

JAVA

并发编程实践

JAVA CONCURRENCY IN PRACTICE

BRIAN GOETZ

WITH TIM PEIERLS, JOSHUA BLOCH,
JOSEPH BOWBEER, DAVID HOLMES,
AND DOUG LEA



JAVA
CONCURRENCY
IN PRACTICE



[美]	Brian Goetz	Tim Peierls	
	Joshua Bloch	Joseph Bowbeer	著
	David Holmes	Doug Lea	
	韩 锴	方 妙	译



电子工业出版社

PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
<http://www.phei.com.cn>

“过去的近三十年间，计算机性能一直由摩尔定律来推动；从今天起，它将由Amdahl定律推动。编写能够高效利用多处理器的代码，将会成为很大的挑战。在为今天以及未来的系统编写安全、可伸缩的Java程序时，你一定会发现《JAVA并发编程实践》提供的概念和技术很有用。”

—— **Doron Rajwan**, Research Scientist, Intel Corp

线程是Java平台的基石。随着多核处理器成为标准，欲构建高性能的应用程序，有效地利用并发将成为关键的步骤。Java SE 5和6是迈向并发应用开发的巨大进步，其中包括对Java虚拟机的改良，从而支持高性能、高可伸缩性的类和丰富的、崭新的并发构建块。在《JAVA并发编程实践》中，这些新特性的创造者们不仅解释了其工作原理和使用方式，同时还揭示其背后的动机和设计模式。

无论如何，开发、测试、调试多线程的程序仍然非常困难：常见的情形总是开发的并发程序看上去可以正常工作，但是在极端情况下就会失败，就生产环境而言这种情况是指高负载。《JAVA并发编程实践》以坚实的理论基础和翔实的实践技术，帮助读者构建可靠的、可伸缩的和可维护的并发应用程序。本书并不是简单地罗列出并发API和机制，相反，它提供了设计规则、模式和理想模型，使读者能够更容易地构建出既正确又高效的并发程序来。

本书主要包括：

- ✓ 并发与线程安全的基本概念
- ✓ 构建与组合线程安全类的技术
- ✓ 如何利用java.util.concurrent中的并发构建块
- ✓ 性能优化的是与非
- ✓ 测试并发程序
- ✓ 更多高级主题，诸如原子变量、非阻塞算法、Java存储模型等

本书作者系Java标准化组织（Java Community Process）JSR 166专家组（并发工具）的主要成员，同时他们还致力于其他多个JCP专家组织。**Brain Goetz**是一位拥有二十年行业经验的软件咨询师，发表过超过75篇关于Java开发的文章。**Tim Peierls**是现代多处理器的权威，在BoxPop.biz、唱片艺术和戏剧表演上也造诣颇深。**Joseph Bowbeer**是一位Java ME专家，他对并发编程的痴迷始于在Apollo计算机上编程的岁月。**David Holmes**是《The Java™ Programming Language》的合著者，目前就职于Sun Microsystems。**Joshua Bloch**是Google的首席Java架构师，《Effective Java》的作者、《Java Puzzlers》的合著者，他不像他的兄弟（his brother，Bloch与Neal主持的Java编程专栏里虚构的人物）那样编程，从来都不。**Doug Lea**是《Concurrent Programming in Java》的作者，SUNY Oswego大学计算机科学的教授。

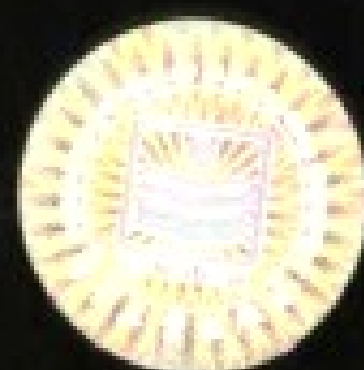
图书分类：程序设计>Java开发

网上订购：www.dearbook.com.cn
第二书店·第一服务



策划编辑：方舟
责任编辑：周筠 王继花

Addison-Wesley
Pearson Education



本书贴有激光防伪标志，凡没有防伪标志者，属盗版图书

ISBN 978-7-121-04316-1



9 787121 043161 >

定价：58.00元

JAVA 并发编程实践

JAVA CONCURRENCY IN PRACTICE

	Brian Goetz	Tim Peierls	
[美]	Joshua Bloch	Joseph Bowbeer	著
	David Holmes	Doug Lea	
	韩 锴	方 妙	译

電子工業出版社

Publishing House of Electronics Industry

北京 • BEIJING

内 容 简 介

随着多核处理器的普及,使用并发成为构建高性能应用程序的关键。Java 5 以及 6 在开发并发程序中取得了显著的进步,提高了 Java 虚拟机的性能以及并发类的可伸缩性,并加入了丰富的新并发构建块。在本书中,这些便利工具的创造者不仅解释了它们究竟如何工作、如何使用,还阐释了创造它们的原因,及其背后的设计模式。

本书既能够成为读者的理论支持,又可以作为构建可靠的、可伸缩的、可维护的并发程序的技术支持。本书并不仅仅提供并发 API 的清单及其机制,还提供了设计原则、模式和思想模型,使我们能够更好地构建正确的、性能良好的并发程序。本书适合于具有一定 Java 编程经验的程序员、希望了解 Java SE 5 以及 6 在线程技术上的改进和新特性的程序员,以及 Java 和并发编程的爱好者。

Authorized translation from the English language edition, entitled JAVA CONCURRENCY IN PRACTICE, FIRST EDITION, 0321349601 BY GOETZ ,BRIAN ; PEIERLS ,TIM ; BLOCH ,JOSHUA ; BOWBEER ,JOSEPH ; HOLMES , DAVID ; LEA ,DOUG , published by Pearson Education, Inc, publishing as Addison Wesley Professional, Copyright ©2006 Pearson Education, Inc.

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from Pearson Education, Inc.

CHINESE SIMPLIFIED language edition published by PEARSON EDUCATION ASIA LTD., and PUBLISHING HOUSE OF ELECTRONICS INDUSTRY Copyright ©2007.

本书简体中文版由电子工业出版社和 Pearson Education 培生教育出版亚洲有限公司合作出版。未经出版者预先书面许可,不得以任何方式复制或抄袭本书的任何部分。

本书简体中文版贴有 Pearson Education 培生教育出版集团激光防伪标签,无标签者不得销售。

版权贸易合同登记 图字: 01-2007-0564

图书在版编目(CIP)数据

JAVA 并发编程实践 / (美) 戈茨 (Goetz,B.) 等著; 韩锴, 方妙译. —北京: 电子工业出版社, 2007.6

书名原文: JAVA CONCURRENCY IN PRACTICE

ISBN 978-7-121-04316-1

I. J… II. ①戈…②韩…③方… III. JAVA 语言—程序设计 IV. TP312

中国版本图书馆 CIP 数据核字 (2007) 第 063572 号

策划编辑: 方 舟

责任编辑: 周 筠 王继花

印 刷: 北京智力达印刷有限公司

装 订: 北京中新伟业印刷有限公司

出版发行: 电子工业出版社

北京市海淀区万寿路 173 信箱 邮编 100036

开 本: 787×980 1/16 印张: 27.5 字数: 585 千字

印 次: 2007 年 6 月第 1 次印刷

定 价: 58.00 元

凡所购买电子工业出版社图书有缺损问题, 请向购买书店调换。若书店售缺, 请与本社发行部联系, 联系及邮购电话: (010) 88254888。

质量投诉请发邮件至 zlts@phei.com.cn, 盗版侵权举报请发邮件至 dbqq@phei.com.cn。服务热线: (010) 88258888。

对《JAVA 并发编程实践》的赞誉

Advance praise for JAVA CONCURRENCY IN PRACTICE

我曾经和一个梦幻般的团队一起工作，设计并实现了在 Java 5.0 和 Java 6 中为 Java 平台引入并发特性的功能，为此我感到无上的荣幸。如今，还是同一个团队，提供了目前为止对这些特性以及更一般的并发性的最佳阐释。并发不再仅仅是高级用户谈论的主题。每一名 Java 开发者都应该阅读本书。

—Martin Buchholz

JDK Concurrency Czar, Sun Microsystems

过去的近 30 年间，计算机性能一直由摩尔定律来推动；从今天起，它将由 Amdahl 定律推动。编写能够高效利用多处理器的代码，将会成为很大的挑战。在为今天以及未来的系统编写安全、可伸缩的 Java 程序时，你一定会发现《JAVA 并发编程实践》提供的概念和技术很有用。

—Doron Rajwan

Research Scientist, Intel Corp

如果你正在编写——或者设计、调试、维护、研究——多线程 Java 程序，这本书正是为你所写。如果曾经写过同步化的方法，但是并不确定其中的缘由，那么你应该给自己和你的用户都买一本来阅读，从头读到尾。

—Ted Neward

《Effective Enterprise Java》的作者

Brain 以不凡的清晰性，回答并解决了并发中的基本问题和复杂问题。对于使用线程和关心性能的人，这是一本必读书。

—Kirk Pepperdine

CTO, JavaPerformanceTuning.com

Java 并发编程实践

这本书言简意赅地覆盖了相当艰深和微妙的主题，使它成为一部理想的 Java 并发参考手册。每一页都充斥着程序员每日与之斗争的问题（和解决它的办法！）。今天，维系摩尔定律的不是更快的内核，而是更多的内核。有效地利用并发性在当今变得越来越重要，而这本书将向你展示如何做到这一切。

—Dr. Cliff Click

Senior Software Engineer, Azul Systems

我对并发有强烈的兴趣，与大多数程序员相比，我大概已经写了更多的线程死锁，也犯了更多的同步错误。Brian 的书在 Java 线程和并发主题中，是可读性最好的一本。通过精彩的循序渐进的方式，这本书讲解了这些困难的主题。我把这本书推荐给 Java Specialists' Newsletter 上的所有读者，因为这本书很有趣，也很有用，而且关系到今天 Java 开发者面对的问题。

—Dr. Heinz Kabutz

The Java Specialists' Newsletter

我所关注的职业是让简单的工作更简单，不过这本书却雄心勃勃并且富有成效地致力于简化一个复杂而关键的主题：并发。《JAVA 并发编程实践》以其革命性的讲述方式，明快而简洁的风格，以及及时地得以面世——它注定会成为一部极为重要的书。

—Bruce Tate

《Beyond Java》的作者

《JAVA 并发编程实践》是献给 Java 开发者的关于线程“所以然”的一份无价的作品。我发现阅读本书让人感到思维上的愉悦，部分原因是它对 Java 并发 API 的介绍相当精彩，但是更重要的原因在于，这本书以透彻的、可理解的方式，揭示了专家关于线程方面的知识，这在其他地方是很难找到的。

—Bill Venners

《Inside the Java Virtual Machine》的作者

To Jessica

作者为中文版作的序

能够为《JAVA 并发编程实践》中文版书写序言，我深感荣幸。

一年前，当这本书首次出版发行的时候，我们曾经写道“多核处理器正变得越来越便宜，足可以用于中端桌面系统了”。现在看来，多处理器系统正变得越来越便宜这一趋势还在继续，如果有什么不同，只是愈演愈烈了。几乎所有主要的计算机生产厂商都在初级笔记本电脑和台式机中加入了多核处理器特性，而在服务器级别的机器上，每个处理器的内核数量也多于去年的内核数量。事实上，寻找出单处理器的系统正变得越来越难了。

这些硬件的发展趋势使软件的开发者的面临着严峻的挑战。我们不能满足于仅仅在新的 CPU 上运行现有的程序，使它们跑得更快；如果我们想要发挥全新的处理器的能力，就必须编写程序来支持并发环境。在做这件事的过程中，会在架构、编程和测试方面遇到重大的挑战。而本书的目的就在于：为了让你能够应付这些挑战，书中提供了技术、模式和工具，可以用来分析并发编程并降低封装并发交互的复杂性。

现在，与以往任何时候相比，“并发”与每一位 Java 开发者都更加息息相关了。我们很高兴地看到，在中国有几十万的 Java 程序员将会成为本书的读者。

Brian Goetz

2007 年 2 月于 Williston, VT

Preface to the Chinese Edition

It is a great honor to be able to introduce the Chinese edition of Java Concurrency in Practice.

When this book was first published, one year ago, we wrote that "multicore processors are just becoming inexpensive enough to appear in midrange desktop systems." Now we see that this trend towards cheap multiprocessor systems has continued, and if anything, accelerated. Entry-level laptops and desktops from nearly every major computer manufacturer feature multi-core processors, and server-class machines are featuring even more cores per processor than they were last year. Indeed, it is getting hard to find single-processor systems.

These trends in hardware pose significant challenges for software developers. No longer can we just run our existing programs on new CPUs and have them run faster; our programs must be written to support concurrent environments if we want to take advantage of the power of newer processors. And doing so presents significant architectural, programming, and testing challenges. It is the goal of this book to respond to these challenges by offering techniques, patterns, and tools for analyzing concurrent programs and for encapsulating the complexity of concurrent interactions.

Concurrency is now more relevant than ever for Java developers. We are pleased that the audience for this book now includes the hundreds of thousands of Java programmers in China.

Brian Goetz

Williston, VT

February 2007

推荐序

在汗牛充栋的 Java 图书堆中，关于并发性的书籍却相当稀少，然而这本书的出现，将极大地弥补这一方面的空缺。即使并发性编程还没进入到您的 JAVA 日常开发当中来，您也应当花些时间来阅读这本重要的图书。该书是由 developerWorks 《JAVA 理论与实践》 <http://www.ibm.com/developerworks/cn/java/j-jtp/> 专刊的作者 Brian Goetz (<http://www.briangoetz.com/>) 执笔，他曾是 Quiotix 软件开发和咨询公司的首席顾问，现在是 Sun Microsystems 的高级工程师，并效力于多个 JCP 专家组。他作为专业的软件开发人员已经有 20 年的经验了，其在 Java 并发性领域的研究与贡献是有目共睹的。

这是一本目前在 Java 并发性领域研究的编程图书中最值得一读的力作。随着计算机技术的不断迅速发展，各种各样的编程模型也越来越多，越来越复杂化与多样化。虽然当前 CPU 主频在不断升高，但是 X86 架构的硬件已经成为瓶颈，这种架构的 CPU 主频最高为 4GHz，事实上目前 3.6GHz 主频的 CPU 已经接近顶峰，多线程编程模型不仅是目前提高应用性能的手段，更是下一代编程模型的核心思想。它的目的就是“最大限度地利用 CPU 资源”，当某一线程的处理不需要占用 CPU 而只需要 I/O 等其他资源时，就可以让需要占用 CPU 资源的其他线程有机会获得 CPU 资源。因此，就目前来说，多线程编程模型仍是计算机系统架构的最有效的编程模型。

Java 提供了语言级的多线程支持，所以在 Java 中使用多线程相对于在 C/C++ 当中使用多线程来说更加简单与快捷。除了 Brian Goetz 自己的研究、经验和热心读者的贡献之外，本书还吸取了一些并发性前沿人员的真知灼见，包括 Tim Peierls、Joshua Bloch、Joseph Bowbeer、David Holmes 和 Doug Lea。在该书中，Brian Goetz 从最基本的知识开始介绍，首先集中描述了在 Java 平台上创建线程应用程序以及同步对共享资源的访问时的细微之处；然后分析了 Java SE 5 提供的更高层次的线程执行构造，以及如何最好地把它们应用到现实世界中的不同场景，并整合了一些最佳实践和最新的研究主张；再就现实中的生存保证、性能、可伸缩性和可测试性的困难问题进行了分析，并把当前的最佳实践调查与相关的研究结果相结合，提供了一些可行的替代方案；最后介绍了一些在开发中可能适用的高级并发性技术，包括显式锁、定制同步器、原子变量与非阻塞同步，还介绍了低级的 Java 存储模型。同时，在全书贯穿了许多简洁的代码示例，用来演示问题和可行

的解决方案。

当从今天以应用程序为核心的开发平台转移到不远的未来支持多核处理器的操作系统和平台机制时，《JAVA 并发编程实践》代表了这个容易出错的领域当前最新的并发性实践和研究。相信这一本优秀的图书将是您案头的必备书籍，强烈建议您阅读并实践之。

俞黎敏

2007 年 3 月于上海



译者序

随着译稿最后一个字符的键入，我长舒一口气，目光不知不觉中落到了我的新笔记本电脑上，上面赫然贴着某处理器厂商新推出的多核处理器的 Logo。几个月来不断地与并发打交道，让我深深地感受到，我们已经进入了一个全新的时代——多核时代。

回想十几年前，提起多处理器或者多核系统，人们总会想到被摆放在实验室中发着轰鸣响声的机器，它们通常体型庞大，而且价格不菲，很多程序员对它们都是顶礼膜拜，又有多少人能为它们编写程序呢？在很多程序员心中，“并发”已经成为了“高深”的代名词。

造成这种现象的原因有两点。其一，在很多人心中，编写“并发”程序相对于编写“串行”程序更加困难。“并发专家”也因此被笼罩了一层神秘的面纱。其二，没有“需求”。多余的并发性在单处理器系统上显得有些力不从心。程序员的懒惰让他们很少有时间去编写没有人需要的代码。几年前，大量客户端代码都是运行在单处理器环境中的。在这种情况下，如果产生多个线程完成同一件事（比如压缩文件），效率只能随着线程生命周期管理与上下文切换而降低。

然而今天，多核已经成为不可阻挡的历史潮流了。计算机在人们的生产生活中发挥着越来越重要的作用。无论是科学计算、商业系统还是个人桌面应用，对性能的要求始终在增长。另一方面，流行 40 年之久的“摩尔定律”已渐渐显露疲态，电子电路的物理极限是无法突破的。在这样的环境下，“多核”应运而生。在一块芯片上嵌入更多的处理单元，相对于复杂昂贵的多核系统，是提升系统并行性与性能的简单、经济的途径，它更容易被普通大众所接受与使用。如果你正在准备或者刚刚购买新的个人电脑，你注意到它的处理器中包含了多少个内核了吗？

多核的影响是广泛且深入的。企业需要改变以往基于 CPU 数的计价方式。系统设计者需要重新审视多核产生的并行性带来的影响。一线的开发人员要学习新的开发思路、技巧和工具。并行的理念需要融入到系统的设计与实现的过程中。对程序员来说，多核带来的变化并不像时钟频率增加那样透明。以前我们已经对软件加入了很多的期望，比如可扩展性、健壮性、可伸缩性、可测试性，等等，今天我们还要考虑软件是否充分发挥了微处理器的全部性能。只有充分挖掘程序的并行性，才能让多核处理器物尽其用，才能让你的软件在今后内核不断增加的日子里，得以保持升级。

面对这些挑战，不禁令人有些担心。好在现代先进的运行时环境和编译器都对并行做好了大量的优化准备。更难得的是，像 Java 这样的高级语言提供了对并发的语言级支持。这个特性正是 Java 的伟大创新之一。尤其是 Java 5 的发布，更是 Java 并发开发的质的飞跃。若干年前必须依靠本机代码才能实现的功能，如今可以使用纯 Java 代码实现；若干年前只有并发专家才能开发的程序，如今你也可以编写出来；若干年前极易产生 bug 的功能模块，如今可以使用 Java 核心类轻松实现。而 Brian 的这本书就是帮你做到这点的最佳指南。Brian 以其特有的师者风范，以精准而不失诙谐的文字，将 Java 并发开发的奥秘抽丝剥茧，让读者在享受阅读乐趣的同时，获得广博而深入的并发知识和技能。这些足以帮助开发者消除（减轻）多核时代的“危机”。

如果是孤军奋战，很难想象可以完成这部著作的翻译，期间我们得到了太多的帮助，即使挂一漏万，也要尽力列在这里：

首先要感谢本书的原作者 Brian Goetz 先生。如果没有他的付出，就谈不上这个译本的出现。还要感谢他不厌其烦地回答了我们大量的问题。Brian 每次平易近人和极为认真的回复，都让我们在提高译本质量的过程中受益匪浅。

法律专业出身的原莹小姐，感谢你以极大的勇气，踏入这个对你来说并不熟悉的领域。得益于你的辛劳，这本书的文字才会摆脱一些计算机科学的冰冷。

感谢博文视点的各位老师和编辑，尤其是晓菲。你们的敬业与认真，深深地感动了我。

感谢中国人民银行软件开发中心的各位同事，是你们的宽容，让我有更多的时间投入到这本书的翻译工作中。

最后，也是最重要的，感谢这本书的另一位合译者——方妙小姐。你的专业知识和文字功底让我惊叹。与你的合作是一段令人愉快且难忘的经历。

.....

计算机领域从来都不乏变革，每一次都带来新的机遇与挑战。“多核”无疑是一场令人激动的变革。迎接技术变革与挑战，是软件开发者的空气和水。很高兴你已经读到了这里，那么请不要停下来，继续享受挑战的乐趣，继续面对这场变革，前进吧！

韩锴

2007 年 3 月于北京



译者简介

韩锴，毕业于北京工业大学软件工程专业；目前就职于中国人民银行软件开发部，负责战略发展部的核心框架开发，对软件开发有其独到的理解和认识。目前主要研究方向为：敏捷软件过程方法论、Eclipse 平台及其相关技术、Java 下的并发编程，等等。联系方式为：isaachanstar@gmail.com。

方妙，毕业于北京工业大学软件工程专业；目前就职于 Ivar Jacobson 软件咨询公司中国分公司，担任软件设计工程师，从事新一代过程构建与 Web 2.0 开发，对 Java 技术、轻量级过程方法论与架构设计有浓厚的兴趣。联系方式为：miaofang@gmail.com。

联系博文视点

您可以通过如下方式与本书的出版方取得联系。

读者信箱: *sheguang@broadview.com.cn*

投稿信箱: *broadvieweditor@gmail.com*

北京博文视点资讯有限公司（武汉分部）

湖北省 武汉市 洪山区 吴家湾 邮科院路特 1 号 湖北信息产业科技大厦 1402 室

邮政编码: 430074

电话: (027)87690813 传真: (027)87690813 转 817

若您希望参加博文视点的有奖读者调查, 或对写作和翻译感兴趣, 欢迎您访问:

<http://bv.csdn.net>

关于本书的勘误、资源下载及博文视点的最新书讯, 欢迎您访问博文视点官方博客:

<http://blog.csdn.net/bvbook>

序

Preface

写作本书时，出于桌面系统的迫切需求，多核处理器正在变得越来越便宜。与此不协调的是，很多开发团队还没有注意到，在他们的项目中，出现了越来越多的关于线程的错误报告。在 NetBeans 开发者站点上最近的一次通告中，一位核心维护者注意到，为了修复某个类的一个线程相关的问题，已经被打了 14 次补丁。Dion Almaer，前 TheServerSide 的编辑（经过一次痛苦的调试过程，最终发现是一个线程的 bug 之后），最近在 Blog 上写道，大多数 Java 程序都充斥着并发 bug，它们仅仅是“碰巧”可以工作。

的确，由于并发性的 bug 不会以可预见的方式自己“蹦”出来，因此多线程程序的开发、测试和调试都会变得极端困难。bug 浮出水面的时刻，通常可能是最坏的时候——对应于生产环境，就是指在高负载的时候。

使用 Java 开发并发程序所要面临的挑战之一，是要面对平台提供的各种并发特性之间的不匹配，还有就是程序员在他们的程序中应该如何思考并发性。语言提供了一些低层机制，比如同步和条件等待，但是这些机制在实现应用级的协议与策略时才是必须的。不顾这些策略的约束，很容易创建一个程序，它在编译和运行时看上去一切正常，不过这其中却存在隐患。很多并发方面相当不错的书都没能达到预期的目标，它们过分地关注于低层的机制和 API，而不是设计层面的策略和模式。

Java 5.0 是在使用 Java 开发并发应用程序的进程中，迈出的巨大一步。它提供了新的高层组件以及更多的低层机制，这些将使得一名新手更容易像专家那样去构建并发应用程序。本书的作者都是 JCP 专家组的主要成员，正是这个专家组创建了这些新工具；除了去描述新工具的行为和特性，我们还向您展示了它们低层的设计模式，预期的使用场景以及将它们纳入平台核心库的动机。

我们的目标是给读者一些设计法则和理念模型，让读者在使用 Java 构建正确、高效的并发类和应用程序时，变得更容易、更有趣。

我们希望你在阅读《JAVA 并发编程实践》的过程中能够获得愉悦感。

Brian Goetz

2006 年 3 月于 Williston, VT

如何使用本书

Java 低层机制与设计层必要的策略之间存在着抽象的不匹配。为了解决这个问题，我们呈现了一个经过简化的规则集，用来编写并发程序。专家们看到这些规则会说：“嗨，这可不是完整的规则，类 C 即使破坏了规则 R，它仍然可以是线程安全的。”尽管打破我们的规则，也可能写出正确的并发程序，不过这样做需要对 Java 存储模型的低层细节有着深入的理解。我们的愿望是：开发者**不用**熟悉这些细节，就有能力编写正确的并发程序。只要坚持遵守我们简单的规则，就能编写出正确的、可维护的并发程序。

我们假设读者已经具备了对 Java 基本并发机制的一些了解。《JAVA 并发编程实践》不是并发的入门指南——关于这个，可以参看任何正统的介绍性的大部头书籍，比如《Java 编程语言》（Arnold et al., 2005）。本书也不是关于并发的百科全书似的参考手册——关于这个，可以参看《JAVA 并发编程（Lea, 2000）》。对本书更恰当的描述是，它提供了实际的设计规则，可以协助开发者，他们正在处于创建安全的、高效的并发类这一艰难的过程中。在适当的地方，我们穿插引用了下列图书的章节：《The Java Programming Language》、《Concurrent Programming in Java》、《The Java Language Specification（Gosling et al., 2005）》以及《Effective Java（Bloch, 2001）》，并使用[JPL n.m]、[CPJ n.m]、[JLS n.m]和[EJ Item n]来表示它们。

结束“介绍”（第 1 章）之后，本书分为 4 部分：

基础。第 1 部分（第 2~5 章）关注于同步和线程安全的基本概念，以及如何使用类库提供的构建块组合线程安全类。第 110 页上，有一个“并发诀窍清单”总结了出现在第 1 部分中最重要的规则。

第 2 章（线程安全性）与第 3 章（共享对象）构成了全书的基础。几乎所有用来避免并发危险、创建线程安全类以及验证线程安全的规则都包括在这里。轻“理论”，重“实践”的读者可能会禁不住诱惑跳过这部分，而直接进入第 2 部分，但是在开始编写任何并发代码之前，一定要确保回过头来阅读这两章！第 4 章所涵盖的技术，用于把线程安全类组合到更大的线程安全类中。第 5 章（构建块）涵盖了平台核心库提供的并发构建块——线程安全的容器和同步工具（synchronizer）。

构建并发应用程序。第 2 部分（第 6~9 章）描述了如何利用线程提高并发应用程序的吞吐量或响应性。第 6 章（任务执行）讲述如何识别可并行执行的任务，并在任务执行框架内部执行它们。第 7 章讲到的技术可以让任务和线程在正常终止之前妥善地终止；区分并发应用程序是健壮的，还是仅仅可以将就工作的，有众多的因素，“程序如何处理取消与关闭”就是其中之一。第 8 章（应用线程池）关注了一些任务执行框架中更为高级的特性。

第 9 章（GUI 应用程序）关注了用来提高单线程化子系统响应性的技术。

活跃度、性能和测试。第 3 部分（第 10~12 章）涉及并发程序自身。要确保并发程序执行了你所希望它做的事情，而且性能是可以接受的。第 10 章（避免活跃度危险）描述了如何避免活跃度失败，活跃度失败会阻止程序继续向前执行。第 11 章（性能和可伸缩性）涵盖的技术用来提高并发代码的性能和可伸缩性。第 12 章（测试并发程序）涵盖的技术用来测试并发代码的正确性和性能。

高级主题。第 4 部分（第 13~16 章）涵盖的主题可能只会引起资深程序员的兴趣：它们是显式锁、原子变量、非阻塞算法和开发自定义的 synchronizer。

代码示例

尽管书中的很多通用概念适用于 Java 5.0 之前版本，甚至是非 Java 环境，不过大多数示例代码（以及关于 Java 存储模型的每一句话）都假定是以 Java 5.0 或更新的 JDK 为基础的。有些代码示例还会用到 Java 6 中添加到类库中的特性。

代码示例已经被裁减，以降低它们的尺寸和突出相关的部分。完整版本的代码示例、辅助示例和勘误表，可以从本书的网站 <http://www.javaconcurrencyinpractice.com> 上获得。

代码示例分为三类：“好”示例，“一般”示例和“坏”示例。“好”示例阐释的技术应该被效仿。“坏”示例阐释的技术绝对不应该被效仿，而且还会用一个“Mr. Yuk”¹ 的图标清楚地表明这是“有害”的代码（参见清单 1）。“一般”示例阐释的技术不一定是错的，但却是脆弱的、有风险的或者执行效果差的，而且会用一个“Mr. CouldBeHappier”图标标识出来，如同清单 2。

清单 1 糟糕的列表排序方法（不要这样做）

```
public <T extends Comparable<? super T>> void sort(List<T> list) {  
    // 永远不要返回错误的答案  
    System.exit(0);  
}
```



有些读者会质疑“坏”示例在本书中的角色；毕竟，一本书应该展现如何做正确的事，而不是错误的事。“坏”示例有两个目的。它们揭示了常见的缺陷，更重要的是它们示范了如何分析程序的线程安全性——完成此事的最佳办法就是观察威胁线程安全的各种方式。

¹ Mr. Yuk 是 the Children's Hospital of Pittsburgh 的注册商标，以授权在本书中使用。

清单 2 缺少优化的列表排序方法

```
public <T extends Comparable<? super T>> void sort(List<T> list) {  
    for (int i=0; i<1000000; i++)  
        doNothing();  
    Collections.sort(list);  
}
```



致谢

java.util.concurrent 包的开发过程催生了本书。这个包是由 Java Community Process JSR 166 创建的，后被加入到 Java 5.0 中。还有很多人为 JSR 166 作出过贡献；我们尤其要感谢 Martin Buchholz，他完成了所有把代码植入 JDK 的相关工作；还要感谢 concurrency-interest 邮件清单的所有读者，他们为 API 的草案提供了他们的建议和反馈。

本书能够大幅度地完善，得益于大量的建议和帮助，这些帮助来自多方面的人员。我们要感谢 Dion Almaer、Tracy Bialik、Cindy Bloch、Martin Buchholz、Paul Christmann、Cliff Click、Stuart Halloway、David Hovemeyer、Jason Hunter、Michael Hunter、Jeremy Hylton、Heinz Kabutz、Robert Kuhar、Ramnivas Laddad、Jared Levy、Nicole Lewis、Victor Luchangco、Jeremy Manson、Paul Martin、Berni Massingill、Michael Maurer、Ted Neward、Kirk Pepperdine、Bill Pugh、Sam Pullara、Russ Rufer、Bill Scherer、Jeffrey Siegal、Bruce Tate、Gil Tene、Paul Tyma，以及硅谷模式小组的成员，他们通过很多饶有趣味的技术交流，提供了指南，提出了建议，这些对本书能够做到更好颇有帮助。

我们格外感激 Cliff Biffle、Barry Hayes、Dawid Kurzyniec、Angelika Langer、Doron Rajwan 和 Bill Venners，他们审阅了全部手稿，还关注了令人烦恼的细节，寻找代码示例中的 bug，并且提出了大量令本书得以改进的建议。

我们感谢 Katrina Avery，他出色地完成了 copy-editing 的工作；Rosemary Simpson 在极端的时间压力下，还制作了索引。我们感谢 Ami Dewar 绘制的插图。

感谢 Addison-Wesley 的全体组员，他们让这本书得以问世。Ann Sellers 让项目得以运行；Greg Doench 让项目的进度有条不紊；Elizabeth Ryan 一直领导着本书写作的进程。

我们还想要感谢成千上万的软件工程师，他们开发了编写本书所用到的软件，间接地为本书作出了贡献。这些软件包括 TEX、LATEX、Adobe Acrobat、pic、grap、Adobe Illustrator、Perl、Apache Ant、IntelliJ IDEA、GNU emacs、Subversion、TortoiseSVN，当然，还有 Java 平台与类库。

目录

代码清单.....	I
序.....	VII
第 1 章 介绍.....	1
1.1 并发的（非常）简短历史	1
1.2 线程的优点	3
1.3 线程的风险	5
1.4 线程无处不在	9
第 1 部分 基础.....	13
第 2 章 线程安全	15
2.1 什么是线程安全性	17
2.2 原子性	19
2.3 锁	23
2.4 用锁来保护状态	27
2.5 活跃度与性能	29
第 3 章 共享对象.....	33
3.1 可见性	33
3.2 发布和逸出	39
3.3 线程封闭	42
3.4 不可变性	46
3.5 安全发布	49
第 4 章 组合对象.....	55
4.1 设计线程安全的类	55
4.2 实例限制	58
4.3 委托线程安全	62
4.4 向已有的线程安全类添加功能	71

4.5 同步策略的文档化	74
第 5 章 构建块	79
5.1 同步容器	79
5.2 并发容器	84
5.3 阻塞队列和生产者-消费者模式	87
5.4 阻塞和可中断的方法	92
5.5 Synchronizer	94
5.6 为计算结果建立高效、可伸缩的高速缓存	101
第 2 部分 构建并发应用程序	111
第 6 章 任务执行	113
6.1 在线程中执行任务	113
6.2 Executor 框架	117
6.3 寻找可强化的并行性	123
第 7 章 取消和关闭	135
7.1 任务取消	135
7.2 停止基于线程的服务	150
7.3 处理反常的线程终止	161
7.4 JVM 关闭	164
第 8 章 应用线程池	167
8.1 任务与执行策略间的隐性耦合	167
8.2 定制线程池的大小	170
8.3 配置 ThreadPoolExecutor	171
8.4 扩展 ThreadPoolExecutor	179
8.5 并行递归算法	181
第 9 章 GUI 应用程序	189
9.1 为什么 GUI 是单线程化的	189
9.2 短期的 GUI 任务	192
9.3 耗时 GUI 任务	195
9.4 共享数据模型	198
9.5 其他形式的单线程子系统	202

第 3 部分 活跃度, 性能和测试	203
第 10 章 避免活跃度危险	205
10.1 死锁	205
10.2 避免和诊断死锁	215
10.3 其他的活跃度危险	218
第 11 章 性能和可伸缩性	221
11.1 性能的思考	221
11.2 Amdahl 定律	225
11.3 线程引入的开销	229
11.4 减少锁的竞争	232
11.5 示例: 比较 Map 的性能	242
11.6 减少上下文切换的开销	243
第 12 章 测试并发程序	247
12.1 测试正确性	248
12.2 测试性能	260
12.3 避免性能测试的陷阱	266
12.4 测试方法补遗	270
第 4 部分 高级主题	275
第 13 章 显式锁	277
13.1 Lock 和 ReentrantLock	277
13.2 对性能的考量	282
13.3 公平性	283
13.4 在 synchronized 和 ReentrantLock 之间进行选择	285
13.5 读-写锁	286
第 14 章 构建自定义的同步工具	291
14.1 管理状态依赖性	291
14.2 使用条件队列	298
14.3 显式的 Condition 对象	306
14.4 剖析 Synchronizer	308
14.5 AbstractQueuedSynchronizer	311
14.6 java.util.concurrent 的 Synchronizer 类中的 AQS	314

第 15 章 原子变量与非阻塞同步机制	319
15.1 锁的劣势	319
15.2 硬件对并发的支持	321
15.3 原子变量类	324
15.4 非阻塞算法	329
第 16 章 Java 存储模型	337
16.1 什么是存储模型，要它何用	337
16.2 发布	344
16.3 初始化安全性	349
附录 A 同步 Annotation	353
A.1 类 Annotation	353
A.2 域 Annotation 和方法 Annotation	353
参考文献	355
索引	359

代码清单

清单 1	糟糕的列表排序方法（不要这样做）	IX
清单 2	缺少优化的列表排序方法	X
清单 1.1	非线程安全的序列生成器	6
清单 1.2	线程安全的序列生成器	7
清单 2.1	一个无状态的 Servlet	18
清单 2.2	Servlet 计算请求数量而没有必要的同步（不要这样做）	19
清单 2.3	惰性初始化中存在竞争条件（不要这样做）	21
清单 2.4	Servlet 使用 AtomicLong 统计请求数	23
清单 2.5	没有正确原子化的 Servlet 试图缓存它的最新结果（不要这样做）	24
清单 2.6	缓存了最新结果的 servlet，但响应性令人无法接受（不要这样做）	26
清单 2.7	如果内部锁不是可重入的，代码将死锁	27
清单 2.8	缓存最新请求和结果的 servlet	31
清单 3.1	在没有同步的情况下共享变量（不要这样做）	34
清单 3.2	非线程安全的可变整数访问器	36
清单 3.3	线程安全的可变整数访问器	36
清单 3.4	数绵羊	39
清单 3.5	发布对象	40
清单 3.6	允许内部可变的数据逸出（不要这样做）	40
清单 3.7	隐式地允许 this 引用逸出（不要这样做）	41
清单 3.8	使用工厂方法防止 this 引用在构造期间逸出	42
清单 3.9	本地的基本类型和引用类型的变量的线程限制	44
清单 3.10	使用 Thread Local 确保线程封闭性	45
清单 3.11	构造于底层可变对象之上的不可变类	47
清单 3.12	在不可变的容器中缓存数字和它的因数	49
清单 3.13	使用到不可变容器对象的 volatile 类型引用，缓存最新的结果	50
清单 3.14	在没有适当的同步的情况下就发布对象（不要这样做）	50
清单 3.15	如果 Holder 没有被正确发布，它将处于失败的风险中	51
清单 4.1	使用 Java 监视器模式的简单线程安全计数器	56
清单 4.2	使用限制确保线程安全	59
清单 4.3	私有锁保护状态	61

清单 4.4	基于监视器的机动车追踪器实现	63
清单 4.5	类似于 java.awt.Point 的可变 Point.....	64
清单 4.6	DelegatingVehicleTracker 使用不可变的 Point 类.....	64
清单 4.7	将线程安全委托到 ConcurrentHashMap	65
清单 4.8	返回 location 集的静态拷贝, 而非“现场 (live)”的	66
清单 4.9	委托线程安全到多个底层的状态变量	66
清单 4.10	NumberRange 类没有完整地保护它的不变约束 (不要这样做)	67
清单 4.11	可变的线程安全 Point 类.....	69
清单 4.12	安全发布底层状态的机动车追踪器	70
清单 4.13	扩展的 Vector 包含一个“缺少即加入”方法.....	72
清单 4.14	非线程安全的“缺少即加入”实现 (不要这样做)	72
清单 4.15	使用客户端加锁实现的“缺少即加入”	73
清单 4.16	使用组合 (composition) 实现“缺少即加入”	74
清单 5.1	操作 Vector 的复合操作可能导致混乱的结果.....	80
清单 5.2	使用客户端加锁, 对 Vector 进行复合操作.....	81
清单 5.3	迭代中可能抛出的 ArrayIndexOutOfBoundsException	81
清单 5.4	使用客户端加锁进行迭代	82
清单 5.5	用 Iterator 对 List 进行迭代	82
清单 5.6	迭代隐藏在字符串的拼接中 (不要这样做)	84
清单 5.7	ConcurrentMap 接口.....	87
清单 5.8	桌面搜索应用程序中的生产者和消费者	91
清单 5.9	开始桌面搜索	92
清单 5.10	恢复中断状态, 避免掩盖中断	94
清单 5.11	在时序测试中, 使用 CountdownLatch 来启动和停止线程.....	96
清单 5.12	使用 FutureTask 预载稍后需要的数据	97
清单 5.13	Throwable 强制转换为 RuntimeException.....	98
清单 5.14	使用信号量来约束容器	100
清单 5.15	在一个细胞的自动系统中用 CyclicBarrier 协调计算	102
清单 5.16	尝试使用 HashMap 和同步来初始化缓存	103
清单 5.17	用 ConcurrentHashMap 替换 HashMap.....	105
清单 5.18	用 FutureTask 记录包装器	106
清单 5.19	Memoizer 最终实现.....	108
清单 5.20	使用 Memoizer 为因式分解的 servlet 缓存结果.....	109
清单 6.1	顺序化的 Web Server	114

清单 6.2	Web Server 为每个请求启动一个新的线程	115
清单 6.3	Executor 接口	117
清单 6.4	使用线程池的 Web server	118
清单 6.5	为每个任务启动一个新线程的 Executor	118
清单 6.6	Executor 在调用线程中同步地执行所有任务	119
清单 6.7	ExecutorService 中的生命周期方法	121
清单 6.8	支持关闭操作的 Web Server	122
清单 6.9	Timer 的混乱行为	124
清单 6.10	顺序地渲染页面元素	125
清单 6.11	Callable interface 和 Future interface	126
清单 6.12	ThreadPoolExecutor 中 newTaskFor 的默认实现	126
清单 6.13	使用 Future 等待图像下载	128
清单 6.14	ExecutorCompletionService 使用的 QueueingFuture 类	129
清单 6.15	使用 CompletionService 渲染可用的页面元素	130
清单 6.16	在预定时间内获取广告信息	132
清单 6.17	在预定时间内请求旅游报价	134
清单 7.1	使用 volatile 域保存取消状态	137
清单 7.2	生成素数的程序运行一秒钟	137
清单 7.3	不可靠的取消把生产者置于阻塞的操作中（不要这样做）	139
清单 7.4	线程的中断方法	139
清单 7.5	通过使用中断进行取消	141
清单 7.6	向调用者传递 InterruptedException	143
清单 7.7	不可取消的任务在退出前保存中断	144
清单 7.8	在外部线程中安排中断（不要这样做）	145
清单 7.9	在一个专门的线程中中断任务	146
清单 7.10	通过 Future 来取消任务	147
清单 7.11	在 Thread 中，通过覆写 interrupt 来封装非标准取消	149
清单 7.12	用 newTaskFor 封装任务中非标准取消	151
清单 7.13	不支持关闭的生产者-消费者日志服务	152
清单 7.14	向日志服务添加不可靠的关闭支持	153
清单 7.15	向 LogWriter 添加可靠的取消	154
清单 7.16	使用 ExecutorService 的日志服务	155
清单 7.17	使用致命药丸来关闭	156
清单 7.18	IndexingService 的生产者线程	157

清单 7.19	IndexingService 的消费者线程.....	157
清单 7.20	使用私有 Executor，将它的寿命限定于一次方法调用中.....	158
清单 7.21	关闭之后，ExecutorService 获取被取消的任务.....	159
清单 7.22	使用 TrackingExecutorService 为后续执行来保存未完成任务.....	160
清单 7.23	典型线程池的工作者线程的构建.....	162
清单 7.24	UncaughtExceptionHandler 接口.....	163
清单 7.25	UncaughtExceptionHandler 将异常写入日志.....	163
清单 7.26	注册关闭钩子来停止日志服务.....	165
清单 8.1	在单线程化的 Executor 中死锁的任务（不要这样做）.....	169
清单 8.2	ThreadPoolExecutor 通用的构造函数.....	172
清单 8.3	创建一个可变长的线程池，使用受限队列和“调用者运行”饱和策略。.....	175
清单 8.4	使用 Semaphore 来遏制任务的提交.....	176
清单 8.5	ThreadFactory 接口.....	176
清单 8.6	定制的线程工厂.....	177
清单 8.7	自定义的线程基类.....	178
清单 8.8	修改一个标准工厂方法创建的 Executor.....	179
清单 8.9	扩展线程池以提供日志和计时功能.....	180
清单 8.10	把顺序执行转换为并行执行.....	181
清单 8.11	把顺序递归转换为并行递归.....	182
清单 8.12	等待并行运算的结果.....	182
清单 8.13	类似于“搬箱子”谜题的抽象.....	183
清单 8.14	谜题解决者框架的链节点.....	184
清单 8.15	顺序化的谜题解决者.....	185
清单 8.16	并发版的谜题解决者.....	186
清单 8.17	ConcurrentPuzzleSolver 使用可携带结果的闭锁.....	187
清单 8.18	能够感知任务不存在的解决者.....	188
清单 9.1	使用 Executor 实现的 SwingUtilities.....	193
清单 9.2	构建于 SwingUtilities 之上的 Executor.....	194
清单 9.3	简单的事件监听器.....	194
清单 9.4	将耗时任务绑定到可视化组件.....	196
清单 9.5	提供用户反馈的耗时任务.....	196
清单 9.6	取消耗时任务.....	197
清单 9.7	支持取消、完成和进度通知的后台任务类.....	199
清单 9.8	在 BackgroundTask 中启动一个耗时的、可取消的任务.....	200

清单 10.1	简单的锁顺序死锁（不要这样做）	207
清单 10.2	动态加锁顺序产生的死锁（不要这样做）	208
清单 10.3	制定锁的顺序来避免死锁	209
清单 10.4	开始一个循环，它在典型条件下制定死锁	210
清单 10.5	协作对象间的锁顺序死锁（不要这样做）	212
清单 10.6	使用开放调用来避免协作对象之间的死锁	214
清单 10.7	发生死锁后线程转储的部分信息	217
清单 11.1	串行访问任务队列	227
清单 11.2	徒劳的同步（不要这样做）	230
清单 11.3	锁省略的候选程序	231
清单 11.4	持有锁超过必要的时间	233
清单 11.5	减少锁持续的时间	234
清单 11.6	应当分拆锁的候选程序	236
清单 11.7	使用分拆的锁重构 ServerStatus	236
清单 11.8	基于哈希的 map 中使用分离锁	238
清单 12.1	利用 Semaphore 实现的有限缓存	249
清单 12.2	BoundedBuffer 的基本单元测试	250
清单 12.3	测试阻塞与响应中断	252
清单 12.4	适用于测试的中等品质的随机数生成器	253
清单 12.5	BoundedBuffer 的生产者-消费者测试程序	255
清单 12.6	PutTakeTest 中的生产者和消费者类	256
清单 12.7	测试资源泄漏	258
清单 12.8	用于测试 ThreadPoolExecutor 的线程工厂	258
清单 12.9	验证线程池扩展的测试方法	259
清单 12.10	使用 Thread.yield 产生更多的交替操作	260
清单 12.11	基于关卡的计时器	261
清单 12.12	使用基于关卡的计时器进行测试	262
清单 12.13	TimedPutTakeTest 的驱动程序	262
清单 13.1	Lock 接口	277
清单 13.2	使用 ReentrantLock 保护对象状态	278
清单 13.3	使用 tryLock 避免锁顺序死锁	280
清单 13.4	具有预定时间的锁	281
清单 13.5	可中断的锁获取请求	281
清单 13.6	ReadWriteLock 接口	286

清单 13.7	用读写锁包装的 Map.....	288
清单 14.1	状态依赖的可阻塞行为的结构.....	292
清单 14.2	有限缓存不同实现的基类.....	293
清单 14.3	如果有限缓存不满足先验条件，会停滞不前.....	294
清单 14.4	调用 GrumpyBoundedBuffer 的客户端逻辑.....	294
清单 14.5	有限缓存使用了拙劣的阻塞.....	296
清单 14.6	有限缓存使用条件队列.....	298
清单 14.7	状态依赖方法的规范式.....	301
清单 14.8	在 BoundedBuffer.put 中使用“依据条件通知”.....	304
清单 14.9	使用 wait 和 notifyAll 实现可重关闭的阀门.....	305
清单 14.10	Conditon 接口.....	307
清单 14.11	有限缓存使用显式的条件变量.....	309
清单 14.12	使用 lock 实现的计数信号量.....	310
清单 14.13	AQS 中获取和释放操作的规范式.....	312
清单 14.14	二元闭锁使用 AbstractQueuedSynchronizer.....	313
清单 14.15	不公平的 ReentrantLock 中 tryAcquire 的实现.....	315
清单 14.16	Semaphore 的 tryAcquireShared 和 tryAcquireShared 方法.....	316
清单 15.1	模拟 CAS 操作.....	322
清单 15.2	使用 CAS 实现的非阻塞计数器.....	323
清单 15.3	使用 CAS 避免多元的不变约束.....	326
清单 15.4	使用 ReentrantLock 实现随机数字生成器.....	327
清单 15.5	使用 AtomicInteger 实现随机数字生成器.....	327
清单 15.6	使用 Treiber 算法（Treiber, 1986）的非阻塞栈.....	331
清单 15.7	Michael-Scott 非阻塞队列算法中的插入（Michael 与 Scott, 1996）.....	334
清单 15.8	在 ConcurrentLinkedQueue 中使用原子化的域更新器.....	335
清单 16.1	没有充分同步的程序可以产生令人惊讶的结果（不要这样做）.....	340
清单 16.2	FutureTask 的内部类示范了如何“驾驭”同步.....	343
清单 16.3	不安全的惰性初始化（不要这样做）.....	345
清单 16.4	线程安全的惰性初始化.....	347
清单 16.5	主动初始化.....	347
清单 16.6	惰性初始化 holder 类技巧.....	348
清单 16.7	双检查锁反模式（不要这样做）.....	349
清单 16.8	不可变对象的初始化安全性.....	350

介绍

Introduction

编写正确的程序并不容易，而编写正确的并发程序就更难了。与顺序执行的程序相比，并发程序中显然更容易出现错误。那么，我们为什么会对并发如此烦恼呢？线程是 Java 语言不可避免的特性，它们把复杂、异步的代码转化为更简单、更直观的代码，从而简化复杂系统的开发。进一步而言，线程是控制和利用多处理器系统计算能力的最简单方式。同时，伴随着处理器数量的增加，有效地采用并发会变得越来越重要。

1.1 并发的（非常）简短历史

在发展的初期，计算机还没有操作系统；它们自始至终执行一个程序，这个程序直接访问机器的所有资源。这样一个程序运行在无保护的金属器件上，不仅写起来困难，而且每次只运行一个程序，不能很好地利用昂贵且稀缺的计算机资源。

操作系统的发展使得多个程序能够同时运行，程序在各自的**进程**（processes）中运行：相互分离，各自独立执行，由操作系统来分配资源，比如内存、文件句柄、安全证书。如果需要的话，进程会通过一些原始的机制相互通信：Socket、信号处理（signal handlers）、共享内存（shared memory）、信号量（semaphores）和文件。

有一些促进因素，它们推动了操作系统支持多程序同时执行的发展：

资源利用。程序有时候需要等待外部的操作，比如输入和输出，并且在等待的时候不可能进行有价值的工作。在等待的时候，让其他的程序运行会提高效率。

公平。多个用户或程序可能对系统资源有平等的优先级别。让他们通过更好的时间片方式来共享计算机，这要比结束一个程序后才开始下一个程序更可取。

方便。写一些程序，让它们各自执行一个单独任务并进行必要的相互协调，这要比编写一个程序来执行所有的任务更容易，更让人满意。

在早期的分时共享系统中，每一个进程都是一个虚拟的冯诺依曼（von Neumann）机；它拥有一个内存空间，储存着指令和数据，根据机器语言的语义来顺序地执行指令，并且通过操作系统的 I/O 原语（primitive）集来实现与外部世界的交互。对于每一条指令的执行，都有一个对“下一条指令”的明确定义，并根据程序中的指令集来进行流程的控制。现在几乎所有广泛使用的编程语言都遵循这个顺序的编程模型，其中语言规范明确定义了一个给定动作完成后，下一个动作是什么。

顺序编程模型是自然的、常规的，就像是遵守着人类的工作方式：一次做一件事情，顺序进行——通常如此。起床，穿上浴衣，下楼，开始准备早茶。在编程语言中，真实世界中的每一个动作，都会抽象成一个规则的动作序列——打开食品柜，选择你喜欢的茶，在罐里放入适量的茶叶，看看茶壶中是不是有足够多的水，如果不够就加些水，把茶壶放在炉子上，打开炉子，等待水的沸腾等等。最后一步——等待水沸腾——也引入了**异步**这个要点。当水在加热的时候，你可以选择做什么——等待，或者开始准备吐丝面包（另一个异步任务），还可以取一份报纸看，同时仍然要记得煮开水的壶马上就会需要你的关注。开水和吐丝面包的生产者知道他们的产品通常在异步的情况下使用，所以在任务结束的时候，它们会提高信号的音量。找到顺序和异步之间最好的平衡，通常是那些高效率人士的一个特点——对于程序来说也是如此。

相同的关注点（资源利用，公平和方便）不仅促进了进程的发展，也促进了**线程**的发展。线程允许程序控制流（control flow）的多重分支同时存在于一个进程。它们共享进程范围内的资源，比如内存和文件句柄，但是每一个线程有其自己的程序计数器（program counter）、栈（stack）和本地变量。线程也为多处理器系统中并行地使用硬件提供了一个自然而然的分解；同一程序内的多个线程可以在多 CPU 的情况下同时调度。

线程有些时候被称为**轻量级进程**（lightweight processes），并且大多数现代操作系统把线程作为时序调度的基本单元，而不是进程。在没有明确协调的情况下，线程相互间同时或异步地执行。因为线程共享其所属进程的内存地址空间，因此所有同一进程中的线程访问相同的变量，并从同一个堆中分配对象，这相对于进程间通信（inter-process）机制来说实现了良好的数据共享。但是如果没有明确的同步来管理共享数据，一个线程可能会修改其他线程正在使用的数据，产生意外的结果。

1.2 线程的优点

恰当地使用线程时，可以降低开发和维护的开销，并且能够提高复杂应用的性能。线程通过把异步的工作流程转化为普遍存在的顺序流程，使程序模拟人类工作和交互变得更容易了。另一方面，它们可以把复杂、难以理解的代码转化为直接、简洁的代码，这样更容易读写及维护。

线程在 GUI 应用程序中是非常有用的，可用来改进用户接口的响应性，并且在服务器应用中，用于提高资源的利用率和吞吐量。它们也可以简化 JVM 的实现——垃圾收集器（garbage collector）通常运行于一个或多个持续工作的线程之间。大部分至关重要的 Java 应用都依赖于线程，某种程度上是因为它们的组织结构需要这样。

1.2.1 使用多处理器

多处理器系统以往比较昂贵、稀少，只用于大的数据中心和科学计算设备。如今，多处理器系统已经比较便宜，数量也增多了；即使低端服务器和中端的桌面系统也常常采用多处理器。这个趋势只会逐渐增加；因为处理器很难再提高它的时钟频率，取而代之的是，处理器厂商会在一片芯片上放置更多的处理器内核。所有主要的芯片制造商都开始了这种转变，并且我们已经显著地看到机器中处理器数量的增加。

因为程序调度的基本单元是线程，一个单线程应用程序一次只能运行在一个处理器上。在双处理器系统中，一个单线程程序，放弃了其中一半的空闲 CPU 资源；在拥有 100 个处理器的系统中，这个单线程程序放弃了 99% 的资源。从另一方面来看，拥有多个活跃线程的程序可以同时有多处理器上运行。在设计良好的情况下，多线程程序通过更有效的利用空闲处理器资源，来提高吞吐量。

使用多线程也可以帮助我们在单处理器系统中实现最佳的吞吐量。如果一个程序是单线程的，这个处理器在等待一个同步 I/O 操作完成的时候，仍然是空闲的。在一个多线程程序中，当第一个线程等待 I/O 结束的同时，另外一个线程也可以运行，这样就使得应用程序在遇到 I/O 阻塞的时候仍然有进展。（这就像在等待水烧开的时间里读报纸，优于等待水开之后再去读报。）

1.2.2 模型的简化

当你需要完成的任务全都是同一类型（修改 12 个 bug）的时候，掌控你的时间通常比完成多种类型的任务要容易（修改 bug，面试系统管理员的替代候选人，完成你团队的效率评估，为你下周的演讲制作幻灯片）。当你只有一种任务的时候，你可以从这堆任务的第一个开始，逐一去做，直到将它们全部完成（或者你自己精疲力竭）；你不

需要花精力去思考接下来去执行哪个任务。另一方面，管理多重优先级和截止日期，还要在各任务之间切换，这通常会带来一些成本开销。

对于软件来说也是同样的道理：相对于需要同时管理多种类型的任务，一个顺序处理相同类型任务的程序，写起来更简单，更少出错，也更容易测试。在模拟的情况下，为每一个类型的任务分配一个线程，或者为每一个元素分配一个线程，提供理想上的顺序，并且这样做可以把域逻辑(domain logic)与时序调度的细节隔绝开来，进行相互交替的操作，进行异步 I/O，以及等待资源。一个复杂、异步的流程可以被分解为一系列更简单的同步流程，它们中每一个在相互独立的线程中运行，只有在特定的同步点才进行彼此间的交互。

这些优点通常被一些框架(framework)所使用，比如 Servlets 或者远程方法调用(RMI, Remote Method Invocation)。这些框架需要处理请求管理，线程创建和负载均衡的细节，与此同时还要处理不同部分转发到合适的组件的相应流程状态中去。Servlet 的开发者不需要担心容器究竟同时正在处理多少个请求，或者 Socket 的输入输出流是否阻塞；当一个 Servlet 的 Service 方法作为 Web 请求的响应被调用时，它可以同步地处理这些请求，就像它是一个单线程程序一样。这可以简化组件开发，并且可以使学习曲线变缓。

1.2.3 对异步事件的简单处理

一个服务器应用程序，接受来自多个远程客户端的连接，如果每一个连接服务器都为其分配一个线程，并允许使用同步 I/O，这样的程序开发起来更容易。

如果程序在读取 Socket 时没有可用数据，那么 read 方法会被阻塞直到有数据可用。在一个单线程应用程序中，这不仅意味着处理相应的请求停止了，也意味着在线程阻塞期间对**所有**请求的处理都停止了。为了避免这样的问题，单线程服务器程序被迫使用了非阻塞 I/O，这要比同步 I/O 复杂得多，也更容易出错。然而，如果每一个请求都拥有自己的线程，那么阻塞就不会影响到其他请求的处理了。

历史上，操作系统把一个进程能够创建的线程限制在相对比较少数量上，大约有几百个(甚至更少)。因此，操作系统为多元化的 I/O 开发了一些高效的机制，比如 Unix 的 select 和 poll 系统调用，为了访问这些机制，Java 类库对非阻塞 I/O 提供了一组包(java.nio)。然而，今天的操作系统在支持更大数量的线程方面有了巨大的进步，这使得即使是在那些拥有许多客户的平台上¹，“每线程每客户(thread-per-client)”模型也是现实的。

¹ NPTL线程包，现在作为Linux发布的一部分，设计它的目的是用来支持数十万甚至于更多的线程。非阻塞的I/O有它自身的优势，但是操作系统对线程更好的支持意味着出现更少的底层困境。

1.2.4 用户界面的更佳响应性

GUI 应用程序过去通常为单线程的，这意味着你要么通过大量输入事件频繁地测试整个代码（这通常杂乱无章且麻烦），要么执行所有应用代码，间接地贯穿整个主事件循环（main event loop）。如果从主事件循环中调用的代码执行的时间过长，那么直到代码执行完毕，用户界面看上去都是冻结的，因为在控制权返回到主事件循环之前，程序无法执行用户界面事件。

AWT 和 Swing 工具集这样的现代 GUI 框架，用**事件派发线程**（event dispatch thread, EDT）取代了主事件循环。当一个用户界面事件发生时，比如按下一个按钮，事件线程会调用程序定义的事件处理器。大部分 GUI 框架都是单线程化的子系统，所以主事件循环的有效性仍然可以得到体现，但是它运行于它自身线程 GUI toolkit 的控制下，而并非受控于应用程序。

在事件发生的线程中如果只有短暂的任务，那么界面总能够作出响应，因为事件线程总能够及时有效地处理好用户的活动。然而，在事件线程中处理一个长期、耗时的任务，比如一个大文档的拼写检查，或者从网络上获取一个资源，这会削减响应的效率。如果用户在这个任务运行的时候发生了一个新的动作，事件线程能够开始处理甚至知晓这个动作都会被延迟很久。雪上加霜的情况是，不仅 UI 失去了响应，而且用户很可能不能取消这个不愉快的任务，即使程序提供了 cancel 按钮，因为事件线程正在忙碌工作，直到那个冗长的任务结束，才能够开始处理 cancel 按钮的按下事件！但是，如果让这个耗时的任务运行在单独的线程里，那么事件线程就能够自由地处理 UI 事件，使之具有更好的响应能力。

1.3 线程的风险

Java 对线程内置的支持是一把双刃剑。它通过提供语言和类库，以及一个规范的跨平台存储模型（这个规范的存储模型使得在 Java 中开发“一次开发，随处运行(write-once, run-anywhere)”的**并发**程序成为可能），简化了并发应用的开发。这样做同时还提高了开发人员的门槛，因为更多的程序需要使用线程。曾几何时，当线程还十分深奥的时候，并发还是一个“高级”的话题；现在，主流的开发人员都必须知道线程安全性的问题。

1.3.1 安全危险

线程安全的问题是微妙且出乎意料的，因为在没有进行充分同步的情况下，多线程中的各个操作的顺序是不可预测的，有时甚至令人惊讶。清单 1.1 中，UnsafeSequence 试图生成一个唯一整数值的序列。下面提供了一个简单的插图来解释多线程中交替（interleaving）的动作如何导致意外结果的。如果在单一线程的环境中，它能够正确运行，但是在多线程环境中却不行。

清单 1.1 非线程安全的序列生成器

```
@NotThreadSafe
public class UnsafeSequence {
    private int value;

    /** 返回一个唯一值. */
    public int getNext() {
        return value++;
    }
}
```

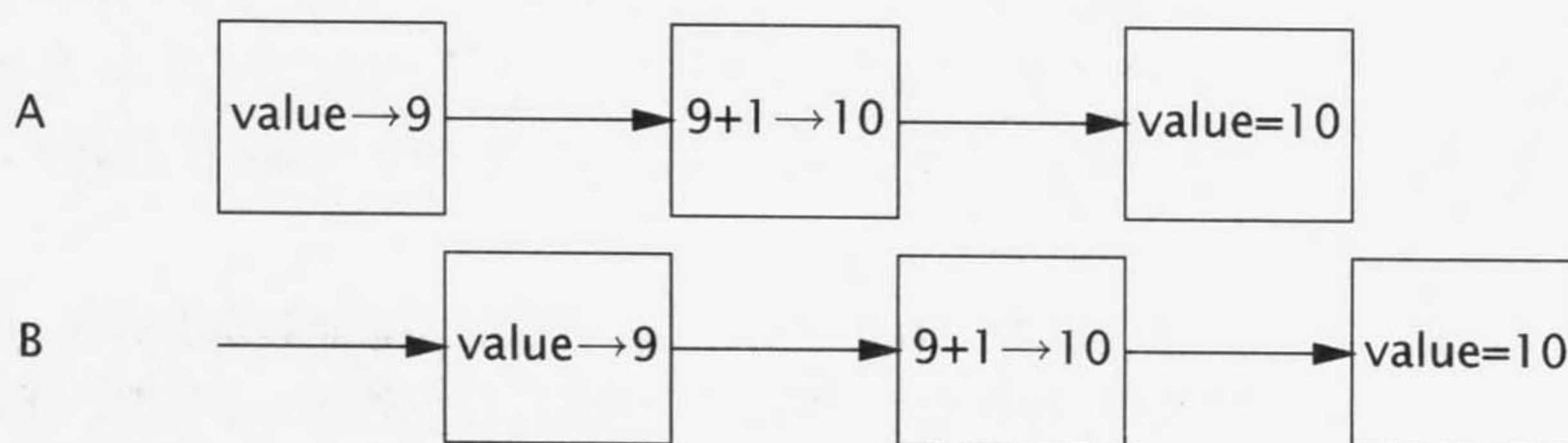


图 1.1 UnsafeSequence.getNext 执行失败

UnsafeSequence 中的问题是, 在一些特殊的时序情况下, 两个线程可以调用 getNext 并得到相同的返回值。图 1.1 表现了这是如何发生的。自增操作 value++ 可能看起来是一个单一的操作, 但是事实上它分为 3 个独立的操作: 读取这个值, 使之加 1, 再写入新值。因为这些操作发生在多个线程中, 这些线程可能交替占有运行时 (runtime), 所以两个线程很可能同时读取这个值, 两个线程都得到相同的值, 并都使之增加了 1。结果就是不同的线程返回了相同的序列数。

图 1.1 中的图表描述了不同线程之间的交替操作。在这些图表中, 时间由左至右发展, 每一行表现一个不同线程的活动。这些交替的图表通常用来描述最坏的情况², 目的是表现特定顺序下产生错误的僭越带来的危险。

UnsafeSequence 使用了一个非标准的标签 (Annotation): @NotThreadSafe。这是本书中几个自定义的 Annotation 之一, 这些 Annotation 用来标明类和类成员的并发特

² 事实上, 像我们将在第3章看到的那样, 最坏的情况可能比图中表现的更糟糕, 因为存在重排序的可能性。

性。(其他的类级 Annotation 还有: @ThreadSafe 和 @Immutable; 详见附录 A) 用 Annotation 给线程安全进行标记对于各类读者来说是非常有用的。如果一个类标记为 @ThreadSafe, 用户就可以充满信心地把它应用于多线程环境, 维护者看到它可以认为是线程安全的必然保证, 而软件分析工具可以转而去识别那些可能存在的代码错误。

UnsafeSequence 阐明了一种常见的并发危险: **竞争条件** (race condition)。当被多线程调用时, getNext 是否能返回不重复的值, 正像它的规约描述的, 而这取决于运行时如何交替进行这些操作——这不是我们所希望看到的势态。

因为线程共享相同的内存地址空间, 且并发地运行, 它们可能访问或修改其他线程正在使用的变量。这是十分方便的, 因为它使得数据共享相对于其他的线程间通信机制都更加简单。但是这其中也存在着巨大的风险: 当数据意外改变时, 线程可能会出现混乱。允许多线程访问和修改相同的变量, 给顺序编程模型引入了一些非顺序因素, 这可能会造成混乱, 并且难以发现错误的原因。为了使多线程程序的行为可预见, 访问共享的变量必须经过合理的协调, 这样线程才不会相互干扰。幸运的是, Java 提供了同步机制来协调这样的访问。

像清单 1.2³ 中那样, 可以通过把 getNext 声明为 synchronized 类型的方法来修正 UnsafeSequence, 因此可以避免图 1.1 所示的那种不应出现的交互。(这样做能够避免这个错误的确切原因将是第 2 章和第 3 章的主题。)

清单 1.2 线程安全的序列生成器

```
@ThreadSafe
public class Sequence {
    @GuardedBy("this") private int value;

    public synchronized int getNext() {
        return value++;
    }
}
```

在缺少同步的时候, 编译器、硬件和运行时事实上对时间和活动 (action) 顺序是很随意的, 比如在寄存器或者高速缓存中存储变量, 这样会使它们对于其他线程暂时 (甚至是永远) 不可见。这样的行为被普遍认为是能提高性能的非可行的办法, 但是这给开发人员带来了负担, 他们需要明确识别数据在线程中究竟如何共享, 这样这些优化才不至于破坏安全。(第 16 章给出了一些非常详细的细节, JVM 确切地产生了什么样的顺序担保,

³ @GuardedBy 在 2.4 小节会进行描述。它文档化了顺序序列的同步策略。

同步如何影响这些担保的，不过只要你遵循第2章、第3章中的规则，那么避开这些底层的细节也能达到安全性。)

1.3.2 活跃度的危险

在开发并发代码时，对线程安全的关注是至关重要的：安全不能妥协。安全的重要性不仅仅存在于多线程程序中，单线程化的程序也必须注意保护安全性和正确性，但是线程的使用引入了不会出现在单线程化程序中的额外安全危险。举例来说，线程的使用引入了又一形式的**活跃度失败**（liveness failure），这不会出现在单线程化的程序中。

如果**安全**意味着“什么坏事都没有发生过”，活跃度关注是与之互补的一面“好事最终发生了”。当一个活动进入某种它永远无法再继续执行的状态时，活跃度失败就发生了。一种活跃度失败可以发生在顺序程序中，这就是粗心造成的无限循环，那些在循环之后的代码永远不会被执行。多线程的引入带来了更多的活跃度危险。例如，如果线程A等待一个线程B独立占有的资源，B永远不释放这个资源，A将永远等待下去。第10章将讲述各种形式的活跃度失败，包括死锁（deadlock，10.1小节）、饥饿（starvation，10.3.1小节）、活锁（livelock 10.3.3小节），以及如何避免它们发生。像大多数同步bug一样，引起活跃度失败的bug总是难以察觉到，因为它们取决于线程间的相关的事件时序，因此在开发和测试中，并没有很多机会发现它们。

1.3.3 性能危险

与活跃度相关的是**性能**（performance）。虽然活跃度意味着好的事情**终究**会发生，但是最后可能还是不够好——我们通常希望好事情尽快发生。性能问题涉及很多方面，包括服务时间、响应性、吞吐量、资源消费或者可伸缩性的不良表现。就像安全和活跃度一样，多线程程序出现所有单线程程序中遇到的性能危险，而且还会有因线程的使用带来的风险。

在设计良好的应用程序中使用线程，能够获得纯粹的性能收益，但是线程仍然会给运行时带来一定程度的开销。**上下文切换**（Context switches）——当调度程序临时挂起当前运行的线程时，另一个线程开始运行——这在多个线程组成的应用程序中是很频繁的，并且带来巨大的系统开销：保存和恢复线程执行的上下文，离开执行现场，并且CPU的时间会花费在对线程的调度而不是在运行上。当线程共享数据的时候，它们必须使用同步机制，这个机制会限制编译器的优化，能够清空或锁定内存和高速缓存，并在共享内存的总线上创建同步通信。所有这些因素又引入了新的性能开销；第11章将介绍分析和降低这些开销的技术。

1.4 线程无处不在

即使你的程序没有显式地创建任何线程，框架也可能为你创建了一些线程，这些线程调用的代码必须是线程安全的（thread-safe）。这一点给开发人员的设计和实现赋予了更重要的一份责任，因为开发线程安全的类要比非线程安全的类需要更加仔细，进行更多的分析。

每一个 Java 应用程序都使用线程。当 JVM 启动后，它创建一些线程来进行自身的常规管理（垃圾回收，终结处理），以及一个运行 main 函数的主线程。AWT 和 Swing 用户接口框架创建线程来管理用户接口事件。Timer 创建执行延迟的任务线程。组件框架，比如 servlet 和 RMI，会创建线程池，池中线程调用组件方法。

如果你使用这些功能——像大多数程序员一样——你必须熟悉并发和线程安全，因为这些框架创建线程，并在线程中调用你的组件。如果你愿意自欺欺人地认为并发是“可选的”、“高级的”语言特性，这很好。不过事实上，几乎所有的 Java 应用程序都是多线程的，并且这些框架也不能让你完全避开对应用程序状态访问作适当的协调。

当并发由一个框架引入到一个应用程序中，并不能仅仅让框架的代码知晓并发的存在。因为框架代码本质上是通过回调来使用程序组件，而这些组件才真正用来访问程序状态。与之类似的是，线程安全的需要并不仅仅在于框架调用的组件——只要它处于组件访问过的程序状态段，它就会扩展到所有代码路径。因此，线程安全的需要是具有传递性的。

通过从框架线程中调用应用程序的组件，框架把并发引入了应用程序。组件总是需要访问程序的状态。因此要求在所有代码路径访问状态时，必须是线程安全的。

以下描述的这些场景，都会引发非应用程序管理线程调用应用程序代码这种情况。当需要线程安全的时候，可能会以这样的情况作为开始，可是这样做几乎都不能正常结束；反而会影响到整个程序。

定时器。Timer 用来调度一些稍后运行的任务，也可以是只运行一次或者周期性运行的任务，这是一种非常方便的机制。引入 Timer 可以使一个通常简单的顺序程序变复杂，因为 TimerTasks 运行在由 Timer 管理的线程中，并不是由应用程序来管理。如果一个 TimerTask 访问了其他应用程序线程正在访问的数据，那么不仅 TimerTask 需要线程安全的手段，并且**其他那些同时访问这个数据的类也需要相应措施**。通常最简单的实现方法

是确保 `TimerTask` 访问的对象本身是线程安全的，因此应该将线程安全性封装到共享对象的内部。

Servlets and JavaServer Pages (JSPs) Servlets 框架的设计目的是处理 Web 应用的部署，分发来自远程 HTTP 客户的请求这些的基础层业务。一个请求到达 Server 并被分发后，可能通过一个过滤器链到达相应的 Servlet 或者 JSP。每一个 Servlet 代表应用逻辑的一个组件，在访问量较大的网站中，许多客户可能同时对相同 Servlet 的服务提出请求。Servlets 的规范规定了一个 Servlet 必须为多个用户同时调用它作好准备。换句话说，Servlets 应该是线程安全的。

即使你能够保证一个 Servlet 一次只被一个线程调用，你在建立一个 Web 应用程序时可能仍然需要注意线程安全。Servlets 通常访问与其他 Servlets 共享的状态信息，比如程序范围内的对象（那些存储于 `ServletContext` 的对象）或者 Session 范围内的对象（这些保存在每个客户的 `HttpSession` 中）。当一个 Servlet 访问的对象是在 Servlets 间共享或者请求间共享时，必须对这些对象的访问控制进行适当协调，因为来自不同线程的多个请求可能同时访问它们。Servlet、JSP 和 Servlet Filter 以及那些存储在 `ServletContext` 和 `HttpSession` 容器中的对象，明显必须是线程安全的。

远程方法调用 (Remote Method Invocation) RMI 使你能够调用在另外一个 JVM 上运行的对象的方法。当你使用 RMI 调用一个远程方法时，这个方法的参数被打包（装配）成一个比特流，并且穿越整个网络到达远程 JVM，在那里它会被解包（分解）并传递给远程方法。

当 RMI 代码调用了你的远程对象时，这个调用发生在哪一个线程？你并不知道，但绝对不是你创建的那个线程——你的对象被 RMI 管理的一个线程所调用。RMI 创建了多少个线程？在许多 RMI 线程中，同一个对象的同一个方法是不是有可能同时被调用⁴？

一个远程对象必须去守卫两种线程安全风险：对那些可能会与其他对象共享的状态进行适当调节，应正确地对远程对象本身进行调控（因为相同的对象可能同时被多个线程调用）。比如 `servlets`，RMI 对象应该对同时发生的多个调用有所准备，并且必须提供它们自己的线程安全。

Swing 和 AWT GUI 应用程序具有固有的异步特性。用户可能选择一个菜单项，或者在任何时候按下一个按钮，他们希望程序迅速作出响应，即使程序正在做其他事情。Swing 和 AWT 通过创建一个单独的线程来处理用户发起的事件，并更新那些展现给用户的图形界面。

⁴ 答案是：是的。但是 Javadoc 里面写的并不是很详细——你必须去阅读 RMI 规范。

Swing 组件, 比如 `JTable`, 都不是线程安全的。相反, Swing 程序通过限制访问事件线程中的 GUI 组件, 实现了它们的线程安全。如果应用程序希望从事件线程之外进行操控, 它必须促使操控 GUI 的代码在事件线程中运行。

当用户进行了一个 UI 活动, 事件线程会调用事件处理器 (handler) 来执行用户请求的操作。如果处理器需要访问应用程序状态, 这里很可能有其他程序在正在访问 (比如正在编辑的文件), 那么事件的处理器, 与其他访问状态的代码必须以线程安全的方式工作。

PART

第 1 部分

基 础

Fundamentals

线程安全

Thread Safety

也许你会惊讶，并发编程并不会涉及过多的线程或锁，不会多于建筑工程中使用的铆钉和 I 型梁。当然，要让桥梁坚固耐用，需要正确使用大量的铆钉和 I 型梁；同样的道理，构建并发程序也要正确使用线程和锁。然而这仅仅是**纸上谈兵**（mechanisms）——获得最终结果的方式。编写线程安全的代码，本质上就是管理对**状态**（state）的访问，而且通常都是**共享的、可变的**状态。

通俗地说，一个对象的**状态**就是它的数据，存储在**状态变量**（state variables）中，比如实例域或静态域。对象的状态还包括了其他附属对象的域。例如，HashMap 的状态一部分存储到对象本身中，但同时也存储到很多 Map.Entry 对象中。一个对象的状态包含了任何会对它外部可见行为产生影响的数据。

所谓**共享**，是指一个变量可以被多个线程访问；所谓**可变**，是指变量的值在其生命周期内可以改变。我们讨论的线程安全性好像是关于**代码**的，但是我们真正要做的，是在不可控制的并发访问中保护**数据**。

一个对象是否应该是线程安全的取决于它是否会被多个线程访问。线程安全的这个性质，取决于程序中如何**使用**对象，而不是对象**完成**了什么。保证对象的线程安全性需要使用同步来协调对其可变状态的访问；若是做不到这一点，就会导致脏数据和其他不可预期的后果。

无论何时，只要有多于一个的线程访问给定的状态变量，而且其中某个线程会写入该变量，此时必须使用同步来协调线程对该变量的访问。Java 中首要的同步机制是 synchronized 关键字，它提供了独占锁。除此之外，术语“同步”还包括 volatile 变量，显示锁和原子变量的使用。

你会想到在一些“特殊”情况下上述规则并不适用，不过你应该抵制住这种想法的诱惑。程序如果忽略了必要的同步，可能看上去可以运行，而且能够通过测试，甚至能正常地运行数年，但它仍然是存在隐患的，任何时刻都有可能崩溃。

在没有正确同步的情况下，如果多个线程访问了同一个变量，你的程序就存在隐患。有3种方法修复它：

- 不要跨线程共享变量；
- 使状态变量为不可变的；或者
- 在任何访问状态变量的时候使用同步。

如果你没有在类的设计中考虑并发访问的因素，需要使用上面的3种方法对类的设计作重大的修改，所以修复程序的问题并不像听上去那样轻而易举。**一开始就将一个类设计成是线程安全的，比在后期重新修复它更容易。**

在一个大型的程序中，识别出是否有多个线程可能访问给定的变量并不是一件容易的事情。幸运的是，面向对象技术——比如封装和数据隐藏——不仅帮助你编写组织良好的、可维护的类，同样的技术还可以帮助你创建线程安全的类。访问特定变量的代码越少，越容易确保使用恰当的同步，也越容易推断访问一个变量所需的条件。Java语言不强迫你封装所有的域，允许你将状态存储到公共域（甚至是公共静态域），或者将它的引用发布到其他的内部对象中，对程序的状态封装得越好，你的程序就越容易实现线程安全，同时有助于维护者保持这种线程安全性。

设计线程安全的类时，优秀的面向对象技术——封装、不可变性以及明确的不变约束——会给你提供诸多的帮助。

很多时候，良好的面向对象设计技术与现实世界需求不匹配；这种情况下，出于对系统性能和遗留代码的向后兼容性的考虑，良好的设计规则必须向现实世界作出妥协。有时候，抽象和封装会与性能产生冲突，虽然不像很多开发者认为的那样频繁，但是，首先让你的代码正确，**然后**（then）再让它跑得快，总是一个良好的实践。即便如此，性能优化也只发生在特定的条件下：性能标准和需求要求你必须去做，或者依据相同的衡量标准，你的优化运行在真实环境中时发生了变化¹。

如果你决定必须打破封装也无所谓。你的程序仍然可以是线程安全的，但是实现起来

¹ 对于并发代码，要更加坚定地遵循这个实践。因为并发性错误很难再现与调试，从一些不常用的代码路径上所获得的些许性能上的提升，与可能给程序带来的失败风险比起来，就得不偿失了。

更困难些。而且，程序的线程安全性会变得更加脆弱，这增加了开发与维护的开销和风险。第4章详述了在什么条件下你可以安全地打破对状态变量的封装。

到目前为止，我们几乎可以互换地使用着术语“线程安全类”和“线程安全程序”。一个线程安全程序是完全由线程安全类构成的么？不必要——完全由线程安全类构成的程序未必是线程安全的，线程安全程序也可以包含非线程安全的类。围绕着组合线程安全类的话题，还会在第4章提到。无论如何，只有当类封装了自己的状态时，“线程安全类”的概念才有意义。“线程安全性”可能成为用于约束**代码**的条款，或成为**状态**的条款，并且只能用于封装了自身状态的代码的整体，这个整体可能是一个对象，或是一个完整的程序。

2.1 什么是线程安全性

给“线程安全性”下个定义相当棘手。很多正式的定义都显得过于复杂，并没有给出实用的指导或者精到的见解；而其他非正式的描述看上去又完全是在兜圈子。在 Google 上搜索了一下，查到很多定义略举一二：

...可以被多个程序线程调用，这些线程之间没有非预期的互交。

...可以同时被多个线程调用，而调用者不需要任何动作（来确保线程的安全性）。

给出这样的定义，让我们对线程安全性产生困惑是不足为奇的！它们听上去令人怀疑：“如果一个类可以安全地被多个线程使用，它就是线程安全的。”你无法对此论述提出任何争议，但也无法从中得到更多有意义的帮助。我们如何辨别线程安全与非线程安全的类？我们甚至又该如何理解“安全”呢？

