

## 图书在版编目 (C I P) 数据

Java程序员修炼之道 / (英) 埃文斯 (Evans, B. J.),  
(荷) 费尔堡 (Verburg, M.) 著 ; 吴海星译. — 北京：  
人民邮电出版社, 2013.8

(图灵程序设计丛书)

书名原文: The well-grounded Java  
developer:vital techniques of Java 7 and polyglot  
programming

ISBN 978-7-115-32195-4

I. ①J… II. ①埃… ②费… ③吴… III. ①  
JAVA语言—程序设计 IV. ①TP312

中国版本图书馆CIP数据核字(2013)第138623号

## 内 容 提 要

本书分为四部分，第一部分全面介绍 Java 7 的新特性，第二部分探讨 Java 关键编程知识和技术，第三部分讨论 JVM 上的新语言和多语言编程，第四部分将平台和多语言编程知识付诸实践。从介绍 Java 7 的新特性入手，本书涵盖了 Java 开发中最重要的技术，比如依赖注入、测试驱动的开发和持续集成，探索了 JVM 上的非 Java 语言，并详细讲解了多语言项目，特别是涉及 Groovy、Scala 和 Clojure 语言的项目。此外，书中含有大量代码示例，帮助读者从实践中理解 Java 语言和平台。

本书适合 Java 开发人员以及对 Java7 和 JVM 新语言感兴趣的各领域人士阅读。

- 
- ◆ 著 [英] Benjamin J. Evans [荷兰] Martijn Verburg
  - 译 吴海星
  - 责任编辑 刘美英
  - 执行编辑 李洁
  - 责任印制 焦志炜
  - ◆ 人民邮电出版社出版发行 北京市崇文区夕照寺街14号
  - 邮编 100061 电子邮件 315@ptpress.com.cn
  - 网址 <http://www.ptpress.com.cn>
  - 北京艺辉印刷有限公司印刷
  - ◆ 开本: 800×1000 1/16
  - 印张: 26
  - 字数: 658千字 2013年8月第1版
  - 印数: 1~3 000册 2013年8月北京第1次印刷
  - 著作权合同登记号 图字: 01-2012-8794号
- 

定价: 89.00元

读者服务热线: (010)51095186转604 印装质量热线: (010)67129223

反盗版热线: (010)67171154

广告经营许可证: 京崇工商广字第 0021 号

# 序

“Kirk说加油站也卖啤酒。”这是Ben Evans跟我说的第一句话。他来克里特岛参加一个开放型Java会议。我说我通常到加油站就是加油，但那边拐角确实有个店卖啤酒。Ben看起来对我的回答有点儿失望。我在这个希腊小岛上生活了5年，还从来没在加油站买过啤酒。

当我在看这本书时，那种似曾相识的感觉又来了。我自认为是一名Java专家：用Java写了15年程序，发表了几百篇文章，在各种会议中演讲，还执教Java高级课程。可阅读Ben和Martijn的这本大作，经常能给我一些意料之外的启发。他们一开始先介绍了为改变Java生态系统所做的开发工作。类库的内部实现修改起来相对容易，一般也能见到成效。例如，`Arrays.sort()`的内部实现在Java 7中不再用MergeSort算法，而是改用了TimSort。由于这个变化，你不用修改自己对偏序数组进行排序的代码就可能看到性能的提升。然而，修改类文件格式或添加新的VM特性则需要大量工作。Ben了解这些情况，因为他在JCP执行委员会任职。这本书也是关于Java 7的，所以你能接触到Java 7中的所有新特性，比如语法糖的完善、String上的switch、分支/合并，还有Java NIO.2。

并发就是线程和同步，对吗？如果这就是你对多线程的认识，那么你需要学习新知识了。就像作者在书中指出的，“并发领域的研究工作正开展得热火朝天”。与并发相关的邮件列表上每天都有讨论，新点子层出不穷。本书会告诉你如何看待分而治之策略以及如何规避某些安全陷阱。

在我看到类加载那一章时，我觉得他们说得有点儿过了。那都是我和朋友们过去用来炫耀的技巧，居然也给摆出来供大家研习了！他们讲解了javap（这个小工具用于透视Java编译器生成的字节码）的工作方式，还谈到了新的invokedynamic指令，并解释了它跟普通反射的区别。

我特别喜欢讲性能调优的第6章。除了Jack Shirazi的*Java Performance Tuning*，这还是第一本能够抓住“如何使系统运行更快”这个本质问题的书。我可以用四个字来总结这一章的内容：“测量，别猜。”这是做好性能调优的本质，因为人们不可能猜到运行慢的是哪段代码。这一章从硬件的角度解读了性能方面的问题，而不是只提供编码技巧。作者还向你展示了如何测量性能。有一个挺有意思的基础测试小工具——CacheTester类，可用于查看缓存未命中时的开销。

本书第三部分介绍了JVM上的多语言编程。Java不仅仅是Java编程语言，它还是一个可以运行其他语言的平台。我们已经见过不同类型语言的爆炸式增长了。有些是函数式的，有些是声明式的，还有一些是平台的接口（Jython和JRuby），让其他语言可以在JVM上运行。语言分为动态的（如Groovy）和静态的（如Java和Scala）。在JVM上我们可能因为多种原因而使用非Java的语言。如果正好要开始一个新项目，在做决定之前先看看都有什么可用吧。你可能不用再写那么多套路化的代码了。

Ben和Martijn向我们介绍了三种备选语言：Groovy、Scala和Clojure。在我看来，它们是当下最切实可行的选择。作者描述了这些语言之间的差异、与Java的差异以及它们的特性。不需要太多的技术细节，介绍每种语言的各章足以帮你弄清楚应该用哪一种。别指望能在书中看到Groovy的参考手册，但你会了解哪种语言更适合你。

之后，你将深入了解如何进行测试驱动开发以及如何持续集成系统。我发现一件很有意思的事，忠实的“老管家” Hudson这么快就被Jenkins取代了。无论如何，这些工具跟Checkstyle和FindBugs一样都是项目管理的基本工具。

你有望通过研读本书成为一名优秀的Java开发人员。不仅如此，你还能了解如何保持优秀。Java一直在变。下一版中我们将见到lambda表达式和模块化。<sup>①</sup>人们也在不断设计新语言，不断更新并发结构。你现在了解的很多真相将来可能不再是真相。因此，我们必须活到老学到老！

一天，我又开车路过Ben想买啤酒的那个加油站。在经济状况如此低迷的希腊，它也像很多公司一样关张了。我再也不可能知道他们卖不卖啤酒了。

Heinz Kabutz博士

知名Java技术教育家、The Java Specialists'Newsletter创始人

---

<sup>①</sup> 实现Java模块化的Jigsaw项目被延后到Java 9了，至少要到2015年。——译者注

# 前　　言

本书最开始是给德意志银行外汇IT部的新人准备的培训笔记。Ben觉得市面上没有面向经验匮乏的Java开发人员的书，所以决定写一本来填补这个空白。

在德意志银行IT管理团队的支持下，Ben去了比利时的Devoxx会议寻找灵感。在那里他见到了IBM的三位工程师（Rob Nicholson、Zoe Slattery和Holly Cummins），他们把他引荐给了伦敦Java社区（LJC，伦敦Java用户组）。

接下来的周六正好是LJC组织的年度开放会议，就在那次会议上，Ben遇到了LJC的一位领导者——Martijn Verburg。两人一见如故，把酒言欢，惺惺相惜，大有相见恨晚之意。也正是两人对技术和教学的共同热爱促成了本书。

软件开发是一项社会活动，我们希望能借助本书唱响这一主题。我们认为，虽然在这项活动中技术占有很重要的地位，但人与人之间微妙的沟通和交互关系也不容忽视。要在书里轻松解释这些东西不容易，但这一主题自始至终贯穿本书。

凭借着对技术的执着和对学习的热爱，开发人员孜孜不倦地工作着。我们希望本书讨论的一些话题能够激发他们的学习热情。这是一次观光之旅，而不是百科全书式的灌输，这就是我们的初衷：帮助你入门，然后让你自己去探索那些激发你想象力的东西。

本书不仅为大学毕业生准备了指引指南，更为所有心有困惑的Java开发人员提供了指导。因为他们都很想知道：“接下来我该学什么？未来要向什么方向发展？我要再好好考虑考虑！”

从Java 7的新特性到现代软件开发的最佳实践，再到平台的未来发展，本书一路向前，向你展示在成长为资深Java开发人员的过程中我们认为至关重要的那些知识。并发、性能、字节码和类加载是最让我们着迷的核心技术。我们还会谈到JVM上那些新的非Java语言（即多语言编程），因为在接下来的几年里，对于很多开发人员来说它们将变得越来越重要。

归根结底，这是一次以你和你的兴趣为核心的、具有前瞻性的旅程。我们认为成为一名优秀的Java开发人员有助于你彻底投入到工作中去并顺利驾驭开发，也有助于你对不断变化的Java世界及它的周边生态系统有更多了解。

我们希望这本“经验的结晶”对你来说既实用又有趣，希望它能让你深思，同时还能带给你快乐。无论如何，写这本书的体验确实如此！

# 致 谢

老话说得好，“众人拾柴火焰高”。对于本书来说，这句话非常贴切。如果没有朋友、亲人、同事、同行，甚至偶尔跟我们对立的那些人，我们就不可能完成本书。我们一直都非常幸运，因为那些对我们批评最强烈的人也可以算是我们的朋友。

帮助过我们的人太多了，很难全部列举出来。本书快付印的时候，我们在<http://www.java7developer.com>上发过一个帖子，其中也列出了一批名单，那些人值得我们感谢。

如果名单里遗漏了谁，都怪我们没有牢记您的大名，请接受我们的歉意！下面这些人对本书出版都有贡献，在此一并感谢（排名不分先后）。

## 伦敦 Java 社区

伦敦Java社区（LJC，[www.meetup.com/londonjavacomunity](http://www.meetup.com/londonjavacomunity)）是我们两位作者相遇相知的地方。我们要感谢下面这些帮忙审校本书的人：Peter Budo、Nick Harkin、Jodev Devassy、Craig Silk、N. Vanderwilt、Adam J. Markham、“Rozallin”、Daniel Lemon、Frank Appiah、P. Franc、“Sebkom”Praveen、Dinuk Weerasinghe、Tim Murray Brown、Luis Murbina、Richard Doherty、Rashul Hussain、John Stevenson、Gemma Silvers、Kevin Wright、Amanda Waite、Joel Gluth、Richard Paul、Colin Vipurs、Antony Stubbs、Michael Joyce、Mark Hindess、Nuno、Jon Poulton、Adrian Smith、Ioannis Mavroukakis、Chris Reed、Martin Skurla、Sandro Mancuso和Arul Dhesiaseelan。

在Java语言之外，我们得到了James Cook、Alex Anderson、Leonard Axelsson、Colin Howe、Bruce Durling和Russel Winder博士非常认真的指导。在这里要特别感谢他们。

我们还要特别感谢LJC JCP委员会成员：Mike Barker、Trisha Gee、Jim Gough、Richard Warburton、Simon Maple、Somay Nakhal和David Illsley。

最后，感谢LJC的发起人Barry Cranford。四年前，他带领几个勇敢的人，怀抱一个梦想创建了LJC。现在，LJC已经有约2500名成员，并且很多其他技术社区也发源于它。LJC已经成为伦敦技术界的中流砥柱。

[www.coderanch.com](http://www.coderanch.com)

我们要感谢Maneesh Godbole、Ulf Ditmer、David O'Meara、Devaka Cooray、Greg Charles、Deepak Balu、Fred Rosenberger、Jesper De Jong、Wouter Oet、David O'Meara、Mark Spritzler和

Roel De Nijs，因为他们提供了详细的评论和宝贵的反馈。

## Manning 出版社

感谢Manning的Marjan Bace接受我们这两个有着疯狂想法的作者。在制作本书出版的整个过程中，我们跟许多人打过交道，非常感谢他们：Renae Gregoire、Karen G. Miller、Andy Carroll、Elizabeth Martin、Mary Piergics、Dennis Dalinnik和Janet Vail。毫无疑问，还要感谢那些我们从未见过面的幕后工作者。没有他们，就没有这本书！

感谢Candace Gillhoolley在营销上的努力，还有Christina Rudloff和Maureen Spencer一直以来的支持。

感谢John Ryan III在本书付印前对书稿的全面技术审校。

感谢那些在本书编写的不同阶段阅读原稿并给编辑和我们提供宝贵反馈的审校者：Aziz Rahman、Bert Bates、Chad Davis、Cheryl Jerozal、Christopher Haupt、David Strong、Deepak Vohra、Federico Tomassetti、Franco Lombardo、Jeff Schmidt、Jeremy Anderson、John Griffin、Maciej Kreft、Patrick Steger、Paul Benedict、Rick Wagner、Robert Wenner、Rodney Bollinger、Santosh Shanbhag、Antti Koivisto和Stephen Harrison。

## 特别致谢

感谢Andy Burgess为本书开发的网站www.java7developer.com，非常棒。感谢实习生Dragos Dogaru测试了所有代码示例。

感谢Matt Raible非常友善地允许我们在第13章引用他关于如何选择Web框架的一些内容。

感谢Alan Bateman，他领导了Java 7的NIO.2。他的反馈对于向所有Java开发人员普及这个伟大“新API”至关重要。

Jeanne Boyarsky友好地充当了我们最优秀的技术把关者。她果然名不虚传，什么都躲不过她鹰一般犀利的眼睛。感谢Jeanne！

感谢Martin Ling非常细致地讲解了计时硬件，那是第4章介绍相关内容的主要原因。

感谢Jason Van Zyl，他非常友善地允许我们在第12章引用Sonatype Company出版的*Maven: The Complete Reference*中的一些内容。

感谢Kirk Pepperdine对第6章的反馈和意见，还有他的热情及对这个行业独到的见解。

感谢Heinz M. Kabutz博士为我们写的推荐序和他在克里特岛的盛情款待，还有非常赞的Java专家简报（[www.javaspecialists.eu/](http://www.javaspecialists.eu/)）。

## Ben Evans 的致谢

许多人都以不同的方式为本书作出了贡献，但篇幅有限，就不一一致谢了。在此，特别感谢下面这些人。

感谢Bert Bates和其他Manning的员工，他们让我了解了稿件和书之间的区别。

感谢Martijn，当然，为了友谊，为了能帮我度过难关坚持写作，为了很多东西，总之一言难尽。

感谢我的家人，特别是我的外祖父John Hinton和祖父John Evans，我之所以成为我，是因为从他们那里继承了很多东西。

最后，感谢E-J（水獭在书中出现如此频繁的原因）和Liz，太多夜晚因为写作不能陪他们，感谢他们的理解。我的爱，献给他们。

## Martijn Verburg 的致谢

感谢我的妈妈Janneke和爸爸Nico，感谢你们在我和姐姐小时候把Commodore 64电脑带回了家。尽管玩《跳跳人》( 这真的是一个非常非常酷的平台游戏。老妈拿着操纵杆跑来跑去真是太好笑了:-))几乎成了我们在家里用电脑做得最多的事，但正是它的编程手册激发了我对技术的热情。爸爸还教会了我一个道理：如果你能把小事情做对，那么由小事情组成的大事也就不在话下了。我至今依然在编码和日常工作中将它奉为真理。

感谢我的姐姐Kim，感谢你小时候跟我一起写代码！我永远也忘不了第一个星域（缓慢地——那时，我可不太擅长做性能调优）出现在屏幕上时的场景，魔法真的显灵了！姐夫Jos给了我们俩很多灵感（不仅仅因为他是一位空间科学家，尽管那确实很酷！）。我那超级可爱的外甥女Gweneth也出现在了本书里，看你能不能找到她！

Ben是我认识的业内最棒的技术专家之一。有时他的技术水平可以用“吓人”来形容！我很荣幸能跟他合写本书，没想到关于JVM还有那么多知识我不知道。Ben一直都是LJC的优秀领导者，我俩在技术大会上一唱一和的演讲之后，甚至有人认为我们该一块去演喜剧了。能跟朋友合写一本书，真好。

最后，感谢我可爱的妻子Kerry，为了保证本书每一章的插图和屏幕截图都恰到好处，我们一次次地取消了活动计划，而你毫无怨言——你总是那么迷人。愿每个人都能拥有这样的爱和支持。

# 关于本书

欢迎阅读本书。通过阅读本书，你将成为紧跟时代潮流的Java程序员，重燃对这一语言和平台的热情。学习过程中，你会发现Java 7的新特性，熟悉重要的现代软件技术（比如依赖注入、测试驱动开发和持续集成），并开始探索JVM上的非Java语言这个美丽新世界。

首先，我们来看看James Iry在他精彩的博文“*A Brief, Incomplete, and Mostly Wrong History of Programming Languages*”（简明、不完整并且漏洞百出的编程语言历史）中对Java语言的描述：

1996年，James Gosling发明了Java。Java相对繁琐、基于类，是支持垃圾收集、静态类型、单派发的面向对象语言，继承方式为实现单继承和接口多继承。Sun大肆宣扬Java的新颖性。

他对Java的描述基本上是在插科打诨，C#在文中也受到了同等待遇。但作为对一种语言的描述，这种方式也不赖。博文还有很多精彩之处，参见James的博客（<http://james-iry.blogspot.com/>）。没事的时候看看还是挺有收获的。

James的描述的确提出了一个很实际的问题。为什么我们还要讨论一种有将近16年历史的语言呢？它真的已经稳定，没有多少新东西或有意思的事情值得探讨了吗？

如果真是那样，本书就会很薄。事实是，我们依然在谈论Java，因为它的一大优点就是其以下几个核心设计决策之上的构造能力，这些都已经在市场中获得了成功：

- 运行时环境的自动管理（比如垃圾收集、即时编译）；
- 语法简单，核心概念相对较少；
- 保守的语言进化方式；
- 在类库中增加功能和复杂性；
- 广泛、开放的生态系统。

这些设计决策一直在推动着Java世界的创新，简单的核心使得开发门槛很低，而广阔的生态系统使得后来者很容易找到适合自己需要的现成组件。

尽管从历史趋势上来看语言的变化很缓慢，但这些特质使得Java平台和语言既强大又充满活力。Java 7仍然延续了这一趋势。语言的改变是演进式，而不是革命式的。然而，Java 7跟之前版本相比有一个主要区别：它是第一个明确着眼于下一次发布的新版本。根据Oracle有关发布的“B计划”，Java 7为Java 8的主要变化打下了基础。

近年来，JVM上非Java语言的崛起也是一个重大变化。这引发了Java和其他JVM语言之间的相

互融合。现在有大量的项目完全运行在JVM之上（这个数量还在增加），而Java只是它们所用的编程语言之一。

多语言特别是涉及Groovy、Scala和Clojure语言的项目，是当前Java生态系统的一个重要因素，也是本书最后一部分的主题。

## 阅读须知

本书内容大体上适合顺序阅读，但我们也理解某些读者想直奔主题的心情，因此也在一定程度上迎合了这种阅读需求。

我们非常认同自己动手的学习方法，所以建议读者在阅读的同时尝试示例代码。接下来介绍本书主要内容，希望习惯跳跃阅读的读者能从这里找到线索。

本书分四部分：

- 用Java 7做开发；
- 关键技术；
- JVM上的多语言编程；
- 多语种项目开发。

第一部分共两章，都是关于Java 7的内容。本书通篇使用Java 7的语法和语义，所以第1章“初识Java 7”是必读的。那些要处理文件、文件系统和网络I/O的开发人员应该会对第2章“新I/O”特别感兴趣。

第二部分共四章（第3~6章），涉及的主题包括依赖注入、现代并发、类文件/字节码以及性能调优。

第三部分共四章（第7~10章）介绍了JVM上的多语言编程。第7章是必读的，因为这一章介绍的JVM上可用语言的类型和使用是阅读后面章节的基础。接下来的三章分别介绍与Java类似的语言Groovy、兼具OO和函数式特色的混合语言Scala和纯函数式语言Clojure。刚接触函数式编程的开发人员可能需要按顺序阅读，但这几章本身是相互独立的，可以跳着读。

第四部分（最后四章）在之前内容基础上介绍了新内容。虽然各章可以独立阅读，但是在某些部分我们会假定你已经读过之前的内容，或者已经熟悉那些主题。

简言之，如果整本书你必看一章，那就看第1章。如果你会看第三部分，那一定要看第7章。其他各章既可以顺序阅读，也可以独立阅读，但后面的某些章节会假定你已经看过前面的内容。

## 读者对象

本书主要是为那些希望掌握Java语言和平台现代化知识的开发人员写的。如果你想跟上Java 7的步伐，就请阅读本书吧。

如果你想提升一下自己的技能，想搞清楚依赖注入、并发、测试驱动开发之类的话题，本书能为你打下良好的基础。

本书也是为已经认识到多语言编程趋势并想深入下去的开发人员准备的。具体来说，如果你想学习函数式编程，那么本书介绍Scala和Clojure的两章会很有帮助。

## 路线图

第一部分只有两章。第1章介绍了Java 7及其Coin项目，该项目包含很多小巧高效的特性。第2章全面介绍了新I/O API，包括对文件系统API的全面梳理，还介绍了新的异步I/O能力。

第二部分分四章介绍了Java 7的关键技术。第3章告诉你依赖注入技术的源流，接着展示了Java中的标准解决方案Guice 3。第4章阐述在Java中如何正确进行现代并发开发。因为硬件行业坚定地朝着多核处理器方向发展，这个话题再次成为焦点。第5章介绍了JVM的类文件和字节码，揭示了它们的秘密，让你明白Java的工作原理。第6章讲解Java应用程序调优的基础知识，并讨论垃圾收集器等内容。

第三部分介绍JVM上的多语言编程，由四章组成。多语言编程的内容从第7章开始，这里讲述了多语言编程背景知识，以及使用另一种语言的恰当时机。第8章介绍了Groovy——Java动态编程的朋友。Groovy突显了语法相似的动态语言如何大幅提升Java开发人员的生产率。第9章将你带入函数式/OO混合的Scala世界。Scala是一种强大精炼的语言。第10章是为Lisp粉丝们准备的。Clojure被广泛誉为“使用得当的Lisp”，它全面展示了JVM上函数式语言的力量。

第四部分以前三部分的内容为基础，讨论多语言编程技术在几个编程领域涉及的问题。第11章谈到了测试驱动开发，还提供了一个围绕处理模拟对象的方法，给出了一些实战建议。第12章介绍了两种得到广泛应用的工具，用于构建流程中的Maven 3和用于持续集成的Jenkins/Hudson。第13章涵盖了与快速Web开发相关的主题，解释了Java在这一领域的传统缺陷，并提供了一些原型化的新技术（Grails和Compojure）。第14章是对全书的总结和对未来的展望，其中包括Java 8可能支持的新功能。

## 代码约定及下载

首先需要下载和安装的是Java 7。只要找到适合你的OS的发布包，按照下载和安装说明来就行。Oracle网站上Java SE部分有安装包的在线下载和说明，网址是[www.oracle.com/technetwork/java/javase/downloads/index.html](http://www.oracle.com/technetwork/java/javase/downloads/index.html)。

另请参见附录A中的安装说明以及源码运行的指南。

书中所有源码都是等宽字体，以区别于周围的文字。很多代码清单中都有注解，指出其中的关键概念，有时候文中使用带圆圈的数字，对代码进行更详细的注解。我们尽量按照版心的宽度设置代码格式，也在必要时换行，并谨慎使用缩进。但有时候代码行太长，因此会有续行标记。

书中所有示例的源码都可在[www.manning.com/TheWell-GroundedJavaDeveloper](http://www.manning.com/TheWell-GroundedJavaDeveloper)找到。<sup>①</sup>书中

<sup>①</sup> 本书源码也可在图灵社区<http://www.ituring.com.cn/book/1027>下载。——编者注

自始至终都有示例代码。比较长的代码清单有清楚的清单标题，短一些的代码清单就在文字的行与行之间。

## 软件需求

如今，Java 7几乎可运行在任何现代平台上。只要你使用以下某个操作系统，就可以运行源码示例：

- MS Windows XP及以上版本；
- 较新版的\*nix；
- Mac OS X 10.6及以上版本。

多数人都想在IDE中试验代码示例。以下这些主流IDE都对Java 7和最新版的Groovy、Scala、Clojure提供良好支持：

- Eclipse 3.7.1及以上版本；
- NetBeans 7.0.1及以上版本；
- IntelliJ 10.5.2 及以上版本。

我们使用了NetBeans 7.1和Eclipse 3.7.1来创建和运行代码示例。

## 作者在线

购买本书英文版的读者可以免费访问由Manning出版社运营的专用Web论坛，并在论坛中对该书进行评论、提出技术问题、从作者和其他用户那里得到帮助。要访问并订阅该论坛，请访问 [www.manning.com/TheWell-GroundedJavaDeveloper](http://www.manning.com/TheWell-GroundedJavaDeveloper)，这个页面介绍了注册后如何访问论坛、可以得到什么帮助以及论坛上的行为规则。

Manning向读者承诺为读者之间、读者与作者之间提供一个可以对话的场所，但不会强制作者参与，作者在论坛上的贡献都是自愿而且不收费的。为使作者感兴趣，提高其参与度，我们建议读者向作者提一些具有挑战性的问题。

只要本书英文版仍然在售，读者就可以从出版社的网站上访问作者在线论坛和之前讨论话题的归档。

# 关于作者

Ben Evans是伦敦Java用户组的发起人、协助定义Java生态系统标准的Java社区过程执行委员会成员。他在技术圈已经度过了很多年“有趣的时光”，现为一家面向金融业的Java技术公司的CEO。Ben经常在公开场合发表关于Java平台、性能和并发的演讲。

Martijn Verburg（是jClarity的CTO）作为一名技术专家和众多初创企业的OSS导师，拥有十多年的经验。他也是伦敦Java用户组的领导者，带领全球的Java用户组成员为JSR（采用JSR计划）和OpenJDK（采用OpenJDK计划）作出了贡献。作为一位公认的技术团队优化专家，他经常应邀出席Java界的大型会议（JavaOne、Devoxx、OSCON、FOSDEM等）并发表演讲，人送雅号“开发魔头”，赞颂他敢于向行业现状挑战的精神。

