HZ BOOKS
华章科技

The Pragmatic Programmers

Java并发编程领域的里程碑之作，资深Java技术专家、并发编程专家、敏捷开发专家和Jolt大奖得主撰写，Amazon五星畅销书

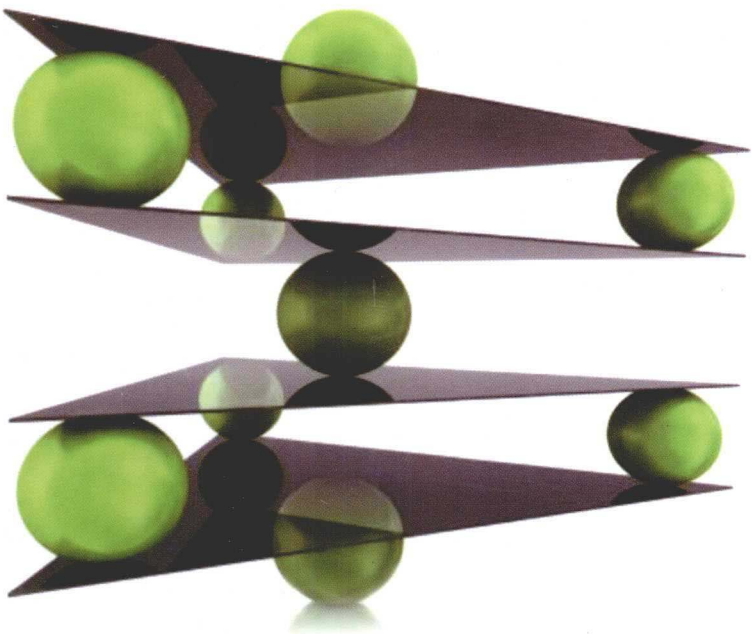
系统深入地讲解在JVM平台上如何利用JDK同步模型、软件事务内存模型和基于角色的并发模型进行并发编程，列举丰富示例，包含大量编程技巧和最佳实践

华章程序员书库

Programming Concurrency on the JVM
Mastering Synchronization, STM, and Actors

Java虚拟机并发编程

（美）Venkat Subramaniam 著
薛笛 译



机械工业出版社
China Machine Press

这是一部优秀的著作！Venkat巧妙地带领我们经历了当今JVM开发人员在多线程编程时所面临的许多设计和实现决策。这本书通过通俗易懂的语言和丰富的例子（使用了当前多种开源工具和JVM语言）使得这个复杂的主题更加一目了然。

—— Albert Scherer 福莱特高等教育集团有限公司电子商务技术部经理

如果JVM是你的首选平台，那么这绝对是你必读的一本书。购买并阅读，然后让你所有的团队成员人手一本。你将顺利找到一条优秀并发问题解决方案之路。

—— Raju Gandhi Integrallis软件有限责任公司高级顾问

这本书非常透彻地阐释了一个极其重要的主题。

—— Chris Richardson CloudFoundry.com创始人

当前计算机领域对新的并发模型和JVM上新语言的兴趣和应用正在逐年递增。Venkat的这本书把二者绑在一起，向一线的开发人员展示了如何构建应用程序，并尽可能地利用现有的库，即使它们是使用不同的语言构建的。

—— Alex Miller Revelytix公司高级工程师

Venkat将引领你进入并发的世界。在开始STM和角色的精彩冒险之旅之前，他教授了Java JDK内置的现代并发工具。我真诚地相信，这本书注定是目前已有的最重要的并发编程书籍之一。Venkat再次做到了这一点！

—— Matt Stine AutoZone公司技术架构师

当前并发是一个热门话题，Venkat将带你领略JVM上有效的并发编程技巧和技术。更重要的是，通过比较和对比五种不同的JVM语言中并发编程的方法，你将会开拓视野。

—— Scott Leberknight Near Infinity公司首席

The Pragmatic Programmers

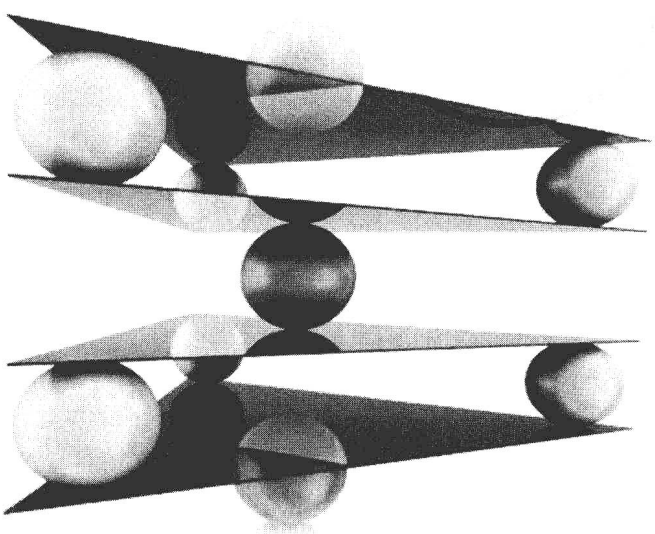
客服热线：(010) 88378991, 88361066
购书热线：(010) 68326294, 88379649, 68995259
投稿热线：(010) 88379604
读者信箱：hzjsj@hzbook.com
华章网站：www.hzbook.com
网上购书：www.china-pub.com

上架指导：计算机/程序设计
ISBN 978-7-111-41893-1
9 787111 418931 >
定价：59.00元

Programming Concurrency on the JVM
Mastering Synchronization, STM, and Actors

Java虚拟机并发编程

（美）Venkat Subramaniam 著
薛笛 译



机械工业出版社
China Machine Press

图书在版编目（CIP）数据

Java 虚拟机并发编程 /（美）苏布拉马尼亚姆（Subramaniam, V.）著；薛笛译 . —北京：机械工业出版社，2013.4
（华章程序员书库）

书名原文：Programming Concurrency on the JVM：Mastering Synchronization, STM, and Actors

ISBN 978-7-111-41893-1

Ⅰ. J… Ⅱ. ①苏… ②薛… Ⅲ. JAVA 语言 – 程序设计 Ⅳ. TP312

中国版本图书馆 CIP 数据核字（2013）第 055404 号

本书版权登记号：图字：01-2011-7873

本书是 Java 并发编程领域的里程碑之作，由资深 Java 技术专家、并发编程专家、敏捷开发专家和 Jolt 大奖得主撰写，Amazon 五星级畅销书。它系统深入地讲解在 JVM 平台上如何利用 JDK 同步模型、软件事务内存模型和基于角色的并发模型更好地进行并发编程。全书以示例驱动，通俗易懂，包含大量编程技巧、注意事项和最佳实践。要重点强调的是，本书并不仅仅只适合于 Java 语言的并发编程，它还适用于 Clojure、Groovy、JRuby 和 Scala 等所有运行在 JVM 平台上的编程语言。

本书共 10 章，分为五个部分。第一部分：并发策略，阐释了影响并发性的因素、如何有效实现并发，以及并发的设计方法等；第二部分：现代 Java/JDK 并发，讨论了现代 Java API 的线程安全和效率，以及如何处理已有应用程序中的现实问题和重构遗留代码时的原则；第三部分：软件事务内存，深入讨论了 STM 并就如何在各种主要的 JVM 语言里使用 STM 给出了指导意见；第四部分：基于角色的并发，详细讲解了如何在基于角色的模型下消除并发问题以及如何在自己的首选语言中使用角色模型；第五部分：后记，回顾了本书讨论的解决方案并总结了并发编程中的注意事项和最佳实践。

机械工业出版社（北京市西城区百万庄大街 22 号 邮政编码 100037）

责任编辑：秦 健

北京京师印务有限公司印刷

2013 年 5 月第 1 版第 1 次印刷

186mm × 240mm • 14.5 印张

标准书号：ISBN 978-7-111-41893-1

定 价：59.00 元

凡购本书，如有缺页、倒页、脱页，由本社发行部调换

客服热线：（010）88378991 88361066 投稿热线：（010）88379604

购书热线：（010）68326294 88379649 68995259 读者信箱：hzjsj@hzbook.com

译 者 序

对于 Java 并发编程领域而言，JDK5 的发布绝对具有里程碑式的意义。由该领域大师级人物 Doug Lea 亲自操刀的新并发 API 以及重新实现的并发容器使得开发人员摆脱 synchronized、notify()、wait() 这些原始的同步原语，为并发应用的开发提供了巨大的便利和性能提升，使 Java 并发编程前进到新的阶段。时过境迁，这几年来，虽然没有再出现类似于新并发 API 这样惊世骇俗的产品，但随着 Java 平台的不断演进，尤其在 JVM 开放了对动态语言的支持之后，一些新的语言及其背后所蕴含的设计方法和编程模型也被引入 JVM，而这些新的设计方法和编程模型也深刻地影响着 JVM 并发编程领域的发展。

但是，出于对新语言的学习门槛、稳定性、性能、系统切换和维护成本等方面的顾虑，互联网企业，尤其是国内的大多数企业对这些新语言、新技术都持比较保守的态度，所以在国内我们很难接触到由这些新语言所引领的 JVM 并发编程的最新成果。当我们打开满是灰尘的并发编程工具箱，发现里面还是 2007 年起就一直在用的并发 API，虽然用多了也感觉很顺手，但这些真的已经足够了吗？

本书为我们推开了一扇通往并发编程新领域的大门。作者以演讲般娓娓道来的方式告诉我们，其实我们一直都被共享可变性设计的缺陷折磨着却不自知。而我们仿佛是《黑客帝国》电影里生活的人们一样，在服下那粒可以看透真相的小药丸之前，总以为世界就应该是自己所熟悉的样子。再回到 JVM 并发编程领域，正是由于新的并发 API 太过成功，以致很多人已经形成了以新并发 API 为基础、以共享可变性设计为核心的固有套路和思维定式，认为这种方式所带来的饥饿、死锁、竞争条件等问题是并发编程固有的复杂性，从而隐忍和压抑着心中为解决这些问题而遭受的痛苦。从这个角度来说，作者很像《黑客帝国》中引领救世主 Neo 走出 Matrix 虚幻世界的导师墨菲斯，而本书就是那粒小药丸，吃了之后就可以看到 JVM 并发编程领域中不一样的风光，同时会感叹：哇哦，原来还有这么多新玩法！

相对于其他讲述并发编程的书籍，本书的亮点在于：

- **理论叙述深入浅出。**由于作者是一位出色的演讲者，所以本书的理论部分也大量运用了例证、类比等叙述手段，使得原本枯燥、生涩的理论叙述变得生动鲜活，从而更易于读者理解和接受。
- **实践内容极具实用性。**本书的内容涵盖了当前流行且成熟的多种 JVM 并发编程解决方案，包括 Java7 Fork-Join API、STM、基于角色的模型，并且在每叙述完一种新模型之后都会有独立的一章专门用于展示前章所述模型在 Clojure、Groovy、Java、JRuby 和 Scala 中的用法和注意事项，有助于使用不同语言的读者都能找到相应的解决方案。虽然 Java 才是本书绝大多数国内读者的工作语言，但是上述几种语言对

于本书所述的并发模型的支持方面可谓各胜擅长，有很多在 Java 中实现起来语法很繁琐的模型在其他语言中可能几行代码就可以完成。所以，了解一下其他语言简洁优雅的实现方式，也不失为开阔眼界的好素材，以后再有类似需求的时候也可以多一种选择。

这里我要感谢本书的策划编辑杨福川，是他让我有机会参与本书的翻译工作，他的认可和鼓励为我完成本书的翻译提供了强大的助力。我还要感谢本书的责任编辑，他严谨、细致的专业态度让我受益匪浅。最后，我要感谢我的老婆璐璐，译书的这段时间里，我经常在电脑前敲键盘而无暇陪伴，感谢老婆的理解与支持。

本书所讲述的内容并不艰涩，领域也属我比较擅长的范畴，但想要将这位爱举例子的作者那很口语化、有点絮叨并且语法不那么严谨的英文转换成流畅且符合国人阅读习惯的文字着实不易。所以，虽然我很希望能够在人生第一部译作中力求完美，呈现给大家一部“很好读”的书，但终因时间和精力有限，译稿可能存在一些疏漏和翻译生硬之处，恳请各位读者批评指正。

前　言

除了咖啡因，我想没有什么能比写出一段执行速度飞快的代码更能令程序员们兴奋了。然而我们如何才能满足这种对计算速度的渴求呢？诚然，摩尔定律可以帮我们解决部分问题，但多核处理器才代表了未来真正的发展方向。为了能够充分发挥多核处理器的优势，我们每个程序员都应该熟悉并掌握并发编程的技能。

在一个并发程序中，两个或多个动作一般会同时发生。例如，一个并发程序可能需要一边下载多份文件、另一边还要执行一些计算任务以及数据库更新操作。在 Java 中，我们通常会使用线程来实现并发程序。虽然 Java 虚拟机在其设计之初就已经在语言层面支持了多线程，但并发编程的方法仍然在不断地进化和完善当中，而这正是我们在本书想要与你分享的内容。

并发编程的难点在于，我们既希望享受并发编程带来的好处，同时又不能让同步问题引火烧身。在 JVM 上的语言中，启动多个线程是很容易的，但它们彼此之间的执行顺序却是不可预知的。所以很快我们就将投身到一场为了协调线程并使其可以一致地处理数据而进行的战斗中去。

假设我们想要从 A 点出发快速抵达 B 点，根据旅行时间有多重要、有哪些可选的交通工具、预算是多少等因素，我们可以选择步行、坐巴士、自驾、坐和谐号或坐飞机飞过去。类似，为了提高 Java 代码的执行速度，我们同样也有很多选择。在 JVM 平台上，我们用得最多的一般是下面这三种并发模型：

- 称为“同步并受罪”的模型
- 软件事务内存（Software Transactional Memory，STM）模型
- 基于角色（actor-based）的并发模型

之所以将大家所熟知的 JDK 同步模型说成是“同步并受罪”，是因为如果我们忘记对共享可变状态进行同步或在错误的层级对其进行同步，所得到的结果将是不可预测的。如果足够幸运的话，我们还能在开发过程中捕获该问题；而如果我们没注意到这些问题，产品发布之后就会暴露出各种古怪且难以定位和重现的问题。而届时我们也无法从这些天杀的代码里找到任何编译错误、警告乃至任何简单的线索。

那些没有对共享可变状态的同步访问进行有效管理的程序本质上都是有问题的，但 Java 编译器却对此无动于衷。用纯 Java 对可变性进行编程就仿佛在和整天挑你毛病的丈母娘一起工作那样难受。相信你应该已经体会到这种痛苦了。

当我们撰写并发程序的时候，有如下三种方法可以帮助我们有效避免遇到并发所带来的问题：

- 在合适的地方进行正确的同步。

- 不共享状态。
- 不改变状态。

在使用现代 JDK 的并发 API 的时候，我们不得不花费大量精力来保证在合适的层级进行准确的同步。而 STM 则可以对用户隐藏同步操作并极大地减少出错几率。另一方面，基于角色的模型可以帮助我们避免使用共享状态，而避开共享状态正是我们赢得这场并发战役的秘密武器。

在本书中，我们将采用示例驱动的方法来学习上述三种模型以及如何利用这三种模型来更好地实现并发。

本书的目标读者

本书主要面向那些有一定经验的、正在使用诸如 Java、Clojure、Groovy、JRuby 和 Scala 这几种语言并且有兴趣进一步学习如何在 JVM 上管理和利用并发能力的 Java 程序员。

如果你是 Java 新手，那么本书将不会对你学习 Java 的基础功能有任何帮助。市面上有很多介绍 Java 编程基础的优秀教程可供选择，你可以先从这些书开始学起。

如果你已经在 JVM 平台上积累了相当的编程经验，但发觉还需要一些有助于进一步深入了解并发编程的资料，那么本书就是为你准备的。

如果你只对 Java 和 JDK 提供的解决方案，即 Java 线程和并发库感兴趣，那么我推荐你阅读两本非常优秀的专著，即 Brian Goetz 的《Java Concurrency in Practice》[Goe06] 和 Doug Lea 的《Concurrent Programming in Java》[Lea00]。这两本书在 Java 内存模型以及如何确保线程安全和一致性方面提供了非常丰富的信息。

本书旨在教会你如何使用 JDK 提供的方案来解决某些实际的并发问题，当然其中也会包含一些额外的技巧和方法。此外，你还将学到一些有助于更方便地实现隔离可变性的第三方 Java 库的知识。最后，你还将学到一些通过消除显式锁来降低复杂性和出错概率的类库等方面的内容。

本书意在帮助你学习一些目前已经比较成熟的工具和方法，以便你可以更方便地从中找出最适合解决你当前所面临的并发问题的方案。

本书的主要内容

本书将帮助你学习三种不同的并发解决方案，即现代 Java JDK 并发模型、软件事务内存（STM）和基于角色的并发模型。

本书主要分为 5 部分，即并发策略、现代 Java/JDK 并发模型、软件事务内存、基于角色的并发模型和后记。

在第 1 章中，我们将讨论是什么使得并发编程如此有用以及并发编程为何如此难以掌握。除此之外，该章还将对上面提到的三种并发模型进行简单介绍。

在深入研究上述三种并发解决方案之前，我们将在第 2 章先介绍一些影响程序并发性和加速效果的关键因素，并讨论如何实现有效并发的相关策略。

正如将在第 3 章中讨论的那样，我们采用的设计方法将产生两种截然不同的效果，即要么使我们在并发的海洋里乘风破浪尽情遨游，要么会让我们淹没其中连个水花都看不见。

自 Java 引入以来，其并发 API 的演进和增强一直非常迅速。所以第 4 章我们将讨论如何利用现代 Java API 来解决线程安全和并发性能问题。

鉴于我们十分渴望避免使用共享可变状态，所以在第 5 章我们将探讨一些解决现存应用程序实际问题的方法以及一些在重构历史遗留代码时所需要注意的问题。

第 6 章我们将深入探讨 STM，并学习如何使用 STM 来解决应用程序（特别那些读多写少的应用程序）中绝大部分的并发问题。

第 7 章我们将学习如何在几个不同的 JVM 语言中使用 STM。

第 8 章我们将学习到，如果我们依托隔离可变性进行设计，那么基于角色的模型就能够帮助我们彻底解决所有并发问题。

同样，如果你对几种不同的 JVM 语言都有兴趣，那么你将会在第 9 章学到如何在你偏爱的语言中使用基于角色的模型。

最后，我们将在第 10 章回顾本书前面所讨论过的所有解决方案并将其总结为一些简明扼要的知识点来帮助你更好地理解和实践。

并发还是并行

在业界，这两个术语其实并没有很明显差别，不同的人对于二者区别的阐述也都是各执一词（请不要就此问题并发地对他人提问……抑或我应该说不要并行提问）。

我们姑且把二者概念上的争论放到一边，先考虑一些更实际的问题。假设我们有一个运行在单核 CPU 机器上的多线程应用程序，在公司升级硬件的时候把该程序重新部署到了一台多核 CPU 的机器上。但其实无论底层硬件的部署方式如何变化，我们的代码本质上还是运行在一个独立的 JVM 进程上的，所以我们在开发过程中所要考虑问题也都基本相同，即如何创建和管理线程？如何保证数据完整性？如何处理锁和同步？以及我们创建的线程是否在合适的时间跨越内存栅栏？……

所以不管我们称之为并发也好还是并行也罢，能够真正解决问题才是保证程序正确、高效运行的关键。而这正是本书所关注的重点。

多语言程序员如何使用并发功能

Java 这个词在今天已经代表一个平台多过一门编程语言，而带有丰富类库的 JVM 已经逐渐演变成一个非常强大的平台。但与此同时，Java 语言却显得愈发老态龙钟。近些年来，在 JVM 平台上忽然涌现出了一批有趣且强大的语言，如 Clojure、JRuby、Grovvy 和 Scala。

在这些现代 JVM 语言中，部分语言如 Clojure、JRuby 和 Groovy 都是动态类型的，而其他如 Clojure 和 Scala 则深受函数式编程的影响。但这些语言都有一个共同的特点，那就是语法简洁且表达能力强。虽然学习这些语言的语法、范式或区别有一定的门槛，但学成之后我们可以用比 Java 更少的代码来实现相同的功能。更令人兴奋的是，我们可以将这些

JVM 语言的代码与 Java 代码混用，这样我们就成了名副其实的多语言程序员，请参阅附录 2 中 Neal Ford 有关“多语言程序员”的论述。

在本书中，我们将学习如何使用 java.util.concurrent API，以及如何通过 Akka 和 GPars 来使用 STM 和基于角色的模型。此外，我们还将学习如何在 Clojure、Java、JRuby、Groovy 和 Scala 中进行并发编程。如果你已经或打算使用其中任何一种语言，那么本书将会为你介绍在使用这些语言进行并发编程时所需要注意的各种相关事项。

示例和性能测评

虽然本书的大部分示例都是用 Java 实现的，但是你也会看到很多用 Clojure、Groovy、JRuby 和 Scala 实现的示例代码。在这些示例当中，我已尽量消除各语言之间的语法差异并尽可能避开语言专属的习惯用法，其目的主要是让大多数习惯使用 Java 的程序员更容易阅读和理解用其他语言实现的代码示例。

下面是本书所使用的语言和类库的版本汇总：

- Akka 1.1.3（http://akka.io/downloads）
- Clojure 1.2.1（http://clojure.org/downloads）
- Groovy 1.8（http://groovy.codehaus.org/Download）
- GPars 0.12（http://gpars.codehaus.org）
- Java SE 1.6（http://www.java.com/en/download）
- JRuby 1.6.2（http://jruby.org/download）
- Scala 2.9.0.1（http://www.scala-lang.org/downloads）

在对不同版本的示例代码的性能进行比较的时候，我会确保比较测试都在相同机器上完成。对于其中大部分示例，我采用的运行环境是搭载 2.8GHz 英特尔双核处理器、4GB 内存并预装了 Mac OS X 10.6.6 和 Java 1.6 update 24 的 MacBook Pro。而其他部分示例我用的则是搭载 8 核 Sunfire 2.33GHz 处理器、8GB 内存和预装 64 位 Windows XP 和 Java 1.6 的电脑。

除非另有说明，否则所有示例都是运行在服务器模式下的“Java HotSpot ™ 64-Bit Server VM”JVM 上。

所有示例全都可以在上面提到过的 Mac 和 Windows 机器上编译通过。

在代码示例的清单中，由于篇幅所限，我并没有列出 import 语句（以及 package 语句）。当你打算亲自运行代码示例的时候，如果你不确定某个类属于哪个 package 的话请别担心，因为我已经将完整的代码放到了网站上，你可以通过本书的网站 http://pragprog.com/titles/vspcom 下载全部示例的源代码。

致谢

有很多人为我完成本书的写作提供了巨大的帮助。这些年来，如果没有那些我所了解和敬重的伟大思想与我的个人思考之间的碰撞所激发出来的灵感，那么本书可能仍然只是

我脑海当中的一个念头而已。

首先我想感谢那些阅读了本书草稿并提供宝贵意见的审稿人，是他们使这本书变得更好。但是你在本书中所发现的任何错误则完全都是我的过失。

感谢 Brian Goetz（@Brian Goetz）、Alex Miller（@puredanger） 和 Jonas Bonér（@jboner）几位审稿人，他们所提出的精辟评论使我受益匪浅。本书几乎每一页都被 Al cherer（@al_scherer）和 Scott Leberknight（@sleberknight）用鹰眼进行了彻底的检查和修订。非常感谢这些先生们。

这里我要特别感谢 Raju Gandhi（@looselytyped）、Ramamurthy Gopalakrishnan、Paul King（@paulk_asert）、Kurt Landrus（@koctya）、Ted Neward（@tedneward）、Chris Richardson（@crichardson）、Andreas Rueger、Nathaniel Schutta（@ntschutta）、Ken Sipe（@kensipe）以及 Matt Stine（@mstine），感谢他们花费大量宝贵时间帮我校对并给予我鼓励。感谢来自 Halloway（@stuarthalloway）的中肯评论，我已经根据他的一些建议对本书进行了改进。

我在各类 NFJS 会议上所进行的关于并发问题的演讲构成了本书的基本框架内容。所以在此感谢 NFJS（@nofluff）主管 Jay Zimmerman 给我在会议上进行演讲的机会，同时还要感谢与会的各位演讲嘉宾和参与者朋友们，与他们的交流和讨论使我深受启发。

在此要特别要感谢 Dave Briccetti（@dcbriccetti）、Frederik De Bleser（@enigmeta）、Andrei Dolganov、Rabea Gransberger、Alex Gout、Simon Sparks、Brian Tarbox、Michael Uren、Dale Visser 和 Tasos Zervos，以及那些花时间阅读本书 beta 版并在本书论坛中给予反馈的程序员。此外，Rabea Gransberger 那些见解深刻的评论、勘误以及观察评论使我受益匪浅。

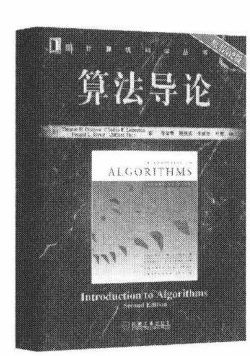
感谢那些我在本书所用到的以及在 JVM 上进行并发应用程序开发所依赖的那些语言和类库的开发者和参与者。

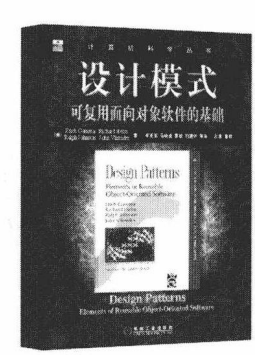
撰写本书的一个额外收获是使我越来越了解审阅本书第 1 版初稿的策划编辑 Steve Peter。他的幽默以及对细节的关注对本书的制作过程起到了极大的帮助。谢谢 Steve。很荣幸与本书的责任编辑 Brian P. Hogan（@bphogan）共同合作。他上手很快，给了我很多鼓励，同时还在许多需要改进的领域提出了很多建设性的意见和建议。谢谢 Brian。

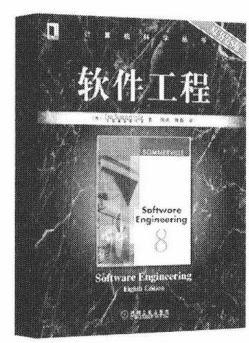
我要感谢整个 Pragmatic Bookshelf 团队一路以来的辛苦付出和给予我的鼓励。感谢 Kim Wimpsett、Susannah Pfalzer（@spfalzer）、Andy Hunt（@pragmaticandy）和 Dave Thomas（@pragdave）的帮助和指导，是他们使得成书过程变得如此有趣。

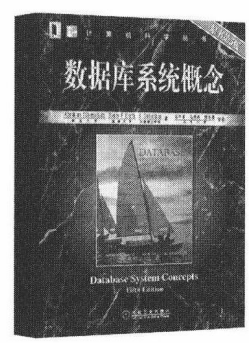
如果没有我妻子的支持，所有这一切都不可能完成，谢谢 Kavitha，感谢她为我付出的耐心与牺牲。此外，我从儿子 karthik 和 Krupa 那里也得到了很多鼓励，谢谢小伙子们，感谢他们总是好奇地问我是否已经把书写完了。最后，希望正在阅读本书的程序员能够将本书所学的知识更好地应用到未来工作中。谢谢！

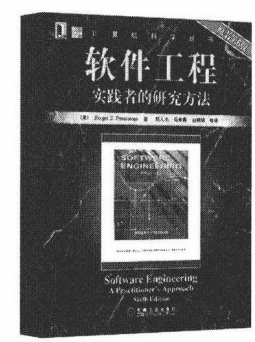
推荐阅读











算法导论（原书第2版）

2006、2007 CSDN、《程序员》杂志评选的十大IT好书之一 算法中的经典权威之作

编译原理（原书第2版）

编译领域无可替代的经典著作，被广大计算机专业人士誉为“龙书”

设计模式：可复用面向对象软件的基础

经典教材 权威之作

软件工程（原书第8版）

最受欢迎的软件工程指南

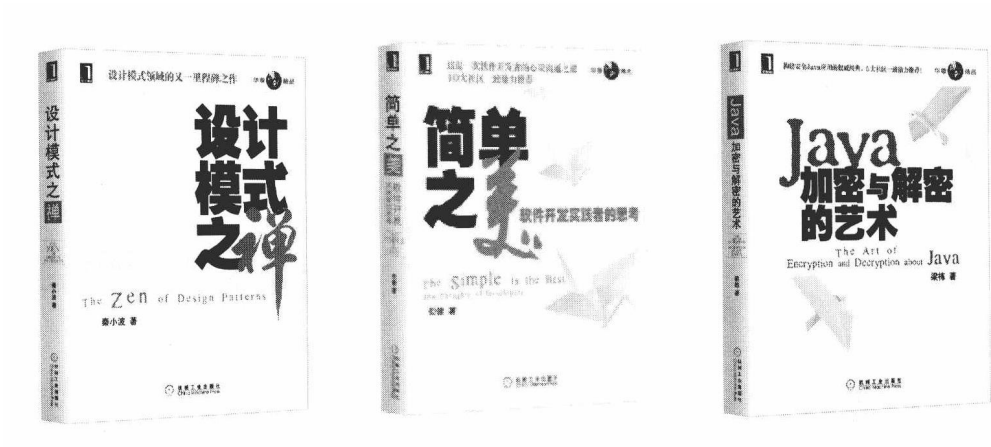
数据库系统概念（原书第5版）

数据库系统方面的经典教材，被美誉为“帆船书”

软件工程：实践者研究方法（原书第6版）

全球上百所大学和学院采用 最受欢迎的软件工程指南

推荐阅读



设计模式之禅

作者：秦小波 ISBN：978-7-111-29544-0 定价：69.00元

禅宗曰："教外别传，不立文字"，禅的境界本不该用文字来描述，言语也道不明白，但为了传道，悟道者仍要藉言语来说明。

何为禅？一种境界，一种体验，一种精神领域的最高修为。何为设计模式？对面向对象思想的深刻理解，对软件设计方法和编码经验的完美总结。

简单之美：软件开发实践者的思考

作者：倪健 ISBN：978-7-111-30103-5 定价：55.00元

多年以来，不管是从事一线的软件开发工作，还是从事管理工作，作者一直在思考这样一个问题：业界有这么多知识财富，可是在实践中真正能够被吸收和应用的却很少，这些知识财富的价值是毋庸置疑的，软件开发人员的热情和渴求也是毋庸置疑的，可问题究竟出在哪里呢？作者最后得出的结论是：这个问题要归结于思想和文化。

Java加密与解密的艺术

作者：梁栋 ISBN：978-7-111-29762-8 定价：69.00元

在如今这个信息化时代，数据是一切应用的核心和基础，有数据存在的地方就会有安全隐患，而密码学则是解决所有安全问题的银弹。

目　录

第一部分　并发策略

第二部分　现代 Java/JDK 并发模型

第三部分　软件事务内存

第四部分 基于角色的并发模型

第五部分 后记

第①章

并发的威力与风险

假设你已经跟老板立下军令状，说你会把新引进的那款强大的多核处理器变成一匹飞驰的骏马，拉着你们开发的应用程序呼啸奔驰。同时，你非常愿意立即着手把这股力量释放出来，并以此开发出响应更快、用户体验更好的应用程序来击败竞争对手。然而这些美好的愿望都被你同事的求助声打断了，他又碰到了一个很麻烦的同步问题。

绝大多数开发人员都对并发又爱又恨。

诚然并发编程是很困难的，但比起能获得的收益而言，所承受的折磨还是值得的。我们以可承受的代价所换来的这种处理能力是父辈们曾经梦寐以求的。只要能充分发挥出多核处理器的多任务并发执行能力，我们就可以创造出更棒的应用程序来。通过提前预判用户行为，我们就能够写出用户体验更好的应用程序。十几年前拖慢程序运行的那些功能现如今已经相当普及，我们不得不采用并发编程的实现方式才能使它们运行得更加流畅。

在本章中，我们将快速回顾一下并发编程技术的来龙去脉，并讨论在实践过程中可能遇到的一些风险。在本章末尾，我们将简单展望一下本书后面将会介绍的那些令人兴奋的并发编程技术。

1.1 线程：程序的执行流程

正如我们所熟知的那样，线程可以看成是进程中的一个执行流程。当我们运行一个程序的时候，其所属进程中至少存在一个执行线程。我们可以通过创建线程的方式来启动新的执行流程以达到并发执行多个任务的效果。此外，我们所使用的类库和框架可能也会根据其需要在后台启动额外的线程。

当多个线程在同一个应用程序或 JVM 实例下运行的时候，实际意味着此时有多个任务或操作在发运行。我们所说的并发程序通常就是指那些使用了多线程或多个并发执行流程的应用程序。

在一个只搭载了单核处理器的系统中，并发任务通常指多工（multiplexed）或多任务（multitasked）执行方式。也就是说，该单核处理器将会不断地在多个执行流程中进行上下文切换，但任意时刻有且只有一个线程（即执行流程）能够被执行。而在一个搭载了多核处理器的系统中，在任意时刻可以有多个执行流程（线程）被执行。其中，可以并发执行的线程数取决于处理器的可用内核数，而应用程序的并发线程数则取决于与该进程相关联的处理器内核数。

1.2 并发的威力

我们对并发编程如此感兴趣主要是基于如下两个原因：即并发编程可以帮助我们提高

程序运行速度、降低响应时间，并以此来改善用户体验。

1.2.1　使应用程序响应速度更快

当我们启动一个应用程序的时候，根据不同应用场景的需要，主线程通常需要承担很多串行任务，如接收一个用户的输入信息、读一个文件、执行一些计算、访问一个 Web 服务、更新数据库、显示一个针对某用户的响应信息等。如果每个操作的耗时都只是毫秒级的话，其实没有必要引入额外的执行流程，单线程就已经足够用了。

然而在大多数实际的应用程序中，很多操作的执行速度并没有这么快，某些计算的执行时间甚至需要几秒到几分钟不等。例如，某些需要从 Web 服务中获取数据的请求可能会遭遇网络延迟，所以执行线程就只好等待对端响应到达后才能继续执行。当单线程的应用程序遇到这种情况时，由于主线程被挂起在某个操作上，所以该应用程序的用户将无法与之进行交互或中断其当前任务的执行。

让我们来考虑一个多线程应用需求的具体实例，并关注多线程对于程序响应方面的影响。在日常生活中，我们经常需要对事件进行计时，如果能有一个带秒表功能的应用程序该多方便啊，下面就让我们动手实现这个程序吧。该程序的使用方法非常简单，即首次点击计时按钮时计时器启动，再次点击按钮时计时结束并输出计时结果。实现该程序的一个比较蠢的方法[⊖]如下所示。（下面的代码片段只是按钮的动作处理函数部分，你可以从本书的网站上下载完整的示例代码。）

introduction/NaiveStopWatch.java

```
//This will not work
public void actionPerformed(final ActionEvent event) {
  if (running) stopCounting(); else startCounting();
}

private void startCounting() {
  startStopButton.setText("Stop");
  running = true;
  for(int count = 0; running; count++) {
    timeLabel.setText(String.format("%d", count));
    try {
      Thread.sleep(1000);
    } catch(InterruptedException ex) {
        throw new RuntimeException(ex);
    }
  }
}

private void stopCounting() {
  running = false;
  startStopButton.setText("Start");
}
```

⊖ 在本例中，为了简单起见我没有记录日志和捕获异常，而只是简单地放过了所有异常。但在你的产品代码中一定要恰当地处理异常。

这个秒表小程序启动之后会弹出一个简单的窗口界面，窗口界面上包含一个 Start 按钮和一个显示为“0”的 label 控件。不幸的是，当我们点击 Start 按钮之后，该按钮上的文字并没有变为“Stop”，同时 label 控件上也没有显示当前已记录的秒数。更糟的是，这个程序甚至没法响应我们的关闭请求（即点击窗口上的关闭按钮无效）。

在 Java GUI 应用程序中，主事件分派线程主要负责通知 UI 相关的事件，并将要执行的操作委派给相应的业务逻辑。当我们点击 Start 按钮之后，主事件分派线程的执行逻辑就走到了事件处理函数[⊖] actionPerformed() 中，而一旦执行到这里，主线程就会被执行计数操作的 startCounting() 函数所挟持。而当我们点击按钮或尝试关闭窗口时，虽然这些事件都会被塞到事件队列中，但由于主线程一直被困在 startCounting() 函数里，所以程序永远无法响应这些事件了。

综上所述，我们所需要做的是启用一个额外的线程或一个 timer（在 Java 的实现中，timer 也会在一个新线程中执行）并把计时任务分派给它，同时释放主事件分派线程。

多线程除了可以降低应用程序响应时间，还有助于提升用户体验。除此之外，应用程序也可以提前预估用户下一步可能执行的操作并提前完成其中关键的动作来提高响应速度，如索引或缓存一些用户所需的数据等。

1.2.2　使程序执行更快

请回头检视一下你之前所写的那些应用程序。看看其中是否存在一些当前是以顺序方式执行但其实可以并发执行的操作。如果有的话，你可以尝试将这些操作分派到独立线程中，来加速这些应用程序的运行。

在实际工作当中其实有很多类型的应用程序可以通过并发的方式来进行提速，包括一些服务程序、计算密集型应用程序和数据分析应用程序等。

1. 服务程序

假设我们接到任务，要开发一个处理多个卖家发货清单的应用程序。该任务要求我们要将某些规则和业务流程应用到每笔订单上，而订单之间的处理顺序不限。在这种场景下，如果我们还是顺序地处理每笔订单，那么系统的处理速度肯定上不去，同时在资源的利用方面（指用多核处理器）也是一种浪费，因此我们的应用程序需要能够并发地处理这些订单。

2. 计算密集型应用程序

我曾经就职于一家化工公司，在那里我主要负责写一些计算冶炼厂各个部门提炼出来的化学制剂成分的应用程序。这些程序的共同特点就是需要大量的计算，所以需要将问题拆分为多个子任务、并发执行各个子任务并最终将各子任务的结果汇总合并，很明显，这种方法可提高效率。也正是因为很多问题可以用分而治之的方法来解决，所以处理程序的执行效率就完全取决于程序员的并发编程能力。

⊖ 在本书中，函数一词对应 Java 语言中的 method，与通常译作“方法”是等价的，请广大读者注意。——译者注

3. 数据分析程序

曾经有人找我帮忙开发一个个人金融理财方面的应用程序，主要功能是从一个 Web 服务器获取当日股票价格和其他详情信息。该应用程序需要为客户展示其总资产净值以及每只股票交易量的详细信息。所以对于一个有钱的客户来说，该应用程序可能需要追踪 100 只不同的股票。在网络使用的高峰时段，从 Web 上查询一只股票的信息可能需要好几秒。而当程序下载好所有相关数据并开始处理的时候，客户可能已经等待好几分钟的时间了。如果假设每个针对 Web 服务的请求的网络时延为 1 ~ 2 秒并且系统可以为应用程序提供足以创建数百个线程的资源和能力的话，我们就可以把从 Web 服务下载信息的请求分派到多个线程，用这种方法可将客户的等待时间缩短至几秒。

1.2.3　并发编程的优点

并发编程可以帮助应用程序提高响应速度、减少等待时间并增加吞吐量。我们可以充分利用多核处理器的性能优势以及多任务并发的方法来提高程序运行效率和响应速度。但正如我们在下一节将讨论的那样，在真正享受这些并发编程所带来的好处之前，我们还需要克服重重困难，前方路上还有很多坑等着我们去填。

1.3　并发的风险

看到这里，你可能会想“只要把任务进行分解并分派到多个线程中执行，程序就能获得更高的吞吐量”。但遗憾的是，绝大多数问题都无法被分解为彼此完全独立的几个子问题。更普遍的情况是，我们可以独立地执行某些操作，但最终还是需要将这些操作所得到的部分结果进行归并才能得到完整的结果。所以线程之间需要能够相互交换彼此的数据，并且有时候某些线程还需要等待其他线程的结果出来之后才能继续运行。于是我们就需要在线程之间进行协调，并由此引出同步和锁等一系列令人头痛的问题。

在开发并发应用程序的时候，我们通常会遇到 3 类问题：饥饿、死锁以及竞争条件。其中前两个问题还算比较易于检测甚至避免，而竞争条件则是一个需要彻底根除的棘手问题。

1.3.1　饥饿和死锁

线程是很容易陷入饥饿状态的。例如，某应用程序正准备执行一个关键任务，但执行该任务之前需要先得到用户的确认，而此时正赶上用户在吃午饭。所以当用户吃得正香的时候，可怜的应用程序就陷入了饥饿状态。当一个线程等待某个需要运行很长时间或永远无法完成的事件发生时，该线程就会陷入饥饿。饥饿的情况可能出现在线程等待用户输入时、等待某些外部事件发生时或等待其他线程释放某个锁时。当线程陷入这种等待状态时，虽然其本身还是活着的，但却什么活也不能干。为了避免线程陷入饥饿，我们可以为其设计一个等待超时的策略，让线程等待有限的时间。即如果所等待的用户输入迟迟不来、事件一直不发生或线程在指定时间内一直没能拿到锁，则该线程将会跳出等待状态并在执行

完超时处理逻辑后再继续运行。

死锁则是指两个或多个线程相互等待对方释放所占用的资源或执行某些动作。然而比饥饿问题更麻烦的是，单凭设定等待超时是无法完全避免死锁的。因为在某些情况下，每个线程因等待超时放弃了其占用的资源之后重做之前的任务，却发现最终又重蹈覆辙，详情请参阅附录 2 中的“哲学家用餐问题”。虽然我们也有一些检测和避免死锁的方法，比如使用 JConsole 这样的工具或令所有线程按照某一特定顺序来获取资源等。但解决死锁问题更好的方案莫过于避免显式加锁以及避免使用可变状态。在本书后面的章节中我们将详细讨论如何使用这些方法。

1.3.2　竞争条件

如果两个线程竞争使用相同的资源或数据，那么我们就将这种情况称为竞争条件（race condition）。竞争条件不仅仅会发生在两个线程同时更改相同数据的场景中，还可能发生在一个线程正在修改某数据而另一个线程同时在读这个数据的时候。竞争条件可能会导致程序行为不可控、执行逻辑异常并产生错误的结果。

竞争条件主要是由下面两大原因造成的：即时下流行的 Just-In-Time(JIT) 编译器的优化以及 Java 内存模型。有关 Java 内存模型相关问题的论述以及该模型对并发的影响，请参阅 Brian Goetzde 的著作《Java Concurrency in Practice》[Goe06]。

下面让我们通过一个非常简单的例子来演示上面所提到的问题。在下面的代码中，主线程首先创建了一个新线程，随即自己 sleep 了 2 秒，最后再将变量 done 的值设为 true。新线程启动之后，会在其内部循环中不停判断 done 变量的值，只要其为 false 就一直循环。下面让我们编译并运行下面的代码并观察其输出结果：

introduction/RaceCondition.java

```
public class RaceCondition {
  private static boolean done;

  public static void main(final String[] args) throws InterruptedException{
    new Thread(
      new Runnable() {
        public void run() {
          int i = 0;
          while(!done) { i++; }
            System.out.println("Done!");
          }
        }
    ).start();

    System.out.println("OS: " + System.getProperty("os.name"));
    Thread.sleep(2000);
    done = true;
    System.out.println("flag done set to true");
  }
}
```

如果我们在 Windows 7（32 位版本）上使用 java RaceCondition 命令来运行上述程序，则可以观察到类似下面的输出结果（每次运行后输出结果的顺序可能不同）

```
OS: Windows 7
flag done set to true
Done!
```

而如果我们在 Mac 上尝试用同样的命令来运行上述程序，则会发现新线程里的循环永远不会停止，即新线程没看到主线程对于 done 变量的修改。具体输出内容如下：

```
OS: Mac OS X
flag done set to true
```

等等，请不要把书扔到一边然后跑去发微博说“Windows 太好用了，Mac 真垃圾”。这个问题其实远比我们从表面现象上所得到的结论复杂。

让我们再试一次，这回我们回到 Windows 系统，执行命令 java –server RaceCondition（即设置程序在 Windows 平台上以 server 模式运行），然后再到 Mac 机上用这个命令 java –d32 RaceCondition（即设置程序在 Mac 上以 client 模式运行）来运行上面的示例。

我们在 Windows 平台上观察到的结果如下所示：

```
OS: Windows 7
flag done set to true
```

然而这次 Mac 平台的输出则与之前不同：

```
OS: Mac OS X
Done!
flag done set to true
```

默认情况下，Java 程序在 32 位 Windows 平台上是以 client 模式运行的，而在 Mac 平台上则是以 server 模式运行的。所以我们的程序在两个平台上的表现是一致的，即以 client 模式运行时可以终止而以 server 模式运行时则无法终止。

当程序以 server 模式运行时，即使主线程将 done 变量的值设为 true，第二个线程也无法看到该变量值的变化。这种现象是由于 Java server JIT 编译器优化所导致的。但是让我们权且先不要抱怨 JIT 编译器，因为它只是任劳任怨地进行代码优化以使程序跑得更快。

从上面的例子中我们所吸取到的教训是，一个烂程序可能在某些情况下工作正常而在另外一些情况下则会失败。

1.3.3　了解可见性：理解内存栅栏

上面那个示例存在的问题是，主线程对其字段 done 所做的变更对新创建出来的线程不可见。造成这种现象的首要原因是，JIT 编译器可能对新线程代码里的 while 循环进行了优化，并因此导致新线程在线程上下文中无法看到变量 done 的变化。此外，新线程可能会只从其寄存器或本地 cache 中读取标记变量 done 的值，而不是每次都跑去速度更慢的内存里进行操作。基于上述原因，新线程就无法看到主线程对其标记变量值的变更了，详情请参阅下文“什么是内存栅栏”。

如果想要快速修复此问题，只需要将变量 done 标记为 volatile 就可以了。具体做法是将

```
private static boolean done;
```

改为：

```
private static volatile boolean done;
```

关键字 volatile 的作用是告知 JIT 编译器不要对被标记变量执行任何可能影响其访问顺序的优化。该关键字警告 JIT 编译器，该变量可能会被某个线程更改，所以任何对该变量的读写访问都需要忽略本地 cache 并直接对内存进行操作。之前我将这个改动称为快速修复，是因为如果我们将所有变量都标记为 volatile 的话，虽然可以完全规避此类问题，但却会使每次变量访问都要跨越内存栅栏并最终导致程序性能下降。此外，在多个字段被多个线程并发访问的场景下，由于针对每个 volatile 字段的访问都是各自独立处理的，并且也无法将这些访问统一协调成一次访问，所以 volatile 关键字无法保证整体操作的原子性。该问题所造成的后果是，线程很可能对某些字段只能看到其中间结果，而对另一些变量则看到的是最终的变更结果。

为了解决这个问题，我们可以屏蔽对变量的直接访问，并将所有访问都引导为通过同步的 getter 和 setter 函数来进行，具体代码如下所示：

```
private static boolean done;
public static synchronized boolean getFlag() { return done; }
public static synchronized void setFlag(boolean flag) { done = flag; }
```

关键字 synchronized 在这里起到了至关重要的作用。由于 synchronized 是为数不多的几个可以令线程在进入和离开同步区块时都跨越内存栅栏的原语之一，所以如果多个线程在相同的实例对象上进行同步并且先申请到对象锁的线程完成了对实例对象的操作，则后面申请到对象锁的线程将肯定可以看到前面完成操作的线程所做的变更。再次提醒，要了解该问题的详情请参阅下文“什么是内存栅栏”。

Joe 问：什么是内存栅栏？

简单来说，内存栅栏（Memory Barrier）就是从本地或工作内存到主存之间的拷贝动作。仅当写操作线程先跨越内存栅栏（参见附录 2 中 Doug Lea 所著《The JSR-133 Cookbook for Compiler Writers》）而读线程后跨越内存栅栏的情况下，写操作线程所做的变更才对其他线程可见。关键字 synchronized 和 volatile 都强制规定了所有的变更必须全局可见，该特性有助于跨越内存边界动作的发生，无论是有意为之还是无心插柳。

在程序运行过程中，所有的变更会先在寄存器或本地 cache 中完成，然后才会被拷贝到主存以跨越内存栅栏。此种跨越序列或顺序称为 happens-before，详情请参阅附录 2 中的“Java 内存模型”以及 Brian Goetz 的著作《Java Concurrency in Practice》[Goe06]。

写操作必须要 happens-before 读操作，即写线程需要在所有读线程跨越内存栅栏之前完成自己的跨越动作，其所做的变更才能对其他线程可见。

Java 并发 API 中很多操作都隐含有跨越内存栅栏的含义：volatile、synchronized、Thread 中的函数如 start() 和 interrupt()、ExecutorService 中的函数以及像 CountDownLatch 这样的同步工具类等。

1.3.4　规避共享可变性

如果我们在本该使用 volatile 或 synchronized 的地方忘了做同步的话，则可能会导致不可预测的程序行为。但其实核心问题并非我们忘记做同步这件事本身，而是因为我们正在处理共享可变性。

在过去的工作经历中，我们已经习惯于使用可变性来开发 Java 应用程序，即通过更改其字段的方式来创建和变更一个对象状态。然而，像 Joshua Bloch 的《Effective Java》[Blo08] 这类伟大的著作则建议我们尝试将不可变性引入到并发编程当中，因为不可变性可以帮助我们从根本上规避上面所提到的那些难题。

虽然很容易被误用以致程序逻辑出错，但可变性本身并非全无优点。共享是一件好事，毕竟妈妈以前一直是这样教导我们的。虽然两件事分开看都还是不错的，但掺和在一起就不是那么回事了。

假设我们在程序中定义了一个非 final（可变）的字段，每当一个线程更改了该字段的值，我们都需要考虑是应该将变更后的值写回内存还是应该将其保留在寄存器 /cache 中。而每当读取该字段的时候，都需要关心所读到的内容究竟是最新的有效值还是 cache 中旧的过期结果。同时，还需要确保针对该变量的变更是原子的，即其他线程会不看到变更过程的中间结果。此外，我们还需要为防止多个线程同时更改同一数据而操心。

对于一个涉及可变性的应用程序而言，每个针对共享可变状态的独立访问我们都需要验证其正确性。只要其中某次访问出问题，则整个程序就会出问题。由此可见，处理可变性是一项艰巨的任务，因为只要并发处理的逻辑中存在一行有问题的代码，那么整个应用程序就可能会挂掉。事实上，有相当数量的并发 Java 应用程序都存在这类问题，只是我们不知道而已。

如果我们在程序中定义了一个指向某不可变实体[⊖]的 final（不可变）字段并让多个线程同时访问该字段，则这种形式的共享就不会有任何隐患。任何线程都可以读取该实体的值，并且这个值就是该实体保存在其 cache 中的值的拷贝。由于值是不可变的，所以后续对该值的访问只要从本地 cache 中获取就行了，而我们则还可以坐享其成，从中获得更好的性能。

共享可变性是魔鬼，千万别碰它！

如果什么都不能改，那怎样才能让应用程序干活啊？话虽如此，但我们还是需要围绕不可变性来设计应用程序。其中一种方案是对可变状态进行严密的封装并只共享不可变数据。而另一种备选方案则是由纯函数式语言所提供的，即除了可以使用函数组合之外，将程序中一切事物都设定为不可变的。值得注意的是，在方案二中，如果我们想要将一个不可变状态转换成另一个不可变状态，则需要利用一系列的转换动作才能完成。除了上述两种方法之外，我们还可以使用一个可以监控所有变化并在出现任何违规行为时对我们发出警告的库来规避可变性所带来的问题。在后面的章节中，我们将先引入一些有问题的示例，

⊖ 例如，像 Java 中的 String、Integer 以及 Long 这样的实体就是不可变的，而 StringBuilder 和 ArrayList 则是可变的。

然后再用并发方法将问题解决，以这种形式为你逐一介绍这些技术。

1.4　小结

无论我们正在开发一个交互式客户端桌面应用程序还是一个后台高性能服务，无论是从充分利用硬件性能增长的角度还是多核处理器逐渐普及的现状来考虑，并发对我们的编程工作都将起到至关重要的作用。通过使用并发，我们就可以更有效地提升用户体验，提高程序响应速度，并使程序在那些拥有强劲性能的机器上跑得更快。传统的基于共享可变性处理的 JVM 并发编程模型存在很多问题。在创建完线程之后，我们需要努力避免线程陷入饥饿、死锁以及竞争条件当中，而这些都是极难定位并极易导致错误发生的问题。通过消除共享可变性，我们就可以从根本上解决上述这些问题，而不可变性的应用程序将会使并发编程更简单、安全和有趣。在本书后面的章节，我们将会学习不可变性的具体实现方式。

接下来，我们将讨论几种用于确定线程数量和分解任务的方法。

第一部分

并发策略

第②章

分工原则

期待已久的多核处理器明天就要到货了，我想你已经迫不及待地想要看看之前写的程序在多核处理器上的运行效果。由于那个程序已经在搭载单核处理器的机器上跑了很久，所以你十分期待该程序在新机器上可以跑得更快。程序执行速度与处理器的核心数成正比？还是更快？抑或性能没啥改善？还是用了多核处理器之后性能反倒变得更差？在经过大量研究和实践之后，我已经找到可以充分发挥多核处理器的能力并达到预期的性能提升的方法。

当我第一次将代码放到多核处理器上跑的时候，我的脸色超级难看，因为程序性能比我预期的差太多了。我当时很郁闷，多核处理器跑程序为啥比单核还慢啊？但这已经是几年前的事了。在这段时间里，我一路摸爬滚打积累了不少经验。现在我对并发问题的直觉变得更加敏锐，并且掌握了一些门道，所以在本章我想把这些经验分享给大家。

2.1 从顺序到并发

我们不能期望一个单线程的程序在多核处理器上能跑出比在单核处理器上更好的效果。为了提升性能，我们需要将问题分解成多个任务，同时将这些任务并发执行。但是程序并不能像分解任务那样拆分，也不是说我们有多少任务就要拆出多少线程。

过去我曾参与过计算密集型科学计算应用程序的开发，也负责过涉及文件、数据库和Web服务调用的IO密集型商业应用程序的研发。我发现这两种类型的应用程序是有本质区别的，所以二者在并发的使用方面也截然不同。

本章我们将研究两种类型的应用程序。首先是一个IO密集型应用程序，用来为某富有

的用户计算其资产净值。第二个应用程序的主要功能则是计算一个给定区间内所有素数的数量，这是一个很简单却相当有用的并发计算密集型程序的实例。这两个应用程序将帮助我们学习如何估计需要创建多少个线程、如何分解问题以及如何估算性能提升的程度。

分而治之

如果我们有数百只待处理的股票，那么每次一只地逐个处理是最简单的方法，但同时我们也会因为工作低效而被老板炒掉。因为当我们的应用程序吭哧吭哧地喘着粗气逐个处理每只股票的时候，用户已经在旁边等得火冒三丈了。

为了加快程序处理速度，我们需要将问题分解成若干个并发执行的任务。接下来我们需要创建任务并将其委派给线程，以便使它们可以并发地执行。但是有一个很大的问题摆在我们面前，即我们希望尽可能多地创建任务，但由于资源所限我们又不能创建过多的线程。

1. 确定线程数

为了解决上述难题，我们希望至少可以创建处理器核心数那么多个线程。这就保证了有尽可能多地处理器核心可以投入到解决问题的工作中去。通过下面的代码，我们可以很容易地获取到系统可用的处理器核心数[⊖]：

```
Runtime.getRuntime().availableProcessors();
```

所以，应用程序的最小线程数应该等于可用的处理器核数。如果所有的任务都是计算密集型的，则创建处理器可用核心数那么多个线程就可以了。在这种情况下，创建更多的线程对程序性能而言反而是不利的。因为当有多个任务处于就绪状态时，处理器核心需要在线程间频繁进行上下文切换，而这种切换对程序性能损耗较大。但如果任务都是 IO 密集型的，那么我们就需要开更多的线程来提高性能。

当一个任务执行 IO 操作时，其线程将被阻塞，于是处理器可以立即进行上下文切换以便处理其他就绪线程。如果我们只有处理器可用核心数那么多个线程的话，则即使有待执行的任务也无法处理，因为我们已经拿不出更多的线程供处理器调度了。

如果任务有 50% 的时间处于阻塞状态，则程序所需线程数为处理器可用核心数的两倍。如果任务被阻塞的时间少于 50%，即这些任务是计算密集型的，则程序所需线程数将随之减少，但最少也不应低于处理器的核心数。如果任务被阻塞的时间大于执行时间，即该任务是 IO 密集型的，我们就需要创建比处理器核心数大几倍数量的线程。

我们可以计算出程序所需线程的总数，总结如下：

线程数 =CPU 可用核心数 /(1 – 阻塞系数)，其中阻塞系数的取值在 0 和 1 之间。

计算密集型任务的阻塞系数为 0，而 IO 密集型任务的阻塞系数则接近 1。一个完全阻塞的任务是注定要挂掉的，所以我们无须担心阻塞系数会达到 1。

为了更好地确定程序所需线程数，我们需要知道下面两个关键参数：

- 处理器可用核心数

⊖ availableProcessors() 函数可以获取 JVM 可使用的逻辑处理器数量。

• 任务的阻塞系数

第一个参数很容易确定，我们甚至可以用之前的方法在运行时查到这个值。但确定阻塞系数就稍微困难一些。我们可以先试着猜测，抑或采用一些性能分析工具或 java.lang.management API 来确定线程花在系统 /IO 操作上的时间与 CPU 密集任务所耗时间的比值。

2. 确定任务的数量

前面我们已经知道如何计算并发应用程序所需的线程数量，现在我们需要确定如何分解问题。每一个子任务都将并发地运行，所以我们脑中第一个念头可能是希望子任务的数量和线程数保持一致。虽然想法不错，但还远远不够，因为这种做法忽略了待解决问题本身的特性。

在计算资产净值的那个程序中，获取每一只股票价格的代价是相同的，所以我们只需要将股票分成和线程数量一样多的组，每个组委派一个线程来处理就可以了。

但是在计算素数数量的程序中，计算一个数是否为素数的代价是各不相同的。偶数可以被很快识别出不是素数，而识别较大的素数要比识别较小的素数更耗时。所以将数的计算范围分成线程数那么多个子区间并分别处理并不会带来显著的性能提升。这种做法会使得某些任务比其他任务完成得快，从而造成处理器核心使用上的不均衡。

换句话说，我们希望各个子任务的工作量是均匀分布的。所以我们可能会花大量精力和时间来进行问题分解，以便使子任务的负载保持均匀分布。但这会带来两个问题，首先，要做到这点是十分困难的，可能会使我们殚精竭虑。其次，将问题分解成彼此相等的子问题，并将工作量等量均摊到各个线程之间的代码将会非常复杂。

事实证明，在解决问题的过程中使处理器一直保持忙碌状态比将负载均摊到每个子任务要实惠得多。从处理问题的角度来看，我们需要保证：只要还有待完成的任务，就不可能有空闲的处理器核心。所以，与其斤斤计较于如何将负载平摊到每个子任务上，不如将任务拆得比线程数多，以使处理器一直不停工作来得更有效。我们应该尽可能地对问题进行拆分，以产生足够多的工作供所有处理器可用核心来执行。

2.2 在 IO 密集型应用程序中使用并发技术

IO 密集型应用程序的阻塞系数一般都很大，所以程序开的线程数通常需要超过处理器可用核心数。

现在让我们一起来构建之前提到的那个处理金融事务的应用程序，该程序的作用是：计算在任意给定时间某（富有的）用户所持股份的总资产净值。让我们先来为一位持有四十只股票的用户提供服务。假定我们已经知道了该用户所持每只股票的股票代码以及股数，接着我们需要通过 Web 查询每只股票的价位。下面让我们来看一下如何写这段计算总资产净值的代码。

2.2.1 计算总资产净值的顺序算法

首先我们需要获取每只股票的价格。幸运的是，Yahoo 提供了我们所需的历史数据。下

面的代码的主要功能是通过 Yahoo 金融 Web 服务接口获取某只股票前一个交易日的收盘价。

divideAndConquer/YahooFinance.java

```
public class YahooFinance {
  public static double getPrice(final String ticker) throws IOException {
    final URL url =
      new URL("http://ichart.finance.yahoo.com/table.csv?s=" + ticker);

    final BufferedReader reader = new BufferedReader(
      new InputStreamReader(url.openStream()));

    //Date,Open,High,Low,Close,Volume,Adj Close
    //2011-03-17,336.83,339.61,330.66,334.64,23519400,334.64
    final String discardHeader = reader.readLine();
    final String data = reader.readLine();
    final String[] dataItems = data.split(",");
    final double priceIsTheLastValue =
      Double.valueOf(dataItems[dataItems.length - 1]);
    return priceIsTheLastValue;
  }
}
```

我们发送查询请求到 http://ichart.finance.yahoo.com，然后从返回值中解析出某股前一交易日的收盘价。

接下来，我们逐一累加用户所持股票的价格并最终显示该客户的总资产净值。此外，我们还打印出了整个操作的总耗时。

divideAndConquer/AbstractNAV.java

```
public abstract class AbstractNAV {
  public static Map<String, Integer> readTickers() throws IOException {
    final BufferedReader reader =
      new BufferedReader(new FileReader("stocks.txt"));

    final Map<String, Integer> stocks = new HashMap<String, Integer>();

    String stockInfo = null;
    while((stockInfo = reader.readLine()) != null) {
      final String[] stockInfoData = stockInfo.split(",");
      final String stockTicker = stockInfoData[0];
      final Integer quantity = Integer.valueOf(stockInfoData[1]);

      stocks.put(stockTicker, quantity);
    }

    return stocks;
  }

  public void timeAndComputeValue()
    throws ExecutionException, InterruptedException, IOException {
    final long start = System.nanoTime();

    final Map<String, Integer> stocks = readTickers();
    final double nav = computeNetAssetValue(stocks);
```

```
    final long end = System.nanoTime();

    final String value = new DecimalFormat("$##,##0.00").format(nav);
    System.out.println("Your net asset value is " + value);
    System.out.println("Time (seconds) taken " + (end - start)/1.0e9);
  }

  public abstract double computeNetAssetValue(
    final Map<String, Integer> stocks)
    throws ExecutionException, InterruptedException, IOException;
}
```

AbstractNAV 类的 readTickers() 函数从一个叫做 stocks.txt 的文件中读取每只股票的股票代码和股数，其中的部分内容如下：

```
AAPL,2505
AMGN,3406
AMZN,9354
BAC,9839
BMY,5099
...
```

timeAndComputeValue() 函数计算了操作的整体耗时，并调用了抽象函数 computeNetAssetvValue()，该函数将会由 AbstractNAV 的派生类来实现。然后，程序输出客户的总资产净值以及计算所需时间。

最终，我们需要调用 Yahoo 金融频道网站的接口来计算总资产净值。这里我们用顺序的方法来实现这个功能：

divideAndConquer/SequentialNAV.java

```
public class SequentialNAV extends AbstractNAV {
  public double computeNetAssetValue(
    final Map<String, Integer> stocks) throws IOException {
    double netAssetValue = 0.0;
    for(String ticker : stocks.keySet()) {
      netAssetValue += stocks.get(ticker) * YahooFinance.getPrice(ticker);
    }
    return netAssetValue;
  }

  public static void main(final String[] args)
    throws ExecutionException, IOException, InterruptedException {
    new SequentialNAV().timeAndComputeValue();
  }
}
```

让我们运行 SequentialNAV 这段代码并观察其输出结果：

```
Your net asset value is $13,661,010.17
Time (seconds) taken 19.776223
```

虽然我们帮客户算出了其总资产净值，但客户并不是很满意。这种不满情绪部分是由于市场环境造成的，但实际上主要还是因为我们算得太慢导致客户等了过长的时间。由于有网络延时，上面的程序在我的机器上运行了近 20 秒[⊖]，这还仅仅只是获取 40 只股票的计

⊖ 超过几秒的延时就会让用户感觉好像过了一辈子那么长的时间。

算结果。我确信将应用程序并行化将有助于提升执行效率并使客户满意。

2.2.2　确定线程数和资产净值的子集划分

上述应用程序在执行计算操作方面只用了很少的时间，而将大部分时间都花在等待Web响应上面。但是我们没有理由等前一个请求的响应回来之后才能发下一个请求，所以这个应用程序是进行并发改造的最佳选择：并发改造后程序性能将会有质的飞跃。

在前面的例子中，我们只计算了40只股票的总资产净值，然而现实生活中我们遇到的计算量可能比这个数要多得多，甚至有可能一次要算几百只股票。我们首先需要确定将股票总集拆分之后的子集数和计算所需的线程数。Web服务（在本例中，特指Yahoo金融频道网站）通常能接受和处理大量并发请求[⊖]，所以客户端的线程数和服务端的请求限制保持一致就可以了。由于对Web服务的请求大部分时间都花在等待服务器响应上了，所以阻塞系数会相当高，因此程序需要开的线程数可能是处理器核心数的若干倍。假设阻塞系数是0.9，即每个任务90%的时间处于阻塞状态而只有10%的时间在干活，则在双核处理器上我们就需要开20个线程（使用第2.1节的公式计算）。如果有很多只股票要处理的话，我们可以在8核处理器上开到80个线程来处理该任务。

