Техники Java 7 и многоязычное программирование

Java

НОВОЕ ПОКОЛЕНИЕ РАЗРАБОТКИ

Бенджамин Эванс
Мартин Вербург

MANNING

ПИТЕР®

The Well-Grounded Java Developer

Vital Techniques of Java 7 and Polyglot Programming

BENJAMIN J. EVANS
MARTIJN VERBURG

MANNING
SHELTER ISLAND

Java

НОВОЕ ПОКОЛЕНИЕ РАЗРАБОТКИ

Техники Java 7 и многоязычное программирование

Бенджамин Эванс
Мартин Вербург

ПИТЕР®

Москва · Санкт-Петербург · Нижний Новгород · Воронеж
Ростов-на-Дону · Екатеринбург · Самара · Новосибирск
Киев · Харьков · Минск
2014

ББК 32.973.2-018.1
УДК 004.43
Э14

Эванс Б., Вербург М.

Э14 Java. Новое поколение разработки. — СПб.: Питер, 2014. — 560 с.: ил.

ISBN 978-5-496-00544-9

В этой книге представлен оригинальный и практичный взгляд на новые возможности Java 7 и новые языки для виртуальной машины Java (JVM), а также рассмотрены некоторые вспомогательные технологии, необходимые для создания Java-программ завтрашнего дня.

Книга начинается с подробного описания новшеств Java 7, таких как работа с ресурсами в блоке try (конструкция try-with-resources) и новый неблокирующий ввод-вывод (NIO.2). Далее вас ждет экспресс-обзор трех сравнительно новых языков для виртуальной машины Java — Groovy, Scala и Clojure. Вы увидите четкие понятные примеры, которые помогут вам ознакомиться с десятками удобных и практичных приемов. Вы изучите современные методы разработки, обеспечения параллелизма, производительности, а также многие другие интересные темы.

В этой книге:

— новые возможности Java 7;
— вводный курс по работе с языками Groovy, Scala и Clojure;
— обсуждение проблем многоядерной обработки и параллелизма;
— функциональное программирование на новых языках для JVM;
— современные подходы к тестированию, сборке и непрерывной интеграции.

12+ (В соответствии с Федеральным законом от 29 декабря 2010 г. № 436-ФЗ.)

ББК 32.973.2-018.1
УДК 004.43

ISBN 978-1617290060 англ.
ISBN 978-5-496-00544-9

Краткое содержание

Часть 3. Многоязычное программирование на виртуальной машине Java

Часть 4. Создание многоязычного проекта

Приложения

Оглавление

Часть 1. Разработка на Java 7

Часть 2. Необходимые технологии

Часть 3. Многоязычное программирование на виртуальной машине Java

Часть 4. Создание многоязычного проекта

Приложения

Вступление

«Кёрк мне сказал, что здесь на автозаправке продают пиво» — это были первые слова, которые я услышал от Бена Эванса (Ben Evans). Он приехал на Крит, чтобы поучаствовать в конференции Open Spaces Java. Я ответил, что на автозаправке обычно покупаю бензин, а за углом есть магазин, где торгуют пивом. Бен выглядел разочарованным. Действительно, я пять лет прожил на этом греческом острове и ни разу даже не подумал купить пиво на местной заправке «Бритиш Петролеум».

Примерно такое же чувство меня одолевало, когда я читал эту книгу. Я считаю себя одним из основателей языка Java. Вот уже 15 лет я программирую на Java, написал сотни статей, выступал с докладами на самых разных конференциях и вел сложные курсы по Java. А теперь, читая книгу Бена и Мартина, я не перестаю поражаться их идеям, о которых даже не задумывался. Книга начинается с рассказа о разработках, цель которых заключалась в изменении отдельных частей экосистемы Java. Изменить внутреннюю структуру библиотеки не так сложно, и тогда производительность при работе с определенными разновидностями входных данных действительно возрастает. В методе `Arrays.sort()` теперь используется алгоритм TimSort, а не MergeSort, как раньше. Если вы сортируете частично упорядоченный массив, то можете заметить небольшой рост производительности, совершенно не меняя вашего кода. Для изменения формата файла класса или дополнения виртуальной машины новой функцией требуется немало потрудиться. Бен это знает, ведь он участвует в работе исполнительного комитета JCP. Итак, эта книга — о языке Java 7. Здесь вы узнаете обо всех нововведениях, которыми обогатился язык. Это и улучшения, касающиеся синтаксического сахара, и `switch` на строках, и механизм «ветвление/слияние», и Java NIO.2.

Параллельная обработка? Это что-то связанное с `Thread` и `synchronized`, правильно? Если ваши знания о многопоточности ограничиваются этим фактом, то самое время подучиться. Как указывают авторы, «в настоящее время в области параллельной обработки ведется огромное множество исследований». Каждый день в почтовой рассылке, посвященной проблемам параллелизма, разворачиваются дискуссии, постоянно появляются новые идеи. Эта книга научит вас мыслить в духе «разделяй и властвуй». Кроме того, вы узнаете, как обходить определенные бреши в области безопасности.

Когда я прочел главу о загрузке классов, мне показалось, что авторы немного перегнули палку. Здесь были выложены на всеобщее обозрение приемы, которые мы с друзьями применяли для создания «волшебного» кода. Читай себе и учись! Авторы объясняют принципы работы `javap` — маленького инструмента, помогающего заглянуть в сущность байт-кода, генерируемого компилятором Java. Кроме того, они рассказывают о новом механизме `invokedynamic` и объясняют, чем он отличается от обычной рефлексии.

Одна из наиболее понравившихся мне глав — глава 6 под названием «Понятие о повышении производительности». Это первая книга со времен выхода труда Джека Ширази (Jack Shirazi) *Java Performance Tuning* («Настройка производительности в Java»), в которой по существу рассказывается, как заставить вашу систему работать быстрее. Эту главу можно охарактеризовать одной фразой: «Измеряйте, а не гадайте». К этому и сводится качественная оптимизация производительности. Человек не в состоянии угадать, какой код будет работать медленно. В этой главе не предлагается единственный программерский трюк — напротив, проблема производительности рассматривается на аппаратном уровне. Кроме того, здесь рассказано, *как* нужно измерять производительность. Интересный небольшой контрольный инструмент, описываемый в главе 6, — это класс `CacheTester`, отображающий затраты на кэш-промахи.

Третья часть книги посвящена многоязычному программированию на виртуальной машине Java (JVM). Java — это далеко не только язык программирования. Это еще и платформа, на которой могут работать другие языки. Мы уже были свидетелями взрывного распространения языков различных типов. Одни из них функциональные, другие — декларативные. Отдельные языки (например, Jython и JRuby) являются «портами», через которые другие языки могут работать на виртуальной машине Java. Языки могут быть динамическими (Groovy) или стабильными (Java и Scala). Существует много причин, по которым может потребоваться использовать на JVM не Java, а иной язык. Начиная новый проект, рассмотрите все имеющиеся варианты, прежде чем определяться с выбором. Возможно, вы избавитесь от необходимости написания большого количества шаблонного кода.

Бен и Мартин представляют нам три альтернативных языка: Groovy, Scala и Clojure. На мой взгляд, эти языки развиваются сейчас особенно динамично. Авторы описывают разницу между ними, сравнивают их с Java, рассматривают характерные черты. Главы, посвященные Groovy, Scala и Clojure, достаточно содержательны, чтобы вы могли выбрать, какой язык использовать, при этом авторы не слишком углубляются в технические детали. Например, вы не найдете здесь справочника по Groovy; вам просто подскажут, какой язык вам лучше всего подходит.

Далее вы узнаете, как в вашей системе можно организовать разработку через тестирование и непрерывную интеграцию. Мне было интересно узнать, что старый добрый «дворецкий» Hudson так быстро уступил место Jenkins. В любом случае эти инструменты очень пригодятся вам для управления проектом, равно как и Checkstyle вместе с Findbugs.

Прочитав эту книгу, вы станете по-настоящему основательным разработчиком Java. Более того, вы узнаете, что делать для сохранения этой основательности. Язык Java постоянно изменяется. Так, в следующей версии мы увидим лямбда-выражения и модуляризацию. Разрабатываются новые языки, обновляются конструкции, предназначенные для параллельной обработки. Многие вещи, которые верны сегодня, могут быть ошибочны в будущем. Вывод один: учиться, учиться и еще раз учиться!

Через какое-то время я снова ехал мимо автозаправки, на которой Бен хотел купить пива. Оказалось, что она закрылась, как и многие другие предприятия в кризисной Греции. Я так и не узнал, продавали ли они пиво.

Доктор Хайнц Кабуц (Heinz Kabutz).
Новостная расылка для специалистов по Java

Предисловие

Эта книга началась как сборник учебных заметок, написанных для новой группы молодых специалистов, принятых в IT-отдел по обслуживанию операций с иностранной валютой в составе «Дойче Банк». Один из нас (Бен), просмотревший имевшиеся в продаже книги, заметил, что на рынке не хватает актуального материала для начинающих Java-разработчиков. В результате он решил написать эту недостающую книгу.

С одобрения менеджеров IT-отдела «Дойче Банк», Бен отправился на конференцию Devoxx в Бельгию, чтобы почерпнуть идеи для работы над дополнительными темами. Там он познакомился с тремя программистами из IBM (это были Роб Николсон (Rob Nicholson), Зоя Слэттери (Zoe Slattery) и Холли Камминс (Holly Cummins)), которые помогли ему влиться в лондонское сообщество Java (LJC — London Java User Group).

В следующую субботу после этого состоялась ежегодная Открытая конференция, организованная LJC. Именно на ней Бен повстречал одного из лидеров LJC, Мартина Вербурга (Martijn Verburg). К вечеру этого дня на волне взаимной любви к учительству, техническим тусовкам и пиву они решили взяться за совместный проект, которому суждено было стать книгой «Java. Новое поколение разработки».

Надеемся, что в этой книге нам удалось убедительно представить разработку программ как вид социальной деятельности. Мы не преуменьшаем значения технической стороны этого искусства, но вместе с тем считаем, что не менее важны более тонкие проблемы, связанные с межличностной коммуникацией и другими гранями человеческого общения. Может быть, в книге и непросто рассказать об этих гранях, но вся наша работа проникнута данной темой.

Для того чтобы карьера разработчика могла состояться, ему необходимо непрестанно интересоваться новыми технологиями и с воодушевлением учиться. Мы надеемся, что в этой книге нам удалось подчеркнуть некоторые моменты, способные разжечь в человеке такую страсть. Это скорее экскурсионный тур, чем энциклопедическое исследование. Но мы хотели именно заинтересовать вас и помочь сделать первые шаги. А потом вы сможете продолжить изучение тех тем, которые вас наиболее заинтересуют.

В ходе работы над проектом акцент книги немного изменился. Она стала не просто вводным курсом для выпускников (хотя и эта задача в ней решена хорошо), а руководством для всех разработчиков Java, интересующихся: «А что изучить дальше? В каком направлении развиваться?» Покой нам только снится.

Мы расскажем вам о новых функциях Java 7, поговорим о наилучших современных методах разработки программ, а также о перспективах платформы. В книге мы подчеркнем несколько моментов, имеющих особое значение для нас с вами

как для Java-разработчиков. Параллельная обработка, производительность, байт-код и загрузка классов — вот основные технологии, которые мы считаем наиболее интересными. Кроме того, мы поговорим о новых языках, которые не являются Java, но работают на виртуальной машине Java (JVM). Такой принцип работы называется многоязычным программированием (polyglot programming). Эти темы мы обсудим потому, что в ближайшие годы они станут очень важны для многих разработчиков.

Прежде всего, это путешествие с заделом на будущее. В первую очередь мы пытались учесть ваши интересы. Мы считаем, что если вы станете хорошими Java-разработчиками, то не потеряете интереса к собственным проектам и сможете их контролировать. Таким образом, вы сможете подробнее изучить изменяющийся мир Java и экосистему, окружающую этот язык.

Мы верим, что квинтэссенция опыта, которую вы держите в руках, будет вам полезна и интересна. Надеемся также, что эта книга даст вам пищу для размышлений и читать ее будет не менее интересно, чем нам было работать над ней.

Благодарности

Согласно одной африканской пословице, «чтобы вырастить ребенка — нужна деревня»[1]. То же можно с полным правом сказать и о написании книги. Мы никогда бы не завершили этот труд без множества наших друзей, партнеров, коллег, товарищей. И даже оппонентов. Нам исключительно повезло, что большинство самых суровых из наших критиков также являются нашими друзьями.

Сложно вспомнить имена всех людей, посодействовавших нам в этом предприятии. Можете посетить сайт http://www.java7developer.com и найти там пост, анонсировавший выход оригинальной книги. Там есть большой список благодарностей. Не стоит забывать ни одного имени.

Если мы кого-то забыли (возможно, были пробелы в наших записях) — пожалуйста, примите наши извинения! Без особого порядка мы хотели бы поблагодарить всех перечисленных ниже людей за то, что эта книга смогла увидеть свет.

Лондонское Java-сообщество

Лондонское Java-сообщество, находящееся в Интернете по адресу www.meetup.com/londonjavacommunity, — это место, где мы встретились, место, ставшее огромной частью нашей жизни. Мы хотели бы поблагодарить следующих людей, которые помогли нам с рецензированием материала: Питера Бьюдо (Peter Budo), Ника Харкина (Nick Harkin), Джодева Девасси (Jodev Devassy), Крейга Силка (Craig Silk), Н. Вандервильдта (N. Vanderwildt), Адама Джея Мэркхама (Adam J. Markham), Розаллин (Rozallin), Дэниела Лемона (Daniel Lemon), Франка Аппиа (Frank Appiah), П. Франка (P. Franc), «Себкома» Правина (“Sebkom” Praveen), Динука Вирасингха (Dinuk Weerasinghe), Тима Мюррея Брауна (Tim Murray Brown), Луиса Мурбину (Luis Murbina), Ричарда Догерти (Richard Doherty), Рашула Хусейна (Rashul Hussain), Джона Стивенсона (John Stevenson), Джемму Силверс (Gemma Silvers), Кевина Райта (Kevin Wright), Аманду Уэйт (Amanda Waite), Джоэла Глюса (Joel Gluth), Ричарда Пола (Richard Paul), Колина Випурса (Colin Vipurs), Энтони Стаббза (Antony Stubbs), Майкла Джойса (Michael Joyce), Марка Хиндесса (Mark Hindess), Нуно (Nuno), Джона Поултона (Jon Poulton), Адриана Смита (Adrian Smith), Иоанниса Маврукакиса (Ioannis Mavroukakis), Криса Рида (Chris Reed), Мартина Скурлу (Martin Skurla), Сандро Манкьюзо (Sandro Mancuso) и Арула Дхезиазеелана (Arul Dhesiaseelan).

---

[1] Эта пословица стала широко известна в Америке после выхода книги Хиллари Клинтон (Hillary Clinton) «Нужна деревня» (It Takes A Village). — *Примеч. перев.*

Подробные консультации по другим языкам мы получили от Джеймса Кука (James Cook), Алекса Андерсона (Alex Anderson), Леонарда Аксельсона (Leonard Axelsson), Колина Хоува (Colin Howe), Брюса Дарлинга (Bruce Durling) и доктора Рассела Уиндера (Dr. Russel Winder). Им — наши особые благодарности.

Кроме того, отдельно благодарим Комитет сертифицированных разработчиков Java при LJC — Майка Баркера (Mike Barker), Тришу Джи (Trisha Gee), Джима Гафа (Jim Gough), Ричарда Уорбертона (Richard Warburton), Саймона Мейпла (Simon Maple), Сомая Нахала (Somay Nakhal) и Дэвида Иллсли (David Illsley).

Наконец, не можем не поблагодарить Барри Крэнфорда (Barry Cranford), основателя LJC, который начал это большое дело несколько лет назад. У него было всего несколько отважных соратников и мечта. Сегодня LJC объединяет около двух с половиной тысяч участников. Это общество породило множество других технических сообществ. LJC — настоящий краеугольный камень лондонского технического бомонда.

www.coderanch.com

С удовольствием благодарим Маниша Годбоула (Maneesh Godbole), Ульфа Дитмера (Ulf Ditmer), Дэвида О'Миру (David O'Meara), Деваку Курея (Devaka Cooray), Грега Чарльза (Greg Charles), Дипака Балу (Deepak Balu), Фреда Розенберга (Fred Rosenberger), Йеспера де Йонга (Jesper De Jong), Вутера Эта (Wouter Oet), Марка Сприцлера (Mark Spritzler) и Роэла де Нийса (Roel De Nijs) за их подробные комментарии и ценнные отзывы.

Manning publications

Благодарим Марджана Бейса (Marjan Bace) из издательства Manning за то, что он услышал двух молодых авторов, объединенных сумасшедшей идеей. В ходе подготовки этой книги нам доводилось общаться со многими сотрудниками издательства. Большое спасибо за неутомимую работу Рене Грегуару (Renae Gregoire), Карен Дж. Миллер (Karen G. Miller), Энди Кэрроллу (Andy Carroll), Элизабет Мартин (Elizabeth Martin), Мэри Пирджис (Mary Piergies), Деннису Далиннику (Dennis Dalinnik), Джанет Вейл (Janet Vail) и, несомненно, всем остальным, кто остался за кадром и кого мы позабыли. Без всех вас этот труд никогда бы не увенчался успехом!

Спасибо Кэндис Гиллхули (Candace Gillhoolley) за ее маркетинговую работу, а также Кристине Рудлофф (Christina Rudloff) и Морин Спенсер (Maureen Spencer) — за их поддержку по ходу дела.

Спасибо Джону Райану III (John Ryan III), который сделал подробную окончательную техническую рецензию рукописи незадолго до передачи книги в печать.

Спасибо следующим рецензентам, читавшим рукопись на разных этапах ее подготовки и оставившим ценные отзывы нашим редаторам и нам. Это Азиз Рахман (Aziz Rahman), Берт Бейтс (Bert Bates), Чед Дэвис (Chad Davis), Черил Джерозал (Cheryl Jerozal), Кристофер Хаупт (Christopher Haupt), Дэвид Стронг (David

Strong), Дипак Вохра (Deepak Vohra), Федерико Томассетти (Federico Tomassetti), Франко Ломбардо (Franco Lombardo), Джефф Шмидт (Jeff Schmidt), Джереми Андерсон (Jeremy Anderson), Джон Гриффин (John Griffin), Мацей Крефт (Maciej Kreft), Патрик Стегер (Patrick Steger), Пол Бенедикт (Paul Benedict), Рик Вагнер (Rick Wagner), Роберт Веннер (Robert Wenner), Родни Боллинджер (Rodney Bollinger), Сантош Шанбхаг (Santosh Shanbhag), Антти Койвисто (Antti Koivisto) и Стивен Харрисон (Stephen Harrison).

Особые благодарности

Благодарим Энди Бёрджесса (Andy Burgess) и изумительный сайт www.java7developer.com, а также Драгоса Догару (Dragos Dogaru), нашего невероятно классного стажера, который тестировал образцы кода по мере того, как мы создавали книгу.

Спасибо Мэтту Рэйблу (Matt Raible) за то, что он любезно позволил нам воспользоваться некоторыми материалами о выборе веб-фреймворка. Речь об этом пойдет в главе 13.

Спасибо Алану Бейтмену (Alan Bateman), руководителю разработки NIO.2 для Java 7. Без его консультаций этот великолепный API никогда бы не стал доступен разработчикам, ежедневно пишущим на Java.

Благодарим Джанну Боярски (Jeanne Boyarsky), которая любезно предложила свои услуги самого лучшего технического редактора, и, как и следовало ожидать, ничто не ускользнуло от ее острого критического взора. Спасибо, Джанна!

Спасибо Мартину Лингу (Martin Ling) за очень подробный рассказ о хронометражном оборудовании. Именно после консультаций с ним мы решили написать соответствующий раздел в главе 4.

Спасибо Джейсону Ван Зилу (Jason Van Zyl) за то, что он любезно разрешил воспользоваться некоторыми материалами из книги *Maven: The Complete Reference* (издательство Sonatype). Они приведены в главе 12.

Спасибо Кёрку Пеппердайну (Kirk Pepperdine) за его ценные замечания и комментарии по главе 6, а также за его дружбу и удивительное отношение к нашему ремеслу.

Спасибо доктору Хайнцу М. Кабуцу (Dr. Heinz M. Kabutz) за его великолепное предисловие и роскошное гостеприимство, оказанное нам на Крите, а также за работу над классным ресурсом Java Specialists' Newsletter (www.javaspecialists.eu/).

Благодарности Бена Эванса

В написании этой книги поучаствовало такое множество людей и такими разными способами, что мне, конечно, не хватит места, чтобы поблагодарить их всех. Мои особые благодарности — следующим людям:

- Берту Бейтсу (Bert Bates) и остальным сотрудникам Manning за то, что рассказали мне, чем рукопись отличается от книги;
- Мартину, конечно же, — за дружбу, за то, что вдохновлял меня продолжать писать в трудный час, и за многое другое;

- моей семье, в особенности обоим моим дедушкам — Джону Хинтону (John Hinton) и Джону Эвансу (John Evans), от которых я унаследовал так много личностных качеств;
- и наконец, И-Джею (E-J) (именно благодаря ему в этой книге так часто упоминаются выдры), а также Лиз, которые всегда прощали меня за «еще один вечер», потраченный на книгу. Я очень люблю их обоих.

Благодарности Мартина Вербурга

Спасибо моей маме Яннеке и моему папе Нико. Спасибо, что смогли заглянуть в будущее и принесли в дом компьютер Commodore 64, когда я и моя сестра были маленькими. Хотя большая часть компьютерного времени в нашей семье посвящалась Джампмену[1], в комплекте с тем компьютером к нам в дом попал и мануал по программированию, который возжег во мне страсть ко всему техническому. Папа научил меня, что если правильно справляться с мелкими делами, то крупные дела, складывающиеся из таких мелочей, начинают налаживаться сами. Я до сих пор придерживаюсь этой философии и при программировании, и вообще в жизни.

Спасибо моей сестре Ким за то, что мы еще в детские и юношеские годы писали код вместе. Я никогда не забуду, как давным-давно на экране высветилось то первое (очень медленное[2]) звездное небо — у меня на глазах свершилось волшебство! Спасибо моему свояку Йосу, который вдохновлял нас всех (не только потому, что он астрофизик, но и потому, что он классный!). Еще в этой книге засветилась моя суперплемяшка Гвинет. Интересно, вы сможете ее найти?

Бен — просто один из самых потрясающих технарей, с которыми мне доводилось встречаться за годы работы. Его техническая подкованность временами просто ошеломляет! Для меня было честью писать эту книгу вместе с ним; я определенно узнал о виртуальной машине Java гораздо больше, чем мог помыслить. Кроме того, Бен оказался отличным лидером в LJC, а также интереснейшим напарником-докладчиком на конференциях (так вышло, что теперь мы пользуемся не меньшей славой, чем какой-нибудь сценический дуэт разговорного жанра). Было просто прекрасно писать книгу вместе с другом.

И наконец, благодарю мою восхитительную жену Керри, стойко мирившуюся с тем, что очередной романтический вечер отменяется ради написания еще одной главы, и великодушно подготовившую все рисунки и скриншоты для книги, — как всегда, ты была неповторима. Ах, если бы кто-то мог познать в жизни такую же удивительную любовь и поддержку, какую дала мне она!

---

[1] Классная, очень классная игра-платформер! Я смеялся до колик, наблюдая, как мама налегает на джойстик!

[2] Скажем так, на тот момент настройка производительности не была моей сильной стороной.

Об этой книге

Мы рады приветствовать вас на страницах книги «Java. Новое поколение разработки». Она призвана превратить вас в современного Java-разработчика, вновь наполнить вас тягой к изучению этого языка и платформы. В ходе повествования вы узнаете новые функции Java 7, закрепите (или приобретете и закрепите) знания современных технологий, таких как внедрение зависимостей, разработка через тестирование и непрерывная интеграция, и приступите к исследованию дивного нового мира JVM, где Java сосуществует с другими языками.

Для начала обратимся к описанию языка Java, предложенному Джеймсом Ири (James Iry) в его чудесной статье «Краткая, неполная и в основном неверная история языков программирования»[1]:

«1996 — Джеймс Гослинг изобретает Java. Это относительно читабельный, поддерживающий сборку мусора, основанный на классах, статически типизированный, объектно-ориентированный язык без множественного наследования реализаций, но с множественным наследованием интерфейсов. Sun громогласно возвещает о новизне Java».

Хотя вся хохма с языком Java в этой статье заключается в том, что ниже автор дает такое же описание языку C# (за исключением последнего предложения), сами определения довольно меткие. В статье содержится еще несколько замечательных характеристик, очень рекомендуем полностью прочитать ее на досуге.

Но возникает довольно серьезный вопрос. Зачем же мы пишем новую книгу о языке, которому уже 16 лет? Разумеется, он уже стабилен. Неужели можно сказать о нем достаточно много нового и интересного?

Если бы все так и было, то у нас получилась бы очень короткая книга. Но мы продолжаем говорить о Java, так как одной из самых сильных сторон этого языка была способность работать на немногочисленных основных дизайнерских решениях, которые зарекомендовали себя как очень успешные:

- автоматическое управление средой времени управления (например, сборка мусора, динамическая компиляция);
- простой синтаксис и относительно небольшое количество концепций в основе языка;
- консервативный подход к развитию языка;
- добавление нового функционала и усложнений в библиотеках;
- широкая и открытая экосистема.

---

[1] Полный русский перевод статьи можно найти по адресу http://habrahabr.ru/post/165093/. — *Примеч. перев.*

Такие дизайнерские решения позволили сохранить в мире Java инновационные процессы. Благодаря сравнительно простому ядру порог для вступления в сообщество разработчиков остался низким, а широкая экосистема дала новичкам возможность легко находить готовые компоненты, подходящие под их нужды.

Эти черты позволили языку и платформе **Java сохранить силу и динамичность** — даже притом, что исторически язык менялся медленно. Такая тенденция сохранилась и в **Java 7. Изменения в языке являются эволюционными, а не революционными**. Серьезное отличие от предыдущих версий заключается в том, что Java 7 еще до выпуска создавался как плацдарм для разработки новой версии. В Java 7 выполнена основная работа по подготовке крупных языковых изменений, запланированных для Java 8, так как в ходе релизов компания Oracle придерживается стратегии «План B».

Еще одним крупным изменением последних лет стал бурный рост языков, не являющихся Java, но работающих на виртуальной машине Java (JVM). В результате началось перекрестное опыление между Java и другими языками JVM. В настоящее время создается множество проектов (со временем их становится все больше), работающих полностью на JVM и включающих Java как один из используемых языков.

Разработка многоязычного проекта, включающего, в частности, такие языки, как Groovy, Scala и Clojure, очень важна в контексте современной экосистемы Java. Этой теме посвящена последняя часть нашей книги.

Как работать с этой книгой

В принципе, приведенный здесь материал рассчитан для изучения от начала до конца. Но мы понимаем, что некоторые читатели сразу захотят перейти к конкретным темам, и частично адаптировали книгу и к такому способу чтения.

Мы убеждены в полезности практического обучения, поэтому рекомендуем вам обязательно опробовать код, сопровождающий материал, по мере чтения текста. Далее в этом разделе мы объясним, как лучше читать книгу, если вы предпочитаете изучить лишь избранные главы.

Книга «Java. Новое поколение разработки» разделена на четыре части:

- «Разработка на Java 7»;
- «Необходимые технологии»;
- «Многоязычное программирование на виртуальной машине Java»;
- «Создание многоязычного проекта».

В первой части содержится две главы о Java 7. Во всей книге используется синтаксис и семантика Java 7, поэтому главу 1 прочитать необходимо. Глава 2 под названием «Новый ввод-вывод» будет особенно интересна специалистам, работающим с файлами, файловыми системами и сетевым вводом-выводом.

Во второй части собраны четыре главы (3–6), посвященные таким темам, как: внедрение зависимостей, современная параллельная обработка, файлы классов/байт-код и настройка производительности.

В третьей части (главы 7–10) мы поговорим о многоязычном программировании на виртуальной машине Java. Главу 7 прочитать необходимо, так как в ней обсуждается категоризация и использование альтернативных языков на JVM. В следующих трех главах мы рассмотрим сначала Groovy (язык, похожий на Java), далее — гибридный язык Scala, совмещающий объектно-ориентированные и функциональные черты, и, наконец, полностью функциональный язык (Clojure). Главы, посвященные этим языкам, можно читать отдельно друг от друга, хотя если вы не знакомы с функциональным программированием, то лучше прочитать их по порядку.

В четвертой части (последние четыре главы) вводится новый материал, который, однако, базируется на темах, рассмотренных ранее. Хотя эти главы можно читать и в отрыве от остальной книги, в некоторых из них мы предполагали, что вы уже знакомы с вышеизложенным материалом и/или ориентируетесь в определенных темах.

Одним словом, глава 1 необходима для понимания всей книги. Глава 7 нужна для понимания части 3. Другие главы можно читать по порядку или в отдельности друг от друга. Но в более поздних главах будут попадаться разделы, предполагающие знание материала, изложенного выше.

Для кого предназначена книга

Основной целевой аудиторией этой книги являются разработчики на Java, желающие усовершенствовать свои знания как о языке, так и о платформе. Если вы хотите идти в ногу со всеми новинками, предлагаемыми в Java 7, то эта книга — для вас.

Если вы хотите освежить свои знания по определенным технологиям и разобраться в таких темах, как внедрение зависимостей, параллельная (конкурентная) обработка и разработка через тестирование, то в нашей книге вы приобретете хорошие базовые знания по этим темам.

Кроме того, издание ориентировано на тех разработчиков, которых интересует развитие многоязычного программирования, тех, кто желает пойти по этой дороге. В частности, если вы хотите обучиться функциональному программированию, то наши главы о языках (особенно о Java и Clojure) очень вам пригодятся.

Дорожная карта

В первой части — всего две главы. Первая глава представляет собой введение в язык Java 7 со всем множеством его мелких функций, оптимизирующих производительность. Сумма всех этих расширений известна под названием «Монета» (или «проект “Монета”»). В главе 2 мы знакомим вас с новыми API ввода-вывода, в частности с полностью модернизированной системой поддержки файловой системы, новыми возможностями асинхронного ввода-вывода и многими другими темами.

В части 2 содержатся четыре главы о необходимых на практике технологиях. Так, в главе 3 мы подробно рассказываем, как в промышленных масштабах стала

использоваться технология внедрения зависимостей. Далее показано стандартизированное решение на языке Java с применением Guice 3. В главе 4 объясняется, как правильно организовать в Java современную параллельную обработку. Эта тема начнет вызывать все больший интерес по мере того, как производство аппаратного обеспечения будет уверенно двигаться в направлении многоядерных процессоров. В главе 5 рассмотрены файлы классов и работа с байт-кодом на виртуальной машине Java. Мы срываем завесу таинственности, окружающую эту тему, и помогаем понять, почему Java работает именно так, а не иначе. В главе 6 дается вводная информация о настройке производительности в ваших приложениях на языке Java. Эта глава помогает разобраться и в таких конкретных темах, как сборка мусора.

Часть 3 посвящена многоязычному программированию на виртуальной машине Java и состоит из четырех глав. В главе 7 начинается рассказ о многоязычном программировании и приводится контекст, показывающий, почему бывает уместно и важно задействовать в проекте другой язык. В главе 8 мы знакомим вас с Groovy — динамическим языком, похожим на Java. Groovy доказывает, что динамический язык, синтаксически схожий с Java, может значительно повысить продуктивность Java-разработчика. В главе 9 мы окунемся в гибридный мир Scala — отчасти объектно-ориентированный, отчасти функциональный. Scala — это очень мощный и лаконичный язык. Глава 10 рассчитана на любителей языка Lisp. Рассматриваемый здесь язык Clojure широко известен как «Lisp с человеческим лицом». В этой главе вы сможете в полной мере оценить потенциал функционального языка на виртуальной машине Java.

В части 4 мы опираемся на материал, изложенный в предыдущих главах, и рассматриваем многоязычные технологии в нескольких областях разработки программного обеспечения. В главе 11 мы затрагиваем разработку через тестирование и рассказываем о методологии работы с подставными объектами (mock objects), а также даем некоторые практические советы. В главе 12 делается обзор двух распространенных инструментов, предназначенных для организации конвейерной сборки (Maven 3) и непрерывной интеграции (Jenkins/Hudson). В главе 13 мы поговорим об ускоренной веб-разработке и обсудим, почему язык Java традиционно отстает в этой области. Здесь мы предложим пару новых технологий для прототипирования (Grails и Compojure). В главе 14 подводится итог и делаются предположения о будущем Java. В частности, мы поговорим о поддержке функциональных подходов, которая появится в Java 8.

Соглашения в коде и материал для скачивания

Для того чтобы начать работу, нужно скачать и установить Java 7. Просто следуйте инструкциям по скачиванию и установке двоичного файла (выберите файл, подходящий для вашей операционной системы). Двоичные файлы и соответствующие инструкции находятся на сайте Oracle, посвященном стандартной версии Java (Java SE): www.oracle.com/technetwork/java/javase/downloads/index.html.

Вся прочая необходимая информация изложена в приложении A, где подробно рассказано, как устанавливать и запускать исходный код.

Весь исходный код в книге записан `примерно таким моноширинным шрифтом`. Этим он отличается от окружающего текста. Во многих листингах мы аннотируем код, чтобы подчеркнуть ключевые моменты. В тексте используются нумерованные списки, в которых сообщается дополнительная информация о коде. Мы старались форматировать код так, чтобы он укладывался в рамки страницы. Для этого мы пользовались разбиением длинных строк и аккуратно добавляли отступы. Но иногда все же встречаются слишком длинные строки — в них мы используем символы продолжения строки.

Исходный код для всех примеров из книги доступен по адресу www.manning.com/TheWell-GroundedJavaDeveloper. Примеры кода встречаются по всей книге. Сравнительно длинные листинги имеют заголовки; краткие листинги идут прямо в тексте.

Требования к программному обеспечению

Java 7 работает практически на всех современных платформах. Вы сможете запускать примеры из книги, если работаете в одной из следующих операционных систем:

- MS Windows XP или выше;
- новые версии *nix;
- Mac OS X 10.6 и выше.

Большинство читателей, вероятно, попробуют использовать примеры кода в IDE (интегрированной среде разработки). Java 7 и последние версии Groovy, Scala и Clojure хорошо поддерживаются в следующих версиях основных IDE:

- Eclipse 3.7.1 и выше;
- NetBeans 7.0.1 и выше;
- IntelliJ 10.5.2 и выше.

При создании и запуске примеров мы пользовались NetBeans 7.1 и Eclipse 3.7.1.

Об авторах

Бен Эванс — организатор в LJC (Лондонское сообщество Java-разработчиков, Лондонская группа пользователей Java) и член Java Community Process Executive Committee. Он участвует в определении стандартов для экосистемы Java. На протяжении «интересных времен» Бен много и активно работал в технологической сфере, а в настоящее время занимает пост CEO (главного исполнительного директора) в софтверной компании, специализирующейся на программировании на Java и работающей в основном в финансовой сфере. Бен часто читает лекции по таким темам, как платформа Java, производительность и параллельная обработка.

Мартин Вербург (CTO, jClarity) более 10 лет профессионально работает в технологической сфере и является консультантом по свободному ПО в самых разных компаниях — от стартапов до крупных предприятий. Мартин — один из руководителей Лондонской группы пользователей Java. Он координирует глобальные разработки членов «Групп пользователей Java», занимающихся подготовкой документов JSR (запросов на спецификацию Java), в рамках программы JSR, а также участвует в деятельности OpenJDK (программа OpenJDK).

Поскольку Мартин — признанный эксперт по оптимизации работы технологических команд, его доклады и презентации пользуются успехом на крупных конференциях (JavaOne, Devoxx, OSCON, FOSDEM и т. д.). В этой среде он известен как «Разработчик от Дьявола» за постоянную конструктивную критику статус-кво, сложившегося в современной софтверной индустрии.

Иллюстрация на обложке

Рисунок на обложке книги называется «Цветочница». Иллюстрация взята из вышедшего в XIX веке четырехтомного компендиума национальных костюмов под авторством Сильвена Марешаля (Sylvain Mare´chal). Этот сборник вышел во Франции. Каждая из иллюстраций аккуратно отрисована и раскрашена вручную. Цветистая коллекция Марешаля живо напоминает нам, насколько разными и разобщенными были города и регионы нашего мира еще каких-то 200 лет назад. Люди, жившие изолированными группами, говорили на разных диалектах и языках. На улицах города и в сельской местности было легко понять, откуда родом прохожий, каковы род его занятий и положение в жизни. Обо всем рассказывало платье.

С тех пор дресс-коды изменились. Региональное разнообразие в одежде, царившее в те времена, сейчас стерлось. Сейчас сложно отличить друг от друга даже жителей разных континентов, не говоря уже об уроженцах различных городов и регионов. Вероятно, мы обменяли культурное разнообразие на более глубокую и интересную личную жизнь — и, несомненно, гораздо более быстрый и разносторонний технологический прогресс.

В век, когда одну книгу по программированию сложно отличить от другой, мы приветствуем инициативность и изобретательность в софтверной индустрии, показывая на обложках наших изданий разнообразные региональные костюмы двухвековой давности. В этом нам помогает собрание Марешаля.

От издательства

Ваши замечания, предложения и вопросы отправляйте по адресу электронной почты vinitski@minsk.piter.com (издательство «Питер», компьютерная редакция).

Мы будем рады узнать ваше мнение!

На сайте издательства http://www.piter.com вы найдете подробную информацию о наших книгах.

ЧАСТЬ 1

Разработка на Java 7

В первых двух главах мы поговорим о том, как можно вырасти профессионально, вооружившись Java 7. В разминочной первой главе мы рассмотрим небольшие синтаксические изменения, которые помогут вам работать более продуктивно. О каждом из этих изменений можно сказать: «мал золотник, да дорог». В ходе работы мы подготовим почву для обсуждения более крупной темы из этой части — нового механизма ввода-вывода в Java, которому посвящена вторая глава.

Основательный Java-разработчик, несомненно, должен знать о новейших функциях, появляющихся в языке. В Java 7 есть несколько таких функций, значительно облегчающих жизнь разработчика-практика. Но понять синтаксис этих изменений, возможно, будет непросто. Чтобы *быстро* писать *эффективный* и *надежный* код, нужно глубоко понимать, как и почему были реализованы эти новые функции. Изменения Java 7 делятся на два больших множества: проект «Монета» (Project Coin) и NIO.2.

Первое множество — проект «Монета» — это несколько небольших изменений на уровне языка. Они призваны повысить продуктивность труда разработчика, не оказывая значительного влияния на базовую платформу. К таким изменениям относятся:

- конструкция `try-with-resources` (автоматически закрывающая ресурсы);
- строки в `switch`;
- усовершенствованные числовые литералы;
- множественный `catch` (объявление нескольких исключений в блоке `catch`);
- ромбовидный синтаксис (требующий меньше шаблонного кода при работе с обобщенными типами).

Все эти изменения могут показаться незначительными. Но если исследовать семантику, которая скрывается за изменениями синтаксиса, можно лучше понять, чем язык Java отличается от платформы Java.

Второе множество изменений — это новый API ввода-вывода (NIO.2). Он кардинально модернизирует поддержку файловой системы Java, а также предоставляет новые мощные возможности асинхронной обработки. К важнейшим изменениям относятся:

- новая конструкция `Path` для ссылки на файлы и файлоподобные сущности;
- вспомогательный класс `Files`, упрощающий создание, копирование, перемещение и удаление файлов;
- встроенная навигация по дереву каталогов;
- асинхронный ввод-вывод, основанный на обратных вызовах и использовании конструкций `future`; позволяет осуществлять в фоновом режиме крупные операции ввода-вывода.

Закончив изучение первой части, вы сможете естественно думать и писать на языке Java 7. Эти новые знания будут закреплены по ходу чтения книги, поскольку все функции Java 7 используются и в последующих главах.

1 Введение в Java 7

В этой главе:

- Java как платформа и как язык;
- небольшие, но мощные синтаксические изменения;
- оператор `try-with-resources`;
- усовершенствованная обработка исключений.

Добро пожаловать в Java 7. Положение дел в этом языке может показаться вам немного непривычным. И это хорошо — так много всего предстоит исследовать теперь, когда шумиха вокруг новой версии утихла и Java 7 раскочегарился на полную мощность. Дочитав эту книгу, вы сделаете только первые шаги в огромный мир — мир новых функций, софтверного искусства, а также в мир других языков, действующих на виртуальной машине Java (JVM).

В качестве разминки мы плавно введем вас в курс Java 7, но уже на этом этапе познакомим вас с некоторыми мощными возможностями. Для начала объясним отличие, которое зачастую неправильно понимается: поговорим об отличиях языка и платформы.

Затем мы рассмотрим проект «Монета» — набор небольших, но эффективных нововведений, появившихся в Java 7. Мы расскажем, как построен процесс одобрения, внедрения и выпуска изменений на платформе Java. Обсудив этот процесс, мы перейдем к рассмотрению шести основных новых функций, появившихся в рамках проекта «Монета».

Вы изучите новый синтаксис, в частности новый способ обработки исключений (множественный `catch`), а также работу с конструкцией `try-with-resources` (использование ресурсов в блоке `try`). Эта конструкция помогает избегать ошибок в коде, работающем с файлами или другими ресурсами. Дочитав эту главу, вы сможете по-новому писать на Java и будете в полной боевой готовности для изучения масштабных тем, ожидающих впереди.

Итак, приступим к делу и поговорим о дуализме языка и платформы — пожалуй, это центральный аспект современного языка Java. Это важнейший вопрос, к которому мы многократно будем возвращаться на протяжении всей книги, поэтому с ним просто необходимо разобраться.

1.1. Язык и платформа

Важнейшая проблема, с которой мы начнем наш разговор, — это разница между языком Java и платформой Java. Удивительно, но разные авторы по-разному определяют и феномен языка, и феномен платформы. Из-за этого может возникать неясность и некоторая путаница и в том, чем отличаются язык и платформа, и в том, к чему относятся те или иные программные функции, используемые в коде приложения.

Четко очертим эти различия прямо сейчас, так как эта разница затрагивает суть самых разных тем из нашей книги. Итак, вот эти определения.

- *Язык Java* — это статически типизированный объектно-ориентированный язык, над которым мы немного пошутили в разделе «Об этой книге». Надеемся, что вы уже довольно хорошо знакомы с ним. Одно из самых очевидных качеств языка Java заключается в том, что он пригоден для чтения человеком (или, по крайней мере, должен таким быть!).
- *Платформа Java* — это программное обеспечение, предоставляющее нам среду времени исполнения. Это виртуальная машина Java (JVM), линкующая и выполняющая ваш код в том виде, в каком он ей предоставляется. Код предоставляется в виде файлов классов, непригодных для чтения человеком. Иными словами, машина не интерпретирует непосредственно файлы с исходным кодом на языке Java, а требует предварительного преобразования этого кода в файлы классов.

Одна из основных причин успеха Java как системы ПО заключается в ее стандартизации. Это означает, что Java имеет спецификации, описывающие, как должна работать платформа. Стандартизация позволяет различным производителям и участникам разнообразных проектов создавать реализации, которые теоретически должны работать одинаково. Такие спецификации не гарантируют, насколько высока будет производительность конкретной реализации одной конкретной задачи, но вполне гарантируют правильность результатов.

Существует несколько отдельных спецификаций, управляющих системой Java. Самые важные из них — это спецификация языка Java (JLS) и спецификация виртуальной машины Java (VMSpec). В версии Java 7 это разделение соблюдается очень строго; на самом деле VMSpec уже нигде не ссылается на JLS. Если вы усматриваете в этом признак того, насколько серьезно в Java 7 поставлена работа с исходными языками, не являющимися Java, то нам нравится ход ваших мыслей, продолжайте в том же духе. Ниже мы подробнее обсудим разницу между двумя этими спецификациями.

При внимательном изучении такого дуализма напрашивается вопрос: «А какова же связь между языком и платформой?» Если в Java 7 они настолько разделены, то как они стыкуются и образуют общую систему Java?

Связь между языком и платформой заключается в совместном использовании файлов классов (файлов в формате `.class`). Рекомендуем серьезно изучить определение файлов классов. Эти сведения вам точно не помешают, а знание этой темы — один из способов, позволяющих хорошему Java-программисту стать выдающимся. На рис. 1.1 показан полный процесс создания и использования кода Java.

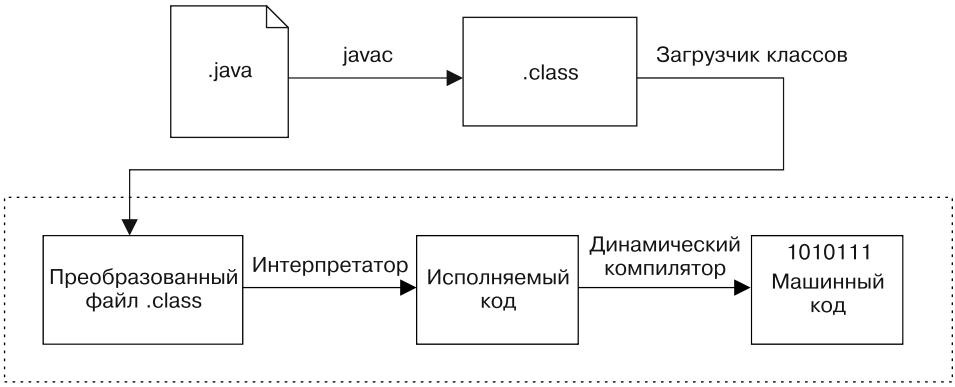


Рис. 1.1. Исходный код Java преобразуется в файлы .class, затем с ним производятся манипуляции в период загрузки, после чего он подвергается динамической компиляции

Как понятно из рисунка, код Java начинается в виде исходного кода, написанного программистом и пригодного для чтения человеком. После этого `javac` компилирует его в файл `.class`. Затем эта информация загружается в виртуальную машину Java. Обратите внимание, что манипуляции с классами и их изменение зачастую осуществляются в ходе процесса загрузки. Многие популярные фреймворки (особенно те, в названии которых присутствует слово Enterprise) преобразуют классы в ходе загрузки.

JAVA — ЭТО КОМПИЛИРУЕМЫЙ ИЛИ ИНТЕРПРЕТИРУЕМЫЙ ЯЗЫК?

Обычно Java видится разработчику как язык, компилируемый в файлы `.class` до того, как код будет использоваться на JVM. Призадумавшись, многие разработчики также объясняют, что работа байт-кода начинается с интерпретации виртуальной машиной JVM, но несколько позже код также проходит динамическую компиляцию (JIT). Но на этом многие специалисты начинают «плыть», выстраивая несколько надуманную концепцию о том, что байт-код, в сущности, является машинным кодом для воображаемого или упрощенного процессора.

На самом деле байт-код виртуальной машины Java можно считать переходной формой между исходным кодом, пригодным для чтения человеком, и машинным кодом. В техническом отношении с точки зрения теории компиляции байт-код — это действительно своеобразный промежуточный язык (intermediate language), а не настоящий машинный код. Это означает, что процесс преобразования исходного кода Java в байт-код не является компиляцией в том смысле, в каком она понимается в языках C и C++. В свою очередь, `javac` не назовешь таким же компилятором, как `gcc`. В сущности, это генератор файлов классов для обработки исходного кода Java. Настоящим компилятором в экосистеме Java является динамический компилятор (JIT) (см. рис. 1.1).

Некоторые специалисты характеризуют систему Java как «динамически компилируемую». При этом акцентируется тот факт, что истинная компиляция — это динамическая компиляция во время исполнения, а не создание файла класса во время исполнения.

Поэтому верным ответом на вопрос «Java — это компилируемый или интерпретируемый язык?» будет: «И такой и такой».

Итак, когда мы немного разъяснили разницу между языком и платформой, поговорим о некоторых заметных изменениях синтаксиса языка, появившихся в Java 7. Начнем с небольших синтаксических перемен, объединенных в рамках проекта «Монета».

1.2. Малое прекрасно — расширения языка Java, или Проект «Монета»

Проект «Монета» (Project Coin) — это свободный проект, разрабатываемый в рамках подготовки Java 7 (и 8) с 2009 года. В этом разделе мы объясним, как подбирались функции и как процесс эволюции языка происходит на уровне небольших изменений, объединенных проектом «Монета». Этот раздел будет построен в форме ситуационного исследования (case study).

ПРИЧЕМ ТУТ «МОНЕТА»

Цель проекта «Монета» — постепенно вносить в язык Java небольшие изменения. Это название в английском языке связано с игрой слов, так как существительное coin означает «монета», а глагол to coin — «чеканить». Это же слово может означать «выдумать», например «выдумать фразу». Именно в таком значении «Монета» понимается здесь — проект «чеканит» новые выражения, добавляя их в язык.

Подобные языковые игры, фантазии и вездесущие несносные каламбуры просто наводнили техническую культуру. Хотя вы, вероятно, к этому уже привыкли.

Мы считаем, что важно рассказать не только о том, *что* изменилось в языке, но и о том, *почему* появилось такое изменение. В ходе разработки Java 7 вокруг новой версии языка творилось множество всего интересного. Но члены сообщества не всегда и не вполне представляли, сколько работы потребуется на полную разработку изменений — до такой степени готовности, чтобы их можно было показать людям. Мы надеемся, что сможем пролить немного света на эту проблему, а также развеять некоторые мифы. Но если вы не очень интересуетесь тем, как развивается язык Java, можете переходить к разделу 1.3 — там мы непосредственно обсуждаем конкретные изменения в языке.

В изменении языка Java прослеживается определенная кривая усилий. Реализация некоторых языковых деталей требует значительно меньшего количества разработок, чем реализация других. На рис. 1.2 мы попытались представить различные линии разработки и относительное количество усилий, требуемое в каждом конкретном случае. Мы расположили эти линии в порядке возрастания необходимых усилий.

В принципе, лучше идти по пути наименьшего сопротивления. Это означает, что если существует возможность реализовать новую функцию как библиотеку, то так и стоит поступить. Но не все функции настолько просты, чтобы их можно было реализовывать в виде библиотек либо новых возможностей интегрированной среды разработки. При реализации некоторые функции должны быть глубже укоренены в платформе.

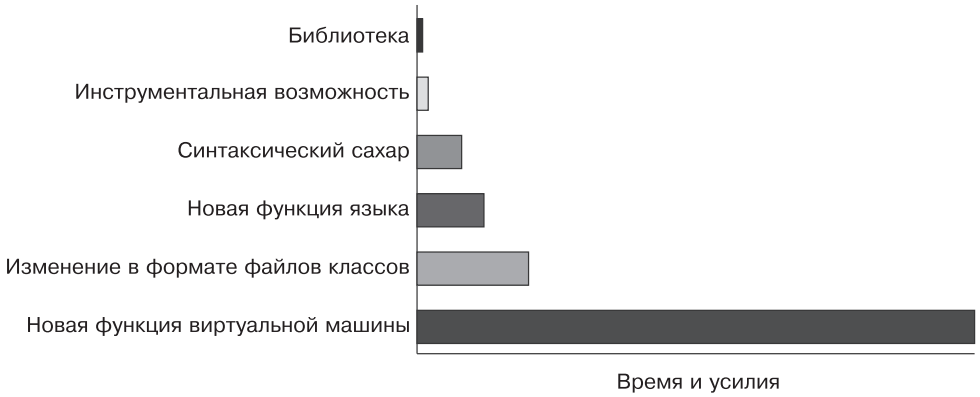


Рис. 1.2. Относительная сложность усилий при реализации новой функциональности различными способами

Ниже перечислены функции (в основном из Java 7), которые иллюстрируют элементы приведенной выше шкалы:

- *синтаксический сахар* — символы подчеркивания в числах (Java 7);
- *небольшая новая языковая функция* — конструкция `try-with-resources` (Java 7);
- *изменение формата файлов классов* — аннотации (Java 5);
- *новая функция виртуальной машины Java* — `invokedynamic` (Java 7).

СИНТАКСИЧЕСКИЙ САХАР

Некоторые функции языка называются выражением «синтаксический сахар» (syntactic sugar). Это означает, что такая синтаксическая форма избыточна — она уже есть в языке, — но синтаксический сахар существует потому, что человеку с ним легче работать.

Действует общее правило, согласно которому функции, являющиеся синтаксическим сахаром, удаляются из представления программы для компилятора на ранней стадии процесса компиляции. Принято говорить об «обессахаривании» кода для получения базового представления конкретной функции.

Внедрять в язык изменения на уровне синтаксического сахара сравнительно легко, так как для этого требуется выполнить небольшую работу. Изменения вносятся только на уровне компилятора (в случае с Java это `javac`).

Проект «Монета» (а также оставшаяся часть этой главы) посвящен переменам, расположенным в диапазоне от синтаксического сахара до небольших изменений языка.

Первая серия предложений по изменениям в рамках «Монеты» высказывалась на рассылке разработчиков «Монеты» в период с февраля по март 2009 года. Было внесено почти 70 предложений, демонстрирующих, насколько широк диапазон возможных оптимизаций. Было даже сделано шутливое предложение добавить многострочные строки в стиле «лолкотов» — юмористических надписей на фотографиях с котами, которые бывают смешными или немного издевательскими, — см. http://icanhascheezburger.com/.

Предложения «Монеты» оценивались в соответствии с совершенно простым набором правил. Автор предложения должен был сделать три вещи:

- подать подробное правильно оформленное предложение, описывающее предлагаемое изменение (принципиально это должно быть изменение языка Java, а не изменение виртуальной машины Java);
- открыто обсуждать сделанное предложение в рассылке и учитывать конструктивную критику от других участников;
- быть готовым предоставить прототипный набор заплаток (патчей), которые позволили бы реализовать предложенное изменение.

На примере проекта «Монета» хорошо видно, как язык и платформа могут развиваться в будущем: изменения обсуждаются открыто, происходит раннее прототипирование функций, работа ведется в коллективном режиме.

На данном этапе вы можете задать вопрос: «А как выглядит такое маленькое изменение в спецификации?» Одно из таких изменений, которое мы рассмотрим ниже, связано с добавлением единственного слова — `String` — в раздел 14.11 спецификации JLS. Сложно представить себе более мелкое изменение, но даже оно затрагивает сразу несколько частей спецификации.

JAVA 7 — ПЕРВАЯ ВЕРСИЯ, КОТОРАЯ РАЗРАБАТЫВАЛАСЬ В СТИЛЕ СВОБОДНОГО ПО

Java не всегда был «свободным» языком программирования. Но после соответствующего объявления, сделанного на конференции JavaOne в 2006 году, исходный код языка Java был выпущен по лицензии GPLv2 (за исключением отдельных фрагментов, исходного кода которых у Sun не было). Это случилось примерно на этапе выхода Java 6, поэтому Java 7 — первая версия языка, которая разрабатывалась по лицензии свободного программного обеспечения (OSS). При создании платформы Java в свободном режиме основное внимание было приковано к проекту OpenJDK.

В рассылках, таких как coin-dev, lambda-dev и mlvm-dev, велись развернутые дискуссии о функциях, которые могут быть добавлены в будущем. Таким образом, широкое сообщество разработчиков могло совместно заниматься созданием Java 7. На самом деле мы помогаем вести программу Adopt OpenJDK, чтобы сориентировать других разработчиков, пока мало знакомых с OpenJDK, — так мы помогаем улучшать и сам язык Java. Посетите страницу http://java.net/projects/jugs/pages/AdoptOpenJDK, может быть, вы захотите к нам присоединиться.

Изменения всегда приводят к последствиям, которые потом сказываются на всей структуре языка.

Вот действия, которые следует осуществить (или как минимум проверить их необходимость) при *любом* изменении:

- обновить JLS;
- реализовать прототип в компиляторе исходного кода;
- добавить библиотечную поддержку, необходимую для данного изменения;
- написать тесты и примеры;
- обновить документацию.

Кроме того, если изменение касается виртуальной машины или платформы, то следует:

- обновить спецификацию VMSpec;
- реализовать изменения виртуальной машины;
- добавить поддержку в файл класса и инструментарий виртуальной машины;
- учесть воздействие изменения на отражение (reflection);
- учесть воздействие изменения на сериализацию;
- обдумать любые воздействия на компоненты, работающие с нативным кодом, такие как нативный интерфейс Java (JNI).

Как видите, работы немало, и это уже после того, как будет учтено воздействие изменения на всю спецификацию языка!

Когда дело доходит до внесения изменений, особенно щекотливые ситуации возникают с системой типов. Не потому, что система типов в Java плоха. Дело в том, что языки с богатой системой статических типов обычно имеют множество возможных точек взаимодействия между различными компонентами таких систем типов. Поэтому любые изменения в системе типов могут приводить к неожиданным изменениям.

В рамках проекта «Монета» была избрана очень разумная стратегия решения таких проблем: участникам проекта рекомендуется не затрагивать систему типов при предложении изменений. Учитывая, какой объем работы требуется для внесения даже малейших изменений, такой подход оказался прагматичным.

Итак, мы немного познакомились с основами проекта «Монета», теперь рассмотрим отдельные функции, которые были отобраны для включения в язык.

1.3. Изменения в рамках проекта «Монета»

В рамках проекта «Монета» в язык Java 7 было добавлено шесть основных нововведений. Это строки (`String`) в конструкции `switch`, новые формы числовых литералов, усовершенствованная обработка исключений, применение ресурсов в блоке `try` (конструкция `try-with-resources`), ромбовидный синтаксис и исправление ситуации с предупреждениями, возникавшими при использовании функций с переменным количеством аргументов.

Мы собираемся подробно обсудить эти изменения, сделанные в рамках проекта «Монета». Нам предстоит поговорить о синтаксисе и о значении новых функций. Кроме того, мы попытаемся, насколько это возможно, объяснить мотивы, по которым были приняты те или иные новые функции. Мы не будем приводить полные формальные обоснования этих предложений, но все эти материалы доступны в архиве расылки coin-dev. Поэтому, если вы молодой и перспективный разработчик языков программирования, то там можете подробно изучить все предложения.

Итак, поговорим о первой из новых функций Java 7 — строковых значениях `String` в конструкции `switch`.

1.3.1. Строки в конструкции switch

Оператор switch в Java позволяет писать эффективные многократно ветвящиеся инструкции без многих и многих уродливо вложенных друг в друга операторов if — вот так:

```
public void printDay(int dayOfWeek) {
  switch (dayOfWeek) {
    case 0: System.out.println("Sunday"); break;
    case 1: System.out.println("Monday"); break;
    case 2: System.out.println("Tuesday"); break;
    case 3: System.out.println("Wednesday"); break;
    case 4: System.out.println("Thursday"); break;
    case 5: System.out.println("Friday"); break;
    case 6: System.out.println("Saturday"); break;
    default: System.err.println("Error!"); break;
  }
}
```

В Java 6 и ниже значениями для условий могли быть только константы типов byte, char, short, int (или технически их эквивалентные ссылочные типы Byte, Character, Short, Integer) либо константы enum. В Java 7 спецификация была расширена и к использованию был допущен тип String. В конце концов, это же константы.

```
public void printDay(String dayOfWeek) {
  switch (dayOfWeek) {
    case "Sunday": System.out.println("Dimanche"); break;
    case "Monday": System.out.println("Lundi"); break;
    case "Tuesday": System.out.println("Mardi"); break;
    case "Wednesday": System.out.println("Mercredi"); break;
    case "Thursday": System.out.println("Jeudi"); break;
    case "Friday": System.out.println("Vendredi"); break;
    case "Saturday": System.out.println("Samedi"); break;
    default: System.out.println("Error: '"+ dayOfWeek
    ➥ +"' is not a day of the week"); break;
  }
}
```

Во всех других отношениях конструкция switch не изменилась. Как и многие другие изменения в рамках проекта «Монета», это совсем простые нововведения, которые, однако, значительно упрощают жизнь Java-программистам.

1.3.2. Усовершенствованный синтаксис для числовых литералов

Было сделано несколько самостоятельных предложений, касающихся нового синтаксиса целочисленных типов. В итоге приняты следующие положения:

- числовые константы (то есть любой из примитивных целочисленных типов) теперь могут выражаться в виде двоичных литералов;

- в целочисленных константах могут использоваться нижние подчеркивания — они повышают удобочитаемость кода.

На первый взгляд оба изменения кажутся довольно неброскими. Но оба нововведения вызвали небольшое раздражение у Java-программистов.

Однако эти усовершенствования очень интересны для низкоуровневых программистов — тех, кто работает с «сырыми» сетевыми протоколами, шифрованием или занимается другими делами, предполагающими операции с битами. Итак, начнем с рассмотрения двоичных литералов.

Двоичные литералы

До выхода Java 7, если вы хотели манипулировать двоичными значениями, приходилось либо пользоваться неуклюжим (и чреватым множеством ошибок) преобразованием к исходному типу (base conversion), либо задействовать методы `parseX`. Например, если вы хотите гарантировать, чтобы `int x` правильно представляло комбинацию битов для десятичного значения `102`, напишите такое выражение:

```
int x = Integer.parseInt("1100110", 2);
```

Это слишком большое количество кода, если требуется просто гарантировать, что `x` в конечном итоге имеет правильную комбинацию битов. Несмотря на красивый вид этой записи, такой подход связан с некоторыми проблемами.

- Очень много текста.
- При вызове такого метода заметно страдает производительность.
- Необходимо знать о двухаргументной форме `parseInt()`.
- Нужно помнить детали работы `parseInt()` при наличии двух аргументов.
- Значительно осложняется работа динамического компилятора.
- Сущность, являющаяся константой во время компиляции, представляется как выражение времени исполнения. Это означает, что константа не сможет использоваться в качестве значения в операторе `switch`.
- Вы получите исключение `RuntimeException` (но не исключение времени компиляции), если допустите опечатку в двоичном значении.

К счастью, после появления Java 7 мы можем поступить так:

```
int x = 0b1100110;
```

Мы не утверждаем, что эта конструкция позволяет совершить что-то ранее невозможное. Но в таком случае не возникает ни одной из перечисленных выше проблем.

Если у вас будут причины для работы с двоичными файлами, то эта маленькая функция вам очень пригодится. Например, при выполнении низкоуровневой обработки байтов вы можете оперировать комбинациями битов как двоичными константами в операторах `switch`.

Еще одно небольшое, но полезное нововведение для представления групп битов или других длинных числовых представлений — применение нижних подчеркиваний в числах.

Нижние подчеркивания в числах

Вы, наверное, не сомневаетесь, что человеческий мозг принципиально отличается от компьютерного процессора. Характерным примером такого отличия является то, как наш мозг обрабатывает числа. Вообще, люди плохо умеют обращаться с длинными последовательностями чисел. Это — одна из причин, по которым появилась шестнадцатеричная система счисления, ведь наш разум лучше оперирует более короткими строками, сильнее насыщенными информацией, чем длинными последовательностями, каждый символ в которых сравнительно малоинформативен.

Так, нам будет проще работать с 1c372ba3, чем с 00011100001101110010101110100011, хотя процессор всегда будет иметь дело со второй формой. Один из способов, которым люди научились справляться с длинными последовательностями чисел, — разбиение их на части. Так, телефонный номер в США обычно имеет следующую форму: 404-555-0122.

ПРИМЕЧАНИЕ

Если любознательные читатели из Европы задумывались о том, почему номера американских телефонов в фильмах и книгах всегда начинаются с 555, поясним: номера в диапазоне 555-01xx специально зарезервированы для такого «художественного» использования, чтобы никто не получал телефонных звонков от зрителей, воспринимающих голливудское кино чересчур серьезно.

В других длинных последовательностях чисел применяются разделительные знаки, например:

- $100,000,000 (большие суммы денег в США);
- 08-92-96 (номера отделений банков в Великобритании).

К сожалению, и запятая и дефис — слишком многозначные символы в сфере обработки чисел в программировании. Поэтому оба символа не подходят нам в качестве разделительных знаков. Вместо этого разработчики проекта «Монета» позаимствовали идею из языка Ruby и ввели в качестве разделительного знака нижнее подчеркивание (_). Обратите внимание: здесь показан всего лишь фрагмент удобного для чтения синтаксиса, применяемого во время компиляции. Компилятор извлекает из кода нижние подчеркивания и сохраняет обычные цифры.

Это означает, что вы можете записать 100_000_000 и, скорее всего, не перепутаете это число с 10_000_000, тогда как перепутать 100000000 с 10000000 очень легко. Рассмотрим пару примеров, как минимум один из них должен показаться вам знакомым:

```
long anotherLong = 2_147_483_648L;
int bitPattern = 0b0001_1100__0011_0111__0010_1011__1010_0011;
```

Обратите внимание, насколько проще стало читать значение, присвоенное `anotherLong`.

ВНИМАНИЕ

В Java по-прежнему допустимо использовать строчную букву l для обозначения типа long. Например, 1010100l. Обязательно используйте L в верхнем регистре, чтобы люди, занимающиеся поддержкой кода, не путали букву l и число 1. Запись 1010100L гораздо понятнее.

Надеемся, что вы уже убедились в пользе этих дополнений при обработке целых чисел. Далее мы поговорим об усовершенствованной обработке исключений в Java 7.

1.3.3. Усовершенствованная обработка исключений

Усовершенствования связаны с двумя основными изменениями: множественным `catch` (multicatch) и переброской неизменяемого исключения (final rethrow). Чтобы разобраться, чем полезны эти изменения, рассмотрим код на Java 6, который пытается найти, открыть файл конфигурации, произвести его синтаксический разбор и обработать несколько различных исключений (листинг 1.1).

Листинг 1.1. Обработка нескольких различных исключений в Java 6

```
public Configuration getConfig(String fileName) {
  Configuration cfg = null;
  try {
    String fileText = getFile(fileName);
    cfg = verifyConfig(parseConfig(fileText));
  } catch (FileNotFoundException fnfx) {
    System.err.println("Config file '" + fileName + "' is missing");
  } catch (IOException e) {
    System.err.println("Error while processing file '" + fileName + "'");
  } catch (ConfigurationException e) {
    System.err.println("Config file '" + fileName + "' is not consistent");
  } catch (ParseException e) {
    System.err.println("Config file '" + fileName + "' is malformed");
  }

  return cfg;
}
```

В этом методе возникает несколько различных ситуаций, приводящих к выбросу исключения:

- возможно, файл конфигурации не существует;
- вероятно, файл конфигурации исчезнет как раз в то время, когда вы считываете из него информацию;
- файл конфигурации может иметь неправильное синтаксическое оформление;
- в файле конфигурации может содержаться неверная информация.

Функционально эти исключительные ситуации относятся к двум разным группам: либо файл отсутствует или каким-то образом поврежден, либо файл присутствует и не содержит ошибок, но его не удается правильно получить (например, из-за ошибки оборудования или отказа сети).

Попробуем свести все варианты к этим двум случаям — тогда мы будем обрабатывать все исключения вида «файл отсутствует или каким-то образом поврежден» в одном условии `catch` (листинг 1.2). В Java 7 это возможно.

Листинг 1.2. Обработка нескольких различных исключений в Java 7

```
public Configuration getConfig(String fileName) {
  Configuration cfg = null;
  try {
    String fileText = getFile(fileName);
    cfg = verifyConfig(parseConfig(fileText));
  } catch (FileNotFoundException|ParseException|ConfigurationException e) {
    System.err.println("Config file '" + fileName +
                       "' is missing or malformed");
  } catch (IOException iox) {
    System.err.println("Error while processing file '" + fileName + "'");
}
return cfg;
}
```

Исключение `e` относится к типу, который не удается точно распознать во время компиляции. Это означает, что оно должно обрабатываться в блоке `catch` как относящееся к общему супертипу, охватывающему все типы исключений, к которым оно *могло бы* относиться (на практике это обычно бывает `Exception` или `Throwable`).

Еще один пример нового синтаксиса помогает разобраться с переброской исключений. Во многих случаях разработчику может потребоваться произвести какие-то манипуляции с полученным исключением до его переброски. Проблема заключается в том, что в более ранних версиях Java часто можно было встретить такой код:

```
try {
  doSomethingWhichMightThrowIOException();
  doSomethingElseWhichMightThrowSQLException();
} catch (Exception e) {
  ...
  throw e;
}
```

Соответственно, вы вынуждены объявить сигнатуру исключения в этом коде как `Exception` — реальный динамический тип исключения оказывается поглощен.

Тем не менее несложно заметить, что исключение в этом коде может относиться только к типу `IOException` или `SQLException`, а если вы это видите, то и компилятор тоже видит. В синтаксисе Java 7 этот фрагмент изменится всего на одно слово:

```
try {
  doSomethingWhichMightThrowIOException();
  doSomethingElseWhichMightThrowSQLException();
} catch (final Exception e) {
  ...
  throw e;
}
```

Поскольку здесь присутствует ключевое слово `final`, понятно, что фактически выбрасывается тип времени исполнения для найденного исключения — в данном

случае это может быть лишь IOException или SQLException. Этот прием и называется *переброской неизменяемого исключения (final rethrow)*. Она защищает нас от получения слишком общего типа исключения, который потом можно было бы перехватить лишь на очень высоком уровне, с помощью сильно обобщенной конструкции catch.

В предыдущем примере ключевое слово final является опциональным. Но практика показывает, что использовать его полезно — это помогает при корректировке новой семантики конструкций catch и rethrow.

Наряду с этими общими усовершенствованиями обработки исключений в Java 7 было улучшено и управление ресурсами. Об этом мы поговорим далее.

1.3.4. Использование ресурсов в блоке try (try-with-resources)

Объяснить это изменение довольно легко, но оказалось, что такой механизм обладает скрытыми нюансами, из-за которых его реализация происходит значительно сложнее, чем ожидалось. Основная идея заключается в том, чтобы поместить ресурс (например, файл или тому подобную сущность) в область видимости блока таким образом, чтобы ресурс автоматически закрывался после выхода управления за пределы блока.

Это важное изменение по той простой причине, что практически никто не умеет вручную закрывать ресурсы на 100 % правильно. До недавнего времени даже в справочных руководствах от Sun этот процесс был описан с ошибками. Предложение, поданное в рамках проекта «Монета» и касающееся этого изменения, содержит ошеломляющее утверждение: якобы две трети случаев закрытия close() в JDK выполнены с ошибками!

К счастью, компилятор можно хорошо приспособить к созданию именно такого «педантичного» шаблонного кода, в котором человек часто допускает ошибки. Как раз такой подход и был выбран при реализации этого улучшения.

Действительно, новая конструкция очень помогает писать безошибочный код. Чтобы оценить, насколько она полезна, просто представьте себе, как бы вы написали в Java 6 блок кода, который считывает информацию из потока, идущего с URL (url), а потом записывает информацию в файл (out). Вот один из вариантов решения (листинг 1.3).

Листинг 1.3. Синтаксис для управления ресурсами в Java 6

```
InputStream is = null;
try {
  is = url.openStream();
  OutputStream out = new FileOutputStream(file);
  try {
    byte[] buf = new byte[4096];
    int len;
    while ((len = is.read(buf)) >= 0)
      out.write(buf, 0, len);
  } catch (IOException iox) {
```

Обработка исключения (read или write)

```
  } finally {
    try {
      out.close();
    } catch (IOException closeOutx) {
  }
}
} catch (FileNotFoundException fnfx) {
} catch (IOException openx) {
} finally {
  try {
    if (is != null) is.close();
  } catch (IOException closeInx) {
  }
}
```

А у вас получилось похоже? Основной момент здесь заключается в том, что при обработке внешних ресурсов действует один из законов Мерфи: «Что угодно может сорваться в любой момент»:

- поток `InputStream`, из которого нужно считывать информацию, может не открыться по URL либо может закрыться неправильно;
- файл `File`, соответствующий потоку `OutputStream`, может не открыться (и в него нельзя будет записать информацию) либо может закрыться неправильно;
- может возникнуть проблема, обусловленная несколькими факторами.

Последняя проблема, доставляющая сильную головную боль, — очень сложно правильно разобраться с комбинацией нескольких исключений.

Именно по этой причине в основном предпочтение отдается новому синтаксису — в нем просто сложнее допустить ошибку. Компилятор не допускает таких ошибок, которые легко совершает любой разработчик, пытающийся написать код такого типа вручную.

Рассмотрим код Java 7, осуществляющий такую же задачу, как и код из листинга 1.3. Как и выше, `url` — это объект URL (универсальный локатор ресурса), указывающий на сущность, которую вы хотите загрузить; `file` — это объект `File`, в котором вы хотите сохранить загружаемую информацию. Вот что получится в Java 7(листинг 1.4).

Листинг 1.4. Синтаксис Java 7 для управления ресурсами

```
try (OutputStream out = new FileOutputStream(file);
     InputStream is = url.openStream() ) {
  byte[] buf = new byte[4096];
  int len;
  while ((len = is.read(buf)) > 0) {
    out.write(buf, 0, len);
  }
}
```

В этом простейшем примере показан синтаксис блока для автоматического управления — операция try с ресурсом в круглых скобках. Программисту, имеющему опыт работы с языком C#, эта конструкция может напомнить условие using. Концептуально это хороший отправной пункт для работы с данной новой функцией. Ресурсы используются в блоке и автоматически высвобождаются после того, как вы закончите работу с ними.

При работе с конструкцией try-with-resources нужно оставаться начеку, так как в некоторых случаях ресурс может остаться незакрытым. Например, следующий код не закроет FileInputStream правильно, если была допущена ошибка при создании ObjectInputStream из файла (someFile.bin).

```
try (ObjectInputStream in = new ObjectInputStream(new
     FileInputStream("someFile.bin")) ) {
  ...
}
```

Предположим, что файл (someFile.bin) существует, но, возможно, это будет не файл ObjectInput. А файл другого формата может не открыться правильно. Следовательно, поток ObjectInputStream не будет создан, а поток FileInputStream не будет закрыт!

Наилучший способ гарантировать правильность работы try-with-resources — разбивать ресурсы на отдельные переменные.

```
try (FileInputStream fin = new FileInputStream("someFile.bin");
     ObjectInputStream in = new ObjectInputStream(fin) ) {
  ...
}
```

Один из аспектов конструкции TWR — появление улучшенных стектрейсов и заблокированных исключений. До Java 7 информация об исключении могла быть поглощена при обработке ресурсов. Такой вариант не исключен и с TWR, поэтому стековые следы были усовершенствованы. Теперь вы можете просматривать в исключениях и информацию о типах, которая ранее просто терялась.

Например, рассмотрим следующий фрагмент кода, в котором нулевой InputStream возвращается из метода:

```
try(InputStream i = getNullStream()) {
    i.available();
}
```

В результате получится улучшенный стектрейс, в котором мы увидим заблокированное исключение NullPointerException (сокращенно — NPE):

```
Exception in thread "main" java.lang.NullPointerException
  at wgjd.ch01.ScratchSuprExcep.run(ScratchSuprExcep.java:23)
  at wgjd.ch01.ScratchSuprExcep.main(ScratchSuprExcep.java:39)
  Suppressed: java.lang.NullPointerException
  at wgjd.ch01.ScratchSuprExcep.run(ScratchSuprExcep.java:24)
    1 more
```

TWR И AUTOCLOSEABLE

На уровне системы функция TWR реализуется с помощью нового интерфейса, называемого `AutoCloseable`. Класс должен реализовывать этот интерфейс, чтобы работать в рамках нового условия `try` конструкции TWR как ресурс. Многие классы платформы Java 7 были преобразованы так, чтобы могли реализовывать интерфейс `AutoCloseable` (в свою очередь, `AutoCloseable` был сделан суперинтерфейсом `Closeable`). Но необходимо учитывать, что эта новая технология задействована еще не во всех местах платформы. Эта функция, правда, была включена в состав JDBC 4.1.

В вашем собственном коде вам определенно следует использовать TWR всегда, когда возникает необходимость работать с ресурсами. Этот механизм позволит вам избежать ошибок при обработке исключений.

Рекомендуем приступать к использованию конструкции `try-with-resources`, как только будет возможность. Тогда вы сможете легко избавиться от ошибок в вашей базе кода.

1.3.5. Ромбовидный синтаксис

В Java 7 есть и такое изменение, которое частично избавляет вас от набора кода при работе с обобщенными сущностями (дженериками). Одна из проблем при работе с дженериками заключается в том, что определение и настройка экземпляров бывают очень пространными. Допустим, у вас есть несколько пользователей, которых вы идентифицируете по `userid` (этот идентификатор представляет собой целое число). И каждый пользователь имеет одну или несколько таблиц поиска, специфичных именно для него. Как все это будет выглядеть в коде?

```
Map<Integer, Map<String, String>> usersLists =
    new HashMap<Integer, Map<String, String>>();
```

Согласитесь, многовато текста, причем почти половина информации дублируется. Может быть, лучше написать что-то в таком роде:

```
Map<Integer, Map<String, String>> usersLists = new HashMap<>();
```

А информацию о типах в правой части выражения пускай обрабатывает компилятор. Проект «Монета» предоставляет вам такую возможность. В Java 7 подобная сокращенная форма объявлений абсолютно допустима. Кроме того, такая форма обладает обратной совместимостью. Поэтому, если вам доведется пересматривать старый код, то можете сократить старые пространные объявления и перейти к использованию нового синтаксиса с выведением типов. Запись получится короче на несколько пикселов.

Необходимо отметить, что при работе с этой функцией компилятор использует новую форму выведения типов. В правой части выражения он выводит правильный тип для выражения, а не просто подставляет текст, определяющий полный тип.

ПОЧЕМУ ТАКОЙ СИНТАКСИС НАЗЫВАЕТСЯ РОМБОВИДНЫМ

Такой синтаксис называется ромбовидным потому, что сокращенная информация о типе имеет форму ромба. Официальное название, фигурирующее в предложении, — «Усовершенствованное выведение типов при создании экземпляров обобщенных сущностей» (Improved Type Inference for Generic Instance Creation). Это чересчур длинное выражение, сокращаемое до аббревиатуры ITIGIC. Звучит довольно глупо, так что остановимся на термине «ромбовидный синтаксис».

Несомненно, новый ромбовидный синтаксис позволяет печатать меньше кода. Наконец, рассмотрим еще одну новую функцию, появившуюся в рамках проекта «Монета». Речь пойдет об удалении предупреждений, возникающих при использовании переменного количества аргументов.

1.3.6. Упрощенный вызов методов с переменным количеством аргументов

Это одно из самых простых изменений. Оно удаляет предупреждение, касающееся информации о типе. Такие предупреждения возникают в очень специфичных случаях, когда переменное количество аргументов комбинируется с дженериками в сигнатуре метода.

Иными словами, если вы не имеете привычки писать код, принимающий в качестве аргументов переменное количество ссылок типа `T`, а потом осуществляете операцию, требующую сделать выборку из этих ссылок, то можете пропустить этот раздел и переходить к следующему. Если же фрагмент ниже напоминает что-то из вашего творчества, то читайте дальше.

```
public static <T> Collection<T> doSomething(T... entries) {
  ...
}
```

Итак, читаем дальше? Хорошо. Разберемся, в чем же проблема.

Вы, наверное, знаете, что метод с переменным количеством аргументов — это такой метод, который принимает переменное количество параметров (все эти параметры относятся к одному типу) в конце списка аргументов. Но вы, возможно, не знаете, как реализуются такие vararg-методы. Как правило, все находящиеся в конце списка переменные параметры помещаются в массив (автоматически создаваемый для вас компилятором) и передаются как единый параметр.

Все это, конечно, хорошо, но тут мы сталкиваемся с одним из признанных слабых мест дженериков Java — как правило, вы не можете создать массив элементов уже известного обобщенного типа. Например, следующий код не скомпилируется:

```
HashMap<String, String>[] arrayHm = new HashMap<>[2];
```

Нельзя создать массив указанного обобщенного типа. Вместо этого приходится сделать так:

```
HashMap<String, String>[] warnHm = new HashMap[2];
```

Выдается предупреждение, которое следует проигнорировать. Обратите внимание: вы можете определить тип `warnHm` как массив `HashMap<String, String>` — но просто не можете создать ни одного экземпляра такого типа. Приходится «наступить себе на горло» (или как минимум заблокировать предупреждение) и принудительно преобразовать экземпляр необработанного типа (представляющий собой массив `HashMap`) в `warnHm`.

Два этих феномена — во-первых, методы с переменным количеством аргументов, обрабатывающие сгенерированные компилятором массивы, и, во-вторых, массивы известных обобщенных типов, которые, однако, невозможно инстанцировать — вместе становятся очередной головной болью. Рассмотрим следующий код:

```
HashMap<String, String> hm1 = new HashMap<>();
HashMap<String, String> hm2 = new HashMap<>();
Collection<HashMap<String, String>> coll = doSomething(hm1, hm2);
```

Компилятор попытается создать массив, содержащий как `hm1`, так и `hm2`, но тип этого массива строго должен быть одним из запрещенных типов массивов. Сталкиваясь с такой дилеммой, компилятор жульничает и нарушает собственное правило о запрещенном массиве обобщенного типа. Он создает экземпляр массива, но при этом «ругается», выдавая невразумительное предупреждение вида «непроверенные или небезопасные операции».

С точки зрения системы типов тут все вполне правильно. Но несчастный разработчик всего-то хотел использовать штуку, которая казалась совершенно нормальным API, а ему выдают какие-то грозные предупреждения по каким-то необъясненным причинам.

Куда деваются предупреждения в Java 7?

Рассматриваемая новая функция в Java 7 изменяет акцент в предупреждении. В конце концов, в конструкциях таких типов есть потенциал для нарушения безопасности типов, и *кого-то* об этом лучше предупредить. Но программисты, пользующиеся такими типами API, могут сделать с их помощью не так много. Код в `doSomething()` либо нарушает безопасность типов, либо не нарушает. В любом случае от пользователя API ничего не зависит.

Кому действительно важно получить такое предупреждение — так это товарищу, написавшему `doSomething()`, то есть производителю API, а не его потребителю. Туда и отправляется предупреждение. Оно попадает не туда, где используется API, а туда, где этот API был определен.

Раньше предупреждение инициировалось при компиляции кода, использовавшего данный API. Теперь же оно выдается в том случае, когда пишется API, обладающий потенциалом для подобного нарушения безопасности типов. Компилятор предупреждает программиста, реализующего API, — и обратить внимание на систему типов должен именно его разработчик.

Чтобы упростить жизнь и разработчикам API, Java 7 также предоставляет новый тип аннотаций: `java.lang.SafeVarargs`. Такая аннотация может применяться к методу

API (или конструктору), который в противном случае выдал бы предупреждение рассмотренного выше вида. Аннотируя метод с помощью `@SafeVarargs`, разработчик, в сущности, гарантирует, что этот метод не будет совершать никаких небезопасных операций. Если это произойдет, то компилятор заблокирует предупреждение.

Изменения в системе типов

Как много слов пришлось написать, чтобы рассказать о совсем маленьком изменении! Итак, возможность переместить предупреждение из одного места в другое — это, конечно, не революционное нововведение в языке. Но рассмотренный пример помогает проиллюстрировать один очень важный момент. Выше в этой главе мы упоминали, что в рамках проекта «Монета» разработчикам рекомендуется не трогать систему типов, предлагая изменения. А на этом примере видно, сколько всего требуется сделать, чтобы понять, как именно взаимодействуют различные компоненты системы типов и как изменятся эти взаимодействия вслед за внесением изменений в структуру языка. Данное изменение даже не назовешь особенно сложным — в отличие от тех, о которых нам еще предстоит поговорить. Изменения, рассмотренные ниже, намного более сложны и потенциально могут разветвляться на десятки несхожих ситуаций, различия между которыми, однако, не такие явные.

Этот последний пример демонстрирует, насколько заковыристым может быть эффект от самых небольших изменений. Хотя здесь мы обсудили в основном мелкие синтаксические изменения, они могут значительно улучшить ваш код. Действительно, мал золотник, да дорог. Как только вы начнете работать с этими изменениями, вы убедитесь, насколько они полезны в создаваемых программах.

1.4. Резюме

Вносить изменения в язык непросто. Новую функцию всегда легче реализовать в виде библиотеки (если это, конечно, возможно — некоторые функции нереализуемы без внесения изменений в язык). Связанные с этим сложности вынуждают разработчиков языка вносить мелкие и сравнительно консервативные изменения, чем им, возможно, хотелось бы.

Теперь перейдем к более обширным изменениям, появившимся в этом релизе. Для начала рассмотрим, какие из основных библиотек поменялись в Java 7. Следующая большая тема — это библиотеки ввода-вывода, которые были во многом пересмотрены. Вам совсем не помешают знания о том, как ввод-вывод был организован в более ранних версиях Java, так как классы Java 7 (иногда называемые NIO.2) построены на базе уже имеющегося фреймворка.

Если вы хотите изучить дополнительные примеры синтаксиса TWR в действии либо узнать о новых высокопроизводительных классах ввода-вывода, то переворачивайте страницу — обо всем этом рассказано в следующей главе.

2 Новый ввод-вывод

В этой главе:

- новые API ввода-вывода в Java 7 (также называемые NIO.2);
- `Path` — новая основа для файлового ввода-вывода и ввода-вывода каталогов;
- вспомогательный класс `Files` и его различные вспомогательные методы;
- решение типичных практических сложностей с вводом-выводом;
- введение в асинхронный ввод-вывод.

Одно из самых значительных изменений API в языке Java — крупные обновления в наборе API для ввода-вывода. Новый набор называется «Обновленный ввод-вывод» или NIO.2 (описан в запросе на спецификацию JSR-203). Именно ему посвящена основная часть данной главы. NIO.2 — это набор новых классов и методов, большинство из которых относится к пакету `java.nio`.

- Этот пакет является полнофункциональной заменой `java.io.File`. Пакет предназначен для написания кода, взаимодействующего с файловой системой.
- Он содержит новые асинхронные классы, с помощью которых вы сможете выполнять файловые и сетевые операции ввода-вывода в фоновом потоке. При этом не придется вручную конфигурировать пулы потоков и другие низкоуровневые конструкции, предназначенные для параллельной обработки.
- Пакет упрощает написание кода с использованием сокетов и каналов, так как появляется новая конструкция `Network-Channel`.

Рассмотрим пример практического использования. Предположим, начальник поручил вам написать процедуру Java, которая просматривает все каталоги на рабочем сервере и находит все файлы свойств, содержащие заданный набор привилегий на чтение/запись и владение файлом. В Java 6 (и ниже) решить такую задачу практически невозможно в силу нескольких причин:

- на уровне классов и методов отсутствует прямая поддержка навигации по дереву каталогов;
- нет способа поиска символьных ссылок и манипулирования ими;
- отсутствует простая операция для считывания атрибутов файла (таких `readable`, `writable` или `executable`).

Новый API Java 7 NIO.2 значительно упрощает решение этой программной задачи, так как непосредственно поддерживает навигацию по дереву каталогов (`Files`.

`walkFileTree()`, см. подраздел 2.3.1), символьные ссылки[1] (`Files.isSymbolicLink()`, см. листинг 2.4), а также простые однострочные операции для считывания атрибутов файлов (`Files.readAttributes()`, см. подраздел 2.4.3).

Кроме того, начальник требует, чтобы эти файлы свойств считывались, не нарушая основного рабочего потока программы. Вы знаете, что один из файлов свойств имеет размер не менее 1 Мбайт; считывание этого файла наверняка помешает основному рабочему потоку программы! В Java 5/6 пришлось бы использовать классы из пакета `java.util.concurrent` для создания пулов потоков и рабочих очередей, чтобы считывание этого файла происходило в отдельном фоновом потоке. Как будет показано в главе 4, современная параллельная обработка в Java остается довольно сложной, разброс возможных ошибок очень велик. Но в Java 7 с помощью API NIO.2 вы сможете считать большой файл в фоновом режиме, не указывая собственных рабочих потоков и очередей, пользуясь новым `AsynchronousFileChannel` (см. раздел 2.5). Уф!

Эти новые API — не панацея (хотя функции очень симпатичные), тем не менее они вам очень пригодятся, учитывая, каковы основные тенденции в нашем деле.

Во-первых, все активнее исследуются альтернативные методы хранения данных, особенно в случаях с нереляционными и очень крупными наборами данных. Это означает, что в ближайшем будущем предстоит много работать со считыванием и записью крупных файлов (например, большие файлы отчетов с сервиса микроблогов). NIO.2 позволяет считывать и записывать крупные файлы в эффективном асинхронном режиме, опираясь при этом на базовые функции операционной системы.

Вторая тенденция — распространение многоядерных процессоров, открывающих возможности для по-настоящему параллельного (а значит, быстрого) ввода-вывода. Параллелизм — искусство, которое не так просто освоить[2], но NIO.2 значительно облегчает эту задачу, предлагая простые абстракции для использования многопоточного доступа к файлам и сокетам. Даже если вы не используете эти функции непосредственно, они все равно будут все сильнее влиять на вашу практику программирования по мере их распространения в IDE, серверах приложений и в популярных фреймворках.

Это всего несколько примеров, демонстрирующих, чем может быть полезен NIO.2. Если у вас сложилось впечатление, что NIO.2 — это механизм, помогающий решить некоторые типичные проблемы, с которыми вы сталкиваетесь в ходе разработки, то эта глава — для вас! Если описанные ситуации вам не встречались, то всегда можно вернуться к разделам этой главы, описывающим задачи программирования ввода-вывода на Java.

Текущая глава позволяет в достаточной мере познакомиться с возможностями ввода-вывода в Java 7, чтобы приступить к написанию кода на основе NIO.2 и уверенно исследовать новые API. Вам поможет и то, что некоторые из рассматриваемых здесь функций уже затрагивались в первой главе. Это еще раз доказывает, что Java 7 действительно умеет пользоваться встроенными в него механизмами!

---

[1] Символьная ссылка — это особый тип файла, который указывает на другой файл или точку в файловой системе. Такая ссылка напоминает ярлык для доступа к файлу.

[2] В главе 4 рассмотрены те сложные нюансы, с которыми может столкнуться программист при реализации параллелизма.

СОВЕТ

Комбинация конструкции try-with-resources (см. главу 1) и новых API из NIO.2 обеспечивает исключительно надежное программирование ввода-вывода — пожалуй, впервые в истории Java!

Полагаем, что вы, скорее всего, планируете воспользоваться новыми возможностями файлового ввода-вывода. Поэтому мы рассмотрим эту тему очень подробно. Сначала мы поговорим о новой абстракции файловой системы — `Path`, а также о классах, поддерживающих эту абстракцию. Опираясь на знания о `Path`, мы обсудим обычные операции, осуществляемые в файловой системе, такие как перемещение и копирование файлов.

Кроме того, мы обсудим асинхронный ввод-вывод и рассмотрим пример, основанный на свойствах файловой системы. Наконец, мы поговорим о взаимопроникновении функционала `Socket` и `Channel`, а также о значении этих изменений для разработчиков сетевых приложений. Но начнем с того, как появился NIO.2.

2.1. История ввода-вывода в Java

Чтобы в полной мере оценить структуру API NIO.2 (и понять, как с ними нужно работать), основательный Java-разработчик должен знать историю ввода-вывода в Java. Но мы отлично понимаем, что вам уже не терпится перейти к коду! Если совсем не терпится — можете переходить к разделу 2.2.

Если какие-то примеры использования API покажутся вам особенно красивыми, а другие, возможно, немного странными, то этот раздел поможет вам взглянуть на NIO.2 с точки зрения создателей API. Это третья крупная реализация ввода-вывода в Java. Поэтому, чтобы понять, как появился NIO.2, обратимся к истории поддержки ввода-вывода в Java.

Одна из причин огромной популярности Java заключается в обширной библиотечной поддержке. В языке предоставляются мощные и лаконичные API, позволяющие решать разнообразные задачи при программировании. Но опытным Java-разработчикам известно, что есть несколько областей, которые были реализованы в ранних версиях Java практически на пещерном уровне. Одной из основных проблем для разработчиков долгое время были API для ввода-вывода (I/O).

2.1.1. Java 1.0–1.3

В самых ранних версиях Java (1.0–3.0) отсутствует полномасштабная поддержка ввода-вывода. В частности, разработчики сталкивались со следующими проблемами при создании приложений, требующих поддержки ввода-вывода.

- Отсутствовали концепции буферов или абстракций каналов для данных. Поэтому разработчикам приходилось много и кропотливо заниматься низкоуровневым программированием.
- Операции ввода-вывода часто блокировались, что ограничивало масштабируемость.
- Существовала лишь ограниченная поддержка различных символьных кодировок. Поэтому имелось множество запрограммированных вручную решений для поддержки определенных типов оборудования.

- Отсутствовала поддержка регулярных выражений, что осложняло манипуляции с данными.

Вообще, в Java отсутствовала поддержка неблокирующего ввода-вывода. Поэтому разработчикам с большим трудом удавалось создавать масштабируемые решения для ввода-вывода. Мы считаем, что эти неудобства частично объясняют, почему язык Java почти не использовался в разработках для серверной стороны до выхода Java 1.4.

2.1.2. Java 1.4 и неблокирующий ввод-вывод

Для того чтобы решить эти проблемы, Java приступила к реализации поддержки неблокирующего ввода-вывода, а также других функций, позволяющих разработчикам создавать более быстрые и надежные решения для реализации ввода-вывода. В этой области в разное время были достигнуты два крупных улучшения:

- в рамках Java 1.4 был реализован неблокирующий ввод-вывод;
- в Java 7 механизм неблокирующего вывода был коренным образом переработан.

На основании запроса на спецификацию JSR-51 неблокирующий ввод-вывод (NIO) был добавлен в Java в 2002 году, когда вышла версия 1.4. На этом этапе добавился следующий обширный набор функций, благодаря которым Java превратился в привлекательный язык для серверной разработки:

- абстрактные уровни (уровень буфера и уровень канала) для операций ввода-вывода;
- возможность кодировать и декодировать наборы символов;
- интерфейс, позволяющий ассоциировать файлы с данными, хранящимися в памяти;
- новая библиотека регулярных выражений, основанная на популярной реализации на языке Perl.

PERL — КОРОЛЬ РЕГУЛЯРНЫХ ВЫРАЖЕНИЙ

Язык программирования Perl — бесспорный лидер в области поддержки регулярных выражений. На самом деле его дизайн и реализация настолько хороши, что несколько языков программирования (в том числе Java) во многом скопировали синтаксис и семантику Perl. Если вас интересует история этого языка, то любую информацию о нем можно прочитать на сайте http://www.perl.org/.

Неблокирующий ввод-вывод был настоящим прорывом, но Java-разработчики по-прежнему сталкивались со сложностями при программировании ввода-вывода. В частности, поддержка обработки файлов и каталогов в файловой системе оставляла желать лучшего. На тот момент класс `java.io.File` содержал несколько досадных недоработок:

- отсутствовала непротиворечивая обработка имен файлов на разных платформах[1];

[1] Некоторые критики Java могли бы сказать, что именно в этой области язык Java нарушает известный принцип «Написать однажды, использовать везде».

- не удавалось создать унифицированную модель работы с атрибутами файлов (например, при моделировании доступа для чтения/для внесения изменений);
- возникали сложности с обходом каталогов;
- не удавалось использовать функции, специфичные для платформы или операционной системы[1];
- в файловых системах не поддерживались неблокирующие операции[2].

2.1.3. Введение в NIO.2

Запрос на спецификацию JSR-203 (составленную под руководством Алана Бейтмена) был подготовлен, чтобы избавиться от описанных выше ограничений, а также обеспечить поддержку для некоторых новых парадигм ввода-вывода, применявшихся в современном программном и аппаратном обеспечении. В результате JSR-203 появилась система, известная нам как API NIO.2 в Java 7. Она предназначалась для достижения трех основных целей, которые подробно описаны в самом документе JSR-203, раздел 2.1 (http://jcp.org/en/jsr/detail?id=203).

- Создание нового интерфейса файловой системы, поддерживающего групповой доступ к атрибутам файлов и обеспечивающего доступ к API, специфичным для данной файловой системы, а также интерфейса для предоставления сервисов для реализаций подключаемых файловых систем.
- Разработка API для асинхронных операций ввода-вывода (которые, в отличие от опроса, являются неблокирующими) как с сокетами, так и с файлами.
- Окончательная доработка сокетно-канального функционала, определенного в запросе на спецификацию JSR-51, включая добавление связывания, конфигурации параметров и мультикаст-датаграмм.

Начнем изучение основ новой поддержки файловых систем с обсуждения `Path` и сопутствующих тем.

2.2. Path — основа файлового ввода-вывода

`Path` — один из основных классов, с которым придется научиться работать, чтобы освоить файловый ввод-вывод в рамках NIO.2. Обычно `Path` означает путь к элементу файловой системы, например к `C:\workspace\java7developer` (каталог в файловой системе Microsoft Windows) или `/usr/bin/zip` (местоположение утилиты ZIP в файловой системе *nix). Если вы подробно изучите, как создавать пути и манипулировать ими, то сможете выполнять навигацию в файловой системе любого типа, в том числе по таким системам, как ZIP-архив.

---

[1] В данном случае обычно подразумеваются сложности с доступом к механизму символьных ссылок в операционных системах Linux/UNIX.

[2] Поддержка неблокирующих операций с сетевыми сокетами в Java 1.4 присутствовала.

На рис. 2.1 (в его основе — структура исходного кода для этой книги) мы кратко напоминаем о нескольких концепциях файловой системы:

- дерево каталогов;
- корень пути;
- абсолютный путь;
- относительный путь.

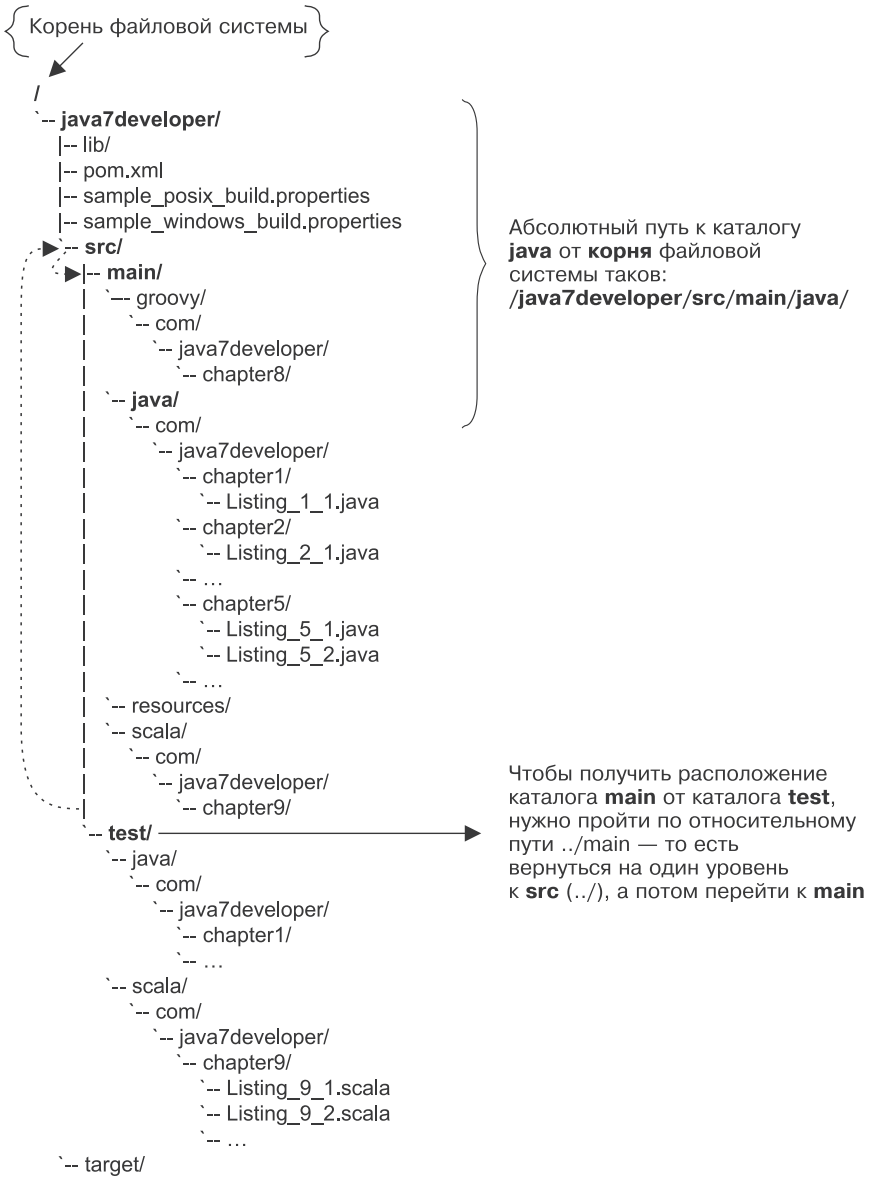


Рис. 2.1. Дерево каталогов, на котором показаны корень, абсолютный путь и относительный путь

Мы говорим об абсолютных и относительных путях, так как важно знать, где находится конкретное местоположение в файловой системе относительно того места, откуда запускается наше приложение. Например, приложение может запускаться из каталога `/java7developer/src/test` и у вас есть код, считывающий имена файлов из каталога `/java7developer/src/main`. Чтобы перейти в `/java7developer/src/main`, можно воспользоваться относительным путем `../main`.

А что, если ваше приложение запускается из `/java7developer/src/test/java/com/java7developer`? В таком случае при использовании относительного пути от `../main` вы не попадете в желаемый каталог (вместо этого вы окажетесь в несуществующем каталоге `/java7developer/src/test/java/com/main`). Действительно, в таком случае следует использовать абсолютный путь, например `/java7developer/src/main`.

Верно и обратное: ваше приложение может неизменно запускаться из одного и того же места (допустим, из целевого каталога на рис. 2.1). Но корень дерева каталогов может измениться (например, с `/java7developer` на `D:\workspace\j7d`). В таком случае вы не можете полагаться на абсолютный путь — придется использовать относительные пути, чтобы надежно переходить в желаемые точки.

Путь (`Path`) в системе NIO.2 — это абстрактная конструкция. Можно создать путь `Path` и манипулировать им, даже если он не связан с каким-то конкретным физическим местоположением. Хотя это и может показаться нелогичным, в некоторых случаях такая возможность очень полезна. Например, может понадобиться создать путь, представляющий собой указание на новый файл, когда вы еще только *собираетесь* создать этот файл. Этот файл не существует, пока вы не задействуете метод `Files.createFile(Path target)`[1]. Если бы вы попытались считать содержимое файла, представленного через `Path`, *до того*, как он создан, то получили бы исключение `IOException`. Та же логика действует и в случае, когда вы указываете несуществующий `Path` и пытаетесь считать из него информацию с помощью метода, например, `Files.readAllBytes(Path)`. Одним словом, виртуальная машина Java связывает `Path` с конкретным физическим местоположением только во время исполнения.

ВНИМАНИЕ

Будьте осторожны при написании кода, специфичного для файловой системы. Если создать путь Path со значением C:\workspace\java7developer, а потом попытаться считать информацию с этого адреса, то это сработает лишь на компьютерах, в файловой системе которых действительно существует местоположение C:\workspace\java7developer. Всегда следите за тем, чтобы ваша логика и порядок обработки исключений охватывали и случаи выполнения вашего кода в другой файловой системе, а также в файловой системе, структура которой может изменяться. Один из авторов книги как-то раз об этом забыл, из-за чего отказала целая серия жестких дисков на факультете информатики в его университете![2]

Еще раз отметим, что в NIO.2 четко разграничивается концепция местоположения (представленного `Path`) и операции с физической файловой системой (такие, как копирование файла). Подобные манипуляции обычно выполняются с помощью вспомогательного класса `Files`.

[1] Чуть ниже, в подразделе 2.4.1, мы поговорим об этом методе.

[2] Не будем никого называть по именам, но можете сами попытаться распутать эту таинственную историю.

Класс `Path` подробно описан в табл. 2.1 (как и кое-какие другие классы, которые встретятся нам в этом разделе).

Таблица 2.1. Основные классы для изучения ввода-вывода файлов

Класс	Описание
Path	Класс Path содержит методы, которые могут использоваться для получения информации о пути, доступа к элементам пути, преобразования пути в другие формы или для извлечения фрагментов пути. Кроме того, здесь есть методы для сопоставления строки пути с другими строками и для удаления избыточности из пути
Paths	Вспомогательный класс, предоставляющий вспомогательные методы для возвращения пути, например get(String first, String... more) и get(URI uri)
FileSystem	Класс, служащий интерфейсом для файловой системы. Это может быть как файловая система, заданная по умолчанию, так и альтернативная файловая система, узнаваемая по уникальному идентификатору ресурса (URI)
FileSystems	Вспомогательный класс, предоставляющий различные методы — например, метод FileSystems.getDefault(), возвращающий файловую систему, заданную по умолчанию

Не забывайте, что `Path` не обязательно должен представлять реальный файл или каталог. Вы можете манипулировать путем `Path` как вам угодно и использовать функционал `Files`, чтобы проверять, существует ли файл на самом деле. Если он существует, то с ним можно выполнять определенные операции.

СОВЕТ

Path можно использовать не только в традиционных файловых системах, но и в таких, как ZIP или JAR.

Исследуем класс `Path`, попытавшись решить пару простых задач:

- создание `Path`;
- получение информации о `Path`;
- удаление избыточности из `Path`;
- объединение двух `Path`, создание `Path`, представляющего собой путь между двумя `Path`, сравнение двух `Path`.

Начнем с создания пути `Path` для представления местоположения в файловой системе.

2.2.1. Создание пути

Создать путь совсем просто. Самый быстрый способ — вызвать метод `Paths.get(String first, String... more)`. Вторая переменная обычно не используется, она просто позволяет прикреплять дополнительные строки, из которых образуется строка `Path`.

СОВЕТ

Вы заметите, что в разных API NIO.2 единственное проверяемое исключение, выдаваемое различными методами в Path или Paths, — это IOException. Мы полагаем, это сделано ради простоты, но иногда единственный вид исключения может затемнять суть проблемы, вызвавшей исключение. В таком случае придется писать дополнительный код для обработки исключений, если вам понадобится решать проблемы, касающиеся конкретных разновидностей (подклассов) IOException.

Воспользуемся методом `Paths.get(String first)` для создания абсолютного пути `Path` к полезной программе-архиватору ZIP, находящейся в каталоге `/usr/bin/`:

```
Path listing = Paths.get("/usr/bin/zip");
```

Вызов `Paths.get("/usr/bin/zip")` эквивалентен вызову следующей более длинной последовательности:

```
Path listing = FileSystems.getDefault().getPath("/usr/bin/zip");
```

СОВЕТ

При создании Path можно использовать относительный путь. Например, ваша программа может работать из каталога /opt, и, чтобы создать путь Path к /usr/bin/zip, допустимо использовать ../usr/bin/zip. Так вы попадаете в каталог, расположенный на один уровень выше /opt (то есть на уровень /), а потом — в /usr/bin/zip. Не составляет труда превратить этот относительный путь в абсолютный, вызвав метод toAbsolutePath(). Например, вот так: listing.toAbsolutePath().

Можно запросить у `Path` определенную информацию: родительский компонент этого `Path`, его имя файла (при условии, что оно существует) и пр.

2.2.2. Получение информации о пути

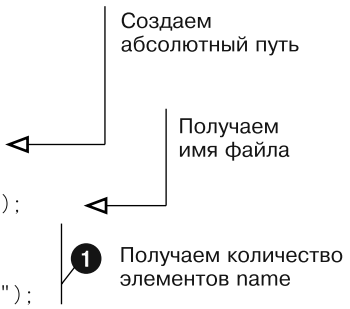
В классе `Path` имеется группа методов, возвращающих полезную информацию о пути, с которым вы работаете. В следующем листинге создается `Path` для полезной утилиты ZIP, расположенной в `/usr/bin`, а также выводится другая важная информация: корневой путь этого `Path` и родительский путь этого `Path`. Если вы работаете с операционной системой, утилита ZIP в которой расположена в `/usr/bin`, то должны получить примерно следующий вывод:

```
File Name [zip]
Number of Name Elements in the Path [3]
Parent Path [/usr/bin]
Root of Path [/]
Subpath from Root, 2 elements deep [usr/bin]
```

Запустив этот листинг на своем компьютере, вы можете получить разные результаты. Все зависит от того, с какой именно операционной системой вы работаете и откуда запускается листинг с кодом (листинг 2.1).

Листинг 2.1. Получение информации о пути

```
import java.nio.file.Path;
import java.nio.file.Paths;

public class Listing_2_1 {

  public static void main(String[] args) {
    Path listing = Paths.get("/usr/bin/zip");
    System.out.println("File Name [" +
                       listing.getFileName() + "]");
    System.out.println("Number of Name Elements
                       in the Path [" +
                       listing.getNameCount() + "]");
```



```
    System.out.println("Parent Path [" +
                       listing.getParent() + "]");
    System.out.println("Root of Path [" +
                       listing.getRoot() + "]");
    System.out.println("Subpath from Root,
                       2 elements deep [" +
                       listing.subpath(0, 2) + "]");

  }
}
```

❷ Получаем информацию о пути

После создания `Path` к `/usr/bin/zip` можно проверить, из какого количества элементов состоит этот `Path` (в данном случае нас интересует количество каталогов) ❶. Всегда полезно знать, где ваш путь `Path` расположен относительно родительского `Path` и корня. Кроме того, вы можете выбрать подпуть, указав начальный и конечный индекс. В данном случае вы находите подпуть, начиная от корня `(0)` до второго элемента в `Path (2)` ❷.

Эти методы окажутся крайне полезными на первом этапе изучения различных API NIO.2, так как вы сможете пользоваться ими для логирования результатов манипуляций с путями.

2.2.3. Избавление от избыточности

При написании служебных программ (например, анализатора файлов свойств) вы, возможно, получите `Path`, который может содержать такие элементы:

- `.` — означает текущий каталог;
- `..` — означает родительский каталог.

Предположим, что ваше приложение запускается из `/java7developer/src/main/java/com/java7developer/chapter2/` (см. рис. 2.1). Вы находитесь в том же каталоге, что и `Listing_2_1.java`. Поэтому, если вы получили `Path` от `./Listing_2.1.java`, то часть `./` (фактически это тот самый каталог, из которого вы запускаетесь) значения не имеет. В данном случае достаточно указать более краткий путь — `Listing_2_1.java`.

При интерпретации пути могут возникнуть и другие случаи избыточности, например символьные ссылки (о них мы поговорим в подразделе 2.4.3). Допустим, вы работаете в операционной системе *nix и ищете информацию о логе (файле журнала), который называется `log1.txt` и находится в каталоге `/usr/logs`. Но на самом деле этот каталог `/usr/logs` — просто символьная ссылка на `/application/logs`, то есть на каталог, в котором фактически находится этот файл журнала. Поскольку вас интересует именно реальное местоположение, от избыточной символьной информации лучше избавиться.

Все эти типы избыточности могут вызвать ситуацию, в которой `Path` не ведет к тому местоположению файла, куда, по вашим расчетам, должен указывать.

В Java 7 имеется пара вспомогательных методов, помогающих выяснить, куда именно направлен ваш `Path`. Во-первых, можно отсечь от `Path` избыточную информацию, воспользовавшись методом `normalize()` этого пути. Следующий фрагмент

кода возвращает Path от Listing_2_1.java, отсекая избыточную запись, указывающую на местоположение в одном и том же каталоге (часть ./).

```
Path normalizedPath = Paths.get("./Listing_2_1.java").normalize();
```

Еще существует мощный метод toRealPath(), в котором объединяется функционал методов toAbsolutePath() и normalize(). Он обнаруживает символьные ссылки и следует по ним.

Вернемся к примеру, где мы работали с операционной системой *nix и имели дело с файлом log1.txt, находящимся в каталоге /usr/logs. Этот каталог на самом деле представляет собой символьную ссылку на каталог /application/logs. С помощью метода toRealPath() вы получаете реальный путь Path к /application/logs/log1.txt.

```
Path realPath = Paths.get("/usr/logs/log1.txt").toRealPath();
```

Последняя функция API Path, которую мы здесь обсудим, позволяет манипулировать с несколькими Path. В ходе таких манипуляций мы сможем сравнивать пути, находить пути между ними и т. д.

2.2.4. Преобразование путей

Преобразование путей обычно требуется выполнять при работе со вспомогательными программами. Например, может понадобиться узнать, как файлы расположены друг относительно друга, — чтобы удостовериться, что структура каталогов с вашим исходным кодом соответствует принятым стандартам. Или вы можете получить ряд аргументов Path, переданных сценарием оболочки, выполняющим вашу программу. И эти аргументы потребуется преобразовать в правильный путь Path. В NIO.2 можно без труда объединять пути, создавать путь между двумя другими путями и сравнивать пути друг с другом.

В следующем фрагменте кода показано объединение двух путей: uat и conf/application.properties. Это делается с помощью метода resolve. Мы собираемся представить полный путь вида /uat/conf/application.properties.

```
Path prefix = Paths.get("/uat/");
Path completePath = prefix.resolve("conf/application.properties");
```

Чтобы получить путь между двумя другими путями, пользуйтесь методом relativize(Path). В следующем фрагменте кода рассчитывается путь к конфигурационному каталогу из каталога логирования.

```
String logging = args[0];
String configuration = args[1];
Path logDir = Paths.get(logging);
Path confDir = Paths.get(configuration);
Path pathToConfDir = logDir.relativize(confDir);
```

Как и ожидалось, можно использовать startsWith(Path prefix) и endsWith(Path suffix), а также полный сравнительный метод equals(Path path) для сравнения путей.

Теперь, когда вы научились работать с классом Path, что делать с уже имеющимся кодом, который был написан до перехода на Java 7? Команда разработчиков

NIO.2 позаботилась и об обратной совместимости и добавила пару специальных функций API, обеспечивающих взаимодействие между новым вводом-выводом на основе Path и предыдущими версиями Java.

2.2.5. Пути NIO.2 и класс File, существующий в Java

Классы в новом API файловой системы позволяют полностью заменить старый API, построенный на базе java.io.File. Но вам зачастую придется иметь дело с большими объемами устаревшего кода, использующего прежний ввод-вывод, основанный на java.io.File. В Java 7 появляются два новых метода, связанных с такими операциями.

- Новый метод toPath() для класса java.io.File, мгновенно преобразующий имеющийся файл File в новую конструкцию Path.
- Метод toFile() для класса Path, моментально преобразующий имеющийся путь Path в файл File.

  Эта возможность кратко продемонстрирована в следующем фрагменте кода.

```
File file = new File("../Listing_2_1.java");
Path listing = file.toPath();
listing.toAbsolutePath();
file = listing.toFile();
```

На этом мы заканчиваем исследование класса Path. Далее рассмотрим, как в Java 7 поддерживается работа с каталогами, в частности с деревьями каталогов.

2.3. Работа с каталогами и деревьями каталогов

Как вы, вероятно, уже поняли из рассказа о путях, приведенного в разделе 2.2, каталог — это просто путь Path со специфическими атрибутами. Потрясающая новая возможность Java 7 заключается в том, что здесь обеспечивается навигация по каталогам. Благодаря появлению нового интерфейса java.nio.file.DirectoryStream<T> и реализующих его классов вы можете выполнять следующие функции широкого применения:

- перебирать записи в каталоге, например находить в нем файлы;
- находить записи в каталоге, пользуясь выражениями-масками (например, *Foobar*) и обнаружением контента по типам MIME (например, файлы text/xml);
- выполнять рекурсивные операции перемещения, копирования и удаления с помощью метода walkFileTree.

В этом разделе мы рассмотрим две наиболее распространенные разновидности задач. Сначала поговорим о нахождении файлов в одном каталоге, а потом выполним ту же задачу во всем дереве каталогов. Начнем с простейшего случая — нахождения произвольных файлов в каталоге.

2.3.1. Поиск файлов в каталоге

Во-первых, рассмотрим простой пример использования фильтра для сопоставления по образцу (pattern matching) для перечисления всех файлов свойств (`.properties`) из каталога с проектом java7developer. Разберем следующий листинг (листинг 2.2).

Листинг 2.2. Перечисление файлов свойств в каталоге

```
Path dir = Paths.get("C:\\workspace\\java7developer");

try(DirectoryStream<Path> stream
    = Files.newDirectoryStream(dir, "*.properties")) {
  for (Path entry: stream)
  {
    System.out.println(entry.getFileName());
  }
}
catch (IOException e)
{
  System.out.println(e.getMessage());
}
```

❶ Задаем начальный путь

❷ Объявляем фильтрующий поток

❸ Перечисляем и выводим все файлы .properties

Начинаем с объявления знакомого вызова `Paths.get(String)` ❶. Основные события происходят при вызове `Files.newDirectoryStream(Path directory, String patternMatch)` ❷, в результате чего возвращается поток `DirectoryStream`, в котором отфильтрованы файлы, оканчивающиеся на `.properties`. Наконец, вы выводите все записи ❸.

Применяемый здесь механизм сопоставления с образцом называется *сопоставлением с маской* (glob pattern match). Он чем-то похож на привычное сопоставление с использованием регулярных выражений, которое вы могли встречать в Perl, но и отличается от такого сопоставления. Подробное описание сопоставления с образцом по маске и его применения дано в приложении B.

В листинге 2.2 продемонстрирован потенциал нового метода при работе с единственным каталогом. Но что, если нам потребуется выполнить рекурсивную фильтрацию по нескольким каталогам?

2.3.2. Движение по дереву каталогов

В Java 7 появилась поддержка навигации по целому дереву каталогов. Это означает, что вы с легкостью сможете искать файлы в дереве каталогов (то есть выполнять поиск в подкаталогах), а также осуществлять над ними желаемые действия. Возможно, вы захотите заполучить вспомогательный класс, который удаляет все файлы `.class` в каталоге `/opt/workspace/java` вашего пространства для разработки. Это может происходить на этапе очистки (cleanup step) при подготовке сборки.

Движение по дереву каталогов — довольно новая функция в Java 7. Для правильной работы с ней нужно знать несколько интерфейсов и подробности ее реализации. Основной метод, используемый при движении по дереву каталогов:

```
Files.walkFileTree(Path startingDir, FileVisitor<? super Path> visitor);
```

Указать `startingDir` легко, но предоставить реализацию интерфейса `FileVisitor` (это довольно затейливый параметр `FileVisitor<? super Path> visitor`) — уже несколько сложнее. Дело в том, что интерфейс `FileVisitor` вынуждает вас реализовать как минимум следующие пять методов (где `T` — это обычно `Path`):

- `FileVisitResult preVisitDirectory(T dir);`
- `FileVisitResult preVisitDirectoryFailed(T dir, IOException exc);`
- `FileVisitResult visitFile(T file, BasicFileAttributes attrs);`
- `FileVisitResult visitFileFailed(T file, IOException exc);`
- `FileVisitResult postVisitDirectory(T dir, IOException exc).`

Выглядит как большой кусок работы, правда? К счастью, создатели API Java 7 предоставили стандартную реализацию — класс `SimpleFileVisitor<T>`.

Мы дополним и изменим поведение, представленное в листинге 2.2. Как вы помните, там мы выводили список всех файлов `.properties` из каталога `C:\workspace\java7developer`. В следующем листинге перечисляются файлы `.java` с исходным кодом из всех каталогов, расположенных не выше `C:\workspace\java7developer\src`. В этом листинге продемонстрировано использование метода `Files.walkFileTree` со стандартной реализацией `SimpleFileVisitor`, улучшенной с помощью специфической реализации метода `visitFile` (листинг 2.3).

Листинг 2.3. Перечисление файлов с исходным кодом Java, содержащимся в подкаталогах

```
public class Find
{

  public static void main(String[] args) throws IOException
  {
    Path startingDir =
      Paths.get("C:\\workspace\\java7developer\\src");
    Files.walkFileTree(startingDir,
                       new FindJavaVisitor());
  }

  private static class FindJavaVisitor
                       extends SimpleFileVisitor<Path>
  {

    @Override
    public FileVisitResult
      visitFile(Path file, BasicFileAttributes attrs)
    {
      if (file.toString().endsWith(".java")) {
        System.out.println(file.getFileName());
      }
      return FileVisitResult.CONTINUE;
    }
  }
}
```

Задаем стартовый каталог

1 Вызов метода walkFileTree

Наследование от SimpleFileVisitor<Path>

2

Переопределение метода visitFile

3

Начинаем с вызова метода `Files.walkFileTree` (1). Основной момент, который здесь нужно отметить, заключается в том, что вы передаете `FindJavaVisitor`, дополняющий стандартную реализацию `SimpleFileVisitor` (2). Вы хотите, чтобы `SimpleFileVisitor` выполнил для вас основную работу по обходу каталогов и т. д. Весь код, который вам требуется написать, пишется при переопределении метода `visitFile(Path, BasicFileAttributes)` (3). Здесь мы пишем простой код на Java, чтобы проверить, оканчивается ли название файла на `.java`. Если оканчивается, то пишем этот файл в стандартный поток вывода[1].

Другой пример практического использования может заключаться в рекурсивном изменении файлов — перемещении, копировании, удалении или ином изменении. В большинстве случаев понадобится дополнять `SimpleFileVisitor`, но API обладает достаточной гибкостью и может работать с вашим собственным `FileVisitor`, если вы захотите реализовать его с нуля.

ПРИМЕЧАНИЕ

Метод walkFileTree не переходит по символьным ссылкам автоматически. Благодаря этому такие операции, как рекурсия, становятся безопаснее. Если вам требуются подобные переходы по символьным ссылкам, то нужно обнаруживать такой атрибут (о том, как это делается, рассказано в подразделе 2.4.3) и принимать необходимые меры.

Итак, вы научились обращаться с путями и деревьями каталогов. Перейдем от манипуляций с местоположением к осуществлению операций с самой файловой системой. Для этого мы будем использовать новый класс `Files` и сопутствующие элементы.

2.4. Ввод-вывод файловой системы при работе с NIO.2

Поддержка выполнения операций над файловой системой — перемещения файлов, изменения атрибутов файлов, работы с содержимым файлов — в NIO.2 была улучшена. Основной класс, предоставляющий такую поддержку, называется `Files`.

Класс `Files` подробнее описан в табл. 2.2 вместе с другим важным классом, о котором мы поговорим в этом разделе (`WatchService`).

Таблица 2.2. Основные классы для работы с файлами

Класс	Описание
Files	Главный вспомогательный класс содержит все методы, обеспечивающие легкость копирования, перемещения, удаления файлов, а также другие манипуляции
WatchService	Основной класс для отслеживания файлов, каталогов, а также наличия или отсутствия изменений в них

[1] Мы поговорим о BasicFileAttributes в разделе 2.4, так что пока отложим этот вопрос.

В этом разделе мы рассмотрим выполнение таких операций над файловой системой, как:

- создание и удаление файлов;
- перемещение, копирование, переименование и удаление файлов;
- считывание и запись атрибутов файлов;
- считывание информации из файлов и запись информации в них;
- работа с символьными ссылками;
- использование `WatchService` для уведомления об изменении файлов;
- использование `SeekableByteChannel` — усовершенствованного байтового канала, позволяющего указывать и положение и размер.

Масштаб этих изменений может показаться ошеломительным, но API хорошо сработан и содержит множество вспомогательных методов, скрывающих уровни абстрагирования и позволяющих легко и быстро работать с файловыми системами.

ВНИМАНИЕ

Хотя различные API NIO.2 стали значительно лучше поддерживать атомарные операции, при работе с файловой системой по-прежнему требуется относиться к коду с повышенной осторожностью. Даже если операция совершается «на автопилоте», может сорваться сетевое соединение, кто-то может пролить кофе на сервер (такой трагический случай был в практике одного из авторов книги), команда shutdown now может быть выполнена не в том окне UNIX. API действительно выдает исключение RuntimeException от некоторых своих методов, но определенные исключительные случаи можно убрать с помощью вспомогательных методов, например Files.exists(Path).

Отличный способ познакомиться с новым API — почитать написанный для него код и самому попытаться написать такой код. Рассмотрим несколько практических случаев, начиная с самых простых — копирования и удаления файлов.

2.4.1. Создание и удаление файлов

С помощью простых вспомогательных методов из класса `Files` можно без труда и создавать файлы, и удалять их. Разумеется, создание и удаление файлов не всегда оказывается таким простым, как в стандартном случае. Поэтому обсудим и дополнительные варианты. В частности, поговорим об установке прав доступа для чтения/записи к новоиспеченному файлу.

СОВЕТ

Если вы будете запускать фрагменты кода из этого раздела, то заменяйте приведенные пути файлов теми, что актуальны для вашей файловой системы!

Следующий фрагмент кода демонстрирует простейшую операцию создания файла с помощью метода `Files.createFile(Path target)`. Допустим, в вашей операционной системе есть каталог `D:\\Backup`. Создадим в нем файл `MyStuff.txt`.

```
Path target = Paths.get("D:\\Backup\\MyStuff.txt");
Path file = Files.createFile(target);
```

Довольно часто вам понадобится указывать определенные атрибуты `FileAttributes` для этого файла — это делается в целях безопасности. Кроме того, атрибуты определяют, создавался файл для чтения, записи, выполнения или какой-либо комбинации трех вариантов. Поскольку эта характеристика зависит от файловой системы, рекомендуется использовать специфичный для файловой системы класс с правами доступа к файлам.

В качестве примера установки прав чтения/записи для владельца файла, пользователей из группы владельца и всех остальных пользователей файловой системы POSIX[1] рассмотрим следующий код. В принципе, это означает, что мы разрешаем всем пользователям читать информацию из файла `D:\\Backup\\MyStuff.txt` и записывать туда новую информацию. Этот файл еще предстоит создать.

```
Path target = Paths.get("D:\\Backup\\MyStuff.txt");
Set<PosixFilePermission> perms =
    PosixFilePermissions.fromString("rw-rw-rw-");
FileAttribute<Set<PosixFilePermission>> attr =
    PosixFilePermissions.asFileAttribute(perms);
Files.createFile(target, attr);
```

В пакете `java.nio.file.attribute` содержится список имеющихся классов `java.nio.file.attribute`. Более подробно о поддержке атрибутов файлов мы поговорим далее, в подразделе 2.4.3.

ВНИМАНИЕ

При создании файлов с конкретными правами доступа учитывайте любые ограничения umask или «ограничения прав», накладываемые родительским каталогом на этот файл. Например, может оказаться, что, хотя вы и задали для нового файла rw-rw-rw, он на самом деле создается как rw-r--r-- из-за маскировки каталогов.

Удаление файла — более простая операция, которая выполняется с помощью метода `Files.delete(Path)`. Следующий фрагмент кода удаляет только что созданный вами файл по адресу `D:\\Backup\\MyStuff.txt`. Разумеется, пользователь, от имени которого работает ваш процесс Java, должен иметь право на такое удаление!

```
Path target = Paths.get("D:\\Backup\\MyStuff.txt");
Files.delete(target);
```

Далее поговорим о перемещении и копировании файлов в файловой системе.

2.4.2. Копирование и перемещение файлов

С помощью простых вспомогательных методов из класса `Files` вы можете с легкостью совершать операции перемещения и копирования.

В следующем фрагменте кода показана простейшая операция копирования с применением метода `Files.copy(Path source, Path target)`.

```
Path source = Paths.get("C:\\My Documents\\Stuff.txt");
Path target = Paths.get("D:\\Backup\\MyStuff.txt");
Files.copy(source, target);
```

[1] Переносимый интерфейс операционных систем UNIX — базовый стандарт, поддерживаемый во многих операционных системах.

Довольно часто вам будет требоваться снабдить операцию копирования дополнительными параметрами. В следующем примере мы используем параметр, позволяющий перезаписать файл, если он уже существует.

```
import static java.nio.file.StandardCopyOption.*;
Path source = Paths.get("C:\\My Documents\\Stuff.txt");
Path target = Paths.get("D:\\Backup\\MyStuff.txt");
Files.copy(source, target, REPLACE_EXISTING);
```

Среди других параметров копирования следует назвать `COPY_ATTRIBUTES` (копирование файла вместе с его атрибутами) и `ATOMIC_MOVE` (гарантирует успешное завершение обеих сторон операции перемещения — в противном случае происходит откат операции).

Операция перемещения очень похожа на операцию копирования и выполняется с помощью атомарного метода `Files.move(Path source, Path target)`. Опять же, как правило, при перемещении файла вы хотите задать параметры копирования. Поэтому можно воспользоваться методом `Files.move(Path source, Path target, CopyOptions...)` (обратите внимание, как происходит работа с переменным количеством аргументов).

В данном случае мы хотим сохранить при перемещении атрибуты исходного файла, а также перезаписать целевой файл (если он существует).

```
import static java.nio.file.StandardCopyOption.*;

Path source = Paths.get("C:\\My Documents\\Stuff.txt");
Path target = Paths.get("D:\\Backup\\MyStuff.txt");

Files.move(source, target, REPLACE_EXISTING, COPY_ATTRIBUTES);
```

Вот вы и научились создавать, удалять, копировать и перемещать файлы. Теперь подробнее рассмотрим, как в Java 7 поддерживаются атрибуты файлов.

2.4.3. Атрибуты файлов

Атрибуты файла определяют, *что* можно делать с файлом и *кто* может это делать. Классические права типа *что* регламентируют, можно ли совершать с файлом одно или более из следующих действий: считывание файла, запись информации в этот файл либо исполнение файла. Классические права *кто* указывают владельца файла, либо группу владельца, либо всех пользователей.

Начнем этот подраздел с рассмотрения некоторых базовых атрибутов файла, в частности указывающих время последнего обращения к файлу, определяющих, является этот файл каталогом или символьной ссылкой, и т. д. Во второй части этого подраздела поговорим о поддержке атрибутов файлов в конкретных файловых системах. Это не такая простая тема, поскольку в различных файловых системах присутствуют свои наборы атрибутов и собственные интерпретации значений этих атрибутов.

Поддержка базовых атрибутов файлов

Хотя существует мало по-настоящему универсальных атрибутов файлов, есть группа атрибутов, поддерживаемых в большинстве файловых систем. Интерфейс

BasicFileAttributes определяет это общее множество, но вам нужно использовать вспомогательный класс Files для получения ответов на различные вопросы о файле, в частности такие.

- Когда файл в последний раз был изменен?
- Каков размер файла?
- Является ли файл символьной ссылкой?
- Не каталог ли это?

В листинге 2.4 приведены методы класса Files, предназначенные для сбора этих базовых атрибутов. В этом листинге приведена информация о /usr/bin/zip, вы должны получить примерно следующий вывод:

```
/usr/bin/zip
2011-07-20T16:50:18Z
351872
false
false
{lastModifiedTime=2011-07-20T16:50:18Z,
fileKey=(dev=e000002,ino=30871217), isDirectory=false,
lastAccessTime=2011-06-13T23:31:11Z, isOther=false,
isSymbolicLink=false, isRegularFile=true,
creationTime=2011-07-20T16:50:18Z, size=351872}
```

Обратите внимание, что все атрибуты показаны при помощи вызова к Files.readAttributes(Path path, Stringattributes, LinkOption... options).

Листинг 2.4. Универсальные файловые атрибуты

```
try
{
  Path zip = Paths.get("/usr/bin/zip");
  System.out.println(Files.getLastModifiedTime(zip));
  System.out.println(Files.size(zip));
  System.out.println(Files.isSymbolicLink(zip));
  System.out.println(Files.isDirectory(zip));
  System.out.println(Files.readAttributes(zip, "*"));
}
catch (IOException ex)
{
  System.out.println("Exception [" + ex.getMessage() + "]");
}
```

Получаем путь

Выводим атрибуты

Выполняем массовое считывание данных

Есть и другая информация, которая содержится в общих файловых атрибутах и которую можно собрать с помощью методов класса Files. К такой информации относятся, в частности, данные о владельце файла, о том, является ли файл символьной ссылкой, и др.

Можете посмотреть документацию Java (Javadoc) о классе Files, где дан полный список этих вспомогательных методов.

Java 7 также обеспечивает поддержку просмотра атрибутов файлов и манипулирования этими атрибутами в конкретных файловых системах.

Поддержка специфичных атрибутов файлов

Мы уже рассмотрели возможности поддержки атрибутов в Java 7, обеспечиваемые с помощью интерфейса `FileAttribute` и класса `PosixFilePermissions`, когда создавали файл в подразделе 2.4.1. Для поддержки атрибутов, специфичных для определенных файловых систем, Java позволяет производителям этих файловых систем реализовывать интерфейсы `FileAttributeView` и `BasicFileAttributes`.

ВНИМАНИЕ

Мы уже упоминали об этом, но считаем необходимым вновь подчеркнуть: будьте внимательны при написании кода, специфичного для отдельных файловых систем. Обязательно удостоверяйтесь, что предложенная вами логика и механизм обработки исключений охватывают и те случаи, в которых ваш код может запускаться в другой файловой системе.

Рассмотрим следующий пример. Допустим, требуется задать на Java 7 правильные права доступа к определенному файлу. На рис. 2.2 показано содержимое каталога, который является домашним каталогом пользователя Admin. Обратите внимание на особый скрытый файл `.profile`, в котором задано разрешение на запись информации в файл, действующее для пользователя Admin (но ни для кого другого). Здесь же задано право чтения файла для всех остальных пользователей.

В листинге 2.5 мы собираемся обеспечить соблюдение этих прав доступа к файлу `.profile` в соответствии с рис. 2.2. Пользователь (Admin) намерен предоставить всем пользователям право чтения этого файла, но вносить в него изменения может только Admin. Воспользовавшись специальными классами для POSIX, которые называются `PosixFilePermission` и `PosixFileAttributes`, мы можем обеспечить правильность прав доступа (`rw-r--r--`).

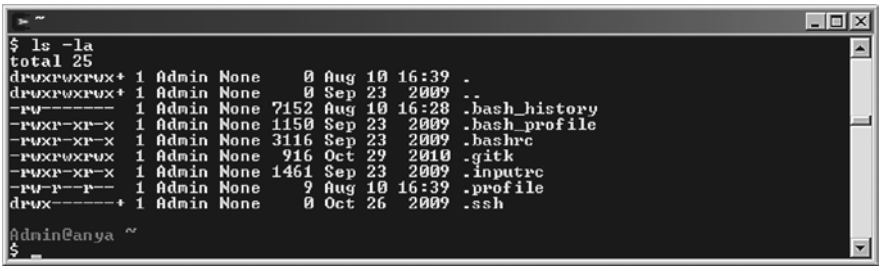


Рис. 2.2. Содержимое домашнего каталога пользователя Admin. Здесь показаны права доступа к файлу .profile

Листинг 2.5. Поддержка атрибутов файлов в Java 7

```
import static java.nio.file.attribute.PosixFilePermission.*;
try
{
  Path profile = Paths.get("/user/Admin/.profile");

  PosixFileAttributes attrs =
    Files.readAttributes(profile,
                         PosixFileAttributes.class);
```

1 Получаем атрибуты

```
  Set<PosixFilePermission> posixPermissions =
                              attrs.permissions();
  posixPermissions.clear();

  String owner = attrs.owner().getName();
  String perms =
    PosixFilePermissions.toString(posixPermissions);
  System.out.format("%s %s%n", owner, perms);

  posixPermissions.add(OWNER_READ);
  posixPermissions.add(GROUP_READ);
  posixPermissions.add(OTHER_READ);
  posixPermissions.add(OWNER_WRITE);
  Files.setPosixFilePermissions(profile, posixPermissions);
}
catch(IOException e)
{
  System.out.println(e.getMessage());
}
```

Начинаем работу с импорта констант `PosixFilePermission` (а также других операций импорта, не показанных здесь), потом получаем путь `Path` к файлу `.profile`. В классе `Files` есть полезный вспомогательный метод, позволяющий считывать те атрибуты, которые специфичны для файловой системы, в данном случае — `PosixFileAttributes` ❶. Затем вы можете получить доступ к `PosixFilePermission` ❷. После очистки имеющихся прав доступа ❸ вы добавляете к файлу новые права доступа — опять же с помощью полезного вспомогательного метода `Files` ❹.

Вы, вероятно, заметили, что `PosixFilePermission` относится к типу `enum` и поэтому не реализует интерфейс `FileAttributeView`. Итак, почему же здесь не используется реализация `PosixFileAttributeView`? Так ведь на самом деле она используется! Но вспомогательный класс `Files` скрывает от вас эту абстракцию, позволяя непосредственно считывать атрибуты файла (с помощью метода `readAttributes`) и напрямую задавать права доступа (с помощью метода `setPosixFilePermissions`).

Кроме базовых атрибутов, Java 7 предоставляет расширяемую систему поддержки специальных функций операционной системы. К сожалению, в рамках этой главы мы не можем обсудить все конкретные случаи, но мы рассмотрим один пример работы с данной системой — поддержку символьных ссылок в Java 7.

Символьные ссылки

Символьную ссылку можно считать указателем на другой файл или каталог. В большинстве случаев работа с символьными ссылками происходит прозрачно. Например, если обозначить каталог через символьную ссылку, то при щелчке на ссылке вы попадете в каталог, на который указывает эта ссылка. Но если вы пишете программу — скажем, утилиту для резервного копирования или сценарий для развертывания, то нужно иметь возможность принимать оправданные решения

о том, переходить ли по символьной ссылке. В NIO.2 такая возможность предоставляется.

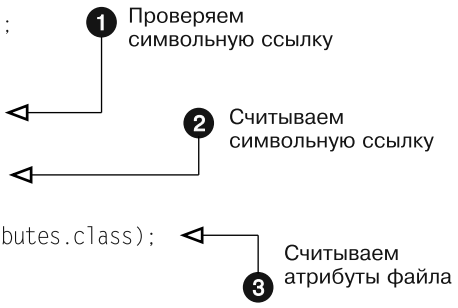
Вновь воспользуемся примером из подраздела 2.2.3. Предположим, что вы работаете в операционной системе *nix и ищете информацию о файле журнала (лога), называемом `log1.txt` и находящемся в каталоге `/usr/logs`. Но на самом деле каталог `/usr/logs` является ссылкой (указателем) на каталог `/application/logs`. Именно там *в действительности* находится интересующий нас файл журнала.

Символьные ссылки применяются во множестве операционных систем, в частности (но не только) в UNIX, Linux, Windows 7 и Mac OS X. Поддержка символьных ссылок в Java 7 основывается на семантике той реализации, которая применяется в операционной системе UNIX.

В листинге 2.6 мы проверяем, является ли путь `Path` к версии Java, установленной в каталоге `/opt/platform`, символьной ссылкой. Уже после этого мы пытаемся считать базовые атрибуты файла. Мы собираемся считывать атрибуты именно из того места, где *действительно* находится файл.

Листинг 2.6. Исследование символьных ссылок

```
Path file = Paths.get("/opt/platform/java");
try
{
  if(Files.isSymbolicLink(file))
  {
    file = Files.readSymbolicLink(file);
  }
  Files.readAttributes(file, BasicFileAttributes.class);
}
catch (IOException e)
{
  System.out.println(e.getMessage());
}
```



Класс `Files` предоставляет метод `isSymbolicLink(Path)` (1) для проверки символьных ссылок. У него есть вспомогательный метод для возвращения реального пути `Path`, являющегося целью символьной ссылки (2). Так вы можете считывать правильные атрибуты файла (3).

По умолчанию в API NIO.2 выполняется следование по символьным ссылкам. Чтобы отключить это, нужно применить `LinkOption.NOFOLLOW_LINKS`. Этот параметр применим для нескольких вызовов методов. Если вы хотите прочитать базовые атрибуты файла, соответствующие самой символьной ссылке, то нужно вызвать:

```
Files.readAttributes(target,
BasicFileAttributes.class,
LinkOption.NOFOLLOW_LINKS);
```

Символьные ссылки — наиболее популярный пример того, как в Java 7 поддерживаются функции конкретных файловых систем. Структура API позволяет в будущем добавлять и поддержку других функций, специфичных для определенной

файловой системы (даже для той суперсекретной квантовой файловой системы, над созданием которой вы тайком трудитесь по ночам).

Теперь, когда вы научились практике обращения с файлами, приступим к манипуляциям с их содержимым.

2.4.4. Быстрое считывание и запись данных

Одна из целей Java 7 — предоставить максимально возможное количество вспомогательных методов для считывания файлов и записи информации в них. Разумеется, такие новые методы используют местоположения `Path`, но разработчики позаботились и о взаимодействии со старыми потоковыми классами из пакета `java.io`. В сухом остатке имеем, что такие задачи, как считывание всех строк (всех байтов) из файла, выполняются путем вызова всего одного метода.

В этом разделе мы рассмотрим процесс открытия файлов (со всеми сопутствующими параметрами), а также группу небольших примеров, описывающих обычные случаи считывания и записи информации. Для начала обсудим, какими различными способами можно открывать файл для обработки.

Открытие файлов

Java 7 позволяет напрямую открывать файлы для обработки с помощью буферизованных считывателей и записывателей либо (для совместимости со старым кодом Java для ввода-вывода) с помощью потоков для ввода и вывода. В следующем фрагменте кода показано, как в Java 7 можно (с помощью метода `Files.newBufferedReader`) открыть файл для считывания строк.

```
Path logFile = Paths.get("/tmp/app.log");
try (BufferedReader reader =
     Files.newBufferedReader(logFile, StandardCharsets.UTF_8)) {
  String line;
  while ((line = reader.readLine()) != null) {
    ...
  }
}
```

Открыть файл для записи не сложнее:

```
Path logFile = Paths.get("/tmp/app.log");
try (BufferedWriter writer =
     Files.newBufferedWrite(logFile, StandardCharsets.UTF_8,
                            StandardOpenOption.WRITE)) {
  writer.write("Hello World!");
  ...
}
```

Обратите внимание на использование `StandardOpenOption.WRITE`. Это один из параметров `OpenOption`, который можно добавить в вызов с переменным количеством аргументов. Так вы гарантируете наличие верных прав доступа, регламентирующих запись информации в файл. К другим распространенным параметрам относятся, в частности, `READ` и `APPEND`.

Взаимодействие с потоками `InputStream` и `OutputStream` обеспечивается с помощью специальных методов `Files.newInputStream(Path, OpenOption...)` и `Files.newOutputStream(Path, OpenOption...)`. Так удается удачно объединить старый ввод-вывод, построенный вокруг пакета `java.io`, и новый файловый ввод-вывод, в основе которого лежит пакет `java.nio`.

СОВЕТ

Не забывайте, что при работе со строками String всегда требуется знать их кодировку. Если вы забудете указать кодировку (это делается с помощью класса StandardCharsets, например, так: new String(byte[], StandardCharsets .UTF_8)), то позже могут возникнуть неожиданные проблемы, связанные именно с ней.

В предыдущих примерах показан тривиальный код для считывания файлов и записи информации в них. Сегодня он часто используется в Java 6 и более старых версиях. Все это — довольно сложный низкоуровневый код, а Java 7 предлагает красивые высокоуровневые абстракции, позволяющие избавиться от множества ненужного шаблонного кода.

Упрощение записи и считывания

Во вспомогательном классе `Files` есть несколько полезных методов, выполняющих обычные задачи считывания всех строк кода или всех байтов из файла. Это означает, что теперь вы можете обойтись без написания шаблонного кода, не считывать в буфер байтовые массивы данных с применением цикла `while`. В следующем фрагменте кода показано, как вызывать вспомогательные методы.

```
Path logFile = Paths.get("/tmp/app.log");
List<String> lines = Files.readAllLines(logFile, StandardCharsets.UTF_8);
byte[] bytes = Files.readAllBytes(logFile);
```

При программировании некоторых ситуаций требуется знать, когда выполнять считывание и когда — запись. Особенно это касается файлов свойств или файлов журнала (логов). Именно здесь может очень пригодиться система уведомлений о внесении новых изменений в файл.

2.4.5. Уведомление об изменении файлов

Java 7 позволяет отслеживать внесение изменений в файл или каталог. Это делается с помощью класса `java.nio.file.WatchService`. Этот класс использует клиентские потоки для отслеживания изменений в зарегистрированных файлах или каталогах. При обнаружении изменения класс возвращает соответствующее событие. Подобное уведомление о событиях может пригодиться при мониторинге безопасности, обновлении информации из файла свойств, а также во многих других случаях. Подобные уведомления идеально подходят для замены сравнительно более медленных механизмов опроса, используемых в некоторых современных приложениях.

В листинге 2.7 служба `WatchService` используется для обнаружения любых изменений в домашнем каталоге пользователя karianna и для вывода информации о таком событии изменения на консоль. Как и в других случаях, в которых

используется непрерывный опрашивающий цикл, всегда стоит добавлять в код легковесный механизм для отключения такого цикла.

Листинг 2.7. Использование WatchService

```
import static java.nio.file.StandardWatchEventKinds.*;
try
{
  WatchService watcher =
    FileSystems.getDefault().newWatchService();

  Path dir =
    FileSystems.getDefault().getPath("/usr/karianna");

  WatchKey key = dir.register(watcher, ENTRY_MODIFY);

  while(!shutdown)
  {
    key = watcher.take();
    for (WatchEvent<?> event: key.pollEvents())
    {
      if (event.kind() == ENTRY_MODIFY)
      {
        System.out.println("Home dir changed!");
      }
    }
    key.reset();
  }
}
catch (IOException | InterruptedException e)
{
  System.out.println(e.getMessage());
}
```

Получив задаваемую по умолчанию службу WatchService, вы регистрируете отслеживание изменений для домашнего каталога пользователя karianna (1). Затем в бесконечном цикле (он продолжается до тех пор, пока не изменится флаг shutdown) (2) к службе WatchService применяется метод take(), дожидающийся, пока будет доступен ключ WatchKey. Как только WatchKey оказывается доступен, код опрашивает этот ключ WatchKey на наличие событий WatchEvents (3). Если обнаруживается событие WatchEvent рода Kind ENTRY_MODIFY (4), то вы сообщаете об этом «во всеуслышание». Наконец, остается еще сбросить ключ (5), чтобы он был готов к приему следующего события.

Вы можете отслеживать и события других видов, например ENTRY_CREATE, ENTRY_DELETE и OVERFLOW (последний случай может означать, что событие было потеряно или отменено).

Далее рассмотрим очень важный новый API абстрагирования, предназначенный для считывания и записи данных. Здесь задействуется асинхронный ввод-вывод с применением интерфейса SeekableByteChannel.

2.4.6. SeekableByteChannel

В Java 7 появился интерфейс `SeekableByteChannel`. Он предназначен для дополнения такими реализациями, которые позволяют разработчикам изменять положение и размер байтового канала. Например, у вас может быть сервер приложений с множеством потоков. Этот сервер получает доступ к байтовому каналу, прикрепленному к большому файлу журналов. После такого доступа сервер может выполнять синтаксический анализ журнала на наличие конкретного кода ошибки.

У интерфейса `java.nio.channels.SeekableByteChannel` в JDK есть один реализующий класс — `java.nio.channels.FileChannel`. В нем вы можете содержать информацию о точной позиции того места, из которого считываете информацию или записываете ее в файл. Например, вам может понадобиться код, считывающий последние 1000 символов из файла журнала. Другой пример — возможно, потребуется написать код, записывающий определенную ценовую информацию в конкретное место в текстовом файле.

В следующем фрагменте показано, как можно использовать новые свойства `FileChannel`, обеспечивающие возможность поиска (так называемые seekable-свойства) для считывания последних 1000 символов из файла журнала.

```
Path logFile = Paths.get("c:\\temp.log");
ByteBuffer buffer = ByteBuffer.allocate(1024);
FileChannel channel = FileChannel.open(logFile, StandardOpenOption.READ);
channel.read(buffer, channel.size() - 1000);
```

Новая возможность поддержки поиска, появившаяся в классе `FileChannel`, теоретически означает, что у разработчиков значительно расширяется поле для маневра при работе с содержимым файлов. Ожидаемо, что в ближайшем будущем появится несколько интересных открытых проектов, использующих такой параллельный доступ к крупным файлам. Учитывая возможности расширения данного интерфейса, не исключено, что появится и возможность работы с непрерывными потоками сетевых данных.

Следующее крупное изменение, появившееся в API NIO.2 и связанное с появлением асинхронного ввода-вывода, позволяет вам использовать множественные фоновые потоки при считывании файлов, сокетов и каналов, а также при записи информации в них.

2.5. Асинхронные операции ввода-вывода

Еще одно значительное нововведение в NIO.2 — возможность асинхронного ввода-вывода при работе как с сокетами, так и с файлами. Асинхронный ввод-вывод — это просто разновидность обработки ввода-вывода, при которой еще до завершения записи или считывания может выполняться другое действие. На практике это означает, что вы можете пользоваться преимуществами новейших наработок в области аппаратного и програмного обеспечения. К ним относятся, в частности, многоядерные процессоры. Кроме того, поддерживается обработка файлов и сокетов на уровне операционной системы. Асинхронный ввод-вывод — это важнейшая

предпосылка для масштабируемости и производительности системы в любом языке, который пытается сохранить позиции в области серверного и системного программирования. Мы считаем, что этот фактор сыграет одну из важнейших ролей в сохранении популярности Java как серверного языка.

Рассмотрим простой практический пример. Нужно записать 100 Гбайт данных в файловую систему или сетевой сокет. В предыдущих версиях Java вам пришлось бы вручную писать многопоточный код (с использованием конструкций `java.util.concurrent`). Так мы смогли бы одновременно записывать информацию в несколько областей этого файла или сокета. Кроме того, не существовало простой возможности одновременно считывать информацию из нескольких частей файла. Опять же, если вы только не написали вручную какой-то умный код, основной поток блокировался при операциях ввода-вывода. Таким образом, приходилось дожидаться завершения операции ввода-вывода, которая могла протекать очень долго, и лишь потом возвращаться к выполнению основной работы.