# 关于封面图片

本书封面上的画像标题为“卖花人”，摘自19世纪法国出版的沙利文·马雷夏尔（Sylvain Maréchal）四卷本的地域服饰风俗纲要。其中每幅插图都是手工精心绘制并上色的。马雷夏尔这套书展示的丰富服饰，令我们强烈感受到200年前乡村与城镇的巨大文化差异。不同地域的人山水阻隔，言语不通。无论奔走于街巷，还是驻足于乡间，通过他们的服饰，一眼就能看出他们的生活场所、职业，以及生活境况。

时过境迁，书中描绘的那些区域性服饰差异到如今已经不复存在。即使是不同国家，都很难再看出人们着装的区别，再不必说城镇和乡村了。或许，我们今天多姿多彩的人生，正是从前那些文化差异的体现。只不过，如今的生活更加多元，而且技术环境下的生活节奏也更快了。

今时今日，计算机图书层出不穷，Manning就以马雷夏尔这套书中多样性的图片，来表达对IT行业日新月异的发明与创造的赞美。

# 目 录

## 第一部分 用 Java 7 做开发

第 1 章 初识 Java 7.....	2
1.1 语言与平台.....	2
1.2 Coin 项目：浓缩的都是精华.....	4
1.3 Coin 项目中的修改.....	7
1.3.1 switch 语句中的 String.....	7
1.3.2 更强的数值文本表示法.....	8
1.3.3 改善后的异常处理.....	9
1.3.4 try-with-resources (TWR) .....	11
1.3.5 钻石语法 .....	13
1.3.6 简化变参方法调用 .....	14
1.4 小结 .....	15

第 2 章 新 I/O.....	17
------------------	----

2.1 Java I/O 简史 .....	18
2.1.1 Java 1.0 到 1.3 .....	19
2.1.2 在 Java 1.4 中引入的 NIO .....	19
2.1.3 下一代 I/O-NIO.2.....	20
2.2 文件 I/O 的基石：Path.....	20
2.2.1 创建一个 Path.....	23
2.2.2 从 Path 中获取信息 .....	23
2.2.3 移除冗余项 .....	24
2.2.4 转换 Path .....	25
2.2.5 NIO.2 Path 和 Java 已有的 File 类 .....	25
2.3 处理目录和目录树 .....	26
2.3.1 在目录中查找文件 .....	26
2.3.2 遍历目录树 .....	27
2.4 NIO.2 的文件系统 I/O .....	28

2.4.1 创建和删除文件.....	29
2.4.2 文件的复制和移动.....	30
2.4.3 文件的属性 .....	31
2.4.4 快速读写数据 .....	34
2.4.5 文件修改通知 .....	35
2.4.6 SeekableByteChannel .....	37
2.5 异步 I/O 操作.....	37
2.5.1 将来式 .....	38
2.5.2 回调式 .....	40
2.6 Socket 和 Channel 的整合 .....	41
2.6.1 NetworkChannel .....	42
2.6.2 MulticastChannel .....	42
2.7 小结 .....	43

## 第二部分 关键技术

第 3 章 依赖注入.....	46
3.1 知识注入：理解 IoC 和 DI.....	46
3.1.1 控制反转 .....	47
3.1.2 依赖注入 .....	48
3.1.3 转成 DI .....	49
3.2 Java 中标准化的 DI .....	53
3.2.1 @Inject 注解 .....	54
3.2.2 @Qualifier 注解 .....	55
3.2.3 @Named 注解 .....	57
3.2.4 @Scope 注解 .....	57
3.2.5 @Singleton 注解 .....	57
3.2.6 接口 Provider<T> .....	58
3.3 Java 中的 DI 参考实现：Guice 3 .....	59
3.3.1 Guice 新手指南 .....	59
3.3.2 水手绳结：Guice 的各种绑定 .....	62

3.3.3 在 Guice 中限定注入对象的生 命周期	64	第 5 章 类文件与字节码	106
3.4 小结	66	5.1 类加载和类对象	107
<b>第 4 章 现代并发</b>	<b>67</b>	5.1.1 加载和连接概览	107
4.1 并发理论简介	68	5.1.2 验证	108
4.1.1 解释 Java 线程模型	68	5.1.3 Class 对象	108
4.1.2 设计理念	69	5.1.4 类加载器	109
4.1.3 这些原则如何以及为何会相互 冲突	70	5.1.5 示例：依赖注入中的 类加载器	110
4.1.4 系统开销之源	71	5.2 使用方法句柄	111
4.1.5 一个事务处理的例子	71	5.2.1 MethodHandle	112
4.2 块结构并发（Java 5 之前）	72	5.2.2 MethodType	112
4.2.1 同步与锁	73	5.2.3 查找方法句柄	113
4.2.2 线程的状态模型	74	5.2.4 示例：反射、代理与方法 句柄	114
4.2.3 完全同步对象	74	5.2.5 为什么选择 MethodHandle	116
4.2.4 死锁	76	5.3 检查类文件	117
4.2.5 为什么是 synchronized	77	5.3.1 介绍 javap	117
4.2.6 关键字 volatile	78	5.3.2 方法签名的内部形式	118
4.2.7 不可变性	79	5.3.3 常量池	119
4.3 现代并发应用程序的构件	80	5.4 字节码	121
4.3.1 原子类：java.util. concurrent.atomic	81	5.4.1 示例：反编译类	121
4.3.2 线程锁：java.util. concurrent.locks	81	5.4.2 运行时环境	123
4.3.3 CountDownLatch	85	5.4.3 操作码介绍	124
4.3.4 ConcurrentHashMap	86	5.4.4 加载和储存操作码	125
4.3.5 CopyOnWriteArrayList	87	5.4.5 数学运算操作码	125
4.3.6 Queue	90	5.4.6 执行控制操作码	126
4.4 控制执行	95	5.4.7 调用操作码	126
4.4.1 任务建模	96	5.4.8 平台操作操作码	127
4.4.2 ScheduledThread- PoolExecutor	97	5.4.9 操作码的快捷形式	127
4.5 分支/合并框架	98	5.4.10 示例：字符串拼接	127
4.5.1 一个简单的分支/合并例子	99	5.5 invokedynamic	129
4.5.2 ForkJoinTask 与工作窃取	101	5.5.1 invokedynamic 如何工作	129
4.5.3 并行问题	102	5.5.2 示例：反编译 invokedynamic 调用	130
4.6 Java 内存模型	103	5.6 小结	132
4.7 小结	104	<b>第 6 章 理解性能调优</b>	133
		6.1 性能术语	134
		6.1.1 等待时间	135
		6.1.2 吞吐量	135

6.1.3 利用率	135
6.1.4 效率	135
6.1.5 容量	136
6.1.6 扩展性	136
6.1.7 退化	136
6.2 务实的性能分析法	136
6.2.1 知道你在测量什么	137
6.2.2 知道怎么测量	137
6.2.3 知道性能目标是什么	138
6.2.4 知道什么时候停止优化	139
6.2.5 知道高性能的成本	139
6.2.6 知道过早优化的危险	140
6.3 哪里出错了？我们担心的原因	140
6.3.1 过去和未来的性能趋势： 摩尔定律	141
6.3.2 理解内存延迟层级	142
6.3.3 为什么 Java 性能调优存在 困难	143
6.4 一个来自于硬件的时间问题	144
6.4.1 硬件时钟	144
6.4.2 麻烦的 nanoTime()	144
6.4.3 时间在性能调优中的作用	146
6.4.4 案例研究：理解缓存未命中	147
6.5 垃圾收集	149
6.5.1 基本算法	149
6.5.2 标记和清除	150
6.5.3 jmap	152
6.5.4 与 GC 相关的 JVM 参数	155
6.5.5 读懂 GC 日志	156
6.5.6 用 VisualVM 查看内存使用 情况	157
6.5.7 逃出分析	159
6.5.8 并发标记清除	160
6.5.9 新的收集器：G1	161
6.6 HotSpot 的 JIT 编译	162
6.6.1 介绍 HotSpot	163
6.6.2 内联方法	164
6.6.3 动态编译和独占调用	165
6.6.4 读懂编译日志	166
6.7 小结	167

### 第三部分 JVM 上的多语言编程

第 7 章 备选 JVM 语言	170
7.1 Java 太笨？纯粹诽谤	170
7.1.1 整合系统	171
7.1.2 函数式编程的基本原理	172
7.1.3 映射与过滤器	173
7.2 语言生态学	174
7.2.1 解释型与编译型语言	175
7.2.2 动态与静态类型	175
7.2.3 命令式与函数式语言	176
7.2.4 重新实现的语言与原生语言	176
7.3 JVM 上的多语言编程	177
7.3.1 为什么要用非 Java 语言	178
7.3.2 崭露头角的语言新星	179
7.4 如何挑选称心的非 Java 语言	180
7.4.1 低风险	181
7.4.2 与 Java 的交互操作	181
7.4.3 良好的工具和测试支持	182
7.4.4 备选语言学习难度	182
7.4.5 使用备选语言的开发者	182
7.5 JVM 对备选语言的支持	183
7.5.1 非 Java 语言的运行时环境	183
7.5.2 编译器小说	184
7.6 小结	185
第 8 章 Groovy：Java 的动态伴侣	187
8.1 Groovy 入门	189
8.1.1 编译和运行	189
8.1.2 Groovy 控制台	190
8.2 Groovy 101：语法和语义	191
8.2.1 默认导入	192
8.2.2 数字处理	192
8.2.3 变量、动态与静态类型、 作用域	193
8.2.4 列表和映射语法	195
8.3 与 Java 的差异——新手陷阱	196
8.3.1 可选的分号和返回语句	196
8.3.2 可选的参数括号	197
8.3.3 访问限定符	197

8.3.4 异常处理	198	9.4.5 case 类和 match 表达式	232
8.3.5 Groovy 中的相等	198	9.4.6 警世寓言	234
8.3.6 内部类	199	9.5 数据结构和集合	235
8.4 Java 不具备的 Groovy 特性	199	9.5.1 List	235
8.4.1 GroovyBean	199	9.5.2 Map	238
8.4.2 安全解引用操作符	200	9.5.3 泛型	239
8.4.3 猫王操作符	201	9.6 actor 介绍	242
8.4.4 增强型字符串	201	9.6.1 代码大舞台	242
8.4.5 函数字面值	202	9.6.2 用 mailbox 跟 actor 通信	243
8.4.6 内置的集合操作	203	9.7 小结	244
8.4.7 对正则表达式的内置支持	204		
8.4.8 简单的 XML 处理	205		
8.5 Groovy 与 Java 的合作	207	<b>第 10 章 Clojure：更安全地编程</b>	245
8.5.1 从 Groovy 调用 Java	207	10.1 Clojure 介绍	245
8.5.2 从 Java 调用 Groovy	208	10.1.1 Clojure 的 Hello World	246
8.6 小结	211	10.1.2 REPL 入门	247
<b>第 9 章 Scala：简约而不简单</b>	212	10.1.3 犯了错误	248
9.1 走马观花 Scala	213	10.1.4 学着去爱括号	248
9.1.1 简约的 Scala	213		
9.1.2 match 表达式	215	10.2 寻找 Clojure：语法和语义	249
9.1.3 case 类	217	10.2.1 特殊形式新手营	249
9.1.4 actor	218	10.2.2 列表、向量、映射和集	250
9.2 Scala 能用在我的项目中吗	219	10.2.3 数学运算、相等和其他操作	252
9.2.1 Scala 和 Java 的比较	219	10.3 使用函数和循环	253
9.2.2 何时以及如何开始使用 Scala	220	10.3.1 一些简单的 Clojure 函数	253
9.2.3 Scala 可能不适合当前项目的 迹象	220	10.3.2 Clojure 中的循环	255
9.3 让代码因 Scala 重新绽放	221	10.3.3 读取器宏和派发器	256
9.3.1 使用编译器和 REPL	221	10.3.4 函数式编程和闭包	257
9.3.2 类型推断	222	10.4 Clojure 序列	258
9.3.3 方法	223	10.4.1 懒序列	260
9.3.4 导入	224	10.4.2 序列和变参函数	261
9.3.5 循环和控制结构	224	10.5 Clojure 与 Java 的互操作	262
9.3.6 Scala 的函数式编程	225	10.5.1 从 Clojure 中调用 Java	262
9.4 Scala 对象模型：相似但不同	226	10.5.2 Clojure 值的 Java 类型	263
9.4.1 一切皆对象	226	10.5.3 使用 Clojure 代理	264
9.4.2 构造方法	228	10.5.4 用 REPL 做探索式编程	264
9.4.3 特质	228	10.5.5 在 Java 中使用 Clojure	265
9.4.4 单例和伴生对象	230	10.6 Clojure 并发	265
		10.6.1 未来式与并行调用	266
		10.6.2 ref 形式	267
		10.6.3 代理	271
		10.7 小结	272

## 第四部分 多语种项目开发

<b>第 11 章 测试驱动开发</b>	274
11.1 TDD 概览	275
11.1.1 一个测试用例	276
11.1.2 多个测试用例	280
11.1.3 深入思考红—绿—重构循 环	282
11.1.4 JUnit	283
11.2 测试替身	285
11.2.1 虚设对象	286
11.2.2 存根对象	287
11.2.3 伪装替身	290
11.2.4 模拟对象	295
11.3 ScalaTest	296
11.4 小结	298
<b>第 12 章 构建和持续集成</b>	300
12.1 与 Maven 3 相遇	302
12.2 Maven 3 入门项目	303
12.3 用 Maven 3 构建 Java7developer 项目	305
12.3.1 POM	305
12.3.2 运行示例	311
12.4 Jenkins：满足 CI 需求	314
12.4.1 基础配置	315
12.4.2 设置任务	316
12.4.3 执行任务	319
12.5 Maven 和 Jenkins 代码指标	320
12.5.1 安装 Jenkins 插件	321
12.5.2 用 Checkstyle 保持代码 一致性	322
12.5.3 用 FindBugs 设定质量 标杆	323
12.6 Leiningen	325
12.6.1 Leiningen 入门	326
12.6.2 Leiningen 的架构	326
12.6.3 Hello Lein	327
12.6.4 用 Leiningen 做面向 REPL 的 TDD	329

12.6.5 用 Leiningen 打包和部署	330
12.7 小结	332
<b>第 13 章 快速 Web 开发</b>	333
13.1 Java Web 框架的问题	334
13.1.1 Java 编译为什么不好	335
13.1.2 静态类型为什么不好	335
13.2 选择 Web 框架的标准	336
13.3 Grails 入门	338
13.4 Grails 快速启动项目	338
13.4.1 创建域对象	340
13.4.2 测试驱动开发	340
13.4.3 域对象持久化	342
13.4.4 创建测试数据	343
13.4.5 控制器	343
13.4.6 GSP/JSP 页面	344
13.4.7 脚手架和 UI 的自动化 创建	346
13.4.8 快速周转的开发	347
13.5 深入 Grails	347
13.5.1 日志	347
13.5.2 GORM：对象关系映射	348
13.5.3 Grails 插件	349
13.6 Compojure 入门	350
13.6.1 Hello Compojure	350
13.6.2 Ring 和路由	352
13.6.3 Hiccup	353
13.7 我是不是一只水獭	353
13.7.1 项目设置	354
13.7.2 核心函数	357
13.8 小结	359
<b>第 14 章 保持优秀</b>	361
14.1 对 Java 8 的期待	361
14.1.1 lambda 表达式（闭包）	362
14.1.2 模块化（拼图 Jigsaw）	363
14.2 多语言编程	365
14.2.1 语言的互操作性及元对象 协议	365
14.2.2 多语言模块化	366

14.3	未来的并发趋势 .....	367	附录 A	java7developer: 源码安装 .....	373
14.3.1	多核的世界 .....	367	附录 B	glob 模式语法及示例 .....	380
14.3.2	运行时管理的并发 .....	367	附录 C	安装备选 JVM 语言 .....	382
14.4	JVM 的新方向 .....	368	附录 D	Jenkins 的下载和安装 .....	388
14.4.1	VM 的合并 .....	368	附录 E	java7developer: Maven POM .....	390
14.4.2	协同程序 .....	369			
14.4.3	元组 .....	370			
14.5	小结 .....	372			

# Part 1

第一部分

## 用 Java 7 做开发

本书前两章主要讨论 Java 7 的高明之处。为便于读者理解下文，第 1 章先介绍了一些可提高开发人员工作效率的语法变化，这些变化并不大，但效果都比较显著。第 1 章在这一部分中主要起抛砖引玉的作用，而另一个主题，Java 中的新 I/O 才是主角。

优秀的 Java 开发人员要了解语言的新特性。Java 7 中的新特性可以使开发人员的工作变得更轻松。但对于这些新变化，光了解语法是不够的。为了能迅速写出高效、安全的代码，你还需要对实现这些新特性的原因和方式有深刻的认识。Java 7 的变化可以大致分为两块：Coin 项目和 NIO.2。

第一块是 Coin 项目，包括语言层面的一些小变化，设计它们的初衷是提高开发人员的生产率，但又不会对底层平台造成太大影响。这些变化包括：

- try-with-resources 结构（可以自动关闭资源）；
- switch 中的字符串；
- 对数字常量的改进；
- Multi-catch（在一个 catch 块中声明多个要捕获的异常）；
- 钻石语法（在处理泛型时不用那么繁琐了）。

这些变化看起来都不大，但探索这些简单的语法修改背后的语义迁移，能让你洞察 Java 语言和 Java 平台之间的差别。

第二块变化是新 I/O（NIO.2）API，跟 Java 原有的文件系统支持相比，它具有压倒性优势，还提供了强大的异步能力。这些变化包括：

- 用于引用文件和类文件实体的新 Path 结构；
- 简化文件的创建、复制、移动和删除的工具类 Files；
- 内建的目录树导航；
- 在后台处理大型 I/O 的将来式和回调式异步 I/O。

第一部分结束时，你会很自然地用 Java 7 的方式来思考问题和编写代码。我们在后续章节中还会用到 Java 7 中的新特性，所以你还有机会不断温习这些新知识。



### 本章内容

- Java既是编程语言，也是平台
- 语法变一点，能力强好多
- try-with-resources语句
- 提升异常处理能力

欢迎进入Java 7的世界！斗转星移，时过境迁。当尘埃落定，我们终于见到了Java 7的真容。虽然看起来有点陌生，但它必将带来全新的体验！跟随我们经历过这段探索之旅，你将进入更广阔的世界，发现更多新特性、更高明的编程技巧，并接触到JVM上运行的更多编程语言。

现在，我们先来热热身。虽然只是简单介绍，但还是能让你了解Java 7的强大特性。我们会先解释Java语言和平台的区别，因为有时人们会对这两种说法产生误解。

接着我们会介绍Coin项目，它汇聚了Java 7里短小精悍的新特性。我们会向你展示Java平台所接受、吸纳和发布的那些特性，就是它们构成了Java的变化。在此之后，我们会介绍Coin项目中新引入的6个主要特性。

你会学到新的语法，比如改进的异常处理方式（multi-catch）以及try-with-resources结构，借此在处理文件或其他资源的代码中躲开那些烦人的bug。读完本章，你将能用全新的方式编写Java代码，并整装待发，准备好迎接更大的挑战！

让我们先来探讨一下Java作为语言和平台的双重角色，这是现代Java的核心。这个知识点将贯穿全书，是一个必须掌握的基本要点。

## 1.1 语言与平台

使用Java之前，我们要先弄清楚Java语言和Java平台之间的区别。然而，有时候不同的作者对语言和平台的构成会有不同的定义，所以人们有时不太清楚两者之间的区别，分不清是语言还是平台提供了代码使用的编程特性。

因为本书的大部分内容都需要你理解两者的区别，所以这里需要说明一下。以下是我们给出的定义。

- Java语言 在“关于本书”中，我们提到Java语言是静态类型、面向对象的语言，希望你对这种说法已经非常熟悉了。Java语言还有一个非常明显的特点，它是（或者说应该是）人类可读的。
- Java平台 平台是提供运行时环境的软件。Java虚拟机（JVM）负责把类文件形式（人类不可读）的代码链接起来并执行。JVM不能直接解释Java语言的源文件，你要先把源文件转换成类文件。

Java作为软件系统之所以能成功，主要因为它是一种标准。也就是说，它有规范文件描述它应该如何工作。不同的厂商或项目组可以据此推出自己的实现，这些不同实现的工作方式在理论上是相同的。规范虽然不能保证这些实现处理同一任务时表现如何，但可以保证处理结果的正确性。

控制Java系统的规范有多种，其中最重要的是《Java语言规范》(JLS)和《JVM规范》(VMSpec)。在Java 7中，这两者之间的界限愈发清晰。实际上，VMSpec不再引用JLS中的任何内容，如果你认为这是Java 7重视Java之外其他语言的信号，说明你有见微知著的能力！希望你能继续关注，接下来我们会更加深入地探讨这两个规范之间的差别。

提到Java的双重角色，你自然想问：“它们两者之间还有什么关联吗？”如果它们在Java 7中如此泾渭分明，又是如何共同形成我们所熟悉的Java系统的呢？

连接Java语言和平台之间的纽带是统一的类文件（即.class文件）格式定义。认真研究类文件的定义能让你获益匪浅，这是优秀Java程序员向伟大Java程序员转变的一个途径。图1-1展示了产生和使用Java代码的整个过程。

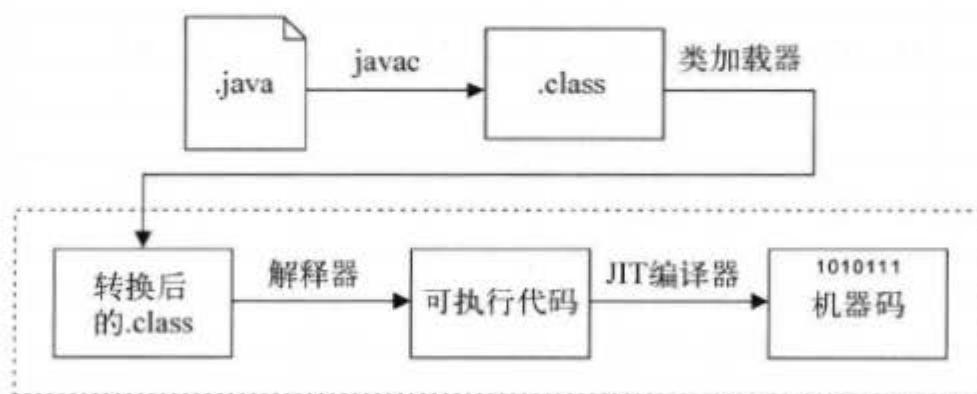


图1-1 Java源码被转换成.class文件，在JIT编译前被加载处理

如图所示，Java代码的演进过程从我们可以看懂的Java源码开始，然后由javac编译成.class文件，变成可以加载到JVM中的形式。值得注意的是，类文件在加载过程中通常都会被处理和修改。大多数流行框架（特别是打着“企业级”旗号的）都会在类加载过程中对类进行改造。

### Java是编译型语言还是解释型语言？

大多数开发人员都知道，Java源文件需要编译成.class文件才能在JVM中运行。如果继续追问，许多开发人员还会告诉你说.class中的字节码首先会被JVM解释，但在稍后即时（JIT）编译。然而很多人将字节码含糊地理解为“在某种虚构的或简化的CPU上运行的机器码”。

实际上，JVM字节码更像是中途的驿站，是一种从人类可读的源码向机器码过渡的中间状态。用编译原理术语讲，字节码实际上是一种中间语言（IL）形态，不是真正的机器码。也就是说，将Java源码变成字节码的过程不是C或C++程序员所理解的那种编译。Java所谓的编译器javac也不同于gcc，实际上它只是一个针对java源码生成类文件的工具。Java体系中真正的编译器是JIT，如图1-1所示。

有人说Java是“动态编译”的，他们所说的编译是指JIT的运行时编译，不是指构建时创建类文件的过程。

所以如果被问及“Java是编译型语言还是解释型语言”，你可以回答“都是”。

希望我们已经把Java语言和Java平台之间的区别解释清楚了。接下来我们进入下一话题，看看Java 7中一些语法上的调整，先从Coin项目中的那些小变化开始。

## 1.2 Coin项目：浓缩的都是精华

自2009年1月起，Coin便是Java 7（和Java 8）中一个开源的子项目。本节，我们会以Coin项目中包含的小变化为例，解释一下Java语言如何演进以及那些特性是如何被选中的。

### 为Coin项目命名

创建Coin项目是为了反映Java语言中的微小变动。项目的名字是个双关语——像硬币一样小的变化（small change comes as coins），而“套用一句老话”（to coin a phrase）指的是给Java语言添一个新的表述方式。

在技术圈子里，这种文字游戏、奇思妙想和躲不掉的恐怖双关语随处可见。你可能已经对此习以为常了。

我们觉得解释语言“为什么要变”和“变成了什么”同样重要。在开发Java 7的过程中，人们对新语言特性产生了很多兴趣，但Java社区有时并不明白要按时实现这些特性需要多大工作量。希望我们在“为什么要变”这一问题上能够对你有所启示，也希望能借此消除一些荒诞的说法。如果你对Java如何发展进化不感兴趣，可以直接阅读1.3节，看看Java语言发生了哪些变化。

关于修改Java语言有一个工作量曲线，某些实现方式可能要比其他方式需要的工作量少。如图1-2所示，我们尽量把不同的修改方式及其所需的工作量呈现出来，以展现不同复杂度及相关工作量的增长。

一般来说，工作量越少越好。也就是说，如果能用类库实现新特性，那就应该用类库。但有些特性用类库或增加IDE功能实现起来有难度，甚至根本不可能，那就必须在平台的更深层中实现。