至于拆分之后的股票子集数，由于在本例中计算每只股票的工作负载都是相同的，所以我们应该尽可能多地将股票总集进行拆分并将其调度给空闲线程执行。

让我们把上面的例子代码改造成并发的执行方式，然后研究使用多线程会带来怎样的影响以及如何进行代码分解。

2.2.3　资产净值的并发计算

目前我们主要面临两大问题。首先，我们需要调度线程来完成子问题的求解。其次，每个股票子集的计算结果都只是总资产净值的一部分，我们需要把它们合理地累加起来才能得到最终的结果。

就本问题而言，我们会将股票总集尽可能多地拆分成多个子集并维护一个线程池来调度这些子集。由于线程池可以很好地进行线程生命周期和资源的管理、有效减少线程创建和销毁的开销并能快速响应调度任务的需求，所以我们最好还是使用线程池而不是自己创建和管理一些单个的线程。

作为一个Java程序员，我们过去经常会用到Thread和Synchronized关键字。然而随着Java 5的发布，我们又有了一些更好的替代品。在java.util.concurrent包里的新一代并发API提供了更为强大的并发和线程同步功能。

⊖ 为了避免拒绝服务攻击（并推销其增值服务），Web服务可能会限制客户端发送的并发请求数。当你把并发请求数增加到50的时候，就将遇到Yahoo金融频道对客户端的请求限制。

Joe 问：Java 的旧线程 API 现在还有用武之地吗？

回答：旧的线程 API 在功能上有很多缺陷。比如，由于线程不允许重新启动，所以一旦线程执行完了我们就必须把 Thread 类的实例丢掉。于是为了同时处理多个任务，我们通常需要不断开新线程而不是复用之前已经创建好的。如果我们想要调度一个线程去处理多个任务，就不得不写很多额外的管理代码。这样既不高效也不具可扩展性。

像 wait() 和 notify() 这样的函数都需要进行线程间同步，我们很难判断何时才是使用它们进行线程间通信的正确时机。而 join() 函数则使我们的注意力都集中在处理线程消亡的逻辑上，从而忽视了对即将结束的任务的处理（任务完成并不代表线程消亡，反之亦然。——译者注）。

此外，synchronized 关键字的粒度太粗。该关键字没有提供当线程没获取到锁的情况下的超时逻辑，而且也不允许对互斥区域的并发读。此外，如果我们用了 synchronized，那么关于线程安全方面的单元测试也很难进行。

现在，由 Doug Lea 等人牵头开发的 java.util.concurrent 包中下一代并发 API 已经很好地替代了旧线程 API。

- 以前代码中使用 Thread 类及其方法的地方，现在都可以考虑用 ExecutorService 类及其相关类来替换。
- 如果想要更好地控制加锁的过程，则最好使用 Lock 接口及其方法。
- 以前代码中使用 wait/notify 方法的地方，现在都可以使用像 CyclicBarrier 和 CountdownLatch 这样的同步工具来替换。

在现代的并发 API 中，Executors 类被用作创建不同类型线程池的工厂类，其创建的线程池都可以用 ExecutorService 接口进行管理。我们甚至可以用它来创建所有任务都在一个线程内逐个调度执行的单线程的线程池。而固定大小的线程池则允许我们配置线程池的大小，并能调度可用线程并发执行我们丢给它的任务。如果任务数大于线程数，则所有任务将排队执行，且只要有可用线程则等待队列中的任务就可以立即被执行。此外，带缓存的线程池会按需创建线程，并尽可能地复用已经创建好的线程。如果某个线程空闲超过 1 分钟，则该闲置线程将被关停。

固定大小的线程池与我们在计算总资产净值应用程序中的需求非常契合。根据处理器核心数以及之前假定的阻塞系数，我们就可以确定线程池的大小。线程池中每个线程都将分别执行其对应股票子集的计算任务。在之前的例子中，我们需要计算 40 只股票。如果我们创建 20 个线程（在双核处理器上），则有半数任务可以立即被调度执行。剩余的一半任务被放入等待队列，一旦有空闲线程就可以立即被执行。下面让我们写一段并发获取股价的代码，其中对于股票总集的划分需要稍微多花点工夫。

divideAndConquer/ConcurrentNAV.java

```
Line 1  public class ConcurrentNAV extends AbstractNAV {
          public double computeNetAssetValue(final Map<String, Integer> stocks)
```

```
    throws InterruptedException, ExecutionException {
  final int numberOfCores = Runtime.getRuntime().availableProcessors();
  final double blockingCoefficient = 0.9;
  final int poolSize = (int)(numberOfCores / (1 - blockingCoefficient));

  System.out.println("Number of Cores available is " + numberOfCores);
  System.out.println("Pool size is " + poolSize);
  final List<Callable<Double>> partitions =
    new ArrayList<Callable<Double>>();
  for(final String ticker : stocks.keySet()) {
    partitions.add(new Callable<Double>() {
      public Double call() throws Exception {
        return stocks.get(ticker) * YahooFinance.getPrice(ticker);
      }
    });
  }

  final ExecutorService executorPool =
    Executors.newFixedThreadPool(poolSize);
  final List<Future<Double>> valueOfStocks =
    executorPool.invokeAll(partitions, 10000, TimeUnit.SECONDS);

  double netAssetValue = 0.0;
  for(final Future<Double> valueOfAStock : valueOfStocks)
    netAssetValue += valueOfAStock.get();

  executorPool.shutdown();
  return netAssetValue;
}

public static void main(final String[] args)
  throws ExecutionException, InterruptedException, IOException {
  new ConcurrentNAV().timeAndComputeValue();
}
}
```

在 computeNetAssetValue() 函数中我们根据假设的阻塞系数和处理器核心数（Runtime 的 availableProcessor() 函数提供了确切的数值）确定了线程池的大小。然后我们将获取每只股票价格的任务逻辑放在 Callable 接口的匿名代码块中。Callable 接口提供了 call() 函数，该函数的返回值是 Callable 接口的泛型定义的实例（在本例中是 Double 类型）。紧接着我们在固定大小的线程池上调用 invokeAll() 函数来调度各个任务开始执行。其中，executorPool 负责尽可能多地并发执行任务。如果划分的子任务数超过线程池大小，则多出来的任务将进入等待队列按顺序轮流执行。由于各个子任务的执行是并发且异步的，所以负责分派任务的主线程无法立即获知计算结果。当所有任务执行完毕之后，invokeAll() 函数将返回一个 Future 对象的集合[⊖]。我们将每个对象的结果累加起来就可以得到总资产净

⊖ 如果不想等到所有任务完成才拿到数据，而是希望要每完成一个任务就返回一个结果的话，可以使用 CompletionService。

值。让我们看一下该程序的并发版本的执行情况：

```
Number of Cores available is 2
Pool size is 20
Your net asset value is $13,661,010.17
Time (seconds) taken 0.967484
```

与之前的顺序执行需要 20 秒相比，并发执行的时间连 1 秒都不到。除了上例中的设定之外，我们可以尝试通过调整阻塞系数来改变线程池大小，然后观察程序的执行时间是否发生变化。我们还可以尝试计算不同数量的股票，并观察程序的顺序执行版和并发执行版之间的速度差异。

2.2.4　隔离可变性

在本问题的处理过程中，ExecutorService 几乎解决了所有的同步问题，我们可以放心地将任务委托给它并从主线程中接收执行结果。在之前的代码中，唯一的可变变量是在代码第 25 行定义的 netAssetValue，唯一修改此变量的地方是代码第 27 行。由于这一修改仅发生在主线程中，所以这里我们做到了隔离可变性而不是共享可变性。同时，因为没有共享状态，所以本例中无需进行任何同步操作。此外，通过使用 Future 类，我们可以安全地在主线程中拿到各个线程从 Web 接口取回的结果。

本例所使用的方法有一个限制。在代码第 26 行，我们循环遍历所有 Future 对象，所以每个循环我们只能取到一个子任务的结果，且所取到的子任务的顺序是与我们创建 / 调度子任务的顺序是一致的。所以即使排在后面的子任务先完成了，我们也无法对其进行处理，除非前一个子任务的结果已经处理完毕。在本例中这可能不是什么大问题，然而如果我们在收到外部接口的响应之后还要进行大量计算的话，则最好能够做到只要有可用结果就立即处理，而不是等到所有任务都结束才一并处理。JDK 中自带的 CompletionService 类可以帮助我们实现这一功能。稍后我们会重新回顾这一议题并研究一些备选方案。下面让我们转换一下脑筋，一起分析一下如何通过并发方法来加速程序运行。

2.3　并发方法对 IO 密集型应用程序的加速效果

IO 密集型应用程序的特性使得程序即使在处理器核心数很少的情况下也可以实现相当好的并发度。当一个任务阻塞在 IO 操作上时，我们可以立即切换执行其他任务或启动其他 IO 操作请求。我们估计，对于计算股票总资产净值的程序来说，在一个双核 CPU 机器上开 20 个线程就足够了。让我们分析一下在一个双核处理器上开不同的线程数（从 1 到 40）对程序性能的影响。由于子任务总数为 40，所以开超过 40 个线程是毫无意义的。我们可以从图 2-1 中观察到，随着线程数的增加，程序执行速度提升的具体效果。

从图中我们可以看到，自线程池大小达到 20 之后，曲线逐渐趋于水平。这说明我们之前的估计是正确的，即开更多的线程对程序的性能提升毫无帮助。

这个应用程序是展示并发效果最佳实例，各子任务的工作负载都大致相同，并且还伴有大量阻塞。正是由于从 Web 接口获取数据导致的延时，使得多线程并发技术在这里产生

了相当不错的执行加速效果。我们能够通过增加线程数量来获得更好的加速效果。然而，并非所有的问题都可以通过类似的方法来提速，正如我们下面要看的那样。

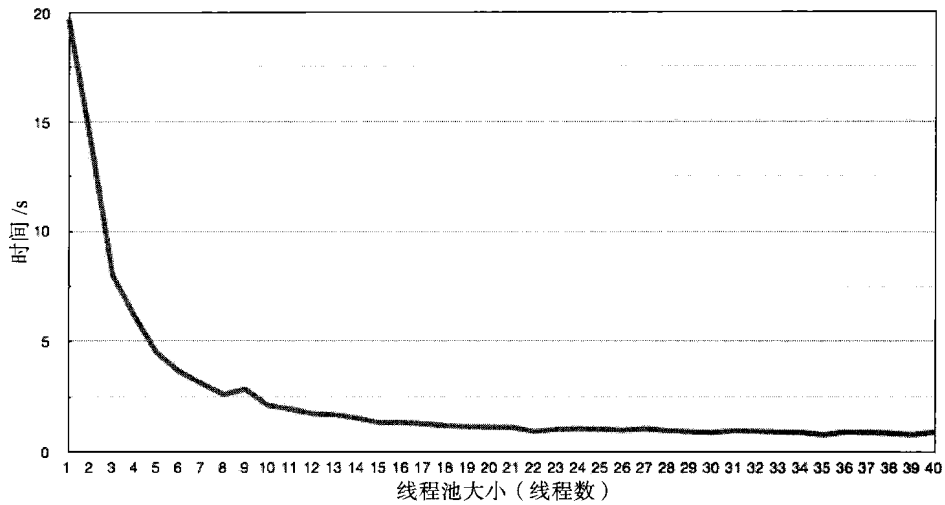


图 2-1　线程池大小增加时程序执行速度提升的效果图

2.4　在计算密集型应用程序中使用并发技术

正如我们本节将看到的那样，相对于 IO 密集型应用程序，处理器核心数对计算密集型应用程序的加速效果影响更大。虽然使用的示例非常简单，但我们将会从中发现工作负载的不均匀分配对加速效果产生的影响。

下面我们将写一个计算 [1, 1 千万] 中素数数量的程序。让我们先用顺序方法，然后再改用并发方法来解决此问题。

2.4.1　素数数量的顺序计算方法

让我们先从抽象类 AbstractPrimeFinder 开始，该类的作用是将一些公共方法聚合在一起解决问题。isPrime() 函数用来判断一个给定的数字是否是素数，而 countPrimesInRange() 函数则使用 isPrime() 来计算一个指定区间内所有素数的数量。最后，timeAndCompute() 函数将会输出计算过程的整体耗时。

divideAndConquer/AbstractPrimeFinder.java

```
public abstract class AbstractPrimeFinder {
  public boolean isPrime(final int number) {
    if (number <= 1) return false;

    for(int i = 2; i <= Math.sqrt(number); i++)
      if (number % i == 0) return false;
```

```
    return true;
  }

  public int countPrimesInRange(final int lower, final int upper) {
    int total = 0;

    for(int i = lower; i <= upper; i++)
      if (isPrime(i)) total++;

    return total;
  }
  public void timeAndCompute(final int number) {
    final long start = System.nanoTime();

    final long numberOfPrimes = countPrimes(number);

    final long end = System.nanoTime();

    System.out.printf("Number of primes under %d is %d\n",
      number, numberOfPrimes);
    System.out.println("Time (seconds) taken is " + (end - start)/1.0e9);
  }

  public abstract int countPrimes(final int number);
}
```

接下来我们将依次调用上面的代码来完成计算。为了顺序地计算 [1,number] 中素数的数量，我们只需简单地将 1 和 number 作为参数传给 countPrimesInRange() 函数即可。

divideAndConquer/SequentialPrimeFinder.java

```
public class SequentialPrimeFinder extends AbstractPrimeFinder {
  public int countPrimes(final int number) {
    return countPrimesInRange(1, number);
  }

  public static void main(final String[] args) {
    new SequentialPrimeFinder().timeAndCompute(Integer.parseInt(args[0]));
  }
}
```

在此我们设定 number 为 10000000，并观察程序运行时间。

```
Number of primes under 10000000 is 664579
Time (seconds) taken is 6.544368
```

在我的双核处理器电脑上用 JDK 的 server 模式跑这个程序大约花了 6 秒多一点，如果用 Client 模式可能还会更慢一些。下面让我们看看将程序改造成并发模式能使程序运行速度有怎样的提升。

2.4.2 素数数量的并发计算方法

由于这是一个计算密集型任务，所以开大量的线程对整体运行速度而言毫无助益。因为阻塞系数为 0，所以根据 2.1 节中的公式计算出来的线程数等于处理器核心数（即为 2）。

线程数超过该值不但不会对程序执行速度有任何提升，反而会导致性能下降。这是因为挂起一个非阻塞任务去执行另一个非阻塞任务对本例来说意义不大，反而会增加上下文切换开销。当所有 CPU 核心都处于忙碌状态时，我们最好让前一个任务执行完再执行下一个。下面让我们先把代码改成并发式的，开两个线程并将问题拆成两个子任务，然后再来看一下运行效果。我们现在暂时回避将问题拆成多少子任务才合理这个问题，等我们取得一些进展之后再回过头来解决它。

计算素数数量的并发版代码和计算总资产净值的并发版代码在结构上十分相似。与之前线程数和任务数都需要事先算出来不同，这里我们将这两个参数都设为 2。下面就是计算素数数量的并发版代码。

divideAndConquer/ConcurrentPrimeFinder.java

```
line 1  public class ConcurrentPrimeFinder extends AbstractPrimeFinder {
        private final int poolSize;
        private final int numberOfParts;

    5   public ConcurrentPrimeFinder(final int thePoolSize,
          final int theNumberOfParts) {
          poolSize = thePoolSize;
          numberOfParts = theNumberOfParts;
        }
   10   public int countPrimes(final int number) {
          int count = 0;
          try {
            final List<Callable<Integer>> partitions =
              new ArrayList<Callable<Integer>>();
   15       final int chunksPerPartition = number / numberOfParts;
            for(int i = 0; i < numberOfParts; i++) {
              final int lower = (i * chunksPerPartition) + 1;
              final int upper =
                (i == numberOfParts - 1) ? number
   20             : lower + chunksPerPartition - 1;
              partitions.add(new Callable<Integer>() {
                public Integer call() {
                  return countPrimesInRange(lower, upper);
                }
   25         });
            }
            final ExecutorService executorPool =
              Executors.newFixedThreadPool(poolSize);
            final List<Future<Integer>> resultFromParts =
   30         executorPool.invokeAll(partitions, 10000, TimeUnit.SECONDS);
            executorPool.shutdown();
            for(final Future<Integer> result : resultFromParts)
              count += result.get();
          } catch(Exception ex) { throw new RuntimeException(ex); }
   35     return count;
        }
        public static void main(final String[] args) {
          if (args.length < 3)
            System.out.println("Usage: number poolsize numberOfParts");
   40     else
```

```
        new ConcurrentPrimeFinder(
            Integer.parseInt(args[1]), Integer.parseInt(args[2]))
          .timeAndCompute(Integer.parseInt(args[0]));
    }
}
```

我们将区间 [1, number] 划分成指定数量（本例为 2）个子区间，其中每个区间所含数字个数基本相同（代码 13 到 26 行）。然后我们将计算每个子区间中素数数量的任务委托给各个 Callable 实例去执行。现在有了子区间，就可以对它们的执行进行调度。我们将使用一个线程池并将各子区间的计算任务交给 ExecutorService 去执行（代码第 28 到 31 行）。在代码第 33 行的最终步骤里，我们把各个子区间的计算结果累加起来即可获得给定区间中的素数总量。

现在到了验证效果的时候了。我的电脑是双核的，所以如果我开了两个线程并将整体区间划为两个子区间，我希望程序性能提高两倍（即并发版代码的执行时间应是顺序版代码执行时间的一半）。下面让我们来看一下程序对 1 千万个数字的执行效果，其中线程池大小和子区间数量都是 2，执行程序的命令为：java ConcurrentPrimeFinder 10000000 2 2。

```
Number of primes under 10000000 is 664579
Time (seconds) taken is 4.236507
```

从结果中我们可以看出，并发版代码算得的素数数量与顺序版代码结果一致，这是件好事，起码说明两个版本的代码都能正常工作。然而我还期望在双核电脑上程序运行时间降到 3 秒左右，但是实际结果却是 4 秒多，差不多提速了 1.5 倍而不是之前期望的 2 倍。所以现在是时候回顾一下我们之前所做的事情，来看看到底我们在哪些点上没有考虑透彻。

你可能会有预感，这个结果应该和子问题的划分数量有关，确实，只增加线程数而不增加子区间的划分数是没用的。然而，因为本例是一个计算密集型问题，我们已经知道把线程数增加到比处理器核心数还多的方法是无效的。所以问题的关键还是在于子区间的划分数。下面让我们来尝试证明这一点并得到最终的解决方案。

让我们打开活动监视器（在 Windows 上是任务管理器，在 Linux 上是系统 / 资源监视器）来观察处理器核心的活动。在用顺序模式和并发模式逐一执行素数数量计算器程序之后，结果如图 2-2 所示。

采用顺序方法实现的程序的 CPU 利用率曲线和我们之前预想的一样。采用并发方式实现的程序则有所不同，即在大多数情况下 CPU 的两个核都是满负载运转的，而对于顺序方式实现的程序来说这个指标只有 60%。这一现象告诉我们，两个线程 / 处理器核心上的工作负载并非均匀分布的，即前半部分计算任务要比后半部分完成得快。如果我们仔细考虑一下所给问题的性质，这个现象就很容以理解了，根据我们之前的划分方式，后半部分任务中所含数字的值要比前半部分大，而数字的值越大则判断其是否素数的计算量也越大，所以后半部分任务执行就会比较慢。为了追求加速极限，我们需要为两个线程分配等量的工作负载。

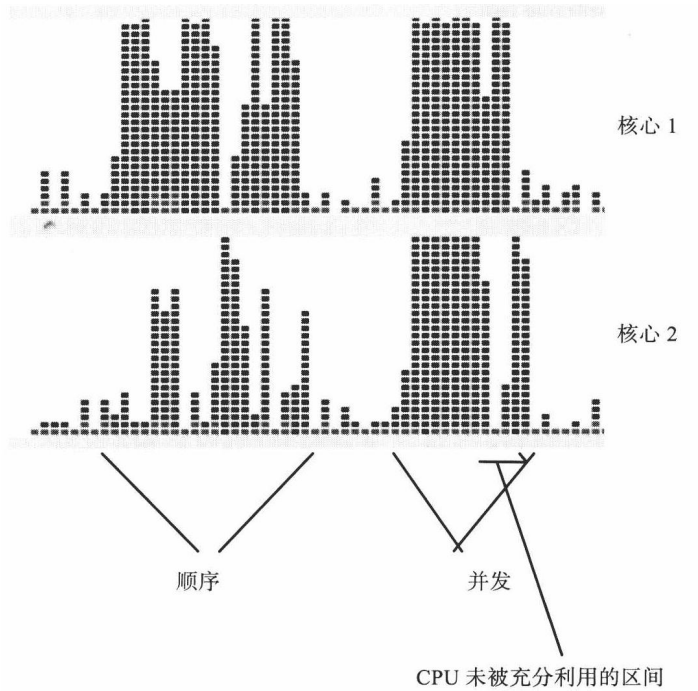


图 2-2　顺序和并发计算过程中处理器核心的活动

一般来说，为每个子任务分配均匀的工作负载是比较困难的，因为这需要对问题本身以及程序在整个输入数据范围内的行为都有相当深入的理解。单就计算素数数量这个问题来说，我们之前采用的方法是将数据按其串行顺序（即自然顺序。——译者注）分成两等份。而现在我们可以尝试将输入数据里的大小数字进行重新组合，以达到负载均匀分布的效果。例如，我们仍将输入数据分为两部分，然后考虑将区间前 25% 和最后 25% 的数据合并为一个子区间，而中间数据为另一个子区间。我们原来的代码并不是这样划分的，如果把之前写的那个并发计算的代码改成这种划分方式，就会发现程序运行时间减少了。更进一步地，如果想把程序放在拥有更多处理器核心的机器上跑，我们可能需要把输入数据划分成更多子区间。虽然子区间数目越多划分难度也就越大，但幸运的是我们找到了一个简单的解决方案。

只将输入数据划分成少量子区间（如两个）的主要问题是可能会导致一个 CPU 核心干了太多活而其他 CPU 核心则很无所事事。所以我们将问题拆分得越好，程序就越容易保持所有处理器核心都一直忙碌的状态。我们将输入数据拆成若干个小的子区间，并使子区间数量超过线程数。当线程完成子区间数据的计算之后，就可以继续挑选其他子区间数据执行运算。在总体的计算过程中，某些处理器核心可能执行了一个需要跑很久的任务，而其他处理器核心则可能挑了若干个耗时相对较短的任务来执行。让我们看一下线程池大小和子区间划分数的变化将对性能产生怎样的影响。我们把线程池大小设为 2 并将子区间划分

数改成 100，结果如下：

```
Number of primes under 10000000 is 664579
Time (seconds) taken is 3.550659
```

进行上述调整之后，运行速度提高了 1.85 倍。虽然仍未达到提速 2 倍的预期结果，但已经十分接近了。我们可以尝试调整子区间划分数或调整划分策略来进一步提高运行速度。例如，采用完全随机（fairly ad hoc）划分方式，我们将会获得相当好的速度提升。然而完全随机划分也并非对所有情况都适用，所以问题的关键还是为了有更高的利用率需要确保每个处理器核心都能分摊到均匀的工作负载。

2.5　并发方法对于计算密集型应用程序的加速效果

虽然计算素数数量的例子本身非常简单，但却展示了子任务的工作负载均衡对计算密集型操作的重要性。

上例中我们提到了输入数据子区间的划分数和线程池大小两个参数，下面让我们看看这两个参数对程序性能的影响。我将程序部署在一个 8 核 CPU 的电脑上，采用客户端模式（因为该程序在此模式下比在服务器模式下运行时间更长，且运行在客户端模式更容易对结果进行比较）并调整上述两个参数值来执行计算素数数量的并发版程序。程序执行的性能曲线如图 2-3 所示。

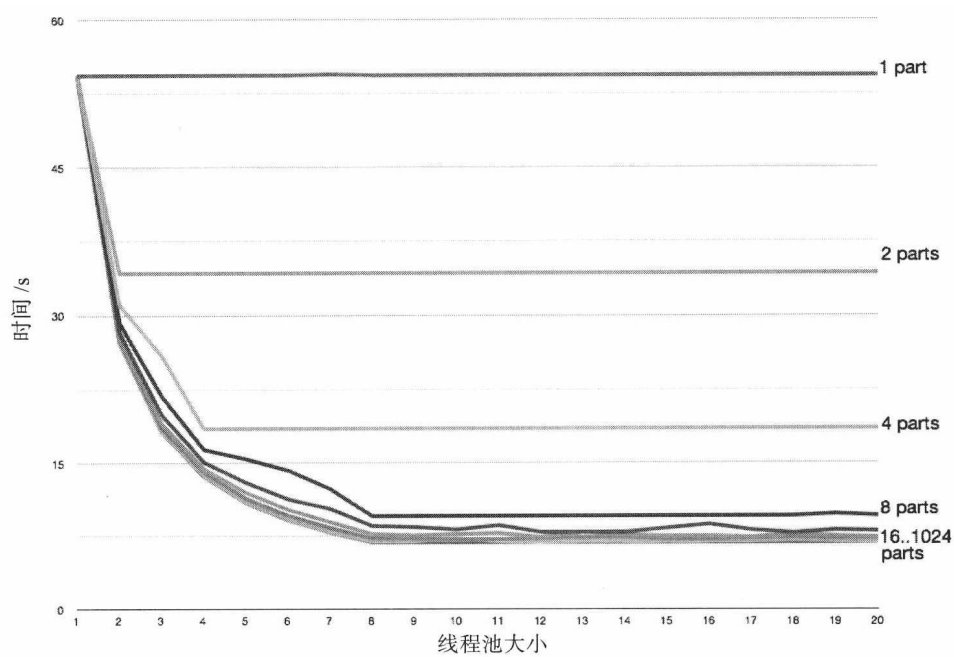


图 2-3　在 8 核处理器上用客户端模式执行素数计算：线程池大小和划分数对性能的影响

通过上图我们可以学到关于构建计算密集型并发应用程序的几点经验：

- 子任务的划分数应不少于处理器核心数，图中我们可以看到子任务划分数小于 8 时性能都是很差的。
- 线程数多于处理器核心数对性能提升毫无帮助，图中我们可以看到，无论哪种划分方式，只要线程数量大于 8，则性能曲线就基本变成水平了。
- 在子任务划分数超过一定数量之后，再增加子问题划分数对于性能的提升将十分有限。例如，设定程序跑在 8 核机器上且线程池大小为 8，则子任务数为 16、64、256、512 和 1024 时程序运行耗时分别为 8.4s、7.2s、6.9s、6.8s 和 6.8s。

最后我们再来看看问题拆分对性能的影响。如果划分操作所需运算过多，则对实际计算而言是一种的资源浪费。所以，保持一个合理的划分数并使所有处理器核心都有足够的工作量才是最关键的。因此我们需要设计一个问题拆分的简单方法并观察该方法是否能够均衡利用所有处理器核心。

2.6　有效的并发策略

在使用并发方法的时候，我们一方面希望保证一致性和准确性，另一方面又希望程序在给定的硬件环境下达到最佳性能。在本章中，我们会一起来探寻两全其美的方法。

一旦完全消除了共享可变状态，我们就可以很容易地避免竞争条件或一致性问题。因为当线程不再竞争访问可变数据的时候，程序就无需考虑变量（在多个线程之间）的可见性问题和穿越内存栅栏的问题。同时，我们也无须担心如何控制线程的执行序列。因为在这种情况下线程之间不存在竞争关系，所以代码中也无需针对互斥访问的部分进行保护。

我们应该尽可能地提供共享不可变性，否则就应该遵循隔离可变性原则，即保证总是只有一个线程可以访问可变变量。再次强调，这里我们讨论的不是对共享状态进行同步，而是确保总是只有一个线程可以访问可变变量。

开多少个线程以及如何将问题进行拆分都会影响到你的并发应用程序的性能。

首先，为了让程序获得并发所带来的好处，我们必须将问题拆分成一些更小的、可以并发执行子任务。如果问题中存在一个不能被拆分的关键部分，则即使将其并发也无法得到实际的性能提升。

如果任务是 IO 密集型或有关键的 IO 操作，则开多个线程将有助于提高性能。在这种情况下，程序开的线程数应该远远大于处理器核心数。我们可以通过第 2.1 节中描述的公式来估计程序所需线程的数量。

对于计算密集型任务，程序开的线程数超处理器核数对程序性能实际是有害无益的，详见第 2.4 节。然而，假设问题可以至少被拆分成线程数那么多个子任务，则程序至少开处理器核心数那么多个线程对性能提升是大有助益的。

虽然线程数是影响性能的一个关键因素，但却并非是唯一要素。每个子任务的工作负载以及完成每个子任务相对于其他子任务的耗时，亦是非常重要的。对问题进行统一划分需要花费非常多的精力，且不一定比随机划分的结果来得更好。权衡投入与产出之后，我们应该尝试去通过一种简单的划分方式来实现子任务的均衡工作负载。在任何情况下，只

有深入理解问题的本质以及问题在不同输入情况下的行为才能设计出好的划分方式。此外，划分的复杂度一般取决于问题的复杂度。

2.7 小结

以下是一些利用多核处理器并发能力来提升性能的方法：

- 我们需要将程序拆分成可以并发运行的多个子任务。
- 我们应该至少开处理器核心数那么多个线程，倘若问题的规模大到可以享受这么多线程带来的好处的话（如果问题规模太小，线程管理和调度反而成为影响性能的瓶颈。——译者注）。
- 对于计算密集型应用程序，我们应将程序线程数限制为与处理器核心数相同。
- 对于 IO 密集型应用，阻塞时间是影响线程数量的关键因素。
- 可以用如下公式估算程序所需线程数量：

线程数 =CPU 可用核心数 /(1– 阻塞系数)，其中 0 ≤阻塞系数 <1。

- 我们应该将问题分解为若干子任务，这样才能提高 CPU 的利用率并使其每个核都有足够多的活干。
- 我们必须避免共享可变状态，并用隔离可变性或共享不可变性取而代之。
- 我们应该充分利用现代线程 API 和线程池。

接下来，我们将讨论一些解决状态问题的设计方法。

第③章

设计方法

我们无法避免操纵状态，因为那是程序不可分割的组成部分。就好像编译器需要能够将一堆源文件组织起来并将其编译成字节码，而邮件应用程序则需要从一堆数据里面捞出那些未读的邮件，诸如此类。

与在 Java 里常用的方法一样，我们经常用设计程序的方法来设置状态。对象封装了状态，而对象的方法可以助其在选定的有效状态之间进行转换。以前当有人说他能够写出不改变状态的代码时，我总会嗤之以鼻。然而后来当他给我展示如何做到这一点的时候，我被他的做法深深吸引住了，同时也感到深受启发，那感觉就像在书店里的小孩一样[⊖]。

操纵状态并不一定意味着改变状态。考虑用状态转换来代替修改状态，这是一种无需修改任何东西就能改变状态的设计方法。在本章中，我们将会研究无需更改内存中任何数据的程序设计方法。不要看本章很短，但其中却涵盖了很多设计的基础方法。

3.1 处理状态

虽然处理状态是不可避免的，但是我们有 3 种方法可以用于处理状态：共享可变性（shared mutability）方法，隔离可变性（isolated mutability）方法和纯粹不可变性（pure immutability）方法。

一个极端情况是共享可变性方法，我们创建的变量允许所有线程修改，当然是在一个可控的模式下。对于大多数像我们这样的 Java 程序员来说，使用共享可变性来进行编程虽然看似简单，但却可能会导致同步问题。我们必须要保证代码在合适的时间穿越内存栅栏，并使变量具有良好的可见性。为了使用共享可变性，我们必须保证不会有两个线程同时修改同一个字段，并且对多个字段的修改必须满足一致性原则。然而，我们无法借助编译器或运行时环境来判断运行结果的正确性：我们需要分析代码来保证我们所做的事情是正确的。当我们接触到代码的那一刻，我们就不得不重新分析其正确性，因为这样写出来的程序在同步方面太容易出错了。幸运的是，我们还有其他选择。

在处理状态方面，隔离可变性方法是一个可选的折中方案。在该方法中，变量是可变的，但在任意时刻只有不超过一个线程可以看到该变量。在使用该方法时，我们保证任何在多个线程之间共享的事物都是不可变的。Java 程序员会发现这种设计很容易实现，所以隔离可变性可能是一个不错的方案。

纯粹不可变性方法是另一种极端的做法，该方法中所有事物都是不允许更改的。要使

⊖ 我的儿子使我明白一件事：其实逛书店比在糖果店玩更有趣。

用这样的设计可不大容易，部分原因是由问题的性质决定的，但更主要的原因还是我们对这种方法缺乏设计使用经验。编程语言的特性更增添了其使用难度，使用不可变性方法来写 Java 代码需要花很多额外的精力、设定很多规则。这是一种范式的转变，可能需要设计一些特殊的数据结构和方法。然而，如果我们真的能够将这种方法付诸实现，我们将得到丰厚的回报，即实现简单且安全的并发。

我们所选择的处理状态的方法取决于所面临的问题和团队探索设计方法的意愿。纯粹不可变性方法是理想化的，但说起来容易做起来难，尤其是对于那些多年来一直使用改变共享状态的方法写代码的程序员来说更是如此。最起码，我们应该将注意力集中到隔离可变性方法上，同时避免使用纯粹的共享可变性方法。

刚刚我们已经讨论了三种处理状态的方法。接下来，我们将通过一个实例来学习如何应用这些方法。

3.2 探寻设计选项

处理状态是一项编程艺术实践活动。当我们处理输入数据、计算的结果以及更改的文件时都会涉及状态。我们无法回避状态，但我们可以选择如何处理它。本节我们将先挑选一个例子，下面几节将会探讨用不同的方法来处理该示例中的状态。

在近期的一个 Java 用户组会议上，在讨论过程中大家的话题逐渐转移到了与会人员的工龄上。我看到与会的既有年轻人也有年长的人，于是就主动提出想把所有人的工龄进行一下汇总。让我们探讨一下我在当时所考虑的几个解决方案。首先我想到的是，当我统计每个人的工龄的时候，总工龄应该是变化的，因为无法避免可变性，所以可变性出现了。由于房间里有很多人，所以我需要快速设计一个获取（与会人员工龄）总数的方法。下面我们将上一节提到的三种方法用在处理此问题的状态方面，并讨论其所存在的优缺点。

3.3 共享可变性设计

我设计的第一个统计工龄的方法是一个大家比较熟悉的方案：先在黑板上写个 0，然后挨个点名，并将其工龄加在总数上。

离黑板最近的是一个叫做 Fred 的家伙，当时大家还不认识他，他就先跳起来把他自己的工龄加到总数里，其他人一字排开，来竞争黑板的写入权。当 Fred 写完之后，下一个就轮到 Jane，依此类推。我们需要紧盯着黑板以防某些捣蛋鬼把总数改成像无限大这样的非法值。

这种方法相当于采用了共享可变性设计，但同时也带来了一些问题。在某一特定时间，我们必须监督多个试图修改总数的人（即让他们老实排队。——译者注）。此外，当一个人正在黑板上累加的时候，其他人都必须耐心等待。我希望我不是站在队尾的那个人……（指等太长时间了。——译者注）

用编程的术语来讲，如果将房间里的每个人都看做一个独立的线程，我们需要将他们对于共享可变变量 total 的访问进行同步。一个行事糟糕的参与者将会毁掉整个组成员的努

力（即写错一个就导致总数都白累加了。——译者注）。同时，这种方式也引入了过多的线程同步阻塞，虽然可以保证很高的线程安全性，但却会导致低并发。

当讨论组人数众多时，该任务可能变得非常耗时且令人沮丧（中间加错了就惨了。——译者注）。我们想要减轻痛苦，所以让我们探讨另一种方法。

3.4　隔离可变性设计

我走到黑板近前，没有像上一节那样写下 total 变量的初始值 0，而是将我的电话号码写了上去，这样一来屋里的每个人都可以将他们的工龄发短信告诉我了。

现在每个人都可以舒服地坐在自己椅子上把信息发给我了。他们无需排队等候，且此时他们完成任务的速度仅取决于他们敲手机键盘的速度。

我逐一收到了其他人发来的工龄数据，而发送者们是并发且非阻塞的。

这里相当于采用隔离可变性的设计方法。总工龄（total）被隔离开来：只有我持有这个变量，其他人不能访问该变量。通过隔离可变状态，我们消除了上一个方法中存在的问题。如此一来我们就不用再担心两个或多个人同时更新相同的事物了。

当收到其他人发来的数据之后，我需要不断地将工龄进行累加。我的手机负责将并发的消息塞进一个顺序的消息队列中供我处理（例如短信列表。——译者注）。当我认为其他人都已经把数据发给我了之后，我就可以在会议结束时公布一下大家工龄的总和了。

用编程的术语来讲，如果屋子里的每一个人，包括我自己在内，都看成是一个线程的话（实际上是一个角色，正如我们将在第 8 章所看到的那样），每个人都简单地发送一条异步消息给我就行了。消息本身是不可变的，就像短信一样，我收到的数据将会被拷贝到我这边，且不会对发送方的视图产生任何影响。我仅仅是简单地更改一个封装好的本地变量。由于我是唯一可以改变数据的人（线程），所以在处理数据的过程中无需同步操作。

如果能够保证共享数据是不可变的且可变数据是隔离的，那么我们就不用再关心同步问题。

3.5　纯粹不可变性设计

用完全不可变性的方法来处理问题与我们之前的编程习惯有所不同。更改数据感觉上是一件很自然的事情，因为我们一直都是用这种方法来写 Java 代码的。如果要考虑在不更改任何数据的情况下获得总工龄的方法是很困难的，但这确实是可能的。

我们可以让屋子里的所有人组成一条人链，所有人可以不用离开座位。然后让每个人接收自己左边那个人的工龄值，将自己的工龄累加到那个值上，并将累加值传递给自己右边的人。我需要把我的工龄告诉屋里的第一个人，因为他正在焦急地等着我（把工龄数据给他，他才好继续传递下去）。

在这个方法中，没有人更改任何事物。人链中的每个人都持有到自己为止前面所有人工龄总和。我从人链中最后一个人那里获取到的值就是整个讨论组的工龄总和。

我们能够在不改变任何事物的情况下计算总量。每个人都持有部分，并且可以产生一

个新的总量。对于收到的旧总量值，我们可以按需保留或丢弃（垃圾回收）。

本方法在函数式编程中是很常见的，可以通过函数组合来实现。我们可以在 Scala、Clojure、Groovy 和 JRuby 中使用像 foldLeft()、reduce() 或 inject() 等方法来实现这些操作。

本方法与并发性是毫无关联的。虽然花费了一点精力来组织或组合这个操作序列，但我们做到了通过完全的不变性方法来获得结果。如果人数非常庞大，我们甚至可以将他们分成若干小组，即构建一棵树而不是一条人链，来达到更高的并发性和速度。

3.6 持久的 / 不可变的数据结构

在用纯粹不可变性设计计算工龄的例子中，我们让每个人都创建一个新的总和。当这些数据在人链中流动的时候，一个新的数量被生成出来而老的数量被丢弃，这在本例中不是什么大问题，因为这个数量（所占空间）通常很小。

但是我们要处理的数据可能不会都如上例般那么小，而有可能是一个链表、一棵树或一个矩阵。我们有可能需要和表示锅炉、卫星和城市等诸如此类的对象打交道。所以我们不会傻到去一遍又一遍地拷贝这些大对象，因为这将严重拉低程序性能。我们将使用不可变的或持久化数据结构来解决大对象拷贝问题。

持久化[⊖]数据结构将其值按版本进行记录，所以新老数据可以并存或保持一段时间而不会降低程序性能。由于数据是不可变的，所以可以将数据共享出去，以便有效地避免拷贝所产生的代价。持久化数据结构可以提供超高效的“更新”操作。像 Clojure 和 Scala 这样的语言就广泛使用了这种类型的数据结构。我们接下来将会讨论两种不同类型的持久化数据结构的设计。

3.6.1 不可变链表

回到用户讨论组会议的例子。假定我们已经收集全了所有与会者的姓名，下面就让我们一起探讨一下如何用纯粹不可变性方法来组织这些数据。我们可以采用单链表，其中每个节点有两个指针或引用。第一个引用用来指向人，第二个引用用来指向链表中的下一个节点，如图 3-1 所示。下面我们将看到如何用不可变链表来进行工作。

因为 Susan 刚刚加入了本会议，所以是时候该修改一下链表了。我们可能需要创建一个节点，该节点持有指向 Susan 的引用。但由于当前链表是不可变的，所以我们不能在链表尾部插入新节点。因为该操作需要改变链表尾节点的第二个引用的值，而这是一个不可变节点所不允许的操作。

我们不将新节点插入链表尾部，而采用头插法在不可变链表增加新节点。新节点的第二个引用将指向当前链表的头部，于是我们就可以在不修改已有链表的情况下得到一个新的链表。如图 3-1 所示，新旧链表共享除头部以外的所有节点。通过这种方法，添加新节点操作仅需常数时间且与链表长度无关。

从链表头部删除节点同样仅需常数时间。执行删除操作后得到的新链表较删除前少了

⊖ 持久化这个词在这里并不表示任何与存储有关的事物，它只表示数据在一段时间内一直留存或保持不变。

一个元素，并持有指向原链表第二个元素（可以通过一个简单的读操作获取到）的指针。

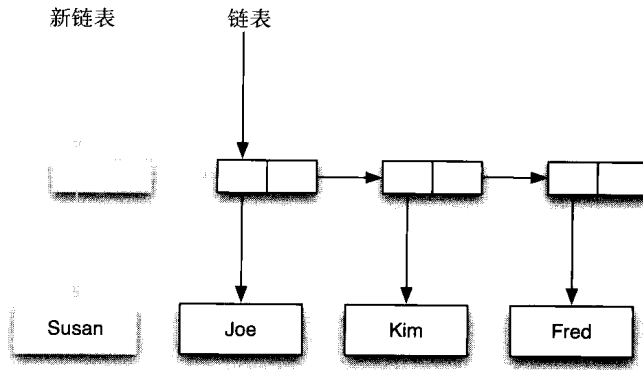


图 3-1　持久化链表处理

由于从不可变链表头部添加删除节点仅需常数时间，所以如果我们能够设计出高效地操作链表头部（而非中部和尾部）的算法或方案，则程序性能将会因此获得提升。

3.6.2　持久化的 Tries

在保存状态方面，不可变链表给我们带来了相当大的性能提升，但我们无法总是围绕链表头结点来组织操作。与此同时，数据通常不是一个简单的链表，而更像树或 hashmap 这样更复杂的结构，所以更适合用树来保存。然而我们无法将不可变链表的操作经验简单应用到对于树的操作上（即只对树的根节点进行操作），因为树需要修改非根节点才能支持插入或移动操作。

如果我们将树展开来看，则对于树的改动可以简单转化为对该树的一个分支的改动。这正是供职于瑞士 EPFL 的研究员 Phil Bagwell 所做的工作，他采用高分支因子（high branching factor），即每节点至少含 32 个子节点，来创建其称之为 trie 的结构，详情请参见附录 2。我们后面将会讨论，高分支因子可以有效减少对于 trie 结构的操作的时间。

除了高分支因子（子节点数≥ 32）之外，trie 结构还利用了一种特殊的性质来组织节点的 key，即节点的 key 是由其路径决定的。换句话说，我们不用在节点中保存 key，因为节点的位置即可作为其 key 值。例如，我们用数字作为路径且令分支因子为 3，虽然这是一个比推荐的值（≥ 32）小很多的数，但更有助于使问题简化并方便我们进行讨论。我们将使用该树来保存前来参加部门会议的人员名单（Joe、Jill、Bob、Sara、Brad、Kate、Paul、Jake、Fred 和 Bill，索引顺序与名单顺序一致），如图 3-2 所示。

由于我们在本例中使用的分支因子为 3，因此以 3 为基底的路径就代表了每个节点的索引。例如，Bill 所在的分支路径为 100（三进制表示法，十进制值为 $1*3^2+0*3^1+0*3^0=9$。——译者注），可知其索引值为 9。同样，Brad 所在路径为 11（三进制表示法，十进制值为 $1*3^1+1*3^0=4$。——译者注），所以其索引值为 4。

然而 Bagwell 的 trie 结构并非不可变的，所以 Rich Hickey（Clojure 的创造者），在 Clojure 中自行实现了一个持久化的哈希 trie 结构。一个分支因子为 32 的 trie 结构至多用 4 层就可以保存一百万个元素，而更改其中任意元素的操作至多只需要拷贝 4 个元素（即插入路径上的所有节点均需拷贝并替换。——译者注），这已经近似于常数时间了。

我们可以使用 trie 来实现多种不同的数据结构，如树、hashmap 和链表等。我们在图 3-1 中所看到的不可变链表只允许在其头部进行插入和删除操作。然而 trie 则没有此限制，若采用 trie 结构，我们就可以在一个不可变链表的任意索引处“插入”和“删除”元素。

例如，我们想要在图 3-2 所示的 Trie 结构的与会名者单尾部添加一个名为 Sam 的新成员。Sam 的索引值为 10，可推算出其路径为 101，所以 Sam 节点应作为 Sara 节点的子节点、Bill 节点的右兄弟节点插入到 Trie 中。因为所有节点都是不可变的，为了完成插入任务，我们需要拷贝 Sara、Jill 和根节点，其他节点则不受影响。进行完本次选择性复制操作之后的 trie 结构如图 3-3 所示。

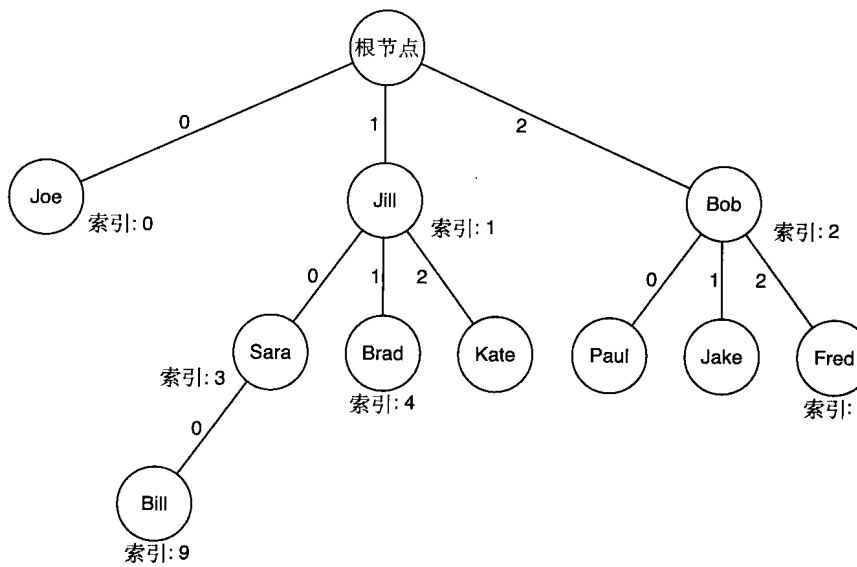


图 3-2 使用 trie 保存成员列表

在此方案中，我们仅需拷贝 3 个节点而非全部节点，即受影响的这 3 个节点都是待插入节点的祖先。这种拷贝操作本身是一种浅拷贝，即仅拷贝指向子节点的指针而非拷贝子节点的数据。当我们将分支因子增加至 32 或更多时，受插入操作影响的节点数将仍然只是 4 个左右，即 trie 结构的层数。而与此同时，该结构可持有的节点数量可达百万。因此，向链表中插入新人的操作是可近似认为是常数时间的。将元素插入到 trie 结构的任意索引位置将会比标准插入代价更高。因为根据插入位置的不同，我们将会需要一些额外的部分拷贝（partial copying）来完成插入操作。未受插入操作影响的节点及其祖先节点将仍然可以完整地共享给别人用。

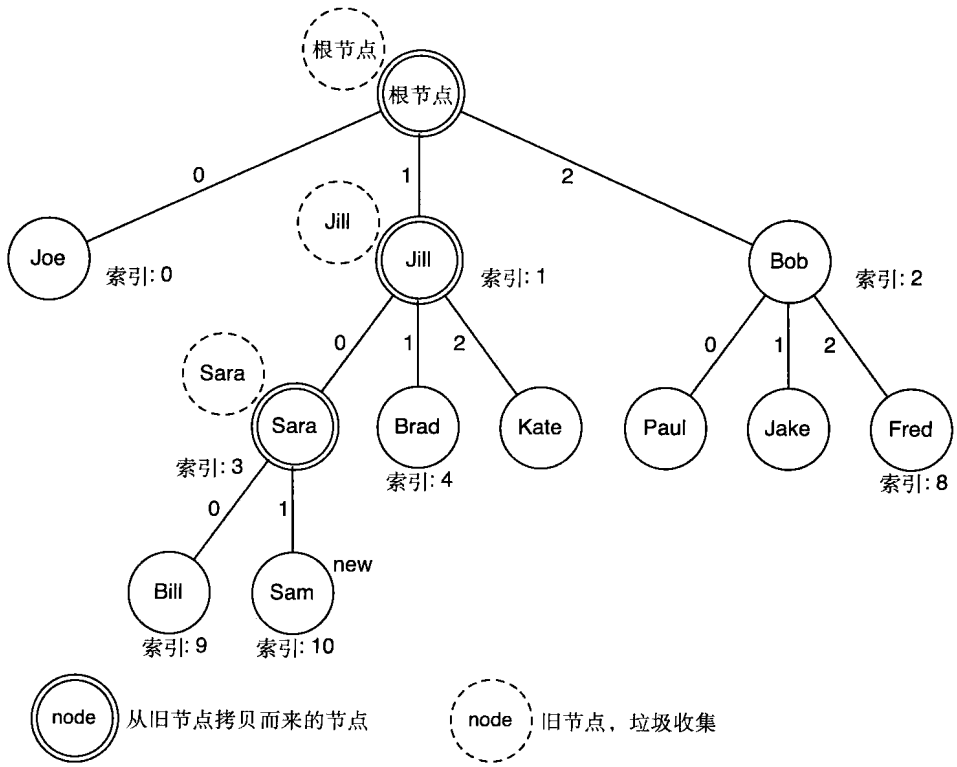


图 3-3　“更改”一个持久化列表

3.7　选择一种设计方法

我们可以通过使用隔离可变性方法和纯粹不可变性方法来避免大多数并发问题。就大部分情况而言，使用隔离可变性方法比使用纯粹不可变方法在编程方面要简单得多。

如果选择了隔离可变性方法，需要保证可变变量切实被隔离开，且绝不会被超过一个线程访问。同时还需保证线程之间传递的消息都是不可变的。在实际应用中，我们既可以使用 JDK 提供的并发 API，也可以使用任何一个基于角色的并发框架来实现消息传递。

使用纯粹不可变性设计将需要我们投入更多精力。在采用面向对象分解方式的应用程序中实现纯粹不可变性设计要比在采用函数式分解的应用程序中更难实现。需要设计更符合不可变性的算法，应用递归结构或函数式组合以及持久化数据结构，才有可能真正实现纯粹不可变性设计。

3.8　小结

虽然无法回避状态处理的问题，但我们还是有三种可选的设计方法：

- 共享可变性方法

- 隔离可变性方法
- 纯粹不可变性方法

虽然我们过去一直在使用共享可变性方法，但应该尽可能地避免它。消除共享可变状态是避免同步问题的最简单途径。然而选择这些设计方法比挑选一个库或一个 API 更耗精力，同时也需要我们退而思考如何去设计应用程序。

我们不想为了维护状态而牺牲性能，所以如果想使用完全不可变性的设计，就需要使用既可以维护状态、性能又好的新式数据结构。

本章我们讨论了如何处理状态的若干方法。下一章我们将使用 JDK 自带的工具来进行并发程序设计。

第二部分

现代 Java/JDK 并发模型

第④章

可扩展性和线程安全

如果你像我一样在上个世纪就开始用 Java 编程，那你肯定早就受够了早期 Java 版本附带的多线程 API 和集合 API。虽然这些 API 提供了线程管理和线程安全方面的基本功能，但却没有很好地顾及性能和可扩展性。尽管线程本身是相对轻量的，但创建线程依然是比较耗费时间和资源的。与此同时，在旧的 API 体系中，线程安全的实现是以牺牲可扩展性为代价的，即过于保守的同步策略对性能产生了很大的损害。

所以，即使这些 API 依然是可用的，如果我们想让代码兼具更好的可扩展性和更高的吞吐量，就不应该选择使用旧的线程 API。

在新的世纪，一组新的并发 API 随着 Java 5 一起发布了，这是对旧线程 API 的一次彻底的梳理和提升，新 API 的优势主要体现在如下三个方面：

- 这组 API 十分易于使用并且对于线程的处理更高效，特别是线程池。
- 新的同步原语提供了更细的粒度，因此比旧体系的原语更易于控制。
- 新的数据结构的设计是可扩展的，即在保证了线程安全的同时又兼具合理的并发性能。

为了深入了解这组 API 的优势和挑战，下面我们将一起构建一个计算指定目录下所有文件大小总和的磁盘工具。对于大型的目录结构，用顺序执行方式实现的程序需要很长的时间才能完成全部计算工作，所以我们随后会将程序改造成并发执行方式来提高运行速度。

4.1 用 ExecutorService 管理线程

为了计算目录大小，需要将整个操作过程拆分成若干个子任务，每个子任务负责一个子目录的计算。由于每个子任务都是在一个单独的线程里进行的，所以可以使用 Thread 类

的实例来完成计算工作。但问题是，线程是不可重用的，我们无法重新启动线程，并且在线程之间调度执行多个任务也不是件容易的事。我们当然不想创建子目录数那么多个线程，因为这种做法不具可扩展性并且也很容易出错。而 Java.util.concurrent API 就是为此目的引入进来的，即通过线程池的方式来管理大量线程。

在第 2.2 节中我们已经使用过 ExecutorService 类及其他各种类型的 Executor 了。ExecutorService 类、Executor 工厂类及其他相关 API 减轻了我们以前操作和管理大量线程时所遭受苦难。与此同时，根据我们需要执行的线程操作，API 也将线程池的类型进行了恰当的划分。

每个 ExecutorService 都代表一个线程池，其作用是将线程的创建与执行过程分离开来，而不是将线程的生命周期管理和任务的执行过程绑在一起。我们可以按需配置线程池的类型，单线程的、带缓存的、基于优先级的、按预定时间调度 / 按周期调度的和固定大小的，待调度任务的等待队列的大小亦可通过代码进行配置。通过这组 API，我们可以很容易地调度执行任意数量的任务。如果只是想简单地丢一个任务进去执行，我们只需将任务的执行过程封装到一个 Runnable 接口中就可以了。而对于那些需要返回结果的任务，我们可以将其封装到 Callable 接口里面。

我们可以使用 ExecutorService 的 execute() 或 submit() 函数来进行单个任务的调度，也可以使用 invokeAll() 函数来执行一组任务的调度。正如在众多可行解中找到最优解的优化问题一样，如果我们只关心某一特定子任务是否完成，则可以使用 invokeAny() 函数。所有这些方法都有带超时参数的重载版本，可通过该参数指定为得到结果所愿意等待的最长时间。

只要我们创建好了一个 ExecutorService，则线程池就已准备就绪并随时可以提供服务。如果没任务给它做，则线程池内的线程就处于空闲状态，带缓存的线程池（cached pool）除外，因为这种类型的线程池会将空闲超过一定时间的线程销毁。如果我们不再需要线程池了，可以调用 shutdown() 函数来关掉线程池。该方法并不会立即将线程池销毁，而是会等所有当前已经被调度的任务执行结束之后才会将其关闭，且在调用该方法之后我们就不能再调度任何新任务了。shutdownNow() 函数的功能则是尝试强制取消所有当前正在执行的任务，但是该方法不保证所有操作都能成功，因为任务能否被取消依赖于任务的实现代码是否正确响应了 interrupt() 函数的调用。

ExecutorService 还提供了一些检测服务是否已经被中止或关闭的方法。但我们最好不要依赖于这些方法，而是应该好好设计如何完成任务而不是操控线程 / 服务的消亡，即专注于任务的完成（应用程序逻辑）而非线程的终止（基础设施的活动）。

4.2　使线程协作

在将任务进行分解之后，我们就可以调度线程池内的多个线程来并发执行各个子任务，然后等待它们返回执行结果就行了。当所有任务完成之后，我们就可以像图 4-1 所示的那样进行后续处理。我们不想眼睁睁看着线程执行完了就被销毁，因为这些线程有些是可以被复用，去执行其他任务。此外，由于被调度子任务的执行结果才是我们真正关心的，所

以我们可以使用 Callable 接口和 ExecutorService 的 invokeAll() 或 submit() 函数来获取这些返回值。下面我们来看一个实例。

为了计算一个可能含有数千个文件的分层目录的大小，我们可以将计算任务分解成若干子任务。在所有子任务完成之后，我们还需要把所有返回的结果累加起来。

让我们首先看一下顺序计算目录大小的代码：

scalabilityAndTreadSafety/coordinating/TotalFileSizeSequential.java

```
public class TotalFileSizeSequential {
  private long getTotalSizeOfFilesInDir(final File file) {
    if (file.isFile()) return file.length();

    final File[] children = file.listFiles();
    long total = 0;
    if (children != null)
      for(final File child : children)
        total += getTotalSizeOfFilesInDir(child);
    return total;
  }

  public static void main(final String[] args) {
    final long start = System.nanoTime();
    final long total = new TotalFileSizeSequential()
      .getTotalSizeOfFilesInDir(new File(args[0]));
    final long end = System.nanoTime();
    System.out.println("Total Size: " + total);
    System.out.println("Time taken: " + (end - start)/1.0e9);
  }
}
```

在上面的代码中，首先我们给定一个目录，然后递归地累加其中所有文件 / 子目录的大小。

和本书中其他计算目录大小的程序一样，上面的代码第一次执行需要花很长时间，但如果在几分钟内再执行一次的话则耗时将会有所下降。这是因为在程序执行一次之后，文件系统会将文件 / 目录信息缓存在内存里，从而加快程序的执行速度。我忽略了程序第一次执行的时间，以便所有的比较测试都能利用系统缓存的加速效果。

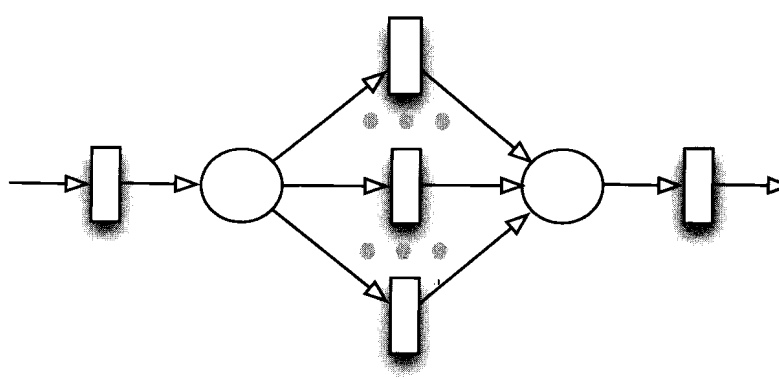


图 4-1　线程协作——调度与加入

我们先在一些目录上跑一下上述测试程序，后续我们还将会把这里的输出结果和该程序的并发版本的性能进行比较。

```
>java TotalFileSizeSequential /etc
Total Size: 2266456
Time taken: 0.011836

>java TotalFileSizeSequential /usr
Total Size: 3793911517
Time taken: 18.570144
```

毫无疑问，我们希望通过将上述代码进行并发改造来加快执行速度。于是我们将问题拆分成若干子任务，其中每个子任务负责一个子目录 / 文件的计算并返回其统计结果。Callable 接口非常适合于本例的应用场景，因为该接口的 call() 函数可以在任务完成之后返回一个结果值。当程序循环扫描目录下的每个文件 / 子目录的时候，我们就可以使用 ExecutorService 的 submit() 函数来调度子任务执行计算工作。随后我们可以调用 Future 对象的 get() 函数来获取子任务的计算结果，该对象主要起到一个委托的作用，即子任务一完成它就将结果返回给我们。下面让我们在 NaivelyConcurrentTotalFileSize 类中实现刚刚讨论过的这些逻辑。

scalabilityAndTreadSafety/coordinating/NaivelyConcurrentTotalFileSize.java

```
Line 1  public class NaivelyConcurrentTotalFileSize {
        private long getTotalSizeOfFilesInDir(
          final ExecutorService service, final File file)
          throws InterruptedException, ExecutionException, TimeoutException {
        if (file.isFile()) return file.length();

        long total = 0;
        final File[] children = file.listFiles();

        if (children != null) {
          final List<Future<Long>> partialTotalFutures =
            new ArrayList<Future<Long>>();
          for(final File child : children) {
            partialTotalFutures.add(service.submit(new Callable<Long>() {
              public Long call() throws InterruptedException,
                ExecutionException, TimeoutException {
                return getTotalSizeOfFilesInDir(service, child);
              }
            }));
          }

          for(final Future<Long> partialTotalFuture : partialTotalFutures)
            total += partialTotalFuture.get(100, TimeUnit.SECONDS);
        }

        return total;
      }
```

这段代码首先循环扫描给定目录。对于每个文件或子目录，我们都会创建一个计算其大小的子任务并在代码第 14 行调度线程池来执行。当把给定目录下所有文件和子目录的计

算任务都委托给线程池之后，我们就可通过 Future 对象来获取计算结果。为了计算整个目录的大小，我们需要在代码第 22 行迭代累加 Future 对象内所含子任务的计算结果。但我们不想无限期地堵在某个 Future 的 get() 函数上，所以就给 get() 函数加了一个最长等待时间的限制参数。换句话说，当其中某个子任务没有在规定时间内完成时，我们将抛出一个错误。每次递归调用 getTotalSizeOfFilesInDir() 函数的时候，只要还有未扫描过的文件或目录，我们都将创建更多子任务供线程池调度执行。

下面我们写一段调用上述函数的代码：

scalabilityAndTreadSafety/coordinating/NaivelyConcurrentTotalFileSize.java

```
private long getTotalSizeOfFile(final String fileName)
  throws InterruptedException, ExecutionException, TimeoutException {
        final ExecutorService service = Executors.newFixedThreadPool(100);
        try {
          return getTotalSizeOfFilesInDir(service, new File(fileName));
        } finally {
                  service.shutdown();
          }
  }

  public static void main(final String[] args)
    throws InterruptedException, ExecutionException, TimeoutException {
    final long start = System.nanoTime();
    final long total = new NaivelyConcurrentTotalFileSize()
      .getTotalSizeOfFile(args[0]);
    final long end = System.nanoTime();
    System.out.println("Total Size: " + total);
    System.out.println("Time taken: " + (end - start)/1.0e9);
  }
}
```

由于线程资源有限，所以我们不能创建太多（如何估算线程数请参见第 2 章）。在创建了线程池之后，就可以使用 getTotalSizeOfFilesInDir() 函数来遍历指定目录并计算其子目录 / 文件的大小。

让我们执行上述代码并观察运行结果：

```
>java NaivelyConcurrentTotalFileSize /etc
Total Size: 2266456
Time taken: 0.12506
```

目录大小的计算结果与顺序执行版本相同，然而速度非但没有提升，反倒比之前还慢了。这是因为很多时间都被消耗在线程调度上，而非花在计算这个目录结构相当扁平的 /etc 目录的实际工作中。但大家先不要失望，或许并发版本的代码在计算 /usr 目录时性能会有所改善，毕竟顺序版本的代码用了 18 秒多才完成。下面就让我们运行一下：

```
>java NaivelyConcurrentTotalFileSize /usr
Exception in thread "main" java.util.concurrent.TimeoutException
        at java.util.concurrent.FutureTask$Sync.innerGet(FutureTask.java:228)
        at java.util.concurrent.FutureTask.get(FutureTask.java:91)
        at NaivelyConcurrentTotalFileSize.getTotalSizeOfFilesInDir(
          NaivelyConcurrentTotalFileSize.java:34)
        at NaivelyConcurrentTotalFileSize.getTotalSizeOfFile(
```

```
    NaivelyConcurrentTotalFileSize.java:45)
at NaivelyConcurrentTotalFileSize.main(
    NaivelyConcurrentTotalFileSize.java:54)
```