СОВЕТ

Если в последнее время вам не доводилось работать с каналами нового ввода-вывода, то самое время освежить знания по предмету и лишь потом возвращаться к чтению этой главы. На эту тему давно не выходило значительных работ. В качестве вводного пособия рекомендуем использовать немного устаревшую книгу Рона Хитченса (Ron Hitchens) Java NIO (издательство O'Reilly, 2002).

В Java 7 появилось три новых асинхронных канала, на которые следует обратить внимание:

- `AsynchronousFileChannel` — для файлового ввода-вывода;
- `AsynchronousSocketChannel` — для сокетного ввода-вывода, поддерживает задержки;
- `AsynchronousServerSocketChannel` — для асинхронных сокетов, принимающих соединения.

Существуют две основные парадигмы (стиля) использования новых API асинхронного ввода-вывода: парадигма *Future* (с ожиданием) и парадигма *Callback* (обратный вызов). Интересно отметить, что эти новые асинхронные API используют некоторые современные технологии параллельной обработки, которые мы рассмотрим в главе 4. Поэтому считайте, что сейчас мы устраиваем небольшой предварительный сеанс!

Для начала поговорим о Future-стиле доступа к файлам, который назовем «с ожиданием». Надеемся, что вам уже доводилось применять такую технологию параллельной обработки, но если нет — не отчаивайтесь. В следующем разделе эта методика описана достаточно подробно, чтобы интересующийся разработчик мог с ней разобраться.

2.5.1. Стиль с ожиданием

Стиль с ожиданием (Future style) — термин, предложенный самими разработчиками API NIO.2. Он указывает на использование интерфейса `java.util.concurrent.Future`. Как правило, стиль асинхронной обработки с ожиданием применяется в тех слу-

чаях, когда вы хотите, чтобы главный поток управления инициировал ввод-вывод, а потом собирал результаты ввода-вывода в ходе опроса.

При стиле с ожиданием используется известная техника `java.util.concurrent`. В таком случае объявляется объект `Future`, в котором будет храниться результат вашей асинхронной операции. Самое важное — это означает, что работа вашего актуального потока не будет тормозиться из-за потенциально медленной операции ввода-вывода. Напротив, операция ввода-вывода инициируется в отдельном потоке. По завершении операции возвращается ее результат. Тем временем ваш основной поток может и далее заниматься выполнением других задач, если это необходимо. Когда эти прочие задачи будут завершены, основной поток дождется завершения операции ввода-вывода и потом продолжит работу. На рис. 2.3 продемонстрировано, как этот процесс применяется при считывании крупного файла (в листинге 2.8 показан код, используемый в такой ситуации).

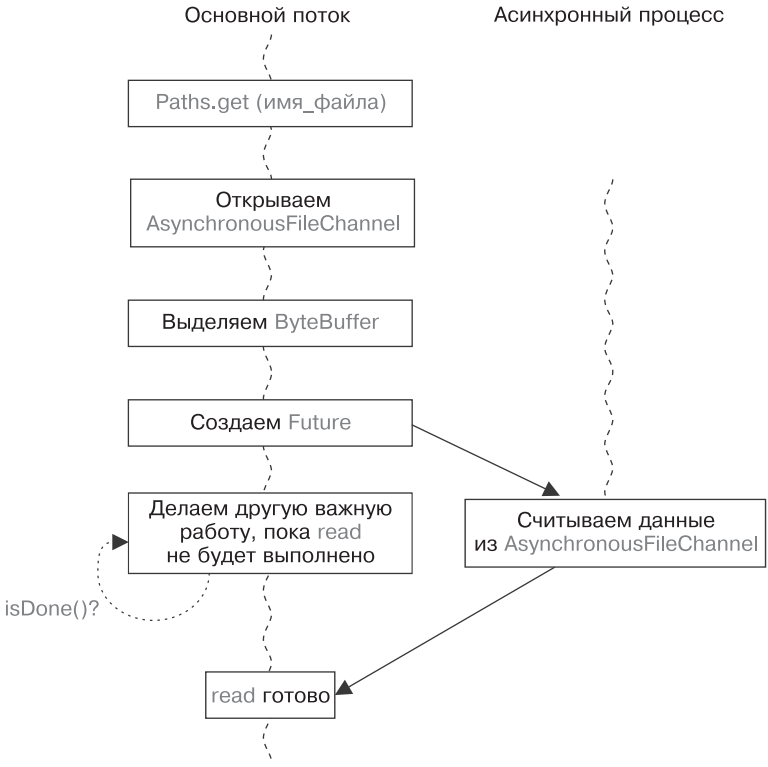


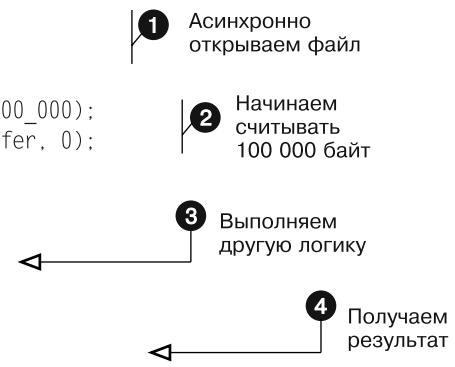
Рис. 2.3. Асинхронное считывание в стиле с ожиданием

Как правило, вы будете пользоваться методом `Future get()` (с параметром задержки или без него) для получения результата после завершения такой асинхронной операции ввода-вывода. Допустим, вы хотите считать 100 000 байт из файла на диске (это достаточно медленная операция) в рамках выполнения другой задачи. В предыдущих версиях Java пришлось бы дожидаться, пока

эта операция считывания не закончится (конечно, если вы не применяете пул потоков и рабочие потоки — для этого требуется использовать блоки `java.util.concurrent`, что уже является нетривиальной задачей). В Java 7 вы можете продолжать заниматься полезной работой в вашем основном потоке. Это показано в листинге 2.8.

Листинг 2.8. Асинхронный ввод-вывод в стиле с ожиданием

```
try
{
  Path file = Paths.get("/usr/karianna/foobar.txt");

  AsynchronousFileChannel channel =
  AsynchronousFileChannel.open(file);

  ByteBuffer buffer = ByteBuffer.allocate(100_000);
  Future<Integer> result = channel.read(buffer, 0);

  while(!result.isDone())
  {
    ProfitCalculator.calculateTax();
  }

  Integer bytesRead = result.get();
  System.out.println("Bytes read [" + bytesRead + "]");
}
catch (IOException | ExecutionException | InterruptedException e)
{
  System.out.println(e.getMessage());
}
```



Начинаем работу с открытия канала `AsynchronousFileChannel` для считывания или записи информации в файл `foobar.txt`. Такое считывание или запись будет выполняться в фоновом потоке ❶. Далее ввод-вывод будет проходить параллельно с потоком, который его инициировал. Такой параллельный процесс ввода-вывода происходит автоматически, поскольку вы используете `AsynchronousFileChannel`. Результаты считывания будут сохраняться в объекте `Future` ❷. Пока это происходит, ваш основной поток продолжает выполнять полезную работу (например, рассчитывать налог) ❸. Наконец, после завершения работы вы проверяете результат такого считывания ❹.

Отметим, что мы искусственно гарантировали обязательное получение конечного результата (добавив в код `isDone()`). На практике конечный результат либо будет получен (и основной поток продолжит работу), либо придется дожидаться, пока фоновый ввод-вывод завершится.

Возможно, вас интересует, как весь этот процесс организован «за кулисами». Если говорить вкратце, API/JVM предоставляет пулы потоков и группы каналов для выполнения задачи. Вы также можете предоставить и сконфигурировать такие элементы самостоятельно. Детали процесса требуют дополнительного разъяснения,

но все они подробно описаны в официальной документации. Поэтому просто процитируем фрагмент Javadoc по `AsynchronousFileChannel`.

Канал AsynchronousFileChannel ассоциирован с пулом потоков, в который подаются задачи для обработки событий ввода-вывода и диспетчеризации результатов к обработчикам завершения (обработчики завершения принимают на канале результаты операций ввода-вывода). Обработчик завершения для операции ввода-вывода, инициированной на канале, гарантированно активизируется одним из потоков, относящихся к пулу потоков.

При создании AsynchronousFileChannel без указания пула потоков этот канал будет ассоциирован с системозависимым пулом потоков, заданным по умолчанию (который, возможно, будет использоваться совместно с другими каналами). Задаваемый по умолчанию пул потоков конфигурируется с помощью системных свойств, определяемых в классе AsynchronousChannelGroup.

Существует и альтернативная техника, называемая Callback (обратный вызов). Некоторые разработчики считают стиль с обратными вызовами более удобным, чем стиль с ожиданием, так как он напоминает методики обработки событий, с которыми приходилось иметь дело в `Swing`, при обмене сообщениями, а также в ходе использования других API Java.

2.5.2. Стиль с применением обратных вызовов

В отличие от стиля с ожиданием, стиль с применением обратных вызовов технологически похож на применение обработчиков событий. Такая техника может быть вам знакома из программирования пользовательских интерфейсов с помощью Swing. Главная идея заключается в том, что основной поток отправит «разведчика» (обработчик завершения `CompletionHandler`) в отдельный поток, выполняющий операцию ввода-вывода. Разведчик получит результат операции ввода-вывода, на основании которого будет запущен собственный метод этого разведчика `completed` или `failed` (переопределяемый вами). После этого разведчик вернется к основному потоку.

Обычно такой стиль применяется в тех случаях, когда необходимо немедленно реагировать на изменение ситуации после успешности или неуспешности выполнения того или иного асинхронного события. Например, вы считываете финансовую информацию, которая совершенно необходима для использования в бизнес-процессе, связанном с расчетом прибыли. И вдруг это считывание срывается. В таком случае вам немедленно потребуется выполнить возврат или обработать исключение.

В более строгой форме ситуация описывается так. Интерфейс `java.nio.channels.CompletionHandler<V, A>` (где `V` — тип результата, а `A` — прикрепляемый объект, от которого вы получаете результат) активизируется после того, как будет выполнена асинхронная операция, связанная с вводом-выводом. Методы `completed(V, A)` и `failed(V, A)` этого интерфейса должны быть реализованы так, чтобы ваша программа четко различала случаи успешного и неуспешного завершения ввода-вывода и правильно на них реагировала. Процесс показан на рис. 2.4 (в листинге 2.9 приведен код, реализующий этот процесс).

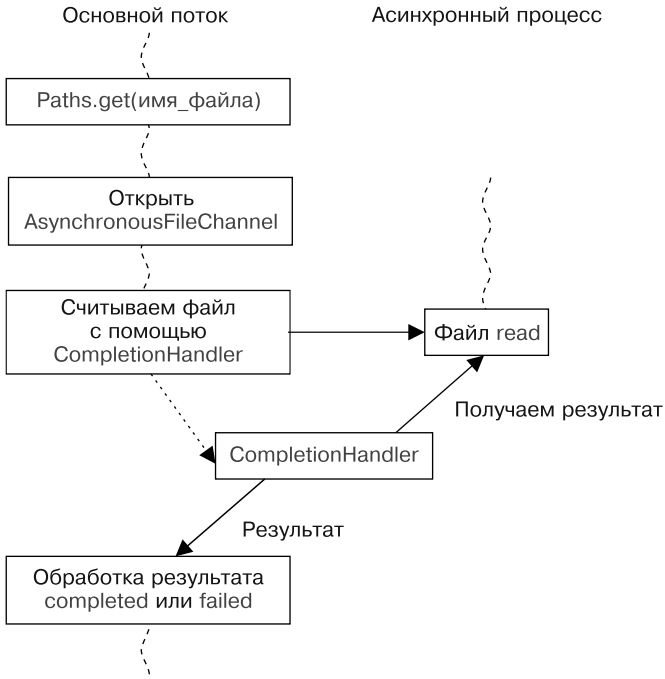


Рис. 2.4. Асинхронное считывание с применением обратного вызова

В следующем примере мы вновь считываем 100 000 байт из `foobar.txt`. Для объявления успешности или неуспешности операции используем `CompletionHandler <Integer, ByteBuffer>`.

Листинг 2.9. Асинхронный ввод-вывод — стиль с применением обратных вызовов

```
try
{
  Path file = Paths.get("/usr/karianna/foobar.txt");
  AsynchronousFileChannel channel =                          | Открываем
    AsynchronousFileChannel.open(file);                      | асинхронный файл

  ByteBuffer buffer = ByteBuffer.allocate(100_000);

  channel.read(buffer, 0, buffer,                            | Выполняем
    new CompletionHandler<Integer, ByteBuffer>()             | считывание из канала
  {
    public void completed(Integer result,
                          ByteBuffer attachment)             | Обратный вызов,
                                                             | выполняемый
    {                                                        | по завершении
      System.out.println("Bytes read [" + result + "]");     | считывания
    }

    public void failed(Throwable exception, ByteBuffer attachment)
```

```
    {
      System.out.println(exception.getMessage());
    }
  });
}
catch (IOException e)
{
  System.out.println(e.getMessage());
}
```

Два листинга, приведенные в этом разделе, описывали работу с файлами, но стили асинхронного доступа с ожиданием и с обратным вызовом также могут применяться к `AsynchronousServerSocketChannel` и `AsynchronousSocketChannel`. Таким образом, разработчики могут писать приложения, работающие с сетевыми сокетами (например, для IP-телефонии) и создавать более производительные клиентские и серверные программы.

Далее обсудим несколько небольших изменений, объединяющих сокеты и каналы и позволяющих вам иметь в API единую точку контакта для управления сокетными и канальными взаимодействиями.

2.6. Окончательная шлифовка технологии сокет — канал

В наше время компьютерные программы как никогда нуждаются в доступе к сети. Кажется, что в недалеком будущем любой утюг сможет подключаться к ней (а может быть, такие утюги уже есть!). В прежних версиях Java программные конструкции `Socket` и `Channel` не очень хорошо взаимодействовали друг с другом — любые попытки их объединить были неудачными. Java 7 несколько упрощает для разработчиков обращение с каналами и сокетами, так как вводит сущность `NetworkChannel`, сочетающую черты `Socket` и `Channel`.

Написание низкоуровневого сетевого кода — достаточно узкоспециализированная область. Если вы не работаете в ней, то этот раздел определенно можете пропустить. Но если эта сфера как раз относится к вашим профессиональным интересам, продолжайте читать — здесь мы сделаем обзор новых возможностей.

Для начала вспомним, какую роль сокеты и каналы играют в Java. Возьмем определения из Javadoc.

Пакет java.nio.channels. Определяет каналы, представляющие собой соединения с сущностями, способными выполнять операции ввода-вывода, — например, с файлами и сокетами; определяет селекторы для мультиплексированных операций неблокирующего ввода-вывода.

Класс java.net.Socket. Реализует клиентские сокеты (также называемые просто сокетами). Сокет — это конечная точка для обмена информацией между двумя машинами.

В старых версиях Java вы действительно пытались привязать канал к реализации `Socket`, чтобы выполнять ту или иную операцию ввода-вывода, например записывать

данные через TCP-порт. Но при объединении Channel и Socket появлялись серьезные проблемы.

- В старых версиях Java приходилось смешивать сокетные и канальные API, чтобы управлять параметрами сокетов и выполнять связывание на сокетах.
- В старых версиях Java не было возможности пользоваться платформоспецифичным сокетным поведением.

Рассмотрим, какие работы были проведены в рамках окончательной шлифовки нового интерфейса NetworkChannel, а также его подинтерфейса — MulticastChannel.

2.6.1. NetworkChannel

Новый интерфейс java.nio.channels.NetworkChannel представляет собой ассоциирование канала с сетевым сокетом. Он определяет группу полезных методов, которые, в частности, позволяют просмотреть доступные параметры сокета и устанавливать его новые параметры на данном канале. В следующем листинге показаны эти вспомогательные методы. Мы демонстрируем их, отображая параметры, которые поддерживает интернет-адрес сокета на порте 3080. Кроме того, мы устанавливаем IP «Условия обслуживания» и идентифицируем параметр SO_KEEPALIVE на соответствующем сокетном канале.

Листинг 2.10. Параметры NetworkChannel

```
SelectorProvider provider = SelectorProvider.provider();
try
{
  NetworkChannel socketChannel =
                       provider.openSocketChannel();
  SocketAddress address = new InetSocketAddress(3080);
  socketChannel = socketChannel.bind(address);

  Set<SocketOption<?>> socketOptions =
    socketChannel.supportedOptions();
  System.out.println(socketOptions.toString());

  socketChannel.setOption(StandardSocketOptions.IP_TOS,
                          3);
  Boolean keepAlive =
    socketChannel.getOption(StandardSocketOptions.SO_KEEPALIVE);
  ..
  ..
}
catch (IOException e)
{
  System.out.println(e.getMessage());
}
```

Связываем NetworkChannel с портом 3080

Проверяем параметры сокета

Задаем параметр сокета «Условия обслуживания»

Получаем параметр SO_KEEPALIVE

Еще одно дополнение, обеспечиваемое функцией NetworkChannel, — это многоадресные операции.

2.6.2. MulticastChannel

Возможность многоадресной передачи — обычное практическое требование, предъявляемое к приложениям для одноранговых сетей, например к BitTorrent. В более ранних версиях Java можно было кое-как сварганить многоадресную реализацию, но среди API Java не было красивой абстракции, предназначенной специально для этого. В Java 7 для решения такой проблемы вводится новый интерфейс `MulticastChannel`.

Термин *«многоадресный»* описывает передачу информации по сети от одного узла ко многим. Часто это делается на основе интернет-протокола (IP). Суть процесса заключается в том, что вы отправляете пакет на групповой адрес и приказываете сети размножать этот пакет столько раз, сколько необходимо для предоставления его копии всем получателям, зарегистрированным в этой многоадресной группе. Пример проиллюстрирован на рис. 2.5.

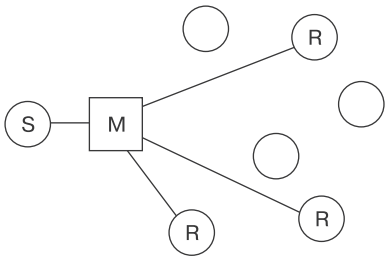


Рис. 2.5. Пример многоадресной передачи

Для того чтобы новые `NetworkChannel` могли поддерживать присоединение к многоадресным группам, появился новый интерфейс `java.nio.channels.MulticastChannel`. По умолчанию его реализует класс под названием `DatagramChannel`. Таким образом, вы с легкостью можете выполнять многоадресную передачу информации и получать информацию от многоадресных групп.

В следующем гипотетическом примере мы отсылаем к многоадресной группе системные статусные сообщения и получаем сообщения от этой группы, присоединяясь к ней на IP-адресе 180.90.4.12 (листинг 2.11).

Листинг 2.11. Параметры NetworkChannel

```
try
{
  NetworkInterface networkInterface =
  NetworkInterface.getByName("net1");

  DatagramChannel dc =
  DatagramChannel.open(StandardProtocolFamily.INET);

  dc.setOption(StandardSocketOptions.SO_REUSEADDR, true);
  dc.bind(new InetSocketAddress(8080));
  dc.setOption(StandardSocketOptions.IP_MULTICAST_IF,
               networkInterface);

  InetAddress group = InetAddress.getByName("180.90.4.12");
  MembershipKey key = dc.join(group, networkInterface);
}
catch (IOException e)
{
  System.out.println(e.getMessage());
}
```

Выбираем сетевой интерфейс

Открываем DatagrammChannel

Задаем для канала многоадресную передачу

Присоединяемся к многоадресной группе

На этом завершается наше первичное исследование новых API NIO.2. Надеемся, вам понравился этот летучий рейд!

2.7. Резюме

Аппаратный и программный ввод-вывод в наше время стремительно развивается. Язык Java 7 позволяет с максимальной пользой задействовать современные возможности благодаря новым API, появившимся в рамках NIO.2. Новые библиотеки Java 7 позволяют оперировать местоположениями (`Path`) и выполнять операции в файловой системе — в частности, управлять файлами, каталогами, символьными ссылками и пр. Например, сущим пустяком теперь стала навигация по файловым системам с полной поддержкой платформоспецифичных поведений. Гораздо проще стало работать с крупными структурами, состоящими из множества каталогов.

NIO.2 создавался так, чтобы предоставить вам универсальные методы для выполнения задач. Ранее для их решения пришлось бы писать довольно объемный код. В частности, новый служебный класс `Files` предоставляет множество вспомогательных методов, значительно ускоряющих написание кода для файлового ввода-вывода. Кроме того, весь процесс упрощается по сравнению со старым вводом-выводом на основе `java.io.File`.

Асинхронный ввод-вывод — это мощная новая возможность для работы с большими файлами без резкого снижения производительности. Кроме того, такой новый ввод-вывод удобен при работе с приложениями, пропускающими большие объемы трафика через сетевые сокеты и каналы.

NIO.2 также использует наработки Java, объединенные проектом «Монета» (см. главу 1), модифицированные для операций ввода-вывода. Благодаря этому операции ввода-вывода становятся в Java 7 значительно надежнее, чем раньше, и вы можете писать менее пространный код.

Теперь прибавим оборотов и перейдем ко второй части нашей книги. Здесь нам придется покорпеть над некоторыми продвинутыми функциями современного языка Java. Мы поговорим о внедрении зависимостей, современной параллельной обработке, а также о настройке программной системы, основанной на Java. Советуем заварить любимого кофе в кружке Дюка[1], подзаправиться и двигаться дальше!

[1] Кстати, Дюк — это талисман Java! http://kenai.com/projects/duke/pages/Home

ЧАСТЬ 2

Необходимые технологии

В этой части книги (главы 3–6) мы углубимся в изучение важнейших тем и технологий, необходимых для работы с современным языком Java.

Начнем с главы о внедрении зависимостей (dependency injection). Это распространенная технология, позволяющая уменьшить связанность кода, оптимизировать его тестируемость и восприятие. Мы не только расскажем об основах внедрения зависимостей, но и рассмотрим, как развивалась эта технология, как оптимальные практики разработки превратились в паттерн (шаблон) проектирования, а потом — и во фреймворк (и даже в стандарт Java).

Далее нам предстоит разобраться с проблемой многоядерной революции в производстве процессоров — это важнейшее явление в производстве аппаратного обеспечения. Основательный Java-разработчик обязательно должен разбираться в возможностях параллельной обработки языка Java, уметь их применять для максимально эффективного использования современных процессоров. Несмотря на то что серьезная поддержка параллельной обработки существует в Java уже с 2006 года (Java 5), эта область недостаточно хорошо изучена, и параллельная обработка задействуется довольно неэффективно. Поэтому мы посвятим данной теме целую главу.

Вам предстоит изучить действующую в Java модель памяти (Java Memory Model), а также реализацию потоков и параллелизма в этой модели. Вооружившись этим теоретическим материалом, мы займемся исследованием возможностей пакета `java.util.concurrent` (и не только), после чего станем нарабатывать практические навыки параллельной обработки на Java.

Затем поговорим о загрузке классов. Многие разработчики Java не вполне понимают, как именно виртуальная машина Java загружает, связывает и проверяет классы. Из-за этого растет раздражение и впустую тратится время, так как из-за какого-то конфликта загрузчиков классов может выполниться «неправильная» версия класса.

Наконец, мы поговорим о сущностях Java 7 `MethodHandle`, `MethodType` и `invokedynamic` и покажем разработчикам, привыкшим пользоваться рефлексией, как можно решать знакомые задачи более эффективными и безопасными способами.

Умея внимательно читать и глубоко понимать содержимое файлов классов Java и содержащегося в них байт-кода, вы приобретаете замечательные навыки отладки. Мы покажем, как использовать `javap` для навигации, и научим вас понимать значение байт-кода.

Отладка производительности часто воспринимается как искусство, а не как наука. Отслеживание и устранение проблем с производительностью часто требуют от команды разработчиков исключительных усилий, а также немало времени. В главе 6, завершающей эту часть, мы расскажем, как измерять производительность, а не гадать о ней. Мы докажем, что «танцы с бубном» — это неверный метод. Мы продемонстрируем научный подход, который позволит быстро заглянуть в суть ваших проблем с производительностью.

В частности, мы обратим особое внимание на сборку мусора (garbage collection) и динамическую компиляцию (JIT) — две основные составляющие виртуальной машины Java, оказывающие влияние на производительность. Исследуя проблемы

производительности, мы научимся читать журналы сборщика мусора и работать с бесплатным инструментом Java VisualVM (jvisualvm), помогающим анализировать использование памяти.

Дочитав эту часть, вы уже не будете обычным разработчиком, который сидит в IDE и думает только об исходном коде. Вы будете знать, как именно функционируют язык Java и виртуальная машина Java, а также сможете в полную силу использовать, пожалуй, самую мощную универсальную виртуальную машину, существующую на нашей планете.

3 Внедрение зависимостей

В этой главе:

- инверсия управления (IoC) и внедрение зависимостей (DI);
- почему так важно освоить технику внедрения зависимостей;
- как документ JSR-330 объединил внедрение зависимостей для Java;
- типичные аннотации JSR-330, например `@Inject`;
- Guice 3, эталонная реализация для JSR-330.

Внедрение зависимостей (форма инверсии управления) — важная парадигма программирования, которая примерно с 2004 года относится к основному направлению разработки для Java. Вкратце: внедрение зависимостей — это технология, при применении которой ваш объект получает необходимые ему зависимости в готовом виде, а не конструирует их сам. Внедрение зависимостей очень положительно отражается на всем коде: он становится слабо связанным, более удобным для чтения и тестирования.

Начнем эту главу с введения в теорию внедрения зависимостей и расскажем, какую пользу эта технология принесет вашей базе кода. Даже если вы уже работаете с фреймворком, использующим инверсию управления/внедрение зависимостей, в этой главе найдется материал, который поможет вам глубже понять сущность внедрения зависимостей. Это особенно важно, если вы (как многие из нас) начали работать с фреймворками внедрения зависимостей еще до того, как у вас появилась возможность подробно изучить причины, по которым используется такой подход.

Вы познакомитесь с документом JSR-330 — официальным стандартом внедрения зависимостей в Java. Так вам будет проще понять причины, по которым стандартный набор аннотаций для внедрения зависимостей в Java сложился именно так, а не иначе. Далее мы познакомим вас с фреймворком Guice 3, который является эталонной реализацией для JSR-330. Многие считают его очень красивым и легким методом внедрения зависимостей в Java.

Итак, начнем с теории и обсудим, как сложилась эта популярная парадигма. А также уточним, почему вам может быть интересно освоить ее.

3.1. Дополнительные знания: понятие об инверсии управления и внедрении зависимостей

Итак, зачем же нужно знать об инверсии управления, внедрении зависимостей и их базовых принципах? На этот вопрос можно ответить по-разному. Действительно, если бы вы обратились с этим вопросом на популярный сайт stackexchange.com Q&A, то получили бы множество разнообразных ответов!

Можно просто приступить к использованию различных фреймворков для внедрения зависимостей и научиться работать с ними на примерах из Интернета. Но, как и в области фреймворков для объектно-реляционного отображения (таких как Hibernate), вы сможете стать гораздо более сильным разработчиком, если будете понимать внутреннюю структуру происходящего процесса.

Этот раздел мы начнем с изучения некоторых теоретических вопросов, лежащих в основе двух важнейших концепций (инверсия управления и внедрение заисимостей), и поговорим о пользе применения данной парадигмы. Чтобы дополнительно прояснить эти концепции, мы рассмотрим пример с HollywoodService. Начнем с версии, которая сама находит собственные зависимости, и преобразуем ее в версию, в которую необходимые зависимости внедряются.

Начнем с инверсии управления — феномена, который зачастую ошибочно смешивают с внедрением зависимостей.

3.1.1. Инверсия управления

Если вы применяете парадигму программирования, не связанную с инверсией управления, то ход программной логики обычно контролируется центральным функционалом. Если программа спроектирована качественно, то центральный функционал вызывает методы применительно к переиспользуемым объектам, которые выполняют конкретные функции.

При использовании инверсии управления этот принцип «централизованного контроля» как будто выворачивается наизнанку. Код вызывающей стороны управляет порядком выполнения программы, но сама логика программы инкапсулируется в вызываемых подпроцедурах.

Инверсия управления, также иногда называемая «принципом Голливуда», сводится к тому, что в системе обязательно есть еще один фрагмент кода, содержащий первичный управляющий поток. Поэтому именно он будет вызывать ваш код, а не ваш код — его.

Взглянем на инверсию управления с другой стороны и разберем пользовательский интерфейс игры Zork (http://ru.wikipedia.org/wiki/Zork_I:_The_Great_Underground_Empire). Это древняя видеоигра. Посмотрим, как управляется текстовая версия этой игры, а как — графическая.

В текстовой версии игры мы увидим в интерфейсе просто пустое поле для ввода текста, получаемого от пользователя. Пользователь может вводить описания

действий, например `go east` (иду на восток) или `run from Grue` (бегу от монстра). После этого основная логика приложения активизирует соответствующий обработчик события, который перерабатывает действие и возвращает результат. Здесь важно отметить, что именно логика приложения управляет тем, какой обработчик события будет активизирован.

Если обратиться к другой версии этой игры, с графическим пользовательским интерфейсом, то здесь уже задействуется инверсия управления. Фреймворк графического пользовательского интерфейса определяет, какой обработчик события будет выполняться; логика приложения больше этим не занимается. Если пользователь щелкает кнопкой мыши на варианте действия, например, `иду на восток`, то происходит непосредственная активизация обработчика события, а логика приложения сосредотачивается на обработке действия.

Основное управление программой *инвертируется*, контроль передается от логики приложения к самому фреймворку графического пользовательского интерфейса.

Существует несколько вариантов реализации инверсии управления. В частности, это паттерны «Фабрика» (Factory), «Локатор Сервисов» (Service Locator) и, конечно же, «Внедрение зависимостей» (Dependency Injection, другое название — «Инъекция зависимостей»). Этот термин был популяризован Мартином Фаулером (Martin Fowler) в его статье «Инверсия управления и паттерн внедрения зависимостей»[1].

3.1.2. Внедрение зависимостей

Внедрение зависимостей — это частный случай инверсии управления. В таком случае процесс нахождения ваших зависимостей не находится под непосредственным управлением кода, выполняемого в настоящий момент. Вы можете написать собственный механизм внедрения зависимостей, но большинство разработчиков предпочитают пользоваться сторонним фреймворком со встроенным контейнером для инверсии управления, например Guice.

ПРИМЕЧАНИЕ

Контейнер инверсии управления можно считать средой времени выполнения. Среди контейнеров для внедрения зависимостей в Java следует назвать Guice, Spring и PicoContainer.

Контейнеры для инверсии управления предоставляют полезные возможности. Например, они позволяют гарантировать, что зависимость, предназначенная для многократного использования, будет сконфигурирована как Singleton («Одиночка»). Некоторые такие возможности будут исследованы в разделе 3.3, где мы рассмотрим Guice.

СОВЕТ

Существует несколько способов внедрения зависимостей в объекты. Для этого можно использовать специализированные фреймворки, но это совершенно не обязательно! Явное инстанцирование и передача объектов (то есть зависимостей) к вашему объекту могут быть не менее удобными, чем внедрение посредством фреймворка[2].

---

[1] Русский перевод статьи доступен по адресу http://yugeon-dev.blogspot.com/2010/07/inversion-of-control-containers-and_21.html

[2] Спасибо за этот совет Тьяго Аррайсу (Thiago Arrais) (http://stackoverflow.com/users/17801/thiago-arrais)!

Как и при работе со многими другими парадигмами программирования, важно понимать, зачем используется внедрение зависимостей. Самые значительные, на наш взгляд, преимущества внедрения зависимостей мы обобщили в табл. 3.1.

Таблица 3.1. Преимущества внедрения зависимостей

Преимущество	Описание	Пример
Слабое связывание	Ваш код перестает быть жестко связан с созданием нужных ему зависимостей. Применяя технику «программирования, основанного на интерфейсах», вы также можете добиться того, что код перестанет быть сильно связан с конкретными вариантами реализации зависимости	Если вашему объекту HollywoodService требуется создать собственный SpreadsheetAgentFinder, то ему можно просто передать такой SpreadsheetAgentFinder. При программировании, основанном на интерфейсах, классу HollywoodService можно передать AgentFinder любого типа
Тестируемость	В дополнение к слабому связыванию необходимо оговорить еще один случай, часто встречающийся на практике. При необходимости тестирования вы можете внедрить в код тестовый двойник именно в качестве зависимости	Можно внедрить «фиктивную» талонную систему оплаты, которая всегда будет возвращать одно и то же значение стоимости. Этим она отличается от «реальной» платежной системы, которая является внешней относительно вашего кода и может быть недоступна
Более высокая слаженность кода	Ваш код сосредотачивается на решении основной задачи, так как ему не требуется заниматься загрузкой и конфигурированием зависимостей. Побочный положительный эффект заключается в повышении удобочитаемости кода	Ваш объект доступа к данным (DAO) занимается именно выполнением запросов, а не поиском подробной информации о драйверах JDBC
Многоразовые компоненты	В дополнение к слабому связыванию достигается еще одно преимущество: ваш код может задействоваться более широкой пользовательской аудиторией. Пользователи могут предоставлять собственные специфичные реализации	Предприимчивый разработчик может продать вам инструмент для поиска агентов в Linkedin
Облегченный код	В коде больше не требуется передавать зависимости между уровнями. Вместо этого зависимости можно внедрять именно в тех точках, где они нужны	Вместо передачи подробной информации о драйвере JDBC от служебного класса можно внедрить драйвер непосредственно в объект для доступа данных — туда, где ему и место

По нашему опыту, преобразование обычного кода в код, содержащий зависимости, — наилучший способ усвоить теорию. Итак, этим и займемся.

3.1.3. Переход к внедрению зависимостей

В этом разделе будет показано, как перевести код, не использующий инверсию управления, к применению реализации в стиле «Фабрика» или «Локатор сервисов», то есть внедрить в код зависимости. В основе большинства этих операций лежит техника «программирования на основе интерфейсов». Эта практика обеспечивает подстановку объектов во время исполнения.

ПРИМЕЧАНИЕ

В этом разделе мы планируем закрепить ваше базовое представление о внедрении зависимостей, поэтому специально опустили некоторый шаблонный код.

Допустим, вам передали работу над небольшим проектом, который будет возвращать всех трудовых агентов из Голливуда, занимающихся подбором Java-разработчиков (назовем их «профессиональными»). В листинге 3.1 есть интерфейс `AgentFinder` с двумя реализациями — `SpreadsheetAgentFinder` и `WebServiceAgentFinder`.

Листинг 3.1. Интерфейс AgentFinder и реализующие его классы

```
public interface AgentFinder
{
  public List<Agent> findAllAgents();
}

public class SpreadsheetAgentFinder implements AgentFinder
{
  @Override
  public List<Agent> findAllAgents(){ ... }
}

public class WebServiceAgentFinder implements AgentFinder
{
@Override
  public List<Agent> findAllAgents(){ ... }
}
```

Несколько реализаций

Чтобы использовать инструменты для поиска агентов, в проекте предусмотрен стандартный класс `HollywoodService`, который получает список агентов от `SpreadsheetAgentFinder`, фильтрует их по показателю профессиональной заинтересованности и возвращает такой список интересующих вас агентов, как показано в листинге 3.2.

Листинг 3.2. HollywoodService с жестко закодированным AgentFinder

```
public class HollywoodService
{

  public static List<Agent> getFriendlyAgents()
  {
    AgentFinder finder = new SpreadsheetAgentFinder();
```

1 Используем SpreadsheetAgentFinder

```
    List<Agent> agents = finder.findAllAgents();
    List<Agent> friendlyAgents =
      filterAgents(agents, "Java Developers");
    return friendlyAgents;
  }

  public static List<Agent> filterAgents(List<Agent> agents,
    String agentType)
  {
    List<Agent> filteredAgents = new ArrayList<>();
    for (Agent agent:agents) {
      if (agent.getType().equals("Java Developers")) {
        filteredAgents.add(agent);
      }
    }
    return filteredAgents;
    }
}
```

Вызываем метод интерфейса

Возвращаем профессиональных агентов

Снова вернемся к `HollywoodService` в этом листинге и разберемся, почему код завязан на использовании всего лишь одной реализации `AgentFinder` — `SpreadsheetAgentFinder` (1).

Подобное ограничение реализации представляло проблему для многих Java-разработчиков. Как и в случаях с другими распространенныими проблемами, развивались паттерны, специально ориентированные на ее решение. Сначала многие разработчики использовали варианты паттернов «Фабрика» и «Локатор сервисов». Все эти варианты представляли собой разные типы инверсии управления.

Класс Hollywood с паттернами «Фабрика» и/или «Локатор сервисов»

Один из следующих паттернов — «Абстрактная фабрика», «Фабричный метод» или «Локатор сервисов» либо их комбинация обычно использовались для решения проблемы «блокирования» при применении зависимости.

ПРИМЕЧАНИЕ

Паттерны «Фабричный метод» и «Абстрактная фабрика» рассмотрены в книге Эриха Гаммы (Erich Gamma), Ричарда Хелма (Richard Helm), Ральфа Джонсона (Ralph Johnson) и Джона Влиссидеса (John Vlissides) «Приемы объектно-ориентированного проектирования. Паттерны проектирования» (СПб.: Питер, 2001). Паттерн «Локатор сервисов» рассмотрен в книге Дипака Алура (Deepak Alur), Джона Крупи (John Crupi) и Дэна Малкса (Dan Malks) «Образцы J2EE. Лучшие решения и стратегии проектирования».

В листинге 3.3 продемонстрирована версия класса `HollywoodService`, применяющая `AgentFinderFactory` для динамического выбора `AgentFinder`, который следует использовать.

Листинг 3.3. HollywoodService с фабричной подстановкой AgentFinder

```
public class HollywoodServiceWithFactory {

  public List<Agent>
        getFriendlyAgents(String agentFinderType)
  {
    AgentFinderFactory factory =
          AgentFinderFactory.getInstance();
    AgentFinder finder =
          factory.getAgentFinder(agentFinderType);
    List<Agent> agents = finder.findAllAgents();
    List<Agent> friendlyAgents =
      filterAgents(agents, "Java Developers");
    return friendlyAgents;
  }

  public static List<Agent> filterAgents(List<Agent> agents,
    String agentType)
  {
    ...
  }
}
```

1 Внедрение agentFinderType

2 Получение agentFinderType с помощью фабрики

Такая же реализация, как в листинге 3.2

Как видите, удалось избежать блокировки, допускающей использование лишь одной конкретной реализации AgentFinder. Вы внедряете agentFinderType 1, затем приказываете AgentFinderFactory получить AgentFinder на основании этого типа 2.

Итак, мы уже почти справились с внедрением зависимости, но пока остается две проблемы.

- Код внедряет справочную ссылку (agentFinderType), а не реальную реализацию объекта AgentFinder.
- Метод getFriendlyAgents содержит код для нахождения нужной зависимости, что в идеале не должно быть его основной задачей.

По мере того как разработчики начинают писать все более чистый код, техника внедрения зависимостей распространяется все шире. Зачастую она даже заменяет реализации абстрактной фабрики и локатора сервисов.

HollywoodService с внедрением зависимостей

Вероятно, вы уже догадываетесь, каков будет наш следующий шаг рефакторинга. Далее мы представим метод getFriendlyAgents уже с тем AgentFinder, который ему нужен. Эта операция показана в листинге 3.4.

Листинг 3.4. HollywoodService с внедренной вручную зависимостью для AgentFinder

```
public class HollywoodServiceWithDI
{

  public static List<Agent>
      emailFriendlyAgents(AgentFinder finder)
```

1 Внедряем AgentFinder

```
  {
    List<Agent> agents = finder.findAllAgents();
    List<Agent> friendlyAgents =
      filterAgents(agents, "Java Developers");
    return friendlyAgents;
  }
  public static List<Agent> filterAgents(List<Agent> agents,
      String agentType)
  {
    ...
  }
}
```

Выполняем поисковую логику (2)

См. листинг 3.2

Теперь у вас фактически есть написанное вручную решение, включающее внедрение зависимости. `AgentFinder` был внедрен в метод `getFriendlyAgents` (1). Вы уже видите, насколько чист стал метод `getFriendlyAgents`, он сосредоточен именно на бизнес-логике (2).

Но если разработчик вручную внедряет собственные зависимости примерно таким образом, у него остается еще одна серьезная головная боль. По-прежнему не решена проблема с выбором того, какую именно реализацию `AgentFinder` вы хотите использовать. Та работа, которую делала фабрика `AgentFinderFactory`, должна и сейчас быть *где-то* выполнена.

Именно здесь может очень пригодиться фреймворк внедрения зависимостей с контейнером для инверсии управления. Для того чтобы вам было легче понять, можно привести простую аналогию: фреймворк внедрения зависимостей — это обертка вокруг вашего кода, действующая во время исполнения. Фреймворк внедряет нужные вам зависимости именно в тот момент, когда они вам требуются.

Преимущество фреймворков для внедрения зависимостей заключается в том, что они могут выполнять такие операции практически в любой точке кода, где может понадобиться зависимость. Фреймворк может выполнять такую работу, поскольку у него есть контейнер для инверсии управления. Контейнер содержит уже готовые зависимости, готовые для применения в вашем коде во время исполнения.

Рассмотрим, как мог бы выглядеть класс `HollywoodService` при применении стандартной аннотации JSR-330, которую могут использовать любые совместимые фреймворки.

HollywoodService с внедрением зависимостей и JSR-330

Перейдем к последнему примеру кода в этом разделе. Здесь мы собираемся перепоручить внедрение зависимостей специальному фреймворку. В данном случае фреймворк для внедрения зависимостей инжектирует зависимость прямо в метод `getFriendlyAgents`, используя стандартную аннотацию JSR-330. Это показано в листинге 3.5.

Листинг 3.5. HollywoodService, внедрение с JSR-330 для AgentFinder

```
public class HollywoodServiceJSR330
{
  @Inject public void emailFriendlyAgents(AgentFinder finder)
  {
    List<Agent> agents = this.finder.findAllAgents();
    List<Agent> friendlyAgents =
      filterAgents(agents, "Java Developers");
    return friendlyAgents;
  }

  public static List<Agent> filterAgents(List<Agent> agents,
    String agentType)
  {
    ...
  }
}
```

(1) AgentFinder, внедренный с применением JSR-330

Выполнение поисковой логики

См. листинг 3.2

Конкретная реализация AgentFinder (например, WebServiceAgentFinder) теперь инжектируется во время исполнения фреймворком для внедрения зависимостей, который поддерживает аннотацию JSR-330 @Inject (1).

СОВЕТ

Хотя аннотации JSR-330 и позволяют внедрять зависимости для метода, это обычно делается лишь для методов-конструкторов или методов-установщиков. Данное соглашение более подробно обсуждается в следующем разделе.

Вновь рассмотрим некоторые преимущества внедрения зависимостей на примере класса HollywoodServiceJSR330 в листинге 3.5.

- **Слабое связывание.** HollywoodService больше не требует конкретного типа AgentFinder при выполнении работы.
- **Тестируемость.** Для того чтобы протестировать класс HollywoodService, можно внедрить базовый класс Java (например, POJOAgentFinder), возвращающий фиксированное количество агентов. В терминологии разработки через тестирование такой элемент называется «класс-заглушка» (stub class). Он отлично подходит для модульного тестирования, так как позволяет обойтись без веб-службы, таблицы либо других сторонних реализаций.
- **Более высокая слаженность кода.** Код больше не работает с фабриками и сопутствующими им поисковыми операциями, а выполняет лишь бизнес-логику.
- **Многоразовые компоненты.** Только представьте, насколько просто теперь будет другому разработчику, использующему ваш API, внедрять любую специфичную реализацию AgentFinder, какая ему только потребуется, — например, JDBCAgentFinder.
- **Облегченный код.** Код класса HollywoodServiceJSR330 существенно сократился по сравнению с исходной версией HollywoodService.

Использование внедрения зависимостей все отчетливее становится стандартной практикой для основательного Java-разработчика. В нескольких популярных контейнерах предоставляются отличные возможности для внедрения зависимостей. Но еще совсем недавно различные фреймворки для внедрения зависимостей определяли довольно несхожие стандарты, которым должен следовать код, чтобы получить все преимущества инверсии управления. Даже если в различных фреймворках и наблюдалось некоторое сходство в стилях конфигурации (например, это касалось использования XML или аннотаций Java), все равно было сложно подобрать общие аннотации или конфигурацию.

Новый стандартизированный подход к внедрению зависимостей в Java (в соответствии с запросом на спецификацию JSR-330) позволяет решить подобные проблемы. Кроме того, этот подход красиво обобщает основные возможности, реализуемые в основанных на Java фреймворках для внедрения зависимостей. Далее мы рассмотрим этот подход несколько подробнее, так как он позволяет составить красивое и надежное представление о «внутрисистемной» основе работы такого фреймворка для внедрения зависимостей, как Guice.

3.2. Стандартизированное внедрение зависимостей в Java

С 2004 года мы могли познакомиться с несколькими широко распространившимися контейнерами с инверсией управления, применявшимися для внедрения зависимостей (стоит назвать хотя бы Guice, Spring и PicoContainer). До недавнего времени все реализации использовали различные подходы к конфигурированию внедрения зависимостей для кода. Поэтому разработчикам было сложно переходить от одного фреймворка к другому.

Более или менее приемлемое решение было найдено примерно в мае 2009 года, когда два ведущих члена сообщества, занимающегося внедрением зависимостей, Боб Ли (Bob Lee) (из Guice) и Род Джонсон (Rod Johnson) (из SpringSource) объявили, что собираются вместе приступить к разработке стандартного набора аннотаций для подобных взаимодействий[1]. После этого был подан запрос на спецификацию JSR-330 (`javax.inject`) для регламентации стандартизированного внедрения зависимостей в версии Java SE. Данная инициатива встретила практически стопроцентную поддержку со стороны всех основных игроков, работающих в этой области.

А КАК НАСЧЕТ JAVA ДЛЯ ПРЕДПРИЯТИЙ?

В корпоративной версии Java (Java EE) собственный механизм для внедрения зависимостей появился уже в версии JEE 6 (он назывался CDI). Этот механизм был описан в запросе на спецификацию JSR-299 («Контексты и внедрение зависимостей на платформе Java EE»). Подробнее этот документ можно изучить на сайте http://jcp.org/. Вкратце: JSR-299 строится на основе JSR-330 и призван обеспечить стандартизированную конфигурацию для корпоративных сценариев.

[1] Боб Ли, анонс @javax.inject.Inject от 8 мая 2009 года. www.theserverside.com/news/thread.tss?thread_id=54499

После того как Java обогатилась `javax.inject` (поддерживаются версии Java SE 5, 6 и 7), появилась возможность использовать стандартизированное внедрение зависимостей и при необходимости переходить от одного фреймворка к другому. Например, вы можете запускать свой код во фреймворке Guice, если требуется легковесное решение для внедрения зависимостей. После этого вы можете перейти к фреймворку Spring, где предлагается более широкий набор возможностей.

ВНИМАНИЕ

На практике все сложнее, чем кажется. Если ваш код использует функцию, которая поддерживается только в конкретном фреймворке внедрения зависимостей, то при работе вы будете ограничены лишь этим фреймворком. В пакете javax.inject предоставляется подмножество общих функций для внедрения зависимостей, но вам могут потребоваться и более сложные функции. Можете себе представить, какие жаркие дебаты развернулись по поводу того, что должно войти в общий стандарт, а что — нет. Ситуация пока далека от совершенства, но теперь хотя бы вырисовывается способ, позволяющий избежать блокировки в рамках единственного фреймворка.

Чтобы понять, как новейшие фреймворки для внедрения зависимостей используют новый стандарт, стоит внимательно изучить пакет `javax.inject`. Необходимо помнить, что данный пакет просто предоставляет интерфейс и несколько типов аннотаций, реализуемых фреймворками внедрения зависимостей. Как правило, вам не придется реализовывать их самостоятельно, если вы не создаете собственный контейнер для инверсии управления в Java, совместимый с JSR-330. (А если вы его разрабатываете — снимаем перед вами шляпы!)

ЗАЧЕМ МНЕ ЗНАТЬ, КАК ВСЕ ЭТО РАБОТАЕТ?

Основательный Java-разработчик не ограничивается простым применением библиотек и фреймворков, не понимая при этом принципов их работы (хотя бы в общих чертах). При работе с внедрением зависимостей недостаточное понимание этих деталей может приводить к появлению неправильно сконфигурированных зависимостей, зависимостей, таинственно выпадающих из области видимости, разделяемых зависимостей, которые разделяться не должны, к неожиданным срывам пошаговой отладки и таинственному возникновению других гнусных проблем.

В документации Javadoc по `javax.inject` отлично рассказано о назначении этого пакета, так что процитируем дословно:

Пакет javax.inject[1].

«В этом пакете перечислены средства для получения объектов такими способами, которые позволяют максимизировать возможности переиспользования, тестируемость и удобство поддержки по сравнению с традиционными подходами, предполагающими применение конструкторов, фабрик и локаторов сервисов (например, JNDI). Этот процесс, называемый "внедрением зависимостей", целесообразен в большинстве нетривиальных приложений».

Пакет `javax.inject` состоит из пяти типов аннотаций (`@Inject`, `@Qualifier`, `@Named`, `@Scope` и `@Singleton`) и из одного интерфейса `Provider<T>`. Об этих компонентах мы поговорим в следующих нескольких подразделах. Начнем с аннотации `@Inject`.

[1] «Пакет javax.inject», доументация по Java, http://atinject.googlecode.com/svn/trunk/javadoc/javax/inject/packagesummary.html.

3.2.1. Аннотация @Inject

Аннотация `@Inject` используется с тремя типами членов классов. С ее помощью вы можете указывать, где хотели бы внедрить зависимость. Типы членов классов, которые могут внедряться для последующей обработки во время исполнения, таковы:

- конструкторы;
- методы;
- поля.

Можно аннотировать конструктор с помощью `@Inject` и быть уверенным, что его параметры будут предоставляться во время исполнения сконфигурированным вами контейнером инверсии управления. Например, объекты `Header` и `Content` внедряются в `MurmurMessage` при вызове конструктора.

```
@Inject public MurmurMessage(Header header, Content content)
{
  this.header = header;
  this.content = content;
}
```

Такая спецификация допускает внедрение нуля и более параметров для конструкторов. Иначе говоря, внедрение конструктора без параметров — вполне допустимая операция.

ВНИМАНИЕ

В соответствии со спецификацией в классе может быть только один конструктор с аннотацией @Inject. И это правильно, так как в противном случае среда времени исполнения Java (JRE) не могла бы определить, какой из внедренных конструкторов имеет приоритет.

Можно аннотировать метод с помощью `@Inject`, и, как и в случае с конструктором, этот метод может получить во время исполнения нуль или более параметров путем внедрения. Существуют некоторые ограничения, сводящиеся к тому, что внедренные методы не могут объявляться как `abstract`, а также не могут сами объявлять параметры типа[1]. В следующем кратком примере кода показано, как аннотация `@Inject` используется с методом-установщиком. Это обычная практика при внедрении зависимостей для задания опциональных полей.

```
@Inject public void setContent(Content content)
{
  this.content = content;
}
```

Этот способ внедрения параметров методов становится особенно мощным, когда приходится предоставлять служебным методам ресурсы, необходимые для выполнения их задач. Например, можно передать аргумент, представляющий собой объект доступа к данным (DAO), методу находящей службы, которая должна найти определенные данные.

[1] Здесь мы имеем в виду, что вы не сможете использовать уловку «Обобщенные методы», которая описана в Руководстве по Java на сайте Oracle по адресу http://download.oracle.com/javase/tutorial/extra/generics/methods.html

СОВЕТ

Уже можно считать стандартными оптимальными подходами использование внедрения конструктора при установке обязательных зависимостей для класса. Если речь идет о задании необязательных зависимостей, например полей с предустановленными разумными умолчаниями, то в таком случае применяется внедрение метода-установщика.

Можно внедрять и поля (при условии, что они не являются `final`), но эта практика не очень распространена, поскольку осложняет модульное тестирование. Синтаксис опять же довольно прост.

```
public class MurmurMessenger
{
  @Inject private MurmurMessage murmurMessage;
  ...
}
```

Вы можете подробнее почитать об аннотации `@Inject` в Javadoc[1]. В этом документе рассмотрены некоторые нюансы того, какие типы значений могут внедряться и как работать с циклическими зависимостями.

Теперь вы уже достаточно хорошо знакомы с аннотацией `@Inject`. Рассмотрим, как вы можете квалифицировать (более подробно идентифицировать) внедренные таким образом объекты для использования в вашем коде.

3.2.2. Аннотация @Qualifier

Аннотация `@Qualifier` определяет контракт для реализующих фреймворков. Он может использоваться для квалификации (идентификации) объектов, которые вы собираетесь внедрить в ваш код. Например, если в вашем контейнере для инверсии управления сконфигурировано два объекта одного и того же типа, то вам понадобится как-то различать их, если придется внедрять в код. Эта концепция более понятна из схемы, показанной на рис. 3.1.

Если вы используете реализацию, предоставляемую одним из фреймворков, то необходимо учитывать: существуют правила, регламентирующие создание реализации аннотации `@Qualifier`.

- Реализация должна быть аннотирована с помощью `@Qualifier` и `@Retention(RUNTIME)`. Так мы обеспечим сохранение квалификатора во время исполнения.
- Обычно реализация должна иметь и аннотацию `@Documented`, чтобы ее можно было добавить к API как часть общедоступного Javadoc.
- Реализация может иметь атрибуты.
- Реализация может быть доступна лишь для ограниченного использования, если она аннотирована `@Target`. Так, использование может быть ограничено полями, тогда как по умолчанию допустимо применение полей и параметров методов.

[1] Javadoc, «Аннотация типа Inject» http://atinject.googlecode.com/svn/trunk/javadoc/javax/inject/Inject.html

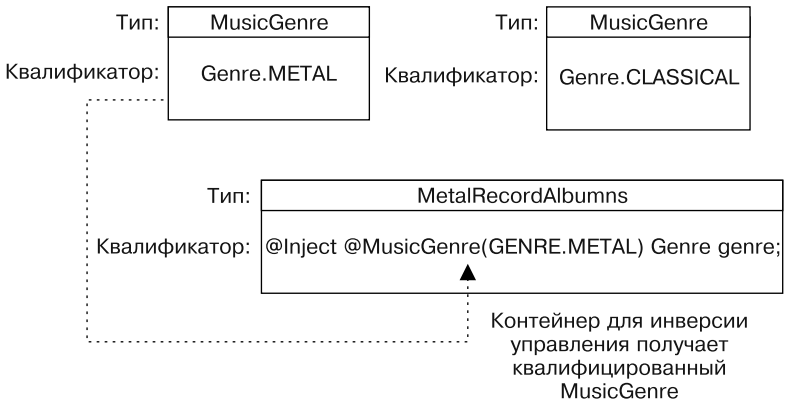


Рис. 3.1. Аннотация @Qualifier используется для различения двух объектов типа MusicGenre

Чтобы увидеть вышеизложенный список «в перспективе», представьте себе краткий гипотетический пример реализации `@Qualifier`, которая может предоставляться в контейнере для инверсии управления. Фреймворк музыкальной библиотеки может давать квалификатор `@MusicGenre`. В дальнейшем разработчик может использовать этот квалификатор при создании класса `MetalRecordAlbumns`. Квалификатор гарантирует, что внедряемый `Genre` будет иметь требуемый тип.

```
@Documented
@Retention(RUNTIME)
@Qualifier
public @interface MusicGenre
{
  Genre genre() default Genre.TRANCE;
  public enum GENRE { CLASSICAL, METAL, ROCK, TRANCE }
}

public class MetalRecordAlbumns
{
  @Inject @MusicGenre(GENRE.METAL) Genre genre;
}
```

Маловероятно, что вам придется создавать собственные аннотации `@Qualifier`, но важно иметь базовое представление о том, как работают различные реализации контейнеров для инверсии управления.

Один из типов `@Qualifier`, который в спецификации определяется как реализуемый в контейнерах инверсии управления всех типов, — это аннотационный интерфейс `@Named`.

3.2.3. Аннотация @Named

Аннотационный интерфейс `@Named` — это специфический `@Qualifier`, предоставляющий контракт для реализующих объектов. Этот контракт применяется для квалификации внедряемых объектов по имени. Если скомбинировать аннотацию `@Inject`

с квалифицирующей аннотацией @Named, то в код будет внедряться объект правильного типа, имеющий конкретное имя.

```
public class MurmurMessenger
{
  @Inject @Named("murmur") private MurmurMessage murmurMessage;
  @Inject @Named("broadcast") private MurmurMessage broadcastMessage;
  ...
}
```

Хотя существуют и другие достаточно распространенные квалификаторы, было принято решение, что только квалификатор @Named должен реализовываться всеми фреймворками внедрения зависимостей. Это решение озвучено в документе JSR-330.

Еще одна область, в которой различные составители исходной спецификации пришли к общему мнению — о необходимости наличия стандартизированного интерфейса, который управлял бы областями видимости для внедренных объектов.

3.2.4. Аннотация @Scope

Аннотация @Scope определяет контракт, который может применяться для определения того, как инъектор (то есть контейнер для внедрения зависимостей) будет переиспользовать экземпляры внедренного объекта. Спецификация определяет несколько стандартных поведений.

- Когда объявляется реализация аннотационного интерфейса @Scope, инъектор должен создать экземпляр объекта для внедрения, но применять этот экземпляр только в целях внедрения.
- Если объявляется реализация аннотационного интерфейса @Scope, то длительность жизненного цикла его внедренного объекта определяется реализацией данной области видимости.
- Если внедряемый объект может использоваться несколькими потоками в реализации @Scope, то он должен быть потокобезопасным. Подробнее о потоках и их безопасности рассказано в главе 4.
- Контейнер с инверсией управления должен генерировать исключение, если в одном и том же классе будет объявлено более одной аннотации @Scope либо если будет обнаружен неподдерживаемый вариант аннотации @Scope.

Такие стандартные поведения задают определенные границы для работы фреймворков внедрения зависимостей, управляя жизненными циклами внедряемых ими объектов. Некоторые контейнеры инверсии управления поддерживают и собственные реализации @Scope, особенно в области обслуживания пользовательских интерфейсов в Сети. (Такая ситуация сохранялась как минимум до тех пор, пока не был повсеместно принят документ JSR-299.) Только аннотация @Singleton была признана общей реализацией @Scope в соответствии с JSR-330. Поэтому @Singleton в спецификации также определяется как отдельный тип аннотации.

3.2.5. Аннотация @Singleton

Аннотационный интерфейс `@Singleton` — это аннотация, широко используемая во фреймворках для внедрения зависимостей. Довольно часто приходится внедрять значение объекта, которое не будет изменяться. В таком случае синглтон — достаточно эффективное решение.

ПАТТЕРН SINGLETON

Singleton («Одиночка») — это просто паттерн проектирования, при котором инстанцирование класса может происходить однажды, и только однажды. Подробнее об этом рассказано в книге Эриха Гаммы (Erich Gamma), Ричарда Хелма (Richard Helm), Ральфа Джонсона (Ralph Johnson) и Джона Влиссидеса (John Vlissides) «Приемы объектно-ориентированного проектирования. Паттерны проектирования» (СПб.: Питер, 2001). Будьте осторожны с паттерном «Синглтон» — иногда он оказывается антипаттерном!

В большинстве фреймворков для внедрения зависимостей `@Singleton` считается скрытым умолчанием. Например, если вы не объявляете область видимости, то по умолчанию фреймворк предполагает, что вы собираетесь использовать синглтон. Если вы явно объявляете синглтон, то это можно сделать так:

```
public MurmurMessage
{
  @Inject @Singleton MessageHeader defaultHeader;
}
```

В данном примере мы предполагаем, что `defaultHeader` никогда не изменится (фактически это статические данные), а значит, этот элемент может быть внедрен как синглтон.

Наконец, рассмотрим наиболее гибкий параметр. Он применяется в тех случаях, когда недостаточно ни одной из стандартных аннотаций.

3.2.6. Интерфейс Provider<T>

Чтобы вы могли приобрести дополнительный контроль над объектом, внедряемым в ваш код с помощью фреймворка, можете запросить фреймворк внедрить реализацию интерфейса `Provider<T>` для объекта (`T`), а не сам этот объект. Таким образом, у вас появятся следующие преимущества при управлении внедренным объектом:

- вы можете получать множественные экземпляры данного объекта;
- при необходимости можно отложить получение объекта (ленивая загрузка) либо вообще его не получать;
- можно разрывать циклические зависимости;
- можно определять область видимости, что позволит вам искать объекты в более узкой области, чем все загруженное приложение.

Интерфейс содержит всего один метод, `T get()`, который должен предоставлять полностью сконструированный внедренный экземпляр объекта (`T`). Например, можно внедрить реализацию интерфейса `Provider<T>` (`Provider<Message>`)

в конструктор MurmurMessage. Он будет получать различные объекты Message на основании произвольных критериев, как показано в листинге 3.6.

Листинг 3.6. Использование интерфейса Provider<T>

```
import com.google.inject.Inject;
import com.google.inject.Provider;

class MurmurMessage
{
  @Inject MurmurMessage (Provider<Message> messageProvider)
  {
    Message msg1 = messageProvider.get();
    if (someGlobalCondition)
    {
      Message copyOfMsg1 = messageProvider.get();
    }
    ...
  }
}
```

Получаем сообщение

Получаем копию Message ❶

Обратите внимание, как можно получать дополнительные экземпляры внедренного объекта Message от Provider<Message>, а не единственный экземпляр Message — при внедрении объекта вручную был бы доступен всего один его экземпляр. В данном случае вы используете вторую копию этого внедренного объекта Message, которая загружается лишь при необходимости ❶.

Итак, вот мы и изучили теорию и рассмотрели несколько небольших примеров работы с новым пакетом javax.inject. Теперь освоим на практике все приобретенные знания. Для этого исследуем Guice — полнофункциональный фреймворк для внедрения зависимостей.

3.3. Guice 3 — эталонная реализация внедрения зависимостей в Java

Guice (произносится как «Джюс») — проект Боба Ли, который тот ведет примерно с 2006 года. Проект расположен по адресу http://code.google.com/p/google-guice/. Здесь подробно описаны мотивы создания проекта, размещена документация. Здесь же вы можете скачать двоичные JAR-файлы для запуска примеров.

Именно в фреймворке для внедрения зависимостей, таком как Guice, вы занимаетесь конфигурированием ваших зависимостей, их связыванием, а также определяете, в какой области видимости они будут связываться, если ваш код использует аннотацию @Inject (а также другие аннотации из JSR-330).

Guice 3 — это полная эталонная реализация для JSR-330, и в данном разделе мы будем пользоваться этой версией. Guice — это нечто большее, чем обычный фреймворк для внедрения зависимостей, но в рамках текущего раздела мы сосредоточимся на возможностях Guice, связанных именно с внедрением зависимостей. Мы покажем примеры написания кода с внедрением зависимостей, в которых благодаря Guice могут применяться стандартные аннотации JSR-330.

3.3.1. Знакомство с Guice

Теперь вы понимаете, как работают различные аннотации из JSR-330. И вы можете использовать их в своем коде с помощью Guice! Guice позволяет собрать коллекцию объектов Java (вместе с их зависимостями), которые вы хотите внедрить. В терминологии Guice, чтобы приказать *инъектору* (injector) построить данный *граф объекта* (object graph), нужно создать *модули* (modules), объявляющие *коллекции связей* (bindings). Связи, в свою очередь, определяют конкретные реализации, которые вы хотите внедрить. Звучит сложно? Не волнуйтесь, все на самом деле станет понятно, когда мы изучим эти концепции в коде.

СОВЕТ

«Граф объекта», «связь», «модуль», «инъектор» — общепринятые термины в среде Guice. Поэтому следует усвоить эту терминологию, если вы собираетесь создавать приложения на базе Guice.

В этом разделе мы вновь рассмотрим пример с классом `HollywoodService`. Начнем с создания конфигурационного класса (модуля), в котором будут содержаться связи. Фактически так происходит внешняя конфигурация зависимостей, которыми фреймворк Guice будет управлять для вас.

ГДЕ СКАЧАТЬ GUICE

Скачайте новейшую версию Guice по адресу http://code.google.com/p/google-guice/downloads/list. Соответствующая документация находится по адресу http://code.google.com/p/google-guice/wiki/Motivation?tm=6.

Чтобы обеспечить полную поддержку контейнеров инверсии управления и внедрения зависимостей, нужно скачать архив Guice и распаковать его в удобную для вас папку. Чтобы использовать Guice в своем коде Java, необходимо обязательно включить в путь к классу JAR-файлы.

При рассмотрении последующих образцов кода в нашей книге JAR-файлы Guice 3 также автоматически входят в состав сборки Maven.

Начнем с создания класса связей `AgentFinderModule`. Этот класс `AgentFinderModule` должен дополнять `AbstractModule`, а сами связи будут объявляться в переопределенном методе `configure()`. В данном конкретном случае класс `WebServiceAgentFinder` привязывается как объект для внедрения, когда `HollywoodServiceGuice` запрашивает внедрение (`@Inject`) `AgentFinder`. Здесь мы будем следовать соглашению по работе с внедрением конструктора. Рассмотрим листинг 3.7.

Листинг 3.7. HollywoodService с внедрением зависимостей посредством Guice для AgentFinder

```
import com.google.inject.AbstractModule;

public class AgentFinderModule extends AbstractModule
{
  @Override
  protected void configure()
  {
```

Дополняет класс AbstractModule

Переопределяет метод configure()

```
    bind(AgentFinder.class).
           to(WebServiceAgentFinder.class);
  }
}

public class HollywoodServiceGuice
{
  private AgentFinder finder = null;
  @Inject
  public HollywoodServiceGuice(AgentFinder finder)
  {
    this.finder = finder;
  }

  public List<Agent> getFriendlyAgents()
  {
    List<Agent> agents = finder.findAllAgents();
    List<Agent> friendlyAgents = filterAgents(agents, "Java Developers");
    return friendlyAgents;
  }

  public List<Agent> filterAgents(List<Agent> agents, String agentType)
  {
    ...
  }
}
```

Привязывает реализацию для внедрения (1)

Такая же реализация, как в листинге 3.2

Основная часть операции связывания осуществляется при использовании метода связывания Guice. Ему вы передаете класс, который хотите привязать (`AgentFinder`), затем используете метод `to` для объявления того, какая реализация будет внедряться (1).

Теперь, когда ваша *связь* объявлена в *модуле*, вы можете получить *инъектор* для построения *графа объекта*. Мы рассмотрим, как это делается в автономном Java-приложении и в веб-приложении.

Построение графа объекта Guice — автономное Java-приложение

В стандартном Java-приложении вы можете построить граф объекта Guice с помощью метода `public static void main(String[] args)`. В листинге 3.8 показано, как это делается.

Листинг 3.8. `HollywoodServiceClient` — построение графа объекта с помощью Guice

```
import com.google.inject.Guice;
import com.google.inject.Injector;
import java.util.List;

public class HollywoodServiceClient
```

```
{
  public static void main(String[] args)
  {
    Injector injector =
      Guice.createInjector(new AgentFinderModule());
    HollywoodServiceGuice hollywoodService =
      injector.getInstance(HollywoodServiceGuice.class);
    List<Agent> agents = hollywoodService.getFriendlyAgents();
    ...
  }
}
```

С веб-приложениями ситуация обстоит немного иначе.

Построение графа объекта Guice — веб-приложение

При работе с веб-приложением необходимо добавить к нему файл `guice-servlet.jar`, а также вставить следующий фрагмент кода в файл `web.xml`.

```
<filter>
  <filter-name>guiceFilter</filter-name>
  <filter-class>com.google.inject.servlet.GuiceFilter</filter-class>
</filter>
<filter-mapping>
  <filter-name>guiceFilter</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
```

Далее считается стандартной практикой дополнять `ServletContextListener` для работы с модулем Guice `ServletModule` (синонимичен `AbstractModule` в листинге 3.7).

```
public class MyGuiceServletConfig extends GuiceServletContextListener {
  @Override
  protected Injector getInjector() {
    return Guice.createInjector(new ServletModule());
  }
}
```

Наконец, добавим следующий код в `web.xml`. В таком случае контейнер сервлетов запустит этот класс при развертывании приложения:

```
<listener>
  <listener-class>com.java7developer.MyGuiceServletConfig</listener-class>
</listener>
```

При создании `HollywoodServiceGuice` из инъектора вы получаете полностью сконфигурированный класс, применительно к которому можете немедленно вызвать метод `getFriendlyAgents`.

Довольно просто, правда? Да, но все несколько усложняется, поскольку вам могут понадобиться более сложные связи, чем обычная связь `WebServiceAgentFinder` с `AgentFinder`, показанная в листинге 3.7.

3.3.2. Морские узлы — различные связи в Guice

В Guice предлагается множество связей. В официальной документации перечислены следующие типы:

- ссылочные связи (linked bindings);
- связывающие аннотации (binding annotations);
- связи экземпляров (instance bindings);
- методы `@Provides`;
- связи поставщиков (provider bindings);
- нецелевые связи (untargeted bindings);
- встроенные связи (built-in bindings);
- динамические связи (just-in-time bindings).

Мы не будем здесь дословно цитировать документацию, а подробно обсудим лишь самые распространенные связи — ссылочные связи, связывающие аннотации, а также связи `@Provides` и `Provider<T>`.

Ссылочные связи

Ссылочная связь — это простейшая форма связи. Связи именно такого типа мы использовали при конфигурировании `AgentFinderModule` в листинге 3.6. Связь этого типа указывает инъектору, что он должен внедрять реализующий или дополняющий класс (да, можно внедрять и классы, являющиеся прямыми потомками данного класса) во время исполнения.

```
@Override
protected void configure()
{
  bind(AgentFinder.class).to(WebServiceAgentFinder.class);
}
```

Вы уже видели такую связь в действии. Рассмотрим следующий по распространенности тип связей — связывающие аннотации.

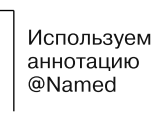
Связывающие аннотации

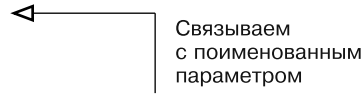
Связывающие аннотации используются для комбинирования типа класса, который вы хотите внедрить, с дополнительным идентификатором, который может точно указывать, какой именно объект следует внедрить. Вы можете писать собственные связывающие аннотации (см. онлайн-документацию по Guice), а мы остановимся на использовании стандартной связи JSR-330 `@Named`, встроенной в Guice.

В данном случае вы также сможете работать с уже знакомой аннотацией `@Inject`, но ее необходимо дополнить аннотацией `@Named`, чтобы извлечь `AgentFinder` с конкретным именем. Вы конфигурируете такой поименованный (`@Named`) тип связи в своем модуле `AgentModule`, используя метод `annotatedWith`, как показано в листинге 3.9.

Листинг 3.9. HollywoodService с использованием @Named

```
public class HollywoodService
{
  private AgentFinder finder = null;

  @Inject
  public HollywoodService(@Named("primary") AgentFinder finder)
  {
    this.finder = finder;
  }
}

public class AgentFinderModule extends AbstractModule
{
  @Override
  protected void configure()
  {
    bind(AgentFinder.class)
    .annotatedWith(Names.named("primary"))
    .to(WebServiceAgentFinder.class);
  }
}
```





Итак, вы научились конфигурировать поименованные зависимости. Теперь перейдем к следующему типу связей. Такие связи позволяют передавать полнофункциональные зависимости с применением аннотации `@Provides` и интерфейса `Provider<T>`.

@Provides и Provider<T> — предоставление полностью инстанцированных объектов

Можно применять аннотацию `@Provides` как вместо использования связи в методе `configure()`, так и вместе с ней, если вы хотите вернуть полностью инстанцированный объект. Например, вам может понадобиться внедрить довольно специфическую реализацию `SpreadsheetAgentFinder` в формате таблицы Microsoft Excel.

Инъектор будет просматривать возвращаемый тип всех методов, снабженных аннотацией `@Provides`, чтобы определить, какой объект внедрить. Например, `HollywoodService` будет использовать `AgentFinder`, предоставленный методом `provideAgentFinder` с аннотацией `@Provides`, как показано в листинге 3.10.

Листинг 3.10. AgentFinderModule, использующий @Provides

```
public class AgentFinderModule extends AbstractModule
{
  @Override
  protected void configure(){ }

  @Provides
  AgentFinder provideAgentFinder()
```

```
  {
    SpreadsheetAgentFinder finder =
                  new SpreadsheetAgentFinder();
    finder.setType("Excel 97");
    finder.setPath("C:/temp/agents.xls");
    return finder;
  }

}
```

Конкретный SpreadsheetAgentFinder

Количество методов `@Provides` может быстро вырасти, тогда их понадобится распределить на отдельные классы (а не переполнять ими классы модулей). Для этого Guice поддерживает интерфейс JSR-330 `Provider<T>`. Вспомните раздел о JSR-330, там упоминался метод `T get()`. Этот метод запускается, когда класс `AgentFinderModule` конфигурирует связь с `AgentFinderProvider` с помощью метода `toProvider`. Такая связь продемонстрирована в листинге 3.11.

Листинг 3.11. `AgentFinderModule`, использующий интерфейс `Provider<T>`

```
public class AgentFinderProvider implements Provider<AgentFinder>
{
  @Override
  public AgentFinder get()
  {
    SpreadsheetAgentFinder finder = new SpreadsheetAgentFinder();
    finder.setType("Excel 97");
    finder.setPath("C:/temp/agents.xls");
    return finder;
  }
}

public class AgentFinderModule extends AbstractModule
{
  @Override
  protected void configure()
  {
    bind(AgentFinder.class)
      .toProvider(AgentFinderProvider.class);
  }
}
```

Использование метода T get()

Привязывание поставщика

На этом завершается рассказ о последнем примере связывания, который мы хотели рассмотреть. Теперь вы научились использовать Guice для связывания зависимостей, готовых для применения в коде. Но еще остается обсудить, в какой области видимости существуют эти зависимости. Очень важно иметь понятие об области видимости, так как если объекты оказываются в неверной области видимости, то они существуют очень долго и потребляют больше памяти, чем нужно.

3.3.3. Задание области видимости для внедренных объектов в Guice

Guice предоставляет несколько уровней области видимости для объектов, которые вы собираетесь внедрять. Самая узкая область видимости — `@RequestScope`, область `@SessionScope` шире, а еще есть область видимости `@Singleton`, знакомая вам из документа JSR-330. Фактически в последнем случае мы имеем область видимости на уровне приложения.

Область видимости зависимостей может применяться в коде несколькими способами, например:

- к классу, который вы хотите внедрить;
- как часть последовательности связывания (например, `bind().to().in()`);
- как дополнительная аннотация к контракту `@Provides`.

Список получился немного абстрактным. Поэтому изучим суть перечисленных случаев на небольших примерах кода. Начнем с области видимости, ограниченной классом, который вы хотите внедрить.

Задание области видимости для класса, который вы хотите внедрить

Предположим, вам всего лишь однажды понадобился единственный экземпляр `SpreadsheetAgentFinder`, который вы собираетесь использовать во всем приложении. Для этого можно применить к объявлению класса область видимости `@Singleton` вот так:

```
@Singleton
public class SpreadsheetAgentFinder
{
  ...
}
```

Дополнительная польза от применения этого метода заключается в том, что разработчик видит на примере, насколько потокобезопасным должен быть класс. Поскольку `SpreadsheetAgentFinder` теоретически может внедряться многократно, область видимости `@Singleton` указывает, что необходимо гарантировать потокобезопасность класса (подробнее о безопасности потоков мы поговорим в главе 4).

Если вы предпочитаете объявлять всю вашу информацию об областях видимости при привязывании зависимости, то можете так и делать.

Использование последовательности BIND() для задания области видимости

Возможно, некоторым разработчикам будет удобнее держать в одном месте все правила, относящиеся к внедренному объекту. Помните, как мы использовали связывание в листинге 3.9, когда привязывали основной `AgentFinder`? При добавлении

области видимости к этой связи используется схожий подход: вы просто добавляете `.in(<Scope>.class)` в последовательность связывания как дополнительный вызов метода.

В следующем фрагменте кода мы усовершенствуем листинг 3.9, добавив в последовательность `in(Session.class)`. Так внедренный основной объект `AgentFinder` станет доступен в области видимости, действующей в пределах сеанса (session scope).

```
public class AgentFinderModule extends AbstractModule
{
  @Override
  protected void configure()
  {
    bind(AgentFinder.class)
     .annotatedWith(Names.named("primary"))
     .to(WebServiceAgentFinder.class)
     .in(Session.class);
  }
}
```

Задание области видимости поставщика объектов @Provides

Можно указать область видимости вместе с аннотацией `@Provides`. Так определяется область видимости объектов, предоставляемых этим методом. Например, возвращаясь к листингу 3.9, мы можем добавить дополнительную аннотацию `@Request`, чтобы привязывать результирующие экземпляры `SpreadsheetAgentFinder` к области видимости запроса.

```
@Provides @Request
AgentFinder provideAgentFinder()
{
  SpreadsheetAgentFinder finder = new SpreadsheetAgentFinder();
  finder.setType("Excel 97");
  finder.setPath("C:/temp/agents.xls");
  return finder;
}
```

Guice также предоставляет конкретные области видимости на основе веб-приложений (например, область видимости запросов сервлета). Кроме того, по мере необходимости вы можете писать собственные области видимости.

Итак, теперь вы имеете хорошее базовое представление о том, как Guice работает с аннотациями JSR-330 при внедрении зависимостей в вашей базе кода. В Guice есть и другие функции, не относящиеся к JSR-330, например поддержка аспектно-ориентированного программирования (AOP). Такая функциональность может быть полезна при реализации сквозных возможностей приложения, связанных с обеспечением безопасности и логированием. Эти вопросы рассмотрены в онлайн-документации по Guice и в примерах кода.

БУДЬТЕ ВНИМАТЕЛЬНЫ ПРИ ВЫБОРЕ ОБЛАСТИ ВИДИМОСТИ!

К ключевым решениям, которые приходится принимать основательному Java-разработчику, несомненно, относится выбор области видимости для используемых в работе объектов. Объекты, не сохраняющие состояния и сравнительно простые в создании, могут обходиться и без установленной области видимости. Виртуальная машина Java сможет без труда создавать и уничтожать их по мере необходимости (о виртуальной машине Java и ее производительности мы подробно поговорим в главе 6).

Напротив, для объектов, сохраняющих состояние, область видимости нужно задавать всегда! Необходимо заранее определять, будет ли жизненный цикл данного объекта равен всему циклу приложения, данной пользовательской сессии или данному запросу. В дальнейшем вы обязательно должны заботиться о потокобезопасности данного объекта (подробнее мы поговорим об этом в главе 4).

3.4. Резюме

Возможно, инверсия управления — довольно сложная концепция, в которой тяжело разобраться. Но внимательно изучив концепции паттернов Factory («Фабрика») и Service Locator («Локатор сервисов»), вы сможете понять, как работает базовая реализация инверсии управления. Паттерн Factory можно считать промежуточным этапом на пути к пониманию внедрения зависимостей и той пользы, которую оно может дать вашей базе кода. Даже если вам сложно иметь дело с парадигмой внедрения зависимостей, стоит ее придерживаться, так как она позволяет писать слабо связанный код, удобный в тестировании и легкий для чтения.

JSR-330 — не просто важный стандарт, унифицирующий общую функциональность внедрения зависимостей. Этот документ описывает неявные правила и ограничения, о которых вы должны знать. Изучая стандартный набор аннотаций для внедрения зависимостей, вы можете гораздо полнее понять, как различные фреймворки внедрения зависимостей реализуют спецификацию. В результате вы сможете их максимально эффективно использовать.

Guice — это эталонная реализация JSR-330, а также популярный и легкий способ приступить к внедрению зависимостей в вашем коде. Действительно, во многих прикладных случаях бывает достаточно использовать Guice и совместимый с JSR-330 набор аннотаций, чтобы решать практически любые задачи, связанные с внедрением зависимостей.

Если вы читаете книгу с самого начала, то, пожалуй, заслужили небольшой перерыв. Оторвитесь от чтения, позанимайтесь чем-нибудь приятным, а потом с новыми силами приступайте к изучению темы, которую должен в совершенстве знать любой основательный Java-разработчик. Речь пойдет о параллельной обработке.

4 Современная параллельная обработка

В этой главе:

- теория параллельной обработки;
- блочный параллелизм;
- библиотеки `java.util.concurrent`;
- легкий параллелизм с применением фреймворка fork/join (ветвление/слияние);
- модель памяти в Java (JMM).

Эта глава начинается с изучения базовых концепций и с шапочного знакомства с блочным параллелизмом. До появления Java 5 блочный параллелизм был единственно возможным вариантом, но и до сих пор не утратил актуальности. Далее мы обсудим тему, в которой должен разбираться любой основательный Java-разработчик — речь пойдет о библиотеке `java.util.concurrent` и о базовых элементах для параллельной обработки, которые в ней предоставляются.

В конце этой главы мы рассмотрим новый фреймворк fork/join, название которого можно перевести как «ветвление/слияние». Итак, закончив чтение этой главы, вы будете готовы к применению новых методов параллельной обработки в своем собственном коде. Кроме того, вы будете знать достаточно много теории, чтобы полностью понимать различные особенности параллелизма, о которых мы поговорим далее по ходу книги, при обсуждении не Java-языков.

Эта глава не претендует на звание компендиума по всем вопросам, которые вы должны изучить по теме параллелизма, но дает достаточно сведений, чтобы сориентировать вас в том, что потребуется изучить в дальнейшем, и позволить приступить к работе. Кроме того, после прочтения этой главы вам будет проще и комфортнее писать параллельный код.

НО Я УЖЕ ЗНАЮ, ЧТО ТАКОЕ ПОТОКИ!

Это одна из наиболее распространенных (и наиболее опасных) ошибок, которые может совершить разработчик. Почему-то многие полагают, что поверхностное знание Thread, Runnable и языковых примитивов, обеспечивающих реализацию параллелизма в Java — все, что нужно, чтобы считать себя компетентным разработчиком параллельного кода. На самом деле тема параллелизма очень обширна, а качественная многопоточная разработка

сложна. Многопоточность по-прежнему продолжает доставлять проблемы даже самым лучшим разработчикам с многолетним опытом.

С другой стороны, необходимо учитывать, что в настоящее время в области многопоточности разворачивается множество актуальных исследований. Определенно они окажут большое влияние на Java и другие языки, с которыми вы будете сталкиваться в ходе профессиональной карьеры. Если бы нам предложили назвать фундаментальную область вычислительной техники, которая должна радикально измениться в практическом отношении в ближайшие несколько лет, то мы назвали бы именно параллельную обработку.

Но вы должны учитывать, что изложенной здесь информации недостаточно, чтобы стать по-настоящему высококлассным разработчиком многопоточного кода. Есть несколько великолепных книг, посвященных исключительно параллельной обработке в Java. Две наилучших — «Параллельное программирование на языке Java» Дага Ли (*Lea Doug*. Concurrent Programming in Java. Second Edition. Prentice Hall, 1999) и «Параллелизм в Java на практике» Брайана Гётца и др. (*Goetz Brian and others*. Java Concurrency in Practice. Addison-Wesley Professional, 2006).

Цель этой главы — познакомить вас с базовыми механизмами работы платформы, помогающими понять, почему параллельная обработка в Java организована именно так, а не иначе. Кроме того, мы достаточно глубоко изучим теорию параллельной обработки, чтобы вы могли усвоить терминологию и понять проблемы, связанные с этим видом деятельности. Вы сможете понять как необходимость, так и сложность, неотделимые от правильной организации параллелизма. На самом деле именно с теории мы и хотели бы начать.

4.1. Теория параллелизма — базовый пример

Чтобы понять, как в Java организуется параллельное программирование, мы познакомимся с теорией. Для начала обсудим основы поточной модели в Java.

Затем мы поговорим о влиянии, которое оказывают «структурные силы» на проектирование и реализацию систем. Мы обсудим две наиболее важные из этих сил: стремление к безопасности и стремление к жизнеспособности. После этого обратимся к изучению сил, между которыми часто возникают конфликты, а также разберем причины, по которым в работе параллельных систем возникают издержки.

Завершим этот раздел рассмотрением примера многопоточной системы и покажем, как можно с удобством использовать `java.util.concurrent` при написании кода.

4.1.1. Рассмотрение модели потоков в Java

Модель потоков в Java основана на двух фундаментальных концепциях:

- разделяемое, видимое по умолчанию изменяемое состояние;
- диспетчеризация потоков по приоритетам.

Рассмотрим некоторые наиболее важные особенности этих идей.

- В ходе обработки объекты могут без проблем разделяться между потоками (совместно ими использоваться).
- Объекты могут быть изменены любыми потоками, имеющими ссылку на эти объекты.
- Диспетчер потоков может перераспределять потоки между ядрами процессора, нагружая эти ядра то больше, то меньше.
- При оперировании методами необходима возможность их выгрузки прямо в процессе работы (в противном случае метод может войти в бесконечный цикл и будет постоянно впустую расходовать ресурсы процессора).

  Правда, при этом возникает риск непредсказуемой переброски потока, из-за которой метод останется «выполненным наполовину», а объект окажется в несогласованном состоянии. Кроме того, существует риск, что изменения, сделанные в одном потоке, будут невидимы в других потоках — в то время как их видимость будет необходима.

  Для снижения этих рисков предусмотрена следующая возможность.
- Объекты могут *блокироваться* для защиты уязвимых данных

Параллелизм Java, основанный на потоках и блокировках, — это очень низкоуровневый механизм, работать с которым зачастую довольно сложно. Чтобы упростить этот процесс, в Java 5 появился набор библиотек для обеспечения параллелизма, называемый `java.util.concurrent`. В нем предоставляется несколько инструментов для написания параллельного кода. Многим программистам удобнее работать с таким инструментарием, чем с классическими блочно-структурированными примитивами для параллелизма.

УСВОЕННЫЕ УРОКИ

Java был первым широко распространенным языком, в котором появилась встроенная поддержка многопоточного программирования. На тот момент это был огромный шаг вперед, но сейчас, через 15 лет, мы узнали много нового о том, как писать параллельный код.

Оказывается, некоторые решения, принятые уже на первых этапах проектирования Java, на практике довольно сложны для большинства программистов. Это не радует, так как в производстве аппаратного обеспечения все сильнее нарастает тенденция к созданию многоядерных процессоров, а единственный довольно эффективный способ использования таких ядер — написание параллельного кода. Некоторые сложности, связанные с параллельным кодом, мы рассмотрим в этой главе. Тема современных процессоров, по природе своей требующих параллельного программирования, частично рассмотрена в главе 6, где мы обсуждаем производительность.

По мере того как разработчики поднаторели в написании параллельного кода, выяснилось, что принципы параллельного программирования начинают вступать в противоречие с некоторыми аспектами, очень важными для параллельных систем. Эти аспекты условно называются *структурными силами* (design forces). Они представляют собой высокоуровневые, априори существующие силы, которые зачастую вступают в противоречие с проектированием реальных параллельных объектно-ориентированных систем.

Потратим еще немного времени и рассмотрим некоторые наиболее важные из этих сил. Об этом пойдет речь в следующих нескольких разделах.

4.1.2. Структурные концепции

Самые важные структурные силы были перечислены Дагом Ли в ходе его эпохальной работы по созданию `java.util.concurrent`:

- безопасность (также называемая *безопасностью типов при параллельной обработке*);
- жизнеспособность;
- производительность;
- возможность многократного использования.

Подробнее рассмотрим каждую из этих сил.

Безопасность и безопасность типов при параллельной обработке

Безопасность — это обеспечение того, что экземпляры объекта остаются самодостаточными независимо от любых других операций, которые могут протекать в системе в настоящее время. Если система объектов обладает таким свойством, то говорят, что она обеспечивает безопасность типов при параллелизме.

Как понятно из названия, параллелизм можно представить как расширение для традиционных концепций моделирования объектов и безопасности типов. В непараллельном коде необходимо гарантировать, что, независимо от того, какие общедоступные методы вы вызываете применительно к объекту, в конце выполнения метода этот объект останется в правильно определенном и непротиворечивом состоянии. Обычно эта цель достигается при сохранении всей информации о состоянии объекта в закрытом виде и предоставлении общедоступного API из методов, которые изменяют состояние объекта только непротиворечивым образом.

Безопасность типов при параллелизме — это, в принципе, такая же концепция, как и безопасность типов объекта, но она применима в гораздо более сложной среде, где одними и теми же объектами потенциально могут оперировать сразу несколько потоков, работающих на разных ядрах процессора.

О ДОЛГОВРЕМЕННОЙ БЕЗОПАСНОСТИ

Одна из стратегий обеспечения безопасности сводится к тому, чтобы никогда не возвращаться из публичного метода в несогласованное состояние, а также никогда не вызывать публичный метод (и любой другой метод к любому объекту), будучи в несогласованном состоянии. Если скомбинировать такой подход с определенным способом защиты объекта (например, с применением блокировки синхронизации или критической секции), пока объект несогласован, то система гарантированно останется безопасной.

Жизнеспособность

«Живой» называется такая система, в которой любое предпринятое действие в итоге либо прогрессирует, либо не удается.

Ключевые слова в этом определении — *«в итоге»*. Есть разница между временной невозможностью прогрессирования (что само по себе не так плохо, хотя и не идеально) и устойчивым отказом. Временный отказ может быть обусловлен несколькими базовыми проблемами, в частности:

- блокировкой или ожиданием получения блокировки;
- ожиданием ввода (например, при сетевом вводе-выводе);
- временным отказом ресурса;
- недостаточным количеством процессорного времени для запуска потока.

Устойчивый отказ также может возникать по многим причинам. Вот некоторые распространенные причины:

- взаимная блокировка;
- необратимая проблема с ресурсами (например, потеря связи с NFS);
- потерянный сигнал.

Ниже в этой главе мы поговорим о блокировках, а также о некоторых других проблемах. Хотя, возможно, вы уже знакомы со многими из них.

Производительность

Количественную оценку производительности системы можно выполнить несколькими способами. В главе 6 мы поговорим об анализе производительности и о методах ее настройки. Кроме того, мы познакомим вас еще с несколькими важными параметрами. Пока будем считать, что *производительность* — это мера того, как много работы может выполнить система, располагая данным объемом ресурсов.

Возможность многократного использования

Возможность многократного использования (переиспользуемость) — это четвертая структурная сила, так как она не затрагивается ничем из вышеперечисленного. Система параллельной обработки, которая была бы спроектирована специально для несложного переиспользования, иногда очень желательна, хотя ее и не так просто реализовать. Один из способов — применение многоразового инструментария (например, `java.util.concurrent`), на базе которого строится код приложения, не рассчитанный на переиспользование.

4.1.3. Как и в каких случаях возникает конфликт

Часто структурные силы противодействуют друг другу, и подобное противоречие можно считать основной причиной, по которой так сложно проектировать качественные системы для параллельной обработки.

- Стандарты безопасности могут противоречить жизнеспособности. Основная цель безопасности — не допустить патологических явлений, а основной смысл жизнеспособности — обеспечить прогресс.
- Переиспользуемые системы часто предоставляют свои внутренние ресурсы, что может приводить к проблемам с безопасностью.

- Нерационально написанная система безопасности обычно не отличается высокой производительностью, так как активно задействует блокировки, чтобы обеспечить высокую безопасность.

Баланс, которого вы в конечном итоге должны попытаться достичь, требует, чтобы код был достаточно гибок и полезен для решения разнообразных проблем, достаточно закрыт, чтобы быть безопасным, и при этом оставался довольно живым и высокопроизводительным. Соответствовать сразу всем этим требованиям непросто, но, к счастью, есть кое-какие практичные методы, облегчающие нашу работу в этом направлении. Вот наиболее распространенные из методов, которые мы постарались расположить в порядке полезности.

- Максимально ограничивать внешнюю коммуникацию каждой из подсистем. Сокрытие данных — мощный инструмент для обеспечения безопасности.
- Делать внутреннюю структуру каждой из подсистем максимально детерминированной. Например, нужно статически проектировать в каждой подсистеме знания о потоках и объектах, даже если взаимодействия в этой подсистеме будут протекать в параллельном, недетерминированном режиме.
- Формулировать рекомендации по разработке, которым необходимо следовать при создании клиентских приложений. Это достаточно сильный метод, но он предполагает, что производители пользовательских приложений действительно готовы с вами сотрудничать. Могут возникнуть большие проблемы с отладкой, если придется отлаживать сломавшееся приложение, разработанное с грубыми нарушениями правил.
- Документировать требуемые поведения. Это наиболее слабая из альтернатив, но иногда прибегать к ней необходимо, если код приходится развертывать в очень общем контексте.

Разработчик должен знать о возможности применения всех этих механизмов обеспечения безопасности и использовать самую сильную из потенциально возможных техник. При этом необходимо учитывать, что в отдельных случаях могут быть приемлемы лишь самые слабые механизмы.

4.1.4. Источники издержек

В системе параллельной обработки есть множество особенностей, неизбежно вызывающих существенные издержки:

- блокировки и мониторы;
- количество контекстных переключений;
- количество потоков;
- диспетчеризация;
- локализация памяти;
- конструкция алгоритмов.

Запомните этот список как основной контрольный перечень. При разработке параллельного кода необходимо гарантировать, что все пункты этого перечня были учтены. Только потом можно считать, что код готов.

КОНСТРУКЦИЯ АЛГОРИТМОВ

Именно в этой области разработчики могут по-настоящему отличаться. Если вы научитесь дизайну алгоритмов, то станете высококлассным программистом независимо от того, с каким языком работаете. Две книги, относящиеся к числу лучших работ по этой теме, — «Алгоритмы. Построение и анализ» Томаса Кормена (Thomas H. Corman) и «Алгоритмы. Руководство по разработке» Стивена Скиены (Steven Skiena), второе издание. Это отличные книги, по которым можно научиться построению как однопоточных, так и параллельных алгоритмов.

Многие из этих источников издержек будут упоминаться и далее в этой главе (а также в главе 6, посвященной производительности).

4.1.5. Пример обработчика транзакций

Завершая этот теоретический раздел, применим изученную теорию на практике и спроектируем образец приложения, выполняющего параллельную обработку. Рассмотрим в обобщенном виде, как это можно сделать с помощью классов из библиотеки `java.util.concurrent`.

Представьте себе несложную систему для обработки транзакций. Простой и стандартный способ создания такого приложения — обеспечить соответствие различных элементов приложения различным этапам интересующего нас процесса. В таком случае каждая фаза будет представлена в виде пула потоков. Приложение будет брать элементы по очереди, выполнять над каждым из них определенные манипуляции, а потом передавать элемент следующему пулу потоков. Вообще, при качественной организации приложения каждый пул потоков должен концентрироваться на такой обработке, которая относится к конкретной функциональной области. Пример приложения продемонстрирован на рис. 4.1.

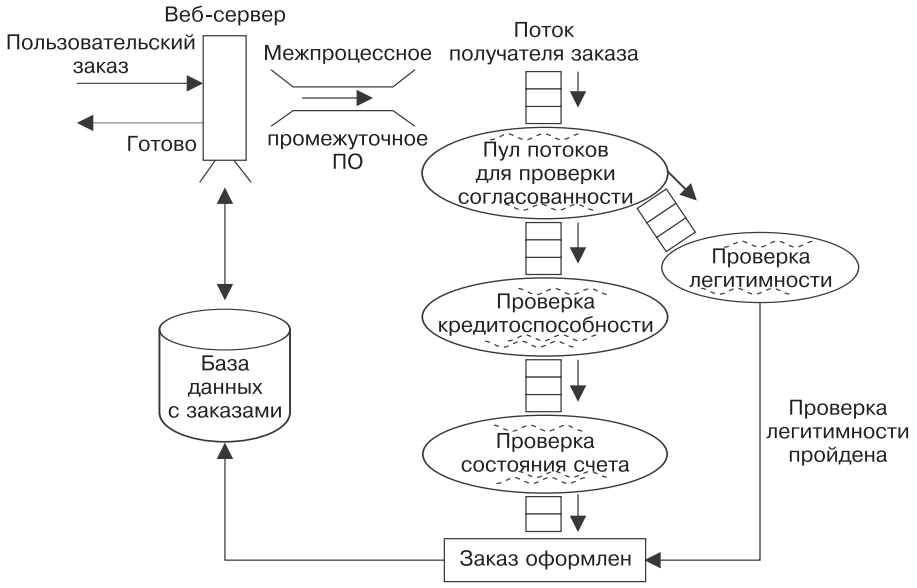


Рис. 4.1. Пример многопоточного приложения

Если вы проектируете приложения именно так, то можете увеличить их пропускную способность, поскольку можно одновременно задействовать несколько элементов. Один элемент может заниматься обработкой фазы «Проверка кредитоспособности», а другому элементу поручим обработку фазы «Проверка состояния счета». В зависимости от тонкостей приложения в фазе «Проверка состояния счета» могут даже одновременно обрабатываться различные команды.

При подобном проектировании получаются приложения, очень удобные для реализации с помощью классов, входящих в состав `java.util.concurrent`. В пакете содержатся пулы потоков, предназначенные для исполнения (а в классе `Executors` есть хороший набор фабричных методов для создания таких пулов) и очереди, позволяющие распределять работу между пулами. Кроме того, здесь есть параллельные структуры данных (для создания разделяемых кэшей и других практических нужд), а также многие другие полезные низкоуровневые инструменты.

Но может возникнуть вопрос: «Хорошо, а как решались подобные проблемы до появления Java 5, когда у нас еще не было этих классов?» Зачастую группы приложений выпускались с собственными наборами библиотек для параллельного программирования — и получались составные компоненты, напоминающие по назначению те, что сегодня находятся в `java.util.concurrent`. Но многие такие специальные компоненты могли вызывать проблемы при проектировании, а также тонкие (и не слишком тонкие) ошибки, возникающие при параллелизме. Если бы `java.util.concurrent` не было, то разработчикам приложений пришлось бы заново изобретать многие из таких компонентов самостоятельно (возможно, получались бы непричесанные, плохо протестированные образцы).

Итак, опираясь на приведенный пример, перейдем к изучению следующей темы — «классического» параллелизма Java. Отдельно остановимся на том, почему может быть сложно программировать с применением такого подхода.

4.2. Параллельная обработка с блочной структурой (до Java 5)

Значительная часть этой главы посвящена обсуждению подходов, представляющих альтернативу блочно-синхронизационному методу обеспечения параллелизма. Но, чтобы такое обсуждение альтернатив получилось максимально полезным, важно четко понимать, что хорошего и что плохого в традиционном подходе к параллельной обработке.

Для этого мы обсудим первоначальный, довольно низкоуровневый подход к многопоточному программированию, при котором применялись ключевые слова Java для параллельной обработки — `synchronized`, `volatile` и т. д. Мы поговорим об этом в контексте структурных сил и с учетом того, о чем пойдет речь в следующих разделах.

Затем мы кратко рассмотрим жизненный цикл потока, а потом обсудим распространенные техники (и подводные камни) параллельного кода — полностью синхронизированные объекты, взаимные блокировки, ключевое слово `volatile` и неизменяемость.

Начнем с обзора синхронизации.

4.2.1. Синхронизация и блокировки

Как вы уже знаете, ключевое слово `synchronized` может применяться либо к блоку, либо к методу. Оно указывает, что перед входом в блок или метод поток должен получить соответствующую блокировку. Для метода это означает получение блокировки, относящейся к экземпляру объекта (либо блокировки, относящейся к объекту `Class`, — для методов `static synchronized`). Применительно к блоку программист обязан указать, блокировка какого объекта должна быть получена.

Лишь один поток может одновременно проходить через все синхронизированные блоки или методы объекта. Если в это пытаются вмешаться другие потоки, то они приостанавливаются виртуальной машиной Java. Это правило соблюдается независимо от того, пытается ли сторонний поток вмешаться в тот же либо в иной синхронизированный блок конкретного объекта. В теории параллелизма такая конструкция именуется *критической секцией*.

ПРИМЕЧАНИЕ

А вы задумывались, почему для работы с критической секцией в Java используется ключевое слово synchronized? Почему не critical или locked? Что значит быть синхронизированным? Мы вернемся к этой теме в разделе 4.2.5, но если вы не знаете ответа на вопрос либо никогда об этом не задумывались, то можете пару минут подумать, а потом читать дальше.

На самом деле в этой главе мы говорим о сравнительно новых технологиях параллельной обработки. Но, беседуя о синхронизации, остановимся на некоторых базовых фактах, касающихся синхронизации и блокировок в Java. Надеемся, что вы помните все следующие факты (или большинство из них).

- Блокировать можно только объекты — но не примитивы.
- При блокировании массива объектов отдельные объекты, входящие в его состав, не блокируются.
- Синхронизированный метод можно считать аналогом блока `synchronized (this) { ... }`, охватывающего целый код (правда, в байт-коде синхронизированный метод и такой блок представляются по-разному).
- Метод `static synchronized` блокирует объект `Class`, поскольку отсутствует какой-либо экземпляр объекта для блокировки.
- Если вы хотите заблокировать объект класса, хорошо взвесьте, стоит ли делать это явно либо путем использования `getClass()`, поскольку на уровне подклассов два этих подхода отразятся по-разному.
- Синхронизация во внутреннем классе не зависит от внешнего класса (чтобы понять, почему так происходит, вспомните, как реализуются внутренние классы).
- Ключевое слово `synchronized` не входит в состав сигнатуры метода. Таким образом, оно не может появляться в интерфейсе при объявлении метода.
- Несинхронизированные методы игнорируют и не проверяют состояния каких-либо блокировок. Несинхронизированные методы могут работать параллельно с синхронизированными методами.

- Блокировки Java допускают многократное вхождение. Это означает, что если поток, удерживающий блокировку, встречает точку синхронизации для этой же блокировки (например, синхронизированный метод вызывает другой синхронизированный метод в этом же классе), то ему разрешается продолжить работу.

ВНИМАНИЕ

Схемы блокировки, не допускающие повторного вхождения (называемые также нереентерабельными) существуют в других языках, и такие схемы можно синтезировать в Java. Если вас интересуют подробности, почитайте в Javadoc раздел о ReentrantLock в java.util.concurrent.locks. Правда, с такими схемами обычно неудобно работать и их лучше избегать, если вы только не знаете абсолютно точно, что делаете.

На этом обзор синхронизации в Java можно закончить. Теперь поговорим о состояниях, в которых поток оказывается на протяжении своего жизненного цикла.

4.2.2. Модель состояния для потока

На рис. 4.2 показано, как протекает жизненный цикл потока — от создания к запуску, возможной приостановке, блокированию на ресурсе или возобновлению работы и, наконец, к завершению.

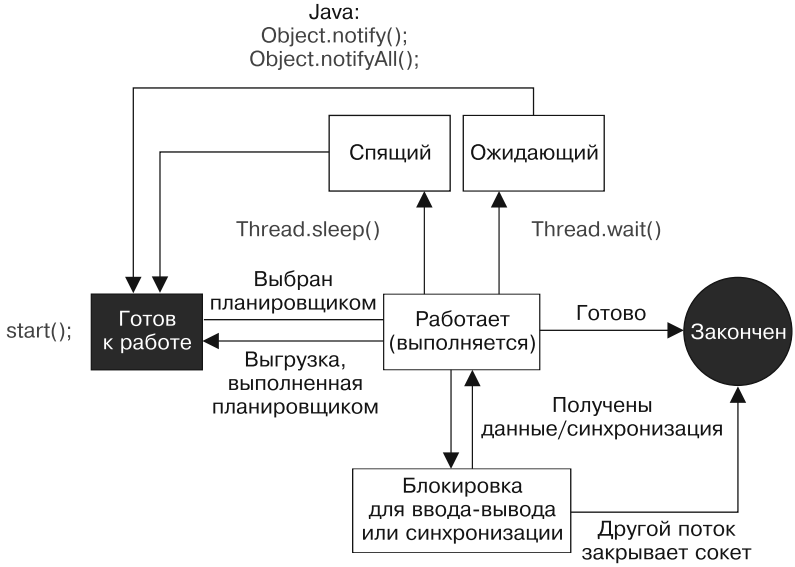


Рис. 4.2. Модель состояний потока Java

Изначально поток создается в состоянии Ready (Готов). После этого планировщик находит ядро, в котором можно запустить данный поток. Если машина сильно загружена, то перед началом работы возможен небольшой период ожидания. Затем поток обычно переходит к потреблению выделенного ему времени, а когда доступное

время будет израсходовано, он перейдет в состояние Ready, чтобы дождаться поступления новых временных квантов от процессора. Именно так организуется принудительная диспетчеризация потоков, о которой мы говорили в подразделе 4.1.1.

Поток может не только приступить к действию по команде планировщика, но и указать, что пока не может начать работу на данном ядре. Это может быть связано с тем, что код программы приказывает данному потоку приостановить работу перед продолжением (посредством `Thread.sleep()`) или дожидается определенного уведомления (обычно о том, что выполнено какое-то внешнее условие). В такой ситуации поток удаляется из ядра и высвобождает все свои блокировки. Для повторного запуска он должен быть «разбужен» (после того как «проспит» заданное количество времени либо получит соответствующий сигнал). «Разбуженный» поток вновь переходит в состояние Ready.

Поток может быть заблокирован, если он ожидает операции ввода-вывода или должен получить блокировку, пока удерживаемую другим потоком. В таком случае поток не выгружается из ядра, но остается занятым, дожидаясь, пока данные или блокировка станут доступны. Как только это произойдет, выполнение потока продолжится до тех пор, пока не будет израсходован выделенный ему квант времени.

Теперь поговорим об одном широко известном способе решения проблем, которые могут возникать при синхронизации. Речь пойдет о полностью синхронизированных объектах.

4.2.3. Полностью синхронизированные объекты

Выше в этой главе мы затрагивали концепцию безопасности типов при параллельной обработке и упоминали об одной из стратегий обеспечения такой безопасности (см. врезку «О долговременной безопасности»). Обратимся к более подробному описанию этой стратегии. Ее центральный феномен — полностью синхронизированные объекты. Если соблюдаются все приведенные ниже правила, то класс определенно является потокобезопасным и «живым».

Итак, полностью синхронизированный класс — это класс, удовлетворяющий следующим условиям:

- все поля всегда инициализируются в согласованное состояние в каждом конструкторе;
- отсутствуют общедоступные поля;
- экземпляры объектов гарантированно являются согласованными после возврата из любого незакрытого (non-private) метода (при условии, что на момент вызова метода состояние было согласованным);
- все методы доказуемо завершаются в ограниченное время;
- все методы синхронизированы;
- при нахождении в несогласованном состоянии не выполняются вызовы к методам другого экземпляра;

- при нахождении в несогласованном состоянии не выполняются вызовы к какому-либо незакрытому методу.

В листинге 4.1 показан пример такого класса как элемента внутреннего интерфейса воображаемого распределенного сервиса микроблогов. Класс `ExampleTimingNode` будет получать обновления с помощью метода `propagateUpdate()`. Кроме того, у этого класса можно запросить, получал ли он конкретное уведомление. Данная ситуация демонстрирует классический конфликт между операциями чтения и записи. Поэтому синхронизация применяется именно для того, чтобы избежать несогласованности.

Листинг 4.1. Полностью синхронизированный класс

```
public class ExampleTimingNode implements SimpleMicroBlogNode {

  private final String identifier;

  private final Map<Update, Long> arrivalTime
  ➥ = new HashMap<>();

  public ExampleTimingNode(String identifier_) {
    identifier = identifier_;
  }

  public synchronized String getIdentifier() {
    return identifier;
  }

  public synchronized void propagateUpdate(
  ➥ Update update_) {
    long currentTime = System.currentTimeMillis();
    arrivalTime.put(update_, currentTime);
  }

  public synchronized boolean confirmUpdateReceived(
  ➥ Update update_) {
    Long timeRecvd = arrivalTime.get(update_);
    return timeRecvd != null;
  }
}
```

Отсутствуют общедоступные поля

Все поля инициализируются в конструкторе

Все методы синхронизируются

На первый взгляд, просто фантастика — класс получился одновременно безопасным и «живым». Проблемы возникают только с производительностью — ведь безопасный и «живой» класс совсем не обязательно должен быть очень быстрым. Приходится использовать `synchronized`, чтобы координировать все операции доступа (и `get`, и `put`) в контейнере `arrivalTime`. В конечном итоге именно такая блокировка вас и замедляет. Это — центральная проблема при организации параллелизма таким способом.

ХРУПКОСТЬ КОДА

Код из листинга 4.1 не только имеет проблемы с производительностью, но и характеризуется достаточной хрупкостью. Как видите, вы не затрагиваете `arrivalTime` вне метода `synchronized` (действительно, весь доступ ограничивается операциями `get` и `put`), но это возможно лишь потому, что у нас здесь совсем немного кода. В реальных крупных системах это будет невозможно именно из-за большого объема работающего кода. Ошибки очень легко просачиваются в огромные базы кода, использующие такой подход. Это еще одна причина, по которой сообщество Java занялось поисками более надежных подходов.

4.2.4. Взаимные блокировки

Следующая классическая проблема параллелизма (над которой бьется не только Java) — это *взаимная блокировка* (deadlock). Рассмотрим листинг 4.2, в котором мы немного расширили предыдущий пример. В этой версии не только записывается время каждого обновления, но и каждый узел, получающий обновление, информирует об этом другой узел.

Это упрощенная попытка построить мнгопоточную систему обработки обновлений. Пример специально составлен для демонстрации взаимного блокирования — не используйте его в качестве основы для реального кода.

Листинг 4.2. Пример взаимных блокировок

```
public class MicroBlogNode implements SimpleMicroBlogNode {
  private final String ident;

  public MicroBlogNode(String ident_) {
    ident = ident_;
  }

  public String getIdent() {
    return ident;
  }

  public synchronized void propagateUpdate(Update upd_, MicroBlogNode
    backup_) {
    System.out.println(ident +": recvd: "+ upd_.getUpdateText()
    ➥ +" ; backup: "+backup_.getIdent());
    backup_.confirmUpdate(this, upd_);
  }

  public synchronized void confirmUpdate(MicroBlogNode other_, Update
    update_) {
    System.out.println(ident +": recvd confirm: "+
    ➥ update_.getUpdateText() +" from "+other_.getIdent()k);
  }
}

final MicroBlogNode local =
➥ new MicroBlogNode("localhost:8888");
```

Необходимо ключевое слово final

```
final MicroBlogNode other = new MicroBlogNode("localhost:8988");
final Update first = getUpdate("1");
final Update second = getUpdate("2");

new Thread(new Runnable() {
  public void run() {
    local.propagateUpdate(first, other);
  }
}).start();

new Thread(new Runnable() {
  public void run() {
    other.propagateUpdate(second, local);
  }
}).start();
```

Первое обновление, отправляемое первому потоку

Второе сообщение, отправляемое другому потоку

На первый взгляд, код вполне нормальный. У вас есть два обновления, отправляемые к отдельным потокам. Каждое обновление должно подтверждаться в резервном потоке. Такая структура не кажется безумной — если какой-то поток не сработает, то у нас предусмотрен второй поток, который может подхватить его работу.

Если вы запустите этот код, то, скорее всего, возникнет взаимная блокировка — оба потока доложат о получении сообщения, но ни один не получит обновления, для которого он является резервным потоком. Причина в том, что каждый поток требует от второго освободить удерживаемую блокировку, прежде чем метод подтверждения сможет сработать. Ситуация проиллюстрирована на рис. 4.3.

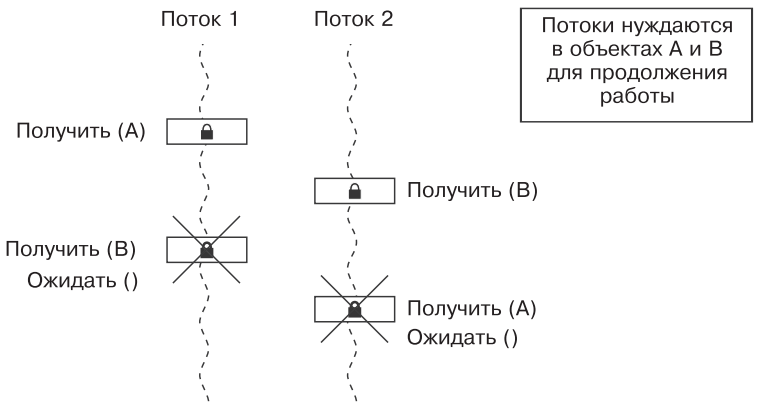


Рис. 4.3. Потоки, оказавшиеся во взаимной блокировке

Один из методов, позволяющий не попадать во взаимные блокировки, предполагает всегда получать блокировки для каждого потока в одном и том же порядке. В предыдущем примере первый поток, чтобы начать работу, получает блокировки в порядке A, B, а второй поток — в порядке B, A. Если бы оба потока «настаивали» на получении блокировок в порядке A, B, то взаимных блокировок можно было бы

избежать, так как второй поток вообще не сработает, если первый завершит работу и высвободит все свои блокировки.

Говоря в терминах подхода с полной синхронизацией объектов, взаимной блокировки удается избежать за счет того, что код нарушает правило согласованности состояния. По прибытии сообщения получающий узел вызывает другой объект, пока обработка сообщения еще продолжается, то есть при совершении такого вызова состояние является несогласованным.

Теперь вернемся к загадке, сформулированной ранее: почему для критической секции в Java используется ключевое слово `synchronized`. Так мы подойдем к теме неизменяемости и обсудим ключевое слово `volatile`.

4.2.5. Почему synchronized?

Одно из самых значительных изменений, произошедших в области параллельного программирования в последние годы, относится к производству оборудования. Еще не так давно практикующий программист мог годами работать на одноядерных системах или, в крайнем случае, встретить систему с двумя ядрами. Поэтому параллельное программирование вполне логично понималось как работа, в основе которой лежит разделение процессорного времени. Потоки загружались и выгружались в единственном ядре.

Сегодня практически все устройства крупнее мобильного телефона имеют по нескольку ядер, поэтому изменяется и представление о многоядерном программировании. Большие группы потоков могут работать на разных ядрах в один и тот же физический момент времени (потенциально они могут одновременно оперировать разделяемыми данными). Эта ситуация показана на рис. 4.4. Ради обеспечения эффективности каждый из одновременно работающих потоков может иметь собственную копию кэшированных данных, которыми он и оперирует. Итак, представим себе такую картину и поговорим о том, почему слово `synchronized` было выбрано для обозначения заблокированной секции или метода.

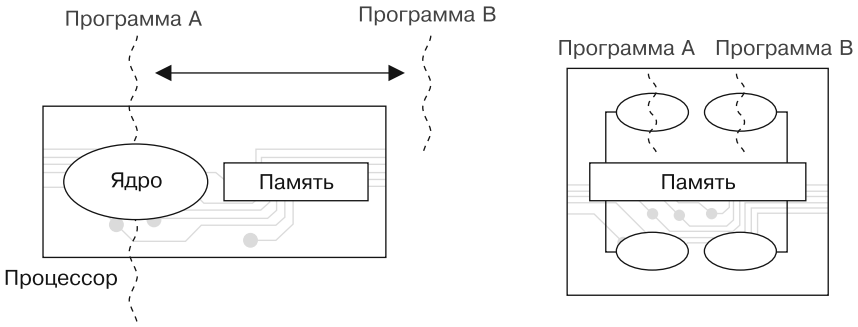


Рис. 4.4. Старый и новый способ представления параллелизма и потоков

Выше мы задавали вопрос: а что же синхронизируется в коде в листинге 4.1? Ответ таков: *синхронизируется представление памяти в различных потоках бло-*

кируемого объекта. Это означает, что после завершения работы блока (или метода) `synchronized` абсолютно все изменения, сделанные в блокированном объекте, сбрасываются в основную память прежде, чем блокировка будет снята, — как показано на рис. 4.5.

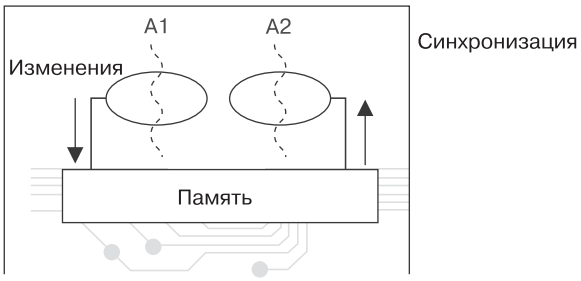


Рис. 4.5. Изменения объекта проникают между потоками через основную память

Кроме того, при входе в блок `synchronized` и последующем получении блокировки все изменения, касающиеся заблокированного объекта, считываются из основной памяти. Таким образом, поток с блокировкой синхронизируется с представлением объекта в основной памяти еще до того, как начнется исполнение кода в заблокированной секции.

4.2.6. Ключевое слово volatile

Ключевое слово `volatile` существует в Java с первых дней существования языка (появилось в Java 1.0). Оно применяется как простой инструмент для синхронизации полей объектов, в том числе при работе с примитивами. Изменчивое (volatile) поле регулируется следующими правилами:

- любое значение, видимое потоком, перед использованием всегда повторно считывается из основной памяти;
- любое значение, записываемое потоком, всегда пробрасывается в основную память до того, как завершится выполнение инструкции.

Можно представить, что такая операция обернута в крошечный блок `synchronized`. Благодаря подобной конструкции программист может писать упрощенный код, но за счет дополнительных перебросок данных при каждом доступе. Обратите также внимание на то, что переменная `volatile` не вызывает никаких блокировок. Поэтому при ее использовании вы гарантированно не попадете во взаимную блокировку.

Несколько менее очевидное следствие использования переменных `volatile` заключается в том, что при обеспечении истинной безопасности потоков переменная `volatile` должна использоваться только для моделирования такой переменной, запись в которую не зависит от текущего (считываемого) состояния. В тех случаях, когда важно учитывать актуальное состояние, всегда нужно применять блокировку для обеспечения полной безопасности.

4.2.7. Неизменяемость

В данном контексте очень ценной представляется техника, позволяющая использовать неизменяемые объекты. Это объекты, либо вообще не имеющие состояния, либо содержащие только поля `final` (которые, следовательно, должны заполняться в конструкторах объектов). Они всегда остаются безопасными и «живыми», так как их состояние не может быть изменено. Соответственно, они никогда не могут оказаться в несогласованном состоянии.

Одна из проблем заключается в том, что все значения, требуемые для инициализации конкретного объекта, должны передаваться в конструктор. Это может приводить к неудобным вызовам конструктора с применением многочисленных параметров. В качестве альтернативы многие программисты пользуются методом `FactoryMethod`. Работа с ним может сводиться к вызову статического (static) класса вместо конструктора для создания новых объектов. Конструкторы обычно делаются защищенными или закрытыми (protected или private). Таким образом, инстанцирование может быть выполнено лишь с помощью статических методов `FactoryMethod`.

Однако по-прежнему сохраняется проблема, связанная с потенциальной необходимостью передачи многочисленных параметров методу `FactoryMethod`. Это не всегда бывает удобно, особенно если требуется аккумулировать состояние из нескольких источников, прежде чем создать новый неизменяемый объект.

Для решения этой проблемы можно использовать паттерн Builder («Строитель»). Он представляет собой комбинацию двух конструкций: статического внутреннего класса, реализующего обобщенный интерфейс строителя, и закрытого конструктора для неизменяемого класса как такового.

Статический внутренний класс выступает как построитель неизменяемого класса и предоставляет разработчику единственную возможность получать контроль над новыми экземплярами неизменяемого типа. Один из распространенных вариантов реализации класса `Builder` предполагает, что у этого класса будут точно такие же поля, как и у неизменяемого класса, но будет разрешено изменение полей.

В листинге 4.3 показано, как такая конструкция может использоваться для моделирования обновлений в микроблогах (опять же мы опираемся на предыдущие листинги из этой главы).

Листинг 4.3. Неизменяемые объекты и строители

```
public interface ObjBuilder<T> {
  T build();
}

public class Update {
  private final Author author;
  private final String updateText;

  private Update(Builder b_) {
    author = b_.author;
    updateText = b_.updateText;
```

Интерфейс строителя

Финальные поля должны инициализироваться в конструкторе

```
  }

  public static class Builder
  ➥ implements ObjBuilder<Update> {
    private Author author;
    private String updateText;

    public Builder author(Author author_) {
      author = author_;
      return this;
    }

    public Builder updateText(String updateText_) {
      updateText = updateText_;
      return this;
    }

    public Update build() {
      return new Update(this);
    }
  }

}
```

Класс-строитель должен быть внутренним и статическим

Методы, применяемые к строителю, возвращают строитель для цепных вызовов

Методы hashCode() и equals() опущены

Имея этот код, вы затем можете создать новый объект `Update` следующим образом:

```
Update.Builder ub = new Update.Builder();
Update u = ub.author(myAuthor).updateText("Hello").build();
```

Этот паттерн очень распространен не только в теории, но и на практике. На самом деле мы уже пользовались свойствами неизменяемых объектов в листингах 4.1 и 4.2.

Последнее замечание, которое хотелось бы сделать о неизменяемых объектах, — ключевое слово `final` применяется только к тому объекту, на который оно указывает. Как показано на рис. 4.6, ссылка на главный объект не может быть присвоена для указания на объект 3. Но внутри объекта ссылка на объект 1 может быть переброшена так, чтобы она стала указывать на объект 2. Иными словами — ссылка `final` может указывать на объект, имеющий нефинальные поля.

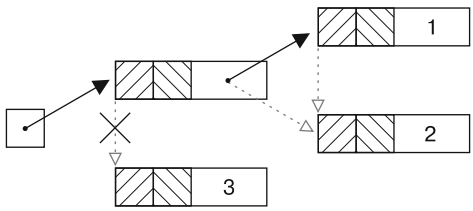


Рис. 4.6. Неизменяемость значения по сравнению со ссылкой

Неизменяемость — очень мощная техника, ее всегда следует задействовать, если это осуществимо. Иногда бывает невозможно добиться эффективной разработки с применением одних лишь неизменяемых объектов, поскольку каждое изменение состояния объекта требует создания нового объекта. Итак, иногда бывает необходимо работать именно с изменяемыми объектами.

Далее мы перейдем к одной из важнейших тем данной главы — поговорим о современных и вместе с тем простых API для обеспечения параллелизма. Они содержатся в библиотеке `java.util.concurrent`. Рассмотрим, как можно приступить к их использованию в вашем собственном коде.

4.3. Составные элементы современных параллельных приложений

После выхода Java 5 в этом языке появился новый способ понимания параллелизма. Все началось с пакета `java.util.concurrent`, в котором содержится богатый новый инструментарий для работы с многопоточным кодом. Этот инструментарий был усовершенствован в более новых версиях Java, но классы и пакеты, появившиеся в Java 5, по-прежнему работают без изменений. Они до сих пор очень ценны для практикующего разработчика.

Мы сделаем экспресс-тур по некоторым основным классам `java.util.concurrent` и связанным с ними пакетам. В частности, поговорим о пакете для работы с атомарными объектами и пакете для работы с блокировками. Мы поможем вам начать работать с классами и покажем примеры их использования. Кроме того, вам следует прочитать соответствующие разделы Javadoc, чтобы получить целостное впечатление обо всех пакетах, — с ними программирование классов для параллельной обработки значительно упрощается.

МИГРАЦИЯ КОДА

Если у вас есть работающий многопоточный код, который был написан с применением старых подходов (до Java 5), то нужно выполнить его рефакторинг с помощью `java.util.concurrent`. По нашему опыту, код улучшается, если сознательно работать над его портированием на новые API. Повысится ясность и надежность кода, а это практически всегда стоит усилий, затрачиваемых на обеспечение миграции.

Считайте это обсуждение вводным курсом по работе с конкурентным кодом, а не полномасштабным тематическим семинаром. Чтобы максимально эффективно использовать `java.util.concurrent`, нужно прочитать гораздо больше, чем мы можем рассказать в этой книге.

4.3.1. Атомарные классы — java.util.concurrent.atomic

В пакете `java.util.concurrent.atomic` содержится несколько классов, названия которых начинаются с `Atomic`. В сущности, они обеспечивают такую же семантику, как и `volatile`. Но они обернуты в класс-API, содержащий атомарные методы (дей-

ствующие по принципу «все или ничего») для подходящих операций. Для разработчика это может быть очень простой способ избежать условий гонки при операциях над совместно применяемыми данными.

Реализации написаны так, чтобы максимально эффективно использовать характеристики современных процессоров. Поэтому операции могут быть неблокирующими (свободными от блокировок), если необходимая поддержка доступна на уровне оборудования и операционной системы. Такие процессоры и операционные системы используются на большинстве современных компьютеров, а подобные классы обычно применяются для реализации последовательных номеров. Для этого предназначен атомарный метод `getAndIncrement()`, который есть у `AtomicInteger` или `AtomicLong`.

Чтобы представлять номер последовательности, класс должен иметь метод `nextId()`, который будет возвращать гарантированно уникальный (причем обязательно возрастающий) номер при каждом вызове. Примерно такая же концепция последовательных номеров присутствует и в базах данных (отсюда и название переменной).

Рассмотрим небольшой фрагмент кода, который реализует номер последовательности:

```
private final AtomicLong sequenceNumber = new AtomicLong(0);

public long nextId() {
  return sequenceNumber.getAndIncrement();
}
```

ПРЕДОСТЕРЕЖЕНИЕ

Атомарные классы не наследуют от классов, имеющих схожие названия. Так, AtomicBoolean нельзя использовать вместо Boolean, а AtomicInteger — вместо Integer (хотя AtomicInteger дополняет класс Number).

Далее поговорим о том, как в `java.util.concurrent` моделируется основной компонент системы синхронизации — интерфейс `Lock`.

4.3.2. Блокировки — java.util.concurrent.locks

В основе блочного подхода к синхронизации лежит, в сущности, простая концепция блокировки. Такой подход имеет несколько недостатков:

- существует всего один тип блокировки;
- блокировка одинаково применяется ко всем синхронизированным операциям, осуществляемым над заблокированным объектом;
- блокировка приобретается в самом начале синхронизированного блока или метода;
- либо приобретается блокировка, либо поток блокируется — третьего не дано.

Если вы собираетесь переработать поддержку блокировок, то можно попробовать изменить к лучшему несколько следующих моментов:

- добавить различные типы блокировок (например, для чтения и для записи);
- не ограничивать блокировки в рамках блоков (скажем, обеспечить возможность блокировки в одном методе и разблокировки — в другом);

- если поток не может получить блокировку (например, потому, что блокировку уже имеет другой поток), нужно позволить ему восстановить предыдущее состояние либо приняться за выполнение какой-то другой задачи (с помощью метода `tryLock()`);
- позволить потоку попытаться получить блокировку и прекратить такие попытки по истечении определенного периода времени.

Основной компонент для воплощения всех этих возможностей — интерфейс `Lock` из пакета `java.util.concurrent.locks`. Он предлагается вместе с парой реализаций.

- `ReentrantLock` — в сущности, это эквивалент уже известной нам блокировки, применяемой с синхронизированными блоками Java. Такая блокировка немного гибче обычной.
- `ReentrantReadWriteLock` — эта блокировка обеспечивает повышение производительности в тех случаях, когда код содержит много считывающих и сравнительно мало записывающих элементов.

Интерфейс `Lock` может использоваться для полного копирования любого функционала, который доступен при блочно-структурированном параллелизме. В листинге 4.4 приведен переработанный пример с взаимной блокировкой, теперь в нем используется вариант `ReentrantLock`.

Листинг 4.4. Вариант примера с взаимной блокировкой и использованием ReentrantLock

```
private final Lock lock = new ReentrantLock();

public void propagateUpdate(Update upd_, MicroBlogNode backup_) {
  lock.lock();
  try {
    System.out.println(ident +": recvd: "+
    ➥ upd_.getUpdateText() +" ; backup: "+
    ➥ backup_.getIdent());
    backup_.confirmUpdate(this, upd_);
  } finally {
    lock.unlock();
  }
}

public void confirmUpdate(MicroBlogNode other_, Update upd_) {
  lock.lock();
  try {
    System.out.println(iden +": recvd confirm: "+
    ➥ upd_.getUpdateText() +" from "+ other_.getIdentifier());
  } finally {
    lock.unlock();
  }
}
```

Каждый поток сначала попадает в собственную блокировку

Вызов confirmUpdate() для подтверждения в другом потоке

Попытка заблокировать другой поток 1

Попытка заблокировать другой поток 1, скорее всего, не удастся, поскольку он уже заблокирован (см. рис. 4.3). Так и возникает взаимная блокировка.

ИСПОЛЬЗОВАНИЕ TRY ... FINALLY С БЛОКИРОВКАМИ

Вариант метода `lock()` с блоком `try ... finally`, где и снимается блокировка, — хорошее дополнение к вашему инструментарию. Такая конструкция очень удобна, если воспроизвести ситуацию, напоминающую ту, что мы рассматривали при изучении блочно-структурированного параллелизма. С другой стороны, если вам требуется пересылать объекты `Lock` (например, при возврате этого объекта от метода), то такой вариант метода использовать не получится.

Работа с объектами `Lock` может быть гораздо более многообещающей, чем применение блочно-структурированного подхода. Но все же использовать такие объекты для разработки надежной стратегии блокировок бывает довольно затруднительно.

Существует несколько стратегий, позволяющих справляться с взаимными блокировками. Но есть среди них и такая, которая обычно не работает, — и об этом необходимо знать. Рассмотрим вариант метода `propagateUpdate()`, показанный в листинге 4.5 (и представьте себе, что такие же изменения внесены в код `confirmUpdate()`). В данном примере мы поменяли безусловную блокировку на метод `tryLock()` с задержкой. Ниже мы попытаемся устранить взаимную блокировку, дав другим потокам возможность получить блокировку.

Листинг 4.5. Некорректный вариант кода для устранения взаимной блокировки

```
public void propagateUpdate(Update upd_, MicroBlogNode backup_) {
  boolean acquired = false;

  while (!acquired) {
    try {
      int wait = (int)(Math.random() * 10);
      acquired = lock.tryLock(wait, TimeUnit.MILLISECONDS);
      if (acquired) {
        System.out.println(ident +": recvd: "+
          upd_.getUpdateText() +" ; backup: "+backup_.getIdent());
        backup_.confirmUpdate(this, update_);
      } else {
        Thread.sleep(wait);
      }
    } catch (InterruptedException e) {
    } finally {
      if (acquired) lock.unlock();
    }
  }
}
```

Пытаемся и блокируем с задержкой случайной длительности

Подтверждение в другом потоке

Разблокируем только при наличии блокировки

Если запустить код из листинга 4.5, то может создаться впечатление, что взаимная блокировка снимается. Но в действительности это происходит лишь иногда. Далеко не всегда вы увидите текст received confirm of update (получено подтверждение об обновлении).

На самом деле взаимная блокировка не снимается. Ведь если исходная блокировка получена (в методе `propagateUpdate()`), то поток вызывает `confirmUpdate()` и так и не освобождает первую блокировку до завершения. Если обоим потокам удастся получить первую блокировку до того, как любой из них вызовет `confirmUpdate()`, то эти потоки окажутся во взаимной блокировке.

Настоящее решение заключается в том, чтобы гарантировать следующее: если попытка получить вторую блокировку не увенчается успехом, то поток должен высвободить удерживаемую им блокировку и немного переждать, как показано в листинге 4.6. В таком случае у остальных потоков будет возможность получить полный набор блокировок, необходимый для продолжения работы.

Листинг 4.6. Устранение взаимной блокировки

```
public void propagateUpdate(Update upd_, MicroBlogNode backup_) {
  boolean acquired = false;
  boolean done = false;

  while (!done) {
    int wait = (int)(Math.random() * 10);
    try {
      acquired = lock.tryLock(wait, TimeUnit.MILLISECONDS);
      if (acquired) {
        System.out.println(ident +": recvd: "+
        ➥ upd_.getUpdateText() +" ; backup: "+backup_.getIdent());
        done = backupNode_.tryConfirmUpdate(this, update_);
      }
    } catch (InterruptedException e) {
    } finally {
      if (acquired) lock.unlock();
    }
    if (!done) try {
      Thread.sleep(wait);
    } catch (InterruptedException e) { }
  }
}

public boolean tryConfirmUpdate(MicroBlogNode other_, Update upd_) {
  boolean acquired = false;
  try {
    int wait = (int)(Math.random() * 10);
    acquired = lock.tryLock(wait, TimeUnit.MILLISECONDS);

    if (acquired) {
      long elapsed = System.currentTimeMillis() - startTime;
      System.out.println(ident +": recvd confirm: "+
      ➥ upd_.getUpdateText() +" from "+other_.getIdent()
      ➥ +" - took "+ elapsed +" millis");
      return true;
    }
```

Проверяем возврат от tryConfirmUpdate()

Если не выполнено — высвобождаем блокировку и ждем

```
    } catch (InterruptedException e) {
    } finally {
      if (acquired) lock.unlock();
    }

    return false;
  }
```

В данной версии вы проверяете наличие кода возврата от `tryConfirmUpdate()`. При получении `false` исходная блокировка будет снята. Поток ненадолго приостановит работу, что потенциально должно позволить другому потоку получить блокировку.

Запустите этот код несколько раз. Вы убедитесь, что оба потока, в принципе, всегда срабатывают — от взаимной блокировки удалось избавиться. Можете поэкспериментировать с какими-нибудь другими приведенными выше вариантами кода, содержащими взаимные блокировки. Мы имеем в виду оригинал, неверную попытку откорректировать оригинал и второй вариант коррекции. Упражняясь с кодом, вы можете лучше понять, что происходит с блокировками, а также выработать интуицию, которая поможет справляться с проблемой взаимных блокировок.

ПОЧЕМУ СОЗДАЕТСЯ ВПЕЧАТЛЕНИЕ, ЧТО ПОКАЗАННЫЙ ВЫШЕ НЕКОРРЕКТНЫЙ ВАРИАНТ ИНОГДА РАБОТАЕТ?

Как мы убедились, в некорректном варианте кода (см. листинг 4.5) взаимная блокировка никуда не девается. Почему же это решение иногда срабатывает? Дело в том, что код дополнительно усложнен. Он влияет на работу планировщика потоков виртуальной машины Java, делая его менее предсказуемым. Это означает, что иногда диспетчеризация потоков будет складываться так, что один из них (обычно первый поток) оказывается способен перейти в `confirmUpdate()` и получить вторую блокировку до того, как сможет начать работу второй поток. Такая ситуация возможна и в исходном варианте кода, но там она гораздо менее вероятна.

Мы лишь слегка коснулись возможностей `Lock` — есть и множество других способов создавать более сложные структуры, напоминающие блокировки. Об одной из подобных концепций — защелке — мы и поговорим далее.

4.3.3. CountDownLatch

`CountDownLatch` — это простой паттерн синхронизации, позволяющий множественным потокам регламентировать между собой минимальные приготовления, которые должны быть выполнены прежде, чем любой из них сможет преодолеть барьер синхронизации.

Для этого предоставляется целочисленное значение `int` (счет) при создании нового экземпляра `CountDownLatch`. Затем для управления защелкой (latch) применяется два метода: `countDown()` и `await()`. Первый уменьшает счет на 1, а второй заставляет вызывающий поток выжидать, пока счет не будет равен 0 (второй метод ничего не делает, если счет уже достиг 0 или менее). Этот простой механизм обеспечивает легкое развертывание паттерна минимальных приготовлений.

В листинге 4.7 группа потоков, работающих внутри единого процесса, «пытается узнать», была ли хотя бы половина из этих потоков правильно инициализирована. При этом предполагается, что на инициализацию потока уходит определенное время. Лишь если такая проверка закончится успешно, система начинает посылать уведомления этим потокам.

Листинг 4.7. Использование защелок для упрощения инициализации

```
public static class ProcessingThread extends Thread {
  private final String ident;
  private final CountDownLatch latch;

  public ProcessingThread(String ident_, CountDownLatch cdl_) {
    ident = ident_;
    latch = cdl_;
  }
  public String getIdentifier() {
    return identifier;
  }
  public void initialize() {
    latch.countDown();
  }
  public void run() {
    initialize();
  }
}

final int quorum = 1 + (int)(MAX_THREADS / 2);
final CountDownLatch cdl = new CountDownLatch(quorum);

final Set<ProcessingThread> nodes = new HashSet<>();
try {
    for (int i=0; i<MAX_THREADS; i++) {
      ProcessingThread local = new ProcessingThread("localhost:"+
      ➥ (9000 + i), cdl);
      nodes.add(local);
      local.start();
    }
    cdl.await();
  } catch (InterruptedException e) {
  } finally {
}
```

Инициализируем узел

Начинаем отправку, так как кворум достигнут

В этом коде мы задаем защелку с «кворумным значением». Как только инициализируется количество потоков, равное этому значению, можно приступать к обработке. Каждый поток будет вызывать срабатывание `countDown()`, как только завершит инициализацию. Поэтому основному потоку остается только дождаться, пока уровень кворума будет достигнут, и можно начинать работу (в том числе отправлять уведомления, хотя мы и опустили эту часть кода).

Далее мы поговорим об одном из самых полезных классов, имеющихся в арсенале разработчика многопоточных приложений: `ConcurrentHashMap` из библиотеки `java.util.concurrent`.

4.3.4. ConcurrentHashMap

Класс `ConcurrentHashMap` предоставляет параллельную версию стандартного `HashMap`. В новом варианте контейнера оптимизирован функционал `synchronizedMap()`, предоставляемый в классе `Collections`. Теперь методы из этого класса возвращают коллекции, обладающие избыточным количеством блокировок.

Как показано на рис. 4.7, классический `HashMap` использует функцию (хеш-функцию), чтобы определить, в каком сегменте памяти будет сохраняться пара ключ/значение. Именно отсюда берется «хешевая» часть имени класса. Напрашивается достаточно прямолинейное обобщение многопоточности: при внесении изменений требуется не блокировать всю структуру, а заблокировать лишь тот участок памяти, в который будут вноситься изменения.

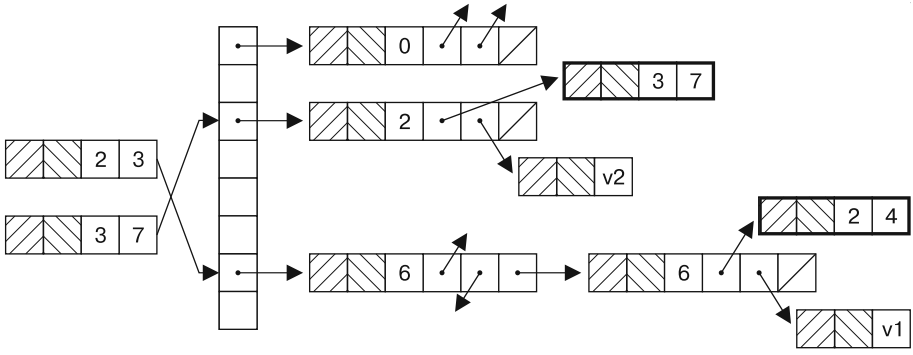


Рис. 4.7. Классический вид HashMap

СОВЕТ

Хорошо написанная реализация параллельного HashMap на самом деле не будет блокироваться при считывании. При записи будет блокироваться только соответствующий сегмент. В принципе, Java позволяет решать такие задачи, но есть еще низкоуровневые детали, которыми не любит заниматься большинство разработчиков.

Класс `ConcurrentHashMap` также реализует интерфейс `ConcurrentMap`, содержащий некоторые новые методы, обеспечивающие по-настоящему атомарную функциональность:

- `putIfAbsent()` — добавляет пару ключ/значение в `HashMap`, если такой ключ там пока отсутствует;
- `remove()` — атомарно удаляет пару ключ/значение, лишь если ключ присутствует, а значение соответствует текущему состоянию;
- `replace()` — API предоставляет различные формы этого метода для внесения атомарных изменений в `HashMap`.

В качестве примера попробуем заменить методы synchronized в листинге 4.1 обычным, несинхронизированным доступом в тех случаях, когда вы изменяете HashMap под названием arrivalTime так, чтобы этот контейнер одновременно являлся ConcurrentHashMap. Обратите внимание на отсутствие блокировок в листинге 4.8 — явная синхронизация здесь вообще не наблюдается.

Листинг 4.8. Использование ConcurrentHashMap

```
public class ExampleMicroBlogTimingNode implements SimpleMicroBlogNode {
...
private final Map<Update, Long> arrivalTime =
➥ new ConcurrentHashMap <>();
  ...
  public void propagateUpdate(Update upd_) {
    arrivalTime.putIfAbsent(upd_, System.currentTimeMillis());
  }
  public boolean confirmUpdateReceived(Update upd_) {
    return arrivalTime.get(upd_) != null;
  }
}
```

ConcurrentHashMap — один из наиболее полезных классов в библиотеке java.util.concurrent. Он обеспечивает дополнительную безопасность при многопоточности и более высокую производительность, а при обычном использовании не имеет серьезных недостатков. Его аналог для работы с List называется CopyOnWriteArrayList, о нем мы поговорим далее.

4.3.5. CopyOnWriteArrayList

Как понятно из названия, класс CopyOnWriteArrayList заменяет стандартный класс ArrayList. CopyOnWriteArrayList был создан потокобезопасным путем с добавлением семантики копирования при записи. Это значит, что любые операции, изменяющие список, создают новую копию массива, копируя список (как показано на рис. 4.8). Это также означает, что любые сформированные итераторы могут «не беспокоиться» о любых модификациях, которых они «не ожидали».

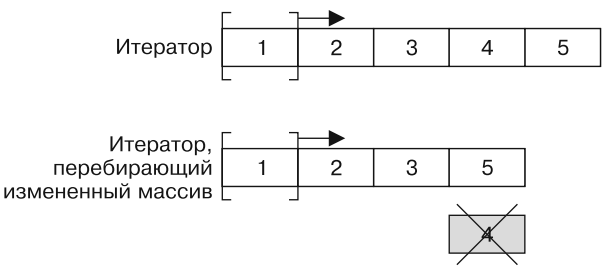


Рис. 4.8. Массив Copy-on-write

Такой способ обращения с совместно используемыми данными идеален, если важно получить быстрый и согласованный мгновенный снимок информации (кстати, такой список может отличаться для конкретных считывателей), несмотря на

значительный спад производительности и небезупречную синхронизацию. Подобная ситуация часто складывается с данными, которые не являются критически важными.

Рассмотрим, как копирование при записи работает на практике. Допустим, у нас есть лента сообщений микроблогов. Это классический пример данных, не являющихся критически важными. Кроме того, при работе с такими данными работоспособный и самодостаточный их снимок для каждого читателя обычно более важен, чем полная глобальная согласованность всех данных. В листинге 4.9 показан класс-контейнер, демонстрирующий ленту сообщений в таком виде, в каком ее просматривает отдельный пользователь. (Далее мы воспользуемся этим классом в листинге 4.10, чтобы подробно продемонстрировать, как осуществляется копирование при записи.)

Листинг 4.9. Пример копирования при записи

```
public class MicroBlogTimeline {
  private final CopyOnWriteArrayList<Update> updates;
  private final ReentrantLock lock;
  private final String name;
  private Iterator<Update> it;

  public void addUpdate(Update update_) {
    updates.add(update_);
  }

  public void prep() {
    it = updates.iterator();
  }

  public void printTimeline() {
    lock.lock();
    try {
      if (it != null) {
        System.out.print(name+ ": ");
        while (it.hasNext()) {
          Update s = it.next();
          System.out.print(s+ ", ");
        }
        System.out.println();
      }
    } finally {
      lock.unlock();
    }
  }
}
```

Конструктор опущен

Задаем итератор

Здесь нужна блокировка

Этот класс специально разработан для того, чтобы проиллюстрировать поведение `Iterator` при использовании семантики копирования при записи. Необходимо обеспечить блокировку в методе печати, чтобы предотвратить смешивание вывода между двумя потоками и чтобы можно было просматривать отдельное состояние двух потоков.

Можно вызвать класс `MicroBlogTimeline` из кода, показанного ниже.

Листинг 4.10. Демонстрация поведения, действующего во время копирования при записи

```
final CountDownLatch firstLatch = new CountDownLatch(1);
final CountDownLatch secondLatch = new CountDownLatch(1);
final Update.Builder ub = new Update.Builder();

final List<Update> l = new CopyOnWriteArrayList<>();
l.add(ub.author(new Author("Ben")).updateText("I like pie").build());
l.add(ub.author(new Author("Charles")).updateText(
➥ "I like ham on rye").build());

ReentrantLock lock = new ReentrantLock();
final MicroBlogTimeline tl1 = new MicroBlogTimeline("TL1", l, lock);
final MicroBlogTimeline tl22 = new MicroBlogTimeline("TL2", l, lock);

Thread t1 = new Thread() {
  public void run() {
    l.add(ub.author(new Author("Jeffrey")).updateText(
    ➥ "I like a lot of things").build());
    tl1.prep();
    firstLatch.countDown();
    try { secondLatch.await(); }
    ➥ catch (InterruptedException e) { }
    tl1.printTimeline();
  }
};

Thread t2 = new Thread(){
  public void run(){
    try {
      firstLatch.await();
      l.add(ub.author(new Author("Gavin")).updateText(
      ➥ "I like otters").build());
      tl2.prep();
      secondLatch.countDown();
    } catch (InterruptedException e) { }
    tl2.printTimeline();
  }
};
t1.start();
t2.start();
```

Этот листинг изобилует примерами скаффолдинга (вспомогательных конструкций) — к сожалению, этого сложно избежать. В приведенном листинге нужно обратить внимание сразу на несколько моментов.

- `CountDownLatch` используется для обеспечения строгого контроля над тем, что происходит между двумя потоками.

- Если заменить `CopyOnWriteArrayList` на обычный `List (1)`, то в результате получим исключение `ConcurrentModificationException`.
- Здесь также есть пример объекта `Lock`, совместно используемого двумя потоками для управления доступа к разделяемому ресурсу (в данном случае `STDOUT`). В блочно-структурированном виде этот код был бы гораздо более запутанным.

Вывод данного кода выглядит так:

```
TL2: Update [author=Author [name=Ben], updateText=I like pie, createTime=0],
     Update [author=Author [name=Charles], updateText=I like ham on rye,
     createTime=0], Update [author=Author [name=Jeffrey], updateText=I like a
     lot of things, createTime=0], Update [author=Author [name=Gavin],
     updateText=I like otters, createTime=0],

TL1: Update [author=Author [name=Ben], updateText=I like pie, createTime=0],
     Update [author=Author [name=Charles], updateText=I like ham on rye,
     createTime=0], Update [author=Author [name=Jeffrey], updateText=I like a
     lot of things, createTime=0],
```

Как видите, во второй строке вывода, помеченной как `TL1`, отсутствует последнее уведомление, в котором упоминаются выдры (otters). И это несмотря на тот факт, что доступ к `mbex1` произошел уже после того, как список был изменен. Здесь мы видим, что `Iterator`, содержащийся в `mbex1`, был скопирован в `mbex2` и добавление последнего обновления произошло незаметно для `mbex1`. Это и есть свойство копирования при записи, которое мы хотим обеспечить для этих объектов.

ПРОИЗВОДИТЕЛЬНОСТЬ COPYONWRITEARRAYLIST

Работа с классом `CopyOnWriteArrayList` требует более тщательного продумывания, чем использование `ConcurrentHashMap`, — последний, в сущности, является ситуативной конкурентной заменой для `HashMap`. Дело в проблемах с производительностью, возникающих при использовании копирования при записи. В таком случае, если список изменяется в процессе считывания или обхода, должен быть скопирован весь массив.

Это означает, что если вам часто приходится вносить изменения в список, количество таких обращений сравнимо с количеством обращений для считывания, а подобный подход не гарантирует увеличения производительности. Но, как мы неоднократно указываем в главе 6, единственный способ получить надежный и производительный код — тестировать, снова тестировать и измерять результаты.

Следующий крупный общий компонент параллельного кода в `java.util.concurrent` — это сущность `Queue` (очередь). Очередь используется для распределения рабочих элементов между потоками и может служить основой для разнообразных гибких и надежных вариантов многопоточного дизайна.

4.3.6. Очереди

Очередь — чудесная абстракция (поверьте, мы так считаем не только потому, что живем в Лондоне, всемирной столице очередей). Очередь обеспечивает простой и надежный способ распределения вычислительных ресурсов между рабочими единицами (либо, с другой стороны, присвоения рабочих единиц вычислительным ресурсам, если так понятнее).

Существует несколько паттернов многопоточного программирования на Java, которые в значительной степени опираются на потокобезопасные реализации `Queue`. Поэтому очень важно хорошо разбираться в очередях. Базовый интерфейс `Queue` расположен в `java.util`, поскольку данный паттерн может быть очень важен и при однопоточном программировании. Но мы сосредоточимся именно на многопоточных практических ситуациях. Предполагаем, что вы уже сталкивались с очередями на практике.

Один из самых распространенных случаев, на котором мы и собираемся сосредоточиться, — использование очереди для передачи рабочих единиц между потоками. Такой паттерн часто идеально подходит для простейшего параллельного расширения `Queue` — `BlockingQueue`.

BlockingQueues

`BlockingQueue` — это очередь, имеющая два дополнительных особых свойства.