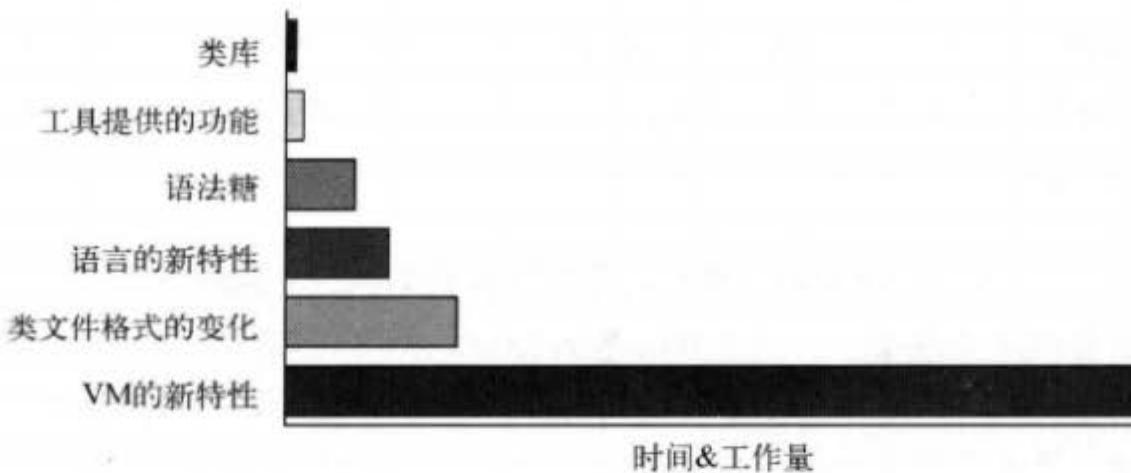


图1-2 用不同方式提供新功能所需工作量的比较

下面是一些特性（大部分是Java 7中的），它们符合我们为语言新特性定义的复杂度。

- 语法糖——数字中的下划线（Java 7）；
- 新的语言小特性——try-with-resources（Java 7）；
- 类文件格式的变化——注解（Java 5）；
- JVM的新特性——动态调用（Java 7）。

### 语 法 糖

“语法糖”是描述一种语言特性的短语。它表示这是冗余的语法——在语言中已经存在一种表示形式了——但语法糖用起来更便捷。

一般来说，程序的语法糖在编译处理早期会从编译结果中移除，变为相同特性的基础表示形式，这称为“去糖化”。

因此，语法糖是比较容易实现的修改。它们通常不需要做太多工作，只需要修改编译器（对于Java来说就是javac）。

Coin项目中（以及本章余下的内容）都是关于从Java语法糖到小的新特性这个范围之内的变化。

Coin项目的最初建议阶段从2009年2月持续到3月，coin-dev的邮件列表上收到了大约70个提议，囊括了各种可能的改进。甚至有人开玩笑说要增加lolcat风格的多行字符串（把标题叠加在好笑或生气的猫咪图片上，看你怎么想了——<http://icanhascheezburger.com/>）。

Coin项目提案的评判规则很简单。贡献者要完成三项任务：

- 提交一份详细的提案来描述修改（本质上应该是对Java语言的修改，而不是针对虚拟机的）；
- 在邮件列表上针对提案进行开放式讨论，能够接受其他参与者建设性的批评和建议；
- 随时可以提供一组能够实现变化的补丁原型。

Coin项目很好地示范了语言和平台在未来如何演进，所有的修改都会公开讨论，提供特性的早期原型，并且呼吁公众参与。

“什么是对规范的小修改？”现在也许就是提出这个问题的最好时机。我们马上要讨论在JLS的第14.11节中增加的一个单词——“String”。你可能再也找不出比这个更小的修改了，可即便是这样一个修改都会涉及规范中其他几个地方。

### Java 7是以开源方式开发后发布的第一个版本

Java一开始并不是开源语言，但在2006年的JavaOne大会上其自身的源码以GPLv2许可发布（去掉了一些Sun不具有版权的源码）。当时正值Java 6发布前后，所以Java 7是Java在开源软件（OSS）许可下发布的第一版。开源的Java平台开发主要集中在项目OpenJDK上。

邮件列表coin-dev、lambda-dev和mlvm-dev等是讨论未来各种可能特性的主要场所，来自五湖四海的开发人员都可以借此参与到创造Java 7的过程中来。实际上，我们参与领导了“Adopt OpenJDK”计划，为新加入OpenJDK的开发人员提供指导，帮助改进Java。如果你想加入我们，请访问<http://java.net/projects/jugs/pages/AdoptOpenJDK>。

任何修改都会产生影响，我们必须从Java语言的整体设计上来追踪这些影响。

任何修改都应该严格执行下面这些操作（或至少调研一下）：

- 更新JLS；
- 在源码编译器中实现一个原型；
- 为修改增加必要的类库支持；
- 编写测试和示例；
- 更新文档。

除此之外，如果修改触及VM或者平台，应该：

- 更新VMSpec；
- 实现VM的修改；
- 在类文件和VM工具中增加支持；
- 考虑对反射的影响；
- 考虑对序列化的影响；
- 想一想对本地代码组件的影响，比如Java本地接口（JNI）。

考虑到这些修改对整体语言规范产生的影响，这可不是一星半点儿的工作。

如果你要修改类型系统，简直就是自寻死路，类型系统可是个不折不扣的雷区。这不是因为Java的类型系统不好，而是因为对于拥有多种静态类型系统的语言来说，这些类型系统之间有可能会产生很多交叉点。修改它们很容易出现意想不到的状况。

Coin项目选的路线非常明智，它建议贡献者们在修改提案中远离类型系统。因为对这种修改而言，即使最小的变化都需要做大量的工作，所以这种做法也是比较务实的。

在简单介绍了Coin项目的背景之后，接下来该看看它包含哪些特性了。

## 1.3 Coin 项目中的修改

Coin项目主要给Java 7引入了6个新特性，它们分别是switch语句中的String、数字常量的新形式、改进的异常处理、try-with-resources、钻石语法，还有变参警告位置的修改。

我们会详细讲解Coin项目中的这些变化，讨论这些新特性的语法和含义，并尽可能解释推出这些特性背后的动机。当然，我们也不是要把提案全部照搬过来，coin-dev邮件列表的归档里有完整的提案，如果你是一个好奇的语言设计师，可以去那里看看，还可以和大家讨论你的想法。

闲言少叙，开始介绍第一个Java 7新特性——switch语句中的string值。

### 1.3.1 switch 语句中的 String

switch语句是一种高效的多路语句，可以省掉很多繁杂的嵌套if判断，比如像这样：

```
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;  
    }  
}
```

在Java 6及之前，case语句中的常量只能是byte、char、short和int（也可以是对应的封装类型Byte、Character、Short和Integer）或枚举常量。Java 7规范中增加了String，毕竟它也是常量类型。

```
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;  
    }  
}
```

除此之外，switch语句和以前完全一样。像Coin项目中的许多新特性一样，这不过是一个让你更轻松的小小改进。

### 1.3.2 更强的数值文本表示法

当时有几个与整型语法相关的提案，最终被选中的是下面这两个：

- 数字常量（如基本类型中的integer）可以用二进制文本表示；
- 在整型常量中可以使用下划线来提高可读性。

这两个改变乍看起来都不起眼，但它们确实解决了一直困扰着Java程序员的一些小麻烦。

这两个新特性对系统底层程序员，就是那些整天处理原始网络协议、加密或沉迷于摆弄比特的人们特别有用。先来看一下二进制文本。

#### 1. 二进制文本

在Java 7之前，如果要处理二进制值，就必须借助棘手（又容易出错）的基础转换，或者调用parseX方法。比如说，如果想让int x用位模式表示十进制值102，你可以这样写：

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

为了确保x是正确的位模式，你需要敲许多代码。这种方式尽管看起来还行，但实际上存在很多问题：

- 十分繁琐；
- 方法调用对性能有影响；
- 需要知道parseInt()的双参形式；
- 需要记住双参的parseInt()的处理细节；
- JIT编译器更难实现；
- 用运行时的表达式表示应该在编译时确定的常量，导致x不能用在switch语句中；
- 如果在位模式中有拼写错误（能通过编译），会在运行时抛出RuntimeException。

现在好了，用Java 7可以写成：

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

我们没说这种方法无所不能，但它确实解决了上面提到的那些问题。

你在跟二进制打交道时，这个小特性会是你的得力助手。比如在处理字节时，可以在switch语句中使用由位模式定义的二进制常量。

另外一个新特性虽然小，但却很实用——可以在表示一组二进制位或其他长数值的数字中加入下划线。

#### 2. 数字中的下划线

众所周知，人脑和电脑有很多不同的地方，对于数字的处理方式就是其中之一。通常人们都不太喜欢面对一大串数字。这也是我们发明十进制的原因之一——因为人脑更擅于处理信息量大的短字串，而不是每个字符信息量都不太多的长字串。

也就是说，我们觉得1c372ba3要比00011100001101110010101110100011更容易处理，但电脑只认第二种。人们在处理长串数字时会采用分隔法，比如用404-555-0122表示电话号码。

**注意** 如果你跟作者(欧洲人)一样,想知道为什么美国电影或书里的电话号码总是以555开头,我可以告诉你。555-01xx是保留号段,用于虚构的情境。这是为了避免现实生活中的人接到那些对好莱坞电影过分投入的人打来的电话。

其他带有分隔符的一长串数字:

- 100 000 000美元(一大笔钱);
- 08-92-96(英国银行的排序代码)。

可在代码中处理数字时不能用逗号(,)和连字符(-)作分隔符,因为它们可能会引发歧义。Coin项目中的提案借用了Ruby的创意,用下划线(\_)作分隔符。注意,这只是为了让你阅读数字时更容易理解而做的一个小修改,编译器会在编译时把这些下划线去掉,只保留原始数字。

也就是说,为了不把100 000 000和10 000 000搞混,你可以在代码中将100 000 000写成100\_000\_000,以便很容易区分它和10\_000\_000的差别。来看下面两个例子,至少你应该对其中一个比较熟悉:

```
long anotherLong = 2_147_483_648L;
int bitPattern = 0b0001_1100_0011_0111_0010_1011_1010_0011;
```

注意:赋给anotherLong的数值现在看起来清楚多了。

**警告** 在Java中可以用小写字母l表示长整型数值,比如10101001。但最好还是用大写字母L,以免维护代码的人把数字l和字母l搞混:1010100L看起来要清楚得多。

现在你应该清楚这些变化给整数处理带来的好处了!让我们继续前进,去看看Java 7中的异常处理。

### 1.3.3 改善后的异常处理

异常处理有两处改进——multicatch和final重抛。要知道它们对我们有什么帮助,请先看一段Java 6代码。下面这段代码试图查找、打开、分析配置文件并处理此过程中可能出现的各种异常:

#### 代码清单1-1 在Java 6中处理不同的异常

```
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;
}

```

这个方法会遇到的下面几种异常：

- 配置文件不存在；
- 配置文件在正要读取时消失了；
- 配置文件中有语法错误；
- 配置文件中可能包含无效信息。

这些异常可以分为两大类。一类是文件以某种方式丢失或损坏，另一类是虽然文件理论上存在并且是正确的，却无法正常读取（可能是因为网络或硬件故障）。

如果能把这些异常情况简化为这两类，并且把所有“文件以某种方式丢失或损坏”的异常放在一个catch语句中处理会更好。在Java 7中就可以做到：

### 代码清单1-2 在Java 7中处理不同的异常

```

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;
}

```

异常e的确切类型在编译时还无法得知。这意味着在catch块中只能把它当做可能异常的共同父类（在实际编码时经常用Exception或Throwable）来处理。

另外一个新语法可以为重新抛出异常提供帮助。开发人员经常要在重新抛出异常之前对它进行处理。在前几个版本的Java中，经常可以看到下面这种代码：

```

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

```

这会强迫你把新抛出的异常声明为Exception类型——异常的真实类型却被覆盖了。

不管怎样，很容易看出来异常只能是IOException或SQLException。既然你能看出来，编译器当然也能。下面的代码中用了Java 7的语法，只改了一个单词：

```

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

```

关键字final表明实际抛出的异常就是运行时遇到的异常——在上面的代码中就是IOException或SQLException。这被称为final重抛，这样就不会抛出笼统的异常类型，从而避免在上层只能用笼统的catch捕获。

上例中的关键字final不是必需的，但实际上，在向catch和重抛语义调整的过渡阶段，留着它可以给你提个醒。

Java 7对异常处理的改进不仅限于这些通用问题，对于特定的资源管理也有所提升，我们马上就会讲到。

#### 1.3.4 try-with-resources (TWR)

这个修改说起来容易，但其实暗藏玄机，最终证明做起来比最初预想的要难。其基本设想是把资源（比如文件或类似的东西）的作用域限定在代码块内，当程序离开这个代码块时，资源会被自动关闭。

这是一项非常重要的改进，因为没人能在手动关闭资源时做到100%正确，甚至不久前Sun提供的操作指南都是错的。在向Coin项目提交这一提案时，提交者宣称JDK中有三分之二的close()用法都有bug，真是不可思议！

好在编译器可以生成这种学究化、公式化且手工编写易犯错的代码，所以Java 7借助了编译器来实现这项改进。

这可以减少我们编写错误代码的几率。相比之下，想想你用Java 6写段代码，要从一个URL(url)中读取字节流，并把读取到的内容写入到文件(out)中，这么做很容易产生错误。代码清单1-3是可行方案之一。

#### 代码清单1-3 Java 6中的资源管理语法

```

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) {
    } finally {
        try {
            out.close();
        }
    }
}

```

处理异常（能  
读或写）

```

        } catch (IOException closeOutx) {
        }
    } catch (FileNotFoundException fnfx) { | 处理异常
} catch (IOException openx) {
} finally {
try {
    if (is != null) is.close();
} catch (IOException closeInx) { | 遇到异常也
}
}

```

看明白了吗？重点是在处理外部资源时，墨菲定律（任何事都可能出错）一定会生效，比如：

- URL中的InputStream无法打开，不能读取或无法正常关闭；
- OutputStream对应的File无法打开，无法写入或不能正常关闭；
- 上面的问题同时出现。

最后一种情况是最让人头疼的——异常的各种组合拳打出来令人难以招架。

新语法能大大减少错误发生的可能性，这正是它受欢迎的主要原因。编译器不会犯开发人员编写代码时易犯的错误。

让我们看看代码清单1-3中的代码用Java 7写出来什么样。和前面一样，url是一个指向下载目标文件的URL对象，file是一个保存下载数据的File对象。

#### 代码清单1-4 Java 7中的资源管理语法

```

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);
    }
}

```

这是资源自动化管理代码块的基本形式——把资源放在try的圆括号内。C#程序员看到这个也许会觉得有点眼熟，是的，它的确很像C#中的从句，带着这种理解使用这个新特性是个不错的起点。在这段代码块中使用的资源在处理完成后会自动关闭。

但使用try-with-resources特性时还是要小心，因为在某些情况下资源可能无法关闭。比如在下面的代码中，如果从文件(someFile.bin)创建ObjectInputStream时出错，FileInputStream可能就无法正确关闭。

```

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

```

假定文件(someFile.bin)存在，但可能不是ObjectInputStream类型的文件，所以文件无法正确打开。因此不能构建ObjectInputStream，所以FileInputStream也没办法关闭。

要确保try-with-resources生效，正确的用法是为各个资源声明独立变量。

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

TWR的另一个好处是改善了错误跟踪的能力，能够更准确地跟踪堆栈中的异常。在Java 7之前，处理资源时抛出的异常信息经常会被覆盖。TWR中可能也会出现这种情况，因此Java 7对跟踪堆栈进行了改进，现在开发人员能看到可能会丢失的异常类型信息。

比如在下面这段代码中，有一个返回InputStream的值为null的方法：

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

在改进后的跟踪堆栈中能看到提示，注意其中被抑制的NullPointerException（简称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与AutoCloseable

目前TWR特性依靠一个新定义的接口实现AutoCloseable。TWR的try从句中出现的资源类都必须实现这个接口。Java 7平台中的大多数资源类都被修改过，已经实现了AutoCloseable（Java 7中还定义了其父接口Closeable），但并不是全部资源相关的类都采用了这项新技术。不过，JDBC 4.1已经具备了这个特性。

然而在你自己的代码里，在需要处理资源时一定要用TWR，从而避免在异常处理时出现bug。

希望你能尽快使用try-with-resources，把那些多余的bug从代码库中赶走。

### 1.3.5 钻石语法

针对创建泛型定义和实例太过繁琐的问题，Java 7做了一项改进，以减少处理泛型时敲键盘的次数。比如你用userid（整型值）标识一些user对象，每个user都对应一个或多个查找表<sup>①</sup>。这用代码应该如何表示呢？

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

这简直太长了，并且几乎一半字符都是重复的。如果能写成

<sup>①</sup>一种为提高处理速度而用查询取代计算的处理机制。一般是将事先计算好的结果存在数组或映射中，然后在需要该结果时直接读取，比如用三角表查某一角度的正弦值。——译者注

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

让编译器推断出右侧的类型信息是不是更好？神奇的Coin项目满足了你这个心愿。在Java 7中，像这样的声明缩写完全合法，还可以向后兼容，所以当你需要处理以前的代码时，可以把过去比较繁琐的声明去掉，使用新的类型推断语法，这样可以省出点儿空间来。

编译器为这个特性采用了新的类型推断形式。它能推断出表达式右侧的正确类型，而不是仅仅替换成定义完整类型的文本。

### 为什么叫“钻石语法”

把它称为“钻石语法”是因为这种类型信息看起来像钻石。原来提案中的名字是“为泛型实例创建而做的类型推断改进”（Improved Type Inference for Generic Instance Creation）。这个名字太长，可编写ITIGIC听上去又很傻，所以干脆就叫钻石语法了。

新的钻石语法肯定会让你少写些代码。我们最后还要探讨Coin项目中的一个特性——使用变参时的警告信息。

### 1.3.6 简化变参方法调用

这是所有修改里最简单的一个，只是去掉了方法签名中同时出现变参和泛型时才会出现的类型警告信息。

换句话说，除非你写代码时习惯使用类型为T的不定数量参数，并且要用它们创建集合，否则你就可以进入下一节了。如果你想要写下面这种代码，那就继续阅读本节：

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

还在？很好。这到底是怎么回事？

变参方法是指参数列表末尾是数量不定但类型相同的参数方法。但你可能还不知道变参方法是如何实现的。基本上，所有出现在末尾的变参都会被放到一个数组中（由编译器自动创建），并作为一个参数传入。

这是个好主意，但是存在一个公认的Java泛型缺陷——不允许创建已知类型的泛型数组。比如下面这段代码，编译就无法通过：

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

不可以创建特定泛型的数组，只能这样写：

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

可这样编译器会给出一个只能忽略的警告。你可以将warnHm的类型定义为HashMap<String, String>数组，但不能创建这个类型的实例，所以你不得不硬着头皮（或至少忘掉警告）硬生生地把原始类型（HashMap数组）的实例塞给warnHm。

这两个特性(编译时生成数组的变参方法和已知泛型数组不能是可实例化类型)碰到一起时,会令人有点头疼。看看下面这段代码:

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

编译器会尝试创建一个包含hm1和hm2的数组,但这种类型的数组应该是被严格禁止使用的。面对这种进退两难的局面,编译器只好违心地创建一个本来不应出现的泛型数组实例,但它又觉得自己不能保持沉默,所以还得嘟囔着警告你这是“未经检查或不安全的操作”。

从类型系统的角度看,这非常合理。但可怜的开发人员本想使用一个十分靠谱的API,一看到这些吓人的警告,却得不到任何解释,不免会内心忐忑。

### 1. Java 7中的警告去了哪里

Java 7的这个新特性改变了警告的对象。构建这些类型毕竟有破坏类型安全的风险,这总得有人知道。但 API 的用户对此是无能为力的,不管doSomething()是不是干了坏事,破坏了类型安全,都不在API用户的控制范围之内。

真正需要看到这个警告信息的是写doSomething()的人,即API的创建者,而不是使用者。所以Java 7把警告信息从使用API的地方挪到了定义API的地方。

过去是在编译使用API的代码时触发警告,而现在是在编译这种可能会破坏类型安全的API时触发。编译器会警告创建这种API的程序员,让他注意类型系统的安全。

为了减轻API开发人员的负担,Java 7还提供了一个新注解java.lang.SafeVarargs。把这个注解应用到API方法或构造方法之中,则会产生类型警告。通过用@SafeVarargs对这种方法进行注解,开发人员就不会在里面进行任何危险的操作,在这种情况下,编译器就不会再发出警告了。

### 2. 类型系统的修改

虽然把警告信息从一个地方挪到另一个地方不是改变游戏规则的语言特性,但也证明了我们之前提到的观点——Coin项目曾奉劝诸位贡献者远离类型系统,因为把这么一个小变化讲清楚要大费周章。这个例子表明搞清楚类型系统不同特性之间如何交互是多么费心费力,而且对语言的修改被实现后又会怎么影响这种交互。这还不是特别复杂的修改,更大的变动所涉及的内容还会更多,其中包括大量微妙的分支。

最后这个例子阐明了由小变化引发的错综复杂的影响。我们对Coin项目中改进的讨论也结束了。尽管它们几乎全都是语法上的小变化,但跟实现它们的代码量相比,它们所带来的正面影响还是很可观的。一旦开始使用,你就会发现这些特性对程序真的很有帮助!

## 1.4 小结

修改语言非常困难。而用类库实现新特性总是相对容易一些,当然并不是所有特性都能用类库实现。面对挑战时,语言设计师可能会做出一些比他们的预想更轻微、更保守的调整。

现在，我们该去看看构成发布版本更重要的东西了，先从Java 7中某些核心类库的变化开始。我们的下一站是I/O类库，那里可以说是发生了天翻地覆的变化。在此之前，希望你已经掌握了Java之前的版本处理I/O的方法，因为Java 7中的这些类（有时候被称为NIO.2）是构建在之前框架基础之上的。

如果你想看到更多关于TWR实战的例子，或者想要了解最新、高性能的I/O类，那就赶快进入下一章吧！

# 新I/O



## 本章内容

- Java 7的新I/O API（即NIO.2）
- Path——基于文件和目录的I/O新基础
- Files应用类及它的各种辅助方法
- 如何实现常见的I/O应用场景
- 介绍异步I/O

本章重点是Java语言中改变较大的I/O API，被称为“再次更新的I/O”或NIO.2（即JSR-203）。NIO.2是一组新的类和方法，主要存在于java.nio包内。下面来看一下它的优点。

- 它完全取代了java.io.File与文件系统的交互。
- 它提供了新的异步处理类，让你无需手动配置线程池和其他底层并发控制，便可在后台线程中执行文件和网络I/O操作。
- 它引入了新的Network-Channel构造方法，简化了套接字（Socket）与通道的编码工作。

先看案例。老板让你写个程序，要扫描生产服务器上的所有目录，找出曾经用各种读/写和所有者权限写入过的所有properties文件。对于Java 6（及更低版本）而言，这几乎是不可能完成的任务，因为：

- 没有直接支持目录树导航的类或方法；
- 没办法检测和处理符号链接；<sup>①</sup>
- 用简单操作读不出文件的属性（比如可读、可写或可执行）。

用Java 7的NIO.2 API可以完成这个不可能的编程任务，它支持目录树的直接导航（Files.walkFileTree（），2.3.1节）、符号链接（Files.isSymbolicLink（），代码清单2-4），能用一行代码读取文件属性（Files.readAttributes（），2.4.3节）。

除此之外，老板还要求你在读取这些properties文件时不能打断主程序的处理流程。可最小的properties文件也有1MB，读取这些文件很可能打断程序的主流程！面对这一要求，在Java 5/6的时代，你很可能会用java.util.concurrent包中的类创建线程池和工作线程队列，再用单独

<sup>①</sup> 符号链接是一种特殊类型的文件，指向文件系统中的另外一个文件或位置——你可以把它理解为快捷方式。

的后台线程读取文件。我们在第4章将会讨论到，现在Java中的并发仍然相当困难，并且非常容易出错。借助Java 7和NIO.2 API，你可以用新的AsynchronousFileChannel（2.5节），不用指定工作线程或队列就可以在后台读取大型文件。咻！

这个新API非常有用，尽管它不能帮你冲咖啡，但它的发展趋势可在那儿摆着呢。

第一个趋势是对其他数据存储方法的探索，特别是在非关系或大数据集领域。你可能很快就会遇到读写大文件（比如微博上的大型报告文件）的问题。NIO.2可以帮助你用一种异步、有效的方式读写大文件，还能利用底层操作系统的特性。

第二个趋势是多核CPU的发展，使得真正并发且更快的I/O成为可能。并发是个难以掌握的领域<sup>①</sup>，但NIO.2会助你一臂之力，它为多线程文件和套接字访问的应用提供了一个简单的抽象层。即便你不直接使用这些特性，它们也会对你的编程生涯产生极大影响，因为IDE、应用服务器和各种流行的框架会大量应用这些特性。

这些只是NIO.2会对你有哪些帮助的例子。如果NIO.2可以解决你眼下面临的一些问题，本章的内容就是为你准备的！否则，你可以在接到Java I/O任务时再回来。

本章你会体验到Java 7新I/O的能力，以便你能够开始编写基于NIO.2的代码，并有信心探索新的API。除此之外，这些API还使用了一些第1章提到的特性，这证明Java 7确实会使用自己的特性。

---

**提示** 将try-with-resources和NIO.2中的新API结合起来可以写出非常安全的I/O程序，这在Java中还是破天荒的第一次！

---

我们觉得你很可能会用到新的文件I/O能力，所以本章会非常详细地介绍。你需要从了解新的文件系统抽象层开始，即先了解Path和它的辅助类。在Path之上，你会接触到常用的文件系统操作，比如复制和移动文件。

我们还会向你介绍异步I/O，给你看一个文件系统的例子。最后我们会讨论套接字和通道功能的融合，以及这对于网络应用开发人员意味着什么。但我们先来看一下NIO.2的由来。

## 2.1 Java I/O 简史

要品出NIO.2 API设计的真正味道，并深刻理解它们的用法，应该先弄清Java I/O的历史。但我们非常理解你想要接触代码的渴望。别着急，你的这种渴望可以在2.2节得到满足。

如果你发现某些API的用法非常简单甚至有点古怪，本节会帮助你从API设计者的角度看待NIO.2。这是Java I/O的第三个主要版本，所以让我们回顾一下Java对I/O支持的发展历史，看看NIO.2是怎么产生的。