呃……任务超时了，这可不是我们希望看到的。不过别担心，这正是我把这个版本的代码命名为 naive 的原因之一啊。

这个版本的代码的主要问题是，在 getTotalSizeOfFilesInDir() 函数中有阻塞线程池的操作。每当扫描到一个子目录的时候，getTotalSizeOfFilesInDir() 函数就将扫描该子目录的任务调度给其他线程（代码第 14 行）。一旦它调度完了所有任务，该函数就等待任何一个任务的响应（代码第 23 行）。当子目录数量不是很多的时候，这种做法不会有什么问题。但是如果待扫描的目录结构很深，则程序就会卡在这个地方。即线程池内的线程在等待某些任务的响应，而这些任务却在 ExecutorService 的队列中等待执行机会（由于程序是递归的且线程池大小是固定的，所以当子目录数超过线程池大小时，就会发生所有线程都在等待最底层子目录的计算结果，而最底层子目录的计算任务又没有额外的线程来执行，以至形成死锁。——译者注）。如果我们没有设置超时的话，这将演变成一种潜在的“线程池诱发型死锁”（Pool Induced Deadlock）。由于设置了超时时间，所以我们至少能够在出问题的时候中断程序运行而不是无休止地等下去。看来将顺序版本的代码并发化并不是一件简单的事，所以让我们一起回到白板前面研究一下如何修复这段代码。

在这个问题中，我们的目标是把计算各子目录大小的任务分派给不同的线程，且当我们等待其他任务 / 线程响应的时候又不能占住当前的主调线程[⊖]（避免死锁。——译者注）。

解决这个问题的一个方法是，令每个任务都返回给定目录的子目录列表而非该目录的大小。于是在主任务中，我们就可以分派其他任务来扫描列表中的子目录。该方法的好处是使线程被堵住的时间不会超过扫描给定目录的直接子目录的时间。当每个任务返回给定目录的子目录列表的时候，也会把该目录中所含文件的大小算好一并返回。下面让我们把符合上述设计思想的代码写进 ConcurrentTotalFileSize 类中并观察效果如何。

在扫描给定目录的子目录和文件的时候，需要把该目录下的子目录列表和所有文件的大小返回给主线程。因此需要一个不可变对象来保存这些返回值，所以让我们创建一个 SubDirectoriesAndSize 的内部类来实现这个功能。

scalabilityAndTreadSafety/coordinating/ConcurrentTotalFileSize.java

```
public class ConcurrentTotalFileSize {
  class SubDirectoriesAndSize {
    final public long size;
    final public List<File> subDirectories;
    public SubDirectoriesAndSize(
      final long totalSize, final List<File> theSubDirs) {
      size = totalSize;
      subDirectories = Collections.unmodifiableList(theSubDirs);
    }
  }
```

⊖ 后面我们将会看到，基于角色的模型非常适合解决这类问题。

接下来，我们要写这样一个函数：给定一个目录作为参数，返回一个包含该目录下所有子目录的列表和该目录下所有文件大小的 SubDirectoriesAndSize 对象。

```
private SubDirectoriesAndSize getTotalAndSubDirs(final File file) {
  long total = 0;
  final List<File> subDirectories = new ArrayList<File>();
  if(file.isDirectory()) {
    final File[] children = file.listFiles();
    if (children != null)
      for(final File child : children) {
        if (child.isFile())
          total += child.length();
        else
          subDirectories.add(child);
      }
  }
  return new SubDirectoriesAndSize(total, subDirectories);
}
```

在扫描到子目录之后，我们把计算任务交给其他线程完成（相当于广度优先搜索目录树。——译者注）。具体并发逻辑在下面的 getTotalSizeOfFilesInDir() 函数中实现。

scalabilityAndTreadSafety/coordinating/ConcurrentTotalFileSize.java

```
private long getTotalSizeOfFilesInDir(final File file)
  throws InterruptedException, ExecutionException, TimeoutException {
  final ExecutorService service = Executors.newFixedThreadPool(100);
  try {
    long total = 0;
    final List<File> directories = new ArrayList<File>();
    directories.add(file);
    while(!directories.isEmpty()) {
      final List<Future<SubDirectoriesAndSize>> partialResults =
        new ArrayList<Future<SubDirectoriesAndSize>>();
      for(final File directory : directories) {
        partialResults.add(
          service.submit(new Callable<SubDirectoriesAndSize>() {
          public SubDirectoriesAndSize call() {
            return getTotalAndSubDirs(directory);
          }
        }));
      }
      directories.clear();
      for(final Future<SubDirectoriesAndSize> partialResultFuture :
        partialResults) {
        final SubDirectoriesAndSize subDirectoriesAndSize =
          partialResultFuture.get(100, TimeUnit.SECONDS);
        directories.addAll(subDirectoriesAndSize.subDirectories);
        total += subDirectoriesAndSize.size;
      }
    }
    return total;
  } finally {
    service.shutdown();
```

```
    }
  }
```

上面的代码中，我们创建了一个大小为 100 的线程池，然后将最顶层目录放入待扫描目录队列。随后，只要还有待扫描的目录，我们就会在单独的线程中调用 getTotalAndSubDirs() 为每个目录执行计算任务。当所有线程的响应返回之后，我们就可以将得到的文件大小的部分和累加，并把子目录列表放进待扫描队列中。当所有的子目录都扫描完毕后，我们就能得到整个目录的大小。最后，为了使上述程序能跑起来，我们还需要写一个 main() 函数。

scalabilityAndTreadSafety/coordinating/ConcurrentTotalFileSize.java

```
  public static void main(final String[] args)
    throws InterruptedException, ExecutionException, TimeoutException {
    final long start = System.nanoTime();
    final long total = new ConcurrentTotalFileSize()
      .getTotalSizeOfFilesInDir(new File(args[0]));
    final long end = System.nanoTime();
    System.out.println("Total Size: " + total);
    System.out.println("Time taken: " + (end - start)/1.0e9);
  }
}
```

并发实现版的代码逻辑比顺序版的实现要复杂一些，但相比于旧的 Naively-ConcurrentTotalFileSize，新的 ConcurrentTotalFileSize 在设计上更胜一筹。由于新方法避免了潜在的线程死锁的问题，并能快速获取子目录列表，所以我们可以独立地调度线程来逐个遍历这些子目录。看来这次我们应该能得到预期的结果了，下面就让我们一起验证一下：

```
>java ConcurrentTotalFileSize /usr
Total Size: 3793911517
Time taken: 8.220475
```

首先，与 navie 版本不同的是，新版代码成功完成了所有计算任务。同样是计算 /usr 的目录大小，新版代码仅耗时 8 秒，而顺序版本则跑了 18 秒。优化效果相当明显，但我们就仅仅满足于此了？

让我们回顾一下这个例子中我们所做的事情。我们将任务分派给线程，然后在主线程中等待各个线程的计算结果。除主线程之外的其他线程都执行得很快，因为每个线程只需计算给定目录下的所有文件大小之和并返回该目录的子目录列表就行了。

虽然上述逻辑的设计思想很简单，但实现起来却不容易。我们不但需要创建一个用于保存计算任务返回结果的不变类，还需要花心思去设计如何不断地分派任务以及协调处理任务的返回结果。所以最终的结论是：最新版本的设计虽然提高了性能，却也同时引入了相当的复杂性。下面让我们看一下是否有办法对上述设计进行简化。

使用 CountDownLatch 辅助实现线程的协作

前面的例子中，我们用到了 Future 类提供的两种功能。首先，我们通过 Future 获取任

务的执行结果。与此同时，Future 也隐含地替我们完成了一些任务 / 线程之间的协调工作。在 Future 的帮助下，我们可以等待子目录的计算结果，然后主线程继续进行工作。但如果任务没有任何返回值，则 Future 就不大适合作为线程间的协作工具了。然而我们又不想只是为了协调的目的就人为地构造一个返回值出来，所以在遇到类似的情况时，我们可以使用 CountDownLatch 作为替代之用。

从代码的角度来讲，NaivelyConcurrentTotalFileSize 要比 ConcurrentTotalFileSize 简洁很多，而我一直倾向于使用简单的代码来解决问题。但 NaivelyConcurrentTotalFileSize 的主要问题在于，每个线程都需要等待手头的任务执行完之后才能去干别的事。唯一的利好消息是两个版本的代码都没有可变的共享状态。但如果我们在共享可变性方面稍微妥协一些的话[⊖]，就可以在保持代码简洁的同时又能使程序正常运转。让我们一起研究一下如何实现。

在下面的实现中，我们不再像之前那样返回子目录列表和文件大小，而是令每个线程都去更新一个共享变量。由于没有任何返回值，代码较之前大大简化了。同样，我们必须保证主线程要等待所有子目录遍历完成之后才能结束。为此，我们可以使用 CountDownLatch 作为等待结束的标记。线程闩（Latch）的作用是作为一个或多个线程等待其他线程到达其完成位置的同步点，这里我们简单地将其作为一个开关来使用。

下面我们创建了一个使用了 CountDownLatch 的 ConcurrentTotalFileSizeWLatch 类。其中，我们递归地将扫描子目录的任务委托给不同的线程。当扫描到一个文件时，线程不再返回一个计算结果，而是去更新一个 AtomicLong 类型的共享变量 totalSize。AtomicLong 提供了更改并取回一个简单 long 型变量值的线程安全的方法。此外，我们还会用到另一个叫做 pendingFileVisits 的 AtomicLong 型变量，其作用是保存当前待访问文件（或子目录）的数量。当该变量值变为 0 时，我们就调用 countDown() 来释放线程闩。

scalabilityAndTreadSafety/coordinating/ConcurrentTotalFileSizeWLatch.java

```
public class ConcurrentTotalFileSizeWLatch {
  private ExecutorService service;
  final private AtomicLong pendingFileVisits = new AtomicLong();
  final private AtomicLong totalSize = new AtomicLong();
  final private CountDownLatch latch = new CountDownLatch(1);
private void updateTotalSizeOfFilesInDir(final File file) {
  long fileSize = 0;
  if (file.isFile())
    fileSize = file.length();
  else {
    final File[] children = file.listFiles();
    if (children != null) {
      for(final File child : children) {
        if (child.isFile())
          fileSize += child.length();
        else {
          pendingFileVisits.incrementAndGet();
```

⊖ 这么做是为了让各位读者领略一下，在编程时陷入使用共享可变性的陷阱是多么容易？

```
            service.execute(new Runnable() {
              public void run() { updateTotalSizeOfFilesInDir(child); }
            });
          }
        }
      }
    }
    totalSize.addAndGet(fileSize);
    if(pendingFileVisits.decrementAndGet() == 0) latch.countDown();
  }
```

至此，我们实现了目录扫描的逻辑。下面我们需要完成创建线程池、启动目录扫描动作以及等待线程闩释放的代码。当 updateTotalSizeOfFilesInDir() 释放了线程闩之后，主线程也同时从 await() 调用中被唤醒并返回目录大小。

scalabilityAndTreadSafety/coordinating/ConcurrentTotalFileSizeWLatch.java

```
  private long getTotalSizeOfFile(final String fileName)
    throws InterruptedException {
    service  = Executors.newFixedThreadPool(100);
    pendingFileVisits.incrementAndGet();
    try {
     updateTotalSizeOfFilesInDir(new File(fileName));
     latch.await(100, TimeUnit.SECONDS);
     return totalSize.longValue();
    } finally {
      service.shutdown();
    }
  }
  public static void main(final String[] args) throws InterruptedException {
    final long start = System.nanoTime();
    final long total = new ConcurrentTotalFileSizeWLatch()
      .getTotalSizeOfFile(args[0]);
    final long end = System.nanoTime();
    System.out.println("Total Size: " + total);
    System.out.println("Time taken: " + (end - start)/1.0e9);
  }
}
```

这个版本的代码较之前简洁很多。下面让我们运行一下看看结果如何。

```
>java ConcurrentTotalFileSizeWLatch /usr
Total Size: 3793911517
Time taken: 10.22789
```

该版本代码的总耗时略大于 ConcurrentTotalFileSize 的执行结果。这是由于我们使用了一些额外的线程间同步导致的。因为一旦使用了共享可变性，我们就需要为了线程安全做一些额外的保护操作，而这些操作会降低程序的并发度。

在前面的例子中，我们把线程闩的值设为 1，于是 CountDownLatch 变成了一个标识所有任务全部结束的开关。我们还可以将线程闩的值设大一些，以便让多个线程可以同时处于等待其释放的状态。如果我们希望多个线程在继续执行其他任务之前都能抵达同一个协作位置，就可以采用这种方法来实现。需要注意的是，CountDownLatch 是不可复用的。所以一旦某个线程闩的实例在同步动作中被用过了之后，就必须废弃掉。如果我们的程序需

要一个可复用的同步点，则应该用 CyclicBarrier 来替代 CountDownLatch。

总体来看，ConcurrentTotalFileSizeWLatch 的执行性能要优于顺序版的 TotalFileSizeSequential。虽然其性能略逊于 ConcurrentTotalFileSize 版本，但代码却较之简洁得多。然而，该版本的代码存在着访问共享可变变量的风险，这是我已经警告了多次的问题。如果能做到在保持代码简洁的同时又能避免共享可变性就更理想了。我们将在第 8 章中研究鱼与熊掌可以兼得的方法。

4.3　数据交换

我们经常需要在多个相互协作的线程之间交换数据。在前面的例子中，我们使用了 Future 和 AtomicLong 来实现数据交换的功能。当需要在任务完成时获得一个返回值的时候，Future 就能派上用场。而 AtomicLong 和在 java.util.concurrent.atomic 包中定义的其他原子操作类对于处理单个共享数据的值来说非常有用的。虽然上述这些工具类都可以用于数据交换，但实际用它们来写代码时就会感觉很不灵活，正如我们在前面例子中所看到的那样。为了能够同时操作多个数据值或频繁地交换数据，我们希望能找到一种比之前用过的工具类更好用的数据传递机制。幸运的是，java.util.concurrent API 已经实现了很多相关的工具，这些工具为我们提供了以线程安全的方式进行线程间任意数据交换的能力。

如果只是想在两个线程之间交换数据，我们可以使用 Exchanger 类。Exchanger 实际上可以看做一个同步点，两个线程在该同步点能够以线程安全的方式互换数据。如果两个线程的运行速度不同，则运行较快的线程将会被阻塞，直到运行较慢的线程赶到同步点之后，两个线程才可开始交换数据。

如果想要在线程间互发多组数据，则 BlockingQueue 接口可以派上用场。顾名思义，该接口的特点是：如果队列里没有可用空间，则插入操作将会被阻塞；而如果队列里没有可用数据，则删除操作将被阻塞。JDK 提供了好几种功能各异的 BlockingQueue。例如，若想要使插入操作和删除操作一一对应，可以使用 SynchronousQueue 类。该类的作用是将本线程的每一个插入操作与其他线程相应的删除操作相匹配，以完成类似于手递手形式的数据传输。而如果希望数据可以根据某种优先级在队列中上下浮动，则可以使用 PriorityBlockingQueue。另外，如果只是想要一个简单的阻塞队列，我们可以选用链表实现的 LinkedBlockingQueue 或数组方式实现的 ArrayBlockingQueue。

我们可以使用阻塞队列来并发地解决计算文件大小问题，并且这里我们没有使用像 AtomicLong 这样的可变变量，而是把每个线程计算所得的部分文件大小的值插入到一个队列中。随后主线程可以遍历该队列获得每部分结果并进行累加。下面让我们先完成遍历目录的代码：

scalabilityAndTreadSafety/coordinating/ConcurrentTotalFileSizeWQueue.java

```
public class ConcurrentTotalFileSizeWQueue {
  private ExecutorService service;
  final private BlockingQueue<Long> fileSizes =
```

```
    new ArrayBlockingQueue<Long>(500);
  final AtomicLong pendingFileVisits = new AtomicLong();
  private void startExploreDir(final File file) {
    pendingFileVisits.incrementAndGet();
    service.execute(new Runnable() {
      public void run() { exploreDir(file); }
    });
  }
  private void exploreDir(final File file) {
    long fileSize = 0;
    if (file.isFile())
      fileSize = file.length();
    else {
      final File[] children = file.listFiles();
      if (children != null)
        for(final File child : children) {
          if (child.isFile())
            fileSize += child.length();
          else {
            startExploreDir(child);
          }
        }
    }
    try {
      fileSizes.put(fileSize);
    } catch(Exception ex) { throw new RuntimeException(ex); }
    pendingFileVisits.decrementAndGet();
  }
```

我们为每个子目录的遍历工作都分配了一个独立的任务。每个任务负责计算给定目录下所有文件大小之和，并在计算结束之后调用阻塞函数 put() 将该值插入到队列当中。每个子目录的遍历都是在一个独立的线程中进行的，即线程之间互不影响。

主线程则首先启动目录遍历任务，随后便只需简单地循环读阻塞队列并将其文件大小值累加起来，直至所有子目录遍历完成为止。

scalabilityAndTreadSafety/coordinating/ConcurrentTotalFileSizeWQueue.java

```
  private long getTotalSizeOfFile(final String fileName)
    throws InterruptedException {
    service  = Executors.newFixedThreadPool(100);
    try {
      startExploreDir(new File(fileName));
      long totalSize = 0;
      while(pendingFileVisits.get() > 0 || fileSizes.size() > 0)
      {
        final Long size = fileSizes.poll(10, TimeUnit.SECONDS);
        totalSize += size;
      }
      return totalSize;
    } finally {
      service.shutdown();
    }
  }
  public static void main(final String[] args) throws InterruptedException {
    final long start = System.nanoTime();
```

```
        final long total = new ConcurrentTotalFileSizeWQueue()
          .getTotalSizeOfFile(args[0]);
        final long end = System.nanoTime();
        System.out.println("Total Size: " + total);
        System.out.println("Time taken: " + (end - start)/1.0e9);
    }
}
```

我们来看一下这个版本的执行情况：

```
>java ConcurrentTotalFileSizeWQueue /usr
Total Size: 3793911517
Time taken: 10.293993
```

这一版的程序在性能方面与前一版相仿，但在代码简化方面又提升了一个档次，这主要得归功于阻塞队列帮我们完成了线程间的数据交换和同步操作。下面我们将会看到如何利用 Java 7 引入的新 API 来进一步改进上述方案。

4.4　Java 7 Fork-Join API

我们可以使用 ExecutorService 来管理线程并调度线程池中的线程来执行任务。然而线程池所含的线程数量是由程序员决定的，而且程序员调度执行的任务和任务在执行过程中所创建的子任务之间也是没有任何区别的。所以为了提高效率和性能，Java 7 ⊖为我们带来了专门针对 ExecutorService 效率和性能的改进版的 fork-join API。

ForkJoinPool 类可以根据可用的处理器数量和任务需求动态地对线程进行管理。Fork-join 使用了 work-stealing 策略，即线程在完成自己的任务之后，发现其他线程还有活没干完，就主动帮其他人一起干。该策略的使用不但提升了 API 的性能，而且还有助于提高线程利用率。

在 Fork-join API 中，活动任务（active task）所创建的子任务都是由与创建主任务所不同的另一套函数来负责调度的。通常我们在一个应用程序中只会使用一个 fork-join 池来调度任务，且由于该池使用了守护线程，所以用过之后也无需执行关闭操作。

为了更好地调度任务，我们提供了一些 ForkJoinTask 的实例（通常是其某个子类的实例）来配合 ForkJoinPool 函数的使用。我们可以用 ForkJoinTask 来创建（fork）任务，然后再将主线程 join 到任务的完成点上。ForkJoinTask 有两个子类：RecursiveAction 和 RecursiveTask。RecursiveAction 的子类主要用来执行不需要返回值的任务，而 RecursiveTask 的子类则主要用于执行需要返回值的任务。

Fork-join API 主要用于处理那些规模合理的任务，即如果每个任务所分担的开销都不大（也没有不确定的循环），则该 API 就可以达到合理的吞吐量。此外，Fork-join API 希望所有任务都不要有副作用（即不要改变共享状态）并且没有同步或锁操作。

⊖ 除本节之外，本书中所有 Java 代码都是使用 Java 6 完成的。在本节中，我们将使用 Java 7，其 SDK 的下载地址为 http://jdk7.java.net/download.html。如果你用的是 Mac 电脑，请参阅 http://wikis.sun.com/display/OpenJDK/Mac+OS+X+Port。

Fork-join API 非常适合于解决那些可以递归分解至小到足以顺序运行的问题。通过使用由 ForkJoinPool 管理的线程，多个较小的分解任务可以被同时执行。

让我们用 fork-join API 来解决计算目录大小的问题。回忆一下在第 4.2 节中我们开发的那个最原始的解决方案，该方案虽然思路很简单，但在处理结构复杂的大型目录时会引发“线程池诱发型死锁”，即目录扫描任务需要等待其子目录扫描任务完成之后才能继续执行，但父任务在等待的过程中又不会释放其主调线程，从而导致子任务无线程可用并最终产生死锁。Fork-join API 通过 work-stealing 完美地解决了这一问题。当一个任务处于等待其子任务结束的状态时，该任务的执行线程可以将该任务挂起，然后转去执行其他任务。下面让我们来看看使用 fork-join API 解决计算目录大小问题的代码。

scalabilityAndTreadSafety/fork-join/FileSize.java

```
public class FileSize {

  private final static ForkJoinPool forkJoinPool = new ForkJoinPool();

  private static class FileSizeFinder extends RecursiveTask<Long> {
    final File file;

    public FileSizeFinder(final File theFile) {
      file = theFile;
    }

    @Override public Long compute() {
      long size = 0;
      if (file.isFile()) {
        size = file.length();
      } else {
        final File[] children = file.listFiles();
        if (children != null) {
          List<ForkJoinTask<Long>> tasks =
            new ArrayList<ForkJoinTask<Long>>();
          for(final File child : children) {
            if (child.isFile()) {
              size += child.length();
            } else {
              tasks.add(new FileSizeFinder(child));
            }
          }

          for(final ForkJoinTask<Long> task : invokeAll(tasks)) {
            size += task.join();
          }
        }
      }

      return size;
    }
  }

  public static void main(final String[] args) {
    final long start = System.nanoTime();
```

```
        final long total = forkJoinPool.invoke(
            new FileSizeFinder(new File(args[0])));
        final long end = System.nanoTime();
        System.out.println("Total Size: " + total);
        System.out.println("Time taken: " + (end - start)/1.0e9);
    }
}
```

在 FileSize 类中，我们创建了一个 ForkJoinPool 实例的引用，该实例使用关键字 static 定义，即它可以在整个应用程序中共享。随后我们定义了一个名为 FileSizeFinder 的静态内部类。该类继承了 RecursiveTask 类并实现了 compute() 函数，我们可以用它来作为任务的执行引擎。在 compute() 函数中，我们将给定目录下的所有文件大小累加起来，并将扫描和计算子目录大小的工作委托给其他任务（即其他 FileSizeFinder 的实例）来完成。invokeAll() 函数将等待所有子任务完成之后才会执行下一步循环累加操作。在任务被阻塞期间，其执行线程并非什么也不做一直傻等所有子任务结束（就像一个优秀的团队中那些有高度责任感的成员所做的那样），而是可以被调度去做其他任务。最后，每个任务都将在 compute() 函数结束时返回给定目录下所有文件和目录的大小。

让我们在 Java 7 中编译并执行上述代码：

```
>java com.agiledeveloper.pcj.FileSize /etc
Total Size: 2266456
Time taken: 0.038218

>java com.agiledeveloper.pcj.FileSize /usr
Total Size: 3793911517
Time taken: 8.35158
```

从输出结果中我们可以看到，新版程序的性能要比本章前面其他并发版本好很多。同时我们也注意到，对于大型的分层目录，程序也没有重蹈 navie 版本的覆辙（即没出现“线程池诱发型死锁”）。

在本例中，我们递归地将扫描任务进行分解，直至任务无法再拆分。但一般来说，拆分粒度过细会显著增加线程调度的开销，所以我们并不希望将问题拆分得过小。

java.util.concurrent 包中定义了很多线程安全的集合类，这些集合类既保证了并发编程环境下的数据安全性，同时也可以被当成同步点来使用。虽然线程安全是很重要的，但我们也不想为此牺牲太多性能。下一节我们将研究 java.util.concurrent 包中与并发应用性能息息相关的一些数据结构。

4.5　可扩展集合类

像 Vector 这种在 Java 早期版本中发布的集合类虽然都是线程安全的，但却是以牺牲部分性能为代价的。所以程序对这些集合类的访问，无论是否有必要，都是线程安全且低效的。

后来，像 ArrayList 这样的新集合类虽然提高了访问速度，但又缺少了线程安全的特性，所以在需要线程安全的地方我们不得不再次牺牲性能来换取线程安全性，即用类似 Collections 的 synchronizedList() 方法作为普通的集合类的同步化“包装器”（wrapper）。

在 Java 5 发布之前，我们只能在性能和线程安全性之间进行痛苦地抉择。而 Java 5 的 java.util.concurrent 包中则定义了很多支持并发访问的数据结构，如 ConcurrentHashMap 和 ConcurrentLinkedQueue 等。这些集合类通过牺牲一些空间的方式实现了更好的并发访问性能，但在语义上与传统集合类会稍有不同。

同步集合类（synchronized collections）实现了线程安全的特性，而并发集合类（concurrent collections）则在保证了线程安全的同时也兼顾了并发访问的性能。使用同步集合类让人感觉好像是开车驶过一个交通灯控制的十字路口（总是走走停停。——译者注）；而使用并发集合类则感觉像行驶在一条拥挤的高速路上一样（拥挤却很少停顿。——译者注）。

如果我们在更改某个同步 map 的同时对其进行遍历，则该同步 map 将会抛出一个 ConcurrentModificationException。究其本质，是由于在进行遍历的时候，我们其实是持有了该 map 的一个互斥锁，即悲观锁（pessimistic lock）。这种做法虽然可以增加线程安全性，但同时也将显著降低并发访问性能。因此，用这些同步集合类开发的程序通常都会导致较低的吞吐量。

另外，并发集合类本身就是为提高吞吐量而设计的，因此可以允许多个操作同时进行。例如，我们能够在遍历一个并发 map 的同时对其进行修改。由于 API 保证了集合的完整性，所以我们不会在一次遍历中重复看到同样的元素。这种做法其实是一种语义上的妥协，因为我们在对集合进行遍历或获取元素的同时，集合内的元素可能正在被改变或者被删除。

下面让我们来探讨一下 ConcurrentHashMap 和旧的 map 之间的区别。

在下面的代码中，我们将在一个独立的线程中遍历一个保存着学生考试成绩的 map。而在遍历过程中，我们会在其中插入一个新的 key 进去。

scalabilityAndTreadSafety/concurrentCollections/AccessingMap.java

```
public class AccessingMap {
  private static void useMap(final Map<String, Integer> scores)
    throws InterruptedException {
    scores.put("Fred", 10);
    scores.put("Sara", 12);

    try {
      for(final String key : scores.keySet()) {
        System.out.println(key + " score " + scores.get(key));
        scores.put("Joe", 14);
      }
    } catch(Exception ex) {
    System.out.println("Failed: " + ex);
  }

  System.out.println("Number of elements in the map: " +
    scores.keySet().size());
}
```

如果我们用的是一个普通的 HashMap 或其同步外覆类，那么这段代码肯定是有问题的，因为在遍历途中执行插入操作将导致我们刚刚讨论过的那些违规问题。事实上，在遍

历一个同步的集合之前，我们需要将集合锁住以避免发生此类违规问题。但如果主函数中用的是 ConcurrentHashMap，那么我们就不会受困于 ConcurrentModificationException。为了观察不同的数据结构在行为上的差异，我们在测试 useMap() 时会先使用 HashMap 的实例，然后改用 HashMap 的同步外覆类，最后再换用 ConcurrentHashMap 的实例。

scalabilityAndTreadSafety/concurrentCollections/AccessingMap.java

```
  public static void main(final String[] args) throws InterruptedException {
    System.out.println("Using Plain vanilla HashMap");
    useMap(new HashMap<String, Integer>());

    System.out.println("Using Synchronized HashMap");
    useMap(Collections.synchronizedMap(new HashMap<String, Integer>()));

    System.out.println("Using Concurrent HashMap");
    useMap(new ConcurrentHashMap<String, Integer>());
  }
}
```

从下面的执行结果中我们可以看到，传统的 map 不允许在遍历过程中对于集合类进行修改（即使是在同一个线程内），而 ConcurrentHashMap 则很好地解决了这一问题。

```
Using Plain vanilla HashMap
Sara score 12
Failed: java.util.ConcurrentModificationException
Number of elements in the map: 3
Using Synchronized HashMap
Sara score 12
Failed: java.util.ConcurrentModificationException
Number of elements in the map: 3
Using Concurrent HashMap
Sara score 12
Fred score 10
Number of elements in the map: 3
```

除了允许交叉读写之外，并发集合类的吞吐量比同步版本也要好很多。这是由于并发集合类并不是只通过一个互斥锁来进行同步管理的，而是将整个集合数据分成若干个块（bulk），并允许更新操作和读操作在多个块上同时进行。在使用并发集合类时，读操作并非总能看到最新的数据值。这是因为如果读操作在某一个块（bulk）的更新期间发生，则这个读操作不会为了等待整个更新操作完成而被阻塞。这就意味着，我们只能看到集合数据中的部分变化。在任何情况下，并发集合类都会保证数据的可见性或满足 happens-before 的行为特性。

下面让我们来观察一下并发 map 与其同步版本之间的性能差异。在这项测试中，每个线程的工作负载都是大致相同的，所以如果线程数增加，则进程的整体工作负载也会线性增长，同时线程之间的竞争也会随之加剧。在每一个线程中，我都会随机访问某个多线程共享的 map 中的一个 key。如果这个 key 不存在，我就会以 80% 的时间将其插入 map 中；而如果这个 key 已经存在，则我会以 20% 的时间将其移除。下面我将先后在一个 ConcurrentHashMap 和一个进行了同步包装过的 HashMap 上执行上述测试。

在一个 8 核处理器上进行上述测试所得吞吐量对比结果如图 4-2 所示。当线程数少于 2 时，同步 HashMap 的表现要略好于 ConcurrentHashMap，也就是说，当没有什么并发操作的时候，同步版本的性能要更好一些。而当线程数增加的时候，ConcurrentHashMap 的性能要远远胜过同步 HashMap。

一旦适应了并发集合类与旧集合类在语义方面的差异，我们就能够写出比用同步集合类性能更好的代码。

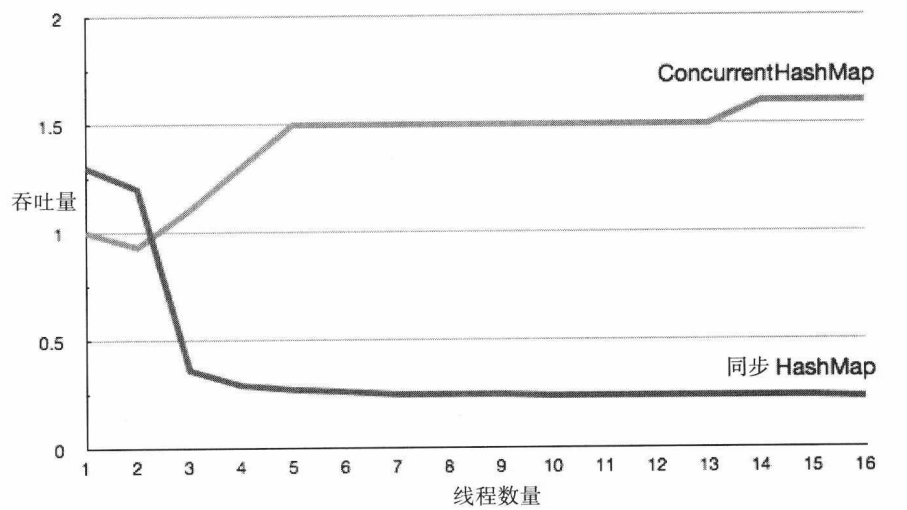


图 4-2　在一个 8 核处理器上，ConcurrentHashMap 和同步 HashMap 之间的吞吐量对比

4.6　Lock 和 Synchronized

到目前为止，在本章的例子中我们都在避免使用显式同步操作。但如果我们想要更充分地使用 JDK 的并发 API 的话，就再也无法避开同步操作了，我们将会在需要协调多个变量或对象的变更之处用到它。

在 Java 中，我们有两种方式可以获得锁，即原始的 synchronized 关键字和新的 Lock 接口。相比 synchronized 关键字，Lock 接口可以让我们对锁操作进行更好的控制。下面让我们一起来看看如何做到这一点。

4.6.1　synchronized 关键字

我们可以使用 synchronized 关键字来显示地获取对象的 monitor/ 锁，并通过“先获取 monitor 并在代码块结尾处释放掉”的方式来帮助线程穿越内存栅栏。但是 synchronized 关键字的能力过于简单并且在使用上也有相当多的限制。

更加不幸的是，由于我们没办法设置 synchronized 关键字在获取锁的时候只等待有限时间，所以 synchronized 可能会导致线程为了加锁而无限期地处于阻塞状态。

在实际工作中我们会发现，想要对使用了 synchronized 的代码进行线程安全方面的单元测试是根本不可能的。因为没有一个有效、简单且确定的方法，例如用模拟的标识（mock）来代替 synchronized 关键字，来检验我们是否已经将一段需要互斥执行的代码块封装到一个被 synchronized 修饰的段 / 块中。

此外，使用 synchronized 关键字等同于使用了互斥锁，即其他线程都无法获得对象 monitor 的访问权。这种策略对于读多写少的应用而言是很不利的，因为即使多个读者看似可以并发运行，但他们实际上还是串行的，并将最终导致并发性能的下降。

正如我们所看到的那样，虽然 synchronized 已经作为一个关键字被固化在 Java 语言中了，但它只提供了一种相当保守的线程安全策略，且该策略开放给程序员的控制能力极弱。

4.6.2　Lock 接口

Java 5 的 Lock 接口为我们提供了比 synchronized 关键字更强的控制能力，详情请参阅附录 2。

与 synchronized 关键字十分相似的是，Lock 接口的实现同样保证了对于其方法的调用将会跨越内存栅栏。我们可以用 Lock 接口的 lock() 和 unlock() 函数来获取和释放锁，例如：

```
aMonitor.lock();
try {
  //...
} finally {
  aMonitor.unlock();
}
```

上例中我们在 finally 块里执行 unlock() 的目的是保证即使在发生异常的情况下解锁动作也能正常完成。虽然 lock() 是一个阻塞式调用，但我们可以用其非阻塞的变体 tryLock() 来获得锁。为了等待有限的时间，可以使用超时参数来调用 tryLock() 函数。此外，在 Lock 接口中，lock() 函数还有一个变体可以允许我们打断一个正在等待锁的线程，以使其恢复运行。

下面让我们看看 Lock 接口是如何解决之前在研究 synchronized 关键字时所提出的那些问题的。

通过 tryLock() 函数，加锁的请求可以不被强制阻塞。相反地，我们可以即时地检查是否我们已经获得了锁。同时，如果程序确实想要等待某个锁的控制权，我们也可以在等待时间上加以限制。

Lock 接口使得进行线程安全方面的单元测试变得非常容易。详情请参见附录 2。我们可以简单地模拟出 Lock 接口的实现，并检查待测试的代码是否在合适的时间请求和释放了锁。

我们可以用 ReadWriteLock 在所需要的代码段上加并发读 / 互斥写锁。这样一来，多个读者就可以并发地连续执行而无需等待，而仅在与写操作冲突时才会有所延迟。

下面让我们看一个展示如何使用实现了 Lock 接口的 ReentrantLock 类的例子。顾名思义，该类允许已经获得锁的线程可以重复请求锁，即允许这些线程反复进入互斥区。

之前有家银行联系我们说想要我们帮忙实现一套在两个账户之间转账的代码。其中 Account 类的定义如下所示：

scalabilityAndTreadSafety/locking/Account.java

```
public class Account implements Comparable<Account> {
  private int balance;
  public final Lock monitor = new ReentrantLock();

  public Account(final int initialBalance) { balance = initialBalance; }
  public int compareTo(final Account other) {
    return new Integer(hashCode()).compareTo(other.hashCode());
  }

  public void deposit(final int amount) {
    monitor.lock();
    try {
      if (amount > 0) balance += amount;
    } finally { //In case there was an Exception we're covered
      monitor.unlock();
    }
  }

  public boolean withdraw(final int amount) {
    try {
      monitor.lock();
      if(amount > 0 && balance >= amount)
      {
        balance -= amount;
        return true;
      }
      return false;
    } finally {
      monitor.unlock();
    }
  }
}
```

这个 Account 类有颇多需要注意的地方，下面让我们逐一讨论这些细节。

假设有两个线程同时在相同的两个账户之间转账并且转账顺序相反，类似这样：

```
thread1: transfer money from account1 to account2
thread2: transfer money from account2 to account1
```

则在这种情况下，thread1 可能会锁定 account1（并申请对 account2 加锁。——译者注），而与此同时 thread2 则可能锁定了 account2（并申请对 account1 加锁。——译者注）。于是两个线程就形成了死锁，即在不释放己方已经持有的锁的情况下，等待对另一个线程所持有的账户进行加锁。为了解决这一问题，我们可以令两个线程都必须按照相同的顺序请求锁。这样一来，其中一个线程就能够获得全部两把锁，而另一个线程则可以在经过短暂的阻塞之后继续执行，这样就可以避免双方同时进入死锁状态的困扰。最后，由于 Account 类实现了 Comparable 接口，所以线程可以很方便地利用账户间的自然顺序作为加锁的顺序（参见 Brian Goetz 的《Java Concurrency in Practice》[Goe06] 一书中关于死锁避免的讨论）。

通过用 ReentrantLock 来获得锁和释放锁，我们确保了 deposit() 和 withdraw() 操作是互斥的。另外，某些类似于转账这样的操作可能需要把多个存款和取款的动作组合起来才能形成一个完整的互斥操作。为了实现这一功能，Account 类对外暴露了其内部定义的可重入锁的实例。

下面我们将在 AccountService 类的 transfer() 函数中实现具体的转账功能。我们首先要对给定的两个账户进行加锁，其中加锁的顺序是由账户的 compare() 函数比较而得的自然顺序来确定的。一旦没能在所期望的时间点对所需账户进行加锁，我们就会简单地抛出一个 LockException 异常。如果加锁成功且资金充裕，我们就能够顺利地完成整个转账操作。

scalabilityAndTreadSafety/locking/AccountService.java

```
public class AccountService {
  public boolean transfer(
    final Account from, final Account to, final int amount)
    throws LockException, InterruptedException {
    final Account[] accounts = new Account[] {from, to};
    Arrays.sort(accounts);
    if(accounts[0].monitor.tryLock(1, TimeUnit.SECONDS)) {
      try {
        if (accounts[1].monitor.tryLock(1, TimeUnit.SECONDS)) {
          try {
            if(from.withdraw(amount)) {
              to.deposit(amount);
              return true;
            } else {
              return false;
            }
          } finally {
            accounts[1].monitor.unlock();
          }
        }
      } finally {
        accounts[0].monitor.unlock();
      }
    }
    throw new LockException("Unable to acquire locks on the accounts");
  }
}
```

通过预先对账户进行排序，transfer() 函数有效避免了线程死锁问题，同时，为了避免线程陷入无限期的等待状态（即线程被饿死），transfer() 函数对每个加锁操作的等待时间都做了限制。由于我们使用的 monitor 是可重入锁，所以在后续 deposit() 和 withdraw() 里对 lock() 函数的调用不会产生任何不良影响。

虽然 transfer() 函数有效地避免了死锁和线程饥饿（starvation）问题，但是 deposit() 函数和 withdraw() 函数所使用的 lock() 可能会在其他上下文环境中出现问题。作为一个练习，我建议你将 deposit() 和 withdraw() 方法中的 lock() 改为 tryLock()，并观察一下执行效果。

虽然我们在本章所使用的方法可能与你之前的习惯用法不尽相同，但是为了能够更好地控制锁的行为，建议你最好还是用 Lock 接口及其相关辅助类来代替原始的 synchronized

关键字。

虽然 Lock 接口的设计非常清晰易懂，但真正用起来还是需要花一番工夫并且还很容易犯错。在后面的章节中，我们将实现一个非常有价值的设计目标，完全避免使用锁 / 同步动作，即真正实现显式的锁无关（lock-free）并发。

4.7　小结

现代的并发 API，即 java.util.concurrenr 对旧的 API 进行了相当多的改进。新 API 可以帮助程序员做到如下几点：

- 管理一个线程池
- 方便地为并发执行调度任务
- 以线程安全的方式在多个并发执行的线程之间进行数据交换
- 获得细粒度的同步能力
- 通过使用并发数据结构，可以使程序获得更好地并发执行性能

虽然新的 API 非常有用，但我们仍然需要注意避免竞争条件和使用共享可变数据。

在下一章里，我们将讨论一些解决共享可变性的方法。

第5章

驯服共享可变性

截至目前，我在前面的章节中已经多次谈到避免使用共享可变性设计的问题。你可能会问，既然不推荐使用，为何还要在本章讨论这个议题？原因很简单：这部分内容是Java领域过去常用的编程方式，而且即便你自己不这么用，但你也很可能在接手一些遗留代码时遇到这种用法。

我衷心希望你已经非常乐于在所有新代码，甚至是正在开发的项目中使用隔离可变性和完全不可变性设计方法。我写本章的目的主要是帮助你更好地处理那些遗留代码，这些代码的问题一定很多，你该考虑重构一下了。

5.1 共享可变性 != Public

共享可变性并非仅仅局限于那些public字段。你可能会认为“我把代码里所有字段都声明为private，总该没啥好担心的了吧”，但事实上却并非如此简单。

如果一个变量可以被多个线程读写，则我们说该变量是可访问的且共享的。另一方面，如果一个变量仅能被一个线程访问，我们就称该变量是隔离的且非共享的。如果我们没处理好变量的可见性或竞争条件（race condition），那么共享可变变量肯定会把我们的程序搞得乱七八糟。甚至有传言说共享可变性是导致Java程序员失眠的罪魁祸首。

在不考虑访问权限的情况下，我们必须保证所有被当成参数传递给其他函数的那些变量都是线程安全的。我们必须假定，我们所调用的那些函数将会从多个线程里访问这些参数。因此，向函数传递一个非线程安全的变量将不会有助于提高你的夜间睡眠质量。同时，对于函数的返回值也存在同样的问题。换句话说，请不要放过任何非线程安全的引用。关于如何才能做到一个不漏地处理这些引用，更详细内容请参阅《Java Concurrency in Practice》[Goe06]。

如何发现遗漏的非线程安全的引用是一个棘手的问题，除非仔细检查手头的代码，否则在平时我们可能根本就意识不到有遗漏。除了传递和返回的引用之外，那些被我们直接塞到其他对象或静态字段里的引用也可能造成非线程安全的变量被漏过。如果我们把变量放入集合类中，比如之前讨论过的BlockingQueue，则该变量也有可能被漏过。所以，当你下次打开使用了共享可变变量的代码时发现脖子后面汗毛倒竖，请不要感到惊讶。

5.2 定位并发问题

让我们通过一个例子来进一步认识共享可变性设计中存在的风险，然后再看看如何解决这些问题。下面我们将对一段控制一个高档电源的代码进行重构。该代码允许用户使用

电池里的能量，并定期地对电源进行自动充电。让我们先简单过一下这段急需我们帮着改进的悲催代码：

tamingSharedMutability/originalcode/EnergySource.java

```
//Bad code
public class EnergySource {
  private final long MAXLEVEL = 100;
  private long level = MAXLEVEL;
  private boolean keepRunning = true;

  public EnergySource() {
    new Thread(new Runnable() {
      public void run() { replenish(); }
    }).start();
  }

  public long getUnitsAvailable() { return level; }

  public boolean useEnergy(final long units) {
    if (units > 0 && level >= units) {
      level -= units;
      return true;
    }
    return false;
  }

  public void stopEnergySource() { keepRunning = false; }

  private void replenish() {
    while(keepRunning) {
      if (level < MAXLEVEL) level++;

      try { Thread.sleep(1000); } catch(InterruptedException ex) {}
    }
  }
}
```

看完 EnergySource 类的代码之后，我们首先要能识别出里面存在的并发问题。其中有些问题是显而易见的，但也有一些隐藏的宝藏正等待我们去挖掘。不过请别着急，下面就让我们挖出这些问题并逐一进行剖析。

准备好了没？好的，让我们开始吧。首先我们应该知道，EnergySource 中定义的方法可能会被多个线程同时调用，所以 EnergySource 的私有类变量 level 就成为一个非线程安全的共享可变变量。从线程安全的角度来说，我们在大多数方法中都没有对该变量的访问进行保护。所以，这种做法既可能会产生变量可见性问题又可能会导致竞争条件。其中变量可见性问题是指：由于我们并未强制要求线程在访问变量时一定要穿越内存栅栏，所以调用这些方法的线程可能无法及时看到 level 值的变化。

这些都是显而易见的，但其实还有很多其他问题。

虽然 replenish() 函数在绝大部分时间都处于睡眠状态，但是在上述代码中我们还是得浪费一个线程在它身上。如果我们试图创建大量的 EnergySource 对象，由于 JVM 通常只

允许我们创建几千个线程，所以程序必将因为创建了过多线程而抛出 OutOfMemoryError 错误。

EnergySource 的实现破坏了类不变式（class invariant）[⊖]。一个精心构建的对象可以保证，在其自身没有恢复到有效状态之前，它的任何方法都不能被访问。而 EnergySource 构造函数就违反了不变式，因为在 EnergySource 实例对象的构造函数还未完成之前就可能有其他线程调用 replenish()，从而导致了 level 值的变化。同时，Thread 类的 start() 函数会自动插入一个内存栅栏（start() 函数是 synchronized 的。——译者注），于是 Thread 类的对象在 EnergySource 的实例对象初始化完成之前就脱离了其控制范围。正如我们下一节中将要讨论的那样，在构造函数中启动线程实在是一个糟糕的主意。

在这么一小段代码里我们就发现了不少问题，是吧？下面就让我们来一一修复这些问题。但鉴于我个人不太喜欢同时解决多个问题，所以我们将采取集中优势兵力逐个击破的方式来进行重构。

5.3　保持不变式

我们通常希望后台任务在对象初始化期间就能够开始运行，所以在构造函数中启动线程对我们来说是很有吸引力的解决方案。虽然我们的设计意图很好，但该方案却存在着很多不良的副作用。当我们在构造函数中调用 Thread 类的 start() 函数时，Thread 类会自动插入一个内存栅栏，并将尚未构造完整的对象曝露给其他线程。同样，我们在构造函数中创建的那个线程可能会在创造它的对象尚未构造完成之前就调用其成员方法。

为了避免上述问题，我们需要严格保持对象的不变式，因此请注意千万不要在构造函数里启动线程。

EnergySource 的实现明显违背了上述原则，所以我们需要将线程的启动代码从构造函数里搬到一个独立的函数当中。然而这种做法又会带来很多新问题：我们需要正确处理那些在线程启动之前就到达的函数调用，此外，某些马虎的程序员还可能会干脆忘记调用那个线程启动函数。虽然我们可以通过在代码中加标记的方法来解决这些问题，但同时也会导致产生很多丑陋的重复代码。最后，我们还得避免同一个实例上的线程启动函数被多次重复调用的问题。

一方面，我们绝不应该从构造函数中启动线程；而另一方面，我们也不愿意在对象没有完全创建好之前就启用它。所以我们必须找到一种两全其美的解决方法。

解决方案就在《Effective Java》[Blo08] 一书的第一项中：“考虑使用静态工厂函数来代替构造函数”。即在一个静态工厂函数中创建对象实例，并在将实例的引用返回给调用方之前启动线程。

⊖ 类不变式是每一个类的对象在任何时候都要遵守的一个条件，请参阅《What Every Programming Should Know About Object-Oriented Design》[Pag95] 和《Object-Oriented Software Construction》[Mey97]。换句话说，我们绝对不要去访问一个处于无效状态中的对象。

tamingSharedMutability/fixingconstructor/EnergySource.java

```
//Fixing constructor...other issues pending
private EnergySource() {}

private void init() {
  new Thread(new Runnable() {
    public void run() { replenish(); }
  }).start();
}

public static EnergySource create() {
  final EnergySource energySource = new EnergySource();
  energySource.init();
  return energySource;
}
```

在上述代码中，我们通过将缺省构造函数置为 private，实现了既能在构造函数中进行一些简单的运算，又能在初始化过程中免受外部方法调用的打扰。其中，私有方法 init() 实现了前一版本的 EnergySource 构造函数的绝大部分功能。通过在静态工厂函数 create() 中调用该函数，我们就可以避免所有违反不变式的情况发生。与此同时，这种方法也可以保证后台任务能够在对象创建完成之后启动，从而保证了对象在创建和初始化过程中一直处于有效状态。

请检查一下你手头的项目，其中有没有在构造函数中启动线程的？如果有，那么你就需要在你的重构任务列表中再添加一个清理任务了。

5.4　管理好资源

由于线程资源是有限的，所以我们的创建行为就不能太过随意。每个 EnergySource 的对象实例都会在 replenish() 函数上浪费一个线程，这将大大限制我们在整个程序中可创建的实例数量。如果其他实例也都像这样创建只属于他们自己的线程的话，那么我们很快就会遇到资源枯竭的问题。通过代码我们可以看到，这个充电操作是既（代码）短又（执行）快的，所以将其放在一个 timer 里运行应该是个不错的选择。

timer 方面我们可以选用 java.util.Timer。为了能获得更大的吞吐量，尤其是在需要创建大量 EnergySource 实例的情况下，我们最好还是通过线程池来实现线程复用。值得高兴的是，ScheduledThreadPoolExecutor 提供了一个用于执行周期性任务的优雅方案。需要注意的是，我们必须确保每个任务都能处理好方法里抛出的异常。否则，那些把异常丢到外层的周期任务将会被停掉。

让我们重构 EnergySource，将 replenish() 函数改成在一个周期任务中执行。

tamingSharedMutability/periodictask/EnergySource.java

```
//Using Timer...other issues pending
public class EnergySource {
  private final long MAXLEVEL = 100;
  private long level = MAXLEVEL;
  private static final ScheduledExecutorService replenishTimer =
```

```
    Executors.newScheduledThreadPool(10);
  private ScheduledFuture<?> replenishTask;

  private EnergySource() {}

  private void init() {
    replenishTask = replenishTimer.scheduleAtFixedRate(new Runnable() {
      public void run() { replenish(); }
    }, 0, 1, TimeUnit.SECONDS);
  }

  public static EnergySource create() {
    final EnergySource energySource = new EnergySource();
    energySource.init();
    return energySource;
  }

  public long getUnitsAvailable() { return level; }

  public boolean useEnergy(final long units) {
    if (units > 0 && level >= units) {
      level -= units;
      return true;
    }
    return false;
  }

  public void stopEnergySource() { replenishTask.cancel(false); }

  private void replenish() { if (level < MAXLEVEL) level++; }
}
```

重构后的代码除了在资源使用上更加合理之外，其结构也变得更简单。我们去掉了 keepRunning 字段，并可以在 stopEnergySource() 函数中直接停掉周期任务。在新的 init() 函数中，我们不再为每个 EnergySource 的实例都启动一个线程，而是调度了一个 timer 来定期执行 replenish() 函数。由此，init() 函数也变得更加简单，我们可以不再关心线程的休眠或计时调度问题，从而能够将注意力集中到增加能量水平（energy level）的逻辑中去。

通过将 replenishTimer 的引用声明为 static 字段，在多个 EnergySource 实例上执行的 replenish() 操作就可以共享同一个线程池了。我们可以根据需要调整线程池内的线程数量。在本例中，根据定时任务的周期和 EnergySource 实例的数量，我们暂定线程数为 10。由于 replenish() 任务非常轻量，所以这样一个小线程池就足够用了。

虽然使 replenishTimer 的引用声明为 static 可以帮助我们在 ScheduledThreadPoolExecutor 中共享线程池里的线程，但这种做法同时也增加了一点点代码的复杂性：即我们必须想出一个将 replenishTimer 关闭的办法。默认情况下，执行线程都是非守护线程，并且如果我们没有显式将其关闭的话，JVM 是不会替我们停掉这些线程的。至少有两种方案[⊖]可以帮助我们解决这个问题：

⊖ Google 的 Guava API（http://code.google.com/p/guava-libraries/）提供了 JDK 并发 API 的很多方便的包装器，其中包括了一个可以创建自动结束的线程池的方法。

- 在 EnergySource 类中提供一个静态方法，例如：

```
public static void shutdown() { replenishTimer.shutdown(); }
```

但这种做法有两个问题。一是 EnergySource 的用户得记着要调用这个方法。二是我们必须添加一些逻辑，以便在调用 shutdown() 之后处理那些已创建的 EnergySource 实例。

- 给 newScheduledThreadPool() 函数传一个额外的 ThreadFactory 参数。该线程工厂能够保证其创建的所有线程都是守护线程，例如：

```
private static final ScheduledExecutorService replenishTimer =
  Executors.newScheduledThreadPool(10,
      new java.util.concurrent.ThreadFactory() {
    public Thread newThread(Runnable runnable) {
      Thread thread = new Thread(runnable);
      thread.setDaemon(true);
      return thread;
    }
  });
```

这个方法的主要缺点是需要多敲几行代码且稍微增加了一些维护成本。

如此折腾一番之后，我们的 EnergySource 代码更加简单清晰，并且比我们在代码中自己创建线程更具可扩展性。

请检查一下你手头的项目，看一下你都是在哪里创建线程的，特别是那些使用了 Thread 类的地方。然后评估一下这些地方是否能够像我上面所做的那样用周期任务调度器来替代。

5.5　保证可见性

在程序开发过程中，如何让线程在合适的时间跨越内存栅栏是很重要的事情。我们在前面例子的重构过程中已经提到过，在构造函数中跨越内存栅栏是不合适的。然而我们又必须保证访问共享可变变量 level 的其他方法也要跨越内存栅栏（即对 level 的访问进行同步。——译者注）。请参阅前面第 1.3 节的内容来了解为何要跨越内存栅栏，也许你会改变对这个问题的印象。

如果只考虑竞争条件的话，我们可能会反对使用同步的 getter 方法，同时我们可能会觉得，在读 level 变量的时候即使拿到稍旧一点的值似乎也问题不大。但是，使用以 synchronized 修饰的或在内部加了锁的 getter 方法的主要目的是为了在存在竞争条件的情况下保证变量的可见性。所以，如果没能正确处理好跨越内存栅栏的问题，我们就无法保证所有线程在未来任意时间段内都能及时看到变量值的变化。

有很多方法可以保证让 EnergySource 的函数都跨越内存栅栏，其中最简单办法是在所有方法前面都加上 synchronized 关键字。虽然有点简单粗暴，但我们还是先通过这种方式来保证变量的可见性，然后再根据其缺点进行改进，我的座右铭是“先跑起来，然后再优化”。

所有与读写共享可变变量 level 有关的方法都需要跨越内存栅栏，所以我们将会把所有相关方法都用 synchronized 进行修饰。在最初的版本中，由于其中包含一个无限循环，所以 replenish() 是不能用 synchronized 进行修饰的。但最新的版本则不存在这个问题，因为

新版代码中该方法是一个由 timer 执行的、不存在内部循环的短方法。下面就是可以保证变量可见性的新版代码：

tamingSharedMutability/ensurevisibility/EnergySource.java

```
//Ensure visibility...other issues pending
  //...
  public synchronized long getUnitsAvailable() { return level; }

  public synchronized boolean useEnergy(final long units) {
    if (units > 0 && level >= units) {
      level -= units;
      return true;
    }
    return false;
  }
  public synchronized void stopEnergySource() {
    replenishTask.cancel(false);
  }

  private synchronized void replenish() { if (level < MAXLEVEL) level++; }
}
```

在上面的代码中，我们在 getUnitsAvailable()、useEnergy() 和 replenish() 函数前面加了 synchronized 关键字，因为这些函数需要访问 level 变量了。虽然理想情况下我们期望只有一个线程会调用 stopEnergySource() 函数，但是这里我们还是给它先加上 synchronized 再说，权当可能会有多个线程调用这个函数。不管怎么说，通过上面的修改我们还是实现了线程安全的变量访问。

请检查一下你手头的项目，看看是否所有对可变变量的访问（包括 getter 和 setter）都用了 synchronized 或其他某种相关的构造来跨越内存栅栏。访问由 final 修饰的不可变变量则无需穿越内存栅栏，因为这些变量的值是不会变的，并且（CPU）缓存的值和内存里的值也都是完全相同的。

5.6　增强并发性

为了保障自己的人身安全，我可以在我的房子周围挖条护城河，再扔进去几条鳄鱼。但是这样一来，我每天回家和出门就成了大问题。过于保守的同步策略就类似于上述做法，虽然可以保证线程安全，但效率很低。我们希望对于每个类的同步动作都能保持合适的粒度，这样我们就能做到既不损害线程安全又可以享有更好的并发性。

使用 synchronized 对整个实例对象进行同步是一种非常普遍的做法，但这种做法有很多问题。其中一个问题是，synchronized 关键字的作用域是整个对象，于是整个程序的并发粒度就被限死在对象级别上，在任意时刻，一个对象最多只能接受一个同步操作。如果对象上的所有操作（例如在一个集合中添加或删除数据等）都是互斥的，那性能可能还不算特别差，虽然这其中肯定还有很大的改进空间。然而如果对象支持多个可以并发执行的操作（如 drive() 和 sing()），并且这些操作需要与其他互斥操作（如 drive() 和 tweet()）进行

同步，那么对象实例级别的同步将会对程序执行速度产生很大的影响。在这种情况下，我们就需要在对象的相关方法中创建多个同步点，用细粒度的同步控制来提高并发执行速度。

EnergySource 类有很多地方都是需要同时保证可见性和线程安全的，但如果采用 synchronized 关键字的话又显得有些过于保守。因为在本例的情况下，实在是没必要采用对象实例级别的同步。下面让我们来修复这个问题。

由于变量 level 是 EnergySource 类里唯一的可变字段，所以我们可以将同步操作直接作用于其上。但这种方法并不总是能奏效，因为如果某个类里面定义了多个字段，那么我们就需要对这些字段的访问都进行保护。所以，正如我们即将看到的那样，我们可能需要在 EnergySource 的代码中定义多个显式的 Lock 实例来应对这种情况。

上面我们已经分析过，将同步操作直接作用到变量 level 上是可行的。但这个方案有个小问题，即 Java 是不允许对像 long 这样的基础类型加锁的，所以我们需要将 level 变量的类型从 long 改为 AtomicLong 来规避这个限制。如此一来，我们就可以针对该变量的访问实现细粒度的线程安全了。在进行更深入的讨论之前，让我们先看一下修改后的代码：

tamingSharedMutability/enhanceconcurrency/EnergySource.java

```
public class EnergySource {
  private final long MAXLEVEL = 100;
  private final AtomicLong level = new AtomicLong(MAXLEVEL);
  private static final ScheduledExecutorService replenishTimer =
    Executors.newScheduledThreadPool(10);
  private ScheduledFuture<?> replenishTask;

  private EnergySource() {}

  private void init() {
    replenishTask = replenishTimer.scheduleAtFixedRate(new Runnable() {
      public void run() { replenish(); }
    }, 0, 1, TimeUnit.SECONDS);
  }

  public static EnergySource create() {
    final EnergySource energySource = new EnergySource();
    energySource.init();
    return energySource;
  }

  public long getUnitsAvailable() { return level.get(); }

  public boolean useEnergy(final long units) {
    final long currentLevel = level.get();
    if (units > 0 && currentLevel >= units) {
      return level.compareAndSet(currentLevel, currentLevel - units);
    }
    return false;
  }

  public synchronized void stopEnergySource() {
    replenishTask.cancel(false);
  }
```

```
    private void replenish() {
            if (level.get() < MAXLEVEL) level.incrementAndGet();
    }
}
```

由于 AtomicLong 自身就能保证对其所持有的值在多线程并发访问环境下的可见性和线程安全性，所以我们去掉了 getUnitsAvailable() 函数前面的 synchronized 修饰符。

基于同样的理由，我们也去掉了 useEnergy() 函数前面的 synchronized 修饰符。然而，这一做法在改进了并发性的同时还带来了一些语义上的变化。在之前的版本中，在检查剩余可用电量时，我们是锁住了 level 变量不让其他任何线程访问的。所以只要发现有足够的电量，我们就一定能够获取得到。然而这种做法会严重降低并发度，即当一个线程正在进行操作的时候，所有与 EnergySource 类相关的其他交互操作都将被阻塞。在这个改进版的代码中，多个线程可以在没持有互斥锁的情况下同时竞争（使用）电源的电量。如果两个或多个线程同时更改 level 的值，则只有一个请求会成功，而其他的更改请求则需要重试。通过上述改进，我们既加快了读操作的速度，同时又增强了写操作的安全性。

replenish() 函数同样不需要使用互斥锁。该函数先通过线程安全的方式取到了 level 的值，然后在不需要持有任何锁的情况下将其值自增 1 个单位。这是没有问题的，因为只有一个线程增加 level 的值。只要该函数发现 level 的值小于 MAXLEVEL，这个操作就会一直进行下去。同时，由于使用了 incrementAndGet() 函数，所以这个自增的操作是线程安全的并且不会有任何一致性方面的问题。

由于程序中是极少调用 stopEnergySource() 方法的，因此没必要在这个点上做更细粒度的锁控制，所以我们最终还是保留了 stopEnergySource() 函数前面的 synchronized 修饰符。在后面的第 6.8 节中，我们将会看到该方法的另外一种实现方式。

请复查一下你手头的项目，并找到那些需要既不能损失线程安全性又需要提高并发度的地方。请检查一下这些地方是否可以用锁对象来代替针对整个对象实例粒度的 synchronized 关键字。在替换的同时，请注意保证所有参与读写可变状态的方法都进行了恰当的同步。

5.7　保证原子性

在上例中，我们在代码里没有使用任何显式的同步操作。但是请先别高兴得太早，因为如果程序中与可变状态相关或依赖的变量不止一个，那么我们就无可避免地要使用显式的同步操作。幸运的是，JDK 的并发 API 已经包含了显式同步所需的工具和方法，下面就让我们研究一下如何用并发 API 来解决多个变量的同步问题。

到目前为止，我们的重构都达到了预想的效果，但我们还要向更高要求的目标迈进：追踪并记录电源的使用情况。即每次电源电量消耗完的时候，我们都需要把电源的使用次数进行累加。这就意味着，我们必须保证对于变量 level 和 usage 的改动是原子的。换句话说，如果要在同一个线程里修改这两个变量，我们需要保证：要么两个变量全部修改成功、要么两个都不改。我们绝不希望看到只有其中一个改成功了，而另一个没改成功的情况。

为了满足上述需求，我们需要在代码中引入显式同步。用 synchronized 关键字当然是最简单的，但是在有多个读者的情况下会严重影响程序整体的并发度。如果不介意一个读者的操作阻塞所有其他读者运行的话，那么使用 synchronized 关键字方案自然是轻松愉快的。但是如果我们想要提供更好的并发度并且不介意因此带来的些许复杂性的话，则最好还是使用 ReentrantReadWriteLock。该类同时提供了两把锁，即读锁和写锁，读者和写者可以根据需要分别使用这两把锁。用了 ReentrantReadWriteLock 之后，我们就可以让多个并发的读者或一个互斥的写者在同一个实例对象上进行操作。下面就让我们着手修改之前的例子代码。由于使用了显式的锁，所以我们可以把 level 字段由 AtomicLong 改回成简单的 long 类型。

tamingSharedMutability/ensureatomicity/EnergySource.java

```
public class EnergySource {
  private final long MAXLEVEL = 100;
  private long level = MAXLEVEL;
  private long usage = 0;
  private final ReadWriteLock monitor = new ReentrantReadWriteLock();
  private static final ScheduledExecutorService replenishTimer =
    Executors.newScheduledThreadPool(10);
  private ScheduledFuture<?> replenishTask;

  private EnergySource() {}

  private void init() {
    replenishTask = replenishTimer.scheduleAtFixedRate(new Runnable() {
      public void run() { replenish(); }
    }, 0, 1, TimeUnit.SECONDS);
  }

  public static EnergySource create() {
    final EnergySource energySource = new EnergySource();
    energySource.init();
    return energySource;
  }

  public long getUnitsAvailable() {
    monitor.readLock().lock();
    try {
      return level;
  } finally {
    monitor.readLock().unlock();
  }
}

public long getUsageCount() {
  monitor.readLock().lock();
  try {
    return usage;
  } finally {
    monitor.readLock().unlock();
  }
}
```

```
  public boolean useEnergy(final long units) {
    monitor.writeLock().lock();
    try {
      if (units > 0 && level >= units) {
        level -= units;
        usage++;
        return true;
      } else {
        return false;
      }
    } finally {
      monitor.writeLock().unlock();
    }
  }

  public synchronized void stopEnergySource() {
    replenishTask.cancel(false);
  }

  private void replenish() {
    monitor.writeLock().lock();
    try {
      if (level < MAXLEVEL) { level++;  }
    } finally {
      monitor.writeLock().unlock();
    }
  }
}
```

