляться новые асинхронные клиентские библиотеки для ввода/вывода и доступа к базам данных, возвращающие результаты в виде объектов `Future`.

В следующем разделе мы посмотрим, как можно объединить объекты `Future` с акторами.

## 5.5. Объединение объектов Future с акторами

В главе 2 мы использовали библиотеку *akka-http* для реализации нашей первой REST-службы. Там мы видели, что метод `ask` может вернуть `Future`, как показано в листинге 5.26.

**Листинг 5.26.** Сбор информации о мероприятии

```

class BoxOffice(implicit timeout: Timeout) extends Actor {
  // ... фрагмент кода пропущен
  case GetEvent(event) =>
    def notFound() = sender() ! None
    def getEvent(child: ActorRef) = child forward TicketSeller.GetEvent
    context.child(event).fold(notFound())(getEvent)
  case GetEvents =>
    import akka.pattern.ask
    import akka.pattern.pipe
    def getEvents: Iterable[Future[Option[Event]]] = context.children.map {
      child =>

```

- Для запроса необходимо определить тайм-аут. Если запрос не завершится в течение этого времени, в объект `Future` будет записано исключение завершения по тайм-ауту
- Импорт шаблона `ask`, добавляющего метод `ask` в `ActorRef`
- Импорт шаблона `pipe`, добавляющего метод `pipe` в `ActorRef`
- Обход дочерних акторов; вызвать `GetEvent` для каждого дочернего актора

```

    self.ask(GetEvent(child.path.name)).mapTo[Option[Event]]
  }
  def convertToEvents(f: Future[Iterable[Option[Event]]]): Future[Events] =
    f.map(_.flatten).map(l=> Events(l.toVector))

pipe(
  convertToEvents(Future.sequence(getEvents))
) to sender()

```

Объект Future передается отправителю. Нет необходимости замыкать обратный вызов в Future на значении

Выход изнутри наружу, `getEvents` превращается из `Iterable [Future[Option[Event]]]` в `Future[Iterable[Option[Event]]]` вызовом `Future.sequence`. `Future[Iterable[Option[Event]]]` превращается в `Future[Events]` вызовом `convertToEvents`

Это локальное определение преобразует коллекцию необязательных значений в список действительных результатов (значения `None` отбрасываются). Затем `Iterable[Event]` преобразуется в значение `Events`

Определение локального метода для вызова своего метода `GetEvent`, то есть метода объекта `BoxOffice`. Метод `ask` возвращает результат Future. Актеры могут посылать обратно любые сообщения, поэтому возвращаемый объект Future не типизирован. Для преобразования `Future[Any]` в `Future[Option[Event]]` мы используем метод `mapTo`, который вернет Future с признаком ошибки, если актер ответит другим сообщением, отличным от `Option[Event]`

Здесь выполняется множество действий, но теперь этот пример должен быть более понятным, чем в главе 2. Пройдемся по нему еще раз. Этот пример показывает, как актер `BoxOffice` может получить количество билетов, оставшихся нереализованными у каждого продавца.

Данный пример демонстрирует несколько важных аспектов. Во-первых, мы *передаем* результат отправителю. Это очень разумно, потому что отправитель является частью контекста актора, который может отличаться для каждого сообщения, получаемого актором. Обратный вызов в Future может *замыкаться на используемых им значениях*. Отправитель может иметь совсем другое значение в момент запуска обратного вызова. Передача сообщения отправителю подобным способом избавляет от необходимости ссылаться на `sender()` из обратного вызова в Future.

Используя объекты Future внутри акторов, помните, что `ActorContext` поддерживает текущее представление актора. И поскольку актеры хранят свое состояние, важно гарантировать невозможность изменения из других потоков значений, на которых выполняется замыкание. Самый простой способ предотвращения этой проблемы – использовать неизменяемые структуры данных и передавать объект Future в актер, как показано в этом примере. Другое решение: «сохранять» текущее значение `sender()` в переменной.

## 5.6. В заключение

В этой главе вы познакомились с объектами Future. Вы научились использовать их для создания цепочек из асинхронных вызовов функций, чтобы

минимизировать применение блокировок и циклов ожидания в потоках, оптимизировать расходование вычислительных ресурсов и избавиться от ненужных задержек.

Объект `Future` – это место, куда рано или поздно будет записан результат выполнения функции. Это отличный инструмент объединения функций в асинхронные последовательности. Объекты `Future` позволяют определять преобразования результатов из одного вида в другой. Поскольку все они фактически являются результатами функций, нет ничего удивительно, что для объединения этих результатов с успехом можно применять приемы в функциональном стиле.

Методы-комбинаторы поддерживают «стиль преобразований», напоминающий комбинаторы, которые можно найти в библиотеке поддержки коллекций в языке `Scala`. Функции выполняются параллельно, и в будущем, когда это потребуется, программа может получить их результаты. Объект `Future` может содержать результат успешных вычислений или признак ошибки. К счастью, ошибки можно исправлять и замещать значениями по умолчанию, чтобы дать возможность продолжить процесс вычислений.

Значение, содержащееся в `Future`, должно быть неизменяемым, чтобы гарантировать невозможность совместного использования изменяемого состояния по ошибке. Объекты `Future` можно использовать в акторах, но при этом старайтесь избегать использования ссылок на состояние актора из `Future`. Например, ссылку на отправителя в акторе следует сохранить в неизменяемом значении, только после этого ее можно безопасно использовать. Объекты `Future` используются в `Actor API` для возвращения результата в ответ на вызов метода `ask`. Результаты в объектах `Future` можно также отправлять акторам с помощью шаблона `pipe`.

Теперь, после знакомства с объектами `Future`, вернемся обратно к акторам. На этот раз мы улучшим масштабирование приложения `GoTicks.com`, применив удаленные акторы.

# Глава 6

## Первое распределенное приложение

В этой главе:

- введение в горизонтальное масштабирование;
- распределенная версия приложения GoTicks.com;
- создание распределенных акторов с помощью модуля akka-remote;
- тестирование распределенных систем акторов.

До сих пор мы рассматривали приемы создания системы акторов Akka только на одном узле. В этой главе мы познакомимся с приемами и примерами горизонтального масштабирования приложений Akka. Здесь вы создадите свое первое распределенное приложение, взяв за основу приложение GoTicks.com из главы 2.

Сначала мы рассмотрим некоторые общие термины и подходы к горизонтальному масштабированию, поддерживаемые в Akka. Вы также познакомитесь с модулем akka-remote и способами организации взаимодействий между акторами в сети, которые он предоставляет. Потом мы масштабируем приложение GoTicks.com, запустив его на двух узлах: веб-интерфейсе и сервере бизнес-логики. И наконец, вы увидите, как тестировать распределенные приложения с использованием тестовых виртуальных машин JVM.

Эта глава – всего лишь введение в горизонтальное масштабирование приложений; в последующих главах вы найдете более обширные сведения. Например, в главе 9 вы узнаете, как с помощью маршрутизаторов распределять нагрузку между несколькими акторами, которые могут действовать на удаленных узлах, а в главе 14 мы расскажем вам о кластерах – в ней вы узнаете еще больше деталей о горизонтальном масштабировании приложений Akka.

## 6.1. Горизонтальное масштабирование

Возможно, вы надеялись найти в этой главе панацею, которая поможет распределить приложение на тысячу машин, но вы должны понимать, что распределенные вычисления очень сложны. Необычайно сложны. Но это не значит, что вы должны на этом закрыть книгу и отложить ее. Продолжайте читать! По крайней мере, Akka даст вам некоторые замечательные инструменты, которые облегчат вашу задачу. Akka не обещает бесплатный сыр, но акторы не только упрощают конкурентное программирование, они также облегчают реализацию по-настоящему распределенных вычислений. Здесь мы вернемся к нашему проекту GoTicks.com и сделаем его распределенным.

Для организации взаимодействий между объектами по сети в большинстве сетевых технологий используется метод блокирующего вызова удаленных процедур (Remote Procedure Call, RPC), который стремится замаскировать различия между вызовами локальных и удаленных объектов. Этот подход основан на идее, что модель локального программирования проще, поэтому предпочтительнее, если программист будет использовать ее, а внутренняя реализация обеспечит прозрачность удаленных взаимодействий. Этот способ хорошо работает для соединений типа «точка-точка», но никуда не годится для крупномасштабных сетей, в чем вы убедитесь в следующем разделе. В Akka используется иной подход к масштабированию приложений в сети. Он объединяет лучшие черты двух других подходов, благодаря чему мы получаем относительную прозрачность удаленных взаимодействий и можем использовать прежний прикладной код с акторами – далее вы увидите, что код на верхнем уровне остается неизменным.

Прежде чем углубиться в тему масштабирования, в следующем разделе мы кратко познакомимся с используемой терминологией и некоторыми примерами топологии сетей, если они вам еще не знакомы. Если вы считаете себя опытным специалистом в этой области, можете сразу перейти к разделу 6.2.

### 6.1.1. Общая терминология

Под термином *узел* (node) в этой главе мы будем подразумевать действующее приложение, поддерживающее возможность обмена данными по сети. Это точка подключения в топологии сети. Это часть распределенной системы. Несколько узлов может выполняться на одном физическом сервере или на нескольких. На рис. 6.1 показаны некоторые типичные топологии сетей.

Узел наделен определенной *ролью* в распределенной системе. Он отвечает за решение конкретных задач. Например, узел может быть частью

распределенной базы данных или одним из нескольких веб-серверов, обеспечивающих доступ к системе через веб-интерфейс.

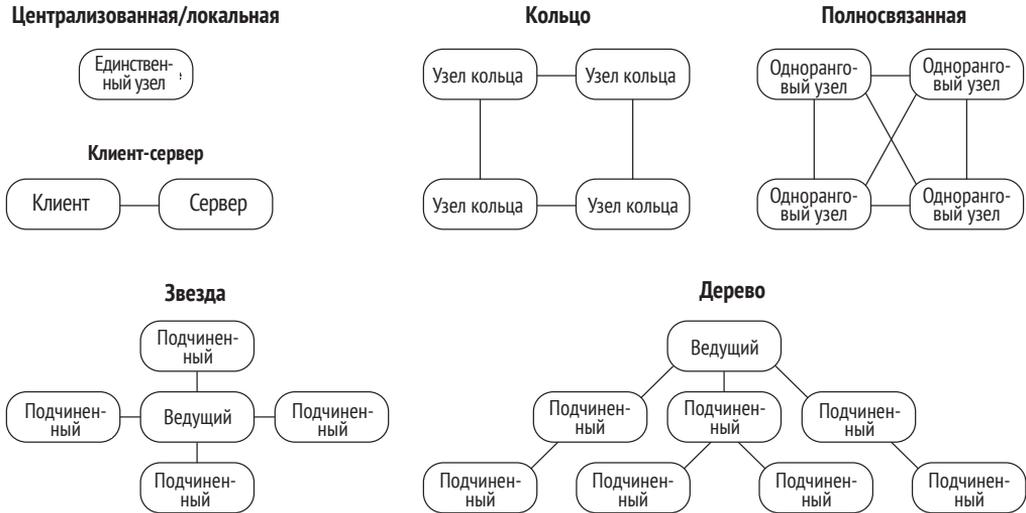


Рис. 6.1. Типичные топологии сетей

Для взаимодействий узел использует определенный *транспортный протокол*. Примерами таких транспортных протоколов могут служить TCP/IP и UDP. Сообщения, пересылаемые узлами друг другу с помощью транспортного протокола, должны кодироваться и декодироваться в *блоки данных протокола* (protocol data units). Блоки данных несут в себе двоичное представление сообщений в виде массивов байтов. Для преобразования сообщений в байты и обратно используются процедуры *сериализации* и *десериализации* соответственно. Для этих целей в Akka имеется свой модуль сериализации, который мы коротко рассмотрим в этой главе.

Узлы, входящие в состав одной распределенной системы, являются *членами группы*. Членство может быть *статическим* или *динамическим* (или даже смешанным). В случае статического членства количество узлов и роль каждого узла фиксированы и не могут изменяться на протяжении всего периода работы такой сети. Динамическое членство позволяет узлам играть разные роли и присоединяться к сети или отключаться от нее.

Статическое членство, как вы понимаете, проще в реализации. Все серверы поддерживают связи с другими узлами в сети, установленные в момент запуска. Но оно также наименее гибкое; узел нельзя просто заменить другим узлом, выполняющимся на другой машине с другим сетевым адресом.

Динамическое членство обладает большей гибкостью и позволяет группам узлов расширяться и сокращаться по мере необходимости. Оно позволяет восстанавливать общую работоспособность сети при выходе из

стройка некоторых узлов автоматической заменой их другими узлами. Но оно также сложнее в реализации. Для нормальной работы динамического членства требуется реализовать механизм динамического присоединения к группе или выхода из нее, обнаружения и устранения сбоев, выявления недоступных/отказавших узлов, а также предоставить некоторый механизм *обнаружения*, который будет помогать новым узлам находить группу в сети, потому что при динамическом членстве сетевые адреса определяются динамически, а не статически.

Теперь, после краткого знакомства с топологиями сетей и общими терминами, попробуем ответить на вопрос, почему в Akka используется модель распределенного программирования и для локальных, и для распределенных систем.

### 6.1.2. Причины использования модели распределенного программирования

Наша конечная цель – развернуть приложение до нескольких узлов, но отправной точкой часто служит локальное приложение, действующее на одном узле: вашем ноутбуке. Какие изменения нужно произвести, чтобы сделать шаг в сторону одной из распределенных топологий, представленных в предыдущем разделе? Можно ли абстрагироваться от факта, что все эти узлы выполняются на одном «виртуальном узле», и воспользоваться какими-нибудь интеллектуальными инструментами, которые возьмут на себя решение всех связанных с этим проблем, и нам не придется менять свой код, прекрасно работающий на ноутбуке? Если говорить кратко – нет, нельзя<sup>1</sup>. Нельзя просто взять и обобщить различия между локальным и распределенным окружением. Впрочем, мы не требуем верить нам на слово. Как описывается в статье «Note on Distributed Computing» (Замечания о распределенных вычислениях)<sup>2</sup>, локальная и распределенная модели программирования имеют четыре важных отличия, которые нельзя игнорировать. К ним относятся: задержки, доступ к памяти, частичный отказ и конкуренция. Следующий список кратко характеризует эти четыре отличия:

- *Задержки* – наличие сети между взаимодействующими объектами предполагает большие затраты времени на передачу каждого сообщения – примерное время извлечения сообщения из кеша L1 составляет что-то около 0.5 нс, время извлечения из основной памяти занимает примерно 100 нс, а для передачи пакета из Нидерландов в Калифорнию требуется около 150 мс – сюда также можно приплюсовать задержки из-за высокого трафика в часы пик, повторной передачи пакетов, разрывов соединения и т. д.

<sup>1</sup> Производители программного обеспечения, продающие эту идею, не согласятся с этим заявлением!

<sup>2</sup> Джим Валдо (Jim Waldo), Джефф Вайант (Geoff Wyant), Энн Уолрат (Ann Wollrath) и Сэм Кендалл (Sam Kendall), Sun Microsystems, Inc., 1994.

- *Частичный отказ* – определение работоспособности всех частей распределенной системы – сложная задача, особенно когда некоторые части системы доступны непостоянно, могут отключаться и снова появляться.
- *Доступ к памяти* – операция получения ссылки на объект в памяти в локальной системе не может прерываться, что может происходить при получении ссылки на объект в распределенном окружении.
- *Конкуренция* – единый «хозяин» отсутствует, поэтому из-за факторов, перечисленных выше, план чередования операций может пойти наперекосяк.

Из-за этих отличий применение модели локального программирования в распределенной среде заканчивается крахом. Но фреймворк Akka пошел другим путем: он предлагает использовать распределенную модель для обоих видов окружений – распределенного и локального. Ранее упомянутая статья ссылается на этот выбор, и в ней отмечается, что модель распределенного программирования упрощает создание распределенных приложений, но может сделать локальное программирование таким же сложным, как распределенное.

Но времена изменились. Спустя два десятка лет мы вынуждены иметь дело с многоядерными процессорами. И все больше задач приходится решать в распределенном облаке. Преимущество использования модели распределенного программирования для создания локальных систем заключается в том, что она упрощает конкурентное программирование, как вы могли видеть в предыдущих главах. Мы уже привыкли к асинхронным взаимодействиям, научились бороться с частичными отказами и используем обобщенный подход к конкурентному выполнению, что упрощает нам программирование для многопроцессорных систем и подготавливает нас к переходу в распределенную среду.

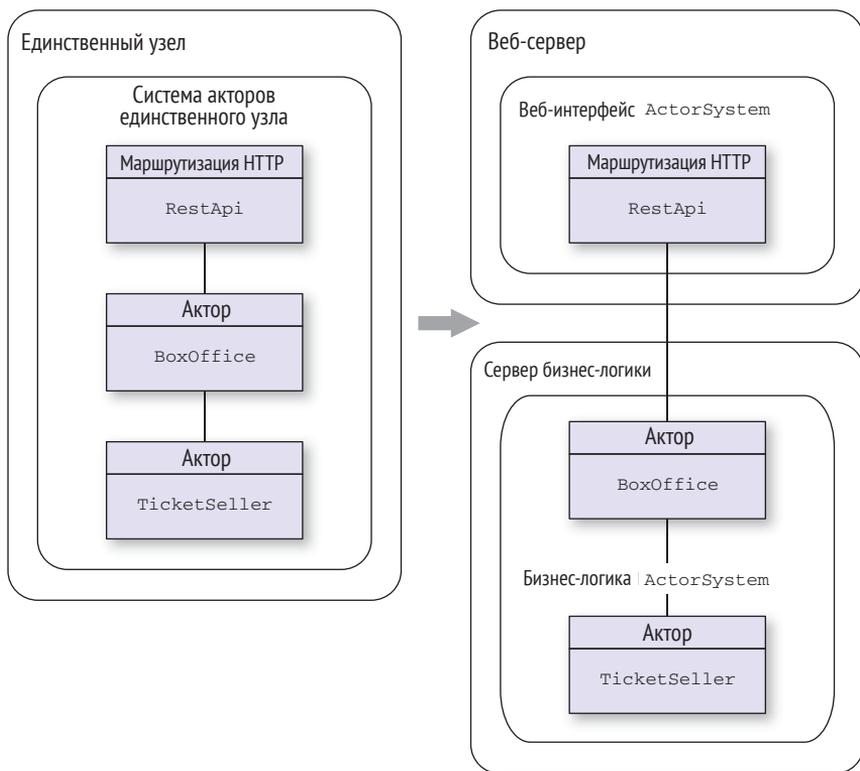
Мы покажем вам, что этот выбор обеспечивает надежный фундамент для создания локальных и распределенных приложений, соответствующих требованиям сегодняшнего дня. Akka предлагает простой API для асинхронного программирования, а также средства и инструменты для тестирования приложений удаленно или локально. Теперь, познакомившись с причинами, оправдывающими использование модели распределенного программирования для создания любых систем, локальных и распределенных, в следующих разделах мы посмотрим, как масштабировать приложение GoTicks.com, которое мы написали в главе 2.

## 6.2. Горизонтальное масштабирование и удаленные взаимодействия

Эта глава является лишь введением в горизонтальное масштабирование, поэтому мы используем относительно простой пример приложения

GoTicks.com из главы 2. В следующих разделах мы изменим приложение так, что оно сможет выполняться на нескольких узлах. Несмотря на простоту GoTicks.com, этот пример поможет вам понять, какие изменения необходимо внести в приложение, первоначально не предназначенное для масштабирования.

Мы определим статическое членство двух узлов, используя топологию клиент-сервер, так как это самый простой путь от локального приложения к распределенному. В этом окружении два узла будут выполнять роли веб-интерфейса и сервера бизнес-логики. На узле с веб-интерфейсом будет использована архитектура REST. А на сервере с бизнес-логикой будут действовать `BoxOffice` и все `TicketSeller`. Оба узла будут использовать статические связи для взаимодействий друг с другом. На рис. 6.2 показано, какие изменения предстоит внести.



**Рис. 6.2.** От централизованной топологии к топологии клиент-сервер

Для поддержки удаленных взаимодействий между узлами мы используем модуль `akka-remote`. В локальной версии, при создании нового мероприятия `Event`, актор `BoxOffice` создает несколько акторов `TicketSeller`. В клиент-серверной версии он будет делать то же самое. Как будет показано далее, модуль `akka-remote` позволяет создавать и разворачивать акторы

удаленно. Веб-интерфейс будет отыскивать актор `BoxOffice` на удаленном узле, адрес которого известен, и передавать ему команду создать акторы `TicketSeller`. Мы также рассмотрим вариант развертывания веб-узлом актора `BoxOffice` на сервере с бизнес-логикой.

В следующем разделе мы займемся организацией удаленных взаимодействий. Сначала посмотрим, какие изменения необходимо внести в файл сборки `sbt`, а затем исследуем изменения в остальном коде.

### 6.2.1. Реорганизация приложения `GoTicks.com`

Измененную версию примера из главы 2 вы найдете в папке *chapter-remoting*, в каталоге *akka-in-action*. Но также можно следовать за примерами, что приводятся далее, и постепенно вносить изменения в исходный код примера непосредственно из главы 2. Первое, что нужно сделать, – добавить в файл сборки `sbt` зависимости от модулей `akka-remote` и `akka-multinode-testkit`.

**Листинг 6.1.** Изменения в файле сборки для распределенной версии `GoTicks`

```
"com.typesafe.akka" %% "akka-remote" % akkaVersion,
"com.typesafe.akka" %% "akka-multi-node-testkit" % akkaVersion % "test",
```

Зависимость от модуля `akka-multinode-testkit` с инструментами  
тестирования распределенных систем акторов

Зависимость от модуля  
`akka-remote`

Эти зависимости будут подключены автоматически в момент запуска `sbt`, но точно так же их можно подключить вручную, командой `sbt update`. Теперь, разобравшись с зависимостями, посмотрим, какие изменения нужно внести, чтобы обеспечить связь между веб-сервером и сервером бизнес-логики. Акторам, действующим на этих серверах, нужно получать ссылки друг на друга, этим мы и займемся в следующем разделе.

### 6.2.2. Удаленные взаимодействия в REPL

`Акка` поддерживает два способа получить ссылку на удаленный актор. Один из них – поиск актора по пути; другой – создание актора, получение ссылки на него и развертывание актора на удаленном узле. Сначала рассмотрим первый вариант.

Консоль REPL – отличный интерактивный инструмент для быстрого исследования новых классов в языке `Scala`. Давайте создадим две системы акторов в двух сеансах REPL, используя консоль `sbt`. Первый сеанс будет представлять систему акторов с бизнес-логикой, а второй – систему акторов веб-интерфейса. Чтобы создать сеанс с бизнес-логикой, запустите терминал в папке *chapter-remoting*, используя консоль `sbt`. Мы должны

включить поддержку удаленных взаимодействий, поэтому первое, что мы должны сделать, – произвести некоторые настройки. Обычно такие настройки определяются в конфигурационном файле *application.conf*, в папке *src/main/resources*, но в случае с сеансом REPL их можно загрузить из строки. В листинге 6.2 показаны команды REPL, выполняемые с помощью команды `:paste`.

**Листинг 6.2.** Команды REPL для создания строки с настройками удаленных взаимодействий

```
scala> :paste
// Вход в режим вставки (выход комбинацией ctrl-D)

val conf = """
akka {
  actor {
    provider = "akka.remote.RemoteActorRefProvider"
  }
  remote {
    enabled-transport = ["akka.remote.netty.tcp"]
    netty.tcp {
      hostname = "0.0.0.0"
      port = 2551
    }
  }
}
"""

// Выход из режима вставки в режим интерпретации.
...
scala>
```

Выбрать провайдера удаленных ссылок ActorRef для поддержки удаленных взаимодействий

Раздел с настройками удаленных взаимодействий

Включить транспорт TCP

Параметры транспорта TCP: хост и порт для приема запросов

Чтобы завершить команду `:paste`, нажмите Ctrl-D

Эту конфигурационную строку мы загрузим в систему акторов ActorSystem. Самое примечательное здесь – использование специализированного провайдера RemoteActorRefProvider, который развернет модуль akka-remote. Как можно догадаться по его имени, он также позаботится о поддержке ссылок ActorRefs на удаленные акторы. Код в листинге 6.3 сначала импортирует конфигурационную строку и необходимые пакеты, а затем загружает конфигурацию в систему акторов.

**Листинг 6.3.** Настройка удаленных взаимодействий

```
scala> import com.typesafe.config._
import com.typesafe.config._

scala> import akka.actor._
```

Преобразование строки в объект Config

```
import akka.actor._

scala> val config = ConfigFactory.parseString(conf)
config: com.typesafe.config.Config = ...

scala> val backend = ActorSystem("backend", config)
[Remoting] Starting remoting
.....
[Remoting] Remoting now listens on addresses:
[akka.tcp://backend@0.0.0.0:2551]
backend: akka.actor.ActorSystem = akka://backend
```

← Создание системы акторов с использованием настроек в объекте Config

Если вы следуете за примерами, вводя предлагаемые команды у себя в терминале, – вы только что запустили свою первую версию системы акторов с поддержкой удаленных взаимодействий в REPL; и это было совсем несложно! Нам потребовалось всего пять строк кода, чтобы настроить и запустить сервер.

Система акторов с бизнес-логикой была создана с использованием конфигурационного объекта, включающего поддержку удаленных взаимодействий. Если забыть передать его в вызов конструктора `ActorSystem`, система акторов запустится, но возможность удаленных взаимодействий в ней будет выключена, потому что файл `application.conf` с настройками по умолчанию, входящий в состав Акка, не включает эту поддержку. Теперь модуль `Remoting` будет принимать входящие запросы на всех сетевых интерфейсах (0.0.0.0), на порте 2551. Давайте добавим простой актер, который просто будет выводить все, что получит с консоли, чтобы мы могли убедиться, что все работает.

**Листинг 6.4.** Создание и запуск актора, который просто выводит поступающие сообщения

```
scala> :paste
// Вход в режим вставки (выход комбинацией ctrl-D)

class Simple extends Actor {
  def receive = {
    case m => println(s"received $m!")
  }
}

// выход из режима вставки в режим интерпретации.

defined class Simple

scala> backend.actorOf(Props[Simple], "simple")
res0: akka.actor.ActorRef = Actor[akka://backend/user/simple#485913869]
```

← Создает в системе акторов backend актер Simple с именем «simple»

Теперь в системе акторов backend выполняется актор Simple. Обратите внимание, что актор Simple создан с именем "simple". Это поможет отыскать актор по имени, когда мы будем подключаться к системе акторов из сети.

Теперь запустите другой терминал: выполните команду `sbt console` и создайте другую систему акторов с поддержкой удаленных взаимодействий, с именем frontend. Используйте те же настройки, кроме номера порта TCP, как показано в листинге 6.5.

#### Листинг 6.5. Создание интерфейсной системы акторов

```
scala> :paste
// Вход в режим вставки (выход комбинацией ctrl-D)

val conf = """
akka {
  actor {
    provider = "akka.remote.RemoteActorRefProvider"
  }
  remote {
    enabled-transport = ["akka.remote.netty.tcp"]
    netty.tcp {
      hostname = "0.0.0.0"
      port = 2552
    }
  }
}
"""
```

← Запустит систему акторов frontend с другим номером порта, чтобы системы frontend и backend могли действовать на одной машине

```
import com.typesafe.config._

import akka.actor._

val config = ConfigFactory.parseString(conf)

val frontend= ActorSystem("frontend", config)
// выход из режима вставки в режим интерпретации.

...
[INFO] ... Remoting now listens on addresses:
      [akka.tcp://frontend@0.0.0.0:2552]
...
frontend: akka.actor.ActorSystem = akka://frontend

scala>
```

Этот код загрузит конфигурацию в интерфейсную систему акторов frontend и запустит ее. Теперь попробуем получить ссылку на актор Simple из системы backend на стороне системы frontend. Сначала сконструируем путь к актору. На рис. 6.3 показано, из каких компонентов состоит путь.

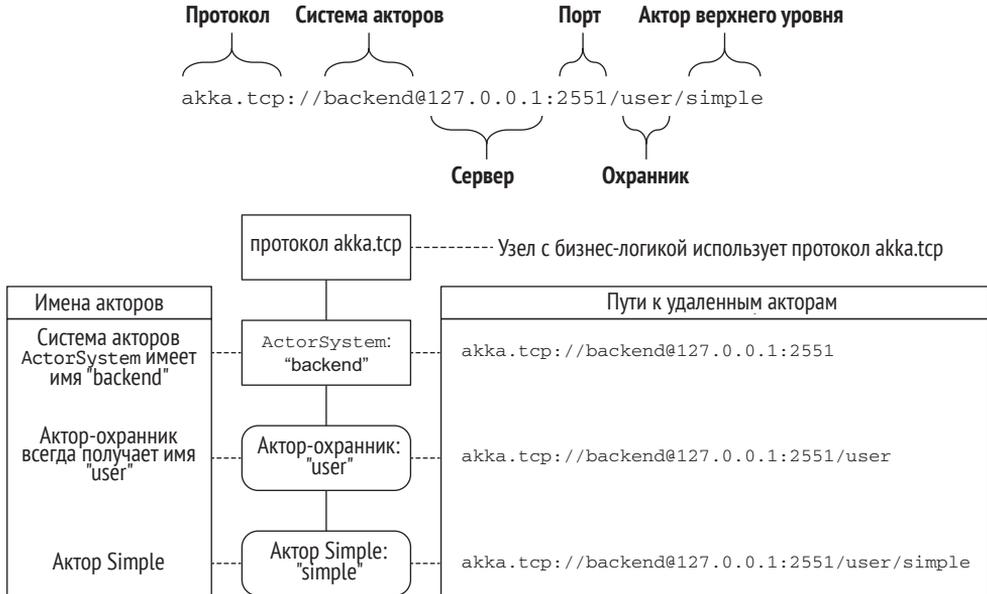


Рис. 6.3. Путь к удаленному актору

Путь можно определить как строку и использовать метод actorSelection системы акторов frontend, чтобы выполнить поиск.

#### Листинг 6.6. Использование actorSelection

```
scala> :paste
// Вход в режим вставки (выход комбинацией ctrl-D)

val path = "akka.tcp://backend@0.0.0.0:2551/user/simple"
val simple = frontend.actorSelection(path)
// Выход из режима вставки в режим интерпретации.
```

| Путь к удаленному актору Simple  
 ←  
 ← Вернет актор в виде объекта ActorSelection

```
path: String = akka.tcp://backend@0.0.0.0:2551/user/simple
simple: akka.actor.ActorSelection = ActorSelection[
  Anchor(akka.tcp://backend@0.0.0.0:2551/), Path(/user/simple)]
```

Метод actorSelection можно считать аналогом запроса в иерархии акторов. В данном случае телом запроса является путь к удаленному актору. ActorSelection – это объект, представляющий все акторы, найденные методом actorSelection в системе акторов. Объект ActorSelection можно использовать для отправки сообщений всем акторам, соответствующим

запросу. Сейчас нам не нужна прямая ссылка `ActorRef` на актор `Simple`; нужна возможность послать ему сообщение, и `ActorSelection` прекрасно подходит для этого. Так как система акторов `backend` уже запущена в другой консоли, следующий код должен выполниться без ошибок:

```
scala> simple ! "Hello Remote World!"  
  
scala>
```

В окне терминала, где действует система акторов `backend`, должно появиться сообщение:

```
scala> received Hello Remote World!!
```

В консоли REPL можно видеть, что сообщение было послано из `frontend` в `backend`. Такая возможность интерактивно исследовать удаленные системы с помощью консоли REPL бесценна, поэтому мы широко будем использовать ее в других главах.

За кулисами сообщение «Hello Remote World!» сериализуется, посылается в сокет TCP, принимается модулем `remoting`, десериализуется и передается актору `Simple`, действующему в системе `backend`.

**ПРИМЕЧАНИЕ.** В консоли REPL с успехом можно использовать механизм сериализации из Java, но его не следует применять в действующих распределенных приложениях. Этот механизм не поддерживает эволюцию схемы; незначительные изменения в коде могут нарушить нормальное взаимодействие систем. Он медленнее других решений и страдает проблемами безопасности, когда десериализации подвергаются данные, полученные из ненадежных источников. Акка предупредит об этом, если попытаться использовать его.

Возможно, вы обратили внимание, что в примере отсутствует какой-либо код, отвечающий за сериализацию. Тогда почему он работает? Это объясняется тем, что мы послали простую строку ("Hello Remote World!"). По умолчанию Акка использует механизм сериализации из Java для всех сообщений, посылаемых в сеть.

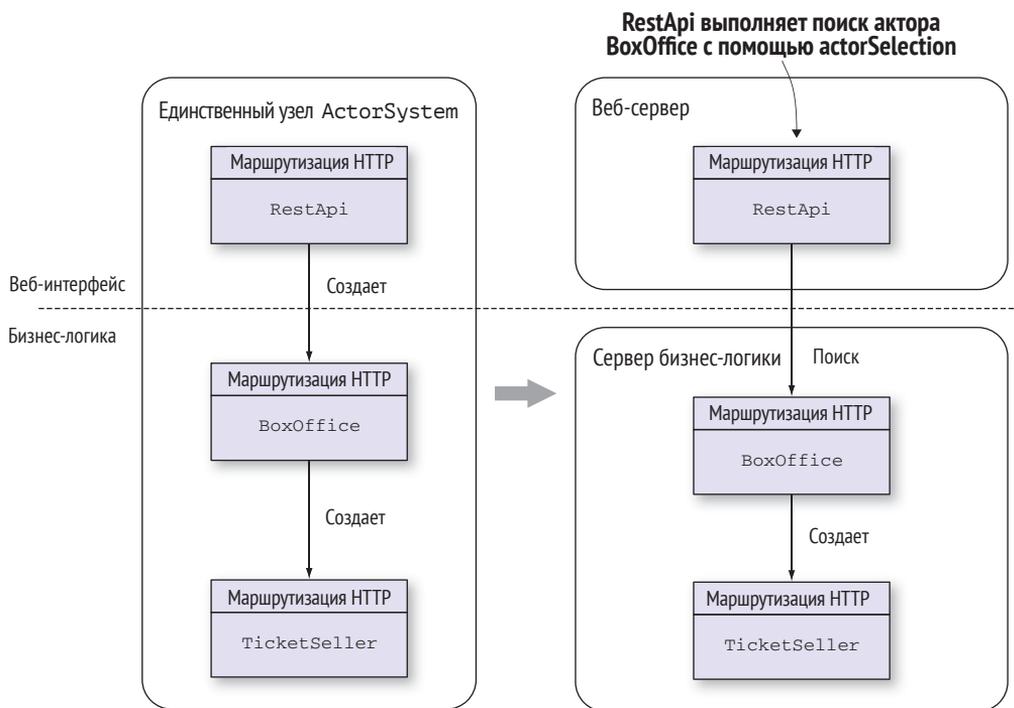
Но, кроме него, доступны другие средства сериализации, а также есть возможность написать свой сериализатор, но об этом мы поговорим в третьей части. *Протокол удаленных сообщений* в Акка имеет поле с именем сериализатора, использованным для сериализации сообщения, благодаря которому модуль `remoting` на принимающей стороне сможет десериализовать полученную последовательность байтов. Если в сообщении потребуется передать объект, его класс должен реализовать интерфейс `Serializable` и присутствовать в пути поиска классов на обеих сторонах.

К счастью, «стандартные» case-классы и case-объекты, которые используются в роли сообщений в приложении GoTicks.com, уже реализуют этот интерфейс<sup>3</sup> по умолчанию.

Теперь, узнав, как отыскать удаленный актер и послать ему сообщение в оболочке REPL, посмотрим, как применить это знание в приложении GoTicks.com.

### 6.2.3. Удаленный поиск

Вместо создания актора `BoxOffice` в `RestApi` мы отыщем его на удаленном узле `backend`. На рис. 6.4 показано, чего мы попытаемся добиться.



**Рис. 6.4.** Поиск удаленного актора `BoxOffice`

В предыдущей версии актер `RestApi` сам создавал дочерний актер `BoxOffice`:

```
val boxOffice = context.actorOf(Props[BoxOffice], "boxOffice")
```

В результате этого вызова получался актер `boxOffice`, подчиненный непосредственно актору `RestApi`. Чтобы сделать приложение более гибким и получить возможность запускать его на одном узле или на двух, добавим специальные объекты `Main`, запускающие приложение в двух разных режимах. Оба класса `Main` создают актер `BoxOffice` или получают ссылку на

<sup>3</sup> `Serializable` – это лишь интерфейс-маркер, ничего не гарантирующий. Вы должны убедиться в поддержке сериализации/десериализации при использовании нестандартных конструкций.

него немного по-разному. Код из главы был немного реорганизован, чтобы упростить воплощение разных сценариев. Этот трейт, а также необходимые изменения в реализации `RestApi` показаны в листинге 6.7.

Объекты `SingleNodeMain`, `FrontendMain` и `BackendMain` предназначены для запуска приложения на одном или на двух разных узлах. В листинге 6.7 показаны наиболее интересные для нас фрагменты кода из определений всех трех основных классов.

### Листинг 6.7. Главные акторы

```
object SingleNodeMain extends App ← Фрагмент из SingleNodeMain
  with Startup {
    ← ...опущен код чтения конфигурации
    ← и создания системы акторов
    ← Запуск HTTP-сервера; привязка
    ← маршрутов перенесена в трейт Startup
    val api = new RestApi() {
      ← Создание анонимного класса из трейта RestApi,
      ← настройка способа создания VoXOffice
      ← ...код опущен
      def createVoXOffice: ActorRef = system.actorOf(VoXOffice.props,
        VoXOffice.name) ← Создается VoXOffice,
        ← как прежде
    }
    startup(api.routes)
  }

object FrontendMain extends App ← Фрагмент из FrontendMain
  with Startup {
    ← ...опущен код чтения конфигурации
    ← и создания системы акторов
    ← Запуск HTTP-сервера; привязка
    ← маршрутов перенесена в трейт Startup
    val api = new RestApi() {
      ← ...опущен код чтения конфигурации
      ← и создания системы акторов
      ← Создание анонимного класса из трейта RestApi,
      ← настройка способа создания VoXOffice
      def createPath(): String =
        ← ...опущен код создания пути
        ← к удаленному актору
      def createVoXOffice: ActorRef = {
        val path = createPath()
        system.actorOf(Props(new RemoteLookupProxy(path)), "lookupVoXOffice") ←
      }
    }
    startup(api.routes)
  }

object BackendMain extends App with RequestTimeout { ← Фрагмент из BackendMain
  ← ...опущен код чтения конфигурации
  ← и создания системы акторов
  system.actorOf(VoXOffice.props, VoXOffice.name) ← Создание актора VoXOffice верхнего
  ← уровня на сервере с бизнес-логикой
}
```

Поиск VoXOffice на удаленном узле

Все классы Main загружают свои настройки из специальных конфигурационных файлов: SingleNodeMain из *singlenode.conf*, FrontendMain из *frontend.conf* и BackendMain из *backend.conf*. Файл *singlenode.conf* является точной копией *application.conf* из главы 2. Файл *backend.conf* определяет конфигурацию удаленного узла, как в примере REPL, и настройки журналирования; его содержимое показано в листинге 6.8.

**Листинг 6.8.** Файл *backend.conf* – конфигурация узла с бизнес-логикой

```
akka {
  loglevel = DEBUG
  stdout-loglevel = WARNING
  event-handlers = ["akka.event.slf4j.Slf4jLogger"]

  actor {
    provider = "akka.remote.RemoteActorRefProvider"
  }
  remote {
    enabled-transport = ["akka.remote.netty.tcp"]
    netty.tcp {
      hostname = "0.0.0.0"
      port = 2551
    }
  }
}
```

Подробнее о настройках журналирования рассказывается в главе 7 «Настройка, журналирование и развертывание».

Файл *frontend.conf* представляет собой смесь *singlenode.conf* и *backend.conf* и включает дополнительную секцию с параметрами для поиска актора `BoxOffice`. Эти дополнительные настройки использует `RemoteBoxOfficeCreator`.

**Листинг 6.9.** Файл *frontend.conf* – конфигурация узла с веб-интерфейсом

```
akka {
  loglevel = DEBUG
  stdout-loglevel = DEBUG
  loggers = ["akka.event.slf4j.Slf4jLogger"]

  actor {
    provider = "akka.remote.RemoteActorRefProvider"
  }

  remote {
    enabled-transport = ["akka.remote.netty.tcp"]
```

```

netty.tcp {
  hostname = "0.0.0.0"
  port = 2552
}

http {
  server {
    server-header = "GoTicks.com REST API"
  }
}

http {
  host = "0.0.0.0"
  host = ${?HOST}
  port = 5000
  port = ${?PORT}
}

backend {
  host = "0.0.0.0"
  port = 2551
  protocol = "akka.tcp"
  system = "backend"
  actor = "user/boxOffice"
}

```

Узлу с веб-интерфейсом необходима информация об узле с бизнес-логикой, чтобы иметь возможность взаимодействовать с удаленным актором `BoxOffice`. В консоли REPL, когда мы были уверены в присутствии удаленной стороны и нам требовалось просто попробовать послать сообщение, объекта `ActorSelection` было вполне достаточно. Но в данном случае нужна ссылка `ActorRef`, поскольку версия с единственным узлом использует именно ее. Для этого создадим в объекте `FrontendMain` новый актор `RemoteLookupProxy`, который возьмет на себя задачу поиска удаленного актора `BoxOffice` и передачи ему сообщений.

#### Листинг 6.10. Поиск удаленного актора `BoxOffice`

```

def createPath(): String = {
  val config = ConfigFactory.load("frontend").getConfig("backend")
  val host = config.getString("host")
  val port = config.getInt("port")
  val protocol = config.getString("protocol")
  val systemName = config.getString("system")
}

```

← Создает путь к `BoxOffice`

←  
Загрузить конфигурацию  
из `frontend.conf` и получить  
настройки узла с бизнес-логикой

```

val actorName = config.getString("actor")
s"$protocol://$systemName@$host:$port/$actorName"
}

def createBoxOffice: ActorRef = {
  val path = createPath()
  system.actorOf(Props(new RemoteLookupProxy(path)), "lookupBoxOffice")
}

```

← Возвращает актор, который ищет `BoxOffice`. При конструировании данного актора используется единственный аргумент: путь к удаленному актору `BoxOffice`

Объект `FrontendMain` создает отдельный актор `RemoteLookupProxy` для поиска `BoxOffice`. В предыдущих версиях Akka можно было получить ссылку `ActorRef` на удаленный актор вызовом метода `actorFor`. Сейчас этот метод объявлен устаревшим, потому что возвращаемая им ссылка `ActorRef` действует не совсем так, как локальная ссылка `ActorRef`, когда соответствующий ей актор завершается. Обращение к ссылке, возвращаемой методом `actorFor`, могло вызывать создание нового экземпляра удаленного актора, хотя такого никогда не происходит в локальном контексте. Кроме того, в прошлом не было возможности следить за завершением удаленных акторов, что стало еще одной причиной отказа от этого метода.

Вот основные причины для использования актора `RemoteLookupProxy`:

- система акторов с бизнес-логикой еще не была запущена или потерпела аварию и только что была перезапущена;
- сам актор `BoxOffice` потерпел аварию и еще не успел перезапуститься;
- в идеале можно запустить узел с бизнес-логикой перед запуском узла с веб-интерфейсом, чтобы поиск можно было выполнить только один раз на запуске.

Актор `RemoteLookupProxy` позаботится обо всех этих сценариях. На рис. 6.5 показано местоположение `RemoteLookupProxy` между `RestApi` и `BoxOffice`. Он прозрачно передает сообщения удаленному актору `BoxOffice`.

Актор `RemoteLookupProxy` – это конечный автомат с двумя состояниями: поиск и активная работа (см. листинг 6.12). Для переключения своего метода `receive` между состояниями он использует метод `become`. В состоянии поиска актор `RemoteLookupProxy` пытается получить действительную ссылку `ActorRef` на `BoxOffice`, а в состоянии активной работы передает все полученные сообщения по этой ссылке актору `BoxOffice`. Если `RemoteLookupProxy` обнаруживает, что `BoxOffice` завершился, он вновь переходит в состояние поиска и снова пытается получить действительную ссылку `ActorRef`. Для этого мы будем использовать механизм `DeathWatch`. Может показаться, что это что-то новенькое, но в действительности это часть стандартного механизма мониторинга акторов.

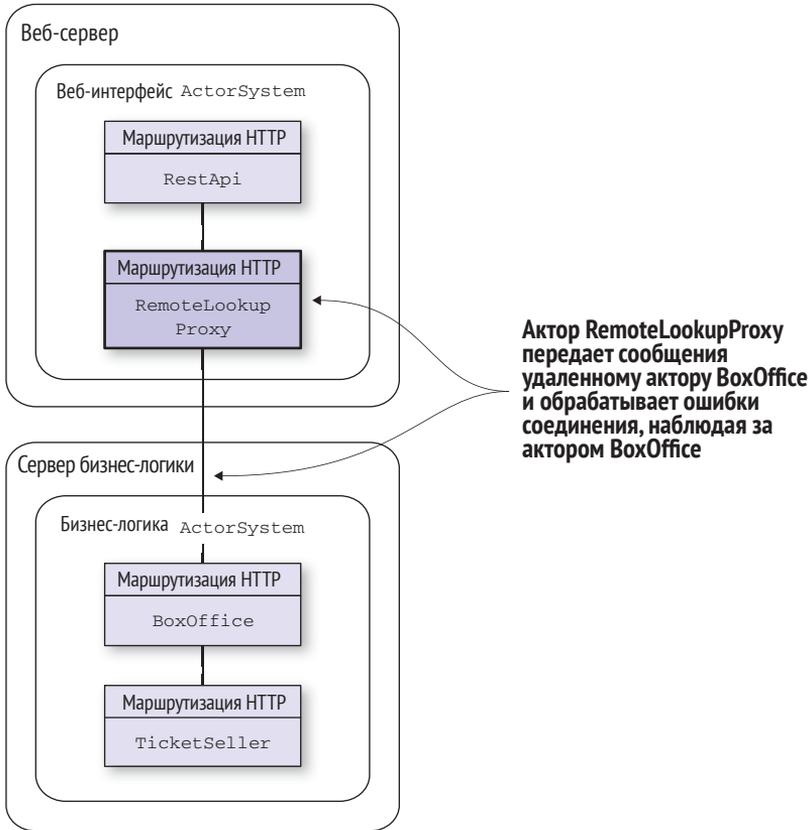


Рис. 6.5. Актор RemoteLookupProxy

Листинг 6.11. Удаленный поиск

```
import scala.concurrent.duration._

class RemoteLookupProxy(path:String) extends Actor with ActorLogging {
  context.setReceiveTimeout(3 seconds)
  sendIdentifyRequest()

  def sendIdentifyRequest(): Unit = {
    val selection = context.actorSelection(path)
    selection ! Identify(path)
  }

  def receive = identify

  def identify: Receive = {
    case ActorIdentity(`path`, Some(actor)) =>
      context.setReceiveTimeout(Duration.Undefined)
  }
}
```

Посылка сообщения ReceiveTimeout, если в течение 3 секунд не было принято ни одного сообщения

Немедленно посылает запрос идентификации актора

Выбирает актор по заданному пути path

Посылка сообщения Identify в actorSelection

Изначально актор находится в состоянии поиска

Актор найден, и ссылка ActorRef на него получена

Не посылать больше сообщений ReceiveTimeout, потому что актор перешел в активное состояние

```

log.info("switching to active state")
context.become(active(actor))
context.watch(actor)

case ActorIdentity(`path`, None) =>
  log.error(s"Remote actor with path $path is not available.")

case ReceiveTimeout =>
  sendIdentifyRequest()

case msg:Any =>
  log.error(s"Ignoring message $msg, not ready yet.")
}

def active(actor: ActorRef): Receive = {
  case Terminated(actorRef) =>
    log.info("Actor $actorRef terminated.")
    context.become(identify)
    log.info("switching to identify state")
    context.setTimeout(3 seconds)
    sendIdentifyRequest()

  case msg: Any => actor forward msg
}

```

← Переход в состояние активной работы  
 ← Наблюдать за завершением удаленного актора  
 ← Актор (пока) недоступен; удаленная система недоступна или не запущена  
 ← Продолжать поиск удаленного актора, если сообщений не поступало  
 ← Не посылать сообщений в состоянии поиска  
 ← Активное состояние  
 ← Если удаленный актор завершился, RemoteLookupProху должен перейти в состояние поиска  
 ← В активном состоянии передавать все другие сообщения удаленному актору

Как видите, для мониторинга локальных и удаленных акторов используется один и тот же API. Простого наблюдения за `ActorRef` достаточно, чтобы получить уведомление в случае завершения актора, находящегося под наблюдением, локального или удаленного. Для определения достижимости удаленного узла Akka использует сложный протокол. Мы рассмотрим его в главе 14. Ссылка `ActorRef` на `BoxOffice` извлекается путем отправки специального сообщения `Identify` в `ActorSelection`. Модуль `remote` системы акторов `backend` отвечает на него сообщением `ActorIdentity`, содержащим `correlationId` и необязательную ссылку `ActorRef` на удаленный актор. В сопоставлении с образцом для `ActorIdentity` мы заключили переменную `path` в обратные кавычки. Это означает, что значение `correlationId` в `ActorIdentify` должно совпадать со значением `path`. Если забыть добавить эти обратные кавычки, будет создана новая переменная `path`, которая получит значение `correlationId` из сообщения.

Это были все изменения, которые нужно внести в приложение `GoTicks.com`, чтобы реализовать переход от архитектуры с одним узлом к архитектуре с двумя узлами. Теперь веб-интерфейс и бизнес-логика могут запускаться по отдельности, при этом веб-интерфейс будет находить актор `BoxOffice` и взаимодействовать с ним, когда тот доступен, или выполнять некоторые другие действия, если недоступен.

Последнее, что осталось сделать, – попробовать запустить объекты `FrontendMain` и `BackendMain`. Для этого откройте два терминала и с помощью `sbt run` запустите класс `Main` проекта. В терминалах должен появиться следующий вывод:

```
[info] ...
[info] ... (сообщения sbt)
[info] ...
```

Multiple main classes detected, select one to run:

```
[1] com.goticks.SingleNodeMain
[2] com.goticks.FrontendMain
[3] com.goticks.BackendMain
```

Enter number:

Выберите `FrontendMain` (число 2) в одном терминале и `BackendMain` (число 3) в другом. Посмотрите, что произойдет, если остановить процесс `sbt`, в котором выполняется `BackendMain`, и вновь запустить его. Проверьте, работает ли приложение с теми же командами `httpie`, как было показано выше; например, команда `http PUT localhost:5000/events event=RHCP nrOfTickets:=10` должна создать мероприятие с 10 билетами, а команда `http GET localhost:5000/ticket/RHCP` – вернуть билет на мероприятие. Если попытаться остановить процесс с бизнес-логикой и запустить его снова, вы должны увидеть в консоли, как `RemoteLookupProxy` переключился в состояние поиска и затем вернулся обратно в активное состояние. Также должно появиться сообщение `Акка` об ошибке подключения к удаленному узлу. Если сообщения о событиях жизненного цикла удаленных взаимодействий вам не нужны, просто отключите их, добавив в конфигурацию следующую строку:

```
remote {
  log-remote-lifecycle-events = off
}
```

Журналирование событий жизненного цикла удаленных взаимодействий включено по умолчанию. Это упрощает поиск проблем, когда вы начинаете использовать модуль `remote` и, например, допускаете незначительную ошибку в определении пути к актору. Вы можете подписаться на получение событий жизненного цикла `remote` с помощью метода `eventStream` системы акторов, о котором подробнее рассказывается в главе 10. Поскольку за удаленными актерами можно наблюдать точно так же, как за локальными, нет никакой необходимости обрабатывать эти сообщения отдельно ради управления соединением.

А теперь еще раз окинем взглядом произведенные изменения:

- добавлен объект `FrontendMain`, выполняющий поиск `BoxOffice` в удаленной системе;
- в объект `FrontendMain` добавлен актор `RemoteLookupProxy`, играющий роль связующего звена между `RestApi` и `BoxOffice`. Он пересылает все полученные сообщения удаленному актору `BoxOffice`. Также он получает ссылку `ActorRef` на `BoxOffice` и осуществляет его мониторинг.

Как отмечалось в начале этого раздела, Akka поддерживает два способа получения ссылок `ActorRef` на удаленные акторы. В следующем разделе мы рассмотрим второй способ – удаленное развертывание.

### 6.2.4. Удаленное развертывание

Удаленное развертывание можно выполнить программно или организовать путем определения настроек в конфигурации. Начнем с более предпочтительного способа: на основе конфигурации. Предпочтительным он считается просто потому, что при его использовании не требуется повторно собирать приложение. Стандартный объект `SingleNodeMain` создает `boxOffice` как актор верхнего уровня:

```
val boxOffice = system.actorOf(Props[BoxOffice], "boxOffice")
```

Локальный путь к этому актору будет иметь вид `/boxOffice`, в котором отсутствует актор-охранник `user`. Когда удаленное развертывание реализуется путем изменения конфигурации, нам достаточно только сообщить системе акторов веб-интерфейса, что при попытке создать актор, имеющий путь `/boxOffice`, он должен создаваться не локально, а удаленно. Необходимые для этого настройки приводятся в листинге 6.12.

**Листинг 6.12.** Настройка `RemoteActorRefProvider`

```
actor {
  provider = "akka.remote.RemoteActorRefProvider"

  deployment {
    /boxOffice {
      remote = "akka.tcp://backend@0.0.0.0:2552"
    }
  }
}
```

Актор с таким путем будет развернут удаленно

Удаленный адрес, где должен быть развернут актор. IP-адрес или имя хоста должны точно совпадать с адресом удаленной системы акторов `backend`

Удаленное развертывание также можно выполнить программно, и мы рассмотрим этот способ для полноты картины. В большинстве случаев предпочтительнее использовать подход на основе конфигурации, но иногда, например если вы ссылаетесь на разные узлы по псевдонимам, раз-

вертывание программным способом может оказаться удобнее. Полностью динамическое удаленное развертывание также предпочтительнее при использовании модуля `akka-cluster`, потому что он создавался специально для поддержки динамического членства. Пример удаленного развертывания программным способом представлен в листинге 6.13.

**Листинг 6.13.** Удаленное развертывание программным способом

```
val uri = "akka.tcp://backend@0.0.0.0:2552"
val backendAddress = AddressFromURIString(uri)
val props = Props[BoxOffice].withDeploy(
  Deploy(scope = RemoteScope(backendAddress))
)
context.actorOf(props, "boxOffice")
```

← Создание адреса к системе акторов backend из URI

← Создание объекта Props с удаленной областью видимости

Код в листинге 6.13 создаст и развернет `BoxOffice` в удаленной системе акторов `backend`. Конфигурационный объект `Props` определяет удаленную область видимости для развертывания.

Важно отметить, что удаленное развертывание не требует от Akka автоматического развертывания фактического файла класса с реализацией актора `BoxOffice` в удаленной системе акторов; код для `BoxOffice` уже должен находиться там, и удаленная система акторов должна быть запущена. Если удаленная система акторов потерпит аварию и перезапустится, ссылка `ActorRef` не изменится автоматически и не будет указывать на новый экземпляр актора. Так как актор развертывается удаленно, он не должен больше запускаться удаленной системой акторов, как предусматривает текущая реализация `BackendMain`. Поэтому мы должны внести пару изменений. Начнем с создания новых классов `Main` для запуска бизнес-логики (`BackendRemoteDeployMain`) и веб-интерфейса (`FrontendRemoteDeployMain`).

**Листинг 6.14.** Главные объекты для запуска бизнес-логики и веб-интерфейса

```
// Главный класс для запуска узла с бизнес-логикой.
object BackendRemoteDeployMain extends App {
  val config = ConfigFactory.load("backend")
  val system = ActorSystem("backend", config)
}
object FrontendRemoteDeployMain extends App
  with Startup {
  val config = ConfigFactory.load("frontend-remote-deploy")
  implicit val system = ActorSystem("frontend", config)

  val api = new RestApi() {
```

← Не создавать актор boxOffice

← Главный класс для запуска веб-интерфейса

```

implicit val requestTimeout = configuredRequestTimeout(config)
implicit def executionContext = system.dispatcher

def createBoxOffice: ActorRef =
  system.actorOf(
    BoxOffice.props,
    BoxOffice.name
  )
}

startup(api.routes)
}

```

← Создание `BoxOffice`, автоматически использует конфигурацию

Если запустить эти классы `Main` в двух разных терминалах, как было показано ранее, и создать несколько мероприятий с помощью `httpie`, в консоли с системой акторов веб-интерфейса вы увидите примерно следующее:

```

// очень длинное сообщение, которое я разделил на несколько строк,
// чтобы уместить по ширине книжной страницы.
INFO [frontend-remote]: Received new event Event(RHCP,10), sending to
Actor[akka.tcp://backend@0.0.0.0:2552/remote/akka.tcp/
frontend@0.0.0.0:2551/user/boxOffice#-1230704641]

```

Оно сообщает, что система акторов веб-интерфейса действительно послала сообщение актору `BoxOffice`, развернутому удаленно. Путь к актору несколько отличается от ожидаемого – в нем присутствует сегмент, указывающий, откуда было выполнено развертывание. Модуль `remote`, действующий на стороне системы акторов с бизнес-логикой, использует эту информацию для возврата ответов в систему акторов с веб-интерфейсом.

В общем и целом наша реализация работает, но в ней кроется одна проблема. Если к моменту, когда веб-интерфейс попытается развернуть актор, удаленная система акторов не будет запущена, попытка, как вы понимаете, потерпит неудачу, но, как бы странно это не выглядело, ссылка `ActorRef` все равно будет создана. И даже если потом удаленная система акторов запустится, созданная ссылка `ActorRef` не будет работать. Это правильное поведение, потому что это уже не тот экземпляр актора, в отличие от случая, обсуждавшегося выше, когда актор сам выполняет перезапуск и ссылка продолжает указывать на повторно созданный экземпляр.

Чтобы что-то предпринять, когда удаленная система акторов или удаленный актор `BoxOffice` терпят аварию, мы должны внести дополнительные изменения. Нужно добавить наблюдение за `boxOfficeActorRef`, как это делалось выше, и предусмотреть действия в ответ на определенные события. Поскольку `RestApi` имеет `val`-ссылку на `boxOffice`, мы должны добавить

промежуточный актор, подобный актору `RemoteLookupProxy`. Назовем этот промежуточный актор `RemoteBoxOfficeForwarder`.

Конфигурацию тоже нужно изменить, потому что теперь `boxOffice` имеет путь `/forwarder/boxOffice` из-за появления промежуточного актора `RemoteBoxOfficeForwarder`. Пути `/boxOffice` в секции `deployment` нужно заменить на `/forwarder/boxOffice`.

Реализация актора `RemoteBoxOfficeForwarder`, который наблюдает за актором, развернутым удаленно, приводится в листинге 6.15.

#### Листинг 6.15. Механизм наблюдения за удаленными акторами

```
object RemoteBoxOfficeForwarder {
  def props(implicit timeout: Timeout) = {
    Props(new RemoteBoxOfficeForwarder)
  }

  def name = "forwarder"
}

class RemoteBoxOfficeForwarder(implicit timeout: Timeout)
  extends Actor with ActorLogging {
  context.setReceiveTimeout(3 seconds)

  deployAndWatch()
  def deployAndWatch(): Unit = {
    val actor = context.actorOf(BoxOffice.props, BoxOffice.name)
    context.watch(actor)
    log.info("switching to maybe active state")
    context.become(maybeActive(actor))
    context.setReceiveTimeout(Duration.Undefined)
  }

  def receive = deploying

  def deploying: Receive = {
    case ReceiveTimeout =>
      deployAndWatch()

    case msg: Any =>
      log.error(s"Ignoring message $msg, remote actor is not ready yet.")
  }

  def maybeActive(actor: ActorRef): Receive = {
    case Terminated(actorRef) =>
      log.info("Actor $actorRef terminated.")
  }
}
```

← Осуществляет удаленное развертывание и наблюдение за `BoxOffice`

← Переключается в состояние «возможно активен» сразу после развертывания актора. Не выполнив поиск, нельзя быть уверенным, что развертывание прошло успешно

← Развернутый `BoxOffice` завершился, поэтому его нужно развернуть повторно

```

log.info("switching to deploying state")
context.become(deploying)
context.setReceiveTimeout(3 seconds)
deployAndWatch()

case msg: Any => actor forward msg
}
}

```

Класс `RemoteBoxOfficeForwarder` выглядит очень похожим на класс `RemoteLookupProxy` из предыдущего раздела – он тоже фактически является конечным автоматом. В данном случае он имеет два состояния: `deploying` и `maybeActive`. Не выполнив поиск актора, мы не можем быть уверены, что удаленный актор действительно был развернут. Добавление поиска удаленного актора с помощью `actorSelection` в класс `RemoteBoxOfficeForwarder` я оставляю вам как самостоятельное упражнение.

Теперь в класс `Main` веб-интерфейса нужно добавить создание `RemoteBoxOfficeForwarder`:

```

object FrontendRemoteDeployWatchMain extends App
  with Startup {
  val config = ConfigFactory.load("frontend-remote-deploy")
  implicit val system = ActorSystem("frontend", config)

  val api = new RestApi() {
    val log = Logging(system.eventStream, "frontend-remote-watch")
    implicit val requestTimeout = configuredRequestTimeout(config)
    implicit def executionContext = system.dispatcher

    def createBoxOffice: ActorRef = {
      system.actorOf(
        RemoteBoxOfficeForwarder.props,
        RemoteBoxOfficeForwarder.name
      )
    }
  }

  startup(api.routes)
}

```

← Создаст промежуточный актор и развернет удаленный `BoxOffice`

Мы создали новый главный класс `FrontendRemoteDeployWatchMain`, содержащий необходимые изменения.

Запустив `FrontendRemoteDeployWatchMain` и `BackendRemoteDeployMain` в двух терминалах, можно увидеть, как осуществляется наблюдение за актором, развернутым удаленно, и как происходит повторное его разверты-

вание после остановки и повторного запуска процесса с бизнес-логикой или когда веб-интерфейс запускается раньше бизнес-логики.

Если, прочитав предыдущий абзац, вы подумали: «ничего особенного», – прочитайте его еще раз. Приложение автоматически развертывает удаленный актор, когда узел, на котором он действует, вновь становится доступным. На самом деле это очень крутая фишка!

На этом мы завершаем раздел об удаленном развертывании. Мы рассмотрели и удаленный поиск, и удаленное развертывание, а также все, что необходимо для обеспечения надежности. Даже в ситуации, когда у вас всего два сервера, важно использовать надежные решения. В обоих примерах, поиска и развертывания, узлы могут запускаться в любом порядке. Удаленное развертывание можно было бы реализовать исключительно изменением конфигурации, но это дало бы нам слишком упрощенное решение, не способное справиться с аварийными ситуациями и требующее определенного порядка запуска.

В следующем разделе мы рассмотрим плагин `multi-JVM` для `sbt` и модуль `akka-multinode-testkit`, которые дают возможность протестировать узлы с веб-интерфейсом и бизнес-логикой в приложении `GoTicks` app.

### 6.2.5. Тестирование с `multi-JVM`

Тестирование акторов в распределенных приложениях существенно сложнее, потому что взаимодействующие акторы часто находятся на разных узлах. В идеале тестирование подобных приложений должно производиться с использованием нескольких узлов. Например, на рис. 6.6 показана схема тестирования нашего веб-интерфейса.

Как можно заметить, в тестировании участвуют две виртуальные машины `JVM`: под управлением одной действует веб-интерфейс и тестовый код, а на другой – сервер с бизнес-логикой, где развертывается `BoxOffice`. Но самое важное: этот способ тестирования требует координации двух `JVM`. Прежде чем развернуться, веб-интерфейс должен получить удаленную ссылку на `BoxOffice` и может запуститься только после запуска `BoxOffice`. Эту задачу координации можно возложить на плагин `multi-JVM` для `sbt`.

Плагин `multi-JVM` для `sbt` дает возможность выполнять тестирование с несколькими `JVM`. Плагин нужно зарегистрировать в `sbt`, в файле `project/plugins.sbt`:

```
resolvers += Classpaths.typesafeResolver

addSbtPlugin("com.eed3si9n" % "sbt-assembly" % "0.13.0")

addSbtPlugin("com.typesafe.sbt" % "sbt-start-script" % "0.10.0")

addSbtPlugin("com.typesafe.sbt" % "sbt-multi-jvm" % "0.3.11")
```

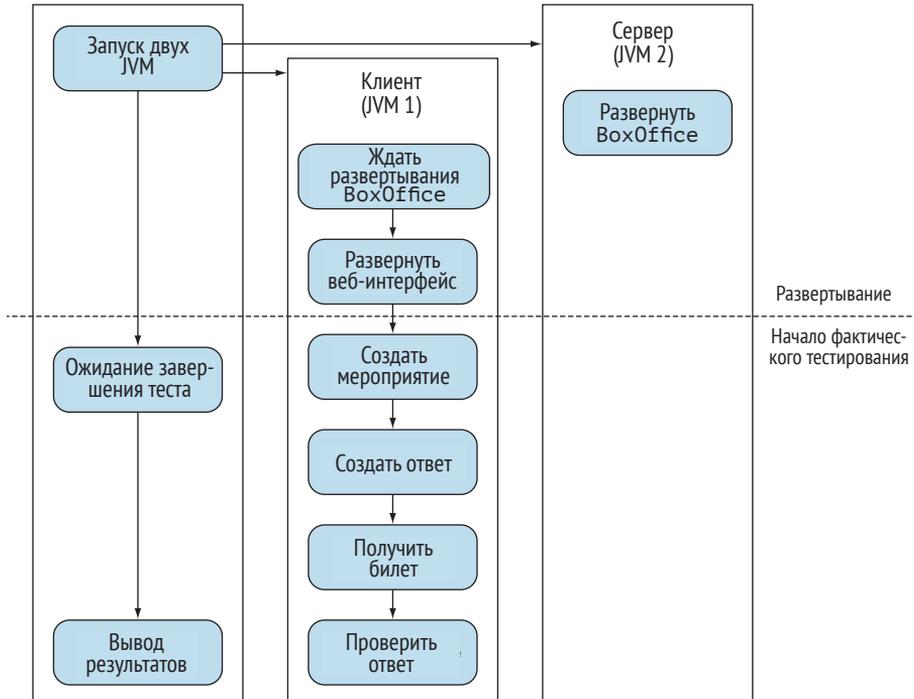


Рис. 6.6. Пример тестирования приложения с удаленными актерами

Также для его использования нужно добавить еще один файл сборки `sbt`. Плагин `multi-JVM` поддерживает только файлы проектов, написанные на языке `Scala`, поэтому в папку `chapter-remoting/project` нужно добавить файл `GoTicksBuild.scala` (листинг 6.16). Утилита `sbt` автоматически объединит файл `build.sbt` и файл в листинге 6.16, а это означает, что мы не должны дублировать зависимости в `GoTicksBuild.scala`.

#### Листинг 6.16. Конфигурация Multi-JVM

```

import sbt._
import Keys._
import com.typesafe.sbt.SbtMultiJvm
import com.typesafe.sbt.SbtMultiJvm.MultiJvmKeys.{ MultiJvm }

object GoTicksBuild extends Build {

  lazy val buildSettings = Defaults.defaultSettings ++
    multiJvmSettings ++
    Seq(
      crossPaths := false
    )
}
  
```

```

lazy val goticks = Project(
  id = "goticks",
  base = file("."),
  settings = buildSettings ++ Project.defaultSettings
) configs(MultiJvm)

lazy val multiJvmSettings = SbtMultiJvm.multiJvmSettings ++
Seq(
  compile in MultiJvm <=<
    (compile in MultiJvm) triggeredBy (compile in Test),
  parallelExecution in Test := false,
  executeTests in Test <=<
    ((executeTests in Test), (executeTests in MultiJvm)) map {
      case (_, testResults), (_, multiJvmResults) =>
        val results = testResults ++ multiJvmResults
        (Tests.overall(results.values), results)
    }
)
}

```

Обеспечивает автоматическую компиляцию тестов вместе с приложением

Запрет параллельного выполнения

Гарантирует выполнение тестов multi-JVM как часть цели тестирования по умолчанию

Если вы не имеете большого опыта использования `sbt`, не переживайте, если что-то в этом файле для вас останется непонятным. Он просто настраивает плагин `multi-JVM` и гарантирует запуск тестов `multi-JVM` вместе с обычными модульными тестами. Если вам интересно поближе познакомиться с `sbt`, прочитайте книгу «SBT in Action» ([www.manning.com/suereth2/](http://www.manning.com/suereth2/)).

Код всех тестов для `multi-JVM` по умолчанию должен находиться в папке `src/multi-jvm/scala`. Теперь, настроив тестирование проекта с помощью `multi-JVM`, можно переходить к созданию модульных тестов для веб-интерфейса и бизнес-логики приложения `GoTicks.com`. Сначала определим объект `MultiNodeConfig`, описывающий роли тестируемых узлов. Мы создали объект `ClientServerConfig`, наследующий `MultiNodeConfig`, который определяет конфигурацию с двумя узлами (клиент и сервер). Определение объекта приводится в листинге 6.17.

#### Листинг 6.17. Описание ролей тестируемых узлов

```

object ClientServerConfig extends MultiNodeConfig {
  val frontend = role("frontend")  ← Роль веб-интерфейса
  val backend = role("backend")    ← Роль сервера с бизнес-логикой
}

```

Здесь определены две роли с говорящими именами: `"frontend"` и `"backend"`. Роли будут использоваться для идентификации узлов в ходе тестирования и выполнения на них определенного кода. Прежде чем пе-

реходить к самим тестам, нужно написать некоторый инфраструктурный код, чтобы связать тесты с фреймворком Scalatest.

**Листинг 6.18.** STMultiNodeSpec связывает тесты с фреймворком Scalatest

```
import akka.remote.testkit.MultiNodeSpecCallbacks
import org.scalatest.{BeforeAndAfterAll, WordSpec}
import org.scalatest.matchers.MustMatchers

trait STMultiNodeSpec extends MultiNodeSpecCallbacks
  with WordSpec with MustMatchers with BeforeAndAfterAll {

  override def beforeAll() = multiNodeSpecBeforeAll()

  override def afterAll() = multiNodeSpecAfterAll()
}
```

Дает доступ к обратным вызовам путем переопределения методов класса TestKit

Остальные трейты, необходимые для тестирования

Заставляет все наши тесты использовать наши версии beforeAll и afterAll

Этот трейт используется для запуска и завершения тестов, выполняющихся на нескольких узлах, и вы можете задействовать его во всех своих таких тестах. Он «подмешивается» в настройки модульного тестирования, которые определяют фактические тесты. В нашем примере мы создадим тест ClientServerSpec. Он содержит довольно много кода, поэтому будем рассматривать его по частям. Первым делом мы должны создать MultiNodeSpec и подмешать трейт STMultiNodeSpec, который только что определили. На двух разных JVM мы должны запустить две разные версии ClientServerSpec. Определения этих двух классов ClientServerSpec показаны в листинге 6.19.

**Листинг 6.19.** Классы с настройками для тестов, выполняющихся на нескольких узлах

```
class ClientServerSpecMultiJvmFrontend extends ClientServerSpec
class ClientServerSpecMultiJvmBackend extends ClientServerSpec

class ClientServerSpec extends MultiNodeSpec(ClientServerConfig)
  with STMultiNodeSpec with ImplicitSender {

  def initialParticipants = roles.size
```

Настройки для JVM с веб-интерфейсом

Настройки для JVM с бизнес-логикой

Настройки для обоих узлов

Количество узлов, участвующих в тестировании

ClientServerSpec использует STMultiNodeSpec и трейт ImplicitSender. Трейт ImplicitSender определяет testActor как отправитель всех сообщений по умолчанию, что дает возможность вызывать expectMsg и другие функции проверки без необходимости каждый раз указывать testActor

как отправителя сообщений. Код в листинге 6.20 демонстрирует, как получить адрес узла с бизнес-логикой.

**Листинг 6.20.** Получение адреса узла с бизнес-логикой

```
import ClientServerConfig._
val backendNode = node(backend)
```

Импорт конфигурации для доступа к роли backend

Вызов `node(role)` вернет адрес узла с ролью backend. В данном случае создается объект `ActorPath`. (Метод `node` должен вызываться из главного потока теста, именно поэтому его результат в тесте присваивается значению `backendNode`)

Узлы с ролями `backend` и `frontend` могут работать с произвольными портами по умолчанию. В тесте `TestRemoteBoxOfficeCreator` замещает `RemoteBoxOfficeCreator`, потому что последний конструирует путь, используя имя хоста, номер порта и имя актора из файла `frontend.conf`. Вместо этого в процессе тестирования мы должны использовать адрес узла с ролью `backend` и искать актор `BoxOffice` на этом узле. Листинг 6.20 обеспечивает это. В листинге 6.21 приводится код для тестирования нашей распределенной архитектуры.

**Листинг 6.21.** Тестирование распределенной архитектуры

```
"A Client Server configured app" must {
  "wait for all nodes to enter a barrier" in {
    enterBarrier("startup")
  }
  "be able to create an event and sell a ticket" in {
    runOn(backend) {
      system.actorOf(BoxOffice.props(Timeout(1 second)), "boxOffice")
      enterBarrier("deployed")
    }
    runOn(frontend) {
      enterBarrier("deployed")
      val path = node(backend) / "user" / "boxOffice"
      val actorSelection = system.actorSelection(path)
      actorSelection.tell(Identify(path), testActor)
      val actorRef = expectMsgPF() {
```

Дать узлам время запуститься

Тестовый сценарий для обоих узлов

Выполнить код в этом блоке в JVM с бизнес-логикой

Сообщить, что система акторов с бизнес-логикой развернута

Создать актор `BoxOffice` с именем `boxOffice`, чтобы класс `RemoteLookupProxy` смог найти его

Ждать развертывания узла с бизнес-логикой

Выполнить код в этом блоке в JVM с веб-интерфейсом

Получить ссылку на удаленный актор `BoxOffice`

Послать сообщение `Identify` выбранному актору

```

    case ActorIdentity(`path`, Some(ref)) => ref
  }

import BoxOffice._

actorRef ! CreateEvent("RHCP", 20000)

expectMsg(EventCreated(Event("RHCP", 20000)))

actorRef ! GetTickets("RHCP", 1)

expectMsg(Tickets("RHCP", Vector(Ticket(1))))
}

enterBarrier("finished")
}
}

```

← Ждать, пока `BoxOffice` сообщит о своей доступности. Класс `RemoteLookupProxy` выполнит все необходимое, чтобы получить ссылку `ActorRef` на `BoxOffice`

← Ждать сообщений, как это принято в `TestKit`

← Признак завершения теста

Этот тест можно разбить на три части. Сначала, вызывая `enterBarrier("startup")` на обоих узлах, он ждет, пока они запустятся. Далее модульный тест делится на фрагменты, один из которых выполняется на узле с веб-интерфейсом (`frontend`), а другой – на узле с бизнес-логикой (`backend`). Узел `frontend` ждет, пока узел `backend` сообщит, что развернулся, после чего выполняется собственно тестирование. В заключение модульный тест ждет, пока все узлы завершатся вызовом `enterBarrier("finished")`.

Узел `backend` просто запускает актор `BoxOffice`, который используется со стороны узла `frontend`. Мы могли бы использовать HTTP-клиента и послать запрос через интерфейс `RestAPI`, но в данном случае есть возможность обратиться к `BoxOffice` непосредственно.

После этого можно, наконец, протестировать взаимодействия между узлами. Для проверки ожидаемых сообщений можно использовать те же методы, что использовались в главе 3. Этот тест `multi-JVM` можно запустить командой `multi-jvm:test` в `sbt`; попробуйте.

На рис. 6.7 показан фактический порядок выполнения этого теста. Обратите внимание, что координация узлов выполняется автоматически, плагином `multi-JVM`. Чтобы написать свой код, реализующий нечто подобное, пришлось бы немало потрудиться.

Проект в папке `chapter-remoting` также включает модульный тест для версии приложения, выполняющейся на одном узле, и, кроме некоторых инфраструктурных настроек, он почти полностью повторяет представленный выше. Пример теста `multi-JVM`, представленный в этом разделе, показывает, как приложение, первоначально созданное для работы на одном узле, можно адаптировать для выполнения на двух узлах. Основное отличие между версиями для одного и двух узлов заключается в получении ссылки на актор, находящийся в удаленной системе. Добавление удален-

ного поиска между RestApi и boxOffice дало нам некоторую гибкость и возможность восстанавливаться после аварии. Такое разделение поставило перед нами интересную проблему в тесте: как дождаться развертывания BoxOffice на удаленном узле и получить ссылку ActorRef на него? Мы решили ее с помощью actorSelection и сообщения Identity.

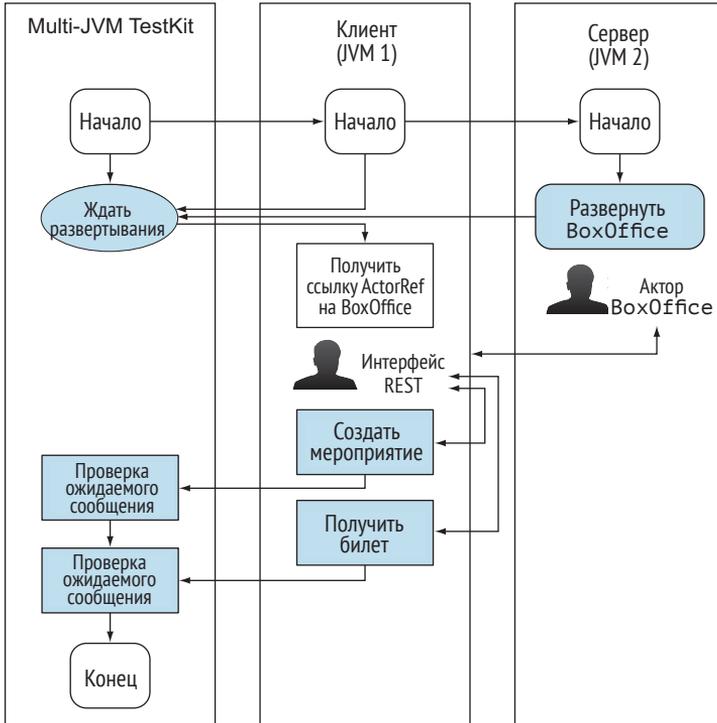


Рис. 6.7. Порядок выполнения теста с помощью multi-JVM

На этом мы завершаем первое знакомство с модулем `multi-node-testkit`. В следующей главе мы продолжим его исследование. Предыдущий тест показал, как можно протестировать приложение `GoTicks.com` в распределенном окружении. В данном случае оно выполнялось в двух JVM на одной физической машине. Но, как будет показано в главе 13, `multi-node-testkit` также можно использовать для тестирования приложений, выполняющихся на нескольких физических машинах.

## 6.3. В заключение

Помните, как в начале главы мы объясняли, почему нельзя просто щелкнуть переключателем и превратить наше приложение в распределенное? Это объясняется наличием обстоятельств, которые нельзя не учитывать в сетевом мире, но которые локальные приложения легко могут игнорировать. Как и предполагалось, многое из того, что мы сделали в этой главе, в

конечном счете свелось к учету этих обстоятельств, и, как мы и говорили, фреймворк Akka упростил нам эту задачу.

Да, нам пришлось внести некоторые изменения, но многое осталось неизменным:

- мы воспользовались тем обстоятельством, что ссылки ActorRef ведут себя одинаково при работе с удаленными и локальными акторами;
- для определения факта завершения актора в распределенной системе используется тот же прикладной интерфейс, что и в локальной;
- несмотря на то что теперь акторы RestApi и BoxOffice разделены сетью, мы смогли обеспечить прозрачность взаимодействий между ними, используя прием перенаправления (RemoteLookupProxy и RemoteBoxOfficeForwarder).

Это очень важно, потому что избавляет от необходимости изучать что-то новое, чтобы поднять приложение на новый уровень; основные операции остались теми же, что свойственно для хорошо продуманных инструментов.

В этой главе вы также узнали кое-что новое:

- оболочка REPL предлагает простой и удобный способ опробования приложений в распределенном окружении с любой топологией на выбор;
- модуль multi-node-testkit делает тестирование распределенных систем акторов невероятно простой задачей, независимо от того, основаны они на модуле akka-remote, akka-cluster или даже на обоих. (Akka обладает уникальными инструментами для модульного тестирования в распределенных окружениях.)

Мы намеренно не касались ошибок в RemoteLookupProxy и RemoteBoxOfficeForwarder, связанных с потерей сообщений, когда узел с бизнес-логикой оказывается недоступным. В следующих главах мы восполним этот пробел и расскажем:

- какие средства можно использовать для увеличения надежности обмена сообщениями между узлами (глава 10);
- как изменить приложение GoTicks.com, чтобы избавиться от ошибки потери состояния в акторах TicketSeller при аварийном завершении узла с бизнес-логикой (глава 14);
- как распределить хранение состояния в кластере (глава 14).

Но эти решения не нужны для понимания Akka на базовом уровне и будут освещены позже. Чтобы создать законченное приложение, вам понадобятся некоторые вспомогательные функции для выполнения настроек, журналирования и развертывания приложения. О них мы поговорим в следующей главе.

# Глава 7

## Настройка, журналирование и развертывание

В этой главе:

- использование библиотеки управления конфигурацией;
- журналирование событий на уровне приложения и отладочных сообщений;
- упаковка и развертывание приложений на основе Akka.

До сих пор основное внимание мы уделяли созданию и работе с системами акторов. Чтобы создать по-настоящему работающее приложение, мы должны связать его с другими пакетами, дабы подготовить его к развертыванию в другой системе. В этой главе мы с вами сначала посмотрим, как в Akka организована поддержка настройки приложений. Затем исследуем приемы журналирования, в том числе с использованием своих средств. И в заключение рассмотрим пример развертывания.

### 7.1. Настройка

По умолчанию в Akka используется библиотека Typesafe Config Library, обладающая набором самых современных возможностей, в том числе возможностью разными способами определять свойства и затем ссылаться на них в коде (одна из задач поддержки настроек как раз и состоит в том, чтобы дать возможность во время выполнения использовать переменные, определяемые за пределами кода). Существуют также развитые средства объединения конфигурационных файлов на основе простых соглашений, которые определяют, как происходит переопределение одноименных параметров. Одним из важнейших требований системы настроек является возможность организации конфигураций для разных окружений (напри-

мер, для разработки, тестирования, промышленной эксплуатации). Далее вы увидите, как это делается.

### 7.1.1. Попытка настройки Акка

Многие библиотеки в Акка стремятся минимизировать число необходимых зависимостей, и Typesafe Config Library – не исключение; она вообще не имеет зависимостей от других библиотек. Для начала коротко рассмотрим приемы ее использования.

Библиотека использует иерархию свойств. На рис. 7.1 изображен пример конфигурации приложения, определяющей четыре свойства в виде иерархии. Мы сгруппировали все свойства в общий узел с именем MyApp1 и два свойства – во вложенный узел database.

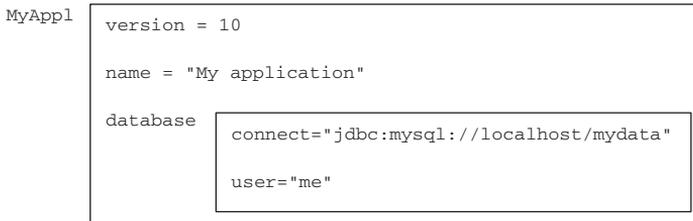


Рис. 7.1. Пример конфигурации

Получить конфигурацию можно с помощью объекта ConfigFactory. Эта операция обычно выполняется в классе Main приложения. Библиотека поддерживает также возможность определить, какой файл конфигурации использовать, но подробнее об этом мы поговорим в следующих разделах, а пока просто будем использовать файл по умолчанию.

#### Листинг 7.1. Загрузка конфигурации

```
val config = ConfigFactory.load()
```

Библиотека поддерживает несколько форматов определения конфигурации, поэтому по умолчанию она будет искать разные файлы в следующем порядке:

- *application.properties* – должен содержать определения параметров настройки в формате файла свойств Java;
- *application.json* – должен содержать определения параметров настройки в формате JSON;
- *application.conf* – должен содержать определения параметров настройки в формате HOCON; этот формат основан на JSON, но удобнее для чтения человеком; более подробную информацию о формате

НОСОН или о Typesafe Config Library можно найти по адресу: <https://github.com/typesafehub/config>.

Есть возможность использовать все эти файлы сразу. Далее мы будем применять файл *application.conf*, представленный в листинге 7.2.

### Листинг 7.2. application.conf

```
MyAppl {
  version = 10
  description = "My application"
  database {
    connect="jdbc:mysql://localhost/mydata"
    user="me"
  }
}
```

← Вложение выполняется простой группировкой с помощью { }

Для простых приложений этого файла обычно бывает достаточно. Его формат выглядит очень похожим на JSON. Главное его преимущество – простота чтения, он позволяет легко увидеть структуру группировки свойств. JDBC – отличный пример свойств, необходимых большинству приложений, которыми удобнее пользоваться, когда они объединены в отдельную группу. В мире внедрения зависимостей вы наверняка предпочтете группировать внедряемые свойства по компонентам (такие как *DataSource*). А теперь посмотрим, как можно использовать эти свойства.

Есть несколько методов получения значений разных типов, и все они поддерживают точку (.) как разделитель в пути к свойству. Поддерживаются не только простые типы, но и списки этих типов.

### Листинг 7.3. Получение свойств

```
val applicationVersion = config.getInt("MyAppl.version")
val databaseConnectionString = config.getString("MyAppl.database.connect")
```

← Так можно получить строку connect из вложенного узла database, заключенную в { } в предыдущем листинге

Некоторым объектам не нужно много настроек. Например, класс *DBaseConnection*, создающий соединение с базой данных, может требовать лишь строки подключения и имени пользователя. Когда классу *DBaseConnection* передается весь комплект настроек, он должен знать полный путь к каждому требуемому свойству. Но если класс *DBaseConnection* повторно использовать в другом приложении, возникнет проблема. Путь к свойствам в конфигурации начинается с *MyAppl*; в другом приложении начало пути может отличаться. Эту проблему можно решить, если использовать метод извлечения поддерева конфигурации, как показано в листинге 7.4.

**Листинг 7.4.** Извлечение поддерева конфигурации

```
val databaseCfg = configuration.getConfig("MyAppl.database")
val databaseConnectionString = databaseCfg.getString("connect")
```

← Получить поддерево по имени, используемое прикладным кодом

← Затем в DBaseConnection извлечь свойство относительно корня поддерева

В этом случае вместо полной конфигурации классу DBaseConnection передается только поддерево databaseCfg, и теперь ему не нужно знать полные пути к свойствам, достаточно последних их компонентов – имен самих свойств. То есть теперь DBaseConnection можно повторно использовать в разных приложениях, не боясь столкнуться с проблемой путей к свойствам.

Также есть возможность использовать подстановку, когда какое-то свойство несколько раз используется в конфигурации, как, например, имя хоста в строке подключения к базе данных.

**Листинг 7.5.** Подстановка

```
hostname="localhost"
MyAppl {
  version = 10
  description = "My application"
  database {
    connect="jdbc:mysql://${hostname}/mydata"
    user="me"
  }
}
```

← Определение простой переменной без объявления типа (но обратите внимание на кавычки)

← Знакомый синтаксис подстановки \${}

Переменные в конфигурационных файлах часто используются для объявления имени приложения или номеров версий, потому что многократное повторение одной и той же информации в разных местах в файле может привести к ошибкам из-за опечаток. В операциях подстановки также можно использовать системные свойства и переменные окружения.

**Листинг 7.6.** Подстановка значения переменной окружения

```
hostname=${?HOST_NAME}
MyAppl {
  version = 10
  description = "My application"
  database {
    connect="jdbc:mysql://${hostname}/mydata"
    user="me"
```

← Знак вопроса (?) указывает, что значение извлекается из переменной окружения

```
}
}
```

Но в этом случае возникает другая проблема: никогда заранее неизвестно, существует ли данное свойство или переменная. Для ее решения можно воспользоваться возможностью переопределения. Подстановка системного свойства или переменной окружения будет просто проигнорирована, если это свойство или переменная не определены, как в случае с переменной `HOST_NAME` в листинге 7.7.

**Листинг 7.7.** Подстановка системного свойства или переменной окружения вместо значения по умолчанию

```
hostname="localhost"
hostname=${?HOST_NAME}
MyAppl {
  version = 10
  description = "My application"
  database {
    connect="jdbc:mysql://${hostname}/mydata"
    user="me"
  }
}
```

← Если переменная окружения существует, ее значение затрет значение по умолчанию

← Сначала определяется значение по умолчанию

Легко догадаться, что здесь происходит.

А теперь поговорим о значениях по умолчанию. Они играют важную роль в конфигурациях, потому что помогают избежать пользователя от лишних настроек. Кроме того, часто так бывает, что приложения должны работать без определенных настроек, пока не окажутся в эксплуатационном окружении; в окружении разработки часто бывает достаточно значений по умолчанию.

### 7.1.2. Использование значений по умолчанию

Продолжим наш простой пример конфигурации JDBC. Обычно можно с полной уверенностью предположить, что разработчики будут подключаться к базе данных на своей машине, ссылаясь на нее по имени хоста `localhost`. Когда нам понадобится провести демонстрацию возможностей приложения, нам нужно запустить приложение на каком-то другом компьютере и организовать доступ к демонстрационной базе данных, хранящейся, возможно, на другой машине. Самое простое, что можно сделать в этом случае, – скопировать конфигурационный файл целиком и изменить значения некоторых параметров. Но проблема такого решения – в том, что теперь все наши настройки хранятся в двух местах. Правильнее было бы переопределить два-три параметра, значения которых будут

отличаться в новом окружении. Механизм поддержки значений по умолчанию позволяет легко достичь этого. Библиотека Typesafe Config Library содержит механизм возврата к значениям по умолчанию, находящимся в объекте с конфигурацией, который затем передается механизму извлечения конфигурации. На рис. 7.2 показан простой пример такой ситуации.

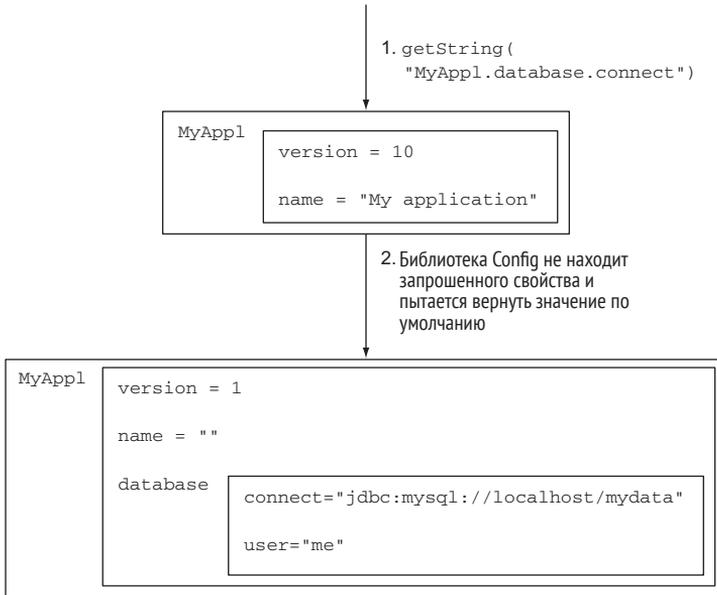


Рис. 7.2. Настройки по умолчанию

### Предотвращение появления пустых свойств

Механизм значений по умолчанию помогает предотвратить ошибки, когда значения отличаются в зависимости от места их использования. Согласно этому принципу, при чтении конфигурационного свойства его значение всегда должно быть установлено. Если бы фреймворк допускал пустые свойства, код мог бы вести себя по-разному, в зависимости от того, как (и где) происходит настройка. Поэтому попытка получить отсутствующее конфигурационное значение вызовет исключение.

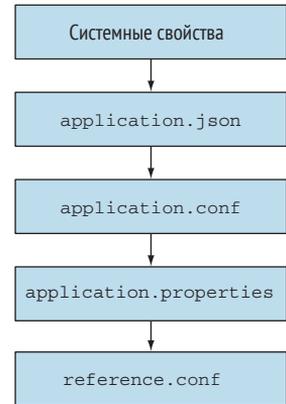
Такой способ возврата к значениям по умолчанию дает нам большую гибкость. Но, чтобы получить возможность вернуться к значениям по умолчанию, нужно прежде определить их. Умолчания настраиваются в файле *reference.conf*, находящемся в корне JAR-файла; исходя из идеи, что каждая библиотека будет определять свои умолчания. Библиотека конфигурации

отыщет все файлы *reference.conf* и интегрирует их содержимое в общую структуру умолчаний. При таком подходе все свойства, необходимые библиотеке, всегда будут иметь значения по умолчанию и принцип отсутствия пустых значений будет соблюден. (Далее вы увидите, что значения по умолчанию можно также определять программным способом.)

Выше уже говорилось, что библиотека конфигурации поддерживает несколько форматов. То есть ничто не мешает нам использовать несколько форматов в одном приложении. Каждый файл будет использоваться как запасной вариант другого. И для поддержки переопределения параметров настройки с помощью системных свойств в них есть возможность определять переменные верхнего уровня. Все файлы имеют одинаковую структуру, поэтому порядок переопределения значений почти всегда один и тот же. На рис. 7.3 показано, в каком порядке библиотека конфигурации просматривает файлы для формирования полного дерева.

Большинство приложений будет использовать только один из этих файлов *application*. Но если у вас появится желание настроить набор значений по умолчанию и затем переопределять некоторые из них, как в случае с настройками JDBC, вы сможете сделать это. Следуя данной схеме, можно организовать переопределение параметров более приоритетными конфигурациями, как показано на рис. 7.3.

По умолчанию конфигурация извлекается из файла с именем *application.{conf,json,properties}*. Есть два способа изменить имя файла. Первый – использовать перегруженный метод `load` объекта `ConfigFactory`, которому можно передать имя конфигурационного файла, как показано в листинге 7.8.



**Рис. 7.3.** Приоритет разных источников информации. Источники выше имеют более высокий приоритет и переопределяют значения, объявленные ниже

#### Листинг 7.8. Изменение имени конфигурационного файла

```
val config = ConfigFactory.load("myapp") ← Явно указать имя конфигурационного файла
```

В этом случае вместо *application.{conf,json,properties}* будет использоваться конфигурационный файл *myapp.{conf,json,properties}*. (Это может пригодиться при необходимости иметь несколько конфигураций в единственной JVM.)

Другой вариант – использовать системные свойства Java. Иногда такое решение оказывается самым простым, потому что позволяет написать сценарий на Bash, определяющий набор свойств, который будет исполь-

зоваться приложением (это лучше, чем «копаться» в файлах JAR или WAR для извлечения файлов из них). В следующем списке перечислены три системных свойства, которые можно использовать для управления выбором конфигурационного файла:

- `config.resource` определяет имя ресурса, но не базовое имя, а имя с расширением, например `application.conf`, а не `application`;
- `config.file` определяет путь к конфигурационному файлу, также с расширением;
- `config.url` определяет URL.

Системные свойства можно использовать для определения имени конфигурационного файла, когда используется метод `load` без аргументов. Определить системные свойства можно с помощью ключа `-D`; например `-Dconfig.file="config/myapp.conf"`, чтобы использовать конфигурационный файл `config/myapp.conf`. При использовании этих свойств правило по умолчанию поиска файлов с разными расширениями – `.conf`, `.json` и `.properties` – не действует.

### 7.1.3. Настройка Акка

Итак, вы увидели, как с помощью библиотеки конфигурации организовать настройку свойств приложения, но как быть, если понадобится изменить какие-то параметры самого фреймворка Акка? Как Акка использует эту библиотеку? Приложение может создавать разные системы акторов с разными настройками. В отсутствие конфигурации будет создана система акторов с параметрами по умолчанию, как показано в листинге 7.9.

**Листинг 7.9.** Конфигурация по умолчанию

```
val system = ActorSystem("mySystem")
```

← В данном случае для создания конфигурации будет использован метод `ConfigFactory.load()` без аргумента

В вызов конструктора `ActorSystem` также можно (и иногда полезно) явно передать конфигурацию, как показано в листинге 7.10.

**Листинг 7.10.** Использование конкретной конфигурации

```
val configuration = ConfigFactory.load("myapp")
val systemA = ActorSystem("mysystem", configuration)
```

← Сначала загрузить конфигурацию по имени

← Затем передать ее конструктору `ActorSystem`

Внутри приложения конфигурация сохраняется в свойстве `settings` системы акторов `ActorSystem`. Она доступна в любом акторе. В листинге 7.11 показано, как получить свойство `MyAppL.name`.

**Листинг 7.11.** Доступ к конфигурации во время выполнения

```

val mySystem = ActorSystem("myAppl")
val config = mySystem.settings.config
val applicationDescription = config.getString("MyAppl.name")

```

После создания системы акторов конфигурация доступна в ее свойстве settings

Получить свойство из конфигурации можно как обычно

Теперь вы знаете, как использовать библиотеку конфигурации для определения своих свойств и как с помощью той же библиотеки организовать настройку системы акторов Akka. В этих первых разделах предполагалось, что имеется только одно приложение Akka в данной системе. В следующем разделе мы обсудим настройку нескольких систем акторов, действующих на одной машине.

### 7.1.4. Настройка для нескольких систем

В зависимости от требований иногда могут понадобиться разные конфигурации, например для нескольких подсистем, действующих на одном экземпляре (или машине). Фреймворк Akka предлагает несколько решений для этого. Сначала рассмотрим случаи с несколькими JVM, выполняющимися в одном окружении и использующими одни и те же файлы. Первый вариант – применение системных свойств – уже описан выше. Второй вариант: при запуске нового процесса использовать другой конфигурационный файл. Но часто бывает так, что большая часть конфигурации для всех подсистем остается неизменной – изменяется только малая ее часть. В такой ситуации задачу можно решить с помощью операции включения. Рассмотрим простой пример. Пусть имеется файл *baseConfig.conf*, как показано в листинге 7.12.

**Листинг 7.12.** *baseConfig.conf*

```

MyAppl {
  version = 10
  description = "My application"
}

```

Для примера будем предполагать, что номер версии для всех подсистем будет оставаться неизменным, но каждая будет иметь свое, уникальное описание.

**Листинг 7.13.** *subAppl.conf*

```

include "baseConfig"
MyAppl {
  description = "Sub Application"
}

```

Только имя подключаемого конфигурационного файла, без расширения

Определение нового описания

Поскольку базовая конфигурация включается перед остальными настройками, значение параметра `description` переопределяется, как в случае с единственным файлом. То есть у вас может быть одна общая базовая конфигурация и конфигурационные файлы для конкретных подсистем, содержащие определения только отличающихся параметров.

Но как быть, если подсистемы выполняются в одной и той же JVM? В этом случае не получится использовать системные свойства для чтения других конфигурационных файлов. Как организовать настройку в такой ситуации? Мы уже обсудили это: можно загрузить конфигурацию, используя имя приложения. Также можно использовать метод `include` для группировки всех конфигураций с одинаковым именем. Единственный недостаток – конфигурационных файлов может оказаться несколько. Если это проблема, можно использовать другое решение, основанное на возможности объединения деревьев конфигурации с использованием механизма значений по умолчанию.

Для начала объединим две конфигурации в одну.

#### Листинг 7.14. `combined.conf`

```
MyAppl {
  version = 10
  description = "My application"
}

subApp1A {
  MyAppl {
    description = "Sub application"
  }
}
```

Подняв это поддерево, мы получим общее свойство (`version`) и переопределим `description`

Суть приема состоит в том, чтобы получить поддерево в разделе `subApp1A` конфигурации и вставить его в начало конфигурационной цепочки. Это называется *подъемом конфигурации*, потому что при этом пути к параметрам сокращаются. На рис. 7.4 показано, как это делается.

Запросив свойство `MyAppl.description` вызовом `config.getString("MyAppl.description")`, мы должны получить `"Sub application"`, потому что значение `description` установлено в конфигурации на более конкретном уровне (`subApp1A.MyAppl.description`). А запросив свойство `MyAppl.version`, мы должны получить значение `10`, потому что параметр `version` не определен на более конкретном уровне (внутри `subApp1A.MyAppl`). То есть в этом случае должен сработать обычный механизм значений по умолчанию. В листинге 7.15 показано, как загрузить конфигурацию, чтобы получить подъем и значения по умолчанию. Обратите внимание, что подстановка значений

по умолчанию выполняется программно (мы не полагаемся на соглашения, описанные выше).