Java之所以能够广泛流传，其强大、丰富、简明的类库功不可没，编程时要解决的大多数问

---

<sup>①</sup> 第4章深入探讨了并发计算可能给你的编程生涯带来的微妙复杂性。

题几乎都可以在其中找到支持。但经验丰富的Java开发人员都知道，在老版本的Java中，有些地方不是那么给力。曾经让他们最崩溃的就是Java的输入/输出（I/O）API。

### 2.1.1 Java 1.0 到 1.3

在Java的早期版本（1.0~1.3）中没有完整的I/O支持。也就是说开发人员在开发需要I/O支持的应用时，要面临如下问题。

- 没有数据缓冲区或通道的概念，开发人员要编程处理很多底层细节。
- I/O操作会被阻塞，扩展能力受限。
- 所支持的字符集编码有限，需要进行很多手工编码工作来支持特定类型的硬件。
- 不支持正则表达式，数据处理困难。

基本上，早期版本的Java缺乏对非阻塞I/O的支持。开发人员万般无奈，只好自己实现可伸缩的I/O解决方案。在Java 1.4发布之前，Java一直没能在服务器端开发领域得到重用，我们认为主要原因就是缺乏对非阻塞I/O的支持。

### 2.1.2 在 Java 1.4 中引入的 NIO

为了解决这些问题，Java开始实现对非阻塞I/O的支持，与其他I/O特性一起，帮助开发人员交付更快、更可靠的I/O解决方案。其中有两次主要改进：

- 在Java 1.4中引入非阻塞I/O；
- 在Java 7中对非阻塞I/O进行修改。

2002年发布Java 1.4时，非阻塞I/O（NIO）以JSR-51的身份加入到Java语言中。下面这些特性就是那时增加的。自此之后，Java语言终于得到了服务器端开发人员的青睐：

- 为I/O操作抽象出缓冲区和通道层；
- 字符集的编码和解码能力；
- 提供了能够将文件映射为内存数据的接口；
- 实现非阻塞I/O的能力；
- 基于流行的Perl实现的正则表达式类库。

#### Perl——正则表达式之王

毋庸置疑，Perl编程语言是正则表达式处理之王。实际上，它的设计和实现相当出色，甚至很多编程语言（包括Java）竞相模仿Perl的语法和语义。如果你对Perl语言感兴趣，可以访问<http://www.perl.org/>一探究竟。

NIO无疑使Java向前迈出了一大步，但I/O编程对Java开发人员来说仍然是个挑战。特别是对于文件系统中的文件和目录处理而言，支持力度还是不够。那时的`java.io.File`类有些比较烦人的局限性。

- 在不同的平台中对文件名的处理不一致。<sup>①</sup>
- 没有统一的文件属性模型。(比如读写访问模型)
- 遍历目录困难。
- 不能使用平台/操作系统的特性。<sup>②</sup>
- 不支持文件系统的非阻塞操作。<sup>③</sup>

### 2.1.3 下一代 I/O-NIO.2

为了突破这些局限性，同时也为了支持现代硬件和软件I/O的新范例，由阿兰·波特曼主导的JSR-203应运而生。JSR-203最终变成了我们在Java 7中见到的NIO.2 API。它有三个主要目标，JSR-203规范中的第2.1节对它们进行了详细的介绍 (<http://jcp.org/en/jsr/detail?id=203>)。

- (1) 一个能批量获取文件属性的文件系统接口，去掉和特定文件系统相关的API，还有一个用于引入标准文件系统实现的服务提供者接口。
- (2) 提供一个套接字和文件都能够进行异步（与轮询、非阻塞相对）I/O操作的API。
- (3) 完成JSR-51中定义的套接字——通道功能，包括额外对绑定、选项配置和多播数据报的支持。

接下来让我们从新文件系统的基础Path和它的朋友们开始吧！

## 2.2 文件I/O的基石：Path

在NIO.2的文件I/O中，Path是必须掌握的关键类之一。Path通常代表文件系统中的位置，比如C:\workspace\java7developer (Windows文件系统中的目录)或/usr/bin/zip (\*nix文件系统中zip程序的位置)。如果你能理解如何创建和处理路径，就能浏览任何类型的文件系统，包括zip归档文件系统。

我们通过图2-1 (基于本书中的源码布局)来复习一下文件系统中的几个概念。

- 目录树
- 根路径
- 绝对路径
- 相对路径

之所以要讨论绝对路径和相对路径，是因为我们需要考虑程序运行的位置。比如说，有个程序可能在/java7developer/src/test目录下运行，代码要读取位于/java7developer/src/main目录下的文件名。为了进入/java7developer/src/main目录，可以用相对路径..../main。

但如果程序运行在/java7developer/src/test/java/com/java7developer，用相对路径 ../main则无法

<sup>①</sup>一些批评者会说Java没有兑现“一次编写，处处运行”的承诺。

<sup>②</sup>人们通常希望能够使用Linux/UNIX中的符号链接机制。

<sup>③</sup>Java 1.4的确支持网络套接字的非阻塞操作。

进入目标目录，而会进到并不存在的目录`/java7developer/src/test/java/com/main`中。所以必须使用绝对路径，比如`/java7developer/src/main`。

反之亦然，你的程序可能一直运行在同一位置，比如图2-1中的`target`目录。但目录树的根目录可能会变，比如从`/java7developer`变成了`D:\workspace\j7d`。这时就不能依靠绝对路径，而要用相对路径转到你想到达的位置。

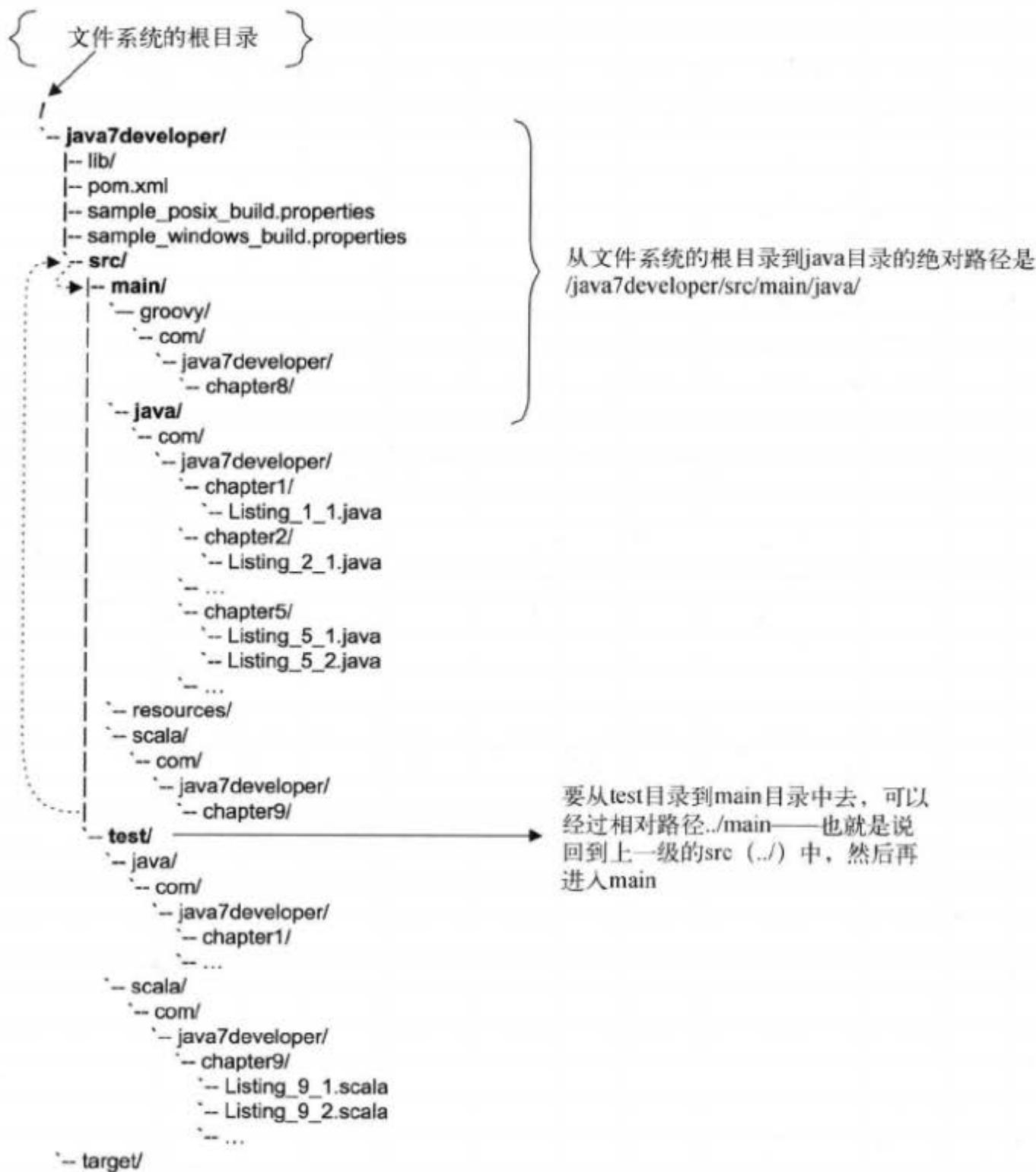


图2-1 说明根目录、绝对路径和相对路径概念的目录树

NIO.2中的Path是一个抽象构造。你所创建和处理的Path可以不马上绑定到对应的物理位置上。尽管看起来奇怪，但有时确实需要如此。比如说，你想创建一个Path来表示即将创建的新

文件。在调用`Files.createFile(Path target)`<sup>①</sup>之前，这个文件是不存在的。如果在Path所对应的文件创建之前，你试图读取这个文件中的内容，就会导致`IOException`。如果你指定了一个并不存在的Path并试图用`Files.readAllBytes(Path)`之类的方法读取，结果是一样的。简言之，JVM只会把Path绑定到运行时的物理位置上。

**警告** 在编写与具体文件系统相关的代码时要小心。创建Path为C:\workspace\java7developer，然后试图读取它的程序，这只能在有C:\workspace\java7developer位置的计算机上才能工作。一定要确保程序逻辑和异常处理考虑到了各种情况，包括可能在不同的文件系统上运行，或文件系统的结构可能被改动。本书的作者之一过去就犯过这个错误，结果把计算机系的一组硬盘全搞坏了！<sup>②</sup>

还是得再重复一遍，NIO.2把位置（由Path表示）的概念和物理文件系统的处理（比如复制一个文件）分得很清楚，物理文件系统的处理通常是由Files辅助类实现的。

有关Path类以及将在本节讨论的其他类的进一步细节请见表2-1。

表2-1 学习文件I/O的关键基础类

类	说 明
Path	Path类中的方法可以用来获取路径信息，访问该路径中的各元素，将路径转换为其他形式，或提取路径中的一部分。有的方法还可以匹配路径字串以及移除路径中的冗余项
Paths	工具类，提供返回一个路径的辅助方法，比如 <code>get(String first, String... more)</code> 和 <code>get(URI uri)</code>
FileSystem	与文件系统交互的类，无论是默认的文件系统，还是通过其统一资源标识（URI）获取的可选文件系统
FileSystems	工具类，提供各种方法，比如其中用于返回默认文件系统的 <code>FileSystems.getDefault()</code>

记住，Path不一定代表真实的文件或目录。你可以随心所欲地操作Path，用Files中的功能来检查文件是否存在，并对它进行处理。

**提示** Path并不仅限于传统的文件系统，它也能表示zip或jar这样的文件系统。

我们来完成几个简单的任务，探索一下Path类：

- 创建一个Path；
- 获取Path的相关信息；
- 移除Path中的冗余项；

① 你一会儿就能在2.4节见到这个方法！

② 我们不会告诉你是谁，不过欢迎你把他查出来！

- 转换一个Path；
  - 合并两个Path，在两个Path中间创建一个Path，并对这两个Path进行比较。
- 我们先从创建表示文件系统中某个位置的Path开始。

### 2.2.1 创建一个 Path

创建Path没什么难的。调用`Paths.get(String first, String... more)`是最快捷的做法，其中第二个变量一般用不到，它仅仅用来把额外的字符串合并起来形成Path字符串。

**提示** 在NIO.2的API中，Path或Paths中的各种方法抛出的受检异常只有`IOException`。我们认为这虽然简单，但有时却会掩藏潜在的问题，而且如果你想处理`IOException`的某个显式子类，则需要额外编写异常处理代码。

我们用`Paths.get(String first)`方法为/usr/bin/目录下的文件压缩工具zip创建一个绝对Path。

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

调用`Paths.get("/usr/bin/zip")`的效果和调用下面这个长一点的代码效果一样：

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

**提示** 创建Path时可以用相对路径。比如，运行在/opt目录下的程序可以用`..`/usr/bin/zip创建一个指向/usr/bin/zip的Path。其含义是指进入/opt的上一层目录（即`/`），然后进入/usr/bin/zip。通过调用`toAbsolutePath()`方法，很容易把这个相对路径转换成绝对路径，如：`listing.toAbsolutePath()`。

你可以从Path中获取信息，比如其父目录、文件名（如果有的话）等。

### 2.2.2 从 Path 中获取信息

Path类中有一组方法可以返回你正在处理的路径的相关信息。代码清单2-1为/usr/bin目录下的zip工具创建一个Path，并输出相关信息，包括它的根目录和父目录。如果你的操作系统中zip也放在/usr/bin目录下，应该能见到下面这种输出信息。

```
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]
```

你在计算机上看到的结果和你所用的操作系统及程序运行的位置有关。

**代码清单2-1 从Path中获取信息**

```

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) + "]");
    }
}

```

在创建了/usr/bin/zip的Path之后，你可以看一下组成Path的元素个数，在此例中就是目录的数量①。相对其父目录和根目录，Path的位置会更有用。也可以通过指定起始和终止的索引来挑出子路径。在本例中是从Path的根（0）到其第二个元素（2）之间的子路径②。

如果这是你初次接触NIO.2文件API，这些方法对你来说非常重要，因为你可以借助它们查看路径处理的结果。

### 2.2.3 移除冗余项

在编写工具软件，比如属性文件分析器时，需要处理的Path中可能会有一个或两个点：

- . 表示当前目录；
- .. 表示父目录。

假设你的程序在/java7developer/src/main/java/com/java7developer/chapter2/目录下运行（见图2-1）。你所在的目录和Listing\_2\_1.java一样，如果传给你的Path是./Listing\_2\_1.java，其中的./部分，即程序正在运行的目录，实际上并没什么用。在这里用短一点儿的Path—Listing\_2\_1.java就够了。

Path中可能还有其他冗余项，比如符号链接（见2.4.3节）。假设在\*nix系统中/usr/logs目录下，你想寻找日志文件log1.txt，但其实/usr/logs只是一个指向/application/logs的符号链接，那里才是存放日志文件的真正位置。要得到这个位置，就需要去掉冗余的符号信息。

所有这些冗余项都会导致Path指向的不是你认为它应该指向的位置。

在Java 7中，有两个辅助方法可以用来弄清Path的真实位置。首先可以用normalize()方法去掉Path中的冗余信息。下面的代码会返回Listing\_2\_1.java的Path，去掉了表明它在当前目录（./部分）中的冗余符号。

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

此外，`toRealPath()`方法也很有效，它融合了`toAbsolutePath()`和`normalize()`两个方法的功能，还能检测并跟随符号连接。

还是回到\*nix系统中的日志文件那个例子，在`/usr/logs`目录下有个日志文件`log1.txt`，而这个目录实际上是指向`/application/logs`的符号链接。通过调用`toRealPath()`，你能得到表示`/application/logs/log1.txt`的真正`Path`。

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

我们要讨论的最后一个`Path` API特性是比较多个`Path`，并找出它们之间的`Path`。

## 2.2.4 转换 Path

转换路径最多的是工具软件。比如你可能需要比较文件之间的关系，以便了解源码目录树的结构是否符合编码规范。或者在shell脚本中执行程序时可能会传入一些`Path`参数，并需要把这些参数变成有意义的`Path`。在NIO.2里可以很容易地合并`Path`，在两个`Path`中再创建`Path`或对`Path`进行比较。

下面的代码将两个`Path`合并，通过调用`resolve`方法，将`uat`和`conf/application.properties`合并成表示`/uat/conf/application.properties`的完整`Path`。

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

要取得两个`Path`之间的路径，可以用`relativize(Path)`方法。下面的代码计算了从日志目录到配置目录之间的路径。

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

如你所愿，你可以用`startsWith(Path prefix)`、`equals(Path path)`等值比较或`endsWith(Path suffix)`来对路径进行比较。

现在使用`Path`类对你来说应该没有问题了，但Java 7之前版本的那些代码怎么办呢？负责NIO.2的团队考虑到了向后兼容性，所以加了两个新的API特性来保证基于`Path`的新I/O和老版本代码之间的互操作性。

## 2.2.5 NIO.2 Path 和 Java 已有的 File 类

新API中的类可以完全替代过去基于`java.io.File`的API。但你不可避免地要和大量遗留下来的I/O代码交互。Java 7提供了两个新方法：

- `java.io.File`类中新增了`toPath()`方法，它可以马上把已有的`File`转化为新的`Path`。
- `Path`类中有个`toFile()`方法，它可以马上把已有的`Path`转化为`File`。

下面的代码演示了这一功能。

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

现在，我们完成了对Path类的探索。接着要研究一下Java 7对目录处理的支持，特别是对目录树。

## 2.3 处理目录和目录树

在读2.2节关于路径的内容时，你可能已经猜到目录不过是带有特别属性的Path。遍历目录的能力是Java 7引入注目的新特性。新加入的java.nio.file.DirectoryStream<T>接口和它的实现类提供了很多功能：

- 循环遍历目录中的子项，比如查找目录中的文件；
- 用glob表达式<sup>①</sup>（比如\*Foobar\*）进行目录子项匹配和基于MIME的内容检测（比如text/xml文件）；
- 用walkFileTree方法实现递归移动、复制和删除操作。

本节主要讨论两个常见用例：在一个目录中查找文件以及在目录树中执行相同任务。我们先从最简单的开始：在一个目录中查找任意文件。

### 2.3.1 在目录中查找文件

我们先讨论一个简单的例子，用模式匹配过滤出java7developer项目中所有的.properties文件。请看下面的代码：

**代码清单2-2 列出目录下的properties文件**

```
Path dir = Paths.get("C:\\workspace\\java7developer");    ① 设定起始路径
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());
}
```

最前面是我们已经熟悉的Paths.get(String)调用①。紧随其后的是关键方法Direct-

<sup>①</sup> glob通常指有限的模式匹配，源自早期Unix中的glob()库函数，该函数用于查找文件系统中指定模式的路径，虽然所用语法和正则表达式类似，但没有正则表达式那么强大的表达力，详见附录B。——译者注

oryStream(Path directory, String patternMatch) ②，它返回一个经过过滤的DirectoryStream，其中包含以.properties结尾的文件。最后输出每个子项③。

过滤流中用到的模式匹配称为glob模式匹配，它和Perl正则表达式类似，但稍有不同。附录B中有如何使用glob模式匹配的详细说明。

代码清单2-2展示了新API处理单个目录的能力，但如果需要递归过滤多个目录时该怎么办？

### 2.3.2 遍历目录树

Java 7支持整个目录树的遍历。也就是说你可以很容易地搜寻目录树中的文件，在子目录中查找，并对它们执行操作。比如你可能想在做开发的机器上弄个工具类来删除目录/opt/workspace/java下的所有.class文件，完成构建前的清除工作。

遍历目录树是Java 7的新特性，要想正确使用，你得掌握一些接口及其实现的细节。其中的关键方法是：

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

提供startingDir非常简单，但给出FileVisitor接口的实现类就比较麻烦了（参数FileVisitor<? superPath> visitor看上去就不是善茬儿），因为最少得实现下面5个方法（T一般就是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)

看起来挺吓人的吧？好在Java 7 API的设计者们已经提供了一个默认实现类，SimpleFileVisitor<T>。

我们要扩展并修改代码清单2-2。下面的代码会列出C:\workspace\java7developer\src目录下及其子目录内的所有.java文件。这段代码展示了Files.walkFileTree方法对默认实现类SimpleFileVisitor的用法，用一个特定的visitFile方法实现来改进它。

#### 代码清单2-3 列出子目录下的所有java源码文件

```
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;
    }
}

```

整个过程从调用`Files.walkFileTree`方法开始①。这里的关键是`FindJavaVisitor`，该类扩展了`FileVisitor`的默认实现类`SimpleFileVisitor`②。你想让`SimpleFileVisitor`来完成大部分工作，比如遍历目录。可实际上你唯一要做的就是重写`visitFile(Path, BasicFileAttributes)`<sup>③</sup>方法③，在这个方法中你也只需要写些代码来判断文件名是否以.java结尾，如果确实是，就在stdout中输出。

其他用例包括递归移动、复制、删除或者修改文件。在大多数应用场景中，你只需要扩展`SimpleFileVisitor`。但如果你想实现自己的`FileVisitor`，API也很灵活。

**注意** 为了确保递归等操作的安全性，`walkFileTree`方法不会自动跟随符号链接。如果你确实需要跟随符号链接，就需要检查那个属性（如2.4.3节所述）并执行相应的操作。

现在你对路径和目录树已经熟悉了，该从处理位置进入真正的文件系统操作上了，接下来我们会向你介绍新的`Files`类及它的朋友们。

## 2.4 NIO.2的文件系统I/O

对于文件系统的操作任务，比如移动文件、修改文件属性，以及处理文件内容等，在NIO.2中都有所改善。对这些操作的支持主要是由`Files`类提供的。

表2-2中有关于`Files`类的详细介绍，此外本节还会介绍另外一个也很重要的类：`WatchService`。

表2-2 文件处理的基础类

类	说 明
<code>Files</code>	让你轻松复制、移动、删除或处理文件的工具类，有你需要的所有方法
<code>WatchService</code>	用来监视文件或目录的核心类，不管它们有没有变化

① 2.4节会介绍`BasicFileAttributes`，所以暂时不用管它。

在本节中，你将学会如何在文件和文件系统上执行下面这些任务：

- 创建和删除文件；
- 移动、复制、重命名和删除文件；
- 文件属性的读写；
- 文件内容的读取和写入；
- 处理符号链接；
- 用WatchService发出文件修改通知；
- 使用SeekableByteChannel——一个可以指定位置及大小的增强型字节通道。

这看起来可能挺恐怖的，但由于设计巧妙，API提供了很多辅助方法，把抽象层隐藏了起来，让你可以轻松快捷地处理文件系统。

**警告** NIO.2 API对原子操作的支持有很大改进，但涉及文件系统处理时，仍然主要依靠代码来提供保护。即使是执行了一半的操作，也很可能会因为突然断网、咖啡泼到服务器上，或某个冒失鬼在错误的UNIX机器上执行了shutdown now命令（本书作者之一亲身经历的著名事件）等诸多原因而出错。尽管API的某些方法还是会偶尔抛出个Runtime-Exception，但某些异常状况可以由Files.exists(Path)这样的辅助方法来缓解。

学习新API最好的办法就是读写代码。接下来我们来看一些实际案例，先从基本的文件创建和删除开始。

### 2.4.1 创建和删除文件

只需要调用Files类里的辅助方法，就可以很容易地创建和删除文件。当然，你接到的任务不可能总像默认情况那么简单，所以我们额外加了一些选项，比如在新创建的文件上设定可读/可写/可执行的安全访问权限。

**提示** 如果你要在自己的机器上运行本节中的代码，请用实际路径替换掉代码中的路径。

下面的代码展示了基本的文件创建操作，用到了Files.createFile(Path target)方法。如果你的操作系统里有个D:\Backup目录，运行代码之后就会在那里创建一个MyStuff.txt文件。

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

通常出于安全考虑，要定义所创建的文件是用于读、写、执行，或三者权限的某种组合时，你要指明该文件的某些FileAttributes。因为这取决于文件系统，所以需要使用与文件系统相关的文件权限类。

下面是一个在POSIX文件系统<sup>①</sup>上为属主、属主组内用户和所有用户设置读/写许可的例子。这种方法允许所有用户对即将创建的文件D:\Backup\MyStuff.txt进行读写操作。

```
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);
```

在java.nio.file.attribute包里有一大串已经写好的\*FilePermission类。对文件属性的支持在2.4.3节中还有更详细的论述。

**警告** 如果在创建文件时要指定访问许可，不要忽略其父目录强加给该文件的umask限制或受限许可。比如说，你会发现即便你为新文件指定了rw-rw-rw许可，但由于目录的掩码，实际上文件最终的访问许可却是rw-r--r--。

删除文件要简单一些，可以用Files.delete(Path)方法。下面的代码删除了刚刚创建的D:\Backup\MyStuff.txt文件。当然，运行这个Java程序的用户需要有删除文件的权限。

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

接下来你将学到如何在文件系统中复制和移动文件。

## 2.4.2 文件的复制和移动

使用Files类中简单的辅助方法可以很轻松地完成文件的复制和移动。

下面的代码演示了如何用Files.copy(Path source, Path target)方法完成基本的复制操作。

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

复制文件时通常需要设置某些选项。下面这个例子用到了覆盖即替换已有文件的选项。

```
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);
```

其他的复制选项包括COPY\_ATTRIBUTES（复制文件属性）和ATOMIC\_MOVE（确保在两边的操作都成功，否则回滚）。

移动和复制很像，都是用原子Files.move(Path source, Path target)方法完成的。通常在移动文件时，你想要用复制选项，此时便可以用Files.move(Path source, Path

<sup>①</sup> 可移植操作系统接口（UNIX），是一种许多操作系统都支持的基本标准。

`target, CopyOptions...)`方法，但要注意变参的使用。

在下面这个例子中，我们要在移动源文件时保留其属性，并且覆盖目标文件(如果存在的话)。

```
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);
```

现在你已经能创建、删除、复制和移动文件了，下面该认真研究一下Java 7对文件属性的支持了。

### 2.4.3 文件的属性

文件属性控制着谁能对文件做什么。一般情况下，做什么许可包括能否读取、写入或执行文件，而由谁许可包括属主、群组或所有人。

本节从讨论文件的基本属性组开始，比如文件最后访问时间以及它是目录还是符号链接等。本节的第二部分讨论对特定文件系统的文件属性的支持，因为不同的文件系统都有它们自己的属性集和属性含义的解释，所以这部分比较难。