在上面的代码中，我们引入了两个新字段：usage 和 monitor。其中，usage 的作用是记录电源被使用的次数；而 monitor 则是 ReentrantReadWriteLock 的一个实例，其作用是保证线程对于两个可变变量的修改是原子的。在 useEnergy() 函数中，我们先获得了写锁，如果需要的话还可以在获得锁的时候指定一个超时时间。一旦获得写锁的操作完成，我们就可以更改 level 和 usage 两个变量的值了。最后，在方法结尾处的 finally 块中，我们会安全地将锁释放掉。

类似地，我们在 replenish() 函数中也需要获得写锁，因为这样就可以保证 useEnergy() 和 replenish() 对于变量 level 的修改是互斥的。

在针对变量的 get 方法中，我们需要锁以保证所读字段的可见性，而在加了锁之后，useEnergy() 函数内部执行过程中对于变量的部分更改（即两个变量只改了一个、另一个还没改的中间状态。——译者注）也不会被其他线程看到。通过使用读锁，我们可以使多个读操作并发地执行，并只在有写操作的时候读者才会被阻塞。

上述代码不但实现了相当好的并发度，同时还能兼顾到线程安全。但我们还是必须时刻保持警惕，才能保证不会出现线程安全方面的纰漏。遗憾的是，这一版的代码比之前的版本复杂了不少，而这些复杂性正是错误滋生的温床。在下一章，我们将会学习如何避免使用显式的同步操作。

5.8　小结

处理共享可变状态对于程序员来说是一个相当沉重的负担，因为这通常需要我们在忍辱负重地往程序里增加很多复杂逻辑，在保证线程安全和提高执行效率的同时还要承受这些复杂性所带来的巨大的出错的风险。所以当我们重构代码的时候，请注意规避一些常见的并发相关的错误做法：

- 请在静态工厂方法而不是构造函数中创建线程，详情请参阅第 5.3 节。
- 请不要随意创建线程，而是使用线程池来降低任务启动时间和资源消耗，详情请参阅第 5.4 节。
- 确保对可变字段的访问跨越内存栅栏，并对线程可见，详情请参阅第 5.5 节。
- 通过正确评估程序所需要的锁的粒度来提高程序的并发度。请确认你当前所使用的锁是否过于保守，如果是的话请将其调整为合适的粒度，以便能够同时满足程序对线程安全和并发度的双重需求，详情请参阅第 5.6 节。
- 当需要同时使用多个可变字段时，请核实对这些变量的访问是否是原子的，也就是说，要保证其他线程不会看到这些变量的部分修改结果（partial changes），详情请参阅第 5.7 节。

在本章中，我们已经见识过显式同步是多么的难用。而在下一章，我们将讨论如何采用一种对于 JVM 来说相对较新的方法来彻底消除显式同步。

第三部分
软件事务内存

第⑥章
软件事务内存导论

请回忆一下你最近完成那个需要对共享可变变量进行同步的项目。在那个项目中，你肯定无法身心愉悦地享受出色地完成工作所带来的乐趣，而是会陷入无尽的质疑之中并抓狂地挨个确认是否在所有需要的地方都作了适当的同步。在过去所经历过的编程工作中，我已经遇到过好几次这样令人神经衰弱的情况了，而其中绝大部分原因都是由于我们在用Java编程的时候没有遵循正确原则和方法来处理共享可变状态。如果我们在某个该同步的地方忘了进行同步，那些不可预知的、潜在的灾难性的结果就将在不远处等待着我们。但是人无完人，遗忘是我们的天性。所以我们应该充分利用工具来弥补我们自身的不足，同时也可以让工具帮助我们实现我们充满创意的大脑所追求的那些伟大的目标，而不是让错误一次次地打击我们的信心。为了能够得到可预测的行为和结果，我们需要再次把目光投向JDK。

在本章中，我们将会通过使用Clojure中十分流行的软件事务内存（STM）模型来学习如何线程安全地处理共享可变性。在需要的时候，我们可能会在示例项目中混入Clojure的代码。但是我们并非强迫你也要使用Clojure，因为随着Multiverse和Akka这些优秀工具的出现，我们也可以在Java中直接使用STM了。在本章中，我们会先来看看STM在Clojure里是什么样子，然后再学习如何用Java和Scala对事务内存进行编程。这种编程模型非常适用于那些读多写少的程序，它简单易用并能提供可预测的结果。

6.1 同步与并发水火不容

同步操作本身就存在一些很基本的缺陷。

如果我们没能处理好或干脆就忘了进行同步，则某个线程所做的更改可能无法被其他线程及时可见。此外，为了同时保证可见性并避免竞争条件，我们还需要通过一些很麻烦的手段来进行同步操作。

不幸的是，当我们执行同步操作的时候，同时也强制了其他竞争相同资源的线程只能等待。由于同步的粒度对并发度是有很大影响的，所以将同步控制的工作完全交由程序员来完将成会大大降低程序整体效率并增加错误发生的概率。

同步操作还可能引发很多活跃度方面的问题。由于某个线程可能本身持有锁，同时还在等待其他锁，所以很容易造成程序死锁。此外，同步还很容易造成活锁（livelock）问题，即线程可能会在申请某一把锁的时候不断遭遇失败。

当然，我们可以尝试使用细粒度的锁来提高程序并发度。虽然一般来说这个主意还不错，但是其中最大的风险是程序员可能没在合适的级别进行同步动作。更糟的是，同步出问题的时候我们还收不到任何提示。此外，因为需要互斥访问的线程加了锁之后还是会阻塞其他线程的访问请求，所以细粒度的锁只是把线程等待的位置换了个地方而已。

熟练掌握 JDK 并发工具包的 Java 程序员在大城市里一般表现都不错。而且由于这么长时间以来，我们在处理可变状态的编程方面都没能找到一个比同步更合适的替代产品，所以导致了我们在这方面的预期一直在不断下降。但是新的编程模型终于还是到来了！

6.2　对象模型的缺陷

作为一个 Java 程序员，我们对面向对象的编程（OOP）自然都是烂熟于胸的，但语言也极大地影响了我们对面向对象应用程序的建模方式。现在的 OOP 已经和 Alan Kay 当初创造这个词时候的初衷大不相同了。他的主要思想是采用消息传递并消灭所有数据（他认为，系统是由一些类似于生物细胞那样的对象构成的，这些对象通过消息传递进行通信，且无需持有任何状态），见附录 2 中《The Meaning of Object-Oriented Programming》一书。随着这一技术的演进，面向对象的语言开始朝着通过抽象数据类型（ADT）来实现数据隐藏（data hiding）的方向发展，并将数据和处理过程绑定或将状态与行为组合在一起。这在很大程度上引领我们走向封装和不断变化的状态。在这个过程中，我们最终还是把状态与实体（identity）进行了融合，即把对象实例与其数据整合在一起。

对于 Java 程序员来说，实体与状态的融合是在潜移默化间悄悄完成的，其结果可能不是很明显。当我们顺着指针或引用找到某个实例的时候，实际上是登录到了持有其状态的一块内存上，于是在那个位置上操纵数据也就成了自然而然的事了。该位置即代表了对象实例及其所包含的数据。将实体与状态进行合并最初看起来是非常简单且易于理解的，但从并发的角度来看，这种做法其实有很多严重的不良后果。

例如，如果我们需要实现一个打印银行账户详情（资金数量、当前余额、交易信息、最小余额等）的程序，我们就会碰到很多并发相关的问题。你会发现手头待处理的引用其实是一个随时都可能发生变化的状态的代理。所以当我们查看账户信息的时候，就需要通过加锁来阻止其他线程对账户内容进行修改，而这也必将导致并发度的大幅下降。但问题

并不是从加锁的那一刻才开始出现的，而是在我们把账户的实体与其状态合并的时候就已经存在了。

我们曾经被告知说面向对象的编程是对真实世界的建模。但悲催的是，真实世界与 OO 范式所试图构建的模型实际是大相径庭的。因为在真实的世界中，状态是不变的，而实体却是不断变化的。接下来我们将讨论这种说法为何是正确的。

6.3　将实体与状态分离

你能快速告诉我 Google 的股价现在是多少吗？我们当然可以说从证券市场开市的那一刻起股价就是在不断变化的，但这只不过是一种文字游戏罢了。举一个简单的例子，2010 年 12 月 10 日 Google 的收盘价是 592 美元，并且这个数字已经被载入史册、是不可变的了。而我们所要查找的只是 Google 股价当时的一个快照。当然，Google 今天的股价和那天已经完全不同了。而如果过几分钟之后再来查看 Google 的股价（假设证券市场是开市的），我们就会看到一个不一样的值，但之前的那个值其实并没有改变。从现在开始，我们得改变一下我们对对象的认识，而这也将同时改变我们使用对象的方式。后面我们会看到，把对象的实体与其不可变状态值进行分离的做法将如何帮助我们实现锁无关（lock-free）编程、提高并发度、同时最大程度地降低竞争。

将实体与状态分离绝对是一个天才的构想，这是 Rich Hickey 在其实现 Clojure 的 STM 模型过程中所采用的一个非常关键的步骤，详情请见附录 2 中。假定我们的 Google 股票对象由两部分组成：第一部分用于表示该股的实体，其中包含一个指向第二部分的指针；第二部分则包含了该股最新股价，其中保存股价的变量即为不可变状态，如图 6-1 所示。

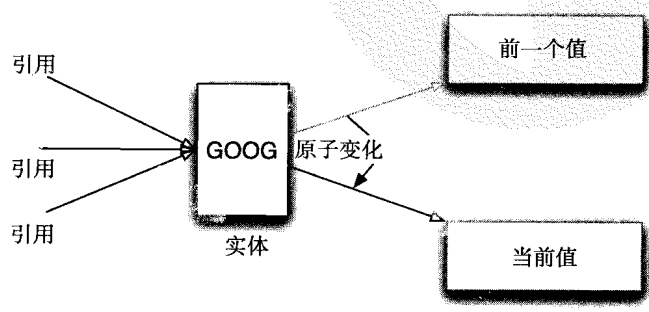


图 6-1　将可变实体部分与不可变状态值进行分离

一旦接收到一个新的股价信息，我们就可以在不更改任何已存在事务的情况下将其放入历史价格指数中。由于旧的股价是不可变的，所以我们可以将其共享出去供所有线程访问。正如我们在 3.6 节中所讨论的那样，如果我们在这里采用持久化数据结构的话，则 Google 股票对象就不需要进行复制，可以多快好省地对外提供数据读取服务。而一旦有新的数据准备就绪之后，我们可以快速更改实体中的指针，以使其指向保存新股价的字段。

实体与状态分离的做法对于并发来说也是一大福音。因为采用了这种方法之后，我们

就可以不用阻塞任何查询股价的请求了。由于状态是不会变的，所以我们可以欣然将其指针传递给发出查询请求的线程。所有在我们更新实体（内部的指针。——译者注）之后到达的查询请求都可以看到更新后的股价。我们知道，非阻塞的读操作即意味着更高的并发度，所以我们只需要确保每个线程都能获得一致的视图即可。而这其中最棒的是，我们其实什么也不用做，STM 已经帮我们都搞定了。相信你已经迫不及待想要了解更多关于 STM 的知识了吧？下面就让我们一起来学习这方面的内容。

6.4　软件事务内存

将实体与状态分离的做法有助于 STM（软件事务内存）解决与同步相关的两大主要问题：跨越内存栅栏和避免竞争条件。让我们先来看一下在 Clojure 上下文中的 STM 是什么样子，然后再在 Java 里面使用它。

通过将对内存的访问封装在事务（transaction）中，Clojure 消除了内存同步过程中我们易犯的那些错误（见《Programming Clojure》[Hal09] 和《The Joy of Clojure》[FH11]）。Clojure 会敏锐地观察和协调线程的所有活动。如果没有任何冲突，例如，每个线程都在存取不同账户，则整个过程中就不会涉及任何锁，于是也就不会有延迟，并最终达到最大的并发度。当有两个线程试图访问相同数据时，事务管理器就会介入解决冲突，而我们的代码也就无需涉及任何显式加锁的操作。下面让我们一起研究一下这套系统是如何运作的。

在设计上，值是不可变的，而实体也仅在 Clojure 的事务中是可变的。在 Clojure 中，压根就没有改变状态的方法，也没有任何相关的编程工具可用。而如果出现任何试图在事务之外改变对象实体的动作时，系统就会抛出 illegalStateException 异常。换句话说，一旦与事务进行了绑定，在没有冲突时，所有变更都是即时生效的；而一旦发生冲突，Clojure 将会自动将事务回滚并重试。我们程序员的主要职责是保证事务中的代码都是幂等的，这是我们在函数式编程中避免副作用的常用手段，而这种手段在 Clojure 的编程模型中也同样适用。

是时候该看一个 Clojure STM 的例子了。我们可以用 ref 在 Clojure 中创建多个可变的实体，其中每个 ref 都提供了对于其表示的不可变状态的实体的可协调同步变更[⊖]。下面就让我们创建一个 ref 并尝试改变它。

usingTransactionalMemory/clojure/mutate.clj

```
(def balance (ref 0))

(println "Balance is" @balance)

(ref-set balance 100)

(println "Balance is now" @balance)
```

⊖ Clojure 还提供了一些我在本书中未能覆盖的其他并发模型，如 Agents，Atoms 和 Vars。更多信息请参阅 Clojure 的官方文档和相关书籍。

在上面的代码中，我们定义了一个名为 balance 的变量并用 ref 将其标记为可变的，此时 balance 就表示了一个带有不可变数值 0 的可变实体。然后我们将该变量的当前值打印了出来。接着我们会通过 ref-set 命令来尝试修改 balance 的值。如果该操作成功，我们就可以通过最后一条打印语句看到 balance 的新值。下面就让我们来看看这段代码的执行结果。我是用 clj mutable.clj 来运行这个脚本的，关于如何在你的系统上安装和运行 Clojure 脚本请参阅 Clojure 的文档。

```
Balance is 0
Exception in thread "main"
  java.lang.IllegalStateException: No transaction running (mutate.clj:0)
...
```

从结果来看，由于控制台正常打印出了初始 balance 的值 0，因此前两行代码应该是没问题的。由于我们在事务之外更改了变量的值，以至于惹恼了 Clojure 之神，所以在执行到第三行的时候系统抛出了 IlligalStateException 异常。当我们在没进行同步就更改共享可变变量的时候，与 Java 里那些人神共愤的行为相比，Clojure 抛出的这种清晰明确的失败简直是程序员的福音。这对于程序开发来说是一个相当大的改进，因为我们宁愿代码要么正确要么明确地抛出错误，而不是默默地产生一些不可预测的结果。问题进行到这里，原本我还想邀你庆祝一下我们获得了这么好的特性呢，不过看你的样子可能更希望先修复一下 Clojure STM 所抛出的那个错误，所以让我们继续往下看。

在 Clojure 中创建一个事务是很容易的，只需要用一个 dosync 调用把代码段包装起来就行了。在结构上，这种做法十分类似于 Java 中用 synchronized 修饰代码段的做法，但二者其实还是有不少区别的：

- 如果我们忘记了为需要同步的可变代码段加上 dosync 的话，系统会对此予以清晰的警告。
- dosync 并没有创建任何互斥锁，而是通过将代码段用一个事务包装起来的方式与其他线程进行公平竞争。
- 由于并未使用任何显式的锁，所以我们可以无须担心加锁顺序并享受这种不会死锁的并发所带来的好处。
- STM 提供了一套简单的运行时事务组合锁，所以我们无需在程序设计期间预先考虑诸如“谁用什么顺序锁住了什么”这样的问题。
- 不使用显式的锁就意味着程序员无须再使用保守的互斥代码块了（即 synchronized 代码块。——译者注）。于是程序整体的并发度可以得到最大程度的开发，并且其效果只受应用程序行为和数据访问方式的影响。

当一个事务运行起来之后，如果没有与之冲突的线程 / 事务，则该事务一定可以完成，并且其所做的更改可以被写到内存中。然而，一旦发现之前启动的其他事务可能会干扰到本事务运行的时候，Clojure 就会自动将之前所做的变更进行回滚并重做本事务。在对代码经过下面的修正之后，我们就可以成功地更改 balance 变量的值了。

usingTransactionalMemory/clojure/mutatesuccess.clj

```
(def balance (ref 0))

(println "Balance is" @balance)

(dosync
  (ref-set balance 100))

(println "Balance is now" @balance)
```

这段代码较之上一版本唯一的变化就是将 ref-set 用 dosync 包裹起来。当然，dosync 的作用域并不仅仅局限于单条语句，而是可以覆盖整个代码块或同一事务中的多个表达式。下面让我们运行这段代码并观察其结果。

```
Balance is 0
Balance is now 100
```

我们知道，balance 的状态值“0”是不可变的，而 balance 本身则是一个可变实体。在 dosync 作用域内所形成的这个事务中，我们先是创建了一个新值 100，然后修改 balance 令其指向这个值（我们要逐步习惯接受不可变值的概念）。但此时旧的值 0 依然还在，balance 也正指向该值，所以我们需要在新的值（100）创建完成之后，立即告知 Clojure 修改 balance 内部的指针，使其指向新值。如果后面不再有任何引用指向旧值“0”，则垃圾回收器会负责将其回收掉。

Clojure 提供了 3 种改变可变实体的方法，并且所有这三种方法都只能被封装在事务中才能使用：

- ref-set 命令可以设置实体的值并在操作完成后返回该值。
- alter 命令能够将实体在事务中的值设置成某特定函数的返回值，并在操作完成后返回该值。使用该命令是改变一个可变实体值的最佳方法。
- commute 命令与 alter 命令功能类似，二者的主要区别是 commute 会将提交点（commit point）与事务中所发生的变化分离开来。该命令的主要功能也是将实体在事务中的值设置成某特定函数的返回值，并在操作完成后返回该值。当代码运行至提交点的时候，只有最后一个对实体的变更操作才能最终生效，而中间值都将被忽略。

commute 是一个非常好用的命令，尤其是我们遇到类似“谁留到最后谁是赢家”这类应用的时候，其并发度要大大高于 alter 命令。但在除此之外的大多数情况下，alter 都要比 commute 更合适。

除了 ref 命令簇之外，Clojure 还提供了能够同步地更改数据的 atom 命令簇。与 ref 命令簇所不同的是，由 atom 命令簇所作出的变更是不接受调整的，并且同一事务下用 atom 所做的变更也不能和其他变更组合在一起。究其本质，是因为 atom 命令簇其实并不属于事务内的操作（我们可以将每个 atom 变更看做一个独立的事务）。为了清晰起见，离散的变更最好用 atom 命令簇，而对于多个需要组合或协调的变更则最好使用 ref 命令簇。

6.5　STM 中的事务

相信你之前一定在数据库中使用过事务，所以对原子性、一致性、隔离性和持久性（ACID）这些事务的基本属性应该非常熟悉了。Clojure 的 STM 对事务的支持与数据库有所不同，由于 STM 中的数据是全都放在内存而不是数据库或文件系统里的，所以 STM 只提供了事务的前三个属性，而缺少了对持久性的支持。

原子性：STM 事务是原子的。即我们在一个事务中所做的变更要么对所有其他外部事务可见，要么完全不可见。更具体一些，就是一个事务中所有 ref 的变更要么都生效，要么都不生效。

一致性：是指事务应该要么执行完成并令外界看到其造成的变化，要么执行失败并使所有相关数据都保持原状。如果有多个事务同时运行，那么从这些事务之外的角度来进行观察，我们可以看到它们所造成的变化始终是一个接着一个发生的，中间不会有任何交叉。例如，在（对同一个账户。——译者注）两个独立且并发的存款和取款事务完成之后，账户余额应该处于一种与两个动作所产生的累加效果相一致的状态（取钱是对账户加上一个负数。——译者注）。

隔离性：本事务无法看到其他事务的局部变更结果，即事务所造成的变更只能在其成功完成后才对外可见。

我们可以看到，这些属性都是侧重于数据的完整性和可见性的。其中，隔离性并不意味着事务之间就不能进行协调了。相反地，STM 会密切监控所有事务的进展情况并努力使所有事务都能跑完（除非遇到由应用程序产生的异常）。

Clojure 的 STM 采用了与数据库相似的多版本并发控制技术（MVCC），其并发控制也和数据库中的乐观锁（optimistic locking）很像。当我们启动一个事务的时候，STM 会记录一下时间戳，并将事务中将会用到所有 ref 都拷贝一份。由于状态是不可变的，所以对于 ref 的拷贝是多快好省的。当对某个不可变状态进行“变更”的时候，我们其实并没有改变它的值，而是为其创建了一个含有新值的拷贝。该拷贝是本事务的一个内部状态，并且由于我们使用了持久化的数据结构（见 3.6 节），这一步也是多快好省的。而如果 STM 识别出我们操作过的 ref 已经被别的事务改了的话，它就会中止并重做本事务。当事务成功完成时，所有的变更都会被写入内存，而时间戳也将被更新（见图 6-1）。

6.6　用 STM 实现并发

用事务来实现并发自然是极好的，但如果两个事务都试图更改同一实体的话会是什么状况呢？放心，我不会让你等很久的，本节我们将研究几个关于这方面问题的例子。

在进入实例研究之前我有句话要提醒你：由于事务可能被重复执行多次，所以在写产品代码的时候请务必确保事务是幂等的并且没有任何副作用。这意味着在事务中控制台不能有任何输出、不能记录日志、不能发邮件、也不能做任何不可逆操作。如果违背了上述任何一点，我们只能后果自负。一般而言，我们最好是把这些有副作用的动作收拢起来，

在事务完成之后再统一执行它们。

在示例代码中，我并没有完全遵循上述警告，所以你将会在代码中看到打印语句。但这只是为了演示的需要才加进去的，请千万不要在办公室这么写代码！

通过之前的例子，我们已经知道了如何在一个事务中更改 balance 的值。现在让我们写一个多事务竞争更改 balance 的程序：

usingTransactionalMemory/clojure/concurrentChangeToBalance.clj

```
(defn deposit [balance amount]
  (dosync
    (println "Ready to deposit..." amount)
    (let [current-balance @balance]
      (println "simulating delay in deposit...")
      (. Thread sleep 2000)
      (alter balance + amount)
      (println "done with deposit of" amount))))

(defn withdraw [balance amount]
  (dosync
    (println "Ready to withdraw..." amount)
    (let [current-balance @balance]
      (println "simulating delay in withdraw...")
      (. Thread sleep 2000)
      (alter balance - amount)
      (println "done with withdraw of" amount))))

(def balance1 (ref 100))

(println "Balance1 is" @balance1)
(future (deposit balance1 20))
(future (withdraw balance1 10))

(. Thread sleep 10000)

(println "Balance1 now is" @balance1)
```

本例中我们创建了两个事务，分别用于存款和取款。在 deposit() 函数中，我们先把 balance 复制了一份，然后插入一个 2 秒的延时以便模拟事务冲突的环境。在延时结束之后，我们将当前余额与待存入钱数（amount 的值）进行累加从而得到存入后的余额。withdraw() 函数的实现与 deposit() 十分类似，唯一区别就是需要用当前余额减去取款金额。这两个方法之后的代码都是用于测试执行结果用的：我们首先将变量 balance1 的初始值设为 100，然后用 future() 函数分别启动两个独立的线程来执行上述两个函数。下面让我们观察一下程序的输出结果：

```
Balance1 is 100
Ready to deposit... 20
simulating delay in deposit...
Ready to withdraw... 10
simulating delay in withdraw...
done with deposit of 20
Ready to withdraw... 10
```

```
simulating delay in withdraw...
done with withdraw of 10
Balance1 now is 110
```

deposit() 函数和 withdraw() 函数都持有一个 balance 的本地拷贝。当模拟延时结束之后，deposit() 事务也随即很快执行完毕，但 withdraw() 事务就没那么幸运了。因为 withdraw() 函数从延时中返回之后发现 balance 的值已经发生了变化，这意味着其本地拷贝已经失效，所以后面的修改再进行下去也没意义了，于是 Clojure 的 STM 迅速终止并重做了该事务。事务代码中的打印语句为我们理解 STM 的运作机制提供了很大的帮助，如果没有那些打印语句，我们就不会注意到这些活动。除了 STM 运作细节之外，本例中我们最值得注意的地方是在两个事务竞争执行的环境下，balance 保持了一致性，其值正确地反映了存取款动作对其造成的影响。

本例中，我们通过读取余额并延迟后续操作执行的方式故意将两个事务置于冲突环境下。但如果把 let 语句都去掉，我们就会发现两个事务都能在保持一致性的前提下无重复地完成。这一现象向我们充分展示了 STM 既能保持一致性又可以提供最大程度并发性的能力。

通过上面的例子我们已经知道如何更改一个简单变量，但如果我们有一堆变量时该怎么办呢？ Clojure 里面的 list 是不可变的，但我们可以通过一个能改变其实体指向的可变引用来模拟 List 变更的行为。在这种方式下，我们只是简单地改变了 list 的视图，list 本身则并没有改变。下面就让我们通过一个例子来看一下具体如何运作。我的家庭梦想表单里原本只有一个 iPad，现在我想往里面再添加一个新的 MacBook Pro（MBP）和一个我儿子想要的新自行车进去，于是我就通过两个线程分别把这两个心愿添加到表单里。下面就是相关的实现代码：

usingTransactionalMemory/clojure/concurrentListChange.clj

```
(defn add-item [wishlist item]
  (dosync (alter wishlist conj item)))

(def family-wishlist (ref '("iPad")))
(def original-wishlist @family-wishlist)

(println "Original wish list is" original-wishlist)

(future (addItem family-wishlist "MBP"))
(future (addItem family-wishlist "Bike"))

(. Thread sleep 1000)

(println "Original wish list is" original-wishlist)
(println "Updated wish list is" @family-wishlist)
```

add-Item() 函数的功能是将给定心愿项添加到心愿单里，其中 alter 方法的功能是用给定的函数（在本例中特指 conj() 函数）来更改事务内的 ref 变量。而 conj() 函数则可以返回一个新的集合，该集合是待加入项与原集合的并集。随后我们在两个不同的线程中调用了 add-Item() 函数。下面就是代码的执行结果，可以看到两个事务的影响：

```
Original wish list is (iPad)
Original wish list is (iPad)
Updated wish list is (Bike MBP iPad)
```

原始的心愿单是不可变的，所以从代码结尾的输出来看它仍然保持不变。当我们向心愿单添加新项目时，本来是应该将原始心愿单数据复制一份来用的。但是由于采用了持久化的数据结构，我们就可以通过共享心愿项的方式兼顾内存使用与性能，其实现方式如图 6-2 所示。

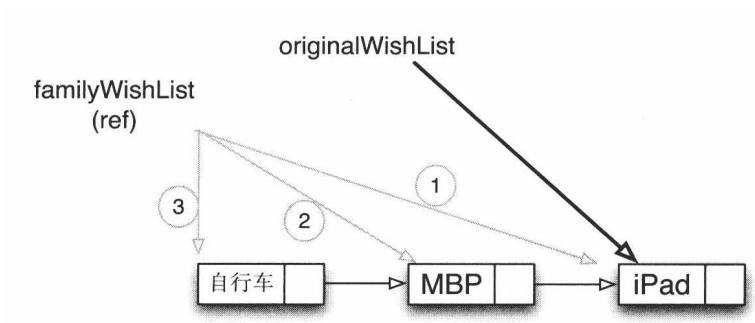


图 6-2　向不可变的心愿单中“添加”心愿项

心愿单的状态与 originalWishList 引用都是不可变的，而 familyWishList 则是一个可变的引用。所有添加心愿项的请求都是在各自独立的事物中运行的。第一个完成添加的事务更改了 familyWishList，使其指向新的心愿单（如图 6-2 中 (2) 所示）。与此同时，由于心愿单本身是不可变的，所以新的心愿单可以与原始的心愿单共享“iPad”这一项。当第二个添加心愿的事务完成时，新的心愿单同样可以与之前的心愿单共享前两个心愿项（如图 6-2 中 (3) 所示）。

处理写偏斜异常

前面我们已经学习了 STM 是如何处理事务间写冲突的，但有些时候冲突并非如此显而易见。假设我们在某银行有一个支票账户和一个储蓄账户，且银行规定两个账户的最小总余额不得低于 $1000，而此时两个账户的余额分别为 $500 和 $600。根据银行的规定可知，我们现在只能从其中一个账户中取 $100。如果我们按顺序分别从两个账户取 $100，那么第一个请求会成功而第二个请求则会失败。如果两个取款请求是并发执行的，那么由于所谓写偏斜异常的存在，两个事务都能够顺利完成，两个取款事务看到的总余额都超过了 $1000，同时二者更改的又是不同的值，所以根本不存在写冲突的问题。其造成的结果是，账户总余额最终为 $900，低于银行设定的最低限额。下面让我们用代码构建出这种异常，然后再研究如何解决这个问题。

usingTransactionalMemory/clojure/writeSkew.clj

```
(def checking-balance (ref 500))
(def savings-balance (ref 600))
```

```
(defn withdraw-account [from-balance constraining-balance amount]
  (dosync
    (let [total-balance (+ @from-balance @constraining-balance)]
      (. Thread sleep 1000)
      (if (>= (- total-balance amount) 1000)
        (alter from-balance - amount)
        (println "Sorry, can't withdraw due to constraint violation")))))
        (println "checking-balance is" @checking-balance)
        (println "savings-balance is" @savings-balance)
        (println "Total balance is" (+ @checking-balance @savings-balance))

        (future (withdraw-account checking-balance savings-balance 100))
        (future (withdraw-account savings-balance checking-balance 100))

        (. Thread sleep 2000)

        (println "checking-balance is" @checking-balance)
        (println "savings-balance is" @savings-balance)
        (println "Total balance is" (+ @checking-balance @savings-balance))
```

代码的前两行是对账户余额进行赋初始值。在 withdraw-account() 函数中，我们会读取两个账户的余额并将二者相加得到总余额（total-balance）。为了制造事务冲突的环境，我们在计算余额之后会人为地插入一个延时。随后，只要两个账户的总余额不低于银行的最小限额的话，我们就会更新 from-balance 所代表的那个账户的余额。在余下的代码里，我们并发地执行了两个取款的事务，其中第一个事务从支票账户中取了 $100 而第二个事务则从储蓄账户中取了 $100。正如我们在程序的输出结果中所看到的那样，由于两个事务是分别独立执行且二者没有任何交集，所以它们无法意识到彼此已经陷入到写偏斜并使最终结果违反了银行的规定。

```
checking-balance is 500
savings-balance is 600
Total balance is 1100
checking-balance is 400
savings-balance is 500
Total balance is 900
```

在 Clojure 中，我们可以通过 ensure() 函数很容易地避免写偏斜问题。通过该方法，我们可以告诉事务要睁大眼睛盯着某个本事务只读不改的变量。这样一来，STM 就可以确保只有在我们读取的这个值没有在本事务外被修改的情况下，本事务的写操作才能被提交，否则 STM 要重做本事务。

根据上述思路，让我们修改一下 withdraw-account() 函数：

```
Line 1  (defn withdraw-account [from-balance constraining-balance amount]
  2    (dosync
  3      (let [total-balance (+ @from-balance (ensure constraining-balance))]
  4        (. Thread sleep 1000)
  5        (if (>= (- total-balance amount) 1000)
  6          (alter from-balance - amount)
  7          (println "Sorry, can't withdraw due to constraint violation")))))
```

在代码第 3 行，我们调用 ensure() 函数来监控 constraining-balance 这个本事务只读不

改的变量的值。事实上，STM 在这行代码中对 constraining-balance 变量加了一个读锁，其目的是阻止其他事务获得该变量的写锁。在本事务临近结束的时候，STM 会在执行提交动作之前释放所有的读锁，这样就可以在并发度增加的时候避免死锁的发生。

正如我们在下面输出结果中所看到的那样，即使我们像刚才一样并发执行两个取款事务，但由于我们在 withdraw-account() 函数里调用了 ensure()，所以银行对两个账户的最小总余额限制仍然能够得以保持。

```
checking-balance is 500
savings-balance is 600
Total balance is 1100
checking-balance is 500
savings-balance is 500
Total balance is 1000
Sorry, can't withdraw due to constraint violation
```

STM 的显式锁无关执行模型是相当强大的。如果事务间没有冲突，那就不会有任何阻塞发生。而一旦存在冲突，则至少有一个事务可以无障碍地执行下去，而其他竞争者则需要重做。对于需要重做的事务，Clojure 设定了一个最大重做次数，并能够保证两个线程不会因为重做节奏相同而导致反复冲突重做。当我们的业务模型是读多、写冲突非常少的情况时，STM 的执行模型就更能体现其优势。例如，该模型非常适合于传统的 Web 应用程序，通常情况下，多个用户并发地更新他们各自的数据，用户之间的共享状态冲突非常少，而这些少量的写冲突则可以通过 STM 来轻松处理掉。

由于能够解决如此多令人头痛的并发问题，Clojure 的 STM 可以称得上是并发编程世界里的阿司匹林了。如果忘了创建事务，我们就会被系统严厉地斥责（指抛异常。——译者注）。而与之相反的是，只需简单地把 dosync 放到正确的位置上，系统就能够回馈给我们多线程环境下的高并发和一致性。如此低的使用门槛，简洁、明确以及可预测的行为都使得 Clojure STM 成为我们开发并发应用时一个非常值得认真考虑的选择。

6.7　用 Akka/Multiverse STM 实现并发

上面我们已经学习了如何在 Clojure 里使用 STM，我猜你现在一定很好奇如何在 Java 代码中使用 STM。而对于这一需求，我们有如下选择：

- 直接在 Java 中使用 Clojure STM。方法非常简单，我们只需将事务的代码封装在一个 Callable 接口的实现中就行了，详情请参见第 7 章。
- 喜欢用注解（annotation）的开发者可能会更倾向于使用 Multiverse 的 STM API。
- 除了 STM 之外，如果我们计划使用角色，那么还可以考虑选择 Akka 库。

Multiverse 是由 Peter Veentjer 主持开发的一个基于 Java 的 STM 实现。通过这个库，我们可以在 Java 代码中使用注解来标识事务边界。我们既可以用 @TransactionalMethod 注解将单个的方法标记为事务性的，也可以用 @TransactionalObject 注解将一个类的所有方法都标记为事务性的。为了与其他 JVM 上的语言进行集成，Multiverse 还提供了一组丰富的 API 来控制事物的开始和结束。

Akka 是一个由 Jonas Boner 主持开发的一个基于 Scala 的解决方案，该方案可以用于包括 Java 在内的很多其他运行于 JVM 上的语言。Akka 不但提供了 STM 和基于角色的并发方案，还提供了将二者混合使用的选项。此外，Akka 使用 Multiverse 作为其 STM 的实现并提供了 ACI（ACID 的子集）特性。

Akka 的性能非常棒，并且由于它既支持 STM 又支持基于角色的模型（详情请参见第 8 章），本章我们将会用它来实现演示 Java STM 的例子。

6.7.1　Akka/Multiverse 中的事务

Akka 的 Java 版采用了 Multiverse 的 Clojure 风格的 STM。与 Java 那繁冗的代码风格相比，Clojure 风格的 Akka 不会强迫我们在能够修改可变实体之前就创建事务。如果我们没有主动提供事务，则 Akka/Multiverse 就会自动把访问请求包装在一个事务中。所以当我们处于事务之中时，Akka 的 ref 与 Clojure 的 ref 的表现是相同的；而当我们位于事务之外时，Akka ref 的表现则更像是 Clojure 的 atom。换句话说，想要使变更同步且有序就必须使其在事务中完成，否则变更将是同步但无序的。在任何情况下，Akka 都会保证对于 ref 的更改是原子的、隔离的且一致的，并同时提供了不同等级的协调粒度。

在 Akka 中，我们既可以用写代码的方式在事务层对事务进行配置，也可以通过配置文件在应用程序 /JVM 层进行配置。例如，我们可以将一个事务定义为只读（readonly），于是 Akka 将不再允许任何位于该事务范围内的 Akka 引用被修改。这样做的好处是，如果我们将一些不可变的事务设置为只读，则程序性能将会得到一定的提升。除此之外，我们还可以控制在冲突情况下事务的最大重试次数。当然，还有很多其他参数可供我们配置，详情请参阅 Akka 的帮助文档。

Akka 扩展了 Multiverse 中的嵌套事务（请参见 6.9 节），所以我们能够很方便地在事务中调用启动其他事务的函数。默认情况下，这些内部事务或嵌套事务都是与其外部事务融为一体的。

6.7.2　使用 Akka 引用和事务

Clojure 中的 ref 是在语言层定义的，而 Akka 是一个公共类库所以不能依赖任何现有语言的支持。所以 Akka 在其 akka.stm 包中提供了一个托管事务引用（managed transactional reference）Ref<T> 和一些为原始类型而设的特殊类，如 IntRef、LongRef 等。Ref<T>（以及所有原始类型的特殊引用）代表指向类型 T 的一个不可变值的托管可变实体（managed mutable identity）。像 Integer、Long、Double、String 这些类型以及其他不可变类型都符合作为值对象的（value object）条件。如果我们用了自己定义的类，则必须保证这个类是不可变的。也就是说，这个自定义的类只能包含 final 字段。

我们可以创建一个 Ref<T> 的实例作为托管事务引用，可以为其指定一个初始值或干脆不指定（默认为 null）。如果想获得引用的当前值，可以使用 get() 函数。如果要使引用指向另一个可变实体，则可以使用 swap() 函数。这些调用可以在我们提供的事务里执行，但如

果我们没提供事务的话，它们也可以在其各自的事务中运行。

当多个线程都试图更改同一个托管引用时，Akka 可以保证只有一个变更可以写入内存而其他变更将全部重做。Akka 有专门的事务工具负责管理事务跨越内存栅栏的过程。也就是说，Akka（通过 Multiverse）保证了在事务中一个托管 ref 变更的提交会先于后续所有其他事务对该 ref 的读操作，即该变更对所有其他事务可见。

6.8　创建事务

我们创建事务的目的是为了协调针对多个托管引用的变更。事务将会保证这些变更是原子的，也就是说，所有的托管引用要么全部被提交要么全部被丢弃，所以在事务之外我们将不会看到有任何局部变更（partial change）出现。此外，我们也可以用创建事务的方式来解决对单个 ref 先读后写所引发的相关问题。

Akka 是用 Scala 开发出来的，所以如果我们工作中用的是 Scala 的话，就可以直接幸福地享用 Akka 简洁明了的 API 了。对于那些日常工作中不能使用 Scala 开发的程序员，Akka 同样也提供了一组方便的 API，以帮助他们通过 Java 语言来使用 Akka 的功能。我们还可以在 Java 中直接使用 Multiverse STM。本节我们将会看到如何利用 Akka 在 Java 和 Scala 中创建事务。

首先我们需要选一个适合用事务来解决的例子。我们在第 5 章中重构的 EnergySource 类使用了显式的加锁和解锁操作（其最终版本详见第 5.7 节），下面让就我们将这些显式的加锁 / 解锁操作换用 Akka 的事务 API 来实现。

6.8.1　在 Java 中创建事务

为了将代码逻辑包装到一个事务中，我们需要继承 Atomic 类并将代码放到该类的 atomically() 函数里。随后，我们可以通过调用 Atomic 实例的 execute() 函数来执行事务代码。类似于下面这样：

```
return new Atomic<Object>() {
  public Object atomically() {
    //code to run in a transaction...
    return resultObject;
  }
}.execute();
```

调用 execute() 函数的线程将负责执行 atomically() 函数里的代码。然而如果调用者本身并没有处在一个事务中的话，那么这个调用将会被包装在一个新的事务中。

下面让我们用 Akka 事务来重新实现 EnergySource。首先，让我们将不可变状态包装到可变的 Akka 托管引用中去。

usingTransactionalMemory/java/stm/EnergySource.java

```
public class EnergySource {
  private final long MAXLEVEL = 100;
  final Ref<Long> level = new Ref<Long>(MAXLEVEL);
```

```
final Ref<Long> usageCount = new Ref<Long>(0L);
final Ref<Boolean> keepRunning = new Ref<Boolean>(true);
private static final ScheduledExecutorService replenishTimer =
  Executors.newScheduledThreadPool(10);
```

在这段变量定义的代码中，level 和 usageCount 都被声明为 Akka Ref，并且各自持有一个不可变的 Long 类型的值。于是在 Java 中我们就不能更改这些 Long 类型的值了，但我们仍然可以通过更改托管引用（即实体）使其安全地指向新值。

在 EnergySource 的上一个版本中，ScheduledExecutorService 会周期性地（每秒钟一次）调用 replenish() 函数直至整个任务结束，这就要求 stopEnergySource() 必须是同步的。而在这个版本中，我们不用再周期性地调用 replenish() 函数，而只会在对象实例初始化的时候执行一下调度操作。在每次调用 replenish() 函数时，我们都会根据 keepRunning 的值来决定该函数是否应该在 1 秒之后再次被调度执行。这一变化消除了 stopEnergySource() 函数和调度器 / 计时器（timer）之间的耦合。相反地，stopEnergySource() 函数现在只依赖于 keepRunning 这个标志，而该标志可以很容易地通过 STM 事务来行管理。

在这一版的代码中，由于可以依赖事务来保证安全性，所以我们没必要再对 stopEnergySource() 函数进行同步了。同时，由于 swap() 函数本身就是以事务方式执行的，所以我们也无需显式地为其创建事务。

usingTransactionalMemory/java/stm/EnergySource.java

```
private EnergySource() {}

private void init() {
  replenishTimer.schedule(new Runnable() {
    public void run() {
      replenish();
      if (keepRunning.get()) replenishTimer.schedule(
        this, 1, TimeUnit.SECONDS);
    }
  }, 1, TimeUnit.SECONDS);
}

public static EnergySource create() {
  final EnergySource energySource = new EnergySource();
  energySource.init();
  return energySource;
}

public void stopEnergySource() { keepRunning.swap(false); }
```

如下所示，返回当前电量和使用次数的方法将会用到托管引用，但也只是需要调用一下 get() 函数而已。

usingTransactionalMemory/java/stm/EnergySource.java

```
public long getUnitsAvailable() { return level.get(); }

public long getUsageCount() { return usageCount.get(); }
```

在 getUnitsAvailable() 函数和 getUsageCount() 函数中，由于其中 get() 的调用都是以事

务方式运行的，所以无需显式地将它们封装在事务里。

由于我们会在 useEnergy() 函数中同时修改电量和使用次数，所以 useEnergy() 函数需要使用一个显式的事务来完成这些操作。在这里，我们需要保证对所有被读取的值的变更都能保持一致性，即确保对这两个字段的变更是原子的。为了实现这一目标，我们将使用 Atomic 接口，并用 atomically() 函数将我们的逻辑代码封装到一个事务中。

usingTransactionalMemory/java/stm/EnergySource.java

```
public boolean useEnergy(final long units) {
  return  new Atomic<Boolean>() {
    public Boolean atomically() {
      long currentLevel = level.get();
      if(units > 0 && currentLevel >= units) {
        level.swap(currentLevel - units);
        usageCount.swap(usageCount.get() + 1);
        return true;
      } else {
        return false;
      }
    }
  }.execute();
}
```

useEnergy() 函数的功能是从当前电量中减掉所消耗的电量（即 unit。——译者注）。为了实现这一目标，我们需要保证所涉及的 get 和 set 操作都在同一个事务中完成，所以我们把所有相关操作都用 atomically() 函数封装了起来。最后，我们会调用 execute() 函数来启动事务并顺序执行的所有操作。

除了上述方法之外，我们还需要关注一下负责给电源充电的 replenish() 函数。由于这个方法也需要使用事务，所以其实现代码同样需要用 Atomic 进行封装。

usingTransactionalMemory/java/stm/EnergySource.java

```
  private void replenish() {
    new Atomic() {
      public Object atomically() {
        long currentLevel = level.get();
        if (currentLevel < MAXLEVEL) level.swap(currentLevel + 1);
        return null;
      }
    }.execute();
  }
}
```

下面是针对 EnergySource 类的测试代码。其主要功能是，用多个线程并发地使用电池，每使用一次消耗一个单位电量。

usingTransactionalMemory/java/stm/UseEnergySource.java

```
public class UseEnergySource {
  private static final EnergySource energySource = EnergySource.create();

  public static void main(final String[] args)
```

```
    throws InterruptedException, ExecutionException {
    System.out.println("Energy level at start: " +
      energySource.getUnitsAvailable());

    List<Callable<Object>> tasks = new ArrayList<Callable<Object>>();
    for(int i = 0; i < 10; i++) {
      tasks.add(new Callable<Object>() {
        public Object call() {
          for(int j = 0; j < 7; j++) energySource.useEnergy(1);
          return null;
        }
      });
    }

    final ExecutorService service = Executors.newFixedThreadPool(10);
    service.invokeAll(tasks);

    System.out.println("Energy level at end: " +
      energySource.getUnitsAvailable());
    System.out.println("Usage: " + energySource.getUsageCount());

    energySource.stopEnergySource();
    service.shutdown();
  }
}
```

上述代码需要把 Akka 相关的 JAR 添加到 Java 的 classpath 中才能编译和执行代码，所以首先我们需要创建一个标识 JAR 位置的环境变量：

```
export AKKA_JARS="$AKKA_HOME/lib/scala-library.jar:\
$AKKA_HOME/lib/akka/akka-stm-1.1.3.jar:\
$AKKA_HOME/lib/akka/akka-actor-1.1.3.jar:\
$AKKA_HOME/lib/akka/multiverse-alpha-0.6.2.jar:\
$AKKA_HOME/config:\
."
```

Classpath 的定义取决于你使用的操作系统以及 Akka 在你的操作系统中被安装的位置。我们可以用 javac 编译器来编译代码，并用 java 命令来负责执行，具体细节如下所示：

```
javac -classpath $AKKA_JARS -d . EnergySource.java UseEnergySource.java
java -classpath $AKKA_JARS com.agiledeveloper.pcj.UseEnergySource
```

万事俱备，下面让我们来编译并执行这段代码。通过代码的实现逻辑我们知道，电源初始有 100 个单位电量，而我们创建的 10 个线程将会消耗掉其中的 70 个单位电量，所以最后电源应该净剩 30 个单位电量。但由于电池电量会每秒恢复一个单位，所以每次运行结果可能会稍有不同，比如最后净剩电量可能是 31 个单位而不是 30 个单位。

```
Energy level at start: 100
Energy level at end: 30
Usage: 70
```

默认情况下，Akka 会将额外的日志消息打印到标准输出上。停掉这个默认的输出也很容易，我们只需要在 $AKKA_HOME/config 目录下创建一个名为 logback.xml 的文件，并在里面添加 <configuration /> 这项配置即可。由于这个文件位于 classpath 中，所以 logger

会自动找到这个文件、读取其中的配置并停掉消息输出。除此之外，我们还可以在这个配置文件中设置很多其他有用的配置项。详情请见 http://logback.qos.ch/manual/configuration.html。

正如我们在本例中所看到的那样，Akka 是在后台默默地对事务进行管理的，所以请你多花些时间研究一下上述示例代码，并对事务和线程的运作过程多做一些尝试以便加深对这块知识的理解。

6.8.2　在 Scala 中创建事务

我们之前已经看到了如何在 Java 中创建事务（并且我假设你已经阅读过那一部分，所以这里我们就不再赘述了），下面我们将会在 Scala 中用更少的代码来完成同样的功能。我们之所以能兼顾简洁与功能，部分得益于 Scala 自身简洁的特点，但更多还是由于 Akka API 使用了闭包 / 函数值（closures/function values）的缘故。

相比 Java 的繁冗，我们在 Scala 中可以通过很简洁的方法来创建事务。我们所需要做的只是调用一下 Stm 的 auomic() 函数就行了，如下所示：

```
atomic {
  //code to run in a transaction....
  /* return */ resultObject
}
```

其中，我们传给 atomic() 的闭包 / 函数值仅在当前线程所运行的那个事务内可见。

下面就是使用了 Akka 事务的 Scala 版本的 EnergySource 实现代码：

usingTransactionalMemory/scala/stm/EnergySource.scala

```
class EnergySource private() {
  private val MAXLEVEL = 100L
  val level = Ref(MAXLEVEL)
  val usageCount = Ref(0L)
  val keepRunning = Ref(true)

  private def init() = {
    EnergySource.replenishTimer.schedule(new Runnable() {
      def run() = {
        replenish
        if (keepRunning.get) EnergySource.replenishTimer.schedule(
          this, 1, TimeUnit.SECONDS)
      }
    }, 1, TimeUnit.SECONDS)
  }

  def stopEnergySource() = keepRunning.swap(false)
  def getUnitsAvailable() = level.get

  def getUsageCount() = usageCount.get

  def useEnergy(units : Long) = {
    atomic {
      val currentLevel = level.get
```

```
      if(units > 0 && currentLevel >= units) {
        level.swap(currentLevel - units)
        usageCount.swap(usageCount.get + 1)
        true
      } else false
    }
  }

  private def replenish() =
    atomic { if(level.get < MAXLEVEL) level.swap(level.get + 1) }
}

object EnergySource {
  val replenishTimer = Executors.newScheduledThreadPool(10)

  def create() = {
    val energySource = new EnergySource
    energySource.init
    energySource
  }
}
```

作为一个完全的面向对象语言，Scala 认为静态方法是不适合放在类的定义中的，所以工厂方法 create() 就被移到其伴生对象里面去了。余下的代码和 Java 版本非常相近，只是较之更为简洁。同时，由于使用了优雅的 atomic() 函数，我们就可以抛开 Atomic 类和 execute() 函数调用了。

Scala 版本的 EnergySource 的测试用例如下所示。在并发和线程控制的实现方面，我们既可以像 Java 版本那样采用 JDK 的 ExecutorService 来管理线程，也可以使用 Scala 的角色[⊖]来为每个并发任务分配执行线程。这里我们将采用第二种方式。当任务完成之后，每个任务都会给调用者返回一个响应，而调用者使用该响应来进行阻塞，等待所有任务结束之后才能继续执行。

usingTransactionalMemory/scala/stm/UseEnergySource.scala

```
object UseEnergySource {
  val energySource = EnergySource.create()

  def main(args : Array[String]) {
    println("Energy level at start: " + energySource.getUnitsAvailable())

    val caller = self
    for(i <- 1 to 10) actor {
      for(j <- 1 to 7) energySource.useEnergy(1)
      caller ! true
    }

    for(i <- 1 to 10) { receiveWithin(1000) { case message => } }
```

⊖ 这里提到 Scala 的角色仅仅是为了说明有这种方法可供使用。后面我们还将会学习如何使用功能更为强大的 Akka 角色。

```
    println("Energy level at end: " + energySource.getUnitsAvailable())
    println("Usage: " + energySource.getUsageCount())

    energySource.stopEnergySource()
  }
}
```

我们可以采用如下命令来引入 Akka 相关的 JAR 并编译运行上述代码，其中环境变量 AKKA_JARS 与我们在 Java 示例中的定义相同：

```
scalac -classpath $AKKA_JARS *.scala
java -classpath $AKKA_JARS com.agiledeveloper.pcj.UseEnergySource
```

Scala 版本代码的输出结果与我们在 Java 版本中所看到的没什么两样，并同样依赖于电量恢复的节奏，即可能最终剩余电量是 31 而不是 30。

```
Energy level at start: 100
Energy level at end: 30
Usage: 70
```

6.9　创建嵌套事务

在之前的示例中，每个用到事务的方法都是各自在其内部单独创建事务，并且事务所涉及的变动也都是各自独立提交的。但如果我们想要将多个方法里的事务协调成一个统一的原子操作的时候，上述做法就无能为力了，所以我们需要使用嵌套事务来实现这一目标。

通过使用嵌套事务，所有被主控函数调用的那些函数所创建的事务都会默认被整合到主控函数的事务中。除此之外，Akka/Multiverse 还提供了很多其他配置选项，如新隔离事务（new isolated transactions）等。总之，使用了嵌套事务之后，只有位于最外层的主控函数事务提交时，其内部所做的变更才会被提交。在具体使用时，为了保证所有嵌套事务能够作为一个整体成功完成，我们需要保证所有函数都必须在一个可配置的超时范围内做完。

我们在第 4.6 节中通过加锁方式实现的 AccountService 的 transfer() 函数将会受益于嵌套事务。因为上一个版本的 transfer() 函数需要按自然顺序对所有账户排序并显式地对锁进行管理。STM 将为我们消除所有这些负担。下面我们会首先在 Java 中用嵌套事务重新实现这一示例，然后再来看一下该示例在 Scala 中是如何实现的。

6.9.1　在 Java 中使用嵌套事务

现在让我们开始对 Account 类进行事务化的改造吧。首先我们需要把保存账户余额的变量 balance 改成托管引用，下面我们就来定义这个字段以及该字段的 getter 函数。

usingTransactionalMemory/java/nested/Account.java

```
public class Account {
  final private Ref<Integer> balance = new Ref<Integer>();

  public Account(int initialBalance) { balance.swap(initialBalance); }

  public int getBalance() { return balance.get(); }
```

在构造函数中，我们用 ref 的 swap() 函数将 balance 的初始值设置为给定余额。由于 swap() 函数运行在自己独立的事务中，所以我们就无需再创建额外的事务了（同时我们假设调用者也不会为这个操作创建额外的事务）。getBalance() 函数的情况与之类似，也在其自己的事务中访问 balance。

由于 deposit() 函数需要对 balance 进行先读后写的操作，所以该函数内的所有操作需要整体封装到一个事务里运行。下面的代码为我们展示了如何将这两个操作封装到一个独立事务中的方法。

usingTransactionalMemory/java/nested/Account.java

```
public void deposit(final int amount) {
  new Atomic<Boolean>() {
    public Boolean atomically() {
      System.out.println("Deposit " + amount);
      if (amount > 0) {
        balance.swap(balance.get() + amount);
        return true;
      }

      throw new AccountOperationFailedException();
    }
  }.execute();
}
```

基于同样的理由，我们需要把 withdraw() 函数里的所有操作也封装到一个独立的事务中。

usingTransactionalMemory/java/nested/Account.java

```
public void withdraw(final int amount) {
  new Atomic<Boolean>() {
    public Boolean atomically() {
      int currentBalance = balance.get();
        if (amount > 0 && currentBalance >= amount) {
          balance.swap(currentBalance - amount);
          return true;
        }

        throw new AccountOperationFailedException();
      }
    }.execute();
  }
}
```

如果运行过程中有异常抛出，则事务将会强制失败。所以当账户内余额不足或存款 / 取款操作输入了非法参数时，我们就可以利用这一点来表示操作失败。相当简单，是吧？从此我们就可以不用再担心同步、加锁、死锁等令人烦恼的问题了。

现在到了该浏览一下执行转账操作的 AccountService 类的时候了，让我们首先来看一下其中的 transfer() 函数：

usingTransactionalMemory/java/nested/AccountService.java

```
public class AccountService {
  public void transfer(
    final Account from, final Account to, final int amount) {
    new Atomic<Boolean>() {
       public Boolean atomically() {
         System.out.println("Attempting transfer...");
         to.deposit(amount);
         System.out.println("Simulating a delay in transfer...");
         try { Thread.sleep(5000); } catch(Exception ex) {}
         System.out.println("Uncommitted balance after deposit $" +
           to.getBalance());
         from.withdraw(amount);
         return true;
       }
    }.execute();
  }
```

在这个示例中，我们会将多个事务置于相互冲突的环境中，以此来演示嵌套事务的行为并帮助你加深对嵌套事务的理解。Transfer() 函数中的所有操作都是在同一个事务中完成的。作为转账过程的一部分，我们首先将钱存到目标账户中。紧接着，在经过一个为引入事务冲突而专门设置的延时之后，我们将钱从源账户中划走。我们希望当且仅当从源账户划款成功之后，向目标账户存款的操作才能够成功，这也是我们这个事务所要完成的目标。

我们可以通过打印 balance 的值来观察转账操作是否成功。如果有一个方便的函数来调用 transfer() 函数，处理异常，并在最后打印 balance 的值就更好了，下面就让我们动手写一个：

usingTransactionalMemory/java/nested/AccountService.java

```
public static void transferAndPrintBalance(
 final Account from, final Account to, final int amount) {
  boolean result = true;
  try {
    new AccountService().transfer(from, to, amount);
  } catch(AccountOperationFailedException ex) {
    result = false;
  }
  System.out.println("Result of transfer is " + (result ? "Pass" : "Fail"));
  System.out.println("From account has $" + from.getBalance());
  System.out.println("To account has $" + to.getBalance());
}
```

最后我们还需要一个 main() 函数来让整个示例运转起来。

usingTransactionalMemory/java/nested/AccountService.java

```
  public static void main(final String[] args) throws Exception {
    final Account account1 = new Account(2000);
    final Account account2 = new Account(100);

    final ExecutorService service = Executors.newSingleThreadExecutor();
```

```
    service.submit(new Runnable() {
      public void run() {
        try { Thread.sleep(1000); } catch(Exception ex) {}
        account2.deposit(20);
      }
    });
    service.shutdown();

    transferAndPrintBalance(account1, account2, 500);

    System.out.println("Making large transfer...");
    transferAndPrintBalance(account1, account2, 5000);
  }
}
```

在 main 函数中，我们创建了两个账户，并在一个单独的线程中从第二个账户里存款 $20。与此同时，我们还启动了一个在账户之间转账的事务。由于这些操作都会影响到公共实例（即两个账户。——译者注），所以这种做法将导致两个事务（存 $20 的事务和转账 $500 的事务。——译者注）产生冲突。于是只有一个事务能够顺利完成，而另一个将会重做。最后，我们会启动一个超出源账户余额的转账操作，以此来演示存款和取款这两个相互关联的事务通过嵌套事务的方式在转账过程中实现了原子性的操作。因为取款失败，存款也应当取消。下面让我们通过输出结果来观察事务的行为：

```
Attempting transfer...
Deposit 500
Attempting transfer...
Deposit 500
Simulating a delay in transfer...
Deposit 20
Uncommitted balance after deposit $600
Attempting transfer...
Deposit 500
Simulating a delay in transfer...
Uncommitted balance after deposit $620
Result of transfer is Pass
From account has $1500
To account has $620
Making large transfer...
Attempting transfer...
Deposit 5000
Simulating a delay in transfer...
Uncommitted balance after deposit $5620
Result of transfer is Fail
From account has $1500
To account has $620
```

输出结果起始处的转账事务重试操作让人看起来有些摸不着头脑。这个非预期的重试是由 Multiverse 对于单个对象上的只读事务的默认优化造成的。虽然有两种方法可以重新配置这一行为，但修改了之后可能会对性能造成影响。请参阅 Akka/Multiverse 文档来进一步了解变更这一配置所造成的影响。

在本例中，向账户 2 存 $20 的操作会先完成。而与此同时，从账户 1 向账户 2 的转账

事务则处于模拟的延迟当中。当转账事务重新恢复运行并察觉到其涉及的对象发生了变化时，该事务将悄悄地回滚并重做。如果事务在运行过程中一直出现内部数据有变化的情况，则该事务会不断重做直至成功或超时退出为止。本例中的转账事务是最终成功了的，账户余额的变化充分地反映了这一结果，账户 1 转出了 $500，而账户 2 则从并发的存款和转账操作中总共获取了 $520。

本例的最后一个操作是从账户 1 向账户 2 转 $5000。在这个事务中，存款操作顺利完成了，但事务能否最终成功还是要看取款操作的结果。不出所料，取款动作由于账户余额不足而失败并抛出了异常。随后，之前的存款动作被回滚，系统最终保证了账户余额数据不受事务失败的影响。

再次声明，在事务中打印信息和插入延时都不是好习惯，我在本例中这样用是为了使你能够更好地观察事务的运行序列和重做行为，在实际工作中请最好不要在事务代码里打印消息或记录日志。请记住，事务是不应该有任何副作用的。如果事务中确实需要包含有副作用的操作，我们可以将这些代码放到后面将会提到的后置提交（post-commit）处理函数里面去。

我可以拍胸脯向你保证，使用事务绝对可以替你分担大部分并发编程方面的烦恼。下面就让我们通过一组对比来看看事务到底效用几何。让我们回顾一下第 4.6 节中我们用加锁方式实现的转账函数 transfer()，为方便起见我将代码列在下面：

scalabilityAndTreadSafety/locking/AccountService.java

```
public boolean transfer(
  final Account from, final Account to, final int amount)
  throws LockException, InterruptedException {
  final Account[] accounts = new Account[] {from, to};
  Arrays.sort(accounts);
  if(accounts[0].monitor.tryLock(1, TimeUnit.SECONDS)) {
    try {
      if (accounts[1].monitor.tryLock(1, TimeUnit.SECONDS)) {
        try {
          if(from.withdraw(amount)) {
            to.deposit(amount);
            return true;
          } else {
            return false;
          }
        } finally {
          accounts[1].monitor.unlock();
        }
      }
    } finally {
      accounts[0].monitor.unlock();
    }
  }
  throw new LockException("Unable to acquire locks on the accounts");
}
```

你可以将上述代码与上一版本相比较，但首先要去掉延时和打印语句：

```
public void transfer(
  final Account from, final Account to, final int amount) {
  new Atomic<Boolean>() {
     public Boolean atomically() {
       to.deposit(amount);
       from.withdraw(amount);
       return true;
     }
  }.execute();
}
```

旧版本的代码既要考虑加锁的问题又要顾及加锁的顺序，所以很容易出错。代码越多越容易出问题，这是显而易见的道理。在新版本中，我们显著地降低了代码量和复杂度。这让我想起了 C.A.R.Hoare 的名言："这世界上有两种构建软件设计的方法。一种方法是使其足够简单以至于不存在明显的缺陷。而另一种方法是使其足够复杂以至于无法看出有什么毛病"。只有让代码更少、结构更简单，我们才能将更多的时间投入到应用程序逻辑的设计开发中去。

6.9.2　在 Scala 中使用嵌套事务

从上例中我们可以看到，使用了嵌套事务的 Java 版转账函数是非常简洁的。然而，虽然事务的使用让我们得以去除 Java 中那些用于同步的冗余代码，但还是会有一些由于 Java 语法需要而存在的一些额外代码。正如我们下面所看到的那样，Scala 的优雅和强大的表达能力使其在代码清晰简洁方面更胜一筹。下面就是 Scala 版的 Account 类：

usingTransactionalMemory/scala/nested/Account.scala

```
class Account(val initialBalance : Int) {
  val balance = Ref(initialBalance)

  def getBalance() = balance.get()

  def deposit(amount : Int) = {
    atomic {
      println("Deposit " + amount)
      if(amount > 0)
        balance.swap(balance.get() + amount)
      else
        throw new AccountOperationFailedException()
    }
  }

  def withdraw(amount : Int) = {
    atomic {
      val currentBalance = balance.get()
      if(amount > 0 && currentBalance >= amount)
        balance.swap(currentBalance - amount)
      else
        throw new AccountOperationFailedException()
    }
  }
}
```

Scala 版本的 Account 是直接从 Java 版本翻译过来的、但代码风格又带有 Scala 和 Akka 简洁优雅特征的一种实现。在 Scala 版本的 AccountService 中我们也可以看到同样的优点：

usingTransactionalMemory/scala/nested/AccountService.scala

```
object AccountService {
  def transfer(from : Account, to : Account, amount : Int) = {
    atomic {
      println("Attempting transfer...")
      to.deposit(amount)
      println("Simulating a delay in transfer...")
      Thread.sleep(5000)
      println("Uncommitted balance after deposit $" + to.getBalance())
      from.withdraw(amount)
    }
  }

  def transferAndPrintBalance(
    from : Account, to : Account, amount : Int) = {
    var result = "Pass"
    try {
      AccountService.transfer(from, to, amount)
    } catch {
      case ex => result = "Fail"
    }

    println("Result of transfer is " + result)
    println("From account has $" + from.getBalance())
    println("To account has $" + to.getBalance())
  }

  def main(args : Array[String]) = {
    val account1 = new Account(2000)
    val account2 = new Account(100)

    actor {
      Thread.sleep(1000)
      account2.deposit(20)
    }

    transferAndPrintBalance(account1, account2, 500)

    println("Making large transfer...")
    transferAndPrintBalance(account1, account2, 5000)
  }
}
```

与 Java 版本一样，Scala 版本的 AccountService 同样会将事务置于相互冲突的环境之下。所以毫无悬念，其输出结果也与 Java 版本完全相同：

```
Attempting transfer...
Deposit 500
Attempting transfer...
Deposit 500
Simulating a delay in transfer...
```

```
Deposit 20
Uncommitted balance after deposit $600
Attempting transfer...
Deposit 500
Simulating a delay in transfer...
Uncommitted balance after deposit $620
Result of transfer is Pass
From account has $1500
To account has $620
Making large transfer...
Attempting transfer...
Deposit 5000
Simulating a delay in transfer...
Uncommitted balance after deposit $5620
Result of transfer is Fail
From account has $1500
To account has $620
```