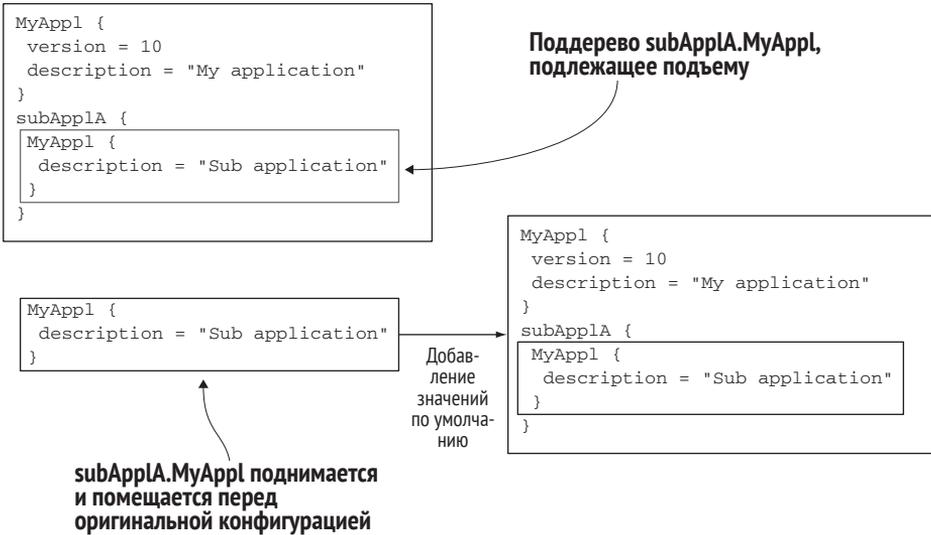


Рис. 7.4. Подъем раздела конфигурации

**Листинг 7.15.** Подъем и добавление значений по умолчанию

```

val configuration = ConfigFactory.load("combined")
val subApp1ACfg = configuration.getConfig("subApp1A")
val config = subApp1ACfg.withFallback(configuration)
    
```

| Выбрать поддереву subApp1A  
 |  
 | Добавить значения по умолчанию из базовой конфигурации

Настройка – важная часть процесса развертывания любого приложения. Сначала требования к настройке невелики и легко удовлетворяются, но с течением времени неизменно возникают потребности, способные усложнить структуру конфигурации, из-за чего она часто становится непрос-той и запутанной. Библиотека Typesafe Config Library включает несколько мощных инструментов, помогающих преодолевать эти сложности:

- простота определения значений по умолчанию на основе соглашений;
- широкие возможности поддержки значений по умолчанию, позволяющие сократить объемы конфигурации;
- поддержка разных синтаксисов для определения конфигураций: традиционный, Java, JSON и HOCON.

Мы далеко не полностью исчерпали тему конфигурации, но рассмотрели достаточно, чтобы вы смогли справиться с широким спектром типичных

требований, возникающих при развертывании решений на основе Akka. В следующем разделе мы затронем проблему журналирования, не менее важную для разработчиков, стремящихся использовать то, что им удобнее. Мы посмотрим, как организовать журналирование в Akka посредством простой конфигурации.

## 7.2. Журналирование

Другая функция, необходимая в каждом приложении, – запись сообщений в файл журнала. Каждый из нас имеет свои предпочтения в выборе библиотеки для журналирования, поэтому в состав Akka входит адаптер, поддерживающий любые фреймворки журналирования и минимизирующий зависимости от сторонних библиотек. Так же как настройка, журналирование делится на две задачи: журналирование на уровне приложения и на уровне фреймворка Akka (что может пригодиться, например, для отладки). Далее мы повторим пройденный путь и сначала посмотрим, как осуществляется журналирование на уровне фреймворка Akka.

### 7.2.1. Журналирование в приложении Akka

Так же как в обычном приложении на Java или Scala, внутри актора необходимо создать экземпляр адаптера для журналирования, который будет помещать сообщения в журнал.

#### Листинг 7.16. Создание адаптера для журналирования

```
class MyActor extends Actor {
  val log = Logging(context.system, this)
  ...
}
```

В первую очередь обратите внимание, что для организации журналирования необходима система акторов ActorSystem, потому что журналирование отделено от фреймворка. Для отправки журналируемых сообщений в eventHandler адаптер использует системный объект eventStream, который является механизмом Akka (описывается ниже). eventHandler принимает сообщение и использует настроенный фреймворк журналирования для его записи. Благодаря этому все акторы получают поддержку журналирования, но только один напрямую зависит от конкретной реализации фреймворка журналирования. Фреймворк, используемый в eventHandler, может настраиваться. Другое преимущество состоит в том, что журналирование – это всегда ввод/вывод, а ввод/вывод, как известно, работает медленно, и в конкурентном окружении это может отрицательно сказаться на производительности из-за необходимости ждать, пока другой поток

закончит запись своих сообщений. Поэтому в высокопроизводительных приложениях ожидание – крайне нежелательный аспект. При использовании поддержки журналирования, встроенной в Akka, актерам не приходится ждать. В листинге 7.17 приводится конфигурация, необходимая для создания механизма журналирования по умолчанию.

#### Листинг 7.17. Настройка eventHandler

```
akka {
  # Обработчик событий для регистрации на этапе запуска
  # (Logging$DefaultLogger выводит сообщения в STDOUT)
  loggers = ["akka.event.Logging$DefaultLogger"]
  # Варианты: ERROR, WARNING, INFO, DEBUG
  logLevel = "DEBUG"
}
```

Этот eventHandler не использует сторонний фреймворк журналирования – все сообщения он выводит в STDOUT. В Akka имеются две реализации eventHandler. Первая – по умолчанию выводит сообщения в STDOUT – уже представлена. Вторая – использует SLF4J. Она находится в файле *akka-slf4j.jar*. Чтобы задействовать этот обработчик, добавьте следующие параметры в свой файл *application.conf*.

#### Листинг 7.18. Настройка eventHandler для использования SLF4J

```
akka {
  loggers = ["akka.event.slf4j.Slf4jLogger"]
  # Варианты: ERROR, WARNING, INFO, DEBUG
  logLevel = "DEBUG"
}
```

Когда вывода в STDOUT и SLF4J недостаточно, можно создать свой адаптер eventHandler. Для этого нужно определить актер, обрабатывающий несколько видов сообщений. Пример такого актера приводится в листинге 7.19.

#### Листинг 7.19. Собственный адаптер eventHandler

```
import akka.event.Logging.InitializeLogger
import akka.event.Logging.LoggerInitialized
import akka.event.Logging.Error
import akka.event.Logging.Warning
import akka.event.Logging.Info
import akka.event.Logging.Debug

class MyEventListener extends Actor{
```

```
def receive = {
  case InitializeLogger(_) =>
    sender ! LoggerInitialized
  case Error(cause, logSource, logClass, message) =>
    println( "ERROR " + message)
  case Warning(logSource, logClass, message) =>
    println( "WARN " + message)
  case Info(logSource, logClass, message) =>
    println( "INFO " + message)
  case Debug(logSource, logClass, message) =>
    println( "DEBUG " + message)
}
```

При получении этого сообщения выполняется инициализация обработчика, по завершении которой отправителю посылается сообщение `LoggerInitialized`

Обработка сообщения об ошибке. Сюда можно добавить логику для фильтрации сообщений, если используемый фреймворк не поддерживает этого

Обработка сообщения с предупреждением

Обработка отладочного сообщения

Обработка информационного сообщения

Это очень простой пример, он демонстрирует лишь протокол обработки сообщений. В действующих приложениях этот актор будет иметь более сложный вид.

## 7.2.2. Использование журналирования

Вернемся к процедуре создания экземпляра адаптера журналирования в Akka, которая была показана в листинге 7.16. Мы обсудили первую часть процесса создания (`ActorSystem`). Как вы помните, там был еще второй параметр, как показано в листинге 7.20.

**Листинг 7.20.** Создание адаптера для журналирования (повтор)

```
class MyActor extends Actor {
  val log = Logging(context.system, this)
  ...
}
```

Второй параметр `Logging` используется как источник канала журналирования, в данном случае это экземпляр класса. Объект-источник преобразуется в строку, которая идентифицирует его. Преобразование выполняется в соответствии со следующими правилами:

- если это `Actor` или `ActorRef`, используется путь к актору;
- если это строка, используется сама строка;
- если это класс, используется результат вызова его метода `simpleName`.

Для удобства можно также использовать трейт `ActorLogging`, чтобы добавить член `log` в актор. Этот трейт был создан специально, чтобы упростить организацию поддержки журналирования, как показано в листинге 7.21.

**Листинг 7.21.** Создание адаптера для журналирования

```
class MyActor extends Actor with ActorLogging {
  ...
}
```

Адаптер также поддерживает шаблонные символы в сообщениях. Они избавляют от необходимости проверять уровни журналирования. Если вы создаете сообщения с помощью операции конкатенации, эта работа будет выполняться всегда, даже если выбранный уровень журналирования не позволит вывести сообщение в журнал! Использование шаблонных символов избавит от необходимости проверять уровень журналирования (например, `if (logger.isDebugEnabled())`), и сообщение будет создаваться, только если выбранный уровень журналирования соответствует уровню важности сообщения. Шаблон – это строка `{}` сообщения, как показано в листинге 7.22.

**Листинг 7.22.** Использование шаблонов

```
log.debug("two parameters: {}, {}", "one", "two")
```

Здесь нет ничего необычного; большинство из вас, кому приходилось заниматься реализацией журналирования в программах на Java или других языках на основе JVM, знакомы с этим синтаксисом.

Еще одна проблема, которая может вызывать сложности, – управление журналированием, осуществляемым различными инструментами или библиотеками, которыми пользуется ваше приложение. В следующем разделе мы покажем, как решить эту задачу в Akka.

### 7.2.3. Управление журналированием из Akka

При разработке сложных приложений часто требуется организовать вывод отладочных сообщений на очень низком уровне. Akka поддерживает возможность вывода сообщений, когда происходят определенные внутренние события или когда обрабатываются определенные сообщения. Этот механизм предназначен для разработчиков и не должен использоваться в промышленной эксплуатации. К счастью, благодаря архитектуре, представленной выше, вам не придется беспокоиться о плохой или, хуже того, полной несовместимости Akka с выбранным фреймворком журналирования. Akka поддерживает простую возможность настройки, которая позволит вам выбирать, какая информация должна выводиться в журнал. В листинге 7.23 показаны настройки, манипулируя которыми, вы сможете управлять подробностью журналируемой информации из Akka.

### Листинг 7.23. Конфигурационный файл с настройками журналирования событий в Акка

```

akka {
  # для всех последующих возможностей нужно установить уровень DEBUG
  loglevel = DEBUG
  # Уровень журналирования устанавливается в момент запуска ActorSystem.
  # Данный адаптер выводит сообщения в stdout (System.out).
  # Варианты: OFF, ERROR, WARNING, INFO, DEBUG
  stdout-loglevel = "WARNING"
  # Когда система акторов запускается, информация о конфигурации
  # журналируется с уровнем INFO. Это помогает устранить сомнения
  # о выбранной конфигурации.
  log-config-on-start = on
  debug {
    # журналировать все пользовательские сообщения, обрабатываемые
    # акторами, которые используют функцию akka.event.LoggingReceive,
    # журналирование выполняется с уровнем DEBUG
    receive = on
    # включить журналирование с уровнем DEBUG всех автоматически
    # принимаемых сообщений
    # (Kill, PoisonPill и пр.)
    autoreceive = on
    # включить журналирование с уровнем DEBUG событий жизненного
    # цикла акторов
    # (restarts, deaths и пр.)
    lifecycle = on
    # включить журналирование с уровнем DEBUG всех событий LoggingFSM,
    # переходов и таймеров
    fsm = on
    # включить журналирование с уровнем DEBUG всех изменений в подписках
    # (subscribe/unsubscribe) в eventStream
    event-stream = on
  }
  remote {
    # Если имеет значение "on", Акка будет журналировать все исходящие
    # сообщения с уровнем DEBUG, иначе журналирование будет отключено
    log-sent-messages = on
    # Если имеет значение "on," Акка будет журналировать все входящие
    # сообщения с уровнем DEBUG, иначе журналирование будет отключено
    log-received-messages = on
  }
}

```

← Для журналирования сообщений, получаемых акторами; требуется использовать `akka.event.LoggingReceive` при обработке сообщений

Комментарии достаточно четко поясняют назначение каждого параметра (обратите внимание на примечание к параметру `receive` и дополнительные требования к нему). Обратите также внимание, что вы защище-

ны от одного из главных недостатков, вынуждающего конфигурировать фреймворки или инструменты: от необходимости знать, какие пакеты изменяют уровень журналирования. Это еще одно преимущество систем, основанных на обмене сообщениями. В листинге 7.24 показано, как использовать `LoggingReceive` в акторе, чтобы обеспечить журналирование всех пользовательских сообщений, получаемых актором.

**Листинг 7.24.** Использование `LoggingReceive`

```
class MyActor extends Actor with ActorLogging {
  def receive = LoggingReceive {
    case ... => ...
  }
}
```

Добавление трейта `ActorLogging`, определяющего функцию `LoggingReceive`, чтобы получить возможность журналировать принимаемые сообщения

Если теперь присвоить свойству `akka.debug.receive` значение "on", все сообщения, принимаемые актором, будут выводиться в журнал.

И снова мы не можем похвастаться, что полностью исчерпали тему журналирования, но показали достаточно, чтобы вам было от чего оттолкнуться и вы смогли обоснованно выбрать между подходом, используемым в Акка, или каким-то другим. Журналирование – важный инструмент, особенно для систем, основанных на обмене сообщениями, где процедура пошаговой отладки, как в обычных приложениях, часто невозможна. В следующем разделе мы обсудим последнюю тему этой главы: собственно развертывание.

## 7.3. Развертывание приложений на основе акторов

Выше вы увидели, как использовать систему акторов и сами акторы для настройки и журналирования. Но для создания приложения требуется больше: все, входящее в состав приложения, нужно упаковать вместе и развернуть после запуска системы. В этом разделе мы посмотрим, как создать дистрибутив приложения. Этот простой пример покажет вам основные принципы создания дистрибутивов.

Для создания дистрибутива автономного приложения мы используем плагин `sbt-native-packager`. Начнем с актора `HelloWorld`. Этот актор просто принимает сообщения и отвечает на них сообщением `hello`.

**Листинг 7.25.** Актор `HelloWorld`

```
class HelloWorld extends Actor with ActorLogging {
  def receive = {
    case msg: String =>
      val hello = "Hello %s".format(msg)
```

Использовать трейт `ActorLogging`, чтобы получить возможность журналировать принимаемые сообщения

```

    sender() ! hello
    log.info("Sent response {}",hello)
  }
}

```

Теперь создадим актор, посылающий сообщения актору HelloWorld. Назовем его HelloWorldCaller.

#### Листинг 7.26. Актор HelloWorldCaller

```

class HelloWorldCaller(timer: FiniteDuration, actor: ActorRef)
  extends Actor with ActorLogging {

  case class TimerTick(msg: String)

  override def preStart() {
    super.preStart()
    implicit val ec = context.dispatcher
    context.system.scheduler.schedule(
      timer,
      timer,
      self,
      new TimerTick("everybody"))
  }

  def receive = {
    case msg: String => log.info("received {}",msg)
    case tick: TimerTick => actor ! tick.msg
  }
}

```

Этот актор использует встроенный планировщик для отправки самому себе сообщений `TimerTick` через регулярные интервалы времени. Получив это сообщение, он пересылает его по ссылке `actor`, переданной в конструктор (в данном случае ссылка указывает на актор `HelloWorld`). В свою очередь, актор `HelloWorld` записывает в журнал любые, полученные им строковые сообщения.

Чтобы получить работающее приложение, нужно в момент запуска развернуть систему акторов.

#### Листинг 7.27. BootHello

```

import akka.actor.{ Props, ActorSystem }
import scala.concurrent.duration._

object BootHello extends App {

```

```

val system = ActorSystem("hellokernel")
val actor = system.actorOf(Props[HelloWorld])
val config = system.settings.config
val timer = config.getInt("helloWorld.timer")
system.actorOf(Props(
  new HelloWorldCaller(
    timer millis,
    actor)))
}

```

← Создание ActorSystem  
 ← Создание актора HelloWorld  
 ← Получить величину задержки из конфигурации  
 ← Создание актора Caller  
 ← Создать задержку Duration из целого числа. Это возможно благодаря импортированию `scala.concurrent.duration._`  
 ← Передать ссылку на актор HelloWorld в HelloWorldCaller

Теперь нужно создать дополнительные ресурсы, необходимые для нормальной работы приложения. Во-первых, добавим конфигурационный файл, определяющий время задержки по умолчанию.

#### Листинг 7.28. reference.conf

```

helloWorld {
  timer = 5000
}

```

По умолчанию величина задержки равна 5000 миллисекунд. Чтобы файл *reference.conf* попал в JAR-архив, его нужно поместить в каталог *main/resources*.

Далее добавим файл *application.conf* с настройками журналирования.

#### Листинг 7.29. application.conf

```

akka {
  loggers = ["akka.event.slf4j.Slf4jLogger"]
  # Варианты: ERROR,WARNING,INFO, DEBUG
  loglevel = "DEBUG"
}

```

Теперь у нас есть все необходимое и можно переходить к созданию дистрибутива. В этом примере дистрибутив будет создаваться с помощью плагина *sbt-native-packager*. Чтобы плагин *sbt-native-packager* включил конфигурационные файлы в дистрибутив, нужно поместить файлы *application.conf* и *logback.xml* в каталог `<каталог_проекта>/src/universal/conf`.

Следующий шаг – подключить *sbt-native-packager* в файле *plugins.sbt*, находящемся в каталоге `<каталог_проекта>/project`.

#### Листинг 7.30. project/plugins.sbt

```

addSbtPlugin("com.typesafe.sbt" % "sbt-native-packager" % "1.0.0")

```

Последний шаг – создание файла сборки *sbt* для нашего проекта.

**Листинг 7.31.** build.sbt

```

name := "deploy"

version := "0.1-SNAPSHOT"

organization := "manning"

scalaVersion := "2.11.8"

enablePlugins(JavaAppPackaging)

scriptClasspath += "./conf"

libraryDependencies ++= {
  val akkaVersion = "2.4.9"
  Seq(
    "com.typesafe.akka" %% "akka-actor"      % akkaVersion,
    "com.typesafe.akka" %% "akka-slf4j"     % akkaVersion,
    "ch.qos.logback"    % "logback-classic" % "1.0.13",
    "com.typesafe.akka" %% "akka-testkit"   % akkaVersion % "test",
    "org.scalatest"    %% "scalatest"      % "2.2.6"      % "test"
  )
}

```

Цель - получить автономное приложение

Добавить каталог conf в путь поиска классов. Если этого не сделать, файлы application.conf и logback.xml не будут найдены

Определение зависимостей приложения

**Простой инструмент сборки: дополнительные сведения**

В какой-то момент вы, без сомнения, пожелаете больше узнать об инструменте sbt. В первую очередь за дополнительной информацией следует обращаться к официальной документации, которая доступна в репозитории проекта (<https://github.com/sbt/sbt>). Кроме того, издательство Manning Publications недавно выпустило книгу «SBT in Action», которая подробно рассказывает не только о том, как пользоваться этим инструментом, но и о том, как он действует.

Теперь мы определили проект sbt и готовы создать дистрибутив. В листинге 7.32 показано, как запускается sbt и выполняется команда dist этого инструмента.

**Листинг 7.32.** Создание дистрибутива

```

sbt
[info] Loading global plugins from home\.sbt\0.13\plugins
[info] Loading project definition from

```

```

\github\akka-in-action\chapter-conf-deploy\project
[info] Set current project to deploy (in build
      file:/github/akka-in-action/chapter-conf-deploy/)
> stage

```

← Когда sbt завершит загрузку, введите stage и нажмите Return

sbt создаст дистрибутив в каталоге *target/universal.stage*. В этом каталоге присутствуют три подкаталога:

- *bin* – содержит сценарии запуска: один для Windows и один для Unix;
- *lib* – содержит все JAR-файлы, от которых зависит приложение;
- *conf* – содержит все конфигурационные файлы для приложения.

Теперь, после создания дистрибутива, нам осталось только запустить приложение. Так как мы дали своему приложению имя *deploy*, команда *stage* создала два сценария запуска: один для Window и один для Unix-подобных систем.

### Листинг 7.33. Запуск приложения

```
deploy.bat
```

```
./deploy
```

Если после этого заглянуть в файл журнала, можно увидеть, что актер *HelloWorld* каждые пять секунд получает сообщения от актора *HelloWorld-Caller*, а *HelloWorldCaller*, в свою очередь, получает ответы от *HelloWorld*. Конечно, это приложение не имеет никакой практической пользы. Но на его примере мы показали, насколько просто создается законченный дистрибутив приложения.

## 7.4. В заключение

Как и многое другое в разработке, реализация развертывания кажется простой задачей. Однако на практике она часто превращается в водоворот конфигурирования каждого компонента из собственных ресурсов без применения какого-то общего системного подхода. Так же как во всех конструктивных решениях, Akka стремится предложить самые современные инструменты, поддерживающие соглашения, простые в реализации. Благодаря этим инструментам мы без особого труда подготовили к запуску наше первое приложение. Но, что особенно важно, вы получили новые знания, которые помогут вам преодолеть трудности в более сложных ситуациях:

- соглашения об именовании файлов, определяющие порядок переопределения конфигурационных значений;

- удобный механизм значений по умолчанию, помогающий определить большую часть значений, необходимых приложениям;
- полный контроль над внедрением конфигураций;
- современные средства журналирования с применением адаптера и единственной точки зависимости;
- простота сборки приложений;
- использование инструмента сборки, конструирующего дистрибутивы приложений и подготавливающего их для запуска.

Время, когда на должность инженера по выпуску назначался самый прилежный член команды, уходит в небытие. Как вы увидите далее в книге, рост сложности приложений вовсе не вызывает роста сложности их сборки и развертывания. В этом огромная заслуга фреймворка Акка: он предоставляет не только мощную среду выполнения со встроенной поддержкой обмена сообщениями и конкуренции, но также средства, позволяющие получать действующие решения быстрее, чем менее мощные прикладные окружения, к которым вы, возможно, привыкли.

# Глава 8

## Шаблоны структуризации акторов

В этой главе:

- конвейеры и фильтры для последовательной обработки;
- параллельная обработка массивов данных дроблением с последующим объединением результатов;
- список получателей: компонент дробления;
- агрегатор: компонент объединения;
- маршрутизация: динамическое управление конвейерами и фильтрами.

Одной из наиболее актуальных задач программирования на основе акторов является организация кода, позволяющая нескольким актерам одновременно работать над решением общей задачи, параллельно выполняя свои единицы работы. Кроме параллельной обработки, иногда возникают ситуации, когда определенные этапы могут выполняться только после других. На примере реализации некоторых классических приемов интеграции мы покажем, как Akka позволяет использовать эти решения, сохраняя врожденные преимущества параллелизма.

Основное внимание в этой главе будет уделяться архитектурным шаблонам, чтобы показать разные способы структурирования приложений и связывания акторов для решения задач.

Для начала рассмотрим простой шаблон конвейеров и фильтров. Он по умолчанию используется в большинстве систем передачи сообщений и отличается простотой реализации. Его классическая версия используется в последовательных вычислениях, но мы адаптируем этот шаблон для работы в конкурентной архитектуре, основанной на обмене сообщениями. Затем мы перейдем к шаблону параллельной обработки массивов

данных дроблением с последующим объединением результатов, который помогает организовать распараллеливание вычислений. Реализации этих шаблонов на основе акторов отличаются не только компактностью и эффективностью, но и отсутствием многих деталей реализации, которые в большинстве случаев неизбежно просачиваются в шаблоны обмена сообщениями.

В заключение мы рассмотрим менее распространенный шаблон маршрутизации. Шаблон динамической маршрутизации основан на применении шаблона конвейеров и фильтров и обеспечивает возможность определения маршрутов между несколькими задачами в начале обработки сообщений.

## 8.1. Конвейеры и фильтры

Идея конвейера предполагает возможность передачи результатов работы одного процесса другому для дополнительной обработки. Многим из нас знакома идея конвейеров по опыту работы в Unix, где она зародилась. И многие из нас воспринимают конвейеры как средство последовательной обработки данных. Тем не менее, как будет показано далее, независимые операции в конвейерах часто могут выполняться параллельно. В этом разделе мы сначала определим область применения данного шаблона, а потом посмотрим, как он реализуется в Akka.

### 8.1.1. Шаблон: конвейеры и фильтры

Во многих системах единственное событие запускает целую последовательность задач. Возьмем, к примеру, камеру, следящую за соблюдением скоростного режима на дороге. Она измеряет скорость и делает фотографию. Но, прежде чем отправить данные в центр обработки, она выполняет ряд дополнительных проверок. Если на фотографии отсутствует номерной знак, система не сможет продолжить обработку, поэтому такая фотография просто удаляется. Фотография также удаляется, если измеренная скорость ниже максимально допустимой на этом участке дороги. Это означает, что в центр для обработки отправляются только фотографии автомобилей, превысивших скорость и на которых виден номерной знак. Возможно, вы уже представили, как мы применим шаблон конвейеров и фильтров в этом примере: ограничения являются фильтрами, а соединения между ними – конвейерами (см. рис. 8.1).

Каждый фильтр состоит из трех частей: входящего конвейера, посредством которого принимаются сообщения, процессора сообщений и исходящего конвейера, посредством которого отсылаются результаты обработки (см. рис. 8.2).



Рис. 8.1. Пример конвейеров и фильтров

Два конвейера на рис. 8.2 изображены как частично находящиеся за пределами фильтра, потому что исходящий конвейер фильтра проверки номера одновременно является входящим конвейером для фильтра проверки скорости. Конвейеры накладывают важное ограничение: каждый фильтр должен принимать и посылать одни и те же сообщения, потому что исходящий конвейер фильтра может быть входящим конвейером любого другого фильтра в шаблоне. То есть все фильтры должны иметь один и тот же интерфейс, включая входящий и исходящий конвейеры. Благодаря этому легко можно добавлять новые виды обработки, изменять порядок их выполнения или исключать. Так как фильтры имеют один и тот же интерфейс и действуют независимо, в них не придется ничего менять, чтобы добавить дополнительные конвейеры.

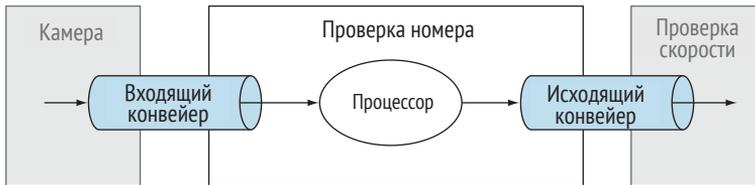


Рис. 8.2. Три части фильтра

### 8.1.2. Конвейеры и фильтры в Акка

Фильтры – это отдельные модули системы обработки сообщений, то есть в контексте Акка фильтры реализуются как акторы. Благодаря автоматической поддержке передачи сообщений вам остается только определить нужные акторы, а конвейеры у вас уже имеются. Казалось бы, эту модель легко реализовать в Акка. Но так ли это на самом деле? Не совсем. Шаблон конвейеров и фильтров предъявляет два важных требования к реализации: все фильтры должны иметь один и тот же интерфейс и быть независимыми. То есть все сообщения, принимаемые разными акторами, должны быть одинаковыми, потому что являются частью интерфейса фильтров, как показано на рис. 8.3. Если сообщения будут разными, тогда следующий актор в цепочке должен иметь другой интерфейс, а это нарушит требование единообразия, что помешает вам свободно применять фильтры.



Рис. 8.3. Сообщения, посылаемые разными акторами

Учитывая требование к одинаковому интерфейсу, оба актора должны принимать и посылать одни и те же сообщения.

Давайте реализуем небольшой пример с сообщением Photo и двумя фильтрами: LicenseFilter и SpeedFilter.

#### Листинг 8.1. Конвейер с двумя фильтрами

```

case class Photo(license: String, speed: Int)
class SpeedFilter(minSpeed: Int, pipe: ActorRef) extends Actor {
  def receive = {
    case msg: Photo =>
      if (msg.speed > minSpeed)
        pipe ! msg
  }
}
class LicenseFilter(pipe: ActorRef) extends Actor {
  def receive = {
    case msg: Photo =>
      if (!msg.license.isEmpty)
        pipe ! msg
  }
}

```

← Сообщение, обрабатываемое фильтрами

← Фильтрует фотографии с автомобилями, скорость которых ниже установленного предела

← Фильтрует фотографии с автомобилями без номерных знаков

В этих акторах-фильтрах нет ничего особенного. Мы уже использовали подобные акторы и сообщения в разделе 2.1.2 и в других примерах. Но, так как два актора обрабатывают и посылают сообщения одного типа, мы можем объединить их в конвейер, в котором один будет передавать свои результаты другому, при этом порядок следования фильтров не имеет значения. В следующем примере мы покажем, насколько гибким и удобным является такое решение, когда обнаруживается, что порядок оказывает заметное влияние на время обработки.

**Листинг 8.2.** Тестирование конвейера

```

val endProbe = TestProbe()
val speedFilterRef = system.actorOf(
  Props(new SpeedFilter(50, endProbe.ref)))
val licenseFilterRef = system.actorOf(
  Props(new LicenseFilter(speedFilterRef)))
val msg = new Photo("123xyz", 60)
licenseFilterRef ! msg
endProbe.expectMsg(msg)

licenseFilterRef ! new Photo("", 60)
endProbe.expectNoMsg(1 second)

licenseFilterRef ! new Photo("123xyz", 49)
endProbe.expectNoMsg(1 second)

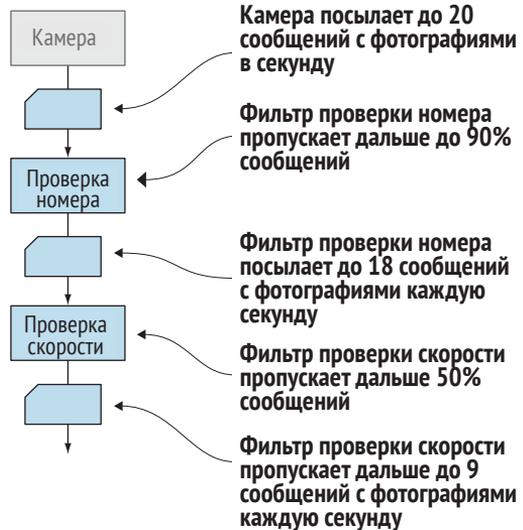
```

← Создание конвейера  
 ← Тестовое сообщение, обрабатываемое обоими фильтрами  
 ← Тестовое сообщение без номерного знака  
 ← Тестовое сообщение с низкой скоростью

Фильтр проверки номерного знака использует большой объем ресурсов. Он должен отыскать на фотографии номерной знак с буквами и цифрами, потребляя для этого значительный объем вычислительных ресурсов. Если камера установлена на дороге с оживленным движением, наша цепочка фильтров может не справляться с потоком поступающих фотографий. Проведя исследования, мы обнаружили, что 90% сообщений благополучно преодолевают фильтр проверки номера и только 50% преодолевают фильтр проверки скорости.

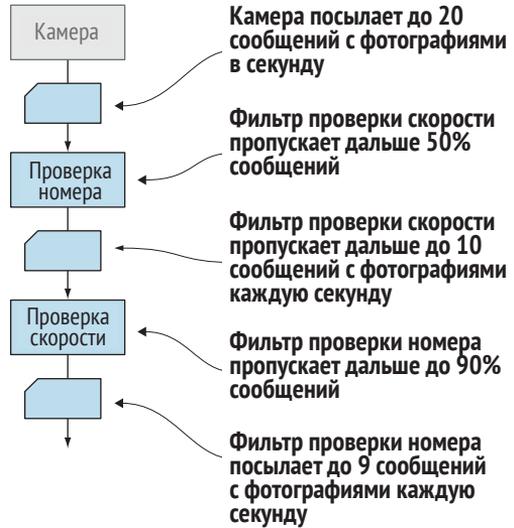
В данном примере (изображен на рис. 8.4) на вход фильтра проверки номера подается до 20 сообщений в секунду. Чтобы увеличить производительность, можно поменять фильтры местами. Поскольку большинство сообщений отбраковываются фильтром проверки скорости, нагрузку на фильтр проверки номера можно существенно снизить.

Как показано на рис. 8.5, если поменять порядок обработки, фильтр проверки номера будет получать до 10 сообщений с фотографиями в секунду; простое переупорядочение уменьшило нагрузку на фильтр впло-



**Рис. 8.4.** Количество сообщений, обрабатываемых каждым фильтром в исходной конфигурации

вину. А так как фильтры имеют одинаковые интерфейсы и действуют независимо, мы легко можем поменять их местами без изменения функциональности или кода. В решении, не использующем шаблон конвейеров и фильтров, нам пришлось бы изменить оба компонента. Но благодаря применению шаблона мы можем просто поменять фильтры местами при конструировании цепочки акторов в момент запуска, а этот этап легко можно сделать настраиваемым.



**Рис. 8.5.** Количество сообщений, обрабатываемых каждым фильтром в измененной конфигурации

### Листинг 8.3. Измененный порядок фильтров

```
val endProbe = TestProbe()
val licenseFilterRef = system.actorOf(
  Props(new LicenseFilter(endProbe.ref)))
val speedFilterRef = system.actorOf(
  Props(new SpeedFilter(50, licenseFilterRef)))
val msg = new Photo("123xyz", 60)
speedFilterRef ! msg
endProbe.expectMsg(msg)

speedFilterRef ! new Photo("", 60)
endProbe.expectNoMsg(1 second)

speedFilterRef ! new Photo("123xyz", 49)
endProbe.expectNoMsg(1 second)
```

← Создание конвейера с другим порядком фильтров

Как видите, с функциональной точки зрения порядок следования фильтров не имеет значения; конвейер дает один и тот же результат в обоих случаях. Такая гибкость является сильной стороной данного шаблона. В нашем примере мы использовали фактические фильтры, но этот шаблон можно расширить; он не ограничивается только фильтрами. Если этапы процесса обработки принимают и посылают сообщения одного типа и не зависят друг от друга, к ним можно применить этот шаблон.

В следующем разделе вы увидите шаблон, действующий по принципу «разделяй и властвуй», который использует преимущества конкурентного выполнения. И снова фреймворк Akka помогает упростить реализацию. Мы будем распределять работу между некоторым количеством акторов и затем объединять результаты в один результат. Такой подход позволит потребителю просто послать запрос и получить ответ.

## 8.2. Параллельная обработка дроблением с последующим объединением результатов

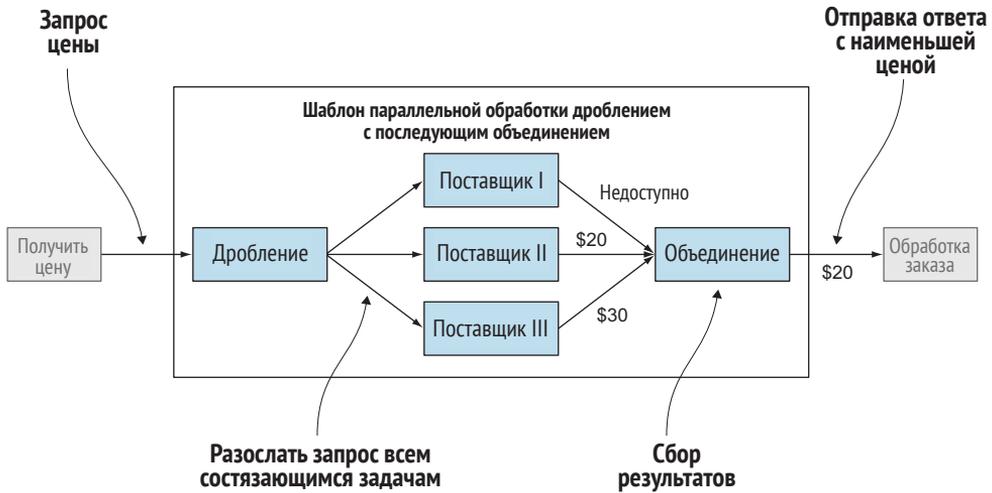
В предыдущем разделе мы создали конвейер из обрабатывающих модулей, выполняющихся последовательно. Часто бывает желательно, чтобы подобные модули выполнялись параллельно. Далее мы рассмотрим шаблон параллельной обработки дроблением с последующим объединением результатов и посмотрим, как он помогает добиться этого. Врожденная способность Akka асинхронно распределять работу между акторами дает нам почти все, что нужно для реализации этого шаблона. Обрабатывающие модули (фильтры в предыдущем примере) – это собираемые элементы; список получателей – компонент дробления. Для сборки общего результата мы используем `Aggregator` – актер Akka.

### 8.2.1. Область применения

Шаблон параллельной обработки дроблением можно использовать в двух разных сценариях. Первый – когда модули обработки функционально идентичны, но компонент, осуществляющий сборку, выбирает только один результат. Второй сценарий – когда общая задача разбивается (дробится) на мелкие подзадачи, которые затем выполняются параллельно, и результат каждого модуля включается в общий результат агрегатором. Далее мы подробно рассмотрим преимущества применения этого шаблона в обоих сценариях.

#### Состязающиеся задачи

Для начала рассмотрим следующую задачу. Клиент покупает что-то, например книгу, в интернет-магазине, но на складе такой книги нет, и магазин должен заказать эту книгу у поставщика. Но он ведет дела с тремя поставщиками и хотел бы приобрести книгу по самой низкой цене. Наша система должна проверить доступность товара, заказанного клиентом, и его цену. Система должна проверить каждого поставщика и из полученных результатов выбрать результат с наименьшей ценой. На рис. 8.6 показано, как шаблон параллельной обработки дроблением с последующим объединением результатов может помочь в этом.



**Рис. 8.6.** Шаблон параллельной обработки дроблением с последующим объединением результатов для состояющихся задач

Сообщение с параметрами заказа клиента передается трем процессам, каждый из которых проверяет доступность товара и цену у определенного поставщика. Процесс объединения собирает результаты проверки и передает дальше только сообщение с наименьшей ценой (в данном примере \$20). Все задачи выполняют одно и то же – получают цену товара, но могут делать это разными способами, в зависимости от особенностей связи с поставщиками. На языке шаблона такие задачи называются *состояющимися*, то есть используется только лучший результат. В нашем примере лучшим считается результат с меньшей ценой, но вообще критерий лучше/хуже может быть каким угодно. Выбор в компоненте объединения результатов не всегда делается на основе содержимого сообщения. Также возможно, что вам нужен лишь самый быстрый ответ, и тогда состязание будет завершаться с получением первого результата. Например, время сортировки списка сильно зависит от используемого алгоритма и начального состояния списка. Когда производительность имеет критически важное значение, сортировку можно выполнить параллельно, используя разные алгоритмы. В Akka для этого можно было бы создать отдельные акторы, выполняющие пузырьковую сортировку, быструю сортировку и, например, древовидную сортировку. Все они будут возвращать одинаковые отсортированные списки, но в зависимости от состояния исходного списка один из акторов справится с задачей быстрее. В такой ситуации объединяющий компонент выберет первое принятое сообщение и сообщит другим акторам прекратить работу. Это еще один пример применения шаблона параллельной обработки дроблением с последующим объединением результатов.

## Параллельная обработка

Другой случай применения шаблона параллельной обработки дроблением – когда одна большая задача делится на множество более мелких подзадач. Вернемся к нашему примеру с дорожной камерой. Допустим, что при обработке фотографии требуется извлекать разную информацию, например время создания фотографии, скорость автомобиля, и добавлять ее в сообщения `Photo`. Оба действия не зависят друг от друга, а значит, могут выполняться параллельно. Когда обе задачи выполнятся, их результаты должны быть объединены в одно сообщение, содержащее время и скорость. На рис. 8.7 показано применение шаблона параллельной обработки дроблением для решения этой задачи.

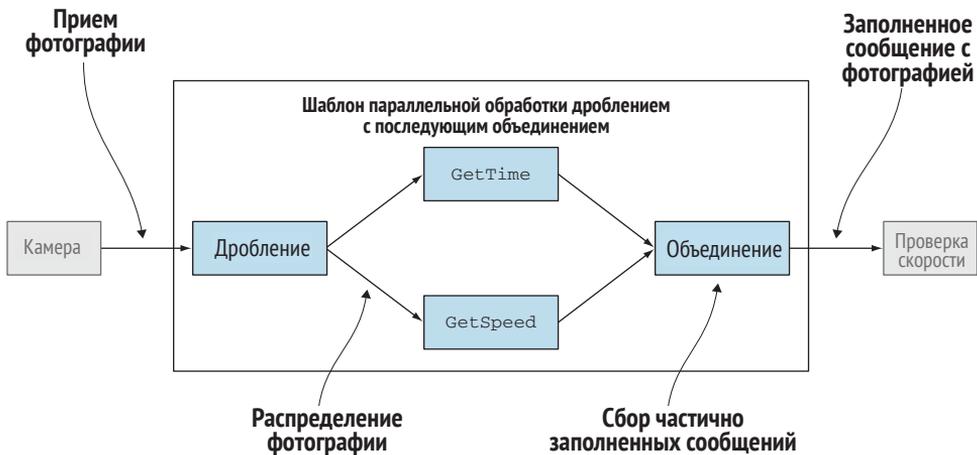


Рис. 8.7. Шаблон параллельной обработки дроблением для распараллеливания задачи

Шаблон начинается с распределения сообщений между несколькими модулями обработки: `GetTime` и `GetSpeed`. Результаты обоих модулей требуется объединить в одно сообщение, которое затем можно передать дальше.

### 8.2.2. Распараллеливание задач в Акка

Посмотрим, как с помощью акторов Акка можно реализовать шаблон параллельной обработки дроблением во втором сценарии. Возьмем за основу пример с дорожной камерой. Каждый компонент этого шаблона реализуется как отдельный актор. В данном примере мы используем один тип сообщений, который поддерживается всеми компонентами. По окончании обработки каждый компонент добавляет данные в сообщение одного типа. Требование независимости компонентов не всегда выполнимо. Это означает лишь, что компоненты нельзя переупорядочить. Но все остальные преимущества, связанные с добавлением и удалением компонентов, остаются на месте.

Определим сообщение, которое будем использовать в примере. Оно принимается и посылается всеми компонентами в примере:

```
case class PhotoMessage(id: String,
    photo: String,
    creationTime: Option[Date] = None,
    speed: Option[Int] = None)
```

Для нашего примера мы симитируем работу камеры и средств распознавания изображений, просто подавая на вход готовые сообщения. Обратите внимание, что изображение имеет идентификационный номер, который может использоваться агрегатором для сборки окончательных сообщений. К числу других атрибутов принадлежат время и скорость; первоначально они имеют пустые значения и заполняются задачами `GetSpeed` и `GetTime`. Наш следующий шаг: реализовать два обрабатывающих модуля, `GetTime` и `GetSpeed`, как на рис. 8.8.

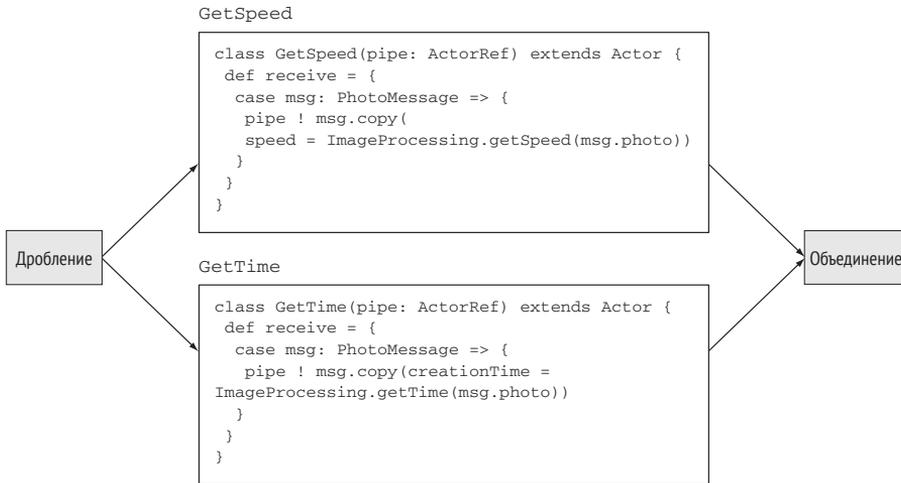


Рис. 8.8. Два обрабатывающих модуля: `GetTime` и `GetSpeed`

Два актора имеют одинаковую организацию, отличаясь только атрибутами, извлекаемыми из изображения. Эти акторы выполняют фактическую работу. Но нам нужен еще актор, выполняющий дробление, который будет передавать изображения для обработки. В следующем разделе для дробления мы используем прием, который называется *список получателей*; а для объединения воспользуемся *агрегатором*.

### 8.2.3. Реализация компонента дробления с использованием списка получателей

Когда на вход шаблона параллельной обработки дроблением подается сообщение `PhotoMessage`, компонент дробления пересылает его для даль-

нейшей обработки акторам `GetTime` и `GetSpeed`. Мы используем простейшую версию компонента дробления – *список получателей*. (Распределение сообщений можно реализовать разными способами; подойдет любой, создающий несколько сообщений из одного.)

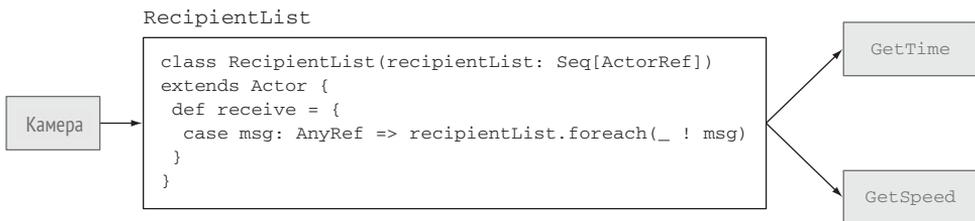
Список получателей – очень простой прием, потому что это единственный компонент; его задача – разослать полученное сообщение нескольким другим компонентам. На рис. 8.9 показано, как полученные сообщения рассылаются акторам `GetTime` и `GetSpeed`.



**Рис. 8.9.** Список получателей

В данном примере из каждого исходного сообщения требуется извлечь точно две единицы информации, поэтому `RecipientList`, по сути, имеет статическую природу и всегда пересылает исходное сообщение двум задачам: `GetTime` и `GetSpeed`. В других случаях список получателей может динамически изменяться, а сами получатели – определяться на основе содержимого сообщения или состояния списка.

На рис. 8.10 представлена простейшая реализация списка получателей; когда полученное сообщение просто пересылается всем его членам. Опробуем наш `RecipientList` в работе. Сначала создадим его с помощью класса `TestProbe` (вы уже встречались с ним в главе 3).



**Рис. 8.10.** `RecipientList`

#### Листинг 8.4. Тестирование списка получателей

```

val endProbe1 = TestProbe()
val endProbe2 = TestProbe()
val endProbe3 = TestProbe()
val list = Seq(endProbe1.ref, endProbe2.ref, endProbe3.ref)
  
```

← Создание списка получателей

```

val actorRef = system.actorOf(
  Props(new RecipientList(list)))
val msg = "message"
actorRef ! msg
endProbe1.expectMsg(msg)
endProbe2.expectMsg(msg)
endProbe3.expectMsg(msg)

```

← Отправка сообщения

← Все получатели должны получить сообщение

После отправки сообщения актору `RecipientList` все тестовые акторы должны получить его.

Этот прием не отличается особой сложностью и используется в шаблоне параллельной обработки дроблением.

## 8.2.4. Реализация компонента объединения с использованием агрегатора

Список получателей создает две копии исходного сообщения и посылает их акторам `GetSpeed` и `GetTime`, каждый из которых выполняет свою часть работы. Когда оба значения, время и скорость, будут извлечены, получившиеся результаты нужно объединить в один. Это делается компонентом объединения. На рис. 8.11 показано место компонента объединения в общем потоке, а его реализация – в листинге 8.5.

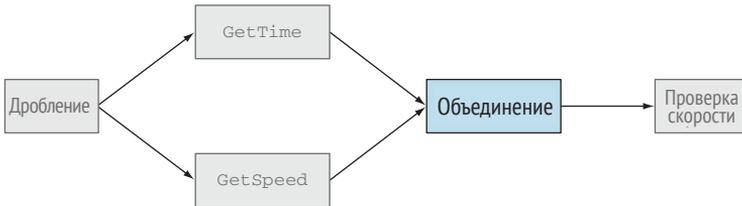


Рис. 8.11. Место компонента объединения в общем потоке

Прием «Агрегатор» используется для объединения нескольких сообщений. Это может быть процедура выбора, когда выбирается и возвращается сообщение, прибывшее первым, или простое объединение нескольких сообщений в одно, как в данном примере. Одна из особенностей агрегатора – необходимость хранения сообщений, чтобы по прибытии их всех агрегатор смог заняться объединением. Для простоты примера мы реализуем класс `Aggregator`, объединяющий два сообщения `PhotoMessage` в одно.

### Листинг 8.5. Aggregator

```

class Aggregator(timeout:Duration, pipe:ActorRef) extends Actor {
  val messages = new ListBuffer[PhotoMessage]
  def receive = {
    case rcvMsg: PhotoMessage => {

```

← Буфер для хранения необработанных сообщений

```

messages.find(_.id == rcvMsg.id) match {
  case Some(alreadyRcvMsg) => {
    val newCombinedMsg = new PhotoMessage(
      rcvMsg.id,
      rcvMsg.photo,
      rcvMsg.creationTime.orElse(alreadyRcvMsg.creationTime),
      rcvMsg.speed.orElse(alreadyRcvMsg.speed) )
    pipe ! newCombinedMsg
    // удалить сообщение
    messages -= alreadyRcvMsg
  }
  case None => messages += rcvMsg
}
}
}
}
}

```

Второе сообщение (из двух),  
 значит, можно объединить их

Удалить обработанное  
 сообщение из списка

Первое сообщение, его  
 нужно сохранить для  
 обработки в будущем

Получив сообщение, мы проверяем – первое оно или второе. Если сообщение первое, оно сохраняется в буфере `messages`. Если сообщение второе, можно приступать к обработке. В данном случае агрегатор объединяет сообщения в одно и посылает результат следующему актору в цепочке.

#### Листинг 8.6. Тестирование агрегатора

```

val endProbe = TestProbe()
val actorRef = system.actorOf(
  Props(new Aggregator(1 second, endProbe.ref)))
val photoStr = ImageProcessing.createPhotoString(new Date(), 60)
val msg1 = PhotoMessage("id1",
  photoStr,
  Some(new Date()),
  None)
actorRef ! msg1

```

← Послать первое сообщение

```

val msg2 = PhotoMessage("id1",
  photoStr,
  None,
  Some(60))
actorRef ! msg2

```

← Послать второе сообщение

```

val combinedMsg = PhotoMessage("id1",
  photoStr,
  msg1.creationTime,
  msg2.speed)

```

← Ждать результата

Класс `Aggregator` действует в точности, как ожидалось. Ему посылаются два сообщения, которые он благополучно объединяет и посылает результат. Но, так как этот актер имеет состояние, мы должны обеспечить его сохранность. Представьте, что получится, если одна из задач благополучно пошлет свое сообщение агрегатору, а вторая потерпит аварию? В этом случае первое сообщение навсегда «застрянет» в буфере, и никто не узнает, что случилось с этим сообщением. С течением времени буфер будет накапливать все больше и больше таких застрявших сообщений и наконец может разрастись до таких размеров, что не уместится в памяти и вызовет аварийное завершение всего приложения. Есть много способов решить эту проблему; в данном примере мы используем тайм-аут. Предполагается, что обеим задачам требуется примерно одинаковое время на обработку; следовательно, оба сообщения должны быть приняты почти одновременно. Время может отличаться из-за особенностей используемых ресурсов или планирования потоков выполнения, в которых действуют задачи. Если второе сообщение не поступает в течение установленного тайм-аута, его следует считать потерянным. Определившись с выбором приема тайм-аута, мы должны решить, как будет реагировать агрегатор на потерю сообщения. В нашем случае потеря сообщения не является чем-то катастрофическим и неполное сообщение можно продолжить обрабатывать. То есть наш агрегатор всегда будет посылать сообщение, даже если оно не было заполнено до конца.

Для реализации тайм-аута используем планировщик. После получения первого сообщения запланируем отправку сообщения `TimeoutMessage` (указав себя – `self` – как получателя). Получив сообщение `TimeoutMessage`, агрегатор проверит наличие в буфере `messages` искомого сообщения, что возможно, только если второе сообщение не было принято вовремя. В этом случае сообщение просто отправляется дальше, минуя объединение. Если сообщения нет в буфере, значит, оно уже было успешно обработано и отправлено.

#### Листинг 8.7. Реализация тайм-аута

```
case class TimeoutMessage(msg:PhotoMessage)

def receive = {
  case rcvMsg: PhotoMessage => {
    messages.find(_.id == rcvMsg.id) match {
      case Some(alreadyRcvMsg) => {
        val newCombinedMsg = new PhotoMessage(
          rcvMsg.id,
          rcvMsg.photo,
          rcvMsg.creationTime.getOrElse(alreadyRcvMsg.creationTime),
          rcvMsg.speed.getOrElse(alreadyRcvMsg.speed) )
```

```

    pipe ! newCombinedMsg
    // удалить сообщение
    messages -= alreadyRcvMsg
  }
  case None => {
    messages += rcvMsg
    context.system.scheduler.scheduleOnce( ← Запланировать тайм-аут
      timeout,
      self,
      new TimeoutMessage(rcvMsg))
  }
}
}
}
case TimeoutMessage(rcvMsg) => { ← Тайм-аут истек
  messages.find(_.id == rcvMsg.id) match {
    case Some(alreadyRcvMsg) => { ← Послать первое сообщение,
      pipe ! alreadyRcvMsg ← если второе не поступило
      messages -= alreadyRcvMsg
    }

    case None => // сообщение уже обработано ← Оба сообщения уже обработаны,
  } ← поэтому ничего делать не нужно
}
}
}
}

```

Итак, мы реализовали тайм-аут; теперь посмотрим, как поведет себя агрегатор, если одно из сообщений не поступит в установленный интервал времени:

```

val endProbe = TestProbe()
val actorRef = system.actorOf(
  Props(new Aggregator(1 second, endProbe.ref)))
val photoStr = ImageProcessing.createPhotoString( ← Создать
  new Date(), 60) ← сообщение
val msg1 = PhotoMessage("id1",
  photoStr,
  Some(new Date()),
  None)
actorRef ! msg1 ← Послать только
← одно сообщение
endProbe.expectMsg(msg1) ← Ждать истечения тайм-аута
← и прибытия сообщения

```

Как показали испытания, когда посылается только одно сообщение, истекает тайм-аут; агрегатор обнаруживает отсутствие второго сообщения и посылает первое как результат объединения.

Но это не единственная проблема, с которой мы можем столкнуться. В разделе 4.2, где описывался жизненный цикл акторов, мы видели, что

необходимо также обеспечить сохранность состояния актора на случай его перезапуска. Если актор `Aggregator` потерпит аварию по какой-либо причине, он потеряет уже принятые сообщения в процессе перезапуска. Как решить эту проблему? Как вы уже знаете, перед перезапуском актора вызывается его метод `preRestart`. Его можно использовать для сохранения состояния актора. В данном случае в классе `Aggregator` можно использовать простое решение: повторно послать сообщения самому себе перед перезапуском. Поскольку порядок сообщений для нас не имеет значения, этот прием хорошо подойдет для наших нужд. Мы отправим сообщения из буфера и сохраним их вновь после перезапуска экземпляра. Полная реализация класса `Aggregator` показана в листинге 8.8.

### Листинг 8.8. Полная реализация класса `Aggregator`

```
class Aggregator(timeout: FiniteDuration, pipe: ActorRef)
  extends Actor {

  val messages = new ListBuffer[PhotoMessage]
  implicit val ec = context.system.dispatcher
  override def preRestart(reason: Throwable, message: Option[Any]) {
    super.preRestart(reason, message)
    messages.foreach(self ! _)
    messages.clear()
  }

  def receive = {
    case rcvMsg: PhotoMessage => {
      messages.find(_.id == rcvMsg.id) match {
        case Some(alreadyRcvMsg) => {
          val newCombinedMsg = new PhotoMessage(
            rcvMsg.id,
            rcvMsg.photo,
            rcvMsg.creationTime.orElse(alreadyRcvMsg.creationTime),
            rcvMsg.speed.orElse(alreadyRcvMsg.speed))
          pipe ! newCombinedMsg
          // удалить сообщение
          messages -= alreadyRcvMsg
        }
        case None => {
          messages += rcvMsg
          context.system.scheduler.scheduleOnce(
            timeout,
            self,
            new TimeoutMessage(rcvMsg))
        }
      }
    }
  }
}
```

← Послать все сообщения из почтового ящика

```

}
case TimeoutMessage(rcvMsg) => {
  messages.find(_.id == rcvMsg.id) match {
    case Some(alreadyRcvMsg) => {
      pipe ! alreadyRcvMsg
      messages -= alreadyRcvMsg
    }
    case None => // сообщение уже обработано
  }
}
case ex: Exception => throw ex
}
}

```

Добавлено для нужд  
тестирования

Мы добавили возбуждение исключения, чтобы обеспечить перезапуск при тестировании. Но что получится, если сообщение об истечении тайм-аута будет получено дважды для одного и того же сообщения? Поскольку по тайм-ауту мы ничего не делаем, если сообщение уже было обработано, повторное срабатывание тайм-аута не является для нас проблемой. В этом примере только первый тайм-аут вынудит нас предпринять какие-то действия, если потребуется, поэтому представленное простое решение вполне работоспособно.

Способны ли наши изменения решить проблему? Давайте посмотрим. Пошлем агрегатору первое сообщение и вызовем его перезапуск перед отправкой второго сообщения. Для перезапуска пошлем исключение `IllegalStateException`, которое возбудит агрегатор. Сможет ли агрегатор объединить два сообщения после перезапуска?

#### Листинг 8.9. Проверка работы агрегатора после перезапуска

```

val endProbe = TestProbe()
val actorRef = system.actorOf(
  Props(new Aggregator(1 second, endProbe.ref)))
val photoStr = ImageProcessing.createPhotoString(new Date(), 60)

val msg1 = PhotoMessage("id1",
  photoStr,
  Some(new Date()),
  None)
actorRef ! msg1

```

← Послать первое сообщение

```

actorRef ! new IllegalStateException("restart")

```

← Перезапустить агрегатор

```

val msg2 = PhotoMessage("id1",
  photoStr,
  None,

```

```
Some(60))
actorRef ! msg2
```

← Послать второе сообщение

```
val combinedMsg = PhotoMessage("id1",
  photoStr,
  msg1.creationTime,
  msg2.speed)
```

```
endProbe.expectMsg(combinedMsg)
```

Тест оказался благополучно пройденным; агрегатор смог объединить сообщения после перезапуска. В системах, основанных на обмене сообщениями, под сохранностью понимается возможность сохранения сообщений во время сбоев. Мы реализовали сохранность в акторе `Aggregator`, просто заставив его повторно посылать самому себе сообщения, хранящиеся в нем, и с помощью модульного теста убедились, что этот прием работает (соответственно, если в механизме сохранения что-то изменится, мы узнаем об этом еще до того, как соберем действующий экземпляр приложения). В модуле `akka-contrib` имеется свой класс `Aggregator` (<http://doc.akka.io/docs/akka/2.4.2/contrib/aggregator.html>), но мы не будем обсуждать его здесь.

### 8.2.5. Объединение компонентов в реализацию шаблона параллельной обработки дроблением

Подготовив и протестировав компоненты, мы можем составить из них законченную реализацию шаблона. Обратите внимание, что после разработки каждой части в отдельности благодаря модульным тестам мы переходим к этому заключительному этапу, пребывая в полной уверенности.

**Листинг 8.10.** Реализация шаблона параллельной обработки дроблением

```
val endProbe = TestProbe()
val aggregateRef = system.actorOf(
  Props(new Aggregator(1 second, endProbe.ref))) ← Создать агрегатор
val speedRef = system.actorOf(
  Props(new GetSpeed(aggregateRef))) ← Создать актер GetSpeed
val timeRef = system.actorOf(
  Props(new GetTime(aggregateRef))) ← и связать его с агрегатором
val actorRef = system.actorOf(
  Props(new RecipientList(Seq(speedRef, timeRef))) ← Создать список получателей
  и добавить в него акторы
  GetTime и GetSpeed

val photoDate = new Date()
val photoSpeed = 60
val msg = PhotoMessage("id1",
```

```
ImageProcessing.createPhotoString(photoDate, photoSpeed))
```

```
actorRef ! msg
```

← Послать сообщение  
в список получателей

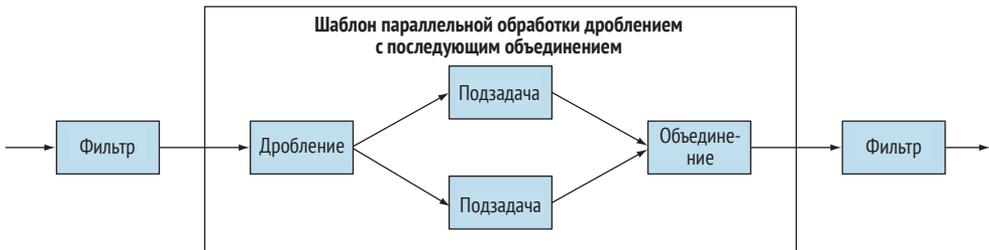
```
val combinedMsg = PhotoMessage(msg.id,  
    msg.photo,  
    Some(photoDate),  
    Some(photoSpeed))
```

```
endProbe.expectMsg(combinedMsg) ← Получить объединенное сообщение
```

В этом примере мы посылаем одно сообщение первому актору; `RecipientList`. Он создает два сообщения, которые обрабатываются параллельно. Оба результата посылаются актору `Aggregator`, и когда он их получит, то обработает и объединит в одно сообщение и передаст на следующий этап: наш тестовый объект `endProbe`. Так работает шаблон параллельной обработки дроблением. В нашем примере мы реализовали две задачи, но вообще количество задач может быть сколь угодно большим.

Шаблон параллельной обработки дроблением можно также объединить с шаблоном конвейеров и фильтров. Сделать это можно двумя способами. Первый – использовать его как часть конвейера. В этом случае он будет являться одним из фильтров: компонент дробления будет принимать, а компонент объединения – отправлять те же сообщения, что и другие фильтры в конвейере.

На рис. 8.12 показан конвейер из фильтров, один из которых реализован с применением шаблона параллельной обработки дроблением. Это позволяет нам гибко менять фильтры местами, добавлять и удалять фильтры из конвейера, не нарушая логику работы остальной части конвейера.



**Рис. 8.12.** Использование шаблона параллельной обработки дроблением для реализации фильтра

Второй способ: использовать конвейер из фильтров как подзадачу в шаблоне параллельной обработки дроблением. В этом случае сообщения будут обрабатываться конвейером перед объединением.

На рис. 8.13 изображена схема простой реализации шаблона параллельной обработки дроблением, в которой исходное сообщение передается в

два потока параллельной обработки. Один из них – конвейер, а второй – обычная задача (как в предыдущем примере). Такое сочетание шаблонов может пригодиться в больших системах; оно обеспечивает высокую гибкость и возможность повторного использования кода.

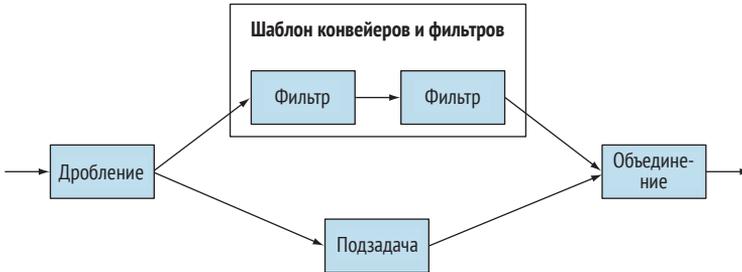


Рис. 8.13. Использование шаблона конвейеров и фильтров в параллельной обработке дроблением

## 8.3. Маршрутизация

Еще одним шаблоном интеграции акторов является шаблон маршрутизации, который можно рассматривать как динамическую разновидность шаблона конвейеров и фильтров. Для объяснения преимуществ этого шаблона используем немного более сложный пример. Представьте, что вы владеете фабрикой, выпускающей автомобили, и по умолчанию они окрашиваются в черный цвет. Заказывая новый автомобиль, клиент может потребовать установить в него дополнительное оборудование, например систему навигации, датчики для парковки, или потребовать покрасить автомобиль в серый цвет – каждый автомобиль можно подготовить под требования клиента. Когда заказывается автомобиль в самой простой комплектации, он будет окрашиваться в черный цвет, а все остальные этапы дооборудования должны пропускаться. Но, когда клиент желает установить все дополнительное оборудование, этап окрашивания в черный цвет должен пропускаться, а все остальные этапы – выполняться. Для решения этой проблемы можно использовать шаблон маршрутизации. Маршрут – это цепочка задач, которые должны участвовать в обработке данного сообщения. Внутри каждого сообщения включается экземпляр маршрута `RouteSlip`. С его помощью каждая задача сможет определить, куда дальше передать сообщение после обработки. Обычно для объяснения идеи маршрутизации используется простая метафора конверта с записанным маршрутом: на нем может быть список людей, которые должны подписать документ. Когда конверт попадает к очередному человеку, он сначала осматривает его и затем ставит подпись и отметку времени, когда передал конверт следующему.

На рис. 8.14 показаны два возможных запроса клиента. В одном случае клиент заказывает автомобиль в простейшей комплектации, а во втором клиент требует установить все дополнительное оборудование. Получив заказ, маршрутизатор SlipRouter должен определить, какие шаги следует выполнить, и передать сообщение на первый этап.

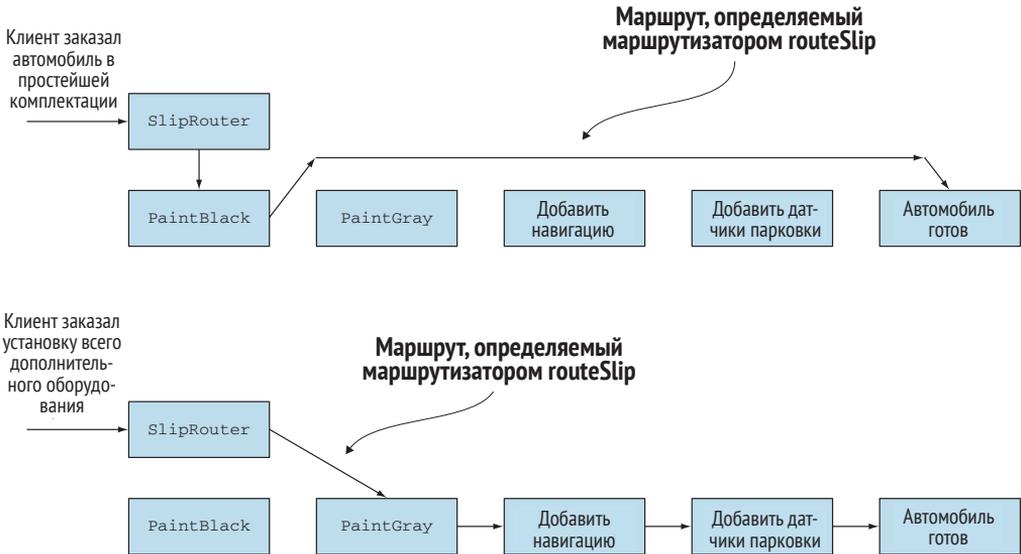


Рис. 8.14. Шаблон динамической маршрутизации

В первом случае на рис. 8.14 маршрутизатор определяет, что следует выполнить только этап PaintBlack, и создает маршрут, содержащий только его и конечный этап. Когда задача PaintBlack завершится, она пошлет сообщение следующему шагу, в данном случае – конечному, пропустив все другие. Во втором случае клиентом заказана установка всего дополнительного оборудования и окраска в серый цвет, поэтому в маршрут были добавлены все этапы, кроме PaintBlack. Каждый раз, когда очередная задача завершается, она посылает сообщение с маршрутом следующему этапу в списке. Чтобы такая схема работала, каждая задача должна реализовать один и тот же интерфейс, потому что маршрут определяется динамически. Задачи можно пропускать или выполнять их в другом порядке, и если использовать разные типы сообщений, может получиться так, что какая-то другая задача не сможет обработать полученное сообщение. Мы уже видели это требование, когда обсуждали шаблон конвейеров и фильтров. Единственное отличие, что в шаблоне конвейеров и фильтров маршрут определяется статически: он один и тот же для всех сообщений. В шаблоне динамической маршрутизации маршрут определяется динамически и каждое сообщение может идти по своему маршруту; этот шаблон можно рассматривать как динамическую версию шаблона конвейеров и фильт-

ров. Для каждого сообщения маршрутизатор `SlipRouter` создает свой конвейер (маршрут).

При использовании этого шаблона важно, чтобы все этапы имели одинаковый интерфейс и не зависели друг от друга, в точности как в шаблоне конвейеров и фильтров. Начнем реализацию этого примера с определения интерфейса сообщений для каждой задачи:

```
object CarOptions extends Enumeration {
  val CAR_COLOR_GRAY, NAVIGATION, PARKING_SENSORS = Value
}
```

```
case class Order(options: Seq[CarOptions.Value])
case class Car(color: String = "",
  hasNavigation: Boolean = false,
  hasParkingSensors: Boolean = false)
```

Нам нужен заказ и все возможные дополнительные варианты, по которым маршрутизатор сможет создать маршрут `RouteSlip`, и нам нужно сообщение `Car`, для которого маршрутизатор `SlipRouter` будет составлять маршрут. Кроме того, для каждой задачи нам понадобится функция, которая направит сообщение на следующий этап в маршруте. Как обычно, нам понадобится класс сообщения, а также трейт, с помощью которого будем добавлять возможность отправки сообщения следующему получателю (согласно маршруту).

#### Листинг 8.11. Маршрутизация сообщений

```
case class RouteSlipMessage(routeSlip: Seq[ActorRef],
  message: AnyRef)
// ← Фактическое сообщение,
//    пересылаемое между задачами

trait RouteSlip {

  def sendMessageToNextTask(routeSlip: Seq[ActorRef],
    message: AnyRef): Unit = {
    val nextTask = routeSlip.head
    val newSlip = routeSlip.tail
    if (newSlip.isEmpty) {
      nextTask ! message
    } else {
      nextTask ! RouteSlipMessage(
        routeSlip = newSlip,
        message = message)
    }
  }
}
```

← Получить следующий шаг

← Если следующий шаг последний,  
послать сообщение без маршрута

← Послать сообщение на следующий шаг  
и обновить маршрут в сообщении

Это будет использоваться в каждой задаче. Когда задача завершится и создаст новое сообщение `Car`, метод `sendMessageToNextTask` должен будет найти следующую задачу. Теперь мы можем реализовать наши задачи.

### Листинг 8.12. Примеры задач

```
class PaintCar(color: String) extends Actor with RouteSlip {
  def receive = {
    case RouteSlipMessage(routeSlip, car: Car) => {
      sendMessageToNextTask(routeSlip,
        car.copy(color = color))
    }
  }
}

```

← Задача окрашивания

```
class AddNavigation() extends Actor with RouteSlip {
  def receive = {
    case RouteSlipMessage(routeSlip, car: Car) => {
      sendMessageToNextTask(routeSlip,
        car.copy(hasNavigation = true))
    }
  }
}

```

← Дооборудование навигатором

```
class AddParkingSensors() extends Actor with RouteSlip {
  def receive = {
    case RouteSlipMessage(routeSlip, car: Car) => {
      sendMessageToNextTask(routeSlip,
        car.copy(hasParkingSensors = true))
    }
  }
}

```

← Дооборудование датчиками для парковки

Каждая задача изменяет одно поле в `Car` и затем вызывает `sendMessageToNextTask`, чтобы отправить сообщение `Car` следующей задаче. Теперь нам осталось только реализовать маршрутизатор `SlipRouter`, который тоже является обычным актором, принимающим заказ и создающим объект маршрута, исходя из параметров заказа.

### Листинг 8.13. SlipRouter

```
class SlipRouter(endStep: ActorRef) extends Actor with RouteSlip {
  val paintBlack = context.actorOf(
    Props(new PaintCar("black")), "paintBlack")
  val paintGray = context.actorOf(
    Props(new PaintCar("gray")), "paintGray")
}

```

Создание задач



```

color = "black",
hasNavigation = false,
hasParkingSensors = false)
probe.expectMsg(defaultCar)

```

Получить автомобиль  
в простой комплектации

Если послать заказ без дополнительных параметров, маршрутизатор создаст маршрут, включающий ссылки на актер `PaintCar` с аргументом `black` и на тестовый актер `probe` в конце. Сообщение `RouteSlipMessage`, содержащее автомобиль и маршрут, передается первой задаче, `PaintCar`. По ее завершении сообщение передается актору `probe`. При использовании всех дополнительных параметров сообщение последовательно пересылается всем задачам, и когда оно достигнет конца, в нем будут установлены все дополнительные параметры.

#### Листинг 8.15. Создание автомобиля с дополнительным оборудованием

```

val fullOrder = new Order(Seq(
  CarOptions.CAR_COLOR_GRAY,
  CarOptions.NAVIGATION,
  CarOptions.PARKING_SENSORS))
router ! fullOrder
val carWithAllOptions = new Car(
  color = "gray",
  hasNavigation = true,
  hasParkingSensors = true)
probe.expectMsg(carWithAllOptions)

```

Послать запрос на создание автомобиля  
в полной комплектации

Получить автомобиль  
в полной комплектации

Шаблон динамической маршрутизации позволяет оперативно создавать конвейеры, обладающие всеми преимуществами шаблона конвейеров и фильтров, и при этом сохранять гибкость обработки сообщений с разным содержанием по-разному.

## 8.4. В заключение

В этой главе мы исследовали примеры гибких приемов интеграции акторов Akka. Комбинируя их, вы сможете создавать сложные системы. Вот несколько уроков этой главы:

- масштабирование обработки требует распределения работы между параллельно выполняющимися задачами;
- рассмотренные шаблоны послужат вам отправной точкой для дальнейшего изучения стандартных способов масштабирования;
- модель программирования акторов позволяет сосредоточиться на организации кода, а не на тонкостях реализации передачи сообщений и их планировании;

- шаблоны – это строительные блоки, которые можно комбинировать разными способами при строительстве больших частей систем.

Применяя эти шаблоны в Акка, можно без особого труда адаптировать базовые подходы конструирования компонентов под более сложные требования. Сообщения являются частью последовательного процесса. Некоторые его фрагменты могут выполняться параллельно, но при этом сам поток обработки должен быть статическим и одинаковым для всех сообщений. В следующей главе мы рассмотрим приемы маршрутизации сообщений между разными акторами для создания динамической структуры задач.

# Глава 9

## Маршрутизация сообщений

В этой главе:

- использование шаблона «маршрутизатор»;
- масштабирование с применением маршрутизаторов;
- создание маршрутизатора с состоянием с применением методов `become/unbecome`.

В предыдущей главе мы рассмотрели несколько шаблонов интеграции акторов для решения широкого круга задач. Однако все описанные подходы были основаны на использовании сообщений одного типа. Но на практике часто бывает нужно обрабатывать разные сообщения.

Маршрутизаторы сообщений необходимы, когда требуется организовать управление масштабированием. Например, для обработки увеличенной нагрузки нужно создать несколько экземпляров одной и той же задачи и маршрутизатор, который будет решать, какому экземпляру послать очередное сообщение. Мы начнем эту главу с описания шаблона «Маршрутизатор» и познакомимся с тремя причинами использования маршрутизации для управления потоком сообщений:

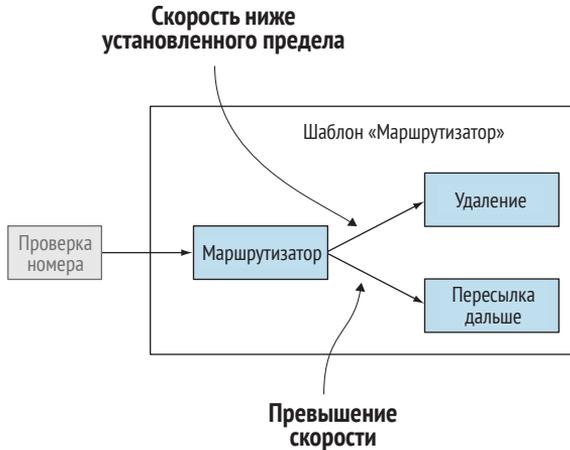
- производительность;
- содержимое сообщения;
- состояние.

Затем мы покажем, как организовать маршрутизацию в каждом из трех случаев.

Если производительность или масштабирование – главная причина внедрения маршрутизации, тогда лучше использовать маршрутизаторы, встроенные в Akka, потому что они высокооптимизированы. Но если маршрутизацию предполагается использовать для разделения сообщений по их содержимому или по состоянию, это верный признак, что лучше использовать обычные акторы.

## 9.1. Шаблон «Маршрутизатор»

Сначала познакомимся с шаблоном в целом – когда и как он применяется, – а потом углубимся в детали реализации каждой конкретной разновидности маршрутизатора. Обсуждение реализаций мы начнем с наиболее известного шаблона распределения сообщений посредством набора шагов. Мы рассмотрим пример ускорения приложения обслуживания дорожной камеры. На этот раз мы будем посылать сообщения задаче удаления или на следующий шаг обработки, в зависимости от скорости автомобиля. Если скорость автомобиля меньше установленного предела, сообщение будет посылаться этапу удаления (вместо простого отбрасывания). Но если скорость окажется выше предела, это уже будет считаться нарушением, и сообщение подвергнется обычной обработке. Для решения этой задачи мы используем шаблон «Маршрутизатор». Как показано на рис. 9.1, маршрутизатор способен посылать сообщения в разные потоки.



**Рис. 9.1.** Логика маршрутизации посылает разные сообщения в разные потоки обработки

Есть много причин использовать логику для выбора маршрута, по которому следует отправить сообщение. Как отмечалось выше, есть три причины для организации управления потоком сообщений в приложениях.

- *Производительность* – на обработку сообщения уходит много времени, но есть возможность обрабатывать сообщения параллельно. То есть сообщения можно обрабатывать несколькими потоками. В примере с дорожной камерой автомобиля, попавшие на камеру, можно обрабатывать несколькими параллельными потоками, потому что вся логика обработки сосредоточена в пределах потока.
- *Получаемые сообщения имеют разное содержимое* – сообщение имеет атрибут (как номерной знак в нашем примере), в зависимости от

значения которого сообщение должно обрабатываться разными задачами.

- *Состояние маршрутизатора* – например, когда камера работает в режиме резервирования, все сообщения должны передаваться в задачу удаления; иначе они должны обрабатываться как обычно.

Во всех случаях (независимо от причины или конкретной используемой логики) маршрутизатор должен решить, какой следующей задаче передать сообщение. Задачи, которым маршрутизатор может послать сообщение, в Akka называются *маршрутами*.

В этой главе мы рассмотрим разные подходы к маршрутизации сообщений. Попутно познакомимся еще с несколькими механизмами Akka, которые могут пригодиться не только для реализации маршрутизаторов, но и для других процессов, например когда необходимо обрабатывать сообщения по-разному, в зависимости от состояния актора. Раздел 9.2 мы начнем с обзора использования маршрутизаторов на примере приложения, где решение должно приниматься на основе производительности. Масштабирование – одна из главных причин использования поддержки маршрутизаторов в Akka, которая занимает центральное место во всей стратегии масштабирования. В разделе 9.3 мы исследуем маршрутизацию с применением обычных акторов, когда содержимое сообщения и состояние являются решающими аспектами, и рассмотрим другие подходы, основанные на применении обычных акторов.

## 9.2. Балансировка нагрузки с помощью маршрутизаторов Akka

Одна из причин использовать маршрутизатор – обеспечить равномерное распределение нагрузки между несколькими акторами, чтобы улучшить производительность системы, обрабатывающей большое количество сообщений. Это могут быть локальные акторы (вертикальное масштабирование) или акторы на удаленных серверах (горизонтальное масштабирование). Главный аргумент в пользу использования приема масштабирования – простота реализации маршрутизации в Akka.

В нашем примере с дорожной камерой относительно много времени занимает шаг распознавания номера. Чтобы организовать параллельное распознавание нескольких номеров, мы используем маршрутизатор.

На рис. 9.2 можно видеть, что маршрутизатор способен посылать сообщения одному из экземпляров GetLicense на выбор. Когда сообщение попадает в маршрутизатор, он выбирает один из доступных процессов и посылает сообщение этому процессу. Получив следующее сообщение, маршрутизатор выбирает другой процесс и посылает сообщение ему.

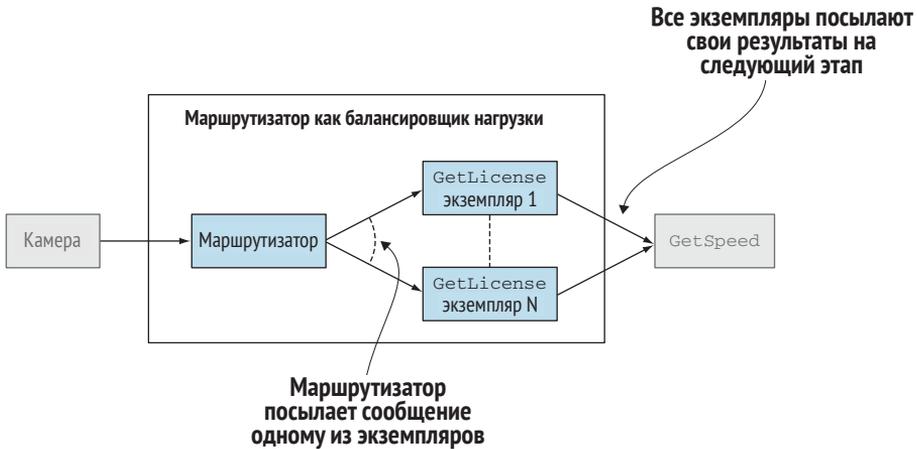


Рис. 9.2. Маршрутизатор как балансировщик нагрузки

Для реализации такого маршрутизатора используем встроенную поддержку маршрутизации в Akka. В Akka есть разделение между маршрутизатором, содержащим логику маршрутизации, и актором, представляющим маршрутизатор. Логика маршрутизации решает, какой маршрут выбрать, и может использоваться внутри актора. Актор-маршрутизатор – это самостоятельный актор, загружающий логику маршрутизации и другие настройки из конфигурации, и может управлять самими маршрутами. В Akka имеется две разновидности встроенных маршрутизаторов:

- *пул* – эти маршрутизаторы управляют маршрутами. Они отвечают за создание маршрутов и удаление их из списка, когда те завершают работу. Пул можно использовать, когда все маршруты создаются и распределяются одинаково и нет необходимости предусматривать особые процедуры для восстановления маршрутов;
- *группа* – маршрутизаторы, которые не управляют маршрутами. Маршруты создаются системой, а маршрутизатор использует операцию выбора акторов для поиска маршрутов. Маршрутизатор с группой не управляет маршрутами. Управление маршрутами должно осуществляться где-то еще в системе. Группу можно использовать, когда требуется по-особенному управлять жизненным циклом маршрутов или требуется больше контроля над местом и временем создания маршрутов.

Пул проще, потому что поддерживает автоматическое управление маршрутами (посредством событий жизненного цикла маршрутов), но за эту простоту приходится платить отсутствием возможности настраивать логику отдельных маршрутов.

На рис. 9.3 изображена иерархия акторов-маршрутов, где можно увидеть разницу между пулом и группой. При использовании пула маршруты

являются потомками маршрутизатора, а когда используется маршрутизатор с группой, маршруты могут быть потомками любого другого актора (в данном примере – актора `RouteeCreator`). Маршруты необязательно должны быть потомками одного и того же родителя. Они просто должны создаваться и запускаться.

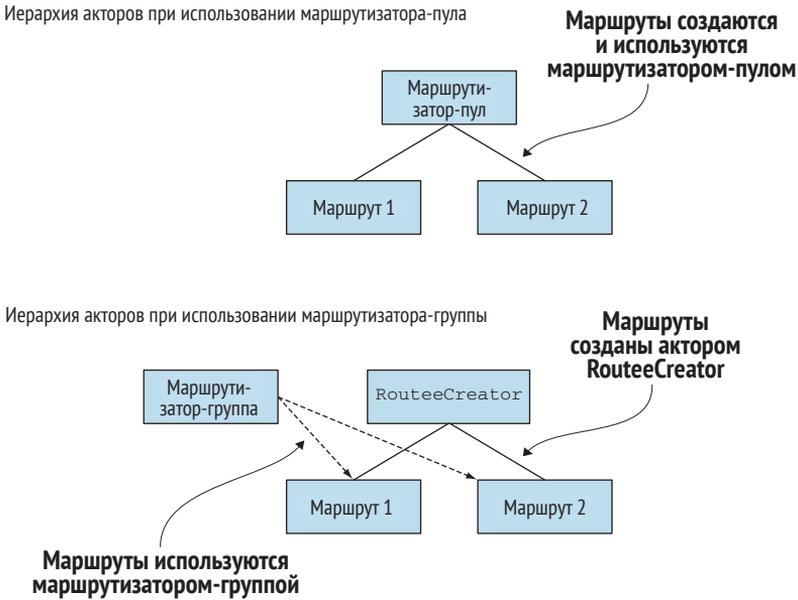


Рис. 9.3. Иерархии акторов при использовании пула и группы

В Акка имеется несколько встроенных маршрутизаторов. Они перечислены в табл. 9.1. В таблице указывается логика маршрутизатора и связанный пул или группа, использующие эту логику.

Таблица 9.1. Список маршрутизаторов в Акка

Логика	Пул	Группа	Описание
RoundRobinRoutingLogic	RoundRobinPool	RoundRobinGroup	Логика посылает первое полученное сообщение первому маршруту, следующее сообщение – второму маршруту и т. д. Когда каждому будет передано по сообщению, маршрутизатор возвращается к первому маршруту, и цикл повторяется

Логика	Пул	Группа	Описание
RandomRoutingLogic	RandomPool	RandomGroup	Эта логика посылает каждое полученное сообщение случайно выбранному маршруту
SmallestMailboxRoutingLogic	SmallestMailbox-Pool	Нет	Этот маршрутизатор проверяет почтовые ящики маршрутов и выбирает тот, у которого в почтовом ящике меньше всего сообщений. Групповая версия отсутствует, потому что логика выбирает актор, используя внутренние особенности реализации, а количество сообщений в почтовом ящике нельзя определить по ссылке на актор
Нет	BalancingPool	Нет	Этот маршрутизатор распределяет сообщения между простаивающими маршрутами. Он делает это, используя внутренние особенности реализации. Использует один почтовый ящик для всех маршрутов. Внутренне маршрутизатор использует собственный диспетчер маршрутов. Это также объясняет, почему доступна версия только для пула
BroadcastRoutingLogic	BroadcastPool	BroadcastGroup	Посылает полученное сообщение всем маршрутам. Этот маршрутизатор не совсем соответствует определению шаблона, но он реализует список получателей

Логика	Пул	Группа	Описание
ScatterGatherFirst-CompletedRoutingLogic	ScatterGather-FirstCompleted-Pool	ScatterGather-FirstCompleted-Group	Посылает полученное сообщение всем маршрутам и возвращает оригинальному отправителю первый полученный ответ. Технически это реализация шаблона параллельной обработки дроблением для случая состояющихся задач
ConsistentHashingRouting-Logic	Consistent-HashingPool	Consistent-HashingGroup	Для выбора маршрута этот маршрутизатор использует хеширование. Его можно применять, когда требуется посылать разные сообщения по одному и тому же маршруту, но при этом не важно, по какому именно

В разделе 9.2.1 мы рассмотрим несколько примеров использования этих маршрутизаторов. Одни и те же логические требования можно удовлетворить с использованием любого из представленных типов (различия, как отмечалось выше, касаются в основном внутренней реализации). В следующих разделах мы используем наиболее часто общие логики маршрутизации – циклическую (round-robin), балансирующую и основанную на хешировании. В первую очередь в разделе 9.2.1 мы рассмотрим маршрутизатор для пула `BalancingPool`, потому что он обладает рядом примечательных особенностей. В разделе 9.2.2 мы познакомимся с маршрутизаторами групп на примере `RoundRobinGroup`. И наконец, в разделе 9.2.3 мы покажем пример использования `ConsistentHashingPool` и объясним, когда и как использовать этот маршрутизатор.

### 9.2.1. Маршрутизатор с пулом

Выше вы увидели, какие маршрутизаторы доступны. Они имеют три разновидности: логика для использования в вашем собственном акторе, группа акторов и пул акторов. Сначала посмотрим, как использовать маршрутизатор для пула. При использовании пула от вас не требуется создавать маршруты или управлять ими; это будет делать маршрутизатор. Пул можно использовать, когда все маршруты создаются и используются одинаково и нет необходимости специально восстанавливать маршруты. То есть для «простых» маршрутов пул – хороший выбор.

## Создание маршрутизатора с пулом

Все маршрутизаторы с пулами просты в использовании. Есть два разных способа настройки пула: в файле конфигурации или в программном коде. Сначала познакомимся с настройкой в файле конфигурации, потому что этот подход позволяет изменять логику маршрутизатора, что невозможно при настройке из программного кода. Используем `BalancingPool` для распределения заданий между нашими акторами `GetLicense`.

Мы должны создать маршрутизатор в коде и получить ссылку `ActorRef` на него, чтобы посылать ему сообщения.

**Листинг 9.1.** Создание маршрутизатора с использованием конфигурационного файла

```
val router = system.actorOf(
  FromConfig.props(Props(new GetLicense(endProbe.ref))),
  "poolRouter"
)
```

← Определение маршрутизатора с помощью конфигурации  
 ← Как маршрутизатор должен создавать маршруты  
 ← Имя маршрутизатора

Это весь код, необходимый для создания маршрутизатора с настройками из конфигурационного файла. Но это еще не все. В листинге 9.2 приводится конфигурация маршрутизатора.

**Листинг 9.2.** Конфигурация маршрутизатора

```
akka.actor.deployment {
  /poolRouter {
    router = balancing-pool
    nr-of-instances = 5
  }
}
```

← Полное имя маршрутизатора  
 ← Число маршрутов в пуле  
 ← Логика, используемая маршрутизатором

Этих трех строк достаточно для настройки маршрутизатора. Первая строка – это имя маршрутизатора, которое должно совпадать с именем в коде. В нашем примере мы создали маршрутизатор вызовом `system.actorOf` на верхнем уровне системы акторов; то есть он получит имя `/poolRouter`. Если бы мы создавали маршрутизатор внутри другого актора, например внутри актора с именем `getLicenseBalancer`, тогда в конфигурации следовало бы указать имя маршрутизатора `/getLicenseBalancer/poolRouter`. Это важно; иначе Akka не найдет настройки для маршрутизатора.

Следующая строка в конфигурации определяет логику маршрутизации, в данном случае пулу назначается логика выбора с балансировкой нагрузки. Последняя строка определяет количество маршрутов (5) в пуле.

Это все, что мы должны сделать, чтобы вместо одного актора `GetLicense` использовать пул акторов `GetLicense`. Единственное отличие в нашем коде заключается в добавлении вызова `FromConfig.props()`. Все остальное оста-

лось прежним. Чтобы сообщение попало в один из маршрутов `GetLicense`, достаточно послать его маршрутизатору по полученной ссылке `ActorRef`:

```
router ! Photo("123xyz", 60)
```

Маршрутизатор сам решит, какой маршрут выбрать для обработки сообщения. Мы начали этот раздел с упоминания, что есть два разных способа определения маршрутизаторов. Второй способ менее гибкий, но мы все равно рассмотрим его для полноты картины. Тот же самый маршрутизатор с пулом можно полностью определить в коде, как показано в листинге 9.3.

### Листинг 9.3. Создание `BalancingPool` в коде

```
val router = system.actorOf(
  BalancingPool(5).props(Props(new GetLicense(endProbe.ref))),
  "poolRouter"
)
```

← Создает `BalancingPool` с 5 маршрутами

Единственное отличие – в том, что мы заменили `FromConfig` вызовом `BalancingPool(5)`, напрямую определив тип пула и количество маршрутов. Код в листинге 9.3 создаст точно такой же пул, как и код в листинге 9.1 с конфигурацией в листинге 9.2.

Обычно сообщения, посылаемые маршрутизатору, передаются маршрутам. Но некоторые сообщения маршрутизатор обрабатывает сам. В этом разделе мы охватим большую часть этих сообщений и начнем с сообщений `Kill` и `PoisonPill`. Эти сообщения не пересылаются маршрутам и обрабатываются самим маршрутизатором. В данном случае маршрутизатор завершится, а поскольку это пул, вместе с ним завершатся все маршруты, фактически являющиеся дочерними акторами.

Как мы уже знаем, когда мы посылаем сообщение маршрутизатору, оно будет передано только одному маршруту, по крайней мере так действует большинство маршрутизаторов. Но есть возможность заставить маршрутизатор разослать сообщение всем маршрутам. Для этого можно использовать еще одно специальное ширококвещательное сообщение: `Broadcast`. Получив такое сообщение, маршрутизатор перешлет его содержимое всем маршрутам. Ширококвещательные сообщения `Broadcast` можно посылать маршрутизаторам обоих видов – пулам и группам.

**ПРИМЕЧАНИЕ.** Единственный маршрутизатор, где сообщение `Broadcast` не работает, – `BalancingPool`. Проблема в том, что в этом маршрутизаторе все маршруты имеют общий почтовый ящик. Рассмотрим, например, `BalancingPool` с пятью экземплярами. Чтобы разослать ширококвещательное сообщение, маршрутизатор должен послать сообщение каждому из пяти маршрутов. Но из-за того, что в данном случае имеется только один общий

почтовый ящик, все пять сообщений окажутся в нем. Сообщения распределяются между маршрутами в зависимости от нагрузки, то есть первые пять запросов извлекут широковещательные сообщения. Все бы ничего, если бы все маршруты были нагружены в равной степени. Но если один маршрут выполняет обработку сообщения, требующую больше времени, чем обработка широковещательного сообщения, другой маршрут успеет обработать несколько широковещательных сообщений до того, как нагруженный маршрут освободится. Возможна даже ситуация, когда один маршрут обработает все широковещательные сообщения, а другие четыре – ни одного. Поэтому не используйте сообщения Broadcast в комбинации с BalancingPool.

### Удаленные маршруты

Роль маршрутов в предыдущем разделе играли локальные акторы, но мы говорили, что маршрутизаторы способны рассылать сообщения акторам, находящимся на разных серверах. Создать маршрут, действующий на удаленном сервере, совсем не сложно. Для этого настройку маршрутизатора нужно завернуть в вызов `RemoteRouterConfig` и определить удаленные адреса.

#### Листинг 9.4. Заворачивание настройки маршрутизатора в вызов `RemoteRouterConfig`

```
val addresses = Seq(
  Address("akka.tcp", "GetLicenseSystem", "192.1.1.20", 1234),
  AddressFromURIString("akka.tcp://GetLicenseSystem@192.1.1.21:1234"))

val routerRemote1 = system.actorOf(
  RemoteRouterConfig(FromConfig(), addresses).props(
    Props(new GetLicense(endProbe.ref))), "poolRouter-config")

val routerRemote2 = system.actorOf(
  RemoteRouterConfig(RoundRobinPool(5), addresses).props(
    Props(new GetLicense(endProbe.ref))), "poolRouter-code")
```

Здесь показаны два примера конструирования адресов: непосредственное использование класса `Address` и создание адреса из URI. Здесь также показаны две версии создания `RouterConfig`. Такой маршрутизатор с пулом будет создавать свои маршруты на разных серверах, циклически перебирая заданные удаленные адреса, пока не создаст заданное количество маршрутов. Благодаря этому маршруты будут равномерно распределены по удаленным серверам.

Как видите, маршрутизаторы обеспечивают простоту горизонтального масштабирования. От вас требуется только использовать `RemoteRouterConfig`. Существует еще одна обертка, упрощающая создание маршрутов на

удаленных серверах: `ClusterRouterPool`. Эту обертку можно использовать, когда имеется кластер (и описывается в главе 14, полностью посвященной кластерам).

До сих пор мы использовали маршрутизаторы с фиксированным числом маршрутов, но когда количество сообщений, поступающих в единицу времени, может меняться в широких пределах, желательно изменять количество маршрутов, чтобы сбалансировать систему. Для этого можно использовать поддержку изменения размера пула.

### Пул с поддержкой динамического изменения размера

Когда нагрузка изменяется в широких пределах, часто бывает желательно синхронно изменять количество маршрутов; когда количества маршрутов недостаточно, будут возникать задержки в обработке сообщений, потому что вновь прибывшим сообщениям придется ждать, пока закончится обработка предыдущих. Но когда маршрутов слишком много, они впустую будут тратить ресурсы системы. В этих случаях хорошо иметь возможность динамически изменять размер пула (в зависимости от нагрузки). Это можно сделать с помощью поддержки динамического изменения размера пула.

Эта поддержка настраивается в широких пределах. Можно установить верхнюю и нижнюю границы количества маршрутов. Когда понадобится уменьшить или увеличить пул, Akka сделает это. Все эти параметры можно определить в конфигурации пула, как показано в листинге 9.5.

**Листинг 9.5.** Настройка динамического изменения размера пула

```
akka.actor.deployment {
  /poolRouter {
    router = round-robin-pool
    resizer {
      enabled = on
      lower-bound = 1
      upper-bound = 10
      pressure-threshold = 1
      rampup-rate = 0.25
      backoff-threshold = 0.3
      backoff-rate = 0.1
      messages-per-resize = 10
    }
  }
}
```

- Включить поддержку динамического изменения размера пула (points to `resizer {`)
- Минимальное число маршрутов в пуле (points to `lower-bound = 1`)
- Максимальное число маршрутов в пуле (points to `upper-bound = 10`)
- Когда считать, что маршруты работают под высокой нагрузкой (points to `pressure-threshold = 1`)
- Скорость добавления новых маршрутов (points to `rampup-rate = 0.25`)
- Когда число маршрутов следует уменьшить (points to `backoff-threshold = 0.3`)
- Скорость удаления маршрутов (points to `backoff-rate = 0.1`)
- Как быстро появится повторная возможность изменить размер (points to `messages-per-resize = 10`)

}

Первый шаг – включить поддержку динамического изменения пула. Далее с помощью атрибутов `lower-bound` и `upper-bound` можно определить нижнюю и верхнюю границы (количества маршрутов).

Далее следуют атрибуты, описывающие, когда пул должен изменяться в размерах и с какой скоростью.

Начнем с описания увеличения. Когда пул оказывается под высокой нагрузкой, нужно увеличить число маршрутов. Но как определить, что пул находится под высокой нагрузкой? Ответ на этот вопрос дает атрибут `pressure-threshold`. Значение этого атрибута определяет, сколько сообщений должно находиться в почтовом ящике маршрута, чтобы считать, что он работает под увеличенной нагрузкой. Например, когда это значение равно 1, маршрут будет считаться действующим под высокой нагрузкой, когда в его почтовом ящике имеется хотя бы одно сообщение, а когда это значение равно 3, в почтовом ящике маршрута должно иметься, по меньшей мере, три сообщения. Значение 0 представляет особый случай. Оно означает, что когда маршрут обрабатывает сообщение, он находится под нагрузкой. Теперь, зная, как определить состояние высокой нагрузки, посмотрим, как действует механизм добавления новых маршрутов.

Рассмотрим пул с пятью экземплярами, которому присвоен атрибут `pressure-threshold` со значением 0. Когда такой маршрутизатор получит четыре сообщения, он передаст их первым четырем маршрутам. В этот момент четыре маршрута заняты, а один простаивает. Первая ситуация, изображенная на рис. 9.4, соответствует моменту, когда поступает пятое сообщение. В этом случае ничего не происходит, потому что проверка выполняется до передачи сообщения маршруту. А в этот момент один из маршрутов все еще простаивает, то есть пул не находится под нагрузкой.