- При попытке выполнить с очередью действие `put()` поток, ставящий элемент в очередь, должен будет подождать, пока не освободится место, если очередь уже заполнена.
- При попытке выполнить с очередью действие `take()` забирающий поток должен будет блокироваться, если очередь пуста.

Два этих свойства очень полезны, поскольку если один поток (или пул потоков) при работе слишком вырывается вперед — так, что другому потоку или пулу за ним не угнаться, — то более быстрому потоку приходится подождать. Так регулируется вся система. Эта ситуация проиллюстрирована на рис. 4.9.

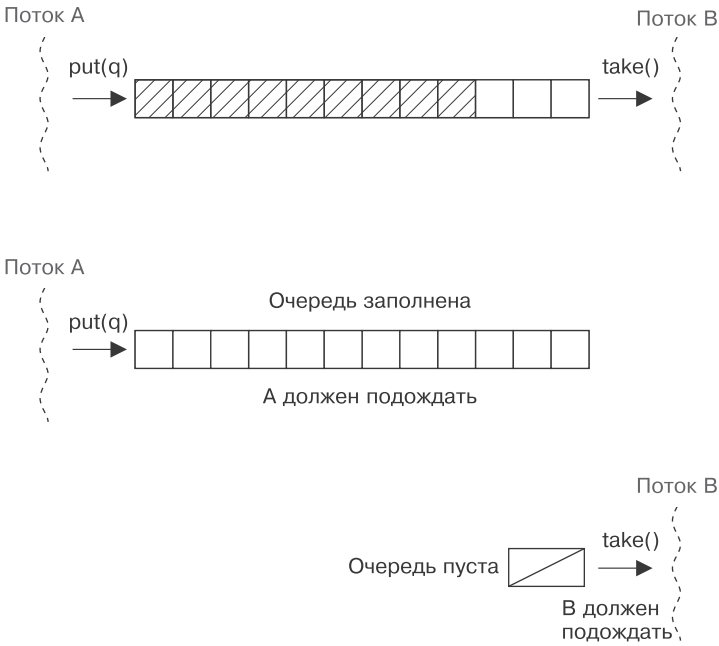


Рис. 4.9. BlockingQueue

ДВЕ РЕАЛИЗАЦИИ BLOCKINGQUEUE

В языке Java присутствуют две простейшие реализации интерфейса `BlockingQueue: LinkedBlockingQueue` и `ArrayBlockingQueue`. Их свойства немного различаются. Например, реализация массива очень эффективна, если известен точный предел, которого может достигать размер очереди. Первая, ссылочная реализация в некоторых ситуациях может работать несколько быстрее.

Работа с WorkUnit

Все интерфейсы `Queue` являются обобщенными — это `Queue<E>`, `BlockingQueue<E>` и т. д. Хотя это и может показаться странным, но иногда бывает целесообразно опираться на это свойство и создавать искусственные контейнеры, в которые обертываются рабочие элементы.

Допустим, у вас есть класс `MyAwesomeClass`, представляющий рабочие единицы, которые вы хотите обработать многопоточным образом. В таком случае, чтобы не делать вот так:

```
BlockingQueue<MyAwesomeClass>
```

лучше поступить так:

```
BlockingQueue<WorkUnit<MyAwesomeClass>>
```

Здесь `WorkUnit` (или `QueueObject`, смотря как вы хотите назвать класс-контейнер) — это интерфейс или класс, который играет роль обертки и может выглядеть примерно так:

```
public class WorkUnit<T> {
  private final T workUnit;

  public T getWork(){ return workUnit; }

  public WorkUnit(T workUnit_) {
    workUnit = workUnit_;
  }
}
```

Мы поступаем так потому, что на данном уровне косвенности появляется возможность добавить новые метаданные, не нарушая при этом концептуальную целостность содержащегося типа (в данном случае `MyAwesomeClass`).

Оказывается, это удивительно полезно. Не счесть практических ситуаций, в которых пригодятся дополнительные метаданные. Вот несколько примеров:

- тестирование (например, отображение истории изменений для объекта);
- индикаторы производительности (допустим, время прибытия или качество обслуживания);
- информация о системе времени исполнения (например, как именно была выполнена маршрутизация `MyAwesomeClass`).

Добавить такую косвенность постфактум бывает довольно сложно. Если вы обнаружите, что в определенных обстоятельствах нужны дополнительные метаданные, то, возможно, для решения проблемы потребуется крупный рефакторинг. А ведь аналогичного результата можно достичь, внеся простейшие изменения в классе `WorkUnit`.

Пример BlockingQueue

Рассмотрим BlockingQueue в действии на простом примере: домашние питомцы ожидают приема у ветеринара. В примере представлена коллекция зверушек, которых можно встретить в ветпункте (листинг 4.1).

Листинг 4.11. Моделирование домашних питомцев на языке Java

```
public abstract class Pet {
  protected final String name;

  public Pet(String name) {
    this.name = name;
  }
  public abstract void examine();
}

public class Cat extends Pet {
  public Cat(String name) {
    super(name);
  }
  public void examine(){
    System.out.println("Meow!");
  }
}

public class Dog extends Pet {
  public Dog(String name) {
    super(name);
  }
  public void examine(){
    System.out.println("Woof!");
  }
}

public class Appointment<T> {
  private final T toBeSeen;
  public T getPatient(){ return toBeSeen; }
  public Appointment(T incoming) {
    toBeSeen = incoming;
  }
}
```

Этот простой пример показывает, как можно смоделировать очередь на прием к ветеринару с помощью LinkedBlockingQueue<Appointment<Pet>>. Здесь класс Appointment играет роль WorkUnit.

Объект ветеринара создается с очередью (в которой будут делаться назначения, за них отвечает объект, моделирующий администратора из приемной) и со временем паузы. Пауза — это длительность перерывов для ветеринара между приемами пациентов.

Можно смоделировать ветеринара так, как показано в листинге 4.12. Пока поток работает, он многократно вызывает `seePatient()` в бесконечном цикле. Разумеется, в реальном мире такая ситуация невозможна, поскольку ветеринар захочет уходить с работы по вечерам и по выходным, а не торчать в кабинете дни и ночи напролет, как доктор Айболит.

Листинг 4.12. Моделирование ветеринара

```
public class Veterinarian extends Thread {
  protected final BlockingQueue<Appointment<Pet>> appts;
  protected String text = "";
  protected final int restTime;
  private boolean shutdown = false;

  public Veterinarian(BlockingQueue<Appointment<Pet>> lbq, int pause) {
    appts = lbq;
    restTime = pause;
  }

  public synchronized void shutdown(){
    shutdown = true;
  }

  @Override
  public void run(){
    while (!shutdown) {
      seePatient();
      try {
        Thread.sleep(restTime);
      } catch (InterruptedException e) {
        shutdown = true;
      }
    }
  }

  public void seePatient() {
    try {
      Appointment<Pet> ap = appts.take();   <-- Блокировка приема
      Pet patient = ap.getPatient();
      patient.examine();
    } catch (InterruptedException e) {
      shutdown = true;
    }
  }
}
```

В методе `seePatient()` поток будет извлекать назначения из очереди и по очереди проверять зверюшек, соответствующих каждому назначению. Если в настоящий момент в очереди не окажется назначений, то поток будет блокирован.

Филигранный контроль BlockingQueue

Кроме простых API `take()` и `offer()`, `BlockingQueue` предлагает и другой способ взаимодействия с очередью. Такой способ обеспечивает еще более полный контроль над процессом за счет небольшого усложнения кода. Речь идет о возможности помещения элемента в очередь и забора элемента из очереди с задержкой. Таким образом, поток, столкнувшийся с проблемами, может отказаться от взаимодействия с очередью и выполнить какую-то другую задачу.

На практике такой вариант используется нечасто, но иногда бывает очень полезен, поэтому продемонстрируем и его для полноты картины. Рассмотрим следующий пример из нашего сценария, описывающего работу с микроблогами (листинг 4.13).

Листинг 4.13. Пример поведения BlockingQueue

```
public abstract class MicroBlogExampleThread extends Thread {
  protected final BlockingQueue<Update> updates;
  protected String text = "";
  protected final int pauseTime;
  private boolean shutdown = false;

  public MicroBlogExampleThread(BlockingQueue<Update> lbq_, int pause_) {
    updates = lbq_;
    pauseTime = pause_;
  }

  public synchronized void shutdown(){
    shutdown = true;
  }

  @Override
  public void run(){
    while (!shutdown) {
      doAction();
      try {
        Thread.sleep(pauseTime);
      } catch (InterruptedException e) {
        shutdown = true;
      }
    }
  }
  public abstract void doAction();
}

final Update.Builder ub = new Update.Builder();
final BlockingQueue<Update> lbq = new LinkedBlockingQueue<>(100);
MicroBlogExampleThread t1 = new MicroBlogExampleThread(lbq, 10) {
  public void doAction(){
    text = text + "X";
```

Обеспечиваем чистое завершение работы потока

Приказываем подклассу реализовать действие

```
    Update u = ub.author(new Author("Tallulah")).updateText(text).build();
    boolean handed = false;
    try {
      handed = updates.offer(u, 100, TimeUnit.MILLISECONDS);
    } catch (InterruptedException e) {
    }
    if (!handed) System.out.println(
    ➥ "Unable to hand off Update to Queue due to timeout");
  }
};
MicroBlogExampleThread t2 = new MicroBlogExampleThread(lbq, 1000) {
  public void doAction(){
    Update u = null;
    try {
      u = updates.take();
    } catch (InterruptedException e) {
      return;
    }
  }
};
t1.start();
t2.start();
```

Запустив пример в такой форме, вы увидите, как быстро заполнится очередь. Это означает, что предлагающий поток обгоняет принимающий поток. Очень скоро начнет появляться сообщение Unable to hand off Update to Queue due to timeout (Невозможно записать обновление в очередь из-за задержки).

Здесь мы сталкиваемся с одной из крайностей модели «связанного пула потоков». В данном случае восходящий пул потоков работает быстрее нисходящего. Такая ситуация бывает проблематичной. В частности, она приводит к переполнению `LinkedBlockingQueue`. В противном случае, если количество потребителей превышает количество производителей, очередь может опустошиться. К счастью, в Java 7 есть новый вариант `BlockingQueue`, выручающий в такой ситуации, — `TransferQueue`.

TransferQueue — новинка в Java 7

В Java 7 появилась очередь `TransferQueue`. В принципе, это очередь `BlockingQueue`, способная выполнять дополнительную операцию — `transfer()`. Такая операция немедленно передает рабочий элемент потоку-получателю, если он находится в состоянии ожидания. В противном случае очередь блокируется до тех пор, пока не появится поток, способный принять элемент. Такую систему можно сравнить с «заказной корреспонденцией» — поток, обрабатывавший элемент, не приступит к обработке следующего элемента, пока не отдаст уже имеющийся. Соответственно, система может регулировать скорость, с которой входящий пул потоков принимается за выполнение новых задач.

Регулировать эту скорость можно было бы и с помощью блокирующей очереди фиксированного размера, но `TransferQueue` имеет более гибкий интерфейс. Кроме того, производительность вашего кода может повыситься после замены `BlockingQueue`

на TransferQueue. Дело в том, что реализация TransferQueue была написана с учетом возможностей современных компиляторов и процессоров, поэтому она работает очень эффективно. Но если уж говорить о производительности, то ее потенциальный рост необходимо измерить и убедиться, что он существует, а не просто предполагать. Учитывайте также, что в составе Java 7 есть только одна версия TransferQueue — ссылочная (linked).

В следующем примере кода мы рассмотрим, насколько просто заменить очередь BlockingQueue очередью TransferQueue. Небольшие изменения, показанные в листинге 4.13, совершенствуют код до реализации TransferQueue (листинг 4.14).

Листинг 4.14. Замена BlockingQueue на TransferQueue

```
public abstract class MicroBlogExampleThread extends Thread {
  protected final TransferQueue<Update> updates;
  ...

  public MicroBlogExampleThread(TransferQueue<Update> lbq_, int pause_) {
    updates = lbq_;
    pauseTime = pause_;
  }
  ...
}

final TransferQueue<Update> lbq = new LinkedTransferQueue<Update>(100);

MicroBlogExampleThread t1 = new MicroBlogExampleThread(lbq, 10) {
  public void doAction(){
    ...
    try {
      handed = updates.tryTransfer(u, 100, TimeUnit.MILLISECONDS);
    } catch (InterruptedException e) {
    }
    ...
  }
};
```

На этом мы завершаем обзор основных компонентов, предоставляющих нам сырье для создания надежных многопоточных приложений. Далее нужно скомбинировать эти компоненты с движками, позволяющими эксплуатировать параллельный код, — их еще называют фреймворками-исполнителями (executor framework). Они позволяют планировать и контролировать выполнение задач. Вы, в свою очередь, можете собирать с их помощью эффективные параллельные потоки задач для обращения с рабочими элементами и выстраивать крупные многопоточные приложения.

4.4. Контроль исполнения

В этой главе мы уже некоторое время обсуждаем фрагменты работы как абстрактные единицы. Но здесь есть одна тонкость. Мы пока не говорили о том, что все упомянутые выше единицы мельче, чем Thread (поток). Они позволяют выполнять

вычисления, содержащиеся во фрагменте работы, не создавая отдельного нового потока для каждого фрагмента. Часто такой способ работы с многопоточным кодом наиболее эффективен, поскольку в этом случае ресурсы на запуск потока `Thread` не приходится тратить при операции с каждой единицей. Вместо этого потоки, которые фактически занимаются выполнением кода, переиспользуются. По окончании обработки одной задачи поток возвращается за новым фрагментом работы.

За счет небольшого усложнения кода вы можете обращаться к таким абстракциям как к пулам потоков, паттернам рабочих и управляющих потоков и исполнителям — это одни из самых многофункциональных паттернов в словаре разработчика. `Callable`, `Future` и `FutureTask` — отвечающие за моделирование задач классы и интерфейсы, которыми мы займемся наиболее плотно. Мы также поговорим о классах-исполнителях, в частности о `ScheduledThreadPoolExecutor`.

4.4.1. Моделирование задач

Наша конечная цель — иметь задачи (рабочие единицы), которые можно планировать, не порождая отдельного потока для каждой из них. В конечном итоге это означает, что они должны моделироваться как код, который можно вызывать (обычно это делает исполнитель), а не как непосредственно запускаемые потоки.

Рассмотрим три различных способа моделирования задач — с помощью интерфейсов `Callable` и `Future` и с помощью класса `FutureTask`.

Интерфейс Callable

Интерфейс `Callable` представляет собой очень распространенную абстракцию. Это фрагмент кода, который можно вызвать и который возвращает результат. Несмотря на то что идея этой конструкции достаточно прямолинейна, сама концепция довольно тонкая и одновременно мощная. Она позволяет получить исключительно полезные паттерны.

Типичный пример использования `Callable` — анонимная реализация. В последней строке следующего фрагмента `s` устанавливается в значение `out.toString()`:

```
final MyObject out = getSampleObject();

Callable<String> cb = new Callable<String>() {
  public String call() throws Exception {
    return out.toString();
  }
};
String s = cb.call();
```

Можно считать анонимную реализацию `Callable` отложенным вызовом единственного абстрактного метода `call()`, который обязательно должен предоставляться в этой реализации.

`Callable` — пример так называемого SAM-типа (SAM расшифровывается как «единственный абстрактный метод»). Именно здесь язык Java ближе всего подходит к реализации функций в виде типов первого класса. Мы подробнее поговорим о концепции функций как значений, они же — функции как типы первого класса, в последующих главах, где столкнемся с ними при изучении других языков.

Интерфейс Future

Интерфейс Future используется для представления асинхронной задачи. Это означает, что он ожидает результата выполнения задачи, которая пока может быть еще не завершена. Мы кратко коснулись таких интерфейсов в главе 2, где говорили о NIO.2 и асинхронном вводе-выводе.

Вот основные методы, применяемые с Future:

- get() — получает результат. Если результата пока нет, то get() блокируется до тех пор, пока результат не появится. Есть также версия, принимающая задержку и поэтому не попадающая в вечную блокировку;
- cancel() — позволяет отменить вычисление до завершения;
- isDone() — позволяет вызывающей стороне определить, завершены ли вычисления.

В следующем фрагменте кода показано, как Future используется в программе для нахождения простых чисел.

```
Future<Long> fut = getNthPrime(1_000_000_000);

Long result = null;
while (result == null) {
  try {
    result = fut.get(60, TimeUnit.SECONDS);
  } catch (TimeoutException tox) { }
System.out.println("Still not found the billionth prime!");
}
System.out.println("Found it: "+ result.longValue());
```

Представьте, что в этом фрагменте кода getNthPrime() возвращает Future, исполняемый в каком-нибудь фоновом потоке (или даже в нескольких потоках). Эта задача может, например, решаться в одном из фреймворков-исполнителей, о которых мы поговорим в следующем подразделе. Даже на современном оборудовании такие вычисления могут длиться довольно долго — в итоге может понадобиться вызвать метод cancel() интерфейса Future.

Класс FutureTask

Класс FutureTask — это распространенная реализация интерфейса Future, также реализующая Runnable. Как вы увидите, это означает, что FutureTask можно отдавать исполнителям — и это самое важное. В принципе, этот интерфейс совмещает методы Future и Runnable: get(), cancel(), isDone(), isCancelled() и run(), хотя последний метод вызывается исполнителем, а не непосредственно вашим кодом.

Для работы с FutureTask также предоставляются два вспомогательных конструктора: принимающий Callable и принимающий Runnable. Связь между этими двумя классами подсказывает очень гибкий подход к решению задач. Задача может быть написана как Callable, а затем обернута в FutureTask, который уже можно назначить исполнителю (и при необходимости отменить), поскольку FutureTask по сути является Runnable.

4.4.2. ScheduledThreadPoolExecutor

`ScheduledThreadPoolExecutor` (STPE) — это основной из классов пулов потоков. Он универсален и поэтому очень широко применяется. STPE принимает работу в форме задач, которые направляет к пулу потоков:

- пулы потоков могут иметь заранее определенный либо адаптивный размер;
- задачи могут назначаться для периодического выполнения либо однократно;
- STPE дополняет класс `ThreadPoolExecutor` (который похож на STPE, но в нем отсутствует возможность периодического назначения).

Один из наиболее распространенных паттернов для многопоточных приложений среднего и крупного размера связан с применением пулов потоков STPE. При этом используются исполняющие потоки, соединенные вспомогательными классами из `java.util.concurrent`, о которых мы уже говорили (речь, в частности, о `ConcurrentHashMap`, `CopyOnWriteArrayList`, `BlockingQueue`).

STPE — это просто один из многих родственных исполнителей, которые можно с легкостью получать благодаря фабричным методам, предоставляемым в классе `Executors` из `java.util.concurrent`. Эти фабричные методы являются в основном вспомогательными; они облегчают разработчику доступ к типичной конфигурации, в то же время предоставляя полнофункциональный интерфейс.

В листинге 4.15 показан пример периодического считывания. Это обычный пример использования `newScheduledThreadPool()`: объект `msgReader` назначается для опроса (`poll()`) очереди, получает кусок работы от объекта `WorkUnit` и ставит его в очередь, а потом печатает эту информацию.

Листинг 4.15. Периодическое считывание с применением STPE

```
private ScheduledExecutorService stpe;
private ScheduledFuture<?> hndl;

private BlockingQueue<WorkUnit<String>> lbq = new LinkedBlockingQueue<>();

private void run(){
  stpe = Executors.newScheduledThreadPool(2);

  final Runnable msgReader = new Runnable(){
    public void run(){
      String nextMsg = lbq.poll().getWork();
      if (nextMsg != null) System.out.println("Msg recvd: "+ nextMsg);
    }
  };
  hndl = stpe.scheduleAtFixedRate(msgReader, 10, 10,
  ➥ TimeUnit.MILLISECONDS);
}

public void cancel() {
  final ScheduledFuture<?> myHndl = hndl;
```

Необходимо для отмены

Фабричный метод исполнителей

```
  stpe.schedule(new Runnable() {
    public void run() { myHndl.cancel(true); } <-- Необходимо для отмены
  }, 10, TimeUnit.MILLISECONDS);
}
```

В данном примере STPE пробуждает поток один раз в 10 мс и приказывает ему попытаться опросить очередь (`poll()`). Если при считывании возвращается `null` (поскольку очередь пуста), то больше ничего не происходит и поток возвращается в спящее состояние. Если рабочая единица была получена, то поток печатает ее содержимое.

ПРОБЛЕМЫ, ВОЗНИКАЮЩИЕ ПРИ ВЫПОЛНЕНИИ С ПОМОЩЬЮ CALLABLE

Существует множество проблем при работе с простыми формами `Callable`, `FutureTask` и родственными им сущностями. В частности, необходимо отметить, что система типов осложняет такую работу.

Рассмотрим случай, в котором мы попытаемся учесть все возможные сигнатуры, которые мог бы иметь неизвестный метод. `Callable` предоставляет лишь модель методов, не принимающих аргументы. Для того чтобы учесть все возможности, вам понадобится множество различных вариантов `Callable`.

В Java эту проблему можно обойти, если специально указывать, какие сигнатуры методов могут существовать в моделируемых вами системах. Но, как будет показано в третьей части этой книги, динамические языки не разделяют такого статического взгляда на мир. Подобное несовпадение между системами типов — крупная проблема, к обсуждению которой мы еще вернемся. Пока просто отметим, что `Callable` при всей полезности выдвигает довольно много ограничений при создании общего фреймворка для выполнения моделирования.

Теперь перейдем к одному из самых замечательных фрагментов Java 7 — фреймворку fork/join (ветвление/слияние). Он предназначен для обеспечения легковесного параллелизма. Новый фреймворк позволяет решать широкий набор проблем еще более эффективно, чем это удается делать с помощью исполнителей, рассмотренных в этом разделе (а это, поверьте, нелегко).

4.5. Фреймворк fork/join (ветвление/слияние)

В главе 6 мы расскажем о том, насколько увеличилась скорость процессоров (или, точнее, количество транзисторов на микросхеме) в последние годы. Побочный эффект такого ускорения заключается в том, что теперь ожидание ввода-вывода стало довольно распространенной ситуацией. И это означает, что мы могли бы более рационально использовать вычислительные мощности наших компьютеров. Фреймворк fork/join призван достичь именно этого. Данные наработки также связаны с самыми крупными нововведениями в области параллельной обработки в Java 7.

Вся суть fork/join заключается в автоматическом назначении задач для пула потоков, причем этот процесс протекает незаметно для пользователя. Чтобы этого добиться, задачи должны легко члениться на составляющие таким образом, как

укажет пользователь. Во многих прикладных ситуациях fork/join различает большие и малые задачи, что очень органично вписывается в данный фреймворк.

Кратко рассмотрим некоторые основные факты и фундаментальные моменты, связанные с fork/join.

- Во фреймворке fork/join появляется новый тип исполняющей службы, которая называется `ForkJoinPool`.
- Служба `ForkJoinPool` обрабатывает единицу параллелизма (`ForkJoinTask`), которая меньше по размеру, чем `Thread`.
- Обычно в fork/join используются задачи двух видов (правда, оба вида чаще всего представлены как экземпляры `ForkJoinTask`):
  - малыми называются такие задачи, которые можно выполнить сразу, не потребляя значительного количества процессорного времени;
  - большими называются такие задачи, которые требуется разбивать на части (иногда на довольно большое количество частей), прежде чем можно будет перейти к их выполнению.
- Фреймворк предоставляет базовые методы для поддержки разбиения больших задач, а также оснащен механизмом автоматического назначения и переназначения задач.

Одна из основных черт этого фреймворка — он рассчитан на то, чтобы такие легковесные задачи без особых проблем порождали другие экземпляры `ForkJoinTask`. Эти новые экземпляры будут исполняться тем же пулом потоков, который исполнял их родителя. Такой паттерн иногда называется «разделяй и властвуй».

Начнем с простого примера использования фреймворка fork/join, затем затронем возможность, называемую «захватом работы» (work stealing). В конце концов, поговорим о свойствах тех проблем, которые хорошо подходят для параллельной обработки. Познакомиться с fork/join удобнее всего на примере.

4.5.1. Простой пример fork/join

В качестве простого примера, иллюстрирующего возможности fork/join, рассмотрим следующий случай. Допустим, у нас есть массив обновлений для сервиса микроблогов и эти обновления могут приходить в разное время. Мы хотим отсортировать их по времени прибытия, чтобы сгенерировать ленты сообщений для пользователей. Такую ленту мы создали в листинге 4.9.

Чтобы достичь этого, применим многопоточную сортировку, которая является вариантом `MergeSort`. В листинге 4.16 использован специализированный подкласс `ForkJoinTask — RecursiveAction`. Он проще обычного `ForkJoinTask`, так как явно не предусматривает получения какого-либо общего результата (обновления переупорядочиваются на месте) и акцентирует рекурсивную природу задач.

Класс `MicroBlogUpdateSorter` предоставляет способ упорядочения списка обновлений путем применения метода `compareTo()` к объектам `Update`. Метод `compute()` (который вам придется реализовать, так как в суперклассе `RecursiveAction` он является абстрактным) просто упорядочивает массив обновлений микроблога по времени создания обновления.

Листинг 4.16. Сортировка с применением RecursiveAction

```
public class MicroBlogUpdateSorter extends RecursiveAction {
  private static final int SMALL_ENOUGH = 32;
  private final Update[] updates;
  private final int start, end;
  private final Update[] result;
  public MicroBlogUpdateSorter(Update[] updates_) {
    this(updates_, 0, updates_.length);
  }
  public MicroBlogUpdateSorter(Update[] upds_,
  ➥ int startPos_, int endPos_) {
    start = startPos_;
    end = endPos_;
    updates = upds_;
    result = new Update[updates.length];
  }
  private void merge(MicroBlogUpdateSorter left_,
  ➥ MicroBlogUpdateSorter right_) {
    int i = 0;
    int lCt = 0;
    int rCt = 0;
    while (lCt < left_.size() && rCt < right_.size()) {
      result[i++] = (left_.result[lCt].compareTo(right_.result[rCt]) < 0)
      ? left_.result[lCt++]
      : right_.result[rCt++];
    }
    while (lCt < left_.size()) result[i++] = left_.result[lCt++];
    while (rCt < right_.size()) result[i++] = right_.result[rCt++];
  }
  public int size() {
    return end - start;
  }
  public Update[] getResult() {
    return result;
  }
  @Override
  protected void compute() {
    if (size() < SMALL_ENOUGH) {
      System.arraycopy(updates, start, result, 0, size());
      Arrays.sort(result, 0, size());
    } else {
      int mid = size() / 2;
      MicroBlogUpdateSorter left = new MicroBlogUpdateSorter(
      ➥ updates, start, start + mid);
      MicroBlogUpdateSorter right = new MicroBlogUpdateSorter(
      ➥ updates, start + mid, end);
      invokeAll(left, right);
```

32 элемента или менее, серийная сортировка

Метод RecursiveAction

```
      merge(left, right)
    }
  }
}
```

Чтобы задействовать сортировщик, его можно запустить с помощью кода наподобие того, что показан ниже. Этот код сгенерирует несколько обновлений (каждое состоит из строки иксов), перемешает их, а потом передаст сортировщику (листинг 4.17). В качестве вывода получим переупорядоченные обновления.

Листинг 4.17. Использование рекурсивного сортировщика

```
List<Update> lu = new ArrayList<Update>();
String text = "";
final Update.Builder ub = new Update.Builder();
final Author a = new Author("Tallulah");

for (int i=0; i<256; i++) {
  text = text + "X";
  long now = System.currentTimeMillis();
  lu.add(ub.author(a).updateText(text).createTime(now).build());
  try {
    Thread.sleep(1);
  } catch (InterruptedException e) {
  }
}
Collections.shuffle(lu);
Update[] updates = lu.toArray(new Update[0]);

MicroBlogUpdateSorter sorter = new MicroBlogUpdateSorter(updates);
ForkJoinPool pool = new ForkJoinPool(4);
pool.invoke(sorter);

for (Update u: sorter.getResult()) {
  System.out.println(u);
}
```

Передаем массив нулевого размера, сохраняем выделение

TIMSORT

С появлением Java 7 применяемый по умолчанию алгоритм для сортировки массивов изменился. Ранее он имел форму QuickSort, но в Java 7 на смену QuickSort пришел TimSort — вариант MergeSort, являющийся гибридом обычной сортировки и сортировки методом вставок. Изначально TimSort был разработан Тимом Петерсом (Tim Peters) для языка Python. В Python этот алгоритм сортировки используется по умолчанию, начиная с версии 2.3 (2002).

Хотите сами убедиться в том, что TimSort присутствует в Java 7? Просто передайте массив `null` объектов `Update` в листинг 4.16. Операции сравнения в сортировочной процедуре массива `Arrays.sort()` будут срываться из-за исключения, происходящего в результате обращения по нулевому адресу. В стектрейсе вы заметите классы TimSort.

4.5.2. ForkJoinTask и захват работы

`ForkJoinTask` — суперкласс для `RecursiveAction`. Это обобщенный класс возвращаемого типа `action` (действие). Соответственно, `RecursiveAction` дополняет `ForkJoinTask<Void>`. Таким образом, `ForkJoinTask` очень удобен при работе по принципу map-reduce, когда возвращаемое значение представляет собой «сухой остаток» от множества данных.

Задачи `ForkJoinTask` направляются к классу `ForkJoinPool`, представляющему собой службу-исполнитель нового типа, разработанную специально для решения таких легковесных задач. Служба поддерживает список задач для каждого потока. Как только одна из задач завершится, служба может перебросить часть задач от полностью загруженного потока к бездействующему.

Причина применения такого алгоритма «захвата работы» заключается в том, что без него могут возникать проблемы диспетчеризации, так как задачи бывают большими и малыми. В принципе, задачи двух разных размеров требуют на выполнение абсолютно разные отрезки времени. Например, очередь выполнения у одного потока может состоять только из маленьких задач, а у другого потока — только из больших. Если маленькая задача выполняется в пять раз быстрее, чем большая, то поток, нагруженный лишь маленькими задачами, вполне может остаться без дела до того, как закончит работу поток, нагруженный большими задачами.

Захват работы был реализован именно для устранения такой проблемы. Он должен обеспечить использование всех потоков из пула на протяжении всего жизненного цикла задачи fork/join. Процесс полностью автоматический, вам не нужно предпринимать никаких специальных действий для пользования преимуществами захвата работы. Перед нами еще один пример того, как среда времени исполнения помогает разработчику управлять параллелизмом, избавляя программиста от необходимости решать эту задачу полностью вручную.

4.5.3. Параллелизация проблем

Перспективы использования fork/join выглядят соблазнительно, но на практике не всякая проблема легко сводится к такой простой форме, как многопоточный сортировщик MergeSort из подраздела 4.5.1.

Вот несколько примеров проблем, которые удобно решать методом fork/join:

- моделирование движения множества простых объектов (например, эффекты частиц);
- анализ файлов журналов;
- операции над данными, где количество вычисляется на основании ввода (уже упоминавшиеся выше операции map-reduce).

Иными словами, fork/join хорошо подходит для решения проблем, которые могут члениться на составные части, как это показано на рис. 4.10.

Для того чтобы на практике определить, насколько легко редуцируется конкретная проблема, требуется проверить ее и ее подзадачи на соответствие пунктам следующего списка.

- Могут ли подзадачи, образующие проблему, выполняться без явного взаимодействия и синхронизации друг с другом?

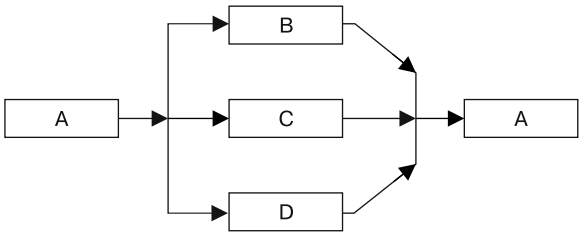


Рис. 4.10. Ветвление и слияние

- Сводятся ли подзадачи к вычислению каких-либо значений из предоставленных данных, но без изменения этих данных (то есть можно ли назвать их «чистыми функциями» с точки зрения функционального программирования)?
- Хорошо ли подходит принцип «разделяй и властвуй» для решения таких подзадач? Возникает ли в итоге выполнения одной подзадачи еще несколько подзадач (которые, в свою очередь, могут быть более тонкими, чем породившая их задача)?

Если вы ответили на предыдущие вопросы «Да!» или «В основном да, но есть некоторые пограничные случаи», то проблема, вероятно, будет хорошо решаться с применением fork/join. Если же, напротив, вы ответили на эти вопросы «Может быть» или «Скорее нет», то fork/join в вашем случае, вероятно, будет работать плохо и лучше попробовать другой способ синхронизации.

ПРИМЕЧАНИЕ

Приведенный выше список удобен для проверки того, насколько хорошо проблема может быть обработана с помощью fork/join (подобные проблемы часто встречаются при работе с базами данных, например, Hadoop или NoSQL).

Создавать хорошие многопоточные алгоритмы сложно, и подход fork/join уместен не в любой ситуации. Он очень полезен в своей области применения, но в конечном итоге именно вам придется решать, подходит ли такой фреймворк для устранения вашей проблемы. Если не подходит — будьте готовы к разработке собственного решения, которое можно построить на базе превосходного инструментария `java.util.concurrent`.

В следующем разделе мы обсудим детали модели памяти Java (JMM), которые зачастую понимаются неправильно. Многие программисты, работающие с Java, знакомы с JMM и пишут код в соответствии с тем, как сами понимают эту модель, без всякого систематического ее изучения. Если вы именно такой специалист, то изложенный ниже новый материал будет строиться на вашем интуитивном понимании этой модели. Вы основательно уясните проблему, что станет базисом для уже имеющегося интуитивного понимания. JMM — довольно сложная тема, так что можете пропустить ее, если вам не терпится перейти к следующей главе.

4.6. Модель памяти языка Java (JMM)

Модель памяти Java (JMM) представлена в разделе 17.4 спецификации языка Java (JLS). Это довольно сухая часть спецификации. Она описывает JMM с точки зрения операций синхронизации и в контексте математического конструкта, называемого

частичным порядком (partial order). Этот раздел великолепен с точки зрения языковых теоретиков и воплотителей спецификации языка Java (авторов компиляторов и виртуальной машины), но гораздо менее интересен для практикующих разработчиков, которым требуется детально понимать, как именно будет выполняться их многопоточный код.

Не будем повторять здесь теоретические детали, а просто перечислим важнейшие правила, сформулировав их в виде двух базовых концепций. Это отношения между блоками кода, которые мы назовем *Synchronizes-With* (синхронизируется с) и *Happens-Before* (происходит до).

- Happens-Before — указывает, что один блок кода должен полностью завершиться, прежде чем выполнение другого блока кода сможет начаться.
- Synchronizes-With — означает, что действие будет синхронизировать свое представление об объекте с представлением об объекте в основной памяти и лишь потом сможет продолжить работу.

Если вы изучали формальные подходы к объектно-ориентированному программированию, то, вероятно, вам знакомы выражения *Has-A* и *Is-A*, используемые для описания компонентов ориентации объекта. Некоторым разработчикам удобно представлять Happens-Before и Synchronizes-With как базовые блоки, помогающие понять параллелизм в Java. Это делается по аналогии с Has-A и Is-A, но между двумя вышеупомянутыми парами концепций нет прямой технической связи.

На рис. 4.11 показан пример записи в изменчивую (`volatile`) переменную, которая синхронизируется с происходящим позже доступом для чтения (для `println`).

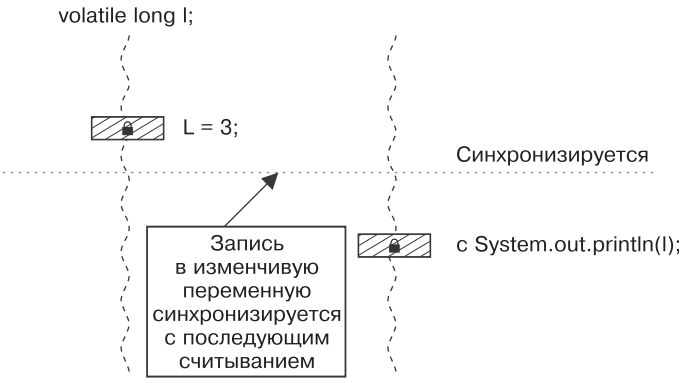


Рис. 4.11. Пример Synchronizes-With

В JMM действуют следующие основные правила:

- операция разблокировки на мониторе синхронизируется с более поздними операциями блокировки;
- запись во временную переменную синхронизируется с последующими считываниями переменной;
- если действие A синхронизируется с действием B, то действие A происходит до действия B;

- если в программе действует порядок, при котором A происходит раньше, чем B, и это совершается в рамках потока, то A происходит до B.

Общая формулировка первых двух правил сводится к тому, что «высвобождение происходит до получения блокировки». Иными словами, блокировки, удерживаемые потоком во время записи, высвобождаются до того, как смогут быть получены другими операциями (в том числе операциями считывания).

Есть и другие правила, которые, в сущности, обеспечивают логичное поведение программы:

- завершение конструктора происходит до того, как начинает работать финализатор этого объекта (объект должен быть полностью сконструирован, лишь после этого его можно будет финализировать);
- действие, запускающее поток, синхронизируется с первым действием нового потока;
- `Thread.join()` синхронизируется с последним (и всеми другими) действием в подсоединяемом потоке;
- если X происходит до Y, а Y происходит до Z, то X происходит до Z (транзитивность).

Эти простые правила полностью определяют представление платформы о том, как на ней работает память и синхронизация. На рис. 4.12 проиллюстрировано правило транзитивности.

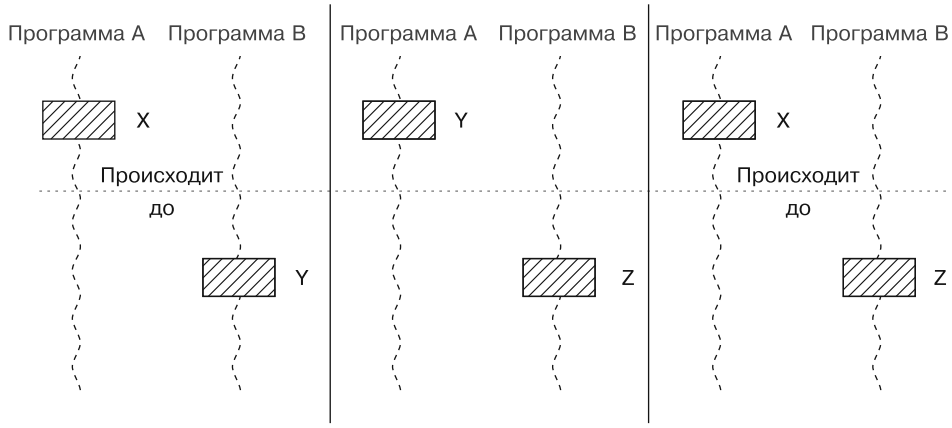


Рис. 4.12. Транзитивность при Happens-Before

ПРИМЕЧАНИЕ

На практике эти правила обеспечивают минимальные гарантии со стороны JMM. Реальные виртуальные машины Java в действительности могут работать гораздо лучше, чем можно предположить на основании этих гарантий. Для разработчика это может стать настоящей западней. Правильное поведение конкретной виртуальной машины Java может дать ложное чувство надежности, которое разобьется о первый же «заскок», вызванный неявной ошибкой в организации параллелизма.

Имея такие минимальные гарантии, легко понять, почему неизменяемость является настолько важной концепцией в параллельном программировании на Java. Если объекты нельзя изменить, то нет и проблем, связанных с обеспечением видимости изменений для всех потоков.

4.7. Резюме

Параллелизм — это одна из самых важных возможностей платформы Java. Хорошему разработчику со временем все больше будет необходимо уверенно разбираться в этой теме. Мы рассмотрели основы параллелизма в Java, а также структурные силы, действующие в многопоточных системах. Мы обсудили модель памяти, действующую в Java, и низкоуровневые подробности реализации параллелизма на этой платформе.

Для современного Java-разработчика еще важнее то, что мы обсудили классы, входящие в состав `java.util.concurrent`. К этому инструментарию нужно обращаться в первую очередь, если вы создаете многопоточный код для новых версий Java. Мы рассказали вам о тонкостях некоторых новейших классов, появившихся в Java 7, в частности о `LinkedTransferQueue` и о фреймворке fork/join.

Надеемся, вы приобрели необходимые базовые знания и сумеете использовать классы `java.util.concurrent` в своем коде. Это основной навык, который должен был у вас сформироваться после прочтения этой главы. Хотя мы рассмотрели некоторые важнейшие теоретические вопросы, самая ценная часть этой главы — практическая. Даже если вы только начинаете работать с `ConcurrentHashMap` и атомарными (`Atomic`) классами, то будете использовать хорошо протестированные классы, которые сразу же принесут коду реальную пользу.

Перейдем к следующей обширной теме, которая поможет вам состояться как Java-разработчику. В главе 5 вы приобретете фундаментальные базовые знания по еще одному важнейшему аспекту работы с платформой — загрузке классов и использованию байт-кода. Эта тема имеет первостепенное значение для реализации многих функций платформы, связанных с производительностью и безопасностью. Кроме того, она подкрепляет многие сложные технологии, применяемые в экосистеме. А значит, ее должен изучить любой разработчик, желающий опередить конкурентов.

5 Файлы классов и байт-код

В этой главе:

- загрузка классов;
- дескрипторы методов;
- строение файлов классов;
- байт-код виртуальной машины Java и почему он так важен;
- `invokedynamic` — новая инструкция.

Один из проверенных и надежных способов стать основательным Java-разработчиком — подробно разобраться в том, как именно работает платформа. Если хорошо ориентироваться в базовых возможностях платформы, таких как загрузка классов и суть байт-кода виртуальной машины Java, то достичь этой цели будет гораздо проще.

Допустим, у вас есть приложение, в котором широко используются возможности внедрения зависимостей, например фреймворк Spring. При запуске приложения начинают возникать проблемы, и оно аварийно завершается, выдавая непонятное сообщение об ошибке. Если проблема не сводится к обычной ошибке в конфигурации, то для ее отслеживания и решения вы должны понимать, как именно реализован фреймворк внедрения зависимостей. Это означает, что необходимо разбираться в загрузке классов.

Возьмем другой пример. Допустим, исполнитель, с которым вы работали, отходит от дел — и у вас остается полная версия скомпилированного кода, исходного кода вообще не остается, а имеющаяся документация пестрит пробелами. Как вам исследовать скомпилированный код и понять, из чего он состоит?

Все приложения, кроме самых простых, могут аварийно завершиться с исключением `ClassNotFoundException` или ошибкой `NoClassDefFoundError`, но многие разработчики не знают, что именно это означает, в чем разница между такими ошибками и почему они происходят.

В этой главе мы сосредоточимся на тех особенностях платформы, которые лежат в основе этих проблем разработки. Кроме того, здесь мы обсудим и некоторые более сложные возможности — этот материал рассчитан на энтузиастов, и вы можете его пропустить, если спешите.

Начнем с обзора загрузки классов. В ходе их загрузки виртуальная машина находит и активизирует новый тип для использования в работающей программе.

Основными проблемами этого обсуждения являются объекты типа `Class`, представляющие в виртуальной машине различные типы. Далее мы рассмотрим новый API дескрипторов методов (method handles) и сравним его с известными подходами из Java 6 — например, с рефлексией.

После этого мы поговорим о различных инструментах для проверки и препарирования файлов классов. В качестве базового инструмента возьмем `javap`, входящий в состав Oracle JDK (инструментария для разработки на Java). Разобрав этот урок о строении файлов классов, перейдем к изучению байт-кода. Мы рассмотрим основные семейства кодов операций виртуальной машины Java и изучим, как происходит низкоуровневая работа среды времени исполнения.

Вооружившись практическими знаниями байт-кода, мы познакомимся с новым кодом операции `invokedynamic`. Этот код появился в Java 7 для того, чтобы другие языки (не Java) могли максимально эффективно использовать виртуальную машину Java как платформу.

Начнем с рассмотрения загрузки классов — в ходе этого процесса новые классы встраиваются в работающий процесс виртуальной машины Java.

5.1. Загрузка классов и объекты классов

Файл с расширением `.class` определяет тип для виртуальной машины Java — вместе с полями, методами, информацией о наследовании, аннотациями и другими метаданными. Формат файлов классов хорошо описан в стандартах, их должен придерживаться любой язык, который предполагается запускать на виртуальной машине Java.

Класс — это мельчайшая единица программного кода, которую может загрузить платформа. Чтобы сделать класс частью исполняемого состояния виртуальной машины Java, нужно выполнить несколько шагов. Во-первых, файл класса должен быть загружен и связан, потом он должен пройти тщательную проверку (верификацию). После этого новый объект `Class`, представляющий тип, будет доступен работающей системе и можно будет создавать его новые экземпляры.

В этом разделе мы обсудим все эти этапы и познакомимся с загрузчиками классов. Это классы, контролирующие весь описываемый процесс. Начнем с изучения загрузки и связывания классов.

5.1.1. Обзор — загрузка и связывание

Назначение виртуальной машины Java — потреблять файлы классов и выполнять байт-код, содержащийся в них. Чтобы это делать, виртуальная машина Java должна получить содержимое файла класса в виде байтового потока данных, преобразовать данные в форму, пригодную для использования, и добавить эту информацию к исполняемому состоянию. Этот двухэтапный процесс называется *загрузкой и связыванием* (loading and linking). При этом связывание подразделяется на несколько более мелких этапов.

Загрузка

На первом этапе берется байтовый поток тех данных, из которых состоит файл класса, после чего это застывшее представление класса вновь «оживляется». Процесс начинается с байтового массива (зачастую считываемого из файловой системы). Создается объект `Class`, соответствующий загружаемому вами классу. В ходе этого процесса класс проходит несколько базовых проверок. По окончании процесса загрузки объект `Class` еще не является полнофункциональным, использовать его пока нельзя.

Связывание

После завершения загрузки класс должен быть связан. Этот этап делится на три более мелких — верификация, подготовка и разрешение. Верификация подтверждает, что файл класса соответствует ожидаемым параметрам и не будет вызывать в работающей системе ошибок времени исполнения либо других проблем. Затем класс проходит подготовку. Все другие типы, на которые присутствуют ссылки в файле класса, будут найдены, чтобы гарантировать, что класс готов к работе.

Соотношение между этапами связывания показано на рис. 5.1.

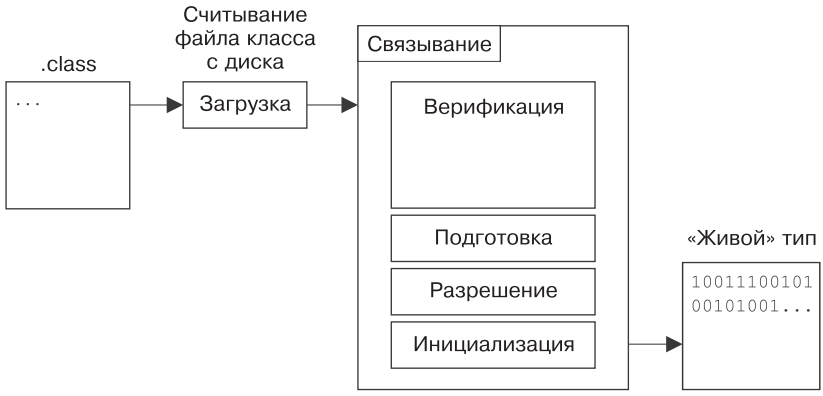


Рис. 5.1. Загрузка и связывание (вместе с второстепенными этапами связывания)

Верификация

Верификация может быть довольно сложным процессом, состоящим из нескольких стадий.

Сначала происходит базовая проверка целостности. В сущности, это часть загрузки, гарантирующая, что файл класса правильно сформирован, чтобы можно было перейти к связыванию.

На следующем этапе осуществляются проверки, гарантирующие, что символьная информация, содержащаяся в пуле констант (см. подраздел 5.3.3), самодостаточна и подчиняется базовым правилам поведения, действующим для констант.

Другие статические проверки, не связанные с просматриванием кода (например, проверка того, что методы `final` не переопределены), также осуществляются именно на этом этапе.

Затем начинается наиболее сложная часть верификации — проверка байт-кода методов. При этом необходимо гарантировать, что байт-код работает правильно и не пытается обойти системы управления экосистемой виртуальной машины. Далее перечислены некоторые из важнейших проверок, осуществляемых на этом этапе:

- проверка того, что все методы при работе учитывают ключевые слова, обеспечивающие контроль доступа;
- проверка того, что все методы вызываются с верным количеством параметров верных статических типов;
- проверка, позволяющая убедиться, что байт-код не пытается манипулировать стеком зловредными способами;
- проверка, гарантирующая, что перед использованием все переменные правильно инициализируются;
- проверка, гарантирующая, что переменным присваиваются лишь правильно типизированные для них значения.

Эти проверки выполняются ради обеспечения производительности — они позволяют пропускать текущие проверки времени исполнения, поэтому интерпретируемый код работает быстрее. Кроме того, они упрощают компиляцию байт-кода в машинный код во время исполнения (это и есть динамическая компиляция, о которой мы поговорим в разделе 6.6).

Подготовка

На этапе подготовки класса под него выделяется память, а статические переменные класса подготавливаются к инициализации. Но на этом этапе сами переменные не инициализируются, а байт-код виртуальной машины не выполняется.

Разрешение

На этапе разрешения виртуальная машина гарантирует, что все типы, на которые проставлены ссылки из нового класса, будут известны во время исполнения. Если какие-то типы неизвестны, то их также может понадобиться загрузить. Поэтому может вновь начаться процесс загрузки классов, уже для любых новых типов, которые теперь стали видны.

После того как все дополнительные типы, обязательные к загрузке, будут найдены и разрешены, виртуальная машина может инициализировать класс, который изначально было приказано загрузить. На этом последнем этапе любые статические переменные могут быть инициализированы и любые блоки статической инициализации могут быть запущены. Теперь вы используете байт-код из свежезагруженного класса. Когда этот шаг будет выполнен, класс будет полностью загружен и готов к работе.

5.1.2. Объекты классов

Конечным результатом процесса загрузки и связывания является готовый объект `Class`, представляющий собой свежезагруженный и связанный тип. Теперь он полностью функционален в виртуальной машине, хотя из соображений производительности некоторые свойства объекта `Class` инициализируются только по требованию. Теперь ваш код может продолжать работу, использовать новый тип и создавать новые экземпляры. Кроме того, объект `Class` того или иного типа предоставляет несколько полезных методов, например `getSuperClass()`, возвращающий объект `Class`, соответствующий базовому типу.

Объекты классов могут использоваться с API Reflection для непрямого доступа к методам, полям, конструкторам и т. д. Объект `Class` имеет ссылки на объекты `Method` и `Field`, соответствующие членам класса. Эти объекты могут использоваться в Reflection API для предоставления непрямого доступа к возможностям класса. Обобщенная структура этих взаимосвязей показана на рис. 5.2.

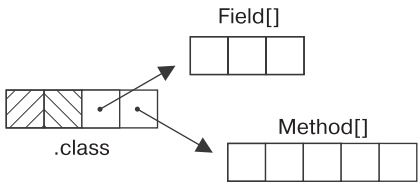


Рис. 5.2. Объект Class и ссылки

Мы еще не говорили о том, какая часть среды времени исполнения отвечает за нахождение и связывание байтового потока, который станет новым загруженным классом. Эта работа выполняется загрузчиками классов. Загрузчики являются подклассами абстрактного класса `ClassLoader`. О них и пойдет речь далее.

5.1.3. Загрузчики классов

В составе платформы присутствует несколько типичных загрузчиков классов, которые применяются для решения различных задач в процессе запуска и обычной эксплуатации платформы.

- *Базовый загрузчик* (bootstrap classloader) — инстанцируется на очень раннем этапе запуска виртуальной машины и обычно реализуется в нативном коде. Зачастую его удобнее представлять как часть самой виртуальной машины. Обычно он используется для загрузки базовых системных архивов JAR — как правило, это `rt.jar`. Никакой верификацией он не занимается.
- *Загрузчик расширений* (extension classloader) — применяется для загрузки стандартных расширений, действующих в пределах отдельной установки. На этом этапе часто устанавливаются расширения, отвечающие за безопасность.
- *Системный загрузчик* (system classloader, он же — загрузчик приложений) — наиболее часто используемый загрузчик классов. Именно он занимается загрузкой

классов приложений, а также выполняет основную часть работы в большинстве SE-окружений.

- *Специальный загрузчик* (custom classloader) — в некоторых окружениях, например в EE или в сравнительно изощренных фреймворках SE, часто используется еще и несколько дополнительных (так называемых специальных) загрузчиков классов. Некоторые команды даже пишут загрузчики классов, специфичные для их собственных приложений.

Наряду с основной областью применения загрузчики классов также используются для загрузки ресурсов (то есть файлов, не являющихся классами, например изображений или конфигурационных файлов). Загрузка ресурсов осуществляется из JAR или других местоположений, относящихся к пути класса.

ПРИМЕР — ИНСТРУМЕНТИРУЮЩИЙ ЗАГРУЗЧИК КЛАССОВ

Простой пример загрузчика классов, трансформирующегося в ходе загрузки, — загрузчик, применяемый в EMMA, инструменте для проверки покрытия программы тестами. EMMA можно скачать по адресу http://emma.sourceforge.net/.

Загрузчик классов EMMA изменяет байт-код загружаемых классов по мере их загрузки и добавляет к ним дополнительную информацию об инструментировании. Когда тестовые сценарии применяются к трансформированному коду, EMMA записывает, какие именно методы и основные ветви тестируются в ходе конкретных сценариев. Здесь разработчик может видеть, насколько полными получаются тесты компонентов для того или иного класса. Подробнее о тестах и о покрытии кода тестами мы поговорим в главах 11 и 12.

Кроме того, нередко встречаются фреймворки и другой код, в которых используются специализированные (и даже определяемые пользователем) загрузчики классов с дополнительными свойствами. Они часто трансформируют байт-код по мере его загрузки, о чем мы кратко упоминали в главе 1.

На рис. 5.3 показана иерархия наследования загрузчиков классов, а также взаимосвязи между различными загрузчиками.

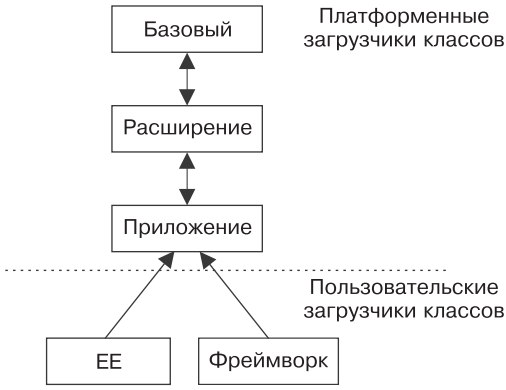


Рис. 5.3. Иерархия загрузчиков классов

Рассмотрим пример специализированного загрузчика классов и исследуем, как загрузка классов может использоваться для реализации внедрения зависимостей.

5.1.4. Пример — загрузчики классов при внедрении зависимостей

Основная идея внедрения зависимостей является двоякой:

- между функциональными единицами в составе системы существуют зависимости и конфигурационная информация; для правильного функционирования необходимо опираться на эти зависимости и информацию;
- обычно зависимости сложно или неудобно выражать в контексте самих объектов.

У вас в голове уже должна сложиться картина из классов, обладающих собственным поведением, а также из конфигурационной информации и зависимостей, которые являются внешними относительно объектов. Именно вторая часть этой модели обычно называется *подключениями объектов, действующими во время исполнения* (runtime wiring).

В главе 3 мы изучали внедрение зависимостей на примере фреймворка Guice. В этом разделе поговорим о том, как фреймворк может использовать загрузчики классов для реализации внедрения зависимостей. Правда, подход, который мы здесь обсудим, совсем не похож на Guice. В сущности, здесь будет представлена упрощенная версия фреймворка Spring.

Рассмотрим, как запускалось бы приложение в нашем воображаемом фреймворке с внедрением зависимостей:

```
java -cp <CLASSPATH> org.framework.DIMain /path/to/config.xml
```

Путь `CLASSPATH` должен содержать JAR-файлы для фреймворка с внедрением зависимостей, а также для любых классов, на которые проставлены ссылки в файле `config.xml` (вместе со всеми другими возможными у них зависимостями).

Адаптируем под этот стиль один из примеров, рассмотренных нами выше. Служба, показанная в листинге 3.7, очень удобна для этого — результат преобразования демонстрируется в листинге 5.1.

Листинг 5.1. HollywoodService — альтернативный пример внедрения зависимостей

```
public class HollywoodServiceDI {
  private AgentFinder finder = null;

  public HollywoodServiceDI() {}

  public void setFinder(AgentFinder finder) {
    this.finder = finder;
  }

  public List<Agent> getFriendlyAgents() {
```

Пустой конструктор

Метод-установщик

```
    ...
  }

  public List<Agent> filterAgents(List<Agent> agents, String agentType) {
    ...
  }
}
```

Такая же реализация, как и в листинге 3.7

Такая же реализация, как и в листинге 3.7

Для того чтобы эта конструкция могла управляться благодаря внедрению зависимостей, нужен и конфигурационный файл, примерно такой:

```
<beans>
  <bean id="agentFinder" class="wgjd.ch03.WebServiceAgentFinder"
    ... />
  <bean id="hwService" class="wgjd.ch05.HollywoodServiceDI"
        p:finder-ref="agentFinder"/>
</beans>
```

При таком подходе фреймворк для внедрения зависимостей будет использовать файл конфигурации, чтобы определить, какие объекты требуется конструировать. В этом примере нам понадобятся бины `hwService` и `agentFinder`, а фреймворк будет вызывать для каждого из них пустой конструктор, за которым следуют методы-установщики (например, `setFinder()` для зависимости `AgentFinder` из `HollywoodServiceDI`).

Это означает, что загрузка классов протекает в два этапа. На первом этапе (обслуживаемом загрузчиком приложений) загружается класс `DIMain` и все другие классы, на которые он ссылается. Потом `DIMain` начинает работать и получает местоположение конфигурационного файла в качестве параметра для `main()`.

На данном этапе фреймворк уже настроен и запущен на виртуальной машине Java, но мы еще не касались пользовательских классов, указанных в `config.xml`. На самом деле, пока `DIMain` не проверит конфигурационный файл, фреймворк никак не сможет узнать, какие классы требуется загрузить.

Чтобы добиться конфигурации приложения, описанной в `config.xml`, нужна вторая фаза загрузки классов. В ходе нее применяется специальный (пользовательский) загрузчик классов. Во-первых, файл `config.xml` проверяется на согласованность, а также на безошибочность. Потом, если все хорошо, специальный загрузчик классов пытается загрузить типы из `CLASSPATH`. Если загрузка какого-либо из типов сорвется, то будет отменен весь процесс.

Если вся операция пройдет успешно, то фреймворк для внедрения зависимостей сможет перейти к инстанцированию требуемых объектов и вызывать применительно к создаваемым экземплярам нужные методы-установщики. Если все вышеперечисленное удастся сделать правильно, то контекст приложения будет полностью настроен и готов к работе.

Мы кратко коснулись Spring-подобного подхода к внедрению зависимостей, при котором широко используется загрузка классов. Многие другие области технологий, относящиеся к Java, также тесно связаны с применением загрузчи-

ков классов и родственных практик. Вот лишь некоторые наиболее известные примеры:

- архитектуры плагинов;
- фреймворки (как полученные от производителя, так и кустарные);
- получение файлов классов из необычных местоположений (не из файловых систем и не по URL);
- Java EE;
- любые обстоятельства, в которых может понадобиться добавить новый неизвестный код уже после того, как процесс виртуальной машины Java начал работу.

На этом мы завершаем обсуждение загрузки классов. Перейдем к следующему разделу, где поговорим о новом API Java 7, который предназначен для удовлетворения некоторых нужд, связанных с рефлексией.

5.2. Использование дескрипторов методов

Если вы не знакомы с API Reflection языка Java (`Class`, `Method`, `Field` и др.), то можете просто просмотреть или даже пропустить этот раздел. С другой стороны, если вы часто используете рефлексию, то этот раздел вас, безусловно, заинтересует, так как здесь объясняются новые способы, применяемые в Java 7 для достижения аналогичных целей, но с сохранением гораздо более чистого кода.

В Java 7 появился новый API для непрямого вызова методов. Основной его компонент — пакет `java.lang.invoke`, содержащий так называемые *дескрипторы методов* (method handles). Можно считать работу с этим API более современным подходом к рефлексии, но без излишней пространности, издержек и острых углов, которые иногда свойственны для API Reflection.

ЗАМЕНА РЕФЛЕКСИВНОГО КОДА

При рефлексии требуется много шаблонного кода. Если вам доводилось писать больше нескольких строк рефлексивного кода, то вы без труда вспомните все те случаи, в которых приходилось ссылаться на типы аргументов всех интроспективных методов, например `Class[]`, аргументы всех объемлющих методов, например `Object[]`. Еще можно вспомнить о необходимости отлавливать гнусные исключения, если что-то пойдет не так, а также о сложности интуитивного понимания при чтении рефлексивного кода.

Существуют довольно веские причины для сокращения шаблонного кода и повышения интуитивной понятности всего кода. Для этого его нужно перевести на использование дескрипторов методов.

Указатели методов были разработаны в рамках проекта по подготовке `invokedynamic` для виртуальной машины Java (об этом подробнее — в разделе 5.5). Но они находят применение как в коде фреймворков, так и в обычном пользовательском коде,

далеко не ограничиваясь использованием вместе с invokedynamic. Начнем с изучения базовой технологии применения дескрипторов методов. Потом рассмотрим расширенный пример, в котором сравним дескрипторы методов с имеющимися альтернативами, и резюмируем разницу.

5.2.1. MethodHandle

Что такое MethodHandle? Общепринятый ответ таков: это типизированная ссылка на метод (или поле, конструктор и т. д.), который уже является непосредственно исполняемым. Другими словами, дескриптор метода — это объект, предоставляющий возможность безопасного вызова метода.

Получим дескриптор на двухаргументный метод (имя которого мы даже можем не знать), а потом вызовем наш дескриптор применительно к объекту obj, передав в качестве аргументов arg0 и arg1:

```
MethodHandle mh = getTwoArgMH();

MyType ret;
try {
  ret = mh.invokeExact(obj, arg0, arg1);
} catch (Throwable e) {
  e.printStackTrace();
}
```

Такая возможность отчасти напоминает рефлексию, а отчасти — интерфейс Callable, о котором мы говорили в разделе 4.4. На самом деле Callable — это более ранняя попытка смоделировать возможность вызова метода. Но с Callable связана одна проблема: этот интерфейс способен лишь моделировать методы, и они не принимают аргументов. Чтобы учесть все возможные на практике наборы различных комбинаций параметров и возможностей вызова, потребуется создавать иные интерфейсы с конкретными комбинациями параметров.

Именно этот подход зачастую использовался в Java 6, но такая практика быстро привела к неконтролируемому росту количества интерфейсов, что может доставлять разработчику существенные проблемы (например, может закончиться память в области для постоянно существующих объектов (PermGen), где хранятся классы (см. главу 6)). Напротив, дескрипторы методов могут моделировать любую сигнатуру метода без необходимости создания множества маленьких классов. Это делается с помощью нового класса MethodType.

5.2.2. MethodType

MethodType — **это неизменяемый объект, представляющий сигнатуру типа метода**. Каждый дескриптор метода имеет экземпляр MethodType, включающий тип возвращаемого значения и типы аргументов. Но он не включает имя метода или «тип получателя» — тот тип, применительно к которому вызывается метод экземпляра.

Для получения новых экземпляров MethodType можно использовать фабричные методы в классе MethodType. Вот несколько примеров:

```
MethodType mtToString = MethodType.methodType(String.class);
MethodType mtSetter = MethodType.methodType(void.class, Object.class);
MethodType mtStringComparator = MethodType.methodType(int.class,
  String.class, String.class);
```

Это экземпляры MethodType, представляющие сигнатуры типа toString(), метод-установщик (для члена типа Object) и метод compareTo(), определяемый Comparator<String>. Обобщенный экземпляр строится по такому же принципу. Сначала передается возвращаемый тип, а потом типы объектов (все — как объекты Class), вот так:

```
MethodType.methodType(RetType.class, Arg0Type.class, Arg1Type.class, ...);
```

Как видите, различные сигнатуры методов теперь могут быть представлены как обычные объекты-экземпляры. Вам уже не требуется определять новый тип для каждой сигнатуры, которая потребовалась при работе. Кроме того, так вы можете обеспечить максимально возможную безопасность типов. Если вы хотите узнать, может ли определенный дескриптор метода быть вызван с конкретным набором аргументов, то можете проверить MethodType, относящийся к дескриптору.

Итак, мы рассмотрели, как объекты MethodType позволяют решать проблему стихийного роста количества интерфейсов. Теперь научимся получать новые дескрипторы методов, которые указывают на методы наших типов.

5.2.3. Поиск дескрипторов методов

В листинге 5.2 показано, как получить дескриптор, указывающий на метод toString() актуального класса. Обратите внимание: mtToString точно совпадает с сигнатурой toString() — он имеет тип возвращаемого значения String и не принимает аргументов. Это означает, что соответствующий экземпляр MethodType является MethodType.methodType(String.class).

Листинг 5.2. Поиск дескриптора метода

```
public MethodHandle getToStringMH() {
  MethodHandle mh;
  MethodType mt = MethodType.methodType(String.class);
  MethodHandles.Lookup lk = MethodHandles.lookup();

  try {
    mh = lk.findVirtual(getClass(), "toString", mt);
  } catch (NoSuchMethodException | IllegalAccessException mhx) {
    throw (AssertionError)new AssertionError().initCause(mhx);
  }
  return mh;
}
```

Получаем контекст поиска

С помощью контекста находим дескриптор

Для получения дескриптора метода нужно использовать объект поиска (lookup object), такой как lk в листинге 5.2. Это объект, способный предоставлять дескриптор

для любого метода, видимого из того контекста исполнения, в котором был создан объект поиска.

Для того чтобы получить дескриптор метода от объекта поиска, требуется передать класс, содержащий нужный вам метод, имя метода и тип `MethodType`, соответствующий желаемой сигнатуре.

ПРИМЕЧАНИЕ

Контекст поиска можно применять для получения дескрипторов методов любого типа, в том числе системных типов. Разумеется, если вы получаете дескрипторы от класса, с которым не имеете связи, то контекст поиска позволит рассмотреть или получать лишь дескрипторы общедоступных (public) методов. Это означает, что дескрипторы методов всегда безопасны для использования с диспетчерами безопасности — аналога уловки с setAccessible(), применяемой при рефлексии, не существует.

Теперь, когда у вас есть дескриптор метода, логично научиться его вызывать. В API Method Handles для этого существует два основных способа: использовать методы `invokeExact()` и `invoke()`. Метод `invokeExact()` требует, чтобы типы аргументов в точности совпадали с теми типами, которые ожидает получить базовый метод. Метод `invoke()` совершает некоторые преобразования, пытаясь получить типы, довольно точно совпадающие с ожидаемыми, если совпадение кажется недостаточно полным (например, при необходимости применяется распаковка или упаковка типов).

В следующем подразделе приведен более обширный пример того, как дескрипторы методов могут использоваться для замены прежних техник — таких как рефлексия и использование небольших классов-посредников.

5.2.4. Пример: сравнение рефлексии, использования посредников и дескрипторов методов

Если вам приходилось подолгу работать с кодом, в котором широко применяется рефлексия, то вы, пожалуй, не понаслышке знакомы с проблемами, характерными для рефлексивного кода. В этом подразделе мы продемонстрируем, как дескрипторы методов используются для замены обширного рефлексивного шаблонного кода. Надеемся, что после этого ваша повседневная программистская практика станет немного проще.

В листинге 5.3 показан адаптированный пример, взятый из одной из предыдущих глав. `ThreadPoolManager` отвечает за назначение новых задач в пуле потоков, он слегка изменен по сравнению с примером из листинга 4.15. Кроме того, он позволяет отменить выполняемую в данный момент задачу, но этот метод скрыт для использования другими классами.

Чтобы продемонстрировать разницу между дескрипторами методов и другими приемами, мы предложили три различных способа доступа к закрытому методу `cancel()` извне класса — эти методы выделены в листинге полужирным шрифтом. Кроме того, мы демонстрируем два приема в стиле Java 6 — использование рефлексии и класса-посредника — и сравниваем их с новым подходом, основанным на `MethodHandle`. Мы используем задачу считывания очереди (эта задача называется

QueueReaderTask и реализует Runnable). Реализация QueueReaderTask находится в исходном коде, сопровождающем эту главу[1].

Листинг 5.3. Предоставление доступа тремя способами

```
public class ThreadPoolManager {
  private final ScheduledExecutorService stpe =
  Executors.newScheduledThreadPool(2);
  private final BlockingQueue<WorkUnit<String>> lbq;

  public ThreadPoolManager(BlockingQueue<WorkUnit<String>> lbq_) {
    lbq = lbq_;
  }

  public ScheduledFuture<?> run(QueueReaderTask msgReader) {
    msgReader.setQueue(lbq);
    return stpe.scheduleAtFixedRate(msgReader, 10, 10,
    TimeUnit.MILLISECONDS);
  }

  private void cancel(final ScheduledFuture<?> hndl) {
    stpe.schedule(new Runnable() {
      public void run() { hndl.cancel(true); }
    }, 10, TimeUnit.MILLISECONDS);
  }

  public Method makeReflective() {
    Method meth = null;
    try {
      Class<?>[] argTypes = new Class[] { ScheduledFuture.class };
      meth = ThreadPoolManager.class.getDeclaredMethod("cancel",
      argTypes);
      meth.setAccessible(true);
    } catch (IllegalArgumentException | NoSuchMethodException
    | SecurityException e) {
      e.printStackTrace();
    }
    return meth;
  }

  public static class CancelProxy {
    private CancelProxy() { }
    public void invoke(ThreadPoolManager mae_, ScheduledFuture<?> hndl_) {
      mae_.cancel(hndl_);
    }
  }

  public CancelProxy makeProxy() {
```

Закрытый метод для доступа

Необходимо для получения доступа к закрытому методу

[1] Можно скачать по адресу www.manning.com/TheWell-GroundedJavaDeveloper. — *Примеч. ред.*

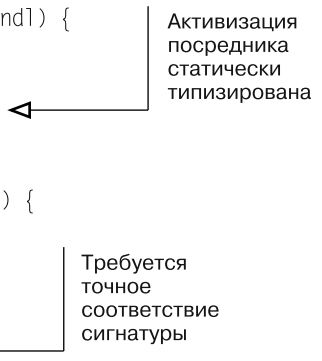
```
    return new CancelProxy();
  }

  public MethodHandle makeMh() {
    MethodHandle mh;
    MethodType desc = MethodType.methodType(void.class,
    ScheduledFuture.class);
    try {
      mh = MethodHandles.lookup()
      .findVirtual(ThreadPoolManager.class, "cancel", desc);
    } catch (NoSuchMethodException | IllegalAccessException e) {
      throw (AssertionError)new AssertionError().initCause(e);
    }
    return mh;
  }
}
```

В этом классе предоставляются возможности для доступа к закрытому методу `cancel()`. На практике обычно предоставляется лишь одна из них — мы демонстрируем все три лишь для того, чтобы объяснить разницу между ними.

Чтобы посмотреть, как использовать эти возможности, обратимся к листингу 5.4.

Листинг 5.4. Использование возможностей доступа

```
private void cancelUsingReflection(ScheduledFuture<?> hndl) {
  Method meth = manager.makeReflective();

  try {
    System.out.println("With Reflection");
    meth.invoke(hndl);
  } catch (IllegalAccessException | IllegalArgumentException
  | InvocationTargetException e) {
    e.printStackTrace();
  }
}

private void cancelUsingProxy(ScheduledFuture<?> hndl) {
  CancelProxy proxy = manager.makeProxy();
  System.out.println("With Proxy");
  proxy.invoke(manager, hndl);
}

private void cancelUsingMH(ScheduledFuture<?> hndl) {
  MethodHandle mh = manager.makeMh();

  try {
    System.out.println("With Method Handle");
    mh.invokeExact(manager, hndl);
```



```
  } catch (Throwable e) {
    e.printStackTrace();
  }
}
```

Необходимо отлавливать Throwable

```
BlockingQueue<WorkUnit<String>> lbq = new LinkedBlockingQueue<>();
manager = new ThreadPoolManager(lbq);

final QueueReaderTask msgReader = new QueueReaderTask(100) {
  @Override
  public void doAction(String msg_) {
    if (msg_ != null) System.out.println("Msg recvd: "+ msg_);
  }
};
hndl = manager.run(msgReader);
```

Используем hndl для того, чтобы позже отменить задачу

Все методы `cancelUsing` принимают `ScheduledFuture` в качестве параметра, поэтому можете воспользоваться предыдущим кодом и поэкспериментировать с различными способами отмены. На практике, будучи пользователем API, вы можете и не вникать в то, как именно он реализован.

В следующем подразделе мы поговорим о том, почему разработчику API или фреймворка стоит использовать именно дескрипторы методов, а не альтернативные варианты.

5.2.5. Почему стоит выбирать дескрипторы методов

В последнем подразделе мы рассмотрели, как дескрипторы методов используются в ситуациях, где в Java 6 применялись бы рефлексия или посредники. Таким образом, возникает вопрос: а почему стоит выбирать именно дескрипторы методов, а не другие альтернативы?

В табл. 5.1 показано, что основное достоинство рефлексии — ее привычность. В простых практических случаях посредники могут быть проще для понимания, но мы считаем, что дескрипторы методов совмещают наилучшие черты двух первых подходов. Настоятельно рекомендуем использовать дескрипторы во всех новых приложениях.

Таблица 5.1. Сравнение технологий непрямого доступа к методам, применяемых в Java

Функция	Рефлексия	Посредник	Дескриптор метода
Контроль доступа	Необходимо использовать setAccesible(). Активный менеджер безопасности может это запретить	Внутренние классы могут получать доступ к ограниченным методам	Из подходящего контекста разрешен полный доступ ко всем методам. Никаких проблем с менеджерами безопасности

Продолжение ⇩

Таблица 5.1 (продолжение)

Функция	Рефлексия	Посредник	Дескриптор метода
Проверка типов	Отсутствует. При несовпадении — некрасивое исключение	Статические. Могут быть слишком строгими. Для всех посредников может потребоваться слишком большой объем постоянного поколения памяти	Во время исполнения гарантируется безопасность типов. Постоянное поколение памяти не потребляется
Производительность	Медленная скорость работы по сравнению с альтернативными вариантами	Такая же скорость, как и при любом другом вызове метода	Стремление достичь такой же скорости, как и при других вызовах методов

Еще одна дополнительная функция, предоставляемая дескрипторами методов, — это возможность определить текущий класс из статического контекста. Если вы когда-либо писали код логирования (например, log4j) приблизительно такого рода:

```
Logger lgr = LoggerFactory.getLogger(MyClass.class);
```

то знаете, насколько он хрупок. Если осуществить его рефакторинг для перемещения в класс-наследник или базовый класс, то при явном использовании имени класса могут возникать проблемы. Но в Java 7 можно написать так:

```
Logger lgr = LoggerFactory.getLogger(MethodHandles.lookup().lookupClass());
```

В этом коде выражение `lookupClass()` можно считать эквивалентом `getClass()`, который допустимо использовать в качестве статического контекста. Это особенно полезно в тех ситуациях, когда требуется работать с фреймворками логирования, в которых обычно применяется по одному логгеру на пользовательский класс.

Имея в арсенале такую новую технологию, как дескрипторы методов, рассмотрим некоторые низкоуровневые детали файлов классов, а также инструменты, помогающие в них разобраться.

5.3. Исследование файлов классов

Файлы классов — это двоичные блобы[1], то есть с ними довольно сложно работать напрямую. Но существуют обстоятельства, в которых бывает необходимо подробно разобраться в файле класса.

Допустим, в вашем приложении требуется сделать общедоступными дополнительные методы, чтобы оптимизировать мониторинг исполнения задач (например, с помощью JMX). Кажется, что повторная компиляция и повторное развертывание выполняются без проблем. Но если проверить управляющий API, то окажется, что этих методов там нет. Дополнительные шаги по пересборке и повторному развертыванию не дают никакого эффекта.

[1] Блоб (от англ. binary linked object — «объект двоичной компоновки») — объектный файл без публично доступных исходных кодов, загружаемый в ядро операционной системы. — *Примеч. ред.*

Чтобы исправить такую проблему с развертыванием, может понадобиться проверить, на самом ли деле javac создал тот файл класса, который должен был создать. Или же придется исследовать класс, к которому у вас нет исходного кода или документация которого (на ваш взгляд) неверна.

Для решения таких и подобных задач требуется пользоваться инструментами, которые проверяют содержимое файлов классов. К счастью, в составе стандартной виртуальной машины Java от Oracle поставляется инструмент, называемый javap. Он очень удобен для заглядывания внутрь файлов классов и их дизассемблирования.

Начнем с введения в javap и изучения основных переключателей, предоставляемых для просмотра характеристик файлов классов. Потом мы исследуем представления имен методов и типов, которыми внутрисистемно пользуется виртуальная машина Java. Далее мы рассмотрим пул констант — в виртуальной машине Java это настоящий «ларчик полезностей», играющий важную роль в понимании принципов работы байт-кода.

5.3.1. Знакомство с javap

Инструмент javap может использоваться для решения множества полезных задач — от просмотра того, какие методы объявлены в классе, до вывода байт-кода. Рассмотрим простейший случай использования javap применительно к классу Update для обновления микроблогов, который мы обсуждали в главе 4.

```
$ javap wgjd/ch04/Update.class
Compiled from "Update.java"
public class wgjd.ch04.Update extends java.lang.Object {
  public wgjd.ch04.Author getAuthor();
  public java.lang.String getUpdateText();
  public int hashCode();
  public boolean equals(java.lang.Object);
  public java.lang.String toString();
  wgjd.ch04.Update(wgjd.ch04.Update$Builder, wgjd.ch04.Update);
}
```

По умолчанию javap отображает методы с уровнем видимости public, protected и — доступ по умолчанию — защищенные на уровне пакета методы. При применении переключателя -p вы также увидите закрытые (private) методы и поля.

5.3.2. Внутренняя форма сигнатур методов

На внутрисистемном уровне виртуальная машина Java использует несколько иную форму сигнатур методов, чем та, что знакома нам по человекочитаемой версии кода, отображаемой в javap. Когда мы подробнее займемся изучением виртуальной машины Java, эти внутрисистемные имена будут встречаться нам чаще. Если вы хотите поскорее отправиться дальше, можете пока пропустить этот раздел, но не забудьте, где он находится, — возможно, к нему придется обращаться при чтении последующих глав и разделов.

В компактной форме имена типов сокращаются. Например, тип `int` обозначается буквой `I`. Такие сокращенные формы иногда также называются *дескрипторами типов*. Их полный список приведен в табл. 5.2.

Таблица 5.2. Дескрипторы типов

Дескриптор	Тип
B	byte
C	char (16-битный символ Unicode)
D	double
F	float
I	int
J	long
L<имя_типа>;	Ссылочный тип (например, Ljava/lang/String; — для строки)
S	short
Z	boolean
[	Массив

В некоторых случаях название дескриптора типа может быть длиннее, чем имя типа, отображаемое в исходном коде (например, `Ljava/lang/Object;` длиннее, чем `Object`; но имена дескрипторов типов генерируются полными, поэтому их можно напрямую разрешать).

`javap` предоставляет полезный переключатель `-s`, выводящий дескрипторы типов сигнатур для некоторых методов, рассмотренных нами выше:

```
$ javap -s wgjd/ch04/Update.class
Compiled from "Update.java"
public class wgjd.ch04.Update extends java.lang.Object {
  public wgjd.ch04.Author getAuthor();
    Signature: ()Lwgjd/ch04/Author;

  public java.lang.String getUpdateText();
    Signature: ()Ljava/lang/String;

  public int compareTo(wgjd.ch04.Update);
    Signature: (Lwgjd/ch04/Update;)I

  public int hashCode();
    Signature: ()I

  ...
}
```

Как видите, каждый тип в сигнатуре метода представлен дескриптором типа.

В следующем примере показан еще один способ использования дескрипторов типов. Такая практика применяется в одной из важных частей файла класса — пуле констант.

5.3.3. Пул констант

Пул констант — это область, в которой предоставляются удобные краткие варианты именования для доступа к другим (константным) элементам файла класса. Если вы изучали такие языки, как C или Perl, где явно используются таблицы символов, то можете считать пул констант аналогом таких таблиц, применяемым в виртуальной машине Java. Но, в отличие от других языков, Java не предоставляет полного доступа к информации, содержащейся в пуле констант.

Рассмотрим очень простой пример работы с пулом констант — так мы не запутаемся в деталях. В листинге 5.5 показан простой «тренировочный» класс. Он позволяет без труда протестировать синтаксическую возможность или библиотеку Java. Для этого напишем небольшой фрагмент кода в `run()`.

Листинг 5.5. Простой тренировочный класс

```
package wgjd.ch04;

public class ScratchImpl {
  private static ScratchImpl inst = null;

  private ScratchImpl() {
  }

  private void run() {
  }

  public static void main(String[] args) {
    inst = new ScratchImpl();
    inst.run();
  }
}
```

Чтобы просмотреть информацию в пуле констант, можно использовать `javap -v`. Эта команда выводит множество дополнительной информации, а не только пул констант. Но мы сосредоточимся на тех записях из пула констант, которые относятся к тренировочному классу.

Вот пул констант:

```
 #1 = Class          #2          // wgjd/ch04/ScratchImpl
 #2 = Utf8           wgjd/ch04/ScratchImpl
 #3 = Class          #4          // java/lang/Object
 #4 = Utf8           java/lang/Object
 #5 = Utf8           inst
 #6 = Utf8           Lwgjd/ch04/ScratchImpl;
 #7 = Utf8           <clinit>
 #8 = Utf8           ()V
 #9 = Utf8           Code
#10 = Fieldref       #1.#11
➥ // wgjd/ch04/ScratchImpl.inst:Lwgjd/ch04/ScratchImpl;
#11 = NameAndType    #5:#6       // instance:Lwgjd/ch04/ScratchImpl;
#12 = Utf8           LineNumberTable
```

```
#13 = Utf8          LocalVariableTable
#14 = Utf8          <init>
#15 = Methodref     #3.#16      // java/lang/Object."<init>":()V
#16 = NameAndType   #14:#8      // "<init>":()V
#17 = Utf8          this
#18 = Utf8          run
#19 = Utf8          ([Ljava/lang/String;)V
#20 = Methodref     #1.#21      // wgjd/ch04/ScratchImpl.run:()V
#21 = NameAndType   #18:#8      // run:()V
#22 = Utf8          args
#23 = Utf8          [Ljava/lang/String;
#24 = Utf8          main
#25 = Methodref     #1.#16      // wgjd/ch04/ScratchImpl."<init>":()V
#26 = Methodref     #1.#27
➥ // wgjd/ch04/ScratchImpl.run:([Ljava/lang/String;)V
#27 = NameAndType   #18:#19     // run:([Ljava/lang/String;)V
#28 = Utf8          SourceFile
#29 = Utf8          ScratchImpl.java
```

Как видите, записи в пуле констант типизированы. Кроме того, они ссылаются друг на друга. Так, запись типа `Class` ссылается на запись типа `Utf8`. Запись `Utf8` означает строку. Таким образом, запись `Utf8`, на которую указывает запись `Class`, будет именем этого класса.

В табл. 5.3 показан набор возможных записей при работе с пулом констант. Иногда записи из пула констант рассматриваются с префиксом `CONSTANT_`, например `CONSTANT_Class`.

Таблица 5.3. Записи в пуле констант

Имя	Описание
Class	Константа класса. Указывает на имя класса (в виде записи Utf8)
Fieldref	Определяет поле. Указывает на Class и NameAndType этого поля
Methodref	Определяет метод. Указывает на Class и NameAndType этого поля
InterfaceMethodref	Определяет метод интерфейса. Указывает на Class и NameAndType этого поля
String	Строковая константа. Указывает на запись Utf8, содержащую символы
Integer	Целочисленная константа (4 байт)
Float	Константа с плавающей точкой (4 байт)
Long	Многоразрядная (длинная) константа (8 байт)
Double	Константа двойной точности с плавающей точкой (8 байт)
NameAndType	Описывает пару из имени и типа. Тип указывает на Utf8, где содержится дескриптор типа
Utf8	Поток байтов, представляющий символы в кодировке Utf8
InvokeDynamic	Новинка Java 7, см. раздел 5.5
MethodHandle	Новинка Java 7, описывает константу MethodHandle
MethodType	Новинка Java 7, описывает константу MethodType

Пользуясь этой таблицей, можете рассмотреть пример разрешения констант из пула, применяемого в тренировочном классе. Возьмем, скажем, `Fieldref` из записи `#10`.

Чтобы разрешить поле, вам потребуется знать имя, тип и класс, в котором оно находится. `#10` имеет значение `#1.#11`, что соответствует константе `#11` из класса `#1`. Легко убедиться, что `#1` действительно является константой типа `Class`, а `#11` — `NameAndType`. Запись `#1` ссылается на сам класс `ScratchImpl`, а `#11` — это `#5:#6`, переменная, именуемая `inst` типа `ScratchImpl`. Итак, в принципе, `#10` ссылается на статическую переменную `inst` в самом классе `ScratchImpl` (о чем вы и сами могли догадаться из кода, показанного в листинге 5.5).

На верификационном этапе загрузки классов есть шаг, в ходе которого мы убеждаемся, что статическая информация в файле класса является непротиворечивой (согласованной). В предыдущем примере показан вариант проверки целостности, которая осуществляется средой времени исполнения при загрузке нового класса.

Мы рассмотрели некоторые основы структурирования файла класса. Перейдем к следующей теме — байт-коду. Когда вы поймете, как исходный код превращается в байт-код, станете лучше разбираться в принципах работы всего нашего кода. В свою очередь, это поможет вам яснее представлять возможности платформы, но об этом подробнее поговорим в главе 6 и далее.

5.4. Байт-код

До сих пор байт-код оставался в нашем разговоре на заднем плане. Сейчас мы рассмотрим его подробнее — для этого вспомним, что мы уже о нем знаем.

- Байт-код — это непосредственное представление программы. Он занимает промежуточное положение между кодом, пригодным для чтения человеком, и машинным кодом.
- Байт-код создается с помощью `javac` из файлов исходного кода на Java.
- Некоторые высокоуровневые языковые возможности удалены из байт-кода при компиляции. Например, отсутствуют циклические конструкции языка Java (`for`, `while` и т. п.). В байт-коде они показаны в виде команд ветвления (перехода).
- Каждый код операции представлен одним байтом (отсюда и название «байт-код»).
- Байт-код — это абстрактное представление, а не «машинный код для воображаемого процессора».
- Байт-код может быть далее скомпилирован в машинный код, обычно это делается динамически.

При изучении байт-кода мы сталкиваемся с ситуацией, немного напоминающей проблему «курицы и яйца». Чтобы полностью понимать, что происходит, нужно ориентироваться как в байт-коде, так и в среде времени исполнения, в которой он работает.

Возникает своего рода циклическая зависимость. Чтобы разрешить ее, рассмотрим относительно простой пример. Даже если вы не совсем досконально его

поймете с первого раза, можете вернуться к нему позже, когда подробнее почитаете о байт-коде в последующих разделах.

После этого примера мы опишем контекст, в котором нужно рассматривать среду времени исполнения, а потом каталогизируем коды операций виртуальной машины Java. В частности, поговорим о кодах для арифметических операций, операций активизации, кратких формах и т. п. Наконец, мы изучим еще один пример, посвященный конкатенации строк. Для начала попробуем исследовать байт-код из файла `.class`.

5.4.1. Пример: дизассемблирование класса

При использовании `javap` с ключом `-c` можно дизассемблировать классы. В нашем примере мы воспользуемся тренировочным классом, который показан в листинге 5.5. Наша основная цель — исследовать байт-код, который содержится внутри методов. Кроме того, мы будем использовать ключ `-p` — так мы увидим и тот байт-код, который находится в закрытых методах.

Мы отдельно рассмотрим каждый раздел вывода `javap` — все они очень информативны. Если попытаться вникнуть во всю эту информацию сразу, то легко можно запутаться. Итак, начнем с заголовка. Здесь нет ничего страшного, неожиданного или даже интересного:

```
$ javap -c -p wgjd/ch04/ScratchImpl.class
Compiled from "ScratchImpl.java"
public class wgjd.ch04.ScratchImpl extends java.lang.Object {
  private static wgjd.ch04.ScratchImpl inst;
```

Далее перейдем к статическому блоку. Здесь и происходит инициализация переменной, а именно инициализация `inst` в `null`. Ситуация такова, что `putstatic` можно представить в виде байт-кода для вставки значения в статическое поле.

```
static {};
  Code:
    0: aconst_null
    1: putstatic #10 // Поле inst:Lwgjd/ch04/ScratchImpl;
    4: return
```

Числа в предыдущем коде означают задержку в потоке байт-кода с момента запуска метода. Итак, байт 1 соответствует коду операции `putstatic`, а байты 2 и 3 представляют 16-разрядный индекс в пуле констант. В данном случае 16-разрядный индекс имеет значение 10, то есть значение (в данном случае `null`) будет сохраняться в поле, которому соответствует запись `#10` в пуле констант. Байт 4 от начала потока байт-кода — это код операции `return`, то есть конец блока кода.

Переходим к конструктору.

```
private wgjd.ch04.ScratchImpl();
  Code:
    0: aload_0
    1: invokespecial #15 // Метод java/lang/Object."<init>":()V
    4: return
```

Не забывайте, что в Java пустой конструктор всегда неявно вызывает конструктор суперкласса. Здесь вы наблюдаете этот шаг в байт-коде — речь о команде `invokespecial`. Вообще, любой вызов метода преобразуется в одну из инструкций вызова виртуальной машины.

В методе `run()` нет никакого кода, так как мы работаем лишь с пустым тренировочным классом:

```
private void run();
  Code:
    0: return
```

В главном методе вы инициализируете `inst` и немного занимаетесь созданием объектов. Это один из самых простых и распространенных паттернов байт-кода, которые нужно научиться узнавать:

```
public static void main(java.lang.String[]);
  Code:
    0: new              #1    // класс wgjd/ch04/ScratchImpl
    3: dup
    4: invokespecial    #21   // метод "<init>":()V
```

Этот паттерн состоит из трех команд байт-кода: `new`, `dup` и `invokespecial` для `<init>` и всегда представляет собой создание нового экземпляра.

Код операции `new` просто выделяет память для нового экземпляра. Код операции `dup` дублирует элемент, находящийся на вершине стека. Чтобы полностью создать объект, необходимо вызвать тело конструктора. В методе `<init>` содержится код конструктора, так что вы вызываете этот блок кода с помощью `invokespecial`. Рассмотрим оставшиеся байт-коды основного метода:

```
       7: putstatic      #10   // поле inst:Lwgjd/ch04/ScratchImpl;
      10: getstatic      #10   // поле inst:Lwgjd/ch04/ScratchImpl;
      13: invokespecial  #22   // метод run:()V
      16: return
}
```

Команда 7 сохраняет предварительно созданный экземпляр-одиночку (синглтон). Команда 10 возвращает его на верхнюю позицию в стеке, так что команда 13 может вызвать метод применительно к этому экземпляру. Обратите внимание: команда 13 — это `invokespecial`, поскольку вызываемый метод `run()` является закрытым (приватным). Закрытые методы не могут быть переопределены, поэтому вам не нужно, чтобы здесь осуществлялся стандартный виртуальный поиск Java. Большинство вызовов методов будут преобразовываться в команды `invokevirtual`.

ПРИМЕЧАНИЕ

Вообще, байт-код, создаваемый javac, — это довольно простое представление, не отличающееся глубокой оптимизацией. Общая стратегия заключается в том, что основную часть оптимизации выполняют динамические компиляторы, а начинается работа со сравнительно простой точки. Выражение «байт-код должен быть тупым» отражает общее отношение разработчиков виртуальной машины к байт-коду, получаемому из исходных языков.

Далее перейдем к обсуждению среды времени исполнения, которая нужна для работы байт-кода. После этого мы познакомим вас с таблицами, которые используются для описания основных семейств инструкций байт-кода. К основным операциям относятся: загрузка/хранение, арифметические операции, контроль выполнения, активизация методов и платформенные операции. Потом мы обсудим возможные формы сокращенного доступа к кодам операций, а далее перейдем к другому примеру.

5.4.2. Среда времени исполнения

Чтобы понимать байт-код, необходимо разбираться в том, как управляется стековая машина, применяемая в виртуальной машине Java.

Одно из самых очевидных свойств, которым виртуальная машина Java отличается от аппаратного процессора (например, x64 или ARM-чипа), — отсутствие в ней регистров процессора. Вместо них для всех вычислений и операций используется стек. Иногда он именуется стеком операндов или стеком вычислений. На рис. 5.4 показано, как стек операндов может использоваться при операции сложения двух целых чисел.

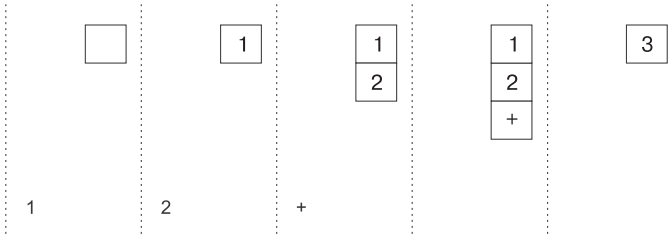


Рис. 5.4. Использование стека для операций над числами

Как было указано выше в этой главе, при связывании класса с работающей средой ее байт-код будет проверяться. Большая часть верификации сводится к анализу паттерна типов в стеке.

ПРИМЕЧАНИЕ

Манипуляции со значениями стека будут возможны лишь в том случае, если значения в стеке имеют правильные типы. Неопределенные или вредные вещи могут произойти, если, например, мы поместили в стек ссылку на объект, а потом попытались поступить с ней как с int и выполнить над ней арифметические операции. На этапе верификации при загрузке классов проходят подробные проверки, призванные гарантировать, что методы из загружаемых классов не пытаются злоупотреблять стеком. Таким образом, неграмотно оформленные (или изначально зловредные) классы даже не попадают в систему и, соответственно, не вызывают проблем.

Пока метод работает, ему требуется область памяти, которую он будет использовать как стек вычислений для расчета новых значений. Кроме того, каждый работающий поток нуждается в стеке вызовов, где на лету записывается, какие методы в настоящий момент находятся в работе (именно этот стек будет отражаться

в стектрейсе). В некоторых случаях два этих стека могут взаимодействовать. Рассмотрим, например, следующий код:

```
return 3 + petRecords.getNumberOfPets("Ben");
```

Чтобы произвести такие вычисления, вы помещаете 3 в стек операндов. Потом нужно вызвать метод, вычисляющий, сколько домашних любимцев (pets) у Бена. Для этого вы помещаете объект-получатель (тот самый, применительно к которому вы вызываете метод, — в данном случае petRecords) в стек вычислений. За объектом следуют все аргументы, которые вы хотите передать.

Затем метод getNumberOfPets() будет вызван с применением кода операции активизации. В результате управление будет передано вызванному методу, а тот метод, в который мы только что перешли, появится в стеке вызовов. Но как только вы входите в новый метод, требуется перейти к использованию нового стека операндов. Это делается для того, чтобы значения, которые уже могут находиться в стеке операндов вызывающего метода, не повлияли на результаты вычислений, выполняемых в вызываемом методе.

Когда метод getNumberOfPets() завершается, возвращаемое значение помещается в стек операндов вызывающей стороны. Это происходит в рамках того же процесса, в ходе которого getNumberOfPets() удаляется из стека вызовов. После этого операция сложения может взять два значения и суммировать их.

Перейдем к исследованию байт-кода. Это обширная тема с множеством специальных случаев. Мы собираемся провести обзор основных возможностей, а не доскональное исследование.

5.4.3. Введение в коды операций

Байт-код виртуальной машины Java состоит из последовательности кодов операций (опкодов), причем за каждой командой может идти по нескольку аргументов. Коды операций ожидают увидеть стек в конкретном состоянии, а потом преобразовать его — так, что из него будут удалены аргументы, на место которых будут вставлены результаты.

Каждый код операции обозначается однобайтным значением. Соответственно, всего может существовать 255 кодов операций. В настоящее время используется около 200 таких кодов. Это слишком большой список, чтобы исследовать его весь в подробностях, но, к счастью, множество кодов операций относятся к тому или иному семейству. Мы по очереди обсудим каждое из семейств, чтобы вы могли составить о них общее впечатление. Некоторые операции нельзя строго отнести к конкретному семейству, но такие коды операций встречаются сравнительно редко.

ПРИМЕЧАНИЕ

Виртуальная машина Java не является полностью объектно-ориентированной средой времени исполнения — ей знакомы и примитивные типы. Это заметно в некоторых семействах кодов операций. Отдельные простейшие коды операций (например, для хранения и сложения данных) обязательно должны иметь несколько вариаций, отличающихся в зависимости от примитивного типа, которым они манипулируют.

В таблицах с кодами операций по четыре столбца.

- *Имя* — означает общее имя типа кода операции. В некоторых случаях может быть несколько похожих кодов операций, выполняющих схожие задачи.
- *Аргументы* — представляют собой те аргументы, что принимаются кодом операции. Аргументы, начинающиеся с `i`, — это коды операций, используемые как справочный индекс в пуле констант или локальной таблице переменных. Если аргументов больше, то они объединяются. Так, `i1`, `i2` означает: «сделать 16-разрядный индекс из этих двух байт». Если аргументы даны в скобках, то они будут использоваться не во всех кодах операций.
- *Компоновка стека* — демонстрирует состояние стека до и после того, как выполнится код операции. Если элементы даны в скобках, это означает, что они будут использоваться не во всех кодах операций либо эти элементы опциональны (например, это касается кодов операций активизации).
- *Описание* — рассказывает, какова функция кода операции.

Рассмотрим пример строки из табл. 5.4, а именно строку о коде операции `getfield`. Он используется для считывания значения из поля объекта.

getfield i1, i2 [obj] → [val]	Получает поле, расположенное по указанному индексу в пуле констант и относящееся к объекту, находящемуся на верхней позиции в стеке

В первом столбце указано имя кода операции — `getfield`. Во втором столбце записано, что за кодом операции в байтовом потоке следуют два аргумента. Вместе эти аргументы составляют 16-разрядное значение, находимое системой в пуле констант для получения искомого поля (как вы помните, индексы в пуле констант всегда 16-разрядные).

Столбец с компоновкой стека показывает, что после нахождения индекса в пуле констант класса объекта, расположенного на верхней позиции в стеке, объект удаляется и заменяется значением этого поля для объекта, занимавшего верхнюю позицию в стеке.

Такой принцип удаления экземпляров объекта в ходе операции — просто один из способов держать байт-код компактным, без длительной кропотливой очистки. Кроме того, не приходится постоянно помнить о необходимости удаления тех экземпляров объектов, с которыми вы закончили работу.

5.4.4. Коды операций для загрузки и сохранения

Семейство кодов операций для загрузки и сохранения предназначено для загрузки значений в стек или получения их. В табл. 5.4 перечислены основные операции, осуществляемые в этом семействе.

Таблица 5.4. Коды операций загрузки и сохранения

Имя	Аргументы	Компоновка стека	Описание
load	(i1)	[] → [val]	Загружает в стек значение (примитив или ссылку) из локальной переменной. Имеет сокращенные формы и типоспецифические варианты
ldc	i1	[] →[val]	Загружает в стек константу из пула. Имеет зависящие от типа варианты и варианты wide
store	(i1)	[val] → []	Сохраняет значение (примитив или ссылку) в локальной переменной, удаляя его из стека в процессе работы. Имеет сокращенные формы и типо-специфичные варианты
dup		[val] → [val, val]	Дублирует значение, занимающее верхнюю позицию в стеке. Имеет вариантные формы
getfield	i1, i2	[obj] → [val]	Получает поле, расположенное по указанному индексу в пуле констант, относящееся к объекту, расположенному на верхней позиции в стеке
putfield	i1, i2	[obj, val] → []	Помещает значение в поле объекта по указанному индексу в пуле констант

Выше мы упоминали, что существует несколько различных форм команд для загрузки и сохранения. Например, есть код операции `dload`, загружающий в стек из локальной переменной число двойной точности. Другой код, `astore`, извлекает из стека ссылку на объект и помещает ее в локальную переменную.

5.4.5. Арифметические коды операций

Эти коды соответствуют операциям, выполняющим над стеком арифметические действия. Такая операция берет аргументы из верхней позиции в стеке и производит над ними требуемые вычисления. Аргументы (всегда относящиеся к примитивным типам) должны в точности совпадать, но платформа предлагает и множество таких кодов операций, которые предназначены для приведения типов друг к другу. В табл. 5.5 перечислены базовые арифметические операции.

Коды операций приведения имеют очень краткие названия, например `i2d` для приведения `int` к `double`. Следует отметить, что слово cast (приведение) не фигурирует в названиях, поэтому в таблице оно дано в скобках.

Таблица 5.5. Коды для арифметических операций

Имя	Аргументы	Компоновка стека	Описание
add	—	[val1, val2] → [res]	Складывает два значения (которые должны относиться к одному и тому же примитивному типу) из верхней позиции в стеке и сохраняет результат в стеке. Имеет сокращенные формы и типоспецифические варианты

Продолжение ⇩

Таблица 5.5 (продолжение)

Имя	Аргументы	Компоновка стека	Описание
sub	—	[val1, val2] → [res]	Вычитает два значения (которые должны относиться к одному и тому же примитивному типу), расположенные в верхней позиции в стеке. Имеет сокращенные формы и типоспецифические варианты
div	—	[val1, val2] → [res]	Делит два значения (которые должны относиться к одному и тому же примитивному типу), расположенные в верхней позиции в стеке. Имеет сокращенные формы и типоспецифические варианты
mul	—	[val1, val2] → [res]	Перемножает два значения (которые должны относиться к одному и тому же примитивному типу), расположенные в верхней позиции в стеке. Имеет сокращенные формы и типоспецифические варианты
(cast)	—	[value] → [res]	Приводит значение одного примитивного типа к другому. Имеет формы, специфичные для каждого возможного варианта приведения

5.4.6. Коды операций для контроля исполнения

Как было указано выше, управляющие конструкции высокоуровневых языков отсутствуют в байт-коде виртуальной машины Java. Вместо этого поток выполнения контролируется небольшим количеством примитивов, приведенных в табл. 5.6.

Таблица 5.6. Коды операций для контроля исполнения

Имя	Аргументы	Компоновка стека	Описание
if	b1, b2	[val1, val2] → [] или [val1] → []	Если конкретное условие выполняется, то необходимо перейти к указанной точке ветвления
goto	b1, b2	[] → []	Безусловный переход к точке ветвления. Имеет широкую форму
jsr	b1, b2	[] → [ret]	Переходит к локальной подпроцедуре и помещает в стек адрес возврата (смещение перед следующим кодом операции). Имеет широкую форму
ret	index	[] → []	Возврат к смещению, указанному в локальной переменной по заданному индексу
tableswitch	{depends}	[index] → []	Используется для реализации переключения
lookupswitch	{depends}	[key] → []	Применяется для реализации переключения

Подобно индексным байтам, используемым для поиска констант, аргументы `b1`, `b2` применяются для создания местоположения байт-кода в заданном методе. Позже именно в это местоположение будут выполняться переходы. Команды `jsr` используются для доступа к небольшим самодостаточным областям байт-кода, которые могут находиться вне основного потока задач (например, в смещениях после окончания

основного байт-кода данного метода). В определенных обстоятельствах (например, в блоках для обработки исключений) такие команды могут быть полезны.

Широкие формы команд `goto` и `jsr` принимают по 4 байта аргументов и создают смещение, которое зачастую бывает больше 64 Кбайт. Часто без такого смещения можно обойтись.

5.4.7. Коды операций для активизации

К этой группе относятся четыре кода операций, предназначенные для общего управления вызовом методов. К ним также относится специальный код операции `invokedynamic`, новинка Java 7. О нем мы подробнее поговорим в разделе 5.5. Пять кодов операций для активизации методов приведены в табл. 5.7.

Таблица 5.7. Коды операций активизации

Имя	Аргументы	Компоновка стека	Описание
invokestatic	i1, i2	[(val1, ...)] → []	Вызывает статический метод
invokevirtual	i1, i2	[obj, (val1, ...)] → []	Вызывает «обычный» метод экземпляра
invokeinterface	i1, i2, count, 0	[obj, (val1, ...)] → []	Вызывает метод интерфейса
invokespecial	i1, i2	[obj, (val1, ...)] → []	Вызывает «специальный» метод экземпляра
invokedynamic	i1, i2, 0, 0	[val1, ...] → []	Динамический вызов; см. раздел 5.5

Существует несколько особенностей, которые необходимо учитывать при работе с кодами операций активизации. Во-первых, `invokeinterface` имеет дополнительные параметры. Они присутствуют по причинам исторического характера и для обратной совместимости, но в настоящее время не используются. Два дополнительных нуля в `invokedynamic` предусмотрены для обеспечения дальнейшей совместимости.

Другой важный момент заключается в различии между обычным и специальным вызовом метода экземпляра. Обычный вызов виртуален. Это означает, что конкретный метод, который необходимо вызвать, отыскивается во время исполнения с применением стандартных правил переопределения методов, действующих в языке Java. При специальных вызовах переопределения не учитываются. Это бывает важно в двух случаях — при работе с закрытыми (private) методами и при вызовах методов суперкласса. В обоих случаях правила переопределения не должны включаться, поэтому вам и требуется для этого случая иной код операции активизации.

5.4.8. Коды операций для работы с платформой

Семейство кодов операций для работы с платформой содержит: код операции `new` для выделения памяти для новых экземпляров объектов, а также коды операций для работы с потоками, а именно `monitorenter` и `monitorexit`. Это семейство подробно описано в табл. 5.8.

Коды операций для работы с платформой используются для управления определенными моментами жизненного цикла объекта — например, для создания новых

объектов и блокировки их. Важно отметить, что код операции new просто выделяет место под объект. Высокоуровневая концепция создания объекта также предусматривает выполнение кода внутри конструктора.

Таблица 5.8. Коды операций для работы с платформой

Имя	Аргументы	Компоновка стека	Описание
new	i1, i2	[] → [obj]	Выделяет память для нового объекта, относящегося к типу, указываемому константой по заданному индексу
monitorenter	—	[obj] → []	Блокирует объект
monitorexit	—	[obj] → []	Разблокирует объект

На уровне байт-кода конструктор преобразуется в метод с особым именем — <init>. Он не может быть вызван из пользовательского кода Java, а вот из байт-кода — может. В результате получается характерный паттерн работы с байт-кодом, непосредственно соответствующий созданию объекта. Паттерн таков: за new следует dup, а после него — invokespecial для вызова метода <init>.

5.4.9. Сокращенные формы записи кодов операций

У многих кодов операций есть сокращенные формы, удобные, если требуется сохранить несколько байтов то тут, то там. Общий принцип заключается в том, что к одним локальным переменным требуется обращаться значительно чаще, чем к другим. Поэтому целесообразно иметь специальный код операции, означающий «произвести общую операцию прямо над локальной переменной», не указывая при этом саму локальную переменную в качестве аргумента. Так и появились коды операций aload_0 и dstore_2, относящиеся к семейству «загрузка и сохранение».

Попробуем применить эту теорию на практике и рассмотрим другой пример.

5.4.10. Пример: сцепление (конкатенация) строк

Добавим информацию в наш тренировочный класс, чтобы продемонстрировать немного более сложный байт-код — код, затрагивающий большинство тех семейств кодов операций, с которыми мы познакомились выше.

Как вы помните, строки String в Java являются неизменяемыми. Итак, что же происходит при сцеплении двух строк вместе с применением оператора +? В таком случае создается новый объект String, но в самом коде происходит больше операций, чем может показаться на первый взгляд.

Рассмотрим тренировочный класс с методом run(), откорректированным следующим образом:

```
private void run(String[] args) {
  String str = "foo";
  if (args.length > 0) str = args[0];
  System.out.println("this is my string: " + str);
}
```

Этому относительно простому методу соответствует следующий байт-код:

```
$ javap -c -p wgjd/ch04/ScratchImpl.class
Compiled from "ScratchImpl.java"

  private void run(java.lang.String[]);
    Code:
      0: ldc           #17   // Строка foo
      2: astore_2
      3: aload_1
      4: arraylength
      5: ifle          12 #A
```

Если размер переданного вами массива меньше или равен нулю, переходите к команде 12.

```
      8: aload_1
      9: iconst_0
     10: aaload
     11: astore_2
     12: getstatic     #19
➥ // поле java/lang/System.out:Ljava/io/PrintStream;
```

В предыдущей строке показано, как байт-код предоставляет доступ к `System.out`.

```
     15: new           #25   // Класс java/lang/StringBuilder
     18: dup
     19: ldc           #27   // Строка this is my string:
     21: invokespecial #29
➥ // Метод java/lang/StringBuilder."<init>":(Ljava/lang/String;)V
     24: aload_2
     25: invokevirtual #32
➥ // Метод java/lang/StringBuilder.append
➥ (Ljava/lang/String;)Ljava/lang/StringBuilder;
     28: invokevirtual #36
➥ // Метод java/lang/StringBuilder.toString:()Ljava/lang/String;
```

В этих командах демонстрируется создание объединенной строки, которую вы желаете получить в качестве вывода. В частности, команды 15–23 демонстрируют создание объекта (`new`, `dup`, `invokespecial`), но в данном случае после `dup` присутствует `ldc` (загрузочная константа). Такой паттерн байт-кода указывает, что вы вызываете непустой конструктор — в данном случае `StringBuilder(String)`.

Результат на первый взгляд может показаться немного странным. Вы просто объединяете несколько строк, а байт-код внезапно вам сообщает, что вы при этом создаете дополнительные объекты `StringBuilder`, а потом вызываете применительно к ним сначала `append()`, а затем `toString()`. Дело в том, что строки в Java неизменяемые. Вы не можете изменить строковый объект путем конкатенации, поэтому приходится создавать новый объект. `StringBuilder` — просто удобный инструмент для такой операции.

Наконец, вызовем метод для вывода результатов:

```
     31: invokevirtual #40
➥ // Метод java/io/PrintStream.println:(Ljava/lang/String;)V
     34: return
```

Теперь, когда строка вывода собрана, можно вызвать метод `println()`. Он вызывается применительно к `System.out`, поскольку два верхних элемента в стеке на данном этапе — это `[System.out, <output string>]`. Ситуация полностью соответствует табл. 5.7, где определена компоновка стека для правильного `invokevirtual`.

Чтобы стать по-настоящему основательным Java-разработчиком, нужно запустить `javap` применительно к некоторым из собственных классов и научиться распознавать распространенные паттерны байт-кода. Пока удовлетворимся этим кратким введением в работу байт-кода и перейдем к следующей теме. Речь пойдет об одной из важнейших новых возможностей в Java 7 — `invokedynamic`.

5.5. invokedynamic

В этом разделе мы поговорим об одной из самых технически затейливых возможностей Java 7. Но, несмотря на исключительный потенциал этой возможности, ее не назовешь той, без которой не обойтись любому практикующему разработчику. Такая функция применима лишь в очень сложных практических случаях. Она будет интересна для разработчиков фреймворков, а также для специалистов, работающих с не Java-языками.

Это означает, что, если вы не собирались вникать в то, как функционирует ядро платформы, либо детально разбираться в работе нового байт-кода, можете прочесть резюме этой главы и переходить к следующей.

Если же вас заинтересовала данная тема — хорошо. Мы расскажем, насколько необычен на практике код `invokedynamic`. Это новейшая разработка в области байт-кода Java — в версиях Java до 7-й ничего подобного не встречалось. Итак, новый код операции называется `invokedynamic`. Это команда вызова нового типа, она применяется при вызовах методов. `invokedynamic` сообщает виртуальной машине, что требуется отложить определение того, какой метод придется вызвать. Это означает, что виртуальной машине не требуется прорабатывать все детали на этапе компилирования и связывания — в отличие от случаев работы с другими кодами операций.

Напротив, здесь детали выбора метода уточняются во время исполнения. Для этого вызывается вспомогательный метод, принимающий решение о том, какой метод вызвать.

JAVAC ПРОТИВ INVOKEDYNAMIC

В Java 7 отсутствует непосредственная языковая поддержка для `invokedynamic`. Ни одно Java-выражение не компилируется `javac` непосредственно в байт-код `invokedynamic`. Ожидается, что в Java 8 появятся новые языковые конструкции (например, стандартные методы), которые будут использовать динамические возможности.

На самом деле `invokedynamic` — усовершенствование, ориентированное как раз не на Java, а на другие языки. Байт-код добавляется в динамические языки для тех случаев, когда их предполагается использовать на виртуальной машине Java 7 (но в некоторых толковых фреймворках Java были найдены способы для применения `invokedynamic`).

В этом разделе мы рассмотрим особенности работы `invokedynamic`, а также разберем подробный пример декомпиляции точки вызова с применением ново-

го байт-кода. Отметим, что необходимо полностью понимать этот механизм, чтобы использовать языки и фреймворки, в которых активно задействуется `invokedynamic`.

5.5.1. Как работает invokedynamic

Для поддержки `invokedynamic` в число определений пула констант было добавлено несколько новых записей. Они обеспечивают необходимую поддержку `invokedynamic`, которую невозможно предоставить в условиях технологических ограничений, действующих в Java 6.

Индекс, сопровождающий команду `invokedynamic`, должен указывать на константу типа `CONSTANT_InvokeDynamic`. К нему прикрепляется два 16-разрядных указателя (то есть 4 байта). Первый указатель служит индексом в таблице методов, по которой мы определяем, что именно вызывать. Иногда такие методы именуются *загрузочными* (bootstrap methods, BSM). Они обязательно должны быть статическими и иметь определенную сигнатуру аргументов. Второй указатель направлен на `CONSTANT_NameAndType`.

В данном случае понятно, что `CONSTANT_InvokeDynamic` напоминает обычный `CONSTANT_MethodRef`. Разница лишь в том, что при вызове `invokedynamic` не указывается, в пуле констант какого класса нужно искать метод. Для получения ответа на этот вопрос вызывается загрузочный метод.

Загрузочные методы принимают информацию о точке вызова и связывают динамический вызов, возвращая экземпляр `CallSite`. Эта точка вызова удерживает `MethodHandle`, что и становится результатом обращения к точке вызова.

Команда `invokedynamic` начинает работать без целевого метода — о ней говорят, что она является *несвязанной* (unlinked). При первом выполнении вызывается загрузочный метод для точки. Он возвращает точку `CallSite`, которая затем связывается с командой `invokedynamic`. Это показано на рис. 5.5.

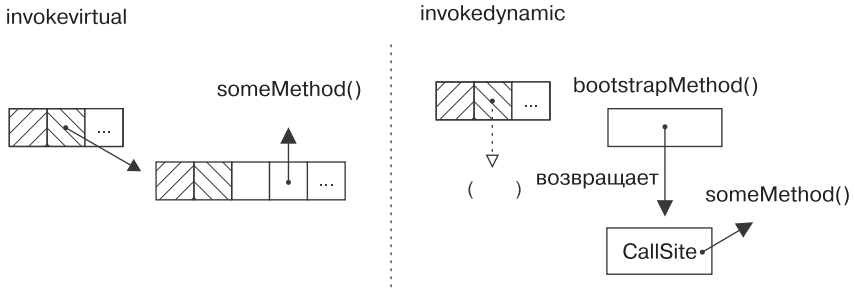


Рис. 5.5. Сравнение виртуального и динамического назначения методов

Когда точка `CallSite` уже связана, можно вызвать и сам метод — эту задачу выполняет `MethodHandle`, удерживаемый `CallSite`. Подобная конструкция означает, что вызовы `invokedynamic` потенциально можно оптимизировать с помощью динамического компилятора примерно так же, как это делается с вызовами `invokevirtual`. Такие способы оптимизации мы подробнее обсудим в следующей главе.

Стоит также отметить, что отдельные объекты CallSite можно «перенаправить» (то есть изменять целевые методы, на которые может быть проставлена ссылка на протяжении жизненного цикла). В некоторых динамических языках такая функция используется очень активно.

В следующем подразделе мы на простом примере разберем, как вызов invokedynamic может быть представлен в байт-коде.

5.5.2. Пример: дизассемблирование invokedynamic-вызова

Выше мы уже упоминали, что поддержка invokedynamic на уровне синтаксиса в Java 7 отсутствует. Поэтому приходится пользоваться библиотекой для манипуляций с байт-кодом, позволяющей создать файл .class, внутри которого содержится команда для динамического вызова. Для этой цели хорошо подходит ASM (http://asm.ow2.org/) — мощная библиотека промышленного масштаба, применяемая в множестве широко известных Java-фреймворков.

Пользуясь этой библиотекой, можно создать представление класса, включающее команду invokedynamic, а потом превратить его в поток байтов. Эта информация может быть либо записана на диск, либо передана загрузчику классов для вставки в работающую виртуальную машину.

В качестве простого примера прикажем ASM создать класс, имеющий всего один статический метод с единственной командой invokedynamic. Затем этот метод можно вызвать из обычного кода Java — он обертывает (скрывает) динамическую природу реального вызова. Реми Форакс (Remi Forax) и команда ASM предоставили простой инструмент для создания тестовых классов, которые делают именно это. Инструмент был создан в рамках разработки invokedynamic, ASM была одной из первых библиотек, полностью поддерживающих новый байт-код.

Рассмотрим байт-код для такого обертывающего метода:

```
public static java.math.BigDecimal invokedynamic();
  Code:
    0: invokedynamic #22, 0
➥   // InvokeDynamic #0:_:()Ljava/math/BigDecimal;
    5: areturn
```

Как видите, пока ничего интересного — все сложное действо разворачивается в пуле констант. Итак, рассмотрим те записи, которые относятся к динамическому вызову:

```
BootstrapMethods:
  0: #17 invokestatic test/invdyn/DynamicIndyMakerMain.bsm:
➥ (Ljava/lang/invoke/MethodHandles$Lookup;Ljava/lang/String;
➥ Ljava/lang/invoke/MethodType;Ljava/lang/Object;)
➥ Ljava/lang/invoke/CallSite;
  Method arguments:
    #19 1234567890.1234567890
#10 = Utf8              ()Ljava/math/BigDecimal;
```

```
#18 = Utf8              1234567890.1234567890
#19 = String            #18     // 1234567890.1234567890
#20 = Utf8              _
#21 = NameAndType       #20:#10 // _:()Ljava/math/BigDecimal;
#22 = InvokeDynamic     #0:#21  // #0:_:()Ljava/math/BigDecimal;
```

Чтобы разобраться в этом коде, его следует рассмотреть внимательнее.

1. Код операции `invokedynamic` указывает на запись `#22`. Она ссылается на загрузочный метод `#0` и `NameAndType` `#21`.
2. Загрузочный метод (BSM) `#0` — это обычный статический метод `bsm()`, применяемый к классу `DynamicIndyMakerMain`. Он имеет верную сигнатуру, какую и должен иметь BSM.
3. В записи `#21` указано имя этой конкретной точки динамического связывания, «_», а также возвращаемый тип, `BigDecimal` (сохраняемый в строке `#10`).
4. Запись `#19` — это статический аргумент, переданный в загрузочный метод.

Как видите, для обеспечения безопасности типов требуется немало манипуляций. Так или иначе, остается еще множество ситуаций, в которых что-то может пойти не так во время исполнения, но этот механизм обеспечивает значительную безопасность, сохраняя немалую гибкость.

ПРИМЕЧАНИЕ

Существует еще один уровень косвенности, если угодно — гибкости. Он заключается в том, что таблица BootstrapMethods указывает на дескрипторы методов, а не на сами методы. Мы не касались этого момента выше, так как он мог только запутать вас и не поспособствовал бы пониманию того, как именно работает весь механизм.

На этом мы завершаем разговор об `invokedynamic`, а также о внутренней структуре байт-кода и загрузки классов.

5.6. Резюме

В этой главе вы бегло познакомились с байт-кодом и загрузкой классов. Мы подробно исследовали формат файлов классов и вкратце рассмотрели среду времени исполнения, предоставляемую виртуальной машиной Java. Чем больше вы будете знать о внутренней организации платформы, тем более профессиональным разработчиком станете.

Надеемся, что после прочтения этой главы вы усвоили следующие моменты.

- Формат файлов классов и загрузка классов — это основные концепции, на которых построена работа виртуальной машины Java. Они важны для любого языка, который предполагается использовать на этой виртуальной машине.
- Различные фазы загрузки классов обеспечивают реализацию функций безопасности и производительности во время исполнения.
- Дескрипторы методов — это важный новый API, появившийся в Java 7. Он является альтернативой рефлексии.

- Байт-код виртуальной машины Java делится на семейства, объединяемые по функциональному признаку.
- В Java 7 появился `invokedynamic` — новый способ вызова методов.

Теперь пора перейти к следующей крупной теме, обязательной для каждого основательного Java-разработчика. Прочитав следующую главу, вы будете уверенно разбираться в анализе производительности — той теме, которая часто остается недопонятой. Вы научитесь измерять и подстраивать производительность системы, а также максимально эффективно использовать некоторые из самых мощных технологий, лежащих в основе виртуальной машины Java. В частности, вы узнаете, как работать с динамическим компилятором, превращающим байт-код в сверхбыстрый машинный код.

6 Понятие о повышении производительности

В этой главе:

- почему так важна производительность;
- новый сборщик мусора G1;
- VisualVM — инструмент для визуализации памяти;
- динамическая компиляция.

Плохая производительность губит приложения. Она раздражает клиентов и портит репутацию приложения. Если только вы не работаете на совершенно монополизированном рынке, ваши клиенты будут голосовать ногами, то есть уходить от вас к конкурентам. Чтобы плохая производительность не портила весь ваш проект, необходимо разбираться в результатах анализа производительности и знать, как извлекать из него пользу.

Анализ и подстройка производительности — это огромная тема. Более того, она часто трактуется неверно, так что при ее изучении внимание заостряется не на тех вещах. Итак, начнем с того, что поведаем вам большой секрет о настройке производительности — *необходимы измерения. Без измерений точная настройка производительности невозможна.*

Почему же? Дело в том, что человеческий мозг практически всегда ошибается, когда мы пытаемся угадать, какие компоненты системы работают наиболее медленно. Здесь ошибаются все. Вы, я, Джеймс Гослинг (James Gosling) — всем нам не уйти от подсознательной необъективности. Мы всегда усматриваем закономерности там, где их нет.

На самом деле правильный ответ на вопрос «Какая часть моего кода на Java нуждается в оптимизации?» обычно звучит так: «Никакая».

Представим себе типичное (пусть и довольно консервативное) веб-приложение для электронной торговли. В таком приложении услуги предоставляются множеству (пулу) зарегистрированных клиентов. В приложении есть SQL-база данных, веб-серверы Apache и серверы приложений Java. Все это объединено совершенно стандартной сетевой конфигурацией. Очень часто самыми сложными узкими местами системы являются те компоненты, которые написаны не на Java (в частности, база данных, файловая система, сеть). Но без измерений Java-разработчик этого так и не узнает. Вместо того чтобы найти и исправить реальную проблему,

разработчик может потратить уйму времени на оптимизацию микрофрагментов кода, которые практически не связаны с возникшей проблемой.

Вы должны уверенно отвечать на следующие фундаментальные вопросы, касающиеся производительности.

- Если на сайте начнется распродажа и у вас появится вдесятеро больше посетителей, чем бывает на ресурсе обычно, найдется ли в системе достаточно памяти, чтобы справиться с таким наплывом?
- Сколько времени вашим пользователям обычно требуется ждать отклика приложения?
- Каковы эти показатели в сравнении с показателями ваших конкурентов?

Для уверенной настройки производительности необходимо перестать гадать о том, какие элементы замедляют работу всей системы. Нужно не гадать, а знать. А чтобы знать наверняка, требуются измерения.

Кроме того, следует знать, чем не является настройка производительности. Это:

- не набор подсказок и трюков;
- не секретный соус;
- не волшебная пыльца, которой нужно осыпать готовый проект.

Особенно осторожно следует относиться к подходам, опирающимся на подсказки и трюки. Дело в том, что виртуальная машина Java — очень сложная и отлично настроенная среда. Поэтому вне контекста большинство таких советов будут бесполезны (а некоторые из них даже вредны). Кроме того, они быстро устаревают, так как виртуальная машина Java становится все интеллектуальнее в области оптимизации кода.

На самом деле анализ производительности — это наука, в основе которой лежит эксперимент. Можете относиться к вашему коду как к научному опыту, который требует ввода и дает выводы — параметры производительности, показывающие, насколько эффективно система решает поставленные перед ней задачи. Работа инженера по настройке производительности заключается в изучении этого вывода и поиске закономерностей. Таким образом, настройка производительности может считаться отраслью прикладной статистики, а не набором бабушкиных сказок и устного народного творчества.

Эта глава — своеобразное введение в практику настройки производительности в языке Java. Но эта тема очень велика, и мы сможем сообщить лишь азбучные истины, важнейшие теоретические сведения, а также обозначить основные вехи. Мы постараемся ответить на самые фундаментальные вопросы.

- Почему производительность так важна?
- Почему сложно анализировать производительность?
- Какие аспекты виртуальной машины Java потенциально вызывают сложности при оптимизации?
- Как следует воспринимать настройку производительности и работать с ней?
- Каковы наиболее распространенные подоплеки плохой производительности?

Кроме того, мы сделаем введение в работу с двумя подсистемами виртуальной машины Java, наиболее важными при настройке производительности. Это:

- подсистема сборки мусора;
- динамический компилятор.

Этого должно быть достаточно, чтобы вы могли познакомиться с темой и начали применять приобретенные знания (достаточно сложные с теоретической точки зрения) при решении реальных проблем, с которыми можете столкнуться в своем коде.

Для начала быстро разберемся с терминологическим аппаратом, который позволит вам выражать и формулировать возникающие проблемы с производительностью, а также очерчивать цели.

6.1. Терминологическое описание производительности — базовые определения

Чтобы дискуссия получилась максимально плодотворной, формализуем отдельные концепции производительности, в которых необходимо ориентироваться. Начнем с определения нескольких важнейших терминов из лексикона специалистов по производительности:

- время отклика;
- пропускная способность;
- коэффициент использования;
- эффективность;
- мощность;
- масштабируемость;
- деградация.

Некоторые из этих проблем рассмотрены Дугом Ли в контексте написания многопоточного кода, но здесь мы рассмотрим их гораздо более подробно. Говоря о производительности, можно иметь в виду как единственный многопоточный процесс, так и гораздо более крупные системы, вплоть до кластерной серверной платформы.

6.1.1. Ожидание

Ожидание (latency) — это сквозное время, необходимое для обработки отдельно взятой единицы материала при известной загруженности. Зачастую ожидание квотируется в расчете лишь на «нормальные» уровни загрузки, но при измерении производительности часто бывает полезен специальный график, на котором ожидание отображается как функция от возрастающей загруженности.

На рис. 6.1 показан график, демонстрирующий внезапное, нелинейное резкое ухудшение (деградацию) показателя производительности — например, увеличение ожидания при возрастании загруженности. Обычно такое явление называется *свечой производительности*.

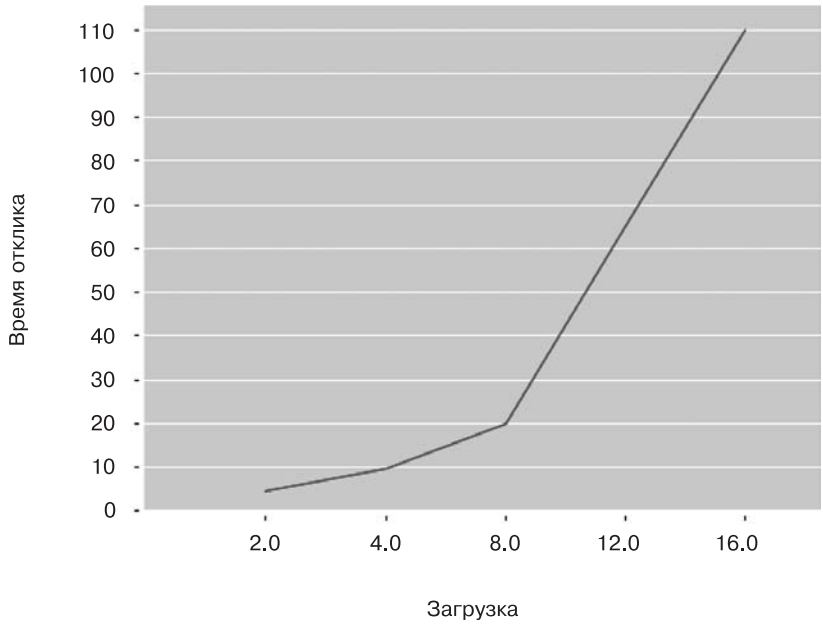


Рис. 6.1. Свеча производительности

6.1.2. Пропускная способность

Пропускная способность (throughput) — это количество единиц, которые система может выполнить за определенный период времени при заданных ресурсах. Часто говорят о количестве транзакций в секунду на какой-нибудь базовой платформе (например, с сервером известной марки, с заранее указанным аппаратным обеспечением, операционной системой и софтверным стеком).

6.1.3. Коэффициент использования

Коэффициент использования (utilization) представляет собой процент доступных ресурсов, которые применяются для выполнения полезной работы, а не для обслуживающей работы (и тем более не простаивают). Часто говорят, что сервер используется всего на 10 % — здесь речь идет о проценте вычислительной мощности процессора, выделяемой на обработку единиц в нормальном рабочем режиме. Обратите внимание: может существовать очень большая разница между степенью использования различных ресурсов, например процессора и памяти.

6.1.4. Эффективность

Эффективность (efficiency) системы представляет собой результат деления пропускной способности на количество используемых ресурсов. Система, требующая большее количество ресурсов для достижения конкретной производительности, менее эффективна, чем та, в которой такая производительность достигается меньшим количеством ресурсов.

Допустим, мы сравниваем два кластерных решения. Если решение A требует для достижения проектной пропускной способности в два раза больше серверов, чем решение B, то решение A в два раза менее эффективно, чем B.

Не забывайте, что ресурсы можно рассматривать и в контексте их стоимости — если решение X стоит вдвое больше, чем решение Y (либо если для эксплуатации производственной среды требуется в два раза больше персонала), то решение Y в два раза эффективнее решения X.

6.1.5. Мощность

Мощность (capacity) — это количество единиц работы (например, транзакций), которые могут выполняться в системе в определенный момент времени. Другими словами, это объем параллельной обработки, реализуемый при известном времени отклика или пропускной способности.

6.1.6. Масштабируемость

По мере того как в систему добавляются ресурсы, ее пропускная способность (или время отклика) изменяется. Такое изменение времени отклика или пропускной способности называется *масштабируемостью* (scalability) системы.

Если пропускная способность решения A удваивается при удвоении количества серверов в его пуле, то мы наблюдаем идеальное линейное масштабирование. В большинстве случаев бывает очень сложно достичь идеального линейного масштабирования.

Необходимо также отметить, что масштабируемость системы зависит от нескольких факторов и не является постоянной величиной. До определенного момента система может масштабироваться практически линейно, а потом вдруг начинается тяжелая деградация. Это своеобразное «пике» производительности.

6.1.7. Деградация

При добавлении в действующую сетевую систему все большего количества рабочих единиц или клиентов обычно наблюдается изменение показателей времени отклика или пропускной способности. Оно называется *деградацией* (degradation) системы под дополнительной нагрузкой.

ПОЛОЖИТЕЛЬНАЯ И ОТРИЦАТЕЛЬНАЯ ДЕГРАДАЦИЯ

Как правило, деградация — негативное явление. Это означает, что добавление новых единиц в систему отрицательно влияет на производительность (например, возрастает длительность ожидания при обработке). Но в некоторых случаях деградация может оказывать и положительный эффект.

Например, если дополнительная нагрузка позволяет определенной части системы преодолеть порог и начать работать в режиме повышенной производительности, то вся система будет действовать более эффективно и укорачивать длительность обработки, хотя абсолютное количество выполняемой работы, конечно, увеличится. Виртуальная машина Java — это очень динамичная среда времени исполнения, отдельные ее компоненты могут давать как раз такой эффект.

Рассмотренные выше термины — наиболее часто используемые характеристики производительности. Есть и другие, которые могут быть важны в определенных условиях, но они относятся к базовой статистике системы и обычно применяются для справки при настройке производительности. В следующем разделе мы опишем подход, выстроенный с внимательным учетом таких показателей. Этот подход в максимальной степени опирается на количественные данные.

6.2. Прагматический подход к анализу производительности

Многие разработчики, которым приходится обращаться к проблеме анализа производительности, в начале работы не имеют четкого представления о том, каких результатов они хотят достичь посредством анализа. Размытое ощущение того, что «код должен работать быстрее», — вот и все, что зачастую есть у разработчиков и менеджеров перед началом работы.

Но все должно быть совершенно наоборот. Чтобы по-настоящему эффективно улучшать производительность, нужно сразу получить представление о некоторых ключевых областях и лишь потом приступать к технической части работы. Итак, требуются ясные ответы на следующие вопросы.

- Какие доступные для наблюдения характеристики вашего кода вы измеряете?
- Как измерять такие характеристики?
- Каков смысл наблюдаемых характеристик?
- Как вы узнаете, что настройка производительности завершена?
- Какова максимальная приемлемая стоимость (выраженная в рабочем времени программиста, затрачиваемом на оптимизацию и в усложнении кода) такой настройки?
- На какие жертвы вы не готовы пойти ради оптимизации?

Важнее всего, как мы не раз еще отметим в этой главе, понимать, что измерения *необходимы*. Без измерений как минимум одной наблюдаемой характеристики вы не сможете выполнить анализ производительности.

Кроме того, когда вы приступаете к измерению производительности кода, очень часто оказывается, что пробуксовка происходит не там, где вы предполагали. Причина многих проблем производительности может заключаться в пропущенном индексе в базе данных или объясняться возникновением взаимоблокировок при захвате блокировок. Раздумывая об оптимизации кода, никогда не забывайте, что проблема может быть совсем не в коде. Для того чтобы количественными методами определить источник проблемы, нужно хорошо знать, что именно вы измеряете.

6.2.1. Знайте, что именно вы измеряете

При оптимизации производительности вы обязательно что-то измеряете. Если не измерять какой-либо наблюдаемой характеристики, то не может быть и речи об оптимизации производительности. Если вы сидите, уставившись в код, и надеетесь, что вас осенит, как по-быстрому решить проблему, то вы не занимаетесь анализом производительности.

СОВЕТ

Чтобы быть хорошим специалистом по анализу производительности, необходимо понимать, что такое среднее значение, медианное значение, дисперсия, перцентиль, стандартное отклонение, объем выборки и нормальное распределение. Если эти термины вам не знакомы, поищите соответствующую информацию в Интернете и почитайте дополнительные материалы.

Принимаясь за анализ производительности, важно в точности знать, какие из наблюдаемых параметров, описанных нами в предыдущем разделе, для вас наиболее важны. Всегда следует привязывать ваши измерения, цели и заключения к одной или более базовым наблюдаемым величинам, которые мы перечислили.

Вот кое-какие наблюдаемые параметры, которыми удобно пользоваться при анализе производительности:

- среднее время, необходимое методу `handleRequest()` для выполнения (после разогрева);
- 90-й перцентиль сквозного времени отклика системы при 10 параллельно работающих клиентах;
- деградация времени отклика при повышении нагрузки от 1 до 1000 параллельно работающих пользователей.

Все эти параметры представляют количественные значения, которые, возможно, инженеру понадобится измерить и откорректировать. Чтобы получить точные и полезные цифры, необходимо обладать базовыми знаниями статистики.

Если вы четко знаете, что именно измеряете, и уверены, что ваши цифры точны, то первый этап пройден. Но нечеткие или слишком свободно сформулированные цели редко дают хорошие результаты. Измерение производительности в данном отношении — не исключение.

6.2.2. Умейте проводить измерения

Есть всего два способа, позволяющих определить, сколько времени нужно на выполнение метода или другого фрагмента кода.

- Измерять непосредственно, вставляя измерительный код в исходный класс.
- Преобразовывать класс, который необходимо оценить, на этапе загрузки класса.

Проще всего пользоваться такими способами прямого измерения производительности, которые базируются на одном (или обоих) из этих методов.

Кроме того, необходимо упомянуть об инструментальном интерфейсе виртуальной машины Java (JVMTI), который часто применяется для создания очень изощренных профилировочных инструментов. У него есть свои недостатки. Для работы с этим интерфейсом необходимо писать нативный код, а выдаваемые им профилировочные показатели являются, в сущности, статистическими средними данными, а не результатами прямых измерений.

Прямые измерения

Прямые измерения — самая простая для понимания практика, но ее можно назвать немного инвазивной. В простейшей форме код выглядит так:

```
long t0 = System.currentTimeMillis();
methodToBeMeasured();
long t1 = System.currentTimeMillis();
long elapsed = t1 - t0;
System.out.println("methodToBeMeasured took "+ elapsed +" millis");
```

В качестве вывода получим строку, которая с точностью до нескольких миллисекунд показывает, сколько времени требуется на выполнение метода `methodToBeMeasured()`. Неудобство заключается в том, что такой код приходится добавлять повсюду в имеющейся кодовой базе. А по мере роста количества измерений информации становится так много, что эти данные вас просто засыпают.

Есть и другие проблемы. Например, что нужно делать, если на выполнение `methodToBeMeasured()` уходит менее одной миллисекунды? Как будет показано далее в этой главе, существуют и некоторые эффекты «холодного старта», о которых не стоит забывать. Дело в том, что на более поздних этапах работы метод вполне может выполняться быстрее, чем на более ранних.

Автоматическое измерение скорости загрузки классов

В главах 1 и 5 мы уже говорили о том, как классы собираются в исполняемую программу. Один из основных шагов, который часто недооценивают, — преобразование байт-кода на этапе загрузки. Этот этап обладает огромным потенциалом и лежит в основе многих современных приемов, используемых на платформе Java. Один из простых примеров его применения — автоматическое измерение скорости загрузки методов.

В такой ситуации метод `methodToBeMeasured()` загружается с помощью специального загрузчика классов, который добавляет байт-код в начале и в конце метода

для записи точного времени входа в метод и выхода из него. Эти данные о хронометраже обычно записываются в разделяемую структуру данных, к которой обращаются другие потоки. Такие потоки выполняют манипуляции над данными — как правило, они либо записывают вывод в файлы журналов, либо связываются с сетевым сервером, обрабатывающим сырые данные.

Подобная технология лежит в основе многих высокоточных инструментов для отслеживания производительности (таких как OpTier CoreFirst). Но на момент написания книги нам не удалось найти ни одного активно поддерживаемого свободного инструмента, который относился бы к этой нише.

ПРИМЕЧАНИЕ

Как будет показано ниже, методы Java начинают работу в интерпретируемом виде, а потом переходят в компилируемый режим. Чтобы получить точные данные о производительности, необходимо сбросить значения хронометража, которые были сгенерированы в интерпретируемом режиме, так как они могут сильно смазать результаты. Ниже мы более подробно остановимся на том, как узнать, переключился ли метод в компилируемый режим.

При использовании одного или обоих описанных подходов вы сможете получать количественные данные о том, насколько быстро выполняется метод. Следующий вопрос — какие цифры вы рассчитываете получить после того, как настройка будет завершена?

6.2.3. Знайте, какого уровня производительности вы хотите достичь

Ничто так не помогает сосредоточиться, как четкая цель. Поэтому важно знать, что требуется измерить, но не менее важно знать и конечный результат настройки, а также уметь его сообщить. В большинстве случаев цель должна быть простой и точно сформулированной:

- снизить 90-й перцентиль сквозного ожидания системы при 10 параллельно работающих клиентах на 20 %;
- снизить среднее время отклика `handleRequest()` на 40 %, а дисперсию — на 25 %.

В более сложных случаях может быть поставлена цель одновременно достичь нескольких взаимосвязанных целевых показателей. Необходимо учитывать, что чем слабее связаны наблюдаемые показатели, которые вы измеряете и пытаетесь скорректировать, тем более сложным становится сам процесс определения производительности. Оптимизация ради повышения производительности может отрицательно сказаться на других показателях.

Иногда бывает необходимо провести некоторый первичный анализ, скажем определить наиболее важные методы, и лишь потом ставить цели. Например, может потребоваться ускорить работу методов. Это хорошо, но после начальных исследований всегда лучше сделать паузу и сформулировать цели, а уже потом пытаться их достичь. Слишком часто разработчики впрягаются в аналитические работы, не потрудившись четко поставить цель.

6.2.4. Знайте, когда следует прекратить оптимизацию

Теоретически несложно определить, когда стоит заканчивать оптимизацию, — когда вы достигнете поставленной цели, тогда и остановитесь. Но на практике оказывается, что повышение производительности — процесс, который затягивает. Если все идет хорошо, то велик соблазн поднажать и сделать все еще лучше. И наоборот — если достичь цели не удается, несмотря на все старания, то хочется пробовать все новые и новые стратегии, надеясь, что какая-то из них позволит добиться успеха.

Чтобы уметь вовремя остановиться, нужно не только четко представлять себе свои цели, но и знать, чего они стоят. Зачастую бывает достаточно добиться и 90%-ного результата от целевого показателя, а рабочее время инженера, возможно, удастся с пользой потратить на другие дела.

Еще один важный метод, помогающий регулировать усилия, затрачиваемые на настройку производительности, — это поиск редко используемых участков кода. Оптимизация кода, на который приходится один процент времени исполнения программы или даже меньше, — это практически всегда напрасный труд. Просто удивительно, как часто разработчики ударяются в такую оптимизацию.

Ниже даны очень простые рекомендации, позволяющие определить, что стоит оптимизировать, а что — нет. Возможно, вам понадобится приспособить эти советы к конкретной ситуации, но достаточно часто они работают и без адаптации.

- Оптимизируйте то, что важно, а не то, что хорошо поддается оптимизации.
- В первую очередь занимайтесь самыми важными методами (обычно это такие методы, которые вызываются чаще всего).
- Если можно что-то оптимизировать без особых усилий — оптимизируйте, но при этом учитывайте, насколько часто вызывается соответствующий код.

Наконец, выполните еще одну серию измерений. Если достичь желаемых улучшений производительности не удалось, подведите предварительные итоги. Посмотрите, в какой степени удалось достичь поставленных целей, оказывают ли полученные результаты желаемый эффект на общее состояние производительности.

6.2.5. Знайте, какой ценой дается повышение производительности

Ни одна из уловок, повышающих производительность, не проходит даром.

- На выполнение анализа и подготовку оптимизации уходит время (а мы помним, что рабочее время программиста — как правило, самая дорогая расходная статья в любом проекте).
- Появляются дополнительные технические сложности, связанные с исправлением найденных недостатков (есть и такие виды оптимизации, после которых код упрощается, но в большинстве случаев складывается иная ситуация).

- Для выполнения вспомогательных задач могут задействоваться дополнительные потоки, и при высокой нагрузке они могут вызывать непредвиденные эффекты во всей системе.

Независимо от того, чем придется заплатить за повышение производительности, обращайте внимание на такие проблемы и пытайтесь выявлять их до того, как завершите очередной этап оптимизации.

Зачастую бывает полезно иметь определенное представление о максимальной цене, которую вы готовы заплатить за повышение производительности. Например, это может быть ограничение по времени, задаваемое разработчикам, занятым настройкой производительности, либо максимальное допустимое количество дополнительных классов или строк кода. Например, разработчик может решить, что оптимизации можно уделить не больше недели рабочего времени, либо что оптимизированные классы допускается увеличить не более чем на 100 % (то есть вдвое по сравнению с исходным размером).

6.2.6. Знайте об опасности поспешной оптимизации

Одна из самых знаменитых цитат на тему оптимизации принадлежит Дональду Кнуту (Donald Knuth):

> «Программисты тратят массу времени и нервов на размышления о скорости работы некритичных компонентов их программ. И эти попытки добиться эффективности оказывают сильное негативное влияние… преждевременная оптимизация — это корень зла»[1].

По поводу этого утверждения в программистском сообществе велись жаркие споры, и во многих случаях оппоненты запоминают лишь последнюю часть этого высказывания. Это плохо по нескольким причинам.

- В первой части высказывания Кнут намекает на необходимость измерений, без которых мы не сможем определить, какие части программы наиболее важны.
- Опять же необходимо помнить, что совсем не всегда задержка возникает из-за проблем в коде, — причина может заключаться совсем в другой части системного окружения.
- Из полной цитаты очевидно, что Кнут имеет в виду такую оптимизацию, которая состоит из сознательных и скоординированных действий.
- В краткой форме эта цитата используется как дежурное оправдание плохого дизайна системы или варианта ее исполнения.

Некоторые виды оптимизации действительно являются элементом хорошего стиля.

- Не выделяйте память на объект, который вам не нужен.
- Удаляйте сообщение отладочного журнала, если оно вам точно не понадобится.

---

[1] *Knuth Donald E.* Structured Programming with go to Statements / Computing Surveys, 6. — No. 4 (Dec. 1974). — http://pplab.snu.ac.kr/courses/adv_pl05/papers/p261-knuth.pdf

В следующем фрагменте кода мы добавили этап проверки, позволяющий узнать, будет ли логирующий объект что-то делать с записью из отладочного журнала. Проверка такого рода называется *страховкой на уровне логирования* (loggability guard). Если подсистема логирования не настроена на ведение отладочных журналов, то код так и не создаст сообщения журнала. Таким образом, будет сэкономлено время, которое пришлось бы потратить на вызов `currentTimeMillis()` и на создание объекта `StringBuilder`, используемого для такого сообщения.

```
if (log.isDebugEnabled()) log.debug("Useless log at: "+
    System.currentTimeMillis());
```

Но если от отладочного сообщения журнала вообще нет никакой пользы, то можно просто удалить этот код, сэкономив еще пару циклов процессора (которые потратились бы на страховку на уровне логирования).

Одна из особенностей настройки производительности заключается в том, что все начинается с написания хорошего работоспособного кода. Чем лучше вы ориентируетесь в работе платформы и ее внутреннем поведении (например, понимаете, как происходят неявные операции выделения объектов при конкатенации двух строк) и чем больше задумываетесь о производительности по ходу работы, тем более качественный код у вас получится.

Итак, вы уже владеете терминологическим аппаратом для описания проблем с производительностью и для постановки целей. Кроме того, мы в общих чертах описали, как подходить к решению проблем. Но мы по-прежнему не объяснили, почему затронутые здесь проблемы должен решать именно программист и в чем их корень. Чтобы ответить на эти вопросы, нужно подробнее поговорить об аппаратном обеспечении.

6.3. Что пошло не так? И почему нас это должно волновать?

В середине прошлого десятилетия (золотые были годы) казалось, что производительность вообще не проблема. Скорость работы часов процессора росла как на дрожжах, создавалось впечатление, что программисту нужно просто подождать несколько месяцев — и любой код, написанный плохо, заработает на новых мощных процессорах.

Почему же потом все пошло настолько плохо? Почему скорости процессоров уже не растут так быстро? Более того, почему компьютер с процессором 3 ГГц с виду работает ненамного быстрее, чем машина с 2 ГГц? Откуда взялась такая тенденция — все наши товарищи по цеху вдруг стали беспокоиться о производительности?

В этом разделе мы обсудим движущие силы упомянутой тенденции, объясним, почему даже самые убежденные разработчики программ не должны забывать и об аппаратном обеспечении. Мы очертим контекст для изучения оставшихся тем этой главы и познакомим вас с концепциями, которые понадобятся для глубокого понимания динамической компиляции и других достаточно сложных примеров.

Возможно, вы слышали широко растиражированный термин «закон Мура». Многим разработчикам известно, что этот закон как-то связан с темпом совершенствования компьютеров, но детали этого закона сформулировать уже сложнее. Подробно рассмотрим смысл этого закона и выясним, почему в самом ближайшем будущем он, вероятно, сойдет на нет.

6.3.1. Закон Мура: прошлые и будущие тенденции изменения производительности

Закон Мура назван по имени Гордона Мура (Gordon Moore), одного из основателей компании Intel. Вот одна из наиболее распространенных формулировок этого закона: «*Максимальное количество транзисторов на кристалле схемы, целесообразное с экономической точки зрения, удваивается примерно за два года*».

Этот закон, который, в сущности, является наблюдением о тенденциях развития компьютерных процессоров, выведен на основании статьи, которую Мур написал в 1965 году. Изначально в ней давался прогноз на 10 лет, то есть до 1975 года. Тот факт, что эта тенденция сохранилась до сих пор (и, вероятно, сохранится примерно до 2015 года), несомненно, замечателен.

На рис. 6.2 мы показали на графике количество реальных процессоров из семейства Intel x86 — с 1980 года до процессора i7, вышедшего в 2010 году. На графике показано количество транзисторов на дату выпуска процессора.