前面我们已经比较过用 Java 实现的加锁同步版本和嵌套事务版本（如下所示）的转账方法：

```
public void transfer(
  final Account from, final Account to, final int amount) {
  new Atomic<Boolean>() {
     public Boolean atomically() {
       to.deposit(amount);
       from.withdraw(amount);
       return true;
     }
  }.execute();
}
```

现在让我们将之与 Scala 版本进行一下比较：

```
def transfer(from : Account, to : Account, amount : Int) = {
  atomic {
    to.deposit(amount)
    from.withdraw(amount)
  }
}
```

从上面的对比中我们可以清晰地看到，Scala 版本的代码除了核心逻辑之外没有任何冗余。这又让我想起了 Alan Perlis 的名言：“如果用某种编程语言写代码时还需要注意一些与核心逻辑无关的东西，那么这个语言就是低级语言。”

截至目前，我们已经学习了如何用 Akka 创建事务以及如何组合嵌套事务，但我们才刚上路呢。下面我们将一起了解一下在 Akka 中如何对事务进行配置。

6.10　配置 Akka 事务

默认情况下，Akka 为其相关的运行参数都设定了默认值，我们可以通过代码或配置文件 akka.conf 来更改这些默认设置。如果想了解如何指定或修改该配置文件位置的详细信息，请参阅 Akka 的文档。

针对单个事务，我们可以利用 TransactionFactory 在程序代码中更改其设置。下面就让我们用这种方式先后在 Java 和 Scala 中更改一些设置来为你展示如何实现设置的变更。

6.10.1　在 Java 中对事务进行配置

通过之前的例子我们知道，在 Java 中我们是通过继承 Atomic 来实现事务的。而在构造 Atomic 实例对象的时候，我们可以为其提供一个 TransactionFactory 类型的可选构造函参来改变事务的属性。例如，我们可以通过将事务设置为 readonly 来提高性能并防止事务改变其内部引用的值。下面我们将创建一个用于盛装很多杯咖啡对象的 CoffeePot 类，并尝试在一个 readonly 的事务中对其进行操作。

usingTransactionalMemory/java/configure/CoffeePot.java

```
public class CoffeePot {
  private static final Ref<Integer> cups = new Ref<Integer>(24);

  public static int readWriteCups(final boolean write) {
    final TransactionFactory factory =
      new TransactionFactoryBuilder().setReadonly(true).build();

    return new Atomic<Integer>(factory) {
      public Integer atomically() {
        if(write) cups.swap(20);
        return cups.get();
      }
    }.execute();
  }
```

为了能够用编程的方式对事务进行配置，我们需要一个 TransactionFactory 实例，而 TransactionFactoryBuilder 则为我们提供了很多方便的函数用于创建该工厂实例。在上例中，我们创建了一个 TranactionFactoryBuilder 实例对象，并调用该对象的 setReadonly() 函数来为 TransactionFactory 添加 readonly 选项。由于 TransactionFactoryBuilder 实现了 Cascade[⊖]设计模式，所以我们可以将更多用于改变事务属性的函数串在一起挂在 TransactionFactoryBuilder 构造函数之后、build() 函数之前。随后我们把工厂实例作为 Atomic 的一个构造函数参数传给它，这样就保证了该事务内的所有动作都不会变更任何托管引用。

通过上述设置我们已经将 readWriteCups() 变成了一个只读事务，接下来你肯定希望了解在一个只读事务中试图改变引用的值将会产生什么后果。下面我们会调用两次 readWriteCups()，第一次仅仅是读取 cups 引用的内容，而第二次调用则会尝试改变 cups 引用的值。

⊖ 近些年来，特别是随着 JVM 上新语言的不断涌现，由 Kent Beck 所著的《Smalltalk Best Practice Patterns》[Bec96] 一书中所讨论的一些设计模式又被重新发掘了出来。

usingTransactionalMemory/java/configure/CoffeePot.java

```
    public static void main(final String[] args) {
      System.out.println("Read only");
      readWriteCups(false);

      System.out.println("Attempt to write");
      try {
        readWriteCups(true);
      } catch(Exception ex) {
        System.out.println("Failed " + ex);
      }
    }
}
```

由于被设置成了只读，所以 readWriteCups() 事务不欢迎变更请求。于是当我们试图更改 cups 引用的值时，系统抛出了 org.multiverse.api.exceptions.ReadonlyException 异常，并且整个事务也将回滚。

```
Read only
Attempt to write
Failed org.multiverse.api.exceptions.ReadonlyException:
Can't open for write transactional object 'akka.stm.Ref@1272670619'
because transaction 'DefaultTransaction' is readonly'
```

上述运行时异常是在调用 swap() 的时候抛出来的。该函数的作用是当且仅当新值与当前值不同时，将其 ref 改为指向新值的地址，否则，该函数将忽略变更请求。所以在本例中，如果我们在调用 swap() 时将参数 20 换成与当前 cpus ref 的值相等的 24，则系统就不会抛出任何异常。

6.10.2　在 Scala 中对事务进行配置

在 Scala 中，我们可以使用 atomic() 函数代替 Atomic 类来创建事务，该函数在使用时需要一个 TransactionFactory 类型的可选参数。同时，由于我们能够在伙伴对象（companion object）上使用工厂方法，所以创建 factory 实例也比在 Java 中要简单许多。

usingTransactionalMemory/scala/configure/CoffeePot.scala

```
object CoffeePot {
  val cups = Ref(24)

  def readWriteCups(write : Boolean) = {
    val factory = TransactionFactory(readonly = true)

    atomic(factory) {
        if(write) cups.swap(20)
        cups.get()
    }
  }

  def main(args : Array[String]) : Unit = {
    println("Read only")
    readWriteCups(false)
```

```
      println("Attempt to write")
      try {
        readWriteCups(true)
      } catch {
        case ex => println("Failed " + ex)
      }
    }
  }
```

除了在代码方面保持了 Scala 和 Akka 特有的简洁优雅之外，上述代码与 Java 版本就没有什么其他不同之处了，所以代码的执行结果也毫无意外地和 Java 版本完全相同。

```
Read only
Attempt to write
Failed org.multiverse.api.exceptions.ReadonlyException:
Can't open for write transactional object 'akka.stm.Ref@1761506447'
because transaction 'DefaultTransaction' is readonly'
```

6.11　阻塞事务——有意识地等待

我们经常会遇到这样一种情况，即某事务能否成功完成依赖于某个变量是否发生了变化，并且由于这种原因所引起的事务运行失败也可能只是暂时性的。作为对这种暂时性失败的响应，我们可能会返回一个错误码并告诉事务等待一段时间之后再重试。然而在事务等待期间，即使其他任务已经更改了事务 T 所依赖的数据，事务 T 也没法立即感知到并重试了。为了解决这一问题，Akka 为我们提供了一个简单的工具 retry()，该函数可以先将事务进行回滚，并将事务置为阻塞状态直到该事务所依赖的引用对象发生变化或事务阻塞的时间超过了之前配置的阻塞超时为止。我本人更愿意将这一过程称为“有意识地等待”，因为这种说法听起来比“阻塞”更合适一些。下面让我们将阻塞（或有意识地等待）用于下面的两个例子当中。

6.11.1　在 Java 中阻塞事务

程序员一般都会对咖啡因上瘾，所以加班的时候任何主动要去拿些咖啡回来喝的人都知道不能空手而归。但是这个拿咖啡的人很聪明，他没有忙等（busy wait）至咖啡罐被重新填满，而是在 Akka 的帮助下给自己设置了一个消息提醒，一旦咖啡罐有变化他就能收到这个通知。下面让我们用 retry() 来实现这个可以有意识等待的 fillCup() 函数。

usingTransactionalMemory/java/blocking/CoffeePot.java

```
public class CoffeePot {
  private static final long start = System.nanoTime();
  private static final Ref<Integer> cups = new Ref<Integer>(24);

  private static void fillCup(final int numberOfCups) {
    final TransactionFactory factory =
      new TransactionFactoryBuilder()
      .setBlockingAllowed(true)
      .setTimeout(new DurationInt(6).seconds())
```

```
    .build();

  new Atomic<Object>(factory) {
    public Object atomically() {
      if(cups.get() < numberOfCups) {
        System.out.println("retry........ at " +
          (System.nanoTime() - start)/1.0e9);
        retry();
      }
      cups.swap(cups.get() - numberOfCups);
      System.out.println("filled up...." + numberOfCups);
      System.out.println("........ at " +
        (System.nanoTime() - start)/1.0e9);
      return null;
    }
  }.execute();
}
```

在 fillCup() 函数中，我们将事务配置成 blockingAllowed 为 true 的状态，并将事务完成的超时时间设为 6 秒。当发现当前没有足够数量的咖啡时，fillCups() 函数没有简单地返回一个错误码，而是调用了 StmUtil 的 retry() 函数进行有意识地等待。这将使得当前事务进入阻塞状态，直到与之相关的 cups 引用发生变化为止。一旦有任何相关的引用发生改变，系统将启动一个新事务将之前包含 retry 的原子性代码进行重做。

下面让我们通过调用 fillCup() 函数来观察 retry() 的实际效果：

usingTransactionalMemory/java/blocking/CoffeePot.java

```
public static void main(final String[] args) {
  final Timer timer = new Timer(true);
  timer.schedule(new TimerTask() {
    public void run() {
      System.out.println("Refilling.... at " +
        (System.nanoTime() - start)/1.0e9);
      cups.swap(24);
    }
  }, 5000);

  fillCup(20);
  fillCup(10);
  try {
    fillCup(22);
  } catch(Exception ex) {
    System.out.println("Failed: " + ex.getMessage());
  }
 }
}
```

在 main() 函数中，我们启动了一个每隔大约 5 秒就往咖啡壶重新装填咖啡的定时任务。随后，第一个跑去拿咖啡的同事 A 立即取走了 20 杯咖啡。紧接着，当我们这边自告奋勇去取咖啡的同事 B 想再取走 10 杯咖啡时，他的动作将被阻塞直至重新装填任务完成为止，而这种等待要比不断重试的方案高效得多。重新装填的事务完成之后，同事 B 的请求将被自动重试，而这一次他的请求成功完成了。如果重新装填任务没有在我们配置的超时时间内

发生，则请求咖啡的事务将会失败，在上例的 try 代码块中的那个请求就属于这种情况。我们可以通过输出日志来观察到这一行为，同时也可以更深入地体会到 retry() 为我们带来的便利：

```
filled up....20
........ at 0.423589
retry........ at 0.425385
retry........ at 0.427569
Refilling.... at 5.130381
filled up....10
........ at 5.131149
retry........ at 5.131357
retry........ at 5.131521
Failed: Transaction DefaultTransaction has timed with a
total timeout of 6000000000 ns
```

从上述输出结果中我们可以看到，第一个倒 20 杯咖啡的请求是在程序开始运行之后 0.4 秒左右完成的。而第一个请求完成之后，咖啡壶里就仅剩余 4 杯咖啡了，所以第二个倒 10 杯咖啡的请求就只能被阻塞，直到程序运行至 5 秒左右时重新装填的任务完成为止。在重新装填的任务完成之后，倒 10 杯咖啡的事务被重新启动，并在程序运行到 5 秒多一点的时候成功完成。最后一个倒 22 杯咖啡的任务则由于在规定的超时时间内没有再次发生重新装填而以失败告终。

其实我们在日常工作中并不会经常用到 retry()，只有当应用程序逻辑需要执行某些操作，而这些操作又依赖于某些相关数据发生变化的情况下，我们才能受益于这个监控数据变化的特性。

6.11.2　在 Scala 中阻塞事务

在上面 Java 版的示例中，我们使用了一个提供了很多 STM 相关的便利接口的 StmUtils 对象。而在 Scala 中，我们可以直接使用 StmUtils 里提供的各种特性（trait）。此外，我们在 Scala 中同样可以用工厂方法来创建 TransactionFactory。

usingTransactionalMemory/scala/blocking/CoffeePot.scala

```
object CoffeePot {
  val start = System.nanoTime()
  val cups = Ref(24)

  def fillCup(numberOfCups : Int) = {
    val factory = TransactionFactory(blockingAllowed = true,
      timeout = 6 seconds)

    atomic(factory) {
      if(cups.get() < numberOfCups) {
        println("retry........ at " + (System.nanoTime() - start)/1.0e9)
        retry()
      }
      cups.swap(cups.get() - numberOfCups)
      println("filled up...." + numberOfCups)
```

```
      println("........ at " + (System.nanoTime() - start)/1.0e9)
    }
  }

  def main(args : Array[String]) : Unit = {
    val timer = new Timer(true)
    timer.schedule(new TimerTask() {
      def run() {
        println("Refilling.... at " + (System.nanoTime() - start)/1.0e9)
        cups.swap(24)
      }
    }, 5000)

    fillCup(20)
    fillCup(10)
    try {
      fillCup(22)
    } catch {
      case ex => println("Failed: " + ex.getMessage())
    }
  }
}
```

在创建 TransactionFactory 对象时，我们并没有直接使用 DurationInt 来配置事务的超时时间，而是用 intToDurationInt() 函数来完成从 int 到 DurationInt 的隐式转换。通过 Scala 隐式转换所带来的语法上的便利，我们在初始化 TransactionFactory 对象时只需简单调用 6 秒即可。示例代码中余下的部分就只是从 Java 到 Scala 的一个简单的翻译而已，其最终的结果输出如下所示：

```
filled up....20
........ at 0.325964
retry........ at 0.327425
retry........ at 0.329587
Refilling.... at 5.105191
filled up....10
........ at 5.106074
retry........ at 5.106296
retry........ at 5.106466
Failed: Transaction DefaultTransaction has timed with a
total timeout of 6000000000 ns
```

6.12　提交和回滚事件

Java 的 try-cache-finally 语法结构不但使我们可以安全地处理异常，还能够在程序抛出异常时选择性地执行一些代码。同样地，我们也可以控制程序在事务成功提交之后去执行某段代码，而当事务回滚时则去执行另一段代码。StmUtils 中的 deferred() 和 compensatiing() 这两个方法分别提供了上述功能。特别，在实现事务的过程中，为保证事务能顺利完成，我们通常会加入一些带副作用的逻辑，而 deferred() 函数则是一个执行所有这部分逻辑的绝佳地点。

6.12.1　Java 中的提交和回滚事件

我们可以把想要在事务成功完成之后执行的代码放在 Runnable 接口实现部分的代码块中，并将其作为参数传给 StmUtils 的 deferred() 函数。同样，我们也可以把想要在事务失败之后执行的代码封装在 Runnable 接口中传给 compensating() 函数。由于这两个函数必须在事务的环境下运行，所以我们只有在 automically() 函数的函数体中才能调用它们。

usingTransactionalMemory/java/events/Counter.java

```
public class Counter {
  private final Ref<Integer> value = new Ref<Integer>(1);

  public void decrement() {
    new Atomic<Integer>() {
      public Integer atomically() {

        deferred(new Runnable() {
          public void run() {
            System.out.println(
              "Transaction completed...send email, log, etc.");
          }
        });
        compensating(new Runnable() {
          public void run() {
            System.out.println("Transaction aborted...hold the phone");
          }
        });

        if(value.get() <= 0)
          throw new RuntimeException("Operation not allowed");

        value.swap(value.get() - 1);
        return value.get();
      }
    }.execute();
  }
}
```

在 Counter 类的定义代码中我们看到，Counter 类仅含有一个名为 decrement() 的实例方法。在这个方法中，我们继承了 Atomic 类并实现了 atomically() 函数。在前面的例子中，我们都仅仅是简单地把事务的逻辑代码放在这个位置。而现在，除了原有的逻辑代码之外，我们把事务成功和事务回滚之后要执行的代码也放到了 atomically() 里面。下面让我们构建一个简单的测试用例来验证一下 Counter 的功能：

usingTransactionalMemory/java/events/UseCounter.java

```
package com.agiledeveloper.pcj;

public class UseCounter {
  public static void main(final String[] args) {
    Counter counter = new Counter();
    counter.decrement();
```

```
      System.out.println("Let's try again...");
      try {
        counter.decrement();
      } catch(Exception ex) {
        System.out.println(ex.getMessage());
      }
    }
  }
```

通过运行 UseCounter，我们可以清楚地观察到事务成功完成和失败时程序的执行逻辑：

```
Transaction aborted...hold the phone
Transaction completed...send email, log, etc.
Let's try again...
Transaction aborted...hold the phone
Operation not allowed
```

当第一次调用 decrement() 函数并成功完成事务之后，封装在 deferred() 函数内的代码块将被执行。而当我们第二次调用 decrement() 时，由于事务执行过程中抛出了异常，所以事务将被回滚，而封装在 compensating() 函数内的代码块也将被执行。最后我们需要注意的是，输出结果中最顶部那个非预期的重试所导致的回滚是由我们之前在 6.9 节中曾讨论过的默认优化设置所导致的。

deferred() 处理函数是一个执行事务收尾工作以便使其动作固化的绝佳地点，所以我们可以在里面随便进行打印、显示消息、发布通知以及提交数据库事务等操作。如果我们在事务之外有什么遗留的工作待完成，那么这个函数无疑是最好的完成地点。与 deferred() 类似的是，compensating() 处理函数是记录事务失败信息的好地方。此外，如果我们之前已经将非托管对象（即那些没有使用 Akka ref 进行管理的对象）与托管对象混杂在一起的话，那么这里也是纠正这一动作的合适地点，但是由于这种做法太容易出错，所以请你最好避免采用这样的设计思路。

6.12.2　Scala 中的提交和回滚事件

在 Scala 中，我们处理提交和回滚事件的方式与 Java 基本相同，唯一区别就是在 Scala 中我们可以将闭包 / 函数值直接传递给 deferred() 和 compensating()。下面让我们将 Counter 类由 Java 转译成 Scala。

usingTransactionalMemory/scala/events/Counter.scala

```
class Counter {
  private val value = Ref(1)

  def decrement() = {
    atomic {

      deferred { println("Transaction completed...send email, log, etc.") }

      compensating { println("Transaction aborted...hold the phone") }

      if(value.get() <= 0)
```

```
        throw new RuntimeException("Operation not allowed")

      value.swap(value.get() - 1)
      value.get()
    }
  }
}
```

在上面的代码中，我们将事务运行成功时所要执行的那部分代码封装在一个闭包中，然后将其作为参数传递给 deferred() 函数。类似地，事务回滚时所要执行的代码也被作为一个闭包赋给了 compensating() 函数。与此同时，这两个函数又与事务逻辑代码一起被置于表示 atomic() 函数的闭包当中。这段代码再次彰显了 Scala 在语法上简洁明了的特征。下面让我们将 UseConuter 类也从 Java 转译成 Scala：

usingTransactionalMemory/scala/events/UseCounter.scala

```
package com.agiledeveloper.pcj

object UseCounter {
  def main(args : Array[String]) : Unit = {
    val counter = new Counter()
    counter.decrement()

    println("Let's try again...")
    try {
      counter.decrement()
    } catch {
      case ex => println(ex.getMessage())
    }
  }
}
```

如下所示，Scala 版代码的执行结果与 Java 版的结果完全相同：

```
Transaction aborted...hold the phone
Transaction completed...send email, log, etc.
Let's try again...
Transaction aborted...hold the phone
Operation not allowed
```

6.13　集合与事务

在我们努力学习这些示例的过程中，很容易就会忘记我们所要处理的值都必须是不可变的。只有实体才是可变的，而状态值则是不可变的。虽然 STM 已经为我们减轻了很多负担，但如果想要在维护不可变性的同时还要兼顾性能的话，对我们来说也将是一个非常严峻的挑战。

为了保证不可变性，我们采取的第一个步骤是将单纯用来保存值的类（value classes）及其内部所有成员字段都置为 final（在 Scala 中是 val）。然后，我们需要传递地保证我们自己定义的类里面的字段所使用的类也都是不可变的。可以说，将字段和类的定义置为 final 这一步是整个过程的基础，这同时也是避免并发问题的第一步。

虽说不可变性可以使代码变得又好又安全，但是由于性能问题，程序员们还是不大愿意使用这一特性。其症结在于，为了维护不可变性，我们可能在数据没发生任何变动的情况下也要进行拷贝操作，而这种无谓的拷贝对性能伤害很大。为了解决这个问题，我们在 3.6 节中曾经讨论过持久化数据结构以及如何使用这类数据结构来减轻程序在性能方面的负担。而在持久化数据结构的实现方面，已经有很多现成的第三方库可供使用，而 Scala 本身也提供了这类数据结构。我们无需专门为使用这个特性而去换用自己不熟悉的语言，可以从 Java 代码中使用这些持久化数据结构。

除了不可变性之外，我们还希望能获得一些事务运行所需要的数据结构，这些数据结构的值是不可变的，但其实体可以在托管事务中被改变。Akka 提供了两种托管数据结构，即 TransactionalVector 和 TransactionalMap。这两种数据结构源自于高效的 Scala 数据结构，其工作原理和 Java 的 list、字典类似。下面就让我们一起来学习如何在 Java 和 Scala 中使用 TransactionalMap。

6.13.1　在 Java 中使用事务集合类

在 Java 中使用 TransactionalMap 是非常简单的。例如，下面我们一起来写一个为运动员们记录得分的程序，其中对于得分的更新操作是并发执行的。这里我们将不采用同步或锁的方式，而是把所有更新操作都放在事务中处理。示例代码如下所示：

usingTransactionalMemory/java/collections/Scores.java

```
public class Scores {
  final private TransactionalMap<String, Integer> scoreValues =
    new TransactionalMap<String, Integer>();
  final private Ref<Long> updates = new Ref<Long>(0L);

  public void updateScore(final String name, final int score) {
    new Atomic() {
      public Object atomically() {
        scoreValues.put(name, score);
        updates.swap(updates.get() + 1);
        if (score == 13)
          throw new RuntimeException("Reject this score");
        return null;
      }
    }.execute();
  }

  public Iterable<String> getNames() {
    return asJavaIterable(scoreValues.keySet());
  }

  public long getNumberOfUpdates() { return updates.get(); }

  public int getScore(final String name) {
    return scoreValues.get(name).get();
  }
}
```

在 updateScore() 函数中，我们把设置某个运动员的得分以及增加更新次数的操作都收敛到一个事务里面，该事务中所用到的 TransactionalMap 类型的 scoreValue 字段以及 Ref 类型 updates 字段都是托管类型。其中 TransactionalMap 支持普通 Map 的所有函数，只不过这些函数都是事务性的，即一旦事务回滚，我们对其进行的任何变更都将被丢弃。为了能够观察到该动作实际的效果，我们人为地设置了一个回滚条件，即当得分为 13 时，我们会先完成变更操作，然后抛异常令事务回滚。

在 Java 中，如果集合类实现了 Iterable 接口的话，我们就可以使用像 for(String name: collectionOfNames) 这样的 for-each 语句。但 TransactionalMap 是一个 Scala 集合类，并且没有直接支持这个接口。别担心，Scala 提供了一个叫做 javaConversions 的门面（facade 设计模式。——译者注），该门面提供了很多方便的函数来获取我们想要的 Java 接口。例如，我们可以使用 asJavaIterable() 函数来获取原本需要使用 getNames() 函数才能拿到的接口。

至此我们已经完成了 Scores 类的全部功能，接下来我们还需要写一个类来检验 Scores 类所实现的这些功能：

usingTransactionalMemory/java/collections/UseScores.java

```
package com.agiledeveloper.pcj;

public class UseScores {
  public static void main(final String[] args) {
    final Scores scores = new Scores();
    scores.updateScore("Joe", 14);
    scores.updateScore("Sally", 15);
    scores.updateScore("Bernie", 12);
    System.out.println("Number of updates: " + scores.getNumberOfUpdates());

    try {
      scores.updateScore("Bill", 13);
    } catch(Exception ex) {
      System.out.println("update failed for score 13");
    }

    System.out.println("Number of updates: " + scores.getNumberOfUpdates());

    for(String name : scores.getNames()) {
      System.out.println(
        String.format("Score for %s is %d", name, scores.getScore(name)));
    }
  }
}
```

上例中，我们先是添加了三个正常的运动员成绩，随后又增加了一个可以导致事务回滚的成绩。但由于事务的存在，所以最后一个成绩更新操作最终是无效的。而在代码的最后，我们会遍历并输出事务性 map 里面的所有数据。下面让我们观察一下这段代码的输出结果：

```
Number of updates: 3
update failed for score 13
Number of updates: 3
```

```
Score for Joe is 14
Score for Bernie is 12
Score for Sally is 15
```

6.13.2　在 Scala 中使用事务集合类

在 Scala 中，我们可以用与 Java 类似的方式来使用事务集合类。只不过由于这次是在 Scala 中，所以这里我们需要使用 Scala 的内部迭代器而不是 javaConversions 门面。下面让我们把 Scores 类翻译成 Scala 代码：

usingTransactionalMemory/scala/collections/Scores.scala

```
class Scores {
  private val scoreValues = new TransactionalMap[String, Int]()
  private val updates = Ref(0L)

  def updateScore(name : String, score : Int) = {
    atomic {
      scoreValues.put(name, score)
      updates.swap(updates.get() + 1)
      if (score == 13) throw new RuntimeException("Reject this score")
    }
  }

  def foreach(codeBlock : ((String, Int)) => Unit) =
    scoreValues.foreach(codeBlock)

  def getNumberOfUpdates() = updates.get()
}
```

如上所示，updateScore() 函数与 Java 版本基本是相同的。唯一有点区别的地方是，我们去掉了 getNames() 函数和 getScore() 函数，并为 foreach() 提供了内部迭代器来遍历 map 中的成绩值。我们在下面列出了 Scala 版 UseScores 类的实现，这段代码是其 Java 版代码的直译：

usingTransactionalMemory/scala/collections/UseScores.scala

```
package com.agiledeveloper.pcj

object UseScores {
  def main(args : Array[String]) : Unit = {
    val scores = new Scores()

    scores.updateScore("Joe", 14)
    scores.updateScore("Sally", 15)
    scores.updateScore("Bernie", 12)

    println("Number of updates: " + scores.getNumberOfUpdates())
    try {
      scores.updateScore("Bill", 13)
    } catch {
      case ex => println("update failed for score 13")
    }
```

```
    println("Number of updates: " + scores.getNumberOfUpdates())

    scores.foreach { mapEntry =>
      val (name, score) = mapEntry
      println("Score for " + name + " is " + score)
    }
  }
}
```

不出所料，测试用例的输出结果也与 Java 版代码如出一辙：

```
Number of updates: 3
update failed for score 13
Number of updates: 3
Score for Joe is 14
Score for Bernie is 12
Score for Sally is 15
```

6.14　处理写偏斜异常

在 6.6 节中，我们曾经简单讨论了写偏斜（write skew）以及 Clojure STM 是如何解决这个问题的。Akka 同样提供了处理写偏斜问题的支持，但是需要我们配置一下才能生效。OK，一听到配置这个词可能让你觉得有些提心吊胆，但实际操作起来其实起来还是蛮简单的。下面就让我们首先了解一下 Akka 在不进行任何配置情况下的默认行为。

让我们回顾一下之前曾经见到过的那个多个账户共享同一个联合余额最低限制例子。首先我们创建了一个名为 Portfolio 的类来保存支票账户余额和储蓄账户余额。根据银行规定，这两个账户的总余额不得低于 $1000。在 Portfolio 类的代码中我们用 Java 重新实现了 withdraw() 函数。在该函数中，我们先读取两个账户的余额，将二者相加得到总余额，并在等待一个故意插进去的延时（引入这个延时的目的是人为制造事务冲突的环境）之后，从其中一个账户余额中减掉给定数量的金额（当然，在操作之前需要判断减掉这个数量后总余额不少于 $1000）。最后需要注意的是，withdraw() 函数是在一个使用了默认设置的事务中完成上述操作的。

usingTransactionalMemory/java/writeSkew/Portfolio.java

```
Line 1  public class Portfolio {
          final private Ref<Integer> checkingBalance = new Ref<Integer>(500);
          final private Ref<Integer> savingsBalance = new Ref<Integer>(600);
    5     public int getCheckingBalance() { return checkingBalance.get(); }
          public int getSavingsBalance() { return savingsBalance.get(); }

          public void withdraw(final boolean fromChecking, final int amount) {
            new Atomic<Object>() {
   10         public Object atomically() {
                final int totalBalance =
                  checkingBalance.get() + savingsBalance.get();
                try { Thread.sleep(1000); } catch(InterruptedException ex) {}
                if(totalBalance - amount >= 1000) {
   15             if(fromChecking)
```

```
            checkingBalance.swap(checkingBalance.get() - amount);
          else
            savingsBalance.swap(savingsBalance.get() - amount);
        }
        else
          System.out.println(
            "Sorry, can't withdraw due to constraint violation");
        return null;
      }
    }.execute();
  }
}
```

下面让我们创建两个事务来并发地更改账户内的余额：

usingTransactionalMemory/java/writeSkew/UsePortfolio.java

```
public class UsePortfolio {
  public static void main(final String[] args) throws InterruptedException {
    final Portfolio portfolio = new Portfolio();

    int checkingBalance = portfolio.getCheckingBalance();
    int savingBalance = portfolio.getSavingsBalance();
    System.out.println("Checking balance is " + checkingBalance);
    System.out.println("Savings balance is " + savingBalance);
    System.out.println("Total balance is " +
      (checkingBalance + savingBalance));

    final ExecutorService service = Executors.newFixedThreadPool(10);
    service.execute(new Runnable() {
      public void run() { portfolio.withdraw(true, 100); }
    });
    service.execute(new Runnable() {
      public void run() { portfolio.withdraw(false, 100); }
    });

    service.shutdown();

    Thread.sleep(4000);

    checkingBalance = portfolio.getCheckingBalance();
    savingBalance = portfolio.getSavingsBalance();
    System.out.println("Checking balance is " + checkingBalance);
    System.out.println("Savings balance is " + savingBalance);
    System.out.println("Total balance is " +
      (checkingBalance + savingBalance));
    if(checkingBalance + savingBalance < 1000)
      System.out.println("Oops, broke the constraint!");
  }
}
```

正如我们在输出结果中所看到的那样，在默认情况下，Akka 没能避免写偏斜问题，两个事务违反了银行余额限制的规定，即都从账户里取出了钱。

```
Checking balance is 500
Savings balance is 600
Total balance is 1100
```

```
Checking balance is 400
Savings balance is 500
Total balance is 900
Oops, broke the constraint!
```

现在到了该彻底解决这个问题的时候了。让我们祭出 TransactionFactory 这个能帮助我们在程序里对事务进行配置的法宝，在 Portfolio 类的第 9 行插入一段创建工厂实例的代码，即把这句话：`new Atomic<Object>() {`

用下面这段代码替换掉：

```
akka.stm.TransactionFactory factory =
  new akka.stm.TransactionFactoryBuilder()
  .setWriteSkew(false)
  .setTrackReads(true)
  .build();

new Atomic<Object>(factory) {
```

在插进来的这几行代码中，我们创建了一个 TransactionFactoryBuilder，并将 writeSkew 和 trackReads 属性分别设置为 false 和 true。与 Clojure STM 对于 ensure 的处理类似，这两个设置项的目的是告诉事务要在其运行过程中对读操作进行追踪，同时也会使事务在读数据的过程中对账户余额变量加读锁直至提交开始为止。

除了上面提到的几处更改之外，Portfolio 和 UsePortfolio 的其他代码都保持不变。而在对事务进行了上述设置之后，其输出结果如下所示：

```
Checking balance is 500
Savings balance is 600
Total balance is 1100
Sorry, can't withdraw due to constraint violation
Checking balance is 400
Savings balance is 600
Total balance is 1000
```

由于并发执行的不可预测性，我们不能确定两个事务到底哪个会胜出。但是我们可以从输出结果中看到，在所有操作结束后两个账户的余额是不同的，而在 6.6 节的 Clojure 示例中，最终两个账户余额是相同的。我们可以通过多次运行这两个实例来观察二者之间的差异。

在本节我们是用 Java 完成整个示例的。如果换成 Scala，则我们可以使用在 6.10 节中学习的语法来配置事务的 writeSkew 和 trackReads 属性。

6.15　STM 的局限性

STM 消除了显式的同步操作，所以我们在写代码时就无需担心自己是否忘了进行同步或是否在错误的级别上进行了同步。然而 STM 本身也存在一些问题，比如在跨越内存栅栏失败或遭遇竞争条件时我们捕获不到任何有用的信息。我似乎可以听到你内心深处那个精明的程序员在抱怨“怎么会这样啊”。确实，STM 是有其局限性的，否则本书写到这里就应该结束了。STM 只适用于写冲突非常少的应用场景，如果你的应用程序存在很多写操作竞争，那么我们就需要在 STM 之外寻找解决方案了。

下面让我们进一步讨论 STM 的局限性。STM 提供了一种显式的锁无关编程模型，这种模型允许多个事务并发地运行，并且在没有发生冲突时所有事务都能毫无滞碍地运行，所以相对其他编程模型而言 STM 可以同时提供更好的并发性和线程安全方面的保障。当事务对相同对象或数据的写访问发生冲突时，只有一个事务能够顺利完成，其他事务都会被自动重做。这种重做机制延缓了写操作冲突时竞争失败的那些写者的执行，但却提升了读者和竞争操作的胜利者的执行速度。当对于相同对象的并发写操作不频繁时，其性能就不会受到太大影响。但是随着冲突的增多，程序整体性能将因此变得越来越差。

如果对相同数据有很高的写冲突概率，那么我们的应用程序轻则写操作变慢，重则会因为重试太多次而导致失败。目前在本章我们所看到的例子都是在展示 STM 的优势，但是在下面的例子中我们将会看到，虽然 STM 是易于使用的，但也并非在所有应用场景下都能得到理想的结果。

在 4.2 节的示例中，当多个线程同时访问多个目录时，我们使用 AtomicLong 来对文件大小的并发更新操作进行同步。此外，如果需要同时更新多个变量，我们也必须依赖同步才能完成。虽然表面看起来使用 STM 对这段代码进行重构似乎是个不错的选择，但大量的写冲突却使得 STM 不适用于这个应用场景。下面就让我们将上述计算目录大小的程序改用 STM 实现，并观察其运行结果是否如我们所预料的那么差。

在下面的代码中，我们没有使用 AtomicLong，而是采用了 Akka 托管引用作为 FileSizeWSTM 的属性字段。

usingTransactionalMemory/java/filesize/FileSizeWSTM.java

```
public class FileSizeWSTM {

  private ExecutorService service;
  final private Ref<Long> pendingFileVisits = new Ref<Long>(0L);
  final private Ref<Long> totalSize = new Ref<Long>(0L);
  final private CountDownLatch latch = new CountDownLatch(1);
```

为了保证安全性，pendingFileVisits 的增减都需要在事务内完成。而在之前使用 AtomicLong 时，我们只需要简单调用 incrementAndGet() 函数和 decrementAndGet() 函数就行了。但是由于托管引用都是通用的（generic），没有专门针对数字类型的处理方法，所以我们还需要针对 pendingFileVisits 进行一些额外的加工，即把对于 pendingFileVisits 的操作封装到一个单独的函数里。

usingTransactionalMemory/java/filesize/FileSizeWSTM.java

```
private long updatePendingFileVisits(final int value) {
  return new Atomic<Long>() {
    public Long atomically() {
      pendingFileVisits.swap(pendingFileVisits.get() + value);
      return pendingFileVisits.get();
    }
  }.execute();
}
```

在完成上述定义之后，访问目录和计算文件大小的函数就相对容易多了，我们只需要把程序中的 AtomicLong 替换成托管引用就好。

usingTransactionalMemory/java/filesize/FileSizeWSTM.java

```
private void findTotalSizeOfFilesInDir(final File file) {
  try {
    if (!file.isDirectory()) {
      new Atomic() {
        public Object atomically() {
          totalSize.swap(totalSize.get() + file.length());
          return null;
        }
      }.execute();
    } else {
      final File[] children = file.listFiles();

      if (children != null) {
        for(final File child : children) {
          updatePendingFileVisits(1);
          service.execute(new Runnable() {
            public void run() {
              findTotalSizeOfFilesInDir(child); }
          });
        }
      }
    }

    if(updatePendingFileVisits(-1) == 0) latch.countDown();
  } catch(Exception ex) {
    System.out.println(ex.getMessage());
    System.exit(1);
  }
}
```

最后，我们还需要写一些创建 Executor 服务池和使程序运行起来的代码：

usingTransactionalMemory/java/filesize/FileSizeWSTM.java

```
  private long getTotalSizeOfFile(final String fileName)
    throws InterruptedException {
    service  = Executors.newFixedThreadPool(100);
    updatePendingFileVisits(1);
    try {
     findTotalSizeOfFilesInDir(new File(fileName));
     latch.await(100, TimeUnit.SECONDS);
     return totalSize.get();
    } finally {
      service.shutdown();
    }
  }

  public static void main(final String[] args) throws InterruptedException {
    final long start = System.nanoTime();
    final long total = new FileSizeWSTM().getTotalSizeOfFile(args[0]);
    final long end = System.nanoTime();
    System.out.println("Total Size: " + total);
```

```
        System.out.println("Time taken: " + (end - start)/1.0e9);
    }
}
```

由于我怀疑这段代码跑起来之后可能有问题，所以如果在程序中抓到事务失败所导致的异常，我就会结束掉整个应用程序。

根据事务的定义，如果变量的值在事务提交之前发生了改变，那么事务将会自动重做。在本例中，多个线程会同时竞争修改这两个可变变量，从而导致程序运行变慢或失败。我们可以在多个不同的目录上分别运行上述示例代码来进行观察，下面就列出了该示例程序在我的电脑上计算 /etc 和 /usr 这两个目录的输出结果：

```
Total file size for /etc
Total Size: 2266408
Time taken: 0.537082

Total file size for /usr
Too many retries on transaction 'DefaultTransaction', maxRetries = 1000
Too many retries on transaction 'DefaultTransaction', maxRetries = 1000
Too many retries on transaction 'DefaultTransaction', maxRetries = 1000
...
```

从输出结果来看，STM 版本对于 /etc 目录的计算结果与之前使用 AtomicLong 的那个版本是完全相同的。但是由于会产生过多的重试操作，所以 STM 版本的运行时间要比后者慢一个数量级。而遍历 /usr 目录的运行情况则更为糟糕，有相当多的事务超过了默认的最大重试限制。虽然我们的逻辑是一抓到异常就会立即终止整个程序，但由于多个事务是并发运行的，所以在程序真正停止之前我们还是能看到多条错误信息输出到控制台。

有个别评论家曾建议说是否用 commute 代替 alter 会对解决这个问题有所帮助。请回忆我们在 6.4 节中曾讨论过的在 Clojure 中用来修改托管引用的那三个函数。由于在事务失败之后不会进行重试，所以 commute 可以提供比 alter 更高的并发度。此外，commute 也不会在没有 hold 住调用方事务的情况下单独执行提交操作。然而单纯就计算目录大小这个程序而言，使用 commute 对性能的提升十分有限。在面对层次结构复杂的大型目录时，使用该函数也无法在提供良好性能的前提下获得一致性的结果。除了将 alter 换成 commute 之外，我们还可以尝试将 atom 与 swap! 函数一起使用。虽然 atom 是不可协调并且同步的操作，但其优点是不需要使用事务。此外，atom 仅能在对单个变量（例如计算目录大小示例中用于记录目录大小的变量）的变更时使用，并且变更期间不会遇到任何事务性重试。然而，由于在对 atom 做变更时会产生对用户透明的同步操作，所以我们依然会遇到同步操作所导致的延迟问题。

由于大量线程会同时尝试更新 totalSize 变量，所以计算目录大小示例在实际执行过程中会产生非常频繁的写冲突，这也就意味着 STM 不适合于解决此问题。事实上，当读操作十分频繁且写冲突被控制在合理范围内时，STM 的性能还是不错的，同时还能帮程序员免除显式同步的负担。但是在不考虑一般应用程序中常见的其他导致延时问题的前提下，如果待解决问题中含有大量写冲突，那就请不要使用 STM，而是考虑采用我们在第 8 章中将

会讨论的角色模型来避免同步操作。

6.16　小结

STM 是一个针对并发问题的非常强大的编程模型，该模型有很多优点：

- STM 可以根据应用程序的行为来充分挖掘出其最大的并发潜力。也就是说，用了 STM 之后，我们可以无需使用过度保守的、需要预先定义的同步操作，而是让 STM 动态地管理竞争冲突。
- STM 是一种锁无关的编程模型，该模型可以提供良好的线程安全性和很高的并发性能。
- STM 可以保证实体仅能在事务内被更改。
- STM 没有显式锁意味着我们从此无需担心加锁顺序及其他相关问题。
- STM 没有显式锁的结果是无死锁的并发执行。
- STM 可以帮助我们减轻前期设计的决策负担，有了它我们就无需关心谁对什么东西上了锁，而只需放心地把这些工作交给动态隐式组合锁（implicit lock composition）。
- 该模型适用于对相同数据存在并发读且写冲突不频繁的应用场景。

如果应用程序的数据访问方式符合 STM 的适用范畴，则 STM 就为我们提供了一种处理共享可变性的高效解决方案。而如果我们的应用场景里写冲突非常多，我们可能就会更倾向于使用将在第 8 章中讨论的基于角色的模型。但在下一章，还是让我们先学习一下如何在其他 JVM 上的语言中使用 STM 编程模型。

第⑦章

在 Clojure、Groovy、Java、JRuby 和 Scala 中使用 STM

JVM 上的语言大多与 JVM 平台集成得很好，然而当我们跨越语言边界的时候，偶尔也会陷入到一些模糊地带。我们通常会有意识地根据优势和能力来挑选特定的语言，并期待该语言可以提升我们的工作效率。所以我们当然不希望让某些小问题来毁掉我们的努力。

如果某应用程序非常适合用 STM 实现的话，那么在 JVM 上采用什么语言就不应该成为我们用好 STM 的绊脚石。至于采用 STM 是否合适的问题则应该取决于应用程序的数据访问模式而不是我们用什么语言进行编码。理想情况下我们应该能够在任何 JVM 上的语言中使用 STM。

目前已有多种 STM 的实现可供我们选择。如果你使用的工作语言是 Clojure，那么最适合的选择莫过于 Clojure STM。除此之外，我们还可以通过 Multiverse 库或 Akka 库来使用 STM。

Clojure STM 在使用上是所有 JVM 语言中最简单的。这是因为 Clojure STM 所使用的两个核心组件：ref 和 dosync 都是对 Clojure API 中 clojure.lang.Ref 以及 LockingTransactions 的 runInTransaction() 函数的简单封装而已。所以，当我们定义一个 ref 时，实际上就是创建了一个 Ref 类的实例；而当我们调用 dosync() 时，我们其实调用的就是 runInTransaction() 函数。下面我们将会学习如何在 Groovy、Java、JRuby 以及其他 JVM 语言中使用 Clojure STM。

如果程序需要用到 Multiverse，那么我们可以使用其 Java API、Groovy API 或 JRuby 集成 API。除此之外，我们还可以通过 Akka 库来使用 Multiverse。

在本章中，我们将会学习如何在不同的 JVM 语言里使用 STM。你可以聚焦在与你感兴趣语言相关的章节中，其他不感兴趣的章节略过即可。

在继续学习之前我需要再次提醒你：事务应该是幂等且无任何副作用的。即我们需要保证程序中所有的值都是不可变的，而只有托管实体才是可变的。虽然这一原则已经被深深地烙印在 Clojure 中，但是如果我们选用了其他语言的话，那就需要自己来保证这一原则了。

7.1 Clojure STM

虽然 STM 的概念已经问世有一段时间了，但却是 Clojure 凭借其革命性的实体与状态分离以及高效的持久化数据结构才将 STM 真正带到了聚光灯下。由于 Clojure 中的状态是完全不可变的，这使得 STM 在 Clojure 中十分易于使用，由于值的不可变性本身就属于 Clojure 运作模式的一部分，所以我们就无需再费心去保证不可变性了。同时，出于安全方

面的考量，Clojure 是不允许在事务之外更改可变实体的。如果应用程序的访问模式确实适用于 STM，那么我们就可以安享有着优异并发性能和线程安全特性的显式锁无关编程所带来的好处。如果 Clojure 是你工作上的可选语言，那么请参考我们曾在第 6 章所展示过的示例、Clojure 的文档和相关书籍来获取进一步的信息。

7.2　Groovy 集成

如果我们正在寻找一个支持简单的元编程、动态类型同时还保有 Java 语义的语言的话，那么 Groovy 就是一个不错的选择。在本节中，我们将展示如何在 Groovy 应用程序中使用 Clojure STM，以及如何通过 Akka 来使用 Multiverse STM。如果想在程序中直接使用 Multiverse，请参阅 Multiverse 的文档。

7.2.1　在 Groovy 中使用 Clojure STM

Clojure STM 在使用上与“把 clojure.lang.Ref 实例用作托管引用”，以及“调用 LockingTransaction 的 runInTransaction() 函数来把代码段置于事务中”同样简单。下面让我们通过一个账户转账的例子来一探究竟。在本例中，我们会创建一个名为 Account 的类，该类中定义了一个用于保有其不可变状态的托管引用，即代码中的 currentBalance 变量：

polyglotSTM/groovy/clojure/Transfer.groovy

```
class Account {
  final private Ref currentBalance
  public Account(initialBalance) {
    currentBalance = new Ref(initialBalance)
  }

  def getBalance() { currentBalance.deref() }
```

在构造函数中，我们用一个起始的账户余额对引用进行初始化。而在 getBalanece() 函数中，我们调用了 deref() 函数来对 currentBalance 解引用，该函数的作用等同于我们之前在 Clojure 中所看到过的 @ 前缀。至此，我们非常轻松地完成了创建托管引用的工作，现在让我们来看看如何在事务中对托管引用进行更新操作。如果想要让一段代码在事务中运行，我们需要将其封装在 runInTransaction() 函数里，注意该函数需要接收一个实现了 java.util.current.Callable 接口的实例作为其参数。Groovy 提供了丰富的语法来帮助我们实现接口，而这些语法对于实现那些仅含有一个函数的接口来说尤为方便。我们只需要创建一个闭包，即一个代码块，并且在闭包前放一个 as 操作符就可以了。Groovy 会在后台替我们实现接口所需要的函数，并将我们放在闭包中的代码块封装到接口的实现中。下面让我们为 Account 类写一个 deposit() 函数，并把其中所有的操作都封装在一个事务里。

polyglotSTM/groovy/clojure/Transfer.groovy

```
def deposit(amount) {
  LockingTransaction.runInTransaction({
      if(amount > 0) {
```

```
        currentBalance.set(currentBalance.deref() + amount)
        println "deposit ${amount}... will it stay"
      } else {
        throw new RuntimeException("Operation invalid")
      }
  } as Callable)
}
```

当 amount 的值大于 0 时，deposit() 函数会在事务中对账户余额进行更改；否则，该函将会抛出一个异常来引发事务失败。withdraw() 函数的实现与 deposit() 函数类似，所不同的仅仅是增加了一个额外的判定条件而已。

polyglotSTM/groovy/clojure/Transfer.groovy

```
  def withdraw(amount) {
    LockingTransaction.runInTransaction({
      if(amount > 0 && currentBalance.deref() >= amount)
        currentBalance.set(currentBalance.deref() - amount)
      else
        throw new RuntimeException("Operation invalid")
    } as Callable)
  }
}
```

以上就是 Account 类的全部代码。下面让我们单独写一个函数来执行转账操作，并通过这里的逻辑来观察嵌套事务的运作过程。在 transfer() 函数中，我们会先执行存款操作、再执行取款操作。通过这种方式，我们在取款失败时就可以观察到存款操作的变更被丢弃的效果。

polyglotSTM/groovy/clojure/Transfer.groovy

```
def transfer(from, to, amount) {
  LockingTransaction.runInTransaction({
    to.deposit(amount)
    from.withdraw(amount)
  } as Callable)
}
```

现在到了对上述示例代码进行检验的时候了，让我们写一个简单的调用序列来执行两个账户之间进行转账动作。首先，我们会先进行一次可以成功完成的小额转账。然后我们会再做一次大额转账，而这次转账将会因为源账户金额不足而失败。通过这种测试，我们就能够很完整地观察到事务运作的具体效果。

polyglotSTM/groovy/clojure/Transfer.groovy

```
def transferAndPrint(from, to, amount) {
  try {
    transfer(from, to, amount)
  } catch(Exception ex) {
    println "transfer failed $ex"
  }
  println "Balance of from account is $from.balance"
  println "Balance of to account is  $to.balance"
```

```
}
def account1 = new Account(2000)
def account2 = new Account(100)
transferAndPrint(account1, account2, 500)
transferAndPrint(account1, account2, 5000)
```

上述代码执行后的输出结果如下所示：

```
deposit 500... will it stay
Balance of from account is 1500
Balance of to account is  600
deposit 5000... will it stay
transfer failed java.lang.RuntimeException: Operation invalid
Balance of from account is 1500
Balance of to account is  600
```

从余额的输出结果中可以看到，第一次小额转账成功完成了。而第二次大额转账则是存款操作先成功，但取款操作失败。事务失败的最终结果是使两个账户都保持事务开始运行之前的状态。

7.2.2　在 Groovy 中使用 Akka STM

在 Groovy 中使用 Akka API 也并不困难，只需要稍微多花点精力以及多点耐心就行了。

在 Akka 中，事务函数 atomic() 是通过一个所谓包对象（package object）暴露给外界使用的。Scala 2.8 中的包对象是以 package 类和 package$ 类的字节码形式对外发布的。而 Groovy 无法识别这些小写的类名以及名字中带 $ 的类，所以如果我们想要在 Groovy 中直接使用这些类就会遇到麻烦。

幸运的是，有一个简单的方案可以解决这个问题，那就是通过 Class.forName() 函数得到包对象的类引用。而 Groovy 的动态特性将使余下的工作变得非常简单。我们只需将事务逻辑的区块表示为一个 Scala 闭包：scala.Function0，并在调用 atomic() 函数时将其作为参数传给 atomic() 函数就可以了。除此之外，我们还想给 atomic() 函数传一个事务工厂的引用进去，而在 atomic() 函数定义中第二个参数是一个隐式可选参数，其类型正是事务工厂类。这个可选参数我们从 Groovy 里是看不出来的，所以我们需要显式地将工厂类传进去。获取事务工厂一般有两个途径，其一是通过包对象来获取一个使用默认配置的默认工厂；另外我们还可以手动创建工厂实例，并将该实例按照我们所需要的方式进行配置。下面让我们重写 Account 类，并把上述所有内容都融汇其中：

polyglotSTM/groovy/akka/Transfer.groovy

```
class Account {
  private final Ref currentBalance
  private final stmPackage = Class.forName('akka.stm.package')
  public Account(initialBalance) { currentBalance = new Ref(initialBalance) }
  def getBalance() { currentBalance.get() }
  def deposit(amount) {
    stmPackage.atomic({
      if(amount > 0) {
        currentBalance.set(currentBalance.get() + amount)
```

```
        println "deposit ${amount}... will it stay"
      }
    } as scala.Function0, stmPackage.DefaultTransactionFactory())
  }
  def withdraw(amount) {
    stmPackage.atomic({
      if(amount > 0 && currentBalance.get() >= amount)
        currentBalance.set(currentBalance.get() - amount)
      else
        throw new RuntimeException("Operation invalid")
        } as scala.Function0, stmPackage.DefaultTransactionFactory())
  }
}
```

在上面的示例代码中，我们将 Scala 包对象的 Java 类名传给 forName 函数，并通过 Class 这个元对象得到我们想要的引用 stmPackage。随后我们只需要将代码闭包和 stmPackage.DefaultTransactionFactory() 这两个参数传递给 stmPackage.atomic() 函数并简单调用一下即可。上述准备工作完成之后，我们在中闭包中所写的代码就可以运行在一个由 Akka/Multiverse 托管的事务里面了。

接下来我们要实现 transfer() 函数和一些测试用例。这里的逻辑和前面一样还是先存钱后取钱，并且两个操作都是在一个事务中完成的。

polyglotSTM/groovy/akka/Transfer.groovy

```
def transfer(from, to, amount) {
  def stmPackage = Class.forName('akka.stm.package')
  stmPackage.atomic({
    to.deposit(amount)
    from.withdraw(amount)
    } as scala.Function0, stmPackage.DefaultTransactionFactory())
}

def transferAndPrint(from, to, amount) {
  try {
    transfer(from, to, amount)
  } catch(Exception ex) {
    println "transfer failed $ex"
  }

  println "Balance of from account is $from.balance"
  println "Balance of to account is  $to.balance"
}

def account1 = new Account(2000)
def account2 = new Account(100)

transferAndPrint(account1, account2, 500)
transferAndPrint(account1, account2, 5000)
```

下面让我们运行 Groovy 代码并观察事务的运行情况：

```
deposit 500... will it stay
deposit 500... will it stay
Balance of from account is 1500
```

```
Balance of to account is  600
deposit 5000... will it stay
transfer failed java.lang.RuntimeException: Operation invalid
Balance of from account is 1500
Balance of to account is  600
```

7.3 Java 集成

尽管 Java 已经是一个老牌语言了，但它仍然是应用最为广泛的编程语言之一。本节我们将会看到，并发模型的选择并不会影响到我们对于工作语言的选择。马上我们就将接触到 Java 中的 STM 模型了，正如我们即将看到的那样，虽然我们不得不忍受 Java 引入的一些繁冗的语法以及需要人工确保永恒不变性的问题，但是 STM API 本身还是很易于使用的。

7.3.1 在 Java 中使用 Clojure STM

由于 Ref 和 LockingTransaction 都是作为简单的类曝露给外界使用的，所以对于 Java 开发者来说，Java 版的 Clojure STM 是十分易用的。由于 runInTransaction() 函数需要一个实现了 Callable 接口的实例作为其参数，所以将代码封装在一个事务中其实和把代码包装在 Callable 接口的 call() 函数里同样简单。

下面我们将会用 Java 版的 Clojure STM 来完成账户转账的示例。首先我们需要创建一个 Account 类，该类中定义了一个用于保有其不可变状态的托管引用，即代码中的 balance 变量：

polyglotSTM/java/clojure/Account.java

```
public class Account {
  final private Ref balance;

  public Account(final int initialBalance) throws Exception {
    balance = new Ref(initialBalance);
  }

  public int getBalance() { return (Integer) balance.deref(); }
```

在 Account 的构造函数中，我们用一个初始的余额值对托管引用进行了初始化。而在 getBalanece() 函数中，我们调用了 deref() 函数来获取当前的账户余额，该函数相当于 Clojure 中用来解引用的 @ 前缀在 Java 侧的 API。在下面 deposit() 函数的实现中我们将会看到，那些需要运行在事务中的代码将会被置于 Callable 接口的匿名内部类的 call() 函数中，并最终传递给 runInTransaction() 函数来执行。

polyglotSTM/java/clojure/Account.java

```
public void deposit(final int amount) throws Exception {
  LockingTransaction.runInTransaction(new Callable<Boolean>() {
    public Boolean call()  {
      if(amount > 0) {
        final int currentBalance = (Integer) balance.deref();
        balance.set(currentBalance + amount);
```

```
        System.out.println("deposit " + amount + "... will it stay");
        return true;
      } else throw new RuntimeException("Operation invalid");
    }
  });
}
```

当 amount 的值大约 0 时，deposit() 函数就会在事务中对账户余额进行更改；否则，该函数将会抛出一个异常并导致事务失败。withdraw() 函数的实现与 deposit() 函数差不多，所不同的仅仅是增加了一个额外的判定条件而已。

polyglotSTM/java/clojure/Account.java

```
public void withdraw(final int amount) throws Exception {
  LockingTransaction.runInTransaction(new Callable<Boolean>() {
    public Boolean call() {
      final int currentBalance = (Integer) balance.deref();
      if(amount > 0 && currentBalance >= amount) {
        balance.set(currentBalance - amount);
        return true;
      } else throw new RuntimeException("Operation invalid");
    }
  });
}
}
```

至此 Account 类的实现就基本完成了，下面让我们把目光转移到转账逻辑上面。我们将把转账相关的逻辑单独放在 Transfer 类里。为了能观察到取款失败后存款操作的变更被丢弃的效果，我们将先执行存款操作、然后再执行取款操作：

polyglotSTM/java/clojure/Transfer.java

```
public class Transfer {
  public static void transfer(
    final Account from, final Account to, final int amount)
    throws Exception {
    LockingTransaction.runInTransaction(new Callable<Boolean>() {
      public Boolean call() throws Exception {
        to.deposit(amount);
        from.withdraw(amount);
        return true;
      }
    });
  }
```

我们还需要写一个函数来处理转账失败的情况并在转账结束后输出账户的状态：

polyglotSTM/java/clojure/Transfer.java

```
public static void transferAndPrint(
  final Account from, final Account to, final int amount) {
  try {
    transfer(from, to, amount);
  } catch(Exception ex) {
    System.out.println("transfer failed " + ex);
```

```
    }
    System.out.println("Balance of from account is " + from.getBalance());
    System.out.println("Balance of to account is " + to.getBalance());
  }
```

最后，让我们先执行一个成功的小额转账，然后再执行一个会因为源账户余额不足而失败的大额转账。

polyglotSTM/java/clojure/Transfer.java

```
  public static void main(final String[] args) throws Exception {
    final Account account1 = new Account(2000);
    final Account account2 = new Account(100);

    transferAndPrint(account1, account2, 500);
    transferAndPrint(account1, account2, 5000);
  }
}
```

下面让我们运行上述代码并观察输出结果：

```
deposit 500... will it stay
Balance of from account is 1500
Balance of to account is 600
deposit 5000... will it stay
transfer failed java.lang.RuntimeException: Operation invalid
Balance of from account is 1500
Balance of to account is 600
```

正如我们所预想的那样，第一个小额转账操作成功完成了，而第二个大额转账操作由于源账户余额不足而失败。从输出结果中我们可以看到，在取款操作导致的事务回滚之后，由存款操作所产生的变更将会全部被丢弃，此时两个账户内的余额仍然和事务执行前保持一致。

7.3.2　在 Java 中使用 Multiverse/Akka STM

如果没有使用角色（详情请参阅第 8 章）的需求，我们可以直接考虑使用 Multiverse STM，因为该类库提供了基于 Java 的注解语法和 API。如果你的计划需要到用到角色或需要将角色与 STM 混合使用，那么 Akka 库是一个不错的选择。虽然 Akka 是在 Scala 上构建的，但是其开发者也为 Java 程序员提供了大量好用的 Java API。本书第 6 章中提供了很多在 Java 中使用 Akka 库的示例，可以帮助你快速入门。

7.4　JRuby 集成

JRuby 在引入了 Ruby 强大功能的同时，还将其优雅、表达能力强以及简洁的语法特性带到了 Java 平台。在本节中，我们将学习如何在 JRuby 代码里使用 Clojure STM 和通过 Akka 库使用 Multiverse STM。如果你希望直接使用 Multiverse，请参阅 Multiverse 帮助文档中关于 JRuby 集成 API 的相关章节。

7.4.1　在 JRuby 中使用 Clojure STM

在 JRuby 中使用 Clojure STM 时，我们可以将 clojure.lang.Ref 用作托管引用，并将 LockingTransaction 的 runInTransaction() 函数用来实现事务。下面让我们通过账户转账示例来熟悉 Clojure STM 的用法。首先我们需要创建一个名为 Account 的类，该类持有一个指向其不可变状态的托管引用，其中 Account 的状态是由 JRuby 字段 @balance 表示的：

polyglotSTM/jruby/clojure/account.rb

```
require 'java'
java_import 'clojure.lang.Ref'
java_import 'clojure.lang.LockingTransaction'

class Account
  def initialize(initialBalance)
    @balance = Ref.new(initialBalance)
  end

  def balance
    @balance.deref
  end
```

在 initialize() 函数中，我们用一个初始余额对托管引用进行了初始化。而如果我们想要获得余额的值，则需要使用 deref() 函数来解引用，这一做法与我们在 Java 版本的实现中是类似的。此外，之前如果我们想让某段代码运行在一个事务中，就必须将这段代码封装在 Callable 类里面，并将该类的实例作为参数赋给 runInTransaction() 函数。这种做法使得代码看起来既复杂又臃肿，而 JRuby 在接口实现方面的用法则十分简便，我们只需要把代码块丢给 runInTransaction() 函数，JRuby 就会自动帮我们将代码块封装到一个接口实现中去。下面让我们来实现 Account 类的 deposit() 函数，该函数的所有操作都需要在同一个事务内完成：

polyglotSTM/jruby/clojure/account.rb

```
def deposit(amount)
  LockingTransaction.run_in_transaction do
    if amount > 0
      @balance.set(@balance.deref + amount)
      puts "deposited $#{amount}... will it stay"
    else
      raise "Operation invalid"
    end
  end
end
```

如果账户没有因为欠款而遭银行禁用的话，存款函数 deposit() 就会启动一个事务，并在该事务中修改余额，否则，程序将通过抛异常的方式使事务失败。取款函数 withdraw() 的实现也与之类似：

polyglotSTM/jruby/clojure/account.rb

```
  def withdraw(amount)
    LockingTransaction.run_in_transaction do
      if amount > 0 && @balance.deref >= amount
        @balance.set(@balance.deref - amount)
      else
        raise "Operation invalid"
      end
    end
  end
end
```

至此 Account 类就基本完成了。接下来，为了能够嵌套事务里，我们还需要实现一个单独的转账函数 transfer()。同时，为了能够观察到取款操作失败后事务回滚的效果，我们将采用先存款后取款的顺序来执行转账操作。

polyglotSTM/jruby/clojure/account.rb

```
def transfer(from, to, amount)
  LockingTransaction.run_in_transaction do
    to.deposit(amount)
    from.withdraw(amount)
  end
end
```

最后，为了验证上述代码是否能够正常工作，我们还需要写一个在账户之间转账的测试用例。该用例的测试流程为：首先我们会进行一次成功的转账，随后再人为制造一次由于余额不足而失败的转账。该用例很好地展示了事务对程序运行所产生的影响。

polyglotSTM/jruby/clojure/account.rb

```
def transfer_and_print(from, to, amount)
  begin
    transfer(from, to, amount)
  rescue => ex
    puts "transfer failed #{ex}"
  end

  puts "Balance of from account is #{from.balance}"
  puts "Balance of to account is #{to.balance}"
end

account1 = Account.new(2000)
account2 = Account.new(100)

transfer_and_print(account1, account2, 500)
transfer_and_print(account1, account2, 5000)
```

运行上述代码所得到的输出结果如下所示：

```
deposited $500... will it stay
Balance of from account is 1500
Balance of to account is 600
deposited $5000... will it stay
transfer failed Operation invalid
```

```
Balance of from account is 1500
Balance of to account is 600
```

从输出结果我们可以看出，示例代码的行为与我们所期望的完全一致，第一个事务是成功的，转账事务完成后账户余额的变化体现了这一点。第二个转账事务不出所料地失败了，而参与转账的账户余额则保持不变。在实际的执行过程中，虽然第二个转账事务中的存款动作已经成功完成，但当事务失败时，该动作所造成的变更最终被丢弃。

7.4.2　在 JRuby 中使用 Akka STM

在 JRuby 中使用 Multiverse STM 相比使用 Clojure STM 来说稍显复杂并且更需要多加练习。其主要原因是 Multiverse 依赖异常来实现事务的重试，但 JRuby 将异常都封装成了 NativeException。所以一旦我们不小心搞错的话，Multiverse 就没法看到其希望捕获的异常并最终导致事务无法提交。在本节的示例中，我们会先面对该问题，然后再想办法来解决它。

在 Akka 中，事务性函数 atomic() 可以经由包对象 akka.stm.package 暴露给外界使用。Scala 包对象是由一对字节码形式的类构成的，即 package 类和 package$ 类。如果在 JRuby 代码中引入这个 package 类，那么我们在编译时将会收到一个怪异的错误提示：cannot import class ‘package’ as ‘package’。为了修复这个错误，我们需要将包名进行重定义，比如我们可以这样做：java_import ‘akka.stm.package’ do |pkgname,classname|"J#{classname}" end。这样一来，当我们在 JRuby 代码中引用 Jpackage 时，实际上就是在引用 Akka 的 akka.stm.package 包对象。

下面让我们用 JRuby 和 Akka STM 重写上面的账户转账示例。首先我们把创建 Akka 事务的函数单独封装成一个独立的模块，以方便其他模块重用这段代码：

polyglotSTM/jruby/akka/broken/akka_stm.rb

```
require 'java'
java_import 'akka.stm.Ref'
java_import 'akka.stm.package' do |pkgname, classname| "J#{classname}" end

module AkkaStm
  def atomic(&block)
    Jpackage.atomic Jpackage.DefaultTransactionFactory, &block
  end
end
```