Но когда маршрутизатор получит еще одно сообщение, все маршруты окажутся занятыми обработкой предыдущих сообщений (вторая ситуация на рис. 9.4). Это означает, что пул оказался под высокой нагрузкой, и он должен добавить новые маршруты. Изменение размера выполняется не синхронно, потому что создание нового маршрута не всегда выполняется быстрее, чем простое ожидание, пока какой-то маршрут закончит обработку своего сообщения. В сбалансированной системе высока вероятность, что один из маршрутов вот-вот закончит обработку своего сообщения. А это значит, что шестое сообщение будет передано не вновь созданному маршруту, а одному из уже имеющихся. Но вполне вероятно, что следующее сообщение попадет в новый маршрут.

Когда пул определяет, что находится под нагрузкой, он добавляет новые маршруты со скоростью, как определяет атрибут `rampup-rate`. Этот атрибут определяет долю от общего количества. Например, когда имеется

пять маршрутов и атрибут `rampup-rate` со значением `0.25`, размер пула будет увеличен на 25% (с округлением вверх до целого числа), то есть на два маршрута ( $5 \times 0.25 = 1.25$ , после округления вверх получается 2), и в результате пул будет содержать семь маршрутов.

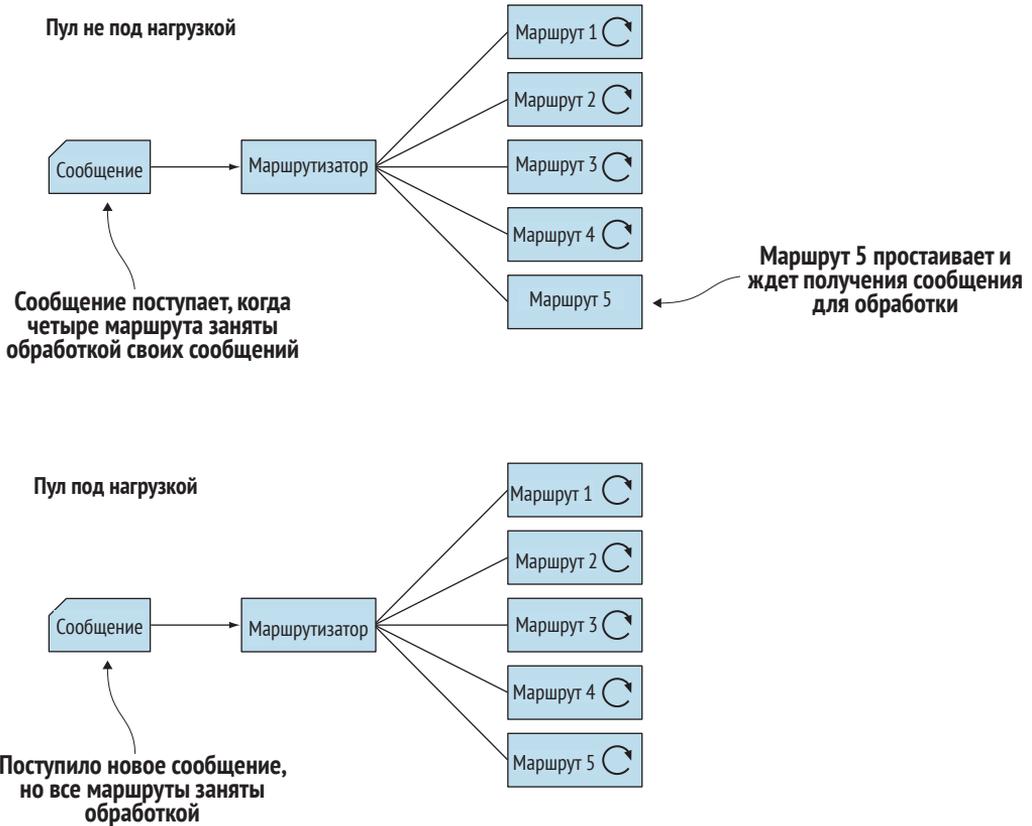


Рис. 9.4. Пул переходит в состояние работы под нагрузкой

Теперь, поняв, как работают настройки, управляющие увеличением размера пула, перейдем к настройкам уменьшения размера. Атрибут `backoff-threshold` определяет момент, когда пул можно уменьшить. Атрибут `back-off` определяет долю маршрутов, которую нужно удалить, если доля занятых маршрутов окажется ниже порога `backoff-threshold`. Например, когда в пуле имеется 10 маршрутов, уменьшение произойдет, когда доля занятых маршрутов окажется ниже 30%. То есть, когда только два (или меньше) маршрута будут заняты, общее количество маршрутов уменьшится.

Количество удаляемых маршрутов определяет атрибут `backoff-rate`. Он действует так же, как `rampup-rate`. В данном примере с 10 маршрутами и `backoff-rate`, равным `0.1`, удален будет один маршрут ( $10 \times 0.1 = 1$ ).

Последний атрибут, `messages-per-resize`, определяет, какое количество сообщений должен получить маршрутизатор до того, как сможет повторно изменить размер пула. Это предотвращает ситуации, когда маршрутизатор постоянно увеличивает или уменьшает пул с каждым сообщением. Это может происходить, когда пул может иметь только два размера: для одного из них нагрузка оказывается слишком велика, а для другого слишком мала, из-за чего маршрутизатор вынужден постоянно переключаться между этими размерами. Или, когда сообщения поступают пакетами, этот атрибут можно использовать для задержки изменения размера до появления следующего пакета.

### Наблюдение

Еще одна функция маршрутизаторов, о которой следует рассказать, – наблюдение. Поскольку маршрутизатор создает маршруты, он также является супервизором для этих акторов. Маршрутизатор по умолчанию всегда передает служебные сообщения своему супервизору. Это может приводить к неожиданным результатам. Когда один из маршрутов потерпит аварию, маршрутизатор сообщит об этом своему супервизору. Этот супервизор почти наверняка захочет перезапустить актор, но вместо маршрута перезапустит маршрутизатор. А перезапуск маршрутизатора вызовет перезапуск всех маршрутов, не только аварийного. Со стороны это выглядит так, будто маршрутизатор использует стратегию `AllForOneStrategy`. Чтобы решить эту проблему, в момент создания маршрутизатора можно настроить свою стратегию.

Для этого нужно создать стратегию и присвоить ее маршрутизатору:

```
val myStrategy = SupervisorStrategy.defaultStrategy    ← Создать стратегию наблюдения
val router = system.actorOf(
  RoundRobinPool(5, supervisorStrategy = myStrategy).props(Props[TestSuper]), ←
  "roundrobinRouter"
)                                                    | Использовать стратегию
                                                    | наблюдения
```

Когда один из маршрутов потерпит неудачу, только он и будет перезапущен, а все остальные продолжают работать, как ни в чем не бывало. Можно использовать супервизор по умолчанию, как в этом примере, а можно создать новую стратегию для маршрутизатора или использовать стратегию актора, родительского для маршрутизатора. В этом случае все маршруты будут действовать так, как если бы они были потомками родителя маршрутизатора.

Потомка можно остановить, если он потерпел неудачу, и пул не будет порождать новый маршрут после завершения прежнего. Когда все маршруты будут остановлены, маршрутизатор тоже остановится. Только когда используется механизм динамического изменения размера, маршрутиза-

тор не завершится, а будет поддерживать заданное минимальное число маршрутов.

В этом разделе вы узнали, насколько гибкими могут быть пулы, особенно когда настройки выполняются с помощью файла конфигурации. Вы можете изменять количество маршрутов и даже логику маршрутизации. А когда имеется несколько серверов, можно без ненужных сложностей создавать маршруты на разных серверах.

Но иногда ограничения, которые накладывают пулы, оказываются слишком строгими, и появляется желание иметь больше свободы в создании и управлении маршрутами. В такой ситуации вам могут пригодиться группы.

## 9.2.2. Маршрутизатор с группой

Пулы, представленные в предыдущем разделе, сами осуществляли управление маршрутами. При использовании маршрутизаторов с группами вы должны будете сами создавать маршруты. Это может пригодиться, когда требуется иметь более полный контроль над тем, когда и как создаются маршруты. Сначала создадим простой маршрутизатор для группы. Затем покажем, как можно динамически изменять маршруты, используя еще один набор сообщений маршрутизатора.

### Создание групп

Группы создаются почти так же, как пулы. Единственное отличие – при создании пула нужно указать количество экземпляров маршрутов, а при создании группы – список путей к маршрутам. Начнем с создания маршрутов. Для этого не требуется ничего особенного, но в нашем примере мы хотели, чтобы у всех акторов `GetLicense` был один родитель. Определим для этого класс `GetLicenseCreator`, который будет отвечать за создание акторов `GetLicense`. Позднее мы используем его для создания новых маршрутов взамен завершившихся.

**Листинг 9.6.** `GetLicenseCreator` создает маршруты

```
class GetLicenseCreator(nrActors: Int) extends Actor {

  override def preStart() {
    super.preStart()
    (0 until nrActors).map { nr =>
      context.actorOf(Props[GetLicense], "GetLicense"+nr)
      system.actorOf(Props( new GetLicenseCreator(2)), "Creator")
    }
  }
  ...
}
system.actorOf(Props( new GetLicenseCreator(2)), "Creator")
```

Создание маршрутов

Создание родительского актора маршрутов

Так же как в случае с пулом, есть два способа создания маршрутизатора для группы: с применением конфигурационного файла и полностью внутри кода. Первым рассмотрим пример с использованием конфигурационного файла.

#### Листинг 9.7. Конфигурация маршрутизатора для группы

```
akka.actor.deployment {
  /groupRouter {
    router = round-robin-group
    routees.paths = [
      "/user/Creator/GetLicense0",
      "/user/Creator/GetLicense1"
    ]
  }
}
val router = system.actorOf(FromConfig.props(), "groupRouter")
```

← Полное имя маршрутизатора

← Пути к актерам маршрутов

← Логика, используемая маршрутизатором

← Создание маршрутизатора для группы

Как видите, настройка группы почти не отличается от настройки пула; параметр `nr-of-instances` заменил параметр `routees.paths`. Создание группы выглядит еще проще, чем пула, потому что нет нужды определять, как должны создаваться маршруты. А так как в определении группы используются пути к актерам, для добавления удаленных акторов не требуется ничего менять, достаточно добавить полный путь к удаленному актору:

```
akka.actor.deployment {
  /groupRouter {
    router = round-robin-group

    routees.paths = [
      "akka.tcp://AkkaSystemName@10.0.0.1:2552/user/Creator/GetLicense0",
      "akka.tcp://AkkaSystemName@10.0.0.2:2552/user/Creator/GetLicense0"
    ]
  }
}
```

Настройка группы в коде тоже выполняется очень просто; нужно лишь подготовить список путей к маршрутам.

#### Листинг 9.8. Создание маршрутизатора для группы в коде

```
val paths = List("/user/Creator/GetLicense0",
  "/user/Creator/GetLicense1")

val router = system.actorOf(
  RoundRobinGroup(paths).props(), "groupRouter")
```

Наш новый маршрутизатор используется почти так же, как пул. Единственное отличие: завершение маршрута. Когда завершается маршрут,

входящий в пул, маршрутизатор обнаруживает это и удаляет его из пула. Маршрутизатор для группы не поддерживает такой возможности. После завершения маршрута маршрутизатор продолжит посылать ему сообщения, потому что маршрутизатор не управляет маршрутами, и, возможно, в какой-то момент в будущем актор станет доступен.

Давайте добавим в класс `GetLicenseCreator` создание нового актора после завершения одного из потомков. Используем для этого прием, описанный в главе 4.

**Листинг 9.9.** Создание новых маршрутов после завершения имеющихся

```
class GetLicenseCreator(nrActors: Int) extends Actor {

  override def preStart() {
    super.preStart()
    (0 until nrActors).map(nr => {
      val child = context.actorOf(
        Props(new GetLicense(nextStep)), "GetLicense"+nr)
      context.watch(child)
    })
  }

  def receive = {
    case Terminated(child) => {
      val newChild = context.actorOf(
        Props(new GetLicense(nextStep)), child.path.name)
      context.watch(newChild)
    }
  }
}
```

← Включить наблюдение за создаваемыми маршрутами

← Повторно создать маршрут после завершения существующего

С этим обновленным классом `GetLicenseCreator` маршрутизатор всегда сможет использовать ссылки на акторы без выполнения любых дополнительных действий. Давайте опробуем этот класс. Сначала создадим маршруты, а затем маршрутизатор, но прежде чем сделать что-либо, отправим всем маршрутам сообщение `PoisonPill`.

**Листинг 9.10.** Проверка управления маршрутами классом `GetLicenseCreator`

```
val endProbe = TestProbe()

val creator = system.actorOf(
  Props(new GetLicenseCreator2(2, endProbe.ref)), "Creator")
val paths = List(
  "/user/Creator/GetLicense0",
  "/user/Creator/GetLicense1")
```

← Создать маршруты

```

val router = system.actorOf(
  RoundRobinGroup(paths).props(), "groupRouter")
router ! Broadcast(PoisonPill)
Thread.sleep(100)

val msg = PerformanceRoutingMessage(
  ImageProcessing.createPhotoString(new Date(), 60, "123xyz"),
  None,
  None)

// проверить отклик маршрутов
router ! msg
endProbe.expectMsgType[PerformanceRoutingMessage](1 second)

```

Создать маршрутизатор

Остановить все маршруты

Если маршрутизатор получит сообщение до того, как произойдет пересоздание маршрутов, оно будет потеряно

Проверить, обработали ли запросы новые маршруты

Как видите, после остановки маршрутов будут созданы новые, которые возьмут на себя обработку входящих сообщений. `Thread.sleep` – самый простой способ гарантировать, что `GetLicenseCreator` получит достаточно времени для воссоздания маршрутов. Также можно было бы сгенерировать событие после воссоздания всех маршрутов и подписаться на это событие в тесте; или добавить сообщения `GetLicenseCreator` для проверки количества воссозданных маршрутов; или использовать сообщение `GetRoutees`, как описывается в следующем разделе. Реализацию всех этих решений мы оставляем вам как самостоятельное упражнение.

В этом примере мы создали новые акторы с теми же путями, но также есть возможность удалять маршруты из группы и добавлять в группу с помощью сообщений маршрутизатора.

### Динамическое изменение размера группы

Мы уже говорили о сообщениях, обрабатываемых маршрутизатором. Теперь поговорим о трех дополнительных сообщениях для управления группой маршрутов, с помощью которых можно получать маршруты, которыми управляет данный маршрутизатор, добавлять их и удалять:

- `GetRoutees` – служит для получения списка текущих маршрутов. В ответ на это сообщение маршрутизатор ответит сообщением `Routees`, содержащим маршруты.
- `AddRoutee(routee: Routee)` – получив это сообщение, маршрутизатор добавит маршрут `routee` в группу. Это сообщение принимает экземпляр трейта `Routee`, содержащий новый маршрут.
- `RemoveRoutee(routee: Routee)` – послав это сообщение, можно удалить указанный маршрут `routee` из группы.

Но использование данных сообщений имеет свои подводные камни. Сами сообщения и ответы на них используют экземпляр трейта `Routee`, имеющего единственный метод `send`. Этот метод позволяет послать сообщение непосредственно маршруту. Другие возможности не поддерживаются без преобразования `Routee` в класс реализации.

В ответ на сообщение `GetRoutees` возвращается меньше информации, чем можно было бы ожидать, если не выполнить приведение `Routee` к фактической реализации. Единственное, для чего оно действительно может использоваться, – определение количества маршрутов в группе или взаимодействие с маршрутами в обход маршрутизатора. Это может пригодиться для отправки особых сообщений особым маршрутам. Наконец, с помощью этого сообщения можно проверить, было ли обработано сообщение, посланное маршрутизатору, для чего достаточно послать `GetRoutees` сразу после маршрутизируемого сообщения. Если в ответ маршрутизатор вернет сообщение `Routees`, значит, сообщение, посланное перед `GetRoutees`, было обработано.

Вместе с сообщениями добавления и удаления нужно посылать экземпляр `Routee`. Чтобы добавить актор в группу, вам придется преобразовать ссылку `ActorRef` или путь к актору в `Routee`.

В `Akka` есть три реализации трейта `Routee`:

- `ActorRefRoutee(ref: ActorRef);`
- `ActorSelectionRoutee(selection: ActorSelection);`
- `SeveralRoutees(routees: immutable.IndexedSeq[Routee]).`

Мы используем последнюю реализацию, `SeveralRoutees`, потому что она создает `Routee` из списка экземпляров `Routee`. Если использовать первый вариант, `ActorRefRoutee`, маршрутизатор возьмет новый маршрут под наблюдение. Казалось бы, это не должно породить никаких проблем, но когда маршрутизатор, не являющийся супервизором `Routee`, получит сообщение `Terminated`, он возбудит исключение `akka.actor.DeathPactException`, которое завершит сам маршрутизатор. Скорее всего, это не совсем то, чего вам хотелось бы; чтобы иметь возможность восстанавливать маршрут после его завершения, используйте второй вариант, реализацию `ActorSelectionRoutee`.

Удаляя маршрут, вы должны использовать тот же тип экземпляра `Routee`, который использовался для добавления. Иначе маршрут не будет удален. Это еще одна причина, почему лучше использовать `ActorSelectionRoutee` при удалении маршрута.

Предположим, что нам нужно иметь возможность изменять размер группы. В листинге 9.11 представлена одна из возможных реализаций этого механизма. В нем определяется класс `DynamicRouteeSizer`, который будет управлять маршрутами в группе маршрутизатора. Изменить размер группы можно отправкой сообщения `PreferredSize`.

**Листинг 9.11.** Пример реализации управления размером группы

```

class DynamicRouteeSizer(nrActors: Int,
                        props: Props,
                        router: ActorRef) extends Actor {
  var nrChildren = nrActors
  var childInstanceNr = 0

  // перезапускает потомка
  override def preStart() {
    super.preStart()
    (0 until nrChildren).map(nr => createRoutee())
  }

  def createRoutee() {
    childInstanceNr += 1
    val child = context.actorOf(props, "routee" + childInstanceNr)
    val selection = context.actorSelection(child.path)
    router ! AddRoutee(ActorSelectionRoutee(selection))
    context.watch(child)
  }

  def receive = {
    case PreferredSize(size) => {
      if (size < nrChildren) {
        //удаление
        context.children.take(nrChildren - size).foreach(ref => {
          val selection = context.actorSelection(ref.path)
          router ! RemoveRoutee(ActorSelectionRoutee(selection))
        })
        router ! GetRoutees
      } else {
        (nrChildren until size).map(nr => createRoutee())
      }
      nrChildren = size
    }

    case routees: Routees => {
      //преобразовать Routees в actorPath
      import collection.JavaConversions._
      var active = routees.getRoutees.map{
        case x: ActorRefRoutee => x.ref.path.toString
        case x: ActorSelectionRoutee => x.selection.pathString
      }
      //обработать список маршрутов
      for(routee <- context.children) {
        val index = active.indexOf(routee.path.toStringWithoutAddress)
        if (index >= 0) {

```

При запуске создает первоначально запрошенное количество маршрутов

После создания нового маршрута добавить его в группу маршрутизатора, преобразовав в экземпляр ActorSelectionRoutee

Число маршрутов изменилось

Удалить лишние маршруты

Создать новые маршруты

Проверить, можно ли завершить маршруты или нужно воссоздать их после завершения

Преобразовать список маршрутов в список путей к акторам

```

    active.remove(index)
  } else {
    //Потомок больше не используется маршрутизатором
    routee ! PoisonPill
  }
}
//active содержит завершившиеся маршруты
for (terminated <- active) {
  val name = terminated.substring(terminated.lastIndexOf("/")+1)
  val child = context.actorOf(props, name)
  context.watch(child)
}
}
case Terminated(child) => router ! GetRoutees
}
}

```

Потомок исключен из группы маршрутизатора; теперь его можно завершить

Перезапустить случайно завершившихся потомков

Потомок завершился; проверить, было ли это запланировано, запросив маршруты у маршрутизатора

Тут происходит много интересного. Начнем с получения сообщения `PreferredSize`. Когда поступает это сообщение, значит, у нас или слишком мало маршрутов, или слишком много. Если маршрутов слишком мало, нужно создать дополнительные дочерние акторы и добавить их в группу маршрутизатора. Если маршрутов слишком много, нужно исключить лишние маршруты из группы и затем завершить их. Исключение из группы необходимо, чтобы маршрутизатор не смог послать сообщение завершившемуся дочернему актору, иначе оно будет потеряно. Поэтому мы посылаем друг за другом сообщения `RemoveRoutee` и `GetRoutees`. Получив ответ, мы будем точно знать, что маршрутизатор не пошлет сообщений удаляемым маршрутам, и их можно завершить. Для этого используется сообщение `PoisonPill`, чтобы дать возможность останавливаемому актору завершить обработку предшествующих сообщений.

Далее следует описание действий в ответ на завершение потомка. И снова возможны два случая, когда приходит сообщение `Terminated`. В первом случае, когда размер группы уменьшается намеренно, мы не должны ничего делать. Второй случай – когда активный маршрут завершается из-за ошибки. В этом втором случае нужно повторно создать маршрут. Мы должны воссоздать потомка, используя то же имя, а не просто удалить потомка из группы и создать нового, потому что удаление завершившегося потомка может вызвать завершение маршрутизатора, если потомок является последним активным маршрутом. Чтобы решить, что делать, мы посылаем сообщение `GetRoutees` и, получив ответ, выбираем, какое действие предпринять.

Последнее, что осталось обсудить, – действия в ответ на сообщение `Routees`. Мы используем это сообщение, чтобы определить, можно ли безопасно завершить потомка или, может быть, его нужно перезапустить.

Для этого нам нужны пути к акторам маршрутов, недоступные в интерфейсе `Routee`. Для решения этой проблемы мы используем классы реализаций `ActorSelectionRoutee` и `ActorRefRoutee`. Последний класс не используется в маршрутизаторе, но мы добавили его для пущей уверенности. Теперь, имея список путей к акторам, можно определить, что делать: остановить поток или перезапустить его.

Чтобы задействовать механизм изменения размера группы, достаточно создать маршрутизатор и актор `DynamicRouteeSizer`:

```
val router = system.actorOf(RoundRobinGroup(List()).props(), "router")
val props = Props(new GetLicense(endProbe.ref))
val creator = system.actorOf(
  Props( new DynamicRouteeSizer(2, props, router)),
  "DynamicRouteeSizer"
)
```

С помощью приемов, описанных в этом разделе, вы сможете динамически изменять маршруты в группе, но поступать так не рекомендуется из-за большого количества ловушек, поджидающих вас на этом пути. Вы узнали, как использовать пулы и группы маршрутов, но остался еще один вид логики маршрутизации, который работает чуточку иначе, чем другие. Это маршрутизатор `ConsistentHashing`.

### 9.2.3. Маршрутизатор `ConsistentHashing`

Как показал предыдущий раздел, маршрутизаторы дают отличный и простой способ вертикального и даже горизонтального масштабирования. Но с отправкой сообщений разным маршрутам может возникнуть проблема. Что получится, если реализовать актор, состояние которого определяется полученным сообщением? Возьмем, например, агрегатор для шаблона параллельной обработки дроблением из раздела 8.2.4. Представьте также, что имеется маршрутизатор с 10 маршрутами типа `Aggregator`, каждый из которых объединяет два связанных сообщения в одно. В этом случае велика вероятность, что первое сообщение будет передано маршруту № 1, а второе – маршруту № 2. В результате оба агрегатора решат, что второе сообщение потерялось, и вернут два неполных сообщения. Для решения подобных проблем был создан маршрутизатор `ConsistentHashing`.

Этот маршрутизатор посылает похожие сообщения одному и тому же маршруту. Когда будет получено второе сообщение, маршрутизатор отправит его тому же маршруту, что и первое. Это позволит агрегатору объединить два сообщения. Для этого маршрутизатор должен идентифицировать сообщения как похожие. С этой целью `ConsistentHashing` вычисляет хеш-код сообщения и отображает его в один из маршрутов. Процесс отображения состоит из нескольких шагов, как показано на рис. 9.5.

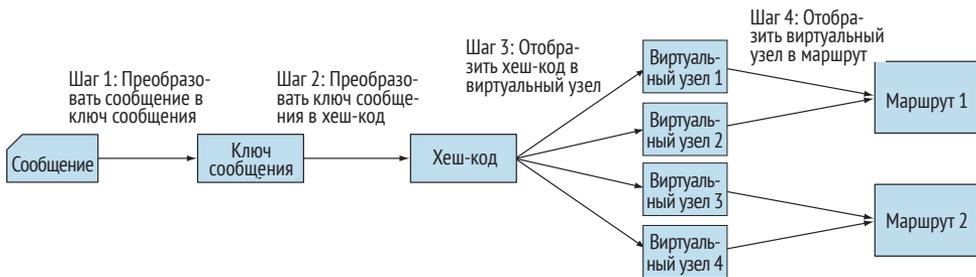


Рис. 9.5. Шаги, выполняемые маршрутизатором ConsistentHashing для выбора маршрута

*Шаг 1* преобразует сообщение в объект ключа сообщения. Похожие сообщения, например с одинаковым идентификационным номером, получают одинаковые ключи. Тип ключа совершенно не важен; единственное ограничение – для похожих сообщений всегда должны получаться одинаковые объекты ключей. Ключи для сообщений разных типов должны отличаться и иметь реализацию. Маршрутизатор поддерживает три способа преобразования сообщения в ключ.

- С помощью частично определенной функции в маршрутизаторе. В этом случае решения принимаются на уровне конкретного маршрутизатора.
- Сообщение реализует интерфейс `akka.routing.ConsistentHashingRouter.ConsistentHashable`. В этом случае решения принимаются на уровне конкретных сообщений.
- Сообщение можно вернуть в `map` `akka.routing.ConsistentHashingRouter.ConsistentHashableEnvelope`. В этом случае решения принимаются на уровне отправителя. Отправитель должен знать, какой ключ использовать.

Последний вариант наименее предпочтительный, потому что образует тесную связь между отправителем и маршрутами. Отправитель должен знать, что маршрутизатор реализует интерфейс `ConsistentHashingRouter` и как он распределяет сообщения. Два других решения образуют более слабую связь. Далее мы покажем, как использовать все три способа.

*Шаг 2* создает хеш-код на основе ключа сообщения. Этот хеш-код используется для выбора виртуального узла (*шаг 3*), и на последнем *шаге* (*4*) выбирается конкретный маршрут для обработки всех сообщений, соответствующих этому виртуальному узлу. Первое, что бросается в глаза, – использование виртуального узла. Нельзя ли сразу по хеш-коду определить маршрут? Виртуальные узлы применяются, чтобы получить более равно-

мерное распределение сообщений между маршрутами. Количество виртуальных узлов, обслуживаемых маршрутом, настраивается при помощи `ConsistentHashingRouter`. В нашем примере имеется по два виртуальных узла для каждого маршрута.

Рассмотрим пример использования маршрута, который объединяет два сообщения на основе идентификационного номера. Для простоты мы убрали из примера весь код, реализующий обработку ошибок.

#### Листинг 9.12. Объединение двух сообщений в одно

```
trait GatherMessage {
  val id:String
  val values:Seq[String]
}

case class GatherMessageNormalImpl(id:String, values:Seq[String])
  extends GatherMessage

class SimpleGather(nextStep: ActorRef) extends Actor {
  var messages = Map[String, GatherMessage]()
  def receive = {
    case msg: GatherMessage => {
      messages.get(msg.id) match {
        case Some(previous) => {
          //объединение
          nextStep ! GatherMessageNormalImpl(
            msg.id,
            previous.values ++ msg.values)
          messages -= msg.id
        }
        case None => messages += msg.id -> msg
      }
    }
  }
}
```

Актор `SimpleGather` объединяет два сообщения с одинаковыми идентификационными номерами в одно. Мы использовали трейт как тип сообщения, чтобы получить возможность использовать разные реализации сообщений, что потребуется в одном из примеров хеширования. Рассмотрим далее три способа вычисления ключей сообщений.

#### Передача в маршрутизатор частично определенной функции

Сначала рассмотрим способ вычисления ключей сообщений, основанный на частично определенной функции. При создании маршрутизатора

вы должны передать ему частично определенную функцию, выбирающую ключ сообщения:

```
def hashMapping: ConsistentHashMap = {
  case msg: GatherMessage => msg.id
}

val router = system.actorOf(
  ConsistentHashingPool(10,
    virtualNodesFactor = 10,
    hashMapping = hashMapping
  ).props(Props(new SimpleGather(endProbe.ref))),
  name = "routerMapping"
)
```

Частично определенная функция отображения

Количество виртуальных узлов на маршрут

Установка функции отображения

Это все, что нужно, чтобы использовать `ConsistentHashingRouter`. От вас требуется только создать частично определенную функцию для выбора ключа сообщения. Если теперь послать два сообщения с одинаковым идентификационным номером, маршрутизатор должен отправить их одному и тому же маршруту. Давайте проверим:

```
router ! GatherMessageNormalImpl("1", Seq("msg1"))
router ! GatherMessageNormalImpl("1", Seq("msg2"))
endProbe.expectMsg(GatherMessageNormalImpl("1", Seq("msg1", "msg2")))
```

Этот метод можно использовать, когда к маршрутизатору предъявляются определенные требования по распределению сообщений между маршрутами. Например, когда в системе имеется несколько маршрутизаторов, принимающих сообщения одного и того же типа, но каждый должен распределять сообщения по-разному, используя разные способы вычисления ключей сообщений. Представьте, что один маршрутизатор объединяет сообщения с одинаковыми идентификационными номерами, а другой подсчитывает сообщения с одинаковыми значениями некоторого поля и роль ключа в этом случае должно выполнять значение. Когда для данного сообщения ключ никогда не изменяется, предпочтительнее реализовать вычисление ключа в самом сообщении.

### Вычисление ключа внутри сообщения

Реализовать преобразование сообщения в его ключ можно также внутри самого сообщения, унаследовав трейт `ConsistentHashable`:

```
case class GatherMessageWithHash(id:String, values:Seq[String])
  extends GatherMessage with ConsistentHashable {

  override def consistentHashKey: Any = id
}
```

При использовании такого сообщения нет нужды определять функцию отображения, потому что будет использоваться функция из сообщения:

```
val router = system.actorOf(
  ConsistentHashingPool(10, virtualNodesFactor = 10)
    .props(Props(new SimpleGather(endProbe.ref))),
  name = "routerMessage"
)

router ! GatherMessageWithHash("1", Seq("msg1"))
router ! GatherMessageWithHash("1", Seq("msg2"))
endProbe.expectMsg(GatherMessageNormalImpl("1", Seq("msg1", "msg2")))
```

Это решение предпочтительно, когда для данного сообщения ключ никогда не изменяется. Но мы говорили, что есть три способа вычисления ключей сообщений. Поэтому рассмотрим третий и последний способ: с использованием `ConsistentHashableEnvelope`.

### Вычисление ключа отправителем

Последний способ – передача ключа сообщения через объект `ConsistentHashableEnvelope`.

```
val router = system.actorOf(
  ConsistentHashingPool(10, virtualNodesFactor = 10)
    .props(Props(new SimpleGather(endProbe.ref))),
  name = "routerMessage"
)

router ! ConsistentHashableEnvelope(
  message = GatherMessageNormalImpl("1", Seq("msg1")), hashKey = "1")
router ! ConsistentHashableEnvelope(
  message = GatherMessageNormalImpl("1", Seq("msg2")), hashKey = "1")
endProbe.expectMsg(GatherMessageNormalImpl("1", Seq("msg1", "msg2")))
```

В этом случае перед отправкой маршрутизатору сообщение заворачивается в объект `ConsistentHashableEnvelope`, который имеет поле `hashKey` для хранения ключа сообщения. Но, как уже отмечалось прежде, это решение требует, чтобы все отправители знали, что используется маршрутизатор `ConsistentHashingRouter` и вместе с сообщением требуется поставлять ключ. Этот способ может применяться, например, когда все сообщения от одного отправителя передавались одному и тому же маршруту: в этом случае в поле `hashKey` можно передавать идентификатор отправителя. Но это не значит, что каждый маршрут будет обрабатывать сообщения только от одного отправителя. Вполне может получиться так, что один маршрут будет обрабатывать сообщения от разных отправителей.

Мы рассмотрели три способа преобразования сообщений в ключи сообщений, но самое замечательное, что в одном маршрутизаторе можно использовать все три решения.

В этом разделе мы узнали, как использовать маршрутизаторы Akka, которые обладают высочайшей производительностью. Но, как уже говорилось, иногда маршрутизация должна выполняться на основе содержимого сообщения или некоторого состояния. Методы такой маршрутизации мы рассмотрим в следующем разделе.

## 9.3. Реализация шаблона маршрутизатора с применением акторов

Реализация шаблона маршрутизатора не всегда требует применения встроенных маршрутизаторов Akka. Когда выбор маршрута зависит от содержимого сообщения или некоторого состояния, проще реализовать обычный актор, потому что такой прием позволяет использовать все преимущества акторов. Кроме того, если вы решите создать свой маршрутизатор по аналогии с маршрутизаторами Akka, вам придется решать проблемы, связанные с конкурентным выполнением.

В этом разделе мы рассмотрим несколько реализаций шаблона маршрутизатора с использованием обычных акторов. Начнем с версии, выполняющей маршрутизацию по содержимому сообщений. В следующем разделе мы используем методы `become/unbecome` для реализации маршрутизации в зависимости от состояния маршрутизатора. После этого мы обсудим, почему для реализации шаблона маршрутизатора необязательно использовать отдельный актор, а вместо этого маршрутизацию можно встроить в актор, обрабатывающий сообщения.

### 9.3.1. Маршрутизация по содержимому

Чаще всего маршрутизация сообщений производится по их содержимому. В начале раздела 9.1 мы привели пример такого маршрутизатора. Когда скорость автомобиля ниже предельно допустимой, значит, правила дорожного движения не были нарушены, и такое сообщение требуется передать на этап удаления. Когда скорость выше предельно допустимой, значит, нарушение имеет место, и обработка должна быть продолжена.

На рис. 9.6 показано, как по содержимому сообщения выбирается тот или иной поток обработки. В данном случае маршрутизация осуществляется на основе значения скорости, но с таким же успехом решение может приниматься на основе любых других проверок содержимого сообщения. Мы не показываем здесь код реализации этой версии, потому что он достаточно прост, чтобы вы, с текущим уровнем знания фреймворка Akka,

смогли написать его самостоятельно. В следующем разделе мы рассмотрим маршрутизацию на основе состояния.

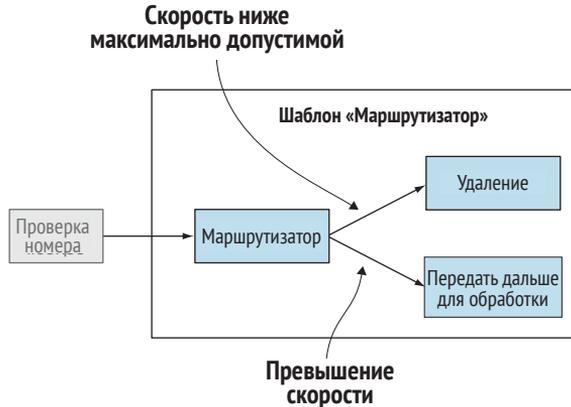


Рис. 9.6. Маршрутизация по значению скорости

### 9.3.2. Маршрутизация на основе состояния

Этот подход подразумевает изменение порядка маршрутизации при изменении состояния маршрутизатора. Простейшим примером может служить переключение маршрутизатора между двумя состояниями: `on` и `off`. Когда маршрутизатор находится в состоянии `on`, он пересылает сообщения как обычно, а состояние `off` передает все сообщения задаче удаления. Для реализации этого примера нельзя использовать маршрутизатор Akka, потому что нам требуется, чтобы маршрутизатор обладал состоянием, а состояние маршрутизаторов Akka по умолчанию не является потокобезопасным. Поэтому используем самый обычный актор. Состояние можно реализовать как атрибут класса, но поскольку есть возможность изменять поведение актора в течение его жизненного цикла с помощью методов `become/unbecome`, используем их в роли механизма представления состояния.

В нашем примере имеются два состояния, `on` и `off`. Когда актор находится в состоянии `on`, сообщения должны маршрутизироваться как обычно, а в состоянии `off` сообщения должны просто уничтожаться. Для этого создадим две функции обработки сообщений. Когда потребуется переключить состояние, мы просто заменим функцию приема с помощью метода `become` контекста актора. Для изменения состояния в данном примере используются два сообщения: `RouteStateOn` и `RouteStateOff`.

#### Листинг 9.13. Маршрутизация на основе состояния

```
case class RouteStateOn()
case class RouteStateOff()

class SwitchRouter(normalFlow: ActorRef, cleanUp: ActorRef)
```

```

extends Actor with ActorLogging {
  def on: Receive = {
    case RouteStateOn =>
      log.warning("Received on while already in on state")
    case RouteStateOff => context.become(off)
    case msg: AnyRef => {
      normalFlow ! msg
    }
  }
  def off: Receive = {
    case RouteStateOn => context.become(on)
    case RouteStateOff =>
      log.warning("Received off while already in off state")
    case msg: AnyRef => {
      cleanup ! msg
    }
  }
  def receive = off
}

```

← Метод Receive для состояния on  
 ← Переключение в состояние off  
 ← В состоянии on передавать сообщения для дальнейшей обработки  
 ← Метод Receive для состояния off  
 ← Переключение в состояние on  
 ← В состоянии off передавать сообщения для удаления  
 ← Актор запускается в состоянии off

Сразу после запуска актор находится в состоянии `off`. В этом состоянии все принимаемые сообщения актор пересылает актору `cleanup`. Если послать нашему маршрутизатору сообщение `RouteStateOn`, он вызовет метод `become` и заменит функцию приема реализацией `on`. После этого все сообщения будут пересылаться актору, представляющему обычный поток обработки сообщений.

#### Листинг 9.14. Тестирование актора `routerRedirect`

```

val normalFlowProbe = TestProbe()
val cleanupProbe = TestProbe()
val router = system.actorOf(
  Props(new SwitchRouter(
    normalFlow = normalFlowProbe.ref,
    cleanup = cleanupProbe.ref)))

val msg = "message"
router ! msg

cleanupProbe.expectMsg(msg)
normalFlowProbe.expectNoMsg(1 second)

router ! RouteStateOn
router ! msg

cleanupProbe.expectNoMsg(1 second)

```

← Переключение в состояние on

```
normalFlowProbe.expectMsg(msg)
```

```
router ! RouteStateOff
router ! msg
```

← Переключение  
в состояние off

```
cleanupProbe.expectMsg(msg)
normalFlowProbe.expectNoMsg(1 second)
```

В нашем примере использовался только метод `become`, но, кроме него, есть также метод `unbecome`. Если вызвать этот метод, он удалит новую функцию приема и вместо нее подставит исходную. Давайте перепишем наш маршрутизатор и применим метод `unbecome`. (Это не только вопрос семантики, но также следования имеющимся соглашениям.)

**Листинг 9.15.** Маршрутизация на основе состояния с применением `unbecome`

```
class SwitchRouter2(normalFlow: ActorRef, cleanUp: ActorRef)
  extends Actor with ActorLogging {
```

```
  def on: Receive = {
    case RouteStateOn =>
      log.warning("Received on while already in on state")
    case RouteStateOff => context.unbecome()
    case msg: AnyRef => normalFlow ! msg
  }
```

← Вызов метода `unbecome`  
вместо `become(off)`

```
  def off: Receive = {
    case RouteStateOn => context.become(on)
    case RouteStateOff =>
      log.warning("Received off while already in off state")
    case msg: AnyRef => cleanUp ! msg
  }
```

```
  def receive = {
    case msg: AnyRef => off(msg)
  }
```

```
}
```

Есть одно важное замечание по поводу использования метода `become`: после перезапуска актор возвращается в исходное состояние. Методы `become/unbecome` могут с успехом использоваться, когда требуется менять поведение в ходе обработки сообщений.

### 9.3.3. Реализации маршрутизаторов

До сих пор мы рассматривали разные примеры маршрутизаторов и способы их реализации. Но все они представляли чистые реализации шаблона «Маршрутизатор»; сами маршрутизаторы никак не обрабатывали сообщения, а просто пересылали их соответствующим адресатам. Это ти-

пичное решение на этапе предварительного проектирования, но иногда целесообразнее включить маршрутизацию прямо в актор, обрабатывающий сообщения, как показано на рис. 9.7. Такой прием имеет смысл, когда результаты обработки могут влиять на выбор следующего этапа.

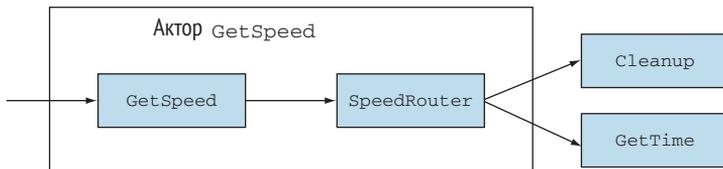


Рис. 9.7. Многократная реализация шаблона

В нашем примере с дорожной камерой актор `GetSpeed` определял скорость. Когда он терпел неудачу или скорость оказывалась ниже установленного порога, сообщение передавалось задаче удаления; иначе сообщение передавалось для дальнейшей обработки, в нашем случае – актору `GetTime`. Чтобы воплотить эту схему, нам потребуется реализовать два шаблона:

- задача обработки;
- маршрутизатор.

При этом оба компонента, `GetSpeed` и `SpeedRouter`, можно реализовать в одном акторе. Сначала актор выполняет задачу обработки и по ее результату определяет, куда дальше отправить сообщение – задаче `GetTime` или `Cleanup`. Решение о реализации этих компонентов в одном акторе или в двух зависит от требований к возможности повторного использования. Если необходимо, чтобы `GetSpeed` была отдельной функцией, мы не сможем совместить оба шага в одном акторе. Но если обрабатывающий актор также должен принимать решение о дальнейшей обработке сообщения, проще будет совместить эти два компонента. Другим фактором могло бы быть разделение обычного потока обработки и потока обработки ошибок для компонента `GetSpeed`.

## 9.4. В заключение

Эта глава была целиком посвящена вопросам маршрутизации сообщений между разными задачами. Маршрутизаторы `Акка` – важный механизм масштабирования приложений, и они обладают большой гибкостью, особенно когда их настройка осуществляется с помощью файла конфигурации. Вы увидели, что на выбор имеется несколько встроенных реализаций логики работы. В этой главе вы также узнали, что:

- маршрутизаторы `Акка` имеют две разновидности: группы и пулы. Пулы управляют созданием и завершением маршрутов, а группы переключают эту обязанность на вас;

- маршрутизаторы Akka с успехом могут использовать удаленные акторы в качестве маршрутов;
- маршрутизатор, опирающийся на свое состояние, который мы реализовали с использованием методов `become/unbecome`, позволяет менять поведение актора, замещая его метод `receive`. Используя этот подход, нужно быть особенно внимательными со случаями перезапуска, потому что после перезапуска в акторе устанавливается исходная реализация функции `receive` обработки сообщений;
- выбор маршрута может основываться на производительности, содержанием сообщения или состоянии маршрутизатора.

В этой главе мы все внимание сосредоточили на структуре шагов внутри приложения и на том, как смоделировать поток программы с использованием средств Akka. В следующей главе мы сосредоточимся на технологиях передачи сообщений между акторами, и вы узнаете, что существует много разных способов передачи сообщений, не только с использованием ссылок на акторы.

# Глава 10

## Каналы обмена сообщениями

В этой главе:

- прямой обмен сообщениями через каналы вида «точка-точка»;
- гибкий обмен сообщениями через каналы вида «издатель/подписчик»;
- публикация и подписка в EventBus;
- чтение недоставленных сообщений с использованием специального канала недоставленных сообщений;
- увеличение надежности доставки с использованием каналов гарантированной доставки;
- гарантированная доставка с ReliableProxy.

В этой главе мы рассмотрим каналы, которые можно использовать для передачи сообщений между акторами. Сначала мы познакомимся с двумя типами каналов: «точка-точка» и «издатель/подписчик». Каналы «точка-точка» мы использовали во всех предыдущих примерах в этой книге, но иногда нужен более гибкий способ отправки сообщений получателям. В разделе, посвященном каналам типа «издатель/подписчик», описывается метод отправки сообщений нескольким получателям, не требующий, чтобы отправитель заранее знал, кому адресовано сообщение. Получатели подключаются к каналу и могут меняться в процессе работы приложения. Для каналов этого вида также часто используются такие имена, как EventQueue и EventBus. В Akka имеется класс EventStream, реализующий каналы вида «издатель/подписчик». Но для случаев, когда возможностей этой реализации оказывается недостаточно, в Akka имеется коллекция трейтов, помогающих реализовать свой канал вида «издатель/подписчик».

Затем будут представлены два особых вида каналов. Первый – *канал недоставленных сообщений* – хранит сообщения, которые не были доставлены. Иногда его также называют *очередью недоставленных сообщений*. Этот

канал может помочь в отладке ситуаций, когда некоторые сообщения по непонятной причине не обрабатываются, или для мониторинга некоторых проблем в процессе выполнения. В последнем разделе описывается тип каналов *гарантированной доставки*. Нельзя создать надежную систему, не давая хотя бы минимальных гарантий доставки сообщений. Но жесткие гарантии доставки нужны далеко не всегда. Akka не обеспечивает полной гарантии доставки, но мы опишем тот уровень, какой поддерживается, отличающийся для локальных и удаленных акторов.

## 10.1. Типы каналов

Начнем эту главу с описания двух типов каналов. Первый – каналы «точка-точка». Название точно определяет характеристики каналов этого типа: они связывают одну точку (отправителя) с другой (получателем). Часто этого достаточно, но иногда нужно послать сообщение сразу нескольким получателям. В этом случае придется создать несколько каналов или использовать каналы второго типа, «издатель/подписчик». Главное преимущество каналов этого типа – возможность динамически изменять состав получателей во время работы приложения. Каналы этого вида в Akka реализует класс `EventBus`.

### 10.1.1. Точка-точка

Канал переносит сообщение от отправителя до получателя. С помощью канала «точка-точка» можно передать сообщение только одному получателю. Мы использовали каналы этого вида во всех предыдущих примерах в этой книге, поэтому здесь мы только напомним некоторые важные аспекты, чтобы подчеркнуть отличия между двумя типами каналов.

В предыдущих примерах отправитель знал следующий этап обработки сообщения и мог выбирать тот или иной канал. Иногда отправителю доступен только один канал, как в примере реализации конвейера с фильтрами в разделе 8.1, где отправитель имел только одну ссылку `ActorRef` для отправки сообщения после обработки. Но в других случаях, например в примере реализации списка получателей в разделе 8.2.3, актор имеет несколько каналов и решает, какой канал использовать для отправки сообщения. То есть связь между акторами имеет статичную природу.

Другая особенность канала этого вида – сообщения доставляются получателю в том же порядке, в каком были отправлены. Канал «точка-точка» доставляет сообщение точно одному получателю, как показано на рис. 10.1.

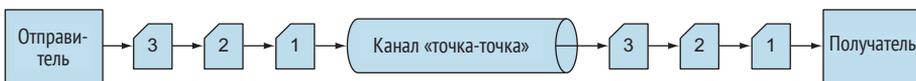


Рис. 10.1. Канал «точка-точка»

Канал «точка-точка» может иметь несколько получателей, но даже в этом случае каждое сообщение доставляется только одному получателю. Примером такого канала может служить циклический маршрутизатор из раздела 9.2.1. Обработка сообщений может выполняться конкурентно разными получателями, но только один получит каждое конкретное сообщение. Такая организация канала показана на рис. 10.2.

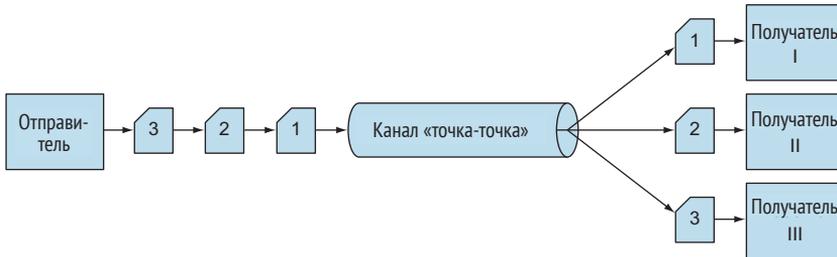


Рис. 10.2. Канал «точка-точка» с несколькими получателями

Канал имеет нескольких получателей, но каждое сообщение доставляется только одному из них. Каналы этого вида используются, когда связи между отправителями и получателями имеют статичную природу – отправитель знает, какой канал использовать, чтобы передать сообщение тому или иному получателю.

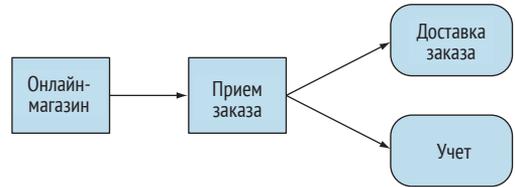
Каналы этого вида чаще других используются в Akka, потому что фактически `ActorRef` является реализацией канала «точка-точка». Все посылаемые сообщения доставляются одному актору и в том же порядке, в каком они были отправлены.

### 10.1.2. Издатель/подписчик

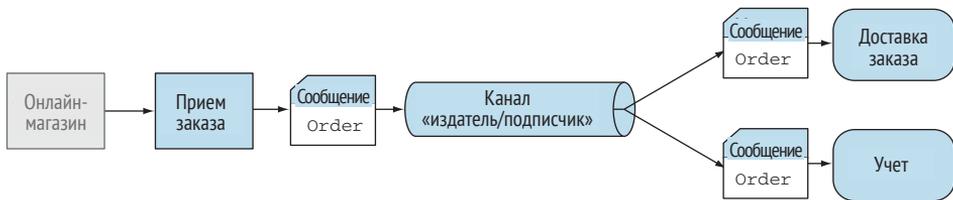
Каналы «точка-точка» доставляют каждое сообщение только одному получателю, как описывалось в предыдущем разделе. Используя каналы этого вида, отправитель знает адресата. Но иногда отправитель может не знать, кому интересны его сообщения. В этом заключается самое большое отличие каналов «издатель/подписчик» от каналов «точка-точка». Канал принимает на себя обязанность следить за получателями, заинтересованными в сообщениях, и снимает ее с отправителя. Он также может доставить одно сообщение нескольким получателям.

Допустим, у нас есть приложение онлайн-магазина. На первом шаге приложение принимает заказ. После этого система должна передать заказ на следующий этап – этап доставки заказа (например, книги) заказчику. Этап приема заказа посылает сообщение этапу доставки заказа. Но для учета товаров на складе мы также должны отправить заказ компоненту учета. То есть полученный заказ должен быть отправлен двум компонентам системы, как показано на рис. 10.3.

В качестве поощрительной меры мы решили делать клиентам маленькие подарки, когда они покупают книги. Для этого мы добавим в систему компонент отправки подарков, которому тоже нужно посылать сообщение с заказом. Каждый раз, когда добавляется новая подсистема, мы должны изменять первый шаг, чтобы добавить в него отpravку сообщений дополнительным получателям. Избавиться от этой проблемы нам поможет канал «издатель/подписчик». Этот канал способен посылать одно и то же сообщение нескольким получателям, не требуя от отправителя хоть что-нибудь знать о получателях. На рис. 10.4 показано, как опубликованное сообщение пересылается подсистемам доставки и учета.



**Рис. 10.3.** Обработка сообщения с заказом в онлайн-магазине



**Рис. 10.4.** Использование канала «издатель/подписчик» для распространения сообщения с заказом

Когда понадобится добавить возможность отправки подарков, мы просто подпишем этот компонент на получение сообщений, и нам не придется ничего менять в компоненте приема заказов. Еще одно преимущество каналов этого вида – количество и состав получателей может меняться в процессе работы. Например, подарки могут посылаться не всегда, а только в дни проведения акций. Используя этот канал, мы сможем добавлять компонент подарков в число подписчиков канала только в определенные периоды и исключать из подписчиков после окончания акции, как показано на рис. 10.5.

Если получатель заинтересован в доставке сообщений от издателя, он подписывает себя в канале. Когда издатель посылает сообщение, канал гарантирует его доставку всем подписчикам. С окончанием акции компонент рассылки подарков больше не нуждается в сообщениях с заказами, поэтому он аннулирует свою подписку на канал. Как результат методы канала можно разбить на две категории по их использованию. К первой относятся методы, используемые стороной-отправителем для публикации сообщений. Ко второй относятся методы, используемые стороной-получателем для управления подпиской. На рис. 10.6 показаны эти две категории методов.

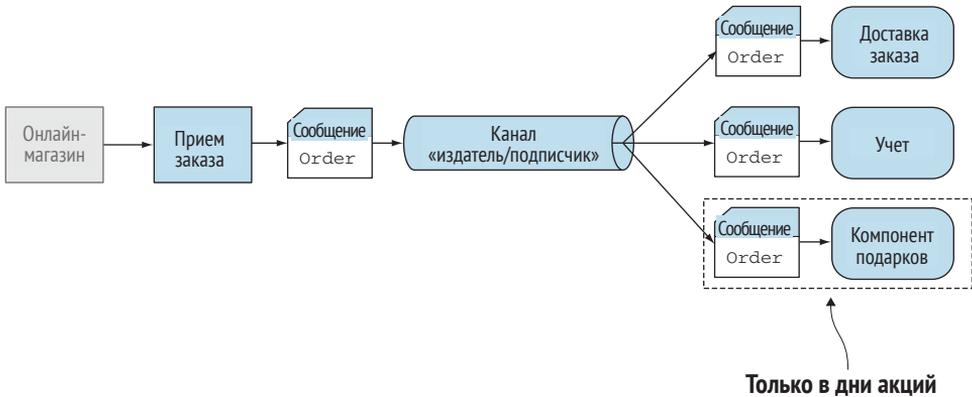


Рис. 10.5. Компонент подарков получает сообщения только в дни акций

Так как получатели могут самостоятельно подписывать себя на получение сообщений из канала, данное решение обладает большой гибкостью. Издателю не нужно знать, сколько подписчиков у него имеется. Может даже так получиться, что в какой-то момент у него не будет ни одного подписчика.



Рис. 10.6. Две категории методов канала «издатель/подписчик»

## EventStream

В Akka поддержка каналов «издатель/подписчик» реализуется объектом `EventStream`. Каждая система акторов `ActorSystem` имеет такой объект, доступный любому актору (как `context.system.eventStream`). `EventStream` можно считать диспетчером нескольких каналов «издатель/подписчик», потому что любой актор может подписаться на сообщения определенного типа, и когда какой-то другой актор опубликует сообщение этого типа, подписчик получит его. Актор не обязательно должен изменять сообщения, получаемые из `EventStream`.

```
class DeliverOrder() extends Actor {
  def receive = {
    case msg: Order => ...// Обработка сообщения
```

```

}
}

```

В данном случае необычным выглядит способ отправки сообщений. Кроме того, чтобы начать получать сообщения, необязательно, чтобы подписка оформлялась самим актором. Подписка может быть выполнена в любом месте в системе, где доступны ссылки на актор и `EventStream`. На рис. 10.7 показано использование интерфейса подписки в Akka. Чтобы подписать актор на получение сообщений `Order`, нужно вызвать метод `subscribe` объекта `EventStream`.

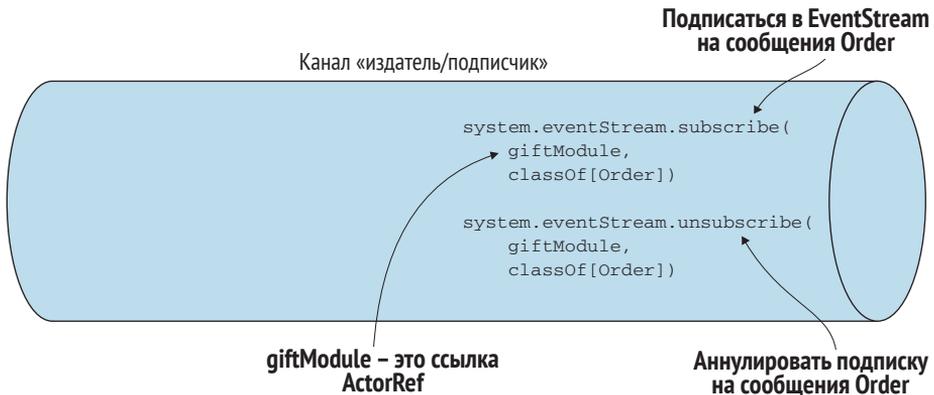


Рис. 10.7. Интерфейс подписки в `EventStream`

Когда необходимость в подписке отпадет, например когда завершится подарочная акция, можно воспользоваться методом `unsubscribe`. В данном примере мы аннулировали подписку компонента `GiftModule`, и после вызова этого метода актор перестанет получать сообщения `Order`.

Это все, что требуется, чтобы подписать компонент `GiftModule` на получение сообщений `Order`. После вызова метода `subscribe` компонент `GiftModule` будет получать все сообщения `Order`, публикуемые в `EventStream`. Этот метод можно вызвать для любого актора, заинтересованного в получении сообщений `Order`. А когда актор должен получать сообщения разных типов, метод `subscribe` можно вызвать несколько раз с разными типами сообщений.

Публикация сообщений в `EventStream` выполняется так же просто; достаточно вызвать метод `publish`, как показано на рис. 10.8. После этого сообщение `msg` будет передано всем акторам-подписчикам. По сути, этим исчерпывается реализация каналов вида «издатель/подписчик» в Akka.

В Akka имеется возможность подписаться сразу на несколько типов событий. Например, наш компонент `GiftModule` также должен обрабатывать сообщения об отмене заказа, потому что в этом случае подарок не должен отправляться. Для этого компонент `GiftModule` должен подписаться в

EventStream на получение сообщений Order и Cancel. Обе подписки действуют независимо, то есть после аннулирования подписки на сообщения Orders подписка на сообщения Cancel продолжит действовать, и эти сообщения будут доставляться.

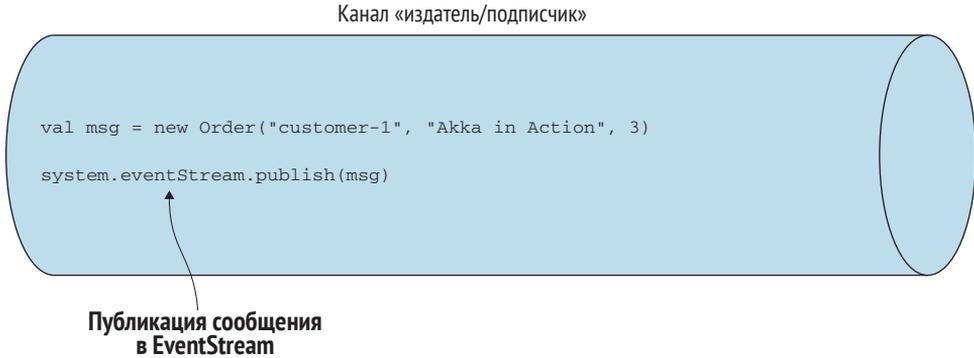


Рис. 10.8. Интерфейс публикации в EventStream

Перед остановкой компонента GiftModule нужно аннулировать все его подписки. Это можно сделать единственным вызовом:

```
system.eventStream.unsubscribe(giftModule)
```

После этого GiftModule не будет подписан ни на какие сообщения. Методы publish, subscribe и unsubscribe образуют интерфейс канала «издатель/подписчик» в Akka, который прост и понятен. В листинге 10.1 показано, как протестировать доставку сообщений Order через EventStream.

#### Листинг 10.1. EventStream в действии

```
val DeliverOrder = TestProbe()
val giftModule = TestProbe()

system.eventStream.subscribe(
  DeliverOrder.ref,
  classOf[Order])
system.eventStream.subscribe(
  giftModule.ref,
  classOf[Order])

val msg = new Order("me", "Akka in Action", 3)
system.eventStream.publish(msg)

DeliverOrder.expectMsg(msg)
giftModule.expectMsg(msg)
```

Создание  
актеров-получателей

Подписка актеров-получателей  
на сообщения Order

Опубликовать  
сообщение Order

Сообщение должно быть  
получено обоими актерами

```

system.eventStream.unsubscribe(giftModule.ref)
system.eventStream.publish(msg)
DeliverOrder.expectMsg(msg)
giftModule.expectNoMsg(3 seconds)

```

← Отписать GiftModule

← GiftModule больше не получит сообщения

В роли получателей здесь использованы акторы `TestProbe`. Оба они подписываются на сообщения `Order`. После публикации первого сообщения в `EventStream` оба актора получают по сообщению. А после аннулирования подписки для `GiftModule` сообщения будет получать только `DeliverOrder`, как и следовало ожидать.

Мы уже говорили о преимуществах слабой связи между отправителем и получателями, а также динамической природе канала «издатель/подписчик». А так как `EventStream` доступен всем акторам, он также является отличным решением для случаев, когда все сообщения со всех концов локальной системы должны стекаться в один или несколько акторов. Ярким примером может служить журналирование. Все журналируемые сообщения должны стекаться в одну точку и записываться в файл журнала. Внутренне `ActorLogging` использует `EventStream` для сбора журнальных записей со всей системы.

Канал `EventStream` удобен в использовании, но иногда требуется иметь более полный контроль, и тогда приходится писать реализацию своего канала «издатель/подписчик». В следующем подразделе мы покажем, как это делается.

### Свой канал `EventBus`

Допустим, мы решили дарить подарки только покупателям, заказавшим две или более книг. В этом случае компонент `GiftModule` должен получать сообщение, только когда количество экземпляров в заказе больше 1. При использовании `EventStream` мы не можем выполнить такую фильтрацию, потому что `EventStream` проверяет только типы сообщений. Мы могли бы организовать проверку внутри `GiftModule`, но предположим, что такая проверка потребляет слишком много ресурсов, что для нас недопустимо. В подобной ситуации остается только реализовать свой канал «издатель/подписчик», и `Akka` обеспечит нас необходимой поддержкой.

В `Akka` имеется обобщенный интерфейс `EventBus`, реализовав который, можно создать свой канал «издатель/подписчик». Интерфейс `EventBus` обобщен так, что может использоваться для реализации любых таких каналов. В обобщенной форме он включает в себя три сущности:

- *Событие* – тип всех событий, публикуемых в этом канале. В `EventStream` используется тип `AnyRef`, что означает, что событием может быть любой ссылочный тип.

- *Подписчик* – тип подписчика, которому позволено подписываться на получение событий. В `EventStream` типом подписчика является тип `ActorRefs`.
- *Классификатор* – определяет классификатор для выбора подписчиков при попытке доставить очередное событие. В `EventStream` классификатором служит тип конкретного сообщения.

Если изменить определения этих сущностей, можно создать канал «издатель/подписчик» с любой функциональностью. Интерфейс включает элементы для определения всех трех сущностей и разных методов публикации и подписки, которые также доступны в `EventStream`. В листинге 10.2 приводится полное определение интерфейса `EventBus`.

#### Листинг 10.2. Интерфейс `EventBus`

```
package akka.event

trait EventBus {
  type Event
  type Classifier
  type Subscriber

  /**
   * Проверяет зарегистрировать подписчика с указанным классификатором
   * @return true в случае успеха, false иначе (потому что
   * подписчик уже подписан с этим классификатором или по иной причине)
   */
  def subscribe(subscriber: Subscriber, to: Classifier): Boolean

  /**
   * Проверяет отписать подписчика от указанного классификатора
   * @return true в случае успеха, false иначе (потому что
   * подписчик не подписан с этим классификатором или по иной причине)
   */
  def unsubscribe(subscriber: Subscriber, from: Classifier): Boolean

  /**
   * Проверяет отписать подписчика от всех классификаторов
   */
  def unsubscribe(subscriber: Subscriber): Unit

  /**
   * Публикует указанное событие Event в этом канале
   */
  def publish(event: Event): Unit
}
```

Этот интерфейс должен быть реализован полностью, а поскольку во многих случаях требуются одни и те же возможности, в Akka было добавлено несколько составных трейтов, реализующих интерфейс `EventBus`, которые можно использовать, чтобы облегчить себе труд по реализации своей версии `EventBus`.

Давайте напишем свою реализацию `EventBus` для нашего компонента `GiftModule`, которая будет передавать только сообщения `Orders` с несколькими книгами в заказе. Наша версия `EventBus`, назовем ее `OrderMessageBus`, должна посылать и принимать только сообщения `Orders`; поэтому используем в качестве типа события класс `Order`, как показано ниже:

```
class OrderMessageBus extends EventBus {
  type Event = Order
}
```

Следующая сущность, которую мы должны определить, – классификатор. По условиям задачи мы должны различать заказы с одной книгой и заказы с несколькими книгами. Мы решили выбрать для классификации сообщений `Order` критерий «несколько книг в заказе» и использовать логический классификатор. То есть мы должны определить классификатор как `Boolean`:

```
class OrderMessageBus extends EventBus {
  type Event = Order
  type Classifier = Boolean
}
```

Пока пропустим определение подписчика, потому что мы будем определять его немного иначе. Итак, мы определили классификатор и теперь должны научиться выбирать подписчиков для всех возможных значений классификатора. Наш классификатор «несколько книг в заказе» может принимать два значения: `true` или `false`. В Akka имеется три составных трейта, которые помогают организовать выбор подписчиков. Все трейты остаются обобщенными, поэтому их можно использовать с любыми сущностями, которые вы определите. Достигается это добавлением новых абстрактных методов.

- `LookupClassification` – этот трейт использует самый простой способ классификации. Он хранит множество подписчиков для каждого возможного значения классификатора и извлекает классификатор из каждого события. Извлечение классификатора производится вызовом метода `classify`, который должен присутствовать в нестандартной реализации `EventBus`.
- `SubchannelClassification` – этот трейт используется, когда классификатор образует иерархию и необходимо обеспечить подписку не

только на листы иерархии, но и на промежуточные узлы. Этот трейт используется в реализации `EventStream`, потому что классы образуют иерархию и иногда желательно использовать суперкласс, чтобы подписаться на классы-наследники.

- `ScanningClassification` – это более сложный трейт; его можно использовать, когда значения классификатора могут перекрываться. То есть одно событие `Event` может быть частью нескольких классификаторов; с помощью этого трейта мы могли бы, например, организовать увеличение ценности подарков с увеличением числа книг в заказах: заказав две или более книг, клиент получит маркер в подарок, а заказав более 10 книг – еще и купон на следующий заказ. То есть когда заказ содержит 11 книг, он становится частью сразу двух классификаторов: «несколько книг в заказе» и «более 10 книг в заказе». Опубликованное сообщение с таким заказом должно быть доставлено не только подписчикам с классификатором «несколько книг в заказе», но и подписчикам с классификатором «более 10 книг в заказе».

В нашей реализации мы используем `LookupClassification`. Другие два способа классификации реализуются аналогично этому. Эти трейты реализуют методы `subscribe` и `unsubscribe` интерфейса `EventBus`. Но они также определяют новые абстрактные методы, которые должны быть реализованы в нашем классе. При использовании `LookupClassification` мы должны реализовать следующие методы:

- `classify(event: Event): Classifier` – используется для извлечения классификатора из входящих событий;
- `compareSubscribers(a: Subscriber, b: Subscriber): Int` – этот метод должен определять порядок сортировки подписчиков, подобно методу `compare` в `java.lang.Comparable`;
- `publish(event: Event, subscriber: Subscriber)` – этот метод будет вызван для каждого события, для всех подписчиков, зарегистрировавшихся с классификатором события;
- `mapSize: Int` – возвращает ожидаемое число разных классификаторов; используется для определения начального размера внутренней структуры данных.

Как уже говорилось, мы используем классификатор «несколько книг в заказе». То есть он имеет два возможных значения; поэтому используем значение 2 для `mapSize`:

```
import akka.event.{LookupClassification, EventBus}
class OrderMessageBus extends EventBus with LookupClassification {
  type Event = Order
  type Classifier = Boolean
```

```

def mapSize = 2
protected def classify(event: StateEventBus#Event) = {
  event.number > 1
}
}

```

Определить mapSize как число 2

Вернуть классификатор true, если number больше 1, и false в противном случае

Выше уже упоминалось, что `LookupClassification` должен иметь возможность получить классификатор из события. Это делается с помощью метода `classify`. В данном случае он просто возвращает результат проверки `event.number > 1`.

Теперь нам осталось только определить подписчика; для этого воспользуемся трейтом `ActorEventBus`. Это, пожалуй, самый часто используемый трейт в системе сообщений Akka, потому что он определяет, что подписчиком является `ActorRef`. Он также реализует метод `compareSubscribers`, необходимый трейту `LookupClassification`. Единственный метод, который мы должны реализовать, – это метод `publish`. Законченная реализация нашего канала приводится в листинге 10.3.

### Листинг 10.3. Законченная реализация `OrderMessageBus`

```

import akka.event.ActorEventBus
import akka.event.{ LookupClassification, EventBus }

class OrderMessageBus extends EventBus
  with LookupClassification
  with ActorEventBus {

  type Event = Order
  type Classifier = Boolean
  def mapSize = 2

  protected def classify(event: OrderMessageBus#Event) = {
    event.number > 1
  }

  protected def publish(event: OrderMessageBus#Event,
    subscriber: OrderMessageBus#Subscriber): Unit = {
    subscriber ! event
  }
}

```

Расширение класса поддержки двух трейтов из Akka

Определение сущностей

Реализация метода classify

Реализация метода publish для отправки событий подписчикам

Мы завершили реализацию своей версии `EventBus`, которую можно применять для подписки на сообщения и их публикации. В листинге 10.4 показано, как можно использовать эту версию `EventBus`.

**Листинг 10.4.** Использование OrderMessageBus

```

val bus = new OrderMessageBus
val singleBooks = TestProbe()
bus.subscribe(singleBooks.ref, false)
val multiBooks = TestProbe()
bus.subscribe(multiBooks.ref, true)

val msg = new Order("me", "Akka in Action", 1)
bus.publish(msg)
singleBooks.expectMsg(msg)
multiBooks.expectNoMsg(3 seconds)

val msg2 = new Order("me", "Akka in Action", 3)
bus.publish(msg2)
singleBooks.expectNoMsg(3 seconds)
multiBooks.expectMsg(msg2)

```

← Создать OrderMessageBus  
 ← Подписать singleBooks на классификатор «одна книга» (false)  
 ← Подписать multiBooks на классификатор «несколько книг» (true)  
 ← Опубликовать заказ на одну книгу  
 ← Сообщение получит только singleBooks  
 ← После публикации заказа на несколько книг сообщение получит только multiBooks

Как видите, наша версия `EventBus` работает точно так же, как `EventStream`, но использует другой классификатор. В Akka имеется еще несколько трейтов, которые можно использовать для конструирования своих каналов обмена сообщениями. Более полную информацию о них ищите в документации к Akka.

Как вы увидели в этом разделе, Akka обладает поддержкой каналов виде «издатель/подписчик». В большинстве случаев с успехом можно использовать встроенный канал `EventStream`. Но если вам понадобится что-то более специализированное, вы сможете написать свою реализацию интерфейса `EventBus`. Это обобщенный интерфейс, который можно реализовать, как вам будет угодно. Для поддержки нестандартных реализаций `EventBus` в Akka имеется несколько трейтов, реализующих отдельные части интерфейса `EventBus`.

В этом разделе вы увидели два основных типа каналов. В следующем разделе мы познакомимся с некоторыми специальными каналами.

## 10.2. Специальные каналы

В этом разделе мы рассмотрим два специальных канала. Первым обсудим канал `DeadLetter`. В этот канал передаются только сообщения, которые не удалось доставить. Проверка этого канала может помочь найти проблему в системе.

Вторым мы обсудим канал с гарантированной доставкой, который позволяет вновь и вновь повторять отправку сообщений, пока их получение не будет подтверждено.

### 10.2.1. DeadLetter

DeadLetter – это канал, или очередь, недоставленных сообщений. Он хранит все сообщения, которые не были обработаны или доставлены. Это самый обычный канал, но чаще всего он не используется для отправки сообщений. Только когда обнаруживаются какие-то проблемы с сообщением, например если оно не может быть доставлено, такое сообщение помещается в данный канал, как показано на рис. 10.9.

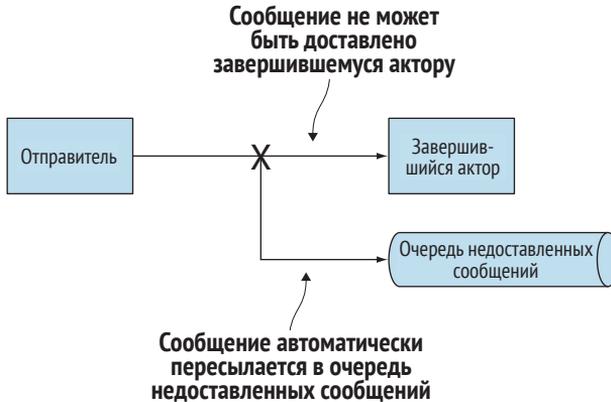


Рис. 10.9. Очередь недоставленных сообщений

Осуществляя мониторинг этого канала, можно узнать, какие сообщения не были обработаны, и предпринять необходимые действия. В частности, в ходе тестирования системы с помощью этой очереди можно выяснить, почему некоторые сообщения остаются необработанными. Если система не должна просто сбрасывать какие-либо сообщения, эту очередь можно использовать для повторной отправки сообщений после решения проблем.

Для реализации очереди недоставленных сообщений в Akka используется `EventStream`, поэтому только акторы могут подписаться на сообщения из этой очереди. Когда сообщение доставляется в почтовый ящик уже завершившегося актора или посылается после его завершения, оно попадает в `EventStream` системы акторов `ActorSystem`. Сообщение заворачивается в объект `DeadLetter`, содержащий оригинальное сообщение, адрес отправителя и адрес получателя. То есть фактически очередь недоставленных сообщений интегрирована в общую очередь `EventStream`. Чтобы получить недоставленные сообщения, достаточно подписать актора на получение из `EventStream` сообщений с классификатором `DeadLetter`. Этот прием описывался в предыдущем разделе, только здесь мы используем другой тип сообщений – `DeadLetter`:

```
val deadletterMonitor: ActorRef = ...
```

← Трехточие (...) – любое выражение, возвращающее ссылку на актор `ActorRef`

```
system.eventStream.subscribe(
```

```

    deadLetterMonitor,
    classOf[DeadLetter]
)

```

После этого вызова `subscribe` актер `deadLetterMonitor` будет получать сообщения, которые не удалось доставить адресату. Рассмотрим небольшой пример. Создадим простой актер `Echo`, возвращающий любые сообщения обратно отправителю, и после его запуска сразу же пошлем ему сообщение `PoisonPill`. Это приведет к завершению актора. Листинг 10.5 показывает, что сообщение можно получить, подписавшись на очередь `DeadLetter`.

### Листинг 10.5. Получение недоставленных сообщений

```

val deadLetterMonitor = TestProbe()    ← Подписка на канал DeadLetter

system.eventStream.subscribe(
  deadLetterMonitor.ref,
  classOf[DeadLetter]
)

val actor = system.actorOf(Props[EchoActor], "echo")
actor ! PoisonPill
val msg = new Order("me", "Akka in Action", 1)
actor ! msg

val dead = deadLetterMonitor.expectMsgType[DeadLetter]
dead.message must be(msg)
dead.sender must be(testActor)
dead.recipient must be(actor)

```

Сообщения, посланные завершившемуся актору, не могут быть обработаны, и ссылка `ActorRef` на этот актер не должна больше использоваться. Когда сообщения посылаются завершившемуся актору, они попадают в очередь `DeadLetter`. Это подтверждает получение сообщения нашим актором `deadLetterMonitor`.

Очередь `DeadLetter` можно также использовать для хранения сообщений, которые не удалось обработать. Но решение об этом принимается на уровне каждого конкретного актора. Актер может выяснить, что не способен обработать полученное сообщение и не знает, что с ним делать. В такой ситуации сообщение можно послать в очередь недоставленных сообщений. В системе акторов `ActorSystem` имеется ссылка на актер `DeadLetter`. Когда возникает необходимость послать сообщение в очередь недоставленных сообщений, его можно передать этому актору:

```

system.deadLetters ! msg

```

Получив сообщение, актер `DeadLetter` завернет его в объект `DeadLetter`. Но только в этом случае оригинальным получателем станет сам актер `DeadLetter`. То есть когда сообщения посылаются в очередь недоставленных сообщений таким способом, информация о первоначальном отправителе теряется; отправителем становится актер-получатель, пославший сообщение в очередь. Иногда этой информации может быть достаточно, но в некоторых случаях бывает важно знать, кто первоначально послал сообщение. Для этого можно сформировать объект `DeadLetter` и послать в очередь его, а не оригинальное сообщение. В этом случае этап обертывания пропускается, и сообщение передается в очередь без любых изменений. В листинге 10.6 мы посылаем объект `DeadLetter` и убеждаемся, что сообщение не было изменено.

### Листинг 10.6. Отправка сообщений `DeadLetter`

```
val deadLetterMonitor = TestProbe()
val actor = system.actorOf(Props[EchoActor], "echo")
system.eventStream.subscribe(
  deadLetterMonitor.ref,
  classOf[DeadLetter]
)

val msg = new Order("me", "Akka in Action", 1)
val dead = DeadLetter(msg, testActor, actor)
system.deadLetters ! dead

deadLetterMonitor.expectMsg(dead)
system.stop(actor)
```

Создание ссылки на актер, который будет играть роль оригинального получателя

Создание сообщения `DeadLetter` и отправка его актору `DeadLetter`

Сообщение `DeadLetter` будет получено монитором

Как показали испытания, сообщение `DeadLetter` действительно не изменилось. Это открывает возможность обрабатывать все сообщения, которые не удалось обработать стандартным способом или которые не были доставлены. Конкретный алгоритм обработки зависит от требований вашей системы. Иногда такие потерянные сообщения можно просто игнорировать, но при разработке высоконадежной системы вам может потребоваться повторно посылать сообщения оригинальному получателю.

В этом разделе мы узнали, как перехватывать сообщения, которые не удалось обработать. В следующем разделе мы познакомимся с еще одним специализированным каналом: каналом гарантированной доставки.

## 10.2.2. Гарантированная доставка

Канал гарантированной доставки – это канал вида «точка-точка», гарантирующий доставку сообщения получателю. Это означает, что доставка бу-

дет выполнена, даже если при этом возникнут любые мыслимые ошибки. Канал должен иметь разнообразные механизмы и проверки, чтобы гарантировать доставку; например, сообщение должно сохраняться на диске на случай аварийного завершения системы. Всегда ли при создании систем нужен канал гарантированной доставки? Можно ли создать высоконадежную систему, не гарантируя доставки сообщений в ней? Да, вам нужны определенные гарантии, но гарантии максимальной доступности нужны далеко не всегда.

Фактически реализация канала гарантированной доставки не может гарантировать доставку во всех возможных ситуациях, например когда в момент отправки сообщения возникает авария. В такой ситуации невозможно отправить сообщение, потому что оно исчезло вместе с отправителем. Вы постоянно должны задавать себе вопрос: достаточен ли уровень гарантий для моих целей?

Создавая систему, вы должны знать, какие гарантии дает канал и достаточно ли их для вашей системы. Давайте посмотрим, какие гарантии дает Akka.

Главное правило доставки сообщений: сообщение должно быть доставлено не более одного раза. Это означает, что Akka обещает доставить сообщение только один раз или не доставить его вообще, если оно будет потеряно. Это качество не особенно хорошо подходит для надежных систем. Почему в Akka отсутствует реализация с полной гарантией доставки? Первая причина в том, что полная гарантия доставки сопряжена с большим количеством сложностей и требует множества накладных расходов. Это ухудшает производительность, даже когда вам не нужны полные гарантии доставки.

Во-вторых, никому не нужна *просто* надежная доставка. Нам нужно знать, был ли обработан запрос, для чего требуется отправка подтверждающего сообщения. Это невозможно реализовать на уровне Akka, потому что многое зависит от конкретной системы. И последняя причина, почему Akka не реализует полностью гарантированную доставку, – потому что поверх базовых гарантий всегда можно наложить более строгие гарантии. Но обратное невозможно: нельзя сделать строгую систему менее строгой, не изменив ее ядра.

Фреймворк Akka не гарантирует точно одну доставку сообщения при любых условиях. Фактически дать такую гарантию не сможет ни одна система. Но она реализует базовое правило доставки сообщений локальным и удаленным акторам. Взглянув на эти две ситуации в отдельности, можно увидеть, что Akka действует не так плохо, как могло бы показаться.

Отправка локальных сообщений вообще почти не терпит неудач, потому что это самый обычный вызов метода. Неудача может быть обусловлена только чем-то катастрофическим, случившимся в недрах VM, например `StackOverflowError`, `OutOfMemoryError` или нарушением прав доступа к памя-

ти. Во всех этих случаях актер физически не будет иметь возможности обработать сообщение. Поэтому гарантии доставки сообщений локальным актерам достаточно высоки.

Потеря сообщений действительно может стать проблемой при использовании удаленных акторов. Неудачи доставки сообщений удаленным актерам более вероятны, особенно если они разделены ненадежными сетями. Если кто-то выдернет кабель Ethernet или выключит питание маршрутизатора, сообщение будет потеряно. Для решения этой проблемы был создан `ReliableProxy`. Он обеспечивает высокие гарантии доставки сообщений удаленным актерам, сопоставимые с гарантиями относительно локальной доставки. Единственное, что может помешать, – критические ошибки в виртуальных машинах JVM отправителя и получателя.

Как работает `ReliableProxy`? Когда `ReliableProxy` запускается, он создает туннель между двумя системами акторов `ActorSystem` на разных узлах.

Как показано на рис. 10.10, этот туннель имеет вход, `ReliableProxy`, и выход, `Egress`. `Egress` – это актер, который запускается актором `ReliableProxy`, и оба актора осуществляют проверки и повторную передачу сообщений для их гарантированной доставки удаленным получателям. Когда доставка терпит неудачу, `ReliableProxy` повторяет отправку сообщений, пока не добьется успеха. Когда `Egress` получает сообщение, он проверяет – было ли оно уже доставлено, и если нет, посылает получателю. Но что получится, если целевой актер завершится? В этом случае доставка станет невозможной. Эта проблема решается завершением `ReliableProxy` вместе с завершением целевого удаленного актора. В такой ситуации система ведет себя так же, как при использовании прямой ссылки. На стороне получателя разница между получением локальных сообщений или через прокси вообще не заметна. Единственное ограничение `ReliableProxy` – туннель является односторонним и связан только с одним получателем. Это означает, что когда получатель отвечает отправителю, туннель не используется. Чтобы обеспечить надежную доставку ответа, между получателем и отправителем нужно создать еще один туннель.

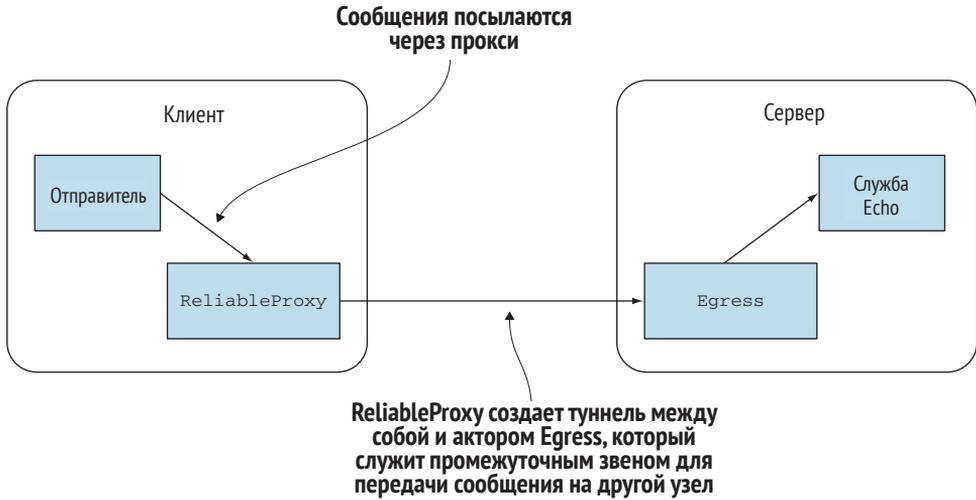
Теперь посмотрим, как эта схема работает на практике. Создать надежный прокси довольно просто, для этого нужна лишь ссылка на удаленный актер:

```
import akka.contrib.pattern.ReliableProxy

val pathToEcho = "akka.tcp://actorSystem@127.0.0.1:2553/user/echo"
val proxy = system.actorOf(
  Props(new ReliableProxy(pathToEcho, 500.millis)), "proxy")
```

Здесь мы создали прокси на основе ссылки `echo` и определили для параметра `retryAfter` значение 500 миллисекунд. Когда доставка сообщения потерпит неудачу, его отправка будет повторена через 500 миллисекунд.

Это все, что нужно, чтобы задействовать `ReliableProxy`. Чтобы продемонстрировать результат, создадим тест с двумя узлами – клиентом и сервером. На стороне сервера запустим актер `EchoActor`, который будет получателем, а на стороне клиента – сам тест. Так же, как в главе 6, нам понадобится конфигурация для нескольких узлов и `STMultiNodeSpec` для тестового класса `ReliableProxySample`.



**Рис. 10.10.** `ReliableProxy`

**Листинг 10.7.** Конфигурация для тестирования с несколькими узлами

```
import akka.remote.testkit.MultiNodeSpecCallbacks
import akka.remote.testkit.MultiNodeConfig
import akka.remote.testkit.MultiNodeSpec
```

```
trait STMultiNodeSpec
  extends MultiNodeSpecCallbacks
  with WordSpecLike
  with MustMatchers
  with BeforeAndAfterAll {

  override def beforeAll() = multiNodeSpecBeforeAll()

  override def afterAll() = multiNodeSpecAfterAll()
}
```

```
object ReliableProxySampleConfig extends MultiNodeConfig {
  val client = role("Client")
  val server = role("Server")
  testTransport(on = true)
```

← Определение клиентского узла  
 ← Определение серверного узла  
 ← Имитация ошибки, возникшей при транспортировке

```
}

```

```
class ReliableProxySampleSpecMultiJvmNode1 extends ReliableProxySample
class ReliableProxySampleSpecMultiJvmNode2 extends ReliableProxySample

```

Так как нам нужно показать, что сообщения посылаются, даже когда сеть временно становится недоступна, нужно добавить один `testTransport`. А также, как упоминалось, запустить `EchoService` на сервере:

```
system.actorOf(
  Props(new Actor {
    def receive = {
      case msg:AnyRef => {
        sender ! msg
      }
    }
  })),
"echo"
)
```

Эта служба возвращает любые полученные сообщения обратно отправителю. После ее запуска можно произвести тестирование на клиентском узле. Ниже показано создание полного окружения для тестирования.

#### Листинг 10.8. Создание окружения для теста `ReliableProxySample`

```
import scala.concurrent.duration._
import concurrent.Await
import akka.contrib.pattern.ReliableProxy
import akka.remote.testconductor.Direction

class ReliableProxySample
  extends MultiNodeSpec(ReliableProxySampleConfig)
  with STMultiNodeSpec
  with ImplicitSender {

  import ReliableProxySampleConfig._

  def initialParticipants = roles.size

  "A MultiNodeSample" must {

    "wait for all nodes to enter a barrier" in {
      enterBarrier("startup")
    }

    "send to and receive from a remote node" in {

```

```

runOn(client) {
  enterBarrier("deployed")
  val pathToEcho = node(server) / "user" / "echo"
  val echo = system.actorSelection(pathToEcho)
  val proxy = system.actorOf(
    Props(new ReliableProxy(pathToEcho, 500.millis)), "proxy")

  ... Фактический тест
}

runOn(server) {
  system.actorOf(Props(new Actor {
    def receive = {
      case msg:AnyRef => {
        sender ! msg
      }
    }
  })), "echo")
  enterBarrier("deployed")
}
enterBarrier("finished")
}
}
}

```

Создание прямой ссылки на службу Echo

Создание туннеля ReliableProxy

Реализация службы Echo

Теперь, имея полное тестовое окружение, можно реализовать фактический тест. В листинге 10.9 мы выполняем отправку сообщения в то время, как связь между узлами нарушена. В результате сообщение обрабатывается только при использовании прокси, при использовании прямой ссылки на актор сообщение теряется.

#### Листинг 10.9. Реализация ReliableProxySample

```

proxy ! "message1"
expectMsg("message1")
Await.ready(
  testConductor.blackhole( client, server, Direction.Both),
  1 second
)

echo ! "DirectMessage"
proxy ! "ProxyMessage"
expectNoMsg(3 seconds)

Await.ready(
  testConductor.passThrough( client, server, Direction.Both),
  1 second
)

```

Тестирование прокси при нормальных условиях

Разорвать связь между узлами

Послать сообщение по обеим ссылкам

Восстановить связь

```

)
expectMsg("ProxyMessage")
echo ! "DirectMessage2"
expectMsg("DirectMessage2")

```

Сообщение, посланное через прокси, получено

Тестовые сообщения, посылаемые по прямой ссылке, достигают целевого актора после восстановления связи

Использование `ReliableProxy` помогает обеспечить более высокие гарантии доставки сообщений удаленным акторам. Пока не происходит никаких критических ошибок в JVM на узлах системы и сеть в конечном итоге восстанавливает нормальное функционирование, сообщения доставляются целевому актору.

В этом разделе вы узнали, что в Akka нет специального канала гарантированной доставки, но все же Akka *может* дать некоторые гарантии. Для локальных акторов доставка гарантируется, если в виртуальной машине в этот момент не произойдет критических ошибок. Для удаленных акторов гарантируется доставка не более одного раза. Когда сообщения передаются через границы JVM, ситуацию можно улучшить с помощью `ReliableProxy`.

Этих гарантий доставки достаточно для большинства систем, но когда нужны более строгие гарантии, можно создать механизм поверх системы доставки в Akka. Этот механизм не реализован в Akka, потому что требования к нему обычно определяются на уровне конкретной системы, а потеря производительности из-за его работы часто ложится ненужным бременем, когда такие гарантии не требуются. Всегда есть сценарии, когда невозможно гарантировать доставку или когда гарантии требуется определять на прикладном уровне.

## 10.3. В заключение

Вы познакомились с двумя типами каналов для обмена сообщениями: «точка-точка», который передает сообщения конкретному получателю, и «издатель/подписчик», который может передавать сообщения нескольким получателям. Получатель может подписаться на канал, что обеспечивает динамическое распространение сообщений. Состав подписчиков может измениться в любой момент. В Akka имеется `EventStream` – реализация по умолчанию канала «издатель/подписчик», которая использует в качестве классификаторов классы сообщений. Существует также несколько трейтов, которые можно использовать для организации своих каналов «издатель/подписчик», когда возможностей канала `EventStream` оказывается недостаточно.

Вы также познакомились с каналом `DeadLetter` в Akka, который использует `EventStream`. Этот канал содержит все сообщения, которые не удалось доставить акторам-получателям, и его можно использовать для отладки системы и обработки ситуаций потери сообщений.

В последнем разделе мы поближе рассмотрели гарантии доставки в Акка и увидели, что есть различия между отправкой сообщений локальным и удаленным акторам. Повысить надежность доставки удаленным акторам можно с помощью `ReliableProxy`. Но будьте осторожны: это однонаправленный канал. Для отправки ответов отправителю `ReliableProxy` не используется.

В этой главе вы увидели, как можно посылать сообщения между акторами. Когда вы будете создавать свое приложение, вам может понадобиться, чтобы акторы имели свое состояние. Акторы часто используются для реализации конечных автоматов (машин состояний), например с использованием механизма `become/unbecome`, представленного в главе 9. В следующей главе вы увидите, как более формально реализовать конечный автомат с помощью акторов. Также мы посмотрим, как можно передавать состояния с помощью другого инструмента: агентов.

# Глава 11

## Конечные автоматы и агенты

В этой главе:

- реализация конечных автоматов;
- использование таймеров в конечных автоматах;
- организация совместного состояния с помощью агентов.

В предыдущих главах было представлено множество причин использования компонентов без состояния, чтобы избежать разного рода проблем, таких как необходимость восстановления состояния после ошибки. Но в большинстве случаев в системе есть компоненты, которым состояние необходимо для функционирования. Вы уже видели два возможных способа хранения состояния актора. Первый – на основе переменных класса – был показан в примере агрегатора (раздел 8.4.2). Это самый простой способ. Второе решение – на основе методов `become/unbecome` – было использовано в реализации маршрутизатора, зависящего от состояния (раздел 9.3.2). Эти два механизма считаются самыми основными способами реализации состояния. Но в некоторых ситуациях эти решения оказываются неэффективными.

В этой главе мы покажем два других решения для поддержки состояния. Начнем с разработки динамического поведения, зависящего от состояния актора, используя модель конечного автомата. Мы определим примерную модель, которую реализуем во втором разделе, где покажем, какие возможности имеются в Akka, упрощающие реализацию конечного автомата. В последнем разделе мы посмотрим, как совместно использовать общее состояние в разных потоках с использованием агентов Akka. Агенты избавляют от необходимости использовать механизмы блокировок, потому что состояние агентов может изменяться только асинхронно, с помощью событий; но читать состояние можно синхронно, без какого-либо ущерба для производительности.

## 11.1. Использование конечного автомата

*Конечный автомат*, который также иногда называют *машиной состояний*, – распространенный прием моделирования в программировании. Конечные автоматы можно использовать для моделирования самых разных проблем; типичными областями их применения являются протоколы обмена данными, парсинг синтаксиса языков и даже анализ бизнес-правил. Они способствуют изоляции состояния; вы увидите, что операции перевода наших акторов из одного состояния в другое фактически являются атомарными. Акторы получают сообщения по очереди, поэтому не нуждаются ни в каких блокировках. Для тех, кто не сталкивался с конечными автоматами раньше, мы начнем с краткого введения. Затем перейдем к примеру конечного автомата, который реализуем в следующем разделе средствами Akka.

### 11.1.1. Краткое введение в конечные автоматы

Простейшим примером конечного автомата является устройство, которое в течение работы оказывается в нескольких разных состояниях, переходя от одного к другому по определенным событиям. В качестве классического примера конечного автомата часто приводят стиральную машину: сначала она выполняет подготовительный этап, а затем переходит через последовательность определенных состояний (заполнение барабана водой, перемешивание, слив, отжим). Все переходы в стиральной машине инициируются программой, которая ждет определенное время в каждом состоянии, исходя из требований пользователя (слабое/сильное загрязнение, предварительное замачивание и т. д.). В каждый конкретный момент времени машина пребывает только в одном состоянии. Аналогичным примером из области бизнеса может служить процесс оформления заказа и покупки, описанный выше: существует четко определенный протокол, определяющий порядок взаимодействий двух сторон в обмене товарами и услугами. На примере бизнеса можно видеть, что на каждом этапе, на каждой стадии имеется некоторое представление состояния (заказ на покупку, смета или запрос на смету). Моделирующее программное обеспечение позволяет иметь дело с состояниями атомарным, изолированным способом, который является базовым принципом модели акторов.

Конечный автомат также называют *конечным*, потому что он может находиться в конечном количестве состояний. Переход из одного состояния в другое инициируется событием или условием. Изменение состояния называют *переходом*. Всякий конечный автомат определяется количеством состояний и набором событий, инициирующих переходы во все разные состояния. Есть много разных способов описать конечный автомат, но чаще их описывают с применением диаграмм. На рис. 11.1 изображена

простая диаграмма, иллюстрирующая описание нашего конечного автомата. Однако существует много других способов оформления таких диаграмм.

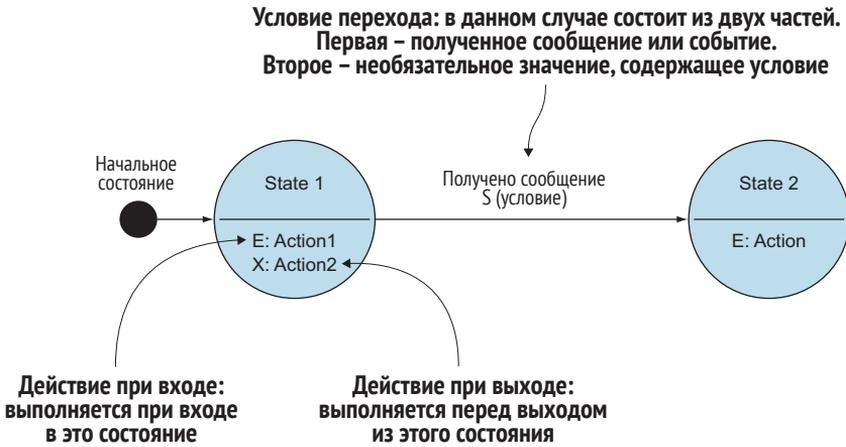


Рис. 11.1. Пример диаграммы конечного автомата

В этом примере демонстрируется конечный автомат с двумя состояниями, State1 и State2. Сразу после создания конечного автомата он переходит из начального состояния, изображенного на диаграмме большой черной точкой, в состояние State1. Состояние State1 предусматривает два разных действия: *действие при входе* и *действие при выходе*. (Действие при выходе мы не будем использовать в этой главе и показываем его здесь, только чтобы вы понимали, как работает модель.) Как следует из названия, первое действие выполняется, когда автомат входит в состояние State1, а второе – перед переходом из состояния State1 в любое другое состояние. В этом примере у нас только два состояния, поэтому данное действие выполняется, только когда происходит переход в состояние State2. В следующих примерах мы будем использовать только действия при входе, потому что это очень простой конечный автомат. Действия при выходе могут освобождать какие-нибудь ресурсы или восстанавливать некоторое состояние, поэтому считается, что они не являются воплощением логики конечного автомата. Их можно сравнить с блоком `finally` в инструкции `try/catch`, который всегда должен выполняться при выходе из блока `try`.

Изменение состояния, *переход*, может произойти только по событию. На диаграмме этот переход показан стрелкой между состояниями State1 и State2. Стрелка показывает событие и необязательное условие (например, мы могли бы выполнить переход к циклу отжима, только если в барабане нет воды). Роль событий в Акка играют сообщения.

Вот и все введение; теперь давайте посмотрим, как можно использовать конечные автоматы для решения реальных задач.

### 11.1.2. Создание модели конечного автомата

В качестве примера для демонстрации создания конечного автомата средствами Акка возьмем систему учета для книжного магазина. Служба учета принимает запрос с названием конкретной книги и возвращает ответ. Если книга имеется в магазине, система оформления заказа получит ответ, что экземпляр зарезервирован. Но может случиться так, что в данный момент в магазине нет ни одного экземпляра запрошенной книги и необходимо обратиться к издателю за получением недостающих экземпляров. Соответствующие сообщения показаны на рис. 11.2.

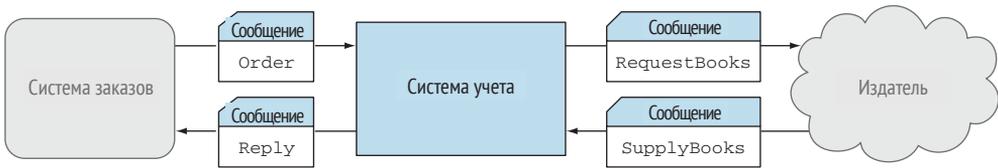


Рис. 11.2. Пример службы учета

Для простоты будем предполагать, что в нашем магазине продаются только экземпляры одной книги и заказывать книги можно только по одной. Получив заказ, служба учета должна проверить наличие экземпляров книги. Если экземпляры имеются, созданный ответ должен указывать, что книга зарезервирована. Но когда в магазине не осталось ни одного свободного экземпляра, обработка заказа должна быть приостановлена, должен быть послан запрос издателю на приобретение дополнительных экземпляров. Издатель может ответить согласием выслать запрошенное число экземпляров или сообщить, что тираж распродан и таких книг больше нет. В процессе ожидания ответа от издателя допускается принимать другие заказы.

Для описания ситуации можно использовать конечный автомат, потому что система учета может находиться в разных состояниях и ожидать разных сообщений перед переходом к следующему шагу. На рис. 11.3 показана наша задача в виде конечного автомата.

Единственное, что не изображено на диаграмме, – это возможность продолжать принимать запросы `BookRequest`, которые добавляются в список `PendingRequest`, пока система находится в состоянии ожидания. Это важно, потому что показывает важность обеспечения параллельной обработки. Обратите внимание, что когда происходит возврат в состояние ожидания, вполне возможно, что в списке `PendingRequest` могут иметься необработанные запросы.