任何一个合理的“线程安全性”定义，其关键在于“**正确性**”的概念。如果我们关于线程安全性的定义是模糊的，那是因为缺少一个明确的“正确性”定义。

正确性意味着一个类**与它的规约保持一致**。良好的规约定义了用于强制对象状态的**不变约束**（invariants）以及描述操作影响的**后验条件**（postconditions）。通常我们不会为类写足够的规约，那么我们还能够知道程序的正确与否么？不能，但是只要我们相信“代码是可以工作的”，就不会阻止我们使用这些类。这种“代码自信”与我们所要实现的正确性紧密相关，所以不妨假设单线程化的正确性是“所见即所知”的事物。乐观地将“正确性”定义为“可被认知事物”后，我们现在可以少兜些圈子来定义“线程安全性”了：一个类是线程安全的，是指在被多个线程访问时，类可以持续进行正确的行为。

当多个线程访问一个类时，如果不用考虑这些线程在运行时环境下的调度和交替执行，并且不需要额外的同步及在调用方代码不必作其他的协调，这个类的行为仍然是正确的，那么称这个类是**线程安全的**。

任何单线程化的程序同时也是合法的多线程化的程序，倘若程序在单线程化的环境尚且不正确²，那么该程序必然不是线程安全的。对于一个正确实现的对象，顺序性的操作——比如调用公共的方法，读写公共域——不会破坏任何一个不变约束以及后验条件。对于线程安全类的实例进行顺序或并发的一系列操作，都不会导致实例处于无效状态。

线程安全的类封装了任何必要的同步，因此客户不需要自己提供。

2.1.1 示例：一个无状态（stateless）的 servlet

在第1章，我们列出了很多框架，这些框架会创建线程，并在这些线程中调用你的组件，而将确保组件线程安全的责任留给了你。通常需要线程安全的，并不是直接使用线程的情况，而是那些使用了便利工具（如 Servlets 框架）的情况。我们会展示一个例子，基于 Servlet 的因数分解服务，并逐步扩展它，添加新特性，同时确保它的线程安全性。

清单 2.1 展示了我们简单的因数分解的 Servlet。它从 Servlet Request 中解包数据，然后将这个数据进行因数分解，最后将结果封包到 Servlet Response 中。

清单 2.1 一个无状态的 Servlet

```
@ThreadSafe
public class StatelessFactorizer implements Servlet {
    public void service(ServletRequest req, ServletResponse resp) {
        BigInteger i = extractFromRequest(req);
        BigInteger[] factors = factor(i);
        encodeIntoResponse(resp, factors);
    }
}
```

² 如果在这里随意使用“正确性”会令你迷惑，你不妨将线程安全类看作这种类：它在并发环境中的隐患不会多于单线程环境下的隐患。

StatelessFactorizer 像大多数 Servlet 一样，是无状态的：它不包含域也没有引用其他类的域。一次特定计算的瞬时状态，会唯一地存在本地变量中，这些本地变量存储在线程的栈中，只有执行线程才能访问。一个访问 StatelessFactorizer 的线程，不会影响访问同一个 Servlet 的其他线程的计算结果；因为两个线程不共享状态，它们如同在访问不同的实例。因为线程访问无状态对象的行为，不会影响其他线程访问该对象时的正确性，所以无状态对象是线程安全的。

无状态对象永远是线程安全的。

多数 Servlet 都可以实现为无状态的，这一事实极大地降低了确保 Servlet 线程安全的负担，只有当 Servlet 要为不同的请求记录一些信息时，才会将线程安全的需求提到日程上来。

2.2 原子性

我们向无状态对象中加入一个状态元素会怎样？假设我们想要添加“命中数（hit counter）”来计算处理请求的数量。显而易见的方法是在 Servlet 中加入一个 long 类型的域，并在每个请求中递增它。如同清单 2.2 的 UnsafeCountingFactorizer 所示。

清单 2.2 Servlet 计算请求数量而没有必要的同步（**不要这样做**）

```
@NotThreadSafe
public class UnsafeCountingFactorizer implements Servlet {
    private long count = 0;

    public long getCount() { return count; }

    public void service(ServletRequest req, ServletResponse resp) {
        BigInteger i = extractFromRequest(req);
        BigInteger[] factors = factor(i);
        ++count;
        encodeIntoResponse(resp, factors);
    }
}
```



很遗憾，UnsafeCountingFactorizer 并非线程安全的，尽管它在单线程的环境中运行良好。正如第 6 页中的 UnsafeSequence，它很容易**遗失更新**（lost updates）。自增操作 ++count 由于其紧凑的语法格式，看上去更像一个单独的操作。然而，它不是**原子操作**。

这意味着，它不能作为一个单独的、不可分割的操作去执行。相反，自增操作是3个离散操作的简写形式：获得当前值，加1，写回新值。这是一个“**读-改-写** (read-modify-write)”操作的实例，其中，结果的状态衍生自它先前的状态。

第6页的图1.1演示了两个线程在缺乏同步的条件下，试图同时更新一个计数器时所发生的事情。假设计数器的初始值为9，在某些特殊的分时里，每个线程都将读它的值，并看到值是9，然后同时加1，最后都将counter设置为10。很明显，这不是我们期望发生的事情：一次递增操作凭空消失了，一次命中计数被永久地取消了。

你可能会想命中计数上的轻微错误所导致准确率的误差，在基于Web的服务中是可以接受的。有时的确如此。但是如果计数器用于生成序列或对象唯一的标识符，多重调用返回相同的结果会导致严重的数据完整性问题³。在一些偶发时段里，出现错误结果的可能性对于并发程序而言非常重要，以致于专门用一个名词来描述它们：**竞争条件**。

2.2.1 竞争条件

UnsafeCountingFactorizer中存在数个**竞争条件**，导致其结果是不可靠的。当计算的正确性依赖于运行时中相关的时序或者多线程的交替时，会产生竞争条件；换句话说，想得到正确的答案，要依赖于“幸运”的时序⁴。最常见的一种竞争条件是“**检查再运行** (check-then-act)”，使用一个潜在的过期值作为决定下一步操作的依据。

在现实生活中，我们也常常会遇到竞争条件。比如说你打算中午到University Avenue的星巴克去见一个朋友。不过当你到达这里后，发现这里有**两个**星巴克，而你并不确定和朋友约在哪个星巴克见面。12:10的时候，你还没有在星巴克A见到你的朋友，于是你向星巴克B走去，看看他是否在那里，可惜也不在。这存在几种可能性：你的朋友迟到了，没有出现在任何一个星巴克中；你的朋友在你离开后到达了星巴克A；你的朋友**曾经**在星巴克B，但是为了找你，现在在去星巴克A的途中。接下来，让我们假设最糟糕的情况，不妨称其为最终的可能性。现在是12:15，你们已经去过了所有的星巴克，你们也想知道对方是否已经等在那里了。你现在做什么？回到另一个星巴克？你打算走上多少个来回？

³ UnsafeSequence与UnsafeCountingFactorizer所采用的方法还存在其他严重的问题，包括出现过期数据的可能性(3.1.1节)。

⁴ 术语竞争条件通常会和一个相关的术语数据竞争(data race)混淆。数据竞争出现于没有使用同步来协调所有那些共享的非final域访问的情况。一个线程写入一个变量，可以被另一个线程读取；一个线程读取刚刚被另一个线程写入的变量，如果两个线程都没有使用同步，你将会处于数据竞争的风险中。处于数据竞争下的代码，在Java存储模型中并没有明确定义的语义。不是所有竞争条件都是数据竞争，同样不是所有的数据竞争都是竞争条件，不过他们都会引起并发程序以不可预期的方式失败。UnsafeCountingFactorizer中既有竞争条件，又有数据竞争，关于数据竞争的更多细节，参见第16章。

除非你和你的朋友间有某些约定，否则你们会无精打采，倍感沮丧地在 University Avenue 走上一整天。

“我打起精神沿街走，看朋友是不是在另一处。”这种作法的问题在于当你沿街走时，你的朋友可能已经离开了。你在星巴克 A 寻找朋友，发现“他不在这里”，然后继续寻找他。你在星巴克 B 可以做完全相同的事，但**不是在做同时**。沿街走要花几分钟，在这几分钟的时间里，**系统状态可能已经更改**。

星巴克的例子阐释了竞争条件的诱因：为获取期望的结果（见到你的朋友），需要依赖相关的事件的分时（当你到达一家星巴克时，会在这里等上多久而后离开，等等）。只要你一走出星巴克 A 的大门，“朋友不在此处”的观察结果就会潜在地变为无效结果：你的朋友可能已经从后门走进来而你并不知道。这些无效的观察结果，指出了大多数竞争条件的特点——使用潜在的过期观察值来作决策或执行计算。这种竞争条件被称作**检查再运行**（check-then-act）：你观察到一些事情为真（文件 X 不存在），然后（then）基于你的观察去执行一些动作（创建文件 X）；但事实上，从观察到执行操作的这段时间内，观察结果可能已经无效了（有人在此期间创建了文件 X），从而引发错误（非预期的异常，重写数据或者破坏文件）。

2.2.2 示例：惰性初始化中的竞争条件

检查再运行的常见用法是**惰性初始化**（lazy initialization）。惰性初始化的目的是延迟对象的初始化，直到程序真正使用它，同时确保它只初始化一次。清单 2.3 示范了惰性初始化的用法，getInstance 方法首先检查 ExpensiveObject 是否已被初始化，如果是，返回已经存在的实例；否则就创建一个新实例，然后保留它的引用，最后将它返回。由此，在这之后的调用可以避免执行代价昂贵的代码路径。

清单 2.3 惰性初始化中存在竞争条件（**不要这样做**）

```
@NotThreadSafe
public class LazyInitRace {
    private ExpensiveObject instance = null;

    public ExpensiveObject getInstance() {
        if (instance == null)
            instance = new ExpensiveObject();
        return instance;
    }
}
```



LazyInitRace 中的竞争条件会破坏其正确性。比如说线程 A 和 B 同时执行 getInstance, A 看到 instance 是 null, 并实例化一个新的 ExpensiveObject。同时 B 也在检查 instance 是否为 null。此时此刻的 instance 是否为 null, 这依赖于时序, 这是无法预期的。它包括调度的无常性, 以及 A 初始化 ExpensiveObject 并设置 instance 域的耗时。如果 B 检查到 instance 为 null, 两个 getInstance 的调用者会得到不同的结果。然而, 我们期望 getInstance 总是返回相同的实例。

UnsafeCountingFactorizer 中的命中计数操作中还存在另一种竞争条件“读-改-写”操作, 比如递增计数器, 它按照对象先前的状态来定义对象的状态转换。递增一个计数器, 你必须要知道先前值, **并且**要确保你在更新的过程中, 没有其他线程改变或使用计数器的值。

像大多数并发错误一样, 竞争条件并不**总是**导致失败: 还需要某些特殊的分时。但是竞争条件会引起严重的问题。如果 LazyInitRace 用于实例化一个应用级的注册器, 让它在多次调用中返回不同的实例, 会引起注册信息的丢失, 或者多个活动得到不一致的已注册对象集合的视图。如果 UnsafeSequence 用于为持久性框架生成实体标识符, 两个对象会由于相同的 ID 而消亡, 因为它们破坏了标识符的完整性约束。

2.2.3 复合操作

LazyInitRace 和 UnsafeCountingFactorizer 都包含一系列操作, 相对于在同一状态下的其他操作而言, 必须是**原子性**的或不可分割的。为了避免竞争条件, 必须阻止其他线程访问我们正在修改的变量, 让我们可以确保: 当其他线程想要查看或修改一个状态时, 必须在我们的线程开始之前或者完成之后, 而不能在操作过程中。

假设有操作 A 和 B, 如果从执行 A 的线程的角度看, 当其他线程执行 B 时, 要么 B 全部执行完成, 要么一点都没有执行, 这样 A 和 B 互为**原子操作**。一个**原子操作**是指: 该操作对于所有的操作, 包括它自己, 都满足前面描述的状态。

如果 UnsafeSequence 中的自增是原子操作, 那么就不会发生第 6 页图 1.1 所阐释的竞争条件。每次执行自增, 都会产生预期的结果, 即计数器准确地加 1。为了确保线程安全, “检查再运行”操作(如惰性初始化)和读-改-写操作(如自增)必须是原子操作。我们将“检查再运行”和读-改-写操作的全部执行过程看作是**复合操作**: 为了保证线程安全, 操作必须原子地执行。我们会在下一节考虑用 Java 内置的原子性机制——**锁**。现在,

我们先用其他方法修复这个问题——使用已有的线程安全类，如清单 2.4 的 CountingFactorizer 所示。

清单 2.4 Servlet 使用 AtomicLong 统计请求数

```
@ThreadSafe
public class CountingFactorizer implements Servlet {
    private final AtomicLong count = new AtomicLong(0);

    public long getCount() { return count.get(); }

    public void service(ServletRequest req, ServletResponse resp) {
        BigInteger i = extractFromRequest(req);
        BigInteger[] factors = factor(i);
        count.incrementAndGet();
        encodeIntoResponse(resp, factors);
    }
}
```

java.util.concurrent.atomic 包中包括了**原子变量**（atomic variable）类，这些类用来实现数字和对象引用的原子状态转换。把 long 类型的计数器替换为 AtomicLong 类型的，我们可以确保所有访问计数器状态的操作都是原子的⁵。计数器是线程安全的了，而计数器的状态就是 Servlet 的状态，所以我们的 Servlet 再次成为线程安全的了。

我们可以向 Factoring Servlet 中加入一个计数器，并利用已有的线程安全类（AtomicLong）管理计数器的状态，维护 Servlet 的线程安全性。当只向无状态类中加入**唯一**的状态元素，而这个状态完全被线程安全的对象所管理，那么新的类仍然是线程安全的。但是，正如我们在下一节所见的，状态的数量从一个增加到多个的情况，远远不像从 0 个增加到 1 个这么简单。

利用像 AtomicLong 这样已有的线程安全对象管理类的状态是非常实用的。相比于非线程安全对象，判断一个线程安全对象的可能状态和状态的转换要容易得多。这简化了维护和验证线程安全性的工作。

2.3 锁

通过使用线程安全对象来管理 Servlet 的全部状态，可以维护 Servlet 的线程安全性，这样我们只能在 Servlet 中加入一个状态变量。但是我们如果想加入更多的状态，可以仅

⁵ CountingFactorizer 调用 incrementAndGet 不但使计数值递增，同时还会返回递增的结果；不过这里忽略了返回值。

仅加入更多的线程安全的状态变量吗？

想象下面的情形：我们缓存最新的计算结果，以应对两个连续的客户请求相同的数字进行因数分解，希望由此提高 Servlet 的性能。（这未必是一个有效的缓存策略；在 5.6 节我们会提供一个更好的。）要实现这个策略，我们需要记住两件事：最新请求的数字和它的因数。

我们在前面曾经是用 `AtomicLong`，以线程安全的方式管理计数量的状态；我们还可以使用同系的 `AtomicReference`⁶ 类型管理缓存的数字和它的因数吗？清单 2.5 中的 `UnsafeCachingFactorizer` 作了这种尝试：

清单 2.5 没有正确原子化的 Servlet 试图缓存它的最新结果（不要这样做）

```
@NotThreadSafe
public class UnsafeCachingFactorizer implements Servlet {
    private final AtomicReference<BigInteger> lastNumber
        = new AtomicReference<BigInteger>();
    private final AtomicReference<BigInteger[]> lastFactors
        = new AtomicReference<BigInteger[]>();

    public void service(ServletRequest req, ServletResponse resp) {
        BigInteger i = extractFromRequest(req);
        if (i.equals(lastNumber.get()))
            encodeIntoResponse(resp, lastFactors.get());
        else {
            BigInteger[] factors = factor(i);
            lastNumber.set(i);
            lastFactors.set(factors);
            encodeIntoResponse(resp, factors);
        }
    }
}
```



很不幸，这种方法并不正确。尽管原子引用（atomic references）自身是线程安全的，不过 `UnsafeCachingFactorizer` 中存在竞争条件，导致它会产生错误的答案。

线程安全性的定义要求无论是多线程中的时序或交替操作，都要保证不破坏那些不变约束。`UnsafeCachingFactorizer` 的一个不变约束是：缓存在 `lastFactors` 中的各个因子的乘积应该等于缓存在 `lastNumber` 中的数值。只有遵守这个不变约束，我们的 Servlet 才是正确的。当一个不变约束涉及多个变量时，变量间不是彼此独立的：某个变量的值会

⁶ 正如 `AtomicLong` 是 `long` 和 `integer` 的线程安全的 holder 类，`AtomicReference` 是对象引用的线程安全 holder 类。原子变量（Atomic variable）以及它们的好处将在第 15 章介绍。

制约其他几个变量的值。因此，更新一个变量的时候，要在**同一原子操作**中更新其他几个。

在一些特殊的时序中，UnsafeCachingFactorizer 可能破坏这一不变约束。即使是用原子引用，并且每个 set 调用都是原子的，我们也无法保证会同时更新 lastNumber 和 lastFactors；当某个线程只修改了一个变量而另一个还没有开始修改时，其他线程将看到 Servlet 违反了不变约束，这样会形成一个程序漏洞。类似地，也不能保证每个线程都会同时获得两个值：当线程 A 尝试获取两个值的时间里，线程 B 可能已经修改了它们，线程 A 过后会观察到 Servlet 违反了不变约束。

为了保护状态的一致性，要在单一的原子操作中更新相互关联的状态变量。

2.3.1 内部锁

Java 提供了强制原子性的内置锁机制：synchronized 块。（第 3 章将介绍锁和同步机制的另一个重要方面：可见性）一个 synchronized 块有两部分：**锁**对象的引用，以及这个锁保护的代码块。synchronized 方法是对跨越了整个方法体的 synchronized 块的简短描述，至于 synchronized 方法的锁，就是该方法所在的对象本身。（静态的 synchronized 方法从 Class 对象上获取锁。）

```
synchronized (lock) {  
    // 访问或修改被锁保护的共享状态  
}
```

每个 Java 对象都可以隐式地扮演一个用于同步的锁的角色；这些内置的锁被称作**内部锁**（intrinsic locks）或**监视器锁**（monitor locks）。执行线程进入 synchronized 块之前会自动获得锁；而无论通过正常控制路径退出，还是从块中抛出异常，线程都在放弃对 synchronized 块的控制时自动释放锁。获得内部锁的唯一途径是：进入这个内部锁保护的同步块或方法。

内部锁在 Java 中扮演了**互斥锁**（mutual exclusion lock，也称作 mutex）的角色，意味着至多只有一个线程可以拥有锁，当线程 A 尝试请求一个被线程 B 占有的锁时，线程 A 必须等待或者**阻塞**，直到 B 释放它。如果 B 永远不释放锁，A 将永远等下去。

同一时间，只能有一个线程可以运行特定锁保护的代码块，因此，由同一个锁保护的 synchronized 块会各自原子地执行，不会相互干扰。在并发的上下文中，原子性的含义与它在事务性应用中相同——一组语句（statements）作为单独的，不可分割的单元运行。

执行 `synchronized` 块的线程，不可能看到会有其他线程能同时执行由同一个锁保护的 `synchronized` 块。

同步机制简化了恢复 `factoring servlet` 线程安全的工作。清单 2.6 中，我们将 `service` 方法声明为 `synchronized`，所以同一时间内只有一个线程可以进入 `service` 方法。现在 `SynchronizedFactorizer` 又是线程安全的了；但是这种方法过于极端，它完全禁止多个用户同时使用 `factoring servlet`——这导致糟糕的、令人无法接受的响应性。这个问题——一个性能问题，而非线程安全问题——将在 2.5 节中深入讨论。

清单 2.6 缓存了最新结果的 `servlet`，但响应性令人无法接受（不要这样做）

```
@ThreadSafe
public class SynchronizedFactorizer implements Servlet {
    @GuardedBy("this") private BigInteger lastNumber;
    @GuardedBy("this") private BigInteger[] lastFactors;

    public synchronized void service(ServletRequest req,
                                    ServletResponse resp) {
        BigInteger i = extractFromRequest(req);
        if (i.equals(lastNumber))
            encodeIntoResponse(resp, lastFactors);
        else {
            BigInteger[] factors = factor(i);
            lastNumber = i;
            lastFactors = factors;
            encodeIntoResponse(resp, factors);
        }
    }
}
```



2.3.2 重进入（Reentrancy）

当一个线程请求其他线程已经占有的锁时，请求线程将被阻塞。然而内部锁是**可重进入**的，因此线程在试图获得它自己占有的锁时，请求会成功。重进入意味着所的请求是基于“每线程（`per-thread`）”，而不是基于“每调用（`per-invocation`）”的⁷。重进入的实现是通过为每个锁关联一个请求计数（`acquisition count`）和一个占有它的线程。当计数为 0 时，认为锁是未被占有的。线程请求一个未被占有的锁时，JVM 将记录锁的占有者，并且将请求计数置为 1。如果同一线程再次请求这个锁，计数将递增；每次占用线程退出同

⁷ 这与 `pthread`（`POSIX threads`）的互斥锁的默认锁行为不同，它的授权是基于“每调用”的。

步块，计数器值将递减。直到计数器达到 0 时，锁被释放。

重进入方便了锁行为的封装，因此简化了面向对象并发代码的开发。清单 2.7 中，子类覆写了父类 `synchronized` 类型的方法，并调用父类中的方法。如果没有可重入的锁，这段看上去很自然的代码就会产生死锁。因为 `Widget` 和 `LoggingWidget` 中的 `doSomething` 方法都是 `synchronized` 类型的，都会在处理前试图获得 `Widget` 的锁。倘若内部锁不是可重入的，`super.doSomething` 的调用者就永远无法得到 `Widget` 的锁，因为锁已经被占有，导致线程会永久地延迟，等待着一个永远无法获得的锁。重进入帮助我们避免了这种死锁。

清单 2.7 如果内部锁不是可重入的，代码将死锁

```
public class Widget {
    public synchronized void doSomething() {
        ...
    }
}

public class LoggingWidget extends Widget {
    public synchronized void doSomething() {
        System.out.println(toString() + ": calling doSomething");
        super.doSomething();
    }
}
```

2.4 用锁来保护状态

因为锁使得线程能够串行地（`serialized`⁸）访问它所保护的代码路径，所以我们可以用锁来创建相关的协议，以保证线程对共享状态的独占访问。只要始终如一地遵循这些协议，就能够确保状态的一致性。

操作共享状态的复合操作必须是原子的，以避免竞争条件，比如递增命中计数器（读-改-写）或者惰性初始化（检查再运行）。复合操作会在**完整的运行期间**占有锁，以确保其行为是原子的。然而，仅仅用 `synchronized` 块包装复合操作是不够的；如果用同步来协调访问变量，**每次访问变量时**都需要同步。进一步讲，用锁来协调访问变量时，每次访问变量都需要用**同一个锁**。

⁸ 对象的串行（Serializing）访问与对象的序列化（serialization，将对象转化为字节流）毫无关系；不间断访问意味着线程依次独占地访问对象，而不是并发访问。

一种常见的错误观念认为只有**写入**共享变量时才需要同步；**其实并非如此**（其中的原因会在 3.1 节说明）。

对于每个可被多个线程访问的可变状态变量，如果所有访问它的线程在执行时都占有同一个锁，这种情况下，我们称这个变量是由这个锁保护的。

清单 2.6 的 `SynchronizedFactorizer` 中的 `lastNumber` 和 `lastFactors` 是由 `Servlet` 对象的内部锁保护的。这一点已由 `@GuardedBy Annotation` 进行了文档化。

对象的内部锁与它的状态之间没有内在的关系。尽管大多数类普遍使用这样一种非常有效的锁机制：用对象的内部锁来保护所有的域，然而这并不是必需的。即使获得了与对象关联的锁也**不能**阻止其他线程访问这个对象——获得对象的锁后，唯一可以做的事情是阻止其他线程再获得相同的锁。作为一种便利，每个对象都有一个内部锁，所以你不需要显式地创建锁对象⁹。你可以构造自己的**锁协议**或**同步策略**，使你可以安全地访问共享状态，并且贯穿程序都始终如一地使用它们。

每个共享的可变变量都需要由唯一一个确定的锁保护。而维护者应该清楚这个锁。

一种常见的锁规则是在对象内部封装所有的可变状态，通过对象的内部锁来同步任何访问可变状态的代码路径，保护它在并发访问中的安全。很多的线程安全类都是这个模式，例如 `Vector` 和其他同步的容器（`collection`）类。这种情况下，对象状态中的一切变量都被对象的内部锁保护。然而这种锁机制并没有什么特殊的，编译器或运行时都不会强制要求这种（或者其他任何一种）锁模式¹⁰。如果添加新的方法或者代码路径而忘记使用锁，这种锁协议也很容易被破坏。

并不是所有数据都需要锁的保护——只有那些被多个线程访问的可变数据。在第 1 章我们看到，添加一个简单的异步事件（比如 `TimerTask`）后，尤其是当你的程序并未良好封装时，整个程序是如何引入线程安全的需求的。考虑一个处理大量数据的单线程化的程序，由于没有跨线程共享的数据，所以不需要同步。现在想象你打算添加新特性，周期性地为进度创建快照，这样当程序崩溃或者必须停止时，不必从起点重新启动。你选择 `TimerTask`，设置每十分钟触发一次，将程序状态保存在文件中，从而完成这个功能。然

⁹ 回顾过去，这一设计决策相当糟糕：不仅会引起混淆，而且强迫 JVM 的实现者要权衡对象的大小与锁的性能。

¹⁰ 像 `FindBugs` 这种代码核查工具如果发现访问某个变量时，线程并不总是占有锁，就会指出这是一个 bug。

而另一个线程（Timer 的管理线程）会调用 TimerTask，因此现在有两个线程会访问存储在快照中的任意数据：程序主线程与 Timer 线程。这意味着访问程序的状态时，不仅 TimerTask 的代码要使用同步，程序中其余的访问相同数据的代码路径也要同步。一个原本不需要同步的程序，现在要在整个程序中贯穿使用同步。

锁保护的变量，意味着**每一次**访问变量时都要获得该锁，确保在同一时刻只有一个线程可以访问这个变量。若类的不变约束涉及多个状态变量，那么另外还需要一个附加需求：每个参与到不变约束的变量由**同一个**锁守护。这样你可以在一个单一的原子操作中访问或者更改它们，从而保证了不变约束。SynchronizedFactorizer 示范了这条规则：servlet 对象的内部锁保护缓存的 number 与缓存的 factors。

对于每一个涉及多个变量的不变约束，需要**同一个**锁保护其**所有的**变量。

既然同步可以避免竞争条件，为什么不将每个方法都声明为 synchronized 类型？如此武断地使用同步，可能导致程序中使用的同步过多或过少。如同 Vector 这样，仅仅同步它每个方法，并不足以确保在 Vector 上执行的复合操作是原子的：

```
if (!vector.contains(element))
    vector.add(element);
```

虽然 contains 和 add 都是原子的，但在尝试“缺少即加入（put-if-absent）”操作的过程中仍然存在竞争条件。虽然同步方法确保了不可分割操作的原子性，但是把多个操作整合到一个复合操作时，还是需要额外的锁。（关于向线程安全对象中安全地添加额外的原子操作的技术，参见 4.4 节。）同时，同步每个方法还会导致活跃度（liveness）或性能（performance）的问题，正如我们在 SynchronizedFactorizer 中看到的那样。

2.5 活跃度与性能

在 UnsafeCachingFactorizer 中，我们引入一些缓存到 factoring servlet 中，希望提高它的性能。缓存需要一些共享状态，需要依次同步来维护这些状态的完整性。但是，在 SynchronizedFactorizer 中，Servlet 在我们的同步下运行，性能变得很糟糕。SynchronizedFactorizer 的同步策略是：用 Servlet 对象的内部锁保护每一个状态变量。这

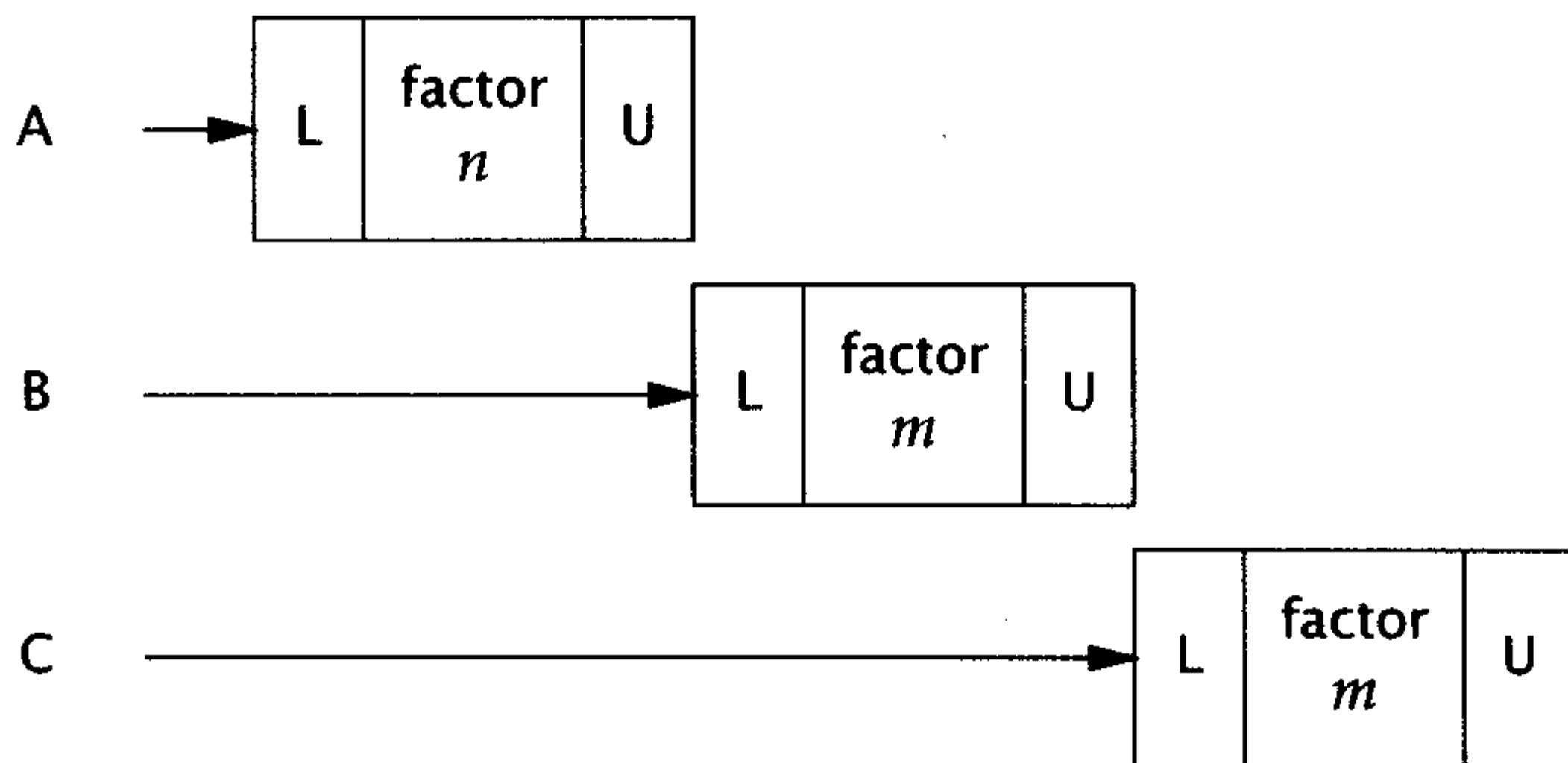


图 2.1 SynchronizedFactorizer 的弱并发

个策略是通过同步整个 Service 方法实现的。这种简单粗糙的方法虽然使我们重获安全性，但是代价高昂。

Service 方法声明为 Synchronized，因此每次只能有一个线程执行它。这违背了 Servlet 框架的使用初衷——Servlet 可以同时处理多个请求——并且当负载过高时会引起用户的不满。如果 Servlet 正忙于处理一个大数的因式分解，那么在它可以处理一个新的运算开始前，其他用户必须等待，直到当前的请求完成。这种情况下，在多 CPU 系统中，即使负载很高，仍然会有处理器处于空闲。无论如何，即使是运行时间短的请求，比如请求缓存的值，仍然可能耗费难以预期长的时间，因为它们必须等待前一个耗时的请求完成。

图 2.1 演示了多个请求到达同步的 Factoring Servlet 时所发生的事情：这些请求排队等候并依次被处理。我们把这种 Web 应用的运行方式描述为**弱并发**（poor concurrency）的一种表现：限制并发调用数量的，并非可用的处理器资源，而恰恰是应用程序自身的结构。幸运的是，通过缩小 synchronized 块的范围来维护线程安全性，我们很容易提升 Servlet 的并发性。你应该谨慎地控制 synchronized 块不要过小；你不可以将一个原子操作分解到多个 synchronized 块中。不过你应该尽量从 synchronized 块中分离耗时的且不影响共享状态的操作。这样即使在耗时操作的执行过程中，也不会阻止其他线程访问共享状态。

清单 2.8 的 CachedFactorizer 重新构造了 servlet：使用两个分离的 synchronized 块，每个都限制在很短的代码段中。第一个 synchronized 块保护着检查再运行的操作，以检查我们是否可以返回缓存的结果；另一个保证缓存 number 和 factors 的同步更新。另外我们还重新引入了命中计数以及一个“cache hit”计数器，并在第一个的 synchronized 块内更新它们。由于这两个计数器构成了共享的、可变的、状态，我们必须在访问它们的每处都使用同步。synchronized 块之外的代码独享地操作本地（基于栈的）变量，这些

变量不被跨线程地共享，因此不需要同步。

清单 2.8 缓存最新请求和结果的 servlet

```
@ThreadSafe
public class CachedFactorizer implements Servlet {
    @GuardedBy("this") private BigInteger lastNumber;
    @GuardedBy("this") private BigInteger[] lastFactors;
    @GuardedBy("this") private long hits;
    @GuardedBy("this") private long cacheHits;

    public synchronized long getHits() { return hits; }
    public synchronized double getCacheHitRatio() {
        return (double) cacheHits / (double) hits;
    }

    public void service(ServletRequest req, ServletResponse resp) {
        BigInteger i = extractFromRequest(req);
        BigInteger[] factors = null;
        synchronized (this) {
            ++hits;
            if (i.equals(lastNumber)) {
                ++cacheHits;
                factors = lastFactors.clone();
            }
        }
        if (factors == null) {
            factors = factor(i);
            synchronized (this) {
                lastNumber = i;
                lastFactors = factors.clone();
            }
        }
        encodeIntoResponse(resp, factors);
    }
}
```

CachedFactorizer 没有使用 AtomicLong 类型的命中计数，而是又重新使用了一个 long 类型的域。在这里使用 AtomicLong 是安全的，但是却得不到比在 CountingFactorizer 中更多的好处。原子变量可以保证单一变量的操作是原子的，然而我们已经使用 synchronized 块构造了原子操作。使用两种不同的同步机制会引起混淆，而且性能与安全也不能从中得到额外的好处。

重新构造后的 CachedFactorizer 提供了简单性（同步整个方法）与并发性（同步尽

可能短的代码路径)之间的平衡。请求与释放锁的操作需要开销,所以将 `synchronized` 块分解得**过于**琐碎(比如将 `++hit` 分解到它自身的 `synchronized` 块中)是不合理的,即使这样做是为了获得更好的原子性。当访问状态变量或者执行复合操作期间, `CachedFactorizer` 会占有锁,但是执行潜在耗时的因数分解之前,它会释放锁。这样既保护了线程安全性,也不会过多地影响并发性;每个 `synchronized` 块的代码路径已经“足够短”了。

决定 `synchronized` 块的大小需要权衡各种设计要求,包括安全性(这是我们决不能妥协的)、简单性和性能。有时简单性与性能会彼此冲突,然而正如 `CachedFactorizer` 示范的那样,通常都能够从中找到一个合理的平衡。

通常简单性与性能之间是相互牵制的。实现一个同步策略时,不要过早地为了性能而牺牲简单性(这是对安全性潜在的妥协)。

当使用锁的时候,你应该清楚块中的代码的功能,以及它的执行过程是否会很耗时。无论是作运算密集型的操作,还是在执行一个可能存在潜在阻塞的操作,如果线程长时间地占有锁,就会引起活跃度与性能风险的问题。

有些耗时的计算或操作,比如网络或控制台 I/O,难以快速完成。执行这些操作期间不要占有锁。

共享对象

Sharing Objects

我们在第 2 章开始的时候提到过，编写正确的并发程序的关键在于对共享的、可变的状态进行访问管理。上一章是关于使用同步来避免多个线程在同一时间访问同一数据的；这一章详细讲述共享和发布对象的技术，使多个线程能够安全地访问他们。这两章合在一起，可以作为构建线程安全类，以及使用 `java.util.concurrent` 类库构造安全的并发应用程序的基础。

我们已经看到了 `synchronized` 关键字是如何阻塞执行的，也看到了方法中如何确保操作执行的原子化，但是这里存在一个普遍的误解，认为 `synchronized` 仅仅用于原子操作或者划定“临界区”。同步同样还具有另一个重要、微妙的方面：内存可见性。我们不仅希望能够避免一个线程修改其他线程正在使用的对象的状态，而且希望确保当一个线程修改了对象的状态后，其他的线程能够真正看到改变。但是没有同步，这些可能都不会发生。你可以使用显式的同步，或者利用内置于类库中的同步机制，来保证对象的安全发布。

3.1 可见性

可见性是微妙的，这是因为可能发生错误的事情总是与直觉大相径庭。在一个单线程化的环境里，如果向一个变量先写入值，然后在没有写干涉的情况下读取这个变量，你希望能得到相同的返回值。这看起来是很自然的。但是当读和写发生在不同的线程中时，情况却根本不是这样——这种说法在刚开始听到时令人难以接受。通常，不能保证读线程及时地读取其他线程写入的值，甚至可以说根本不可能。为了确保跨线程写入的内存可见性，你必须使用同步机制。

在清单 3.1 中，`NoVisibility` 阐明了多个线程在没有同步的情况下共享数据所引发的错误。这里有主线程和读线程两个线程访问共享变量 `ready` 和 `number`。主线程启动读线程，然后把 `number` 的值设置为 42，`ready` 赋值为 `true`。读线程进行循环，直到发现

ready 的值变为 true，然后打印出 number 的值。虽然看起来 NoVisibility 会输出 42，但事实上，它很有可能打印出 0，或者根本不会终止。这是因为它没有使用恰当的同步机制，没能保证主线程写入 ready 和 number 的值对读线程是可见的。

清单 3.1 在没有同步的情况下共享变量（不要这样做）

```
public class NoVisibility {  
    private static boolean ready;  
    private static int number;  
  
    private static class ReaderThread extends Thread {  
        public void run() {  
            while (!ready)  
                Thread.yield();  
            System.out.println(number);  
        }  
    }  
  
    public static void main(String[] args) {  
        new ReaderThread().start();  
        number = 42;  
        ready = true;  
    }  
}
```



NoVisibility 可能会一直保持循环，因为对于读线程来说，ready 的值可能永远不可见。甚至更奇怪的现象是，NoVisibility 可能会打印 0，因为早在对 number 赋值之前，主线程就已经写入 ready 并使之对读取线程可见，这是一种“**重排序 (reordering)**”现象。在单个线程中，只要重排序不会对结果产生影响，那么就不能保证其中的操作一定按照程序写定的顺序执行——即使重排序对于其他线程来说会产生明显的影响¹。当主线程写入 number，然后在没有同步的情况下继续执行，读线程看到的顺序可能与发生写入的顺序正好相反，或者完全不同。

¹ 这看起来可能是个失败的设计，但是它意味着Java虚拟机能够充分地利用现代硬件的多核处理器的性能。例如在没有同步的情况下，Java存储模型允许编译器重排序操作，在寄存器中缓存数值，还允许CPU重排序，并在处理器特有的缓存中缓存数值。关于更多细节，参见第16章。

在没有同步的情况下，编译器，处理器，运行时安排操作的执行顺序可能完全出人意料。在没有进行适当同步的多线程程序中，尝试推断那些“必然”发生在内存中的动作时，你总是会判断错误。

NoVisibility 是一个我们能想到的最简单的并发程序——两个线程，两个共享变量。但是仍然很容易得到错误的结果，包括程序的运行以及程序的终止。对那些没有恰当同步的并发程序进行分析和推断是十分困难的。

这看起来有点可怕，的确是这样。幸运的是，有一个简单的方法来避免这些复杂的问题：只要数据需要被跨线程共享，就进行**恰当的同步**。

3.1.1 过期数据

NoVisibility 演示了一种没有恰当同步的程序，它能够引起意外的后果：过期数据。当读线程检查 ready 变量时，它可能看到一个过期的值。除非每一次访问变量都是同步的，否则很可能得到变量的过期值。更坏的情况是，过期既不会发生在全部变量上，也不会完全不出现：一个线程可能会得到一个变量最新的值，但是也可能得到另一个变量先前写入的过期值。

当食物不新鲜的时候，它通常还是可以食用的——只不过缺少了一些美味。但是陈旧的、过期的数据可能更加危险。虽然一个过期的数据应用到 Web 程序的计数器时可能还不至于太危险²，但是过期值能够引起严重的安全错误，或者带来生死攸关的失败。在 NoVisibility 中，过期数据可能导致它打印错误数值，或者程序无法终止。过期数据可能会使对象引用中的数据更加复杂，比如链指针在链表中的实现。过期数据还可能引发严重且混乱的错误，比如意外的异常，脏的数据结构，错误的计算和无限的循环。

清单 3.2 的 MutableInteger 并不是线程安全的，因为 get 和 set 都访问了 value 域，却没有进行同步。在众多危害中，以下情况对过期数据尤为敏感：如果一个线程调用了 set，而另一个线程此时正在调用 get，它可能就看不到更新的数据了。

我们可以通过同步化 getter 和 setter，使 MutableInteger 成为线程安全的，就像在清单 3.3 的 SynchronizedInteger 所表现的那样。仅仅是同步的 setter 是不够的：调用 get 的线程仍然能够看见过期值。

² 在没有同步的情况下读取数据类似于数据库中使用 READ_UNCOMMITTED 隔离级别，这时你更愿意用准确性来交换性能。然而，在这种情况下，非同步的读取失掉了更大的准确度，因为一个共享变量的可见值很容易就变成过期的数据。

清单 3.2 非线程安全的可变整数访问器

```
@NotThreadSafe
public class MutableInteger {
    private int value;

    public int get() { return value; }
    public void set(int value) { this.value = value; }
}
```



清单 3.3 线程安全的可变整数访问器

```
@ThreadSafe
public class SynchronizedInteger {
    @GuardedBy("this") private int value;

    public synchronized int get() { return value; }
    public synchronized void set(int value) { this.value = value; }
}
```

3.1.2 非原子的 64 位操作

当一个线程在没有同步的情况下读取变量，它可能会得到一个过期值。但是至少它可以看到某个线程在那里设定的一个真实数值，而不是一个凭空而来的值。这样的安全保证被称为是**最低限的安全性**（out-of-thin-air safety）。

最低限的安全性应用于所有的变量，除了一个例外：没有声明为 `volatile` 的 64 位数值变量（`double` 和 `long`）（参见 3.1.4 节）。Java 存储模型要求获取和存储操作都为原子的，但是对于非 `volatile` 的 `long` 和 `double` 变量，JVM 允许将 64 位的读或写划分为两个 32 位的操作。如果读和写发生在不同的线程，这种情况读取一个非 `volatile` 类型 `long` 就可能会出现得到一个值的高 32 位和另一个值的低 32 位³。因此，即使你并不关心过期数据，但仅仅在多线程程序中使用共享的、可变的 `long` 和 `double` 变量也可能是不安全的，除非将它们声明为 `volatile` 类型，或者用锁保护起来。

3.1.3 锁和可见性

内置锁可以个用来确保一个线程以某种可预见的方式看到另一个线程的影响，就像图 3.1 说明的。当线程 A 执行一个同步块时，线程 B 也随后进入了被同一个锁监视的同步块中，这时可以保证，在锁释放之前对 A 可见的变量的值，B 获得锁之后同样是可见的。换

³ 当Java虚拟机的规范完成时，很多主流的处理器架构还不能有效地支持64位算数原子操作。

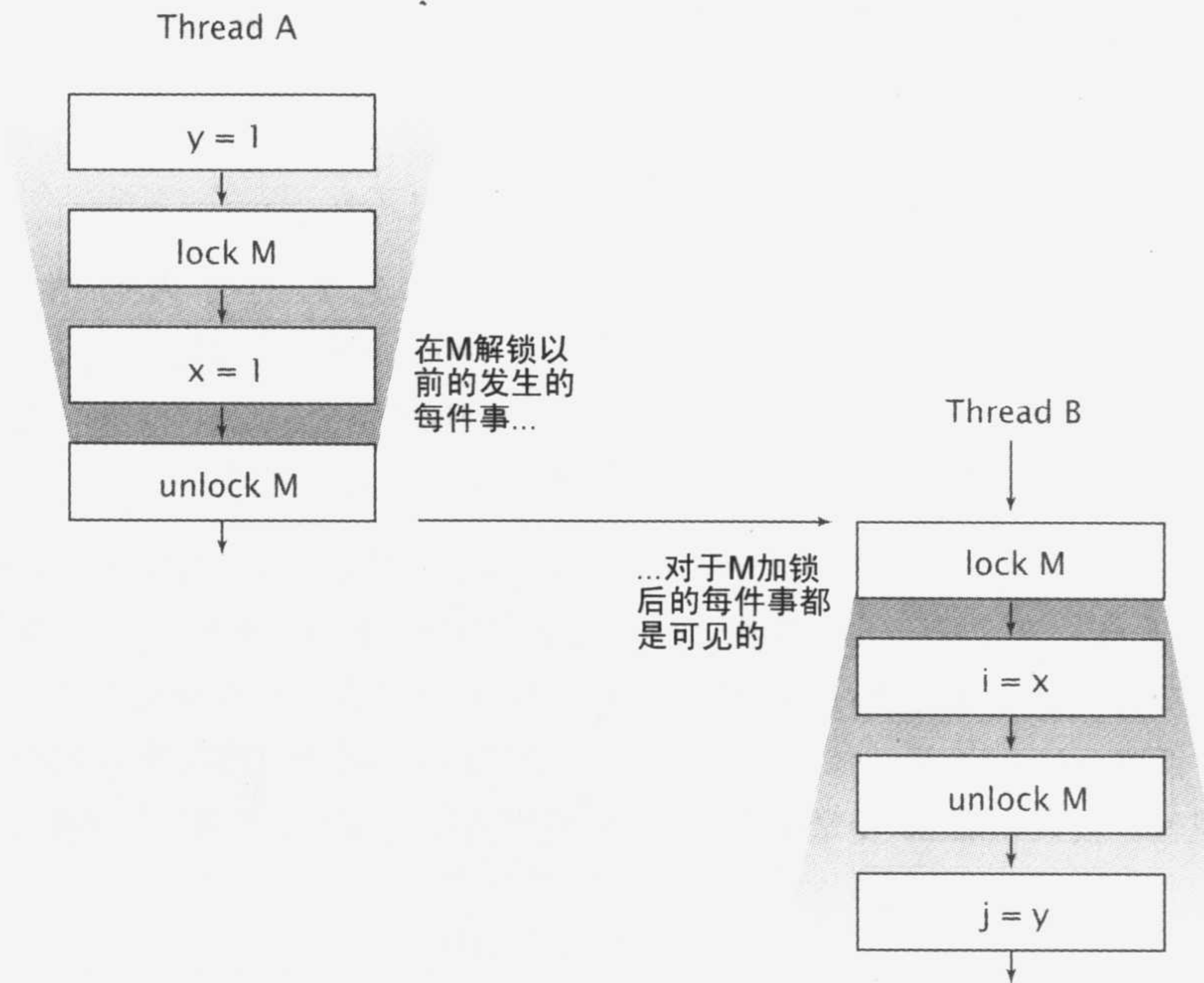


图 3.1 同步对可见性的保证

句话说,当 *B* 执行到与 *A* 相同的锁监视的同步块时,*A* 在同步块之中或之前所做的每件事,对 *B* 都是可见的。**如果没有同步,就没有这样的保证。**

当访问一个共享的可变变量时,为什么要求所有线程由同一个锁进行同步,我们现在可以给出另一个理由——为了保证一个线程对数值进行的写入,其他线程也都可见。另一方面,如果一个线程在没有恰当地使用锁的情况下读取了变量,那么这个变量很可能是一个过期的数据。

锁不仅仅是关于同步与互斥的,也是关于内存可见的。为了保证所有线程都能够看到共享的、可变变量的最新值,读取和写入线程必须使用公共的锁进行同步。

3.1.4 Volatile 变量

Java 语言也提供了其他的选择,即一种同步的弱形式: `volatile` 变量。它确保对一个变

量的更新以可预见的方式告知其他的线程。当一个域声明为 `volatile` 类型后，编译器与运行时监视这个变量：它是共享的，而且对它的操作不会与其他的内存操作一起被重排序。`volatile` 变量不会缓存在寄存器或者缓存在对其他处理器隐藏的地方。所以，读一个 `volatile` 类型的变量时，总会返回由某一线程所写入的最新值。

一个理解 `volatile` 变量的好方法是：想象它们的行为与清单 3.3 的 `SynchronizedInteger` 类大致相似，只不过用 `get` 和 `set` 方法⁴取代了对 `volatile` 变量的读写操作。然而访问 `volatile` 变量的操作不会加锁，也就不会引起执行线程的阻塞，这使得 `volatile` 变量相对于 `synchronized` 而言，只是轻量级的同步机制⁵。

`volatile` 变量对可见性的影响所产生的价值远远高于变量本身。线程 A 向 `volatile` 变量写入值，随后线程 B 读取该变量，所有 A 执行写操作前可见的变量的值，在 B 读取了 `volatile` 变量后，成为对 B 也是可见的。所以从内存可见性的角度看，写入 `volatile` 变量就像退出同步块，读取 `volatile` 变量就像进入同步块。但是我们并不推荐过度依赖 `volatile` 变量所提供的可见性。依赖 `volatile` 变量来控制状态可见性的代码，比使用锁的代码更脆弱，更难以理解。

只有当 `volatile` 变量能够简化实现和同步策略的验证时，才使用它们。当验证正确性必须推断可见性问题时，应该避免使用 `volatile` 变量。正确使用 `volatile` 变量的方式包括：用于确保它们所引用的对象状态的可见性，或者用于标识重要的生命周期事件（比如初始化或关闭）的发生。

清单 3.4 示范了一种 `volatile` 变量的典型应用：检查状态标记，以确定是否退出一个循环。在这个例子中，我们的不幸失眠的线程正试图通过传统的数绵羊方法入眠。为了让这个范例正常工作，`asleep` 标记就必须为 `volatile` 的，否则执行检查的线程不会注意到 `asleep` 已被其他线程修改⁶。我们也可以使用锁来代替 `volatile`，同样能够保证对 `asleep`

⁴ 这个类比不是很准确，`SynchronizedInteger` 对内存可见性的影响比 `volatile` 变量更强一些。参见第 16 章。

⁵ 对于当今大多数处理器架构而言，读取 `volatile` 变量的开销只比读取非 `volatile` 变量的开销略高。

⁶ 调试提示：对于服务器应用程序，确保无论是在开发阶段还是测试阶段，启动 JVM 时都使用 `-server` 命令行选项。server 模式的 JVM 会比 client 模式的 JVM 执行更多的优化，比如把没有在循环体中修改的变量提升到循环体外部；在开发环境（client 模式的 JVM）中可以工作的代码，可能会在部署环境（server 模式的 JVM）中失败。举个例子：在清单 3.4 中，如果我们忘记把 `asleep` 变量声明为 `volatile`，server 模式的 JVM 会将检查 `asleep` 的工作提升到循环体外部（将它变为一个无限循环），但是 client 模式的 JVM 不会这样做。开发环境中的无限循环的开销远小于将它置于生产环境中所产生的开销。

变量修改的可见性。但是锁会使代码变得复杂。

清单 3.4 数绵羊

```
volatile boolean asleep;
...
while (!asleep)
    countSomeSheep();
```

`volatile` 变量固然方便，但也存在限制。它们通常被当作标识完成、中断、状态的标记使用，比如清单 3.4 中的 `asleep` 标记。尽管 `volatile` 也可以用来标示其他类型的状态信息，但是决定这样做之前请格外小心。比如，`volatile` 的语义不足以使自增操作（`count++`）原子化，除非你能保证只有一个线程对变量执行写操作。（原子变量提供了“读-改-写”原子操作的支持，而且常被用作“更优的 `volatile` 变量”，参见第 15 章）。

加锁可以保证可见性与原子性；`volatile` 变量只能保证可见性。

只有满足了下面所有的标准后，你才能使用 `volatile` 变量：

- 写入变量时并不依赖变量的当前值；或者能够确保只有单一的线程修改变量的值；
- 变量不需要与其他的状态变量共同参与不变约束；
- 而且，访问变量时，没有其他的原因需要加锁。

3.2 发布和逸出

发布（publishing）一个对象的意思是使它能够被当前范围之外的代码所使用。比如将一个引用存储到其他代码可以访问的地方，在一个非私有的方法中返回这个引用，也可以把它传递到其他类的方法中。在很多情况下，我们需要确保对象及它们的内部状态不被暴露（publish）。在另外一些情况下，为了正当的使用目的，我们又的确希望发布一个对象，但是用线程安全的方法完成这些工作时，可能需要同步。如果变量发布了内部状态，就可能危及到封装性，并使程序难以维持稳定；如果发布对象时，它还没有完成构造，同样危及线程安全。一个对象在尚未准备好时就将它发布，这种情况称作逸出（escape）。3.5 节涵盖了关于如何安全发布对象的诸多模式；现在，让我们看看一个对象是如何逸出的。

最常见的发布对象的方式是将对象的引用存储到公共静态域中。任何类和线程都能看见这个域，正如清单 3.5 所示。Initialize 方法实例化一个新的 HashSet 实例，并通过将它存储到 knownSecrets 引用，从而发布了这个实例。

清单 3.5 发布对象

```
public static Set<Secret> knownSecrets;

public void initialize() {
    knownSecrets = new HashSet<Secret>();
}
```

发布一个对象还会间接地发布其他对象。如果你将一个 Secret 对象加入集合 knownSecrets 中，你就已经发布了这个对象，因为任何代码都可以遍历并获得新 Secret 对象的引用。类似地，从非私有方法中返回引用，也能发布返回的对象。清单 3.6 的 UnsafeStates 发布了包含有洲名缩写的数组，而这个数组本应是私有的。

清单 3.6 允许内部可变的数据逸出（不要这样做）

```
class UnsafeStates {
    private String[] states = new String[] {
        "AK", "AL" ...
    };
    public String[] getStates() { return states; }
}
```



以这种方式发布 states 会出问题。任何一个调用者都能修改它的内容。在这个例子中，数组 states 已经逸出了它所属的范围。这个本应是私有的数据，事实上已经变成公有了。

发布一个对象，同样也发布了该对象所有非私有域所引用的对象。更一般地，在一个已经发布的对象中，那些非私有域的引用链，和方法调用链中的可获得对象也都会被发布。

假设有一个类 C，从它的视角而言，一个**外部**（alien）方法的行为不是完全由 C 定义的。这包括在其他类中的方法，和 C 自身可被覆盖的方法（非 private 和非 final）。将一个对象传递给外部方法，相当于将这个对象发布了。因为实际上你并不知道它会激发哪些代码，也不知道外部方法是发布这个对象，还是只保留它的引用，以供其他线程使用。

其他线程是否真的会利用发布的引用做些什么并不重要，因为无论如何，被误用的风险还是存在的⁷。一旦一个对象逸出，你就要假设存在其他的类或线程可能误用它，无论

⁷ 如果有人盗取了你的密码并散布到 alt.free-password 新闻组中，你的机密就“逸出”了：无论是否有人已经（或尚未）使用这些个人信息恶作剧或做非法的事，你的账户都已经存在隐患了。（错误地）发布引用也会引发同种类型的风险。

是出于恶意还是粗心。这是使用封装的强制原因：封装使得程序的正确性分析变得更可行，而且更不易偶然地破坏设计约束。

最后一种发布对象和它的内部状态的机制是发布一个内部类实例，正如清单 3.7 的 ThisEscape 所表现的那样。当 ThisEscape 发布 EventListener 时，它也无条件地发布了封装（enclosing）ThisEscape 的实例，因为内引类（inner class instances）的实例包含了对封装实例隐含的引用。

清单 3.7 隐式地允许 this 引用逸出（**不要这样做**）

```
public class ThisEscape {
    public ThisEscape(EventSource source) {
        source.registerListener(
            new EventListener() {
                public void onEvent(Event e) {
                    doSomething(e);
                }
            });
    }
}
```



3.2.1 安全构建的实践

ThisEscape 演示了一种重要的逸出特例——this 引用在构造时逸出。发布的内部 EventListener 实例是一个封装的 ThisEscape 中的实例。但是对象只有通过构造函数返回后，才处于可预言的、稳定的状态，所以从构造函数内部发布的对象，只是一个未完成构造的对象。**甚至即使是在构造函数的最后一行发布的引用也是如此**。如果 this 引用在构造过程中逸出，这样的对象被认为是“**没有正确构建的（not properly constructed）**”⁸。

不要让 this 引用在构造期间逸出。

一个导致 this 引用在构造期间逸出的常见错误，是在构造函数中启动一个线程。当对象在构造函数中创建了一个线程时，无论是显式地（通过将它传给构造函数）还是隐式地（因为 Thread 或 Runnable 是所属对象的内部类），this 引用几乎总是被新线程共享。

⁸ 更明确的说法是，this 引用在构造函数完成前不会从线程中逸出。只要构造函数结束前没有其他线程使用 this 引用，this 引用就可以通过构造函数存储到某处。清单 3.8 的 SafeListener 使用了这项技术。

于是新的线程在所属对象完成构造前就能看见它。在构造函数中**创建**线程并没有错误，但是最好不要立即**启动**它。取而代之的是，发布一个 `start` 或 `initialize` 方法来启动对象拥有的线程。（更多关于服务生命周期的话题，参见第7章）。在构造函数中调用一个可覆盖的（那些既不是 `private`，也不是 `final` 的）实例方法同样会导致 `this` 引用在构造期间逸出。

如果想在构造函数中注册监听器或启动线程，你可以使用一个私有的构造函数和一个公共的工厂方法，这样避免了不正确的创建，正如清单 3.8 中的 `SafeListener` 所示的那样。

清单 3.8 使用工厂方法防止 `this` 引用在构造期间逸出

```
public class SafeListener {
    private final EventListener listener;

    private SafeListener() {
        listener = new EventListener() {
            public void onEvent(Event e) {
                doSomething(e);
            }
        };
    }

    public static SafeListener newInstance(EventSource source) {
        SafeListener safe = new SafeListener();
        source.registerListener(safe.listener);
        return safe;
    }
}
```

3.3 线程封闭

访问共享的、可变的数据要求使用同步。一个可以避免同步的方式就是**不共享数据**。如果数据仅在单线程中被访问，就不需要任何同步。线程封闭（Thread confinement）技术是实现线程安全的最简单的方式之一。当对象封闭在一个线程中时，这种做法会自动成为线程安全的，即使被封闭的对象本身并不是[CPJ 2.3.2]。

Swing 发展了线程封闭技术。Swing 的可视化组件和数据模型对象并不是线程安全的，它们是通过将它们限制到 Swing 的事件分发线程中，实现线程安全的。为了正确地使用 Swing，运行在不同于事件线程（event thread）的其他线程中的代码不应该访问这些对象（为了简化这些，Swing 提供了 `invokeLater` 机制，用于在事件线程中安排执行 `Runnable` 实例）。

很多 Swing 应用中的并发错误都滋生于从其他线程中错误地使用这些被限制的对象。

另一种常见的使用线程限制的应用程序是应用池化的 JDBC (Java Database Connectivity) Connection 对象。JDBC 规范并没有要求 Connection 对象是线程安全的⁹。然而在典型的服务器应用中，线程总是从池中获得一个 Connection 对象，并且用它处理一个单一的请求，最后把它归还。每个线程都会同步地处理大多数请求（比如 Servlet 请求或者 EJB (Enterprise Java Bean) 调用），而且在 Connection 对象在被归还前，池不会将它再分配给其他线程，因此，这种连接管理模式隐式地将 Connection 对象限制在处于请求处理期间的线程中。

正如语言并未提供强迫变量被锁保护的机制一样，语言也没有办法将对象限制在某一线程中。线程限制是你在程序设计中需要考虑的一个元素，它是在程序的实现中完成的。语言自身以及核心库提供了某些机制（本地变量和 ThreadLocal 类）有助于维护线程限制，尽管如此，程序员仍然要自己负责确保线程限制对象不会从它所在的线程中逸出。

3.3.1 Ad-hoc 线程限制

Ad-hoc 线程限制¹⁰是指维护线程限制性的任务全部落在实现上的这种情况。因为没有可见性修饰符与本地变量等语言特性协助将对象限制在目标线程上，所以这种方式是非常容易出错的。事实上，对于像 GUI 应用中的可视化组件或者数据模型这些线程限制对象，对它们的引用通常是公用域。

如果决定将一个像 GUI 这样特定的子系统实现为“单线程化”的子系统，通常就要使用线程限制技术。单线程化子系统有时所带来的简便性的好处远远胜过 ad-hoc 线程限制的易损性¹¹。

线程限制的一种特例是将它用于 volatile 变量。只要你确保只通过单一线程写入共享的 volatile 变量，那么在这些 volatile 变量上执行“读一改一写”操作就是安全的。在这种情况下，你就将修改操作限制在单一的线程中，从而阻止了竞争条件。并且，可见性保证 volatile 变量能够确保其他线程能看到最新的值。

鉴于 ad-hoc 线程限制固有的易损性，因此应该有节制地使用它。如果可能的话，用一种线程限制的强形式（栈限制或者 Thread Local）取代它。

⁹ 应用服务器提供了连接池实现的线程安全性。连接池在多线程访问中是必须的，所以一个非线程安全的实现毫无意义。

¹⁰ 译注：Ad-hoc的言外之意是“非正式的”。比如，在大学的餐厅中，你决定和某人谈谈业务，这就是一种“Ad-hoc”会谈。在这里，Ad-hoc是指未经过设计而得到的线程封闭行为。

¹¹ 避免死锁是将子系统设计为单线程化的另一个原因。这也是为什么大多数GUI框架都是单线程化的。单线程化子系统将在第9章详细介绍。

3.3.2 栈限制

栈限制是线程限制的一种特例，在栈限制中，只能通过本地变量才可以触及对象。正如封装使不变约束更容易被保持，本地变量使对象更容易被限制在线程本地中。本地变量本身就被限制在执行线程中；它们存在于执行线程栈。其他线程无法访问这个栈。栈限制（也称线程内部或者线程本地用法，但是不要与核心库类的 `ThreadLocal` 混淆）与 ad-hoc 线程限制相比，更易维护，更健壮。

像清单 3.9 中 `loadTheArk` 方法的 `numPairs`，对于这些基本（primitive）类型的本地变量，你无法尝试去利用栈限制。由于无法获得基本类型的引用，所以语言语义确保了基本本地变量总是线程封闭的。

清单 3.9 本地的基本类型和引用类型的变量的线程限制

```
public int loadTheArk(Collection<Animal> candidates) {
    SortedSet<Animal> animals;
    int numPairs = 0;
    Animal candidate = null;

    // animals 被限制在方法中，不要让它们逃出！
    animals = new TreeSet<Animal>(new SpeciesGenderComparator());
    animals.addAll(candidates);
    for (Animal a : animals) {
        if (candidate == null || !candidate.isPotentialMate(a))
            candidate = a;
        else {
            ark.load(new AnimalPair(candidate, a));
            ++numPairs;
            candidate = null;
        }
    }
    return numPairs;
}
```

维护对象引用的栈限制，需要程序员多一些付出，来确保引用的对象没有逸出。在 `loadTheArk` 中，我们实例化一个 `TreeSet` 对象 `animals`，并且保存了一个到 `animals` 中一个元素的引用。此时只有一个引用指向集合 `animals`，因此它被限制在保存本地变量的执行线程中。但是，倘若我们发布了到集合 `animals`（或者其他任何内部数据）的引用，那么将会破坏限制性，也导致了 `animals` 对象的逸出。

在线程内部上下文使用非线程安全的对象仍然可以保证线程安全性。不过请小心：无论是对象被限制于执行线程的设计需求，还是对“被限制的对象并非线程安全”这一情况

的明确知晓，都只是存在于一线开发人员编码的那一刻。如果线程内部用法的设定没有清楚地文档化，那么后期维护人员会错误放任对象的逸出。

3.3.3 ThreadLocal

一种维护线程限制的更加规范的方式是使用 ThreadLocal，它允许你将每个线程与持有数值的对象关联在一起。ThreadLocal 提供了 get 与 set 访问器，为每个使用它的线程维护一份单独的拷贝。所以 get 总是返回由**当前执行线程**通过 set 设置的最新值。

线程本地 (Thread Local) 变量通常用于防止在基于可变的单体 (Singleton) 或全局变量的设计中，出现 (不正确的) 共享。比如说，一个单线程化的应用程序可能会维护一个全局的数据库连接，这个 Connection 在启动时就已经被初始化了。这样就可以避免为每个方法都传递一个 Connection。因为 JDBC 规范并未要求 Connection 自身一定是线程安全的，因此，如果没有额外的协调时，使用全局变量的多线程应用程序同样不是线程安全的。通过利用 ThreadLocal 存储 JDBC 连接，如同清单 3.10 的 ConnectionHolder，每个线程都会拥有属于自己的 Connection。

清单 3.10 使用 Thread Local 确保线程封闭性

```
private static ThreadLocal<Connection> connectionHolder
    = new ThreadLocal<Connection>() {
        public Connection initialValue() {
            return DriverManager.getConnection(DB_URL);
        }
    };

public static Connection getConnection() {
    return connectionHolder.get();
}
```

这项技术还用于下面的情况：一个频繁执行的操作既需要像 buffer 这样的临时对象，同时还需要避免每次都重分配 (realloc) 该临时对象。举例来说，在 Java 5.0 以前，Integer.toString() 方法使用 ThreadLocal 存储一个 12-byte 的缓冲区来格式化结果，而不是使用共享的静态缓冲区 (这需要用锁) 或者在每次调用前都分配一个新的缓冲¹²。

线程首次调用 ThreadLocal.get 方法时，会请求 initialValue 提供一个初始值。概念上，你可以将 ThreadLocal<T> 看作 map< Thread, T> 它存储了与线程相关的值，不

¹² 除非操作运行得非常频繁，或者分配操作异乎寻常地昂贵，否则这项技术不太可能有任何性能优势。因此在 Java 5.0 中，它被一种更直接的方式所取代，就是为每一次调用分配一个新的缓冲区，这说明，为临时缓存这种简单的事物而使用 ThreadLocal，并没有什么性能优势。

过事实上它并非这样实现的。与线程相关的值存储在线程对象自身中，线程终止后，这些值会被垃圾回收。

假设你正在将一个单线程的应用迁移到多线程环境中，你可以将共享的全局变量都转换为 `ThreadLocal` 类型，这样可以确保线程安全。前提是全局共享（shared globals）的语义允许这样。如果将应用级的缓存变成一堆线程本地缓冲，它将毫无价值。

实现一个应用程序框架会广泛地使用 `ThreadLocal`。举例来说，在 EJB 调用期间，J2EE 容器把一个事务上下文与一个可执行线程关联起来。下面是一种简单的实现，它利用静态 `ThreadLocal` 持有事务上下文：当框架代码需要获知当前正在运行的是哪个事务时，只要从 `ThreadLocal` 中获得事务的上下文即可。这样很方便，因为它降低了为每个方法传递执行上下文信息的需要，不过却增加了任何使用该机制的代码与框架间的耦合。

`ThreadLocal` 很容易被滥用：比如将它们所封闭的属性作为使用全局变量的许可证，或者是创建一种将方法的参数“隐藏”起来的方法。如同全局变量，线程本地变量会降低重用性，引入隐晦的类间的耦合。因此应该谨慎地使用它们。

3.4 不可变性

为了满足同步的需要，另一种方法是使用不可变对象[EJ Item13]。到目前为止，几乎所有我们已经描述过的原子性与可见性的危险，比如访问过期数据，未及时更新或者观察一个处于不一致状态的对象，它们都产生于多线程下各种难以预测的行为协同工作，多个线程总试图同时访问相同的可变状态。如果对象的状态不能被修改，这些风险与复杂度就自然而然地消失了。

创建后状态不能被修改的对象叫做不可变对象。不可变对象天生是线程安全的。它们的常量(域)是在构造函数中创建的。既然它们的状态无法被修改，这些常量永远不会变。

不可变对象永远是线程安全的。

不可变对象是**简单的**。它们只有一种状态，构造函数谨慎地控制着这个状态。程序设计中最为困难的元素之一是推断复杂对象的可能状态。另一方面，推断不可变对象的状态却是很轻松的。

不可变对象也是更**安全**的。将可变对象传递给不可信的代码，或者将它发布到不可信代码可以找到的地方，都是危险的——不可信代码可能会改变它们的状态，甚至更糟的，还会保留引用并在其他线程中修改它们的状态。另一方面，不可变对象不会被恶意的或者

漏洞百出的代码所破坏，所以它们是安全的，可以放心地共享和发布，不需要创建防御性拷贝 [EJ Item 24]。

无论是 Java 语言规范还是 Java 存储模型都没有关于不可变性的正式定义，但是不可变性并不简单地等于将对象中的所有域都声明为 `final` 类型，所有域都是 `final` 类型的对象仍然可以是可变的，因为 `final` 域可以获得一个到可变对象的引用。

只有满足如下状态，一个对象才是不可变的：

- 它的状态不能在创建后再被修改；
- 所有域都是 `final` 类型¹³；并且，
- 它被正确创建(创建期间没有发生 `this` 引用的逸出)。

在不可变对象的内部，同样可以使用可变性对象来管理它们的状态，如同清单 3.11 中示范的那样。尽管存储姓名的 `set` 是可变的，但是 `ThreeStooges` 的设计使得它在被创建后就不可能再修改 `set`。 `Stooges` 引用是 `final` 类型的，所以所有的对象状态只能通过 `final` 域询问。前面列出的最后一条要求，“正确创建”是容易满足的。因为构造函数不会做什么事情，从而引起 `this` 的调用者之外的和不同于构造函数的代码也可以访问 `this` 引用。

清单 3.11 构造于底层可变对象之上的不可变类

```
@Immutable
public final class ThreeStooges {
    private final Set<String> stooges = new HashSet<String>();

    public ThreeStooges() {
        stooges.add("Moe");
        stooges.add("Larry");
        stooges.add("Curly");
    }

    public boolean isStooge(String name) {
        return stooges.contains(name);
    }
}
```

因为程序的状态自始至终都在变化着，你可能会想使用不可变对象会有很多限制，但情况并非如此。“对象是不可变的”与“到对象的引用是不可变的”之间并不等同。

¹³ 从技术上讲，不可变对象的域并未全部声明为 `final` 类型，这样的情况是可能存在的，`String` 就是这种类。设计这种类依赖于对良性数据竞争的精准分析，还需要对 Java 存储模型有深入的理解。（满足一下你的好奇心：`String` 会惰性地（*Lazily*）计算哈希值：当第一次调用 `hashCode` 时，`String` 计算哈希值，并将它缓存在一个非 `final` 域中。之所以可以这样做，仅是因为这个域所表现的非默认的（*nondefault*）值，在每次计算后都得到相同的结果，因为该结果来自一个已经确定的不可变的状态。但请不要自己这样做！）

程序存储在不可变对象中的状态仍然可以通过替换一个带有新状态的不可变对象的实例得到更新。后面的章节提供了使用这项技术的例子¹³。

3.4.1 Final 域

`final` 关键字源于 C++ 的 `const` 机制，不过受到了更多的限制。它对不可变性对象的创建提供了支持。`final` 域是不能修改的（尽管如果 `final` 域指向的对象是可变的，这个对象仍然可被修改），然而它在 Java 存储模型中还有着特殊的语义。`final` 域使得确保初始化安全性（initialization safety）成为可能，初始化安全性让不可变性对象不需要同步就能自由地被访问和共享。

即使对象是可变的，将一些域声明为 `final` 类型仍然有助于简化对其状态的判断。因为限制了对对象的可见性，也就约束了其可能的状态集，即使有一两个可变的变量，这样一个“几乎不可变”的对象仍然比有很多的可变变量的对象要简单。将域声明为 `final` 类型，还向维护人员明确指出这些域是不能变的。

正如“将所有的域声明为私有的，除非它们需要更高的可见性”[EJ Item 12] 一样，“将所有的域声明为 `final` 型，除非它们是可变的”，也是一条良好的实践。

3.4.2 示例：使用 `volatile` 发布不可变对象

在第 24 页的 `UnsafeCachingFactorizer` 中，我们试图用两个 `AtomicReference` 存储最新的数字和它的因数。但是这并非是线程安全的，因为我们无法原子化地获取或者更新这两个相关的值。由于同样的原因，使用 `volatile` 变量也是不能保证线程安全性的。但是，有时不可变对象也可以提供一种弱形式的原子性。

用于因式分解的 `Servlet` 执行两个必须是原子性的操作：更新缓冲的结果，以及根据缓冲数值是否与请求数值相匹配，有条件的选取缓存中的结果。无论何时，对一组相关数值都应该执行原子性操作，并且可以考虑为它们创建不可变的容器（`holder`）类，比如清单 3.12 中的 `OneValueCache`¹⁴。

通过使用不可变对象来持有所有的变量，可以消除在访问和更新这些变量时的竞争条

¹³ 很多开发者担心这种方法会带来性能问题，这通常是杞人忧天。内存分配（allocation）没有你想象的那么昂贵，而且不可变对象提供了额外的性能优势：降低了对锁和防御性拷贝的需要，减少了后续垃圾回收产生的冲突。

¹⁴ 如果没有调用 `getter` 与构造函数中的 `copyOf`，`OneValueCache` 就不是不可变的。`Arrays.copyOf` 作为一个工具方法，将在 Java 6 中被加入。`Clone` 在这里同样胜任。

清单 3.12 在不可变的容器中缓存数字和它的因数

```
@Immutable
class OneValueCache {
    private final BigInteger lastNumber;
    private final BigInteger[] lastFactors;

    public OneValueCache(BigInteger i,
                        BigInteger[] factors) {
        lastNumber = i;
        lastFactors = Arrays.copyOf(factors, factors.length);
    }

    public BigInteger[] getFactors(BigInteger i) {
        if (lastNumber == null || !lastNumber.equals(i))
            return null;
        else
            return Arrays.copyOf(lastFactors, lastFactors.length);
    }
}
```

件。若使用可变的容器对象，你就必须使用锁以确保原子性；使用不可变对象，一旦一个线程获得了它的引用，永远不必担心其他线程会修改它的状态。如果更新变量，会创建新的容器对象，不过在此之前任何线程都还和原先的容器打交道，仍然看到它处于一致的状态。

清单 3.13 的 `VolatileCachedFactorizer` 利用 `OneValueCache` 存储缓存的数字及其因数。当一个线程设置 `volatile` 类型的 `cache` 域引用到一个新的 `OneValueCache` 后，新数据会立即对其他线程可见。

与 `cache` 域相关的操作不会相互干扰，因为 `OneValueCache` 是不可变的，而且每次只有一条相应的代码路径访问它。不可变的容器对象持有与不变约束相关的多个状态变量，并利用 `volatile` 引用确保及时的可见性，这两个前提保证了即使 `VolatileCachedFactor` 没有显式地用到锁，但仍然是线程安全的。

3.5 安全发布

目前为止我们都关注确保对象不会被发布。比如，让对象限制在线程中或者另一个对象的内部。当然，有时我们又的确希望跨线程共享对象，这时，我们必须安全地共享它。很不幸，像清单 3.14 那样简单地将对象的引用存储到公共域中，还不足以安全地发布它。

清单 3.13 使用到不可变容器对象的 volatile 类型引用，缓存最新的结果

```
@ThreadSafe
public class VolatileCachedFactorizer implements Servlet {
    private volatile OneValueCache cache =
        new OneValueCache(null, null);

    public void service(ServletRequest req, ServletResponse resp) {
        BigInteger i = extractFromRequest(req);
        BigInteger[] factors = cache.getFactors(i);
        if (factors == null) {
            factors = factor(i);
            cache = new OneValueCache(i, factors);
        }
        encodeIntoResponse(resp, factors);
    }
}
```

清单 3.14 在没有适当的同步的情况下就发布对象（不要这样做）

```
// 不安全发布
public Holder holder;

public void initialize() {
    holder = new Holder(42);
}
```



你可能感到惊讶，这个看上去无恙的例子竟然也会失败。由于可见性的问题，容器还是会在其他线程中被设置为一个不一致的状态，即使它的不变约束已经在构造函数中得以正确创建！这种不正确的发布导致其他线程可以观察到“**局部创建对象**（partially constructed object）”。

3.5.1 不正确发布：当好对象变坏时

你无法信赖局部创建对象，一个监视着处于不一致状态对象的线程，会看到尽管该对象自发布之后从未修改过，但是它的状态还是会发生突变。事实上，如果清单 3.15 的 Holder 使用清单 3.14 的不安全发布方式发布，那么除了发布线程，其他线程调用 `assertSanity` 时都可能抛出 `AssertionError`¹⁵。

¹⁵ 此种问题并非出在 Holder 类自身，而是 Holder 没有被正确发布，但是将域 `n` 声明为 `final` 类型，使 Holder 成为不可变的，可以避免出现不正确发布的问题。参见 3.5.2 节。

清单 3.15 如果 Holder 没有被正确发布，它将处于失败的风险中

```
public class Holder {  
    private int n;  
  
    public Holder(int n) { this.n = n; }  
  
    public void assertSanity() {  
        if (n != n)  
            throw new AssertionError("This statement is false.");  
    }  
}
```



因为没有同步来确保 Holder 对其他线程可见，所以我们称 Holder 是“非正确发布的”。没有正确发布的对象会导致两种错误。首先，发布线程以外的任何线程都可以看到 Holder 域的过期值，因而看到的是一个 null 引用或者旧值，即使此刻 Holder 已经被赋予新值。其次，更坏的情况是，线程看到的 Holder 引用是最新的，然而 Holder 状态却是过期的¹⁶。这使程序执行变得更加不可预测：线程首次读取某个域可能会看到过期值，再次读取该域会得到一个更新值，这正是 assertSanity 会抛出 AssertionError 原因。

我们处于自我复制的风险中，如果没有充足的同步，跨线程共享数据时会发生一些非常奇怪的事情。

3.5.2 不可变对象与初始化安全性

出于不可变对象的重要性，Java 存储模型为共享不可变对象提供了特殊的初始化安全性的保证。正如我们所见的，对象的引用对其他线程可见，并不意味着对象的状态一定对消费线程可见。为了保证对象状态有一个一致性视图，我们需要同步。

另一方面，**即使发布对象引用时没有使用同步**，不可变对象仍然可以被安全地访问。为了获得这种初始化安全性的保证，应该满足所有不可变性的条件：不可修改的状态，所有域都是 final 类型的以及正确的构造。（如果清单 3.15 中的 Holder 是不可变的，那么即使 Holder 没有正确地发布，assertSanity 也不会抛出 AssertionError。）

¹⁶ 可以看出，在构造函数中设置的域值，应该是向这些域写入的第一个值，因此，没有“更旧”的值可以作为所谓的过期值。但是 Object 的构造函数会先于子类的构造函数运行，并首先向所有域写入默认值。因此这些默认值可能成为域的过期值。

不可变对象可以在没有额外同步的情况下，安全地用于任意线程；甚至发布它们时亦不需要同步。

这个保证还会延伸到一个正确创建的对象中所有 `final` 类型域的值。没有额外的同步，`final` 域也可以被安全地访问。然而，如果 `final` 域指向可变对象，那么访问这些对象的状态时仍然需要同步。

3.5.3 安全发布的模式

如果一个对象不是不可变的，它就必须被安全地发布，通常发布线程与消费线程都必须同步化。此刻让我们关注一下，如何确保消费线程能够看到处于发布当时的对象状态；我们要解决对象发布后对其修改的可见性问题。

