Содержание

Предисловие 9
Об авторе 11
Благодарности 12
О технических рецензентах 14
Вступление 17

Глава 1. Введение 25
Конкурентное программирование 25
Краткий обзор традиционных подходов к организации конкурентного выполнения 26
Современные парадигмы конкуренции 27
Преимущества языка Scala 28
Начальные сведения 29
Выполнение программ на Scala 30
Основы Scala 31
Обзор новых особенностей в Scala 2.12 35
В заключение 36
Упражнения 36

Глава 2. Конкуренция в JVM и модель памяти в Java 38
Процессы и потоки 39
Создание и запуск потоков 41
Атомарное выполнение 45
Переупорядочение 49
Мониторы и синхронизация 51
Взаимоблокировки 53
Защищенные блокировки 55
Прерывание потоков и корректная остановка 59
Изменчивые переменные 60
Модель памяти в Java 62
Неизменяемые объекты и финальные поля 64
В заключение 65
Упражнения 66

Глава 3. Традиционные строительные блоки конкурентных программ 69
Объекты Executor и ExecutionContext 70
Атомарные примитивы 73
Атомарные переменные 73
Неблокирующее программирование 76
Явная реализация блокировок 78
Проблема ABA 80
Ленивые значения 82
Конкурентные коллекции 86
Конкурентные очереди 88
Конкурентные множества и словари 91
Конкурентные итерации 95

Собственные конкурентные структуры данных 97
Реализация неблокирующего конкурентного пула 98
Создание и обработка процессов 102
В заключение 103
Упражнения 104

Глава 4. Асинхронное программирование с объектами Future и Promise 107
Объекты Future 108
Запуск асинхронных вычислений 109
Объекты Future и обратные вызовы 111
Объекты Future и исключения 113
Использование типа Try 114
Фатальные исключения 115
Композиция функций в объектах Future 116
Объекты Promise 123
Преобразование программных интерфейсов на основе обратных вызовов 125
Расширение программного интерфейса объектов Future 127
Отмена асинхронных вычислений 128
Объекты Future и блокировка выполнения 130
Ожидание завершения Future 130
Блокировка в асинхронных вычислениях 131
Библиотека Scala Async 132
Альтернативные фреймворки асинхронных вычислений 134
В заключение 135
Упражнения 136

Глава 5. Параллельные коллекции данных 139
Краткий обзор коллекций в Scala 140
Использование параллельных коллекций 140
Иерархия классов параллельных коллекций 144
Настройка уровня параллелизма 146
Измерение производительности в JVM 146
Особенности параллельных коллекций 149
Непараллелизуемые коллекции 149
Непараллелизуемые операции 150
Побочные эффекты в параллельных операциях 152
Недетерминированные параллельные операции 153
Коммутативность и ассоциативность операторов 154
Совместное использование параллельных и конкурентных коллекций 155
Слабо согласованные итераторы 156
Реализация собственных параллельных коллекций 157
Сплиттеры 158
Комбинаторы 161
В заключение 163
Упражнения 164

Глава 6. Конкурентное программирование с Reactive Extensions 166
Создание объектов Observable 167
Объекты Observable и исключения 169
Контракт наблюдаемого объекта 170
Реализация собственных объектов Observable 172

Создание наблюдаемых объектов из объектов Future ... 173
Подписки ... 174
Объединение объектов Observable ... 176
Вложенные наблюдаемые объекты ... 178
Обработка ошибок в наблюдаемых объектах ... 182
Планировщики Rx ... 184
Использование собственных планировщиков в приложениях с графическим интерфейсом ... 185
Субъекты и реактивное программирование сверху вниз ... 190
В заключение ... 194
Упражнения ... 194

Глава 7. Программная транзакционная память ... 197
Недостатки атомарных переменных ... 198
Использование программной транзакционной памяти ... 201
Транзакционные ссылки ... 204
Использование инструкции atomic ... 205
Комбинирование транзакций ... 206
Взаимодействие транзакций и побочные эффекты ... 206
Транзакции с одной операцией ... 210
Вложенные транзакции ... 211
Транзакции и исключения ... 214
Повторение транзакций ... 218
Повторения с тайм-аутами ... 221
Транзакционные коллекции ... 222
Локальные переменные транзакций ... 222
Транзакционные массивы ... 224
Транзакционные словари ... 225
В заключение ... 226
Упражнения ... 227

Глава 8. Акторы ... 230
Работа с акторами ... 231
Создание экземпляров и систем акторов ... 233
Управление необработанными сообщениями ... 236
Поведение и состояние актора ... 237
Иерархии акторов в Akka ... 241
Идентификация акторов ... 244
Жизненный цикл акторов ... 246
Взаимодействия между акторами ... 249
Шаблон «запрос» ... 251
Шаблон «пересылка» ... 253
Остановка акторов ... 254
Диспетчеризация акторов ... 255
Удаленные акторы ... 260
В заключение ... 263
Упражнения ... 264

Глава 9. Конкуренция на практике ... 266
Выбор правильных инструментов для решения конкретных задач ... 266
Объединяем все вместе – сетевой браузер файлов ... 270

Моделирование файловой системы ... 272
Интерфейс связи с сервером ... 275
Программный интерфейс навигации на стороне клиента ... 276
Пользовательский интерфейс на стороне клиента ... 279
Реализация логики клиента ... 282
Усовершенствование сетевого браузера файлов ... 286
Отладка конкурентных программ ... 287
Взаимоблокировки и отсутствие прогресса ... 288
Отладка ошибочных результатов ... 292
Отладка производительности ... 296
В заключение ... 302
Упражнения ... 303

Глава 10. Реакторы ... 305
Необходимость реакторов ... 306
Введение в фреймворк Reactors ... 307
Программа «Hello World» ... 308
Потоки событий ... 309
Жизненный цикл потока событий ... 310
Комбинирование потоков событий ... 311
Реакторы ... 313
Определение и настройка реакторов ... 314
Использование каналов ... 315
Планировщики ... 317
Жизненный цикл реактора ... 319
Службы системы реакторов ... 320
Служба журналирования ... 321
Служба времени ... 321
Служба каналов ... 322
Пользовательские службы ... 323
Протоколы ... 325
Собственная реализация протокола клиент-сервер ... 325
Стандартный протокол сервер-клиент ... 327
Протокол маршрутизации ... 330
Протокол двустороннего обмена ... 331
В заключение ... 334
Упражнения ... 335

Предисловие

Конкурентное и параллельное программирование постепенно превращается из узкоспециализированной дисциплины, интересной только специалистам, занимающимся разработкой ядра операционной системы или высокопроизводительными вычислениями, до уровня знаний, которыми должен обладать каждый профессиональный программист. По мере превращения параллельных и распределенных вычислений в норму большинство приложений будет создаваться конкурентными – для увеличения производительности или обработки асинхронных событий.

Но пока большинство разработчиков не готово к этой революции. Возможно, уже давно они изучали традиционную модель параллельных вычислений, основанную на потоках и блокировках, но эта модель не способна обеспечить высокую надежность и приемлемую производительность в приложениях с массовым параллелизмом. В действительности потоки и блокировки сложны в использовании, а пользоваться ими правильно еще сложнее. Чтобы добиться успеха, необходимо использовать абстракции конкурентного выполнения, находящиеся на более высоком уровне и допускающие объединение.

15 лет назад я работал над предшественником Scala – экспериментальным языком Funnel со встроенной семантикой конкуренции. Все понятия программирования были реализованы в этом языке как синтаксический сахар поверх функциональных сетей, объектно-ориентированного варианта исчисления соединений процессов (join calculus). Даже притом, что исчисление соединений процессов считается замечательной теорией, после нескольких экспериментов мы поняли, что проблема конкуренции более многогранна, из-за чего ее трудно выразить единственной формулировкой. Нет общего рецепта, решающего все проблемы конкуренции; правильное решение во многом зависит от конечной цели. Хотите реализовать асинхронные вычисления, запускаемые в ответ на события или при получении потоков значений? Или получить автономные объекты, изолированные сущности, взаимодействующие друг с другом посредством сообщений? Или определить транзакции поверх изменяемого хранилища? Или, может быть, главной целью организации параллельных вычислений является увеличение производительности? Для каждой из этих задач существует своя абстракция, решающая свои проблемы: отложенные вычисления, реактивные потоки данных, акторы, транзакционная память или коллекции с поддержкой параллельных операций.