Действие при входе проверяет это, и если действительно существуют сообщения, ожидающие обработки, инициирует один или несколько переходов, в зависимости от числа книг в магазине. Когда выясняется, что все книги распроданы, происходит переход в состояние `ProcessSoldOut`. В этом

состоянии конечный автомат посылает сообщение с признаком ошибки и инициирует переход в состояние `SoldOut`. Конечные автоматы дают нам возможность описать сложное поведение простым и кратким способом.

Теперь, когда мы описали решение с применением конечного автомата, давайте посмотрим, как Akka может помочь реализовать его.

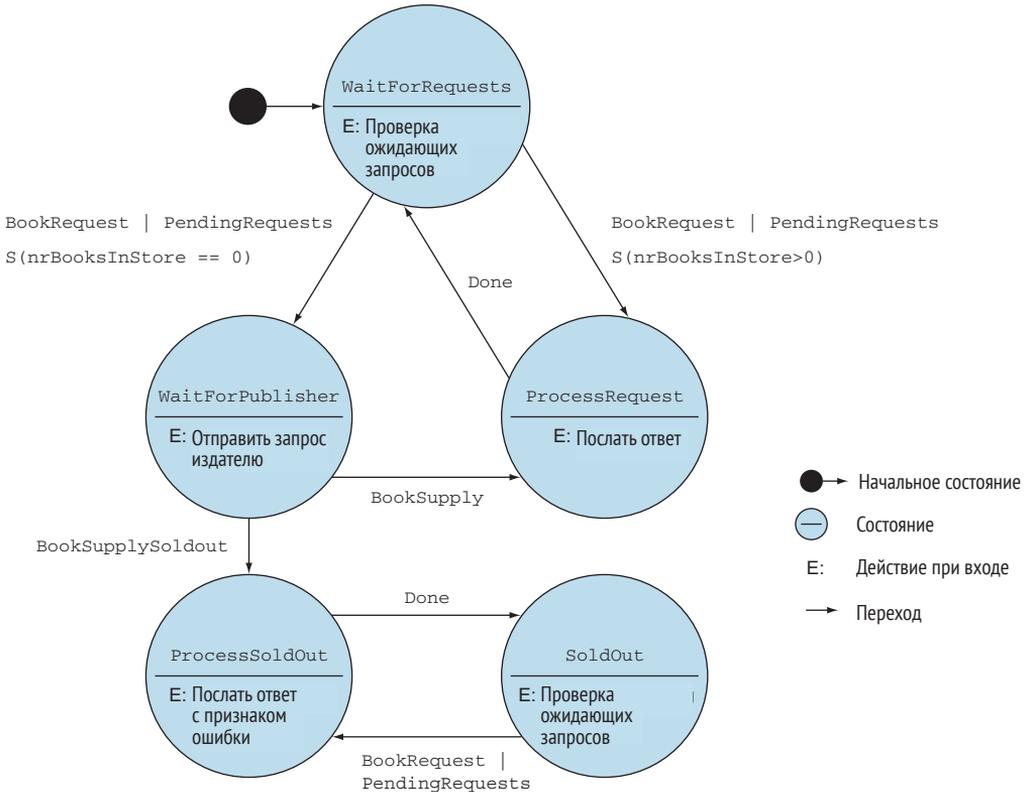


Рис. 11.3. Конечный автомат системы учета

## 11.2. Реализация модели конечного автомата

В разделе 9.3.2 вы видели, как действует механизм `become/unbecome`. Он также может помочь нам реализовать конечный автомат: мы можем отобразить поведения в состояния. Механизм `become/unbecome` с успехом можно использовать для реализации маленьких и простых моделей конечных автоматов. Но когда имеется несколько переходов в некоторое состояние, действие при входе придется реализовать разными версиями методов `become/receive`, которые будет очень сложно поддерживать в случае использования больших конечных автоматов. Поэтому для этих целей в Akka имеется трейт `FSM`, который можно использовать для реализации модели

конечного автомата. В результате получается более простой и ясный код. В этом разделе мы объясним, как пользоваться этим трейтом. Для начала реализуем переходы нашего конечного автомата учета, а потом реализуем действия при входе, завершив реализацию конечного автомата. На данный момент мы реализуем спроектированный конечный автомат, но внутри трейта FSM в Akka имеется возможность использования таймеров, которая обсуждается дальше. В заключение мы познакомимся с возможностью выполнения заключительных операций с помощью трейта FSM.

### 11.2.1. Реализация переходов

Начнем реализацию конечного автомата с создания актора с применением трейта FSM. (Трейт FSM можно подмешивать только в акторы.) Такой подход выбран в Akka вместо расширения класса Actor, чтобы сделать очевидным, что фактически создается актор. Реализуя конечный автомат, мы должны выполнить несколько шагов, прежде чем получим законченный актор конечного автомата. Два самых крупных шага – определение состояний и переходов. Итак, начнем создание нашего конечного автомата с помешивания трейта FSM:

```
import akka.actor.{Actor, FSM}

class Inventory() extends Actor with FSM[State, StateData] {
  ...
}
```

Трейт FSM принимает параметры двух типов:

- State – супертип для всех имен состояний;
- StateData – тип данных состояний, отслеживаемых конечным автоматом.

Супертипом обычно является запечатанный трейт с расширяющими его case-объектами, потому что нет смысла создавать дополнительные состояния без создания переходов к ним. Начнем с определения наших состояний.

#### Определение состояний

Процесс определения состояний начинается с объявления трейта (с подходящим именем State) и добавления case-объектов, по одному для каждого конкретного состояния, в котором может находиться наш конечный автомат (обратите внимание: это помогает сделать код самодокументированным):

```
sealed trait State
case object WaitForRequests extends State
```

```
case object ProcessRequest extends State
case object WaitForPublisher extends State
case object SoldOut extends State
case object ProcessSoldOut extends State
```

Объявленные здесь состояния соответствуют изображенным на рис. 11.3. Далее мы должны создать данные состояний:

```
case class StateData(nrBooksInStore:Int,
                    pendingRequests:Seq[BookRequest])
```

Эти данные будут использоваться, когда потребуются проверить условие, прежде чем выполнить фактический переход, поэтому они включают все ожидающие запросы и количество книг на складе. В нашем случае имеется один класс, содержащий `StateData` (будет использоваться во всех состояниях), но вообще это не является обязательным требованием. Точно так же мы могли бы использовать трейт `StateData` и создать разные классы `StateData`, наследующие базовый трейт состояния. Прежде всего, реализуя трейт `FSM`, нужно определить начальное состояние и начальные данные `StateData`. Делается это с помощью метода `startWith`:

```
class Inventory() extends Actor with FSM[State, StateData] {
  startWith(WaitForRequests, new StateData(0,Seq()))
  ...
}
```

Здесь мы указали, что сразу после создания наш конечный автомат должен оказаться в состоянии `WaitForRequests` с пустыми данными `StateData`. Далее мы должны реализовать все возможные переходы между состояниями. Переходы выполняются только по событиям. В трейте `FSM` для каждого состояния нужно определить, какие события в нем являются ожидаемыми. Определяя следующие состояния, мы задаем переходы. Начнем с событий для состояния `WaitForRequests`. В следующем разделе мы определим фактические переходы и посмотрим, как перейти от плана к действующему коду.

### Определение переходов

Взгляните на рис. 11.4, где изображено начальное состояние и два возможных перехода из него. Как видите, в этом состоянии предполагается возможность появления двух событий: получение сообщения `BookRequest` или `PendingRequests`. В зависимости от значения `nrBooksInStore` состояние может измениться на `ProcessRequest` или `WaitForPublisher`, то есть выполнится один из переходов. Мы должны реализовать эти переходы в нашем конечном автомате. Сделаем это с помощью объявления `when`.

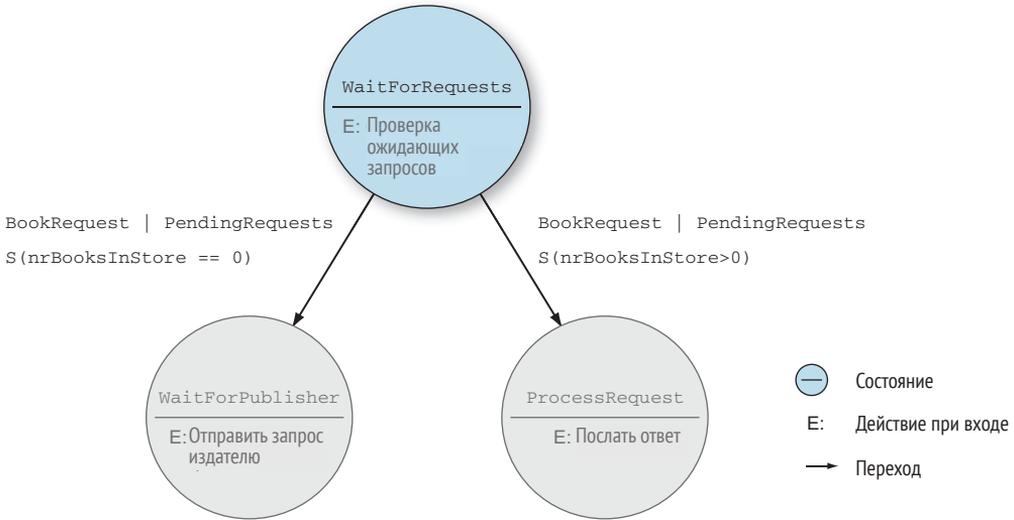


Рис. 11.4. Переходы из состояния WaitForRequests

**Листинг 11.1.** Определение переходов в трейте FSM

```
class Inventory() extends Actor with FSM[State, StateData] {
  startWith(WaitForRequests, new StateData(0, Seq()))

  when(WaitForRequests) {
    case Event(request:BookRequest, data:StateData) => {
      ...
    }
    case Event(PendingRequests, data:StateData) => {
      ...
    }
  }
  ...
}
```

← Объявление переходов для состояния WaitForRequests  
 ← Объявление возможного события Event при получении сообщения BookRequest  
 ← Объявление возможного события Event при получении сообщения PendingRequests

Мы начали с объявления when для состояния WaitForRequests. Это частично вычисленная функция, предназначенная для обработки всех возможных событий в указанном состоянии. В нашем случае возможны два разных события. Когда конечный автомат находится в состоянии WaitForRequests, допускается получение новых сообщений BookRequest и PendingRequests. Далее мы должны реализовать переходы.

Мы можем или остаться в прежнем состоянии, или перейти в другое состояние. Эти действия можно обозначить следующими двумя методами:

```
goto(WaitForPublisher)    ← Объявляет, что следующее состояние - WaitForPublisher
stay                    ← Объявляет, что состояние не изменяется
```

Кроме того, при переходе необходимо обновить данные состояния `StateData`. Например, когда приходит новое сообщение `BookRequest`, мы должны сохранить запрос в списке `PendingRequests`. Для этого можно использовать объявление `using`. В листинге 11.2 приводится законченное объявление перехода для состояния `WaitForRequests`.

**Листинг 11.2.** Реализация переходов для `WaitForRequests`

```
when(WaitForRequests) {
  case Event(request:BookRequest, data:StateData) => {
    val newStateData = data.copy(
      pendingRequests = data.pendingRequests :+ request)
    if (newStateData.nrBooksInStore > 0) {
      goto(ProcessRequest) using newStateData
    } else {
      goto(WaitForPublisher) using newStateData
    }
  }
  case Event(PendingRequests, data:StateData) => {
    if (data.pendingRequests.isEmpty) {
      stay
    } else if (data.nrBooksInStore > 0) {
      goto(ProcessRequest)
    } else {
      goto(WaitForPublisher)
    }
  }
}
```

Создать новое состояние, добавив  
новый запрос в конец очереди

Объявить следующее состояние  
и обновить `StateData`

Остаться в текущем состоянии, если  
в очереди нет ожидающих запросов

Выполнить переход без  
обновления данных `StateData`

В этом примере мы использовали метод `stay` без обновления `StateData`, но вообще можно оставаться в текущем состоянии и обновлять данные с помощью `using`, в точности как в объявлении `goto`. Это все, что нужно, чтобы определить переходы для первого состояния. Следующий шаг – реализация переходов для всех поддерживаемых состояний. Исследовав события ближе, можно заметить, что в большинстве состояний событие `BookRequest` должно вызывать один и тот же эффект: добавление запроса в конец очереди. Для таких событий мы можем объявить `whenUnhandled`. Эта частично вычисленная функция вызывается, когда функция состояния не обрабатывает событие. Здесь мы можем реализовать реакцию по умолчанию в ответ на полученное сообщение `BookRequest`. Те же объявления можно использовать в объявлении `when`:

**Листинг 11.3.** Реализация реакции по умолчанию с помощью `whenUnhandled`

```

whenUnhandled {
  // общий код для всех состояний
  case Event(request:BookRequest, data:StateData) => {
    stay using data.copy(
      pendingRequests = data.pendingRequests :+ request)
  }
  case Event(e, s) => {
    log.warning("received unhandled request {} in state {}/{}",
      e, stateName, s)
    stay
  }
}

```

Только обновить StateData

Записать в журнал, если событие не обработано

В этой частично вычисленной функции можно также фиксировать в журнале необработанные события, что может пригодиться для отладки реализации конечного автомата. Теперь реализуем все остальные состояния.

**Листинг 11.4.** Реализация переходов для других состояний

```

when(WaitForPublisher) {
  case Event(supply:BookSupply, data:StateData) => {
    goto(ProcessRequest) using data.copy(
      nrBooksInStore = supply.nrBooks)
  }
  case Event(BookSupplySoldOut, _) => {
    goto(ProcessSoldOut)
  }
}

when(ProcessRequest) {
  case Event(Done, data:StateData) => {
    goto(WaitForRequests) using data.copy(
      nrBooksInStore = data.nrBooksInStore - 1,
      pendingRequests = data.pendingRequests.tail)
  }
}

when(SoldOut) {
  case Event(request:BookRequest, data:StateData) => {
    goto(ProcessSoldOut) using new StateData(0,Seq(request))
  }
}

when(ProcessSoldOut) {

```

Переход для состояния WaitForPublisher

Переход для состояния ProcessRequest

Переход для состояния SoldOut

Переход для состояния ProcessSoldOut

```

case Event(Done, data:StateData) => {
  goto(SoldOut) using new StateData(0,Seq())
}
}

```

Теперь мы определили все переходы для всех возможных состояний. На этом заканчивается первый шаг в создании актора Akka, играющего роль конечного автомата. На данный момент у нас есть конечный автомат, реагирующий на события и изменяющий свое состояние, но фактическая модель – действия при входе – пока не реализована. Мы займемся этим в следующем разделе.

### 11.2.2. Реализация действий при входе в состояния

Фактическая работа в конечном автомате выполняет его действия при входе и выходе, которые мы сейчас и реализуем. Наша модель определяет несколько действий при входе. Подобно переходам, для каждого состояния также требуется реализовать действия. На рис. 11.5 снова изображено начальное состояние `WaitForRequests` и действие при входе, которое мы должны реализовать. Такая организация кода, как вы увидите далее, также способствует упрощению модульного тестирования.

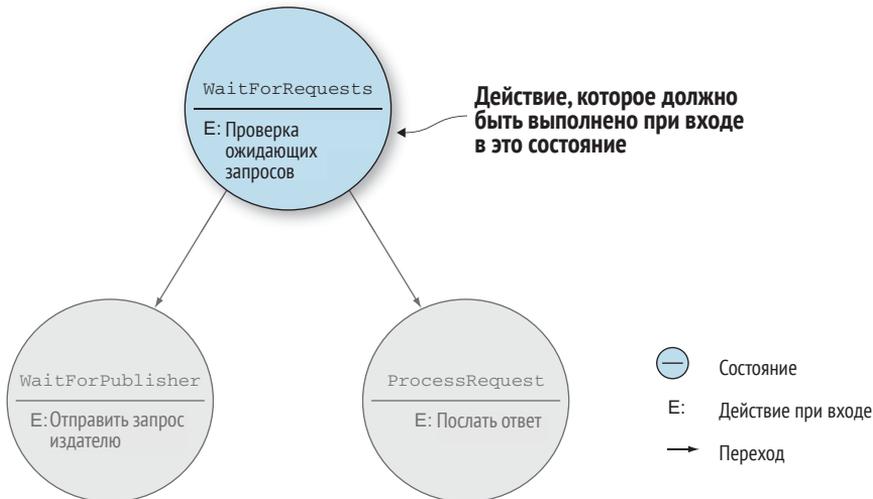


Рис. 11.5. Действие при входе для состояния `WaitForRequests`

### Действия, выполняемые при переходах

Действие при входе можно реализовать в объявлении `onTransition`. Таким способом можно объявить действия для всех возможных переходов, потому что функция обратного вызова перехода также является частично вычисленной функцией и принимает текущее и следующее состояния:

```

onTransition {
  case WaitForRequests -> WaitForPublisher => {
    ...
  }
}

```

В этом примере определяется действие, которое должно выполниться при переходе из состояния `WaitForRequests` в состояние `WaitForPublisher`. Кроме того, допускается использовать шаблонные символы. В данном примере нас не интересует предыдущее состояние, поэтому вместо него можно использовать шаблонный символ. В реализации действия вам, вероятно, понадобятся данные состояния `StateData`; при этом доступны и могут использоваться оба экземпляра данных – до и после перехода. Данные для нового состояния доступны в переменной `nextStateData`, а для старого состояния – в переменной `StateData`. В нашем примере мы используем только новые данные, потому что мы реализуем лишь действие при входе и данные всегда содержат полную информацию о состоянии. В листинге 11.5 представлена реализация всех действий при входе для нашего конечного автомата.

#### Листинг 11.5. Реализация всех действий при входе

```

class Inventory(publisher:ActorRef) extends Actor
  with FSM[State, StateData] {

  startWith(WaitForRequests, new StateData(0,Seq()))

  when...

  onTransition {
    case _ -> WaitForRequests => {
      if (!nextStateData.pendingRequests.isEmpty) {
        // перейти к следующему состоянию
        self ! PendingRequests
      }
    }
    case _ -> WaitForPublisher => {
      publisher ! PublisherRequest
    }
    case _ -> ProcessRequest => {
      val request = nextStateData.pendingRequests.head
      reserveId += 1
      request.target !
        new BookReply(request.context, Right(reserveId))
      self ! Done
    }
  }
}

```

При входе проверить очередь ожидающих событий

При входе послать запрос издателю

При входе послать ответ отправителю и сигнал, что обработка завершена

```

case _ -> ProcessSoldOut => {
  nextStateData.pendingRequests.foreach(request => {
    request.target !
    new BookReply(request.context, Left("SoldOut"))
  })
  self ! Done
}
}
}

```

← При входе послать сообщение с признаком ошибки в ответ на все запросы в PendingRequests и сигнал, что обработка завершена

При внимательном рассмотрении можно заметить, что мы не объявили действие при входе в состояние `SoldOut`. Дело в том, что для этого состояния не требуется выполнять никаких действий при входе. Теперь, когда мы закончили реализацию конечного автомата, нужно вызвать его самый важный метод `initialize`. Этот метод выполняет инициализацию и запуск конечного автомата.

#### Листинг 11.6. Инициализация конечного автомата

```

class Inventory(publisher:ActorRef) extends Actor
  with FSM[State, StateData] {

  startWith(WaitForRequests, new StateData(0,Seq()))

  when...

  onTransition...

  initialize
}

```

Конечный автомат готов; теперь нам осталось написать фиктивную реализацию, представляющую издателя, и можно опробовать его, как показано в следующем разделе.

#### Тестирование конечного автомата

В следующем примере приводится фиктивная реализация актора `Publisher`, имитирующего издателя, готового поставить predetermined количество книг. Когда книги заканчиваются, он посылает ответ `BookSupplySoldOut`.

#### Листинг 11.7. Реализация актора Publisher

```

class Publisher(totalNrBooks: Int, nrBooksPerRequest: Int)
  extends Actor {

  var nrLeft = totalNrBooks

```

```

def receive = {
  case PublisherRequest => {
    if (nrLeft == 0)
      sender() ! BookSupplySoldOut
    else {
      val supply = min(nrBooksPerRequest, nrLeft)
      nrLeft -= supply
      sender() ! new BookSupply(supply)
    }
  }
}

```

Тираж распродан

Поставить заказанное количество книг

Теперь у нас есть все, необходимое для тестирования конечного автомата. В процессе тестирования мы будем посылать сообщения и проверять соответствие результатов ожидания. Кроме того, в процессе отладки нам доступна дополнительная информация. Трейт FSM в Akka имеет одну полезную особенность: он позволяет подписаться на события изменения состояния конечного автомата. Это может пригодиться не только для реализации прикладных функций, но и для тестирования. Благодаря этой возможности можно убедиться, что состояния наступают и переходы выполняются в соответствующие моменты времени. Для подписки на событие перехода достаточно послать конечному автомату сообщение `SubscribeTransitionCallBack`. В нашем тесте мы должны организовать сбор таких событий перехода.

#### Листинг 11.8. Подписка на события перехода

```

val publisher = system.actorOf(Props(new Publisher(2,2)))
val inventory = system.actorOf(Props(new Inventory(publisher)))
val stateProbe = TestProbe()
inventory ! new SubscribeTransitionCallBack(stateProbe.ref)
stateProbe.expectMsg(new CurrentState(inventory, WaitForRequests))

```

Сначала создадим актор Publisher

Обратите внимание: при создании актора учета ему передается актор-издатель

Тестовый актор должен получить уведомление

Подписать тестовый актор на получение событий перехода

Когда посылается запрос на подписку, конечный автомат отвечает сообщением `CurrentState`. Наш конечный автомат изначально оказывается в состоянии `WaitForRequests`, в точности как ожидается. Теперь, после подписки на переходы, можно послать сообщение `BookRequest` и посмотреть, что получится:

```

inventory ! new BookRequest("context1", replyProbe.ref)
stateProbe.expectMsg(
  new Transition(inventory, WaitForRequests, WaitForPublisher))
stateProbe.expectMsg(
  new Transition(inventory, WaitForPublisher, ProcessRequest))
stateProbe.expectMsg(
  new Transition(inventory, ProcessRequest, WaitForRequests))
replyProbe.expectMsg(new BookReply("context1", Right(1)))

```

← Отправка сообщения должна вызвать изменение состояния

← Актор учета выполнит переход через три состояния для обработки запроса

← Наконец мы получили ответ

Как видите, конечный автомат переходит через несколько состояний, прежде чем прислать ответ. Сначала он получает книги от издателя. На следующем шаге он фактически обрабатывает запрос. И наконец, возвращается в состояние `WaitForRequests`. Но мы знаем, что в магазине имеется две копии, поэтому после отправки второго запроса конечный автомат переходит другую последовательность состояний:

```

inventory ! new BookRequest("context2", replyProbe.ref)
stateProbe.expectMsg(
  new Transition(inventory, WaitForRequests, ProcessRequest))
stateProbe.expectMsg(
  new Transition(inventory, ProcessRequest, WaitForRequests))
replyProbe.expectMsg(new BookReply("context2", Right(2)))

```

← На этот раз конечный автомат принимает только два состояния и затем отвечает, как ожидается

Поскольку книга имеется в наличии, состояние `WaitForPublisher` пропускается. После этого все книги оказываются распроданными, поэтому давайте посмотрим, что получится, если послать еще одно сообщение `BookRequest`.

```

inventory ! new BookRequest("context3", replyProbe.ref)
stateProbe.expectMsg(
  new Transition(inventory, WaitForRequests, WaitForPublisher))
stateProbe.expectMsg(
  new Transition(inventory, WaitForPublisher, ProcessSoldOut))
replyProbe.expectMsg(
  new BookReply("context3", Left("SoldOut")))
stateProbe.expectMsg(
  new Transition(inventory, ProcessSoldOut, SoldOut))

```

← В каждом тесте можно просто посылать одно и то же сообщение

← На этот раз результат отличается: тираж распродан

Теперь мы получили ответ `SoldOut`, как и предусматривали. Это одна из самых простых реализаций, но во многих сценариях модели конечного автомата используют таймеры для возбуждения событий и выполнения переходов. Акка также поддерживает таймеры в трейте `FSM`.

### 11.2.3. Таймеры в конечном автомате

Как отмечалось выше, конечные автоматы позволяют моделировать самые разные задачи, решение многих из которых основано на применении

таймеров, например для определения простоя соединения или истечения тайм-аута. Для демонстрации приемов использования таймеров немного изменим наш конечный автомат. Оказавшись в состоянии `WaitingForPublisher`, он не должен ждать ответа издателя вечно. На случай, если издатель потерпит неудачу во время попытки ответить, мы должны послать запрос еще раз. Необходимые для этого изменения показаны на рис. 11.6.

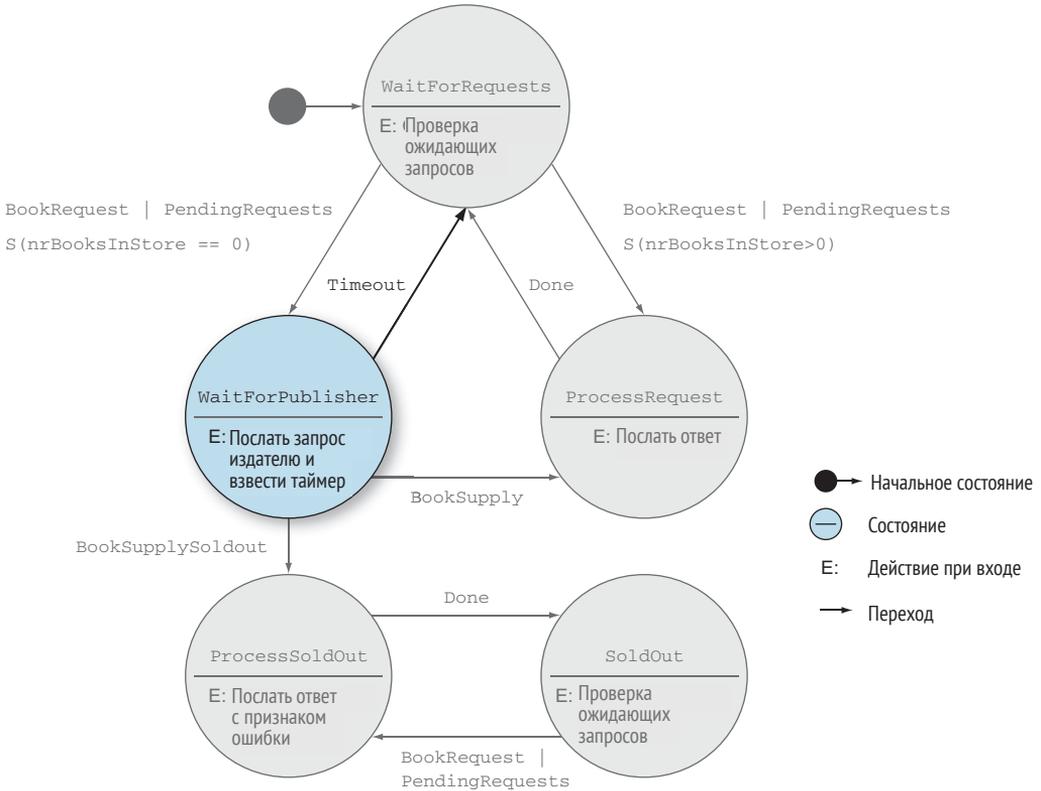


Рис. 11.6. Использование таймеров в конечном автомате

Единственное, что изменилось, – в действии при входе в состояние `WaitForPublisher` добавилась установка таймера, по истечении которого происходит переход в состояние `WaitForRequests`. Когда это происходит, проверяется наличие запросов в `PendingRequests` (и они там есть; иначе бы конечный автомат не перешел в состояние `WaitForPublisher`). А так как в очереди есть ожидающие запросы, конечный автомат вновь переходит в состояние `WaitForPublisher`, снова вызывается действие при входе, и издателю посылается новое сообщение.

Как видите, изменений немного. Во-первых, мы должны взвести таймер. Это делается установкой параметра `stateTimeout` в объявлении перехода в состояние `WaitForPublisher`. Во-вторых, нужно определить переход по истечении тайм-аута. Вот как выглядит измененное объявление `when`:

```

when(WaitForPublisher, stateTimeout = 5 seconds) {
  case Event(supply:BookSupply, data:StateData) => {
    goto(ProcessRequest) using data.copy(
      nrBooksInStore = supply.nrBooks)
  }
  case Event(BookSupplySoldOut, _) => {
    goto(ProcessSoldOut)
  }
  case Event(StateTimeout, _) => goto(WaitForRequests)
}

```

← Установка параметра stateTimeout

↑ Определение перехода по истечении тайм-аута

Это все, что мы должны были сделать, чтобы обеспечить повторную передачу запроса издателю с использованием таймера. Таймер останавливается при получении любого другого сообщения в текущем состоянии. Вы можете положиться на тот факт, что сообщение StateTimeout не будет обрабатываться после получения любого сообщения. Давайте посмотрим, как все это работает, выполнив тест в листинге 11.9.

#### Листинг 11.9. Тестирование конечного автомата с таймерами

```

val publisher = TestProbe()
val inventory = system.actorOf(
  Props(new InventoryWithTimer(publisher.ref)))
val stateProbe = TestProbe()
val replyProbe = TestProbe()

inventory ! new SubscribeTransitionCallBack(stateProbe.ref)
stateProbe.expectMsg(
  new CurrentState(inventory, WaitForRequests))

// начало тестирования
inventory ! new BookRequest("context1", replyProbe.ref)
stateProbe.expectMsg(
  new Transition(inventory, WaitForRequests, WaitForPublisher))
publisher.expectMsg(PublisherRequest)
stateProbe.expectMsg(6 seconds,
  new Transition(inventory, WaitForPublisher, WaitForRequests))
stateProbe.expectMsg(
  new Transition(inventory, WaitForRequests, WaitForPublisher))

```

← Ждать этого перехода больше 5 секунд

Как видите, когда издатель не отвечает, через 5 секунд состояние изменяется на WaitForRequests. Существует другой способ установки параметра stateTimer. Таймер можно также взвести, указав следующее состояние с помощью метода forMax, например когда нужно установить другое значение в stateTimer при переходе из иного состояния. В следующем фрагменте показано, как можно использовать метод forMax:

```
goto(WaitForPublisher) using (newData) forMax (5 seconds)
```

Вызов этого метода будет отменять настройки таймера по умолчанию, указанные в объявлении `WaitForPublisherwhen`. С помощью этого метода также можно отключить таймер, передав ему значение `Duration.Inf`.

Помимо изменения состояния с помощью таймеров, можно также организовать отправку сообщений. В этом нет никаких сложностей, поэтому вам достаточно будет краткого описания API. В конечных автоматах поддерживается три метода для работы с таймерами. Первый создает таймер:

```
setTimer(name: String,
         msg: Any,
         timeout: FiniteDuration,
         repeat: Boolean)
```

Все ссылки на таймеры осуществляются по их именам. Вызовом этого метода вы определяете имя таймера `name`, сообщение `msg` для отправки после истечения заданного времени, время срабатывания `timeout` и признак необходимости автоматического повторного запуска таймера `repeat`.

Следующий метод останавливает таймер:

```
cancelTimer(name: String)
```

Он немедленно останавливает таймер, то есть даже если таймер уже сработал и поставил сообщение в очередь, это сообщение не будет обрабатываться после вызова `cancelTimer`.

Последний метод из трех позволяет получить состояние таймера:

```
isTimerActive(name: String): Boolean
```

Он возвращает `true`, если таймер активен. Активным считается таймер, который еще не сработал или был создан со значением `true` в параметре `repeat`.

### 11.2.4. Завершение конечного автомата

Иногда бывает нужно выполнить какие-то заключительные действия перед завершением актора. Трейт `FSM` предлагает для этих целей специальный обработчик `onTermination`. Этот обработчик тоже является частично вычисленной функцией и принимает аргумент `StopEvent`:

```
StopEvent(reason: Reason, currentState: S, stateData: D)
```

Причина `Reason` может иметь три значения.

- `Normal` – нормальное завершение;
- `Shutdown` – конечный автомат завершается из-за остановки системы;
- `Failure(cause: Any)` – завершение вызвано ошибкой.

Вот как выглядит типичный обработчик завершения:

```
onTermination {
  case StopEvent(FSM.Normal, state, data)    => // ...
  case StopEvent(FSM.Shutdown, state, data) => // ...
  case StopEvent(FSM.Failure(cause), state, data) => // ...
}
```

Конечный автомат можно остановить изнутри, например вызовом метода `stop`, который принимает аргумент `Reason`, описывающий причину остановки. Когда актер останавливается с использованием ссылки `ActorRef`, указывается причина `Shutdown`.

Трейт `FSM` в `Akka` предлагает полный комплект инструментов для реализации любых конечных автоматов. Он обеспечивает четкое разделение между действиями в состояниях и переходами между состояниями. Поддержка таймеров упрощает определение состояния простоя или ошибок. И модель конечного состояния очень легко воплотить в конкретную реализацию.

Во всех примерах, приводившихся до сих пор, где использовалось состояние, это состояние всегда использовалось одним актором. Но как быть, если некоторое состояние нужно совместно использовать в нескольких акторах? В следующем разделе мы покажем, как этого добиться с помощью агентов.

## 11.3. Реализация общего состояния с помощью агентов

Лучший способ использовать состояние – применять его только в одном акторе, но это возможно не всегда. Порой нужно использовать одни и те же данные сразу в нескольких акторах, но, как уже упоминалось, для использования общего состояния требуется применять блокировки, а правильное управление блокировками – сложная задача. Для таких случаев в арсенале `Akka` имеются агенты, которые избавляют от необходимости пользоваться блокировками. Агенты защищают общее состояние и позволяют нескольким потокам читать данные и посылать запросы на их изменение. Поскольку все изменения осуществляются агентом, другие потоки не должны задумываться о блокировках. В этом разделе мы расскажем, как этим агентам удастся защитить общее состояние и как с их помощью можно организовать общий доступ к нему. Сначала мы познакомимся с самими агентами, а затем покажем примеры их использования. После этого мы познакомим вас с дополнительными возможностями агентов, помогающими следить за изменениями.

### 11.3.1. Простой доступ к общим данным с помощью агентов

Как можно получить состояние агента с помощью синхронных вызовов и изменять это состояние асинхронно? В Akka эта задача решается отправкой действий агентам, где инфраструктура поддержки обмена сообщениями исключает состояние гонки (гарантируя выполнение только одного действия в каждый конкретный момент времени в данном контексте `ExecutionContext`). В качестве примера рассмотрим хранение количества проданных экземпляров каждой книги, то есть создадим агента, хранящего это значение:

```
case class BookStatistics(val nameBook: String, nrSold: Int)
case class StateBookStatistics(val sequence: Long,
                               books: Map[String, BookStatistics])
```

`StateBookStatistics` – это объект состояния, содержащий последовательное число, которое можно использовать для проверки изменений, и фактическую информацию о продажах книг. Каждая книга представлена экземпляром `BookStatistics`, который помещается в словарь, роль ключа в котором играет название книги. На рис. 11.7 показано, как получить объект состояния из агента простым вызовом метода.

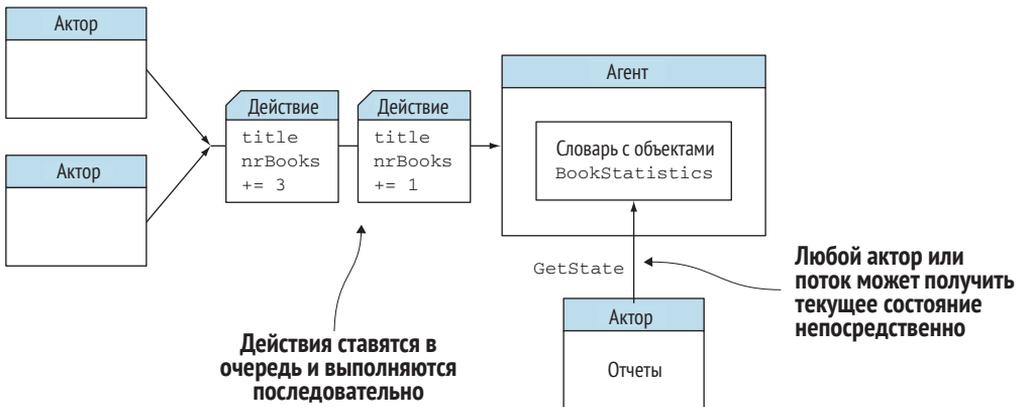


Рис. 11.7. Изменение и извлечение состояния с помощью агента

Чтобы изменить количество книг, нужно послать требуемое действие агенту. В примере первое сообщение обновления увеличивает количество проданных книг на единицу, а второе увеличивает их количество на три. Такие действия могут посылаются разными акторами и из разных потоков, но все они ставятся в очередь, подобно сообщениям, которые посылаются акторам.

И так же, как сообщения, посылаемые акторам, они выполняются последовательно, друг за другом, благодаря чему отпадает необходимость в блокировках.

Кроме того, эта работа выполняется в строгом соответствии с одним правилом: все изменения состояния должны производиться в контексте выполнения агента. Это означает, что объект состояния, хранимый объектом, должен быть неизменяемым. В нашем примере мы не можем изменить содержимое словаря за пределами агента. Чтобы внести изменения, нужно послать действие агенту, который изменит фактическое состояние. Давайте посмотрим, как это реализуется в коде.

Начнем с создания агента. Создавая агента, нужно определить начальное состояние; в данном случае это будет пустой экземпляр `StateBookStatistics`:

```
import scala.concurrent.ExecutionContext.Implicits.global
import akka.agent.Agent
```

```
val stateAgent = new Agent(new StateBookStatistics(0,Map()))
```

Также необходимо определить неявный контекст выполнения `ExecutionContext`, который будет использоваться агентом. Здесь мы используем глобальный контекст `ExecutionContext`, импортировав `scala.concurrent.ExecutionContext.Implicits.global`. С этого момента состояние находится под защитой агента. Как уже упоминалось выше, состояние агента легко получить с помощью синхронных вызовов. Сделать это можно двумя способами. Первый – прямой вызов агента:

```
val currentBookStatistics = stateAgent()
```

И второй – вызов его метода `get`, который действует точно так же:

```
val currentBookStatistics = stateAgent.get
```

В обоих случаях возвращается текущее состояние `BookStatistics`. Пока ничего необычного. Но обновить `BookStatistics` можно только асинхронно, пошлав действие агенту. Чтобы изменить состояние, нужно вызвать метод `send` агента, например вот как можно послать новое состояние:

```
val newState = StateBookStatistics(1, Map(book -> bookStat ))
stateAgent send newState
```

Но будьте осторожны, посылая обновленное состояние целиком; такой подход можно считать правильным, только если новое состояние никак не зависит от предыдущего. В нашем случае такая зависимость имеется, потому что потоки могут увеличивать количество книг или даже добавлять другие книги перед нами. Поэтому мы не должны использовать только что показанный способ. Чтобы гарантировать правильное изменение состояния, мы должны вызвать функцию внутри агента:

```

val book = "Akka in Action"
val nrSold = 1

stateAgent send( oldState => {
  val bookStat = oldState.books.get(book) match {
    case Some(bookState) =>
      bookState.copy(nrSold = bookState.nrSold + nrSold)
    case None => new BookStatistics(book, nrSold)
  }
  oldState.copy(oldState.sequence+1,
                oldState.books + (book -> bookStat))
})

```

Здесь мы использовали тот же метод `send`, но послали не новое состояние, а функцию. Эта функция преобразует прежнее состояние в новое – она обновляет атрибут `nrSold`, прибавляя заданное число, а если объект `BookStatistics` для заданной книги отсутствует в словаре, она создаст его. Последний шаг – обновление словаря.

Так как все действия выполняются по очереди, можно не беспокоиться, что во время выполнения этой функции состояние изменится из какого-то другого потока, а значит, нет нужды применять блокировки.

Теперь вы знаете, как получить текущее состояние и как изменить его, – это основные функции агента. Но, так как изменения производятся асинхронно, иногда возникает необходимость дождаться, пока они завершатся. Об этом мы поговорим в следующем разделе.

### 11.3.2. Ожидание изменения состояния

Иногда требуется дождаться, пока выполнится изменение общего состояния, чтобы тут же использовать его. Например, представим, что нам нужно узнать, какая из книг является самой популярной, чтобы в тот момент, когда она станет таковой, мы могли известить авторов. Для этого мы должны дождаться, когда наше изменение будет обработано, чтобы потом проверить – не стала ли данная книга самой популярной. Для этой цели агенты предоставляют метод `alter`. Он действует в точности как метод `send`, но возвращает объект `Future`, который можно использовать для ожидания обновления состояния.

```

implicit val timeout = Timeout(1000)
val future = stateAgent alter( oldState => {
  val bookStat = oldState.books.get(book) match {
    case Some(bookState) =>
      bookState.copy(nrSold = bookState.nrSold + nrSold)
    case None => new BookStatistics(book, nrSold)
  }
  oldState.copy(oldState.sequence+1,
                oldState.books + (book -> bookStat))
})

```

← Для ожидания нужно установить тайм-аут  
 ← Агент вернет объект Future  
 ← Здесь мы изменяем значение

```

        oldState.books + (book -> bookStat))
    })
    val newState = Await.result(future, 1 second)

```

← Новое состояние будет получено  
здесь, когда оно станет доступно

В этом примере мы выполнили изменение состояния с помощью функции, но, так же как при использовании метода `send`, мы могли бы использовать изменившееся состояние в методе `alter`. Как видите, измененное состояние возвращается внутри объекта `Future`. Но завершение `Future` не означает, что это было последнее изменение. Может так получиться, что к этому моменту своей очереди ждут другие изменения. Мы знаем лишь, что наше изменение было обработано и возвращен его результат, но одновременно с нашим изменением могло быть выполнено множество других изменений. Как быть, если нам потребуется дождаться применения всех ожидающих изменений? Для этого агент может вернуть другой объект `Future`. Он завершится, когда выполнятся все запланированные изменения.

```

val future = stateAgent.future
val newState = Await.result(future, 1 second)

```

Так можно гарантировать, что текущее состояние действительно является последним на данный момент. Имейте в виду, что при создании новых агентов с помощью `map` или `flatMap` исходные агенты остаются нетронутыми. По этой причине их часто называют *постоянными*. Вот пример создания нового агента с помощью `map`:

```

import scala.concurrent.ExecutionContext.Implicits.global
val agent1 = Agent(3)

val agent2 = agent1 map (_ + 1)

```

В данном случае `agent2` – это вновь созданный агент, содержащий значение 4, а `agent1` остается тем же, каким был прежде (он все еще хранит значение 3).

В этой главе мы показали вам, что когда возникает необходимость в использовании общего состояния, для управления этим состоянием можно использовать агента. Целостность состояния при этом гарантируется требованием выполнять изменения только в контексте агента, посылая действия агенту, подобно сообщениям.

## 11.4. В заключение

Очевидно, что создание приложений, не хранящих никакого состояния, – недостижимая цель. В этой главе вы увидели несколько подходов к управлению состоянием, поддерживаемых в Akka. Вот ключевые идеи:

- конечные автоматы, которые могут показаться чем-то особенным и крайне сложным, относительно легко реализуются в Акка, и код получается ясным и легко сопровождаемым. Реализация в виде трейта позволяет получить код, в котором реализация действий отделена от определений переходов;
- агенты представляют еще одно средство управления состоянием, которое особенно удобно, когда необходимо организовать доступ к общему состоянию из нескольких акторов;
- оба приема – конечные автоматы и агенты – позволяют использовать некоторое общее состояние без необходимости прибегать к услугам блокировок;
- использование таймеров в конечных автоматах и объектов Future с агентами обеспечивает гармоничное управление изменением состояний.

В этой главе были представлены примеры реализации сложных, взаимозависимых взаимодействий по изменению общего состояния. Нам удалось достичь желаемого, сохранив дух наших принципов, основанных на использовании обмена сообщениями, благодаря использованию механизмов координации действий множества акторов с набором общих состояний.

# Глава 12

## Интеграция с другими системами

В этой главе:

- что такое конечные точки и как их использовать;
- использование Apache Camel в Akka;
- реализация HTTP-интерфейса с Akka;
- управление потребителями и производителями.

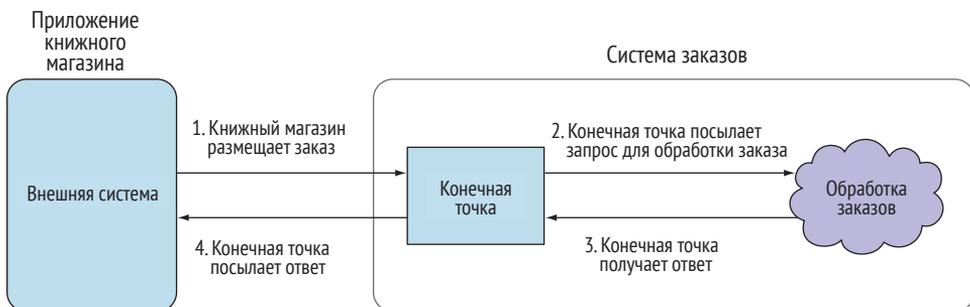
В этой главе мы рассмотрим несколько примеров использования акторов для интеграции с внешними системами. В наши дни сложность приложений непрерывно растет, и все чаще требуется организовать взаимодействие с другими информационными службами и приложениями. Ныне почти невозможно создать систему, которая не использует информацию из других систем или не предоставляет ее им. Чтобы две системы могли взаимодействовать друг с другом, они должны иметь согласованный интерфейс. Сначала мы рассмотрим несколько шаблонов интеграции. Затем опишем, как с помощью *akka-camel* (расширение Akka для поддержки Apache Camel, проекта, упрощающего интеграцию с применением множества разных транспортов) интегрировать свою систему с другими внешними системами в стиле запрос/ответ. В завершение главы вашему вниманию будет представлен пример интеграции через HTTP с помощью *akka-http* и описаны разные подходы к интеграции с Akka.

### 12.1. Конечные точки сообщений

В предыдущих главах мы показали, как конструировать системы с использованием разных шаблонов проектирования. В этом разделе мы опишем шаблоны, которые применяются, когда необходимо организовать обмен

информацией между разными системами. Мы рассмотрим систему, которой необходимы данные из приложения, осуществляющего контакты с клиентами, но вы не хотите управлять этими данными из множества приложений. Реализовать интерфейс между двумя системами не всегда просто, потому что всякий интерфейс состоит из двух частей: транспортного уровня и данных, пересылаемых через транспортный уровень, и для интеграции систем нужно реализовать обе части. Однако нам в помощь существуют шаблоны проектирования, помогающие интегрировать несколько систем друг с другом.

Например, представьте, что мы создаем систему оформления заказов для использования в книжном магазине; наша система должна обрабатывать заказы, поступающие от разных клиентов. Клиенты могут заказывать книги, посещая магазин. Магазин уже использует приложение, позволяющее заказывать и продавать книги, поэтому новая система должна поддерживать обмен данными с существующим приложением. Это возможно, только если обе системы договорятся о том, какие сообщения будут посылаться и как они будут посылаться. Поскольку часто изменить внешнее приложение не представляется возможным, вам придется написать компонент, который сможет посылать и/или принимать сообщения от существующего приложения. Этот компонент называют *конечной точкой*. Конечные точки являются частью системы и связующим звеном между внешней и нашей системой, как показано на рис. 12.1.

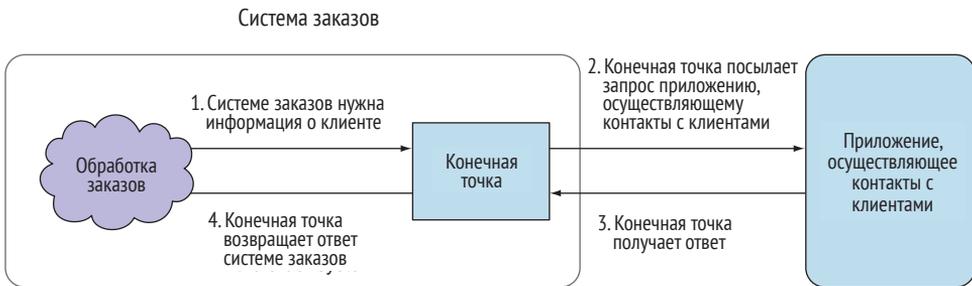


**Рис. 12.1.** Конечная точка – связующее звено между системой обработки заказов и приложением книжного магазина

Конечная точка должна так инкапсулировать интерфейс между двумя системами, чтобы самому приложению не нужно было знать детали получения запросов. Этого можно добиться путем реализации транспорта в виде сменного модуля, использования канонического формата данных и стандартизации стиля взаимодействий вида запрос/ответ. Существует множество транспортных протоколов, поддержку которых можно организовать: HTTP, TCP, очереди сообщений или простые файлы. Получив сообщение, конечная точка должна преобразовать его в формат, поддерживаемый

нашей системой обработки заказов. Благодаря такому преобразованию остальная система не будет знать, что заказ получен из внешней системы. В этом примере конечная точка принимает запросы из внешней системы и посылает обратно ответы. Она называется *конечной точкой-потребителем*, потому что потребляет запросы. Также нашей системе могут понадобиться некоторые данные из другой системы, например информация о клиенте, хранящаяся в приложении, осуществляющем контакты с клиентами.

На рис. 12.2 взаимодействие инициируется системой обработки заказов, и поскольку эта конечная точка производит сообщения, посылаемые внешней системе, ее называют *конечной точкой-производителем*. Обе конечные точки скрывают детали реализации взаимодействий от остальной системы, и когда интерфейс между системами изменится, достаточно будет изменить только конечные точки. Существует несколько шаблонов реализации таких конечных точек. Первый шаблон, который мы опишем, называется «Нормализатор» (Normalizer).



**Рис. 12.2.** Конечная точка – связующее звено между системой обработки заказов и приложением, осуществляющим контакты с клиентами

### 12.1.1. Нормализатор

Вы уже видели, что наша система заказов принимает заказы от приложения книжного магазина, но точно так же наша система могла бы принимать заказы от веб-магазина или получать их от клиента напрямую по электронной почте. Для подключения всех этих источников к единому интерфейсу на стороне приложения можно использовать шаблон «Нормализатор». Суть шаблона заключается в приведении разных внешних сообщений к общему каноническому виду. Благодаря этому можно повторно использовать всю цепочку обработки сообщений, при этом системе не придется различать типы исходных сообщений.

Создадим три разные конечные точки для приема разных сообщений, но будем преобразовывать их в сообщения общего типа, которые можно передать дальше в систему. На рис. 12.3 изображены три конечные точки, реализующие преобразование разных сообщений в общий канонический формат, понятный остальной системе.

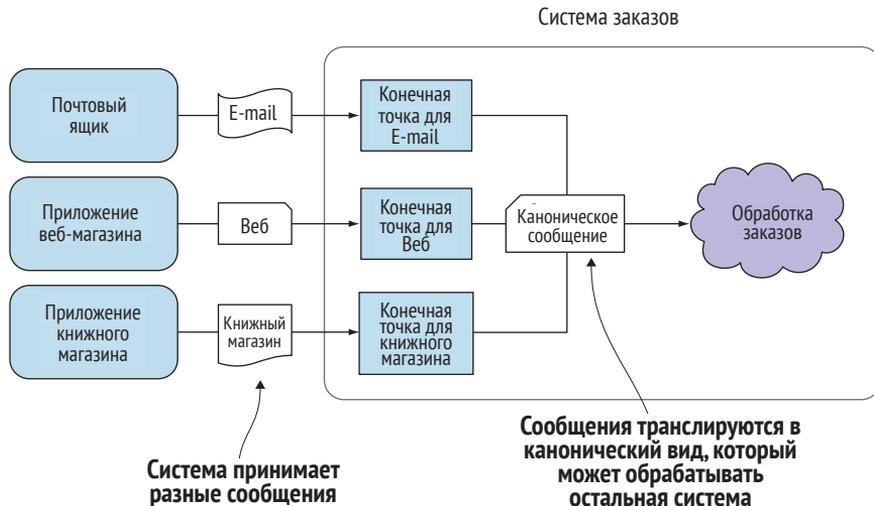


Рис. 12.3. Пример с несколькими конечными точками

Шаблон преобразования разных сообщений в общий формат называется «Нормализатор». Этот шаблон объединяет шаблоны маршрутизатора и транслятора в одной конечной точке. Представленная здесь реализация шаблона «Нормализатор» считается наиболее распространенной. Но когда осуществляется подключение нескольких разных систем с использованием разных транспортных протоколов и типов сообщений, желательно иметь возможность повторно использовать трансляторы сообщений, даже притом, что это несколько усложнит реализацию шаблона. Допустим, что имеется еще один книжный магазин, подключенный к этой системе и посылающий те же сообщения, но в качестве транспорта использующий очередь сообщений. В такой ситуации нормализатор можно разделить на три части, все они показаны на рис. 12.4. Первая – реализация протокола; вторая – маршрутизатор, выбирающий транслятор для преобразования сообщения; и третья – фактический транслятор.

Для правильного выбора транслятора нужно уметь определять типы входящих сообщений. Организация выбора в значительной степени зависит от внешних систем и типов посылаемых ими сообщений. В нашем примере предполагается поддерживать сообщения трех типов – в виде простого текста, в формате JSON и в формате XML, – которые могут поступать из любого из трех транспортных уровней – электронной почты, HTTP и очереди сообщений. В большинстве случаев предпочтительнее использовать простейшую реализацию: конечную точку и транслятор без маршрутизации, в виде единого компонента. В нашем примере можно было бы пропустить маршрутизацию для электронной почты и очереди сообщений и передавать сообщения, поступающие по этим каналам, сразу в транслятор, потому что с их помощью передаются сообщения только од-

ного типа. Это можно рассматривать как компромисс между простотой и гибкостью; использование маршрутизатора дает возможность принимать все типы сообщений по всем каналам без дополнительных усилий, но при этом число компонентов увеличится. Для различения всех типов сообщений нам достаточно одного маршрутизатора. Но трассировка сообщений усложняет решение, тогда как часто на практике такая гибкость совсем не нужна, потому что требуется организовать связь с системой одного типа (и поддерживать только один тип сообщений).

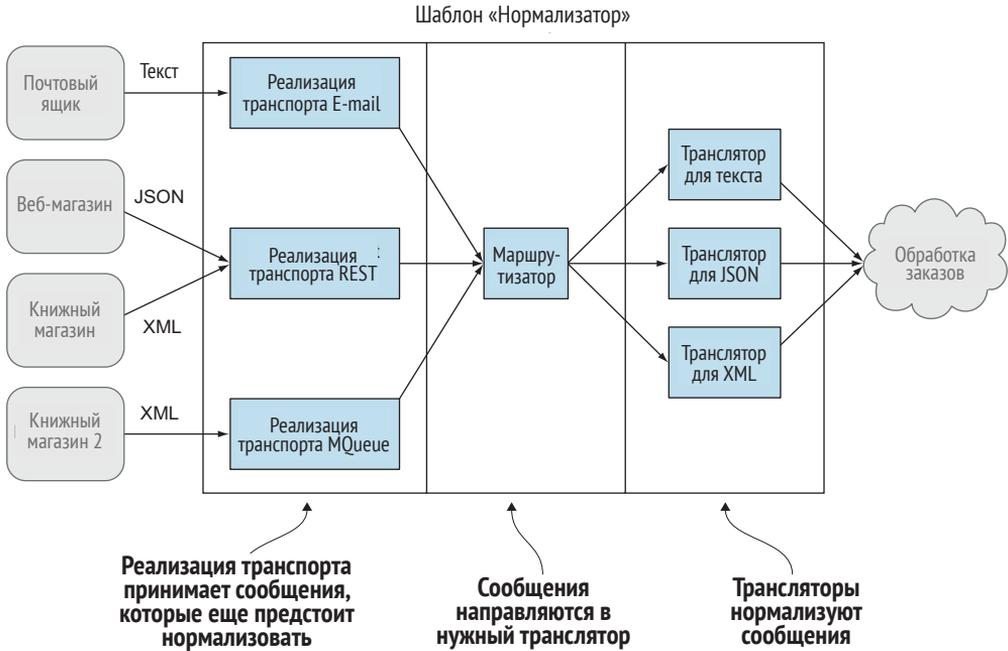


Рис. 12.4. Три части шаблона «Нормализатор»

### 12.1.2. Модель канонических данных

Шаблон «Нормализатор» хорошо работает, когда требуется связать две системы. Но с расширением требований к средствам связи вам придется увеличивать число конечных точек. Давайте вернемся к нашему примеру. У нас есть две системы, откуда исходят сообщения: система обработки заказов и система, осуществляющая контакты с клиентами. В предыдущем примере магазин был подключен только к системе обработки заказов. Но когда возникает необходимость взаимодействовать с системой, осуществляющей контакты с клиентами, реализация усложняется, как показано на рис. 12.5.

На данный момент не важно, какая система реализует конечные точки; проблема в том, что с добавлением новых систем мы должны добав-

лять новые конечные точки: одну – для приложений магазинов и две – для интеграции с имеющимися системами. С течением времени увеличение поддерживаемых систем может привести к неконтролируемому увеличению количества конечных точек.

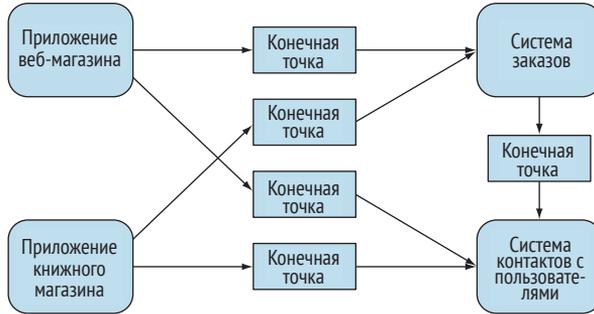


Рис. 12.5. Диаграмма соединений между системами

Для решения этой проблемы можно использовать модель канонических данных. Это шаблон связывания нескольких приложений с использованием интерфейсов, независимых от конкретных систем. В таком случае все входящие и исходящие сообщения каждой интегрируемой системы будут преобразовываться в каноническую форму для данной конечной точки.

При таком подходе каждая система имеет конечную точку, реализующую обобщенный интерфейс обобщенных сообщений. На рис. 12.6 показано, что когда приложение книжного магазина посылает сообщение системе обработки заказов, это сообщение сначала преобразуется в канонический формат, а затем пересылается с использованием общего транспортного уровня. Конечная точка на стороне системы заказов принимает каноническое сообщение и преобразует его в формат сообщений системы заказов. Может показаться, что такое многократное преобразование избыточно, но если добавить в картину несколько разнородных систем, выгоды станут очевидны; см. рис. 12.7.

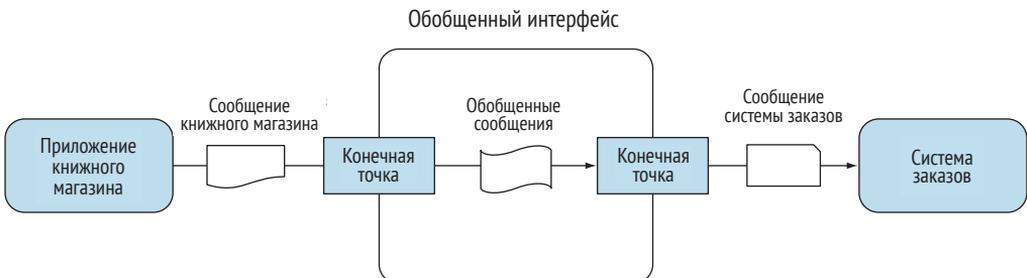
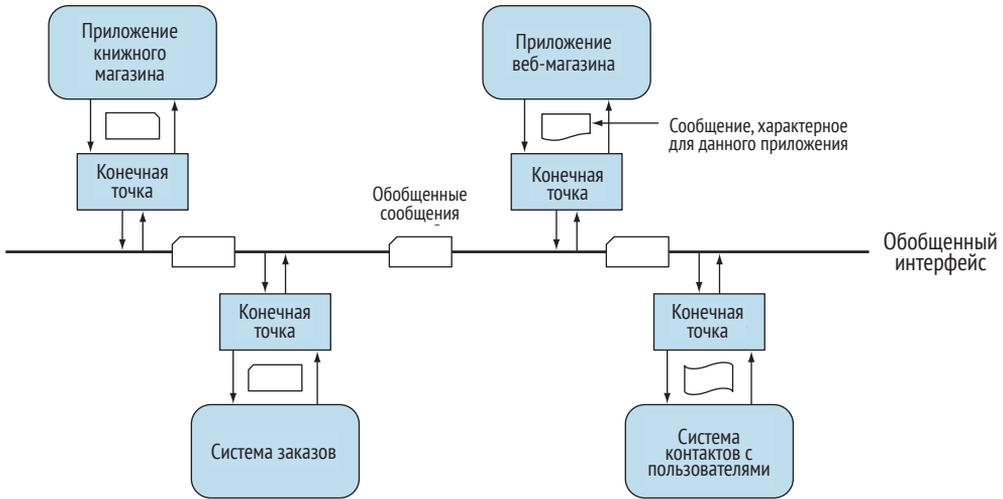


Рис. 12.6. Использование общего интерфейса между системами

Как видите, каждая система или приложение имеет по одной конечной точке. И когда веб-магазин посылает сообщение системе заказов, он ис-

пользует ту же конечную точку, что применяется для отправки сообщений системе, осуществляющей контакты с клиентами. Когда в общую картину добавится новая система, нам понадобится создать только одну конечную точку вместо четырех, как на рис. 12.5. Это поможет уменьшить общее количество конечных точек при интеграции воедино большого количества систем.



**Рис. 12.7.** Использование шаблона канонической модели данных для объединения нескольких систем

Шаблон «Нормализатор» и модель канонических данных очень удобны для интеграции систем с другими внешними системами и приложениями. Шаблон «Нормализатор» применяется для объединения нескольких похожих клиентов с другой системой. Но когда требуется интегрировать большое количество систем, предпочтительнее использовать модель канонических данных, которая похожа на шаблон «Нормализатор» тем, что так же предусматривает нормализацию сообщений. Разница лишь в том, что модель канонических данных включает дополнительный уровень косвенности между форматами данных приложений и форматами, используемыми удаленными системами, тогда как шаблон «Нормализатор» целиком реализуется в границах одного приложения. С другой стороны, при добавлении нового приложения дополнительный уровень косвенности требует создать только один транслятор для обобщенных сообщений; никаких изменений в существующие системы вносить не требуется.

Теперь, когда вы знаете, как использовать конечные точки, рассмотрим особенности их реализации. Для создания конечной точки необходимо реализовать транспортный уровень и преобразование сообщений. Реализация транспортного уровня может оказаться сложной задачей, но в большинстве случаев она не зависит от особенностей приложения. Было

бы очень хорошо, если бы имелись уже готовые реализации транспортных уровней. И такие реализации есть! Они сосредоточены в библиотеке Apache Camel. Давайте посмотрим, как с ее помощью реализовать конечную точку.

## 12.2. Реализация конечных точек с использованием Apache Camel

Цель Apache Camel – максимально упростить интеграцию. Эта библиотека позволяет реализовать стандартные шаблоны интеграции несколькими строками кода. Она решает три типичные задачи:

- конкретные реализации широко используемых шаблонов;
- обеспечивает связь с наиболее распространенными транспортом и API;
- предоставляет простые в использовании предметно-ориентированные языки для связывания реализаций шаблонов с транспортом.

Поддержка широкого спектра транспортных уровней – главная причина использования Apache Camel с Akka, потому что позволяет реализовать самые разные транспортные уровни почти без усилий. В этом разделе мы расскажем, что такое Apache Camel и как с помощью этой библиотеки посылать и принимать сообщения.

Для работы с Apache Camel из Akka используется модуль Akka Camel. Он позволяет работать со всеми транспортными протоколами и API, реализованными в Apache Camel. Вот несколько примеров поддерживаемых протоколов: HTTP, SOAP, TCP, FTP, SMTP и JMS. На момент написания этих строк библиотека поддерживала около 80 протоколов и API.

Модуль akka-camel прост в применении. Достаточно добавить akka-camel в зависимости проекта, и вы получаете доступ к классам Consumer и Producer для создания конечных точек. Эти классы скрывают за своим фасадом реализацию транспортного уровня. Единственное, что вам придется реализовать, – преобразование системных сообщений в обобщенные и обратно.

Поскольку реализации транспортных уровней полностью скрыты, выбирать протокол для использования можно прямо во время выполнения. Это еще одна сильная сторона akka-camel. Пока структура сообщений остается неизменной, в коде не придется ничего менять. Поэтому если во время тестирования внешняя система недоступна, все сообщения можно записывать в файлы, а когда такая система появится, вы легко сможете одним конфигурационным параметром заменить файлы протоколом HTTP и взаимодействовать уже с внешней системой.

Модуль akka-camel внутренне взаимодействует с классами в Apache Camel. К числу наиболее важных классов Apache Camel относятся CamelContext и ProducerTemplate. Класс CamelContext представляет единственное правило

маршрутизации Camel, а класс `ProducerTemplate` необходим для производства сообщений. За дополнительной информацией обращайтесь к документации для Apache Camel, доступной по адресу: <http://camel.apache.org>. Модуль `akka-camel` скрывает использование этих классов из Apache Camel, но иногда может возникнуть необходимость более тонкого управления отправкой и приемом сообщений. Модуль `akka-camel` создает расширение Camel для каждой системы акторов. Получить ссылку на системное расширение Camel можно с помощью объекта `CamelExtension`:

```
val camelExtension = CamelExtension(system)
```

Когда возникает потребность в конкретном классе из Apache Camel, таком как `CamelContext` или `ProducerTemplate`, можно воспользоваться этим расширением, как будет показано в следующих разделах. Начнем с реализации простого потребителя, который читает файлы, а потом заменим их использованием протоколов TCP и ActiveMQ. Этот раздел мы закончим созданием производителя, способного посылать сообщения созданному потребителю. Итак, начнем с создания потребителя с использованием `akka-camel`.

### 12.2.1. Реализация конечной точки-потребителя для приема сообщений из внешней системы

В данном примере мы реализуем систему заказов, принимающую сообщения от приложения книжного магазина. Эта система должна уметь принимать сообщения от разных книжных магазинов. Допустим, что сообщения хранятся в XML-файлах. Роль транспортного уровня в данном случае будет играть файловая система. Конечная точка в системе обработки заказов должна следить за содержимым файловой системы и при появлении новых файлов читать их содержимое в формате XML и создавать сообщения, которые система сможет обработать. Прежде чем приступить к реализации конечной точки-потребителя, нужно создать сообщения, как показано на рис. 12.8.

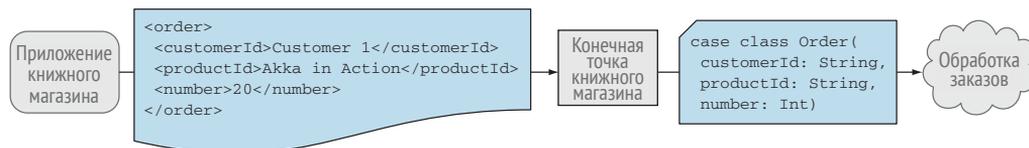


Рис. 12.8. Сообщения, принимаемые и посылаемые конечной точкой

Первое сообщение, которое мы рассмотрим, имеет формат XML, посылается приложением книжного магазина и указывает, что клиент 1 желает приобрести 20 копий книги «Akka в действии». Второе сообщение – определение класса сообщений, которые может обрабатывать система заказов.

## Реализация потребителя Camel

Теперь, когда у нас есть пара сообщений, можно приступить к реализации конечной точки-потребителя. Начнем с создания класса `Actor`, дополненного трейтом `Consumer` из библиотеки Camel:

```
class OrderConsumerXml extends akka.camel.Consumer {
  // реализация описывается далее
}
```

Следующий шаг – настройка транспортного протокола; делается это переопределением идентификатора ресурса URI конечной точки. Этот URI используется библиотекой Apache Camel для определения транспортного протокола и его свойств. В нашем случае нам нужна возможность изменять URI, поэтому добавим его в конструктор. А еще нам нужно реализовать метод `receive`, потому что этот класс все еще остается актором Akka. Реализация потребителя представлена на рис. 12.9.

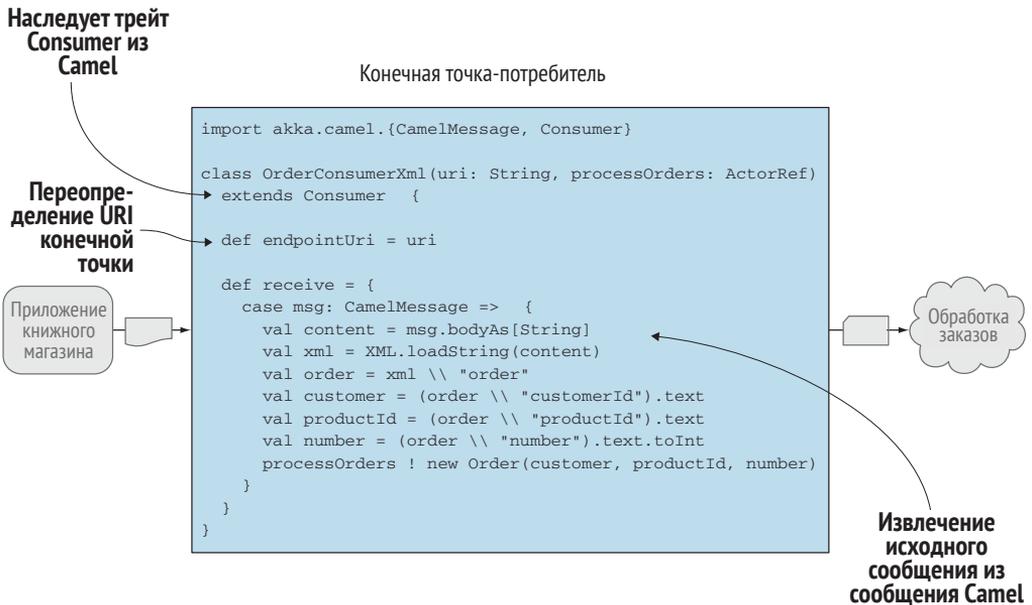


Рис. 12.9. `OrderConsumerXml` – реализация конечной точки-потребителя

Сообщения от компонента Camel поступают в виде объектов `CamelMessage` – сообщений из модуля `akka-camel`, которые не зависят от выбранного протокола, несут в своем теле фактически полученное сообщение, а также включают словарь с заголовками. Содержимое заголовков зависит от используемого протокола. Мы не будем использовать заголовки в примерах в этом разделе.

Структура тела зависит от приложения, поэтому вам придется самим реализовать преобразование данных. В этом примере мы преобразуем

тело в строку, преобразуем XML в сообщение `Order` и посылаем его следующему актору, участвующему в процессе обработки заказа.

Мы реализовали преобразование текста в формате XML в объект `Order`, но как получить файлы с сообщениями? Все необходимое сделает Apache Camel. Вам нужно только правильно установить URI. Используйте следующий URI, чтобы сообщить Apache Camel, что сообщения должны извлекаться из файлов:

```
val camelUri = "file:messages"
```

Apache Camel определяет компоненты для всех поддерживаемых видов транспорта. Идентификатор ресурса URI в примере выше начинается с имени компонента. В данном случае используется компонент `file`. Вторая часть URI зависит от выбранного компонента. Для компонента `file` во второй части указывается каталог, куда помещаются файлы с сообщениями. В данном случае предполагается, что файлы будут помещаться в каталог `messages`. Список всех возможных компонентов и их параметры можно найти по адресу: <http://camel.apache.org/components.html>.

Теперь создадим экземпляр потребителя и посмотрим, как он работает:

```
val probe = TestProbe()
val camelUri = "file:messages"
val consumer = system.actorOf(
  Props(new OrderConsumerXml(camelUri, probe.ref)))
```

Класс `CamelExtension` создает внутренние компоненты асинхронно, поэтому мы должны дождаться, пока они запустятся, прежде чем продолжить тестирование. Чтобы убедиться, что потребитель завершил процедуру запуска, используйте `CamelExtension.activationFutureFor`, как показано в листинге 12.1.

#### Листинг 12.1. Проверка запуска потребителя Camel

```
val camelExtension = CamelExtension(system)
val activated = camelExtension.activationFutureFor(
  consumer)(timeout = 10 seconds, executor = system.dispatcher)
Await.ready(activated, 5 seconds)
```

← Получить расширение `CamelExtension` для этой системы акторов

← Получить объект `Future` с результатом активации

← Ждать, пока Camel завершит запуск

Это расширение имеет метод `activationFutureFor`, возвращающий объект `Future`, который завершается, когда библиотека Camel завершает запуск. После этого можно продолжить тестирование.

Ожидание с помощью `Await.ready` вполне допустимо в контексте тестирования, но в действующем коде результат объекта `Future` лучше обрабатывать, как описывается в главе 5.

**Листинг 12.2.** Тестирование потребителя заказов

```

val msg = new Order("me", "Akka in Action", 10)
val xml = <order>
    <customerId>{ msg.customerId }</customerId>
    <productId>{ msg.productId }</productId>
    <number>{ msg.number }</number>
</order>
val msgFile = new File(dir, "msg1.xml")
FileUtils.write(msgFile, xml.toString())
probe.expectMsg(msg)
system.stop(consumer)

```

Создать сообщение в формате XML  
 Записать в файл в каталоге messages  
 Ждать, пока сообщение Order будет передано потребителю Consumer

Сообщение будет передано потребителю сразу, как только XML-файл будет помещен в каталог *messages*. Обратите внимание: вам не придется писать какой-либо код, выполняющий проверку наличия файлов в каталоге и их чтение; все это обеспечивает компонент `file` из Apache Camel.

**Изменение транспортного уровня в потребителе**

Все это хорошо, но это лишь начало пути к настоящим выгодам, которые несет Apache Camel. Представьте, что те же самые XML-сообщения могут поступать через TCP-соединение. Как реализовать этот вариант? Фактически у вас уже есть все, что нужно. Для поддержки TCP-соединения достаточно просто изменить URI и добавить несколько библиотек во время выполнения.

**Листинг 12.3.** Тестирование потребителя заказов, использующего TCP-соединение

```

val probe = TestProbe()
val camelUri =
  "mina:tcp://localhost:8888?textline=true&sync=false"
val consumer = system.actorOf(
  Props(new OrderConsumerXml(camelUri, probe.ref)))
val activated = CamelExtension(system).activationFutureFor(
  consumer)(timeout = 10 seconds, executor = system.dispatcher)
Await.ready(activated, 5 seconds)

val msg = new Order("me", "Akka in Action", 10)
val xml = <order>
    <customerId>{ msg.customerId }</customerId>
    <productId>{ msg.productId }</productId>
    <number>{ msg.number }</number>

```

Используется другой URI

```
</order>
```

```
val xmlStr = xml.toString().replace("\n", "")
val sock = new Socket("localhost", 8888)
val ouputWriter = new PrintWriter(sock.getOutputStream, true)
ouputWriter.println(xmlStr)
ouputWriter.flush()
```

← Отправка XML-сообщения  
через TCP

← Из-за использования  
параметра `textline` в конце  
сообщения добавляется  
символ перевода строки,  
поэтому его нужно удалить

```
probe.expectMsg(msg)
```

```
ouputWriter.close()
system.stop(consumer)
```

Для обслуживания TCP-соединения в этом примере использован компонент Apache Mina. Вторая часть URI теперь выглядит совершенно иначе и определяет настройки соединения. Прежде всего в ней определяется протокол (TCP), а затем определяются интерфейс и порт, используемые для приема сообщений. Далее следуют два параметра:

- `textline=true` – указывает, что ожидается получение простого текста и что каждое сообщение должно завершаться символом перевода строки;
- `sync=false` – указывает, что отправлять ответ не требуется.

Как видите, мы смогли изменить транспортный протокол без малейшего вмешательства в код потребителя. Возникает резонный вопрос: возможно ли то же самое для любого протокола? К сожалению, нет; некоторые протоколы требуют вносить изменения в код. Например, вообразите протокол, требующий подтверждения приема сообщения. Давайте посмотрим, как можно организовать такое подтверждение. Допустим, что мы должны вернуть XML-подтверждение через TCP-соединение. Для этого придется изменить потребителя, но это совсем несложно. Мы должны просто послать ответ отправителю, а об остальном позаботится Camel.

#### Листинг 12.4. Подтверждение получения заказа

```
class OrderConfirmConsumerXml(uri: String, next: ActorRef)
  extends Consumer {

  def endpointUri = uri

  def receive = {
    case msg: CamelMessage => {
      try {
        val content = msg.bodyAs[String]
        val xml = XML.loadString(content)
```

```

val order = xml \ "order"
val customer = (order \ "customerId").text
val productId = (order \ "productId").text
val number = (order \ "number").text.toInt
next ! new Order(customer, productId, number)
sender() ! "<confirm>OK</confirm>"
} catch {
    case ex: Exception =>
        sender() ! "<confirm>%s</confirm>".format(ex.getMessage)
}
}
}
}
}

```

← Отправка ответа  
отправителю

Вот и все. Теперь можно протестировать нового потребителя, изменив URI, но прежде обратите внимание на наличие кода, обрабатывающего возможные исключения. Разве в главе 4 мы не говорили, что можем позволить нашим акторам потерпеть неудачу в случае появления любых проблем? И что эти проблемы должен исправлять супервизор? Да, это так, но сейчас мы реализуем конечную точку, отделяющую синхронный интерфейс от системы передачи сообщений, которая действует асинхронно. На таких границах между синхронными и асинхронными интерфейсами правила меняются, потому что синхронный интерфейс всегда ожидает получения результата, даже в случае неудачи. Если попытаться использовать механизм супервизора для исправления проблем, вам не хватит информации об отправителе, чтобы правильно обслужить запрос. И вы не сможете использовать обработчик перезапуска, потому что супервизор может решить продолжить выполнение после исключения, вследствие чего обработчик перезапуска не будет вызван. Учитывая все это, мы должны перехватить исключение и вернуть ожидаемый ответ. А теперь протестируем нашего потребителя.

**Листинг 12.5.** Тестирование потребителя, посылающего подтверждение получения заказа через TCP-соединение

```

val probe = TestProbe()
val camelUri =
    "mina:tcp://localhost:8887?textline=true"
val consumer = system.actorOf(
    Props(new OrderConfirmConsumerXml(camelUri, probe.ref)))
val activated = CamelExtension(system).activationFutureFor(
    consumer)(timeout = 10 seconds, executor = system.dispatcher)
Await.ready(activated, 5 seconds)

val msg = new Order("me", "Akka in Action", 10)

```

← Удален параметр `sync`; он получит  
значение по умолчанию `true`

```

val xml = <order>
    <customerId>{ msg.customerId }</customerId>
    <productId>{ msg.productId }</productId>
    <number>{ msg.number }</number>
</order>

val xmlStr = xml.toString().replace("\n", "")
val sock = new Socket("localhost", 8887)
val ouputWriter = new PrintWriter(sock.getOutputStream, true)
ouputWriter.println(xmlStr)
ouputWriter.flush()
val responseReader = new BufferedReader(
    new InputStreamReader(sock.getInputStream))
val response = responseReader.readLine()
response must be("<confirm>OK</confirm>")
probe.expectMsg(msg)

responseReader.close()
ouputWriter.close()
system.stop(consumer)

```

← Прием подтверждения

← Получение сообщения Order

Внеся небольшое изменение, мы смогли организовать отправку подтверждения через TCP, что еще раз наглядно показывает неоспоримые преимущества Apache Camel.

### Использование CamelContext

Мы хотели бы показать еще один пример. Иногда компонент Camel требует более обширных настроек, чем только URI.

Например, представьте, что вы решили использовать компонент ActiveMQ. Для этого вы должны добавить компонент в CamelContext и определить брокера очереди сообщений. Для этого требуется получить ссылку на CamelContext.

#### Листинг 12.6. Настройка брокера в CamelContext

```

val camelContext = CamelExtension(system).context
camelContext.addComponent("activemq",
    ActiveMQComponent.activeMQComponent(
        "vm:(broker:(tcp://localhost:8899)?persistent=false)")

```

← Это имя компонента должно указываться в URI

Сначала нужно получить ссылку на расширение CamelExtension в текущей системе акторов, а затем добавить компонент ActiveMQ в CamelContext. В данном случае создается брокер, прослушивающий порт 8899 (и не использующий хранимых очередей).

Теперь протестируем этот транспорт. Для этого воспользуемся предыдущей реализацией потребителя, возвращающей подтверждение.

**Листинг 12.7.** Тестирование с использованием ActiveMQ

```

val camelUri = "activemq:queue:xmlTest"
val consumer = system.actorOf(
  Props(new OrderConsumerXml(camelUri, probe.ref)))

val activated = CamelExtension(system).activationFutureFor(
  consumer)(timeout = 10 seconds, executor = system.dispatcher)
Await.ready(activated, 5 seconds)

val msg = new Order("me", "Akka in Action", 10)
val xml = <order>
  <customerId>{ msg.customerId }</customerId>
  <productId>{ msg.productId }</productId>
  <number>{ msg.number }</number>
</order>

sendMQMessage(xml.toString())
probe.expectMsg(msg)

system.stop(consumer)

```

URI компонента ActiveMQ начинается с того же имени, которое использовалось при добавлении компонента ActiveMQ

Этот тест отличается от других только способом доставки сообщения.

Поскольку теперь мы запускаем брокера, его нужно остановить по завершении. Это делается с помощью `BrokerRegistry`:

```

val brokers = BrokerRegistry.getInstance().getBrokers
brokers.foreach { case (name, broker) => broker.stop() }

```

С помощью `BrokerRegistry` можно закрыть все брокеры. Обратите внимание, что вызов `getBrokers` возвращает словарь `java.util.Map`. Чтобы получить из него словарь `Scala`, можно воспользоваться библиотекой `collection.JavaConversions`.

Итак, как вы могли убедиться, реализация потребителя не составляет большого труда. А благодаря наличию большого количества компонентов в `Camel` мы можем организовать поддержку множества разных транспортных протоколов без особых усилий.

### 12.2.2. Реализация конечной точки-производителя для отправки сообщений во внешнюю систему

В предыдущем разделе мы создали конечную точку, принимающую сообщения. В этом разделе мы реализуем отправку сообщений с помощью `Camel`. Для демонстрации перейдем на другую сторону нашего примера – противоположную стороне с конечной точкой-потребителем – и реализуем конечную точку для приложения книжного магазина (см. рис. 12.10).

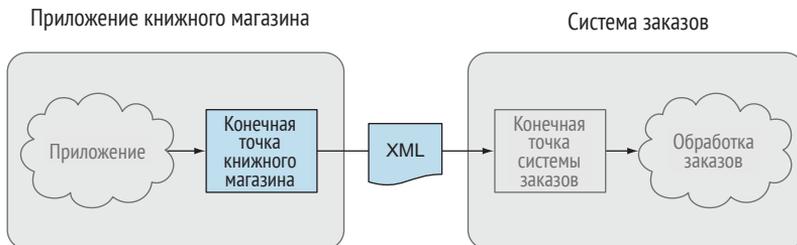


Рис. 12.10. Конечная точка-производитель, посылающая сообщения

Для реализации производителя в akka-camel имеется другой трейт: `Producer`. Производитель так же должен быть актором, но на этот раз трейт `Producer` уже предоставляет готовый метод `receive`. Чтобы получить простейшую реализацию, достаточно унаследовать трейт `Producer` и установить URI, как показано на рис. 12.11.



Рис. 12.11. Реализация простейшей конечной точки-производителя

Этот производитель посылает все сообщения, полученные от компонента Camel, определяемого идентификатором ресурса URI. То есть, создав строку XML и пошав ее производителю, вы сможете передать сообщение через TCP-соединение. В этом примере для приема сообщений мы будем использовать потребителя из предыдущего раздела. И поскольку теперь у нас имеется два актора Camel, мы не сможем запустить тестирование, пока оба актора не будут готовы. Чтобы дождаться завершения процедуры запуска обоих акторов, используем метод `Future.sequence`, который обсуждался в главе 5.

#### Листинг 12.8. Тест простого производителя

```

implicit val ExecutionContext = system.dispatcher
val probe = TestProbe()
val camelUri = "mina:tcp://localhost:8885?textline=true"
val consumer = system.actorOf(

```

```

Props(new OrderConfirmConsumerXml(camelUri, probe.ref)))

val producer = system.actorOf(
  Props(new SimpleProducer(camelUri)))
val activatedCons = CamelExtension(system).activationFutureFor(
  consumer)(timeout = 10 seconds, executor = system.dispatcher)
val activatedProd = CamelExtension(system).activationFutureFor(
  producer)(timeout = 10 seconds, executor = system.dispatcher)
val camel = Future.sequence(List(activatedCons, activatedProd))
Await.result(camel, 5 seconds)

```

Создание простого производителя

Создание объекта Future для ожидания завершения запуска обоих акторов

Каждое сообщение будет посылаться по объявленному URI. Но перед отправкой часто требуется преобразовать сообщение в другой формат. В нашей системе роль сообщений, пересылаемых между акторами, играет объект `Order`. Чтобы решить эту задачу, можно переопределить метод `transformOutgoingMessage`, который вызывается перед отправкой сообщения. В листинге 12.9 демонстрируется реализация преобразования объекта `Order` в ожидаемый формат XML.

#### Листинг 12.9. Преобразование сообщения на стороне производителя

```

class OrderProducerXml(uri: String) extends Producer {
  def endpointUri = uri
  override def oneway: Boolean = true

  override protected def transformOutgoingMessage(message: Any): Any =
    message match {
      case msg: Order => {
        val xml = <order>
          <customerId>{ msg.customerId }</customerId>
          <productId>{ msg.productId }</productId>
          <number>{ msg.number }</number>
        </order>
        xml.toString().replace("\n", "")
      }
      case other => message
    }
}

```

Указать, что ответ не ожидается

Реализация `transformOutgoingMessage`

Создание сообщения, завершающегося символом перевода строки

В методе `transformOutgoingMessage` мы создаем строку XML и, так же как в тесте потребителя, завершаем ее символом перевода строки. Поскольку используемый здесь потребитель не посылает подтверждений, мы должны сообщить фреймворку, что не нужно ожидать их, иначе фреймворк будет потреблять лишние ресурсы. Возможна даже ситуация, когда отправка большого количества сообщений израсходует все доступные потоки и

ваша система остановится. Поэтому очень важно переопределить атрибут `oneway`, если не предполагается получать ответы.

Теперь можно послать объект `Order` конечной точке-производителю, а она преобразует его в строку XML. Но как быть, если подтверждения все же будут нужны, как, например, при использовании `OrderConfirmConsumerXML`? На рис. 12.12 изображено поведение производителя по умолчанию, который посылает полученные сообщения `CamelMessage`, содержащие XML-ответ оригинальному отправителю.

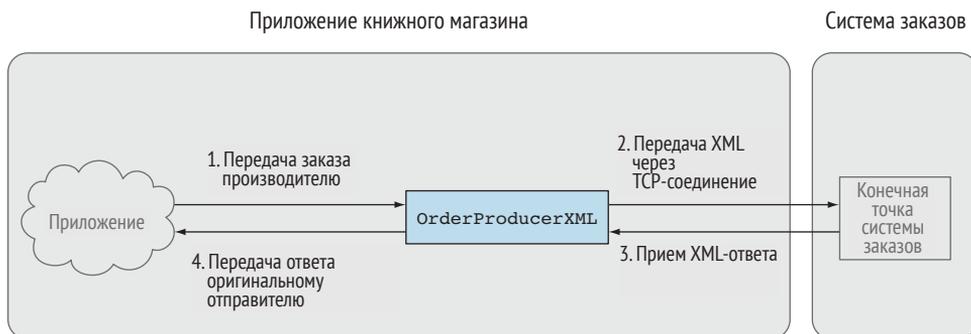


Рис. 12.12. Использование ответов в производителе Camel

Но, кроме преобразования сообщения перед отправкой, вам также нужно преобразовывать получаемые ответы – очень нежелательно, чтобы остальная часть системы знала о существовании объектов `CamelMessage`. Для решения этой задачи можно задействовать метод `transformResponse`. Он используется для преобразования принимаемых сообщений в сообщения, понятные для системы.

#### Листинг 12.10. Преобразование ответа и передача его производителю

```
class OrderConfirmProducerXml(uri: String) extends Producer {
  def endpointUri = uri
  override def oneway: Boolean = false

  override def transformOutgoingMessage(message: Any): Any =
    message match {
      case msg: Order => {
        val xml = <order>
          <customerId>{ msg.customerId }</customerId>
          <productId>{ msg.productId }</productId>
          <number>{ msg.number }</number>
        </order>
        xml.toString().replace("\n", "") + "\n"
      }
      case other => message
    }
```

```

}

override def transformResponse(message: Any): Any =
  message match {
    case msg: CamelMessage => {
      try {
        val content = msg.bodyAs[String]
        val xml = XML.loadString(content)
        val res = (xml \ "confirm").text
        res
      } catch {
        case ex: Exception =>
          "TransformException: %s".format(ex.getMessage)
      }
    }
    case other => message
  }
}

```

← Преобразует CamelMessage в строку с результатом

Метод `transformResponse` вызывается при получении ответа, перед его передачей отправителю запроса. В этом примере мы выполняем парсинг полученного текста в формате XML и выбираем значение из тега `<confirm>`. Проверим работу нашего решения с помощью теста в листинге 12.11.

#### Листинг 12.11. Тест производителя, требующего возврата подтверждений

```

implicit val ExecutionContext = system.dispatcher
val probe = TestProbe()
val camelUri = "mina:tcp://localhost:9889?textline=true"
val consumer = system.actorOf(
  Props(new OrderConfirmConsumerXml(camelUri, probe.ref)))

val producer = system.actorOf(
  Props(new OrderConfirmProducerXml(camelUri)))

val activatedCons = CamelExtension(system).activationFutureFor(
  consumer)(timeout = 10 seconds, executor = system.dispatcher)
val activatedProd = CamelExtension(system).activationFutureFor(
  producer)(timeout = 10 seconds, executor = system.dispatcher)

val camel = Future.sequence(List(activatedCons, activatedProd))
Await.result(camel, 5 seconds)
val probeSend = TestProbe()
val msg = new Order("me", "Akka in Action", 10)
probeSend.send(producer, msg)
probe.expectMsg(msg)

```

← Получение сообщения потребителем

```
probeSend.expectMsg("OK")
system.stop(producer)
system.stop(consumer)
```

← Подтверждение  
в приеме

Все это замечательно, но представьте ситуацию, когда подтверждение должно пересылаться не оригинальному отправителю, а другому актору. Такое возможно? Да, для этих целей имеется метод `routeResponse`, который отвечает за отправку полученного ответа оригинальному отправителю. Его можно переопределить и организовать передачу сообщений другому актору. Но будьте внимательны с методом `transformResponse`: если вы решите переопределить метод `routeResponse`, не забудьте вызвать `transformResponse` в своей реализации, потому что стандартная версия метода `routeResponse` вызывает `transformResponse` перед отправкой сообщения ответа оригинальному отправителю.

Как можно видеть в примерах выше, конечная точка-производитель создается так же просто, как конечная точка-потребитель. Библиотека Apache Camel предлагает массу возможностей для использования в конечных точках и поддерживает большое количество транспортных протоколов. Применение модуля `akka-camel` дает большие преимущества: поддержку большого количества протоколов без дополнительных усилий.

В следующем разделе мы рассмотрим два примера конечных точек-потребителей, включающих фактическое соединение с системой заказов для передачи ответов.

## 12.3. Реализация HTTP-интерфейса

В предыдущих разделах мы показали, как Apache Camel помогает реализовать конечные точки, способные использовать самые разные виды транспорта. Отрицательной стороной обобщенного способа взаимодействий со множеством протоколов/транспортов является сложность использования характерных особенностей конкретного протокола; например, представьте, что вам понадобилось написать HTTP-интерфейс, использующий все достоинства HTTP.

Далее рассматривается пример создания HTTP-службы на основе Akka. Для реализации минимального HTTP-интерфейса можно использовать Apache Camel, но чтобы задействовать какие-то специфические особенности HTTP, возможностей Apache Camel явно недостаточно. Развитый API для создания серверов и клиентов HTTP можно найти в модуле `akka-http`. Пример, который мы покажем, очень прост, но он наглядно иллюстрирует приемы решения типичных задач интеграции. Начнем с описания примера, а потом покажем реализацию на основе модуля `akka-http`.

### 12.3.1. Пример HTTP-интерфейса

Мы вновь реализуем пример конечных точек для системы заказов. Но на этот раз дополнительно будет реализована имитация системы обработки заказов. Благодаря этому мы сможем увидеть, как конечная точка переправляет запросы в систему и ждет ответов. В качестве транспорта конечная точка будет использовать протокол HTTP. Мы также используем архитектурный стиль REST для определения API службы HTTP. Общая структура примера показана на рис. 12.13.

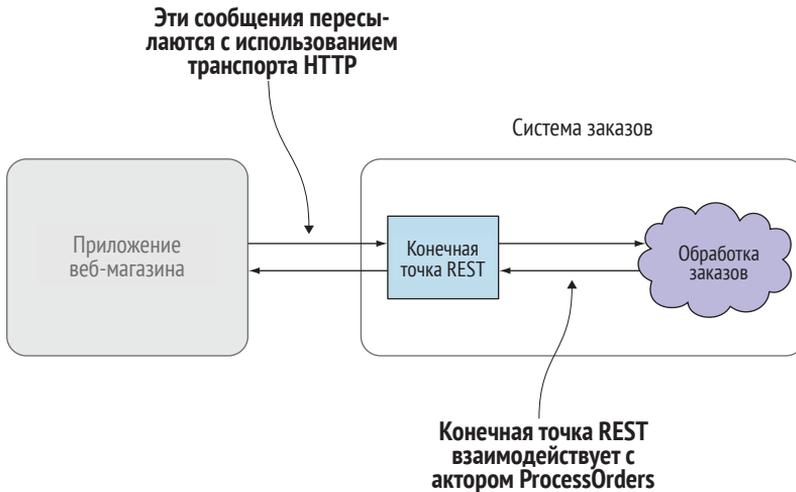
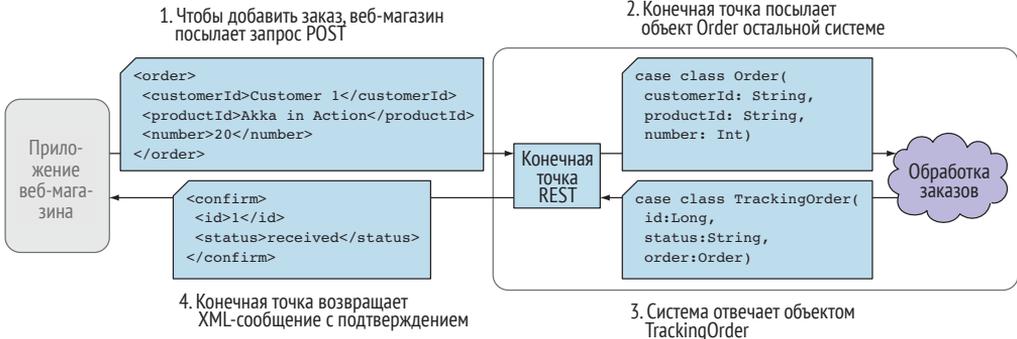


Рис. 12.13. Структура примера HTTP REST

Пример имеет два интерфейса: один между веб-магазином и конечной точкой и другой – между конечной точкой и актором ProcessOrders. Начнем с определения сообщений для обоих интерфейсов. Система заказов будет поддерживать две функции. Первая будет добавлять новый заказ, а вторая – получать состояние заказа. Интерфейс HTTP REST будет поддерживать запросы POST и GET. Запрос POST будет добавлять заказ в систему, а запрос GET – извлекать состояние заказа. Первым рассмотрим добавление заказа. На рис. 12.14 показаны сообщения и последовательность их обработки.

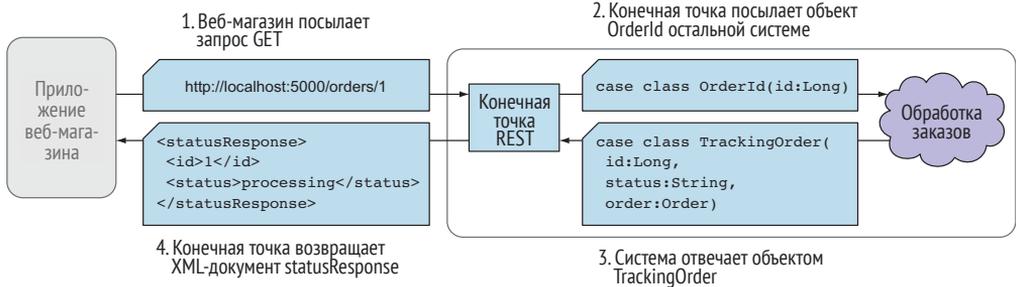
Веб-магазин посылает конечной точке запрос POST с XML-сообщением в формате, который мы уже использовали в примерах Camel в разделе 12.2.1. Конечная точка преобразует его в объект `Order` и передает системе (актору ProcessOrders). По завершении обработки система возвращает ответ – объект `TrackingOrder`, – содержащий уникальный номер заказа и его текущее состояние. Конечная точка преобразует его в XML-сообщение подтверждения и посылает обратно в веб-магазин. В этом примере новый заказ получает номер 1 и состояние `received`.



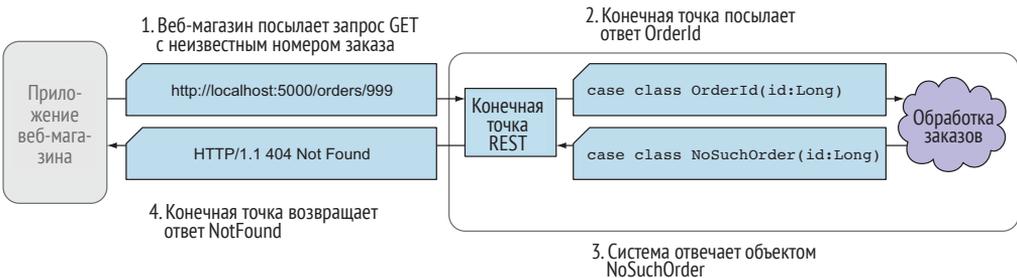
**Рис. 12.14.** Последовательность обработки сообщения при добавлении заказа

На рис. 12.15 показаны сообщения, пересылаемые для получения состояния существующего заказа.

Чтобы получить состояние заказа с номером 1, веб-магазин посылает запрос GET по адресу `/orders/1`. Конечная точка REST преобразует запрос GET в объект `OrderId` и передает его системе. В ответ система снова возвращает объект `TrackingOrder`. Конечная точка преобразует ответ в XML-документ `statusResponse`. Если заказ с запрошенным номером отсутствует, система отвечает объектом `NoSuchOrder`, как показано на рис. 12.16.



**Рис. 12.15.** Последовательность обработки сообщения при получении состояния заказа



**Рис. 12.16.** Последовательность обработки сообщения при попытке получить состояние несуществующего заказа

Конечная точка REST преобразует объект `NoSuchOrder` в HTTP-код 404 `NotFound`. Теперь, когда мы определились с сообщениями, передаваемыми между системами, можно приступить к реализации обработки заказов. На рис. 12.17 представлена реализация только что описанного интерфейса.