为了安全地发布对象，对象的引用以及对象的状态必须同时对其他线程可见。一个正确创建的对象可以通过下列条件安全地发布：

- 通过静态初始化器初始化对象的引用；
- 将它的引用存储到 `volatile` 域或 `AtomicReference`；
- 将它的引用存储到正确创建的对象 `final` 域中；
- 或者将它的引用存储到由锁正确保护的域中。

线程安全容器的内部同步，意味着将对象置入这些容器（比如 `Vector` 或者 `synchronizedList`）的操作遵守了前述的最后一条要求。如果线程 `A` 将对象 `X` 置入某个线程安全容器，随后线程 `B` 重新获得 `X`，这时可以保证 `B` 所看到 `X` 的状态，正是 `A` 设置的。尽管程序的代码并未控制 `X` 的行为，就是说这里没有显式的同步，但仍然可以保证上述描述的事情发生。线程安全库中的容器提供了如下的线程安全保证（即使在 `Javadoc` 上，也没有把这个主题表述清楚）：

- 置入 `Hashtable`、`synchronizedMap`、`ConcurrentMap` 中的主键以及键值，会安全地发布到可以从 `Map` 获得它们的任意线程中，无论是直接获得还是通过迭代器（`iterator`）获得；
- 置入 `Vector`、`CopyOnWriteArrayList`、`CopyOnWriteArraySet`、`synchronizedList` 或者 `synchronizedSet` 中的元素，会安全地发布到可以从容器中获得它的任意线程中；
- 置入 `BlockingQueue` 或者 `ConcurrentLinkedQueue` 的元素，会安全地发布到可以从队列中获得它的任意线程中。

类库中的其他交互 (Handoff) 机制 (比如 Future 和 Exchanger) 同样创建了安全发布; 我们会在介绍到它们时指出它们提供的安全发布。

通常, 以最简单和最安全的方式发布一个被静态创建的对象, 就是使用静态初始化器:

```
public static Holder holder = new Holder(42);
```

静态初始化器由 JVM 在类的初始阶段执行, 由于 JVM 内在的同步, 该机制确保了以这种方式初始化的对象可以被安全地发布 [JLS 12.4.2]。

3.5.4 高效不可变对象 (Effectively immutable objects)

有些对象在发布后就不会被修改, 其他线程想要在没有额外同步的情况下安全地访问它们, 此时安全地发布至关重要。所有的安全发布机制都能保证, 只要一个对象“发布当时 (as-published)”)”的状态对所有访问线程都可见, 那么到它的引用也都是可见的。如果“发布当时”的状态不会再改变, 那么确保任意访问是安全的, 就变得重要。

一个对象在技术上不是不可变的, 但是它的状态不会在发布后被修改, 这样的对象称作有效不可变对象。这种对象不必满足在 3.4 节里提出的不可变性的约束条件; 这些对象发布后程序只需简单地把它们当作不可变对象。用高效不可变对象可以简化开发, 并且由于减少了同步的使用, 还会提高性能。

任何线程都可以在没有额外的同步下安全地使用一个安全发布的高效不可变对象。

比如, Date 自身是可变的¹⁷, 但是如果你把它当作不可变对象来使用就可以忽略锁。否则, 每当 Date 被跨线程共享时, 都要用锁确保安全。假设你正在维护一个 Map, 它储存了每位用户的最近登录时间:

```
public Map<String, Date> lastLogin =  
    Collections.synchronizedMap(new HashMap<String, Date>());
```

如果 Date 值在置入 Map 中后就不会改变, 那么, synchronizedMap 中同步的实现, 对于安全地发布 Date 值, 是至关重要的。而访问这些 Date 值时就不再需要额外的同步。

¹⁷ 这也许是类库设计中的一个错误。

3.5.5 可变对象

如果对象在创建后被修改，那么安全发布仅仅可以保证“发布当时”状态的可见性。对于可变状态，同步不仅仅由于对象发布，而且还用于在每次对象被访问后，保证后续变化的可见性。为了保证安全地共享可变对象，可变对象必须被安全发布，同时必须是线程安全的或者是被锁保护的。

发布对象的必要条件依赖于对象的可变性：

- 不可变对象可以通过任意机制发布；
- 高效不可变对象必须要安全发布；
- 可变对象必须要安全发布，同时必须要线程安全或者是被锁保护。

3.5.6 安全地共享对象

当你获得一个对象的引用时，你都要知道可以用它来做什么。是否需要在使用它前先获得一个锁？是否可以修改它的状态，还是仅仅可读？很多同步错误都源自没有理解共享对象的这些“预设规则”。当你发布一个对象后，应该将如何访问它们写入文档。

在并发程序中，使用和共享对象的一些最有效的策略如下：

线程限制：一个线程限制的对象，通过限制在线程中，而被线程独占，且只能被占有它的线程修改。

共享只读 (shared read-only)：一个共享的只读对象，在没有额外同步的情况下，可以被多个线程并发地访问，但是任何线程都不能修改它。共享只读对象包括可变对象与高效不可变对象。

共享线程安全 (shared thread-safe)：一个线程安全的对象在内部进行同步，所以其他线程无须额外同步，就可以通过公共接口随意地访问它。

被守护的 (Guarded)：一个被守护的对象只能通过特定的锁来访问。被守护的对象包括那些被线程安全对象封装的对象，和已知被特定的锁保护起来的已发布对象。

组合对象

Composing Objects

到目前为止，我们已经介绍了线程安全与同步的基础知识。但是我们不希望为了获得线程安全而去分析每次内存访问；而希望线程安全的组件能够以安全的方式组合成更大的组件或程序。这一章介绍了一些构造类的模式，这些模式让类更容易成为线程安全的，并且不会让程序意外破坏这些类的线程安全性。

4.1 设计线程安全的类

尽管将所有的状态都存储在公共静态域中，仍然能写出线程安全的程序，但是比起那些经过适当封装的类来说，我们难以验证这种程序的线程安全性，也很难在修改它们的同时，保证不破坏它的线程安全性。在没有进行全局检查的情况下，封装能够保证类的线程安全性。

设计线程安全类的过程应该包括下面 3 个基本要素：

- 确定对象状态是由哪些变量构成的；
- 确定限制状态变量的不变约束；
- 制定一个管理并发访问对象状态的策略。

对象的状态首先要从它的域说起。如果对象的域都是基本类型（primitive）的，那么这些域就组成了对象的完整状态。清单 4.1 的 Counter 只有一个 value 域，因此 value 就构成 Counter 的完整状态。一个对象有 n 个基本域（primitive fields），它的状态就是域值组成的 n 元组（ n -tuple）；2D Point 的状态是它的坐标值 (x, y) 。如果一个对象的域引用了其他对象，那么它的状态也同时包含了被引用对象的域。举个例子，LinkedList 的状态包括了所有存储在链表中的节点对象的状态。

同步策略（synchronization policy）定义了对象如何协调对其状态的访问，并且不会违

清单 4.1 使用 Java 监视器模式的简单线程安全计数器

```
@ThreadSafe
public final class Counter {
    @GuardedBy("this") private long value = 0;

    public synchronized long getValue() {
        return value;
    }
    public synchronized long increment() {
        if (value == Long.MAX_VALUE)
            throw new IllegalStateException("counter overflow");
        return ++value;
    }
}
```

反它的不变约束或后验条件。它规定了如何把不可变性、线程限制和锁结合起来，从而维护线程的安全性，还指明了哪些锁保护哪些变量。为了保证开发者与维护者可以分析并维护类，应该将类的同步策略写入文档。

4.1.1 收集同步需求

维护类的线程安全性意味着要确保在并发访问的情况下，保护它的不变约束；这需要对其状态进行判断。对象与变量拥有一个**状态空间**（state space）：即它们可能处于的状态的范围。状态空间越小，越容易判断它们。尽量使用 final 类型的域，就可以简化我们对对象的可能状态进行分析。（不可变对象是一种极限情况，它只可能处于唯一的状态。）

很多类通过不可变约束来判定某一种状态是**合法的**还是**非法的**。Counter 中的 value 域是 long 类型。long 的状态空间跨越了从 Long.MIN_VALUE 到 Long.MAX_VALUE 的范围，但是 Counter 约束了 value 的取值：不允许是负值。

类似地，操作的后验条件会指出某种**状态转换**（state transitions）是非法的。如果 Counter 当前状态是 17，下一个**唯一**合法的状态是 18。如果下一个状态源自于当前状态，那么这个操作必须是复合操作。不是所有的操作都会受限于状态转换的约束，当更新一个表示温度的数值时，计算结果并不受先前状态的影响。

不变约束与后验条件施加在状态及状态转换上的约束，引入了额外的同步与封装的需要。如果某些状态可能是非法的，则必须封装该状态下的状态变量，否则客户代码会将对象置于非法状态。如果一个操作的过程中可能出现非法状态转换，则该操作必须是原子的。另一方面，如果类并未强制任何约束，我们就可以开放一些关于类的封装或序列化的条件，以此获得更佳的灵活性或更好的性能。

一个类的不变约束也可以约束多个状态变量。比如像清单 4.10 的 `NumberRange` 这种表示数值范围的类，它维护一个状态变量范围的最小和最大边界，这是很典型的情况。这些变量必须服从下面的约束：最小值边界应该小于或等于最大值边界。这种多变量的不变约束需要原子性：必须在单一的原子操作中获取或更新相互关联的变量。你不能先更新一个变量，然后释放锁，再重获锁，再更新其他的变量。因为当释放了锁后，可能会使对象处于无效状态。当不变约束涉及多个变量时，任何一个操作在访问相关变量期间，线程必须占有保护这些变量的锁。

不理解对象的不变约束和后验条件，你就不能保证线程安全性。要约束状态变量的有效值或者状态转换，就需要原子性与封装性。

4.1.2 状态依赖的操作

类的不变约束与方法的后验条件约束了对象合法的状态和合法状态转换。某些对象的方法也有基于状态的先验条件（`preconditions`）。例如，你无法从空队列中移除一个条目；在你删除元素前，队列必须处于“非空”状态。若一个操作存在基于状态的先验条件，则把它称为是状态依赖的（`state-dependent`）。

在单线程化的程序中，操作如果无法满足先验条件，必然失败，别无他选。但是在并发程序中，原本为假的先验条件可能会由于其他线程的活动而变成真。并发程序中有这种可能：持续等待，直到先验条件为真，再继续处理操作。

在 Java 中，等待特定条件成立的内置高效机制——`wait` 和 `notify`——与内部锁紧密地绑定在一起，因而想正确使用它们并不容易。创建一个操作，让它在执行前必须等待先验条件为真，不如使用现有类库来提供期望的状态依赖行为更容易，比如阻塞队列（`blocking queue`）或信号量（`semaphore`），以及其他同步工具（`Synchronizer`），这些将在第 5 章讲述；创建一个状态依赖的类，要使用平台与类库提供的底层机制，这些将在第 14 章讲述。

4.1.3 状态所有权

我们在 4.1 节曾暗示了对象的状态可以是在以该对象为根的对象图中所有域的一个子集。为什么可以是“子集”？在什么条件下，一个从给定对象可以到达的域不是该对象状态的一部分？

在定义对象状态是由哪些变量构成时，我们只考虑那些对象所拥有的数据。所有权（`Ownership`）并不是语言中明确具化的概念，而是类设计中的元素。如果你实例化并组

成了一个 `HashMap`，那么你就创建了多个对象：`HashMap` 对象、大量用于实现 `HashMap` 的 `Map.Entry` 对象，可能还包括其他的内部对象。`HashMap` 的逻辑状态包括所有 `Map.Entry` 和内部对象的状态，即使它们被实现为独立的对象。

不管是好是坏，垃圾回收省去了我们很多麻烦，我们不用再仔细考虑所有权的问题。在 C++ 中，向方法传递一个对象时，你必须认真地考虑是否传递对象的所有权，对象是短期借出所有权，还是长期赋予所有权。在 Java 中，这些所有权模型都是可能的，只是垃圾回收器降低了很多常见错误的开销，包括引用共享的错误，这使得对所有权的思考不必那么认真了。

在很多情况下，所有权与封装性总是在一起出现的：对象封装它拥有的状态，且拥有它封装的状态。拥有给定状态的所有者决定了锁协议，该协议用于维护变量状态的完整性。所有权意味着控制权，不过一旦你将引用发布到一个可变对象上，你就不再拥有独占的控制权，充其量只可能有“共享控制权”。类通常不会拥有由构造函数或方法传递进来的对象，除非该方法是被明确设计用来转换传递对象的所有权的（比如同步容器的包装工厂方法）。

容器类通常表现出一种“所有权分离”的形式。这是指容器拥有容器框架的状态，而客户代码拥有存储在容器中的对象的状态。以 `servlet` 框架中的 `ServletContext` 为例。`ServletContext` 为 `Servlet` 提供了类似于 `Map` 的对象容器服务。`Servlet` 可以通过名称，使用 `setAttribute` 和 `getAttribute` 在 `ServletContext` 中注册与重获应用程序对象。由于 `Servlet` 容器实现的 `ServletContext` 对象一定会被多个线程访问，因此 `ServletContext` 必须是线程安全的。所以在调用 `setAttribute` 和 `getAttribute` 时，`Servlet` 是不必同步的。但是使用存储在 `ServletContext` 中的对象时，可能必须要同步。这些对象属于应用程序，`Servlet` 容器只是存储它们并代替应用程序保管它们。正如所有的共享对象那样，它们必须安全地被共享。为了防止多线程并发访问同一对象时所带来的干扰，这些对象应该是线程安全对象、高效不可变对象或者由锁明确保护的对象¹。

4.2 实例限制

即使一个对象不是线程安全的，仍然有许多技术可以让它安全地用于多线程程序。比如，你可以确保它只被单一的线程访问（线程限制），也可以确保所有的访问都正确地被锁保护。

¹ 有趣的是，`HttpSession` 对象也实现了 `servlet` 框架中类似功能，但可能需要（may have）更严格的限制条件。因为 `servlet` 容器可以访问 `HttpSession` 里的对象，所以它们能够被序列化，从而实现复制和钝化，这些对象必须是线程安全的，因为容器将作为 Web Application 程序来访问它们。我们说可能需要（may have），是因为 `servlet` 规范并未要求复制与钝化，但是多数 `servlet` 容器将它作为常用的特性加以提供。

通过使用**实例限制**（instance confinement），封装简化了类的线程安全化工作，这通常称为“**限制**”[CPJ 2.3.3]。当一个对象被另一个对象封装时，所有访问被封装对象的代码路径就是全部可知的，这相比于让对象可被整个系统访问来说，更容易对代码路径进行分析。把限制与各种适当的锁策略相结合，可以确保程序以线程安全的方式使用其他非线程安全对象。

将数据封装在对象内部，把对数据的访问限制在对象的方法上，更易确保线程在访问数据时总能获得正确的锁。

被限制对象一定不能逸出到它的期望可用范围之外。可以把对象限制在类实例（比如私有的类成员）、语汇范围（lexical scope，比如本地变量）或线程（比如对象在线程内部从一个方法传递到另一个方法，不过前提是对象不被跨线程共享）中。对象不会自发地逸出自己，当然——这需要程序员的努力，将它们发布在其期望的可用范围内。

清单 4.2 的 PersonSet 示范了限制与锁如何协同确保一个类的线程安全性，即使它的组件状态变量并不是线程安全的。非线程安全的 HashSet 管理着 PersonSet 的状态。不过由于 mySet 是私有的，不会逸出，因此 HashSet 被限制在 PersonSet 中。唯一可以访问 mySet 的代码路径是 addPerson 与 containsPerson，执行它们都要获得 PersonSet 的锁。PersonSet 的内部锁保护了它所有的状态，因而确保了 PersonSet 是线程安全的。

清单 4.2 使用限制确保线程安全

```
@ThreadSafe
public class PersonSet {
    @GuardedBy("this")
    private final Set<Person> mySet = new HashSet<Person>();

    public synchronized void addPerson(Person p) {
        mySet.add(p);
    }

    public synchronized boolean containsPerson(Person p) {
        return mySet.contains(p);
    }
}
```

本例中未对 Person 的线程安全性作任何假设。但如果它是可变的，那么访问从 PersonSet 中获得的 Person 时，还需要额外的同步。为了安全地使用 Person 对象，最

可靠的方法是让 `Person` 自身是线程安全的；对 `Person` 对象加锁并不十分可靠，因为它还需要所有的用户都遵守协议：访问 `Person` 前先获得正确的锁。

实例限制是构建线程安全类的最简单的方法之一。它还使得对锁策略的选择更富有灵活性；`PersonSet` 刚好使用了它的内部锁来保护它的状态，其实无论任何锁，只要自始至终都使用它，同样可以很好地保护状态。实例限制还允许不同的锁保护不同的状态变量。

（请参见 236 页的 `ServerStatus`，它是一个由多个锁对象保护不同状态的类。）

平台类库中有很多线程限制的实例，包括一些类，它的存在就是为了把非线程安全的类转化为线程安全的。`ArrayList` 和 `HashMap` 这样的基本容器类是非线程安全的，但是类库提供了包装器工厂方法（`Collections.synchronizedList` 及其同族的方法），使这些非线程安全的类可以安全地用于多线程环境中。这些工厂方法利用 `Decorator` 模式（Gamma et al., 1995），使用一个同步的包装器对象包装容器；包装器将相关接口的每个方法实现为同步方法，并将请求转发到下层的容器对象上。只要包装器对象占有着对下层容器唯一的可触及的引用（底层容器限制于包装器内），包装器对象就是线程安全的。这些方法的 Javadoc 已经警告，所有对下层容器的访问必须经过包装器。

当然，发布一个受限对象，仍然可能破坏限制性。如果一个对象期望限制在特定的可用范围内，那么让它从这个范围内逸出的就是一个 bug。发布其他对象，比如迭代器或内部类实例，可能会间接地发布受限对象，这样受限对象同样会逸出。

限制性使构造线程安全的类变得更容易。因为类的状态被限制后，分析它的线程安全性时，就不必检查完整的程序。

4.2.1 Java 监视器模式

线程限制原则的直接推论之一是 **Java 监视器模式**（Java monitor pattern）²。遵循 Java 监视器模式的对象封装了所有的可变状态，并由对象自己的内部锁保护。

清单 4.1 的 `Counter` 演示了这个模式的典型案例：`Counter` 封装了一个状态变量 `value`，所有对该变量的访问都要通过 `Counter` 的方法，这些方法都是同步的。

² 虽然 Java 监视器模式与 Hoare 关于 **监视器**（monitor）的研究（Hoare, 1974）之间存在重要的不同，但是该模式的确从 Hoare 的工作中获得了灵感。进入和退出同步块的字节码指令甚至叫做 `monitorenter` 和 `monitorexit`，而且 Java 的内置（内部）锁有时也被叫做 **监视器锁**（monitor locks）或 **监视器**（monitors）。

很多像 `Vector` 和 `Hashtable` 这样的核心库类都使用了 Java 监视器模式。有时程序需要一个更加精巧的同步策略；第 11 章讲述了如何通过精巧的锁策略来提高可伸缩性。Java 监视器模式最大的优势在于简单。Java 监视器模式仅仅是一种习惯约定；任意锁对象，只要始终如一地使用，都可以用来保护对象的状态。清单 4.3 示范了一个使用私有锁保护状态的类。

清单 4.3 私有锁保护状态

```
public class PrivateLock {
    private final Object myLock = new Object();
    @GuardedBy("myLock") Widget widget;

    void someMethod() {
        synchronized(myLock) {
            // 访问或修改 Widget 的状态
        }
    }
}
```

使用私有锁对象，而不是对象的内部锁（或任何其他可公共访问的锁），有很多好处。私有的锁对象可以封装锁，这样客户代码无法得到它。然而可公共访问的锁允许客户代码涉足它的同步策略——正确地或不正确地。客户不正确地得到另一个对象的锁，会引起活跃度方面的问题。另外要验证程序是正确地使用着一个可公共访问的锁，需要检查完整的程序，而不是一个单独的类。

4.2.2 范例：机动车追踪器（tracking fleet vehicles）

清单 4.1 的 `Counter` 是一个简单的、没有实际意义的 Java 监视器模式示例。我们举一个更加实际些的例子：一个用于调度出租车、警车、货运卡车等机动车的“机动车追踪器（vehicle tracker）”。我们先使用监视器模式构建它，然后看看在维护线程安全性的前提下，如何释放一些封装性条件。

每一辆机动车都有一个 `String` 标识，并有一个与之对应的位置（`x, y`）。每个 `VehicleTracker` 对象都封装了一辆已知机动车的标识（`identity`）和位置（`location`），以使它更适合作为 MVC（model-view-controller）架构 GUI 应用中的数据模型（`data model`）。视图（`view`）线程和多个更新线程可能会共享数据模型。视图线程会获取机动车的名称和位置，将它们显示在显示器上：

```
Map<String, Point> locations = vehicles.getLocations();
for (String key : locations.keySet())
    renderVehicle(key, locations.get(key));
```

类似地，更新线程会通过从 GPS 设备上获取的数据或者调度员通过 GUI 界面手工输入的数据，修改机动车的位置。

```
void vehicleMoved(VehicleMovedEvent evt) {  
    Point loc = evt.getNewLocation();  
    vehicles.setLocation(evt.getVehicleId(), loc.x, loc.y);  
}
```

视图线程与更新线程会并发地访问数据模型，因此模型必须是线程安全的。清单 4.4 展示了一个使用 Java 监视器模式的“机动车追踪器”实现，清单 4.5 的 `MutablePoint` 描述了机动车的位置。

虽然 `MutablePoint` 不是线程安全的，但 `tracker` 类是的。程序没有将 `Map` 或是它包含的任何可变点发布出去。当我们需要将机动车的位置返回给调用者时，正确的返回值是从 `MutablePoint` 执行拷贝的构造函数或者 `deepCopy` 方法拷贝而来的，`deepCopy` 会创建一个新的 `Map`，它的值是从旧 `Map` 的 `key` 和 `value` 拷贝而来的³。

先复制可变的数据，再返回给用户，这种实现方式部分地维护着线程安全。如果 `vehicle` 的集合不是非常大的话⁴，这种做法通常不会造成性能问题。每次调用 `getLocation` 前先复制全部数据还会产生另一种副作用——即使真实的 `location` 已经改变，返回的容器的内容仍然不会改变。这是好是坏，取决于你的需求。如果 `location` 集合需要内部的一致性，比如当返回 `location` 集合快照的一致性至关重要时，这样做是有好处的，但是如果调用者需要每一辆机动车最新的信息，这样做又会是个问题，因而你需要更频繁地刷新 `location` 集合快照。

4.3 委托线程安全

几乎所有对象都是组合对象。当凭空构建一个类，或者使用非线程安全对象组装一个类时，Java 监视器模式十分有用。但是如果我们的类组件已经是线程安全的呢？我们需要添加一个额外的线程安全层吗？答案是：“依情况而定”。一些情况下，一个由线程安全的组件组合而成的组件是线程安全的（清单 4.7 和清单 4.9），另外的情况下，这仅仅是一个不错的开始（清单 4.10）。

在 23 页的 `CountingFactorizer` 中，我们向一个无状态的类中加入了一个 `AtomicLong` 类型的属性，所得组合对象仍然是线程安全的。因为 `CountingFactorizer`

³ 注意 `deepCopy` 不能仅仅用一个 `unmodifiableMap` 包装 `Map`，因为这样做只保护容器不被修改，并不能防止调用者修改存储在其中的可变对象。出于同样的原因，在 `deepCopy` 通过一个拷贝构造函数（`copy constructor`）生成 `HashMap` 同样是不正确的，因为这样做只复制了 `Point` 的引用，而不是 `Point` 对象本身。

⁴ 因为 `deepCopy` 是从 `synchronized` 类型的方法中调用，线程会在一个可能很耗时的拷贝操作期间一直占有 `tracker` 的内部锁。当追踪大量的机动车时，这会降低用户接口的响应性。

清单 4.4 基于监视器的机动车追踪器实现

```
@ThreadSafe
public class MonitorVehicleTracker {
    @GuardedBy("this")
    private final Map<String, MutablePoint> locations;

    public MonitorVehicleTracker(
        Map<String, MutablePoint> locations) {
        this.locations = deepCopy(locations);
    }

    public synchronized Map<String, MutablePoint> getLocations() {
        return deepCopy(locations);
    }

    public synchronized MutablePoint getLocation(String id) {
        MutablePoint loc = locations.get(id);
        return loc == null ? null : new MutablePoint(loc);
    }

    public synchronized void setLocation(String id, int x, int y) {
        MutablePoint loc = locations.get(id);
        if (loc == null)
            throw new IllegalArgumentException("No such ID: " + id);
        loc.x = x;
        loc.y = y;
    }

    private static Map<String, MutablePoint> deepCopy(
        Map<String, MutablePoint> m) {
        Map<String, MutablePoint> result =
            new HashMap<String, MutablePoint>();
        for (String id : m.keySet())
            result.put(id, new MutablePoint(m.get(id)));
        return Collections.unmodifiableMap(result);
    }
}

public class MutablePoint { /* 清单 4.5 */ }
```

清单 4.5 类似于 java.awt.Point 的可变 Point

```

@NotThreadSafe
public class MutablePoint {
    public int x, y;

    public MutablePoint() { x = 0; y = 0; }
    public MutablePoint(MutablePoint p) {
        this.x = p.x;
        this.y = p.y;
    }
}

```



的状态就是线程安全类 AtomicLong 的状态，而且 CountingFactorizer 并未对 counter 的状态施加额外的有效性约束，所以，很显然 CountingFactorizer 是线程安全的。我们可以说 CountingFactorizer 将它的线程安全性委托给了 AtomicLong：因为 AtomicLong 是线程安全的，所以 CountingFactorizer 也是⁵。

4.3.1 范例：使用委托的机动车追踪器

作为一个更真实的委托示例，让我们创建一个新版本的机动车追踪器，它将会委托线程安全的类。我们使用 Map 存储 location，所以就从一个线程安全的 Map 实现——ConcurrentHashMap——开始。我们用不可变的 Point 类取代 MutablePoint，来存储 location 信息，正如清单 4.6 所示。

清单 4.6 DelegatingVehicleTracker 使用不可变的 Point 类

```

@Immutable
public class Point {
    public final int x, y;

    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
}

```

Point 类是不可变的，因而是线程安全的。程序可以自由地共享与发布不可变值，所以我们返回 location 时不必再复制它们。

⁵ 如果 count 不是 final 类型的，那么分析 CountingFactorizer 的线程安全将会更复杂。如果 CountingFactorizer 将 count 修改为到另一个的 AtomicLong 引用，我们必须确保这一更新是对所有访问 count 的线程都是可见的，并且 count 引用的值不存在竞争条件。这是尽可能使用 final 域的另一个原因。

清单 4.7 的 `DelegatingVehicleTracker` 没有使用任何显式的同步；`ConcurrentHashMap` 管理了所有对状态的访问，而且 `Map` 的键与值都是不可变的。

清单 4.7 将线程安全委托到 `ConcurrentHashMap`

```
@ThreadSafe
public class DelegatingVehicleTracker {
    private final ConcurrentMap<String, Point> locations;
    private final Map<String, Point> unmodifiableMap;

    public DelegatingVehicleTracker(Map<String, Point> points) {
        locations = new ConcurrentHashMap<String, Point>(points);
        unmodifiableMap = Collections.unmodifiableMap(locations);
    }

    public Map<String, Point> getLocations() {
        return unmodifiableMap;
    }

    public Point getLocation(String id) {
        return locations.get(id);
    }

    public void setLocation(String id, int x, int y) {
        if (locations.replace(id, new Point(x, y)) == null)
            throw new IllegalArgumentException(
                "invalid vehicle name: " + id);
    }
}
```

如果我们使用原先的 `MutablePoint` 类代替 `Point`，就会让 `getLocations` 发布一个非线程安全的可变引用，从而会破坏封装性。请注意，我们已经略微改变了 `VehicleTracker` 类的行为；基于“监视器”的代码返回 `location` 的快照，基于“委托”的代码返回一个不可变的，但却是“现场（live）”的 `location` 视图。这意味着如果线程 *A* 调用 `getLocations` 时，线程 *B* 修改了一些 `Point` 的 `location`，这些变化会反映到返回给线程 *A* 的 `Map` 值中。正如我们在前面提到的，这可能是好事（获得最新的数据）也可能是坏事（车辆移动时位置的潜在不一致性），一切取决于你的需求。

如果需要一个不可变的瞬时（fleet）视图，`getLocations` 可以返回一个 `location Map` 的浅拷贝（shallow copy，只复制对象的引用，因此复制的对象与原始的对象是同一个对象；相反，深度拷贝（deepCopy）会复制对象的所有成员，因此复制的对象与原始的对象是不同的对象。）。因为 `Map` 的内容是不可变的，因此需要复制的只有 `Map` 的结构，而不包括它的内容，正如清单 4.8 所示（它返回一个普通的 `HashMap`，`getLocations` 并不承诺返回一个线程安全的 `Map`）。

清单 4.8 返回 location 集的静态拷贝，而非“现场（live）”的

```
public Map<String, Point> getLocations() {  
    return Collections.unmodifiableMap(  
        new HashMap<String, Point>(locations));  
}
```

4.3.2 非状态依赖变量

目前为止，在委托示例中仅仅委托了一个单一的线程安全的状态变量。我们也可以将线程安全委托到多个隐含的状态变量上，只要这些变量是彼此独立的，这意味着组合对象并未增加任何涉及多个状态变量的不变约束。

清单 4.9 的 `VisualComponent` 是一个允许客户注册鼠标键盘事件监听器的图形组件。它为每种类型的事件维护一个已注册监听器的清单，因而一个事件发生时，程序会调用相应的监听器。但是鼠标事件的监听器集与键盘事件的监听器集之间没有关系；它们彼此独立，因此 `VisualComponent` 可以将它的线程安全委托到这两个线程安全的清单上。

清单 4.9 委托线程安全到多个底层的状态变量

```
public class VisualComponent {  
    private final List<KeyListener> keyListeners  
        = new CopyOnWriteArrayList<KeyListener>();  
    private final List<MouseListener> mouseListeners  
        = new CopyOnWriteArrayList<MouseListener>();  
  
    public void addKeyListener(KeyListener listener) {  
        keyListeners.add(listener);  
    }  
  
    public void addMouseListener(MouseListener listener) {  
        mouseListeners.add(listener);  
    }  
  
    public void removeKeyListener(KeyListener listener) {  
        keyListeners.remove(listener);  
    }  
  
    public void removeMouseListener(MouseListener listener) {  
        mouseListeners.remove(listener);  
    }  
}
```

`VisualComponent` 使用 `CopyOnWriteArrayList` 存储每个监听器清单；这个线程安全

的 List 实现尤其适合管理监听器的清单（参见 5.2.3 节）。在 VisualComponent 中，不但每个 List 是线程安全的，而且不存在哪个不变约束会增加一个状态与另一个状态间的耦合，所以 VisualComponent 可以将它的线程安全责任委托到 mouseListeners 和 keyListeners 对象上。

4.3.3 当委托无法胜任时

大多数组合对象不像 VisualComponent 这样简单：它们的不变约束与组件的状态变量相联系。清单 4.10 的 NumberRange 使用两个 AtomicInteger 管理它的状态，而且受到一个额外的约束限制——第一个数小于或等于第二个。

清单 4.10 NumberRange 类没有完整地保护它的不变约束（不要这样做）

```
public class NumberRange {
    // 不变约束: lower <= upper
    private final AtomicInteger lower = new AtomicInteger(0);
    private final AtomicInteger upper = new AtomicInteger(0);

    public void setLower(int i) {
        // 警告 -- 不安全的“检查再运行”
        if (i > upper.get())
            throw new IllegalArgumentException(
                "can't set lower to " + i + " > upper");
        lower.set(i);
    }

    public void setUpper(int i) {
        // 警告 -- 不安全的“检查再运行”
        if (i < lower.get())
            throw new IllegalArgumentException(
                "can't set upper to " + i + " < lower");
        upper.set(i);
    }

    public boolean isInRange(int i) {
        return (i >= lower.get() && i <= upper.get());
    }
}
```



NumberRange 不是线程安全的；它没有保护好用于约束 lower 和 upper 的不变约束。setLower 和 setUpper **试图**保护不变约束，但又显得力不从心。setLower 和 setUpper 都是“检查再运行”的操作，但是它们没有适当地加锁以保证其原子性。假设值域是 (0,

10)，一个线程调用 `setLower(5)` 时，另一个线程正在调用 `setUpper(4)`，在一些偶发时段里，它们都能满足 `set` 方法中的检查，使修改全部生效。结果是现在的值域变成 (5, 4) —— 一个无效的状态。虽然底层的 `AtomicInteger` 是线程安全的，但是组合的类却不是。因为状态变量 `lower` 和 `upper` 不是彼此独立的，`NumberRange` 不能简单地将线程安全性委托给线程安全的状态变量上。

通过加锁维护不变约束，可以使 `NumberRange` 成为线程安全的，比如用一个公共锁保护 `lower` 和 `upper`。`NumberRange` 也要避免发布 `lower` 和 `upper`，以防止用户潜在地破坏不变约束。

如果类中还存在复合操作，就像 `NumberRange`，单独的委托仍然不是线程安全的正确方法。在这种情况下，类必须提供它自有的锁以保证复合操作是原子的。除非整个复合操作也可以委托给一个状态变量。

如果一个类由多个**彼此独立的**线程安全的状态变量组成，并且类的操作不包含任何无效状态转换时，可以将线程安全委托给这些状态变量。

即使 `NumberRange` 拥有线程安全状态的组件，也不能保证 `NumberRange` 的线程安全性，这个问题非常类似于 3.1.4 节中描述的关于 `volatile` 变量的一条规则：当且仅当一个变量没有参与那些涉及其他状态变量的不变约束时，才适合声明为 `volatile` 类型。

4.3.4 发布底层的状态变量

当你将线程安全性委托给一个对象底层的状态变量时，在什么条件下你可以发布这些变量，允许其他类也修改它们？答案再一次取决于不变约束对这些变量施加了何种影响。尽管 `Counter` 的 `value` 域自身允许等于任何整数值，但是 `Counter` 限制它只能等于正数，同时自增操作还限制了任何给定当前状态的下一有效状态的集合。如果你将 `value` 声明为 `public`，客户可以将它改为一个无效值，所以发布它可能导致类出现错误。另一方面，如果变量表示的是当前温度或者上一个登录用户的 ID，那么即使其类修改了这个值大概也不会破坏任何不变约束，进而发布这个变量是允许的。（这可能仍然不是一个好主意，因为发布一个可变变量会限制未来一些子类的开发，但这并不是造成类不安全的**必要因素**。）

如果一个状态变量是线程安全的，没有任何不变约束限制它的值，并且没有任何状态转换限制它的操作，那么它可以被安全发布。

举个例子，VisualComponent 发布 mouseListeners 或 keyListeners 是安全的。因为 VisualComponent 没有对监听器清单的合法状态施加任何限制，所以把这些域声明为 public，或者发布它们，都不会危及到线程的安全性。

4.3.5 示例：发布了状态的机动车追踪器

让我们构造另一个版本的机动车追踪器，它发布了底层的可变状态。我们又一次要略微改变一下接口，以适应这一变化。这次使用可变但线程安全的 Point。

清单 4.11 可变的线程安全 Point 类

```
@ThreadSafe
public class SafePoint {
    @GuardedBy("this") private int x, y;

    private SafePoint(int[] a) { this(a[0], a[1]); }

    public SafePoint(SafePoint p) { this(p.get()); }

    public SafePoint(int x, int y) {
        this.x = x;
        this.y = y;
    }

    public synchronized int[] get() {
        return new int[] { x, y };
    }

    public synchronized void set(int x, int y) {
        this.x = x;
        this.y = y;
    }
}
```

清单 4.11 的 SafePoint 提供的 getter 方法可以一次获得 x 与 y 的值，并返回一个二元数组⁶。如果我们为 x 和 y 分别提供 getter 方法，那么 x 与 y 的值可以在获得两个不同坐标

⁶ 私有构造函数的存在可以避免一些竞争条件，这些竞争条件会发生在复制构造函数实现为 `this(p.x, p.y)` 的时候；这是**私有构造函数捕获模式**（private constructor capture idiom）的实例（*Bloch and Gafter, 2005*）。

的时间之间发生改变，结果是调用者会看到一个不一致的值：位置 (x, y) 上从来没有任何机动车。我们可以使用 `SafePoint` 构建一个机动车追踪器，它发布了底层的可变状态却不破坏线程安全性，如清单 4.12 的 `PublishingVehicleTracker` 类所示。

清单 4.12 安全发布底层状态的机动车追踪器

```
@ThreadSafe
public class PublishingVehicleTracker {
    private final Map<String, SafePoint> locations;
    private final Map<String, SafePoint> unmodifiableMap;

    public PublishingVehicleTracker(
        Map<String, SafePoint> locations) {
        this.locations
            = new ConcurrentHashMap<String, SafePoint>(locations);
        this.unmodifiableMap
            = Collections.unmodifiableMap(this.locations);
    }

    public Map<String, SafePoint> getLocations() {
        return unmodifiableMap;
    }

    public SafePoint getLocation(String id) {
        return locations.get(id);
    }

    public void setLocation(String id, int x, int y) {
        if (!locations.containsKey(id))
            throw new IllegalArgumentException(
                "invalid vehicle name: " + id);
        locations.get(id).set(x, y);
    }
}
```

`PublishingVehicleTracker` 的线程安全性源自于它所委托的底层 `ConcurrentHashMap`，不过这次 `Map` 的内容是线程安全的可变 `Point`，而非不可变的。`getLocation` 方法返回底层 `Map` 的不可变拷贝，调用者在其上无法添加或移除车辆，却可以通过修改返回的 `Map` 中 `SafePoint` 的值，改变一个机动车的位置。`Map` 这一“现场”特性是否有价值，还是一个缺陷，仍然取决于需求。只有 `PublishingVehicleTracker` 对机动车追踪器的合法值没有施加任何额外的约束时，它才是线程安全的。如果需要对机动车的

location 的改变进行判断或者执行一些其他的操作，那么 PublishingVehicleTracker 的做法可能就不正确了。

4.4 向已有的线程安全类添加功能

Java 类库中包含了很多有用的“构建块”类。重用这些已有的类要好于创建一个新的：重用能够降低开发的难度、风险（因为已有的组件都已经过测试）和维护成本。有时一个线程安全类支持我们需要的全部操作，但更多时候，一个类只支持我们需要的大部分操作，这时我们需要在不破坏其线程安全性的前提下，向它添加一个新的操作。

作为例子，不妨假设我们需要一个线程安全的 List，它需要提供给我们一个原子的“缺少即加入（put-if-absent）”操作。一个同步的 List 实现几乎可以胜任，因为我们可以利用它提供的 contains 方法和 add 方法，构建一个“缺少即加入”操作。

“缺少即加入”的概念相当直观——在向容器中添加一个元素前先查看它是否已经存在，如果存在，就不添加。（“检查再运行（check-then-act）”的警钟现在该响起了）。对类的线程安全需求会隐式地加入另一个条件——像“缺少即加入”这样的操作必须是原子的。任何有合理的要求都会告诉你，如果你有一个不包含对象 X 的 List，并且执行两次“缺少即加入”的 add X，那么结果容器只应该包含一个 X 的拷贝。但是，倘若“缺少即加入”操作不是原子的，在一些偶发的时序中，两个线程都看到 X 不在容器中，并且都执行了 add X，导致容器中包含两份 X 的拷贝。

添加一个新原子操作的最安全的方式是，修改原始的类，以支持期望的操作。但是你可能无法访问源代码或者没有修改的自由，所以这通常是不可能的。即使你可以修改原始的类，也需要理解其实现的同步策略，才能在维持原有设计的前提下完善它的功能。直接向类中加入新方法，意味着所有实现类同步策略的代码仍然包含在一个源代码文件中，因此便于理解与维护。

另一种方法是扩展这个类，假如这个类在设计上是可以扩展的。清单 4.13 的 BetterVector 扩展了 Vector，并添加了一个 putIfAbsent 方法。虽然扩展 Vector 的做法相当直观，但是并非所有的类都给子类暴露了足够多的状态，以支持这种方案。

因为扩展后，同步策略的实现会被分布到多个独立维护的源代码文件中，所以扩展一个类比直接在类中加入代码更脆弱。如果低层的类选择了不同的锁保护它的状态变量，从而会改变它的同步策略，子类就在不知不觉中被破坏，因为它不能再用正确的锁控制对基类状态的并发访问。（Vector 的同步策略已由其规约固定住，所以 BetterVector 不会遇到这个问题。）

清单 4.13 扩展的 Vector 包含一个“缺少即加入”方法

```
@ThreadSafe
public class BetterVector<E> extends Vector<E> {
    public synchronized boolean putIfAbsent(E x) {
        boolean absent = !contains(x);
        if (absent)
            add(x);
        return absent;
    }
}
```

4.4.1 客户端加锁

对于一个由 `Collections.synchronizedList` 封装的 `ArrayList`，两种方法——向原始类中加入方法或者扩展类——都不正确，因为客户代码甚至不知道同步封装工厂方法返回的 `List` 对象的类型。第三个策略是扩展功能，而不是扩展类本身，并将扩展代码置入一个“助手（helper）”类。

清单 4.14 演示了一个错误的尝试，它创建一个助手类，该助手类包含一个作用于线程安全 `List` 的原子“缺少即加入”的操作。

清单 4.14 非线程安全的“缺少即加入”实现（不要这样做）

```
@NotThreadSafe
public class ListHelper<E> {
    public List<E> list =
        Collections.synchronizedList(new ArrayList<E>());
    ...
    public synchronized boolean putIfAbsent(E x) {
        boolean absent = !list.contains(x);
        if (absent)
            list.add(x);
        return absent;
    }
}
```



为什么这是不正确的？毕竟，`putIfAbsent` 声明为 `synchronized` 了，不是吗？这里的问题在于同步行为发生在**错误的锁**上。无论 `List` 使用哪个锁保护它的状态，可以确定的是这个锁并没用到 `ListHelper` 上。`ListHelper` 仅仅描绘了**同步的幻象**（*illusion of synchronization*）；即使一个 `list` 的操作全部声明为 `synchronized`，但是使用了不同锁，将意味着 `putIfAbsent` 对于 `List` 的其他操作而言，**并不是原子化的**。所以当 `putIfAbsent` 执行时，不能保证另一个线程不会修改 `list`。

为了让这个方法正确工作，我们必须保证方法所使用的锁，与 `List` 用于**客户端加锁**与**外部加锁**时所用的锁是**同一个锁**。客户端加锁必须保证客户端代码与对象 `X` 保护自己状态时使用的是相同的锁。为了正确执行客户端加锁，你必须知道 `X` 使用了哪个锁。

虽然 `Vector` 和同步的封装器类的文档不是很明确，但是通过使用 `Vector` 或者封装器容器（`wrapper collection`，而不是“被封装的容器（`wrapped collection`）”）的内部锁，它们声明可以支持客户端加锁。清单 4.15 演示了线程安全的 `List` 上的 `putIfAbsent` 操作，它使用了正确的客户端加锁。

清单 4.15 使用客户端加锁实现的“缺少即加入”

```
@ThreadSafe
public class ListHelper<E> {
    public List<E> list =
        Collections.synchronizedList(new ArrayList<E>());
    ...
    public boolean putIfAbsent(E x) {
        synchronized (list) {
            boolean absent = !list.contains(x);
            if (absent)
                list.add(x);
            return absent;
        }
    }
}
```

如果说为了添加另一个原子操作而去扩展一个类容易出问题，是因为它将加锁的代码分布到对象继承体系中的多个类里。然而客户端加锁其实是更加脆弱的，因为它必须将类 `C` 中的加锁代码（`locking code`）置入与 `C` 完全无关的类中。在那些不关注锁策略的类中使用客户端加锁时，一定要小心。

客户端加锁与扩展类有很多共同之处——所得类的行为与基类的实现之间都存在耦合。正如同扩展会破坏实现的封装性一样[EJ Item 14]，客户端加锁会破坏同步策略的封装性。

4.4.2 组合（`composition`）

向已有的类中添加一个原子操作，还有更健壮的选择：**组合**。清单 4.16 的 `ImprovedList` 通过将操作委托给底层的 `List` 实例，实现了 `List` 的操作，同时还添加了一个原子的 `putIfAbsent` 方法。（就像 `Collections.synchronizedList` 和其他容器封装器那样，`ImprovedList` 假设一旦有一个 `list` 传给它的构造函数后，客户将不再直接使用这个 `list`，而仅仅通过 `ImprovedList` 访问它。）

清单 4.16 使用组合（composition）实现“缺少即加入”

```
@ThreadSafe
public class ImprovedList<T> implements List<T> {
    private final List<T> list;

    public ImprovedList(List<T> list) { this.list = list; }

    public synchronized boolean putIfAbsent(T x) {
        boolean contains = list.contains(x);
        if (contains)
            list.add(x);
        return !contains;
    }

    public synchronized void clear() { list.clear(); }
    // ... similarly delegate other List methods
}
```

通过使用内部锁，ImprovedList 引入了一个新的锁层。它并不关心底层的 List 是否线程安全，即使 List 不是线程安全的，或者会改变 ImprovedList 的锁实现，ImprovedList 都有自己兼容的锁可以提供线程安全性。虽然额外的一层同步可能会带来一些微弱的性能损失⁷，但是相比于去尝试模拟另一个对象的锁策略而言，ImprovedList 并不那么脆弱。我们已经使用 Java 监视器模式有效地封装了一个已有的 List，而且只要我们的类持有底层 List 的唯一外部引用，那么就能保证提供线程安全性。

4.5 同步策略的文档化

在维护线程安全性的过程中，文档是最强大的（同时令人遗憾的是，也是最未被充分利用的）工具之一。用户为了确定一个类是否是线程安全的，会查阅文档；维护者为了理解实现策略以避免在维护中无意威胁到安全性，会查阅文档。不幸的是，所有这些文档的利益既得者们从文档中得到的信息通常比他们期望的要少得多。

为类的用户编写类线程安全性担保的文档；为类的维护者编写类的同步策略文档。

⁷ 因为可以保证底层 List 的同步是无竞争的（uncontended），速度很快，所以性能的损失是很小的；参见第 11 章。

每次使用 `synchronized`, `volatile` 或者任何线程安全类, 都表现了一种同步策略, 这项策略是为了确保处于并发访问中的数据完整性。这个策略是你程序设计的一个元素, 因此应该将它文档化。当然, 设计的时候是编写设计决策文档的最佳时间。几周或几个月之后, 一些设计细节会变得模糊——所以在你忘记前将它写下来。

起草一个同步策略需要作出一系列的决策: 将哪些变量声明为 `volatile` 类型, 哪些变量被锁保护, 哪个(哪些)锁保护哪些变量, 哪些变量是不可变的或者被限制在线程中的, 哪些操作必须是原子的, 等等。其中有些是非常严格的实现细节, 出于日后维护的考虑, 应该为它们编写文档。还有一些会影响类的公共可视的锁行为, 应该将它们作为类规约的一部分而写入文档。

最起码的, 要为类的线程安全性担保编写文档。它是线程安全的么? 它的回调是否持有一个锁? 有没有特定的锁会影响它的行为? 不要强迫用户冒着风险去猜测这些, 如果你并没有打算为客户端加锁提供支持的话, 好吧, 到此为止不必多费口舌了。如果你希望用户能够在你的类中创建新的原子操作, 就像我们在 4.4 节所作的, 你需要编写文档, 告诉用户应该获得哪个锁才能保证安全地添加新操作。如果你利用锁保护状态, 也要编写文档以便日后的维护, 其实这非常简单——窍门就是使用 `@GuardedBy` 标签 (Annotation)。如果你使用了更为精巧的方法去维护线程安全性, 一定要将它们写入文档, 否则维护者难以清楚地理解它们。

线程安全性方面的文档, 甚至平台核心库类的文档, 它们的现状都很难令人满意。你曾经有几次可以通过查看一个类的 Javadoc, 就能判断它是否是线程安全的⁸? 大多数类并未对此提供任何线索。更令人遗憾的是, 很多像 `servlet` 和 `JDBC` 这样的官方 Java 技术规范, 也没有在文档中提及线程安全性的承诺和条件。

尽管谨慎的态度要求我们不要随意猜测规约之外的行为, 但是我们还是完成了工作, 并且我们还要继续在很多糟糕的假设间进行选择。仅仅因为一个对象看上去应该是线程安全的, 我们就可以假设它是安全的么? 我们先获得了一个对象的锁, 就可以假设对它的访问是线程安全的么? (这个风险性高的技术只有在我们控制了**所有**访问这个对象的代码时, 才能正确工作; 否则, 它只描绘了一个线程安全性的幻象。) 无论哪种选择都很难令人满意。

更糟糕的是, 我们的感觉通常是错误的: 我们认为“可能是线程安全”的类, 其实并不是。举个例子, `java.text.SimpleDateFormat` 并不是线程安全的, 然而直到 JDK 1.4 以前的 Javadoc, 都忽略了提及这一点。这个特殊的类不是线程安全的, 引起了很多开发者的惊讶。有多少程序已经错误地创建了这个非线程安全对象的共享实例, 并在多线程中使用它? 这些程序在大负载下引发的错误结果是否被察觉到了?

如果一个类没有明确指明, 就不要假设它是线程安全的, 这样可以避免

⁸ 如果你从未考虑过这些, 我们倒是很钦佩你的乐观。

SimpleDateFormat 的问题。另一方面，倘若不对容器提供的对象（比如 HttpSession）的线程安全性作一些合理的假设，又不可能开发出一个基于 Servlet 的应用。不要让你的客户或者同事必须也做这样的猜测。

4.5.1 含糊不清的文档（Interpreting vague documentation）

很多 Java 技术规范对于接口的线程安全性的担保和条件都只字未提，或者只有只言片语，比如 ServletContext、HttpSession 或 DataSource⁹。这些接口是由你的容器或数据库供应商实现的，而你通常无法查看它的源代码去了解它的实现细节。另外，你也不希望依赖于一个特定 JDBC driver 的实现细节——你希望遵从标准，让你的代码可以工作于任何一个 JDBC driver。但是“线程”和“并发”这些词汇从未出现在 JDBC 的规范中，并且令人遗憾的是，在 Servlet 规范中也只有只言片语。那么你能做什么呢？

你只能去猜测。有一种方式可以提高你猜测的准确性：从规范**实现者**的角度去解读规范（比如容器或数据库的供应商），而不是从规范使用者的角度去看。调用 Servlet 的通常是容器管理的（container-managed）线程，因而我们可以安全地去假设容器知道是否有多个容器管理的线程在运行。Servlet 容器确保为多个 Servlet 提供服务的对象是可用的，比如 HttpSession 或 ServletContext。所以 Servlet 容器应该预见到这些对象会被并发地访问，因为容器创建了多个会调用像 Servlet.service 这样的方法的线程，这些线程都会访问 ServletContext。

无法想象如果这些对象的上下文是单线程化的，它还会有什么用途。所以我们必须要做的一条假设是，它们已被实现为线程安全的了，即使规范并没有明确地要求这一点。另外，如果它们需要客户端锁，那么客户端代码应该同步哪个锁？文档没有说明这一点，而且企图猜测它也是很荒唐的。规范中的示例和官方演示如何访问 ServletContext 或 HttpSession 的向导进一步验证了这个“合理的假设”，同时也说明不要使用任何客户端的同步。

另一方面，通过 setAttribute 置入 ServletContext 或 HttpSession 的对象由 Web Application 所有，而不是 Servlet 容器所有。Servlet 规范没有建议任何用来协调对这些共享属性并发访问的机制。所以容器代替 Web Application 所存储的这些对象应该是线程安全的，或者是高效不可变的。如果容器所做的全部只是代替 Web Application 存储这些属性，那么当 Servlet Application 代码访问它们时，应该确保它们始终被一个锁保护。但是容器出于复制或钝化的目的，可能会序列化 HttpSession 中的对象，而且容器不可能知道你的锁协议，所以你应该自己保证这些对象是线程安全的。

⁹ 我们感到格外令人沮丧的是，连续多个主要版本的规范始终都未提及到这一点。

一个表现为可重用数据库连接的池，就可以作为一个类似 **JDBC DataSource** 的接口。**DataSource** 为一个应用程序提供服务。在单线程化的应用程序的上下文中，它并不是非常重要。然而很难想象在多线程的情况下不会调用 `getConnection`。同样地，在 **Servlet** 中，**JDBC** 规范关于 **DataSource** 的许多示例代码中，并没有指出任何 **DataSource** 都是线程安全的。所以，尽管 **JDBC** 规范没有承诺 **DataSource** 是线程安全的，也没有强迫供应商提供一个线程安全的实现，不过同样是因为“如果不这样将是荒唐的”，我们别无选择，只能认定 `DataSource.getConnection` 不需要额外的客户端加锁。

另一方面，我们不能对 **DataSource** 分配的 **JDBC Connection** 对象也作同样的讨论，因为直到它们返回到连接池前，程序不必期望其他的活动共享它们。所以如果一个获得了 **JDBC Connection** 的活动跨越了多个线程，那么它必须负责确保利用同步正确地保护到 **Connection** 的访问。（大多数应用程序中，实现一个使用 **JDBC Connection** 的活动时，总是将 **Connection** 限制在某个特定的线程中。）

构建块

Building Blocks

上一章探究了一些关于构造线程安全类的技术，包括将线程安全性委托给现有的线程安全类。在实践中，委托是创建线程安全类最有效的策略之一：只需要用已有的线程安全类来管理所有状态即可。

平台类库包含了一个并发构建块¹的丰富集合，比如线程安全容器和多种**同步工具**（Synchronizer）。Synchronizer 用来调节相互协作的线程间的控制流。这一章将涵盖其中最有用的并发构建块，特别是 Java 5.0 和 Java 6 引入的一些新特性，以及使用它们构造并发应用程序时用到的一些模式。

5.1 同步容器

同步容器类包括两部分，一个是 Vector 和 Hashtable，它们是早期 JDK 的一部分；另一个是它们的同系容器，在 JDK1.2 中才被加入的同步包装（wrapper）类。这些类是由 Collections.synchronizedXxx 工厂方法创建的。这些类通过封装它们的状态，并对每一个公共方法进行同步而实现了线程安全，这样一次只有一个线程能访问容器的状态。

5.1.1 同步容器中出现的問題

同步容器都是线程安全的。但是对于复合操作，有时你可能需要使用额外的客户端加锁（client-side locking）进行保护。通常对容器的复合操作包括：迭代（反复获取元素，直到获得容器中的最后一个元素）、导航（navigation，根据一定的顺序寻找下一个元素）以及条件运算，比如“缺少即加入”（put-if-absent），检查 Map 中是否存在关键字 K，如果没有，就加入 mapping（K,V）。在一个同步的容器中，这些复合操作即使没有客户端加锁的保护，技术上也是线程安全的，但是当其他线程能够并发修改容器的时候，它们就可能不会按照你期望的方式工作了。

¹ 译注：原文为 Building Blocks，意为儿童游戏用的积木玩具，暗指构成大型并发程序的基本要素，本书中统一译为“构建块”。

清单 5.1 表现了两个操作 `Vector` 的方法, `getLast` 和 `deleteLast`。这两个方法都是按“检查再运行”(check-then-act)的顺序操作。两个方法首先都调用 `size` 来确定数组的长度, 然后分别用这个长度值获取和删除 `Vector` 的最后一个元素。

清单 5.1 操作 `Vector` 的复合操作可能导致混乱的结果

```
public static Object getLast(Vector list) {  
    int lastIndex = list.size() - 1;  
    return list.get(lastIndex);  
}  
  
public static void deleteLast(Vector list) {  
    int lastIndex = list.size() - 1;  
    list.remove(lastIndex);  
}
```



这些方法看起来是没有危害的。在某种程度上来说是这样的——无论多少个线程同时调用, 也不破坏 `Vector`。但是对于这些方法的调用者来说, 情况就不同了。如果线程 A 调用 `getLast`, 从一个包含 10 个元素的 `Vector` 中获得最后一个元素, 同时线程 B 调用 `deleteLast` 删除该 `Vector` 的最后一个元素。这些操作按照图 5.1 所示的顺序交替执行, 最后 `getLast` 抛出了一个 `ArrayIndexOutOfBoundsException` 异常。`getLast` 中调用 `size` 与随后调用 `get` 之间, `Vector` 缩短了, 并且第一步中计算出来的索引值已经失效了。这很好地保持了 `Vector` 规约的一致性——如果请求了一个不存在的元素, 它会抛出一个异常。但是这不是调用者调用 `getLast` 所期望的结果, 即使面对并发的修改也是如此, 除非 `Vector` 在一开始就是空的。

好在同步容器遵守一个支持客户端加锁²的同步策略, 因此只要我们知道应该使用哪一个锁, 就有可能针对其他的容器操作创建新的原子操作。同步容器类通过对它的对象自身进行加锁, 保护它的每一个方法。通过获得容器的锁, 我们可以使 `getLast` 和 `deleteLast` 变为原子操作, 确保 `Vector` 的大小在调用 `size` 和 `get` 的之间不会发生变化, 就像我们在清单 5.2 中看到的。在调用 `size` 和调用相应的 `get` 之间, `Vector` 的长度可能发生变化, 这个的风险依然会在我们迭代访问 `Vector` 元素的过程中出现, 就像清单 5.3 显示的那样。

这种迭代方式依赖于对其他线程的信任, 相信没有线程会在调用 `size` 和 `get` 之间修改 `Vector`。在一个单线程化的环境中, 这种假设是绝对没有问题的, 但是当其他线程可能并发地修改 `Vector` 时, 麻烦就出现了。就像 `getLast` 那样, 如果你在对 `Vector` 进行迭代的时候, 另一个线程正在删除掉一个元素, 并且这两个操作不巧地交替发生了, 那么这种迭代最终会抛出 `ArrayIndexOutOfBoundsException`。

² 这作为正确使用迭代的模式案例, 在 Java 5.0 的文档中只间接提及。

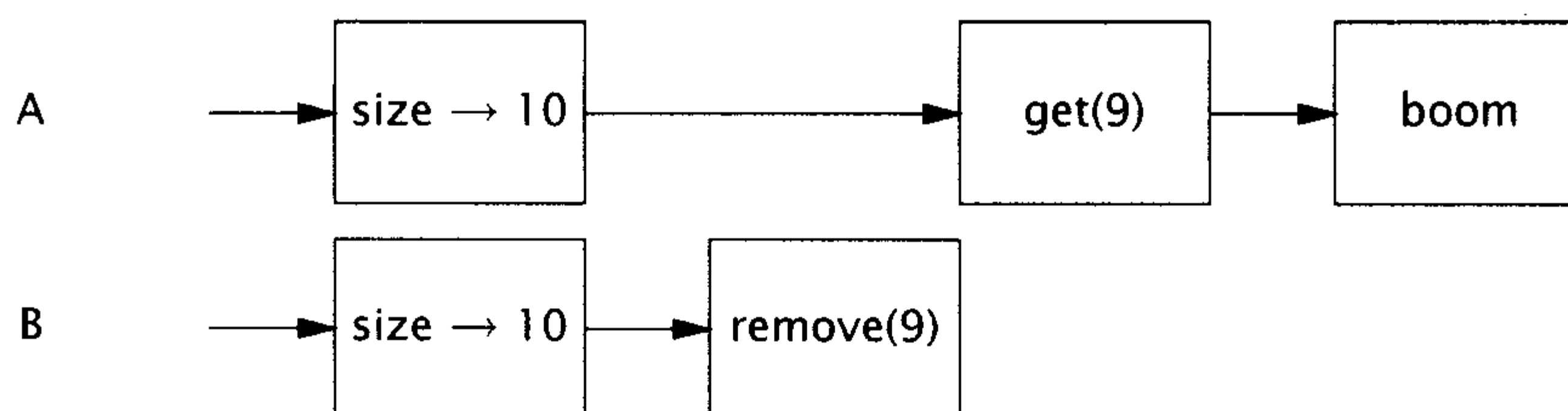


图 5.1 getLast 和 deleteLast 交替发生，抛出 ArrayIndexOutOfBoundsException

清单 5.2 使用客户端加锁，对 Vector 进行复合操作

```

public static Object getLast(Vector list) {
    synchronized (list) {
        int lastIndex = list.size() - 1;
        return list.get(lastIndex);
    }
}

public static void deleteLast(Vector list) {
    synchronized (list) {
        int lastIndex = list.size() - 1;
        list.remove(lastIndex);
    }
}

```

清单 5.3 迭代中可能抛出的 ArrayIndexOutOfBoundsException

```

for (int i = 0; i < vector.size(); i++)
    doSomething(vector.get(i));

```

尽管清单 5.3 中的迭代可能抛出异常，但并不意味着 Vector 就不是线程安全的。Vector 的状态仍然是有效的，事实上异常恰好使它保持了规约一致性。然而，在正常读取最后一个元素或者进行迭代时抛出异常，的确不是人们所期望的。

造成迭代不可靠的问题同样可以通过在客户端加锁来解决，这要增加一些针对可伸缩性的开销。像在清单 5.4 显示的那样，通过在迭代期间持有 Vector 的锁，我们防止其他线程在迭代期间修改 Vector。不幸的是，我们同样完全阻止了其他线程在这期间访问它，这削弱了并发性。

清单 5.4 使用客户端加锁进行迭代

```
synchronized (vector) {  
    for (int i = 0; i < vector.size(); i++)  
        doSomething(vector.get(i));  
}
```

5.1.2 迭代器和 ConcurrentModificationException

尽管 Vector 是一个“遗留”的容器类，但为了将问题阐述清楚，我们在很多例子中都使用了它。其实，更多“现代”的容器类也并没有消除复合操作产生的问题。对 Collection 进行迭代的标准方式是使用 Iterator，无论是显式地使用还是通过 Java 5.0 引入新的 for-each 循环语法。但是当有其他线程可能并发修改容器时，使用迭代器(iterator)仍不可避免地需要在迭代期间对容器加锁。在设计同步容器返回的迭代器时，并没有考虑到并发修改的问题，它们是“及时失败(fail-fast)”的——意思是当它们察觉容器在迭代开始后被修改，会抛出一个未检查的 ConcurrentModificationException。

这些“及时失败”的迭代器，并没有很好的稳定性设计——它们被设计用来在“善意运用”的情况下捕获并发的错误，因此只作为并发问题的提前预警。这是通过把修改计数器(modification count)与容器关联起来实现的：如果在迭代期间计数器被修改，hasNext 或 next 会抛出一个 ConcurrentModificationException。不过，这样的检查是在没有同步的情况下进行的，所以存在一定风险，看到修改计数器上的过期数据，导致迭代器并没有发现修改。这里进行过深思熟虑的权衡，希望能减少那些监测并发修改的代码给性能带来的影响³。

清单 5.5 阐释了对容器进行迭代的 for-each 循环语法。javac 生成的代码内在地使用一个 Iterator，重复调用 hasNext 和 next 对 List 进行迭代。正像 Vector 中的迭代一样，为了避免出现 ConcurrentModificationException，需要在迭代期间持有一个容器锁。

清单 5.5 用 Iterator 对 List 进行迭代

```
List<Widget> widgetList  
    = Collections.synchronizedList(new ArrayList<Widget>());  
...  
// 可能抛出 ConcurrentModificationException  
for (Widget w : widgetList)  
    doSomething(w);
```



³ ConcurrentModificationException也可能出现在单线程的代码中；当对象并非通过 Iterator.remove，而是被直接从容器中删除时，就会出现这个情况。

但是，有一些原因造成我们不愿意在迭代期间对容器加锁。当其他线程需要访问容器时，必须等待，直到迭代结束；如果容器很大，或者对每一个元素执行的任务耗时比较长，它们可能需要等待很长一段时间。同样，如果容器像清单 5.4 中那样被锁，doSomething 调用还要持有另一个锁，这是一个产生死锁风险的因素（参见第 10 章）。即使没有饥饿（starvation）或者死锁的风险，在相当长的一段时间内把容器锁住，也会破坏程序的可伸缩性。保持锁的时间越长，对锁的竞争就可能会越激烈，并且如果很多线程在等待锁的时候阻塞，吞吐量和 CPU 的效能都会遭受影响（参见第 11 章）。

在迭代期间，对容器加锁的一个替代办法是复制容器。因为复制是线程限制（thread-confined）的，没有其他的线程能够在迭代期间对其进行修改，这样消除了 ConcurrentModificationException 发生的可能性。（容器仍然需要在复制期间对自己加锁）。复制容器会有明显的性能开销；这样做是好是坏取决于许多因素，包括容器的大小、每一个元素的工作量、迭代操作相对于容器其他操作的频率，以及响应性和吞吐量的需求。

5.1.3 隐藏迭代器

尽管锁可以避免迭代器抛出 ConcurrentModificationException，但你必须还要记得，在一个可能发生迭代的共享容器中，各处都需要使用锁。这个问题比听上去要棘手得多，因为迭代器有时候是隐藏的，就像清单 5.6 中的 HiddenIterator。在 HiddenIterator 中没有显式的迭代，但是粗体字的代码承担与迭代相同的功能。字符串的拼接操作经过编译转换成调用 StringBuilder.append(Object) 来完成，它会调用容器的 toString 方法。标准容器中的 toString 的实现会通过迭代容器中的每个元素，来获得关于容器内容格式良好的展现。

addTenThings 方法可能会抛出 ConcurrentModificationException，因为容器是由 toString 进行迭代的，这些发生在生成调试消息的过程中。当然，真正的问题在于 HiddenIterator 不是线程安全的；在调用 println 时，必须在使用 set 之前请求到 HiddenIterator 的锁，但是调试和记录日志的代码通常不会这么做。这里真正的教训是，状态和保护它的同步之间差距越大，人们越容易忘记在访问状态时正确使用同步。如果 HiddenIterator 把 HashSet 包装为 synchronizedSet，封装了同步，这种错误就不会发生了。

正如封装一个对象的状态，能够使它更加容易地保持不变约束一样，封装它的同步则可以迫使它符合同步策略。

清单 5.6 迭代隐藏在字符串的拼接中（不要这样做）

```
public class HiddenIterator {
    @GuardedBy("this")
    private final Set<Integer> set = new HashSet<Integer>();

    public synchronized void add(Integer i) { set.add(i); }
    public synchronized void remove(Integer i) { set.remove(i); }

    public void addTenThings() {
        Random r = new Random();
        for (int i = 0; i < 10; i++)
            add(r.nextInt());
        System.out.println("DEBUG: added ten elements to " + set);
    }
}
```



容器的 `hashCode` 和 `equals` 方法也会间接地调用迭代，比如当容器本身作为一个元素时，或者作为另一个容器的 `key` 时。类似地，`containsAll`、`removeAll`、`retainAll` 方法，以及把容器作为参数的构造函数，都会对容器进行迭代。所有这些对迭代的间接调用，都可能会引起 `ConcurrentModificationException`。

5.2 并发容器

Java 5.0 通过提供几种**并发的**容器类来改进同步容器。同步容器通过对容器的所有状态进行串行访问，从而实现了它们的线程安全。这样做的代价是削弱了并发性，当多个线程共同竞争容器级的（collection-wide）锁时，吞吐量就会降低。

另一方面，并发容器是为多线程并发访问而设计的。Java 5.0 添加了 `ConcurrentHashMap`，来替代同步的哈希 Map 实现；当多数操作为读取操作时，`CopyOnWriteArrayList` 是 `List` 相应的同步实现。新的 `ConcurrentMap` 接口加入了对常见复合操作的支持，比如“缺少即加入（`put-if-absent`）”、替换和条件删除。

用并发容器替换同步容器，这种作法以有很小风险带来了可扩展性显著的提高。

Java 5.0 同样添加了两个新的容器类型：`Queue` 和 `BlockingQueue`。`Queue` 用来临时保存正在等待被进一步处理的一系列元素。JDK 提供了几种实现，包括一个传统的 FIFO 队列，`ConcurrentLinkedQueue`；一个（非并发）具有优先级顺序的队列，`PriorityQueue`。

Queue 的操作并不会阻塞；如果队列是空的，那么从队列中获取元素的操作会返回空值（`null`）。尽管你可以用 `List` 来模拟 Queue 的行为——事实上，`LinkedList` 就实现了 Queue——但是我们还是需要 Queue 的类，因为如果忽略掉 `List` 的随机访问需求的话，使用 Queue 能得到更加高效的并发实现。

`BlockingQueue` 扩展了 Queue，增加了可阻塞的插入和获取操作。如果队列是空的，一个获取操作会一直阻塞直到队列中存在可用元素；如果队列是满的（对于有界队列），插入操作会一直阻塞直到队列中存在可用空间。阻塞队列在生产者-消费者设计中非常有用，在 5.3 节会有更详细的介绍。

正像 `ConcurrentHashMap` 作为同步的哈希 Map 的一个并发替代品，Java 6 加入了 `ConcurrentSkipListMap` 和 `ConcurrentSkipListSet`，用来作为同步的 `SortedMap` 和 `SortedSet` 的并发替代品（比如用 `synchronizedMap` 包装的 `TreeMap` 或 `TreeSet`）。

5.2.1 ConcurrentHashMap

同步容器类在每个操作的执行期间都持有一个锁。有一些操作，比如 `HashMap.get` 或者 `List.contains`，可能会涉及到比预想更多的工作量：为寻找一个特定对象而遍访整个哈希容器或清单，必须调用大量候选对象的 `equals`（`equals` 本身还涉及相当数量的计算）。在一个哈希容器中，如果 `hashCode` 没有能很好地分散哈希值，元素很可能不均衡地分布到整个容器中；最极端的情况是，一个不良的哈希函数将会把一个哈希表转化为一个线性链表。遍历一个很长的清单并调用其中部分或者全部元素的 `equals` 方法，这会花费很长时间，并且在这段时间内，其他线程都不能访问这个容器。

`ConcurrentHashMap` 和 `HashMap` 一样是一个哈希表，但是它使用完全不同的锁策略，可以提供更好的并发性和可伸缩性。在 `ConcurrentHashMap` 以前，程序使用一个公共锁同步每一个方法，并严格地限制只能有一个线程可以同时访问容器。而 `ConcurrentHashMap` 使用一个更加细化的锁机制，名叫**分离锁**（参见 11.4.3 节）。这个机制允许更深层次的共享访问。任意数量的读线程可以并发访问 Map，读者和写者也可以并发访问 Map，并且有限数量的写线程还可以并发修改 Map。结果是，为并发访问带来更高的吞吐量，同时几乎没有损失单个线程访问的性能。

`ConcurrentHashMap` 与其他的并发容器一起，进一步改进了同步容器类：提供不会抛出 `ConcurrentModificationException` 的迭代器，因此不需要在容器迭代中加锁。`ConcurrentHashMap` 返回的迭代器具有**弱一致性**，而非“及时失败”的。弱一致性的迭代器可以容许并发修改，当迭代器被创建时，它会遍历已有的元素，并且可以（但是不保证）感应到在迭代器被创建后，对容器的修改。

尽管有这么多改进，我们仍然有一些需要权衡的地方。那些对整个 Map 进行操作的

方法，比如 `size` 和 `isEmpty`，它们的语义在反映容器并发特性上被轻微地弱化了。因为 `size` 的结果相对于在计算的时候可能已经过期，它仅仅只是一个估算值，所以允许 `size` 返回一个近似值而不是一个精确值。这在一开始让人有些困扰，不过事实上像 `size` 和 `isEmpty` 这样的方法在并发环境下几乎没有什么用处，因为它们的目标是运动的。所以对这些操作的需求被弱化了。相反，应该保证对最重要的操作进行性能优化，首先就包括 `get`、`put`、`containsKey` 和 `remove`。

同步 `Map` 实现提供的一个特性是为独占的访问加锁，这在 `ConcurrentHashMap` 中并没有实现。在 `Hashtable` 和 `synchronizedMap` 中，获得 `Map` 的锁就可以防止任何其他线程访问该 `Map`。这对于一些罕见的情况来说是必要的，比如原子化地加入一些映射（`mapping`），或者对元素进行若干次迭代，在这期间需要看到元素以同样的顺序出现。尽管如此，从总体来看，下面的权衡还是合理的：应该期望去连续修改并发容器中的内容。

相比于 `Hashtable` 和 `synchronizedMap`，`ConcurrentHashMap` 有众多的优势，而且几乎不存在什么劣势，因此在大多数情况下用 `ConcurrentHashMap` 取代同步 `Map` 实现只会带来更好的可伸缩性。只有当你的程序需要在独占访问³中加锁时，`ConcurrentHashMap` 才无法胜任。

5.2.2 Map 附加的原子操作

因为 `ConcurrentHashMap` 不能够在独占访问中被加锁，我们不能使用客户端加锁来创建新的原子操作，比如我们在 4.4.1 节里面对 `Vector` 做的“缺少即加入”操作。不过一些常见的复合操作，比如“缺少即加入”，“相等便移除（`remove-if-equal`）”和“相等便替换（`replace-if-equal`）”，都已被实现为原子操作，并且这些操作已在 `ConcurrentMap` 接口中声明，如清单 5.7 所示。如果你发现自己正在已有的同步 `Map` 实现中加入这样一个功能，那么这可能标志着你应该考虑使用一个 `ConcurrentMap` 替代手头的同步 `Map`。

5.2.3 CopyOnWriteArrayList

`CopyOnWriteArrayList` 是同步 `List` 的一个并发替代品，通常情况下它提供了更好的并发性，并避免了在迭代期间对容器加锁和复制。（相似地，`CopyOnWriteArraySet` 是同步 `Set` 的一个并发替代品。）

“写入时复制（`copy-on-write`）”容器的线程安全性来源于这样一个事实，只要有效的不可变对象被正确发布，那么访问它将不再需要更多的同步。在每次需要修改时，它们会创建并重新发布一个新的容器拷贝，以此来实现可变性。“写入时复制（`copy-on-write`）”容器的迭代器保留一个底层基础数组（`the backing array`）的引用。这个数组作为迭代器的起

³ 或者如果你依赖于同步 `Map` 实现的同步化边界效应。

清单 5.7 ConcurrentMap 接口

```
public interface ConcurrentMap<K,V> extends Map<K,V> {  
    // 只有当没有找到匹配 K 的值时才插入  
    V putIfAbsent(K key, V value);  
  
    // 只有当 K 与 V 匹配时才移除  
    boolean remove(K key, V value);  
  
    // 只有当 K 与 oldValue 匹配时才取代  
    boolean replace(K key, V oldValue, V newValue);  
  
    // 只有当 K 匹配某值时才取代  
    V replace(K key, V newValue);  
}
```

点，永远不会被修改，因此对它的同步只不过是确保了数组内容的可见性。因此，多个线程可以对这个容器进行迭代，并且不会受到另一个或多个想要修改容器的线程带来的干涉。“写入时复制”容器返回的迭代器不会抛出 `ConcurrentModificationException`，并且返回的元素严格与迭代器创建时相一致，不会考虑后续的修改。

显而易见，在每次容器改变时复制基础数组（backing array）需要一定的开销，特别是当容器比较大的时候；当对容器迭代操作的频率远远高于对容器修改的频率时，使用“写入时复制”容器是个合理的选择。这个准则准确描述了许多事件通知系统：递交一个通知需要迭代已注册的监听器，并调用其中每一个。在多数情况下，注册和注销一个事件监听器的次数要比收到事件通知的次数少很多。（更多的关于“写入时复制”参见[CPJ 2.4.4]。）

5.3 阻塞队列和生产者-消费者模式

阻塞队列（Blocking queue）提供了可阻塞的 `put` 和 `take` 方法，它们与可定时的 `offer` 和 `poll` 是等价的。如果 `Queue` 已经满了，`put` 方法会被阻塞直到有空间可用；如果 `Queue` 是空的，那么 `take` 方法会被阻塞，直到有元素可用。`Queue` 的长度可以有限，也可以无限；无限的 `Queue` 永远不会充满，所以它的 `put` 方法永远不会阻塞。

阻塞队列支持**生产者-消费者**设计模式。一个生产者-消费者设计分离了“识别需要完成的工作”和“执行工作”。该模式不会发现一个工作便立即处理，而是把工作置入一个任务（“to do”）清单中，以备后期处理。生产者-消费者模式简化了开发，因为它解除了生产者类和消费者类之间相互依赖的代码。生产者和消费者以不同的或者变化的速度生

产和消费着数据，生产者-消费者模式将这些活动解耦，因而简化了工作负荷的管理。

生产者-消费者设计是围绕阻塞队列展开的，生产者把数据放入队列，并使数据可用，当消费者为适当的行为做好准备时会从队列中获取数据。生产者不需要知道消费者的身份或者数量，甚至根本没有消费者——它们只负责把数据放入队列。类似地，消费者也不需要知道生产者是谁，以及是谁给它们安排的工作。BlockingQueue 可以使用任意数量的生产者和消费者，从而简化了生产者-消费者设计的实现。最常见的生产者-消费者设计是将线程池与工作队列相结合；讲述 Executor 任务执行框架时会具体介绍这个模式，这是第6章和第8章的主题。

两个人洗盘子的劳动分工也是一个与生产者-消费者设计相类似的例子：一个人洗盘子，并把洗好的盘子放在盘子架上，另一个人从架子上得到盘子，并把它烘干。在这个场景中，盘子架充当了阻塞队列；如果架子上没有盘子，消费者会一直等待，直到有盘子需要烘干，如果架子被放满了，生产者会停止清洗直到架子上拥有新空间。这样我们可以类推扩展到多个生产者（尽管可能存在对水槽的竞争）和多个消费者；每一个工人只与盘子架产生互动。他们不需要知道究竟存在多少生产者和消费者，或者谁生产了某个给定工作条目。

“生产者”和“消费者”的标签是相对的；在某个上下文中存在一个作为消费者的活动，可能在另一个上下文中成为生产者。烘干盘子“消费”干净的湿盘子，产生干盘子。第三个人希望能帮助把干净的盘子收起来，这样的话，现在就有两个共享的工作队列(每个都可能阻塞烘干机的运行。)

阻塞队列简化了消费者的编码，因为 take 会保持阻塞直到可用数据出现。如果生产者不能足够快地产生工作，让消费者忙碌起来，那么消费者只能一直等待，直到有工作可做。有时候，这让人非常愉快（当服务器应用程序没有客户请求服务时），有时候这种情况也表明生产者线程和消费者线程的数量需要进行调整，以获得更好的资源利用率。在一个 Web Crawler（网络蜘蛛）或者其他应用程序中，总有许多事情，永远做不完。

如果生产者产生工作的速度总是比消费者处理的速度快，那么应用程序的工作条目会排在一个没有边界的队列中，最终耗尽内存。同样地，put 方法的阻塞特性大大简化了生产者的编码；如果我们使用一个**有界队列**，那么当队列充满的时候，生产者就会阻塞，暂不能生成更多的工作，从而给消费者时间来追赶进度。

阻塞队列同样提供了一个 offer 方法，如果条目不能被加入到队列里，它会返回一个失败状态。这使得你能够创建更多灵活的策略来处理超负荷工作，比如减轻负载，序列化剩余工作条目并写入硬盘，减少生产者线程，或者用其他的方法遏制生产者线程。

有界队列是强大的资源管理工具，用来建立可靠的应用程序：它们遏制那些可以产生过多工作量、具有威胁的活动，从而让你的程序在面对超负荷工作时更加健壮。

虽然生产者-消费者模式可以把生产者和消费者的**代码**相互解耦合，但是它们的行为**还是**间接地通过共享工作队列耦合在一起了。它理想地假定消费者将会持续工作，所以你不需为工作队列的大小划定边界，但是这将成为日后需要重新架构系统的预兆。**在你的设计初期就使用阻塞队列建立对资源的管理——提早做这件事情会比日后再修复容易得多。**在某些情况下，阻塞队列使这更加简单，但是如果阻塞队列并不完全适合于你的设计，你也可以用信号量（Semaphore）创建其他的阻塞数据结构（参见 5.5.3 节）。

类库中包含一些 BlockingQueue 的实现，其中 LinkedBlockingQueue 和 ArrayBlockingQueue 是 FIFO 队列，与 LinkedList 和 ArrayList 相似，但是却拥有比同步 List 更好的并发性能。PriorityBlockingQueue 是一个按优先级顺序排序的队列，当你不希望按照 FIFO 的顺序处理元素时，这个 PriorityBlockingQueue 是非常有用的。正如其他排序的容器一样，PriorityBlockingQueue 可以比较元素本身的自然顺序（如果它们实现了 Comparable），也可以使用一个 Comparator 进行排序。