Это привело нас к языку Scala и к данной книге. Из-за большого количества полезных абстракций конкуренции идея встроить их все в язык программирования не показалась нам захватывающей. Главной целью создания Scala было упрощение определения высокоуровневых абстракций в пользовательском коде и библиотеках. Благодаря этому любой сможет определять модули, реализующие разные аспекты конкурентного программирования. Все эти модули могут быть основаны на низкоуровневых примитивах, предоставляемых операционной системой. По прошествии времени можно уверенно сказать, что этот подход оправдал себя. Современный язык Scala имеет несколько мощных и элегантных библиотек для

конкурентного программирования. Эта книга познакомит вас с некоторыми из них, объяснит область применения каждой и представит прикладные шаблоны.

Трудно найти более авторитетного специалиста, чем автор этой книги. Александр Прокопец (Aleksandar Prokopec) участвовал в развитии нескольких наиболее популярных библиотек поддержки конкурентного и параллельного программирования в Scala. Он также предложил некоторые из особенно сложных структур данных и алгоритмов. В лице этой книги он создал простое и понятное руководство и одновременно надежный справочник для области, в которой он работает. Я уверен, что книгу «Конкурентное программирование на Scala» должен прочитать каждый, кто пишет конкурентные и параллельные программы на Scala. Я также предполагаю, что она заинтересует всех, кто просто желает узнать больше об этой интереснейшей и быстро развивающейся области компьютерных вычислений.

Мартин Одерски (*Martin Odersky*),
профессор федеральной
политехнической школы Лозанны (EPFL),
создатель Scala

Об авторе

Александр Прокопец (Aleksandar Prokopec) – исследователь в области конкурентного и распределенного программирования. Имеет степень доктора компьютерных наук, полученную в федеральной политехнической школе Лозанны (École Polytechnique Fédérale de Lausanne), Швейцария. Работал в Google и в настоящее время работает ведущим исследователем в Oracle Labs.

Став членом команды разработчиков Scala в EPFL, Александр активно участвовал в создании и развитии этого языка программирования и занимался разработкой абстракций конкуренции, параллельной обработки данных и конкурентных структур данных для Scala. Он создал библиотеку параллельных коллекций для Scala, предназначенную для высокоуровневого программирования алгоритмов параллельной обработки данных, и участвовал в рабочих группах по разработке таких конкурентных библиотек для Scala, как Futures/Promises и ScalaSTM. Александр – главный автор реактивной модели программирования для распределенных вычислений.

Вступление

Конкуренция повсюду. С появлением на рынке многоядерных процессоров на разработчиков обрушился лавинообразный спрос на конкурентные программы. Конкурентное программирование, которое прежде использовалось для выражения асинхронных операций в программах и компьютерных системах и считалось академической дисциплиной, в настоящее время превратилось в распространенную методологию. Как результат с невероятной скоростью стали развиваться перспективные фреймворки и библиотеки для поддержки конкурентного программирования. В последние годы конкурентное программирование переживает свой ренессанс.

С ростом уровня абстракций конкуренции в современных языках и фреймворках становится все важнее знать, как и когда их использовать. Хорошее понимание особенностей работы классических примитивов конкуренции и синхронизации, таких как потоки выполнения, блокировки и мониторы, больше не является обязательным условием. Высокоуровневые фреймворки, решающие многие традиционные проблемы конкуренции и ориентированные на решение конкретных задач, постепенно завоевывают мир конкурентного программирования.

Эта книга описывает высокоуровневые приемы конкурентного программирования на языке Scala. В ней представлены подробные объяснения разных тем, связанных с конкуренцией, и охватывается базовая теория конкурентного программирования. Одновременно в ней описываются современные фреймворки поддержки конкурентного программирования, их семантика и приемы использования. Цель книги – познакомить вас с важнейшими абстракциями конкурентного программирования и в то же время показать, как они работают в реальном коде.

Мы верим, что, прочитав эту книгу, вы получите прочное понимание теории конкурентного программирования и приобретете полезные практические навыки, необходимые для создания правильных и эффективных конкурентных программ. Эти навыки станут первым шагом на пути к превращению вас в эксперта по конкурентному программированию.

Мы надеемся, что чтение этой книги доставит вам столько уже удовольствия, сколько получили мы, когда работали над ней.

О чем рассказывается в книге

Эта книга организована как последовательность глав, описывающих разные аспекты конкурентного программирования. Книга охватывает основные программные интерфейсы (API) конкурентного программирования в библиотеке времени выполнения Scala, знакомит с примитивами конкуренции и дает обширный обзор высокоуровневых абстракций.

Глава 1 «*Введение*» разъясняет необходимость конкурентного программирования и дает некоторое философское обоснование. Также она охватывает основы языка программирования Scala, необходимые для понимания остальной части книги.

Глава 2 «*Конкуренция в JVM и модель памяти в Java*» рассказывает об основах конкурентного программирования. В этой главе вы узнаете, как работать с потоками выполнения и защищать доступ к общей памяти, а также познакомитесь с моделью памяти в Java.

Глава 3 «*Традиционные строительные блоки конкурентных программ*» представляет классические инструменты реализации конкуренции, такие как пулы потоков, атомарные переменные и коллекции с поддержкой параллельного доступа, и демонстрирует примеры их использования с учетом особенностей языка Scala. Основное внимание в этой книге уделяется современным высокоуровневым фреймворкам конкурентного программирования. Поэтому данная глава, представляющая обзор традиционных приемов конкурентного программирования, не погружается слишком глубоко в их детали.

Глава 4 «*Асинхронное программирование с объектами Future и Promise*» – это первая глава, посвященная мезанизмам поддержки конкуренции в Scala. Эта глава знакомит с программным интерфейсом объектов `Future` и `Promise` и показывает, как правильно пользоваться ими для реализации асинхронных программ.

Глава 5 «*Параллельные коллекции данных*» описывает множество параллельных коллекций в Scala. В этой главе вы узнаете, как распараллелить операции с коллекцией, когда это возможно, и как оценить прирост производительности от такого распараллеливания.

Глава 6 «*Конкурентное программирование с Reactive Extensions*» научит приемам событийного и асинхронного программирования с использованием фреймворка Reactive Extensions. Здесь вы увидите взаимосвязь операций с потоками событий и коллекциями, узнаете, как передавать события между потоками выполнения и как проектировать реактивные пользовательские интерфейсы с использованием потоков событий.

Глава 7 «*Программная транзакционная память*» знакомит с библиотекой ScalaSTM для транзакционного программирования, реализующей безопасную и понятную модель общей памяти. В этой главе вы узнаете, как защитить доступ к общим данным с помощью масштабируемых транзакций и в то же время снизить риск взаимоблокировки и состояния гонки.

Глава 8 «*Акторы*» описывает модель программирования с акторами, реализованную в фреймворке Akka. В этой главе вы узнаете, как создавать распределенные приложения, выполняющиеся на нескольких компьютерах, которые опираются на механизм обмена сообщениями.

Глава 9 «*Конкуренция на практике*» сравнивает разные библиотеки поддержки конкуренции, представленные в предыдущих главах. В этой главе вы узнаете, как выбрать правильную абстракцию конкуренции для решения той или иной задачи и как комбинировать разные абстракции при проектировании больших конкурентных приложений.

Глава 10 «*Реакторы*» представляет модель программирования «Реактор», целью которой является усовершенствование структуры конкурентных и распределенных программ. Эта новая модель позволяет выделить шаблоны конкурентного и распределенного программирования в модульные компоненты, называемые протоколами.