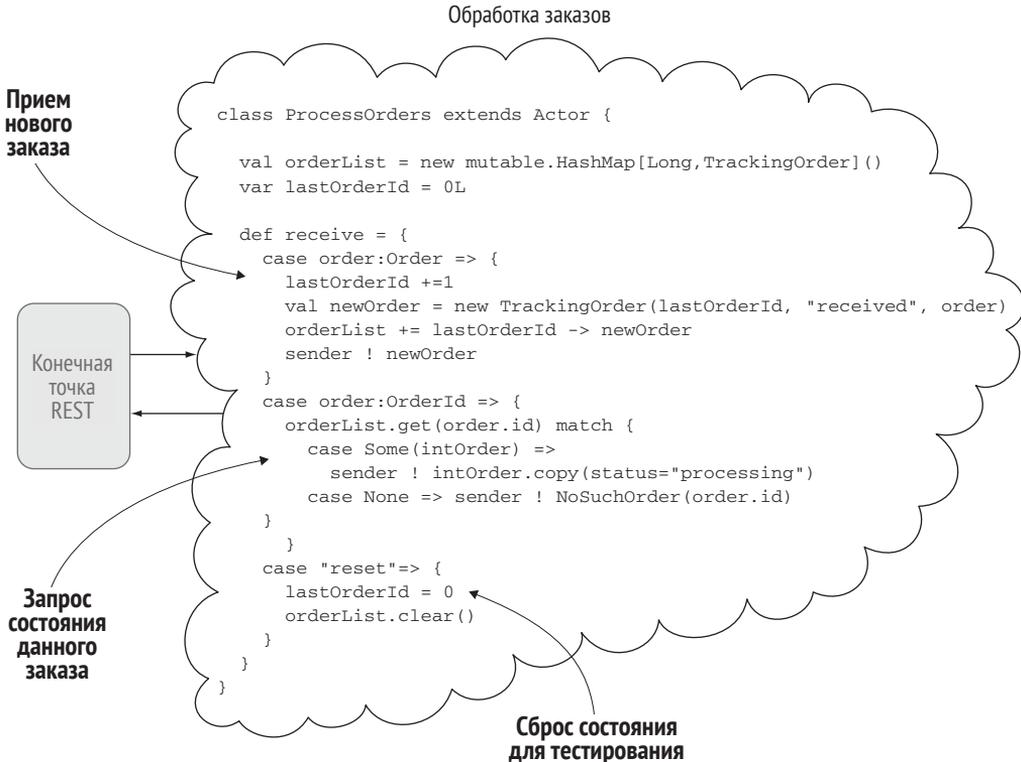


Рис. 12.17. Реализация обработки заказа

Это простая имитация законченной системы, реализующей обработку двух возможных запросов. Мы также добавили функцию `reset` для использования на этапе тестирования.

Теперь перейдем к реализации конечной точки REST с использованием `akka-http`.

### 12.3.2. Реализация конечной точки REST на основе `akka-http`

Чтобы вы смогли прочувствовать, чем модуль `akka-http` может помочь в реализации интерфейса REST, мы повторим тот же пример конечной точки, что был представлен выше, но на этот раз с использованием модуля `akka-http`. Имейте в виду, что здесь будет показана лишь малая часть возможностей `akka-http`; на самом деле они намного шире. Например, в главе 13 мы рассмотрим примеры потоковой передачи данных через HTTP.

Итак, начнем с определения HTTP-маршрутов для конечной точки REST в трейте `OrderService`. Трейт `OrderService` определяет абстрактный метод, возвращающий ссылку `ActorRef` на актор `ProcessOrders`. Отделение маршрутов от используемых акторов считается хорошей практикой, потому что позволяет тестировать маршруты без запуска акторов или внедрением `TestProbe`, например. На рис. 12.18 представлены определения класса `OrderServiceApi`, включающего необходимые объекты `ExecutionContext` и `Timeout` для отправки запросов в `ProcessOrder`, и трейта `OrderService` с маршрутами. Модуль `akka-http` имеет свой набор инструментов для тестирования, позволяющий проверить маршруты. В листинге 12.12 показано, как можно протестировать `OrderService`.

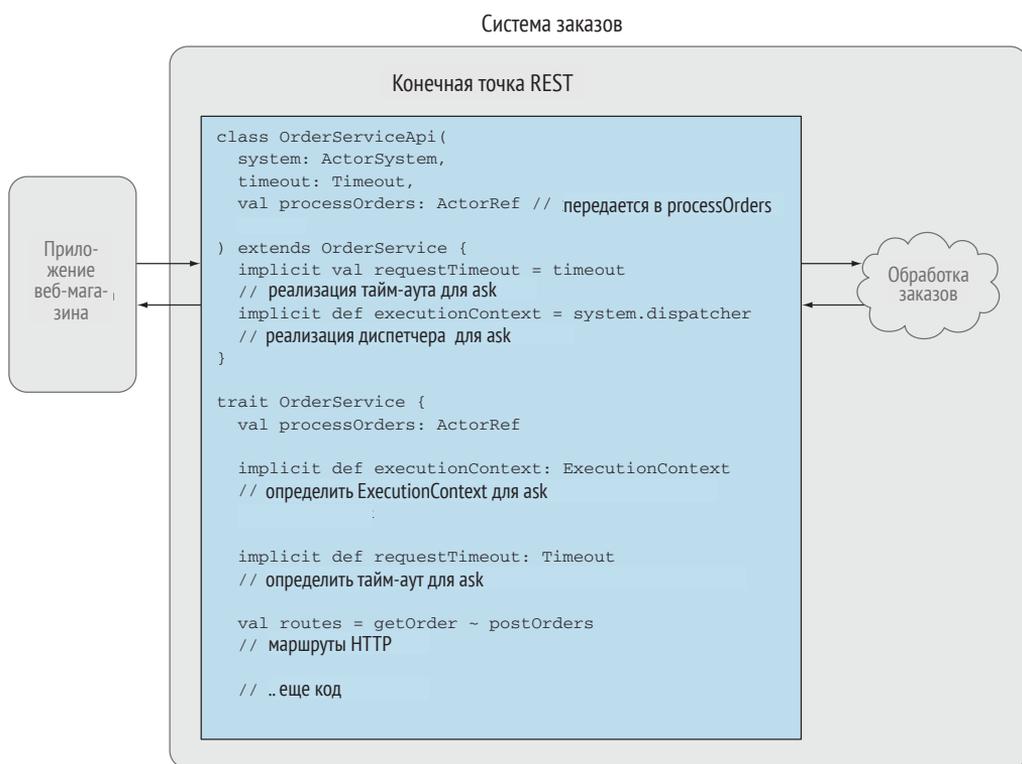


Рис. 12.18. Реализация конечной точки на основе akka-http

#### Листинг 12.12. Тестирование OrderService

```

package aia.integration

import scala.concurrent.duration._
import scala.xml.NodeSeq

```

```
import akka.actor.Props
```

```
import akka.http.scaladsl.marshallers.xml.ScalaXmlSupport._
import akka.http.scaladsl.model.StatusCodes
import akka.http.scaladsl.server._
import akka.http.scaladsl.testkit.ScalatestRouteTest
```

← Импорт неявных преобразований для поддержки XML, чтобы можно было использовать `responseAs[NodeSeq]`

```
import org.scalatest.{ Matchers, WordSpec }
```

```
class OrderServiceTest extends WordSpec
```

```
  with Matchers
```

```
  with OrderService
```

```
  with ScalatestRouteTest {
```

← Подмешивание `OrderService` в тест

```
  implicit val executionContext = system.dispatcher
  implicit val requestTimeout = akka.util.Timeout(1 second)
  val processOrders =
    system.actorOf(Props(new ProcessOrders), "orders")
```

← Определение DSL для тестирования маршрутов

```
  "The order service" should {
    "return NotFound if the order cannot be found" in {
      Get("/orders/1") ~> routes ~> check {
        status shouldEqual StatusCodes.NotFound
      }
    }
  }
```

← Убедиться, что в ответ на запрос несуществующего заказа возвращается 404 NotFound

```
  "return the tracking order for an order that was posted" in {
    val xmlOrder =
      <order><customerId>customer1</customerId>
      <productId>Akka in action</productId>
      <number>10</number>
    </order>
```

```
    Post("/orders", xmlOrder) ~> routes ~> check {
      status shouldEqual StatusCodes.OK
      val xml = responseAs[NodeSeq]
      val id = (xml \ "id").text.toInt
      val orderStatus = (xml \ "status").text
      id shouldEqual 1
      orderStatus shouldEqual "received"
    }
  }
```

← Убедиться, что запрос GET вслед за запросом POST вернет состояние заказа

```
    Get("/orders/1") ~> routes ~> check {
      status shouldEqual StatusCodes.OK
      val xml = responseAs[NodeSeq]
      val id = (xml \ "id").text.toInt
      val orderStatus = (xml \ "status").text
```

←

```

        id shouldEqual 1
        orderStatus shouldEqual "processing"
    }
}
}
}

```

Определение маршрутов выполняется с использованием директив. Директивы можно считать правилами, которым должны соответствовать получаемые HTTP-запросы. Директива выполняет одну или несколько функций из числа следующих:

- преобразует запрос;
- фильтрует запрос;
- выполняет запрос.

Директивы – это маленькие строительные блоки, из которых можно конструировать маршруты и структуры обработки произвольной сложности. Вот стандартная форма директивы:

```
имя(аргументы) { выборка => ... // внутренний маршрут }
```

В akka-http имеется большое количество предопределенных директив, но вы также можете создавать свои директивы. В этом примере мы использовали несколько наиболее распространенных директив. Route использует директивы для сопоставления HTTP-запроса и извлечения данных из них. Маршрут должен заканчиваться HTTP-ответом для каждого найденного совпадения. Начнем с определения routes в OrderService, скомпоновав два маршрута вместе: один – для получения и другой – для добавления заказов.

#### Листинг 12.13. Определение маршрутов в OrderService

```
val routes = getOrder ~ postOrders
```

Оператор ~ объединяет маршруты и/или директивы. Это выражение можно читать как «getOrder или postOrders». Каждый запрос, не соответствующий маршруту postOrders или getOrder, завершается возвратом HTTP-ответа 404 Not Found; например, запрос по адресу order или запрос DELETE.

Теперь рассмотрим метод getOrder. Он использует директиву get, которой соответствуют запросы GET. Затем с помощью директивы pathPrefix проверяется соответствие маршрута шаблону "/orders/[id]" и с помощью IntNumberPathMatcher извлекается номер заказа.

#### Листинг 12.14. Обработка GET-запроса по адресу orders/id в OrderService

```
def getOrder = get {
    pathPrefix("orders" / IntNumber) { id =>

```

← Проверяет соответствие типа запроса GET  
 ← Извлекает номер заказа

```

onSuccess(processOrders.ask(OrderId(id))) {
  case result: TrackingOrder =>
    complete(
      <statusResponse>
        <id>{ result.id }</id>
        <status>{ result.status }</status>
      </statusResponse>)
  case result: NoSuchOrder =>
    complete(StatusCodes.NotFound)
}
}
}

```

← onSuccess передает результат в Future  
внутреннему маршруту; посылает  
OrderId актору ProcessOrders

← Завершение запроса ответом XML.  
Преобразование TrackingOrder в  
XML-ответ

← Завершение запроса  
с HTTP-кодом 404 NotFound

Директива `IntNumber` извлекает номер заказа `id` из URL и преобразует его в число типа `Int`. Если запрос `GET` не содержит номера заказа, соответствие признается несостоявшимся и обратно возвращается HTTP-ответ `404 Not Found`. Если номер заказа указан, создается бизнес-объект `OrderId`, который затем передается системе для обработки.

Получив объект `OrderId`, его можно переслать в систему и создать ответ. Это делается с применением директивы `complete`.

Директива `complete` возвращает ответ на запрос. В простейшей реализации результат возвращается непосредственно, без всяких изменений. Но в нашем случае требуется асинхронно обработать ответ от актора `ProcessOrders`, прежде чем создать ответ для передачи внешней системе. Поэтому мы используем обработчик `onSuccess`, который передает результат из объекта `Future` по внутреннему маршруту. Блок кода в методе `onSuccess` выполняется, когда `Future` получает результат асинхронных вычислений, поэтому будьте внимательнее при использовании ссылок. После передачи `scala.xml.NodeSeq` в директиву `complete` модуль `akka-http` выполнит маршаллинг (преобразует) `NodeSeq` в текст и автоматически установит тип содержимого ответа как `text/xml`. Это все, что необходимо для обработки запросов `GET`.

**МАРШАЛЛИНГ ОТВЕТОВ.** Возможно, вам интересно, как `akka-http` получает HTTP-ответ из `scala.xml.Elem`. С этой целью вы должны предоставить `ToEntityMarshaller` в неявной области видимости, который будет выполнять маршаллинг `scala.xml.Elem` в сущность `text/html`. Для этого достаточно импортировать `akka.http.scaladsl.marshallers.xml.ScalaXmlSupport._`, благодаря чему будут импортированы `ToEntityMarshaller` и `FromEntityUnmarshaller` для XML.

Теперь перейдем к реализации обработки запросов `POST`. Она во многом похожа на обработку запросов `GET`. Единственное отличие – нет необ-

ходимости указывать номер заказа в URL, но нужно предусмотреть обработку тела запроса. Используем для этого директиву `entity`:

```
post {
  path("orders") {
    entity(as[NodeSeq]) { xml =>
      val order = toOrder(xml)
    }
  }
  //... еще код
}
```

Директива `entity(as[NodeSeq])` может работать, только если в неявной области видимости присутствует `FromEntityUnmarshaller`, для чего нужно импортировать пакет `ScalaXmlSupport._`, содержащий неявное определение `ToEntityMarshaller([NodeSeq])`.

Метод `toOrder`, который здесь не показан, преобразует `scala.xml.NodeSeq` в `Order`.

Теперь, получив объект `Order`, можно создать ответ на запрос POST. В листинге 12.15 показана законченная реализация метода `postOrders`.

**Листинг 12.15.** Обработка результатов попытки создать заказ в `OrderService`

```
def postOrders = post {
  path("orders") {
    entity(as[NodeSeq]) { xml =>
      val order = toOrder(xml)
      onSuccess(processOrders.ask(order)) {
        case result: TrackingOrder =>
          complete(
            <confirm>
              <id>{ result.id }</id>
              <status>{ result.status }</status>
            </confirm>
          )
        case result =>
          complete(StatusCodes.BadRequest)
      }
    }
  }
}
```

← Проверяет соответствие типа запроса POST

← Проверяет соответствие адресу /orders

← Преобразовать в объект Order

← Преобразование тела запроса в scala.xml.NodeSeq

← Завершает запрос XML-ответом

← Если ProcessActor вернет любое другое сообщение, ответить кодом состояния BadRequest

Теперь у нас реализованы все маршруты. Но как должна выполняться дальнейшая обработка? Чтобы создать настоящий сервер, нужно связать маршруты с HTTP-сервером. Сервер можно создать в момент запуска приложения, задействовав расширение `Http`, как показано в листинге 12.16.

**Листинг 12.16.** Запуск HTTP-сервера

```

object OrderServiceApp extends App
  with RequestTimeout {
  val config = ConfigFactory.load()
  val host = config.getString("http.host")
  val port = config.getInt("http.port")

  implicit val system = ActorSystem()
  implicit val ec = system.dispatcher

  val processOrders = system.actorOf(
    Props(new ProcessOrders), "process-orders"
  )

  val api = new OrderServiceApi(system,
    requestTimeout(config),
    processOrders).routes

  implicit val materializer = ActorMaterializer()
  val bindingFuture: Future[ServerBinding] =
    Http().bindAndHandle(api, host, port)

  val log = Logging(system.eventStream, "order-service")
  bindingFuture.map { serverBinding =>
    log.info(s"Bound to ${serverBinding.localAddress} ")
  }.onFailure {
    case ex: Exception =>
      log.error(ex, "Failed to bind to {}:{}!", host, port)
      system.terminate()
  }
}

```

← Трейт RequestTimeout читает akka.http.server.request-timeout из конфигурации  
 ← Получить имя хоста и порт из конфигурации  
 ← Создать актор ProcessOrders  
 ← OrderServiceApi возвращает маршруты  
 ← Связать маршруты с HTTP-сервером  
 ← Записать в журнал сообщение об успешном запуске службы  
 ← Записать в журнал сообщение о неудачной попытке установить соединение с заданным хостом и номером порта

Вы можете сами протестировать OrderServiceApp с помощью любого HTTP-клиента, запустив приложение в sbt.

## 12.4. В заключение

Для интеграции с другими системами требуется множество инструментов, которые Akka предлагает, что называется, «из коробки»:

- асинхронное выполнение задач и поддержка обмена сообщениями;
- средства преобразования данных;
- поддержка шаблона производитель/потребитель.

Для упрощения интеграции мы использовали модуль akka-http и библиотеку Apache Camel, что позволило нам сосредоточиться на реализации ти-

пичных шаблонов интеграции исключительно с применением фреймворка Akka, и при этом нам не пришлось писать много кода, который был бы тесно связан с выбором транспортного или компонентного уровня.

Фреймворк Akka способен принять на себя основные тяготы, связанные с интеграцией систем. Довольно часто самым сложным аспектом интеграции является организация оптимального обслуживания входящих и исходящих потоков данных при ограниченной производительности или высоких требованиях к надежности. Кроме тем, рассмотренных здесь, – потребление услуг, получение данных, их преобразование и предоставление услуг другим потребителям – не менее важными факторами, обеспечивающими высокую надежность и масштабируемость интеграции, являются основные особенности модели акторов, поддержка конкурентного выполнения и отказоустойчивость. Легко представить, как можно было бы расширить примеры шаблонов, представленные здесь, приемами увеличения отказоустойчивости из главы 4 и механизмами масштабирования из глав 6 и 9.

# Глава 13

## Потоковые приложения

В этой главе:

- обработка потоков событий в ограниченной области памяти;
- потоковая передача событий через HTTP с помощью akka-http;
- ветвление и слияние с помощью специализированного предметного языка;
- промежуточная обработка потоковых данных между производителями и потребителями.

В главе 12 вы узнали, как интегрировать приложения на основе Akka с внешними службами, используя методику «запрос/ответ». В этой главе мы рассмотрим интеграцию с внешними системами, использующими механизмы потоковой передачи данных.

Потоковые данные – это последовательность элементов, которая может не иметь конца. Концептуально поток имеет временный характер, он существует, только пока существует производитель, посылающий элементы в поток, и потребитель, извлекающий элементы из потока.

Одна из сложностей, которые приходится преодолевать в приложениях, потребляющих потоки данных, – отсутствие информации об объеме данных, которые потребуются обработать, потому что в любой момент могут поступить новые данные. Другая сложность – разная скорость работы производителя и потребителя. Если ваше приложение занимает промежуточное положение между производителем и потребителем потоковых данных, вам придется решить проблему буферизации данных так, чтобы не исчерпать память. Как на стороне производителя узнать, что потребитель не справляется с обработкой поступающих данных?

Как вы увидите в этой главе, модуль akka-stream дает возможность обрабатывать неограниченные объемы данных с использованием ограниченных буферов. Модуль akka-stream является основополагающим API для создания потоковых приложений в Akka. Модуль akka-http (который внутренне использует akka-stream) поддерживает потоковые операции через

HTTP. Создание потоковых приложений с Akka – обширная тема, поэтому данную главу нужно рассматривать лишь как введение в akka-stream API и использование akka-http для потоковой обработки, от простых конвейеров до более сложных графоподобных компонентов.

В этой главе мы рассмотрим пример структурированной обработки прикладного журнала. Многие приложения создают свои файлы журналов, куда во время выполнения сохраняют информацию для отладки. Мы начнем с обработки файлов журналов произвольного размера и организуем выборку интересующих нас событий, не загружая файл в память целиком.

После этого мы напишем службу потоковой обработки журналируемых сообщений с использованием akka-http.

## 13.1. Основы потоковой обработки

Посмотрим для начала, что подразумевается под потоковой обработкой с использованием модуля akka-stream. На рис. 13.1 показано, как узел обработки осуществляет обработку элементов по одному. Обработка по одному – важное условие, помогающее предотвратить исчерпание памяти. На рис. 13.1 также показано, как в некоторых местах в цепочке обработки можно использовать ограниченные буферы.

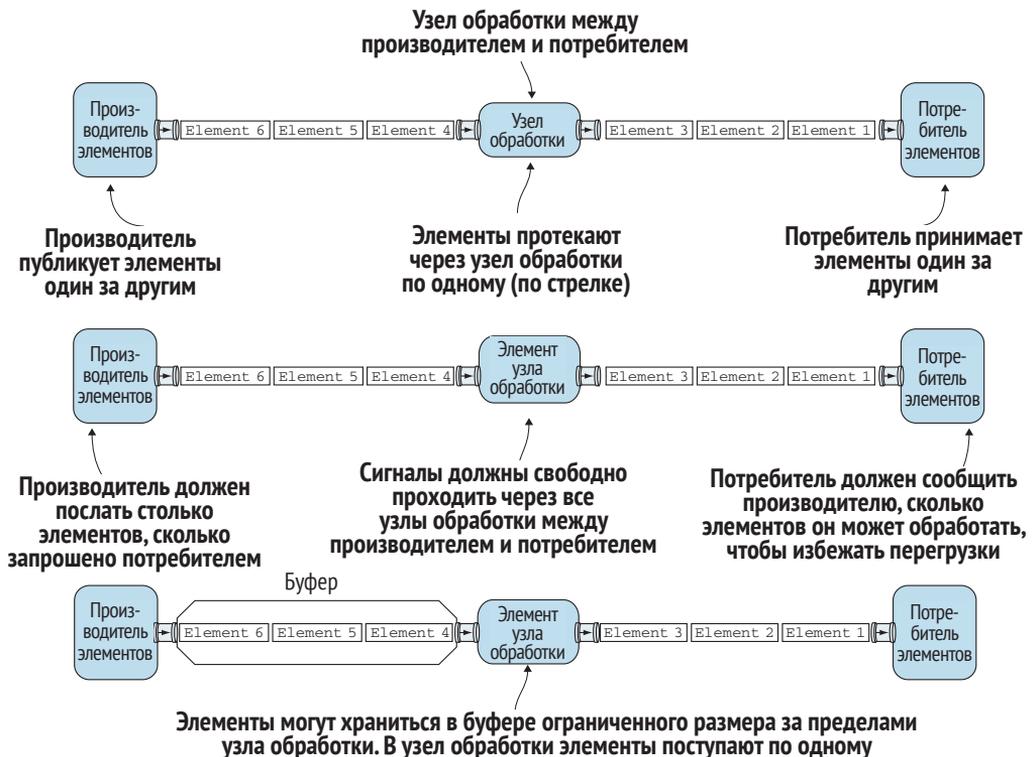


Рис. 13.1. Потоковая обработка

Сходство с акторами более чем очевидно. Разница, как показано на рис. 13.1, лишь в обмене сигналами между производителями и потребителями, ограничивающими количество посылаемых элементов, чтобы избежать исчерпания памяти, который вы должны организовать сами. На рис. 13.2 показаны примеры линейных цепочек обработки, которые мы должны реализовать в обработчике журналируемых событий, такие как фильтрация, преобразование и сборка журналируемых событий.

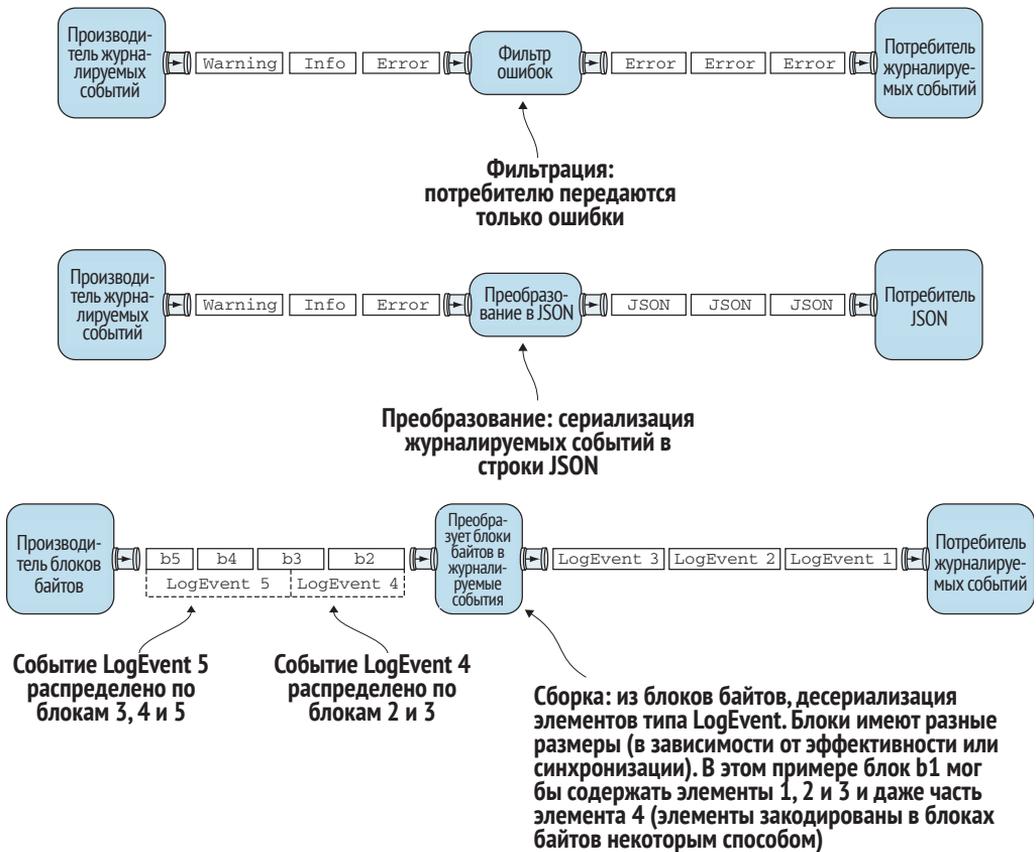
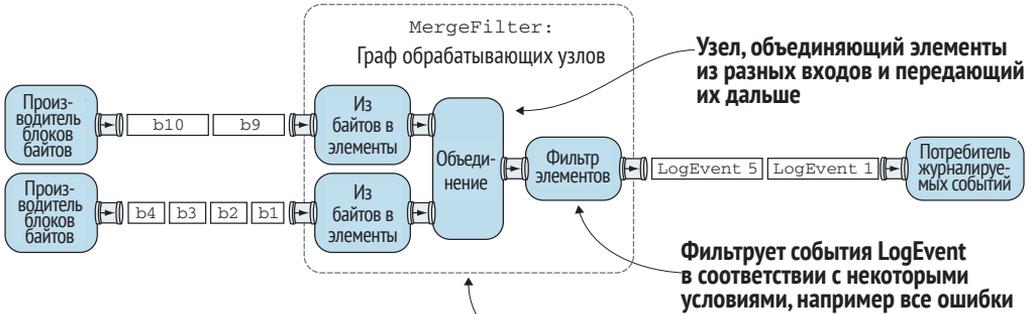
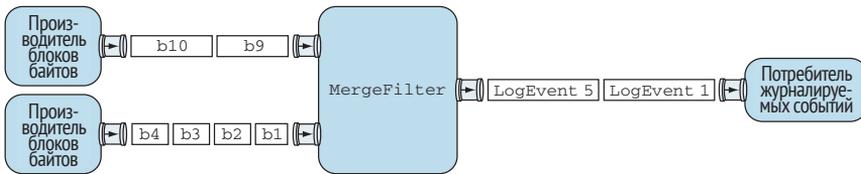


Рис. 13.2. Линейная обработка потока

Обработчику журналируемых событий придется делать чуть больше, чем просто передавать события от производителя потребителю, и в его реализации мы используем *граф*. Граф обработки позволяет сконструировать сложную логику обработки из имеющихся обрабатывающих узлов. Например, на рис. 13.3 показан граф, объединяющий два потока и выполняющий фильтрацию элементов. По сути, любой обрабатывающий узел является графом; граф – это обрабатывающий элемент с несколькими входами и выходами.



Граф MergeFilter обрабатывающих узлов может реализовать более сложную логику обработки путем комбинирования простых узлов с произвольным количеством входов и выходов (в данном случае объединяются два потока байтов и фильтруются определенные события)



Узел – это граф определенной формы. Форма задается количеством входов и выходов. Графы могут состоять из других графов. Модуль akka-stream предоставляет несколько стандартных графов обработки

Рис. 13.3. Графоподобная обработка

Окончательная версия обработчика журналируемых событий будет принимать потоки событий от нескольких служб по сети с использованием протокола HTTP и объединять их в потоки разного типа. Он будет производить фильтрацию, анализ, преобразование и пересылать результаты другим службам. На рис. 13.4 показан гипотетический пример использования такого обработчика.

На рисунке видно, что обработчик принимает события из разных разделов приложения Tickets. События передаются обработчику немедленно или с некоторой задержкой. Веб-приложение Tickets и HTTP-служба посылают события немедленно, как только они случаются, тогда как служба перенаправления журналируемых событий посылает события только после их извлечения из журналов сторонних служб.

В варианте, представленном на рис. 13.4, обработчик потока событий посылает идентифицированные события службе архивирования, чтобы позднее пользователь мог выполнять запросы в поисках нужных событий. Служба журналирования также будет выявлять проблемы, возникающие в

прикладных службах, и использовать службу уведомлений для извещения команды сопровождения о ситуациях, требующих участия человека. Некоторые события преобразуются в показатели, которые можно передать службе отчетов для последующего анализа.

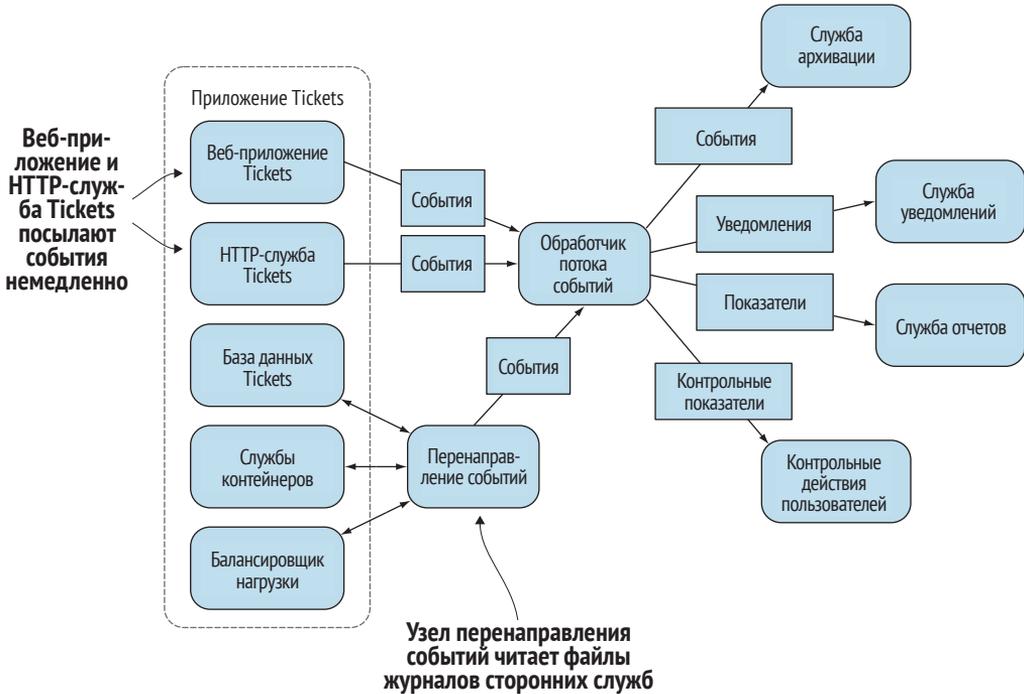


Рис. 13.4. Вариант использования обработчика журналируемых событий

Преобразование журналируемых событий в архивные записи, уведомления, показатели и контрольные показатели будет производиться разными потоками обработки, каждый из которых имеет свою, отличную от других логику работы.

Этот пример обработчика потока событий преследует несколько целей, приемы достижения которых будут продемонстрированы в следующих разделах.

- *Ограниченное использование памяти* – обработчик потока событий не должен сталкиваться с проблемой нехватки памяти. События должны обрабатываться по одному, допускается накопление нескольких событий во временных буферах, но никогда они не должны читаться в память все сразу.
- *Асинхронный и неблокирующий ввод/вывод* – ресурсы должны использоваться эффективно, и количество блокировок должно быть сведено к минимуму. Например, обработчик потока событий не должен

последовательно посылать данные всем службам и ждать, пока все они ответят.

- *Регулировка скорости* – производители и потребители должны иметь возможность работать с разной скоростью.

Окончательная реализация обработчика потока событий будет представлена потоковой НТТР-службой, но для начала было бы хорошо написать упрощенную версию, которая извлекает события для обработки из файла и записывает результаты в файл. Как вы увидите далее, модуль akka-stream обладает большой гибкостью. Он позволяет легко отделить логику обработки от выбора типов потоков для чтения и записи. В следующих разделах мы шаг за шагом будем конструировать приложение обработки потока событий, начав с простой программы, которая всего лишь копирует данные из одного потока в другой. Попутно мы исследуем прикладной интерфейс akka-stream и обсудим варианты, предлагаемые этим модулем для потоковой обработки.

### 13.1.1. Копирование файлов

Первым шагом на пути создания приложения обработки потока событий для нас станет создание примера копирования данных из одного потока в другой. Каждый байт, прочитанный из входного потока, будет записываться в выходной поток.

Как обычно, сначала добавим зависимости в файл сборки, как показано в листинге 13.1.

#### Листинг 13.1. Зависимости

```
"com.typesafe.akka" %% "akka-stream" % version, ← Зависимость для потоковой обработки
```

Работа с akka-stream обычно делится на два этапа:

1. *определение модели* – набора компонентов, составляющих *граф* обработки потока. Граф определяет порядок обработки потоков;
2. *запуск модели* – запуск графа в системе акторов ActorSystem. Граф преобразуется в акторы, которые выполняют фактическую обработку потока данных.

Граф (модель) доступен во всей программе. После создания он становится неизменяемым. Граф можно запустить столько раз, сколько понадобится, и каждый раз будет создаваться новый набор акторов. Запущенный граф может возвращать результаты из компонентов, участвующих в обработке потока. Детали работы графа будут подробно обсуждаться далее в этой главе, поэтому не волнуйтесь, если прямо сейчас вам будет что-то непонятно.

Начнем с очень простого прототипа будущего приложения и создадим обработчик потока событий, который просто копирует файлы журналов. Приложение `StreamingCopy` копирует входной файл в выходной. Моделью в данном случае является простой *конвейер*. Любые данные, прочитанные из входного файла, записываются в выходной файл. Большая часть кода, относящегося к задаче, приводится в листингах 13.3 и 13.4, в первом определяется модель, а во втором показано, как запустить эту модель.

В эти листинги не попал код, получающий входной и выходной файлы, `inputFile` и `outputFile`, из аргументов командной строки. В листинге 13.2 показаны наиболее важные инструкции импорта.

### Листинг 13.2. Инструкции импорта в приложении `StreamingCopy`

```
import akka.actor.ActorSystem
import akka.stream.{ ActorMaterializer, IOResult }
import akka.stream.scaladsl.{ FileIO, RunnableGraph, Source, Sink }
import akka.util.ByteString
```

← Пакет `scaladsl` содержит определение предметно-ориентированного языка для работы с потоками; кроме него, существует также `javadsl`

### Блокирующие операции ввода/вывода с файлами

В этом примере используется `FileIO`, потому что проверить входной и выходной файлы не составляет труда, и ввод/вывод в файловые конечные точки `Source` и `Sink` осуществляется намного проще.

Типы конечных точек легко поменять, например заменить файлы чем-то другим.

Обратите внимание, что конечные точки, созданные с использованием `FileIO`, внутренне используют блокирующие операции ввода/вывода. Акторы, созданные для конечных точек с `FileIO`, выполняются под управлением отдельного диспетчера, который можно настроить глобально с помощью `akka.stream.blocking-io-dispatcher`. Также можно реализовать свой диспетчер для элементов графа посредством `withAttributes`, принимающего `ActorAttributes`. В разделе 13.1.2 будет показан пример настройки `supervisorStrategy` с применением `ActorAttributes`.

Блокирующие операции ввода/вывода с файлами на самом деле не так страшны, как могло бы показаться. Задержка операций с диском, например, намного меньше задержек, возникающих при выполнении сетевых операций. В будущем может появиться асинхронная версия `FileIO`, если выяснится, что она позволяет получить более высокую производительность при работе сразу с несколькими файловыми потоками.

**Листинг 13.3.** Определение приложения `RunnableGraph`, копирующего файлы

```

val source: Source[ByteString, Future[IOResult]] =
  FileIO.fromPath(inputFile)
val sink: Sink[ByteString, Future[IOResult]] =
  FileIO.toPath(outputFile, Set(CREATE, WRITE, APPEND))
val runnableGraph: RunnableGraph[Future[IOResult]] =
  source.to(sink)

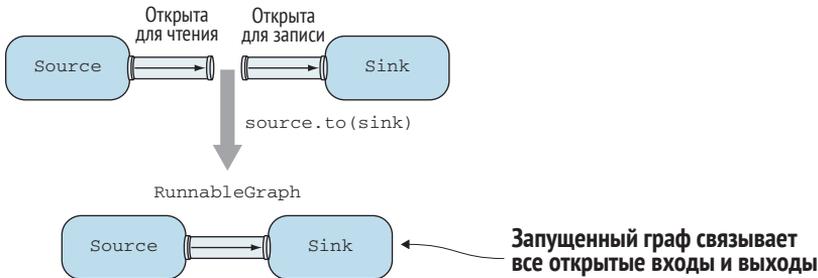
```

← Входной файл для чтения (источник)  
 ← Выходной файл для записи (приемник)  
 ← Соединение входного и выходного файлов создает `RunnableGraph`

Сначала определяются источник (`Source`) и приемник (`Sink`) вызовом `FileIO.fromPath` и `FileIO.toPath` соответственно.

`Source` и `Sink` – это конечные точки потока (источник и приемник). Конечная точка `Source` открыта для чтения, а `Sink` – для записи. `Source` и `Sink` типизированы; в данном случае элементами потока для обеих являются объекты `ByteString`.

`Source` и `Sink` связываются вместе и образуют `RunnableGraph`, как показано на рис. 13.5.



**Рис. 13.5.** `Source`, `Sink` и простейший `RunnableGraph`

**МАТЕРИАЛИЗОВАННЫЕ ЗНАЧЕНИЯ.** Источники и приемники могут передавать вспомогательное значение после запуска графа, которое называется *материализованным значением*. В данном случае это значение `Future[IOResult]`, содержащее количество прочитанных или записанных байтов. Мы подробно обсудим материализацию в разделе 13.1.2.

Приложение `StreamingCopy` создает самый простой граф, какой только можно определить с использованием `source.to(sink)` – `RunnableGraph`, – который принимает данные из источника `Source` и передает их в приемник `Sink`.

Строки, создающие источник и приемник, являются декларативными. Они не создают файлы и не открывают дескрипторы, а просто сохраняют всю информацию, которая может понадобиться позже, когда будет запущен `RunnableGraph`.

Также важно отметить, что создание `RunnableGraph` не означает его запуск. Эта операция просто определяет модель копирования.

В листинге 13.4 показано, как выполнить запуск `RunnableGraph`.

**Листинг 13.4.** Запуск графа `RunnableGraph`, осуществляющего копирование потока

```
implicit val system = ActorSystem()
implicit val ec = system.dispatcher
implicit val materializer = ActorMaterializer()

runnableGraph.run().foreach { result =>
  println(s"${result.status}, ${result.count} bytes read.")
  system.terminate()
}
```

Материализатор в конечном счете создаст акторы, выполняющие работу, возложенную на граф

Операция запуска графа вернет `Future[IOResult]`; в данном случае `IOResult` содержит количество прочитанных байтов

Операция запуска графа `runnableGraph` возвращает количество байтов, скопированных из источника в приемник – в данном случае из входного файла в выходной. Говорят, что граф *материализуется* после запуска.

В данном примере граф автоматически останавливается после копирования всех данных. Подробнее об этом мы поговорим в следующем разделе.

Объект `FileIO` является частью `akka-stream` и обеспечивает удобные средства для создания источников и приемников на основе файлов. Соединение источника и приемника приводит к поочередному чтению всех строк типа `ByteString` из входного файла и их записи в выходной файл, сразу после материализации `RunnableGraph`.

## Запуск примеров

Примеры из этой главы можно запускать как обычно – в консоли `sbt`. Команде запуска `run` можно передать любые аргументы, необходимые запускаемому приложению.

Для опробования примеров можно использовать очень удобный плагин `sbt-revolver`, который позволяет запускать, перезапускать и останавливать приложения (командами `re-start` и `re-stop`) без выхода из консоли `sbt`. Его можно найти по адресу: <https://github.com/spray/sbt-revolver>. В папке `chapter-stream` проекта на GitHub также имеется приложение `GenerateLogFile`, с помощью которого можно создавать огромные файлы журналов для тестирования.

Чтобы убедиться, что приложение тайно не загружает файл в память целиком, попробуйте использовать параметр `-Xmx`, ограничивающий объем памяти, доступной виртуальной машине Java.

В следующем разделе мы рассмотрим особенности материализации и узнаем, как действует `RunnableGraph`.

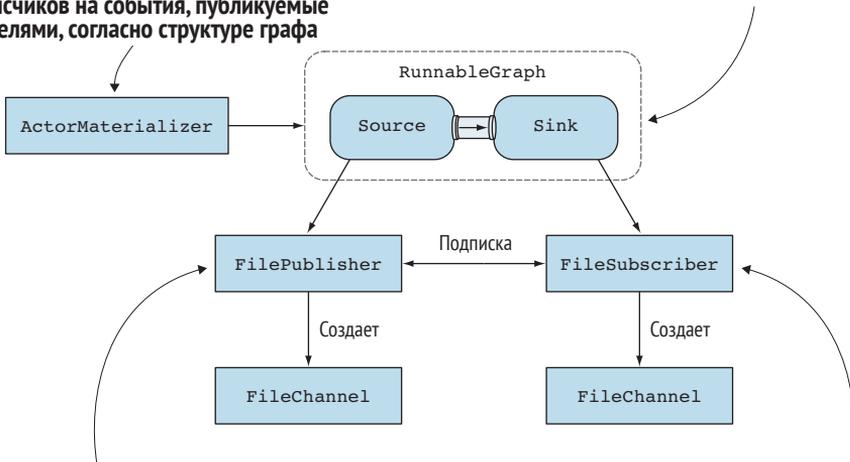
### 13.1.2. Материализация запускаемых графов

Метод `run` в листинге 13.4 требует присутствия `Materializer` в неявной области видимости. Объект `ActorMaterializer` преобразует `RunnableGraph` в набор акторов, которые выполняют всю работу в графе.

Давайте посмотрим, что при этом происходит в данном конкретном примере копирования файлов. Некоторые из этих деталей могут измениться в будущем, потому что являются внутренними особенностями Акка, тем не менее очень полезно пройти по коду и посмотреть, как все это работает. На рис. 13.6 показана упрощенная версия материализации графа `StreamingCopy`.

1. `RunnableGraph` создается, только если все входы и выходы в графе связаны друг с другом. Материализатор проверяет соединения между входами и выходами в графе, посылает команды конечным точкам `Source` и `Sink` создать акторы издателей и подписчиков и подписывает подписчиков на события, публикуемые издателями, согласно структуре графа

2. `Source` и `Sink` объединяются в один модуль, представляющий модель графа



3. `Source` создает актор `FilePublisher`, который, в свою очередь, создает `FileChannel` для чтения из файла

4. `Sink` создает актор `FileSubscriber`, который, в свою очередь, создает `FileChannel` для записи в файл

Рис. 13.6. Материализация графа

- `ActorMaterializer` проверяет правильность соединения `Source` и `Sink` в графе и посылает команды внутренним механизмам `Source` и `Sink` выполнить подготовку ресурсов. Внутренне `fromPath` создает `Source` из `FileSource` (внутренняя реализация `SourceShape`).
- Объекту `FileSource` посылается команда создать его ресурсы, и затем создается актор `FilePublisher`, который открывает `FileChannel`.

- Метод `toPath` создает `Sink` из `FileSinkSinkModule`. Объект `FileSink` создает актер `FileSubscriber`, который открывает `FileChannel`.
- Метод `to`, используемый в примере для соединения источника и приемника, внутренне объединяет модули источника и приемника в один модуль.
- `ActorMaterializer` подписывает подписчиков на события, публикуемые издателями, согласно описанным соединениям модулей, в данном случае `FileSubscriber` подписывается на события, публикуемые `FilePublisher`.
- `FilePublisher` читает строки `ByteString` из файла, пока не достигнет конца, и в момент остановки закрывает файл.
- Все строки `ByteString`, полученные от `FilePublisher`, актер `FileSubscriber` записывает в выходной файл. В момент остановки `FileSubscriber` закрывает `FileChannel`.
- `FilePublisher` завершает поток, как только прочитает все данные из файла. `FileSubscriber` получает сообщение `OnComplete` и также закрывает файл, в который производил запись.

Поток можно остановить с помощью таких операторов, как `take`, `takeWhile` и `takeWithin`, которые соответственно останавливают поток после обработки заданного числа элементов, когда функция-предикат вернет `true` и когда истечет заданный интервал времени. Внутренне эти операторы останавливают поток похожими способами.

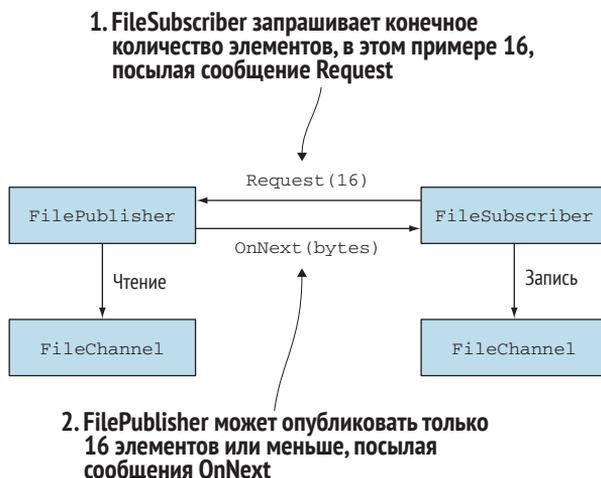
В этой точке останавливаются все акторы, созданные для выполнения работы. Повторный запуск `RunnableGraph` создаст новый набор акторов, и весь процесс повторится с самого начала.

### Предотвращение переполнения памяти

Если бы `FilePublisher` загружал файл в память целиком (чего он не делает), это могло бы привести к исключению `OutOfMemoryException`. Но как тогда он действует на самом деле? Ответ на этот вопрос заключается во взаимодействиях между издателем и подписчиком, как показано на рис. 13.7.

Издатель `FilePublisher` может опубликовать элементов не больше, чем запрошено подписчиком `FileSubscriber`.

В данном случае `FilePublisher` может прочитать очередную порцию данных из входного файла, только когда `FileSubscriber` запросит их. Это означает, что данные читаются из входного файла с той же скоростью, с какой они могут записываться в выходной файл. В этом простом примере мы имеем в графе только два простых компонента. В более сложных графах требование на загрузку дополнительных данных проходит весь путь от конца графа к началу, гарантируя, что издатель не сможет публиковать события быстрее, чем требуется подписчикам.



**Рис. 13.7.** Подписчик запрашивает у издателя ровно столько данных, сколько сможет обработать

Подобным образом работают все компоненты графа в akka-stream. В конечном счете они все превращаются в издателей или подписчиков Reactive Streams. Именно этот API позволяет akka-stream обрабатывать бесконечные потоки данных с ограниченным объемом памяти и устанавливает правила взаимодействий издателей и подписчиков, например чтобы издатель не мог опубликовать элементов больше, чем было запрошено.

Здесь мы существенно упростили протокол взаимодействий издателя и подписчика. Что самое важное – подписчик и издатель обмениваются сообщениями асинхронно. Они никак не блокируют друг друга. Подписчик может сообщить издателю уменьшить или увеличить скорость подачи данных. Такой способ работы подписчика называется *неблокирующим обратным давлением*.

**ИНИЦИАТИВА REACTIVE STREAMS.** Reactive Streams – это инициатива по разработке стандарта асинхронной обработки потоков с неблокирующим обратным давлением. Есть несколько библиотек, реализующих прикладной интерфейс Reactive Streams API, обеспечивающий возможность их интеграции друг с другом. Модуль akka-stream также реализует Reactive Streams API и предоставляет высокоуровневый API для работы с ним. Более подробную информацию вы найдете по адресу: [www.reactive-streams.org/](http://www.reactive-streams.org/).

## Внутренние буферы

Модуль akka-stream использует внутренние буферы для оптимизации. Вместо того чтобы запрашивать и публиковать элементы по одному, они внутренне запрашиваются и публикуются пакетами.

## Оператор слияния

В случаях, когда обработка потока в графе выполняется несколькими узлами, модуль `akka-stream` использует прием оптимизации, который называют *оператором слияния*, для устранения избыточных асинхронных границ и создания линейных цепочек обработки в графе.

По умолчанию `akka-stream` стремится выполнить как можно больше этапов в единственном акторе, чтобы уменьшить накладные расходы на передачу элементов и запросов между потоками выполнения. Для создания асинхронной границы в графе можно явно использовать метод `async`, вследствие чего этапы обработки элементов, разделенные вызовами `async`, гарантированно будут выполняться в отдельных акторах.

Слияние осуществляется во время материализации. Его можно отключить, установив параметр `akka.stream.materializer.auto-fusing=off`. Также есть возможность принудительно выполнить слияние (до материализации графа) вызовом `Fusing.aggressive(graph)`.

Актор `FileSubscriber` может запросить фиксированное число элементов за раз. Модуль `akka-stream` гарантирует, что для чтения данных из файлов и записи их в файлы будет использоваться ограниченный объем памяти. Этот механизм надежен, и вам не нужно беспокоиться о его работе, но у любопытствующих вполне может возникнуть вопрос: какое максимальное количество элементов запрашивает `FileSubscriber`?

Если углубиться в код, можно заметить, что `FileSubscriber` использует `WatermarkRequestStrategy` со входным буфером максимального размера. То есть `FileSubscriber` не будет запрашивать элементов больше, чем определено этим параметром.

Кроме того, сами элементы тоже имеют определенный размер, который мы не обсуждали. В данном случае это фрагмент блока данных, извлекаемого из файла, который можно задать в вызове метода `fromPath`, и по умолчанию он равен 8 Кбайт.

Максимальный размер входного буфера определяется максимальным количеством элементов, которое можно задать в конфигурационном параметре `akka.stream.materializer.max-input-buffer-size`. По умолчанию он принимает значение 16, то есть по умолчанию в данном примере во входном буфере может храниться до 128 Кбайт данных.

Максимальный размер входного буфера также можно установить в объекте `ActorMaterializerSettings` и передать его материализатору или конкретному компоненту-графу, как будет показано далее в главе. Объект `ActorMaterializerSettings` позволяет также настроить некоторые другие аспекты материализации, включая тип используемого диспетчера акто-

ров, действующих в составе графа, и порядок управления компонентами графа.

Мы еще раз вернемся к вопросу буферизации в разделе 13.4.

### Объединение материализованных значений

Как отмечалось выше, источник и приемник могут передавать вспомогательное значение после запуска графа. После завершения работы с входными и выходными файлами возвращается объект `Future[IOResult]`, содержащий количество прочитанных и записанных байтов.

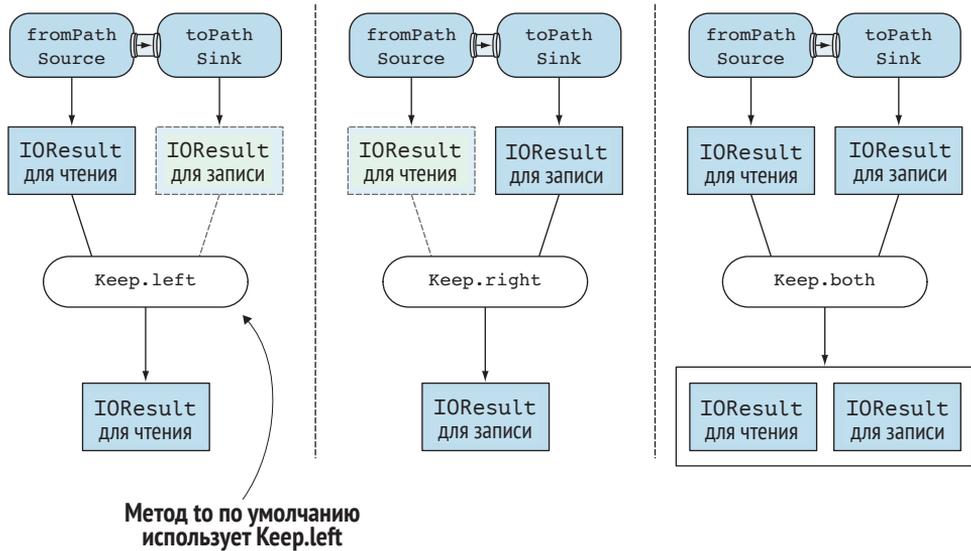


Рис. 13.8. Сохранение материализованных значений в графе

После запуска `RunnableGraph` возвращает одно материализованное значение, поэтому возникает вопрос: какое значение передается через граф?

Метод `to` – это сокращенная форма вызова метода `toMat`, принимающего дополнительный аргумент с функцией объединения материализованных значений. Объект `Keep` определяет для этой цели несколько стандартных функций.

По умолчанию метод `to` использует `Keep.left` и сохраняет материализованное значение слева. Это объясняет, почему для графа в примере `StreamingCopy` возвращается `Future[IOResult]` с количеством байтов, прочитанных из исходного файла (см. рис. 13.8).

При желании с помощью метода `toMat` можно сохранить значение слева, справа, ни одного или оба, как показано в листинге 13.5.

#### Листинг 13.5. Сохранение материализованных значений

```
import akka.Done
```

```
import akka.stream.scaladsl.Keep
```

```
val graphLeft: RunnableGraph[Future[IOResult]] =
  source.toMat(sink)(Keep.left)
val graphRight: RunnableGraph[Future[IOResult]] =
  source.toMat(sink)(Keep.right)
val graphBoth: RunnableGraph[(Future[IOResult], Future[IOResult])] =
  source.toMat(sink)(Keep.both)
val graphCustom: RunnableGraph[Future[Done]] =
  source.toMat(sink) { (l, r) =>
    Future.sequence(List(l,r)).map(_ => Done)
  }
```

Сохранение IOResult для исходного файла

Сохранение IOResult для целевого файла

Сохранение обоих значений

Нестандартная функция, которая просто сообщает, что работа с потоком завершена

`Keep.left`, `Keep.right`, `Keep.both` и `Keep.none` – простые функции, возвращающие значение слева, справа, оба или ни одного соответственно. `Keep.left` – хороший выбор по умолчанию; при материализации больших графов сохраняется материализованное значение, полученное на входе. Если бы по умолчанию использовалась функция `Keep.right`, вам пришлось бы передавать `Keep.left` на каждом этапе, чтобы сохранить первое материализованное значение.

Теперь мы знаем, как объединяются источники и приемники. В следующем разделе мы вернемся к примеру обработки журналируемых сообщений, познакомимся с компонентом `Flow` и поближе рассмотрим потоковые операции в контексте обработки и фильтрации событий.

### 13.1.3. Обработка событий в потоке

Теперь, после знакомства с основами определения и материализации графа, самое время рассмотреть пример, который делает чуть больше, чем просто копирует байты. Для начала создадим первую версию обработки журналов.

Простое приложение командной строки `EventFilter` принимает три аргумента: имя входного файла журнала с событиями, имя выходного файла для записи событий в формате JSON и тип событий для фильтрации (события этого типа будут записываться в выходной файл).

Прежде чем углубляться в операции с потоком, обсудим формат журнала. Каждая запись в журнале представлена одной текстовой строкой, а поля записи разделены символом вертикальной черты (`|`). В листинге 13.6 приводится пример записей в этом формате.

**Листинг 13.6.** Формат записей в файле журнала

```
my-host-1 | web-app | ok      | 2015-08-12T12:12:00.127Z | 5 tickets sold.||
my-host-2 | web-app | ok      | 2015-08-12T12:12:01.127Z | 3 tickets sold.||
```

```
my-host-1 | web-app | ok | 2015-08-12T12:12:02.127Z | 1 tickets sold.||
my-host-2 | web-app | error | 2015-08-12T12:12:03.127Z | exception!!!
```

Каждая запись включает имя хоста, имя службы, тип события, время и описание. Тип события может принимать одно из значений: 'ok', 'warning', 'error' или 'critical'. Каждая запись завершается символом перевода строки (\n).

Каждая строка в файле будет анализироваться и преобразовываться в case-класс Event.

### Листинг 13.7. case-класс Event

```
case class Event(
  host: String,
  service: String,
  state: State,
  time: ZonedDateTime,
  description: String,
  tag: Option[String] = None,
  metric: Option[Double] = None
)
```

Этот класс имеет отдельное поле для каждого элемента записи в журнале.

Для преобразования Event в формат JSON используем библиотеку `spray-json`. Трейт `EventMarshalling`, который здесь не показан, содержит JSON-форматы для case-класса Event. Этот трейт можно найти в репозитории GitHub, вместе с кодом примеров из этой главы, в папке `chapter-stream`.

Мы будем использовать поток обработки Flow между источником Source и приемником Sink, как показано на рис. 13.9.

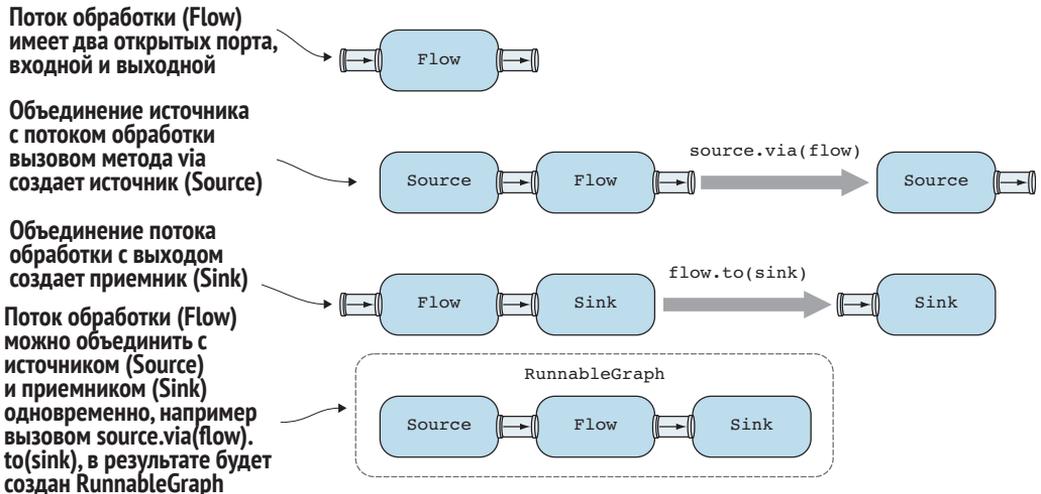


Рис. 13.9. Соединение входов и выходов потоками обработки

Вся логика обработки будет сосредоточена в этом потоке, и мы повторно используем ее позднее, в примере HTTP-версии. `Source` и `Flow` предоставляют методы для работы с потоком данных. На рис. 13.10 представлена концептуальная последовательность операций в фильтре событий.

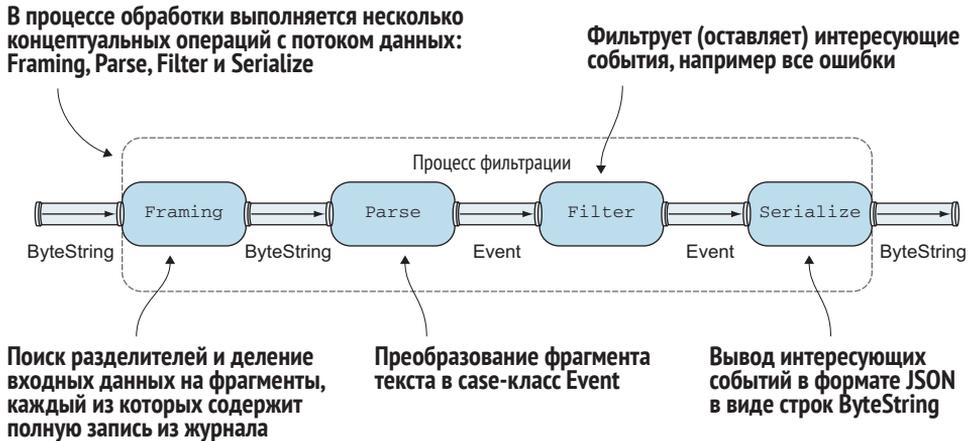


Рис. 13.10. Фильтрация событий

Первая проблема, возникающая перед нами, обусловлена тем, что на вход блока обработки подаются строки `ByteString` произвольного размера, прочитанные из источника. Каждая такая строка необязательно содержит единственную и полную запись.

В `akka-stream` есть несколько predefined версий `Flow`, предлагающих разные способы выделения записей из потока данных. В данном случае мы будем использовать версию `Framing.delimiter`, которая выделяет записи по заданной строке-разделителю `ByteString`. Он буферизует до `maxLine` байт, пытаясь выделить запись, завершающуюся указанным разделителем, и гарантирует, что поврежденные входные данные не приведут к исключению `OutOfMemoryException`.

В листинге 13.8 показано, как строки `ByteString` произвольного размера можно преобразовать в строки `ByteString` с записями, завершающимися символом перевода строки.

#### Листинг 13.8. Выделение записей из потока данных

```
val frame: Flow[ByteString, String, NotUsed] =
  Framing.delimiter(ByteString("\n"), maxLine)
    .map(_.decodeString("UTF8"))
```

Вернет Flow[ByteString, String, NotUsed]

Декодирует каждую выделенную строку ByteString в строку String с событием

Класс `Flow` имеет много методов, которые поддерживаются коллекциями, таких как `map` и `filter`. Их можно использовать для преобразования эле-

ментов в потоке. В листинге 13.8 показано, как использовать метод `map` для преобразования строк `ByteString` в строки `String`.

Теперь мы готовы перейти к преобразованию записи в `case`-класс `Event`. Мы не будем приводить фактическую логику преобразования записи (вы найдете ее в репозитории GitHub), а просто покажем, как она используется для преобразования строки `String` в `Event` (листинг 13.9).

**ПОТОК – ЭТО НЕ КОЛЛЕКЦИЯ.** Многие операции с потоками похожи на операции с коллекциями, например `map`, `filter` и `collect`. Поэтому кто-то из вас мог бы подумать, что поток – это просто разновидность стандартных коллекций, хотя в действительности это не так. Самое важное отличие заключается в том, что размер потока данных неизвестен, тогда как почти для всех стандартных коллекций, таких как `List`, `Set` и `Map`, его всегда можно определить. Некоторые методы, свойственные коллекциям, отсутствуют в `Flow` просто потому, что нет никакой возможности выполнить обход всех элементов в потоке данных.

**Листинг 13.9.** Преобразование строк

```
val parse: Flow[String, Event, NotUsed] =
  Flow[String].map(LogStreamProcessor.parseLineEx)
    .collect { case Some(e) => e }
```

Парсинг строки с помощью метода `parseLineEx` объекта `LogStreamProcessor`, который возвращает `Option[Event]` или `None`, если передать ему пустую строку

Отбрасывает пустые строки и извлекает событие в ветке `Some`

`Flow[String]` создает экземпляр `Flow`, который принимает и возвращает элементы типа `String`.

В данном случае тип материализованного значения не важен – трудно представить хоть какой-то осмысленный тип, когда создается `Flow[String]`. Поэтому, чтобы показать, что тип материализованного значения не играет никакой роли и само это значение не должно использоваться, для него указан тип `NotUsed`. Поток `parse` принимает строки `String` и возвращает события `Event`.

Далее следует этап фильтрации.

**Листинг 13.10.** Фильтрация событий

```
val filter: Flow[Event, Event, NotUsed] =
  Flow[Event].filter(_.state == filterState)
```

Все события с типом, соответствующим `filterState`, пропускаются на выход фильтра, а остальные отбрасываются.

Следующий этап – этап сериализации.

**Листинг 13.11.** Сериализация событий

```
val serialize: Flow[Event, ByteString, NotUsed] =
  Flow[Event].map(event => ByteString(event.toJson.compactPrint))
```

Сериализация в  
формат JSON с  
помощью библиотеки  
spray-json

Потоки можно комбинировать с помощью метода `via`. В листинге 13.12 представлено полное определение фильтра событий и способ его материализации.

**Листинг 13.12.** Объединение компонентов фильтра

```
val composedFlow: Flow[ByteString, ByteString, NotUsed] =
  frame.via(parse)
    .via(filter)
    .via(serialize)

val runnableGraph: RunnableGraph[Future[IOResult]] =
  source.via(composedFlow).toMat(sink)(Keep.right)

runnableGraph.run().foreach { result =>
  println(s"Wrote ${result.count} bytes to '$outputFile'.")
  system.terminate()
}
```

Здесь мы использовали метод `toMat`, чтобы сохранить материализованное значение справа, то есть материализованное значение из приемника `Sink`, благодаря чему можем сообщить количество байтов, записанных в выходной файл. Конечно, логику обработки можно объявить в едином потоке, как показано в листинге 13.13.

**Листинг 13.13.** Единый поток фильтра событий

```
val flow: Flow[ByteString, ByteString, NotUsed] =
  Framing.delimiter(ByteString("\n"), maxLine)
    .map(_.decodeString("UTF8"))
    .map(LogStreamProcessor.parseLineEx)
    .collect { case Some(e) => e }
    .filter(_.state == filterState)
    .map(event => ByteString(event.toJson.compactPrint))
```

В следующем разделе мы рассмотрим приемы обработки ошибок, например когда входной файл содержит поврежденную запись.

### 13.1.4. Обработка ошибок в потоках

Приложение `EventFilter` оказывается слишком простым, когда дело доходит до ошибок. Метод `LogStreamProcessor.parseLineEx` возбуждает

исключение, когда оказывается не в состоянии выполнить парсинг строки, но это лишь одна из множества ошибок, которые могут возникнуть. Приложение может также получить путь к несуществующему файлу.

По умолчанию, когда возникает исключение, обработка потока прекращается. В качестве материализованного значения граф вернет объект `Future` с исключением. Это не очень удобно в данном случае. Предпочтительнее было бы просто игнорировать поврежденные строки.

Рассмотрим сначала пример игнорирования строк, не поддающихся парсингу. По аналогии с актерами мы можем определить стратегию наблюдения. В листинге 13.14 показано, как можно использовать `Resume`, чтобы просто отбросить элемент, вызвавший исключение, благодаря чему обработка потока не останавливается.

#### Листинг 13.14. Возобновление обработки потока в случае исключения `LogParseException`

```
import akka.stream.ActorAttributes
import akka.stream.Supervision

import LogStreamProcessor.LogParseException

val decider : Supervision.Decider = {
  case _: LogParseException => Supervision.Resume
  case _                     => Supervision.Stop
}

val parse: Flow[String, Event, NotUsed] =
  Flow[String].map(LogStreamProcessor.parseLineEx)
    .collect { case Some(e) => e }
    .withAttributes(ActorAttributes.supervisionStrategy(decider))
```

↑ Определение стратегии наблюдения по аналогии с актерами

← Возобновляет выполнение в случае исключения `LogParseException`

↑ Передача супервизора в атрибутах

Стратегия наблюдения передается вызовом метода `withAttributes`, доступного во всех компонентах графа. Стратегию можно также определить для всего графа целиком, с помощью `ActorMaterializerSettings`, как показано в листинге 13.15.

#### Листинг 13.15. Настройка стратегии наблюдения за графом

```
val graphDecider : Supervision.Decider = {
  case _: LogParseException => Supervision.Resume
  case _                     => Supervision.Stop
}
```

```
import akka.stream.ActorMaterializerSettings
implicit val materializer = ActorMaterializer(
```

```
ActorMaterializerSettings(system)
  .withSupervisionStrategy(graphDecider)
)
```

← Передача супервизора через ActorMaterializerSettings

Для потоков поддерживаются стратегии Resume, Stop и Restart. Некоторые операции с потоками конструируют состояние, которое отбрасывается при использовании стратегии Restart; стратегия Resume не отбрасывает созданного состояния.

**ОШИБКИ КАК ЭЛЕМЕНТЫ ПОТОКА.** Другой способ обработки ошибок – обработка исключений и передача ошибок в поток наряду с остальными элементами. Можно, например, объявить case-класс UnparsableEvent и сделать оба класса – Event и UnparsableEvent – наследниками общего запечатанного трейта Result, обеспечив возможность их определения с помощью операции сопоставления. В этом случае объявление потока будет иметь вид: Flow[ByteString, Result, NotUsed]. Также можно использовать тип Either и кодировать ошибки, как материализованные значения слева, а события – справа, например: Flow[ByteString, Either[Error, Result], NotUsed]. В сообществе имеются более удачные альтернативы, чем Either, такие как Disjunction в Scalaz, Xor в Cats или Or в Scalactic. Отображение Either-подобного типа слева мы оставляем читателям в качестве самостоятельного упражнения.

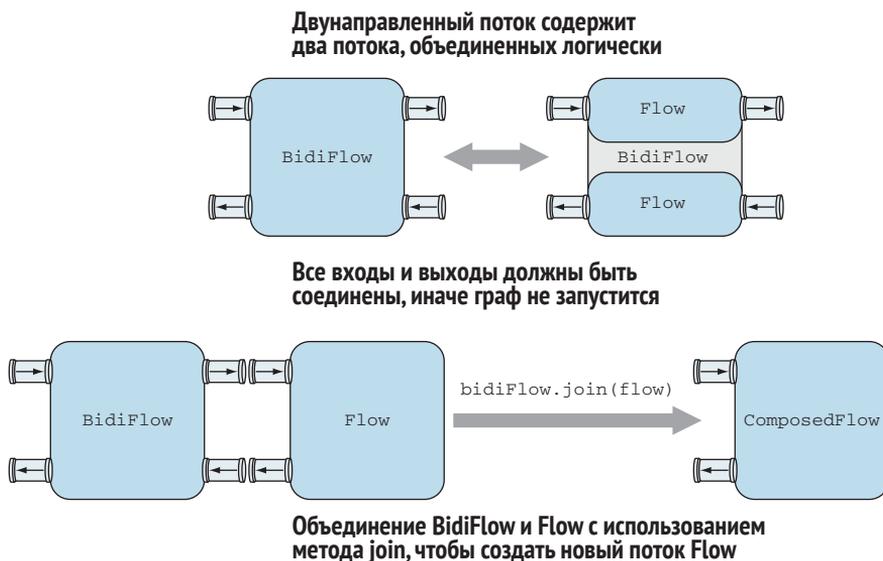
Теперь, когда мы кратко обсудили обработку ошибок в потоке, перейдем к рассмотрению приемов отделения протокола сериализации от логики фильтрации событий. EventFilter – очень простое приложение, суть которого заключается в фильтрации событий определенного типа. Нам очень пригодилась бы возможность повторного использования этапов парсинга, фильтрации и сериализации. Кроме того, первоначально мы произвольно решили поддерживать определенный формат журналов на входе и формат JSON на выходе. Нам также пригодилась бы, например, поддержка формата JSON на входе и текстового формата журналов на выходе. В следующем разделе мы рассмотрим двунаправленный поток, позволяющий определить многократно используемый протокол сериализации, который можно приложить к потоку фильтрации.

### 13.1.5. Создание протокола с BidiFlow

BidiFlow – это компонент графа с двумя открытыми входами и выходами. Одно из применений BidiFlow – использование в качестве адаптера.

Мы будем использовать BidiFlow как два парных потока, но вообще экземпляры BidiFlow можно создавать многими разными способами, а не только на основе двух потоков, что открывает довольно интересные перспективы.

А теперь перепишем приложение `EventFilter` так, чтобы оно состояло главным образом лишь из метода `filter` типа `Flow[Event, Event, NotUsed]`, выполняющего фильтрацию событий. Операции чтения событий из входного потока байтов и записи в выходной поток будут реализованы как сменные адаптеры протокола. Структура `BidiFlow` показана на рис. 13.11.



**Рис. 13.11.** Двунаправленный поток

В приложении `BidiEventFilter` протокол сериализации отделен от логики фильтрации событий, как показано на рис. 13.12. В данном случае «выходной» поток содержит только сериализованный поток, потому что разделители (символы перевода строки) автоматически добавляются механизмом сериализации.

В листинге 13.16 показано, как создается конкретный `BidiFlow` на основе аргументов командной строки. Любые параметры, кроме `"json"`, интерпретируются как формат файла журнала.

#### Листинг 13.16. Создание `BidiFlow` на основе аргументов командной строки

```
val inFlow: Flow[ByteString, Event, NotUsed] =
  if(args(0).toLowerCase == "json") {
    JsonFraming.json(maxJsonObject)
      .map(_.decodeString("UTF8").parseJson.convertTo[Event])
  } else {
    Framing.delimiter(ByteString("\n"), maxLine)
      .map(_.decodeString("UTF8"))
      .map(LogStreamProcessor.parseLineEx)
      .collect { case Some(event) => event }
  }
```

← Выделение записей для потока данных в формате JSON; `maxJsonObject` – это максимальное количество байтов для любого `JsonObject`

```

}

val outFlow: Flow[Event, ByteString, NotUsed] =
  if(args(1).toLowerCase == "json") {
    Flow[Event].map(event => ByteString(event.toJson.compactPrint))
  } else {
    Flow[Event].map{ event =>
      ByteString(LogStreamProcessor.logLine(event))
    }
  }
}

```

← Метод `LogStreamProcessor.logLine` сериализует событие в текстовую строку в формате журнала

```

val bidiFlow = BidiFlow.fromFlows(inFlow, outFlow)

```

`JsonFraming` разбивает входящую последовательность байтов на объекты JSON. Для парсинга байтов в объект JSON и последующего его преобразования в событие `Event` здесь используется библиотека `spray-json`. В состав проекта в репозитории GitHub включено определение `JsonFraming`, взятое из рукописи статьи Конрада Малавски (Konrad Malawski) о маршаллерах для обработки потоковых данных в формате JSON (которые, как ожидается, будут включены в одну из ближайших версий Akka).

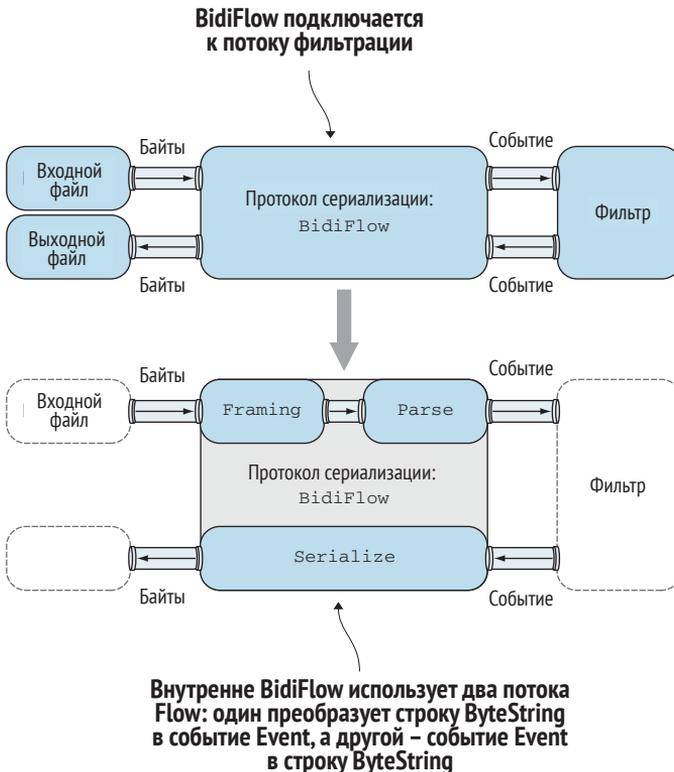


Рис. 13.12. Протокол сериализации на основе двунаправленного потока

Метод `fromFlows` создает `BidiFlow` из двух потоков, выполняющих десериализацию и сериализацию. `BidiFlow` можно подключить к потоку `filter` вызовом `join`, как показано в листинге 13.17.

**Листинг 13.17.** Подключение `BidiFlow` к потоку фильтрации

```
val filter: Flow[Event, Event, NotUsed] =
  Flow[Event].filter(_.state == filterState)
val flow = bidiFlow.join(filter)
```

Входящий поток в `BidiFlow` – это поток слева от потока фильтрации; исходящий поток в `BidiFlow` – справа

Также потоки в `BidiFlow` можно подключать до и после существующего потока для преобразования входных данных перед передачей в рассматриваемый поток и выходных данных, получаемых из него. В этом случае он используется для чтения и записи в согласованном формате.

В следующем разделе мы сконструируем потокковую HTTP-службу, добавим в обработчик потока событий ряд дополнительных возможностей и получим результат, более близкий к практическому приложению. До сих пор мы использовали линейные конвейеры потоковых операций, а теперь дополнительно рассмотрим распределение и объединение потоков данных.

## 13.2. Потокковая передача данных через HTTP

Обработчик потока событий должен действовать как HTTP-служба. Давайте посмотрим, что для этого необходимо сделать. Модуль `akka-http` использует `akka-stream`, поэтому нам не придется писать много кода для преобразования приложения, обрабатывающего файлы, в HTTP-службу. Модуль `akka-http` – отличный пример библиотеки, обертывающей `akka-stream`. В этом вы вскоре убедитесь сами.

Для начала добавим в проект необходимые зависимости.

**Листинг 13.18.** Зависимости для `akka-http`

```
"com.typesafe.akka" %% "akka-http-core" % version,
"com.typesafe.akka" %% "akka-http-experimental" % version,
"com.typesafe.akka" %% "akka-http-spray-json-experimental" % version,
```

Зависимость для подключения `akka-http`

Для интеграции `akka-http` с `spray-json`

На этот раз мы создадим приложение `LogApp`, позволяющее передавать журналы из хранилища и в хранилище. В данном случае, ради простоты, потоки будут записываться непосредственно в файлы.

Существует довольно много реактивных клиентских библиотек для обработки потоковых данных. Соединение примера с некоторыми другими

видами хранилищ (базами данных) мы оставляем читателям в качестве самостоятельного упражнения.

### 13.2.1. Прием потока данных по HTTP

Мы разрешим клиентам службы посылать поток событий с использованием HTTP-метода `POST`. Данные будут сохраняться в файле на стороне сервера. Адрес URL будет иметь вид: `/logs/[log_id]`, где `[log_id]` – это имя файла в каталоге `logs`. Например, для URL `/logs/1` будет создан файл с именем `1` в каталоге `logs`. Позднее мы будем возвращать поток данных из этого файла при получении HTTP-запроса `GET` с URL `/logs/[log-id]`. Здесь мы опустим часть приложения `LogsApp`, выполняющую настройку HTTP-сервера.

Маршрут HTTP определяется в классе `LogsApi`, как показано в листинге 13.19. Класс `LogsApi` имеет поле `logsDir`, определяющее путь к каталогу, где будут храниться файлы. Метод `logFile` просто возвращает `File` для заданного идентификатора `log-id`. Трейт `EventMarshalling` реализует поддержку преобразования в формат JSON. Отметьте также, что `ExecutionContext` и `ActorMaterializer` находятся в неявной области видимости; они необходимы для запуска потоков `Flow`.

**Листинг 13.19.** `LogsApi`

```
class LogsApi(
  val logsDir: Path,
  val maxLine: Int
){
  implicit val executionContext: ExecutionContext,
  val materializer: ActorMaterializer
} extends EventMarshalling {
  def logFile(id: String) = logsDir.resolve(id)
  // далее следует логика обработки маршрутов...
```

Мы повторно используем `BidiFlow` из предыдущего раздела, потому что этот компонент уже определяет протокол преобразования записей из файла журнала в события JSON. В листинге 13.20 показаны реализации `Flow` и `Sink` для обработки входящего потока данных, а также `Source`, используемого для обработки HTTP-запросов `GET`.

**Листинг 13.20.** `Flow` и `Sink` для обработки запросов `POST`

```
import java.nio.file.StandardOpenOption
import java.nio.file.StandardOpenOption._

val logToJsonFlow = bidiFlow.join(Flow[Event])
def logFileSink(logId: String) =
```

← Двухнаправленный поток подключается к входному потоку и передает события без изменений

```
FileIO.toPath(logFile(logId), Set(CREATE, WRITE, APPEND))
def logFileSource(logId: String) = FileIO.fromPath(logFile(logId))
```

В этом примере события не изменяются; все строки просто преобразуются в события JSON. Реализацию потока, подключаемого к `VidiFlow` для фильтрации событий на основе параметров запроса, мы оставляем читателям. `logFileSink` и `logFileSource` – это вспомогательные методы, вы увидите их в примерах ниже.

### Полностью прочитайте запрос перед ответом

Очень важно полностью прочитать все данные из `dataBytes`. Если ответить раньше, чем будут прочитаны все данные, клиент, поддерживающий *постоянное HTTP-соединение*, например, может использовать TCP-сокеты для отправки следующего запроса, вследствие чего данные в конечной точке `Source` будут потеряны.

В большинстве случаев HTTP-клиенты полагают, что сервер не ответит, пока не обработает запрос полностью, поэтому не будут пытаться читать ответ, не закончив отправку. Даже если вы не используете постоянные соединения, все равно лучше обработать запрос полностью.

Обычно это вопрос блокировки HTTP-клиентов, которые не приступают к чтению ответа, не закончив отправку запроса.

Это не означает, что цикл запрос/ответ действует синхронно. В примерах в этом разделе ответ посылается обратно асинхронно, после обработки запроса.

HTTP-запрос `POST` обрабатывается методом `postRoute`, представленным в листинге 13.21. Так как модуль `akka-http` внутренне использует `akka-stream`, организация приема потока данных через HTTP – относительно простая задача. HTTP-запрос имеет конечную точку `Source` с именем `dataBytes` для чтения данных.

#### Листинг 13.21. Обработка запроса POST

```
def postRoute =
  pathPrefix("logs" / Segment) { logId =>
    pathEndOrSingleSlash {
      post {
        entity(as[HttpEntity]) { entity =>
          onComplete(
            entity
              .dataBytes
              .via(logToJsonFlow)
          )
        }
      }
    }
  }
```

Извлечение `HttpRequest`

Поток данных в этой сущности имеет тип `Source[ByteString, Any]`

Протокол потока: запись из журнала на входе, JSON – на выходе



```

    }
  }
}

```

HttpEntity имеет метод `apply`, который принимает экземпляры `Content-Type` и `Source`. Чтобы осуществить потокковую передачу данных клиенту, достаточно передать `Source` этому методу и завершить ответ директивой `complete`. В примере обработки запроса `POST` мы просто предположили, что данные будут поступать на сервер в простом текстовом виде, в формате записей из файла журнала. В примере обработки запроса `GET` мы возвращаем данные в формате `JSON`.

**ОБРАТНЫЕ АПОСТРОФЫ В ИДЕНТИФИКАТОРАХ.** Модуль `akka-http` стремится оставаться как можно ближе к спецификации `HTTP`, и это обстоятельство находит свое отражение в именах `HTTP`-заголовков, названиях типов содержимого и других элементах `HTTP`. В `Scala` допускается использовать в идентификаторах иначе недопустимые символы, такие как дефисы и слешы, которые широко применяются в спецификации `HTTP`, если заключить их в обратные апострофы для окружения.

Теперь, реализовав простую потокковую передачу данных по запросам `GET` и `POST`, давайте посмотрим, как с помощью `akka-http` можно организовать *согласование контента*, чтобы позволить клиенту запрашивать и посылать данные в формате журналируемых записей или `JSON`.

### 13.2.3. Согласование контента

Заголовок `Accept` позволяет `HTTP`-клиенту указать формат данных, возвращаемых в ответ на запрос `GET`, если такая возможность поддерживается. Также `HTTP`-клиент может установить заголовок `Content-Type`, чтобы сообщить серверу, в каком формате передаются данные в запросе `POST`. В этом разделе мы рассмотрим оба случая, обеспечив возможность отправки/приема данных в любом формате – `JSON` или в формате журнала, как это было сделано в примере `VidiEventFilter`.

К счастью, `akka-http` включает средства для организации преобразований входных и выходных данных, автоматически решая задачу согласования контента, а это значит, что нам останется меньше работы. Начнем с обработки заголовка `Content-Type` в запросе `POST`.

#### Обработка заголовка `Content-Type`

Модуль `akka-http` поддерживает множество predefined типов для преобразования входящих данных из массивов байтов, строк и т. д. Он также дает возможность подставлять свои реализации `Unmarshaller` для

преобразования. В этом примере мы будем поддерживать только два типа контента: `text/plain` – для формата журнала и `application/json` – для событий в формате JSON. Исходя из значений `Content-Type`, конечная точка `entity.dataBytes` будет разграничивать строки по символу перевода строки или интерпретировать данные как поток объектов JSON.

Трейт `Unmarshaller` требует реализовать только один метод.

### Листинг 13.23. Обработка `Content-Type` в `EventUnmarshaller`

```
import akka.http.scaladsl.unmarshalling.Unmarshaller
import akka.http.scaladsl.unmarshalling.Unmarshaller._

object EventUnmarshaller extends EventMarshalling {
  val supported = Set[ContentTypeRange](
    ContentTypeRanges.`text/plain(UTF-8)`,
    ContentTypeRanges.`application/json`
  )

  def create(maxLine: Int, maxJsonObject: Int) = {
    new Unmarshaller[HttpEntity, Source[Event, _]] {
      def apply(entity: HttpEntity)(implicit ec: ExecutionContext,
        materializer: Materializer): Future[Source[Event, _]] = {

        val future = entity.contentType match {
          case ContentTypeRanges.`text/plain(UTF-8)` =>
            Future.successful(LogJson.textInFlow(maxLine))
          case ContentTypeRanges.`application/json` =>
            Future.successful(LogJson.jsonInFlow(maxJsonObject))
          case other =>
            Future.failed(
              new UnsupportedContentTypeException(supported)
            )
        }
        future.map(flow => entity.dataBytes.via(flow))(ec)
      }
    }.forContentTypes(supported.toList:_* )
  }
}
```

Множество поддерживаемых типов контента

apply превратит entity в объект Future, возвращающий конечную точку – источник событий

Своя реализация Unmarshaller

Определение типа содержимого, заключение Flow в Future

Перемещение потоков для форматов в объект LogJson

Возбудит исключение, если тип контента не поддерживается

Создание нового источника вызовом dataBytes.via

Ограничивает допустимые типы контента для поведения модуля akka-http по умолчанию

Метод `create` создает анонимный экземпляр `Unmarshaller`. Его метод `apply` создает поток `Flow` для обработки входящих данных, который конструируется вызовом `dataBytes.via`.

Эту реализацию `Unmarshaller` следует поместить в неявную область видимости, чтобы директива `entity` могла использовать ее для извлечения `Source[Event, _]` из класса `ContentNegLogApi`.

**Листинг 13.24.** Использование EventUnmarshaller в обработке запроса POST

```
implicit val unmarshaller = EventUnmarshaller.create(maxLine, maxJsonObject)
def postRoute =
  pathPrefix("logs" / Segment) { logId =>
    pathEndOrSingleSlash {
      post {
        entity(as[Source[Event, _]]) { src =>
          onComplete(
            src.via(outFlow)
              .toMat(logFileSink(logId))(Keep.right)
              .run()
          ) {
            // Обработка результата Future здесь опущена,
            // она выполняется точно так же, как прежде.
```

← Создает Unmarshaller и поместит его в неявную область видимости

← entity(as[T]) требует присутствия Unmarshaller в неявной области видимости

Опробование `aia.stream.ContentNegLogsApp` мы оставляем читателям в качестве самостоятельного упражнения. Не забудьте указать заголовок `Content-Type`, используя `httpie`. В листинге 13.25 приводится несколько примеров.

**Листинг 13.25.** Примеры использования утилиты `httpie` для отправки запросов POST с заголовком `Content-Type`

```
http -v POST localhost:5000/logs/1 Content-Type:text/plain < test.log
http -v POST localhost:5000/logs/2 Content-Type:application/json < test.json
```

В следующем разделе мы посмотрим, как обработать заголовок `Accept` для согласования контента.

**Обработка заголовка `Accept`**

Напишем свою реализацию трейта `Marshaller` для поддержки типов контента `text/plain` и `application/json` при подготовке ответов на стороне сервера. Клиент может послать серверу заголовок `Accept`, чтобы перечислить типы контента, которые он может принять. В листинге 13.26 приводится несколько примеров отправки таких запросов с помощью `httpie`.

**Листинг 13.26.** Примеры использования утилиты `httpie` для отправки запросов GET с заголовком `Accept`

```
http -v GET localhost:5000/logs/1 'Accept:application/json'
http -v GET localhost:5000/logs/1 'Accept:text/plain'
http -v GET localhost:5000/logs/1 \
'Accept: text/html, text/plain;q=0.8, application/json;q=0.5'
```

← Принимаются ответы только в формате JSON

← Предпочтительнее формат `text/html`, менее предпочтителен `text/plain`; еще менее предпочтителен JSON

← Принимаются ответы только в текстовом формате (формат журнала)

Клиент может заявить, что принимает контент только определенного типа или что имеет определенные предпочтения. В Akka уже имеется логика, определяющая, какой тип содержимого надлежит отправить в ответе. Нам остается только создать `Marshaller`, поддерживающий множество типов.

Объект `LogEntityMarshaller` создает `ToEntityMarshaller`.

### Листинг 13.27. Реализация преобразований для согласования типа контента

```
import akka.http.scaladsl.marshalling.Marshaller
import akka.http.scaladsl.marshalling.ToEntityMarshaller

object LogEntityMarshaller extends EventMarshalling {

  type LEM = ToEntityMarshaller[Source[ByteString, _]]
  def create(maxJsonObject: Int): LEM = {
    val js = ContentTypes.`application/json`
    val txt = ContentTypes.`text/plain(UTF-8)`

    val jsMarshaller = Marshaller.withFixedContentType(js) {
      src: Source[ByteString, _] =>
        HttpEntity(js, src)
    }

    val txtMarshaller = Marshaller.withFixedContentType(txt) {
      src: Source[ByteString, _] =>
        HttpEntity(txt, toText(src, maxJsonObject))
    }

    Marshaller.oneOf(jsMarshaller, txtMarshaller)
  }

  def toText(src: Source[ByteString, _],
             maxJsonObject: Int): Source[ByteString, _] = {
    src.via(LogJson.jsonToLogFlow(maxJsonObject))
  }
}
```

Файл хранит данные в формате JSON, поэтому его содержимое возвращается непосредственно

Содержимое файла необходимо преобразовать обратно в формат журнала

oneOf создает из двух маршаллеров «супермаршаллер»

Переместить потоки для форматов в объект LogJson

`Marshaller.withFixedContentType` – это вспомогательный метод, создающий экземпляр `Marshaller` для конкретного типа контента. Он принимает функцию с сигнатурой `A => B`, которая в данном случае имеет вид `Source[ByteString, Any] => HttpEntity`. Объект `src` открывает доступ к байтам в файле JSON, которые преобразуются в `HttpEntity`.

Метод `LogJson.jsonToLogFlow` использует тот же трюк, что применялся выше, соединяя `VidiFlow` с `Flow[Event]`, на этот раз для преобразования из формата JSON в формат журнала.

Этот экземпляр `Marshaller` следует поместить в неявную область видимости, чтобы его мог использовать маршрут обработки HTTP-запросов GET.

### Листинг 13.28. Использование `LogEntityMarshaller` в обработке запроса GET

```
implicit val marshaller = LogEntityMarshaller.create(maxJsObject)
def getRoute =
  pathPrefix("logs" / Segment) { logId =>
    pathEndOrSingleSlash {
      get {
        extractRequest { req =>
          if(Files.exists(logFile(logId))) {
            val src = logFileSource(logId)
            complete(Marshal(src).toResponseFor(req))
          } else {
            complete(StatusCodes.NotFound)
          }
        }
      }
    }
  }
}
```

Создает маршаллер и помещает его в неявную область видимости

Директива `extractRequest` извлекает запрос

`toResponseFor` использует неявный маршаллер

`Marshal(src).toResponseFor(req)` принимает `Source` файла журнала и создает ответ, опираясь на запрос (в том числе и на заголовок `Accept`), то есть учитывая результат согласования контента с использованием `LogEntityMarshaller`.

На этом мы завершаем изучение примеров поддержки обоих форматов с использованием заголовка `Content-Type` и процедуры согласования контента на основе заголовка `Accept`.

Оба приложения, `LogsApi` и `ContentNegLogsApp`, читают и сохраняют события без изменений. Мы можем фильтровать события по типу (`OK`, `warning`, `error`, `critical`) в процессе подготовки ответа на запрос GET, но разумнее разделить их и хранить в отдельных файлах, чтобы, например, можно было извлечь все ошибки, не выполняя фильтрацию при каждой попытке чтения. В следующем разделе мы посмотрим, как с помощью `akka-stream` можно организовать ветвление и слияние. Мы организуем сохранение разных событий в разных файлах на сервере, а также дадим возможность извлекать события нескольких типов одновременно, например все события, кроме `OK`.

**ПОДДЕРЖКА ПОТОКОВОЙ ОБРАБОТКИ JSON.** Пример, представленный здесь, поддерживает обработку событий в двух форматах – формате журнала и JSON. Для случая, когда достаточно только поддержки формата

JSON, есть более простое решение. Объект `EntityStreamingSupport` в пакете `akka.http.scaladsl.common` предоставляет объект `JsonEntityStreamingSupport` через `EntityStreamingSupport.json`, который можно поместить в неявную область видимости и обрабатывать HTTP-запрос со списком событий непосредственно, используя `complete(events)`. Этот подход позволяет также получить `Source[Event, NotUsed]` непосредственно из `entity(asSourceOf[Event])`.

## 13.3. Ветвление и слияние со специализированным языком описания графов

До настоящего момента мы рассматривали только линейную обработку с одним входом и одним выходом. Модуль `akka-stream` поддерживает специализированный предметно-ориентированный язык описания графов (`Domain-Specific Language, DSL`), позволяющий определить точки слияния и ветвления, которые могут иметь произвольное количество входов и выходов. Код на языке DSL описания графов чем-то напоминает изображения, создаваемые символами ASCII, – во многих случаях диаграмму графа, нарисованную на доске, можно преобразовать в код на DSL.

Существует множество разновидностей `GraphStage`, которые можно использовать для создания самых разных графов, подобно `Source`, `Flow` и `Sink`. Также есть возможность определить свою версию `GraphStage`.

На языке описания графов можно определить граф любой формы `Shape`. В `akka-stream` форма `Shape` определяет количество входов и выходов графа (`Inlet` и `Outlet`). В следующем примере мы создадим граф в форме `Flow`, чтобы его можно было использовать в маршруте обработки запросов `POST`, как и прежде. Внутренне он будет иметь форму ветвления.

### 13.3.1. Ветвление потоков

В нашем следующем примере мы используем язык описания графов, чтобы разделить события по их типам (один приемник `Sink` будет получать все ошибки, другой – все предупреждения и т. д.), чтобы не пришлось каждый раз выполнять фильтрацию при получении запроса `GET` с параметром, требующим вернуть события одного из этих типов. На рис. 13.13 показано, как использовать `BroadcastGraphStage` для передачи событий в разные потоки `Flow`. Язык описания графов включает метод `GraphDSL.Builder` для создания узлов графа, а также `~>` для соединения узлов подобно методу `via`. Узлы в графе имеют тип `Graph`, что может показаться странным, потому что узел – это лишь часть графа, поэтому далее мы будем использовать термин «узел» вместо имени типа.

В листинге 13.29 приводится определение графа, изображенного на рис. 13.13. Здесь также можно видеть, как определяется поток, соединяющий вход и выход графа.

**Листинг 13.29.** Ветвление для разделения событий по типам

```
import akka.stream.{ FlowShape, Graph }
import akka.stream.scaladsl.{ Broadcast, GraphDSL, RunnableGraph }

type FlowLike = Graph[FlowShape[Event, ByteString], NotUsed]

def processStates(logId: String): FlowLike = {
  val jsFlow = LogJson.jsonOutFlow
  Flow.fromGraph(
    GraphDSL.create() { implicit builder =>
      import GraphDSL.Implicits._
      // все события, ok, warning, error, critical, то есть всего 5 выходов
      val bcast = builder.add(Broadcast[Event](5))
      val js = builder.add(jsFlow)

      val ok = Flow[Event].filter(_.state == Ok)
      val warning = Flow[Event].filter(_.state == Warning)
      val error = Flow[Event].filter(_.state == Error)
      val critical = Flow[Event].filter(_.state == Critical)

      bcast ~> js.in
      bcast ~> ok ~> jsFlow ~> logFileSink(logId, Ok)
      bcast ~> warning ~> jsFlow ~> logFileSink(logId, Warning)
      bcast ~> error ~> jsFlow ~> logFileSink(logId, Error)
      bcast ~> critical ~> jsFlow ~> logFileSink(logId, Critical)

      FlowShape(bcast.in, js.out)
    }
  )
}

def logFileSource(logId: String, state: State) =
  FileIO.fromPath(logStateFile(logId, state))
def logFileSink(logId: String, state: State) =
  FileIO.toPath(logStateFile(logId, state), Set(CREATE, WRITE, APPEND))
def logStateFile(logId: String, state: State) =
  logFile(s"$logId-${State.norm(state)}")
```

Создается поток из экземпляра Graph для использования в маршруте POST

builder – это GraphDSL.Builder

Импорт методов DSL в текущую область видимости

Добавляет узел Broadcast в Graph

Добавляет в Graph узел Flow для передачи всех событий в формате JSON без изменений

Один из выходов Broadcast выполняет запись всех событий непосредственно во вход узла js

Для всех других выходов перед потоком Flow событий в формате JSON добавляются фильтры

Создается Graph в форме потока Flow из входа Broadcast и выхода потока JSON Flow

Аргумент `builder` – это изменяемый экземпляр `GraphDSL.Builder`. В этом примере он используется только в анонимной функции для настройки графа. Метод `GraphDSL.Builder.add` возвращает экземпляр `Shape`, описывающий входы и выходы экземпляра `Graph`.



Метод `processStates` используется в полном соответствии с ожиданиями, как обычный поток.

### Листинг 13.30. Использование `processStates` в маршруте POST

```
src.via(processStates(logId))
  .toMat(logFileSink(logId))(Keep.right)
  .run()
```

Маршрут для запросов GET, возвращающий только ошибки, очень похож на обычный маршрут GET, с той лишь разницей, что в соответствии с соглашением об именовании читает содержимое файла `[log-id]-error`.

В следующем разделе мы рассмотрим объединение нескольких источников для возврата в одном потоке событий всех типов, а также событий всех типов, кроме OK.

## 13.3.2. Слияние потоков

Рассмотрим граф, объединяющий несколько источников. В первом примере мы объединим события всех типов, кроме OK. В ответ на запрос GET по адресу `/logs/[log-id]/not-ok` будут возвращаться все события, кроме OK. На рис. 13.14 показано, как использовать `MergeGraphStage` для объединения трех источников `Source` в один.

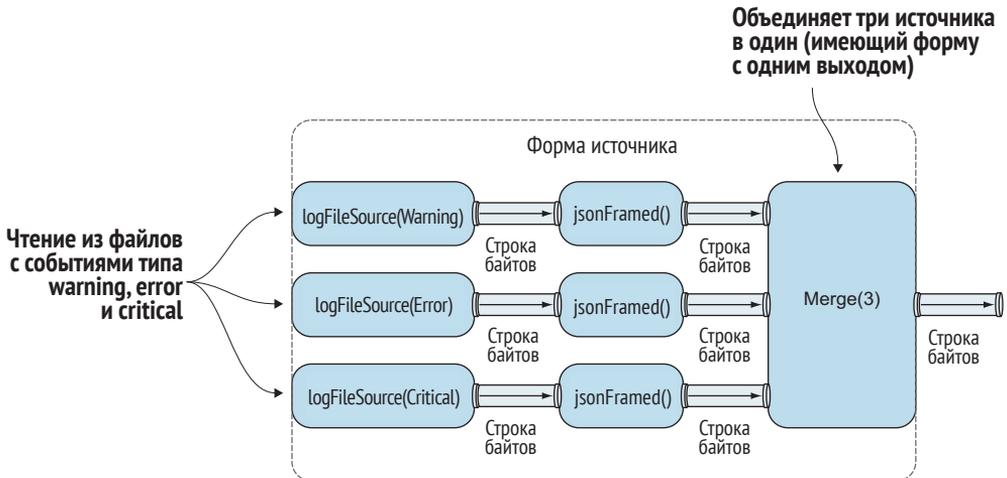


Рис. 13.14. Объединение событий всех типов, кроме OK, с помощью `MergeGraphStage`

В листинге 13.31 показан порядок использования `MergeGraphStage` в коде на языке описания графов. Он определяет метод `mergeNotOk`, объединяющий все источники, кроме OK, с конкретным идентификатором `logId` в один источник `Source`.