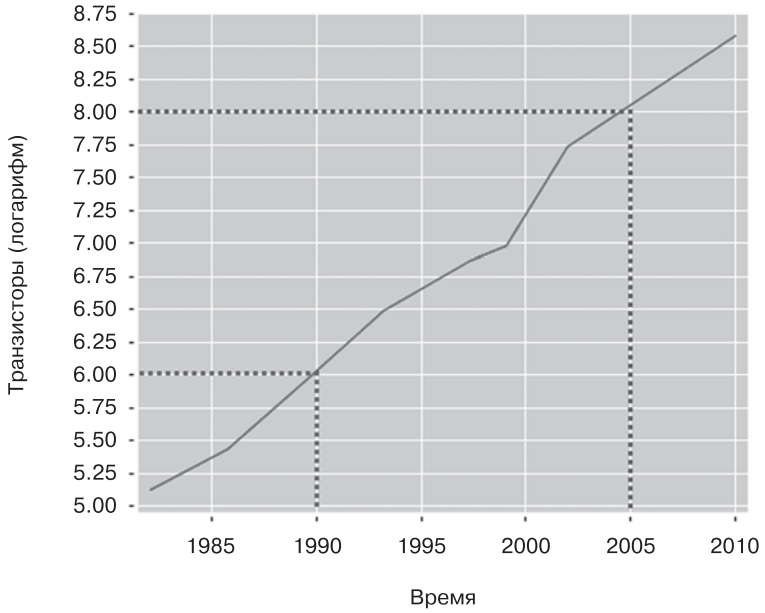


Рис. 6.2. Логарифмический линейный график увеличения количества транзисторов с течением времени

Это линейный логарифмический график. Таким образом, каждое приращение по оси *Y* в 10 раз больше предыдущего приращения. Как видите, линия почти прямая, на пересечение каждого вертикального уровня уходит шесть-семь лет. Этот график подтверждает справедливость закона Мура, так как десятикратное увеличение за шесть-семь лет — практически то же, что и удвоение за два года.

Учитывайте, что ось *Y* в этой системе координат градуирована логарифмической шкалой. Это означает, что мейнстримовый процессор Intel, произведенный в 2005 году, имеет около 100 миллионов транзисторов. Это *в 100 раз больше*, чем у процессора, произведенного в 1990 году.

Важно отметить, что закон Мура описывает именно рост количества транзисторов. Это основной момент, который необходимо уяснить, почему одного лишь закона Мура для программиста недостаточно, чтобы роскошествовать за счет инженеров-системотехников.

ПРИМЕЧАНИЕ

Количество транзисторов — не то же самое, что скорость таймера процессоров. Даже общеизвестная идея о том, что чем выше скорость таймера процессора, тем выше производительность, — это грубое упрощение.

Раньше закон Мура был хорошим ориентиром, и он, по-видимому, будет действовать еще какое-то время (оценки разнятся, но до 2015 года тенденция должна сохраниться). Но закон Мура сформулирован именно на основании количества транзисторов, а они все хуже отражают производительность, которую разработчики могут ожидать достичь в своем коде. Как видим, реальность гораздо сложнее.

Истина заключается в том, что практическая производительность зависит от множества факторов и все они важны. Но если бы пришлось выбрать всего один фактор, то это была бы скорость поиска данных, релевантных для выполнения команд. Это настолько важная концепция в области производительности, что мы должны остановиться на ней подробнее.

6.3.2. Понятие об иерархии латентности памяти

Компьютерному процессору необходимы данные, которые он будет обрабатывать. Если у него нет этих данных, то неважно, насколько быстро тикают циклы процессора. Процессор просто вынужден простаивать, пока данные не станут доступны.

Это означает, что существует два следующих фундаментальных вопроса, которые связаны с задержками (латентностью) и на которые необходимо ответить: «Где находится ближайший экземпляр данных, который требуется обработать в ядре процессора?» и «Как скоро программа сможет дойти до этих данных, чтобы процессор мог приступить к их обработке?» Ниже перечислены возможные варианты ответов на эти вопросы.

- *Регистры* — это ячейки с памятью, расположенные в процессоре и готовые для немедленной обработки. Это именно та часть памяти, над которой непосредственно работают команды.

- *Основная память* — обычно это динамическое ОЗУ (DRAM). На обращение к этой памяти тратится около 50 нс, но ниже мы подробно опишем, как кэш процессора помогает избежать такой задержки.
- *Твердотельные диски (SSD)* — для доступа к ним требуется 0,1 мс или менее; это гораздо быстрее, чем обращение к обычным жестким дискам.
- *Жесткие диски* — для обращения к такому диску и для загрузки требуемых данных в основную память требуется около 5 мс.

Закон Мура описывает экспоненциальный рост количества транзисторов, который также положительно сказывается на возможностях памяти, так как скорость доступа к ней тоже возрастает по экспоненте. Но оказывается, что две эти экспоненты неодинаковы. Скорость работы памяти возрастала не так быстро, как увеличение количества процессоров транзистора. Из-за этого постепенно повышался риск, что ядру процессора придется работать вхолостую, так как в нем нет нужных данных, готовых к обработке.

Для решения этой проблемы были встроены кэши между регистрами процессора и основной памятью. Это небольшие участки более быстрой памяти (статическая оперативная память, SRAM). Такая быстрая память оказывается гораздо дороже, чем динамическое ОЗУ (DRAM) как в денежном выражении, так и в количестве транзисторов. Именно поэтому SRAM не заменила в компьютерах всю DRAM.

Кэши обозначаются как L1 и L2 (в некоторых машинах также есть L3). Число указывает, насколько близко к процессору расположен кэш (а чем ближе кэш, тем быстрее он работает). Мы подробнее поговорим о кэшах в разделе 6.6 (посвященном динамической компиляции) и на примере покажем, насколько важны эффекты кэша L1 для работающего кода. На рис. 6.3 продемонстрировано, насколько кэши L1 и L2 превосходят по скорости работы основную память. Ниже мы приведем пример того, как сильно эта разница в скорости сказывается на производительности работающего кода.

Рис. 6.3. Относительная длительность обращения к регистрам, кэшам процессора и основной памяти

В 1990-х и начале 2000-х годов наряду с добавлением кэшей активно использовалась и иная технология. Для внедрения все более сложных функций процессора предпринимались попытки избавиться от латентности памяти. Для обеспечения постоянной загруженности процессора обрабатываемыми данными использовались довольно сложные аппаратные технологии, такие как параллелизм на уровне команд

(ILP) и внутрипроцессорная многопоточность (CMT). Эти попытки предпринимались в условиях растущей пропасти между мощностью процессоров и латентностью памяти.

Новые технологии стали потреблять значительный объем транзисторных мощностей процессора и оказывать серьезное негативное влияние на реальную производительность. Именно поэтому сформировалась точка зрения, что в будущем в производстве процессоров будут доминировать механизмы со многими ядрами.

Это означает, что прогресс производительности неразрывно связан с многопоточностью. Чтобы сделать систему более работоспособной во всех отношениях, в ней необходимо увеличить количество процессорных ядер. Таким образом, если даже одно ядро будет дожидаться данных для обработки, остальные ядра смогут функционировать в прежнем режиме. Эта связь настолько важна, что мы хотели бы еще раз подчеркнуть:

- будущее принадлежит многоядерным процессорам;
- из-за этого возникает тесная связь между производительностью и многопоточностью.

Такие аппаратные проблемы стоят не только перед Java-программистами. Однако управляемая природа виртуальной машины Java оборачивается еще несколькими сложностями в этой сфере. Поговорим о них в следующем разделе.

6.3.3. Почему так сложно выполнять оптимизацию производительности в Java

Повышение производительности на виртуальной машине Java (впрочем, как и в любой другой управляемой среде времени исполнения) по природе более сложно, чем при работе с кодом, работающим в неуправляемом виде. Причина в том, что программисты, пишущие на C/C++, практически все делают сами. Операционная система предоставляет лишь минимальный набор служб, например рудиментарное планирование потоков.

Вся суть управляемой системы заключается в том, что среда времени исполнения получает возможность приобрести определенный контроль над всей экосистемой, так что разработчику не приходится самому заботиться обо всех деталях. Благодаря этому повышается общая производительность труда программиста, но от контроля над системой в какой-то степени приходится отказаться. Единственная альтернатива — отказаться вместо этого от всех преимуществ, существующих в управляемой среде времени исполнения, что практически всегда оборачивается более серьезными неудобствами, чем необходимость дополнительной оптимизации производительности.

Вот некоторые важные функции платформы, из-за которых усложняется оптимизация производительности:

- диспетчеризация потоков;
- сборка мусора;
- динамическая компиляция.

Между этими функциями могут возникать довольно тонкие взаимодействия. Например, система компиляции использует таймеры, чтобы решать, какие методы компилировать. Это означает, что на множество методов — кандидатов для компиляции могут влиять такие факторы, как диспетчеризация потоков и сборка мусора. Набор компилируемых методов может отличаться от запуска к запуску.

Весь этот раздел доказывает, что точные измерения — основной фактор, определяющий пути принятия решений при анализе производительности. Понимание того, как платформа Java обращается со временем (а также понимание ограничений, связанных с этим процессом), — очень важный навык для всех, кто намерен серьезно заниматься повышением производительности.

6.4. Вопрос времени — от железа и вверх

Вы когда-нибудь задумывались, где в компьютере хранится и обрабатывается время? В конечном счете за отслеживание времени отвечает аппаратное обеспечение, но картина не так проста, как это может показаться!

Для повышения производительности нужно хорошо понимать детали работы со временем. Чтобы приобрести такое понимание, поговорим о базовом оборудовании, потом обсудим, как Java интегрируется с этими подсистемами, и, наконец, обратимся к некоторым сложностям, связанным с методом `nanoTime()`.

6.4.1. Аппаратные часы

В современной машине, работающей на процессоре x64, может быть до четырех аппаратных источников времени: RTC, 8254, TSC и, возможно, HPET.

Часы реального времени (real-time clock, RTC) — это, в принципе, такая же электроника, которая работает в дешевых цифровых часах на кварцевых кристаллах. Когда система выключена, они работают от батареи материнской платы. Именно эти часы устанавливают системное время при запуске, хотя многие машины используют протокол сетевого времени (Network Time Protocol, NTP) для синхронизации с сетевым сервером времени в процессе загрузки операционной системы.

ВСЕ СТАРОЕ КОГДА-ТО БЫЛО НОВЫМ

Выражение «часы реального времени» довольно неудачное. В 1980-е годы, когда такие часы появились, их действительно можно было считать сравнимыми по точности с реальным временем, но теперь они просто непригодны для использования в реальных приложениях. Такова судьба многих новинок, которые получили название за свою новизну или быстроту. Например, в Париже есть мост, который называется Понт-Нёф (Новый мост). Он был возведен в 1607 году и является самым старым мостом, сохранившимся в городе.

8254 — это программируемый чип времени, старый, как само время. Источником времени в нем является кристалл с частотой 119,318 КГц — это втрое меньше частоты цветовой поднесущей NTSC. Такое соотношение частоты уходит корнями в структуру графической системы CGA. Именно этот тип ранее использовался для

того, чтобы отсчитывать время для планировщиков операционной системы (и поддерживать процесс квантования времени). Сегодня эта функция уже выполняется в других местах или вообще не требуется.

Итак, переходим к наиболее распространенному в наши дни современному счетчику — он называется *счетчиком меток реального времени* (Time Stamp Counter, TSC). В принципе, это счетчик процессора, отслеживающий, сколько циклов процессора истекло. На первый взгляд кажется, что это идеальные часы. Но этот счетчик зависит от процессора, во время исполнения на него потенциально могут влиять энергосберегающие и другие факторы. Это означает, что разные процессоры могут рассинхронизироваться как друг с другом, так и с часами, висящими на стене.

И наконец, есть еще таймеры событий высокой точности (HPET). Они стали появляться в последнее время, чтобы инженеры могли бросить неблагодарную работу по уточнению старых часовых микросхем. HPET работает как минимум с таймером 10 МГц, то есть имеет точность до 1 мкс. Правда, такие механизмы пока доступны не во всем оборудовании и поддерживаются не во всех операционных системах.

Если после всего сказанного вам показалось, что в области процессорного времени творится какой-то беспорядок — да, вы угадали, это полный беспорядок. К счастью, платформа Java предлагает возможности, позволяющие разобраться с этой путаницей. Java скрывает зависимость от оборудования и поддержки операционной системы и позволяет работать с этими характеристиками на уровне конкретной машинной конфигурации. Но, как будет показано далее, попытки скрыть эту зависимость не всегда оказываются успешными.

6.4.2. Проблема с nanoTime()

В Java есть два основных метода для доступа к времени: `System.currentTimeMillis()` и `System.nanoTime()`. Второй используется для измерения времени с точностью выше миллисекунды. В табл. 6.1 перечислены основные различия между двумя этими методами.

Таблица 6.1. Сравнение встроенных в Java методов для доступа к времени

currentTimeMillis	**nanoTime**
Разрешение в миллисекундах	Квотируется в наносекундах
Близко соответствует времени настенных часов практически в любых обстоятельствах	Может отклоняться от хода времени на настенных часах

Если описание метода `nanoTime()` из табл. 6.1 показалось вам несколько противоречивым — хорошо. Дело в том, что в большинстве современных операционных систем этот метод получает информацию от «контрчасов» процессора — TSC.

Вывод `nanoTime()` относителен к определенному моменту времени. Это означает, что он должен использоваться для записи длительности — для этого результат вызова `nanoTime()` вычитается из более раннего результата. Следующий фрагмент

кода, взятый из ситуативного исследования, рассмотренного ниже, демонстрирует именно такой случай:

```
long t0 = System.nanoTime();
doLoop1();
long t1 = System.nanoTime();
...
long el = t1 - t0;
```

Здесь `el` — это время в наносекундах, истекшее с тех пор, как выполнялся `doLoop1()`.

Для того чтобы правильно использовать эти базовые методы при настройке производительности, нужно хорошо понимать принцип работы `nanoTime()`. В листинге 6.1 продемонстрировано максимальное наблюдаемое отклонение времени между миллисекундным таймером и нанотаймером (обычно эта информация предоставляется TSC).

Листинг 6.1. Отклонение таймера

```
private static void runWithSpin(String[] args) {
  long nowNanos = 0, startNanos = 0;
  long startMillis = System.currentTimeMillis();
  long nowMillis = startMillis;

  while (startMillis == nowMillis) {
    startNanos = System.nanoTime();
    nowMillis = System.currentTimeMillis();
  }

  startMillis = nowMillis;
  double maxDrift = 0;
  long lastMillis;

  while (true) {
    lastMillis = nowMillis;
    while (nowMillis - lastMillis < 1000) {
      nowNanos = System.nanoTime();
      nowMillis = System.currentTimeMillis();
    }

    long durationMillis = nowMillis - startMillis;
    double driftNanos = 1000000 *
  (((double)(nowNanos - startNanos)) / 1000000 - durationMillis);
    if (Math.abs(driftNanos) > maxDrift) {
      System.out.println("Now - Start = "+ durationMillis
    +" driftNanos = "+ driftNanos);
      maxDrift = Math.abs(driftNanos);
    }
  }
}
```

Выравнивание startNanos с точностью до миллисекунд

В результате выводится максимальное наблюдаемое отклонение. Оказывается, что оно в некоторой степени определяется операционной системой. Вот пример из Linux:

```
Now - Start = 1000 driftNanos = 14.99999996212864
Now - Start = 3000 driftNanos = -86.99999989403295
Now - Start = 8000 driftNanos = -89.00000011635711
Now - Start = 50000 driftNanos = -92.00000204145908
Now - Start = 67000 driftNanos = -96.0000033956021
Now - Start = 113000 driftNanos = -98.00000407267362
Now - Start = 136000 driftNanos = -98.99999713525176
Now - Start = 150000 driftNanos = -101.0000123642385
Now - Start = 497000 driftNanos = -2035.000012256205
Now - Start = 1006000 driftNanos = 20149.99999664724
Now - Start = 1219000 driftNanos = 44614.00001309812
```

Обратите внимание на разницу

А вот результат, полученный в более старой версии Solaris на том же оборудовании:

```
Now - Start = 1000 driftNanos = 65961.0000000157
Now - Start = 2000 driftNanos = 130928.0000000399
Now - Start = 3000 driftNanos = 197020.9999999497
Now - Start = 4000 driftNanos = 261826.99999981196
Now - Start = 5000 driftNanos = 328105.9999999343
Now - Start = 6000 driftNanos = 393130.99999981205
Now - Start = 7000 driftNanos = 458913.9999998224
Now - Start = 8000 driftNanos = 524811.9999996561
Now - Start = 9000 driftNanos = 590093.9999992261
Now - Start = 10000 driftNanos = 656146.9999996916
Now - Start = 11000 driftNanos = 721020.0000008626
Now - Start = 12000 driftNanos = 786994.0000000497
```

Плавное возрастание

Обратите внимание: в Solaris максимальное значение медленно повышается. А Linux подолгу работает нормально, но иногда происходят резкие скачки. Код для этого примера был очень аккуратно подобран так, чтобы в нем не создавалось никаких дополнительных потоков и даже объектов и можно было свести к минимуму вмешательство в платформы (например, если не создается никаких объектов, значит, не происходит и сборки мусора). Но даже здесь заметно влияние виртуальной машины Java.

Оказывается, что такие крупные скачки в хронометраже Linux обусловлены разницей в работе счетчиков TSC на разных процессорах. Виртуальная машина Java периодически приостанавливает работающий поток Java и перебрасывает его для исполнения на другое ядро. В таком случае разница между работой счетчиков различных процессоров становится заметна в коде приложения.

Это означает, что по прошествии достаточно длительного времени данные `nanoTime()` могут стать недостоверными. Этот таймер полезен при измерении небольших временных промежутков, однако в случае с более длительными (макро-

скопическими) отрезками его приходится периодически сверять с данными метода `currentTimeMillis()`.

Чтобы обеспечить максимально качественную оптимизацию производительности, важно обладать базовыми знаниями теории измерений, а также понимать тонкости реализации.

6.4.3. Роль времени при повышении производительности

При оптимизации производительности нужно знать, как интерпретировать результаты измерений, записанные во время выполнения кода. Это означает, что также необходимо понимать ограничения, присущие любым измерениям времени на платформе.

Точность

Временные отрезки обычно округляются до ближайшей единицы на определенной шкале. Этот феномен называется *прецизионностью* измерений. Например, время часто измеряется с прецизионностью до миллисекунд. Хронометраж характеризуется высокой прецизионностью, если при многократных замерах получается малый разброс значений вокруг среднего.

Прецизионность — это мера содержания случайных помех, зафиксированных при конкретном акте измерения. Предположим, что измерения конкретного фрагмента кода имеют нормальное распределение. В таком случае обычный способ квотирования прецизионности — это квотирование ширины 95%-ного доверительного интервала.

Правильность

Правильность измерений (в нашем случае — измерений времени) — это возможность получения значения, максимально близкого к истинному. В реальности верное значение, как правило, остается неизвестным, поэтому определить правильность бывает сложнее, чем точность.

Правильность показывает уровень систематической ошибки при измерениях. Можно иметь правильные данные, которым не хватает точности (то есть базовое значение правильное, но при разбросе значений попадаются случайные помехи). Кроме того, можно иметь неправильные результаты, отличающиеся высокой точностью.

ПОНЯТИЕ ОБ ИЗМЕРЕНИЯХ

Интервал, указанный с прецизионностью до наносекунд и равный 5945 нс, полученный от таймера, достигающего точности до 1 мкс, на самом деле находится где-то между 3945 и 7945 нс (с 95%-ной вероятностью). Остерегайтесь показателей производительности, которые кажутся чрезмерно точными, — всегда проверяйте прецизионность и точность измерений.

Гранулярность

Истинная *гранулярность* (granularity) системы — это предел частоты самого быстрого таймера. Обычно это таймер прерываний, тикающий не реже чем раз в 10 нс. Иногда данный показатель называется *отличимостью*. Отличимость (distinguishability) — это кратчайший интервал, пролегающий между двумя гарантированно автономными событиями. О них говорят, что они произошли «совсем рядом, но не одновременно».

По мере того как мы будем прорабатывать уровни операционной системы, виртуальной машины и библиотечного кода, различать такие экстремально краткие временные промежутки будет почти невозможно. В большинстве случаев информация о подобных кратчайших отрезках времени недоступна разработчику приложений.

Хронометраж, распределенный в сети

Основная часть нашего обсуждения производительности сосредоточена на центрах оптимизации в таких системах, где вся работа происходит только на одной машине (хосте). Но необходимо учитывать, что существует несколько особых проблем, возникающих при оптимизации производительности в системах, распределенных по сети. Обеспечение синхронизации и хронометража в больших сетях — дело совсем не простое, причем речь не только об Интернете. В Ethernet-сетях возникают похожие проблемы.

Подробное обсуждение хронометража, распределенного в сети, выходит за рамки этой книги. Просто не забывайте, что в принципе очень сложно получить точные значения о хронометраже рабочих процессов, охватывающих несколько машин. Кроме того, даже стандартные протоколы, например NTP, могут быть недостаточно точны для высокоточной работы.

Прежде чем перейти к обсуждению сборки мусора, рассмотрим пример, о котором мы уже упоминали выше. Речь пойдет о воздействии кэшей памяти на производительность кода.

6.4.4. Практический пример: понятие о кэш-промахах

При работе со многими фрагментами кода, обладающими высокой пропускной способностью, одним из основных факторов снижения производительности является количество кэш-промахов в кэше L1. Такие промахи сопровождают исполнение кода приложения.

В листинге 6.2 цикл проходит через массив размером 1 Мбайт и записывается время, которое потребовалось на выполнение одного из двух циклов. Первый цикл приращивается на 1 через каждые 16 записей `int[]`. В строке кэша L1 обычно находится 64 байт (а целые числа в Java на 32-битной виртуальной машине имеют ширину 4 байт). Таким образом, мы по одному разу затрагиваем каждую из строк кэша.

Учтите, что прежде, чем можно будет получить точные результаты, потребуется «разогнать» код, чтобы виртуальная машина Java смогла скомпилировать инте-

ресующие вас методы. О необходимости такого разгона мы подробнее поговорим в разделе 6.6.

Листинг 6.2. Понятие о кэш-промахах

```
public class CacheTester {
  private final int ARR_SIZE = 1 * 1024 * 1024;
  private final int[] arr = new int[ARR_SIZE];

  private void doLoop2() {
    for (int i=0; i<arr.length; i++) arr[i]++;
  }

  private void doLoop1() {
    for (int i=0; i<arr.length; i += 16) arr[i]++;
  }

  private void run() {
    for (int i=0; i<10000; i++) {
      doLoop1();
      doLoop2();
    }
    for (int i=0; i<100; i++) {
      long t0 = System.nanoTime();
      doLoop1();
      long t1 = System.nanoTime();
      doLoop2();
      long t2 = System.nanoTime();
      long el = t1 - t0;
      long el2 = t2 - t1;
      System.out.println("Loop1: "+ el +" nanos ; Loop2: "+ el2);
    }
  }

  public static void main(String[] args) {
    CacheTester ct = new CacheTester();
    ct.run();
  }
}
```

Затрагиваем каждый элемент

Затрагиваем каждую строку кэша

Разгоняем код

Вторая функция `loop2()` приращивается с каждым байтом массива, поэтому кажется, что она выполняет в 16 раз больше работы, чем `loop1()`. Но вот пример типичных результатов, полученных на ноутбуке:

```
Loop1: 634000 nanos ; Loop2: 868000
Loop1: 801000 nanos ; Loop2: 952000
Loop1: 676000 nanos ; Loop2: 930000
Loop1: 762000 nanos ; Loop2: 869000
Loop1: 706000 nanos ; Loop2: 798000
```

ФОКУС ПРИ РАБОТЕ С ПОДСИСТЕМАМИ ХРОНОМЕТРАЖА

Обратите внимание: результаты всех значений `nanos` — красивые круглые тысячи. Это означает, что лежащий в их основе системный вызов (выполняемый в конечном итоге `System.nanoTime()`) просто возвращает целое количество микросекунд. В микросекунде 1000 нс. Этот пример взят с ноутбука Mac. Несложно догадаться, что в OS X базовые системные вызовы выполняются с точностью до миллисекунд. Здесь мы имеем дело с результатами работы `gettimeofday()`.

Результаты выполнения кода показывают, что на самом деле `loop2()` не требует в 16 раз больше времени, чем `loop1()`. А значит, весь профиль производительности определяется количеством обращений к памяти. В циклах `loop1()` и `loop2()` происходит одинаковое количество считываний строк кода, и циклы, затрачиваемые собственно на преобразование данных — это всего лишь малая толика от необходимого рабочего времени.

Прежде чем двигаться дальше, вспомним самые важные моменты о системах хронометража в Java:

- в большинстве систем работает сразу несколько разных часов;
- миллисекундный хронометраж надежен и удобен для работы;
- при работе с высокоточными данными времени необходимо действовать очень аккуратно, чтобы избегать рассогласования;
- нужно знать точность и правильность измерений хронометража.

Далее мы поговорим о подсистеме сборки мусора, примеряемой на платформе. Это одна из самых важных характеристик общей картины производительности. В этой подсистеме есть великое множество настраиваемых элементов, которые могут очень пригодиться в работе программиста, занятого анализом производительности.

6.5. Сборка мусора

Автоматическое управление памятью — одна из важнейших характеристик платформы Java. До появления управляемых платформ, таких как Java и .NET, разработчик вполне мог потратить значительную часть карьеры на охоту за ошибками, возникающими из-за несовершенного управления памятью.

Но в последние годы технологии автоматического выделения памяти стали такими совершенными и надежными, что работа без них уже непредставима. Многие Java-разработчики не знают, как именно происходит управление памятью на платформе, какие функции доступны разработчику и какая оптимизация возможна в условиях ограничений, действующих во фреймворке.

Это показывает, насколько успешным оказался подход, выбранный в Java. Большинству разработчиков неизвестны подробности функционирования памяти и системы сборки мусора, так как эти знания обычно им и не нужны. Виртуальная машина отлично справляется с обработкой большинства приложений, не требуя специальной настройки. Поэтому большинство приложений вполне обходятся без настройки.

В этом разделе мы обсудим, что делать в тех случаях, когда оптимизация все-таки требуется. Мы рассмотрим теоретические основы, расскажем, как происходит управление памятью для работающего процесса Java, исследуем основы применения алгоритма «отслеживание и очистка» (mark and sweep) и обсудим два полезных инструмента — `jmap` и VisualVM. Наконец, мы поговорим о двух распространенных альтернативных сборщиках мусора: Concurrent Mark-Sweep (Параллельное отслеживание и очистка, CMS) и о более новом — Garbage First (G1).

Возможно, у вас есть серверное приложение, в котором из-за долгих пауз в работе расходуется вся память. В подразделе 6.5.3, посвященном `jmap`, мы рассмотрим простой способ, позволяющий узнать, не используют ли какие-то из ваших классов слишком много памяти. Кроме того, мы поговорим о функциях, которыми можно пользоваться для управления профилем памяти в виртуальной машине.

Итак, начнем с основ.

6.5.1. Основы

У стандартного процесса Java есть стек и куча. *Стек* — это место, где создаются локальные переменные, содержащие примитивы (но локальные переменные ссылочного типа будут указывать на память, выделяемую в куче). *Куча* — это место, в котором создаются объекты. На рис. 6.4 показано, где создаются хранилища для переменных различных типов.

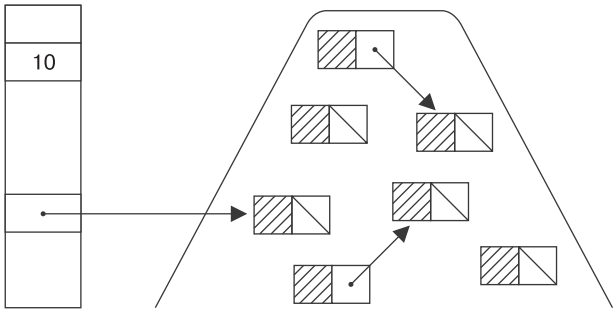


Рис. 6.4. Переменные в стеке и в куче

Обратите внимание: примитивные поля объекта тоже выделяются по адресам внутри кучи. Базовый алгоритм, применяемый платформой для восстановления и переиспользования динамической памяти (памяти кучи), не задействованной кодом приложения, называется *«отслеживание и очистка»* (mark and sweep)[1].

[1] Другие варианты перевода этого названия — «алгоритм пометок» (http://ru.wikipedia.org/wiki/Сборка_мусора), «отметить и очистить» (http://oooportal.ru/?cat=article&id=992), «отметить и подмести» (http://tinyurl.com/apvpobt), «очистка по меткам» (http://ruslanshestopal.com/actionscript3_tips/45), «пометка и очистка» (http://www.dissercat.com/content/upravlenie-pamyatyu-v-sistemakh-avtomatizirovannogo-rasparallelivaniya-programm) и др. Наш перевод взят из источника http://tinyurl.com/apb95jc. — *Примеч. перев.*

6.5.2. Отслеживание и очистка

«Отслеживание и очистка» — простейший алгоритм сборки мусора, первый из разработанных алгоритмов такого рода. Существуют и другие техники автоматического управления памятью, в частности системы подсчета ссылок, используемые в таких языках, как Perl и PHP. Пожалуй, такие механизмы проще, но они и не требуют сборки мусора.

В простейшей форме алгоритм «отслеживание и очистка» приостанавливает все работающие потоки программы и начинает работу с множества объектов, о которых точно известно, что они «живые». «Живыми» называются объекты, на которые присутствуют ссылки в каком-либо фрейме стека (ссылка может указывать на содержимое локальной переменной, параметра метода, временной переменной, также есть и менее распространенные варианты) любого пользовательского потока. Затем алгоритм проследует по дереву ссылок от живых объектов, помечая любой найденный по пути объект как «живой». По завершении этой работы все оставшиеся объекты считаются мусором и собираются (подметаются). Обратите внимание: высвобожденная таким образом память может быть возвращена виртуальной машине Java, а не операционной системе.

На платформе Java базовый метод отслеживания и очистки был усовершенствован. К нему был добавлен «сборщик мусора, учитывающий поколения объектов» (generational GC). В таком случае куча уже не является однородной областью памяти. Напротив, в динамической памяти присутствует несколько различных областей, участвующих в жизненном цикле объекта Java. В зависимости от того, насколько долго существует объект, ссылки могут указывать на различные области памяти на протяжении его жизни (как показано на рис. 6.5). На разных этапах сборки мусора объект может перемещаться от области к области.

Причина такого упорядочения такова: анализ работающих систем показывает, что объекты живут либо очень недолго, либо очень долго. Различные области динамической памяти подобраны специально для того, чтобы платформа могла пользоваться таким свойством жизненных циклов объектов.

Рис. 6.5. Области памяти: Эдем (Eden), область уцелевших (Survivor), хранилище (Tenured) и постоянное поколение памяти (Permgen)

НЕСКОЛЬКО СЛОВ О ПАУЗАХ НЕОПРЕДЕЛЕННОЙ ДЛИТЕЛЬНОСТИ

Один из часто критикуемых моментов работы Java и .NET заключается в том, что при сборке мусора методом отслеживания и очистки неизбежно возникают состояния «тотальной остановки», в которых все пользовательские потоки могут быть ненадолго остановлены. Возникают паузы, которые могут длиться неопределенно долго.

Эта проблема часто утрируется. Если говорить о серверных программах, то существует совсем мало приложений, для которых критичны такие паузы, возникающие в Java. Иногда изобретаются сложные приемы, помогающие исключать паузы, либо применяется полная уборка памяти. Этого следует избегать, только если тщательный анализ не покажет, что существуют реальные проблемы, решаемые благодаря такой полной уборке.

Области памяти

В виртуальной машине Java предусмотрены различные области памяти, используемые для хранения объектов в ходе их естественного жизненного цикла.

- *Эдем* (Eden) — это область динамической памяти, в которой изначально выделяются все объекты. Многие объекты никогда не покидают этой области памяти.
- *Область уцелевших* (Survivor) — как правило, в памяти присутствует две области уцелевших. Или же можно считать, что область уцелевших обычно делится пополам. Именно в нее попадают объекты, пережившие «изгнание из Эдема» (отсюда и ее название). Иногда два этих пространства называются *Из* (From) и *В* (To). По причинам, рассмотренным ниже, одна из областей уцелевших пустует, если только не происходит процесс сбора.
- *Хранилище* (Tenured) — это область (также называемая «старым поколением»), где оказываются уцелевшие объекты, которые признаются «достаточно старыми» (таким образом, они покидают область уцелевших). Хранилище не очищается в ходе молодой сборки.
- *Постоянное поколение памяти* (PermGen) — здесь выделяется место для внутренних структур, например для определений классов. Строго говоря, постоянное поколение не входит в состав динамической памяти, обычные объекты сюда никогда не попадают.

Как упоминалось выше, эти области памяти также по-разному участвуют в сборке. Сборка бывает двух типов: молодая и полная.

Молодые сборки

В ходе *молодой сборки* (young collection) система пытается очистить только области с молодыми объектами — Эдем и область уцелевших. Этот процесс довольно прост.

1. Все живые молодые объекты, найденные на этапе отслеживания, перемещаются в следующие места:
   - объекты, которые уже достаточно стары (пережившие достаточное количество предыдущих циклов сборки мусора), попадают в хранилище;
   - все остальные молодые «живые» объекты отправляются в пустую область уцелевших.

2. После этого Эдем и только что очищенная область уцелевших могут быть перезаписаны и переиспользованы, поскольку в них больше нет ничего, кроме мусора.

Молодая сборка начинается после того, как Эдем оказывается целиком заполнен. Обратите внимание: на этапе отслеживания требуется обойти весь граф живых объектов. Это означает, что если у молодого объекта есть ссылка на объект из хранилища, то ссылки, удерживаемые объектом из хранилища, также должны быть просмотрены и отслежены. В противном случае может возникнуть ситуация, в которой объект из хранилища удерживает ссылку на объект из Эдема, но больше на этот объект из Эдема нет никаких ссылок. Если не произвести полного обхода на этапе отслеживания, то объект из Эдема никогда не удастся увидеть и, соответственно, не получится правильно обработать.

Полные сборки

Если молодая сборка не может перевести объект в хранилище (из-за недостатка пространства), то запускается *полная сборка* (full collection). В зависимости от того, какой механизм сборки применяется при работе со старым поколением, может потребоваться перемещать объекты в старом поколении. Это позволяет гарантировать, что в старом поколении хватает места, чтобы при необходимости выделить крупный объект. Такой процесс называется *уплотнением* (compacting).

Безопасные состояния

Сборка мусора не может происходить без хотя бы самых кратких общих остановок работы всех потоков приложения. Но поток нельзя приостановить для сборки мусора в произвольный момент. Напротив, существуют специальные периоды, в которые может происходить сборка мусора, — их называют *безопасными состояниями* (safepoint). Типичным примером безопасного состояния является точка вызова метода (call site), но бывают и другие безопасные состояния. Чтобы могла произойти сборка мусора, все потоки приложения должны быть остановлены в безопасном состоянии.

Сделаем небольшой перерыв и познакомимся с простым инструментом, `jmap`, который помогает понять, как используется память в запущенных приложениях, а также разобраться, куда попадает вся эта память. Далее в главе мы познакомимся с более сложным подобным инструментом, оснащенным графическим интерфейсом. Но многие проблемы можно классифицировать с помощью очень простых инструментов командной строки. Поэтому вы определенно должны уметь обращаться с такими инструментами (а не переходить сразу к поиску средства с графическим интерфейсом).

6.5.3. jmap

В комплекте со стандартной виртуальной машиной Java от Oracle поставляются простые инструменты, позволяющие заглянуть вглубь работающих процессов. Самый незатейливый из них — `jmap` — отображает карты распределения памяти работающих процессов в памяти Java (он также может работать с файлом ядра Java

и даже подключаться к удаленному отладочному серверу). Вернемся к нашему примеру с серверным приложением для электронного магазина и исследуем это приложение в ходе работы с помощью jmap.

Стандартный вывод (по умолчанию)

В простейшей форме jmap демонстрирует нативные библиотеки, связанные с процессом. Обычно это не слишком интересно, если только у вас в приложении нет большого объема JNI-кода. Но мы все равно покажем такой вариант, чтобы вы не запутались, если когда-нибудь забудете указать ту перспективу, которую хотите отображать в jmap:

```
$ jmap 19306
Attaching to process ID 19306, please wait...
Debugger attached successfully.
Server compiler detected.
JVM version is 20.0-b11
0x08048000   46K     /usr/local/java/sunjdk/1.6.0_25/bin/java
0x55555000   108K    /lib/ld-2.3.4.so
... some entries omitted
0x563e8000   535K    /lib/libnss_db.so.2.0.0
0x7ed18000   94K     /usr/local/java/sunjdk/1.6.0_25/jre/lib/i386/libnet.so
0x80cf3000   2102K   /usr/local/kerberos/mitkrb5/1.4.4/lib/
libgss_all.so.3.1
0x80dcf000   1440K   /usr/local/kerberos/mitkrb5/1.4.4/lib/libkrb5.so.3.2
```

Гораздо более полезны переключатели -heap и -histo, о которых мы поговорим ниже.

Вывод кучи

Параметр -heap дает мгновенный снимок кучи по состоянию на тот момент, когда вы его используете. В выводе -heap вы увидите базовые параметры, определяющие структуру динамической памяти процессов Java.

Размер кучи равен общему размеру молодого и старого поколений плюс размер постоянного поколения памяти. Но в составе молодого поколения у вас есть Эдем и область уцелевших, а мы еще не рассказали, как соотносятся размеры этих областей. Число, определяющее отношение этих областей, называется *коэффициентом выживаемости* (survivor ratio).

Рассмотрим пример вывода. Здесь вы видите Эдем, области уцелевших (помеченные From и To) и хранилище (Old Generation), а также сопутствующую информацию:

```
$ jmap -heap 22186
Attaching to process ID 22186, please wait...
Debugger attached successfully.
Server compiler detected.
JVM version is 20.0-b11

using thread-local object allocation.
```

```
Parallel GC with 13 thread(s)

Heap Configuration:
  MinHeapFreeRatio = 40
  MaxHeapFreeRatio = 70
  MaxHeapSize = 536870912 (512.0MB)
  NewSize = 1048576 (1.0MB)
  MaxNewSize = 4294901760 (4095.9375MB)
  OldSize = 4194304 (4.0MB)
  NewRatio = 2
  SurvivorRatio = 8   // Эдем = (From +To) * коэффициент выживаемости
  PermSize = 16777216 (16.0MB)
  MaxPermSize = 67108864 (64.0MB)
Heap Usage:
PS Young Generation
Eden Space:
  capacity = 163774464 (156.1875MB)  // Эдем = (From +To) * коэффициент выживаемо-
сти
  used = 58652576 (55.935455322265625MB)
  free = 105121888 (100.25204467773438MB)
  35.81301661289516% used
From Space:
  capacity = 7012352 (6.6875MB)  // Эдем = (From +To) * коэффициент выживаемости
  used = 4144688 (3.9526824951171875MB)
  free = 2867664 (2.7348175048828125MB)
  59.10553263726636% used
To Space:
  capacity = 7274496 (6.9375MB)  // Эдем = (From +To) * коэффициент выживаемости
  used = 0 (0.0MB)
  free = 7274496 (6.9375MB)
  0.0% used  // область "To" в настоящий момент пуста
PS Old Generation
  capacity = 89522176 (85.375MB)
  used = 6158272 (5.87298583984375MB)
  free = 83363904 (79.50201416015625MB)
  6.87904637170571% used
PS Perm Generation
  capacity = 30146560 (28.75MB)
  used = 30086280 (28.69251251220703MB)
  free = 60280 (0.0574874877929687 5MB)
  99.80004352072011% used
```

Конечно, такая базовая структура областей может быть очень полезна, но в этом выводе вы не видите, что происходит в куче. Имея возможность видеть, какие объекты наполняют память, вы можете получить ценные подсказки о том, что вообще сейчас происходит в памяти. К счастью, в `jmap` присутствует и режим гистограммы, в котором вы получаете удобную статистику именно такого рода.

Режим гистограммы

В режиме гистограммы мы видим, какой объем памяти в системе занят экземплярами каждого типа (а также просматриваем некоторые внутрисистемные записи). Типы перечисляются в порядке убывания занимаемой ими памяти, чтобы было легко заметить самых крупных пожирателей памяти.

Разумеется, если вся память уходит на обслуживание классов платформы и фреймворка, то вы мало чем сможете помочь. Но если какой-то из классов явственно выделяется по потреблению памяти, то вам будет гораздо удобнее как-то наладить ее использование.

Небольшое предупреждение: `jmap` применяет для типов внутренние имена, перечисленные в главе 5. Например, массив символов записывается как `[C`, а массивы объектов классов будут показаны как `[Ljava.lang.Class;`.

```
$ jmap -histo 22186 | head -30

num  #instances      #bytes   class name
----------------------------------------------
  1:     452779    31712472   [C
  2:      76877    14924304   [B
  3:      20817    12188728   [Ljava.lang.Object;
  4:       2520    10547976   com.company.cache.Cache$AccountInfo
  5:     439499     9145560   java.lang.String
  6:      64466     7519800   [I
  7:      64466     5677912   <constMethodKlass>
  8:      96840     4333424   <methodKlass>
  9:       6990     3384504   <symbolKlass>
 10:       6990     2944272   <constantPoolKlass>
 11:       4991     1855272   <instanceKlassKlass>
 12:      25980     1247040   <constantPoolCacheKlass>
 13:      17250     1209984   java.nio.HeapCharBuffer
 14:      13515     1173568   [Ljava.util.HashMap$Entry;
 15:       9733      778640   java.lang.reflect.Method
 16:      17842      713680   java.nio.HeapByteBuffer
 17:       7433      713568   java.lang.Class
 18:      10771      678664   [S
 19:       1543      489368   <methodDataKlass>
 20:      10620      456136   [[I
 21:      18285      438840   java.util.HashMap$Entry
 22:       9985      399400   java.util.HashMap
 23:      13725      329400   java.util.Hashtable$Entry
 24:       9839      314848   java.util.LinkedHashMap$Entry
 25:       9793      249272   [Ljava.lang.String;
 26:      11927      241192   [Ljava.lang.Class;
 27:       6903      220896   java.lang.ref.SoftReference
// Внутренние объекты виртуальной машины и информация о типах
```

Мы удалили весь последующий вывод, ограничившись парой десятков строк, так как в гистограмме может быть *очень много* информации. Возможно, вам

понадобится grep или другие инструменты, чтобы просматривать вывод гистограммы и находить интересующие вас детали.

В данном примере вы видите, что значительную часть памяти потребляют такие записи, как [C. Массивы символьных данных часто находятся внутри объектов String (где они заключают в себе содержимое строки), что совсем не удивительно — в большинстве программ на Java строки очень распространены. Но из гистограммы можно узнать и другие довольно интересные вещи, две из которых мы сейчас и рассмотрим.

Записи Cache$AccountInfo — это единственные классы приложения, которые попадаются среди верхних записей — все остальное в верхней части вывода относится к типам платформы или фреймворка. Таким образом, Cache$AccountInfo — это самый важный тип, над которым разработчику нужен полный контроль. Объекты AccountInfo занимают много места: 10,5 Мбайт примерно на 2500 записей или по 4 Кбайт на запись. Согласитесь, это как-то много для учетной информации.

Эта информация может быть очень полезна. Вы уже узнали, какая часть вашего кода потребляет наибольшее количество памяти. Предположим, теперь к вам обращается начальник и говорит, что ожидается крупная акция с промопродажами. Поэтому в ближайший месяц в системе будет находиться вдесятеро больше пользователей, чем обычно. Вы знаете, что это окажет серьезную нагрузку, учитывая, насколько тяжелы объекты AccountInfo. Конечно, вы немного взволнованы, но вы хотя бы уже приступили к анализу проблемы.

Информация от jmap может применяться в качестве исходных данных, чтобы помочь управлять процессом принятия решений о том, как справляться с потенциальными проблемами. Возможно, вам придется разделить кэш учетных записей, попробовать уменьшить объем информации, содержащейся в типе, или купить для сервера дополнительную оперативную память. Прежде чем вносить какие-либо изменения, потребуется провести очень подробный анализ, но вы хотя бы знаете, с чего начать.

Чтобы разобраться еще с одной интересной вещью, вернемся к гистограмме. Но на этот раз укажем -histo:live. Так мы прикажем jmap обрабатывать только «живые» объекты, а не всю кучу (по умолчанию jmap работает сразу со всей информацией, в том числе с мусором, пока остающимся в памяти и еще не собранным). Вот как это выглядит:

```
$ jmap -histo:live 22186 | head -7

num   #instances    #bytes   class name
----------------------------------------------
  1:        2520   10547976  com.company.cache.Cache$AccountInfo
  2:       32796    4919800  [I
  3:        5392    4237628  [Ljava.lang.Object;
  4:      141491    2187368  [C
```

Только посмотрите, как изменилась картина. Объем символьных данных уменьшился с 31 Мбайт примерно до 2 Мбайт, а около двух третей объектов String, которые вы видели при первом запуске, оказались мусором, ожидающим сборки. Но все объекты с учетной информацией «живы», что еще раз доказывает, что подобная информация потребляет много памяти.

Работая с такими режимами jmap, всегда будьте внимательны. Когда вы выполняете такие операции, виртуальная машина Java продолжает работать (а если вам не повезло — она даже могла успеть провести сборку мусора между вашими мгновенными снимками). Поэтому всегда следует делать несколько запусков, особенно если система выдает странные результаты или если они слишком хороши, чтобы быть правдой.

Создание файлов офлайнового дампа

Последний режим работы с jmap, который мы здесь рассмотрим, связан с его возможностью создавать файл дампа, вот так:

```
jmap -dump:live,format=b,file=heap.hprof 19306
```

Дампы готовы для офлайнового анализа, как в самом jmap когда-нибудь попозже, так и в инструменте jhat от Oracle (полное его название — инструмент для анализа кучи Java, Java Heap Analysis Tool). К сожалению, мы не можем здесь подробно его обсудить.

Благодаря jmap можно начать обзор некоторых базовых настроек и потребления памяти в вашем приложении. Но для настройки производительности часто требуется получить более полный контроль над подсистемой сборки мусора. Обычно это делается с помощью параметров командной строки. Поэтому рассмотрим несколько параметров, которыми можно пользоваться для управления виртуальной машиной Java и изменения режимов ее работы, чтобы они лучше отвечали запросам вашего приложения.

6.5.4. Полезные переключатели виртуальной машины Java

В составе виртуальной машины Java есть множество полезных параметров-переключателей (не менее сотни), которые можно использовать для настройки многих характеристик работы машины во время исполнения. В этом разделе мы рассмотрим некоторые переключатели, применяемые при сборке мусора. Ниже поговорим о других переключателях.

НЕСТАНДАРТНЫЕ ПЕРЕКЛЮЧАТЕЛИ ВИРТУАЛЬНОЙ МАШИНЫ JAVA

Если переключатель начинается с -X:, это означает, что он нестандартный. Возможно, такой переключатель не удастся портировать между различными реализациями виртуальной машины Java.

Если переключатель начинается с -XX:, то это расширенный переключатель (extended switch), не рекомендуемый для нецелевого использования. Многие переключатели, связанные с обеспечением производительности, являются расширенными.

Некоторые переключатели по сути являются булевыми. Чтобы их включить или выключить, перед ними нужно указать + или - соответственно. Другие переключатели принимают параметр, как, например, -XX:CompileThreshold=1000 (здесь мы задаем, сколько раз метод должен быть вызван, прежде чем он будет рассмотрен для динамической компиляции; это значение равно 1000). А другие переключатели, в частности многие из стандартных, ничего не принимают.

В табл. 6.2 перечислены основные переключатели, применяемые при сборке мусора. Если у переключателя есть значение по умолчанию, оно также дается в таблице.

Таблица 6.2. Основные переключатели для сборки мусора

Переключатель	Эффект
-Xms<размер в Мбайт>m	Исходный размер кучи (по умолчанию 2 Мбайт)
-Xmx<размер в Мбайт>m	Максимальный размер кучи (по умолчанию 64 Мбайт)
-Xmn<размер в Мбайт>m	Размер молодого поколения в куче
-XX:-DisableExplicitGC	Предотвращает какой-либо эффект от вызова System.gc()

Очень часто размер `-Xms` задается равным размеру `-Xmx`. В таком случае процесс проработает именно такой объем кучи, и во время исполнения не потребуется менять размеры (а такие изменения могут приводить к неожиданному замедлению работы).

Последний переключатель из списка выводит в журнал стандартную информацию о сборщике мусора. Об интерпретации этой информации мы поговорим в следующем разделе.

6.5.5. Чтение журналов сборщика мусора

Чтобы извлечь из сборки мусора максимальную пользу, часто бывает полезно посмотреть, что делает подсистема. Наряду с базовым флагом `verbose:gc` существует и несколько других переключателей, которые управляют выводимой информацией.

Иногда чтение журналов сборщика мусора оказывается непростой задачей — может показаться, что вы просто утопаете в информации. Как будет продемонстрировано в следующем подразделе о VisualVM, вам может очень пригодиться инструмент с графическим интерфейсом, позволяющий визуализировать поведение виртуальной машины. Тем не менее важно уметь читать форматы журналов и знать основные переключатели, влияющие на работу сборщика мусора, так как под рукой может и не оказаться такого графического инструмента. Некоторые наиболее полезные переключатели, применяемые с журналами сборщика мусора, приведены в табл. 6.3.

Таблица 6.3. Дополнительные переключатели для получения расширенной информации

Переключатель	Эффект
-XX:+PrintGCDetails	Дает расширенные подробности о сборщике мусора
-XX:+PrintGCDateStamps	Сопровождает операции сборщика мусора метками времени
-XX:+PrintGCApplicationConcurrentTime	Время, затраченное на сборщик мусора в условиях, когда потоки приложения продолжают работать

Вместе эти переключатели дают примерно такой вывод:

```
6.580: [GC [PSYoungGen: 486784K->7667K(499648K)]
1292752K->813636K(1400768K), 0.0244970 secs]
```

Разберемся, что означает приведенный выше фрагмент:

```
<time>: [GC [<collector name>: <occupancy at start>
➥ -> <occupancy at end>(<total size>)] <full heap occupancy at start>
➥ -> <full heap occupancy at end>(<total heap size>), <pause time> secs
```

В первом поле видим время, в которое началась сборка мусора. Оно указано в секундах с момента запуска виртуальной машины Java. Затем идет имя сборщика (`PSYoungGen`), который подбирает молодое поколение. Далее указана память, используемая до и после сборки молодого поколения, а также общий размер молодого поколения. Наконец, идут несколько полей для полной кучи.

Наряду с логирующими флагами сборщика мусора есть и такой флаг, который может вас немного запутать, поэтому он требует объяснений. Флаг `-XX:+PrintGCApplicationStoppedTime` дает в журнале вот такие строки:

```
Application time: 0.9279047 seconds
Total time for which application threads were stopped: 0.0007529 seconds
Application time: 0.0085059 seconds
Total time for which application threads were stopped: 0.0002074 seconds
Application time: 0.0021318 seconds
```

Эта информация не обязательно описывает общую длительность сборки мусора. На самом деле здесь мы видим, насколько долго были остановлены потоки во время операций, которые начались в безопасном состоянии. Среди этих операций не только сборка мусора, но и другие виды работ, начинаемые в безопасном состоянии (например, операции привязанных блокировок Java 6).

Конечно, вся эта информация полезна для логировании и анализа, проводимого постфактум, но визуализировать ее не так просто. Поэтому многие разработчики и предпочитают пользоваться каким-нибудь инструментом с графическим пользовательским интерфейсом для выполнения первичного анализа. К счастью, виртуальная машина HotSpot (из стандартной установки Oracle, которую мы подробно опишем ниже) поставляется с очень удобным инструментом такого рода.

6.5.6. Визуализация использования памяти с помощью VisualVM

VisualVM — это инструмент для визуализации, поставляемый в комплекте со стандартной виртуальной машиной Java от Oracle. Он имеет модульную архитектуру и в стандартной конфигурации может использоваться как удобная замена для уже устаревающей программы JConsole.

На рис. 6.6 показан стандартный экран со сводными показателями программы VisualVM. Этот экран вы увидите, если запустите VisualVM и подключите ее

к приложению, работающему на вашем локальном компьютере (VisualVM может подключаться и к удаленным приложениям, но при работе в сети доступен не весь набор функций).

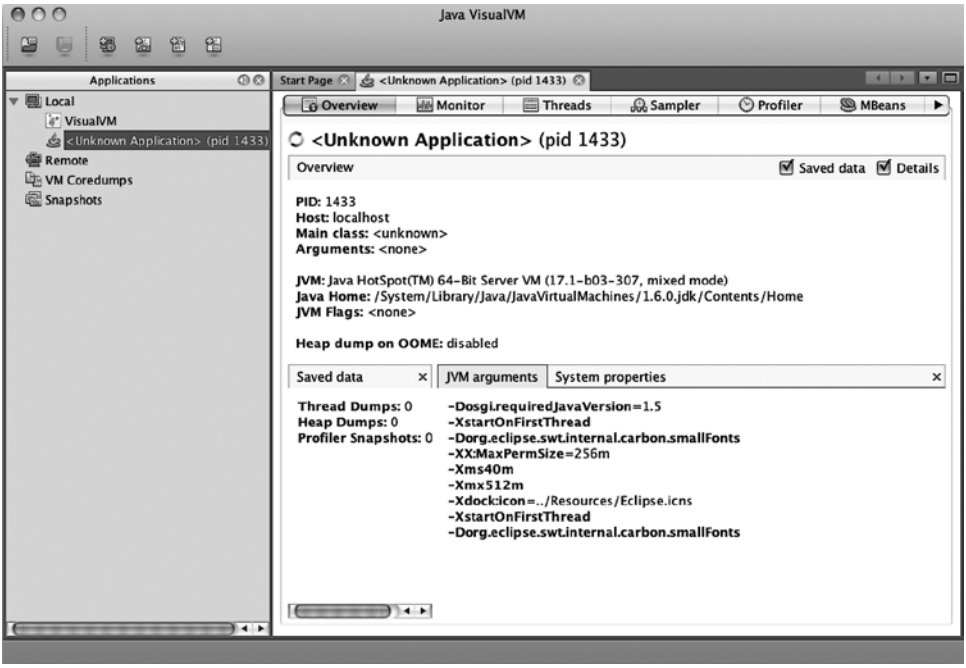


Рис. 6.6. Экран со сводными показателями программы VisualVM

Здесь вы видите, как VisualVM профилирует экземпляр Eclipse, работающий на MacBook Pro. Именно эту установку мы использовали для подготовки примеров кода к нашей книге.

В верхней части правой половины окна расположено несколько полезных вкладок. Мы работаем с подключаемыми модулями (плагинами), которые называются Extensions, Sampler, JConsole, MBeans и VisualVM. Это отличный инструментарий, хорошо помогающий разобраться в некоторых динамических характеристиках среды времени исполнения Java. Рекомендуем установить эти модули в VisualVM прежде, чем приступать к серьезной работе с программой.

На рис. 6.7 вы видите «пилообразный» вариант использования памяти. Это классическая визуализация того, как задействуется память на платформе Java. Здесь мы видим объекты, выделенные в Эдеме, использованные, а потом собранные при уборке молодого поколения.

После каждой молодой сборки объем используемой памяти вновь падает до базового уровня, который соответствует общей нагрузке, возникающей при применении области уцелевших и хранилища. По нему можно отслеживать состояние процесса Java с течением времени. Если базовый уровень остается стабилен (или даже снижается со временем), пока процесс продолжает работать, — это значит, что память используется исключительно разумно.

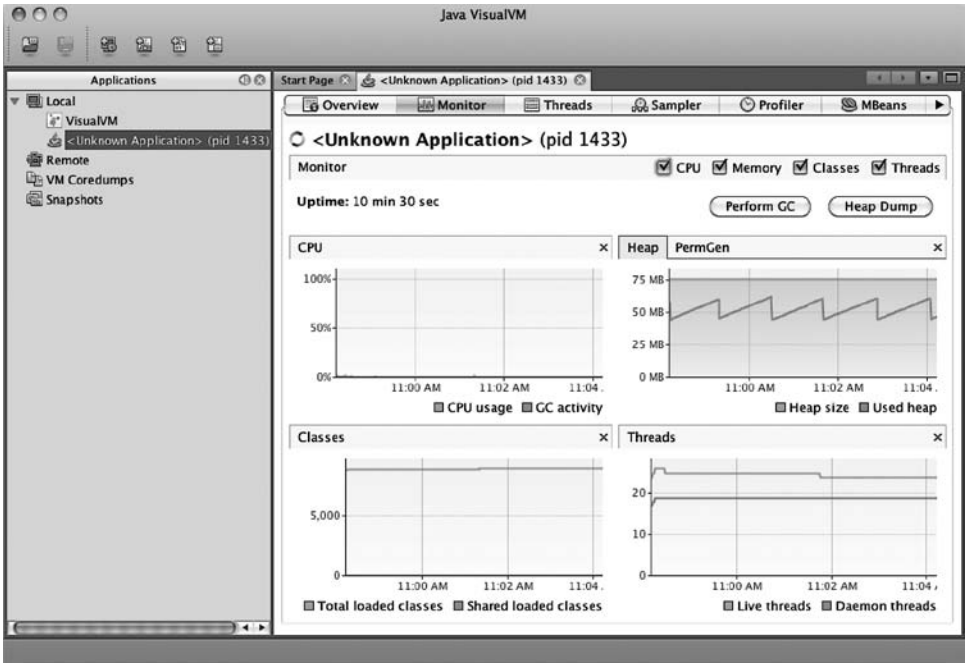


Рис. 6.7. Экран обзора в VisualVM

Если базовый уровень постепенно повышается, это не обязательно означает, что что-то пошло неправильно. Просто некоторые объекты живут достаточно долго и попадают в хранилище. В таком случае рано или поздно начнется полная сборка. При полной сборке на экране образуется второй пилообразный паттерн. В нем также будет наблюдаться свой базовый уровень использования памяти, соответствующий состоянию, в котором память содержит лишь действительно «живые» объекты. Если базовый уровень для полных сборок остается стабильным достаточно долго, то процесс вряд ли израсходует всю доступную ему память.

Один из ключевых моментов в данной визуализации заключается в том, что градиент уклона зубца на графике показывает скорость, с которой процесс расходует молодые области памяти (как правило, Эдем). Если требуется уменьшить частотность молодых сборок, то это означает, что нужно попытаться сгладить крутизну зубца графика.

Другой способ визуализации текущего потребления памяти показан на рис. 6.8. Здесь вы видите Эдем, две области уцелевших (S0 и S1), старое поколение и постоянное поколение памяти. Когда приложение работает, можно наблюдать, как изменяются размеры поколений. В частности, по окончании молодой сборки видно, как сжимается Эдем, а области уцелевших меняются ролями.

Исследуя систему памяти и другие характеристики среды времени исполнения, вы сможете лучше понять, как работает ваш код. Это, в свою очередь, показывает, как службы, предоставляемые виртуальной машиной, влияют на производительность. Поэтому вам определенно не помешает поэкспериментировать с VisualVM, особенно в комбинации с такими переключателями, как `Xmx` и `Xms`.

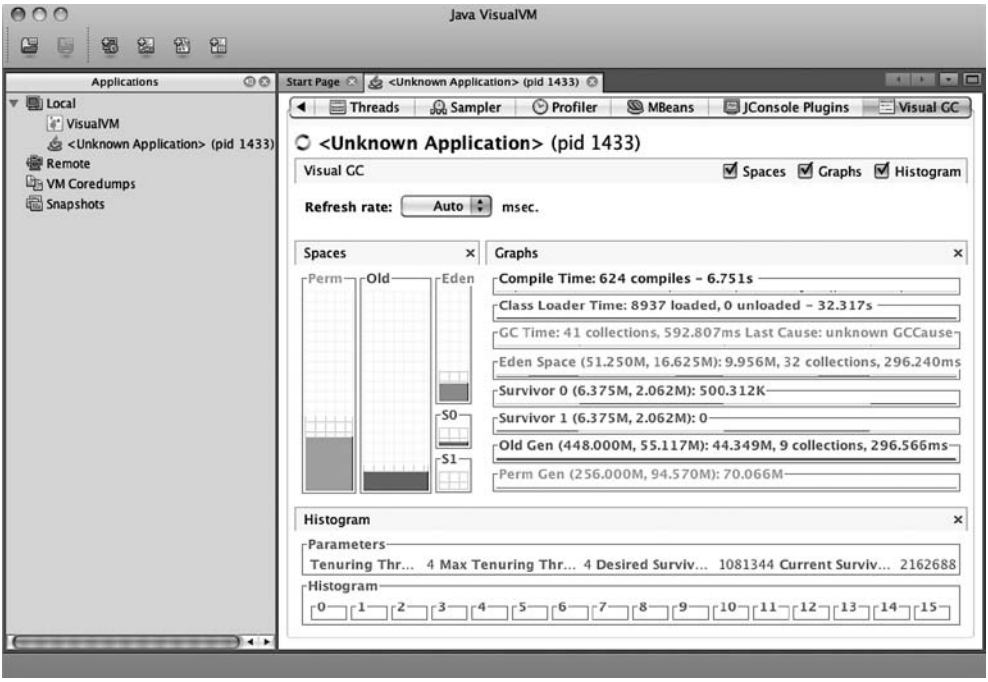


Рис. 6.8. Визуальный плагин VisualVM, демонстрирующий сборку мусора

Перейдем к следующему подразделу, где поговорим о новой технологии виртуальной машины Java, позволяющей автоматически уменьшать объем динамической памяти, задействуемой во время исполнения.

6.5.7. Анализ локальности

Этот подраздел в основном справочный — он описывает изменения, недавно появившиеся в виртуальной машине Java. Программист не может напрямую влиять на эти изменения или контролировать их. В последних версиях Java оптимизация активизируется по умолчанию. Эти изменения можно проиллюстрировать многочисленными наглядными и описательными примерами. Если вы хотите сами посмотреть, какие уловки применяются в виртуальной машине Java для улучшения производительности, то читайте далее. В противном случае можете переходить к подразделу 6.5.8, где рассказывается о параллельном сборщике.

На первый взгляд, *анализ локальности* (escape analysis) кажется довольно нетривиальной идеей. Ее суть заключается в том, чтобы проанализировать метод и посмотреть, какие локальные переменные (ссылочного типа) используются только внутри метода. Так можно определить, какие переменные не передаются другим методам или не возвращаются от актуального метода.

После этого виртуальная машина Java может создать объект в стеке внутри фрейма, относящегося к актуальному методу, а не использовать еще немного динамической памяти. Так можно улучшить производительность, уменьшив количе-

ство молодых сборок, которые требуется выполнить вашей программе. Это показано на рис. 6.9.

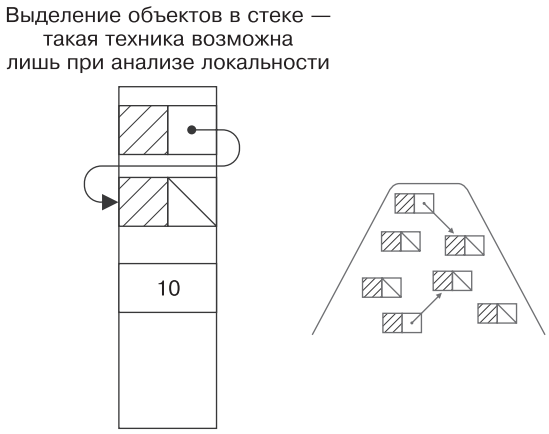


Рис. 6.9. Анализ локальности позволяет избегать выделения объектов в куче

Это означает, что можно избежать выделения в куче, так как когда происходит возврат актуального метода, автоматически высвобождается память, которая использовалась для содержания локальной переменной. В этом случае, если выделение происходит не в куче, то выделенные таким образом переменные не переходят в категорию мусора и никогда не требуют сборки.

Анализ локальности — это новый подход, позволяющий снизить интенсивность сборки мусора в виртуальной машине Java. Это может радикально повлиять на количество молодых сборок, которые происходят в ходе выполнения процесса. По опыту работы с этой возможностью можно утверждать, что, в принципе, такой подход позволяет улучшить общую производительность на несколько процентов. Это не так много, но и не помешает, особенно если ваши процессы активно собирают мусор.

В Java 6.23 и выше анализ локальности активизирован по умолчанию, поэтому в более новых версиях Java этот бонус скорости достается даром.

Теперь рассмотрим еще одну функцию, которая может значительно повлиять на ваш код, — выбор стратегии сборки. Начнем с классического высокопроизводительного варианта (параллельное отслеживание и очистка), а потом поговорим о более новом сборщике, который называется Garbage First.

По многим причинам можно предпочесть высокопроизводительный сборщик. Если паузы, затрачиваемые в приложении на сборку мусора, станут короче, это пойдет на пользу программе. Поэтому для такого сокращения пауз может быть целесообразно запускать дополнительные потоки (и немного сильнее нагружать процессор). Возможно также, что вы пожелаете сами управлять тем, как часто программа прерывается на сборку мусора. Вместе с основным сборщиком вы можете использовать переключатели, позволяющие переводить платформу на другие стратегии сборки. В следующих разделах мы рассмотрим два сборщика, обеспечивающих такие возможности.

6.5.8. Параллельное отслеживание и очистка

Сборщик для параллельного отслеживания и очистки (Concurrent Mark-Sweep, CMS) — это высокопроизводительный сборщик мусора, рекомендованный для использования в Java 5 и на протяжении большей части существования Java 6. Он задействуется с помощью комбинации переключателей, приведенных в табл. 6.4.

Таблица 6.4. Переключатели для сборщика CMS

Переключатель	Эффект
-XX:+UseConcMarkSweepGC	Переключается на CMS-сборку
-XX:+CMSIncrementalMode	Инкрементный режим (обычно необходим)
-XX:+CMSIncrementalPacing	Инкрементный режим (обычно необходим)
-XX:+UseParNewGC	Параллельное выполнение молодых сборок
-XX:ParallelGCThreads=<N>	Количество потоков, которые следует использовать при сборке мусора

Эти переключатели переопределяют стандартные настройки сборки мусора, конфигурируя сборщик мусора CMS с *N* параллельных потоков для сборки. Такой сборщик будет выполнять максимальное количество работы по сборке, реализуемое при параллельной обработке в данной системе.

Как работает такой параллельный подход? Существует три ключевых фактора алгоритма отслеживания и очистки, которые важны в данном случае:

- неизбежно некоторое количество пауз, когда приостанавливаются все работающие потоки;
- подсистема сборки мусора ни в коем случае не должна собирать «живые» объекты — в противном случае мы рискуем аварийным завершением виртуальной машины Java или даже чем-нибудь похуже;
- вы можете гарантированно собрать весь мусор лишь в том случае, если на время сборки все потоки приложения будут остановлены.

При работе CMS активно использует именно последний пункт. Он делает две очень краткие паузы с полной остановкой всех потоков, а также работает параллельно с другими потоками приложения в оставшуюся часть цикла по сборке мусора. Таким образом, CMS избегает получения «ложноотрицательных» результатов, но, так как в описанной ситуации возникают условия гонки, может упустить часть мусора (такой упущенный мусор будет собран в следующем цикле).

Кроме того, CMS при работе приходится вести более сложный учет того, что является и что не является мусором. Приходится пойти на эти дополнительные издержки, чтобы работа могла протекать в основном без остановки потоков приложения. Обычно такая работа лучше выполняется на машинах с большим количеством ядер, где возможны более частые и краткие паузы. В журнале получаются примерно такие строки:

```
2010-11-17T15:47:45.692+0000: 90434.570: [GC 90434.570:
[ParNew: 14777K->14777K(14784K), 0.0000595 secs]90434.570:
```

```
[CMS: 114688K->114688K(114688K), 0.9083496 secs] 129465K->117349K(129472K),
[CMS Perm : 49636K->49634K(65536K)] icms_dc=100 , 0.9086004 secs]
[Times: user=0.91 sys=0.00, real=0.91 secs]
```

Эти строки напоминают базовый фрагмент журнала для сборки мусора, рассмотренный в подразделе 6.4.4, но здесь есть дополнительные разделы, относящиеся к сборщикам CMS и CMS Perm.

В последние годы у CMS появился новый серьезный конкурент, самый лучший высокопроизводительный сборщик мусора — Garbage First (G1). Рассмотрим этот новаторский инструмент, выясним, в чем заключается его функциональная новизна и чем он принципиально отличается от всех существующих сборщиков мусора для Java.

6.5.9. G1 — новый сборщик мусора для Java

G1 — это новейший сборщик мусора для платформы Java. Первоначально планировалось, что он войдет в состав Java 7, но ранняя версия этого инструмента была доступна уже в технических релизах Java 6. Полнофункциональный статус G1 приобрел в Java 7. В установках Java 6 такой сборщик используется нечасто, но по мере распространения Java 7 ожидается, что G1 станет стандартным вариантом для высокопроизводительных (а возможно, и для любых) приложений.

Центральная идея, лежащая в основе G1, называется *«желательная пауза»* (pause goal). Этот параметр показывает, на какое время программа может прервать работу во время исполнения ради сборки мусора (например, на 20 мс один раз в 5 минут). G1 будет делать все возможное, чтобы работать с заданными желательными паузами. Это коренным образом отличает его от всех сборщиков мусора, с которыми мы сталкивались раньше. Вооружившись G1, разработчик может гораздо полнее контролировать процесс сборки мусора.

G1 не из тех сборщиков мусора, которые работают с учетом поколений (хотя в нем и используется алгоритм отслеживания и очистки). При работе G1 делит кучу на области равного размера (например, по 1 Мбайт), не различая молодых и старых областей. Во время паузы объекты эвакуируются в другую область (подобно тому как объекты из Эдема переходят в область уцелевших), а освобожденная область помещается в список пустых областей. Такое новое разделение кучи на области одинаковых размеров проиллюстрировано на рис. 6.10.

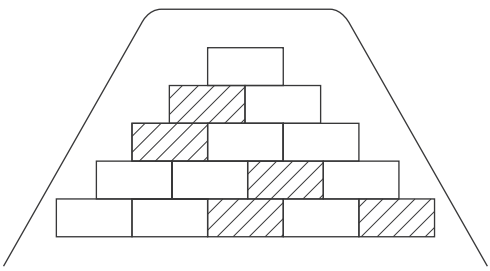


Рис. 6.10. Разделение кучи в сборщике G1

Подобное изменение стратегии сборки мусора позволяет платформе собирать статистическую информацию о том, сколько времени (в среднем) уходит на сборку мусора в конкретной области. Зная это, вы сможете задавать обоснованную целевую паузу. G1 соберет мусор из стольких областей, сколько успеет обработать за заданную паузу (хотя возможны и превышения отпущенного времени, если на уборку последней области уходит больше времени, чем ожидалось).

Чтобы приступить к работе с G1, используйте настройки, перечисленные в табл. 6.5.

Таблица 6.5. Флаги для сборщика мусора G1

Переключатель	Эффект
-XX:+UseG1GC	Включает сборку G1
-XX:MaxGCPauseMillis=50	Указывает G1, что в ходе отдельно взятой сборки необходимо избегать пауз дольше 50 мс
-XX:GCPauseIntervalMillis=200	Указывает G1, что между сборками мусора должно проходить не менее 200 мс

Переключатели можно комбинировать: например, установить максимальную желательную паузу 50 мс и указать, что паузы должны отстоять друг от друга не менее чем на 200 мс. Разумеется, у системы G1 есть предел скорости. Пауза должна быть достаточно длительной, чтобы G1 успел убрать мусор. Если задать желательную паузу длительностью 1 мс один раз в 100 лет, это будет недостижимое и неграмотное требование.

G1 кажется очень многообещающим сборщиком при самых разных нагрузках и типах приложений. Если вы дошли до этапа, на котором возникает необходимость настройки сборки мусора в вашей программе, обязательно обратите внимание на этот инструмент.

В следующем разделе мы поговорим о динамической компиляции. Во многих программах (или даже в большинстве программ) динамическая компиляция — важнейший фактор создания высокопроизводительного кода. Мы рассмотрим основы динамической компиляции, а в конце раздела расскажем, как активизировать логирование динамической компиляции, чтобы вы могли видеть, какие методы компилируются в настоящий момент.

6.6. Динамическая компиляция с применением HotSpot

В главе 1 мы уже говорили о том, что платформу Java лучше всего представлять как динамически компилируемую. Это означает, что классы приложения подвергаются дальнейшей компиляции во время исполнения, в результате чего преобразуются в машинный код.

Такой процесс именуется *динамической* (just-in-time) компиляцией или джейтингом. Обычно такой компиляции подвергается только один метод в единицу време-

ни. Необходимо понимать этот процесс, чтобы идентифицировать важные части любой объемной базы кода.

Рассмотрим некоторые базовые факты о динамической компиляции:

- практически в любой современной виртуальной машине Java найдется тот или иной динамический компилятор;
- полностью интерпретируемые виртуальные машины работают очень медленно по сравнению с теми, которые обеспечивают динамическую компиляцию;
- компилируемые методы работают гораздо быстрее, чем интерпретируемый код;
- целесообразно сначала компилировать те методы, которые используются наиболее активно;
- занимаясь динамической компиляцией, всегда стоит идти по пути наименьшего сопротивления.

Последний пункт означает, что сначала следует рассматривать скомпилированный код, поскольку в обычных условиях любой метод, остающийся в интерпретируемом виде, задействуется не так часто, как скомпилированный (иногда компиляция метода не удается, но это случается довольно редко).

На первом этапе работы с методом он является интерпретируемым представлением, находящимся в байт-коде. Виртуальная машина Java отслеживает, как часто вызывается метод (а также ведет другую статистику). Как только достигается пороговое значение (по умолчанию равное 10 000 раз) и метод готов для дальнейшего использования, поток виртуальной машины Java компилирует код метода в машинный код, что делается в фоновом режиме. Если компиляция будет выполнена успешно, то при всех последующих вызовах метода будет применяться скомпилированная форма, если только такой скомпилированный вариант не будет по какой-то причине поврежден или деоптимизирован.

В зависимости от конкретной природы кода в методе скомпилированный метод может работать вплоть до 100 раз быстрее, чем тот же метод в интерпретируемом виде. Понимание того, какие методы особенно важны для программы и какие из важных методов компилируются, — это обычно залог оптимизации производительности.

ЗАЧЕМ НУЖНА ДИНАМИЧЕСКАЯ КОМПИЛЯЦИЯ?

Иногда приходится слышать вопрос: а зачем вообще на платформе Java нужна динамическая компиляция, почему бы не делать ее заранее, как в C++. Обычно это объясняется тем, что гораздо удобнее использовать в качестве базовой единицы развертывания независимые от платформы артефакты (файлы `.jar` и `.class`), чем иметь дело с по-разному скомпилированными бинарными файлами для каждой из целевых платформ.

Другой, более амбициозный ответ, заключается в том, что языки, применяющие динамическую компиляцию, обычно могут предоставить компилятору больше информации. В частности, те языки, которые выполняют компиляцию перед исполнением (ahead-of-time compilation), не имеют доступа к какой-либо информации среды времени исполнения — например, о доступности определенных команд, характеристик аппаратного обеспечения, а также к статистике о работе кода. Открывается интересная возможность: по сути, динамически компилируемые языки, подобные Java, могут работать быстрее языков, использующих раннюю компиляцию.

В оставшейся части нашего разговора о динамической компиляции речь пойдет о виртуальной машине Java, которая называется HotSpot. Многие общие соображения, высказанные далее, применимы и к другим виртуальным машинам, но специфические детали могут значительно разниться.

Начнем с вводного рассказа об особом динамическом компиляторе, который входит в состав HotSpot, а потом опишем два наиболее мощных доступных варианта оптимизации — встраиваемую подстановку (inlining) и мономорфную диспетчеризацию (monomorphic dispatch). В заключение этого краткого раздела мы покажем, как активизировать логирование компиляции методов. Благодаря ему вы сможете отслеживать, какие именно методы компилируются в настоящий момент. Итак, начнем с введения в HotSpot.

6.6.1. Знакомство с HotSpot

HotSpot — это виртуальная машина, приобретенная Oracle после покупки компании Sun Microsystems (до этого Oracle уже владела виртуальной машиной JRockit, разработанной в компании BEA Systems). HotSpot — виртуальная машина, которая служит основой OpenJDK. Она может работать в двух самостоятельных режимах — клиентском и серверном. Для выбора режима нужно задать переключатель `-client` или `-server` при запуске виртуальной машины (это должен быть первый переключатель, указываемый в командной строке). Два этих режима предпочтительны для применения в различных практических ситуациях.

Клиентский компилятор

Клиентский компилятор ориентирован в первую очередь на использование в приложениях с графическим пользовательским интерфейсом. Это область, в которой жизненно важна согласованность (непротиворечивость) программы. Поэтому клиентский компилятор (иногда именуемый C1) в ходе компиляции обычно принимает сравнительно консервативные решения. Это означает, что он не может неожиданно остановиться, если откажется от оптимизационного решения, которое оказалось неверным или принималось на основании неверных предположений.

Серверный компилятор

Напротив, серверный компилятор (C2) в ходе компиляции активно оперирует предположениями. Чтобы гарантировать корректность кода, выполняемого в любой момент, C2 добавляет быструю динамическую проверку (обычно называемую *охранным условием* (guard condition)), проверяющую валидность сделанного предположения. Если валидность не подтвердится, то машина откажется от активной компиляции и попробует применить другой вариант. Такой активный подход может обеспечивать более высокую производительность, чем использование клиентского компилятора, всегда работающего «с оглядкой».

Java реального времени

В последнее время была разработана особая форма языка Java, именуемая *Java реального времени* (real-time Java). Многие разработчики задаются вопросом, почему код, от которого требуется высокая производительность, просто не использует эту платформу (кстати, это самостоятельная виртуальная машина Java, а не вариант HotSpot). Оказывается, что система реального времени, несмотря на распространенное заблуждение, не всегда оказывается самой быстрой.

Программирование реального времени целиком завязано на гарантиях, которые можно дать в том или ином случае. С точки зрения статистики система реального времени пытается уменьшить разброс временных промежутков, требуемых для выполнения определенных операций. Для этого системы реального времени могут идти на повышение средней длительности ожидания. Для обеспечения более согласованной работы приложения система может немного пожертвовать производительностью.

На рис. 6.11 мы видим две серии точек, отражающих ожидание. Серия 2 (верхняя группа точек) характеризуется повышенной средней длительностью ожидания (поскольку находится выше по шкале ожидания), но более слабым разбросом значений. Точки более плотно сгруппированы вокруг среднего уровня, чем точки из серии 1, которые сравнительно сильно разбросаны.

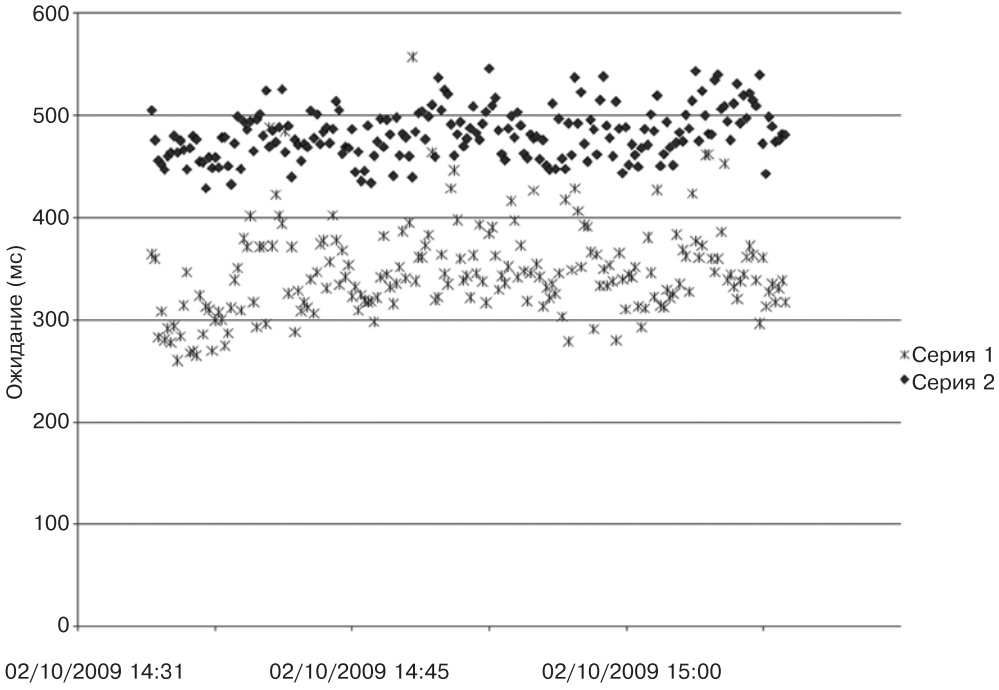


Рис. 6.11. Изменение разброса значений и средней длительности ожидания

Команды, стремящиеся повысить производительность, обычно пытаются снизить среднюю длительность задержки даже ценой большего разброса значений. Таким образом, серверный компилятор обычно активно занимается оптимизацией, и в результате получается такой разброс точек, как в серии 1.

Далее мы поговорим о практике, широко используемой для повышения эффективности в любых средах времени исполнения — серверных, клиентских и реального времени.

6.6.2. Встраиваемая подстановка методов

Встраиваемая подстановка (inlining) — один из самых мощных приемов, предоставляемых в HotSpot. При его применении вызов встроенного метода не выполняется. Вместо этого код вызываемого метода встраивается прямо в вызывающий элемент.

Одно из преимуществ платформы — компилятор может принять решение о встраивании на основании последних статистических данных о том, как часто вызывается конкретный метод, а также на основании других факторов (например, не станет ли вызывающий метод в результате встраивания слишком велик и не повлияет ли это на кэши кода). Таким образом, компилятор HotSpot может принимать значительно более разумные решения о встраивании, нежели механизмы, выполняющие раннюю компиляцию.

Встраивание методов — полностью автоматический процесс, практически во всех случаях вам подойдут значения, задаваемые по умолчанию. Существуют переключатели, позволяющие регулировать, методы какого размера должны встраиваться, а также насколько часто должен вызываться метод, прежде чем он будет рассмотрен для встраиваемой подстановки. Эти переключатели будут интересны в основном любопытным программистам, которые желают лучше понять, как встраивание методов организовано на внутрисистемном уровне. В «промышленном» коде они не слишком нужны; прибегать к ним стоит лишь в тех случаях, когда никакими другими способами не удается повысить производительность. Дело в том, что такие переключатели могут оказывать непредсказуемые эффекты на производительность всей среды времени исполнения.

А ЧТО НАСЧЕТ МЕТОДОВ ДОСТУПА?

Некоторые разработчики ошибочно полагают, что метод доступа (общедоступный метод-установщик, обращающийся к закрытой переменной экземпляра) нельзя встраивать с помощью HotSpot. Ход рассуждений при этом таков: если переменная закрыта, то от вызова метода нельзя отказаться ради оптимизации, поскольку доступ к такой переменной извне класса запрещен. Это неверно. HotSpot может игнорировать контроль доступа и делает это, компилируя методы в машинный код. При этом метод доступа просто заменяется прямым доступом к закрытому полю. Это не противоречит модели безопасности, действующей в Java, так как весь контроль доступа осуществляется на этапах загрузки и связывания классов.

Если вы все же хотите убедиться, что такое поведение реализуемо, напишите тестовую программу примерно как в листинге 6.2 и сравните скорость разогнанного метода доступа и скорость обращения к общедоступному полю.

6.6.3. Динамическая компиляция и мономорфные вызовы

Примером активной оптимизации, описанной выше, является оптимизация с применением мономорфных вызовов. Этот прием основан на том наблюдении, что в большинстве случаев подобный вызов метода к объекту:

```
MyActualClassNotInterface obj = getInstance();
obj.callMyMethod();
```

подразумевает, что все объекты, которым адресованы такие вызовы, будут относиться к одному и тому же типу. Иными словами, точка вызова `obj.callMyMethod()` практически никогда не будет иметь дело и с классом, и с его подклассом. В таком случае подстановку метода Java можно заменить прямым вызовом скомпилированного кода, соответствующего `callMyMethod()`.

СОВЕТ

Мономорфная диспетчеризация — это пример профилирования во время исполнения, применяемого в виртуальной машине Java. Таким образом, платформа может выполнять оптимизацию, на которую совершенно не способны языки, использующие раннюю компиляцию.

Нет никакой технической причины, по которой метод `getInstance()` в определенных обстоятельствах не мог бы вернуть объект типа `MyActualClassNotInterface`, а в других обстоятельствах — объект какого-либо подкласса. Но на практике это практически никогда не происходит. Тем не менее, чтобы застраховаться от такого случая, среда времени исполнения путем проверки удостоверяется, что тип `obj` был вставлен компилятором, как это и ожидалось. Если такое предположение когда-нибудь не оправдается, то среда времени исполнения уклонится от такого шага оптимизации, а программа этого даже не заметит и, конечно, не сделает чего-то неправильного.

Итак, мы рассмотрели достаточно агрессивный вариант оптимизации, применяемый только серверными компиляторами. Клиентские компиляторы и компиляторы реального времени так не работают.

6.6.4. Чтение журналов компиляции

Рассмотрим пример, позволяющий понять, какую пользу можно извлечь из сообщений журналов, регистрирующих работу динамического компилятора. В каталоге звезд Гиппарха перечислены звезды, которые можно наблюдать с Земли. Предположим, у нас есть приложение, обрабатывающее этот каталог и генерирующее звездные карты с картиной неба, которую можно наблюдать в определенную ночь в конкретной точке Земли.

Возьмем образец вывода, демонстрирующего, какие методы компилируются, когда мы запускаем наше приложение для построения звездных карт. Основной флаг виртуальной машины, которым мы будем пользоваться, — `-XX:+PrintCompilation`. Это один из расширенных переключателей, вкратце рассмотренных выше. Если добавить его в командную строку для запуска виртуальной машины Java, то потоки

динамической компиляции получат команду записывать сообщения в стандартный журнал. Эти сообщения указывают, когда те или иные методы преодолевают порог компиляции и преобразуются в машинный код.

```
  1       java.lang.String::hashCode (64 bytes)
  2       java.math.BigInteger::mulAdd (81 bytes)
  3       java.math.BigInteger::multiplyToLen (219 bytes)
  4       java.math.BigInteger::addOne (77 bytes)
  5       java.math.BigInteger::squareToLen (172 bytes)
  6       java.math.BigInteger::primitiveLeftShift (79 bytes)
  7       java.math.BigInteger::montReduce (99 bytes)
  8       sun.security.provider.SHA::implCompress (491 bytes)
  9       java.lang.String::charAt (33 bytes)
  1% !    sun.nio.cs.SingleByteDecoder::decodeArrayLoop @ 129 (308 bytes)
...
 39       sun.misc.FloatingDecimal::doubleValue (1289 bytes)
 40       org.camelot.hipparcos.DelimitedLine::getNextString (5 bytes)
 41 !     org.camelot.hipparcos.Star::parseStar (301 bytes)
...
  2% !    org.camelot.CamelotStarter::populateStarStore @ 25 (106 bytes)
 65 s     java.lang.StringBuffer::append (8 bytes)
```

Перед нами типичный вывод от `PrintCompilation`. Приведенные выше строки показывают, какие методы были признаны настолько «востребованными», что их стоит скомпилировать. Как вы, наверное, и ожидали, первыми в разряд компилируемых попадают методы платформы (например, `String#hashCode`). Со временем будут компилироваться и методы приложения (например, `org.camelot.hipparcos.Star#parseStar`, применяемый в данном примере для синтаксического анализа формы записи из астрономического каталога).

Строки вывода сопровождаются номерами, указывающими, в каком порядке методы компилируются в ходе работы. Обратите внимание, что от запуска к запуску этот порядок может немного меняться, что объясняется динамической природой платформы. А вот описание некоторых других полей:

- `s` — указывает, что метод синхронизирован;
- `!` — говорит о том, что в методе есть обработчики исключений;
- `%` — замещение в стеке (on-stack replacement, OSR). Метод был скомпилирован и заменил интерпретируемую версию прямо в работающем коде. Обратите внимание: OSR-методы имеют собственный порядок нумерации, начинающийся с 1.

Опасайтесь зомби

Изучая образцы журналов работающего кода, который был создан с помощью серверного компилятора (C2), вы иногда можете заметить строки made not entrant (стал непригоден для входа) и made zombie (стал зомби). Эти строки означают, что определенный метод, который был скомпилирован, теперь поврежден. Обычно это происходит из-за какой-то операции, выполненной при загрузке класса.

Деоптимизация

Машина HotSpot умеет деоптимизировать код, если оптимизация была произведена на основании предположения, оказавшегося ошибочным. Зачастую код после этого пересматривается, после чего проводится альтернативная оптимизация. Следовательно, один и тот же метод может быть многократно деоптимизирован и перекомпилирован.

Со временем вы заметите, что количество скомпилированных методов стабилизируется. Код достигает равновесного скомпилированного состояния и остается в основном без дальнейших изменений. Конкретные детали того, какие методы будут компилироваться, могут зависеть от точной версии виртуальной машины Java и используемой операционной системы. Ошибочно полагать, что на всех платформах будет образовываться один и тот же набор скомпилированных методов и что скомпилированный код конкретного метода будет иметь на разных платформах приблизительно одинаковый размер. Как и многие другие параметры производительности, эти моменты необходимо измерять, а результаты измерений могут получаться неожиданными. Оказывается, что самый безобидный на вид метод Java, преобразованный в машинный код посредством динамической компиляции, может пятикратно различаться в размерах на Linux и Solaris. Итак, без измерений здесь не обойтись.

6.7. Резюме

Принимаясь за настройку производительности, недостаточно просто уставиться в код и молиться об озарении. Стандартными быстрыми заплатками тоже не обойтись. Нет, настройка производительности — это скрупулезные измерения, внимание к деталям и терпение. Настройка производительности требует постоянного устранения источников ошибок, выявляемых в ходе тестирования. Лишь так можно найти истинные причины проблем с производительностью.

Вспомним рассмотренные нами ключевые моменты, связанные с оптимизацией производительности в динамической среде, предоставляемой виртуальной машиной Java:

- виртуальная машина Java — невероятно мощная и сложная среда времени исполнения;
- природа виртуальной машины Java такова, что оптимизация кода в такой машине иногда может быть нетривиальной задачей;
- чтобы точно представлять, откуда проистекают те или иные проблемы, необходимо заниматься измерениями;
- уделяйте особое внимание подсистеме сборки мусора и динамическому компилятору;
- вам могут очень пригодиться инструменты мониторинга и другие инструменты;
- учитесь читать журналы и другие индикаторы, предоставляемые на платформе, — специальные инструменты будут под рукой не всегда;
- необходимо измерять и ставить цели (это настолько важно, что мы скажем об этом повторно).

Теперь вы уже должны обладать базовыми знаниями, необходимыми для исследования продвинутых возможностей оптимизации производительности на платформе и проведения соответствующих экспериментов. Вы уже понимаете, как механизмы обеспечения производительности влияют на ваш собственный код. Надеемся, что вы чувствуете себя достаточно опытными и уверенными, чтобы непредвзято анализировать все эти данные и применять полученные выводы для решения проблем, связанных с производительностью.

В следующей главе мы выйдем за пределы языка Java и рассмотрим другие языки, предлагаемые для работы на виртуальной машине Java. Но многие параметры производительности платформы окажутся полезны и в более широком контексте — особенно детали, касающиеся динамической компиляции и сборки мусора.

ЧАСТЬ 3

Многоязычное программирование на виртуальной машине Java

Эта часть книги посвящена исследованию новых языковых парадигм и многоязычному программированию на виртуальной машине Java.

Виртуальная машина Java — замечательная среда времени исполнения. Она не только обеспечивает производительность и мощность, но и дает программисту удивительную гибкость в работе. На самом деле виртуальная машина Java позволяет сделать первый шаг к исследованию других языков, а не только Java. На ней вы можете опробовать совершенно новые для вас подходы к программированию.

Если до сих пор вы программировали только на Java, то можете спросить: «А зачем мне учить другие языки?» Как мы говорили в главе 1, профессиональный Java-разработчик старается постоянно совершенствоваться во всех тонкостях языка Java, полнее осваивать платформу и экосистему. Для этого необходимо вникать в темы, которые пока только вырисовываются на горизонте, но в ближайшем будущем станут неотъемлемыми частями технологической среды.

Будущее уже здесь — просто оно неравномерно распределено.

Уильям Гибсон[1]

Оказывается, что многие новые идеи, которые будут востребованы в будущем, уже сегодня присутствуют в других языках виртуальной машины Java. Это касается, например, функционального программирования. Изучив новый язык для виртуальной машины Java, вы можете украдкой заглянуть в другой мир, напоминающий мир будущего, в котором вам доведется реализовывать новые проекты. Исследуя незнакомую точку зрения на проблему, вы сможете применить уже имеющиеся знания под новым углом. Таким образом, изучая новый язык, вы можете обнаружить у себя новые таланты и освоить навыки, которые пригодятся вам в дальнейшем.

Эта часть начинается с главы, объясняющей, почему Java не всегда идеально подходит для решения любых проблем, чем полезны концепции функционального программирования и почему для конкретного проекта может понадобиться не Java, а другой язык.

В последнее время появилось много книг и статей, в которых продвигается мысль о том, что в ближайшем будущем функциональное программирование будет основным рабочим навыком любого практикующего разработчика. Во многих подобных статьях функциональное программирование описано так, что непросто понять, с чем его едят. Не более понятно и то, как функциональное программирование может проявляться в таком языке, как Java.

На самом деле функциональное программирование вообще не является чем-то единым целым. Это скорее стиль и постепенное изменение общих принципов мышления разработчика. В главе 8 мы покажем примеры, которые слегка напоминают функциональное программирование. Но они сводятся всего лишь к тому,

---

[1] Интересная статья об этом человеке находится по адресу http://lenta.ru/articles/2007/08/13/gibson/. — *Примеч. перев.*

чтобы обрабатывать код коллекций в более чистом и безошибочном стиле. Для этого мы применим язык Groovy. В главе 9 мы поговорим об «объектно-функциональном» стиле, взяв для примера язык Scala. Более чистый подход к функциональному программированию (который полностью избавлен от объектной ориентации) мы обсудим в главе 10, посвященной языку Clojure.

В части 4 мы рассмотрим несколько практических случаев, в которых можно написать превосходные решения на альтернативных языках. Если хотите убедиться в этом, то прямо сейчас загляните в часть 4, а потом возвращайтесь сюда и познакомьтесь с языками, необходимыми для реализации таких приемов.

7 Альтернативные языки для виртуальной машины Java

В этой главе:

- зачем вам могут понадобиться альтернативные языки для виртуальной машины Java;
- типы языков;
- критерии выбора альтернативных языков;
- как виртуальная машина Java работает с другими языками.

Если вы применяли язык Java для создания достаточно объемных проектов, то, вероятно, замечали, что иногда код получается слишком пространным и неуклюжим. Наверное, порой вам хотелось, чтобы все было иначе — как-то попроще.

К счастью, на свете есть неподражаемая виртуальная машина Java — вы могли оценить широту ее возможностей в нескольких последних главах. Она действительно настолько хороша, что на ней с успехом приживаются не только Java, но и другие языки. В этой главе мы покажем, как можно подмешивать в проект, написанный на Java, другой язык для нашей виртуальной машины.

В этой главе мы обсудим способы описания языков различных типов (например, статических и динамических). Кроме того, мы объясним, зачем вам могут понадобиться альтернативные языки и какими критериями следует руководствоваться при их подборе. Мы познакомим вас с тремя языками (Groovy, Scala и Clojure), которые будут более подробно рассмотрены в третьей и четвертой частях нашей книги.

Но прежде, чем ко всему этому приступить, мы хотели бы более убедительно рассказать о некоторых недостатках Java. Следующий раздел представляет собой объемный пример, в котором акцентируются отрицательные стороны. Здесь же мы начнем двигаться к пониманию того языкового стиля, который известен под названием *«функциональное программирование»*.

7.1. Языку Java недостает гибкости? Это провокация!

Допустим, вы создаете новый компонент в системе, которая занимается обработкой торговых операций (транзакций). Упрощенный вид системы показан на рис. 7.1.

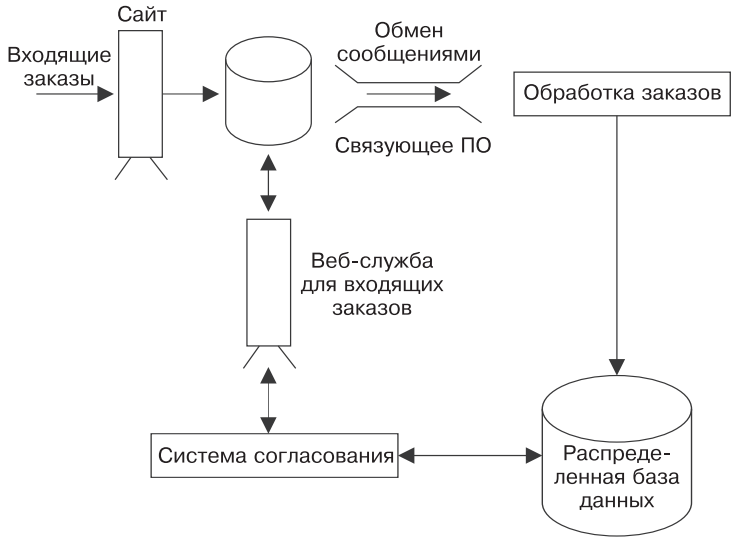


Рис. 7.1. Пример системы для обработки транзакций

Как видите, в системе два источника данных — подсистема поступающих заказов, которую можно запрашивать через веб-сервис, расположенная в верхней части, и находящаяся ниже распределенная база данных.

Такие системы на языке Java — как хлеб насущный, Java-разработчикам постоянно приходится создавать что-то подобное. В этом разделе мы покажем простой пример кода. Этот код занят согласованием работы двух источников данных. Далее мы объясним, почему с таким кодом может быть неудобно работать. Мы познакомим вас с сущностью функционального программирования и покажем, как функциональные идиомы, например *словарь* (map) и *фильтр* (filter), могут упростить многие распространенные задачи, решаемые при программировании. Вы увидите, что из-за отсутствия в Java прямой поддержки для этих идиом процесс программирования становится сложнее, чем следует.

7.1.1. Система согласования

Нам требуется система согласования, которая позволит гарантировать, что информация действительно будет попадать в базу данных. Основа такой системы — метод `reconcile()`, принимающий два параметра: `sourceData` (данные от веб-службы, собранные в словаре) и `dbIds`.

Необходимо получить ключ main_ref из sourceData и сравнить его с первичным ключом строк, которые вы получили из базы данных. В листинге 7.1 показано, как делается такое сравнение.

Листинг 7.1. Согласование работы двух источников данных

```
public void reconcile(List<Map<String, String>> sourceData,
Set<String> dbIds) {
  Set<String> seen = new HashSet <String>();
  MAIN: for (Map<String, String> row : sourceData) {
    String pTradeRef = row.get("main_ref");

    if (dbIds.contains(pTradeRef)) {
      System.out.println(pTradeRef +" OK");
      seen.add(pTradeRef);
    } else {
      System.out.println("main_ref: "+ pTradeRef +" not present in DB");
    }
  }

  for (String tid : dbIds) {
    if (!seen.contains(tid)) {
      System.out.println("main_ref: "+ tid +" seen in DB but not Source");
    }
  }
}
```

Предполагается, что pTradeRef никогда не равен нулю

Исключительный случай

Основной момент, который необходимо проверить, — попадает ли в распределенную базу данных вся информация, поступающая в систему в качестве входящих заказов. Эта задача решается в верхнем цикле for, который для ясности помечен как MAIN (главный).

Но есть и другая возможность. Предположим, что сотрудник, работающий внутри компании, сделал несколько тестовых заказов через административный интерфейс (не зная, что воспользовался «живой» системой). В таком случае заказы отобразятся в распределенной базе данных, а вот в системе входящих заказов их не будет.

Чтобы справиться с таким исключительным случаем, нужен второй цикл. Он проверяет, содержит ли видимое множество (все операции, произошедшие в двух системах) все строки, которые были зафиксированы в базе данных. Кроме того, этот цикл показывает, какие из строк отсутствуют. Вот вывод от пробного запуска:

```
7172329 OK
1R6GV OK
1R6GW OK
main_ref: 1R6H2 not present in DB
main_ref: 1R6H3 not present in DB
1R6H6 OK
```

Что пошло не так? Оказывается, что восходящий поток нечувствителен к регистру, а нисходящий — наоборот, чувствителен. `1R6H2` присутствует в распределенной базе данных, но там он называется `1r6h2`.

Если внимательно рассмотреть код из листинга 7.1, становится ясно, что проблема состоит в использовании метода `contains()`. Он проверяет, встречается ли искомый аргумент в рассматриваемой коллекции, но возвращает `true` лишь при точном совпадении.

Это означает, что на самом деле вам требуется метод `containsCaseInsensitive()`, которого пока нет! Поэтому следующий фрагмент кода:

```
if (dbIds.contains(pTradeRef)) {
  System.out.println(pTradeRef +" OK");
  seen.add(pTradeRef);
} else {
  System.out.println("main_ref: "+ pTradeRef +" not present in DB");
}
```

придется заменить вот таким циклом:

```
for (String id : dbIds) {
  if (id.equalsIgnoreCase(pTradeRef)) {
    System.out.println(pTradeRef +" OK");
    seen.add(pTradeRef);
    continue MAIN;
  }
}
System.out.println("main_ref: "+ pTradeRef +" not present in DB");
```

Получается громоздко. Приходится просматривать коллекцию в цикле, а не обрабатывать ее как единое целое. Код не слишком лаконичен, но при этом хрупок.

По мере роста вашего приложения этот недостаток лаконичности станет все более заметным. Действительно, нужно писать краткий код, ведь его к тому же проще осмысливать.

7.1.2. Концептуальные основы функционального программирования

В последнем примере есть две идеи, на которых мы хотели бы заострить ваше внимание.

- Если оперировать целыми коллекциями, то код получается и короче и, как правило, лучше, чем при переборе содержимого коллекции.
- Как было бы здорово, если бы можно было добавить небольшой логический компонент, позволяющий корректировать поведение имеющихся методов при работе с объектами.

Возможно, вам когда-либо доводилось писать код коллекций и раздражаться из-за того, что вот он, метод, который делает практически то, что надо, — лишь

немного бы его подправить. В таком случае сообщаем, что подобные досадные неудобства легко исправить благодаря функциональному программированию (ФП).

Другими словами, основной лимитирующий фактор, не позволяющий писать более лаконичный (и безопасный) объектно-ориентированный код, заключается в невозможности добавления дополнительной логики к уже имеющимся методам. Это подводит нас к одной из основных идей функционального программирования: что, если у вас будет возможность немного изменить функциональность метода, добавив в него часть собственного кода?

Поясним, что имеется в виду. Чтобы постфактум добавить собственный код, нужно передать в метод представление вашего блока кода в качестве параметра. Код, который вы хотели бы писать в таком случае, будет выглядеть примерно так (мы выделили полужирным шрифтом специальный метод `contains()`):

```
if (dbIds.contains(pTradeRef, matchFunction)) {
  System.out.println(pTradeRef +" OK");
  seen.add(pTradeRef);
} else {
  System.out.println("main_ref: "+ pTradeRef +" not present in DB");
}
```

Если бы у вас была такая возможность, то метод `contains()` можно было бы дорабатывать для любого тестового использования, которое могло бы вам понадобиться, — в данном случае для проверки совпадения при нечувствительности к регистру. Чтобы достичь этой цели, вам потребуется придумать способ представления вашей сопоставительной функции так, как если бы она была значением. Иначе говоря, нужна возможность написать часть кода как функциональный литерал, а потом присвоить эту конструкцию переменной.

Для того чтобы заниматься функциональным программированием, необходимо уметь представлять фрагменты логики (обычно — методы) так, как будто они являются значениями. Это центральная идея функционального программирования, и мы будем возвращаться к ней. Но сначала рассмотрим еще один пример на Java, в котором скрываются еще кое-какие идеи функционального программирования.

7.1.3. Идиомы словаря и фильтра

Немного расширим наш пример и рассмотрим контекст, в котором вызывается метод `reconcile()`:

```
reconcile(sourceData, new HashSet<String>(extractPrimaryKeys(dbInfos)));

private List<String> extractPrimaryKeys(List<DBInfo> dbInfos) {
  List<String> out = new ArrayList<>();
  for (DBInfo tinfo : dbInfos) {
    out.add(tinfo.primary_key);
  }

  return out;
}
```

Метод `extractPrimaryKeys()` возвращает список значений первичных ключей (в строковом виде), которые были извлечены из объектов базы данных. Любители функционального программирования назвали бы такое выражение `map()` — метод `extractPrimaryKeys()` принимает список `List` и возвращает другой список `List`, получаемый после совершения определенной операции над каждым элементом списка. Так и создается новый список, возвращаемый от метода.

Обратите внимание: тип элементов, содержащихся в новом списке `List`, может отличаться (`String`) от типа, находящегося во входящем списке `List` (`DBInfo`). Исходный список при этом никак не затрагивается.

Именно поэтому такое программирование и называется функциональным — функции ведут себя как математические выражения. Ведь, если передать в функцию `f(x) = x * x` значение 2, она не изменит этого значения, а вернет уже новое значение — 4.

ПРОСТОЙ ПРИЕМ ОПТИМИЗАЦИИ

В вызове `reconcile()` мы применили полезный и немного хитрый трюк. Вы передаете возвращенный список `List` от `extractPrimaryKeys()` в конструктор HashSet для преобразования его в `Set`. Так мы с удобством дедуплицируем список `List`, и вызову `contains()` приходится выполнять меньше работы в методе `reconcile()`.

Такое применение `map()` — классическая идиома функционального программирования. Она часто применяется в паре с другим широко известным паттерном, формой `filter()`, которая показана в листинге 7.2.

Листинг 7.2. Форма filter

```
List<Map<String, String>> filterCancels(List<Map<String, String>> in) {
  List<Map<String, String>> out = new ArrayList<>();
  for (Map<String, String> msg : in) {
    if (!msg.get("status").equalsIgnoreCase("CANCELLED")) {
      out.add(msg);
    }
  }
  return out;
}
```

Защитное копирование

Обратите внимание на шаг защитного копирования — здесь вы возвращаете новый экземпляр `List`. Вы не изменяете имеющийся `List` (соответственно форма `filter()` действует как математическая функция), а строите новый список `List`, проверяя каждый его элемент в функции, возвращающей булево значение (`boolean`). Если результат проверки элемента равен `true`, то вы добавляете его в конечный список `List`.

Чтобы форма-фильтр работала, нужна функция, указывающая, следует ли включить конкретный элемент в готовый список. Можете считать, что эта функция отвечает на вопрос «Следует ли пропустить этот элемент через фильтр?» относительно всех элементов, образующих коллекцию.

Такие функции называются *предикативными* (predicate function), и нам нужен способ, чтобы их представлять. Вот пример написания такой функции на псевдокоде (перед нами почти Scala):

```
(msg) -> { !msg.get("status").equalsIgnoreCase("CANCELLED") }
```

Это функция, принимающая один аргумент (`msg`) и возвращающая в качестве результата булево значение. Она возвращает `false`, если `msg` был отменен, а в противном случае — `true`. При применении такой функции в качестве фильтра она отсеет все отмененные сообщения.

Именно это нам и требуется. Перед вызовом согласующего кода нужно удалить все отмененные заказы, так как если заказ отменен — он будет отсутствовать в распределенной базе данных.

Оказывается, именно с таким синтаксисом данная функция будет записываться в Java 8 (но этот код в значительной мере напоминает Scala и C#). Мы вернемся к этой теме в главе 14, но прежде встретим такие функциональные литералы (также именуемые *лямбда-выражениями*) в некоторых других контекстах.

Для начала поговорим о том, языки каких типов могут работать на виртуальной машине Java. Такая классификация иногда называется *языковым зоопарком*.

7.2. Языковой зоопарк

Существует множество разновидностей языков программирования и вариантов их классификации. Иными словами, встречаются самые разнообразные стили и подходы к программированию, воплощенные в различных языках. Если вы хотите освоить эти стили и применять их себе на пользу, то должны понимать разницу между языками и принципы их классификации.

ПРИМЕЧАНИЕ

Эти классификации помогают осмысливать разнообразие языков. Одни варианты классификации более четкие, чем другие, но никакая из систем классификации не является совершенной.

В последние годы в языках также наблюдается тенденция к добавлению функций из всего спектра имеющихся возможностей. Это означает, что зачастую полезно представлять конкретный язык как «менее функциональный», чем другой язык, либо «динамически типизированный, при необходимости допускающий возможность статической типизации».

Мы поговорим о следующих вариантах классификации: «интерпретируемые и компилируемые языки», «императивные и функциональные языки», а также оригинальные языки и их повторные реализации. В принципе, такими классификациями следует пользоваться как удобными инструментами, помогающими мыслить в нужном контексте, а не как строгими академическими схемами.

Java — это компилируемый во время исполнения статически типизированный императивный язык. В нем делается акцент на безопасности, ясности кода и производительности. Поэтому Java мирится с некоторой пространностью и малой гибкостью кода (например, при развертывании). В различных языках могут суще-

ствовать разные приоритеты. Например, динамически типизированные языки могут делать акцент на скорости развертывания.

Для начала сравним интерпретируемые и компилируемые языки.

7.2.1. Сравнение интерпретируемых и компилируемых языков

В интерпретируемом языке каждый шаг исходного кода выполняется как есть — при этом для исполнения не требуется предварительно преобразовывать всю программу в машинный код. Этим интерпретируемые языки отличаются от компилируемых, которые начинают работу с применения компилятора для превращения человекочитаемого исходного кода в двоичную форму.

В последнее время названное различие стало размываться. В 1980-е годы и в начале 1990-х разница была довольно четкой: C/C++ и им подобные были компилируемыми языками, а Perl и Python — интерпретируемыми. Но, как мы уже указывали в главе 1, Java обладает чертами как первых, так и вторых. Использование байт-кода еще более усложняет проблему. Определенно байт-код непригоден для чтения человеком, но настоящим машинным кодом он также не является.

Говоря о языках для виртуальной машины Java, которые мы изучим в этой части книги, мы будем пользоваться следующим признаком: одни языки создают из исходного кода файл класса и выполняют его, а другие — нет. Во втором случае в языке будет интерпретатор (возможно, написанный на Java), используемый для исполнения исходного кода, строка за строкой. В некоторых языках есть и компилятор, и интерпретатор, а в других предоставляется интерпретатор и динамический компилятор, порождающий байт-код для виртуальной машины Java.

7.2.2. Сравнение динамической и статической типизации

В языках с динамической типизацией переменная в разное время может содержать различные типы. Для примера рассмотрим небольшой фрагмент кода, написанный на JavaScript — широко известном динамическом языке. Надеемся, что этот пример будет понятен, даже если вы не знакомы с этим языком в деталях:

```
var answer = 40;
answer = answer + 2;
answer = "What is the answer? " + answer;
```

В примере переменная `answer` начинает работу в значении `40` — естественно, это числовое значение. Потом мы прибавляем к нему 2 и получаем 42. А затем ситуация немного меняется, и мы записываем в переменную `answer` строковое значение. В динамическом языке это очень распространенный прием — из-за него не возникает никаких синтаксических ошибок.

Интерпретатор JavaScript также может различать два варианта использования оператора `+`. Первый вариант — это арифметическая операция сложения, в данном

случае сложения 2 и 40. А из контекста следующей строки интерпретатор заключает, что разработчик, применивший этот оператор, имел в виду сцепление (конкатенацию) строк.

ПРИМЕЧАНИЕ

Основная идея в данном случае — при динамической типизации отслеживается информация о том, значения каких типов содержатся в переменных (например, числовые или строковые). При статической же типизации отслеживается информация о типах переменных.

Статическая типизация хорошо подходит для компилируемого языка, поскольку вся информация о типах связана с переменными, а не со значениями, содержащимися в них. В результате становится очень легко судить о потенциальных нарушениях системы типов во время компиляции.

В динамически типизируемых языках информация о типах содержится в значениях, записываемых в переменных. В таком случае судить о потенциальных нарушениях типов становится гораздо сложнее, поскольку необходимая для этого информация остается неизвестной вплоть до времени исполнения.

7.2.3. Сравнение императивных и функциональных языков

Java 7 — классический образец императивного языка. Императивными называются языки, моделирующие рабочее состояние программы в виде изменяемых данных и выдающие набор команд, преобразующих это рабочее состояние. Таким образом, состояние программы — центральная концепция императивных языков.

Императивные языки делятся на два основных подтипа. Процедурные языки, такие как BASIC и Fortran, обрабатывают код и данные совершенно раздельно и действуют в соответствии с простой парадигмой «код оперирует данными». Второй подтип — это объектно-ориентированные языки, где данные и код (в форме методов) связываются вместе, образуя объекты. В объектно-ориентированных языках в большей или меньшей степени присутствует еще одна информационная структура — метаданные (например, информация о классах).

В функциональных языках во главу угла ставятся сами вычисления. Функции оперируют значениями, как и в процедурных языках, но не изменяют полученный ввод, а работают в точности как математические функции и возвращают новые значения.

Как показано на рис. 7.2, функции можно считать «маленькими вычислительными машинками», принимающими значения и выдающими новые значения. Сами по себе они не имеют состояния, и нет никакого смысла связывать их с каким-то внешним состоянием. Это означает, что объектно-ориентированная картина мира противоречит картине мира, присущей функциональным языкам.

В трех следующих главах мы поговорим о трех разных языках, опираясь на изложенную выше трактовку функционального программирования. Начнем с Groovy, где реализуется «почти функциональный стиль» для обработки таких коллекций, о которых мы говорили в разделе 7.1. Потом поговорим о языке Scala, который

в значительно большей степени связан с функциональным программированием. Наконец, обсудим Clojure (чисто функциональный язык, без всякой объектной ориентации).

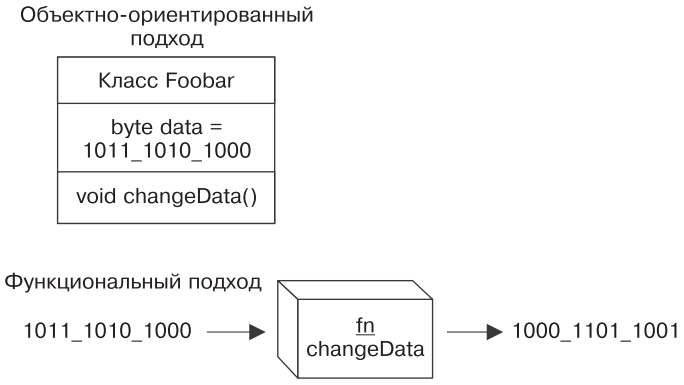


Рис. 7.2. Императивные и функциональные языки

7.2.4. Сравнение повторной реализации и оригинала

Еще одно важное отличие между языками для виртуальной машины Java заключается в том, что одни языки сразу писались с учетом специфики виртуальной машины Java, а другие являются лишь адаптациями (повторными реализациями) уже имеющихся языков, рассчитанными на применение на JVM. Вообще, те языки, которые создавались специально для работы на виртуальной машине Java, обеспечивают гораздо более тесную связь между своими системами типов и нативными типами виртуальной машины.

Следующие три языка для виртуальной машины Java являются повторными реализациями уже существующих языков.

- *JRuby* — повторная реализация языка Ruby для JVM. Ruby — это динамически типизированный объектно-ориентированный язык с некоторыми функциональными чертами. На виртуальной машине Java он, в сущности, является интерпретируемым, но в последних версиях появился динамический компилятор времени исполнения, который в подходящих условиях может создавать байт-код виртуальной машины Java.
- *Jython* — его разработка была начата в 1997 году. Автором проекта стал Джим Хаганин (Jim Hugunin), искавший способ использовать из Python высокопроизводительные библиотеки Java. Итак, Jython — это повторная реализация Python для виртуальной машины Java, динамический, в основном объектно-ориентированный язык. При работе он генерирует внутриязыковой байт-код Python, а потом преобразует его в байт-код виртуальной машины Java. Таким образом, работа протекает примерно так же, как и в типичном интерпретируемом

режиме Python. Кроме того, работа может идти с применением ранней компиляции, с генерированием байт-кода виртуальной машины Java и сохранением результирующих файлов классов на диск.

- *Rhino* — изначально разрабатывался компанией Netscape, потом — в рамках проекта Mozilla. Представляет собой реализацию JavaScript для виртуальной машины Java. JavaScript — это динамически типизированный объектно-ориентированный язык (но объектная ориентация в нем организована во многом иначе, нежели в Java). Rhino может работать как в компилируемом, так и в интерпретируемом режиме. Язык поставляется вместе с Java 7 (подробнее см. в пакете `com.sun.script.javascript`).

КАКОЙ ЯЗЫК ДЛЯ ВИРТУАЛЬНОЙ МАШИНЫ JAVA САМЫЙ ДРЕВНИЙ?

Сложно назвать самый ранний не Java-язык для виртуальной машины Java. Возможно, это язык Kawa — реализация Lisp, появившаяся в 1997 году или около того. В дальнейшем языки стали расти как грибы после дождя, и отследить их все уже практически невозможно.

На момент написания этой книги можно предположить, что для работы с виртуальной машиной Java может применяться около 200 языков. Не все они могут считаться активными и широко используемыми, но такое большое количество показывает, насколько оживленной платформой для разработки и внедрения языков является виртуальная машина Java.

ПРИМЕЧАНИЕ

В тех версиях спецификаций языка и виртуальной машины Java, которые были выпущены с появлением Java 7, из спецификации виртуальной машины были удалены все ссылки на язык Java. Теперь Java — один из многих языков, применяемых на JVM. У него нет каких-либо привилегий.

Основной технологический момент, позволяющий столь многим языкам работать на виртуальной машине Java, — формат файлов классов, которые мы рассмотрели в главе 5. Любой язык, в котором можно создавать файлы классов, считается на виртуальной машине Java компилируемым.

Теперь поговорим о многоязычном программировании и о том, как эта область стала особенно интересной для программистов, работающих с Java. Мы объясним базовые концепции многоязычного программирования и расскажем, почему вам в вашем проекте может понадобиться альтернативный язык, работающий на виртуальной машине Java.

7.3. Многоязычное программирование на виртуальной машине Java

Феномен «многоязычное программирование на виртуальной машине Java» относительно нов. Это выражение было придумано для описания проектов, использующих один или несколько языков для виртуальной машины Java, в то время как основа программы написана на Java. Многоязычное программирование зачастую

понимается как одна из форм функционального разделения ответственности. Как вы видите на рис. 7.3, потенциально существует три уровня, на которых могут пригодиться технологии, не относящиеся к Java. Эта схема иногда называется «пирамидой многоязычного программирования», ее автор — Ола Бини (Ola Bini).

В пирамиде мы видим три четко разграниченных уровня: уровень предметных областей, динамический и стабильный.

Рис. 7.3. Пирамида многоязычного программирования

СЕКРЕТ МНОГОЯЗЫЧНОГО ПРОГРАММИРОВАНИЯ

Многоязычное программирование целесообразно, поскольку длительность эксплуатации разных частей кода значительно отличается. Мезанизм оценки рисков в банке может работать на протяжении пяти и более лет. JSP-страницы для сайта могут существовать несколько месяцев. Самый короткоживущий код на стартапе может использоваться в течение всего нескольких дней. Чем дольше существует код, тем ближе к основанию пирамиды он располагается.

Это — своеобразный компромисс между такими проблемами, как производительность и тщательное тестирование, расположенными внизу пирамиды, и гибкостью и стремительным развертыванием в верхней части схемы.

В табл. 7.1 все три уровня описаны более подробно.

Таблица 7.1. Три уровня пирамиды многоязычного программирования

Название	Описание	Примеры
Языки предметных областей	Тесно связан с конкретной частью прикладной области	Apache Camel DSL, Drools, веб-шаблонизаторы
Динамические	Быстрая, продуктивная, гибкая разработка функционала	Groovy, Jython, Clojure
Стабильные	Основной функционал, стабильный, хорошо протестированный, высокопроизводительный	Java, Scala

Как видите, в уровнях наблюдаются закономерности — статически типизированные языки обычно используются для решения задач, относящихся к стабильному

уровню. И наоборот — менее мощные и менее универсальные технологии хорошо подходят для выполнения задач, находящихся в верхней части пирамиды.

В средней части пирамиды важнейшую роль играют языки динамического слоя. Этот слой также отличается наибольшей гибкостью — во многих случаях он может частично перекрывать оба соседствующих слоя.

Еще подробнее изучим эту схему и посмотрим, почему Java — не самый лучший язык для решения некоторых задач, представленных в пирамиде. Для начала обсудим, почему нужно обращать внимание на другие языки, кроме Java, а потом опишем кое-какие из основных критериев, которые следует учитывать, подбирая для проекта иной язык.

7.3.1. Зачем использовать другой язык вместо Java

Поскольку Java — это универсальный, статически типизированный компилируемый язык, у него немало достоинств. Благодаря этим качествам язык отлично подходит для реализации функционала на стабильном уровне. Но именно эти качества становятся обременительными в верхних частях пирамиды. Например:

- перекомпиляция отличается трудоемкостью;
- статическая типизация бывает негибкой, из-за этого возникают длительные этапы рефакторинга;
- развертывание — очень тяжеловесный процесс;
- синтаксис Java плохо подходит для создания языков предметных областей.

Длительность перекомпиляции и пересборки проекта Java быстро доходит до полутора и даже до двух минут. Это достаточно долго, чтобы нарушить нормальный ход работы программиста. Такие длительные сборки плохо подходят для разработки кода, который может использоваться в работе лишь на протяжении нескольких недель.

Прагматичное решение — воспользоваться сильными сторонами Java, а также богатым набором API этого языка и обширной библиотечной поддержкой. Благодаря этим качествам Java хорошо подходит для выполнения трудоемких задач, относящихся к стабильному уровню.

ПРИМЕЧАНИЕ

Если вы начинаете новый проект с нуля, то, возможно, придете к выводу, что отдельные возможности, важные в вашем будущем проекте, лучше представлены в другом языке для стабильного уровня, например в Scala (пример такой возможности в Scala — превосходная поддержка параллелизма). Но, как правило, не следует отказываться от готового рабочего кода для стабильного уровня и переписывать этот код на другом стабильном языке.

Здесь у вас могут возникнуть вопросы: «Какие типы проблем программирования относятся к этим уровням? Какие языки мне следует выбрать?» Профессиональный Java-разработчик знает, что панацеи не существует, зато есть критерии, которые можно учитывать при выборе. В этой книге мы не сможем обсудить все имеющиеся альтернативы, поэтому сосредоточимся на языках, которые, на наш взгляд, покрывают широкий спектр возможных разумных вариантов для Java-среды.

7.3.2. Многообещающие языки

В оставшейся части книги мы поговорим о трех языках, которые, на наш взгляд, в обозримом будущем окажутся наиболее долговечными и влиятельными. Это языки для виртуальной машины Java (Groovy, Scala и Clojure), у которых уже есть множество сторонников в среде многоязычного программирования. Итак, почему же три этих языка наращивают обороты? Вкратце рассмотрим каждый из них.

Groovy

Язык Groovy был изобретен Джеймсом Страханом (James Strachan) в 2003 году. Это динамический компилируемый язык, синтаксис которого очень напоминает синтаксис Java, но более гибок. Groovy широко используется как сценарный язык и язык для быстрого прототипирования. Зачастую именно Groovy оказывается первым после Java языком для JVM, за исследование которого берется разработчик или команда. Можно считать, что Groovy относится к динамическому уровню. Кроме того, он известен как отличная отправная точка для создания языков предметных областей (DSL). В главе 8 мы делаем введение в Groovy.

Scala

Scala — это объектно-ориентированный язык, также поддерживающий многие методы функционального программирования. Его история уходит корнями в 2003 год, когда разработкой этого нового языка занялся Мартин Одерски (Martin Odersky). Ранее Мартин вел проекты, связанные с дженериками в Java. Scala — это, как и Java, статически типизированный компилируемый язык, но, в отличие от Java, в Scala активно задействуется выведение типов во время исполнения. В результате Scala зачастую напоминает динамический язык.

Язык Scala многое заимствовал из Java, но в его структуре исправлены досадные недостатки, с которыми давно были вынуждены мириться Java-разработчики. Scala относится к стабильному уровню, и отдельные специалисты утверждают, что в один прекрасный день он может бросить вызов Java как «новый язык номер один для JVM». В главе 9 мы делаем введение в Scala.

Clojure

Clojure — язык, разработанный Ричардом Хики (Rich Hickey) и относящийся к семейству Lisp. Он наследует от этого семейства многие синтаксические черты (и знаменитое множество скобок)[1]. Это динамически типизированный функциональный язык, что неудивительно для представителя Lisp. Язык Clojure — компилируемый, но обычно программы, написанные на нем, распространяются в виде исходных текстов — причины этого мы рассмотрим ниже. В Clojure, наряду с Lisp-основой, содержится значительное количество новых возможностей (особенно в области параллельной обработки).

---

[1] Имеется в виду ироническая расшифровка LISP как аббревиатуры: Lots of Irritating Superfluous Parentheses — «Множество раздражающих ненужных скобок», см. http://lisper.ru/pcl/syntax-and-semantics. — *Примеч. перев.*

Принято считать, что Lisp — это язык для экспертов. Clojure несколько более прост в изучении, чем другие языки семейства Lisp, но тем не менее наделяет разработчика титанической силой (а также великолепно подходит для работы в стиле «разработка через тестирование»). Но Clojure, вероятно, не войдет в мейнстрим, оставаясь языком энтузиастов и средством для решения специфических задач (например, набор его возможностей будет очень интересен разработчикам приложений для финансовой сферы).

Обычно считается, что Clojure относится к динамическому уровню. Но поскольку он хорошо поддерживает параллелизм и некоторые другие возможности, можно сказать, что он справляется и со многими задачами из стабильного уровня. В главе 10 мы делаем введение в Clojure.

Итак, мы перечислили несколько перспективных языков, теперь рассмотрим проблемы, в зависимости от круга которых стоит выбирать тот или иной язык.

7.4. Как подобрать для проекта другой язык вместо Java

Если вы решили поэкспериментировать в вашем проекте с не Java-языками, то нужно определить, какие части проекта естественно вписываются в разные уровни, описанные выше, — стабильный, динамический и уровень предметных областей. В табл. 7.2 перечислены задачи, которые удобно решать на каждом из уровней.

Таблица 7.2. Области проекта, удобные для решения на каждом уровне

Уровень	Примеры решаемых проблем
Уровень предметной области	Сборка, непрерывная интеграция, непрерывное развертывание. Devops. Моделирование по принципу Enterprise integration pattern. Моделирование бизнес-правил
Динамический	Быстрая веб-разработка. Прототипирование. Интерактивные административные и пользовательские консоли. Скриптинг (написание сценариев). Тесты (для разработки через тестирование и разработки через реализацию поведения)
Стабильный	Параллельный код. Контейнеры приложений. Основной бизнес-функционал

Как видите, существует множество возможностей для применения альтернативных языков. Но идентификация задачи, для решения которой хорошо подходит альтернативный язык, — это только начало. Далее следует оценить, допустимо ли использование альтернативного языка. Ниже перечислены критерии, которые полезно учитывать при оценке технологического стека.

- Высоки ли риски в области реализации проекта?
- Насколько легко язык взаимодействует с Java?

- Какая инструментальная поддержка (например, на уровне интегрированной среды разработки) предоставляется для данного языка?
- Насколько крута кривая изучения этого языка?
- Насколько легко нанять разработчиков, умеющих разрабатывать программы на данном языке?

Подробно рассмотрим каждую из этих областей, чтобы было понятно, какие именно вопросы следует себе задавать.

7.4.1. Высоки ли риски в области проекта

Допустим, вы работаете с базовым движком, реализующим правила расчетно-кассового обслуживания и обрабатывающим более миллиона транзакций в день. Это стабильный кусок кода, написанного на Java, который используется уже несколько лет. Но нельзя сказать, что код хорошо протестирован, — в нем есть множество темных мест. Основа движка для расчетно-кассового обслуживания — такая область, в которой слишком рискованно опробовать новый язык, особенно если движок хорошо работает, но при этом плохо протестирован, и не так много разработчиков уверенно в нем разбираются.

Но система не ограничивается основным движком. Например, некоторые ситуации не помешает хорошо протестировать. В Scala есть хороший фреймворк тестов, называемый ScalaTest (о нем мы подробно поговорим в главе 11). С его помощью можно создавать JUnit-подобные тесты для кода на Java или Scala, но без такого огромного количества шаблонного кода, которое генерируется в JUnit. Итак, как только разработчик освоит базовые навыки обращения со ScalaTest, он сможет гораздо эффективнее улучшать тестовое покрытие программы. Кроме того, ScalaTest очень удобен для постепенного внедрения в код проекта таких концепций, как разработка через реализацию поведения (behavior-driven development). Если у вас под рукой есть современные инструменты для тестирования, то они очень пригодятся вам при рефакторинге или при замене элементов основного ядра кода. При этом неважно, будет новый движок для обработки транзакций написан на Java или на Scala.

Можно также предположить, что необходимо написать веб-консоль для того, чтобы операторы системы могли администрировать некоторые некритичные статические данные, связанные с системой обработки транзакций. Члены команды разработчиков знают технологии Struts и JSF, но не испытывают энтузиазма к ним обеим. Это еще одна не особенно рискованная область, в которой вполне можно опробовать новый язык и технологический стек. Такую консоль было бы логично построить на базе Grails (веб-фреймворк на основе Groovy, базирующийся на идеях из Ruby on Rails). Судя по разговорам в среде разработчиков, а также на основании нескольких исследований (одно из наиболее интересных было проведено Мэттом Рэйблом (Matt Raible)), можно утверждать, что Grails — самый продуктивный из имеющихся веб-фреймворков.

Работая над небольшим пилотным проектом в области с низкими рисками, менеджер всегда может остановить такой проект и перейти к использованию иной технологии реализации. При этом не возникнет каких-либо неожиданных последствий, если окажется, что задействованный технологический стек плохо подошел для команды или для всей системы.

7.4.2. Насколько хорошо язык взаимодействует с Java

Конечно же, вы не хотите отказываться от всего того отличного кода на Java, который уже написали! Это — одна из основных причин, по которым организации не любят включать в свой технологический стек новые языки программирования. Но если речь идет об альтернативных языках, работающих на виртуальной машине Java, ситуация может стать противоположной. Такие языки позволяют максимизировать ценность актуальной базы кода, не отказываясь при этом от уже имеющихся рабочих программных компонентов.

Альтернативные языки, работающие на виртуальной машине Java, могут аккуратно взаимодействовать с Java и, конечно же, развертываться в готовом окружении. Это особенно важно при обсуждении такого перехода с сотрудниками отдела продукт-менеджмента. Используя в своей системе не Java-язык, пригодный для работы на виртуальной машине Java, вы можете опираться на имеющийся опыт по поддержке уже функционирующего окружения. Так будет проще сглаживать любые проблемы, которые могут быть связаны с поддержкой нового решения, а также снизить риски.

ПРИМЕЧАНИЕ

Языки предметных областей обычно строятся на основе динамического (а иногда и стабильного) языка. Поэтому многие такие предметно-ориентированные языки работают на виртуальной машине Java на базе тех языков, из которых они были созданы.

Некоторые языки взаимодействуют с Java легче, чем другие. Мы пришли к выводу, что самые популярные альтернативные языки для виртуальной машины Java (в частности, Groovy, Scala, Clojure, Jython и JRuby) хорошо взаимодействуют с Java (у некоторых языков интеграция с Java просто отличная и практически бесшовная). Если даже вы крайне осторожны, можно быстро и легко сначала провести несколько экспериментов. Так вы сможете убедиться, что понимаете, какую пользу вам может принести такая интеграция.

Возьмем, к примеру, Groovy. Вы можете импортировать пакеты Java напрямую в код на этом языке, что делается обычной командой `import`. Воспользовавшись фреймворком Grails на основе Groovy, вы можете быстро написать сайт, сохранив ссылки на объекты из вашей модели Java. И наоборот: можно с легкостью вызывать код Groovy из Java несколькими способами. А в результате будут получаться знакомые объекты Java. Например, вызов кода Groovy из Java может понадобиться для обработки JSON, в результате чего будет возвращен объект Java.

7.4.3. Имеется ли хороший инструментарий и поддержка данного языка на уровне тестов

Большинство разработчиков недооценивают, сколько времени они могли бы сэкономить, если бы хорошо ориентировались в своем рабочем окружении. Есть мощные интегрированные среды разработки, а также инструменты для сборки и тестирования, позволяющие быстро создавать высококачественные программы. Уже много лет Java-разработчики пользуются превосходной инструментальной под-

держкой, поэтому необходимо учитывать, что другие языки зачастую являются гораздо более незрелыми, чем Java. Например, интегрированные среды разработки для Scala не так отточены, как в Java. Тем не менее сторонники Scala считают, что сила и лаконичность Scala более чем компенсируют то несовершенство, которым отличаются современные IDE этого языка.

В этой области есть и другая родственная проблема. Если в альтернативном языке был разработан мощный инструмент для внутриязыкового использования (например, великолепный инструмент Leiningen, применяемый для сборки в языке Clojure), то он может быть плохо приспособлен для работы с другими языками. А значит, команда должна предварительно разобраться, какие части выделить в проекте. Особенно это касается разработки независимых, но связанных компонентов.

7.4.4. Насколько сложно выучить данный язык

На изучение любого языка требуется время. И если парадигма конкретного языка отличается от той, с которой привыкла работать ваша команда разработчиков, то времени на изучение требуется больше. Как правило, команда Java-разработчиков легко сможет освоить новый язык, если он объектно-ориентированный и имеет синтаксис как у C (таков, например, Groovy).

Чем сильнее приходится отклоняться от этой парадигмы, тем сложнее становится задача для Java-разработчиков. Scala пытается сгладить разрыв между объектно-ориентированными и функциональными языками, но еще непонятно, реализуемо ли такое слияние в достаточно крупных проектах. Clojure — наиболее далекий от Java альтернативный язык, относящийся к числу популярных, — может быть исключительно полезен в проекте. Но команде разработчиков придется достаточно долго осваивать этот язык, поскольку потребуется впитать функциональную природу Clojure и изучить синтаксис Lisp.

В качестве альтернативы можно попробовать такие языки для виртуальной машины Java, которые являются повторными реализациями уже существующих языков. Ruby и Python — зрелые языки, разработчик найдет по ним множество материалов для работы и обучения. Варианты этих языков для виртуальной машины Java могут стать для ваших команд золотой серединой, так как специалисты смогут приступить к работе с новым языком, легким для изучения, но не являющимся Java.

7.4.5. Насколько много разработчиков использует данный язык

Организациям приходится проявлять здоровый прагматизм. Просто невозможно нанять лучших из лучших (несмотря на рекламные заявления), и даже в течение года кто-то будет покидать команду, а кто-то — приходить им на смену. Отдельные языки, например Groovy и Scala, становятся все популярнее, и формируется сообщество разработчиков, в котором можно нанять нужных специалистов. Но такой язык, как Clojure, пока еще не слишком распространен, и может быть сложно найти хорошего специалиста.

ВНИМАНИЕ

Необходимо сделать одно замечание относительно повторно реализованных языков. Например, многие имеющиеся пакеты и приложения, написанные на Ruby, тестировались только в оригинальной реализации, основанной на C. Это означает, что при использовании на виртуальной машине Java таких пакетов могут возникать проблемы. Принимая решения в контексте платформы, учитывайте, что вам понадобится дополнительное время на тестирование, если вы собираетесь полностью использовать весь стек, написанный на повторно реализованном языке.

Если найти специалиста по конкретному языку сложно, то опять же могут пригодиться повторно реализованные языки. Считаные разработчики укажут в резюме JRuby. Но поскольку JRuby — это просто Ruby для виртуальной машины Java, в вашем распоряжении есть многочисленная армия разработчиков. Если специалист знает Ruby на основе C, то он очень легко сможет усвоить разницу, возникающую при использовании Ruby на виртуальной машине Java.

Чтобы понять некоторые варианты подготовки проекта и ограничения, связанные с альтернативными языками для виртуальной машины Java, необходимо понимать, как виртуальной машине Java удается поддерживать множество языков.

7.5. Как виртуальная машина Java поддерживает альтернативные языки

Язык может работать на виртуальной машине Java одним из двух способов:

- использовать компилятор, создающий файлы классов;
- иметь интерпретатор, реализованный на байт-коде виртуальной машины Java.

В обоих случаях у вас обычно есть среда времени исполнения, обеспечивающая специфичную языковую поддержу для выполнения программ. На рис. 7.4 представлен стек среды времени исполнения для Java и для типичного не Java-языка.

Сложность таких систем для поддержки времени исполнения варьируется в зависимости от объема ручной поддержки, которая требуется конкретному языку при исполнении. Практически всегда среда времени исполнения реализуется как набор JAR-файлов, которые должны быть указаны в пути класса к исполняемой программе. Загрузка этих JAR-файлов будет происходить непосредственно перед запуском программы.

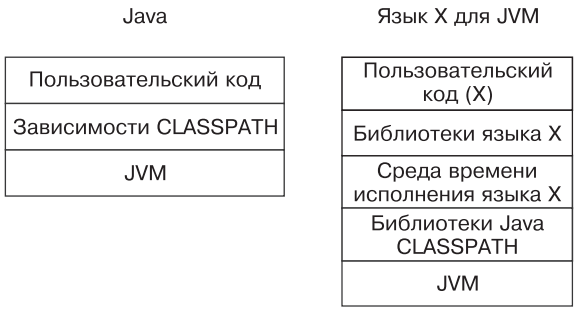


Рис. 7.4. Поддержка времени исполнения в не Java-языках

В этой книге мы уделяем основное внимание компилируемым языкам. Интерпретируемые языки, например Rhino, упомянуты для полноты картины, но мы не будем подробно на них останавливаться. В оставшейся части раздела 7.5 мы поговорим о необходимости поддержки среды времени исполнения для альтернативных языков (даже для компилируемых), а потом затронем фикции компилятора. *Фикции компилятора* — это специфические языковые элементы, синтезируемые компилятором, но не всегда отображаемые в байт-коде.

7.5.1. Среда времени исполнения для не Java-языков

Для того чтобы оценить сложность среды времени исполнения, необходимой для конкретного языка, просто посмотрите, каков размер JAR-файлов, представляющих реализацию этой среды. Если использовать этот показатель в качестве параметра, то становится ясно, что среда времени исполнения для Clojure сравнительно легковесна, а вот JRuby требует более серьезной поддержки.

Это не совсем объективный тест, так как в типичных дистрибутивах некоторых языков содержатся сравнительно крупные стандартные библиотеки, нежели в других языках, а также дополнительный функционал. Но этот параметр обычно является полезным (хотя и грубым) ориентиром.

Вообще, назначение среды времени исполнения заключается в том, чтобы помочь системе типов и другим характеристикам не Java-языка приобрести необходимую семантику. В альтернативных языках базовые концепции программирования не всегда трактуются так же, как и в Java.

Например, действующая в Java поддержка объектной ориентации не является универсальной для всех других языков. В Ruby к отдельно взятому экземпляру объекта могут во время исполнения прикрепляться дополнительные методы, которые еще не были известны на момент описания класса, а также не определены в других экземплярах того же класса. Эта особенность (имеющая несколько путаное название «открытые классы») должна реплицироваться и в реализации JRuby. Кроме того, она возможна лишь при расширенной поддержке со стороны среды времени исполнения JRuby.

7.5.2. Фикции компилятора

Определенные языковые функции синтезируются программной средой и высокоуровневым языком, которые отсутствуют в базовой реализации виртуальной машины Java. Они называются фикциями компилятора (compiler fiction). Мы уже встречали хороший пример такой фикции в главе 6 — имеется в виду сцепление строк в Java.

СОВЕТ

Полезно знать, как именно реализуются такие функции. В противном случае вы можете обнаружить, что код работает медленно, а иногда даже происходит аварийное завершение процесса. Иногда среде времени исполнения приходится выполнить немало работы, чтобы синтезировать определенную функцию.

Другими подобными функциями из Java являются проверяемые исключения и внутренние классы (которые всегда преобразуются в классы верхнего уровня и при необходимости снабжаются специально синтезируемыми методами доступа, как показано на рис. 7.5). Возможно, вы когда-нибудь заглядывали в недра JAR-файла (с помощью команды `jar tvf`) и видели множество классов, в имени которых встречается `$`. Это и есть распакованные внутренние классы, преобразованные в «обычные».

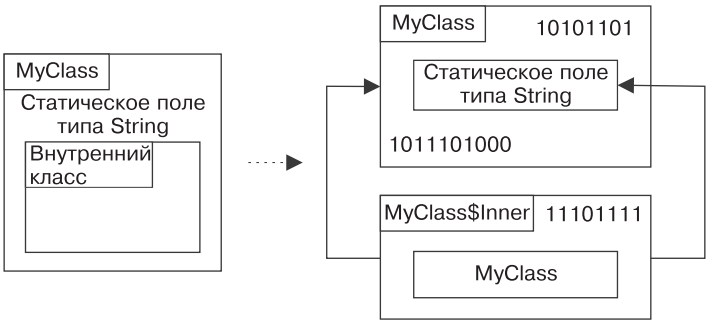


Рис. 7.5. Внутренние классы как фикция компилятора

В альтернативных языках также есть фикции компилятора. В некоторых случаях такие фикции даже образуют основу функциональности языка. Рассмотрим несколько важных примеров такого рода.

Функции первого класса

В разделе 7.1 мы рассказали об основной черте функционального программирования. Она заключается в том, что функции должны быть значениями, которые можно записывать в переменные. Обычно эта мысль выражается так: «функции являются значениями первого класса». Мы также показали, что в Java не лучшим образом поставлено моделирование функций.

Все альтернативные языки, которые нам предстоит рассмотреть в третьей части этой книги, поддерживают функции как значения первого класса. Другими словами, в них можно записывать функции в переменные, передавать в методы, а также выполнять с ними иные манипуляции, точно как с любыми другими значениями. Виртуальная машина Java просто оперирует классами, воспринимаемыми в ней как мельчайшие единицы кода и функциональности. Иначе говоря, на внутрисистемном уровне все не Java-языки создают маленькие анонимные классы для содержания функций (хотя в Java 8 это может измениться).

Чтобы уладить такое несоответствие между исходным кодом и байт-кодом виртуальной машины Java, достаточно вспомнить, что объекты — это просто пучки данных, связанные с методами, обрабатывающими эти данные. Теперь представьте, что у вас есть объект, не имеющий состояния и обладающий всего одним методом. Пример такого объекта — простая анонимная реализация интерфейса Java `Callable`, описанная в главе 4. Нет ничего необычного в том, чтобы записать такой

объект в переменную, передать его куда-то, а позже активизировать его метод `call()` вот так:

```
Callable<String> myFn = new Callable<String>() {
  @Override
  public String call() {
    return "The result";
  }
};

try {
  System.out.println(myFn.call());
} catch (Exception e) {
}
```

Мы опустили здесь обработку исключений, поскольку в данном случае метод `call()`, относящийся к переменной `myFn`, не может их выбрасывать.

ПРИМЕЧАНИЕ

Переменная myFn в этом случае относится к анонимному типу. Поэтому после компиляции она будет иметь примерно такой вид: NameOfEnclosingClass$1.class. Нумерация классов начинается с 1 и возрастает на 1 с каждым новым анонимным типом, который встречается компилятору. Если они создаются динамически и таких типов много (что нередко случается в таких языках, как JRuby), то может возникнуть серьезная нагрузка на постоянное поколение памяти, где хранятся определения классов.

Java-программисты очень активно пользуются такой уловкой, связанной с созданием анонимной реализации, хотя в Java для этого не предусмотрен какой-либо специальный синтаксис. Из-за этого результат может получиться несколько длинноватым. Во всех языках, которые нам предстоит рассмотреть, предоставляется специальный синтаксис для записи этих *функциональных значений* (они же — *функциональные литералы*, или *анонимные функции*). Это настоящие краеугольные камни функционального программирования, и они хорошо поддерживаются как в Scala, так и в Clojure.

Множественное наследование

Приведем другой пример. Известно, что в языке Java и в JVM отсутствует метод для выражения множественного наследования реализации. Единственный способ обеспечить такое наследование — использовать интерфейсы, в которых не допускается указание каких-либо тел методов.

Напротив, используемый в Scala механизм типажей (traits) позволяет подмешивать реализации методов в класс. Таким образом, в Scala доступен иной вид наследования. Мы подробно поговорим о нем в главе 9. Пока достаточно запомнить, что такое поведение в Scala требуется синтезировать, и этим занимается компилятор во время исполнения. Предпосылок для реализации такого поведения на уровне виртуальной машины нет.

На этом мы завершаем наше введение в различные типы языков, используемые на виртуальной машине Java.

7.6. Резюме

Альтернативные языки для виртуальной машины Java имеют долгую историю. Теперь они могут предложить более подходящие решения определенных проблем, чем сам язык Java, при этом сохраняя совместимость с имеющимися системами и инвестициями, сделанными в него. Это означает, что даже в Java-мастерских язык Java уже не является единственным решением любой конкретной задачи программирования.

Понимая различные способы классификации языков (статические и динамические, императивные и функциональные, компилируемые и интерпретируемые), мы можем осознанно выбирать правильный язык для решения каждой конкретной задачи.

С точки зрения многоязычного программиста, все языки могут условно относиться к одному из трех уровней: стабильному, динамическому и предметно-ориентированному. Такие языки, как Java и Scala, более удобны на стабильном уровне, а другие, например Groovy и Clojure, лучше подходят для решения задач в динамической и предметно-ориентированной области.

Определенные проблемы программирования четко соотносятся с тем или иным уровнем. Например, быстрая веб-разработка — это проблема для динамического уровня, а моделирование корпоративной системы обмена сообщений следует выполнять на предметно-ориентированном уровне.

Стоит еще раз подчеркнуть, что основной бизнес-функционал существующего приложения, активно применяемого на практике, почти никогда не подходит для экспериментального внедрения нового языка. Ядро проекта, обеспеченное высокой степенью поддержки, отлично протестированное и заведомо стабильное, — это область первостепенной важности. Если вы планируете опробовать развертывание нового языка, начинайте не с ядра, а с менее рискованной области.

Никогда не забывайте, что каждая команда и каждый проект обладают собственными уникальными характеристиками, которые необходимо учитывать при выборе языка. В данном случае не существует универсальных верных ответов. Если принимается решение о внедрении нового языка, то менеджеры и технические руководители должны отталкиваться именно от сути проекта и от специфики команды.

Небольшая команда, состоящая только из проверенных бойцов, может выбрать Clojure за чистый дизайн, затейливость и мощность (неважно, что язык концептуально довольно сложен, а нанять специалистов по нему затруднительно). В то же время веб-мастерская, желающая быстро расширить штат и привлечь молодых разработчиков, может выбрать Groovy и Grails, так как это позволит быстро достичь нужной производительности и не испытывать проблем при наборе кадров.

Мы рассмотрим три альтернативных языка для виртуальной машины Java, которые считаем наиболее перспективными: Groovy, Scala и Clojure. Дочитав эту книгу, вы усвоите основы трех наиболее многообещающих альтернативных языков для JVM и расширите свои навыки программирования в новом интересном направлении.

Следующая глава посвящена одному из этих языков — Groovy.

8 Groovy — динамический приятель Java

В этой главе:

- почему стоит изучить Groovy;
- основы синтаксиса Groovy;
- различия между Groovy и Java;
- мощные возможности Groovy, которых нет в Java;
- Groovy как сценарный язык;
- взаимодействие Groovy и Java.

Groovy — это объектно-ориентированный динамически типизированный язык, который, как и Java, работает на виртуальной машине JVM. Его можно считать языком, предоставляющим динамические возможности, удачно дополняющие статическую экосистему Java. Начало проекту Groovy положили Джеймс Страхан (James Strachan) и Боб Маквиртер (Bob McWhirter) в конце 2003 года, но в начале 2004 года руководителем проекта стал Гийом Лафорж (Guillaume Laforge). Сообщество разработчиков этого языка находится по адресу http://groovy.codehaus.org/, в наши дни оно продолжает цвести и пахнуть. Groovy считается наиболее популярным языком для JVM, не считая самого языка Java.

Groovy основан на идеях Smalltalk, Ruby и Python и реализует кое-какие языковые черты, которых нет в языке Java, в частности:

- функциональные литералы;
- первоклассная[1] поддержка коллекций;
- первоклассная поддержка регулярных выражений;
- первоклассная поддержка XML-обработки.

ПРИМЕЧАНИЕ

В Groovy функцинальные литералы называются замыканиями (closure). В главе 7 мы рассказали, что функциональные литералы — это функции, которые можно записывать в переменные, передавать в методы. Ими можно оперировать точно так же, как и любыми другими значениями.

[1] Под термином «первоклассная» понимается, что поддержка встроена в языковой синтаксис и не требует выполнения вызовов к библиотекам.

Итак, почему же стоит работать с Groovy? Если вы помните пирамиду многоязычного программирования, представленную в главе 7, то догадываетесь, что язык Java не слишком хорош для решения задач динамического уровня. К таким проблемам относится быстрая веб-разработка, прототипирование, написание сценариев и др. Groovy ориентирован на решение именно таких проблем.