最后一个 BlockingQueue 的实现是 SynchronousQueue，它根本上不是一个真正的队列，因为它不会为队列元素维护任何存储空间。不过，它维护一个排队的**线程**清单，这些线程等待把元素加入（enqueue）队列或者移出（dequeue）队列。在洗盘子的比喻中，这好比没有盘子架，但是却把洗好的盘子直接放入下一步的空闲烘干机。这种实现队列的方式看起来很奇怪。它非常直接地移交工作，减少了在生产者和消费者之间移动数据的延迟时间。（在传统队列中，加入和移出操作必须在一个工作单元移交之前顺序地完成。）直接地移交同样会给生产者带来更多关于任务状态的反馈信息；当移交被接受，它就知道消费者已经得到了任务，而不是简单地把任务放在一个队列或是什么其他地方——就像是把文件直接递给你的同事，还是把文件发送到她的邮箱期待她一会可以得到此文件之间的不同。因为 SynchronousQueue 没有存储的能力，所以除非另一个线程已经准备好参与移交工作，否则 put 和 take 会一直阻止。SynchronousQueue 这类队列只有在消费者充足的时候比较合适，它们总能为下一个任务作好准备。

5.3.1 实例：桌面搜索

有一种类型的程序适合分解为生产者和消费者：扫描本地驱动器并归档文件，为之后的搜索建立索引的代理，这类似于 Google Desktop 或者 Windows 索引服务。清单 5.8 中的

DiskCrawler 表现了一个生产者任务，这个任务是搜索一个文件结构，找到符合给定标准的文件并把它们的名称放入工作队列；清单 5.8 中的 Indexer 展现的是消费者从队列中取出文件名称并制作索引的任务。

生产者-消费者模式提供了线程友好的手段，从而能够将桌面搜索问题细化成更简单的组件。与其让一个大活动完成各项工作，不如把文件查找和建立索引的事情划分给不同的活动，这会使代码更合理，具有可读性；每一个活动只有一个单一的任务，并且阻塞队列掌控所有的控制流，所以每一个活动的代码都会更简单更清晰。

生产者-消费者模式同样带来了一些性能方面的提高。生产者和消费者可以并发地执行，如果一个受限于 I/O，另一个受限于 CPU，那么并发执行的全部产出会高于顺序执行的产出。如果生产者和消费者在不同层面并行执行，那么紧密地耦合会减弱并行性，减少并行化的活动。

清单 5.9 开始了几个搜索者 (crawler) 和索引建立者，它们在各自的线程中运行。正如我们提到的，消费者线程不会退出，以避免程序终止，我们将在第 7 章介绍一些技术来解决这个问题。使用这个例子的时候明确地对线程进行了管理，许多生产者和消费者设计都可以通过使用 Executor 任务执行框架实现，它自身就应用了生产者-消费者模式。

5.3.2 连续的线程限制

在 `java.util.concurrent` 中实现的阻塞队列，全部都包含充分的内部同步，从而能安全地将对象从生产者线程发布至消费者线程。

对于可变对象，生产者-消费者设计和阻塞队列一起，为生产者和消费者之间移交对象所有权提供了**连续的线程限制** (serial thread confinement)。一个线程约束的对象完全由单一线程所有，但是所有权可以通过安全的发布被“转移”，这样其他线程中只有唯一一个能够得到访问这个对象的权限，并且保证移交之后原线程不能再访问它。安全发布确保了对象状态对新的所有者是可见的，并且因为原始的所有者不会再触及它，这样使得对象完全受限于新线程中。这个新的主人可以任意修改，因为它具有独占访问权。

对象池拓展了连续的线程限制，把对象“借给”一个请求线程。只要对象池中含有充足的内部同步，使对象池能够安全地发布，并且只要客户端本身不会发布对象池，或者在返回对象池后不再继续使用，所有权可以在线程间安全地传递。

我们也可以使用其他的发布机制来传递可变对象的所有权，但是必须确保只有一个线程接收到了对象的移交。阻塞队列简化了这项工作；只需要多一点工作，它就可以通过 `ConcurrentMap` 的原子方法 `remove` 或 `AtomicReference` 的原子方法 `compareAndSet` 来完成。

清单 5.8 桌面搜索应用程序中的生产者和消费者

```
public class FileCrawler implements Runnable {
    private final BlockingQueue<File> fileQueue;
    private final FileFilter fileFilter;
    private final File root;
    ...
    public void run() {
        try {
            crawl(root);
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }
    }

    private void crawl(File root) throws InterruptedException {
        File[] entries = root.listFiles(fileFilter);
        if (entries != null) {
            for (File entry : entries)
                if (entry.isDirectory())
                    crawl(entry);
                else if (!alreadyIndexed(entry))
                    fileQueue.put(entry);
        }
    }
}

public class Indexer implements Runnable {
    private final BlockingQueue<File> queue;
    public Indexer(BlockingQueue<File> queue) {
        this.queue = queue;
    }
    public void run() {
        try {
            while (true)
                indexFile(queue.take());
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }
    }
}
```

清单 5.9 开始桌面搜索

```
Public static void startIndexing(File[] roots) {
    BlockingQueue<File> queue = new LinkedBlockingQueue<File>(BOUND);
    FileFilter filter = new FileFilter() {
        public boolean accept(File file) { return true; }
    };

    for (File root : roots)
        new Thread(new FileCrawler(queue, filter, root)).start();
    for (int i = 0; i < N_CONSUMERS; i++)
        new Thread(new Indexer(queue)).start();
}
```

5.3.3 双端队列和窃取工作

Java 6 同样新增了两个容器类型，Deque（发音是 deck）和 BlockingDeque，它们分别扩展了 Queue 和 BlockingQueue。Deque 是一个双端队列，允许高效地在头和尾分别进行插入和移除。实现它们的是 ArrayDeque 和 LinkedBlockingDeque。

正如阻塞队列适用于生产者-消费者模式一样，双端队列使它们自身与一种叫做**窃取工作**（work stealing）的模式相关联。一个消费者生产者设计中，所有的消费者只共享一个工作队列；在窃取工作的设计中，每一个消费者都有一个自己的双端队列。如果一个消费者完成了自己双端队列中的全部工作，它可以偷取其他消费者的双端队列中的**末尾**任务。因为工作者线程并不会竞争一个共享的任务队列，所以**窃取工作模式**比传统的生产者-消费者设计有更佳的可伸缩性；大多数时候它们访问自己的双端队列，减少竞争。当一个工作者必须要访问另一个队列时，它会从尾部截取，而不是从头部，从而进一步降低对双端队列的争夺。

窃取工作恰好适合用于解决消费者与生产者同体的问题——当运行到一个任务的某单元时，可能会识别出更多的任务。比如，Web Crawler 处理一个页面时，通常会发现有更多页面可以搜索。类似的还有许多图形扫描算法，比如在垃圾回收时对堆做了记号，可以并行使用窃取工作。当一个线程发现了一个新的任务单元时，它会把它放在自己队列的末尾（或者在另一种共享工作设计的情况下，可以放入其他工作者的队列中）；当双端队列为空的时候，它会去其他队列的队尾寻找新的任务，这样能确保每一个线程都保持忙碌状态。

5.4 阻塞和可中断的方法

线程可能会因为几种原因被**阻塞**或暂停：等待 I/O 操作结束，等待获得一个锁，等待从 Thread.sleep 中唤醒，或者是等待另一个线程的计算结果。当一个线程阻塞时，它通常被挂起，并被设置成线程阻塞的某个状态（BLOCKED、WAITING，或是 TIMED_WAITING）。

一个阻塞的操作和一个普通的操作之间的差别仅仅在于，被阻塞的线程必须等待一个事件的发生才能继续进行，并且这个事件是超越它自己控制的，因而需要花费更长的时间——等待 I/O 操作完成，锁可用，或者是外部计算结束。当外部事件发生后，线程被置回 `RUNNABLE` 状态，重新获得调度的机会。

`BlockingQueue` 的 `put` 和 `take` 方法会抛出一个受检查的 `InterruptedException`，这与类库中其他的一些方法是相同的，比如 `Thread.sleep`。当一个方法能够抛出 `InterruptedException` 的时候，是在告诉你这个方法是一个可阻塞方法，进一步看，如果它被**中断**，将可以提前结束阻塞状态。

`Thread` 提供了 `interrupt` 方法，用来中断一个线程，或者查询某线程是否已经被中断。每一个线程都有一个布尔类型的属性，这个属性代表了线程的中断状态；中断线程时需要设置这个值。

中断是一种**协作机制**。一个线程不能够迫使其他线程停止正在做的事情，或者去做其他事情；当线程 *A* 中断 *B* 时，*A* 仅仅是要求 *B* 在达成某个方便停止的关键点时，停止正在做的事情——如果它这样做是正确的。然而，在 API 或者语言规范中，任何特定应用级别的中断语义中，最常用的是取消一个活动。从时间角度来看，响应中断的阻塞方法，可以更容易地取消耗时的活动。

当你在代码中调用了一个会抛出 `InterruptedException` 的方法时，你自己的方法也就成为了一个阻塞方法，要为响应中断作好准备。在类库代码中，有两种基本选择：

传递 `InterruptedException`。如果你能侥幸避开异常的话，这通常是最明智的策略——只需要把 `InterruptedException` 传递给你的调用者。这可能包括不捕获 `InterruptedException`，也可能是先捕获，然后对其中特定活动进行简洁地清理后，再抛出。

恢复中断。有时候你不能抛出 `InterruptedException`，比如当你的代码是 `Runnable` 的一部分时。在这样的情况下，你必须捕获 `InterruptedException`，并且，在当前线程中通过调用 `interrupt` 从中断中恢复，这样调用栈中更高层的代码可以发现中断已经发生。清单 5.10 示范了这种情况。

你还可以有更加复杂的中断处理方案，不过上述的两种方案差不多可以应付大多数情况了。但是你不应该这样使用 `InterruptedException`——捕获它，但不作任何响应。这样做会丢失线程中断的证据，从而剥夺了上层栈的代码处理中断的机会。**只有一种情况允许掩盖中断：你扩展了 `Thread`，并因此控制了所有处于调用栈上层的代码。**取消和中断会在第 7 章中更详细地讲述。

清单 5.10 恢复中断状态，避免掩盖中断

```
public class TaskRunnable implements Runnable {
    BlockingQueue<Task> queue;
    ...
    public void run() {
        try {
            processTask(queue.take());
        } catch (InterruptedException e) {
            // 恢复中断状态
            Thread.currentThread().interrupt();
        }
    }
}
```

5.5 Synchronizer

阻塞队列在容器类中是独一无二的：它们不仅作为对象的容器，而且能够协调生产者线程和消费者线程之间的控制流，这是因为 `take` 和 `put` 会保持阻止状态直到队列进入了期望的状态（不空也不满）。

Synchronizer 是一个对象，它根据本身的状态调节线程的控制流。阻塞队列可以扮演一个 **Synchronizer** 的角色；其他类型的 **Synchronizer** 包括信号量（semaphore）、关卡（barrier）以及闭锁（latch）。在平台类库中存在一些 **Synchronizer** 类；如果这些不能满足你的需要，你同样可以按照第 14 章里描述的那样，创建一个你自己的 **Synchronizer**。

所有 **Synchronizer** 都享有类似的结构特性：它们封装状态，而这些状态决定着线程执行到在某一点时是通过还是被迫等待；它们还提供操控状态的方法，以及高效地等待 **Synchronizer** 进入到期望状态的方法。

5.5.1 闭锁

闭锁（latch）是一种 **Synchronizer**，它可以延迟线程的进度直到线程到达**终止**（terminal）状态 [CPJ 3.4.2]。一个闭锁工作起来就像一道大门：直到闭锁达到终点状态之前，门一直是关闭的，没有线程能够通过，在终点状态到来的时候，门开了，允许所有线程都通过。一旦闭锁到达了终点状态，它就不能够再改变状态了，所以它会永远保持敞开状态。闭锁可以用来确保特定活动直到其他的活动完成后才发生，比如：

- 确保一个计算不会执行，直到它需要的资源被初始化。一个二元闭锁（两个状态）可以用来表达“资源 *R* 已经被初始化”，并且所有需要用到 *R* 的活动首先都要在闭锁中等待。

- 确保一个服务不会开始，直到它依赖的其他服务都已经开始。每一个服务会包含一个相关的二元闭锁；开启服务 S 会首先开始等待闭锁 S 中所依赖的其他服务，在启动结束后，会释放闭锁 S ，这样所有依赖 S 的服务也可以开始处理了。
- 等待，直到活动的所有部分都为继续处理作好充分准备，比如在多玩家的游戏中的所有玩家是否都准备就绪。这样的闭锁会在所有玩家准备就绪时，达到终点状态。

`CountDownLatch` 是一个灵活的闭锁实现，用于上述各种情况；允许一个或多个线程等待一个事件集的发生。闭锁的状态包括一个计数器，初始化为一个正数，用来表现需要等待的事件数。`countDown` 方法对计数器做减操作，表示一个事件已经发生了，而 `await` 方法等待计数器达到零，此时所有需要等待的事件都已发生。如果计数器入口时值为非零，`await` 会一直阻塞直到计数器为零，或者等待线程中断以及超时。

清单 5.11 中，`TestHarness` 阐释了闭锁的两种常见用法。`TestHarness` 创建了一些线程，并发地执行给定的任务。它使用两个闭锁，一个“开始阀门”和一个“结束阀门”。这个开始阀门将计数器初始化为 1。结束阀门将计数器初始化为工作线程的数量。每一个工作线程要做的第一件事情是等待开始阀门打开；这样做能确保直到所有线程都做好准备时，才开始工作。每个线程的最后一个工作是为结束阀门减一；这样做使控制线程有效地等待，直到最后一个工作线程完成任务，这样就能计算整个的用时了。

在 `TestHarness` 中，我们为什么不在线程创建后就立即运行，却要自寻烦恼地使用 `TestHarness` 呢？或许因为我们想要计算在线程 n 倍并发的情况下执行一个任务的时间。如果我们简单地创建并启动线程，那么先启动的就比后启动的具有“领先优势”，并且根据活动线程数的增加或者减少，这样的竞争度也在不断改变。开始阀门让控制线程能够同时释放所有工作者线程，结束阀门让控制线程能够等待最后一个线程完成任务，而不是顺序等待每一个线程结束。

5.5.2 FutureTask

`FutureTask` 同样可以作为闭锁。（`FutureTask` 的实现描述了一个抽象的可携带结果的计算 [CPJ 4.3.3]）。`FutureTask` 的计算是通过 `Callable` 实现的，它等价于一个可携带结果的 `Runnable`，并且有 3 个状态：等待、运行和完成。完成包括所有计算以任意的方式结束，包括正常结束、取消和异常。一旦 `FutureTask` 进入完成状态，它会永远停止在这个状态上。

`Future.get` 的行为依赖于任务的状态。如果它已经完成，`get` 可以立刻得到返回结果，否则会被阻塞直到任务转入完成状态，然后会返回结果或者抛出异常。`FutureTask` 把计算

清单 5.11 在时序测试中，使用 CountdownLatch 来启动和停止线程

```
public class TestHarness {
    public long timeTasks(int nThreads, final Runnable task)
        throws InterruptedException {
        final CountdownLatch startGate = new CountdownLatch(1);
        final CountdownLatch endGate = new CountdownLatch(nThreads);
        for (int i = 0; i < nThreads; i++) {
            Thread t = new Thread() {
                public void run() {
                    try {
                        startGate.await();
                        try {
                            task.run();
                        } finally {
                            endGate.countDown();
                        }
                    } catch (InterruptedException ignored) { }
                }
            };
            t.start();
        }
        long start = System.nanoTime();
        startGate.countDown();
        endGate.await();
        long end = System.nanoTime();
        return end-start;
    }
}
```

的结果从运行计算的线程传送到需要这个结果的线程；FutureTask 的规约保证了这种传递建立在结果的安全发布基础之上。

Executor 框架利用 FutureTask 来完成异步任务，并可以用来进行任何潜在的耗时计算，而且可以在真正需要计算结果之前就启动它们开始计算。清单 5.12 中，Preloader 使用了 FutureTask 来执行一个代价昂贵的计算，结果稍后会被用到；尽早开始计算，你可以减少等待结果所需要花费的时间。

清单 5.12 使用 FutureTask 预载稍后需要的数据

```
public class Preloader {
    private final FutureTask<ProductInfo> future =
        new FutureTask<ProductInfo>(new Callable<ProductInfo>() {
            public ProductInfo call() throws DataLoadException {
                return loadProductInfo();
            }
        });
    private final Thread thread = new Thread(future);
    public void start() { thread.start(); }
    public ProductInfo get()
        throws DataLoadException, InterruptedException {
        try {
            return future.get();
        } catch (ExecutionException e) {
            Throwable cause = e.getCause();
            if (cause instanceof DataLoadException)
                throw (DataLoadException) cause;
            else
                throw launderThrowable(cause);
        }
    }
}
```

Preloader 创建了一个 FutureTask，记录从数据库加载结果的信息，还创建了一个将要执行运算的线程。在构造函数或者静态初始化方法中启动线程并不是明智的举措，所以它提供了 start 方法启动线程。当程序经过一段时间后需要 ProductInfo 时，它可以调用 get，如果已经加载就绪的话，就会返回这些数据，否则会等待加载结束再返回。

Callable 记录的这些任务，可以抛出受检查或者未受检查的异常，并且任何代码都可能抛出 Error。无论执行任务的代码抛出什么，它都被封装为一个 ExecutionException，

并被 `Future.get` 重新抛出。调用 `get` 的代码如此复杂，不仅因为它必须处理可能出现的 `ExecutionException`（和未受检查的 `CancellationException`），而且还因为 `ExecutionException` 的诱因是作为 `Throwable` 返回的，`Throwable` 处理起来并不方便。

在 `Preloader` 中，如果 `get` 抛出一个 `ExecutionException`，那么原因有以下 3 种：`Callable` 抛出的受检查的异常，或是一个 `RuntimeException`，或者一个 `Error`。我们必须对这 3 种情况进行分别处理，但是我们将使用清单 5.13 中的 `launderThrowable` 方法来封装复杂的异常处理逻辑。在调用 `launderThrowable` 之前，`Preloader` 先检查已知异常，并重新抛出。这样只留下那些未受检查的异常，`Preloader` 通过调用 `launderThrowable` 来处理它们，并抛出未经检验的异常。如果 `Throwable` 向 `launderThrowable` 传递了一个 `Error`，`launderThrowable` 会直接再将其抛出；如果不是一个 `RuntimeException`，它会抛出一个 `IllegalStateException` 用来指出这是一个逻辑错误。这样就只剩下 `RuntimeException` 了，`launderThrowable` 会把它返回给它的调用者，通常由调用者再将其抛出。

清单 5.13 `Throwable` 强制转换为 `RuntimeException`

```
/** 如果 Throwable 是 Error，那么将它抛出；如果；
 * 是 RuntimeException 那么返回，否则抛出 IllegalStateException
 */
public static RuntimeException launderThrowable(Throwable t) {
    if (t instanceof RuntimeException)
        return (RuntimeException) t;
    else if (t instanceof Error)
        throw (Error) t;
    else
        throw new IllegalStateException("Not unchecked", t);
}
```

5.5.3 信号量

计数信号量（Counting semaphore）用来控制能够同时访问某特定资源的活动的数量，或者同时执行某一给定操作的数量 [CPJ 3.4.1]。计数信号量可以用来实现资源池或者给一个容器限定边界。

一个 `Semaphore` 管理一个有效的**许可**（permit）集；许可的初始量通过构造函数传递给 `Semaphore`。活动能够获得许可（只要还有剩余许可），并在使用之后释放许可。如果已经没有可用的许可了，那么 `acquire` 会被阻塞，直到有可用的为止（或者直到被中断或者操作超时）。`release` 方法向信号量返回一个许可⁴。计算信号量的一种退化形式是二

⁴ 实现起来并没有真正的许可对象，并且 `Semaphore` 并没有真正向线程分配许可，所以一个线程得到的许可，可能是其他线程释放的。你可以把 `acquire` 看作是消费掉一个许可，而 `release` 看作创建一个许可；许可的数量并不会限制 `Semaphore`。

元信号量：一个计数初始值为 1 的 Semaphore。二元信号量可用作**互斥**（mutex）锁，它有不可重入锁的语意：谁拥有这个唯一的许可，就拥有了互斥锁。

信号量可以用来实现资源池，比如数据库连接池。有一个定长的池，当它为空时，你向它请求资源会失败。构建这种池很容易，然而当池为空时，你真正需要做的是**阻塞**它，然后在它不空时，再次解除阻塞。如果你以池的大小初始化一个 Semaphore，在你从池中获取资源之前，你应该调用 acquire 方法获取一个许可，调用 release 把许可放回资源池。acquire 会一直阻塞，直到池不再为空。这个技术用在第 12 章的有界缓冲类中。（一个创建阻塞对象池最简单的方法是使用 BlockingQueue 来保存资源池的资源。）

相似地，你可以使用 Semaphore 把任何容器转化为有界的阻塞容器，就像清单 5.14 中 BoundedHashSet 所示的一样。信号量被初始化为容器所期望容量的最大值。add 操作在向底层容器中添加条目之前，需要先获取一个许可。事实上，如果 add 操作没有能加入任何东西，它会立刻释放一个许可。同样，一个成功的 remove 操作释放一个许可，使更多的元素能够加入其中。底层的 Set 实现并不知道边界在哪里；这是由 BoundedHashSet 控制的。

5.5.4 关卡

我们已经看到闭锁究竟如何帮助启动一组相关的活动，或者等待一组相关的活动结束。闭锁是一次性使用的对象；一旦进入到最终状态，就不能被重置了。

关卡（barrier）类似于闭锁，它们都能够阻塞一组线程，直到某些事件发生[CPJ 4.4.3]。其中关卡与闭锁关键的不同在于，所有线程必须**同时**到达关卡点，才能继续处理。闭锁等待的是**事件**；关卡等待的是**其他线程**。关卡实现的协议，就像一些家庭成员指定商场中的集合地点：“我们每个人 6:00 在麦当劳见，到了以后不见不散，之后我们再决定接下来做什么。”

CyclicBarrier 允许一个给定数量的成员多次集中在一个**关卡点**，这在并行迭代算法中非常有用，这个算法会把一个问题拆分成一系列相互独立的子问题。当线程到达关卡点时，调用 await，await 会被阻塞，直到**所有**线程都到达关卡点。如果所有线程都到达了关卡点，关卡就被成功地突破，这样所有线程都被释放，关卡会重置以备下一次使用。如果对 await 的调用超时，或者阻塞中的线程被中断，那么关卡就被认为是**失败**的，所有对 await 未完成的调用都通过 BrokenBarrierException 终止。如果成功地通过关卡，await 为每一个线程返回一个唯一的到达索引号，可以用它来“选举”产生一个领导，在下一次迭代中承担一些特殊工作。CyclicBarrier 也允许你向构造函数传递一个**关卡行为**（barrier

清单 5.14 使用信号量来约束容器

```
public class BoundedHashSet<T> {
    private final Set<T> set;
    private final Semaphore sem;
    public BoundedHashSet(int bound) {
        this.set = Collections.synchronizedSet(new HashSet<T>());
        sem = new Semaphore(bound);
    }

    public boolean add(T o) throws InterruptedException {
        sem.acquire();
        boolean wasAdded = false;
        try {
            wasAdded = set.add(o);
            return wasAdded;
        }

        finally {
            if (!wasAdded)
                sem.release();
        }
    }

    public boolean remove(Object o) {
        boolean wasRemoved = set.remove(o);
        if (wasRemoved)
            sem.release();
        return wasRemoved;
    }
}
```

action)；这是一个 Runnable，当成功通过关卡的时候，会（在一个子任务线程中）执行，但是在阻塞线程被释放之前是不能执行的。

关卡通常被用来模拟这种情况：一个步骤的计算可以并行完成，但是要求必须完成所有与一个步骤相关的工作后才能进入下一步。比如，在一个 n 体质点系统的模拟中，每一步都根据其他粒子的位置和属性，计算每一个粒子位置的更新。（译注：N-body Particle， n 体质点法，是指将系统内的 n 个质点当作模拟对象，每个质点所受的作用力来自其他 $n-1$ 个的质点，然后将作用力全部加起来，就是该质点所受的力。该技术通常用在宇宙学、航空学、生物科学等领域的模拟。）在每次更新之间等待关卡，能够确保所有第 k 步的更新都已经完成，可以进入第 $k+1$ 步。

清单 5.15 中，CellularAutomata 演示了使用关卡来计算一个细胞的自动化模拟，比如 Conway's 的生命游戏（Gardner, 1970）。当模拟处于并行状态时，它通常会不假思索地为每一个元素（在这个例子中就是细胞）分配独立的线程；这将带来太多的线程，协调线程所花费的开销会延缓计算。代替的作法是，把问题划分成不同的子部分，让每一个线程解决其中一个部分，之后再合并结果。CellularAutomata 把问题分成了与可用 CPU 数量相同的几部分，并把每个部分分配给一个线程⁵。在每一步中，工作线程都为自己部分的细胞计算一个新的值。当所有线程到达关卡时，关卡的操作是向数据模型提交一个新的值。在关卡的操作完成后，工作线程被释放进行下一步计算，这包括询问一个 isDone 方法来决定是否需要下一次迭代。

Exchanger 是关卡的另一种形式，它是一种两步关卡，在关卡点会交换数据 [CPI 3.4.3]。当两方进行的活动不对称时，Exchanger 是非常有用的，比如当一个线程向缓冲写入一个数据，这时另一个线程充当消费者使用这个数据；这些线程可以使用 Exchanger 进行会面，并用完整的缓冲与空缓冲进行交换。当两个线程通过 Exchanger 交换对象时，交换为双方的对象建立了一个安全的发布。

交换的时机取决于应用程序的响应需求。最简单的方案是当写入任务的缓冲写满时就发生交换，并且当清除任务的缓冲清空后也发生交换；这样做使交换的次数最少，但是如果新数据的到达率不可预测的话，处理一些数据会发生延迟。另一个方案是，缓冲满了就发生交换，但是当缓冲部分充满却已经存在了特定长的时间时，也会发生交换。

5.6 为计算结果建立高效、可伸缩的高速缓存

几乎每一个服务器应用程序都使用某种形式的高速缓存。复用已有的计算结果可以缩

⁵ 在计算问题中，如果不涉及 I/O 或者访问共享数据，Ncpu 或者 Ncpu+1 个线程产生最优吞吐量；更多的线程也不会有任何帮助，并可能在某种程度上降低性能，因为线程会竞争 CPU 和内存等资源。

清单 5.15 在一个细胞的自动系统中用 CyclicBarrier 协调计算

```
public class CellularAutomata {
    private final Board mainBoard;
    private final CyclicBarrier barrier;
    private final Worker[] workers;

    public CellularAutomata(Board board) {
        this.mainBoard = board;
        int count = Runtime.getRuntime().availableProcessors();
        this.barrier = new CyclicBarrier(count,
            new Runnable() {
                public void run() {
                    mainBoard.commitNewValues();
                }
            });

        this.workers = new Worker[count];
        for (int i = 0; i < count; i++)
            workers[i] = new Worker(mainBoard.getSubBoard(count, i));
    }

    private class Worker implements Runnable {
        private final Board board;
        public Worker(Board board) { this.board = board; }
        public void run() {
            while (!board.hasConverged()) {
                for (int x = 0; x < board.getMaxX(); x++)
                    for (int y = 0; y < board.getMaxY(); y++)
                        board.setNewValue(x, y, computeValue(x, y));
                try {
                    barrier.await();
                } catch (InterruptedException ex) {
                    return;
                } catch (BrokenBarrierException ex) {
                    return;
                }
            }
        }
    }

    public void start() {
        for (int i = 0; i < workers.length; i++)
            new Thread(workers[i]).start();
        mainBoard.waitForConvergence();
    }
}
```

短等待时间，提高吞吐量，代价是占用更多的内存。

正像是其他一些“重复发明的轮子”，缓存通常看起来再简单不过了。一个天真的高速缓存，即使能够改进在单线程下的性能，也不过是将一个性能瓶颈转化为一个可伸缩的瓶颈。在本节，我们将开发一个高效的、可伸缩的高速缓存，为一个昂贵的函数保存计算结果。让我们从最明显的方案开始——一个简单的 `HashMap`——之后着眼于它在并发方面的劣势，并且讨论如何解决。

清单 5.16 的 `Computable<A, V>` 接口描述了一个功能，输入类型是 `A`，输出结果的类型是 `V`。 `ExpensiveFunction` 实现了 `Computable`，需要花很长时间来计算结果，我们喜欢创建一个 `Computable` 包装器，使它记住之前的计算结果，并封装缓存步骤（这项技术被称为**备忘录** *memoization*。）

清单 5.16 尝试使用 `HashMap` 和同步来初始化缓存

```
public interface Computable<A, V> {
    V compute(A arg) throws InterruptedException;
}

public class ExpensiveFunction
    implements Computable<String, BigInteger> {
    public BigInteger compute(String arg) {
        // after deep thought...
        return new BigInteger(arg);
    }
}

public class Memoizer1<A, V> implements Computable<A, V> {
    @GuardedBy("this")
    private final Map<A, V> cache = new HashMap<A, V>();
    private final Computable<A, V> c;
    public Memoizer1(Computable<A, V> c) {
        this.c = c;
    }

    public synchronized V compute(A arg) throws InterruptedException {
        V result = cache.get(arg);
        if (result == null) {
            result = c.compute(arg);
            cache.put(arg, result);
        }
        return result;
    }
}
```



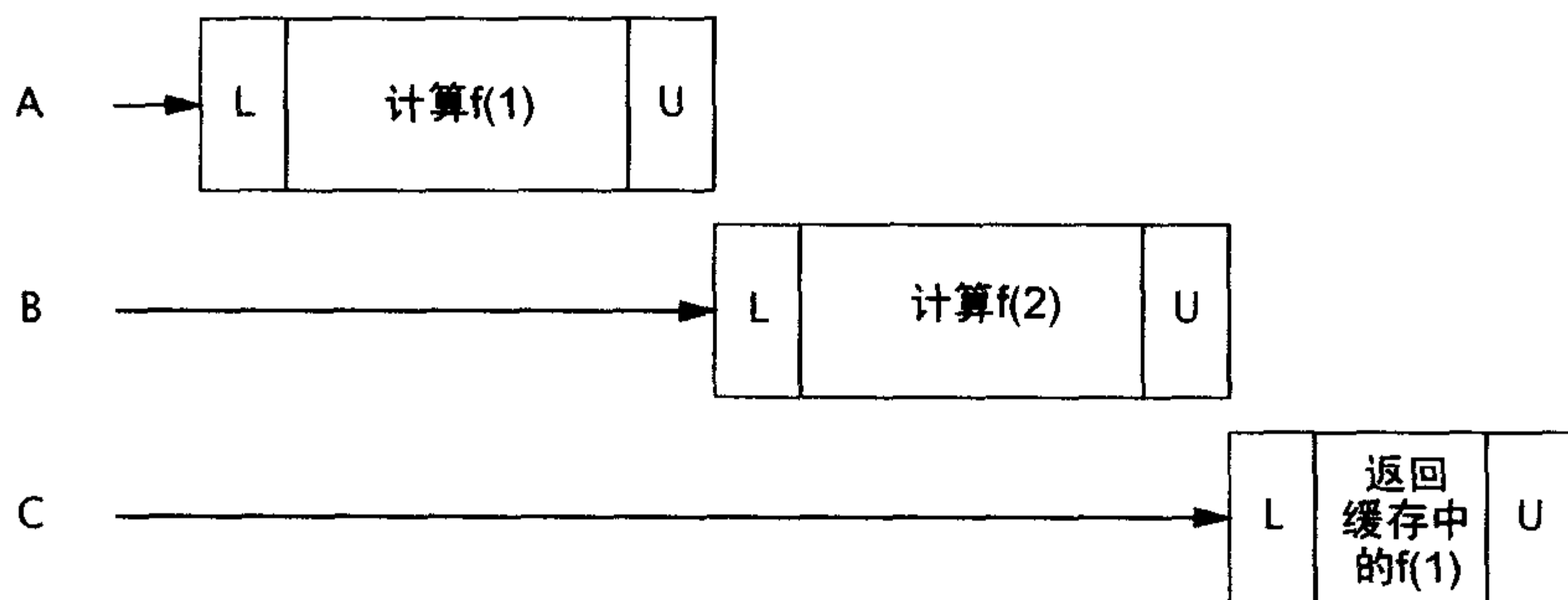


图 5.2 Memoizer1 的弱并发性

清单 5.16 中，Memoizer1 做了第一种尝试：使用 `HashMap` 存储前面的计算结果。`compute` 方法首先检查期待的结果是否已经在缓存中，如果有则返回之前计算的数值。否则，会进行计算并在返回之前将结果存储在 `HashMap` 中。

`HashMap` 不是线程安全的，所以，为了保证两个线程不会同时访问 `HashMap`，Memoizer1 采取了保守的方案，同步整个 `compute` 方法。这保证了线程安全，但是却带来一个明显的可伸缩性问题：一次只有一个线程能够执行 `compute`。如果另外一个线程正忙于计算结果，其他调用 `compute` 的线程可能被阻塞很长时间。如果有多个线程都在排队等待尚未计算出来的结果，那么事实上，`compute` 可能会比不使用备忘录形式花费更长的时间。图 5.2 解释了当几个线程尝试使用一个备忘录功能时会发生的情况。这不是我们希望通过缓存得到的性能优化结果。

清单 5.17 中，Memoizer2 用 `ConcurrentHashMap` 取代 `HashMap`，改进了 Memoizer1 中这种糟糕的并发行为。因为 `ConcurrentHashMap` 是线程安全的，所以不需要在访问底层 `Map` 时对它进行同步，这样就减少了在 Memoizer1 中同步 `compute` 带来的冗长代码。

Memoizer2 与 Memoizer1 相比，毫无疑问具有更好的并发性：多线程可以真正并发地使用它了。但是作为高速缓存仍然存在缺陷——当两个线程同时调用 `compute` 时，存在一个漏洞，会造成它们计算相同的值。在备忘录这种情况下，这仅仅是效率低而已——高速缓存的目的就是避免重复计算相同的数据。但是对于高速缓存机制更多元化的用途而言，这就不只是效率的问题了；一个缓存对象仅仅能够被初始化一次的话，这个漏洞就会带来安全风险。

Memoizer2 的问题在于，如果一个线程启动了一个开销很大的计算，其他线程并不知道这个计算正在进行中，所以可能又会重复这个计算，正像图 5.3 中表现的那样。我们希望无论用什么方法，能够表现出“线程 X 正在计算 $f(27)$ ”，这样如果另一个线程到达，

清单 5.17 用 ConcurrentHashMap 替换 HashMap

```

public class Memoizer2<A, V> implements Computable<A, V> {
    private final Map<A, V> cache = new ConcurrentHashMap<A, V>();
    private final Computable<A, V> c;

    public Memoizer2(Computable<A, V> c) { this.c = c; }

    public V compute(A arg) throws InterruptedException {
        V result = cache.get(arg);
        if (result == null) {
            result = c.compute(arg);
            cache.put(arg, result);
        }
        return result;
    }
}

```

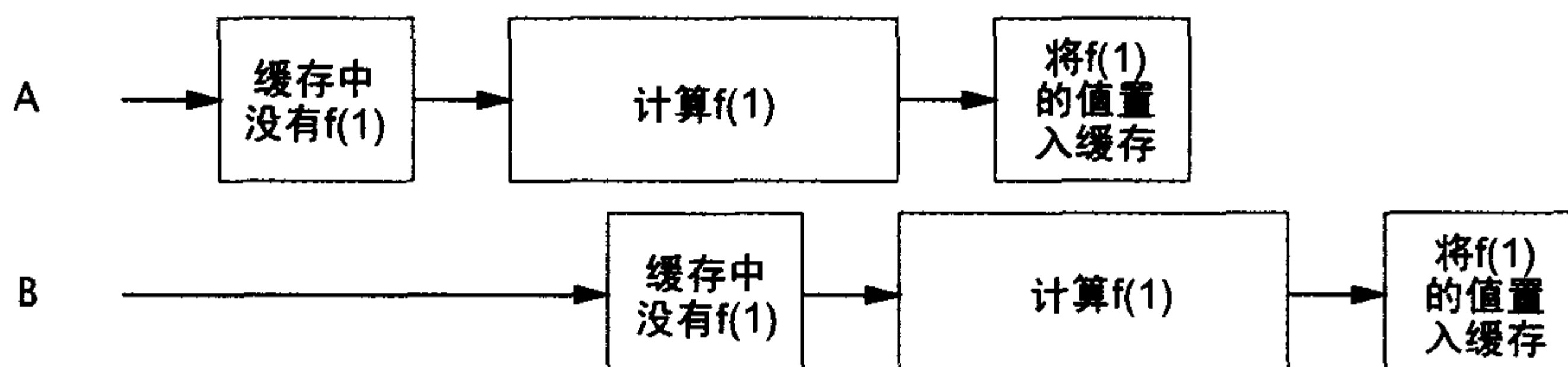


图 5.3 在使用 Memoizer2 时，两个线程计算相同值

并查找 $f(27)$ 时，它能够判断出最有效的方法是等待线程 X，直到线程结束，然后动动嘴唇“嘿， $f(27)$ 的结果是多少？”。

我们已经看到有一个类做的几乎就是这些事：FutureTask。FutureTask 代表了一个计算的过程，可能已经结束，也可能正在运行中。FutureTask.get 只要结果可用，就会立刻将结果返回；否则它会一直阻塞，直到结果被计算出来，并返回。

清单 5.18 中，Memoizer3 为缓存的值重新定义可存储 Map，用 ConcurrentHashMap<A, Future<V>> 取代了 ConcurrentHashMap<A, V>。Memoizer3 首先检查一个相应的计算是否已经开始，（Memoizer2 与它相反，它判断计算是否完成）。如果不是，就创建一个 FutureTask，把它注册到 Map 中，并开始计算；如果是，那么它会等待正在进行的计算。结果可能很快就会得到，或者正在运算的过程中——但是这对于调用者 Future.get 来说是透明的。

Memoizer3 的实现近乎是完美的：它展现了非常好的并发性（大部分来源于 ConcurrentHashMap 良好的并发性），能很快返回已经计算过的结果，如果新到的线程请求的是其他线程正在计算的结果，它会耐心地等待。它只存在一个缺陷——两个线程可能同

清单 5.18 用 FutureTask 记录包装器

```

public class Memoizer3<A, V> implements Computable<A, V> {
    private final Map<A, Future<V>> cache
        = new ConcurrentHashMap<A, Future<V>>();
    private final Computable<A, V> c;
    public Memoizer3(Computable<A, V> c) { this.c = c; }
    public V compute(final A arg) throws InterruptedException {
        Future<V> f = cache.get(arg);
        if (f == null) {
            Callable<V> eval = new Callable<V>() {
                public V call() throws InterruptedException {
                    return c.compute(arg);
                }
            };
            FutureTask<V> ft = new FutureTask<V>(eval);
            f = ft;
            cache.put(arg, ft);
            ft.run(); // 调用 c.compute 发生在这里
        }
        try {
            return f.get();
        } catch (ExecutionException e) {
            throw launderThrowable(e.getCause());
        }
    }
}

```



时计算相同的值，它仅仅存在这一个漏洞。这个漏洞远不如 Memoizer2 的严重，仅仅因为 compute 中的 if 代码块是非原子（nonatomic）的“检查再运行”序列，仍然存在这种可能：两个线程几乎在同一时间调用 compute 计算相同的值，双方都没有在缓存中找到期望的值，并都开始计算。这个特殊的时序在图 5.4 中进行了说明。

Memoizer3 的这个问题之所以会是一个漏洞，因为复合操作（缺少即加入）运行在底层的 map 中，不能加锁来使它原子化。清单 5.19 中的 Memoizer 利用了 ConcurrentMap 中原子化的 putIfAbsent 方法，消除了 Memoizer3 的隐患。

缓存一个 Future 而不是一个值，带来了**缓存污染**（cache pollution）的可能性：如果一个计算被取消或者失败，未来尝试对这个值进行计算都会表现为取消或者失败。为了避免这个结果，Memoizer 如果发现计算被取消，就会把 Future 从缓存中移除；如果发现有 RuntimeException，也会移除 Future，这样新请求中的计算才有可能成功。Memoizer

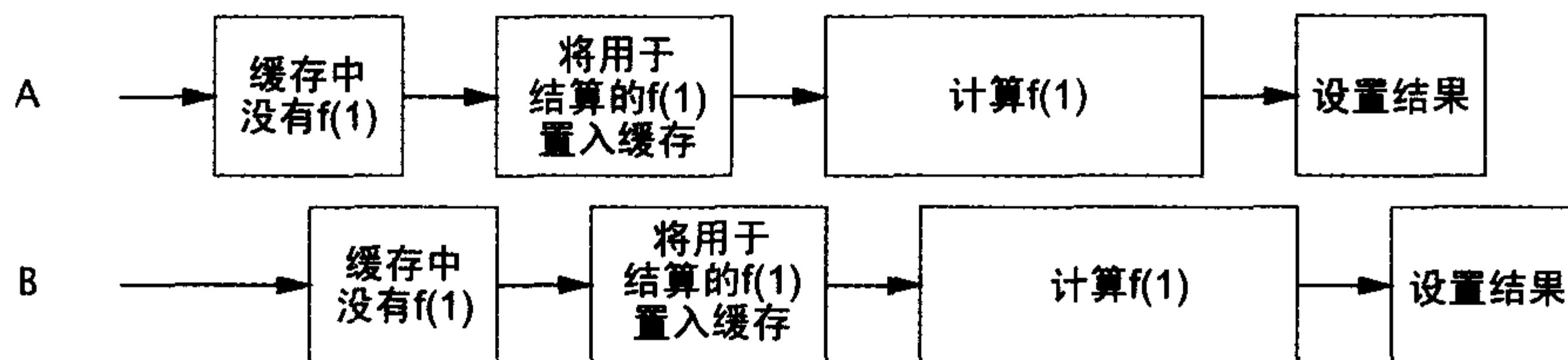


图 5.4 偶发的时序导致 Memoizer3 两次计算相同值

同样有缓存过期的问题，但是这些可以通过 `FutureTask` 的一个子类来完成，它会为每一个结果关联一个过期时间，并周期性地扫描缓存中过期的访问。（相似地，它也不能解决缓存清理不能解决的问题，把旧的计算移除，给新的腾出空间，这样缓存就不会占用多少内存空间了。）

我们并发缓存的实现已经完成，我们现在可以真正把第 2 章中 `servlet` 因式分解的结果缓存起来。清单 5.20 中的 `Factorizer` 使用 `Memoizer` 高效地、可扩展地、缓存之前的计算结果。

清单 5.19 Memoizer 最终实现

```
public class Memoizer<A, V> implements Computable<A, V> {
    private final ConcurrentMap<A, Future<V>> cache
        = new ConcurrentHashMap<A, Future<V>>();
    private final Computable<A, V> c;

    public Memoizer(Computable<A, V> c) { this.c = c; }

    public V compute(final A arg) throws InterruptedException {
        while (true) {
            Future<V> f = cache.get(arg);
            if (f == null) {
                Callable<V> eval = new Callable<V>() {
                    public V call() throws InterruptedException {
                        return c.compute(arg);
                    }
                };
                FutureTask<V> ft = new FutureTask<V>(eval);
                f = cache.putIfAbsent(arg, ft);
                if (f == null) { f = ft; ft.run(); }
            }
            try {
                return f.get();
            } catch (CancellationException e) {
                cache.remove(arg, f);
            } catch (ExecutionException e) {
                throw launderThrowable(e.getCause());
            }
        }
    }
}
```

清单 5.20 使用 Memoizer 为因式分解的 servlet 缓存结果

```
@ThreadSafe
public class Factorizer implements Servlet {
    private final Computable<BigInteger, BigInteger[]> c =
        new Computable<BigInteger, BigInteger[]>() {
            public BigInteger[] compute(BigInteger arg) {
                return factor(arg);
            }
        };

    private final Computable<BigInteger, BigInteger[]> cache
        = new Memoizer<BigInteger, BigInteger[]>(c);

    public void service(ServletRequest req,
                        ServletResponse resp) {
        try {
            BigInteger i = extractFromRequest(req);
            encodeIntoResponse(resp, cache.compute(i));
        } catch (InterruptedException e) {
            encodeError(resp, "factorization interrupted");
        }
    }
}
```

第一部分的总结

我们目前已经介绍了许多基础知识。下面这个“并发诀窍清单”总结了第一部分中的主要概念和规则。

- 可变状态，伙计们¹！
所有并发问题都归结为如何协调访问并发状态。可变状态越少，保证线程安全就越容易。
- 尽量将域声明为 `final` 类型，除非它们的需要是可变的。
- 不可变对象天生是线程安全的。
不可变对象极大地减轻了并发编程的压力。它们简单而且安全，可以在没有锁或者防御性复制的情况下自由地共享。
- 封装使管理复杂度变得更可行。
你固然可以用存储于全局变量的数据来写一个线程安全类。但是你为什么还要这样做？在对象中封装数据，让它们能够更加容易地保持不变；在对象中封装同步，使它能够更容易地遵守同步策略。
- 用锁来守护每一个可变变量。
- 对同一不变约束中的所有变量都使用相同的锁。
- 在运行复合操作期间持有锁。
- 在非同步的多线程情况下，访问可变变量的程序是存在隐患的。
- 不要依赖于可以需要同步的小聪明。
- 在设计过程中就考虑线程安全。或者在文档中明确地说明它不是线程安全的。
- 文档化你的同步策略。

¹ 1992年美国总统竞选时，竞选战略专家James Carville在Bill Clinton的行动总部里挂了一个牌子，上面写着“经济，伙计们”，来明确行动主旨。

PART

第 2 部分

构建并发应用程序

Structuring Concurrent Applications

任务执行

Task Execution

大多数并发应用程序是围绕执行**任务**（task）进行管理的。所谓**任务**就是抽象、离散的工作单元（unit of work）。把一个应用程序的工作（work）分离到任务中，可以简化程序的管理；这种分离还在不同事务间划分了自然的分界线，可以方便程序在出现错误时进行恢复；同时这种分离还可以为并行工作提供一个自然的结构，有利于提高程序的并发性。

6.1 在线程中执行任务

围绕执行任务来管理应用程序时，第一步要指明一个清晰的**任务边界**（task boundaries）。理想情况下，任务是**独立**的活动：它的工作并不依赖于其他任务的状态、结果或者边界效应（side effect）。独立有利于并发性，如果能得到相应的处理器资源，独立的任务还可以并行执行。为了使调度与负载均衡并具有更好的灵活性，每项任务只占用处理器的一小部分资源。

在正常的负载下，服务器应用程序应该兼具**良好的吞吐量**和**快速的响应性**。应用程序提供商希望程序支持尽可能多的用户，所以会努力降低每个用户的开销；而用户希望尽快获得响应。进一步讲，应用程序应该在负荷过载时**平缓地劣化**，而不应该负载一高就简单地以失败告终。为了达到这些目的，你要选择一个清晰的任务边界，并配合一个明确的**任务执行策略**（参见 6.6.2 节）。

大多数服务器应用程序都选择了下面这个自然的任务边界：单独的客户请求。Web 服务器，邮件服务器，文件服务器，EJB 容器和数据库服务器，这些服务器都接受远程客户通过网络连接发送的请求。将独立的请求作为任务边界，可以让任务兼顾独立性和适当的大小。例如，向邮件服务器提交一个消息后产生的结果，并不会被其他正在同时处理的消息所影响；而且，通常服务器只须用其总能力的很小一部分就能处理单一的消息。

6.1.1 顺序地执行任务

应用程序内部的任务调度，存在多种可能的调度策略，这些策略可以在不同程度上发挥出潜在的并发性。其中最简单的策略是在单一的线程中顺序地执行任务。清单 6.1 的 `SingleThreadWebServer` 顺序地处理它的任务——接受达到 80 端口的 HTTP 请求。处理请求的细节在这里并不重要；我们感兴趣的是刻画不同调度策略的同步特性。

清单 6.1 顺序化的 Web Server

```
class SingleThreadWebServer {  
    public static void main(String[] args) throws IOException {  
        ServerSocket socket = new ServerSocket(80);  
        while (true) {  
            Socket connection = socket.accept();  
            handleRequest(connection);  
        }  
    }  
}
```



`SingleThreadedWebServer` 很简单，它在理论上是正确的，但是它一次只能处理一个请求，因此在生产环境中的执行效率却很糟糕。主线程不断地在“接受连接”与“处理相关请求”之间交替运行，并且直到主线程完成了当前的请求并再次调用 `accept`，此前新的请求都必须等待。如果请求的处理速度很快，`handleRequest` 可以立即返回，那么这种方法未尝不可。但是这个例子描述的不是任何现实世界中的 Web Server。

一个 Web 请求的处理包括执行运算与进行 I/O 操作。服务器必须处理 Socket I/O，以读取请求和写回响应，网络拥堵或连通性问题会导致这个操作阻塞。服务器还要处理文件 I/O、发送数据库请求，这些同样会引起操作的阻塞。对于一个单线程化的服务器，阻塞不仅仅延迟了当前请求的完成，而且还完全阻止了需要被处理的等待请求。如果请求阻塞的时间过长，用户将看不到响应，可能认为服务器已经不可用了。同时，单线程在等待它的 I/O 操作时，CPU 会处于闲置状态，因此导致了资源利用率非常低。

顺序化处理几乎不能为服务器应用程序提供良好的吞吐量或快速的响应性。不过也有少数的特例——比如，当任务的数量很少但生命周期很长时，或者当服务器只服务于唯一的用户时，服务器在同一时间内只需同时处理一个请求——但是大多数服务器应用程序都不是以这种方式工作的¹。

¹ 在某些情况下，顺序化处理在简单性或者安全性上具有优势；大多数 GUI 框架使用单一的线程，并顺序地处理任务。我们会在第 9 章再次讨论顺序化模型。

6.1.2 显式地为任务创建线程

为了提供更好的响应性，可以为每个服务请求创建一个新的线程。正如清单 6.2 的 ThreadPerTaskWebServer 所示。

清单 6.2 Web Server 为每个请求启动一个新的线程

```
class ThreadPerTaskWebServer {  
    public static void main(String[] args) throws IOException {  
        ServerSocket socket = new ServerSocket(80);  
        while (true) {  
            final Socket connection = socket.accept();  
            Runnable task = new Runnable() {  
                public void run() {  
                    handleRequest(connection);  
                }  
            };  
            new Thread(task).start();  
        }  
    }  
}
```



ThreadPerTaskWebServer 在结构上类似于单线程版本——主线程仍然不断地交替运行“接受外部连接”与“转发请求”。不同点在于，主循环为每个连接都创建一个新线程以处理请求，而不是在主循环的内部处理它。由此得出下面 3 条主要结论：

- 执行任务的负载已经脱离了主线程，这让主循环能够更迅速地重新开始等待下一个连接。这使得程序可以在完成前面的请求之前接受新的请求，从而提高了响应性。
- 并行处理任务，这使得多个请求可以同时得到服务。如果有多个处理器，或者出于 I/O 未完成、锁请求以及资源可用性等任何因素需要阻塞任务时，程序的吞吐量会得到提高。
- 任务处理代码必须是线程安全的，因为有多多个任务会并发地调用它。

在中等强度的负载水平下，“每任务每线程（thread-per-task）”方法是对顺序化执行的良好改进。只要请求的到达速度尚未超出服务器的请求处理能力，那么这种方法可以同时带来更快的响应性和更大的吞吐量。

6.1.3 无限制创建线程的缺点

当用于生产环境中时，“每任务每线程（thread-per-task）”方法存在一些实际的缺陷，尤其在需要创建大量的线程时会更加突出：

线程生命周期的开销。线程的创建与关闭不是“免费”的。实际的开销依据不同平台而不同，但是创建线程的确需要时间，带来处理请求的延迟，并且需要在 JVM 和操作系统之间进行相应的处理活动。如果请求是频繁的且轻量的，就像大多数服务器程序一样，那么为每个请求创建一个新线程的做法就会消耗大量的计算资源。

资源消耗量。活动线程会消耗系统资源，尤其是内存。如果可运行的线程数多于可用的处理器数，线程将会空闲。大量空闲线程占用更多内存，给垃圾回收器带来压力，而且大量线程在竞争 CPU 资源时，还会产生其他的性能开销。如果你已经有了足够多的线程保持所有 CPU 忙碌，那么再创建更多的线程是有百害而无一利的。

稳定性。应该限制可创建线程的数目。限制的数目依不同平台而定，同时也受到 JVM 的启动参数、Thread 的构造函数中请求的栈大小等因素的影响，以及底层操作系统线程的限制²。如果你打破了这些限制，最可能的结果是收到一个 OutOfMemoryError。企图从这种错误中恢复是非常危险的；更简单的办法是构造你的程序时避免超出这些限制。

在一定范围内，增加线程可以提高系统的吞吐量，一旦超出了这个范围，再创建更多的线程只会拖垮你的程序。而一味地创建一个线程，会导致应用程序面临崩溃。为了摆脱这种危险，应该设置一个范围来限制你的应用程序可以创建的线程数，然后彻底地测试你的应用程序，确保即使线程数达到了这个范围的极限，程序也不至于耗尽所有的资源。

“每任务每线程（thread-per-task）”方法的问题在于它没有对已创建线程的数量进行任何限制，除非对客户端能够抛出的请求速率进行限制。像其他的并发危险一样，无限制创建线程的行为可能在原型和开发阶段还能表现得运行良好，而当应用程序部署后，并运行于高负载下，它的问题才会暴露出来。所以一个恶意用户或者足够多的用户，都会使你的 Web Server 的负载超过某个确定的极限值，从而导致服务器的崩溃。对于一个服务器，我们希望它具有高可用性，而且在高负载下可以平缓地劣化，但是上面的问题对我们的目标是个严重的阻碍。

² 在32位的机器上，主要的限制因素是线程栈的地址空间。每个线程都维护着两个执行栈（execution stack），一个用于Java代码，另一个用于原生代码。典型的JVM默认会产生一个组合的栈，大小在半兆字节左右。（你可以通过-Xss JVM参数或者通过Thread的构造函数修改这个值。）如果你为每个线程分配了大小是232字节的栈，那么你的线程数量将被限制在几千到几万间不等。其他方面，比如OS的限制，可能产生更加严格的约束。

6.2 Executor 框架

任务是逻辑上的工作单元，线程是使任务异步执行的机制。我们已经分析了两种线程执行任务的策略——所有任务在单一的线程中顺序执行，以及每个任务在自己的线程中执行。每一种策略都有严重的局限性：顺序执行会产生糟糕的响应性和吞吐量，“每任务每线程”会给资源管理带来麻烦。

在第 5 章我们看到，如何使用**有界队列**防止应用程序过载而耗尽内存。**线程池**（Thread pool）为线程管理提供了同样的好处。作为 Executor 框架的一部分，`java.util.concurrent` 提供了一个灵活的线程池实现。在 Java 类库中，任务执行的首要抽象不是 Thread，而是 Executor，如清单 6.3 中所示。

清单 6.3 Executor 接口

```
public interface Executor {  
    void execute(Runnable command);  
}
```

Executor 只是个简单的接口，但它却为一个灵活而且强大的框架创造了基础，这个框架可以用于异步任务执行，而且支持很多不同类型的任务执行策略。它还为**任务提交**和**任务执行**之间的解耦提供了标准的方法，为使用 Runnable 描述任务提供了通用的方式。Executor 的实现还提供了对生命周期的支持以及钩子函数，可以添加诸如统计收集、应用程序管理机制和监视器等扩展。

Executor 基于生产者-消费者模式。提交任务的执行者是生产者（产生待完成的工作单元），执行任务的线程是消费者（消耗掉这些工作单元）。**如果要在你的程序中实现一个生产者-消费者的设计，使用 Executor 通常是最简单的方式。**

6.2.1 示例：使用 Executor 实现的 Web Server

利用 Executor 可以很容易地构建 Web Server。清单 6.4 的 TaskExecutionWebServer 使用 Executor 代替了硬编码创建的线程。在这个例子中，我们用到了 Executor 标准实现之一，一个定长的线程池，可以容纳 100 个线程。

在 TaskExecutionWebServer 中，我们通过使用 Executor，将处理请求任务的提交与它的执行体进行了解耦。只要替换一个不同的 Executor 实现，就可以改变服务器的行为。改变 Executor 的实现或者配置，所产生的影响远远小于直接改变任务的执行方式；通常，Executor 的配置是一次性的事件，因此可以简单地在部署阶段完成，然而用于提交任务的代码却会不断地扩散到整个程序中，难以集中管理。

清单 6.4 使用线程池的 Web server

```
class TaskExecutionWebServer {
    private static final int NTHREADS = 100;
    private static final Executor exec
        = Executors.newFixedThreadPool(NTHREADS);

    public static void main(String[] args) throws IOException {
        ServerSocket socket = new ServerSocket(80);
        while (true) {
            final Socket connection = socket.accept();
            Runnable task = new Runnable() {
                public void run() {
                    handleRequest(connection);
                }
            };
            exec.execute(task);
        }
    }
}
```

只要作些简单的修改，就可以让 TaskExecutionWebServer 像 ThreadPerTaskWeb-Server 一样运行：替换一个 Executor，它为每个请求都创建一个新线程。编写这样一个 Executor 相当简单，正如清单 6.5 的 ThreadPerTaskExecutor 所示。

清单 6.5 为每个任务启动一个新线程的 Executor

```
public class ThreadPerTaskExecutor implements Executor {
    public void execute(Runnable r) {
        new Thread(r).start();
    }
}
```

类似地，编写下面这个 Executor 也很容易：让 TaskExecutionWebServer 的行为是单线程化的，所有任务都同步地执行，在前一个任务从 execute 返回后再执行下一个，正如清单 6.6 的 WithinThreadExecutor 所示。

6.2.2 执行策略

将任务的提交与任务的执行体进行解耦，它的价值在于让你可以简单地为一个类给定的任务制定**执行策略**，并且保证后续的修改不至于太困难。一个执行策略指明了任务执行的“what, where, when, how”几个因素，具体包括：

清单 6.6 Executor 在调用线程中同步地执行所有任务

```
public class WithinThreadExecutor implements Executor {  
    public void execute(Runnable r) {  
        r.run();  
    };  
}
```

- 任务在什么 (what) 线程中执行?
- 任务以什么 (what) 顺序执行 (FIFO, LIFO, 优先级)?
- 可以有多少个 (how many) 任务并发执行?
- 可以有多少个 (how many) 任务进入等待执行队列?
- 如果系统过载, 需要放弃一个任务, 应该挑选哪一个 (which) 任务? 另外, 如何 (how) 通知应用程序知道这一切呢?
- 在一个任务的执行前与结束后, 应该做什么 (what) 处理?

执行策略是资源管理工具。最佳策略取决于可用的计算资源和你对服务质量 (quality-of-service) 的需求。通过限制并发任务的数量, 你能够确保应用程序不会由于资源耗尽而失败, 大量任务也不会在争夺稀缺资源时出现性能问题³。将任务的提交与任务的执行策略规则进行分离, 有助于在部署阶段选择一个与当前硬件最匹配的执行策略。

无论何时当你看到这种形式的代码:

```
new Thread(runnable).start()
```

并且你可能最终希望获得一个更加灵活的执行策略时, 请认真考虑使用 Executor 代替 Thread。

6.2.3 线程池

正如名称中所称的那样, 线程池管理一个工作者线程的同构池 (homogeneous pool)。线程池是与工作队列 (work queue) 紧密绑定的。所谓工作队列, 其作用是持有所有等待执行的任务。工作者线程的生活从此轻松起来: 它从工作队列中获取下一个任务, 执行它, 然后回来继续等待另一个线程。

³ 这类似于企业应用程序中事务监视器 (transaction monitor) 的角色: 它将可运行事务的数量控制在一个合理的水平中, 不会因过渡滥用事务而耗尽有限资源。

在线程池中执行任务线程，这种方法有很多“每任务每线程”无法比拟的优势。重用存在的线程，而不是创建新的线程，这可以在处理多请求时抵消线程创建、消亡产生的开销。另一项额外的好处就是，在请求到达时，工作者线程通常已经存在，用于创建线程的等待时间并不会延迟任务的执行，因此提高了响应性。通过适当地调整线程池的大小，你可以得到足够多的线程以保持处理器忙碌，同时还可以防止过多的线程相互竞争资源，导致应用程序耗尽内存或者失败。

类库提供了一个灵活的线程池实现和一些有用的预设配置。你可以通过调用 `Executors` 中的某个静态工厂方法来创建一个线程池：

`newFixedThreadPool` 创建一个定长的线程池，每当提交一个任务就创建一个线程，直到达到池的最大长度，这时线程池会保持长度不再变化（如果一个线程由于非预期的 `Exception` 而结束，线程池会补充一个新的线程）。

`newCachedThreadPool` 创建一个可缓存的线程池，如果当前线程池的长度超过了处理的需要时，它可以灵活地回收空闲的线程，当需求增加时，它可以灵活地添加新的线程，而并不会对池的长度作任何限制。

`newSingleThreadExecutor` 创建一个单线程化的 `executor`，它只创建唯一的工作者线程来执行任务，如果这个线程异常结束，会有另一个取代它。`executor` 会保证任务依照任务队列所规定的顺序（FIFO，LIFO，优先级）执行⁴。

`newScheduledThreadPool` 创建一个定长的线程池，而且支持定时的以及周期性的任务执行，类似于 `Timer`（参见 6.2.5 节）。

`newFixedThreadPool` 和 `newCachedThreadPool` 两个工厂方法返回通用目的的 `ThreadPoolExecutor` 实例。直接使用 `ThreadPoolExecutor`，也能创建更满足某些专有领域的 `executor`。我们将在第 8 章深入讨论线程池的配置选项。

`TaskExecutionWebServer` 中的 `Web Server` 使用 `Executor` 创建了一个有界的线程池。通过 `execute` 方法把任务提交到工作队列中，工作者线程周而复始地从工作队列中弹出任务并执行它们。

从“每任务每线程”策略迁移到基于池的策略，会对应用程序的稳定性产生重大的影响：`Web Server` 再也不会因过高的负载失败了⁵。

⁴ 单线程化的 `executor` 同样作了大量的内部同步，以保证任何任务的写内存操作对后续线程是可见的；这意味着对象可以被安全地限制在“任务线程”中，尽管这个线程是不断交替更换的。

⁵ 尽管服务器不会因为创建了过多的线程而失败，但是如果在足够长的时间内，任务到达的速度总是超过任务执行的速度，服务器仍然可能（只是更不易）耗尽内存，因为等待执行的 `Runnable` 队列会不断地增长。使用一个有限的工作队列，可以在 `Executor` 框架内部解决这个问题——参见 8.3.2 节。

由于服务器没有创建数以千计的线程去争夺有限的 CPU 和内存资源，所以服务器仍然会更加平缓地劣化。使用 Executor 为你打开了一扇通往无限可能的大门，你有各种机会去做调优、管理、监视、记录日志、错误报告和其他可能的事情，如果不使用任务执行框架的话，添加这些特性是非常困难的。

6.2.4 Executor 的生命周期

我们已经看过如何创建一个 Executor 了，现在来关注如何关闭它。Executor 实现通常只是为执行任务而创建线程。但是 JVM 会在所有（非后台的，nondaemon）线程全部终止后才退出。因此，如果无法正确关闭 Executor，将会阻止 JVM 的结束。

因为 Executor 是异步地执行任务，所以在任何时间里，所有之前提交的任务的状态都不能立即可见。这些任务中，有些可能已经完成，有些可能正在运行，其他的还可能在队列中等待执行。关闭应用程序时，程序会出现很多种情况：从最平缓的关闭（已经启动的任务全部完成而且没有再接到任何新的工作）到最唐突的关闭（拔掉机房的电源），以及介于这两种极端情况之间的各种可能。既然 Executor 是为应用程序提供服务的，它们理应可以关闭，无论是平缓地还是唐突地。另外，关闭操作还会影响到记录应用程序任务状态的反馈信息。

为了解决这个执行服务的生命周期问题，ExecutorService 接口扩展了 Executor，并且添加了一些用于生命周期管理的方法（同时还有一些用于任务提交的便利方法）。清单 6.7 演示了 ExecutorService 中用于生命周期管理的方法。

清单 6.7 ExecutorService 中的生命周期方法

```
public interface ExecutorService extends Executor {
    void shutdown();
    List<Runnable> shutdownNow();
    boolean isShutdown();
    boolean isTerminated();
    boolean awaitTermination(long timeout, TimeUnit unit)
        throws InterruptedException;
    // ... 其他用于任务提交的便利方法
}
```

ExecutorService 暗示了生命周期有 3 种状态：**运行**（running）、**关闭**（shutting down）和**终止**（terminated）。ExecutorService 最初创建后的初始状态是**运行**状态。shutdown 方法会启动一个平缓的关闭过程：停止接受新的任务，同时等待已经提交的任务完成——包括尚未开始执行的任务。shutdownNow 方法会启动一个强制的关闭过程：尝试取消所有运行中的任务和排在队列中尚未开始的任务。

在关闭后提交到 ExecutorService 中的任务，会被**拒绝执行处理器**（rejected execution handler）处理（参见 8.3.3 节）。**拒绝执行处理器**（拒绝执行处理器是 ExecutorService 的一种实现，ThreadPoolExecutor 提供的，ExecutorService 接口

中的方法并不提供拒绝执行处理器。)可能只是简单地放弃任务,也可能引起 `execute` 抛出一个未检查的 `RejectedExecutionException`。一旦所有的任务全部完成后, `ExecutorService` 会转入 **终止状态**。你可以调用 `awaitTermination` 等待 `ExecutorService` 到达终止状态,也可以轮询检查 `isTerminated` 判断 `ExecutorService` 是否已经终止。通常 `shutdown` 会紧随 `awaitTermination` 之后,这样可以产生同步地关闭 `ExecutorService` 的效果。(第7章涵盖了关于关闭 `Executor` 和取消任务的更多细节。)

清单 6.8 的 `LifecycleWebServer` 扩展自一个提供了生命周期支持的 `Web Server`。关闭这个 `Web Server` 有两种方法:通过在程序中调用 `stop`, 或者经由客户端请求向 `Web Server` 发送一个特定格式的 `HTTP` 请求。

清单 6.8 支持关闭操作的 Web Server

```
class LifecycleWebServer {
    private final ExecutorService exec = ...;

    public void start() throws IOException {
        ServerSocket socket = new ServerSocket(80);
        while (!exec.isShutdown()) {
            try {
                final Socket conn = socket.accept();
                exec.execute(new Runnable() {
                    public void run() { handleRequest(conn); }
                });
            } catch (RejectedExecutionException e) {
                if (!exec.isShutdown())
                    log("task submission rejected", e);
            }
        }
    }

    public void stop() { exec.shutdown(); }

    void handleRequest(Socket connection) {
        Request req = readRequest(connection);
        if (isShutdownRequest(req))
            stop();
        else
            dispatchRequest(req);
    }
}
```


6.2.5 延迟的、并具周期性的任务

Timer 工具管理任务的延迟执行（“100ms 后执行该任务”）以及周期执行（“每 10ms 执行一次该任务”）。但是，Timer 存在一些缺陷，因此你应该考虑使用 ScheduledThreadPoolExecutor 作为代替品⁶。你可以通过构造函数或者通过 newScheduledThreadPool 工厂方法，创建一个 ScheduledThreadPoolExecutor。

Timer 只创建唯一的线程来执行所有 timer 任务。如果一个 timer 任务的执行很耗时，会导致其他 TimerTask 的时效准确性出问题。例如一个 TimerTask 每 10ms 执行一次，而另一个 TimerTask 每 40ms 执行一次，重复出现的任务要么会在耗时的任务完成后快速连续地被调用 4 次，要么完全“丢失”4 次调用（取决于它是否按照固定的频率或延迟进行调度）。调度线程池（Scheduled thread pool）解决了这个缺陷，它让你可以提供多个线程来执行延迟、并具周期性的任务。

Timer 的另一个问题在于，如果 TimerTask 抛出未检查的异常，Timer 将会产生无法预料的行为。Timer 线程并不捕获异常，所以 TimerTask 抛出的未检查的异常会终止 timer 线程。这种情况下，Timer 也不会再重新恢复线程的执行了；它错误地认为整个 Timer 都被取消了。此时，已经被安排但尚未执行的 TimerTask 永远不会再执行了，新的任务也不能被调度了。（这个问题叫做“线程泄漏”，7.3 节中会讲述它，同时还会讲述用于避免这个问题的技术。）

清单 6.9 的 OutOfTime 展示了 Timer 是如何出现问题的，常言道祸不单行，Timer 还将它的问题传染给下一个倒霉的调用者，这个调用者原本试图提交一个 TimerTask 的。你可能希望程序会运行 6 秒钟然后退出，然而实际情况是程序 1 秒钟后就中止了，还伴随着一个异常，异常的消息是“Timer already cancelled”。ScheduledThreadPoolExecutor 妥善地处理了这个行为异常的任务；在 Java 5.0 或更高的 JDK 中，几乎没有理由再使用 Timer 了。

如果你需要构建自己的调度服务，仍然可以使用类库提供的 DelayQueue，它是 BlockingQueue 的实现，为 ScheduledThreadPoolExecutor 提供了调度功能。DelayQueue 管理着一个包含 Delayed 对象的容器。每个 Delayed 对象都与一个延迟时间相关联：只有在元素过期后，DelayQueue 才让你能执行 take 操作获取元素。从 DelayQueue 中返回的对象将依据它们所延迟的时间进行排序。

6.3 寻找可强化的并行性

Executor 框架让制定一个执行策略变得简单。不过想要使用 Executor，你还必须能够将你的任务描述为 Runnable。在大多数服务器应用程序中，都存在这样一个明显的任

⁶ Timer 对调度的支持是基于绝对时间，而不是相对时间的，由此任务对系统时钟的改变是敏感的；ScheduledThreadPoolExecutor 只支持相对时间。

清单 6.9 Timer 的混乱行为

```
public class OutOfTime {  
    public static void main(String[] args) throws Exception {  
        Timer timer = new Timer();  
        timer.schedule(new ThrowTask(), 1);  
        SECONDS.sleep(1);  
        timer.schedule(new ThrowTask(), 1);  
        SECONDS.sleep(5);  
    }  
  
    static class ThrowTask extends TimerTask {  
        public void run() { throw new RuntimeException(); }  
    }  
}
```



务边界：单一的客户请求。但是，正如很多桌面应用程序一样，合理的任务边界有时并非如此显而易见。即使就是服务器应用程序，一个单一的客户请求内部仍然会有可以进一步细化的并行性，正像数据库服务器遇到的情况。（如何在选择任务边界时权衡设计的要求，关于这个问题的更多讨论，参见[CPI 4.4.1.1]。）

在本节里，我们会为一个组件开发不同的版本，每个版本允许不同程度的并发性。我们的示例组件源自于浏览器程序中渲染页面（page-rendering）的那部分功能，它获得页面的 HTML，并将它渲染到图像缓存里。为了保持它的简单性，我们假设 HTML 页面只包含标签文本，其中穿插预定义了尺寸和 URL 的图片。

6.3.1 示例：顺序执行的页面渲染器

处理 HTML 文档最简单的方法是顺序处理。当遇到一个文本标签，就将它渲染到图像缓存里；当遇到一个图像的引用时，先通过网络获取它，然后也将它渲染到图像缓存里。这很容易实现，程序只需要与每个输入的元素打一次交道（甚至不需要缓存文档）。但是这可能会令用户感到烦恼，他们必须一直等待很长时间，直到呈现出所有的文本。

另一种同样是顺序执行的方法会稍微好一些，它先渲染文本元素，并为图像预留出矩形的占位符；在完成了第一趟处理文本后，程序返回到开始，并下载图像，将它们绘制到相应的占位符上。清单 6.10 的 SingleThreadRenderer 演示了这种方法。

下载图像总免不了等待 I/O 操作的完成，在这段时间里，CPU 几乎不做任何工作。因此这种顺序执行的方法可能没有充分利用 CPU，并且用户如果必须看到完整的文档的话，要等待更长的时间。通过将问题分散到独立的可以并发执行的任务中，我们能会获得更好的 CPU 利用率和响应性。

清单 6.10 顺序地渲染页面元素

```
public class SingleThreadRenderer {  
    void renderPage(CharSequence source) {  
        renderText(source);  
        List<ImageData> imageData = new ArrayList<ImageData>();  
        for (ImageInfo imageInfo : scanForImageInfo(source))  
            imageData.add(imageInfo.downloadImage());  
        for (ImageData data : imageData)  
            renderImage(data);  
    }  
}
```



6.3.2 可携带结果的任务：Callable 和 Future

Executor 框架使用 Runnable 作为其任务的基本表达形式。Runnable 只是个相当有限的抽象；虽然它能够产生一些边界效应，比如记录日志文件或者将结果存入一个共享的数据结构，但是 run 不能返回一个值或者抛出受检查的异常。

很多任务都会引起严重的计算延迟——执行数据库查询，从网络上获取资源，进行复杂的计算。对于这些任务，Callable 是更佳的抽象：它的主进入点（main entry point）——call——等待返回值，并为可能抛出的异常预先做好了准备⁷。Executors 包含了一些工具方法，可以把其他类型的任务封装成一个 Callable，比如 Runnable 和 java.security.PrivilegedAction。

Runnable 和 Callable 描述的是抽象的计算型任务。这些任务通常是有限的：它们有一个明确的开始点，而且最终会结束。一个 Executor 执行的任务的生命周期有 4 个阶段：**创建、提交、开始和完成**。由于任务的执行可能会花费很长时间，我们也希望可以取消一个任务。在 Executor 框架中，总可以取消已经提交但尚未开始的任務，但是对于已经开始的任务，只有它们响应中断，才可以取消。取消一个已经完成的任務没有影响（第 7 章涵盖了取消任务的更多细节）。

Future 描述了任务的生命周期，并提供了相关的方法来获得任务的结果、取消任务以及检验任务是否已经完成还是被取消。清单 6.11 展示了 Callable 和 Future。Future 的规约中暗示了任务的生命周期是单向的，不能后退——就像 ExecutorService 的生命周期一样。一旦任务完成，它就永远停留在完成状态上。

任务的状态（尚未开始，运行中，完成）决定了 get 方法的行为。如果任务已经完成，get 会立即返回或者抛出一个 Exception，如果任务没有完成，get 会阻塞直到它完成。如果任务抛出了异常，get 会将该异常封装为 ExecutionException，然后重新抛出；如

⁷ 可以使用 Callable<Void> 表达无返回值的任务。

清单 6.11 Callable interface 和 Future interface

```
public interface Callable<V> {
    V call() throws Exception;
}

public interface Future<V> {
    boolean cancel(boolean mayInterruptIfRunning);
    boolean isCancelled();
    boolean isDone();
    V get() throws InterruptedException, ExecutionException,
        CancellationException;
    V get(long timeout, TimeUnit unit)
        throws InterruptedException, ExecutionException,
        CancellationException, TimeoutException;
}
```

果任务被取消，get 会抛出 CancellationException。当抛出了 ExecutionException 时，可以用 getCause 重新获得被封装的原始异常。

有很多种方法可以创建一个描述任务的 Future。ExecutorService 中的所有 submit 方法都返回一个 Future，因此你可以将一个 Runnable 或一个 Callable 提交给 executor，然后得到一个 Future，用它来重新获得任务执行的结果，或者取消任务。你也可以显式地为给定的 Runnable 或 Callable 实例化一个 FutureTask。（FutureTask 实现了 Runnable，所以既可以将它提交给 Executor 来执行，又可以直接调用 run 方法运行。）

在 Java 6 中，ExecutorService 的所有实现都可以覆写 AbstractExecutorService 中的 newTaskFor 方法，以此控制 Future 的实例化，以及对应的已提交的 Runnable 或 Callable。默认的实现仅仅创建一个新的 FutureTask，正如清单 6.12 所示的。

清单 6.12 ThreadPoolExecutor 中 newTaskFor 的默认实现

```
protected <T> RunnableFuture<T> newTaskFor(Callable<T> task) {
    return new FutureTask<T>(task);
}
```

将 Runnable 或 Callable 提交到 Executor 的行为，可以建立一个安全发布（参见 3.5 节），以保证 Runnable 或 Callable 从提交线程暴露到最终执行任务的线程的过程是线程安全的。类似地，设置 Future 结果值的行为，也可以建立一个安全发布，以保证这个结果从计算它的线程暴露到通过 get 重获它的任何线程的过程是线程安全的。

6.3.3 示例：使用 Future 实现页面渲染器

为了使我们的页面渲染器具有更高的并发性，第一步是将渲染过程分成两个任务，一个是渲染所有的文本，一个是下载所有的图像。（因为一个任务是受限于 CPU 的，另一个则是受限于 I/O，所以即使在单 CPU 系统上，这种方法也能带来生产效率的提升。）

Callable 和 Future 可以帮助我们表述所有协同工作的任务之间的互交。在清单 6.13 的 FutureRenderer 中，我们创建了一个 Callable 来下载所有的图像，并将这个 Callable 提交到 ExecutorService。之后返回一个描述任务执行的 Future；当到达需要所有图像的时间点时，主任务会等待 Future.get 调用的结果。如果幸运的话，我们请求的当时所有图像已经下载完成；即使没有，至少图像的下载已经预先开始了。

“状态依赖性”是 get 的内在特性，它意味着调用者不必知晓任务的状态。任务提交和重获的结果的安全发布特性，确保了这个方法是线程安全的。包围 Future.get 的异常处理代码解决了两个可能出现的问题：任务遇到一个 Exception，或者调用 get 的线程在获得结果前被提前中断（参见 5.5.2 节和 5.4 节）。

FutureRenderer 允许渲染文本与下载图像数据并发地执行。当下载完所有的图像后，它们会被呈现到页面上。在让用户快速看到结果的这一方面，该方法是一种提高，同时该方法还加强了并行性。然而，我们还可以做得更好。用户不必等到**所有**的图像都下载完成；他们或许更希望只要下载完一幅图像，就把它绘制在页面上，让人们看到。

6.3.4 并行运行异类任务的局限性

在最后一个任务里，我们试图并行执行两个不同类型的任务——下载图像与渲染页面。但是，对于这种试图并行执行连续的异类任务（heterogeneous task），以获得性能重大提升的方法，还需要谨慎对待。

两个人可以有效公平地分担清洗晚餐餐具的工作：一个人清洗的同时，另一个人烘干。但是，很好地按照比例分配不同类型的任务是困难的；当有更多的人出现时，如何确保他们在帮忙的同时没有变得碍手碍脚的，或者没有导致大范围的劳动力重新分工，这并不是很容易办到的。如果没有在相似的任务之间发现更好的并行性，那么并行运行方法应有的好处会逐渐减少。

更为严重的问题是，在给多个工作者划分相异任务时，各个任务的大小可能完全不同。比如你为两个工作者划分了两个任务 A 和 B，但是 A 执行花费的时间是 B 的 10 倍，那么你的整个过程仅仅加速了 9% 而已。最后，在多个工作者之间划分任务，总会涉及到一些任务协调上的开销；为了使划分任务是值得的，这一开销不能多于通过并行性带来的生产力的提高。

FutureRenderer 用到了两个任务：一个负责渲染文本，一个负责下载图像。如果渲染文本的速度远远大于下载图像的（这是完全有可能的），那么最终的性能与顺序执行版

清单 6.13 使用 Future 等待图像下载

```
public class FutureRenderer {
    private final ExecutorService executor = ...;

    void renderPage(CharSequence source) {
        final List<ImageInfo> imageInfos = scanForImageInfo(source);
        Callable<List<ImageData>> task =
            new Callable<List<ImageData>>() {
                public List<ImageData> call() {
                    List<ImageData> result
                        = new ArrayList<ImageData>();
                    for (ImageInfo imageInfo : imageInfos)
                        result.add(imageInfo.downloadImage());
                    return result;
                }
            };

        Future<List<ImageData>> future = executor.submit(task);
        renderText(source);

        try {
            List<ImageData> imageData = future.get();
            for (ImageData data : imageData)
                renderImage(data);
        } catch (InterruptedException e) {
            // 重新声明线程的中断状态
            Thread.currentThread().interrupt();
            // 我们不需要结果，故取消任务。
            future.cancel(true);
        } catch (ExecutionException e) {
            throw launderThrowable(e.getCause());
        }
    }
}
```