Мы рекомендуем читать главы по порядку, сверху вниз, но в целом в этом нет необходимости. Если вы хорошо знакомы с темами, обсуждаемыми в главе 2 «Кон-

куренция в JVM и модель памяти в Java», вы можете пропустить ее и сразу перейти к другим главам. Из всех глав на содержимое предыдущих глав опираются только глава 9 «Конкуренция на практике», где дается практический обзор предыдущих тем, и глава 10 «Реакторы», при чтении которой полезно иметь представление о том, как действуют акторы и потоки событий.

Что потребуется для работы с книгой

В этом разделе описываются некоторые требования, которые должны быть выполнены, чтобы вы смогли читать и понимать эту книгу. Мы расскажем, как установить библиотеку Java Development Kit, необходимую для запуска программ на Scala, и покажем, как использовать Simple Build Tool для запуска разных примеров.

Мы не требуем наличия интегрированной среды разработки. Вам выбирать, какие программы использовать для написания кода – Vim, Emacs, Sublime Text, Eclipse, IntelliJ IDEA, Notepad++ или какие-то другие.

Установка JDK

Программы на Scala не компилируются непосредственно в машинный код, поэтому не могут запускаться как выполняемые файлы на разных аппаратных платформах. Компилятор Scala производит промежуточный код в особом формате, который называют байт-кодом Java. Чтобы вы могли запустить этот промежуточный код, на вашем компьютере должна быть установлена виртуальная машина Java. В этом разделе мы расскажем, как загрузить и установить библиотеку Java Development Kit, включающую виртуальную машину Java и другие полезные инструменты.

Существует несколько разных реализаций JDK от разных производителей. Мы рекомендуем использовать дистрибутив Oracle JDK. Чтобы загрузить и установить Java Development Kit, выполните следующие шаги:

1. Откройте в веб-браузере следующую страницу: www.oracle.com/technetwork/java/javase/downloads/index.html.
2. Если эта страница не открывается, перейдите на главную страницу какой-нибудь поисковой системы и введите слова: **JDK Download** (JDK скачать).
3. Найдите в результатах ссылку для загрузки Java SE на сайте Oracle, загрузите соответствующую версию JDK 7 для своей операционной системы: Windows, Linux или Mac OS X; 32- иди 64-разрядную.
4. Если вы пользуетесь операционной системой Windows, просто запустите программу установки. В Mac OS X откройте dmg-архив, чтобы установить JDK. Наконец, в Linux распакуйте архив в каталог по своему выбору, например XYZ, и добавьте подкаталог bin в переменную PATH:

   ```
   export PATH=XYZ/bin:$PATH
   ```

5. Теперь должна появиться возможность запустить в терминале команды `java` и `javac`. Введите команду `javac` и убедитесь, что она доступна (далее в книге вам не придется запускать ее непосредственно, но данный шаг помогает убедиться, что она доступна).

Может так получиться, что в вашей системе уже имеется установленная версия JDK. Чтобы проверить это, просто попробуйте выполнить команду `javac`, как в последнем шаге в описании выше.

Установка и использование SBT

Simple Build Tool (SBT) – это инструмент командной строки, предназначенный для сборки проектов на Scala. Его задача – компиляция кода на Scala, управление зависимостями, непрерывная компиляция и тестирование, развертывание и многое другое. На протяжении всей книги мы будем использовать SBT для управления зависимостями наших проектов и запуска примеров кода.

Чтобы установить SBT, выполните следующие инструкции:

1. Перейдите на страницу http://www.scala-sbt.org/.
2. Загрузите дистрибутив для своей платформы. Если вы используете Windows, загрузите файл `msi`. Если вы пользуетесь Linux или OS X, загрузите архив `zip` или `tgz`.
3. Установите SBT. В Windows просто запустите программу установки. В Linux или OS X распакуйте содержимое архива в домашний каталог.

Теперь SBT готов к использованию. Выполните следующие шаги, чтобы создать новый проект SBT:

1. В Windows запустите программу Command Prompt (Командная строка), а в Linux или OS X откройте окно терминала.
2. Создайте пустой каталог с именем `scala-concurrency-examples`:

   ```
   $ mkdir scala-concurrency-examples
   ```

3. Перейдите в каталог `scala-concurrency-examples`:

   ```
   $ cd scala-concurrency-examples
   ```

4. Создайте каталог для наших примеров:

   ```
   $ mkdir src/main/scala/org/learningconcurrency/
   ```

5. Теперь откройте редактор и создайте в нем файл с именем `build.sbt`. Этот файл определяет различные свойства проекта. Создайте его в корневом каталоге проекта (`scala-concurrency-examples`). Добавьте в файл следующие строки (имейте в виду, что пустые строки являются обязательными):

   ```
   name := "concurrency-examples"

   version := "1.0"

   scalaVersion := "2.11.1"
   ```

6. Наконец, вернитесь в окно терминала и запустите SBT в корневом каталоге проекта:

   ```
   $ sbt
   ```

7. SBT запустит интерактивную оболочку, которую можно использовать для ввода различных команд сборки.

Теперь вы можете начинать писать программы на Scala. Откройте редактор и создайте в каталоге `src/main/scala/org/learningconcurrency` файл с именем `HelloWorld.scala`. Добавьте в него следующие строки:

```
package org.learningconcurrency

object HelloWorld extends App {
  println("Hello, world!")
}
```

Теперь вернитесь в окно терминала с запущенной интерактивной оболочкой SBT и запустите программу следующей командой:

```
> run
```

Эта программа должна вывести в терминал:

```
Hello, world!
```

Этих шагов достаточно для большинства примеров в данной книге. Иногда мы будем полагаться на внешние библиотеки, запуская примеры. Эти библиотеки автоматически обнаруживаются инструментом SBT в стандартных репозиториях. Для некоторых библиотек нам понадобится указать дополнительные репозитории, поэтому добавим следующие строки в фал `build.sbt`:

```
resolvers ++= Seq(
  "Sonatype OSS Snapshots" at
    "https://oss.sonatype.org/content/repositories/snapshots",
  "Sonatype OSS Releases" at
    "https://oss.sonatype.org/content/repositories/releases",
  "Typesafe Repository" at
    "http://repo.typesafe.com/typesafe/releases/"
)
```

Теперь, включив все необходимые репозитории, можно добавить несколько библиотек. Добавив следующую строку в файл `build.sbt`, мы получим доступ к библиотеке Apache Commons IO:

```
libraryDependencies += "commons-io" % "commons-io" % "2.4"
```

После изменения файла `build.sbt` его нужно перезагрузить во всех запущенных экземплярах SBT. Для этого в интерактивной оболочке SBT введите следующую команду:

```
> reload
```

Это позволит SBT определить любые изменения в определениях в файле сборки и загрузить необходимые дополнительные программные пакеты.

Разные библиотеки Scala находятся в разных пространствах имен, которые называют пакетами. Чтобы получить доступ к содержимому определенного пакета, мы используем инструкцию `import`. Когда в примерах мы впервые используем некоторую библиотеку, мы всегда показываем набор необходимых инструкций `import`, но потом мы просто опускаем их ради экономии места.

По той же причине мы не повторяем объявление пакета примерах кода. Мы всегда предполагаем, что все примеры в каждой конкретной главе принадлежат одному пакету. Например, все примеры в главе 2 «*Конкуренция в JVM и модель памяти в Java*» находятся в пакете `org.learningconcurrency.ch2`. Файлы с исходным кодом примеров из этой главы начинаются со следующих строк:

```
package org.learningconcurrency
package ch2
```

Наконец, в этой книге рассказывается о конкуренции и асинхронном выполнении. Многие примеры запускают параллельные вычисления, которые могут

продолжаться после завершения основной программы. Чтобы гарантировать, что такие параллельные вычисления всегда будут завершаться вместе с главной программой, когда выполняются в одной оболочке SBT, добавим следующую строку в файл `build.sbt`:

```
fork := false
```

В примерах, когда процессы должны выполняться под управлением разных экземпляров JVM, мы будет ясно указывать на это и давать четкие инструкции.