Вот пример полезности Groovy. Предположим, начальник просит вас написать процедуру Java, преобразующую комплект бинов в XML. На Java, безусловно, можно решить такую задачу, на выбор вам предоставляются самые разные API и библиотеки:

- архитектура Java для связывания XML (JAXB) с API Java для обработки XML (JAXP), входящая в состав Java 6;
- библиотека XStream, расположенная на сайте Codehaus;
- библиотека XMLBeans от Apache;
- и многое другое...

Такая обработка довольно трудоемка. Например, для формирования объекта `Person` под JAXB нужно написать примерно такой код:

```
JAXBContext jc = JAXBContext.newInstance("net.teamsparq.domain");
ObjectFactory objFactory = new ObjectFactory();
Person person = (Person)objFactory.createPerson();
person.setId(2);
person.setName("Gweneth");
Marshaller m = jc.createMarshaller();
m.setProperty(Marshaller.JAXB_FORMATTED_OUTPUT, Boolean.TRUE);
m.marshal(person, System.out);
```

Кроме того, вам понадобится полностью запрограммированный класс `Person`. Чтобы он был полноценным Java-бином, его нужно снабдить методами-установщиками и методами-получателями.

В Groovy применяется иной подход, так как в этом языке обеспечивается первоклассная поддержка XML. Вот как выглядит аналогичный код на Groovy:

```
def writer = new StringWriter();
def xml = new groovy.xml.MarkupBuilder(writer);
xml.person(id:2) {
  name 'Gweneth'
  age 1
}
println writer.toString();
```

Как видите, этот язык очень быстр в работе и достаточно похож на Java, чтобы Java-разработчику было легко на него перейти.

Кроме того, функции Groovy нацелены на то, чтобы при работе требовалось меньше шаблонного кода. Например, для работы с XML и перебора коллекций на Groovy нужен гораздо более краткий код, чем в Java. Groovy хорошо взаимодействует с Java, и вы можете пользоваться динамической природой Groovy и функциями этого языка, совершенно не теряя связи с Java.

Для Groovy характерна плавная кривая обучения при условии, что вы уже владеете Java. Чтобы приступить к работе, вам понадобится только JAR-файл Groovy. Полагаем, что по окончании этой главы вы вполне освоитесь с этим новым языком.

Groovy проходит полный синтаксический анализ, компиляцию и генерацию перед тем, как выполняться на виртуальной машине Java. Из-за этого у многих разработчиков возникает вопрос: «Почему же Groovy не отлавливает очевидных ошибок во время компиляции?» Ответ прост: не забывайте, что Groovy — это динамический язык, выполняющий проверку типов и связывание во время исполнения.

ПРОИЗВОДИТЕЛЬНОСТЬ GROOVY

Groovy — неоптимальный язык, если к производительности будущей программы предъявляются строгие требования. Объекты Groovy строятся на основе `GroovyObject`, где содержится метод `invokeMethod(String name, Object args)`. Каждый раз при вызове метода Groovy этот метод, в отличие от ситуации с Java, не вызывается напрямую. Он выполняется с помощью вышеупомянутого метода `invokeMethod(String name, Object args)`, который, в свою очередь, осуществляет несколько рефлексивных вызовов и операций подстановки. Разумеется, из-за этого обработка тормозится. Создатели Groovy провели кое-какие оптимизации в этой области. Кроме того, планируются и другие оптимизации, которые будут реализованы после того, как новые версии Groovy смогут использоваться новым байт-кодом `invokedynamic` на виртуальной машине Java.

Groovy по-прежнему опирается на Java для выполнения определенной сложной работы. Из Groovy очень просто делать вызовы к имеющимся библиотекам Java. Благодаря такой возможности Java, а также динамической типизации Groovy и новым языковым функциям Groovy отлично подходит для быстрого прототипирования. Он также может применяться как язык сценариев (сценарный язык). Поэтому за Groovy закрепилась репутация гибкого и динамичного собрата Java.

В начале этой главы мы разберем несколько простых примеров на Groovy. Как только вы освоитесь с запуском простейшей программы на Groovy, мы перейдем к изучению специфического синтаксиса Groovy и тех разделов этого языка, на которых часто спотыкаются Java-разработчики. В оставшейся части этой главы мы разберем сущность Groovy, перечислим несколько основных возможностей этого языка, не имеющих аналогов в Java. Наконец, мы рассмотрим вариант многоязычного программирования на виртуальной машине Java, при котором Java комбинируется с Groovy!

8.1. Знакомство с Groovy

Если вы еще не установили Groovy, посмотрите в приложении C, как это делается. После того как вы установите язык на своей машине, мы перейдем к компиляции и запуску первого примера из этой главы.

В текущем разделе мы покажем, как запускать простейшие команды для компиляции и запуска Groovy, чтобы вы могли справиться с этой задачей в любой операционной системе. Мы также познакомим вас с консолью Groovy — ценной

и оперативной рабочей средой. Она не зависит от операционной системы, и в ней очень удобно опробовать небольшие фрагменты кода Groovy.

Установили? Тогда перейдем к компиляции и запуску нашего первого кода на Groovy!

8.1.1. Компиляция и запуск

Существует несколько полезных инструментов командной строки, которые следует освоить для работы с Groovy. В частности, это компилятор (`groovyc`) и исполнитель запросов (`groovy`). Две эти команды синонимичны `javac` и `java` соответственно.

ПОЧЕМУ СТИЛЬ ПРИМЕРОВ КОДА МЕНЯЕТСЯ?

Синтаксис и семантика приводимых нами примеров кода будет постепенно меняться. В этой главе мы будем переходить от Java-подобных примеров к полностью идиоматическим образцам Groovy. Так мы надеемся облегчить для вас переход от Java к Groovy. Кроме того, рекомендуем почитать книгу Кеннета А. Коусена Making Java Groovy, издательство Manning, 2012.

Кратко ознакомимся с этими инструментами командной строки, скомпилировав и запустив простой сценарий Groovy, который будет выводить на экран следующую строку[1]:

```
It's Groovy, baby, yeah!
```

Откройте приглашение командной строки и выполните следующие шаги.

1. Создайте в любом каталоге файл `HelloGroovy.groovy`.
2. Измените файл, добавив в него следующую строку:

   ```
   System.out.println("It's Groovy, baby, yeah!");
   ```

3. Сохраните файл `HelloGroovy.groovy`.
4. Скомпилируйте его, выполнив такую команду:

   ```
   groovyc HelloGroovy.groovy
   ```

5. Запустите файл с помощью следующей команды:

   ```
   groovy HelloGroovy
   ```

СОВЕТ

Этап компиляции можно пропустить, если файл исходников Groovy указан в вашем CLASSPATH. При необходимости среда времени исполнения Groovy сначала выполнит groovyc применительно к файлу исходников.

Поздравляем, вы впервые выполнили строку кода на Groovy!

Как и при работе с Java, код Groovy можно писать, компилировать и выполнять прямо в командной строке, но это быстро становится неудобным, если приходится

[1] Спасибо, Остин Пауэрс!

обращаться с такими вещами, как CLASSPATH. Groovy хорошо поддерживается в основных интегрированных средах разработки для Java (Eclipse, IntelliJ и NetBeans). Но в Groovy предоставляется и удобная консоль, в которой можно запускать код. Эта консоль идеальна в тех случаях, когда вы хотите быстро набросать решение или спрототипировать небольшой фрагмент кода, поскольку работать с командной строкой получается гораздо быстрее, чем с полнофункциональной IDE.

8.1.2. Консоль Groovy

Мы будем использовать консоль Groovy для запуска примеров кода из этой главы. Консоль Groovy — приятная, легкая в использовании интегрированная среда разработки. Для запуска консоли выполните в командной строке groovyConsole.

Должно открыться отдельное окно, как на рис. 8.1.

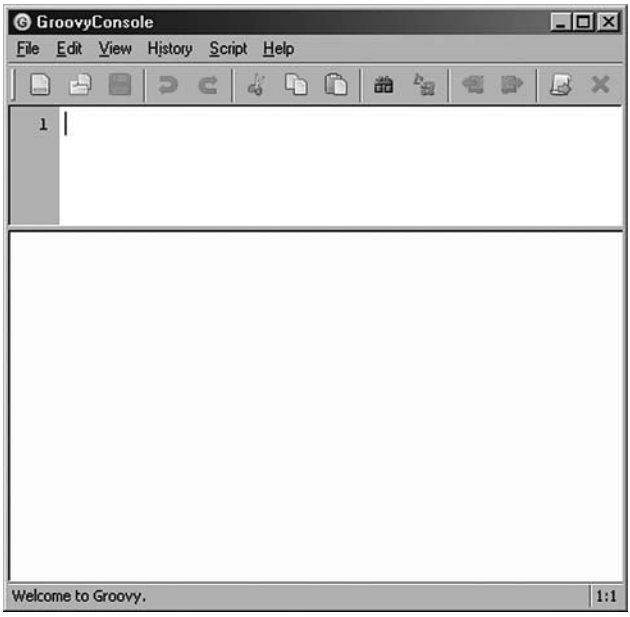


Рис. 8.1. Консоль Groovy

Сначала снимите флажок Show Script (Отобразить сценарий) в подменю Output (Вывод) меню View (Вид). В таком случае вывод станет немного чище. Теперь вы можете убедиться, что консоль работает правильно. Для этого запустите ту же строку кода на Groovy, что и в предыдущем примере. На верхней панели консоли введите следующую строку:

```
System.out.println("It's Groovy, baby, yeah!");
```

Затем нажмите кнопку Execute Script (Выполнить сценарий) или сочетание клавиш Ctrl+R. На нижней панели консоли Groovy отобразится такой вывод:

```
It's Groovy, baby, yeah!
```

Как видите, на панели вывода отображается результат выражения, которое вы только что выполнили.

Теперь, когда вы знаете, как быстро выполнять код на Groovy, познакомимся с синтаксисом и семантикой этого языка.

8.2. Groovy 101 — синтаксис и семантика

В предыдущем разделе вы написали однострочное выражение Groovy, не содержащее ни классов, ни методов (которые обычно требуются в Java). На самом деле вы написали обычный сценарий Groovy.

GROOVY КАК СЦЕНАРНЫЙ ЯЗЫК

В отличие от Java-кода, код Groovy можно выполнять как сценарий. Например, если у вас есть код вне определения класса, то этот код все равно выполнится. Как и сценарии других динамических сценарных языков, например Ruby или Python, сценарий Groovy проходит полный синтаксический разбор, компилируется и генерируется в памяти, прежде чем будет запущен на виртуальной машине Java. Любой код, который можно выполнить в консоли Groovy, также можно сохранить в файле `.groovy`, скомпилировать, а потом запустить как сценарий.

Некоторые разработчики стали использовать сценарии Groovy вместо сценариев оболочки, так как первые более мощные, их проще писать, а также они пригодны для кросс-платформенной работы, если в системе установлена виртуальная машина Java. В качестве экспресс-рекомендации по повышению производительности рекомендуем использовать здесь библиотеку `groovyserv`, запускающую расширения JVM и Groovy. В таком случае ваши сценарии будут работать гораздо быстрее.

Основная черта Groovy заключается в том, что в этом языке можно использовать такие же конструкции, как и в Java, синтаксис у двух языков также похож. Чтобы подчеркнуть это сходство, выполните следующий Java-подобный код в консоли Groovy:

```
public class PrintStatement
{
  public static void main(String[] args)
  {
    System.out.println("It's Groovy, baby, yeah!");
  }
}
```

В результате получим такой же вывод `It's Groovy, baby, yeah!`, как и от выполнения показанного выше однострочного сценария Groovy. Если не хотите пользоваться консолью Groovy, можете просто вставить приведенный выше код в файл исходников `PrintStatement.groovy`, воспользоваться командой `groovyc` для компиляции, а потом командой `groovy` для запуска кода. Иными словами, можно написать исходный код Groovy с классами и методами точно так же, как это делалось бы на Java.

СОВЕТ

В Groovy можно применять практически любые распространенные синтаксические конструкции Java. Ваши циклы while/for, конструкции if/else, операторы switch и т. д. будут работать именно так, как вы ожидаете. Весь новый синтаксис и основные различия будут подчеркиваться в этом и в следующих разделах.

Далее в этой главе мы познакомим вас со специфичными для Groovy синтаксическими идиомами. Наши примеры постепенно будут переходить от Java-подобного синтаксиса к более чистому синтаксису Groovy. Вы заметите четкую разницу между сильно структурированным кодом Java, с которым привыкли работать, и не таким многословным сценарным синтаксисом Groovy.

В оставшейся части этого раздела мы рассмотрим основной синтаксис и семантику Groovy, а также расскажем, почему это будет полезно вам как разработчику. В частности, мы обсудим:

- стандартные импорты;
- числовую обработку;
- переменные, сравнение динамических и статических типов, определение контекста (scoping);
- синтаксис для списков и словарей.

Первым делом нужно понять, что предоставляется в Groovy в готовом виде. Рассмотрим те языковые функции, которые по умолчанию импортируются в сценарий или программу Groovy.

8.2.1. Стандартный импорт

Groovy по умолчанию импортирует некоторые языковые и вспомогательные пакеты, предоставляя базовую языковую поддержку. Он также импортирует несколько пакетов Java, чтобы предоставить более широкий исходный функционал. Следующий список импортов всегда неявно присутствует в вашем коде Groovy:

- `groovy.lang.*`;
- `groovy.util.*`;
- `java.lang.*`;
- `java.io.*`;
- `java.math.BigDecimal`;
- `java.math.BigInteger`;
- `java.net.*`;
- `java.util.*`.

Для того чтобы задействовать и другие пакеты и классы, используйте оператор `import` так же, как делали бы это в Java. Например, чтобы получить все классы Math из Java, просто добавьте в исходный код на Groovy строку `import java.math.*;`.

ДОБАВЛЕНИЕ ОПЦИОНАЛЬНЫХ JAR-ФАЙЛОВ

Для того чтобы добавить определенный функционал (например, базу данных для работы в оперативной памяти и драйвер для этой базы данных), установку Groovy необходимо дополнить опциональными JAR-файлами. Для этого в Groovy есть специальные идиомы — наиболее распространена аннотация `@Grab`, вставляемая в сценарий. Другой способ (вспомним о нем, раз уж изучаем Groovy) — добавить JAR-файлы в путь `CLASSPATH`, так же как это делается в Java.

Воспользуемся некоторыми элементами этой стандартной языковой поддержки и рассмотрим, чем отличается числовая обработка в Java и Groovy.

8.2.2. Числовая обработка

Groovy может на лету интерпретировать математические выражения. Наблюдаемые при этом эффекты называются *принципом наименьшего удивления*. Этот принцип проявляется наиболее ярко, если вы работаете с числовыми литералами с плавающей точкой (например, 3,2). Функция `BigDecimal` в Java используется для представления числовых литералов с плавающей точкой на внутрисистемном уровне, но Groovy гарантирует, что поведение функции будет наиболее ожидаемым для разработчика.

Java и BigDecimal

Затронем область, в которой Java-разработчики часто испытывают затруднения при числовой обработке. Например, если в Java требуется сложить 3 и 0,2 с помощью `BigDecimal` — какой ответ должен получиться? Неопытный Java-разработчик, не сверяющийся с документацией Javadoc, вполне может написать следующий код, который возвратит поистине шокирующий результат `3.2000000000000000001110223024625156540423631668090820312 5`:

```
BigDecimal x = new BigDecimal(3);
BigDecimal y = new BigDecimal(0.2);
System.out.println(x.add(y));
```

Более опытный Java-разработчик знает, что в таком случае лучше воспользоваться конструктором `BigDecimal (String val)`, а не просто `BigDecimal`. Здесь `BigDecimal (String val)` получает числовой литерал в качестве аргумента. Такой улучшенный вариант записи даст результат `3.2`:

```
BigDecimal x = new BigDecimal("3");
BigDecimal y = new BigDecimal("0.2");
System.out.println(x.add(y));
```

Ситуация немного нелогична. В Groovy эта проблема решается путем применения по умолчанию оптимального конструктора.

Groovy и BigDecimal

В Groovy оптимальный конструктор автоматически используется при работе с числовыми литералами с плавающей точкой (как вы помните, на внутрисистемном уровне в таких случаях применяется `BigDecimal`). 3 + 0,2 = 3,2. Можете сами в этом убедиться, выполнив в консоли Groovy следующее выражение:

```
3 + 0.2;
```

Вы увидите, что принцип BEDMAS[1] корректно поддерживается и Groovy при необходимости интеллектуально переключается между числовыми типами (например, между `int` и `double`).

---

[1] Принцип, известный со школьной скамьи. Аббревиатура переводится на русский язык как «скобки, экспонента, деление, умножение, сложение, вычитание». Это стандартный порядок выполнения арифметических операций. — *Примеч. перев.*

Groovy уже позволяет выполнять арифметические операции чуть легче, чем это делается в Java. Если вы желаете знать, что именно происходит на внутрисистемном уровне, можете посмотреть на странице http://groovy.codehaus.org/Groovy+Math подробное описание всех деталей.

Следующая особенность семантики Groovy, которую необходимо изучить, связана с тем, как Groovy обращается с переменными и их контекстом (областью видимости). Здесь правила работы немного отличаются от действующих в Java. Это объясняется динамической природой Groovy и сценарным исполнением.

8.2.3. Переменные, сравнение динамических и статических типов, а также контекст

Поскольку Groovy — динамический язык, удобный для написания сценариев, в нем существуют определенные нюансы, связанные с разницей между динамическими и статическими типами, в определении контекстов переменных. В этих нюансах необходимо разбираться.

СОВЕТ

Если вы планируете обеспечить взаимодействие кода Groovy с кодом Java, старайтесь как можно чаще обходиться статическими типами, так как в таком случае упростится перегрузка и диспетчеризация типов.

Для начала нужно уяснить разницу между динамическими и статическими типами в Groovy.

Динамические и статические типы

Groovy — это динамический язык. Таким образом, вы можете не указывать тип объявляемой (или возвращаемой) переменной. Например, можно присвоить дату `Date` переменной `x` и сразу же после этого присвоить переменной `x` иной тип.

```
x = new Date();
x = 1;
```

При использовании динамических типов код получается более кратким (так как пропускаются «очевидные» типы), обеспечивается более быстрая обратная связь и гибкость при присваивании различных типов объектов единственной переменной, с которой вы хотите выполнить операцию. Для тех, кто хочет более конкретно знать, какой тип сейчас используется, Groovy также предоставляет статическую типизацию, например:

```
Date x = new Date();
```

Среда времени исполнения Groovy может обнаруживать случаи, в которых вы определили статический тип, а потом пытаетесь присвоить ему неверный тип, вот так:

```
Exception thrown

org.codehaus.groovy.runtime.typehandling.GroovyCastException: Cannot cast
```

```
    object 'Thu Oct 13 12:58:28 BST 2011' with class 'java.util.Date' to
    class 'double'
...
```

Можно получить точно такой же вывод, запустив следующий код в консоли Groovy:

```
double y = -3.1499392;
y = new Date();
```

Как и ожидалось, нельзя присвоить тип `Date` числу `double`. Итак, с разницей между динамическими и статическими типами в Groovy все ясно, поговорим о контексте.

Контексты в Groovy

При работе с классами в Groovy контекст (область видимости) задается точно так же, как и в Java, — с переменными, видимыми в пределах класса, метода и цикла. Это неудивительно. Но когда вы имеете дело со сценарием Groovy, тема контекстов становится интересной.

СОВЕТ

Напоминаем, что код Groovy, не входящий в состав обычного класса или метода, считается сценарием Groovy. Пример такого сценария приведен в подразделе 8.1.1.

Говоря простым языком, сценарий Groovy имеет два контекста:

- *привязки* — является глобальной областью видимости сценария;
- *локальный* — его суть понятна из названия. Контекст переменных ограничивается тем блоком, в котором они объявлены; например, если переменная объявлена в сценарном блоке (скажем, в начале сценария), то она будет относиться к локальному контексту, если определена.

Имея возможность использовать локальные переменные в сценарии, вы приобретаете значительную гибкость в работе. Переменная сценария немного похожа на переменную класса в языке Java. *Определенной* называется такая переменная, для которой объявлен статический тип, либо переменная, использующая специальное ключевое слово `def`. Оно указывает, что перед нами — определенная переменная без типа.

Методы, объявленные в сценарии, не имеют доступа к локальному контексту. Если вы вызываете метод, который пытается сослаться на переменную, ограниченную локальным контекстом, то эта попытка не удастся, в ответ вы получите примерно такое сообщение:

```
groovy.lang.MissingPropertyException: No such property: hello for class:
    listing_8_2
...
```

Следующий код выводит показанное выше исключение, подчеркивая эту проблему работы с контекстом.

```
String hello = "Hello!";
void checkHello()
```

```
{
  System.out.println(hello);
}
checkHello();
```

Если заменить первую строку в предыдущем коде на `hello = "Hello!";`, то метод успешно выведет на экран слово `Hello!`. Поскольку переменная `hello` не определена как строка (`String`), она оказывается в контексте *привязки*.

Не считая разницы, связанной с написанием сценариев Groovy, динамические и статические типы, определение контекста и объявление переменных происходят примерно так же, как вы и ожидали. Теперь поговорим о встроенной в Groovy поддержке коллекций (списков и словарей).

8.2.4. Синтаксис списков и словарей

Списки и словари (в том числе множества) являются в Groovy объектами первого класса. Поэтому, в отличие от Java, не требуется явно объявлять объекты `List` или `Map`. На базовом уровне списки и словари в Groovy реализованы так же, как знакомые из Java конструкции `ArrayList` и `LinkedHashMap`.

При использовании синтаксиса Groovy в данном случае вы приобретаете серьезное преимущество — приходится писать меньше шаблонного кода, весь код получается максимально лаконичным, оставаясь при этом вполне удобочитаемым.

Чтобы создать список в Groovy и дальше работать с ним, нужно использовать синтаксис с квадратными скобками (напоминающий нативный синтаксис массивов в Java). В следующем коде демонстрируется это поведение. Мы ссылаемся на первый элемент (`Java`), узнаем размер списка (`4`), а потом делаем список пустым `[]`.

```
jvmLanguages = ["Java", "Groovy", "Scala", "Clojure"];
println(jvmLanguages[0]);
println(jvmLanguages.size());
jvmLanguages = [];
println(jvmLanguages);
```

Как видите, если списки являются первоклассными объектами, работа с ними строится значительно проще, чем при обращении с `java.util.List` и его реализациями!

Динамическая типизация Groovy позволяет сохранять в списке (и, конечно, в словаре) значения смешанных типов. Поэтому следующий код тоже будет валиден:

```
jvmLanguages = ["Java", 2, "Scala", new Date()];
```

Работа со словарями в Groovy строится схожим образом, с применением квадратных скобок и двоеточия (`:`) для разделения пар ключ/значение. Для ссылки на значение в словаре требуется специальная нотация `map.key`. Это поведение демонстрируется в следующем коде, где мы наблюдаем:

- ссылку на значение `100` для ключа `"Java"`;
- ссылку на значение `"N/A"` для ключа `"Clojure"`;

- изменение значения ключа `"Clojure"` на `75`;
- установку пустого словаря (`[:]`):

```
languageRatings = [Java:100, Groovy:99, Clojure:"N/A"];
println(languageRatings["Java"]);
println(languageRatings.Clojure);
languageRatings["Clojure"] = 75;
println(languageRatings["Clojure"]);
languageRatings = [:];
println languageRatings;
```

СОВЕТ

Обратите внимание: ключи в словаре являются строками, но без кавычек. Это еще один случай, в котором Groovy ради краткости делает часть синтаксиса опциональной. При работе с ключами словаря вы можете выбирать, применять кавычки или нет.

Все это понятно на интуитивном уровне и очень удобно при работе. Groovy еще сильнее развивает концепцию первоклассной поддержки словарей и списков.

Можно пользоваться некоторыми синтаксическими хитростями, например ссылаться на диапазон объектов в коллекции или даже на последний элемент коллекции, пользуясь специальным отрицательным индексом. Следующий код демонстрирует такое поведение — здесь мы ссылаемся на первые три элемента в списке (`[Java, Groovy, Scala]`) и на последний элемент (`Clojure`)).

```
jvmLanguages = ["Java", "Groovy", "Scala", "Clojure"];
println(jvmLanguages[0..2]);
println(jvmLanguages[-1]);
```

Итак, мы рассмотрели некоторые базовые примеры синтаксиса и семантики, применяемые в Groovy. Но в этой области предстоит еще немало всего исследовать, прежде чем вы сможете эффективно использовать Groovy. В следующем разделе мы продолжим разговор о синтаксисе и семантике. Особый акцент сделаем на моментах, в которых Java-разработчику будет легко запутаться.

8.3. Отличия от Java — ловушки для новичков

Теперь вы уже вполне освоились с базовым синтаксисом Groovy, причем во многом благодаря синтаксическому сходству этого языка и Java. Но это сходство бывает обманчивым, и в текущем разделе мы более подробно поговорим о тех особенностях синтаксиса и семантики, которые могут запутать Java-разработчика.

Значительная часть синтаксических элементов в Groovy не обязательна. В частности, это касается:

- точек с запятой, завершающих инструкцию;
- операторов возврата (`return`);
- скобок при указании параметров методов;
- модификаторов доступа `public`.

Все это сделано для того, чтобы ваш исходный код получился более кратким. Несомненно, краткость полезна, когда вы быстро прототипируете какую-либо программу.

Среди других изменений следует упомянуть отсутствие разницы между проверяемыми и непроверяемыми исключениями, альтернативный способ обращения с концепцией равенства и идиома, связанная с отказом от использования внутренних классов. Начнем с простейшего из таких изменений: опциональные точки с запятой и опциональные операторы возврата (return).

8.3.1. Опциональные точки с запятой и операторы возврата

В Groovy символы точек с запятой (;) в конце инструкции не обязательны, если только вы не записываете несколько инструкций в одну строку.

Еще один опциональный синтаксический элемент, который можно не указывать, — это ключевое слово return, применяемое при возвращении объекта или значения от метода. Groovy автоматически возвращает результат последнего интерпретированного выражения.

В листинге 8.1 проиллюстрированы эти опциональные операторы. Значение 3 возвращается как результат последнего выражения, интерпретированного в методе.

Листинг 8.1. Точки с запятой и операторы возврата опциональны

```
Scratchpad pad = new Scratchpad()
println(pad.doStuff())

public class Scratchpad
{
  public Integer doStuff()
  {
    def x = 1
    def y; def String z = "Hello";
    x = 3
  }
}
```

Нет точек с запятой

Нет оператора возврата

Этот пример кода по-прежнему выглядит очень похожим на Java-код, но Groovy позволяет еще значительнее сократить его. Далее поговорим о скобках, которые в Groovy опциональны при указании параметров методов.

8.3.2. Опциональные скобки для параметров методов

При вызовах методов в Groovy скобки можно опускать, если у метода всего один параметр и отсутствует какая-либо двусмысленность. Таким образом, следующий код:

```
println("It's Groovy, baby, yeah!")
```

можно записать так:

```
println "It's Groovy, baby, yeah!"
```

В данном случае код опять же немного сокращается, не теряя при этом удобочитаемости.

Следующая особенность, отличающая код Groovy от кода Java, — это опциональный модификатор доступа `public`.

8.3.3. Модификаторы доступа

Основательный Java-разработчик знает, что определение уровня доступа к создаваемым классам, методам и переменным — важнейшая часть объектно-ориентированного проектирования. Groovy различает такие же уровни доступа `public`, `private` и `protected`, как и Java, но, в отличие от Java, в Groovy по умолчанию задается модификатор `public`. Итак, изменим листинг 8.1, удалив некоторые модификаторы `public`, и добавим модификаторы `private`, чтобы пояснить это изменение (листинг 8.2).

Листинг 8.2. `public` — модификатор доступа, задаваемый по умолчанию

```
Scratchpad2 pad = new Scratchpad2()
println(pad.doStuff())

class Scratchpad2
{
  def private x;
  Integer doStuff()
  {
    x = 1
    def y; def String z = "Hello";
    x = 3
  }
}
```

Модификатор доступа public отсутствует

Продолжая тему сокращения синтаксиса, коснемся и знакомых любому Java-разработчику условных операторов `throws`, применяемых в сигнатурах методов для передачи/вывода проверяемых исключений.

8.3.4. Обработка исключений

В отличие от Java, в Groovy отсутствует разница между проверяемыми и непроверяемыми исключениями. Любые условные операторы `throws` в сигнатурах методов игнорируются компилятором Groovy.

Groovy использует определенные синтаксические сокращения, чтобы исходный код стал более лаконичным, но не потерял удобочитаемости. Итак, рассмотрим одно синтаксическое изменение, которое сильно влияет на семантику, — речь о появлении оператора равенства.

8.3.5. Оператор равенства в Groovy

Groovy, следуя принципу максимальной предсказуемости, интерпретирует символ == как аналог метода `equals()`, применяемого в Java. Опять же это логично с точки зрения разработчика. Можно забыть о переключении между == и `equals()` при переходе от работы с примитивами к работе с объектами (а в Java такое различие необходимо учитывать).

Если вы хотите проверить равенство ссылок на текущий объект, то для этого придется воспользоваться функцией `is()` из Groovy. Из этого правила есть исключение — вы можете продолжать пользоваться ==, проверяя, не равен ли объект `null`. Эти поведения продемонстрированы в листинге 8.3.

Листинг 8.3. Равенство в Groovy

```
Integer x = new Integer(2)
Integer y = new Integer(2)
Integer z = null

if (x == y)
{
  println "x == y"
}

if (!x.is(y))
{
  println "x is not y"
}

if (z.is(null))
{
  println "z is not null"
}

if (z == null)
{
  println "z is null"
}
```

Вызван неявный equals()

Проверка идентификатора объекта

Проверка на нуль с помощью is()

Проверка на нуль

Конечно, можно проверять равенство и с помощью метода `equals()`, если вам так удобнее.

Остается еще одна конструкция Java, о которой мы хотели кратко рассказать. Это внутренние классы, вполне заменяемые в Groovy новой языковой конструкцией.

8.3.6. Внутренние классы

Внутренние классы поддерживаются в Groovy, но в большинстве случаев вместо них используются функциональные литералы. Мы поговорим о функциональных литералах в следующем разделе, так как эта мощная современная конструкция в программировании заслуживает подробного обсуждения.

Синтаксис и семантика Groovy позволяют писать меньше кода, сохраняя при этом достаточную удобочитаемость. В большинстве случаев вы можете продолжать использовать синтаксические конструкции, с которыми вам удобно работать. Далее мы покажем кое-какие языковые возможности Groovy, которых пока нет в Java. Вероятно, отдельные из этих возможностей могут стать переломными моментами в вашем проекте — например, если вы выберете Groovy для решения конкретной задачи, скажем XML-обработки.

8.4. Функции Groovy, пока отсутствующие в Java

В Groovy есть несколько важных языковых свойств, которых в Java (по крайней мере в Java 7) еще нет. Именно такие возможности и будут интересны основательному Java-разработчику, желающему воспользоваться новым языком для более изящного решения определенных проблем. В этом разделе мы исследуем несколько таких функций, в частности такие, как:

- GroovyBeans — упрощенные бины;
- безопасная работа с `null`-объектами с использованием оператора `?`;
- оператор Элвис — сокращенный способ записи конструкций `if/else`;
- строки Groovy — более мощная строковая абстракция;
- функциональные литералы (они же — замыкания) — для передачи функций;
- нативная поддержка регулярных выражений;
- упрощенная XML-обработка.

Начнем с GroovyBeans, так как эти бины будут регулярно встречаться вам в коде Groovy. Вам как Java-разработчику они могут показаться довольно подозрительными, поскольку они и близко не кажутся такими полными, как JavaBeans. Но можете не сомневаться: GroovyBeans не только полнее, но и удобнее, чем JavaBeans.

8.4.1. GroovyBeans

GroovyBeans во многом напоминают JavaBeans — за исключением того, что они обходятся без явных получателей и установщиков, предоставляют автоконструкторы и позволяют вам ссылаться на переменные экземпляров с помощью точечной нотации. Если вы хотите сделать конкретный получатель или установщик закрытым (private), либо желаете явно изменить определенное поведение, то можете явно предоставить метод для такого поведения и нужным образом его изменить. Автоконструкторы просто позволяют построить GroovyBean, передав словарь параметров, соответствующих переменным экземпляра этого GroovyBean.

Благодаря всем этим возможностям вы избавляетесь от необходимости работать с огромными объемами шаблонного кода, генерируемого при обращении с JavaBeans. Ведь при написании шаблонного кода создается множество методов-установщиков

и методов-получателей — их требуется старательно набирать от руки либо приходится загружать этой работой вашу IDE.

Рассмотрим, как работает GroovyBean. Для примера возьмем класс `Character` из ролевой игры (RPG)[1]. Код из листинга 8.4 порождает вывод `STR[18], WIS[15]`, демонстрирующий переменные экземпляров, соответствующих уровням силы и мудрости персонажа в GroovyBean.

Листинг 8.4. Исследование GroovyBean

```
class Character
{
  private int strength
  private int wisdom
}

def pc = new Character(strength: 10, wisdom: 15)
pc.strength = 18
println "STR [" + pc.strength + "] WIS [" + pc.wisdom + "]"
```

Такое поведение очень напоминает работу аналогичного JavaBean в Java (инкапсуляция сохраняется), но синтаксис более краток.

СОВЕТ

Можно использовать аннотацию @Immutable, чтобы сделать GroovyBean неизменяемым (то есть указать, что он не может изменяться). Эта возможность может пригодиться для передачи потокобезопасных конструкций данных, которые очень удобны при работе с параллельным кодом. Такая концепция неизменяемых структур данных подробнее рассматривается в главе 10, посвященной Clojure.

Далее поговорим о том, как Groovy помогает выполнять проверку на `null`. Опять же удается избавиться от значительного количества шаблонного кода, и вы можете более оперативно прототипировать свои идеи.

8.4.2. Оператор безопасного разыменования

Исключение `NullPointerException`[2] (NPE) — такая вещь, с которой (к сожалению) не понаслышке знакомы все Java-разработчики. Чтобы избежать NPE, им обычно требуется выполнять проверку на `null` прежде, чем ссылаться на объект, особенно если нельзя гарантировать, что конкретный объект, с которым приходится иметь дело, не является `null`. Если бы мы перенесли такой стиль разработки в Groovy, чтобы перебирать список объектов `Person`, то нуль-безопасный код пришлось бы писать следующим образом (такой код просто выводит `"Gweneth"`):

```
List<Person> people = [null, new Person(name:"Gweneth")]
for (Person person: people) {
```

[1] Здесь хотелось бы поаплодировать PCGen (http://pcgen.sf.net) — крайне полезному открытому проекту для нас, РПГ-шников.

[2] Одна из самых постыдных черт Java состоит не в том, что это исключение называется NullReferenceException, а в том, чем на самом деле оно является. По этому поводу один из авторов может разразиться целой тирадой.

```
  if (person != null) {
    println person.getName()
  }
}
```

Groovy позволяет избавиться от части шаблонного кода, «если объект равен нулю». Для проверки кода на нуль вводится синтаксис безопасного разыменования, при этом применяется нотация `?.`. Используя такую нотацию, Groovy вводит специальную конструкцию `null`, означающую фактически «ничего не делать», в отличие от настоящей `null`-ссылки.

В Groovy можно переписать предыдущий фрагмент кода с применением синтаксиса безопасного разыменования следующим образом:

```
people = [null, new Person(name:"Gweneth")]
for (Person person: people) {
  println person?.name
}
```

Такая поддержка безопасного разыменования распространяется в Groovy и на функциональные литералы. Поэтому используемые по умолчанию методы коллекций Groovy, например `max()`, автоматически начинают слаженно взаимодействовать с `null`-ссылками.

Далее обсудим оператор Элвис, который выглядит почти как оператор безопасного разыменования, но имеет и особый случай использования — позволяет сократить синтаксис отдельных конструкций `if/else`.

8.4.3. Оператор Элвис

Оператор Элвис (`?:`) позволяет писать конструкции `if/else`, имеющие значение по умолчанию. Для этого применяется исключительно краткий синтаксис. Почему он так называется? Дело в том, что изгиб вопросительного знака немного напоминает волнистую шевелюру Элвиса Пресли[1]. Оператор Элвис позволяет обходиться без явной проверки на `null`, а также избегать какого-либо повторения переменных.

Допустим, вы хотите проверить, участвовал ли Остин Пауэрс в выбранной миссии в качестве агента. В Java для этого придется использовать тернарный оператор вот так:

```
String agentStatus = "Active";
String status = agentStatus != null ? agentStatus : "Inactive";
```

В Groovy эту запись можно сократить, так как он при необходимости приводит типы к булевым значениям (`boolean`). Например, это происходит при условных проверках, такой способ работы напоминает использование операторов `if`. В предыдущем фрагменте кода Groovy приводит `String` к `boolean`. При условии, что строка `String` была равна `null`, она преобразуется в булево значение `false` — таким

[1] Авторы, правда, не знают, как именно Элвис Пресли выглядел в зените славы. Мы гораздо моложе, честное слово!

образом, можно обойтись без проверки на null. После этого предыдущий фрагмент кода можно записать так:

```
String agentStatus = "Active"
String status = agentStatus ? agentStatus : "Inactive"
```

Но переменная agentStatus по-прежнему повторяется, и Groovy может освободить вас от ввода дополнительного синтаксиса. Оператор Элвис позволяет избавиться от дублирования имени переменной:

```
String agentStatus = "Active"
String status = agentStatus ?: "Inactive"
```

Второй экземпляр переменной agentStatus удаляется, и мы еще немного сокращаем код.

Теперь рассмотрим строки Groovy и поговорим об их небольших отличиях от обычных строк String из Java.

8.4.4. Улучшенные строки

В Groovy есть расширение класса String, известного нам из Java. Этот расширенный класс называется GString и отличается немного большим потенциалом и гибкостью, чем привычная строка String.

Принято определять обычные строки между открывающей и закрывающей одиночной кавычкой, хотя двойные кавычки здесь также уместны. Например:

```
String ordinaryString = 'ordinary string'
String ordinaryString2 = "ordinary string 2"
```

В свою очередь, строки GString *необходимо* определять в двойных кавычках. Основная польза, которую они приносят разработчику, заключается в том, что такие строки могут содержать выражения (применяющие синтаксис ${}), пригодные для интерпретации во время исполнения. Если строка GString впоследствии будет преобразована в обычную строку String (для этого ее необходимо передать в вызов println), то все выражения, содержащиеся в строке GString, будут интерпретированы. Например:

```
String name = 'Gweneth'
def dist = 3 * 2
String crawling = "${name} is crawling ${dist} feet!"
```

В результате интерпретации выражений получается либо объект Object, к которому можно применить метод toString(), либо функциональный литерал (по адресу http://groovy.codehaus.org/Strings+and+GString можно подробнее прочитать о сложных правилах, регулирующих работу с функциональными литералами и GString).

ВНИМАНИЕ

Строки GString на внутрисистемном уровне совершенно не похожи на строки String из Java! В частности, не пытайтесь использовать GString в качестве ключей в словарях, а также не пробуйте проверять их на равенство. Скорее всего, результаты получатся неожиданными!

Еще один полезный конструкт из Groovy — это строка `String` или `GString` в тройных кавычках, которая может занимать в исходном коде несколько строк.

```
""" Эта строка GString
Заверстывается на две строки!"""
```

Далее поговорим о функиональных литералах. В последние годы эта техника программирования вновь стала темой жарких дискуссий, что объясняется ростом интереса к функциональным языкам. Чтобы понимать фунциональные литералы, требуется некоторая вычурность мышления. Если вы не привыкли с ними работать, но сейчас планируете основательно в них разобраться, — выпейте, пожалуй, чашечку любимого кофе, а потом возвращайтесь к чтению.

8.4.5. Функциональные литералы

Функциональный литерал — это представление блока кода, которое можно передавать как значение. Оно поддается таким же манипуляциям, как любое другое значение. Функциональный литерал можно передавать в методы, присваивать переменным и т. д. Функциональный литерал как языковая черта активно обсуждается в сообществе Java-разработчиков, но в инструментарии для программирования на Groovy функциональный литерал — привычная вещь.

Как обычно, новую концепцию лучше всего осваивать на примерах, так что рассмотрим парочку!

Допустим, у вас есть статический метод, выстраивающий строку `String` с приветствием для авторов или читателей. Вы, как обычно, вызываете его извне служебного класса (листинг 8.5).

Листинг 8.5. Простая статическая функция

```
class StringUtils
{
  static String sayHello(String name)
  {
    if (name == "Martijn" || name == "Ben")
      "Hello author " + name + "!"
    else
      "Hello reader " + name + "!"
    }
}
println StringUtils.sayHello("Bob");
```

Объявление статического метода

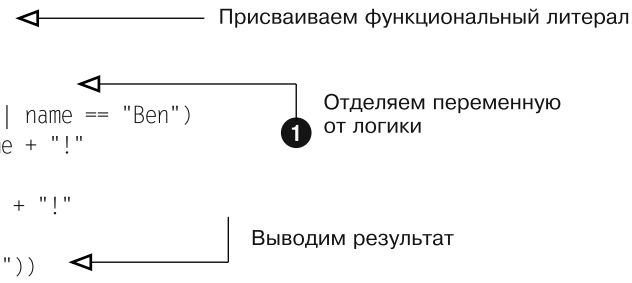
Вызывающий элемент

Пользуясь функциональными литералами, можно определять код, который будет работать, не требуя использования структур методов или классов. Иначе говоря, вы сможете предоставить в функциональном литерале такие же возможности, как и в методе `sayHello(String name)`. Этому функциональному литералу можно, в свою очередь, присвоить переменную, которую впоследствии можно будет передавать и исполнять.

Листинг 8.6 выводит текст `Hello author Martijn` в результате передачи функционального литерала, присвоенного `sayHello`, в который была отправлена переменная `"Martijn"`.

Листинг 8.6. Задействуем простой функциональный литерал

```
def sayHello =
{
  name ->
  if (name == "Martijn" || name == "Ben")
    "Hello author " + name + "!"
    Else
    Hello reader " + name + "!"
  }
println(sayHello("Martijn"))
```



Обратите внимание на синтаксис: фигурная скобка { открывает функциональный литерал. Стрелка (->) 1 позволяет отделять параметры, передаваемые в функциональный литерал, от выполняемой при этом логики. Наконец, символ } закрывает функциональный литерал.

В листинге 8.6 мы поступили с функциональным литералом примерно так же, как поступили бы с методом. Поэтому вы могли бы подумать: «Вот видите — не так уж они и полезны!» Но их достоинства действительно становятся очевидны лишь тогда, когда вы начинаете обращаться с ними творчески, как функциональный программист. Например, функциональные литералы особенно удобны в сочетании с первоклассной поддержкой коллекций, предоставляемых в Groovy.

8.4.6. Первоклассная поддержка для операций с коллекциями

В Groovy есть несколько встроенных методов, которые можно применять при работе с коллекциями (списками и словарями). Такая поддержка коллекций на уровне языка в сочетании с использованием функциональных литералов позволяет существенно уменьшить объем шаблонного кода, без которого обычно не обойтись в Java. Важно отметить, что удобочитаемость при этом не портится, благодаря чему поддержка кода остается сравнительно несложной.

Полезные встроенные функции, использующие функциональные литералы, приведены в табл. 8.1.

Таблица 8.1. Подмножество функций Groovy для работы с коллекциями

Метод	Описание
each	Перебирает коллекцию, применяя функциональный литерал
collect	Собирает значения, возвращаемые в результате вызова функционального литерала применительно к каждому элементу коллекции (подобно парной конструкции map/reduce, используемой в других языках)
inject	Обрабатывает коллекцию с применением функционального литерала и выстраивает возвращаемое значение (подобно парной конструкции map/reduce, используемой в других языках)
findAll	Находит в коллекции все элементы, совпадающие с функциональным литералом
max	Возвращает максимальное значение из данной коллекции
min	Возвращает минимальное значение из данной коллекции

Одна из типичных задач при программировании на Java — необходимость перебрать коллекцию объектов и совершить определенное действие над каждым из них. Например, если вы захотите создать на Java 7 программу, выводящую названия фильмов, то напишите для этого примерно такой код (листинг 8.7)[1].

Листинг 8.7. Вывод коллекции в Java 7

```
List<String> movieTitles = new ArrayList<>();
movieTitles.add("Seven");
movieTitles.add("Snow White");
movieTitles.add("Die Hard");

for (String movieTitle : movieTitles)
{
  System.out.println(movieTitle);
}
```

В Java существует несколько синтаксических приемов, которыми можно пользоваться для уменьшения объема исходного кода, но это не избавляет от следующего факта: все равно приходится вручную перебирать список `List` названий кинофильмов, используя при этом тот или иной цикл.

Работая с Groovy, вы можете применять первоклассную поддержку коллекций — это встроенная функциональность, предназначенная для перебора коллекции (функция `each`), а также функциональный литерал, позволяющий радикально снизить количество исходного кода, который приходится писать. Вместо того чтобы передавать коллекцию в метод, вы фактически передаете метод в коллекцию!

Следующий код выполняет ровно ту же работу, что и код в листинге 8.7, но здесь мы имеем всего две удобочитаемые строки:

```
movieTitles = ["Seven", "SnowWhite", "Die Hard"]
movieTitles.each({x -> println x})
```

В сущности, этот код записывается еще более кратко благодаря неявной переменной `it`, которая может использоваться с одноаргументными функциональными литералами следующим образом[2]:

```
movieTitles = ["Seven", "SnowWhite", "Die Hard"]
movieTitles.each({println it})
```

Как видите, код получился лаконичным и удобочитаемым. Он оказывает такой же наблюдаемый эффект, как и версия для Java 7.

СОВЕТ

Размеры этой главы ограничены. Поэтому, если вас интересуют другие примеры, рекомендуем познакомиться с разделом о коллекциях на сайте Groovy (http://groovy.codehaus.org/JN1015-Collections) или почитать второе издание отличной книги Groovy in Action, которую написали Дирк Кёниг (Dierk Konig), Гийом Лафорж, Пол Кинг (Paul King), Джон Скит (Jon Skeet) и Гамлет Д'Арси (Hamlet D'Arcy) (издательство Manning, 2012).

[1] Нет, мы не скажем, кто из нас выбрал «Белоснежку» в качестве любимого фильма. Но да, это был один из авторов.

[2] Настоящие гуру Groovy отметили бы, что этот код можно сократить даже до одной строки.

Следующая языковая функция Groovy, для работы с которой стоит подготовиться, — это встроенная поддержка регулярных выражений. Поэтому, если у вас еще остался кофе, подзаправьтесь, и идем дальше!

8.4.7. Первоклассная поддержка работы с регулярными выражениями

В Groovy регулярные выражения — встроенная часть языка. Поэтому такие виды деятельности, как обработка текста, протекают гораздо проще, чем в Java. В табл. 8.2 описан синтаксис регулярных выражений, который можно использовать в Groovy, а также приведены эквивалентные конструкции Java.

Таблица 8.2. Синтаксис регулярных выражений Groovy

Метод	Описание и эквивалент из Java
~	Создает паттерн (создается скомпилированный Java-объект Pattern)
=~	Создает обнаружитель совпадений (создается скомпилированный Java-объект Matcher)
==~	Интерпретирует строку (фактически метод Java match() вызывается применительно к Pattern)

Допустим, вы хотите найти частичные совпадения с определенными некорректными данными журнала, которые получили от аппаратного компонента. В частности, вы ищете, встречаются ли вхождения паттерна `1010`, который необходимо преобразовать в `0101`. В Java 7 вы, вероятно, напишете следующий код:

```
Pattern pattern = Pattern.compile("1010");
String input = "1010";
Matcher matcher = pattern.matcher(input);
if (input.matches("1010"))
{
  input = matcher.replaceFirst("0101");
  System.out.println(input);
}
```

В Groovy отдельно взятые строки кода короче, так как объекты `Pattern` и `Matcher` встроены в язык. Вывод (`0101`), разумеется, остается прежним, как показано в следующем коде:

```
def pattern = /1010/
def input = "1010"
def matcher = input =~ pattern
if (input ==~ pattern)
{
  input = matcher.replaceFirst("0101")
  println input
}
```

Groovy целиком поддерживает семантику регулярных выражений по тому же принципу, что и Java. Поэтому при работе в данной области вы приобретаете полную гибкость.

Кроме того, регулярные выражения можно красиво комбинировать с функциональными литералами. Например, вы можете вывести учетные данные о человеке, выполнив синтаксический разбор `String` и получив из строки имя и возраст.

```
("Hazel 1" =~ /(\w+) (\d+)/).each {full, name, age
                                   -> println "$name is $age years old."}
```

А теперь, пожалуй, самое время отдохнуть, так как далее мы переходим к рассмотрению иной тематической области. Речь пойдет об XML-обработке.

8.4.8. Простая XML-обработка

В Groovy существует понятие *«построитель»* (builder) — это абстракция, позволяющая работать с произвольными древовидными структурами с помощью нативного синтаксиса Groovy. Такие структуры могут быть написаны на HTML, XML и JSON. В Groovy понимается необходимость в простой обработке данных такого типа. Для решения подобных задач в языке предоставляются готовые построители.

XML — ШИРОКО ЗЛОУПОТРЕБЛЯЕМЫЙ ЯЗЫК

XML — отличный и лексически богатый язык для обмена данными, но в последнее время им все больше злоупотребляют. Почему? Дело в том, что разработчики программ пытаются обращаться с XML как с языком программирования, а он таковым не является. Язык XML не обладает полнотой по Тьюрингу. Надеемся, что в ваших проектах XML используется по назначению — для обмена данными в человекочитаемом формате.

В этом разделе мы сосредоточимся на XML — общеизвестном формате для обмена данными. Хотя в основной части языка Java (посредством JAXB и JAXP) и во множестве сторонних Java-библиотек (XStream, Xerces, Xalan и т. д.) предоставляются широчайшие возможности для обработки XML, выбрать правильное решение зачастую бывает сложно. Код Java, в котором используются такие решения, может получиться довольно пространным.

В этом разделе мы расскажем, как создавать XML из Groovy, и покажем, как выполнять синтаксический анализ такого XML для обратного его превращения в GroovyBean.

Создание XML

Groovy позволяет без труда создавать XML-документы, например выводить информацию о человеке:

```
<person id='2'>
  <name>Gweneth</name>
  <age>1</age>
</person>
```

Благодаря Groovy можно создать такую XML-структуру с помощью встроенного построителя MarkupBuilder для XML. В листинге 8.8 мы создаем запись person XML.

Листинг 8.8. Создание простого XML

```
def writer = new StringWriter()
def xml = new groovy.xml.MarkupBuilder(writer)
xml.person(id:2) {
  name 'Gweneth'
  age 1
}
println writer.toString()
```

Обратите внимание: здесь начальный элемент person (с атрибутом id, которому присвоено значение 2) просто создается. При этом не требуется определять, что собой представляет объект person. В Groovy не нужно обязательно подкреплять такое создание XML явным GroovyBean, что опять же экономит ваше время и усилия.

В листинге 8.8 приведен совершенно простой пример. Можете поэкспериментировать с ним, изменив тип вывода со StringWriter, а также попробуйте различные построители, например groovy.json.JsonBuilder(), позволяющий мгновенно создавать JSON[1]. Работая с более сложными XML-структурами, вы также приобретаете дополнительную поддержку при оперировании пространствами имен и другими специфическими конструкциями.

Кроме того, вам понадобится возможность осуществлять обратную операцию: считывать XML и преобразовывать его в полезный GroovyBean.

Синтаксический разбор XML

Groovy может выполнять синтаксический анализ входящего XML несколькими способами. В табл. 8.3 перечислены три метода. Данные для таблицы взяты из официальной документации Groovy (http://docs.codehaus.org/display/GROOVY/Processing+XML).

Таблица 8.3. Методы синтаксического анализа Groovy

Метод	Описание
XMLParser	Поддерживает выражения GPath для XML-документов
XMLSlurper	Похож на XMLParser, но работает по принципу ленивой загрузки
DOMCategory	Низкоуровневый синтаксический анализ объектной модели документа с определенной синтаксической поддержкой

Пользоваться всеми тремя довольно несложно, но в этом разделе мы сосредоточимся на работе с XMLParser.

[1] Дастин написал об этом отличный пост под названием «Groovy 1.8 знакомит Groovy с JSON» в своем блоге Inspired by actual events по адресу http://marxsoftware.blogspot.com/

ПРИМЕЧАНИЕ

GPath — это язык выражений. О нем можно почитать в документации Groovy по адресу http://groovy.codehaus.org/GPath.

Возьмем XML-представление `Gweneth` (лицо), полученное в листинге 8.8, и преобразуем его обратно в `Person` GroovyBean. Эта операция показана в листинге 8.9.

Листинг 8.9. Синтаксический разбор XML с помощью XMLParser

```
class XmlExample {
  static def PERSON =
  """
  <person id='2'>
    <name>Gweneth</name>
    <age>1</age>
  </person>
  """
}

class Person {def id; def name; def age}

def xmlPerson = new XmlParser().
                    parseText(XmlExample.PERSON)

Person p = new Person(id: xmlPerson.@id,
                      name: xmlPerson.name.text(),
                      age: xmlPerson.age.text())

println "${p.id}, ${p.name}, ${p.age}"
```

Сразу немного сократим себе работу и отобразим XML-документ в коде так, чтобы он уже был указан в `CLASSPATH` ❶. Далее приступим к самой работе и воспользуемся методом `parseText()`, относящимся к `XMLParser`. Таким образом, мы считываем XML-данные ❷. После этого создается новый объект `Person` и ему присваиваются значения ❸. Наконец, `Person` выводится на экран, чтобы вы могли сами проверить код.

На этом завершается наш вводный курс в Groovy. Вам уже наверняка не терпится опробовать какие-нибудь функции этого языка в ваших проектах на Java! В следующем разделе мы расскажем, как обеспечить взаимодействие между Java и Groovy. Вы сделаете важнейший шаг на пути к профессиональной Java-разработке и станете многоязычным программистом, умеющим применять альтернативные языки на виртуальной машине Java.

8.5. Взаимодействие между Groovy и Java

Этот раздел обманчиво краток, но его важность невозможно переоценить! Если вы по порядку изучали весь предыдущий материал, то именно в этом разделе совершите качественный прорыв и станете больше чем просто Java-разработчиком, умеющим обращаться с виртуальной машиной Java. Основательный Java-разработчик должен

уметь работать на JVM с несколькими языками, дополняющими Java. И начинать в данном случае лучше как раз с Groovy!

Сначала вспомним, насколько просто вызвать Java из Groovy. После этого мы проработаем три распространенных способа объединения Java с Groovy — с помощью сценариев `GroovyShell`, `GroovyClassLoader` и `GroovyScriptEngine`.

Итак, начнем с вызовов Java из Groovy.

8.5.1. Вызов Java из Groovy

Помните, мы говорили о том, что для вызова Java из Groovy нужно всего лишь указать JAR в `CLASSPATH` и использовать стандартную запись `import`? Вот пример импортирования классов из пакета `org.joda.time`, относящегося к популярной библиотеке Joda, управляющей датой и временем[1]:

```
import org.joda.time.*;
```

Вы будете использовать классы так же, как делали бы это в Java. Следующий фрагмент кода выводит числовое представление текущего месяца:

```
DateTime dt = new DateTime()
int month = dt.getMonthOfYear()
println month
```

Хм-м-м, неужели так просто?

Адмирал Акбар уверен: «Это ловушка!»[2]

Да нет, мы шутим. Нет никакой ловушки, все действительно настолько просто. Поэтому сразу перейдем к более сложному случаю. Вызывать Groovy из Java и получать значимый результат будет немного труднее.

8.5.2. Вызов Groovy из Java

Для вызова Groovy из приложения, написанного на Java, нужно поместить JAR Groovy и другие необходимые JAR-файлы в `CLASSPATH` приложения Java, так как они являются зависимостями времени исполнения.

СОВЕТ

Просто добавьте в CLASSPATH файл GROOVY_HOME/embeddable/groovy-all-1.8.6.jar.

Можно вызывать код Groovy из приложения Java с помощью:

- сценарной среды Bean Scripting Framework (BSF) — описана в спецификации JSR 223;
- `GroovyShell`;
- `GroovyClassLoader`;
- `GroovyScriptEngine`;
- встроенной консоли Groovy.

---

[1] Joda де-факто является библиотекой для управления датой и временем в Java, до выпуска Java 8.

[2] Поклонники «Звездных войн», несомненно, узнали этот интернет-мем!

В этом разделе мы сосредоточимся на наиболее широко используемых способах (GroovyShell, GroovyClassLoader и GroovyScriptEngine). Начнем с простейшего — GroovyShell.

GroovyShell

GroovyShell можно временно активизировать для быстрого вызова к Groovy и интерпретации определенных выражений или сценарного кода. Например, некоторые разработчики, предпочитающие применяемую в Groovy обработку числовых литералов, могут выполнять вызовы к GroovyShell для проведения математических операций. Следующий код Java возвращает значение 10.4, применяя при этом числовые литералы Groovy для выполнения сложения (листинг 8.10).

Листинг 8.10. Использование GroovyShell для выполнения Groovy из Java

```
import groovy.lang.GroovyShell;
import groovy.lang.Binding;
import java.math.BigDecimal;

public class UseGroovyShell {

  public static void main(String[] args) {
    Binding binding = new Binding();
    binding.setVariable("x", 2.4);
    binding.setVariable("y", 8);
    GroovyShell shell = new GroovyShell(binding);
    Object value = shell.evaluate("x + y");
    assert value.equals(new BigDecimal(10.4));
  }

}
```

Задаем связывание для оболочки

Интерпретируем и возвращаем выражение

Такой пример использования GroovyShell соответствует варианту, когда вам требуется выполнить фрагмент кода на Groovy. А что делать, если требуется обеспечить взаимодействие с полнофункциональным классом Groovy? В таком случае попытайтесь воспользоваться GroovyClassLoader.

GroovyClassLoader

С точки зрения разработчика, GroovyClassLoader функционально во многом напоминает ClassLoader из Java. Вы ищете класс и метод, который хотите вызвать, — и просто вызываете его!

В следующем фрагменте кода содержится простой класс CalculateMax с методом getMax. Этот метод, в свою очередь, использует встроенную в Groovy функцию max. Чтобы запустить его из Java посредством GroovyClassLoader, необходимо создать файл Groovy (CalculateMax.groovy) с таким исходным кодом:

```
class CalculateMax {
  def Integer getMax(List values) {
    values.max();
  }
}
```

Итак, теперь у вас есть сценарий Groovy, который вы хотите выполнить, и можно вызвать его из Java. В листинге 8.11 показан вызов, написанный на Java и обращенный к функции `CalculateMax` `getMax`, возвращающей `10` как максимальное значение всех переданных аргументов.

Листинг 8.11. Использование `GroovyClassLoader` для выполнения Groovy из Java

```
import java.io.File;
import java.io.IOException;
import java.util.ArrayList;
import groovy.lang.GroovyClassLoader;
import groovy.lang.GroovyObject;
import org.codehaus.groovy.control.CompilationFailedException;

public class UseGroovyClassLoader {

  public static void main(String[] args) {
    GroovyClassLoader loader = new GroovyClassLoader();

    try {
      Class<?> groovyClass = loader.parseClass(
               new File("CalculateMax.groovy"));

      GroovyObject groovyObject = (GroovyObject)
               groovyClass.newInstance();

      ArrayList<Integer> numbers = new ArrayList<>();
      numbers.add(new Integer(1));
      numbers.add(new Integer(10));
      Object[] arguments = {numbers};

      Object value =
        groovyObject.invokeMethod("getMax", arguments);
      assert value.equals(new Integer(10));
    }
    catch (CompilationFailedException | IOException | InstantiationException
           | IllegalAccessException e) {
      System.out.println(e.getMessage());
    }
  }
}
```

Подготовка ClassLoader

Получение класса Groovy

Получение экземпляра класса Groovy

Подготовка аргументов

Вызов метода Groovy

Такая техника, вероятно, окажется полезна, если у вас будет несколько классов на Groovy, к которым вы собираетесь направлять вызовы из Java. Но если у вас значительно больше кода на Groovy, к которому необходимо обеспечить доступ, то рекомендуем использовать полнофункциональный `GroovyScriptEngine`.

GroovyScriptEngine

Работая с `GroovyScriptEngine`, вы указываете URL или местоположение каталога, в котором находится ваш код на Groovy. Затем движок при необходимости загружает и компилирует сценарии из этого кода, в частности, это касается зависимых

сценариев. Например, если вы изменяете сценарий B, а сценарий A от него зависит, то движок перекомпилирует весь затронутый код.

Допустим, у вас есть сценарий Groovy (`Hello.groovy`), определяющий простую инструкцию `"Hello"`, а за ней следует имя (параметр, который вы хотите получить от вашего приложения, написанного на Java):

```
def helloStatement = "Hello ${name}"
```

После этого ваше приложение на Java опирается на движок `GroovyScriptEngine`, чтобы задействовать `Hello.groovy`, и выводит приветствие, как показано в листинге 8.12.

Листинг 8.12. Использование `GroovyScriptEngine` для выполнения Groovy из Java

```
import groovy.lang.Binding;
import groovy.util.GroovyScriptEngine;
import groovy.util.ResourceException;
import groovy.util.ScriptException;
import java.io.IOException;

public class UseGroovyScriptEngine {

  public static void main(String[] args)
  {
    try {
      String[] roots = new String[] {"/src/main/groovy"};
      GroovyScriptEngine gse =
        new GroovyScriptEngine (roots);

      Binding binding = new Binding();
      binding.setVariable("name", "Gweneth");

      Object output = gse.run("Hello.groovy", binding);
      assert output.equals("Hello Gweneth");
    }
    catch (IOException | ResourceException | ScriptException e) {
      System.out.println(e.getMessage());
    }
  }
}
```

Задаем корни

Инициализируем движок

Запускаем сценарий

Помните, что теперь вы можете с легкостью изменить любой сценарий Groovy, отслеживаемый `GroovyScriptEngine`. Например, если бы вы хотели придать сценарию `Hello.groovy` такой вид:

```
def helloStatement = "Hello ${name}, it's Groovy, baby, yeah!"
```

то в следующий раз при запуске соответствующего кода на Java использовалось бы уже новое, более длинное сообщение. Так вы обеспечиваете для Java-приложений динамическую гибкость, на которую раньше никак не могли рассчитывать. Эта возможность может оказаться неоценимой, например, при отладке рабочего кода, изменении свойств системы во время исполнения и не только!

На этом завершается наше введение в Groovy. Лиха беда начало!

8.6. Резюме

Groovy обладает замечательными возможностями, благодаря которым становится отличным языком для использования в паре с Java. Можно писать синтаксис, очень напоминающий Java, но в то же время оформлять более краткий код, идентичный Java-подобному коду с логической точки зрения. При достижении лаконичности не требуется жертвовать удобочитаемостью, и Java-разработчикам не составляет труда освоиться с новым сокращенным синтаксисом, обслуживающим операции с коллекциями, обработку ссылок на нуль и обращение с компонентами GroovyBean. Java-разработчика в Groovy подстерегают некоторые ловушки, но выше мы проработали все достаточно распространенные случаи. Надеемся, что вы сможете поделиться свежеприобретенными знаниями с вашими коллегами.

В Groovy имеется несколько новых языковых свойств, которые многие Java-разработчики надеются однажды увидеть и в Java. Самый сложный для усвоения, но вместе с тем и самый мощный из таких элементов — функциональные литералы, серьезный инструмент программирования, который (в частности) помогает с легкостью оперировать коллекциями. Разумеется, коллекции в Groovy обеспечены первоклассной поддержкой. Вы можете их создавать, изменять и вообще легко обращаться с ними благодаря краткому и удобному синтаксису.

Большинству Java-разработчиков приходится либо генерировать XML, либо осуществлять его синтаксический анализ. Здесь вам также может пригодиться Groovy, поскольку в этом языке работа с XML значительно упрощается благодаря встроенной в язык поддержке такой разметки.

Итак, вы сделали первые шаги на пути к тому, чтобы стать многоязычным программистом. Вы освоили несколько техник, позволяющих интегрировать код Java и Groovy так, чтобы можно было решать стоящие перед вами проблемы программирования с применением обоих этих языков.

На этом ваше путешествие не заканчивается. Мы вновь будем активно пользоваться Groovy в главе 13, где поговорим о быстрой веб-разработке.

Далее нас ждет глава о Scala — другом языке для виртуальной машины Java. Scala вызвал небольшой переполох в среде программистов. Этот язык одновременно является и объектно-ориентированным, и функциональным, поэтому заслуживает внимания со стороны всех, кому при программировании приходится решать сложные современные задачи.

9 Язык Scala — мощный и лаконичный

В этой главе:

- Scala — не Java;
- синтаксис Scala и более функциональный стиль;
- сопоставимые выражения и паттерны;
- система типов и коллекции Scala;
- параллельное программирование на Scala с помощью акторов.

Scala — это язык, созданный в среде ученых-информатиков и исследователей языков программирования. Он уже получил определенное распространение, так как обладает очень мощной системой типов и продвинутыми возможностями, которые оказались очень полезны для элитных команд.

В настоящее время Scala вызывает большой интерес, но пока рано говорить, сможет ли этот язык полностью пронизать экосистему Java и составить Java конкуренцию в качестве основного языка для разработки.

Мы полагаем, что в Scala будут появляться новые команды и некоторые проекты будут писаться именно на нем. Мы также предполагаем, что множество разработчиков будут вплетать Scala в какие-нибудь проекты в течение ближайших 3–4 лет. Это означает, что основательный Java-разработчик должен иметь представление о Scala и уметь определять, подходит ли этот язык для его проектов.

Разработчик, переходящий с Java к работе со Scala, должен иметь в виду некоторые особенности. Самое главное, что нужно помнить: Scala — это не Java.

Ну да, это очевидно, скажете вы. В конце концов, все языки разные, поэтому, разумеется, Scala отличается от Java. Но, как было указано в главе 7, некоторые языки имеют больше общих черт, другие — меньше. Когда мы знакомили вас с языком Groovy в главе 8, мы подчеркивали сходство между Groovy и Java. Возможно, это пригодилось вам при изучении вашего первого языка для JVM, не являющегося Java.

В этой главе мы будем действовать иначе — для начала подчеркнем те черты Scala, которые составляют его специфику. Считайте это небольшой экскурсией под названием «Scala в естественной среде обитания» — в ходе нее мы покажем, как писать на Scala, а не на построчном переводе с Java на Scala. Далее мы обсудим характеристики проектов, то есть покажем, как определить, подходит ли Scala для вашего проекта. Затем мы рассмотрим некоторые синтаксические инновации Scala,

позволяющие писать на Scala красивый и лаконичный код. Потом поговорим о том, как в Scala организована объектная ориентация, после чего изучим раздел о коллекциях и структурах данных. Завершим главу разделом о том, как в Scala организован параллелизм, и изучим мощную модель акторов, действующую в этом языке.

Изучая наиболее замечательные свойства Scala, мы постепенно расскажем о синтаксисе (и обо всех других необходимых концепциях). Scala — довольно большой язык по сравнению с Java. Здесь больше базовых концепций, а также синтаксических тонкостей, которые необходимо учитывать. Это означает, что чем больше кода на Scala вы будете просматривать, тем больше языковых свойств у вас получится усвоить самостоятельно.

Сделаем краткий обзор тем, с которыми предстоит плотнее познакомиться по ходу этой главы. Это поможет вам привыкнуть к характерному синтаксису Scala и подготовиться к дальнейшему чтению.

9.1. Быстрый обзор Scala

Вот основные моменты, которые мы хотели бы предвосхитить:

- краткость языка Scala и действующая в нем мощная система выведения типов;
- сопоставимые выражения и связанные с ними концепции — паттерны и `case`-классы;
- параллелизм в Scala, основанный на системе сообщений и акторов, а не на блокировках, как в классическом коде на Java.

Эти темы еще не позволяют получить представление обо всем языке или стать профессиональным Scala-разработчиком. Мы просто подогреем ваше любопытство и покажем несколько конкретных примеров, в которых удобно применять Scala. Чтобы пойти дальше и исследовать язык более подробно, можете почитать специализированные онлайн-ресурсы или книгу, в которой подробно рассматривается язык Scala. В качестве примера такой книги можем порекомендовать труд Scala in Depth Джошуа Сьюреса (Joshua Suereth) (издательство Manning, 2012).

Важнейшая и наиболее заметная особенность Scala, о которой мы хотели упомянуть, — это лаконичность синтаксиса. Поговорим об этом.

9.1.1. Scala — лаконичный язык

Scala — это компилируемый статически типизированный язык. Слыша такую характеристику, некоторые разработчики ожидают увидеть примерно такой же пространный код, как и на Java. К счастью, Scala отличается гораздо большей краткостью. Иногда он даже производит впечатление сценарного языка. Работая на Scala, программист может действовать гораздо быстрее и продуктивнее, развивая такую скорость, которая обычно характерна лишь для динамически типизированных языков.

Рассмотрим очень простой образец кода и выясним, как в Scala строится работа с конструкторами и классами. Например, попробуем написать простой класс, моделирующий поток денежной наличности. Для этого пользователь должен сообщить сумму наличных денег и валюту. В Scala это делается так:

```
class CashFlow(amt : Double, curr : String) {
  def amount() = amt
  def currency() = curr
}
```

Этот класс состоит всего из четырех строк кода (одна из которых содержит лишь закрывающую скобку). Тем не менее он предоставляет методы-получатели (но не предоставляет методы-установщики) для параметров, а также один конструктор. Код обеспечивает недюжинную отдачу при такой-то краткости. Эквивалентный код на Java получается гораздо длиннее:

```
public class CashFlow {
  private final double amt;
  private final String curr;

  public CashFlow(double amt, String curr) {
    this.amt = amt;
    this.curr = curr;
  }

  public double getAmt() {
    return amt;
  }

  public String getCurr() {
    return curr;
  }
}
```

В версии на Java видим гораздо больше повторений, чем в Scala, и такие повторения — лишь одна из причин пространности.

Философия Scala состоит в том, чтобы не вынуждать разработчика повторять информацию. Поэтому в любой ситуации в окне интегрированной среды разработки вы сможете уместить больше кода Scala, чем аналогичного кода Java. А значит, при обработке сложного логического фрагмента разработчик сможет ухватить взглядом больше информации на Scala, чем на Java, и, вероятно, получит дополнительные наводки, помогающие понять код.

ХОТИТЕ СЭКОНОМИТЬ 1500 ДОЛЛАРОВ?

Вариант класса `CashFlow`, написанный на Scala, почти на 75 % короче, чем аналогичный код на Java. Существует оценка, согласно которой стоимость кода составляет 32 доллара за строку в год. Если предположить, что показанный пример кода будет использоваться на протяжении пяти лет, то версия Scala будет стоить на 1500 долларов в год дешевле, чем код Java. Такая экономия получается потому, что более краткий код легче поддерживать.

На данном этапе самое время рассмотреть синтаксические особенности, которые заметны в предыдущем примере.

- Определение класса (в контексте его параметров) и конструктора класса в данном случае — одно и то же. Scala допускает использование дополнительных «вспомогательных конструкторов», о которых мы поговорим ниже.
- Классы по умолчанию являются общедоступными, поэтому нет необходимости использовать ключевое слово `public`.
- Возвращаемые типы методов получаются путем выведения типов, но в условии `def` необходимо ставить знак равенства. В таком случае это условие должно сообщить компилятору о том, что необходимо вывести тип.
- Если тело метода состоит из единственной инструкции (или выражения), эту инструкцию не обязательно заключать в скобки.
- В Scala отсутствует феномен примитивных типов, которые имеются в Java. Числовые типы являются объектами.

Этим краткость не ограничивается. Она заметна даже в такой программе, как классический Hello World:

```
object HelloWorld {
  def main(args : Array[String]) {
    val hello = "Hello World!"

    println(hello)
  }
}
```

Scala обладает чертами, которые позволяют сократить шаблонный код даже в простейшем примере.

- Ключевое слово `object` приказывает компилятору Scala сделать этот класс одиночкой (синглтоном).
- Вызов `println()` не требуется квалифицировать (благодаря стандартным импортам).
- К методу `main()` не нужно применять ключевые слова `public` или `static`.
- Не приходится объявлять тип `hello` — компилятор просто получает его сам.
- Не требуется объявлять возвращаемый тип `main()` — он автоматически является `Unit` (это применяемый в Scala эквивалент `void`).

В этом примере есть и несколько других важных синтаксических моментов.

- В отличие от Java и Groovy, тип переменной указывается после ее имени.
- Квадратные скобки указываются в Scala для обозначения обобщенного типа. Поэтому тип `args` указывается как `Array[String]`, а не как `String[]`.
- `Array` — это подлинный обобщенный тип.
- Типы коллекций обязательно должны быть обобщенными (эквивалент необработанного типа Java в Scala отсутствует).
- Точки с запятой практически всегда необязательны.

- `Val` — это эквивалент `final`-переменной из языка Java — так объявляется неизменяемая переменная.
- Точки входа в приложение на Scala всегда содержатся в `object`.

В следующих разделах мы подробнее объясним, как работает синтаксис, который мы уже успели рассмотреть. Кроме того, мы рассмотрим еще несколько инноваций Scala, позволяющих щадить клавиатуру. Мы также обсудим подход Scala к функциональному программированию, который довольно хорошо помогает писать краткий код. А пока остановимся на одной из наиболее мощных «нативных возможностей» Scala.

9.1.2. Сопоставимые выражения

В Scala есть очень мощная конструкция — выражение `match`. Простые случаи применения `match` напоминают использование оператора `switch` в Java, но `match` встречается и в гораздо более выразительных формах. Форма выражения `match` зависит от структуры выражения, указанного в условии `case`. В Scala различные типы `case`-условий называются «паттернами», но необходимо отметить, что речь идет не о тех самых паттернах, которые применяются в работе с регулярными выражениями (правда, как будет показано ниже, паттерн регулярного выражения вполне можно использовать в выражении `match`).

Рассмотрим знакомый пример. В следующем фрагменте кода показан пример использования строк со `switch`, который мы уже затрагивали в подразделе 1.3.1. Перед вами — вольный перевод этого кода на язык Scala.

```
var frenchDayOfWeek = args(0) match {
  case "Sunday"    => "Dimanche"
  case "Monday"    => "Lundi"
  case "Tuesday"   => "Mardi"
  case "Wednesday" => "Mercredi"
  case "Thursday"  => "Jeudi"
  case "Friday"    => "Vendredi"
  case "Saturday"  => "Samedi"
  case _           => "Error: '"+ args(0) +"' is not a day of the week"
}
println(frenchDayOfWeek)
```

В данном примере показаны лишь два основных паттерна — паттерн с константами, обозначающими дни недели, и паттерн _, в котором обрабатывается случай, заданный по умолчанию. С другими примерами паттернов мы встретимся далее в этой главе.

С точки зрения языкового пуризма можно отметить, что синтаксис Scala чище и правильнее синтаксиса Java как минимум в двух отношениях.

- В стандартном случае не требуется применять отдельное ключевое слово.
- В отличие от Java, обработчик выполнения условия не передает управление следующему обработчику, потому нет необходимости использовать ключевое слово `break`.

Вот еще синтаксические детали, на которые следует обратить внимание в этом примере.

- Ключевое слово `var` используется для объявления изменяемой (нефинальной) переменной. Старайтесь не применять его без необходимости, но иногда оно действительно требуется.
- При доступе к массиву используются круглые скобки; например `args(0)` для первого аргумента к `main()`.
- Всегда следует включать случай по умолчанию. Если Scala не сможет найти совпадения хотя бы с одним случаем во время исполнения, то будет выдана ошибка `MatchError`. Нам бы этого совсем не хотелось.
- Scala поддерживает синтаксис *непрямого вызова методов*, поэтому запись `args(0) match { ... }` эквивалентна `args(0).match({ ... })`.

Пока все ясно. Конструкция `match` напоминает немного более чистый вариант `switch`. Но это наиболее Java-подобный из многих возможных паттернов. В Scala есть множество языковых конструкций, в которых используются разнообразные паттерны. Например, рассмотрим типизированный паттерн, с помощью которого удобно обрабатывать данные неизвестного происхождения. При этом не требуется выполнять неудобные приведения либо проверки `instanceof` в стиле Java.

```
def storageSize(obj: Any) = obj match {
  case s: String => s.length
  case i: Int    => 4
  case _         => -1
}
```

Этот очень простой метод принимает значение типа `Any` (то есть значения, тип которых неизвестен). Потом паттерны применяются для раздельной обработки значений, относящихся к типам `String` и `Int`. В каждом случае к значению привязывается временный псевдоним. Это делается для того, чтобы при необходимости можно было вызывать методы применительно к этому значению.

Синтаксическая форма, очень напоминающая паттерн работы с переменными, используется в коде Scala для обработки исключений. Рассмотрим немного адаптированный пример кода для программирования классов. Пример взят из фреймворка ScalaTest, с которым мы встретимся в главе 11:

```
def getReporter(repClassName: String, loader: ClassLoader): Reporter = {
  try {
    val reporterCl: java.lang.Class[_] = loader.loadClass(repClassName)
    reporterCl.newInstance.asInstanceOf[Reporter]
  }
  catch {
    case e: ClassNotFoundException => {
      val msg = "Can't load reporter class"
      val iae = new IllegalArgumentException(msg)
      iae.initCause(e)
      throw iae
    }
```

```
    case e: InstantiationException => {
      val msg = "Can't instantiate Reporter"
      val iae = new IllegalArgumentException(msg)
      iae.initCause(e)
      throw iae
    }
    ...
  }
}
```

В методе `getReporter()` мы пытаемся загрузить специальный отчетный класс (это делается с применением рефлексии). Класс сообщает о состоянии тестовой последовательности по мере ее выполнения. При загрузке и инстанцировании классов кое-какие вещи могут пойти неправильно, поэтому на этапе выполнения вы страхуетесь от ошибок с помощью блока `try-catch`.

Блоки `catch` очень напоминают выражения `match`, где выполняется сопоставление (match) с типом наблюдаемого исключения. Эту идею можно развить с применением концепции `case`-классов, которую мы обсудим далее.

9.1.3. Case-классы

Выражения `match` в языке Scala особенно полезны при использовании вместе с `case`-классами (такие классы можно считать объектно-ориентированными расширениями концепции перечислений). Рассмотрим пример — сигнал тревоги, указывающий, что температура слишком высока:

```
case class TemperatureAlarm(temp : Double)
```

В этой единственной строке кода определен полноценный `case`-класс. Класс Java, являющийся его примерным аналогом, будет выглядеть примерно так:

```
public class TemperatureAlarm {
  private final double temp;
  public TemperatureAlarm(double temp) {
    this.temp = temp;
  }

  public double getTemp() {
    return temp;
  }

  @Override
  public String toString() {
    return "TemperatureAlarm [temp=" + temp + "]";
  }

  @Override
  public int hashCode() {
    final int prime = 31;
    int result = 1;
    long temp;
    temp = Double.doubleToLongBits(this.temp);
```

```
    result = prime * result + (int) (temp ^ (temp >>> 32));
    return result;
  }

  @Override
  public boolean equals(Object obj) {
    if (this == obj)
        return true;
    if (obj == null)
       return false;
    if (getClass() != obj.getClass())
       return false;
    TemperatureAlarm other = (TemperatureAlarm) obj;
    if (Double.doubleToLongBits(temp) !=
        Double.doubleToLongBits(other.temp))
        return false;
    return true;
  }
}
```

Добавив единственное ключевое слово `case`, мы приказываем компилятору Scala создать эти дополнительные методы. Кроме того, так создается еще и множество вспомогательных методов. Как правило, разработчик не пользуется такими дополнительными методами напрямую. Но они нужны для того, чтобы обеспечивать во время исполнения поддержку для определенных функций Scala. В общем, дополнительные методы Scala позволяют использовать `case`-класс «по-скаловски».

`Case`-классы создаются без ключевого слова `new`, вот так:

```
val alarm = TemperatureAlarm(99.9)
```

Это еще более подчеркивает схожесть `case`-классов с «параметризованными перечислениями» или с определенной формой типа значения.

РАВЕНСТВО В SCALA

В Java символ двойного равенства (==) означает обычное равенство, но в Scala это не так. В Scala == эквивалентно `equals()`. Для указания обычного равенства можно использовать запись ===. В `case`-классах есть метод `.equals()`, возвращающий `true` тогда и только тогда, когда два экземпляра имеют абсолютно тождественные значения для всех параметров.

`Case`-классы естественно вписываются в паттерн конструктора, как показано ниже:

```
def ctorMatchExample(sthg : AnyRef) = {
  val msg = sthg match {
    case Heartbeat => 0
    case TemperatureAlarm(temp) => "Tripped at temp "+ temp
    case _ => "No match"
  }
  println(msg)
}
```

Теперь кратко рассмотрим последнюю особенность, на которую хотели обратить внимание в ходе нашей вводной экскурсии по Scala. Речь пойдет об акторах — особых конструктах, обеспечивающих параллелизм.

9.1.4. Акторы

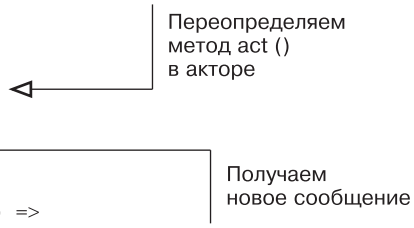
Акторы — это альтернативная сущность, с помощью которой в Scala реализуется параллельное программирование. Акторы обеспечивают асинхронную модель параллелизма, основанную на передаче сообщений между исполняемыми единицами кода. Это своеобразная высокоуровневая модель параллелизма, которую многие разработчики находят более простой, чем по умолчанию разделяемая модель с блокировками, применяемая в Java. Но необходимо отметить, что память Scala построена на той же низкоуровневой модели памяти, что и Java, — JMM.

Рассмотрим пример. Предположим, что ветеринар, о котором мы говорили в главе 4, должен наблюдать за состоянием здоровья (в частности, за температурой) питомцев, попавших в клинику. Можно предположить, что аппаратные температурные датчики будут отсылать сообщения со своими показаниями в центральную отслеживающую программу.

В Scala такую конструкцию можно смоделировать с помощью акторного класса `TemperatureMonitor`. Он рассчитан на прием сообщений двух разновидностей — стандартного периодического контрольного сообщения и сообщения `TemperatureAlarm`. Второе сообщение будет принимать параметр, указывающий температуру, при которой сработала тревожная сигнализация. Эти классы приведены в листинге 9.1.

Листинг 9.1. Простой обмен сообщениями с актором

```
case object Heartbeat
case class TemperatureAlarm(temp : Double)

import scala.actors._

class TemperatureMonitor extends Actor {
  var tripped : Boolean = false
  var tripTemp : Double = 0.0

  def act() = {
    while (true) {
      receive {
        case Heartbeat => 0
        case TemperatureAlarm(temp) =>
        tripped = true
        tripTemp = temp
        case _ => println("No match")
      }
    }
  }
}
```



Возможны три независимых случая, в которых мы можем получить отклик от отслеживающего актора (посредством `receive`). Первый вариант отклика — это периодическое контрольное сообщение, означающее, что все идет хорошо. Поскольку этот `case`-класс не принимает параметров, он фактически является одиночкой. Его можно назвать `case`-объектом. Получая такое сообщение, актор не должен предпринимать никакого действия.

Если вы получите сообщение `TemperatureAlarm`, то актор сохранит температуру, при которой сработал сигнал тревоги. Логично, что у ветеринара может быть другой фрагмент кода, периодически проверяющий `TemperatureMonitor`, чтобы узнать, не возникла ли тревожная ситуация.

Наконец, есть универсальный случай. Это своего рода инструмент-ловушка, перехватывающий все неожиданные сообщения, которые могут просочиться в среду актора. Без такой ловушки актор будет выбрасывать исключение, получив любое сообщение непредусмотренного типа. Мы вернемся к акторам в конце этой главы и рассмотрим их подробнее, но параллелизм в Scala — это гигантская тема, в нашей книге мы можем лишь слегка очертить ее.

Итак, мы галопом пронеслись по наиболее замечательным особенностям Scala. Надеемся, что некоторые из упомянутых возможностей вас уже заинтересовали. В следующем разделе мы подробнее остановимся на причинах, по которым вы можете выбрать (или не выбрать) язык Scala для реализации какой-то части вашего проекта.

9.2. Подходит ли Scala для моего проекта?

Принимая решение использовать альтернативный язык в своем проекте, разрабатываемом на Java, всегда нужно опираться на обоснованные рассуждения и конкретные доказательства. В этом разделе предлагаем поразмыслить о таких доводах и подумать, насколько они применимы к вашему проекту.

Начнем с краткого сравнения Scala и Java, а потом расскажем, когда и где следует начинать использовать Scala. Завершая этот небольшой раздел, мы рассмотрим некоторые характерные признаки, свидетельствующие о том, что Scala не является оптимальным языком для вашего проекта.

9.2.1. Сравнение Scala и Java

Основные различия между двумя этими языками обобщены в табл. 9.1. Поверхностной областью языка программирования называется количество ключевых слов и самостоятельных языковых конструкций, которые требуется освоить практикующему разработчику, чтобы продуктивно работать с конкретным языком.

Таблица 9.1. Сравнение Java и Scala

Черта	Java	Scala
Система типов	Статическая, довольно пространная	Статическая, сильно завязанная на выведении типов
Уровень в пирамиде многоязычного программирования	Стабильный	Стабильный, динамический
Модель параллелизма	Основанная на блокировках	Основанная на акторах
Функциональное программирование	Требует особого стиля написания кода, неестественного для языка	Встроенная поддержка, естественно вписывается в язык
Поверхностная область	Небольшая/средняя	Большая/очень большая
Стиль синтаксиса	Простой, правильный, относительно многословный	Гибкий, лаконичный, много особых случаев

Эти отличия позволяют понять, по каким причинам язык Scala может быть интересен основательному Java-разработчику в качестве альтернативного языка для реализации определенных проектов или компонентов. Подробнее обсудим, как лучше приступать к внедрению Scala в ваш проект.

9.2.2. Когда и каким образом приступать к использованию Scala

Как мы говорили в главе 7, приступать к внедрению нового языка в уже реализуемом проекте лучше в таких областях, которые характеризуются низкой степенью риска. Примером такого нерискованного участка может быть фреймворк для тестирования ScalaTest, о котором мы поговорим в главе 11. Если эксперимент со Scala не удастся, то вы потеряете на нем лишь рабочее время специалиста (возможно, удастся спасти тесты компонентов и превратить их в обычные JUnit-тесты).

Вообще, такой компонент, на котором будет удобно обкатать Scala в уже существующем проекте, должен иметь большинство или все следующие характеристики:

- можно с достаточной степенью уверенности оценить усилия, требуемые на проведение работ;
- существует ограниченная, хорошо определенная проблемная область;
- у вас есть четко поставленные требования;
- известны четкие требования по взаимодействию данного компонента с другими частями системы;
- вы нашли разработчиков, которые хотят изучить новый язык.

Если вы подыскали такую область, можете переходить к реализации вашего первого компонента на Scala. Следующие рекомендации будут очень полезны в тех случаях, когда необходимо гарантировать, что исходный компонент не выходит за установленные рамки:

- потренируйтесь для начала на очень маленьком фрагменте;
- на раннем этапе протестируйте взаимодействие с другими Java-компонентами;

- сформулируйте оценочные критерии (в зависимости от стоящих перед вами требований), позволяющие определить успешность или неуспешность первой попытки;
- сформулируйте резервный план на случай, если первая попытка провалится;
- учтите в бюджете дополнительное время, требуемое на рефакторинг нового компонента.

Другой ракурс, под которым также будет целесообразно оценить Scala, заключается в проверке: нет ли очевидных фактов, свидетельствующих, что этот язык не слишком хорошо подходит для вашего проекта.

9.2.3. Признаки, указывающие, что Scala может не подойти для вашего проекта

Существует несколько признаков, свидетельствующих, что Scala плохо подходит для конкретного проекта. Если в вашей команде заметны один или несколько следующих признаков, то нужно еще раз подумать, стоит ли на данном этапе внедрять Scala в ваш проект. Если два и более таких признака очевидны — это большой повод для беспокойства.

- Сопротивление или отсутствие поддержки со стороны разработчиков и других специалистов, которые должны участвовать в поддержке вашего проекта.
- Отсутствие в команде выраженного желания к изучению Scala.
- Несхожие или радикально противоположные взгляды на проблему в команде.
- Отсутствие поддержки со стороны старших разработчиков.
- Очень жесткие сроки (отсутствие времени на изучение языка).

Еще один фактор, требующий особого внимания, — глобальное распределение вашей команды. Если ваши специалисты, которым придется разрабатывать (или поддерживать) код на Scala, работают в разных местах, то стоимость и трудоемкость переучивания разработчиков на Scala возрастет.

Теперь, когда мы обсудили механизм введения Scala в ваш проект, рассмотрим синтаксические особенности этого языка. Мы сосредоточимся на тех моментах, которые облегчают жизнь Java-разработчика, расскажем, как писать сравнительно краткий код с меньшим количеством шаблонных элементов и без навязшего в зубах многословия.

9.3. Как вновь сделать код красивым с помощью Scala

Мы начнем этот раздел со знакомства с компилятором Scala и интерактивной средой (REPL). Потом поговорим о выведении типов, объявлении методов (объявление методов в Scala серьезно отличается от той же процедуры в Java). При использовании двух этих возможностей вместе можно значительно сократить количество шаблонного кода и повысить производительность труда.

Мы поговорим о том, как в Scala поставлена работа с пакетами, затронем более мощную инструкцию `import` этого языка, а потом подробно рассмотрим циклы и управляющие структуры. В их основе лежит совершенно иная традиция программирования, нежели та, что применяется в Java, и мы воспользуемся этим, чтобы обсудить некоторые методики функционального программирования, присутствующие в Scala. В частности, мы рассмотрим функциональный подход к организации циклов, к работе с сопоставимыми выражениями и функциональными литералами.

Изучив весь этот материал, вы сможете извлечь максимум пользы из оставшейся части главы и обрести уверенность и первые навыки в программировании на Scala. Итак, начнем с обсуждения компилятора и интерактивной рабочей среды.

9.3.1. Использование компилятора и REPL

Scala — это компилируемый язык. Таким образом, при обычном исполнении программ Scala они сначала компилируются в `.class`-файлы, а потом выполняются в среде виртуальной машины Java. При этом в составе пути к классу (`CLASSPATH`) указывается файл `scala-library.jar` (библиотека времени исполнения для Scala).

Если вы еще не установили Scala, посмотрите в приложении C, как это делается, выполните необходимые операции, а потом продолжайте читать. Программу, приведенную в качестве примера (`HelloWorld` из подраздела 9.1.1), можно скомпилировать с помощью команды `scalac HelloWorld.scala` (предполагается, что сейчас вы находитесь в том каталоге, где сохранен файл `HelloWorld.scala`).

Когда у вас будет файл `.class`, можете запустить его с помощью команды `scala HelloWorld`. Она запускает виртуальную машину Java с указанием среды времени исполнения Scala в пути к классу, а потом входит в главный метод указанного файла класса.

Дополнительно к этим параметрам для компиляции и запуска в Scala имеется встроенная интерактивная среда. Она немного напоминает консоль Groovy, с которой мы работали в предыдущей главе. Но, в отличие от Groovy, в Scala такая интерактивная среда реализуется без командной строки. Это означает, что в типичной среде UNIX/Linux (с правильно настроенным `Path`) можно ввести `scala`, и этот ввод откроется в окне терминала. Новое окно создаваться не будет.

ПРИМЕЧАНИЕ

Интерактивные среды такого рода иногда называются циклами чтения-вычисления-печати, сокращенно — REPL. Они достаточно часто встречаются в языках, более динамических, чем Java. В среде REPL результаты предыдущих строк ввода остаются под рукой и могут переиспользоваться в последующих выражениях и расчетах. В оставшейся части этой главы мы время от времени будем прибегать к среде REPL, чтобы проиллюстрировать синтаксис Scala.

Теперь рассмотрим следующую важную языковую возможность, заслуживающую особого внимания, — продвинутое выведение типов в Scala.

9.3.2. Выведение типов

Во фрагментах кода Scala, которые мы уже успели рассмотреть, вы могли заметить такой момент: когда мы объявляем `hello` как `val`, то можем и не сообщать компилятору, какого типа эта переменная. Языку «очевидно», что она строковая. На первый взгляд это напоминает Groovy, где переменные не имеют типов (язык Groovy характеризуется динамической типизацией), но в Scala происходит нечто совершенно иное.

Scala — это статически типизированный язык (то есть у переменных есть совершенно определенные типы), но компилятор этого языка способен анализировать исходный код и во многих случаях «понимает из контекста», каким должен быть тип переменной. Если Scala способен дедуктивно узнавать типы, то их можно и не сообщать. Это и есть *выведение типов* (type inference) — возможность, о которой мы уже несколько раз упоминали. Scala отличается очень широкими возможностями в этой сфере — настолько, что иногда разработчик даже забывает о статической типизации и код выстраивается «по наитию». Поэтому в большинстве случаев этот язык кажется более динамическим, чем есть на самом деле.

ВЫВЕДЕНИЕ ТИПОВ

В Java возможности выведения типов ограниченны, но они существуют. Наиболее очевидный пример — ромбовидный синтаксис для работы с обобщенными типами, рассмотренный в главе 1. При выведении типов в Java выводимое значение обычно находится в правой части выражения присваивания. В Scala, как правило, выводится тип переменной, а не тип значения, но и выведение типов значений в Java также осуществимо.

Вы уже видели простейшие примеры использования этих возможностей при работе с ключевыми словами `var` и `val`. Эти слова заставляют переменную вывести ее тип на основании того, какое значение ей присвоено. Еще одна важная особенность выведения типов в Scala связана с объявлениями методов. Рассмотрим пример (держим в уме, что тип `AnyRef` в Scala идентичен `Object` в Java):

```
def len(obj : AnyRef) = {
  obj.toString.length
}
```

Это метод с выведенным типом. Компилятор может «понять», что этот метод возвращает `Int`, ознакомившись с возвращаемым кодом `java.lang.String#length`, который является `int`. Обратите внимание: возвращаемый тип здесь явно не указывается, а также не требуется использовать ключевое слово `return`. На самом деле если запрограммировать явный возврат, вот так:

```
def len(obj : AnyRef) = {
  return obj.toString.length
}
```

то получится ошибка времени компиляции:

```
error: method len has return statement; needs result type
       return obj.toString.length
       ^
```

Если вообще убрать = из def, то компилятор предположит, что перед ним метод, возвращающий Unit (в Scala это эквивалент метода Java, возвращающий void).

Дополнительно к вышеупомянутым ограничениям существуют еще две основные области, в которых выведение типов не применяется в полной мере:

- *типы параметров в объявлениях методов* — всегда необходимо указывать тип параметров в методах;
- *рекурсивные функции* — компилятор Scala не может вывести возвращаемый тип рекурсивной функции.

Ранее мы немного поговорили о методах Scala, но не обсуждали их систематическим образом. Поэтому подведем теоретические основы под уже известные сведения.

9.3.3. Методы

Вы уже видели, как определять метод с помощью ключевого слова def. При работе с методами Scala нужно учитывать и другие важные факты, особенно по мере того, как вы будете ближе знакомиться с этим языком.

- В Scala нет ключевого слова static. Аналоги статических методов из Java в Scala должны находиться в конструкциях object (это одиночки). Ниже мы познакомим вас с *объектами-спутниками* — удобной сущностью Scala, связанной с такими конструкциями.
- Среда времени исполнения языка Scala достаточно тяжеловесна по сравнению с аналогичной средой Groovy (или Clojure). В классах Scala может присутствовать множество дополнительных методов, автоматически генерируемых платформой.
- Концепция вызова методов имеет в Scala центральное значение. Операторы, аналогичные операторам Java, в Scala отсутствуют.
- Scala более гибко, нежели Java, регулирует то, какие символы могут употребляться в названиях методов. В частности, символы, которые в других языках применяются в качестве операторов, могут присутствовать в названиях методов Scala (например, символ +).

Синтаксис непрямого вызова методов (с которым мы сталкивались выше) помогает догадаться, как Scala удается объединять синтаксис вызовов методов и операторы. Возьмем, например, сложение двух целых чисел. В Java для этого необходимо написать выражение вида a + b. В Scala такой синтаксис также уместен, но можно написать и a.+(b). Здесь вы вызываете метод +() применительно к a и передаете ему b в качестве параметра. Именно так Scala удается избавиться от операторов как отдельного феномена.

ПРИМЕЧАНИЕ

Вы, наверное, заметили, что форма a.+(b) предполагает вызов метода применительно к a. Но что делать, если a — переменная примитивного типа? Мы подробно поговорим об этом в разделе 9.4, но пока достаточно указать, что система типов Scala, в принципе, воспринимает любую сущность как объект, поэтому методы можно вызывать применительно к чему угодно — даже к таким переменным, которые в Java были бы примитивными.

Вы уже видели пример использования ключевого слова `def` для объявления метода. Рассмотрим другой пример, в котором простой рекурсивный метод реализует функцию факториала:

```
def fact(base : Int) : Int = {
  if (base <= 0)
    return 1
  else
    return base * fact(base - 1)
}
```

В этой функции мы немного жульничаем, возвращая `1` для всех отрицательных чисел. На самом деле факториала от отрицательного числа не существует, но тут же все свои? Код немного напоминает Java — в нем есть возвращаемый тип (в данном случае `Int`), а также используется ключевое слово `return`, указывающее, какое значение следует передать вызывающей стороне. Единственный дополнительный момент, который здесь следует отметить, — посмотрите, как используется символ `=` перед блоком, определяющим тело функции.

В Scala есть и еще одна концепция, отсутствующая в Java, — локальная функция. Это функция, определяемая внутри другой функции (и действующая лишь в области ее видимости). Локальные функции удобны в качестве вспомогательных, если разработчик не хочет открывать такую вспомогательную функцию в чрезмерно широкой области видимости. В Java подобная проблема решалась бы лишь путем применения закрытого метода, но функция все равно была бы видима для других методов в пределах класса, к которому относится этот метод. Но в Scala можно написать следующий код:

```
def fact2(base : Int) : Int = {

  def factHelper(n : Int) : Int = {
    return fact2(n-1)
  }

  if (base <= 0)
    return 1
  else
    return base * factHelper(base)
}
```

Функция `factHelper()` ни при каких условиях не будет видна вне объемлющей ее области видимости `fact2()`.

Далее поговорим о том, как Scala работает с организацией кода и операциями импорта.

9.3.4. Импорт

Scala работает с пакетами так же, как и Java, применяя при этом такие же ключевые слова: `package` и `import`. Scala без проблем может импортировать и использовать пакеты и классы Java. В переменных Scala `var` или `val` может содержаться

экземпляр любого класса Java, он не требует никакого специального синтаксиса или особого обращения:

```
import java.io.File
import java.net._
import scala.collection.{Map, Seq}
import java.util.{Date => UDate}
```

Первые две строки здесь — эквивалент стандартного импорта и импорта по шаблону из Java. Третья строка позволяет импортировать одной строкой несколько классов, относящихся к одному и тому же пакету. В последней строке мы видим совмещение имени класса во время импорта (это необходимо во избежание нежелательных конфликтов сокращенных имен).

В отличие от Java, в Scala импорт может происходить в любой части кода (а не только в начале файла), поэтому его можно изолировать просто в каком-то разделе файла. Scala предоставляет и стандартные импорты; например, `scala._` всегда импортируется в любой файл `.scala`. Здесь содержится несколько полезных функций, отдельные из которых, например `println`, мы уже обсуждали. Все подробности о стандартных импортах описаны в документации по API на сайте www.scala-lang.org/.

Теперь поговорим о том, как можно управлять ходом выполнения программы на Scala. Этот процесс может немного отличаться от того, с которым вам приходилось сталкиваться в Java и Groovy.

9.3.5. Циклы и управляющие структуры

В Scala существует несколько новых вариантов управляющих и циклических конструкций. Но прежде, чем мы опробуем эти незнакомые формы, обратимся к знакомым — например, к стандартному циклу `while`:

```
var counter = 1
while (counter <= 10) {
  println("." * counter)
  counter = counter + 1
}
```

А вот цикл `do-while`:

```
var counter = 1
do {
  println("." * counter)
  counter = counter + 1
} while (counter <= 10)
```

Еще одна довольно знакомая форма — простейший цикл `for`:

```
for (i <- 1 to 10) println(i)
```

Итак, все это вы помните. Но в Scala есть и более гибкие конструкции, например условный цикл `for`:

```
for (i <- 1 to 10; if i %2 == 0) println(i)
```

Цикл for также пригоден для последовательного прохода по нескольким переменным, вот так:

```
for (x <- 1 to 5; y <- 1 to x)
  println(" " * (x - y) + x.toString * y)
```

Эти более гибкие варианты цикла for существуют в Scala благодаря тому, что в Scala такая конструкция обрабатывается совершенно по-иному. Для реализации цикла for в Scala применяются особые сущности из функционального программирования, называемые *списковыми выражениями* (list comprehension).

Общая идея спискового выражения сводится к тому, что вы начинаете работать со списком и запускаете преобразование (или — для условного цикла for — фильтр) применительно к элементам, образующим список. В результате генерируется новый список, который вы потом прорабатываете в теле цикла for, по одному элементу за раз.

Мы даже можем отделить определение отфильтрованного списка от выполнения тела for; это делается с помощью ключевого слова yield. Например, в следующем фрагменте кода:

```
val xs = for (x <- 2 to 11) yield fact(x)
for (factx <- xs) println(factx)
```

Здесь мы задаем xs как новую коллекцию перед тем, как проработать ее элементы во втором цикле for, выводящем значения на экран. Такой альтернативный синтаксис будет крайне полезен, если вы хотите собрать коллекцию однажды и использовать ее многократно.

Эта конструкция реализуема потому, что язык Scala пригоден для функционального программирования. Итак, рассмотрим, как в Scala воплощаются функциональные идеи.

9.3.6. Функциональное программирование на Scala

Как было указано в подразделе 7.5.2, функции поддерживаются в Scala как первоклассные значения. Иными словами, в языке Scala вы можете писать функции так, что они могут вставляться в var или val, а потом с ними можно обращаться как с любыми другими значениями. Такие функции называются *функциональными литералами* или *анонимными функциями* и играют очень важную роль в языковой картине мира, существующей в Scala.

Писать функциональные литералы на Scala очень легко. Основной элемент синтаксиса — это стрелка =>, с помощью которой в Scala выражается прием списка параметров и передача их в блок кода:

```
(<list of function parameters>) => { ... function body as a block ... }
```

Продемонстрируем эту возможность в интерактивной среде Scala. В следующем простом примере определяется функция, принимающая целое число Int и удваивающая его:

```
scala> val doubler = (x : Int) => { 2 * x }
doubler: (Int) => Int = <function1>

scala> doubler(3)
```

```
res4: Int = 6

scala> doubler(4)
res5: Int = 8
```

Обратите внимание, как Scala выводит тип `doubler`. Его можно охарактеризовать как функцию, принимающую `Int` и возвращающую `Int`. Этот тип не имеет достаточно точного соответствия в системе типов Java. Как видите, для вызова `doubler` со значением используется стандартный синтаксис вызова со скобками.

Несколько разовьем эту концепцию. В Scala функциональные литералы — это просто значения. А значения возвращаются от функций. Это означает, что у вас может быть функция, создающая другие функции: это будет функциональный литерал, принимающий значение и возвращающий в качестве значения новую функцию.

Например, можно определить функциональный литерал под названием `adder`. Назначение `adder()` — создавать функции, которые будут прибавлять константное значение к своему аргументу.

```
scala> val adder = (n : Int) => { (x : Int) => x + n }
adder: (Int) => (Int) => Int = <function1>

scala> val plus2 = adder(2)
plus2: (Int) => Int = <function1>

scala> plus2(3)
res2: Int = 5

scala> plus2(4)
res3: Int = 6
```

Как видите, функциональные литералы хорошо поддерживаются в Scala. На самом деле на Scala можно программировать в очень функциональном стиле, а также пользоваться более императивным стилем, который мы применяли до сих пор. Мы собирались лишь слегка затронуть возможности функционального программирования на Scala — просто не забывайте, что они есть.

В следующем разделе мы обсудим применяемую в Scala объектную модель и детально рассмотрим подход Scala к объектной ориентации. В Scala имеется несколько продвинутых возможностей, благодаря которым обращение с объектной ориентацией в Scala значительно отличается от аналогичной работы в Java в кое-каких важных деталях.

9.4. Объектная модель Scala — знакомая, но своеобразная

Иногда Scala характеризуется как «чистый» объектно-ориентированный язык. Это означает, что все значения являются объектами, поэтому на Scala практически ничего нельзя написать, не сталкиваясь с объектно-ориентированными концепциями. Мы начнем этот раздел, разобрав последствия ситуации, в которой «все

является объектами». Естественно, такая ситуация заставляет задуматься о существующей в Scala иерархии типов. Эта иерархия в некоторых важных моментах отличается от аналогичной иерархии Java и также определяет подход Scala к обработке примитивных типов, например к их упаковке и распаковке.

После этого мы поговорим о конструкторах Scala и определениях классов, расскажем, как эти элементы позволяют обходиться небольшим количеством кода. Далее мы рассмотрим важную тему типажей, а потом поговорим об одиночках (синглтонах) языка Scala, объектах-спутниках и пакетных объектах. В завершение этого раздела мы изучим, как `case`-классы позволяют еще значительнее сократить количество шаблонного кода, и, наконец, поведаем назидательную историю из синтаксиса Scala.

Поехали.

9.4.1. Любая сущность — это объект

В Scala считается, что любой тип является объектным. К объектам относятся и такие типы, которые в Java называются примитивными. На рис. 9.1 показано наследование типов в Scala, в частности указаны эквиваленты как значимых (примитивных), так и ссылочных типов.

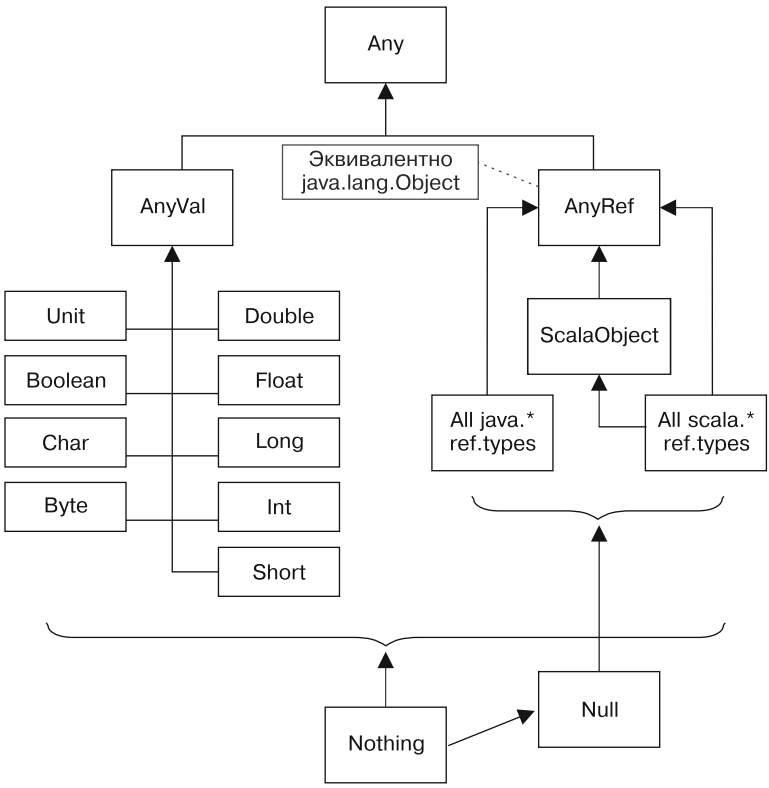


Рис. 9.1. Иерархия наследования в Scala

Как видите, тип `Unit` — это полноценный тип Scala, находящийся в ряду других значимых типов. Класс `AnyRef` в Scala аналогичен сущности `java.lang.Object`, имеющейся в Java. Каждый раз, встречая в коде `AnyRef`, можете смело считать, что перед вами `Object`. Такое специальное название появилось потому, что ранее язык Scala был также ориентирован на работу в исполняемой среде .NET, и отдельное название для этой концепции действительно казалось целесообразным.

При наследовании классов Scala использует ключевое слово `extends`, действующее почти так же, как и в Java: наследуются все незакрытые члены и между двумя типами будут установлены отношения вида «суперкласс/субкласс». Если при определении класса отсутствует явное дополнение другого класса, то компилятор сочтет определяемый таким образом класс прямым потомком `AnyRef`.

Принцип «любая сущность — это объект» объясняет синтаксическую черту Scala, с которой мы уже сталкивались, — обычная запись при вызовах методов. В подразделе 9.3.3 мы убедились, что `obj.meth(param)` и `obj meth param` — это равнозначные способы вызова метода. Теперь мы знаем, что выражение `1 + 2`, которое в Java включает числовые примитивы и оператор сложения, в Scala будет эквивалентно выражению `1.+(2)`, где используется вызов метода применительно к классу `scala.Int`.

Scala избавляется от некоторой путаницы, связанной с упакованными числовыми значениями, иногда возникающей в Java. Рассмотрим следующий код на Java:

```
Integer one = new Integer(1);
Integer uno = new Integer(1);
System.out.println(one == uno);
```

Он дает результат `false`, который может показаться странным. Как вы увидите ниже, в Scala используется специальный подход к упакованным числовым значениям и к равенствам вообще, который удобен в нескольких отношениях.

- Числовые классы нельзя инстанцировать из конструкторов. Фактически они являются одновременно `abstract` и `final` (такая комбинация была бы недопустима в Java).
- Чтобы получить новый экземпляр числового класса, этот класс должен быть представлен как литерал. Так мы гарантируем, что значение `2` всегда будет равно тому же `2`.
- Равенство, получаемое с помощью оператора `==`, по определению является идентичным `equals()`, а не арифметическим равенством.
- Оператор `==` нельзя переопределить, а `equals()` — можно.
- В Scala предоставляется метод `eq` для выражения арифметического равенства. Он требуется нечасто.

9.4.2. Конструкторы

Теперь, когда мы рассмотрели некоторые базовые объектно-ориентированные концепции Scala, немного ближе познакомимся с синтаксисом этого языка. Класс также может иметь дополнительные вспомогательные конструкторы. Они обо-

значаются с помощью синтаксиса this(), но их возможности более ограниченны, чем у перегруженных конструкторов Java.

В качестве первой инструкции, содержащейся во вспомогательном конструкторе Scala, должен выполняться вызов другого конструктора в том же классе. Такое ограничение служит для того, чтобы направить поток управления к основному конструктору, который и является единственной реальной входной точкой в класс. Таким образом, вспомогательные конструкторы действуют, в сущности, как поставщики стандартных параметров к основному конструктору.

Рассмотрим следующий вспомогательный конструктор, добавленный к CashFlow:

```
class CashFlow(amt : Double, curr : String) {
  def this(amt : Double) = this(amt, "GBP")
  def this(curr : String) = this(0, curr)

  def amount = amt
  def currency = curr
}
```

Вспомогательные конструкторы в данном примере помогают вам указывать лишь сумму, и в этом случае класс CashFlow «предположит», что пользователь желает получить наличные в британских фунтах. Другой вспомогательный конструктор позволяет создавать конструкцию, где указана лишь валюта; сумма в данном случае предположительно равна 0.

Обратите внимание и на то, что мы определили amount и currency как методы, не включая скобок или списка параметров (даже пустого). Так мы сообщаем компилятору, что, используя этот класс, код может вызывать amount или currency без применения скобок, вот так:

```
val wages = new CashFlow(2000.0)
println(wages.amount)
println(wages.currency)
```

Определения классов в Scala в целом соответствуют таким определениям в Java. Но есть несколько характерных отличий в том, как в Scala организуются детали объектно-ориентированного наследования. Об этом мы поговорим в следующем разделе.

9.4.3. Типажи

Типажи (traits) — это основной элемент реализуемого в Scala подхода к объектно-ориентированному программированию. В широком смысле типажи играют в Scala ту же роль, что и интерфейсы в Java. Но, в отличие от интерфейсов Java, типажи могут включать реализацию метода и использовать этот код совместно с другими классами, имеющими такой же типаж.

Рассмотрим проблему из Java, которую позволяет решить такой механизм. На рис. 9.2 показаны два класса Java, являющиеся производными от разных базовых классов. Если обоим требуется проявлять дополнительную общую функциональность, то достаточно объявить, что классы реализуют общий интерфейс.

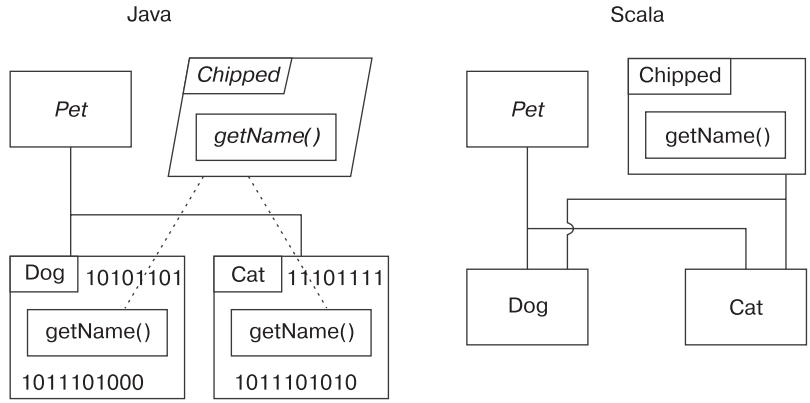


Рис. 9.2. Дублирование реализации в модели Java

В листинге 9.2 показан простой пример Java, в котором описанная ситуация реализована в коде. Вспомним заботливого ветеринара, о котором шла речь в подразделе 4.3.6. У многих питомцев, которых ему приносят на лечение, будут специальные идентифицирующие микрочипы. Так, они будут практически у всех кошек и собак, а у других домашних питомцев их может и не быть.

Возможность обладания таким чипом должна быть выделена в самостоятельный интерфейс. Обновим код из листинга 4.11, чтобы дополнить его такой возможностью (для ясности кода опустим метод `examine()`).

Листинг 9.2. Пример реализации дублирования кода

```
public abstract class Pet {
  protected final String name;
  public Pet(String name_) {
    name = name_;
  }
}

public interface Chipped {
  String getName();
}

public class Cat extends Pet implements Chipped {
  public Cat(String name_) {
    super(name_);
  }
  public String getName() {
    return name;
  }
}

public class Dog extends Pet implements Chipped {
  public Dog(String name_) {
    super(name_);
  }
```

```
  public String getName() {
    return name;
  }
}
```

Как видите, в Dog и Cat требуется дублировать код, реализующий getName(), поскольку интерфейсы Java не могут содержать кода реализации. В листинге 9.3 показано, как аналогичная ситуация реализуется на Scala с применением типажей.

Листинг 9.3. Домашние любимцы на Scala

```
class Pet(name : String)

trait Chipped {
  var chipName : String
  def getName = chipName
}

class Cat(name : String) extends Pet(name : String) with Chipped {
  var chipName = name
}

class Dog(name : String) extends Pet(name : String) with Chipped {
  var chipName = name
}
```

В Scala требуется присвоить значение каждому параметру, присутствующему в условии конструктора суперкласса, — такие условия находятся в составе субклассов. Но объявления методов, присутствующие в типаже, наследуются каждым субклассом. Благодаря этому дублирование при реализации сокращается. Это свойство заметно на практике в ситуации, где параметр name должен обрабатываться как в Cat, так и в Dog. Оба субкласса имеют доступ к реализации, предоставляемой Chipped, — в данном случае имеем параметр chipName, в котором можно хранить кличку, записанную в микрочипе.

9.4.4. Одиночка и объект-спутник

Рассмотрим, как реализуются одиночки Scala (одиночки — это классы, начинающиеся с ключевого слова object). Вспомните код нашего первого примера Hello World, приведенного в подразделе 9.1.1:

```
object HelloWorld {
  def main(args : Array[String]) {
    val hello = "Hello World!"
    println(hello)
  }
}
```

На языке Java было бы логично предположить, что этот код будет преобразовываться в отдельный файл HelloWorld.class. В Scala же этот код компилируется в два файла: HelloWorld.class и HelloWorld$.class.

Поскольку это обычные файлы классов, можно воспользоваться декомпилирующим инструментом javap, описанным в главе 5, чтобы просмотреть байт-код, сгенерированный компилятором Scala. Так вы сможете получить определенные подсказки о том, как построена модель типов в Scala и как она реализуется. В листинге 9.4 показан результат применения javap -c -p к двум файлам классов.

Листинг 9.4. Декомпиляция объектов-одиночек в Scala

```
Compiled from "HelloWorld.scala"
public final class HelloWorld extends java.lang.Object {
  public static final void main(java.lang.String[]);
    Code:
      0: getstatic #11
  ➥ // Поле HelloWorld$.MODULE$:LHelloWorld$;
      3: aload_0
      4: invokevirtual #13
  ➥ // Метод HelloWorld$.main:([Ljava/lang/String;)V
      7: return
}

Compiled from "HelloWorld.scala"
public final class HelloWorld$ extends java.lang.Object
➥ implements scala.ScalaObject {
  public static final HelloWorld$ MODULE$;

  public static {};
    Code:
      0: new           #9    // Класс HelloWorld$
      3: invokespecial #12   // Метод "<init>":()V
      6: return

  public void main(java.lang.String[]);
    Code:
      0: getstatic     #19   // Поле scala/Predef$.MODULE$:Lscala/Predef$;
      3: ldc           #22   // Строка Hello World!
      5: invokevirtual #26
➥ // Метод scala/Predef$.println:(Ljava/lang/Object;)V
      8: return

  private HelloWorld$();
    Code:
      0: aload_0
      1: invokespecial #33   // Метод java/lang/Object."<init>":()V
      4: aload_0
      5: putstatic     #35   // Поле MODULE$:LHelloWorld$;
      8: return
}
```

Получаем одиночку-спутник MODULE$

Вызываем main() к объекту-спутнику

Экземпляр одиночки-спутника

Закрытый конструктор

Теперь понятно, откуда взялось утверждение, что в Scala нет статических методов или полей. Вместо этих конструкций компилятор Scala работает с авто-

матически сгенерированным кодом, соответствующим паттерну Singleton. Паттерн предполагает наличие неизменяемого статического экземпляра и закрытого конструктора. Такой код автоматически вставляется в файл `.class`, название которого оканчивается на `$`. Метод `$` — это обычный метод экземпляра, но он вызывается применительно к классу `HelloWorld$`, который является одиночкой.

Таким образом, возникает двойственность в паре файлов `.class`. Один из них будет одноименным файлу Scala, а название другого оканчивается на `$`. Статические методы и поля помещаются именно во второй класс-синглтон.

В таких случаях очень часто одновременно применяются одноименные класс Scala и объект. Класс-синглтон еще называется *объектом-спутником*. Отношение между исходным файлом на Scala и двумя классами виртуальной машины (основным классом и объектом-спутником) показано на рис. 9.3.

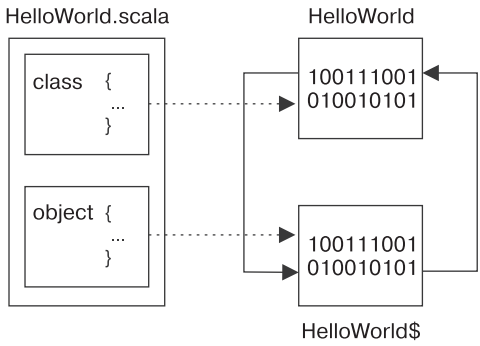


Рис. 9.3. Объекты-одиночки в Scala

Мы уже сталкивались с объектами-спутниками, но не знали, что это они. В примере Hello World нам не пришлось указывать, в каком классе содержится метод `println()`. Этот метод кажется статическим, поэтому логично предположить, что он будет относиться к объекту-спутнику.

Вновь обратимся к байт-коду из листинга 9.2, соответствующему методу `main()`:

```
public void main(java.lang.String[]);
  Code:
    0: getstatic      #19    // Поле scala/Predef$.MODULE$:Lscala/Predef$;
    3: ldc            #22    // Строка Hello World!
    5: invokevirtual  #26
➥// Метод scala/Predef$.println:(Ljava/lang/Object;)V
    8: return
```

На данном этапе вы видите, что `println()` и другие непременно доступные функции Scala содержатся в объекте, который является спутником класса `Scala.Predef`.

Объект-спутник имеет привилегированное отношение к своему классу. В частности, он может обращаться к закрытым методам этого класса. Благодаря этому Scala удается с удобством определять закрытые вспомогательные конструкторы. При работе с закрытыми конструкторами Scala использует синтаксис,

в котором ключевое слово `private` предшествует списку параметров конструктора, вот так:

```
class CashFlow private (amt : Double, curr : String) {
  ...
}
```

Если конструктор, заявленный как основной, является закрытым, то остается только два способа создания новых экземпляров такого класса: либо с помощью фабричного метода, содержащегося в объекте-спутнике (поскольку такой метод будет иметь доступ к закрытому конструктору), либо путем вызова общедоступного вспомогательного конструктора.

Теперь перейдем к следующей теме — поговорим о применяемых в Scala case-классах. Вы уже сталкивались с ними, но напомним, что такие классы удобны для сокращения шаблонного кода, так как автоматически предоставляют несколько простых методов.

9.4.5. Case-классы и сопоставимые выражения

Вспомним, как в Java моделируется простая сущность, например класс `Point` (листинг 9.5).

Листинг 9.5. Реализация простого класса в Java

```
public class Point {
  private final int x;
  private final int y;

  public Point(int x, int y) {
    this.x = x;
    this.y = y;
  }

  public String toString() {
    return "Point(x: " + x + ", y: " + y + ")";
  }

  @Override
  public boolean equals(Object obj) {
    if (!(obj instanceof Point)) {
      return false;
    }
    Point other = (Point)obj;
    return other.x == x && other.y == y;
  }

  @Override
  public int hashCode() {
    return x * 17 + y;
  }
}
```

Шаблонный код

Как видите, здесь присутствует огромное количество шаблонного кода. К тому же методы `hashCode()`, `toString()`, `equals()`, а также все без исключения методы-установщики должны автоматически генерироваться интегрированной средой разработки. Как было бы хорошо, если бы язык допускал более простой синтаксис, а автоматическая генерация обрабатывалась в языковом ядре!

Действительно, Scala обеспечивает такую поддержку в форме языковой сущности, называемой `case`-классами. Листинг 9.5 можно переписать в очень простом виде:

```
case class Point(x : Int, y : Int)
```

Здесь заключена вся функциональность из более длинного примера на Java, но есть и дополнительные достоинства — посмотрите, насколько краток этот код.

Если бы вы хотели изменить код Java (допустим, добавить в него координату *Z*), то пришлось бы обновить `toString()` и другие методы. На практике для этого понадобилось бы удалить весь имеющийся метод и заново сгенерировать его в IDE.

В версии для Scala такой необходимости попросту нет, так как отсутствует явное определение метода, которое потребовалось бы держать актуальным. Такая особенность лежит в основе очень мощного принципа — вы просто не можете допустить ошибок в коде, который не существует на уровне исходников.

Создавая новые экземпляры `case`-класса, можно пропускать ключевое слово `new`. Например, допускается написать следующий код:

```
val pythag = Point(3, 4)
```

Такой синтаксис подчеркивает, что `case`-классы напоминают перечисления с одним или более свободными параметрами. На внутрисистемном уровне происходит буквально следующее: определение `case`-класса предоставляет фабричный метод, который будет создавать новые экземпляры.

Рассмотрим один из наиболее распространенных случаев использования `case`-классов: с паттернами и выражениями `match`. При программировании на Scala `case`-классы могут использоваться в рамках паттерна, который называется Constructor («Конструктор»). Взгляните на код листинга 9.6.

Листинг 9.6. Паттерн Constructor с match-выражением

```
val xaxis = Point(2, 0)
val yaxis = Point(0, 3)
val some = Point(5, 12)
val whereami = (p : Point) => p match {
  case Point(x, 0) => "On the x-axis"
  case Point(0, y) => "On the y-axis"
  case _ => "Out in the plane"
}
println(whereami(xaxis))
println(whereami(yaxis))
println(whereami(some))
```

Мы вновь обратимся к паттерну Constructor и `case`-классам в разделе 9.6, где обсудим акторы и подход Scala к параллельной обработке.

Прежде чем завершить этот раздел, мы хотели бы добавить одно предупреждение. Благодаря богатству синтаксиса Scala и интеллектуальности синтаксического

анализатора этого языка существуют очень элегантные и лаконичные способы представления довольно сложного кода. Но в языке Scala отсутствует официальная спецификация, а новые функции добавляются достаточно часто. Необходимо очень внимательно работать — даже опытные Scala-программисты иногда обжигаются на языковых функциях, которые вдруг начинают вести себя неожиданно. Такие проблемы особенно актуальны, когда синтаксические функции комбинируются друг с другом.

Рассмотрим пример — попробуем смоделировать перегрузку операторов в синтаксисе Scala.

9.4.6. Предостережение

Рассмотрим case-класс Point, который мы создали выше. Возможно, вам потребуется простой способ сложения координат или их линейного масштабирования. Если у вас есть математическое образование, то вы, вероятно, помните, что при представлении координат на плоскости они имеют векторные свойства.

В листинге 9.7 показан простой способ определения методов, которые в обычной практике будут действовать как операторы.

Листинг 9.7. Моделирование перегрузки операторов

```
case class Point(x : Int, y : Int) {
  def *(m : Int) = Point(this.x * m, this.y * m)
  def +(other : Point) = Point(this.x + other.x, this.y + other.y)
}

var poin = Point(2, 3)
var poin2 = Point(5, 7)
println(poin)
println(poin 2)
println(poin * 2)
println(poin + poin2)
```

Запустив этот код, вы получите следующий вывод:

```
Point(2,3)
Point(5,7)
Point(4,6)
Point(7,10)
```

Здесь мы видим, насколько удобнее работать с case-классами Scala, чем с их эквивалентами из Java. Написав совсем небольшое количество кода, вы создаете понятный класс, который дает разумный вывод. При возможности определения методов для + и * вы смогли смоделировать особенности перегрузки операторов.

Но при таком подходе возникает и одна проблема. Рассмотрим следующий код:

```
var poin = Point(2, 3)
println(2 * poin)
```

Здесь возникнет ошибка времени компиляции:

```
error: overloaded method value * with alternatives:
  (Double)Double <and>
  (Float)Float <and>
  (Long)Long <and>
  (Int)Int <and>
  (Char)Int <and>
  (Short)Int <and>
  (Byte)Int
cannot be applied to (Point)
               println(2 * poin)
                         ^
one error found
```

Причина возникновения ошибки заключается в том, что, хотя вы и определили метод `*(m : Int)` для `case`-класса `Point`, этот не тот метод, в котором нуждается Scala. Чтобы предыдущий код скомпилировался, придется предоставить метод `*(p : Point)` для стандартного класса `Int`. Это нереализуемо, поэтому иллюзия перегрузки операторов остается неполной.

На этом примере мы наблюдаем интересное свойство Scala — у многих синтаксических функций есть свои ограничения, которые могут в определенных ситуациях давать неожиданные эффекты. Синтаксический анализатор и среда времени исполнения Scala выполняют большую работу на внутрисистемном уровне, но вся эта скрытая машинерия, в сущности, просто пытается сделать то, что надо.

На этом мы завершаем вводный рассказ о подходе Scala к объектной парадигме. Немало продвинутых возможностей мы даже не затронули. В Scala удалось воплотить многие современные идеи о том, как должны работать объекты и системы типов. Если вы хотите подробнее познакомиться с этими темами, то вам предстоит немало всего изучить. Почитайте книгу Джошуа Сюрета Scala in Depth (издательство Manning Publications, 2012) или другую специализированную книгу по Scala — возможно, нам удалось пробудить в вас интерес к тому, как в Scala строится система типов и организуется объектно-ориентированный подход.

Как вы уже могли догадаться, эта часть языковой теории имеет важное прикладное значение в области работы со структурами данных и коллекциями. Именно об этом мы и поговорим в следующем разделе.

9.5. Структуры данных и коллекции

Мы уже сталкивались с примером простой структуры данных Scala — со списком `List`. Такая структура данных имеет фундаментальное значение в любом языке программирования и в Scala также очень важна. Здесь мы подробно рассмотрим `List`, а потом перейдем к изучению словаря (`Map`) из Scala.

Далее мы подробно обсудим применение обобщенных типов (дженериков) в Scala, в частности их отличие от дженериков Java и дополнительную мощность

по сравнению с аналогичными конструкциями из Java. При этом предполагается, что вы проработали приведенные выше примеры стандартных коллекций Scala, чтобы закрепить теорию.

Начнем с нескольких общих замечаний о коллекциях Scala. Отдельно коснемся их неизменяемости и взаимодействия с коллекциями Java.

9.5.1. Список

В Scala работа с коллекциями организована иначе, нежели в Java. Это может показаться удивительным, так как во многих других областях Scala переиспользует и дополняет различные компоненты и концепции Java. Рассмотрим важнейшие характерные черты философии Scala при работе с коллекциями:

- коллекции Scala обычно являются неизменяемыми;
- многие свойства `List`-подобных коллекций в Scala обосабливаются в отдельные концепции;
- ядро сущности `List` в Scala состоит из очень небольшого количества концепций;
- при работе с коллекциями Scala стремится обеспечить единообразие в работе различных типов коллекций;
- в Scala у разработчика есть стимул создавать собственные коллекции, которые можно воспринимать как встроенные классы коллекций.

По очереди рассмотрим эти основные отличия.

Неизменяемые и изменяемые коллекции

Одна из первых вещей, которую необходимо учесть, заключается в том, что в Scala существуют как изменяемые, так и неизменяемые варианты одних и тех же коллекций. По умолчанию применяются неизменяемые коллекции (они всегда доступны для любого файла с исходным кодом Scala).

Необходимо описать четкую разницу между изменяемостью коллекции и изменяемостью ее содержимого. Рассмотрим эту разницу на практике (листинг 9.8).

Листинг 9.8. Изменяемые и неизменяемые

```
import scala.collection.mutable.LinkedList
import scala.collection.JavaConversions._
import java.util.ArrayList

object ListExamples {
  def main(args : Array[String]) {
    var list = List(1,2,3)
    list = list :+ 4
    println(list)

    val linklist = LinkedList(1,2,3)
    linklist.append(LinkedList(4))
```

```
    println(linklist)

    val jlist = new ArrayList[String]()
    jlist.add("foo")
    val slist = jlist.toList
    println(slist)
  }
}
```

Как видите, `list` является изменяемой ссылкой (это `var`). Она указывает на экземпляр неизменяемого списка, поэтому вы можете заново присвоить ее, чтобы она указывала на новый объект. Метод `:+` возвращает новый (неизменяемый) экземпляр `List`, к которому прикреплен дополнительный элемент.

Напротив, `linklist` является неизменяемой ссылкой (`val`) на список `LinkedList`, который уже является изменяемым. Вы можете откорректировать содержимое `linklist`; для этого можно, например, применить к нему `append()`. Разница показана на рис. 9.4.

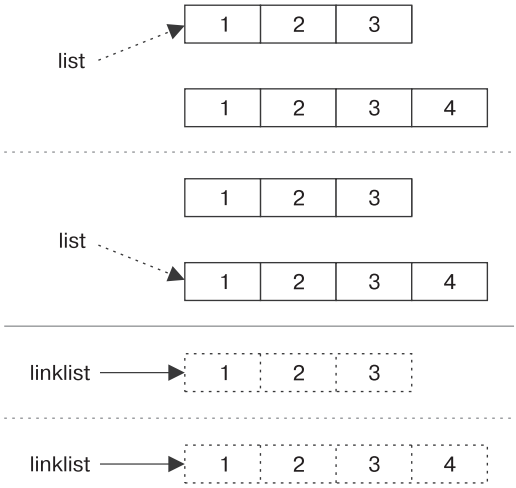


Рис. 9.4. Неизменяемые и изменяемые коллекции

В листинге 9.8 мы также продемонстрировали группу полезных функций для преобразования — класс `JavaConversions`. Он используется для превращения коллекций Java в их эквиваленты на Scala и обратно.

Типажи для списков

Важная характерная особенность Scala заключается в том, что в этом языке акцентируются типажи и поведенческие аспекты коллекций. Для примера рассмотрим `ArrayList` из Java. Не считая `Object`, этот класс прямо или косвенно дополняет:

- `java.util.AbstractList`;
- `java.util.AbstractCollection`.

Следует также поговорить об интерфейсах. `ArrayList` или один из его суперклассов реализует такие интерфейсы:

- `Serializable`;
- `Collection`;
- `Cloneable`;
- `List`;
- `Iterable`;
- `RandomAccess`.

В Scala ситуация немного сложнее. Рассмотрим класс `LinkedList`. Он может быть производным от 27 других классов или типажей:

- `Serializable`;
- `LinearSeqLike`;
- `SeqLike`;
- `PartialFunction`;
- `IterableLike`;
- `GenIterableLike`;
- `GenTraversable`;
- `GenTraversableLike`;
- `LinkedListLike`;
- `Cloneable`;
- `GenSeq`;
- `Function1`;
- `Equals`;
- `Mutable`;
- `GenTraversableTemplate`;
- `Parallelizable`;
- `LinearSeq`;
- `Seq`;
- `GenSeqLike`;
- `Iterable`;
- `GenIterable`;
- `Traversable`;
- `TraversableLike`;
- `TraversableOnce`.

Коллекции Scala не отличаются друг от друга столь же четко, как коллекции Java. В Java `List`, `Map`, `Set` и другие обрабатываются с применением немного разных паттернов, в зависимости от конкретного типа использования. Но в Scala, благо-

даря типажам, типы получаются гораздо более рафинированными, чем в Java. Поэтому вы можете сосредоточиться на одном или нескольких характерных чертах той или иной коллекции и более точно выразить свое намерение, воспользовавшись типом, который максимально точно соответствует интересующему вас объекту.

По этой причине код для обработки коллекций в Scala кажется значительно более единообразным, чем аналогичный код Java.

МНОЖЕСТВА В SCALA

Как вы уже, наверное, догадываетесь, в Scala поддерживаются как изменяемые, так и неизменяемые множества. Как правило, множества используются здесь по такому же принципу, как и в Java, — с применением промежуточного объекта, выполняющего ту или иную операцию с каждым элементом коллекции. Но в тех случаях, где в Java применялись бы `Iterator` или `Iterable`, Scala использует `Traversable`, непригодный для взаимодействия с типами Java.

Список строится на основании всего двух фундаментальных элементов. Во-первых, это `Nil`, представляющий собой пустой список, а во-вторых, оператор `::`, создающий новые списки из старых. Оператор `::` называется *cons*, он похож на форму (`concat`) из `Clojure`, о которой мы поговорим в главе 10. Оба этих элемента показывают, насколько язык Scala связан с `Lisp`.

Оператор cons принимает два аргумента — элемент типа `T` и объект типа `List[T]`. Он создает новое значение типа `List[T]`, в котором объединяется содержимое двух аргументов:

```
scala> val x = 2 :: 3 :: Nil
x: List[Int] = List(2, 3)
```

Или же можно написать это напрямую:

```
scala> val x = List(2, 3)
x: List[Int] = List(2, 3)

scala> 1 :: x
res0: List[Int] = List(1, 2, 3)
```

ОПЕРАТОР CONS И СКОБКИ

Определение оператора cons говорит о том, что `A :: B :: C` находятся в однозначном соответствии. Таким образом, `A :: (B :: C)`. Это объясняется тем, что первый аргумент `::` — это единственное значение типа `T`. Но `A :: B` — это значение типа `List[T]`, поэтому `(A :: B) :: C` не имеет никакого смысла как возможное значение. Академические специалисты по информатике сказали бы, что оператор `::` является правоассоциативным.

Итак, теперь понятно, почему необходимо говорить, что `2 :: 3 :: Nil`, а не просто `2 :: 3`. Требуется, чтобы второй аргумент `::` был значением типа `List`, а `3` не является `List`.

Наряду с `List` в Scala предусмотрены собственные формы других известных коллекций. Далее мы поговорим о словаре (`Map`).

9.5.2. Словарь

Коллекция Map — еще одна классическая структура данных. В Java она чаще всего встречается в форме HashMap. Scala предоставляет неизменяемый класс Map по умолчанию, а также предлагает HashMap как стандартную изменяемую форму.

В листинге 9.9 показаны некоторые простые способы определения словарей и работы с ними.

Листинг 9.9. Словари в Scala

```
import scala.collection.mutable.HashMap

var x = Map(1 -> "hi", 2 -> "There")
for ((key, vau) <- x) println(key + ": " + vau)
x = x + (3 -> "bye")

val hm = HashMap(1 -> "hi", 2 -> "There")
hm += (3 -> "bye")
println(hm)
```

Как видите, в Scala используется отличный компактный синтаксис для определения словарного литерала: Map(1 -> "hi", 2 -> "There"). Запись с применением стрелок наглядно показывает, на какое значение указывает каждый ключ. Чтобы возвращать значения от словарей, применяется метод get(), как и в Java.

Как в изменяемых, так и в неизменяемых словарях символ + означает добавление к словарю, а символ - — удаление из словаря. Но здесь есть некоторые нюансы. При применении оператора + с изменяемым словарем оператор изменяет словарь и возвращает его. В случае с неизменяемым словарем возвращается новый словарь, содержащий новую пару ключ/значение. В результате возникает следующий патологический случай, связанный с применением оператора +=:

```
scala> val m = Map(1 -> "hi", 2 -> "There", 3 -> "bye", 4 -> "quux")
m: scala.collection.immutable.Map[Int,java.lang.String]
➥ = Map(1 -> hi, 2 -> There, 3 -> bye, 4 -> quux)

scala> m += (5 -> "Blah")
<console>:10: error: reassignment to val
      m += (5 -> "Blah")
          ^

scala> val hm = HashMap(1 -> "hi", 2 -> "There", 3 -> "bye", 4 -> "quux")
hm: scala.collection.mutable.HashMap[Int,java.lang.String]
➥ = Map(3 -> bye, 4 -> quux, 1 -> hi, 2 -> There)

scala> hm += (5 -> "blah")
res6: hm.type = Map(5 -> blah, 3 -> bye, 4 -> quux, 1 -> hi, 2 -> There)
```

Причина заключается в том, что += по-разному реализуется с неизменяемыми и с изменяемыми словарями. В случае с изменяемым словарем += — это метод, меняющий имеющийся словарь. Соответственно, такой метод с полным правом можно вызывать применительно к val (точно так же мы вызываем put() примени-

тельно к `final HashMap` в Java). В случае с неизменяемым словарем `+=` разбивается на `=` и `+`, как это показано в листинге 9.9; этот оператор нельзя будет использовать с `val`, поскольку повторное присваивание не допускается.

Еще один интересный образец синтаксиса, показанный в листинге 9.9, — это синтаксис цикла `for`. В нем используется идея списковых выражений (о которых мы поговорили в подразделе 9.3.5), но наряду с применением этой идеи каждая пара ключ/значение разделяется на ключ и значение. Этот процесс называется деструктуризацией пары — еще одна концепция из функционального наследия Scala.

Итак, мы лишь слегка затронули сущность словарей Scala и немного коснулись их широких возможностей, но теперь нужно двигаться дальше. Поговорим об обобщенных типах.

9.5.3. Обобщенные типы

Мы уже видели, что Scala использует квадратные скобки для указания параметризованных типов, а также познакомились с некоторыми простейшими структурами данных из Scala. Копнем поглубже и рассмотрим, чем подход Scala к работе с обобщенными типами отличается от подхода Java.

Сначала разберем, что произойдет, если попытаться игнорировать такие типы при определении функционального параметра:

```
scala> def junk(x : List) = println("hi")
<console>:5: error: type List takes type parameters
       def junk(x : List) = println("hi")
                    ^
```

В Java это совершенно нормальный ход. Компилятор может «пожаловаться», но справится с ним. А в Scala в таком случае вы получите серьезную ошибку времени компиляции. Списки (и другие обобщенные типы) должны быть параметризованы — и точка. В Scala отсутствует аналог «необработанных типов», присутствующих в Java.

Выведение обобщенных типов

При присваивании значения переменной обобщенного типа Scala обеспечивает удобное выведение типов по их параметрам. Это соответствует генеральной линии Scala по обеспечению полезного выведения типов и по максимальному сокращению шаблонного кода:

```
scala> val x = List(1, 2, 3)
x: List[Int] = List(1, 2, 3)
```

Одна из черт обобщенных типов в Scala, которая на первый взгляд может показаться странной, заключается в использовании оператора сцепления `:::`, объединяющего списки для получения нового списка:

```
scala> val y = List("cat", "dog", "bird")
y: List[java.lang.String] = List(cat, dog, bird)
scala> x ::: y
res0: List[Any] = List(1, 2, 3, cat, dog, bird)
```

Мы видим, что, когда среда времени исполнения пытается создать таким способом новый список `List`, ошибка не возникает, а создается список с наименьшим общим супертипом для `Int` и `String`. Это супертип `Any`.

Пример использования дженериков — питомцы, ожидающие ветеринара

Предположим, у нас есть несколько домашних питомцев, ожидающих своей очереди к ветеринару, и вы хотите смоделировать эту очередь. В листинге 9.10 показаны знакомые базовые классы и вспомогательная функция, которую можно использовать в качестве отправной точки.

Листинг 9.10. Зверюшки, ожидающие осмотра

```
class Pet(name : String)
class Cat(name : String) extends Pet(name : String)
class Dog(name : String) extends Pet(name : String)
class BengalKitten(name : String) extends Cat(name : String)

class Queue[T](elts : T*) {
  var elems = List[T](elts : _* )

  def enqueue(elem : T) = elems ::: List(elem)

  def dequeue = {
    val result = elems.head
    elems = elems.tail
    result
  }
}

def examine(q : Queue[Cat]) {
  println("Examining: " + q.dequeue)
}
```

Необходима подсказка о типе

Теперь рассмотрим, как эти классы можно использовать в Scala. Вот простейшие примеры:

```
scala> examine(new Queue(new Cat("tiddles")))
Examining: line5$object$$iw$$iw$Cat@fb0d6fe

scala> examine(new Queue(new Pet("george")))
<console>:10: error: type mismatch;
  found   : Pet
  required: Cat
         examine(new Queue(new Pet("george")))
                           ^
```

Пока что очень похоже на Java. Разберем еще пару простых примеров:

```
scala> examine(new Queue(new BengalKitten("michael")))
Examining: line7$object$$iw$$iw$BengalKitten@464a149a

scala> var kitties = new Queue(new BengalKitten("michael"))
kitties: Queue[BengalKitten] = Queue@2976c6e4

scala> examine(kitties)
<console>:12: error: type mismatch;
 found : Queue[BengalKitten]
 required: Queue[Cat]
       examine(kitties)
               ^
```

Тут тоже нет почти ничего удивительного. В примере, где вы не создаете `kitties` как временную переменную, Scala выводит тип очереди, выясняя, что это `Queue[Cat]`, а потом принимает `michael` как подходящий тип — `BengalKitten`, который можно добавить в очередь. Во втором примере вы явно указываете тип `kitties`. Это означает, что Scala не сможет использовать выведение типов и, соответственно, не сумеет выполнить сопоставление параметров.

Далее мы рассмотрим, как устранять такие проблемы с типами путем использования *вариантности* языковой системы типов — в частности, в форме так называемой *ковариантности* (есть и другие возможные типы вариантности, но ковариантность используется чаще всего). В Java эта система очень гибкая, но иногда она может показаться несколько мудреной. Мы покажем, как она работает и в Scala, и в Java.

Ковариантность

Вы когда-нибудь задавались вопросами вроде «является ли `List<String>` подтипом `List<Object>` в Java»? Если да — то эта тема вас заинтересует.

В принципе, Java отвечает на этот вопрос «Нет», но можно устроить и так, что ответ будет «Да». Чтобы понять как, рассмотрим следующий фрагмент кода:

```
public class MyList<T> {
  private List<T> theList;
}

MyList<Cat> katzchen = new MyList<Cat>();
MyList<? extends Pet> petExt = petl;
```

Условие `? extends Pet` означает, что `petExt` — частично неизвестная переменная (знак `?` в типе Java всегда означает «неизвестно»). Но вы точно знаете, что параметр типа `MyList` должен быть `Pet` или подтипом `Pet`. В таком случае компилятор Java разрешает `petExt` иметь присвоенное значение, где параметр типа является подтипом.

Фактически мы здесь утверждаем, что `MyList<Cat>` является подтипом `MyList<? extends Pet>`. Обратите внимание, как было устроено такое отношение типов именно

при применении типа `MyList`, а не при его определении. Такое свойство типов называется *ковариантностью*.

В Scala ситуация складывается иначе, чем в Java. Там вариантность типа определяется не в момент использования типа. Напротив, в этом языке допускается явная ковариантность в момент объявления типа. Второй вариант обладает некоторыми преимуществами:

- во время компиляции компилятор может проверять систему на наличие случаев, которые не допускают ковариантности;
- любая гипотетическая нагрузка ложится на элемент, записывающий тип, а не на пользователей этого типа;
- можно встраивать интуитивно понятные взаимоотношения в базовые типы коллекций.

В результате возникает теоретический недостаток: технически такая вариативность оказывается менее гибкой, чем применяемая в Java вариативность по месту использования. Но на практике польза подхода, применяемого в Scala, вполне компенсирует эту проблему. По-настоящему продвинутые возможности дженериков Java редко используются большинством программистов.

Стандартные коллекции Scala, например `List`, реализуют ковариантность. Это означает, что `List[BengalKitten]` является подтипом `List[Cat]`, который, в свою очередь, является подтипом `List[Pet]`. Чтобы рассмотреть такой случай на практике, запустим интерпретатор:

```
scala> val kits = new BengalKitten("michael") :: Nil
kits: List[BengalKitten] = List(BengalKitten@71ed5401)

scala> var katzen : List[Cat] = kits
katzen: List[Cat] = List(BengalKitten@71ed5401)

scala> var haustieren : List[Pet] = katzen
haustieren: List[Pet] = List(BengalKitten@71ed5401)
```

Мы используем явные типы применительно к `var`, чтобы гарантировать, что Scala не будет чрезмерно сужать выведение типов.

На этом мы завершаем краткое рассмотрение дженериков Scala. Следующая крупная тема, к которой хотелось бы обратиться, — это новаторский подход Scala к параллелизму, связанный с использованием акторов. Это альтернативная модель, позволяющая явно управлять множественными потоками.

9.6. Знакомство с акторами

Модель Java с применением явных блокировок и синхронизации уже поизносилась. На момент создания языка это оказалась фантастическая инновация, но с ней была связана и значительная проблема. В сущности, модель параллелизма в Java — это балансирование между двумя нежелательными результатами.

При малом количестве блокировок параллельный код получается небезопасным, в нем будут легко возникать условия гонки. Если, напротив, блокировок будет

слишком много, код станет негибким и будет регулярно стопориться. В подобных условиях невозможно обеспечить значительный прогресс. Именно такое противоречие между безопасностью и гибкостью мы обсуждали в главе 4.

Модель, основанная на блокировках, требует учитывать все параллельные операции, которые могут одновременно разворачиваться в конкретный момент. Это означает, что по мере разрастания приложения становится сложнее учитывать все моменты, которые могут пойти неправильно. Хотя в Java существуют способы сглаживания подобных осложнений, основная проблема сохраняется. В Java ее можно полностью устранить только в таком релизе, который не будет обеспечивать обратной совместимости.

В альтернативных языках все можно начать сначала. Вместо того чтобы открывать программисту низкоуровневые детали работы с блокировками и потоками, такие языки могут предоставлять в своих средах времени исполнения специальные возможности, обеспечивающие дополнительную поддержку параллелизма.

В этой идее нет ничего удивительного. В конце концов, когда появился язык Java, многим разработчикам на C и C++ казалась странной идея о том, что среда времени исполнения сможет управлять памятью, а программист будет от этого избавлен.

Рассмотрим модель параллелизма, применяемую в Scala. Она основана на технологии, подразумевающей использование акторов и обеспечивающей особый (сравнительно простой) подход к параллельному программированию.

9.6.1. Весь код — театр

Актор — это объект, производный от `scala.actors.Actor` и реализующий метод `act()`. Вероятно, такая сущность напоминает вам об определении потока в Java. Самое серьезное отличие между потоками и акторами заключается в том, что акторы в большинстве случаев не взаимодействуют с помощью явно разделяемых данных.

Обратите внимание: работа с совместно используемыми (разделяемыми) данными относится к обязательному набору профессиональных навыков программиста. В Scala вам ничто не помешает разделять состояние между акторами, если вы этого пожелаете. Но такой стиль просто считается некрасивым. В качестве альтернативы совместного использования акторы применяют специальный механизм коммуникации, именуемый *почтовым ящиком* (mailbox). Почтовый ящик предназначен для отправки сообщения (рабочего элемента) в актор из другого контекста. На рис. 9.5 показано, как это делается.