本的性能不会有很大的不同，反倒是代码的复杂度大大提高了。我们使用两个线程可以得到最理想的结果是速度提高一倍。因此，试图通过并行执行异类活动以提高并发行，还需要完成很多工作，并且，你能从中获得的并发性是十分有限的。（参见 11.4.2 和 11.4.3，可以看到同一现象的另一个示例。）

大量相互独立且同类的任务进行并发处理，会将程序的任务量分配到不同的任务中，这样才能真正获得性能的提升。

6.3.5 CompletionService: 当 Executor 遇见 BlockingQueue

如果你向 Executor 提交了一个批处理任务，并且希望在它们完成后获得结果，为此你可以保存与每个任务相关联的 Future，然后不断地调用 timeout 为零的 get，来检验 Future 是否完成。这样做固然可以，但却相当乏味。幸运的是，还有一种更好的方法：完成服务（completion service）。

CompletionService 整合了 Executor 和 BlockingQueue 的功能。你可以将 Callable 任务提交给它去执行，然后使用类似于队列中的 take 和 poll 方法，在结果完整可用时获得这个结果，像一个打包的 Future。ExecutorCompletionService 是实现 CompletionService 接口的一个类，并将计算任务委托给一个 Executor。

ExecutorCompletionService 的实现是相当直观的。它在构造函数中创建一个 BlockingQueue，用它去保存完成的结果。计算完成时会调用 FutureTask 中 done 方法。当提交了一个任务后，首先把这个任务包装为一个 QueueingFuture，它是 FutureTask 的一个子类，然后覆写 done 方法，将结果置入 BlockingQueue 中，如同清单 6.14 所示。take 和 poll 方法委托给了 BlockingQueue，它会在结果不可用时阻塞。

清单 6.14 ExecutorCompletionService 使用的 QueueingFuture 类

```
private class QueueingFuture<V> extends FutureTask<V> {
    QueueingFuture(Callable<V> c) { super(c); }
    QueueingFuture(Runnable t, V r) { super(t, r); }

    protected void done() {
        completionQueue.add(this);
    }
}
```

6.3.6 示例：使用 CompletionService 的页面渲染器

使用 CompletionService，我们可以从两方面提高页面渲染器的性能：缩短总的运行时间以及提高响应性。我们可以**每需要下载一个图像**，就创建一个独立的任务，并在线程池中执行它们，将顺序的下载过程转换为并行的：这能减少下载**所有**图像的总时间。而且从 CompletionService 中获取结果，只要任何一个图像下载完成，就立刻呈现，由此我们可以给用户提供一个更加动态和有更高响应性的用户界面。清单 6.15 的 Renderer 演示了这个实现。

清单 6.15 使用 CompletionService 渲染可用的页面元素

```
public class Renderer {
    private final ExecutorService executor;

    Renderer(ExecutorService executor) { this.executor = executor; }

    void renderPage(CharSequence source) {
        final List<ImageInfo> info = scanForImageInfo(source);
        CompletionService<ImageData> completionService =
            new ExecutorCompletionService<ImageData>(executor);
        for (final ImageInfo imageInfo : info)
            completionService.submit(new Callable<ImageData>() {
                public ImageData call() {
                    return imageInfo.downloadImage();
                }
            });

        renderText(source);

        try {
            for (int t = 0, n = info.size(); t < n; t++) {
                Future<ImageData> f = completionService.take();
                ImageData imageData = f.get();
                renderImage(imageData);
            }
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        } catch (ExecutionException e) {
            throw launderThrowable(e.getCause());
        }
    }
}
```

多个 ExecutorCompletionServices 可以共享单一的 Executor，因此一个明智的做

法是创建一个 `ExecutorCompletionService`，它对于特定的计算服务是私有的，然后再共享一个公共的 `Executor`。按照这种做法，`CompletionService` 所扮演的批处理计算的句柄与 `Future` 所扮演的单一计算的句柄，在很大程度上是一样的。记录下提交给 `CompletionService` 的任务的个数，然后计算出获得了多少个已完成的结果，这样即使你使用的是共享的 `Executor`，你也能知晓什么时候批处理任务的所有结果已经全部获得。

6.3.7 为任务设置时限

有时候如果一个活动无法在某个确定时间内完成，那么它的结果就失效了，此时程序可以放弃该活动。举个例子，一个 `Web Application` 会从外部的广告服务器上获取广告信息，但是如果应用程序在两秒钟内得不到响应，就会显示一个默认的信息，这样即使得不到广告信息也不会破坏站点的响应性需求。类似地，一个门户网站可以从多个数据源并行地获取数据，但是可能会在限定时间内等待数据，到了时间就会只呈现现有的数据了。

在预定时间内执行任务的主要挑战是，你要确保在得到答案，或者发现无法从任务中获得结果的这一过程所花费的时间，不会比预定的时间更长。`Future.get` 的限时版本符合这个条件：它在结果准备好后立即返回，如果在时限内没有准备好，就会抛出 `TimeoutException`。

使用限时任务的第二个问题是，当它们超时后应该能够停止它们，这样才不会为继续计算一个无用的结果而浪费计算资源。为了达到这个目的，可以让任务自己严格管理它的预定时间，超时后就中止执行；或者也可以在超出时限后取消任务。`Future` 再次派上了用场；如果一个限时的 `get` 抛出 `TimeoutException`，你可以通过 `Future` 取消任务。如果你编写的任务是可取消的（参见第 7 章），就可以更灵敏地中止它，以避免消耗过多的资源。清单 6.13 和 6.16 的代码就使用了这项技术。

清单 6.16 演示了限时的 `Future.get` 的一种典型应用。它生成一个混合了不同内容的页面，包括响应用户请求的内容和从广告服务器上获得的广告内容。它将获取广告的任务提交给 `executor`，然后计算剩余的文本页面内容，最后等待广告信息，直到预定时间耗尽⁸。如果 `get` 超时，它会取消⁹广告获取的任务，并使用默认的信息取代它。

6.3.8 示例：旅游预定门户网站

上一节中谈到的“预定时间”方法可以简单地推广到任意数量的任务上。考虑这样一

⁸ 计算传递给 `get` 的 `timeout` 的方法是从当前时间中减去 `deadline`；这事实上会产生负数，不过 `java.util.concurrent` 中所有关于时间的方法都将负数作为零处理，因此不需要额外的代码处理这种情况。

⁹ `Future.cancel` 的参数 `true` 意味着任务线程可以在运行时被中断；参见第 7 章。

个旅游预定门户网站 (travel reservation portal)：用户输入旅游的日期和其他条件，门户网站获取并显示多条航线、旅店或者汽车租赁公司的报价。根据不同的公司，获取报价的过程可能涉及到调用一个 Web Service、访问数据库、执行一个 EDI 事务或者其他的机制。与其让页面的响应时间受限于最慢的一个响应，不如让页面只显示在给定预定时间内获得的信息，这样更好些。对于没有及时响应的服务提供者，页面要么完全忽略它们，要么显示一个占位符，比如像 “Did not hear from Air Java in time。”

清单 6.16 在预定时间内获取广告信息

```
Page renderPageWithAd() throws InterruptedException {
    long endNanos = System.nanoTime() + TIME_BUDGET;
    Future<Ad> f = exec.submit(new FetchAdTask());
    // 等待广告是呈现页面
    Page page = renderPageBody();
    Ad ad;
    try {
        // 在预计时间内等待
        long timeLeft = endNanos - System.nanoTime();
        ad = f.get(timeLeft, NANOSECONDS);
    } catch (ExecutionException e) {
        ad = DEFAULT_AD;
    } catch (TimeoutException e) {
        ad = DEFAULT_AD;
        f.cancel(true);
    }
    page.setAd(ad);
    return page;
}
```

从一个公司获得报价的过程和从其他公司获得报价的过程是无关的，所以获取单一报价是一个明显的任务边界，它允许并发地处理获得的报价。为此创建 n 个任务，提交到一个线程池，再获得 n 个 Futures，最后使用一个限时的 get 方法通过 Future 顺序地获取每一个结果，这一切做起来是很简单的，然而还有一个更简单的方法——invokeAll。

清单 6.17 使用限时版本的 invokeAll，将多个任务提交到一个 ExecutorService，并且获得其结果。InvokeAll 方法处理一个任务的容器 (collection)，并返回一个 Future 的容器。两个容器具有相同的结构；invokeAll 将 Future 添加到返回的容器中，这样可以使用任务容器的迭代器，从而调用者可以将它表现的 Callable 与 Future 关联起来。当所有任务都完成时、调用线程被中断时或者超过时限，限时版本的 invokeAll 都会返回结果。超过时限后，任何尚未完成的任务都会被取消。作为 invokeAll 的返回值，每个任务要么正常地完成，要么被取消；客户端代码可以调用 get 或者 isCancelled 来查明是属于哪一种情况。

总结

围绕**任务**的执行来构造应用程序，可以简化开发，便于同步。Executor 框架有助于你在任务的提交与任务的执行策略之间进行解耦，同时还支持很多不同类型的执行策略。你发现自己为执行任务而创建线程时，可以考虑使用 Executor 取代以前的方法。把应用程序分解为不同的任务，为了使这一行为产生最大的效益，你必须指明一个清晰的任务边界。在一些应用程序中，存在明显的工作良好的任务边界，然而还有一些程序，你需要作进一步的分析，以揭示更多可加强的并行性。

清单 6.17 在预定时间内请求旅游报价

```
private class QuoteTask implements Callable<TravelQuote> {
    private final TravelCompany company;
    private final TravelInfo travelInfo;
    ...
    public TravelQuote call() throws Exception {
        return company.solicitQuote(travelInfo);
    }
}

public List<TravelQuote> getRankedTravelQuotes(
    TravelInfo travelInfo, Set<TravelCompany> companies,
    Comparator<TravelQuote> ranking, long time, TimeUnit unit)
    throws InterruptedException {
    List<QuoteTask> tasks = new ArrayList<QuoteTask>();
    for (TravelCompany company : companies)
        tasks.add(new QuoteTask(company, travelInfo));

    List<Future<TravelQuote>> futures =
        exec.invokeAll(tasks, time, unit);

    List<TravelQuote> quotes =
        new ArrayList<TravelQuote>(tasks.size());
    Iterator<QuoteTask> taskIter = tasks.iterator();
    for (Future<TravelQuote> f : futures) {
        QuoteTask task = taskIter.next();
        try {
            quotes.add(f.get());
        } catch (ExecutionException e) {
            quotes.add(task.getFailureQuote(e.getCause()));
        } catch (CancellationException e) {
            quotes.add(task.getTimeoutQuote(e));
        }
    }

    Collections.sort(quotes, ranking);
    return quotes;
}
```

取消和关闭

Cancellation and Shutdown

启动任务和线程都很容易。大多数时候，我们通常允许它们在结束任务后自行停止。但是，有时候我们希望在任务或线程自然结束之前就停止它们，可能因为用户取消了操作，或者应用程序需要快速关闭。

要做到安全、快速、可靠地停止任务或线程并不容易。Java 没有提供任何机制，来安全地强迫线程停止手头的工作¹。它提供**中断**——一个协作机制，使一个线程能够要求另一个线程停止当前的工作。

这种协作方法是必须的，因为我们很少需要一个任务、线程或者服务**立即**停止，立即停止会导致共享的数据结构处于不一致的状态。任务和服务可以这样编码：当要求它们停止时，它们首先会清除当前进程中的工作，**然后再**终止。这提供了更好的灵活性，因为任务代码本身比发出取消请求的代码更明确应该清除什么。

生命周期结束的问题使任务、服务以及程序的设计和实现变得复杂起来，这个程序设计中非常重要的元素却经常被忽略。处理好失败、关闭和取消是好的软件和勉强运行的软件最大的区别。这一章将开始着手处理取消和中断的机制，并且还将涉及如何编写任务和服务的代码，使它们响应取消请求。

7.1 任务取消

当外部代码能够在活动自然完成之前，把它更改为完成状态，那么这个活动被称为**取消的**（cancellable）。我们可能会因为很多原因取消一个活动：

¹ Thread.stop和suspend方法试图提供这样的机制，但是很快发现了严重的缺陷，应该避免使用。这些方法的问题说明参见 <http://java.sun.com/j2se/1.5.0/docs/guide/misc/threadPrimitiveDeprecation.html>。

用户请求的取消。用户点击程序界面上的“cancel”按钮，或者通过管理接口请求取消，比如 JMX(Java Management Extensions)。

限时活动。一个应用程序，需要在有限的时间内搜索问题空间，并且在规定时间内选择最好的解决方案。如果计时器超时，正在搜索的任务会被取消。

应用程序事件。一个应用程序对问题空间进行分解搜索，使不同的任务搜索问题空间中不同的区域。当一个任务发现了解决方案，所有其他仍在工作的搜索会被取消。

错误。一个 Web Crawler 搜索几个相关页面，并把页面或概要数据存储在硬盘。当 Crawler 的任务遭遇错误（比如，磁盘空间已满），那么所有的搜索任务都会被取消，不过很可能会记录它们当前的状态，这样稍后可以重新启动。

关闭。当一个程序或者服务关闭时，必须对正在处理的和等待处理的工作进行一些操作。一个优雅的关闭，可能允许当前的任务完成；在一个更加紧迫的关闭中，当前的任务就可能被取消了。

在 Java 中，没有哪一种用来停止线程的方式是绝对安全的，因此没有哪一种方式优先用来停止任务。这里只有选择相互协作的机制，通过协作，使任务和代码遵循一个统一的协议，用来请求取消（cancellation）。

在这些的协作机制中，有一种会设置“cancellation requested”标志，任务会定期查看；如果发现标志被设置过，任务就会提前结束。清单 7.1 中的 PrimeGenerator 的功能是列举素数，直到它被取消，这个例子解释了这项技术。cancel 方法会设置 cancelled 标志，主循环会在搜索下一个素数之前，轮询检查这个标志。（为了让这个过程变可靠，cancelled 必须是 volatile 类型的。）

清单 7.2 展示了一个使用这个类的例子，让素数生成器运行一秒钟后就取消。生成器没有必要严格在一秒内停止，这时因为请求取消的时间和 run 循环的下一次检查之间可能存在延迟。cancel 方法由 finally 块调用，来保证即使在调用 sleep 被中断的情况下，素数生成器也能被取消。如果 cancel 没有被调用，寻找素数的线程会永远运行下去，占用 CPU 的周期，使 JVM 不能正常退出。

一个可取消的任务必须拥有**取消策略**（cancellation policy），这个策略详细说明关于取消的“how”、“when”、“what”——其他代码如何请求取消该任务，任务在什么时机检查取消的请求是否到达，响应取消请求的任务中应有的行为。

考虑现实世界中停止支付支票的例子。对于如何提交一个停止支付的请求，何时响应以保证这个请求一定会被处理，以及当支付真的被停止后还要遵守哪些流程（比如通知其

清单 7.1 使用 volatile 域保存取消状态

```
@ThreadSafe
public class PrimeGenerator implements Runnable {
    @GuardedBy("this")
    private final List<BigInteger> primes
        = new ArrayList<BigInteger>();
    private volatile boolean cancelled;

    public void run() {
        BigInteger p = BigInteger.ONE;
        while (!cancelled) {
            p = p.nextProbablePrime();
            synchronized (this) {
                primes.add(p);
            }
        }
    }
    public void cancel() { cancelled = true; }

    public synchronized List<BigInteger> get() {
        return new ArrayList<BigInteger>(primes);
    }
}
```

清单 7.2 生成素数的程序运行一秒钟

```
List<BigInteger> aSecondOfPrimes() throws InterruptedException {
    PrimeGenerator generator = new PrimeGenerator();
    new Thread(generator).start();

    try {
        SECONDS.sleep(1);
    } finally {
        generator.cancel();
    }

    return generator.get();
}
```

他银行参与进来，对付款人的账户进行评估），这些银行都有相关的规定。这些流程和保证加在一起组成了支票支付的取消策略。

`PrimeGenerator` 使用了简单的取消政策：客户端代码通过调用 `cancel` 请求取消，`PrimeGenerator` 每次搜索素数前检查是否有取消请求，当发现取消请求时就退出。

7.1.1 中断

`PrimeGenerator` 中的取消机制最终会导致寻找素数的任务退出，但是并不是立刻发生，需要花费一些时间。但是如果一个任务使用这个方案调用一个阻塞方法，比如 `BlockingQueue.put`，我们可能会遇到一个更严重的问题——任务可能永远都不检查取消标志，因此永远不会终结。

清单 7.3 中的 `BrokenPrimeProducer` 揭示了这个问题。生产者线程生成素数，并把它们放入一个阻塞队列。如果生产者的速度超过了消费者，队列会被填满，`put` 方法会被阻塞。当 `put` 方法被阻塞的时候，消费者如果试图去取消生产者的任务，会发生什么事情呢？它会调用 `cancel` 方法设置 `cancelled` 标志——但是此时生产者永远不可能检查这个标志了，因为它已经被 `put` 方法阻塞住了（又因为消费者此时已经停止从队列中取出素数，所以 `put` 方法会一直保持阻塞状态）。

正像我们在第 5 章中所暗示的那样，特定阻塞库类的方法支持**中断**。线程中断是一个协作机制，一个线程给另一个线程发送信号（`signal`），通知它在方便或者可能的情况下停止正在做的工作，去做其他事情。

在 API 和语言规范中，并没有把中断与任何取消的语意绑定起来，但是，实际上，使用中断来处理取消之外的任何事情都是不明智的，并且很难支撑起更大的应用。

每一个线程都有一个 `boolean` 类型的**中断状态**（`interrupted status`）；在中断的时候，这个中断状态被设置为 `true`。`Thread` 包含其他用于中断线程的方法，以及请求线程中断状态的方法，正如我们在清单 7.4 中所表现的那样。`interrupt` 方法中断目标线程，并且 `isInterrupted` 返回目标线程的中断状态。静态的 `interrupted` 方法名并不理想，它仅仅能够**清除**当前线程的中断状态，并返回它之前的值；这是清除中断状态唯一的方法。

阻塞库函数，比如：`Thread.sleep` 和 `Object.wait`，试图监测线程何时被中断，并提前返回。它们对中断的响应表现为：清除中断状态，抛出 `InterruptedException`；这表示阻塞操作因为中断的缘故提前结束。**JVM** 并没有对阻塞方法发现中断的速度作出保证，不过在现实中这样的响应还是比较迅速的。

清单 7.3 不可靠的取消把生产者置于阻塞的操作中（不要这样做）

```
class BrokenPrimeProducer extends Thread {
    private final BlockingQueue<BigInteger> queue;
    private volatile boolean cancelled = false;

    BrokenPrimeProducer(BlockingQueue<BigInteger> queue) {
        this.queue = queue;
    }

    public void run() {
        try {
            BigInteger p = BigInteger.ONE;
            while (!cancelled)
                queue.put(p = p.nextProbablePrime());
        } catch (InterruptedException consumed) { }
    }

    public void cancel() { cancelled = true; }
}

void consumePrimes() throws InterruptedException {
    BlockingQueue<BigInteger> primes = ...;
    BrokenPrimeProducer producer = new BrokenPrimeProducer(primes);
    producer.start();

    try {
        while (needMorePrimes())
            consume(primes.take());
    } finally {
        producer.cancel();
    }
}
```



清单 7.4 线程的中断方法

```
public class Thread {
    public void interrupt() { ... }
    public boolean isInterrupted() { ... }
    public static boolean interrupted() { ... }
    ...
}
```

当线程在并不处于阻塞状态的情况下发生中断时，会设置线程的中断状态，然后一直等到被取消的活动获取中断状态，来检查是否发生了中断。通过这样的方法使中断变“粘”——如果不触发 `InterruptedException`，中断状态会一直保持，直到有人特意去清除中断状态。

调用 `interrupt` 并不意味着必然停止目标线程正在进行的工作；它仅仅传递了请求中断的消息。

我们对中断本身最好的理解应该是：它并不会真正中断一个正在运行的线程；它仅仅发出**中断请求**，线程自己会在下一个方便的时刻中断（这些时刻被称为**取消点**，`cancellation point`）。有一些方法对这样的请求很重视，比如 `wait`、`sleep` 和 `join` 方法，当它们接到中断请求时会抛出一个异常，或者进入时中断状态就已经被设置了。运行良好的方法能够完全忽略这样的请求，只要它们把中断请求置于适当的位置上，留给调用代码进行处理。运行不好的方法可能会掩藏中断请求，这样会使调用栈中的其他的代码失去对其作出响应的机会。

静态的 `interrupted` 应该小心使用，因为它会清除并发线程的中断状态。如果你调用了 `interrupted`，并且它返回了 `true`，你必须对其进行处理，除非你想掩盖这个中断——你可以抛出 `InterruptedException`，或者通过再次调用 `interrupt` 来保存中断状态，正如第 94 页清单 5.10 所示。

`BrokenPrimeProducer` 解释了为什么常用的取消机制不能与可阻塞的库函数进行良好互动的的原因。如果你的任务代码响应中断，那么你可以使用中断作为你的取消机制，并利用很多库类对中断的支持。

中断通常是实现取消最明智的选择。

`BrokenPrimeProducer` 可以通过使用中断，而不是 `boolean` 标志来请求取消（`cancellation`），这将很容易解决（并简化）问题，正如清单 7.5 中所示。在每一个迭代循环中，存在两个点可能发现中断：在调用阻塞的 `put` 方法时，以及在循环开始处显式地采集到中断状态时。在这里，因为调用了阻塞的 `put` 方法，显式的检测并不是绝对必要的，但是这种检测会使 `PrimeProducer` 对中断具有更好的响应性。这是因为它在耗时的任务——寻找素数——开始之前就检查了中断，而不是在任务完成之后才去检查。当我们调用可中断的阻塞方法时，通常并不能得到期望的响应，对中断状态进行显式的检测会对此有一定的帮助。

清单 7.5 通过使用中断进行取消

```
class PrimeProducer extends Thread {
    private final BlockingQueue<BigInteger> queue;
    PrimeProducer(BlockingQueue<BigInteger> queue) {
        this.queue = queue;
    }

    public void run() {
        try {
            BigInteger p = BigInteger.ONE;
            while (!Thread.currentThread().isInterrupted())
                queue.put(p = p.nextProbablePrime());
        } catch (InterruptedException consumed) {
            /*允许线程退出 */
        }
    }

    public void cancel() { interrupt(); }
}
```

7.1.2 中断策略

正如需要为任务制定取消策略一样，也应该制定线程**中断策略**（interruption policy）。一个中断策略决定线程如何应对中断请求——当发现中断请求时，它会做什么（如果确实响应中断的话），哪些工作单元对于中断来说是原子操作，以及在多快的时间里响应中断。

中断策略中最有意义的是对线程级（thread-level）和服务级（service level）取消的规定：尽可能迅速退出，如果需要的话进行清理，可能的话通知其拥有的实体，这个线程已经退出。很可能建立其他中断策略，比如暂停和重新开始，但是那些具有非标准中断策略的线程或线程池，需要被约束于那些应用了该策略的任务中。

区分**任务**和**线程**对中断的反应是很重要的。一个单一的中断请求可能有一个或一个以上预期的接收者——在线程池中中断一个工作者线程，意味着取消当前任务，并关闭工作线程。

任务不会在自己拥有的线程中执行；它们借用属于服务的线程，比如线程池。代码如果并不是线程的所有者（对于线程池而言，是指任何线程池实现以外的代码）就应该小心地保存中断状态，这样所有者的代码才能够最终对其起到作用，甚至是“访客”代码也可能起到作用（当你为一户人家打扫房屋时，如果房屋主人不在，你不会把他的这段时间收到的邮件丢掉——你会把邮件收起来，等他们回来以后再交给他们处理，尽管你会悄悄阅读他们的杂志）。

这就是为什么大多数可阻塞的库函数，仅仅抛出 `InterruptedException` 作为中断的响应。它们决不可能自己运行在一个线程中，所以他们为任务或者库代码实现了大多数合理的取消策略：它们会尽可能快地为异常信息让路，把它们向后传给调用者，这样上层栈的代码就可以进一步行动了。

当检查到中断请求时，任务并不需要放弃所有事情——它可以选择推迟，直到更合适的时机。这需要记得它已经被请求过中断了，完成当前正在进行的任务，**然后**抛出 `InterruptedException` 或者指明中断。当更新的过程中发生中断时，这项技术能够保证数据结构不被彻底破坏。

一个任务不应该假设其执行线程的中断策略，除非它显式地设计用来运行在服务中，并且这些服务有明确的中断策略。无论任务把中断解释为取消，还是其他的一些关于中断的操作，它都应该注意保存执行线程的中断状态。如果对中断的处理不仅仅是把 `InterruptedException` 传递给调用者，那么它应该在捕获 `InterruptedException` 之后恢复中断的状态：

```
Thread.currentThread().interrupt();
```

正像任务代码不应该猜测中断对于它的执行策略意味着什么，取消代码也不应该对任何线程的中断策略进行假设。线程应该只能够被线程的所有者中断；所有者可以把线程的中断策略信息封装到一个合适的取消机制中，比如关闭（shutdown）方法。

因为每一个线程都有其自己的中断策略，所以你不应该中断线程，除非你知道中断对这个线程意味着什么。

批评者嘲笑 Java 的中断工具，因为它没有提供优先中断的能力，而且还强迫开发者处理 `InterruptedException`。但是推迟中断请求的能力使开发者能够制定更灵活的中断策略，从而实现适合于程序的响应性和健壮性之间的平衡。

7.1.3 响应中断

我们在 5.4 节中提到过，当你调用可中断的阻塞函数时，比如 `Thread.sleep` 或者 `BlockingQueue.put`，有两种处理 `InterruptedException` 的实用策略：

- 传递异常（很可能发生在特定任务的清除时），使你的方法也成为可中断的阻塞方法；
- 或者保存中断状态，上层调用栈中的代码能够对其进行处理。

传递 `InterruptedException` 只需要简单地把 `InterruptedException` 添加到 `throws` 子句中，正如清单 7.6 中 `getNextTask` 所示。

清单 7.6 向调用者传递 `InterruptedException`

```
BlockingQueue<Task> queue;  
...  
public Task getNextTask() throws InterruptedException {  
    return queue.take();  
}
```

如果你不想，或不能传递 `InterruptedException`（或许因为你的任务被定义为 `Runnable` 类型），你需要寻找另一种方式保存中断请求。实现这个目的的标准方法是再次调用 `interrupt` 来恢复中断状态。你不应该掩盖 `InterruptedException`，在 `catch` 块中捕获到异常却什么也不做，倘若你的代码真正实现了线程的中断策略，你才应该捕获这个异常。虽然 `PrimeProducer` 也掩盖了中断，但是它之所以可以这样做，是因为线程即将要终止，因此调用栈上方已经没有代码需要知晓中断了。大多数代码并不知道它们会在哪个线程中运行，所以应该保存中断状态。

只有实现了线程中断策略的代码才可以接收中断请求。通用目的的任务和库的代码绝不应该接收中断请求。

有一些活动不支持取消，却仍可能调用可中断的阻塞方法，那么它们必须在循环中调用这些方法，当发现中断后重新尝试。在这样的情况下，它们应该如清单 7.7 所示的，在本地保存中断状态，并在返回前恢复状态，而不是立刻上前捕获 `InterruptedException`。过早设置中断可能会引起无限循环，因为大多数可中断的阻塞方法在入口时检查中断状态，并且如果该状态已被设置，那么就会立刻抛出 `InterruptedException`（可中断方法通常会在阻塞或进行重要工作前检查中断，这样可以尽快响应中断）。

如果你的代码不会调用可中断的阻塞方法，它仍然可以通过检查任务代码当前线程的中断状态来响应中断。选择适当的检查频率需要在效率和响应性之间进行权衡。如果你有高响应性的需求，那么你不应该调用潜在耗时的方法，当这些方法不会响应中断时，这里有一个不成文的规定：限制你的选择，不要去调用库代码。

取消可能涉及 `state`，而不是中断状态（`status`）；中断可以用来获得线程的关注，同时，中断线程在其他地方存储下来的信息，可以用来进一步说明被中断的线程（访问这些信息时，要确保使用同步）。

清单 7.7 不可取消的任务在退出前保存中断

```
public Task getNextTask(BlockingQueue<Task> queue) {
    boolean interrupted = false;
    try {
        while (true) {
            try {
                return queue.take();
            } catch (InterruptedException e) {
                interrupted = true;
                // 失败并重试
            }
        }
    } finally {
        if (interrupted)
            Thread.currentThread().interrupt();
    }
}
```

例如，一个工作者线程的所有者为 `ThreadPoolExecutor`，这个线程在检测中断时，它会检查线程池是否被关闭。如果是，它执行一些资源池清理工作；否则它可能创建一个新线程，用来保持线程池处在期望的大小。

7.1.4 示例：计时运行

很多问题永远也不可能解决（比如，列举所有质数）；对于其他看起来能够解决的问题，有些可能很快就能得到答案，但是也有可能永远得不到答案。在这样的情况下，如果能够设定“花十分钟寻找答案”或者“列举出十分钟内能够找到的答案”，是非常有用处的。

清单 7.2 中的 `aSecondOfPrimes` 方法，启动了 `PrimeGenerator`，并在一秒钟后中断。然而，`PrimeGenerator` 可能需要大于一秒的时间才能停止，但是它最终会发现中断，并发出停止指令，停止线程。但是执行任务的另外一个方面是，你希望知道执行任务的过程中是否会抛出异常。如果在时间期限内，`PrimeGenerator` 抛出一个未经检查的异常，那么这个异常很容易被忽略，因为生成素数的程序在另一个可独立线程中运行，并没有显式地处理异常。

清单 7.8 的程序表现的是一段给定时间内尝试运行一个 `Runnable`。它在调用线程中运行任务，并在调度中安排了一个取消任务，由这个任务在给定时间间隔后中断它。这样解决了任务抛出未检查的异常的问题，因为异常会被 `timedRun` 的调用者捕获。

这是一个很简单的解决方法，但是破坏了下面的规则：在中断线程之前，你应该了解它的中断策略。因为 `timedRun` 可以被任意一个线程调用，但是我们不可能了解这个调

清单 7.8 在外部线程中安排中断（不要这样做）

```
private static final ScheduledExecutorService cancelExec = ...;

public static void timedRun(Runnable r,
                           long timeout, TimeUnit unit) {
    final Thread taskThread = Thread.currentThread();
    cancelExec.schedule(new Runnable() {
        public void run() { taskThread.interrupt(); }
    }, timeout, unit);
    r.run();
}
```



用线程的中断策略。如果任务在时限之前完成，执行中断一个线程（这个线程调用了 `timedRun`）的取消任务可能会在 `timedRun` 返回到它调用者之后，才被启动。我们不知道这发生的时候，将执行什么样的代码，但是结果一定不好。（我们可以通过使用 `schedule` 返回的 `ScheduledFuture` 来取消这个取消任务，以此来规避风险。这种做法是可能的，而且相当地取巧。）

进一步来看，如果任务不响应中断，`timedRun` 将不会返回，直到任务结束，这时可能已经超过期望的限定时间很久了（或者根本还没有到时）。一个限时运行的服务如果没有在给定时间内返回的话，这非常有可能惹恼调用者。

清单 7.9 解决了 `aSecondOfPrimes` 的异常处理问题，并且也解决了在前面的做法中引发的问题。用来执行任务的线程拥有自己的执行策略，即使任务不响应中断，限时运行的方法仍然能够返回到它的调用者。在开始任务线程之后，`timedRun` 会在新创建的线程中执行一个限时的 `join` 方法。在 `join` 返回后，它会检查任务中是否有异常抛出，如果是，那么会在调用 `timedRun` 的线程中再抛出一次。保存下来的 `Throwable` 在两线程间共享，声明为 `volatile` 也是同样道理，它被安全地发布，从任务线程再到 `timedRun` 线程。

这个版本解决了前面的例子中出现的问题，但是因为它依赖于一个限时的 `join` 方法，因此它也受到 `join` 不足之处的影响：我们不知道控制权的返回是因为线程自然退出还是 `join` 超时²。

7.1.5 通过 Future 取消

我们曾经用到过一个抽象体，它可以管理任务的生命周期，处理异常，并有利于取消——`Future`。根据通常的原理，使用现有的库类好于生成一个自己的类，就让我们继续使用 `Future` 和任务执行框架来构建 `timedRun` 吧。

² 这是线程API的一个缺陷，因为无论`join`是否成功完成，在Java存储模型中都会相应存在内存可见性，但是`join`本身不会返回表示它成功与否的任何状态。

清单 7.9 在一个专门的线程中断任务

```

public static void timedRun(final Runnable r,
                             long timeout, TimeUnit unit)
    throws InterruptedException {
    class RethrowableTask implements Runnable {
        private volatile Throwable t;
        public void run() {
            try { r.run(); }
            catch (Throwable t) { this.t = t; }
        }
        void rethrow() {
            if (t != null)
                throw launderThrowable(t);
        }
    }

    RethrowableTask task = new RethrowableTask();
    final Thread taskThread = new Thread(task);
    taskThread.start();
    cancelExec.schedule(new Runnable() {
        public void run() { taskThread.interrupt(); }
    }, timeout, unit);
    taskThread.join(unit.toMillis(timeout));
    task.rethrow();
}

```



`ExecutorService.Submit` 会返回一个 `Future` 来描述任务。`Future` 有一个 `cancel` 方法，它需要一个 `boolean` 类型的参数，`mayInterruptIfRunning`，它的返回值表示取消尝试是否成功（这仅仅告诉你它是否能够接收中断，而不是任务是否检测并处理了中断）。当 `mayInterruptIfRunning` 为 `true`，并且任务当前正在运行于一些线程中，那么这个线程是应该中断的。把这个参数设置成 `false` 意味着“如果还没有启动的话，不要运行这个任务”，这应该用于那些不处理中断的任务。

除非你知道线程的中断策略，否则你不应该中断线程，那么什么时候可以采用一个 `true` 作为参数调用 `cancel`？任务执行线程是由标准的 `Executor` 实现创建的，它实现了一个中断策略，使得任务可以通过中断被取消，所以当它们在标准 `Executor` 中运行时，通过它们的 `Future` 来取消任务，这时设置 `mayInterruptIfRunning` 是安全的。当尝试取消一个任务的时候，你不应该直接中断线程池，因为你不知道中断请求到达时，什么任务正在运行——只能通过任务的 `Future` 来做这件事情。这便是编写任务，让它视中断为取消请求的另一个理由：能够通过它们的 `Future` 被取消。

清单 7.10 展现了 `timedRun` 的另一个版本，向 `ExecutorService` 提交了任务，并通过定时的 `Future.get` 获得结果。如果 `get` 终止于一个 `TimeoutException`，那么任务是由 `Future` 取消的（为了简化代码，这个版本在 `finally` 块中无条件调用 `Future.cancel`，可以这样做是基于下面的事实：取消一个已完成的任务不会有任何影响）。如果深层计算在取消前就抛出一个异常，这个异常在 `timedRun` 中会重新被抛出，这是调用者处理异常最简单的方法。清单 7.10 也说明了另一个非常好的实践：取消那些不再需要结果的任务。（这个技术同样还用于第 128 页的清单 6.13 和第 132 页的清单 6.16 中）。

清单 7.10 通过 Future 来取消任务

```
public static void timedRun(Runnable r,
                           long timeout, TimeUnit unit)
    throws InterruptedException {
    Future<?> task = taskExec.submit(r);
    try {
        task.get(timeout, unit);
    } catch (TimeoutException e) {
        // 下面任务会被取消
    } catch (ExecutionException e) {
        // task 中抛出的异常；重抛出
        throw launderThrowable(e.getCause());
    } finally {
        // 如果任务已经结束，是无害的
        task.cancel(true); // interrupt if running
    }
}
```

当 `Future.get` 抛出 `InterruptedException` 或 `TimeoutException` 时，如果你知道不再需要结果时，就可以调用 `Future.cancel` 来取消任务了。

7.1.6 处理不可中断阻塞

很多可阻塞的库方法通过提前返回和抛出 `InterruptedException` 来实现对中断的响应，这使得构建可以响应取消的任务更加容易了。但是，并不是所有的阻塞方法或阻塞机制都响应中断；如果一个线程是由于进行同步 `Socket I/O` 或者等待获得内部锁而阻塞的，那么中断除了能够设置线程的中断状态以外，什么都不能改变。对于那些被不可中断的活动所阻塞的线程，我们可以使用与中断类似的手段，来确保可以停止这些线程。但是这需要我们更清楚地知道线程为什么会被阻塞。

java.io 中的同步 Socket I/O。在服务器应用程序中，阻塞 I/O 最常见的形式是读取和写入 Socket。不幸的是，InputStream 和 OutputStream 中的 read 和 write 方法都不响应中断，但是通过关闭底层的 Socket，可以让 read 或 write 所阻塞的线程抛出一个 SocketException。

java.nio 中的同步 I/O。中断一个等待 InterruptibleChannel 的线程，会导致抛出 ClosedByInterruptException，并关闭链路（也会导致其他线程在这条链路的阻塞，抛出 ClosedByInterruptException）。关闭一个 InterruptibleChannel 导致多个阻塞在链路操作上的线程抛出 AsynchronousCloseException。大多数标准 Channels 都实现 InterruptibleChannel。

Selector 的异步 I/O。如果一个线程阻塞于 Selector.select 方法（在 java.nio.channels 中），close 方法会导致它通过抛出 ClosedSelectorException 提前返回。

获得锁。如果一个线程在等待内部锁，那么如果不能确保它最终获得锁，并且作出足够的努力，让你能够以其他方式获得它的注意，你是不能停止它的。然而，显式 Lock 类提供了 lockInterruptibly 方法，允许你等待一个锁，并仍然能够响应中断——见第 13 章。

清单 7.11 中 ReaderThread 展现了一项用来封装非标准取消的技术。ReaderThread 管理一个单线程 Socket 连接，同步地从 Socket 中读取数据，把接收到的数据传递给 processBuffer。为了方便终止一个用户的连接，或关闭服务器，ReaderThread 重写了 interrupt 方法，既为了支持标准的中断，也为了关闭潜在 Socket，因此中断 ReaderThread 使得它能够在进行工作时停止，而无论它是被 read 阻塞的，还是被可中断的阻塞方法所阻塞。

7.1.7 用 newTaskFor 封装非标准取消

在 ReaderThread 中，可以使用 newTaskFor 钩子函数来改进用来封装非标准取消的方法，这是 Java 6 中添加到 ThreadPoolExecutor 的新特性。当提交一个 Callable 给 ExecutorService 时，submit 返回一个 Future，可以用 Future 来取消任务。newTaskFor 钩子是一个工厂方法，创建 Future 来代表任务。它返回一个 RunnableFuture，这是一个接口，它扩展了 Future 和 Runnable（并由 FutureTask 实现）。

自定义的任务 Future 允许你覆写 Future.cancel 方法。自定义的取消代码可以实现日志或者收集取消的统计信息，并可以用来取消那些不响应中断的活动。通过覆写 interrupt，ReaderThread 封装了使用 Socket 的线程的取消行为；同样的事情也可以通过覆写任务的 Future.cancel 方法来实现。

清单 7.12 中的 CancellableTask 定义了一个 CancellableTask 接口，这个接口扩展了 Callable，并加入了 cancel 方法和一个新的 newTask 工厂方法，来构造 RunnableFuture。CancellingExecutor 扩展了 ThreadPoolExecutor，并覆写了

清单 7.11 在 Thread 中, 通过覆写 interrupt 来封装非标准取消

```
public class ReaderThread extends Thread {
    private final Socket socket;
    private final InputStream in;
    public ReaderThread(Socket socket) throws IOException {
        this.socket = socket;
        this.in = socket.getInputStream();
    }

    public void interrupt() {
        try {
            socket.close();
        }
        catch (IOException ignored) { }
        finally {
            super.interrupt();
        }
    }

    public void run() {
        try {
            byte[] buf = new byte[BUFSZ];
            while (true) {
                int count = in.read(buf);
                if (count < 0)
                    break;
                else if (count > 0)
                    processBuffer(buf, count);
            }
        } catch (IOException e) { /* 允许线程退出 */ }
    }
}
```

`newTaskFor`，让 `CancellableTask` 可以创建自己的 `Future`。

`SocketUsingTask` 实现了 `CancellableTask`，并定义了 `Future.cancel` 来关闭 `Socket`，`super.cancel` 也是同样的道理。如果 `SocketUsingTask` 通过自身的 `Future` 被取消，`Socket` 会被关闭并且执行线程会被中断。这提高了任务对取消的响应性。这样做在保证响应取消的同时，不仅可以安全地调用可中断方法，而且它也可以调用阻塞中的 `Socket I/O` 的方法。

7.2 停止基于线程的服务

应用程序通常会创建拥有线程的服务，比如线程池，这些服务的存在时间通常比创建它们的方法存在的时间更长。如果应用程序优雅地退出，这些服务的线程也需要结束。因为没有退出线程惯用的优先方法，它们需要自行结束。

明智的封装实践指出，你不应该操控某个线程——中断它，改变它的优先级，等等——除非你拥有这个线程。线程 API 没有关于线程所属权正规的概念。线程通过一个 `Thread` 对象表示，像其他对象一样，线程可以被自由地共享。但是，认为线程有一个对应的拥有者是有道理的，这个拥有者就是创建线程的类。所以线程池拥有它的工作者线程，如果需要中断这些线程，那么应该由线程池来负责。

正如其他被封装的对象一样，线程的所有权是不可传递的：应用程序可能拥有服务，服务可能拥有工作者线程，但是应用程序并不拥有工作者线程，因此应用程序不应该试图直接停止工作者线程。相反，服务应该提供生命周期方法（`lifecycle methods`）来关闭它自己，并关闭它所拥有的线程；那么当应用程序关闭这个服务时，服务就可以关闭所有的线程了。`ExecutorService` 提供了 `shutdown` 和 `shutdownNow` 方法，其他线程持有的服务也应该都提供类似的关闭机制。

对于线程持有的服务，只要服务的存在时间大于创建线程的方法存在的时间，那么就应该提供生命周期方法。

7.2.1 示例：日志服务

大多数服务器应用程序都使用日志，日志可以简单到向代码中插入一条 `println` 语句。`PrintWriter` 这样的字符流类是线程安全的，所以这样简单的方法不需要显式地调用同步³。

³ 如果你在单一日志消息中写入多行，你可能需要附加的客户端锁，来避免多线程交替输出。如果两个线程把多行栈追踪信息追加到同一 `println` 提供的相同的字符流中，结果就会以无法预测的方式交错输出，很可能看起来像就是杂乱无章的毫无意义的栈追踪信息。

清单 7.12 用 newTaskFor 封装任务中非标准取消

```
public interface CancellableTask<T> extends Callable<T> {
    void cancel();
    RunnableFuture<T> newTask();
}

@ThreadSafe
public class CancellingExecutor extends ThreadPoolExecutor {
    ...
    protected<T> RunnableFuture<T> newTaskFor(Callable<T> callable) {
        if (callable instanceof CancellableTask)
            return ((CancellableTask<T>) callable).newTask();
        else
            return super.newTaskFor(callable);
    }
}

public abstract class SocketUsingTask<T>
    implements CancellableTask<T> {
    @GuardedBy("this") private Socket socket;

    protected synchronized void setSocket(Socket s) { socket = s; }

    public synchronized void cancel() {
        try {
            if (socket != null)
                socket.close();
        } catch (IOException ignored) { }
    }

    public RunnableFuture<T> newTask() {
        return new FutureTask<T>(this) {
            public boolean cancel(boolean mayInterruptIfRunning) {
                try {
                    SocketUsingTask.this.cancel();
                } finally {
                    return super.cancel(mayInterruptIfRunning);
                }
            }
        };
    }
}
```


然而，在 11.6 节中，我们将看到这样单行的日志在一些大容量的（high volume）应用中可能存在一些性能开销。另外一种替代方法是使 log 把日志记录派发给其他线程处理。

清单 7.13 中 LogWriter 展现了一个简单的日志服务示例，其中，日志活动被分离到一个单独的日志线程中。产生消息的线程不会将消息直接写入输出流，而是由 LogWriter 通过 BlockingQueue 把这个任务移交给了日志线程，并由日志线程写入。这是一个多生产者、单消费者的设计：所有活动都调用 log，这些活动就作为生产者，后台的日志线程是消费者。如果消费者落后了，BlockingQueue 稍后会阻塞生产者，直到日志线程跟上来。

清单 7.13 不支持关闭的生产者-消费者日志服务

```
public class LogWriter {
    private final BlockingQueue<String> queue;
    private final LoggerThread logger;
    public LogWriter(Writer writer) {
        this.queue = new LinkedBlockingQueue<String>(CAPACITY);
        this.logger = new LoggerThread(writer);
    }

    public void start() { logger.start(); }

    public void log(String msg) throws InterruptedException {
        queue.put(msg);
    }

    private class LoggerThread extends Thread {
        private final PrintWriter writer;
        ...
        public void run() {
            try {
                while (true)
                    writer.println(queue.take());
            } catch (InterruptedException ignored) {
            } finally {
                writer.close();
            }
        }
    }
}
```



为了让一个类似于 LogWriter 的服务在生产中真正可用，我们需要一个方法来终止日志线程，这样不会让 JVM 无法正常关闭。停止日志线程是很容易的，因为它重复调用

take, 而 take 响应中断; 如果日志线程被修改为捕获到 InterruptedException 就退出, 那么中断日志线程就能够停止服务。

但是, 简单地使日志线程退出并不是令人十分满意的关闭机制。这样突然的关闭忽略了等待中需要被记录的日志, 但更重要的是, 线程会因为队列已满, 在 log 处被阻塞, 却永远不可能从阻塞解脱出来。取消一个生产者-消费者活动既要取消生产者, 又要取消消费者。中断日志线程会着手处理消费者, 但是这个例子中, 因为生产者非专一线程, 取消它们将变得非常困难。

另一个关闭 LogWriter 的方案可以设置“已请求关闭”标志, 避免消息进一步被提交进来, 正如清单 7.14 中所示。在接收到关闭请求后, 消费者会离开队列, 写出所有等待中的消息, 并将 log 中所有阻塞的生产者解除阻塞。但是这个方案存在竞争条件, 这使它并不可靠。log 的实现是一个“检查再运行”序列: 生产者观察没有被关闭的服务, 但即便关闭后, 仍然会把消息放入队列, 同样地, 这里存在这样的风险, 生产者在 log 中被阻塞, 却不能解脱出来。有一些窍门帮助我们减少可能性(比如让消费者在离开队列后, 过几秒再发出声明), 但是这些不能解决最基本的问题, 很小的可能性也会导致失败。

清单 7.14 向日志服务添加不可靠的关闭支持

```
public void log(String msg) throws InterruptedException {  
    if (!shutdownRequested)  
        queue.put(msg);  
    else  
        throw new IllegalStateException("logger is shut down");  
}
```



为 LogWriter 提供可靠的关闭方法是解决竞争条件, 这意味着创建新日志消息的各个子任务必须是原子的。但是我们不希望在消息加入队列时加锁, 因为 put 方法可能发生阻塞。我们能够做的事情是原子化地检查关闭, 并有条件地递增记录获得提交消息权利的计数器, 正如清单 7.15 中 LogService 所示。

7.2.2 关闭 ExecutorService

在 6.2.4 节中, 我们看到 ExecutorService 提供了关闭的两种方法: 使用 shutdown 优雅的关闭, 和使用 shutdownNow 强行的关闭。在强行关闭中, shutdownNow 首先尝试关闭当前正在执行的任务, 然后返回待完成任务的清单。

清单 7.15 向 LogWriter 添加可靠的取消

```
public class LogService {
    private final BlockingQueue<String> queue;
    private final LoggerThread loggerThread;
    private final PrintWriter writer;
    @GuardedBy("this") private boolean isShutdown;
    @GuardedBy("this") private int reservations;

    public void start() { loggerThread.start(); }

    public void stop() {
        synchronized (this) { isShutdown = true; }
        loggerThread.interrupt();
    }

    public void log(String msg) throws InterruptedException {
        synchronized (this) {
            if (isShutdown)
                throw new IllegalStateException(...);
            ++reservations;
        }
        queue.put(msg);
    }

    private class LoggerThread extends Thread {
        public void run() {
            try {
                while (true) {
                    try {
                        synchronized (LogService.this) {
                            if (isShutdown && reservations == 0)
                                break;
                        }
                        String msg = queue.take();
                        synchronized (LogService.this) { --reservations; }
                        writer.println(msg);
                    } catch (InterruptedException e) { /* 重试 */ }
                }
            } finally {
                writer.close();
            }
        }
    }
}
```

两种不同的终结选择在安全性和响应性之间进行了权衡：强行终结的速度更快，但是风险大，因为任务很可能在执行到一半的时候被终结，而正常终结虽然速度慢，却安全，因为直到队列中的所有任务完成前，`ExecutorService` 都不会关闭。其他拥有线程的服务应该考虑提供类似的关闭模式以供选择。

简单的程序可以避免在 `main` 函数中启动和关闭一个全局 `ExecutorService`。更复杂的程序很可能把 `ExecutorService` 封装于一个高层级的服务中，在其中提供自己的生命周期方法，比如清单 7.16 中的变量 `LogService`，它由 `ExecutorService` 进行代理。`ExecutorService` 不会管理 `LogService` 拥有的线程。封装 `ExecutorService` 通过增加链接，把所有权链从应用程序扩展到了服务，再到线程；每一个链上的成员管理它拥有的服务或线程的生命周期。

清单 7.16 使用 `ExecutorService` 的日志服务

```
public class LogService {
    private final ExecutorService exec = newSingleThreadExecutor();
    ...
    public void start() { }

    public void stop() throws InterruptedException {
        try {
            exec.shutdown();
            exec.awaitTermination(TIMEOUT, UNIT);
        } finally {
            writer.close();
        }
    }
    public void log(String msg) {
        try {
            exec.execute(new WriteTask(msg));
        } catch (RejectedExecutionException ignored) { }
    }
}
```

7.2.3 致命药丸

另一种保证生产者和消费者服务关闭的方式是使用致命药丸 (poison pill)：一个可识别的对象，置于队列中，意味着“当你得到它时，停止一切工作”。在先进先出队列中，致命药丸保证了消费者完成队列中关闭之前的所有工作，因为所有早于致命药丸提交的工作都会在处理它之前就完成了，生产者不应该在提交了致命药丸后，再提交任何工作。清

清单 7.17 使用致命药丸来关闭

```
public class IndexingService {
    private static final File POISON = new File("");
    private final IndexerThread consumer = new IndexerThread();
    private final CrawlerThread producer = new CrawlerThread();
    private final BlockingQueue<File> queue;
    private final FileFilter fileFilter;
    private final File root;

    class CrawlerThread extends Thread { /* 清单 7.18 */ }
    class IndexerThread extends Thread { /* 清单 7.19 */ }

    public void start() {
        producer.start();
        consumer.start();
    }

    public void stop() { producer.interrupt(); }

    public void awaitTermination() throws InterruptedException {
        consumer.join();
    }
}
```

单 7.17、7.18 和 7.19 表现了单个生产者，单个消费者的 Desktop Search 的例子（第 91 页，清单 5.8），这里使用致命药丸来关闭服务。

致命药丸只有在生产者和消费者数量已知的情况下使用。IndexingService 中的解决方案可以被扩展到多生产者，只要让每一个生产者都向队列置入一个药丸，并且消费者在接收到第 $N_{\text{producers}}$ （生产者数量）个药丸时停止。这个方法同样也可以扩展为多个消费者使用，只要让生产者向队列置入 $N_{\text{consumers}}$ （消费者数量）个药丸，不过这在生产者和消费者的数量较大时很难处理。致命药丸只有在无限队列中工作时，才是可靠的。

7.2.4 示例：只执行一次的服务

如果一个方法需要处理一批任务，并在所有任务结束前不会返回，那么它可以通过使用私有的 Executor 来简化服务的生命周期管理，其中 Executor 的寿命限定在该方法中（在这种情况下，通常会用到 invokeAll 和 invokeAny 方法）。

清单 7.20 中的 checkMail 方法同时在多个服务器上并行地检查新邮件。它创建一个

清单 7.18 IndexingService 的生产者线程

```
public class CrawlerThread extends Thread {
    public void run() {
        try {
            crawl(root);
        } catch (InterruptedException e) { /* 失败 */ }
        finally {
            while (true) {
                try {
                    queue.put(POISON);
                    break;
                } catch (InterruptedException e1) { /* 重试 */ }
            }
        }
    }

    private void crawl(File root) throws InterruptedException {
        ...
    }
}
```

清单 7.19 IndexingService 的消费者线程

```
public class IndexerThread extends Thread {
    public void run() {
        try {
            while (true) {
                File file = queue.take();
                if (file == POISON)
                    break;
                else
                    indexFile(file);
            }
        } catch (InterruptedException consumed) { }
    }
}
```

私有的 Executor，并向每一个主机提交任务：在这之后，当所有检查邮件的任务完成后，它会关闭 Executor，并等待结束⁴。

清单 7.20 使用私有 Executor，将它的寿命限定于一次方法调用中

```
boolean checkMail(Set<String> hosts, long timeout, TimeUnit unit)
    throws InterruptedException {
    ExecutorService exec = Executors.newCachedThreadPool();
    final AtomicBoolean hasNewMail = new AtomicBoolean(false);
    try {
        for (final String host : hosts)
            exec.execute(new Runnable() {
                public void run() {
                    if (checkMail(host))
                        hasNewMail.set(true);
                }
            });
    } finally {
        exec.shutdown();
        exec.awaitTermination(timeout, unit);
    }
    return hasNewMail.get();
}
```

7.2.5 shutdownNow 的局限性

当通过 shutdownNow 强行关闭一个 ExecutorService 时，它试图取消正在进行的任务，并返回那些已经提交、但并没有开始的任务的清单，这样，这些任务可以被日志记录，或者存起来等待进一步处理⁵。

然而，我们并没有任何常规的方法，用于找出那些已经开始、却没有结束的任务。这意味着，我们不可能在关闭时知道进行中的任务的状态，除非任务本身设立了某些检查点。为了得知哪些任务没有结束，你不仅需要知道哪些任务还没有开始，而且应该知道 Executor 关闭时，哪些任务正在进行中⁶。

清单 7.21 中 TrackingExecutor 展现了如何在关闭过程中判定哪些任务还在进行的技术。通过封装 ExecutorService 并使用 execute（类似的，还有 submit，不过没有显

⁴ 使用 AtomicBoolean 取代 volatile boolean 的原因是：为了从内部 Runnable 访问 hasNewMail 标志，那么它必须是 final 类型的，这样才能避免被更改。

⁵ shutdownNow 返回的 Runnable 对象可能并不是提交给 ExecutorService 的相同对象：它们可能是经过包装的已提交任务的实例。

⁶ 不幸的是，关闭选择只会把那些还没有开始的任務返回给调用者，但是却不支持把正在进行的任务执行完成，如果有这样的支持，会大大削减这些不确定的中间状态。

示)，来记录哪些任务是在关闭后取消的。TrackingExecutor 可以识别哪些任务已经开始，却没能正常结束。在 executor 结束后，getCancelledTasks 返回被取消的任务清单。为应用这个技术，任务必须在返回时保存线程的中断状态，这是运行良好的任务无论如何应该完成的。

清单 7.21 关闭之后，ExecutorService 获取被取消的任务

```
public class TrackingExecutor extends AbstractExecutorService {
    private final ExecutorService exec;
    private final Set<Runnable> tasksCancelledAtShutdown =
        Collections.synchronizedSet(new HashSet<Runnable>());
    ...
    public List<Runnable> getCancelledTasks() {
        if (!exec.isTerminated())
            throw new IllegalStateException(...);
        return new ArrayList<Runnable>(tasksCancelledAtShutdown);
    }

    public void execute(final Runnable runnable) {
        exec.execute(new Runnable() {
            public void run() {
                try {
                    runnable.run();
                } finally {
                    if (isShutdown()
                        && Thread.currentThread().isInterrupted())
                        tasksCancelledAtShutdown.add(runnable);
                }
            }
        });
    }

    // 将 ExecutorService 中的其他方法委托到 exec
}
```

在清单 7.22 中，WebCrawler 展现了 TrackingExecutor 的应用。Web Crawler 的工作通常都是无穷尽的，所以，如果 Crawler 必须关闭的时候，我们可能希望保存它的状态，这样它可以稍后被重新启动。CrawlTask 方法提供了一个 getPage 方法，这个方法识别了 Crawler 正在哪一个页面工作。当 Crawler 关闭后，无论是还没有开始的任務，还是那些被取消的任务，都会被检查并记录下 URL，当 Crawler 重新启动时，这些 URL 的 page-crawling 任务就可以加入到队列了。

TrackingExecutor 存在不可避免的竞争条件，使它产生假阳性（false positive）现象：识别出的被取消任务事实上可能已经结束。产生的原因是在任务执行的最后一条指令，以及线程池记录任务结束之间，线程池可能发生关闭。如果任务是幂等的（idempotent，如

清单 7.22 使用 TrackingExecutorService 为后续执行来保存未完成任务

```
public abstract class WebCrawler {
    private volatile TrackingExecutor exec;
    @GuardedBy("this")
    private final Set<URL> urlsToCrawl = new HashSet<URL>();
    ...
    public synchronized void start() {
        exec = new TrackingExecutor(
            Executors.newCachedThreadPool());
        for (URL url : urlsToCrawl) submitCrawlTask(url);
        urlsToCrawl.clear();
    }

    public synchronized void stop() throws InterruptedException {
        try {
            saveUncrawled(exec.shutdownNow());
            if (exec.awaitTermination(TIMEOUT, UNIT))
                saveUncrawled(exec.getCancelledTasks());
        } finally {
            exec = null;
        }
    }

    protected abstract List<URL> processPage(URL url);

    private void saveUncrawled(List<Runnable> uncrawled) {
        for (Runnable task : uncrawled)
            urlsToCrawl.add(((CrawlTask) task).getPage());
    }
    private void submitCrawlTask(URL u) {
        exec.execute(new CrawlTask(u));
    }
    private class CrawlTask implements Runnable {
        private final URL url;
        ...
        public void run() {
            for (URL link : processPage(url)) {
                if (Thread.currentThread().isInterrupted())
                    return;
                submitCrawlTask(link);
            }
        }
        public URL getPage() { return url; }
    }
}
```

果执行两次得到的结果与执行一次相同)，那么这不会有什么问题，典型性的 Web Crawler 就是这样。另一方面，应用程序得到已被取消的任务必须注意这个风险，应该为这样的假阳性现象作好准备。

7.3 处理反常的线程终止

当一个单线程化的控制台程序因为未捕获的异常终止的时候，程序停止运行，并产生了栈追踪，这与典型的程序输出有很大的不同——这是很明显的。并发程序中线程的失败往往就没有这么明显了。栈追踪可能会从控制台输出，但是没有人会去观察控制台，并且，当线程失败的时候，应用程序可能看起来仍在工作，所以它的失败可能就会被忽略。幸运的是，我们有方法可以监测和防止线程从程序中“泄漏”。

导致线程死亡的最主要原因是 `RuntimeException`。因为这些异常表明一个程序错误或者其他不可修复的错误，它们通常是不能被捕获的。它们不会顺着栈的调用传递，此时，默认的行为是在控制台打印栈追踪的信息，并终止线程。

线程非正常退出的后果包括良性的与恶性的，取决于线程在应用程序中的角色，但是程序能跑在 50 个线程的线程池上，通常也能够安全地跑在 49 个线程的线程池上。然而在 GUI 程序中，失去分派事件的线程会非常显著——应用程序会停止处理事件，GUI 会被冻结。第 124 页的 `OutOfTime` 展现了一种严重的线程泄漏的后果：`Timer` 代表的服务永远不能使用了。

任何代码都可以抛出 `RuntimeException`。无论何时，当你调用另一个方法，你都要对它的行为保持怀疑，不要盲目地认为它一定会正常返回，或者一定会抛出在方法签名中声明的受检查的异常。对你调用的代码越不熟悉，你就越应该坚持对代码行为的怀疑。

任务处理线程，比如线程池中的工作者线程或者 `Swing` 的事件派发线程，它们生命周期中调用的所有代码都是通过抽象关卡（`abstraction barrier`，比如 `Runnable`）调用的未知代码，这些线程应该受到质疑，质疑它们调用的代码是否都能良好地运行。`Swing` 的事件线程可能仅仅因为个别不好的事件处理器抛出 `NullPointerException` 而失败，如果整个服务因为这个问题失败，就非常不值得了。因此，这些线程应该在 `try-catch` 块中调用这些任务，这样就能捕获那些未检查的异常了，或者也可以使用 `try-finally` 块来确保框架能够知晓线程的非正常退出，并作出正确的反应。这是为数不多的几次，你需要考虑捕获 `RuntimeException`——当你通过 `Runnable` 这样的抽象体（`abstraction`）调用未知、不可信的代码时⁷。

⁷ 对于这个技术的安全性是有一些争议的；当线程抛出一个未检查的异常时，整个应用程序的安全都是受到威胁的。但是从另一个角度上看——关闭整个应用程序——通常是不可取的。

清单 7.23 阐述了如何在线程池内部构建一个工作者线程。如果任务抛出了一个未检查的异常，它将允许线程终结，但是会首先通知框架：线程已经终结。然后，框架可能会用新的线程取代这个工作线程，也可能不这样做，因为线程池也许正在关闭，抑或当前已有足够多的线程，能够满足需要了。ThreadPoolExecutor 和 Swing 使用这项技术来确保那些不能正常运转的任务不会影响到后续任务的执行。如果你正在写一个工作者线程类，它会向线程池提交任务，或者调用不可信的外部代码（比如动态载入的插件），使用其中一种解决方案可以避免不好的任务或插件牵连到调用它的整个线程。

清单 7.23 典型线程池的工作者线程的构建

```
public void run() {
    Throwable thrown = null;
    try {
        while (!isInterrupted())
            runTask(getTaskFromWorkQueue());
    } catch (Throwable e) {
        thrown = e;
    } finally {
        threadExited(this, thrown);
    }
}
```

7.3.1 未捕获异常的处理

前面的小节讲述了一种主动解决未检查异常问题的方案。线程的 API 同样提供了 UncaughtExceptionHandler 的工具，使你能够监测到线程因未捕获的异常引起的“死亡”。这两个方案互为补充：合在一起，组成了对抗线程泄漏的强有力的保障。

当一个线程因为未捕获异常而退出时，JVM 会把这个事件报告给应用程序提供的 UncaughtExceptionHandler（见清单 7.24）；如果处理器（handler）不存在，默认的行为是向 System.err 打印出栈追踪信息⁸。

⁸ 在Java5.0之前，操控UncaughtExceptionHandler唯一的方法是子类化ThreadGroup。在Java 5.0以及之后版本的JDK中，你可以通过Thread.setUncaughtExceptionHandler为每个线程设置一个UncaughtExceptionHandler；也可以使用setDefaultUncaughtExceptionHandler来设置默认的UncaughtExceptionHandler。然而，只有其中一个处理器能够被调用——JVM首先寻找针对每个线程的处理器，然后再查找ThreadGroup的。ThreadGroup中，默认的处理器的实现会委托给它们的父线程组，并且会依次向上传递委托，直到其中一个ThreadGroup的处理器能够处理未捕获的异常，或者用冒泡的方式传递到更高层的线程组。高层的线程组处理器委托给默认的系统处理器（如果有的话；默认是空），否则向控制台打印栈的追踪信息。

清单 7.24 UncaughtExceptionHandler 接口

```
public interface UncaughtExceptionHandler {  
    void uncaughtException(Thread t, Throwable e);  
}
```

如何处理未捕获异常取决于对服务质量的需求。最常见的响应是记录一个错误信息，并把栈追踪信息写入应用程序日志中，正如清单 7.25 所示。处理器也可以进行更直接的反应，比如尝试重新启动线程，关闭应用程序，对操作分页，或者其他纠正行为，或诊断行为。

清单 7.25 UncaughtExceptionHandler 将异常写入日志

```
public class UEHLogger implements Thread.UncaughtExceptionHandler {  
    public void uncaughtException(Thread t, Throwable e) {  
        Logger logger = Logger.getAnonymousLogger();  
        logger.log(Level.SEVERE,  
            "Thread terminated with exception: " + t.getName(),  
            e);  
    }  
}
```

在一个长时间运行的应用程序中，所有的线程都要给未捕获异常设置一个处理器，这个处理器至少要将异常信息记入日志中。

为了给线程池设置 UncaughtExceptionHandler，需要向 ThreadPoolExecutor 的构造函数提供一个 ThreadFactory（正如在所有线程的操控中，只有线程的所有者能够改变其 UncaughtExceptionHandler）。标准线程池允许未捕获的任务异常去结束池线程，但是使用一个 try-finally 块来接收通知的话，当池线程被终结后，能够有新的线程取代它。如果没有非捕获异常的处理器，或者其他失败通知机制，任务会无声无息地失败，这会导致混乱。如果你想在任务因为异常而失败时获得通知，那么你应该采取一些特定任务的恢复行为，或者用 Runnable 与 Callable 把任务包装起来，这样就能够捕获异常，或者覆写 ThreadPoolExecutor 的 afterExecute 钩子方法。

令人有些混淆的是，只有通过 execute 提交的任务，才能将它抛出的异常送交给未捕获异常的处理器；而通过 submit 提交的任务，抛出的**任何**异常，无论是否为受检查的，都被认为是任务返回状态的一部分。如果一个有 submit 提交的任务以异常作为终结，这个异常会被 Future.get 重抛出，包装在 ExecutionException 中。

7.4 JVM 关闭

JVM 既可以通过正常手段关闭，也可以强行关闭。当最后一个“正常（非精灵）”线程终结时，或者当有人调用了 `System.exit` 时，以及通过使用其他与平台相关手段时（比如发送了 `SIGINT`，或键入 `Ctrl-C`），都可以开始一个正常的关闭。尽管 JVM 可以通过这些标准的首选方法关闭，它仍然能够通过调用 `Runtime.halt` 或者“杀死”JVM 的操作系统进程被强行关闭（比如发送 `SIGKILL`）。

7.4.1 关闭钩子

在正常的关闭中，JVM 首先启动所有已注册的 *Shutdown hook*。关闭钩子（Shutdown hook）是使用 `Runtime.addShutdownHook` 注册的尚未开始的线程。JVM 并不能保证关闭钩子的开始顺序。如果关闭应用程序线程（精灵或非精灵）时，它仍然在运行，它们接下来将与关闭进程并发执行。当所有关闭钩子结束的时候，如果 `runFinalizersOnExit` 为 `true`，JVM 可以选择运行 `finalizer`，之后停止。JVM 不会尝试停止或中断任何关闭时仍然在运行中的应用程序线程；它们在 JVM 最终终止时被强制退出。如果关闭钩子或 `finalizer` 没有完成，那么正常的关闭进程“挂起”并且 JVM 必须强行关闭。在强行关闭中，JVM 不需要完成除了关闭 JVM 以外的任何事情；不会运行关闭钩子。

关闭钩子应该是线程安全的：它们在访问共享数据时必须使用同步，并应该小心地避免死锁，这与其他并发代码是相同的。进一步而言，它们不仅需要假设应用程序的状态（比如其他服务是否已经关闭，或者正常的线程是否已经完成任务）或者关于 JVM 为何关闭，因此在代码过程中必须格外小心。最后，在用户可能希望 JVM 快速终止的情况下，它们必须尽快退出，因为它们的存在会延迟 JVM 的终止。

关闭钩子可以用于服务或应用程序的清理，比如删除临时文件，或者清除 OS 不能自动清除的资源。清单 7.26 表现了清单 7.16 中 `LogService` 如何从它的 `start` 方法注册一个关闭钩子，来确保它的日志文件在退出时关闭。

因为关闭钩子全部都是并发执行的，关闭日志文件可能引起其他需要使用日志服务的关闭钩子的麻烦。为了避免这个问题，关闭钩子不应该依赖于可能被应用程序或其他关闭钩子关闭的服务。实现它的一种方式是对所有服务使用唯一关闭钩子，让它调用一系列关闭行为，而不是每个服务使用一个。这确保了关闭的动作在单线程中顺序发生，因此避免

清单 7.26 注册关闭钩子来停止日志服务

```
public void start() {
    Runtime.getRuntime().addShutdownHook(new Thread() {
        public void run() {
            try { LogService.this.stop(); }
            catch (InterruptedException ignored) {}
        }
    });
}
```

了竞争条件的出现，或关闭动作之间的死锁；顺序地而非并发地执行关闭动作，可以消除许多潜在的资源失败。当某个应用程序要维护服务间显式依赖信息时，这个技术可以保证关闭动作按照正确的顺序进行。

7.4.2 精灵线程

有时你想要创建一个线程，执行一些辅助工作，但是你不希望这个线程的存在阻碍 JVM 的关闭。这时你需要用到**精灵线程**（daemon thread）。

线程被分为两种：普通线程和精灵线程。JVM 启动时创建所有的线程，除了主线程以外，其他的都是精灵线程（比如垃圾回收器和其他类似线程）。当一个新的线程创建时，新线程继承了创建它的线程的后台状态，所以默认情况下，任何主线程创建的线程都是普通线程。

普通线程和精灵线程之间的差别仅仅在于退出时会发生什么。当一个线程退出时，JVM 会检查一个运行中线程的详细清单，如果只剩下精灵线程，它会发起正常的退出。当 JVM 停止时，所有仍然存在的精灵线程都会被抛弃——不会执行 `finally` 块，也不会释放栈——JVM 直接退出。

精灵线程应该小心使用——在任何时候，几乎没有哪些活动的处理可以在不进行清理的情况下，被安全地抛弃。特别是执行 I/O 操作的任务运行在精灵线程中是很危险的。精灵线程最好用于“家务管理（housekeeping）”的任务，比如一个背景线程可以从内存的缓存中周期性地移除过期的访问。

应用程序中，精灵线程不能替代对服务的生命周期恰当、良好的管理。

7.4.3 Finalizer

当我们已经不再需要资源后，垃圾回收器重新获得内存资源是非常有益的，但是一些资源，比如文件或者 Socket 句柄，当我们不再需要时，必须显式地归还给操作系统。为了

在这方面提供帮助，垃圾回收器对那些具有特殊 `finalize` 方法的对象会进行特殊对待：在回收器获得它们后，`finalize` 被调用，这样就能保证持久化的资源可以被释放。

因为 `finalizer` 可以运行在一个 JVM 管理的线程中，任何 `finalizer` 访问的状态都会被多个线程访问，因此必须被同步。`finalizer` 运行时不提供任何保证，并且拥有复杂的 `finalizer` 会带来对象巨大的性能开销。正确的书写 `finalizer` 也十分困难⁹。在大多数情况下，使用 `finally` 块和显式 `close` 方法的结合来管理资源，会比使用 `finalizer` 起到更好的作用。当你需要管理对象，并且这个对象持有的资源是通过本地方法获得的，这时会产生独特的异常。鉴于这些原因，和其他一些原因，应努力的避免编写或者使用包含 `finalizer` 的类（除非是在平台库的类中）[EJ Item 6]。

避免使用 `finalizer`。

总结

任务、线程、服务以及应用程序在生命周期结束时的的问题，可能会导致向它们引入复杂的设计和实现。Java 没有提供具有明显优势的机制来取消活动或者终结线程。它提供了协作的中断机制，能够用来帮助取消，但是这将取决于你如何构建取消的协议，并是否能一致地使用该协议。使用 `FutureTask` 和 `Executor` 框架可以简化构建可取消的任务和服务。

⁹ 请参阅 (Boehm, 2005)，以获得更多涉及 `finalize` 的挑战。

应用线程池

Applying Thread pools

第 6 章介绍的任务执行框架，可以简化任务与线程生命周期的管理。它提供一种简便、灵活的方式，可以在任务的提交与任务的执行策略之间解耦。第 7 章包含一些繁杂的细节内容，包括了在真实的应用程序中使用任务执行框架的生命周期服务。本章关注在配置和调整线程池时用的高级选项，讲述了使用任务执行框架的过程中需要注意的危险，还提供了一些使用 `Executor` 更高级的例子。

8.1 任务与执行策略间的隐性耦合

前面我们声称 `Executor` 框架可以将任务的提交与执行策略解耦。如同很多对复杂过程解耦的尝试一样，这或多或少有些言过其实了。尽管 `Executor` 框架为制定和修改执行策略提供了相当大的灵活性，但是并非所有的任务都能适合所有的执行策略。有些类型的任务需要明确地指定一个执行策略，包括：

依赖性任务。多数运行良好的任务是独立的：它们不依赖于时序，或者其他任务的结果与边界效应。当线程池中运行的任务都是独立的时候，你就可以随意地改变池的长度和配置，这样做不会影响到性能以外的任何事情。另一方面，如果你提交给线程池的任务要依赖于其他的任务，你就隐式地给执行策略带来了约束，这样你必须仔细地管理执行策略以避免活跃度的问题（参见 8.1.1 节）。

采用线程限制的任务。单线程化的 `Executor` 相比于任意的线程池，可以对同步作出更强的承诺。它可以保证任务不会并发地执行，允许你放宽任务代码对线程安全的要求。可以把对象限制在任务线程中，这使得任务线程所执行的任务在访问该对象时，不需要同步，即使那些资源不是线程安全的亦如此。这会在任务与执行策略之间形成隐式的耦合——

任务需要它们的 `Executor` 来确保单线程化¹。如果你将 `Executor` 从一个单线程化的改为一个线程池的话，就会失去线程安全性。

对响应时间敏感的任务。GUI 应用程序对于响应时间是敏感的：如果用户点击按钮后需要很长的延迟才能得到可视的反馈，他们会对此感到烦恼。将一个长时间运行的任务提交到单线程化的 `Executor` 中，或者将多个长时间运行的任务提交给一个只包含少量线程的线程池中，会削弱由 `Executor` 管理的服务的响应性。

使用 `ThreadLocal` 的任务。`ThreadLocal` 让每个线程可以保留一份变量的私有“版本”。但是，只要条件允许，`Executor` 就会随意重用这些线程。标准的 `Executor` 实现是：在需求不高时回收空闲的线程，在需求增加时添加新的线程，如果任务抛出了异常，就会用一个全新的工作者线程取代出错的那个。只有当线程本地（`thread-local`）值的生命周期被限制在当前的任务中时，在池的某线程中使用 `ThreadLocal` 才有意义；在线程池中，不应该用 `ThreadLocal` 传递任务间的数值。

当任务都是同类的、独立的时候，线程池才会有最佳的工作表现。如果将耗时的与短期的任务混合在一起，除非线程池很大，否则会有“塞车”的风险；如果提交的任务要依赖于其他的任务，除非池是无限的，否则有产生死锁的风险。幸运的是，那些典型的基于网络的服务器应用程序——Web 服务器、邮件服务器、文件服务器——它们的请求通常都是同类的、独立的。

一些任务具有这样的特征：需要或者排斥某种特定的执行策略。对其他任务具有依赖性的任务，就会要求线程池足够大，来保证它所依赖任务不必排队或者不被拒绝；采用线程限制的任务需要顺序地执行。把这些需求都写入文档，这样将来的维护者就不会使用一个与原先相悖的执行策略，而破坏安全性或活跃度了。

8.1.1 线程饥饿死锁

在线程池中如果一个任务依赖于其他任务的执行，就可能产生死锁。对于一个单线程化的 `Executor`，一个任务将另一个任务提交到相同的 `Executor` 中，并等待新提交的任务的结果，这总会引发死锁。第二个任务滞留在工作队列中，直到第一个任务完成，但是第一个任务不会完成，因为它在等待第二个任务的完成。在一个大的线程池中，如果所有线程

¹ 这个要求不一定这么严格；只需要保证任务不会并发执行，并提供充分的同步，这样一个任务对内存的影响，肯定对下一个任务可见——这刚好是 `newSingleThreadExecutor` 提供的担保。

执行的任务都阻塞在线程池中，等待着仍然处于同一工作队列中的其他任务，那么会发生同样的问题。这被称作**线程饥饿死锁**（thread starvation deadlock），满足以下叙述就会发生：只要池任务开始了无限期的阻塞，其目的是等待一些资源或条件，此时只有另一个池任务的活动才能使那些条件成立，比如等待返回值或者另一个任务的边界效应。除非你能保证这个池足够大，否则会发生线程饥饿死锁。

清单 8.1 的 ThreadDeadlock 演示了线程饥饿死锁。RenderPageTask 向 Executor 提交了两个附加的任务，来完成获取页眉、页脚、渲染页面、等待获取页眉和页脚任务的结果，以及将页眉、页面主体和页脚结合到最终页面中这一系列的工作。作为一个单线程化的 Executor，ThreadDeadlock 会经常死锁。类似地，如果池不够足够大，所有任务在它们自身进行协调时会遇到关卡（barrier），这也会引起线程饥饿死锁。

无论何时，你提交了一个非独立的 Executor 任务，要明确出现线程饥饿死锁的可能性，并且，在代码或者配置文件以及其他可以配置 Executor 的地方，任何有关池的大小和配置约束都要写入文档。

除了显性限制，可能因为其他资源的约束存在一些隐性限制，这源自于其他资源的约束。如果你的应用程序使用一个包含 10 个连接的 JDBC 连接池，每个任务需要一个数据库连接，这样你的线程池中就好像只有 10 个线程一样，因为超过 10 个的任务将要等待一个连接。

清单 8.1 在单线程化的 Executor 中死锁的任务（不要这样做）

```
public class ThreadDeadlock {
    ExecutorService exec = Executors.newSingleThreadExecutor();

    public class RenderPageTask implements Callable<String> {
        public String call() throws Exception {
            Future<String> header, footer;
            header = exec.submit(new LoadFileTask("header.html"));
            footer = exec.submit(new LoadFileTask("footer.html"));
            String page = renderBody();
            // 出现死锁 - 任务等待子任务的结果
            return header.get() + page + footer.get();
        }
    }
}
```



8.1.2 耗时操作

如果任务由于过长的时间周期而阻塞，那么即使不可能出现死锁，线程池的响应性也会变得很差。耗时任务会造成线程池堵塞，还会拖长服务时间，即使小任务也不能幸免。耗时操作的数目会期望线程有一个稳定的数量，如果线程池的大小相对于这个数字来说太小，那么最后可能所有的线程都会处于运行耗时任务的状态中，从而就会影响响应性。

有一项技术可以用来缓解耗时操作带来的影响，这就是限定任务等待资源的时间，而不要无限制地等下去。大多数平台类库中的阻塞方法，都同时有限时的和非限时两个版本，比如 `Thread.join`、`BlockingQueue.put`、`CountDownLatch.await` 和 `Selector.select`。如果等待超时，你可以把任务标识为失败，中止它或者把它重新放回队列，准备之后执行。这样，无论每个任务的最终结果是成功还是失败，该办法都保证了任务总会向前发展，这样可以更快地将线程从任务中解放出来。如果线程池频频被阻塞的任务充满，这同样也可能是池太小的一个信号。

8.2 定制线程池的大小

线程池合理的长度取决于未来提交的任务类型和所部署系统的特征。很少会有人把线程池的长度硬编码；池的长度应该由某种配置机制来提供，或者利用 `Runtime.availableProcessors` 的结果，动态地进行计算。

幸运的是，定制线程池的长度并不是一门精密科学，你需要做的仅仅是避免“过大”和“过小”这两种极端情况。如果一个线程池过大，那么线程对稀缺的 CPU 和内存资源的竞争，会导致内存的高使用量，还可能耗尽资源。如果过小，由于存在很多可用的处理器资源却未在工作，会对吞吐量造成损失。

为了正确地定制线程池的长度，你需要理解你的计算环境、资源预算和任务的自身特性。部署系统中安装了多少个 CPU？多少内存？任务主要执行的是计算、I/O 还是一些混合操作？它们是否需要像 `JDBC Connection` 这样的稀缺资源？如果你有不同类别的任务，它们拥有差别很大行为，那么请考虑使用多个线程池，这样每个线程池可以根据不同任务的工作负载进行调节。

对于计算密集型的任务，一个有 N_{cpu} 个处理器的系统通常通过使用一个 $N_{cpu}+1$ 个线程的线程池来获得最优的利用率（计算密集型的线程恰好在某时因为发生一个页错误或者其他原因而暂停，刚好有一个“额外”的线程，可以确保在这种情况下 CPU 周期不会中断工作）。对于包含了 I/O 和其他阻塞操作的任务，不是所有的线程都会在所有的时间被调度，因此你需要一个更大的池。为了正确地设置线程池的长度，你必须估算出任务花在等待的时间与用来计算的时间的比率；这个估算值不必十分精确，而且可以通过一些