Использование Eclipse, IntelliJ IDEA или другой IDE

Преимущество использования интегрированной среды разработки (Integrated Development Environment, IDE), такой как Eclipse или IntelliJ IDEA, заключается в возможности писать, компилировать и автоматически запускать программы на Scala. В этом случае нет необходимости устанавливать SBT, как описывалось в предыдущем разделе. Хотя мы советуем опробовать примеры из книги с помощью SBT, вы свободно можете использовать для этой цели свою IDE.

Однако мы должны предупредить, что такие редакторы, как Eclipse и IntelliJ IDEA, запускают программы в отдельном процессе JVM. Как отмечалось в предыдущем разделе, некоторые параллельные вычисления продолжают выполняться после завершения основной программы. Чтобы гарантировать завершение параллельных вычислений вместе с самой программой, вам может понадобиться добавить инструкцию `sleep` в конец главной программы с целью задержать ее. В большинстве примеров в этой книге инструкция `sleep` уже добавлена, но в некоторых программах вам может потребоваться добавить ее самим.

Кому адресована книга

Эта книга адресована в первую очередь разработчикам, умеющим писать последовательные программы на Scala и желающим научиться писать конкурентные программы. Предполагается, что у вас уже есть опыт программирования на языке Scala. На протяжении всей книги мы стремились использовать только наиболее простые особенности языка, чтобы показать, как писать конкурентный код, поэтому даже при наличии элементарных знаний вы не должны испытывать проблем в изучении предлагаемых в книге тем.

Однако нельзя сказать, что книга адресована исключительно разработчикам на Scala. Даже если вы имеете опыт программирования на Java, в .NET или на любом другом языке, вы наверняка найдете эту книгу полезной. Понимание основ объектно-ориентированного или функционального программирования является достаточным условием для чтения этой книги.

Наконец, эта книга является отличным введением в современное конкурентное программирование вообще. Даже если вы обладаете навыками реализации многопоточных вычислений и знаете, как устроена модель конкуренции в JVM, вы все равно узнаете здесь много нового о современных высокоуровневых средствах поддержки конкуренции. Многие библиотеки, представленные в этой книге, только начинают прокладывать свой путь к главенствующим языкам программирования, и некоторые из них являются по-настоящему передовыми технологиями.

Соглашения

В этой книге вы обнаружите несколько стилей оформления текста, которые разделяют различные виды информации. Ниже приводятся примеры этих стилей и поясняется их значение.

Элементы программного кода в тексте, имена таблиц в базах данных, имена папок и файлов, расширения файлов, пути к каталогам в файловой системе, фиктивные адреса URL, ввод пользователя и учетные записи в Twitter оформляются так: «Следующие инструкции читают ссылку и передают ее в функцию `BeautifulSoup`».

Блоки кода оформляются следующим образом:

```
package org
package object learningconcurrency {
  def log(msg: String): Unit =
    println(s"${Thread.currentThread.getName}: $msg")
}
```

Когда потребуется привлечь ваше внимание к определенному фрагменту в блоке программного кода, он будет выделяться жирным:

```
object ThreadsMain extends App {
  val t: Thread = Thread.currentThread
  val name = t.getName
  println(s"I am the thread $name")
}
```

Ввод или вывод в командной строке будет оформляться так:

```
$ mkdir scala-concurrency-examples
```

Новые термины и **важные слова** будут выделены жирным. Текст, отображаемый на экране, например в меню или в диалогах, будет оформляться так: «Чтобы загрузить новые модули, выберите пункт меню Files | Settings | Project Name | Project Interpreter (Файлы | Настройки | Имя проекта | Интерпретатор проекта)».

Так оформляются предупреждения и важные примечания.

Так оформляются советы и рекомендации.

Отзывы и пожелания

Мы всегда рады отзывам наших читателей. Расскажите нам, что вы думаете об этой книге – что понравилось или, может быть, не понравилось. Отзывы важны для нас, чтобы выпускать книги, которые будут для вас максимально полезны.

Вы можете написать отзыв прямо на нашем сайте www.dmkpress.com, зайдя на страницу книги и оставив комментарий в разделе «Отзывы и рецензии». Также можно послать письмо главному редактору по адресу dmkpress@gmail.com, при этом напишите название книги в теме письма.

Если есть тема, в которой вы квалифицированы, и вы заинтересованы в написании новой книги, заполните форму на нашем сайте по адресу http://dmkpress.com/authors/publish_book/ или напишите в издательство по адресу dmkpress@gmail.com.

Скачивание исходного кода примеров

Скачать файлы с дополнительной информацией для книг издательства «ДМК Пресс» можно на сайте www.dmkpress.com или www.дмк.рф в разделе «Читателям – Файлы к книгам».

Скачивание цветных иллюстраций

Мы также подготовили файл PDF с цветными иллюстрациями, диаграммами и скриншотами, которые в этой книге имеют черно-белое оформление. Цветные иллюстрации помогут вам лучше понять обсуждаемые темы. Вы можете загрузить файл с иллюстрациями по адресу https://www.packtpub.com/sites/default/files/downloads/LearningConcurrentProgrammingInScalaSecondEdition_ColorImages.pdf.

Список опечаток

Хотя мы приняли все возможные меры для того, чтобы удостовериться в качестве наших текстов, ошибки все равно случаются. Если вы найдете ошибку в одной из наших книг – возможно, ошибку в тексте или в коде, – мы будем очень благодарны, если вы сообщите нам о ней. Сделав это, вы избавите других читателей от расстройств и поможете нам улучшить последующие версии этой книги.

Если вы найдете какие-либо ошибки в коде, пожалуйста, сообщите о них главному редактору по адресу dmkpress@gmail.com, и мы исправим это в следующих тиражах.

Нарушение авторских прав

Пиратство в Интернете по-прежнему остается насущной проблемой. Издательства «ДМК Пресс» и Packt очень серьезно относятся к вопросам защиты авторских прав и лицензирования. Если вы столкнетесь в Интернете с незаконно выполненной копией любой нашей книги, пожалуйста, сообщите нам адрес копии или веб-сайта, чтобы мы могли принять меры.

Пожалуйста, свяжитесь с нами по адресу электронной почты dmkpress@gmail.com со ссылкой на подозрительные материалы.

Мы высоко ценим любую помощь по защите наших авторов, помогающую нам предоставлять вам качественные материалы.

Вопросы

Вы можете присылать любые вопросы, касающиеся данной книги, по адресу dm@dmk-press.ru или questions@packtpub.com. Мы постараемся разрешить возникшие проблемы.

Глава 1

Введение

> «Более десяти лет пророки утверждали, что возможности единственного компьютера достигли своего предела и по-настоящему значительные успехи могут быть достигнуты только за счет объединения множества компьютеров».
>
> *– Джин Амдал (Gene Amdahl), 1967*

Дисциплина конкурентного программирования имеет длинную историю, но основное развитие она получила лишь в последние несколько лет, с появлением многоядерных процессоров. Последние достижения в области вычислительной техники не только дали вторую жизнь некоторым классическим приемам, но и вызвали изменения в основной парадигме конкурентного программирования. С ростом важности конкуренции обладание навыками конкурентного программирования стало обязательным условием для каждого программиста.

Эта глава описывает базовые принципы организации конкурентных вычислений и основы языка Scala, знание которых понадобится при чтении книги. В частности, она:

- содержит краткий обзор идей конкурентного программирования;
- знакомит с преимуществами Scala в отношении конкурентного программирования;
- охватывает основы языка, знание которых понадобится при чтении книги.

Сначала мы узнаем, что такое конкурентное программирование и почему владение его приемами так важно.

Конкурентное программирование

В **конкурентном программировании** программа выражается как набор конкурентных вычислений, которые выполняются в перекрывающиеся интервалы времени и координируют свои действия друг с другом некоторым способом. Реализовать конкурентную программу, действующую правильно, обычно намного сложнее, чем последовательную. Все ловушки, присутствующие в последовательной программе, становятся особенно опасными в конкурентной, но, кроме того, существует масса других причин, которые могут вызвать неправильный ход выполнения, о которых вы узнаете в этой книге. Возникает естественный вопрос: как быть? Мы не можем остановить развитие и писать только последовательные программы!