**Листинг 13.31.** Объединение всех событий, кроме ОК

```

import akka.stream.SourceShape
import akka.stream.scaladsl.{ GraphDSL, Merge }

def mergeNotOk(logId: String): Source[ByteString, NotUsed] = {
  val warning = logFileSource(logId, Warning)
    .via(LogJson.jsonFramed(maxJsObject))
  val error = logFileSource(logId, Error)
    .via(LogJson.jsonFramed(maxJsObject))
  val critical = logFileSource(logId, Critical)
    .via(LogJson.jsonFramed(maxJsObject))

  Source.fromGraph(
    GraphDSL.create() { implicit builder =>
      import GraphDSL.Implicits._

      val warningShape = builder.add(warning)
      val errorShape = builder.add(error)
      val criticalShape = builder.add(critical)
      val merge = builder.add(Merge[ByteString](3))

      warningShape ~> merge
      errorShape ~> merge
      criticalShape ~> merge
      SourceShape(merge.out)
    })
}

```

На основе Graph создается источник, который используется в маршруте GET

Обратите внимание, что события типа `warning`, `error` и `critical` предварительно передаются через поток выделения объектов JSON, потому что иначе может получиться искаженное представление в виде `ByteString`.

Три источника объединяются с помощью `MergeGraphStage`, имеющего три входа. Граф имеет единственный выход (`merge.out`), из которого создается источник `SourceShape`. Этот источник имеет вспомогательный метод `fromGraph`, который преобразует `Graph` с `SourceShape` в `Source`.

Далее в `getLogNotOkRoute` с помощью метода `mergeNotOk` создается `Source` для чтения, как показано в листинге 13.32.

**MERGE PREFERRED GRAPH STAGE.** Граф `MergeGraphStage` случайно выбирает элементы из своих входов. В `akka-stream` также имеется граф `MergePreferredGraphStage`, который имеет один выходной порт `out`, один основной входной порт `preferred` и ноль или более вторичных входных портов `in`. Граф этого типа выводит элемент, когда на одном из его входов появляется входной элемент. Если входные элементы появляются сразу на нескольких входах, предпочтение отдается входу `preferred`.

**Листинг 13.32.** Формирование ответа на запрос GET с адресом /logs/[log-id]/not-ok

```
def getLogNotOkRoute =
  pathPrefix("logs" / Segment / "not-ok") { logId =>
    pathEndOrSingleSlash {
      get {
        extractRequest { req =>
          complete(Marshal(mergeNotOk(logId)).toResponseFor(req))
        }
      }
    }
  }
}
```

Есть также упрощенный API объединения источников, который мы используем для вывода всех событий. В ответ на запрос GET с адресом /logs должны возвращаться все события. В листинге 13.33 показано, как пользоваться этим упрощенным API.

**Листинг 13.33.** Метод mergeSources

```
import akka.stream.scaladsl.Merge

def mergeSources[E](
  sources: Vector[Source[E, _]]
): Option[Source[E, _]] = {
  if(sources.size == 0) None
  else if(sources.size == 1) Some(sources(0))
  else {
    Some(Source.combine(
      sources(0),
      sources(1),
      sources.drop(2) : _*
    ))(Merge(_))
  }
}
```

← Объединяет все источники в Vector

← Ничего не возвращать, если аргумент sources пуст

← Объединить указанные источники; первые два аргумента имеют тип Source, а третий – список переменной длины

← Merge передается как стратегия объединения

Метод `Source.combine` создает источник `Source` из заданного количества источников, подобно тому, как это делалось при использовании конструкций языка описания графов. Метод `mergeSources` объединяет источники одного типа. Например, `mergeSources` используется в маршруте /logs, как показано в листинге 13.34.

**Листинг 13.34.** Формирование ответа на запрос GET с адресом /logs

```
def getLogsRoute =
  pathPrefix("logs") {
    pathEndOrSingleSlash {
```

```

get {
  extractRequest { req =>
    val sources = getFileSources(logsDir).map { src =>
      src.via(LogJson.jsonFramed(maxJsObject))
    }
    mergeSources(sources) match {
      case Some(src) =>
        complete(Marshal(src).toResponseFor(req))
      case None =>
        complete(StatusCodes.NotFound)
    }
  }
}
}
}
}
}

```

`getFileSources`, здесь не показан, перечисляет файлы в `logsDir` и преобразует их в экземпляры `Source` с помощью `FileIO.fromPath`

Содержимое каждого исходного файла нужно пропустить через поток выделения объектов JSON

Объединяет все исходные файлы, найденные в каталоге `logsDir`

`BroadcastGraphStage`, показанный в этом разделе, использует механизм обратного давления для согласования скорости передачи информации со всеми выходами. Это означает, что распределять события можно со скоростью самого медленного потребителя. В следующем разделе мы обсудим, как использовать буферизацию, чтобы позволить производителям и потребителям работать с разными скоростями, и как осуществлять посредничество между ними.

**ПРЕДОПРЕДЕЛЕННЫЕ И НЕСТАНДАРТНЫЕ ГРАФЫ.** В `akka-stream` есть несколько предопределенных типов `GraphStages`, которые не были показаны здесь, в том числе с поддержкой балансировки нагрузки (`Balance`), с поддержкой архивирования (`Zip`, `ZipWith`) и с поддержкой конкатенации потоков (`Concat`). В коде на языке описания графов они используются точно так же, как было показано в предыдущих примерах. Во всех случаях вы должны добавить узлы в построитель (`builder`), связать входы и выходы формы (возвращается методом `add`) и вернуть некоторую форму из функции, которую затем нужно передать в вызов метода `Graph.create`. Также есть возможность написать свой класс, наследующий `GraphStage`, однако мы не будем рассматривать эту тему, так как она выходит далеко за рамки вводной главы в `akka-stream`.

## 13.4. Посредничество между производителями и потребителями

Далее мы рассмотрим пример рассылки событий службам-потребителям. До сих пор мы записывали события на диск – в один общий файл и в несколько файлов по типам событий. Переключение приемника `Sink` для пе-

передачи всех событий мы оставляем читателям в качестве самостоятельного упражнения.

В этой заключительной версии обработчика потока событий мы реализуем передачу событий службе архивирования, службе уведомления и службе показателей.

Обработчик потока событий должен балансировать скорость передачи событий в соответствии с требованиями внешних служб-потребителей и гарантировать, что обратное давление служб не приведет к замедлению работы производителя событий. В следующем разделе мы обсудим порядок применения буферизации с этой целью.

### Интеграция со службами

В примере, представленном в этом разделе, предполагается, что все службы предоставляют приемники Sink. Модуль akka-stream предлагает также несколько решений для интеграции с внешними службами, не связанных с использованием Source или Sink. Например, метод mapAsync принимает Future и возвращает результат в Future, что может пригодиться, если у вас уже есть клиент службы, использующий Future.

Также есть возможность интеграции с другими реализациями Reactive Streams посредством Source.fromPublisher и Sink.fromSubscriber, превращая любого издателя Reactive Streams в Source, а подписчика в Sink. Кроме того, интеграцию можно осуществить с помощью акторов, используя трейты ActorPublisher и ActorSubscriber, что может быть удобно в некоторых особых случаях.

Но лучший и самый простой путь – использовать библиотеку akka-stream-based, предоставляющую реализации Source и Sink.

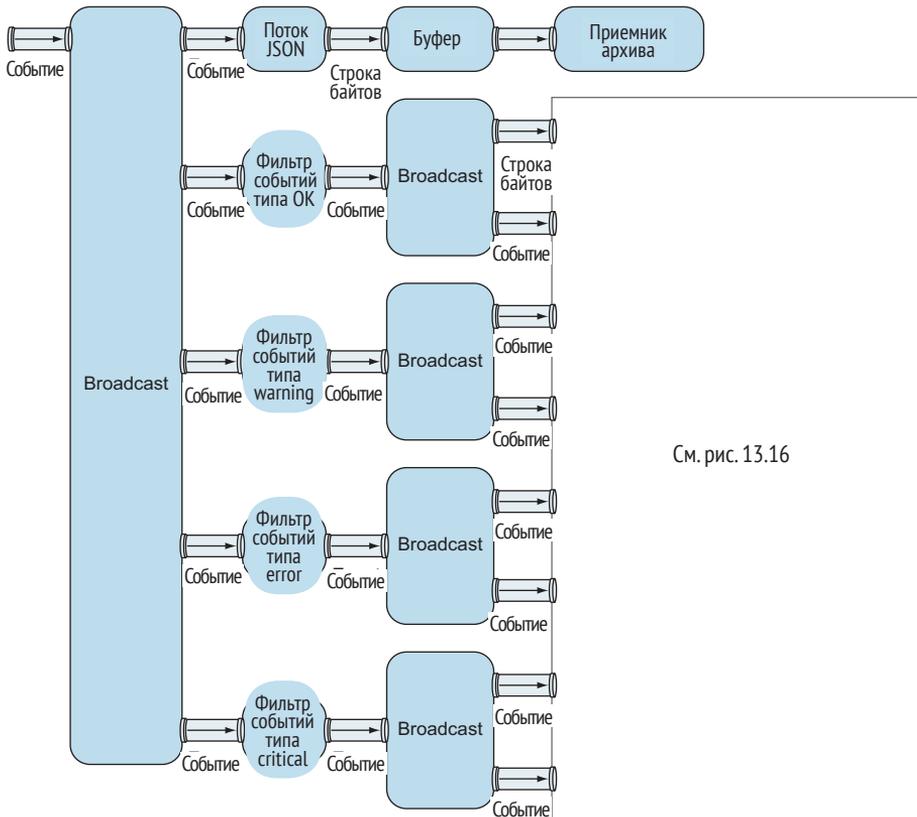
#### 13.4.1. Использование буферов

Рассмотрим граф обработки событий, теперь адаптированный для передачи данных трем службам-приемникам. В этом разделе мы подробно рассмотрим все компоненты графа, изображенные на рис. 13.15–13.16.

После каждого фильтра добавляется компонент Broadcast. Один выход выполняет запись в файловый приемник Sink, как обычно, а другой используется для отправки данных службам. (Здесь предполагается, что запись в файлы происходит быстро, поэтому записываемые события не буферизуются.) Этап MergePreferred объединяет предупреждения, ошибки и критические ошибки в блоки и записывает их в приемник Sink службы уведомлений. Для каждой критической ошибки создается свой отдельный блок и передается службе незамедлительно и вне очереди.

События типа ОК расщепляются на два потока с помощью Broadcast и пересылаются службе показателей.

На рисунке также видно, где осуществляется буферизация. Буферы позволяют работать с более медленными службами-потребителями. Но когда буфер оказывается заполненным, необходимо принять решение.



**Рис. 13.15.** Граф обработки журналируемых событий

Метод `buffer` потока `Flow` принимает два аргумента: размер буфера и стратегию – вариант перечисления `OverflowStrategy`, определяющий, что должно происходить при переполнении буфера. Перечисление `OverflowStrategy` содержит следующие варианты: `dropHead`, `dropTail`, `dropBuffer`, `dropNew`, `backpressure` или `fail`, чтобы, соответственно, выбросить первый элемент из буфера, последний элемент, сбросить весь буфер, выбросить новейший элемент из буфера, применить механизм обратного давления или остановить весь поток. Выбор варианта зависит от требований к приложению, что наиболее важно в данном конкретном случае.

В этом примере принято решение, что события обязательно должны архивироваться, даже при высокой нагрузке, то есть поток обработчика журналируемых событий должен завершаться с ошибкой, если оказывает-

ся не в состоянии выполнить запись в приемник архивной службы. В этом случае производитель сможет попробовать соединиться с ним позже. Для буфера выбирается очень большой размер, чтобы смягчить удар, если приемник архивной службы не будет справляться с нагрузкой в течение какого-то времени.

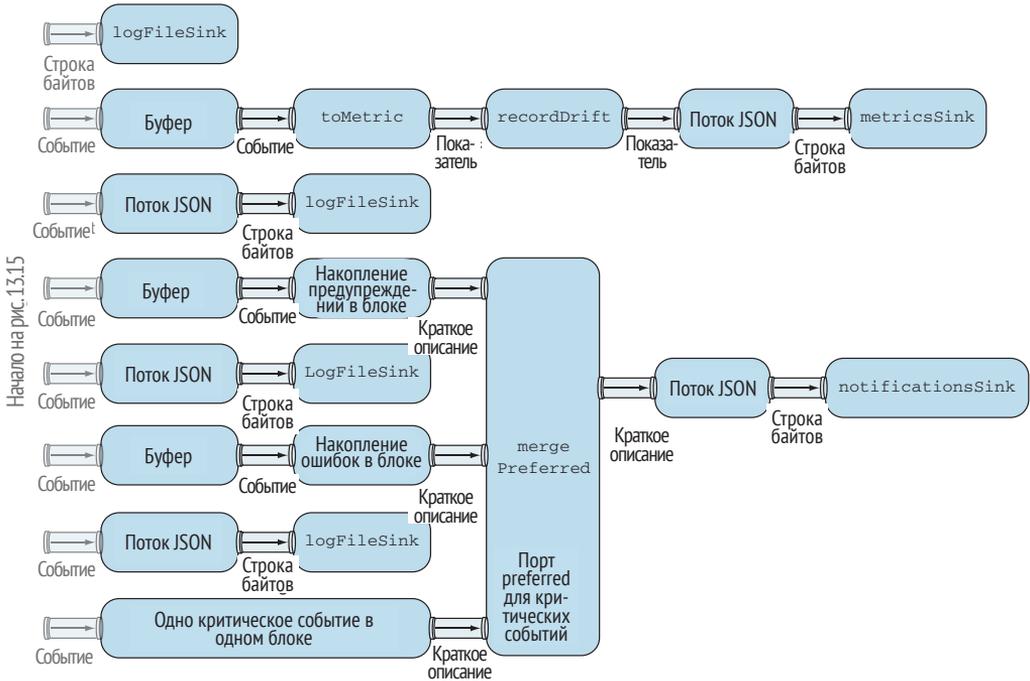


Рис. 13.16. Обработка событий типов OK, warning, error и critical

В листинге 13.35 показана настройка буферов в графе.

**Листинг 13.35.** Буферы в графе

```

val archBuf = Flow[Event]
    .buffer(archBufSize, OverflowStrategy.fail)

val warnBuf = Flow[Event]
    .buffer(warnBufSize, OverflowStrategy.dropHead)

val errBuf = Flow[Event]
    .buffer(errBufSize, OverflowStrategy.backpressure)

val metricBuf = Flow[Event]
    .buffer(errBufSize, OverflowStrategy.dropHead)
    
```

← Переполнение буфера в потоке архивирования вызывает остановку потока  
 ← При переполнении буфера предупреждений отбрасывается самый старый элемент  
 ← При переполнении буфера ошибок задействуется механизм обратного давления  
 ← При переполнении буфера показателей отбрасывается самый старый элемент

Самые старые предупреждения можно отбросить, если нагрузка оказалась настолько высока, что служба уведомлений не справляется с их обработкой. Сведения об ошибках нельзя отбрасывать. Критические ошибки не буферизуются, поэтому для них по умолчанию будет использован механизм обратного давления.

В листинге 13.36 показано, как конструируется этот граф.

### Листинг 13.36. Конструирование узлов графа

```
val bcast = builder.add(Broadcast[Event](5))
val wbcast = builder.add(Broadcast[Event](2))
val ebcast = builder.add(Broadcast[Event](2))
val cbcast = builder.add(Broadcast[Event](2))
val okcast = builder.add(Broadcast[Event](2))

val mergeNotify = builder.add(MergePreferred[Summary](2))
val archive = builder.add(jsFlow)
```

`MergePreferred` всегда имеет один порт `preferred` и несколько вторичных портов, в данном случае их два. Сначала посмотрим, как соединяются между собой узлы графа, а потом рассмотрим отдельные потоки.

### Листинг 13.37. Соединение узлов графа

```
bcast ~> archBuf ~> archive.in           ← Поток архивных событий буферизуется
bcast ~> ok      ~> okcast              и соединяется с исходящим потоком
bcast ~> warning ~> wbcast              архивной службы
bcast ~> error   ~> ebcast
bcast ~> critical ~> cbcast

okcast ~> jsFlow ~> logFileSink(logId, Ok)
okcast ~> metricBuf ~>
  toMetric ~> recordDrift ~> metricOutFlow ~> metricsSink ← Поток показателей

cbcast ~> jsFlow ~> logFileSink(logId, Critical)
cbcast ~> toNot ~> mergeNotify.preferred ← Критические ошибки пересылаются в
                                          первую очередь, если на других входах
                                          тоже присутствуют данные

ebcast ~> jsFlow ~> logFileSink(logId, Error)
ebcast ~> errBuf ~> rollupErr ~> mergeNotify.in(0) ← Поток ошибок

wbcast ~> jsFlow ~> logFileSink(logId, Warning)
wbcast ~> warnBuf ~> rollupWarn ~> mergeNotify.in(1) ← Поток предупреждений

mergeNotify ~> notifyOutFlow ~> notificationSink

FlowShape(bcast.in, archive.out)
```

В следующем разделе мы посмотрим, как обрабатывать элементы в потоках с разными скоростями. Для обособления сторон в потоках, действующих с разными скоростями, используется специальная потоковая операция.

## 13.5. Обособление частей графа, действующих с разной скоростью

Механизм обратного давления, по умолчанию действующий для всех компонентов akka-stream, иногда желательно отключить для некоторых частей графа. В некоторых ситуациях, в результате действия механизма обратного давления, один медлительный узел в графе может замедлить работу всех других узлов. В других ситуациях требуется, чтобы один высокоскоростной потребитель мог действовать быстрее других узлов, чему опять же может помешать механизм обратного давления.

Типичный способ обособления частей графа, действующих с разной скоростью, заключается в добавлении буфера между узлами. Буфер отсрочит влияние обратного давления, пока в нем имеется место.

Чтобы объяснить, как действует прием обособления, предположим, что служба уведомлений действует медленнее других. Вместо немедленной отправки каждого отдельного уведомления по его прибытии мы будем помещать их в буфер, накапливая в течение некоторого времени.

Допустим также, что служба показателей действует быстрее других, поэтому мы можем добавить дополнительную обработку. В данном случае будем фиксировать, насколько обработчик потока действует медленнее службы показателей.

Можно также организовать вычисление других расширенных характеристик, например вычислять и отправлять разность между фактическими и интерполируемыми показателями. Все это мы оставляем читателям в качестве самостоятельного упражнения.

### 13.5.1. Медленный потребитель, накопление событий в блоках

Обработчик потока должен записывать уведомления в блоки Summary и посылать их службе уведомлений для извещения операторов о важных событиях. Блоки имеют разные приоритеты: критические события посылаются по одному, тогда как обычные ошибки и предупреждения накапливаются в блоки за определенный интервал времени или до достижения максимального числа событий.

Для упрощения интерфейса все уведомления будут посылаться в виде блоков Summary, поэтому критические события будут посылаться в блоках Summary по одному.

**Листинг 13.38.** Блок с одним критическим событием

```
val toNot = Flow[Event].map(e=> Summary(Vector(e)))
```

Предупреждения и ошибки накапливаются в блоках с помощью `groupedWithin`, как показано в листинге 13.39. Для создания значений `rollupErr` и `rollupWarn`, показанных выше в листинге 13.37, используется метод `rollup`, объявленный здесь, который преобразует события `Event` в объекты `Summary`.

**Листинг 13.39.** Накопление событий с помощью `groupedWithin`

```
def rollup(nr: Int, duration: FiniteDuration) =
  Flow[Event].groupedWithin(nr, duration)
    .map(events => Summary(events.toVector))
```

← `groupedWithin` возвращает  
`List[Event]`

```
val rollupErr = rollup(nrErrors, errDuration)
val rollupWarn = rollup(nrWarnings, warnDuration)
```

Как показано в листинге 13.37, `rollupErr` и `rollupWarn` передаются во вторичные входы `MergePreferred`, а в приоритетный вход `preferred` передаются критические ошибки.

**ПОВЕДЕНИЕ GROUPEDWITHIN В МОМЕНТ ЗАКРЫТИЯ ПОТОКА.**

Важно отметить, что `groupedWithin` автоматически посылает остаток буфера как `Summary`, когда происходит закрытие потока. В данном примере предполагается, что производитель посылает данные непрерывно. Но если запустить `aia.stream.LogStreamProcessorApp` и послать файл журнала службе, остаток буфера всегда будет записываться в файл с уведомлениями, даже если событий меньше, чем вмещает `Summary`, или время накопления еще не истекло.

**13.5.2. Быстрый потребитель, дополнительные показатели**

Служба показателей в этом примере считается быстрым потребителем. Мы все еще используем буфер на всякий случай, но, поскольку служба способна потреблять события быстрее, чем их отдает обработчик потока, было бы интересно узнать, насколько последний отстает от потенциальных возможностей службы. В листинге 13.40 показано, как событие `Event` преобразуется в показатель `Metric`.

**Листинг 13.40.** Преобразование события `Event` в показатель `Metric`

```
val toMetric = Flow[Event].collect {
  case Event(_, service, _, time, _, Some(tag), Some(metric)) =>
```

```
Metric(service, time, metric, tag)
}
```

Также можно было бы добавить вывод некоторой дополнительной информации, когда потребитель запрашивает больше, чем доступно. Вместо оказания обратного давления можно генерировать дополнительные элементы и посылать их обратно службе-потребителю.

В данном случае мы будем повторно посылать тот же показатель, но с дополнительным полем `drift`, показывающим количество элементов, которое могла бы потребить служба показателей, если бы обработчик журналируемых записей действовал достаточно быстро.

#### Листинг 13.41. Добавление дополнительной информации в Metric

```
val recordDrift = Flow[Metric]
    .expand { metric =>
        Iterator.from(0).map(d => metric.copy(drift = d))
    }
```

Метод `expand` принимает аргумент с функцией типа `Out =< Iterator[U]`. Тип `U` выводится из функции, и в данном случае это тип `Metric`. Когда в потоке отсутствуют элементы, он выбирает элементы из этого итератора. Поле `drift` получает нулевое значение, если обработчик потока успевает оставлять новые элементы, и начинает возрастать при повторной отправке данных `Metric` службе показателей, когда она оказывается быстрее.

Важно отметить, что использование буферов в этом сценарии не предотвращает завершения потоков с ошибкой. В данном контексте это просто означает, что потоковый запрос HTTP потерпит неудачу.

## 13.6. В заключение

Разработка потоковых приложений с Akka – очень обширная тема, и в этой главе мы лишь слегка коснулись ее. И все же мы показали достаточно, чтобы вы могли убедиться, что `akka-stream` предлагает универсальный и гибкий API для потоковых приложений. Возможность определения структуры графа и его запуска позднее имеет большое значение для создания многократно используемого кода.

В случае простых линейных потоков можно применять комбинаторы к `Source`, `Flow` и `Sink`. Компонент `BidiFlow` отлично подходит для реализации многократно используемых протоколов. Имеющие опыт создания потоковых HTTP-приложений наверняка согласятся, что переключиться с файлов на `akka-http` получилось на удивление просто, как было показано в разделе 13.2.

Тот факт, что модуль `akka-http` основан на `akka-stream`, действительно дает большое преимущество, как вы могли убедиться на примере реализации согласования типа контента с использованием своих инструментов преобразования.

Механизм обратного давления позволяет обрабатывать потоки в ограниченном объеме памяти. Он действует по умолчанию, и его поведение можно изменять с помощью определенных методов, таких как `buffer` и `expand`.

Модуль `akka-stream` включает множество готовых и удобных этапов графов. Мы обсудили лишь некоторые из них, но в будущем, как предполагается, будет добавлено множество новых, скорее всего, в отдельной библиотеке `akka-stream-contrib`, которые уменьшат необходимость создания своих этапов `GraphStage` (не рассматривавшихся в этой главе).

# Глава 14

## Кластеры

В этой главе:

- динамическое масштабирование акторов в кластерах;
- отправка сообщений в кластер через маршрутизатор с поддержкой кластеров;
- создание приложения для кластера с помощью Akka.

В главе 6 вы узнали, как создавать распределенные приложения с фиксированным количеством узлов. Подход со статическим членством прост, но не имеет готового решения балансировки нагрузки или обеспечения отказоустойчивости. Кластер позволяет динамически увеличивать и уменьшать количество узлов, используемых распределенным приложением, и устраняет страх из-за наличия единой точки отказа.

Многие распределенные приложения работают в средах, которые не полностью находятся под вашим контролем, как, например, облачные платформы или вычислительные центры, разбросанные по всему миру. Чем больше кластер, тем выше вероятность отказа. Тем не менее в вашем распоряжении имеются средства мониторинга и управления жизненным циклом кластера. В первом разделе этой главы мы посмотрим, как узел становится членом кластера, как принимать события, связанные с членством в кластере, и как определять аварийные узлы в кластере.

Сначала мы создадим кластерное приложение, подсчитывающее вхождение каждого слова в некоторый фрагмент текста. В ходе разработки примера вы узнаете, как можно использовать маршрутизаторы для взаимодействий с акторами в кластере, как создавать устойчивые и скоординированные процессы из множества акторов в кластере и как тестировать кластерные системы акторов.

### 14.1. Зачем нужны кластеры?

Кластер – это динамическая группа узлов. На каждом узле имеется своя система акторов, принимающая запросы из сети (подобно тому, как описы-

валось в главе 6). Кластеры конструируются на основе модуля `akka-remote` и помогают поднять прозрачность местоположения акторов на новый уровень. Актор может существовать в локальной или удаленной системе и вообще находиться где угодно в пределах кластера; вам не придется беспокоиться о его местонахождении в своем коде. На рис. 14.1 показан кластер с четырьмя узлами.

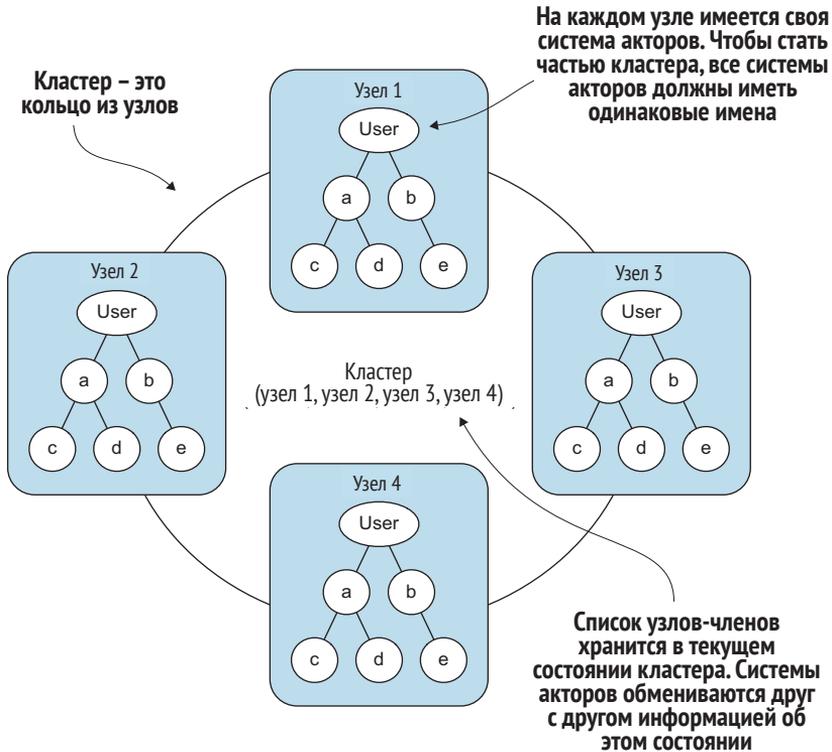


Рис. 14.1. Кластер с четырьмя узлами

Конечной целью модуля `akka.cluster` является предоставление полностью автоматизированного способа распределения акторов, балансировки нагрузки и поддержание отказоустойчивости. В настоящее время модуль `akka.cluster` поддерживает следующие функции:

- *членство в кластере* – надежное членство для систем акторов;
- *балансировка нагрузки* – маршрутизация сообщений в кластере на основе специализированного алгоритма;
- *разделение узлов* – узлу можно назначить определенную роль; маршрутизаторы можно настраивать для отправки сообщений только узлам с определенной ролью;
- *разделение точек* – систему акторов можно разделить на поддеревья акторов, расположенные на разных узлах.

В этой главе мы подробно рассмотрим все эти функции, уделив особое внимание членству в кластере и маршрутизации. В главе 15 мы обсудим механизмы репликации состояния и автоматическую поддержку отказоустойчивости.

Приложение обработки данных с единственной целью – отличный пример кандидата на создание кластера. К таким приложениям можно отнести приложения распознавания образов или анализа социальных сетей в режиме реального времени. Узлы в таких приложениях могут добавляться или удаляться по мере увеличения или уменьшения нагрузки на приложение. Задания, осуществляющие обработку, находятся под наблюдением: если актер терпит неудачу, задание перезапускается, и такие попытки повторяются, пока обработка не завершится успехом. В данной главе мы рассмотрим простой пример приложения такого типа. На рис. 14.2 показана его общая структура; пока не обращайтесь особого внимания на детали, поскольку все они будут подробно рассматриваться далее в главе.

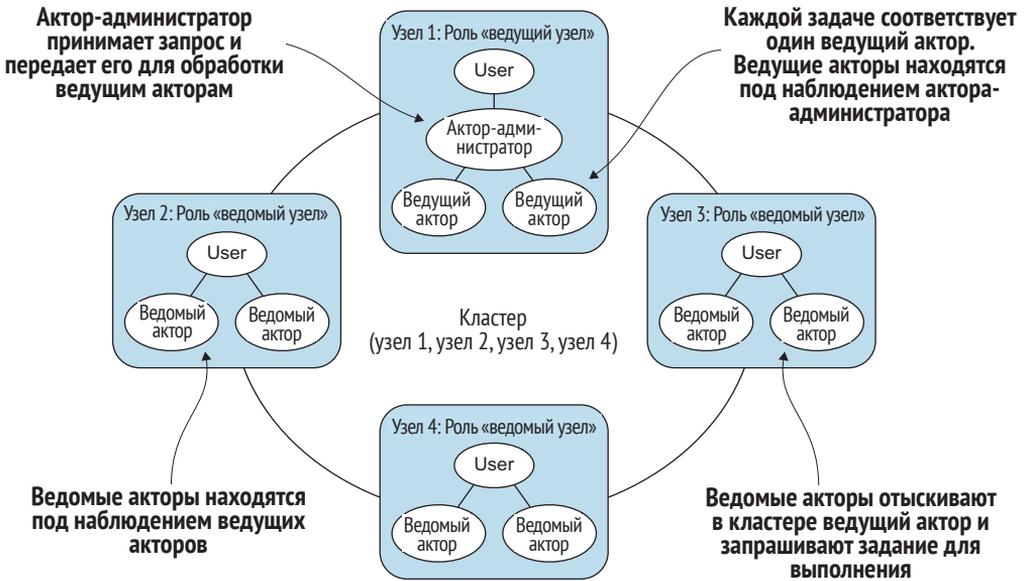


Рис. 14.2. Обработка заданий в кластере

А теперь перейдем к реализации нашего кластеризованного приложения подсчета слов. В следующем разделе мы углубимся в детали членства в кластере и посмотрим, как ведущие и ведомые актеры могут находить друг друга и сотрудничать для выполнения заданий.

## 14.2. Членство в кластере

Начнем с создания кластера. Кластер состоит из ведущего и ведомых узлов. На рис. 14.3 изображен кластер, который нам предстоит создать.

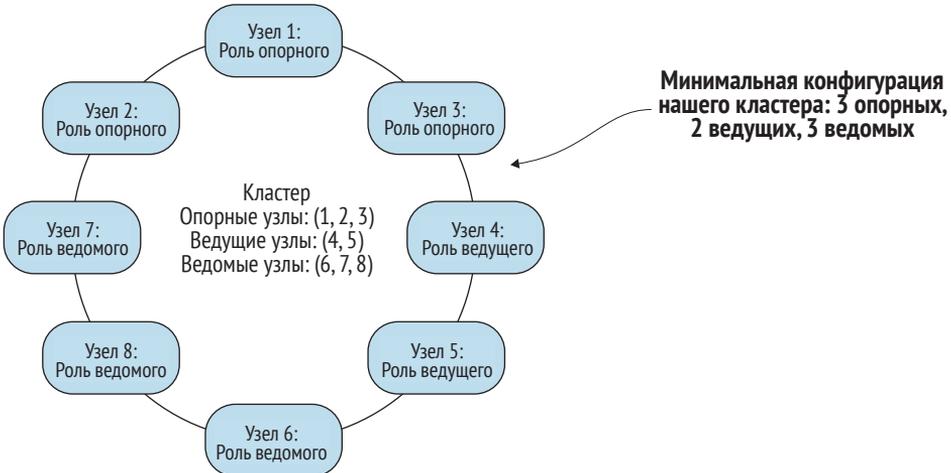


Рис. 14.3. Кластер для подсчета слов

Ведущие узлы управляют выполнением заданий подсчета слов. Ведомые узлы запрашивают задания у ведущих, обрабатывают свои части текста и возвращают полученные результаты ведущему узлу. Ведущий узел формирует окончательный результат, как только все части задания будут выполнены. Задание повторно запускается, если какой-то ведущий или ведомый узел потерпит неудачу.

На рис. 14.3 также можно видеть узлы еще одного типа, которые необходимы в кластере, а именно *опорные узлы* (seed nodes). Опорные узлы необходимы для запуска кластера. В следующем разделе мы посмотрим, как узлы становятся опорными узлами и как они могут присоединяться к кластеру и отсоединяться от него. Мы подробно рассмотрим процесс создания кластера и поэкспериментируем с формированием и изменением простого кластера в консоли REPL. Вы познакомитесь с разными состояниями, через которые проходят узлы-члены, и узнаете, как подписаться на уведомления об их изменении.

### 14.2.1. Присоединение к кластеру

Как в любых других группах, для запуска процесса необходимо иметь несколько «основателей». В Akka для этой цели поддерживается возможность запуска опорных узлов. Опорные узлы фактически являются начальными точками роста кластера и служат первой инстанцией для осуществления контактов с другими узлами. Узлы присоединяются к кластеру, посылая сообщение *join*, содержащее уникальный адрес присоединяемого узла. Модуль `akka.cluster` автоматически посылает это сообщение одному из зарегистрированных опорных узлов. Опорный узел не обязан содержать каких-либо пользовательских акторов, поэтому есть возможность созда-

вать опорные узлы, отвечающие исключительно за поддержание кластера. На рис. 14.4 показано, как первый опорный узел инициализирует кластер и как затем другие узлы присоединяются к нему.

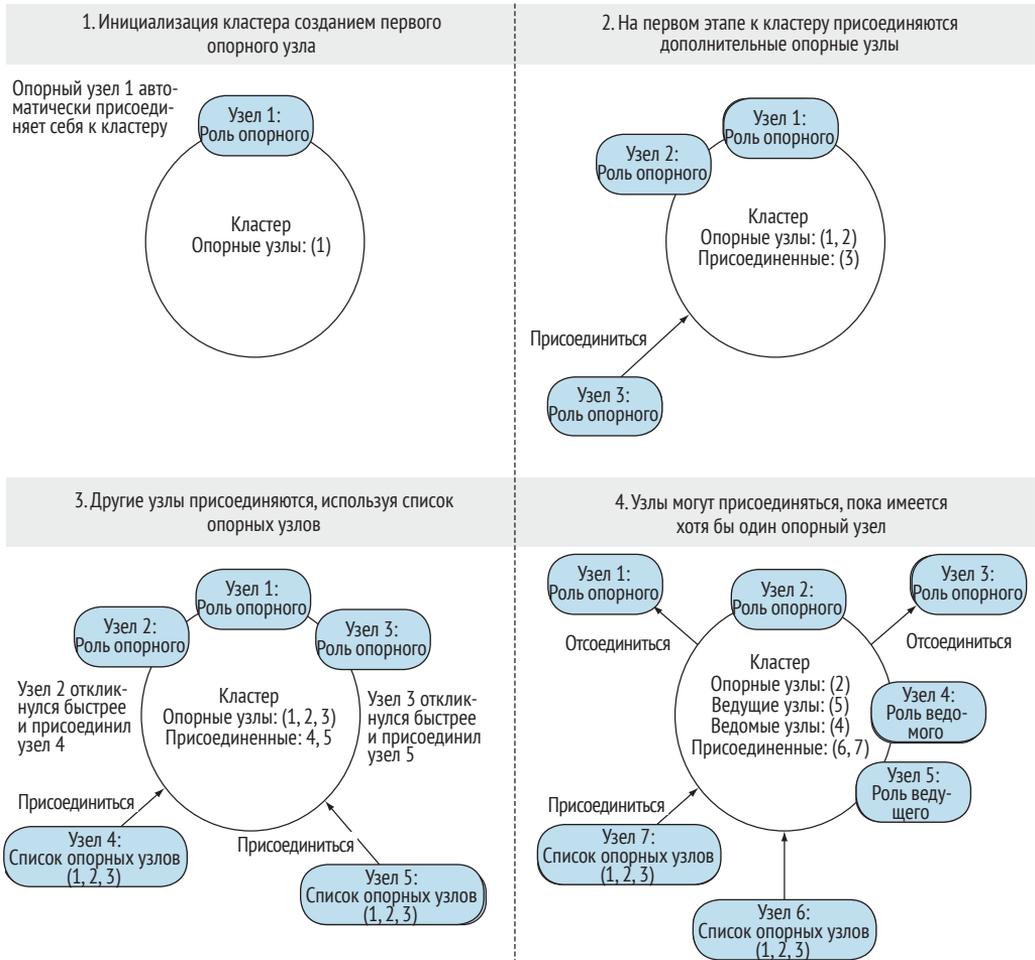


Рис. 14.4. Инициализация кластера опорными узлами

Модуль `akka.cluster` не поддерживает (пока) протокол автоматического обнаружения с пустой конфигурацией, подобно широковещательным рассылкам TCP или обнаружению службы DNS. Поэтому необходимо явно указать список опорных узлов или каким-то образом узнать имя хоста и номер порта узла кластера для присоединения. Первый опорный узел в списке играет особую роль в начальном формировании кластера. Следующий опорный узел зависит от первого. Первый опорный узел в списке автоматически формирует кластер и присоединяет себя к нему. Он должен быть запущен и настроен до того, как следующие опорные узлы предпримут попытку присоединиться к кластеру. Это ограничение было введе-

но для предотвращения образования отдельных кластеров, пока первый опорный узел еще не успел запуститься.

### Присоединение узлов к кластеру вручную

Опорные узлы не являются чем-то обязательным; кластер можно сформировать, вручную запустив узел, который сам присоединит себя. Все последующие узлы необходимо будет присоединять к этому узлу, посылая сообщение `Join`.

Это означает, что все они должны знать адрес первого узла, поэтому процедура формирования кластера с привлечением опорных узлов выглядит проще. Иногда нет никакой возможности узнать IP-адрес или DNS-имена серверов в сети. В таких ситуациях на выбор есть три варианта:

- использовать список опорных узлов с известными IP-адресами или DNS-именами, находящихся за пределами сети, где адреса и имена хостов невозможно определить заранее. Такие опорные узлы не выполняют никакого прикладного кода и служат исключительно для формирования кластера;
- спроектировать и реализовать свой протокол автоматического обнаружения, соответствующий вашей конфигурации сети. Но это очень непростая задача;
- использовать существующие технологии обнаружения/регистрации, такие как Apache ZooKeeper, HashiCorp Consul или CoreOS/etcd, и добавить свой «связующий» код. Добавить в каждый узел кластера некоторый код для регистрации в некоторой службе реестра в момент запуска и написать адаптер, который будет обращаться к этой службе за списком доступных узлов в кластере.

Но имейте в виду, что для решения на основе ZooKeeper все еще необходимо иметь полную информацию об адресах хостов и номерах портов, поэтому данный путь фактически заключается в подмене одного множества известных адресов другим. Организация службы реестра для хранения списка доступных узлов также может оказаться непростой задачей, потому что она может зависеть от множества других факторов, незаметных на первый взгляд. Будьте осторожны. (Может потребоваться использовать разные модели согласования, и из-за недостатка опыта вас могут поджидать разные подводные камни.)

Опорные узлы могут запускаться независимо друг от друга, при условии что первый опорный узел обязательно будет запущен рано или поздно. Последующие опорные узлы будут ждать, пока первый узел завершит запуск. Другие узлы присоединяются к кластеру посредством любых опор-

ных узлов, после запуска первого и присоединения к кластеру еще хотя бы одного. Сообщения посылаются всем опорным узлам; первый ответивший узел осуществит присоединение. Первый опорный узел может безопасно покинуть кластер, после того как появятся еще хотя бы два члена. На рис. 14.5 показано, как можно сформировать кластер с ведущими и ведомыми узлами после запуска уже первого опорного узла.

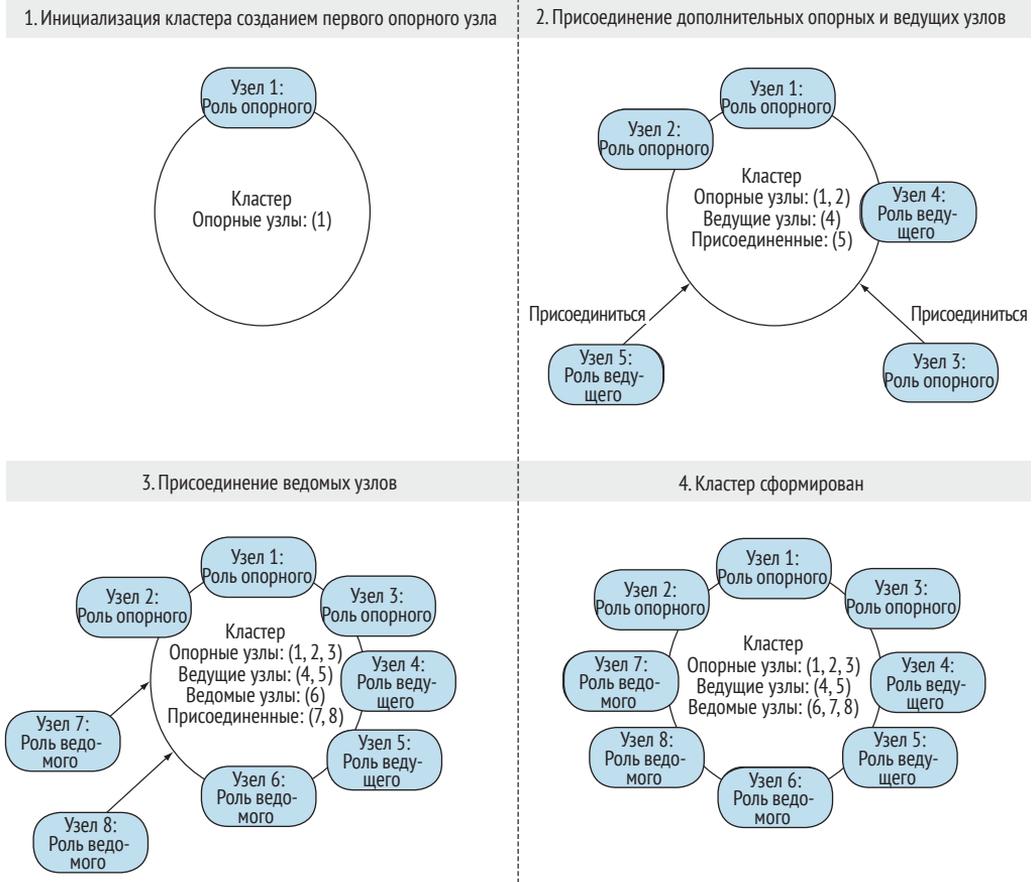


Рис. 14.5. Кластер для выполнения заданий

Начнем наши эксперименты с создания опорных узлов в консоли REPL, чтобы лучше понять, как формируется кластер.

Имейте в виду, что необязательно выполнять все эти шаги вручную после развертывания кластерного приложения. В зависимости от окружения присваивание адресов и запуск опорных узлов можно реализовать в сценариях развертывания или запуска.

Проект с этим примером можно найти в папке *chapter-cluster*.

Прежде всего нужно настроить на использование модуля *cluster*. Для этого в файл сборки добавьте зависимость *akka-cluster*:

```
"com.typesafe.akka" %% "akka-cluster" % akkaVersion
```

← Файл сборки определяет значение версии Akka

Для настройки модуля необходим `akka.cluster.ClusterActorRefProvider`, так же как для настройки модуля `akka-remote` необходим `akka.remote.RemoteActorRefProvider`. Cluster API поддерживается как расширение Akka. Класс `ClusterActorRefProvider` инициализирует расширение `Cluster` при создании системы акторов.

В листинге 14.1 приводится минимально необходимая конфигурация для опорных узлов (ее можно найти в файле `src/main/resources/seed.conf`).

#### Листинг 14.1. Конфигурация для опорных узлов

```
akka {
  loglevel = INFO
  stdout-loglevel = INFO
  event-handlers = ["akka.event.Logging$DefaultLogger"]

  log-dead-letters = 0
  log-dead-letters-during-shutdown = off

  actor {
    provider = "akka.cluster.ClusterActorRefProvider"
  }

  remote {
    enabled-transport = ["akka.remote.netty.tcp"]
    log-remote-lifecycle-events = off
    netty.tcp {
      hostname = "127.0.0.1"
      hostname = ${?HOST}
      port = ${PORT}
    }
  }

  cluster {
    seed-nodes = [
      "akka.tcp://words@127.0.0.1:2551",
      "akka.tcp://words@127.0.0.1:2552",
      "akka.tcp://words@127.0.0.1:2553"
    ]
    roles = ["seed"]

    role {
      seed.min-nr-of-members = 1
    }
  }
}
```

← Инициализация модуля `akka.cluster`

← Конфигурация для данного опорного узла

← Раздел конфигурации кластера

← Опорные узлы кластера

← Опорному узлу назначается соответствующая роль, отличающая его от ведущих и ведомых узлов

← Минимальное количество членов кластера для каждой роли, чтобы кластер считался «запущенным». В случае с опорными узлами кластер считается запущенным после запуска хотя бы одного опорного узла

```
}
}
```

### Гарантируйте неизменность адресов

Следуя за примерами, обязательно используйте IP-адрес 127.0.0.1; имя localhost может разрешаться в разные IP-адреса в зависимости от системных настроек, а Akka интерпретирует адреса буквально. Вы не должны зависеть от особенностей разрешения имен в DNS. Значение в akka.remote.netty.tcp.host используется *в точности* как указано; для его разрешения не используется DNS. Точное значение адреса используется, когда выполняется сериализация ссылок на акторы между удаленными узлами Akka. То есть после отправки сообщения удаленному актору с использованием такой ссылки для подключения к удаленному серверу будет использоваться этот точный адрес. Главная причина отказа от DNS – производительность. Разрешение имен в DNS, если настройки выполнены неправильно, может занимать до нескольких секунд, а в паталогических ситуациях – даже минут. Поиск причин задержек, обусловленных неправильной конфигурацией DNS, – сложная задача, и не всегда очевидная. Отказ от использования DNS помогает избежать этой проблемы, но это означает, что вы должны проявлять осторожность, выполняя настройку адресов.

Все узлы в последующих примерах будут запускаться на локальной машине. Если вы захотите попробовать проверить их работу в сети, просто передайте в параметрах -DHOST и -DPORT соответствующее имя хоста и номер порта соответственно, чтобы установить переменные окружения HOST и PORT. Эти переменные окружения, если они имеются, переопределяют конфигурационные параметры в файле *seed.conf*. Запустите sbt в трех терминалах, в каталоге *chapter-cluster*, указав разные порты. Вот как выглядит команда запуска sbt для первого опорного узла:

```
sbt -DPORT=2551 -DHOST=127.0.0.1
```

Выполните аналогичную команду в двух других терминалах, передав в параметре -DPORT значения 2552 и 2553. Все узлы в кластере должны запустить систему акторов с одним и тем же именем (*words* в данном примере). Перейдите в первый терминал, где запущен первый опорный узел.

Первый опорный узел должен автоматически запустить и сформировать кластер. Давайте проверим это в сеансе REPL. Запустите консоль в sbt (введите команду *console* в строке приглашения sbt) в первом терминале и следуйте далее за листингом 14.2. Результат показан на рис. 14.6.

**Листинг 14.2.** Запуск опорного узла

```

...
scala> :paste
// Вход в режим вставки (выход комбинацией ctrl-D)

import akka.actor._

import akka.cluster._

import com.typesafe.config._

val seedConfig = ConfigFactory.load("seed")
val seedSystem = ActorSystem("words", seedConfig)

// выход из режима вставки в режим интерпретации.

[Remoting] Starting remoting
[Remoting] listening on addresses :
[akka.tcp://words@127.0.0.1:2551]
...
[Cluster(akka://words)]
Cluster Node [akka.tcp://words@127.0.0.1:2551]
- Started up successfully
Node [akka.tcp://words@127.0.0.1:2551] is JOINING, roles [seed]
[Cluster(akka://words)] Cluster Node [akka.tcp://words@127.0.0.1:2551]
- Leader is moving node [akka.tcp://words@127.0.0.1:2551] to [Up]

```

← Загрузка конфигурации для опорного узла из файла src/main/resources/seed.conf  
 ← Запуск системы акторов words на опорном узле  
 ← Автоматический запуск модулей remote и cluster  
 ← Имя кластера совпадает с именем системы акторов  
 ← Опорный узел кластера words запущен  
 ← Опорный узел автоматически присоединяется к кластеру words



**Рис. 14.6.** Запуск первого опорного узла

Запустите консоли в двух других терминалах и вставьте тот же код из листинга 14.2, чтобы запустить опорные узлы 2 и 3. Эти узлы будут про-

слушивать порты, указанные в параметре `-DPORT`, переданном команде `sbt`. На рис. 14.7 показаны результаты выполнения команд в интерактивных оболочках REPL для опорных узлов 2 и 3.

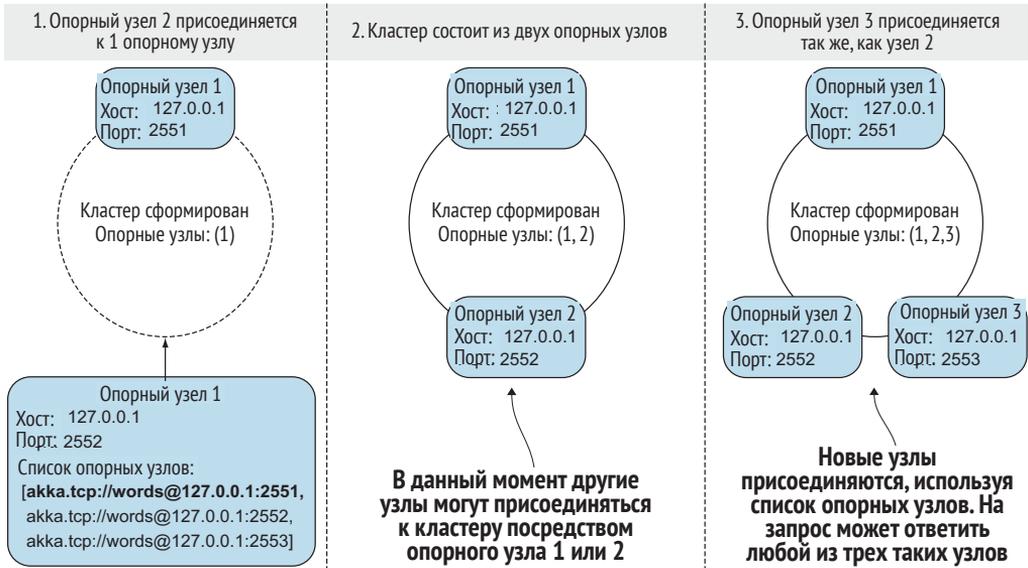


Рис. 14.7. Запуск второго и третьего опорных узлов

В двух других терминалах вы должны увидеть строки, подобные тем, что показаны в листинге 14.3, подтверждающие присоединение узлов к кластеру.

#### Листинг 14.3. Подтверждение присоединения опорного узла к кластеру

```
[Cluster(akka://words)] Cluster Node [akka.tcp://words@127.0.0.1:2553]
- Welcome from [akka.tcp://words@127.0.0.1:2551]
```

← Вывод отформатирован для удобочитаемости; в окне терминала текст отображается в одну строку

В листинге 14.4 показан вывод в консоли с первым опорным узлом. Как можно заметить, первый опорный узел получил запросы двух других опорных узлов и присоединил их к кластеру.

#### Листинг 14.4. Вывод в терминале с первым опорным узлом

```
Первый опорный узел присоединил сам себя и стал лидером
Вывод отформатирован для удобочитаемости
[Cluster(akka://words)] Cluster Node [akka.tcp://words@127.0.0.1:2551]
- Node [akka.tcp://words@127.0.0.1:2551] is JOINING, roles [seed]
- Leader is moving node [akka.tcp://words@127.0.0.1:2551] to [Up]
```

```

- Node [akka.tcp://words@127.0.0.1:2552] is JOINING, roles [seed]
- Leader is moving node [akka.tcp://words@127.0.0.1:2552] to [Up]
- Node [akka.tcp://words@127.0.0.1:2553] is JOINING, roles [seed]
- Leader is moving node [akka.tcp://words@127.0.0.1:2553] to [Up]

```

← Присоединение опорного узла 2  
← Присоединение опорного узла 3

Один из узлов кластера берет на себя особые функции: функции *лидера* кластера. Задача лидера – вовремя определить подключение или отключение узла-члена. В данном случае лидером становится первый опорный узел.

В каждый момент времени лидером может быть только один узел. Лидером может стать любой узел кластера. Опорные узлы 2 и 3 запрашивают присоединение к кластеру и переходят в состояние ожидания присоединения JOINING. Лидер переводит узлы в состояние Up, делая их частью кластера. Теперь все три узла благополучно присоединились к кластеру.

## 14.2.2. Выход из кластера

Теперь посмотрим, что случится, когда первый опорный узел покинет кластер. В листинге 14.5 приводятся команды, заставляющие первый опорный узел выйти из кластера.

### Листинг 14.5. Опорный узел 1 покидает кластер

```

scala> val address = Cluster(seedSystem).selfAddress
address: akka.actor.Address = akka.tcp://words@127.0.0.1:2551
scala> Cluster(seedSystem).leave(address)
[Cluster(akka://words)] Cluster Node [akka.tcp://words@127.0.0.1:2551]
- Marked address [akka.tcp://words@127.0.0.1:2551] as [Leaving]
[Cluster(akka://words)] Cluster Node [akka.tcp://words@127.0.0.1:2551]
- Leader is moving node [akka.tcp://words@127.0.0.1:2551] to [Exiting]
[Cluster(akka://words)] Cluster Node [akka.tcp://words@127.0.0.1:2551]
- Shutting down...
[Cluster(akka://words)] Cluster Node [akka.tcp://words@127.0.0.1:2551]
- Successfully shut down

```

← Получение адреса этого узла  
← Выход опорного узла 1 из кластера  
← Отмечается как покидающий кластер  
← Отмечается как вышедший

В листинге 14.5 видно, как опорный узел отмечает себя как *Leaving* (покидающий кластер) и затем *Exiting* (вышедший из кластера), оставаясь при этом лидером. Эти изменения состояния передаются всем узлам в кластере. Затем узел кластера останавливается. Сама система акторов (*seedSystem*) на узле не останавливается автоматически. Что происходит с кластером? Узел, выполняющий роль лидера, просто останавливается. На рис. 14.8 показано, как первый опорный узел покидает кластер и как лидерство передается другому узлу.



Рис. 14.8. Первый опорный узел покидает кластер

### Протокол обмена служебной информацией

Возможно, вам интересно, как опорные узлы в этом примере узнают о том, что первый узел решил покинуть кластер, затем вышел и, наконец, был удален новым лидером. Для этого в Akka используется *протокол обмена служебной информацией* о состоянии между всеми узлами в кластере.

Каждый узел рассылает другим узлам информацию о своем состоянии и о состояниях других узлов, которые он видит. Этот протокол позволяет всем узлам в кластере согласовать сведения о состоянии между узлами. Это *согласование* выполняется в течение длительного времени, пока узлы не достигнут договоренности друг с другом.

Лидер кластера определяется после достижения договоренности. Лидером автоматически становится первый узел (в порядке сортировки), находящийся в состоянии Up или Leaving. (Сортировка выполняется по полным адресам узлов, таким как akka.tcp://words@127.0.0.1:2551.)

Далее посмотрим, что происходит в других терминалах. В одном из двух оставшихся терминалов должен отображаться вывод, как показано в листинге 14.6.

#### Листинг 14.6. Опорный узел 2 становится лидером и удаляет узел 1 из кластера

```
[Cluster(akka://words)] Cluster Node [akka.tcp://words@127.0.0.1:2552]
- Marking exiting node(s) as UNREACHABLE
[Member(address = akka.tcp://words@127.0.0.1:2551, status = Exiting)].
```

This is expected and they will be removed.

← Вышедший узел получает статус `Exiting`

```
[Cluster(akka://words)] Cluster Node [akka.tcp://words@127.0.0.1:2552]
```

```
- Leader is removing exiting node [akka.tcp://words@127.0.0.1:2551]
```

← Лидер удаляет  
вышедший узел

Оба оставшихся опорных узла обнаруживают, что первый узел отмечен как `UNREACHABLE` (недостижимый). Кроме того, оба опорных узла знают, что первый опорный узел предупредил о выходе из кластера. Второй опорный узел автоматически становится лидером, когда первый переходит в состояние `Exiting`. После этого узел, покинувший кластер, переводится из состояния `Exiting` в состояние `Removed`. Теперь в кластере имеется только два опорных узла.

Система акторов первого опорного узла теперь не может повторно присоединиться к кластеру простым вызовом `Cluster(seedSystem).join(selfAddress)`. Система акторов удаляется, и для повторного присоединения ее нужно перезапустить. В листинге 14.7 показано, как выполнить «повторное присоединение» первого опорного узла.

#### Листинг 14.7. Повторное присоединение первого опорного узла к кластеру

```
scala> seedSystem.terminate()
```

```
scala> val seedSystem = ActorSystem("words", seedConfig)
```

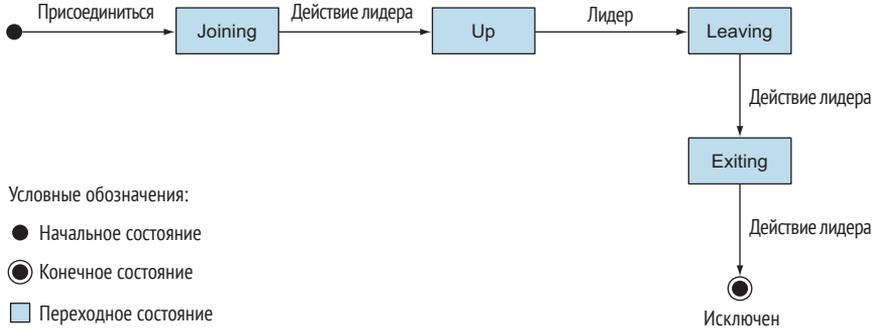
← Завершение  
системы акторов

Запуск новой системы акторов  
с той же конфигурацией.  
Система акторов автоматически  
присоединится к кластеру

В течение своего жизненного цикла система акторов может присоединиться к кластеру только один раз. Но мы всегда можем запустить новую систему акторов с той же конфигурацией, используя тот же адрес хоста и номер порта, как показано в листинге 14.7.

Теперь вы знаете, как узлы могут присоединяться к кластеру и покидать его. На рис. 14.9 изображена диаграмма состояний, которые мы видели к данному моменту. Лидер выполняет определенные функции, управляя состоянием членов, переводя их из состояния `Joining` в состояние `Up` и из состояния `Exiting` в состояние `Removed`.

Однако это не полная картина происходящего. Давайте рассмотрим случай, когда один из опорных узлов завершится аварийно. Для этого можно просто остановить процесс терминала, в котором запущен опорный узел 1, и посмотреть на вывод в других терминалах. В листинге 14.8 показан вывод в окне терминала, где запущен опорный узел 2 после остановки процесса терминала с опорным узлом 1.



**Рис. 14.9.** Диаграмма состояний узлов, присоединяющихся к кластеру и покидающих его

#### Листинг 14.8. Аварийное завершение узла 1

```
Cluster Node [akka.tcp://words@127.0.0.1:2552]
- Marking node(s) as UNREACHABLE
  [Member(address = akka.tcp://words@127.0.0.1:2551, status = Up)]
```

Опорный узел 1  
становится  
недоступным

Опорный узел 1 отмечается как UNREACHABLE. Для обнаружения недоступности узлов в кластере используется специальный механизм. В момент аварии опорный узел находился в состоянии Up. Вообще, к моменту аварии узел может находиться в любом состоянии. Лидер не может выполнить никаких специальных действий, пока любой из узлов остается недостижимым, это означает, что в такие моменты узел не может покинуть кластер или присоединиться к нему. Недостижимый узел сначала нужно исключить из кластера. Сделать это можно из любого узла в кластере вызовом метода down. В листинге 14.9 показано, как производится исключение первого опорного узла из REPL.

#### Листинг 14.9. Исключение первого опорного узла

```
scala> val address = Address("akka.tcp", "words", "127.0.0.1", 2551)
scala> Cluster(seedSystem).down(address)
```

Исключение  
опорного узла 1

```
[Cluster(akka://words)] Cluster Node [akka.tcp://words@127.0.0.1:2552]
- Marking unreachable node [akka.tcp://words@127.0.0.1:2551] as [Down]
- Leader is removing unreachable node [akka.tcp://words@127.0.0.1:2551]
[Remoting] Association to [akka.tcp://words@127.0.0.1:2551]
having UID [1735879100]
is irrecoverably failed. UID is now quarantined and
all messages to this UID
will be delivered to dead letters.
Remote actorsystem must be restarted to recover from this situation.
```

Опорный узел 1  
изолируется  
и исключается

### Механизм обнаружения аварий

Для определения недостижимости узлов модуль `cluster` использует реализацию *алгоритма определения аварий по нарастающему итогу  $\varphi$*  ( $\varphi$  accrual failure detector). Принцип его работы описан в статье Наохи-ро Хаясибара (Naohiro Hayashibara), Ксавье Дефаго (Xavier Défago), Рами Яред (Rami Yared) и Такуя Катаяма (Takuya Katayama)<sup>1</sup>. Определение аварий – фундаментальная задача поддержания отказоустойчивости в распределенных системах.

Алгоритм определения аварий по нарастающему итогу непрерывно вычисляет значение (называется  $\varphi$  (фи)) вместо определения логического признака, указывающего на аварию (достижим или недостижим узел). В упомянутой статье говорится: «Грубо говоря, это значение отражает степень уверенности в аварийном завершении данного наблюдаемого процесса. Если процесс действительно потерпел аварию и стал недостижимым, с течением времени это значение будет стремиться к бесконечности, поэтому для него выбрано такое название». Это значение используется как индикатор подозрения, что что-то пошло не так (степень подозрительности).

Идея степени подозрительности позволяет настраивать алгоритм определения аварий в широких пределах и отделить прикладные требования от требований мониторинга. Настройка этого механизма в модуле `cluster` осуществляется с помощью параметров в разделе `akka.cluster.failure-detector`, где можно задать пороговое значение  $\varphi$ , выше которого узлы будут считаться недостижимыми.

Узлы часто оказываются недостижимыми в моменты, когда работает сборщик мусора, а это означает, что в эти периоды JVM ничего не сможет делать, пока сборка мусора не завершится.

<sup>1</sup> «The  $\varphi$  Accrual Failure Detector», 10 мая 2004, [www.jaist.ac.jp/~defago/files/pdf/IS\\_RR\\_2004\\_010.pdf](http://www.jaist.ac.jp/~defago/files/pdf/IS_RR_2004_010.pdf).

В выводе также сообщается, что для повторного присоединения опорного узла 1 его систему акторов нужно перезапустить. Есть возможность организовать автоматическое исключение недостижимых узлов. За это отвечает параметр конфигурации `akka.cluster.auto-down-unreachable-after`. Лидер автоматически исключит недостижимый узел по истечении времени, указанного в этом параметре. На рис. 14.10 показаны все возможные переходные состояния узлов в кластере.

Иногда бывает желательно знать, когда какой-то узел в кластере вышел из строя. Для этого можно подписать актер на получение служебных уведомлений, рассылаемых в кластере, вызовом метода `subscribe` расшире-

ния Cluster. В листинге 14.10 представлен актер, подписывающийся на служебные события в кластере (*src/main/scala/aia/cluster/words/ClusterDomainEventListener.scala*).

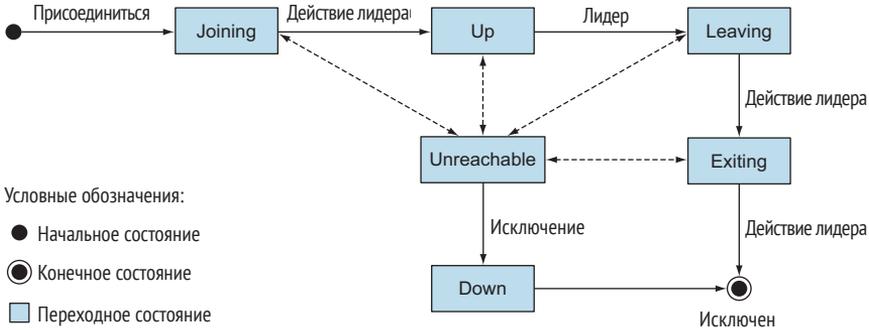


Рис. 14.10. Все возможные переходные состояния узлов в кластере

#### Листинг 14.10. Подписка на служебные события

```

...
import akka.cluster.{MemberStatus, Cluster}
import akka.cluster.ClusterEvent._

class ClusterDomainEventListener extends Actor with ActorLogging {
  Cluster(context.system).subscribe(self, classOf[ClusterDomainEvent])

  def receive = {
    case MemberUp(member) => log.info(s"$member UP.")
    case MemberExited(member) => log.info(s"$member EXITED.")
    case MemberRemoved(m, previousState) =>
      if(previousState == MemberStatus.Exiting) {
        log.info(s"Member $m gracefully exited, REMOVED.")
      } else {
        log.info(s"$m downed after unreachable, REMOVED.")
      }
    case UnreachableMember(m) => log.info(s"$m UNREACHABLE")
    case ReachableMember(m) => log.info(s"$m REACHABLE")
    case s: CurrentClusterState => log.info(s"cluster state: $s")
  }

  override def postStop(): Unit = {
    Cluster(context.system).unsubscribe(self)
    super.postStop()
  }
}

```

Подписка на служебные события кластера в момент создания актора

Принимает служебные события

Аннулирование подписки после остановки актора

Актор `ClusterDomainEventListener` в примере просто записывает все происходящее в журнал.

Служебные события `Cluster` позволяют получать некоторую информацию о членах кластера, но во многих случаях достаточно знать, что некоторый актер все еще присутствует в кластере. Для наблюдения за актерами в кластере можно использовать механизм `DeathWatch` и метод `watch`, как будет показано в следующем разделе.

### 14.3. Обработка заданий в кластере

Теперь попробуем обработать несколько заданий в кластере. Для начала посмотрим, как акторы в кластере взаимодействуют между собой для выполнения задания. Кластер получает текст, в котором нужно подсчитать слова, делит его на фрагменты и раздает нескольким ведомым узлам. Каждый ведомый узел обрабатывает свой фрагмент и подсчитывает вхождения всех слов. Обработка ведомыми узлами выполняется параллельно, что должно привести к увеличению общей скорости. В заключение результаты возвращаются пользователю кластера. Тот факт, что мы в этом примере подсчитываем вхождения слов, не имеет никакого значения; таким способом можно обрабатывать многие другие задания.

Код примера можно найти в том же каталоге *chapter-cluster*, где хранятся примеры присоединения к кластеру и выхода из него. Структура приложения показана на рис. 14.11.

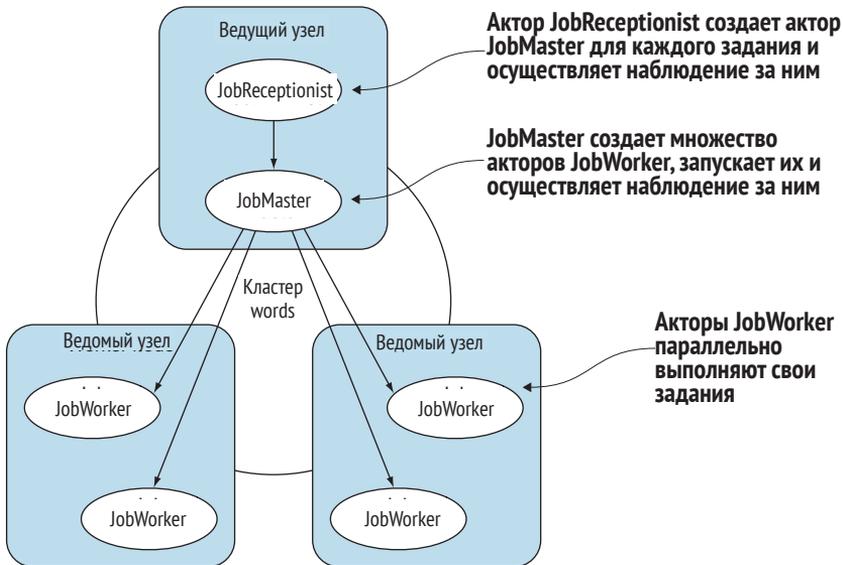


Рис. 14.11. Кластер акторов для подсчета вхождений слов

Актеры `JobReceptionist` и `JobMaster` будут играть роль ведущих узлов. Актеры `JobWorker` будут действовать как ведомые узлы. Оба актора – `Job-`

Master и JobWorker – создаются динамически, по требованию. Всякий раз, когда JobReceptionist получает запрос на выполнение задания JobRequest, он запускает актор JobMaster и передает ему полученное задание для выполнения. Актор JobMaster создает удаленные акторы JobWorker на ведомых узлах. Общую схему процесса можно видеть на рис. 14.12. В оставшейся части главы мы подробно опишем каждый шаг.

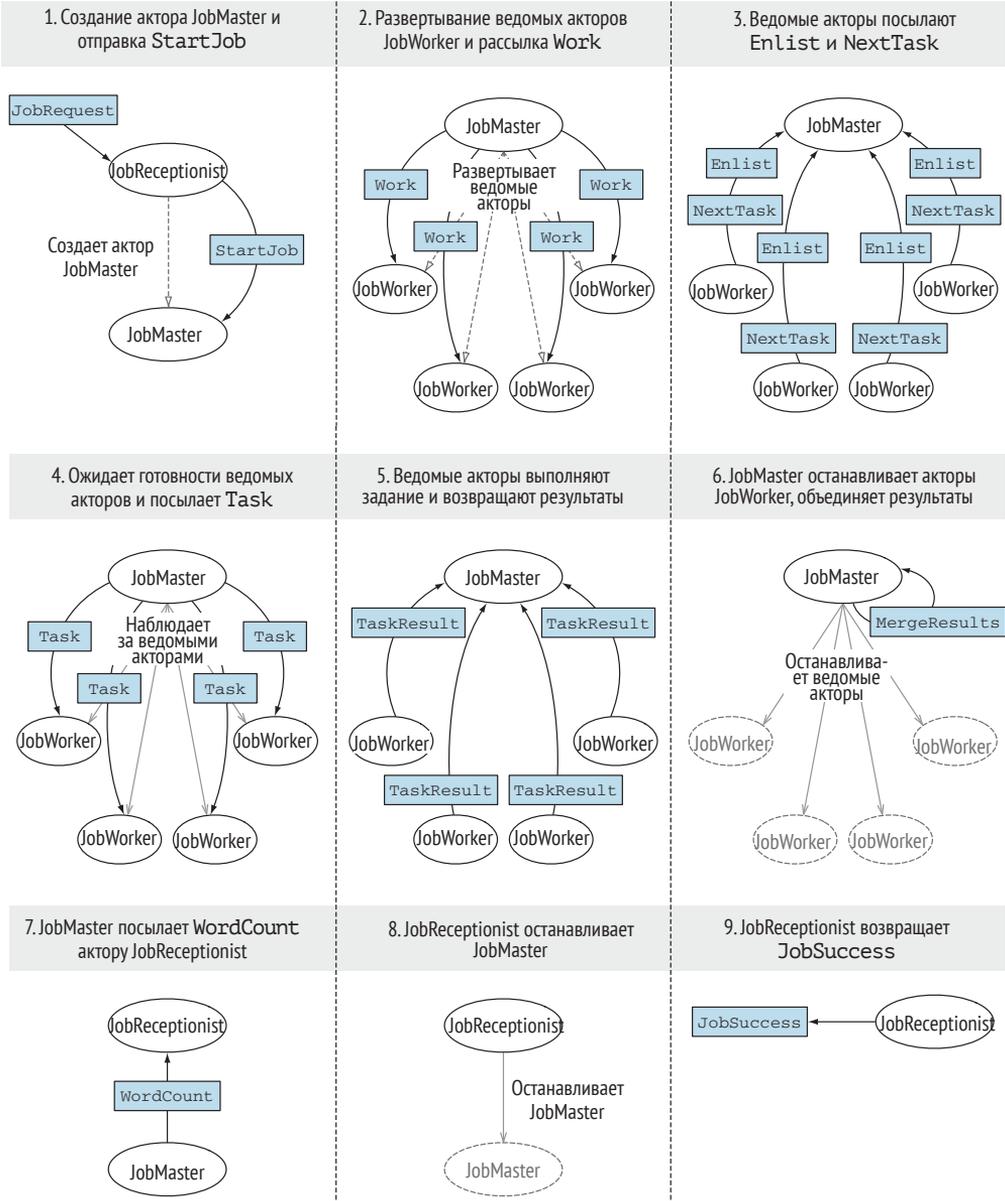


Рис. 14.12. Обработка задания

Каждый актер `JobWorker` получает сообщение `Task` с фрагментом текста, разбивает его на слова, подсчитывает количество вхождений для каждого из них и возвращает сообщение `TaskResult` со словарем `Map`, содержащим счетчики слов. Актор `JobMaster` принимает сообщения `TaskResult` и объединяет словари из них, складывая счетчики одинаковых слов, фактически выполняя шаг свертки. В заключение актору `JobReceptionist` возвращается результат `WordCount`.

### Некоторые замечания относительно примера

Мы постарались сохранить этот пример максимально простым. Актор `JobMaster` сохраняет в памяти промежуточные результаты и все данные, пересылаемые между актерами.

Если вам требуется организовать пакетную обработку очень больших объемов данных, вы должны позаботиться об их сохранении на том же сервере, где осуществляется их обработка, и о сборке результатов, не уместящихся в памяти. В системах на основе Hadoop, например, это означает сохранение данных в распределенной файловой системе HDFS (Hadoop Distributed File System) перед обработкой и запись результатов обратно в HDFS. В нашем примере мы просто пересылаем задания между узлами кластера. Этап свертки, на котором происходит объединение результатов, для простоты будет выполняться одним актором вместо нескольких, действующих параллельно.

Мы могли использовать все эти механизмы в нашем примере, но размер главы не позволяет сделать это. Однако мы покажем, как увеличить надежность обработки заданий, и вы сможете использовать наш пример как отправную точку для своих изысканий.

В следующих разделах мы рассмотрим все эти шаги по отдельности. Сначала создадим кластер, а затем распределим работу между ведомыми актерами. После этого посмотрим, как повысить надежность обработки задания, включая его перезапуск в случае выхода из строя одного из узлов. В заключение мы займемся вопросами тестирования кластера.

#### 14.3.1. Запуск кластера

Вы можете собрать пример в каталоге `chapter-cluster` командой `sbt assembly`. Она создаст файл `words-node.jar` в каталоге `target`. JAR-файл содержит три разных конфигурационных файла: один для настройки ведущего узла, один для настройки ведомого узла и один для настройки опорного узла. В листинге 14.11 показано, как запустить один опорный, один ведущий и два ведомых узла на локальной машине.

**Листинг 14.11.** Запуск узлов

```

java -DPORT=2551 \
    -Dconfig.resource=/seed.conf \
    -jar target/words-node.jar
java -DPORT=2554 \
    -Dconfig.resource=/master.conf \
    -jar target/words-node.jar
java -DPORT=2555 \
    -Dconfig.resource=/worker.conf \
    -jar target/words-node.jar
java -DPORT=2556 \
    -Dconfig.resource=/worker.conf \
    -jar target/words-node.jar

```

Здесь запускается только один опорный узел, чего пока вполне достаточно. (Файлы *master.conf* и *worker.conf* определяют список локальных опорных узлов, прослушивающих адрес 127.0.0.1 и порты 2551, 2552 и 2553; так как 2551 – это порт первого опорного узла в списке, все работает прекрасно.) Список опорных узлов также можно настроить с использованием системного свойства, если понадобится запустить опорные узлы на разных хостах и с другими номерами портов.

**ПЕРЕОПРЕДЕЛЕНИЕ СПИСКА ОПОРНЫХ УЗЛОВ ИЗ КОМАНДНОЙ СТРОКИ.** Список опорных узлов можно переопределить, передав параметр командной строки `-Dakka.cluster.seed-nodes.[n]=[seednode]`, где `[n]` нужно заменить номером позиции в списке (счет начинается с 0), а `[seednode]` – значением опорного узла.

Ведущий актер ничего не сможет сделать без ведомых, поэтому запускать актер `JobReceptionist` имеет смысл только при наличии некоторого минимального количества ведомых узлов. В конфигурации кластера можно указать минимальное количество ведомых членов с определенной ролью. В листинге 14.12 показан соответствующий фрагмент файла *master.conf*.

**Листинг 14.12.** Настройка минимального количества ведомых узлов для события `MemberUp`

```

role {
    worker.min-nr-of-members = 2
}

```

Конфигурация ведущего узла указывает, что в кластере должно иметься не менее двух ведомых узлов. Модуль `Cluster` поддерживает метод `regis-`

terOnMemberUp для регистрации функции, которая должна вызываться после запуска узла – в данном случае ведущего узла, – и получает минимальное количество ведомых узлов. Вызов функции происходит сразу после присоединения ведущего узла к кластеру и когда в кластере запущено два или более ведомых узлов. В листинге 14.13 представлен класс Main, используемый для запуска узлов всех типов в кластере words.

**Листинг 14.13.** Запуск узлов в кластере

```
object Main extends App {
  val config = ConfigFactory.load()
  val system = ActorSystem("words", config)

  println(s"Starting node with roles: ${Cluster(system).selfRoles}")

  val roles = system.settings
    .config
    .getStringList("akka.cluster.roles")
  if(roles.contains("master")) {
    Cluster(system).registerOnMemberUp {
      val receptionist = system.actorOf(Props[JobReceptionist],
        "receptionist")
      println("Master node is ready.")
    }
  }
}
```

Запустить JobReceptionist, только если узел имеет роль "master"

Регистрация блока кода для выполнения при присоединении очередного члена

Создавать JobReceptionist, только если в кластере имеется хотя бы два ведомых узла

Ведомый узел не должен запускать никаких акторов; акторы JobWorker запускаются по требованию, как будет показано в следующем разделе. Для развертывания акторов JobWorker и взаимодействий с ними мы будем использовать маршрутизатор.

### 14.3.2. Распределение заданий с использованием маршрутизаторов

Актор JobMaster должен сначала создать ведомые акторы JobWorker и затем разослать им сообщение work. Маршрутизаторы используются в кластере точно так же, как в локальной системе. Нужно только изменить порядок создания маршрутизаторов. Для взаимодействий с акторами JobWorker мы будем использовать маршрутизатор BroadcastPoolRouterConfig. Pool-версия – это RouterConfig, создающий акторы, тогда как Group-версия использует уже имеющиеся акторы, как уже говорилось в главе 9. В данном случае нам требуется динамически создавать акторы JobWorker и останавливать их после выполнения задания, поэтому Pool-версия является лучшим выбором. Для создания маршрутизатора актор JobMaster использует

отдельный трейт. Это обстоятельство пригодится нам позднее, при тестировании. Данный трейт показан в листинге 14.14 (его определение также можно найти в файле `src/main/scala/aia/cluster/words/JobMaster.scala`).

**Листинг 14.14.** Создание маршрутизатора `BroadcastPool` для кластера

```

trait CreateWorkerRouter { this: Actor =>
  def createWorkerRouter: ActorRef = {
    context.actorOf(
      ClusterRouterPool(BroadcastPool(10),
        ClusterRouterPoolSettings(
          totalInstances = 1000,
          maxInstancesPerNode = 20,
          allowLocalRoutees = false,
          useRole = None
        )
      ).props(Props[JobWorker]),
      name = "worker-router")
  }
}

```

← Должен подмешиваться в актор

← ClusterRouterPool принимает Pool

← Максимальное количество ведомых акторов в кластере

← Максимальное количество ведомых акторов на узел

← Не создавать локальные маршруты. Акторы JobWorker должны создаваться только на других узлах

← Маршрутизация будет выполняться для узлов с этой ролью

← Создание акторов JobWorker стандартным способом с Props

**КОНФИГУРАЦИЯ МАРШРУТИЗАТОРА.** В нашем примере `JobMaster` создается динамически для каждого задания, поэтому маршрутизатор тоже будет создаваться динамически, именно поэтому данная операция выполняется в коде. Однако развернуть маршрутизатор можно также с использованием конфигурации, как было описано в главе 9. В конфигурацию развертывания можно добавить секцию `cluster`, чтобы разрешить работу маршрутизатора в кластере, и настроить для `ClusterRouter` такие параметры, как `use-role` и `allow-local-routees`.

Трейт `CreateWorkerRouter` реализует только одну операцию: создание маршрутизатора для ведомых акторов. Создание маршрутизатора для кластера почти не отличается от создания обычного маршрутизатора. Достаточно лишь передать `ClusterRouterPool`, который можно использовать с любым поддерживаемым пулом – `BroadcastPool`, `RoundRobinPool`, `ConsistentHashingPool` и т. д. `ClusterRouterPoolSettings` управляет количеством создаваемых экземпляров `JobWorker`. Они будут добавляться в присоединенные ведомые узлы до достижения порогового значения `totalInstances`. В конфигурации, представленной в листинге 14.14, к кластеру может присоединиться до 50 узлов, прежде чем маршрутизатор прекратит создание новых акторов `JobWorker`. Когда создается актор `JobMaster`, он создает маршрутизатор, как показано в листинге 14.15, и использует его для отправки сообщений ведомым акторам, как показано на рис. 14.13.

**Листинг 14.15.** Использование маршрутизатора для рассылки сообщений Work

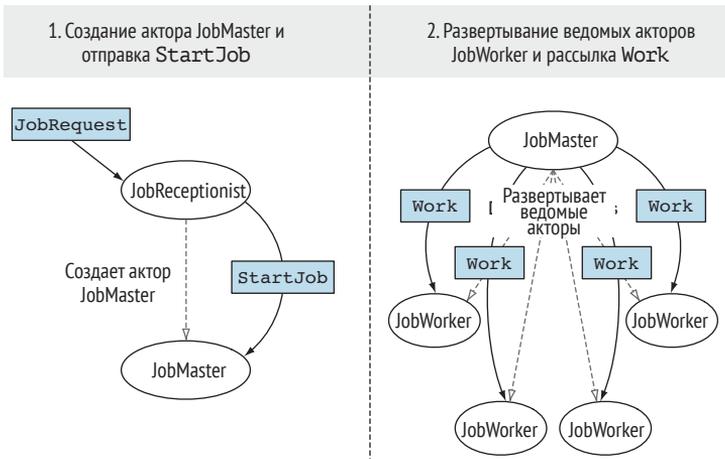
```

class JobMaster extends Actor
  with ActorLogging
  with CreateWorkerRouter {
// в теле актора JobMaster..
val router = createWorkerRouter
def receive = idle

def idle: Receive = {
  case StartJob(jobName, text) =>
    textParts = text.grouped(10).toVector
    val cancel = system.scheduler.schedule(0 millis,
      1000 millis,
      router,
      Work(jobName, self))
    become(working(jobName, sender(), cancel))
}
// дальше следует другой код

```

Подмешать трейт CreateWorkerRouter  
 Создать маршрутизатор  
 Планирование сообщения для маршрутизатора

**Рис. 14.13.** Развертывание актора JobWorker и рассылка сообщений Work

Фрагмент кода в листинге 14.15 иллюстрирует еще один аспект. Актор JobMaster – это конечный автомат и использует become для перехода между состояниями. В момент запуска он оказывается в состоянии простоя и пребывает в нем, пока JobReceptionist не пошлет сообщение StartJob. Когда JobMaster получает это сообщение, он разбивает текст на фрагменты по 10 строк и планирует рассылку сообщений Work без задержки ведомым акторам. Затем он переходит в состояние Working и начинает обработку ответов от ведомых акторов. Сообщения Work рассылаются с помощью планировщика на случай, если после запуска задания к кластеру присо-

единятся новые ведомые узлы. Конечные автоматы делают координацию распределенных задач более понятной. Фактически оба актора – `JobMaster` и `JobWorker` – являются конечными автоматами.

Существует также `ClusterRouterGroup`, который настраивается подобно `ClusterRouterPool`. Акторы, обслуживаемые этим маршрутизатором, должны быть запущены до того, как маршрутизатор `ClusterRouterGroup` начнет рассылку сообщений. Кластер `words` может иметь несколько узлов с ролью ведущего. На каждом узле с ролью ведущего запускается актор `JobReceptionist`. Если потребуется рассылать сообщения всем акторам `JobReceptionist`, например для отмены всех текущих выполняемых заданий, можно воспользоваться маршрутизатором `ClusterRouterGroup`. В листинге 14.16 показано, как создать маршрутизатор, отыскивающий акторы `JobReceptionist` на узлах с ролью ведущего (пример можно найти в файле `src/main/scala/aia/cluster/words/ReceptionistRouterLookup.scala`).

**Листинг 14.16.** Рассылка сообщений всем акторам `JobReceptionist` в кластере

```
val receptionistRouter = context.actorOf(
  ClusterRouterGroup(
    BroadcastGroup( Nil ),
    ClusterRouterGroupSettings(
      totalInstances = 100,
      routeesPaths = List("/user/receptionist"),
      allowLocalRoutees = true,
      useRole = Some("master")
    )
  ).props(),
  name = "receptionist-router")
```

← `ClusterRouterGroup`

← Количество экземпляров переопределяется настройками группы

← Путь для поиска (верхнего уровня) акторов `JobReceptionist`

← Обслуживать только узлы с ролью ведущего

Теперь вы знаете, как `JobMaster` рассылает сообщения `Work` акторам `JobWorker`. В следующем разделе мы посмотрим, как акторы `JobWorker` запрашивают дополнительную работу у актора `JobMaster`, пока задание не будет выполнено полностью, и как актор восстанавливается после ошибок, возникающих в процессе обработки.

### 14.3.3. Надежная обработка заданий

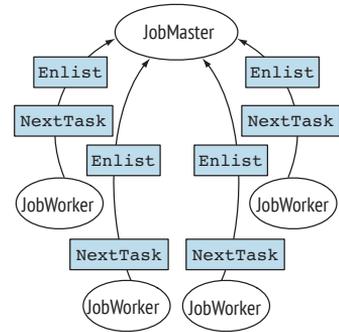
Актор `JobWorker` получает от `JobMaster` сообщение `Work` и возвращает сообщение `Enlist`, подтверждая свое желание включиться в работу. Сразу вслед за этим он посылает сообщение `NextTask`, запрашивая первый фрагмент задания для обработки. На рис. 14.14 показано, как выглядит поток сообщений. В листинге 14.17 демонстрируется, как `JobWorker` переходит из состояния простоя в состояние готовности.

Актор `JobWorker` извещает `JobMaster`, что желает включиться в работу, посылая сообщение `Enlist`, которое содержит ссылку `ActorRef` на `JobWor-`

**Рис. 14.14.** JobWorker подтверждает желание включиться в работу и запрашивает задание

3. Ведомые акторы посылают Enlist и NextTask

ker, которую JobMaster сможет использовать позднее. JobMaster наблюдает за всеми акторами JobWorker, включившимися в работу, чтобы определить момент, когда какой-то из них потерпит аварию, и остановить все акторы JobWorker по завершении задания.



**Листинг 14.17.** JobWorker переходит из состояния простоя в состояние готовности

```

def receive = idle      ← Запускается в состоянии простоя

def idle: Receive = {
  case Work(jobName, master) =>      ← Прием сообщения Work
    become(enlisted(jobName, master))  ← Переход в состояние готовности

  log.info(s"Enlisted, will start working for job '${jobName}'.")

  master ! Enlist(self)  ← Отправка сообщения Enlist ведущему
  master ! NextTask      ← Отправка сообщения NextTask ведущему

  watch(master)
  setReceiveTimeout(30 seconds)

def enlisted(jobName:String, master:ActorRef): Receive = {
  case ReceiveTimeout =>
    master ! NextTask
  case Terminated(master) =>
    setReceiveTimeout(Duration.Undefined)
    log.error(s"Master terminated for ${jobName}, stopping self.")
    stop(self)
  ...
}

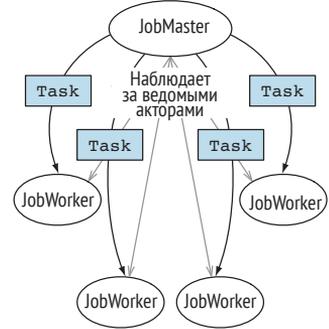
```

Актор JobWorker переходит в состояние Enlisted (готовности) и ожидает сообщения Task от ведущего для обработки. Затем он начинает наблюдение за JobMaster и устанавливает ReceiveTimeout. Если JobWorker не получит сообщений в течение ReceiveTimeout, он повторно пошлет запрос NextTask ведущему актору JobMaster, как показано в функции enlisted, в обработке ReceiveTimeout. Если актор JobMaster останавливается, JobWorker тоже

**Рис 14.15.** JobMaster посылает сообщения Task актерам JobWorker и наблюдает за ними

4. Ожидает готовности ведомых акторов и посылает Task

останавливает себя. Взглянув на обработчик сообщения Terminated, можно убедиться, что он не содержит ничего необычного; механизм DeathWatch действует точно так же, как в обычных системах акторов, не являющихся кластерами. Действия актора JobMaster, находящегося в состоянии работы, показаны на рис. 14.15 и в листинге 14.18.



**Листинг 14.18.** JobMaster составляет список акторов JobWorker, готовых к работе, и посылает им сообщения Task

```

// внутри JobMaster..
import SupervisorStrategy._
override def supervisorStrategy: SupervisorStrategy = stoppingStrategy

def working(jobName:String,
            receptionist:ActorRef,
            cancellable:Cancellable): Receive = {
  case Enlist(worker) =>
    watch(worker)
    workers = workers + worker

  case NextTask =>
    if(textParts.isEmpty) {
      sender() ! WorkLoadDepleted
    } else {
      sender() ! Task(textParts.head, self)
      workGiven = workGiven + 1
      textParts = textParts.tail
    }

  case ReceiveTimeout =>
    if(workers.isEmpty) {
      log.info(s"No workers responded in time. Cancelling $jobName.")
      stop(self)
    } else setReceiveTimeout(Duration.Undefined)

  case Terminated(worker) =>
    log.info(s"Worker $worker got terminated. Cancelling $jobName.")
    stop(self)
}
// дальше следует другой код..

```

**Использовать StoppingStrategy**

**Начать наблюдение за ведомым актором, включившимся в работу, и добавить его в список workers**

**При получении запроса NextTask от ведомого послать в ответ сообщение Task**

**JobMaster останавливается, если за период ReceiveTimeout не зарегистрировалось ни одного ведомого актора**

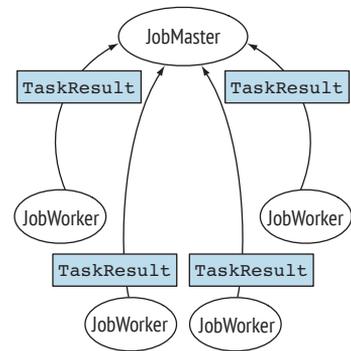
**JobMaster останавливается, если какой-то из ведомых JobWorker потерпел аварию**

Как видно в листинге 14.18, актор `JobMaster` регистрирует все акторы `JobWorker`, заявившие об участии в работе, и начинает наблюдение за ними. Если задание выполнено, в ответ `JobMaster` посылает ведомому актору сообщение `WorkLoadDepleted`.

Кроме того, `JobMaster` использует `ReceiveTimeout` (устанавливается в момент запуска задания) на тот случай, если ни один из акторов `JobWorker` не сообщит о готовности включиться в работу. После истечения тайм-аута `ReceiveTimeout` актор `JobMaster` останавливается. Он также останавливается, если неожиданно остановился один из ведомых акторов `JobWorker`. Кроме того, `JobMaster` является супервизором всех зарегистрировавшихся акторов `JobWorker` (маршрутизатор автоматически будет передавать сообщения о проблемах на уровень выше). Стратегия `StoppingStrategy` гарантирует автоматическую остановку аварийных акторов `JobWorker`, что вызовет отправку сообщения `Terminated` ведущему актору `JobMaster`, наблюдающему за ним.

Актор `JobWorker` принимает сообщение `Task`, выполняет задание в нем, возвращает `TaskResult` и запрашивает новое задание, посылая сообщение `NextTask`. На рис. 14.16 и в листинге 14.19 показано, как действует `JobWorker`, находясь в состоянии готовности.

5. Ведомые акторы выполняют задание и возвращают результаты



**Рис. 14.16.** `JobWorker` обрабатывает сообщение `Task` и возвращает `TaskResult`

#### Листинг 14.19. `JobWorker` обрабатывает `Task` и возвращает `TaskResult`

```

def enlisted(jobName:String, master:ActorRef): Receive = {
  case ReceiveTimeout =>
    master ! NextTask

  case Task(textPart, master) =>
    val countMap = processTask(textPart)    ← Обработка задания
    processed = processed + 1
    master ! TaskResult(countMap)          ← Отправка результата актору JobMaster
    master ! NextTask                       ← Запрос нового задания

  case WorkLoadDepleted =>
    log.info(s"Work load ${jobName} is depleted, retiring...")
    setReceiveTimeout(Duration.Undefined) ← Сброс тайм-аута ReceiveTimeout и
    become(retired(jobName))              остановка; задание выполнено

  case Terminated(master) =>

```

```

    setReceiveTimeout(Duration.Undefined)
    log.error(s"Master terminated for ${jobName}, stopping self.")
    stop(self)
}

def retired(jobName: String): Receive = { ← Завершение работы
  case Terminated(master) =>
    log.error(s"Master terminated for ${jobName}, stopping self.")
    stop(self)
  case _ => log.error("I'm retired.")
} // определение processTask следует далее...

```

Такой подход, как в данном примере, когда ведомый запрашивает у ведущего очередную порцию данных для обработки, имеет свои преимущества. Самое главное из них – автоматическое равномерное распределение нагрузки между актерами `JobWorker`. Актор `JobWorker`, обладающий большим объемом вычислительных ресурсов, будет запрашивать новые данные для обработки чаще, чем актер `JobWorker`, действующий в высоконагруженной системе. Если бы `JobMaster` принудительно рассылал задания всем актерам `JobWorker` по очереди, вполне могла бы сложиться ситуация, когда одни ведомые акторы не успевают справляться со своими порциями данных, а другие простаивают.

### ADAPTIVELOADBALANCINGPOOL И ADAPTIVELOADBALANCINGGROUP.

Есть альтернативное решение распределения нагрузки между ведомыми узлами. Маршрутизаторы `AdaptiveLoadBalancingPool` и `AdaptiveLoadBalancingGroup` используют показатели, вычисляемые кластером, и на их основе решают, какой из узлов является лучшим кандидатом для отправки следующего сообщения. Настроить показатели можно с использованием *JMX* или *Hyperic Sigar*<sup>1</sup>.

Получая сообщения `TaskResult`, `JobMaster` объединяет результаты в них. На рис. 14.17 и в листинге 14.20 показано, как `JobMaster` переходит в состояние завершения задания, когда все данные будут обработаны, и посылает сообщение `WordCount`.

**Листинг 14.20.** `JobMaster` сохраняет и объединяет промежуточные результаты, завершает задание

```

def working(jobName:String,
            receptionist:ActorRef,
            cancellable:Cancellable): Receive = {
  ...

```

<sup>1</sup> Больше информации о `Hyperic Sigar` можно найти по адресу: <http://sigar.hyperic.com/>.

```

case TaskResult(countMap) =>
  intermediateResult = intermediateResult :+ countMap
  workReceived = workReceived + 1

  if(textParts.isEmpty && workGiven == workReceived) {
    cancellable.cancel()
    become(finishing(jobName, receptionist, workers))
    setReceiveTimeout(Duration.Undefined)
    self ! MergeResults
  }
}
...
def finishing(jobName: String,
             receptionist: ActorRef,
             workers: Set[ActorRef]): Receive = {
  case MergeResults =>
    val mergedMap = merge()
    workers.foreach(stop(_))
    receptionist ! WordCount(jobName, mergedMap)

  case Terminated(worker) =>
    log.info(s"Job $jobName is finishing, stopping.")
}
...

```

← Сохранение промежуточных результатов, поступающих от акторов JobWorker

← Помните запланированное задание, которое послало сообщение Work? Теперь пришло время отменить его

← Отправка MergeResults себе, чтобы объединить результаты в состоянии завершения

← Переход в состояние завершения

← Обработка сообщения MergeResults, отправленного самому себе

← Объединение результатов

← Отправка результатов актору JobReceptionist

← Остановка всех ведомых акторов; задание выполнено

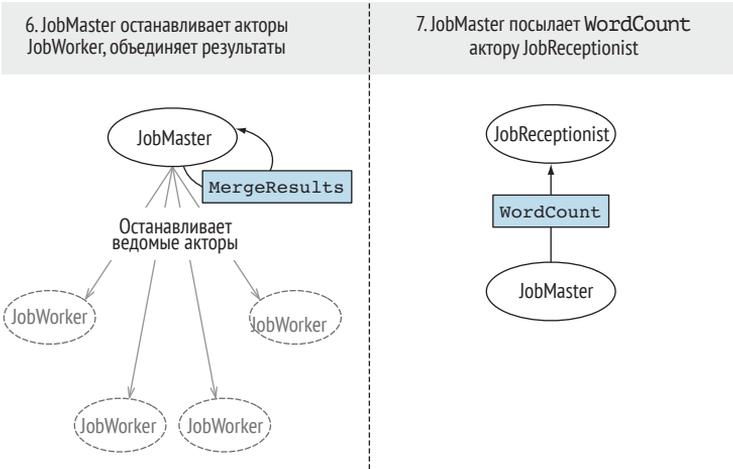


Рис. 14.17. JobMaster объединяет промежуточные результаты и возвращает WordCount

Наконец, JobReceptionist принимает сообщение WordCount и останавливает актор JobMaster, завершивший обработку. JobWorker автоматически завершается, встретив слово FAIL, имитируя ошибку и возбуждая исключение. Актор JobReceptionist наблюдает за созданными им акторами Job-

Master. Он также использует стратегию `StoppingStrategy` на случай аварии `JobMaster`. Рассмотрим иерархию наблюдения в этой системе акторов и как с помощью механизма `DeathWatch` определяются аварийные ситуации (см. рис. 14.18).

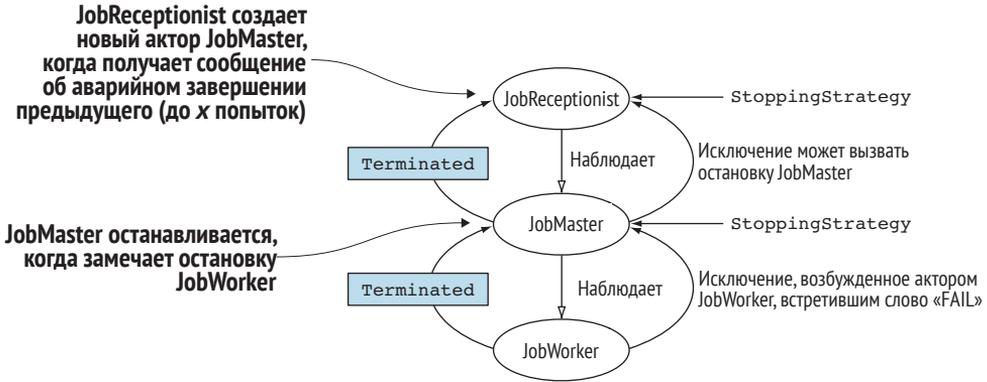


Рис. 14.18. Иерархия наблюдения в системе акторов `words`

Посылая сообщения акторам, мы устанавливаем тайм-аут `ReceiveTimeout`, тем самым определяя факт неполучения сообщений акторами и получая возможность предпринять какие-то действия. Актор `JobReceptionist` следит за заданиями, которые он раздает. Получив сообщение `Terminated`, он проверяет признак выполнения задания. Если задание не выполнено, он посылает самому себе первоначальное сообщение `JobRequest`, чтобы запустить процесс обработки с самого начала. Для имитации исправления проблем `JobReceptionist` удаляет слово `FAIL` из текста после заданного числа попыток, как показано далее.