在上面这段代码中，我们引入了 Akka 的 Ref 类和包对象 akka.stm.packaged，并在 AkkaStm 模块中创建了一个名为 atomic() 的函数，其中 atomic() 函数的主要功能是接受一个代码块并将其传递给 akka.stm.packages 的 atomic() 函数。由于 Scala 包对象暴露出来的这个 atomic() 函数还需要一个工厂类作为参数，所以我们在调用该函数时还传给它一个包对象本身自带的默认事务工厂。接下来就让我们用上面定义的这个 atomic 函数来实现对 Account 类中相关函数的事务化处理：

polyglotSTM/jruby/akka/transfer.rb

```
require 'akka_stm'

class Account
  include AkkaStm
  def initialize(initialBalance)
    @balance = Ref.new(initialBalance)
  end
  def balance
    @balance.get
  end
  def deposit(amount)
    atomic do
      if amount > 0
        @balance.set(@balance.get + amount)
        puts "deposited $#{amount}... will it stay"
      end
    end
  end
  def withdraw(amount)
    atomic do
      raise "Operation invalid" if amount < 0 || @balance.get < amount
      @balance.set(@balance.get - amount)
    end
  end
end
```

如上所示，我们先是为@balance字段创建了一个托管引用，随后又通过我们在AkkaStm模块中实现的atomic()函数将deposit()和withdraw()函数封装到各自的事务中。值得注意的是，这里我们是通过include这个JRuby的mixin工具将AkkaStm模块引入到Account类的实现中来的。同样，我们可以将transfer()函数作为一个单独的方法实现，我们还可以通过mixin工具引入AkkaStm模块来复用atomic()函数：

polyglotSTM/jruby/akka/transfer.rb

```
def transfer(from, to, amount)
  include AkkaStm
  atomic do
    to.deposit(amount)
    from.withdraw(amount)
  end
end
```

最后，我们通过调用两组转账操作来对上述代码的功能进行检验：

polyglotSTM/jruby/akka/transfer.rb

```
def transfer_and_print(from, to, amount)
  begin
    transfer(from, to, amount)
  rescue => ex
    puts "transfer failed #{ex}"
  end
```

```
  puts "Balance of from account is #{from.balance}"
  puts "Balance of to account is #{to.balance}"
end

account1 = Account.new(2000)
account2 = Account.new(100)

transfer_and_print(account1, account2, 500)
transfer_and_print(account1, account2, 5000)
```

由于待转金额小于 account1 的账户余额，所以我们预计第一个转账操作是可以成功完成的。而对于第二个转账操作，由于待转金额远远大于 account1 的账户余额，所以这次转账操作将会失败，并且两个账户的余额都会恢复到第二次转账操作开始之前的状态。下面就让我们运行这段代码并观察其输出结果：

```
transfer failed
  org.multiverse.api.exceptions.SpeculativeConfigurationFailure: null
...
```

上面的输出并非我们预期的结果。而更加奇怪的是，输出结果显示本该成功的第一个转账事务却失败了，要定位这个错误将耗费我们大量宝贵时间并会使我们因思虑过度而脱发。这就是我们在本节一开始就提到过的那个问题，即导致事务失败的原因是程序抛出了一个定义在 org.multiverse.api.exceptions 包里的 SpeculativeConfigurationFailure 异常。下面让我们一起追根寻源，找到这个异常是哪里抛出来的，并了解为何在其他语言的版本里运行无误的代码放到 JRuby 里就突然失败。

默认情况下，Multiverse（Akka STM 也是类似）使用了激进的事务策略（speculative transaction），该策略我们曾在 6.9 节中进行过详细讨论，这里就不再赘述。正是由于该策略导致了 Multiverse 在事务开始阶段会假设事务是只读的，所以当 Multiverse 执行到对托管引用的第一个 set 操作时，它就会抛 SpeculativeConfigurationFailure 异常。在 Multiverse 层面，它会先自己处理这个异常并重做事务，而在重做时 Multiverse 将允许事务对托管引用进行变更。这也是我们曾经在第 6 章中曾经提到的导致事务重做的原因之一。这就是 Multiverse 的实现逻辑，并且这套机制目前看来在其他语言中貌似也是没问题的，所以下面让我们一起来理清到底在 JRuby 发生了什么事情。

如果我们将 puts "transfer failed #{ex}" 替换成 puts "transfer failed #{ex.class}"，就会发现异常类的名字不是 SpeculativeConfigurationFailure 而是 NativeException。这是由于 JRuby 将原始的异常替换成了它自己包装过的异常类，而问题的根源也就在这里。当我们在 JRuby 代码中调用 Akka 库的 atomic 函数时，由于 atomic 的参数是一个 JRuby 代码闭包，所以在事务开始执行之后程序流程就又会回到 JRuby 闭包中。而当这段 JRuby 代码试图更新托管引用的时候，Multiverse 将会抛出异常，紧接着我们在上一段中提到过的 Multiverse 的异常重做机制将会生效。但不幸的是，我们的 JRuby 闭包代码将会把 Multiverse 抛出来的异常在内部包装成 NativeException。所以，当 Multiverse 看到的不是自己熟悉的 SpeculativeConfigurationFailure 而是一个陌生的 NativeException 时，它就不会再

重做事务而是简单地将事务终止并把异常沿着调用链向上传播。

那么怎样才能修复这个问题呢？有一个不怎么好的方法是：显式地对异常进行处理。我们需要检查抛出来的异常是否属于 Multiverse，而如果是的话，我们就得将该异常解包装（unwrap）然后再重新抛出去。这显然是一个丑陋的权宜之计，但是胜在需要写的代码很少。下面让我们尝试用这种方法修改 AkkaStm 的 atomic() 函数来对抵消 JRuby 的行为对 Multivverse 异常机制的破坏。

polyglotSTM/jruby/akka/akka_stm.rb

```
require 'java'
java_import 'akka.stm.Ref'
java_import 'akka.stm.package' do |pkgname, classname| "J#{classname}" end
module AkkaStm
  def atomic(&block)
    begin
      Jpackage.atomic Jpackage.DefaultTransactionFactory, &block
    rescue NativeException => ex
      raise ex.cause if
        ex.cause.java_class.package.name.include? "org.multiverse"
      raise ex
    end
  end
end
```

首先，我们将 akka.stm.package.atomic 函数的调用包装在一个 begin-rescue 区块中，如果该调用抛出的异常是一个 NativeException 并且其内嵌的实际类型是一个 Multiverse 异常，则我们就将该异常解包装以后抛给上层调用者。这样一来，Multiverse 就能够识别出该异常并执行合适的动作。

至此该问题就已修复完毕，下面让我们一起来看看之前测试用例里的两个事务的行为是否符合预期，即第一个事务成功而第二个事务失败并且两个账户的余额与第二个事务执行前保持不变：

```
deposited $500... will it stay
deposited $500... will it stay
Balance of from account is 1500
Balance of to account is 600
deposited $5000... will it stay
transfer failed Operation invalid
Balance of from account is 1500
Balance of to account is 600
```

从输出结果中可以看出，在修复了上述问题之后，JRuby 版本的行为与其他语言实现版本的表现基本一致。

7.5　Scala 中的可选方案

Scala 是 JVM 上的一种静态类型语言，该语言的特点是将面向对象风格的编程方法与函数式风格的编程方法进行了有机的结合。在这个表达能力极强的语言中，我们有很多种

STM 的实现方案可供选择。在这些备选方案中，由于 Akka 提供的 Scala API 使用起来最为顺手，所以我们更倾向于选择 Akka。而即便我们接手的是已经使用了 Clojure STM 的多语言项目，我们在 Scala 中使用 Clojure STM 也是毫无问题的。下面就让我们一起研究一下如何在 Scala 中使用 Clojure STM。

7.5.1　在 Scala 中使用 Clojure STM

在 Scala 中使用 Clojure STM 的方法与在 Java 中十分相似。在 Scala 中，我们同样可以使用 Ref 类和 LockingTransaction 类的 runInTransaction() 函数。此外，我们还需要创建一个实现了 Callable 接口的实例，以便将需要在事务中运行的函数包装起来。下面让我们用 Scala 和 Clojure STM 来重新实现账户转账示例。该示例的代码其实基本上都是从 Java 到 Scala 的直译。

polyglotSTM/scala/clojure/Transfer.scala

```
class Account(val initialBalance : Int) {
  val balance = new Ref(initialBalance)

  def getBalance() = balance.deref
```

在上述用 Scala 实现的 Account 类定义中，在构造函数片段中我们用一个初始余额值对 Account 的内部引用进行了初始化。而在 getBalance() 函数中，我们通过调用 deref() 函数来获取当前账户的余额值。如果想要在事务中更新托管引用，我们只需将操作代码包装成一个 Callable 接口的实例对象并将其传递给 runInTransaction() 函数即可。下面让我们先来实现需要运行在事务中的存款函数 deposit()：

polyglotSTM/scala/clojure/Transfer.scala

```
def deposit(amount : Int) = {
  LockingTransaction runInTransaction new Callable[Boolean] {
    def call() = {
      if(amount > 0) {
        val currentBalance = balance.deref.asInstanceOf[Int]
        balance.set(currentBalance + amount)
        println("deposit " + amount + "... will it stay")
        true
      } else throw new RuntimeException("Operation invalid")
    }
  }
}
```

取款函数 withdraw() 的代码与存款函数大同小异：

polyglotSTM/scala/clojure/Transfer.scala

```
  def withdraw(amount : Int) = {
    LockingTransaction runInTransaction new Callable[Boolean] {
      def call() = {
        val currentBalance = balance.deref.asInstanceOf[Int]
        if(amount > 0 && currentBalance >= amount) {
          balance.set(currentBalance - amount)
```

```
            true
          } else throw new RuntimeException("Operation invalid")
        }
      }
    }
  }
```

至此我们就完成了 Account 类的定义。我们可以把 transfer() 函数作为一个单独的函数来实现，如果我们只想将本示例作为一个脚本通过命令行来运行的话，那就只需要把该方法实现出来即可，并不需要为此编写一个类；而如果我们想要将示例代码进行编译的话，就需要将其封装到一个对象中并写一个 main() 函数来启动整个示例程序。

polyglotSTM/scala/clojure/Transfer.scala

```
def transfer(from : Account, to : Account, amount : Int) = {
  LockingTransaction runInTransaction new Callable[Boolean] {
    def call() = {
      to.deposit(amount)
      from.withdraw(amount)
      true
    }
  }
}
```

在 transfer() 函数中，我们会先执行存款操作然后再执行取款操作。与之前的示例一样，我们希望通过一次成功、一次失败的两个转账动作来研究事务的执行效果。

polyglotSTM/scala/clojure/Transfer.scala

```
def transferAndPrint(from : Account, to : Account, amount : Int) = {
  try {
    transfer(from, to, amount)
  } catch {
    case ex => println("transfer failed " + ex)
  }
  println("Balance of from account is " + from.getBalance())
  println("Balance of to account is " + to.getBalance())
}

val account1 = new Account(2000)
val account2 = new Account(100)

transferAndPrint(account1, account2, 500)
transferAndPrint(account1, account2, 5000)
```

下面让我们运行上述代码并观察输出结果：

```
deposit 500... will it stay
Balance of from account is 1500
Balance of to account is 600
deposit 5000... will it stay
transfer failed java.lang.RuntimeException: Operation invalid
Balance of from account is 1500
Balance of to account is 600
```

上述执行结果符合我们的预期，即第一个转账操作成功完成，并且事务结束后两个账

户的余额正确反映了转账动作；第二个转账动作则以失败告终，并且两个账户的余额与第二个转账动作发生前保持一致。

7.5.2　在 Scala 中使用 Akka/Multiverse

由于 Akka 本身就是用 Scala 实现的，并且其暴露给用户的 API 对 Scala 开发者而言也是非常顺手，因此 Akka 是 Scala 中 STM 工具的不二之选。你可以通过第 6 章中的 Scala 示例来初步学习如何使用 Akka 所提供的 STM 功能。作为一个混合函数式（hybrid functional）编程语言，我们在 Scala 中既可以创建可变 (var) 变量也可以创建不可变值 (val)。而在实际工作中我们还是更提倡不可变性，即多使用 val 而少使用 var。请快速地 grep 或搜索一下你的项目代码，检查一下是否里面使用了 var 以及用得是否正确。检验原则是，确保只有实体 Ref 是可变的，而所有状态都是不可变的。

7.6　小结

在本章中，我们学到了如下几个知识点：

- 并发能力的实现是和语言无关的，即我们可以在任何 JVM 语言里用 STM 实现并发。
- 在使用不同语言实现并发的集成过程中，虽然偶尔会遇到一些类库和语言集成方面的问题，但是我们都有很简单的解决方案来绕过这些问题。
- 我们必须保证所有的值都是不可变的，并且事务都是幂等的，

对于那些读频繁且写冲突相对较少的应用程序而言，我们可以在任何 JVM 语言中使用 STM 来实现高效的并发操作。但是在面对有大量写冲突的应用场景时，STM 可能就不再适用。在这种情况下，为了避免再次陷入同步以及共享可变性的梦魇，我们可以尝试使用下一章中讨论的基于角色的编程模型。

第四部分

基于角色的并发模型

第⑧章

讨喜的隔离可变性

曾有个的医嘱是这样说的：“如果它伤到了你，那就别再用它了。”在并发编程领域，共享可变性就是那个“它”。

虽然 JDK 的线程 API 使我们可以非常容易地创建线程，但如何防止线程冲突和逻辑混乱却又成了大问题。STM 虽然可以解决部分问题，但是在一些类似 Java 这样的语言中，我们仍不得不非常小心谨慎地避免非托管可变变量和事务逻辑中产生某些副作用。而令人惊讶的是，当共享可变性消失的时候，所有那些令人纠结的问题也都随之消失了。

事实证明，在相同数据上让多个线程互相冲突地执行是行不通的。幸运的是我们有更好的办法，即基于事件的消息传递。通过这种方法，我们可以将任务当成是应用程序 /JVM 内部的轻量进程来对待。同时，我们将不可变消息传递给各个任务，从而避免每个任务都去抢占共享数据。一旦这些异步任务执行完毕，它们会将不可变的执行结果返回给另外的协调线程。在下面的章节中我们将学习如何设计这种带有“用于异步交换不可变消息的协调角色（actor）[⊖]”的应用程序。

虽然这种方法已经问世多年了，但是在 JVM 领域却还是相对较新的技术。基于角色的模型在 Erlang 中是非常成功的（参见《Programming Erlang： Software for a Concurrent World》[Arm07] 以及《Concurrent Programming in Erlang》[VWWA96]）。而当 Scala 2003 年被引入时，Erlang 的基于角色的模型也同时被采纳并引入 JVM 大家庭中（参见《Programming in Scala》

⊖ 有人曾经问我这些角色与用例中其他角色如何交互的问题，答案是什么交互也不做。这些角色只对它们收到的消息起作用，即收到消息之后执行一些专门的任务，然后将响应消息传递给其他角色……如此周而复始地顺序执行。

[OSV08] 以及《Programming Scala》[Sub09]）。

在 Java 中，有大把基于角色的并发库[⊖]可供我们选择：ActorFoundary、Actorom、Actors Guild、Akka、FunctionalJava、Kilim、Jetlang 等。其中部分类库是以面向切面（AspectOriented）的字节码形式来进行组织的。同时，这些类库的成熟度和被大家的接受度也都不尽相同。

在本章中，我们将学习如何编写基于角色的并发程序。在大多数情况下，我们将使用 Akka 以便能够直奔核心概念。Akka 是一个基于 Scala 的高性能解决方案，同时还提供了相当好用的 Java API。我们既可以将其用于基于角色的并发，也可以将之用于 STM（参见第 6 章）。

8.1　用角色实现隔离可变性

Java 将 OOP 变成了可变性驱动（mutability-driven）的开发模式[⊜]，而函数式编程则着重强调不可变性，而这两种极端的方式其实都是有问题的。如果每样事物都是可变的，那么我们就需要妥善处理可见性和竞争条件。而在一个真实的应用程序中，也并非所有事物都是不可变的。即使是纯函数式语言也提供了代码限制区，在该区域内允许出现带副作用的逻辑以及按顺序执行这些逻辑的方法。但无论我们倾向于哪种编程模型，避免共享可变性都是毋庸置疑的。

共享可变性是并发问题的根源所在，是指多个线程可以同时更改相同的变量。而隔离可变性是一个可以消除大部分并发问题的不错的折衷方案，是指任意时刻有且只有一个线程（或角色）可以访问某个可变变量。

在 OOP 中，由于对象的状态都被封装在对象中，所以只有实例方法才能够操作对象的状态。然而不同的线程可以同时调用这些方法，从而导致并发问题的发生。在基于角色的编程模型中，我们只允许一个角色操作对象的状态。因此，即使整个应用程序是多线程的，那些角色本身也都是单线程的，所以也就不会有任何可见性和竞争条件相关的问题。虽然角色都可以发出操作执行请求，但它们却无法接触到其他角色托管的可变状态。

我们在使用基于角色的模型进行编程时，往往会采取与普通的面向对象编程不同的设计方法。即将问题拆分成若干个可计算的异步任务，并将它们赋予不同的角色。其中，每个角色只负责执行分配给自己的那部分任务。这样我们就可以将任意可变状态限定在至多一个角色当中（见图 8-1）。此外，我们还需要保证在角色之间传递的消息都是完全不可变的才行。

在这种设计方法中，我们令每个角色都负责解决问题的一个组成部分，而它们所接收的必要数据则都当做不可变对象来处理。一旦完成了分配给自己的任务，这些角色就会把处理结果封装到不可变对象中并返回给调用角色或其他特定的后置处理（post-processing）角色。我们可以将这种方式想象为 OOP 进化版，即我们让可变且活动的对象分别运行在属

⊖　试想一下，如果我们只能选择其中一种方案，那将会是多么令人头痛的问题。

⊜　在这个问题上 Java 还有其他同案犯，所以我们不应该把所有责任都归咎于 Java。

于自己的线程里。在这种模式下，对对象进行操作的唯一方法是将消息传递给它们而不是直接调用其成员函数。

8.2　角色的特性

角色是一种能够接收消息、处理请求以及发送响应的自由运行的活动（activity），主要被设计用来支持异步化且高效的消息传递机制。

每个角色都有一个内建的消息队列，该队列与手机上所使用的短信队列十分相似。假设 Sally 和 Sean 同时给 Bob 的手机发了短信，则运营商将会把这两条短信都保存起来以便 Bob 在方便的时候取走。类似地，基于角色的并发库允许多个角色并发地发送消息。默认情况下，消息发送者都是非阻塞的，它们会先把消息发送出去，然后再继续处理自己的业务逻辑。类库一般会让特定的角色顺序地拾取并处理消息队列中消息，只有将当前消息处理完或将消息委派给其他角色进行并发处理之后，这个角色才能够接收下一个消息。

角色的生命周期如图 8-2 所示。一个角色在被创建出来之后既可以被启动也可以被终止。一旦被启动，角色即已准备就绪，随时可以接收消息。当角色处于活跃状态时，则不是在处理消息就是在等待新消息到达。而一旦停止之后，角色就不会再接收任何消息了。就角色的整体生命周期而言，其用于等待和处理消息的耗时比取决于它们所处的应用程序的动态特性。

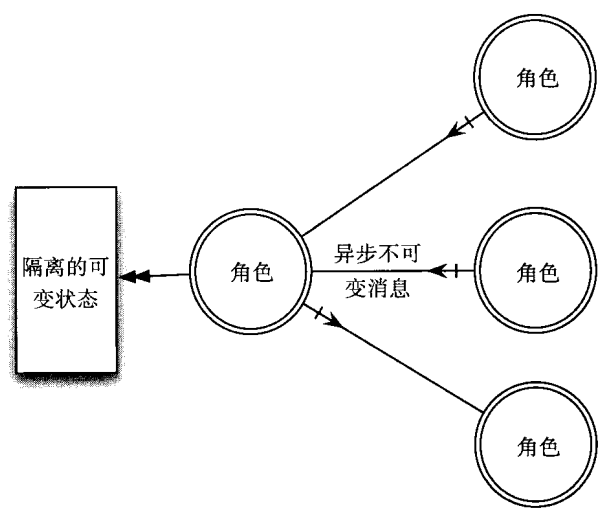


图 8-1　角色对可变状态进行了隔离，不同角色之间通过传递不可变消息来进行通信

如果角色在程序的设计中起到了举足轻重的作用，那么我们就会期望在程序执行过程中创建足够多可供使用的角色。然而线程是有限的资源，所以我们不能把角色与线程一对一地捆绑在一起。为了避免资源不足的问题，支持角色的类库通常都会将角色与线程解耦。角色与线程之间的关系好比公司食堂和公司雇员。例如，Bob 在其公司食堂里是没有专门

的座位的（如果他想要这种待遇的话恐怕得另找一份工作了），所以每次他去食堂吃饭都是随便找一个没人占的座位就行了。与此相类似地，当收到一个待处理的消息或待运行的任务时，角色就可以分配到一个可用的线程来执行任务。一个好的角色在不执行任务时是不会占住线程不放的，因为只有这样才能够让更多不同状态下的角色处于活动状态，并将有限的可用线程资源充分地利用起来。虽然在任意时刻都可能有多个角色处于活动状态，但在任何情况下一个角色中只有一个线程是活动的。这样既保证了多个角色之间的并发性，同时又消除了单个角色之内的竞争。

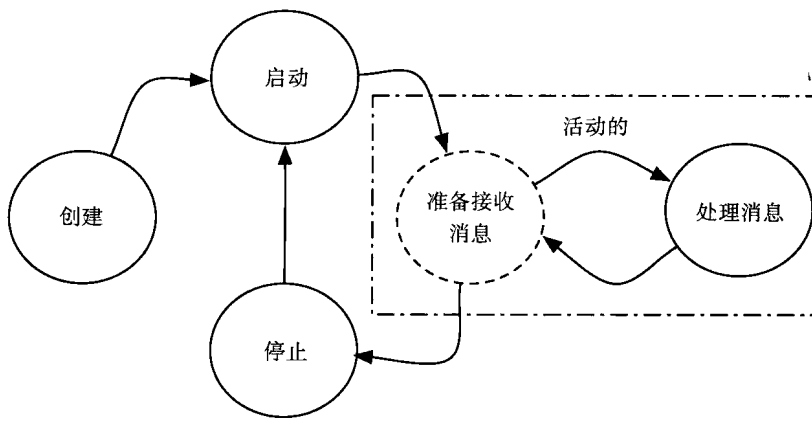


图 8-2　某个角色的生命周期

8.3　创建角色

正如前面曾经提到过的那样，虽然我们有很多支持角色的库可供选择，但是在本书中我们将使用 Akka。这是一个基于 Scala 的库，该库拥有非常好的性能和可扩展性，并同时支持角色和 STM。此外，该库还可以被用于多种 JVM 上的语言中。在本章中，我们将注意力集中在 Java 和 Scala 身上。而在下一章，我们将会学习如何在其他语言中使用 Akka 的角色。

由于 Akka 是用 Scala 实现的，所以在 Scala 中创建和使用角色非常简单并且更加自然，从 Akka API 的实现里也可以看到 Scala 简约而不简单的风格。除此之外，Akka 的开发者们还设计了一套相当出色的传统 Java API，可以使我们在 Java 代码中很方便地创建和使用角色。下面我们将先学习如何在 Java 中使用这套 API，然后再体验一下用 Scala 时将有着怎样的简化和改变。

8.3.1　用 Java 创建角色

在 Akka 中，抽象类 akka.actor.UntypedActor 用于表示一个角色，而具体的角色定义则只需简单继承这个抽象类并实现其 onReceive() 方法就可以了，每当有消息到达此角色时该方法将被调用。下面让我们通过一个简单的实例来对上述过程建立一个直观感受。下面我

们将会创建一个角色……不如就写一个可以对扮演不同荧幕人物（role）的请求进行响应的 HollywoodActor 怎么样？

favoringIsolatedMutability/java/create/HollywoodActor.java

```
public class HollywoodActor extends UntypedActor {
  public void onReceive(final Object role) {
    System.out.println("Playing " + role +
      " from Thread " + Thread.currentThread().getName());
  }
}
```

如上所示，onReceive() 函数接受一个 Object 对象作为其参数。在本例中，我们只是简单地将该参数以及负责处理消息的线程的详情打印出来。稍后我们将会学习如何处理不同类型的消息。

在完成了角色的定义之后，我们还需要创建一个角色的实例，并将该角色要演的荧幕人物以消息的形式发送给它，下面让我们来实现这部分内容：

favoringIsolatedMutability/java/create/UseHollywoodActor.java

```
public class UseHollywoodActor {
  public static void main(final String[] args) throws InterruptedException {
    final ActorRef johnnyDepp = Actors.actorOf(HollywoodActor.class).start();
    johnnyDepp.sendOneWay("Jack Sparrow");
    Thread.sleep(100);
    johnnyDepp.sendOneWay("Edward Scissorhands");
    Thread.sleep(100);
    johnnyDepp.sendOneWay("Willy Wonka");
    Actors.registry().shutdownAll();
  }
}
```

在 Java 中我们通常都是用 new 来创建对象的，但由于 Akka 的角色并非简单对象而是活动对象，所以我们需要用一个特殊函数 actorOf() 来完成创建动作。此外，我们还可以先用 new 生成一个实例，然后再调用 actorOf() 对该实例进行包装以获得一个角色的引用，关于这种创建方式我们稍后会再研究具体细节。当我们创建好了角色之后，就可以通过调用其 start() 函数来启动该角色。而当我们启动一个角色时，Akka 会将其写入一个注册表（registry）中，于是在这个角色停止运行之前我们都可以通过注册表来访问它。在本例中，johnnyDeep 即为角色实例的引用，其类型为 ActorRef。

接下来，我们通过 sendOneWay() 函数向 johnnyDeep 发送了一些附带着我们希望其扮演的荧幕人物（role）的消息。当消息发出之后，其实我们本不用加入那几个 100 毫秒等待时间的，但插入延时将有助于我们更好地学习角色如何进行线程切换的运作细节。在代码的结尾处，我们关闭了所有运行中的角色。除了代码示例中所使用的 shutdownAll() 之外，我们还可以逐个调用每个角色的 stop() 函数或给所有角色发送 kill 消息的方式来达到关停所有角色的目的。

为了能够运行上面的实例，我们需要先把 Akka 的库文件都添加到 classpath 中，然

后通过 javac 对代码进行编译。编译完成之后，我们就可以像运行其他常规 Java 程序一样运行本节的示例程序。需要再次提醒你的是，请务必记得将所有相关的 JAR 都添加到 classpath 中。下面就是我在我的系统上所使用的编译和运行指令：

```
javac -d . -classpath $AKKA_JARS HollywoodActor.java UseHollywoodActor.java
java -classpath $AKKA_JARS com.agiledeveloper.pcj.UseHollywoodActor
```

其中 AKKA_JARS 的定义如下所示：

```
export AKKA_JARS="$AKKA_HOME/lib/scala-library.jar:\
$AKKA_HOME/lib/akka/akka-stm-1.1.3.jar:\
$AKKA_HOME/lib/akka/akka-actor-1.1.3.jar:\
$AKKA_HOME/lib/akka/multiverse-alpha-0.6.2.jar:\
$AKKA_HOME/lib/akka/akka-typed-actor-1.1.3.jar:\
$AKKA_HOME/lib/akka/aspectwerkz-2.2.3.jar:\
$AKKA_HOME/config:\
."
```

为了使实例代码能顺利地编译运行，请根据你所使用的操作系统来定义 AKKA_JARS 环境变量，以便编译器能够正确定位到 Scala 和 Akka 的安装路径。我们既可以使用 Akka 自带的 Scala-library.jar，也可以使用 Scala 安装路径下的那一份。

默认情况下 Akka 会将额外的日志消息输出到标准输出，关于如何对这一行为进行配置请参阅 6.8 节。

下面让我们编译并运行示例代码，并观察角色对于消息的响应情况：

```
Playing Jack Sparrow from Thread akka:event-driven:dispatcher:global-1
Playing Edward Scissorhands from Thread akka:event-driven:dispatcher:global-2
Playing Willy Wonka from Thread akka:event-driven:dispatcher:global-3
```

通过输出结果我们可以看到，示例角色每次只响应一个消息，并且每次运行角色的线程都是不同的。对于消息处理的过程而言，既可以一个线程处理多个消息，也可以像本例这样由不同线程处理不同的消息，但无论是哪种处理模式，在任意时刻都只能有一个消息被处理。该模式的关键点在于，所有角色都是单线程的，但是在陷入等待状态时角色会优雅地将线程释放而不是抓住线程不撒手。我们在发送消息之后插入的 sleep 语句的目的就是为了将角色引入等待状态以便更清晰地演示这一运作细节。

上例中，我们创建角色时没有带任何构造函数参数。而如果需要的话，我们可以在角色的创建过程中引入一些参数。例如，我们可以用好莱坞演员的名字来初始化之前的 HollywoodActor：

favoringIsolatedMutability/java/params/HollywoodActor.java

```
public class HollywoodActor extends UntypedActor {
  private final String name;
  public HollywoodActor(final String theName) { name = theName; }

  public void onReceive(final Object role) {
    if(role instanceof String)
      System.out.println(String.format("%s playing %s", name, role));
    else
```

```
        System.out.println(name + " plays no " + role);
    }
}
```

新版的 HollywoodActor 类的构造函数定义了一个名为 name 的 String 类型参数。而在 onReceive() 函数中，我们对于不能识别的消息进行了专门的处理，即简单地在屏幕输出该好莱坞演员未曾饰演过那个未识别的消息所代表的荧幕人物（role）。当然我们也可以采取其他动作，比如返回一个错误码、记录日志、向上层调用者抛异常等。下面让我们看看如何给这个构造函数传递参数：

favoringIsolatedMutability/java/params/UseHollywoodActor.java

```
public class UseHollywoodActor {
  public static void main(final String[] args) throws InterruptedException {

    final ActorRef tomHanks = Actors.actorOf(new UntypedActorFactory() {
        public UntypedActor create() { return new HollywoodActor("Hanks"); }
      }).start();

    tomHanks.sendOneWay("James Lovell");
    tomHanks.sendOneWay(new StringBuilder("Politics"));
    tomHanks.sendOneWay("Forrest Gump");
    Thread.sleep(1000);
    tomHanks.stop();
  }
}
```

一般情况下，我们都是通过发送消息而不是直接调用函数的方式与角色进行交互的。Akka 不希望我们拿到角色的直接引用，而是希望我们只针对 ActorRef 的引用进行操作。这样一来，Akka 就可以确保我们不会往角色里添加其他函数，并且也不会与角色实例进行直接的交互。直接操纵角色实例的行为会将我们带回到共享可变性的泥淖中，而这正是我们极力想要避免的。此外，这种受控的角色创建方式也便于 Akka 更好地回收废弃的角色。所以如果我们试图直接创建一个角色类的实例，Akka 将抛出一个内容为“请不要用‘new’操作符显示地创建角色实例”的 akka.actor.ActorInitializationException 异常。

Akka 允许我们在 creat() 函数中以一种受控的方式创建角色实例，即我们可以在一个实现 UntypedActorFactory 接口的匿名类中实现 create() 函数，并在其 create() 函数中实现创建角色实例的逻辑，并发送正确的构造函数参数。而接下来的 actorOf() 则把一个继承自 UntypedActor 的普通对象转换为为一个 Akka 角色。随后，我们和之前一样向这个角色发送几条消息并观察输出结果。

在本例中，HollywoodActor 只接受 String 类型的消息，但我们在测试用例中向其发送了一条值为 Politics、类型为 StringBuilder 的消息。而我们在 onReceive() 函数中设计的检查逻辑将会发现并处理这一情况。最后，我们会调用 stop() 函数来终止角色的运行。代码结尾处插入 sleep(1000) 的目的是为了让角色在结束之前有机会响应所有未处理的消息。最终的输出结果如下所示：

```
Hanks playing James Lovell
Hanks plays no Politics
Hanks playing Forrest Gump
```

8.3.2　用 Scala 创建角色

在 Scala 中创建 Akka 角色时，我们没有像在 Java 版本中那样继承 UntypedActor 类，而是要继承 Actor trait 并实现 receive() 函数。下面让我们用 Scala 来实现之前刚刚用 Java 写过的 HollywoodActor 类：

favoringIsolatedMutability/scala/create/HollywoodActor.scala

```
class HollywoodActor extends Actor {
  def receive = {
    case role =>
    println("Playing " + role +
      " from Thread " + Thread.currentThread().getName())
  }
}
```

在上面的代码中，receive() 函数实现了一个 PartialFunction 并采用了 Scala 模式匹配的形式，但为了避免分散注意力我们现在先忽略这些细节。当有消息到达时，receive() 函数将被调用。如果对 Scala 语法还不熟悉的话，你可以暂时先把 receive() 函数想象成一个大的 switch 语句，其实现的功能与 Java 版本是完全相同的。

至此我们已经看到了如何定义一个角色，下面让我们把注意力集中到角色的使用上面：

favoringIsolatedMutability/scala/create/UseHollywoodActor.scala

```
object UseHollywoodActor {
  def main(args : Array[String]) :Unit = {
    val johnnyDepp = Actor.actorOf[HollywoodActor].start()

    johnnyDepp ! "Jack Sparrow"
    Thread.sleep(100)
    johnnyDepp ! "Edward Scissorhands"
    Thread.sleep(100)
    johnnyDepp ! "Willy Wonka"

    Actors.registry.shutdownAll
  }
}
```

Actor 类的 actorOf() 函数有多个重载定义，这里我们所采用的是接受一个角色类名（即代码中的 [HollywoodActor]）作为其参数的版本。在角色被创建出来之后，我们随即通过调用 start() 函数将其启动。在本例中，ActorRef 类型的变量 johnnyDepp 即为我们所创建的角色实例的引用。由于 Scala 可以进行类型推断，所以我们可以不必在代码中明确指定 johnnyDepp 的类型。

接下来，我们给 johnnyDepp 发送了 3 个附带着我们希望其扮演的荧幕人物的消息。噢，稍等一下，这里有一个细节请你注意，即我们是通过特殊函数“！”来发送消息的。当你见到 actor!message 时，请从右向左阅读这个语句，就能明白这条语句的意思是把消息

发送给指定的角色。这处细节再次展现了 Scala 在语法方面的简洁与优雅。通过这种方式，我们就无需再将发送消息的语句写成 actor.!(message)，而是简单地将句点和括号拿掉，简写成 actor!message 就行了。如果我们更喜欢 Java 里发送消息的那个函数，那么我们也可以把 Scala 简洁的语法用在 Java 风格的函数上，即把语句写成 actor sendOneWay message。上面示例中余下的代码与之前 Java 版本的示例完全相同，这里就不再赘述。

下面我们将通过 scalac 编译器对上述代码进行编译，但首先请务必记住要把 Akka 库文件添加到 classpath 中。编译完成后，我们就可以像之前运行普通 Java 程序那样运行上面的 scala 示例程序。需要再次提醒你的是，请务必记得将所需的 JAR 加入你系统的 classpath 中。下面是我在我的系统上所使用的编译和运行指令，请你根据你系统中 Scala 和 Akka 的安装目录来自行调整 classpath 中相关的路径信息：

```
scalac -classpath $AKKA_JARS HollywoodActor.scala UseHollywoodActor.scala
java -classpath $AKKA_JARS com.agiledeveloper.pcj.UseHollywoodActor
```

如果我们想要禁止日志消息输出到标准输出的话，请参阅 6.8 节中的相关内容。在将上述示例代码编译并运行之后，我们可以看到其输出结果与之前的 Java 版本是非常相似的：

```
laying Jack Sparrow from Thread akka:event-driven:dispatcher:global-1
laying Edward Scissorhands from Thread akka:event-driven:dispatcher:global-2
laying Willy Wonka from Thread akka:event-driven:dispatcher:global-3
```

如果想在创建角色时传些参数给它，如好莱坞演员的名字等，你会发现用 Scala 来实现会比之前的 Java 版本简单很多。下面让我们先对 HollywoodActor 类进行改造，以使其可以接受构造函数参数：

favoringIsolatedMutability/scala/params/HollywoodActor.scala

```
class HollywoodActor(val name : String) extends Actor {
  def receive = {
    case role : String => println(String.format("%s playing %s", name, role))
    case msg => println(name + " plays no " + msg)
  }
}
```

如上所示，新版本的 HollywoodActor 类接受一个名为 name 的 String 类型的构造函数参数。而在 receive() 函数中，我们对于格式无法识别的消息做了专门的处理。在 Scala 中我们无需再使用 instanceof，receive() 函数中的 case 语句即可实现消息与各种模式之间的匹配，在本例中特指消息类型的匹配。

我们用 Java 创建接受一个构造函数参数的角色时还是花了不少力气的，但在 Scala 中一切变得如此简单：

favoringIsolatedMutability/scala/params/UseHollywoodActor.scala

```
object UseHollywoodActor {
  def main(args : Array[String]) : Unit = {

    val tomHanks = Actor.actorOf(new HollywoodActor("Hanks")).start()
```

```
    tomHanks ! "James Lovell"
    tomHanks ! new StringBuilder("Politics")
    tomHanks ! "Forrest Gump"
    Thread.sleep(1000)
    tomHanks.stop()
  }
}
```

在上面的代码中，我们先用 new 关键字对角色进行初始化，随后又将初始化好的实例传给 actorOf() 函数（这是由于 Akka 禁止在 actorOf() 函数之外随意地创建 actor 实例）。通过这一动作，我们就将一个继承自 Actor 的普通对象转换成了一个 Akka 角色。接下来，我们同样会给新创建的角色发送 3 条消息。剩下的代码与 Java 版本非常相似，这里就不再赘述。最后让我们运行上述示例代码，并确认其输出与 Java 版本是否相同：

```
Hanks playing James Lovell
Hanks plays no Politics
Hanks playing Forrest Gump
```

8.4　收发消息

我们可以向角色发送任何类型的消息：String、Integer、Long、Double、List、Map、元组（tuple）、Scala 的 case 类等，但其中有一点需要注意的是，上述所有类型的消息都必须是不可变的。在上述这些类型中，我对于元组有着特殊的偏好，这并非因为我听到别人把元组误读成“two-ples”时感到很有趣，而是由于元组是轻量的、不可变的并且是最容易创建的实例之一。例如，在 Scala 中，我们可以简单地用（number1，number2）来创建一个含有两个数字的元组。除了元组之外，Scala 的 case 类也是用来定义消息的理想类型——因为 case 类是不可变的、可以进行模式匹配并且还很容易进行复制。在 Java 中，我们可以通过将消息定义为一个不可修改（unmodifiable）的 Collection 的方式来将多个对象塞到一个消息中。当我们向角色传递消息时，如果发送者和接收者都在同一个 JVM 里[⊖]，则默认情况下我们传递的是消息的引用。需要注意的是，保证所传递消息的不可变性是程序员自己的责任，尤其是当所发送的消息是我们自定义的类实例时则更需要加倍小心。为了解决这个问题，我们可以让 Akka 替我们先将消息序列化，然后将序列化出来的拷贝而不是原对象的引用发送出去，这样就可以避免由于类定义不严谨所造成的问题。

与角色交互最简单的方式莫过于“发送并忘记”（fire and forget），即先将消息发出去，然后不等响应继续做下面的事。这种做法从性能角度考虑也是最好的选择。其中，发送动作是非阻塞的，调用方角色 / 线程可以继续工作。我们可以使用 sendOneWay() 函数或 Scala 的“!”函数来发送一个单向消息。

除了“发送并忘记”交互模式之外，Akka 还提供了双向消息交互模式，以应对我们在发出消息之后需要等待对端角色响应的情况。在这种模式下，调用线程将被阻塞，直至收到对方响应或达到超时时间为止。下面让我们一起来看看如何在 Java 和 Scala 中收发消息。

⊖　Akka 同样支持远程角色，以便使我们可以在不同机器的离散进程之间发送消息。

8.4.1　在 Java 中收发消息

我们可以通过 sendRequestReply() 函数来发送消息并等待接收方响应。如果接收方的响应未在（可配置的）超时时间内到达，则系统将抛出 ActorTimeoutException 异常。下面让我们通过一个示例来学习这种双向消息通信方式：

favoringIsolatedMutability/java/2way/FortuneTeller.java

```
public class FortuneTeller extends UntypedActor {
  public void onReceive(final Object name) {
      getContext().replyUnsafe(String.format("%s you'll rock", name));
  }

  public static void main(final String[] args) {
    final ActorRef fortuneTeller =
      Actors.actorOf(FortuneTeller.class).start();
    try {
      final Object response = fortuneTeller.sendRequestReply("Joe");
      System.out.println(response);
    } catch(ActorTimeoutException ex) {
      System.out.println("Never got a response before timeout");
    } finally {
      fortuneTeller.stop();
    }
  }
}
```

在上面的代码中，我们定义了一个名为 FortuneTeller 的角色，它对收到的消息都会直接进行响应。为了响应发送方的消息，我们需要先调用 getContext() 函数获取调用上下文，然后再调用其 replyUnsafe() 函数来发送消息内容。调用 replyUnsafe() 函数向发送方发送响应的动作是非阻塞的，并且请注意，在发送响应消息的过程中我们没有调用任何与角色有关的代码。在 main() 函数中我们调用了 sendRequestReply() 函数，该函数会在内部创建一个 Future 类并等待对方响应或超时抛出异常。下面让我们通过运行上述代码来看看 Joe 的命运如何：

```
Joe you'll rock
```

我们上面实现的这个 FortuneTeller 实际上还有个问题没解决，即该角色依赖于消息发送方的发送方式。当消息发送方调用 sendRequestReply() 函数时，该函数会创建一个内部的 Future 用于等待对方响应。而如果我们换用 sendOneWay() 来发送消息的话，则 replyUnsafe() 函数将会失败。为了避免这种情况的发生，我们需要在调用 replyUnsafe() 函数之前先检查一下是否能匹配到一个处于阻塞状态的发送方。我们可以通过从上下文中读取发送方引用的方式来进行这个检查，也可以通过 replySafe() 函数的返回值来进行判断。因为当能取到发送方的引用时该函数会返回 true，反之则返回 false。下面我们就着手对 FortuneTeller 进行修改，以使其可以处理发送方没有阻塞地等待响应消息的情况：

favoringIsolatedMutability/java/2waysafe/FortuneTeller.java

```
public class FortuneTeller extends UntypedActor {
  public void onReceive(final Object name) {
    if(getContext().replySafe(String.format("%s you'll rock", name)))
      System.out.println("Message sent for " + name);
    else
      System.out.println("Sender not found for " + name);
  }

  public static void main(final String[] args) {
    final ActorRef fortuneTeller =
      Actors.actorOf(FortuneTeller.class).start();

    try {
      fortuneTeller.sendOneWay("Bill");
      final Object response = fortuneTeller.sendRequestReply("Joe");
      System.out.println(response);
    } catch(ActorTimeoutException ex) {
      System.out.println("Never got a response before timeout");
    } finally {
      fortuneTeller.stop();
    }
  }
}
```

如上所示，新版的 FortuneTeller 代码很优雅地处理了我们之前提到的那些问题，即使接收方没找到发送方也不会导致处理失败。

```
Sender not found for Bill
Message sent for Joe
Joe you'll rock
```

我们知道，sendRequestReply() 函数是需要等待对方响应的阻塞式函数，而 sendOneWay() 函数则是单向且非阻塞的。而如果既想要接收响应又不想被阻塞，则可以使用更复杂一些的 sendRequestReplyFuture() 函数。该函数可以返回一个 Future 对象，而拿到 Future 对象之后我们就可以继续干其他的事，直到我们真正需要用到对方的响应时，再选择阻塞式地等待或通过之前拿到的那个 Future 对象来查询对方的响应是否已经可用。类似地，在角色这一侧我们可以从上下文引用中取到 senderFuture，并通过它来立即与发送方进行通信，或在响应可用时再与对方通信。在后面的示例中，我们会看到上述这些函数的具体用法，这里就不再赘述了。

请务必谨慎使用 sendRequestReply() 和 sendRequestReplyFuture() 函数，因为这两个函数都是阻塞的，所以调用它们对程序的性能和可扩展性都会造成负面影响。

8.4.2 在 Scala 中收发消息

如果想要在 Scala 中与角色进行消息收发，我们需要有些心理准备，因为在 Scala 中我们所采用的方法将会与 Java API 有不小的差别：

- 在 Scala 中，我们可以直接使用 self 属性来访问角色。通过该属性，我们可以调用

reply() 函数或 replySafe() 函数，其中 reply() 就是 replyUnsafe() 在 Scala 侧的等价函数。

- 在 Scala 中，我们既可以调用 sendRequestReply() 函数，也可以调用更优雅的 !! 函数，当然美是仁者见仁智者见智的。同样，也可以用来替换 sendRequestReplyFuture() 函数。
- 在 Scala 中，sendRequestReply() 函数不再返回一个 Object，而是返回一个 Scala 的 Option。当接收方的响应抵达时，这个 Option 将是一个 Some[T] 的实例。该实例中存有响应的具体内容，而在超时的情况下则响应内容为 None。所以，与 Java 版本所不同的是，在 Scala 中调用 sendRequestReply() 函数在超时的时候不会抛异常。

下面让我们先用不安全的 reply() 函数实现 Scala 版的 FortuneTeller：

favoringIsolatedMutability/scala/2way/FortuneTeller.scala

```
class FortuneTeller extends Actor {
  def receive = {
    case name : String =>
      self.reply(String.format("%s you'll rock", name))
  }
}
object FortuneTeller {
  def main(args : Array[String]) : Unit = {
    val fortuneTeller = Actor.actorOf[FortuneTeller].start()

    val response = fortuneTeller !! "Joe"
    response match {
      case Some(responseMessage) => println(responseMessage)
      case None => println("Never got a response before timeout")
    }

    fortuneTeller.stop()
  }
}
```

在角色的实现代码中，我们可以看到与 Java 版本的两点不同：其一是这里中我们用 self 代替了 getContext() 函数，另一个则是用 reply() 代替了 replyUnsafe() 函数。在调用方这一侧，我们使用了 !!，即 java 中的 sendRequestReply() 函数来给角色发送消息，并在所收到的响应内容上应用了模式匹配。如果发送方收到对端的响应，则第一个 case 语句将被执行，而如果响应超时则第二个 case 语句将被执行。不出所料，该示例代码的运行结果与 Java 版完全相同：

```
Joe you'll rock
```

除了我们之前所讨论过的那些变更，安全版 reply() 函数的使用方式与 Java 版本差别不大。在 Scala 中，我们使用的是 reply_?() 或 replySafe()。

favoringIsolatedMutability/scala/2waysafe/FortuneTeller.scala

```
class FortuneTeller extends Actor {
  def receive = {
    case name : String =>
      if(self.reply_?(String.format("%s you'll rock", name)))
```

```
        println("Message sent for " + name)
      else
        println("Sender not found for " + name)
  }
}
object FortuneTeller {
  def main(args : Array[String]) : Unit = {
    val fortuneTeller = Actor.actorOf[FortuneTeller].start()
    fortuneTeller ! "Bill"
    val response = fortuneTeller !! "Joe"
    response match {
      case Some(responseMessage) => println(responseMessage)
      case None => println("Never got a response before timeout")
    }
    fortuneTeller.stop()
  }
}
```

通过上述修改，即使是在发送者未知的情况下，新版本的 FortureTeller 也不会失败：

```
Sender not found for Bill
Message sent for Joe
Joe you'll rock
```

Akka 有一点很方便的行为就是，当我们用 Akka 发送消息时，它会将发送方的引用透明地传递过去。于是我们就无需显式地将发送方作为消息的一部分传递出去，从而省去了很多繁冗的代码。

如果不习惯使用像 !、!!、!!! 以及 reply_? 这样的函数名，我们也可以分别用 sendOneWay()、sendRequestReply()、sendRequestReplyFuture() 以及 replySafe() 这些函数来替换使用。

8.5　同时使用多个角色

通过前面的学习，我们已经了解了如何创建角色以及如何给角色发送消息，下面让我们来一起学习如何让多个角色协同工作。在第 2 章中，我们创建了一个统计给定区间内所有素数的并发程序。在该程序中，我们使用了 ExecutorService、Callable、Future 以及其他差不多超过一页纸那么多代码。本节我们将会学习如何用 Akka 角色对该示例进行重构，并且根据之前的惯例我们的介绍顺序还是先 Java 后 Scala。

在 Java 中同时使用多个角色

假定待统计数字集合中的数字是 1 千万个，为了统计其中的素数数量，之前我们是将数字集合划分为若干个不相交的子集合，并将这些子集合丢给一些线程去执行统计操作。但这里我们将使用角色来完成同样的功能，下面就让我们从角色的 onRecevie() 函数开始说起吧：

favoringIsolatedMutability/java/primes/Primes.java

```
public class Primes extends UntypedActor {
  public void onReceive(final Object boundsList) {
```

```
    final List<Integer> bounds = (List<Integer>) boundsList;
    final int count =
      PrimeFinder.countPrimesInRange(bounds.get(0), bounds.get(1));
    getContext().replySafe(count);
  }
```

为了统计给定区间内的素数数量，我们需要指定区间的上下限。在本例中，onReceive() 函数的消息参数是一个 List，其中前两个元素即为区间的上下限。在 onReceive() 函数内部，我们调用了 PrimeFinder 类的 countPrimesInRage() 函数来统计区间内的素数数量，最后又使用 replySafe() 函数将统计结果返回给调用方。

在给定了待统计的数字集合之后，我们需要将其划分成若干个不相交的子集合并将统计这些子集合中素数数量的任务委托给各个不同的角色来执行。下面就让我们在静态方法 countPrimes() 中实现这些逻辑：

favoringIsolatedMutability/java/primes/Primes.java

```
public static int countPrimes(
  final int number, final int numberOfParts) {
  final int chunksPerPartition = number / numberOfParts;
  final List<Future<?>> results = new ArrayList<Future<?>>();
  for(int index = 0; index < numberOfParts; index++) {
    final int lower = index * chunksPerPartition + 1;
    final int upper = (index == numberOfParts - 1) ? number :
        lower + chunksPerPartition - 1;
    final List<Integer> bounds = Collections.unmodifiableList(
      Arrays.asList(lower, upper));
    final ActorRef primeFinder = Actors.actorOf(Primes.class).start();
    results.add(primeFinder.sendRequestReplyFuture(bounds));
  }

  int count = 0;
  for(Future<?> result : results)
    count += (Integer)(result.await().result().get());

  Actors.registry().shutdownAll();
  return count;
}
```

在确定了每个子集合的范围之后，我们会将其包装在一个不可变集合里，请记住，所有的消息都必须是不可变的。接下来，我们调用 sendRequestReplyFuture() 这个非阻塞函数来将统计请求发送给各个角色进行处理。在把请求发送出去之后，我们将 sendRequestReplyFuture() 返回的 Future 对象（注意这里是 akka.dispatch.Future 而不是 JDK 中的 java.util.concurrent.Future）保存在一个数组中以便稍后从其中取回各个子集合的统计结果。在任务分派完毕之后，我们就可以循环查询每个 Future，即先调用 Future 的 await() 函数，待 await() 函数返回之后再调用其返回实例的 result() 函数来获取一个 Scala 的 Option 实例，你可以将其假想为一个包含统计结果的数据单元（如果数据存在的话）。最后我们可以通过调用该实例对象的 get() 函数来得到一个 Integer 类型的统计值。

OK，下面就让我们写一个用来检验上述代码的测试用例，其中的待统计数字和子集合划分数是通过命令行传给程序的：

favoringIsolatedMutability/java/primes/Primes.java

```
public static void main(final String[] args) {
  if (args.length < 2)
    System.out.println("Usage: number numberOfParts");
  else {
    final long start = System.nanoTime();
    final int count = countPrimes(
      Integer.parseInt(args[0]), Integer.parseInt(args[1]));
      final long end = System.nanoTime();
      System.out.println("Number of primes is " + count);
      System.out.println("Time taken " + (end - start)/1.0e9);
    }
  }
}
```

main() 函数主要负责对上面的统计代码进行测试并记录执行耗时。最后我们还需要实现 PrimeFinder 这个真正负责统计工作的类：

favoringIsolatedMutability/java/primes/PrimeFinder.java

```
public class PrimeFinder {
  public static boolean isPrime(final int number) {
    if (number <= 1) return false;
    final int limit = (int) Math.sqrt(number);
    for(int i = 2; i <= limit; i++) if(number % i == 0) return false;
    return true;
  }
  public static int countPrimesInRange(final int lower, final int upper) {
    int count = 0;
    for(int index = lower; index <= upper; index++)
      if(isPrime(index)) count += 1;
    return count;
  }
}
```

令待统计区间为 [1, 1 千万]、划分的子区间为 100 个，则上述示例程序的输出结果如下所示：

```
Number of primes is 664579
Time taken 3.890996
```

下面让我们将本节的代码和输出结果与第 2.4 节的示例代码和输出结果进行比较。虽然两个版本都将子集合数设为 100，但 Akka 版本的示例代码无需显式设定线程池大小。此外，由于这是一个计算密集型问题，所以对于使用 ExecutorService 的版本而言，其线程池大小的设定是需要随机器 CPU 核心数计算而定的，所以两个版本的性能都差不多，而 Akka 版本在代码的形式上要比使用 ExecutorServer 的版本简洁一些。但正如我们在本章后面将会看到的那样，当我们需要让多个线程 / 角色相互协作的时候，这些区别将会愈发明显。

8.5.2　在 Scala 中同时使用多角色

如果用 Scala 来实现这个统计素数数量的程序，那么我们就可以深切体会到 Scala 在角

色的实现以及与角色交互方面的简洁和优雅。下面让我们来看看 Scala 版本的 Primes 类是如何实现的：

favoringIsolatedMutability/scala/primes/Primes.scala

```
class Primes extends Actor {
  def receive = {
    case (lower : Int, upper : Int) =>
      val count = PrimeFinder.countPrimesInRange(lower, upper)
      self.replySafe(new Integer(count))
  }
}

object Primes {
  def countPrimes(number : Int, numberOfParts : Int) = {
    val chunksPerPartition : Int = number / numberOfParts

    val results = new Array[Future[Integer]](numberOfParts)
    var index = 0

    while(index < numberOfParts) {
      val lower = index * chunksPerPartition + 1
      val upper = if (index == numberOfParts - 1)
        number else lower + chunksPerPartition - 1
      val bounds = (lower, upper)
      val primeFinder = Actor.actorOf[Primes].start()
      results(index) = (primeFinder !!! bounds).asInstanceOf[Future[Integer]]
      index += 1
    }

    var count = 0
    index = 0
    while(index < numberOfParts) {
      count += results(index).await.result.get.intValue()
      index += 1
    }
    Actors.registry.shutdownAll
    count
  }

  def main(args : Array[String]) : Unit = {
    if (args.length < 2)
      println("Usage: number numberOfParts")
    else {
      val start = System.nanoTime
      val count = countPrimes(args(0).toInt, args(1).toInt)
      val end = System.nanoTime
      println("Number of primes is " + count)
      println("Time taken " + (end - start)/1.0e9)
    }
  }
}
```

Scala 版本的代码与 Java 版本有几点不同。首先，Scala 版本所使用的消息格式是简单的元组而不是一个不可变列表。其次，receive() 函数中的 case 语句与应用场景十分契合。

第三，Java 版本中 countPrimes() 函数里的 for 循环在这里变成了一个 while 循环。其原因是，虽然 Scala 的 for 循环表达式十分优雅，但会增加 Object 到基本类型之间的转换开销。为了能够得到比较真实的性能对比，我在这里放弃了优雅。

类似，在 PrimeFinder 中，我们也用 while 循环代替了 for 循环。

favoringIsolatedMutability/scala/primes/PrimeFinder.scala

```
object PrimeFinder {
  def isPrime(number : Int) : Boolean = {
    if (number <= 1) return false

    var limit = scala.math.sqrt(number).toInt
    var i = 2
    while(i <= limit) {
      if(number % i == 0) return false
      i += 1
    }
    return true
  }

  def countPrimesInRange(lower : Int, upper : Int) : Int = {
    var count = 0
    var index = lower
    while(index <= upper) {
      if(isPrime(index)) count += 1
      index += 1
    }
    count
  }
}
```

令待统计区间为 [1, 1000 万]、划分的子区间为 100 个，则 Scala 版示例程序的性能如下所示：

```
Number of primes is 664579
Time taken 3.88375
```

8.6　多角色协作

在使用基于角色的编程模型时，只有当多个角色互相协作、同心协力解决问题时，我们才能真正从中获益并感受到其中的乐趣。为了更好地利用并发的威力，我们通常需要把问题拆分成若干个子问题。不同的角色可以负责不同的子问题，而我们则需要对角色之间的通信进行协调。下面我们将通过重写计算目录大小的例子来学习如何在进行多角色协作。

在 4.2 节中，我们写了一个计算给定目录下所有文件大小的程序。在那个例子中，我们启动了 100 个线程，每个线程都负责扫描不同的子目录，并在最后异步地将所有计算结果累加在一起。而本节中我们将看到一些通过 AtomicLong 和队列来实现上述功能的方法，同时我们还会把这些方法归纳起来以备你以后处理共享可变性问题之用。

使用带有角色的隔离可变性方法可以使我们在解决并发问题时省去不少麻烦。与本书

第 4 章所介绍的处理共享可变性的方案相比，我们在本节所介绍的方法可以无需为换取线程安全而牺牲性能。此外，正如我们即将看到的那样，作为使用角色编程模型的专属福利，我们可以在角色内部以同步无关的方式写代码，而这也将使得代码更加简洁明了。

首先，我们需要为计算目录大小问题创建一个多角色协作的设计方案，我们可以使用图 8-3 中定义的两类角色。我们将会把可变状态都隔离保存到名为 SizeCollector 的角色中。该角色通过接收消息来记录需要被扫描的目录、保存目录大小的当前值以及给 FileProcessor 角色提供需要进行扫描的目录。而主程序代码则负责让所有角色跑起来。此外，我们还将创建 100 个 FileProcessor 角色用于遍历给定目录下的文件 / 目录。

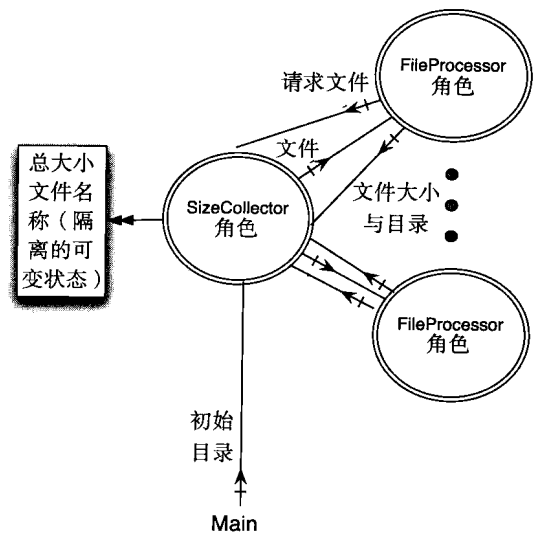


图 8-3　使用角色解决计算目录大小问题的设计方案

下面我们还是用 Akka 先在 Java 中实现这一设计，随后再用 Scala 进行重写。

8.6.1　在 Java 中进行多角色协作

下面让我们先定义一下 SizeCollector 类所要接收的消息：

favoringIsolatedMutability/java/filesize/ConcurrentFileSizeWAkka.java

```
class RequestAFile {}

class FileSize {
  public final long size;
  public FileSize(final long fileSize) { size = fileSize; }
}

class FileToProcess {
  public final String fileName;
  public FileToProcess(final String name) { fileName = name; }
}
```

上面这两种消息都是由不可变类定义的。每个 FileProcessor 都将通过 RequestAFile 类型的消息来将自己加入到 SizeCollector 的等待队列当中。FileSize 消息内携带有 FileProcessor 所统计的子目录大小的数据，并由 FileProcessor 在统计结束之后返回给 SizeCollector。最后，FileToProcess 消息持有需要进行遍历的子目录名称，该消息是由 SizeCollector 作为对 RequestAFile 请求的应答返回给 FileProcessor 的。

FileProcessor 主要负责遍历给定目录并返回该目录下所有文件的大小以及其下所有子目录的名称。在完成当前任务之后，FileProcessor 会发送 RequestAFile 类给 SizeCollector，以便使其知道他们已经做好接下一单目录遍历任务的准备了。此外，这些 FileProcessor 还需要事先向 SizeCollector 注册，以便可以收到第一个目录遍历的任务。此外，由于 preStart() 函数会在每次角色启动之前被调用，所以最适合做这件事的地方莫过于该函数了。下面就让我们一起来实现 FileProcessor，请千万别忘了要令其构造函数参数接受一个指向 SizeCollector 的引用。

favoringIsolatedMutability/java/filesize/ConcurrentFileSizeWAkka.java

```
class FileProcessor extends UntypedActor {
  private final ActorRef sizeCollector;

  public FileProcessor(final ActorRef theSizeCollector) {
    sizeCollector = theSizeCollector;
  }

  @Override public void preStart() { registerToGetFile(); }

  public void registerToGetFile() {
    sizeCollector.sendOneWay(new RequestAFile(), getContext());
  }

  public void onReceive(final Object message) {
    FileToProcess fileToProcess = (FileToProcess) message;
    final File file = new java.io.File(fileToProcess.fileName);
    long size = 0L;
    if(file.isFile()) {
      size = file.length();
    } else {
      File[] children = file.listFiles();
      if (children != null)
        for(File child : children)
          if (child.isFile())
            size += child.length();
          else
            sizeCollector.sendOneWay(new FileToProcess(child.getPath()));
    }

    sizeCollector.sendOneWay(new FileSize(size));
    registerToGetFile();
  }
}
```

在 registerToGetFile() 函数中，FileProcessor 会向 SizeCollector 发送一个 RequestAFile

消息，同时 getContext() 函数则会把一个指向自身的引用也一并发送给 SizeCollector 角色的实例。SizeCollector 将会把这个引用加入到空闲可用 FileProcessor 队列中以备需要执行遍历目录任务时使用。

我们稍后将会看到，SizeCollector 类将通过发送一个 FileToProcess 消息来指令 FileProcessor 去遍历指定目录。而 FileProcessor 的 onReceive() 函数将负责响应这个消息。在 onReceive() 函数中，我们将给定目录下的所有子目录通过 sendOneWay() 函数发给 SizeCollector。而对于给定目录下的文件，我们会将其大小累加起来并在任务结束前将其发送个给 SizeCollector。在任务的结尾，我们需要再次将 FileProcessor 类注册给 SizeCollector 以便 SizeCollector 可以继续为其分配目录遍历任务。

至此，FileProcessor 已经基本完成了，可以遍历目录了，下面让我们讨论一下 SizeCollector 相关的问题。SizeCollector 是负责管理隔离可变状态的决策者，它可以让 FileProcessor 们始终保持满负荷工作状态直至得到最终的统计结果为止。在与其他角色的交互方面，它主要负责处理我们之前曾讨论过的那三类消息。下面让我们先看一下 SizeCollector 的实现代码，然后逐一讨论 SizeCollector 针对这三种消息的具体动作：

favoringIsolatedMutability/java/filesize/ConcurrentFileSizeWAkka.java

```
class SizeCollector extends UntypedActor {
  private final List<String> toProcessFileNames = new ArrayList<String>();
  private final List<ActorRef> idleFileProcessors =
    new ArrayList<ActorRef>();
  private long pendingNumberOfFilesToVisit = 0L;
  private long totalSize = 0L;
  private long start = System.nanoTime();

  public void sendAFileToProcess() {
    if(!toProcessFileNames.isEmpty() && !idleFileProcessors.isEmpty())
      idleFileProcessors.remove(0).sendOneWay(
        new FileToProcess(toProcessFileNames.remove(0)));
  }

  public void onReceive(final Object message) {
    if (message instanceof RequestAFile) {
      idleFileProcessors.add(getContext().getSender().get());
      sendAFileToProcess();
    }

    if (message instanceof FileToProcess) {
      toProcessFileNames.add(((FileToProcess)(message)).fileName);
      pendingNumberOfFilesToVisit += 1;
      sendAFileToProcess();
    }

    if (message instanceof FileSize) {
      totalSize += ((FileSize)(message)).size;
      pendingNumberOfFilesToVisit -= 1;

      if(pendingNumberOfFilesToVisit == 0) {
        long end = System.nanoTime();
```

```
        System.out.println("Total size is " + totalSize);
        System.out.println("Time taken is " + (end - start)/1.0e9);
        Actors.registry().shutdownAll();
      }
    }
  }
}
```