让我们先从了解Java 7 对基本文件属性的支持开始吧。

#### 1. 基本文件属性支持

真正通用的文件属性并不多，但确实有一组大多数文件系统都支持的属性。接口BasicFileAttributes定义了这个通用集，但实际上工具类Files就可以回答与文件相关的各种问题，比如下面这些：

- 最后修改时间是什么时候？
- 它有多大？
- 它是符号连接吗？
- 它是目录吗？

代码清单2-4说明了Files类中用于收集这些基本文件属性的方法。代码输出了/usr/bin/zip的相关信息，你看到的输出应该和下面的类似：

```
/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}
```

注意，所有这些属性都是调用`Files.readAttributes(Path path, String attributes, LinkOption... options)`得到的。代码清单2-4如下所示：

**代码清单2-4 通用的文件属性**

```

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() + "]");
}

```

还有一些可以从`Files`类的方法中采集到的通用文件属性信息。这样的信息包括文件属主，是否为符号链接等。请参照`Files`类的Javadoc查看完整的辅助方法列表。

Java 7也支持跨文件系统的文件属性查看和处理功能。

**2. 特定文件属性支持**

在2.4.1节创建文件时你已经见过`FileAttribute`接口和`PosixFilePermissions`类了。为了支持文件系统特定的文件属性，Java 7允许文件系统提供者实现`FileAttributeView`和`BasicFileAttributes`接口。

**警告** 我们之前已经说过了，但有必要再重复一次。在编写特定文件系统的代码时一定要小心。一定要确保你的逻辑和异常处理考虑到了代码在不同文件系统上运行的情况。

来看一个例子，其中你想用Java 7保证正确的访问许可被设置在特定文件中。图2-2显示了Admin用户的home目录的列表。注意那个特殊的`.profile`隐藏文件，它只允许Admin用户写，其他任何人都没有写权限，但所有人都可以读取该文件。

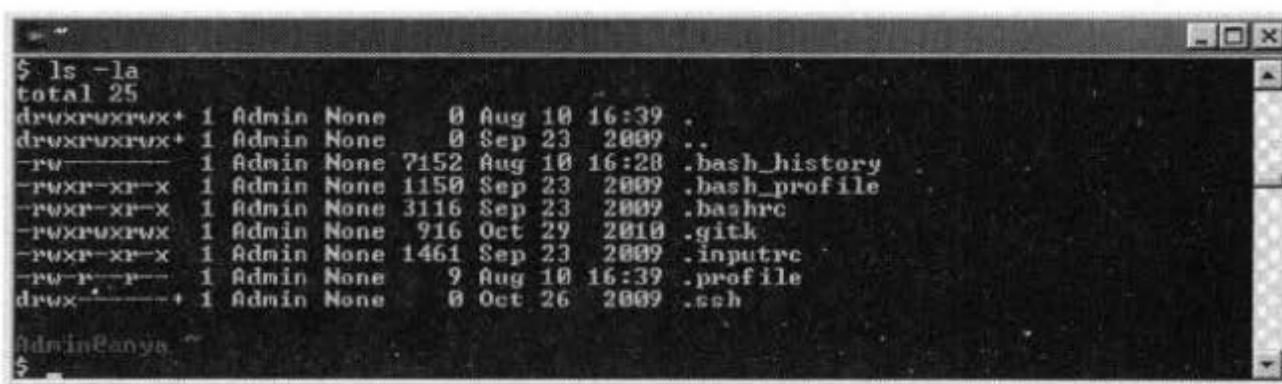


图2-2 Admin用户的home目录列表，显示`.profile`的访问许可

在下面的代码中，你要确保`.profile`文件的访问许可设置正确，与图2-2对应。Admin用户希望其他所有用户都可以读取该文件，但只有他自己来写。你可以用特定的POSIX `PosixFilePermission`和`PosixFileAttributes`类来保证访问许可（`rw-r--r--`）是正确的。

### 代码清单2-5 Java 7对文件属性的支持

```

import static java.nio.file.attribute.PosixFilePermission.*;
try
{
    Path profile = Paths.get("/user/Admin/.profile");
    PosixFileAttributes attrs =
        Files.readAttributes(profile,
            PosixFileAttributes.class);
    ① 获取属性视图
    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());
}

```

2

代码从导入PosixFilePermission常量还有其他未显示的导入开始，然后得到.profile文件的Path。Files类中有个辅助方法让你可以读取特定文件系统的属性，在这个例子中是PosixFileAttributes①。然后你就可以访问PosixFilePermission②。在清除了已有的许可之后③，你可以为文件添加新的访问许可，当然还是用Files中的方法④。

你可能已经注意到了，PosixFilePermission是一个enum，因此没有实现FileAttributeView接口。为什么这里没用PosixFileAttributeView呢？实际上是Files辅助类把它隐藏了起来，这样你就可以用readAttributes方法直接读取文件属性了，也可以用setPosixFilePermissions方法直接设置访问许可。

除了基本属性，Java 7还有一个用来支持特别操作系统特性的扩展系统。可惜，我们不可能囊括所有特殊情况，但我们会给你看一个扩展系统的例子：Java 7对符号链接的支持。

### 3. 符号链接

你可以把符号链接看做指向另一个文件或目录的入口，并且在大多数情况下它们都是透明的。比如切换到符号链接的目录下会把你带到符号链接所指向的目录下。但在写软件时，比如备份工具或部署脚本，你需要慎重考虑是否应该跟随符号链接，NIO.2允许你做出选择。

我们再用一下2.2.3节的例子。你要在\*nix系统上查询/usr/logs目录下的日志文件log1.txt的信息。但/usr/logs目录实际上是一个指向/application/logs目录的符号链接（指针），/application/logs目录才是日志文件的真正位置。

符号链接在宿主操作系统中使用，包括（但不限于）UNIX、Linux、Windows 7和Mac OS X。Java 7对符号链接的支持遵循UNIX操作系统中实现的语义。

下面的代码在读取基本文件属性之前先检查指向安装Java的/opt/platform目录的Path，看它是否为符号链接，我们想读取文件真正位置的属性。代码清单2-6如下所示：

### 代码清单2-6 探索符号链接

```
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());
}
```

Files类提供了一个isSymbolicLink(Path)方法来检查符号链接①。它还有一个辅助方法，可以用于返回符号链接目标的真实Path②，所以你能读到正确的文件属性③。

NIO.2 API默认会跟随符号链接。如果不想跟随，需要用LinkOption.NOFOLLOW\_LINKS选项。这一选项可以用在几个方法调用上。如果你要读取符号链接本身的基本文件属性，应该调用：

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

符号链接是Java 7对特定文件系统支持最常用的例子，API设计者也考虑到了未来对特定文件系统支持特性的扩展，比如量子加密文件系统。

你已经做过文件处理了，现在可以开始研究对文件内容的处理了。

#### 2.4.4 快速读写数据

Java 7可以尽可能多地提供用来读取和写入文件内容的辅助方法。当然，这些新方法使用Path，但它们也可以与那些java.io包里基于流的类进行互操作。因此，你用一个方法就可以读取文件中的所有行或全部字节。

本节会向你介绍打开文件（带选项）的过程，以及一小组常用的文件读/写例子。让我们先从打开文件的不同方式开始。

##### 1. 打开文件

Java 7可以直接用带缓冲区的读取器和写入器或输入输出流（为了和以前的Java I/O代码兼容）打开文件。下面的代码演示了Java 7如何用Files.newBufferedReader方法打开文件并按行读取其中的内容。

```

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

```

打开一个用于写入的文件也很简单。

```

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

```

注意StandardOpenOption.WRITE选项的使用，这是可以添加的几个OpenOption参数之一。它可以确保写入的文件有正确的访问许可。其他常用的文件打开选项还有READ和APPEND。

与InputStream和OutputStream的交互是通过Files.newInputStream(Path, OpenOption...)和Files.newOutputStream(Path, OpenOption...)实现的。它们为过去基于java.io包的I/O和新的基于java.nio包的文件I/O之间架起了一座桥梁。

---

**提示** 在处理String时，不要忘了查看它的字符编码。忘记设置字符编码（通过StandardCharsets类，比如new String(byte[], StandardCharsets.UTF\_8)）可能导致不可预料的字符编码问题。

---

前面的代码片段还是用Java 6及之前版本编写的读取和写入文件代码，仍然属于比较繁琐的底层代码。而Java 7具备更高层的抽象能力，可以帮你避免很多不必要的繁琐编码工作。

## 2. 简化读取和写入

辅助类Files有两个辅助方法，用于读取文件中的全部行和全部字节。也就是说你没必要再用while循环把数据从字节数组读到缓冲区里去。下面的代码演示了如何调用辅助方法。

```

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

```

对于某些软件来说，什么时候读、写是个问题，特别是在处理属性文件或日志时。这时就该文件修改通知系统大显身手了。

### 2.4.5 文件修改通知

在Java 7中可以用java.nio.file.WatchService类监测文件或目录的变化。该类用客户线程监视注册文件或目录的变化，并且在检测到变化时返回一个事件。这种事件通知对于安全监

测、属性文件中的数据刷新等很多用例都很有用。是现在某些应用程序中常用的轮询机制（相对而言性能较差）的理想替代品。

下面的代码用WatchService监测用户karianna主目录的变化，每当发现变化时就会在控制台中输出一个事件通知。和很多持续轮询的设计一样，它也需要一个轻量的退出机制。代码清单2-7如下所示：

### 代码清单2-7 使用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());
    }
}

```

① 监测变化  
② 检查shutdown标志  
③ 得到下一个key及其事件  
④ 检查是否为变化事件  
⑤ 重置监测key

在得到默认的WatchService后，将karianna的主目录登记到变化监测名单中①。然后在一个无限循环（直到shutdown标志改变）②中执行WatcherService的take()方法，直到WatchKey的到来。一旦得到WatchKey，代码就遍历其WatchEvent进行检测③。如果发现了类型为ENTRY\_MODIFY的WatchEvent④，就诏告天下karianna的主目录发生了变化！最后重置key⑤准备迎接下一个事件，继续等待。

还有其他可以监测的事件，比如ENTRY\_CREATE、ENTRY\_DELETE和OVERFLOW（可以表明事件已经丢失或被丢弃了）。

接下来，我们要进入一个非常重要的、抽象的新API——用于数据的读写，使异步I/O成为现实的SeekableByteChannel。

#### 2.4.6 SeekableByteChannel

Java 7引入SeekableByteChannel接口，是为了让开发人员能够改变字节通道的位置和大小。比如，应用服务器为了分析日志中的某个错误码，可以让多个线程去访问连接在一个大型日志文件上的字节通道。

JDK中有一个java.nio.channels.SeekableByteChannel接口的实现类——java.nio.channels.FileChannel。这个类可以在文件读取或写入时保持当前位置。比如说，你可能想要写一段代码读取日志文件中的最后1000个字符，或者向一个文本文件中的特定位置写入一些价格数据。

下面的代码展示了如何运用FileChannel的寻址能力读取日志文件中的最后1000个字符。

```
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);
```

FileChannel类的寻址能力意味着开发人员可以更加灵活地处理文件内容。我们期待能由此产生一些有趣的开源项目，比如针对大型文件的并行访问。随着对该接口的不断扩展，可能还会有网络数据流的续传。

NIO.2 API中下一个主要修改是异步I/O，它可以使用多个后台线程读写文件、套接字和通道中的数据。

## 2.5 异步 I/O 操作

NIO.2另一个新特性是异步能力，这种能力对套接字和文件I/O都适用。异步I/O其实只是一种在读写操作结束前允许进行其他操作的I/O处理。实际上，就是可以充分利用最新的硬件和软件特性，比如多核CPU及操作系统对套接字和文件处理的支持。对于任何想在服务器端和系统级编程领域占有一席之地的编程语言来说，异步I/O都是必不可少的特性。我们相信，Java在服务器端编程语言中所取得的重要地位会因为该特性得以延续。

举个简单的例子，想象一下你要把100GB的数据写入文件系统或网络套接字中。如果你用的是老版本的Java，在同时把数据写入文件或套接字的多个区域时，必须亲自动手用java.util.concurrent写多线程代码。当然，同时进行多路读取也不容易。除非你写的代码十分巧妙，否则在使用I/O时也会阻塞主线程，这意味着在你完成漫长的I/O操作之前，除了等待，还是等待。

**提示** 如果你还没接触过NIO通道，也许可以趁此机会充实下你的知识结构。不过这个领域的  
新内容很少，但在继续本节内容之前，我们建议你去看看Ron Hitchens写的*Java NIO*  
( O'Reilly, 2002 )一书，你会从中获益匪浅。

Java 7中有三个新的异步通道：

- `AsynchronousFileChannel`——用于文件I/O；
- `AsynchronousSocketChannel`——用于套接字I/O，支持超时；
- `AsynchronousServerSocketChannel`——用于套接字接受异步连接。

使用新的异步I/O API时，主要有两种形式，将来式和回调式。有趣的是，这些异步API用到了第4章讨论的一些现代并发技术，所以这真是让你先睹为快了。

我们会从异步文件访问的将来式开始。希望你已经用过这种并发技术，但如果没用过，也不用担心，本节将会讲解得非常详细，即便是刚接触这个话题的新手也能看明白。

### 2.5.1 将来式

NIO.2 API的设计人员用将来式（future）这个术语来表明使用`java.util.concurrent.Future`接口。当你希望由主线程发起I/O操作并轮询等待结果时，一般都会用将来式异步处理。

将来式用现有的`java.util.concurrent`技术声明一个Future，用来保存异步操作的处理结果。这很关键，因为这意味着当前线程不会因为比较慢的I/O操作而停滞。相反，有一个单独的线程发起I/O操作，并在操作完成时返回结果。与此同时，主线程可以继续执行其他需要完成的任务。在其他任务结束后，如果I/O操作还没有完成，主线程会一直等待。图2-3演示了一个用将来式读取大型文件的过程。（代码清单2-8是相应的实现代码。）

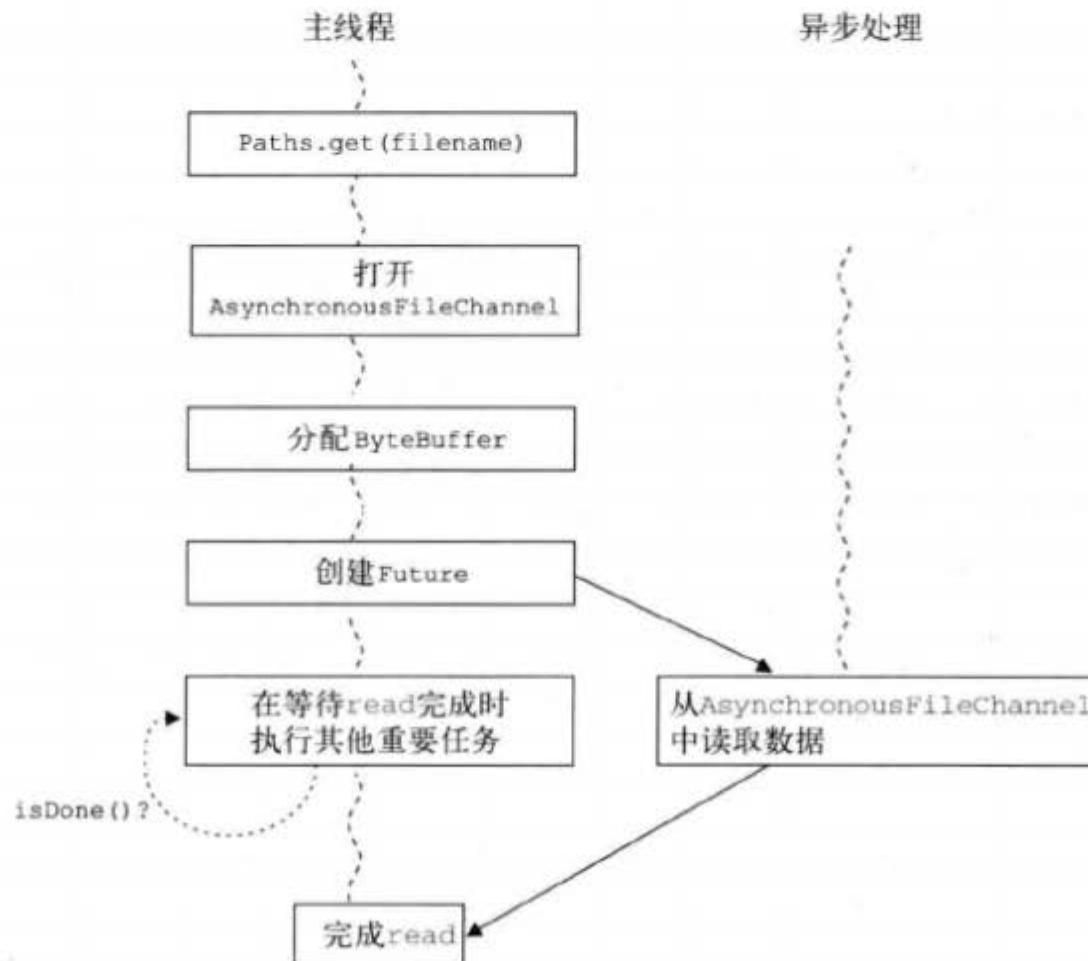


图2-3 将来式异步读取

通常会用Future get()方法（带或不带超时参数）在异步I/O操作完成时获取其结果。假设你要从硬盘上的文件里读取100 000个字节，在旧版的Java中，你需要等待数据读取完成（除非你实现了一个线程池，而且工作线程使用java.util.concurrent技术，这可不是件轻松的事儿）。而在Java 7中，主线程可以在读取数据的同时继续完成其他工作，如下面的代码所示。

### 代码清单2-8 异步I/O——将来式

```

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);           ① 异步打开文件

    while(!result.isDone())
    {
        ProfitCalculator.calculateTax();                         ② 读取100 000字节
    }

    Integer bytesRead = result.get();                          ③ 干点儿别的事情
    System.out.println("Bytes read [" + bytesRead + "]");
}

catch (IOException | ExecutionException | InterruptedException e)
{
    System.out.println(e.getMessage());
}

```

上面的代码一开始先用后台进程中打开一个AsynchronousFileChannel读/写foobar.txt①。接下来的这一步是为了让I/O处理能跟发起它的线程同步进行。因为采用AsynchronousFileChannel，并用Future保存读取结果，所以会自动采用并发的I/O处理②。在读取数据时，主线程可以继续执行任务（比如算一下要交多少税）③。最后，当任务完成时，你可以检查数据读取结果④。

一定要注意，我们在这里用isDone()手工判定result是否结束。通常情况下，result或结束（主线程会继续执行），或等待后台I/O完成。

你可能会好奇这究竟是怎么实现的。长话短说，API/JVM为执行这个任务创建了线程池和通道组。另外，你也可以自己提供和配置一个。解释其中的细节颇费口舌，并且官方文档都解释过了，所以我们只是直接引用了AsynchronousFileChannel的Javadoc：

AsynchronousFileChannel会关联线程池，它的任务是接收I/O处理事件，并分发给负责处理通道中I/O操作结果的结果处理器。跟通道中发起的I/O操作关联的结果处理器确保是由线程池中的某个线程产生的。

如果在创建AsynchronousFileChannel时没有为其指明线程池，那就会为其分配一个系统默认的线程池（可能会和其他通道共享）。默认线程池是由AsynchronousChannelGroup类定义的系统属性进行配置的。

此外还有一种被称为回调的技术。有些开发人员可能会发现回调式用起来更方便，因为它很像Swing、消息和其他Java API中出现过的事件处理技术。

### 2.5.2 回调式

与将来式相反，回调式（callback）所采用的事件处理技术类似于在Swing UI编程时采用的机制。其基本思想是主线程会派一个侦查员CompletionHandler到独立的线程中执行I/O操作。这个侦查员将带着I/O操作的结果返回到主线程中，这个结果会触发它自己的completed或failed方法（你会重写这两个方法）。

在异步事件刚一成功或失败并需要马上采取行动时，一般会用回调式。比如在读取对盈利计算业务处理至关重要的金融数据时，如果读取失败了，你最好马上就执行回滚操作，或进行异常处理。

在异步I/O活动结束后，接口`java.nio.channels.CompletionHandler<V, A>`会被调用，其中V是结果类型，A是提供结果的附着对象。此时必须已经有了该接口的`completed(V, A)`和`failed(V, A)`方法的实现，你的程序才能知道在异步I/O操作成功完成或因某些原因失败时该如何处理。图2-4展示了这一过程（代码清单2-9是该过程的实现代码）。

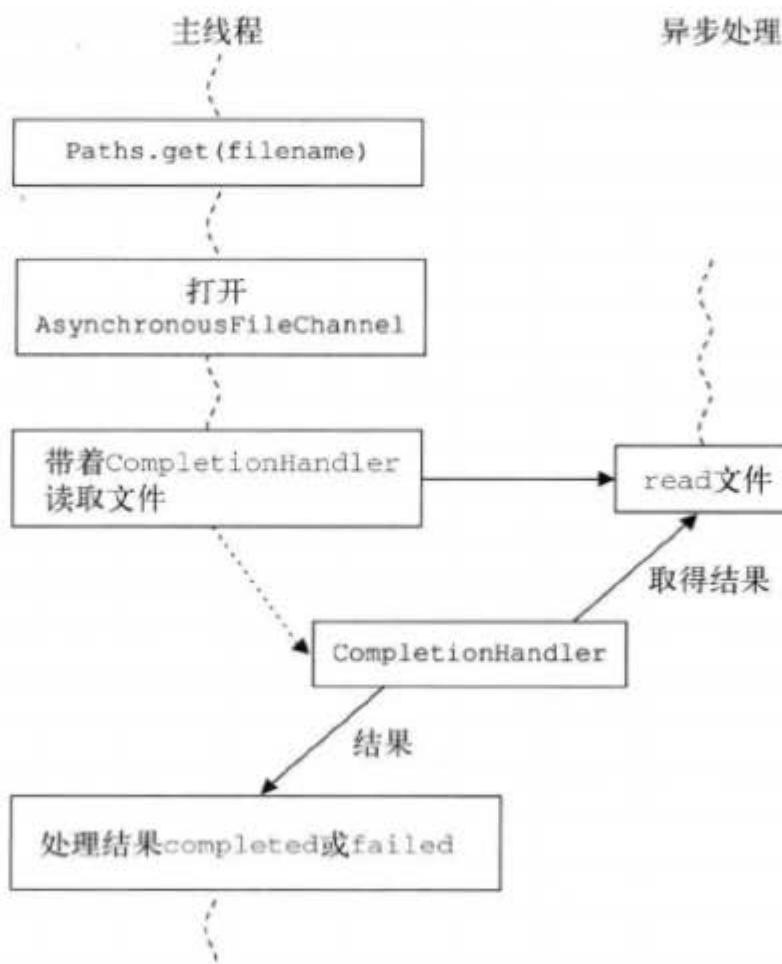


图2-4 回调式异步读取

在下例中，你又一次从`foobar.txt`文件中读取了100 000字节的数据，用`CompletionHandler<Integer, ByteBuffer>`声明是成功或是失败。

**代码清单2-9 异步I/O——回调式**

```

try
{
    Path file = Paths.get("/usr/karianna/foobar.txt");
    AsynchronousFileChannel channel =
        AsynchronousFileChannel.open(file);
    ByteBuffer buffer = ByteBuffer.allocate(100_000);

    channel.read(buffer, 0, buffer,
        new CompletionHandler<Integer, ByteBuffer>()
    {
        public void completed(Integer result,
            ByteBuffer attachment)
        {
            System.out.println("Bytes read [" + result + "]");
        }

        public void failed(Throwable exception, ByteBuffer attachment)
        {
            System.out.println(exception.getMessage());
        }
    });
}
catch (IOException e)
{
    System.out.println(e.getMessage());
}

```

以异步方式打开文件

从通道中读取数据

读取完成时的  
回调方法

本节中的两个例子都是基于文件的，但将来式和回调式异步访问也适用于 `AsynchronousServerSocketChannel` 和 `AsynchronousSocketChannel`。开发人员可以用它们编写程序来处理网络套接字，比如语音IP或写出性能更优异的客户端和服务器端软件。

接下来的一系列变化统一了套接字和通道，让你可以将套接字和通道交互的管理归结到API中。

## 2.6 Socket 和 Channel 的整合

应用软件对网络接入的需求比以往任何时候都要迫切。仿佛一夜之间，家里所有东西都要联网了。在旧版Java中，套接字和通道结合得并不是很好，将它们两个配合在一起是件棘手的事情。因此Java 7推出了`NetworkChannel`，把`Socket`和`Channel`结合到一起，让开发人员可以轻松应对。

编写底层网络代码算是专业领域。如果你的工作领域与此无关，完全可以跳过这一节！但如果恰好你就是干这个的，你可以在本节对Java 7的新特性有一个初步了解。

我们先来看看套接字和通道在Javadoc中的定义，重温一下它们在Java中扮演的角色：

`java.nio.channels`包

定义通道，表示连接到执行I/O操作的实体，比如文件和套接字。定义用于多路传输、非阻塞I/O操作的选择器。

`java.net.Socket`类

该类实现了客户端套接字（也称为“套接字”）。套接字是两个机器间通信的端点。

在旧版Java中，为了执行I/O操作，比如向TCP端口中写入数据，你需要将通道绑定到Socket的实现类上，但Channel和Socket彼此之间却有“代沟”：

- 在旧版Java中，为了配置套接字选项和绑定在套接字上，必须把通道和套接字的API整合在一起；
- 在旧版Java中，不能利用平台特定的套接字行为。

让我们来看看新接口NetworkChannel和其子接口MulticastChannel对这两个领域做的“整理”工作。

### 2.6.1 NetworkChannel

新接口java.nio.channels.NetworkChannel代表一个连接到网络套接字通道的映射。它定义了一组实用的方法，比如查看及设置通道上可用的套接字选项等。下面的代码运用这些方法输出互联网套接字地址在端口3080上所支持的选项，设置IP服务条款选项以及确认套接字通道上的SO\_KEEPALIVE选项。

**代码清单2-10 NetworkChannel选项**