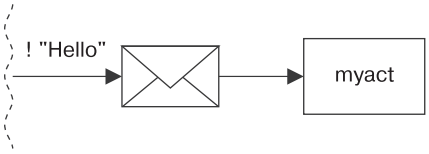


Рис. 9.5. Акторы и почтовые ящики в Scala

Для того чтобы реализовать актор, нужно просто дополнить класс `Actor`:

```
import scala.actors._
class MyActor extends Actor {
  def act() {
    ...
  }
}
```

Это немного напоминает способ объявления подкласса `Thread` в коде Java. Как и при обращении с потоками, необходимо приказать актору запуститься и перейти в состояние, в котором он сможет приступить к получению методов. Это делается с помощью метода `start()`.

Как вы, наверное, уже догадываетесь из опыта работы со Scala, язык также предоставляет удобный фабричный метод `actor`, позволяющий создавать новые акторы (аналогичным феноменом в Java можно считать статический фабричный метод, создающий анонимные реализации `Runnable`). Поэтому код Scala может быть вот таким кратким:

```
val myactor = actor {
  ...
}
```

Содержимое блока, передаваемое актору, преобразуется в содержимое метода `act()`. Кроме того, формируемые таким образом акторы не требуется запускать с помощью отдельного вызова `start()` — они запускаются автоматически.

Это интересный пример синтаксического сахара, но мы еще должны разобраться с центральным объектом модели параллелизма в Scala — с почтовым ящиком.

9.6.2. Обмен информацией с акторами через почтовый ящик

Отправить в актор сообщение от другого объекта совсем легко — вы просто вызываете метод ! применительно к объекту актора.

Но на стороне получателя вам понадобится кое-какой код для обработки сообщений, иначе они будут просто накапливаться в ящике. Кроме того, тело метода актора обычно должно работать в циклическом режиме, чтобы иметь возможность обрабатывать весь поток входящих сообщений. Рассмотрим практическое применение этого механизма на примере Scala REPL:

```
scala> import scala.actors.Actor._
       val myact = actor {
         while (true) {
           receive {
             case incoming => println("I got mail: "+ incoming)
           }
         }
}
```

```
myact: scala.actors.Actor = scala.actors.Actor$$anon$1@a760bb0

scala> myact ! "Hello!"
I got mail: Hello!

scala> myact ! "Goodbye!"
I got mail: Goodbye!

scala> myact ! 34
I got mail: 34
```

В этом примере метод `receive` используется для того, чтобы актор мог обработать сообщение. Он принимает в качестве аргумента блок, представляющий собой тело обрабатывающего метода, а обрабатывающий метод в Scala уже будет использоваться для операций с методом `receive`.

ПРИМЕЧАНИЕ

В принципе, модель Scala напоминает паттерн обработки, который мы обсудили в главе 4 (см. листинг 4.13). В том случае обрабатывающие потоки Java играли роль акторов, а LinkedBlockingQueue — роль почтового ящика. Scala обеспечивает простую поддержку этого паттерна на языковом и библиотечном уровне. Это действительно помогает сократить количество необходимого шаблонного кода.

Несмотря на то что мы рассмотрели очень простой пример, из него понятны многие основы работы с акторами:

- использование цикла в методе `actor` для обработки потока входящих сообщений;
- применение метода `receive` для обработки отдельных входящих сообщений;
- использование набора условий в качестве тела `receive`.

Последний момент заслуживает более подробного обсуждения. Набор условий определяет так называемую *частичную функцию* (partial function). Это удобно благодаря другому свойству акторов Scala, делающему их более функциональными, чем эквивалентные конструкции Java. Речь идет о том, что создаваемые здесь почтовые ящики Scala являются нетипизированными. Это означает, что вы можете послать в актор сообщение любого типа и настроить паттерны для получения сообщений различных типов, пользуясь типизированными паттернами и паттернами конструкторов, рассмотренными выше в этой главе.

Кроме этих основных возможностей, есть еще несколько рекомендаций по работе с акторами. Вот некоторые из них, старайтесь придерживаться их в собственном коде:

- делайте входящие сообщения неизменяемыми;
- попробуйте делать типы сообщений `case`-классами;
- не выполняйте внутри актора никаких блокирующих операций.

Не каждое сообщение может соответствовать всем этим рекомендациям, но в большинстве программ нужно пытаться выполнять как можно больше таких рекомендаций.

При работе с усложненными акторами часто бывает необходимо управлять и запуском, и остановкой его работы. Часто это делается с помощью цикла, использующего булево условие, которое контролирует останов актора. В зависимости от предпочитаемого варианта вы можете написать актор и в функциональном стиле так, чтобы у него не было состояния, которое могли бы затрагивать входящие сообщения.

Scala значительно полнее поддерживает параллельное программирование в акторном стиле. Мы лишь слегка коснулись этой темы. Для подробного ее изучения рекомендуем книгу *Scala in Action* Нилаяна Райчаудхири (Nilanjan Raychaudhuri) (издательство Manning, 2010).

9.7. Резюме

Язык Scala существенно отличается от Java:

- в Scala можно пользоваться функциональными приемами, обеспечивая более гибкий стиль программирования;
- благодаря выведению типов можно заставить статически типизированный язык действовать в более динамическом духе;
- продвинутая система типов Scala позволяет существенно расширить концепцию объектной ориентации, известную вам из Java.

В следующей главе мы поговорим о последнем из упомянутых выше альтернативных языков для виртуальной машины Java. Речь пойдет о Clojure, представляющем собой диалект Lisp. Этот язык во многих отношениях очень не похож на Java. В ходе работы мы будем опираться на концепции неизменяемости, функционального программирования и альтернативного параллелизма, которые затрагивались в этой главе. Мы покажем, как Clojure воплощает все эти идеи и выстраивает невероятно мощную и красивую программную экосистему.

10 Clojure: программирование повышенной надежности

В этой главе:

- концепция тождественности и состояния в Clojure;
- модель REPL в Clojure;
- синтаксис Clojure, структуры данных и последовательности;
- взаимодействие Clojure с Java;
- многопоточная разработка на языке Clojure;
- программная транзакционная память.

Язык Clojure весьма отличается по стилю от Java и других языков, рассмотренных нами выше. Clojure — это адаптированный для виртуальной машины Java вариант одного из старейших языков программирования — Lisp. Если вы не знаете Lisp — не волнуйтесь. Мы расскажем вам все необходимое о семействе языков Lisp, чтобы вы могли начать работать с Clojure.

Кроме большого багажа мощных приемов программирования, взятых из классического Lisp, Clojure предлагает и интереснейшие ультрасовременные технологии, важные для профессионального Java-разработчика. Благодаря такой комбинации качеств Clojure оказывается выдающимся языком для виртуальной машины Java, очень привлекательным для разработки приложений.

В качестве примеров из нового арсенала Clojure можно привести инструментарии для параллельного программирования на этом языке, а также структуры данных. Параллельные абстракции позволяют программистам писать гораздо более безопасный многопоточный код. Их можно комбинировать с присутствующей в Clojure абстракцией `seq` (это новый подход к работе с коллекциями и структурами данных), чтобы разработчик приобретал очень широкие возможности.

Для обеспечения всех этих возможностей Clojure организует некоторые языковые концепции принципиально иначе, нежели Java. Благодаря этой разнице Clojure интересен для изучения. Возможно, освоив этот язык, вы станете по-другому представлять себе программирование. Изучив Clojure, вы сможете более качественно программировать на любом языке.

Начнем с обсуждения того, как Clojure работает с состояниями и переменными. Разобрав несколько примеров, мы познакомимся с глоссарием языка, со специальными формами, на основе которых можно составить представление обо всем языке. Кроме того, мы подробно проработаем синтаксис Clojure, обеспечивающий использование структур данных, циклов и функций. Так будет легче разобраться с последовательностями — это одна из наиболее мощных абстракций в составе Clojure. В заключение этой главы мы рассмотрим еще две многообещающие возможности: тесную интеграцию между Clojure и Java, а также замечательную поддержку параллелизма.

10.1. Введение в Clojure

Начнем с изучения наиболее важных концептуальных отличий между Clojure и Java. Речь пойдет о работе с состоянием, переменными и хранением данных. Как показано на рис. 10.1, Java (подобно Groovy и Scala) имеет модель памяти и состояния. В соответствии с ней переменная представляет собой «коробочку» (на самом деле — местоположение в памяти), содержимое которой может изменяться с течением времени.

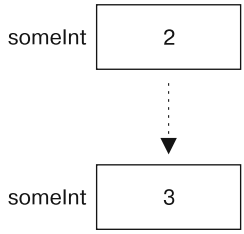


Рис. 10.1. Использование памяти в императивном языке

Язык Clojure построен немного иначе. Важнейшей концепцией в нем является *значение* (value). В качестве значений могут выступать числа, строки, векторы, словари, множества, а также многие другие элементы. Будучи создано, значение уже не изменяется. Это очень важный момент, поэтому подчеркнем его: в Clojure *созданное значение нельзя изменить — оно является неизменяемым*.

Это означает, что «коробочная» модель памяти из императивного языка, в которой есть изменяемое содержимое, неприменима к Clojure. На рис. 10.2 показано, как Clojure работает с состоянием и памятью. Язык создает ассоциацию между именем и значением.

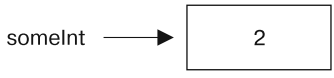


Рис. 10.2. Использование памяти в Clojure

Такая ассоциация называется *связыванием* (binding) и осуществляется с помощью специальной формы `(def)`. Специальные формы в Clojure эквивалентны

ключевым словам Java, но учтите, что *ключевым словом* (keyword) в Clojure называется иная сущность, с которой мы познакомимся ниже.

Синтаксис `(def)` такой:

```
(def <name> <value>)
```

Не волнуйтесь, если синтаксис кажется странноватым — для Lisp он совершенно нормален, вы привыкнете к нему очень быстро. Пока достаточно обратить внимание на то, что скобки упорядочены немного необычно, а метод вызывается так:

```
def(<name>, <value>)
```

Рассмотрим `(def)` на проверенном временем примере, использующем интерактивную среду Clojure.

10.1.1. Hello World на языке Clojure

Если вы еще не установили Clojure, посмотрите в приложении D, как это делается. Потом перейдите в тот каталог, куда вы установили Clojure, и выполните такую команду:

```
java -cp clojure.jar clojure.main
```

Она выводит пользовательское приглашение для работы с циклом «чтение — вычисление — печать» (REPL). Это интерактивная сессия. Вы будете проводить за такими сессиями немало времени, разрабатывая код на Clojure.

Часть `user=>` — это приглашение Clojure к началу сессии. Его можно считать продвинутым отладчиком или инструментом командной строки:

```
user=> (def hello (fn [] "Hello world"))
#'user/hello
user=> (hello)
"Hello world"
```

В этом коде мы начинаем работу с привязки идентификатора `hello` к значению. `(def)` всегда связывает идентификаторы (называемые в Clojure символами) со значениями. В фоновом режиме при этом также создается объект, называемый `var` и представляющий связь (и имя символа).

Что же за значение вы привязываете? Вот оно:

```
(fn [] "Hello world")
```

Это функция, которая является в Clojure подлинным значением (соответственно, она неизменяемая). Это функция, не принимающая никаких аргументов и возвращающая строку `"Hello world"`.

После привязывания вы выполняете ее с помощью команды `(hello)`. В таком случае среда времени исполнения Clojure выводит результат вычисления функции, который гласит `"Hello world"`.

На данном этапе следует открыть пример Hello World (если вы еще не сделали этого) и убедиться, что он работает правильно. Затем можно немного глубже исследовать данную тему.

10.1.2. Знакомство с REPL

Интерактивная среда REPL позволяет входить в код Clojure и выполнять функции на этом языке. В ней сохраняются результаты вычислений, выполненных ранее. Она позволяет работать в стиле, называемом *«исследовательское программирование»* (exploratory programming). Этот подход мы рассмотрим в подразделе 10.5.4. Суть его сводится к тому, что в ходе исследовательского программирования с кодом можно экспериментировать. Зачастую вам не помешает поиграть с REPL, выстраивая все более крупные функции и зная, что «кирпичики» этих функций корректны.

Сразу рассмотрим подобный пример. Первым делом следует отметить, что привязку символа к значению можно изменить с помощью другого вызова к `def`. Работать будем в среде REPL, но воспользуемся слегка измененным вариантом `(def)`, который называется `(defn)`:

```
user=> (hello)
"Hello world"
user=> (defn hello [] "Goodnight Moon")
#'user/hello
user=> (hello)
"Goodnight Moon"
```

Обратите внимание: исходная связь для работы с `hello` остается действительной до тех пор, пока вы ее не измените, — это основная черта REPL. Существует стабильное состояние, описывающее, какие функции связаны с какими значениями, и это состояние сохраняется между разными строками, которые вводит пользователь.

Возможность изменения конкретного значения, с которым связан символ, компенсирует в Clojure отсутствие изменяемого состояния. Clojure не позволяет изменяться содержимому «коробки с памятью», зато разрешает привязывать символ к различным неизменяемым значениям в разные моменты времени. Иными словами, `var` может указывать на различные значения в ходе жизненного цикла программы. Соответствующий пример показан на рис. 10.3.

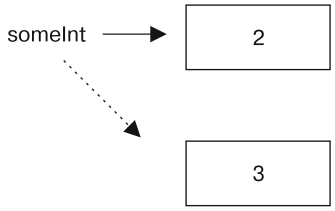


Рис. 10.3. Связи Clojure, изменяемые с течением времени

ПРИМЕЧАНИЕ

Иногда разница между изменяемым состоянием и различными связями довольно тонка, но усвоить эту концепцию очень важно. Помните, что изменяемое состояние подразумевает изменение содержимого блока памяти, а повторная привязка означает, что в различные моменты времени символ указывает на разные блоки.

В последнем примере кода мы также коснулись еще одной концепции Clojure — макрокоманды (defn), позволяющей определить функцию. Макрокоманды — это одна из ключевых составляющих Lisp-подобных языков. Основная идея заключается в том, чтобы максимально сгладить разницу между встроенными конструкциями и обычным кодом.

Макрокоманды позволяют создавать формы, действующие в виде встроенного синтаксиса. Создание макрокоманды — сложная тема, но если вы научитесь этому искусству, то сможете изготавливать себе невероятно мощные инструменты.

Таким образом, истинные языковые примитивы системы (специальные формы) могут использоваться для выстраивания ядра языка так, что вы даже не заметите границы между этими примитивами и ядром. Макрокоманда (defn) иллюстрирует как раз такую возможность. Она просто облегчает привязку значения к символу (и, разумеется, позволяет создавать удобные var).

10.1.3. Как делаются ошибки

А что делать, если вы допустите ошибку? Например, пропустите скобки [] (они употребляются при объявлении функции и означают, что данная функция не принимает аргументов):

```
user=> (defn hello "Goodnight Moon")
#'user/hello
user=> (hello)
java.lang.IllegalArgumentException: Wrong number of args (0) passed to:
user$hello (NO_SOURCE_FILE:0)
```

Что здесь произошло? Ваш идентификатор hello оказался связан с чем-то почти бессмысленным. В среде REPL ситуацию можно исправить благодаря обычной повторной привязке:

```
user=> (defn hello [] (println "Dydh da an Nor"))
➥ ; "Hello World" на корнуэльском
#'user/hello
user=> (hello)
Dydh da an Nor
nil
user=>
```

Как вы могли догадаться из предыдущего фрагмента кода, символ точки с запятой (;) означает, что вся строка является комментарием, а (println) — это функция, выводящая на экран строку. Обратите внимание: (println), как и все функции, возвращает значение, передаваемое обратно в REPL по завершении выполнения функции. Это значение равно nil, в Clojure его можно считать эквивалентом null из Java.

10.1.4. Учимся любить скобки

В цеху программистов всегда любили пошутить и покаламбурить. Одна из самых старых шуток гласит, что аббревиатура LISP расшифровывается как Lots of Irritating Silly Parentheses («Множество раздражающих глупых скобок»). Реальная

расшифровка, конечно же, более прозаична — это всего лишь List Processing («Обработка списков»). Эта немного самоуничижительная шутка популярна среди самих Lisp-программистов — отчасти потому, что в ней признается печальная правда о том, насколько Lisp сложен для изучения.

На самом деле эта беда сильно преувеличена. Синтаксис Lisp действительно отличается от того, к которому привыкли многие программисты, но это не мешает работать, что бы иногда ни говорили. Кроме того, в Clojure воплощено много инноваций, помогающих еще сильнее снизить начальный барьер, который требуется преодолеть для работы с языком.

Вновь взглянем на пример Hello World. Чтобы вызвать функцию, возвращающую `"Hello World"`, вы написали следующее:

```
(hello)
```

Если бы вы писали это на Java, то получилось бы нечто такое:

```
hello();
```

предполагается, что где-то в созданном вами классе была определена функция под названием `hello`.

Но язык Clojure построен иначе. В нем не употребляются выражения вида `myFunction(someObj)`, код записывается вот так: `(myFunction someObj)`. Такой способ записи называется *польской нотацией*, так как он был разработан польскими математиками.

Если вы изучали теорию компиляции, то можете спросить, связана ли данная нотация с такими концепциями, как абстрактное синтаксическое дерево (AST). Если ответить кратко — да, связь присутствует. Программа Clojure (или другого диалекта Lisp) может записываться польской нотацией (сами Lisp-программисты обычно называют такую запись *s-выражением* или *точечным выражением*) и оказываться очень простым и непосредственным представлением AST-дерева данной программы.

Можете считать, что программа Lisp пишется непосредственно в контексте своего синтаксического дерева. Нет никаких явных отличий между структурой данных, представляющей программу Lisp, и кодом этой программы. Таким образом, в языке Lisp код и данные практически взаимозаменяемы. Именно поэтому в языке и используется довольно странная нотация — в Lisp-подобных языках она призвана стереть грань между встроенными языковыми примитивами и пользовательским и библиотечным кодом. Эти возможности обладают таким большим потенциалом, что вполне компенсируют некоторую странность синтаксиса, непривычного для Java-программиста.

Подробнее познакомимся с синтаксисом и начнем писать программы на Clojure.

10.2. Поиск Clojure — синтаксис и семантика

В предыдущем разделе нам повстречались специальные формы `(def)` и `(fn)`. Есть еще и другие специальные формы, с которыми нужно познакомиться уже сейчас. Они составляют базовый глоссарий языка. Кроме того, существует еще множество

полезных форм и макрокоманд, с которыми вы постепенно познакомитесь на практике.

Язык Clojure изобилует множеством полезных функций, помогающих выполнять широкий диапазон самых разнообразных мыслимых задач. Не пугайтесь этого многообразия — просто примите его. Радуйтесь, что при решении на Clojure многих практических проблем программирования, с которыми вы можете столкнуться, кто-то уже сделал за вас массу сложной работы.

В этом разделе мы обсудим базовый рабочий набор специальных форм, затем перейдем к нативным типам данных, применяемым в Clojure (они эквивалентны коллекциям Java). После этого мы познакомим вас с естественным стилем программирования на Clojure — в этом языке основная роль отводится не переменным, а функциям. На более глубоком уровне в наших программах по-прежнему будет ощущаться объектно-ориентированная природа виртуальной машины Java, но акцент, который Clojure делает на функциях, наделит вас особой силой.

10.2.1. Базовый курс по работе со специальными формами

В табл. 10.1 приведены определения некоторых специальных форм, наиболее широко используемых в Clojure. Чтобы максимально эффективно использовать эту таблицу, просмотрите ее сейчас, а потом возвращайтесь к ней, когда будете изучать конкретные примеры в разделе 10.3 и далее.

Таблица 10.1. Некоторые из специальных форм Clojure

Специальная форма	Значение
(def <symbol> <value?>)	Привязывает символ к значению (при наличии значения). При необходимости создает var, соответствующий символу
(fn <name>? [<arg>*] <expr>*)	Возвращает функциональное значение, принимающее указанные аргументы arg и применяющее их к выражениям expr. Часто комбинируется с (def) для образования форм типа (defn)
(if <test> <then> <else>?)	Если при вычислении test получается логическое значение true, то необходимо вычислить и возвратить значение then. В противном случае нужно вычислить и возвратить значение else при наличии else
(let [<binding>*] <expr>*)	Совмещает значения с локальным именем и неявно определяет область видимости. Обеспечивает доступность полученного псевдонима внутри всех выражений expr в пределах области видимости let
(do <expr>*)	По порядку интерпретирует выражения expr и выдает значение последнего
(quote <form>)	Возвращает форму в исходном виде (без обработки). Принимает только одну форму form и игнорирует все другие аргументы
(var <symbol>)	Возвращает var, соответствующий символу (при этом мы получаем объект Clojure для виртуальной машины Java, а не значение)

Это далеко не полный список специальных форм, к тому же значительное их количество может использоваться несколькими способами. Таблица 10.1 — это начальный набор для простейших практических нужд, она не претендует на какую-либо полноту.

Теперь у вас есть представление о синтаксисе некоторых базовых специальных форм. Обратимся к структурам данных Clojure и рассмотрим, как формы позволяют оперировать данными.

10.2.2. Списки, векторы, словари и множества

В языке Clojure есть несколько нативных структур данных. Самая известная из них — список, который в Clojure является односвязным.

Обычно списки заключаются в скобки. С синтаксической точки зрения это представляет небольшую проблему, поскольку круглые скобки используются и с общими формами. В частности, круглые скобки применяются и при интерпретации функциональных вызовов. В результате часто возникает следующая характерная для новичков синтаксическая ошибка:

```
1:7 user=> (1 2 3)
java.lang.ClassCastException: java.lang.Integer cannot be cast to
clojure.lang.IFn (repl-1:7)
```

Проблема заключается в том, что, поскольку Clojure очень гибко обращается со значениями, язык ожидает встретить здесь функциональное значение (или символ, разрешаемый в такое значение) в качестве первого аргумента. Тогда язык сможет вызвать эту функцию и передать ей `2` и `3` как аргументы. Значение `1` не является функцией, поэтому Clojure не сможет скомпилировать эту форму. Принято говорить, что такое точечное выражение является неправильным. Только правильные точечные выражения могут быть формами Clojure.

Чтобы решить данную проблему, воспользуйтесь формой `(quote)`, которую мы упомянули в предыдущем разделе. У нее есть удобный сокращенный вариант — `'`. В результате имеем два эквивалентных способа записи такого списка:

```
1:22 user=> '(1 2 3)
(1 2 3)
1:23 user=> (quote (1 2 3))
(1 2 3)
```

Обратите внимание: `(quote)` обрабатывает свои документы особым образом. В частности, эта форма не пытается интерпретировать аргумент, поэтому не возникает ошибки, которая была бы обусловлена отсутствием функционального значения в первой ячейке.

В Clojure есть векторы — это сущности, напоминающие массивы (но на самом деле мы не сильно погрешим против истины, если будем считать списки принципиально похожими на `LinkedList` из Java, а векторы — похожими на `ArrayList`). У векторов есть удобная литеральная форма, в которой применяются квадратные скобки. Поэтому все следующие варианты эквивалентны:

```
1:4 user=> (vector 1 2 3)
[1 2 3]
```

```
1:5 user=> (vec '(1 2 3))
[1 2 3]
1:6 user=> [1 2 3]
[1 2 3]
```

Мы уже встречались с векторами. Когда мы объявляли функцию Hello World и другие функции, мы использовали вектор для указания параметров, принимаемых объявляемой функцией. Обратите внимание: форма `(vec)` принимает список и создает из него вектор. В свою очередь, `(vector)` — это форма, принимающая несколько отдельных символов и возвращающая состоящий из них вектор.

Функция `(nth)` для работы с коллекциями принимает два параметра: коллекцию и индекс. Она напоминает метод `get()`, применяемый с интерфейсом Java `List`. Такая функция может использоваться с векторами и списками, но также пригодна для работы с коллекциями и Java и даже строками, которые воспринимаются как коллекции символов. Вот пример:

```
1:7 user=> (nth '(1 2 3) 1)
2
```

Clojure также поддерживает словари (очень напоминающие элементы `HashMap` из Java), применяя при этом следующий простой литеральный синтаксис:

```
{key1 value1 key2 "value2}
```

Чтобы обратно получить значение из словаря, используем очень простой синтаксис:

```
user=> (def foo {"aaa" "111" "bbb" "2222"})
#'user/foo
user=> foo
{"aaa" "111", "bbb" "2222"}
user=> (foo "aaa")
"111"
```

Отметим здесь одну важную стилистическую черту — в коде используются ключи, перед которыми ставится двоеточие. Именно такие ключи называются в Clojure *ключевыми словами*.

```
1:24 user=> (def martijn {:name "Martijn Verburg",
➥ :city "London", :area "Highbury"})
#'user/martijn
1:25 user=> (:name martijn)
"Martijn Verburg"
1:26 user=> (martijn :area)
„Highbury"
1:27 user=> :area
:area
1:28 user=> :foo
:foo
```

Отметим несколько важных деталей о работе с ключевыми словами и словарями:

- ключевое слово в Clojure — это функция, принимающая один аргумент, который обязательно должен быть словарем;

- при вызове функции — ключевого слова применительно к словарю возвращается значение, соответствующее функции — ключевому слову в словаре;
- при работе с ключевыми словами в синтаксисе наблюдается удобная симметрия, так как и конструкция `(my-map :key)`, и `(:key my-map)` совершенно допустимы;
- в качестве значения ключевое слово возвращает само себя;
- ключевые слова перед использованием не обязательно объявлять или сопровождать формой `def`;
- не забывайте, что функции Clojure являются значениями, то есть они вполне могут применяться в качестве ключей в словарях;
- запятые можно использовать для разделения пар ключ/значение (хотя в этом и нет необходимости); в Clojure запятая приравнивается к пробелу;
- в качестве ключей для словарей Clojure могут применяться не только ключевые слова, но и другие символы, но синтаксис ключевых слов очень удобен и стоит его придерживаться при написании кода.

Наряду со словарными литералами в Clojure также имеется функция `(map)`. Но не ведитесь на, казалось бы, очевидное название. В отличие от `(list)`, функция `(map)` не создает словаря. Вместо этого она применяет специально предоставляемую функцию к каждому из элементов коллекции по очереди и выстраивает новую коллекцию (если быть точными — последовательность Clojure; о таких последовательностях мы подробно поговорим в разделе 10.4) из новых значений, которые были возвращены этой функции.

```
1:27 user=> (def ben {:name "Ben Evans", :city "London", :area "Holloway"})
#'user/ben
1:28 user=> (def authors [ben martijn])
#'user/authors
1:29 user=> (map (fn [y] (:name y)) authors)
("Ben Evans" "Martijn Verburg")
```

Существуют и дополнительные формы `(map)`, способные обрабатывать по нескольку коллекций за раз, но наиболее распространена форма, принимающая в качестве ввода только одну коллекцию.

Clojure также поддерживает работу с множествами, которые очень напоминают `HashSet` из Java. Множества также предусматривают краткую форму структур данных:

```
#{"apple" "pair" "peach"}
```

Такие структуры данных являются основными «строительными блоками» для создания программ Clojure.

Одна из деталей, которая может удивить специалиста по Java, — это отсутствие какого-либо прямого упоминания об объектах как первоклассных сущностях. Это еще не означает, что Clojure не является объектно-ориентированным языком, просто он трактует объектную ориентацию совсем не так, как Java. Картина мира Java состоит из статически типизированных пакетов с данными и кода, заключенного в явных определениях классов или указанных пользователем типах данных.

Clojure, напротив, делает акцент на формах и функциях, хотя на уровне виртуальной машины Java эти формы и функции и реализуются как объекты.

Такое философское различие между Clojure и Java со всей ясностью проявляется в написании кода на двух этих языках. Чтобы полностью понимать «точку зрения» Clojure, необходимо писать программы на этом языке и знать преимущества, обретаемые в отрыве от типичных для Java объектно-ориентированных конструкций.

10.2.3. Арифметика, проверка на равенство и другие операции

В Clojure отсутствуют такие операторы, которые знакомы вам из работы с Java. Итак, как же нам сложить два числа? В Java это делается проще простого:

```
3 + 4
```

Но в Clojure операторов нет. Вместо них приходится использовать функцию:

```
(add 3 4)
```

Что ж, и так неплохо, но мы можем сделать лучше. Поскольку в Clojure нет операторов, нет и необходимости резервировать какие-либо клавиатурные символы для представления этих операторов. Это означает, что имена наших функций могут быть более необычными, чем в Java, и мы вполне можем написать так:

```
(+ 3 4)
```

Функции Clojure зачастую имеют *переменное* количество аргументов, то есть принимают переменное количество элементов ввода. Поэтому можно написать и так:

```
(+ 1 2 3)
```

В результате получим значение `6`.

При работе с формами равенства (они эквивалентны применяемым в Java элементам `equals()` и `==`) ситуация немного усложняется. В Clojure есть две основные формы, связанные с равенствами: `(=)` и `(identical?)`. Обратите внимание: оба примера красноречиво свидетельствуют, что благодаря отсутствию операторов в Clojure в названиях функций можно использовать гораздо больше разных символов. Кроме того, `(=)` — это одиночный знак равенства, так как в Clojure отсутствует феномен присваивания, известный вам из Java-подобных языков.

Следующий фрагмент кода из интерактивной среды REPL создает список `list-int` и вектор `vect-int`, после чего применяет к ним логику тождества:

```
1:1 user=> (def list-int '(1 2 3 4))
#'user/list-int
1:2 user=> (def vect-int (vec list-int))
#'user/vect-int
1:3 user=> (= vect-int list-int)
true
1:4 user=> (identical? vect-int list-int)
false
```

Основной момент здесь заключается в том, что при работе с коллекциями форма (=) проверяет, содержат ли сравниваемые коллекции одни и те же объекты в одном и том же порядке (для list-int и vect-int это так). В свою очередь, (identical?) проверяет, действительно ли два объекта являются экземплярами одного и того же объекта.

Вы также могли заметить, что в именах наших символов не используется «верблюжий» регистр (Camel case). В Clojure это нормально. Обычно все символы записываются в нижнем регистре, между словами ставятся дефисы.

ИСТИННО И ЛОЖНО В CLOJURE

В Clojure существуют два значения для передачи логического «ложно»: false и nil. Все остальные случаи являются логически верными. Подобная ситуация складывается во многих динамических языках, но программистов, привыкших к работе с Java, она на первых порах может озадачить.

Ознакомившись с базовыми структурами, сочленим некоторые из рассмотренных нами специальных форм и функций и напишем более длинный функциональный пример Clojure.

10.3. Работа с функциями и циклами в Clojure

В этом разделе мы переходим к сути программирования на Clojure. Начнем писать функции для осуществления операций над данными и сосредоточимся на том привилегированном положении, которое занимают функции в Clojure. Далее поговорим о циклических конструкциях Clojure, потом — о макросах чтения и формах для диспетчеризации. В заключение обсудим подход Clojure к функциональному программированию, а также поговорим о замыканиях.

Лучше всего начать всю эту работу с рассмотрения примера. Поэтому изучим несколько простых случаев и подготовим почву для изучения некоторых мощных приемов функционального программирования, доступных в Clojure.

10.3.1. Простые функции Clojure

В листинге 10.1 определяются три функции. Две из них — очень простые, каждая имеет по одному аргументу; третья функция немного сложнее.

Листинг 10.1. Определение простых функций в Clojure

```
(defn const-fun1 [y] 1)

(defn ident-fun [y] y)

(defn list-maker-fun [x f]
  (map (fn [z] (let [w z]
    (list w (f w))
 )) x))
```

В этом листинге (const-fun1) принимает значение и возвращает 1, а (ident-fun) принимает значение и возвращает то же самое значение. Математики назвали бы их константной и тождественной функциями соответственно. Как видите, при определении функции векторные литералы используются для обозначения аргументов функции и для формы (let).

Третья функция немного сложнее. Функция (list-maker-fun) принимает два аргумента. Первый — это вектор значений (x), над которыми мы собираемся проводить операции, а второй — это значение, которое должно быть функцией.

Рассмотрим, как работает list-maker-fun:

Листинг 10.2. Работа с функциями

```
user=> (list-maker-fun ["a"] const-fun1)
(("a" 1))
user=> (list-maker-fun ["a" "b"] const-fun1)
(("a" 1) ("b" 1))
user=> (list-maker-fun [2 1 3] ident-fun)
((2 2) (1 1) (3 3))
user=> (list-maker-fun [2 1 3] "a")
java.lang.ClassCastException: java.lang.String cannot be cast to
clojure.lang.IFn
```

Обратите внимание: при вводе этих выражений в среду REPL вы взаимодействуете с компилятором Clojure. Выражение (list-maker-fun [2 1 3] "a") не скомпилируется, так как (list-maker-fun) ожидает, что второй аргумент будет функцией, а строка функцией не является. В разделе 10.5 мы расскажем о том, что с точки зрения виртуальной машины функции Clojure являются объектами, реализующими clojure.lang.IFn.

Этот пример показывает, что при взаимодействии с REPL вы так или иначе имеете дело с определенным объемом статической типизации. Проблема в том, что Clojure не является интерпретируемым языком. Даже в среде REPL любая типизированная форма Clojure компилируется в байт-код виртуальной машины Java и привязывается к действующей системе. Функция Clojure компилируется в байт-код виртуальной машины Java на этапе определения, поэтому возникает исключение ClassCastException, обусловленное нарушением правил типизации на виртуальной машине Java.

В листинге 10.3 показан сравнительно объемный фрагмент кода Clojure, представляющий собой преобразование Шварца. Это отдельная страница истории программирования. Такое преобразование приобрело популярность в контексте языка Perl в 1990-е годы. Идея его заключается в том, чтобы произвести над вектором операцию сортировки, но не на основе предоставленного вектора, а на базе какого-либо свойства элементов этого вектора. Для нахождения тех значений свойств, по которым будет производиться сортировка, применительно к элементам вызывается кодирующая функция.

При определении преобразования Шварца в листинге 10.3 вызывается кодирующая функция key-fn. Когда вы хотите вызвать саму функцию (schwartz), необходимо также предоставить функцию, которая будет применяться для кодирования. В листинге 10.3 мы воспользуемся старой доброй функцией (ident-fun), уже знакомой нам из листинга 10.1.

Листинг 10.3. Преобразование Шварца

```
1:65 user=> (defn schwartz [x key-fn]
  (map (fn [y] (nth y 0))
       (sort-by (fn [t] (nth t 1))
                (map (fn [z] (let [w z]
                     (list w (key-fn w)))
                ) x))))
#'user/schwartz
1:66 user=> (schwartz [2 3 1 5 4] ident-fun)
(1 2 3 4 5)
1:67 user=> (apply schwartz [[2 3 1 5 4] ident-fun])
(1 2 3 4 5)
```


Код выполняется в три отдельных этапа.

1. Создается список, состоящий из пар.
2. Пары сортируются на основании значений кодирующей функции.
3. Создается новый список. При этом берется лишь исходное значение из каждой пары, входящей в отсортированный список пар (значения кодирующей функции отбрасываются).

Соответствующая схема показана на рис. 10.4.

Обратите внимание: в листинге 10.3 мы встретили новую форму — `(sort-by)`. Это функция, принимающая два аргумента: функцию, используемую для выполнения сортировки, и вектор, который будет сортироваться. Мы также продемонстрировали форму `(apply)`, принимающую два аргумента: функцию, которую следует вызвать, и вектор аргументов, которые требуется передать этой функции.

С преобразованием Шварца связан один любопытный факт — Рэндалл Шварц (Randall Schwartz), по имени которого названа эта операция, сознательно передразнивал Lisp, когда впервые предложил такое преобразование на языке Perl. Теперь мы делаем такое преобразование в коде Clojure — то есть мы описали полный круг и вернулись обратно на Lisp!

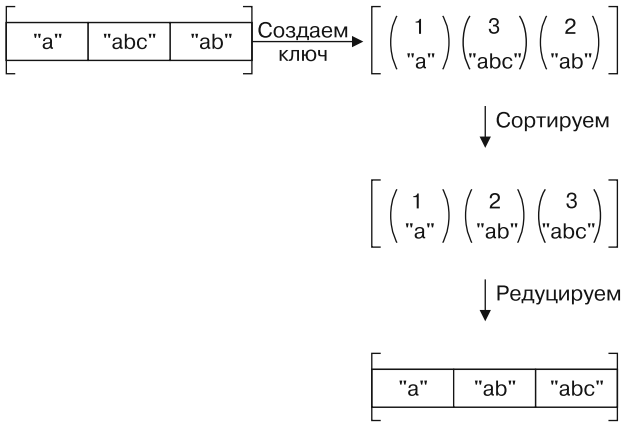


Рис. 10.4. Преобразование Шварца

Преобразование Шварца — полезный пример, и позже мы к нему вернемся. Дело в том, что он сложен как раз настолько, чтобы на нем можно было продемонстрировать несколько интересных концепций.

А пока пойдем дальше и поговорим о циклах в Clojure — на выбор разработчику предлагаются `for`, `while` и еще несколько типов циклов. Как правило, основу цикла составляет повторяющееся выполнение группы инструкций до тех пор, пока не будет соблюдено определенное условие (зачастую выражаемое в виде изменяемой переменной).

Здесь в Clojure возникает путаница. Допустим, как мы можем выразить цикл `for`, если в языке нет изменяемых переменных, которые могли бы использоваться как индекс цикла? В более традиционных диалектах Lisp такая проблема обычно решается путем переписывания итеративных циклов в форму, использующую рекурсию. Но на виртуальной машине Java не гарантируется оптимизация концевой рекурсии (такая оптимизация требуется в языке Scheme и других диалектах Lisp). Поэтому при использовании рекурсии стек может просто «взорваться».

Но Clojure предлагает удобные конструкции, обеспечивающие циклическое поведение без увеличения объема стека. Одна из наиболее распространенных — `loop-recur`. В следующем фрагменте кода показано, как можно использовать `loop-recur` для выстраивания простой конструкции, напоминающей цикл `for` из Java.

```
(defn like-for [counter]
(loop [ctr counter]
  (println ctr)
  (if (< ctr 10)
    (recur (inc ctr))
     ctr
 )))
```

Форма `(loop)` принимает вектор аргументов локальных имен для символов — фактически она совершает совмещение имен, как и `(let)`. Затем, когда выполнение программы доходит до формы `(recur)` (в данном примере это возможно лишь тогда, когда псевдоним `ctr` меньше 10), форма `(recur)` выполняет передачу управления обратно к форме `(loop)`, но уже с указанием нового значения. Таким образом, мы можем выстраивать итеративные конструкции (например, циклы `for` и `while`), но при этом сохраняется рекурсивный вариант реализации.

Далее перейдем к следующей нашей теме — рассмотрим удобную краткую синтаксическую форму Clojure, которая поможет сделать наши программы еще более лаконичными.

10.3.2. Макросы чтения и диспетчеризация

У Clojure есть такие синтаксические черты, которые могут удивить многих Java-программистов. Одна из таких черт — это отсутствие операторов. При отсутствии операторов в языке возникает побочный эффект: смягчаются типичные для Java ограничения на то, какие символы могут использоваться в названиях функций, а какие — не могут. Нам уже встречались такие функции, как `(identical?)`, которые

в Java были бы недопустимы. Но мы пока не пытались разобраться, какие именно знаки недопустимы в названиях символов Clojure.

Такие знаки перечислены в табл. 10.2. Это знаки, зарезервированные в Clojure для нужд синтаксического анализатора. Обычно их называют *макросами чтения* (reader macros).

Таблица 10.2. Макросы чтения

Знак	Название	Значение
'	Кавычка	Полная форма — (quote). Выдает невычисленную форму
;	Комментарий	Указывает, что вся строка после точки с запятой является комментарием. Аналогично // в Java
\	Символ	Создает литерал (самозначимый символ)
@	Разыменование	Имеет полную форму (deref), принимающую объект var и возвращающую в этом объекте значение (таким образом, она обратна форме (var)). В контексте транзакционной памяти имеет дополнительное значение, см. раздел 10.6
^	Метаданные	Прикрепляет к объекту словарь с метаданными. Подробнее см. в документации по языку Clojure
`	Синтаксическая кавычка	Разновидность кавычки, применяемая при определении макросов. Это элемент не для новичков. Подробнее см. в документации по языку Clojure
#	Диспетчеризация	Имеет несколько разновидностей. Подробнее см. в табл. 10.3

Макрос чтения, применяемый для диспетчеризации, существует в нескольких разновидностях, в зависимости от того, что следует за символом #. Эти формы перечислены в табл. 10.3.

Таблица 10.3. Различные формы макроса чтения для диспетчеризации

Форма диспетчеризации	Значение
#'	Имеет полную форму (var)
#{}	Создает литерал множества. О таких литералах мы говорили в подразделе 10.2.2
#()	Создает анонимный функциональный литерал. Форма удобна для однократного использования, когда (fn) кажется слишком пространной
#_	Пропускает следующую форму. Может применяться для создания многострочных комментариев, вот так: #_(... multi-line ...)
#"<pattern>"	Создает литерал регулярного выражения (подобный объекту Java java.util.regex.Pattern)

Вот еще несколько моментов, которые необходимо учитывать, работая с формами диспетчеризации. Форма #' позволяет понять, почему REPL действует после (def) именно так, а не иначе:

```
1:49 user=> (def someSymbol)
#'user/someSymbol
```

Форма (def) возвращает новоиспеченный объект var под названием someSymbol, относящийся к актуальному пространству имен. Итак, #'user/someSymbol — полное значение, возвращаемое от (def).

Анонимный функциональный литерал также имеет инновационную черту, позволяющую сократить код. Такой литерал обходится без вектора аргументов, так как использует специальный синтаксис, позволяющий читающему механизму Clojure дедуктивно вывести, сколько аргументов требуется для функционального литерала. Перепишем преобразование Шварца и посмотрим, как используется этот синтаксис.

Листинг 10.4. Переписанное преобразование Шварца

```
(defn schwartz [x f]
  (map #(nth %1 0)                          Анонимные
       (sort-by #(nth %1 1)                 функциональные
                (map #(let [w %1]           литералы
                        (list w (f w))
                     ) x))))
```

Поскольку %1 используется в качестве подстановочного символа для аргумента функционального литерала (для последующих аргументов применяются %2, %3 и т. д.), работать с кодом становится очень удобно, да и читать его не составляет труда. Такая визуальная подсказка может по-настоящему помочь программисту, подобно тем символам стрелок, которые применяются с литералами в Scala. Об этом мы говорили в подразделе 9.3.6.

Итак, мы видим, что язык Clojure коренным образом зависит от функций, играющих в нем роль вычислительных первоэлементов, а не от объектов, которые являются основой таких языков, как Java. Естественно, такой язык просто создан для функционального программирования, о котором мы и поговорим далее.

10.3.3. Функциональное программирование и замыкания

Мы готовимся вступить в неизведанный мир функционального программирования на языке Clojure. Правда, не такой уж он и неизведанный. На самом деле мы уже занимаемся функциональным программированием на протяжении всей этой главы — мы только пока не говорили об этом, чтобы вас заранее не озадачивать.

Как мы уже упоминали в подразделе 7.3.2, в функциональном программировании функция является значением. Функции можно передавать, записывать в переменные, манипулировать ими, так же как числом 2 или словом "hello". Ну и? Мы делали это уже в нашем первом примере: (def hello (fn []"Hello world")). Мы создали функцию (не принимающую аргументов и возвращающую строку "Hello world") и связали ее с символом hello. Функция была просто значением, ничем принципиально не отличавшимся от значения 2.

В подразделе 10.3.1 мы рассмотрели преобразование Шварца как пример функции, принимающей в качестве ввода другую функцию. Итак, перед нами обычная функция, принимающая в качестве вводимого значения конкретный тип. Единственная ее характерная черта заключается в том, что этот вводимый тип — функция.

А что насчет замыканий? Это уж действительно нечто пугающее, верно? Ну не настолько. Взглянем на простой пример — надеемся, он напомнит вам некоторые примеры на Scala, разобранные в главе 9.

```
1:5 user=> (defn adder [constToAdd] #(+ constToAdd %1))
#'user/adder
1:6 user=> (def plus2 (adder 2))
#'user/plus2
1:7 user=> (plus2 3)
5
1:8 user=> 1:9 user=> (plus2 5)
7
```

Сначала мы написали функцию под названием `(adder)`. Она создает другие функции. Если вам знаком паттерн Factory Method («Фабричный метод») из Java, то можете считать приведенный выше пример аналогом этого паттерна. Нет ничего странного в функциях, которые выдают в качестве возвращаемых значений другие функции. В этом и заключается концепция, в рамках которой функции воспринимаются как обычные значения.

Обратите внимание и на то, что в этом примере используется сокращенная форма анонимных функциональных литералов `#()`. Функция `(adder)` принимает число и возвращает функцию, а функция, возвращенная от `(adder)`, принимает один аргумент.

Затем вы используете `(adder)` для определения новой формы — `(plus2)`. Это функция, принимающая один числовой аргумент и прибавляющая к нему 2. Таким образом, значение, которое было привязано к `constToAdd` внутри `(adder)`, было равно 2. Теперь создадим новую функцию:

```
1:13 user=> (def plus3 (adder 3))
#'user/plus3
1:14 user=> (plus3 4)
7
1:15 user=> (plus2 4)
6
```

Здесь мы видим, как можно сделать еще одну функцию — `(plus3)`, в которой к `constToAdd` будет привязано другое значение. Принято говорить, что функции `(plus3)` и `(plus2)` «захватили», или «замкнули», значение из своего окружения. Обратите внимание: значения, захваченные `(plus3)` и `(plus2)`, разные, и определение `(plus3)` никак не повлияло на значение, захваченное `(plus2)`.

Функции, «замыкающие» определенные значения в своем окружении, называются *замыканиями* `(closure)`. `(plus3)` и `(plus2)` — это примеры замыканий. Паттерн, при котором функция, способная создавать другие функции, сама возвращает другую функцию с каким-то замкнутым элементом, очень распространен в языках, использующих замыкания.

Теперь обратимся еще к одной мощной возможности Clojure — последовательностям. Последовательности `(sequence)` немного напоминают коллекции или ите-

раторы Java, но все же значительно отличаются от них. Последовательности — важнейшая часть кода Clojure, они опираются на сильные стороны языка и позволяют по-новому взглянуть на то, как в Java обрабатываются подобные сущности.

10.4. Введение в последовательности Clojure

Рассмотрим пример интерфейса `Iterator` из Java, показанный в следующем фрагменте кода. Это немного старомодный способ использования итератора. На самом деле именно такую форму имел цикл `for` в Java 5 на внутрисистемном уровне:

```
Collection<String> c = ...;

for (Iterator<String> it = c.iterator(); it.hasNext();) {
  String str = it.next();
  ...
}
```

Такой механизм удобен при циклическом переборе небольшой коллекции, например `Set` или `List`. Но интерфейс `Iterator` имеет лишь методы `next()` и `hasNext()`, а также необязательный метод `remove()`.

Принципиальные проблемы при работе с итераторами Java. Но при работе с итераторами Java существует и определенная проблема. Интерфейс `Iterator` не предоставляет для работы с коллекциями такого богатого набора методов, как бы нам хотелось. С помощью `Iterator` из Java вы можете выполнять лишь две операции:

- проверять, остались ли еще в коллекции элементы;
- получать следующий элемент, продвигая итератор на один шаг.

Суть проблем, возникающих при работе с `Iterator`, заключается в том, что получение следующего элемента и продвижение итератора объединены в одну операцию (рис. 10.5). Это означает, что вы никак не сможете просмотреть следующий элемент коллекции, решить, нуждается ли он в специальной обработке, и, если так, обработать его и вернуться к предыдущему элементу.

Уже сам акт получения следующего элемента от итератора изменяет этот элемент. Таким образом, изменение элементов неотделимо от подхода Java к коллекциям и итераторам, в результате чего создать на Java надежное многопроходное решение практически невозможно.

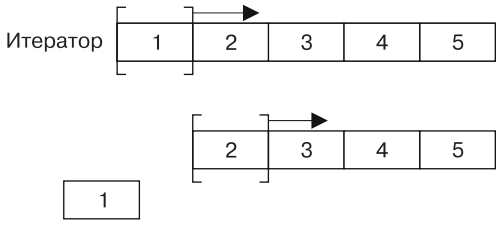


Рис. 10.5. Природа итераторов в Java

Абстракция ключа в Clojure. Clojure иначе подходит к данной проблеме. В ядре языка Clojure есть мощная абстракция, совмещающая черты итераторов и коллекций из Java. Эта абстракция называется *последовательностью* (seq). Она объединяет две сущности Java в одну. Это делается в расчете на достижение трех целей, таких как:

- получение более надежных итераторов, особенно для многопроходных алгоритмов;
- неизменяемость, благодаря которой последовательности можно передавать от функции к функции без всяких проблем;
- возможность применения ленивых последовательностей (подробнее о них ниже).

Несколько основных функций, применяемых при работе с последовательностями, приведены в табл. 10.4. Обратите внимание: ни одна из них не изменяет свои аргументы, получаемые в качестве ввода; если функция должна вернуть новое значение, то она возвращает уже новую последовательность.

Таблица 10.4. Основные функции для работы с последовательностями

Функция	Эффект
(seq <coll>)	Возвращает последовательность, представлющую собой «перспективу» коллекции, к которой эта последовательность применена
(first <coll>)	Возвращает первый элемент коллекции, сначала применяя к ней (seq), если это необходимо. Возвращает nil, если коллекция равна nil
(rest <coll>)	Возвращает новую последовательность, созданную из коллекции, минус первый элемент этой коллекции. Возвращает nil, если коллекция равна nil
(seq? <o>)	Возвращает true, если o — это последовательность (то есть если o реализует ISeq)
(cons <elt> <coll>)	Возвращает последовательность, составленную из коллекции, добавляя к ней новый элемент
(conj <coll> <elt>)	Возвращает новую коллекцию с новым элементом, добавленным с нужного конца, то есть в конце вектора или в начале списка
(every? <pred-fn> <coll>)	Возвращает true, если (pred-fn) возвращает логическое true для каждого элемента коллекции

Вот несколько примеров:

```
1:1 user=> (rest '(1 2 3))
(2 3)
1:2 user=> (first '(1 2 3))
1
1:3 user=> (rest [1 2 3])
(2 3)
1:13 user=> (seq ())
nil
1:14 user=> (seq [])
nil
```

```
1:15 user=> (cons 1 [2 3])
(1 2 3)
1:16 user=> (every? is-prime [2 3 5 7 11])
true
```

Здесь очень важно отметить, что у списков Clojure есть собственные последовательности, а у векторов — нет. Теоретически это могло бы означать, что вы не сможете применить `(rest)` к вектору. Но на практике такая возможность есть, поскольку `(rest)` действует, применяя к вектору `(seq)`, и лишь потом оперирует им. Это очень распространенное свойство последовательностей — многие их функции принимают более общий объект, чем последовательность, а потом применяют к такому объекту `(seq)` перед началом работы.

В этом разделе мы исследуем некоторые базовые свойства и примеры использования абстракции последовательностей. Особое внимание уделим ленивым последовательностям и функциям с переменным количеством аргументов. Первая из этих концепций, так называемая *ленивость*, — это прием программирования, нечасто применяемый в Java, поэтому, возможно, вы с ней пока не знакомы. С нее и начнем.

10.4.1. Ленивые последовательности

Ленивость — это мощная концепция, существующая во многих языках программирования. В сущности, ленивость позволяет отложить вычисление выражения до тех пор, пока это не потребуется. В Clojure это говорит о том, что мы можем не иметь полного списка всех значений, присутствующих в последовательности, а получать их по мере необходимости (например, вызывать функцию для генерирования значений по требованию).

В Java для воплощения такой идеи потребовалась бы какая-нибудь собственная реализация List, тем не менее подобный список можно было бы написать лишь с огромным количеством шаблонного кода. В Clojure присутствуют мощные макросы, специально предназначенные для упрощения создания ленивых последовательностей без малейших усилий.

Рассмотрим, как можно было бы представить ленивую, потенциально бесконечную последовательность. Очевидно, пришлось бы воспользоваться функцией, генерирующей элементы последовательности. Эта функция должна делать две вещи:

- возвращать следующий элемент последовательности;
- принимать фиксированное, конечное количество аргументов.

Математик сказал бы, что такая функция определяет рекуррентное соотношение, а теория таких отношений сразу же подсказывает, что их удобнее всего реализовывать по принципу рекурсии.

Допустим, у вас есть машина, в которой отсутствуют стековое пространство и другие ограничивающие факторы. Далее предположим, что вы можете создать два потока выполнения задач: один будет готовить бесконечную последовательность, другой — применять ее. Затем вы сможете воспользоваться рекурсией для

определения ленивой последовательности в генерирующем потоке. Для этого можно задействовать примерно такой фрагмент псевдокода:

```
(defn infinite-seq <vec-args>
(let [new-val (seq-fn <vec-args>)]
  (cons new-val (infinite-seq <new-vec-args>)))).
```

На практике в языке Clojure этот код не сработает, так как рекурсия `(infinite-seq)` «взорвет» стек. Но вам только потребуется добавить конструкцию, которая прикажет Clojure не переусердствовать с рекурсией, а продолжать работу лишь по мере необходимости — и эта задача решается.

Более того, вы можете справиться с такой задачей, имея и всего один рабочий поток, как показано в следующем примере. В листинге 10.5 определяется ленивая последовательность `k, k+1, k+2, ...` для некоторого числа `k`.

Листинг 10.5. Пример ленивой последовательности

```
(defn next-big-n [n] (let [new-val (+ 1 n)]
  (lazy-seq
    (cons new-val (next-big-n new-val))
)))
(defn natural-k [k]
  (concat [k] (next-big-n k)))

1:57 user=> (take 10 (natural-k 3))
(3 4 5 6 7 8 9 10 11 12)
```

Создается ленивая последовательность

Бесконечная рекурсия

concat ограничивает рекурсию

Здесь следует обратить внимание в первую очередь на форму `(lazy-seq)`, отмечающую точку, в которой должна начаться бесконечная рекурсия, и на форму `(concat)`, обеспечивающую здесь безопасность работы. Затем можно воспользоваться формой `(take)`, чтобы извлечь требуемое количество элементов из ленивой последовательности. В сущности, это количество определяется формой `(next-big-n)`.

Ленивые последовательности — исключительно мощная возможность. Научившись обращаться с ними, вы убедитесь, что это незаменимый инструмент в арсенале Clojure-разработчика.

10.4.2. Последовательности и функции с переменным количеством аргументов

У подхода Clojure к работе с функциями есть еще одна мощная черта, которую мы до сих пор подробно не обсудили. Речь пойдет о естественной способности функций принимать переменное количество аргументов. Иногда количество аргументов именуется *арностью* функции. Функция, принимающая переменное количество аргументов, называется *вариабельной*.

В качестве простейшего примера вновь обратимся к константной функции `(const-fun1)`, которая приводилась в листинге 10.1. Эта функция принимает единственный аргумент и пропускает его, не производя никаких операций и всегда

возвращая значение 1. Но что же произойдет, если передать функции (const-fun1) более одного аргумента?

```
1:32 user=> (const-fun1 2 3)
java.lang.IllegalArgumentException: Wrong number of args (2) passed to:
user$const-fun1 (repl-1:32)
```

Компилятор Clojure во время исполнения принудительно проверяет количество аргументов, передаваемых (const-fun1) (а также типы этих аргументов). При работе с функцией, которая всегда пропускает все свои аргументы и возвращает константное значение, такие проверки кажутся излишней перестраховкой. Как бы выглядела в Clojure функция, способная принимать любое количество аргументов?

В листинге 10.6 показано, как привести в такой вид константную функцию (const-fun1), рассмотренную выше в этой главе. Назовем новую версию (const-fun-arity1), так как это будет вариант const-fun1 с переменной *арностью*. Это наша самодельная версия функции (constantly), предлагаемой в стандартной библиотеке функций Clojure.

Листинг 10.6. Функция с переменным количеством аргументов

```
1:28 user=> (defn const-fun-arity1
  ([] 1)
  ([x] 1)
  ([x & more] 1)
)
#'user/const-fun-arity1
1:33 user=> (const-fun-arity1)
1
1:34 user=> (const-fun-arity1 2)
1
1:35 user=> (const-fun-arity1 2 3 4)
1
```

Несколько defn с различными сигнатурами

Основной момент, который следует отметить, заключается в том, что за функцией не следует вектор ее параметров, сопровождаемый формой, которая определяет поведение функции. Вместо этого мы видим здесь список пар, каждая из которых включает, во-первых, вектор параметров (фактически сигнатуру данной версии функции) и, во-вторых, реализацию данной версии функции. Эта концепция немного напоминает перегрузку методов в Java. Обычно принято определять несколько ситуативных форм (принимающих ноль, один или два параметра соответственно), а также дополнительную форму, имеющую в качестве последнего параметра последовательность. В листинге 10.6 это форма, имеющая вектор параметров [x & more]. Символ & означает, что перед нами версия функции, принимающая переменное количество аргументов.

Последовательности — исключительно мощная инновация в составе Clojure. На самом деле, чтобы научиться мыслить в стиле Clojure, важнее всего понять, как абстракция последовательности может применяться для решения конкретных прикладных задач.

Еще одна важная инновация Clojure — это интеграция между Clojure и Java, о которой мы поговорим в следующем разделе.

10.5. Взаимодействие между Clojure и Java

Язык Clojure с самого начала проектировался для работы на виртуальной машине Java, и он не пытается полностью скрыть характер JVM от программиста. Такие конкретные дизайнерские решения очевидны в нескольких случаях. Например, на уровне системы типов и списки и векторы Clojure реализуют `List` — стандартный интерфейс из библиотеки коллекций Java. Кроме того, мы с легкостью можем использовать в Clojure библиотеки Java и наоборот.

Эти свойства исключительно полезны, поскольку программистам Clojure открывается доступ к разнообразнейшим библиотекам и инструментарию Java. Кроме того, разработчик Clojure может опираться на поддержку производительности, действующую в JVM, и на другие возможности. В этом разделе мы обсудим несколько функций, связанных с таким взаимодействием, в частности:

- вызов Java из Clojure;
- способ, которым Java трактует тип функций Clojure;
- посредников Clojure;
- исследовательское программирование в среде REPL;
- вызов Clojure из Java.

Изучим такую интеграцию и для начала рассмотрим, как получать доступ к методам Java из Clojure.

10.5.1. Вызов Java из Clojure

Рассмотрим следующий фрагмент кода Clojure, интерпретируемый в среде REPL:

```
1:16 user=> (defn lenStr [y] (.length (.toString y)))
#'user/lenStr
1:17 user=> (schwartz ["bab" "aa" "dgfwg" "droopy"] lenStr)
("aa" "bab" "dgfwg" "droopy")
1:18 user=>
```

В этом фрагменте мы осуществили преобразование Шварца, чтобы отсортировать строки в массиве по их длине. Для этого мы воспользовались формами `(.toString)` и `(.length)`, представляющими собой методы Java. Они вызываются применительно к объектам Clojure. Точка в начале символа означает, что среда времени исполнения должна активизировать поименованный метод и применить его к следующему аргументу. На внутрисистемном уровне это делается с помощью макроса `(.)`.

Все значения Clojure определяются формой `(def)` или ее вариантом. Значения помещаются в экземпляры `clojure.lang.Var`, и такой экземпляр может содержать

любой объект `java.lang.Object`. Поэтому любой метод, который может быть вызван применительно к `java.lang.Object`, может быть применен и к значению Clojure. Некоторые другие формы для взаимодействия с миром Java таковы:

```
(System/getProperty "java.vm.version")
```

Эта форма используется для вызова статических методов, в данном случае — метода `System.getProperty()`. Форма:

```
Boolean/TRUE
```

используется для доступа к статическим общедоступным переменным (например, к константам). В двух последних примерах мы неявно задействовали применяемую в Clojure сущность пространства имен. *Пространства имен* (namespace) в Clojure подобны пакетам Java. В наиболее общих случаях, подобных показанному выше, применяются отображения с сокращенных форм Clojure на пакеты Java.

ПРИРОДА ВЫЗОВОВ CLOJURE

Вызов функции в Clojure — это истинный вызов метода на виртуальной машине Java. Виртуальная машина Java не гарантирует избавления от хвостовой рекурсии путем оптимизации, хотя такая оптимизация и достигается в некоторых диалектах Lisp (в частности, в реализациях Scheme). Некоторые другие диалекты Lisp, применяемые на JVM, при работе рассчитывают на истинную хвостовую рекурсию и поэтому готовы к тому, что вызов функции Lisp не будет абсолютно эквивалентен вызову метода JVM в любых обстоятельствах. Но Clojure целиком поддерживает JVM как платформу, даже ценой неполного соответствия традиционной практике работы с Lisp.

Если вы хотите создать новый экземпляр объекта Java, а потом манипулировать им в Clojure, то это можно с легкостью сделать с помощью формы `(new)`. У нее есть и альтернативная сокращенная форма: имя класса, за которым следует точка. Эта форма фактически является еще одним случаем использования макроса `(.)`:

```
(import '(java.util.concurrent CountDownLatch LinkedBlockingQueue))
(def cdl (new CountDownLatch 2))
(def lbq (LinkedBlockingQueue.))
```

Здесь мы также используем форму `(import)`, позволяющую импортировать несколько классов Java из одного пакета всего в одной строке.

Выше мы отмечали, что существует определенное соответствие между системами типов Clojure и Java. Рассмотрим его более подробно.

10.5.2. Тип Java у значений Clojure

Работая с REPL, удобно просмотреть, какие типы Java присущи некоторым значениям Clojure:

```
1:8 user=> (.getClass "foo")
java.lang.String
1:9 user=> (.getClass 2.3)
java.lang.Double
1:10 user=> (.getClass [1 2 3])
```

```
clojure.lang.PersistentVector
1:11 user=> (.getClass '(1 2 3))
clojure.lang.PersistentList
1:12 user=> (.getClass (fn [] "Hello world!"))
user$eval110$fn__111
```

Первым делом отметим, что все значения Clojure являются объектами; примитивные типы виртуальной машины Java по умолчанию не раскрываются (хотя существуют способы добраться до примитивных типов, что важно для тех, кого волнуют вопросы производительности). Как вы уже догадались, строковые и числовые значения непосредственно отображаются на соответствующие ссылочные типы Java (`java.lang.String`, `java.lang.Double` и т. д.).

Анонимная функция `"Hello world!"` имеет имя, указывающее, что это экземпляр динамически сгенерированного класса. Данный класс реализует интерфейс `clojure.lang.IFn`. Clojure использует его, чтобы указать, что значение является функцией. Данный интерфейс можно считать аналогом Clojure для интерфейса Java Callable из `java.util.concurrent`.

Последовательности реализуют интерфейс `clojure.lang.ISeq`. Он обычно является одним из конкретных подклассов абстрактного `ASeq` либо одиночной ленивой реализацией `LazySeq`.

Мы рассмотрели типы различных значений, а вот как хранятся эти значения? Как было отмечено в начале этой главы, `(def)` связывает символ со значением, в результате чего создается переменная. Переменные — это объекты типа `clojure.lang.Var` (реализующие несколько интерфейсов, в частности `IFn`).

10.5.3. Использование посредников Clojure

В Clojure есть мощный макрос, называемый `(proxy)`. Он позволяет создавать полноценный объект Clojure, строящийся на основе класса Java (либо реализующий интерфейс). Например, в листинге 10.7 мы возвращаемся к более раннему примеру (см. листинг 4.13), но основная часть самого выполнения теперь осуществляется в совсем кратком коде, что объясняется компактностью синтаксиса Clojure.

Листинг 10.7. Новый пример с планированием потоков-исполнителей

```
(import '(java.util.concurrent Executors LinkedBlockingQueue TimeUnit))
(def stpe (Executors/newScheduledThreadPool 2))
(def lbq (LinkedBlockingQueue.))

(def msgRdr (proxy [Runnable] []
  (run [] (.toString (.poll lbq)))
))

(def rdrHndl (.scheduleAtFixedRate stpe msgRdr 10 10 TimeUnit/MILLISECONDS))
```

Фабричный метод STPE

Определяем анонимную реализацию Runnable

Общая форма `(proxy)` такова:

```
(proxy [<superclass/interfaces>] [<args>] <impls of named functions>+)
```

Первый векторный аргумент содержит интерфейсы, которые должен реализовывать класс-посредник. Если посредник также должен строиться на основе класса Java (а он, конечно же, может служить расширением лишь одного класса Java), то имя этого класса должно быть первым элементом вектора.

Второй векторный аргумент включает в себя параметры, которые должны быть переданы конструктору суперкласса. Зачастую этот вектор является пустым, и он определенно будет пуст во всех случаях, когда форма `(proxy)` просто реализует интерфейсы Java.

После двух этих аргументов идут формы, представляющие реализации отдельных методов, требуемые указанными интерфейсами или суперклассами.

Форма `(proxy)` обеспечивает простую реализацию любого интерфейса Java. В результате возникает интересная возможность: можно использовать среду REPL из Clojure как полигон для экспериментов с кодом Java и JVM.

10.5.4. Исследовательское программирование в среде REPL

Суть исследовательского программирования заключается в том, что, во-первых, благодаря синтаксису Clojure мы можем писать меньше кода, а во-вторых, система REPL предоставляет «живую» интерактивную рабочую среду. По этим причинам REPL оказывается великолепным полигоном не только для исследования программирования на Clojure, но и для подробного изучения библиотек Java.

Вспомним, как в Java реализуются списки. В них есть метод `iterator()`, возвращающий объект типа `Iterator`. Но `Iterator` — это интерфейс, поэтому может возникнуть вопрос: а каков же реальный тип его реализации? Это легко узнать с помощью REPL:

```
1:41 user=> (import '(java.util ArrayList LinkedList))
java.util.LinkedList
1:42 user=> (.getClass (.iterator (ArrayList.)))
java.util.ArrayList$Itr
1:43 user=> (.getClass (.iterator (LinkedList.)))
java.util.LinkedList$ListItr
```

Форма `(import)` подключает к работе сразу два класса из пакета `java.util`. После этого вы сможете использовать из REPL метод Java `getClass()`, как мы делали это в подразделе 10.5.2. Как видите, на практике итераторы предоставляются внутренними классами. Наверное, это неудивительно; как мы говорили в разделе 10.4, итераторы тесно связаны с коллекциями, из которых они происходят. Поэтому им может понадобиться видеть внутренние детали реализации соответствующих коллекций.

Обратите внимание: в предыдущем примере мы не использовали ни одной конструкции Clojure — только немного синтаксиса. Все элементы, которыми мы манипулировали, были подлинными конструкциями Java. Но что, если мы пойдем другим путем и воспользуемся мощными абстракциями, которыми Clojure может обогатить программу Java? В следующем подразделе будет показано, как это делается.

10.5.5. Использование Clojure из Java

Как вы помните, система типов Clojure тесно связана с системой типов Java. Структуры данных из Clojure — практически те же коллекции Java, и они реализуют многие обязательные интерфейсы Java. Необязательные интерфейсы обычно не реализуются, поскольку зачастую предназначены для изменения структур данных, а такие изменения в Clojure не поддерживаются.

Корреляция между системами типов позволяет использовать структуры данных Clojure в программе, написанной на Java. Более того, к этому располагает сама организация Clojure — ведь это компилируемый язык с механизмом вызовов, соответствующим требованиям JVM. Таким образом, сводится к минимуму притирка во время исполнения, и с классом, полученным из Clojure, можно обращаться практически так же, как и с любым классом Java. Интерпретируемым языкам было бы гораздо сложнее взаимодействовать с Java, как правило, им требовалась бы хотя бы минимальная не Java-поддержка во время исполнения.

В следующем примере показано, как последовательность из Clojure может применяться к обычной строке Java. Чтобы запустить этот код, необходимо включить в путь к классу файл `clojure.jar`:

```
ISeq seq = StringSeq.create("foobar");
while (seq != null) {
  Object first = seq.first();
  System.out.println("Seq: "+ seq +" ; first: "+ first);
  seq = seq.next();
}
```

В этом фрагменте кода используется фабричный метод `create()` из класса `StringSeq`. Таким образом, последовательность символов, образующих строку, может рассматриваться как сущность-последовательность из Clojure. Методы `first()` и `next()` возвращают новые значения, а не изменяют имеющуюся последовательность — как раз об этом и шла речь в разделе 10.4.

До сих пор мы говорили только об однопоточном коде Clojure. В следующем разделе мы обсудим параллелизм в этом языке. В частности, поговорим о подходе Clojure к состояниям и изменяемости. Этот подход позволяет выстроить модель параллелизма, отличную от существующей в Java.

10.6. Параллелизм в Clojure

Модель состояния, действующая в Java, целиком завязана на идее изменяемых объектов. Как было продемонстрировано в главе 4, это прямо приводит к проблемам с безопасностью параллельного кода. Приходится задействовать достаточно сложные стратегии блокировки, чтобы не позволить другим потокам видеть обрабатываемые в данный момент (а значит, *несогласованные*) состояния объектов, над которыми трудится конкретный поток. Эти стратегии сложны в реализации, отладке и тестировании.

Абстракции Clojure для обеспечения параллелизма в некоторых отношениях не являются настолько низкоуровневыми, как в Java. Например, может показаться

странным использование пулов потоков, управляемых средой времени исполнения Clojure (сам разработчик практически не контролирует эти потоки). Но достоинство этой модели заключается в том, что платформа (в данном случае — среда времени исполнения Clojure) получает возможность тщательно вести за вас весь учет, и вы можете сосредоточиться на решении более важных задач, например на общем дизайне.

Clojure работает по принципу, в соответствии с которым потоки изначально по умолчанию изолируются друг от друга. Таким образом, в языке по определению не может быть нарушена безопасность типов при параллельной обработке. Исходя из предпосылки «ничего не требуется совместно использовать» и имея неизменяемые значения, Clojure избегает многих проблем, присущих Java, и может сосредоточиться на поиске способов безопасного разделения состояния для параллельного программирования.

ПРИМЕЧАНИЕ

Для обеспечения безопасности среда времени исполнения Clojure предоставляет механизмы для координирования работы потоков. Настоятельно рекомендуем вам опираться при работе на эти механизмы, а не пытаться использовать идиомы Java и выстраивать собственные параллельные конструкции.

На самом деле Clojure использует несколько различных методов для реализации ряда разновидностей моделей параллелизма. Это, во-первых, функции `future` и `pcall`, во-вторых — ссылки, в-третьих — агенты. Рассмотрим все три способа, начиная с наиболее простого.

10.6.1. Функции future и pcall

Первый и наиболее очевидный способ, позволяющий справиться с разделением состояния, — вообще его не разделять. На самом деле переменная (конструкция `var`) из Clojure, с которой мы до сих пор работали, практически непригодна для совместного использования. Если два разных потока наследуют одно и то же имя от `var` и заново привязывают его уже внутри потока, то такие операции повторной привязки видимы лишь в рамках конкретного потока и не могут при этом совместно использоваться другими потоками.

Можно запускать новые потоки, пользуясь тесной связью между Clojure и Java. Таким образом, вы можете без труда писать параллельный код Java на Clojure. Но некоторые подобные абстракции существуют в Clojure в «очищенной» форме. Например, Clojure обеспечивает очень чистый подход к концепции Future, об этом элементе Java мы говорили в главе 4. В листинге 10.8 приведен простой пример.

Листинг 10.8. Future в языке Clojure

```
user=> (def simple-future
  (future (do
    (println "Line 0")
    (Thread/sleep 10000)
    (println "Line 1")
    (Thread/sleep 10000)
    (println "Line 2"))))
```

```
#'user/simple-future                          Здесь сразу
Line 0                                        начинаем выполнение
user=> (future-done? simple-future)
user=> false
Line 1
user=> @simple-future                         Блокируется
Line 2                                        при разыменовании
nil
user=>
```

В этом листинге `Future` создается с помощью формы `(future)`. Как только `Future` создан, он начинает работать в фоновом потоке. Именно поэтому мы увидим вывод `Line 0` (а позже `Line 1`) в среде REPL Clojure — ведь код начинает действовать в другом потоке. Потом вы можете проверить, закончил ли выполняться код `(future-done?)`, — это неблокирующий вызов. Но если попытаться разыменовать `future`, то вызывающий поток блокируется до тех пор, пока функция не завершит работу.

Фактически здесь мы имеем дело с тонкой оберткой Clojure вокруг класса `Future` на Java, синтаксис при этом становится немного чище. Clojure также предоставляет полезные вспомогательные формы, которые могут очень пригодиться разработчику при организации параллельного программирования. Одна из простых подобных функций называется `(pcalls)`. Она принимает переменное количество безаргументных функций и выполняет их параллельно. Они применяются к управляемому во время исполнения пулу потоков и возвращают ленивую последовательность результатов. Если вы попытаетесь получить доступ к каким-либо элементам последовательности, которые еще не скомпилированы, то обращающийся к ним поток будет блокирован.

В листинге 10.9 создается одноаргументная функция под названием `(wait-with-for)`. Она использует одноаргументную форму, подобную той, что была показана в подразделе 10.3.2. После этого вы создаете несколько безаргументных функций `(wait-1)`, `(wait-2)` и т. д., которые можете передавать к `(pcalls)`.

Листинг 10.9. Параллельные вызовы в Clojure

```
user=> (defn wait-with-for [limit]
  (let [counter 1]
    (loop [ctr counter]
      (Thread/sleep 500)
      (println (str "Ctr=" ctr))
      (if (< ctr limit)
          (recur (inc ctr))
          ctr))))
#'user/wait-with-for
user=> (defn wait-1 [] (wait-with-for 1))
user=> #'user/wait-1
user=> (defn wait-2 [] (wait-with-for 2))
user=> #'user/wait-2
user=> (defn wait-3 [] (wait-with-for 3))
user=> #'user/wait-3
user=> (def wait-seq (pcalls wait-1 wait-2 wait-3))
#'user/wait-seq
Ctr=1
```

```
Ctr=1
Ctr=1
Ctr=2
Ctr=2
Ctr=3

user=> (first wait-seq)
1
user=> (first (next wait-seq))
2
```

Если поток проводит в спящем состоянии всего 500 мс, то функции ожидания завершаются очень быстро. Экспериментируя с задержкой (например, увеличив ее до 10 с), легко убедиться, что ленивая последовательность `wait-seq`, возвращаемая функциями `(pcalls)`, блокируется в соответствии с описанными выше принципами.

Такой доступ к простым многопоточным конструкциям очень удобен в тех случаях, когда не требуется разделять состояние. Но встречается и много таких ситуаций, в которых различные рабочие потоки должны взаимодействовать на лету. В Clojure предусматривается пара моделей для обработки таких случаев. Рассмотрим одну из них: разделение состояния с помощью формы `(ref)`.

10.6.2. Ссылки

Ссылки Clojure — один из механизмов, позволяющих разделять состояние между потоками. Они работают на базе модели, обеспечиваемой средой времени исполнения специально для таких изменений состояния, которые должны быть видимы для множества потоков. Подобная модель вводит дополнительный уровень косвенности между символом и значением. Соответственно, символ связывается со ссылкой, указывающей на значение, а не непосредственно со значением. В сущности, получается транзакционная система, координируемая средой времени исполнения Clojure. Такая система показана на рис. 10.6.

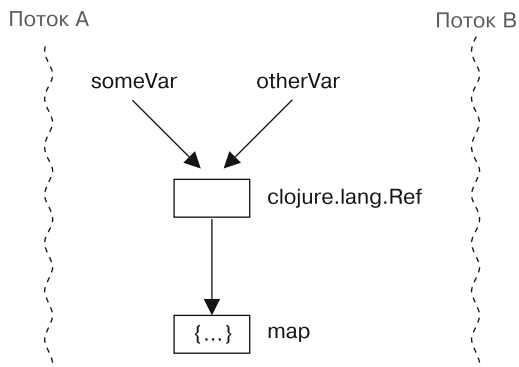


Рис. 10.6. Программная транзакционная память

Такая косвенность означает, что, прежде чем ссылку можно будет изменить или обновить, она должна быть помещена в транзакцию. Когда транзакция завершится,

вступят в силу все обновления либо не вступит ни одно из них. Такую операцию можно сравнить с транзакцией базы данных.

Описываемая ситуация может показаться немного абстрактной, поэтому рассмотрим пример, в котором моделируется работа банкомата. В Java нам бы потребовалось защищать все конфиденциальные фрагменты информации с помощью блокировок. В листинге 10.10 показан простой пример моделирования работы банкомата, в том числе моделирования блокировок.

Листинг 10.10. Моделирование банкомата на Java

```
public class Account {
  private double balance = 0;
  private final String name;
  private final Lock lock = new ReentrantLock();

  public Account(String name_, double initialBal_){
    name = name_;
    balance = initialBal_;
  }

  public synchronized double getBalance(){
    return balance;
  }

  public synchronized void debit(double debitAmt_) {
    balance -= debitAmt_;
  }

  public String getName() {
    return name;
  }

  public String toString() {
    return "Account [balance=" + balance + ", name=" + name + "]";
  }

  public Lock getLock() {
    return lock;
  }
}

public class Debitter implements Runnable {
  private final Account acc;
  private final CountDownLatch cdl;

  public Debitter(Account account_, CountDownLatch cdl_) {
    acc = account_;
    cdl = cdl_;
  }

  public void run() {
    double bal = acc.getBalance();
    Lock lk = acc.getLock();
```

```
    while (bal > 0) {
      try {
        Thread.sleep(1);
      } catch (InterruptedException e) { }
      lk.lock();
      bal = acc.getBalance();
      if (bal > 0) {
        acc.debit(1);
        bal--;
      }
      lk.unlock();
    }
    cdl.countDown();
  }
}

Account myAcc = new Account("Test Account", 500 * NUM_THREADS);
CountDownLatch stopl = new CountDownLatch(NUM_THREADS);

for (int i=0; i<NUM_THREADS; i++) {
  new Thread(new Debitter(myAcc, stopl)).start();
}
stopl.await();
System.out.println(myAcc);
```

Можно синхронизироваться на учетной записи

Необходимо повторно получить баланс

А теперь посмотрим, как можно переписать этот код на Clojure. Начнем с однопоточной версии. После этого мы сможем разработать параллельную версию и сравнить ее с однопоточным кодом. Тогда нам будет проще понять параллельный код.

В листинге 10.11 представлена простая однопоточная версия.

Листинг 10.11. Простая модель банкомата на языке Clojure

```
(defn make-new-acc [account-name opening-balance]
  {:name account-name :bal opening-balance})

(defn loop-and-debit [account]
  (loop [acc account]
    (let [balance (:bal acc) my-name (:name acc)]
      (Thread/sleep 1)
      (if (> balance 0)
        (recur (make-new-acc my-name (dec balance)))
        acc
      ))))

(loop-and-debit (make-new-acc "Ben" 5000))
```

loop/recur заменяет цикл while из Java

Полюбуйтесь, насколько компактен этот код по сравнению с версией, написанной на Java. Конечно же, он пока однопоточный, но количество кода все равно значительно сократилось по сравнению с тем, что нам потребовалось бы написать для Java. Запустив этот код, вы получите желаемый результат — словарь под названием `acc`, содержащий нулевой баланс. Теперь перейдем к параллельной форме.

Чтобы сделать этот код параллельным, в него нужно добавить ссылки Clojure. Они создаются с помощью формы `(ref)` и являются объектами виртуальной машины Java типа `clojure.lang.Ref`. Обычно они создаются со словарем Clojure, в котором содержится информация о состоянии. Вам также понадобится форма `(dosync)`, создающая транзакцию. Вместе с такой транзакцией вы сможете использовать форму `(alter)`, благодаря которой можно модифицировать ссылку. Функции, использующие ссылки для моделирования такой многопоточной модели банкомата, показаны в листинге 10.12.

Листинг 10.12. Многопоточная модель банкомата

```
(defn make-new-acc [account-name opening-balance]
  (ref {:name account-name :bal opening-balance}))

(defn alter-acc [acc new-name new-balance]
  (assoc acc :bal new-balance :name new-name))
(defn loop-and-debit [account]
  (loop [acc account]
    (let [balance (:bal @acc)
          my-name (:name @acc)]
      (Thread/sleep 1)
      (if (> balance 0)
        (recur (dosync (alter acc alter-acc my-name (dec balance)) acc))
        acc
      ))))

(def my-acc (make-new-acc "Ben" 5000))

(defn my-loop [] (let [the-acc my-acc]
  (loop-and-debit the-acc)
))

(pcalls my-loop my-loop my-loop my-loop my-loop)
```

Должно возвращаться значение, а не ссылка

Как было отмечено выше, функция `(alter-acc)` работает со значением и должна возвращать значение. Значение, на которое действует функция, — локальное, видимое данному потоку во время транзакции. Такое значение называется *внутритранзакционным*. Возвращаемое значение — это новое значение ссылки после возврата модифицирующей функции. Оно останется невидимым вне модифицирующего потока до тех пор, пока вы не выйдете из транзакционного блока, определенного в `(dosync)`.

Одновременно с этой могут осуществляться и другие транзакции. В таком случае программная транзакционная память Clojure будет отслеживать ход этих транзакций и допускать завершение транзакции лишь при условии ее согласованности с другими транзакциями, которые успели запуститься после нее. При несогласованности произойдет откат транзакции, после чего система может попытаться повторно ее выполнить с учетом обновленных данных.

Подобные повторные попытки могут вызывать проблемы, если при совершении транзакции возникают какие-либо побочные эффекты (например, создается файл логов или какой-нибудь другой вывод). Вам следует держать этапы транзакций как

можно более простыми и чистыми с точки зрения функционального программирования (то есть не допускать подобных побочных эффектов).

В некоторых многопоточных ситуациях такое «безоблачное» транзакционное поведение может требовать довольно тяжеловесной обработки. В отдельных многопоточных приложениях обмен информацией между потоками должен происходить лишь от случая к случаю, причем в довольно асимметричной манере. К счастью, в Clojure предусмотрен еще один, автономный механизм обеспечения параллелизма. Именно о нем и пойдет речь в следующем разделе.

10.6.3. Агенты

Агент — это применяемый в Clojure асинхронный примитив, ориентированный на работу с сообщениями и обеспечивающий параллельную обработку. Агент Clojure не имеет разделяемого состояния, а является фрагментом состояния, относящимся к другому потоку. Но агент может получать сообщения (в форме функций) от другого потока. На первый взгляд такая идея может показаться странной, но не слишком, если вспомнить об акторах Scala, рассмотренных в разделе 9.5.

«Придется отправлять их с посыльным, — думала она. — Вот будет смешно! Подарки собственным ногам! И адрес какой странный!»

Льюис Кэрролл, «Алиса в стране чудес»

Функции, применяемые к агенту, выполняются на потоке агента. Этот поток управляется средой времени исполнения Clojure и находится в пуле потоков, который обычно недоступен для программиста. Среда времени исполнения также гарантирует, что значения агента, видимые извне, будут изолированными и атомарными. Таким образом, пользовательский код будет видеть значение агента только до или после операции.

В листинге 10.13 приведен простой пример агента, подобный тому примеру, в котором мы обсуждали `future`-классы.

Листинг 10.13. Агенты Clojure

```
(defn wait-and-log [coll str-to-add]
  (do (Thread/sleep 10000)
    (let [my-coll (conj coll str-to-add)]
      (Thread/sleep 10000)
      (conj my-coll str-to-add))))

(def str-coll (agent []))

(send str-coll wait-and-log "foo")

@str-coll
```

Вызов `send` направляет к агенту вызов `(wait-and-log)`. Разыменовав его с помощью REPL, вы можете убедиться, что, как и предполагалось, вы никогда не увидите промежуточного состояния агента, а лишь окончательное состояние (где строка `"foo"` будет добавлена дважды).

На самом деле вызов (send) из листинга 10.13 чем-то напоминает адреса ног Алисы. Эти адреса можно с успехом написать на Clojure, так как Кэрролл формулирует адрес следующим образом:

```
Каминный Коврик

  (что возле Каминной Решетки)
    Госпоже Правой Ноге.
     С приветом от Алисы.
```

Такой адрес кажется нелепым, лишь поскольку вы считаете ноги неотъемлемой частью собственного тела. Не менее странным может показаться отправка сообщения к агенту, который назначен для потока в управляемом Clojure пуле потоков, при совместном использовании обоими одного и того же адресного пространства. Но одна из особенностей параллелизма, с которой мы сталкивались уже не раз, сводится к следующему: вполне можно пойти на усложнение кода, если в результате код станет более чист и удобен в использовании.

10.7. Резюме

Пожалуй, из всех языков, рассмотренных нами в этой части, Clojure наиболее сильно отличается от Java. Наследие, полученное от Lisp, неизменяемость и своеобразные подходы создают впечатление, что это совершенно не родственный Java язык. Но тесная интеграция между Clojure и виртуальной машиной Java, соответствие систем типов двух языков (даже при наличии альтернативных сущностей, например последовательностей), а также потенциал исследовательского программирования делают Clojure отличным дополнением Java.

Синергия двух языков особенно ярко проявляется в том, как Clojure делегирует многие низкоуровневые методы работы с потоками и управления параллелизмом в среде времени исполнения. Таким образом, программист освобождается от довольно кропотливой работы и может сосредоточиться на общем проектировании многопоточности и более высокоуровневых проблемах. Ситуацию можно сравнить с тем, как механизм сборки мусора в Java позволяет вам не задумываться о тонкостях управления памятью.

Различия между языками, изученными в этой части книги, красноречиво демонстрируют, как активно продолжает развиваться платформа Java и каким перспективным полем для разработки приложений она остается. Кроме того, здесь мы можем убедиться в гибкости и многофункциональности виртуальной машины Java.

В последней части книги мы продемонстрируем, как три рассмотренных выше новых языка обеспечивают новые подходы к практике разработки ПО. В следующей главе мы поговорим о разработке через тестирование — возможно, вы уже сталкивались с этой темой при программировании на Java. Но Groovy, Scala и Clojure позволяют взглянуть на эту проблему под совершенно новым углом. Надеемся, что эти языки помогут вам закрепить и расширить уже имеющиеся знания.

ЧАСТЬ 4

Создание многоязычного проекта

В последней части этой книги мы применим приобретенные выше знания о платформе и многоязычном программировании при работе с некоторыми наиболее распространенными и важными приемами современного программирования.

Профессиональный Java-разработчик должен не просто освоить виртуальную машину Java и языки, способные на ней работать. Чтобы успешно создавать проекты в области программирования, нужно также придерживаться важнейших практических рекомендаций, действующих в нашем деле. К счастью, довольно многие из этих практик зародились именно в экосистеме Java, поэтому мы найдем о чем поговорить.

Целую главу мы посвятим основам разработки через тестирование (TDD) и обсудим, как тестовые двойники применяются в ложных сценариях тестирования. Еще одна глава будет посвящена знакомству с организацией формального жизненного цикла сборки при сборочном процессе. В частности, мы поговорим о непрерывной интеграции. В двух этих главах мы познакомимся с некоторыми стандартными инструментами, например с JUnit для тестирования, Maven для организации жизненного цикла сборки и Jenkins для непрерывной интеграции.

Кроме того, мы поговорим о веб-разработке в эпоху Java 7, расскажем, как подобрать фреймворк, оптимально подходящий для реализации вашего проекта, и обсудим, как добиться высокой скорости разработки в конкретной среде.

Продолжая темы из части 3, мы расскажем, какую огромную роль альтернативные языки для JVM играют при разработке через тестирование, жизненном цикле сборки и быстрой веб-разработке. Будь то фреймворк ScalaTest для разработки через тестирование или фреймворки Grails и Compojure (из Groovy и Clojure соответственно) для создания веб-приложений — очевидно, что появление новых языков отразилось на самых разных областях экосистемы Java/JVM.

Мы расскажем, как эффекктивно задействовать сильные стороны этих новых языков для разработки знакомых проблем из области программирования. Вместе с основательной базой, которой служит вся среда Java и JVM, легко представить себе, какие широкие перспективы открываются перед разработчиком, в совершенстве владеющим многоязычным стилем.

В заключительной главе книги мы поговорим о будущем платформы и попытаемся спрогнозировать, что ждет ее впереди. Часть 4 — это рассказ о новых горизонтах, поэтому перевернем страницу и направимся к ним.

11 Разработка через тестирование

В этой главе:

- польза от применения разработки через тестирование (TDD);
- цикл «красный — зеленый — рефакторинг» — центральный компонент разработки через тестирование;
- краткое введение в JUnit — де-факто наиболее распространенный фреймворк для тестирования в Java;
- четыре типа тестовых двойников: пустой объект, поддельный объект, заглушка и подставной объект;
- тестирование на основе находящейся в памяти базы данных для кода DAO-объектов;
- имитация подсистем с помощью Mockito;
- использование ScalaTest — тестировочного фреймворка Scala.

Разработка через тестирование (test-driven development, TDD) уже довольно давно применяется при создании программного обеспечения. Базовая предпосылка разработки через тестирование заключается в том, что тесты пишутся до создания самого кода, обеспечивающего реализацию, а потом по мере необходимости выполняется рефакторинг реализации. Например, чтобы создать реализацию сцепления двух строковых объектов `String` (`"foo"` и `"bar"`), нужно сначала написать тест (результат выполнения которого будет равен `"foobar"`). Так вы будете гарантированно знать, что ваша реализация верна.

Многие фреймворки уже знакомы с тестировочным фреймворком JUnit и пользуются им более или менее регулярно. Но тесты обычно пишутся после реализации, а не до нее, из-за чего теряются некоторые основные преимущества разработки через тестирование.

Несмотря на кажущуюся вездесущность разработки через тестирование, многие программисты не понимают, зачем ею нужно заниматься. Специалист зачастую не может ответить на простые вопросы: «Зачем создавать код через тестирование? Какая от этого польза?»

Мы уверены, что *избавление от страха и неуверенности* — вот основной мотив для написания кода через тестирование. Кент Бек (Kent Beck) (один из авторов

тестировочного фреймворка JUnit) сумел очень красиво обосновать эту мысль в своей книге *Test-Driven Development: by Example*, издательство Addison-Wesley Professional, 2002; русское издание: *Бек Кент*, Экстремальное программирование. Разработка через тестирование. — СПб.: Питер, 2003:

- страх заставляет нас заблаговременно и тщательно обдумывать, к чему может привести то или иное действие;
- страх заставляет нас меньше общаться;
- страх заставляет нас пугаться результатов своей работы;
- страх делает нас раздражительными.

Разработка через тестирование избавляет нас от страха, и основательный Java-разработчик становится более уверенным, коммуникабельным, восприимчивым и удовлетворенным. Иными словами, разработка через тестирование помогает избавиться от стереотипного мышления, приводящего к следующим приемам:

- приступая к новому фрагменту работы, я не знаю, с чего начать, а поэтому просто начинаю импровизировать;
- изменяя имеющийся код, я не знаю, как этот код должен вести себя, поэтому в глубине души я очень боюсь что-то в нем нарушить.

Разработка через тестирование обладает и многими другими достоинствами, которые на первый взгляд неочевидны:

- *код становится чище* — ведь вы пишете только тот код, который вам нужен;
- *повышается качество дизайна* — некоторые разработчики расшифровывают аббревиатуру TDD как *test-driven design*;
- *повышается гибкость кода* — разработка через тестирование стимулирует вас программировать с расчетом на взаимодействие с интерфейсами;
- *вы быстро получаете отдачу* — то есть узнаете об ошибках *сразу*, а не потом.

Когда программист только начнет знакомиться с разработкой через тестирование, ему, возможно, придется избавиться от мысли, что эта технология «не для рядового специалиста». Может сложиться впечатление, что разработка через тестирование — это удел каких-то воображаемых адептов Agile или приверженцев еще каких-то эзотерических движений. Мы докажем, что такое восприятие проблемы совершенно ошибочно. Разработка через тестирование — это технология для всех и каждого.

Кроме того, гибкие разработки (Agile) и движение за мастерство программирования (software craftmanship) создавались именно для того, чтобы облегчить жизнь разработчику. Разумеется, никто и не пытался запретить коллегам задействовать разработку через тестирование или иные технологии.

Мы начнем эту главу с рассказа об основной идее, на которой базируется разработка через тестирование, — о цикле «красный — зеленый — рефакторинг». Потом мы познакомим вас с главным тестировочным двигателем Java — фреймворком JUnit и рассмотрим несколько простых примеров, иллюстрирующих описанные принципы.

МАНИФЕСТ ГИБКОЙ РАЗРАБОТКИ И ДВИЖЕНИЕ ЗА МАСТЕРСТВО ПРОГРАММИРОВАНИЯ

Движение гибкой разработки (http://agilemanifesto.org/iso/ru/) существует уже довольно давно и, несомненно, смогло изменить индустрию разработки ПО к лучшему. Многие великолепные технологии, в частности разработка через тестирование, были впервые созданы и поддерживаются в рамках этого движения. Мастерство программирования (software craftmanship) — более новое движение, приверженцы которого стремятся писать чистый код (http://manifesto.softwarecraftsmanship.org/).

Нам нравится подкалывать наших братьев по цеху, практикующих гибкую разработку и стремящихся достичь мастерства. Да что там, мы и сами (нередко) продвигаем эти методы. Но не будем отвлекаться от того, что может заинтересовать вас, читателей, желающих стать основательными Java-разработчиками. Разработка через тестирование — это техника програмирования, ни больше ни меньше.

Далее мы рассмотрим четыре основных типа воображаемых объектов, используемых в разработке через тестирование. Они очень важны, так как упрощают процесс отграничивания тестируемого кода от кода из сторонней библиотеки либо от поведения определенной подсистемы, например базы данных. По мере усложнения таких зависимостей вам требуются все более сложноорганизованные вспомогательные воображаемые объекты. Наконец, мы расскажем об имитации и специальной библиотеке Mockito — популярном имитационном инструменте, взаимодействующем с Java и помогающем изолировать тесты от каких-либо внешних воздействий.

Тестировочный фреймворк Java (мы поговорим о фреймворке JUnit) очень хорошо знаком разработчикам, и вы, вероятно, имеете определенный опыт написания тестов в такой среде. Но вы, возможно, не знаете, как тестируются новые языки, например Scala и Clojure. Чтобы вы научились грамотно тестировать программы на таких языках и применять разработку через тестирование при программировании на них, мы познакомим вас со ScalaTest — тестировочным фреймворком для Scala.

Приступим к введению в разработку через тестирование и начнем издалека.

11.1. Суть разработки через тестирование

Разработка через тестирование может применяться на многих уровнях. В табл. 11.1 показано четыре уровня тестирования, на которых обычно задействуется методология TDD.

Таблица 11.1. Уровни применения разработки через тестирование

Уровень	Описание	Пример
Компонент	Тестирование для проверки кода, содержащегося в классе	Проверка методов в классе BigDecimal
Интеграция	Тестирование для проверки взаимодействия между классами	Проверка класса Currency и его взаимодействия с классом BigDecimal

Продолжение ⇩

Таблица 11.1 (продолжение)

Уровень	Описание	Пример
Система	Тестирование для проверки работающей системы	Тестирование учетной системы из пользовательского интерфейса через класс Currency
Системная интеграция	Тестирование для проверки работающей системы, в том числе сторонних компонентов	Тестирование учетной системы, в том числе ее взаимодействия со сторонней системой, генерирующей отчеты

Разработку через тестирование удобнее всего использовать на уровне отдельных компонентов, и если вы пока не знакомы с этой методологией, отсюда и начнем. В этом разделе речь пойдет об использовании разработки через тестирование преимущественно на уровне компонентов. Ниже мы обсудим и другие уровни, а также поговорим о тестировании с учетом сторонних систем и подсистем.

СОВЕТ

Работа с имеющимся кодом, который не тестировался либо тестировался недостаточно хорошо, — порой изнурительная задача. Практически невозможно провести все необходимые тесты задним числом. Напротив, нужно писать тесты для каждого из последовательно создаваемых элементов функционала. Подробнее об этом — в отличной книге Майкла Физерса (Michael Feathers) Working Effectively with Legacy Code (издательство Prentice Hall, 2004).

Для начала вкратце рассмотрим один из краеугольных камней разработки через тестирование — цикл «красный — зеленый — рефакторинг». Воспользуемся фреймворком JUnit и протестируем код для расчета прибыли от продажи театральных билетов[1]. Если вы придерживаетесь этого принципа, то уже можно сказать, что вы практикуете разработку через тестирование! Затем мы рассмотрим некоторые философские идеи, лежащие в основе данного принципа, — так станет понятнее, почему следует использовать эту технику. Наконец, мы познакомим вас с JUnit — де-факто основным тестировочным фреймворком для разработки на Java. Мы опишем основные приемы работы с этой библиотекой.

Итак, изучим рабочий пример трехэтапной разработки через тестирование — цикла «красный — зеленый — рефакторинг». Как уже было сказано, возьмем за основу расчет прибыли от продажи театральных билетов.

11.1.1. Образец разработки через тестирование с одним случаем использования

Если вы — опытный специалист по разработке через тестирование, то, возможно, вам будет лучше пропустить этот маленький пример. Правда, здесь мы высказываем кое-какие идеи, которые могут оказаться для вас новыми. Допустим, перед вами поставлена задача написать безотказный метод, который будет рассчитывать

[1] Продажа театральных билетов — большой бизнес в Лондоне, где авторы живут и работают.

прибыль от продажи определенного количества театральных билетов. Исходные бизнес-правила, которыми должен руководствоваться бухгалтер театральной компании, довольно просты:

- исходная цена билета составляет $30;
- общая прибыль = количество проданных билетов, умноженное на их цену;
- в театральном зале могут разместиться 100 зрителей.

Поскольку в театре нет достаточно хорошей компьютерной программы для учета продаж, пользователю приходится вручную указывать количество проданных билетов.

Если вам приходилось заниматься разработкой через тестирование, то вы должны помнить три основных этапа, в которые она протекает: «красный — зеленый — рефакторинг». Если же вы пока не имеете опыта разработки через тестирование или хотите немного освежить эти знания, то обратимся к характеристикам этих этапов, предложенным Кентом Беком (пример взят из вышеупомянутой книги «Экстремальное программирование. Разработка через тестирование»):

- *красный* — пишем небольшой неработающий тест *(заведомо неуспешный тест)*;
- *зеленый* — пишем тест, который можно пройти максимально быстро *(проходной тест)*;
- *рефакторинг* — избавляемся от дублирования *(улучшенный проходной тест)*.

Чтобы вы получили представление о реализации `TicketRevenue`, которую мы стремимся написать, рассмотрим такой псевдокод:

```
estimateRevenue(int numberOfTicketsSold)
if (numberOfTicketsSold is less than 0 OR greater than 100)
then
  Deal with error and exit
else
  revenue = 30 * numberOfTicketsSold;
  return revenue;
endif
```

Постарайтесь не зацикливаться на этом коде. Процесс проектирования будет выстраиваться вслед за тестами, и реализация отчасти тоже.

ПРИМЕЧАНИЕ

В подразделе 11.1.2 мы рассмотрим способы того, как можно начать работу с заведомо неуспешным тестом, но для начала достаточно усвоить следующее: мы собираемся написать такой тест, который гарантированно не может быть пройден!

Напишем заведомо неуспешный тест компонента, воспользовавшись популярным фреймворком JUnit. Если вы не знакомы с JUnit, почитайте о нем в подразделе 11.1.4, а потом возвращайтесь сюда.

Написание заведомо неуспешного теста (красный)

Основной смысл этого этапа — начать работу с теста, пройти который гарантированно не удастся. На самом деле тест даже не скомпилируется, так как вы еще не написали класс `TicketRevenue`!

После краткой консультации с бухгалтером вы приходите к выводу, что нужно написать тесты для проверки пяти случаев: убыточная продажа билетов, 0, 1, 2–100 и >100.

СОВЕТ

При написании тестов (особенно подразумевающих работу с числами) удобно пользоваться следующим общим правилом: сосредоточьтесь на проработке случаев для нулевого результата, результата «один» и результата «много» (N). На следующем этапе можно продумать дополнительные ограничения для N, такие как отрицательная сумма или сумма, превышающая максимальный лимит.

Для начала напишем тест, охватывающий получение прибыли от продажи одного билета. Ваш тест в JUnit будет выглядеть примерно как код в листинге 11.1 (не забывайте, на этом этапе мы пока не пишем идеального, «проходного» теста).

Листинг 11.1. Заведомо неуспешный тест для `TicketRevenue`

```
import java.math.BigDecimal;
import static junit.framework.Assert.*;
import org.junit.Before;
import org.junit.Test;
public class TicketRevenueTest {

  private TicketRevenue venueRevenue;
  private BigDecimal expectedRevenue;

  @Before
  public void setUp() {
    venueRevenue = new TicketRevenue();
  }

  @Test
  public void oneTicketSoldIsThirtyInRevenue() {
    expectedRevenue = new BigDecimal("30");
    assertEquals(expectedRevenue, venueRevenue.estimateTotalRevenue(1));
  }
}
```

Случай: продан один билет

Как показывает тест, прибыль от продажи одного билета должна быть равна 30.

Но в таком виде тест не скомпилируется, так как вы еще не написали класс `TicketRevenue` с методом `estimateTotalRevenue(int numberOfTicketsSold)`. Чтобы избе-

жать ошибок компиляции и успешно провести тест, добавим произвольную реализацию, при наличии которой тест скомпилируется.

```
public class TicketRevenue {
  public BigDecimal estimateTotalRevenue(int i) {
    return BigDecimal.ZERO;
  }
}
```

Теперь, когда тест работает, можете запустить его в своей любимой интегрированной среде разработки. Каждая IDE по-своему запускает тесты JUnit, но, в принципе, в любой из них можно щелкнуть правой кнопкой мыши на тестовом классе и выбрать команду Run Test (Запустить тест). После этого IDE должна обновить окно или область окна, в которой вам будет выведена информация о том, что тест не пройден, так как ожидаемое значение `30` не возвращено в результате вызова `estimateTotalRevenue(1)`; вместо этого получено значение `0`.

Итак, у вас готов заведомо неуспешный тест. Далее переходим к зеленому этапу (обеспечиваем его прохождение).

Написание проходного теста (зеленый)

Основная цель данного этапа — обеспечить прохождение теста. Реализация может быть и неидеальной. Предоставив класс `TicketRevenue` с улучшенной реализацией `estimateTotalRevenue` (такой, которая не сводится к возвращению `0`), вы обеспечиваете прохождение теста (зеленый этап).

Не забывайте, на данном этапе нас интересует не совершенство кода как таковое, а успешное прохождение теста. Изначальный вариант решения может выглядеть примерно так, как показано в листинге 11.2.

Листинг 11.2. Первая версия `TicketRevenue`, проходящая тест

```
import java.math.BigDecimal;

public class TicketRevenue {

  public BigDecimal estimateTotalRevenue(int numberOfTicketsSold) {
    BigDecimal totalRevenue = BigDecimal.ZERO;
    if (numberOfTicketsSold == 1) {
      totalRevenue = new BigDecimal("30");
    }
    return totalRevenue;
  }
}
```

Реализация, проходящая тест

Если запустить тест теперь, он будет пройден, в большинстве IDE он будет отмечен зеленой полосой или флажком. На рис. 11.1 показано, как успешно пройденный тест обозначается в IDE Eclipse.

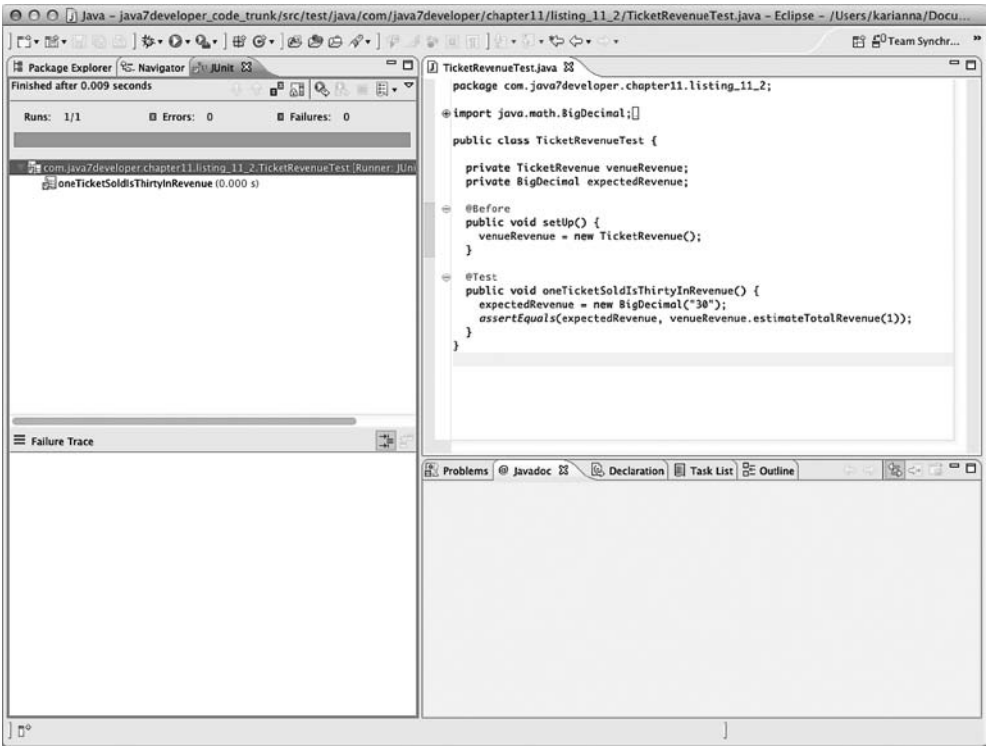


Рис. 11.1. Зеленая полоса (в книге она серая) обозначает в IDE Eclipse проходной тест

Возникает вопрос: можно ли после этого воскликнуть «Ура, готово!» и перейти к решению следующей задачи? Нет и еще раз нет. Нам, например, не терпится привести в порядок предыдущий листинг, так что этим и займемся.

Рефакторинг теста

Основной смысл данного этапа — рассмотреть написанную вами быструю реализацию, обеспечивающую прохождение теста, и убедиться, что вы следуете общепринятой практике. Разумеется, код из листинга 11.2 можно переписать так, чтобы он стал чище и аккуратнее. Иными словами, можно провести рефакторинг и уменьшить технический долг как для себя, так и для тех, кто будет работать над этим кодом после вас.

Технический долг — это метафора, предложенная Уордом Каннингемом (Ward Cunningham) и обозначающая дополнительные издержки (усилия), которые понадобятся впоследствии, чтобы привести в порядок быструю черновую реализацию решения, подготовленную вами сейчас.

Учтите, что проходной тест у вас готов, и вы можете смело *приступать к рефакторингу*. Вы не упустите из виду ту бизнес-логику, которую должен реализовывать код.

СОВЕТ

Еще одна польза, которую вы и ваша команда получаете от написания первичного проходного теста, — это общее ускорение всего процесса разработки. Все остальные члены команды сразу могут брать быструю черновую версию кода и тестировать ее применительно к более обширной базе кода (для интеграционных тестов и не только).

В этом примере мы не собираемся использовать магические числа — нам нужно гарантировать, что цена билета, равная 30, будет прямо указана в коде.

Листинг 11.3. Версия кода TicketRevenue, проходящая тест, после проведения рефакторинга

```
import java.math.BigDecimal;

public class TicketRevenue {

  private final static int TICKET_PRICE = 30;

  public BigDecimal estimateTotalRevenue(int numberOfTicketsSold) {
    BigDecimal totalRevenue = BigDecimal.ZERO;
    if (numberOfTicketsSold == 1) {
      totalRevenue =
        new BigDecimal(TICKET_PRICE *
                       numberOfTicketsSold);
    }
    return totalRevenue;
  }
}
```

Магических чисел нет

Цена после рефакторинга

В результате рефакторинга код улучшился, но этот вариант, разумеется, не охватывает всех возможных на практике вариантов (0, 2–100 и >100 проданных билетов). Чтобы не гадать, как должна выглядеть реализация для других возможных на практике случаев, нужно писать тесты и далее по ходу разработки. В следующем разделе мы подробнее изучим проектирование через тестирование и рассмотрим дополнительные практические случаи из нашего примера с продажей театральных билетов.

11.1.2. Образец разработки через тестирование с несколькими случаями использования

Если вы используете конкретный стиль разработки через тестирование, то продолжите добавлять по тесту на каждом этапе работы — для отрицательного результата, нулевого результата, для продажи 2–100 билетов или большего количества. Но есть и другой вполне допустимый подход — можно заранее написать набор родственных тестов, особенно если все они связаны с исходным тестом.

При этом важно продолжать следовать циклу «красный — зеленый — рефакторинг». Добавив эти случаи использования, можно получить следующий тестовый класс для заведомо неуспешного (красного) теста (листинг 11.4).

Листинг 11.4. Неуспешные тесты компонентов для TicketRevenue

```
import java.math.BigDecimal;
import static junit.framework.Assert.*;
import org.junit.Test;

public class TicketRevenueTest {

  private TicketRevenue venueRevenue;
  private BigDecimal expectedRevenue;

  @Before
  public void setUp() {
    venueRevenue = new TicketRevenue();
  }

  @Test(expected=IllegalArgumentException.class)
  public void failIfLessThanZeroTicketsAreSold() {
    venueRevenue.estimateTotalRevenue(-1);
  }

  @Test
  public void zeroSalesEqualsZeroRevenue() {
    assertEquals(BigDecimal.ZERO, venueRevenue.estimateTotalRevenue(0));
  }

  @Test
  public void oneTicketSoldIsThirtyInRevenue() {
    expectedRevenue = new BigDecimal("30");
    assertEquals(expectedRevenue, venueRevenue.estimateTotalRevenue(1));
  }

  @Test
  public void tenTicketsSoldIsThreeHundredInRevenue() {
    expectedRevenue = new BigDecimal("300");
    assertEquals(expectedRevenue, venueRevenue.estimateTotalRevenue(10));
  }

  @Test(expected=IllegalArgumentException.class)
  public void failIfMoreThanOneHundredTicketsAreSold() {
    venueRevenue.estimateTotalRevenue(101);
  }
}
```

Случай с отрицательной прибылью

Случай с нулевой прибылью

Прибыль от продажи одного билета

Прибыль от продажи N билетов

Прибыль от продажи 100 и более билетов

Первичная базовая (зеленая) реализация, позволяющая пройти все эти тесты, будет выглядеть примерно как в листинге 11.5.

Листинг 11.5. Первая версия TicketRevenue, позволяющая пройти все тесты

```
import java.math.BigDecimal;

public class TicketRevenue {

  public BigDecimal estimateTotalRevenue(int numberOfTicketsSold)
  throws IllegalArgumentException {

    BigDecimal totalRevenue = null;
    if (numberOfTicketsSold < 0) {
      throw new IllegalArgumentException("Must be > -1");
    }
    if (numberOfTicketsSold == 0) {
      totalRevenue = BigDecimal.ZERO;
    }
    if (numberOfTicketsSold == 1) {
      totalRevenue = new BigDecimal("30");
    }
    if (numberOfTicketsSold == 101) {
      throw new IllegalArgumentException("Must be < 101");
    }
    else {
      totalRevenue =
      new BigDecimal(30 * numberOfTicketsSold);
    }
    return totalRevenue;
  }
}
```

Исключительные случаи

Прибыль от продажи N билетов

С помощью такой реализации вы сможете пройти все написанные выше тесты.

Опять же, придерживаясь жизненного цикла разработки через тестирование, нужно выполнить рефакторинг этой реализации. Например, можно скомбинировать недопустимые случаи `numberOfTicketsSold (< 0 или > 100)` в одну инструкцию `if` и воспользоваться формулой `(TICKET_PRICE * numberOfTicketsSold)`, чтобы возвратить значения прибыли от всех других допустимых значений `numberOfTicketsSold`. У вас должен получиться код, похожий на приведенный в листинге 11.6.

Листинг 11.6. TicketRevenue после рефакторинга

```
import java.math.BigDecimal;

public class TicketRevenue {

  private final static int TICKET_PRICE = 30;

  public BigDecimal estimateTotalRevenue(int numberOfTicketsSold)
```

```
  throws IllegalArgumentException {

    if (numberOfTicketsSold < 0 || numberOfTicketsSold > 100) {
      throw new IllegalArgumentException
      ("# Tix sold must == 1..100");
    }

    return new BigDecimal
    (TICKET_PRICE * numberOfTicketsSold);
  }
}
```

Исключительный случай

Все остальные случаи

Итак, класс `TicketRevenue` стал более компактным, но он по-прежнему проходит все тесты! Теперь вы завершили полный цикл «красный — зеленый — рефакторинг» и можете спокойно переходить к следующему фрагменту бизнес-логики. Можно также заново провести весь цикл, если вы или ваш бухгалтер заметите какие-либо пограничные случаи, например необходимость обработки переменной цены билета.

Настоятельно рекомендуем вам усвоить, каковы фундаментальные причины использования цикла «красный — зеленый — рефакторинг» при разработке через тестирование — о них мы поговорим далее. Но если вам недосуг тратить время на теорию, можете перейти к подразделу 11.1.4 и подробнее почитать о фреймворке JUnit, либо к разделу 11.2, где рассказывается о тестовых двойниках и тестировании стороннего кода.

11.1.3. Дальнейшие размышления о цикле «красный — зеленый — рефакторинг»

Этот раздел построен на основании рабочего примера, здесь мы исследуем некоторые теоретические основы разработки через тестирование. Мы еще раз рассмотрим жизненный цикл «красный — зеленый — рефакторинг» — как вы помните, он начинается с написания заведомо неуспешного теста. Но эта задача решается несколькими способами.

Неуспешный тест (красный)

Некоторые разработчики предпочитают писать тест, не проходящий даже компиляции, и создают какой-либо действующий код реализации на зеленом этапе. Другие разработчики пишут как минимум заглушки методов, вызываемые тестом, поэтому тест удается скомпилировать, но он не проходится. Мы считаем, что оба стиля хороши. Выбирайте тот, который покажется вам наиболее удобным.

СОВЕТ

Такие тесты — первые примеры применения вашего кода, поэтому стоит внимательно их спроектировать — например, продумать, как будут выглядеть определения методов. В частности, стоит задаться вопросом, какие параметры вы будете передавать в эти методы. Какие возвращаемые значения вы рассчитываете получить? Кроме того, не забудьте написать тесты для важнейшей предметной области — методов equals() и hashCode().

Когда ваш заведомо неуспешный тест будет готов, перейдем к следующей стадии: обеспечим его прохождение.

Проходной тест (зеленый)

На этом этапе нужно попробовать написать минимальный код, необходимый для прохождения теста. Это не означает, что от вас требуется идеальная реализация! Реализация должна быть рабочей, а дорабатывать ее будем на этапе рефакторинга.

Как только проходной тест будет готов, можете сообщить другим членам вашей команды, что ваш код действительно делает то, что должен, и с ним можно начинать работать.

Рефакторинг

На этом этапе мы приступаем к рефакторингу кода нашей реализации. Просто не счесть категорий, на которые можно ориентироваться при рефакторинге. Существует несколько очевидных мер рефакторинга — например, удаление жестко запрограммированных переменных или разбиение одного метода на два. Если вы пишете объектно-ориентированный код, то старайтесь просто следовать принципам SOLID.

Аббревиатура SOLID была предложена Робертом Мартином (Robert Martin), известным в профессиональных кругах как «дядя Боб». Это пять принципов, кратко описанных в табл. 11.2. Подробнее о принципах SOLID рассказано в статье Мартина «Принципы объектно-ориентированного проектирования» (http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod).

Таблица 11.2. Принципы SOLID при написании объектно-ориентированного кода

Принцип	Описание
Принцип единственной обязанности (SRP)	На каждый объект должна быть возложена единственная обязанность
Принцип открытости/закрытости (OCP)	Программные сущности должны быть открыты для расширения, но закрыты для изменения
Принцип подстановки Барбары Лисков (LSP)	Объекты в программе могут быть заменены их подтипами
Принцип разделения интерфейса (ISP)	Много специализированных интерфейсов лучше, чем один универсальный
Принцип инверсии зависимостей (DIP)	Не допускаются зависимости от конкретной реализации (подробнее о внедрении зависимостей рассказано в главе 3)

СОВЕТ

Рекомендуем также использовать при работе статические анализаторы кода, в частности Checkstyle и FindBugs (подробнее о них — в главе 12). Вам также будет интересна книга Джошуа Блоха (Joshua Bloch) Effective Java, второе издание (издательство Addison-Wesley, 2008). В ней содержится множество полезной информации по работе с языком Java.

На этапе рефакторинга часто забывают о рефакторинге самого теста. Зачастую удается удалить из теста какой-нибудь общий настроечный и переналадочный код, переименовать тест, чтобы его имя более точно отражало его назначение, а также внести другие небольшие изменения, которые вам порекомендуют инструменты статического анализа.

Итак, вы овладели тремя этапами разработки через тестирование. Теперь познакомимся с JUnit — стандартным инструментом, без которого не обойтись при разработке кода Java через тестирование.

11.1.4. JUnit

JUnit де-факто является основным тестировочным фреймворком, применяемым при разработке проектов на Java. Существуют и другие подобные фреймворки, например TestNG, и у них тоже есть ярые приверженцы, но факт остается фактом: большая часть проектов на Java использует именно JUnit.

ПРИМЕЧАНИЕ

Если вы уже умеете работать с JUnit, можете сразу переходить к разделу 11.2.

JUnit предоставляет три следующие основные возможности:

- утверждения для тестирования программы на ожидаемые результаты и исключения, в частности `assertEquals()`;
- возможность настраивать и перенастраивать общие данные о тестировании, например `@Before` и `@After`;
- механизмы-исполнители для прогона наборов тестов.

Как видите, в JUnit задействуется простая модель, основанная на аннотациях, обеспечивающая большую часть важного функционала.

JUnit встроен в большинстве интегрированных сред разработки (например, Eclipse, IntelliJ и NetBeans), поэтому вам не придется скачивать, устанавливать и конфигурировать JUnit, если вы уже работаете с одной из этих IDE. Если ваша IDE в своей стандартной сборке не поддерживает JUnit, то можете посмотреть на сайте www.junit.org, как скачать и установить этот фреймворк[1].

ПРИМЕЧАНИЕ

При подготовке примеров для этой главы мы пользовались JUnit 4.8.2. Рекомендуем вам работать с этой же версией, если вы будете прорабатывать приведенные примеры.

Простейший тест в JUnit состоит из следующих элементов:

- метод с аннотацией `@Before` для настройки тестовых данных перед каждым прогоном теста;
- метод с аннотацией `@After` для повторного приведения тестовых данных в исходный вид после каждого прогона теста;
- сами тесты (помечаемые аннотацией `@Test`).

[1] В главе 12 мы расскажем об интеграции фреймворка JUnit со сборочным инструментом Maven.

Чтобы проиллюстрировать эти этапы на практике, рассмотрим пару простейших тестов в JUnit.

Предположим, мы помогаем команде OpenJDK писать тесты компонентов для проверки класса `BigDecimal`. Один тест нужен для проверки использования метода `add (1.5 + 1.5 == 3.0)`, а другой — для того чтобы убедиться, что программа выдает исключение `NumberFormatException` при попытке создать экземпляр класса `BigDecimal` с нечисловым значением.

ПРИМЕЧАНИЕ

Для примеров кода из этой главы мы зачастую писали по нескольку вариантов неуспешного теста (красный), реализации (зеленый) и рефакторинга. Это противоречит классической методологии разработки через тестирование, где в цикле «красный — зеленый — рефакторинг» должен использоваться лишь один тест, но так нам удалось подготовить больше примеров для текущей главы. Рекомендуем вам в ходе решения практических задач как можно чаще обходиться одним тестом.

Можете запустить код из листинга 11.7 в своей IDE, щелкнув правой кнопкой мыши на исходном коде и выбрав команду запуска, она же — команда тестирования (напоминаем, что во всех распространенных интегрированных средах разработки есть команда с самоочевидным названием, например Run Test (Запустить тест) или Run File (запустить файл)).

Листинг 11.7. Базовая структура теста в JUnit

```
import java.math.BigDecimal;
import org.junit.*;
import static org.junit.Assert.*;

public class BigDecimalTest {

  private BigDecimal x;

  @Before
  public void setUp() { x = new BigDecimal("1.5"); }

  @After
  public void tearDown() { x = null; }

  @Test
  public void addingTwoBigDecimals() {
    assertEquals(new BigDecimal("3.0"), x.add(x));
  }

  @Test(expected=NumberFormatException.class)
  public void numberFormatExceptionIfNotANumber() {
    x = new BigDecimal("Not a number");
  }
}
```

Стандартные импорты JUnit

1 Устанавливается перед тестом

2 Возврат тестовых параметров после прохождения теста

3 Выполнение теста

4 Обработка ожидаемого исключения

Перед запуском каждого теста `x` присваивается значение `BigDecimal("1.5")` в разделе `@Before` ❶. Так мы гарантируем, что каждый тест работает с известным значением `x`, а не с промежуточным значением `x`, изменившимся в результате одного из предыдущих тестов. После выполнения каждого теста необходимо убедиться, что значение `x` вновь установлено в `null` в разделе `@After` ❷ (это нужно, чтобы значение `x` можно было утилизировать при сборке мусора). После этого вы проверяете, правильно ли работает метод `BigDecimal.add()`, что делается с помощью метода `assertEquals()` ❸ (один из многих применяемых в JUnit методов `static assertX`). Чтобы обрабатывать ожидаемые исключения, вы добавляете к аннотации `@Test` опциональный параметр `expected` ❹.

Разработку через тестирование удобнее всего осваивать на практике. Хорошо запомнив принципы разработки через тестирование и понимая, как работает фреймворк JUnit, можете приступать к делу! Как показывают примеры, освоить разработку через тестирование на уровне отдельных компонентов совсем просто.

Но любой специалист, практикующий разработку через тестирование, рано или поздно сталкивается с ситуацией, когда требуется использовать зависимости или подсистему. В следующем разделе описаны приемы, помогающие эффективно тестировать такие ситуации.

11.2. Тестовые двойники

Поработав какое-то время в стиле разработки через тестирование, вы рано или поздно столкнетесь с ситуацией, в которой ваш код ссылается на какие-то (зачастую сторонние) зависимости или подсистему. В таком случае обычно требуется гарантировать, что тестируемый код изолирован от этой зависимости, поскольку вам нужно написать тест лишь для проверки вашей собственной реализации. Кроме того, вам требуется, чтобы тесты выполнялись максимально быстро. Для того чтобы задействовать стороннюю зависимость или подсистему, например базу данных, часто нужно много времени, и вы лишаетесь одного из значительных преимуществ разработки через тестирование — быстрого отклика. Эта проблема особенно актуальна при тестировании на уровне компонентов. Она решается с помощью *тестовых двойников*.

В этом разделе мы расскажем о том, как тестовые двойники помогают эффективно изолировать зависимости и подсистемы. Мы проработаем примеры, в которых используется четыре вида тестовых двойников (пустые объекты, заглушки, поддельные объекты и подставные объекты).

При самом сложном варианте работы, предполагающем тестирование с учетом внешних зависимостей (при обращении с распределенными или сетевыми службами), вам может пригодиться технология внедрения зависимостей (рассмотренная в главе 3) в комбинации с тестовыми двойниками. Такой прием действует и в системах, которые могут показаться безнадежно большими.

ПОЧЕМУ БЫ НЕ ВОСПОЛЬЗОВАТЬСЯ GUICE?

Если вы пока еще хорошо помните главу 3, то, вероятно, припоминаете и Guice — базовую реализацию фреймворка внедрения зависимостей для Java. Читая этот раздел, вы вполне можете задаться вопросом: «А почему бы не воспользоваться Guice?»

Очевидный ответ — мы не хотели излишне усложнять листинги с кодом, что неизбежно случилось бы даже при использовании самого простого фреймворка, например Guice. Не забывайте, что внедрение зависимостей — это технология. Чтобы ее применять, не обязательно задействовать целый фреймворк.

Нам очень нравится простое определение тестового двойника, предложенное Джерардом Месаросом (Gerard Meszaros) в книге *xUnit Test Patterns* (издательство Addison-Wesley, 2007), поэтому мы с удовольствием его здесь процитируем: «Тестовый двойник (фактически тест-дублер) — это общий термин для обозначения любого подставного объекта, используемого вместо реального объекта в целях тестирования».

Далее Месарос определяет четыре разновидности тестовых двойников, которые кратко охарактеризованы в табл. 11.3.

Таблица 11.3. Типы тестовых двойников

Тип	Описание
Пустой объект	Объект, который передается, но никогда не используется. Обычно применяется при заполнении списка параметров метода
Объект-заглушка	Объект, всегда возвращающий один и тот же заготовленный отклик. Может содержать какое-нибудь фиктивное состояние
Поддельный объект	Готовая рабочая реализация (по качеству или конфигурации может отличаться от промышленной), способная заменить промышленную
Подставной объект	Объект, отражающий несколько ожиданий и предоставляющий готовые ответы

Эти четыре типа тестовых двойников будет гораздо проще освоить, если проработать примеры кода, в которых они применяются. Этим и займемся, а начнем с пустого объекта.

11.2.1. Пустой объект

Пустой объект (пустышка) — наиболее простой в использовании тип тестовых двойников. Не забывайте: он помогает заполнять списки параметров или выполнять какие-либо обязательные требования, предъявляемые к полям, если вы наверняка знаете, что объект на данной позиции применяться не будет. Зачастую можно передавать и просто `null`-объект.

Вернемся к нашему примеру с театральными билетами. Конечно, хорошо знать оценку прибыли, получаемую от продаж в отдельно взятом киоске, но владельцы театра начали мыслить масштабнее. Нужна более качественная модель продаж

билетов и ожидаемой прибыли, в вашей команде программистов начинаются разговоры о том, что вот-вот будут предъявлены новые требования и вся система усложнится.

Теперь перед вами ставится задача отслеживать все продаваемые билеты и давать 10%-ную скидку на некоторые из них. По всей видимости, вам понадобится класс `Ticket`, предоставляющий метод для расчета скидочной цены. Мы начинаем уже знакомый цикл разработки через тестирование с написания заведомо неуспешного теста, особое внимание уделяем новому методу `getDiscountPrice()`. Кроме того, мы знаем, что у нас должно быть два конструктора — один для моделирования неуцененного билета, а другой — для создания экземпляров таких билетов, номинальная стоимость которых может варьироваться. В конечном счете объект `Ticket` должен принимать два аргумента:

- имя зрителя — строка `String`, которой мы вообще не будем касаться в этом тесте;
- обычная цена — класс `BigDecimal`, который будет использоваться в этом тесте.

Мы можем быть совершенно уверены, что имя клиента не будет играть никакой роли в методе `getDiscountPrice()`. Это означает, что мы можем передать конструктору пустой объект (в данном случае — пустую строку «Riley», как показано в листинге 11.8).

Листинг 11.8. Реализация TicketTest с применением пустого объекта

```
import org.junit.Test;
import java.math.BigDecimal;
import static org.junit.Assert.*;

public class TicketTest {

  @Test
  public void tenPercentDiscount() {
    String dummyName = "Riley";
    Ticket ticket = new Ticket(dummyName,
                               new
    BigDecimal("10"));
    assertEquals(new BigDecimal("9.0"), ticket.getDiscountPrice());
  }
}
```

Создаем пустой объект

Передаем пустой объект

Как видите, концепция пустого объекта проста.

Чтобы полностью представлять себе суть такого объекта, рассмотрим следующий код, в котором присутствует частичная реализация класса `Ticket` (листинг 11.9).

Листинг 11.9. Класс Ticket, тестируемый с применением пустого объекта

```
import java.math.BigDecimal;

public class Ticket {
  public static final int BASIC_TICKET_PRICE = 30;
```

Стандартная цена

```
  private static final BigDecimal DISCOUNT_RATE =
                       new BigDecimal("0.9");
  private final BigDecimal price;
  private final String clientName;

  public Ticket(String clientName) {
    this.clientName = clientName;
    price = new BigDecimal(BASIC_TICKET_PRICE);
  }

  public Ticket(String clientName, BigDecimal price) {
    this.clientName = clientName;
    this.price = price;
  }

  public BigDecimal getPrice() {
    return price;
  }

  public BigDecimal getDiscountPrice() {
    return price.multiply(DISCOUNT_RATE);
  }
}
```

Стандартная скидка

Отдельные программисты путаются при работе с пустыми объектами — они ищут в них сложность, которой на самом деле там нет. Пустые объекты действительно абсолютно бесхитростны — пустым можно считать любой объект, который раньше применялся во избежание исключений `NullPointerException` и ради обеспечения работоспособности кода.

Перейдем к следующему типу тестового двойника. Вторым по сложности является объект-заглушка.

11.2.2. Объект-заглушка

Как правило, *объект-заглушка* применяется в тех случаях, когда нужно заменить реальную реализацию таким объектом, который всякий раз возвращает один и тот же ответ. Вернемся к нашему примеру с расценками театральных билетов, чтобы познакомиться с подобными объектами на практике.

Итак, вы возвращаетесь после выходных с чувством выполненного долга к работе над реализованным на прошлой неделе классом `Ticket`. Открываете рабочую почту — и сразу натыкаетесь на отчет об ошибке, согласно которому ваш тест `tenPercentDiscount()` из листинга 11.8 периодически не удается пройти. Вы заглядываете в базу кода и обнаруживаете, что метод `tenPercentDiscount()` изменен. Теперь экземпляр `Ticket` создается с применением конкретного класса `HttpPrice`, который реализует недавно написанный интерфейс `Price`.

Вы начинаете разбираться в проблеме внимательнее и обнаруживаете, что метод `getInitialPrice()` из класса `HttpPrice` вызывается для получения исходной цены с внешнего сайта, и это делается с помощью стороннего класса `HttpPricingService`.

Соответственно, при вызове `getInitialPrice()` всякий раз может возвращаться иная цена, и по ряду причин тест время от времени может оказываться неуспешным. Иногда изменяются правила, регулирующие работу корпоративного брандмауэра, а порой сторонний сайт просто оказывается недоступен.

В результате ваш тест иногда оказывается неуспешным, а его цели смазываются. Не забывайте: все, что требовалось от данного теста компонента, — это расчет 10%-ной скидки!

ПРИМЕЧАНИЕ

Обращения к стороннему сайту для расчета цены — явно чрезмерное требование для такого теста. Но, возможно, вам не помешает организовать отдельные тесты для проверки интеграции системы, которые будут охватывать класс HttpPrice и сторонний класс HttpPricingService.

Прежде чем заменить класс `HttpPrice` заглушкой, посмотрим, как у нас сейчас выглядит весь код. Он представлен в листингах 11.10–11.12. Владельцы театра не только хотят внести изменения, связанные с интерфейсом `Price`, но и собираются отказаться от регистрации имен покупателей билетов. Соответствующие изменения показаны в листинге 11.10.

Листинг 11.10. Реализация `TicketTest` с новыми требованиями

```
import org.junit.Test;
import java.math.BigDecimal;
import static org.junit.Assert.*;

public class TicketTest {

  @Test
  public void tenPercentDiscount() {
    Price price = new HttpPrice();
    Ticket ticket = new Ticket(price);
    assertEquals(new BigDecimal("9.0"),
    ticket.getDiscountPrice());
  }
}
```

HttpPrice реализует интерфейс Price

Создаем Ticket (билет)

Здесь тест может провалиться

В листинге 11.11 продемонстрирована новая реализация `Ticket`, которая теперь включает закрытый класс `FixedPrice`. Он нужен для обработки простого случая, в котором цена является известной и фиксированной, а не выводится из какого-то внешнего источника.

Листинг 11.11. Реализация `Ticket` с новыми требованиями

```
import java.math.BigDecimal;

public class Ticket {
  public static final int BASIC_TICKET_PRICE = 30;
  private final Price priceSource;
```

```
  private BigDecimal faceValue = null;
  private final BigDecimal discountRate;

  private final class FixedPrice implements Price {
    public BigDecimal getInitialPrice() {
      return new BigDecimal(BASIC_TICKET_PRICE);
    }
  }
  public Ticket() {
    priceSource = new FixedPrice();
    discountRate = new BigDecimal("1.0");
  }

  public Ticket(Price price) {
    priceSource = price;
    discountRate = new BigDecimal("1.0");
  }

  public Ticket(Price price,
                BigDecimal specialDiscountRate) {
    priceSource = price;
    discountRate = specialDiscountRate;
  }

  public BigDecimal getDiscountPrice() {
    if (faceValue == null) {
      faceValue = priceSource.getInitialPrice();
    }
    return faceValue.multiply(discountRate);
  }
}
```

Измененный конструктор

Новый вызов getInitialPrice

Неизменное вычисление

Если мы попытаемся исследовать полную реализацию класса `HttpPrice`, то слишком отвлечемся от темы. Просто предположим, что он обращается к другому классу, `HttpPricingService` (см. листинг 11.12).

Листинг 11.12. Интерфейс Price и реализация HttpPrice

```
import java.math.BigDecimal;

public interface Price {
  BigDecimal getInitialPrice();
}

public class HttpPrice implements Price {
  @Override
  public BigDecimal getInitialPrice() {
    return HttpPricingService.getInitialPrice();
  }
}
```

Возвращает случайные результаты

Итак, как же нам предоставить результат, эквивалентный получаемому от HttpPricingService? Здесь нужно внимательно продумать, что именно вы собираетесь протестировать. В данном случае мы хотим убедиться, что умножение метода getDiscountPrice() из класса Ticket действует именно так, как и ожидалось.

Таким образом, можно заменить класс HttpPrice реализацией-заглушкой StubPrice, которая гарантированно будет возвращать согласованную цену в ответ на вызов getInitialPrice(). Так мы изолируем тест от несогласованного и периодически не проходящего тест класса HttpPrice. Прохождение теста с данной реализацией обеспечивается в листинге 11.13.

Листинг 11.13. Реализация TicketTest с применением объекта-заглушки

```
import org.junit.Test;
import java.math.BigDecimal;
import static org.junit.Assert.*;

public class TicketTest {

  @Test
  public void tenPercentDiscount() {
    Price price = new StubPrice();
    Ticket ticket = new Ticket(price);
    assertEquals(9.0,
                 ticket.getDiscountPrice().doubleValue(),
                 0.0001);
  }
}
```

Заглушка StubPrice

Создаем Ticket (билет)

Проверяем цену

StubPrice — это простой маленький класс, исправно возвращающий исходную цену, равную 10 (листинг 11.14).

Листинг 11.14. Фиктивная реализация StubPrice

```
import java.math.BigDecimal;

public class StubPrice implements Price {

  @Override
  public BigDecimal getInitialPrice() {
    return new BigDecimal("10");
  }
}
```

Возвращаем согласованную цену

Уф! Тест снова проходится без проблем, и, что не менее важно, вы можете смело приступать к рефакторингу всей оставшейся части реализации.

Заглушки — очень удобная разновидность тестового двойника, но иногда может потребоваться, чтобы заглушка выполняла какую-нибудь реальную работу и вела себя как можно более похоже на окончательно доработанную систему. Для этого используется еще одна разновидность тестового двойника — поддельный объект.

11.2.3. Поддельный объект

Поддельный объект можно считать улучшенной заглушкой, которая работает практически так же, как и ваш готовый код, но кое-где упрощает функционал, чтобы соответствовать тестовым требованиям. Поддельные объекты особенно удобны в тех случаях, когда вы хотите протестировать код по отношению к какой-то программе, очень напоминающей реальную стороннюю подсистему или зависимость, которую придется задействовать в реальном продукте.

Как правило, основательному Java-разработчику рано или поздно приходится писать код, взаимодействующий с базой данных. Обычно такой код совершает с объектами набор операций CRUD (создание, чтение, обновление, удаление). Проверка работоспособности вашего кода, образующего объект доступа к данным, чаще всего откладывается до этапа интеграционного теста всей системы, а иногда такой проверкой вообще пренебрегают! Было бы очень удобно проверять работоспособность такого кода во время тестирования компонента или интеграционного теста. Так вы получите важнейший, а главное — быстрый отклик системы.

Именно в таком случае можно воспользоваться поддельным объектом — он будет играть роль базы данных, с которой предстоит взаимодействовать. Но написать собственный поддельный объект, выступающий в роли базы данных, будет очень непросто! К счастью, в последние годы базы данных, хранимые в оперативной памяти, стали достаточно компактны, легковесны и многофункциональны, чтобы их можно было использовать в качестве поддельного объекта. HSQLDB (www.hsqldb.org) — популярная база данных, которая функционирует в оперативной памяти и применяется именно для таких целей.

Разработка приложения для работы с театральными билетами идет своим чередом, и на следующем этапе от вас требуется организовать хранение информации о билетах в базе данных, откуда она может быть извлечена позднее. Наиболее распространенный фреймворк Java, применяемый для обеспечения персистентности баз данных, называется Hibernate (www.hibernate.org).

HIBERNATE И HSQLDB

Если вы не знакомы с Hibernate или HSQLDB — не волнуйтесь! Hibernate — это фреймворк для объектно-реляционного отображения (ORM), реализующий стандарт API Java для персистентного хранения данных (JPA). В сущности, он позволяет вызывать простые методы Java `save`, `load`, `update` и многие другие для выполнения операций CRUD. Этим он отличается от применения неадаптированных SQL и JDBC, так как позволяет абстрагироваться от синтаксиса и семантики, специфичных для баз данных.

HSQLDB — это просто база данных Java, хранимая в оперативной памяти. Чтобы начать работать с ней, нужно просто добавить файл `hsqldb.jar` в `CLASSPATH`. Эта система в функциональном отношении сильно напоминает обычную RDBMS, но если вы ее отключите, то потеряете находившиеся в ней данные (правда, с потерями данных можно справляться — о том, как это делается, подробно рассказано на сайте HSQLDB).

Возможно, мы только что упомянули две совершенно новые для вас технологии. Но в исходном коде, сопровождающем эту книгу, вы найдете сборочные сценарии, которые помогут вам создать корректные зависимости JAR и правильные конфигурационные файлы в нужном месте.

Для начала вам понадобится конфигурационный файл Hibernate, в котором вы определите соединение с базой данных HSQLDB (листинг 11.15).

Листинг 11.15. Конфигурация Hibernate для работы с HSQLDB

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-configuration PUBLIC
"-//Hibernate/Hibernate Configuration DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">

<hibernate-configuration>
  <session-factory>
    <property name="hibernate.dialect">
      org.hibernate.dialect.HSQLDialect
    </property>
    <property name="hibernate.connection.driver_class">
      org.hsqldb.jdbcDriver
    </property>
    <property name="hibernate.connection.url">
      jdbc:hsqldb:mem:wgjd
    </property>
    <property name="hibernate.connection.username">sa</property>
    <property name="hibernate.connection.password"></property>
    <property name="hibernate.connection.autocommit">true</property>
    <property name="hibernate.hbm2ddl.auto">
      Create
    </property>
    <property name="hibernate.show_sql">true</property>
    <mapping resource="Ticket.hbm.xml"/>
  </session-factory>
</hibernate-configuration>
```

Как видите, в последней строке листинга содержится ссылка на отображение класса `Ticket` в качестве ресурса (`<mapping resource="Ticket.hbm.xml"/>`) (1). Эта строка подсказывает Hibernate, как отобразить файлы Java на столбцы базы данных. Вместе с информацией о диалекте, предоставляемой в конфигурационном файле Hibernate (в данном случае HSQLDB), это фактически все данные, требуемые Hibernate для автоматического создания SQL в фоновом режиме.

Хотя Hibernate и позволяет добавлять информацию об отображении непосредственно к классу Java в виде аннотаций, мы предпочитаем отображение в XML-стиле, как показано в листинге 11.16.

ВНИМАНИЕ

В профессиональных рассылках разворачивается множество жарких споров о том, какой стиль лучше — с применением аннотаций или с помощью XML. Рекомендуем выбрать такой стиль, который вам кажется наиболее удобным, и придерживаться его.

Листинг 11.16. Отображение Ticket в Hibernate

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-mapping PUBLIC
```

```
"-//Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">

<hibernate-mapping>

  <class
    name="com.java7developer.chapter11 .listing_11_18.Ticket">

  <id name="ticketId"
      type="long"
      column="ID" />

  <property name="faceValue"
            type="java.math.BigDecimal"
            column="FACE_VALUE"
            not-null="false" />

  <property name="discountRate"
            type="java.math.BigDecimal"
            column="DISCOUNT_RATE"
            not-null="true" />

  </class>
</hibernate-mapping>
```

Указываем класс для отображения

Указываем ticketID как открытый ключ

Отображаем faceValue

Отображаем discountRate

Теперь, когда у нас готовы конфигурационные файлы, подумаем, что нужно протестировать. От нас требуется обеспечить возможность получать `Ticket` по уникальному ID. Чтобы поддерживать и эту возможность, и отображение Hibernate, нам потребуется изменить класс `Ticket` следующим образом (листинг 11.17).

Листинг 11.17. `Ticket` с ID

```
import java.math.BigDecimal;

public class Ticket {

  public static final int BASIC_TICKET_PRICE = 30;
  private long ticketId;
  private final Price priceSource;
  private BigDecimal faceValue = null;
  private BigDecimal discountRate;

  private final class FixedPrice implements Price {
    public BigDecimal getInitialPrice() {
      return new BigDecimal(BASIC_TICKET_PRICE);
    }
  }

  public Ticket(long id) {
    ticketId = id;
```

Добавление идентификатора

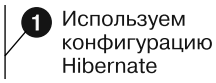
```
    priceSource = new FixedPrice();
    discountRate = new BigDecimal("1.0");
  }

  public void setTicketId(long ticketId) {
    this.ticketId = ticketId;
  }

  public long getTicketId() {
    return ticketId;
  }

  public void setFaceValue(BigDecimal faceValue) {
    this.faceValue = faceValue;
  }

  public BigDecimal getFaceValue() {
    return faceValue;
  }

  public void setDiscountRate(BigDecimal discountRate) {
    this.discountRate = discountRate;
  }

  public BigDecimal getDiscountRate() {
    return discountRate;
  }

  public BigDecimal getDiscountPrice() {
    if (faceValue == null) faceValue = priceSource.getInitialPrice();
    return faceValue.multiply(discountRate);
  }
}
```

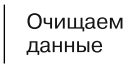
Итак, теперь у нас есть и отображение `Ticket`, и измененный класс `Ticket`, поэтому можно провести тест, который будет активизировать метод `findTicketById` применительно к классу `TicketHibernateDao`. Нужно добавить еще один вспомогательный тест `JUnit`, как показано в листинге 11.18.

Листинг 11.18. Тестовый класс TicketHibernateDaoTest

```
import java.math.BigDecimal;
import org.hibernate.cfg.Configuration;
import org.hibernate.SessionFactory;
import org.junit.*;
import static org.junit.Assert.*;
public class TicketHibernateDaoTest {

  private static SessionFactory factory;
  private static TicketHibernateDao ticketDao;
  private Ticket ticket;
```

```
  private Ticket ticket2;

  @BeforeClass
  public static void baseSetUp() {
    factory =
    new Configuration().
        configure().buildSessionFactory();
    ticketDao = new TicketHibernateDao(factory);
  }

  @Before
  public void setUpTest()
  {
    ticket = new Ticket(1);
    ticketDao.save(ticket);
    ticket2 = new Ticket(2);
    ticketDao.save(ticket2);
  }

  @Test
  public void findTicketByIdHappyPath() throws Exception {
    Ticket ticket = ticketDao.findTicketById(1);
    assertEquals(new BigDecimal("30.0"),
                 ticket.getDiscountPrice());
  }

  @After
  public static void tearDown() {
    ticketDao.delete(ticket);
    ticketDao.delete(ticket2);
  }

  @AfterClass
  public static void baseTearDown() {
    factory.close();
  }
}
```





До запуска каких-либо тестов мы используем конфигурацию Hibernate для создания объекта доступа к данным, который хотим протестировать ❶. Начиная с этого этапа перед запуском каждого теста мы сохраняем пару билетов в базе данных HSQLDB (в качестве тестовых данных) ❷. Тест запускается, и в ходе него проверяется метод `findTicketById`, относящийся к объекту доступа к данным ❸.

Поначалу тест проходить не будет, ведь мы еще не написали класс `TicketHibernateDao` и относящиеся к нему методы. Если мы работаем с фреймворком Hibernate, то не нуждаемся в SQL и можем не учитывать тот факт, что при тестировании задействуется база данных HSQLDB. Соответственно, реализация объекта доступа к данным может выглядеть так, как показано в листинге 11.19.

Листинг 11.19. Класс TicketHibernateDao

```
import java.util.List;
import org.hibernate.Criteria;
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.criterion.Restrictions;

public class TicketHibernateDao {
  private static SessionFactory factory;
  private static Session session;

  public TicketHibernateDao(SessionFactory factory)
  {
    TicketHibernateDao.factory = factory;
    TicketHibernateDao.session = getSession();
  }

  public void save(Ticket ticket)
  {
    session.save(ticket);
    session.flush();
  }

  public Ticket findTicketById(long ticketId)
  {
    Criteria criteria =
      session.createCriteria(Ticket.class);
    criteria.add(Restrictions.eq("ticketId", ticketId));
    List<Ticket> tickets = criteria.list();
    return tickets.get(0);
  }

  public void delete(Ticket ticket) {
    session.delete(ticket);
    session.flush();
  }

  private static synchronized Session getSession() {
    return factory.openSession();
  }
}
```

Задаем фабрику и сессию

1 Сохраняем Ticket

2 Находим Ticket по ID

Метод save, относящийся к объекту доступа к данным, элементарен. Он просто активизирует метод save из фреймворка Hibernate, после чего следует сброс данных, гарантирующий, что объект сохраняется в базе данных HSQLDB 1. Чтобы получить Ticket, мы задействуем функцию Hibernate Criteria (она аналогична созданию инструкции WHERE на языке SQL) 2.

Итак, объект доступа к данным теперь готов и тест выполняется успешно. Вы, наверное, заметили, что и метод `save` частично протестирован. Теперь можно продолжить работу и написать более скрупулезные тесты — например, чтобы проверить, возвращаются ли билеты из базы данных с правильным значением `discountRate`. Теперь вы можете проверять ваш код доступа к базе данных на гораздо более раннем этапе цикла тестирования — и все достоинства разработки через тестирование будут ощутимы и на уровне доступа к данным.

Поговорим о следующем типе тестового двойника — подставном объекте.

11.2.4. Подставной объект

Подставные объекты похожи на объекты-заглушки, рассмотренные выше, правда, заглушки — гораздо более незатейливые штуки. Например, заглушки обычно просто имитируют методы, и при вызове такие методы неизменно дают один и тот же результат. С помощью заглушек никак нельзя смоделировать поведение, зависящее от состояния.

Рассмотрим пример. Допустим, вы стараетесь придерживаться разработки через тестирование при создании системы для анализа текста. В ходе одного из тестов компонентов вы приказываете классам анализа текста подсчитать количество экземпляров текста «Java7», встречающихся в конкретном посте из блога. Но поскольку пост из блога — это сторонний ресурс, может возникнуть несколько сценариев отказа, практически не связанных с разрабатываемым вами алгоритмом подсчета. Иными словами, тестируемый код не изолирован, а на вызов стороннего ресурса может уходить достаточно много времени. Вот несколько распространенных сценариев отказа:

- вашему коду не удается выйти в Интернет и запросить пост из блога из-за ограничений, обусловленных применением брандмауэра в вашей организации;
- возможно, интересующий вас пост был перемещен, а переадресация на него не выполняется;
- вероятно, пост был отредактирован и в нем изменилось количество экземпляров `"Java7"`.

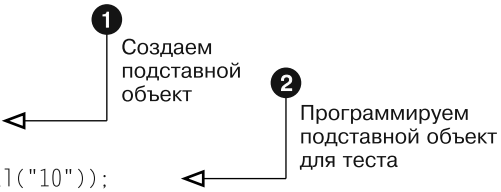
Такой тест практически невозможно написать, пользуясь одними лишь объектами-заглушками. Если вам это даже удастся, то код для каждого тестового случая получится невероятно длинным. Именно в такой ситуации нам пригодится *подставной объект* (mock object). Это особый вид тестового двойника, который можно считать предварительно программируемой заглушкой или, если хотите, суперзаглушкой. Работать с подставным объектом очень просто — вы подготавливаете его к применению, сообщаете ему, какую последовательность вызовов следует ожидать, а также указываете, как нужно реагировать на каждый конкретный вызов. Подставные объекты хорошо взаимодействуют с внедрением зависимостей — вы можете внедрить формальный объект, который будет функционировать по точно известному шаблону.

Познакомимся с этими объектами на практике и изучим простой пример, в котором вновь пойдет речь о ситуации с театральными билетами. Мы будем пользоваться

популярной библиотекой подставных объектов, которая называется Mockito и доступна по адресу http://mockito.org/. Пример показан в листинге 11.20.

Листинг 11.20. Подставные объекты в примере с театральными билетами

```
import static org.mockito.Mockito.*;
import static org.junit.Assert.*;

import java.math.BigDecimal;
import org.junit.Test;

public class TicketTest {
  @Test
  public void tenPercentDiscount() {
    Price price = mock(Price.class);
    when(price.getInitialPrice()).
         ➥ thenReturn(new BigDecimal("10"));
    Ticket ticket = new Ticket(price, new BigDecimal("0.9"));
    assertEquals(9.0, ticket.getDiscountPrice().doubleValue(), 0.000001);
    verify(price).getInitialPrice();
  }
}
```



Чтобы создать подставной объект, мы вызываем статический метод `mock()` (1) с объектом класса такого типа, который хотим имитировать. Потом мы записываем поведение, которое должен проявлять наш подставной объект. Для этого вызывается метод `when()`, указывающий, для какого метода мы собираемся записывать поведение, а потом `thenReturn()`, указывающий, каким должен быть ожидаемый результат (2). Наконец, вы убеждаетесь, что вызывали нужные методы применительно к имитируемому объекту. Этот шаг нужен для того, чтобы случайно не получить правильные результаты неверным способом.