SizeCollector 维护了两个链表，其中一个负责保存待访问目录，而另一个负责保存空闲的 FileProcessor。除此之外，SizeCollector 还定义了 3 个 long 型变量用于记录当前还有多少个待访问的目录、当前已统计的文件大小总数以及统计动作开始的时间戳。

sendAFileToProcess() 函数主要用于为空闲的 FileProcessor 们分配待访问目录。

从代码上看，SizeCollector 可以在 onReceive() 消息处理函数中接受三种类型的消息，而每种消息都有其各自的目的。

当 FileProcessor 干完手头的活之后，它们会立即向 SizeCollector 发送一个 RequestAFile 消息，而 SizeCollector 则将这些空闲角色的引用保存在其空闲 FileProcessor 列表中。

FileToProcess 是 SizeCollector 既收且发的一类消息。当需要挑选空闲 FileProcessor 执行统计任务时，SizeCollector 就会在 sendAFileToProcess() 函数中发出一条这类消息给一个空闲的 FileProcessor。而当在遍历过程中发现有子目录时，FileProcessor 就会用这类消息把所发现的目录告知 SizeCollector，以便 SizeCollector 可以调度其他 FileProcessor 来执行该目录的遍历。

最后我们要介绍的由 SizeCollector 处理的一类消息是 FileSize，该消息是由 FileProcessor 发出的，并且其中承载了由 FileProcessor 所统计出来的给定目录下的文件大小。

每当收到一个待访问目录名的时候，SizeCollector 都会将名为 pendingNumberOfFilesToVisit 的隔离可变计数器加 1。而每当收到一个带有某目录文件大小的 FileSize 消息时，SizeCollector 就会将该值计数器减 1。一旦发现这个计数器的值变为 0，则 SizeCollector 会输出当前统计到的目录大小和所有操作的总耗时，并关闭所有活动的角色，即结束整个程序的运行。

下面让我们实现总体设计的最后一个板块，即主程序代码：

favoringIsolatedMutability/java/filesize/ConcurrentFileSizeWAkka.java

```
public class ConcurrentFileSizeWAkka {
  public static void main(final String[] args) {
    final ActorRef sizeCollector =
      Actors.actorOf(SizeCollector.class).start();

    sizeCollector.sendOneWay(new FileToProcess(args[0]));

    for(int i = 0; i < 100; i++)
      Actors.actorOf(new UntypedActorFactory() {
        public UntypedActor create() {
          return new FileProcessor(sizeCollector);
        }
```

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

在主函数代码中，我们首先创建了一个 SizeCollector 的实例，并通过一个 FileToProcess 消息告诉它所要进行统计的是哪个目录。随后我们创建了一个 FileProcessor 角色，并由 SizeCollector 负责协调这些 FileProcessor 共同来完成统计任务。

下面让我们通过上述示例程序来统计一下 /usr 目录下的文件大小：

```
Total size is 3793911517
Time taken is 8.599308
```

通过将使用隔离可变性的本例与 4.2 节中使用共享可变性完成相同功能的代码进行比较后我们可以发现，所有这些示例所输出的 /usr 目录下的文件大小都是相同的，并且彼此之间的性能也相差无几。但基于角色的版本与其他实现版本最大的区别是其实现中不含任何同步相关代码、也没有线程闩、队列以及 AtomicLong 这些会惹麻烦的东西。而这一区别所造成的结果是：在保证了性能的基础上，我们既能保持代码逻辑简洁，同时又免去了同步和加锁等烦恼。

8.6.2　在 Scala 中进行多角色协作

在上面的内容里，我们用 Akka 角色在 Java 中实现了统计指定目录下文件大小的程序。我们同样也可以在 Scala 中实现相同的设计，并且还将比 Java 版本代码写得更简洁。但 Scala 的实现版本与 Java 版本有一个显著区别在于消息的处理，Scala 有 case 类，而该类可以为创建不可变类型提供表达力极强的语法。正是考虑到这种 case 表达式非常适用于区分消息类型，所以下面我们将会使用 case 类来实现消息的分类处理：

favoringIsolatedMutability/scala/filesize/ConcurrentFileSizeWAkka.scala

```
case object RequestAFile
case class FileSize(size : Long)
case class FileToProcess(fileName : String)
```

Scala 版的 FileProcessor 是从 Java 到 Scala 的直接翻译，所以这里面没什么超出我们之前讨论范围的新东西：

favoringIsolatedMutability/scala/filesize/ConcurrentFileSizeWAkka.scala

```
class FileProcessor(val sizeCollector : ActorRef) extends Actor {
  override def preStart = registerToGetFile

  def registerToGetFile = { sizeCollector ! RequestAFile }

  def receive = {
    case FileToProcess(fileName) =>
      val file = new java.io.File(fileName)

      var size = 0L
      if(file.isFile()) {
```

```
        size = file.length()
      } else {
        val children = file.listFiles()
        if (children != null)
          for(child <- children)
            if (child.isFile())
              size += child.length()
            else
              sizeCollector ! FileToProcess(child.getPath())
      }

      sizeCollector ! FileSize(size)
      registerToGetFile
    }
  }
```

下面让我们将 SizeCollector 角色也翻译成 Scala 版。由于我们采用了 case 类作为消息类型，所以 Scala 的模式匹配用在这里简直是天作之合。此外，这种模式也有助于我们更方便地从合适的消息中提取像文件名和文件大小这样数据值。

favoringIsolatedMutability/scala/filesize/ConcurrentFileSizeWAkka.scala

```
class SizeCollector extends Actor {
  var toProcessFileNames = List.empty[String]
  var fileProcessors = List.empty[ActorRef]
  var pendingNumberOfFilesToVisit = 0L
  var totalSize = 0L
  val start = System.nanoTime()

  def sendAFileToProcess() : Unit = {
    if(!toProcessFileNames.isEmpty && !fileProcessors.isEmpty) {
      fileProcessors.head ! FileToProcess(toProcessFileNames.head)
      fileProcessors = fileProcessors.tail
      toProcessFileNames = toProcessFileNames.tail
    }
  }
  def receive = {
    case RequestAFile =>
      fileProcessors = self.getSender().get :: fileProcessors
      sendAFileToProcess()
    case FileToProcess(fileName) =>
      toProcessFileNames = fileName :: toProcessFileNames
      pendingNumberOfFilesToVisit += 1
      sendAFileToProcess()
    case FileSize(size) =>
      totalSize += size
      pendingNumberOfFilesToVisit -= 1
      if(pendingNumberOfFilesToVisit == 0) {
        val end = System.nanoTime()
        println("Total size is " + totalSize)
        println("Time taken is " + (end - start)/1.0e9)
        Actors.registry.shutdownAll
      }
  }
}
```

最后，我们还需要将主函数代码从 Java 翻译成 Scala，而这一次仍然是一个直译的过程。

favoringIsolatedMutability/scala/filesize/ConcurrentFileSizeWAkka.scala

```
object ConcurrentFileSizeWAkka {
  def main(args : Array[String]) : Unit = {
    val sizeCollector = Actor.actorOf[SizeCollector].start()

    sizeCollector ! FileToProcess(args(0))

    for(i <- 1 to 100)
      Actor.actorOf(new FileProcessor(sizeCollector)).start()
  }
}
```

下面让我们运行 Scala 版的示例程序统计 /usr 目录的大小，看看在程序性能和文件大小的统计结果与 Java 版相比是否一致，如下所示：

```
Total size is 3793911517
Time taken is 8.321386
```

8.7　使用类型化角色

到目前为止我们所接触过的角色都是可以接收消息的，而消息的类型也是五花八门，如 String、元组、case 类 / 自定义消息等。然而发送消息的行为在感觉上与我们日常编程工作中所使用的常规函数调用还是有很大区别的，为了弥合二者之间的鸿沟，类型化角色（Typed Actor）就应运而生了。这种类型的角色可以将发送消息的动作在形式上伪装成常规的函数调用，而将消息传输动作隐藏在后台执行。我们可以将类型化角色想象成一个活动的对象，该对象运行在一个属于自己的轻量消息驱动的线程里面，并且还带有一个用于将正常的函数调用转换成异步非阻塞消息的拦截代理。

由于类型化角色可以将普通函数调用在后台转换成消息，所以我们就可以最大程度地享受到静态类型所带来的好处。由于有了类型化角色，我们就无需在接收消息的角色里对收到的消息类型猜来猜去，同时也可以更好地利用如代码补全等这些由 IDE 所提供的编程支持。

如果要实现一个普通的角色，我只需简单地写一个类，并使其继承 UntypedActor 或 Actor trait/ 抽象类即可。而如果我们要实现一个类型化的角色，则需要创建一个接口 - 实现对（在 Scala 中，我们可以不用写接口定义，而只需使用一个不含实现内容的 trait 即可）。

我们可以使用 Actor 类的 actorOf() 函数来实例化一个普通角色。而如果要实例化一个类型化的角色，则我们需要使用 TypedActor 的 newInstance() 函数才行。

我们从 TypedActor 中拿到的引用是一个可以将函数调用转换成异步消息的拦截代理。其中，返回值为 void 的函数将被转换成 sendOneWay 或 ! 函数，返回普通类型内容的函数会被转换成 sendRequestReply() 或 !! 函数，而返回值类型为 Future 的函数则被转换成 sendRequestReplyFuture() 或 !!! 函数。

我们在第 5 章中用现代 Java 并发 API 和在 6.7 节中用 STM 重构的 EnergySource 类示例是我们演示类型化角色功能和用法的最佳选择。由于该示例中存在可变状态，所以我们可以用一个角色来对其进行隔离。又因为每个 EnergySource 实例都将各自运行在一个独立的线程中，所以其中不存在竞争条件的问题。当多个线程调用同一个 EnergySource 实例上的方法时，这些调用将会跳出线程并在该实例上顺序执行。请记住，角色是不会一直占住线程不放的，所以它们可以在各实例间共享线程资源并提供更大的吞吐量，类型化的角色同样也可以做到。

EnergySource 其实本身没什么复杂的业务逻辑，只有查询当前的电量、使用次数和消耗电量，以及可以在后台自动恢复电量这几个简单的功能。我们当然希望基于角色的版本也能实现上述全部功能，但请先不要着急，我们将采取递增式的构建方法以便可以每次只关注一件事情。示例展示顺序则还是沿用之前的老规矩：先构建 Java 版然后再构建 Scala 版。

8.7.1　在 Java 中使用类型化角色

类型化角色需要一个接口 / 实现对，所以下面让我们先从 EnergySource 的接口开始：

favoringIsolatedMutability/java/typed1/EnergySource.java

```
public interface EnergySource {
  long getUnitsAvailable();
  long getUsageCount();
  void useEnergy(final long units);
}
```

与上面这个接口相对应的实现类是 EnergySourceImpl。该类与普通 Java 类的唯一区别是我们为了将其转换成一个活动对象而让其继承了 TypedActor 类：

favoringIsolatedMutability/java/typed1/EnergySourceImpl.java

```
public class EnergySourceImpl extends TypedActor implements EnergySource {
  private final long MAXLEVEL = 100L;
  private long level = MAXLEVEL;
  private long usageCount = 0L;

  public long getUnitsAvailable() { return level; }
  public long getUsageCount() { return usageCount; }

  public void useEnergy(final long units) {
    if (units > 0 && level - units >= 0) {
      System.out.println(
        "Thread in useEnergy: " + Thread.currentThread().getName());
      level -= units;
      usageCount++;
    }
  }
}
```

TypedActor 保证 EnergySourceImpl 里定义的所有函数都是互斥的；也就是说，在任意给定的实例上，每次只能有一个函数能被调用，所以在函数的实现中无需对任何成员字

段的访问进行同步或加锁。为了能够感知到执行角色的线程的存在，我们会在示例代码中插入少量的打印语句。最后，为了验证类型化角色的实际效果，我们还需要实现测试用例 UseEnergySouce 类。

favoringIsolatedMutability/java/typed1/UseEnergySource.java

```
public class UseEnergySource {
  public static void main(final String[] args)
    throws InterruptedException {
    System.out.println("Thread in main: " +
      Thread.currentThread().getName());

    final EnergySource energySource =
      TypedActor.newInstance(EnergySource.class, EnergySourceImpl.class);

    System.out.println("Energy units " + energySource.getUnitsAvailable());

    System.out.println("Firing two requests for use energy");
    energySource.useEnergy(10);
    energySource.useEnergy(10);
    System.out.println("Fired two requests for use energy");
    Thread.sleep(100);
    System.out.println("Firing one more requests for use energy");
    energySource.useEnergy(10);

    Thread.sleep(1000);
    System.out.println("Energy units " + energySource.getUnitsAvailable());
    System.out.println("Usage " + energySource.getUsageCount());

    TypedActor.stop(energySource);
  }
}
```

首先，我们通过 TypedActor 的 newInstance() 函数创建一个类型化角色的实例。随后，我们调用了 getUnitsAvailable() 函数来获取电源当前电量值。请注意，该函数会返回一个值，此时我们的主调线程（即主线程）将会被阻塞，直至类型化角色响应为止。与 getUnitsAvailable() 所不同的是，由于返回值是 void，不会返回任何响应，所以对于 useEnergy() 的调用是非阻塞的，即接下来的连续两次 useEnergy() 函数调用都是立刻返回的。在经历一个短暂的延时之后，我们会再调用一次 useEnergy() 以研究角色和线程的行为。接下来，为了让异步消息能够被处理完，我们又插入了一个 1 秒的延时，并紧接着又再次查询电源的使用次数和当前电量。在程序结尾，我们关停了所有角色。下面让我们来看看这段代码的输出结果：

```
Thread in main: main
Energy units 100
Firing two requests for use energy
Fired two requests for use energy
Thread in useEnergy: akka:event-driven:dispatcher:global-2
Thread in useEnergy: akka:event-driven:dispatcher:global-2
Firing one more requests for use energy
Thread in useEnergy: akka:event-driven:dispatcher:global-3
Energy units 70
Usage 3
```

由于类型化角色 EnergySourceImpl 每次只会执行一个函数，所以虽然前两次 useEnergy() 请求并未阻塞主线程，但这两个任务都是在角色线程中顺序执行的。上面的测试用例优雅地将执行线程在 main 函数的调用与角色里的函数之间进行了数次切换，这使得我们可以更清楚地观察到：当 main() 函数运行在主线程中时，角色中的函数也在另外一个由 Akka 管理的线程（global-2）中顺序执行着。此外，我们还注意到角色并没有占住执行线程不放，证据就是最后一个 useEnergy() 函数是在另外一个 Akka 管理的线程（global-3）中运行的。

电源的可变状态被隔离在 EnergySourceImpl 角色中，我称之为隔离的，不仅是因为可变状态被封装在 EnergySourceImpl 类的定义代码中，而是由于在类型化角色的控制之下的任意时刻，最多只会有一个角色的执行线程可以访问该可变状态。

8.7.2　在 Scala 中使用类型化角色

前面我们已经看到，在 Java 中使用类型化角色需要一组接口 / 实现对。而在 Scala 中，我们不再创建接口，而是改为创建一个不含实现代码的 trait。下面让我们将 EnergySource 用 Scala 改写为一个 trait：

favoringIsolatedMutability/scala/typed1/EnergySource.scala

```
trait EnergySource {
  def getUnitsAvailable() : Long
  def getUsageCount() : Long
  def useEnergy(units : Long) : Unit
}
```

EnergySourceImpl 的实现完全是对其 Java 版本同名类的直接翻译。该类继承了 TypedActor 并附带了之前定义的 EnergySource trait。

favoringIsolatedMutability/scala/typed1/EnergySourceImpl.scala

```
class EnergySourceImpl extends TypedActor with EnergySource {
  val MAXLEVEL = 100L
  var level = MAXLEVEL
  var usageCount = 0L

  def getUnitsAvailable() = level

  def getUsageCount() = usageCount

  def useEnergy(units : Long) = {
    if (units > 0 && level - units >= 0) {
      println("Thread in useEnergy: " + Thread.currentThread().getName())
      level -= units
      usageCount += 1
    }
  }
}
```

最后，我们还需要实现一个对上面所有功能进行检验的测试用例 UseEnergySource：

favoringIsolatedMutability/scala/typed1/UseEnergySource.scala

```
object UseEnergySource {
  def main(args : Array[String]) : Unit = {
    println("Thread in main: " + Thread.currentThread().getName())

    val energySource = TypedActor.newInstance(
      classOf[EnergySource], classOf[EnergySourceImpl])

    println("Energy units " + energySource.getUnitsAvailable)

    println("Firing two requests for use energy")
    energySource.useEnergy(10)
    energySource.useEnergy(10)
    println("Fired two requests for use energy")
    Thread.sleep(100)
    println("Firing one more requests for use energy")
    energySource.useEnergy(10)

    Thread.sleep(1000);
    println("Energy units " + energySource.getUnitsAvailable)
    println("Usage " + energySource.getUsageCount)

    TypedActor.stop(energySource)
  }
}
```

下面让我们运行 Scala 版本的示例代码并观察其输出结果：

```
Thread in main: main
Energy units 100
Firing two requests for use energy
Fired two requests for use energy
Thread in useEnergy: akka:event-driven:dispatcher:global-2
Thread in useEnergy: akka:event-driven:dispatcher:global-2
Firing one more requests for use energy
Thread in useEnergy: akka:event-driven:dispatcher:global-3
Energy units 70
Usage 3
```

与 Java 版本的实现相比，Scala 版本的示例程序除了受益于活动对象之外，其代码简洁性也更胜一筹。

8.8　类型化角色和 murmurs

使用了类型化角色的 EnergySource 使我们能够以调用函数的形式来掩盖后台顺序处理异步消息的过程，在实现了线程安全的同时又可以免去显式同步的困扰。虽然创建类型化角色并不困难，但此时我们的 EnergySource 却还是一个丢失了关键特性的半成品，即还没有可以周期性自动补充电量的能力。

在上一章我们所实现的版本中，由于整个动作都是在后台完成，所以电量补充的动作是不需要任何用户介入的。只要我们启动了电源，就会有一个专门的 timer 负责每秒钟为电源增加一单位电量。

然而在使用了类型化角色的版本中，实现这一特性还需要多费一番工夫才行，我们需要确保自动恢复动作不会破坏类型化角色的单线程特性。在真正进入编码阶段之前，让我们先考虑一下有哪些备选的实现方案。

方案一。我们可以将 replenish() 函数加入到 EnergySource 接口中，这样一来，电源的使用者就可以每秒一次周期性地调用这个函数。但不幸的是，这种做法不但为电源的使用者增添了不必要的负担，而且还面临着用户忘记执行函数调用的风险。此外，电量自动恢复并非电源的通用能力，其他版本的 EnergySource 或许根本就没这个功能，所以 replenish() 并不适合加入到 EnergySource 接口中。综上所述，我们排除此方案。

方案二。我们可以在类型化角色中创建一个 timer，并让这个 timer 负责周期性地恢复电源电量。此外，TypedActor 中还定义了两个名为 preStart() 和 postStop() 的特殊函数，前者会在角色被创建之时调用，而后者则会在角色被终止或关闭之后被调用。这两个函数的特性很符合本方案的需求，即我们可以在 preStart() 函数里启动 timer，并在 postStop() 函数里将其关停。然而虽然粗看起来本方案似乎挺不错，但实际上这种做法会引发其他的问题。

方案二的缺陷在于，timer 运行在自己独立的线程中，而我们不想让那些线程修改角色里的可变状态。请记住，我们希望状态是隔离可变的，而不是共享可变的。所以，我们所需要的是一种能够被角色正确执行、可以引发内部函数调用的方法，我称该方法为 murmurs。这些 murmurs 对类型化角色的外部用户来说是透明的，而实际上 murmurs 是以异步消息的形式运作的，其本质与那些受外部调用而产生消息是相同的。下面让我们来研究如何用这种方式来编写代码。

请记住，类型化角色实际上是加强版的角色，它们同样也像普通角色那样有接收消息的功能。基类 TypedActor 的 receive() 函数主要负责从代理中接收消息，并将其分派给类中合适的函数。我们可以覆写该函数来实现针对 murmurs 的特殊消息，即内部操作。通过这种方式，角色的使用者还是可以调用随接口发布出去的那些函数，而我们的内部类则可以使用这个（未发布出去）的消息，二者互不干扰。此外，如果需要的话，我们甚至还可以增加一个额外的步骤来确保这个消息的发送方真的是我们自己的角色。

后面我们将会看到，用 Java 实现上述方案还是有些麻烦的，而在 Scala 中则会简单很多。下面我们还是先研究在 Java 中如何实现，然后再将代码翻译成 Scala 版本。

8.8.1　用 Java 实现 murmurs

所有 EnergySourceImpl 的外部用户都是通过 EnergySource 接口与其进行交互的，而在内部，我们将利用一个 timer 来实现让电源角色每秒向自己发送一个 Replenish 请求消息的动作。虽然为了阻止外部用户的直接调用，我们会将 replenish() 函数定义为 private，但我们同样也应避免从 timer 里直接调用该函数，以保持角色消息收发的交互方式不被破坏。下面让我们来看看其中的部分代码：

favoringIsolatedMutability/java/typed2/EnergySourceImpl.java

```
@SuppressWarnings("unchecked")
public class EnergySourceImpl extends TypedActor implements EnergySource {
  private final long MAXLEVEL = 100L;
  private long level = MAXLEVEL;
  private long usageCount = 0L;
  class Replenish {}
  @Override public void preStart() {
    Scheduler.schedule(
      optionSelf().get(), new Replenish(), 1, 1, TimeUnit.SECONDS);
  }
@Override public void postStop() { Scheduler.shutdown(); }

private void replenish() {
  System.out.println("Thread in replenish: " +
    Thread.currentThread().getName());
  if (level < MAXLEVEL) level += 1;
}
```

之前我们曾经提到过，TypedActor 定义了一个角色启动之后会被自动调用的 preStart() 函数，本例中启动 timer 的工作就是在这个函数里完成的。这里我们所使用的 Akka Scheduler 是一种基于角色的 timer，它提供了一组重载的 schedule() 函数来执行一次性或重复性任务。我们可以用 timer 来执行任意函数，亦或向本例那样向角色发送消息。

在 preStart() 函数中，我们设定了一个每秒向角色发送一个 Replenish 消息的 timer，该 timer 将在初始化动作结束 1 秒钟后启动。通过调用 TypedActor 实例的 optionSelf() 函数，我们可以拿到角色的 ActorRef 引用的句柄。当角色停止时，timer 也需要立刻终止运行，因此这里我们还会用到 postStop() 函数。在私有的 replenish() 函数中，我们并没有对 Replenish 消息进行任何处理，而是简单地将 level 变量加 1 便返回了。

类型化角色的用户所使用的代理负责将函数调用转换成消息。而 TypedActor 基类的 receive() 函数负责将这些消息转换成实现类的函数调用。如果我们检查 receive() 函数的定义就会发现，其实该函数的返回值是一个 scala.PartialFunction ⊖。为了不过多纠结于语言细节，这里你将 PartialFunction 想象成一个经过修饰的 switch 语句就可以了，其功能是根据所收到的消息类型将消息分发到不同的代码处理逻辑中去。虽然基类替我们做了消息与函数之间的映射，但我们还希望它把我们的私有消息与函数的映射放在一起处理。换句话说，我们希望把自己的消息处理逻辑合并到基类的 receive() 函数的处理流程中去。PartialFunction 的 orElse() 函数可以帮助我们很轻松地实现上述需求，所以我们将会像下面这样实现 receive() 函数：

favoringIsolatedMutability/java/typed2/EnergySourceImpl.java

```
@Override public PartialFunction receive() {
  return processMessage().orElse(super.receive());
}
```

⊖ 其实在技术上，我们应该更精确地将其描述为 scala.PartialFunction<Any,Unit>，其中 scala.Any 可以看成是 Java 里的 Object。但遗憾的是，这段代码中 Java 的类型擦除会导致一些编译期警告。

在这个覆写的 receive() 函数中，我们并将基类的 receive() 函数所返回的 PartialFunction 与尚待实现的 processMessage() 函数所返回的 PartialFunction 进行了合并。而现在我们可以将注意力转向 processMessage() 函数的实现上来了。该函数可以接收 Replenish 消息并调用私有的 replenish() 函数。由于这是消息处理序列的一部分，所以我们在不知不觉中就已经用基于角色的通信方式解决了线程同步问题。请看一下你杯子里的咖啡是否还够，不够的话请倒满吧，因为你可能在实现这个函数的时候需要一些额外的咖啡因来提提神。

PartialFunction 是一个 Scala trait，而 Java 中 trait 是作为一个接口 / 抽象类对实现的。所以要想在 Java 中实现 trait，我们需要实现 PartialFunction 接口，并将调用酌情委托给相应的抽象类。

除了 receive() 函数之外，我们还需要实现关键函数 apply() 和 isDefinedAt()。前者主要负责处理 Replenish 消息，而后者主要用于判断我们的 PartialFunction 是否支持某个特殊的消息格式或类型，而该接口的其他函数则委派给 PartialFunction$class 就可以了。最后，为了少实现一些接口函数，我们可以继承与 PartialFunction 共享相同 Function1 接口的 AbstractFunction1 类。

favoringIsolatedMutability/java/typed2/EnergySourceImpl.java

```
private PartialFunction processMessage() {

  class MyDispatch extends AbstractFunction1 implements PartialFunction {
    public boolean isDefinedAt(Object message) {
      return message instanceof Replenish;
    }

    public Object apply(Object message) {
      if (message instanceof Replenish) replenish();
      return null;
    }

    public Function1 lift() {
      return PartialFunction$class.lift(this);
    }

    public PartialFunction andThen(Function1 function) {
      return PartialFunction$class.andThen(this, function);
    }

    public PartialFunction orElse(PartialFunction function) {
      return PartialFunction$class.orElse(this, function);
    }
  };

  return new MyDispatch();
}
```

在 apply() 函数中，我们会检查所收到的消息是否 Replenish 类型，如果是就调用私有函数 replenish()，否则返回 null。函数 isDefinedAt() 的作用是指明我们只支持 Replenish 这一种消息类型，而其他消息将交由基类的 receive() 函数酌情处理。OK，最后的步骤就是把我们之前在类型化角色版本中定义的几个不需要变动的方法原封不动地搬过来，如下所示：

favoringIsolatedMutability/java/typed2/EnergySourceImpl.java

```
  public long getUnitsAvailable() { return level; }
  public long getUsageCount() { return usageCount; }
  public void useEnergy(final long units) {
    if (units > 0 && level - units >= 0) {
      System.out.println(
        "Thread in useEnergy: " + Thread.currentThread().getName());
      level -= units;
      usageCount++;
    }
  }
}
```

在这一版本的代码中，由于EnergySource接口没发生任何变化，所以测试用例UseEnergySource可以直接复用上一个版本的代码。下面让我们对新版本的EnergySourceImpl进行编译，并用在上一节中写的UseEnergySource来运行这个新的电源实现类。在上一节中，当测试用例运行到结尾处时，电源中还剩了70个单位电量。而现在，由于我们有了自动电量恢复机制，所以测试用例完成之后电源应该会比上一节的运行结果多1、2个单位电量。

```
Thread in main: main
Energy units 100
Firing two requests for use energy
Fired two requests for use energy
Thread in useEnergy: akka:event-driven:dispatcher:global-2
Thread in useEnergy: akka:event-driven:dispatcher:global-2
Firing one more requests for use energy
Thread in useEnergy: akka:event-driven:dispatcher:global-3
Thread in replenish: akka:event-driven:dispatcher:global-4
Energy units 71
Usage 3
```

从我们在代码里插入的打印语句所输出线程信息来看，消耗电量的请求和恢复电量的请求都是运行在Akka角色的线程里面的，所以我们从此可以摆脱同步问题的困扰了。由于角色是单线程的，所以如果我们在代码中插入一些sleep调用来迟滞这些任务的话，就可以看到接下来的那些调用角色的动作也同样也会被延迟。

8.8.2　用 Scala 实现 Murmurs

Murmurs方法的本质是将我们自己的PartialFunction实现与基类的receive()函数所返回的PartialFunction合并在一起。这种方式用Java实现起来比较繁琐，但用Scala实现则会简单很多。下面就让我们一起来看看如何在Scala中处理murmus（即内部消息）。

favoringIsolatedMutability/scala/typed2/EnergySourceImpl.scala

```
class EnergySourceImpl extends TypedActor with EnergySource {
  val MAXLEVEL = 100L
  var level = MAXLEVEL
  var usageCount = 0L
  case class Replenish()
```

```
override def preStart() =
  Scheduler.schedule(self, Replenish, 1, 1, TimeUnit.SECONDS)
override def postStop() = Scheduler.shutdown
override def receive = processMessage orElse super.receive

def processMessage : Receive = {
  case Replenish =>
    println("Thread in replenish: " + Thread.currentThread.getName)
    if (level < MAXLEVEL) level += 1
}
```

在上面的代码中，preStart() 函数和 postStop() 函数是从 Java 版代码简单翻译过来的，这里就不再赘述。消息类 Replenish 在这里变成了一个 case 类。而 receive() 函数除了在形式上保持了 Scala 语法简洁的特性之外，其功能和 Java 版本完全相同。这里面变动最大的要数 processMessage() 函数了。由于 Scala 版本的实现无需对 Replenish 消息进行模式匹配，所以也就不会出现 Java 实现版本中那些乱七八糟的继承和委派（delegation）。也正是基于这种简单性，我们还可以将 Replenish() 函数的逻辑也放在这里，这样我们就不用再创建一个私有函数了。此外，对于 processMessage() 函数的返回值类型而言，我们既可以将其定义为 PartialFunction[Any, Unit]，也可以和上例一样使用 Receive。由于 Receive 是 PartialFunction[Any,Unit] 的别名（alias），所以二者并无本质区别。

下面要实现的代码由于功能本身没有任何变化，所以我们直接把前一个 Scala 版本的代码抄过来即可：

favoringIsolatedMutability/scala/typed2/EnergySourceImpl.scala

```
  def getUnitsAvailable() = level
  def getUsageCount() = usageCount
  def useEnergy(units : Long) = {
    if (units > 0 && level - units >= 0) {
      println("Thread in useEnergy: " + Thread.currentThread.getName)
      level -= units
      usageCount += 1
    }
  }
}
```

最后，对于 EnergySource 和 UseEnergySource 的实现，我们也直接把上一个 Scala 版本的代码拿过来用即可。下面让我们编译运行新版本的 EnergySourceImpl，并将输出结果与之前的版本进行比较：

```
Thread in main: main
Energy units 100
Firing two requests for use energy
Fired two requests for use energy
Thread in useEnergy: akka:event-driven:dispatcher:global-2
Thread in useEnergy: akka:event-driven:dispatcher:global-2
Firing one more requests for use energy
Thread in useEnergy: akka:event-driven:dispatcher:global-3
Thread in replenish: akka:event-driven:dispatcher:global-4
Energy units 71
Usage 3
```

就代码实现角度而言，同样的功能用 Scala 实现要比用 Java 实现简单得多。即使两段实例代码都得到了相同的逻辑结果，但是 Scala 版本明显要比 Java 版本在代码和逻辑实现方面容易一些。

8.9　混合使用角色和 STM

角色可以帮助很好地隔离可变状态。尤其是当问题能够被拆分成可以独立运行的多个并发任务、并且并发任务彼此之间都是通过消息进行异步通信时，角色的表现更佳。但是，角色并未提供对跨任务的一致性进行管理的方法。所以如果我们希望两个或多个角色的动作要么全部成功、要么全部失败，则角色就无法独立实现，而此时我们就需要通过引入 STM 来与角色配合完成此类功能。在本节中，我假定你已经阅读过第 6 章以及本章中有关角色和类型化角色的相关内容。

我们在 4.6 节和 6.9 节中曾经实现过一个用于在两个 Account 之间转账的 AccountService 类，该类可以作为我们理解角色和 STM 之间这种相互作用的绝佳示例。在这个示例中，存款和取款这两个操作对于单个账户来说都是隔离的，所以 Account 可以用一个简单的角色或一个类型化角色来实现。然而转账动作需要一对儿跨越两个账户的存取款动作配合起来才能完成。换句话说，在一个完整的转账过程中，由某个角色所负责的存款动作若想成功，当且仅当另一个角色所负责的取款动作也成功时才能达成。下面就让我们用角色和 STM 混搭的方式来实现这个转账示例。

Akka 提供了一些混合使用角色和 STM 的选项。例如，我们可以创建一个独立的事务协调对象，并手动将多个角色按顺序摆放在事务中（关于这一层次的控制细节，请参阅 Akka 的帮助文档）。除此之外，我们还有两种用于管理角色间事务的方法可供选择。我们将会在下面两节分别来研究这两种方法：即如何使用 transactor 以及如何协调类型化角色。

8.10　使用 transactor

Akka transactor 或事务角色为我们提供了一种将多个角色的执行过程合并到一个事务中的方法。顾名思义，transactor 可以将多个角色对于托管 STM Ref 对象的更改变成原子操作，即仅当外围事务提交成功之后，对于那些托管对象的变更才能生效，否则所有的变更都会被丢弃。

Transactor 提供了三种处理消息的方法：

- 默认情况下，Transactor 会在其自己的事务中处理消息。
- 实现 normally() 函数。该函数不属于任何事物，其主要功能是独立地处理我们所选择的消息。
- 申请让消息被协调处理，即使其作为总控事务的一部分来执行。

总体而言，Transactor 为我们提供了将其他角色链接到我们的协调事务里的弹性。此外，transactor 还提供了前置和后置事务的可选函数，以便于我们可以提前为事务做好准备或执行某些后置提交操作。

还是老规矩，我们先用 Java 创建一个 transactor，然后再用 Scala 实现一遍。

8.10.1　在 Java 中使用 transactor

为了能够在 Java 中使用 transactor，我们需要继承 UntypedTransactor 类并实现 atomically() 函数。除此之外，如果我们想要在事务中包含其他角色，则还需要实现 coordinate() 函数。下面就让我们用 transactor 来重新实现账户转账的例子。首先还是从我们将会用到的消息类开始说起。

在新版的示例中，我们使用 Deposit 消息来完成账户的存款操作，该消息由一个含有存款金额字段的不可变类 Deposit 表示。

favoringIsolatedMutability/java/transactors/Deposit.java

```
public class Deposit {
  public final int amount;
  public Deposit(final int theAmount) { amount = theAmount; }
}
```

接下来，我们还需要定义一个与 Deposit 结构完全相同的消息类 Withdraw：

favoringIsolatedMutability/java/transactors/Withdraw.java

```
public class Withdraw {
  public final int amount;
  public Withdraw(final int theAmount) { amount = theAmount; }
}
```

对于获取账户余额的请求消息而言，由于本身不需要携带任何数据，所以我们只需定义一个空类即可：

favoringIsolatedMutability/java/transactors/FetchBalance.java

```
public class FetchBalance {}
```

与此相对地，针对上述请求消息的响应消息 Balance 则是一个含有有实际账户余额的不可变类：

favoringIsolatedMutability/java/transactors/Balance.java

```
public class Balance {
  public final int amount;
  public Balance(final int theBalance) { amount = theBalance; }
}
```

最后，我们还需要定义 Transfer 消息，该消息将会包含转账操作的源账户和目的账户以及待转金额：

favoringIsolatedMutability/java/transactors/Transfer.java

```
public class Transfer {
  public final ActorRef from;
  public final ActorRef to;
  public final int amount;
```

```
  public Transfer(final ActorRef fromAccount,
    final ActorRef toAccount, final int theAmount) {
    from = fromAccount;
    to = toAccount;
    amount = theAmount;
  }
}
```

在本例中，AccountService tranactor 将会用到 Transfer 消息，而 Account transactor 则会使用我们刚才定义的其他消息。下面让我们先来看一下 Account transactor 的实现代码：

favoringIsolatedMutability/java/transactors/Account.java

```
public class Account extends UntypedTransactor {
  private final Ref<Integer> balance = new Ref<Integer>(0);

  public void atomically(final Object message) {
    if(message instanceof Deposit) {
      int amount = ((Deposit)(message)).amount;
      if (amount > 0) {
        balance.swap(balance.get() + amount);
        System.out.println("Received Deposit request " + amount);
      }
    }

    if(message instanceof Withdraw) {
      int amount = ((Withdraw)(message)).amount;
      System.out.println("Received Withdraw request " + amount);
      if (amount > 0 && balance.get() >= amount)
        balance.swap(balance.get() - amount);
      else {
        System.out.println("...insufficient funds...");
        throw new RuntimeException("Insufficient fund");
      }
    }

    if(message instanceof FetchBalance) {
      getContext().replySafe(new Balance(balance.get()));
    }
  }
}
```

Account 类继承自 UntypedTransactor 并且实现了 atomically() 函数。该函数将会运行在一个给定事务的上下文环境中，这里的事务可以是调用方所在的事务，如果没显式给出的话，也可能是一个独立的事务。在 atomically() 函数中，如果接收到的消息类型为 Deposit，我们就会把存款的数额与当前余额相加之后保存在 STM 托管的 Ref 对象中。如果接收到的消息类型为 Withdraw 并且当前余额大于取款金额时，我们才会在 balance 中减去取款金额，否则我们会抛出一个异常。而一旦抛出异常，该行为将会触发外围事务的回滚。最后，如果收到的消息类型为 FetchBalance，我们只需把当前账户余额 balance 的值返回给发送方即可。由于整个函数都是在一个事务中运行的，所以在一个 transactor 中对于 Ref 对象进行多次访问是没关系的。而仅当外围事务被提交之后，我们对 Ref 对象所做的变更才能生效，

请记住，示例中所涉及的不可变状态是需要我们人工维护的。

下面我们将实现AccountService transactor，其主要功能就是负责协调目标账户（transactor）上的存款操作和源账户（另一个transactor）上的取款操作，实现代码如下所示：

favoringIsolatedMutability/java/transactors/AccountService.java

```
public class AccountService extends UntypedTransactor {

  @Override public Set<SendTo> coordinate(final Object message) {
    if(message instanceof Transfer) {
      Set<SendTo> coordinations = new java.util.HashSet<SendTo>();
      Transfer transfer = (Transfer) message;
      coordinations.add(sendTo(transfer.to, new Deposit(transfer.amount)));
      coordinations.add(sendTo(transfer.from,
        new Withdraw(transfer.amount)));
      return java.util.Collections.unmodifiableSet(coordinations);
    }

    return nobody();
  }

  public void atomically(final Object message) {}
}
```

由于AccountService的唯一职责就是协调存取款操作，所以在coordinate()函数中，我们需要将合适的消息分别发送给源账户和目标账户。为了实现这一目的，我们需要将角色以及每个角色所对应的消息都聚集在一个集合中。当我们将该集合自coordinate()函数返回给调用方时，AccountService transactor的父类将会把合适的消息发往集合中的每一个transactor。而一旦消息被发出，则其自身的automically()实现将会被调用。但由于这里我们没有额外的事情要做，所以就只写了一个空的atomically()函数。

下面让我们写一些的测试代码来检验上述这些transactor的功能：

favoringIsolatedMutability/java/transactors/UseAccountService.java

```
public class UseAccountService {
  public static void printBalance(
    final String accountName, final ActorRef account) {

    Balance balance =
      (Balance)(account.sendRequestReply(new FetchBalance()));
    System.out.println(accountName + " balance is " + balance.amount);
  }

  public static void main(final String[] args)
    throws InterruptedException {
    final ActorRef account1 = Actors.actorOf(Account.class).start();
    final ActorRef account2 = Actors.actorOf(Account.class).start();
    final ActorRef accountService =
      Actors.actorOf(AccountService.class).start();

    account1.sendOneWay(new Deposit(1000));
    account2.sendOneWay(new Deposit(1000));
```

```
        Thread.sleep(1000);

        printBalance("Account1", account1);
        printBalance("Account2", account2);

        System.out.println("Let's transfer $20... should succeed");
        accountService.sendOneWay(new Transfer(account1, account2, 20));
        Thread.sleep(1000);

        printBalance("Account1", account1);
        printBalance("Account2", account2);

        System.out.println("Let's transfer $2000... should not succeed");
        accountService.sendOneWay(new Transfer(account1, account2, 2000));

        Thread.sleep(6000);

        printBalance("Account1", account1);
        printBalance("Account2", account2);

        Actors.registry().shutdownAll();
    }
}
```

就交互和使用方法而言，角色和 transactor 实际是没什么区别的。如果我们将一个普通的消息（如 new Deposit(1000)）发送给 transactor，则该消息将会自动被包装到一个事务中。此外，我们也可以通过创建 akka.transactor.Coordinated 实例并把消息包装进去（例如，new Coordinated(new Deposit(1000))）的方法来构建我们自己的协调事务。在本例中，由于我们只处理单向消息，所以在执行下一步查询之前我们都会插入一些延时以便使消息处理能够彻底完成。这种做法为协调事务成功执行或失败回滚提供了时间，同时也便于我们从随后的打印函数中观察到事务执行的效果。

只有在相关 transactor 的消息全部成功处理完之后，协调事务才能提交，其中协调请求的等待时间至多为事务超时时间（可配置）。上述测试代码的输出结果如下所示：

```
Received Deposit request 1000
Received Deposit request 1000
Account1 balance is 1000
Account2 balance is 1000
Let's transfer $20... should succeed
Received Deposit request 20
Received Withdraw request 20
Account1 balance is 980
Account2 balance is 1020
Let's transfer $2000... should not succeed
Received Withdraw request 2000
...insufficient funds...
Received Deposit request 2000
Account1 balance is 980
Account2 balance is 1020
```

从输出结果中我们可以看到，前两次转存款操作和第一次转账操作都干脆利落地完成，而第二次转账操作则由于待转金额大于源账户的当前余额而失败。所以虽然转账操作的存

款步骤顺利完成（由于存款和取款动作是并发执行的，所以我们从输出结果中所看到的最后一次转账操作的执行步骤可能每次都不一样），但取款步骤却没有成功，从而导致整个转账操作失败并回滚。从最后两条输出结果我们看出，第二次转账的存款步骤所产生的变更被丢弃，两个账户的余额又恢复到第二次转账之前的状态。

8.10.2　在 Scala 中使用 transactor

为了在 Java 中使用 transactor，我们需要继承 Transactor 类并实现其 atomically() 函数。而如果我们想要在事务中包含其他角色，则还需要实现 coordinate() 函数。接下来我们会将上面的示例从 Java 翻译成 Scala，首先我们还是从一些消息类的定义开始入手，如下所示，这些消息类用 Scala 的 case 类实现起来非常简洁。

favoringIsolatedMutability/scala/transactors/Messages.scala

```
case class Deposit(val amount : Int)

case class Withdraw(val amount : Int)

case class FetchBalance()

case class Balance(val amount : Int)

case  class Transfer(val from : ActorRef, val to : ActorRef, val amount : Int)
```

接下来，我们需要把 Account transactor 翻译成 Scala 的实现方式，这里我们可以使用模式匹配来处理上面定义的三种消息。

favoringIsolatedMutability/scala/transactors/Account.scala

```
class Account extends Transactor {
  val balance = Ref(0)

  def atomically = {
    case Deposit(amount) =>
      if (amount > 0) {
        balance.swap(balance.get() + amount)
        println("Received Deposit request " + amount)
      }

    case Withdraw(amount) =>
      println("Received Withdraw request " + amount)
      if (amount > 0 && balance.get() >= amount)
        balance.swap(balance.get() - amount)
      else {
        println("...insufficient funds...")
        throw new RuntimeException("Insufficient fund")
      }

    case FetchBalance =>
      self.replySafe(Balance(balance.get()))
  }
}
```

下面让我们一起来翻译 AccountService transactor。这里我们仍然将 atomically() 函数置空，同时我们在 coordinate() 函数中指定了哪些对象是需要参与到事务执行过程中的。与 Java 版的代码相比，Scala 这边的实现代码在语法上要更加简洁：

favoringIsolatedMutability/scala/transactors/AccountService.scala

```
class AccountService extends Transactor {

  override def coordinate = {
    case Transfer(from, to, amount) =>
      sendTo(to -> Deposit(amount), from -> Withdraw(amount))
  }

  def atomically = { case message => }
}
```

最后，我们用下面的测试代码来检验这些 transactor 的运行情况：

favoringIsolatedMutability/scala/transactors/UseAccountService.scala

```
object UseAccountService {

  def printBalance(accountName : String, account : ActorRef) = {
    (account !! FetchBalance) match {
      case Some(Balance(amount)) =>
        println(accountName + " balance is " + amount)
      case None =>
        println("Error getting balance for " + accountName)
    }
  }

  def main(args : Array[String]) = {
    val account1 = Actor.actorOf[Account].start()
    val account2 = Actor.actorOf[Account].start()
    val accountService = Actor.actorOf[AccountService].start()

    account1 ! Deposit(1000)
    account2 ! Deposit(1000)

    Thread.sleep(1000)

    printBalance("Account1", account1)
    printBalance("Account2", account2)
    println("Let's transfer $20... should succeed")
    accountService ! Transfer(account1, account2, 20)

    Thread.sleep(1000)

    printBalance("Account1", account1)
    printBalance("Account2", account2)

    println("Let's transfer $2000... should not succeed")
    accountService ! Transfer(account1, account2, 2000)

    Thread.sleep(6000)
```

```
    printBalance("Account1", account1)
    printBalance("Account2", account2)

    Actors.registry.shutdownAll
  }
}
```

虽然上述代码只是 Java 版本对应代码的直译，但在这里我们再次见证了 Scala 在语法简洁方面的优势。通过观察下面的输出结果我们可以看到，Scala 版示例的行为与 Java 版本是完全相同的。

```
Received Deposit request 1000
Received Deposit request 1000
Account1 balance is 1000
Account2 balance is 1000
Let's transfer $20... should succeed
Received Deposit request 20
Received Withdraw request 20
Account1 balance is 980
Account2 balance is 1020
Let's transfer $2000... should not succeed
Received Deposit request 2000
Received Withdraw request 2000
...insufficient funds...
Account1 balance is 980
Account2 balance is 1020
```

8.10.3　使用 transactor

通过上面两个示例，我们学习了如何在 Java 和 Scala 中实现 transactor。Transactor 集角色与 STM 的优点于一身，并支持多个独立运行角色之间的一致性。与 STM 的使用场景类似，transactor 非常适用于那些写冲突非常不频繁的应用程序。理想情况下，如果多个角色需要进行某种形式的投票以作出某项决定，则用 transactor 来实现将会非常方便。

8.11　调和类型化角色

正如我们在 8.7 节中所看到的那样，类型化角色是吸取了面向对象的程序设计和基于角色的程序设计二者的精华所孕育出来的新编程模型。该编程模型集所有我们耳熟能详的方法于一身，既可以方便地使用函数调用，又能享受角色所带来的好处。所以，在一个面向对象应用程序中，相比起普通的角色，我们可能会更倾向于使用类型化的角色。然而与普通角色相类似的是，类型化角色也是各自独立运行且不提供彼此间事务协调功能的，下面我们将会使用调和类型化角色（coordinating typed actor）来解决这个问题。

Akka 帮助我们简化了将一个普通的类型化角色转换成一个调和类型化角色的过程。我们只需简单地用一个特殊的 Coordinated 注解对相应的接口函数进行标记即可。为了标明某个函数调用序列运行在一个协调事务中，我们需要将这些函数调用包装在一个 coordinate() 函数里。在进行下一步操作之前，该函数（默认情况下）会等待所有函数都提交或回滚。

这种方案有一个限制，那就是只有 void 函数才能够用 Coordinated 注解进行标记。这是因为 void 函数被翻译为单向调用，并且这些函数可以参与到事务中。而返回值不为 void 的函数则会被翻译成双向阻塞调用，所以这些函数将无法参与到自由运行的并发事务中。

现在让我们再次重新实现那个我们曾在 8.10 节中摆弄过很多次的转账示例吧。

8.11.1　在 Java 中使用调和类型化角色

为了使用类型化角色，我们需要准备一对接口 / 实现类，所以我们先从 Account 和 AccountService 这两个下面将会用到的接口开始入手：

favoringIsolatedMutability/java/coordtyped/Account.java

```
public interface Account {
  int getBalance();
  @Coordinated void deposit(final int amount);
  @Coordinated void withdraw(final int amount);
}
```

favoringIsolatedMutability/java/coordtyped/AccountService.java

```
public interface AccountService {
  void transfer(final Account from, final Account to, final int amount);
}
```

在 Account 接口中，唯一比较特别的部分就是其两个接口函数都用 @Coordinated 注解进行了标记。通过这两个标记，我们声明了这些函数既可以在其自身的事务中运行，也可以加入到其调用者的事务当中。与此相对的是，由于 AccountService 的类实现将会自行管理其事务，所以 AccountService 的接口函数都没有进行标记。

favoringIsolatedMutability/java/coordtyped/AccountImpl.java

```
public class AccountImpl extends TypedActor implements Account {
  private final Ref<Integer> balance = new Ref<Integer>(0);

  public int getBalance() { return balance.get(); }

  public void deposit(final int amount) {
    if (amount > 0) {
      balance.swap(balance.get() + amount);
      System.out.println("Received Deposit request " + amount);
    }
  }

  public void withdraw(final int amount) {
    System.out.println("Received Withdraw request " + amount);
    if (amount > 0 && balance.get() >= amount)
      balance.swap(balance.get() - amount);
    else {
      System.out.println("...insufficient funds...");
      throw new RuntimeException("Insufficient fund");
    }
  }
}
```

通过继承 TypedActor，我们将 AccountImpl 声明为一个角色。在这里我们没有使用简单的本地字段，而是采用了托管的 STM 引用（Ref）。此外，虽然我们没有写任何针对事务的代码，但是由于 AccountImpl 类相关接口函数都是用 @Coordinated 标记过的，所以该类的所有函数都将运行在一个事务中。在 deposit() 函数中，如果参数 amount 的值大于 0，则 deposit() 函数将负责把 amount 的值累加到其当前余额中。相反地，如果当前余额 balance 的值大于参数 amount，则 withdraw() 函数就会在当前余额中减去 amount 的值。否则，withdraw() 函数将会抛出一个异常以表明当前操作及外围事务执行失败。如果我们没有为这些函数指定事务，则它们将会运行在自身的默认事务中。而在转账的情境下，我们希望存款和取款操作都运行在同一个事务中，所以接下来我们要写一个 AccountServiceImpl 来对这两个操作进行管理：

favoringIsolatedMutability/java/coordtyped/AccountServiceImpl.java

```
public class AccountServiceImpl
  extends TypedActor implements AccountService {

  public void transfer(
    final Account from, final Account to, final int amount) {

    coordinate(true, new Atomically() {
      public void atomically() {
        to.deposit(amount);
        from.withdraw(amount);
      }
    });
  }
}
```

在上面的代码中，transfer() 函数保证了存取款操作都将在同一事务中完成。需要在事务中执行的代码都被包装在 Atomically 接口的成员函数 atomically() 里。而从 akka.transactor.Coordination 类中静态引入的 coordinate() 函数则将以事务的形式运行 atomically() 函数中的代码块。其第一个参数 true 的作用是指示 coordinate() 要等待其事务完成（成功或回滚）之后才能返回。所有事务中的函数调用本质上都是（发送）单向消息，所以 coordinate() 函数只关心事务是否完成，而不会阻塞式地等待各函数的返回结果。

除了角色对象的创建过程略有不同之外，使用这些类型化角色的代码与使用普通对象的代码看起来没什么两样。在创建对象时，我们没有使用 new 操作符，而是使用了一个工厂类来负责创建工作，具体代码如下所示：

favoringIsolatedMutability/java/coordtyped/UseAccountService.java

```
public class UseAccountService {

  public static void main(final String[] args)
    throws InterruptedException {

    final Account account1 =
      TypedActor.newInstance(Account.class, AccountImpl.class);
```

```
    final Account account2 =
      TypedActor.newInstance(Account.class, AccountImpl.class);
    final AccountService accountService =
      TypedActor.newInstance(AccountService.class, AccountServiceImpl.class);

    account1.deposit(1000);
    account2.deposit(1000);

    System.out.println("Account1 balance is " + account1.getBalance());
    System.out.println("Account2 balance is " + account2.getBalance());

    System.out.println("Let's transfer $20... should succeed");

    accountService.transfer(account1, account2, 20);

    Thread.sleep(1000);

    System.out.println("Account1 balance is " + account1.getBalance());
    System.out.println("Account2 balance is " + account2.getBalance());

    System.out.println("Let's transfer $2000... should not succeed");
    accountService.transfer(account1, account2, 2000);

    Thread.sleep(6000);

    System.out.println("Account1 balance is " + account1.getBalance());
    System.out.println("Account2 balance is " + account2.getBalance());

    Actors.registry().shutdownAll();
  }
}
```

上述代码行为与我们之前在 8.10 节中所使用的示例完全相同。其输出结果如下所示：

```
Received Deposit request 1000
Account1 balance is 1000
Received Deposit request 1000
Account2 balance is 1000
Let's transfer $20... should succeed
Received Deposit request 20
Received Withdraw request 20
Account1 balance is 980
Account2 balance is 1020
Let's transfer $2000... should not succeed
Received Deposit request 2000
Received Withdraw request 2000
...insufficient funds...
Account1 balance is 980
Account2 balance is 1020
```

正如我们所期待的那样，调和类型化角色版本的输出结果与 transactor 版本的输出结果基本相同，最后一个失败的转账事务所产生的所有变更最终都被丢弃。

8.11.2　在 Scala 中使用调和类型化角色

下面让我们将上述 Java 版本的示例代码翻译成 Scala。在 Scala 中，我们才采用 trait 来

代替接口，而这也是两种语言在实现方面的第一个不同点。

favoringIsolatedMutability/scala/coordtyped/Account.scala

```
trait Account {
  def getBalance() : Int
  @Coordinated def deposit(amount : Int) : Unit
  @Coordinated def withdraw(amount : Int) : Unit
}
```

favoringIsolatedMutability/scala/coordtyped/AccountService.scala

```
trait AccountService {
  def transfer(from : Account, to : Account, amount : Int) : Unit
}
```

Account trait 的实现是从 Java 版本直译过来的：

favoringIsolatedMutability/scala/coordtyped/AccountImpl.scala

```
class AccountImpl extends TypedActor with Account {
  val balance = Ref(0)

  def getBalance() = balance.get()

  def deposit(amount : Int) = {
    if (amount > 0) {
      balance.swap(balance.get() + amount)
      println("Received Deposit request " + amount)
    }
  }

  def withdraw(amount : Int) = {
    println("Received Withdraw request " + amount)
    if (amount > 0 && balance.get() >= amount)
      balance.swap(balance.get() - amount)
    else {
      println("...insufficient funds...")
      throw new RuntimeException("Insufficient fund")
    }
  }
}
```

同样，AccountService trait 的实现也从 Java 代码直译过来即可：

favoringIsolatedMutability/scala/coordtyped/AccountServiceImpl.scala

```
class AccountServiceImpl extends TypedActor with AccountService {
  def transfer(from : Account, to : Account, amount : Int) = {
    coordinate {
      to.deposit(amount)
      from.withdraw(amount)
    }
  }
}
```

在 Scala 版本的示例中，定义了事务各个组成部分的 coordinate() 函数的调用方式比在

Java 版里简化了很多。默认情况下，coordinate() 函数需要等待其事务执行完毕（成功或回滚）之后才能返回。同时，由于事务中所有的函数调用本质上都是（发送）单向消息，所以 coordinate() 函数只关心事务是否完成，而不会阻塞式地等待各函数的返回结果。此外，我们还可以向其传递一个可选参数，如 coordinate(wait=false)，来告知 coordinate() 函数不用等待事务完成。

最后，我们还需要一个检验上述代码的测试用例：

favoringIsolatedMutability/scala/coordtyped/UseAccountService.scala

```
object UseAccountService {

  def main(args : Array[String]) = {
    val account1 =
      TypedActor.newInstance(classOf[Account], classOf[AccountImpl])
    val account2 =
      TypedActor.newInstance(classOf[Account], classOf[AccountImpl])
    val accountService =
      TypedActor.newInstance(
        classOf[AccountService], classOf[AccountServiceImpl])

    account1.deposit(1000)
    account2.deposit(1000)

    println("Account1 balance is " + account1.getBalance())
    println("Account2 balance is " + account2.getBalance())


    println("Let's transfer $20... should succeed")

    accountService.transfer(account1, account2, 20)

    Thread.sleep(1000)

    println("Account1 balance is " + account1.getBalance())
    println("Account2 balance is " + account2.getBalance())

    println("Let's transfer $2000... should not succeed")
    accountService.transfer(account1, account2, 2000)

    Thread.sleep(6000)

    println("Account1 balance is " + account1.getBalance())
    println("Account2 balance is " + account2.getBalance())

    Actors.registry.shutdownAll
  }
}
```

正如我们从下面的输出结果中所看到的那样，Scala 实现版本的行为与 Java 版本完全相同：

```
Received Deposit request 1000
Received Deposit request 1000
Account1 balance is 1000
```

```
Account2 balance is 1000
Let's transfer $20... should succeed
Received Deposit request 20
Received Withdraw request 20
Account1 balance is 980
Account2 balance is 1020
Let's transfer $2000... should not succeed
Received Deposit request 2000
Received Withdraw request 2000
...insufficient funds...
Account1 balance is 980
Account2 balance is 1020
```

8.12　远程角色

截至目前我们所写的关于角色的例子中，所有角色及其客户端都运行于同一 JVM 进程中。但在现实生活中，有一部分开发者认为角色也应该像在 Erlang 中那样被用于进程间通信。而另一部分开发者则像我们在前面所演示的那样只将其应用于进程内通信。值得说明的一点是，Scala 和 Akka 同时兼顾了这两个阵营的需求。

在 Akka 中，远程角色的用法与进程内角色的用法十分相似，唯一的区别就在于我们如何访问角色。Akka 在底层使用了 JBoss Netty 和 Google Protocol Buffers 库来实现远程操作与本地调用的无缝衔接，使我们可以跨越进程边界，将任意角色所产生的序列化消息和引用传递给任意的远程角色。Akka 提供了通过编程和配置选项两种方式来配置主机名、端口号、消息帧的大小、安全设置等配置信息。这些配置信息的详情可以参阅 Akka 的帮助文档，为了简单起见，在本节的示例中我们将只使用默认设置。

为了能够远程访问角色，除了创建角色，我们需要对角色进行注册以便将角色与一个名字或 ID 绑定起来，从而使客户端能够借此识别所需访问的远程角色。在完成注册之后，客户端就可以通过注册 ID、主机名和端口号来给角色发请求了。发送给角色的请求既可以是单向消息也可以是双向消息，形式上与和本地角色的交互基本类似。

下面让我们一起来创建一个使用远程角色的例子。过去我曾是一个系统管理员，有很多单机系统指标，如可用磁盘空间、系统性能、CPU 使用率等都是系统管理员经常需要关注的。除此之外，通过在机房各处安装的传感器，我们可以通过监测室内温度、湿度等环境指标来保证高性能计算实验室可以一直正常运作。此时如果能有一个应用程序可以帮我从这些远程机器上把这些信息收集汇总起来就更好了，下面就让我们一起来写一个吧。

我们令 Monitor 角色负责接收从各个远程客户端发来的系统信息。部署于各台机器上的客户端可以自行决定何时向 Monitor 发送这一信息。在本例中，上述客户端每次都会将系统信息分作几组数据发送给 Monitor。

下面就让我们先从 Monitor 类开始入手。该角色就是各个客户端将要与之交互的远程角色：

favoringIsolatedMutability/java/remote/monitor/Monitor.java

```
public class Monitor extends UntypedActor {
  public void onReceive(Object message) {
    System.out.println(message);
  }

  public static void main(final String[] args) {
    Actors.remote().start("localhost", 8000)
      .register("system-monitor", Actors.actorOf(Monitor.class));

    System.out.println("Press key to stop");
    System.console().readLine();
    Actors.registry().shutdownAll();
    Actors.remote().shutdown();
  }
}
```

在上面的代码中，我们创建远程角色的方法与创建进程内角色别无二致，我们继承了 UntypedActor 并实现了其 onReceive() 函数。如果需要的话，我们还可以将收到的数据汇总生成为一个带图表的、展示更丰富的简报。但为了清晰起见这里我们还是先将主要注意力集中到角色的使用上面，所以这里我们只是简单地将收到的消息打印输出来。

在 main() 函数中，我们首先用指定的主机名和端口号启动了一个远程服务。随后我们以“system-monitor”为 ID 将其注册为可远程访问的角色。当需要关闭监控服务的时候，我们调用可以帮助我们终止远程服务的函数。以上就是监控服务所需的全部代码。下面再让我们看一下客户端的代码：

favoringIsolatedMutability/java/remote/client/Client.java

```
public class Client {
  public static void main(final String[] args) {
    ActorRef systemMonitor = remote().actorFor(
      "system-monitor", "localhost", 8000);

    systemMonitor.sendOneWay("Cores: " +
      Runtime.getRuntime().availableProcessors());
    systemMonitor.sendOneWay("Total Space: " +
      new File("/").getTotalSpace());
    systemMonitor.sendOneWay("Free Space: " +
      new File("/").getFreeSpace());
  }
}
```

为了能够访问远程角色，客户端需要在初始化时指定远程角色的 ID、主机名和端口号。一旦拿到 ActorRef 的引用，我们就可以将系统信息（本例中用的是字符串）通过消息发送给远程的监控服务。

下面请打开两个命令行窗口，首先在一个窗口中启动监控服务，然后在另一个窗口中启动客户端。每次运行客户端程序之后，监控服务都会把收到的新信息打印出来，具体内容如下所示：

```
Press key to stop
Cores: 2
Total Space: 499763888128
Free Space: 141636308992
```

为了使 Monitor 和客户端的代码能够通过编译和运行，我们需要引入几个额外的 jar 文件：akka-remote-1.1.2.jar、protobuf-java-2.3.0.jar、netty-3.2.3.Final.jar 和 commons-io-2.0.1.jar。

请不要被远程角色易于创建的假象所蒙蔽。在远程角色的使用场景中，我们还是需要保证所有消息都是不可变的（且可序列化的）并且角色不能更改任何共享可变状态。

8.13　基于角色模型的局限性

基于角色的模型降低了隔离可变性编程的难度，但该模型在适用场景上还是存在一些限制。

由于角色是通过消息来进行彼此间通信的，所以在那些没有强制不可变性的语言中，我们就必须人工来保证消息都是不可变的。传递可变消息将导致线程安全问题并最终使整个应用陷入共享可变性的险境当中，所以当手头的辅助工具还没有发展到可以帮助我们自动查验消息的不可变性之前，保证消息不可变性的重担暂时还是得由我们程序员来肩负。

角色都是各自异步运行的，彼此之前可以通过传递消息来进行协作。但某些角色的意外失败有可能导致其他角色饿死，一个或多个角色可能会一直等待某些关键的协作消息，而这些消息可能由于本应发出该消息的角色失败而永远无法抵达。因此，我们需要在编码时更加谨慎，在角色中加入异常情况的处理逻辑，并将错误消息传播给那些等待响应的角色。

当两个或多个角色互相等待对方发来的消息时，则所有角色都将陷入死锁。我们必须在设计层面谨小慎微，以确保角色在协作过程中不会陷入死锁状态。与此同时，我们应该使用超时来保证程序不会由于协作环节中个别角色失败无法响应而导致其他角色无限期的等待。

角色一次只能处理一个消息请求，所以无论是动作消息还是请求某个响应或状态的消息，在角色内部都是顺序处理的。对于那些只读取某些值的任务而言，这种做法可能会降低程序整体的并发度。所以我们最好还是用粗粒度的消息来设计应用程序。此外，我们还可以通过设计单向的“发送并遗忘”类型的消息代替双向消息来减少角色之间的相互等待。

并非所有的应用程序都适合用基于角色的模型实现。只有在待解决的任务可以被拆分成多个彼此相互独立的子任务并且子任务之间只有少量交互的情况下，使用角色才是合适的。但如果子任务之间需要频繁交互或者子任务们需要通过形成一个裁决集（quorum）的形式来进行协作，则使用基于角色的模型就是不合适的。而对于这类问题，我们可以尝试将基于角色的模型与其他并发模型混搭或干脆考虑重新设计。

8.14　小结

角色是单线程的，彼此间通过传递消息来进行交互。

通过本章的学习，我们了解到角色有如下特性：

- 降低了隔离可变性的使用门槛
- 从根本上消除了同步问题
- 提供了高效的单向消息，但同时也提供了不怎么高效的发送并等待功能
- 可扩展性非常强，同时由于角色都是单线程的，所以我们可以很方便地用线程池来对其进行管理
- 允许我们发送消息，但同时也通过接口的方式支持类型化版本（在 Akka 中）。
- 允许我们通过事务来完成多角色之间的协作。

虽然角色为我们提供了一种强大的编程模型，但该模型在某些方面仍有一些限制。例如，如果我们在使用角色进行设计的时候没有考虑周到，则程序可能存在角色饿死或死锁的潜在风险。此外，我们还需要保证消息的不可变性，这一点在没有语言层面的支持的环境中尤为重要。