监控工具获得。你还可以选择另一种方法来调节线程池的大小，在一个基准负载下，使用几种不同大小的线程池运行你的应用程序，并观察 CPU 利用率的水平。

给定下列定义：

N_{cpu} = CPU 的数量

U_{cpu} = 目标 CPU 的使用率, $0 \leq U_{cpu} \leq 1$

W/C = 等待时间与计算时间的比率

为保持处理器达到期望的使用率，最优的池的大小等于：

$$N_{threads} = N_{cpu} \times U_{cpu} \times (1 + W/C)$$

你可以使用 Runtime 来获得 CPU 的数目：

```
int N_CPUS = Runtime.getRuntime().availableProcessors();
```

当然，CPU 周期并不是唯一你可以使用线程池管理的资源。其他可以约束资源池大小的资源包括：内存、文件句柄、套接字句柄和数据库连接等。计算这些类型资源池的大小约束非常简单：首先累加出每一个任务需要的这些资源的总量，然后除以可用的总量。所得的结果是池大小的上限。

当任务需要使用池化的资源时，比如数据库连接，那么线程池的长度和资源池的长度会相互影响。如果每一个任务都需要一个数据库连接，那么连接池的大小就限制了线程池的有效大小；类似地，当线程池中的任务是连接池的唯一消费者时，那么线程池的大小反而又会限制了连接池的有效大小。

8.3 配置 ThreadPoolExecutor

ThreadPoolExecutor 为一些 Executor 提供了基本的实现，这些 Executor 是由 Executors 中的工厂 newCachedThreadPool、newFixedThreadPool 和 newScheduledThreadPool 返回的。ThreadPoolExecutor 是一个灵活的、健壮的池实现，允许各种各样的用户定制。

如果默认的执行策略不能满足你的需要，你可以通过构造函数实例化一个 ThreadPoolExecutor，自己定制它直到你满意为止；你可以参考 Executors 的源代码是如何实现默认配置的执行策略的，然后以它们为起点开始你自己的实现。ThreadPoolExecutor 有很多个构造函数，其中清单 8.2 中的是最通用的一个。

8.3.1 线程的创建与销毁

核心池大小(core pool size)、最大池的大小(maximum pool size)和存活时间(keep-alive time)共同管理着线程的创建与销毁。核心池大小是目标的大小；线程池的实现试图维护池的大小：即使没有任务执行²，池的大小也等于核心池的大小，并且直到工作队列充满前，

² 当一个 ThreadPoolExecutor 被初始创建后，所有核心线程并非立即开始，而是要等到有任务提交的时刻，除非你调用 prestartAllCoreThreads。

清单 8.2 ThreadPoolExecutor 通用的构造函数

```
public ThreadPoolExecutor(int corePoolSize,
                          int maximumPoolSize,
                          long keepAliveTime,
                          TimeUnit unit,
                          BlockingQueue<Runnable> workQueue,
                          ThreadFactory threadFactory,
                          RejectedExecutionHandler handler) { ... }
```

池都不会创建更多的线程³。最大池的大小是可同时活动的线程数的上限。如果一个线程已经闲置的时间超过了存活时间，它将成为一个被回收的候选者，如果当前的池的大小超过了核心池的大小，线程池会终止它。

通过调节核心大小和存活时间，可以促进线程池归还空闲线程占有的资源，让这些资源可以用于更有用工作（让我们习以为常的是，这是一种折中：回收空闲线程的行为会招致额外的延迟，而需求增加后，必须重新创建回收的线程）。

`newFixedThreadPool` 工厂为请求的池设置了核心池的大小和最大池的大小，而且池永远不会超时；`newCachedThreadPool` 工厂将最大池的大小设置为 `Integer.MAX_VALUE`，核心池的大小设置为零，超时设置为一分钟，这样创建出来的可无限扩大的线程池，会在需求量减少的情况下减少线程数量（译注：因为多余的线程会被回收）。其他的组合可以使用显式的 `ThreadPoolExecutor` 构造函数来实现。

8.3.2 管理队列任务

有限线程池限制了可以并发执行的任务的数量（单线程化的 `Executor` 是一个值得注意的特例：它们保证没有并发执行的任务，通过线程限制，提供了获得线程安全的可能性）。

在 6.1.2 节我们看到无限制地创建线程是如何导致不稳定性的，然后用一个定长的线程池取代每个请求都创建一个新线程的做法，我们解决了这个问题。但是，这只是个片面的方案；在高负载下，应用程序仍然可能耗尽资源，只不过更难了。如果新请求到达的频率超过了线程池能够处理它们的速度，请求将在队列中等待。有了线程池，这些多出来的请求会在一个由 `Executor` 管理的 `Runnable` 队列中等待，而不是作为竞争 CPU 资源的线

³ 开发者有时会禁不住诱惑要把核心池的大小设置为零，认为这样工作者线程最终会销毁，因此不会妨碍 JVM 的退出。但是，这会在没有使用 `SynchronousQueue` 作为工作队列的线程池中（如 `newCachedThreadPool` 就是如此），引起一些看似奇怪的行为。如果池已经具有与核心数量相同的线程，`ThreadPoolExecutor` 只会在工作队列已满的情况下创建新的线程。所以如果任务通过工作队列提交给核心长度为零的线程池，那么不等到队列填满，这些任务就不会开始工作。这通常不是期望发生的事情。在 Java 6 中，通过 `allowCoreThreadTimeOut`，你可以允许所有的池线程响应超时；如果你有一个有限的线程池和一个有限的工作队列，同时又要所有的线程在没有任务的情况下销毁，可以设置非零的核心大小，来激活这个特性。

程队列。使用 Runnable 和一个清单结点来表现一个等待中的请求，的确比使用一个线程要便宜很多。但是如果客户抛给服务器的请求快过了服务器可以处理它们的速度，就仍然存在耗尽资源的风险。

即使通常平均请求率都很稳定，也难免会突然激增。尽管队列有助于缓和瞬时的任务激增，但是如果任务持续快速地到来，你最终还是必须要遏制住请求达到率，以避免耗尽内存⁴。即使没有耗尽内存，响应时间也会随着任务队列的增长而逐渐地变糟。

ThreadPoolExecutor 允许你提供一个 BlockingQueue 来持有等待执行的任务。任务排队有 3 种基本方法：无限队列、有限队列和同步移交（synchronous handoff）。队列的选择和很多其他的配置参数都有关系，比如池的大小等等。

newFixedThreadPool 和 newSingleThreadExecutor 默认使用的是一个无限的 LinkedBlockingQueue。如果所有的工作者线程都处于忙碌状态，任务将会在队列中等候。如果任务持续地快速到达，超过了它们被执行的速度，队列也会无限制地增加。

一个稳妥的资源管理策略是使用有限队列，比如 ArrayBlockingQueue 或者有限的 LinkedBlockingQueue 以及 PriorityBlockingQueue。有界队列有助于避免资源耗尽的情况发生，但是它又引入了新的问题：当队列已满后，新的任务怎么办？有很多的**饱和策略**（saturation policie）可以处理这个问题；参见 8.3.3 节。对于一个有界队列，队列的长度与池的长度必须一起调节。一个大队列加一个小池，可以控制对内存和 CPU 的使用，还可以减少上下文切换，但是要接受潜在吞吐量约束的开销。

对于庞大或者无限的池，你也可以使用 SynchronousQueue，完全绕开队列，将任务直接从生产者移交给工作者线程。SynchronousQueue 并不是一个真正的队列，而是一种管理直接在线程间移交信息的机制。为了把一个元素放入到 SynchronousQueue 中，必须有另一个线程正在等待接受移交的任务。如果没有这样一个线程，只要当前池的大小还小于最大值，ThreadPoolExecutor 就会创建一个新的线程；否则根据饱和策略，任务会被拒绝。使用直接提交会更加高效，因为任务不必先放置到队列中，就可以立即交由即将执行的线程处理，然后再让工作者线程从队列中获取它。只有当池是无限的，或者可以接受任务被拒绝，SynchronousQueue 才是一个有实际价值的选择。newCachedThreadPool 工厂就使用了 SynchronousQueue。

使用 LinkedBlockingQueue 或 ArrayBlockingQueue 这种 FIFO(先进先出)的队列，会造成任务以它们到达的顺序开始执行。如果想更进一步地控制任务执行顺序，你还可以

⁴ 这类似于通信网络中的流控制（flow control）：你希望缓冲确定数量的数据，但是你最终仍然需要找到一种方式，让对方停止向你发送数据，或者你会索性丢弃过剩的数据，并希望发送者可以在你空闲的时候重新发送数据。

使用 `PriorityBlockingQueue`，它通过优先级安排任务。（如果任务实现了 `Comparable`）自然顺序或者 `Comparator` 都可以定义优先级。

`newCachedThreadPool` 工厂提供了比定长的线程池更好的队列等候性能⁵，它是 `Executor` 的一个很好的默认选择。出于资源管理的目的，当你需要限制当前任务的数量，一个定长的线程池就是很好的选择。就像一个接受网络客户端请求的服务器应用程序，如果不进行限制，就会很容易因为过载而遭受攻击。

只有当任务彼此独立时，才能使有限线程池或者有限工作队列的使用是合理的。倘若任务之间相互依赖，有限的线程池或队列就可能引起线程饥饿死锁；使用一个无限的池配置可以避免这类问题，就像 `newCachedThreadPool`⁶ 所作的。

8.3.3 饱和策略

当一个有限队列充满后，**饱和策略**开始起作用。`ThreadPoolExecutor` 的饱和策略可以通过调用 `setRejectedExecutionHandler` 来修改（如果任务提交到了一个已经被关闭的 `Executor` 时，也会用到饱和策略）。JDK 提供了几种不同的 `RejectedExecutionHandler` 实现，每一个都实现了不同的饱和策略：`AbortPolicy`、`CallerRunsPolicy`、`DiscardPolicy` 和 `DiscardOldestPolicy`

默认的“中止（abort）”策略会引起 `execute` 抛出未检查的 `RejectedExecutionException`；调用者可以捕获这个异常，然后编写能满足自己需求的处理代码。当最新提交的任务不能进入队列等待执行时，“**遗弃（discard）**”策略会默认放弃这个任务；“**遗弃最旧的（discard-oldest）**”策略选择丢弃的任务，是本来应该接下来就执行的任务，该策略还会尝试去重新提交新任务。（如果工作队列是优先级队列，那么“遗弃最旧的”策略选择丢弃的刚好是优先级最高的元素，所以混合使用“遗弃最旧的”饱和策略和优先级队列是不可行的）。

“**调用者运行（caller-runs）**”策略的实现形式，既不会丢弃哪个任务，也不会抛出任何异常。它会把一些任务推回到调用者那里，以此减缓新任务流。它不会在池线程中执行最新提交的任务，但是它会在一个调用了 `execute` 的线程中执行。我们修改 `WebServer`

⁵ 这个性能差异源自于使用 `SynchronousQueue` 取代了 `LinkedBlockingQueue`。在 Java 6 中，一个新的非阻塞算法取代了 `SynchronousQueue` 的原有算法，以此提高 `Executor` 的吞吐量。新算法以 `SynchronousQueue` 为基准，在 3 个以上的特性上胜过了 Java 5.0 中的 `SynchronousQueue` 实现（Scherer et al., 2006）。

⁶ 假设有一个任务会提交其他任务并等待它们的结果。对于这种情况，还有一个可选的配置策略是：使用一个受限的线程池，工作队列选用 `SynchronousQueue`，饱和策略选择“调用者运行”策略。

的例子，让它使用有限队列和**调用者运行**策略。当所有的池线程都被占用，而且工作队列已充满后，下一个任务会在主线程中执行。主线程调用 `execute` 执行这个任务。因为这将会花费一些时间，所以主线程在一段时间内不能提交任何任务。同时这也给了工作者线程时间来追赶进度。这期间主线程也不会调用 `accept`，所以外来的请求不会出现在应用程序中，而会在 TCP 层的队列中等待。如果持续高负载的话，最终会由 TCP 层判断它的连接请求队列是否已经排满，如果已满就开始丢弃请求任务。当服务器过载时，它的负荷会逐渐地外移——从池线程到工作队列到应用程序再到 TCP 层，最终转嫁到用户头上——这使得服务器在高负载下可以平缓地劣化（*graceful degradation*）。

Executor 创建时可以同时选择饱和策略，并对执行策略进行其他改变。清单 8.3 阐释了如何创建一个定长的线程池，它使用“调用者运行”饱和策略。

清单 8.3 创建一个可变长的线程池，使用受限队列和“调用者运行”饱和策略

```
ThreadPoolExecutor executor
= new ThreadPoolExecutor(N_THREADS, N_THREADS,
    0L, TimeUnit.MILLISECONDS,
    new LinkedBlockingQueue<Runnable>(CAPACITY));
executor.setRejectedExecutionHandler(
    new ThreadPoolExecutor.CallerRunsPolicy());
```

当工作队列充满后，并没有预置的饱和策略来阻塞 `execute`。但是，使用 `Semaphore`（信号量）可以实现这个效果。`Semaphore` 会限制任务注入率（*injection rate*）。清单 8.4 的 `BoundedExecutor` 演示了这一点。这种方法使用一个非受限队列（没有理由同时限制队列大小和注入率），设置 `Semaphore` 的限制范围等于在池的大小上加上你希望允许的排队任务数量，因为 `Semaphore` 限制的是当前执行的任务数和等待执行的任务数。

8.3.4 线程工厂

无论何时，线程池需要创建一个线程，都要通过一个**线程工厂**（*thread factory*，参见清单 8.5）来完成。默认的线程工厂创建一个新的、非后台（*nondaemon*）的线程，并没有特殊的配置。详细指明一个线程工厂，能允许你定制池线程的配置信息。`ThreadFactory` 只有唯一的方法：`newThread`，它会在线程池需要创建一个新线程时调用。

有很多原因需要使用定制的线程工厂。你可能希望为池线程指明一个 `UncaughtExceptionHandler`，或者实例化一个定制 `Thread` 类的实例，比如用来执行调试日志的线程。你也可能希望修改池线程的优先级（通常这不是一个非常好的主意；参见 10.3.1 节）或者后台状态（*daemon status*，同上，这也并不都是好主意；参见 7.4.2 节）。或者你可能只是希望给池线程一个更有意义的名称，来简化对线程转储和错误日志的解释。

清单 8.4 使用 Semaphore 来遏制任务的提交

```
@ThreadSafe
public class BoundedExecutor {
    private final Executor exec;
    private final Semaphore semaphore;

    public BoundedExecutor(Executor exec, int bound) {
        this.exec = exec;
        this.semaphore = new Semaphore(bound);
    }

    public void submitTask(final Runnable command)
        throws InterruptedException {
        semaphore.acquire();
        try {
            exec.execute(new Runnable() {
                public void run() {
                    try {
                        command.run();
                    } finally {
                        semaphore.release();
                    }
                }
            });
        } catch (RejectedExecutionException e) {
            semaphore.release();
        }
    }
}
```

清单 8.5 ThreadFactory 接口

```
public interface ThreadFactory {
    Thread newThread(Runnable r);
}
```

清单 8.6 的 `MyThreadFactory` 演示了一个自定义的线程工厂。它实例化一个新的 `MyAppThread`，并且向构造函数中传递池的名称，这样可以在线程转储和错误信息中分辨出线程来自于哪个池。`MyAppThread` 也可以用于应用程序的其他地方，这样所有的线程都能利用它的这一可调试特性。

清单 8.6 定制的线程工厂

```
public class MyThreadFactory implements ThreadFactory {
    private final String poolName;

    public MyThreadFactory(String poolName) {
        this.poolName = poolName;
    }

    public Thread newThread(Runnable runnable) {
        return new MyAppThread(runnable, poolName);
    }
}
```

有趣的定制行为发生在 `MyAppThread` 中，正如清单 8.7 所示的，它让你可以给线程提供名字，设置自定义 `UncaughtExceptionHandler`，以此向 `Logger` 中写入信息，还能够维护统计信息，记录已经创建与销毁的线程数，最后可以选择线程在创建或终止时，把调试消息写入日志。

如果你希望利用**安全策略**为某些特定的代码基（codebase）授予权限，你可能想要使用 `Executors` 中的 `privilegedThreadFactory` 工厂来构建你的线程工厂。这样创建出来的池线程，与创建 `privilegedThreadFactory` 的线程拥有相同的权限、`AccessControlContext` 和 `contextClassLoader`。不使用 `privilegedThreadFactory` 的话，线程池创建的线程所继承的权限，是客户调用 `execute` 或 `submit` 的当时，一个新线程所需要的权限。这会引起令人迷惑的与安全相关的异常。

8.3.5 构造后再定制 ThreadPoolExecutor

大多数通过构造函数传递给 `ThreadPoolExecutor` 的参数（比如核心池大小，最大池大小，存活时间，线程工厂和拒绝执行处理器（`rejected execution handler`）），都可以在创建后通过 `setters` 进行修改。如果 `Executor` 是通过 `Executors` 中的某一个工厂方法（除 `newSingleThreadExecutor` 以外）创建的，你可以像清单 8.8 那样，首先把结果转型为 `ThreadPoolExecutor`，然后访问 `setters` 方法。

`Executors` 中包括一个名为 `unconfigurableExecutorService` 的工厂方法，它返回一个现有的 `ExecutorService`，并对它进行包装。它只暴露出 `ExecutorService` 的方法，因此不能进行进一步的配置。`newSingleThreadExecutor` 与其他池化的实现不同，它用

清单 8.7 自定义的线程基类

```
public class MyAppThread extends Thread {
    public static final String DEFAULT_NAME = "MyAppThread";
    private static volatile boolean debugLifecycle = false;
    private static final AtomicInteger created = new AtomicInteger();
    private static final AtomicInteger alive = new AtomicInteger();
    private static final Logger log = Logger.getAnonymousLogger();

    public MyAppThread(Runnable r) { this(r, DEFAULT_NAME); }

    public MyAppThread(Runnable runnable, String name) {
        super(runnable, name + "-" + created.incrementAndGet());
        setUncaughtExceptionHandler(
            new Thread.UncaughtExceptionHandler() {
                public void uncaughtException(Thread t,
                                                Throwable e) {
                    log.log(Level.SEVERE,
                        "UNCAUGHT in thread " + t.getName(), e);
                }
            });
    }

    public void run() {
        // 复制 debug, 确保它的值自始至终是一致的
        boolean debug = debugLifecycle;
        if (debug) log.log(Level.FINE, "Created "+getName());
        try {
            alive.incrementAndGet();
            super.run();
        } finally {
            alive.decrementAndGet();
            if (debug) log.log(Level.FINE, "Exiting "+getName());
        }
    }

    public static int getThreadsCreated() { return created.get(); }
    public static int getThreadsAlive() { return alive.get(); }
    public static boolean getDebug() { return debugLifecycle; }
    public static void setDebug(boolean b) { debugLifecycle = b; }
}
```

清单 8.8 修改一个标准工厂方法创建的 Executor

```
ExecutorService exec = Executors.newCachedThreadPool();
if (exec instanceof ThreadPoolExecutor)
    ((ThreadPoolExecutor) exec).setCorePoolSize(10);
else
    throw new AssertionError("Oops, bad assumption");
```

这样的方式返回一个封装过的 `ExecutorService`，而不是原始的 `ThreadPoolExecutor`。尽管使用只包含唯一线程的线程池，确实也可以实现单线程化的 `Executor`，但是它也承诺不并发地执行任务。如果一些具有误导性的代码试图增加一个单线程化 `executor` 的池大小，这会破坏代码预期的语义。

你可以将这项技术用在你自己的 `Executor` 中，来防止执行策略被修改。如果你将 `ExecutorService` 暴露给你不信任的代码，不希望它会被修改，可以用一个 `unconfigurableExecutorService` 包装它。

8.4 扩展 ThreadPoolExecutor

`ThreadPoolExecutor` 的设计是可扩展的，它提供了几个“钩子”让子类去覆写——`beforeExecute`、`afterExecute` 和 `terminate`——这些可以用来扩展 `ThreadPoolExecutor` 行为。

执行任务的线程会调用钩子函数 `beforeExecute` 和 `afterExecute`，用它们添加日志、时序、监视器或统计信息收集的功能。无论任务是正常地从 `run` 中返回，还是抛出一个异常，`afterExecute` 都会被调用。（如果任务完成后抛出一个 `Error`，则 `afterExecute` 不会被调用。）如果 `beforeExecute` 抛出一个 `RuntimeException`，任务将不被执行，`afterExecute` 也不会被调用。

`terminated` 钩子会在线程池完成关闭动作后调用，也就是当所有任务都已完成并且所有工作者线程也已经关闭后，会执行 `terminated`。`terminated` 可以用来释放 `Executor` 在生命周期里分配到的资源，还可以发出通知、记录日志或者完成统计信息。

8.4.1 示例：给线程池加入统计信息

清单 8.9 的 `TimingThreadPool` 显示了一个定制的线程池，它通过使用 `beforeExecute`、`afterExecute` 和 `terminated`，加入了日志和统计收集功能。为了监控任务的运行时，`beforeExecute` 必须记录开始时间并把它存储到一个 `afterExecute` 可以找到的地方。因为钩子函数是在执行任务的线程中被调用，因此 `beforeExecute` 可以把

值存入到一个 `ThreadLocal` 中,就可以让 `afterExecute` 获得那个值。`TimingThreadPool` 使用了一对 `AtomicLong`, 分别用于追踪已处理的任务数和处理时间的总和, 最后使用 `terminated` 钩子将平均任务时间的日志消息打印出来。

清单 8.9 扩展线程池以提供日志和计时功能

```
public class TimingThreadPool extends ThreadPoolExecutor {
    private final ThreadLocal<Long> startTime
        = new ThreadLocal<Long>();
    private final Logger log = Logger.getLogger("TimingThreadPool");
    private final AtomicLong numTasks = new AtomicLong();
    private final AtomicLong totalTime = new AtomicLong();

    protected void beforeExecute(Thread t, Runnable r) {
        super.beforeExecute(t, r);
        log.fine(String.format("Thread %s: start %s", t, r));
        startTime.set(System.nanoTime());
    }

    protected void afterExecute(Runnable r, Throwable t) {
        try {
            long endTime = System.nanoTime();
            long taskTime = endTime - startTime.get();
            numTasks.incrementAndGet();
            totalTime.addAndGet(taskTime);
            log.fine(String.format("Thread %s: end %s, time=%dns",
                t, r, taskTime));
        } finally {
            super.afterExecute(r, t);
        }
    }

    protected void terminated() {
        try {
            log.info(String.format("Terminated: avg time=%dns",
                totalTime.get() / numTasks.get()));
        } finally {
            super.terminated();
        }
    }
}
```

8.5 并行递归算法

6.3 节的页面渲染程序历经了一系列的精细化，不断地发掘可以利用的并行性。第一次的做法是让程序完全顺序化执行；第二次用到了两个线程，但仍然是顺序地下载所有图像；最终版本把每个图像的下载视为一个独立的任务，从而达到了更好的并行性。循环体如果包含重要的计算，或者要执行潜在的阻塞操作，那么这种循环常常都是并行化工作的目标，当然前提是，每次迭代都是彼此独立的。

如果一个循环的每次迭代都是独立的，并且我们不必等待所有的迭代都完成后再一起处理，那么我们可以使用 `Executor` 把一个顺序的循环转化为并行的循环，如清单 8.10 所示的 `processSequentially` 和 `processInParallel`。

清单 8.10 把顺序执行转换为并行执行

```
void processSequentially(List<Element> elements) {
    for (Element e : elements)
        process(e);
}

void processInParallel(Executor exec, List<Element> elements) {
    for (final Element e : elements)
        exec.execute(new Runnable() {
            public void run() { process(e); }
        });
}
```

调用 `processInParallel` 会比调用 `processSequentially` 更快地得到返回，因为只要所有的下载任务都进入了 `Executor` 的队列，`processInParallel` 就会立刻返回，而不用等到这些任务全部完成。如果你需要提交一个任务集并等待它们完成，那么可以使用 `ExecutorService.invokeAll`；当所有结果都可用后，你可以使用 `CompletionService` 来获取结果，就像第 130 页的 `Renderer`。

当每个迭代彼此独立，并且完成循环体中每个迭代的工作，意义都足够重大，足以弥补管理一个新任务的开销时，这个顺序循环是适合并行化的。

循环并行化同样可以应用于一些递归设计中；通常递归算法的内部会存在顺序循环，这些循环可以按照清单 8.10 的方式进行并行化。一种简单的情况是，每个迭代都不需要来自于它所调用的迭代的结果。举个例子，清单 8.11 的 `sequentialRecursive` 以深度优先遍历

一棵树，并在每个节点上执行计算，把结果放入一个容器中。修改后的版本 `parallelRecursive`，同样进行深度优先的遍历，但是它并不是在访问节点时进行计算，而是为每个节点都提交一个任务来完成计算。

清单 8.11 把顺序递归转换为并行递归

```
public<T> void sequentialRecursive(List<Node<T>> nodes,
                                   Collection<T> results) {
    for (Node<T> n : nodes) {
        results.add(n.compute());
        sequentialRecursive(n.getChildren(), results);
    }
}

public<T> void parallelRecursive(final Executor exec,
                                List<Node<T>> nodes,
                                final Collection<T> results) {
    for (final Node<T> n : nodes) {
        exec.execute(new Runnable() {
            public void run() {
                results.add(n.compute());
            }
        });
        parallelRecursive(exec, n.getChildren(), results);
    }
}
```

当 `parallelRecursive` 返回的时候，树上的每个节点都已经被访问了（不过遍历的过程仍然是顺序的：只有对 `compute` 的调用才是并行执行的），每个节点的计算任务已经进入 `Executor` 的工作队列。`parallelRecursive` 的调用者可以创建一个 `Executor` 用来遍历，并使用 `shutdown` 和 `awaitTermination`，等待所有的结果。就像清单 8.12 所示的。

清单 8.12 等待并行运算的结果

```
public<T> Collection<T> getParallelResults(List<Node<T>> nodes)
    throws InterruptedException {
    ExecutorService exec = Executors.newCachedThreadPool();
    Queue<T> resultQueue = new ConcurrentLinkedQueue<T>();
    parallelRecursive(exec, nodes, resultQueue);
    exec.shutdown();
    exec.awaitTermination(Long.MAX_VALUE, TimeUnit.SECONDS);
    return resultQueue;
}
```

8.5.1 示例：谜题框架

这项技术的一种有趣的应用是解决谜题，它所解决的谜题包括从一些初始状态寻找到达目标状态的转换序列，比如类似于“搬箱子”⁷，“Hi-Q”，“Instant Insanity”和其他的棋牌谜题。

我们这样定义一个“谜题”，它包含了一个初始位置，一个目标位置，为了确定起点与目标之间的有效移动，还包含一个规则集。规则集有两部分：计算一个清单，包含始于给定位置的合法移动；计算移动到某位置的可能结果。清单 8.13 演示了这个谜题的抽象；类型参数（type parameter）P 和 M 代表了位置（position）和移动（move）的类。根据这个接口，我们可以写一个简单的顺序执行的解决者（solver），它会在谜题空间（puzzle space）中查找，直到找的一个解答，或者找遍了整个空间而没有发现答案。

清单 8.13 类似于“搬箱子”谜题的抽象

```
public interface Puzzle<P, M> {
    P initialPosition();
    boolean isGoal(P position);
    Set<M> legalMoves(P position);
    P move(P position, M move);
}
```

清单 8.14 的 Node 代表一个位置（position），经过一系列的移动（move）后到达该位置。它有一个 move 引用，move 创建了当前位置和前一个 Node。我们沿着 Node 链接逐步后退，可以重新构建出到达当前位置的移动序列。

清单 8.15 的 SequentialPuzzleSolver 演示了一个谜题框架的顺序化的解决器，它在谜题空间中执行一个深度优先搜索，并会在发现一个解答（不必是最短的方案）后终止。

改写前一个解决者并加强一些并发性，我们就可以同时判断目标条件和计算下一步移动。因为计算一次移动的过程，几乎和计算其他移动的过程是彼此独立的。（我们之所以说“几乎”，是因为不同任务会共享一些可变状态，比如可见位置的集合。）如果有多个处理器可用，就能减少寻找方案所花费的时间。

清单 8.16 的 ConcurrentPuzzleSolver 使用了内部类 SolverTask，SolverTask 同时扩展了 Node 并实现了 Runnable。大多数工作是在 run 中完成的：首先计算下一步可能的位置集，从中除去已经到达的位置，然后判断是否（这个任务或者其他一些任务）已经成功地完成，最后要把尚未搜索的位置提交给 Executor。

为了避免无限循环，顺序版本的解决者维护了一个保存先前已搜索到的位置的 Set；ConcurrentPuzzleSolver 使用 ConcurrentHashMap 也是出于同样的目的。这种做法提供了线程安全性，还避免了依据条件更新共享容器时固有的竞争条件，因为 ConcurrentHashMap

⁷ 参见<http://www.puzzleworld.org/SlidingBlockPuzzles>。

清单 8.14 谜题解决者框架的链节点

```
@Immutable
static class Node<P, M> {
    final P pos;
    final M move;
    final Node<P, M> prev;

    Node(P pos, M move, Node<P, M> prev) {...}

    List<M> asMoveList() {
        List<M> solution = new LinkedList<M>();
        for (Node<P, M> n = this; n.move != null; n = n.prev)
            solution.add(0, n.move);
        return solution;
    }
}
```

使用了 `putIfAbsent`, 可以原子化地添加一个位置, 当且仅当这个位置先前是不存在的。`ConcurrentPuzzleSolver` 使用内部线程池的工作队列保留搜索的状态, 而不是调用栈。

并发的方法还用一种限制形式换取了另一个可能更加适合这个问题域的有利条件。顺序版本的程序执行的是深度优先的搜索, 所以搜索的能力是受可用栈的大小限制的。并发版本的程序执行的是广度优先的搜索, 因此不会受到栈大小的影响 (但是仍然可以耗尽内存: 待搜索的或已搜索的位置集超出了可用的内存)。

为了在发现一个解法后可以停止搜索行为, 我们需要一种方式来检查是否有哪个线程已经找到了一个方案。如果我们想接受第一个被发现的方案, 我们只有在还没发现任何解法时才去更新方案。这些条件描述了一种闭锁 (latch) (参见 5.5.1 节) 的行为, 具体地说, 是一种可携带结果的闭锁。我们可以利用第 14 章介绍的技术, 简单地构造一个阻塞的可携带结果的闭锁。不过, 通常使用现有的核心库类, 比使用底层的语言机制更简单, 能够避免更多的错误。清单 8.17 的 `ValueLatch` 使用 `CountDownLatch` 提供了我们需要的闭锁行为, 并且使用锁保证了方案仅被设置一次。

每个任务首先会向闭锁请求方案, 如果方案已被发现, 就停止。在找到一个方案之前, 主线程需要等待; `ValueLatch` 中的 `getValue` 会一直阻塞, 直到有线程设置了 `value`。`ValueLatch` 为我们提供了方法来持有 `value`, 这样它只有在第一次调用时才进行设置。调用者能够检查 `value` 是否能够被设置, 而且调用者可以阻塞, 等候 `value` 被设置。第一次调用 `setValue`, 会更新方案, 并且消耗 `CountDownLatch`, 从 `getValue` 中释放主解决器线程。

第一个发现解法的线程还会关闭 `Executor`, 以拒绝接受新的任务。为了避免处理 `RejectedExecutionException`, 我们可以设置一个拒绝执行处理器, 让它丢弃已提交的任务。

清单 8.15 顺序化的谜题解决者

```
public class SequentialPuzzleSolver<P, M> {
    private final Puzzle<P, M> puzzle;
    private final Set<P> seen = new HashSet<P>();

    public SequentialPuzzleSolver(Puzzle<P, M> puzzle) {
        this.puzzle = puzzle;
    }

    public List<M> solve() {
        P pos = puzzle.initialPosition();
        return search(new Node<P, M>(pos, null, null));
    }

    private List<M> search(Node<P, M> node) {
        if (!seen.contains(node.pos)) {
            seen.add(node.pos);
            if (puzzle.isGoal(node.pos))
                return node.asMoveList();
            for (M move : puzzle.legalMoves(node.pos)) {
                P pos = puzzle.move(node.pos, move);
                Node<P, M> child = new Node<P, M>(pos, move, node);
                List<M> result = search(child);
                if (result != null)
                    return result;
            }
        }
        return null;
    }

    static class Node<P, M> { /* 清单 8.14 */ }
}
```

清单 8.16 并发版的谜题解决者

```
public class ConcurrentPuzzleSolver<P, M> {
    private final Puzzle<P, M> puzzle;
    private final ExecutorService exec;
    private final ConcurrentMap<P, Boolean> seen;
    final ValueLatch<Node<P, M>> solution
        = new ValueLatch<Node<P, M>>();
    ...
    public List<M> solve() throws InterruptedException {
        try {
            P p = puzzle.initialPosition();
            exec.execute(newTask(p, null, null));
            // 阻塞, 直到发现一个方案
            Node<P, M> solnNode = solution.getValue();
            return (solnNode == null) ? null : solnNode.asMoveList();
        } finally {
            exec.shutdown();
        }
    }

    protected Runnable newTask(P p, M m, Node<P, M> n) {
        return new SolverTask(p, m, n);
    }

    class SolverTask extends Node<P, M> implements Runnable {
        ...
        public void run() {
            if (solution.isSet())
                || seen.putIfAbsent(pos, true) != null)
                return; // 已找到一个解决方案, 或者该位置曾经到达过
            if (puzzle.isGoal(pos))
                solution.setValue(this);
            else
                for (M m : puzzle.legalMoves(pos))
                    exec.execute(
                        newTask(puzzle.move(pos, m), m, this));
        }
    }
}
```

清单 8.17 ConcurrentPuzzleSolver 使用可携带结果的闭锁

```
@ThreadSafe
public class ValueLatch<T> {
    @GuardedBy("this") private T value = null;
    private final CountDownLatch done = new CountDownLatch(1);

    public boolean isSet() {
        return (done.getCount() == 0);
    }

    public synchronized void setValue(T newValue) {
        if (!isSet()) {
            value = newValue;
            done.countDown();
        }
    }

    public T getValue() throws InterruptedException {
        done.await();
        synchronized (this) {
            return value;
        }
    }
}
```

于是，所有未完成任务最终还会运行完成，然而任何后续的执行新任务的尝试都会默认为失败，允许终止 `Executor`（如果任务运行花费的时间很长，我们可能更希望中止它，而不是等它们完成）。

`ConcurrentPuzzleSolver` 还不能很好地处理“不存在任何方案”的情况：所有可能的移动（`move`）和位置（`position`）都已经被检查过，并没有发现一个方案，`solve` 会在 `getSolution` 的调用中永远等下去。顺序版本的程序会在考察了全部搜索空间后终止，但是想让并发的程序终止有时并不简单。一种可能的方法是：维护一个活动（`active`）解决者任务的计数器，当计数器减到零时，就将结果方案设置为 `null`，如同清单 8.18 所示。

发现一个解法所花费的时间，仍然可能比我们期望的要长；因此我们可以给解决器设置几种额外的中止条件。其中之一是时间限制；可以通过在 `ValueLatch` 中实现一个限时版本的 `getValue`（它将会使用 `await` 的 `timed` 版本），就能简单地做到这一点。如果 `getValue` 超时，程序会声明一个失败，并关闭 `Executor`。另一种中断条件是与谜题相关

清单 8.18 能够感知任务不存在的解决者

```
public class PuzzleSolver<P,M> extends ConcurrentPuzzleSolver<P,M> {
    ...
    private final AtomicInteger taskCount = new AtomicInteger(0);

    protected Runnable newTask(P p, M m, Node<P,M> n) {
        return new CountingSolverTask(p, m, n);
    }

    class CountingSolverTask extends SolverTask {
        CountingSolverTask(P pos, M move, Node<P, M> prev) {
            super(pos, move, prev);
            taskCount.incrementAndGet();
        }
        public void run() {
            try {
                super.run();
            } finally {
                if (taskCount.decrementAndGet() == 0)
                    solution.setValue(null);
            }
        }
    }
}
```

的标准，比如只搜索某个确定数量的位置。或者，我们也可以提供一个取消机制，让用户自己决定何时停止搜索。

总结

对于并发执行的任务，Executor 框架是强大且灵活的。它提供了大量可调节的选项，比如创建和关闭线程的策略，处理队列任务的策略，处理过剩任务的策略，并且提供了几十个钩子函数用于扩展它的行为。然而，和大多数强大的框架一样，草率地将一些设定组合在一起，并不能很好地工作；一些类型的任务需要特定的执行策略，而一些调节参数组合在一起后可能产生意外的结果。

GUI 应用程序

GUI Applications

即使你只写过简单的 Swing GUI 应用程序，也应该知道 GUI 应用程序有它们特定的线程问题。为了维护安全，某一个任务必须运行在 Swing 的事件线程中。但是你不应该在事件线程中执行耗时操作，以免 UI 失去响应。而且 Swing 的数据结构不是线程安全的，所以你必须小心地把它们限制在事件线程中。

几乎所有的 GUI 工具集都实现为**单线程化子系统**（single-threaded），意味着所有 GUI 的活动都被限制在一个单独的线程中，这其中就包括了 Swing 和 SWT。如果你没有打算写一个完全单线程化的程序，就会有一些活动部分运行在事件线程中，部分运行在另一个应用程序线程中。像其他与线程相关的 bug 一样，这种错误的分割并不会导致你的应用程序立即崩溃；你的程序会在一些难以发现的条件下出现古怪的行为。GUI 框架自身是单线程子系统，可能你的应用程序还不是，那么编写 GUI 代码的时候，你需要仔细地考虑线程问题。

9.1 为什么 GUI 是单线程化的

早期的 GUI 应用程序就是单线程化的，GUI 事件在“主事件循环”进行处理。现代的 GUI 框架使用了一个略微不同的模型：模型创建了一个专门的线程，**事件派发线程**（event dispatch thread, EDT）来处理 GUI 事件。

单线程化的 GUI 框架并不仅仅存在于 Java 中；Qt、NextStep、MacOS Cocoa、X Windows，等等都是单线程化的。也并不缺少反面的尝试；有很多试图写出多线程的 GUI 框架的努力，最终都由于竞争条件和死锁导致的稳定性问题，又回到了单线程化的事件队列模型的老路上来：采用一个专门的线程从队列中抽取事件，并把它们转发给应用程序定义的事件处理器。（AWT 最初曾尝试在某种程度上支持多线程访问，单线程化地实现 Swing 的决定主要基于 AWT 中的经验和教训。）

多线程的 GUI 框架会尤其易受死锁的影响，部分原因在于，输入事件处理与任何 GUI 组件背后的对象模型之间存在偶发的交互。用户发起的动作总会冒泡似的从操作系统传递给应用程序——先是由 OS 检测到一次鼠标点击，然后工具集把它转化为“鼠标点击”事件，最终它会作为一个高层事件（比如“ButtonPressed”事件）转发给应用程序的监听器。另一方面，应用程序发起的动作又会以冒泡的形式从应用程序传回操作系统——应用程序发起一个动作要改变某个组件的背景颜色，这会被转发给一个特定的组件类，最终转发给 OS 进行渲染。两种动作以完全相反的顺序访问相同的 GUI 对象，需要保证让每个对象都是线程安全的，这会导致一系列的锁顺序的不一致，这会直接引发死锁（参见第 10 章）。这个问题几乎在每一次 GUI 工具集的开发中都会出现，是经验之谈。

模型-视图-控制器（MVC）模式的普遍流行形成了导致多线程 GUI 框架出现死锁的另一诱因。把用户的交互分拨到模型、视图和控制器之间的协作中，极大地简化了 GUI 应用程序的实现，但这让不一致的锁顺序再次雪上加霜。控制器调用模型，模型通知视图已经发生了一些事情。控制器同样可以调用视图，视图可以依次回调模型来查询模型的状态。结果是，不一致的锁顺序再次伴随着死锁的风险一同到来。

Graham Hamilton，Sun 公司的 VP（Value Player），在他的 weblog¹ 中详尽地概括了这些挑战，描述了多线程 GUI 工具集之所以会成为计算机科学史上又一次“失败的梦（failed dreams）”。

如果多线程 GUI 工具集经过非常谨慎的设计；如果工具集能使它加锁的方法鲜明地显露；如果你非常聪明，非常仔细，并且对工具集的整体框架有着全局的把握，我相信你还是可以成功地编写出多线程的 GUI 程序来。但是如果这些事情有一些轻微的偏差，程序多数时候仍然运行良好，但是你会偶尔看到（死锁引起的）程序挂起或者（竞争引起的）运行故障。那些密切参与了工具集设计的人才能够很好地运用这种多线程方案。

不幸的是，我认为这些特性并没有和商业流行度成正比。一个中等能力的程序员，整日构建着被一些莫名其妙的原因困扰着而不能稳定运行的应用程序，我们很容易落入这种境地。于是应用程序的作者会倍感怨恨与失落，对无辜的工具集恶语相加。

单线程化的 GUI 框架通过线程限制来达到线程安全性；所有 GUI 中的对象，包括可视化组件和数据模型，都只能被事件线程访问。当然，这只把线程安全负担的一部分推给了应用程序的开发者，他们必须确保这些对象是被正确限制的。

¹ <http://weblogs.java.net/blog/kgh/archive/2004/10>

9.1.1 顺序事件处理

GUI 应用程序总要去处理精细的事件，比如点击鼠标、按下键盘或者定时器到时等等。事件不过是另一种类型的任务；而 AWT 和 Swing 提供的事件处理工具在结构上也类似于 Executor。

因为只有唯一的一个线程在处理 GUI 任务，所以它会依次处理 GUI 任务——一个任务结束后才开始下一个，不会让两个任务交迭。了解了这一点，你就可以更容易地编写任务代码，不必担心来自其他任务的干扰。

顺序任务处理不利的一面是，如果一个任务的执行要花费很长时间，其他任务也必须等到它结束。如果后面的任务是负责响应用户输入或者提供一个可视化反馈的，那么应用程序将会表现出冻结状态。如果一个冗长的任务运行在事件线程中，用户甚至不能点击“取消”按钮，因为直到这个任务完成前，取消按钮的监听器都不会被调用。因此，在事件线程中执行的任务必须尽快地把控制权返回给事件线程。对大文档执行拼写检查，在文件系统中执行搜索，或者通过网络获取资源，为了启动类似的耗时任务，你必须在事件线程以外的线程中运行它，这样才能将控制权尽快返回给事件线程。为了能够在一个耗时任务运行中更新它的进度标识符，或者当任务完成后提供一个可视化的反馈，你再一次需要执行事件线程中的代码。这也会让程序快速地变复杂。

9.1.2 Swing 中的线程限制

所有的 Swing 组件（比如 JButton 和 JTable）和数据模型（比如 TableModel 和 TreeModel）都被限制于事件线程中，所以任何访问它们的代码必须在事件线程中运行。GUI 对象不用同步，仅仅依靠线程限制来保持一致性。这样做有利的一面是，运行于事件线程的任务，在访问表现对象（presentation objects）时不必担心同步的问题；不利的一面是，你完全无法从事件线程之外的地方访问表现对象。

Swing 的单线程规则： Swing 的组件和模型只能在事件分派线程中被创建、修改和请求。

纵观所有规则，也有一些例外。有非常少量的 Swing 方法可以安全地被任意线程调用；这些方法的线程安全性已经在 Javadoc 中清楚地说明了。单线程规则中还有一些其他的特例，其中包括：

- `SwingUtilities.isEventDispatchThread`，它用来判断当前线程是否是事件线程；
- `SwingUtilities.invokeLater`，它可以安排一个 `Runnable` 任务在事件线程中执行（可从任意线程中调用）；

- `SwingUtilities.invokeLater`，它可以安排一个 `Runnable` 任务在事件线程中执行，并且会阻塞当前线程直到它完成（只能从非 GUI 线程中调用）；
- 用于将一个 `repaint` 或 `revalidation` 请求插入队列的方法集（可从任意线程中调用）；
- 最后用于添加或移除监听器的方法集（可被任意线程调用，但是监听器一定在事件线程中调用）。

`invokeLater` 和 `invokeAndWait` 两个方法行使的职责很像一个 `Executor`。事实上，如果用一个单线程化的 `Executor` 来实现 `SwingUtilities` 中与线程相关的方法，是很简单的，就像清单 9.1 那样。这并不是 `SwingUtilities` 真实的实现，因为 `Swing` 的产生要早于 `Executor` 框架。但是如果今天再来看的话，`Swing` 也许真的应该这样实现。

`Swing` 的事件线程可以看作是一个单线程化的 `Executor`，它执行事件队列中的任务。与线程池一样，有时一个死亡的工作者线程会被另一个新的取代，不过这一切对任务来说是透明的。如果所有的任务都是短期的，或者任务调度的可预见性并不重要，或者迫切需要任务进行非并发执行时，顺序的、单线程化的执行策略就是明智之选。

清单 9.2 的 `GuiExecutor` 是一个 `Executor`，它把任务的执行委托给 `SwingUtilities` 完成。用其他的 GUI 框架也可以实现它；比如说，SWT 提供的 `Display.asyncExec` 方法就类似于 `Swing` 中的 `invokeLater`。

9.2 短期的 GUI 任务

在 GUI 应用程序中，事件起源于事件线程，冒泡似的传递到达应用程序提供的监听器，监听器进而可能会执行一些影响表现模型（`presentation object`）的运算。为了简单起见，短期的任务可以把全部动作留在事件线程中完成；而对于耗时的任务，则应该将一些工作负荷分压到另一个线程中。

在简单的情况下，将表现对象限制于事件队列，是一件自然而然的事情。清单 9.3 中创建了一个按钮，它的颜色在被按下时会随机变化。当用户点击按钮后，工具集（`toolkit`）在事件线程中向所有注册的 `ActionListener` 发送一个 `ActionEvent`。作为响应，`ActionListener` 会挑选一个新的颜色，并把按钮的背景色设置为这个新颜色。如此一来，事件先在 GUI 工具集中产生，后被发送给应用程序，应用程序再改变 GUI 以响应用户的动作。在此期间控制权从来都不用脱离事件线程，如图 9.1 所示。

这个简单的例子揭示了 GUI 应用程序和 GUI 工具集之间主要的互交。只要任务是短期的，而且只访问 GUI 对象（或者被其他线程限制以及与线程安全的应用程序对象），那么你可以几乎完全忽略线程的问题，在事件线程中做任何事，一定不会出问题的。

清单 9.1 使用 Executor 实现的 SwingUtilities

```
public class SwingUtilities {
    private static final ExecutorService exec =
        Executors.newSingleThreadExecutor(new SwingThreadFactory());
    private static volatile Thread swingThread;

    private static class SwingThreadFactory implements ThreadFactory {
        public Thread newThread(Runnable r) {
            swingThread = new Thread(r);
            return swingThread;
        }
    }

    public static boolean isEventDispatchThread() {
        return Thread.currentThread() == swingThread;
    }

    public static void invokeLater(Runnable task) {
        exec.execute(task);
    }

    public static void invokeAndWait(Runnable task)
        throws InterruptedException, InvocationTargetException {
        Future f = exec.submit(task);
        try {
            f.get();
        } catch (ExecutionException e) {
            throw new InvocationTargetException(e);
        }
    }
}
```

清单 9.2 构建于 SwingUtilities 之上的 Executor

```

public class GuiExecutor extends AbstractExecutorService {
    // Singleton 包含一个私有的构造函数和一个公共的工厂
    private static final GuiExecutor instance = new GuiExecutor();

    private GuiExecutor() { }

    public static GuiExecutor instance() { return instance; }

    public void execute(Runnable r) {
        if (SwingUtilities.isEventDispatchThread())
            r.run();
        else
            SwingUtilities.invokeLater(r);
    }

    // 添加其他有关生命周期方法的实现
}

```

清单 9.3 简单的事件监听器

```

final Random random = new Random();
final JButton button = new JButton("Change Color");
...
button.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        button.setBackground(new Color(random.nextInt()));
    }
});

```

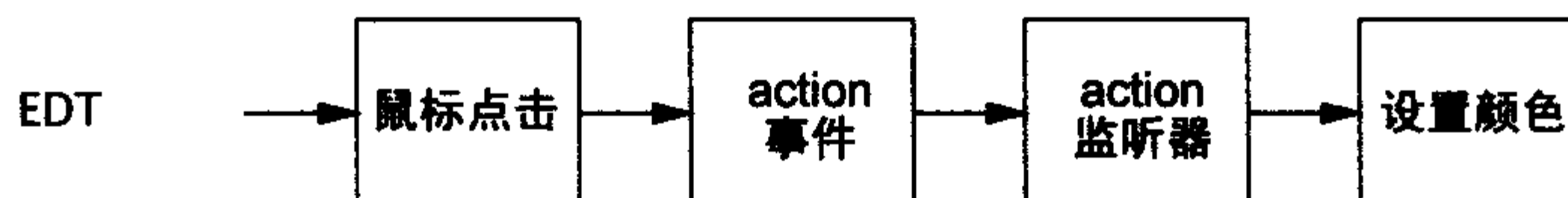


图 9.1 简单的点击按钮事件的控制流

图 9.2 显示了同一场景略微复杂些的版本，它涉及了规范数据模型的使用，比如 `TableModel` 或 `TreeModel`。Swing 把大多数可视化组件都分成了两个对象，模型（model）与视图（view）。模型中存储了将被显示的数据，视图中存储了管理显示方式的规则。模型对象发起一个事件表明自身已经发生了变化，视图会收到这些事件。视图收到表示模型数据已变的事件后，就向模型查询新的数据，然后更新显示。所以，在一个改变表内容的按钮监听器中，事件监听器会更新模型并调用 `fireXxx` 方法中的一个，`fireXxx` 方法再依

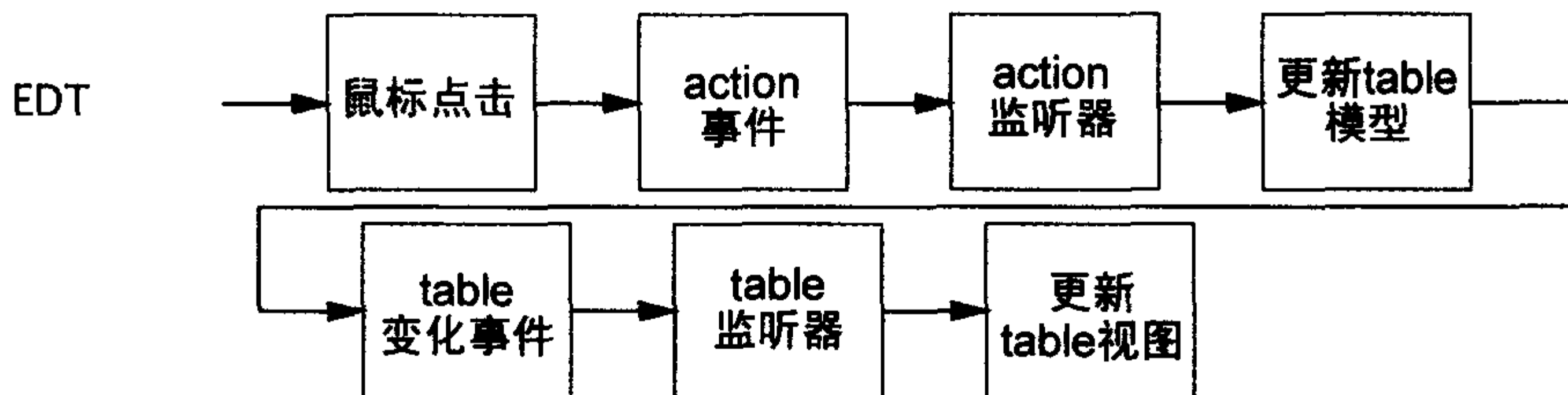


图 9.2 分拆了模型和视图对象的控制流

次调用视图中表的模型监听器，以此来更新视图。这一次，控制权仍然没有脱离事件线程。

（Swing 数据模型的 `fireXxx` 方法总是直接去调用模型监听器，并不会向线程队列中提交新的事件，所以 `fireXxx` 方法只能在事件线程中调用。）

9.3 耗时 GUI 任务

如果所有的任务都是短期的（而且应用程序不存在重要的非 GUI 的部分），那么整个应用程序就可以在事件线程内部运行，你可以完全不必关心线程。但是，成熟的 GUI 应用程序可能会运行一些耗时的任务，以致于超过了用户预期的时间，比如拼写检查、后台编辑或者获取远程资源。这些任务必须在另外的线程中运行，而使 GUI 在它们运行中可以作出响应。

Swing 让在事件线程中运行任务更容易，但是（在 Java 6 之前）并没有提供任何机制帮助 GUI 任务能在其他线程中执行代码。好在我们也不用 Swing 来帮忙：我们可以创建自己的 `Executor` 来执行耗时的任务。对于耗时的任务，可缓存线程池是个不错的选择；只有很少的 GUI 应用程序会发起大量的耗时任务，即使不限制池的增长几乎也没什么风险。

我们从一个简单的任务开始，它不支持取消操作和进度指示，而且它也不会完成后更新 GUI。在后面，我们会将这些特性一个个地放进来。清单 9.4 演示了一个，绑定了 `ActionListener` 的可视化组件，并向一个 `Executor` 提交耗时任务。尽管包含了两级内部类，不过这种从 GUI 任务启动另一个任务的方式还是相当直观的：UI 动作监听器在事件线程中被调用，然后提交一个执行于线程池中的 `Runnable`。

在这个例子中，耗时任务会以“触发并失效（fire and forget）”的方式（这很可能没有用处）从事件线程中溜出来。耗时任务完成后，通常会产生一些可视化的反馈。但是你并不能从后台线程中访问到表现对象，所以任务完成后必须向事件线程提交另一个任务来更新用户界面。

清单 9.4 将耗时任务绑定到可视化组件

```

ExecutorService backgroundExec = Executors.newCachedThreadPool();
...
button.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        backgroundExec.execute(new Runnable() {
            public void run() { doBigComputation(); }
        });
    }
});

```

清单 9.5 阐释的是完成这件事最容易想到的一种方式，不过这已经开始变得复杂了；我们现在已经开始依赖三层的内部类了。ActionListener 首先使按钮无效，并设置一个标签标识有一个计算正在进行；然后向后台 Executor 提交任务；待任务完成后，它会在事件线程中排入另一个任务等候运行，以此重新激活按钮并且恢复标签的文本。

清单 9.5 提供用户反馈的耗时任务

```

button.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        button.setEnabled(false);
        label.setText("busy");
        backgroundExec.execute(new Runnable() {
            public void run() {
                try {
                    doBigComputation();
                } finally {
                    GuiExecutor.instance().execute(new Runnable() {
                        public void run() {
                            button.setEnabled(true);
                            label.setText("idle");
                        }
                    });
                }
            }
        });
    }
});

```

按下按钮触发的任务由 3 个连续的子任务组成，它们在事件线程与后台线程之间交替运行。第一个子任务更新用户接口，提示一个耗时操作已经开始了，然后启动后台线程中

的第二个子任务。第二个子任务完成后，会提交第三个子任务，再次运行在事件线程中运行，它会更新用户接口，反映已经完成操作。在 GUI 应用程序中，这种“线程接力”是处理耗时操作的典型办法。

9.3.1 取消

任何一个在线程中运行了足够长时间的任务，都可能会耗费用户大量的时间，因此用户可能希望取消它。你可以使用线程中断直接实现取消的操作，不过更简单的办法是使用 `Future`，它就是设计用来管理可取消任务的。

如果你调用 `Future` 的 `cancel`，并设置 `mayInterruptIfRunning` 参数为 `true`，`Future` 的实现可以中断一个执行运行中的任务的线程。如果你编写的任务能处理中断，那么被取消后可以提前返回。清单 9.6 阐释了一个任务，它会轮询线程的中断状态，并且在中断时提前返回。

清单 9.6 取消耗时任务

```
Future<?> runningTask = null;    // 线程限制的
...
startButton.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        if (runningTask == null) {
            runningTask = backgroundExec.submit(new Runnable() {
                public void run() {
                    while (moreWork()) {
                        if (Thread.interrupted()) {
                            cleanUpPartialWork();
                            break;
                        }
                        doSomeWork();
                    }
                }
            });
        }
    }
});

cancelButton.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent event) {
        if (runningTask != null)
            runningTask.cancel(true);
    }
});
```

因为 `runningTask` 被限制在事件线程中，因此设置或检查它的时候不需要同步，并且 `Start Button` 的监听器可以确保同时只有一个后台任务在运行。然而，可以用更好的方

式通知任务完成，比如说可以禁用 `cancel` 按钮。我们将在下一节解决这个问题。

9.3.2 进度和完成标识

使用 `Future` 表现一个耗时任务，可以极大地简化取消的实现。类似地，`FutureTask` 中也有一个 `done` 钩子函数，可以方便任务完成后的通知。后台 `Callable` 完成后，会调用 `done`。通过由 `done` 触发一个事件线程中已经完成的任务，我们能够构建 `BackgroundTask` 类，它提供一个可在事件线程中调用的 `onCompletion` 钩子函数，如清单 9.7 所示。

`BackgroundTask` 同时还支持进度标识。`compute` 方法可以调用 `setProgress` 来以数字的形式指定进度。这会引发事件线程调用 `onProgress`，更新用户接口，显示进度的可视化信息。

要实现 `BackgroundTask`，你只需要实现 `compute`，它会被后台线程调用。你也可以选择是否覆写 `onCompletion` 和 `onProgress`，它们在事件线程中被调用。

基于 `FutureTask` 的 `BackgroundTask` 还简化了取消操作。与其要去检查线程的中断状态，不如让 `compute` 调用 `Future.isCancelled`。清单 9.8 使用 `BackgroundTask` 重新实现了清单 9.6 的程序。

9.3.3 SwingWorker

我们已经利用 `FutureTask` 和 `Executor` 构建了一个简单的框架，它会在后台线程中执行耗时任务，不会影响 GUI 的响应性。这些技术可以被用于任何单线程化的 GUI 框架，不仅只有 `Swing`。在 `Swing` 中，本章展示的诸多特性是由 `SwingWorker` 类提供的，包括取消，完成的通知，以及进度指示。不同版本的 `SwingWorker` 已经出版在 *The Swing Connection* 和 *The Java Tutorial* 中，更新版本包含在 Java 6 中。

9.4 共享数据模型

`Swing` 的表现对象（包括 `TableModel`、`TreeModel` 这些数据模型）是被限制在事件线程中的。在简单的 GUI 程序中，所有的可变对象都保存在表现对象中，事件线程之外唯一的线程就只有主线程。强制这些程序遵守单线程规则很容易：不要在主线程中访问数据模型或表现组件（`presentation component`）。更多的复杂程序可能要从持久化存储中移出或移入数据，比如文件系统、数据库，这就要使用其他线程，以免破坏了响应性。

最简单的情况是，数据模型中的数据全部由用户输入或者应用程序启动时静态地从文件或其他数据源加载的。在这种情况下，事件线程之外的任何线程都不可能触及到数据。

清单 9.7 支持取消、完成和进度通知的后台任务类

```

abstract class BackgroundTask<V> implements Runnable, Future<V> {
    private final FutureTask<V> computation = new Computation();

    private class Computation extends FutureTask<V> {
        public Computation() {
            super(new Callable<V>() {
                public V call() throws Exception {
                    return BackgroundTask.this.compute() ;
                }
            });
        }
        protected final void done() {
            GuiExecutor.instance().execute(new Runnable() {
                public void run() {
                    V value = null;
                    Throwable thrown = null;
                    boolean cancelled = false;
                    try {
                        value = get();
                    } catch (ExecutionException e) {
                        thrown = e.getCause();
                    } catch (CancellationException e) {
                        cancelled = true;
                    } catch (InterruptedException consumed) {
                    } finally {
                        onCompletion(value, thrown, cancelled);
                    }
                }
            });
        }
    }

    protected void setProgress(final int current, final int max) {
        GuiExecutor.instance().execute(new Runnable() {
            public void run() { onProgress(current, max); }
        });
    }

    // 在后台线程中调用
    protected abstract V compute() throws Exception;
    // 在事件线程中调用
    protected void onCompletion(V result, Throwable exception,
                                boolean cancelled) { }
    protected void onProgress(int current, int max) { }
    // 其他用于完成计算的方法
}

```

清单 9.8 在 BackgroundTask 中启动一个耗时的、可取消的任务

```
startButton.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        class CancelListener implements ActionListener {
            BackgroundTask<?> task;
            public void actionPerformed(ActionEvent event) {
                if (task != null)
                    task.cancel(true);
            }
        }
        final CancelListener listener = new CancelListener();
        listener.task = new BackgroundTask<Void>() {
            public Void compute() {
                while (moreWork() && !isCancelled())
                    doSomeWork();
                return null;
            }
            public void onCompletion(boolean cancelled, String s,
                                    Throwable exception) {
                cancelButton.removeActionListener(listener);
                label.setText("done");
            }
        };
        cancelButton.addActionListener(listener);
        backgroundExec.execute(listener.task);
    }
});
```

但是有时候，表现对象只不过是视图之上的另一个数据源，比如数据库、文件系统或者远程服务。这些情况下，数据进出应用程序的过程中，可以被多个线程可能触及到。

举个例子，你可以使用树控件来显示远程文件系统的内容。在你显示控件之前，并不用枚举完整的文件系统——那样做会花费大量的时间和内存。树应该在展开节点时惰性组装起来。即使只枚举远程卷上的单一目录，也会花费很长的时间，所以你可以考虑在后台任务中执行枚举。后台任务完成后，你还必须设法把数据送入树中。可以通过使用线程安全的树模型来实现：让 `invokeLater` 安排一个任务，把数据从后台任务中“推入”事件线程；或者也可以让事件线程池轮询查看数据是否可用。

9.4.1 线程安全的数据模型

只要阻塞不过度影响响应性，多线程操作数据的问题就可以通过线程安全的数据模型来解决。如果数据模型支持精细的并发，事件线程和后台线程就能够共享它，而且不存在响应性的问题。举个例子，第 65 页中的 `DelegatingVehicleTracker` 在底层使用了一个 `ConcurrentHashMap`，它的获取操作提供了高级别的并发性。不利的一面是，`ConcurrentHashMap` 不能提供一个一致的数据快照，而这可能是也可能不是需求的一部分。线程安全的数据模型必须在更新时生成事件，才能在数据变化后更新视图。

有时使用**版本化数据模型**（versioned data model，比如 `CopyOnWriteArrayList` [CPJ 2.2.3.3]）、线程安全性、一致性和良好的响应性会一举三得。当你请求到一个 copy-on-write 容器的迭代器，这个迭代器遍历的是它创建时的那个容器。但是，只有在遍历操作远远多于修改操作时，copy-on-write 容器才会提供更好的性能。一个机动车追踪应用程序可能就不适合用这个办法。细化的版本化数据结构可以避免上述限制。但是构建一个版本化的数据结构，让它既提供高效的并发访问，又能在旧数据无效后适时地丢弃，并不容易做到，因此，只有其他方法都不奏效后，才应该考虑使用它。

9.4.2 分拆数据模型

从 GUI 的角度看，Swing 的表模型类，比如 `TableModel` 和 `TreeModel`，是用于存储显示数据的最规范的地方。但是，通常这些对象模型自身又成为其他受控于应用程序的对象的“视图”。应用程序的数据模型既包含表现域，又包含应用域，我们称这种应用程序是**分拆模型**（split-model）的设计

在分拆模型的设计中，表现模型被限制在事件线程中。其他的模型，即**共享模型**（shared model），是线程安全的，事件线程和应用程序线程都可以访问。表现模型会注册共享模型的监听器，这样可以在更新时得到通知。共享模型在更新的消息中嵌入一个相关状态的快照，表现对象可以通过这个快照进行更新。或者，表现对象在收到一个更新事件后，也可以直接从共享模型中获取数据。

快照的方法固然简单，但却存在限制。当数据模型很小，更新频率不高，两个模型的结构类似时，它可以运行良好。如果数据模型很庞大，或者更新频率极高，不然就是分拆模型的一方或者双方包含的信息对另一方不可见，那么发送更新增量，要比发送一个完整的快照更加高效。与表现模型不同的是，这个方法在共享模型中对连续的更新能够起到很

好的效果，并会在事件线程中重新创建。增量更新的另一个好处是，细粒度化有改变的信息可以提高显示的视觉效果——如果只有一辆车移动，我们只需更新受影响的区域，不用重绘整个显示。

如果一个数据模型必须被多个线程共享，而且实现一个线程安全模型的尝试却由于阻塞、一致性或者复杂度等原因而失败，这时可以考虑运用分拆模型设计。

9.5 其他形式的单线程子系统

线程限制不仅仅限制在 GUI 系统：无论何时，它都可以用作实现单线程化子系统的便利工具。有时当程序员对避免同步与死锁束手无策的时候，使用线程限制成为了他们不得不使用的办法。比如，一些原生库（native library）要求所有对库的访问，甚至 `System.loadLibrary` 加载库时，必须在同一个线程中运行。

借用 GUI 框架中的方法，你可以简单地访问原生库创建一个专门的线程或者一个单线程化的 `Executor`，然后提供一个代理对象拦截所有对线程限制对象的调用，把拦截的调用当作任务提交给前面创建的线程中。`Future` 和 `newSingleThreadExecutor` 的组合可以简化这项工作；代理方法调用 `submit` 提交任务，然后立即调用 `Future.get` 等待结果。（如果线程限制的类实现了一个的接口，你就可以自动完成每次方法向后台线程 `Executor` 提交 `Callable` 的过程，然后利用动态代理收集结果。）（译注：Java 中的动态代理功能只支持 `interface`。）

总结

GUI 框架几乎都是作为单线程化子系统实现的，所有与表现相关的代码都作为任务在一个事件线程中运行。因为只有唯一一个线程，耗时任务会损害响应性，所以它们应该在后台线程中运行。像 `SwingWorker` 以及本章中构建 `BackgroundTask` 这些助手类，提供了对取消、进度指示、完成指示的支持。无论是 GUI 组件还是非 GUI 组件，都能借助它们简化耗时任务的开发。

PART

第 3 部分

活跃度，性能和测试

Liveness, Performance, and Testing

避免活跃度危险

Avoiding Liveness Hazards

安全性和活跃度通常相互牵制。我们使用锁来保证线程安全，但是滥用锁可能引起**锁顺序死锁**（lock-ordering deadlock）。类似地，我们使用线程池和信号量来约束资源的使用，但是却不能知晓那些管辖范围内的活动可能形成的**资源死锁**（resource deadlock）。Java 应用程序不能从死锁中恢复，所以确保你的设计能够避免死锁出现的先决条件是非常有价值的。这一章将讲述一些引发活跃度失败的原因，以及避免发生这些失败的方法。

10.1 死锁

经典的“哲学家进餐”问题很好地阐释了死锁，尽管这有点影响食欲。5 个哲学家一起出门去吃中餐，他们围坐在一个圆桌边。他们有 5 只筷子（不是 5 双），每两个人中间放有一只。哲学家边吃边思考，交替进行。每个人都需要获得两只筷子才能吃东西，但是吃后要把筷子放回原处继续思考。有一些管理筷子的算法，使每一个人都能够或多或少，及时吃到东西（一个饥饿的哲学家试图获得两只邻近的筷子，但是如果其中的一只正在被别人占用，那么他应该放弃其中一只可用的筷子，等待几分钟再尝试）。但是这样做可能导致一些哲学家或者所有哲学家都“饿死”（每个人都迅速抓住自己左边的筷子，然后等待自己右边的筷子变为可用，同时并不放下左边的筷子）。这后一种情况，当每个人都拥有他人需要的资源，并且等待其他人正在占有的资源，如果大家一直占有资源，直到获得自己需要却没被占有的其他资源，那么就会产生死锁。

当一个线程永远占有一个锁，而其他线程尝试去获得这个锁，那么它们将永远被阻塞。当线程 A 占有锁 L 时，想要获得锁 M，但是同时，线程 B 持有 M，并尝试获得 L，两个线程将永远等待下去。这种情况是死锁最简单的形式（或称**致命的拥抱**，deadly embrace），

发生在多个线程因为环路的锁依赖关系而永远等待的情况下。（把这些线程假想为有向图的节点，图的边表现了这个关系：“线程A等待线程B占有的资源”，如果图最后连成了一个环路，那么死锁也就产生了。）

数据库系统的设计就针对了监测死锁，以及从死锁中恢复。一个事务（transaction）可能需要取得许多锁，并可能一直持有这些锁，直到所有事务提交。如此说来两个事务非常有可能发生死锁，但这却并不常见。在没有干涉的情况下，它们将永远等待下去（持有的锁很可能是其他事务也需要的）。但是数据库服务器不允许这样的事情发生。当它监测到一个事务集发生了死锁（通过在表示**正在等待**（is-waiting-for）关系的有向图上搜索循环），它会选择一个牺牲者，使它退出事务。这个牺牲者释放的资源，使得其他事务能够继续进行。应用程序可以重新执行那个被强行退出的事务，现在这个事务可能就能够成功完成了，因为所有跟它竞争资源的事务都已经完成了。

JVM 在解决死锁问题方面与数据库服务还是有相当差距的。当一个 Java 线程集发生死锁时，“游戏”到此结束——这些线程永远不能再使用了。根据线程完成的不同工作，应用程序可能完全停止或者特定子系统停止，亦可能是性能受到影响。恢复应用程序健康的唯一方式就是中止并重启，然后寄希望于不要再发生同样的事情。

与许多其他的并发危险相同，死锁很少能被立即发现。一个类如果有发生死锁的潜在可能并不意味着死锁**每次都**将发生，它只发生在该发生的时候。当死锁出现的时候，往往是遇到了最不幸的时候——在高负载之下。

10.1.1 锁顺序死锁

清单 10.1 的 LeftRightDeadlock 存在死锁风险。leftRight 和 rightLeft 分别获得 left 锁和 right 锁。如果一个线程调用了 leftRight，另一个线程调用了 rightLeft，像图 10.1 中显式的那样交替进行，那么它们会发生死锁。

LeftRightDeadlock 发生死锁的原因是：两个线程试图通过**不同的顺序**获得多个相同的锁。如果请求锁的顺序相同，就不会出现循环的锁依赖现象，也就不会产生死锁了。如果你能够保证同时请求锁 L 和锁 M 的每一个线程，都是按照从 L 到 M 的顺序，那么就不会发生死锁了。

如果所有线程以通用的固定秩序获得锁，程序就不会出现锁顺序死锁问题了。

验证锁顺序的一致性需要对程序中锁的行为进行整体分析。单独观察每一个锁的代码路径仍然是不充分的；leftRight 和 rightLeft 都是获得两个锁“合理的”方式，它们只是不能相互协调而已。当死锁发生时，它们需要知道彼此究竟在做什么。

清单 10.1 简单的锁顺序死锁（不要这样做）

```
// 警告：易产生死锁
public class LeftRightDeadlock {
    private final Object left = new Object();
    private final Object right = new Object();

    public void leftRight() {
        synchronized (left) {
            synchronized (right) {
                doSomething();
            }
        }
    }

    public void rightLeft() {
        synchronized (right) {
            synchronized (left) {
                doSomethingElse();
            }
        }
    }
}
```

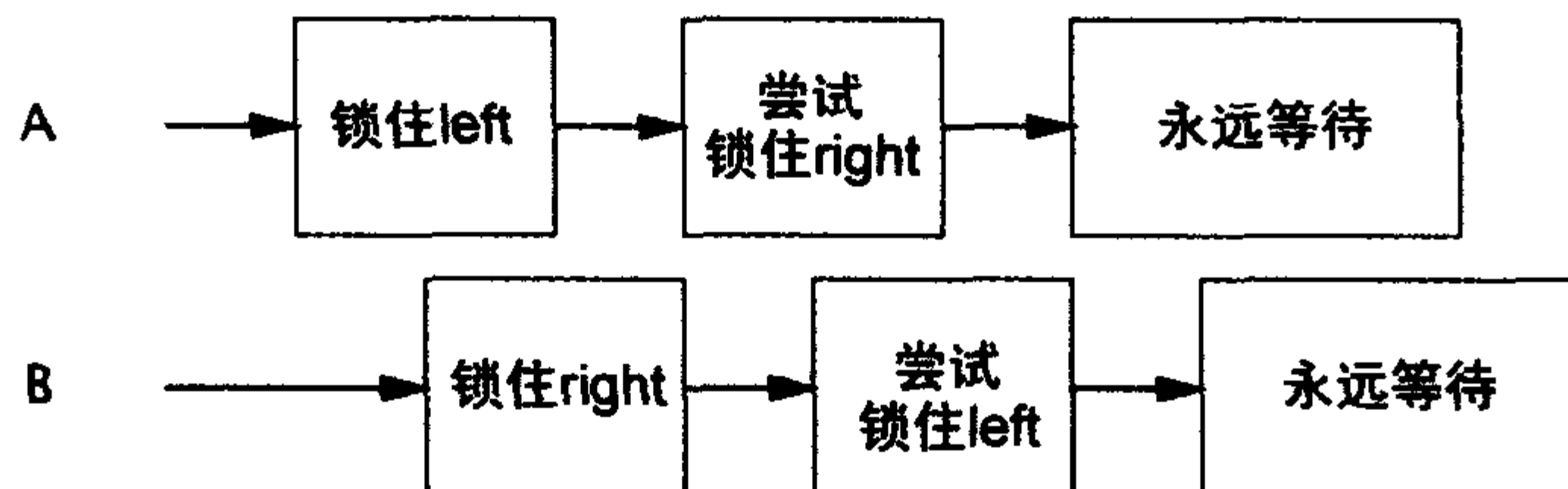


图 10.1 LeftRightDeadlock 的偶发时序

10.1.2 动态的锁顺序死锁

有时候你并不能一目了然地看清你是否已经对锁有足够的控制，来避免死锁的发生。思考清单 10.2 中看似无害的代码，它把资金从一个账户转入另一个账户。在执行转账之前要获得 Account 对象的锁，为了账目的平衡要保证更新操作是原子化的，同时不能破坏固有约束，比如“一个账户的余额不能为负数”。

transferMoney 怎么会发生死锁呢？所有线程看起来都是通过同样的顺序获得锁的，但是事实上锁的顺序取决于传递给 transferMoney 的参数顺序，这个参数的顺序实际上又取决于外部输入的顺序。

清单 10.2 动态加锁顺序产生的死锁（不要这样做）

//警告，易产生死锁

```

public void transferMoney(Account fromAccount,
                           Account toAccount,
                           DollarAmount amount)
    throws InsufficientFundsException {
    synchronized (fromAccount) {
        synchronized (toAccount) {
            if (fromAccount.getBalance().compareTo(amount) < 0)
                throw new InsufficientFundsException();
            else {
                fromAccount.debit(amount);
                toAccount.credit(amount);
            }
        }
    }
}

```



如果两个线程同时调用 `transferMoney`，一个从 *X* 向 *Y* 转账，另一个从 *Y* 向 *X* 转账，那么就会发生死锁：

```

A: transferMoney(myAccount, yourAccount, 10);
B: transferMoney(yourAccount, myAccount, 20);

```

在偶发的时序中，*A* 会获得 `myAccount` 的锁，并等待 `yourAccount` 的锁，然而 *B* 此时持有 `yourAccount` 的锁，正在等待 `myAccount` 的锁。

这样的死锁可以用清单 10.1 中的方法检查——获得锁时，查看是否有嵌套。因为参数的顺序是超出我们控制的，为了解决这个问题，我们必须制定锁的顺序，并且在整个应用程序中，获得锁都必须始终遵守这个既定的顺序。

我们在制定对象顺序的时候，可以使用 `System.identityHashCode` 这样一种方式，它会返回 `Object.hashCode` 所返回的值。清单 10.3 表现了 `transferMoney` 的又一个版本，它使用了 `System.identityHashCode` 定义了锁的顺序。虽然它带来了一些新的代码，但是它能够减少死锁发生的可能性。

在极少数的情况下，两个对象具有相同的哈希码，我们必须使用任意的“中数”来决定锁的顺序，这又重新引入了死锁的可能性。为了在这种条件下避免出现锁顺序的不一致，我们使用第三种“加时赛（tie-breaking）”锁。在获得两个 `Account` 锁之前，就要获得这个“加时赛”锁，这样我们就能保证一次只有一个线程执行这个有风险的操作以未知的顺序获得锁，从而减小死锁发生的可能性（只要坚持使用这个机制）。如果经常出现哈希冲突，那么这个技术可能会成为并发性的瓶颈（就好像是一个单独的进程级的锁发挥的作用），但是因为 `System.identityHashCode` 的哈希冲突出现频率很低，所以这个技术以最小的代价，换来了最大的安全性。

清单 10.3 制定锁的顺序来避免死锁

```
private static final Object tieLock = new Object();

public void transferMoney(final Account fromAcct,
                          final Account toAcct,
                          final DollarAmount amount)
    throws InsufficientFundsException {
    class Helper {
        public void transfer() throws InsufficientFundsException {
            if (fromAcct.getBalance().compareTo(amount) < 0)
                throw new InsufficientFundsException();
            else {
                fromAcct.debit(amount);
                toAcct.credit(amount);
            }
        }
    }
    int fromHash = System.identityHashCode(fromAcct);
    int toHash = System.identityHashCode(toAcct);

    if (fromHash < toHash) {
        synchronized (fromAcct) {
            synchronized (toAcct) {
                new Helper().transfer();
            }
        }
    } else if (fromHash > toHash) {
        synchronized (toAcct) {
            synchronized (fromAcct) {
                new Helper().transfer();
            }
        }
    } else {
        synchronized (tieLock) {
            synchronized (fromAcct) {
                synchronized (toAcct) {
                    new Helper().transfer();
                }
            }
        }
    }
}
```

如果 Account 具有一个唯一的, 不可变的, 并且具有可比性的 key, 比如账号, 那么制定锁的顺序就更加容易了: 通过 key 来排定对象顺序, 这样能省去“加时赛”锁的需要。

你可能认为我们夸大了死锁的风险, 因为占有锁的时间通常很短暂, 但是死锁在真实系统中是很严重的问题。一个程序成为产品每天可能出现数十亿获取锁和释放锁的循环。只要其中有一项需要考虑时序的地方发生错误, 就可能导致程序发生死锁, 而且即使通过了压力测试也不可能找出所有潜在的死锁¹。清单 10.4²中 DemonstrateDeadlock 在大多数系统下都会很快发生死锁。

清单 10.4 开始一个循环, 它在典型条件下制定死锁

```
public class DemonstrateDeadlock {
    private static final int NUM_THREADS = 20;
    private static final int NUM_ACCOUNTS = 5;
    private static final int NUM_ITERATIONS = 1000000;

    public static void main(String[] args) {
        final Random rnd = new Random();
        final Account[] accounts = new Account[NUM_ACCOUNTS];

        for (int i = 0; i < accounts.length; i++)
            accounts[i] = new Account();

        class TransferThread extends Thread {
            public void run() {
                for (int i=0; i<NUM_ITERATIONS; i++) {
                    int fromAcct = rnd.nextInt(NUM_ACCOUNTS);
                    int toAcct = rnd.nextInt(NUM_ACCOUNTS);
                    DollarAmount amount =
                        new DollarAmount(rnd.nextInt(1000));
                    transferMoney(accounts[fromAcct],
                                accounts[toAcct], amount);
                }
            }
        }

        for (int i = 0; i < NUM_THREADS; i++)
            new TransferThread().start();
    }
}
```

¹ 具有讽刺意味的是, 短暂持有锁, 目的是减少锁的竞争, 但却增加了在测试中发现潜在死锁风险的难度。

² 为了简化问题, DemonstrateDeadlock 忽视了账户负数余额的问题。

10.1.3 协作对象间的死锁

获取多重的锁并不总是像在 `LeftRightDeadlock` 或者 `transferMoney` 中那么明显；可能不是在同一种方法中请求两个锁。思考清单 10.5 中相互协作的类，它可能用于出租车调遣系统。`Taxi` 代表个体出租车，具有位置和方向两个属性；`Dispatcher` 代表一组出租车。

尽管没有方法显式地获得两个锁，不同的 `setLocation` 和 `getImage` 的调用者同样可以获得两个锁。如果一个线程调用 `setLocation` 作为对 GPS 接收器更新的响应，它首先更新出租车的位置，然后检查其是否到达了目的地，如果已经到达，那么它会通知 `Dispatcher`，它需要一个新的目标。因为 `setLocation` 和 `notifyAvailable` 都是 `synchronized` 方法，调用 `setLocation` 的线程获取了 `Taxi` 的锁，然后又获取了 `Dispatcher` 的锁。类似地，调用 `getImage` 的线程先获取了 `Dispatcher` 锁，然后再获取每一个 `Taxi` 的锁（一次一个）。正如在 `LeftRightDeadlock` 中发生的，两个锁被两个线程以不同的顺序占有，产生死锁风险。