```
SelectorProvider provider = SelectorProvider.provider();
try
{
    NetworkChannel socketChannel =
        provider.openSocketChannel();
    SocketAddress address = new InetSocketAddress(3080);
    socketChannel.bind(address);                                | 将 NetworkChannel
                                                               | 绑定到端口3080上

    Set<SocketOption<?>> socketOptions =
        socketChannel.supportedOptions();                         | 检查套接字选项
    System.out.println(socketOptions.toString());

    socketChannel.setOption(StandardSocketOptions.IP_TOS,
                           3);                                         | 设置套接字的TOS
                                                               | (服务条款) 选项

    Boolean keepAlive =
        socketChannel.getOption(StandardSocketOptions.SO_KEEPALIVE); | <-->
                                                               | 获取 SO_KEEPALIVE
                                                               | 选项
    ...
}
catch (IOException e)
{
    System.out.println(e.getMessage());
}
```

此外，NetworkChannel的出现使得多播操作成为可能。

### 2.6.2 MulticastChannel

像BitTorrent这样的对等网络程序一般都具备多播的功能。在Java的早期版本中，虽然拼凑一下也能实现多播，但却没有很好的API抽象层。Java 7中的新接口MulticastChannel解决了这个问题。

术语多播（或组播）表示一对多的网络通讯，通常用来指代IP多播。其基本前提是将一个包发送到一个组播地址，然后网络对该包进行复制，分发给所有接收端（注册到组播地址中），如图2-5所示。

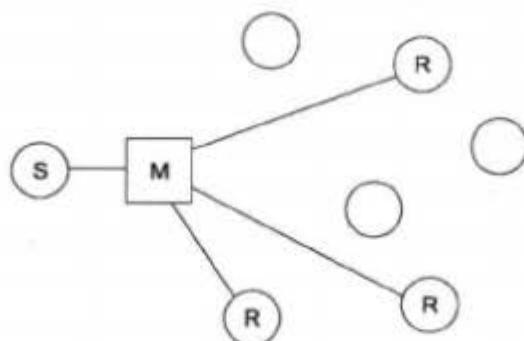


图2-5 多播示例

为了让新来的NetworkChannel加入多播组，Java 7提供了一个新接口java.nio.channels.MulticastChannel及其默认实现类DatagramChannel。也就是说你可以很轻松地对多播组发送和接收数据。

下面的代码说明了如何加入IP地址为180.90.4.12的多播组，并对其发送和接收系统状态信息。

#### 代码清单2-11 NetworkChannel选项

```

try
{
    NetworkInterface networkInterface =
    NetworkInterface.getByName("net1");           | 选择网络接口

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

    dc.setOption(StandardSocketOptions.SO_REUSEADDR,
                true);
    dc.bind(new InetSocketAddress(8080));
    dc.setOption(StandardSocketOptions.IP_MULTICAST_IF,
                networkInterface);           | 将通道设置为多播
    InetAddress group =
        InetAddress.getByName("180.90.4.12");
    MembershipKey key = dc.join(group, networkInterface); | 加入多播组
}
catch (IOException e)
{
    System.out.println(e.getMessage());
}

```

到此为止，我们对NIO.2 API的初步研究已经结束了。希望你喜欢这次行色匆匆的旅程！

## 2.7 小结

硬件和软件I/O的发展突飞猛进，而Java 7也紧随其后，充分利用了NIO.2的新API。Java 7提供的新类库可以用来处理位置（Path），用来在文件系统上执行操作，比如处理文件、目录、符

号链接等。特别值得一提的是，在平台特性的支持下，Java 7可以任意穿梭于文件系统中，并能够处理大型目录结构。

NIO.2致力于为那些通常需要大量编码工作的任务提供一站式的解决办法。尤其是新的File工具类，它有很多辅助方法，比起原来的java.io.File，它使得编写文件I/O代码更快，也更简单。

异步I/O是一个强大的新特性，可以保证在处理大文件时性能不受到显著影响。它对网络套接字和通道流量异常繁忙的程序也很有帮助。

NIO.2也用到了来自Coin项目（第1章）的新特性。这使得在Java 7中处理I/O比以往的版本更安全，而且所需的代码会更少。

现在是时候进入本书的第二部分了。让你的大脑准备好迎接挑战吧！依赖注入（Dependency Injection）、现代并发（Modern Concurrency）和基于Java的软件系统性能调优，这些都等着你去探索，把你喜爱的公爵（Duke）<sup>①</sup>杯加满咖啡，准备好向前冲吧！

---

<sup>①</sup> Duke是Java的吉祥物！<http://kenai.com/projects/duke/pages/Home>。

# Part 2

## 第二部分

## 关键技术

本书的这一部分（第 3 章～第 6 章）全部是对 Java 中的关键编程知识和技术的探索。

本部分的开篇之章是关于依赖注入的，这是一项对代码解耦并增强其可测试性和易读性的通用技术。除了依赖注入的基础知识，我们还介绍了它的演进过程，并探讨了一个最佳实践是如何变成设计模式并形成一个框架的（最终甚至变成了 Java 标准）。

之后，我们会探究出现在硬件领域的多核 CPU 革命。优秀的 Java 开发人员要了解 Java 的并发能力，并知道如何利用它们充分发挥现代处理器的效用。尽管 Java 自 2006 年（Java 5）就大力支持并发编程，但人们对这一领域的理解和应用仍然很少，所以我们会用一整章的内容介绍它。

你将看到 Java 内存模型，以及这个模型中的线程和并发是如何实现的。一旦你掌握了这些理论知识，我们就会指导你用 `java.util.concurrent` 包及其他一些特性为 Java 并发实战打下基础。

接下来，我们会介绍类加载。很多 Java 开发人员不太明白 JVM 如何加载、链接和验证类。所以当某些类的“错误”版本由于某种类加载冲突被执行时，他们会备感沮丧并浪费很多时间。

我们还会谈到 Java 7 的 `MethodHandle`、`MethodType` 和动态调用，让用 `Reflection`（反射）编码的开发人员能以一种更快、更安全的方式完成相同任务。

能够深入到 Java 类文件的内部和它所包含的字节码中是非常强的调试技能。我们会向你展示如何用 `javap` 浏览和理解字节码的含义。

性能调优经常被当做一门艺术，而不是科学。跟踪和解决性能问题经常会占用开发团队大量的时间和精力。在第 6 章，也就是本部分的最后一章，我们会教你评测（而不是猜测），并且告诉你“传说中的调优”是错误的。我们会给你指出一条直指性能问题核心的科学之路。

我们特别关注垃圾回收（GC）和即时（JIT）编译器，这是 JVM 中能够影响性能的两个主要部分。除了其他与性能有关的知识，你还将学到如何阅读 GC 日志，以及如何用免费的 Java VisualVM（jvisualvm）工具分析内存的使用情况。

读完第二部分之后，你就不再是个只想着 IDE 中那些源码的开发人员了。你将知道 Java 和 JVM 的内部工作机制，并能够充分发挥这个星球上最强大的通用 VM（这么说并不为过）的潜力。

### 本章内容

- 控制反转（IoC）和依赖注入（DI）
- 掌握依赖注入技术为什么如此重要
- JSR-330如何统一了Java中的DI
- 常见的JSR-330注解，比如@`Inject`
- Guice 3简介，JSR-330的参考实现（RI）

大约从2004年开始，依赖注入（控制反转的一种形式）就是Java开发主流中一个重要的编程范式<sup>①</sup>。简言之，使用DI技术可以让对象从别处得到依赖项，而不是由它自己来构造。使用DI有很多好处，它能降低代码之间的耦合度，让代码更易于测试、更易读。

本章会先对DI理论以及其给代码带来的好处进行强化。即便你用过IoC/DI框架，本章内容亦能帮你更深入地了解DI的本质。如果你刚刚开始接触DI框架（许多人都是如此），那本章中的内容对你就尤为重要了。

你将会了解Java DI的官方标准JSR-330，并从中了解到Java DI标准注解集的幕后故事。随后，我们会介绍JSR-330的参考实现（RI）Guice 3——一个众所周知的轻量、精巧的DI框架。

我们先来看一些理论知识，好让你明白这个范式大行其道的原因，以及你为什么需要掌握它。

## 3.1 知识注入：理解 IoC 和 DI

为什么需要了解控制反转（IoC、依赖注入（DI）以及它们的基本原理？对于这个问题，仁者见仁智者见智。如果你在知名的问答网站programmers.stackexchange.com上问这个问题，肯定会得到很多不同的答案！

你可能只是刚开始使用不同的DI框架并学习网上的示例，但如果你能够掌握对象关系映射（Object Relational Mapping，ORM）框架，比如Hibernate，你就可以变成编程高手。

<sup>①</sup> 范式（paradigm）在1960年之后是指在科学领域和知识论行文中的思维方式。——译者注

本节首先介绍核心术语IoC和DI背后的一些原理，并探讨使用这一范式的好处。为了让这些概念不至于那么抽象，我们会以HollywoodService为例展示它的转变过程——从自己构造依赖项变成被注入依赖项。

我们先从IoC开始，这个术语经常被（错误地）和DI互换使用。<sup>①</sup>

### 3.1.1 控制反转

在使用非IoC范式编程时，程序逻辑的流程通常是由一个功能中心来控制的。如果设计得好，这个功能中心会调用各个可重用对象中的方法执行特定的功能。

使用IoC，这个“中心控制”的设计原则会被反转过来。调用者的代码处理程序的执行顺序，而程序逻辑则被封装在接受调用的子流程中。

IoC也被称为好莱坞原则，其思想可以归结为会有另一段代码拥有最初的控制线程，并且由它来调用你的代码，而不是由你的代码调用它。

#### 好莱坞原则——“不要给我们打电话，我们会打给你”

好莱坞经纪人总是给人打电话，而不让别人打给他们！如果你曾经跟好莱坞经纪人提议，在明年夏天筹划一个“让Java程序员成为拯救世界的英雄”的大片，你也许会深谙其道。

换一种方式来看IoC，回想一下视频游戏Zork(<http://en.wikipedia.org/wiki/Zork>)用户界面的发展过程，从早期由命令行中的文本控制到如今用图形用户界面控制。

在命令行版本中，用户界面只有一个空白提示符，等着用户输入。当用户输入“向东”或者“Grue，快逃”的指令后，游戏的主应用逻辑会调用恰当的事件处理器来处理这些指令，并返回结果。这里的关键点是应用逻辑要控制调用哪个事件处理器。

而在GUI版本中，IoC开始发挥作用。由GUI框架来控制调用事件处理器，而不是由应用逻辑。当用户点击了一个动作，比如“向东”时，GUI框架会直接调用对应的事件处理器，而应用逻辑可以把重点放在处理动作上。

程序的主控被反转了，将控制权从应用逻辑中转移到GUI框架。<sup>②</sup>

IoC有几种不同的实现，包括工厂模式、服务定位器模式，当然，还有依赖注入。这一术语最初由Martin Fowler在“控制反转容器和依赖注入模式”<sup>③</sup>中提出，然后迅速传遍大街小巷，反响强烈。

<sup>①</sup> 从字面上来看，IoC是指一种机制，使用这种机制的用例很多，实现方式也很多。DI只是其中一种具体用例的具体实现方式。但因为DI非常流行，所以人们经常误以为IoC就是DI，并且认为DI这种叫法比IoC更贴切。这是来自stackoverflow的更全面解释（英文）：<http://stackoverflow.com/questions/6550700/inversion-of-control-vs-dependency-injection>。——译者注

<sup>②</sup> 程序中出现了专门用来实现调用和控制逻辑的GUI框架，应用逻辑中的代码只需关注应用请求的处理。——译者注

<sup>③</sup> 在Martin Fowler的网站<http://martinfowler.com/>中搜索Dependency Injection，你就可以找到这篇文章。

### 3.1.2 依赖注入

依赖注入是IoC的一种特定形态，是指寻找依赖项的过程不在当前执行代码的直接控制之下。虽然你也可以自己编写代码实现依赖注入机制，但大多数开发人员都会使用自带IoC容器的第三方DI框架，比如Guice。

**注意** 可以把IoC容器看做运行时环境。Java中为依赖注入提供的容器有Guice、Spring和PicoContainer。

IoC容器可以提供实用的服务，比如确保一个可重用的依赖项会被配置成单例模式。我们在3.3节介绍Guice时会探讨它的一些服务。

**提示** 把依赖项注入对象的方法有很多种。可以用专门的DI框架，但也可以不这么做！显式地创建对象实例（依赖项）并把它们传入对象中也可以和框架注入做的一样好。<sup>①</sup>

与很多编程范式一样，理解使用DI的原因很重要。我们在表3-1中总结了它的主要好处。

表3-1 DI的好处

好 处	描 述	例 子
松耦合	你的代码不再紧紧地绑定到依赖项的创建上了。如果能与面向接口编程的技术相结合，意味着你的代码再也不用紧紧地绑定到依赖项的具体实现上了	HollywoodService对象不再需要创建它所需的SpreadsheetAgentFinder对象，而是使用从外部传入的对象。如果面向接口编程，HollywoodService可以接受任何类型的AgentFinder传入
易测性	作为松耦合的延伸，还有个特殊的用例值得一提。为了测试，可以把测试替身 <sup>②</sup> 作为依赖项注入到对象中	你可以注入一个总是返回相同价格的虚设票价服务，而不是使用“真实”的价格服务，因为它是外部服务，而且有时无法访问
更强的内聚性	你的代码可以专注于执行自己的任务，不用为了载入和配置依赖项而分心。这样还能增强代码的可读性	你的DAO对象可以专注于查询工作，不用考虑JDBC驱动的细节
可重用组件	作为松耦合的延伸，你的代码应用范围会更加宽广，那些可以提供自己特定实现的用户都可以使用你的代码	一个积极进取的软件开发人员可能会卖给你一个LinkedIn代理人查找器
更轻盈的代码	你的代码不再需要跨层传递依赖项，而是在需要依赖项的地方直接注入	你不再需要把JDBC驱动的细节信息从service类往下传递，而是直接在真正需要它的DAO中直接注入这个驱动实例

把普通代码改写成依赖注入的代码是掌握这些理论的最佳方法，所以我们进入下一节吧。

<sup>①</sup> 感谢Thiago Arrais (<http://stackoverflow.com/users/17801/thiago-arrais>) 提供了这个提示。

<sup>②</sup> 第11章会详细介绍测试替身。

### 3.1.3 转成 DI

本节会重点介绍如何把不用IoC的代码变成使用工厂（或服务定位器）模式的代码，再变成使用DI的代码。在这些转变之后有一个共同的关键技术，即面向接口编程。使用面向接口编程，甚至可以在运行时更换对象。

**注意** 本节的目的是巩固你对DI的理解。因此某些比较套路化的代码被省略掉了。

假设你刚接手了一个小项目，要找出所有对Java开发人员比较友善的好莱坞经纪人。在下面的代码中，AgentFinder接口有两个实现类：SpreadSheetAgentFinder和WebServiceAgentFinder。

**代码清单3-1 接口AgentFinder及其实现类**

```
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(){ ... }
}
```

为了使用经纪人查找器，项目中有个默认的HollywoodService类，它会从Spreadsheet-AgentFinder里得到一个经纪人列表，并根据是否友善对他们进行过滤，最终返回友善的经纪人列表。如下面的代码所示。

**代码清单3-2 HollywoodService——自己创建AgentFinder的具体实现类实例**

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

```

再看一遍上面代码里的HollywoodService，注意到了吗？它被SpreadsheetAgentFinder这个AgentFinder的具体实现死死地黏上了①。

过去这种黏糊糊的实现对Java开发者来说司空见惯，不胜其烦！为了解决这些共性问题，设计模式应运而生。一开始，很多开发人员用工厂模式和服务定位器模式的各种变体来解决这类问题，它们全都是IoC的一种。

### 1. 使用工厂和/或服务定位器模式的HollywoodService

抽象工厂、工厂方法或服务定位器模式中的某个（或它们的某种组合）通常用来解决这种被依赖项黏上的问题。

**注意** 工厂方法和抽象工厂模式在Erich Gamma、Richard Helm、Ralph Johnson和John Vlissides写的《设计模式：可复用面向对象软件的基础》(Addison-Wesley Professional, 1994)中有所论述。服务定位器模式出现在Deepak Alur、John Crupi和Dan Malks编写的《J2EE核心模式》第二版(Prentice Hall, 2003)中。

下面这个版本的HollywoodService类用AgentFinderFactory实现对AgentFinder的动态选择。

#### 代码清单3-3 HollywoodService由工厂负责提供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)
{
    ...
}

```

与代码清单3-2中  
的实现一样

如你所见，现在没有特定的AgentFinder实现类来黏你了。注入agentFinderType①，让AgentFinderFactory根据这一类型挑选令人满意的AgentFinder供你享用②。

3

这和DI相当接近了，但还有两个问题。

- 代码中注入的是一个引用凭据（agentFinderType），而不是真正实现AgentFinder的对象。
- 方法getFriendlyAgents中还有获取其依赖项的代码，达不到只关注自身职能的理想状态。

随着开发人员编写更清晰代码的意愿不断增强，DI技术也越来越流行，正在逐步取代工厂和服务定位器模式。

## 2. 使用DI的HollywoodService

你可能已经猜出接下来重构代码该做什么了！下一步要让AgentFinder直接提供所需的getFriendlyAgents方法。代码如下所示：

**代码清单3-4 HollywoodService——纯手工DI注入AgentFinder**

```

public class HollywoodServiceWithDI
{
    public static List<Agent>
        emailFriendlyAgents(AgentFinder finder) ① 注入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)
    {
        ...
    }
}

```

参见代码清单3-2

看看这个纯手工打造的DI方案，AgentFinder被直接注入到getFriendlyAgents方法中①。现在这个getFriendlyAgents方法干净利落，只专注于纯业务逻辑②。

不过这种手工打造DI的生产方式还是有让人头疼的地方。如何配置AgentFinder具体实现的问题并没有解决，原本AgentFinderFactory要做的工作还是要找个地方完成。

所以，能够真正让我们脱离苦海的只有自带IoC容器的DI框架。打个比方，DI框架就是把你的代码包起来的运行时环境，在你需要时为你注入依赖项。

DI框架的优势在于它可以随时随地为你的代码提供依赖项。因为框架中有IoC容器，在运行时，你的代码需要的所有依赖项都会在那里准备好。

如果HollywoodService类使用标准的JSR-330注解（可以使用任何与JSR-330兼容的框架），那么它会是什么样子？

### 3. 使用JSR-330 DI的HollywoodService

来看看这个例子的最终版本，用框架来执行DI操作。在这里，DI框架用标准的JSR-330@.Inject注解将依赖项直接注入到getFriendlyAgents方法中，代码如下所示：

**代码清单3-5 HollywoodService——用JSR-330 DI注入AgentFinder**

```
public class HollywoodServiceJSR330
{
    @Inject public void emailFriendlyAgents(AgentFinder finder) <① JSR-330注入
    {
        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)
    {
        ...
    }
}
```

参见代码清单3-2

执行查找逻辑

现在AgentFinder的某个具体实现（比如WebServiceAgentFinder）类的实例是由支持JSR-330@Inject注释的DI框架在运行时注入的①。

**提示** 尽管JSR-330注解可以在方法上注入依赖项，但通常只用于构造方法或set方法中。下一节会对这一规范做进一步探讨。

让我们结合代码清单3-5中的HollywoodServiceJSR330类再来重温一下依赖注入对我们的帮助。

- 松耦合——HollywoodService不再依赖于AgentFinder的具体类来完成工作。
- 可测性——为了测试HollywoodService类，你可以注入一个返回固定数量经纪人的基本Java类（比如POJOAgentFinder），这在测试驱动的开发中被称为存根类。这对于单元测试来说非常完美，因为你不再需要Web服务、电子表格或其他第三方实现之类的东西。
- 更强的内聚性——你的代码不用再和工厂打交道，不用四下里去抓依赖项，只需要执行业务逻辑。
- 可重用的组件——假如有个开发人员想用你的API，现在需要注入一个他们定制的AgentFinder实现类，就说JDBCAGentFinder吧，想象一下他轻松惬意的表情吧。

□ 更轻盈的代码——HollywoodServiceJSR330中的代码量与最初的HollywoodService相比明显减少了很多。

使用DI正在逐步成为优秀Java开发人员的标准实践，几个流行的容器都提供了优异的DI能力。然而就在不久之前，DI框架领域还是群雄割据，它们风格迥异，各行其是，使用IoC容器的配置标准都自成体系。即便遵循类似配置风格的框架（比如XML或Java注解），还是存在什么是共通的注解和配置这个问题。

新的DI标准化方式（JSR-330）就是要解决这个问题。它对大多数Java DI框架的核心能力做了很好的汇总。因为有DI框架（比如Guice）的内部工作机制作为其坚实的基础，所以接下来我们会深入探讨这一标准化方式。

## 3.2 Java 中标准化的 DI

从2004年开始，有几个用于依赖注入的IoC容器得到了广泛的应用（仅以Spring、Guice和PicoContainer为例）。曾几何时，它们在DI的配置方式上仍然各自为政，这使开发人员很难在不同框架之间迁移。

这一问题直到2009年5月才出现转机，DI社区的两大巨头Bob Lee（Guice）和Rod Johnson（SpringSource）宣布要齐心协力，共同打造一组标准的接口注解<sup>①</sup>。他们紧接着提出了JSR-330（javax.inject）规范请求，倡导Java SE的DI标准化。DI社区的各路诸侯纷纷响应，全部表示支持。

### Java EE中的DI标准化情况如何？

Java企业应用从JEE 6开始构建了自己的依赖注入体系（即CDI），由JSR-299（Java EE平台中的上下文及依赖注入）规范确定，你可在<http://jcp.org/>中搜索JSR-299了解其详细信息。简言之，JSR-299构建在JSR-330基础之上，旨在为企业应用提供标准化的配置。<sup>②</sup>

自从javax.inject出现在Java中（Java SE 5、6和7都支持）以来，代码中就可以使用标准的依赖注入了，也可以在不同的DI框架中进行迁移。比如，你原来在轻量级的Guice框架中运行的代码，为了利用其丰富的特性，也可以迁移到Spring框架中去。

**警告** 实际上，代码迁移并不容易。一旦你的代码用到了仅由特定ID框架支持的特性，就不太可能摆脱这一框架了。尽管javax.inject包提供了常用DI功能的子集，但是你可能需要使用更高级的DI特性。正如你想象的那样，对于哪些特性应该作为通用的标准也是众说纷纭，很难统一。虽然现状不尽如人意，但Java毕竟朝DI框架的标准化方向迈出了一步。

① Bob Lee, “Announcing @javax.inject.Inject” (2009-05-08), [www.theserverside.com/news/thread.tss?thread\\_id=54499](http://www.theserverside.com/news/thread.tss?thread_id=54499)。

② JSR-299 (Java Contexts and Dependency Injection) 目前由Redhat的Gavin King (Hibernate的创建者) 主导，因为它比较新，所以设计理念上解决了以前DI框架中的一些问题，而且也不是非得在Java EE容器上才能使用，在Servlet容器上也可以使用。其参考实现为weld，详情请参见官网：<http://www.seamframework.org/Weld>。——译者注

为了理解最新的DI框架如何应用新标准，我们需要对javax.inject包进行一番研究。记住，javax.inject包只是提供了一个接口和几个注解类型，这些都会被遵循JSR-330标准的各种DI框架实现。也就是说，除非你在创建与JSR-330兼容的IoC容器（如果如此，向你致敬），通常不用自己实现它们。

### 我为什么要知道这东西怎么工作？

优秀的Java开发人员不能满足于只作为类库和框架的使用者，还要明白其内部的基本工作原理。在DI领域，不理解其原理可能会面临各种难缠的问题，比如依赖项配置错误、依赖项诡异地超出作用域、依赖项在不该共享时被共享以及分步调试离奇宕机等。

javax.inject的文档对这个包的目的做出了精彩的解释，所以我们全盘照搬过来了：

#### javax.inject包<sup>①</sup>

这个包指明了获取对象的一种方式，与传统的构造方法、工厂模式和服务定位器模式（比如JNDI）等相比，这种方式的可重用性、可测试性和可维护性都得到了极大提升。这种方式称为依赖注入，对于大多数非小型应用程序都很有帮助。

javax.inject包里包括一个Provider<T>接口和5个注解类型（@Inject、@Qualifier、@Named、@Scope和@Singleton），后面几节会逐一对它们进行介绍。先从@Inject开始。

### 3.2.1 @Inject 注解

@Inject注解可以出现在三种类成员之前，表示该成员需要注入依赖项。按运行时的处理顺序，这三种成员类型是：

- (1) 构造器
- (2) 方法
- (3) 属性

在构造器上使用@Inject时，其参数在运行时由配置好的IoC容器提供。比如在下面的代码中，运行时调用MurmurMessage类的构造器时，IoC容器会注入其参数Header和Content对象。

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

规范中规定向构造器注入的参数数量为0或多个，所以在不含参数的构造器上使用@Inject也是合法的。

① <http://atinject.googlecode.com/svn/trunk/javadoc/javax/inject/package-summary.html>。

**警告** 因为JRE无法决定构造器注入的优先级，所以规范中规定类中只能有一个构造器带@Inject注解。

也可以用@Inject注解方法，与构造器一样，运行时可注入参数的数量也可以是0或多个。但使用参数注入的方法不能声明为抽象方法，也不能声明其自身的类型参数<sup>①</sup>。下面这段代码在set方法前使用@Inject，这是注入可选属性的常用技术。

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

向方法中注入参数的技术对于服务类方法来说非常有用，其所需的资源可以作为参数注入，比如向查询数据的服务方法中注入数据访问对象（DAO）。

**提示** 向构造器中注入的通常是类中必需的依赖项，而对于非必需的依赖项，通常是在set方法上注入。比如已经给出了默认值的属性就是非必需的依赖项。这一最佳实践已经成了惯例。

也可以直接在属性上注入（只要它们不是final），虽然这样做简单直接，但你最好不要用。因为这样可能会让单元测试更加困难。直接注入的语法也非常简单。

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

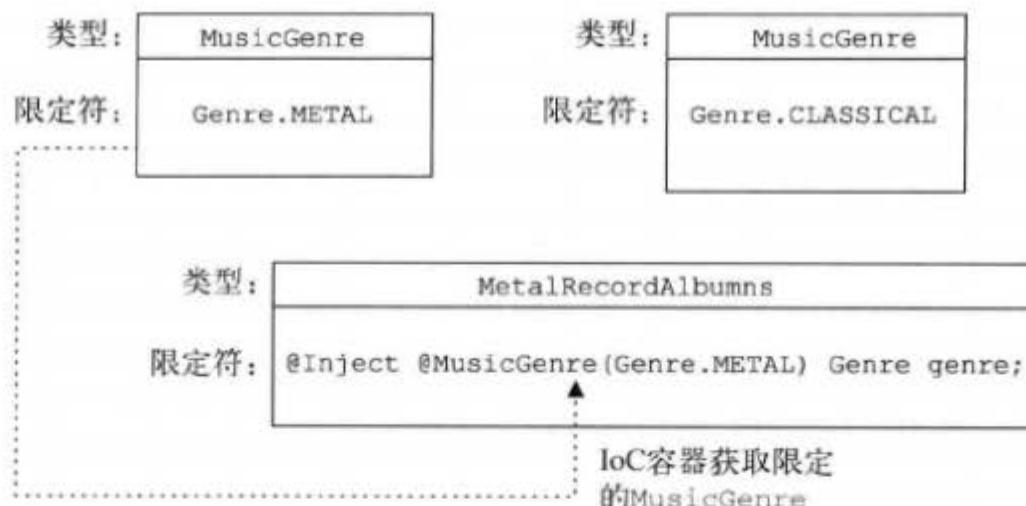
你可以在Javadoc中了解更多关于@Inject注解的详细内容，可以在其中找到哪些类型的值可以注入以及如何处理依赖循环。

对于@Inject，现在你应该不再感到陌生了。接下来就该了解如何限定（进一步标识）那些注入到你的代码中的对象了。

### 3.2.2 @Qualifier 注解

支持JSR-330规范的框架要用注解@Qualifier限定（标识）要注入的对象。比如在IoC容器中有两个类型相同的对象，当把它们注入到你的代码中时，肯定要把它们区别开。图3-1解释了这一概念。

<sup>①</sup> 即不能使用Oracle网站上的Java教程(<http://download.oracle.com/javase/tutorial/extras/generics/methods.html>)中所讲的泛型方法技巧。

图3-1 用`@Qualifier`注解区分同一类型`MusicGenre`的不同bean

如果你用过由框架实现的限定符，应该知道要创建一个`@Qualifier`实现必须遵循如下规则。

- 必须标记为`@Qualifier`和`@Retention(RUNTIME)`，以确保该限定注解在运行时一直有效。
- 通常还应该加上`@Documented`注解，这样该实现就能加到API的公共Javadoc中了。
- 可以有属性。
- `@Target`注解可以限定其使用范围；比如将其使用范围限制为属性，而不是限定为属性的默认值和方法中的参数。

为了让你对上面的规则有直观的感受，下面给出一个`@Qualifier`实现。某音乐库框架中的IoC容器提供了一个限定符`@MusicGenre`，开发人员在创建`MetalRecordAlbumns`类时可以使用该限定符，以确保注入了正确的`Genre`。