**Листинг 14.21.** `JobReceptionist` повторяет запрос `JobRequest` в случае аварии `JobMaster`

```
case Terminated(jobMaster) =>
  jobs.find(_.jobMaster == jobMaster).foreach { failedJob =>
    log.error(s"$jobMaster terminated before finishing job.")

    val name = failedJob.name
    log.error(s"Job ${name} failed.")
    val nrOfRetries = retries.getOrElse(name, 0)

    if(maxRetries > nrOfRetries) {
      if(nrOfRetries == maxRetries - 1) {
        // Сымитировать разрешение проблемы
        // после максимального числа попыток

        val text = failedJob.text.filterNot(_.contains("FAIL"))
        self.tell(JobRequest(name, text), failedJob.respondTo)
```

← Отправить `JobRequest` для обработки без имитации ошибки

```

    } else self.tell(JobRequest(name, failedJob.text),
                     failedJob.respondTo)
    updateRetries
  }
}
}

```

← Повторная отправка JobRequest

Мы вернемся к имитации ошибок в следующем разделе, где будем производить тестирование кластера words.

### 14.3.4. Тестирование кластера

Для тестирования можно использовать плагин `sbt-multi-jvm` и модуль `multi-node-testkit`, а также модуль `akka-remote`. Кроме того, протестировать акторы можно локально, если изолировать создание акторов и маршрутизаторов в трейтах. В листинге 14.22 показаны тестовые версии `JobReceptionist` и `JobMaster` (которые также можно найти в файле `src/test/scala/aia/cluster/words/LocalWordsSpec.scala`). Трейты используются для переопределения создания маршрутизаторов и ведущих акторов `JobMaster`.

**Листинг 14.22.** Тестовые версии акторов

```

trait CreateLocalWorkerRouter extends CreateWorkerRouter {
  def context: ActorContext
  override def createWorkerRouter: ActorRef = {
    context.actorOf(BroadcastPool(5).props(Props[JobWorker]),
                   "worker-router")
  }
}

class TestJobMaster extends JobMaster
  with CreateLocalWorkerRouter

class TestReceptionist extends JobReceptionist
  with CreateMaster {
  override def createMaster(name: String): ActorRef = {
    context.actorOf(Props[TestJobMaster], name)
  }
}

```

← Этот трейт определяет контекст, которым в данном случае является класс JobMaster

← Создает обычный, некластерный маршрутизатор

← Создает тестовую версию JobMaster и переопределяет создание маршрутизатора

← Создает тестовую версию JobMaster

← Создает тестовую версию JobReceptionist, переопределяет создание JobMaster

Локальный тест показан в листинге 14.23. Как видите, в нем нет ничего необычного: он посылает `JobRequest` актору `JobReceptionist`. Ответ проверяется с помощью `expectMsg` (`ImplicitSender` автоматически делает `testActor` отправителем всех сообщений, как описывалось в главе 3).

**Листинг 14.23.** Локальный тест

```

class LocalWordsSpec extends TestKit(ActorSystem("test"))
    with WordSpec
    with MustMatchers
    with StopSystemAfterAll
    with ImplicitSender {

  val receptionist = system.actorOf(Props[TestReceptionist],
                                     JobReceptionist.name)

  val words = List("this is a test ",
                   "this is a test",
                   "this is",
                   "this")

  "The words system" must {
    "count the occurrence of words in a text" in {
      receptionist ! JobRequest("test2", words)
      expectMsg(JobSuccess("test2", Map("this" -> 4,
                                         "is" -> 3,
                                         "a" -> 2,
                                         "test" -> 2)))

      expectNoMsg
    }
    ...
    "continue to process a job with intermittent failures" in {
      val wordsWithFail = List("this", "is", "a", "test", "FAIL!")
      receptionist ! JobRequest("test4", wordsWithFail)
      expectMsg(JobSuccess("test4", Map("this" -> 1,
                                         "is" -> 1,
                                         "a" -> 1,
                                         "test" -> 1)))

      expectNoMsg
    }
  }
}

```

Создает тестовую версию  
JobReceptionist

Встретив слово «FAIL»  
в тексте, JobWorker  
сымитирует ошибку,  
возбудив исключение

Тест `multi-node` не изменяет процедуры создания акторов и маршрутизатора. Для тестирования кластера нужно сначала создать `MultiNodeConfig`, как показано в листинге 14.24, который также можно найти в файле `src/multi-jvm/scala/aia/cluster/words/WordsClusterSpecConfig.scala`.

**Листинг 14.24.** Конфигурация `MultiNode`

```

import akka.remote.testkit.MultiNodeConfig
import com.typesafe.config.ConfigFactory

object WordsClusterSpecConfig extends MultiNodeConfig {

```

```

val seed = role("seed")
val master = role("master")
val worker1 = role("worker-1")
val worker2 = role("worker-2")

commonConfig(ConfigFactory.parseString("""
  akka.actor.provider="akka.cluster.ClusterActorRefProvider"
  """))
}

```

← Определение ролей в тесте

← Передача конфигурации теста. ClusterActorRefProvider гарантирует инициализацию кластера. Вы можете добавить сюда дополнительные конфигурационные параметры для всех узлов, участвующих в тесте

Как рассказывалось в главе 6, MultiNodeConfig используется в MultiNodeSpec. WordsClusterSpecConfig используется в WordsClusterSpec, как показано в листинге 14.25 (который также можно найти в файле `src/multi-jvm/scala/aia/cluster/words/WordsClusterSpec.scala`).

#### Листинг 14.25. Тестирование кластера words

```

class WordsClusterSpecMultiJvmNode1 extends WordsClusterSpec
class WordsClusterSpecMultiJvmNode2 extends WordsClusterSpec
class WordsClusterSpecMultiJvmNode3 extends WordsClusterSpec
class WordsClusterSpecMultiJvmNode4 extends WordsClusterSpec

class WordsClusterSpec extends MultiNodeSpec(WordsClusterSpecConfig)
with STMultiNodeSpec with ImplicitSender {

  import WordsClusterSpecConfig._

  def initialParticipants = roles.size

  val seedAddress = node(seed).address
  val masterAddress = node(master).address
  val worker1Address = node(worker1).address
  val worker2Address = node(worker2).address

  muteDeadLetters(classOf[Any])(system)
  "A Words cluster" must {

    "form the cluster" in within(10 seconds) {

      Cluster(system).subscribe(testActor, classOf[MemberUp])
      expectMsgClass(classOf[CurrentClusterState])

      Cluster(system).join(seedAddress)

      receiveN(4).map { case MemberUp(m) => m.address }.toSet must be(
        Set(seedAddress, masterAddress, worker1Address, worker2Address))
    }
  }
}

```

← Все узлы в тесте наследуют один и тот же класс WordsClusterSpec

← Получение адресов всех узлов

← Присоединение опорного узла. В конфигурации не используется список опорных узлов, поэтому мы вручную запускаем опорный узел

← Подписывает актер testActor на служебные события в кластере

← Проверка присоединения всех узлов

```

Cluster(system).unsubscribe(testActor)

enterBarrier("cluster-up")
}

"execute a words job" in within(10 seconds) {
  runOn(master) {
    val receptionist = system.actorOf(Props[JobReceptionist],
                                      "receptionist")

    val text = List("some", "some very long text", "some long text")
    receptionist ! JobRequest("job-1", text)
    expectMsg(JobSuccess("job-1", Map("some" -> 3,
                                       "very" -> 1,
                                       "long" -> 2,
                                       "text" -> 2)))
  }
  enterBarrier("job-done")
}
...
}
}

```

← Отправка задания и проверка результатов. Другие узлы просто вызывают enterBarrier

Как видите, сам тест не изменился, по сравнению с локальной версией. Кластерная версия лишь обеспечивает запуск кластера перед тестированием. Тестирование восстановления после ошибки здесь не показано, но оно реализовано точно так же, как было показано в листинге 14.23, где в текст для обработки добавлялось слово *FAIL*.

### Клиент кластера ClusterClient

Тест посылает сообщение JobRequest из ведущего узла. Возможно, вас волнует вопрос: как послать то же сообщение в кластер извне, то есть послать JobRequest одному из узлов кластера из-за его пределов? Модуль akka-contrib содержит пару шаблонов для работы с кластерами; один из них ClusterClient – актор, инициализируемый списком контактов (например, опорных узлов), которые пересылают сообщения акторам в кластере с помощью акторов ClusterRecipient на каждом узле.

На этом мы завершаем обсуждение темы тестирования акторов в кластере. Мы только что показали несколько тестов; в реальной жизни вам наверняка придется столкнуться с большим разнообразием сценариев тестирования. Локальное тестирование позволяет проверить работу логики и правильность взаимодействий между акторами, тогда как multi-node-

testkit поможет вам отыскать проблемы, возникающие на этапе запуска кластера и в процессе его работы. Мы надеемся, нам удалось показать, что тестирование кластерной системы акторов не сильно отличается от тестирования локальных акторов и обычно не вызывает больших сложностей. Модуль multi-node-testkit с успехом можно использовать в интеграционных тестах, последовательно проверяющих процедуру инициализации кластера и что происходит, когда узлы выходят из строя.

## 14.4. В заключение

Организация динамического масштабирования простого приложения с применением расширения Cluster оказалась довольно простым делом. Присоединение к кластеру и выход из него выполняются легко, и работу этих операций можно проверить в консоли REPL – инструменте, дающем возможность поэкспериментировать и проверить, как действуют те или иные механизмы. Если вы следовали за примерами сеансов в консоли REPL, то могли сами убедиться, насколько надежно работает это расширение; аварийные ситуации на узлах правильно определяются, и механизм DeathWatch действует безотказно.

Сложности, связанные с переносом приложений в кластер, широко известны. Часто для этого требуется коренным образом менять подходы к администрированию и программированию. В этой главе вы увидели, что Akka делает такой переход менее болезненным и не требует переписывать код. В этой главе вы также:

- узнали, насколько легко можно сформировать кластер;
- познакомились с жизненным циклом конечного автомата;
- увидели, как объединить все компоненты в действующее приложение;
- научились тестировать логику кластера.

В заключительном примере речь шла не о подсчете слов, а об универсальном способе параллельной обработки заданий в кластере с помощью Akka. Мы использовали кластеризованные маршрутизаторы и группу простых сообщений, чтобы организовать совместную работу акторов и противостоять сбоям.

Наконец, мы смогли протестировать все в комплексе. Это еще одно большое достоинство Akka, к которому быстро привыкаешь. Возможность модульного тестирования кластера уникальна. Она позволяет находить ошибки в приложениях до их развертывания в промышленном окружении. Акторы в кластере words хранят некоторую промежуточную информацию о задании, разбросанную между узлами. Мы организовали повторную отправку запроса JobRequest, хранящегося в JobReceptionist, при

остановке ведущих или ведомых акторов. Однако такое решение не позволяет восстановиться и завершить задание в случае сбоя `JobReceptionist`, потому что в этом случае данные в запросе `JobRequest` будут утеряны. В следующей главе мы посмотрим, как можно восстанавливать состояние акторов из некоторого долговременного хранилища с помощью модуля `akka-persistence`.

# Глава 15

## Хранимые акторы

В этой главе:

- хранение последовательности событий для увеличения надежности;
- запись и восстановление состояния хранимых акторов;
- организация кластера на основе хранимых акторов.

Состояние актора, хранящееся в оперативной памяти, теряется, когда актор останавливается или перезапускается или когда останавливается или перезапускается вся система акторов. В этой главе мы расскажем, как обеспечить сохранность этого состояния с помощью модуля `akka-persistence`.

Очень часто для создания, извлечения, изменения и удаления записей используются базы данных (эти операции также называют CRUD-операциями). Записи в базе данных нередко используются для хранения текущего состояния системы. В этом случае база данных действует как контейнер.

Неудивительно, что разработчики Akka также включили поддержку хранимого состояния в свой фреймворк. Для организации хранения состояния в Akka выбрана технология под названием *Event Sourcing* (история событий), и ей будет посвящен первый раздел этой главы. В нем вы узнаете, как можно хранить изменения состояний в виде последовательности неизменяемых событий в журнале базы данных.

Затем мы займемся исследованием хранимого актора `PersistentActor`. Хранимый актор упрощает запись своего состояния в виде событий и его восстановление после аварии или перезапуска.

Модули `akka-cluster` и `akka-persistence` можно использовать вместе для создания кластерных приложений, которые продолжают нормальную работу даже после аварии или замены нескольких узлов. Далее мы рассмотрим два расширения для кластеров – *cluster singleton* и *cluster sharding*, – ко-

торые оба можно использовать для запуска хранимых акторов в кластере Akka.

Но сначала займемся историей событий (Event Sourcing), технологией, лежащей в основе механизма хранения в Akka. Если вы уже знакомы с этой технологией, можете пропустить следующий раздел и сразу перейти к разделу 15.1.3 «История событий для акторов».

### **Когда может понадобиться восстанавливать состояние актора?**

Заметим сразу, что нет смысла делать каждый актор хранимым только потому, что это возможно. Когда же действительно следует обеспечивать сохранность состояния актора?

Это во многом зависит от требований к системе и особенностей взаимодействий других систем с акторами.

Актор, жизненный цикл которого длится дольше, чем требуется для обработки одного сообщения, и который накапливает ценную информацию с течением времени, часто требует долговременного хранения состояния. Акторы такого рода обладают идентичностью – некоторым идентификатором, посредством которого к такому актору можно обратиться позднее. Примером может служить актор корзины покупателя; пользователь переходит между разделами веб-магазина в поисках нужных товаров, добавляет товары в корзину и удаляет их оттуда. Было бы неправильно показывать пустую корзину после неожиданной аварии или перезапуска актора. Другой случай – когда множество систем посылают сообщения актору, моделирующему конечный автомат согласования сообщений между системами. Актор может запоминать получаемые сообщения и конструировать некоторый контекст для окружающих систем. Примером может служить система, получающая заказы с веб-сайта и координирующая процесс получения товаров со склада, их доставку и учет путем интеграции с несколькими существующими службами. В случае перезапуска актор должен продолжить работу с последнего известного правильного состояния, чтобы подключенные системы могли продолжить работу в требуемом порядке.

Это лишь несколько примеров, когда необходимо обеспечить сохранность состояния актора. Не нужно стараться обеспечить сохранность состояния, если актор выполняет всю работу от начала до конца в ходе обработки единственного сообщения. Примером может служить актор, который обрабатывает HTTP-запросы, не имеющие состояния и содержащие всю необходимую информацию. В этом случае, если произойдет сбой, клиент может просто повторить HTTP-запрос.

## 15.1. Восстановление состояния с технологией Event Sourcing

Технология Event Sourcing (история событий) появилась довольно давно. Мы рассмотрим простой пример, демонстрирующий разницу между хранением истории событий и простых записей с информацией о состоянии, последнее из которых, как мы полагаем, уже хорошо вам знакомо. В этом примере мы реализуем калькулятор, который должен запоминать только последний вычисленный результат.

### 15.1.1. Обновление записей на месте

Обновление записей на месте – простейший прием, который применяется при использовании базы данных SQL для оперативной обработки транзакций (Online Transaction Processing, OLTP). На рис. 15.1 показано, как можно использовать SQL-инструкции `insert` и `update` для сохранения результатов вычислений в калькуляторе.

	SQL-инструкция	Таблица Results				
1. При запуске создать единственную запись:	<code>insert into results values (1, 0)</code>	<table border="1"> <thead> <tr> <th>id</th> <th>Result</th> </tr> </thead> <tbody> <tr> <td>1</td> <td>0</td> </tr> </tbody> </table>	id	Result	1	0
id	Result					
1	0					
2. Прибавить 1:	<code>update results set result = result + 1 where id = 1</code>	<table border="1"> <thead> <tr> <th>id</th> <th>Result</th> </tr> </thead> <tbody> <tr> <td>1</td> <td>1</td> </tr> </tbody> </table>	id	Result	1	1
id	Result					
1	1					
3. Умножить на 3:	<code>update results set result = result * 3 where id = 1</code>	<table border="1"> <thead> <tr> <th>id</th> <th>Result</th> </tr> </thead> <tbody> <tr> <td>1</td> <td>3</td> </tr> </tbody> </table>	id	Result	1	3
id	Result					
1	3					
4. Разделить на 4:	<code>update results set result = result / 4 where id = 1</code>	<table border="1"> <thead> <tr> <th>id</th> <th>Result</th> </tr> </thead> <tbody> <tr> <td>1</td> <td>0.75</td> </tr> </tbody> </table>	id	Result	1	0.75
id	Result					
1	0.75					

Рис. 15.1. Сохранение состояния с обновлениями

В момент запуска калькулятор вставляет одну запись; инструкция поиска уже существующей записи для простоты опущена. Все вычисления выполняются непосредственно в базе данных, с использованием инструкции `update` для изменения записи в таблице. (Вы можете полагать, что каждая инструкция `update` выполняется в своей транзакции.)

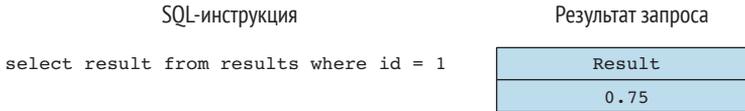
На рис. 15.1 видно, что калькулятор отображает результат 0.75. После перезапуска приложение запрашивает результат, как показано на рис. 15.2.

Все довольно просто; запись хранит только последний вычисленный результат. Однако в этом примере нет никакой возможности узнать, как пользователь пришел к результату 0.75 и какие промежуточные результаты имели место *после* окончания вычислений. (Мы не учитываем, что

теоретически можно наблюдать некоторые промежуточные результаты в SQL-запросах, выполняющихся параллельно с инструкциями update на низком уровне изоляции.)

Если вы захотите узнать, какие вычисления производил пользователь, вам придется сохранить их в отдельной таблице.

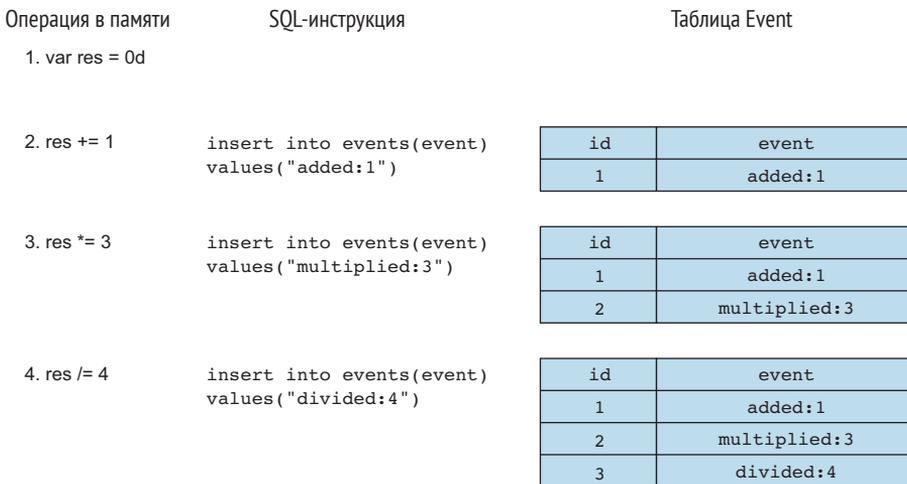
Все вычисления выполняются внутри SQL-инструкций. Каждое последующее вычисление зависит от предыдущего результата, хранящегося в записи.



**Рис. 15.2.** Калькулятор извлекает последнее известное состояние из базы данных

### 15.1.2. Сохранение состояния без изменения

Далее описывается идея технологии Event Sourcing (история событий). Вместо хранения последнего результата в одной записи можно записывать все успешно выполненные операции в журнал в виде *событий*. Событие должно в точности описывать происходящее; в данном случае мы будем сохранять имя операции и ее аргумент. И снова в примере используются простые SQL-инструкции; журнал организован как простая таблица в базе данных. На рис. 15.3 показаны инструкции, запоминающие всю историю вычислений.



**Рис. 15.3.** Сохранение всех успешных операций в виде событий

Вычисления выполняются в памяти; событие сохраняется после успешной операции. Столбец ID использует механизм последовательностей в базе данных, и его значение автоматически увеличивается на единицу в каждой новой записи. (Вы можете полагать, что вставка каждой записи выполняется в своей собственной транзакции или используется функция автоматического подтверждения.)

События описывают операции, выполненные успешно с момента начального состояния. Для хранения результата используется простая переменная. Каждое событие преобразуется в строку, содержащую имя события и аргумент, разделенные двоеточием.

Теперь посмотрим, как калькулятор восстанавливает последнее известное состояние на основе событий. Первоначально калькулятор инициализируется значением 0 и затем последовательно выполняет все сохраненные операции в том же порядке. Приложение читает события, преобразует их в операции, выполняет эти операции и в конечном счете получает значение 0.75, как показано на рис. 15.4.

1. `Select * from events order by id ASC`

id	event
1	added:1
2	multiplied:3
3	divided:4

2. Инициализация начальным состоянием

`var res = 0`

3. 

1	added:1
---	---------

Преобразование записи в операцию с аргументом, выполнение операции

`res +=1`

4. 

2	multiplied:3
---	--------------

Преобразование записи в операцию с аргументом, выполнение операции

`res *=3`

5. 

3	divided:4
---	-----------

Преобразование записи в операцию с аргументом, выполнение операции

`res /=4`

Вычисления завершаются с результатом `res = 0.75`, который отображается перед пользователем

`res = 0.75`

**Рис. 15.4.** Калькулятор повторно вычисляет последнее известное состояние на основе начального состояния и списка событий

Как видите, Event Sourcing (история событий) – простая технология. В следующем разделе мы посмотрим, как эту технологию можно использовать для сохранения состояния актора.

### Какое решение «проще»?

Решение в стиле CRUD прекрасно подходит для простого калькулятора. Оно требует меньше места в хранилище, и восстановление последнего результата осуществляется проще. Мы использовали приложение калькулятора, только чтобы подсветить различия между двумя подходами. Общедоступное изменяемое состояние – наш враг. Сохранение его в базе данных не решает проблемы. Взаимодействия с базой данных становятся все сложнее при неконтрольном использовании операций CRUD. Комбинация акторов и модуля akka-persistence дает простой способ реализации технологии Event Sourcing, не требуя от программиста больших усилий, как вы увидите сами далее в этой главе.

### 15.1.3. Event Sourcing для акторов

Одно из самых больших преимуществ Event Sourcing (истории событий) заключается в разделении операций записи и чтения с базой данных на два этапа. Чтение из журнала происходит только при восстановлении состояния хранимого актора. После восстановления актер продолжает действовать как обычно: обрабатывает сообщения и хранит свое состояние в памяти, одновременно сохраняя события в постоянном хранилище.

Журнал имеет простой интерфейс. Если не вдаваться в детали, он должен поддерживать только добавление сериализованных событий, а также чтение десериализованных событий с определенного места в журнале. События в журнале фактически являются неизменяемыми – их нельзя изменить после записи в таблицу. Отметим еще раз, что неизменяемость предпочтительнее изменяемости, когда речь заходит о сложностях организации параллельного доступа к данным.

Модуль akka-persistence определяет интерфейс журнала, который позволяет любому желающему написать свою реализацию. Существует даже специальный комплект инструментов TCK (Technology Compatibility Kit) для проверки совместимости с akka-persistence. События могут храниться в базе данных SQL, в базе данных NoSQL, во встраиваемой базе данных и даже в файлах, главное, чтобы реализация журнала могла добавлять события и читать их в одном и том же порядке. Вы можете найти уже готовые плагины журналов на странице сообщества Akka: <http://akka.io/community/>.

Event Sourcing имеет также свои недостатки. Наиболее очевидным из них является увеличенное потребление пространства в хранилище. Чтобы получить последнее известное состояние, все события, случившиеся с начального момента, должны быть восстановлены после аварии, и каждое

изменение состояния должно быть применено, а это может потребовать массу времени.

Создание *моментальных снимков* (snapshots) состояния актора, которое мы рассмотрим в разделе 15.2.3, может ослабить требование к пространству в хранилище и увеличить скорость восстановления состояния. Этот прием позволяет пропустить множество событий и обрабатывать только те, что случились с момента создания последнего мгновенного снимка.

Если состояние актора не уместается в памяти, мы неизбежно встаем перед другой проблемой. Технология фрагментации *Sharding*, позволяющая распределить состояние между серверами, делает возможным масштабирование пространства, необходимого для хранения этого состояния. Модуль *cluster-sharding*, который обсуждается в разделе 15.3.2, поддерживает стратегию фрагментации для акторов.

Можно с уверенностью сказать, что технология Event Sourcing требует некоторой формы сериализации событий. В некоторых случаях сериализация может выполняться автоматически; в других вам придется самому написать необходимый код. Представьте, что вы изменили событие в своем приложении (переименовали поле, например, или добавили новое поле), – как тогда обеспечить десериализацию старой и новой версий события из журнала? Версионирование сериализованных данных – сложная задача; мы обсудим несколько вариантов ее решения в разделе 15.2.4.

В действительности технология Event Sourcing дает лишь способ восстановления состояния из событий; она не решает проблему создания специализированных запросов. Хорошо известной областью применения специализированных запросов является репликация событий в систему, оптимизированную для анализа.

В следующем разделе мы начнем конструировать калькулятор, чтобы заложить основу для дальнейшего обсуждения. После этого мы рассмотрим более сложный пример онлайн-магазина.

## 15.2. Хранимые акторы

Прежде всего добавим необходимые зависимости в файл сборки.

### Листинг 15.1. Зависимости для akka-persistence

```
parallelExecution in Test := false
fork := true
libraryDependencies += {
  val akkaVersion = "2.4.9"
  Seq(
    "com.typesafe.akka" %% "akka-actor" % akkaVersion,
```

← Запретить параллельное тестирование, потому что для хранения журнала используется файл

← Порождать тесты в случае использования базы данных LevelDB

```

"com.typesafe.akka" %% "akka-persistence" % akkaVersion,
  // далее следуют другие зависимости для данного примера
)
}

```

← Зависимости для akka-persistence

В состав akka-persistence входят два плагина поддержки журналов в LevelDB (<https://github.com/google/leveldb>), которые должны использоваться только для тестирования: локальный и разделяемый. Локальный плагин может использоваться только одной системой акторов; разделяемый – несколькими, что очень удобно для тестирования хранимых акторов в кластере.

Плагины используют оригинальную библиотеку LevelDB (<https://github.com/fusesource/leveldbjni>) или версию LevelDB для Java (<https://github.com/dain/leveldb>). В листингах 15.2 и 15.3 показано, как настроить плагины для работы с версией LevelDB для Java.

**Листинг 15.2.** Настройка использования Java-библиотеки LevelDB локальным плагином журналирования

```
akka.persistence.journal.leveldb.native = off
```

**Листинг 15.3.** Настройка использования Java-библиотеки LevelDB разделяемым плагином журналирования

```
akka.persistence.journal.leveldb-shared.store.native = off
```

Параллельное тестирование возможно только при использовании оригинальной библиотеки; иначе попытка их ветвления будет вызывать ошибку.

Теперь, имея минимальный файл сборки, можно начинать конструировать калькулятор, чем мы и займемся в следующем разделе.

### 15.2.1. Хранимый актор

*Хранимый актор* действует в двух режимах: обрабатывает команды или восстанавливает последнее известное состояние из истории событий. *Команды* – это сообщения, посылаемые актору для выполнения некоторой логики; *события* служат доказательством, что актор выполнил логику без ошибок. Первое, что мы сделаем, – определим команды и события для актора-калькулятора. В листинге 15.4 определяются команды, которые калькулятор может выполнять, и события, возникающие после проверки допустимости команд калькулятором.

**Листинг 15.4.** Команды и события калькулятора

```
sealed trait Command
case object Clear extends Command
```

← Все команды наследуют трейт Command. Вывод в консоль упростит отображение наиболее релевантных сообщений

```

case class Add(value: Double) extends Command
case class Subtract(value: Double) extends Command
case class Divide(value: Double) extends Command
case class Multiply(value: Double) extends Command
case object PrintResult extends Command
case object GetResult extends Command

```

```

sealed trait Event
case object Reset extends Event
case class Added(value: Double) extends Event
case class Subtracted(value: Double) extends Event
case class Divided(value: Double) extends Event
case class Multiplied(value: Double) extends Event

```

← Все события наследуют  
трейт Event

Команды и события наследуют разные запечатанные трейты, команды наследуют `Command`, а события – `Event`. (Запечатанный (`sealed`) трейт позволяет компилятору Scala проверить полноту перечисления классов, наследующих трейт, в операторе сопоставления с шаблоном.) В `akka-persistence` имеется трейт `PersistentActor`, наследующий трейт `Actor`. Каждый хранимый актор должен иметь `persistenceId`, используемый для уникальной идентификации событий в журнале, принадлежащих каждому конкретному актору. (Без этого мы не смогли бы отличить события, записанные разными акторами.) Значение `persistenceId` автоматически передается в журнал при сохранении события. В нашем примере имеется только один актор-калькулятор; в листинге 15.5 показано, что он использует фиксированное значение для `persistenceId` – `my-calculator`, – которое определяется в `Calculator.name`. Состояние калькулятора хранится в `case`-классе `CalculationResult`, представленном в листинге 15.8.

#### Листинг 15.5. Расширение хранимого актора и определение `persistenceId`

```

class Calculator extends PersistentActor with ActorLogging {
  import Calculator._

  def persistenceId = Calculator.name

  var state = CalculationResult()
  // далее следует остальной код..

```

Хранимый актор должен иметь две процедуры приема сообщений – `receiveCommand` и `receiveRecover`, в отличие от предыдущей реализации. `receiveCommand` используется для обработки сообщений после восстановления актора, а `receiveRecover` принимает события и моментальные снимки во время восстановления.

Функция `receiveCommand`, как показано в листинге 15.6, использует метод `persist` для немедленного сохранения команд в виде событий, исключе-

ние составляет деление – для этой команды сначала выполняется проверка аргумента, чтобы предотвратить ошибку деления на ноль.

**Листинг 15.6.** Метод `receive` для обработки сообщений после восстановления

```
val receiveCommand: Receive = {
  case Add(value)      => persist(Added(value))(updateState)
  case Subtract(value) => persist(Subtracted(value))(updateState)
  case Divide(value)  => if(value != 0) persist(Divided(value))(updateState)
  case Multiply(value) => persist(Multiplied(value))(updateState)
  case PrintResult    => println(s"the result is: ${state.result}")
  case GetResult      => sender() ! state.result
  case Clear          => persist(Reset)(updateState)
}
```

`updateState` выполняет затребованную операцию и обновляет результат вычислений (в коде называется `state` и является экземпляром `CalculationResult`). Функция `updateState`, показанная в листинге 15.7, передается в метод `persist`, который принимает два аргумента: событие для сохранения и функцию для обработки сохраняемого события, которая должна вызываться после успешной записи события в журнал.

Функция обработки сохраняемого события вызывается асинхронно, но `akka-persistence` гарантирует, что следующая команда будет обрабатываться только после завершения этой функции, поэтому в ней безопасно вызывать `sender()`, что не распространяется на другие асинхронные вызовы в акторах. Это несколько снижает производительность. Но если вашему приложению не требуется эта гарантия, используйте функцию `persistAsync`, которая не задерживает обработку входящих команд.

**Листинг 15.7.** Обновление внутреннего состояния

```
val updateState: Event => Unit = {
  case Reset          => state = state.reset
  case Added(value)   => state = state.add(value)
  case Subtracted(value) => state = state.subtract(value)
  case Divided(value) => state = state.divide(value)
  case Multiplied(value) => state = state.multiply(value)
}
```

`CalculationResult` поддерживает операции, выполняемые калькулятором, и для каждой возвращает новое неизменяемое значение, как показано в листинге 15.8. Функция `updateState` вызывает один из методов `CalculationResult` и присваивает результат переменной `state`.

**Листинг 15.8.** Выполнение вычислений и возврат следующего состояния

```

case class CalculationResult(result: Double = 0) {
  def reset = copy(result = 0)
  def add(value: Double) = copy(result = this.result + value)
  def subtract(value: Double) = copy(result = this.result - value)
  def divide(value: Double) = copy(result = this.result / value)
  def multiply(value: Double) = copy(result = this.result * value)
}

```

Подведем итоги в отношении записывающей стороны актора-калькулятора: каждая корректная команда превращается в событие и записывается в журнал, после чего в переменную `state` записывается новый результат вычислений.

Теперь обратим свое внимание на функцию `receiveRecover`, представленную в листинге 15.9, которая вызывается в момент (пере)запуска актора со всеми событиями, сохраненными в журнале. Она выполняет ту же логику, что используется после записи в журнал корректных команд, поэтому здесь вызывается та же самая функция `updateState`.

**Листинг 15.9.** Метод `receive` для восстановления

```

val receiveRecover: Receive = {
  case event: Event      => updateState(event)
  case RecoveryCompleted => log.info("Calculator recovery completed")
}

```

←  
Это сообщение посылается один раз  
после завершения восстановления

Каждое событие калькулятора, добавленное в журнал с одним и тем же значением `persistenceId`, передается в ту же функцию `updateState`, чтобы получить тот же эффект, что и прежде.

Функция `receiveRecover` вызывается при запуске и перезапуске. Новые команды, принимаемые, пока актор восстанавливается, будут обработаны после восстановления в порядке поступления.

Вот несколько ключевых аспектов этого короткого фрагмента кода, которые мы будем учитывать в наших следующих примерах:

- команды немедленно преобразуются в события или, как в случае с делением, предварительно проверяются на допустимость;
- команды преобразуются в события, если они допустимы и оказывают влияние на состояние актора; события вызывают обновление состояния актора после восстановления и в процессе восстановления — в обоих случаях действует одна и та же логика;
- логика записи в функции `updateState` помогает избежать дублирования кода в `receiveCommand` и `receiveRecover`;

`CalculationResult` содержит логику калькулятора и неизменяемое состояние (возвращая копию результата каждой операции); это делает функцию `updateState` по-настоящему простой для реализации и чтения.

В следующем разделе мы напишем тест для этого простого примера.

### 15.2.2. Тестирование

Далее мы посмотрим, как протестировать калькулятор. Модульный тест в листинге 15.10 показывает, что тестирование хранимого актора осуществляется как обычно, при условии использования базового класса для тестирования с `akka-persistence`, и включает трейт для очистки журнала. Это необходимо, потому что журнал `LevelDB` находится в одном каталоге со всеми тестами, по умолчанию в каталоге `journal`, внутри текущего рабочего каталога. К сожалению, в `akka-persistence` нет комплекта инструментов для тестирования, который сделал бы эту настройку автоматически или предоставлял бы другие функции тестирования хранилища, но нам пока достаточно будет некоторых вспомогательных функций. Поскольку прежде мы отключили параллельное выполнение тестов, каждый тест может использовать журнал в изоляции от других.

#### Листинг 15.10. Модульный тест для калькулятора

```
package aia.persistence.calculator

import akka.actor._
import akka.testkit._
import org.scalatest._

class CalculatorSpec extends PersistenceSpec(ActorSystem("test"))
  with PersistenceCleanup {

  "The Calculator" should {
    "recover last known result after crash" in {
      val calc = system.actorOf(Calculator.props, Calculator.name)
      calc ! Calculator.Add(1d)
      calc ! Calculator.GetResult
      expectMsg(1d)

      calc ! Calculator.Subtract(0.5d)
      calc ! Calculator.GetResult
      expectMsg(0.5d)

      killActors(calc)

      val calcResurrected = system.actorOf(Calculator.props, Calculator.name)
      calcResurrected ! Calculator.GetResult
```

```

    expectMsg(0.5d)

    calcResurrected ! Calculator.Add(1d)
    calcResurrected ! Calculator.GetResult
    expectMsg(1.5d)
  }
}
}

```

Модульный тест содержит простой пример, помогающий убедиться в корректном восстановлении калькулятора после аварии. Калькулятор возвращает результат вычислений в ответ на сообщение `GetResult`. Класс `PersistenceSpec` определяет метод `killActors`, который наблюдает, останавливает и ждет завершения всех указанных акторов. Этот тест останавливает калькулятор, создает новый и продолжает вычисления с того места, где произошел сбой.

В листинге 15.11 представлены определения класса `PersistenceSpec` и трейта `PersistenceCleanup`.

#### Листинг 15.11. Базовый класс для тестирования калькулятора

```

import java.io.File
import com.typesafe.config._

import scala.util._

import akka.actor._
import akka.persistence._
import org.scalatest._

import org.apache.commons.io.FileUtils

abstract class PersistenceSpec(system: ActorSystem) extends TestKit(system)
  with ImplicitSender
  with WordSpecLike
  with Matchers
  with BeforeAndAfterAll
  with PersistenceCleanup {

  def this(name: String, config: Config) = this(ActorSystem(name, config))
  override protected def beforeAll() = deleteStorageLocations()

  override protected def afterAll() = {
    deleteStorageLocations()
    TestKit.shutdownActorSystem(system)
  }

  def killActors(actors: ActorRef*) = {

```

```

actors.foreach { actor =>
  watch(actor)
  system.stop(actor)
  expectTerminated(actor)
}
}
}

trait PersistenceCleanup {
  def system: ActorSystem

  val storageLocations = List(
    "akka.persistence.journal.leveldb.dir",
    "akka.persistence.journal.leveldb-shared.store.dir",
    "akka.persistence.snapshot-store.local.dir").map { s =>
    new File(system.settings.config.getString(s))
  }

  def deleteStorageLocations(): Unit = {
    storageLocations.foreach(dir => Try(FileUtils.deleteDirectory(dir)))
  }
}

```

Трейт `PersistenceCleanup` определяет метод `deleteStorageLocations`, который удаляет каталоги, созданные журналом LevelDB (а также журнал по умолчанию для моментальных снимков, который мы рассмотрим в разделе 15.2.3). Он получает настроенные каталоги из конфигурации Akka. Класс `PersistenceSpec` удаляет любые оставшиеся каталоги перед запуском модульных тестов, удаляет каталоги после тестирования и останавливает систему акторов, использовавшуюся для тестирования.

Класс `CalculatorSpec` создает тестовую систему акторов с конфигурацией по умолчанию, но он также может использовать конфигурацию пользователя, вызвав вспомогательный конструктор `PersistenceSpec` и передав ему имя системы акторов и объект с настройками. Трейт `PersistenceCleanup` использует `org.apache.commons.io.FileUtils` для удаления каталогов; зависимость для библиотеки `commons-io` можно найти в файле сборки `sbt`.

Класс `PersistenceSpec` будет использоваться для тестирования в следующих разделах. В разделе 15.2.3 мы рассмотрим создание моментальных снимков для ускорения процедуры восстановления.

### 15.2.3. Моментальные снимки

Как упоминалось выше, моментальные снимки можно использовать для ускорения восстановления акторов. Моментальные снимки хранятся в от-

дельном хранилище `SnapshotStore`. По умолчанию моментальные снимки сохраняются в файлы на жестком диске, в каталоге, указанном в параметре `akka.persistence.snapshot-store.local.dir`.

Для демонстрации использования моментальных снимков возьмем за основу актор, реализующий покупательскую корзину. В следующих разделах мы внедрим в службу онлайн-магазина поддержку сохранения покупательской корзины. В листинге 15.12 приводятся определения команд и событий для актора `Basket`.

### Листинг 15.12. Команды события для актора `Basket`

```
sealed trait Command extends Shopper.Command
case class Add(item: Item, shopperId: Long) extends Command
case class RemoveItem(productId: String, shopperId: Long) extends Command
case class UpdateItem(productId: String,
                      number: Int,
                      shopperId: Long) extends Command
case class Clear(shopperId: Long) extends Command
case class Replace(items: Items, shopperId: Long) extends Command
case class GetItems(shopperId: Long) extends Command

case class CountRecoveredEvents(shopperId: Long) extends Command
case class RecoveredEventsCount(count: Long)

sealed trait Event extends Serializable
case class Added(item: Item) extends Event
case class ItemRemoved(productId: String) extends Event
case class ItemUpdated(productId: String, number: Int) extends Event
case class Replaced(items: Items) extends Event
case class Cleared(clearedItems: Items) extends Event

case class Snapshot(items: Items)
```

Команды корзины – это команды покупателя; актор `Shopper`, представляющий покупателя, будет показан ниже

Корзина очищается после оплаты

Событие, показывающее, что корзина очищена

Корзина содержит товары, и, как ожидается, покупатель может добавлять в нее новые товары, удалять или изменять имеющиеся в ней, очищать корзину, оплатив покупку. Каждый покупатель в онлайн-магазине имеет корзину, может добавлять товары в нее и в какой-то момент оплатить покупку товаров в корзине. Состояние актора `Basket` представлено классом `Items`.

### Листинг 15.13. Класс `Items`

```
case class Items(list: List[Item]) {
  // далее следует код для работы с товарами...
```

**Листинг 15.14.** Класс `Item`

```
case class Item(productId: String, number: Int, unitPrice: BigDecimal) {
  // далее следует код для работы с товаром...
```

Детали реализации класса `Items` не имеют принципиального значения для нас; он имеет методы добавления и удаления товаров из списка, а также очистки списка, которые возвращают новую неизменяемую копию, подобно классу `CalculationResult`. Корзина очищается сразу после оплаты покупки. Поэтому имеет смысл сделать мгновенный сразу после очистки; нет необходимости знать о том, что хранилось в корзине прежде, если нам требуется лишь быстро восстановить состояние корзины после перезапуска.

Далее мы рассмотрим только код, имеющий отношение к моментальным снимкам, а в последующих разделах мы посмотрим, как создается актор `Basket` в составе законченной службы. Начнем с методов `updateState` и `receiveCommand`.

**Листинг 15.15.** Метод `updateState` актора `Basket`

```
private val updateState: (Event => Unit) = {
  case Added(item)           => items = items.add(item)
  case ItemRemoved(id)      => items = items.removeItem(id)
  case ItemUpdated(id, number) => items = items.updateItem(id, number)
  case Replaced(newItems)   => items = newItems
  case Cleared(clearedItems) => items = items.clear
}
```

**Листинг 15.16.** Метод `receiveCommand` актора `Basket`

```
def receiveCommand = {
  case Add(item, _) =>
    persist(Added(item))(updateState)

  case RemoveItem(id, _) =>
    if(items.containsProduct(id)) {
      persist(ItemRemoved(id)){ removed =>
        updateState(removed)
        sender() ! Some(removed)
      }
    } else {
      sender() ! None
    }

  case UpdateItem(id, number, _) =>
```

```

if(items.containsProduct(id)) {
  persist(ItemUpdated(id, number)){ updated =>
    updateState(updated)
    sender() ! Some(updated)
  }
} else {
  sender() ! None
}

case Replace(items, _) =>
  persist(Replaced(items))(updateState)

case Clear(_) =>
  persist(Cleared(items)){ e =>
    updateState(e)
    // корзина очищается после оплаты.
    saveSnapshot(Basket.Snapshot(items))
  }
}

case GetItems(_) =>
  sender() ! items

case CountRecoveredEvents(_) =>
  sender() ! RecoveredEventsCount(nrEventsRecovered)

case SaveSnapshotSuccess(metadata) =>
  log.info(s"Snapshot saved with metadata $metadata")
}

case SaveSnapshotFailure(metadata, reason) =>
  log.error(s"Failed to save snapshot: $metadata, $reason.")
}

```

← Сохраняет моментальный снимок, когда корзина очищается

← Моментальный снимок успешно сохранен

← Моментальный снимок не может быть сохранен

Состояние `Items` корзины сохраняется как моментальный снимок вызовом метода `saveSnapshot`. Он возвращает `SaveSnapshotSuccess` или `SaveSnapshotFailure`, показывая результат попытки сохранения моментального снимка. В этом примере сохранение моментального снимка делается исключительно ради оптимизации, поэтому мы не предпринимаем никаких действий, когда попытка сохранения не увенчалась успехом. В листинге 15.17 показан метод `receiveRecover` актора `Basket`.

#### Листинг 15.17. Метод `receiveRecover` актора `Basket`

```

def receiveRecover = {
  case event: Event =>
    nrEventsRecovered = nrEventsRecovered + 1
    updateState(event)
  case SnapshotOffer(_, snapshot: Basket.Snapshot) =>

```

← Моментальный снимок, предлагаемый на этапе восстановления

```

log.info(s"Recovering baskets from snapshot: $snapshot for $persistenceId")
items = snapshot.items
}

```

Метод `receiveRecover` ожидаемо вызывает `updateState`, но он также обрабатывает сообщение `SnapshotOffer`. По умолчанию актору сначала передается моментальный снимок, а потом все события, следующие за ним. Любые события, предшествующие моментальному снимку, не передаются в `receiveRecover`. Это означает, что все события, предшествовавшие оплате, не будут обрабатываться в ходе восстановления.

**ИЗМЕНЕНИЕ ПОРЯДКА ВОССТАНОВЛЕНИЯ.** По умолчанию в восстановлении участвует только последний моментальный снимок, чего вполне достаточно в большинстве случаев. Чтобы изменить выбор моментального снимка для восстановления хранимого актора, можно переопределить метод `recovery`. Значение `Recovery`, возвращаемое этим методом, определяет начальный моментальный снимок для восстановления в виде номера `sequenceNr` и/или `timestamp` и дополнительно `sequenceNr`, до которого нужно восстановить, или максимальное количество сообщений, которые нужно обработать.

Команда `CountRecoveredEvents` добавляется в тест, если в процессе восстановления происходит пропуск событий. В листинге 15.17 видно, что `nrEventsRecovered` увеличивается с каждым событием; в ответ на команду `CountRecoveredEvents` актер `Basket` возвращает количество событий, обработанных в процессе восстановления, как показано в листинге 15.16. В листинге 15.18 приводится модульный тест `BasketSnapshotSpec`, проверяющий пропуск событий, предшествующих моментальному снимку (моменту очистки корзины).

#### Листинг 15.18. `BasketSnapshotSpec`

```

package aia.persistence

import scala.concurrent.duration._

import akka.actor._
import akka.testkit._
import org.scalatest._

class BasketSpec extends PersistenceSpec(ActorSystem("test"))
  with PersistenceCleanup {

  val shopperId = 2L
  val macbookPro = Item("Apple Macbook Pro", 1, BigDecimal(2499.99))

```

```

val macPro = Item("Apple Mac Pro", 1, BigDecimal(10499.99))
val displays = Item("4K Display", 3, BigDecimal(2499.99))
val appleMouse = Item("Apple Mouse", 1, BigDecimal(99.99))
val appleKeyboard = Item("Apple Keyboard", 1, BigDecimal(79.99))
val dWave = Item("D-Wave One", 1, BigDecimal(14999999.99))

```

```

"The basket" should {
  "skip basket events that occurred before Cleared during recovery" in {
    val basket = system.actorOf(Basket.props, Basket.name(shopperId))
    basket ! Basket.Add(macbookPro, shopperId)
    basket ! Basket.Add(displays, shopperId)
    basket ! Basket.GetItems(shopperId)
    expectMsg(Items(macbookPro, displays))

    basket ! Basket.Clear(shopperId)

    basket ! Basket.Add(macPro, shopperId)
    basket ! Basket.RemoveItem(macPro.productId, shopperId)
    expectMsg(Some(Basket.ItemRemoved(macPro.productId)))

    basket ! Basket.Clear(shopperId)
    basket ! Basket.Add(dWave, shopperId)
    basket ! Basket.Add(displays, shopperId)

    basket ! Basket.GetItems(shopperId)
    expectMsg(Items(dWave, displays))

    killActors(basket)

    val basketResurrected = system.actorOf(Basket.props,
      Basket.name(shopperId))
    basketResurrected ! Basket.GetItems(shopperId)
    expectMsg(Items(dWave, displays))

    basketResurrected ! Basket.CountRecoveredEvents(shopperId)
    expectMsg(Basket.RecoveredEventsCount(2))

    killActors(basketResurrected)
  }
}

```

← Очистка корзины вызывает создание моментального снимка

← Убедиться, что в процессе восстановления обрабатываются только события, следующие за моментальным снимком

Подсчет количества предлагаемых моментальных снимков мы оставляем читателям в качестве самостоятельного упражнения. Это был простой пример: моментальный снимок всегда представлял пустую корзину, поэтому мы использовали маркер в журнале, чтобы предотвратить обработку всех предшествующих операций с корзиной.

Как видите, хранимые акторы могут восстанавливать себя из моментальных снимков и событий в журнале. В некоторых случаях требуется организовать чтение событий из журнала вне процесса восстановления хранимого актора. Как это делается, мы покажем в следующем разделе.

#### 15.2.4. Запрос хранимых событий

В Акка имеется экспериментальный модуль `persistence-query`, позволяющий запрашивать события из журнала вне рамок процесса восстановления хранимого актора. В этом разделе мы дадим лишь краткий обзор этого модуля, потому что он считается экспериментальным и не используется в процессе восстановления состояния актора. Важно отметить, что он не является средством обработки запросов, подобно SQL. Лучший вариант использования `persistent-query` – последовательное чтение событий для определенного актора из журнала и добавление их в какую-то другую базу данных, более пригодную для выполнения произвольных запросов.

Если вам достаточно простой поддержки запросов, тогда модуль `persistence-query` может быть вполне достаточно для ваших нужд. Модуль `persistence-query` поддерживает получение всех событий, событий с определенным `persistenceId` и событий по заданной метке (для этого необходимо, чтобы адаптер событий явно отмечал события в журнале; здесь не будет рассматриваться).

Этих возможностей может показаться маловато, но их вполне достаточно, чтобы прочитать все или подмножество событий и записать их в базу данных для дальнейшей обработки. Модуль `persistence-query` предоставляет API для чтения событий из конечной точки `Source`.

В этом разделе мы рассмотрим, как получить доступ к конечной точке `Source` с событиями, а добавление событий в базу данных для дальнейшей обработки мы оставляем вам в качестве самостоятельного упражнения.

Сначала добавим зависимости.

#### Листинг 15.19. Добавление зависимости для `persistence-query`

```
libraryDependencies += {
  val akkaVersion      = "2.4.9"
  Seq(
    // другие зависимости опущены ...
    "com.typesafe.akka" %% "akka-persistence-query-experimental" % akkaVersion,
    // другие зависимости опущены ...
  )
}
```

В число зависимостей также входит механизм чтения журнала LevelDB, который мы используем в этом разделе. Предполагается, что большинство плагинов журнала, создаваемых сообществом, будут поддер-

живать этот механизм. В листинге 15.20 показано, как получить доступ к `LeveldbReadJournal`.

#### Листинг 15.20. Доступ к `LeveldbReadJournal`

```
implicit val mat = ActorMaterializer()(system)
val queries =
  PersistenceQuery(system).readJournalFor[LeveldbReadJournal](
    LeveldbReadJournal.Identifier
  )
```

Вы должны создать экземпляр `ActorMaterializer` в неявной области видимости, что является общим требованием при использовании `akka-stream`. Метод `readJournalFor` расширения `PersistenceQuery` возвращает конкретный экземпляр `ReadJournal`, в данном случае `LeveldbReadJournal`.

`LeveldbReadJournal` поддерживает все типы запросов, которые можно найти в трейтах `AllPersistenceIdsQuery`, `CurrentPersistenceIdsQuery`, `EventsByPersistenceIdQuery`, `CurrentEventsByPersistenceIdQuery`, `EventsByTagQuery` и `CurrentEventsByTagQuery`. Другие плагины журналов могут реализовать все или только некоторые из этих трейтов.

По сути, существует два типа запросов: методы, начинающиеся с приставки `current`, возвращают конечную точку `Source` и закрывают поток, как только все имеющиеся события будут переданы через нее, а методы, не начинающиеся с `current`, не закрывают поток и продолжают передавать события «вживую», по мере их поступления. (Конечно, поток может закрыться при возникновении ошибки, когда журнал оказывается недоступным для чтения; вам придется самим обрабатывать отказ потока.)

В листинге 15.21 показано, как прочитать текущие события конкретной корзины, хранящиеся в журнале `LevelDB`.

#### Листинг 15.21. Получение текущих событий конкретной корзины

```
val src: Source[EventEnvelope, NotUsed] =
  queries.currentEventsByPersistenceId(
    Basket.name(shopperId), 0L, Long.MaxValue)
val events: Source[Basket.Event, NotUsed] =
  src.map(_._event.asInstanceOf[Basket.Event])
val res: Future[Seq[Basket.Event]] = events.runWith(Sink.seq)
```

Получить источник текущих событий для конкретной корзины от первого до последнего хранящегося события

Поскольку мы записываем в журнал только события типа `Basket.Event`, здесь можно смело выполнить приведение типа

Для тестирования можно запустить источник с `Sink.seq`, чтобы получить все события. Возможно, вам придется дождаться завершения `Future` для сравнения его со списком известных событий

Значение `persistenceId` в акторе `Basket` имеет то же значение, возвращаемое обращением к `Basket.name`, именно поэтому данный пример ра-

ботает. Метод `currentEventsByPersistenceId` принимает два аргумента, `fromSequenceNr` и `toSequenceNr`, используя 0 и `Long.MaxValue` соответственно; возвращает все события, имеющиеся в журнале с `persistenceId`. В листинге 15.22 показано, как прочитать «живой» поток событий из корзины, хранящихся в журнале LevelDB.

**Листинг 15.22.** Чтение «живого» потока событий из корзины

```
val src: Source[EventEnvelope, NotUsed] =
  queries.eventsByPersistenceId(
    Basket.name(shopperId), 0L, Long.MaxValue)
val dbRows: Source[DbRow, NotUsed] =
  src.map(eventEnvelope => toDbRow(eventEnvelope))
  events.runWith(reactiveDatabaseSink)
```

Получение источника всех событий для конкретной корзины, который не закрывается

Вы могли бы транслировать события в «записи базы данных», которые можно было бы записать в приемник некоторой базы данных; детали здесь не рассматриваются

Конечная точка `Source`, возвращаемая методом `eventsByPersistenceId`, никогда не закрывается; она будет продолжать поставлять события по мере их появления. Трансляцию событий в некоторое представление, пригодное для сохранения в базе данных, показанную в листинге 15.22, следует читать как «незаконченный псевдокод»; конкретную реализацию мы оставляем за читателями. Вам может понадобиться хранить порядковый номер на случай появления ошибок или если эта логика перезапустится по какой-то другой причине. Хорошим вариантом для этого может быть сохранение порядковых номеров в целевой базе данных. (`EventEnvelope` имеет поле `sequenceNr`.) В случае перезапуска вы сможете получить максимальный порядковый номер из целевой базы данных и продолжить с него.

В следующем разделе мы посмотрим, как реализовать сериализацию событий и моментальных снимков.

### 15.2.5. Сериализация

Сериализация настраивается посредством инфраструктуры сериализации в Akka. По умолчанию используется механизм сериализации Java. Он отлично подходит для целей тестирования, но его нельзя использовать в промышленном окружении; в большинстве случаев необходим более эффективный механизм сериализации.

Чтобы использовать иной механизм, кроме механизма по умолчанию, придется приложить некоторые усилия, о чем мы и расскажем в этом разделе.

Создание собственного механизма сериализации – лучший выбор, если требуется иметь полный контроль над сериализацией моментальных снимков и событий. Именно этим мы займемся сейчас.

### Серьезно? Мы напишем свой механизм сериализации?

Создание собственного механизма сериализации требует приложения немалых усилий. Если вы обнаружите, что механизм по умолчанию не обладает достаточным быстродействием, или если вам понадобится выполнить дополнительную логику для автоматической миграции сериализованных данных в старом формате, тогда создание собственного механизма сериализации – действительно лучший выбор.

Разве нет других вариантов, спросите вы? Модуль `akka-remote` содержит механизм сериализации, поддерживающий формат `Google Protocol Buffers` (<https://github.com/google/protobuf>), но он работает только с классами, сгенерированными с помощью механизма `protobuf` – он хорошо подходит только для случаев, когда код с самого начала пишется с использованием определений `protobuf`. Из этих определений генерируются классы, которые затем используются непосредственно в роли событий. Другой вариант – использовать стороннюю библиотеку сериализации для Акка. Библиотека с именем `akka-kryo-serialization` (<https://github.com/romix/akka-kryo-serialization>), как утверждается разработчиками, автоматически поддерживает сериализацию большинства классов `Scala` и использует формат `kryo`. Однако эта библиотека также требует настройки и не поддерживает миграцию версий.

`Stamina` (<https://github.com/scalapenos/stamina>) – еще один механизм сериализации для Акка, специально созданный для совместной работы с `akka-persistence`. Он имеет дополнительный модуль поддержки формата `JSON` через `spray-json`. `Stamina` предоставляет специализированный предметно-ориентированный язык для версионирования и автоматической миграции сериализованных данных (также называется *приведением вверх* (`upcasting`)), что позволяет обновлять службу без необходимости останавливать ее и преобразовывать весь журнал перед запуском.

Прежде чем заняться фактической реализацией своего механизма сериализации, посмотрим, как настраиваются такие механизмы. Код в листинге 15.23 демонстрирует, как настроить свой механизм сериализации для поддержки всех классов `Basket.Event` и класса моментального снимка `Basket.Snapshot`.

#### Листинг 15.23. Настройка механизма сериализации

```
akka {
  actor {
    serializers {
```

```

basket = "aia.persistence.BasketEventSerializer"
basketSnapshot = "aia.persistence.BasketSnapshotSerializer"
}
serialization-bindings {
  "aia.persistence.Basket$Event" = basket
  "aia.persistence.Basket$Snapshot" = basketSnapshot
}
}
}

```

Регистрация собственных механизмов сериализации

Привязка классов для сериализации с использованием конкретных механизмов сериализации

Любой класс, не привязанный к конкретному механизму, автоматически будет сериализоваться с использованием механизма по умолчанию. Имена классов должны квалифицироваться полностью. В данном случае `Event` и `Snapshot` являются частью объекта-компаньона `Basket`. Компилятор Scala создаст для объекта Java-класс с именем, оканчивающимся знаком доллара, что объясняет странные, полностью квалифицированные имена классов для `Basket.Event` и `Basket.Snapshot`.

Любой механизм сериализации должен создавать представления в виде массивов байтов для заданных событий или моментальных снимков и уметь позднее воссоздавать их из массивов байтов. В листинге 15.24 представлен трейт, который мы должны реализовать в своем механизме сериализации.

#### Листинг 15.24. Трейт `Serializer` в Akka

```

trait Serializer {
  /**
   * Уникальное значение для идентификации этой
   * реализации Serializer,
   * используется для оптимизации сетевого трафика
   * Значения от 0 до 16 зарезервированы для внутренних нужд Akka
   */
  def identifier: Int

  /**
   * Сериализует заданный объект в массив байтов
   */
  def toBinary(o: AnyRef): Array[Byte]

  /**
   * Возвращает признак необходимости передачи манифеста
   * в метод fromBinary
   */
  def includeManifest: Boolean

  /**

```

```

* Воссоздает объект из массива байтов
* с необязательной подсказкой о типе;
* класс должен загружаться с использованием ActorSystem.dynamicAccess.
*/
def fromBinary(bytes: Array[Byte], manifest: Option[Class[_]]): AnyRef
}

```

Вообще, любой механизм сериализации должен записывать в массив байтов некоторый дискриминатор (отличительный признак), чтобы потом этот массив можно было преобразовать обратно в объект. Это может быть сериализованное имя класса или просто числовой идентификатор типа. При использовании механизма сериализации из Java Akka автоматически может записывать в массив байтов манифест класса, если `includeManifest` имеет значение `true`.

В своих механизмах сериализации событий и моментальных снимков `Basket` мы будем использовать библиотеку `spray-json` для сериализации/десериализации в формат JSON (в конце концов, должны же мы выбрать какой-то формат). Полный список форматов JSON определяется в объекте `JsonFormats`, который мы опустили в наших листингах.

Рассмотрим для начала `BasketEventSerializer`, осуществляющий сериализацию `Basket.Event`.

#### Листинг 15.25. Нестандартный механизм сериализации для событий `Basket`

```

import scala.util.Try
import akka.serialization._
import spray.json._

class BasketEventSerializer extends Serializer {
  import JsonFormats._

  val includeManifest: Boolean = false
  val identifier = 123678213

  def toBinary(obj: AnyRef): Array[Byte] = {
    obj match {
      case e: Basket.Event =>
        BasketEventFormat.write(e).compactPrint.getBytes
      case msg =>
        throw new Exception(s"Cannot serialize $msg with ${this.getClass}")
    }
  }

  def fromBinary(bytes: Array[Byte],
                 clazz: Option[Class[_]]): AnyRef = {
    val jsonAst = new String(bytes).parseJson

```

```

BasketEventFormat.read(jsonAst)
}
}

```

← Преобразует текст JSON в Basket.Event с помощью BasketEventFormat

Формат `BasketEventFormat` в `JsonFormats` записывает каждое событие в виде массива JSON. Первый элемент – это дискриминатор, определяющий тип события во втором элементе массива. Тот же дискриминатор используется для определения формата, который должен применяться при десериализации. Определение `BasketEventFormat` показано в листинге 15.26.

### Листинг 15.26. `BasketEventFormat`

```

implicit object BasketEventFormat
  extends RootJsonFormat[Basket.Event] {
  import Basket._
  val addedId = JsNumber(1)
  val removedId = JsNumber(2)
  val updatedId = JsNumber(3)
  val replacedId = JsNumber(4)
  val clearedId = JsNumber(5)

  def write(event: Event) = {
    event match {
      case e: Added =>
        JsArray(addedId, addedEventFormat.write(e))
      case e: ItemRemoved =>
        JsArray(removedId, removedEventFormat.write(e))
      case e: ItemUpdated =>
        JsArray(updatedId, updatedEventFormat.write(e))
      case e: Replaced =>
        JsArray(replacedId, replacedEventFormat.write(e))
      case e: Cleared =>
        JsArray(clearedId, clearedEventFormat.write(e))
    }
  }

  def read(json: JsValue): Basket.Event = {
    json match {
      case JsArray(Vector(`addedId`, jsEvent)) =>
        addedEventFormat.read(jsEvent)
      case JsArray(Vector(`removedId`, jsEvent)) =>
        removedEventFormat.read(jsEvent)
      case JsArray(Vector(`updatedId`, jsEvent)) =>
        updatedEventFormat.read(jsEvent)
      case JsArray(Vector(`replacedId`, jsEvent)) =>
        replacedEventFormat.read(jsEvent)
      case JsArray(Vector(`clearedId`, jsEvent)) =>

```

```

        clearedEventFormat.read(jsEvent)
    case j =>
        deserializationError("Expected basket event, but got " + j)
    }
}
}

```

Определение класса `BasketSnapshotSerializer` показано в листинге 15.27. Он использует неявное определение формата в `JsonFormats` для преобразования между JSON и `BasketSnapshot`.

Подобный нестандартный механизм сериализации можно также использовать для автоматической миграции старых версий сериализованных данных. Одно из возможных решений: записывать в массив байтов значение дискриминатора версии, подобно дискриминатору типа, и затем использовать его для выбора логики чтения старых версий сериализованных данных и преобразования их в текущую версию. (Библиотека `Stamina` делает нечто подобное и предоставляет хороший предметно-ориентированный язык для определения миграций версий.)

#### Листинг 15.27. Механизм сериализации моментальных снимков `Basket`

```

class BasketSnapshotSerializer extends Serializer {
  import JsonFormats._

  val includeManifest: Boolean = false
  val identifier = 1242134234

  def toBinary(obj: AnyRef): Array[Byte] = {
    obj match {
      case snap: Basket.Snapshot => snap.toJson.compactPrint.getBytes
      case msg => throw new Exception(s"Cannot serialize $msg")
    }
  }

  def fromBinary(bytes: Array[Byte],
                 clazz: Option[Class[_]]): AnyRef = {
    val jsonStr = new String(bytes)
    jsonStr.parseJson.convertTo[Basket.Snapshot]
  }
}

```

Собственные механизмы сериализации, показанные здесь, являются лишь примерами. Создание универсального механизма сериализации, пригодного для всех случаев, – слишком сложная задача. Решение, описанное здесь, не идеально, но оно поможет вам понять суть проблем, с которыми можно столкнуться. Интеграцию плагинов сериализации, создан-

ных в сообществе, мы оставляем читателям в качестве самостоятельного упражнения. В следующем разделе мы рассмотрим применение хранимых акторов в кластере.

### Адаптеры событий

Модуль `akka-persistence` не просто сериализует события и моментальные снимки в массивы байтов в `Journal` или `SnapshotStore`. Сериализованные объекты также заворачиваются во внутренний формат `protobuf`, что необходимо для внутреннего учета. Это означает, что у вас не получится просто послать запрос плагину базы данных или журнала и получить структуру в формате JSON, в которую ваш механизм сериализации преобразует события.

Решить эту проблему можно с помощью адаптера `EventAdapter`. Он находится между журналом и событием, которое читается или записывается, и обеспечивает все необходимые преобразования. С его помощью легко отделить события от модели хранимых данных.

`EventAdapter` должен *соответствовать* плагину `Journal`. `EventAdapter` может преобразовывать события в объекты JSON, но для этого `Journal` должен обрабатывать объекты JSON иначе, чем любые объекты событий, которые обычно подвергаются сериализации в байты и заворачиваются во внутреннюю структуру.

## 15.3. Кластер на основе хранимых акторов

До сих пор мы обсуждали проблему восстановления состояния актора в локальной системе акторов. В этом разделе мы продолжим реализацию службы онлайн-магазина. Для начала подробнее рассмотрим решение на основе локальной системы акторов.

Затем мы изменим приложение так, чтобы его можно было запускать в кластере с расширением `cluster singleton`. Это расширение позволяет запустить в кластере Akka только один экземпляр актора (и его потомков) на узле.

Расширение `cluster singleton` автоматически будет запускать все акторы-корзины на другом узле в случае выхода из строя узла с актором-одиночкой, поэтому состояние корзин должно храниться где-то еще, предпочтительно в распределенной базе данных, такой как Apache Cassandra. Это повысит надежность службы онлайн-магазина, но не избавит от вероятной необходимости хранить в памяти больше корзин, чем может уместиться на одном узле в кластере.

Для решения этой проблемы мы рассмотрим расширение *cluster sharding*, которое позволит распределить акторы-корзины по узлам кластера согласно стратегии распределения.

Но, прежде чем углубиться в детали, рассмотрим общую структуру службы онлайн-магазина, изображенную на рис. 15.5.

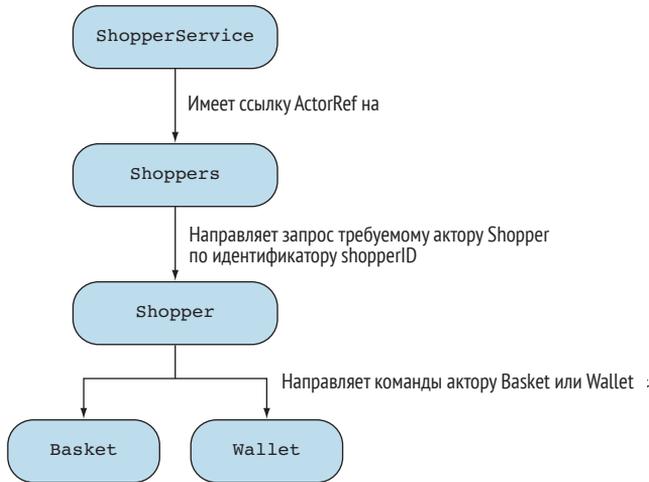


Рис. 15.5. Структура службы онлайн-магазина

Проект в репозитории GitHub включает HTTP-службу доступа к корзинам покупателей (*ShopperService*). Служба *ShopperService* принимает ссылку *ActorRef* на актор *Shoppers*. Он создает или отыскивает актор *Shopper* для каждого уникального идентификатора *shopperId*. Если актор *Shopper* с таким идентификатором отсутствует, он создается; если искомый актор существует, возвращается ссылка на него. Актор *Shoppers* пересылает любые запросы конкретному актору *Shopper*, исходя из команды. Команды всегда содержат *shopperId*. Команды для *Basket* и *Wallet* также передаются через актор *Shopper*.

Когда пользователь посещает онлайн-магазин в первый раз, для него автоматически генерируется блок данных *cookie*. Этот же блок данных используется, когда пользователь повторно возвращается в онлайн-магазин. В *cookie* хранится уникальный идентификатор покупателя, который в данном примере имеет имя *shopperId*. (Мы исключили из примера HTTP-службы код, отвечающий за создание *cookie*.) В данном примере *shopperId* – это простое значение типа *Long*; но на практике для этих целей обычно используются *универсально-уникальные идентификаторы* (*Universally Unique Identifier, UUID*). В листинге 15.28 показан фрагмент трейта *ShoppersRoutes*, который определяет маршруты для HTTP-службы.

**Листинг 15.28.** ShoppersRoutes

```

trait ShoppersRoutes extends
  ShopperMarshalling {
  def routes =
    deleteItem ~
    updateItem ~
    getBasket ~
    updateBasket ~
    deleteBasket ~
    pay

  def shoppers: ActorRef

  implicit def timeout: Timeout
  implicit def executionContext: ExecutionContext

  def pay = {
    post {
      pathPrefix("shopper" / ShopperIdSegment / "pay") { shopperId =>
        shoppers ! Shopper.PayBasket(shopperId)
        complete(OK)
      }
    }
  }
}

```

В листинге 15.29 представлено определение актора LocalShoppers.

**Листинг 15.29.** Актор LocalShoppers

```

package aia.persistance

import akka.actor._

object LocalShoppers {
  def props = Props(new LocalShoppers)
  def name = "local-shoppers"
}

class LocalShoppers extends Actor
  with ShopperLookup {
  def receive = forwardToShopper
}

trait ShopperLookup {
  implicit def context: ActorContext

  def forwardToShopper: Actor.Receive = {

```

```

case cmd: Shopper.Command =>
  context.child(Shopper.name(cmd.shopperId))
    .fold(createAndForward(cmd, cmd.shopperId))(forwardCommand(cmd))
}

def forwardCommand(cmd: Shopper.Command)(shopper: ActorRef) =
  shopper forward cmd

def createAndForward(cmd: Shopper.Command, shopperId: Long) = {
  createShopper(shopperId) forward cmd
}

def createShopper(shopperId: Long) =
  context.actorOf(Shopper.props(shopperId),
    Shopper.name(shopperId))
}

```

Процедура поиска актора `Shopper` выделена в `ShopperLookup`, чтобы ее можно было использовать в обоих расширениях, `cluster singleton` и `cluster sharding`.

В листинге 15.30 приводится определение актора `Shopper`.

### Листинг 15.30. Актор `Shopper`

```

import akka.actor._

object Shopper {
  def props(shopperId: Long) = Props(new Shopper)
  def name(shopperId: Long) = shopperId.toString

  trait Command {
    def shopperId: Long
  }

  case class PayBasket(shopperId: Long) extends Command
  // для простоты предполагается, что каждый покупатель
  // может потратить до 40 000.
  val cash = 40000
}

class Shopper extends Actor {
  import Shopper._

  def shopperId = self.path.name.toLong

  val basket = context.actorOf(Basket.props,

```

← Все команды для `Shopper` имеют `shopperId`

```

Basket.name(shopperId))

val wallet = context.actorOf(Wallet.props(shopperId, cash),
  Wallet.name(shopperId))

def receive = {
  case cmd: Basket.Command => basket forward cmd
  case cmd: Wallet.Command => wallet forward cmd

  case PayBasket(shopperId) => basket ! Basket.GetItems(shopperId)
  case Items(list) => wallet ! Wallet.Pay(list, shopperId)
  case Wallet.Paid(_, shopperId) => basket ! Basket.Clear(shopperId)
}
}

```

Актор `Shopper` создает акторы `Basket` и `Wallet` и пересылает команды им. Оплата товаров в корзине реализуется в акторе `Shopper`. Он сначала посылает `GetItems` актору `Basket`; получив в ответ сообщение `Items`, он посылает сообщение `Pay` актору `Wallet`, который отвечает сообщением `Paid`. Затем `Shopper` посылает сообщение `Clear` корзине, завершая процесс покупки. В следующем разделе мы рассмотрим изменения, которые необходимо внести, чтобы запустить актор `Shoppers` как расширение `cluster singleton`.

### 15.3.1. Расширение `cluster singleton`

Следующая тема, которую мы рассмотрим, – расширение `cluster singleton`. Мы запустим актор `Shoppers` как единственный экземпляр, в том смысле, что в кластере всегда будет иметься только один экземпляр актора `Shoppers`.

Расширение `cluster singleton` является частью модуля `cluster-tools`, а расширение `cluster sharding` – частью модуля `cluster-sharding`; поэтому мы должны добавить в проект соответствующие зависимости.

**Листинг 15.31.** Зависимости для расширений `cluster singleton` и `cluster sharding`

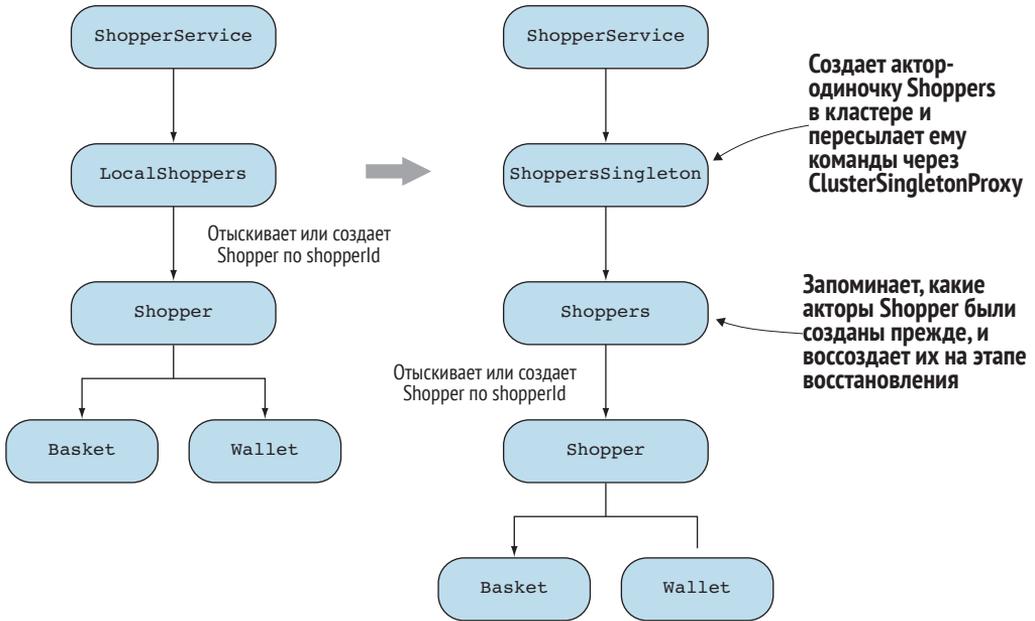
```

libraryDependencies += {
  val akkaVersion = "2.4.9"
  Seq(
    // другие зависимости опущены ...
    "com.typesafe.akka" %% "akka-cluster-tools" % akkaVersion,
    "com.typesafe.akka" %% "akka-cluster-sharding" % akkaVersion,
    // другие зависимости опущены ..
  )
}

```

Расширение `cluster singleton` требует правильной конфигурации кластера, которую можно найти в файле `application.conf` в каталоге `src/main/`

*resources*, как обсуждалось в главе 13. На рис. 15.6 показаны необходимые изменения.



**Рис. 15.6.** Изменения для перехода от локальной системы акторов к расширению cluster singleton

В *ShoppingService* вместо ссылки на *LocalShoppers* будет передаваться ссылка на *ShoppersSingleton*. Определение актора *ShoppersSingleton* показано в листинге 15.32.

#### Листинг 15.32. *ShoppersSingleton*

```
import akka.actor._
import akka.cluster.singleton.ClusterSingletonManager
import akka.cluster.singleton.ClusterSingletonManagerSettings
import akka.cluster.singleton.ClusterSingletonProxy
import akka.cluster.singleton.ClusterSingletonProxySettings
import akka.persistance._

object ShoppersSingleton {
  def props = Props(new ShoppersSingleton)
  def name = "shoppers-singleton"
}

class ShoppersSingleton extends Actor {

  val singletonManager = context.system.actorOf(
```

```

ClusterSingletonManager.props(
  Shoppers.props,
  PoisonPill,
  ClusterSingletonManagerSettings(context.system)
    .withRole(None)
    .withSingletonName(Shoppers.name)
)
)

val shoppers = context.system.actorOf(
  ClusterSingletonProxy.props(
    singletonManager.path.child(Shoppers.name)
      .toStringWithoutAddress,
    ClusterSingletonProxySettings(context.system)
      .withRole(None)
      .withSingletonName("shoppers-proxy")
  )
)

def receive = {
  case command: Shopper.Command => shoppers forward command
}
}

```

Актор `ShoppersSingleton` действует как ссылка на фактический актор-одиночку в кластере. Актор `ShoppersSingleton` будет запускаться на каждом узле, но только на одном из них действительно будет запущен актор `Shoppers`.

`ShoppersSingleton` создает ссылку на `ClusterSingletonManager`. Диспетчер акторов-одиночек гарантирует, что в каждый конкретный момент времени в кластере будет присутствовать только один актор `Shoppers`. От нас требуется только передать ему `Props` и имя актора-одиночки (через `singletonProps` и `singletonName`). Кроме того, `ShoppersSingleton` создает прокси для доступа к актору `Shoppers`, который должен пересылать ему сообщения. Прокси всегда ссылается на текущий действующий в кластере актор-одиночку.

Актор `Shoppers` является фактическим актором-одиночкой. Это хранимый актор `PersistentActor`, который запоминает в виде событий все созданные акторы `Shopper`. Актор `Shoppers` может повторно читать эти события и воссоздавать акторы `Shopper` после аварии, что делает возможным перемещение актора-одиночки на другой узел.

Важно отметить, что главной задачей расширения `cluster singleton` является предотвращение появления в кластере более одного активного актора-одиночки. Когда механизм `cluster singleton` выходит из строя, возникает некоторый промежуток времени, когда прежний актор-одиночка уже не функционирует, а новый еще не запущен, то есть в этот период времени возможна потеря сообщений.

Акторы `Basket` и `Wallet` также являются хранимыми, то есть они автоматически восстанавливают свое состояние из событий, когда повторно создаются актором `Shopper`.

Определение актора `Shoppers` показано в листинге 15.33.

#### Листинг 15.33. Актор `Shoppers`

```
object Shoppers {
  def props = Props(new Shoppers)
  def name = "shoppers"

  sealed trait Event
  case class ShopperCreated(shopperId: Long)
}

class Shoppers extends PersistentActor
  with ShopperLookup {
  import Shoppers._

  def persistenceId = "shoppers"

  def receiveCommand = forwardToShopper

  override def createAndForward(cmd: Shopper.Command, shopperId: Long) = {
    val shopper = createShopper(shopperId)
    persistAsync(ShopperCreated(shopperId)) { _ =>
      forwardCommand(cmd)(shopper)
    }
  }

  def receiveRecover = {
    case ShopperCreated(shopperId) =>
      context.child(Shopper.name(shopperId))
        .getOrElse(createShopper(shopperId))
  }
}
```

Метод `createAndForward` переопределен с целью дополнительно сохранить событие `ShopperCreated`. Здесь без опаски можно использовать `persistAsync`, потому что порядок создания акторов `Shopper` не имеет никакого значения. Опробовать расширение `cluster singleton` можно, выполнив команду `sbt run` и выбрав для запуска класс `aia.persistence.SingletonMain`. На практике перед всеми узлами желательно было бы добавить балансировщик нагрузки. Служба REST на любом узле сможет взаимодействовать с расширением `cluster singleton` через прокси в акторе `ShoppersSingleton`. Но для тестирования вы должны выбрать другую реализацию журнала.

Если вы собираетесь выполнять тестирование локально, используйте разделяемый журнал LevelDB, но имейте в виду, что он создавался исключительно для тестирования, потому что хранит данные локально.

Для практического использования используйте плагин журнала akka-persistence-cassandra (<https://github.com/krasserm/akka-persistence-cassandra/>) для Apache Cassandra.

Apache Cassandra – масштабируемая база данных с высокой доступностью, поддерживающая копирование данных между узлами кластера. Использование Apache Cassandra в качестве хранилища для кластеров позволит противостоять авариям на узлах в кластере базы данных и в кластере Akka.

Как видите, нам потребовалось изменить не так много, чтобы заменить локальную службу кластерной, способной противостоять отказам в кластере. В следующем разделе мы рассмотрим изменения, которые нужно внести, чтобы обеспечить распределение акторов по кластеру.

### 15.3.2. Расширение cluster sharding

Теперь займемся расширением cluster sharding. Распределять акторы Shopper по узлам мы будем, опираясь на их идентификаторы shopperId. Расширение cluster sharding распределяет акторы по группам узлов. Каждый shopperId может находиться только в одной группе. Модуль ClusterSharding берет на себя всю заботу о равномерном распределении акторов в кластере. На рис. 15.7 показаны изменения, которые нам предстоит внести.

Расширение ClusterSharding имеет метод shardRegion, возвращающий ссылку на актор ShardRegion. Актор ShardRegion обеспечивает пересылку команд акторам в его группе узлов. В данном случае это несколько измененная версия актора Shopper с именем ShardedShopper. В листинге 15.34 представлена распределенная версия актора Shoppers – актор ShardedShoppers.

#### Листинг 15.34. ShardedShoppers

```
package aia.persistence.sharded

import aia.persistence._
import akka.actor._
import akka.cluster.sharding.{ClusterSharding, ClusterShardingSettings}

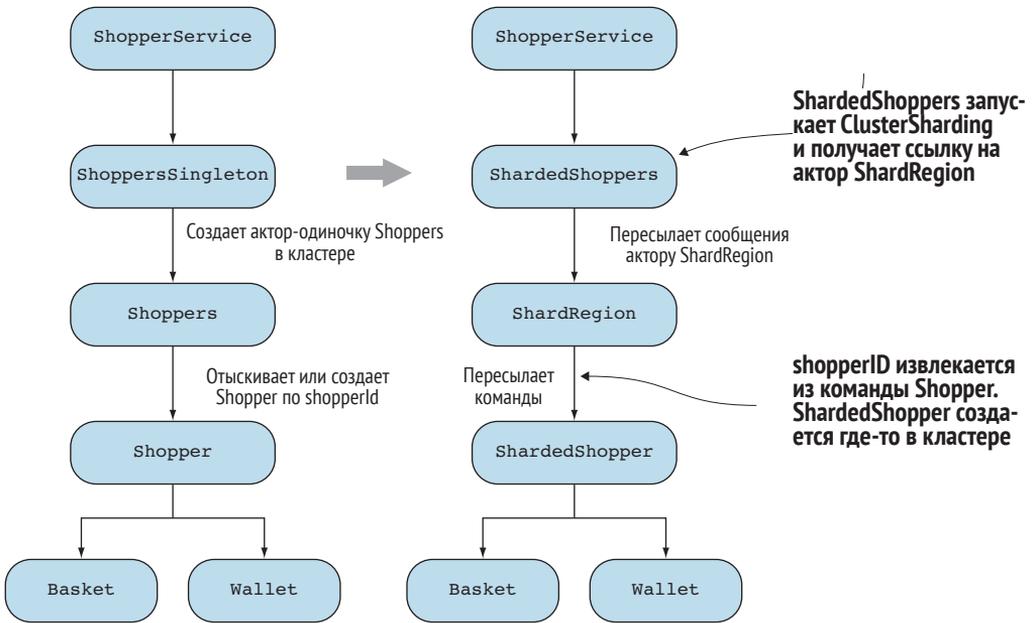
object ShardedShoppers {
  def props= Props(new ShardedShoppers)
  def name = "sharded-shoppers"
}
```

```
class ShardedShoppers extends Actor {

  ClusterSharding(context.system).start(
    ShardedShopper.shardName,
    ShardedShopper.props,
    ClusterShardingSettings(context.system),
    ShardedShopper.extractEntityId,
    ShardedShopper.extractShardId
  )

  def shardedShopper = {
    ClusterSharding(context.system).shardRegion(ShardedShopper.shardName)
  }

  def receive = {
    case cmd: Shopper.Command =>
      shardedShopper forward cmd
  }
}
```



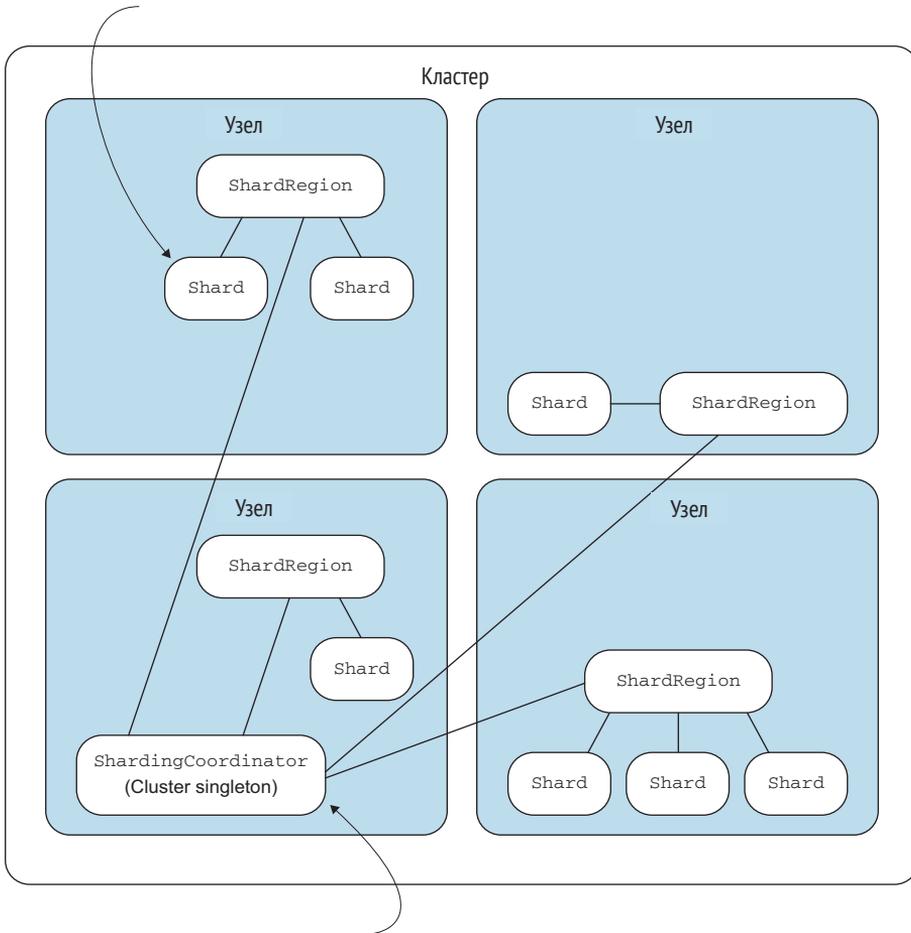
**Рис. 15.7.** Изменения для перехода от расширения cluster singleton к расширению cluster sharding

Актор `ShardedShoppers` запускает расширение `ClusterSharding`, передавая всю информацию, необходимую для запуска `ShardedShopper` где-то в кластере. `typeName` – это имя типа распределенного актора. Распределен-

ный актер также называется *входом* (entry), именно поэтому для параметра выбрано имя `entryProps`. Расширение `ClusterSharding` предоставляет ссылку на актер `ShardRegion` группы, в которой должен быть создан актер `ShardedShopper`, который будет пересылать ему сообщения.

На каждом узле запускается свой актер `ShardRegion`; но решение, какой из них будет управлять группой, принимает координатор `ShardingCoordinator` (который выполняется в акторе в единственном экземпляре), как показано на рис. 15.8.

**Каждый актер `Shard` может управлять множеством подчиненных распределенных акторов; в данном примере множеством акторов `ShardedShoppers`**



**ShardingCoordinator решает, какой ShardRegion будет управлять теми или иными актерами Shard**

Рис. 15.8. Группы в кластере

Актор `ShardRegion` управляет несколькими акторами `Shard`, которые фактически представляют группы распределенных акторов. Именно актор `Shard` создает распределенные акторы из шаблонов, переданных в вызов метода `ClusterSharding.start`. Однако вас не должен беспокоить тот факт, что между вашими распределенными акторами и актором `ShardRegion` находится еще один актор `Shard`.

В листинге 15.35 показана распределенная версия актора `Shopper` – актор `ShardedShopper`.

#### Листинг 15.35. `ShardedShopper`

```
package aia.persistence.sharded

import aia.persistence._
import akka.actor._
import akka.cluster.sharding.ShardRegion
import akka.cluster.sharding.ShardRegion.Passivate

object ShardedShopper {
  def props = Props(new ShardedShopper)
  def name(shopperId: Long) = shopperId.toString

  case object StopShopping

  val shardName: String = "shoppers"

  val extractEntityId: ShardRegion.ExtractEntityId = {
    case cmd: Shopper.Command => (cmd.shopperId.toString, cmd)
  }

  val extractShardId: ShardRegion.ExtractShardId = {
    case cmd: Shopper.Command => (cmd.shopperId % 12).toString
  }
}

class ShardedShopper extends Shopper {
  import ShardedShopper._

  context.setReceiveTimeout(Settings(context.system).passivateTimeout)

  override def unhandled(msg: Any) = msg match {
    case ReceiveTimeout =>
      context.parent ! Passivate(stopMessage = ShardedShopper.StopShopping)
    case StopShopping => context.stop(self)
  }
}
```

Объект-компаньон `ShardedShopper` определяет две важные функции: `ExtractEntityID`, извлекающую идентификатор из команды, и `ExtractShardId`, создающую уникальный идентификатор группы для каждой команды. В данном случае мы просто использовали для этого `shopperId`. Мы можем быть уверены, что в кластере никогда не появятся дубликаты акторов `ShardedShopper`.

Обратите внимание, что `ShardedShoppers` в листинге 15.34 не запускает акторы `ShardedShopper`, в отличие от `singleton`-версии `Shoppers`. Модуль `ClusterSharding` автоматически запустит `ShardedShopper` при попытке переслать ему команду. Он извлечет идентификаторы `shopperId` и `shardId` из команды и создаст соответствующий экземпляр `ShardedShopper`, используя `entryProps`, переданный ранее. Все последующие команды будут передаваться этому актору `ShardedShopper`.

Класс `ShardedShopper` просто наследует класс `Shopper` и определяет лишь, что должно происходить, если команды не поступают долгое время. Актор `ShardedShopper` может перейти в пассивное состояние, когда не используется долгое время, чтобы освободить занятую память.

Актор `ShardedShopper` предлагает актору `Shard` перевести его в пассивное состояние, если долго не получает никаких команд. Актор `Shard`, в свою очередь, посылает сообщение `stopMessage` актору `ShardedShopper`, запросившему переход в пассивное состояние, благодаря чему тот останавливается. Это сообщение напоминает `PoisonPill`: перед остановкой `ShardedShopper` обрабатывает все сообщения, находящиеся в очереди `ShardRegion`.

Опробовать решение с распределенными акторами можно, выполнив команду `sbt run` и выбрав для запуска класс `aia.persistence.sharded.ShardedMain`. Так же, как в главе 14, вы должны изменить порт, который прослушивает приложение. Настройку использования Apache Cassandra (или чего-то подобного) в роли журнала мы оставляем читателям в качестве самостоятельного упражнения.

А теперь подведем некоторые итоги: мы начали эту главу с создания приложения покупательской корзины, которое могло выполняться только на одном узле. Чтобы запустить приложение как расширение `cluster singleton`, потребовалось внести совсем немного изменений. На следующем шаге мы распределили акторы по всему кластеру, и для этого тоже не пришлось прикладывать значительных усилий, что еще раз показало выгоды использования подходов на основе сообщений в Akka.

Одна из причин такой простоты перехода от локальной версии к распределенной состоит в том, что команды уже содержат `shopperId`, который необходим для кластерной версии.

## 15.4. В заключение

- Технология Event Sourcing (история событий) оказалась простой и удобной для хранения состояния акторов. Хранимые акторы преобразуют допустимые команды в события и сохраняют их в журнал; в случае аварии эти события используются для восстановления состояния. Хранимые акторы относительно легко поддаются тестированию (при условии, что для удаления журнала используется простой набор функций).
- Не имеет смысла хранить все события в журнале на том же узле, где действует актор, потому что легко можно потерять все данные, если этот узел выйдет из строя. Использование журнала на основе распределенной базы данных повышает отказоустойчивость.
- Запуск всех акторов на одном узле – тоже не самое лучшее решение, потому что в случае выхода узла из строя приложение тут же станет недоступно.
- Для переноса акторов с аварийного узла на действующий можно использовать расширение `cluster singleton`. Мы показали, что это не требует больших усилий.
- Применение расширения `cluster sharding` поднимает доступность на еще более высокий уровень, позволяя снять ограничение на объем памяти, которой располагает единственный узел. Распределенные акторы создаются по требованию и могут переходить в пассивное состояние после долгого простоя. Это позволяет гибко распределять нагрузку на узлы в кластере. И снова для перехода к распределенной версии не потребовалось много усилий.

# Глава 16

## Советы по повышению производительности

В этой главе:

- ключевые параметры производительности в системах акторов;
- устранение узких мест, отрицательно влияющих на производительность;
- оптимизация использования CPU настройкой диспетчера;
- оптимизация использования пулов потоков;
- увеличение производительности изменением стратегии освобождения потоков.

К настоящему моменту мы рассмотрели широкий спектр возможностей акторов Akka. Мы начали с преобразования традиционных приложений в системы акторов, посмотрели, как организовать поддержку состояний и обработку ошибок, а также исследовали приемы интеграции с внешними системами и сохранения состояния акторов в базах данных. Кроме того, вы имели возможность увидеть, как масштабировать приложения с использованием кластеров. Мы использовали акторы Akka как черные ящики: посылали сообщения по ссылкам `ActorRef` и обрабатывали их в своих реализациях метода `receive`. Отсутствие необходимости знать внутреннее устройство является одной из самых сильных сторон Akka. Но иногда все же бывает желательно контролировать параметры, управляющие производительностью системы акторов. В этой главе мы покажем приемы настройки Akka для увеличения общей производительности.

Настройка производительности – сложная задача, и уникальная для каждого приложения, потому что разные приложения обычно имеют разные требования к производительности, и все компоненты системы по-разному влияют друг на друга. В общем случае нужно выяснить, какая часть ра-

ботает особенно медленно и почему, а затем на основе этой информации искать решение. В этой главе мы сосредоточимся на увеличении производительности за счет настройки механизма потоков выполнения, в которых действуют акторы.

Ниже приводится план этой главы.

- Краткое введение в приемы управления производительностью и знакомство с основными показателями.
- Оценка производительности акторов в системе путем создания собственного почтового ящика и трейта Actor. Обе реализации создают статистические сообщения, помогающие в поиске проблемных участков.
- Следующий шаг – решение проблем. Для начала мы опишем разные варианты увеличения производительности одного актора.
- Но иногда достаточно просто организовать более эффективное использование ресурсов. В последних разделах мы сосредоточимся на управлении потоками выполнения. Сначала обсудим, как убедиться, что проблемы действительно связаны с неэффективным использованием потоков выполнения; затем рассмотрим разные решения, основанные на изменении настроек диспетчера, используемого акторами. После этого расскажем, как настроить актор для одновременной обработки множества сообщений в одном и том же потоке выполнения. Изменяя настройки, вы сможете увеличить производительность отдельных акторов за счет справедливости распределения процессорного времени между ними.
- В заключение мы покажем, как создать своего диспетчера для динамического создания нескольких пулов потоков.

## 16.1. Анализ производительности

Для решения проблем производительности важно понимать их причины и как разные части системы взаимодействуют друг с другом. Научившись понимать механику, вы сможете определять, что нужно измерять при анализе системы, находить проблемы производительности и решать их. В этом разделе вы узнаете, как меняется производительность и какие показатели помогают оценить общую производительность системы.

Начнем с определения областей в системе, влияющих на производительность, а потом познакомимся с основными показателями.

### 16.1.1. Производительность системы

Как показывает опыт, даже при всей сложности определения, какие из множества взаимодействующих компонентов в наибольшей степени ограничивают производительность, часто лишь очень ограниченная

их часть действительно влияет на общую производительность системы. Здесь действует известный *принцип Парето* (более известный как *правило 80/20*): 80% увеличения производительности можно добиться оптимизацией лишь 20% системы. Это одновременно и хорошая, и плохая новость. Хорошо, что для увеличения производительности требуется внести небольшие изменения. Плохо, что изменение только этих 20% действительно повлияет на производительность. Такие компоненты, ограничивающие производительность всей системы, называют *узкими местами*.

Возьмем за основу простой пример из главы 8, где мы занимались реализацией шаблона конвейеров и фильтров. В примере с дорожной камерой мы использовали два фильтра, создав конвейер, изображенный на рис. 16.1.

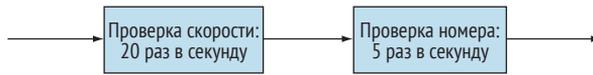


Рис. 16.1. Конвейер с узким местом

Взглянув на эту систему, легко обнаружить узкое место. Шаг «проверка номера» может обрабатывать не более 5 изображений в секунду, тогда как первый шаг может повторяться до 20 раз в секунду. Здесь четко видно, что производительность всей цепочки определяет единственный шаг, то есть узкое место в этой системе – шаг проверки номера.

Когда нашей простой системе требуется обработать два изображения в секунду, проблема производительности вообще не возникает, потому что система легко справляется с этим объемом информации и даже имеет немалый запас. В любой системе есть некоторая ее часть, про которую можно сказать, что она ограничивает производительность, но она является узким местом, только если пропускная способность системы оказывается ниже необходимого уровня.

Поэтому мы обычно продолжаем устранять узкие места, пока не достигнем желаемого уровня производительности. Но есть одна загвоздка. Устранение первого узкого места дает самое большое увеличение производительности. Устранение второго узкого места дает уже меньший эффект (закон убывания отдачи).

Это связано с тем, что с каждым изменением система становится все более сбалансированной и приближается к физическому пределу эффективности использования ресур-



Рис. 16.2. Эффект увеличения производительности по мере устранения узких мест

сов. На рис. 16.2 можно видеть, как производительность достигает предела по мере устранения узких мест. Чтобы получить производительность выше этого предела, нужно увеличить объем доступных ресурсов, например за счет масштабирования по горизонтали.

Это является одной из причин, почему усилия должны быть сосредоточены на требованиях. Как показывают многочисленные исследования, программисты склонны тратить много времени на оптимизацию кода, мало влияющего на общую производительность системы. Akka помогает решить эту проблему, приближая модель к требованиям; мы говорим здесь о понятиях, которые напрямую транслируются в практическую область, – номера, скорости и т. д.

До сих пор мы говорили о производительности в целом, однако проблемы производительности делятся на два типа:

- *низкая пропускная способность* – когда количество запросов, которое может быть обработано в единицу времени, недостаточно велико, как в случае с проверкой автомобильных номеров в нашем примере;
- *высокие задержки* – каждый запрос обрабатывается слишком долго; например, отображение запрошенной веб-страницы занимает слишком много времени.

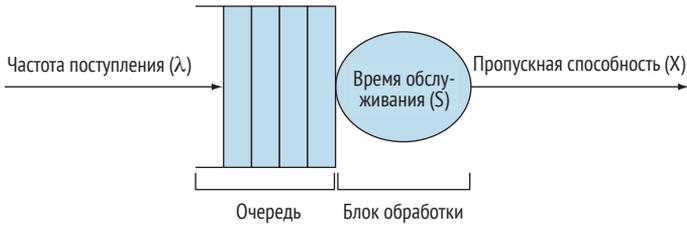
Обе эти проблемы большинство людей называют проблемами производительности. Но для их решения может потребоваться совершенно разное время. Проблема низкой пропускной способности обычно решается горизонтальным масштабированием, но проблема высокой задержки обычно требует переделки архитектуры приложения. Решение проблем производительности акторов будет главной темой раздела 16.3, где мы покажем, как увеличить производительность устранением узких мест. Но сначала вы должны больше узнать о показателях производительности и факторах, влияющих на них. Вы только что видели два из таких показателей: пропускная способность и задержки. Мы подробно опишем их и некоторые другие показатели, чтобы вы получили полное понимание, как можно влиять на производительность, прежде чем мы перейдем к приемам повышения производительности в разделе 16.3.

Учитывая тот факт, что наша система состоит из акторов, обменивающихся сообщениями, а не из классов и функций, мы будем опираться на уже имеющиеся у вас знания о производительности в мире Akka.