在 `LeftRightDeadlock` 或 `transferMoney` 中，发现死锁存在的可能性是比较简单的，只需要寻找获得两个锁的方法。在 `Taxi` 和 `Dispatcher` 中发现死锁的可能性就不那么容易了：如果外部（*alien*，在 40 页中定义）方法被调用时正持有锁，则是该死锁的一种警示。

在持有锁的时候调用外部方法是在挑战活跃度问题。外部方法可能会获得其他锁（产生死锁的风险），或者遭遇严重超时的阻塞。当你持有锁的时候会延迟其他试图获得该锁的线程。

10.1.4 开放调用

当然，`Taxi` 和 `Dispatcher` 并不知道它们马上就会陷入死锁中。它们本不该这样：一个方法调用是一个抽象的关卡，用来屏蔽你确切了解另外一方发生了什么。但是由于你并不能知道调用的另一方所发生的事情，**在持有锁的时候调用一个外部方法很难进行分析，因此是危险的。**

当调用的方法不需要持有锁时，这被称为**开放调用**（open call）[CPJ 2.4.1.3]，依赖于开放调用的类会具有更好的行为，并且比那些需要获得锁才能调用的方法相比，更容易与其他类合作。使用开放调用来避免死锁类似于使用封装来提供线程安全：尽管我们能够保证在没有封装的情况下构建线程安全的程序，但是对一个有效封装的类进行线程安全分析，要比分析没有封装的类容易得多。类似地，分析一个完全依赖于开放调用的程序的程序活跃度，比分析那些非开放调用的程序更简单。尽量让你自己使用开放调用，这要比获

清单 10.5 协作对象间的锁顺序死锁（不要这样做）

// 警告：可能产生死锁！

```
class Taxi {
    @GuardedBy("this") private Point location, destination;
    private final Dispatcher dispatcher;

    public Taxi(Dispatcher dispatcher) {
        this.dispatcher = dispatcher;
    }

    public synchronized Point getLocation() {
        return location;
    }

    public synchronized void setLocation(Point location) {
        this.location = location;
        if (location.equals(destination))
            dispatcher.notifyAvailable(this);
    }
}

class Dispatcher {
    @GuardedBy("this") private final Set<Taxi> taxis;
    @GuardedBy("this") private final Set<Taxi> availableTaxis;

    public Dispatcher() {
        taxis = new HashSet<Taxi>();
        availableTaxis = new HashSet<Taxi>();
    }

    public synchronized void notifyAvailable(Taxi taxi) {
        availableTaxis.add(taxi);
    }

    public synchronized Image getImage() {
        Image image = new Image();
        for (Taxi t : taxis)
            image.drawMarker(t.getLocation());
        return image;
    }
}
```



得多重锁后识别代码路径更简单，因此可以确保用一致的顺序获得锁³。

清单 10.5 的 Taxi 和 Dispatcher 可以很容易被重构到使用开放调用，因此可以减小死锁风险。这需要我们减少 `synchronized` 块的使用，只需要守护那些调用共享状态的操作，如清单 10.6 中所示。如果仅仅是出于语法紧凑或简单性的原因，而不是因为整个方法必须被锁保护，就使用 `synchronized` 方法取代一个更小的 `synchronized` 块，这通常就会导致出现清单 10.5 中的问题。（作为附加收益，缩减 `synchronized` 块还能够改善可伸缩性；11.4 节给出了关于确定 `synchronized` 块大小的建议。）

在程序中尽量使用开放调用。依赖于开放调用的程序，相比于那些在持有锁的时候还调用外部方法的程序，更容易进行死锁自由度（`deadlock-freedom`）的分析。

重新构建 `synchronized` 块有时会使开放调用产生出乎预料的结果，因为它使得一个原子操作变为非原子操作。有时候，这样损失了原子性的操作是完全可以接受的；更新出租车的位置，并通知分派程序这辆车已经准备好随时待命，这两件事情并不需要成为原子操作。不过有时候损失操作的原子性是值得关注的，但是这样造成的语意变化仍然是可以接受的。在易产生死锁的版本中，`getImage` 会产生当时瞬间的所有 `location` 的快照；在重构后的版本中，`getImage` 获得的是在略微不同时间下的每辆出租车的位置。

然而，有的时候原子性的损失却会引发问题，这里你需要使用另一种技术来实现原子性。构建一个并发的对象就是这些解决技术的一种，这样做能够让唯一的线程执行开放调用的代码路径。例如，在关闭一个服务的时候，你可能想要等待运行中的操作完成，在这之后释放服务占用的资源。等待操作完成的同时把持服务的锁，这是很容易导致死锁的，但是在服务关闭前就释放服务的锁可能导致其他线程开启新的操作。解决的方法是，在把服务状态更新为“关闭”之前一直持有锁，这样其他的线程想要开始新的操作——包括关闭服务——发现服务已经不可用，就不会再尝试了。然后，你可以等待结束关闭，并且清楚地知道，在开放调用之后，只有关闭线程的操作访问了服务状态。因此，这项技术依赖于构建的协议，能够防止其他线程尝试进入，这胜于使用锁来避免其他线程进入代码的临界区。

10.1.5 资源死锁

当线程间相互等待对方持有的锁，并且谁都不会释放自己的锁时就会发生死锁，当线程持有和等待的目标变为资源时，会发生与之相类似的死锁。假设你有两个放入池中的资

³ 对开放调用的依赖，以及对锁排序的细心关注，这些反映了在组装一个同步对象的过程中要面对的麻烦，而并未反映出一个组合同步对象的复杂度。

清单 10.6 使用开放调用来避免协作对象之间的死锁

```
@ThreadSafe
class Taxi {
    @GuardedBy("this") private Point location, destination;
    private final Dispatcher dispatcher;
    ...
    public synchronized Point getLocation() {
        return location;
    }

    public synchronized void setLocation(Point location) {
        boolean reachedDestination;
        synchronized (this) {
            this.location = location;
            reachedDestination = location.equals(destination);
        }
        if (reachedDestination)
            dispatcher.notifyAvailable(this);
    }
}

@ThreadSafe
class Dispatcher {
    @GuardedBy("this") private final Set<Taxi> taxis;
    @GuardedBy("this") private final Set<Taxi> availableTaxis;
    ...
    public synchronized void notifyAvailable(Taxi taxi) {
        availableTaxis.add(taxi);
    }

    public Image getImage() {
        Set<Taxi> copy;
        synchronized (this) {
            copy = new HashSet<Taxi>(taxis);
        }
        Image image = new Image();
        for (Taxi t : copy)
            image.drawMarker(t.getLocation());
        return image;
    }
}
```

源，比如分别是到两个数据库的连接池。资源池通常通过信号量实现（参见第 5.3.3 节），当池为空的时候发生阻塞。如果一个任务需要连接到两个数据库，并且两个资源并不是按照相同顺序进行调用的，线程 A 可能持有至数据库 D1 的连接，并等待连接到数据库 D2，而线程 B 持有至 D2 的连接并等待到 D1 的连接。（线程池越大，这种情况发生的可能性就越小，如果每个线程池都有 N 个连接，死锁的发生需要 N 套循环等待的线程，并且偶发时序多次产生。）

另一种形式的基于资源的死锁是**线程饥饿死锁**（thread-starvation deadlock）。我们在第 8.1.1 节见到过这种危害的一个示例，一个任务将工作提交到单线程化的 Executor，并等待该 Executor 中的其他任务返回它的结果。在这样的情况下，第一个任务将永远会等待下去，使这个任务以及所有在 Executor 中执行的其他任务永久停滞下来。需要等待其他任务的结果的任务是生成线程饥饿死锁的来源；有界池和相互依赖的任务不能放在一起使用。

10.2 避免和诊断死锁

如果一个程序一次至多获得一个锁，那么就不会产生锁顺序死锁。当然，这通常并不现实，但是如果你能够避免这个情况，就能够省去很多工作。如果你必须获得多个锁，那么锁的顺序必须是你设计工作的一部分：尽量减少潜在锁之间的交互数量，遵守并文档化该锁顺序协议，这些缺一不可。

在使用定义良好的锁的程序中，监测代码中死锁自由度的策略分为两个部分：首先识别什么地方会获取多个锁(使这个集合尽量小)，对这些示例进行全局的分析，确保它们锁的顺序在程序中保持一致。尽可能使用开放调用，这样做能够从根本上简化分析的难度。在没有非开放调用的程序中，发现那些获得多重锁的实例是非常简单的，既可以通过复查代码，也可以通过对字节码或源代码自动地进行分析来完成。

10.2.1 尝试定时的锁

另一项监测死锁和从死锁中恢复的技术，是使用每个显式 Lock 类中定时 tryLock 特性（参见第 13 章），来替代使用内部锁机制。在内部锁的机制中，只要没有获得锁，就会永远保持等待，而显式的锁使你能够定义超时（timeout）的时间，在规定时间之后 tryLock 还没有获得锁就返回失败。通过使用超时，尽管这段时间比你预期能够获得锁的时间长很多，你仍然可以在意外发生后重新获得控制权。（参见第 280 页，清单 13.3 表现了 transferMoney 的另一种实现，其中使用可轮询检查的 tryLock 基本上避免了死锁发生的可能性）。

当尝试获得定时锁失败时，你并不需要知道**原因**。也许是因为有死锁发生，也许是线程在持有锁的时候错误地进入了无限循环；也可能是执行一些活动所花费的时间比你的预期慢了许多。不过至少你有机会了解到你的尝试已经失败，记录下这次尝试中有用的信息，并重新开始计算，这远比关闭整个进程要优雅得多。

即使定时锁并没有应用于整个系统中，使用它来获得多重锁还是能够有效应对死锁。如果获取锁的请求超时，你可以释放这个锁，并后退，等待一会儿后再尝试，这很可能消除了死锁发生的条件，并且允许程序恢复。（这项技术只有在同时获得两个锁的时候才有效；如果多个锁是在嵌套的方法中被请求的，你无法仅仅释放外层的锁，尽管你知道自己已经持有该锁。）

10.2.2 通过线程转储分析死锁

预防死锁是你面临的最大问题，JVM 使用**线程转储**（thread dump）来帮助你识别死锁的发生。线程转储包括每个运行中线程的栈追踪信息，以及与之相似并随之发生的异常。线程转储也包括锁的信息，比如，哪个锁由哪个线程获得，其中获得这些锁的栈结构，以及阻塞线程正在等待的锁究竟是哪一个⁴。在生成线程转储之前，JVM 在表示“正在等待（is-waiting-for）”关系的（有向）图中搜索循环来寻找死锁。如果发现了死锁，它就会包括死锁的识别信息，其中参与了哪些锁和线程，以及程序中造成不良后果的锁请求发生在哪里。

在 Unix 平台，你可以通过向 JVM 的进程发送 SIGQUIT 信号（kill -3）来触发线程转储，或者在 Unix 平台按下 Ctrl-\ 键，在 Windows 平台按下 Ctrl-Break 键。很多 IDE（集成开发环境）都要求线程转储。如果你使用了显式的 Lock 类，而不是内部的锁，Java 5.0 并不支持与 Lock 相关的转储信息；显式的 Lock 并不会直接获得转储。Java 6 中，显式的 Lock 支持线程转储和死锁的监测，但是其中获得的 Lock 的信息与内部锁提供的信息相比，却粗略了很多。内部锁与获得它的线程的栈框架相关联；显式的 Lock 只会与获得它的线程相关联。

清单 10.7 表现了在一个生产环境下，J2EE 应用程序产生的线程转储的一部分。死锁导致的失败牵涉到 3 个组件——一个 J2EE 应用，一个 J2EE 容器，以及一个 JDBC 驱动，分别由不同的供应商提供。（为避免尴尬，它们的名字已经被修改）。这 3 个组件均为商业产品，都经过了大量的测试，但每一个都存在潜在的 bug，直到它们进行交互的时候才会体现出来，从而导致服务器致命的失败。

我们已经展示了线程转储信息中，关于死锁的一部分。JVM 在诊断死锁方面可以为我们做很多事情——哪些锁引发了问题，哪些线程参与其中，它们持有哪些其他的锁，是否

⁴ 即使你没有遇到死锁，这些信息在调试中也是有用的；周期性的触发线程转储可以让你观察到程序加锁的行为。

清单 10.7 发生死锁后线程转储的部分信息

```

Found one Java-level deadlock:
=====
"ApplicationServerThread":
  waiting to lock monitor 0x080f0cdc (a MumbleDBConnection),
  which is held by "ApplicationServerThread"
"ApplicationServerThread":
  waiting to lock monitor 0x080f0ed4 (a MumbleDBCallableStatement),
  which is held by "ApplicationServerThread"

Java stack information for the threads listed above:
"ApplicationServerThread":
  at MumbleDBConnection.remove_statement
  - waiting to lock <0x650f7f30> (a MumbleDBConnection)
  at MumbleDBStatement.close
  - locked <0x6024ffb0> (a MumbleDBCallableStatement)
  ...

"ApplicationServerThread":
  at MumbleDBCallableStatement.sendBatch
  - waiting to lock <0x6024ffb0> (a MumbleDBCallableStatement)
  at MumbleDBConnection.commit
  - locked <0x650f7f30> (a MumbleDBConnection)
  ...

```

给其他线程造成间接的不便。一个线程持有 `MumbleDBConnection` 的锁，并等待获得守护 `MumbleDBCallableStatement` 的锁；另一个持有 `MumbleDBCallableStatement` 的锁，并等待 `MumbleDBConnection` 的锁。

这里使用的 JDBC 驱动明显存在一个锁顺序 bug：不同的调用链通过 JDBC 驱动程序以不同的顺序获取多个锁。但是这个问题如果没有另一个 bug 存在的话是永远都不会显现出来的：多个线程企图同时使用相同的 JDBC Connection。这并不是应用程序设计的本意——程序员很惊讶地发现相同的 Connection 被两个线程同时使用。JDBC 的规约中并没有提到 Connection 的线程安全性，通常把 Connection 的使用限制在单线程中，在这里被作为自然而然的事情。供应商试图去交付一个线程安全的 JDBC 驱动，因此在代码内部对多个 JDBC 对象加入了同步机制。不幸的是，供应商并没有考虑锁的顺序，使得驱动很容易发生死锁。恰好是易发生死锁的驱动程序和程序中不正确的 Connection 共享，这两者之间的互动暴露了这个问题。因为其中任何一个 bug 都不能单独产生致命错误，即使分别进行了大量测试也不会出现问题。

10.3 其他的活跃度危险

尽管死锁是我们遇到的最主要的活跃度危险，并发程序中仍然可能遇到一些其他的活跃度危险，包括：饥饿，丢失信号和活锁。（丢失信号将在 14.2.3 节中介绍。）

10.3.1 饥饿

当线程访问它所需要的资源时却被永久拒绝，以至于不能再继续进行，这样就发生了**饥饿**（starvation）；最常见的引发饥饿的资源是 CPU 周期。在 Java 应用程序中，使用线程的优先级不当可能引起饥饿。在锁中执行无终止的构建也可能引起饥饿（无限循环，或者无尽等待资源），因为其他需要这个锁的线程永远不可能得到它。

线程 API 定义的线程优先级仅仅作调度的参考。线程 API 定义了十个优先级级别，并对应到操作系统相应的调度优先级中。这样的映射是与平台相关的，所以两个 Java 优先级可能映射到某操作系统相同的优先级，也可能映射到另一操作系统的不同优先级中。有一些操作系统优先级别本身就少于 10 个，那么其中多个 Java 的优先级会映射到操作系统相同的优先级。

操作系统的调度程序努力提供公平的、活跃度良好的调度，甚至超越了 Java 语言规范（Java Language Specification）所要求的。在大多数 Java 应用程序中，所有线程都具有相同的优先级，`Thread.NORM_PRIORITY`。线程优先级并不是方便的工具，它改变线程优先级的效果往往不明显；提高一个线程的优先级往往什么都不能改变，或者总是会引起一个线程的调度优先级高于其他线程，从而导致饥饿。

通常情况下，抵制住改变线程优先级的诱惑是明智的。只要你开始改变线程的优先级，程序的行为就变为与平台相关的了，并且你会引入饥饿发生的风险。你经常能够发现程序试图从修改了优先级的问题中恢复，也可能是某些其他地方，为了尝试给低优先级线程分配更多的时间，偶然调用了 `Thread.sleep` 或 `Thread.yield` 造成的响应性问题⁵。

抵制使用线程优先级的诱惑，因为这会增加平台依赖性，并且可能引起活跃度问题。大多数并发应用程序可以对所有线程使用相同的优先级。

⁵ `Thread.yield` 的语意（以及 `Thread.sleep(0)`）是未定义的[JLS 17.9]；JVM 可以自由实现它们，比如 no-ops，或者作为调度的参考。尤其是在 Unix 系统中，`sleep(0)` 是没有意义的——根据优先级，把当前的线程放在运行队列的末尾，并产生与其他线程相同的优先级——尽管一些 JVM 就是这样实现 `yield` 的。

10.3.2 弱响应性

除饥饿以外的另一个问题是弱响应性，在 GUI 应用程序中使用后台线程的情况下，这是很常见的。第 9 章开发了一个框架，把耗时的任务分载到后台线程中，这样就不会造成用户界面的冻结了。CPU 密集型的后台任务仍然可能会影响响应性，因为它们会与事件线程共同竞争 CPU 的微周期。这里是线程优先级发挥作用的时候：当计算密集型后台计算任务会影响到响应性时，如果由其他线程完成的工作为真正的后台任务时，应该降低它们的优先级，从而使前台程序具有更佳的响应性。

不良的锁管理也可能引起弱响应性。如果一个线程长时间占有一个锁（可能正在对一个大容器进行迭代，并对每一个元素进行耗时的工作），其他想要访问该容器的线程就必须等待很长时间。

10.3.3 活锁

活锁（livelock）是线程中活跃度失败的另一种形式，尽管没有被阻塞，线程却仍然不能继续，因为它不断重试相同的操作，却总是失败。活锁通常发生在消息处理应用程序中，如果消息处理失败的话，其中传递消息的底层架构会回退整个事务，并把它置回队首。如果消息处理程序对某种特定类型的消息处理存在 bug，每次处理都会失败，那么每一次这个消息都会被从队列中取出，传递到存在问题的处理器（handler），然后发生事务回退。因为这条消息又会到队首，处理器会不断被这样重复调用，并返回重复结果。这就是通常称为**毒药信息**（poison message）的问题。信息处理线程并没有发生阻塞，但是永远都不会前进了。这种形式的活锁通常来源于过渡的错误恢复代码，误将不可修复的错误当作是可修复的错误。

活锁同样发生在多个相互协作的线程间，当它们为了彼此间响应而修改了状态，使得没有一个线程能够继续前进，那么就发生了活锁。这好比两个过于礼貌的人在半路相遇：他们都避开对方的路，于是在另一条路上又相遇了。所以就这样不停地一直避让下去了.....

解决这些多样的活锁的一种方案就是对重试机制引入一些随机性。例如，在以太网络上，两个机站尝试使用相同的载波发送数据包，包会发生冲突。机站发现了冲突，并且都在稍后重发。如果它们都非常精确地在一秒后重试，它们又会发生冲突，并不断冲突下去，导致数据包永远不能发送，即使有大量的带宽都是闲置的。为了避免这种情况发生，我们通过使用一个随机组件它们进行等待。（以太协议在重复发生冲突时，同样包含一个指数的撤销协议，减少了拥塞和多个冲突的机站重复失败的风险。）在并发程序中，通过随机

等待和撤回来进行重试能够相当有效地避免活锁的发生。

总结

活跃度失败是非常严重的问题，因为除了短时间地中止应用程序，没有任何机制可以恢复这种失败。最常见的活跃度失败是锁顺序死锁。应该在设计时就避免锁顺序死锁：确保多个线程在获得多个锁时，使用一致的顺序。最好的解决方法是在程序中使用开放调用。这会大大减少一个线程一次请求多个锁的情况，并且使这样的多重锁请求的发生更加明显。

性能和可伸缩性

Performance and Scalability

使用线程最主要的原因是提高性能¹。使用线程可以使程序更加充分地发挥出闲置的处理能力，从而更好地利用资源；并能够使程序在现有任务正在运行的情况下立刻开始着手处理新的任务，从而提高系统的响应性。

这一章将探讨用于并发程序性能的分析、监测和改进的技术。不幸的是，很多改进性能的技术同样增加了复杂度，因此增加了安全和活跃度失败的可能性。更糟糕的是，有些技术的目的是改进性能，事实上产生了相反的作用，带来了其他的性能问题。尽管我们希望获得更好的性能——改进性能带来了成就感——但安全总是第一位的。首先要保证程序是正确的，然后再让它更快——而后只有当你的性能需求和评估标准需要程序运行得更快时，才去进行改进。在设计并发应用程序的时候，最大可能地改进性能，通常并不是最重要的事情。

11.1 性能的思考

改进性能意味着用更少的资源做更多的事情。“资源”的概念很广泛，对于给定的活动而言，一些特定的资源通常非常缺乏，无论是 CPU 周期、内存、网络带宽、I/O 带宽、数据库请求、磁盘空间、以及其他一些资源。当活动的运行因某个特定资源受阻时，我们称之为**受限**于该资源：受限于 CPU，受限于数据库。

尽管目标是希望全面提升性能，与单线程方法相比，使用多线程总会引入一些性能的开销。这些开销包括：与协调线程相关的开销（加锁、信号、内存同步），增加的上下文

¹ 有人可能会争辩：这是我们引入复杂线程的唯一理由。

切换，线程的创建和消亡，以及调度的开销。当线程被过度使用后，这些开销会超过提高后的吞吐量响应性和计算能力带来的补偿。从另一方面，一个没能经过良好并发设计的应用程序，甚至比相同功能的顺序的程序性能更差²。

为了利用并发来实现更好的性能，我们需要努力做两件事情：更有效地利用我们现有的处理资源，让我们的程序尽可能地开拓更多可用的处理资源。从性能监视器的视角来看，这意味着我们期望使 CPU 尽可能处于忙碌状态。（当然，这并不是让 CPU 周期忙于应付无用的计算；我们希望 CPU 做**有用的**事情。）如果程序是受限于计算能力的，那么我们通过增加更多的处理器就能够提高生产量；如果程序都不能保持现有的处理器处于忙碌工作的状态，添加更多的处理器也无济于事。线程通过分解应用程序，总是让空闲的处理器进行未完成的工作，从而保持所有 CPU “热火朝天”地工作，

11.1.1 性能“遭遇”可伸缩性

应用程序可以从很多个角度来衡量；比如服务时间、等待时间、吞吐量、效率、可伸缩性、生产量。有一些标准（服务时间、等待时间）是用来衡量“有多快”，即给定的任务单元需要多长时间进行处理，得到回馈；另一些（生产量、吞吐量）用来衡量“有多少”，即限定计算资源的情况下，究竟能够完成多少工作。

可伸缩性指的是：当增加计算资源的时候（比如增加额外 CPU 数量、内存、存储器、I/O 带宽），吞吐量和生产量能够相应地得以改进。

在传统的性能调优中，为配合可伸缩性来设计和调试并发应用程序是非常困难的。为并发而进行的调试，其目的通常是用**最小的**代价完成**相同**的工作，比如通过缓存来重用以前计算的结果，或者用时间复杂度为 $O(n \log n)$ 算法取代 $O(n^2)$ 算法。在为可伸缩性进行调试的时候，你的目的是如何并行化你的问题，使你能够利用额外的计算资源，用**更多的**资源做**更多**的事情。

性能的这两个方面——“**有多快**”和“**有多少**”是完全分离的，有时候甚至是相悖的。为了实现更好的可伸缩性，或者更好地利用硬件，我们通常会停止**增加**每个**独立**任务所要完成的工作量，比如我们把任务分解到多个管道线的子任务中。据有讽刺意味的是，大多数在单线程化的程序中提高性能的窍门，都会损害可伸缩性（实例参见 11.4.4）。

² 一个同事告诉我这样一件有趣的事情：他曾经参与了一个复杂、耗资巨大的应用程序的测试，这个程序是通过可调节的线程池进行管理的。在系统结束时，测试结果显示，线程池最优的线程数是...1。这里的问题显然出在一开始；目标系统是单CPU系统，程序几乎完全受限于CPU。

我们所熟知的程序的三层（tier）模型——其中的表现层、业务逻辑层和持久层是分离的，并且可能由不同的系统掌控——这个例子阐明了提高可伸缩性是如何造成性能损失的。把表现层、业务逻辑层、持久层拼装到同一个程序中，相比于在多个系统间进行了良好分解的分布式实现，前者在完成**首个**任务单元的性能上要高出许多。那劣势呢？即使能够在同一个应用程序的不同层之间传递任务，并且不存在网络的延迟，仍然要把对计算的处理清晰地分离到各个抽象层（layer）中（比如排队的开销，协调的开销，和数据拷贝），并为此付出代价。

然而，当单一的系统到达它处理的极限时，会遇到一个严重的问题：提升它的处理能力会相当困难。所以，我们通常会接受更长处理时间的性能开销，或者为每个任务单元分配更多的计算资源，从而使应用程序能够通过增加更多的资源来相应承担更大的负荷。

从性能中的多个角度来看，“有多少”方面——可伸缩性，吞吐量和生产量——在 Server 应用程序中往往比“有多快”受到更多的关注。（在交互式应用中，对等待时间的关注可能更加重要，这样用户就不用等待进度条的显示，也不需要知道系统究竟在做什么。）这一章主要集中在可伸缩性而不是原始的单线程化系统的性能。

11.1.2 对性能的权衡进行评估

几乎所有的工程上的决定都会遇到某些形式的折中。在建设桥跨时使用更粗的钢筋可以提高桥的负载能力和安全性，却同时会提高建造成本。尽管软件工程的决定通常不会遇到金钱和事关人类生命的风险这两者之间的抉择，但是我们经常会缺少那些能帮助我们作出正确权衡的信息。例如，“快速排序”算法对于大的数据集具有非常好的效率，但是我们熟知的“冒泡排序”对小数据集非常有效。如果你想要为实现排序例程选择一个算法，你需要知道面临处理的数据集的大小，还有你试图进行优化的目标：是平均计算时间，允许的最差时间，还是可预知性。不幸的是，库排序例程的作者所拿到的需求里面往往没有包括这些信息。这就是为什么大多数优化都不成熟的原因之一：**他们通常在获得清晰的需求之前进行了假设。**

避免不成熟的优化。首先使程序正确，然后再加快——如果它运行得还不够快。

当我们面临工程上的决定时，有时候会用某种形式的成本换取其他东西（用内存开销换取更短的服务时间）；有时候也会用开销换取安全性。安全性并不完全指对人们生活的威胁，比如桥梁的那个例子。很多性能的优化会损害可读性或可维护性——代码越“聪

明”，越“晦涩”，就越难理解和维护。有时候，优化需要违背好的面向对象的设计原则，比如打破封装；有时候，它们会带来很大的风险和错误，因为通常越快的算法越复杂。（如果你不能识别代价或者风险，你可能还没能对将发生的场景进行彻底、仔细地思考。）

大多数性能的决定需要多个变量，并且高度依赖于发生的环境。在决定某个方案比其他方案“更快”之前，先问你自己一些问题：

- 你所谓的更“快”指的是什么？
- 在什么样的条件下你的方案能够**真正**运行得更快？在轻负载还是重负载下？大数据集还是小数据集？是否支持你的测量标准的答案？
- 这些条件在你的环境中发生的频率？是否支持你的测量标准的答案？
- 这些代码在其他环境的不同条件下被用到的可能性？
- 你用什么样隐含的代价，比如增加的开发风险或维护性，换取了性能的提高？这个权衡的决定是否正确？

作出任何与性能相关的工程决定时，都应该考虑这些问题，但是这本书只关注于并发。我们为什么要推荐这样一个保守的优化方案？**对性能的追求很可能是并发 bug 唯一最大的来源**。认为同步“太慢”而导致使用看似聪明实际危险的手法，从而减少同步（比如第 16.2.4 小节将要讨论的双检查锁），这也成为了不遵守同步规定常用的一个借口。然而，因为并发的 bug 是最难追踪和消除的缺陷，所以任何引入这类 bug 的风险行动都需要慎重进行。

更糟的是，当你用安全性换取了性能的时候，你可能什么都没得到。特别是，当提到并发的时候，很多开发者对于产生性能问题的原因，或者哪一个方案能够更迅速，具有更好的可伸缩性，他们的直觉往往是错误的。因此，依据性能的需求（这样你能知道什么时候需要调节，什么时候该**停止**），根据适当的评估纲要，并使用现实中的配置和负载状况，来进行性能调节的活动是非常必要的。在调节过后，你需要再评估，以验证你已经实现了期望的改进。优化带来的安全性和可维护性风险足够严重，以至于——如果你不需要的話，你不想付出这样的代价。并且，如果你甚至没有从中得到一点好处，你绝对不希望付出此代价。

测评。不要臆测。

市场上有一些成熟的剖析工具，用来评估性能，追踪性能瓶颈，但是你不必花费大量的金钱用于了解你的程序做了什么。例如，免费的 `perfbars` 应用程序可以给你一张相当不错的图表，告诉你 CPU 究竟是如何忙碌地工作，并且你的目标通常是保持 CPU 的忙碌，这便是一个很好的方式，使你能够评估你是否需要性能调节，或者你调节的效果如何。

11.2 Amdahl 定律

有些问题使用越多的资源就能越快地解决——越多的工人参与收割庄稼，那么就能越快地完成收获。另一些任务根本就是串行化的——增加更多的工人根本不可能提高收割速度。如果我们使用线程的重要原因之一是为了支配多处理器的能力，我们必须保证问题被恰当地进行了并行化的分解，并且我们的程序有效地使用了这种并行的潜能。

大多数并发程序都与农耕有着很多相似之处，由一系列并行和串行化的片断组成。**Amdahl 定律**描述了在一个系统中，基于可并行化和串行化的组件各自所占的比重，程序通过获得额外的计算资源，理论上能够加速多少。如果 F 是必须串行化执行的比重，那么 Amdahl 定律告诉我们，在一个 N 处理器的机器中，我们最多可以加速：

$$Speedup \leq \frac{1}{F + \frac{(1-F)}{N}}$$

当 N 无限增大趋近无穷时， $speedup$ 的最大值无限趋近 $1/F$ ，这意味着一个程序中如果 50% 的处理都需要串行进行的话， $speedup$ 只能提升 2 倍（不考虑事实上有多少线程可用）；如果程序的 10% 需要串行进行， $speedup$ 最多能够提高近 10 倍。Amdahl 定律同样量化了串行化的效率开销。在拥有 10 个处理器的系统中，程序如果有 10% 是串行化的，那么最多可以加速 5.3 倍（53% 的使用率），在拥有 100 个处理器的系统中，这个数字可以达到 9.2（9% 的使用率）。这使得无效的 CPU 利用永远不可能到达 10 倍。

图 11.1 展示了随着串行执行和处理器数量变化，处理器最大限度的利用率的曲线。随着处理器数量的增加，我们很明显地看到，即使串行化执行的程度发生细微的百分比变化，都会大大限制吞吐量随计算资源增加。

第 6 章探究了如何识别逻辑边界，从而把应用程序分解为不同的任务。但是为了在多处理器系统中预知你的程序是否存在加速的可能性，你同样需要识别你的任务中串行的部分。

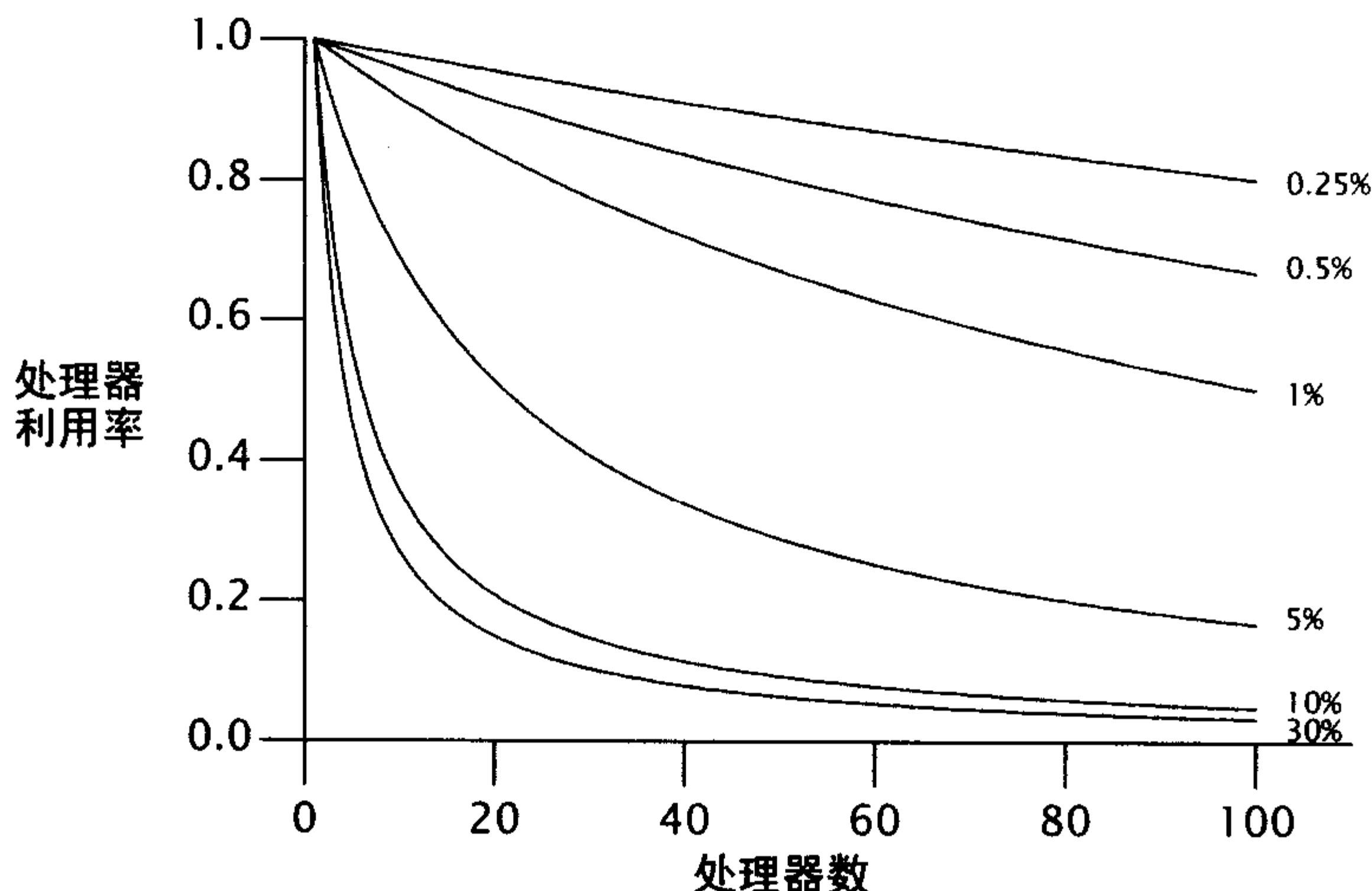


图 11.1 Amdahl 定律中不同串行化的百分比，带来的最大的效能

清单 11.1 中，假设应用程序中 N 个线程正在执行 `doWork`，从一个共享的工作队列中取出任务，并处理；假设这里的任务并不依赖其他任务的结果或边界效应。忽略任务进行队列操作的时间，如果我们增加处理器，应用程序会随之发生什么样的改进呢？乍看这个程序可能完全由并行任务组成，并不会相互等待，那么处理器越多，更多的任务就越可能并发处理。然而，其中也包含串行组件——从队列中获取任务。所有工作者线程都共享工作队列，因此它会需要一些同步机制，从而在并发访问中保持完整性。如果通过加锁来守卫队列状态，那么当一个线程从队列中取出任务的时候，其他线程想要取得下一个任务就必须等待——这便是任务处理中串行的部分。

单个任务的处理时间不仅包括执行任务 `Runnable` 的时间，也包括从共享队列中取出任务的时间。如果工作队列是 `LinkedBlockingQueue` 类型的，这个取出的操作被阻塞的可能性小于使用同步的 `LinkedList` 的阻塞可能，这是因为 `LinkedBlockingQueue` 使用了更具伸缩性的算法，但是访问所有共享的数据结构，本质上都会向程序引入一个串行的元素。

这个例子同样忽略了另一个的相同的串行源（source of serialization）：结果处理。所有有用的计算都产生一些结果集或者边界效应——如果不是，它们可以当作死代码（dead code）被遗弃掉。因为 `Runnable` 没有提供明确的结果处理，这些任务必须具有一些边界效应，设定把它们的结果写入日志还是存入一个数据结构。日志文件和结果容器通常由多

清单 11.1 串行访问任务队列

```
public class WorkerThread extends Thread {
    private final BlockingQueue<Runnable> queue;

    public WorkerThread(BlockingQueue<Runnable> queue) {
        this.queue = queue;
    }

    public void run() {
        while (true) {
            try {
                Runnable task = queue.take();
                task.run();
            } catch (InterruptedException e) {
                break; /* 允许线程退出 */
            }
        }
    }
}
```

个工作者线程共享，并且因此成为了同源的串行部分。如果不是每个线程各自维护自己的结果的数据结构，而是在所有任务都执行完成后合并所有的结果，这最终的合并就成为了一个串行源。

所有的并发程序都有一些串行源；如果你认为你没有，那么去仔细检查吧。

11.2.1 示例：框架中隐藏的串行化

为了观察并行化如何被隐藏在应用程序的架构中，我们可以比较加入线程时的吞吐量，基于观察到的可伸缩性变化来推断串行源。图 11.2 展示了一个简单的应用程序中，多个线程重复从共享 Queue 中移出元素，并处理它们，与清单 11.1 类似。处理的步骤只需要线程本地的计算。如果有一个线程发现队列是空的，它会向队列置入一批新的元素，这样其他的线程在下一次迭代中就不会无事可做。访问共享的队列显然要承担一定程度的串行化，但是处理步骤是完全并行化的，因为它不会引用共享数据。

图 11.2 的曲线比较了两个均为线程安全 Queue 的实现：synchronizedList 包装了 LinkedList，另一个是 ConcurrentLinkedQueue。测试跑在 8-way Sparc V880，OS 为

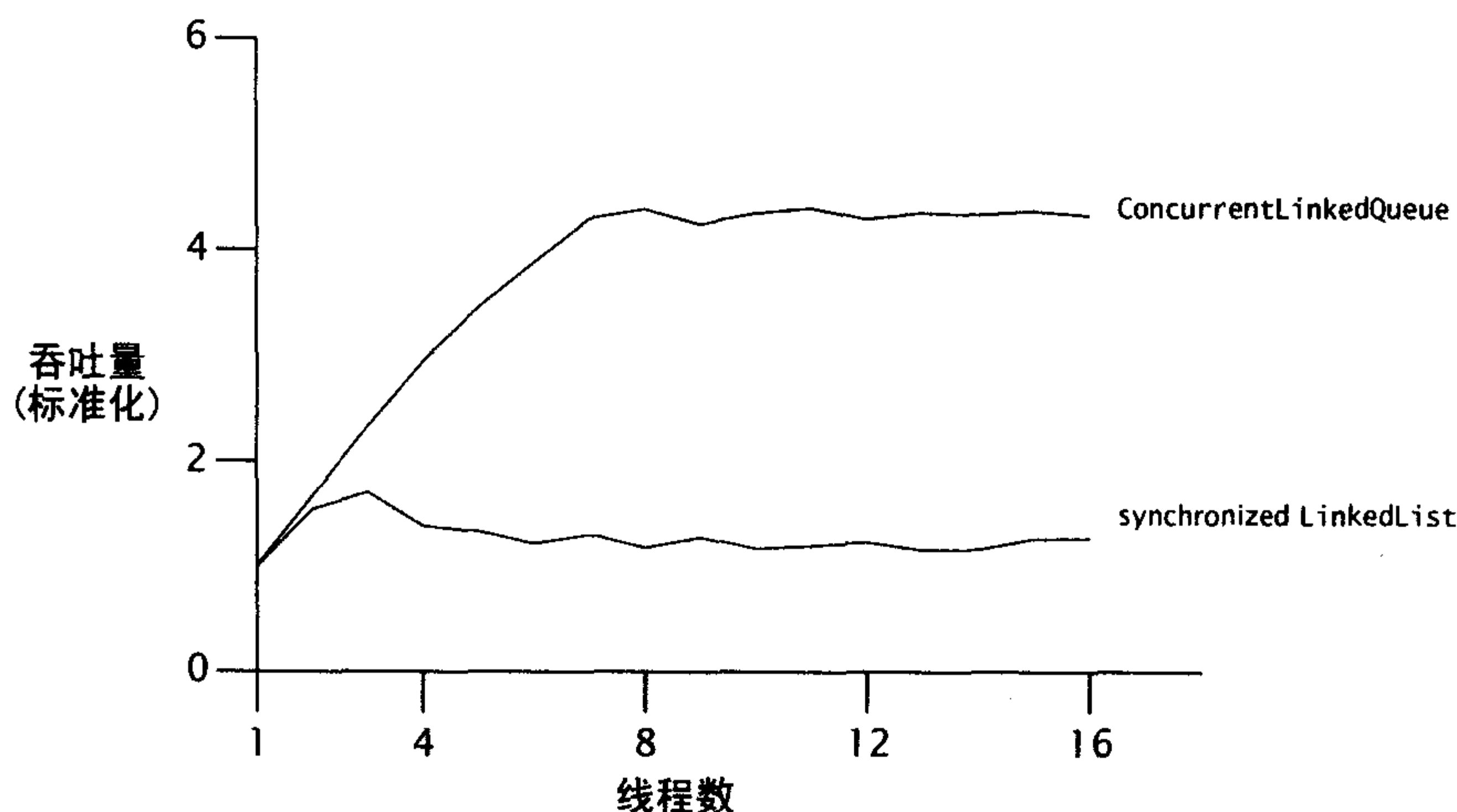


图 11.2 不同队列实现之间的比较

Solaris 的系统上。尽管每一次运行代表相同数量的“工作”，我们能够看到，只有改变队列的实现，才能明显影响可伸缩性。

ConcurrentLinkedQueue 的吞吐量持续改进，直到它到达了处理器数量，之后会保持不变。另一方面同步 LinkedList 的吞吐量，在 3 个线程时表现了其带来的改进，但是之后会下跌，因为同步的开销增加了。图中当线程数为 4 或 5 时，竞争是非常激烈的，以至于每次访问队列都要竞争锁，并且吞吐量受控于上下文切换的次数。

吞吐量的不同源自于两个队列实现的串行化不同。同步的 LinkedList 用一个锁守护着整个队列的状态，在 offer 和 remove 调用时都要获取这个锁；ConcurrentLinkedQueue 使用了精妙的非阻塞队列算法（参见 15.4.2 小节），它使用了原子引用来更新各个链接指针。这两者，其中一个是把整个的插入或删除都实现为串行化的，而另一个则是把每个指针的更新变成串行化的。

11.2.2 定性地应用 Amdahl 定律

如果我们能精确估算出执行中串行部分所占的比重，Amdahl 定律量化了使用更多资源时加速的可能性。尽管直接衡量串行化非常困难，Amdahl 定律在没有这样的衡量的情况下仍然有用。

因为我们的理想模型受我们所在环境的影响，我们中间很多人都习惯性地认为多处理器系统具有 2 个或 4 个处理器，或者可能（如果我们进行大胆假设）有几打（a few dozen），因为这是近年来技术上普遍可以达到的。但是随着多核 CPU 成为主流，系统将具有成百

上千个处理器³。看上去适合于4路系统的算法可能含有隐藏的可伸缩性瓶颈，只不过还没有遇到而已。

当我们评估一个算法的时候，考虑其在成百甚至上千个处理器的情况下受到的限制，能够帮助我们洞察伸缩性的极限的出现。例如，11.4.2和11.4.3两节中探讨了两种技术，用来减小锁的粒度：分拆锁（把一个锁分拆成两个），分离锁（把一个锁分拆成多个锁）。透过Amdahl定律来审视它们，我们发现把一个锁分拆成两个，看上去没能在利用多处理器上帮助我们很多，但是分离锁的效果却很好，因为分离出的数量可随着处理器数量的增加而增长。（当然，性能优化应该总是依据真实的性能需求；有时候，把一个锁分拆成两个足可以满足需求了。）

11.3 线程引入的开销

单线程程序既不存在调度问题，也不存在同步的开销，不需要使用锁来保证数据结构的一致性。调度和线程内部的协调都要付出性能的开销；对于性能改进的线程来说，并行带来的性能优势必须超过并发所引入的开销。

11.3.1 切换上下文

如果主线程是唯一可调度的线程，它决不会被排除在调度之外。从另一方面看，如果可运行的线程数大于CPU的数量，那么OS最终会强行换出正在执行的线程，从而使其他线程能够使用CPU。这会引起上下文切换，它会保存当前运行线程的执行上下文，并重建新调入线程的执行上下文。

切换上下文是要付出代价的；线程的调度需要操控OS和JVM中共享的数据结构。你的程序与OS、JVM使用相同的CPU；CPU在JVM和OS的代码花费越多时间，意味着用于你的程序的时间就越少。但是JVM和OS活动的花费并不是切换上下文开销的唯一来源。当一个新的线程被换入后，它所需要的数据可能不在当前处理器本地的缓存中，所以切换上下文会引起缓存缺失的小恐慌，因此线程在第一次调度的时候会运行得稍慢一些。即使有很多其他正在等待的线程，调度程序也会为每一个可运行的线程分配一个最小执行时间的定额。就是因为这个原因：它分期偿付切换上下文的开销，获得更多不中断的执行时间，从整体上提高了吞吐量（以损失响应性为代价）。

³ 最新行情：在写作过程中，Sun正在推广低端服务器系统基于8-核的Niagara处理器，并且Azul正在推广高端服务器系统（96、192和384路）基于24核的Vega处理器。

清单 11.2 徒劳的同步（不要这样做）

```
synchronized (new Object()) {  
    // 工作代码  
}
```



当线程因为竞争一个锁而阻塞时，JVM 通常会将这个线程挂起，允许它被换出。如果线程频繁发生阻塞，那线程就不能完整使用它的调度限额了。一个程序发生越多的阻塞（阻塞 I/O，等待竞争锁，或者等待条件变量），与受限于 CPU 的程序相比，就会造成越多的上下文切换，这增加了调度的开销，并减少了吞吐量。（无阻塞的算法同样能够帮助我们减小上下文切换；参见第 15 章。）

切换上下文真正的开销根据不同的平台而变化，但是一条好的经验性原则是：在大多数通用的处理器中，上下文切换的开销相当于 5 000 到 10 000 个时钟周期，或者几微秒。

Unix 系统的 `vmstat` 命令和 Windows 系统的 `perfmon` 工具都能报告上下文切换次数和内核占用的时间等信息。高内核占用率（超过 10%）通常象征繁重的调度活动，这很可能是由 I/O 阻塞，或竞争锁引起的。

11.3.2 内存同步

性能的开销有几个来源。`synchronized` 和 `volatile` 提供的可见性保证要求使用一个特殊的、名为存储关卡（memory barrier）的指令，来刷新缓存，使缓存无效，刷新硬件的写缓冲，并延迟执行的传递。存储关卡可能同样会对性能产生影响，因为它们抑制了其他编译器的优化；在存储关卡中，大多数操作是不能被重排序的。

在我们评估同步给性能带来影响的同时，区分竞争同步和无竞争同步也是非常重要的。`synchronized` 机制对无竞争同步进行了优化（`volatile` 总是非竞争的），在写作本书的时候，一个普通“fast-path”的非竞争同步，其性能开销在 20 至 250 个时钟周期。虽然这个开销不为零，但是它产生的影响已经微乎其微了。另一种选择是对安全性进行妥协，把自己陷入痛苦地搜寻 bug 的行动来保证你（或者你的后续事务）的安全。

现代的 JVM 能够通过优化，解除经确证不存在竞争的锁，从而减少额外的同步。如果一个锁对象只能由当前线程访问，那么允许 JVM 对其优化，去除对这个锁的请求，因为其他线程根本无法在同一个锁发生同步。例如，JVM 总是能够消除类似清单 11.2 中对锁的请求。

更加成熟的 JVM 可以使用**逃出分析**（escape analysis）来识别本地对象的引用并没有

在堆中被暴露，并且因此成为线程本地的。在清单 11.3 的 `getStoogeNames` 中，对 `List` 仅有的引用是本地变量 `stooges`，栈限制（`stack-confined`）的变量自动默认为线程本地的。在本地执行 `getStoogeNames`，至少需要获取/释放 `Vector` 的锁 4 次，每个 `add` 一次，`toString` 一次。然而，一个聪明的运行时编译器，能够合并这些调用，然后发现 `stooges` 和它的内部状态一直都没有逸出，因此这 4 次对锁的请求就可以被消除了⁴。

清单 11.3 锁省略的候选程序

```
public String getStoogeNames() {  
    List<String> stooges = new Vector<String>();  
    stooges.add("Moe");  
    stooges.add("Larry");  
    stooges.add("Curly");  
    return stooges.toString();  
}
```

即使没有逸出分析，编译器同样可以进行**锁的粗化**（`lock coarsening`），把邻近的 `synchronized` 块用相同的锁合并起来。在 `getStoogeNames` 中，JVM 如果进行锁的粗化，可能会把 3 个 `add` 调用结合起来，并对 `toString` 使用单独的锁请求和释放，在 `synchronized` 块的内部，利用启发式方法产生同步开销，而不是指令式方法⁵。这不仅仅减少了同步的开销，同时也给予优化者更大的代码块，很可能成就了进一步的优化。

不要过分担心非竞争的同步带来的开销。基础的机制已经足够快了，在这个基础上，JVM 能够进行额外的优化，大大减少或消除了开销。关注那些真正发生了锁竞争的区域中性能的优化。

一个线程中的同步也可能影响到其他线程的性能。同步造成了共享内存总线上的通信量；这个总线的带宽是有限的，所有的进程都共享这条总线。如果线程必须竞争同步带宽，所有使用到同步的线程都会受阻⁶。

⁴ 这个编译器优化被称为**锁省略**（`lock elision`），由 IBM 的 JVM 实现，期望可以用于 Java 7 的 HotSpot。

⁵ 聪明的动态编译器可以发现这个方法总是返回相同的字符串，在首次执行过后，它会重新编译 `getStoogeNames`，简单返回第一次执行的结果。

⁶ 这个方面有时候用来挑战不存在 `backoff` 的非阻塞算法，因为在激烈的竞争下，非阻塞算法比基于锁的算法会生成更多的同步通信量。参见第 15 章。

11.3.3 阻塞

非竞争的同步可以由 JVM 完全掌控 (Bacon 等, 1998); 而竞争的同步可能需要 OS 的活动, 这会增大开销。当锁为竞争性的时候, 失败的线程 (一个或多个) 必然发生阻塞。JVM 既能**自旋等待** (spin-waiting, 不断尝试获取锁, 直到成功), 或者在操作系统中**挂起** (suspending) 这个被阻塞的线程。哪一个效率更高, 取决于上下文切换的开销, 以及成功地获取锁需要等待的时间这两者之间的关系。自旋等待更适合短期的等待, 而挂起适合长时间等待。有一些 JVM 基于过去等待时间的数据剖析来在这两者之间进行选择, 但是大多数等待锁的线程都是被挂起的。

需要挂起线程可能因为线程无法得到锁, 或者因为它正在等待某个条件, 亦或被 I/O 操作阻塞。挂起需要两次额外的上下文切换, 以及 OS 和缓存的相关活动: 阻塞的线程在它时间限额还没有到期前就被换出, 稍后如果能够获得锁或者其等待的资源, 又会再被换入。(阻塞归因于锁的竞争, 线程持有这样的锁: 当它释放该锁的时候, 它必须通知 OS, 重新开始因该锁而阻塞的线程。)

11.4 减少锁的竞争

我们已经看到串行化会损害可伸缩性, 上下文切换会损害性能。竞争性的锁会同时导致这两种损失, 所以减少锁的竞争能够改进性能和可伸缩性。

访问独占锁守护的资源是串行的——一次只能有一个线程访问它。当然, 我们有很好的理由使用锁, 比如避免数据过期, 但是这样的安全性是用很大的代价换来的。对锁长期的竞争会限制可伸缩性。

并发程序中, 对可伸缩性首要的威胁是独占的资源锁。

有两个原因影响着锁的竞争性: 锁被请求的频率, 以及每次持有该锁的时间⁷。如果这两者的乘积足够小, 那么大多数请求锁的尝试都是非竞争的, 这样竞争性的锁将不会成为可伸缩性巨大的阻碍。但是, 如果这个锁的请求量很大, 线程将会阻塞以等待锁; 在极端的情况下, 处理器将会闲置, 即使仍有大量工作等着完成。

⁷ 这是**Little定律**的必然结论, 这是排队原理的一个成果, “稳定的系统中, 顾客的平均数量等于他们的平均到达率乘以他们在系统中平均的停留时间” (Little, 1961)。

有 3 种方式来减少锁的竞争:

- 减少持有锁的时间;
- 减少请求锁的频率;
- 或者用协调机制取代独占锁, 从而允许更强的并发性。

11.4.1 缩小锁的范围 (“快进快出”)

减小竞争发生可能性的有效方式是尽可能缩短把持锁的时间。这可以通过把与锁无关的代码移出 `synchronized` 块来实现, 尤其是那些花费 “昂贵” 的操作, 以及那些潜在的阻塞操作, 比如 I/O 操作。

我们很容易观察到长时间持有 “热门” 锁究竟是如何限制可伸缩性的; 我们在第 2 章看到了 `SynchronizedFactorizer` 的例子。如果一个操作持有锁超过 2 毫秒并且每一个操作都需要那个锁, 吞吐量不会超过每秒 500 个操作, 无论你拥有多少个空闲处理器。减少持有这个锁的时间到 1 毫秒, 能够把这个与锁相关的吞吐量提高到每秒 1 000 个操作⁸。

清单 11.4 展示了一个例子, 持有锁的时间超过了必要的时间。`userLocationMatches` 方法在 `Map` 中查找用户的位置, 使用正则表达式进行匹配, 看得到的结果值是否符合提供的模式。整个 `userLocationMatches` 方法都是 `synchronized` 的, 但是只有 `Map.get` 这一部分是真正需要调用锁的。

清单 11.4 持有锁超过必要的时间

```
@ThreadSafe
public class AttributeStore {
    @GuardedBy("this") private final Map<String, String>
        attributes = new HashMap<String, String>();

    public synchronized boolean userLocationMatches(String name,
                                                    String regexp) {
        String key = "users." + name + ".location";
        String location = attributes.get(key);
        if (location == null)
            return false;
        else
            return Pattern.matches(regexp, location);
    }
}
```



⁸ 事实上, 这里保守地计算了过长时间持有锁的开销, 因为它并没有计算锁的竞争带来的切换上下文的开销。

清单 11.5 中的 `BetterAttributeStore` 重写了 `AttributeStore`，从而大大减少了锁的持续时间。第一个步骤是构建 `Map` 的 `key` 与用户位置相关联，形式为：字符串 `users.name.location`。这需要实例化一个 `StringBuilder` 对象，并向其添加几个字符串，并实例化 `String` 类型的返回值。在获得了位置之后，就用正则表达式与位置的结果字符串进行匹配。因为构建 `key` 字符串和处理正则表达式都不需要访问共享状态，因而不需要将它们置于锁守护的范围之内。`BetterAttributeStore` 把这些步骤分解到 `synchronized` 块之外了，因此减少了占有锁的时间。

清单 11.5 减少锁持续的时间

```
@ThreadSafe
public class BetterAttributeStore {
    @GuardedBy("this") private final Map<String, String>
        attributes = new HashMap<String, String>();

    public boolean userLocationMatches(String name, String regexp) {
        String key = "users." + name + ".location";
        String location;
        synchronized (this) {
            location = attributes.get(key);
        }
        if (location == null)
            return false;
        else
            return Pattern.matches(regexp, location);
    }
}
```

缩小 `userLocationMatches` 方法中锁守护的范围，这大大减少了调用中遇到锁住情况的次数。由 Amdahl 定律得知，这消除了可伸缩性的一个阻碍，因为串行化的代码少了。

因为 `AttributeStore` 只有一个状态变量，`attributes`，我们可以使用代理线程安全（delegating thread safety）的技术（第 4.3 节）。通过使用线程安全的 `Map`（`Hashtable`、`synchronizedMap`，或者 `ConcurrentHashMap`）来取代 `attributes`，`AttributeStore` 可以通过底层线程安全的容器来代理所有线程安全的职责。这样能够省去 `AttributeStore` 中显式的同步，减少 `Map` 访问中锁的范围，并减小了未来的维护者因为忘记在访问 `attributes` 前获得相应锁而造成的风险。

尽管缩小 `synchronized` 块能够提高可伸缩性，`synchronized` 块可以变得极小——需要原子化的操作（比如在限定约束的情况下更新多个变量）必须包含在一个 `synchronized`

块中。并且因为同步的开销非零，把一个 `synchronized` 块分拆成多个 `synchronized` 块（保证正确的情况下），在某些时刻反而会对性能产生反作用⁹。理想的平衡当然是与平台相关的，但是在实践中，仅当你能够“切实”地把计算和阻塞操作从 `synchronized` 块中移开，这时担心 `synchronized` 块的大小才是有意义的。

11.4.2 减小锁的粒度

减小持有锁的时间比例的另一种方式是让线程减少调用它的频率（因此减小发生竞争的可能性）。这可以通过**分拆锁**（lock splitting）和**分离锁**（lock striping）来实现，也就是采用相互独立的锁，守卫多个独立的状态变量，在改变之前，它们都是由一个锁守护的。这些技术减小了锁发生时的粒度，潜在实现了更好的可伸缩性——但是使用更多的锁同样会增加死锁的风险。

做一个思想实验，想象如果整个应用程序**只有一个锁**，而不是为每一个对象分配一个独立的锁。那么，执行所有的 `synchronized` 块，不考虑它们的锁的情况，就会成为串行化执行。因为很多线程都在竞争相同的全局锁，因此两个线程同时请求同一个锁的情况增加了，导致了更多的竞争。所以如果对于锁的请求替代为请求锁的大集合，就会减少很多竞争。更少的线程会因为等待锁而发生阻塞，因此增加了可伸缩性。

如果一个锁守卫数量大于一、且**相互独立**的状态变量，你可能能够通过分拆锁，使每一个锁守护不同的变量，从而改进可伸缩性。结果是每个锁被请求的频率都减小了。

清单 11.6 中的 `ServerStatus` 展现了一个数据库服务器监视器接口的一部分，它维护了当前已登录的用户和当前正在执行的请求。当一个用户登录、注销、执行请求开始或结束的时候，`ServerStatus` 对象通过调用适当的 `add` 和 `remove` 方法进行更新。两种类型的信息完全是独立的，`ServerStatus` 甚至能够被分拆成两个类，不会影响到功能。

不使用 `ServerStatus` 的锁守护 `users` 和 `queries`，我们能够通过两个分离的锁来进行守卫的工作，正如清单 11.7 中所示。在分拆了锁之后，每一个新的更精巧的锁，相比于那些原始的粗糙锁，将会看到更少的通信量。（对于 `users` 和 `queries`，使用线程安全的 `Set` 对其进行代理，而不是使用显式的同步，这样做隐式地分拆了锁，因为每一个 `Set` 会使用不同的锁来守护其状态。）

当我们将希望对竞争不是很激烈的锁进行改进时，把一个锁分拆为两个，针对这种改进提供了最大的可能性。分拆锁对于竞争并不激烈的锁，能够在性能和吞吐量方面产生一些纯粹的改进，尽管这可能会在性能开始因为竞争而退化时增加负载的极限。分拆锁对于

⁹ 如果JVM进行了锁的粗化，它可能无论如何都会撤销对`synchronized`块的分拆。

清单 11.6 应当分拆锁的候选程序

```
@ThreadSafe
public class ServerStatus {
    @GuardedBy("this") public final Set<String> users;
    @GuardedBy("this") public final Set<String> queries;
    ...
    public synchronized void addUser(String u) { users.add(u); }
    public synchronized void addQuery(String q) { queries.add(q); }
    public synchronized void removeUser(String u) {
        users.remove(u);
    }
    public synchronized void removeQuery(String q) {
        queries.remove(q);
    }
}
```

清单 11.7 使用分拆的锁重构 ServerStatus

```
@ThreadSafe
public class ServerStatus {
    @GuardedBy("users") public final Set<String> users;
    @GuardedBy("queries") public final Set<String> queries;
    ...
    public void addUser(String u) {
        synchronized (users) {
            users.add(u);
        }
    }

    public void addQuery(String q) {
        synchronized (queries) {
            queries.add(q);
        }
    }
    // 使用分拆时对 remove 进行类似的重构
}
```

中等竞争强度的锁，能够切实地把它们大部分转化成非竞争的锁，这个结果是性能和可伸缩性都期望得到的。

11.4.3 分离锁

把一个竞争激烈的锁分拆成两个，很可能形成两个竞争激烈的锁。尽管这可以通过两个线程并发执行，取代一个线程，从而对可伸缩性有一些小的改进，但这仍然不能大幅地提高多个处理器在同一系统中并发性的前景。作为分拆锁的例子，`ServerStatus` 类并没有提供明显的机会来进行进一步分拆。

分拆锁有时候可以被扩展，分成可大可小加锁块的集合，并且它们归属于相互独立的对象，这样的情况就是分离锁。例如，`ConcurrentHashMap` 的实现使用了一个包含 16 个锁的 `Array`，每一个锁都守护 `Hash Bucket` 的 $1/16$ ；`Bucket N` 由第 $N \bmod 16$ 个锁来守护。假设哈希提供合理的拓展特性，并且关键字能够以统一的方式访问，这将会把对于锁的请求减少到约为原来的 $1/16$ 。这项技术使得 `ConcurrentHashMap` 能够支持 16 个并发的 `Writer`。（为了对多处理器系统的大负荷访问提供更好的并发性，这里锁的数量还可以增加，但是只有当你有足够的证据证明并发 `Writer` 的竞争强度足够大时，你可以增大锁的数量超过默认的 16，合理突破这个限制。）

分离锁的一个负面作用是：对容器加锁，进行独占访问更加困难，并且更加昂贵了。通常，一个操作可以通过获取最多不超过一个锁来进行，但是有个别的情况需要对整个容器加锁，比如当 `ConcurrentHashMap` 的值需要被扩展、重排，放入一个更大的 `Bucket` 时。这是获取所有分离的锁最典型的例子¹⁰。

清单 11.8 中，`StripedMap` 阐释了基于哈希的 `Map` 实现，其中用到了锁分离。它拥有 `N_LOCKS` 个锁，每一个守护 `Bucket` 的一个子集，大部分方法，比如 `get`，只需要得到一个 `Bucket` 锁。有些方法需要获得所有的锁，但是如 `clear` 方法实现的那样，并不需要同时获得所有这些锁¹¹。

11.4.4 避免热点域

分拆锁和分离锁能够改进可伸缩性，因为它们能够使不同的线程操作不同的数据（或者相同数据结构的不同部分），而不会发生相互干扰。能够从分拆锁受益的程序，通常是

¹⁰ 获得内部锁的一个任意集合的唯一方式是递归。

¹¹ 用这种方式清除 `Map` 不是原子操作，所以不排除当 `StripedMap` 为空时其他的线程正并发地向其加入元素；使操作原子化需要一次获得所有的锁。然而，在并发的容器中，客户端通常不能因为独占的访问加锁，否则 `size` 或 `isEmpty` 这样的操作在返回时可能会过期，尽管这样的小意外通常是可以接受的。

清单 11.8 基于哈希的 map 中使用分离锁

```
@ThreadSafe
public class StripedMap {
    // 同步策略: buckets[n] 由 locks[n%N_LOCKS] 保护
    private static final int N_LOCKS = 16;
    private final Node[] buckets;
    private final Object[] locks;

    private static class Node { ... }

    public StripedMap(int numBuckets) {
        buckets = new Node[numBuckets];
        locks = new Object[N_LOCKS];
        for (int i = 0; i < N_LOCKS; i++)
            locks[i] = new Object();
    }

    private final int hash(Object key) {
        return Math.abs(key.hashCode() % buckets.length);
    }

    public Object get(Object key) {
        int hash = hash(key);
        synchronized (locks[hash % N_LOCKS]) {
            for (Node m = buckets[hash]; m != null; m = m.next)
                if (m.key.equals(key))
                    return m.value;
        }
        return null;
    }

    public void clear() {
        for (int i = 0; i < buckets.length; i++) {
            synchronized (locks[i % N_LOCKS]) {
                buckets[i] = null;
            }
        }
    }
    ...
}
```

那些对锁的竞争普遍大于对锁守护数据竞争的程序。如果一个锁守护两个独立变量 X 和 Y , 线程 A 想要访问 X , 而线程 B 想要访问 Y (这就好像 `ServerStatus` 中一个线程调用了 `addUser`, 而另一个调用了 `addQuery`), 这两个线程没有竞争任何数据, 然而它们竞争相同的锁。

当每一个操作都请求变量的时候，锁的粒度很难被降低。这是性能和可伸缩性相互牵制的另一个方面；通常使用的优化方法，比如缓存常用的计算结果，会引入“热点域（hot fields）”，从而限制可伸缩性。

如果由你来实现 `HashMap`，你会遇到一个选择：`size` 方法如何计算 `Map` 条目的大小？最简单的方法是每次调用的时候数一遍。通常使用的优化方法是在插入和移除的时候更新一个单独的计数器；这会给 `put` 和 `remove` 方法造成很小的开销，以保证计数器的更新，但是，这会减少 `size` 方法的开销，从 $O(n)$ 减至 $O(1)$ 。

在单线程或完全同步的实现中，保存一个独立的计数能够很好地提高类似 `size` 和 `isEmpty` 这样的方法的速度，但是却使改进可伸缩性变得更难了，因为每一个修改 `map` 的操作都要更新这个共享的计数器。即使你对每一个哈希链（hash chain）都使用了锁的分离，对计数器独占锁的同步访问还是重新引入了可伸缩性问题。这看起来像是一个性能的优化——缓存 `size` 操作的结果——却已经转化为一个可伸缩性问题。这种情况下，计数器被称为**热点域**（hot field），因为每个变化操作都要访问它。

为避免这个问题，`ConcurrentHashMap` 通过枚举每个条目获得 `size`，并把这个值加入到每个条目，而不是维护一个全局计数。为了避免列举所有元素，`ConcurrentHashMap` 为每一个条目维护一个独立的计数域。同样由分离的锁守护¹²。

11.4.5 独占锁的替代方法

用于减轻竞争锁带来的影响的第三种技术是提前使用独占锁，这有助于使用更友好的并发方式进行共享状态的管理。这包括使用并发容器、读-写锁、不可变对象，以及原子变量。

`ReadWriteLock`（参见第 13 章）实行了一个多读者-单写者（multiple-reader, single-write）加锁规则：只要没有更改，那么多个读者可以并发访问共享资源，但是写者必须独占获得锁。对于多数操作都为读操作的数据结构，`ReadWriteLock` 与独占的锁相比，可以提供更好的并发性；对于只读的数据结构，不变性可以完全消除加锁的必要。

原子变量（参见第 15 章）提供了能够减少更新“热点域”的方式，如静态计数器、

¹² 如果对 `size` 的调用与修改 `map` 的操作相比更频繁，分离的数据结构能够为此进行优化，在调用 `size` 时用缓存容器大小的 `volatile` 值，在修改容器的时候锁定缓存（把值设为 -1）。如果进入时缓存的值为非负数，说明它是准确的，可以直接返回该值；否则，需要重新计算。

序列发生器、或者对链表数据结构头节点的引用。（第 2 章中的例子中，我们使用 `AtomicLong` 来维护 `Servlet` 的计数器。）原子变量类提供了针对整数或对象引用的非常精妙的原子操作（因此更具可伸缩性），并且使用现代处理器提供的低层并发原语，比如比较并交换（`compare-and-swap`）实现。如果你的类只有少量热点域，并且该类不与其他变量的不变约束，那么使用原子变量替代它可能会提高可伸缩性。（改变你的代码，减少它的热点域，这样会提高可伸缩性，甚至——原子变量减少更新热点域的开销，但是它们完全消除这种开销。）

11.4.6 监测 CPU 利用率

当我们测试可伸缩性的时候，我们的目标通常是保持处理器的充分利用。Unix 系统的 `vmstat` 和 `mpstat`，或者 Windows 系统的 `perfmon` 都能够告诉你处理器有多“忙碌”。

如果所有的 CPU 都没有被均匀地利用（有时 CPU 很忙碌地运行，有时候很“清闲”），那么你的首要目标应该是增强你程序的并行性。不均匀的利用率表明，大多数计算都由很小的线程集完成，你的应用程序将不能够利用额外的处理器资源。

如果你的 CPU 没有完全利用，你需要找出原因。有下面几种原因：

不充足的负载。可能被测试的程序还没有被加入足够多的负载。你可以增加负载，并检查利用率、响应时间和服务运行时间的变化。产生足够多的负载，使应用程序饱和需要计算机强大的能力；问题可能在于客户端系统是否具有足够的能力，而不是被测试系统。

I/O 限制。你可以通过 `iostat` 或者 `perfmon` 判定一个应用程序是否是受限于磁盘的，或者通过监测它的网络通信量级判断它是否有带宽限制

外部限制。如果你的应用程序取决于外部服务，比如数据库，或者 Web Service，那么瓶颈可能不在于你自己的代码。你可以通过使用 `Profiler` 工具，或者数据库管理工具来判断等待外部服务结果的用时。

锁竞争。使用 `Profiling` 工具能够告诉你，程序中存在多少个锁的竞争，哪些锁很“抢手”。如果不使用剖析器（`profiler`），你也可以通过随机取样来获得相同的信息，触发一些线程转储，并寻找其中线程竞争锁的信息。如果线程因等待锁被阻塞，与线程转储相应的栈框架会声明“`waiting to lock monitor...`”。非竞争的锁几乎不会出现在线程转储中；竞争激烈的锁几乎总会至少有一个线程在等待获得它，所以会频繁出现在线程转储中。

如果你的应用程序能够保持 CPU 处于忙碌状态，你可以使用监视工具来判断是否能够通过增加额外的 CPU 受益。一个程序如果只有 4 个线程，那么可能能够保持充分利用一个 4 路系统，但是移植到 8 路未必能看到性能的提升，因为很有可能需要等待可运行的线程来利用剩余的处理器资源。（你可能能够通过重新配置程序，把工作量分配给更多的线程，比如调整线程池大小。）`vmstat` 报告的一项是处于可运行态的线程数，但是这并不是当前运行中的线程数，因为有 CPU 不可用。如果 CPU 的利用率很高，并且没有并发的可运行线程等待 CPU，你的程序也许能够很好地利用更多的处理器资源。

11.4.7 向“对象池”说“不”

在早期的 JVM 版本中，对象的分配（allocation）和垃圾回收是非常慢的¹³，但是它们的性能在那之后有了本质的提高。事实上，Java 中的分配现在已经比 C 语言中的 `malloc` 更快了：在 HotSpot 1.4.x 和 5.0 中，`new Object` 的代码路径几乎只有十个机器指令。

针对对象的“慢”生命周期，很多程序员都会选择使用对象池化技术（object pooling），这项技术中，对象会被循环使用，而不是由垃圾收集器回收并在需要时重新分配。在单线程化的程序中，即使考虑到减少的垃圾收集开销，对象池化技术对于所有不那么“昂贵”的对象（最严重的损失是轻量和中量级对象）仍然存在性能缺失¹⁴（Click, 2005）。

在并发的应用程序中，池化表现得更糟糕。当线程分配新的对象时，需要线程内部非常细微的协调，因为分配运算通常使用线程本地的分配块来消除对象堆中的大部分同步。但是，如果这些线程从池中请求对象，那么协调访问池的数据结构的同步就成为必然了，这便产生了线程阻塞的可能性。又因为由锁的竞争产生的阻塞，其代价比直接分配的代价多几百倍，即使是一个很小的池竞争都会造成可伸缩性的瓶颈。（甚至是非竞争的同步，其代价也会比分配一个对象大很多。）这虽然被认为是性能优化的另一技术，但是会产生可伸缩性危险。池化有其用途¹⁵，但是对于性能优化来说，使用是有局限性的。

¹³ 与其他所有东西一样——同步、图形化、JVM 启动、反射——作为实验性技术的第一个版本。

¹⁴ 除此以外，作为 CPU 周期的损失，对象池化技术还有一些其他的问题，其中正确地设定池的大小就是一个挑战（太小，池会失去效率；太大会对垃圾回收器形成压力，占用本应该用于其他事务的内存资源）；一个对象没能被正确重置的风险会引入“隐蔽的”bug；也可能出现一个线程把对象归还给池，却仍在继续使用该对象。这会产生一种“old-to-young”引用的模式，给垃圾回收器不断制造新的麻烦。

¹⁵ 在特定的环境中，比如 J2ME 或 RTSJ 的目标，对象池化技术需要用于有效的内存管理，或者管理响应性。

分配对象通常比同步要“便宜”。

11.5 示例：比较 Map 的性能

单线程化的 `ConcurrentHashMap` 的性能要比同步的 `HashMap` 的性能稍好一些，而且在并发应用中，这种作用就十分明显了。`ConcurrentHashMap` 的实现，假定大多数常用的操作都是获取已存在的某个值，因此它的优化是针对 `get` 操作，提供最好的性能和并发性。

同步的 `Map` 实现中，可伸缩性最主要的阻碍在于整个 `Map` 存在一个锁，所以一次只有一个线程能够访问 `map`，从另一方面来看，`ConcurrentHashMap` 并没有对成功的读操作加锁，对写操作和真正需要锁的读操作使用了分离锁的方法。因此，多线程能够并发访问 `Map`，而不被阻塞。

图 11.3 阐释了几种 `Map` 实现不同的可扩展性：`ConcurrentHashMap`，`ConcurrentSkipListMap`，以及由 `synchronizedMap` 包装的 `HashMap` 和 `TreeMap`。前两个的设计都是线程安全的；后两个通过同步的包装器实现了线程安全。每次运行中， N 个线程并发执行一个紧密的循环，随机选择 `key`，并尝试获取相应的值。如果不存在符合结果，它会增加 `Map` 的可能性 $p = .6$ ，如果存在，它会减少可能性 $p = .02$ 。这个测试在预发布的 Java 6，8 路 Sparc V880 系统下进行，并且这个图表展示了规范为单线程情况的 `ConcurrentHashMap` 的吞吐量。（并发容器和同步容器间的可伸缩性差别比 Java 5.0 还要明显。）

`ConcurrentHashMap` 和 `ConcurrentSkipListMap` 的数据显示，它们能很好地应对数量很大的线程；吞吐量随着线程数量的增加而增长。尽管图 11.3 中线程的数量看起来不是很大，但是与普通的程序相比，这个测试程序为每个线程都产生了更多的竞争，因为它除了向 `Map` 施加压力，几乎没有做任何事情；而真正的程序会在每个迭代中进行额外的线程本地工作。

同步容器的数量并不是越多越好。单线程的情况与 `ConcurrentHashMap` 一致，但是一旦负载由多数为非竞争的情况变成多数为竞争性的情况——这里是两个线程——同步的容器就会很糟糕。这在锁竞争的代码行为中是很常见的。只要竞争小，每个操作所花费的时间取决于真正工作的时间，吞吐量会因为线程数的增加而增加。一旦竞争变得激烈，每个操作花费的时间就由上下文切换和调度延迟决定了，并且加入更多的线程不会对吞吐量有什么帮助。

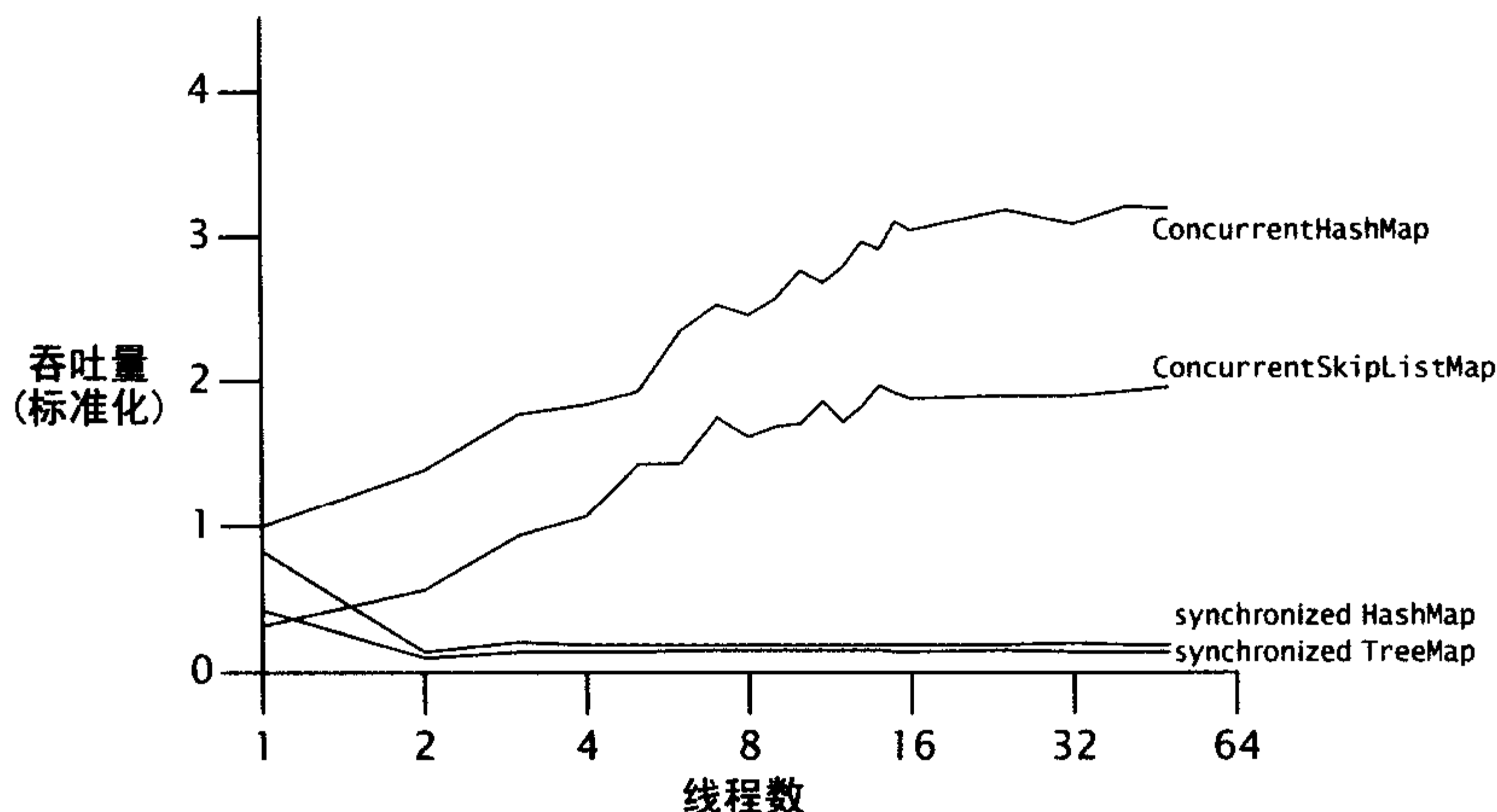


图 11.3 比较不同 Map 实现间的可伸缩性

11.6 减少上下文切换的开销

很多任务引入的操作都会发生阻塞；在运行和阻塞这两个状态之间转换需要使用上下文切换。服务器应用程序发生阻塞的一个缘由是在处理请求期间产生日志消息；为了了解如何通过减少上下文切换的开销来提高吞吐量，我们将对两种日志方案的调度行为进行分析。

大多数日志框架都是围绕 `println` 进行“瘦”包装的；当你有些事情需要记录日志的时候，仅仅需要把它打印出来就好了。另一个方案是第 152 页，`LogWriter` 所显示的：日志记录的工作由一个专职的后台线程完成，而不是由请求线程完成。从开发人员的视角来看，两种方案基本上相同。但是性能上可能会有差别，取决于日志活动的工作量，即有多少线程正在记录日志，还取决于另一些因素，比如上下文切换的开销¹⁶。

日志操作的服务时间包括与输入/输出流相关的类花费的时间；如果 I/O 操作发生阻塞，它很可能包括线程被阻塞的时间。操作系统会把这个阻塞线程从调度队列中移出，直到 I/O 结束，并且很可能比这个花费的时间更长。当 I/O 结束的时候，其他线程可能被唤醒，并被允许完成它们的调度时间限额，并且在调度队列中有些线程可能已经排在我们前