Конкурентное программирование дает множество преимуществ. Во-первых, конкурентные программы имеют **более высокую производительность**. Вместо выполнения на единственном процессоре такие программы разбивают решаемые задачи на несколько подзадач, которые выполняются на разных процессорах, благодаря чему ускоряется работа программы в целом. С появлением многоядерных процессоров это стало одной из основных причин оживления внимания к конкурентному программированию.

Модель конкурентного программирования способна ускорить выполнение операций ввода/вывода. Чисто последовательная программа должна периодически опрашивать устройства ввода/вывода, такие как клавиатура, сетевой интерфейс или другое устройство, и проверять наличие входных данных. Конкурентная программа, напротив, может реагировать на запросы ввода/вывода немедленно. Если программа ведет интенсивный обмен данными с внешними устройствами, поддержка конкурентного выполнения в ней может привести к увеличению пропускной способности, и это одна из причин, почему поддержка конкуренции в языках программирования появилась раньше, чем многопроцессорные системы. То есть конкурентное выполнение помогает увеличить отзывчивость программ, взаимодействующих со своим окружением.

Наконец, конкурентное выполнение может упростить реализацию и эксплуатацию компьютерных программ. В некоторых случаях реализация конкурентных программ оказывается более компактной. Иногда удобнее разделить программу на небольшие блоки, выполняющиеся независимо, чем **computations than to incorporate everything into one large program. User interfaces, web servers, and game engines are typical examples of such systems**. *ПЕРЕВОД*

В этой книге мы будем следовать соглашению о том, что конкурентные программы взаимодействуют посредством общей памяти и выполняются на одном компьютере. Напротив, когда программа выполняется на нескольких компьютерах, каждый ее экземпляр имеет собственную память – такие программы называются **распределенными**, а дисциплина разработки таких программ называется **распределенным программированием**. Как правило, распределенная программа должна быть готова, что каждый из используемых ею компьютеров в любой момент может выйти из строя, и предусматривать некоторые гарантии надежности на этот случай. Основное внимание мы сосредоточим на конкурентных программах, но также рассмотрим несколько примеров распределенных программ.

Краткий обзор традиционных подходов к организации конкурентного выполнения

В компьютерных системах конкурентные вычисления могут быть организованы на уровне аппаратуры, операционной системы или языка программирования. Мы в основном будем исследовать организацию на уровне языка программирования.

Согласование работы конкурентных вычислений в системе называется **синхронизацией** и является ключевым аспектом успешной реализации конкуренции. Под синхронизацией подразумеваются механизмы, используемые для согласования конкурирующих вычислений по времени. Кроме того, механизмы синхронизации определяют, как конкурирующие вычисления взаимодействуют

друг с другом, то есть как они изменяют информацию. Взаимодействия в конкурентных программах осуществляются путем изменения значений в общей памяти. Этот вид синхронизации называют **взаимодействием через общую память**. Взаимодействия в распределенных программах осуществляются посредством сообщений, поэтому такой вид синхронизации называют **взаимодействием посредством обмена сообщениями**.

На самом нижнем уровне конкурирующие вычисления представлены сущностями, которые называют процессами и потоками выполнения. Подробнее о них рассказывается в главе 2 «*Конкуренция в JVM и модель памяти в Java*». Для управления выполнением процессы и потоки традиционно используют такие сущности, как блокировки и мониторы. Упорядочение взаимодействий между потоками гарантирует, что изменения в памяти в каждый конкретный момент времени сможет выполнять только один поток, и они будут видимы другим потокам, которые обратятся к этой же области памяти позже.

Часто реализация конкурентных программ с использованием потоков и блокировок оказывается чересчур громоздкой. Для решения этой проблемы были разработаны более сложные механизмы поддержки конкуренции, такие как коммуникационные каналы, конкурентные коллекции, барьеры, защелки с обратным отсчетом и пулы потоков. Эти механизмы создавались с целью упростить реализацию конкретных шаблонов конкурентного программирования, и некоторые из них рассматриваются в главе 3 «*Традиционные строительные блоки конкурентных программ*».

Традиционно конкуренция реализуется на относительно низком уровне и чревата множеством разнообразных ошибок, таких как взаимоблокировки, зависание и гонка за ресурсами. Используя для разработки конкурентных программ язык Scala, вам не придется сталкиваться с низкоуровневыми примитивами. И все же знание низкоуровневых основ конкурентного программирования поможет вам понять, как действуют высокоуровневые механизмы.

Современные парадигмы конкуренции

Современные парадигмы организации конкурентного выполнения более совершенные, чем традиционные подходы. Главное отличие заключается в том, что высокоуровневые механизмы позволяют выразить, *какая* цель преследуется, а не *как* она должна быть достигнута.

На практике различия между высоко- и низкоуровневыми механизмами менее очевидны, и разные их реализации образуют непрерывный спектр, а не различные группы. Тем не менее последние исследования в области конкурентного программирования показывают четко выраженную склонность к декларативному и функциональному стилям программирования.

Как будет показано в главе 2 «*Традиционные строительные блоки конкурентных программ*», чтобы вычислить значение конкурентным способом, требуется создать поток выполнения со своим методом `run`, вызвать метод `start`, дождаться завершения потока и затем извлечь результат из известной области памяти. Намного предпочтительнее было бы иметь возможность просто сказать: *вычисли это значение конкурентно и сообщи, когда работа будет выполнена*. Также было бы желательно, чтобы модель программирования скрывала детали координации конкурентных вычислений и позволяла интерпретировать результат, как если бы

он уже имелся, а не ждать, когда вычисления завершатся и затем читать данные из памяти. Именно эту модель, позволяющую выражать свои требования в программе подобным образом, представляет парадигма **асинхронного программирования с использованием объектов Future**. Мы поближе познакомимся с ней в главе 4 «*Асинхронное программирование с объектами Future и Promise*». Аналогично парадигма **реактивного программирования с потоками событий**, которая рассматривается в главе 6 «*Конкурентное программирование с Reactive Extensions*», помогает декларативно выразить конкурентные вычисления, которые производят множество значений.

Декларативный стиль все чаще встречается также в последовательном программировании. Такие языки, как Python, Haskell, Ruby и Scala, позволяют выражать операции с коллекциями в терминах функциональных операторов, например: *отфильтровать все отрицательные целые числа из этой коллекции*. Это предложение выражает цель, а не реализацию ее достижения, что дает простую возможность распараллелить такую операцию за кулисами. В главе 5 «*Параллельные коллекции данных*» описывается поддержка параллельных коллекций в Scala, целью которой является ускорение операций с коллекциями при выполнении на многоядерных процессорах.

Еще одна тенденция, наблюдаемая в высокоуровневых механизмах конкуренции, – специализация для поддержки конкретных задач. Технология программной транзакционной памяти специально предназначена для выражения **транзакций при работе с памятью** и вообще не связана с запуском конкурирующих процессов и/или потоков. Под транзакцией в данном случае подразумевается последовательность операций с памятью, которые либо выполняются все, либо не выполняется ни одна. В этом можно увидеть сходство с транзакциями в базе данных. Преимущество использования транзакционной памяти заключается в возможности избежать массы ошибок, часто возникающих при работе с низкоуровневыми механизмами конкурентного выполнения. Более подробно о транзакционной памяти рассказывается в главе 7 «*Программная транзакционная память*».

Наконец, некоторые высокоуровневые механизмы конкуренции имеют целью обеспечить прозрачную поддержку распределенного программирования. Это особенно верно для механизмов параллельной обработки данных и обмена сообщениями, таких как **акторы**, о которых рассказывается в главе 8 «*Акторы*».

Преимущества языка Scala

Язык Scala все еще считается молодым языком и пока не получил такого распространения, как Java; тем не менее он обладает развитой поддержкой конкурентного программирования. В экосистеме Scala присутствуют и активно разрабатываются механизмы поддержки практически всех видов конкурентного программирования. На протяжении всего своего развития язык Scala расширял спектр современных, высокоуровневых программных интерфейсов, или API для конкурентного программирования. И тому есть множество причин.