Можно использовать подставной объект как обычный и просто передать его вашему вызову, направляемому к конструктору `Ticket`. В таком качестве подставной объект оказывается очень мощным инструментом при разработке через тестирование. Некоторые специалисты вообще не пользуются другими тестовыми двойниками, предпочитая решать практически любые задачи с помощью подставных объектов.

Независимо от того, изберете ли вы именно такой стиль разработки через тестирование, следует хорошо разбираться в тестовых двойниках (а также в некоторых случаях иметь понятие о внедрении зависимостей), чтобы уверенно заниматься как написанием, так и рефакторингом кода. Обладая такими знаниями, вы не запутаетесь даже при работе со сложными зависимостями и сторонними подсистемами.

Вам, как специалисту по Java, будет сравнительно несложно научиться работать в стиле разработки через тестирование. Но с языком Java регулярно возникает одна и та же проблема — код может получаться слишком пространным. При применении разработки через тестирование в проекте, реализуемом исключительно на языке Java, может потребоваться писать много шаблонного кода. К счастью, вы уже по-

знакомились с некоторыми альтернативными языками для виртуальной машины Java и вполне можете пользоваться ими для выстраивания разработки через тестирование в более лаконичном стиле. На самом деле один из классических путей внедрения альтернативного языка виртуальной машины Java с последующим превращением проекта в многоязычный начинается именно с тестов.

В следующем разделе мы поговорим о ScalaTest — многофункциональном тестировочном фреймворке. Мы познакомим вас со ScalaTest и расскажем, как этот фреймворк может использоваться для применения тестов JUnit к классам Java.

11.3. Знакомство со ScalaTest

Как вы помните, в разделе 7.4 мы сказали, что разработку через тестирование особенно удобно выполнять на динамическом языке. Но язык Scala также очень удобен для организации тестирования, поскольку в нем действует выведение типов. Благодаря выведению типов язык Scala кажется динамическим, несмотря на действующую в нем статическую систему типов.

Флагманский тестировочный фреймворк языка Scala называется ScalaTest. В нем содержится несколько исключительно полезных типажей и классов для осуществления всевозможных видов тестирования — от тестов компонентов в стиле JUnit до полномасштабных интеграционных и приемочных тестов. Рассмотрим пример практического использования ScalaTest, переписав с помощью этого фреймворка некоторые тесты, выполненные выше в этой главе.

В листинге 11.21 показан вариант теста из листинга 11.4, выполненный с помощью ScalaTest. Здесь мы добавляем новый тест для метода `sellTicket()` — этот тест называется `fiftyDiscountTickets()`.

Листинг 11.21. Тесты JUnit в стиле ScalaTest

```
import java.math.BigDecimal
import java.lang.IllegalArgumentException
import org.scalatest.junit.JUnitSuite
import org.scalatest.junit.ShouldMatchersForJUnit
import org.junit.Test
import org.junit.Before
import org.junit.Assert._

class RevenueTest extends JUnitSuite with ShouldMatchersForJUnit {

  var venueRevenue: TicketRevenue = _
  @Before def initialize() {
    venueRevenue = new TicketRevenue()
  }

  @Test def zeroSalesEqualsZeroRevenue() {
    assertEquals(BigDecimal.ZERO, venueRevenue estimateTotalRevenue 0);
  }

  @Test def failIfTooManyOrTooFewTicketsAreSold() {
```

```
    evaluating { venueRevenue.estimateTotalRevenue(-1) }
➥ should produce [IllegalArgumentException]
    evaluating { venueRevenue.estimateTotalRevenue(101) }
➥ should produce [IllegalArgumentException]
  }

  @Test def tenTicketsSoldIsThreeHundredInRevenue() {
    val expected = new BigDecimal("300");
    assert(expected == venueRevenue.estimateTotalRevenue(10));
  }

  @Test def fiftyDiscountTickets() {
    for (i <- 1 to 50)
➥ venueRevenue.sellTicket(new Ticket())
    for (i <- 1 to 50)
➥ venueRevenue.sellTicket(new Ticket(new StubPrice(),
➥ new BigDecimal(0.9)))
    assert(1950.0 ==
➥ venueRevenue.getRevenue().doubleValue());
  }
}
```

Ожидаемое исключение

Тестовое утверждение в стиле Scala

Одна из важных особенностей языка Scala, которую мы пока не затрагивали, касается того, как он обращается с аннотациями. Как видите, они очень похожи на аннотации Java. Действительно, ничего сверхъестественного. Ваши тесты находятся в классе, который строится на базе JUnitSuite, — таким образом, ScalaTest сможет идентифицировать этот класс как сущность, которую он способен запустить.

Вы можете легко запустить ScalaTest из командной строки, воспользовавшись нативным исполнителем тестов Scala вот так:

```
ariel:scalatest boxcat$ scala -cp /Users/boxcat/projects/tickets.jar:/Users/
    boxcat/projects/wgjd/code/lib/scalatest-1.6.1.jar:/Users/boxcat/
    projects/wgjd/code/lib/junit-4.8.2.jar org.scalatest.tools.Runner -o -s
    com.java7developer.chapter11.scalatest.RevenueTest
```

При этом запуске тестируемые классы Java находятся в архиве `tickets.jar`. Соответственно, этот файл потребуется указать в пути к классу вместе с архивами ScalaTest и JUnit.

Приведенная выше команда указывает конкретный набор тестов для запуска — это делается с помощью переключателя `-s` (если пропустить `-s`, то будут выполнены все тесты из всех тестовых наборов). Переключатель `-o` задает стандартный вывод результатов теста в окне терминала (напротив, переключатель `-e` задает вывод стандартного потока ошибок). Такой процесс в Scala называется *конфигурированием информатора для вывода* (бывают и другие информаторы, например графический). Предыдущий пример даст нам примерно такой вывод:

```
Run starting. Expected test count is: 4
RevenueTest:
- zeroSalesEqualsZeroRevenue
- failIfTooManyOrTooFewTicketsAreSold
```

```
- tenTicketsSoldIsThreeHundredInRevenue
- fiftyDiscountTickets
Run completed in 820 milliseconds.
Total number of tests run: 4
Suites: completed 1, aborted 0
Tests: succeeded 4, failed 0, ignored 0, pending 0
All tests passed.
```

Обратите внимание: тесты были скомпилированы в файл класса. Если у вас в пути к классу указаны оба нужных архива JAR — с JUnit и ScalaTest, то можно воспользоваться командой `scala` и запустить эти тесты с помощью исполнителя из JUnit:

```
ariel:scalatest boxcat$ scala -cp /Users/boxcat/projects/tickets.jar:/Users/
     boxcat/projects/wgjd/code/lib/scalatest-1.6.1.jar:/Users/boxcat/
     projects/wgjd/code/lib/junit-4.8.2.jar org.junit.runner.JUnitCore
     com.java7developer.chapter11.scalatest.RevenueTest
JUnit version 4.8.2
...
Time: 0.096

OK (4 tests)
```

В таком случае, разумеется, получится немного иной вывод, так как вы выполняете тесты с помощью другого инструмента (исполнителя JUnit).

ПРИМЕЧАНИЕ

В главе 12 мы будем пользоваться таким исполнителем JUnit при сборке проекта java7developer с применением Maven.

ТЕСТИРОВАНИЕ КОДА SCALA С ПРИМЕНЕНИЕМ SCALATEST

В этом разделе мы говорим в основном об использовании фреймворка ScalaTest для тестирования кода, написанного на Java. Но что, если вы работаете с проектом, который написан преимущественно на языке Scala?

Обычно Scala считается одним из языков, относящихся к «стабильному уровню». Поэтому если база вашего кода написана на Scala, этот код нужно тестировать не менее тщательно, чем код на Java. Таким образом, возникают все условия для применения разработки через тестирование, только вместо JUnit будет использоваться ScalaTest.

Этот краткий обзор фреймворка ScalaTest завершает наш разговор о разработке через тестирование. В главе 14 мы вновь обратимся к изученной здесь теории, когда будем обсуждать разработку через реализацию поведений (BDD) — такой подход логически проистекает из TDD.

11.4. Резюме

Разработка через тестирование призвана снизить или вообще исключить неуверенность в процессе программирования. Придерживаясь приемов разработки через тестирование, в частности выполняя цикл «красный — зеленый — рефакторинг»

при написании тестов компонентов, вы сможете избавиться от порочной практики неаккуратного «клепания» кода.

Основной тестировочной библиотекой при разработке на Java де-факто является JUnit. Она позволяет осуществлять наборы независимых тестов, при работе с ней вы можете задавать блоки кода перед тестом и после него. Механизм работы с тестовыми утверждениями, применяемый в JUnit, гарантирует, что при вызове вашей логики будет достигаться интересующий вас результат.

Существуют различные типы тестовых двойников, с помощью которых вы можете писать тесты, нацеленные на моделирование именно такого объема поведения системы, который для вас актуален. Существуют четыре типа тестовых двойников (пустые объекты, заглушки, поддельные объекты и подставные объекты). Вы можете заменить зависимость тестовым двойником, и ваш тест будет выполняться быстро и точно. Максимальная гибкость при написании тестов достигается при использовании подставных объектов.

Фреймворк ScalaTest позволяет значительно уменьшить объем шаблонного кода при написании тестов, а также получить представление о тестировании при программировании в стиле «разработка через реализацию поведений».

В следующей главе мы поговорим о феномене автоматической сборки и о методологии, называемой непрерывной интеграцией, которая основывается на разработке через тестирование. При применении непрерывной интеграции вы получаете оперативный автоматический отклик на любые вносимые изменения. Код становится необычайно прозрачным для всех членов команды разработчиков.

12 Сборка и непрерывная интеграция

В этой главе:

- почему так важно создавать конвейеры и организовывать непрерывную интеграцию (CI);
- знакомство с Maven 3 — сборочным инструментом, обеспечивающим программирование по соглашениям;
- знакомство с Jenkins — де-факто наиболее популярным инструментом для непрерывной интеграции;
- использование инструментов для статического анализа кода, в частности FindBugs и CheckStyle;
- знакомство с Leiningen — сборочным инструментом для Clojure.

История, которую мы вам собираемся поведать, основана на реальных событиях, имевших место в компании MegaCorp. Предупреждаем — все имена изменены, чтобы не навредить невинным людям! Итак, наши герои:

- Райли, новый сотрудник, молодой специалист;
- Элис и Боб, два опытных разработчика;
- Хейзел, их уставшая начальница (менеджер проектов).

Пятница, 14:00. Новая платежная функция системы Sally должна быть полностью доработана, прежде чем будут подведены итоги рабочей недели.

Райли: Ребята, помочь вам с релизом?

Элис: Да, не откажемся. Кажется, Боб уже собрал окончательную версию. Да, Боб?

Боб: Да давно уже, я ее сгенерировал в Eclipse пару недель назад.

Райли: Хм. А мы ведь уже недели две работаем с IntelliJ. И как же теперь делать сборку?

Боб: Ну, вот это и называется «практика». Не волнуйся, стажер, как-нибудь сделаем!

Райли: Ладно. Я и правда не очень в этом разбираюсь, но функция должна собраться без проблем, так?

Элис: Да, конечно, я просто сделала ветку кода две недели назад. Определенно, остальные ребята из команды не успели слишком уж изменить код.

Боб: Хм, кстати, вы же в курсе, что мы добавили изменение в Generics? (*Неловкое молчание.)*

Хейзел: Да сколько можно! Народ, почаще сверяйте курс, особенно изменения!

Райли: Так, может, я пиццу закажу? Домой мы сегодня, наверное, не попадаем?

Хейзел: Вот именно! Соображаешь, кадет!

Элис: Кстати, у меня тут гора непрочитанных сообщений по этому поводу. Каждый раз на одни и те же грабли!

Хейзел: Просто отсортируй — мы уже потеряли уйму поздних корявых релизов. Увидите, сейчас начальство разберется и покатятся головы!

Вы уже понимаете, что Элис, Боб и Райли не владеют качественными методами сборки и непрерывной интеграции (CI). Но что же такое сборка и непрерывная интеграция?

Сборка и непрерывная интеграция — это практика быстрого и систематического создания высококачественных бинарных артефактов для развертывания программ в разнообразных средах.

В командах разработчиков часто говорят о сборке или о сборочном процессе[1]. В этой главе, говоря о *сборке*, мы будем иметь в виду использование сборочного инструмента для преобразования вашего исходного кода в бинарный артефакт. Этот процесс протекает в соответствии с жизненным циклом сборки. В таких сборочных инструментах, как Maven, сборочный цикл очень длинный и детализированный, но большая его часть скрыта от разработчика. Упрощенная модель типичного жизненного цикла показана на рис. 12.1.

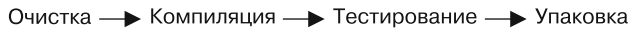


Рис. 12.1. Упрощенная картина типичного жизненного цикла сборки

Непрерывная интеграция — это процесс, в ходе которого члены команды работают с частой периодичностью, по принципу «отправляем заранее, отправляем часто». Каждый разработчик должен отправлять написанный код в систему контроля версий не реже раза в день, а сервер непрерывной интеграции будет запускать регулярные автоматизированные сборки, чтобы максимально быстро обнаруживать ошибки интеграции[2]. Реакция системы часто выводится разработчикам в виде веселой/грустной рожицы на большом экране.

Итак, почему же так важна сборка и непрерывная интеграция? В каждом из разделов этой главы мы остановимся на отдельных достоинствах такого подхода, но некоторые наиболее важные черты обобщены в табл. 12.1.

---

1 Если в вашей команде эти темы обсуждаются приглушенным, благоговейным или опасливым тоном, поверьте, эта глава для вас!

2 Это настраиваемый параметр — можно запускать сборки через определенное количество минут, после получения новой порции кода либо в специально заданные моменты времени.

Таблица 12.1. Основные причины, по которым важно не пренебрегать сборками и непрерывной интеграцией

Причина	Объяснение
Воспроизводимость	Кто угодно может запустить сборку когда угодно, где угодно. Таким образом, все разработчики смогут с удобством заниматься сборками, не обращаясь при этом к мастеру сборок, можно будет обойтись вообще без этой должности. Если новый член команды должен запустить сборку в 3:00 в воскресенье, он может смело за нее приниматься именно в это время
Быстрое оповещение	Если что-то пойдет не так — вы сразу об этом узнаете. Это особенно касается непрерывной интеграции, когда программисты работают над кодом непосредственно перед интеграцией
Согласованность	Вы знаете, какую именно версию программы развертываете, и вам точно известно, какой код применяется в каждой из версий
Управление зависимостями	Как правило, проект на Java содержит несколько зависимостей, например log4j, Hibernate, Guice и др. Управлять такими зависимостями вручную бывает очень сложно, а стоит изменить версию — и программа может сломаться. Качественные сборки и непрерывная интеграция гарантируют, что любая компиляция и сборка всегда происходят с учетом одних и тех же сторонних зависимостей

Чтобы развернуть ваш код в той или иной среде, вы должны пропустить его через сборочный цикл и превратить файл в бинарный артефакт (JAR, WAR, RAR, EAR и т. д.). В сравнительно старых проектах на Java обычно использовался сборочный инструмент Ant, в более новых применяется Maven или Gradle. Многие команды разработчиков также выполняют ночные интеграционные сборки, а иногда для проведения регулярных сборок даже задействуется специальный сервер непрерывной интеграции.

ВНИМАНИЕ

Если вы собираете JAR-файлы или другие артефакты в вашей интегрированной среде разработки (IDE), то явно нарываетесь на неприятности. Если работать таким образом, то у вас не будет воспроизводимой сборки, независимой от локальных настроек вашей IDE — а это значит, что вы сидите на пороховой бочке. Эту опасность невозможно переоценить — ответственный коллега никогда не посоветует вам собирать артефакты прямо из IDE!

Но большинству разработчиков сборка и непрерывная интеграция не кажутся настолько интересными и плодотворными областями, чтобы отдельно ими заниматься. Сборочные инструменты и серверы для непрерывной интеграции часто настраиваются на старте проекта, а потом о них благополучно забывают. Годами мы слышим примерно такие ремарки: «Зачем тратить лишнее время на сборки и обслуживание сервера непрерывной интеграции? Мы что-то написали — оно типа работает. Ну и хорошо».

Мы совершенно уверены, что, обеспечивая качественную сборку и непрерывную интеграцию, вы сможете писать код не только быстрее, но и гораздо качественнее.

Если применять сборку и непрерывную интеграцию вместе с разработкой через тестирование (эта методология подробно рассмотрена в главе 11), то вы сможете *быстро и уверенно заниматься рефакторингом*. Представьте, что у вас за спиной все время стоит консультант, помогающий работать уверенно и быстро, даже если вы решитесь на самые смелые изменения.

Мы начнем эту главу со знакомства с Maven 3, популярным сборочным инструментом, стимулирующим вас строго придерживаться сборочного цикла при разработке. Следует отметить, что Maven вызывает полярные мнения — некоторые его просто не выносят. В ходе этого знакомства мы будем собирать код на Groovy и Scala вместе с более традиционным кодом Java.

Jenkins — это исключительно популярный сервер непрерывной интеграции, который можно настраивать самыми разными способами для непрерывного выполнения сборок и генерации параметров, характеризующих качество процесса (в Jenkins действует множество плагинов). Изучая Jenkins, мы подробно остановимся на параметрах, характеризующих качество разработки кода. Эти параметры удобно отслеживать в плагинах Checkstyle и FindBugs.

Познакомившись с Maven и Jenkins, вы получите целостное представление о ходе типичной сборки проекта на Java, а также о процессе непрерывной интеграции. Наконец, мы рассмотрим сборку и развертывание под совершенно иным углом — в контексте сборочного инструмента Leiningen, применяемого в языке Clojure. Вы увидите, как реализовать очень быстрый и удобный вариант разработки через тестирование и в то же время обеспечить полномасштабные возможности сборки и развертывания.

Итак, начинаем знакомство со сборкой и непрерывной интеграцией с изучения Maven 3!

12.1. Знакомство с Maven 3

Maven — это популярный, но крайне неоднозначно воспринимаемый сборочный инструмент для работы с Java и родственными языками для виртуальной машины Java. В основе работы инструмента лежит такая посылка: строгий жизненный цикл сборки, подкрепленный мощной системой управления зависимостями, необходим для успешного выполнения сборок. Но Maven — это не просто сборочный инструмент. Скорее его можно назвать инструментом для управления проектами, отвечающим за техническую составляющую вашего проекта. Действительно, сборочные сценарии называются POM-объектами (POM — это модель объекта проекта). Такие POM-файлы пишутся на XML, каждый проект или модуль Maven будет сопровождаться файлом `pom.xml`.

ПРИМЕЧАНИЕ

Планируется организовать поддержку POM-файлов на альтернативных языках, чтобы при необходимости пользователи могли рассчитывать на максимальную гибкость (примерно как при работе со сборочным инструментом Gradle).

НЕСКОЛЬКО СЛОВ ОБ ANT И GRADLE

Ant — популярный сборочный инструмент, особенно активно использовавшийся в сравнительно старых проектах на Java. Долгое время этот инструмент де-факто применялся по умолчанию. Мы не будем рассказывать о нем здесь, так как о нем уже множество раз писали ранее. Но гораздо важнее, что, на наш взгляд, Ant не подходит для общепринятого сборочного цикла и не имеет набора общепринятых (обязательных) целей для сборки. Таким образом, разработчику приходится разбираться в тонкостях каждой сборки Ant, с которой ему доведется столкнуться. Если вам необходимо работать именно с Ant, то на сайте http://ant.apache.org вы найдете всю необходимую подробную информацию.

Gradle — замечательный новичок в ряду сборочных инструментов. В нем применяется подход, противоположный избранному в Maven. В Gradle вы освободитесь от жестких ограничений и сможете выстраивать сборку по своему усмотрению. Как и в Maven, в Gradle предоставляется управление зависимостями и множество других возможностей. Если вы хотите попробовать Gradle на практике, посетите сайт www.gradle.org, где подробно рассказано об этом инструменте.

Поскольку мы хотим научить вас качественным приемам организации сборок, то все же полагаем, что для достижения нашей цели лучше всего подходит Maven. При работе с ним вы должны придерживаться строго организованного сборочного цикла, а также можете запускать сборки любых проектов Maven, подготовленных где и когда угодно.

В Maven приветствуется *программирование в соответствии с соглашениями*. Это особый подход к компоновке исходного кода, фильтрации свойств и т. д. Некоторых разработчиков он может удручать, но вы даже не представляете, какая огромная умственная работа по организации жизненных циклов была проделана за долгие годы и вложена в Maven. Как правило, Maven подсказывает вам наиболее целесообразный путь сборки. Тем, кто люто ненавидит любые соглашения, Maven позволяет переопределять настройки, заданные по умолчанию, но тогда получается более пространный и менее стандартизированный набор сборочных сценариев.

Чтобы выполнять сборки с помощью Maven, система требует от вас запустить одну или несколько *целей* (goal), представляющих собой конкретные задачи (такие как компилирование кода, выполнение тестов и т. д.). Цели связываются в задаваемый по умолчанию сборочный цикл. Поэтому, если вы приказываете Maven запустить какие-то тесты (скажем, `mvn test`), он скомпилирует и ваш код, и тестовый код и лишь потом попытается запустить тесты. Одним словом, он требует от вас придерживаться правильного сборочного цикла.

Если вы еще не скачали и не установили Maven, откройте раздел A.2 приложения A. Как только выполните эту работу, возвращайтесь сюда и приступайте к созданию своего первого проекта в Maven.

12.2. Экспресс-проект с Maven 3

В Maven действует принцип программирования в соответствии с соглашениями. Такие соглашения, регулирующие структуру проекта, становятся очевидны, если

создать небольшой экспресс-проект. Типичная структура проекта, предпочитаемая в Maven, выглядит примерно так:

```
project
|-- pom.xml
`-- src
    |-- main
    | `-- java
    | `-- com
    | `-- company
    | `-- project
    | `-- App.java
    | `-- resources
    `-- test
        `-- java
            `-- com
                `-- company
                    `-- project
                        `-- AppTest.java
        `-- resources
`-- target
```

В частности, в рамках таких соглашений Maven отделяет основной код от тестового. Кроме того, в нем есть специальные каталоги ресурсов, содержащие все прочие файлы, которые необходимо включить в состав сборки (например, файл `log4.xml` для логирования, конфигурационные файлы Hibernate и другие подобные ресурсы). Файл `pom.xml` — это сборочный сценарий для Maven. Он подробно рассмотрен в приложении Е.

Если вы занимаетесь мнрогоязычным программированием, то ваш исходный код на Scala и Groovy должен быть структурирован точно так же, как и исходный код Java (находящийся в каталоге java). Просто в первых двух случаях корневые каталоги будут называться scala и groovy соответственно. В проекте Maven коды на Scala, Java и Groovy могут мирно соседствовать друг с другом.

Целевой каталог не будет создан до тех пор, пока не будет запущена сборка. Все классы, артефакты, отчеты и другие файлы, генерируемые при сборке, будут попадать в этот каталог. Подробное описание структуры проекта, требуемой для работы с Maven, приводится в разделе Introduction to the Standard Directory Layout («Введение в стандартную структуру каталогов») на сайте Maven (http://maven.apache.org/guides/introduction/introduction-to-the-standard-directory-layout.html).

Чтобы создать такую правильную структуру для нового проекта, выполните следующую цель с заданными параметрами:

```
mvn archetype:generate
    -DgroupId=com.mycompany.app
    -DartifactId=my-app
    -DarchetypeArtifactId=maven-archetype-quickstart
    -DinteractiveMode=false
```

На текущем этапе ваша консоль начнет заполняться выводом от Maven, сообщающим, что система скачивает плагины и сторонние библиотеки. Они нужны

Maven для работы над заданной целью. По умолчанию они загружаются из Maven Central — де-факто основного онлайнового репозитория, в котором хранятся такие артефакты.

ПОЧЕМУ КАЖЕТСЯ, ЧТО MAVEN ГОТОВ ВЫКАЧАТЬ ВЕСЬ ИНТЕРНЕТ?

«Ох, опять этот Maven решил выкачать весь Интернет», — любят шутить те, кому часто приходится заниматься сборкой Java-проектов. Но виноват ли в этом Maven? Мы считаем, что такое поведение обусловлено двумя основными причинами. Во-первых, дело заключается в неаккуратном отношении к упаковке и управлению зависимостями, чем грешат многие разработчики сторонних библиотек (например, некоторые указывают зависимость в файле `pom.xml`, хотя это и не требуется). Вторая причина — слабость системы упаковки, основанной на JAR-файлах, существовавшая всегда. Из-за этой слабости не удается сформировать более рафинированный набор зависимостей.

Кроме информации о скачивании файлов (`Downloading…`), в вашей консоли должны появиться следующие данные:

```
[INFO] ------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------
[INFO] Total time: 1.703s
[INFO] Finished at: Fri Jun 24 13:51:58 BST 2011
[INFO] Final Memory: 6M/16M
[INFO] ------------------------------------------------------------------
```

Если на данном этапе сборка сорвется, то дело, вероятно, в том, что ваш прокси-сервер блокирует доступ к Maven Central — репозиторию, где хранятся плагины и сторонние библиотеки. Чтобы решить эту проблему, просто отредактируйте файл `settings.xml` (он упоминается в разделе A.2 приложения A). Добавьте в него следующий раздел, указав нужные значения для различных элементов:

```
<proxies>
  <proxy>
    <active>true</active>
    <protocol></protocol>
    <username></username>
    <password></password>
    <host></host>
    <port></port>
  </proxy>
</proxies>
```

Перезапустите сборку и увидите, что в вашем каталоге создан проект my-app.

СОВЕТ

Добавьте конфигурацию прокси-сервера в файл $M2_HOME/conf/settings.xml, если кто-либо из членов вашей команды сталкивается с подобной проблемой.

Maven поддерживает практически неограниченный набор архетипов (шаблонов для проектов). Если вы хотите сгенерировать конкретный тип проекта, например проект для JEE6, то можете выполнить цель `mvn archetype:generate` и просто следовать подсказкам системы.

Чтобы исследовать Maven более подробно, рассмотрим проект, для которого уже есть исходный код и готовые тесты. На его примере мы сможем без труда продемонстрировать вам весь сборочный цикл.

12.3. Maven 3 — сборка java7developer

Помните сборочный цикл, показанный на рис. 12.1? Maven реализует подобный цикл, и сейчас нам предстоит собрать исходный код, сопровождающий эту книгу, внимательно проработав все касающиеся его этапы сборки. Несмотря на то что исходный код к этой книге не образует цельного приложения, мы назовем эту сборку *«проект java7developer»*.

В этом разделе мы сосредоточимся на следующих вопросах:

- исследуем основы файла POM, применямого в Maven (то есть вашего сборочного сценария);
- научимся компилировать, тестировать и упаковывать код (в том числе на языках Scala и Groovy);
- научимся работать сразу в нескольких окружениях — для этого будем пользоваться профилями;
- научимся генерировать сайт, содержащий различные отчеты.

Итак, начнем с файла `pom.xml`, определяющего проект java7developer.

12.3.1. Файл POM

Файл `pom.xml` представляет проект java7developer, со всеми его плагинами, ресурсами и другими элементами, необходимыми для сборки. Вы найдете файл `pom.xml` в корне того каталога, в который разархивировали код проекта по этой книге (здесь и далее мы будем называть это местоположение `$BOOK_CODE`). В файле POM содержится четыре основных раздела:

- базовая информация о проекте;
- конфигурация сборки;
- управление зависимостями;
- профили.

Это довольно длинный файл, но он гораздо проще, чем может показаться на первый взгляд. Если вы хотите разобрать его до мельчайших подробностей, можете посмотреть справку по POM (раздел POM Reference на сайте Maven http://maven.apache.org/pom.html).

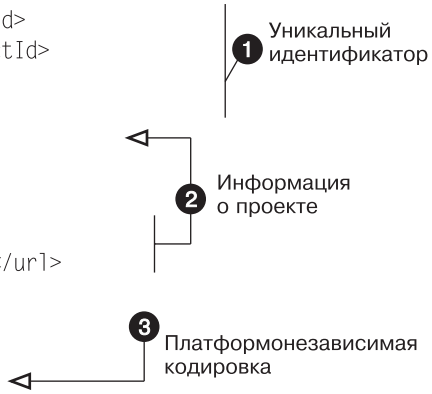
Мы расскажем о содержимом файла `pom.xml` для проекта java7developer в четыре этапа и начнем с базовой информации о проекте.

Базовая информация о проекте

Maven позволяет указывать в файле pom.xml множество основополагающей информации о проекте. В листинге 12.1 приведен необходимый минимум, который мы считаем достаточным для того, чтобы приступить к работе.

Листинг 12.1. POM — базовая информация о проекте

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.java7developer</groupId>
  <artifactId>java7developer</artifactId>
  <packaging>jar</packaging>
  <version>1.0.0</version>
  <name>java7developer</name>
  <description>
    Project source code for the book!
  </description>
  <url>http://www.java7developer.com</url>
  <properties>
    <project.build.sourceEncoding>
      UTF-8
    </project.build.sourceEncoding>
  </properties>
...
```



Значение элемента <groupId>, равное com.java7developer, является первой частью уникального идентификатора данного артефакта в репозитории Maven ❶. Значение элемента <artifactId>, равное java7developer, — это вторая часть уникального идентификатора. Значение элемента <packaging>, равное jar, сообщает Maven, что вы планируете собрать файл JAR (как вы догадываетесь, здесь также можно указывать значения war, ear, rar, sar и har). Последний элемент уникального идентификатора — <version>, значение которого равно 1.0.0[1] и указывает, какую именно версию вы собираете. Это значение постепенно возрастает по мере появления новых версий SNAPSHOT, создаваемых при выполнении новых релизов Maven.

Здесь также указаны значения <projectName> и <url>, за которыми следует еще много другой, необязательной информации по проекту ❷. Чтобы сборка была согласованной на любых платформах, вы также указываете исходную кодировку <sourceEncoding>, задавая для нее значение UTF-8 ❸.

Для того чтобы показать более широкий контекст, в котором используется данный файл, отметим, что на основе этой информации Maven соберет артефакт java7developer-1.0.0.jar, который будет сохранен в репозитории Maven по адресу com/java7developer/1.0.0.

---

[1] Номер соответствует схеме нумерации версий Major.Minor.Trivial (основная — второстепенная — тривиальная), мы предпочитаем работать именно с ней!

ВЕРСИИ MAVEN И МГНОВЕННЫЕ СНИМКИ (SNAPSHOT)

Следуя концепции программирования по соглашениям, Maven нумерует ваши версии в формате major.minor.trivial (основная версия — второстепенная версия — тривиальная версия). При работе с артефактами, зависящими от времени, он традиционно добавляет к номеру версии суффикс `-SNAPSHOT`. Например, если ваша команда в несколько этапов собирает JAR-файл для планируемого релиза 1.0.0, то в соответствии с действующими соглашениями нужно указывать промежуточные версии как `1.0.0-SNAPSHOT`. Таким образом, вы сообщаете плагинам Maven, что идет доработка неготовой версии, и система будет обращаться с ней соответственно. Когда вы выдаете этот артефакт в готовую версию, ему присваивается номер версии 1.0.0, а следующая версия, которая будет выпущена после исправления ошибок, получит номер `1.0.1-SNAPSHOT`.

Maven помогает автоматизировать весь этот процесс с помощью плагина Release (Релиз). Подробнее о нем можно почитать на соответствующей странице сайта Maven (http://maven.apache.org/plugins/maven-release-plugin/). Теперь, когда мы разобрались с разделом, включающим базовую информацию о проекте, рассмотрим раздел `<build>`.

Конфигурация сборки

В сборочном разделе находятся плагины[1] и соответствующие им варианты конфигурации, необходимые для достижения целей, заданных для жизненного цикла сборки. Во многих проектах этот раздел весьма невелик, так как обычно удается обойтись задаваемыми по умолчанию плагинами и их стандартными настройками.

В нашем проекте java7developer в разделе `<build>` будет содержаться несколько плагинов, переопределяющих некоторые стандартные настройки. Мы так поступили, чтобы в проекте java7developer можно было выполнять несколько видов работы, в частности:

- собирать код Java 7;
- собирать код Scala и Groovy;
- запускать тесты для Java, Scala и Groovy;
- представлять отчеты с параметрами кода, генерируемые в плагинах Checkstyle и FindBugs.

Плагины — это артефакты, построенные на основе JAR-архивов (в основном написанные на Java). Чтобы сконфигурировать сборочный плагин, его нужно вставить в раздел `<build><plugins>` вашего файла `pom.xml`. Как и все артефакты Maven, каждый плагин снабжается уникальным идентификатором. Поэтому для плагинов также следует указывать информацию `<groupId>`, `<artifactId>` и `<version>`. После этого вся дополнительная информация, необходимая для конфигурации плагина, указывается в разделе `<configuration>`, и подробности из этого раздела являются специфичными для каждого плагина. Например, в плагине компилятора есть конфигурационные элементы `<source>`, `<target>` и `<showWarnings>`, содержащие информацию, присущую именно компилятору.

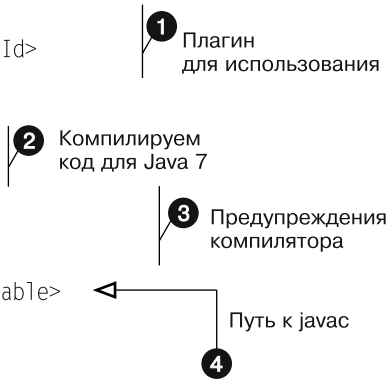
---

[1] Если вам требуется сконфигурировать какие-либо части сборки, то можете посмотреть полный список плагинов Maven на странице http://maven.apache.org/plugins/index.html

В листинге 12.2 показан раздел о конфигурации сборки, относящийся к проекту java7developer (полный листинг со всеми необходимыми пояснениями приведен в приложении Е).

Листинг 12.2. Файл POM – информация о сборке

```
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-compiler-plugin</artifactId>
      <version>2.3.2</version>
      <configuration>
        <source>1.7</source>
        <target>1.7</target>
        <showDeprecation>true</showDeprecation>
        <showWarnings>true</showWarnings>
        <fork>true</fork>
        <executable>${jdk.javac.fullpath}</executable>
      </configuration>
    </plugin>

    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-surefire-plugin</artifactId>
      <version>2.9</version>
      <configuration>
        <excludes>
          <exclude>
            com/java7developer/chapter11/
➥ listing_11_2/TicketRevenueTest.java
          </exclude>
          <exclude>
            com/java7developer/chapter11/
➥ listing_11_7/TicketTest.java
          </exclude>
          ...
        </excludes>
      </configuration>
    </plugin>
  </plugins>
</build>
```



Здесь потребуется указать, что вы используете плагин компилятора (конкретную версию) 1, поскольку вы собираетесь изменить стандартное поведение, реализуемое при компиляции кода Java 1.5 в код Java 1.7 2.

Поскольку вы уже нарушили соглашение, можете добавить к компилятору еще несколько полезных предупреждающих сообщений 3. Далее нужно убедиться, что вы правильно указали, где установлена Java 7 4. Просто скопируйте файл

sample_<os>_build.properties для вашей операционной системы в build.properties, а затем отредактируйте значение jdk.javac.fullpath, чтобы оно было воспринято системой.

Плагин Surefire позволяет конфигурировать тесты. В конфигурации данного проекта мы специально опустили некоторое количество тестов ❺, которые заведомо неуспешны (мы рассматривали два примера подобных тестов в главе 11, посвященной разработке через тестирование).

Итак, мы разобрались с разделом, в котором регламентируется сборка проекта. Переходим к важнейшей части файла POM, в которой выполняется управление зависимостями.

Управление зависимостями

Список зависимостей в большинстве проектов Java бывает достаточно длинным, и проект java7developer не исключение. Maven помогает вам управлять такими зависимостями, опираясь на богатейший набор сторонних библиотек, собранных в репозитории Maven Central. Крайне важно отметить, что в этих сторонних библиотеках есть собственные файлы pom.xml, где объявляются зависимости, присущие этим библиотекам. На базе этой информации Maven может определить и скачать любые другие библиотеки, необходимые для вашего проекта.

Изначально вы будете работать с двумя основными областями видимости — compile и test[1]. В целом определение двух этих областей видимости можно сравнить с записью JAR-файлов в путь CLASSPATH; в первом случае — для компиляции кода, а во втором — для запуска тестов. В листинге 12.3 показан раздел <dependencies> для проекта java7developer. Полный листинг и все необходимые пояснения приведены в приложении E.

Листинг 12.3. POM — зависимости

```
<dependencies>
  <dependency>
    <groupId>com.google.inject</groupId>
    <artifactId>guice</artifactId>
    <version>3.0</version>
    <scope>compile</scope>
  </dependency>
  <dependency>
    <groupId>javax.inject</groupId>
    <artifactId>javax.inject</artifactId>
    <version>1</version>
    <scope>compile</scope>
  </dependency>

  <dependency>
    <groupId>junit</groupId>
```

❶ Уникальный ID артефакта

❷ Область видимости для компиляции

[1] В проектах для J2EE/JEE также обычно присутствуют зависимости, объявляемые в области видимости runtime.

```
    <artifactId>junit</artifactId>
    <version>4.8.2</version>
    <scope>test</scope>
  </dependency>
  <dependency>
    <groupId>org.mockito</groupId>
    <artifactId>mockito-all</artifactId>
    <version>1.8.5</version>
    <scope>test</scope>
  </dependency>
  ...
</dependencies>
```

Чтобы Maven мог найти артефакт, на который вы ссылаетесь, системе требуется верная информация из `<groupId>`, `<artifactId>` и `<version>` ❶. Как мы уже намекали выше, если задать для `<scope>` значение `compile` ❷, то фактически мы добавим в `CLASSPATH` файлы JAR, нужные для компиляции кода. Если же установить `<scope>` в значение `test` ❸, то мы гарантируем, что в `CLASSPATH` будут добавлены файлы JAR, нужные для тестирования. Maven сможет скомпилировать и запустить тесты.

Но как узнать, какие именно значения `<groupId>`, `<artifactId>` и `<version>` следует указать? Практически всегда нужные значения можно найти в репозитории Maven Central по адресу http://search.maven.org/.

Если вам не удастся найти подходящий артефакт, то можете сами вручную скачать и установить плагин. Для этого воспользуйтесь целью `install:install-file`. Вот пример, в котором устанавливается библиотека `asm-4.0_RC1.jar`.

```
mvn install:install-file
-Dfile=asm-4.0_RC1.jar
-DgroupId=org.ow2.asm
-DartifactId=asm
-Dversion=4.0_RC1
-Dpackaging=jar
```

После запуска этой команды вы должны обнаружить, что артефакт был установлен в ваш локальный репозиторий по адресу `$HOME/.m2/repository/org/ow2/asm/asm/4.0_RC1/`, как если бы Maven скачал его сам!

МЕНЕДЖЕР АРТЕФАКТОВ

Скачав стороннюю библиотеку, вы ее, конечно, установите, а вот как быть с остальными членами команды? Подобная проблема возникает и в тех случаях, когда вы создаете артефакт, который желаете использовать совместно с коллегами, но не можете поместить его в Maven Central (так как артефакт является проприетарным кодом).

Чтобы решить такую проблему, следует пользоваться бинарным менеджером артефактов, например Nexus (http://nexus.sonatype.org/). Менеджер артефактов действует как локальная копия Maven Central, которой можете пользоваться и вы, и ваша команда. Можете совместно использовать артефакты в рамках своей команды, но не вне ее. В большинстве менеджеров артефактов также кэшируются и Maven Central, и другие репозитории. Так менеджер артефактов становится незаменимым складом полезностей для вас и вашей команды.

Последняя часть файла POM, с которой необходимо разобраться, — это раздел профилей, нужный фактически для подготовки экосистемы для ваших сборок.

Профили

Профили позволяют Maven готовить окружения, необходимые для функционирования ваших сборок (например, окружение для проведения пользовательского приемочного тестирования, которое отличается от обычного рабочего окружения), а также вносить другие небольшие вариации в вашу нормальную сборку. В проекте java7developer рассмотрим профиль, в котором отключены предупреждения компилятора и предупреждения об использовании *устаревающих конструкций* (deprecation warnings). Соответствующий код показан в листинге 12.4.

Листинг 12.4. POM — профили

```
<profiles>
  <profile>
    <id>ignore-compiler-warnings</id>
    <build>
      <plugins>
        <plugin>
          <groupId>org.apache.maven.plugins</groupId>
          <artifactId>maven-compiler-plugin</artifactId>
          <version>2.3.2</version>
          <configuration>
            <source>1.7</source>
            <target>1.7</target>
            <showDeprecation>false</showDeprecation>
            <showWarnings>false</showWarnings>
            <fork>true</fork>
            <executable>${jdk.javac.fullpath}</executable>
          </configuration>
        </plugin>
      </plugins>
    </build>
  </profile>
</profiles>
```

1 ID для этого профиля

2 Отключение предупреждений

Для ссылки на профиль, который вы собираетесь использовать, применяется параметр `-P <id>`, если вы работаете с Maven (например, `mvn compile -P ignore-compile-warnings`) 1. После активизации этого профиля будет применяться заданная здесь версия компилятора, а какие-либо предупреждения компилятора или предупреждения об использовании устаревающих конструкций не будут выводиться 2.

Вы можете подробнее почитать об использовании профилей и о том, как они применяются при подготовке окружений, в разделе сайта Maven «Введение в профили сборки» по адресу http://maven.apache.org/guides/introduction/introduction-to-profiles.html.

Итак, теперь мы окончательно разобрались в структуре файла `pom.xml` для проекта java7developer, и вам, наверное, не терпится собрать проект, верно?

12.3.2. Запуск примеров

Полагаем, вы уже загрузили листинги с кодом для этой книги. Среди них вы найдете и несколько файлов pom.xml. Это файлы, управляющие ходом сборок Maven.

В этом подразделе мы подробно изучим стадии, входящие в типичный цикл сборки Maven (clean, compile, test и install). Первая стадия в рамках сборочного цикла — очистить пространство от всех артефактов, которые могли остаться там после окончания последней сборки.

Очистка

При выполнении стадии Maven clean (очистка) удаляется конечный каталог (target). Чтобы увидеть эту операцию на практике, перейдите в каталог $BOOK_CODE и выполните стадию Maven clean.

```
cd $BOOK_CODE
mvn clean
```

В отличие от других стадий, выполняемых в ходе жизненного цикла сборки Maven, clean не вызывается автоматически. Если вы хотите удалить артефакты, оставшиеся от предыдущей сборки, то всегда сами включайте стадию clean.

Теперь, когда вы удалили все остатки предыдущих сборок, обычно требуется выполнить следующую стадию в рамках жизненного цикла сборки — скомпилировать код (compile).

Компиляция

Стадия Maven compile использует конфигурацию плагина компиляции, записанную в файле pom.xml, и на основании этой информации компилирует исходный код, находящийся в каталогах src/main/java, src/main/scala и src/main/groovy. Фактически при этом выполняются компиляторы Java, Scala и Groovy (javac, scalac и groovyc), а в CLASSPATH добавляются зависимости, чья область видимости ограничена compile.

Скомпилированные в результате классы оказываются в каталоге target/classes. Чтобы увидеть это на практике, выполним следующую стадию Maven:

```
mvn compile
```

Стадия compile должна выполниться достаточно быстро, и в консоли вы увидите примерно такой вывод:

```
...
[INFO] [properties:read-project-properties {execution: default}]
[INFO] [groovy:generateStubs {execution: default}]
[INFO] Generated 22 Java stubs
[INFO] [resources:resources {execution: default-resources}]
[INFO] Using 'UTF-8' encoding to copy filtered resources.
[INFO] Copying 2 resources
[INFO] [compiler:compile {execution: default-compile}]
[INFO] Compiling 119 source files to
     C:\Projects\workspace3.6\code\trunk\target\classes
[INFO] [scala:compile {execution: default}]
```

```
[INFO] Checking for multiple versions of scala
[INFO] includes = [**/*.scala,**/*.java,]
[INFO] excludes = []
[INFO] C:\Projects\workspace3.6\code\trunk\src\main\java:-1: info: compiling
[INFO] C:\Projects\workspace3.6\code\trunk\target\generated-sources\groovy-
     stubs\main:-1: info: compiling
[INFO] C:\Projects\workspace3.6\code\trunk\src\main\groovy:-1: info:
     compiling
[INFO] C:\Projects\workspace3.6\code\trunk\src\main\scala:-1: info:
     compiling
[INFO] Compiling 143 source files to
     C:\Projects\workspace3.6\code\trunk\target\classes at 1312716331031
[INFO] prepare-compile in 0 s
[INFO] compile in 12 s
[INFO] [groovy:compile {execution: default}]
[INFO] Compiled 26 Groovy classes
[INFO]------------------------------------------------------------------
[INFO] BUILD SUCCESSFUL
[INFO] ------------------------------------------------------------------
[INFO] Total time: 43 seconds
[INFO] Finished at: Sun Aug 07 12:25:44 BST 2011
[INFO] Final Memory: 33M/79M
[INFO] ------------------------------------------------------------------
```

На данном этапе тестовые классы в каталогах `src/test/java`, `src/test/scala` и `src/test/groovy` еще не скомпилированы. Для этого предусмотрена специальная стадия `test-compile`, но, как правило, достаточно просто запросить Maven выпустить стадию `test`.

Тестирование

Именно стадия `test` позволяет в полной мере оценить, каков жизненный цикл сборки Maven в действии. Когда вы приказываете Maven выполнить тесты, система знает, что должна выполнить все более ранние стадии жизненного цикла сборки, чтобы стадия `test` была выполнена успешно (в частности, это касается стадий `compile`, `test-compile` и многих других).

Maven запускает тесты с помощью плагина Surefire, пользуясь поставщиком тестов (в данном случае — JUnit), который вы указываете в файле `pom.xml` в качестве одной из зависимостей, ограниченных областью видимости `test`. Maven не только запускает тест, но и готовит файлы с отчетами, которые позже можно проанализировать для исследования заведомо неуспешных тестов и сбора тестовых показателей.

Чтобы увидеть этот этап на практике, выполните следующие стадии Maven:

```
mvn clean test
```

Как только Maven закончит компилировать и запускать тесты, вы должны получить на экране примерно такой вывод:

```
...
Running com.java7developer.chapter11.listing_11_3.TicketRevenueTest
Tests run: 5, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0 sec
```

```
Running com.java7developer.chapter11.listing_11_4.TicketRevenueTest
Tests run: 5, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0 sec
Running com.java7developer.chapter11.listing_11_5.TicketTest
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.015 sec

Results :

Tests run: 20, Failures: 0, Errors: 0, Skipped: 0

[INFO]------------------------------------------------------------------
[INFO] BUILD SUCCESSFUL
[INFO]------------------------------------------------------------------
[INFO] Total time: 16 seconds
[INFO] Finished at: Wed Jul 06 13:50:07 BST 2011
[INFO] Final Memory: 24M/58M
[INFO]------------------------------------------------------------------
```

Результаты тестов сохраняются в каталоге `target/surefire-reports`. Можете уже сейчас изучить текстовые файлы, которые там находятся. Позже мы будем просматривать эти результаты через удобный графический веб-интерфейс.

СОВЕТ

Как видите, мы указали здесь и стадию clean. Мы это сделали по привычке, просто на тот случай, если при сборке нам попадется какой-нибудь старый мусор.

Итак, вы скомпилировали и протестировали код, осталось только его упаковать. Хотя для этого можно просто воспользоваться стадией `package`, мы предпочтем `install`. Как это делается — подробно описано дальше.

Установка

Стадия `install` решает две основные задачи. Она упаковывает код в соответствии с указаниями, приведенными в элементе `<packaging>` вашего файла `pom.xml` (в данном случае создается JAR-файл). Этот артефакт устанавливается в вашем локальном репозитории Maven (по адресу `$HOME/.m2/repository`), после чего он может использоваться в виде зависимости в других ваших проектах. Как обычно, если система обнаружит, что какие-то более ранние этапы жизненного цикла сборки не были выполнены, она выполнит и их.

Переходим к делу, выполняем такую стадию Maven:

```
mvn install
```

Как только Maven выполнит ее, вы должны увидеть на экране примерно следующий вывод:

```
...
[INFO] [jar:jar {execution: default-jar}]
[INFO] Building jar: C:\Projects\workspace3.6\code\trunk\target\
java7developer-1.0.0.jar
[INFO] [install:install {execution: default-install}]
[INFO] Installing C:\Projects\workspace3.6\code\trunk\target\java7developer-
```

```
     1.0.0.jar
to C:\Documents and Settings\Admin\.m2\repository\com\java7developer\
     java7developer\1.0.0\java7developer-1.0.0.jar
[INFO]------------------------------------------------------------------------
[INFO] BUILD SUCCESSFUL
[INFO]------------------------------------------------------------------------
[INFO] Total time: 17 seconds
[INFO] Finished at: Wed Jul 06 13:53:04 BST 2011
[INFO] Final Memory: 28M/66M
[INFO]------------------------------------------------------------------------
```

Вы сможете просмотреть артефакт `java7developer-1.0.0.jar` в конечном каталоге (это результат выполнения стадии `package`), а также в вашем локальном репозитории Maven по адресу `$HOME/.m2/repository/com.java7developer/1.0.0`.

СОВЕТ

Возможно, вы пожелаете выделить ваш код на Scala и Groovy в отдельные JAR-файлы. Maven позволяет это сделать, но не забывайте, что в Maven любой отдельный артефакт JAR должен быть самодостаточным проектом. Таким образом, вас может заинтересовать концепция мультимодального проекта, присутствующая в Maven. См. на сайте Maven раздел «Руководство по работе с множественными модулями» по адресу http://maven.apache.org/guides/mini/guide-multiple-modules.html.

Как правило, программисты работают командами и совместно задействуют общую базу кода. Итак, как же нам обеспечить быстрое создание надежных сборок, которыми смогут пользоваться все заинтересованные члены команды? Именно здесь нам пригодится сервер непрерывной интеграции. Что касается разработки на Java, подобным сервером номер один для этого языка сегодня является Jenkins.

12.4. Jenkins — обеспечение непрерывной интеграции

Чтобы обеспечить качественный сборочный процесс в условиях непрерывной интеграции, требуется одновременно и дисциплина разработчика, и правильно подобранный инструментарий. Для поддержки основных характеристик добротного процесса непрерывной интеграции Jenkins предоставляет многие необходимые средства, перечисленные в табл. 12.2.

Таблица 12.2. Характерные черты качественного процесса непрерывной интеграции и их поддержка в Jenkins

Черта	Вклад Jenkins
Регулярные отправки кода	Зависит от разработчиков
Сборка каждой отправленной порции кода	Jenkins может выполнять сборку каждый раз, когда система зафиксирует поступление новой порции кода в репозиторий для контроля версий

Черта	Вклад Jenkins
Быстрые сборки	Этот параметр наиболее важен для сборок, используемых при тестировании компонентов, поскольку такие сборки должны иметь как можно более краткий оборотный цикл. Здесь Jenkins может помочь вам, отсылая задачи на второстепенные узлы. Но именно разработчик должен создать гибкий сценарий сборки и сконфигурировать Jenkins так, чтобы система вовремя вызывала нужные стадии в рамках выполнения жизненного цикла сборки
Легко отслеживаемые результаты	В Jenkins имеется информационная панель (на веб-основе), а также множество методов, поставляющих уведомления

Все серверы непрерывной интеграции способны опрашивать репозиторий контроля версий и выполнять стадии `compile` и `test` в рамках жизненного цикла сборки. Jenkins особенно хорош своим удобным в использовании графическим интерфейсом и развернутой экосистемой плагинов.

Пользовательский интерфейс Jenkins очень удобен на этапе конфигурирования самого сервера и его плагинов. Система часто применяет вызовы в стиле AJAX, помогающие проверять допустимость ввода по мере заполнения каждого из полей. Кроме того, Jenkins предлагает множество контекстной справочной информации. Чтобы наладить работу Jenkins, не нужно быть семи пядей во лбу.

Jenkins обладает обширной экосистемой плагинов, позволяет опрашивать практически любые репозитории, применяемые для контроля версий, запускать сборки для нескольких языков сразу и просматривать множество информативных отчетов о вашем коде.

JENKINS И HUDSON

В специальной литературе и в Интернете наблюдается небольшая путаница с названием этого сервера непрерывной интеграции. Jenkins — это сравнительно новое ответвление проекта Hudson, заинтересовавшее большинство разработчиков и активных членов сообщества, занятых в родительском проекте. Hudson был и остается отличным самодостаточным сервером непрерывной интеграции, но Jenkins сейчас определенно более активен, чем Hudson.

Jenkins — свободно распространяемая программа с открытым кодом. Этот сервер объединяет очень активное сообщество, где всегда готовы поделиться знаниями со сравнительно неопытными участниками.

В приложении D описано, как скачать и установить Jenkins. Как только закончите эту работу, возвращайтесь сюда и читайте дальше.

ВНИМАНИЕ

Предполагается, что вы установите Jenkins в виде WAR-файла на вашем любимом веб-сервере, а базовый URL для установки Jenkins будет http://localhost:8080/jenkins/. Если запустить непосредственно сам WAR-файл, то базовый URL будет таким: http://localhost:8080/.

В этом разделе мы рассмотрим основы конфигурирования простейшей установки Jenkins, потом поговорим о том, как поставить задачу по сборке и выполнить

ее. В качестве примера используем проект java7developer, но вы можете проделать то же самое на вашем любимом проекте.

Чтобы настроить Jenkins на мониторинг репозитория вашего исходного кода и выполнять сборки, сначала нужно настроить базовую конфигурацию.

12.4.1. Базовая конфигурация

Для начала откроем главную страницу Jenkins по адресу http://localhost:8080/jenkins/. Чтобы приступить к конфигурации Jenkins, щелкните в меню слева на ссылке Manage Jenkins (Управление Jenkins) (http://localhost:8080/jenkins/manage). На странице управления будут перечислены различные параметры настройки, с которыми стоит познакомиться поближе.

Пока щелкните на ссылке Configure System (Сконфигурировать систему) по адресу http://localhost:8080/jenkins/configure. Вы должны оказаться на экране, верхняя часть которого выглядит примерно как на рис. 12.2.

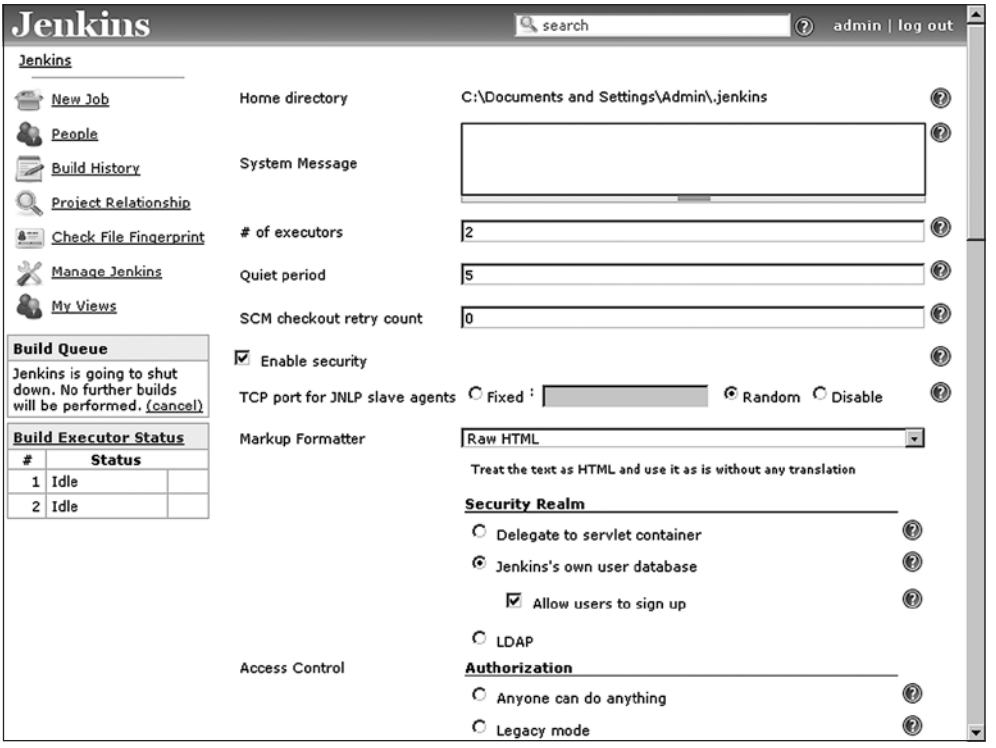


Рис. 12.2. Страница для конфигурации Jenkins

В верхней части этого экрана Jenkins сообщает вам, где находится его домашний каталог. Если вам в какой-то ситуации потребуется выполнить конфигурацию вне пользовательского интерфейса, можете этим заняться.

СОВЕТ

Если вы устанавливаете Jenkins сразу для всей команды и должны позаботиться о безопасности, то установите флажки Enable Security (Активизировать функции безопасности) и Prevent Cross Site Request Forgery Exploits (Предотвращать эксплойты, связанные с подделкой межсайтовых запросов) и организуйте соответствующую конфигурацию. Для начала лучше всего воспользоваться собственной базой данных Jenkins. Всегда можно переключиться на использование корпоративного протокола LDAP (упрощенный протокол доступа к каталогам) или на применение уровня аутентификации и авторизации, построенного на базе Active Directory.

Для выполнения сборок Jenkins должен знать, где находится ваш сборочный инструмент. Он будет находиться ниже на странице конфигурации. Ищите слово Maven.

Конфигурация сборочного инструмента

Jenkins и без дополнительных настроек поддерживает работу с Ant и Maven (для поддержки других сборочных инструментов можно подключить соответствующие плагины). При работе с проектом java7developer мы используем Maven (для Windows), поэтому конфигурируем Jenkins так, как показано на рис. 12.3.

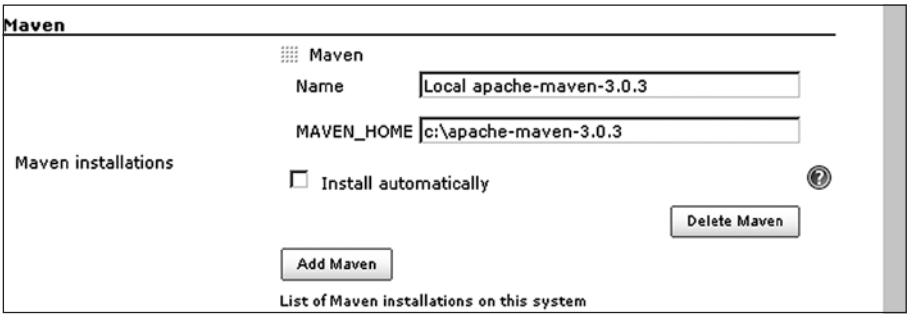


Рис. 12.3. Конфигурация сборочного инструмента Maven

Обратите внимание: в Jenkins есть команда, позволяющая автоматически установить Maven. Это удобно, если вы начинаете работу с установки на «чистой» машине.

Теперь, когда у вас сконфигурирован Maven, необходимо сообщить Jenkins, какой тип репозитория для контроля версий вы собираетесь использовать. Эта команда расположена ниже на странице конфигурации. Ищите аббревиатуру SVN (система контроля версий).

Конфигурирование контроля версий

Без дополнительных настроек Jenkins поддерживает две системы контроля версий: CVS и Subversion (SVN). Для поддержки других систем контроля версий, например Git и Mercurial, можно подключить соответствующие плагины. При работе с проектом java7developer мы используем систему SVN версии 1.6. Соответствующая конфигурация показана на рис. 12.4.

Завершив настройку этой конфигурации, нажмите кнопку Save (Сохранить) в нижней части экрана, чтобы конфигурация не потерялась.

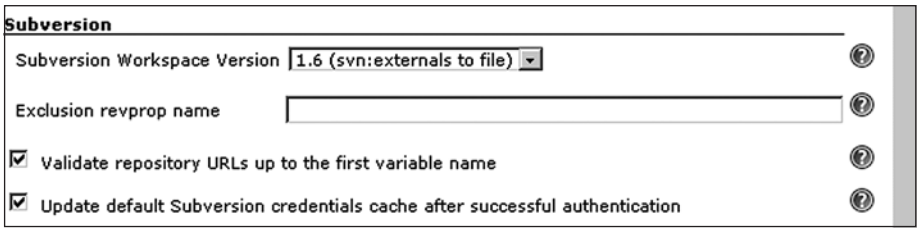


Рис. 12.4. Конфигурация системы контроля версий SVN

Итак, основные элементы Jenkins у нас сконфигурированы, займемся созданием вашей первой задачи.

12.4.2. Настройка задачи

Чтобы приступить к настройке новой задачи, вернитесь на информационную панель и щелкните в меню слева на ссылке New Job (Новая задача), чтобы перейти на предназначенную для этого страницу (http://localhost:8080/jenkins/view/All/newJob). Вам на выбор предлагается множество параметров.

Для того чтобы настроить задачу для сборки проекта java7developer, озаглавьте ее (java7developer), выполните команду Build a Maven2/3 Project (Собрать проект Maven 2/3) и нажмите кнопку OK (Готово), чтобы продолжить. Вы попадаете на конфигурационный экран, верхняя часть которого выглядит примерно как на рис. 12.5.

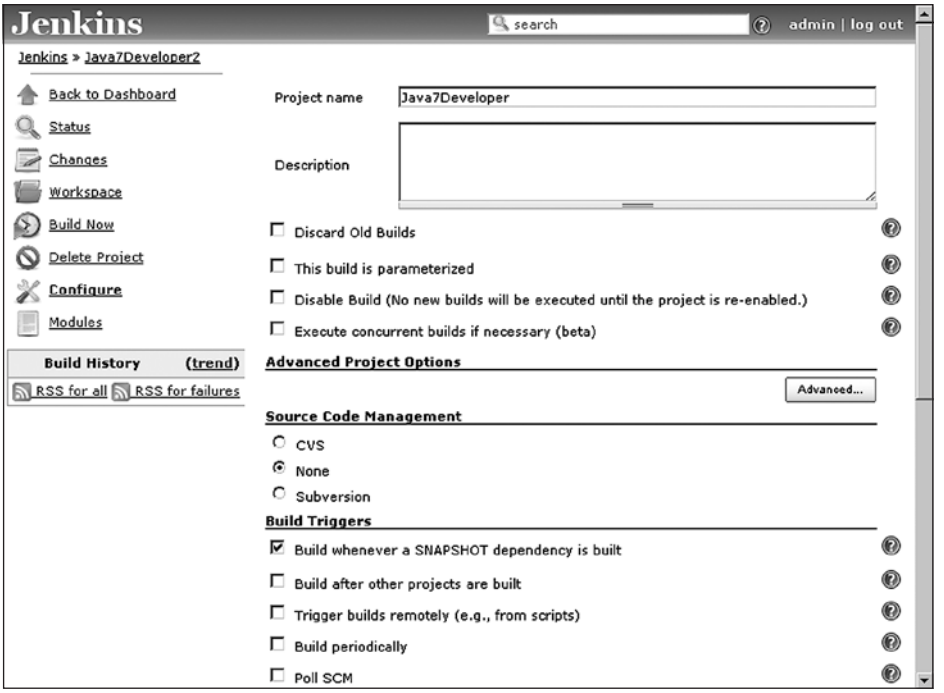


Рис. 12.5. Страница конфигурации задачи в Maven 2/3

Здесь можно заполнить несколько полей, но на начальном этапе вас особенно заинтересуют следующие разделы:

- Source Code Management (Управление исходным кодом);
- Build Triggers (Триггеры сборки);
- Build (Сборка).

Для начала укажем настройки управления исходным кодом.

Управление исходным кодом

Раздел управления исходным кодом отвечает преимущественно за указание того, на основании какой ветви контроля версий, или тега, или метки вы будете собирать исходный код. Это начало непрерывной интеграции, в ходе которой ваша команда будет систематически добавлять новый код в систему контроля версий.

При работе с проектом java7developer мы используем систему SVN и хотим собрать исходный код, начиная с главной ветки. На рис. 12.6 показаны соответствующие настройки.

Как только вы укажете Jenkins, откуда брать исходный код, нужно сконфигурировать, насколько часто Jenkins должен выполнять для вас сборки. Это делается с помощью триггеров сборки.

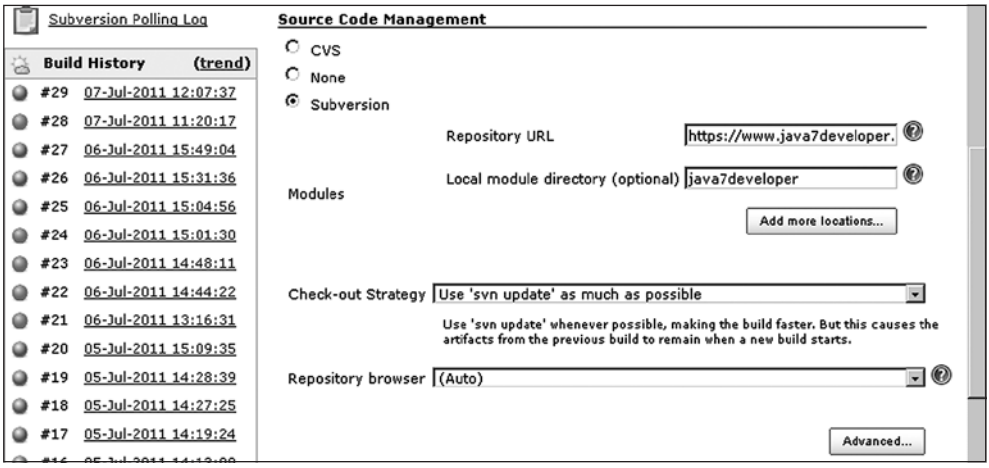


Рис. 12.6. Конфигурация для управления исходным кодом проекта java7developer

Триггеры сборки

Триггеры сборки — это те самые элементы, которые обеспечивают непрерывность непрерывной интеграции. Вы можете приказать Jenkins выполнять новую сборку, как только новая порция информации попадает в репозиторий исходного кода. Можно настроить и менее активный режим сборок — раз в день.

Для проекта java7developer мы просто прикажем Jenkins каждые 15 минут опрашивать SVN, как показано на рис. 12.7.

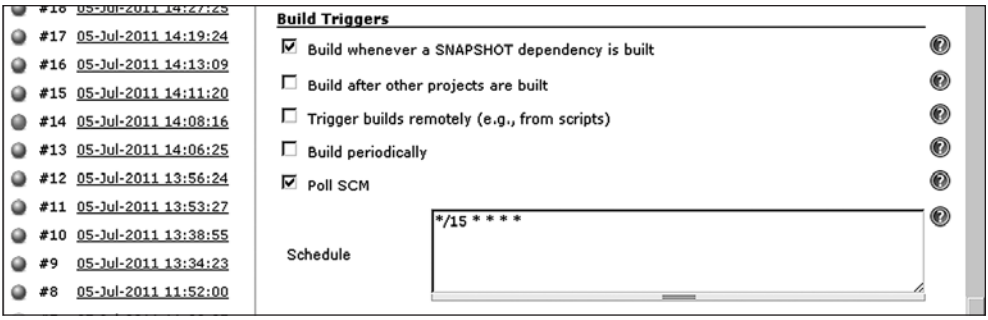


Рис. 12.7. Конфигурация триггера сборки для проекта java7developer

Чтобы выяснить, как именно в Jenkins используется то или иное поле, нужно щелкнуть на пиктограмме справки (она обозначается знаком ?). В данном случае вам может понадобиться помощь при написании cron-подобного выражения для указания периода опроса.

На данном этапе Jenkins известно, где брать код и как часто его собирать. На следующем этапе мы должны сообщить Jenkins, какие этапы жизненного цикла сборки (стадии, указанные в сборочном сценарии) следует выполнить.

Сборка

Работая с Jenkins, вы можете установить сколько угодно задач, в рамках каждой из которых будет выполняться тот или иной этап сборочного цикла. Возможно, потребуется задача, которая будет включать в себя выполнение полного набора системных интеграционных тестов каждую ночь. Но обычно задачи выполняются с большей частотой: код компилируется и тесты компонентов выполняются всякий раз, когда кто-нибудь из команды отправляет порцию кода в систему контроля версий.

В проекте java7developer мы приказываем Jenkins выполнить знакомые стадии Maven `clean` и `install`, как показано на рис. 12.8.

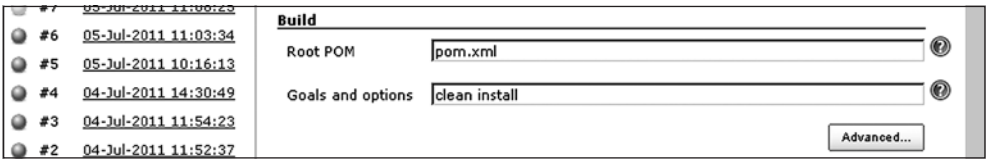


Рис. 12.8. Стадии, ожидающие выполнения в рамках жизненного цикла сборки проекта java7developer, система Maven (clean, install)

Теперь, если говорить о проекте java7developer, Jenkins располагает всей необходимой информацией, чтобы опрашивать главную ветку репозитория SVN раз в 15 минут и выполнять стадии Maven `clean` и `install`. Не забудьте нажать кнопку Save (Сохранить), чтобы сохранить вашу задачу!

Теперь мы можем вернуться к информационной панели, и на ней будет отображаться наша задача — примерно как на рис. 12.9.

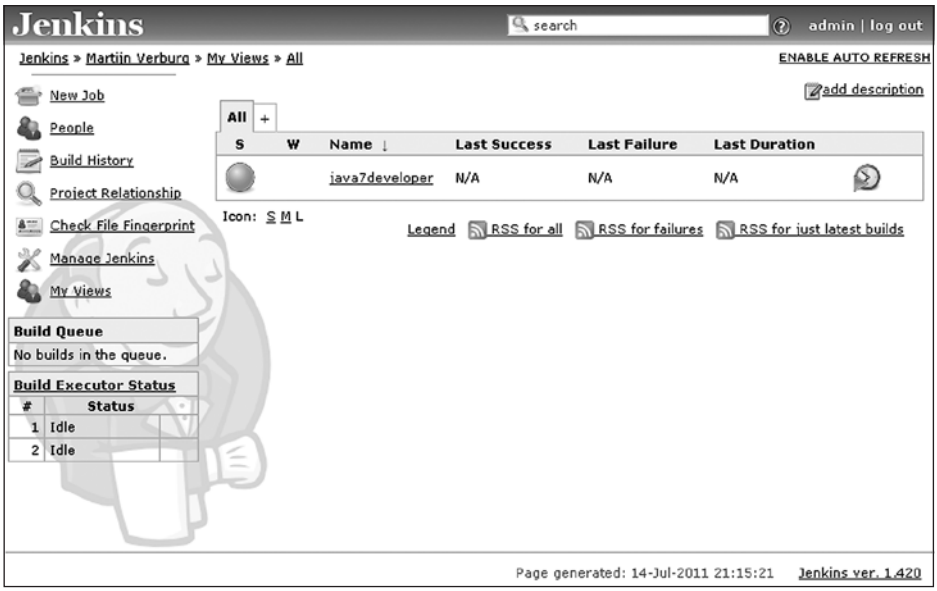


Рис. 12.9. Информационная панель с задачей java7developer

В столбце Last Success (S) (Последняя успешная операция) круглая пиктограмма представляет статус последней сборки для данной задачи. В столбце Weather (Погода) после нескольких циклов как успешных, так и провальных сборок появится пиктограмма, отражающая общее состояние проекта. Этот показатель зависит от того, как часто происходил отказ сборки, пройдены ли все тесты, а также от множества других потенциальных сценариев — смотря какие плагины вы сконфигурировали. Чтобы подробнее узнать об интерпретации этих ярлыков, можете щелкнуть на ссылке Legend (Легенда) на информационной панели (http://localhost:8080/jenkins/legend).

Теперь ваша задача готова к запуску, и вы, вероятно, хотите увидеть ее в действии! Можете подождать 15 минут до первого опроса либо просто выполнить однократную сборку.

12.4.3. Выполнение задачи

Чтобы оперативно проверить новую конфигурацию, достаточно всего лишь принудительно запустить сборку. Для этого перейдите в информационную панель, где отображается задача java7developer, нажмите для нее кнопку Schedule a Build (Назначить сборку) — это зеленая стрелка на часах рядом с полем Last Duration (Длительность последней сборки). После этого можете обновить страницу и посмотреть, как выполняется сборка.

СОВЕТ

Если щелкнуть на ссылке Enable Auto Refresh (Активизировать автоматические обновления) в правом верхнем углу информационной панели, то информационная панель будет автоматически обновляться. Вы сможете просматривать постоянно актуализируемый статус всех ваших сборок, выполняемых в настоящий момент.

Пока сборка выполняется, первая пиктограмма для задачи java7developer будет мерцать, указывая, что идет процесс сборки. Кроме того, в левой части страницы вы будете видеть параметр Build Executor Status (Статус выполнения сборки). Как только сборка завершится, вы увидите, что пиктограмма в столбце Last Success (Последняя успешная операция) стала красной, что означает провал сборки! Эта неудача объясняется отсутствием файла `build.properties`. Если вы еще не исправили эту неполадку в разделе 12.2, то можете быстро скопировать один из готовых файлов `build.properties`, предоставленных в качестве образца, и отредактировать его, чтобы из файла стояла ссылка на ваш локальный экземпляр JDK для Java 7. Вот пример из операционной системы UNIX:

```
cd $USER/.jenkins/jobs/java7developer/workspace/java7developer
cp sample_build_unix.properties build.properties
```

Теперь можно вернуться обратно на информационную панель и вновь запустить сборку вручную. На этот раз она должна пройти успешно, а на информационной панели должна отобразиться задача java7developer с голубой пиктограммой в столбце Last Success (Последняя успешная операция), что означает правильно выполненную сборку.

Еще одна характеристика сборки, которую можно сразу же проверить, — это отчет о тестах. Ведь Jenkins способен интерпретировать вывод, генерируемый Maven. Чтобы сразу перейти к результатам тестов, можете щелкнуть на ссылке в столбце Last Success (Последняя успешная операция) для задачи java7developer (http://localhost:8080/jenkins/job/java7developer/lastSuccessfulBuild/). Если перейти по ссылке Latest Test Result (Последние результаты тестов), то откроется экран с результатами, примерно как на рис. 12.10.

Рис. 12.10. Результаты тестов для успешной сборки java7developer

Все тесты пройдены, и это просто отлично! Если какой-то из них не будет пройден, то вы сможете тщательно изучить его детали.

На этом мы завершаем обсуждение запуска неуспешных и успешных сборок. При работе над проектом java7developer Jenkins и далее будет опрашивать SVN и запускать новые сборки, как только в репозиторий попадет новая порция кода.

Мы рассмотрели, как Jenkins запускает сборку, как он визуально предупреждает вас о том, что по каким-то причинам сборка не состоялась, а также как система проверяет успешность или неуспешность выполнения тестов. Но это еще далеко не все. Jenkins также может сообщать вам множество полезных параметров кода, позволяя оценить качество вашего кода.

12.5. Параметры кода в Maven и Jenkins

Язык Java и виртуальная машина Java используются довольно давно. Со временем для них были разработаны мощные инструменты и библиотеки, помогающие разработчику писать гораздо более качественный код. Мы достаточно вольно определим качественную сторону кода как *параметры кода* (code metrics) или его *статический анализ* (static code analysis). Как Maven, так и Jenkins поддерживают наиболее популярные инструменты, применяемые в данной сфере в настоящее время. Эти инструменты ориентированы в основном на работу с самим языком Java, но в наше время все больше набирают популярность и другие инструменты, обеспечивающие поддержку и для иных языков (иногда для новых языков создаются собственные специфические инструменты).

СОВЕТ

Некоторые инструменты для статического анализа кода и библиотеки также поддерживаются в современных интегрированных средах разработки (например, в Eclipse, IntelliJ и NetBeans). Не поленитесь и потратьте время на ознакомление с этой поддержкой.

Инструменты для расчета параметров кода ориентированы на исправление мелких распространенных ошибок, которые допускает практически любой разработчик. Можно считать, что такой инструмент задает минимальную планку для качества вашего кода, позволяя вам получить следующие полезные сведения:

- насколько хорошо ваш код покрывается написанными тестами[1];
- аккуратно ли отформатирован ваш код (правильное форматирование упрощает diff-сравнения и повышает удобочитаемость кода);
- насколько вероятно, что в вашем коде возникнет исключение NPE;
- не забыли ли вы переопределить методы `equals()` и `hashCode()` для объекта предметной области.

Список проверок, выполняемых различными инструментами, довольно велик, и каждой команде разработчиков приходится самостоятельно решать, какие проверки будут проводиться в конкретном проекте.

[1] Мы не будем рассматривать в этой книге распространенные инструменты для тестового покрытия кода, так как они пока несовместимы с Java 7.

О ЧЕМ НЕ ПОЗВОЛЯЮТ СУДИТЬ ПАРАМЕТРЫ КОДА

Члены отдельных команд искренне полагают, что их код почти идеален, так как они исправно решают все проблемы, о которых становится известно при отслеживании инструментов с параметрами кода. Это ошибка. Предупреждения, выдаваемые такими инструментами, безусловно, полезны, так как спасают вас от множества низкоуровневых ошибок и в целом улучшают практику написания кода. Но они не гарантируют высокого качества и никак не влияют на то, насколько верно или неверно вы запрограммируете вашу бизнес-логику!

Еще один опасный соблазн заключается в том, что менеджеры любят указывать эти параметры в своих отчетах. Окажите услугу и менеджерам, и себе — постарайтесь, чтобы эти параметры не выходили за пределы команды разработчиков. Они совершенно непригодны для оценки качества управления проектами.

Сочетая в работе использование Maven и Jenkins, вы получаете качественную информацию о характеристиках кода — как обзорную, так и достаточно подробную. В этом разделе вы научитесь двум основным вещам:

- устанавливать и конфигурировать плагины Jenkins;
- конфигурировать плагины, обеспечивающие согласованность кода (Checkstyle) и поиск ошибок (FindBugs).

Эти операции мы также будем рассматривать на примере проекта java7developer. Для начала разберем, как устанавливаются плагины для Jenkins — они необходимы для реализации функций отчетности о параметрах кода.

12.5.1. Установка плагинов Jenkins

Устанавливать плагины Jenkins совсем не сложно, так как этот сервер предоставляет удобный менеджер, оснащенный пользовательским интерфейсом, помогающий как при скачивании, так и при установке плагинов. Перед установкой плагинов Jenkins необходимо перезапустить. Поэтому для начала откройте страницу http://localhost:8080/jenkins/manage и перейдите по ссылке Prepare to Shutdown (Подготовка к остановке). Так вы приостановите все задачи, которые в настоящий момент должны выполняться, сможете спокойно установить нужные плагины и перезапустить Jenkins.

После того как Jenkins будет подготовлен к остановке, познакомимся с менеджером плагинов. На странице управления щелкните на ссылке Manage Plugins (Управление плагинами) (http://localhost:8080/jenkins/pluginManager/). Должен открыться примерно такой экран, как на рис. 12.11.

Начинаем работу с вкладки Updates (Обновления). Перейдите на вкладку Available (Доступные) — и увидите длинный список доступных плагинов. В контексте этой главы нас заинтересуют следующие плагины, рядом с которыми нужно поставить флажки:

- Checkstyle;
- FindBugs.

Далее перейдите в нижнюю часть экрана и нажмите кнопку Install (Установить), чтобы начать установку. Когда установка будет завершена, можно перезапустить Jenkins — для этого щелкните на ссылке http://localhost:8080/jenkins/restart.

Рис. 12.11. Менеджер плагинов Jenkins

Jenkins перезапустится уже со всеми установленными плагинами. Теперь сконфигурируем их и начнем с плагина Checkstyle.

12.5.2. Обеспечение согласованности кода с помощью плагина Checkstyle

Checkstyle — это инструмент для статического анализа кода Java, проверяющий, как скомпонован ваш код, есть ли у вас все нужные уровни Javadocs, а также уточняющий другие детали, связанные с синтаксическим сахаром. Он также проверяет код на наличие типичных ошибок программиста, но для решения последней задачи лучше подходит инструмент FindBugs.

Важно пользоваться Checkstyle по нескольким причинам. Во-первых, этот анализатор помогает соблюдать минимальный набор стилистических правил по написанию кода, и члены команды могут без труда читать код коллег (кстати, одна из основных причин популярности Java — удобство чтения кода на этом языке). Во-вторых, при соблюдении единообразия пробелов между элементами кода и размещении этих элементов становится значительно удобнее работать с diff-сравнениями и патчами.

Плагин Checkstyle сконфигурирован в файле Maven `pom.xml`, поэтому вам остается просто изменить задачу java7developer, добавив к ней стадию `checkstyle:check-style`. Чтобы сконфигурировать эту задачу, достаточно щелкнуть на ссылке java7developer в задаче, отображаемой на информационной панели, а потом перейти на следующий экран и щелкнуть на ссылке Configure (Сконфигурировать) в меню слева.

Далее займемся конфигурацией отчета, а также определим, должна ли сборка обрываться, если в ней будет выявлено слишком много нарушений. На рис. 12.12 показаны конфигурация сборки Maven и отчет, который мы используем при работе над проектом java7developer.

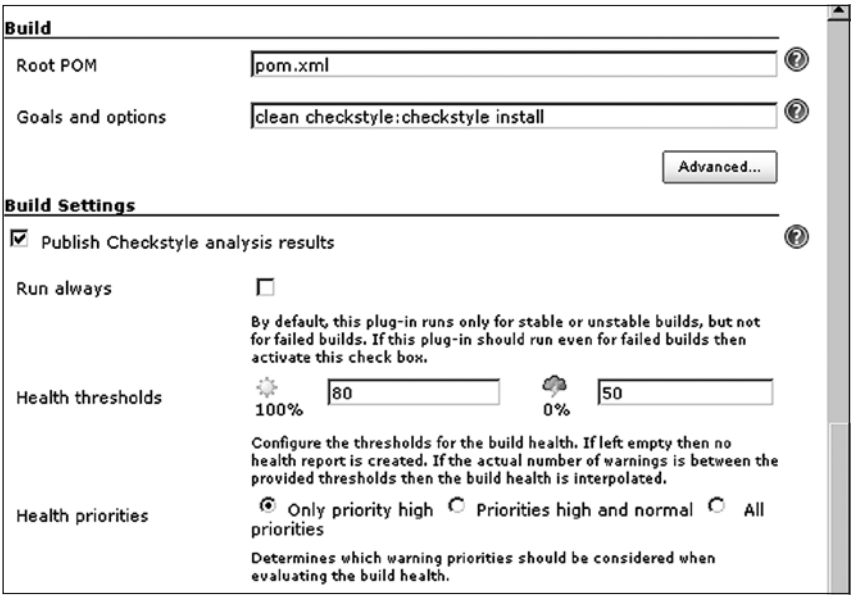


Рис. 12.12. Конфигурация Checkstyle

Не забудьте нажать кнопку Save (Сохранить), чтобы сохранить эту конфигурацию! По умолчанию в Checkstyle применяется еще тот набор соглашений программирования, которые разработала компания Sun Microsystems для языка Java. Checkstyle можно значительно перенастроить так, чтобы соблюдались соглашения по программированию, актуальные именно в вашей команде.

ВНИМАНИЕ

Возможно, наиболее актуальная версия Checkstyle пока не полностью поддерживает синтаксис Java 7, поэтому вы можете столкнуться со случаями ложной тревоги, особенно при обработке конструкций try-with-resources, оператора ромбовидного синтаксиса и других синтаксических элементов, объединяемых проектом «Монета».

Проверим, насколько проект java7developer согласуется с заданным по умолчанию набором правил. Мы уже помним, что для этого нужно вернуться на информационную панель Jenkins и вручную выполнить сборку. Как только сборка завершится, мы вернемся к странице с последней успешной сборкой (как вы помните, туда мы попадаем через ссылку в столбце Last Success (Последняя успешная операция)). На этой странице в меню слева щелкаем на ссылке Checkstyle Warnings (Предупреждения Checkstyle) и открываем новую страницу, где показан отчет Checkstyle. В случае с проектом java7developer вы должны увидеть примерно такой экран, как показан на рис. 12.13.

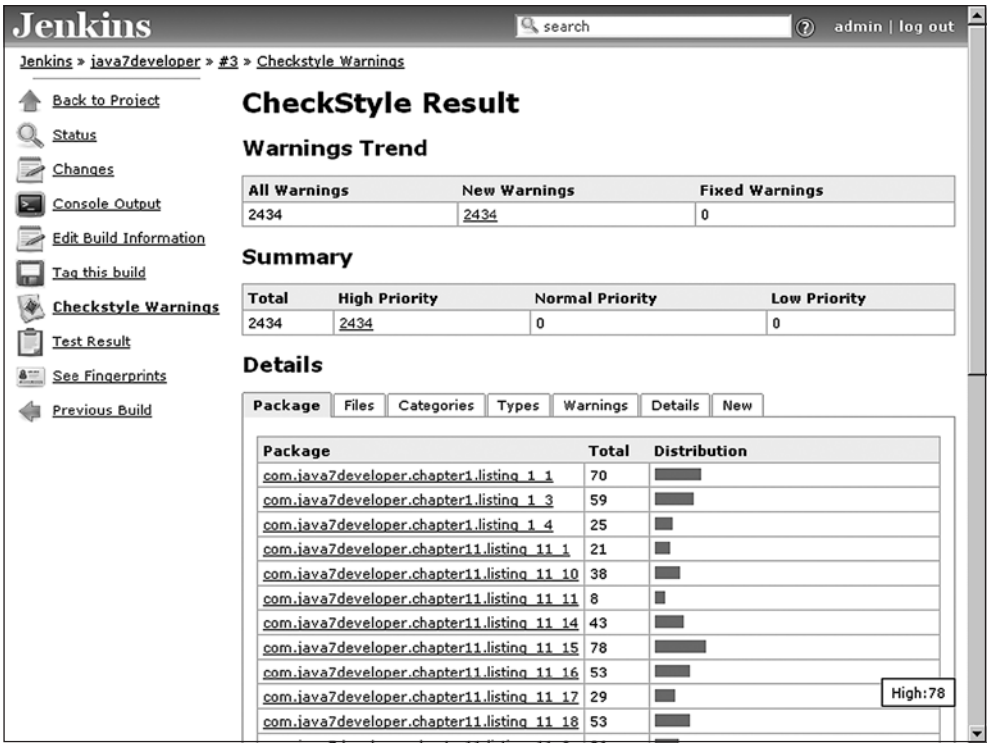


Рис. 12.13. Отчет Checkstyle

Как видите, в базе кода java7developer есть несколько актуальных предупреждений. Полюбуйтесь, сколько еще работы предстоит сделать! Можете подробно изучить каждое из предупреждений и разобраться, почему возникло то или иное нарушение. Попробуйте исправить эти нарушения до того, как начнется следующий сборочный цикл.

Инструмент Checkstyle, безусловно, полезен, но он не «заточен» под поиск ошибок в коде. Обнаружение ошибок — важная задача, и для этого лучше использовать специальный плагин FindBugs.

12.5.3. Обеспечение качества кода с помощью FindBugs

Программа FindBugs (ее автором является Билл Пью (Bill Pugh)) — это инструмент для анализа байт-кода, позволяющий отыскивать в вашем коде потенциальные ошибки. Поскольку инструмент работает именно на уровне байт-кода, это означает, что он пригоден и для проверки кода Scala и Groovy. Но правила, в соответствии с которыми работает инструмент, ориентированы прежде всего на поиск ошибок в коде Java, поэтому в коде Scala и Groovy велика вероятность возникновения ложной тревоги.

Инструмент FindBugs создан на базе множества исследований, выполненных разработчиками, очень хорошо знающими и понимающими язык Java. FindBugs позволяет идентифицировать нежелательные ситуации, в которых:

- код приводит к исключению NPE;
- происходит присваивание значения переменной, которая никогда не используется;
- при сравнении строковых объектов `String` используется оператор ==, а не метод `equals`;
- в цикле применяется обычная конкатенация `+ String` (а не буфер `StringBuffer`).

Целесообразно сначала запускать FindBugs со стандартными настройками, а лишь затем дополнительно настраивать его, указывая, проверка соблюдения каких правил вас наиболее интересует.

ВНИМАНИЕ

Иногда FindBugs дает «ложноположительные» результаты даже в языке Java! Все предупреждения следует внимательно изучать, и если их можно игнорировать, то стоит специально исключить проверку конкретных практических ситуаций.

FindBugs важно использовать по нескольким причинам. Во-первых, он прививает разработчику хорошие навыки, работая как программист-напарник (по крайней мере в том, что касается поиска потенциальных ошибок). Во-вторых, повышается общее качество кода в проекте и ваша система отслеживания ошибок будет не так сильно наполняться надоедливыми ошибками — команда сможет освободить время для решения существенных проблем, например на корректирование бизнес-логики.

Как и при работе с плагином Checkstyle, можно сконфигурировать задачу, просто щелкнув на ссылке java7developer в задаче, отображаемой на информационной панели. На следующем экране понадобится щелкнуть на ссылке Configure (Сконфигурировать) в меню слева.

Чтобы выполнить плагин FindBugs, нужно добавить к сборочной команде Maven в Jenkins стадии `compile findbugs:findbugs` (`compile` требуется для того, чтобы FindBugs мог работать с байт-кодом).

Вы также можете сконфигурировать отчет и установить, должна ли сборка обрываться, если в ней будет зафиксировано слишком много нарушений. Такая конфигурация показана на рис. 12.14.

Не забудьте нажать кнопку Save (Сохранить), чтобы сохранить эту конфигурацию! FindBugs приступает к работе, опираясь на стандартный набор правил, который можно значительно перенастраивать, чтобы система поиска ошибок в точности учитывала правила программирования, принятые в вашей команде. Посмотрим, насколько проект java7developer соответствует стандартному набору правил.

Как обычно, можно вернуться на информационную панель Jenkins и вручную выполнить сборку. Как только сделаете это, можете перейти на страницу последней успешной сборки (напоминаем, соответствующая ссылка находится в столбце Last Success (Последняя успешная операция)) и щелкнуть на ссылке FindBugs Warnings (Предупреждения FindBugs) в меню слева. Так вы попадете на страницы с отчетом. На рис. 12.15 показан отчет, напоминающий тот, который должен получиться у вас по проекту java7developer.

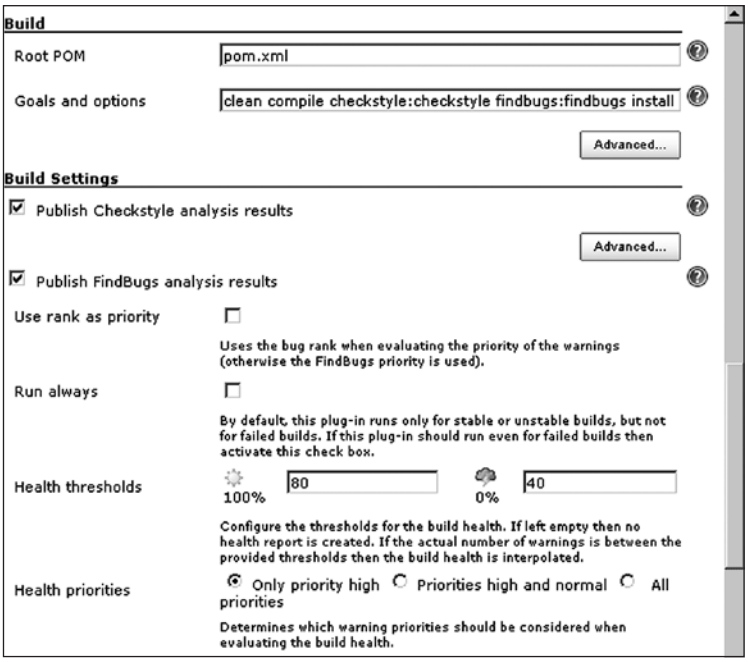


Рис. 12.14. Конфигурация FindBugs

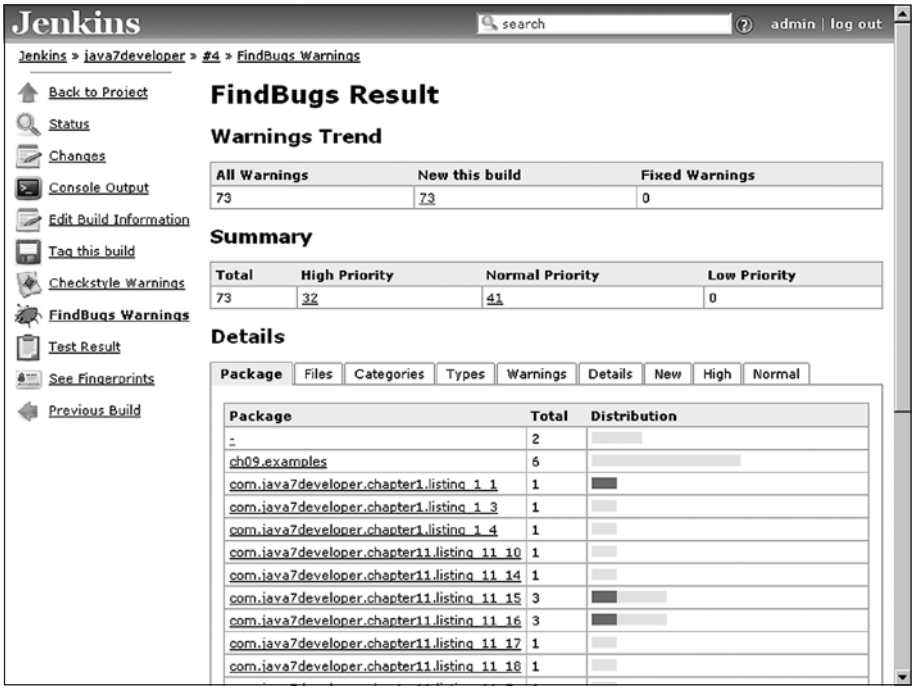


Рис. 12.15. Отчет FindBugs

Как видите, база кода java7developer содержит актуальные предупреждения. Пожалуй, авторов этой книги не всегда можно назвать безупречными программистами. Можете подробно изучить ве предупреждения и разобраться, почему возникло то или иное нарушение. Постарайтесь исправить все ошибки перед следующим сборочным циклом (если, конечно, вам это интересно).

FindBugs способен обнаруживать подавляющее большинство применяемых в Java трюков и ошибок программирования. По мере того как команда разработчиков будет учиться на этих ошибках, количество предупреждений в отчетах будет снижаться. Вы не только улучшите качество вашего кода, но и повысите собственные навыки программирования!

На этом мы завершаем раздел о Jenkins, Maven и параметрах кода. В этой области уже существует довольно солидный инструментарий (правда, поддержка Scala и Groovy пока далека от идеала), и вы сможете без труда организовать работу. Если вы — энтузиаст непрерывной интеграции и хотите в полной мере оценить потенциал Jenkins, настоятельно рекомендуем вам почитать постоянно обновляемую книгу Джона Смарта (John Smart) *Jenkins: The Definitive Guide* («Jenkins: подробное руководство») (издательство O'Reilly). Но вы, вероятно, заметили, что у нас остается нерассмотренным еще один раздел нашей темы, связанный с многоязычным программированием на JVM, сборками и непрерывной интеграцией. Мы еще не затрагивали работу с проектами на языке Clojure. К счастью, в сообществе Clojure уже создано несколько сборочных инструментов, «заточенных» под обработку проектов именно на этом языке. Эти инструменты довольно широко используются. Один из наиболее популярных называется Leiningen и написан как раз на языке Clojure.

12.6. Leiningen

Как вы уже знаете, сборочный инструмент должен предоставлять несколько возможностей, чтобы быть максимально полезным для разработчика. Основные возможности таковы:

- управление зависимостями;
- компиляция;
- автоматизация тестов;
- упаковка для последующего развертывания.

Leiningen исходит из того, что их лучше разграничивать. Он переиспользует имеющуюся технологию Java для предоставления всех этих возможностей, но таким образом, что система не зависит при этом от какого-то отдельно взятого общего пакета функций.

Формулировка может показаться слишком сложной и даже страшноватой, но на практике вы как разработчик не ощущаете этой сложности. На самом деле с Leiningen могут управиться даже те программисты, которые не имеют опыта работы с базовыми инструментами Java. Начнем с установки Leiningen — процесс автоматической загрузки очень прост. Затем мы изучим компоненты Leiningen

и его общую архитектуру, после чего наконец опробуем эту систему на элементарном проекте Hello World.

Мы покажем, как начать новый проект, добавить зависимость и работать с ней в интерактивной среде REPL, предоставляемой Clojure. Естественно, мы уделим внимание и тому, как в Clojure организуется разработка через тестирование с применением Leiningen. В завершение главы рассмотрим, как следует упаковать код для развертывания в виде приложения или библиотеки, чтобы им могли пользоваться другие.

Итак, приступим к работе с Leiningen.

12.6.1. Знакомство с Leiningen

Начать работу с Leiningen совсем не сложно. В UNIX-подобных системах (в частности, в Linux и Mac OS X) для начала берем сценарий `lein`. Он предоставляется на GitHub (откройте страницу https://github.com/ и поищите по запросу Leiningen, то же самое можно сделать и через поисковик).

Как только сценарий `lein` указан в пути `PATH`, он становится исполняемым и его можно запускать. При первом запуске `lein` обнаружит, какие зависимости требуется установить (а какие уже имеются). Кроме того, он установит все необходимые компоненты, не входящие в состав ядра Leiningen. Поэтому первый запуск может протекать немного медленнее, чем последующие, так как при дальнейших запусках устанавливать зависимости уже не придется.

В следующем разделе мы расскажем об архитектуре Leiningen и тех технологиях Java, на которые этот инструмент опирается для предоставления основного функционала.

УСТАНОВКА LEININGEN ДЛЯ WINDOWS

С точки зрения старого хакера-юниксоида, одна из наиболее досадных черт Windows заключается в том, что в системе нет простых стандартных инструментов, с которыми удобно работать в командной строке. Например, в классической сборке Windows отсутствуют утилиты `curl` или `wget`, позволяющие скачивать файлы по протоколу HTTP (ведь Leiningen приходится скачивать JAR-файлы из репозиториев Maven, и эти утилиты как раз бы пригодились). Чтобы справиться с этой проблемой, можно установить Leiningen для Windows. Он предоставляется в виде ZIP-архива, содержащего командный файл `lein.bat`, а также готовый исполняемый файл `wget.exe`, который нужно поместить в каталог `PATH` вашей системы Windows, чтобы команда `self-install` сценария `lein` работала правильно.

12.6.2. Архитектура Leiningen

Как было указано выше, инструмент Leiningen заключает в себе некоторые классические технологии Java и, упрощая их, реализует на их основе свои возможности. Основные компоненты, входящие в состав Leiningen, — Maven (версия 2), Ant и `javac`.

На рис. 12.16 показано, что Maven используется для разрешения зависимостей и управления ими, а `javac` и Ant применяются для осуществления самой сборки, запуска тестов и обеспечения других шагов сборочного процесса.

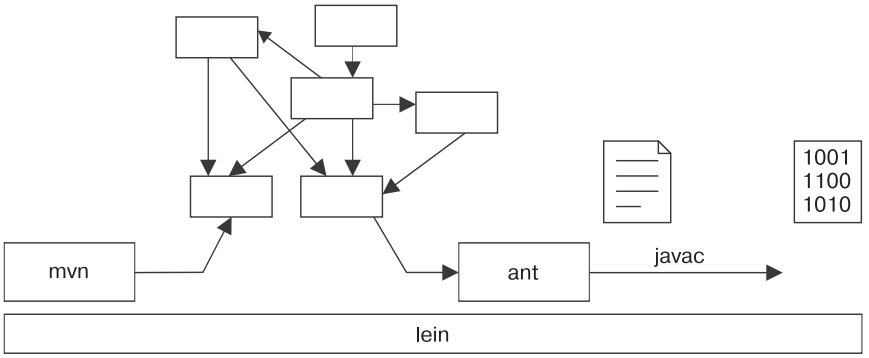


Рис. 12.16. Leiningen и его компоненты

Такой метод позволяет опытному пользователю проникнуть через абстракцию, выстраиваемую Leiningen, и на всех этапах получать полномасштабный доступ к базовым инструментам. Но базовый синтаксис элементарен, приступить к использованию этих базовых инструментов можно без всякого опыта.

На простом примере рассмотрим, как действует синтаксис файла `project.clj`, и изучим базовые команды, используемые в ходе жизненного цикла проекта Leiningen.

12.6.3. Пример: Hello Lein

После того как `lein` будет указан в пути к файлу, начнем новый проект. Для этого воспользуемся командой `lein new`:

```
ariel:projects boxcat$ lein new hello-lein
Created new project in: /Users/boxcat/projects/hello-lein
ariel:projects boxcat$ cd hello-lein/
ariel:hello-lein boxcat$ ls
README project.clj src test
```

Эта команда создает проект под названием hello-lein. В нем есть каталог, содержащий простой файл описания `README`, файл `project.clj` (мы обсудим его подробнее чуть ниже), а также параллельные каталоги src и test.

Если импортировать в Eclipse проект, только что созданный Leiningen (например, при установленном в Eclipse плагине CounterClockwise, предназначенном для работы с Clojure), то компоновка проекта будет выглядеть примерно как на рис. 12.17.

Структура этого проекта в точности соответствует простой структуре проектов Java — мы видим параллельные структуры src и test, в каждой из которых присутствует файл `core.clj` (для кода верхнего уровня и тестов соответственно). Другой важный файл — `project.clj`, используемый Leiningen для управления сборкой и содержания метаданных.

Рассмотрим базовый файл, сгенерированный командой `lein new`.

```
(defproject hello-lein "1.0.0-SNAPSHOT"
  :description "FIXME: write description"
  :dependencies [[org.clojure/clojure "1.2.1"]])
```

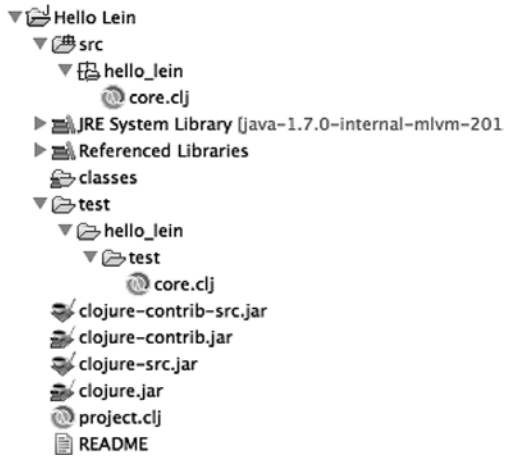


Рис. 12.17. Новоиспеченный проект Leiningen

Эта форма Clojure относительно проста для синтаксического анализа — здесь есть макрос `(defproject)`, который создает новые значения, представляющие проекты Leiningen. Этому макросу необходимо сообщить, как называется проект — в данном случае hello-lein. Мы также сообщаем макросу номер версии этого проекта — по умолчанию задается `1.0.0-SNAPSHOT` (номер задан по правилам Maven, описанным в подразделе 12.3.1). Наконец, мы предоставляем словарь метаданных, описывающих проект.

Без дополнительных настроек `lein` предоставляет два фрагмента метаданных: строку с описанием и вектор зависимостей, в который удобно добавлять новые зависимости. Добавим зависимость — это будет библиотека `clj-time`. Она служит интерфейсом для использования удобной библиотеки Java Joda-Time, предназначенной для управления датой и временем. Правда, чтобы понять этот пример, вам не обязательно разбираться в названной библиотеке. После того как вы добавите новую зависимость, файл `project.clj` будет выглядеть вот так:

```
(defproject hello-lein "1.0.0-SNAPSHOT"
  :description "FIXME: write description"
  :dependencies [[org.clojure/clojure "1.2.1"]
                 [clj-time "0.3.0"]])
```

Второй элемент вектора, описывающий новую зависимость, — это версия библиотеки, которую предполагается использовать. Именно эта версия должна быть получена из репозитория, если Leiningen не найдет нужной копии в своем локальном репозитории зависимостей.

По умолчанию Leiningen обращается для получения недостающих библиотек к репозиторию, расположенному по адресу http://clojars.org/. Поскольку на внутрисистемном уровне Leiningen использует Maven, это, в сущности, лишь репозиторий Maven. Clojars предоставляет специальный поисковый инструмент, с которым удобно работать, если вы знаете, какие библиотеки вам нужны, но не знаете необходимых версий.

Имея новую зависимость, нужно обновить окружение локальной сборки. Это делается с помощью команды `lein deps`:

```
ariel:hello-lein boxcat$ lein deps
Downloading: clj-time/clj-time/0.3.0/clj-time-0.3.0.pom from central
Downloading: clj-time/clj-time/0.3.0/clj-time-0.3.0.pom from clojure
Downloading: clj-time/clj-time/0.3.0/clj-time-0.3.0.pom from clojars
Transferring 2K from clojars
Downloading: joda-time/joda-time/1.6/joda-time-1.6.pom from clojure
Downloading: joda-time/joda-time/1.6/joda-time-1.6.pom from clojars
Transferring 5K from clojars
Downloading: clj-time/clj-time/0.3.0/clj-time-0.3.0.jar from central
Downloading: clj-time/clj-time/0.3.0/clj-time-0.3.0.jar from clojure
Downloading: clj-time/clj-time/0.3.0/clj-time-0.3.0.jar from clojars
Transferring 7K from clojars
Downloading: joda-time/joda-time/1.6/joda-time-1.6.jar from clojure
Downloading: joda-time/joda-time/1.6/joda-time-1.6.jar from clojars
Transferring 522K from clojars
Copying 4 files to /Users/boxcat/projects/hello-lein/lib
ariel:hello-lein boxcat$
```

Leiningen выкачал через Maven не только интерфейс для работы с Clojure, но и базовый JAR-файл библиотеки Joda-Time. Воспользуемся этими элементами в коде и покажем, как применять Leiningen в качестве интерактивной REPL-среды для разработки в присутствии зависимостей.

Необходимо изменить основной файл исходников `src/hello_lein/core.clj` вот так:

```
(ns hello-lein.core)

(use '[clj-time.core :only (date-time)])

(defn isodate-to-millis-since-epoch [x]
  (.getMillis (apply date-time
➥ (map #(Integer/parseInt %) (.split x "-")))))
```

В результате получаем функцию Clojure, преобразующую стандартную дату ISO (в формате `ГГГГ-ММ-ДД`) в количество миллисекунд, истекших с начала эпохи UNIX.

Протестируем этот код в стиле REPL, воспользовавшись Leiningen. Сначала нужно добавить дополнительную строку в файл `project.clj`, чтобы он принял следующий вид:

```
(defproject hello-lein "1.0.0-SNAPSHOT"
  :description "FIXME: write description"
  :dependencies [[org.clojure/clojure "1.2.1"]
                 [clj-time "0.3.0"]]
  :repl-init hello-lein.core)
```

Имея эту строку, мы уже можем открыть REPL с полным набором нужных зависимостей. Среда REPL добавляет в область видимости функции из пространства имен `hello-lein.core`:

```
ariel:hello-lein boxcat$ lein repl
REPL started; server listening on localhost:10886.

hello-lein.core=> (isodate-to-millis-since-epoch "1970-01-02")
86400000
hello-lein.core=>
```

Здесь указано точное количество миллисекунд, содержащееся в сутках. На данном примере мы можем оценить основной принцип работы с REPL в реальном проекте. Но расширим этот образец и рассмотрим очень мощный способ разработки через тестирование в среде REPL системы Leiningen.

12.6.4. REPL-ориентированная разработка через тестирование с применением Leiningen

В основе любой хорошей разновидности разработки через тестирование должен лежать простой базовый цикл, который можно использовать для создания новой функциональности. При применении Clojure и Leiningen подобный простой цикл может быть построен следующим образом.

1. Добавляем любые требуемые новые зависимости (и перезапускаем `lein deps`).
2. Запускаем среду REPL (`lein repl`).
3. Проектируем новую функцию и добавляем ее в область видимости в пределах REPL.
4. Тестируем функцию внутри REPL.
5. Повторяем шаги 3 и 4 до тех пор, пока функция не станет работать правильно.
6. Добавляем окончательный вариант функции в соответствующий файл `.clj`.
7. Добавляем тестовые варианты, которые проверяем на файлах `.clj` из набора для тестирования.
8. Перезапускаем REPL и повторяем процесс, начиная с шага 3 (или 1, если вам требуются новые зависимости).

Разумеется, такая работа — вариант разработки через тестирование. Но при разработке с применением REPL нам не приходится отвечать на вопрос о том, что пишется раньше — тесты или код. В стиле REPL и код и тесты пишутся одновременно.

На этапе 8 мы перезапускаем среду REPL при добавлении в код проекта новоиспеченной функции. Это делается для того, чтобы гарантировать свободную компиляцию новой функции. Иногда при создании новой функции мы, чтобы обеспечить ее поддержку, вносим небольшие изменения в другие функции или

в окружение. При переносе функции в постоянный код об изменении таких мелких деталей легко забыть. Перезапуская REPL, мы можем заблаговременно отследить такие забытые изменения, если они вдруг имеют место.

Итак, процесс прост и ясен, но нам остается ответить еще на один вопрос, которого мы пока не касались ни здесь, ни в главе 11, посвященной разработке через тестирование. К счастью, это совсем не сложно. Рассмотрим шаблон, предоставляемый `lein new` при создании нового проекта:

```
(ns hello-lein.test.core
  (:use [hello-lein.core])
  (:use [clojure.test]))

(deftest replace-me ;; FIXME: write
  (is false "No tests have been written."))
```

При запуске тестов используется команда `lein test`. Применим ее к автоматически сгенерированному тестовому варианту и посмотрим, что получится (хотя вы уже, вероятно, догадываетесь).

```
ariel:hello-lein boxcat$ lein test
Testing hello-lein.test.core
FAIL in (replace-me) (core.clj:6)
No tests have been written.
expected: false
  actual: false
Ran 1 tests containing 1 assertions.
1 failures, 0 errors.
```

Как видите, в данной ситуации тест оказывается неуспешным, и у вас, наверное, уже руки чешутся самостоятельно написать несколько тестов. Сделаем это, запишем в файл `core.clj` в каталоге для тестов следующую информацию:

```
(ns hello-lein.test.core
  (:use [hello-lein.core])
  (:use [clojure.test]))

(deftest one-day
  (is true
    (= 86400000 (isodate-to-millis-since-epoch "1970-01-02"))))
```

Структура теста очень проста — вы используете макрос `(deftest)`, даете тесту имя `(one-day)` и предоставляете форму, очень напоминающую оператор контроля.

Структура кода Clojure такова, что форму `(is)` можно читать совершенно естественным образом — почти как в языке предметной области. Фактически тест формулируется следующим образом: «Верно ли, что число 86 400 000 равно количеству миллисекунд, истекших с 1 февраля 1970 года?» Рассмотрим этот тест на практике:

```
ariel:hello-lein boxcat$ lein test
Testing hello-lein.test.core
Ran 1 tests containing 1 assertions.
0 failures, 0 errors.
```

Основной пакет в данном случае — `clojure.test`. В нем предоставляется несколько других полезных форм для выстраивания тестовых вариантов. При применении некоторых вариантов требуется задействовать более сложные среды или тестовые конструкции. Разработка через тестирование на языке Clojure подробно рассмотрена в книге *Clojure in Action* Амита Ратхора (издательство Manning, 2011).

Организовав разработку через тестирование с применением интерактивной среды REPL, вы можете написать на Clojure основательное приложение и поработать с ним. Но рано или поздно наступит ситуация, в которой вам потребуется поделиться с командой своим кодом, чтобы совместно его обрабатывать. К счастью, в Leiningen есть несколько команд, упрощающих упаковку и развертывание кода.

12.6.5. Упаковка и развертывание кода с помощью Leiningen

В Leiningen предусмотрено два основных способа распространения готового кода. В принципе, можно сказать, что первый способ подразумевает использование зависимостей, а второй — нет. Для этого применяются команды `lein jar` и `lein uberjar` соответственно.

Рассмотрим `lein jar` на практике:

```
ariel:hello-lein boxcat$ lein jar
Copying 4 files to /Users/boxcat/projects/hello-lein/lib
Created /Users/boxcat/projects/hello-lein/hello-lein-1.0.0-SNAPSHOT.jar
```

А вот что будет записано в результирующем JAR-файле:

```
ariel:hello-lein boxcat$ jar tvf hello-lein-1.0.0-SNAPSHOT.jar
    72 Sat Jul 16 13:38:00 BST 2011 META-INF/MANIFEST.MF
  1424 Sat Jul 16 13:38:00 BST 2011 META-INF/maven/hello-lein/hello-lein/
     pom.xml
   105 Sat Jul 16 13:38:00 BST 2011
META-INF/maven/hello-lein/hello-lein/pom.properties
   196 Fri Jul 15 21:52:12 BST 2011 project.clj
   238 Fri Jul 15 21:40:06 BST 2011 hello_lein/core.clj
ariel:hello-lein boxcat$
```

Очевидная характерная особенность этого процесса заключается в том, что базовые команды Leiningen на выходе дают готовые к распространению файлы Clojure, а не CLASS-файлы. Это обычная ситуация для кода на Lisp, поскольку макросам и компонентам системы, работающим в реальном времени, будет затруднительно обращаться со скомпилированным кодом.

Теперь посмотрим, что происходит, когда задействуется команда `lein uberjar`. Мы должны получить файл JAR, в котором содержится не только наш код, но и необходимые зависимости.

```
ariel:hello-lein boxcat$ lein uberjar
Cleaning up.
Copying 4 files to /Users/boxcat/projects/hello-lein/lib
Copying 4 files to /Users/boxcat/projects/hello-lein/lib
```

```
Created /Users/boxcat/projects/hello-lein/hello-lein-1.0.0-SNAPSHOT.jar
Including hello-lein-1.0.0-SNAPSHOT.jar
Including clj-time-0.3.0.jar
Including clojure-1.2.1.jar
Including clojure-contrib-1.2.0.jar
Including joda-time-1.6.jar
Created /Users/boxcat/projects/hello-lein/
➥ hello-lein-1.0.0-SNAPSHOT-standalone.jar
```

Действительно, получается файл JAR с нашим кодом и необходимыми зависимостями, а также вторичными зависимостями, относящимися к нашим. Такая структура называется *транзитивным замыканием вашего графа зависимостей*. Итак, ваш код полностью упакован и готов для автономного использования.

Разумеется, это также означает, что результат выполнения `lein uberjar` будет гораздо больше, чем результат `lein jar`, поскольку вам потребуется упаковать все эти зависимости. Даже в случае с простым примером, который мы рассматриваем здесь, разница довольно заметная:

```
ariel:hello-lein boxcat$ ls -lh h*.jar
-rw-r--r-- 1 boxcat staff 4.1M 16 Jul 13:46
hello-lein-1.0.0-SNAPSHOT-standalone.jar
-rw-r--r-- 1 boxcat staff 1.7K 16 Jul 13:46
hello-lein-1.0.0-SNAPSHOT.jar
```

Чтобы ориентироваться в командах `lein jar` и `lein uberjar`, можно подходить к ним следующим образом: `lein jar` применяется при создании библиотеки, которую вы выстраиваете на основе иных библиотек и которую другие разработчики могут использовать в своих приложениях либо писать приложения на ее основе. Если же вы пишете на Clojure приложение для конечного пользователя, а не расширение (подразумевается типичный пользователь), то нужно задействовать `lein uberjar`.

Итак, мы рассмотрели, как использовать Leiningen для запуска проектов Clojure, управления ими, сборки и развертывания готовых программ. В Leiningen присутствует еще множество полезных встроенных команд и мощная система плагинов, обеспечивающая обширную настройку инструмента с учетом конкретной ситуации. Чтобы более полно оценить возможности Leiningen, просто вызовите его без команд, вот так: `lein`.

В следующей главе мы вновь затронем Leiningen, когда будем писать веб-приложение на языке Clojure.

12.7. Резюме

О качестве любого проекта, в котором занят основательный Java-разработчик, свидетельствует наличие быстрой, легко воспроизводимой и при этом простой сборки. Если вы не можете быстро собирать ваши программы в достаточно согласованном виде, то будете впустую тратить множество (собственных) денег и времени.

Понимание базового жизненного цикла сборки (компиляция — тестирование — упаковка) — ключевой момент организации качественного сборочного

процесса. Если уж на то пошло, вы не сможете протестировать код, который еще не скомпилирован!

Инструмент Maven берет концепцию жизненного цикла сборки и развивает ее до жизненного цикла всего проекта. Такой цикл может согласованно использоваться во всех проектах Maven. Метод программирования по соглашениям очень удобен при работе больших команд разработчиков, но некоторые проекты могут потребовать лучшей гибкости.

Maven также значительно упрощает решение задач, связанных с управлением зависимостями. В мире свободных проектов и Java управление зависимостями порой оказывается непростым делом, так как среднестатистический проект зачастую опирается на множество сторонних библиотек.

Привязывая ваш процесс сборки к среде, в которой выполняется непрерывная интеграция создаваемого кода, вы приобретаете такие преимущества, как необычайно быстрый отклик системы на вносимые изменения и возможность быстро и уверенно объединять вносимые изменения.

Jenkins — это популярный сервер, обеспечивающий непрерывную интеграцию, который не только позволяет собирать проекты практически любых типов, но и обеспечивает обширную поддержку при генерировании отчетов, поскольку поддерживает богатую систему плагинов. Постепенно команда может возложить на Jenkins задачи, связанные с выполнением самых разнообразных сборок — от быстрой сборки, предназначенной для тестирования компонентов, до сложных и длительных сборок на этапе системной интеграции.

Инструмент Leiningen — наиболее подходящий вариант для сборки проектов, написанных на языке Clojure. Он очень тесно связывает методологию разработки через тестирование и работу в интерактивной среде REPL с аккуратным механизмом, предназначенным для сборки и развертывания.

В следующей главе мы поговорим о быстрой веб-разработке. Это одна из тем, с которой основательные Java-разработчики бьются с тех пор, как появились первые веб-фреймворки на языке Java.

13 Быстрая веб-разработка

В этой главе:

- почему Java — не лучший язык для быстрой веб-разработки;
- критерии, которыми следует руководствоваться при выборе веб-фреймворка;
- сравнение веб-фреймворков, построенных на базе виртуальной машины Java;
- знакомство с Grails (на языке Groovy);
- знакомство с Compojure (на языке Clojure).

Порой скорость веб-разработки бывает важна. Очень важна. Многочисленные сайты и приложения, работающие на основе веб-технологий, определяют коммерческие и социальные взаимоотношения между людьми во всем мире. Бизнес-проекты (в особенности стартовые) выживают либо рушатся в зависимости от того, насколько быстро удается вывести новый продукт или услугу на рынок и вступить в конкурентную борьбу. В наши дни конечные пользователи ожидают немедленного ввода анонсированных новых функций и почти мгновенного исправления любых найденных ошибок. Современный пользователь исключительно нетерпелив.

К сожалению, большинство веб-фреймоворков, основанных на Java, плохо подходят для быстрой веб-разработки. Для того чтобы выдерживать конкуренцию, приходится обращаться к другим технологиям, например PHP или Rails.

Итак, чем эта ситуация оборачивается для основательного Java-разработчика? В последнее время на виртуальной машине Java активно развиваются языки для разработки на динамическом уровне и у вас появляются просто замечательные возможности для быстрой веб-разработки. Сейчас существуют такие фреймворки, как Grails (для языка Groovy) и Compojure (для языка Clojure), с помощью которых мы приобретаем все возможности быстрой веб-разработки. Таким образом, мы можем не отказываться от мощности и гибкости виртуальной машины Java и в то же время не тратить лишнего времени для освоения других технологий, таких как PHP или Rails.

JAVA EE 6 — ЕЩЕ ОДИН ШАГ К БЫСТРОЙ ВЕБ-РАЗРАБОТКЕ НА JAVA?

Новая версия платформы Java для предприятий (Java EE 6) значительно развилась со времен J2EE, многие компоненты которой — JSP, сервлеты и API EJB — оставляли желать лучшего. Несмотря на значительные оптимизации, достигнутые на платформе Java EE 6 (которые касаются и JSP, и сервлетов, и API EJB), она по-прежнему испытывает проблемы, связанные со статической типизацией и компиляцией. Эти сложности обусловлены именно тем, что базовой технологией платформы является язык Java.

В этой главе мы для начала объясним, почему веб-фреймворки, основанные на Java, не совсем подходят для быстрой веб-разработки. После этого мы обсудим широкий набор критериев, которым должен соответствовать качественный веб-фреймворк. Опираясь на кое-какие исследования и работы Мэтта Рэйбла, мы рассмотрим, как можно ранжировать различные веб-фреймворки для виртуальной машины Java по 20 критериям.

Одним из наиболее эффективных веб-фреймворков, если судить по количеству критериев, которым он удовлетворяет, является Grails. Мы подробно изучим этот веб-фреймворк, в основе которого лежит язык Groovy. На Grails сильно повлиял фреймворк Rails, который сейчас исключительно популярен.

В качестве альтернативы для Grails мы также поговорим о Compojure. Это веб-фреймворк, в основе которого лежит язык Clojure. Compojure помогает программировать для Сети в очень лаконичном стиле и быстро вести разработку.

Обсудим, почему основанные на Java веб-фреймворки плохо подходят для реализации современных веб-проектов.

13.1. Проблема с веб-фреймворками на основе Java

Как помните, в главе 7 мы рассмотрели пирамиду многоязычного программирования и три уровня, на которые она делится. Эта пирамида вновь приведена на рис. 13.1.

Рис. 13.1. Пирамида многоязычного программирования

Java — классический язык стабильного уровня, к этому же уровню относятся все основанные на Java веб-фреймворки. Поскольку Java — популярный и довольно развитый язык, в нем существует множество веб-фреймворков, в частности следующие:

- Spring MVC;
- GWT;
- Struts 2;

- Wicket;
- Tapestry;
- JSF (и другие родственные ему Faces-библиотеки);
- Vaadin;
- Play;
- классический JSP/Servlet.

Де-факто Java далеко не лидирует в области веб-разработки, и частично это объясняется тем, что как язык он просто не слишком подходит для этой сферы. Бывший руководитель проекта Struts 2, популярного веб-фреймворка на основе Java, сказал по этому поводу следующее:

«Я ушел на темную сторону ☺ и теперь предпочитаю работать с Rails — конечно, благодаря его лаконичности, упомянутой выше, но в этом фреймворке можно также смело забыть об отдельных этапах "сборки" и "развертывания". А вам, ребята, советую учесть, что именно с "этим" вам и придется конкурировать, если вы желаете привлечь в свои ряды Rails-разработчиков… и не желаете множить ряды таких "Java-дезертиров", каким стал я».

Крейг Мак-Кланахан (Craig McClanahan), 23 октября 2007 года (http://markmail.org/thread/qfb5sekad33eobh2)

Для начала выясним, почему компилируемый язык замедляет процесс разработки веб-приложений.

13.1.1. Почему компиляция Java не подходит для быстрой веб-разработки

Java — это компилируемый язык. Как мы уже указывали выше, это означает, что каждый раз, когда вы изменяете код веб-приложения, вам приходится выполнять все следующие шаги.

1. Перекомпилировать код Java.
2. Остановить ваш веб-сервер.
3. Заново развернуть изменения на веб-сервере.
4. Запустить веб-сервер.

Можете себе представить, сколько времени на это тратится! Ситуация тем более усложняется, если вы вносите в код множество мелких изменений, например меняете места назначения в контроллере или слегка дорабатываете представление.

ПРИМЕЧАНИЕ

В настоящее время граница между веб-сервером и сервером приложений все сильнее размывается. Это обусловлено появлением JEE6 (где вы можете запускать EJB в веб-контейнере), а также тем фактом, что большинство серверов приложений отличаются высокой модульностью. Говоря о веб-сервере, мы подразумеваем любой сервер, в котором есть контейнер для сервлетов.

Если вы опытный веб-разработчик, то уже догадываетесь, какие технологии можно применить для решения подобной проблемы. Большинство таких подходов опирается на возможность активизировать вносимые в код изменения без останов-

ки и перезапуска веб-сервера. Такая методика также именуется *оперативным развертыванием* (hot deployment). При оперативном развертывании можно заменять все ресурсы (например, целый файл WAR) либо выбирать и заменять лишь некоторые из них (допустим, единственную JSP-страницу). К сожалению, оперативное развертывание никогда не бывает абсолютно надежным (это объясняется ограничениями, связанными с загрузкой классов, а также бывает вызвано ошибками контейнеров). Зачастую веб-серверу все равно приходится выполнять ресурсозатратную перекомпиляцию кода.

В целом основанные на Java веб-фреймворки не позволяют вам гарантированно обеспечивать быстрый ввод вносимых изменений в работу. Но это не единственная проблема, связанная с такими веб-фреймворками. Еще один фактор, замедляющий веб-разработку, — недостаточная гибкость языка Java. Именно в данном контексте статическая типизация является недостатком, а не преимуществом.

ОПЕРАТИВНОЕ РАЗВЕРТЫВАНИЕ С ПОМОЩЬЮ JREBEL И LIVEREBEL

Если при работе вы обязаны использовать веб-фреймворк на Java, то рекомендуем обратить внимание на программы JRebel и LiveRebel (http://www.zeroturnaround.com/jrebel/). JRebel располагается между вашей IDE и веб-сервером. Когда вы вносите в код изменения на локальном компьютере, они автоматически вступают в силу на уже работающем веб-сервере. Это происходит благодаря поистине замечательным фокусам JVM (LiveRebel применяется при развертывании готовых продуктов). В принципе, это и есть наиболее правильный вариант оперативного развертывания и де-факто эти инструменты считаются «промышленным стандартом» для решения подобных проблем.

13.1.2. Почему статическая типизация не подходит для быстрой веб-разработки

На ранних этапах разработки нового продукта или новой функции зачастую бывает целесообразно сохранять расширяемый дизайн (применительно к типизации) на пользовательском уровне представления. Пользователю ничего не стоит вдруг потребовать, чтобы числовое значение стало рассчитываться с точностью до десятых, либо чтобы список книг стал списком книг и игрушек. В такой ситуации язык со статической типизацией может быть, мягко говоря, неудобен. Если вам придется превратить список объектов `Book` в список объектов `BookOrToy`[1], то потребуется менять статические типы во всем коде.

Верно и то, что вы всегда можете использовать базовый тип в качестве типа объектов, содержащихся в контейнерных классах (например, в таком качестве подойдет класс `Object` языка Java). Но лучше так не делать, поскольку это фактически означает возврат к языку Java, в котором еще не было дженериков.

Таким образом, определенно следует обратить внимание на веб-фреймворк, основанный на одном из языков, относящихся к динамическому уровню.

---

[1] Дело пахнет керосином! Если у вас будет класс предметной области, в названии которого встречается Or или And, то вы, скорее всего, нарушаете принципы SOLID, описанные в главе 11.

ПРИМЕЧАНИЕ

Разумеется, Scala — это статически типизированный язык. Но поскольку в Scala хорошо организовано выведение типов, в этом языке можно уклониться от большинства проблем, характерных для статической типизации в Java. Соответственно, Scala должен быть (и действительно является) перспективным языком для веб-разработки.

Прежде чем окончательно и бесповоротно приступить к веб-разработке на динамическом языке, сделаем шаг назад и рассмотрим проблему в более широком контексте. Поговорим о том, каким критериям должен соответствовать фреймворк, который хорошо подойдет для быстрой веб-разработки.

13.2. Критерии при выборе веб-фреймворка

В наше время нет недостатка в веб-фреймворках, основанных на языке Java, — и это неудивительно, учитывая, как долго Java остается наиболее востребованным языком программирования в мире. В последнее время появилось множество веб-фреймворков, основанных на альтернативных языках для виртуальной машины Java, в частности на Groovy, Scala и Clojure. Но, к сожалению, за много лет в этой области не выделился явный лидер, и вам придется тщательно подбирать оптимальный фреймворк.

Ваш веб-фреймворк должен помогать вам в решении многих задач, и все доступные фреймворки следует оценивать по нескольким критериям. Чем больше ваших пожеланий удовлетворяется во фреймворке, тем удобнее будет быстро разрабатывать с его помощью веб-приложения.

Мэтт Рэйбл составил список из 20 критериев для оценки веб-фреймворков[1]. Эти критерии кратко объяснены в табл. 13.1.

Таблица 13.1. Двадцать критериев для оценки веб-фреймворков

Критерий	**Примеры**
Продуктивность разработчика	Сколько дней понадобится на создание страницы, отвечающей принципам CRUD, — 1 или 5?
Восприятие с точки зрения разработчика	С таким фреймворком интересно работать?
Кривая обучения	Вы сможете выдавать практический результат через неделю или через месяц обучения?
Состояние проекта	В насколько сложной ситуации протекает разработка проекта?
Наличие специалистов	Много ли на рынке труда разработчиков, имеющих опыт обращения с данным фреймворком?
Тенденции на рынке труда	Насколько просто будет нанять разработчиков в будущем?

[1] Мэтт Рэйбл, презентация «Сравнение веб-фреймворков для разработки на виртуальной машине Java» (март 2011 года), см. http://raibledesigns.com/rd/page/publications

Критерий	Примеры
Поддержка шаблонов	Насколько легко придерживаться в данном фреймворке принципа DRY (не повторяться)?
Компоненты	Насколько данный фреймворк богат готовыми компонентами, например такими, как панель даты/времени?
AJAX	Поддерживает ли фреймворк асинхронные клиентские вызовы на языке JavaScript?
Плагины или дополнения	Можно ли встраивать в программу дополнительный функционал, например интегрировать ее с Facebook?
Масштабируемость	Может ли стандартный контроллер фреймворка справиться с 500 пользователями и более, работающими параллельно?
Поддержка тестирования	Можно ли делать тестовые запуски разрабатываемой программы?
i18n и i10n	Поддерживает ли фреймворк в готовом виде другие естественные языки и варианты локализации?
Валидация	Можно ли в данном фреймворке с легкостью проверять пользовательский ввод и обеспечивать быстрый отклик?
Многоязычная поддержка	Можете ли вы, скажем, одновременно использовать во фреймворке языки Java и Groovy?
Качество документации и руководств	Хорошо ли документированы распространенные практические случаи и разъяснены ли часто возникающие вопросы?
Опубликованные книги	Пользовались ли опытные специалисты из вашей отрасли этим фреймворком и написали ли об этом в каких-либо источниках?
Поддержка REST на клиенте и сервере	Поддерживает ли фреймворк обмен информацией по протоколу HTTP (по замыслу разработчиков)?
Поддержка в мобильной среде	Хорош ли данный фреймворк в поддержке операционных систем iOS, Android и других систем для мобильных устройств?
Степень риска	Какое приложение вы пишете: кулинарную записную книжку или электронную начинку для управления атомной станцией?

Как видите, список довольно велик, и при выборе фреймворка вы должны сами решать, какие критерии для вас наиболее важны. К счастью, Мэтт не так давно провел несколько замечательных исследований в этой области[1]. Хотя полученные им результаты и вызывают жаркие споры, уже можно утверждать, что начинает вырисовываться четкая картина. На рис. 13.2 показан рейтинг различных фреймворков (по 100-балльной шкале) при оценке в соответствии с теми из вышеприведенных критериев, которые наиболее важны для быстрой веб-разработки. К таким критериям относятся продуктивность разработчика, поддержка при тестировании и качество документации.

Возможно, ваши потребности будут немного иными. Вы можете без труда провести анализ по интересующим вас параметрам, изменив важность отдельных критериев на странице http://bit.ly/jvm-frameworks-matrix и сгенерировав подобную диаграмму.

[1] Можете себе представить, насколько некоторые программисты увлекаются работой с любимым фреймворком!

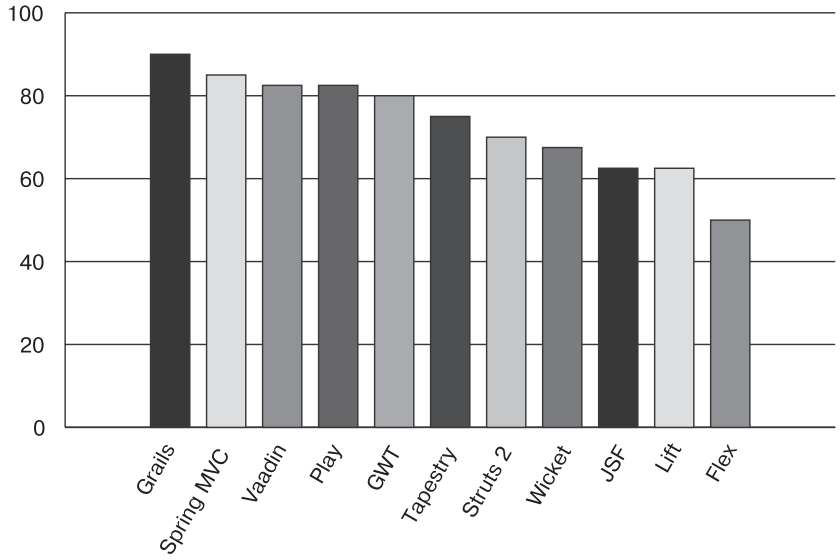


Рис. 13.2. Ценность различных фреймворков при быстрой веб-разработке в соответствии с критериями Мэтта Рэйбла

СОВЕТ

Настоятельно рекомендуем попробовать прототипировать какой-либо функционал в двух-трех лидирующих фреймворках, соответствующих вашим критериям, и лишь потом делать окончательный выбор.

Итак, теперь, когда вы знаете, на какие критерии ориентироваться, и научились пользоваться удобным инструментом Мэтта, мы полагаем, что вы сможете принимать взвешенные решения при выборе фреймворка для быстрой веб-разработки. В соответствии с нашими критериями на первое место вышел фреймворк Grails (что касается Compojure — он не в лидерах, но он совсем новый, и мы полагаем, что в самом ближайшем будущем он попадет в их число).

Итак, для начала познакомимся с победителем — Grails.

13.3. Знакомство с Grails

Grails — это фреймворк для быстрой веб-разработки, в основе которого лежит язык Groovy. Для реализации всех необходимых функций Grails использует несколько сторонних библиотек, в частности Spring, Hibernate, JUnit и сервер Tomcat, а также некоторые другие. Это многоуровневый веб-фреймворк, предлагающий решения по всем 20 критериям, перечисленным в разделе 13.2. Еще один важный момент заключается в том, что Grails активно заимствует применяемую в Rails *концепцию программирования по соглашениям*. Если вы пишете код в соответствии с соглашениями, то фреймворк будет выполнять за вас множество задач, решаемых на уровне шаблонного кода.

В этом разделе мы рассмотрим, как создать ваше первое быстрое приложение. Создавая его, вы не раз сможете оценить, какой смысл вкладывается в слово «быстрый» при веб-разработке в Grails. Кроме того, мы упомянем здесь важные технологии, которые стоит изучить, если вы собираетесь более плотно работать с Grails и создавать серьезные коммерческие приложения.

НЕ НРАВИТСЯ GROOVY? ПОПРОБУЙТЕ SPRING ROO

Spring Roo (www.springsource.org/roo) — это фреймворк для быстрой веб-разработки, использующий те же основные принципы, что и Grails, но базирующийся на языке Java. Он предоставляет разработчику более обширный доступ к фреймворку внедрения зависимостей Spring. Нам он кажется не таким зрелым, как Grails, но если вам совсем не нравится Groovy, то эта альтернатива может вас заинтересовать.

Если вы не знаете языка Groovy, то вам стоит изучить главу 8, посвященную этому языку. Если вы уже проштудировали эту главу, то скачайте и установите Grails. О том, как это делается, подробно рассказано в приложении C.

Когда справитесь с этим делом, можно приступать к написанию вашего первого проекта с Grails.

13.4. Экспресс-проект с Grails

В этом разделе мы быстро напишем несложный проект с Grails, а также выделим особенности, которые делают Grails поистине великолепным веб-фреймворком. При изучении материала из этого раздела мы рассмотрим следующие вопросы:

- создание объекта предметной области (объекта домена);
- разработку через тестирование;
- сохраняемость объекта предметной области;
- создание тестовых данных;
- контроллеры;
- GSP-виды;
- скаффолдинг и автоматическое создание пользовательского интерфейса;
- цикл быстрой веб-разработки.

В текущем разделе книги мы поработаем над созданием базового персонажа (`PlayerCharacter`) для ролевой игры[1]. В конце раздела у вас будет готов простой объект предметной области (`PlayerCharacter`), для которого мы реализуем:

- несколько рабочих тестов;
- ряд готовых предварительно заполненных тестовых данных;
- возможность сохранения в базе данных;
- готовый пользовательский интерфейс, в котором можно выполнять CRUD-операции.

[1] Например, для «Подземелий и драконов» или для «Властелина Колец».

В первую очередь Grails помогает сэкономить время при работе над автоматическим созданием структуры быстрого проекта. Выполнив команду `grails create-app <my-project>`, вы получаете уже собранный каркас проекта! Вам лишь нужно убедиться, что у вас есть активное соединение с Интернетом, так как программа будет загружать стандартные зависимости Grails (Spring, Hibernate, JUnit, сервер Tomcat и пр.).

Для загрузки зависимостей и управления ими Grails использует технологию под названием Apache Ivy. Весь процесс очень напоминает работу с зависимостями в системе Maven (о ней мы говорили в главе 12). Следующая команда создает приложение под названием pcgen_grails и готовит для вас структуру проекта, оптимизированную в соответствии с соглашениями, действующими в Grails.

```
grails create-app pcgen_grails
```

Как только будут загружены все зависимости и завершены другие автоматизированные этапы установки, у вас должна получиться структура проекта примерно как на рис. 13.3.