第 9 章我们将学习如何在各种 JVM 支持的语言中使用角色。

第⑨章

在 Groovy、Java、JRuby 和 Scala 中使用角色

在前面的章节中，我们学习了角色的概念和基本用法。本章我们将学习如何在一些流行的 JVM 语言中使用基于角色的模型来编程。

在我们熟知的诸多语言中，Erlang 广泛使用了进程间基于角色的编程模型；而 Scala 则将一种 Erlang 风格的模型引入到进程内角色间的通信当中；Clojure 则不直接支持角色，Rich Hickey 在 http://clojure.org/state#actors 中叙述了其理由。但 Clojure 支持一种表示共享可变实体但又可以异步、独立且互斥更新的代理（agent）。我们将在附录 1 中讨论与 Clojure 代理相关的内容。

虽然 Akka 角色也可以用在 Groovy 中，但本章首节我们将使用与 Groovy 联系更为紧密的并发库 GPars 来代替 Akka 角色。除此之外，本章剩余部分将主要讲述如何在其他 JVM 语言中使用 Akka 角色。

向角色发消息看上去似乎没什么太大问题，而真正值得我们关心的问题是创建角色，在某些语言中，这一过程涉及继承某些类并且覆写其部分函数。此外，虽然在后面的小节中我们会遇到一些小问题，但是这些问题对于已经将本书阅读到这一章的你来说都是小菜一碟。

在阅读本章时，你可以将注意力集中在你所感兴趣的语言上面，其他不感兴趣的部分略过就好。

9.1 在 Groovy 中使用 GPars 提供的角色实现

在 Groovy 中，我们有很多种方式可以用来创建角色。

例如，我们可以通过继承 Akka 的 UntypedActor 来实现 Akka 角色，抑或继承用其他 JVM 语言写的类等。所以如果想要简单地实现角色，我们只需要直接覆写 UntypedActor 的 onReceive() 函数就行了。在该函数中，我们可以检查消息类型并执行相应的动作。除此之外，我们还可以使用其他基于 Java 的角色框架。但是在本章中，我们将使用一种与 Groovy 契合度更好的并发类库 GPars。

GPars 是一个用 Java 编写的类库，它同时也把对于并发编程的热爱带入了 Groovy 和 Java。该类库兼具 Java 的性能和 Groovy 在代码编写上的优雅，同时还提供了大量的有用功能，异步、并发数组操作、fork/join、代理、数据流以及角色。虽然本书不会讨论 GPars 的全部细节，但是我将会使用 GPar 角色来为你展示如何实现我们之前曾经见到过的一些示

例。如果想要了解更多有关 GPars 的信息，请参阅 GPars 的帮助文档。

首先，GPars 中的角色有很完善的基础架构：它们会在合适的时间跨越内存栅栏、在任意给定时刻只允许一个线程运行其接收函数，并在底层提供消息队列的支持。GPars 提供了多种不同风格的角色，并允许我们对诸如线程亲和力的公平性等参数进行设置。此外，正如我们下面将会看到的那样，GPars 创建和控制角色的 API 用起来也是相当顺畅。

9.1.1　创建角色

在 GPars 中，我们可以通过继承 DefaultActor 类或简单传递一个代码闭包给 Actors 类的 actor() 函数这两种方式来创建角色。如果只是需要实现一个简单的消息循环，我们可以使用后者。而如果待实现的逻辑比较复杂或者需要调用 Actor 类的其他函数，则我们应该采用继承 DefaultActor 类的方法来实现。

需要注意的是，我们需要先启动角色，然后再接收消息。因为任何试图在角色启动之前就向其发送消息的行为都将导致系统抛出异常。下面让我们先用 DefaultActor 类创建一个角色，随后再使用 actor() 函数来完成同样的功能。

下面我们将要创建的这个角色的主要功能是接收 String 类型的消息并将收到的消息输出到控制台作为响应。在下面的代码示例中我们将会看到，为了实现这一功能我们继承了 DefaultActor 类并实现了其 act() 函数。该函数包含了一个消息循环，即通过调用无限循环的 loop() 函数和 react() 函数来接收和处理消息。当角色执行到 react() 函数时，该角色将被阻塞并交出其执行线程的所有权。而当有消息到达时，线程池将会调度一个线程来执行我们为 react() 函数所提供的闭包。一旦 react() 中的逻辑执行完毕，线程的执行将会重新回到 loop() 函数，并同时将角色再次插入消息等待队列里。在本例中，我们只简单地实现将收到的消息外加处理该消息的线程信息打印出来：

polyglotActors/groovy/create/createActor.groovy

```
class HollywoodActor extends DefaultActor {
  def name

  public HollywoodActor(actorName) { name = actorName }

  void act() {
    loop {
      react { role ->
        println "$name playing the role $role"
        println "$name runs in ${Thread.currentThread()}"
      }
    }
  }
}
```

创建角色实例的方法与创建普通类实例的过程完全相同，唯一需要注意的是，在角色实例创建完之后要调用一下其 start() 函数。下面让我们先创建一对角色：

polyglotActors/groovy/create/createActor.groovy

```
depp = new HollywoodActor("Johnny Depp").start()
hanks = new HollywoodActor("Tom Hanks").start()
```

在创建动作完成之后，这两个角色就可以开始接收消息了。下面让我们通过 send() 函数分别向它们发送一个消息：

polyglotActors/groovy/create/createActor.groovy

```
depp.send("Wonka")
hanks.send("Lovell")
```

除了传统的 Java 风格的方法调用之外，我们也可以使用重载操作符 << 来发送消息。如果连这样也嫌麻烦的话，我们还可以直接把消息丢在角色对象后面，类似下面这样：

polyglotActors/groovy/create/createActor.groovy

```
depp << "Sparrow"
hanks "Gump"
```

默认情况下，角色都是运行在守护线程（daemon threads）里的，所以只要 main 函数完结则所有代码也将终止。但我们希望将所创建的角色的生命周期延长以便能够观察到消息被处理的过程，所以我们需要在 main() 函数结尾处加入延时操作或调用可以设置超时时间的 join() 函数。在使用后一种方法时，join() 函数会等待角色终止，由于我们没有调用角色的 terminate() 函数，所以角色不会主动消亡。但是由于我们在调用 join() 函数时设定了超时时间，所以当 join() 函数超时之后主线程将会跳出阻塞状态并结束。

polyglotActors/groovy/create/createActor.groovy

```
[depp, hanks]*.join(1, java.util.concurrent.TimeUnit.SECONDS)
```

为了执行上述示例代码，我们需要运行 groovy 并将 GPars 的 JAR 添加到 classpath 中：

```
roovy -classpath $GPARS_HOME/gpars-0.11.jar createActor.groovy
```

上例中的两个 HollywoodActor 的功能是在收到消息之后输出我们希望其扮演的荧幕人物的名字（这个名字就是消息的内容。——译者注）。虽然两个角色是并发运行的，但是每个角色每次只会处理一个消息。对于不同的角色实例，线程池可能会调度不同的线程来运行其方法以便对收到的消息作出回应。与此同时，并发程序的性质决定了并发操作的执行顺序是不确定的，所以每次运行这段示例代码时，我们从输出结果中观察到的函数调用顺序可能会有些许不同。巧合的是，在某次运行该示例时，我偶然观察到两个不同的角色实例在处理第一个消息的时候共享了相同的线程，并且其中一个角色在处理第二个消息的过程中切换到了其他线程。

```
ohnny Depp playing the role Wonka
ohnny Depp runs in Thread[Actor Thread 3,5,main]
om Hanks playing the role Lovell
om Hanks runs in Thread[Actor Thread 3,5,main]
om Hanks playing the role Gump
```

```
om Hanks runs in Thread[Actor Thread 2,5,main]
ohnny Depp playing the role Sparrow
ohnny Depp runs in Thread[Actor Thread 3,5,main]
```

如果想要令角色以无限循环（直至角色停止运行）的方式来处理消息，我们可以使用 loop() 函数，其用法如上例所示。而如果想要令角色的消息处理循环只运行有限次或达到某种条件后终止，则我们可以使用一些控制能力更强的 loop() 函数的变种。例如，我们可以使用 loop(count){} 来执行有限次消息处理逻辑；而如果我们想要指定循环结束条件，则可以使用 loop({->expression})。这个函数的效果是，只要 expression 的值为 true 则角色就会不断重复进入消息处理逻辑。此外，我们还可以提供一个可选的代码闭包（closure）并使其在角色被终止之后调用，具体实现方式在下面的例子中会详细介绍。

下面让我们用 Actors 类的 act() 函数来创建一个角色。在本例中，我们将会调用 act() 函数并将消息处理逻辑作为一个代码闭包传递给它。在上一个示例中，我们使用了只带有一个代码闭包的 loop() 函数，其中消息处理的主要逻辑是在 react() 函数中实现的。而在本例中，我们将使用带两个参数版本的 loop() 函数。其中第一个参数既可以是一个条件表达式也可以是一个数字。这里我们为其赋值为 3，以此来告知角色可以响应 3 个消息然后就结束。第二个参数为可选参数，其类型为一个代码段（以代码闭包的形式给出），当角色结束的时候这段代码将被执行。

polyglotActors/groovy/create/loop3.groovy

```
depp = Actors.actor {
  loop(3, { println "Done acting" }) {
    react { println "Johnny Depp playing the role $it" }
  }
}
```

由于 loop() 函数第一个参数为 3，所以角色只能接收并处理 3 个消息。下面我们尝试向其发送 4 个消息并观察其行为：

polyglotActors/groovy/create/loop3.groovy

```
depp << "Sparrow"
depp << "Wonka"
depp << "Scissorhands"
depp << "Cesar"

depp.join(1, java.util.concurrent.TimeUnit.SECONDS)
```

运行上述代码后我们发现，depp 角色只处理了前三个消息之后就结束了，而第四个消息则被其忽略。

```
Johnny Depp playing the role Sparrow
Johnny Depp playing the role Wonka
Johnny Depp playing the role Scissorhands
Done acting
```

通过上面的示例，我们了解了 GPars 在创建角色和消息循环的控制方面的灵活性。下面让我们来看看 GPars 在双向消息传递方面的功能。

9.1.2　发送并接收消息

除了提供单向的消息传递能力之外，GPars 还支持发送消息并接收响应的双向角色通信模式。我们可以通过两种方法来使用这一模式，即使用带有可选超时参数的阻塞式函数 sendAndWait()，或使用可以自定义回调逻辑的非阻塞函数 sendAndContinue()，该函数接受一个闭包作为其参数，当响应到达时调用该闭包。非阻塞版本的函数可以使我们很容易与多个角色进行消息交互并等待它们的响应。

为了能更方便地给消息发送方发送响应数据，Actor 类提供了一个很方便的 sender 属性。我们可以通过调用 sender 的 send() 函数来发送响应内容，具体使用方法如下所示：

polyglotActors/groovy/create/fortune.groovy

```
fortuneTeller = Actors.actor {
  loop {
    react { name ->
      sender.send("$name, you have a bright future")
    }
  }
}
```

在下面的示例代码中，fortuneTeller 角色的主要功能是输入一个待算命者的姓名，然后返回一个不靠谱的算命结果。从该角色的用户角度来看，我们实际上是调用了 sendAndWait() 方法将待算命者的姓名发送出去然后阻塞式地等待响应结果：

polyglotActors/groovy/create/fortune.groovy

```
message = fortuneTeller.sendAndWait("Joe", 1, TimeUnit.SECONDS)
println message
```

虽然 sendAndWait() 函数的用法很简单，但是如果我们想要一次性发送多个消息的话该函数就不能胜任了。为了解决这一问题，我们可以使用非阻塞的 sendAndContinue() 函数来一次性发送多条消息。但需要注意的是，在一次性发送了多条消息之后，我们需要使用像 CountDownLatch 这样的阻塞式工具来等待所有响应到达。每收到一个响应消息，我们就可以对线程闩的计数值减一。当线程闩的计数值降为 0 的时候，即意味着所有消息的响应均已到达，此时阻塞在线程闩上的主线程就可以继续运行了。如果在线程闩的计数值没到 0 的时候就超时了，则我们会根据 await() 函数的返回值来进行相应的处理：

polyglotActors/groovy/create/fortune.groovy

```
latch = new CountDownLatch(2)

fortuneTeller.sendAndContinue("Bob") { println it; latch.countDown() }
fortuneTeller.sendAndContinue("Fred") { println it; latch.countDown() }

println "Bob and Fred are keeping their fingers crossed"

if (!latch.await(1, TimeUnit.SECONDS))
  println "Fortune teller didn't respond before timeout!"
else
  println "Bob and Fred are happy campers"
```

上述示例的某次运行结果如下所示：

```
Joe, you have a bright future
Bob, you have a bright future
Bob and Fred are keeping their fingers crossed
Fred, you have a bright future
Bob and Fred are happy campers
```

从上面的示例中我们可以看到，Groovy 的表达能力以及代码闭包那灵活、方便的使用方法为 GPars 中的双向消息传递的实现提供了相当大的便利。在学习了处理 String 类型消息的方法之后，下面让我们一起来看看如何发送其他类型的消息。

9.1.3　处理离散消息

在真实的生产环境中，消息的种类并非仅局限于上面 GPars 示例中所使用的 String 类型。事实上，任何对象都可以被当做消息发送出去，但前提是我们需要确保这些对象都是不可变的。下面让我们创建一个可以处理多种消息类型的模拟股票交易的示例角色。在开始编写角色代码之前，我们需要预先定义好角色将会用到的一些消息类。我们可以使用 Groovy 定义的 Immutable 注解来确保所定义的消息都是不可变的。该注解不仅可以将所有字段都定义为 final，还可以用来修饰构造函数（或其他函数）。

polyglotActors/groovy/multimessage/stock1.groovy

```
@Immutable class LookUp {
  String ticker
}

@Immutable class Buy {
  String ticker
  int quantity
}
```

LookUp 消息用于表示查询某支股票价格，其中定义的 ticker 变量表示股票代码。而 Buy 消息中除了有代表股票代码的 ticker 变量之外，还定义了一个表示买进数量的 quantity 变量。我们的示例角色要能够针对这两种消息执行不同的处理动作。具体实现代码如下所示：

polyglotActors/groovy/multimessage/stock1.groovy

```
trader = Actors.actor {
  loop {
    react { message ->
        if(message instanceof Buy)
          println "Buying ${message.quantity} shares of ${message.ticker}"

        if(message instanceof LookUp)
          sender.send((int)(Math.random() * 1000))
    }
  }
}
```

每当 trader 角色收到一个消息之后，我们都会用运行时类型识别（RTTI）关键字 instanceof 来判断该消息是否为我们所期望收到的消息类型。对于可以识别的消息，我们将根据消息类型来采取相应的动作；而对于那些识别不了的消息，我们既可以像本例这样直接忽略掉，也可以抛出异常让调用方来处理。在本例中，如果收到的消息是一个 Buy 类型，我们就简单地将其中的内容打印出来；而如果收到的是一个 LookUp 消息，我们将返回一个随机数作为待查询股票的价格。

对于一个可以处理多种消息类型的角色而言，其使用方式与普通角色别无二致，具体代码如下所示：

polyglotActors/groovy/multimessage/stock1.groovy

```
trader.sendAndContinue(new LookUp("XYZ")) {
  println "Price of XYZ sock is $it"
}
trader << new Buy("XYZ", 200)
trader.join(1, java.util.concurrent.TimeUnit.SECONDS)
```

上述示例代码的输出结果如下所示：

```
Price of XYZ sock is 27
Buying 200 shares of XYZ
```

虽然该示例非常简单，但角色处理消息的方法还是有改进空间的。除了示例中用于检查类型的条件语句之外，我们还可以利用 Groovy 的类型系统来实现检查消息类型的分支判断逻辑。为了实现这一功能，我们需要将 DynamicDispatchActor 作为示例角色的基类。修改之后的代码如下所示：

polyglotActors/groovy/multimessage/stock2.groovy

```
class Trader extends DynamicDispatchActor {
  void onMessage(Buy message) {
    println "Buying ${message.quantity} shares of ${message.ticker}"
  }

  def onMessage(LookUp message) {
    sender.send((int)(Math.random() * 1000))
  }
}
```

在上面的代码中，我们令 Trader 继承 DynamicDispatchActor 并为每种角色处理的消息重载了对应的 onMessage() 函数。于是，基类 DynamicDispatchActor 将会负责检查收到的消息类型并将其分派给角色中合适的 onMessage() 函数。由于我们为角色定义了一个单独的类，所以请记得要调用 start() 来初始化角色的消息循环，具体代码如下所示：

polyglotActors/groovy/multimessage/stock2.groovy

```
trader = new Trader().start()
trader.sendAndContinue(new LookUp("XYZ")) {
  println "Price of XYZ sock is $it"
}
```

```
trader << new Buy("XYZ", 200)

trader.join(1, java.util.concurrent.TimeUnit.SECONDS)
```

这样一来，我们就无需再为处理离散消息而专门创建一个单独的处理类。相反地，我们只需将各种类型消息的处理函数封装到一个代码闭包中，并将其传递给 DynamicDispatchActor 的构造函数就可以了，具体代码如下所示：

polyglotActors/groovy/multimessage/stock3.groovy

```
trader = new DynamicDispatchActor({
  when { Buy message ->
    println "Buying ${message.quantity} shares of ${message.ticker}"
  }

  when { LookUp message ->
    sender.send((int)(Math.random() * 1000))
  }
}).start()
```

如上所见，我们作为参数传递给构造函数的那个代码闭包采用了一种很漂亮的语法形式，其中每类消息的处理代码都被封装到了一个 when 子句中。乍看起来，这段代码很像一组 if 条件表达式，但实际上二者是有显著区别的。首先，代码闭包里的实现方式无需使用低效的 RTTI。其次，每一个 when 子句都会在后台被转换成一个特定的消息处理函数。在使用了 when 子句的代码闭包中，无效或未处理的消息将会导致系统抛出异常；而在那些使用显式条件语句的实现中，我们需要专门写代码来处理这些问题。

9.1.4　使用 GPars

到目前为止，我们已经学习了如何创建并协作 GPars 角色，下面让我们运用这些知识来重写计算指定目录文件大小的程序。关于这个程序，我们曾在第 8.6 节中讨论过一个基于角色的设计，并在随后的章节中看到过 Scala 和 Java 的实现代码。下面让我们一起研究一下如何在 Groovy 中用 GPars 来实现同样的设计。

计算指定目录文件大小的程序在设计上需要两个角色，即 SizeCollector 和 FileProcessor。每个 FileProcessor 将会接收一个字符串类型的消息（待扫描目录名）并将计算后所得文件大小和给定目录的子目录列表作为消息发送给 SizeCollector。而 SizeCollector 则是一个主角色，即主控者。按照惯例我们还是先从消息类型的定义开始做起：

```
@Immutable class RequestAFile {}
@Immutable class FileSize { long size }
@Immutable class FileToProcess { String fileName }
```

如上所示，我们定义了 RequestAFile、FileSize 以及 FileToProcess 这 3 种消息类型，请注意它们都是用 Immutable 注解修饰的不可变类。

接下来就需要着手创建 FileProcessor 了。我们令 FileProcessor 继承 DefaultActor，以

此来为其赋予角色的各种能力。此外，该 FileProcessor 还需要向 SizeCollector 角色进行注册以便可以接收待扫描目录，而注册动作可以在 afterStart() 函数中完成。GPars 提供了与 Akka 的 preStart() 和 postStop() 函数功能相同的事件函数 afterStart() 和 afterStop()。其中，afterStart() 函数会在角色启动之后、第一个消息处理动作开始之前被执行。下面让我们一起来看看 FileProcessor 角色的实现代码：

polyglotActors/groovy/filesize/findFileSize.groovy

```
class FileProcessor extends DefaultActor {
  def sizeCollector

  public FileProcessor(theSizeCollector) { sizeCollector = theSizeCollector }

  void afterStart() { registerToGetFile() }

  void registerToGetFile() { sizeCollector << new RequestAFile() }
  void act()  {
    loop {
      react { message ->
        def file = new File(message.fileName)
        def size = 0
        if(!file.isDirectory())
          size = file.length()
        else {
          def children = file.listFiles()
          if (children != null) {
            children.each { child ->
              if(child.isFile())
                size += child.length()
              else
                sizeCollector << new FileToProcess(child.path)
            }
          }
        }
        sizeCollector << new FileSize(size)
        registerToGetFile()
      }
    }
  }
}
```

在与 FileProcessor 的交互过程中，SizeCollector 角色将会收到三种不同类型的消息。由于本例中只定义了 3 种确定的消息类型，所以如果我们是通过继承 DynamicDispatchActor 来实现这个角色的话，那么我们就可以写 3 个独立的函数而不是 switch 语句来处理消息了。与本设计方法的上一个实现版本相同，我们维护了一个待处理文件列表、一个空闲文件处理器列表、一个待扫描文件数量计数器以及最重要的已扫描文件大小总量。当 SizeCollector 收到 FileToProcess 消息或 RequestAFile 消息时，我们就会将待处理的文件分派给空闲的 FileProcessor 去处理。下面让我们一起看看 SizeCollector 的实现代码。

polyglotActors/groovy/filesize/findFileSize.groovy

```
class SizeCollector extends DynamicDispatchActor {
  def toProcessFileNames = []
  def idleFileProcessors = []
  def pendingNumberOfFilesToVisit = 0
  def totalSize = 0L
  final def start = System.nanoTime()

  def sendAFileToProcess() {
    if(toProcessFileNames && idleFileProcessors) {
      idleFileProcessors.first() <<
        new FileToProcess(toProcessFileNames.first())
      idleFileProcessors = idleFileProcessors.tail()
      toProcessFileNames = toProcessFileNames.tail()
    }
  }

  void onMessage(RequestAFile message) {
    idleFileProcessors.add(sender)
    sendAFileToProcess()
  }

  void onMessage(FileToProcess message) {
    toProcessFileNames.add(message.fileName)
    pendingNumberOfFilesToVisit += 1
    sendAFileToProcess()
  }

  void onMessage(FileSize message) {
    totalSize += message.size
    pendingNumberOfFilesToVisit -= 1
    if(pendingNumberOfFilesToVisit == 0) {
      def end = System.nanoTime()
      println "Total size is $totalSize"
      println "Time taken is ${(end - start)/1.0e9}"
      terminate()
    }
  }
}
```

当 pendingNumberOfFilesToVisit 变量的值降为 0 时，SizeCollector 角色将会打印出总文件大小以及处理过程的总耗时，然后它就会自我终结。现在到了将上述所有角色们串联起来的时候了，通常我们都是在类似下面这样的主代码中完成启动步骤：

polyglotActors/groovy/filesize/findFileSize.groovy

```
sizeCollector = new SizeCollector().start()
sizeCollector << new FileToProcess(args[0])

100.times { new FileProcessor(sizeCollector).start() }

sizeCollector.join(100, java.util.concurrent.TimeUnit.SECONDS)
```

在主代码的实现中，我们创建了一个 SizeCollector 和 100 个 FileProcessor。此外，我

们还为 SizeCollector 设定了超时时间，超过这个时间之后 SizeCollector 将会自我终结。用上述代码计算 /usr 目录的输出结果如下所示：

```
Total size is 3793911517
Time taken is 8.69052900
```

从上面的结果可以看出，Groovy-GPars 版本的运行结果与其他实现版本完全相同，并且其总耗时也与之前运行最快的实现版本非常接近。而在语法的简洁程度上，Groovy 版本也能够与 Scala 版本相媲美。总而言之，Groovy 简直太棒了。

9.1.5　数据流

数据流编程在很多年前曾风靡一时。很多公司都曾希望通过构建数据流处理器来实现应用程序的并发，从而摆脱编程结构的束缚。这种方法的优点是，只要将所需要的数据准备好，数据流处理器就可以帮助我们完成相关的计算任务。除此之外，与传统的冯·诺依曼风格的指令执行方式所不同的是，贯穿应用程序的数据流可以决定所要执行的指令集。尽管数据流处理器并未流行起来，但是数据流编程却再一次吸引了大家的目光。

在一个数据流程序中，在所需数据未就绪之前所有计算或任务都将阻塞，直至其全部数据可用为止。这种编程模型不需要同步或加锁。下面我们将会看到，只要能够保持任务本身的纯粹性（即所有任务都是幂等的、无任何副作用并且不会更改任何数据，换句话说，就是这些任务在风格上都是功能性的），数据流程序还是很易于编写的。虽然 Akka 也提供了数据流编程的 API，但是本节我们将选用更成熟的 GPars API 来进行数据流编程的实践。

假设我们接了一个从多个不同的网站抓取其页面内容并输出每个站点数据内容大小的任务。其中，打印或输出抓取内容大小的代码必须要顺序执行，因为该功能通常会涉及更新控制台或 GUI 控件，而这些东西通常都是单线程的。但与此不同的是，从各大网站抓取数据的任务却是可以并发执行的。并且一旦网站内容大小的数据可用之后，打印相关信息的代码就可以开始运行了。下面让我们先完成这部分代码然后再讨论其中的细节：

polyglotActors/groovy/dataflow/dataflowvariable/dataflow.groovy

```
def fetchContent(String url, DataFlowVariable content) {
  println("Requesting data from $url")
  content << url.toURL().text
  println("Set content from $url")
}

content1 = new DataFlowVariable()
content2 = new DataFlowVariable()

task { fetchContent("http://www.agiledeveloper.com", content1) }
task { fetchContent("http://pragprog.com", content2) }

println("Waiting for data to be set")
println("Size of content1 is ${content1.val.size()}")
println("Size of content2 is ${content2.val.size()}")
```

在 GPars 中，DataFlowVarable 类型的变量是一种只能写入一次，但可以多次读取的变量。如果需要的话，对于该类型变量的第一次读操作将被阻塞，直至数据可用为止。而接下来的读操作将会一直返回预写入的值（prewritten value）。在第一次写操作完成之后，该变量就会与写入值进行绑定，任何对已经进行过绑定的变量执行写操作的行为都将会导致系统抛出异常。由于所有读操作都会被阻塞直至变量与数据完成绑定，所以从一个 DataFlowVariable 类型的变量中读数据就没必要进行同步了。

在本例中，fetchContent() 函数的两个参数分别为：表示网站地址的字符串变量 url 和用于保存网站内容的 DataFlowVariable 类型的变量 content。通过优雅地组合调用 toURL() 函数和 text，我们获取到了给定网站的页面内容。随后我们又使用了 << 操作符将网页内容写入到 DataFlowVariable 类型的变量 content 中。

在定义完 fetchContent() 函数之后，我们接着又创建了两个数据流变量来负责保存待访问网站的内容。随后我们使用了 DataFlow 类的 task() 函数创建了两个并发运行的数据流任务。当抓取网站页面内容的两个任务开始执行以后，我们随即调用了 content1 变量的 val 属性。该操作将会使主线程阻塞直至数据可用为止。当所有数据准备就绪以后，程序最后两行代码就会将网站内容的大小打印出来，具体输出结果如下所示：

```
Waiting for data to be set
Requesting data from http://pragprog.com
Requesting data from http://www.agiledeveloper.com
Set content from http://www.agiledeveloper.com
Size of content1 is 2914
Set content from http://pragprog.com
Size of content2 is 13003
```

虽然实现的功能非常简单，但麻雀虽小五脏俱全。本例充分展示了一个数据流程序的本质特征，即指令的执行顺序完全取决于数据的可用性和数据流。

虽然有了 DataFlowVariable 之后我们就可以不用再关注同步问题了，但是这种在类型的变量生命周期内只能执行一次写入操作的特性使得其实用性受到了很大的限制。此外，GPars 还专门提供了 DataFlowQueue 用于处理流式数据（stream data）。DataFlowQueue 既可以同时服务于多个读者，也可以单独服务于关注数据流中所有数据值的单一读者，更详细的信息请参阅 GPars 的帮助文档。

在上例中，我们使用了 task 创建了两个异步任务。在任务规模小的时候这种做法是很方便的，但是当我们需要创建大量任务的时候，最好还是用一个线程池来对任务执行线程进行管理。针对这一需求，GPars 提供了相应的线程组类型 DefaultPGroup。有了这个工具类之后，我们就可以抛开 DataFlow 的静态函数 task()，而改用线程组实例对象的同名方法来创建任务了。

下面让我用数据流 API 来重写计算给定目录文件大小程序，具体代码如下所示：

polyglotActors/groovy/dataflow/filesize/fileSize.groovy

```
class FileSize {
  private final pendingFiles = new DataFlowQueue()
```

```
  private final sizes = new DataFlowQueue()
  private final group = new DefaultPGroup()

  def findSize(File file) {
    def size = 0
    if(!file.isDirectory())
      size = file.length()
    else {
      def children = file.listFiles()
      if (children != null) {
        children.each { child ->
          if(child.isFile())
            size += child.length()
          else {
            pendingFiles << 1
            group.task { findSize(child) }
          }
        }
      }
    }

    pendingFiles << -1
    sizes << size
  }

  def findTotalFileSize(File file) {
    pendingFiles << 1
    group.task { findSize(file) }

    int filesToVisit = 0
    long totalSize = 0
    while(true) {
      totalSize += sizes.val
      if(!(filesToVisit += (pendingFiles.val + pendingFiles.val))) break
    }

    totalSize
  }
}

start = System.nanoTime()
totalSize = new FileSize().findTotalFileSize(new File(args[0]))
println("Total size $totalSize")
println("Time taken ${(System.nanoTime() - start) / 1.0e9}")
```

在上面的代码中，我们创建了两个 DataFlowQueue 实例，其中一个主要用来标识是否还有未扫描的子目录，而另一个则用于接收待扫描目录及其子目录的大小。此外，为了将所有任务线程交由线程池来管理，我们还创建了一个 DefaultPGroup 类的实例对象。

findSize() 函数的主要功能是将给定目录下所有文件大小累加起来，并将累加结果通过 << 操作符放进 DataFlowQueue 类型的变量 sizes 中。对于在给定目录下所找到的每一个子目录，我们都会调用 group.task 创建一个新任务来负责该子目录的扫描。

在每个目录扫描任务开始之前，我们都会在 pendingFiles 中压入一个 1，并在任务结束之后再压入一个 –1。这一巧妙的小手段可以帮助我们在主代码中很方便地识别是否所有目

录都已经扫描完成。

在 findTotalFileSize() 函数中，我们创建了一个用于扫描给定文件 / 目录的任务。随后我们便不断接收从该任务以及该任务在扫描子目录时所创建的子任务的返回结果，并将其累加起来。而当给定目录下的所有子目录扫描完成之后，我们就跳出循环结束扫描任务。最后我们会将累计所得的文件大小返回给函数调用者。

本例的最后几行代码主要负责启动计算流程并记录上述所有函数的耗时。下面让我们运行该示例并观察其输出结果：

```
Total size is 3793911517
Time taken is 9.46729800
```

由于计算指定目录大小的程序本身非常适合用数据流方法来实现，所以本例的代码看上去给人感觉特别简单，并且其执行效率与其他解决方案相比也不遑多让。

9.2　在 Java 中使用 Akka 提供的角色实现

我们之前曾经讨论过，在 Java 中我们有很多基于角色的并发库可供选择，如 ActorFoundary、Actorom、Actors Guild、Akka、FunctionalJava、Kilim、Jetlang 等。在第 8 章中我们曾深入探讨了 Akka 库。虽然 Akka 使用 Scala 实现的，但为了能让广大 Java 开发人员方便地使用 Akka，其维护人员做了大量与 Java 的接口对接工作，使我们在 Java 中也能够使用 Akka 所提供的接口。关于 Akka 的详细信息，请参阅 Akka 的官方文档和本书中给出的示例。关于其他相关类库的信息请参阅其各自的文档。

9.3　在 JRuby 中使用 Akka 提供的 Actor 实现

Akka 库在 JRuby 中的使用方法与在 Java 中基本相同，二者主要区别在于我们用 JRuby 实现的代码可能比 Java 更简洁。在第 8 章我们曾经介绍了 Akka 的主要功能及其强大的 API，下面让我们一起来研究一下如何用 JRuby 实现第 8.6 节中计算给定目录文件大小的程序。请回忆一下之前的设计，我们需要一大堆消息类型，两个角色类型以及一些主代码来验证上述所有功能。下面让我们从消息类型的定义开始做起：

polyglotActors/jruby/FileSize.rb

```
class RequestAFile; end

class FileSize
  attr_reader :size
  def initialize(size)
    @size = size
  end
end

class FileToProcess
  attr_reader :file_name
  def initialize(file_name)
    @file_name = file_name
```

```
  end
end
```

上面这几个用于表示消息的类都是从之前 Java 示例中功能相同的类直接翻译过来的，这里就不再赘述。接下来要实现的 FileProcessor 角色稍微有一些麻烦，主要问题是当我们继承一个 Java 类时，JRuby 不允许我们更改构造函数签名（即构造函数名称及其参数），Akka 中基于角色的类 UntypedActor 只有一个无参构造函数，而我们的 FileProcessor 却需要一个类型为 SizeCollector 的构造函数参数。所以如果我们还是按之前的习惯来实现 initialize() 函数的话，程序会在初始化 FileProcessor 对象时抛出运行时错误。为了解决这一问题，我们需要在 initialize() 函数的第一行调用一下 super() 函数。除了上述改动之外，FileProcessor 剩余部分的实现代码都只是从 Java 到 JRuby 的简单直译：

polyglotActors/jruby/FileSize.rb

```
require 'java'
java_import java.lang.System
java_import 'akka.actor.ActorRegistry'
java_import 'akka.actor.Actors'
java_import 'akka.actor.UntypedActor'

class FileProcessor < UntypedActor
  attr_accessor :size_collector

  def initialize(size_collector)
    super()
    @size_collector = size_collector
  end

  def preStart
    register_to_get_file
  end

  def register_to_get_file
    @size_collector.send_one_way(RequestAFile.new, context)
  end

  def onReceive(fileToProcess)
    file = java.io.File.new(fileToProcess.file_name)
    size = 0
    if !file.isDirectory()
      size = file.length()
    else
      children = file.listFiles()
      if children != nil
        children.each do |child|
          if child.isFile()
            size += child.length()
          else
            @size_collector.send_one_way(FileToProcess.new(child.getPath()))
          end
        end
      end
    end
```

```
    @size_collector.send_one_way(FileSize.new(size))
    register_to_get_file
  end
end
```

下面让我们来实现 SizeCollector 角色。当我们尝试用 UntypedActor 的某个重载的 actor_of() 函数创建一个该角色实例的时候会发现程序会报该类缺少 create() 函数的错误。所以，我们还需要把这个工厂方法也补上。除此之外，剩余部分的实现代码都只是从 Java 到 JRuby 的简单直译：

polyglotActors/jruby/FileSize.rb

```
class SizeCollector < UntypedActor
  def self.create(*args)
    self.new(*args)
  end

  def initialize
    @to_process_file_names = []
    @file_processors = []
    @fetch_size_future = nil
    @pending_number_of_files_to_visit = 0
    @total_size = 0
    @start_time = System.nano_time
  end
  def send_a_file_to_process
    if !@to_process_file_names.empty? && !@file_processors.empty?
      @file_processors.first.send_one_way(
        FileToProcess.new(@to_process_file_names.first))
      @file_processors = @file_processors.drop(1)
      @to_process_file_names = @to_process_file_names.drop(1)
    end
  end

  def onReceive(message)
    case message
      when RequestAFile
        @file_processors << context.sender.get
        send_a_file_to_process

      when FileToProcess
        @to_process_file_names << message.file_name
        @pending_number_of_files_to_visit += 1
        send_a_file_to_process

      when FileSize
        @total_size += message.size
        @pending_number_of_files_to_visit -= 1
        if @pending_number_of_files_to_visit == 0
          end_time = System.nano_time()
          puts "Total size is #{@total_size}"
          puts "Time taken is #{(end_time - @start_time)/1.0e9}"
          Actors.registry().shutdownAll()
        end
      end
```

```
  end
end
```

最后，我们还需要补充一些用于检验上述两个角色功能的测试代码：

polyglotActors/jruby/FileSize.rb

```
size_collector = Actors.actor_of(SizeCollector).start()
size_collector.send_one_way(FileToProcess.new(ARGV[0]))

100.times do
  Actors.actor_of(lambda { FileProcessor.new(size_collector) }).start
end
```

Actors 类的 actor_of() 函数主要用于创建角色的实例对象，该函数主要有两种风格，即普通的 actor_of() 函数和 Java 风格的 actorOf() 函数，而每种风格都实现有一组重载函数。为了让 actor_of() 函数可以顺利地创建 SizeCollector 角色的实例，我们需要一个带有 create() 函数的工厂类，而 SizeCollector 中的静态函数 create() 可以用于此处。由于需要在 FileProcessor 的实例对象构造期间向其设定一个参数，所以在上述代码的结尾处，我们采用了可以接受一个内含创建角色实例功能的代码闭包作为参数的 actor_of() 函数的重载版本。由于 JRuby 允许需用 lambda 表达式来对代码闭包进行封装，所以在这里我们就利用了 JRuby 这一优点。除此之外，剩余部分的实现代码都只是从 Java 到 JRuby 的简单直译。

下面让我们运行 JRuby 版的示例程序来统计一下 /usr 目录的大小是多少：

```
Total size is 3793911517
Time taken is 9.69524
```

总体而言，虽然我们在实现 JRuby 版示例时踩了一些语言层面的坑，但是经过上面的学习，我们已经知道了如何绕开 JRuby 在对象继承方面的限制，在今后的工作中我们就可以写出更高效简洁的 Ruby 代码。

9.4　在 Scala 中使用角色

在 Scala 中，我们有很多基于角色的并发库可供选择。其中，scala.actor 库是与随 Scala 一起安装到电脑上的。而 Akka 库则可以提供更好的性能，并且该库更适合企业应用的开发（参见附录 2 中 Akka 的测试报告）。如果想要使用 Scala 角色库，请参阅 Scala 的帮助文档、《Programming in Scala》[OSV08] 或《Programming Scala》[Sub9]。如果想要使用 Akka，请参阅 Akka 的帮助文档以及我们在第 8 章中所提供的示例。

9.5　小结

我们可以用 JVM 上的任何语言来进行并发程序设计。而在本章中，我们主要学到了以下几点：

- Akka 的 Java API 可以很容易地在包括 JRuby 在内的诸多 JVM 语言中使用。
- GPars 是一个很强大的并发编程类库，其优点是既保留了 Groovy 语法优雅的特点又

可以达到 Java 程序的性能。

- 在消息处理时，使用模式匹配并非唯一途径。在 GPars 中，我们还可以根据消息类型来重载消息处理函数，并以此来达到自动消息分派的效果。
- 在将一些 API 进行混用的时候，我们可能会遇到一些语言层面上的集成问题。但请不要担心，所有问题我们都是有办法解决的。
- 在进行并发程序设计时，我们可以充分享受到现代 JVM 语言在其表达能力和语法简化方面所提供的福利。

不知不觉中，我们已经介绍了这么多内容！在下一章中，我们将用一些并发编程方面的建议作为全书的总结。

第五部分

后　记

第 10 章

并发编程之禅

我衷心希望你可以在并发之海的旅程中乘风破浪、尽兴而归。只要做事的方向没错，你就一定能安全地抵达彼岸，即能够充分用利多核处理器并发能力来实现高性能、响应迅速的应用程序。

并发编程其实无关乎我们所调用的方法或所传的参数，而更重要的是我们所采用的方法、努力的方向以及选用的库。所以我们须做出正确的选择才能设计出一个明确的并发解决方案。

编码的过程并非全都是火中取栗，而应该是不断取得成功并充满无穷乐趣的旅程。我们需要确定待使用的线程数和如何将应用程序拆分成多个并发的任务。而线程数则取决于应用程序在运行时可用的 CPU 核心数以及任务被阻塞的时间与其真正处理计算任务所耗时间的比重。将应用程序拆分成多个并发任务可不是一件容易的事。我们不但要了解应用程序的特性，而且还要学会如何分割才能得到负载均衡的任务。而正如我们在第 2 章中所讨论的那样，选择正确的线程数并保持各任务间的负载均衡将会有助于我们在应用程序运行时更好地利用多核 CPU 的强大能力。

10.1　慎重选择

在并发的世界里，选择合适的状态处理方法将对并发性和正确性起到决定性的影响。而这方面可选的方法有：共享可变性、隔离可变性以及完全不可变性。请谨慎考虑你的具体情况，并作出合适的选择。

毋庸置疑，共享可变性是我们极力想要避开的 " 破船 "（fail-boat）。当我们拒绝登上这

艘破船的那一刻，许多并发问题也会随之消失。虽然我们仍然需要一些努力和自律才能避免所有已经烂熟于胸的那些共享可变的编程风格，但不久你就会发现所有这些努力都是值得的。

登上隔离可变性和完全不可变性之船以后，你会发现抵达目的地的航程是如此风平浪静。这些方法都是着眼于从根本上解决问题。有了它们，我们无需再花费宝贵的时间和精力来避免竞争条件，因为这些方法完全消除了竞争条件，因此我们可以将注意力更多地集中在应用程序的逻辑上。

即使真的遇到由于状态变化所导致的问题，我们也可以通过第 3 章中介绍的隔离可变性和完全不可变方法来加以解决。

10.2　并发：程序员指南

升级到现代 JDK 附带的并发 API 是我们进行并发程序设计的第一步。请不要再使用第一版 Java 里附带的那些老旧的多线程 API 了。java.util.concurrent API 仿佛一缕清新的空气吹散了旧线程 API 的陈腐味道。正如我们在第 4 章中所看到的那样，通过这组新 API，我们可以轻松地通过线程池来管理线程、更好地控制同步操作并享受高性能的并发数据结构给我们带来的好处。很多类库现在都采用了这组现代的并发 API，而且一些专门的并发库（如 Akka 和 GPars）在这组并发 API 之上又加了一层抽象以方便我们使用。

如果你的工作职责包括“可变状态管理”，那么你就能体会到既要保证线程安全又得达到良好的并发性能是多么困难。我们曾在第 4 章和第 5 章里学习过一些解决这类问题的工具和技术：通过 ExecutorService 类，我们可以优雅地利用线程池来管理线程；通过并发数据结构，我们能够更有效地协作各线程 / 任务；通过 Lock 接口及相关工具，我们可以方便地执行细粒度的同步操作、改进并发性并对锁超时进行更好地控制。

除非遇到极个别的情况，否则请默认将类字段和本地变量置为 final。请尽量避免共享可变性。如果你在付出合理的努力之后仍无法找到避免可变性的方法，那么请考虑使用隔离可变性来代替共享可变性。事实证明，通过事先考虑好的设计来避免并发问题要比事后在代码的泥淖中挣扎要好得多。

请花些时间学习一下使用 STM 和基于角色模型的原型解决方案。

第 6 章和第 7 章中的示例有助于你创建原型，并获得一些比较有限但非常强大的处理共享可变性方面的经验。如果一个应用程序有读多、写冲突少但又存在明显的共享可变性这几个特点，那么我们就可以在软件事务内存（STM）中寻找慰藉。这里的窍门是，我们必须确保状态值都是不可变的，并且所有变更都是在托管引用上执行的。

通过角色，我们可以从根源上消除并发问题并对可变状态进行有效的隔离。由于可以有效地利用线程资源，所以基于角色的模型具有很高的可扩展性。作为活动对象，它们在内部对操作进行了排队，从而消除了并发所带来的问题。请花些时间用第 8 章和第 9 章中的示例来创建一些原型以便加深对这部分内容的理解。

如果一个应用程序无法做到如此清晰的隔离，但可以拆分成需要彼此协作的几个独立

的任务，那么请参阅我们在 8.9 节中曾经介绍过的将两种模型混合使用的方法。

本书所介绍的方法可能有些对你来说比较陌生，所以可能会增加一点学习曲线。但还记得你第一次学骑自行车时候的情景吗？当时我们需要花时间努力练习才能最终掌握骑车的技巧，而一旦你熟练掌握之后，这些技巧会帮助你达到比步行更快的行进速度。这些陌生的技术就好比骑自行车的技巧。在初始阶段，我们可能会感到有些别扭并需要花时间进行练习。而一旦真正掌握了这些技术之后，我们不但可以避免并发所带来的风险，还可以投入更多的时间和精力关注应用程序逻辑。

我们需要多花些精力好好学习才能掌握用角色和不可变消息传递的编程方法。而这种编程模式还涉及一些虽然本书未提及、但我们已经在 OO 编程中曾经用过的一些方法。所以我们不但要学会使用新的 API，还必须适应这种与以往经验有别的应用程序设计方法。因此，请花些时间磨炼一下你的设计技巧。

10.3　并发：架构师指南

并发性和性能可能是你在日常工作中最常需要解决的两个问题。幸运的是，现在已经有很多技术手段能够帮助我们通过改进应用程序的设计和架构来高效地解决这些问题。

对于并发问题来说最好的解决方法是从根本上消灭它而不是花很多时间解决它。要做到这一点其实很简单，只要消除可变状态就可以了，即我们要围绕不可变性或至少是隔离可变性来设计应用程序。

当我们试图远离共享可变性的时候，我们将面临两方面的挑战，即设计和性能。在设计方面，由于我们对共享可变性设计方法用得太熟了，以至于在转型期间需要付出更多努力才能达成目标。而在性能方面，持久化和并发数据结构的表现非常值得期待。

可能会对你产生影响的另一个领域是语言的选择。当开发者都对自己已经用顺手了的语言十分依恋的时候，换用其他语言通常会成为一个敏感的问题。如果我们自己都不认为那种语言更好，那么我们也无法说服团队其他成员和项目经理。对于并发解决方案的选择也是同样的道理。那些在团队中被广泛使用的方法既有可能是、也有可能不是最佳的解决方案。但如果我们自己都无法说服自己，那又如何去说服其他人呢？所以，只有我们认定了某个方案确实对于这个应用程序来说是最优的，然后我们才有可能去说服其他人。

所以，我们首先需要确认手头这些可选方法的实际效果如何。为了达到这一目标，我们需要花时间进行原型开发，并用最适合该应用程序的语言或并发模型来创建短小但能体现应用程序关键部分的示例程序。但此时请不要马上纠结于语言和并发模型的选择，因为这无助于你或你的团队明晰方法之间的差别。所以你还需要对这些方法进行逐个尝试，并进行客观的对比。

一旦确定好备选方案，我们会发现引领团队共同转向新方法其实是很容易的。团队在接触新方法的初期可能效率会有所降低，而一旦团队成员熟悉了该方法之后，他们将受益于新方法所带来的效率提升。他们之前花在管理共享可变状态或语言限制这些问题上的时间现在都可以用来解决实际的问题。本书所记录的那些用不同并发模型和不同语言实现的

示例不但可以帮助你逐一比较各个方法 / 模型之间的有效性，还能帮你判定你设计的方案是否真的优于你的团队当前正在使用的方法。

10.4　明智地进行选择

时至今日，JVM 上的并发编程已经有了长足的发展，我们可以不必局限与某一规定的模型，而是可以为我们所创建的每个应用程序自由选择对其最有效的模型。

对于并发编程而言，我们至少有三种选择：

- 编程道路上布满荆棘的“同步并受罪”模型。
- 软件事务内存（STM）
- 基于角色的并发模型

没有哪一个模型能够为所有的应用程序提供完美的解决方案，每个模型都有其优势和弱点。这就需要我们明智地挑选对手头待开发应用程序而言优点最多、代价最少的解决方案。

10.4.1　现代 JDK 附带的并发 API

由于现代 JDK 都附带了并发 API，所以每个装了 JDK 1.5 或更新版本的程序员都可以使用它。该 API 不但可以帮助我们方便地创建和管理线程池，调度并发任务，还提供了丰富的数据结构供我们使用。其种类从管理并发集合到负责线程之间数据交换阻塞队列，品种繁多、应有尽有。而新的 Lock 接口除了允许我们以组合的方式使用之外，还提供了一个可以超时的 API 以避免线程陷入死锁或饥饿的状态。

这种编程模型的主要缺陷不是 API 设计层面的问题，而是要看我们如何使用它。为了正确地使用这个 API，我们需要了解内存栅栏、在何处加锁、锁住多久等一大堆问题。正如我们在第 4 章和第 5 章曾经讨论过的那样，由于这种编程模型是复杂且易出错的，所以对于所有重要的应用程序而言，能正确运行可能都不是一件容易的事。

10.4.2　软件事务内存

STM 是针对共享可变性问题的一次大胆的尝试，其核心思路是将可变实体从不可变状态值中分离出来。STM 是经由 Clojure 成功实践之后才逐渐流行起来的。在 Clojure 语言中，状态是不可变的，而托管实体仅在 STM 的事务控制范围内是可变的。这种做法不仅可以使可变实体的行为具备可预测性和确定性，同时还为我们提供了一种显式的锁无关的编程方法。

然而 STM 存在两个主要缺陷。其一，如果项目所使用的编程语言不是 Clojure，那么必须格外小心，以确保所处理的状态除了托管实体都是不可变的。而这一问题对于 JVM 上的绝大多数语言都同样存在。其二，必须确保事务的实现代码都是幂等的且无任何副作用。也正是由于这个原因，STM 更适用于那些写冲突不频繁的应用场景。因为在冲突不频繁的情况下自动重做的策略是 OK 的，而在某些极端情况下该策略可能会导致严重的性能问题。总而言之，虽然只要我们能够小心谨慎地处理托管的共享可变性，STM 也能够实现理想的并发度。但是在没有底层语言支持的情况下，我们要达到这一目标的代价会比用 Clojure 实

现大很多。

10.4.3　基于角色的模型

通过将隔离可变性的思想发扬光大，该模型从根本上消除了同步问题。其主要特点为：每个角色均各自独立运行，相互之间依靠高效的异步消息进行通信，并保证同时到达的消息通过排队的方式逐个处理。

该模型的主要问题是，包括读操作在内的并发任务之间的所有通信都是通过消息来完成的。所以那些需要获得完全一致的状态值的读者们就不得不交叉地发请求以相互印证。使用多角色协作来设计应用程序的方法与设计一个 OO 应用程序的方法是有很大区别的。在基于角色的模型中，我们不但要确保消息是不可变的，并且为了使其能够独立完成关键任务，我们对角色的定义也都是相当的粗粒度的。此外，我们应当注意不要将角色们之间的交互设计得过于繁复，以避免角色们都将时间浪费在相互等待上而无法处理手头堆积的问题。

纵观本书，我们所选择的并发模型并未将我们强行捆绑到特定的语言上。模型的选择是由应用程序的特性、我们使用的设计方法以及团队进行自我调整以适应新模型的意愿共同决定的。请明智地进行选择并享受编程带来的乐趣。

祝你并发编程之旅愉快！

附录 1

Clojure agent

Clojure agent[⊖]代表了内存中的一个独立实体或位置。让我们请回忆一下，STM 的 ref 主要功能是管理针对多个实体的可协调同步变更，而 Agent 则允许对其托管的单个实体进行独立的异步变更。这些变更是由异步地作用在该位置上的函数或动作来表达的，多个独立的并发动作会一个接着一个地按顺序运行。当一个动作成功完成之时，被操作的 agent 即被更改为由该动作所返回的新状态。这个新状态将被用于后续的读操作或后续施加于该 agent 之上的其他动作。

在一个 agent 上执行动作的调用都是立即返回的，该动作会随后由 Clojure 托管的线程池里的某个线程来完成。如果某动作预计会比较耗 CPU，那么我们应该使用 send() 函数来执行该动作。而如果函数中可能会发生 IO 阻塞，那么我们最好用 send-off() 来代替 send() 函数，以避免当某些 agent 被阻塞时其他 agent 由于没可用线程而被饿死。

下面让我们通过创建一个好莱坞演员（actor）并让其扮演不同的荧幕人物（role）这样一个具体实例来进一步了解 agent。我们将会导演一场由多个不同演员参演不同人物的影片。其中，当多个动作并发运行的时候，我们将会看到发送至单个 agent 上的所有动作都是顺序进行的。

首先让我们创建一个为演员分配人物并将该人物放入这个演员曾经饰演过的人物列表中的函数。在下面的示例代码中，act() 函数接受两个参数：即 actor 和 role。其中 actor 是一个带有两个元素的元组：即演员的名字和其饰演过的荧幕人物列表。在定义完 actor 之后，我们会将 actor 的名字及其所饰演的荧幕人物打印出来。随后，为了能更清楚地展示针对相同 agent 的多个并发调用是顺序进行的，我们又插入了一小段模拟的延迟，并在最后返回 actor 的新状态、actor 的名字以及一份新的该 actor 所饰演过的荧幕人物列表。在这一动作中，我们不会更改任何东西。而 agent 则会把我们从该动作中所返回的值作为其新状态。

```
defn act [actor role]
 (let [[name roles] actor]
   (println name "playing role of" role)
   (. Thread sleep 2000)
   [name (conj roles role)]))
```

在上面的示例代码中，let 语句的作用是将 name 和 roles 与 actor 元组的第一和第二个元素分别进行绑定。而最后一行代码则返回了一个元组，其中元组的第一个元素是 name 的值，而第二个元素则是一个新的荧幕人物列表。新的荧幕人物列表不但含有之前列表中的

⊖ 详情请见 http://clojure.org/agents。

全部内容，而且还包括了我们当做参数传给 act() 函数的那个新荧幕人物。

上面我们已经学习了如何定义一个 agent 的函数，但到目前为止我们还没有见过任何 agent 的具体示例，所以现在就让我们开始讨论如何创建 agent。创建 agent 其实是非常简单的，不需要任何复杂或花哨的语法。在下面的示例代码中，我们定义了两个 agent，其中每个 agent 都代表了一个著名的好莱坞演员。所有这些 agent 都包含了一个带有演员名字和一个空列表的元组。代码中的“agent”即为用于定义 agent 的关键字。

```
(try
  (def depp (agent ["Johnny Depp" ()]))
  (def hanks (agent ["Tom Hanks" ()])))
```

至此，新定义 agent 们已经可以开始接收消息了，所以下面让我们向这两个 agent 发送一些消息来令它们扮演一些荧幕人物，具体代码如下所示。其中，send() 函数的第一个参数即为 agent，而紧随其后的则是待调用函数。如果待调用函数还需要额外参数的话（例如本例中的 role），则将其放到函数名的后面即可。

```
(send depp act "Wonka")
(send depp act "Sparrow")
(send hanks act "Lovell")
(send hanks act "Gump")
```

send() 是一个非阻塞函数。如果想要等待 agent 返回所有自本线程发送给它的所有消息的响应之后再继续运行，我们可以使用 await() 或与该函数功能类似的超时版本 await-for()。await-for() 的第一个参数是一个毫秒级的超时变量，后面可以跟多个的我们需要等待其响应的 agent。当所有 agent 完成全部调用或有超时发生时，无论哪种情况先发生，程序都会从阻塞状态中跳出。

```
(println "Let's wait for them to play")
(await-for 5000 depp hanks)
```

如果想要对 agent 解引用，我们只需在 agent 前面加上一个 @ 符号就可以了：

```
(println "Let's see the net effect")
(println (first @depp) "played" (second @depp))
(println (first @hanks) "played" (second @hanks))
```

还有一个细节请千万注意，那就是当 agent 运行在非守护线程中时，我们一定要在逻辑结尾处把所有的 agent 结束掉。我们可以将结束动作放到 finally 区块中，以确保即使碰到异常抛出时结束动作也能被执行。具体代码如下所示：

```
(finally (shutdown-agents)))
```

下面让我们把所有代码组合在一起，看看完整的代码序列：

polyglotActors/clojure/clojureagent.clj

```
(defn act [actor role]
  (let [[name roles] actor]
    (println name "playing role of" role)
    (. Thread sleep 2000)
    [name (conj roles role)]))
```

```
(try
  (def depp (agent ["Johnny Depp" ()]))
  (def hanks (agent ["Tom Hanks" ()]))

  (send depp act "Wonka")
  (send depp act "Sparrow")
  (send hanks act "Lovell")
  (send hanks act "Gump")

  (println "Let's wait for them to play")
  (await-for 5000 depp hanks)

  (println "Let's see the net effect")
  (println (first @depp) "played" (second @depp))
  (println (first @hanks) "played" (second @hanks))

  (finally (shutdown-agents)))
```

让我们运行上述示例代码并观察其输出结果。请注意，由于 agent 都是并发运行的并且其运行顺序也不确定，所以输出结果的序列可能每次运行都有所不同。

```
Johnny Depp playing role of Wonka
Tom Hanks playing role of Lovell
Let's wait for them to play
Johnny Depp playing role of Sparrow
Tom Hanks playing role of Gump
Let's see the net effect
Johnny Depp played (Sparrow Wonka)
Tom Hanks played (Gump Lovell)
```

从输出结果中我们可以很清楚地看到 send() 函数是非阻塞的。此外，每个 agent 都立刻处理了发送给它们的第一个消息，而第二个消息则由于 act 函数中的模拟延时而被延后处理。与此同时，主线程则在等待所有 agent 的消息处理结果。当所有消息都处理完之后，主线程会将所有 agent 里面演员及其所扮演过的荧幕人物都打印出来。

下面让我们简单对比一下 Clojure agent 和 Akka 角色二者有何异同。agent 本身并不含有任何特定的函数或专属的消息。我们可以调度任何函数在 agent 上运行。与此相反的是，Akka 角色则带有专门接收消息的 onReceive() 或 receive() 函数以及为其量身定制的消息，所以消息的发送方必须要知道哪些消息是角色所支持的。二者的相同之处在于，角色和 agent 都会将所有的调用请求序列化，即在任意时间只允许一个线程在其上运行。

默认情况下，agent 并不运行在任何事务中。请回忆一下，本书前面曾经提到，由于事务可能会被回滚并多次重做，所以事务性代码必须要没有任何副作用。那么如果我们想要在事务中执行某些动作或发消息给 agent 的话又将如何呢？ Clojure 的处理方式比较巧妙，它的做法是通过将发送给 agent 的所有消息都聚集在一个事务中，并仅当事务提交时才将所有消息分派出去。

下面就让我们通过在事务中发送消息给某 agent 的示例来观察的这一行为。在下面的代码中，sendMessageInTXN() 函数的主要功能是发送一个 act 消息给扮演某荧幕人物的那个演员。而该消息将会在一个由 dosync() 函数隐含的事务中被发送出去。如果 shouldFail 参

数被设为 true 的话，则事务将会被强制失败。而如果事务失败，则消息发送动作将被丢弃；否则该消息将被正常派发给 agent。在下面的示例中，我们创建了一个名为 depp 的 agent，并首先向起发送了一个扮演荧幕人物“Wonka”的事务性消息。而在随后的一个失败的事务中，我们向其发送了一个扮演“Sparrow”的消息。最后，我们会等待消息处理结果，然后将 depp 所扮演的所有荧幕人物打印出来。

```
(defn act [actor role]
  (let [[name roles] actor]
    (println name "playing role of" role)
    [name (conj roles role)]))

(defn sendMessageInTXN [actor role shouldFail]
  (try
    (dosync
      (println "sending message to act" role)
      (send actor act role)
      (if shouldFail (throw (new RuntimeException "Failing transaction"))))
    (catch Exception e (println "Failed transaction"))))

(def depp (agent ["Johnny Depp" ()]))

(try
  (sendMessageInTXN depp "Wonka" false)
  (sendMessageInTXN depp "Sparrow" true)
  (await-for 1000 depp)
  (println "Roles played" (second @depp))
  (finally (shutdown-agents)))
```

我们预期第一个消息会被正常发送给 agent，而第二个消息则被丢弃。具体的输出结果如下所示：

```
sending message to act Wonka
Johnny Depp playing role of Wonka
sending message to act Sparrow
Failed transaction
Roles played (Wonka)
```

由于发送给 agent 的动作都是在单独的线程中异步执行的并且发送消息的函数也都是非阻塞的，所以输出结果的顺序每次运行可能都会稍有不同。

当把发送消息的动作与事务进行绑定时，则发送给 agent 的消息都会被聚集起来，并且只有当事务成功完成之后消息才能被分发下去。如果将这一行为与我们在介绍 Akka transactor 那一节中曾提到过的将角色与事务混用的场景进行对比，就会发觉 Clojure 解决方案更加简洁，并且不存在我们之前在 8.10 节讨论 transactor 时所提到的那些缺点。

附录 2

一些网络资源

• Akka

http://akka.io

Akka 是由 Jonas Boner 开发的一套基于 Scala 的、可以在 JVM 上的多个语言中使用的可扩展并发库。该类库同时提供了 STM 和基于角色的并发功能。

• Clojure 数据结构

http://clojure.org/data_structures

这里主要讨论了 Clojure 中的定义的包括持久化数据结构在内的各种数据结构。

• Values and Change——Clojure's approach to Identity and State

http://clojure.org/state

Rich Hickey 在这个页面中讨论了实体与状态分离的 Clojure 模型，并分享了他如何在必要性和功能性之间进行权衡的一些经验方法。

• Designing and Building Parallel programs

http://www.mcs.anl.gov/~itf/dbpp/book-info.html

这里包含了 Ian Foster 同名著作的在线版本。

• 哲学家就餐问题

http://www.cs.utexas.edu/users/EWD/ewd03xx/EWD310.PDF

这里是 Edsger W. Dijkstra 于 1971 年在 Acta Informatia 发表的论文“Hierarchical Ordering of Sequential Processes”的原始内容。

• GPars

http://gpars.codehaus.org

GPars 的前身是 Vaclav Pech 所开发的 GParallelizer，该类库是一个基于 Groovy 的并发库，可以在 Java 和 Groovy 中使用。

• 完美哈希树

http://lamp.epfl.ch/papers/idealhashtrees.pdf

这里是 Phil Bagwell 所著的讨论持久化 Tries 的论文。

• Java 内存模型

http://java.sun.com/docs/books/jls/third_edition/html/memory.html

该页面包含了 Java 语言规范中讨论 Java 线程与 Java 内存模型部分的内容。

• JSR-133 Cookbook for Compiler Writers

http://g.oswego.edu/dl/jmm/cookbook.html

在这篇文章中，Doug Lea 解释了什么是内存栅栏。

• Lock 接口

http://download.oracle.com/javase/1.5.0/docs/api/java/util/concurrent/locks/Lock.html

这里是 Java5 引入的 Lock 接口的 Java 文档。

• 面向对象编程的意义

http://userpage.fu-berlin.de/~ram/pub/pub_jf47ht81Ht/doc_kay_oop_en

在一些技术交流的邮件中，Alan Kay 讨论了 OOP 的真正含义。

• Multiverse：软件事物内存的 Java 实现版

http://multiverse.codehaus.org/overview.html

Multiverse 是一个基于 Java 的、可以在多种 JVM 语言里使用的 STM 实现。其创始人是 Peter Veentijer。

• Ployglot Programming

http://memeagora.blogspot.com/2006/12/polyglot-programming.html

这里是 Neal Ford 的“Ployglot Programmers”版本。

• 测试驱动的多线程代码

http://www.agiledeveloper.com/presentations/TestDrivingMultiThreadedCode.zip

这里包含了我在题为《Test-Driving Multithreaded Code》演讲中所涉及的所有示例源码。

参考文献

[Arm07] Joe Armstrong. *Programming Erlang: Software for a Concurrent World.* The Pragmatic Bookshelf, Raleigh, NC and Dallas, TX, 2007.

[Bec96] Kent Beck. *Smalltalk Best Practice Patterns.* Prentice Hall, Englewood Cliffs, NJ, 1996.

[Blo08] Joshua Bloch. *Effective Java.* Addison-Wesley, Reading, MA, 2008.

[FH11] Michael Fogus and Chris Houser. *The Joy of Clojure.* Manning Publications Co., Greenwich, CT, 2011.

[Goe06] Brian Goetz. *Java Concurrency in Practice.* Addison-Wesley, Reading, MA, 2006.

[Hal09] Stuart Halloway. *Programming Clojure.* The Pragmatic Bookshelf, Raleigh, NC and Dallas, TX, 2009.

[Lea00] Doug Lea. *Concurrent Programming in Java: Design Principles and Patterns.* Addison-Wesley, Reading, MA, Second, 2000.

[Mey97] Bertrand Meyer. *Object-Oriented Software Construction.* Prentice Hall, Englewood Cliffs, NJ, Second, 1997.

[OSV08] Martin Odersky, Lex Spoon, and Bill Venners. *Programming in Scala.* Artima, Inc., Mountain View, CA, Second, 2008.

[Pag95] Meilir Page-Jones. *What Every Programmer Should Know About Object-Oriented Design.* Dorset House, New York, NY, USA, 1995.

[Sub09] Venkat Subramaniam. *Programming Scala: Tackle Multi-Core Complexity on the Java Virtual Machine.* The Pragmatic Bookshelf, Raleigh, NC and Dallas, TX, 2009.

[VWWA96] Robert Virding, Claes Wikstrom, Mike Williams, and Joe Armstrong. *Concurrent Programming in Erlang.* Prentice Hall, Englewood Cliffs, NJ, Second, 1996.