Главная причина, объясняющая наличие богатого набора современных механизмов конкуренции в Scala, – врожденная гибкость синтаксиса. Благодаря таким особенностям, как использование функций в роли обычных объектов, именованные параметры, автоматическое определение типа и сопоставление с шаблоном,

о которых рассказывается в следующих разделах, имеется возможность определять API, похожие на встроенные конструкции языка.

Такие API имитируют различные модели программирования посредством встраиваемых предметно-ориентированных языков (Domain-Specific Languages, DSL), для которых Scala служит языком-носителем: акторы, программная транзакционная память и объекты Future – вот примеры программных интерфейсов, выглядящих подобно встроенным конструкциям языка, которые в действительности реализованы как библиотеки. С одной стороны, Scala избавляет от необходимости разрабатывать новый язык для каждой новой модели конкурентного программирования и дает богатую почву для разработчиков конкурентных программ. С другой – увеличенная синтаксическая нагрузка, присутствующая во многих других языках, привлекает к Scala все больше пользователей.

Вторая причина быстрого продвижения языка Scala заключается в его безопасности. Автоматическая сборка мусора, автоматическая проверка границ и отсутствие арифметики указателей помогают избежать таких проблем, как утечка памяти, переполнение буфера и ошибки обращения к недоступной памяти. Статическая типизация также позволяет предотвратить целый класс ошибок на самых ранних стадиях разработки. То же касается конкурентного программирования, которое само чревато разнообразными ошибками: отсутствие необходимости заботиться о различных аспектах многое меняет.

Третья важная причина – совместимость. Программы на Scala компилируются в байт-код Java, поэтому полученный в результате код может выполняться в виртуальной машине Java (Java Virtual Machine, JVM). Это означает, что в программах на Scala можно беспрепятственно использовать существующие библиотеки и взаимодействовать с богатой экосистемой Java. Часто переход на другой язык – болезненный процесс. В случае с языком Scala такой переход с языка Java может происходить постепенно и дается намного проще. Это одна из причин быстрого распространения этого языка, а также объясняет, почему для реализации некоторых Java-совместимых фреймворков выбирается язык Scala.

Важно также отметить, что благодаря возможности выполнения под управлением JVM программы на Scala легко переносятся на другие аппаратные платформы. Кроме того, JVM имеет зрелую модель потоков выполнения и памяти, что гарантирует единообразную работу на разных компьютерах. Переносимость играет важную роль для последовательных программ, но еще более важную роль она играет для конкурентных вычислений.

Познакомившись с некоторыми преимуществами Scala, мы теперь готовы приступить к изучению особенностей языка, имеющих отношение к теме данной книги.

Начальные сведения

Эта книга предполагает знание приемов последовательного программирования. Мы советуем читателям поближе познакомиться с языком программирования Scala, однако для понимания идей, описываемых в этой книге, достаточно будет знания похожих языков, таких как Java или C#. Вам пригодится знание основных понятий объектно-ориентированного программирования, таких как классы, объ-

екты и интерфейсы. Также полезным будет знание основных принципов функционального программирования, таких как функции-объекты, чистые функции и полиморфизм типов, но это не является обязательным требованием.

Выполнение программ на Scala

Чтобы лучше понять модель выполнения программ на Scala, рассмотрим простой пример, который использует метод `square` для вычисления квадрата числа `5` и затем выводит результат в стандартный вывод:

```
object SquareOf5 extends App {
  def square(x: Int): Int = x * x
  val s = square(5)
  println(s"Result: $s")
}
```

Эту программу можно запустить с помощью **Simple Build Tool (SBT)**, как описывалось во вступлении. Когда запускается программа на Scala, среда времени выполнения JVM выделяет ей необходимый объем памяти. Наибольшую важность для нас представляют **стек вызовов** и **куча объектов**. Стек вызовов – это область памяти, где программа хранит информацию о локальных переменных и параметрах методов, выполняющихся в данный момент. Куча объектов (или просто куча) – это область памяти, где программа размещает свои объекты. Чтобы понять разницу между этими двумя областями, рассмотрим упрощенный сценарий выполнения данной программы.

На этапе **1** (рис. 1.1) программа выделяет память в стеке вызовов для локальной переменной `s`. Затем, на этапе **2**, она вызывает метод `square` для вычисления квадрата числа, хранящегося в локальной переменной `s`. Программа помещает число `5` на стек вызовов, которое служит значением для параметра `x`. Она также резервирует на стеке место для значения, возвращаемого методом. В этой точке (этап **3**) программа передает управление методу `square`, который умножает параметр `x` на самого себя и помещает результат `25` в стек.

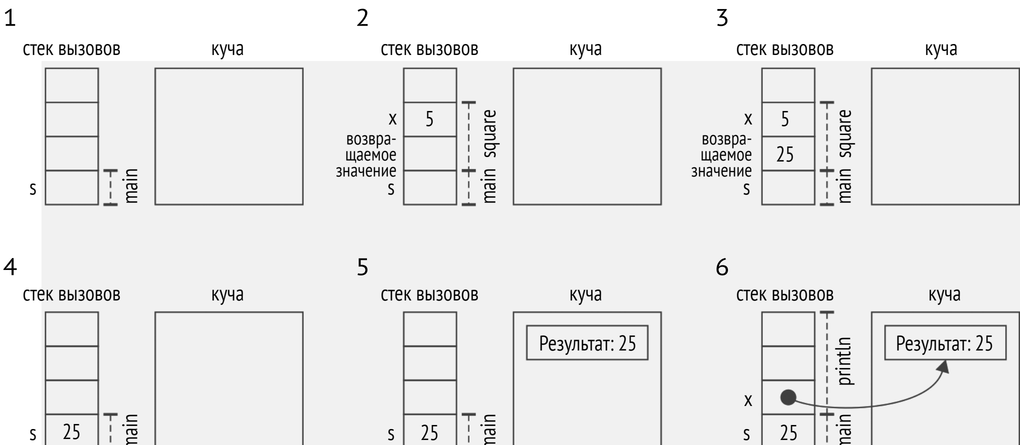


Рис. 1.1 ❖ Этапы выполнения примера программы на Scala

После того как метод `square` вернет результат, число `25` копируется в стек, в память, выделенную для локальной переменной `s` (этап **4**). Теперь программа должна создать строку для инструкции `println`. В языке Scala строки являются экземплярами класса `String`, поэтому программа создает новый объект `String` в куче, как показано на этапе **5**. Наконец, на этапе **6** программа сохраняет ссылку на вновь созданный объект в элементе `x` в стеке и вызывает метод `println`.

Хотя это весьма упрощенное представление, тем не менее оно достаточно наглядно демонстрирует модель выполнения программ на Scala. В главе 2 «*Конкуренция в JVM и модель памяти в Java*» вы узнаете, что каждый поток выполнения имеет свой, отдельный стек вызовов и основные взаимодействия между потоками заключаются в изменении данных, хранящихся в куче. Вы также узнаете, что несоответствия между состоянием кучи и локального стека вызовов часто порождают в конкурентных программах ошибки определенного вида.

Узнав, как выполняется типичная программа на Scala, перейдем к обзору особенностей этого языка, знание которых необходимо для понимания идей, излагаемых в книге.

Основы Scala

В этом разделе мы коротко рассмотрим некоторые особенности языка Scala, которые используются в примерах в этой книге. Это очень краткий и поверхностный обзор основ языка Scala. Имейте в виду, что данный раздел не является полноценным введением в язык. Его цель – только напомнить некоторые особенности языка и сравнить его с другими похожими языками, с которыми вы можете быть знакомы. Желающих узнать больше о Scala мы отсылаем к книгам, перечисленным в разделе «В заключение» в конце этой главы.

Ниже приводится объявление класса `Printer`, который принимает параметр `greeting` и имеет два метода, `printMessage` и `printNumber`:

```
class Printer(val greeting: String) {
  def printMessage(): Unit = println(greeting + "!")
  def printNumber(x: Int): Unit = {
    println("Number: " + x)
  }
}
```

Метод `printMessage` в этом примере не имеет аргументов и содержит единственную инструкцию `println`. Метод `printNumber` принимает единственный аргумент `x` типа `Int`. Оба метода не имеют возвращаемого значения, о чем говорит тип `Unit`.