### 16.1.2. Показатели производительности

При изучении характеристик производительности компьютерных систем вам встретится много новых терминов. Мы начнем с описания наиболее важных из них. Затем рассмотрим работу единственного актора, включая почтовый ящик, как показано на рис. 16.3. На этом рисунке пока-

заны три наиболее важных показателя производительности: частота поступления, пропускная способность и время обслуживания.



**Рис. 16.3.** Узел с актором

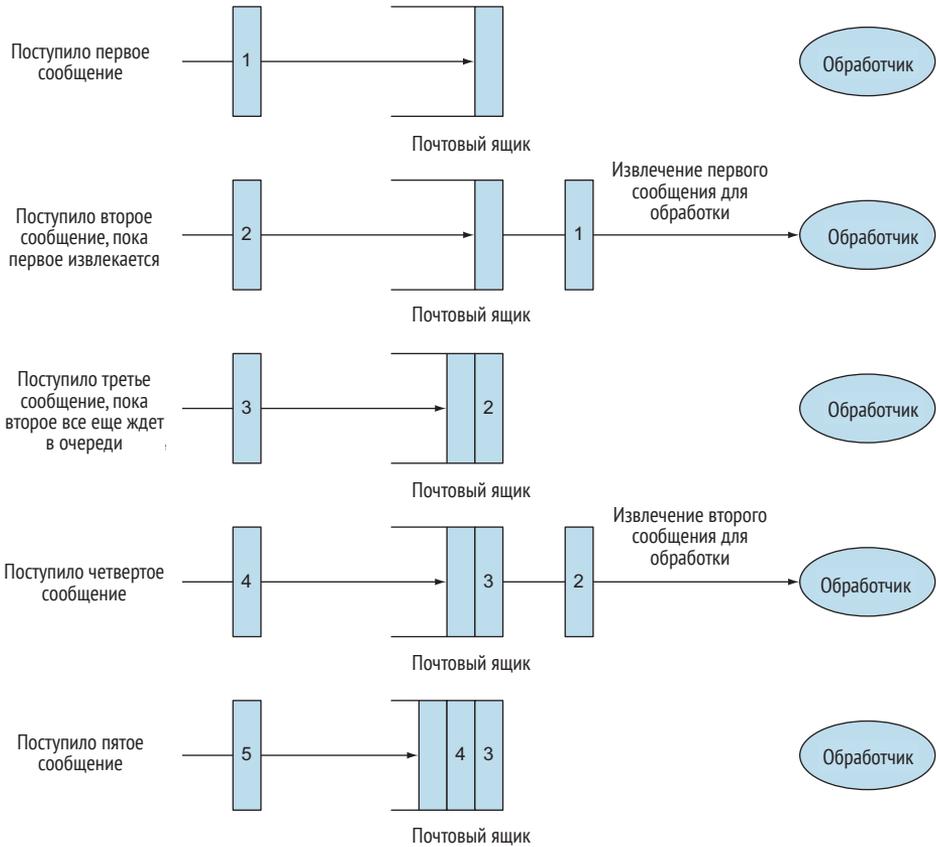
Начнем с *частоты поступления*. Это количество заданий или сообщений, поступающих в единицу времени. Например, если в течение двух секунд поступило восемь сообщений, частота поступления составит четыре сообщения в секунду.

Следующий показатель: скорость обработки. Он называется *пропускной способностью* актора. Пропускная способность – это количество заданий или сообщений, обработанных в единицу времени. Многие знакомы с этим термином, если не в связи с регулировкой производительности, то, по крайней мере, как с количеством успешно обработанных сетевых пакетов. Как показано на рис. 16.4, когда система сбалансирована (вверху на рисунке), она способна обработать все поступающие задания без задержек. Это означает, что частота поступления равна пропускной способности (или, по крайней мере, не превосходит ее). Когда служба не сбалансирована (ниже на рисунке), неизбежно возникают периоды ожидания, потому что все обработчики оказываются перегруженными. То есть в действительности системы, основанные на обмене сообщениями, ничем не отличаются от пулов потоков выполнения (в чем вы убедитесь ниже).

В этом случае узел не успевает обрабатывать поступающие сообщения, из-за чего они накапливаются в почтовом ящике. Это – классическая проблема производительности. Однако важно заметить, что наша цель – не ликвидация ожидания; если в системе исчезнут ожидающие задания, у нас появятся простаивающие и ничем не занятые обработчики. Оптимальный уровень производительности находится где-то посередине – каждый раз, когда обработка задания завершается, обработчику тут же предлагается новое, но время ожидания должно быть исчезающе малым.

Последний показатель, изображенный на рис. 16.3, – *время обслуживания*. Время обслуживания – это время, необходимое для обработки одного задания. Иногда в подобных моделях упоминается термин *частота обслуживания*. Это среднее количество заданий, обработанных в единицу времени, обозначается символом  $\mu$ . Время обслуживания ( $S$ ) и частота обслуживания связаны отношением:

$$\mu = 1 / S.$$



**Рис. 16.4.** Несбалансированный узел. Частота поступления выше пропускной способности

Время обслуживания тесно связано с понятием задержки – временем между входом и выходом. Разность между временем обслуживания и задержкой – это время ожидания в почтовом ящике. Когда сообщения не ждут завершения обработки других сообщений, находясь в почтовом ящике, время обслуживания равно времени задержки.

Последний показатель, который часто используется при анализе производительности, – *потребление времени*. Это доля времени, когда узел занят обработкой сообщений. Когда потребление времени составляет 50%, значит, процесс занимался обработкой заданий 50% времени, а 50% времени простаивал. Потребление времени помогает понять, какой объем работы теоретически может выполнять система. И когда потребление времени равно 100%, значит, система не сбалансирована или перегружена. Почему? Потому что с ростом спроса тут же возникнет необходимость ожидания.

Это наиболее важные термины, имеющие отношение к производительности. Внимательный читатель наверняка сделает вывод, что размер оче-

реди – важный показатель, указывающий на появление проблемы. Когда размер очереди растет, это означает, что актор перегружен и сдерживает всю систему.

Теперь, когда вы узнали значение разных показателей производительности и их взаимосвязь друг с другом, мы можем перейти к объяснению приемов устранения проблем производительности. Первым делом нужно отыскать акторы, имеющие проблемы производительности. В следующем разделе мы представим возможные способы поиска узких мест в системе акторов.

## 16.2. Оценка производительности акторов

Прежде чем приступать к увеличению производительности системы, нужно выяснить особенности ее поведения. Как рассказывалось в разделе 16.1.1, изменения следует вносить только в проблемные области, а значит, вы должны выявить их. Для этого требуется оценить производительность всех частей системы. Вы уже знаете, что увеличение размеров очередей и потребляемого времени являются важными индикаторами проблем в акторах. Но как получить эту информацию из приложения? В этом разделе мы покажем, как конструировать свои средства оценки производительности.

Размеры очередей и потребляемое время – это два отдельных набора данных. Размеры очередей можно получить из почтовых ящиков, а для оценки потребляемого времени необходимо организовать сбор статистики о работе обрабатывающего блока. На рис. 16.5 показаны моменты времени, интересующие нас больше всего, в период от отправки сообщения до завершения его обработки актором.

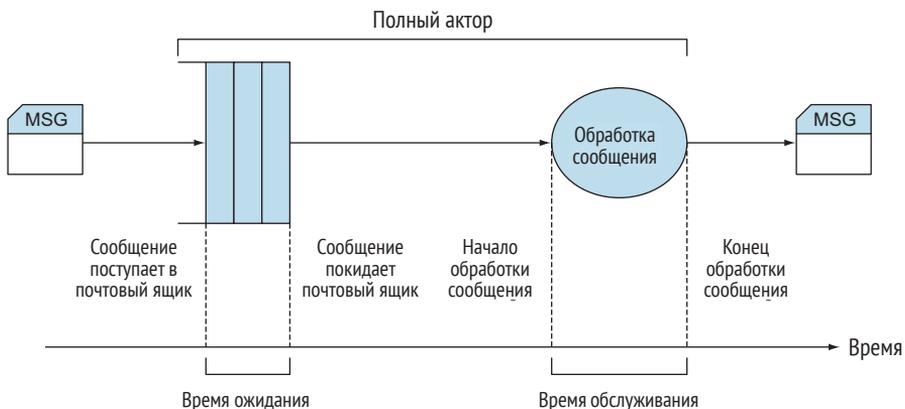


Рис. 16.5. Важные моменты времени в процессе обработки сообщения

Для оценки производительности своих акторов Akka вы должны получить следующие данные (из почтового ящика Akka):

- когда сообщение поступило и было добавлено в почтовый ящик;
- когда оно было передано для обработки, то есть извлечено из почтового ящика и вручено обрабатывающему блоку;
- когда обработка сообщения завершилась и оно покинуло обрабатывающий блок.

Определив эти времена для каждого сообщения, вы сможете получить показатели производительности, необходимые для анализа. Например, задержка – это разность между временем поступления и передачей дальше. В этом разделе мы создадим пример, получающий эту информацию. Начнем с создания своего почтового ящика, который извлекает данные, необходимые для трассировки сообщений в почтовом ящике. Во второй части мы создадим трейт, собирающий статистическую информацию о работе метода `receive`. Оба примера будут посылать собранные сведения в `EventStream`. В зависимости от требований вы можете просто журналировать эти сообщения или подвергать их какой-то предварительной обработке. Здесь мы не будем описывать, как накапливать эти сообщения, но знаний, полученных вами к данному моменту, вполне достаточно, чтобы самостоятельно реализовать это.

**МИКРОТЕСТИРОВАНИЕ ПРОИЗВОДИТЕЛЬНОСТИ.** Распространенная проблема поиска проблем производительности заключается в таком добавлении кода, выполняющего измерения, чтобы он не влиял на результаты измерений.

Простое добавление отладочных инструкций `println` в код, измеряющих время с помощью `System.currentTimeMillis`, во многих случаях позволяет получить примерные оценки. Но в других случаях он совершенно не годится. Когда необходимо получить более точные оценки, используйте инструменты микротестирования производительности, такие как JMH (<http://openjdk.java.net/projects/code-tools/jmh/>).

### 16.2.1. Сбор данных в почтовом ящике

Из почтового ящика мы должны получить максимальный размер очереди и среднее время ожидания. Для этого нужно знать, когда сообщение поступает в очередь и когда покидает ее. Сначала создадим свой почтовый ящик. В этом почтовом ящике мы будем собирать данные и посылать их с помощью `EventStream` актору, который будет преобразовывать их в статистики производительности, необходимые для выявления узких мест.

Для создания своего почтового ящика и его использования нам потребуются два компонента. Первый – очередь сообщений, которая будет играть роль почтового ящика, и второй – фабричный класс, создающий почтовый ящик по требованию. На рис. 16.6 показана диаграмма классов нашей реализации почтового ящика.

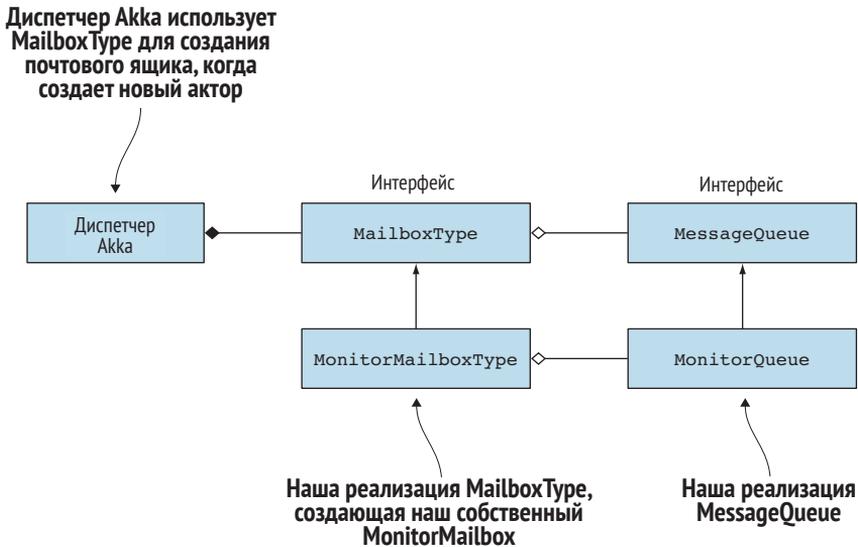


Рис. 16.6. Диаграмма классов реализации собственного почтового ящика

Для создания новых почтовых ящиков диспетчер Akka использует фабричный класс (реализующий трейт MailboxType). Выбирая реализацию MailboxType, можно создавать разные почтовые ящики. Чтобы получить свой почтовый ящик, мы должны реализовать MessageQueue и MailboxType. Приступим.

### Создание собственного почтового ящика

Чтобы создать свой почтовый ящик, нужно реализовать трейт MessageQueue.

- `def enqueue(receiver: ActorRef, handle: Envelope)` – этот метод вызывается при попытке добавить `Envelope`, содержащий ссылку на отправителя и само сообщение;
- `def dequeue(): Envelope` – этот метод вызывается для получения следующего сообщения;
- `def numberOfMessages: Int` – хранит текущее количество сообщений в очереди;
- `def hasMessages: Boolean` – признак присутствия сообщений в очереди;
- `def cleanUp(owner: ActorRef, deadLetters: MessageQueue)` – этот метод вызывается при удалении почтового ящика; обычно он должен отправить все оставшиеся сообщения в очередь недоставленных сообщений.

Мы реализуем свою очередь `MonitorQueue`, которая будет создаваться классом `MonitorMailboxType`. Но сначала определим case-класс `MailboxSta-`

tistics, содержащий данные трассировки, необходимые для вычисления статистик. Мы также определим класс `MonitorEnvelope` для хранения данных трассировки в процессе сбора данных, пока сообщение ожидает обработки в почтовом ящике, как показано в листинге 16.1.

**Листинг 16.1.** Контейнеры данных для хранения статистик почтового ящика

```
case class MonitorEnvelope(queueSize: Int,
                          receiver: String,
                          entryTime: Long,
                          handle: Envelope)
case class MailboxStatistics(queueSize: Int,
                             receiver: String,
                             sender: String,
                             entryTime: Long,
                             exitTime: Long)
```

← Сообщение для отправки данных трассировки

← Конверт Envelope для сбора данных трассировки

Класс `MailboxStatistics` содержит поля: `receiver`, представляющее актор для мониторинга; `entryTime` и `exitTime`, определяющие время, когда сообщение прибыло в почтовый ящик и покинуло его. Поле `queueSize`, если честно, не особенно нужно, потому что размер очереди можно вычислить из статистик, но нам будет проще иметь такое поле.

Класс `MonitorEnvelope` будет обрабатывать оригинальные конверты `Envelope`, получаемые от фреймворка Akka. Теперь можно создать `MonitorQueue`.

Конструктор класса `MonitorQueue` будет принимать параметр `system`, чтобы потом с его помощью можно было получить `eventStream`, как показано в листинге 16.2. Мы также должны определить семантику, которую будет поддерживать наша очередь. Поскольку этот почтовый ящик предполагается использовать для всех акторов в системе, мы используем семантику `UnboundedMessageQueueSemantics` и `LoggerMessageQueueSemantics`. Последняя необходима по той простой причине, что ее должны поддерживать все акторы Akka, осуществляющие журналирование.

**Листинг 16.2.** Реализация трейта `MessageQueue` и подмешивание семантик

```
class MonitorQueue(val system: ActorSystem)
  extends MessageQueue
  with UnboundedMessageQueueSemantics
  with LoggerMessageQueueSemantics {
  private final val queue = new ConcurrentLinkedQueue[MonitorEnvelope]()
```

Позднее мы будем использовать `system.eventStream` для публикации статистик

← Семантики, необходимые для поддержки работы со стандартными акторами

← Семантики, необходимые для поддержки журналирования Akka

← Тип очереди, используемой внутренне

**ВЫБОР ПОЧТОВОГО ЯЩИКА С ИСПОЛЬЗОВАНИЕМ КОНКРЕТНОЙ СЕМАНТИКИ ОЧЕРЕДИ СООБЩЕНИЙ.**

Трейты, определяющие используемые семантики, служат простыми маркерами (они не определяют никаких методов). В данном случае нам не потребовалось определять своих семантик, но иногда это может быть удобно, потому что актор может потребовать наличия конкретных семантик для использования `RequiresMessageQueue`; например, `DefaultLogger` требует подмешивания семантики `LoggerMessageQueueSemantics` в трейт `RequiresMessageQueue[LoggerMessageQueueSemantics]`. Связать почтовый ящик с семантиками можно посредством конфигурационного параметра `akka.actor.mailbox.requirements`.

Далее мы реализуем метод `enqueue`, создающий `MonitorEnvelope` и добавляющий его в очередь.

**Листинг 16.3.** Реализация метода `enqueue` трейта `MessageQueue`

```
def enqueue(receiver: ActorRef, handle: Envelope): Unit = {
  val env = MonitorEnvelope(queueSize = queue.size() + 1,
    receiver = receiver.toString(),
    entryTime = System.currentTimeMillis(),
    handle = handle)
  queue add env
}
```

В `queueSize` передается текущий размер очереди, увеличенный на 1, потому что это новое сообщение еще не добавлено в очередь.

Теперь реализуем метод `dequeue`. Он проверяет тип очередного сообщения, и если оно является экземпляром `MailboxStatistics`, пропускает его, потому что мы предполагаем использовать `MonitorQueue` для всех потовых ящиков, и если не исключить такие сообщения, это приведет к рекурсивному созданию новых сообщений `MailboxStatistics`, когда они будут использоваться нашим сборщиком статистической информации. Реализация метода `dequeue` показана в листинге 16.4.

**Листинг 16.4.** Реализация метода `dequeue` трейта `MessageQueue`

```
def dequeue(): Envelope = {
  val monitor = queue.poll()
  if (monitor != null) {
    monitor.handle.message match {
      case stat: MailboxStatistics => //пропустить сообщение
      case _ => {
        val stat = MailboxStatistics(
          queueSize = monitor.queueSize,
          receiver = monitor.receiver,
```

Пропустить `MailboxStatistics`, чтобы избежать рекурсивной отправки сообщения

Послать `MailboxStatistics` в поток событий

```

        sender = monitor.handle.sender.toString(),
        entryTime = monitor.entryTime,
        exitTime = System.currentTimeMillis()
    system.eventStream.publish(stat)
    }
}
monitor.handle
} else {
    null
}
}
}

```

← Вернуть системе Akka оригинальный конверт

← Вернуть null в отсутствие ожидающих сообщений

Обработывая обычное сообщение, мы создаем `MailboxStatistics` и посылаем его в `EventStream`. В отсутствие сообщений нужно просто вернуть `null` как признак отсутствия сообщений.

Теперь мы реализовали всю основную функциональность, и нам осталось добавить только методы поддержки, объявленные в трейте `MessageQueue`.

### Листинг 16.5. Завершение реализации трейта `MessageQueue`

```

def numberOfMessages = queue.size
def hasMessages = !queue.isEmpty

```

← Реализация `hasMessages`

← Возвращает количество конвертов в очереди

```

def cleanUp(owner: ActorRef, deadLetters: MessageQueue): Unit = {
    if (hasMessages) {
        var envelope = dequeue
        while (envelope ne null) {
            deadLetters.enqueue(owner, envelope)
            envelope = dequeue
        }
    }
}
}

```

← При очистке очереди послать все сообщения из нее в очередь недоставленных сообщений

Мы используем метод `dequeue`, поэтому статистики тоже будут создаваться.

Наш трейт почтового ящика готов к использованию в фабричном классе, который описывается в следующем разделе.

## Реализация `MailboxType`

Фабричный класс, который создает фактический почтовый ящик, реализует трейт `MailboxType`. Этот трейт определяет только один метод, но нам также необходим особый конструктор, поэтому его тоже нужно добавить в интерфейс `MailboxType`. Таким образом, мы получаем следующий интерфейс:

- `def this(settings: ActorSystem.Settings, config: Config)` – конструктор, используемый фреймворком Akka для создания экземпляров `MailboxType`;
- `def create(owner: Option[ActorRef], system: Option[ActorSystem]): MessageQueue` – метод, создающий новый почтовый ящик.

Чтобы получить возможность использовать свой почтовый ящик, мы должны реализовать этот интерфейс. В листинге 16.6 показана полная реализация `MailboxType`.

**Листинг 16.6.** Реализация `MailboxType` для использования нашего почтового ящика

```
class MonitorMailboxType(settings: ActorSystem.Settings, config: Config)
  extends akka.dispatch.MailboxType
  with ProducesMessageQueue[MonitorQueue]{
  final override def create(owner: Option[ActorRef],
                             system: Option[ActorSystem]): MessageQueue = {
    system match {
      case Some(sys) =>
        new MonitorQueue(sys)
      case _ =>
        throw new IllegalArgumentException("requires a system")
    }
  }
}
```

Реализация конструктора для использования фреймворком Akka

Создает очередь `MonitorQueue` и передает ей ссылку на систему

В отсутствие системы акторов `ActorSystem` нельзя создать и использовать `MessageQueue`

Если ссылка на систему акторов не была указана, возбуждается исключение, потому что она необходима для работы очереди. Теперь, закончив реализацию своего почтового ящика, перейдем к конфигурации фреймворка Akka и настроим его на использование этого почтового ящика.

### Настройка почтовых ящиков

Настроить использование другого почтового ящика можно в файле `application.conf`. Включить почтовые ящики в работу можно множеством способов. Тип почтового ящика привязывается к диспетчеру, поэтому можно создать новый тип диспетчера и использовать в нем наш почтовый ящик. Для этого следует определить параметр `mailbox-type` в файле `application.conf` и использовать конфигурацию диспетчера при создании новых акторов, как показано в следующем фрагменте:

```
my-dispatcher {
  mailbox-type = aia.performance.monitor.MonitorMailboxType
}
val a = system.actorOf(
  ← Установка типа почтового ящика в настройках диспетчера в файле application.conf
```

```
Props[MyActor].withDispatcher("my-dispatcher")
```

← Создание актора с применением наших настроек диспетчера

Выше уже упоминалось, что существуют другие способы заставить Акка использовать наш почтовый ящик. Почтовый ящик все еще привязывается к выбранному диспетчеру, но также можно определить, какой почтовый ящик должен использовать диспетчер по умолчанию. В этом случае отпадает необходимость что-либо менять в коде создания акторов. Чтобы подменить почтовый ящик, используемый диспетчером по умолчанию, добавьте следующие строки в конфигурационный файл:

```
akka {
  actor {
    default-mailbox {
      mailbox-type = "aia.performance.monitor.MonitorMailboxType"
    }
  }
}
```

После этого все акторы будут использовать наш собственный почтовый ящик. Давайте посмотрим, все ли работает, как задумывалось.

Для проверки почтового ящика нам понадобится актор, за которым мы будем наблюдать. Создадим что-нибудь простое: этот актор будет выполнять задержку перед приемом сообщения, имитируя обработку. Эта задержка будет служить временем обслуживания нашей модели.

```
class ProcessTestActor(serviceTime:Duration) extends Actor {
  def receive = {
    case _ => {
      Thread.sleep(serviceTime.toMillis)
    }
  }
}
```

Теперь, создав актор для мониторинга, пошлем ему несколько сообщений.

#### Листинг 16.7. Тестирование почтового ящика

```
val statProbe = TestProbe()
system.eventStream.subscribe(
  statProbe.ref,
  classOf[MailboxStatistics])
val testActor = system.actorOf(Props(
  new ProcessTestActor(1.second)), "monitorActor2")
statProbe.send(testActor, "message")
statProbe.send(testActor, "message2")
```

← Актор создается как обычно  
← Отправка трех сообщений

```

statProbe.send(testActor, "message3")
val stat = statProbe.expectMsgType[MailboxStatistics]

stat.queueSize must be(1)
val stat2 = statProbe.expectMsgType[MailboxStatistics]

stat2.queueSize must (be(2) or be(1))
val stat3 = statProbe.expectMsgType[MailboxStatistics]

stat3.queueSize must (be(3) or be(2))

```

Последнее сообщение должно  
увеличить размер очереди до 3  
или 2, в зависимости от того,  
успело ли первое сообщение  
покинуть очередь

Как видите, мы получили `MailboxStatistics` для каждого сообщения, отправленного в `EventStream`. На данный момент у нас имеется законченный код трассировки почтового ящика наших акторов. Мы вынуждены были создать свой почтовый ящик, чтобы передавать данные трассировки из почтового ящика в `EventStream`, и узнали, что для этого необходимы фабричный класс и тип почтового ящика. В конфигурации можно определить, какой фабричный класс должен использоваться для создания почтового ящика при создании нового актора. Теперь, когда у нас есть возможность выполнить трассировку почтового ящика, перейдем к обработке сообщений.

### 16.2.2. Сбор и обработка данных

Данные трассировки производительности можно извлекать, переопределив метод `receive` актора. В этом примере мы должны изменить метод `receive` исследуемого актора, что немного сложнее, потому что новая функциональность должна добавляться без изменения оригинального кода. Поэтому в следующем примере мы определим трейт, который будет добавляться в каждый актор, подлежащий трассировке. И снова начнем с определения сообщения со статистической информацией:

```

case class ActorStatistics( receiver: String,
                           sender: String,
                           entryTime: Long,
                           exitTime: Long)

```

Здесь `receiver` – это исследуемый актор, а `entryTime` и `exitTime` – наблюдаемые статистики (время входа и выхода сообщения). Мы также добавили параметр `sender`, чтобы получить больше информации об обрабатываемых сообщениях, но мы не используем его в этих примерах. Теперь, определив `ActorStatistics`, можно реализовать функциональность созданием трейта, который переопределяет метод `receive`.

#### Листинг 16.8. Метод `receive` исследуемого актора

```

trait MonitorActor extends Actor {

  abstract override def receive = {

```

```

case m: Any => {
  val start = System.currentTimeMillis()
  super.receive(m)
  val end = System.currentTimeMillis()

  val stat = ActorStatistics(
    self.toString(),
    sender.toString(),
    start,
    end)

  context.system.eventStream.publish(stat)
}
}
}

```

← Вызов метода `receive` актора

← Создание и отправка статистик

Мы используем абстрактное переопределение, чтобы внедриться между актором и фреймворком Akka. Благодаря этому мы получаем возможность получить время начала и конца обработки сообщения. После обработки сообщения мы создаем `ActorStatistics` и посылаем его в поток событий.

Теперь нужно лишь подмешать трейт в создаваемые акторы:

```

val testActor = system.actorOf(Props(
  new ProcessTestActor(1.second) with MonitorActor,
  "monitorActor"))

```

В ответ на отправку сообщения актору `ProcessTestActor` в поток `EventStream` должно быть отправлено сообщение `ActorStatistics`.

### Листинг 16.9. Тестирование трейта `MonitorActor`

```

val statProbe = TestProbe()
system.eventStream.subscribe(
  statProbe.ref,
  classOf[ActorStatistics])

val testActor = system.actorOf(Props(
  new ProcessTestActor(1.second) with MonitorActor,
  "monitorActor"))

statProbe.send(testActor, "message")

val stat = statProbe.expectMsgType[ActorStatistics]
stat.exitTime -
  stat.entryTime must be (1000L plusOrMinus 10)

```

← Создание актора с трейтом `MonitorActor`

← Результат должен быть близок к времени обслуживания, равному 1 секунде

И в точности как ожидалось, время обработки (время выхода минус время входа) близко к времени обслуживания в нашем тестовом акторе.

На данный момент мы получили также возможность выполнять трассировку обработки сообщений. Теперь у нас есть трейт, который создает данные трассировки и посылает их с использованием `EventStream`. Можно приступить к анализу данных и поиску акторов, страдающих проблемами производительности. После выявления узких мест можно начинать их устранение. В следующих разделах мы рассмотрим разные способы решения проблем.

## 16.3. Улучшение производительности устранением узких мест

Чтобы улучшить производительность системы, достаточно лишь улучшить производительность в узких местах. Существуют разные решения этой проблемы, но одни в большей степени влияют на пропускную способность, а другие – на задержку. В зависимости от требований и реализации вы можете выбрать наиболее подходящее решение. Когда узким местом является ограниченность ресурса, совместно используемого несколькими акторами, вы должны перераспределить его так, чтобы большая часть ресурса была отдана задачам, наиболее важным для системы, а меньшая – задачам, которые могут подождать. Такое решение всегда заключается в поиске компромисса. Но если узкое место не связано с нехваткой ресурса, можете попробовать внести изменения в свою систему.

Взглянув на устройство узла, изображенного на рис. 16.7, можно заметить, что он состоит из двух частей: очереди и блока обработки. Обратите также внимание на два параметра производительности, исследованных в разделе 16.1.2: частоту поступления и время обслуживания. Мы добавили третий параметр, добавляющий больше экземпляров актора: *количество служб*. Фактически это прием масштабирования. Улучшить производительность актора можно изменением трех параметров:

- *количество служб* – увеличение количества служб увеличивает пропускную способность узла;
- *частота поступления* – уменьшение числа сообщений, поступающих в единицу времени, помогает легче справиться с ними;
- *время обслуживания* – ускорение обработки уменьшает задержку и позволяет обрабатывать больше сообщений, что также способствует увеличению пропускной способности.

Чтобы улучшить производительность, вы должны изменить один или несколько из этих параметров. На практике чаще всего увеличивают количество служб. Это действенный прием, когда проблема заключается в низкой пропускной способности и не связана с ограниченной мощностью процессора. Когда задача интенсивно использует процессор, увеличение

числа служб может, напротив, увеличить время обслуживания. Службы будут состязаться за процессорное время, и это может ухудшить общую производительность.

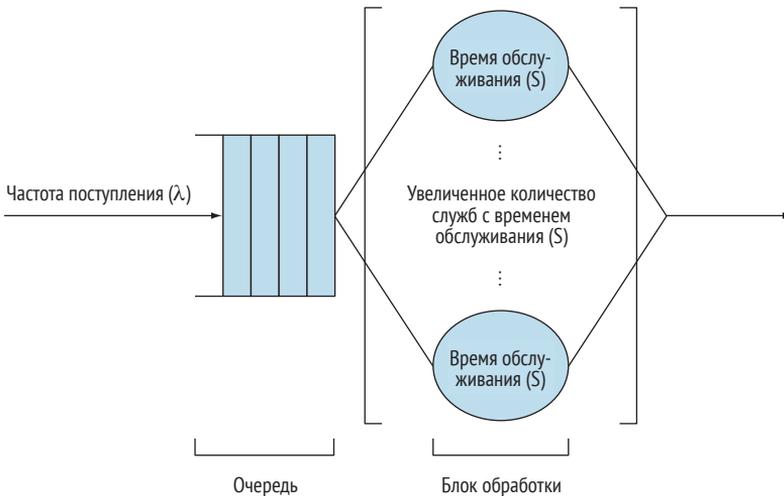


Рис. 16.7. Показатели производительности узла

Другое решение – уменьшение числа обрабатываемых заданий. Этот подход часто забывают, между тем, уменьшив частоту поступления, можно легко добиться существенного увеличения производительности. Часто для этого требуется пересмотреть архитектуру системы. Но это не всегда сопряжено с большими сложностями. В разделе 8.1.2, описывающем шаблон конвейеров и фильтров, вы видели, что простое изменение порядка выполнения двух операций может значительно улучшить производительность.

Последний подход – уменьшение времени обслуживания. Он увеличивает пропускную способность, уменьшает время отклика и всегда улучшает производительность. Однако это также самый сложный путь, потому что функциональность должна оставаться прежней, и часто довольно трудно убрать какие-то этапы, чтобы уменьшить время обслуживания. Выбирая этот путь, первым делом следует проверить использование блокирующих вызовов. Такие вызовы увеличивают время обслуживания, а избавиться от них обычно довольно просто: нужно переписать актор так, чтобы он использовал неблокирующие вызовы, реализовав модель актора, управляемого событиями. Другой способ уменьшить время обслуживания – распараллелить обработку. Распределив выполнение задачи между несколькими акторами и обеспечив их параллельное выполнение, используя, например, шаблон параллельной обработки дроблением, как рассказывалось в главе 8.

Но есть также другие подходы к увеличению производительности. Например, когда проблемой является ограниченность ресурсов сервера, таких как процессор, память или диск. Если уровень потребления этих ресурсов составляет 80% или больше, скорее всего, их нехватка тянет вашу систему назад. Это может быть связано с тем, что вы используете ресурсов больше, чем имеется. Устранить эту проблему можно покупкой более емкой и быстрой платформы или горизонтальным масштабированием. Этот подход также может потребовать изменения архитектуры, но благодаря поддержке горизонтального масштабирования в Akka это не должно быть большой проблемой, как было показано в главе 13 на примере кластеров.

Однако нехватка ресурсов не всегда означает, что вам необходимо что-то новое, более емкое и быстрое. Иногда нужно лишь разумно подойти к использованию того, что имеется в наличии. Например, потоки выполнения могут вызывать проблемы, когда их слишком много или слишком мало. Устранить их порой можно простой настройкой фреймворка Akka и более эффективно используя доступные потоки выполнения. В следующем разделе вы узнаете, как настраивать пулы потоков в Akka и связывать задачи с потоками, чтобы избавиться от накладных расходов на переключение контекста выполнения.

## 16.4. Настройка диспетчера

В главе 1 мы упоминали, что диспетчер актора отвечает за его привязку к потоку выполнения, когда в почтовом ящике появляется сообщение, ожидающее обработки. До настоящего момента нам не требовалось вникать в детали работы диспетчера (хотя в главе 9 мы уже пробовали корректировать его поведение с помощью маршрутизаторов). В большинстве случаев экземпляр диспетчера обслуживает сразу несколько акторов. Есть возможность изменить настройки диспетчера по умолчанию или создать нового диспетчера с другими настройками. Мы сначала посмотрим, как выявить проблемы, связанные с потоками выполнения. Затем мы создадим нового диспетчера для группы акторов. А после этого покажем, как можно изменить размер пула потоков и как использовать динамические пулы.

### 16.4.1. Выявление проблем с пулами потоков

В главе 9 вы видели, что есть возможность изменить поведение акторов с использованием `BalancingDispatcher`. Но, кроме того, диспетчер по умолчанию имеет множество конфигурационных параметров, изменение которых влияет на производительность. Начнем с простого примера. На рис. 16.8 показаны актор-приемник и 100 акторов-обработчиков.

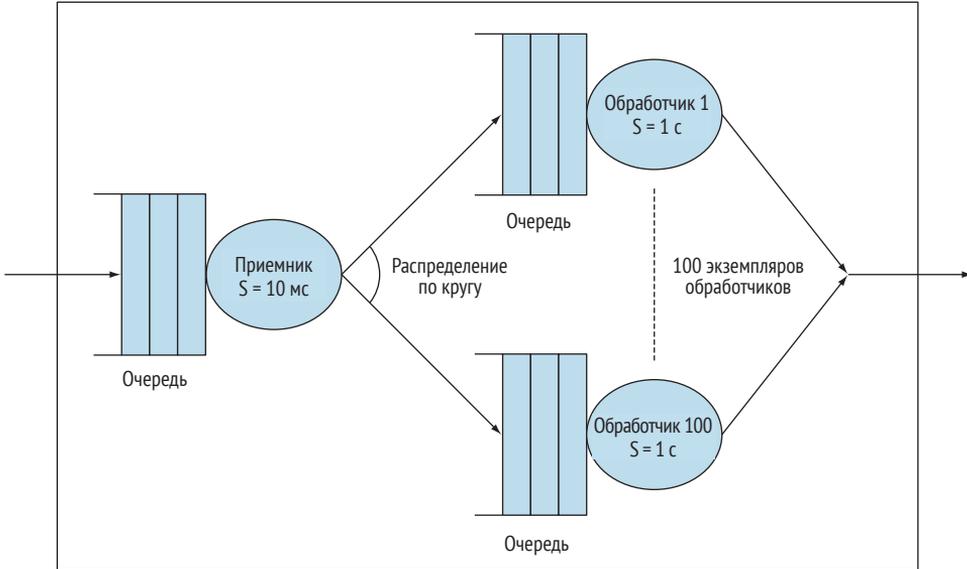


Рис. 16.8. Пример системы с приемником и сотней обработчиков

Приемник имеет время обслуживания 10 мс. Он распределяет задания между 100 обработчиками, каждый из которых имеет время обслуживания 1 с. Максимальная пропускная способность системы составляет 100/с. Реализовав этот пример в Akka, мы получили неожиданный результат. Мы подавали на вход задания с частотой 66/с, что составляет меньше 80% теоретической пропускной способности. Система, казалось бы, не должна проявлять никаких проблем, связанных с производительностью. Но, проведя мониторинг, мы обнаружили, что размер очереди приемника увеличивается с течением времени (как показано в столбце 2 в табл. 16.1).

Таблица 16.1. Результаты мониторинга параметров тестового примера

Номер периода	Приемник: максимальный размер почтового ящика	Приемник: использованное время	Обработчик 1: максимальный размер почтового ящика	Обработчик 1: использованное время
1	70	5%	1	6%
2	179	8%	1	6%
3	285	8%	1	10%
4	385	7%	1	6%

Согласно этим результатам, приемник не успевает обслужить сообщение до поступления следующего, что очень странно, потому что время об-

служивания составляет всего 10 мс, а время между сообщениями – 15 мс. Что случилось? Из обсуждения показателей производительности вы уже знаете, что когда актер является узким местом, его очередь растет в размерах, а использование процессора приближается к 100%. Но в нашем примере наблюдается только рост очереди, а использование процессора остается довольно низким, на уровне 6%. Это означает, что актер тратит много времени на ожидание чего-то еще. Проблема заключается в количестве потоков выполнения, доступных актерам для обработки сообщений. По умолчанию количество потоков в три раза превышает количество доступных процессоров, но не менее 8 и не более 64. В данном случае эксперимент проводился на компьютере с двухъядерным процессором, то есть всего было доступно 8 потоков выполнения (минимальное их количество). Пока 8 обработчиков заняты своими заданиями, приемнику приходилось ждать, пока какой-то из них не завершится, чтобы передать очередное ожидающее сообщение для обработки.

Как повысить производительность в данном случае? За назначение потоков выполнения актерам, когда имеются ожидающие сообщения, отвечает диспетчер. Чтобы улучшить ситуацию, мы должны изменить конфигурацию диспетчера, используемого актерами.

## 16.4.2. Использование нескольких экземпляров диспетчеров

Поведение диспетчера можно изменить, изменив его настройки или тип. В Акка имеется четыре встроенных типа, как показано в табл. 16.2.

В примере с одним актером-приемником и 100 актерами-обработчиками можно было бы использовать `PinnedDispatcher` для приемника. В этом случае он не будет делить потоки выполнения с обработчиками. Когда мы сделали это, проблема решилась, и приемник перестал быть узким местом. В большинстве случаев `PinnedDispatcher` не является лучшим решением. Для начала мы решили использовать пулы, чтобы уменьшить число потоков и использовать их более эффективно. В нашем примере, если задействовать `PinnedDispatcher`, поток приемника будет простаивать 33% времени. Но сама идея избавить приемник от состязания с обработчиками за обладание потоком показалась нам заслуживающей внимания. Чтобы воплотить ее, мы решили дать обработчикам отдельный пул, задействовав новый экземпляр диспетчера. В результате у нас появились два диспетчера, каждый со своим пулом потоков.

Сначала мы определили конфигурацию диспетчера и использовали его для обслуживания обработчиков.

Таблица 16.2. Типы встроенных диспетчеров

Тип	Описание	Когда используется
Dispatcher	Диспетчер по умолчанию. Связывает свои акторы с пулом потоков. В нем можно настроить управляющий механизм, но по умолчанию используется <code>fork-join-executor</code> . Это означает, что он использует пул потоков фиксированного размера	Этот диспетчер используется в большинстве случаев. Почти во всех предыдущих примерах действовал этот диспетчер
PinnedDispatcher	Этот диспетчер связывает каждый актор со своим отдельным потоком выполнения. Это означает отсутствие потоков, которые совместно использовались бы несколькими акторами	Этот диспетчер можно использовать, когда актор активно использует процессор и имеет высокий приоритет, то есть всегда, когда актор должен что-то делать и не может ждать получения нового потока выполнения. Обычно это лучшее решение из доступных, как будет показано ниже
BalancingDispatcher	Этот диспетчер перераспределяет сообщения между акторами, балансируя нагрузку на них	Этот диспетчер использовался нами в примере балансирующего маршрутизатора, в разделе 9.1.1
CallingThreadDispatcher	Этот диспетчер использует текущий поток выполнения для обработки сообщений. Может использоваться только для тестирования	Этот диспетчер автоматически используется всякий раз, когда вы используете <code>TestActorRef</code> для создания актора в своих модульных тестах

### Листинг 16.10. Определение и использование новой конфигурации диспетчера

В файле `application.conf`:  
`worker-dispatcher {}`

← Определение нового диспетчера в `application.conf`

В коде:

```
val end = TestProbe()
val workers = system.actorOf(
  Props( new ProcessRequest(1 second, end.ref) with MonitorActor)
    .withDispatcher("worker-dispatcher")
    .withRouter(RoundRobinRouter(nrOfInstances = nrWorkers))
  , "Workers")
```

← Использование `worker-dispatcher` для обслуживания обработчиков

Когда мы провели такое же тестирование еще раз, мы получили результаты, показанные в табл. 16.3.

**Таблица 16.3.** Результаты мониторинга параметров тестового примера с отдельным пулом потоков для обработчиков

Номер периода	Приемник: максимальный размер почтового ящика	Приемник: использованное время	Обработчик 1: максимальный размер почтового ящика	Обработчик 1: использованное время
1	2	15%	1	6%
2	1	66%	2	0%
3	1	66%	5	33%
4	1	66%	7	0%

Как видите, теперь приемник действует в полном соответствии с нашими ожиданиями и успевает обслуживать поступающие сообщения, о чем говорит размер очереди 1. Это значит, что предыдущее сообщение было удалено из очереди до прибытия следующего. А заглянув в колонку «Приемник: использованное время», можно увидеть число 66%, точно соответствующее ожиданиям: каждую секунду мы обрабатываем 66 сообщений, на каждое из которых тратится 10 мс.

Но теперь, как показывает столбец 5, уже обработчики не успевают за поступающими сообщениями. Фактически мы видим, что есть периоды, когда обработчик не обрабатывает *никаких* сообщений в измеряемый период (использованное время составляет 0%). Закрепив за обработчиками отдельный пул потоков выполнения, мы просто переместили проблему с приемника на обработчиков. Такое часто происходит при настройке производительности системы. Настройка – это обычно обмен одного на другое. Давая больше ресурсов одной задаче, мы отнимаем их у других. Поэтому мы должны передавать ресурсы важным задачам, отбирая их у второстепенных, которые не так важны и могут подождать. Но значит ли это, что мы не можем улучшить ситуацию? Неужели придется мириться с таким результатом?

### 16.4.3. Изменение размера пула потоков статически

Как мы видели, обработчики в нашем примере не успевают обрабатывать поступающие сообщения из-за нехватки потоков выполнения. Так почему бы нам не увеличить их количество? Мы можем сделать это, но результат в значительной мере зависит от того, насколько интенсивно обработчики используют процессор.

Увеличение числа потоков выполнения окажет отрицательное влияние на общую производительность, если обработка является в основном вы-

числительной задачей, потому что в каждый конкретный момент одно ядро процессора может выполнять только один поток. Когда требуется обслуживать большое число потоков выполнения, возникает большое количество переключений контекста выполнения. Такое переключение также занимает процессорное время, уменьшая время, которое потоки могут потратить на выполнение полезной работы. Когда отношение количества потоков выполнения к количеству ядер процессора становится слишком велико, производительность будет только уменьшаться. На рис. 16.9 показана связь производительности с количеством потоков выполнения для заданного количества ядер в процессоре.

В первой части (до первой вертикальной пунктирной линии) график растет почти линейно, до момента, когда число потоков не сравняется с числом доступных ядер. Если продолжить увеличивать число потоков, производительность продолжит увеличиваться, но все меньше и меньше, пока не будет достигнуто некоторое оптимальное значение. После этого дальнейшее увеличение числа потоков влечет снижение производительности. Этот график предполагает, что во всех потоках выполнения выполняются преимущественно вычислительные задачи. То есть всегда есть некоторое оптимальное число потоков. Но как определить его? Обычно в этом может помочь величина использования процессора. Когда процессор используется на 80% или больше, увеличение числа потоков едва ли поможет увеличить производительность.

Но когда загрузка процессора ниже, вы можете увеличить число потоков. В этом случае обработка сообщений в основном состоит из периодов ожидания. Первое, что нужно сделать, – проверить, можно ли избежать ожидания. В этом примере ситуация выглядит так, будто актер замораживается и не использует неблокирующих вызовов. Исправить ее можно было бы, например, с использованием шаблона `ask`. Если проблему ожидания решить нельзя, можно попробовать увеличить количество потоков выполнения. В нашем примере мы не используем всю мощность процессора, поэтому перейдем к следующему примеру конфигурации, соответствующему ситуации, когда увеличение потоков дает прирост производительности.



**Рис. 16.9.** Связь между производительностью и количеством потоков выполнения

Число используемых потоков выполнения можно настроить с помощью трех параметров в конфигурации диспетчера:

```
worker-dispatcher {
  fork-join-executor {
    parallelism-min = 8
    parallelism-factor = 3.0
    parallelism-max = 64
  }
}
```

Число потоков – это число доступных процессоров, умноженное на `parallelism-factor`, но не меньше `parallelism-min` и не больше `parallelism-max`. Например, в системе с восьмиядерным процессором вы получите 24 потока ( $8 \times 3$ ). Но в системе с двухъядерным процессором вы получите 8 потоков, хотя их число должно бы быть равно 6 ( $2 \times 3$ ), потому что 8 – это минимальное число.

Мы хотим использовать 100 потоков, независимо от числа доступных ядер; поэтому установим минимальное и максимальное значения равными 100:

```
worker-dispatcher {
  fork-join-executor {
    parallelism-min = 100
    parallelism-max = 100
  }
}
```

После этого, запустив пример, мы смогли обработать все сообщения вовремя. В табл. 16.4 показано, что использование времени в обработчиках тоже достигло 66%, а размер очереди уменьшился до 1.

**Таблица 16.4.** Результаты мониторинга параметров тестового примера со 100 потоками для обработчиков

Номер периода	Приемник: максимальный размер почтового ящика	Приемник: использованное время	Обработчик 1: максимальный размер почтового ящика	Обработчик 1: использованное время
1	2	36%	2	34%
2	1	66%	1	66%
3	1	66%	1	66%
4	1	66%	1	66%
5	1	66%	1	66%

В данном случае нам удалось улучшить производительность системы, увеличив число потоков. В этом примере мы использовали нового диспетчера только для обработчиков. Это лучше, чем менять диспетчера по умолчанию и увеличивать число потоков, потому что это лишь небольшая часть системы, и когда мы увеличиваем число потоков, возможно, что одновременно запустятся 100 других акторов, и это приведет к существенному падению производительности, потому что эти акторы решают преимущественно вычислительные задачи, и отношение между количеством активных потоков и ядер процессора выйдет из равновесия. При использовании отдельного диспетчера только большое число обработчиков может выполняться одновременно, а другие акторы будут использовать потоки, доступные по умолчанию.

В этом разделе мы увидели, как можно увеличить число потоков статически, но иногда бывает желательно менять размер пула динамически, например если нагрузка на обработчик существенно возрастает во время выполнения. В Akka это тоже возможно, но для этого придется изменить управляющий механизм, используемый диспетчером.

#### 16.4.4. Изменение размера пула потоков динамически

В предыдущем разделе у нас имелось фиксированное количество акторов-обработчиков. Мы могли увеличить число потоков выполнения, потому что точно знали число обработчиков. Но когда количество обработчиков зависит от нагрузки на систему, нельзя заранее сказать, сколько потоков потребуется. Например, представьте, что у нас есть веб-сервер и для обработки каждого запроса пользователя создается новый актер. Количество обработчиков зависит от количества пользователей, одновременно работающих с веб-сервером. Мы могли бы запустить фиксированное число потоков, подходящее большую часть времени. Но в этом случае возникает важный вопрос: какой размер пула выбрать? Когда потоков слишком мало, производительность начинает падать, потому что запросы стоят в очереди, ожидая, пока освободится поток выполнения, как в первом примере в предыдущем разделе, результаты мониторинга которого представлены в табл. 16.1. Но если потоков слишком много, они впустую расходуют ресурсы системы. И снова приходится чем-то жертвовать – ресурсами или производительностью. Если бы пул мог стабильно поддерживать небольшое количество потоков, а в моменты увеличения нагрузки – существенно наращивать его, это помогло бы увеличивать производительность в нужные моменты без напрасной траты ресурсов в периоды низкой нагрузки. Динамический пул потоков увеличивается в размерах, когда количество обработчиков возрастает, и уменьшается, когда потоки простаивают слишком долго. Он останавливает неиспользуемые потоки, которые иначе вхолостую расходовали бы ресурсы системы.

Чтобы получить в свое распоряжение динамический пул, мы должны изменить механизм управления пулом, используемый диспетчером. Для этого нужно изменить параметр `executor` в конфигурации диспетчера. В этом параметре можно указать три значения, которые перечислены в табл. 16.5.

**Таблица 16.5.** Настройка механизма управления пулом

Механизм управления	Описание	Когда используется
<code>fork-join-executor</code>	Механизм управления по умолчанию	Этот механизм лучше справляется с высокой нагрузкой, чем <code>thread-pool-executor</code>
<code>thread-pool-executor</code>	Другой стандартный механизм управления пулом	Этот механизм используется, когда необходимо получить динамический пул потоков, потому что динамическое поведение не поддерживается механизмом <code>fork-join-executor</code>
Полное квалифицированное имя класса	Есть возможность создать свой <code>ExecutorServiceConfigurator</code> , возвращающий фабрику для создания Java-класса <code>ExecutorService</code> , который будет использоваться как механизм управления	Когда возможностей стандартных механизмов недостаточно, можно реализовать свой механизм управления

Если вам нужен динамический пул потоков, используйте `thread-pool-executor`, не забудьте при этом настроить его. Конфигурация по умолчанию показана в листинге 16.11.

**Листинг 16.11.** Настройка диспетчера на использование механизма управления `thread-pool-executor`

```
my-dispatcher {
  type = "Dispatcher"
  executor = "thread-pool-executor"

  thread-pool-executor {
    core-pool-size-min = 8
    core-pool-size-factor = 3.0
    core-pool-size-max = 64

    max-pool-size-min = 8
    max-pool-size-factor = 3.0
    max-pool-size-max = 64
  }
}
```

← Использовать механизм `thread-pool-executor`

Минимальный размер пула.  
Имеет тот же смысл, что и параметр `fork-join-parallelism`

Максимальный размер пула

```

task-queue-size = -1

# Определяет тип очереди задач,
# может быть "array" (массив) или
# "linked" (связанный список, по умолчанию)
task-queue-type = "linked"

# Максимальное время простоя потоков
keep-alive-time = 60s

# Позволить базовым потокам превышать тайм-аут
allow-core-timeout = on
}
}

```

← Размер очереди ожидающих запросов, при превышении которого пул будет увеличен. Число -1 означает «не ограничено», то есть размер пула никогда не будет увеличиваться

← Время простоя, после которого поток будет остановлен

Минимальный и максимальный размеры пула вычисляются, как было показано в разделе 16.4.1 на примере `fork-join-executor`. Когда возникает необходимость в динамическом изменении размера пула, вы должны определить параметр `task-queue-size`. Он определяет, как быстро будет увеличиваться размер пула с увеличением числа запросов. По умолчанию он имеет значение `-1`, указывающее, что размер пула никогда не будет увеличиваться. Последний параметр, требующий настройки, – это `keep-alive-time`. Он определяет время простоя потока, по истечении которого поток будет остановлен.

В нашем примере мы установили размер `core-pool-size` близким или чуть меньше обычного количества потоков, а в параметре `max-pool-size` задали размер, при котором система все еще сможет нормально функционировать и близкий к максимальному числу одновременно обрабатываемых запросов.

В этом разделе вы узнали, как можно влиять на поведение пула потоков, используемого диспетчером, который выделяет потоки для акторов. Но есть еще один механизм, позволяющий управлять освобождением потоков и возвратом их в пул. Захватывая поток, когда у актора имеются еще сообщения для обработки, вы устраняете необходимость ждать появления нового потока и накладные расходы на связывание актора с потоком. В высоконагруженных приложениях это может способствовать увеличению общей производительности системы.

## 16.5. Изменение поведения механизма освобождения потоков

В предыдущих разделах вы узнали, как увеличить количество потоков и что существует некоторое оптимальное число, тесно связанное с количеством ядер процессора. Когда потоков слишком много, увеличение на-

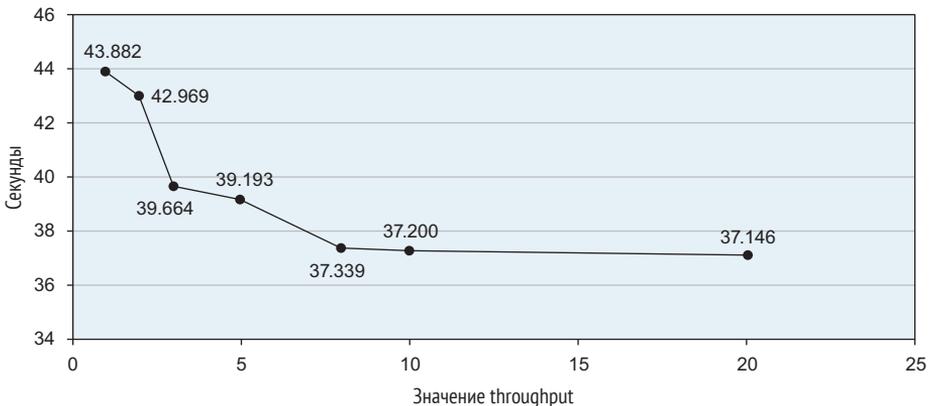
кладных расходов на переключение контекста выполнения влечет ухудшение производительности. Аналогичная проблема может возникнуть, когда имеются разные акторы с большим количеством сообщений для обработки. Чтобы обработать каждое сообщение, актор должен получить поток. Когда имеется много ожидающих сообщений для разных акторов, потоки должны переключаться между акторами. Такое переключение также отрицательно сказывается на производительности.

В Akka имеется механизм, управляющий переключением потоков между акторами. Хитрость заключается в том, чтобы запретить переключение потока, если в почтовом ящике актора остаются сообщения, ожидающие обработки. Диспетчер имеет параметр настройки, `throughput`, в котором можно установить максимальное количество сообщений, которые актор может обработать, прежде чем вернет поток обратно в пул:

```
my-dispatcher {
  fork-join-executor {
    parallelism-min = 4
    parallelism-max = 4
  }
  throughput = 5
}
```

По умолчанию он имеет значение 5. Благодаря этому количество переключений сокращается, а общая производительность увеличивается.

Чтобы оценить влияние параметра `throughput`, мы написали пример с диспетчером, имеющим четыре потока, и 40 обработчиками, со временем обслуживания, близким к нулю. В самом начале мы послали каждому обработчику по 40 000 сообщений и измерили время, затраченное на обработку всех сообщений. Затем повторили эксперимент с разными значениями параметра `throughput`. Результаты показаны на рис. 16.10.



**Рис. 16.10.** Зависимость производительности от значения параметра `throughput`

Как видите, увеличение параметра `throughput` улучшает производительность, потому что сообщения обрабатываются быстрее<sup>1</sup>.

В этих экспериментах производительность неуклонно росла с увеличением параметра, так почему по умолчанию выбрано число 5, а не 20? Как оказывается, увеличение `throughput` оказывает также отрицательное влияние. Представьте, что имеется актер с 20 сообщениями в почтовом ящике, но он имеет очень большое время обслуживания, например 2 секунды. Если установить параметр `throughput` равным 20, актер будет удерживать поток в течение 40 секунд и только потом освободит его. То есть другим актерам придется довольно долго ждать, прежде чем они смогут обработать свои сообщения. Получается, что с ростом времени обслуживания выгоды от изменения параметра `throughput` уменьшается, так как время на переключение контекста несоизмеримо меньше времени обслуживания. В результате актеры с большим временем обслуживания могут удерживать потоки долгое время. Для борьбы с этой проблемой есть еще один параметр, `throughput-deadline-time`, который определяет, как долго актер может удерживать поток даже при наличии сообщений в очереди и когда максимальное значение `throughput` еще не достигнуто:

```
my-dispatcher {
  throughput = 20
  throughput-deadline-time = 200ms
}
```

По умолчанию этот параметр имеет значение `0ms`, то есть предельное время удержания потока не ограничивается. Этот параметр можно использовать, когда имеется смесь актеров с разным временем обслуживания. Актер с малым временем обслуживания успеет обработать максимальное количество сообщений, а актер с большим временем обслуживания сможет обработать только одно или несколько сообщений, пока не достигнет предела в 200 мс.

### 16.5.1. Ограничения механизма освобождения потоков

Почему мы не можем использовать эти настройки по умолчанию? На то есть две причины. Первая – справедливость. Поскольку поток не освобождается актером после обработки первого же сообщения, другие актеры вынуждены ждать своей очереди. Для системы в целом это может быть выгодно, но для отдельных сообщений такое решение является скорее несправедливым. Для систем пакетной обработки может быть не важно, когда обрабатывается сообщение, но когда вы ждете обработки сообщения, которое, например, используется при создании веб-страницы, вам точно не понравится ждать дольше, чем ваш сосед. В таких случаях предпочти-

<sup>1</sup> В блоге Akka «Let it crash» имеется замечательная статья, где рассказывается, как удалось обработать 50 миллионов сообщений в секунду за счет изменения параметра `throughput`: <http://letitcrash.com/post/20397701710/50-million-messages-per-second-on-a-single-machine>.

тельное уменьшать значение throughput, даже если это влечет общее снижение производительности.

Другая проблема, проявляющаяся при большом значении параметра throughput, заключается в том, что процесс балансировки нагрузки между несколькими потоками может отрицательно влиять на производительность. Рассмотрим еще один пример, изображенный на рис. 16.11. Здесь имеется три актора и только два потока. Время обслуживания акторов составляет 1 секунду. Мы послали акторам 99 сообщений (по 33 каждому).

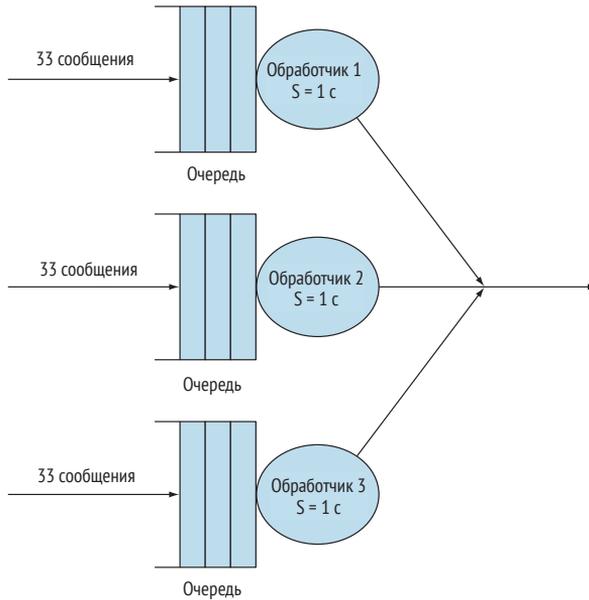


Рис. 16.11. Пример: три актора с двумя потоками

Эта система должна бы обработать все сообщения примерно за 50 секунд ( $99 \times 1$  секунда / 2 потока). Но, изменив throughput, мы получили неожиданные результаты, изображенные на рис. 16.12.

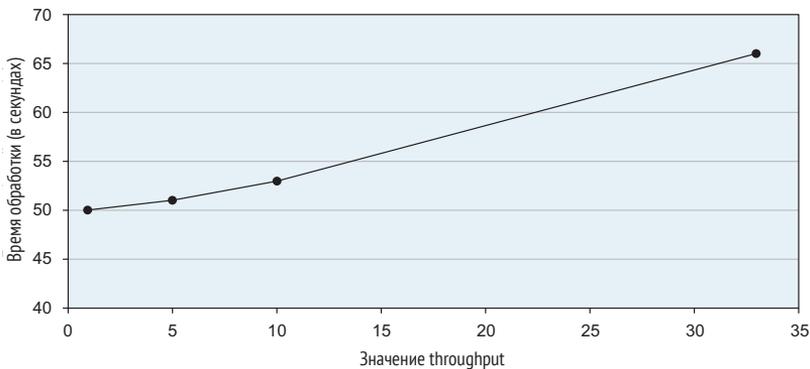
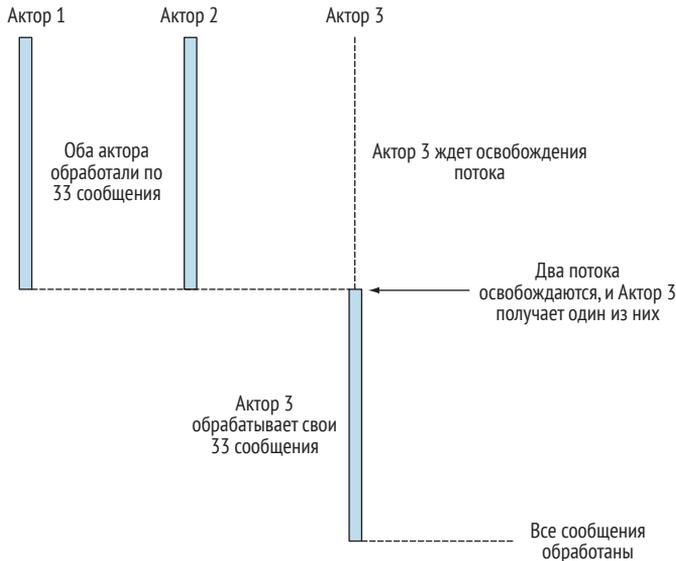


Рис. 16.12. Влияние параметра throughput

Чтобы понять, почему это произошло, рассмотрим время, затрачиваемое актерами на обработку. Возьмем самое большое значение: 33. Когда мы запустили тест, мы увидели (см. рис. 16.13), что первые два актора обработали все свои сообщения. Так как параметр имел значение 33, они смогли полностью опустошить свои почтовые ящики. После этого осталось только третьему актору обработать его сообщения. Это означает, что во второй половине один поток простаивает и вызывает удвоение времени обработки.



**Рис. 16.13.** Три актора обрабатывают сообщения в двух потоках

Изменение поведения механизма освобождения потоков может помочь улучшить производительность, но как поступить правильно – увеличить или уменьшить параметр `throughput`, – полностью зависит от частоты поступления и особенностей функционирования системы. Выбор неправильного значения может ухудшить производительность.

В предыдущих разделах вы видели, как улучшить производительность за счет увеличения числа потоков, но иногда этого недостаточно и требуется вмешательство в поведение диспетчеров.

## 16.6. В заключение

Теперь вы знаете, что механизм управления потоками выполнения может оказывать существенное влияние на производительность системы. Когда потоков слишком мало, акторы ждут завершения друг друга. Когда потоков слишком много, процессор тратит слишком много на бессмысленное

переключение потоков, многие из которых ничего не делают. В этой главе вы узнали, как:

- выявлять акторы, ожидающие освобождения потоков выполнения;
- создать несколько пулов потоков;
- изменить количество потоков (статически и динамически);
- настроить поведение механизма освобождения потоков акторами.

В предыдущих главах мы рассмотрели, как можно оптимизировать взаимодействия между узлами, чтобы повысить общую производительность распределенной системы. В этой главе мы показали вам, что многие из этих приемов могут применяться к локальным приложениям для достижения той же цели.

# Глава 17

## Заглядывая вперед

В этой главе:

- статически типизированные сообщения с использованием модуля akka-typed;
- распределение состояния в памяти с использованием Akka Distributed Data.

В предыдущих главах мы упомянули несколько интересных особенностей, которые должны появиться в Akka. Akka – это быстро развивающийся проект; на момент написания этих строк велась разработка нескольких интересных и важных особенностей, которые достойны того, чтобы следить за моментом их появления.

В этой главе мы обсудим две важные особенности, которые должны появиться в самое ближайшее время. В каждом разделе мы дадим краткий обзор новых модулей, их возможностей и как они могут изменить приемы использования Akka.

Модель акторов, описанная в этой книге, использует нетипизированные сообщения. Язык Scala имеет богатую систему типов, и он привлекает многих разработчиков прежде всего безопасной работой с типами, поэтому некоторые активно осуждают акторов в их нынешнем виде, особенно их применение в качестве компонентов приложений. Мы рассмотрим модуль akka-typed, который позволяет писать акторы, обрабатывающие типизированные сообщения.

Другой модуль, который мы рассмотрим, – Akka Distributed Data. Этот модуль позволяет распределить состояние в памяти между узлами кластера Akka с использованием типов данных, поддерживающих бесконфликтную репликацию (Conflict-free Replicated Data Type, CRDT). Мы кратко обсудим данный модуль и узнаем, какие новые возможности он несет.

## 17.1. Модуль akka-typed

Модуль akka-typed предоставляет типизированный интерфейс Actor API. Иногда бывает трудно понять, почему простое изменение в нетипизированном коде актора вызывает сбой во время работы приложения. В листинге 17.1 показан пример, который прекрасно компилируется, но не работает. В нем приводится реализация актора Basket из главы 14, а также некоторый код, пытающийся получить элементы, находящиеся в корзине.

**Листинг 17.1.** Пример актора Basket

```
object Basket {
  // .. другие сообщения ..
  case class GetItems(shopperId: Long)
  // .. другие сообщения ..
}
```

```
class Basket extends PersistentActor {
  def receiveCommand = {
    // .. другие команды ..
    case GetItems(shopperId) =>
      // .. другие команды ..
  }
}
```

```
//.. где-то в другом месте, отправка запроса актору Basket ..
val futureResult = basketActor.ask(GetItems).mapTo(Items)
```

Возвращает объект Future, потерпевший неудачу; причина – AskTimeoutException

Код, запрашивающий элементы корзины у актора Basket, и сам актор показаны вместе по простой причине: так проще понять, почему этот запрос терпит неудачу. Для тех, кто не смог заметить проблему: мы посылаем в Basket запрос GetItems вместо GetItems(shopperId). Код компилируется, потому что ask принимает параметр типа Any, а GetItems наследует Any. Опробование примера в REPL делает причину очевидной.

**Листинг 17.2.** Проверка типа в REPL

```
chapter-looking-ahead > console
[info] Starting scala interpreter...
[info]
Welcome to Scala version 2.11.8 ... (вывод обрезан)
Type in expressions to have them evaluated.
Type :help for more information.

scala> GetItems
```

```
res0: aia.next.Basket.GetItems.type = GetItems
```

← GetItems – это тип case-класса  
GetItems, а не его экземпляр

```
scala> GetItems(1L)
```

```
res1: aia.next.Basket.GetItems = GetItems(1)
```

Как видите, мы по ошибке попытались послать актору `Basket` тип `GetItems` case-класса. Это одна из тех небольших ошибок, которые легко допускаются, но трудно обнаруживаются. Такого рода проблемы часто могут возникать в процессе сопровождения приложения: кто-то может добавить или удалить поле в сообщении и забудет изменить метод `receive` актора. Иногда подобные ошибки возникают из-за банальных опечаток. Компилятор должен был бы помочь нам предотвратить их, но он не может сделать этого из-за особенностей работы `ActorRef`; вы можете послать актору буквально любое сообщение.

Модуль `akka-typed` предоставляет предметно-ориентированный язык для создания акторов, сильно отличающихся от тех, что мы видели до сих пор. Сообщения проверяются на этапе компиляции, что помогает предотвратить проблему, которую вы только что наблюдали. В листинге 17.3 показано, как могло бы выглядеть получение элементов из актора `TypedBasket` в модульном тесте.

### Листинг 17.3. Получение элементов с помощью `akka-typed`

```
"return the items in a typesafe way" in {
  import akka.typed._
  import akka.typed.ScalaDSL._
  import akka.typed.AskPattern._
  import scala.concurrent.Future
  import scala.concurrent.duration._
  import scala.concurrent.Await

  implicit val timeout = akka.util.Timeout(1 second)

  val macbookPro =
    TypedBasket.Item("Apple Macbook Pro", 1, BigDecimal(2499.99))
  val displays =
    TypedBasket.Item("4K Display", 3, BigDecimal(2499.99))

  val sys: ActorSystem[TypedBasket.Command] =
    ActorSystem("typed-basket", Props(TypedBasket.basketBehavior))

  sys ! TypedBasket.Add(macbookPro, shopperId)
  sys ! TypedBasket.Add(displays, shopperId)

  val items: Future[TypedBasket.Items] =
```

```

    sys ? (TypedBasket.GetItems(shopperId, _))

    val res = Await.result(items, 10 seconds)
    res should equal(TypedBasket.Items(Vector(macbookPro, displays)))
    //sys ? Basket.GetItems
    sys.terminate()
  }

```

← Теперь этот код не компилируется

Самое большое изменение здесь – теперь ActorSystem и ActorRef получают аргумент типа, описывающий сообщения, которые они могут принимать. Это позволяет компилятору проверить тип сообщения во время компиляции.

В листинге 17.4 показана часть реализации актора TypedBasket, отвечающая за извлечение элементов. (В главе 14 актор Basket в действительности был реализован как хранимый актор типа PersistentActor, но мы пока проигнорируем это обстоятельство; предполагается, что в будущем будет возможно создавать хранимые акторы с помощью модуля akka-typed.)

#### Листинг 17.4. TypedBasket

```

package aia.next

import akka.typed._
import akka.typed.ScalaDSL._
import akka.typed.AskPattern._
import scala.concurrent.Future
import scala.concurrent.duration._
import scala.concurrent.Await

object TypedBasket {
  sealed trait Command {
    def shopperId: Long
  }

  final case class GetItems(shopperId: Long,
                           replyTo: ActorRef[Items]) extends Command
  final case class Add(item: Item, shopperId: Long) extends Command

  // упрощенная версия Items и Item
  case class Items(list: Vector[Item]= Vector.empty[Item])
  case class Item(productId: String, number: Int, unitPrice: BigDecimal)

  val basketBehavior =
    ContextAware[Command] { ctx =>
      var items = Items()

      Static {

```

```

    case GetItems(productId, replyTo) =>
      replyTo ! items
    case Add(item, productId) =>
      items = Items(items.list :+ item)
  //case GetItems =>
}
}
}

```

← Теперь этот код не компилируется

Во время исследований разработчики Akka обнаружили, что больше всего проблем возникает в методе `sender()`. Это одна из причин, мешающих простому преобразованию текущего API акторов в типизированный интерфейс: любое сообщение может быть отправлено любым отправителем, что делает невозможным типизировать его на принимающей стороне.

Модуль `akka-typed` не имеет метода `sender()`, а это означает, что если вы захотите послать сообщение обратно вызывающему актору, вы должны будете послать в сообщении ссылку на актор. Добавление ссылки на отправителя в сообщения дает преимущества также обычным, нетипизированным акторам: вы всегда будете знать, кто послал сообщение, и вам не придется хранить это состояние где-то еще. Модуль `akka-typed` избавляет от необходимости определять метод `sender()` для сохранения этого состояния, позволяя полностью обойти проблему стороной.

Определение актора сильно отличается от привычного; теперь актор определяется с точки зрения типизированного поведения. Каждое сообщение передается неизменному поведению. Поведение актора может меняться с течением времени, путем переключения между поведением, или оставаться прежним.

В данном случае поведение определяется как `Static`, то есть актор `TypedBasket` никогда не меняет своего поведения. Это поведение завернуто в `ContextAware`, принимающий функцию `ActorContext[T] => Behavior[T]`, определяющую поведение актора. Это позволяет получить контекст, а также определить состояние актора.

В `akka-typed` имеется также множество других изменений, по сравнению с текущим модулем `actor`. Методы `preStart`, `preRestart` и другие, например, заменены специальными сигнальными сообщениями.

Интерфейс модуля `akka-typed` выглядит многообещающе, но, скорее всего, он еще не раз изменится, потому что продолжает активно разрабатываться, и поэтому прямо сейчас его не следует использовать для промышленных приложений. Преимущества контроля типов огромны, поэтому мы ожидаем, что этот модуль станет важным нововведением в будущих версиях Akka.

## 17.2. Akka Distributed Data

Теперь перейдем к модулю Akka Distributed Data, который реализует структуры данных в памяти для репликации в кластере Akka. Это так называемые *типы с бесконфликтной репликацией* (Conflict-free Replicated Data Type, CRDT), которые рано или поздно оказываются в непротиворечивом состоянии. В каком бы порядке не выполнялись операции с типом данных, результат всегда будет правильным.

Типы CRDT имеют функцию `merge`, которая может взять несколько записей данных, хранящихся на нескольких узлах, и объединить их в одно непротиворечивое представление без всякой координации между узлами. Круг типов структур данных, которые можно использовать, ограничен: они должны быть типами CRDT. Модуль Akka Distributed Data предоставляет несколько predefined структур данных, но при этом есть возможность создавать свои структуры, реализующие функцию `merge`, действующую в соответствии с правилами CRDT (они должны быть ассоциативными, коммутативными и идемпотентными).

В Akka Distributed Data имеется также актор `Replicator`, выполняющий репликацию структуры данных в кластере Akka. Структуры данных хранятся с пользовательским ключом, и предлагается возможность подписаться по ключу, чтобы получать обновления в структуре данных.

Пример корзины покупателя, рассматривавшийся в этой книге, является отличным кандидатом на использование Akka Distributed Data. Элементы можно смоделировать как множество CRDT с именем `ORset`. Каждая такая корзина в конечном итоге окажется в непротиворечивом состоянии на каждом узле кластера. Существует структура данных CRDT для совместного редактирования документов – это еще один пример, где можно было бы применить Akka Distributed Data. Объединив Akka Distributed Data с механизмом хранения акторов Akka, можно было бы реализовать восстановление состояния в памяти из журнала.

## 17.3. В заключение

В этой главе мы затронули два важных нововведения, которые способны оказать большое влияние на будущие версии Akka. Как вы понимаете, это не полный список новых возможностей, и время покажет, насколько успешной будет каждая из них.

Модуль `akka-typed` обеспечивает более высокую безопасность типов, то есть с его помощью можно будет выявлять больше ошибок на этапе компиляции, что, в свою очередь, упростит создание и (что особенно важно) сопровождение приложений на основе акторов, которые, как нам кажется, станут еще удобнее в использовании.

Безусловно, это очень важный первый шаг, но предстоит провести еще множество исследований в области применения современной теории типов для повышения безопасности протоколов взаимодействий, не говоря уже о том, как это будет реализовано на языке Scala в модуле akka-typed.

Akka Distributed Data потребует от вас подумать о вашей прикладной задаче с позиции CRDT; возможность выразить задачу с применением таких типов данных способна дать существенные преимущества.

Порой удивительно, как самые разные задачи могут получить дополнительные выгоды от применения подхода на основе обмена сообщениями и акторов Akka, и почти невозможно предсказать, какие задачи еще попадут в это число.

# Предметный указатель

## Символы

-Xmх, параметр 354  
|, символ 360

## Нумерованный

80/20, правило 473

## А

агенты 309  
    общее состояние 309  
    ожидание изменения 312  
агрегатор 221  
адаптеры событий 457  
ад обратных вызовов 49  
актор  
    сбор данных в почтовом ящике 478  
акторы  
    асинхронное выполнение 22  
    в приложении продажи билетов  
        BoxOffice, актер 63, 65  
        ReastApi, актер 68  
        TicketSeller, актер 64  
    структура приложения 59  
жизненный цикл 107  
    restart, событие 109  
    start, событие 107  
    stop, событие 108  
    мониторинг 113  
    обзор 111  
и почтовые ящики 47  
и сеть 49  
и традиционный подход 32  
краткий обзор 22  
модель 22  
модель асинхронного выполнения 40  
независимость по трем осям 45  
немые 79  
объединение с объектами Future 152  
операции с 41  
    отправка 42

    переход 43  
    создание 43  
    управление 43  
оценка производительности 477  
посылающие сообщения другим актерам 79  
почтовые ящики  
    настройка 483  
производительность 475  
сбор и обработка данных 485  
создание своих почтовых ящиков 479  
тестирование  
    SendingActor 79, 84  
    SideEffectingActor 79, 89  
    SilentActor 79  
    обзор 76  
    типизированный API 505  
    удаленное развертывание 175  
    универсальная модель программирования 39  
асинхронное тестирование 77  
асинхронный и неблокирующий ввод/вывод 350

## Б

балансировка нагрузки  
    маршрутизатор с группой  
        динамическое изменение группы 253  
    создание групп 250  
наблюдение 249  
обзор 238  
блокирующий ввод/вывод 23

## В

ветвление потоков 378  
внутренние буферы 357

## Г

горизонтальное масштабирование 19, 156  
    тестирование с multi-JVM 180  
    удаленное развертывание 175  
    удаленные взаимодействия 159  
    удаленный поиск 167  
граф обработки 348  
графы

ветвление потоков 378  
 внутренние буферы 357  
 обособление частей, действующих с разной скоростью 389  
 объединение материализованных значений 359  
 предопределенные и нестандартные 384  
 предотвращение переполнения памяти 356  
 слияние потоков 381  
 язык описания 378

**Д**

двунаправленные потоки 366  
 двусторонние взаимодействия 92  
 действие при входе 293  
 действие при выходе 293  
 десериализация 157  
 динамическое членство 157  
 директивы 341  
 диспетчеры  
 выявление проблем с пулами потоков 489  
 изменение поведения механизма освобождения потоков 498  
 изменение размера пула потоков динамически 496  
 изменение размера пула потоков статически 493  
 настройка 489  
 доступ к памяти 159

**Ж**

журналирование 199  
 применение 201  
 создание адаптера 200

**З**

зависимости 161  
 зависимости в файле сборки 161  
 задержки 158, 474, 476  
 замыкания на значениях 132  
 запуск модели 351

**И**

изменение поведения механизма освобождения потоков 498  
 изменение размера пула потоков динамически 496

изменение размера пула потоков статически 493  
 интеграция систем  
 конечные точки 315  
 канонические данные, модель 319  
 нормализатор, шаблон 317  
 реализация с Apache Camel 322  
 реализация HTTP-интерфейса 335  
 интеграция со службами 385  
 и пусть падает, философия 103  
 исключения  
 в Future 138  
 нефатальные 139  
 обработка 98  
 фатальные 139

**К**

канал недоставленных сообщений 281  
 каналы обмена сообщениями  
 DeadLetter 281  
 EventBus, интерфейс 275, 276  
 EventStream 272, 274  
 гарантированная доставка 283  
 издатель-подписчик 270  
 обзор 268  
 точка-точка 269  
 канонические данные, модель 319  
 классификатор, EventBus, интерфейс 276  
 кластеры  
 на основе хранимых акторов 457  
 обзор 393  
 обработка заданий  
 запуск кластера 412  
 маршрутизация заданий 414  
 надежная обработка заданий 417  
 обзор 410  
 тестирование кластера 424  
 тестирование 424  
 клонирование проектов 52  
 коллекции 143  
 команды 438  
 конвейерная обработка 211  
 конвейеры и фильтры  
 обзор 211  
 с Akka 212  
 конечная точка

потребитель 323  
     смена транспортного уровня 326  
 производитель 330  
 конечные автоматы  
   FSM, трейт 295  
   завершение 308  
   обзор 292  
   определение состояний 296  
   реализация переходов 296  
   создание модели 294  
   таймеры 305  
   тестирование 303  
 конечные точки  
   Apache Camel, библиотека  
     использование CamelContext 329  
     обзор 322  
     смена транспортного уровня 326  
   REST-реализация 336  
   канонические данные, модель 319  
   нормализатор, шаблон 317  
   обзор 315  
   потребители 317  
   производители 317  
 конкуренция 18  
   в распределенном программировании 159  
 конфигурация  
   Akka 195  
   использование значений по умолчанию 192  
   обзор 189  
   подсистем 196  
   пустые свойства 193  
 красный-зеленый-рефакторинг, стиль  
   разработки 80

**М**

манифест реактивного программирования 23  
 маршрутизатор с группой 250  
   динамическое изменение группы 253  
   создание групп 250  
 маршрутизатор с пулом 243  
   динамическое изменение размера 246  
   создание 243  
 маршрутизатор, шаблон 237  
   маршрутизация на основе состояния 263  
   маршрутизация по содержимому 262

  обзор 237  
   реализация с применением акторов 262  
 маршрутизаторы  
   балансировка нагрузки 238  
   группы, определение 239  
   на основе акторов 262  
   пулы, определение 239  
 маршрутизаторы с пулом  
   удаленные маршруты 245  
 маршруты, RestApi 60  
 масштабирование  
   горизонтальное 19  
   два подхода 24  
   на основе Akka 24, 32  
   общее обсуждение 156  
   ожидания 20  
   тестирование с multi-JVM 180  
   традиционное 24, 26  
     большая проблема 29  
   удаленное развертывание 175  
   удаленный поиск 167  
 материализованные значения 353  
 механизм обнаружения аварий 408  
 многопоточное программирование 21  
 модель акторов 22  
 моментальные снимки 443

**Н**

настройка  
   Akka 195  
   использование значений по умолчанию 192  
   обзор 189  
   подсистем 196  
   пустые свойства 193  
 настройка тайм-аута 83  
 неблокирующее обратное давление 357  
 неизменяемость 22  
 нефатальные исключения 139  
 нормализатор, шаблон 317

**О**

обнаружения, механизм 158  
 обработка заданий  
   запуск кластера 412  
   маршрутизация заданий 414

- надежная обработка заданий 417
- обзор 410
- тестирование кластера 424
- обработка ошибок в потоках 364
- общее состояние
  - с использованием агентов 309
  - ожидание изменения 312
- объединение материализованных значений 359
- ограничения механизма освобождения
  - потоков 500
- ограниченное использование памяти 350
- односторонние взаимодействия
  - тестирование акторов с побочными эффектами 89
- односторонние взаимодействия
  - тестирование 79
  - тестирование акторов, посылающих сообщения 84
  - тестирование немых акторов 79
- оператор слияния 358
- опорные узлы 396
- определение модели 351
- отказоустойчивость
  - акторы
    - жизненный цикл 107
    - обзор 111
  - обзор 96
  - пример приложения 98
  - с акторами 103
  - стратегии 97
  - супервизоры
    - обзор 103
- отравление, почтового ящика 105
- охранник пространства пользователя 117
- П**
- параллельная обработка дроблением 216
  - область применения 216
  - параллельная обработка 218
  - с Akka 218
  - состояющиеся задачи 216
- передача по имени 132
- переходы 292
- плагины поддержки журналов 437
- подписчик, EventBus, интерфейс 276
- подъем конфигурации 197
- последовательный опрос 23
- постоянные соединения 371
- потоковая обработка
  - JSON 377
  - копирование файлов 351
  - основы 347
  - ошибки в потоках 364
  - передача данных через HTTP 369
- потоковая передача данных через HTTP 369
- потребители, конечные точки 317
- потребитель, конечная точка 323
  - смена транспортного уровня 326
- потребление времени 476
- почтовые ящики
  - настройка 483
  - обзор 478
  - создание своих почтовых ящиков 479
- почтовый ящик
  - отравление 105
- предопределенные и нестандартные графы 384
- предотвращение переполнения памяти 356
- приведение вверх (upcasting) 452
- примеры проектов
  - клонирование 52
  - сборка с помощью sbt 53
- принцип Парето 473
- производители, конечные точки 317
- производитель, конечная точка 330
- производительность
  - изменение поведения механизма освобождения потоков 498
  - изменение размера пула потоков динамически 496
  - изменение размера пула потоков статически 493
  - и маршрутизация 237
  - параметры 474, 475
  - сбор и обработка данных 485
  - узкие места
    - обзор 473
    - улучшение устранением узких мест 487
- пропускная способность 474, 475
- протокол 157
- протокол обмена служебной информацией 405

пул

- динамическое изменение размера 246
- создание 243
- удаленные маршруты 245

## Р

развертывание

- обзор 204

развертывание приложения

- в Heroku 72

разделение точек 394

распределенное программирование 158

реактивное программирование

- манифест 23

регулировка скорости 351

родительский контроль 104

## С

сбор и обработка данных 485

сериализация 157, 166, 451

сериализация потока 363

сериализация событий 363

сети

- терминология 156

синхронные взаимодействия 23

слияние потоков 381

событие, EventBus, интерфейс 275

события 438

согласование контента 373

- обработка заголовка Accept 375

- обработка заголовка Content-Type 373

состояние

- конечные автоматы

- FSM, трейт 295

- завершение 308

- обзор 292

- определение состояний 296

- реализация переходов 296

- создание модели 294

- таймеры 305

- тестирование 303

- общее

- с использованием агентов 309

состояющиеся задачи 216

список получателей 219

статическое членство 157

супервизоры 114

- иерархия 114

- обзор 103

- стратегии

- по умолчанию 117

- предопределенные 117

- собственные 118

## Т

тайм-аута, настройка 83

таймеры в конечных автоматах 305

тестирование

- конечных автоматов 303

- с multi-JVM 180

- хранимых акторов 441

типизированный API акторов 505

типы данных с бесконфликтной репликацией

- (Conflict-free Replicated Data Type, CRDT) 504, 509

топологии сетей 157

традиционное масштабирование 26

транспортные протоколы 157

## У

удаленное развертывание 175

удаленный поиск 167

узкие места

- обзор 473

узлы 156

универсально-уникальные идентификаторы

- (Universally Unique Identifier, UUID) 458

## Ф

файлы сборки, sbt 53

фатальные исключения 139

фильтрация событий 362

## Х

хранимые акторы

- в кластерах 457

- восстановление состояния из событий 435

- адаптеры событий 457

- моментальные снимки 444

- обновление записей на месте 432

- сериализация 451

- сохранение состояния без изменения 433
- обзор 436
- общее обсуждение 430
- расширение cluster sharding 465
- расширение cluster singleton 461
- случаи использования 431
- тестирование 441

**Ч**

- частичный отказ 159
- частота обслуживания 475
- частота поступления 475
- членство в кластере 394
  - выход 404
  - присоединение 397
  - типы узлов 396
- члены группы 157

**Ш**

- шаблоны структуризации акторов 210
  - агрегатор 221
  - конвейеры и фильтры 211
    - с Akka 212
  - маршрутизация 229
  - объединение компонентов 227
  - параллельная обработка дроблением 216
    - область применения 216
    - параллельная обработка 218
    - с Akka 218
    - состояющиеся задачи 216
  - список получателей 219

**Я**

- язык описания графов 378

**А**

- activationFutureFor, метод 325
- ActiveMQ, компонент 329
- ActorMaterializer 355, 370, 450
- ActorMaterializerSettings 358, 365
- actorOf, метод 107, 243
- ActorPublisher, трейт 385
- ActorRef 47
- actorSelection, метод 165, 179

- ActorSubscriber, трейт 385
- ActorSystem 45
- AdaptiveLoadBalancingPool 421
- AddRoutee, сообщение 253
- Akka
  - отправка и получение сообщений 37
- akka-camel, модуль 322
- akka-cluster, модуль 176
- Akka Distributed Data, модуль 509
- akka-http 369
- akka-http, модуль
  - REST-реализация конечной точки 338
- akka-http, обзор 346
- akka-kryo-serialization, модуль 452
- akka-persistence-cassandra, модуль 465
- akka-persistence, модуль 430
- akka-remote, модуль 162, 394
- akka-stream-based 385
- akka.stream.blocking-io-dispatcher 352
- akka.stream.materializer.max-input-buffer-size 358
- akka-stream, модуль 347
- akka-testkit, модуль 77
- Akka, фреймворк
  - обзор 22
- AllForOneStrategy 249
- allow-local-routees, параметр настройки 415
- alter, метод 312
- Apache Camel, библиотека 322
  - CamelContext, класс 322
  - CamelExtension, класс 323
  - ProducerTemplate, класс 322
  - использование CamelContext 329
  - конечная точка-потребитель 323
  - конечная точка-производитель 330
  - обзор 322
  - реализация конечных точек 322
  - смена транспортного уровня 326
- Apache Cassandra, база данных 465
- Apache Kafka 135
- Apache ZooKeeper 398
- application.conf, файл 162, 163, 189, 206
- application.json, файл 189
- application.properties, файл 189
- apply, метод 132
- async, метод 358

auto-down-unreachable-after, параметр  
настройки 408

## В

backoff-rate, атрибут 248  
backoff-threshold, атрибут 248  
BackOfSupervisor, актор 123  
Balance 384  
BalancingDispatcher 489, 492  
BalancingPool, маршрутизатор 241  
BasketEventSerializer 454  
BasketSnapshotSerializer 456  
BasketSnapshotSpec, класс 447  
Basket, актор 444  
BDD (Behavior-Driven Development разработка  
через поведение) 75  
become, метод 264  
BidiEventFilter, приложение 367  
BidiFlow, создание протокола 366  
BoxOffice, актор 65  
Broadcast 385  
BroadcastGraphStage 378  
BroadcastGroup, маршрутизатор 241  
BroadcastPool 415  
BroadcastPool, маршрутизатор 241  
Broadcast, сообщение 244  
BrokerRegistry 330  
build.sbt, файл 207  
ByteString 353

## С

CalculationResult, класс 441  
CallingThreadDispatcher 492  
CamelContext, класс 322  
CamelContext, компонент 329  
CamelExtension, класс 323  
cancelTimer, метод 308  
classify, метод 278  
cleanUp, метод 479  
ClusterClient 427  
ClusterRouterPool 415  
ClusterRouterPool, маршрутизатор 246  
ClusterSharding.start, метод 468  
cluster sharding, расширение 465  
ClusterSingletonManager 463

cluster singleton, расширение 461  
combine, метод 383  
compareSubscribers, метод 278  
complete, директива 342  
Concat 384  
ConfigFactory 189  
config.file, свойство 195  
config.resource, свойство 195  
config.url, свойство 195  
ConsistentHashableEnvelope 261  
ConsistentHashingGroup, маршрутизатор 242  
ConsistentHashingPool, маршрутизатор 242  
ConsistentHashing, маршрутизатор 257  
    вычисление ключа внутри сообщения 260  
    вычисление ключа отправителем 261  
    обзор 257  
    передача частично определенной функции 259  
ContentNegLogsApi, класс 374  
Content-Type, заголовок 374  
ControlThrowable 140  
CountRecoveredEvents, команда 447  
CRDT (Conflict-free Replicated Data Type), типы  
    данных с бесконфликтной  
    репликацией 504, 509  
createAndForward, метод 464  
CreateWorkerRouter, трейт 415  
currentEventsByPersistenceId, метод 451  
CurrentState, сообщение 304

## D

dataBytes 371  
DeathPactException, исключение 254  
deleteStorageLocations, метод 443  
dequeue, метод 479, 481  
Dispatcher 492  
DSL (Domain-Specific Language предметно-  
    ориентированный язык) 53  
Duration.Inf, значение 308

## E

EchoActor, актор 92  
Egress 285  
Either, тип 366  
enqueue, метод 479, 481  
entity, директива 343

EventBus, интерфейс 275, 276  
EventFilter, приложение 360, 366  
eventHandler 199  
EventMarshalling, трейт 361  
Event Sourcing (история событий), технология 430  
  адаптеры событий 457  
  для акторов 435  
  моментальные снимки 444  
  обновление записей на месте 432  
  сериализация 451  
  сохранение состояния без изменения 433  
eventStream 199  
EventStream 272, 274  
eventStream, метод 174  
Event, класс 361  
ExecutionContext 135, 310, 370  
Exiting, состояние 406  
expectMsg, метод 86, 183  
expectNoMsg, метод 88  
ExtractEntityID, функция 469

## F

FileChannel 355  
FileIO 352, 353  
FileIO.fromPath 353  
FileIO.toPath 353  
FilePublisher 355  
FileSink 356  
FileSource 355  
FileSubscriber 356  
FilteringActor, актор 86  
filterState 363  
filter, оператор 362  
firstCompletedOf, метод 145  
FlowShape, тип 380  
Flow, компонент 361  
fold, метод 150  
foreach, метод 133  
forMax, метод 307  
ForwardingActor, актор 86  
for-регенератор 148  
Framing.delimiter 362  
FromEntityUnmarshaller 342  
FSM, трейт 295  
Fusing.aggressive 358

Future, класс в Java 127  
Future, объекты 125  
  вызов веб-служб 131  
  исключения 138  
  комбинирование 143  
  обработка ошибок 138  
  объединение с акторами 152  
  определение 67, 126  
  примеры использования 126

## G

GenerateLogFile, приложение 354  
getBrokers, метод 330  
getOrder, метод 341  
GetRoutees, сообщение 253  
get, метод 311  
Google Protocol Buffers, формат 452  
Gossip, протокол 405  
GoTicks.com REST API 54  
Graph, тип 378  
groupedWithin, метод 390

## H

HashiCorp Consul 398  
Heroku  
  развертывание приложения 72  
  создание приложения 71  
HOCON, формат 190  
HTTP  
  REST-реализация конечной точки 338  
  реализация 335  
  пример 336  
httpie, утилита 56

## I

ignoreMsg, метод 88  
IllegalStateException, исключение 226  
ImplicitSender, трейт 92, 183  
initialize, метод 303  
InterruptedException 140  
inputFile 352  
isTimerActive, метод 308

## J

Java  
  класс Future 127

Joining, состояние 406  
 jsonToLogFlow, метод 376

## К

Keep.both, функция 360  
 Keep.left, функция 360  
 Keep.none, функция 360  
 Keep.right, функция 360  
 Keep, объект 359  
 killActors, метод 442

## L

LevelDB 437  
 LevelDbReadJournal 450  
 Lightbend, репозиторий 54  
 LinkageError 140  
 ListBuffer, тип 65  
 List, класс 363  
 logFileSink, метод 371  
 logFileSource, метод 371  
 logFile, метод 370  
 LoggerMessageQueueSemantics 480  
 LogParseException 365  
 LogsApi, класс 370  
 LogsApp, приложение 369  
 LogStreamProcessor.parseLineEx, метод 364  
 LookupClassification, трейт 277  
 lower-bound, атрибут 247

## М

MailboxStatistics 481  
 MailboxStatistics, класс 480  
 MailboxType, реализация 482  
 Map, класс 363  
 map, метод 134  
 map, оператор 362  
 master.conf, файл 413  
 MergeGraphStage 382  
 mergeNotOk, метод 381  
 MergePreferredGraphStage 382  
 mergeSources, метод 383  
 merge, функция 509  
 MessageQueue 482  
 messages-per-resize, атрибут 249  
 MonitorEnvelope, класс 480

MonitorQueue, класс 479  
 multi-JVM, плагин, тестирование 180  
 MultiNodeConfig 425  
 multi-node-testkit, модуль 424  
 MutatingCopyActor, актер 86

## N

NoSuchOrder, объект 337  
 NotUsed, тип 363

## O

onComplete, метод 140  
 OnComplete, сообщение 356  
 onTransition, метод 301  
 OrderService, трейт 339  
 Order, объект 332  
 OutOfMemoryException 356  
 outputFile 352  
 OverflowStrategy 386

## P

persistAsync, функция 439  
 PersistenceCleanup, трейт 443  
 PersistenceSpec, класс 442  
 PersistentActor, трейт 438  
 persistentId, значение 438  
 persist, метод 438  
 PinnedDispatcher 492  
 plugins.sbt, файл 206  
 PoisonPill, сообщение 108, 252  
 postOrders, метод 341  
 postRestart, метод 112  
 postRoute, метод 371  
 postStop, метод 108, 109, 112  
 preRestart, метод 109, 225  
 pressure-threshold, атрибут 247  
 preStart, метод 108  
 ProcessOrders, актер 342  
 processStates, метод 380  
 Procfile, файл, Heroku 72  
 ProducerTemplate, класс 322  
 Producer, трейт 331  
 Promise, объекты 135  
 Props, объект 82, 110  
 publish, метод 278

**Q**

queueSize 481

**R**

rampup-rate, атрибут 248

RandomGroup, маршрутизатор 241

RandomPool, маршрутизатор 241

Reactive Streams API 357

ReastApi, актор 68

receiveCommand, метод 438, 445

receiveRecover, метод 440, 447

receiveWhile, метод 87

receive, метод 324

recoverWith, метод 142

recover, метод 141

reference.conf, файл 194, 206

registerOnMemberUp, метод 414

ReliableProxy 285

Removed, состояние 406

RemoveRoutee, сообщение 253

REPL

удаленные взаимодействия в 161

Replicator, актор 509

RequestTimeout, трейт 63

RequiresMessageQueue 481

RestApi, актор 60

re-start, команда 354

restart, событие 109

Restart, стратегия 366

re-stop, команда 354

REST-реализация 336

Resume 365

Resume, стратегия 366

retryAfter, параметр 285

RoundRobinGroup, маршрутизатор 240

RoundRobinPool, маршрутизатор 240

routeResponse, метод 335

RunnableGraph 353

Runnable, класс 137

run, команда 354

**S**

saveSnapshot, метод 446

sbt-multi-jvm, плагин 424

sbt-native-packager, плагин 204, 206

sbt-revolver, плагин 354

sbt, инструмент сборки 52

ScalaTest, фреймворк 76

scala.xml.Elem 342

scala.xml.NodeSeq 342

ScanningClassification, трейт 278

ScatterGatherFirstCompletedGroup,  
маршрутизатор 242

ScatterGatherFirstCompletedPool,  
маршрутизатор 242

seed.conf, файл 400

sender, метод 508

sendMessageToNextTask, метод 232

send, метод 312

sequence, метод 149, 331

SequencingActor, актор 86

setTimer, метод 308

Set, класс 363

Sink, конечная точка 353

SmallestMailboxPool, маршрутизатор 241

Source, конечная точка 353

Stamina, библиотека 452

Started, состояние 107

startWith, метод 297

start, событие 107

stateTimeout, значение 306

StoppingStrategy 420, 423

stop, событие 108

Stop, стратегия 366

StreamingCopy, приложение 352

SubchannelClassification, трейт 277

SubscribeTransitionCallBack, сообщение 304

subscribe, метод 273, 408

supervisorStrategy, метод 352

sync=false, параметр 327

**T**

takeWhile, оператор 356

takeWithin, оператор 356

take, оператор 356

TDD (Test-Driven Development разработка  
через тестирование)

тестирование акторов 75

Terminated, состояние 107, 110

textline=true, параметр 327

ThreadDeath 140  
TicketSeller, актор 64  
TimerTick 205  
ToEntityMarshaller 342  
toMat, метод 359, 364  
toOrder, метод 343  
toPath, метод 356  
to, метод 356  
TrackingOrder, объект 336  
TransformingActor, актор 86  
transformOutgoingMessage, метод 332  
transformResponse, метод 333  
traverse, метод 149  
Typesafe Config Library, библиотека 189

## U

unbecome, метод 265  
UnboundedMessageQueueSemantics 480  
Unmarshaller, трейт 373, 374  
unsubscribe, метод 273  
updateState, функция 439, 445  
upper-bound, атрибут 247  
Up, состояние 406

## V

VirtualMachineError 140

## W

web-дыно, экземпляр в Heroku 74  
whenUnhandled, объявление 299  
withFixedContentType, метод 376  
worker.conf, файл 413

Книги издательства «ДМК Пресс» можно заказать в торгово-издательском холдинге «Планета Альянс» наложенным платежом, выслав открытку или письмо по почтовому адресу: **115487, г. Москва, 2-й Нагатинский пр-д, д. 6А.**

При оформлении заказа следует указать адрес (полностью), по которому должны быть высланы книги; фамилию, имя и отчество получателя. Желательно также указать свой телефон и электронный адрес.

Эти книги вы можете заказать и в интернет-магазине: **[www.aliants-kniga.ru](http://www.aliants-kniga.ru)**.

Оптовые закупки: тел. +7 (499) 782-38-89

Электронный адрес: **[books@aliants-kniga.ru](mailto:books@aliants-kniga.ru)**.

Раймонд Рестенбург, Роб Баккер, Роб Уильямс

## **Акка в действии**

Главный редактор *Мовчан Д. А.*  
[dmkpress@gmail.com](mailto:dmkpress@gmail.com)

Перевод с английского *Киселев А. Н.*  
Корректор *Синяева Г. И.*  
Верстка *Паранская Н. В.*  
Дизайн обложки *Мовчан А. Г.*

Формат 70×100<sup>1/16</sup>. Печать цифровая.  
Усл. печ. л. 42,41. Тираж 200 экз.

Веб-сайт издательства: [www.dmkpress.com](http://www.dmkpress.com)