```
pcgen_grails
  application.properties  ---> Базовая информация о приложении и его версиях
  + grails-app
    + conf                ---> Расположение конфигурационных артефактов
      + hibernate         ---> Опциональная конфигурация Hibernate
      + spring            ---> Опциональная конфигурация Spring
    + controllers         ---> Местоположение артефактов контроллеров
    + domain              ---> Местоположение классов предметной области
    + i18n                ---> Местоположение пакетов сообщений для локализаций
    + services            ---> Местоположение служб
    + taglib              ---> Местоположение библиотек тегов
    + util                ---> Местоположение специальных вспомогательных классов
    + views               ---> Местоположение видов
      + layouts           ---> Местоположение макетов
  + lib
  + scripts               ---> Сценарии
  + src
    + groovy              ---> Опционально; расположение файлов с исходным
                               кодом Groovy (или типов, не указанных в grails-app/*)
    + java                ---> Опционально; расположение файлов
                               с исходным кодом Java
  + test                  ---> Сгенерированные тестовые классы
  + web-app
    + WEB-INF
```

Рис. 13.3. Компоновка проекта Grails

Что ж, теперь можно приступать к написанию готового кода! Для начала создадим классы предметной области.

13.4.1. Создание объекта предметной области

В Grails объекты предметной области (domain objects) считаются центральной частью проекта. Таким образом, с этим фреймворком логично работать в стиле проблемно-ориентированного проектирования (domain-driven design или DDD)[1]. Для создания объектов предметной области применяется команда `grails create-domain-class`.

В следующем примере мы создаем объект `PlayerCharacter`, представляющий собой персонаж из ролевой игры:

```
cd pcgen_grails
grails create-domain-class com.java7developer.chapter13.PlayerCharacter
```

Grails автоматически создает для вас следующие файлы:

- файл исходников `PlayerCharacter.groovy`, представляющий ваш объект предметной области (располагается по адресу `grails-app/domain/com/java7developer/chapter13`);
- файл исходников `PlayerCharacterTests.groovy` для разработки тестов компонентов (находится по адресу `test/unit/com/java7developer/chapter13`).

Итак, вы уже видите, что Grails стимулирует разработчика писать тесты компонентов!

Далее мы дополним объект `PlayerCharacter`, определив для персонажа несколько атрибутов, в частности силу, ловкость и магическую силу. Зная параметры этих атрибутов, можно представить, как персонаж будет действовать в воображаемом игровом мире[2]. Но если вы прочитали главу 11, то, вероятно, решите начать работу с написания тестов.

13.4.2. Разработка через тестирование

Итак, расширим набор функций нашего персонажа `PlayerCharacter`, придерживаясь разработки через тестирование. Напишем заведомо неуспешный тест, а потом добьемся его прохождения, реализовав `PlayerCharacter`.

Мы воспользуемся еще одной возможностью быстрой веб-разработки, предоставляемой в Grails: в этом фреймворке поддерживается автоматическая валидация объектов предметной области. Метод `validate()` можно автоматически вызывать применительно к любому объекту предметной области в Grails. Метод позволяет удостовериться, что конкретный объект валиден. В листинге 13.1 вы убедитесь, что все три качества персонажа — `strength`, `dexterity` и `charisma` — выражены числовыми значениями в диапазоне от 3 до 18.

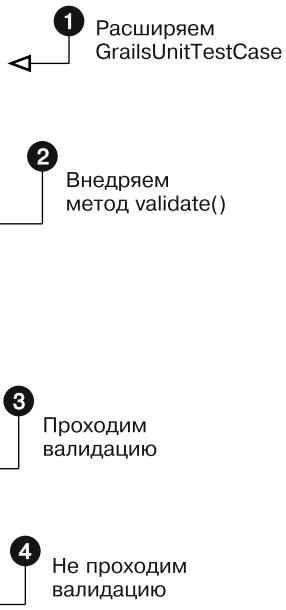
---

[1] Подробнее о DDD (термин предложен Эриком Эвансом) можно почитать на сайте сообщества, занимающегося проблемно-ориентированным проектированием (http://domaindrivendesign.org/).

[2] Как Гвинет будет побеждать врагов: могучим ударом, неуловимо ускользая от них или обезоруживая улыбкой?

Листинг 13.1. Тесты компонентов для PlayerCharacter

```
package com.java7developer.chapter13

import grails.test.*

class PlayerCharacterTests extends GrailsUnitTestCase {

  PlayerCharacter pc;

  protected void setUp() {
    super.setUp()
    mockForConstraintsTests(PlayerCharacter)
  }

  protected void tearDown() {
    super.tearDown()
  }

  void testConstructorSucceedsWithValidAttributes {
    pc = new PlayerCharacter(3, 5, 18)
    assert pc.validate()
  }

  void testConstructorFailsWithSomeBadAttributes() {
    pc = new PlayerCharacter(10, 19, 21)
    assertFalse pc.validate()
  }
}
```



Тесты компонентов во фреймворке Grails всегда являются расширениями класса GrailsUnitTestCase (1). Как и всегда при работе с тестами, основанными на JUnit, у вас есть методы setUp() и tearDown(). Но для того, чтобы использовать на этапе тестирования компонентов встроенный в Grails метод validate(), его нужно получить с помощью метода mockForConstraintsTest (2). Просто Grails считает, что метод validate() должен применяться лишь на этапе интеграционного тестирования, и только тогда автоматически предоставляет его. Вам же требуется более быстрый отклик, и целесообразно будет задействовать данный метод уже во время тестирования компонентов. Далее можно вызывать метод validate() и проверять, валиден ли объект предметной области (3), (4).

Теперь вы можете запускать тесты, выполнив в командной строке следующее:

```
grails test-app
```

Эта команда запускает как тесты компонентов, так и интеграционные тесты (хотя до сих пор мы занимались лишь тестами компонентов). По выводу в консоли будет понятно, что тесты пройти не удается.

Чтобы подробно изучить, почему именно не удается пройти тесты, посмотрите каталог target/test-reports/plain. Например, чтобы просмотреть информацию о том приложении, над которым вы сейчас работаете, найдите файл TEST-unit-unit-com.java7developer.chapter13.PlayerCharacterTests.txt. В нем будет написано, что тест не пройден из-за отсутствия подходящего конструктора при попытке создать

новый PlayerCharacter. И это действительно так, поскольку мы еще не разобрались с объектом предметной области PlayerCharacter!

Теперь мы можем написать класс PlayerCharacter, запуская по ходу работы тесты до тех пор, пока их не удастся пройти. Как мы и собирались, добавим три атрибута: strength, dexterity и charisma. Но при указании минимального и максимального значения для этих атрибутов (3 и 18 соответственно) необходимо соблюдать специальный синтаксис constraint. В таком случае можно будет пользоваться удобным стандартным методом validate(), предоставляемым в Grails.

ОГРАНИЧЕНИЯ В GRAILS

Ограничения реализуются с помощью базового API валидации, относящегося к Spring. Такие ограничения позволяют задавать требования к валидации свойств ваших классов предметной области. Список ограничений, применяемых в Grails, довольно велик (в листинге 13.2 мы воспользуемся ограничениями min и max). Кроме того, вы можете писать собственные ограничения. Подробнее об этом рассказывается на сайте http://grails.org/doc/latest/guide/validation.html.

В листинге 13.2 показан образец класса PlayerCharacter, в котором предоставляется минимальный набор атрибутов и ограничений, необходимых для прохождения теста.

Листинг 13.2. Класс PlayerCharacter

```
package com.java7developer.chapter13

class PlayerCharacter {

  Integer strength
  Integer dexterity
  Integer charisma

  PlayerCharacter() {}

  PlayerCharacter(Integer str, Integer dex, Integer cha) {
    strength = str
    dexterity = dex
    charisma = cha
  }

  static constraints = {
    strength(min:3, max:18)
    dexterity(min:3, max:18)
    charisma(min:3, max:18)
  }
}
```

1 Типизированные переменные сохраняются

2 Конструктор для прохождения теста

3 Ограничения для валидации

Класс PlayerCharacter довольно прост. Имеем три базовых атрибута, которые автоматически сохраняются в таблице PlayerCharacter (1). Вы предоставляете

конструктор, принимающий все три атрибута в качестве аргументов ❷. Специальный блок static для constraints позволяет указывать значения min и max ❸, в сравнении с которыми будет производиться валидация в методе validate().

Итак, мы разобрались с классом PlayerCharacter. Теперь он должен без проблем проходить тесты (чтобы убедиться в этом, снова запустите grails test-app). Если вы тщательно придерживаетесь цикла разработки через тестирование, то на данном этапе займитесь рефакторингом класса PlayerCharacter и тестов, чтобы код стал чуть чище.

13.4.3. Сохраняемость объектов предметной области

Поддержка сохраняемости обеспечивается автоматически, поскольку Grails воспринимает любую переменную класса, имеющую конкретный тип, как поле, которое должно сохраняться в базе данных. Grails автоматически отображает объект предметной области на одноименную таблицу. В случае с объектом предметной области PlayerCharacter все три атрибута (strength, dexterity и charisma) относятся к типу Integer и, следовательно, будут отображаться на таблицу PlayerCharacter. По умолчанию Grails на внутрисистемном уровне использует Hibernate и предоставляет базу данных HSQLDB, хранимую в оперативной памяти. Мы упоминали об этой базе в главе 11, когда изучали подставные объекты — один из видов тестовых двойников. Но вы вполне можете переопределить эту стандартную настройку, указав собственный источник данных.

В файле grails-app/conf/DataSource.groovy содержится конфигурация источника данных. Здесь можно задать настройки источника данных для каждого окружения. Не забывайте, что Grails уже предоставляет стандартную реализацию HSQLDB в проекте pcgen_grails, поэтому для запуска приложения вам не потребуется вообще ничего менять! Но в листинге 13.3 показано, как можно изменить конфигурацию, чтобы перейти к использованию другой базы данных.

Например, в рабочей версии вашего проекта может использоваться база MySQL, но при этом разрабатываемая и тестовая версии проекта будут завязаны на HSQLDB. Синтаксис соответствует стандартной конфигурации Java при работе с JDBC (стандарт соединения Java с базами данных), с которой вы, конечно же, знакомы из предыдущего опыта программирования на этом языке.

Листинг 13.3. Вариант источника данных для pcgen_grails

```
dataSource {}

environments {
  development { dataSource {} }
  test { dataSource {} }

  production {
    dataSource {
      dbCreate = "update"
      driverClassName = "com.mysql.jdbc.Driver"
```

Источник данных для рабочей версии

Драйвер базы данных

```
      url = "jdbc:mysql://localhost/my_app"
      username = "root"
      password = ""
    }
  }
}
```

URL для соединения по JDBC

Специалисты, разрабатывавшие Grails, подумали и о том, как неудобно вручную вводить большой объем тестовых данных. Поэтому во фреймворке предоставляется механизм для автоматического заполнения базы данных при запуске приложения.

13.4.4. Создание тестовых данных

Создание тестовых данных обычно выполняется в классе Grails `BootStrap`, который находится по адресу `grails-app/conf/BootStrap.groovy`. Каждый раз при запуске приложения Grails или контейнера сервлетов срабатывает метод `init`. По значению он синонимичен пусковому сервлету (startup servlet), который используется в большинстве веб-фреймворков на Java.

ПРИМЕЧАНИЕ

С помощью класса Bootstrap вы можете инициализировать практически что угодно, но пока мы сосредоточимся на тестовых данных.

В листинге 13.4 мы генерируем на этапе инициализации два объекта предметной области `PlayerCharacter`, которые затем сохраняем в базе данных.

Листинг 13.4. Начальная загрузка тестовых данных для pcgen_grails

```
import com.java7developer.chapter13.PlayerCharacter

class BootStrap {

  def init = { servletContext ->
    if (!PlayerCharacter.count()) {
      new PlayerCharacter(strength: 3, dexterity: 5, charisma: 18)
                                    ➥ .save(failOnError: true)
      new PlayerCharacter(strength: 18, dexterity: 10, charisma: 4)
                                    ➥ .save(failOnError: true)
    }
  }

  def destroy = {}
}
```

1 Начальная загрузка при запуске контекста Servlet

Метод `init` выполняется всякий раз, когда код развертывается в контейнере сервлета (то есть при запуске приложения и при каждом автоматическом развертывании Grails) 1. Чтобы гарантировать, что имеющиеся данные не будут переопределяться, можно выполнить простое действие `count()` применительно к уже существующим экземплярам `PlayerCharacter`. Если вы определили, что такие объекты

отсутствуют, то можете создать несколько экземпляров. Здесь важно отметить следующий момент: вы можете быть уверены, что `save` не выполнится, если система выдаст исключение или при конструировании объекта не будет пройдена валидация. Если желаете, можете выполнять логику прерывания действия в методе `destroy`.

Итак, теперь у вас есть базовый объект предметной области с поддержкой сохранения состояния. Переходим к следующему этапу работы — визуализации ваших объектов предметной области на веб-странице. Для этого нам потребуется построить элемент, который в Grails называется *контроллером*. Этот термин может быть знаком вам из работы с паттерном MVC (модель — вид — контроллер).

13.4.5. Контроллеры

Grails работает в соответствии с паттерном проектирования MVC (модель — вид — контроллер) и применяет контроллер для обработки веб-запросов, поступающих от клиента. Как правило, на клиенте при этом используется браузер. В Grails действует соглашение, в соответствии с которым необходимо иметь по контроллеру для каждого объекта предметной области.

Чтобы создать контроллер для вашего объекта `PlayerCharacter`, просто выполните такую команду:

```
grails create-controller com.java7developer.chapter13.PlayerCharacter
```

Следует указать здесь полностью квалифицированное имя класса объекта предметной области, в том числе имя пакета.

Как только выполнение команды завершится, вы увидите, что система подготовила следующие файлы:

- файл исходников `PlayerCharacterController.groovy`, представляющий ваш контроллер для объекта предметной области `PlayerCharacter` (по адресу `grails-app/controller/com/java7developer/chapter13`);
- файл исходников `PlayerCharacterControllerTests.groovy` для разработки тестов компонентов для вашего контроллера (`test/unit/com/java7developer/chapter13`);
- каталог `grails-app/view/playerCharacter` (с ним мы будем работать позже).

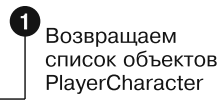
Эти контроллеры поддерживают URL с передачей состояния представления (RESTful) и в достаточно простой форме организуют отображение действий (action mapping). Допустим, вам требуется отобразить RESTful URL http://localhost:8080/pcgen_grails/playerCharacter/list для возврата списка объектов `PlayerCharacter`. В соответствии с применяемым в Grails программированием по соглашениям вы можете отобразить этот URL в классе `PlayerCharacterController`, для чего потребуется минимальное количество исходного кода. URL состоит из таких элементов, как:

- сервер (http://localhost:8080/);
- базовый проект (pcgen_grails/);
- производная часть имени контроллера (playerCharacter/);
- переменная блока действий, определенная в контроллере (list).

Чтобы посмотреть, как это отразится в коде, заменим имеющийся исходный код `PlayerCharacterController.groovy` кодом из листинга 13.5.

Листинг 13.5. PlayerCharacterController

```
package com.java7developer.chapter13

class PlayerCharacterController {
  List playerCharacters
  def list = {
    playerCharacters = PlayerCharacter.list()
  }
}
```



При работе в соответствии с соглашениями Grails атрибут `playerCharacters` будет использоваться в виде, на который ссылается RESTful URL ❶.

Но если ваше приложение с Grails уже было запущено и работало, а вы просто перешли по ссылке http://localhost:8080/pcgen_grails/playerCharacter/list, то ничего не получится, так как вы еще не создали вид JSP или GSP. Займемся этой проблемой.

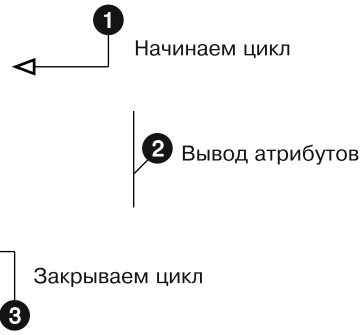
13.4.6. Виды GSP/JSP

В Grails можно создавать виды GSP или JSP. В этом разделе мы создадим простую страницу GSP, на которой будут перечисляться объекты `PlayerCharacter` (дизайнеры, веб-разработчики, специалисты по HTML/CSS, сейчас будет интересно).

Страница GSP, соответствующая коду из листинга 13.6, находится по адресу `grails-app/view/player-Character/list.gsp`.

Листинг 13.6. Страница GSP, на которой перечисляются объекты PlayerCharacter

```
<html>
  <body>
    <h1>PC's</h1>
    <table>
      <thead>
        <tr>
          <td>Strength</td>
          <td>Dexterity</td>
          <td>Charisma</td>
        </tr>
      </thead>
      <tbody>
      <% playerCharacters.each({ pc -> %>
        <tr>
          <td><%="${pc?.strength}"%></td>
          <td><%="${pc?.dexterity}"%></td>
          <td><%="${pc?.charisma}"%></td>
        </tr>
        <%})%>
      </thead>
    </table>
  </body>
</html>
```



Этот HTML-код очень прост. Здесь важнее всего понять, как используются скриплеты Groovy. Вы заметите знакомый синтаксис функциональных литералов Groovy (они были рассмотрены в главе 8), упрощающий перебор коллекций ❶. Потом мы ссылаемся на атрибуты персонажей (обратите внимание на безопасный оператор разыменования `null`, его мы также рассматривали в главе 8) ❷. Наконец, мы закрываем функциональный литерал ❸.

Теперь, когда вы подготовили объект предметной области, его контроллер и вид, можно в первый раз запустить приложение Grails! Просто выполните в командной строке следующий код:

```
grails run-app
```

Grails автоматически запустит экземпляр Tomcat по адресу http://localhost:8080 и развернет на нем приложение pcgen_grails.

ВНИМАНИЕ

Многие программисты уже освоили работу с популярным сервером Tomcat для Java. Если вы хотите запустить одновременно несколько экземпляров Tomcat, то потребуется изменить имена портов так, чтобы лишь один экземпляр слушал порт 8080.

Откройте свой любимый браузер и перейдите к странице http://localhost:8080/pcgen_grails/, где будут перечислены ваши `PlayerCharacterController` (рис. 13.4).

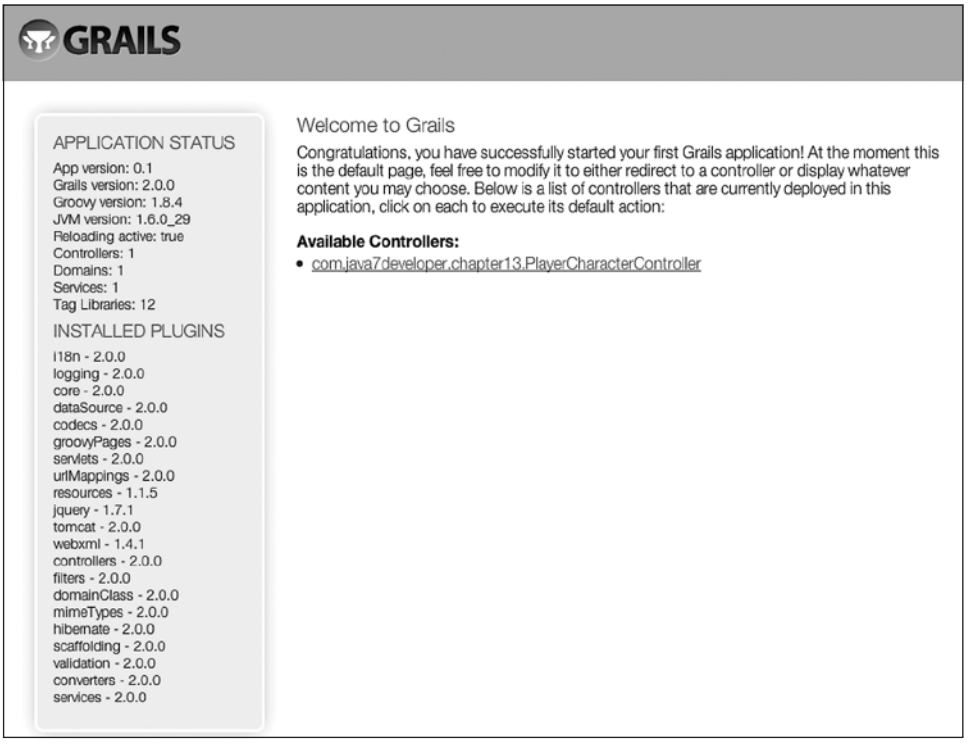


Рис. 13.4. Домашняя страница pcgen_grails

Щелкнув на ссылке com.java7developer.chapter13.PlayerCharacterController, вы перейдете на экран, где будет выведен список объектов предметной области PlayerCharacter.

Итак, эта GSP-страница получилась у нас практически без труда, но не можем ли мы и эту работу перепоручить фреймворку? Благодаря функции скаффолдинга, поддерживаемой в Grails, вы можете быстро прототипировать CRUD-страницы для вашего домена.

13.4.7. Скаффолдинг и автоматическое создание пользовательского интерфейса

В Grails поддерживается функция скаффолдинга, позволяющая автоматически создавать пользовательский интерфейс, через который вы сможете выполнять над вашим объектом предметной области CRUD-операции.

Для пользования этой функцией замените код, который сейчас присутствует в файле исходников PlayerCharacterController.groovy, приведенным в листинге 13.7.

Листинг 13.7. PlayerCharacterController с применением скаффолдинга

```
package com.java7developer.chapter13

class PlayerCharacterController {
  def scaffold = PlayerCharacter
}
```

❶ Скаффолдинг для PlayerCharacter

Класс PlayerCharacterController очень прост. Grails использует здесь соглашение, в соответствии с которым имя объекта предметной области присваивается переменной scaffold ❶. После этого Grails автоматически создает для вас стандартный пользовательский интерфейс, что происходит почти мгновенно.

Кроме того, потребуется временно переименовать ваш файл list.gsp в list_original.gsp, чтобы он не помешал операциям, которые должны выполняться при скаффолдинге. Как только закончите эту работу, обновите страницу http://localhost:8080/pcgen_grails/playerCharacter/list — и увидите автоматически сгенерированный список объектов предметной области PlayerCharacter, как на рис. 13.5.

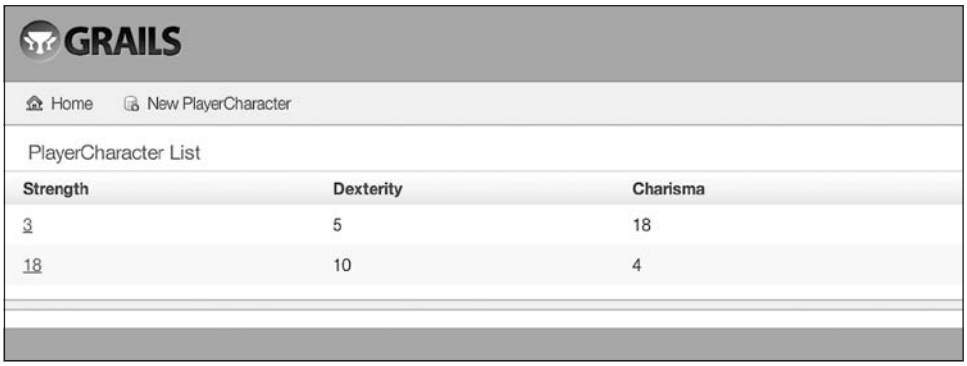


Рис. 13.5. Список экземпляров PlayerCharacter

На этой странице вы также можете создавать, обновлять или удалять столько объектов PlayerCharacter, сколько захотите. Убедитесь, что добавили пару объектов предметной области PlayerCharacter, а потом переходите к чтению следующего раздела, где мы поговорим о быстром циклическом вводе изменений.

13.4.8. Быстрая циклическая разработка

У команды Grails run-app есть одна небольшая особенность, которая позволяет делать разработку не просто «быстрой», а «стремительной». Если приложение запускается с помощью команды grails run-app, то Grails может служить связующим звеном между исходным кодом и работающим сервером. Хотя такой подход и не рекомендуется при работе с «чистовыми» установками (поскольку в данном случае возникают проблемы с производительностью), он целесообразен на этапах разработки и тестирования.

СОВЕТ

В чистовых разработках обычно используется команда grails war, создающая файл с расширением WAR. Такой файл поддается стандартному процессу развертывания.

Если вы измените какой-либо исходный код в вашем приложении Grails, то эти изменения автоматически отразятся на вашем работающем сервере[1]. Попробуем изменить объект предметной области PlayerCharacter. Добавим переменную name в файл PlayerCharacter.groovy и сохраним его.

```
String name = 'Gweneth the Merciless'
```

Далее можете перезагрузить страницу http://localhost:8080/pcgen_grails/playerCharacter/list и увидите, что атрибут name добавлен как новый столбец для объектов PlayerCharacter. Как видите, не пришлось ни останавливать Tomcat, ни перекомпилировать код, ни вообще что-либо делать. Ввод новых изменений получается практически мгновенным, и это основной фактор, благодаря которому Grails является лидирующим фреймворком для быстрой веб-разработки.

На этом мы завершаем краткое знакомство с Grails. Надеемся, вы смогли составить впечатление о том, насколько быстрой бывает веб-разработка при умелом применении фреймворка Grails. Стандартный функционал фреймворка можно настраивать самыми разными способами, эта тема заслуживает дополнительного изучения. Вкратце рассмотрим, на какие моменты стоит обратить внимание.

13.5. Дальнейшее исследование Grails

К сожалению, в этой главе мы не можем достаточно подробно описать фреймворк Grails. В этом небольшом разделе мы кратко расскажем о дополнительных облас-

[1] Это верно для большинства типов исходного кода и при условии, что вносимые изменения не вызывают ошибок.

тях, которые могут заинтересовать вас как начинающего Grails-разработчика. Итак, мы обсудим:

- логирование;
- GORM — объектно-реляционное отображение в Grails;
- плагины Grails.

Кроме того, можете просмотреть сайт http://www.grails.org, на котором предлагаются базовые руководства по этим темам. Мы также очень рекомендуем познакомиться с книгой Глена Смита и Питера Ледбрука *Grails in Action* (издательство Manning, 2009), где этот фреймворк описан исключительно подробно.

Итак, для начала рассмотрим, как в Grails организуется логирование.

13.5.1. Логирование

На внутрисистемном уровне логирование обеспечивается с помощью кода log4j. Функция логирования конфигурируется в файле `grails-app/conf/Config.groovy`.

Возможно, вы захотите отслеживать сообщения WARN (предупреждения) для кода из пакета chapter13, но в классе предметной области `PlayerCharacter` решите ограничиться лишь сообщениями об ошибках (`ERROR`). Для этого можете добавить следующий фрагмент кода в начале имеющегося конфигурационного раздела log4j в файле `Config.groovy`:

```
log4j = {
  ...
  warn  'com.java7developer.chapter13'
  error 'com.java7developer.chapter13.PlayerCharacter',
        'org.codehaus.groovy.grails.web.servlet', // контроллеры
  ...
}
```

Такая конфигурация логирования может быть не менее гибкой, чем в конфигурационном файле `log4j.xml`, которым мы пользовались при изучении работы с log4j в Java.

Далее рассмотрим GORM — технологию объектно-реляционного отображения для Grails.

13.5.2. GORM — объектно-реляционное отображение

На внутрисистемном уровне технология GORM реализуется с применением Spring/Hibernate — этот тандем хорошо знаком большинству Java-разработчиков. GORM имеет несколько особенностей, но принципиально во многом напоминает технологию JPA из Java.

Чтобы быстро протестировать некоторые функции этого механизма хранения состояния, откроем консоль Grails. Для этого выполним в командной строке следующее:

```
grails console
```

Помните, мы говорили о консоли Groovy в главе 8? Сейчас мы будем работать в очень похожей среде, предназначенной для обращения с приложениями Grails.

Сначала сохраним объект предметной области `PlayerCharacter`:

```
import com.java7developer.chapter13.PlayerCharacter
new PlayerCharacter(strength:18, dexterity:15, charisma:15).save()
```

Теперь, когда объект `PlayerCharacter` сохранен, вы можете получить его несколькими способами. Простейший способ — возвратить полный вариант экземпляра, пригодный для записи. Это делается с помощью неявного свойства `id`, которое Grails добавляет к вашему классу предметной области. Замените предыдущий код в консоли следующим фрагментом и выполните его.

```
import com.java7developer.chapter13.PlayerCharacter
def pc = PlayerCharacter.get(1)
assert 18 == pc.strength
```

Чтобы обновить объект, измените какие-либо свойства и вновь вызовите `save()`. Вновь очистите консоль, а потом выполните следующий фрагмент кода.

```
import com.java7developer.chapter13.PlayerCharacter
def pc = PlayerCharacter.get(1)
pc.strength = 5
pc.save()
pc = PlayerCharacter.get(1)
assert 5 == pc.strength
```

Чтобы удалить экземпляр, воспользуйтесь методом `delete()`. Как и выше, очистите консоль и запустите следующий код, чтобы избавиться от `PlayerCharacter`.

```
import com.java7developer.chapter13.PlayerCharacter
def pc = PlayerCharacter.get(1)
pc.delete()
```

GORM обладает массой возможностей для указания отношений «многие к одному» и «многие ко многим», а также обеспечивает другие функции, знакомые вам по работе с Hibernate/JPA.

Теперь поговорим о плагинах. Эта концепция Grails заимствована из Rails и помогает значительно экономить время.

13.5.3. Плагины Grails

Grails значительно упрощает быструю веб-разработку потому, что располагает большим репозиторием плагинов, выполняющих за вас распространенные виды работы. К числу наиболее популярных плагинов относятся следующие:

- плагин для интеграции с Cloud Foundry (для развертывания приложений Grails в облаке);
- Quartz (для диспетчеризации);
- Mail (для работы с электронной почтой);
- Twitter, Facebook (для интеграции с социальными сетями).

Чтобы посмотреть, какие плагины доступны, выполните в командной строке следующий код:

```
grails list-plugins
```

После этого можно выполнить команду `grails plugin-info [name]`, чтобы получить более подробную информацию о конкретном плагине. Замените `[name]` именем плагина, о котором вам нужна такая информация. Вы также можете открыть страницу http://grails.org/plugins/ и почитать подробности о плагинах и их экосистеме.

Чтобы установить один из этих плагинов, выполните команду `grails install-plugin [name]`, заменив `[name]` названием плагина, который хотите установить. Например, можно установить плагин Joda-Time для более качественной поддержки даты и времени.

```
grails install-plugin joda-time
```

Устанавливая плагин Joda-Time, вы можете изменить объект предметной области `PlayerCharacter` и добавить атрибут `LocalDate`. Запишите следующие операторы `import` в класс предметной области.

```
import org.joda.time.*
import org.joda.time.contrib.hibernate.*
```

Далее добавьте следующий атрибут к классу предметной области `PlayerCharacter`.

```
LocalDate timestamp = new LocalDate()
```

Чем же такой подход отличается от обычного выставления ссылки на API в JAR-файле? Все дело в том, что плагин Joda-Time гарантирует соответствие своих типов философии программирования по соглашениям, соблюдаемой в Grails. Это означает, что типы Joda-Time отображаются на типы базы данных, в полной мере поддерживается как отображение, так и скаффолдинг. Если теперь вы вернетесь на страницу http://localhost:8080/pcgen_grails/playerCharacter/list, то увидите перечисленные даты.

Разработчики, умеющие обращаться с Grails, способны реализовать поразительно большой объем функций за очень короткое время именно благодаря такой поддержке в применении плагинов.

На этом мы заканчиваем ознакомительную экскурсию по Grails, но история о быстрой веб-разработке и сама глава еще не закончены. В следующем разделе мы обсудим Compojure — библиотеку для быстрой веб-разработки на языке Clojure. С ее помощью разработчик, знающий язык Clojure, может быстро и аккуратно создавать небольшие веб-приложения.

13.6. Знакомство с Compojure

Одна из самых пагубных идей в современной веб-разработке заключается в том, что любой сайт якобы должен быть спроектирован так же безупречно, как Google Search. На самом деле переконструирование может доставлять не меньше проблем, чем значительная сыроватость кода.

Прагматичный и основательный разработчик обязательно учитывает контекст, в котором будет функционировать веб-приложение, и никогда не пытается чрезмерно усложнять программу. Тщательный анализ нефункциональных требований к любому приложению — основная предпосылка для того, чтобы не сконструировать чего-то ненужного.

Compojure — как раз такой веб-фреймворк, который помогает не палить из пушки по воробьям. Он отлично подходит для разработки информационных веб-панелей (личных кабинетов), программ для отслеживания операций и многих других простых элементов, в которых широкая масштабируемость и прочие нефункциональные детали не столь важны, как простота и скорость разработки. Из этого описания легко понять, что фреймворк Compojure располагается в пирамиде многоязычного программирования между предметно-ориентированным и динамическим уровнем.

В этом разделе мы напишем простое приложение Hello World, потом обсудим простые правила, по которым в Compojure конструируется веб-приложение. Далее мы поговорим об удобной библиотеке Hiccup, применяемой в Clojure для работы с HTML. А после этого мы воспользуемся всеми этими стандартами и напишем с их помощью пример приложения.

Как понятно из рис. 13.6, Compojure строится на основе фреймворка Ring, связывающего Clojure с веб-контейнером Jetty. Но чтобы использовать Compojure/Ring, не обязательно обладать глубоким знанием Jetty.

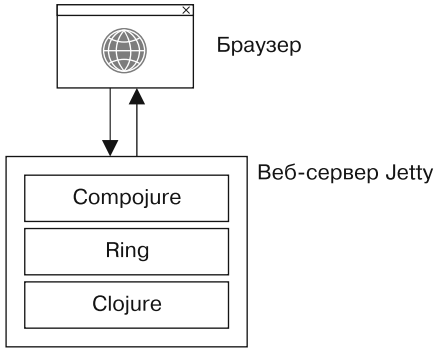


Рис. 13.6. Compojure и Ring

Попробуем написать простое приложение Hello World, в котором используется Compojure.

13.6.1. Hello World с Compojure

Начать работу с Compojure очень просто, поскольку он естественно вписывается в рабочий цикл сборочного инструмента Leiningen, с которым мы познакомились в предыдущей главе. Если вы еще не установили Leiningen и не прочли раздел о нем в главе 12, то сделайте это сейчас, так как дальнейший материал требует базового понимания работы с Leiningen.

Чтобы запустить проект, выполните обычную команду Leiningen:

```
lein new hello-compojure
```

Как говорилось выше, в разделе о Leiningen, вы можете без труда задать зависимости для проекта в файле `project.clj`. В листинге 13.8 показано, как это делается в вашем простом проекте Hello World.

Листинг 13.8. Простой проект Compojure project.clj

```
(defproject hello-compojure "1.0.0-SNAPSHOT"
  :description "FIXME: write description"
  :dependencies [[org.clojure/clojure "1.2.1"]
                 [compojure "0.6.2"]]
  :dev-dependencies [[lein-ring "0.4.0"]]
  :ring {:handler hello-compojure.core/app})
```

Как видите, макрос `(defproject)` во многом напоминает базовый пример, рассмотренный в главе 12. Здесь есть два дополнительных фрагмента метаданных.

- `:dev-dependencies` позволяет использовать во время разработки удобные команды `lein`. Такой пример будет рассмотрен ниже, когда мы будем обсуждать `lein ring server`.
- `:ring` обеспечивает подключение библиотеки Ring и принимает словарь с Ring-специфичными метаданными.

В данном случае мы передаем Ring свойство `:handler`. Очевидно, оно ожидает символьное выражение `app` из пространства имен `hello-compojure.core`. Рассмотрим описание этого соединительного элемента в `core.clj`, чтобы понять, как все сочетается вместе.

Листинг 13.9. Простой файл project.clj для проекта Hello World в Compojure

```
(ns hello-compojure.core
  (:use compojure.core)
  (:require [compojure.route :as route]
            [compojure.handler :as handler]))
(load "hello")

(defroutes main-routes
  (GET "/" [] (page-hello-compojure))

  (route/resources "/")
  (route/not-found "Page not found"))

(def app (handler/site main-routes))
```

Определяет основной маршрут

Регистрирует маршруты

Такой принцип использования `core.clj` для обеспечения связей и учета другой информации представляет собой полезное соглашение. Это простой способ загрузки отдельного файла, содержащего функции, которые вызываются при запросе определенного URL (они называются *страничными функциями*). На самом деле это просто соглашение, помогающее повысить удобочитаемость кода и более четко разделить ответственность.

В Compojure используется набор правил, называемых *маршрутами* (routes) и помогающих определить, как следует обработать входящий HTTP-запрос. Эти правила предоставляются фреймворком Ring, от которого зависит Compojure. Такие правила одновременно и просты, и очень полезны. Вы, вероятно, догадываетесь, что правило `GET "/"` в листинге 13.9 сообщает веб-серверу, как обрабатывать запросы GET, относящиеся к корневому URL. Мы подробнее обсудим маршруты в следующем подразделе.

Чтобы завершить код нашего простого примера, нам еще требуется файл `hello.clj`, находящийся в каталоге src/hello_compojure. В этом файле нужно определить единственную страничную функцию `(page-hello-compojure)`, как показано ниже:

```
(ns hello-compojure.core)
(defn page-hello-compojure [] "<h1>Hello Compojure</h1>")
```

Страничная функция — это обычная функция Clojure. Она возвращает строку, которая применяется в качестве содержимого HTML-тега `<body>` в документе, возвращаемом пользователю.

Рассмотрим этот пример в действии. Как вы уже догадываетесь, запустить его в Compojure совсем не сложно. Для начала убедимся, что у нас установлены все необходимые зависимости:

```
ariel:hello-compojure boxcat$ lein deps
Downloading: org/clojure/clojure/1.2.1/clojure-1.2.1.pom from central
Downloading: org/clojure/clojure/1.2.1/clojure-1.2.1.jar from central
Copying 9 files to /Users/boxcat/projects/hello-compojure/lib
Copying 17 files to /Users/boxcat/projects/hello-compojure/lib/dev
```

Пока все нормально. Далее нужно вывести приложение, воспользовавшись одним из тех удобных элементов, которые входят в состав Ring — основы Compojure. Мы имеем в виду метод `ring server`.

```
ariel:hello-compojure boxcat$ lein ring server
2011-04-11 18:02:48.596:INFO::Logging to STDERR via org.mortbay.log.StdErrLog
2011-04-11 18:02:48.615:INFO::jetty-6.1.26
2011-04-11 18:02:48.743:INFO::Started SocketConnector@0.0.0.0:3000
Started server on port 3000
```

Так мы запускаем простой веб-сервер Ring/Jetty (по умолчанию на порте 3000). Этот сервер может использоваться для разработки, при которой очень важен быстрый отклик системы. Сервер будет перезагружать все файлы, которые вы будете изменять.

ВНИМАНИЕ

Необходимо учитывать, что перезагрузка сервера, используемого при разработке, происходит на файловом уровне. Поэтому вполне возможно, что при перезагрузке страницы работающий сервер может сбросить свое состояние (или того хуже — частично сбросить). Если вы подозреваете, что возникла именно такая ситуация, и из-за этого стали возникать проблемы, то нужно остановить сервер и перезапустить его. Сервер Ring/Jetty запускается очень быстро, поэтому такая операция не замедлит ход вашей разработки.

Если перейти на порт 3000 на той машине, где вы ведете разработку (либо перейти на http://127.0.0.1:3000 на локальной машине) в вашем любимом браузере, то на экране появится текст Hello Compojure.

13.6.2. Ring и маршруты

Подробнее рассмотрим, как маршруты конфигурируются в приложении Compojure. Спецификация маршрутов немного напоминает код на предметно-ориентированном языке.

```
(GET "/" [] (page-hello-compojure))
```

Такие правила маршрутизации следует понимать как правила, которые пытаются найти совпадения, удовлетворяющие входящим запросом. Они очень легко членятся:

```
(<HTTP-метод> <URL> <params> <action>)
```

- *HTTP-метод* — обычно это `GET` или `POST`, но Compojure также поддерживает методы `PUT`, `DELETE` и `HEAD`. Метод должен соответствовать входящему запросу, если это правило должно быть соблюдено.
- *URL* — адрес, по которому направлен запрос. URL должен соответствовать входящему запросу, если это правило должно быть соблюдено.
- `params` — выражение, описывающее, как должны обрабатываться параметры. Чуть ниже мы подробнее об этом поговорим.
- `action` — выражение, которое следует возвратить (обычно имеет вид вызова функции с аргументами, которые необходимо передать), если это правило соблюдается.

Соответствие правилам проверяется в нисходящем порядке до тех пор, пока не будет найдено совпадение. Как только правило оказывается соблюдено, выполняется указанное действие, а значение выражения используется в качестве содержимого тега `<body>` в возвращаемом документе.

При указании правил Compojure отличается удивительной гибкостью. Например, будет очень просто создать правило, позволяющее извлекать параметр функции из фрагмента URL. Изменим маршруты из примера Hello World, показанного в листинге 13.5:

```
(defroutes main-routes
  (GET "/" [] (page-hello-compojure))
  (GET ["/hello/:fname", :fname #"[a-zA-Z]+" ]
➥ [fname] (page-hello-with-name fname))
  (route/resources "/")
  (route/not-found "Page not found"))
```

Это новое правило будет соблюдаться лишь в том случае, если браузер перейдет по URL, в адресе которого содержится `/hello/<name>`. Указанное имя `<name>` должно состоять из единой последовательности букв (в верхнем регистре, нижнем регистре

или обоих регистрах сразу). Такое ограничение накладывается с помощью регулярного выражения `Clojure #"[a-zA-Z]+"`.

Если правило соблюдается, то Compojure вызовет `(page-hello-with-name)` с совпавшим именем, указываемым в качестве параметра. Функция определяется очень просто:

```
(defn page-hello-with-name [fname]
  (str "<h1>Hello from Compojure " fname "</h1>"))
```

Хотя вы и можете писать внутристрочный HTML для очень простых приложений, такой подход быстро оказывается неудобным. К счастью, существует довольно простой модуль под названием Hiccup, предоставляющий полезный функционал для веб-приложений, в которых необходимо получать HTML на выходе. Мы рассмотрим Hiccup в следующем разделе.

13.6.3. Hiccup

Чтобы связать Hiccup с вашим приложением hello-compojure, нужно сделать три вещи.

1. Добавить соответствующую зависимость в `project.clj`, например, `[hiccup "0.3.4"]`.
2. Перезапустить `lein deps`.
3. Перезапустить веб-контейнер.

Это несложно. Теперь посмотрим, как можно пользоваться Hiccup для создания красивого HTML-кода из программы, написанной на Clojure.

Одна из основных форм, предоставляемых Hiccup, называется `(html)`. Прямо в ней можно писать HTML-код. Вот как нужно изменить `(page-hello-with-name)` для работы с Hiccup:

```
(defn page-hello-html-name [fname]
  (html [:h1 "Hello from Compojure " fname]
        [:div [:p "Paragraph text"]]))
```

Привычная структура из вложенных HTML-тегов теперь читается почти как настоящий код Clojure и очень удачно вписывается в этот язык. Форма `(html)` принимает один или несколько векторов (тегов) в качестве аргументов и допускает настолько глубокие структуры вложенных тегов, насколько требуется.

Далее мы изучим более крупное приложение — простой сайт, на котором вы можете проголосовать за любимую выдру.

13.7. Пример проекта с Compojure: «А не выдра ли я?»

Иногда кажется, что в Интернете никогда не переведутся только два вида контента — онлайновые опросы и картинки с милыми зверюшками. Допустим, вы поступили на работу на стартовый проект, который пытается зарабатывать на контекстной рекламе, а свою стратегию выстраивает на комбинировании двух этих трендов.

Посетители сайта могут голосовать за выкладываемые на ресурсе изображения выдр. Признаем, некоторые стартовые проекты базируются на куда более глупых идеях.

Для начала решим, какими будут основные страницы сайта и какой функционал должен реализовываться на сайте для голосования за выдр.

- На главной странице сайта пользователю нужно предложить выбрать из двух выдр.
- Пользователь должен будет проголосовать за одну из двух выдр, которых мы ему покажем.
- На отдельной странице нужно предоставить пользователю возможность загружать новые изображения выдр.
- На странице с информационной панелью должен отображаться актуальный рейтинг каждой из выдр, зависящий от количества поданных голосов.

На рис. 13.7 показано, как будут располагаться страницы и HTTP-запросы, обеспечивающие работу нашего приложения.

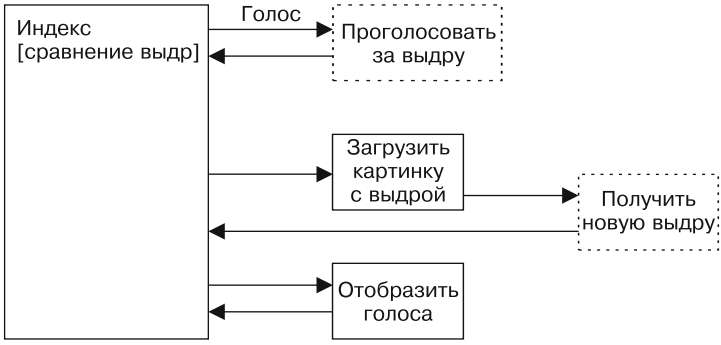


Рис. 13.7. Структура и взаимосвязи страниц на сайте «А не выдра ли я?»

Не менее важно учесть и нефункциональные требования, решением которых мы не будем заниматься в рамках данного приложения.

- Нас не интересует контроль доступа к сайту.
- Не будет никаких проверок корректности, которые могли бы применяться к загружаемым изображениям. Загружаемые картинки будут просто отображаться на странице, но ни их контент, ни безопасность не будут контролироваться. Мы доверяем нашим пользователям и уверены, что они не загрузят на сайт ничего неуместного.
- На сайте не будет предприниматься никаких дополнительных мер по сохранению информации. Если веб-контейнер откажет, все голоса потеряются. Но перед запуском приложение будет сканировать диск, чтобы заблаговременно заполнить хранилище с изображениями выдр.

На github.com есть готовая версия этого проекта, с которой вам, возможно, будет проще работать. В этой главе мы упомянем и обсудим все важные файлы проекта.

13.7.1. Настройка программы «А не выдра ли я?»

Чтобы запустить этот проект Compojure, нам понадобится определить базовый проект, его зависимости, перечень маршрутов и страничных функций. Для начала рассмотрим файл `project.clj` (листинг 13.10).

Листинг 13.10. Файл project.clj для проекта "А не выдра ли я?"

```
(defproject am-i-an-otter "1.0.0-SNAPSHOT"
  :description "Am I an Otter or Not?"
  :dependencies [[org.clojure/clojure "1.2.0"]
                 [org.clojure/clojure-contrib "1.2.0"]
                 [compojure "0.6.2"]
                 [hiccup "0.3.4"]
                 [log4j "1.2.15" :exclusions [javax.mail/mail
                                              javax.jms/jms
                                              com.sun.jdmk/jmxtools
                                              com.sun.jmx/jmxri]]
                 [org.slf4j/slf4j-api "1.5.6"]
                 [org.slf4j/slf4j-log4j12 "1.5.6"]]
  :dev-dependencies [[lein-ring "0.4.0"]]
  :ring {:handler am-i-an-otter.core/app})
```

Думаем, здесь все понятно. Все элементы, кроме библиотек log4j, уже встречались нам в предыдущих примерах.

Перейдем к подключению и маршрутизации логики в файле `core.clj` (листинг 13.1).

Листинг 13.11. Маршруты — core.clj

```
(ns am-i-an-otter.core
  (:use compojure.core)
  (:require [compojure.route :as route]
            [compojure.handler :as handler]
            [ring.middleware.multipart-params :as mp]))

(load "imports")
(load "otters-db")
(load "otters")

(defroutes main-routes
  (GET "/" [] (page-compare-otters))
  (GET ["/upvote/:id", :id #"[0-9]+" ] [id] (page-upvote-otter id))
  (GET "/upload" [] (page-start-upload-otter))
  (GET "/votes" [] (page-otter-votes))

  (mp/wrap-multipart-params
    (POST "/add_otter" req (str (upload-otter req) (page-start-upload-otter))))

  (route/resources "/")
  (route/not-found "Page not found"))

(def app
  (handler/site main-routes))
```

Это функции импорта

Основные маршруты

Обработчик загрузки файлов

Обработчик загрузки файлов предоставляет новый способ обращения с параметрами. Мы подробнее поговорим об этом в следующем подразделе, а пока считайте, что эта строка означает: «передаем целый HTTP-запрос к страничной функции для обработки».

Файл `core.clj` обеспечивает подключение кода и позволяет вам четко отслеживать, какие страничные функции к каким URL относятся. Как видите, названия всех страничных функций начинаются с `page` — это просто удобное соглашение об именовании.

В листинге 13.12 показаны страничные функции для нашего приложения.

Листинг 13.12. Страничные функции для приложения "А не выдра ли я?"

```
(ns am-i-an-otter.core
  (:use compojure.core)
  (:use hiccup.core))

(defn page-compare-otters []
  (let [otter1 (random-otter), otter2 (random-otter)]
    (.info (get-logger) (str "Otter1 = " otter1 " ; Otter2 = "
    ➥ otter2 " ; " otter-pics))
    (html [:h1 "Otters say 'Hello Compojure!'"]
          [:p [:a {:href (str "/upvote/" otter1)}
                  [:img {:src (str "/img/"
    ➥ (get otter-pics otter1))} ]]]
          [:p [:a {:href (str "/upvote/" otter2)}
                  [:img {:src (str "/img/"
    ➥ (get otter-pics otter2))} ]]]
          [:p "Click " [:a {:href "/votes"} "here"]
              " to see the votes for each otter"]
          [:p "Click " [:a {:href "/upload"} "here"]
              " to upload a brand new otter"])))

(defn page-upvote-otter [id]
  (let [my-id id]
    (upvote-otter id)
    (str (html [:h1 "Upvoted otter id=" my-id]) (page-compare-otters))))

(defn page-start-upload-otter []
  (html [:h1 "Upload a new otter"]
        [:p [:form {:action "/add_otter" :method "POST"
    ➥ :enctype "multipart/form-data"}
          [:input {:name "file" :type "file" :size "20"}]
          [:input {:name "submit" :type "submit" :value "submit"}]]]
        [:p "Or click " [:a {:href "/"} "here" ] " to vote on some otters"]))

(defn page-otter-votes []
  (let []
    (.debug (get-logger) (str "Otters: " @otter-votes-r))
    (html [:h1 "Otter Votes" ]
          [:div#votes.otter-votes
            (for [x (keys @otter-votes-r)]
              [:p [:img {:src (str "/img/" (get otter-pics x))} ]
    ➥ (get @otter-votes-r x)])])))
```

Сравнение для страницы с выдрами

Обработка поданных голосов

Выбор выдры для страницы загрузки

Настройка формы

Показ голосов

В этом листинге используется две полезные возможности Hiccup. Первая позволяет циклически перебрать группу элементов. В данном случае элементы — это загруженные выдры. Таким образом, функционально Hiccup во многом напоминает простой шаблонизатор, оснащенный встроенной формой `(for)` (см. следующий фрагмент кода):

```
[:div#votes.otter-votes
  (for [x (keys @otter-votes-r)]
    [:p [:img {:src (str "/img/" (get otter-pics x))} ]
➥ (get @otter-votes-r x)])]
```

Еще одна полезная особенность этого кода — синтаксис `:div#votes.otter-votes`. Он позволяет быстро указывать атрибуты `id` и `class` конкретного тега. В HTML эта конструкция выглядит как тег `<div class="otter-votes" id="votes">`. Соответственно, разработчик может выделить атрибуты, которые, скорее всего, будут применяться в CSS, не усложняя при этом структуру HTML-кода сверх меры.

Вообще, CSS и другой код (например, файлы исходников на JavaScript) будут предоставляться вне статического каталога с содержимым. По умолчанию для этого использовался бы каталог resources/public проекта Compojure.

ПОДБОР HTTP-МЕТОДОВ

В приложении для голосования за любимых выдр мы намеренно допустили архитектурную ошибку. На той странице, где посетитель отдает голос, в правиле маршрутизации мы указали, что при работе будет использоваться метод `GET`. Это неверно.

В приложении ни в коем случае нельзя применять `GET`-запросы для изменения состояния на серверной стороне (например, при работе с информацией о количестве голосов, отданных за любимую выдру). Это объясняется тем, что браузерам разрешено многократно посылать `GET`-запросы, если кажется, что сервер не отвечает (например, в момент поступления запроса сервер был приостановлен для сборки мусора). Из-за таких повторных попыток некоторые голоса будут дублироваться, даже если пользователь и проголосовал лишь один раз. А если такое случится в приложении электронного магазина, то результаты могут быть катастрофическими!

Просто усвойте следующее правило: любые значимые состояния сервера ни при каких условиях не должны изменяться под действием входящего запроса `GET`.

Итак, мы рассмотрели подключение приложения, настройку его маршрутов и страничных функций. Далее разберем кое-какие функции взаимодействия с базой данных, которые нам потребуется реализовать, чтобы организовать голосование за выдр.

13.7.2. Основные функции в программе «А не выдра ли я?»

Обсуждая основной функционал приложения при подготовке проекта, мы указали, что приложение должно просматривать каталог с картинками и находить любые изображения выдр, которые могут оказаться на диске. В листинге 13.13 приведен код, в котором просматриваются каталоги и выполняется предварительное заполнение.

Листинг 13.13. Функции просмотра каталогов

```
(def otter-img-dir "resources/public/img/")
(def otter-img-dir-fq
  (str (.getAbsolutePath (File. ".")) "/" otter-img-dir))

(defn make-matcher [pattern]
  (.getPathMatcher (FileSystems/getDefault) (str "glob:" pattern)))

(defn file-find [file matcher]
  (let [fname (.getName file (- (.getNameCount file) 1))]
    (if (and (not (nil? fname)) (.matches matcher fname))
      (.toString fname)
      nil)))

(defn next-map-id [map-with-id]
  (+ 1 (nth (max (let [map-ids (keys map-with-id)]
    (if (nil? map-ids) [0] map-ids))) 0 )))

(defn alter-file-map [file-map fname]
  (assoc file-map (next-map-id file-map) fname))

(defn make-scanner [pattern file-map-r]
  (let [matcher (make-matcher pattern)]
    (proxy [SimpleFileVisitor] []
      (visitFile [file attribs]
        (let [my-file file,
              my-attrs attribs,
              file-name (file-find my-file matcher)]
          (.debug (get-logger) (str "Return from file-find " file-name))
          (if (not (nil? file-name))
            (dosync (alter file-map-r alter-file-map file-name) file-map-r)
            nil)
          (.debug (get-logger)
    (str "After return from file-find " @file-map-r))
          FileVisitResult/CONTINUE))

        (visitFileFailed [file exc] (let [my-file file my-ex exc]
          (.info (get-logger)
            (str "Failed to access file " my-file " ; Exception: " my-ex))
          FileVisitResult/CONTINUE)))))

(defn scan-for-otters [file-map-r]
  (let [my-map-r file-map-r]
    (Files/walkFileTree (Paths/get otter-img-dir-fq
➥   (into-array String [])) (make-scanner "*.jpg" my-map-r))
    my-map-r))

(def otter-pics (deref (scan-for-otters (ref {}))))
```

При обнаружении совпадения возвращаем усеченное имя файла

Используем (toString) для обеспечения работы тегов :img

Получаем ID следующей выдры

Изменяем функцию и добавляем имя файла в словарь

Возвращаем просмотрщик каталогов

Обратный вызов, выполняется для каждого файла

Задаем картинки выдр

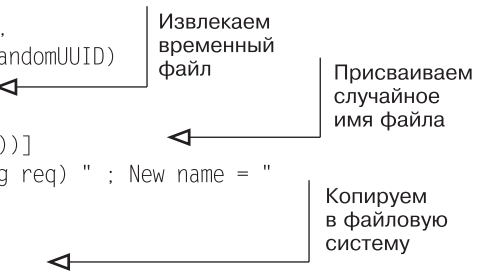
Точка входа для этого кода — (scan-for-otters). Здесь мы используем класс Files из Java 7 для прохода по файловой системе, начиная с otter-img-dir-fq. По завершении этого процесса возвращается ссылка на словарь. В этом коде применяется простое соглашение о том, что имя символа, оканчивающееся на -r, является ссылкой на интересующую нас структуру.

Код, который используется для прохода по файлам, — это посредник для класса SimpleFileVisitor, написанный на Clojure (класс SimpleFileVisitor взят из пакета java.nio.file), рассмотренного в главе 2. Вы предоставляете специальные реализации двух методов — (visitFile) и (visitFileFailed), — для данного случая их достаточно.

Кроме того, очень интересны функции, реализующие возможность голосования. Они приведены в листинге 13.14.

Листинг 13.14. Функции голосования за выдр

```
(def otter-votes-r (ref {}))

(defn otter-exists [id] (contains? (set (keys otter-pics)) id))

(defn alter-otter-upvote [vote-map id]
  (assoc vote-map id (+ 1 (let [cur-votes (get vote-map id)]
    (if (nil? cur-votes) 0 cur-votes)))))

(defn upvote-otter [id]
  (if (otter-exists id)
    (let [my-id id]
      (.info (get-logger) (str "Upvoted Otter " my-id))
      (dosync (alter otter-votes-r alter-otter-upvote my-id)
➥ otter-votes-r))
      (.info (get-logger) (str "Otter " id " Not Found " otter-pics))))

(defn random-otter [] (rand-nth (keys otter-pics)))

(defn upload-otter [req]
  (let [new-id (next-map-id otter-pics),
        new-name (str (java.util.UUID/randomUUID)
➥ ".jpg"),
        tmp-file (:tempfile
➥ (get (:multipart-params req) "file"))]
    (.debug (get-logger) (str (.toString req) " ; New name = "
➥ new-name " ; New id = " new-id))
    (ds/copy tmp-file (ds/file-str
➥ (str otter-img-dir new-name)))
    (def otter-pics (assoc otter-pics new-id new-name))
    (html [:h1 "Otter Uploaded!"])))
```



В функции (upload-otter) мы работаем со словарем HTTP-запросов. В нем содержится много полезной информации, к которой может обращаться веб-разработчик. Вероятно, некоторые из этих данных вам уже знакомы:

```
{:remote-addr "127.0.0.1",
 :scheme :http,
```

```
:query-params {},
:session {},
:form-params {},
:multipart-params {"submit" "submit", "file" {:filename "otter_kids.jpg",
   :size 122017, :content-type "image/jpeg", :tempfile #<File /var/tmp/
   upload_646a7df3_12f5f51ff33__8000_00000000.tmp>}},
:request-method :post,
:query-string nil,
:route-params {},
:content-type "multipart/form-data; boundary=----
   WebKitFormBoundaryvKKZehApamWrVFt0",
:cookies {},
:uri "/add_otter",
:server-name "127.0.0.1",
:params {:file {:filename "otter_kids.jpg", :size 122017, :content-type
   "image/jpeg", :tempfile #<File /var/tmp/
   upload_646a7df3_12f5f51ff33__8000_00000000.tmp>}, :submit "submit"},
:headers {"user-agent" "Mozilla/5.0 (Macintosh; U; Intel Mac OS X 10_6_6;
   en-US) AppleWebKit/534.16 (KHTML, like Gecko) Chrome/10.0.648.205
   Safari/534.16", "origin" "http://127.0.0.1:3000", "accept-charset" "ISO-
   8859-1,utf-8;q=0.7,*;q=0.3", "accept" "application/xml,application/
   xhtml+xml,text/html;q=0.9,text/plain;q=0.8,image/png,*/*;q=0.5", "host"
   "127.0.0.1:3000", "referer" "http://127.0.0.1:3000/upload", "contenttype"
   "multipart/form-data; boundary=----
   WebKitFormBoundaryvKKZehApamWrVFt0", "cache-control" "max-age=0",
   "accept-encoding" "gzip,deflate,sdch", "content-length" "122304",
   "accept-language" "en-US,en;q=0.8", "connection" "keep-alive"},
:content-length 122304,
:server-port 3000,
:character-encoding nil,
:body #<Input org.mortbay.jetty.HttpParser$Input@206bc833>}
```

Из этого словаря запросов понятно, что контейнер уже закачал содержимое входящего файла во временный файл в `/var/tmp`. Вы можете получить доступ к объекту `File`, соответствие с которым устанавливается путем `(:tempfile (get (:multipart-params req) "file"))`. Потом достаточно воспользоваться функцией `(copy)` из `clojure.contrib.duck-streams`, чтобы сохранить информацию в файловой системе.

Приложение для голосования за выдр получилось маленьким, но полным. Учитывая разграничение функциональных и нефункциональных требований, которые мы определили в начале этого раздела, приложение работает именно так, как было задумано. На этом мы завершаем наше знакомство с Compojure и некоторыми библиотеками данного фреймворка.

13.8. Резюме

Быстрая веб-разработка — это навык, которым должен обладать любой профессиональный Java-разработчик. Но если не угадать с выбором языка или фреймворка, то ваше решение быстро начнет проигрывать другим, основанным на не относящихся к Java/JVM технологиях — например, на Rails или PHP. В частности, Java

не очень подходит для веб-разработки в силу своей статической и компилируемой природы. Но если язык и фреймворк подобраны правильно, то вы сможете быстро разрабатывать новые функции, ничуть не жертвуя их качеством. Вы окажетесь на вершине «пищевой цепи» веб-разработчиков и сумеете оперативно реагировать на пожелания ваших пользователей.

Как основательный Java-разработчик, вы, разумеется, не хотите отказываться от силы и гибкости виртуальной машины Java. К счастью, на платформе JVM сейчас бурно развиваются новые языки и фреймворки, отказываться от чего-либо и не требуется! Фреймворки, относящиеся к динамическому уровню, например Grails и Compojure, предоставляют вам все возможности, необходимые для быстрой веб-разработки.

В частности, в Grails можно очень быстро создавать полнофункциональные прототипы (от пользовательского интерфейса до базы данных), а потом дополнить части этого стека мощной технологией работы с представлениями (GSP), технологией долговременного хранения (GORM), а также множеством полезных плагинов.

Фреймворк Compojure отлично подходит для работы с проектами, которые уже написаны на языке Clojure. Этот фреймворк позволяет добавлять небольшие веб-компоненты, например информационные панели и пульты управления, в проекты, написанные на Java или другом языке. Основные достоинства Compojure — чистота кода и высокая скорость разработки.

На этом мы заканчиваем серию глав, в которой рассматривали примеры многоязычного программирования на виртуальной машине Java. В последней главе мы подытожим весь изученный материал и заглянем за пределы уже разобранных тем.

Вам еще многому предстоит научиться, но теперь вы хорошо подготовлены к дальнейшему пути.

14 О сохранении основательности

В этой главе:

- что ждет разработчиков в Java 8;
- будущее многоязычного программирования;
- в каком направлении развивается параллельная обработка;
- новые возможности на уровне виртуальной машины Java.

Чтобы на шаг опережать актуальные тенденции, основательный Java-разработчик всегда должен знать, как развиваются новые технологии в его профессиональной сфере. В последней главе нашей книги мы затронем несколько тем, которые, на наш взгляд, позволяют заглянуть в будущее Java — и языка, и платформы.

Поскольку у нас нет машины времени и гадать на кофейной гуще мы не умеем, в этой главе речь пойдет только о тех функциональных изменениях языка и платформы, которые уже разрабатывались на момент подготовки этой книги. Таким образом, у нас неизбежно получится лишь временной срез — мы аккуратно намекаем, что наш прогноз может быть не слишком объективным.

Мы говорим лишь о возможном будущем, о том, каким оно видится нам на момент написания книги. А реальность может оказаться и иной. Более того, она, несомненно, будет отличаться от описанной ниже картины в тех или иных важных аспектах, а будущее вообще обещает быть интересным. Оно всегда интересное.

Итак, рассмотрим нашу первую подтему — важнейшие новые возможности, которые должны появиться в Java 8.

14.1. Чего ожидать в Java 8

Осенью 2010 года Исполнительный комитет по разработке Java SE на очередном заседании решил действовать по плану B. Этот план заключался в том, чтобы выпустить версию Java 7 в максимально сжатые сроки, а реализацию некоторых крупных нововведений отложить до Java 8. Такое решение было принято после подробных консультаций и опросов разработчиков о том, какой вариант кажется им предпочтительным.

Отдельные функции, которые изначально планировалось включить в Java 7, оказались сдвинуты на Java8, а несколько других было решено выполнить в сжатом

виде, чтобы они фактически сформировали базис для возможностей, ожидаемых в 8-й версии. В этом разделе мы кратко охарактеризуем основные новинки, планируемые в Java 8, в частности те функции, которые ранее были отложены. На данном этапе ничего еще не определено окончательно, в особенности это касается языкового синтаксиса. Все примеры кода являются предварительными и могут сильно отличаться от аналогичных в готовой версии Java 8.

14.1.1. Лямбда-выражения (замыкания)

Хорошими примерами характерных черт Java 7, которые будут достраиваться в Java 8, являются `MethodHandles` и `invokedynamic`. Эти функции полезны и сами по себе (в Java 7 `invokedynamic` больше заинтересует тех, кто занимается разработкой языков и фреймворков).

В Java 8 на основе этих возможностей будут реализованы лямбда-выражения — впервые в языке Java. Лямбда-выражения немного напоминают функциональные литералы, с которыми вы познакомились при изучении альтернативных языков для JVM. На практике они будут применяться для решения примерно таких же проблем, для которых в предыдущих главах мы использовали функциональные литералы.

Что касается языкового синтаксиса Java 8, еще предстоит решить, как именно лямбда-выражения будут выглядеть в коде. Но их основные черты уже определены, поэтому рассмотрим базовый синтаксис Java 8 (листинг 14.1).

Листинг 14.1. Преобразование Шварца на языке Java с применением лямбда-выражений

```
public List<T> schwarz(List<T> x, Mapper<T, V> f) {
  return x.map(w -> new Pair<T,V>(w, f.map(w)))
          .sorted((l,r) -> l.hashed.compareTo(r.hashed))
          .map(l -> l.orig).into(new ArrayList<T>());
}
```

Метод `schwartz()` должен показаться знакомым — это реализация преобразования Шварца, рассмотренного в разделе 10.3, которое там было сделано на языке Clojure. В листинге 14.1 показаны базовые синтаксические возможности лямбда-выражений в Java 8:

- здесь есть список параметров для предполагаемого лямбда-выражения;
- в скобках записан блок, заключающий в себе тело лямбда-выражения;
- стрелка (->) отделяет список параметров от тела лямбда-выражения;
- типы параметров для списка аргументов узнаются при выведении.

В главе 9 мы поговорили о функциональных литералах Scala, которые очень напоминают лямбда-выражения. Поэтому такой синтаксис не должен сильно вас озадачивать. В листинге 14.1 лямбда-выражения очень короткие — в каждом всего по одной строке. Но они могут быть и многострочными, и даже иметь большие тела. Предварительный анализ отдельных довольно объемных блоков кода, которые можно переписать с применением лямбда-выражений, показывает, что тело большинства лямбда-выражений будет укладываться в 1–5 строк.

В листинге 14.1 показана еще одна новая черта. Это переменная `x` типа `List<T>`. При выполнении кода мы применяем метод `map()` к переменной `x`. Метод `map()` принимает лямбда-выражение в качестве аргумента. Но постойте! Ведь у интерфейса `List` нет метода `map()`, а в Java 7 и ранее не было никаких лямбда-выражений.

Подробнее рассмотрим, как можно решить эту проблему.

Методы расширения и базовые методы

В сущности, стоящая переда нами проблема такова: как добавлять методы к уже существующим интерфейсам, чтобы «лямбдировать» их, не нарушая при этом обратной совместимости?

Проблема решается с помощью нового элемента Java — *методов расширения* (extension method). Такие сущности предоставляют и стандартный метод, который может использоваться в том случае, если в реализации конкретного интерфейса не предусмотрена реализация нового метода расширения.

Такие стандартные реализации методов должны определяться внутри самого интерфейса.

Например, `List` сочетается с `AbstractList`, `Map` — с `AbstractMap`, а `Queue` — с `AbstractQueue`. Эти классы идеально подходят для хранения стандартных реализаций любых новых методов расширений для соответствующих интерфейсов. Встроенные в Java классы коллекций — как раз тот материал, на котором можно активно пользоваться как методами расширений, так и «лямбдафикацией», но нам эта модель представляется уместной для применения и в коде для конечного пользователя.

КАК В JAVA МОЖНО РЕАЛИЗОВЫВАТЬ МЕТОДЫ РАСШИРЕНИЙ

Методы расширений обрабатываются на этапе загрузки классов. Когда новая реализация интерфейса с методами расширений будет загружена, загрузчик классов попробует проверить, определены ли в нем собственные реализации методов расширений. Если это не так, то загрузчик классов найдет стандартный метод и вставит в байт-код метод-мостик для связи с только что загруженным классом. Метод-мостик будет вызывать стандартную реализацию с помощью `invokedynamic`.

Методы расширений обеспечивают важную новую возможность — теперь интерфейсы можно развивать уже после выпуска их стартовых версий, не нарушая обратной совместимости. Таким образом, обогащаясь лямбда-выражениями, старые API получают новые жизненные силы. Но как лямбда-выражения воспринимаются на уровне виртуальной машины Java? Это объекты? А если не объекты, то что же?

SAM-преобразование

Лямбда-выражения — это компактные конструкции, позволяющие объявить небольшой объем кода прямо внутри строки и передавать этот код так, как если бы он был данными. Соответственно, лямбда-выражение — это объект, примерно такой, как функциональный литерал (мы говорили о лямбда-выражениях и функциональных литералах в части 3 книги, когда обсуждали альтернативные языки для виртуальной машины Java). В частности, лямбда-выражение можно считать подклассом

Object; у такого подкласса не будет параметров (и, соответственно, состояния), а будет всего один метод.

Этот метод удобно представлять как единственный абстрактный метод (SAM). Концепция SAM развилась в результате изучения различных существующих API языка Java — было замечено, что у многих API есть одна общая черта. Зачастую API имеет интерфейс, в котором указывается всего один метод. Runnable, Comparable, Callable и слушатели, например ActionListener, — все определяют только по одному методу и, соответственно, относятся к категории SAM.

Приступая к работе с лямбда-выражениями, можете считать их своеобразным синтаксическим сахаром. Лямбда-выражение — это, в сущности, сокращенный вариант написания анонимной реализации того или иного интерфейса. Со временем вы научитесь включать в код все больше функциональных приемов, возможно, даже начнете импортировать любимые трюки из Scala и Clojure в код Java. Обучение функциональному программированию — это обычно постепенный процесс, поскольку на первом этапе требуется освоить работу с отображением, сортировкой и фильтрацией коллекций, а потом приобретать более сложные навыки на этой основе.

Обратимся к следующей важной теме: обсудим программу модуляризации, которая реализуется в рамках проекта Jigsaw.

14.1.2. Модуляризация (проект Jigsaw)

Согласитесь, работать с путями к классам в Java порой бывает достаточно неудобно. Существуют широко известные проблемы, связанные с экосистемой и возникающие при обращении с JAR-файлами и путями к классам:

- сам JRE достаточно массивен;
- JAR-файлы требуют использовать монолитные модели развертывания;
- приходится загружать слишком много мусора и классов, которые могут пригодиться лишь изредка;
- запуск медленный;
- пути к классам — очень «хрупкие» конструкции, к тому же они тесно связаны с файловой системой конкретной машины;
- как правило, путь к классу — это плоское пространство имен;
- JAR-файлы не имеют строгой системы версий и плохо вписываются в такие системы;
- возникают сложные взаимные зависимости даже между такими классами, которые логически никак не связаны.

Для решения всех этих проблем требуется новая модульная система. Но существуют и архитектурные вопросы, которые также необходимо решить. Важнейшие из них проиллюстрированы на рис. 14.1.

Следует ли нам произвести начальную загрузку виртуальной машины, а потом задействовать систему модулей пользовательского пространства, например OSGi, либо работать со стопроцентно модульной платформой?

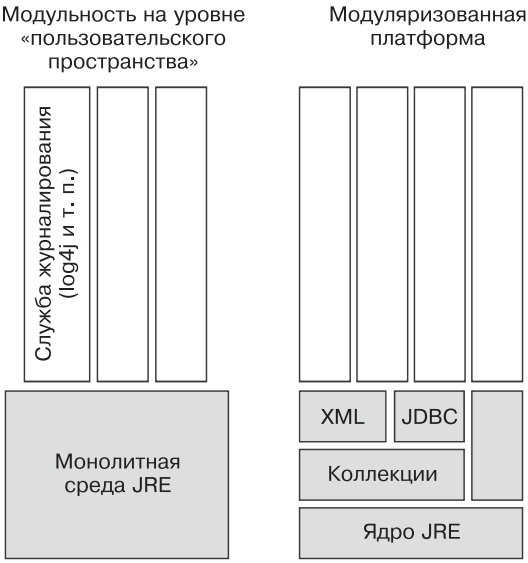


Рис. 14.1. Варианты организации архитектуры в модульной системе

Во втором случае потребуется загрузить минимальный вариант виртуальной машины, приспособленный к работе с модулями, так называемое ядро, а потом надстроить над ним лишь те модули, которые нужны для работы конкретного приложения, запускаемого в настоящий момент. В такой ситуации придется внести коренные изменения и в виртуальную машину, и во многие имеющиеся классы JRE, но потенциально мы можем получить кардинальные преимущества.

- Приложения для виртуальной машины Java смогут соперничать по скорости запуска с командами оболочки и сценарными языками.
- Развертывание приложений должно значительно упроститься.
- Область памяти, занимаемая Java-компонентами, при узкоспециальных вариантах установки может быть существенно снижена (со всеми вытекающими положительными последствиями для диска, памяти и безопасности). Если вам не требуется CORBA или RMI — зачем их устанавливать?
- Обновление и усовершенствование установок Java становятся гораздо более гибкими. Если в области коллекций будет найдена критическая ошибка, то потребуется обновить лишь этот модуль.

На момент написания книги складывается впечатление, что проект Jigsaw будет развиваться по второму сценарию. Тем не менее до полной готовности и релиза проекта Jigsaw должно пройти немало времени. Остается решить еще немало вопросов, важнейшие из которых таковы.

- Какова должна быть оптимальная единица для распространения платформ и приложений?
- Нужна ли новая сущность, которая будет отличаться как от пакета, так и от JAR?

Последствия от принятия того или иного решения будут всепроникающими — ведь Java уже *везде*, а модульный дизайн должен масштабироваться в таком исключительно широком пространстве. Кроме того, модульная архитектура должна работать на разных платформах и в операционных системах.

Платформа Java должна обеспечивать возможность развертывания модульных приложений как минимум в следующих операционных системах: Linux, Solaris, Windows, Mac OS X, BSD UNIX и AIX. На некоторых из них есть менеджеры пакетов, с которыми Java придется интегрироваться (в их числе `apt` для Debian, `rpm` для Red Hat, а также пакеты Solaris). В других операционных системах, например в Windows, отсутствуют такие механизмы для управления пакетами, с которыми обязательно должен взаимодействовать язык Java.

При дизайне требуется учитывать и другие ограничения. В этой области уже существует несколько зрелых самостоятельных проектов, например системы управления зависимостями Maven и Ivy, а также инициатива OSGi. Новая модульная система должна интегрироваться со всеми вышеупомянутыми системами, если это в принципе реально, а также обеспечивать удобные возможности усовершенствования системы в том случае, если в актуальном состоянии полная интеграция и совместимость с системой нереализуемы.

Независимо от того, что ожидает нас впереди, в Java 8 нас ждут революционные изменения в области передачи и развертывания приложений.

Рассмотрим кое-какие перспективы, которые JDK 8 позволит воплотить на других языках виртуальной машины Java, в частности на тех, о которых мы говорили в предыдущих главах.

14.2. Многоязычное программирование

Как мы уже неоднократно упоминали, виртуальная машина Java — превосходная база для развертывания среды времени исполнения языка программирования. Проект OpenJDK, о котором шла речь в главе 1, в ходе работ по релизу Java 7 стал базовой реализацией для языка Java. Благодаря этому наблюдается очень интересное побочное явление: виртуальная машина Java языковонезависима (language-agnostic), а значит — идеальна для многоязычных разработок.

В частности, после выхода седьмой версии язык Java утратил привилегированные позиции на этой виртуальной машине. Все языки на платформе JVM теперь равноправны. В результате возник отчетливый интерес к дополнению виртуальной машины такими функциями, которые очень важны в альтернативных языках, а в Java играют незначительную роль.

Множество подобной работы выполняется в рамках подпроекта, который называется Da Vinci Machine, или mlvm (что означает *«многоязычная виртуальная машина»*). Возможности, которые «вынашиваются» в этом подпроекте, перемещаются в основное дерево исходников. Например, именно так сложилась судьба функции `invokedynamic`, которую мы рассматривали в разделе 5.5, но есть и другие

черты, которые будут особенно полезны именно в альтернативных языках. Остается и немало нерешенных проблем.

Познакомимся с первой из подобных инноввационных возможностей. Она заключается в обеспечении бесшовной коммуникации различных языков, которые могут обмениваться информацией друг с другом, работая на одной и той же JVM.

14.2.1. Межъязыковые взаимодействия и метаобъектные протоколы

Равноправие языков — это значительный шаг к созданию тепличных условий для многоязычного программирования, но еще остается решить некоторые досадные проблемы. Главнейшая из них сводится к тому тривиальному факту, что разные языки имеют неодинаковые системы типов. Строки в Ruby являются изменяемыми, а в Java — неизменяемыми. Scala считает объектами все сущности, даже такие, которые в Java воспринимались бы как примитивы.

Сглаживание этих различий и обеспечение более удобных способов обмена информацией и взаимодействия между различными языками в рамках одного экземпляра виртуальной машины Java в настоящее время представляет собой большую нерешенную проблему, причем ее желательно устранить как можно скорее.

Представьте себе веб-приложение, которое вы, возможно, будете разрабатывать в недалеком будущем. Его ядро может быть написано на Java, веб-уровень — на Clojure (с применением фреймворка Compojure). Кроме того, в приложении, вероятно, будет использоваться библиотека для обработки JSON, написанная на чистом JavaScript. А теперь представьте, что вам нужно все это протестировать и вы хотите воспользоваться замечательными возможностями разработки через тестирование, предоставляемыми во фреймворке ScalaTest.

В таком случае может возникнуть ситуация, в которой сразу несколько языков — JavaScript, Clojure, Scala и Java — могут напрямую обращаться друг к другу. В результате возникает необходимость межъязыковых взаимодействий и требуется обеспечить стандартизированный способ, которым языки JVM будут вызывать объекты из других языков, работающих на этой машине. Со временем такая необходимость будет только обостряться. Сообщество разработчиков пришло к общему мнению о том, что для решения таких проблем потребуется метаобъектный протокол (MOP), который бы обеспечил стандартизацию всех этих разнообразных операций. Метаобъектный протокол можно считать неким посредником, который мог бы описать на уровне кода, как именно тот или иной язык реализует объектную ориентацию и связанные с ней функции.

Чтобы достичь этой цели, требуется найти способы, которыми можно было бы использовать объекты из одного языка в другом языке. Один из простых способов решения такой задачи — приводить конкретный тип к тому типу, который является нативным для чуждого языка (можно даже попробовать создавать «теневой»

объект в среде времени исполнения чуждого языка). Но этот метод прост только на первый взгляд. На самом деле он сопряжен с серьезными проблемами.

- В каждом из взаимодействующих языков должен присутствовать общий ведущий интерфейс (или суперкласс), реализуемый всеми типами внутри данной версии языка (как `IRubyObject` в JRuby).
- Если прибегать к использованию теневых объектов, то они потребуют многочисленных операций по выделению памяти, что будет негативно сказываться на производительности.

Другой способ — написать службу, которая будет действовать как входная точка в среду времени исполнения чуждого языка. Такая служба будет предоставлять интерфейс, позволяющий одной среде времени исполнения выполнять стандартные операции над объектами чуждой среды, в частности:

- создавать новый объект в чуждой среде времени исполнения и возвращать ссылку на него;
- получать доступ к свойству (читать его или устанавливать) чуждого объекта;
- вызывать метод применительно к чужеродному объекту;
- приводить чужеродный объект к иному релевантному типу;
- получать доступ к дополнительным возможностям чужеродного объекта, которые могут обладать иной семантикой, нежели вызов метода в отдельных языках.

В подобной системе для доступа к чужеродному методу или свойству можно выполнять вызов к «навигатору», действующему в чужой среде времени исполнения. Вызывающая сторона должна будет предоставлять способ идентификации метода, к которому мы обращаемся, — `someMethod`. Обычно в данном случае речь идет о строке, но иногда это может быть и указатель на метод — `MethodHandle`.

```
navigator.callMethod(someObject, someMethod, param1, param2, ...);
```

Чтобы обеспечить эффективность такого подхода, интерфейс `Navigator` в средах времени исполнения должен быть идентичным для всех взаимодействующих языков. На внутрисистемном уровне фактические связи между языками будут выстраиваться, вероятно, с применением `invokedynamic`.

Итак, посмотрим, как могла бы выглядеть многоязычная виртуальная машина Java, использующая среди прочего и систему модулей из Java 8.

14.2.2. Многоязычная модуляризация

Появление проекта Jigsaw и возможностей модуляризации платформы — важные нововведения, которые будут полезны не только в языке Java. Другие языки также смогут поучаствовать в таких разработках.

Логично предположить, что навигационные интерфейсы и вспомогательные классы будут образовывать один модуль, а поддержка времени исполнения для конкретных альтернативных языков будет предоставляться в одном или нескольких модулях. На рис. 14.2 показано, как может функционировать система модулей.

Как видите, с помощью системы модулей можно построить самодостаточное многоязычное приложение. Модуль Clojure предоставляет базовую платформу Clojure, а модуль Compojure добавляет компоненты, необходимые для запуска стека веб-приложения, в частности специфические версии JAR-файлов, различные варианты которых могут применяться где-либо в рабочем процессе. Здесь присутствуют Scala и XML-стек, а также модуль Navigator, обеспечивающий взаимодействие между языками Scala и Clojure.

Рис. 14.2. Система модулей, реализующих многоязычное решение

В следующем разделе мы рассмотрим еще одну тенденцию современного программирования, обусловленную бурным развитием альтернативных языков для виртуальной машины Java на нашей платформе. Речь пойдет о параллелизме.

14.3. Будущие тенденции параллелизма

Компьютерное оборудование XXI века не всегда удобно использовать с языками программирования, появившимися в XX веке. Это эмпирическое наблюдение, на которое мы уже не раз исподволь указывали в нашей книге. В подразделе 6.3.1 мы говорили о законе Мура, описывающем рост количества транзисторов на микросхеме, а также вскользь затронули одно важное следствие из этого закона. Мы имеем в виду взаимосвязь между законом Мура, производительностью и параллелизмом. Об этом мы и поговорим далее.

14.3.1. Многоядерный мир

В то время как количество транзисторов на кристалле микросхемы увеличивалось экспоненциально, в точном соответствии с прогнозами, скорость доступа к памяти так сильно не выросла. В 1990-е и начале 2000-х разработчики микрочипов пытались справиться с этой проблемой, задействуя большее количество доступных транзисторов для компенсаци низкой скорости работы с памятью.

В главе 6 мы говорили, что такой подход позволяет наладить стабильный поток данных, поступающих на ядра процессора и постепенно обрабатываемых там. Но это тупиковый путь: польза, извлекаемая от применения транзисторов для повышения скорости работы основной памяти, становится все более незначительной. Дело

в том, что применяемые уловки (например, параллелизм на уровне инструкций и упреждающее исполнение команд) уже не дают быстрого эффекта и должны становиться все более интеллектуальными.

В последние годы все больше внимания уделяется использованию транзисторов для того, чтобы создавать процессоры с многочисленными ядрами. Теперь практически на всех обычных ПК и ноутбуках уже установлены процессоры с двумя ядрами. Распространены и четырехъядерные, и восьмиядерные процессоры. В сложных специализированных серверных установках встречается по 6–8 ядер на процессор, а вся машина может содержать и 32 ядра, и даже больше. Мы уже давно живем в многоядерном мире — и чтобы пользоваться его благами, нам нужны программы, написанные в менее линейном стиле. Таким программам требуется и соответствующая поддержка на уровне языка и среды времени исполнения.

14.3.2. Параллельная обработка, управляемая во время исполнения

Мы уже вкратце рассмотрели возможные перспективы параллельного программирования. В Scala и Clojure мы поговорили о таких вариантах обеспечения параллелизма, которые значительно отличаются от применяемой в Java модели потоков и блокировок. Так, в Scala параллелизм реализуется с помощью акторов, а в Clojure для этого используется программная транзакционная память.

Акторная модель Scala обеспечивает обмен сообщениями между выполняемыми блоками кода, причем эти блоки кода вполне могут исполняться на совершенно разных ядрах (есть даже расширения, позволяющие использовать удаленные акторы). Соответственно, если код написан с четким акцентом на акторах, он потенциально может без значительных проблем масштабироваться на многоядерной машине.

В Clojure есть агенты — сущности, занимающие практически ту же нишу, что и акторы Scala. Но в Clojure к тому же есть и разделяемые данные, которые можно изменить лишь в рамках транзакции, протекающей внутри памяти. Здесь задействуется уже упоминавшийся механизм программной транзакционной памяти.

В обоих рассмотренных случаях четко виден краеугольный камень новой концепции — управление параллелизмом возлагается на среду времени исполнения, а не на самого разработчика. Да, виртуальная машина Java обеспечивает диспетчеризацию потоков в рамках предоставления низкоуровневых служб, но на этой платформе совершенно нет высокоуровневых конструкций для управления параллельными программами.

Такие недостатки языка Java очевидны. Фактически он открывает программисту доступ к низкоуровневой модели обеспечения параллелизма, действующей на виртуальной машине Java.

Учитывая, как много кода уже написано на Java, будет очень сложно создать какой-либо совершенно новый механизм для этого языка, обеспечить широкое употребление такого механизма на практике, а также гарантировать бесшовное

взаимодействие с уже имеющимся кодом. Вот почему при поиске новых направлений развития параллелизма такое пристальное внимание уделяется альтернативным языкам для виртуальной машины Java. Такие языки имеют две важные черты:

- в качестве низкоуровневой модели работы с памятью в них применяется JMM (модель памяти для Java);
- в этих языках используется языковая среда исполнения, способная работать «с чистого листа»; такая среда может предоставлять иные (и более разнообразные) абстракции, чем предлагает язык Java.

НЕ ТРЕБУЙТЕ СЛИШКОМ МНОГОГО ОТ ПАРАЛЛЕЛИЗМА JAVA

Когда Java вышел в 1996 году, это был один из первых крупных языков, который изначально создавался в расчете на написание параллельного кода. Сегодня, оглядываясь на пятнадцатилетнюю историю широкого применения Java в промышленных масштабах, можно не сомневаться, что модель изменяемых данных с применяемым по умолчанию разделением состояния, а также теми исключениями, без которых невозможно пользоваться блокировками, — неидеальна. Но разработчики, которые создавали язык Java 1.0, не имели возможности изучить такую ретроспективу. Во многих отношениях именно первичная модель параллелизма, появившаяся в Java, помогла нам выйти на тот уровень, который есть сейчас.

Вполне возможно, что вся эта дополнительная поддержка параллелизма может появиться на уровне виртуальной машины (об этом мы поговорим в следующем разделе). Но пока основное направление развития инноваций лежит в плоскости выстраивания новых языков на базе надежной модели JMM, а не в контексте внесения низкоуровневых изменений в фундаментальную модель многопоточности.

Несомненно, на платформе JVM есть области, которые могут измениться в JDK 8 и далее. Некоторые из подобных изменений могут быть обусловлены дальнейшей разработкой `invokedynamic`, начавшейся в Java 7. О них мы и поговорим далее.

14.4. Новые направления в развитии виртуальной машины Java

В главе 1 мы познакомились с VMSpec — спецификацией виртуальной машины Java. Это документ, в котором четко описано, как именно должна работать виртуальная машина, чтобы ее можно было считать реализацией стандарта JVM. Если в машину добавляется новое поведение (например, `invokedynamic` в Java 7), то все реализации должны быть усовершенствованы и включать поддержку новых возможностей.

В этом разделе мы обсудим возможные изменения виртуальной машины Java, которые в настоящее время обсуждаются и прототипируются. Эта работа идет в рамках проекта OpenJDK, который, в частности, стал основой для базовой реализации Java и исходной точкой для JDK Oracle. Мы рассмотрим не только возможные изменения спецификации, но и значительные изменения базы кода OpenJDK/Oracle JDK.

14.4.1. Конвергенция виртуальных машин

После того как компания Oracle приобрела Sun Microsystems, в активе Oracle оказались две очень мощные виртуальные машины Java: HotSpot VM (унаследованная от Sun) и jRockit (попавшая в Sun в результате более раннего приобретения компании BEA).

Вскоре был сделан вывод, что продолжение поддержки обеих виртуальных машин — напрасная трата ресурсов. Поэтому компания Oracle решила объединить две эти виртуальные машины. В качестве основы была выбрана машина HotSpot, а функции jRocket тщательно портируются на нее в последующих релизах Java.

А КАК ЖЕ ОНА НАЗЫВАЕТСЯ?

Общая виртуальная машина пока не имеет официального названия, но энтузиасты из Java-сообщества уже прозвали ее HotRockit. Несомненно, это удачное название. Остается дожидаться, одобрит ли его маркетинговый отдел Oracle.

Итак, почему же все это важно для разработчика? Современная виртуальная машина, которой вы пользуетесь сегодня (скорее всего, это HotSpot), постепенно будет обрастать множеством новых функций, среди которых нужно особо отметить такие, как:

- *удаление постоянного поколения памяти* — в результате удастся избавиться от большого количества сбоев, возникающих на этапе загрузки классов;
- *улучшенная поддержка JMX-агента* — благодаря этому изменению вы сможете тщательнее контролировать функции работающей виртуальной машины;
- *новый подход к динамической компиляции* — в машину добавлены оптимизации, привнесенные из кода проекта jRockit;
- *управление миссиями* — предоставляет богатый инструментарий для настройки и профилирования приложений для релиза; некоторые из этих инструментов будут дополнительными платными компонентами виртуальной машины, недоступными для обычного скачивания.

УДАЛЕНИЕ ПОСТОЯННОГО ПОКОЛЕНИЯ ПАМЯТИ

В подразделе 6.5.2 мы говорили о том, что метаданные ваших классов содержатся в особом разделе памяти, выделяемом в виртуальной машине (это так называемое *постоянное поколение памяти* — PermGen). Это пространство может быстро заполниться до основания, особенно в альтернативных языках и фреймворках, которые создают много новых классов во время исполнения. Постоянное поколение памяти не высвобождается автоматически, и если оно переполнится, то работа виртуальной машины будет аварийно завершена. Ведутся работы по переносу хранения метаданных в нативную память — когда они будут закончены, страшное сообщение `java.lang.OutOfMemory Error: PermGen space` станет достоянием истории.

Разрабатывается и еще множество мелких изменений, и все они нацелены на то, чтобы сделать виртуальную машину компактнее, быстрее, а также повысить ее гибкость. Учитывая, что на работу над HotSpot потрачено около 1000 человеко-лет,

можно не сомневаться, что общую виртуальную машину ожидает блестящее будущее, так как она обогатится и многолетними наработками, выполненными для jRockit.

Наряду со слиянием виртуальных машин ведется работа над реализацией еще многих полезных функций. Одно из возможных перспективных дополнений — сущность из области параллелизма, называемая *сопрограммой* (coroutine).

14.4.2. Сопрограммы

Многопоточность — это такая форма параллельной обработки, которая кажется специалисту по Java и JVM наиболее естественной. Многопоточность базируется на комплексе служб JVM, занимающихся диспетчеризацией потоков, запускающих и останавливающих потоки на физических ядрах процессора. Потоки не имеют каких-либо возможностей для влияния на эту диспетчеризацию. Именно поэтому многопоточность также называется *вытесняющей (приоритетной) многозадачностью* — механизм диспетчеризации способен вытеснять работающие потоки и принудительно отбирать у них контроль над процессором.

Суть сопрограмм заключается в том, что исполняемые единицы получают частичный контроль над тем, как будет происходить их диспетчеризация. В частности, сопрограмма может работать как обычный поток до тех пор, пока не получит команду «уступить». В таком случае сопрограмма приостановится и позволит сработать другой сопрограмме. Когда и первой сопрограмме вновь предоставится возможность продолжить работу, она возобновит выполнение со следующего оператора после «уступки», а не с начала метода.

Поскольку такой подход к многопоточности основан на взаимодействии сопрограмм, активных в настоящий момент и способных по требованию предоставлять друг другу возможность выполниться, такая форма многопоточности называется *невытесняющей многозадачностью*.

Точная модель работы сопрограмм пока не сформулирована, нет даже уверенности, что они будут включены в Java 8. Один из возможных вариантов таков: сопрограммы могут создаваться и диспетчеризовываться в области видимости единого совместно используемого потока (или, возможно, пула потоков наподобие того, что применяется в `java.util.concurrent`). Такая модель продемонстрирована на рис. 14.3.

Потоки, выполняющие сопрограммы, можно будет вытеснять любым другим потоком системы, однако диспетчер потоков виртуальной машины Java не может принудить сопрограмму выполнить уступку. Это означает, что при условии доверия всем остальным сопрограммам в вашем пуле выполнения любая сопрограмма может управлять своими контекстными переключениями.

Возможность такого управления означает, что синхронизация между сопрограммами может происходить лучше. В многопоточном коде, чтобы обеспечить защиту данных, приходится выстраивать сложные и неустойчивые стратегии блокировок, так как переключение контекста может произойти в любой момент. Эту проблему, связанную с безопасностью типов при параллелизме, мы обсуждали

в разделе 4.1. Сопрограмма же должна лишь гарантировать, что ее данные будут согласованы в точках уступки, так как она может быть вытеснена только в такой точке.

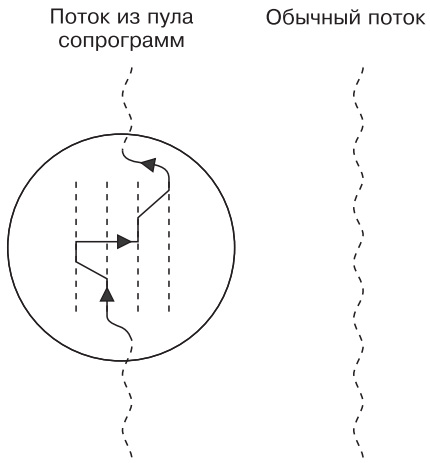


Рис. 14.3. Возможная модель работы сопрограмм

Подобный размен дополнительных гарантий на обычное доверие другим сопрограммам значительно упрощает многопоточные решения отдельных проблем, с которыми приходится сталкиваться при программировании. В некоторых других языках уже поддерживаются сопрограммы (по крайней мере, очень похожие элементы, называемые *волокнами* (fibers)) — как минимум это касается Ruby и новых версий JavaScript. Если сопрограммы будут добавлены на уровне виртуальной машины (но не обязательно на уровне языка Java), то это очень пригодится в других языках, способных их использовать.

В заключение поговорим еще об одном предполагаемом изменении виртуальной машины. Речь пойдет о сущностях, которые планируется называть кортежами. Они могут сыграть очень важную роль при решении таких задач, где важна высокая производительность.

14.4.3. Кортежи

В современной виртуальной машине Java все информационные элементы являются либо примитивными, либо ссылочными (ссылочные элементы могут быть объектами или массивами). Единственный способ создать более сложный тип — определить его в классе и передавать ссылки на объекты, являющиеся экземплярами нового типа. Это простая и довольно красивая модель, которая давно используется в языке Java.

Но оказывается, такая модель обладает и недостатками, которые проявляются при создании высокопроизводительных систем. В частности, в игровых программах

и финансовых приложениях такая простая модель часто не позволяет справиться со всеми возможными ситуациями. Для успешной работы и в этих сферах можно применить концепцию, которая называется «кортеж».

Кортежи (иногда также называемые *объектами значений*) — это языковые конструкции, занимающие промежуточное положение между примитивами и классами. Подобно классам, они допускают определение специальных сложных типов, в которых могут содержаться примитивы, ссылочные типы, а также другие кортежи. С примитивами их роднит то, что кортежи имеют целое значение, которое можно передавать методам и забирать из методов, а также сохранять их в массивах и других объектах. Кортежи можно сравнить со структурами (struct) из языка C (или .NET), если вы знакомы с этими экосистемами.

Рассмотрим пример — уже существующий API Java:

```
public class MyInputStream {
  public void write(byte[], int off, int len);
}
```

С помощью этого API пользователь может записать указанный объем данных в заданную точку массива, и это очень удобно. Но сама конструкция несовершенна. В идеальном объектно-ориентированном мире и необходимый отступ, и длина будут инкапсулированы в массиве и ни пользователь, ни создатель метода не должны будут отдельно отслеживать дополнительную информацию.

Фактически же с появлением неблокирующего ввода-вывода появился буфер `ByteBuffer`, в котором и инкапсулируется такая информация. И, к сожалению, работа с таким буфером не обходится без лишних издержек. Чтобы отрезать от него новый блок, требуется выделить новый объект, а это чревато дополнительной нагрузкой на подсистему сборки мусора. Большинство сборщиков мусора отлично справляются с подметанием короткоживущих объектов, но в среде, чувствительной к задержкам и требующей очень высокой производительности, подобное выделение объектов может аккумулироваться и приводить к недопустимым паузам в работе приложения.

А что бы произошло, если бы мы могли определить `Slice` как тип объекта значения (то есть тип кортежа), в котором содержалась бы ссылка на массив, отступ в массиве и длину информационного фрагмента. В листинге 14.2 для обозначения новой концепции используется ключевое слово `tuple`.

Листинг 14.2. Срез массива, представленный в виде кортежа

```
public tuple Slice {
  private int offset;
  private int length;
  private byte[] array;

  public byte get(int i) {
    return array[offset + i];
  }
}
```

Такая конструкция-срез обладает многими преимуществами как примитивных, так и ссылочных типов.

- Срезовые значения можно копировать из методов и вставлять в методы не менее эффективно, чем передавать вручную значения `int` или ссылки на массив.
- Срезовые значения попадают под очистку сразу после выхода из метода (так как они напоминают типы значений).
- Управление отступом от начала массива и длиной среза инкапсулируется в кортеже.

В современном программировании существуют самые разные типы, которые было бы удобно применять с кортежами. Например, это рациональные числа с числителем и знаменателем, сложные числа с реальным и мнимым значением либо корневые записи с ID пользователя и названием сервера, на котором этот пользователь находится (вам на заметку, уважаемые фанаты MMORPG!).

Еще одна область, при работе с которой кортежи помогают повысить производительность, — это обработка массивов. В настоящее время массивы содержат однородные коллекции данных либо примитивных, либо ссылочных типов. Благодаря кортежам мы можем полнее контролировать компоновку памяти при работе с массивами.

Рассмотрим пример — простую хеш-таблицу, в которой примитивный `long` используется в качестве ключа.

```
public class MyHashTable {
  private Entry[] entries;
}

public class Entry {
  private long key;
  private Object value;
}
```

В современной версии виртуальной машины Java массив `entries` должен содержать ссылки на экземпляры `Entry`. Каждый раз, когда вызывающая сторона ищет ключ в таблице, необходимо разыменовать соответствующую запись `Entry`, прежде чем ее ключ можно будет сравнить с переданным значением.

При реализации подобной конструкции с использованием кортежа мы могли бы расположить тип `Entry` внутристрочно прямо в массиве. Таким образом, мы избавимся от шага разыменования, требовавшегося для доступа к ключу, а также от связанных с ним издержек. На рис. 14.4 проиллюстрирован рассматриваемый случай, а также вариант его оптимизации, связанной с использованием кортежей.

Суть оптимизации производительности при использовании кортежей становится еще понятнее, если рассмотреть применение массивов кортежей. Как было рассказано в главе 6, производительность типичного кода приложений определяется количеством промахов в кэше L1. На рис. 14.4 показан код, который просматривает хеш-таблицу. Из рисунка понятно, что при использовании кортежей этот

код будет более эффективен. Он сможет считывать значения ключей без дополнительных операций выборки из кэша. В этом и заключается суть оптимизации производительности с помощью кортежей — программист может рациональнее располагать информацию в памяти.

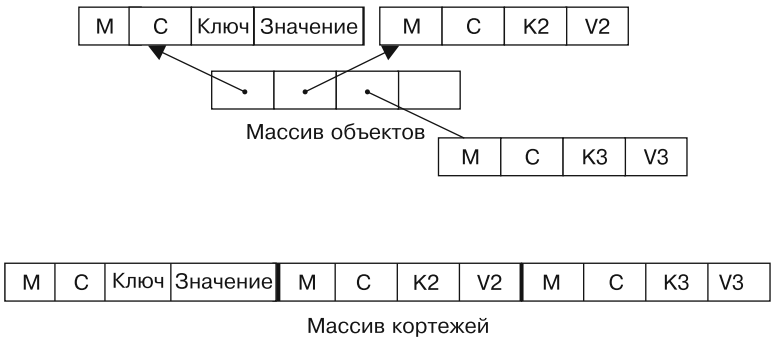


Рис. 14.4. Сравнение массивов виртуальной машины Java и кортежей

На этом мы завершаем обсуждение новых функций, которые могут войти в состав следующей версии Java и JDK 8. Вскоре сами увидим, какие из них окажутся в новом релизе. Если вас интересует эволюция этих возможностей, присоединяйтесь к проекту OpenJDK и Java Community Process и сами примите участие в разработке этих функций. Познакомьтесь с этими проектами и, если хотите, уточните, как к ним присоединиться.

14.5. Резюме

Итак, вот и закончилась книга о Java 7, а скоро выходит Java 8 — новая версия языка, полная различных усовершенствований, необходимых для написания качественного кода для современного аппаратного обеспечения. Приобретенные знания пригодятся вам при программировании для самых разных устройств — от встроенных контроллеров до гигантских мейнфреймов.

Миф о существовании единственного языка, на котором можно решить любые проблемы, возникающие при программировании, сегодня уже всерьез не воспринимается. Для создания эффективных решений — например, торговой системы, рассчитанной на выполнение большого количества параллельных задач, — необходимо изучать новые языки, способные эффективно взаимодействовать с основным кодом на Java.

Параллелизм будет оставаться актуальной темой, поскольку многоядерные процессоры и новые операционные системы предлагают все новые варианты сложных параллельных архитектур, для которых вы можете программировать. Чтобы успешно разрабатывать приложения, которые должны оперировать большими объемами данных, выполнять сложные вычисления либо достигать высокой скорости, вы должны исключительно хорошо разбираться в параллельной обработке.

Виртуальная машина Java заслуженно считается наилучшей из подобных платформ, по крайней мере на данный момент. Профессиональный Java-разработчик должен внимательно следить за развитием этой машины, так как она, возможно, будет развиваться в таких новых направлениях, как высокопроизводительные вычисления.

Как видите, жизнь кипит! Мы полагаем, что экосистема Java сейчас переживает коренную реорганизацию и в ближайшие несколько лет толковые Java-разработчики останутся крайне востребованными специалистами.

Приложения

A Установка исходного кода java7developer

Читая техническую книгу, любой разработчик предпочитает работать с живым кодом. В таком случае можно сразу получить впечатление о том, как функционирует описываемая в тексте программа, а на одном лишь коде это не всегда удается понять.

Исходный код для этой книги можно скачать по ссылке www.manning.com/evans/ или www.java7developer.com/. В тексте мы называем тот каталог, в котором сохранен исходный код, `$BOOK_CODE`.

Проект, в котором содержится весь исходный код для этой книги, называется java7developer. В нем представлен код на четырех языках — Java, Groovy, Scala и Clojure, а также библиотеки и ресурсы, необходимые для поддержки кода. Еще раз подчеркнем: это не обычный проект, написанный только на Java. Для того чтобы запустить эту сборку (скомпилировать код и выполнить все тесты), в точности следуйте нашим инструкциям. Для достижения различных целей сборочного цикла (например, `compile` и `test`) мы будем использовать инструмент Maven 3.

Рассмотрим, как наш исходный код организован в проекте java7developer.

A.1. Структура исходного кода java7developer

Структура проекта java7developer соответствует соглашениям Maven, рассмотренным в главе 12. Код организован следующим образом:

```
 java7developer
|-- lib
|-- pom.xml
|-- sample_posix_build.properties
|-- sample_windows_build.properties
`-- src
    |-- main
    |   '-- groovy
    |       `-- com
    |           `-- java7developer
    |               `-- chapter8
```

```
    |   `-- java
    |       `-- com
    |           `-- java7developer
    |               `-- chapter1
    |               `-- ...
    |               `-- ...
    |   `-- resources
    |   `-- scala
    |       `-- com
    |           `-- java7developer
    |               `-- chapter9
    `-- test
        `-- java
            `-- com
                `-- java7developer
                    `-- chapter1
                    `-- ...
                    `-- ...
        `-- scala
            `-- com
                `-- java7developer
                    `-- chapter9
`-- target
```

В рамках вышеупомянутых соглашений Maven отделяет основной код программы от кода тестов. Кроме того, он создает специальный каталог ресурсов, куда попадают все остальные файлы, которые требуется включить в состав сборки (например, `log4.xml` для логирования, конфигурационные файлы Hibernate и другие подобные ресурсы). Файл `pom.xml` — это сборочный сценарий для Maven. Мы подробно обсудим его в приложении Е.

Исходный код Scala и Groovy имеет такую же структуру, как исходный код из каталога java, но каталоги с исходным кодом для этих языков называются scala и groovy соответственно. В проекте Maven коды на языках Java, Scala и Groovy отлично уживаются бок о бок. А вот исходный код Clojure должен обрабатываться немного иначе. В большинстве случаев для такой обработки применяется интерактивная среда (а также используется сборочный инструмент Leiningen). Поэтому мы просто создали в проекте каталог для исходного кода на языке Clojure и назвали его clojure. Фрагменты из этого исходного кода можно скопировать в интерактивную среду Clojure REPL.

Целевой каталог не будет создан до тех пор, пока вы не запустите сборку Maven. Все классы, артефакты, отчеты и другие файлы, создаваемые в ходе сборки, будут лежать в этом каталоге.

В каталоге lib находятся некоторые библиотеки — на случай, если ваша сборка Maven не сможет обратиться к Интернету и скачать там все необходимые файлы.

Изучите структуру проекта, запомните, где лежит исходный код для разных глав. Как только справитесь с этой задачей, перейдем к установке и конфигурированию Maven 3.

A.2. Скачивание и установка Maven

Можете скачать Maven по адресу http://maven.apache.org/download.html. В примерах из главы 12 мы пользовались версией Maven 3.0.3. Скачайте файл `apache-maven-3.0.3-bin.tar.gz`, если работаете с операционной системой *nix, или `apache-maven-3.0.3-bin.zip`, если у вас система Windows. Как только скачаете файл, просто разархивируйте его в каталог на ваш выбор.

ВНИМАНИЕ

Как и при установке многих других программ, предназначенных для работы с Java/JVM, лучше ставить Maven в такой каталог, в имени которого отсутствуют пробелы, так как из-за них могут возникать ошибки в PATH и CLASSPATH. Например, если вы работаете с операционной системой Microsoft Windows, то установите Maven в каталог вида C:\ Program Files\Maven\.

После того как скачаете и разархивируете программу, нужно настроить переменную окружения `M2_HOME`. В операционных системах *nix потребуется добавить строку такого вида:

```
M2_HOME=/opt/apache-maven-3.0.3
```

А в операционных системах Windows — такую строку:

```
M2_HOME=C:\apache-maven-3.0.3
```

Вы можете поинтересоваться: «А почему M2_HOME, а не M3_HOME? Это же Maven 3». Дело в том, что команда разработчиков Maven просто очень беспокоится об обратной совместимости с широко распространенной версией Maven 2.

Для работы Maven требуется Java JDK. Подойдет любая версия выше 1.5 (правда, если вы дочитали книгу до этого раздела, то у вас определенно установлен JDK 1.7). Кроме того, убедитесь, что у вас правильно настроена переменная окружения `JAVA_HOME` — ее нужно было настроить уже на этапе установки Java. Далее необходимо гарантировать, что вы сможете в любой момент работы запускать команды Maven через командную строку. Поэтому задайте каталог `M2_HOME/bin` в качестве вашего пути `PATH`. В операционных системах *nix нужно добавить и следующую строку:

```
PATH=$PATH:$M2_HOME/bin
```

А в системах Windows — такую строку:

```
PATH=%PATH%;%M2_HOME%\bin
```

Теперь вы можете выполнить команду Maven (`mvn`) с параметром `-version`, чтобы убедиться, что базовая установка работает:

```
mvn -version
```

Если Maven работает правильно, то на экране появится вывод следующего содержания:

```
Apache Maven 3.0.3 (r1075438; 2011-02-28 17:31:09+0000)
Maven home: C:\apache-maven-3.0.3
Java version: 1.7.0, vendor: Oracle Corporation
```

```
Java home: C:\Java\jdk1.7.0\jre
Default locale: en_GB, platform encoding: Cp1252
OS name: "windows xp", version: "5.1", arch: "x86", family: "windows"
```

Как видите, Maven предлагает нам полезную конфигурационную информацию, и вы сразу можете удостовериться, что все зависимости между ним и вашей платформой в порядке.

СОВЕТ

Maven поддерживается тремя основными интегрированными средами разработки (Eclipse, IntelliJ и NetBeans). Если вам не слишком удобно работать с Maven в командной строке, можете воспользоваться версией этого инструмента, интегрированной с IDE.

Итак, система Maven установлена. Посмотрим, где находятся пользовательские настройки. Чтобы инициировать создание каталога с пользовательскими настройками, убедитесь, что у вас скачан и установлен плагин Maven. Для этого проще всего выполнить плагин Help:

```
mvn help:system
```

Эта команда скачивает, устанавливает и запускает справочный плагин Help, который сообщает вам информацию дополнительно к той, которая содержится в `mvn -version`. Кроме того, так вы гарантируете, что в системе будет создан каталог .m2. Очень важно знать, где именно находятся пользовательские настройки, поскольку в некоторых случаях их требуется изменить. Например, эта информация может быть важна в том случае, когда Maven должен работать через прокси-сервер. В вашем домашнем каталоге, который мы называем `$HOME`, будут находиться каталоги и файлы, перечисленные в табл. А.1.

Таблица А.1. Пользовательские каталоги и файлы Maven[1]

Название	Объяснение
$HOME/.m2	Скрытый каталог, в котором содержится пользовательская конфигурация для Maven
$HOME/.m2/settings.xml	Файл, содержащий конфигурационную информацию о конкретном пользователе. Здесь можно указать пути для обхода прокси, добавить приватные репозитории и включить другую информацию, позволяющую настраивать функционал вашего экземпляра Maven
$HOME/.m2/repository/	Локальный репозиторий Maven на вашем компьютере. Когда Maven скачивает плагин или зависимость из хранилища Maven Central (либо из другого удаленного репозитория Maven), он сохраняет копию такой зависимости в вашем локальном репозитории. Аналогичный процесс происходит при установке зависимостей, когда выполняется стадия install. Далее Maven может работать уже с локальной копией, а не скачивать каждый раз новый экземпляр

[1] Информация любезно предоставлена сайтом Sonatype, взята из онлайнового справочника «Maven: Полное руководство», расположенного по адресу www.sonatype.com/Request/Book/Maven-The-Complete-Reference

Опять же обратите внимание на то, как обеспечивается обратная совместимость с Maven 2: каталог называется .m2, а не .m3, как можно было бы ожидать.

Итак, на данном этапе Maven у вас уже установлен и вы знаете, где находятся конфигурационные файлы с пользовательской информацией. Можно приступать к работе со сборкой java7developer.

A.3. Запуск сборки java7developer

В этом разделе мы рассмотрим пару разовых операций, которые необходимо выполнить при подготовке сборки[1]. В частности, мы вручную установим одну библиотеку, а также переименуем файл свойств и отредактируем его так, чтобы он указывал на локальный экземпляр Java 7.

Затем мы проработаем наиболее распространенные стадии сборочного цикла Maven (`clean`, `compile` и `test`). Первая стадия в рамках сборочного цикла (`clean`) применяется для уборки всех ненужных артефактов, которые могли остаться в системе после окончания предыдущей сборки.

Сборочные сценарии Maven называются POM-файлами (POM означает модель объекта проекта). Такие POM-файлы написаны на языке XML. Каждый проект или модуль Maven имеет сопутствующий файл `pom.xml`. В будущем должна быть реализована языковая поддержка, которая послужит альтернативой POM-файлам. Таким образом, вы приобретете дополнительную гибкость в работе, если она вам потребуется (примерно такая альтернатива существует в сборочном инструменте Gradle).

Чтобы выполнять сборки в Maven, вы приказываете программе запустить одну или несколько стадий, каждая из которых представляет конкретную задачу (компиляция кода, запуск тестов и др.). Все стадии привязаны к выполняемому по умолчанию сборочному циклу. Поэтому если вы приказываете Maven провести какие-либо тесты (например, `mvn test`), то он скомпилирует как основной исходный код, так и исходный код тестов, и лишь потом попытается запустить их. Одним словом, Maven стимулирует вас придерживаться правильного сборочного цикла.

Итак, разберемся с первой «одноразовой» задачей.

A.3.1. Однократная подготовка сборки

Чтобы успешно запустить сборку, нужно первым делом переименовать и отредактировать файл свойств. Если вы еще не сделали этого в разделе 12.2, то перейдите в каталог `$BOOK_CODE`, скопируйте файл `sample_<os>_build.properties`, предлагаемый для образца (выберите файл для вашей операционной системы) в `build.properties`, а потом измените значение свойства `jdk.javac.fullpath`, чтобы оно указывало на ваш локальный экземпляр установки Java 7. Так мы гарантируем, что система возьмет нужный JDK, и Maven будет пользоваться им при сборке кода Java.

---

[1] Несмотря на последние оптимизации сборочного инструмента Maven и действующей в нем поддержки многоязычного программирования, в Maven еще есть кое-какие недоработки.

Когда закончите эту работу, можно будет запустить стадию `clean`, которую нужно выполнять в рамках любой сборки.

А.3.2. Очистка

Стадия Maven `clean` просто предполагает удаление целевого каталога. Чтобы посмотреть, как это происходит на практике, перейдите в каталог `$BOOK_CODE` и выполните стадию Maven `clean`:

```
cd $BOOK_CODE
mvn clean
```

После этого окно консоли начнет заполняться выводом Maven, сообщающим, что система занята скачиванием различных сторонних плагинов и библиотек. Они нужны Maven для запуска стадий, и по умолчанию эти данные скачиваются из Maven Central — основного онлайнового репозитория с такими артефактами. Проект java7developer конфигурируется и с другим репозиторием — для этого загружается файл `asm-4.0.jar`.

ПРИМЕЧАНИЕ

Иногда Maven осуществляет такие операции и при выполнении других стадий. Поэтому не волнуйтесь, если программа вдруг попытается «выкачать Интернет», когда об очистке речь не идет. Все подобные загрузки Maven выполняет лишь по одному разу.

Кроме информации о загрузке (`Downloading…`), в вашей консоли должен отобразиться текст примерно следующего содержания:

```
[INFO] ------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------
[INFO] Total time: 1.703s
[INFO] Finished at: Fri Jun 24 13:51:58 BST 2011
[INFO] Final Memory: 6M/16M
[INFO] ------------------------------------------------------------------
```

Если стадию `clean` выполнить не удается, то причина, скорее всего, в прокси-сервере, который блокирует доступ к Maven Central. Как вы помните, в этом репозитории содержатся различные плагины и сторонние библиотеки. Чтобы решить эту проблему, просто отредактируйте файл `$HOME/.m2/settings.xml` и добавьте в него следующую информацию, указав правильные значения для различных элементов:

```
<proxies>
  <proxy>
    <active>true</active>
    <protocol></protocol>
    <username></username>
    <password></password>
    <host></host>
    <port></port>
  </proxy>
</proxies>
```

Перезапустите стадию — после этого вы должны увидеть сообщение BUILD SUCCESS.

СОВЕТ

В отличие от других стадий сборочного жизненного цикла Maven, которые вам доведется выполнять, clean не вызывается автоматически. Если вы хотите убрать артефакты, которые могли остаться после предыдущей сборки, всегда указывайте стадию clean.

Теперь, когда удалены все остатки, «застрявшие» в системе после прошлой сборки, обычно требуется скомпилировать код. За это отвечает следующая стадия сборочного цикла — compile.

A.3.3. Компиляция

Стадия Maven compile использует конфигурацию компиляционного плагина, записанную в файле pom.xml. На основании этих данных система компилирует исходный код из каталогов src/main/java, src/main/scala и src/main/groovy. Фактически это означает, что мы запускаем компиляторы Java, Scala и Groovy (javac, scalac и groovyc) с зависимостями, ограниченными областью компиляции. Эти зависимости добавляются в CLASSPATH. Кроме того, Maven обрабатывает ресурсы, находящиеся в src/main/resources, гарантируя, что при компиляции они также будут учтены в CLASSPATH.

Скомпилированные в результате классы оказываются в каталоге target/classes. Чтобы просмотреть этот этап на практике, выполните следующую стадию Maven:

```
mvn compile
```

Компиляция должна выполниться достаточно быстро, и в вашей консоли отобразится примерно такой вывод:

```
...
[INFO] [compiler:compile {execution: default-compile}]
[INFO] Compiling 119 source files to
    C:\Projects\workspace3.6\code\trunk\target\classes
[INFO] [scala:compile {execution: default}]
[INFO] Checking for multiple versions of scala
[INFO] includes = [**/*.scala,**/*.java,]
[INFO] excludes = []
[INFO] C:\Projects\workspace3.6\code\trunk\src\main\java:-1: info: compiling
[INFO] C:\Projects\workspace3.6\code\trunk\target\generated-sources\groovy-
    stubs\main:-1: info: compiling
[INFO] C:\Projects\workspace3.6\code\trunk\src\main\groovy:-1: info:
    compiling
[INFO] C:\Projects\workspace3.6\code\trunk\src\main\scala:-1: info:
    compiling
[INFO] Compiling 143 source files to
    C:\Projects\workspace3.6\code\trunk\target\classes at 1312716331031
[INFO] prepare-compile in 0 s
[INFO] compile in 12 s
[INFO] [groovy:compile {execution: default}]
[INFO] Compiled 26 Groovy classes
```

```
[INFO]------------------------------------------------------------------
[INFO] BUILD SUCCESSFUL
[INFO] ------------------------------------------------------------------
[INFO] Total time: 43 seconds
[INFO] Finished at: Sun Aug 07 12:25:44 BST 2011
[INFO] Final Memory: 33M/79M
[INFO] ------------------------------------------------------------------
```

На данном этапе тестовые классы, находящиеся в каталогах `src/test/java`, `src/test/scala` и `src/test/groovy`, еще не скомпилированы. Чтобы они были готовы к этому моменту, можно было бы воспользоваться специальной стадией `test-compile`, но обычно в Maven принято отдельно выполнять стадию `test`.

A.3.4. Тестирование

Именно стадия `test` позволяет оценить сборочный жизненный цикл Maven в действии. Если вы прикажете Maven запустить тесты, то система будет знать, что предварительно необходимо выполнить все предыдущие стадии сборочного жизненного цикла (это касается стадий `compile`, `test-compile` и многих других) — лишь в таком случае `test` может выполниться успешно.

Maven запускает тесты с помощью плагина Surefire, пользуясь поставщиком тестов (в данном случае JUnit), который вы указываете в файле `pom.xml` в качестве одной из зависимостей, ограниченных областью видимости `test`. Maven не только запускает тест, но и готовит файлы с отчетами, которые позже можно проанализировать для исследования заведомо неуспешных тестов и сбора тестовых показателей.

Чтобы увидеть этот этап на практике, выполните следующие стадии Maven:

```
mvn clean test
```

Как только Maven закончит компилировать и запускать тесты, вы должны получить на экране примерно такой вывод:

```
...
Running com.java7developer.chapter11.listing_11_3.TicketRevenueTest
Tests run: 5, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0 sec
Running com.java7developer.chapter11.listing_11_4.TicketRevenueTest
Tests run: 5, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0 sec
Running com.java7developer.chapter11.listing_11_5.TicketTest
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.015 sec

Results :

Tests run: 20, Failures: 0, Errors: 0, Skipped: 0
[INFO]------------------------------------------------------------------
[INFO] BUILD SUCCESSFUL
[INFO]------------------------------------------------------------------
[INFO] Total time: 16 seconds
[INFO] Finished at: Wed Jul 06 13:50:07 BST 2011
[INFO] Final Memory: 24M/58M
[INFO]------------------------------------------------------------------
```

Результаты тестов сохраняются в каталоге `target/surefire-reports`. Можете уже сейчас изучить текстовые файлы и убедиться, что тесты пройдены успешно.

A.4. Резюме

Запуская примеры исходного кода по ходу изучения отдельных глав, вы сможете полнее усвоить материал, представленный в этой книге. Если вам хочется поимпровизировать, можете изменить этот код или даже добавить свой, а потом скомпилировать и запустить ваши примеры так, как это описано выше.

Практика показывает, что сборочные инструменты, подобные Maven 3, удивительно сложны на внутрисистемном уровне. Если вы хотите подробнее разобраться с этой темой, прочтите главу 12 — в ней рассматриваются вопросы сборки и непрерывной интеграции.

B Синтаксис и примеры паттернов подстановки

Паттерны подстановки (glob-паттерны) применяются в библиотеках NIO.2 в Java 7 в качестве фильтров в процессе перебора информации в каталогах и при решении других подобных задач, о которых мы говорили в главе 2.

B.1. Синтаксис паттернов подстановки

Паттерны подстановки проще регулярных выражений и строятся в соответствии с основными правилами, приведенными в табл. B.1.

Таблица B.1. Синтаксис паттернов подстановки

Синтаксис	Описание
*	Совпадение с 0 или более символов
**	Совпадение с 0 или более символов во всех каталогах
?	Точное совпадение с одним символом
{}	Отграничивает коллекцию субпаттернов для совпадения с неявным OR для каждого паттерна; например, совпадает с паттерном A, или B, или C и т. д.
[]	Совпадает с единым набором символов, либо, если применяется дефис (–), совпадает с диапазоном символов
\	Экранирующий символ; применяется, если требуется обеспечить совпадение со специальными символами, например *, ? или \

Более подробная информация о синтаксисе паттернов подстановки приводится в руководстве по Java, представленном на сайте Oracle (http://docs.oracle.com/javase/tutorial/essential/io/fileOps.html#glob), а также в документе Javadoc по классу `FileSystem`.

B.2. Примеры паттернов подстановки

Простейшие примеры использования паттернов подстановки (так называемого «глоббинга») приведены в табл. B.2.

Таблица В.2. Примеры паттернов подстановки

Синтаксис	Описание
*.java	Совпадает со всеми строками, оканчивающимися на .java, например Listing_2_1.java
??	Совпадает с любыми двумя символами, например ab или x1
[0–9]	Совпадает с любой цифрой от 0 до 9
{groovy, scala}.*	Совпадает с любыми строками, начинающимися с groovy или scala, например со scala.txt или groovy.pdf
[]a-z, A-Z	Совпадает с символами латиницы в верхнем или нижнем регистре
\\	Совпадает с символом \
/usr/home/**	Совпадает со всеми строками, начинающимися с /usr/home/, например с usr/home/karianna или /usr/home/karianna/docs

Более подробная информация о совпадении с паттернами подстановки приводится в руководстве по Java, предоставленном на сайте Oracle (http://docs.oracle.com/javase/tutorial/essential/io/fileOps.html#glob), а также в документе Javadoc по классу `FileSystem`.

ВНИМАНИЕ

В спецификации Java 7 определяется собственная семантика подстановочных конструкций (вместо общепринятой). В результате появляется возможность совершать некоторые трюки, особенно в UNIX. Так, если выполнить в Java 7 эквивалент rm *, то команда удалит все файлы, имена которых начинаются с точки. А команда UNIX rm/glob этого не сделает.

C Установка альтернативных языков для виртуальной машины Java

В этом приложении вы узнаете, как скачать и установить три языка для виртуальной машины Java (Groovy, Scala и Clojure). Мы также опишем, как устанавливать фреймворк Grails, основанный на Groovy. Языки Groovy, Scala, Clojure и Grails рассмотрены в главах 8, 9, 10 и 13 соответственно.

C.1. Groovy

Установить Groovy совсем не сложно, но если вы не знаете, как задавать переменные окружения или пока плохо разбираетесь в той или иной операционной системе, то это руководство вам пригодится.

C.1.1. Скачивание Groovy

Для начала перейдите по ссылке http://groovy.codehaus.org/Download и скачайте последнюю стабильную версию Groovy. В примерах из нашей книги мы пользовались Groovy 1.8.6, поэтому рекомендуем скачать бинарный файл `groovy-binary-1.8.6.zip`. Далее распакуйте содержимое этого ZIP-файла в каталог на ваш выбор.

ВНИМАНИЕ

Как и при установке многих других программ для Java/JVM, не рекомендуется устанавливать Groovy в такой каталог, в названии которого присутствуют пробелы, так как это может вызывать ошибки в PATH и CLASSPATH. Например, если вы работаете в Microsoft Windows, то не устанавливайте Groovy в каталог вида C:\Program Files\Groovy\.

Что ж, осталось сделать не так много. На следующем этапе нужно настроить переменные окружения.

C.1.2. Установка Groovy

Скачав и распаковав файл, вы должны установить три переменные окружения, чтобы эффективно работать с Groovy. Мы рассмотрим эту проблему и для операционных систем на базе POSIX (Linux, UNIX, Mac OS X), и для Microsoft Windows.

Операционные системы на базе POSIX (Linux, UNIX, Mac OS X)

Установка переменных окружения в операционных системах, работающих на базе POSIX, обычно зависит от того, в какой пользовательской оболочке вы находитесь, когда открываете окно терминала. В табл. C.1 перечислены распространенные имена и варианты расположения конфигурационного файла вашей пользовательской оболочки в различных операционных системах на основе POSIX.

Таблица C.1. Типичные местоположения конфигурационных файлов пользовательской оболочки

Оболочка	Местоположение файла
Bash	~/.bashrc и/или ~/.profile
Korn (ksh)	~/.kshrc и/или ~/.profile
Sh	~/.profile
Mac OS X	~/.bashrc и/или ~./.profile и/или ~./bash_profile

Далее откройте конфигурационный файл пользовательской оболочки в вашем любимом текстовом редакторе и добавьте три переменные окружения: `GROOVY_HOME`, `JAVA_HOME` и `PATH`.

Сначала нужно задать переменную окружения `GROOVY_HOME`. Добавьте приведенную ниже строку и замените `<installation directory>` тем каталогом, в который вы распаковали содержимое бинарного файла Groovy.

```
GROOVY_HOME=<installation directory>
```

В следующем примере мы распаковали бинарный файл Groovy в `/opt/groovy-1.8.6`:

```
GROOVY_HOME=/opt/groovy-1.8.6
```

Для работы Groovy требуется Java JDK. Подойдет любая версия выше 1.5 (правда, на данном этапе у вас уже наверянка настроена версия JDK 1.7). Кроме того, убедитесь, что у вас установлена переменная окружения `JAVA_HOME`. Скорее всего, вы задали ее уже на этапе установки Java, но если нет — добавьте следующую строку:

```
JAVA_HOME=<путь к каталогу, где установлен язык Java>
```

В следующем примере мы задали для `JAVA_HOME` значение `/opt/java/java-1.7.0`:

```
JAVA_HOME=/opt/java/java-1.7.0
```

Наконец, требуется возможность выполнять команды Groovy из любой точки рабочей среды через командную строку. Для этого нужно указать в пути PATH каталог GROOVY_HOME/bin:

```
PATH=$PATH:$GROOVY_HOME/bin
```

Сохраните конфигурационный файл вашей пользовательской оболочки. Когда вы в следующий раз запустите новую оболочку, три переменные окружения уже будут заданы. Теперь можете выполнить команду groovy с параметром -version в командной строке, чтобы убедиться, что данная простейшая установка работает:

```
groovy -version
Groovy Version: 1.8.6 JVM: 1.7.0
```

На этом мы завершаем раздел об установке Groovy. Можете также повторить главу 8, где рассказано, как компилировать и запускать код Groovy.

MS Windows

В Microsoft Windows переменные окружения удобнее всего устанавливать через специальный графический пользовательский интерфейс, предоставляемый для управления компьютером. Выполните следующие шаги.

1. Щелкните правой кнопкой мыши на значке My Computer (Мой компьютер) и в контекстном меню выберите пункт Properties (Свойства).
2. Перейдите на вкладку Advanced (Дополнительно).
3. Нажмите кнопку Environment Variables (Переменные окружения).
4. Теперь нажмите кнопку New (Создать), чтобы добавить имя и значение для новой переменной.

Далее нужно задать переменную окружения GROOVY_HOME. Добавьте приведенную ниже строку и замените <installation directory> тем каталогом, в который вы распаковали содержимое бинарного файла Groovy.

```
GROOVY_HOME=<installation directory>
```

В следующем примере мы распаковали бинарный файл Groovy в C:\languages\groovy-1.8.6:

```
GROOVY_HOME=C:\languages\groovy-1.8.6
```

Для работы Groovy требуется Java JDK. Подойдет любая версия выше 1.5 (правда, на данном этапе у вас уже наверняка настроена версия JDK 1.7). Кроме того, убедитесь, что у вас установлена переменная окружения JAVA_HOME. Скорее всего, вы задали ее уже на этапе установки Java, но если нет — добавьте такую строку:

```
JAVA_HOME=<путь к каталогу, где установлен язык Java>
```

В следующем примере мы задали для JAVA_HOME значение C:\Java\jdk-1.7.0:

```
JAVA_HOME= C:\Java\jdk-1.7.0
```

Наконец, требуется возможность выполнять команды Groovy из любой точки рабочей среды через командную строку. Для этого нужно указать в пути PATH каталог GROOVY_HOME/bin:

```
PATH=$PATH:$GROOVY_HOME/bin
```

Нажимайте кнопку OK, пока не выйдете из раздела управления компьютером. Когда вы в следующий раз запустите новую оболочку, три переменные окружения уже будут заданы. Теперь можете выполнить команду groovy с параметром -version в командной строке, чтобы убедиться, что данная простейшая установка работает:

```
groovy -version
Groovy Version: 1.8.6 JVM: 1.7.0
```

На этом мы завершаем раздел об установке Groovy для Microsoft Windows. Можете также повторить главу 8, где рассказано, как компилировать и запускать код Groovy.

C.2. Scala

Рабочую среду для Scala можно скачать по адресу www.scala-lang.org/downloads. На момент написания книги наиболее новой была версия 2.9.1, но вы наверняка найдете и более новые версии. В каждой новой версии Scala обычно вводятся языковые изменения. Поэтому если окажется, что в новых версиях Scala какие-то примеры не работают — проверьте версию языка и попробуйте запускать наши примеры в версии 2.9.1.

Пользователям Windows следует скачать ZIP-архив. Пользователи UNIX-подобных операционных систем (в частности, систем Mac и Linux) должны скачивать версию в формате TGZ. Распакуйте архив и сохраните его в вашей файловой системе. Как и в случае с Groovy, не распаковывайте этот файл в каталог, в имени которого присутствуют пробелы.

Существует несколько способов настроить рабочую среду Scala на вашем компьютере. Проще всего задать переменную окружения под названием SCALA_HOME, которая указывает на тот каталог, в котором вы установили Scala. После этого следуйте инструкциям, приведенным в разделе C.1.2 для Groovy и относящимся к вашей операционной системе, но везде замените GROOVY_HOME на SCALA_HOME.

Когда закончите конфигурирование среды, можете ввести в командной строке scala — должна открыться интерактивная сессия Scala. Если этого не происходит, то, вероятно, ваша рабочая среда сконфигурирована неправильно. Проследите в обратном направлении все свои шаги и убедитесь, что значения SCALA_HOME и PATH заданы верно.

Теперь вы должны без проблем запускать листинги Scala и интерактивные фрагменты кода из главы 9.

C.3. Clojure

Чтобы скачать Clojure, перейдите по адресу http://clojure.org/ и скачайте ZIP-файл с наиболее актуальной стабильной версией. В наших примерах мы использовали Clojure 1.2. Если вы будете работать с более новой версией, то возможны небольшие отличия в работе.

Распакуйте скачанный файл и перейдите в созданный каталог. При условии, что в вашем пути PATH уже заданы значения JAVA_HOME и java, у вас должна без проблем запуститься базовая интерактивная среда REPL, описанная в главе 10, вот так:

```
java -cp clojure.jar clojure.main
```

Clojure немного отличается от двух других языков, рассматриваемых в этом приложении. Чтобы работать с Clojure, вам, в сущности, нужен лишь файл clojure.jar. Устанавливать переменную окружения, как это делалось в Scala и Groovy, не придется.

Пока вы только изучаете Clojure, с этим языком лучше работать через интерактивную среду REPL. Если вам придется применять Clojure в реальных боевых проектах, то понадобится воспользоваться специальным сборочным инструментом, например Leiningen (он подробно рассмотрен в главе 12). Такой инструмент управляет не только развертыванием приложений, но и установкой самого языка Clojure (путем загрузки JAR-файла из удаленного репозитория Maven).

У базовой установки Clojure есть несколько ограничений, но, к счастью, Clojure очень хорошо интегрируется с большинством распространенных IDE. Например, если вы работаете с Eclipse, то мы настоятельно рекомендуем плагин Counterclockwise для этой среды — он очень удобен и прост в установке.

При разработке кода целесообразно иметь более насыщенную рабочую среду, чем обычная REPL, так как писать в REPL большие программы может быть довольно утомительно. Но во многих случаях (и особенно на этапе обучения) REPL будет вполне достаточно.

C.4. Grails

Установить Grails совсем не сложно, но если вы не знаете, как задавать переменные окружения или пока плохо разбираетесь в той или иной операционной системе, то это руководство вам пригодится. Подробные инструкции по установке Grails приводятся по адресу www.grails.org/installation.

C.4.1. Скачивание Grails

Для начала перейдите по ссылке www.grails.org и скачайте последнюю стабильную версию Grails. В примерах из нашей книги мы пользовались версией 2.0.1. Далее распакуйте содержимое этого ZIP-файла в каталог на ваш выбор.

ВНИМАНИЕ

Как и при установке многих других программ для Java/JVM, не рекомендуется устанавливать Groovy в такой каталог, в названии которого присутствуют пробелы, так как это может вызывать ошибки в PATH и CLASSPATH. Например, если вы работаете в Microsoft Windows, то не устанавливайте Groovy в каталог вида C:\Program Files\Grails\

На следующем этапе нужно настроить переменные окружения.

C.4.2. Установка Grails

Скачав и распаковав файл, вы должны установить три переменные окружения, чтобы эффективно работать с Grails. Мы рассмотрим эту проблему и для операционных систем на базе POSIX (Linux, UNIX, Mac OS X), и для Microsoft Windows.

Операционные системы на базе POSIX (Linux, UNIX, Mac OS X)

Установка переменных окружения в операционных системах, работающих на базе POSIX, обычно зависит от того, в какой пользовательской оболочке вы находитесь, когда открываете окно терминала. В табл. C.2 перечислены распространенные имена и варианты расположения конфигурационного файла вашей пользовательской оболочки в различных операционных системах на основе POSIX.

Таблица C.2. Типичные местоположения конфигурационных файлов пользовательской оболочки

Оболочка	Местоположение файла
Bash	~/.bashrc и/или ~/.profile
Korn (ksh)	~/.kshrc и/или ~/.profile
Sh	~/.profile
Mac OS X	~/.bashrc и/или ~./.profile и/или ~./bash_profile

Далее откройте конфигурационный файл пользовательской оболочки в вашем любимом текстовом редакторе и добавьте три переменные окружения: `GRAILS_HOME`, `JAVA_HOME` и `PATH`.

Сначала нужно задать переменную окружения `GRAILS_HOME`. Добавьте приведенную ниже строку и замените `<installation directory>` тем каталогом, в который вы распаковали содержимое бинарного файла Grails.

```
GRAILS_HOME=<installation directory>
```

В следующем примере мы распаковали бинарный файл Groovy в `/opt/grails-2.0.1`:

```
GRAILS_HOME=/opt/grails-2.0.1
```

Для работы Grails требуется Java JDK. Подойдет любая версия выше 1.5. Кроме того, убедитесь, что у вас установлена переменная окружения `JAVA_HOME`. Скорее

всего, вы задали ее уже на этапе установки Java, но если нет — добавьте следующую строку:

```
JAVA_HOME=<путь к каталогу, где установлен язык Java>
```

В следующем примере мы задали для JAVA_HOME значение /opt/java/java-1.7.0:

```
JAVA_HOME=/opt/java/java-1.7.0
```

Наконец, требуется возможность выполнять команды Grails из любой точки рабочей среды через командную строку. Для этого нужно указать в пути PATH каталог GRAILS_HOME/bin:

```
PATH=$PATH:$GRAILS_HOME/bin
```

Сохраните конфигурационный файл вашей пользовательской оболочки. Когда вы в следующий раз запустите новую оболочку, три переменные окружения уже будут заданы. Теперь можете выполнить команду grails с параметром -version в командной строке, чтобы убедиться, что данная простейшая установка работает:

```
grails -version
Grails version: 2.0.1
```

На этом мы завершаем раздел об установке Grails. Можете также повторить главу 13, где рассказано, как выполнить ваш первый проект с Grails!

MS Windows

В Microsoft Windows переменные окружения удобнее всего устанавливать через специальный графический пользовательский интерфейс, предоставляемый для управления компьютером. Выполните следующие шаги.

1. Щелкните правой кнопкой мыши на значке My Computer (Мой компьютер), далее выберите пункт Properties (Свойства).
2. Перейдите на вкладку Advanced (Дополнительно).
3. Нажмите кнопку Environment Variables (Переменные окружения).
4. Теперь нажмите кнопку New (Создать), чтобы добавить имя и значение для новой переменной.

Далее нужно задать переменную окружения GRAILS_HOME. Добавьте приведенную ниже строку и замените <installation directory> тем каталогом, в который вы распаковали содержимое бинарного файла Grails.

```
GRAILS_HOME=<installation directory>
```

В следующем примере мы распаковали бинарный файл Grails в C:\languages\grails-2.0.1:

```
GRAILS_HOME=C:\languages\grails-2.0.1
```

Для работы Grails требуется Java JDK. Подойдет любая версия выше 1.5. Кроме того, убедитесь, что у вас установлена переменная окружения JAVA_HOME. Скорее

всего, вы задали ее уже на этапе установки Java, но если нет — добавьте следующую строку:

```
JAVA_HOME=<путь к каталогу, где установлен язык Java>
```

В следующем примере мы задали для JAVA_HOME значение C:\Java\jdk-1.7.0:

```
JAVA_HOME=C:\Java\jdk-1.7.0
```

Наконец, требуется возможность выполнять команды Grails из любой точки рабочей среды через командную строку. Для этого нужно указать в пути PATH каталог GRAILS_HOME/bin:

```
PATH=$PATH:$GRAILS_HOME/bin
```

Нажимайте кнопку OK, пока не выйдете из раздела управления компьютером. Когда вы в следующий раз запустите новую оболочку, три переменные окружения уже будут заданы. Теперь можете выполнить команду grails с параметром -version в командной строке, чтобы убедиться, что данная простейшая установка работает:

```
grails -version
Grails version: 2.0.1
```

На этом мы завершаем раздел об установке Grails для Microsoft Windows. Можете также повторить главу 13, где рассказано, как выполнить ваш первый проект с Grails!

D Скачивание и установка Jenkins

D.1. Загрузка Jenkins

Сервер Jenkins можно скачать по адресу http://mirrors.jenkins-ci.org/. В примерах из главы 12 мы использовали версию Jenkins 1.424.

Обычный способ установки Jenkins, независимый от конкретной операционной системы, предполагает использование пакета `jenkins.war`. Но если вы пока не умеете запускать собственный веб-сервер, например Tomcat или Jetty, то можете скачать пакет, предназначенный именно для вашей операционной системы.

СОВЕТ

Команда Jenkins выпускает новые релизы поразительно часто, это может раздражать разработчиков, привыкших иметь дело с более стабильными серверами непрерывной интеграции. Чтобы справиться с этой проблемой, создатели Jenkins предлагают специальную версию, рассчитанную на длительную поддержку (LTS).

Далее необходимо выполнить несколько простых шагов по установке программы.

D.2. Установка Jenkins

После того как вы скачаете либо WAR-файл, либо пакет, предназначенный именно для вашей операционной системы, нужно установить Jenkins. Для работы Jenkins требуется Java JDK. Подойдет любая версия выше 1.5 (правда, на данном этапе у вас уже наверянка настроена версия JDK 1.7). Сначала мы обсудим установку WAR-файла, а потом установку пакета для конкретной операционной системы.

D.2.1. Запуск WAR-файла

Чтобы максимально быстро установить Jenkins, нужно выполнить WAR-файл Jenkins прямо в командной строке:

```
java -jar jenkins.war
```

Этот способ целесообразен, только если вы устанавливаете пробную версию Jenkins для ознакомления. При установке Jenkins таким образом будет гораздо сложнее сконфигурировать другие связанные с веб-сервером переменные, что может понадобиться для гладкой работы системы.

D.2.2. Установка WAR-файла

Если вас интересует долговременная установка Jenkins, разверните WAR-файл на своем любимом веб-сервере для Java. Для проекта java7developer мы просто скопировали файл `jenkins.war` в каталог webapps на работающем сервере Apache Tomcat 7.0.16.

Если вы пока не умеете работать с WAR-файлами и основанными на Java веб-серверами, можете установить пакет, предназначенный специально для вашей операционной системы.

D.2.3. Установка специализированного пакета

Установить специализированный пакет для вашей операционной системы также совсем не сложно. Если вы работаете с одной из версий Windows, распакуйте файл `jenkins-<version>.zip` и запустите установщик в формате EXE или MSI. В различных дистрибутивах Linux потребуется запустить соответствующий диспетчер пакетов YUM или RPM. В Mac OS X нужно запустить файл PKG.

В любом случае можно выбрать параметры, предлагаемые установщиком по умолчанию. Если хотите, можете настроить путь установки и другие свойства.

По умолчанию Jenkins сохраняет конфигурацию и задачи в домашнем каталоге пользователя (который в нашей книге называется `$USER`), располагающемся в директории .jenkins. Если вам потребуется отредактировать какую-либо конфигурацию вне пользовательского интерфейса, то и это возможно.

D.2.4. Первый запуск Jenkins

Чтобы убедиться, что Jenkins установлен правильно, следует открыть его информационную панель (личный кабинет) в любом браузере. Обычно эта страница находится по адресу http://localhost:8080/ или http://localhost:8080/jenkins. Должна открыться примерно такая страница, как показана на рис. D.1.

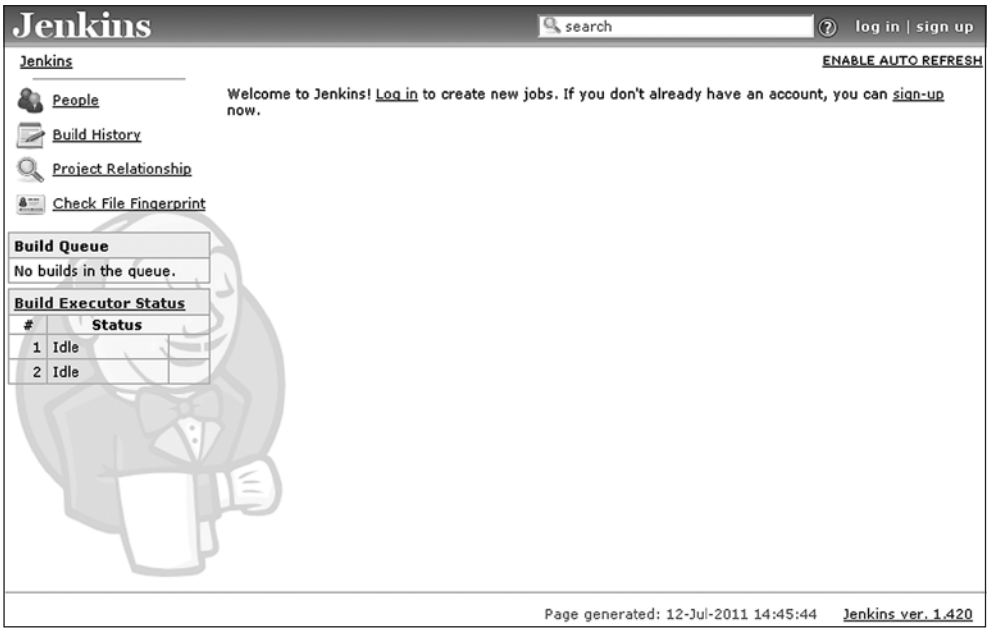


Рис. D.1. Информационная панель Jenkins

Итак, Jenkins установлен, и вы можете приступить к созданию первой задачи. Возвращайтесь к разделу о Jenkins в главе 12!

E java7developer — Maven POM

В этом приложении мы подробно рассмотрим разделы файла `pom.xml`, используемого при сборке проектов Maven. Мы говорили об этом файле в главе 12. В данном приложении подробно разобрана информация, содержащаяся в важных частях файла `pom.xml`, так что, опираясь на приведенные здесь пояснения, вы будете полностью представлять себе весь сборочный цикл. Базовая информация о проекте (см. раздел 12.1) и профили (см. листинг 12.4) были достаточно подробно описаны уже в главе 12. Поэтому здесь мы рассмотрим два следующих раздела POM.

- Конфигурация сборки.
- Зависимости.

Начнем с более длинного раздела, описывающего конфигурацию сборки.

E.1. Конфигурация сборки

В разделе файла `pom.xml`, регламентирующем сборку, описаны плагины и их конфигурация. Эти компоненты нужны для выполнения целей в рамках сборочного цикла Maven. Во многих проектах этот раздел невелик, поскольку зачастую хватает стандартных плагинов с задаваемыми по умолчанию настройками. Но в случае с проектом java7developer в разделе `<build>` содержится несколько плагинов, которые переопределяют стандартные настройки. Мы внесли такие изменения для того, чтобы в проекте java7developer можно было:

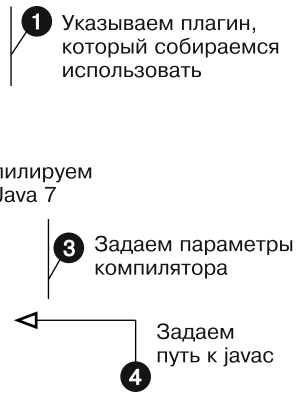
- собирать код Java 7;
- собирать код Scala и Groovy;
- запускать тесты Java, Scala и Groovy;
- предоставлять отчеты с параметрами кода, генерируемые инструментами Checkstyle и FindBugs.

Если вам требуется самостоятельно сконфигурировать еще какие-то характеристики вашей сборки, то можете просмотреть полный список плагинов по адресу http://maven.apache.org/plugins/index.html.

В листинге E.1 показана конфигурация сборки для проекта java7developer.

Листинг E.1. POM — информация о сборке

```
<build>
  <plugins>

    <plugin>

      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-compiler-plugin</artifactId>
      <version>2.3.2</version>

      <configuration>
        <source>1.7</source>
        <target>1.7</target>
        <showDeprecation>true</showDeprecation>
        <showWarnings>true</showWarnings>
        <fork>true</fork>
        <executable>${jdk.javac.fullpath}</executable>
      </configuration>

    </plugin>

    <plugin>
      <groupId>org.scala-tools</groupId>
      <artifactId>maven-scala-plugin</artifactId>
      <version>2.14.1</version>
      <executions>
        <execution>
          <goals>
            <goal>compile</goal>
            <goal>testCompile</goal>
          </goals>
        </execution>
      </executions>
      <configuration>
        <scalaVersion>2.9.0</scalaVersion>
      </configuration>
    </plugin>

    <plugin>
      <groupId>org.codehaus.gmaven</groupId>
      <artifactId>gmaven-plugin</artifactId>
      <version>1.3</version>

      <dependencies>
      <dependency>
      <groupId>org.codehaus.gmaven.runtime</groupId>
      <artifactId>gmaven-runtime-1.7</artifactId>
      <version>1.3</version>
```



```
    </dependency>
    </dependencies>
    <executions>
    <execution>
    <configuration>
    <providerSelection>1.7</providerSelection>
    </configuration>
    <goals>
    <goal>generateStubs</goal>
    <goal>compile</goal>
    <goal>generateTestStubs</goal>
    <goal>testCompile</goal>
    </goals>
    </execution>
    </executions>
  </plugin>

  <plugin>
    <groupId>org.codehaus.mojo</groupId>
    <artifactId>properties-maven-plugin</artifactId>
    <version>1.0-alpha-2</version>
    <executions>
      <execution>
        <phase>initialize</phase>
        <goals>
          <goal>read-project-properties</goal>
        </goals>
        <configuration>
          <files>
            <file>${basedir}/build.properties</file>
          </files>
        </configuration>
      </execution>
    </executions>
  </plugin>
</plugins>

<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-surefire-plugin</artifactId>
  <version>2.9</version>
  <configuration>
    <excludes>
      <exclude>
      com/java7developer/chapter11/listing_11_2
➥ /TicketRevenueTest.java
      </exclude>
      <exclude>
```

6 Исключаем тесты

```
          com/java7developer/chapter11/listing_11_7
➥ /TicketTest.java
        </exclude>
        ...
      </excludes>
    </configuration>
  </plugin>

  <plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-checkstyle-plugin</artifactId>
    <version>2.6</version>
    <configuration>
      <includeTestSourceDirectory>
        true
      </includeTestSourceDirectory>
    </configuration>
  </plugin>

  <plugin>
    <groupId>org.codehaus.mojo</groupId>
    <artifactId>findbugs-maven-plugin</artifactId>
    <version>2.3.2</version>
    <configuration>
      <findbugsXmlOutput>true</findbugsXmlOutput>
      <findbugsXmlWithMessages>
        true
      </findbugsXmlWithMessages>
      <xmlOutput>true</xmlOutput>
    </configuration>
  </plugin>

</build>
```

6 Исключаем тесты

Применяем с тестами Checkstyle

Генерируем отчет FindBugs

Необходимо указать, что вы используете плагин Compiler (и его версию) ❶, так как вы собираетесь изменить стандартный ход компиляции кода Java 1.5 на Java 1.7 ❷.

Поскольку вы уже нарушаете соглашения, можно также добавить несколько полезных предупреждающих параметров компилятора ❸. Еще можно указать, где находится ваша установка Java 7 ❹. Просто скопируйте файл `sample_build.properties` для вашей операционной системы в `build.properties` и измените значение свойства `jdk.javac.fullpath`, чтобы система учла расположение `javac`.

Для того чтобы обеспечить правильную работу плагина Scala, необходимо гарантировать, что он будет выполняться при запуске стадий `compile` и `testCompile` ❺[1].

[1] Ожидается, что следующие версии этого плагина будут автоматически интегрироваться со стадиями.

Конфигурировать тесты можно с помощью плагина Surefire. В конфигурации данного проекта мы опустили несколько тестов (6), которые заведомо неуспешны. Мы обсуждали такие тесты в главе 11, посвященной разработке через тестирование.

Итак, мы рассмотрели сборочный раздел. Теперь перейдем к другой важнейшей части POM — управлению зависимостями.

E.2. Управление зависимостями

Список зависимостей для большинства проектов Java обычно достаточно велик, и проект java7developer не исключение. Система Maven помогает управлять зависимостями — в репозитории Maven Central есть множество предназначенных для этого библиотек. Важно отметить, что в этих сторонних библиотеках есть собственные файлы `pom.xml`, описывающие зависимости, специфичные для данных библиотек. На основании этой информации Maven может догружать дополнительные библиотеки, если это потребуется.

Изначально вы будете работать с двумя основными областями видимости — `compile` и `test`[1]. Настройка этих зависимостей практически сводится к вставке JAR-файлов в `CLASSPATH` для компиляции кода и последующего выполнения тестов.

В листинге E.2 показан раздел `<dependencies>` вашего проекта java7developer.

Листинг E.2. Зависимости POM

```
<dependencies>
  <dependency>
    <groupId>com.google.inject</groupId>
    <artifactId>guice</artifactId>
    <version>3.0</version>
    <scope>compile</scope>
  </dependency>
  <dependency>
    <groupId>javax.inject</groupId>
    <artifactId>javax.inject</artifactId>
    <version>1</version>
    <scope>compile</scope>
  </dependency>
  <dependency>
    <groupId>org.codehaus.groovy</groupId>
    <artifactId>groovy-all</artifactId>
    <version>1.8.6</version>
    <scope>compile</scope>
  </dependency>
```

(1) Уникальный ID артефакта

(2) Область видимости для компиляции

[1] В проектах J2EE/JEE также есть некоторые зависимости, определяемые в области видимости runtime.

```
  <dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-core</artifactId>
    <version>3.6.3.Final</version>
    <scope>compile</scope>
  </dependency>
  <dependency>
    <groupId>org.ow2.asm</groupId>
    <artifactId>asm</artifactId>
    <version>4.0</version>
    <scope>compile</scope>
  </dependency>
  <dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.8.2</version>
    <scope>test</scope>
  </dependency>
  <dependency>
    <groupId>org.mockito</groupId>
    <artifactId>mockito-all</artifactId>
    <version>1.8.5</version>
    <scope>test</scope>
  </dependency>
  <dependency>
    <groupId>org.scalatest</groupId>
    <artifactId>scalatest_2.9.0</artifactId>
    <version>1.6.1</version>
    <scope>compile</scope>
  </dependency>
  <dependency>
    <groupId>org.hsqldb</groupId>
    <artifactId>hsqldb</artifactId>
    <version>2.2.4</version>
    <scope>test</scope>
  </dependency>
  <dependency>
    <groupId>javassist</groupId>
    <artifactId>javassist</artifactId>
    <version>3.12.1.GA</version>
    <scope>test</scope>
  </dependency>
</dependencies>
```

Чтобы Maven мог найти артефакт, на который вы ссылаетесь, нужно сообщить системе верные параметры `<groupId>`, `<artifactId>` и `<version>` ❶. Как было указано выше, если задать для `<scope>` значение `compile` ❷, то соответствующие JAR-файлы будут добавлены в `CLASSPATH` для компиляции кода. Если же `<scope>` будет иметь

значение `test` ❸, то в `CLASSPATH` будут добавлены JAR-файлы, необходимые для тестирования и используемые, когда Maven компилирует и запускает тесты. Библиотека scalatest в этом отношении довольно странная — ее нужно указывать в области `test`, но работать она будет в области видимости `compile` ❹[1].

Конечно, файл `pom.xml` из Maven не назовешь компактным, но ведь мы выполняем здесь трехъязычную сборку (Java, Groovy и Scala), а также настраиваем вывод нескольких отчетных показателей. Полагаем, что в будущем сборочные сценарии Maven для многоязычной разработки станут короче — для этого требуется, чтобы улучшилась общая инструментальная поддержка для данной области.

[1] Полагаем, это несоответствие будет устранено в следующих версиях плагина.

Б. Эванс, М. Вербург

Java. Новое поколение разработки

Перевел с английского О. Сивченко

Заведующий редакцией	*Д. Виницкий*
Ведущий редактор	*Н. Гринчик*
Научные редакторы	*О. Трафимович, В. Хаустов, А. Абловацкий*
Художник	*Л. Адуевская*
Корректор	*Е. Павлович*
Верстка	*Г. Блинов*

ООО «Питер Пресс», 192102, Санкт-Петербург, ул. Андреевская (д. Волкова), 3, литер А, пом. 7Н.

Налоговая льгота — общероссийский классификатор продукции ОК 005-93, том 2; 95 3005 — литература учебная.

Подписано в печать 22.10.13. Формат 70×100/16. Усл. п. л. 45,150. Тираж 1700. Заказ 0000.

Отпечатано в полном соответствии с качеством предоставленных издательством материалов в ГППО «Псковская областная типография». 180004, Псков, ул. Ротная, 34.

ВАМ НРАВЯТСЯ НАШИ КНИГИ? ЗАРАБАТЫВАЙТЕ ВМЕСТЕ С НАМИ!

У Вас есть свой сайт?
Вы ведете блог?
Регулярно общаетесь на форумах? Интересуетесь литературой, любите рекомендовать хорошие книги и хотели бы стать нашим партнером?
ЭТО ВПОЛНЕ РЕАЛЬНО!

СТАНЬТЕ УЧАСТНИКОМ ПАРТНЕРСКОЙ ПРОГРАММЫ ИЗДАТЕЛЬСТВА «ПИТЕР»!

- *Зарегистрируйтесь на нашем сайте в качестве партнера по адресу* ***www.piter.com/ePartners***
- *Получите свой персональный уникальный номер партнера*
- *Выбирайте книги на сайте* ***www.piter.com,*** *размещайте информацию о них на своем сайте, в блоге или на форуме и добавляйте в текст ссылки на эти книги (на сайт* ***www.piter.com****)*

ВНИМАНИЕ! *В каждую ссылку необходимо добавить свой персональный уникальный номер партнера.*

С этого момента получайте 10% от стоимости каждой покупки, которую совершит клиент, придя в интернет-магазин «Питер» по ссылке с Вашим партнерским номером. А если покупатель приобрел не только эту книгу, но и другие издания, Вы получаете дополнительно по 5% от стоимости каждой книги.

Деньги с виртуального счета Вы можете потратить на покупку книг в интернет-магазине издательства «Питер», а также, если сумма будет больше 500 рублей, перевести их на кошелек в системе Яндекс.Деньги или Web.Money.

Пример партнерской ссылки:
http://www.piter.com/book.phtml?978538800282 – обычная ссылка
http://www.piter.com/book.phtml?978538800282**&refer=0000** – партнерская ссылка, где 0000 – это ваш уникальный партнерский номер

Подробно о Партнерской программе ИД «Питер» читайте на сайте WWW.PITER.COM

ИЗДАТЕЛЬСКИЙ ДОМ
ПИТЕР®
WWW.PITER.COM

**ПРЕДСТАВИТЕЛЬСТВА ИЗДАТЕЛЬСКОГО ДОМА «ПИТЕР»
предлагают профессиональную и популярную литературу по различным направлениям: история и публицистика, экономика и финансы, менеджмент и маркетинг, компьютерные технологии, медицина и психология.**

РОССИЯ

Санкт-Петербург: м. «Выборгская», Б. Сампсониевский пр., д. 29а
тел./факс: (812) 703-73-73, 703-73-72; e-mail: sales@piter.com

Москва: м. «Электрозаводская», Семеновская наб., д. 2/1, стр. 1
тел./факс: (495) 234-38-15; e-mail: sales@msk.piter.com

Воронеж: тел.: 8 951 861-72-70; e-mail: voronej@piter.com

Екатеринбург: ул. Бебеля, д. 11а
тел./факс: (343) 378-98-41, 378-98-42; e-mail: office@ekat.piter.com

Нижний Новгород: тел.: 8 960 187-85-50; e-mail: nnovgorod@piter.com

Новосибирск: Комбинатский пер., д. 3
тел./факс: (383) 279-73-92; e-mail: sib@nsk.piter.com

Ростов-на-Дону: ул. Ульяновская, д. 26
тел./факс: (863) 269-91-22, 269-91-30; e-mail: piter-ug@rostov.piter.com

Самара: ул. Молодогвардейская, д. 33а, офис 223
тел./факс: (846) 277-89-79, 229-68-09; e-mail: samara@piter.com

УКРАИНА

Киев: Московский пр., д. 6, корп. 1, офис 33
тел./факс: (044) 490-35-69, 490-35-68; e-mail: office@kiev.piter.com

Харьков: ул. Суздальские ряды, д. 12, офис 10
тел./факс: (057) 7584145, +38 067 545-55-64; e-mail: piter@kharkov.piter.com

БЕЛАРУСЬ

Минск: ул. Розы Люксембург, д. 163
тел./факс: (517) 208-80-01, 208-81-25; e-mail: minsk@piter.com

Издательский дом «Питер» приглашает к сотрудничеству зарубежных торговых партнеров или посредников, имеющих выход на зарубежный рынок
тел./факс: (812) 703-73-73; e-mail: spb@piter.com

Издательский дом «Питер» приглашает к сотрудничеству авторов
тел./факс издательства: (812) 703-73-72, (495) 974-34-50

Заказ книг для вузов и библиотек
тел./факс: (812) 703-73-73, доб. 6250; e-mail: uchebnik@piter.com

Заказ книг по почте: на сайте www.piter.com; по тел.: (812) 703-73-74, доб. 6225