Вот как осуществляются создание экземпляра класса и вызов его методов:

```
val printy = new Printer("Hi")
printy.printMessage()
printy.printNumber(5)
```

Scala позволяет объявлять **объекты-одиночки**. Такие объявления совмещают в себе объявление класса и создание экземпляра. В предыдущем разделе мы уже видели объект-одиночку `SquareOf5`, который использовался как основа простой программы на Scala. Следующий объект-одиночка `Test` объявляет единственное поле `Pi` со значением `3.14`:

```
object Test {
  val Pi = 3.14
}
```

В похожих языках классы могут наследовать такие сущности, как интерфейсы, а классы в Scala могут наследовать трейты (traits). Трейты в Scala позволяют объявлять конкретные поля и реализации методов. В следующем примере объявляется трейт `Logging`, который реализует вывод сообщений об ошибках и предупреждений с использованием абстрактного метода `log`, который затем подмешивается в класс `PrintLogging`:

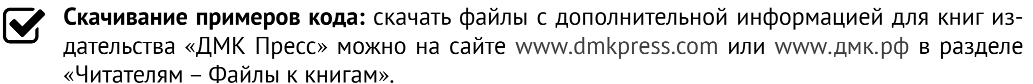
```
trait Logging {
  def log(s: String): Unit
  def warn(s: String) = log("WARN: " + s)
  def error(s: String) = log("ERROR: " + s)
}

class PrintLogging extends Logging {
  def log(s: String) = println(s)
}
```

Классы могут иметь **параметры типов**. Следующий обобщенный (generic) класс `Pair` принимает два параметра типов, `P` и `Q`, которые определяют типы аргументов `first` и `second`:

```
class Pair[P, Q](val first: P, val second: Q)
```

Scala поддерживает также функции-объекты, которые еще называют **лямбда-функциями** или **лямбда-выражениями**. В следующем фрагменте объявляется лямбда-функция `twice`, умножающая свой аргумент на два:

```
val twice: Int => Int = (x: Int) => x * 2
```


Скачивание примеров кода: скачать файлы с дополнительной информацией для книг издательства «ДМК Пресс» можно на сайте www.dmkpress.com или www.дмк.рф в разделе «Читателям – Файлы к книгам».

В этом примере часть `(x: Int)` определяет аргумент лямбда-функции, а `x * 2` – ее тело. Аргументы и тело лямбда-функции должны отделяться символами `=>`. Та же пара символов `=>` используется для выражения типа лямбда-функции, в данном примере `Int => Int`, произносится как «Int в Int». Аннотацию типа `Int => Int` в данном случае можно опустить, как показано в следующем фрагменте, потому что компилятор автоматически определит тип лямбда-функции `twice`:

```
val twice = (x: Int) => x * 2
```

Также в лямбда-функции можно опустить объявление типа аргумента и использовать более компактный синтаксис:

```
val twice: Int => Int = x => x * 2
```

Наконец, если аргумент появляется в теле лямбда-функции только один раз, Scala позволяет использовать еще более краткое объявление:

```
val twice: Int => Int = _ * 2
```

Функции-объекты позволяют манипулировать блоками кода, как если бы они были обычными значениями. Они поддерживают упрощенный и более компактный синтаксис. В следующем примере объявляется метод runTwice с **именованным параметром**, который дважды выполняет заданный фрагмент кода:

```
def runTwice(body: => Unit) = {
  body
  body
}
```

Именованный параметр формируется путем добавления аннотации => перед типом. Всякий раз, когда метод runTwice ссылается на аргумент body, выражение в нем вычисляется заново, как показано в следующем фрагменте:

```
runTwice { // выведи слово Hello дважды
  println("Hello")
}
```

Выражение for в Scala дает удобный способ обхода и преобразования элементов коллекций. Следующий цикл for выводит числа в диапазоне от 0 до 10; причем число 10 не входит в диапазон:

```
for (i <- 0 until 10) println(i)
```

В этом фрагменте диапазон создается выражением 0 until 10, эквивалентным выражению 0.until(10), которое вызывает метод until значения 0. В Scala иногда разрешается отбрасывать точку в вызовах методов объектов.

Циклы for эквивалентны вызовам метода foreach. Предыдущий цикл for компилятор Scala автоматически преобразует в следующее выражение:

```
(0 until 10).foreach(i => println(i))
```

Так называемые for-генераторы часто используются для преобразования данных. Следующий for-генератор преобразует все числа в диапазоне от 0 до 10, умножая их на -1:

```
val negatives = for (i <- 0 until 10) yield -i
```

Переменной negatives будет присвоена коллекция целых чисел от 0 до -10. Этот for-генератор эквивалентен следующему вызову map:

```
val negatives = (0 until 10).map(i => -1 * i)
```

Также есть возможность преобразовывать данные, поступающие из нескольких источников. Следующий for-генератор создаст все возможные пары целых чисел в диапазоне от 0 до 4:

```
val pairs = for (x <- 0 until 4; y <- 0 until 4) yield (x, y)
```

Предыдущий for-генератор эквивалентен следующему выражению:

```
val pairs = (0 until 4).flatMap(x => (0 until 4).map(y => (x, y)))
```

В for-генератор можно вложить сколько угодно выражений-генераторов. Компилятор Scala автоматически преобразует их в последовательность вложенных вызовов flatMap с последующим вызовом map на самом глубоком уровне вложенности.

К числу наиболее часто используемых коллекций в языке Scala относятся последовательности, обозначаемые типом `Seq[T]`; словари, обозначаемые типом `Map[K, V]`; и множества, обозначаемые типом `Set[T]`. Например, в следующем фрагменте создается последовательность строк:

```
val messages: Seq[String] = Seq("Hello", "World.", "!")
```

На протяжении всей книги мы широко будем использовать поддержку **интерполяции строк**. Обычно строки в Scala обозначаются как последовательности символов, заключенные в двойные кавычки. Интерполируемые строки обозначаются символом `s`, предшествующим открывающей кавычке, и могут содержать символы `$` с произвольными идентификаторами, доступными в текущей области видимости, как показано в следующем примере:

```
val magic = 7
val myMagicNumber = s"My magic number is $magic"
```

Сопоставление с шаблоном – еще одна важная особенность Scala. Читателям с опытом программирования на Java, C# и C будет понятнее, если сказать, что инструкция `match` в Scala фактически является инструкцией `switch` с расширенными возможностями (или, как говорят, «на стероидах»). Инструкция `match` способна анализировать данные любых типов и позволяет кратко выражать варианты выбора.

В следующем примере объявляется коллекция `Map` с именем `successors`, в которой целочисленным ключам соответствуют целочисленные значения на единицу больше. Затем вызывается метод `get`, чтобы получить число, следующее за числом `5`. Метод `get` возвращает объект типа `Option[Int]`, который может принадлежать классу `Some`, что указывает на присутствие ключа `5` в словаре, либо классу `None`, указывающему, что ключ `5` отсутствует в словаре. Сопоставление объекта `Option` с шаблоном позволяет выбрать требуемый вариант:

```
val successors = Map(1 -> 2, 2 -> 3, 3 -> 4)
successors.get(5) match {
  case Some(n) => println(s"Successor is: $n")
  case None => println("Could not find successor.")
}
```

Для большинства операторов в Scala можно реализовать перегруженную версию. Перегрузка оператора ничем не отличается от объявления метода. В следующем фрагменте объявляется класс `Position` с оператором `+` («плюс»):

```
class Position(val x: Int, val y: Int) {
  def +(that: Position) = new Position(x + that.x, y + that.y)
}
```

Наконец, Scala позволяет объявлять **объекты пакетов** для хранения определений методов и значений на верхнем уровне для данного пакета. В следующем фрагменте объявляется объект пакета для пакета `org.learningconcurrency`. В нем присутствует реализация метода верхнего уровневого `log`, который выводит заданную строку и имя текущего потока выполнения:

```
package org
package object learningconcurrency {
  def log(msg: String): Unit =
```

```
    println(s"${Thread.currentThread.getName}: $msg")
}
```

Мы будем использовать метод `log` во многих примерах в этой книге для трассировки выполнения программ.