```

@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;
}
  
```

Java开发人员一般不需要自己创建`@Qualifier`注解，但要对各种IoC容器如何实现限定有个基本的了解。

JSR-330规范中要求所有IoC容器都要提供一个默认的`@Qualifier`注解：`@Named`。

### 3.2.3 @Named 注解

@Named是一个特别的@Qualifier注解，借助@Named可以用名字标明要注入的对象。将@Named和@Inject一起使用，符合指定名称并且类型正确的对象会被注入。

在下面这个例子中，注入了名称为“murmur”和“broadcast”的两个MurmurMessage对象。

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

尽管还有其他比较常用的限定注解，但最终只有@Named被选作JSR-330的标准限定注解，所有DI框架都要实现。

发起规范的各方支持者还在另外一个问题上达成了一致——同意用标准化接口来确定注入对象的生命周期。

### 3.2.4 @Scope 注解

@Scope注解用于定义注入器（即IoC容器）对注入对象的重用方式。JSR-330规范中明确了如下几种默认行为。

- 如果没有声明任何@Scope注解接口的实现，注入器应该创建注入对象并且仅使用该对象一次。
- 如果声明了@Scope注解接口的实现，那么注入对象的生命周期由所声明的@Scope注解实现决定。
- 如果注入对象在@Scope实现中要由多个线程使用，则需要保证注入对象的线程安全性。  
关于线程及线程安全的更多细节，请参阅第4章。
- 如果某个类上声明了多个@Scope注解，或声明了不受支持的@Scope注解，IoC容器应该抛出异常。

DI框架管理注入对象的生命周期时不会超出这些默认行为划定的界限。有些IoC容器支持自己特有的@Scope，尤其是在Web前端领域，至少在JSR-299被广泛应用之前是这样。因为大家公认的通用@Scope实现只有@singleton一个，所以JSR-330规范中仅确定了它这么一个标准的生命周期注解。

### 3.2.5 @Singleton 注解

@Singleton注解接口在DI框架中应用广泛。在需要注入一个不会改变的对象时，就要用@Singleton。

### 单例模式

单例设计模式是为了确保类仅被实例化一次，详情参见由Erich Gamma, Richard Helm, Ralph Johnson, 和 John Vlissides合著的《设计模式：可复用面向对象软件的基础》( Addison-Wesley Professional, 1994 ) 第127页。请谨慎使用单例模式，因为它有时候会变成反模式。

大多数DI框架都将`@Singleton`作为注入对象的默认生命周期，无需显式声明。也就是说如果没有明确指定注入对象的生命周期，框架就会认为你想注入一个单例对象。如果你想显式声明它为单例对象，可以用下面这种方式：

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

在这个例子中，我们假定`defaultHeader`从来不会改变（切实有效的静态数值），所以它可以作为单例对象注入。

最后，我们来讨论一下当你觉得标准注解无法满足你的需求时该怎么办。

### 3.2.6 接口 `Provider<T>`

如果你想对由DI框架注入代码中的对象拥有更多的控制权，可以要求DI框架将`Provider<T>`接口实现注入对象<sup>①</sup>(`T`)。控制对象的好处在于：

- 可以获取该对象的多个实例。
- 可以延迟获取该对象（延迟加载）。
- 可以打破循环依赖。
- 可以定义作用域，能在比整个被加载的应用小的作用域中查找对象。

该接口仅有一个`T get()`方法，这个方法会返回一个构造好的注入对象(`T`)。例如，在`MurmurMessage`需要依赖项`Message`对象时，可以向其构造方法中注入对应的`Provider<T>`接口实现的实例 (`Provider<Message>`)。根据限定条件的不同，得到的`Message`对象也会不同。请看下面的代码。

#### 代码清单3-6 使用接口 `Provider<T>`

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

class MurmurMessage
{
    @Inject MurmurMessage (Provider<Message> messageProvider)
    {
        Message msal = messageProvider.get();
```

得到一个`Message`

<sup>①</sup> 原文如此，应为该类，后面还有几处笔误，请注意。——译者注

```

if (someGlobalCondition)
{
    Message copyOfMsg1 = messageProvider.get();           ←
}
...
}

```

① 得到Message的  
复本

注意上面的代码中是如何从Provider<Message>中获取第二个Message对象的。如果直接注入Message，就无法获取另外一个Message实例。在这个例子中，第二个注入对象仅在需要时才会加载①。

我们已经对新javax.inject包背后的理论做了介绍，还给出了一些例子进行讲解。现在就该挽起袖子，用成熟的DI框架Guice实际操练一把了。

### 3.3 Java 中的 DI 参考实现：Guice 3

Guice（读作“Juice”）是由Bob Lee在大约2006年发起的开源项目，项目站点地址为<http://code.google.com/p/google-guice/>。你可以在网站上看到该项目的设计初衷、相关文档并下载运行本节示例代码所需的二进制JAR文件。

在Guice这样的DI框架里，你可以配置依赖项、绑定依赖项，并在使用@Inject注解（和它在JSR-330中的朋友们）注入依赖项时声明它们的作用域。

Guice 3是JSR-330规范的完整参考实现，本节所有内容都是基于Guice 3的。虽然Guice不仅仅是一个简单的DI框架，但我们关注的主要还是它的DI能力和示例代码，其中你可以使用JSR-330标准注解和Guice一起编写DI代码。

#### 3.3.1 Guice 新手指南

现在你已经了解JSR-330的各种注解了。可以借助Guice构建一个Java注入对象集合（包括它们的依赖项）。用Guice的话说，为了让注入器创建对象关系图，需要创建声明各种绑定关系的模块，其中绑定是用来明确要注入的具体实现类的。晕了没？不用担心，看到代码你就明白了，这些概念实际上非常简单。

**提示** 对象关系图、绑定、模块和注入器都是Guice里的常用语，如果你想用好Guice，最好尽快搞清楚它们是什么意思。

本节我们还是用HollywoodService的例子。一开始先创建一个拥有各种绑定关系的配置类（模块）。实际上这是Guice框架要管理的那些依赖项的外部配置。

**在哪里下载Guice?**

最新版的Guice 3可以从<http://code.google.com/p/google-guice/downloads/list>上下载，对应的文档可以在<http://code.google.com/p/google-guice/wiki/Motivation?tm=6>上找到。

要得到完整的IoC容器和DI支持，还需要下载Guice zip文件并将它解压到你选定的位置。为了在Java代码中使用Guice，要确保这些JAR文件包含在CLASSPATH中。

对于本书中后续代码示例而言，构建Maven时也会自动下载Guice 3的JAR文件。

我们先来创建一个定义绑定关系的AgentFinderModule。这个AgentFinderModule类扩展了AbstractModule，绑定关系在重写的configure()方法中声明。在本例中，当客户类HollywoodService要求@Inject一个AgentFinder的时候，就会绑定WebServiceAgentFinder类作为注入对象。我们在这里遵循构造方法注入的惯例，具体实现请见代码：

**代码清单3-7 HollywoodService——用Guice注入AgentFinder**

```
import com.google.inject.AbstractModule;
public class AgentFinderModule extends AbstractModule { ← 扩展AbstractModule
    @Override
    protected void configure() { ← 重写configure()方法
        bind(AgentFinder.class). to(WebServiceAgentFinder.class); ← ① 绑定要注入的
    }
}
public class HollywoodServiceGuice { ← 实现类
    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)
    {
        ...
    }
}
```

同代码清单3-2

绑定关系的确立在调用Guice的bind方法时发生，把要绑定的类（AgentFinder）传给它，然后调用to方法指明要注入到哪个实现类①。

现在已经在模块中声明了绑定关系，可以让注入器构建对象关系图了。接下来我们要看看在独立Java程序和Web应用程序这两种情况下分别要如何实现。

### 1. 构建Guice对象关系图——独立Java程序

在标准的Java程序中，可以通过public static void main(String[] args)方法构建对象关系图。代码清单3-8如下所示。

**代码清单3-8 HollywoodServiceClient——用Guice构建对象关系图**

```
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();
        ...
    }
}
```

对于Web应用，情况稍有不同。

### 2. 构建Guice对象关系图——Web应用程序

在Web应用程序中，需要把guice-servlet.jar加到Web应用的类库中，然后在web.xml中添加下面的配置项：

```
<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>
```

然后是标准动作，扩展ServletContextListener以便使用Guice的ServletModule（与代码清单3-7中的AbstractModule类似）。

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

最后一步，把下面这些配置加到web.xml文件中，以便servlet容器在部署应用时触发该类。

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

经由注入器创建HollywoodServiceGuice，你得到了一个配置完备的类，马上就可以调用其中的getFriendlyAgents方法。

非常简单，对不对？没错，但这种把WebServiceAgentFinder绑定到AgentFinder上的情况很简单，而你所需要的绑定方式可能要比这个复杂，所以我们还需要了解一下如何定义更复杂的绑定方式。

### 3.3.2 水手绳结：Guice 的各种绑定

Guice提供了多种绑定方式，官方文档列出的绑定类型如下所示：

- 链接绑定
- 绑定注解
- 实例绑定
- @Provides方法
- Provider绑定
- 无目标绑定
- 内置绑定
- 即时绑定

我们无意在此重复Guice的官方文档，因此只挑最常用的讲解一下，其中包括链接绑定、绑定注解和@Provides方法及Provider<T>绑定。

#### 1. 链接绑定

链接绑定是最简单的绑定方式，代码清单3-6中配置AgentFinderModule时用的就是这种方式。这种绑定方式只是告诉注入器运行时应该注入实现类或扩展类（是的，可以直接注入子类）。

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

你已经见过这种绑定代码了，让我们看看另外一种最常用的绑定方式：绑定注解。

#### 2. 绑定注解

绑定注解是指将注入类的类型和额外的标识符组合起来，以标识恰当的注入对象。你可以自定义绑定注解（参见Guice在线文档），不过我们还是先介绍一下JSR-330标准注解@Named的用法。

在下面的例子中，你所熟悉的@Inject依然会出现，但它这次会与@Named注解联袂登场，以注入特定名称的AgentFinder。为了配置@Named绑定，需要在AgentModule中调用annotatedWith方法，代码清单3-9如下所示：

**代码清单3-9 HollywoodService——使用@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); ←
    }
}

```

使用@Named  
注解

与命名参数  
绑定在一起

现在你已经知道如何配置命名依赖项了，可以继续学习另一种绑定方式了。下面就用@Provides注解和Provider<T>接口绑定完全由你自己定制的依赖项吧。

**3. @Provides和Provider：提供完全定制的对象**

你可以用@Provides注解，或者在configure()方法中绑定，以返回一个完全由你自己定制的对象。比如说，你可能想注入一个非常特别的SpreadsheetAgentFinder（微软的Excel电子表格实现）。

注入器会查看所有标记了@Provides注解方法的返回类型，以决定要注入哪个对象。比如在下面的代码中，HollywoodService会用由provideAgentFinder()方法提供的并带有@Provides注解的AgentFinder。

**代码清单3-10 AgentFinderModule——使用@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; ←
    }
}

```

返回注入器  
需要的类型

创建SpreadsheetAgentFinder  
的实例并设定具体值

`@Provides`方法会变得越来越多，为了不把模块类撑爆，你可能要把它们拆分出去建立自己的类。因此，Guice支持JSR-330的`Provider<T>`接口；如果你还记得第3.2.6节的内容，应该不会忘记`T get()`方法。当在`AgentFinderModule`类中通过`toProvider`方法绑定到`AgentFinderProvider`时，就会调用这个方法。代码如下所示：

#### 代码清单3-11 AgentFinderModule——使用Provider接口

```
public class AgentFinderProvider implements Provider<AgentFinder>
{
    @Override
    public AgentFinder get()                                ◀ 使用 T get() 方法
    {
        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);           ◀ 绑定 provider
    }
}
```

这是最后一个关于绑定的例子。现在你应该能用Guice绑定你需要的依赖项了。但我们还没讨论依赖项的生命周期范围。了解生命周期非常重要，因为如果对象生命周期设置错误的话，它们可能会存在更长时间并占用更多的内存空间。

### 3.3.3 在 Guice 中限定注入对象的生命周期

Guice为注入对象提供了不同级别的生命周期。其中最短的是`@RequestScope`，然后是`@SessionScope`，还有就是JSR-330规范中定义的`@Singleton`，也就是应用级别的生命周期。

在代码中有以下几种方式应用依赖项的生命周期：

- 在要注入的类中；
- 作为绑定声明的一部分（比如`bind().to().in()`）；
- 和`@Provides`一起使用注解声明。

上面的列表有点儿抽象，我们还是用限定依赖项生命周期的一小段代码来说明上面这些方式究竟是什么意思吧。

#### 1. 限定注入项的生命周期

假设你希望程序的整个生命周期中只用一个`SpreadsheetAgentFinder`实例。为此需在类声明中设置`@Singleton`，如下所示：

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

用这个方法还有个好处，可以提醒开发人员注入项对线程安全的要求。因为从理论上来说，`SpreadsheetAgentFinder`类可以多次注入，`@Singleton`范围意味着要保证这个类的线程安全性（第4章会讨论线程安全）。

如果你更喜欢在绑定依赖项时声明其生命周期，你就可以这样做。

## 2. 用**bind()**方法设置生命周期

有些开发人员可能喜欢把所有和注入对象相关的规则都放在一起。还记得代码清单3-9中如何绑定“primary” `AgentFinder`吗？在绑定时设置生命周期与此类似，只要在`bind()`方法串后面再加上`.in(<Scope>.class)`就行了。

下面的代码改进了代码清单3-9，在`bind()`方法串后面加上了`in(Session.class)`，使得在会话（session）范围中能够获取“primary” `AgentFinder`对象。

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

还有最后一种设置注入对象生命周期的方法：与`@Provides`注解联合设置。

## 3. 设置**@Provides**对象的生命周期

你可以在`@Provides`注解旁边加上一个生命周期，以定义由该方法所提供对象的生命周期。比如在代码清单3-9中，可以加上`@Request`注解，将最终提供的`SpreadsheetAgentFinder`实例限定在请求（request）范围中。

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

Guice也提供了针对Web应用的注入项生命周期（比如Servlet request范围），当然你也可以根据需要实现自己的注入项生命周期。

现在你已经基本掌握了在Guice中用JSR-330注解满足你的依赖注入需求了。其实Guice还提供许多非JSR-330特性，比如它还支持面向方面的编程（AOP），让你可以实现安全性和日志处理

的横切关注点。要了解这些内容，请参考Guice的在线文档和代码示例。

#### 谨慎选择对象的生命周期！

一个成熟的Java开发人员总是会认真考虑对象的生命周期。创建无状态对象相对来说成本低廉，无需考虑其生命周期。JVM会在需要时创建和销毁它们，毫无障碍。（第6章详细讨论了JVM和性能。）

另一方面，状态对象总是需要设定生命周期！你应该认真考虑是否要让一个对象的生命周期贯穿整个应用程序的运行期，或者仅存在于当前会话，还是只在当前请求中。接下来就要考虑对象的线程安全性了（第4章会进一步讨论这个问题）。

## 3.4 小结

IoC是个复杂的概念。但通过对工厂和服务定位器模式的探讨，你能了解基本IoC实现是如何工作的。工厂模式有助于你理解DI以及DI给代码带来的好处。即便DI范式接受起来非常困难，它也值得你继续坚持，因为它能让你编写松散耦合的代码，让代码更易测试和更易读。

JSR-330不仅仅是统一DI通用功能的重要标准，它还提供了你需要了解的幕后规则及限制。通过研究标准DI注解集，你会更加欣赏不同DI框架对规范的实现，因而可以更有效地使用它们。

Guice是JSR-330的参考实现，同时它也是一个流行的、轻量级的DI框架。实际上，对于很多应用程序来说，使用Guice和与JSR-330兼容的注解集可能就足以满足你对DI的需求了。

如果你是从头开始看这本书的，我们认为你应该稍事休息了！放下手中的书，去做些别的事情，然后再精神饱满地回来继续阅读下一主题——所有优秀的Java开发人员都应该掌握的并发。

# 现代并发

4

4

## 本章内容

- 并发理论
- 块结构并发
- java.util.concurrent包
- 用分支/合并框架实现轻量并发
- Java内存模型（JMM）

本章会从基本概念开始，并在块结构并发上做短暂停留。在Java 5之前，这是唯一值得把玩的技术，就是搁在现在，也很值得玩味。之后，我们会介绍每个开发人员都应该了解的java.util.concurrent包以及如何使用其提供的基本并发构建块。

我们会以新的分支/合并框架为结尾。看完本章后，你就有能力使用这些新的并发技术了。你还能掌握充分的理论知识，并以此为基础，能够完全理解本书后面讨论到的非Java语言的不同并发视图。

我们不打算在本章把所有的并发知识都讲完，先告诉你一些起步的知识就足够了。另外我们还会告诉你在编写并发代码时需要深入了解些什么，并阻止你在编写代码时涉足险境。但如果你想成为一名真正一流的多线程编程高手，只知道这里讲的知识还不够。这里向你推荐两本专门讨论Java并发编程的书，其中一本是Doug Lea写的《Java并发编程》（第二版，Prentice Hall，1999），另一本是和Brian Goetz跟人合著的《Java并发编程实战》（Addison-Wesley Professional，2006）。

## 但我已经了解线程了！

这是开发人员最常犯的（也许是致命的）错误之一。他们自认为熟悉Thread、Runnable和语言层面那些原始的基础Java并发机制就是合格的并发编程人员了。实际上，并发是个非常广泛的主题，即便是最好的开发人员，从事多线程开发工作多年，有丰富的经验，要想写出优质的多线程代码也很困难，而且很难保证不出问题。

还有一点你应该注意，目前并发领域正如火如荼地开展着研究工作，这些研究肯定会对所用到的Java及其他语言产生影响。如果非要我们挑一个在未来五年中很可能会改变行业惯例的计算机基础领域，那就是并发。

本章的目的是让你了解决定Java并发工作方式的底层平台机制。我们还会充分讨论通行的并发理论和词汇，让你理解其中涉及的问题，并教你认识到让并发正确工作的必要性和在这个过程中所遇到的困难。实际上，这里是我们的起点。

## 4.1 并发理论简介

为了理解在Java中编写并发程序的方法，我们来聊聊相关理论。先讨论一下Java线程模型的基础知识。

之后，我们会讨论系统设计和实现中“设计原则”的影响以及其中最主要的两个原则：安全性和活跃度。我们还会提到其他一些原则，然后讨论这些原则经常相互冲突的原因，以及并发系统中为什么会有开销。

本节的最后我们会看一个多线程系统的例子，并向你证明`java.util.concurrent`是多么自然的编码方法。

### 4.1.1 解释 Java 线程模型

Java线程模型建立在两个基本概念之上：

- 共享的、默认可见的可变状态
- 抢占式线程调度

我们从几个侧面思考一下这两个概念。

- 所有线程可以很容易地共享同一进程中的对象。
- 能够引用这些对象的任何线程都可以修改这些对象。
- 线程调度程序差不多任何时候都能在核心上调入或调出线程。
- 必须能调出运行时的方法，否则无限循环的方法会一直占用CPU。

然而这种不可预料的线程调度可能会导致方法“半途而废”，并出现状态不一致的对象。

某一线程对数据做出修改时，会让其他线程无法见到本应可见的修改。为了缓解这些风险，Java提出了最后一点要求。

- 为了保护脆弱的数据，对象可以被锁住。

Java基于线程和锁的并发非常底层，并且一般都比较难用。为了解决这个问题，Java 5引入了一组并发类库`java.util.concurrent`。这个包中提供了一套编写并发代码的工具，很多程序员都觉得它要比传统的块结构并发原语易用。

### 经验教训

Java是第一个内置多线程编码支持的主流编程语言。这在当时可以说是一个巨大的进步，但15年之后的今天，我们对于如何编写并发代码已经是了若指掌了。

事实证明，Java最初的一些设计决策给大多数程序员编写多线程代码带来了很多困难。这的确很糟糕，因为硬件一直朝着多核处理器方向发展，而唯一能利用好这些核心的就是并发代码。本章会讨论一些在编写并发代码时所遇到的困难。现代处理器对并发编程有着合理的需求，我们在第6章讨论性能时还会涉及其中的一些细节。

随着开发人员编写并发代码的经验越来越丰富，他们发现自己所关注的一些重要系统问题一再出现。我们把这些关注点称为“设计原则”——存在于并发OO系统实际设计中的指导性原则（并经常相互冲突）。

在后面几节，我们会花点时间了解一下其中最重要的几个原则。

4

## 4.1.2 设计理念

Doug Lea在创造他那里程碑式的作品`java.util.concurrent`时列出了下面这些最重要的设计原则：

- 安全性（也叫做并发类型安全性）
- 活跃度
- 性能
- 重用性

下面我们来逐一解读。

### 1. 安全性与并发类型安全性

安全性是指不管同时发生多少操作都能确保对象保持自相一致。如果一个对象系统具备这一特性，那它就是并发类型安全的。

可能你从它的名字就猜出来了，并发可以看做是常规对象建模和类型安全概念的一种延伸。在非并发代码中，要确保不管调用了对象中的什么公开方法，对象最后总是处于一个定义良好并且一致的状态下。通常用来达成这一点的做法是保证对象所有状态都私有，并且开放出来的公开API方法只能以自相一致的方式修改对象状态。

并发类型安全的概念跟对象类型安全一样，但它用在更复杂的环境下。在这样的环境中，其他线程在不同CPU内核上同时操作同一对象。

### 保证安全

保证安全的策略之一是在处于非一致状态时绝不能从非私有方法中返回，也绝不能调用任何非私有方法，而且也绝不能调用其他任何对象中的方法。如果把这个策略跟某种对非一致对象的保护办法（比如同步锁或临界区）结合起来，就可以保证系统是安全的。

### 2. 活跃度

在一个活跃的系统中，所有做出尝试的活动最终或者取得进展，或者失败。

这个定义中的关键词是“最终”——运行中的瞬时故障（尽管不理想，但单独来看这不是问题）和永久故障是不同的。下面这几种底层问题可能会导致系统出现瞬时故障：

- 处于锁定状态或者在等待得到线程锁
- 等待输入（比如网络I/O）
- 资源的暂时故障
- CPU没有足够的空闲时间运行该线程

导致系统出现永久故障的原因较多，其中最常见的是：

- 死锁
- 不可恢复的资源问题（比如NFS不可访问）
- 信号丢失

尽管你对它们可能都已经很熟悉了，但本章后续还是会讨论一下锁定和其他几个问题。

### 3. 性能

系统性能可以通过几种不同的方式量化。我们会在第6章讨论性能分析和优化技术，并且会介绍一些你应该了解的指标。现在，你可以把性能看成是测量系统用给定资源能做多少工作的办法。

### 4. 可重用性

可重用性是第四个设计原则，其他它原则中并没涉及这一点。尽管有时不容易实现，但我们还是非常希望能设计出易于重用的并发系统。用可重用工具集（比如`java.util.concurrent`），并把不可重用的应用代码构建在工具集之上是一种可行的办法。

#### 4.1.3 这些原则如何以及为何会相互冲突

设计原则经常相互对立，这种紧张关系使得并发系统的设计很难达到优秀的水准。

- 安全性与活跃度相互对立——安全性是为了确保坏事不会发生，而活跃度要求见到进展。
- 可重用的系统倾向于对外开放其内核，可这会引发安全问题。
- 一个安全但编写方式幼稚的系统性能通常都不会太好，因为里面一般会用大量的锁来保证安全性。

最终应该尽量让代码达到一种平衡的状态，使其能够灵活地适用于各种问题，却又能保证安全性，同时活跃度和性能也可以达到一定水平。这种境界相当高，但你很幸运，我们马上教你一些实战技巧。下面是几个最常见的粗浅办法。

- 尽可能限制子系统之间的通信。隐藏数据对安全性非常有帮助。
- 尽可能保证子系统内部结构的确定性。比如说，即便子系统会以并发的、非确定性的方式进行交互，子系统内部的设计也应该参照线程和对象的静态知识。
- 采用客户端应用必须遵守的策略方针。这个技巧虽然强大，却依赖于用户应用程序的合作程度，并且如果某个糟糕的应用不遵守规则，便很难发现问题所在。
- 在文档中记录所要求的行为。这是最逊的办法，但如果代码要部署在非常通用的环境中，就必须采用这个办法。

开发人员应该了解所有可能的安全机制，而且尽可能采用最强的技术，但同时你也应该知道，在某些情况下只能采用那些比较逊的办法。

#### 4.1.4 系统开销之源

并发系统中的系统开销是与生俱来的，这些开销来自：

- 锁与监测
- 环境切换的次数
- 线程的个数
- 调度
- 内存的局部性<sup>①</sup>
- 算法设计

你应该以此为基础在大脑中列一个检查列表。在编写并发代码时，应该确保自己对列表中的每一项都认真考虑过了，然后再来“搞定”代码。

#### 算法设计

这是一个能让开发人员脱颖而出的领域。无论用什么语言，学习算法设计都能让你成为更好的程序员。在这里我们向你推荐由Thomas H. Cormen等人编著的《算法导论》(MIT, 2009)和Steven Skiena写的《算法设计手册》(Springer-Verlag, 2008)。无论你是想了解单线程算法还是想学习并发算法，它们都是值得阅读的好书。

本章会提到许多系统开销的源头（还有第6章讨论性能的部分）。

#### 4.1.5 一个事务处理的例子

本节前面的内容都太理论化了，所以我们补充一个并发程序设计的例子来实证一下。在这个例子中你将看到如何用java.util.concurrent中的高层类完成这个任务。

假设有一个基本事务处理系统。构建这种程序有个简单的标准办法，就是先将业务流程的不同环节对应到应用程序的不同阶段，然后用不同的线程池表示不同的应用阶段，每个线程池逐一接受工作项，在对每个工作项进行一系列的处理后，交给下一个线程池。通常来说，好的设计会让每个线程池所做的处理集中在一个特定功能区内。如图4-1所示。

如果你设计成这样的程序，就可以提高吞吐量，因为可以设计成同时处理几个工作项。比如在检查一个工作项的信用情况时，可以检查另一个工作项的库存。根据应用程序的处理细节不同，甚至可以同时检查多个订单的库存。

<sup>①</sup> 局部性指的是程序行为的一种规律：在程序运行中的短时间内，程序访问数据位置的集合限于局部范围。局部性有两种基本形式：时间局部性与空间局部性。时间局部性指的是反复访问同一个位置的数据；空间局部性指的是反复访问相邻的数据。——译者注

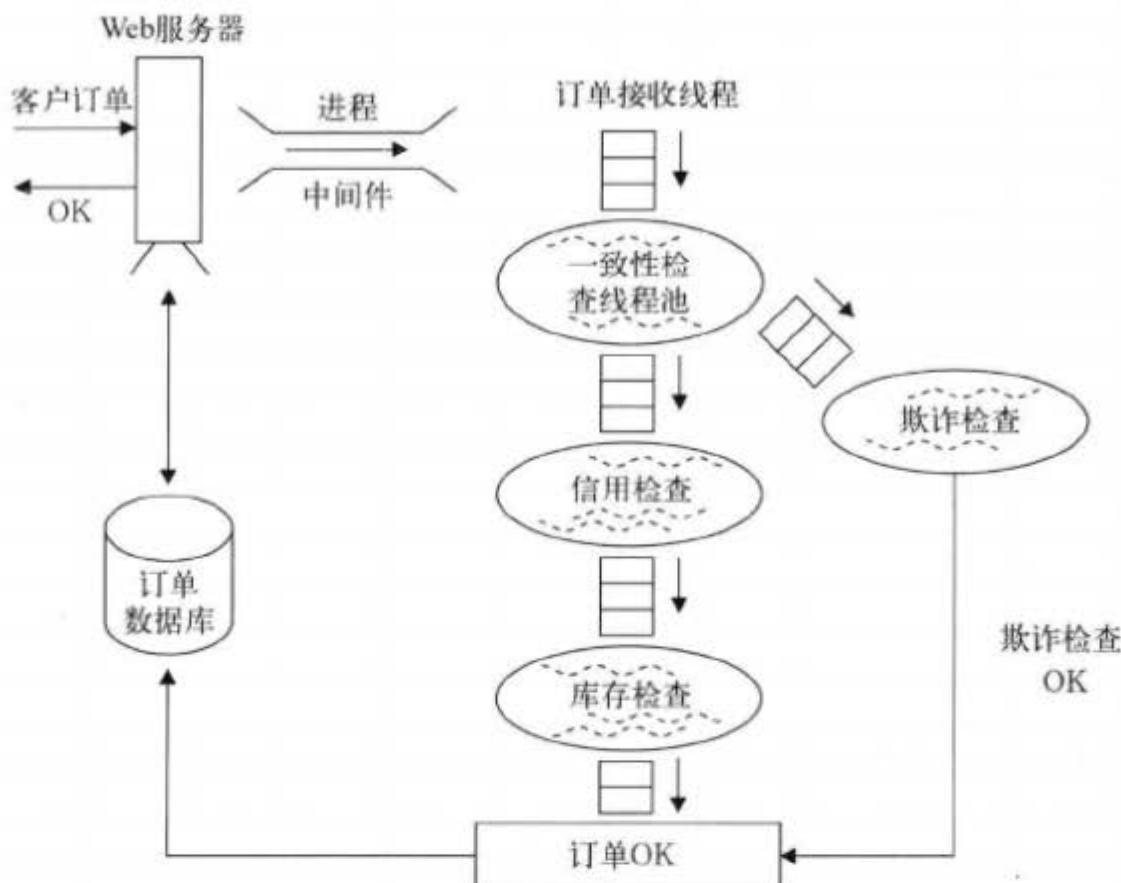


图4-1 多线程应用程序示例

这种设计非常适合用`java.util.concurrent`包中的类来实现。这个包里有用于执行任务的线程池(`Executors`类中有一套工厂方法可以创建它们)和在不同线程池之间传递工作的队列，还有并发数据结构(可以用来构建共享缓存，或用于其他用途)和很多其他底层工具。

你可能会问，在Java 5之前，还没有这些类时是怎么办的？一般情况下，开发小组会自己编写并发编程类库，最终会构建出跟`java.util.concurrent`类似的组件。但这种定制组件大多存在设计缺陷，还会有难以捉摸的并发bug。如果没有`java.util.concurrent`，开发人员就得重复实现其中的大部分组件(可能会有很多bug，测试也不充分)。

请记住这个例子，我们要转入下一主题——温习一下Java的“传统”并发，并深入了解用它编程困难的原因。

## 4.2 块结构并发 (Java 5 之前)

本章大部分内容都在讨论块同步并发方式的替代方案。如果你想从我们的讨论中获益，就需要深刻理解传统并发的优缺点的重要性。

为此我们要讨论用到`synchronized`、`volatile`等并发关键字的那种原始、低级的多线程编程方式。我们把这个讨论放在设计原则的情境中，并且会着眼于下一节将要讨论的内容。

之后我们会简略地解释一下线程的生命周期，然后讨论常见的并发编程技巧和陷阱，比如完全同步的对象，死锁，`volatile`关键字和不变性。

我们先重温一下同步吧。

### 4.2.1 同步与锁

你知道的，`synchronized`既可以用在代码块上也可以用在方法上。它表明在执行整个代码块或方法之前线程必须取得合适的锁。对于方法而言，这意味着要取得对象实例锁（对于静态方法而言则是类锁）。对于代码块，程序员则应该指明要取得哪个对象的锁。

在任何一个对象的同步块或方法中，每次只能有一个线程进入；如果其他线程试图进入，JVM会挂起它们。无论其他线程试图进入的是该对象的同一同步块还是不同的同步块，JVM都会如此处理。这种结构在并发理论中被称为临界区。

**注意** 你有没有想过Java中用于确立临界区的关键字为什么是`synchronized`？为什么不是“critical”或“locked”？同步的是什么？我们会在4.2.5节回到这一话题上来，但如果你不知道，或者从来没想过这个问题，你最好花几分钟想一想再继续。

本章要讨论一些比较新的并发技术。但既然说到了同步，我们就顺便看看与Java中的同步和锁相关的一些基本事实吧。希望你对这里的大多数（或全部）知识都已经烂熟于心了。

- 只能锁定对象，不能锁定原始类型。
- 被锁定的对象数组中的单个对象不会被锁定。
- 同步方法可以视同为包含整个方法的同步(`this`) { ... }代码块（但要注意它们的二进制码表示是不同的）。
- 静态同步方法会锁定它的`Class`对象，因为没有实例对象可以锁定。
- 如果要锁定一个类对象，请慎重考虑是用显式锁定，还是用`getClass()`，两种方式对子类的影响不同。
- 内部类的同步是独立于外部类的（要明白为什么会这样，请记住内部类是如何实现的）。
- `synchronized`并不是方法签名的组成部分，所以不能出现在接口的方法声明中。
- 非同步的方法不查看或关心任何锁的状态，而且在同步方法运行时它们仍能继续运行。
- Java的线程锁是可重入的。也就是说持有锁的线程在遇到同一个锁的同步点（比如一个同步方法调用同一个类内的另一个同步方法）时是可以继续的。

**警告** 在其他语言中存在不可重入的锁机制（用java也能实现相同的效果，如果你想了解那些让人看了毛骨悚然的详细信息，请参见`java.util.concurrent.locks`中的`ReentrantLock`的Javadoc），但和它们打交道太痛苦了，除非你真的知道自己在做什么，否则还是躲之为妙。

对Java同步的温习就到此为止吧。现在我们来看一下线程在其生命周期中的状态变迁。

## 4.2.2 线程的状态模型

图4-2展示了线程生命周期的发展过程——从创建到运行，到再次运行之前（或被资源阻塞）可能被挂起，再到最终完成。

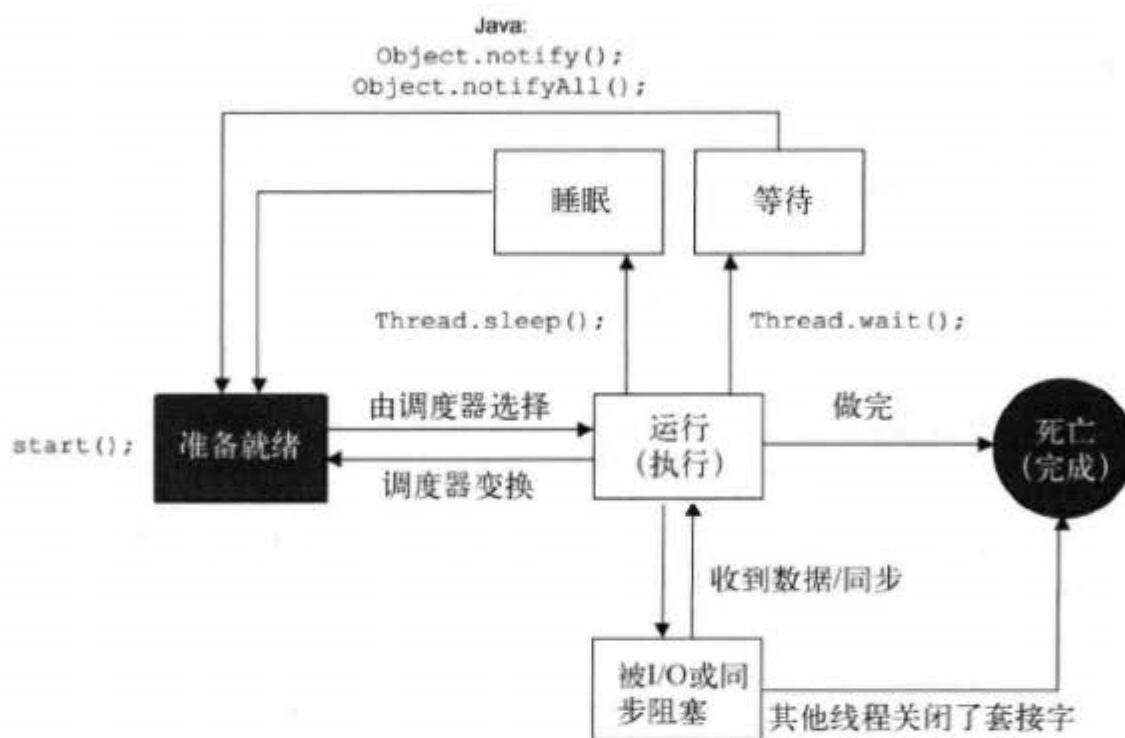


图4-2 Java的线程状态模型

线程最初创建时处于就绪（Ready）状态。然后调度器会找个核心来运行它，如果机器负载过重，那它就可能需要多些时间。开始运行之后，线程通常会消耗掉分配给它的时间，然后回到就绪状态，等到下次再有处理器分配时间片给它。这是我们在4.1.1节提过的抢占式线程调度的标准动作。

除了由调度器发起的标准动作，线程本身也能表明它此时无法使用核心工作。这可能是因为程序代码通过`Thread.sleep()`告诉线程在继续之前应该暂停，或者因为线程必须等待通知（通常需要满足某些外部条件）。这时线程会从核心中移走，并释放它持有的锁。只有通过唤醒才能再次运行线程（在达到睡眠时长之后，或收到了恰当的信号），进入就绪状态。

线程可能会因为等待I/O或等待获取其他线程持有的锁而被阻塞。这时线程并没有被交换出核心，而是仍然处于繁忙状态，等着获取可用的锁或数据。在得到锁或数据之后，线程会继续执行直到它的时间片结束。

我们接下来讨论一个著名的解决同步问题的办法——完全同步对象。

## 4.2.3 完全同步对象

前面介绍了并发类型安全的概念，还提到了一种用来达成这种安全性的策略（在“保证安全”的边栏中）。现在我们来看一下这个策略更完整的描述，它通常被称为完全同步对象。如果一个

类遵从下面所有规则，就可以认为它是线程安全并且活跃的。

一个满足下面所有条件的类就是完全同步类。

- 所有域在任何构造方法中的初始化都能达到一致的状态。
- 没有公共域。
- 从任何非私有方法返回后，都可以保证对象实例处于一致的状态（假定调用方法时状态是一致的）。
- 所有方法经证明都可在有限时间内终止。
- 所有方法都是同步的。
- 当处于非一致状态时，不会调用其他实例的方法。
- 当处于非一致状态时，不会调用非私有方法。

假定有一个分布式微博工具，代码清单4-1是其后台中的类。在它的propagateUpdate()方法被调用时，ExampleTimingNode类会收到更新，也可以通过查询看它是否收到了特定更新。这是经典的读写操作相互冲突的情景，需要通过同步防止出现不一致状态。

#### 代码清单4-1 完全同步类

```
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;
    }
}
```

这是一个既安全又活跃的类，第一眼看上去让人感觉很了不起。但随之而来的是性能问题，既安全又活跃的东西速度不一定也能很快。必须用synchronized去协调对Map arrivalTime的所有访问（get和put），而这个锁最终会把你的速度拖慢。这是并发处理方式的主要问题。

### 代码的脆弱性

除了性能问题，代码清单4-1中的代码还很脆弱。你看，它从来不会在同步方法之外去碰arrivalTime，实际上只是调用get和put方法，但这只有在代码量很小的情况下才有可能。在真实的大型系统中，代码太多而无法实现这种方法。同时，bug也很容易潜伏在庞大的代码库中，这也是Java社区开始寻求更完善的解决方法的另一个原因。

#### 4.2.4 死锁

并发的另一个经典问题是死锁。代码清单4-2稍微扩展了一下上个例子。在这一版中，除了记录最近一次更新的时间，每个节点收到更新时还会通知另外一个节点。

这段代码试图构建一个多线程的更新处理系统。注意，这段代码是为了解释死锁，不要把它用到你的工作中。

#### 代码清单4-2 死锁的例子

```
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());
    }
}

final MicroBlogNode local =
    new MicroBlogNode("localhost:8888");
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);
    }
})
```

关键字final是必需的

第一个更新发送给第一个线程

```

}).start();

new Thread(new Runnable() {
    public void run() {
        other.propagateUpdate(second, local);
    }
}).start();

```

第二个更新发送给第二个线程

乍一看，这段代码没什么毛病。有两个更新分别发送给不同的线程，每个都必须由后备线程进行确认。这看起来不是什么离奇古怪的设计——如果一个线程失效，另外一个线程还可以挑起重担。

如果你运行这段代码，一般都会碰到死锁——两个线程都说自己收到了更新，但它俩谁都不会以备份线程的身份确认收到了更新。因为每个线程在确认方法能够确认之前都要求另外一个线程释放线程锁，如图4-3所示。

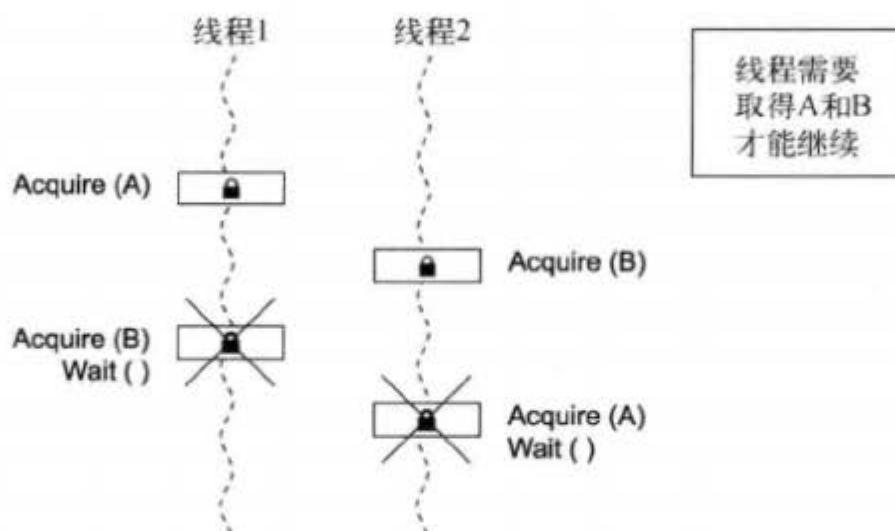


图4-3 死锁线程

有一个处理死锁的技巧，就是在所有线程中都以相同的顺序获取线程锁。在前例中，第一个线程以A、B的顺序获取锁，而第二个线程获取锁的顺序是B、A。如果两个线程都用A、B的顺序，死锁的情况就可以避免，因为第二个线程在第一个线程完成并释放锁之前会一直被阻塞住。

就完全同步对象方式而言，要防止这种死锁出现是因为代码破坏了状态一致性规则。当有消息到达时，接受节点会在消息处理过程中调用另外一个对象——它发起这个调用时状态是不一致的。

接下来，我们会返回来解释前面抛出的那个问题：为什么Java中用来标识临界区的关键字是synchronized？这会引导我们转而讨论不可变性和关键字volatile。

#### 4.2.5 为什么是synchronized

最近几年并发编程变化最大的是硬件领域。在以前，程序员可能常年累月都碰不到需要支持多处理器核心（两个或最多三个）的系统。因此并发编程过去主要考虑如何分享CPU时间——线程们在单核上轮流上位，相互调换。

现如今，任何比手机大点儿的东西都是多核的，所以我们的认知模型也该换换了，应该把多个线程在同一物理时刻运行在不同核心（并且很可能会操作共享的数据）的情况也考虑在内。如图4-4所示。为了提高效率，同时运行的每个线程可能都会有它正在处理的数据的缓存复本。记住这幅图，让我们回到选择用什么关键字来表示被锁定的代码块或方法这个问题上。

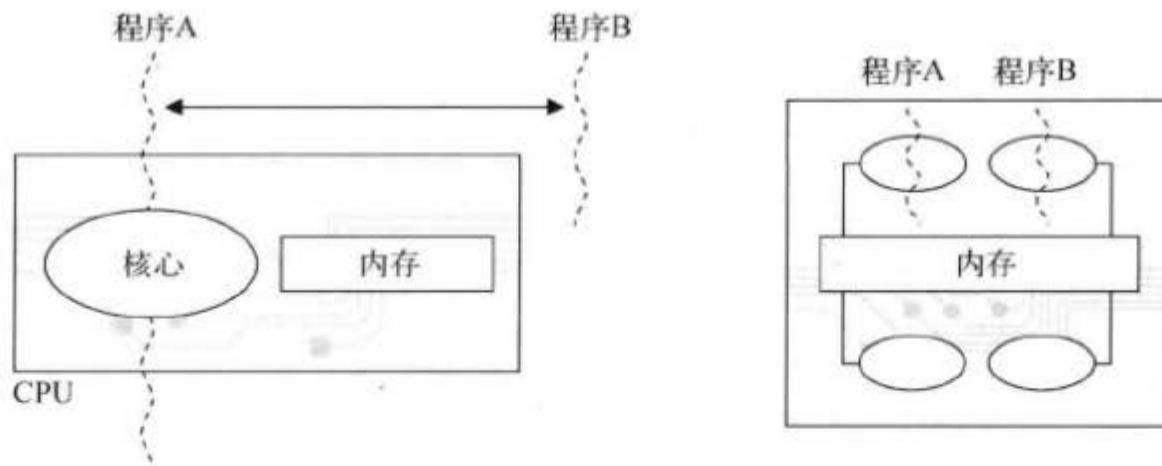


图4-4 考虑并发和线程的新、老方式

我们在前面问过，代码清单4-1中被同步的是什么？答案是：被同步的是在不同线程中表示被锁定对象的内存块。也就是说，在synchronized代码块（或方法）执行完之后，对被锁定对象所做的任何修改全部都会在线程锁释放之前刷回到主内存中，如图4-5所示：

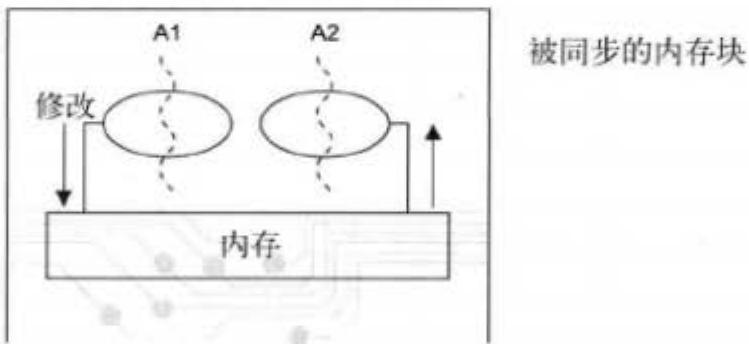


图4-5 不同线程对一个对象的修改通过主内存传播

另外，当进入一个同步的代码块，得到线程锁之后，对被锁定对象的任何修改都是从主内存中读出来的，所以在锁定区域代码开始执行之前，持有锁的线程就和锁定对象主内存中的视图同步了。

#### 4.2.6 关键字 volatile

Java在其混沌初开的时期（Java 1.0）就已经把volatile作为关键字了，它是一种简单的对象域同步处理办法，包括原始类型。一个volatile域需遵循如下规则：

- 线程所见的值在使用之前总会从主内存中再读出来。
- 线程所写的值总会在指令完成之前被刷回到主内存中。

可以把围绕该域的操作看成是一个小小的同步块。程序员可以借此编写简化的代码，但付出的代价是每次访问都要额外刷一次内存。还有一点要注意，`volatile`变量不会引入线程锁，所以使用`volatile`变量不可能发生死锁。

更加微妙的是，`volatile`变量是真正线程安全的，但只有写入时不依赖当前状态（读取的状态）的变量才应该声明为`volatile`变量。对于要关注当前状态的变量，只能借助线程锁保证其绝对安全性。

#### 4.2.7 不可变性

不可变对象的应用是十分有价值的技术。这些对象或没有状态，或只有`final`域（因此只能在构造方法中赋值）。它们总是安全而又活跃的。它们的状态不能修改，所以不可能出现不一致的情况。

可这样对象初始化的所有值都必须传入构造方法。这会导致构造方法的参数很多，看起来又蠢又笨。因此很多程序员选择工厂方法`FactoryMethod`代替构造方法。工厂方法很简单，就是类中的一个静态方法，用来代替构造方法创建新对象。此时构造方法通常被声明为`protected`或`private`的，从而使工厂方法成为实例化对象的唯一办法。

但是还存在要将众多参数传入`FactoryMethod`的问题。有时候这不太方便，尤其是初始化对象所需的状态参数有多个不同来源时。

构建器模式可以解决这个问题。它由两部分组成：一个是实现了构建器泛型接口的内部静态类，另一个是构建不可变类实例的私有构造方法。

内部静态类是不可变类的构建器，开发人员只能通过它获取不可变类的新实例。比较常见的实现方式是让构建器类拥有与不可变类一模一样的域，但构建器的域是可修改的。

下面这段代码展示了如何建立不可变的微博更新模型（根据本章前面的例子所构建）。

**代码清单4-3 不可变对象及构建器**

```
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;
    }

    public static class Builder
        implements ObjBuilder<Update> {
        private Author author;
        private String updateText;
    }
}
```

构建器接口

必须在构造方法中初始化  
final域

构造器类必须是静态  
内部类