На этом мы завершаем краткий обзор важнейших особенностей языка Scala. Желающим получить более подробные сведения мы предлагаем обратиться к одной из вводных книг по последовательному программированию на Scala.

Обзор новых особенностей в Scala 2.12

На момент написания этой книги планировалась к выпуску следующая версия языка – Scala 2.12. С точки зрения пользователя и API, в Scala 2.12 не появилось особенно заметных изменений. Цель выпуска версии 2.12 состоит в усовершенствовании оптимизации кода и обеспечении совместимости со средой выполнения Java 8. Поскольку Scala в первую очередь ориентирован на среду выполнения Java, поддержка совместимости с Java 8 позволит уменьшить размеры скомпилированных программ в JAR-файлах, увеличить производительность и скорость компиляции. С точки зрения пользователя главное отличие заключается в том, что для этой версии требуется установить JDK 8 вместо JDK 7.

Ниже перечислены особенно значимые изменения в Scala 2.12.

- В предыдущих версиях трейты компилировались в интерфейсы, если включали только абстрактные методы. Если трейт имел конкретную реализацию метода, компилятор создавал два файла класса – один с интерфейсом JVM и другой с классом, реализующим конкретные методы. В Scala 2.12 компилятор генерирует единственный файл интерфейса, содержащий **методы по умолчанию**, поддерживаемые в Java 8. Это позволяет уменьшить общий размер кода.
- Прежде каждое замыкание в Scala компилировалось в отдельный класс. Начиная с версии 2.12 замыкания компилируются в лямбда-выражения Java 8. Это также способствует уменьшению размера кода и теоретически позволяет применять оптимизации, предусмотренные средой выполнения Java 8.
- Исходный код на Scala компилируется в байт-код Java, который затем интерпретируется виртуальной машиной Java. В Scala 2.12 на смену старому пришел новый, более производительный компилятор, положительно влияющий на скорость компиляции.
- В Scala 2.12 включен новый оптимизатор, который можно включить с помощью флага `-opt` компилятора. Новый оптимизатор действует более агрессивно в отношении встраиваемых финальных методов, точнее определяет объекты и функции, которые создаются и используются в единственном методе, и устраняет «мертвый» код. Все это положительно сказывается на производительности Scala-программ.
- Scala 2.12 позволяет использовать лямбда-функции для типов с единственным абстрактным методом (Single Abstract Method, SAM). SAM-типы – это классы или трейты, имеющие только один абстрактный метод, который обычно реализуется в наследующем его классе. Представьте, что у нас есть метод, принимающий аргумент типа SAM. Если пользователь передаст в нем лямбда-функцию, то есть литерал функции вместо экземпляра типа SAM, компилятор 2.12 автоматически превратит этот объект функции в экземпляр типа SAM.

В заключение

В этой главе мы в общих чертах познакомились с конкурентным программированием и узнали, почему Scala удобно использовать для разработки конкурентных программ. Мы дали краткий обзор сведений, которые вы найдете в этой книге, и описали ее организацию. Наконец, мы отметили некоторые ключевые особенности Scala, знание которых пригодится при чтении последующих глав. Желающим узнать больше о последовательном программировании на языке Scala мы предлагаем прочитать книгу *Programming in Scala* Мартина Одерски (Martin Odersky), Лекса Спуна (Lex Spoon) и Билла Веннерса (Bill Venners), выпущенную издательством Artima Inc.

В следующей главе мы начнем изучение основ конкурентного программирования в JVM. Там вы познакомитесь с базовыми понятиями, низкоуровневыми утилитами, доступными в JVM, и моделью памяти в Java.

Упражнения

Цель упражнений, предлагаемых ниже, – дать вам возможность проверить свои знания языка программирования Scala. Они охватывают все, о чем рассказывалось в этой главе, а также некоторые дополнительные особенности Scala. Последние два упражнения подчеркивают различия между конкурентным и распределенным программированием. От вас требуется просто набросать схематическое решение на псевдокоде – не нужно писать полноценных программ на Scala.

1. Реализуйте метод `compose` со следующей сигнатурой:

   ```
   def compose[A, B, C]
   (g: B => C, f: A => B): A => C = ???
   ```

   Этот метод должен возвращать функцию `h`, являющуюся композицией функций `f` и `g`.
2. Реализуйте метод `fuse` со следующей сигнатурой:

   ```
   def fuse[A, B]
   (a: Option[A], b: Option[B]): Option[(A, B)] = ???
   ```

   Возвращаемый объект `Option` должен содержать кортеж значений из объектов `A` и `B` типа `Option`, если оба они непустые. Используйте `for`-генераторы.
3. Реализуйте метод `check`, принимающий множество значений типа `T` и функцию типа `T => Boolean`:

   ```
   def check[T](xs: Seq[T])(pred: T => Boolean): Boolean = ???
   ```

   Метод должен возвращать `true`, только если функция `pred` вернет `true` для всех значений в `xs`. Метод не должен возбуждать исключений. Попробуйте использовать метод `check`, как показано ниже:

   ```
   check(0 until 10)(40 / _ > 0)
   ```

Метод `check` имеет каррированное определение: вместо одного списка параметров он имеет два. Каррированные определения позволяют использовать более удобный синтаксис вызова, но в остальном они полностью эквивалентны определениям с одним списком параметров.

4. Измените класс `Pair`, представленный в этой главе, так, чтобы он мог использоваться в инструкциях сопоставления с шаблоном.

Познакомьтесь с поддержкой сопоставления с шаблонами в Scala, если вы этого еще не сделали.

5. Реализуйте функцию `permutations`, которая принимает строку и возвращает последовательность строк, которые являются вариантами лексикографической перестановки символов в исходной строке:

```
def permutations(x: String): Seq[String]
```

6. Реализуйте функцию `combinations`, которая принимает последовательность элементов и возвращает итератор для обхода всех возможных их комбинаций с длиной `n`. При составлении комбинации каждый элемент может извлекаться из коллекции только один раз, при этом порядок следования элементов не имеет значения. Например, для коллекции `Seq(1, 4, 9, 16)` можно составить такие комбинации с длиной 2: `Seq(1, 4)`, `Seq(1, 9)`, `Seq(1, 16)`, `Seq(4, 9)`, `Seq(4, 16)` и `Seq(9, 16)`. Функция `combinations` должна иметь следующую сигнатуру:

```
def combinations(n: Int, xs: Seq[Int]): Iterator[Seq[Int]]
```

   Прочитайте описание класса `Iterator` в документации к стандартной библиотеке.

7. Реализуйте метод, принимающий регулярное выражение и возвращающий частично примененную функцию, которая, в свою очередь, возвращает список совпадений, найденных в строке:

```
def matcher (regex: String): PartialFunction[String, List[String]]
```

   Частично примененная функция не должна определяться, если в строковом аргументе нет совпадений. Иначе она должна использовать регулярное выражение для вывода списка совпадений.

8. Представьте, что вы с тремя вашими коллегами работаете в кабинете, разделенном перегородками на отсеки. Вы не видите друг друга и не должны общаться голосом, потому что это может побеспокоить других сотрудников. Но вы можете перебрасываться записками. Из-за перегородок никто из вас не может быть уверен, что его записка попала к адресату. В любой момент вас или ваших коллег могут вызвать в кабинет к начальнику и задержать там на неопределенное время. Спроектируйте алгоритм, следуя которому вы и ваши коллеги смогли бы договориться о встрече в соседнем кафе. Кроме тех, кто задержится в кабинете у начальника, все должны собраться в кафе в одно и то же время. Что, если какая-то из записок не долетела до отсека одного из ваших коллег?

9. В дополнение к предыдущему упражнению представьте, что холле рядом с вашим кабинетом стоит доска whiteboard. Каждый из вас, проходя через холл, может что-то написать на ней, но нет никакой гарантии, что другие будут проходить через холл одновременно с вами.
   Решите задачу, предложенную в упражнении 8, но на этот раз с помощью доски.