¹⁶ 构建一个 logger，它会把 I/O 移到另一个线程，这样可以提高性能，但也会引入更多的设计复杂度，比如中断（如果一个阻塞在日志操作的线程被中断，还出现什么情况？）、服务担保（logger 能保证成功加入队列的消息都能在服务终止前被记录么？）、饥饿策略（当生产者记录消息比 logger 线程的处理能力更快的时候会如何？），还有服务的生命周期（如何关闭 logger，如何就服务的状态与生产者进行沟通？）。

面了——这会进一步增加服务时间。其他的情况可能是，如果多个线程同时记录日志，它们可能还会竞争输出流的锁，这样的情况下结果与阻塞的 I/O 是一样的——线程阻塞并等待锁，并被换出。这样的记录方式牵涉到了内联和加锁，会导致上下文切换的增多，以及服务时间的增加。

延长请求的服务时间是令人不快的，有以下几条原因。首先服务时间反映服务的质量：越长的服务时间意味着有人等待了更长的时间获得结果。但更重要的是，更长的服务时间在这里意味着更多锁的竞争。在 11.4.1 小节中，我们提到的“快进快出”原理告诉我们，应该尽可能短地占有锁，因为锁被占用的时间越长，它被争夺的可能性就越大。如果线程因为等待 I/O 发生阻塞，并且此时持有一个锁，另一个线程很可能想要得到这个锁，然而却已经被前一个线程得到了。并发系统在多数锁为非竞争锁的时候会有更好的性能，因为请求竞争性的锁意味着更多的上下文切换。代码如果造成更多的上下文切换意味着产生更少的吞吐量。

把 I/O 操作从请求处理线程中分离出来很可能缩短处理请求的平均服务时间。调用 `log` 的线程不再因等待输出流的锁，或者等待 I/O 完成而发生阻塞；它们只需使消息加入队列，然后返回到它们自己的任务。另一方面，我们已经向消息队列引入了竞争的可能性，但是 `put` 操作相对于日志的 I/O（可能需要系统调用）是更加轻量的，并且在真实使用中更不易发生阻塞（只要队列没有填满）。因为请求线程现在不易发生阻塞了，因此在请求中间就会减少上下文被换出。我们完成的工作是把复杂、不确定的代码路径，包括 I/O 和锁竞争的可能性，转化为一致的代码路径。

进一步扩展，我们刚刚拆分了工作，把 I/O 操作移到了另一个线程，使用户看不到这个开销（这本身已经获得成功）。而且通过把**所有的**日志 I/O 移入一个线程，我们同样消除了输出流的竞争，因此又减少了竞争源。这改进了整体的吞吐量，因为需要调度的资源更少了，上下文切换更少了，锁的管理更简单了。

把 I/O 操作从请求处理线程中移到一个 `logger` 线程，就好像一个 bucket brigade（排成长队以传水救火的队列）和一群跑来跑去想要救火的人之间的差别。在“上百个人拿着桶跑来跑去”的方案中，你对水源和火都存在着很大的竞争（结果是更少的水能够传递到灭火点），还会造成更低的效率，因为每一个工人都在不停的变换模式（注水、跑步、倒水、跑步，等等）。在 bucket brigade 的解决方案中，从水源到燃烧的建筑物间的水流是不断的，付出更少的体力却换得了更多的水传输过来。每个工人只负责自己的一项工作。如同中断对于人而言会造成麻烦，降低效率，阻塞和上下文切换会给线程造成麻烦。

总结

因为使用线程最主要的目的是利用多处理器资源，在并发程序性能的讨论中，我们通常更多地关注吞吐量和可伸缩性，而没有强调自然服务时间。Amdahl 定律告诉我们，程序的可伸缩性是由必须连续执行的代码比例决定的。因为 Java 程序中串行化首要的来源是独占的资源锁，所以可伸缩性通常可以通过以下这些方式提升：减少用于获取锁的时间，减小锁的粒度，减少锁的占用时间，或者用非独占或非阻塞锁来取代独占锁。

测试并发程序

Testing Concurrent Programs

并发程序在设计上采用与顺序程序大致相同的原则和模式。不同的是并发程序存在一定程度的不确定性，顺序程序却没有这个问题。这样的不确定性增加了潜在的交互和失败模式的数量，我们必须为此做好计划和分析。

类似地，很多用来测试并发程序的方法，都借鉴并发展了测试顺序程序的方法。一些用于测试顺序程序正确性与性能的技术，同样可以用在测试并发程序上。不过对于并发程序而言，可能出错的地方远比顺序程序要多。为并发程序创建测试，所要面临的主要挑战在于：那些潜在错误的发生并不具有确定性，而是随机的；能够揭示这种失败的测试，与普通的顺序测试相比，一定会有更广泛的覆盖度和更长的运行时间。

并发类的测试基本分为两类，对**安全性**与**活跃度**的测试。大多数测试都包括其中的一项或全部。在第 1 章，我们曾经把安全性定义为“什么坏事都没有发生过”，把活跃度定义为“好的事情终究会发生”。

安全性测试通常会验证不变约束，以这种方式来检查类的行为是否遵循了它的规约。例如，假设存在如下的链表实现，该链表缓存了最后一次被修改时的长度，那么应该有这么一项安全性测试，它会比较缓存中记录的长度与链表中实际元素的数量。在单线程化的程序中这很简单，因为链表的内容不会在你测试其属性时发生改变。但是在并发程序中，仅仅这样的一个测试也可能充满了竞争，除非你能得到计数域，并在单独的原子操作中计算元素的数量。为了做到这一点，可以锁住链表以独占访问，然后使用链表提供的一些“原子快照”特性；或者也可以使用链表提供的“测试点（test points）”，这些都可以让测试去断言不变约束；或者索性原子性地执行测试代码。

在本书中，我们曾经使用时序图来展现“不幸的”交互，它导致了未被正确构造的类的失败；测试程序会尝试在足够大的状态空间里查找，从而让这些偶发的事件最终得以发生。不幸的是，测试代码同样可能引入时序或者同步带来的影响，这会屏蔽掉 bug，而其实这些 bug 本可以自己暴露出来的¹。

¹ 添加了调试和测试代码后就会消失的 bug 被戏称为 *Heisenbugs*。

活跃度特性自身，就是对测试的一大挑战。活跃度测试包括“应该执行”和“不该执行”两个方面，这些都是很难量化的——你怎样才能验证一个方式是被阻塞了，而不仅仅是运行得有些慢？类似地，你又如何测试一个算法**不会**死锁？要等上多久才应该宣告它的失败？

与活跃度测试相关的是性能测试。性能可以通过很多方式来测量，其中包括：

吞吐量：在一个并发任务集里，已完成任务所占的比例；

响应性：从请求到完成一些动作之间的延迟（也被称作**等待时间**）；

可伸缩性：增加更多的资源（通常是指 CPU），就能提高（或者缓解短缺）吞吐量。

12.1 测试正确性

为并发类开发单元测试的流程，始于和顺序类相同的分析——识别出不变约束和后验条件，这些都要接受例行检查。如果幸运的话，它们中的大部分都明确地写在了规约中；余下的时间里，编写测试不亚于一次反复探索规约的历险。

为了演示一个具体的例子，我们接下来要为一个有限缓存构建一系列的测试用例。清单 12.1 实现了 `BoundedBuffer`，它使用 `Semaphore` 来实现必要的限制和阻塞。

`BoundedBuffer` 实现了一个基于数组的定长队列，它的 `put` 和 `take` 方法是可阻塞的，并受控于一对计数信号量。信号量 `availableItems` 代表了可以从缓存中**删除**的元素个数，它的初始值为零（因为缓存的初始状态是空）。类似地，信号量 `availableSpaces` 代表了可以**插入**到缓存的元素个数，它的初始值就是缓存的大小。

`take` 操作首先请求一个许可（`permit`），这个许可从 `availableItems` 中获得。如果缓存不为空，请求会立即成功，否则请求会被阻塞，直到缓存不空为止。一旦获得了许可，`take` 会删除缓存中的下一个元素，同时还释放一个许可给 `availableSpaces` 信号量²。`put` 操作定义的顺序刚好相反。因此无论是从 `put` 或是 `take` 方法中退出，两个信号量计数器的总和永远等于缓存的限制长度。（在实际工作中，如果你需要一个有限缓存，应该使用 `ArrayBlockingQueue` 或 `LinkedBlockingQueue`，而不是自己产生一个。不过这里所示的关于如何控制添加、删除的技术，同样可以用于其他的数据结构。）

² 在计数信号量中，许可不会被显式地表现出来，也不会和它所在的线程有任何关联；`release` 创建许可，`acquire` 消费许可。

清单 12.1 利用 Semaphore 实现的有限缓存

```
@ThreadSafe
public class BoundedBuffer<E> {
    private final Semaphore availableItems, availableSpaces;
    @GuardedBy("this") private final E[] items;
    @GuardedBy("this") private int putPosition = 0, takePosition = 0;

    public BoundedBuffer(int capacity) {
        availableItems = new Semaphore(0);
        availableSpaces = new Semaphore(capacity);
        items = (E[]) new Object[capacity];
    }
    public boolean isEmpty() {
        return availableItems.availablePermits() == 0;
    }
    public boolean isFull() {
        return availableSpaces.availablePermits() == 0;
    }

    public void put(E x) throws InterruptedException {
        availableSpaces.acquire();
        doInsert(x);
        availableItems.release();
    }
    public E take() throws InterruptedException {
        availableItems.acquire();
        E item = doExtract();
        availableSpaces.release();
        return item;
    }

    private synchronized void doInsert(E x) {
        int i = putPosition;
        items[i] = x;
        putPosition = (++i == items.length)? 0 : i;
    }
    private synchronized E doExtract() {
        int i = takePosition;
        E x = items[i];
        items[i] = null;
        takePosition = (++i == items.length)? 0 : i;
        return x;
    }
}
```

12.1.1 基本的单元测试

对 `BoundedBuffer` 最基本的单元测试与我们在顺序化上下文中所做的事情类似——先创建一个有限缓存，再调用它的方法，最后断言它的后验条件和不变约束。有一些不变约束很快进入了我们的视野：一个新建立的缓存应该确定自己是空的，同时不是满的。另一个类似的，但略微有些难度的安全测试是，把 N 个元素插入到容量为 N 的缓存中（这个过程应该成功，不能被阻塞），然后测试缓存是否意识到自己已经满（不空）了。在 JUnit 中，针对上述特性的测试方法如清单 12.2 所示。

清单 12.2 `BoundedBuffer` 的基本单元测试

```
class BoundedBufferTest extends TestCase {
    void testIsEmptyWhenConstructed() {
        BoundedBuffer<Integer> bb = new BoundedBuffer<Integer>(10);
        assertTrue(bb.isEmpty());
        assertFalse(bb.isFull());
    }

    void testIsFullAfterPuts() throws InterruptedException {
        BoundedBuffer<Integer> bb = new BoundedBuffer<Integer>(10);
        for (int i = 0; i < 10; i++)
            bb.put(i);
        assertTrue(bb.isFull());
        assertFalse(bb.isEmpty());
    }
}
```

这些简单的测试方法完全是顺序化的。在你的测试套件（test suite）中包含一个顺序化的测试集，这种做法通常是有帮助的，因为在你开始将注意力转向数据竞争之前，这些测试集就可能在错误还没有涉及并发问题时就会发现它们。

12.1.2 测试阻塞操作

测试并发的本质属性时，需要引入更多的线程。大多数测试框架对并发性并不友好：它们只包含很少的工具，用来创建线程或者监视线程，确保它们不会意外地终结。如果测试用例创建的助手线程发现了一个错误，框架通常不能得知与这个助手线程相关的是哪一个测试。所以需要做一些工作，来把成功或失败的信息传递回主测试运行程序线程，让这些消息可以被呈现出来。

为了让 `java.util.concurrent` 有一致的测试，能否把失败明确地与一个特定测试关联起来，就变得很重要。因此 JSR 166 专家组（Expert Group）创建了一个基类³，它提供的

³ <http://gee.cs.oswego.edu/cgi-bin/viewcvs.cgi/jsr166/src/test/tck/JSR166TestCase.java>

方法可以在 `tearDown` 期间传递失败信息，然后将它们呈现出来。这个类遵循下述的约定：每个测试在它创建的所有线程终止前，必须等待。你不需要考虑这么多；最重要的还在于测试是否通过的信息，以及失败信息是否能够呈现出来，以供诊断问题时使用，这一点应该明确。

如果一个方法应该在某些特定条件下被阻塞，那么测试这种行为时，只有在线程**没有**执行完毕前，测试才是成功的。测试方法的阻塞，类似于测试方法抛出的异常；如果方法可以正常返回，则意味着失败。

对方法阻塞的测试，带来了额外的复杂性：当方法成功地被阻塞后，你还必须想办法解除方法的阻塞。做到这一点最显而易见的方法是通过中断——在独立的线程中开始一个阻塞活动，等到线程阻塞后，再中断它。这样就可以断言阻塞操作成功。当然，这需要你的阻塞活动可以提前返回或者抛出 `InterruptedException`，以此响应中断。

“等到线程阻塞后……”这句话说起来简单，做起来难。实际上，你必须确定执行一些指令可能花费多长时间。测试所等待的时间要比这个时间长。如果你估计的时间不准确（在这种情况下，你会看到测试的假失败），应该随时准备调高这个值。

清单 12.3 演示了测试阻塞操作的一种方法。这种方法会创建一个“taker”线程，尝试从空缓存中获得（take）一个元素。如果 take 操作成功，表明测试失败。Test Runner 线程启动 taker 线程，等上一段时间，然后中断 taker。如果 taker 线程正确地阻塞在 take 操作上，就会抛出 `InterruptedException`，紧接着这个异常的 catch 块把这个视为成功，让线程退出。主 Test Runner 线程会尝试拼接（join）到 taker 线程之后，通过调用 `Thread.isAlive` 验证 join 是否成功返回；如果 taker 线程可以响应中断，join 可以更快地完成。

take 操作可能由于一些无法预期的方式无法结束，定时的 join 能确保测试完成。这个测试方法测试了 take 的几种特性——不仅能阻塞，而且被中断后还能抛出 `InterruptedException`。只有在极少数的情况下，显式地子类化 `Thread` 会比在池中使用 `Runnable` 更合适：为使用 join 来测试正确的终止，就是这样的一种情况。同样的方法，还可以测试主线程将一个元素放入队列后，是否会解除 taker 线程的阻塞。

使用 `Thread.getState` 验证线程是否在一个条件等待下被真正阻塞，这种方法很具诱惑力，但其实并不可靠。线程甚至进入 `WAITING` 或者 `TIMED_WAITING` 状态，也不需要真正被阻塞。JVM 能选择自旋等待（spin-waiting）实现阻塞。类似地，`Object.wait` 和 `Condition.await` 允许发出假唤醒（spurious wakeup，参见第 14 章），让一个线程从 `WAITING` 或 `TIMED_WAITING` 临时地转换为 `RUNNABLE`，即使它等待的条件尚未成真。即使忽略这些实现在选择上的不同，目标线程进入阻塞状态也会花费一些时间。`Thread.getState` 的结果不能用在并发控制中，它会限制测试的有效性——它主要的功能还是作为调试信息的来源。

清单 12.3 测试阻塞与响应中断

```
void testTakeBlocksWhenEmpty() {
    final BoundedBuffer<Integer> bb = new BoundedBuffer<Integer>(10);
    Thread taker = new Thread() {
        public void run() {
            try {
                int unused = bb.take();
                fail(); // 如果运行到这里, 说明有错误
            } catch (InterruptedException success) { }
        }
    };
    try {
        taker.start();
        Thread.sleep(LOCKUP_DETECT_TIMEOUT);
        taker.interrupt();
        taker.join(LOCKUP_DETECT_TIMEOUT);
        assertFalse(taker.isAlive());
    } catch (Exception unexpected) {
        fail();
    }
}
```

12.1.3 测试安全性

清单 12.2 和清单 12.3 中的测试用例, 测试了有限缓存的重要属性, 但它们不可能发现数据竞争引发的错误。要想测试一个并发类在不可预知的并发访问下是否能够正确执行, 我们需要安排多个线程运行 put 和 take 操作, 运行一段时间过后, 看看测试会不会仍旧好端端的不出什么问题。

构建测试去发现并发类中的安全性错误, 这是一个“鸡生蛋, 蛋孵鸡”的问题: 测试程序自身就是并发程序。开发良好的并发测试程序, 可能比开发它们要去测试的类还困难。

为并发类创建有效的安全测试, 其挑战在于: 如何在程序出现问题并导致某些属性极度可能失败时, 简单地识别出这些受检查的属性来, 同时不要人为地让查找错误的代码限制住程序的并发性。最好能做到在检查测试的属性时, 不需要任何的同步。

有一种方法, 与 BoundedBuffer 这类用于生产者-消费者设计的类配合使用效果很好, 它是这样做的: 只核对所有放入队列和缓冲的元素最终是否都出来了, 其他的什么也不做。这种方法的一种简单的实现是, 当元素被置入队列时, 同时把它插入到一个“影子”清单中,

当它从队列中被删除时，同时从“影子”清单中删除，然后等测试完成后断言影子清单是否为空。但是，修改影子清单需要同步，而且可能阻塞，所以这种方法会干扰测试线程的调度。

还有一个更好的办法：使用一个对顺序敏感的检验求和（checksums）函数，计算出入队与出队的元素个数，然后进行比较。如果它们相等，那么测试通过。在只有一个生产者向缓存置入元素，同时只有一个消费者取出元素的情况下，这种方法不仅可以测试是否取出了(可能)正确的元素，而且还能测试元素被取出的顺序是否正确。因此当只有唯一的生产者和消费者时，这种方法的效果很好。

如果想扩展这种方法，让它应用于多生产者、多消费者的情况，就需要使用一个不同的检验求和函数，它对元素入队的顺序是**不敏感的**，这样可以在测试完成后，再将多个检查结果组合在一起。任何满足交换率的操作，比如加法、XOR（异或），都符合这些条件。否则的话，同步访问一个共享的 checksum 域，会成为并发的瓶颈，也会打乱测试的时序。

为了确保你的测试的确测试了你所想的事情，有一点很重要：不能让编译器预先得知 checksum 的值。使用恒定的整数作为测试数据并不是好办法。如果这样的话，结果永远是相同的，而且聪明的编译器还可能提前计算出这个结果来。

应该使用随机生成的测试数据来避免这个问题。但是另一方面，可选择的随机数生成器（Random Number Generator, RNG）寥寥无几，也因此影响到了大量本应该很有效的测试。大多数随机数生成器类是线程安全的，因此会引入额外的同步⁴，由此，随机数的生成会给类与时间组件之间带来耦合。如果每个线程都有它自己的 RNG，那么就能使用非线程安全的 RNG 了。

与其使用一个通用目的的 RNG，不如使用简化的伪随机函数。你并不需要高质量的随机性；只要能够保证每次运行都有不同的数字，这样的随机性对你就足够了。清单 12.4 中的 xorShift 函数(Marsaglia, 2003)是随机数函数中“性价比”最好的之一。它基于 hashCode 和 nanoTime 计算随机数，所得的结果既是不可预知的，而且几乎每次运行都不同。

清单 12.4 适用于测试的中等品质的随机数生成器

```
static int xorShift(int y) {
    y ^= (y << 6);
    y ^= (y >>> 21);
    y ^= (y << 7);
    return y;
}
```

⁴ 有很多不为开发者和用户所了解的基准测试，都是关于RNG的并发瓶颈有多大的简单测试。

清单 12.5 和清单 12.6 的 `PutTakeTest` 启动了 N 个生产者线程同时生成元素并把它们加入队列，还启动了 N 个消费者线程从队列中取出元素。当元素进出队列时，每个线程都会更新元素的 `checksum`。每个线程使用一个 `checksum`，就可以在测试结束后把它们汇总到一起。相比于需要被测试的缓存，这样做并没有引入过多的同步或竞争。

创建与启动线程可能是适度的重量级操作，这依赖于你的平台。如果你在循环中启动了大量短期运行的线程，那么在最坏的情况下，这些线程都是顺序而非并发地运行。即使在略微好一些的情况下，第一个线程仍然比其他线程具有“领先优势”。这一事实意味着你可能无法得到你期望中那么多的交替执行的行为：第一个线程会在没有竞争的情况下运行一段时间，然后前两个线程会并发地运行一段时间，只有到了最后，所有线程才会一起并发执行。（同样的事情还会在线程终止运行时发生：第一个运行的线程也会提前完成。）

我们在第 5.5.1 节中介绍过的一种技术，可以缓解这个问题，那就是使用两个不同的 `CountDownLatch`，分别作为开始阀门和结束阀门。使用 `CyclicBarrier` 也可以获得同样的效果，其初始化参数的数字是工作者线程的数量再加一，在测试运行的开始前和结束后，工作者线程和驱动测试的线程都在 `Barrier` 中等待。这能确保所有线程都站在同一起跑线后，再开始任何工作。`PutTakeTest` 正是利用了这一技术来协调工作者线程的启动和停止的，从而产生更多潜在并发的交替操作。我们仍然无法保证调度器不会完全顺序地运行每个线程，但是只要它们运行的时间足够长，就能降低调度器对结果产生的影响。

`PutTakeTest` 中的最后一个小技巧是使用一个明确的终结条件，这样就不需要额外的交互线程进行协调，来通知测试何时完成。`test` 方法会准确地启动相同数量的生产者和消费者，它们会分别置入（`put`）和取出（`take`）相同数量的元素，这样，添加与删除的条目的总数是一样的。

像 `PutTakeTest` 这样的测试，更能发现安全隐患。比如，在实现由信号量控制的缓存时，常犯的一个错误是忘记这一点：真正执行插入和取出的代码需要（使用 `synchronized` 或 `ReentrantLock` 实现）互斥。如果有这样一个版本的 `BoundedBuffer`，它忘记将 `doInsert` 和 `doExtract` 声明为 `synchronized`，那么 `PutTakeTest` 的执行就会相当快速地失败。如果在 `PutTakeTest` 中存在几十个线程，让它们针对不同容量的缓存以及不同的系统，重复运行几百万次，这会提高我们对于 `put` 和 `take` 方法在数据安全方面的信心。

测试应该在多处理器系统上运行，以提高潜在交替运行的多样性。但是，多个 CPU 未必会使测试更加高效。为了能够最大程度地检测到时序敏感的数据竞争的发生机会，应该让测试中的线程数多于 CPU 数，这样在任何给定的时间里，都有一些线程在运行，一些被交换出执行队列，这样可以增加线程间交替行为的随机性。

清单 12.5 BoundedBuffer 的生产者-消费者测试程序

```
public class PutTakeTest {
    private static final ExecutorService pool
        = Executors.newCachedThreadPool();
    private final AtomicInteger putSum = new AtomicInteger(0);
    private final AtomicInteger takeSum = new AtomicInteger(0);
    private final CyclicBarrier barrier;
    private final BoundedBuffer<Integer> bb;
    private final int nTrials, nPairs;

    public static void main(String[] args) {
        new PutTakeTest(10, 10, 100000).test(); // sample parameters
        pool.shutdown();
    }

    PutTakeTest(int capacity, int npairs, int ntrials) {
        this.bb = new BoundedBuffer<Integer>(capacity);
        this.nTrials = ntrials;
        this.nPairs = npairs;
        this.barrier = new CyclicBarrier(npairs * 2 + 1);
    }

    void test() {
        try {
            for (int i = 0; i < nPairs; i++) {
                pool.execute(new Producer());
                pool.execute(new Consumer());
            }
            barrier.await(); // 等待所有线程作好准备
            barrier.await(); // 等待所有线程最终完成
            assertEquals(putSum.get(), takeSum.get());
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
    }

    class Producer implements Runnable { /* 清单 12.6 */ }

    class Consumer implements Runnable { /* 清单 12.6 */ }
}
```

清单 12.6 PutTakeTest 中的生产者和消费者类

```
/* PutTakeTest 中的内部类 (清单 12.5) */
class Producer implements Runnable {
    public void run() {
        try {
            int seed = (this.hashCode() ^ (int)System.nanoTime());
            int sum = 0;
            barrier.await();
            for (int i = nTrials; i > 0; --i) {
                bb.put(seed);
                sum += seed;
                seed = xorShift(seed);
            }
            putSum.getAndAdd(sum);
            barrier.await();
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
    }
}

class Consumer implements Runnable {
    public void run() {
        try {
            barrier.await();
            int sum = 0;
            for (int i = nTrials; i > 0; --i) {
                sum += bb.take();
            }
            takeSum.getAndAdd(sum);
            barrier.await();
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
    }
}
```

如果测试会在完成了一定数量的操作之前一直运行，那么倘若被测试的代码由于 bug 而遇到一个异常，测试用例就可能永远不会结束。解决这个问题最常用的方式是，让测试框架去终止那些没有在规定时间内完成的测试。这个“规定的时间”取决于你的经验，同时要分析每次失败，以确保问题出现的起因不是你没有等待足够的时间。（这个问题并不是并发类测试所特有的；顺序化测试也必须要区分长时间的运行和无限循环。）

12.1.4 测试资源管理

到目前为止，所有的测试都关注于一个类对其规约的“忠诚度”——这是类应该做的。测试的另一个方面是测试类**没有**做它**不应该**做的，比如资源泄漏。任何持有或管理着其他对象的对象，都应该在不需要某个对象时，放弃该对象的引用。像存储泄漏就会抑制垃圾回收器回收内存（以及线程、文件句柄、套接字、数据库连接或者其他的有限资源），还会导致资源耗尽和应用程序的失败。

资源管理的问题对于像 `BoundedBuffer` 这样的类尤其重要——之所以要限制一个缓存，全部原因就是防止由资源耗尽导致的应用程序失败的发生，这在生产者的数量远远多于消费者时就会出现。对缓存的限制，会引起生产力过剩的生产者阻塞，这样它们就不会持续创建更多的工作，否则这些工作将要消耗越来越多的内存及其他资源。

对内存不合理的占有，可以简单地通过堆检查工具测试出来，这些工具可以测量应用程序内存的使用；很多不同的商业和开源的堆追踪（heap-profiling）工具可供选择。清单 12.7 中的 `testLeak` 方法中，预留了一个占位符，在这里堆检查工具会记录堆的快照，这会强制一次垃圾回收⁵，然后记录下堆大小和内存用量的信息。

`testLeak` 方法会向一个有限缓存中插入几个巨型对象，然后将它们移除；2 号堆快照的内存用量应该大致等于 1 号堆快照的内存用量。另一方面，`doExtract` 如果忘记将返回元素的引用置空（`items[i] = null`），那么两次快照所报告的内存用量将会明显不同。（这是为数不多需要显式置空的情况之一；大多数时候，这种做法不仅无益，事实上甚至是有害的 [EJ Item 5]。）

12.1.5 使用回调

回调用户提供的代码，有助于创建测试用例；回调常常发生在一个对象生命周期的已知点上，这些点提供了很好的机会，来断言不变约束。例如，`ThreadPoolExecutor` 就把调用转到了任务的 `Runnable` 和 `ThreadFactory` 上。

⁵ 在技术上是不可强制垃圾回收执行的；`System.gc` 仅仅建议 JVM 在一个合适的时间执行垃圾回收。通过 `-XX:+DisableExplicitGC`，可以告诉 HotSpot 忽略 `System.gc` 调用。

清单 12.7 测试资源泄漏

```
class Big { double[] data = new double[100000]; }

void testLeak() throws InterruptedException {
    BoundedBuffer<Big> bb = new BoundedBuffer<Big>(CAPACITY);
    int heapSize1 = /* heap 的快照 */ ;
    for (int i = 0; i < CAPACITY; i++)
        bb.put(new Big());
    for (int i = 0; i < CAPACITY; i++)
        bb.take();
    int heapSize2 = /* heap 的快照 */ ;
    assertTrue(Math.abs(heapSize1-heapSize2) < THRESHOLD);
}
```

测试一个线程池，涉及到对其执行策略的大量要素的测试：当需要时就创建额外的线程，不需要时就不要创建；当需要时就回收空闲线程，等等。创建一个覆盖了所有可能性的全面的测试套件是一件非常好的事情，但是大多数可能性都可以简单地、独立地进行测试。

通过使用自定义的线程工厂，我们可以对创建线程施加更多的控制。清单 12.8 中的 `TestingThreadFactory` 维护已创建线程的数目；在测试运行期间，测试用例可以验证已创建线程的数量。我们还可以扩展 `TestingThreadFactory`，让它返回一个可以记录自身何时终结的自定义 `Thread`，这样测试用例还可以验证线程是否遵守了执行策略，在适当的时候被回收。

清单 12.8 用于测试 `ThreadPoolExecutor` 的线程工厂

```
class TestingThreadFactory implements ThreadFactory {
    public final AtomicInteger numCreated = new AtomicInteger();
    private final ThreadFactory factory
        = Executors.defaultThreadFactory();

    public Thread newThread(Runnable r) {
        numCreated.incrementAndGet();
        return factory.newThread(r);
    }
}
```

如果核心池大小小于最大值，线程池会在执行的任务增多时相应地增长。向池提交几个耗时任务，会使池中的执行任务的数量在足够长的时间内都是常量，这就可以进行一些断言，比如测试池是否如期地扩展。参见清单 12.9。

清单 12.9 验证线程池扩展的测试方法

```
public void testPoolExpansion() throws InterruptedException {
    int MAX_SIZE = 10;
    ExecutorService exec = Executors.newFixedThreadPool(MAX_SIZE);

    for (int i = 0; i < 10 * MAX_SIZE; i++)
        exec.execute(new Runnable() {
            public void run() {
                try {
                    Thread.sleep(Long.MAX_VALUE);
                } catch (InterruptedException e) {
                    Thread.currentThread().interrupt();
                }
            }
        });
    for (int i = 0;
        i < 20 && threadFactory.numCreated.get() < MAX_SIZE;
        i++)
        Thread.sleep(100);
    assertEquals(threadFactory.numCreated.get(), MAX_SIZE);
    exec.shutdownNow();
}
```

12.1.6 产生更多的交替操作

因为大多数并发代码中潜在的错误都是低可能性的事件，所以测试并发错误就要千篇一律的重复，不过有些事情可能提高你发现错误的几率。在多处理器系统上，如果处理器的数量少于活动线程的数量，那么相比于单处理器或含有多个处理器系统，可以产生更多的交替行为，我们已经提到过这是如何做到的。类似地，在不同数量的处理器、不同种类的操作系统和处理器架构的情况下进行测试，可以发现那些特定于执行环境的问题。

有一个有用的技巧可以提高交替操作的数量，来更有效地探索程序的状态空间，那就是在访问共享状态的操作期间，使用 `Thread.yield` 激发更多的上下文转换。（这项技术的有效性是与平台相关的，因为 JVM 只是简单地把它当作一个 no-op [JLS 17.9]；使用短暂而非零的 `sleep`，尽管会有些慢，但是更可靠。）清单 12.10 中的方法将转账信息从一个账户转到另一个上；在两次更新操作之间，“所有账户的总额等于零”这条不变约束可能遭到破坏。有时在一个操作之间进行 `yield`，你可以在那些没有使用充足同步去访问状态的代码中，激活时序敏感的 bug。为了测试而加入的这些调用，在生产环境中还要将它们删除，这是很麻烦的。而面向方面编程（AOP）工具的使用，可以减少这些麻烦。

清单 12.10 使用 Thread.yield 产生更多的交替操作

```
public synchronized void transferCredits(Account from,
                                         Account to,
                                         int amount) {
    from.setBalance(from.getBalance() - amount);
    if (random.nextInt(1000) > THRESHOLD)
        Thread.yield();
    to.setBalance(to.getBalance() + amount);
}
```

12.2 测试性能

性能测试通常是功能测试的延伸。事实上，在性能测试中包含一些基本的功能测试，这一做法永远是值得的，这样就能确保你正在做性能测试的代码肯定是正确的。

尽管性能与功能测试之间有明确的迭交，但是它们目标终归是不同的。性能测试要探寻具有代表性的用例从头至尾的性能标准。通常，获得关于被测试对象的合理使用场景并不容易；理想上，测试应该能够反映出被测试对象是如何运用在应用程序中的。

在某些情况下，合理的测试场景是很容易发现的。几乎所有的生产者-消费者设计都会用到有限缓存，所以很明显的，要去测量生产者给消费者供应数据的吞吐量。我们只要简单地扩展 PutTakeTest，就能让它变成针对这一场景的性能测试。

性能测试的第二个目标是为那些基于经验的不同界限——线程数、缓存容量等等——选择一个合适的大小。这些数值可能会非常依赖于平台的特征（比如处理器类型甚至处理器的“stepping level（译注：表示处理器部件生产工艺的，它的版本会随着这一系列处理器生产工艺的改进而增加。）”，处理器的数量，或者内存大小，需要动态地配置，这相当于为它们选择一个合理的值，可以广泛地应用在各种系统中。

12.2.1 扩展 PutTakeTest，加入时间特性

我们对 PutTakeTest 所作的主要扩展是让它测量运行花费的时间。与其测量一个单一操作的耗时，我们不如选择对整个运行计时，然后除以操作的数量，得到每个操作的耗时，这样可以获得更精确测试值。我们已经使用 CyclicBarrier 去启动和结束工作者线程了，所以我们只要使用一个关卡动作来测量启动和结束时间，就完成了对该测试的扩展。如清单 12.11 所示。

我们能够修改关卡的初始化，让 CyclicBarrier 的构造函数接受这个关卡动作，使用它来计时：

```
this.timer = new BarrierTimer();
this.barrier = new CyclicBarrier(npairs * 2 + 1, timer);
```

清单 12.11 基于关卡的计时器

```
public class BarrierTimer implements Runnable {
    private boolean started;
    private long startTime, endTime;

    public synchronized void run() {
        long t = System.nanoTime();
        if (!started) {
            started = true;
            startTime = t;
        } else
            endTime = t;
    }
    public synchronized void clear() {
        started = false;
    }
    public synchronized long getTime() {
        return endTime - startTime;
    }
}
```

修改 test 方法，让它使用基于关卡的计时器，如清单 12.12 所示。

我们可以从 TimedPutTakeTest 的运行中，学到几件事情。第一，生产者-消费者的吞吐量取决于不同参数组合的操作；第二，有限缓存如何根据不同数量的线程进行伸缩；第三，我们如何选择缓存边界的大小。要回答这些问题，需要针对不同的参数组合进行测试，所以我们需要一个主测试驱动程序（main test driver），如清单 12.13 所示。

图 12.1 所示的是在一个 4 路机器上，使用 1, 10, 100 和 1000 4 种容量的缓存，得到的测试结果。我们立刻能够看到大小为 1 的缓存导致了非常糟糕的吞吐量；这是因为每个线程在被阻塞和等待其他线程之前，只能做一点事情。将缓存的长度提高到 10，能得到奇迹般的收效；但是 10 以后再继续提高，所获得的收效又开始减少。

添加更多的线程，只引起性能轻微的下降，这起初看起来令人有些困惑。其中的原因很难从数据中看出来，倒是可以在测试运行中简单地通过 CPU 的性能指标（比如 perfbar）看出来：即使线程有很多，却没有更多的计算让它们执行，大多数 CPU 时间都花在线程的阻塞和解除阻塞上面了。因为更多的线程在做相同的事情，所以大量 CPU 都处于闲置状态，不会过多影响性能。

但是，对待从上面数据中得出的结论应该谨慎——如果生产者-消费者程序使用有限缓存，你就可以无休止地向它加入线程。这个测试在模拟**应用程序**时，掺入了太多人为的因素；生产者不费什么力气就能生成一个条目，将它置入队列，同时消费者也不用费什么

清单 12.12 使用基于关卡的计时器进行测试

```
public void test() {
    try {
        timer.clear();
        for (int i = 0; i < nPairs; i++) {
            pool.execute(new Producer());
            pool.execute(new Consumer());
        }
        barrier.await();
        barrier.await();
        long nsPerItem = timer.getTime() / (nPairs * (long)nTrials);
        System.out.print("Throughput: " + nsPerItem + " ns/item");
        assertEquals(putSum.get(), takeSum.get());
    } catch (Exception e) {
        throw new RuntimeException(e);
    }
}
```

清单 12.13 TimedPutTakeTest 的驱动程序

```
public static void main(String[] args) throws Exception {
    int tpt = 100000; // 每个线程尝试的次数
    for (int cap = 1; cap <= 1000; cap *= 10) {
        System.out.println("Capacity: " + cap);
        for (int pairs = 1; pairs <= 128; pairs *= 2) {
            TimedPutTakeTest t =
                new TimedPutTakeTest(cap, pairs, tpt);
            System.out.print("Pairs: " + pairs + "\t");
            t.test();
            System.out.print("\t");
            Thread.sleep(1000);
            t.test();
            System.out.println();
            Thread.sleep(1000);
        }
    }
    pool.shutdown();
}
```

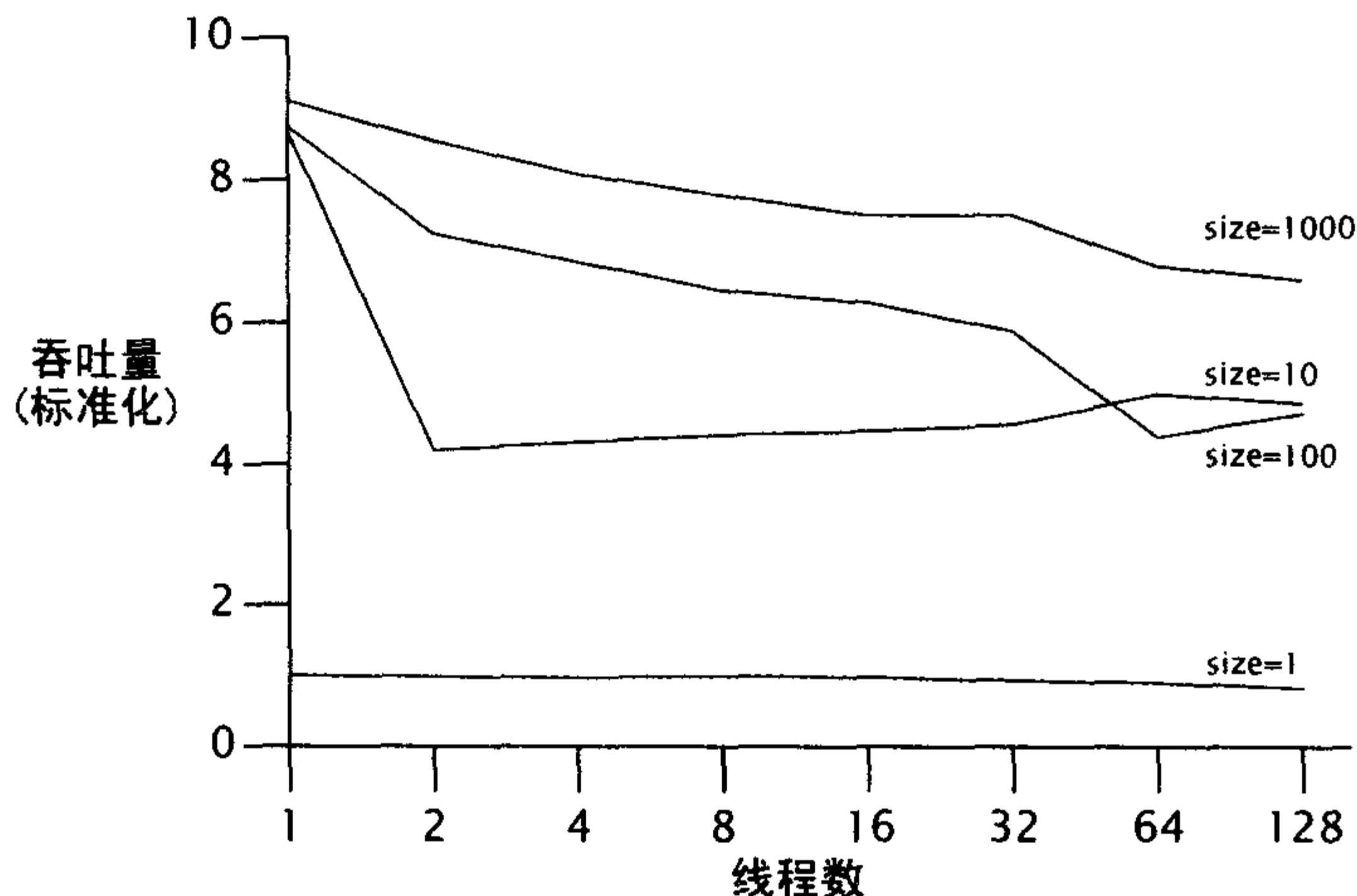


图 12.1 不同容量缓存下 TimedPutTakeTest 的运行效果

力气，就能获取一个条目。在真实的生产者-消费者应用程序中，如果工作者线程要通过一些复杂的工作来生产和消费条目（就像通常的情况那样），那么前面那种 CPU 的闲置状态就会消失，多线程所产生的效果就会变得明显起来。这个测试的主要目的，是测量在生产者和消费者通过有限缓存交换数据时，哪个约束条件影响了总体的吞吐量。

12.2.2 比较多种算法

虽然 BoundedBuffer 是一种相当可靠的实现，它的运行机制也非常合理，但是它还不足以和 ArrayBlockingQueue 与 LinkedBlockingQueue 相提并论（这也解释了为什么这种缓存算法没有被选入类库中）。java.util.concurrent 中的算法已经被选择并调整到我们已知的最佳性能状态，这其中部分地用到了像本章中描述的测试；不过这些算法还能提供更为广阔的功能⁶。BoundedBuffer 的运行效率不高，其主要原因是：put 和 take 操作分别都有多个操作可能遇到竞争——获取一个信号量，获取一个锁，释放信号量。在其他的实现方法中存在的竞争点少很多，这些点上会有与其他线程的竞争。

假设上述 3 个类都包含能容纳 256 个元素的缓存，使用不同的 TimedPutTakeTest，运行在双多线程核心的机器上，图 12.2 显示了比较后的吞吐量。这个测试表明 LinkedBlockingQueue 的伸缩性好于 ArrayBlockingQueue。这最初看起来有些奇怪：链接队列每次插入链接节点对象时，必须分配一个链接节点对象，这看上去比基于数组的

⁶ 如果你是一个并发专家，有胆略舍弃一些已提供的功能，那么你可能超越它们。

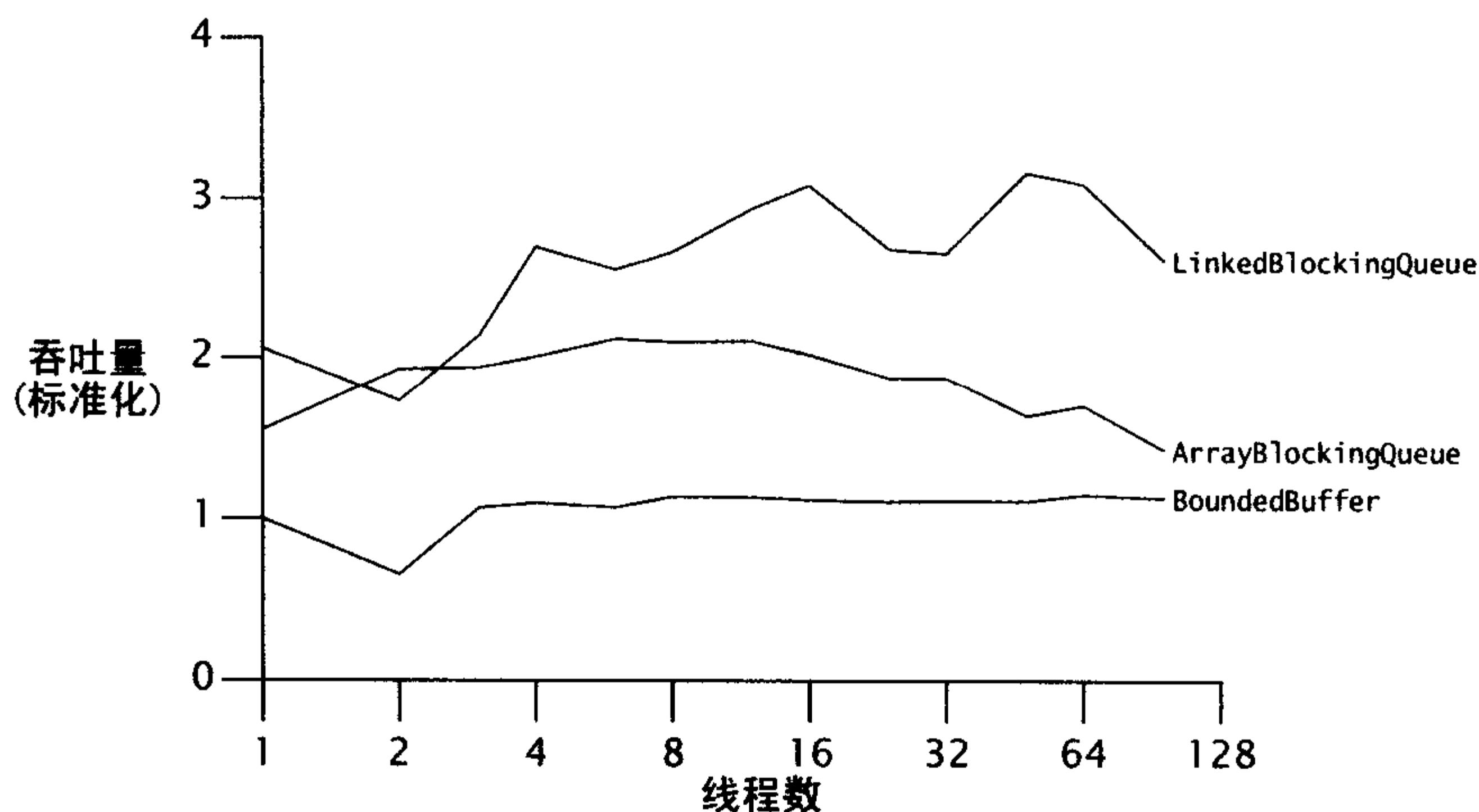


图 12.2 阻塞队列不同实现之间的比较

队列做了更多的工作。然而，尽管它用在分配与 GC（垃圾回收）的开销更多，但是链接队列的 `put` 和 `take` 操作允许有比基于数组的队列更好的并发访问。这是因为，最佳的链接队列算法允许队列的头和尾彼此独立地更新。由于分配操作通常是线程本地的，算法通过多做一些分配操作，可以降低竞争，这样的算法通常具有更好的可扩展性。（这是又一个实例：基于直觉的传统性能调优，与实际情况所需要的可伸缩性背道而驰。）

12.2.3 测量响应性

到目前为止，我们已经关注过测量吞吐量的问题了，这通常是并发程序里最重要的性能指标。但也有时候，知道一个独立的动作完成要花费多少时间，也是非常重要的。这种情况下，我们要测量的是服务时间的**差异性**。有时，为了能让我们获得较小的服务时间的差异性，那么允许较长的平均服务时间是有意义的；“可预言性”也是一个很有价值的性能特征。测量差异性，有助于我们对一些关于服务质量（quality-of-service）的问题进行估计，比如“在 100 毫秒内能够成功的操作的百分比是多少？”

任务完成时间的柱状图，通常是最佳的可视化服务时间差异性的方式。差异性的测量，比平均值的测量略微困难些——除了总计的完成时间，你还需要追踪每个任务的完成时间。因为计时器的粒度会成为测量独立任务时间的一个因素（一个独立的任务可能小于或者接近于最小的时钟滴答，这会破坏任务执行期间的测量），为了避免测量中人为的影响，我们可以测量 `put` 和 `take` 操作批处理语句的运行时间。

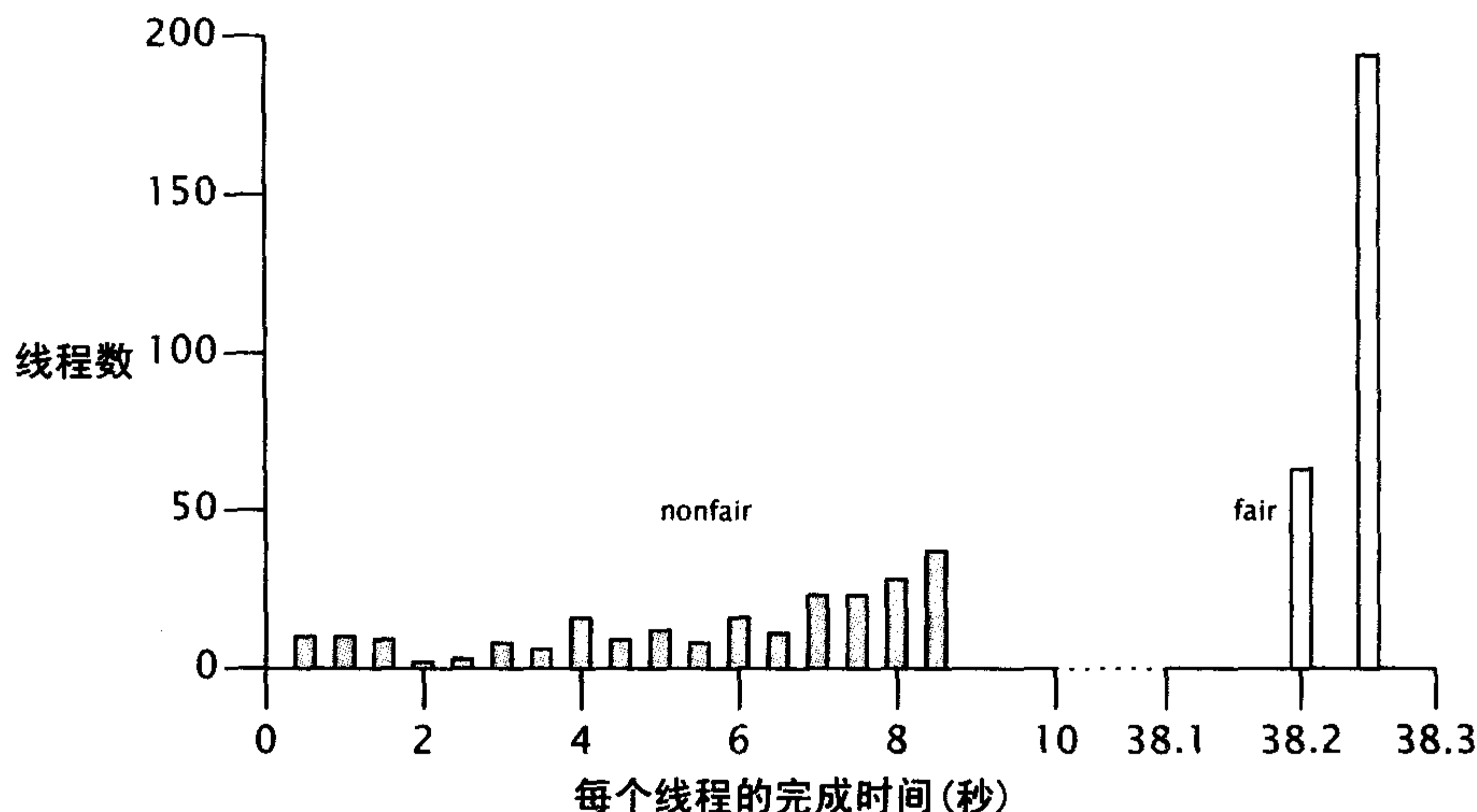


图 12.3 使用默认(非公平)和公平信号量的 TimedPutTakeTest 的完成时间柱状图

图 12.3 显示了不同的 TimedPutTakeTest 中每个任务的完成时间, TimedPutTakeTest 使用大小为 1 000 的缓存, 同时存在 256 个并发任务, 每一个都会分别使用非公平(隐蔽栅栏, shaded bars)和公平(开放栅栏, open bars)的信号量, 反复 1 000 个条目。(13.3 节会对锁和信号量的公平/非公平排队进行比较。)非公平信号量的完成时间, 跨度从 104 到 8 714ms, 前后相差超过 80 倍。在同步控制中强迫更多的公平性, 可以缩减这一跨度; 在 BoundedBuffer 中做到这一点很简单, 只要将信号量初始化为公平模式就可以了。如图 12.3 所示的那样, 这成功地缩减了差异性(现在的跨度为 38 194 到 38 207ms), 但不幸的是, 这也严重地影响了吞吐量。(一个长时间运行的测试, 运行更多具有普遍意义的任务, 吞吐量可能会有更为明显的下降。)

我们在前面看到了, 过小的缓存大小会引起过多的上下文转换以及糟糕的吞吐量, 几乎每一个操作都涉及了上下文转换, 因此即使在非公平的模式下也不能幸免。维护公平性的开销所带来的主要表象是线程阻塞。我们可以把缓存的长度定为 1, 再重新运行这个测试, 可以看出现在非公平信号量与公平信号量的执行效果已经不相上下了。如图 12.4 所示, 这种情况下公平性既没有让平均完成时间更糟, 又使差异性变得更好。

所以, 除非线程总是由于密集的同步条件而持续地被阻塞, 非公平的信号量通常能够提供更好的吞吐量, 公平的信号量提供更低的差异性。因为结果差异如此之大, 所以 Semaphore 强迫它的客户来决定针对哪一个特性进行优化。

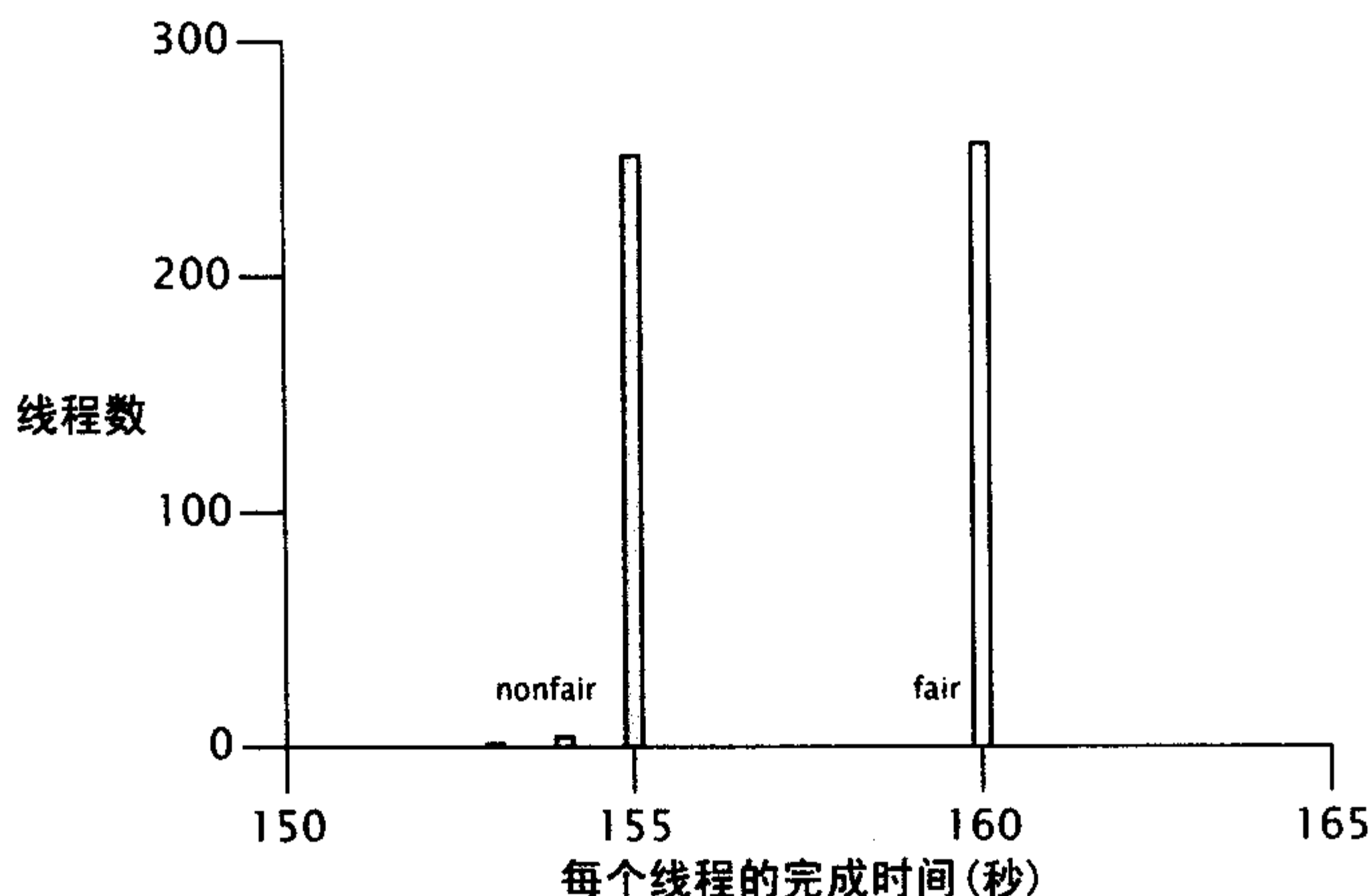


图 12.4 表示单元素缓存的 TimedPutTakeTest 完成时间的柱状图

12.3 避免性能测试的陷阱

理论上，开发性能测试是简单的——发现一个典型的使用场景，编写一段程序以多次执行这一场景，并对它计时。在实际中，你还必须要提防很多编码的陷阱，它们会导致性能测试产生毫无意义的结果。

12.3.1 垃圾回收

垃圾回收的时序是不可预知的，在一个测量数据的测试运行中，任何时候垃圾回收器都有可能运行。如果一个测试程序反复执行了 N 次都没有触发垃圾回收，但是在第 $N+1$ 次的时候触发了一次垃圾回收，运行中大小上差之毫厘，可能引发每次迭代运行时间谬以千里（但却不真实）。

有两种策略可以避免垃圾回收对你的结果带来的误差。第一种策略是，确保在测试运行的整个期间，垃圾回收根本不会执行（通过调用 JVM 时使用 `-verbose:gc` 可以做到）；另一种策略是，你能确保执行测试期间垃圾回收器运行多次，这样测试程序能够充分反映出运行期间的分配与垃圾回收的开销。通常后一种策略更佳——它需要更长的测试时间，并且更可能反映现实环境下的性能。

大多数基于生产者-消费者设计的应用程序都会涉及相当数量的内存分配与垃圾回收的操作——生产者分配新对象，随后被消费者使用并丢弃。运行有限缓存测试的时间足够长，就会引发多次垃圾回收，从而得到更加精确的结果。

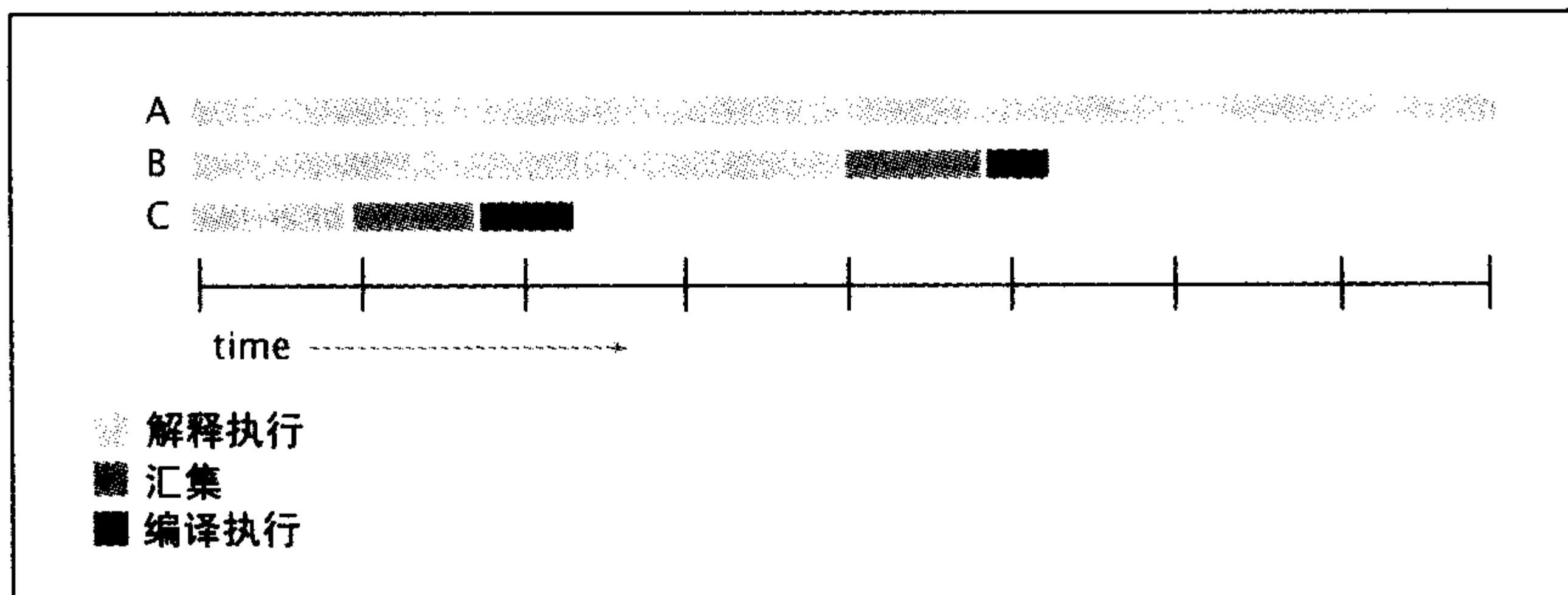


图 12.5 动态编译带来的结果偏差

12.3.2 动态编译

对于像 Java 这样的动态编译语言，编写和解读它们的性能基准测试，要比 C 或 C++ 这样的静态编译语言困难得多。HotSpot JVM（以及其他最新的 JVM）结合了字节码解释和动态编译。当一个类被首次加载后，JVM 会以解释字节码的方式执行。如果一个方法运行得足够频繁，动态编译器最终会将它挑出来，转成本机代码；当编译完成后，执行方式将由解释执行转换到直接执行。

编译的时机是不可预知的。你的时序测试应该在所有代码编译完成后再运行；测量解释执行的代码速度是没有意义的，因为多数程序运行得足够长后，所有频繁执行的代码路径都会被编译。允许编译器可以在测试期间运行，会从两方面给测试结果带来偏差：编译消耗 CPU 资源，另外测量解释型与编译型混合代码的运行时间，所得的性能指标并没有很大的意义。图 12.5 显示了动态编译是如何给结果带来偏差的。3 条时间线代表了相同次数迭代的执行：时间线 A 代表的是全部为解释执行，B 代表了中途开始编译执行，C 代表了编译是提前于运行的。每次操作运行时间的测量，在很大程度上都受到编译运行的开始点的影响⁷。

由于很多原因，代码同样可能被解除编译（decompiled，回复到解释型执行）以及再次重编译，比如加载了一个类，会破坏以前的编译结果，或者在收集到充足的统计信息后，决定为了不同的优化，而重编译一个代码路径。

有一种方式可以避免编译对你的结果产生的影响，那就是让你的程序长时间运行（至少几分钟），这样编译过程和解释执行仅仅占了总体运行时间的很小一部分。另一种方法是让代码先进行不做测量的“热身”运动，使它得以充分执行，这样在开始计时前，代码就被完全编译了。在 HotSpot 中，运行程序时使用 `-XX:+PrintCompilation`，那么程序会在

⁷ JVM 会选择是在应用程序线程中，还是后台线程中执行编译；每种选择都会以不同的方式影响到时序的结果。

动态编译运行时打印出信息，你可以通过它来验证动态编译发生在测试运行前，而不是运行中。

在相同 JVM 实例中多次运行同一个测试，可以用来验证测试方法的有效性。第一组结果应该作为“热身”而被丢弃；在剩下的结果组中还会观察到不一致的结果，这就要求我们还需要进一步检查测试，来确定为什么结果是不可重复的。

JVM 会启动不同的后台线程去执行常规任务。当在单一的运行中测量多个**不相关**的计算密集型活动时，在不同的测量活动之间置入显式的暂停，是个很好的做法。这会让 JVM 得以与后台任务保持一致，而且把来自被测线程的干扰降至最低。（然而，当测量多个相关的活动时，比如相同测试的多次运行，以这种方式拒绝 JVM 后台任务，可能给出不真实的乐观结果。）

12.3.3 代码路径的非真实取样

运行时编译器使用收集到的信息来帮助优化已经编译的代码。JVM 被允许使用执行期间的细则信息以此产生更好的代码，这意味着在一个程序中编译方法 *M* 生成的代码可能与另一个不同。在某些情况下，JVM 可以进行基于下述假设的优化：这种优化只是临时的，过后，当它们不适合动态编译模式后，JVM 会使被编译的代码无效，从而把它们回收⁸。

作为结果，你的测试程序不仅应该尽量地接近于一个典型应用程序的使用模式，还应该尽量覆盖这个应用程序会用到的代码路径的集合。这一点很重要。否则，动态编译器会针对一个纯粹单线程化的测试程序，进行一些专门的优化。只要一个真实的应用程序中包含了一点偶发的并行，这种优化就不会出现在该程序中。因此，即便你只是想测量单线程的性能，也应该与多线程的性能测试结合在一起。（这个问题不会出现在 `TimedPutTakeTest` 中，因为即使是最小的测试用例都用了两个线程。）

12.3.4 不切实际的竞争程度

并发的应用程序总是交替执行两种非常不同的工作：第一是访问共享数据，比如从共享工作队列中获取下一个任务，第二是线程本地的计算（执行任务，假设任务自身并不访问共享数据）。依赖于两种工作类型的相关特性，应用程序会经历不同级别的竞争，并表现出不同的性能与伸缩性行为。

如果有 *N* 个线程从共享工作队列中获取任务并执行，这些任务都是计算密集型的、耗

⁸ 举例来说，如果当前没有加载可以覆写某一方法的类，JVM 将使用单一调用转换 (monomorphic call transformation) 将一个虚拟方法调用转变为一个直接方法调用。但是如果后来加载了一个类，覆写了该方法，这会使已编译的代码失效。

时的（但并未频繁地跨线程访问数据），这种情况几乎没有竞争；吞吐量只受限于可用的 CPU 资源。另一方面，如果任务的生命周期很短，在工作队列上就会存在大量竞争，此时吞吐量受限于同步的开销。

为了获得有实际意义的结果，并发性能测试，除了需要考虑协调并发的因素，应该尽量去模拟让线程本地的计算由某一个特有的应用程序来完成。如果每个任务在真实应用程序中完成的工作，与测试程序相比，其本质和范围有相当大的不同，那么得到的关于性能瓶颈位置的结论将是毫无根据的。我们在 11.5 节看到过，对于基于锁的类，比如同步的 Map 实现，访问锁的时候是否总是竞争的或者是非竞争的，这会对吞吐量产生令人惊讶的影响。本节中的测试除了不断地访问 Map，并没有做其他事；尽管只有两个线程，所有访问 Map 的尝试都是有竞争的。但是，如果一个应用程序在每次访问到共享的数据结构后，还会做大量的线程本地计算，那么可以将竞争级别降到足够低，来提供令人满意的性能。

从这个角度看，TimedPutTakeTest 对有些应用程序来说并不是很好的模式。因为工作者线程并没有做很多事情，吞吐量只受限于在协调过程中产生的开销。其实对于其他的在生产者和消费者之间通过有限缓存交换数据的应用程序来说，情况并不总是这样的。

12.3.5 死代码的消除

在任何语言中，编写良好的基准测试程序，都要面对的挑战之一是：优化过的编译器擅长发现并遗弃死代码（dead code），这些代码不会对结果产生任何影响。由于基准测试通常不会进行任何计算，它们就成为了这类优化的目标。大多数时候，编译器从程序中删减掉死代码都是一件好事，但是对于基准测试程序来说，这是个大问题，因为被测试到的内容比你认为的要少。如果幸运，优化器将删除你的整个程序，这样你会得到一份明显是不真实的测试数据。如果不幸运，死代码的消除仅仅会通过某些特性加快你程序的执行，这将使你的测试被赋予其他的解释。

尽管在静态编译语言的基准测试中，也存在被忽略的死代码这个问题，但是你可以观看机器代码，发现你的部分程序不见了，所以很容易检查出基准测试里的已经被编译器丢弃的大块代码。但是在动态编译的语言中，访问这个信息就不容易了。

很多微型的基准测试在 HotSpot 的 -server 编译模式下的运行效果，要好于 -client 模式，这不仅是因为 server 模式的编译器可以产生更有效的代码，同时还因为这种模式更擅长优化死代码。不幸的是，由于死代码会被遗弃，这使得你的基准测试程序中短小的工作，并不会像实际工作的代码那样运行。但是，在多处理器系统上，无论生产环境还是测试环境，你都应该选择 -server 模式而不是 -client 模式——只不过你写的测试必须是不受死代码忽略行为影响的。

编写有效的性能测试，就需要哄骗优化器不要把你的基准测试当作死代码而优化掉。这需要每一个计算的结果都要应用在你的程序中——以一种不需要的同步或真实计算的方式。

在 `PutTakeTest` 中，我们计算了在队列中被添加与删除的元素的总和，但是如果我们没有真正使用总和的数值，它仍然有可能被优化掉。幸好我们刚好用它去验证算法的正确性，不过你也可以通过打印一个值来确保它被真正用到了。但是你应该避免在测试实际运行中做 I/O 操作，以免破坏运行时间的测量。

有一个代价很低的技巧，可以在不引入过高开销的大前提下，防止一个计算被优化掉，这就是计算期望对象中域的 `hashCode`，让它与一个无意义的数值进行比较，比如当前 `System.nanoTime` 的值，如果它们碰巧相等，就打印一个没有实际意义、可被忽略的消息：

```
if (foo.x.hashCode() == System.nanoTime())
    System.out.print(" ");
```

这个比较很少能成功，如果成功了，它唯一的影响是会在输出中插入到一个无关痛痒的空字符。（`print` 方法会缓存输出，直到调用了 `println`，所以，即使 `hashCode` 和 `System.nanoTime` 的返回值罕见地相等了，也没有真正地执行 I/O。）

不仅每个计算结果都应该被用到，而且结果应该是不能预先猜测的。否则，一个聪明的动态优化编译器会用预计算的结果取代计算过程。我们通过构建 `PutTakeTest` 来解决这个问题，任何以静态数据作为输入的测试程序，都会受到这项优化的干扰。

12.4 测试方法补遗

虽然我们愿意相信一个有效的测试程序能够“发现所有 bug”，但这是一个不切实际的目标。NASA 投入到测试的工程资源多于任何商业运作应该提供的（据估计，他们每雇用一名开发者，就会相应雇用 20 测试人员）——他们生产的代码仍然不是完美无瑕的。在复杂的程序中，再多的测试也无法发现所有的错误。

测试的目标不是更多地发现错误，而是提高信心，相信代码能够如期地工作。因为设想发现所有 bug 是不现实的，所以质量审查（quality assurance, QA）计划的目标应该定为利用现有的测试资源，最大程度上获得对代码的信心。并发程序会比顺序程序出更多的问题，所以想要获得同样级别的信心，还需要更多的测试。目前为止，我们已经关注了创建有效的单元测试与性能测试时使用的主要技术。在获得对并发类正确行为的信心时，测试如此重要，但并不是 QA 的唯一方法。

不同的 QA 方法，在发现某些类型的错误时更有效，而对于某些错误则无效。使用一些补充的测试方法，比如代码审查、静态分析，你可以获得比在单一方法下更多的信心。

12.4.1 代码审查

就像单元测试和压力测试对于发现并发 bug 的有效性和重要性一样，没有什么可以取代严格的多人代码审查。（另一方面，代码审查也不能取代测试。）即使并发专家也会犯错；花些时间让其他人也来审查代码总是值得的。你可以，并也应该设计测试，以使发现安全性错误的机会最大化，同时还应该频繁地运行它们；但是你也不应该忽略让代码作者以外的人仔细地审查并发代码。专家级的并发程序员比大多数测试程序能更有效地发现隐蔽的竞争。（同时，平台的问题，比如 JVM 的实现细节或者存储器的存储模型，都会在特定的硬件或软件配置下，屏蔽一些 bug 的出现。）代码审查还有另外的好处：它不仅可以发现错误，通常还能改善描述实现细节的注释的质量，因而降低了后期维护的成本和风险。

12.4.2 静态分析工具

在写作本书的时候，**静态分析工具**（static analysis tool）正在快速地崭露头角，它正成为正规的测试和代码审查时一个有效的组件。静态代码分析是这样的过程：不执行代码，对它进行分析。代码核查工具可以分析类，需要常见的**错误模式**（bug pattern）的实例。像开源的 FindBugs⁹ 这种静态分析工具，都包含了很多常见代码错误的**错误模式侦测器**（bug-pattern detectors），其中的很多错误都是很容易在测试与代码审查中遗漏的。

静态分析工具会生成一个警告清单，它必须被手工地检查，以确定这些警告是否代表了一个真正的错误。在以前，像 lint 这种工具会产生很多错误的警告，吓跑了开发者，但是 FindBugs 这类工具已经被调优，只产生很少的错误警告。静态分析工具仍然有些原始（尤其是它们在与开发工具与开发生命周期整合过程中），但是作为测试过程的有价值的补充，它已经足够有效了。

在写作本书的时候，FindBugs 已经包含了下列并发相关的错误模式的侦测器，而且不断地还有更多的被加入：

不一致的同步性。很多对象都遵循下面的同步策略：使用对象的内部锁保护所有变量。如果频繁访问一个域的线程并未总是持有 this 锁，这就可能暗示同步策略没有被一贯地坚持。

Java 类没有正式的同步规约，所以分析工具必须对同步策略进行猜测。在今后，如果

⁹ <http://findbugs.sourceforge.net>

@GuardedBy 这种 Annotation 可以标准化, 核查工具解析 Annotation, 而不用去猜测变量与锁之间的关系, 因而可以提高分析的质量。

调用 Thread.run。Thread 实现了 Runnable, 有一个 run 方法。然而, 直接调用 Thread.run 几乎总是个错误; 程序员应该调用 Thread.start。

未释放的锁。不同于内部锁, 显式锁 (参见第 13 章) 在控制退出了它们被请求的范围时, 不会自动释放。标准的技巧是从 finally 块中释放锁; 否则, 遇到 Exception 事件后, 锁可以保留在未释放的状态。

空 synchronized 块。虽然空 synchronized 块在 Java 存储模型下是有语意的, 但是它们被频繁地误用了。无论开发者试图去解决的是什么问题, 通常都会有更佳解决方案。

双检查锁。在惰性初始化时, 双检查锁作为降低同步开销的技巧是有缺陷的 (参见 16.2.4 节)。它涉及的问题是, 读取一个共享可变域时, 缺少适当的同步。

从构造函数中启动线程。在构造函数中启动线程, 会引入子类化问题的风险, 同时还会引起 this 引用从构造函数中溢出。

通知错误。notify 和 notifyAll 方法预示着, 一个对象的状态可能已经以某种方式发生改变, 进而那些正在等待与该对象相关联的条件队列的线程, 会被解除阻塞。只有在与条件队列关联的状态发生改变后, 才应该去调用这些方法。如果在一个 synchronized 块中调用了 notify 和 notifyAll, 但是没有修改任何状态, 这通常都可能是个错误。(参见第 14 章。)

条件等待错误。在一个条件队列中等待时, Object.wait 和 Condition.await 不仅应该在持有正确的锁的情况下, 在循环中被调用, 而且要在测试过一些状态谓词之后 (参见第 14 章)。调用 Object.wait 和 Condition.await 时, 不持有锁、不在循环中、或者没有测试某些状态断言, 这总是一个错误。

误用 Lock 和 Condition。使用 Lock 作为 synchronized 块的锁的代替品, 就像查找/替换一样, 只要将 wait 调用替换成 Condition.await 调用 (由于前者在它第一次被调用时, 会抛出 IllegalMonitorStateException, 因此它需要在测试中被捕获)。

休眠或等待时持有锁。调用 Thread.sleep 时持有锁, 会导致其他线程在很长的一段时间内无法执行, 因此这是一个潜在的严重的活跃度危险。调用 Object.wait 或者 Condition.await 时持有锁, 会引发相同的危险。

自旋循环。如果代码除了循环检查（忙等待）一个域是否有期望值之外，不做任何事情，就会浪费 CPU 时间，并且如果域不是 `volatile` 类型的，就无法保证循环检查可以终止。如果需要等待一个状态转换的发生，闭锁或者条件等待通常是更好的技术。

12.4.3 面向方面的测试技术

在写作本书的时候，面向方面编程（AOP）技术只有很有限的并发能力，因为大多数流行的 AOP 工具还不支持在同步点的 `pointcut`。但是，AOP 可以用来断言不变约束，或者遵从同步策略的一些方面（`aspect`）。举个例子，在（Laddad, 2003）提供的示例中，使用一个方面包装了所有对非线程安全的 Swing 方法的调用，断言这些调用是发生在事件线程中的。由于不需要改变代码，这项技术的应用起来很简单，它可以发现可能出错的发布和线程限制相关的错误。

12.4.4 统计与剖析工具

大多数商业的统计工具都提供了一些对线程的支持。这会改变程序的特征集和执行效果，但是通常提供了对于程序正在做的事情的一个深入的洞察力（虽然统计工具通常是插入式的，可以极大地干扰程序的时序和行为）。它们通常还为每个线程提供一个时间线显示，不同的线程状态（可运行态、等待锁的阻塞、等待 I/O 的阻塞，等等）用不同的颜色表示。这样可以显示你的程序在使用可用的 CPU 资源时，效率如何；如果效果很差，还可以显示在哪里查找原因。（很多统计工具还声称提供了特性，可以识别出哪一个锁引起的竞争，但是实践中，比起在分析程序的锁行为时所期望的能力，这些特性不过是些迟钝的工具而已。）

内置的 JMX 代理也为监控线程行为提供了有限的特性。`ThreadInfo` 类就包含了线程的当前状态，而且当线程被阻塞时，它还包含引起阻塞的锁或者条件队列的信息。如果激活了“线程竞争监视器”特性（由于对性能的影响，这个属性默认是关闭的），`ThreadInfo` 还会包括很多线程阻塞等待一个锁或通知的时间，以及它花费在等待上累计的时间。

总结

测试并发程序的正确性是一项极大的挑战，因为并发程序很多可能的失败模式都是低可能性的事件，它们很容易受到时序、加载和其他一些难以再现的条件的影响。更进一步而言，在测试基础架构时，还会引入额外的同步或者分时的约束，这些会屏蔽被测代码中的并发问题。测试并发程序的性能同样是一项不小的挑战；比起使用像 C 这种静态编译语

言编写的程序，Java 程序格外地难以测试，因为动态编译、垃圾回收以及自动的优化，都会影响对时间的测量。

为了尽可能发现潜在的 bug，避免在生产环境中才发现它们，应该在运用传统的测试技术（谨慎地避免这里讨论过的各种缺陷）的同时，结合代码审查和自动化分析工具。每种技术都会发现其他技术可能忽略掉的问题。

PART

第 4 部分

高级主题

Advanced Topics

显式锁

Explicit Locks

在 Java 5.0 之前,用于调节共享对象访问的机制只有 `synchronized` 和 `volatile`。Java 5.0 提供了新的选择: `ReentrantLock`。与我们已经提到过的机制相反, `ReentrantLock` 并不是作为内部锁机制的替代,而是当内部锁被证明受到局限时,提供可选择的高级特性。

13.1 Lock 和 ReentrantLock

清单 13.1 中所示的 `Lock` 接口,定义了一些抽象的锁操作。与内部加锁机制不同, `Lock` 提供了无条件的、可轮询的、定时的、可中断的锁获取操作,所有加锁和解锁的方法都是显式的。`Lock` 的实现必须提供具有与内部加锁相同的内存可见性的语义。但是加锁的语义,调度算法,顺序保证,性能特性这些可以不同。(`Lock.newCondition` 将在第 14 章讲解。)

清单 13.1 Lock 接口

```
public interface Lock {  
    void lock();  
    void lockInterruptibly() throws InterruptedException;  
    boolean tryLock();  
    boolean tryLock(long timeout, TimeUnit unit)  
        throws InterruptedException;  
    void unlock();  
    Condition newCondition();  
}
```

`ReentrantLock` 实现了 `Lock` 接口,提供了与 `synchronized` 相同的互斥和内存可见性的保证。获得 `ReentrantLock` 的锁与进入 `synchronized` 块有着相同的内存语义,释放 `ReentrantLock` 锁与退出 `synchronized` 块有相同的内存语义。(内存可见性在第 3.1

小节中涉及。) ReentrantLock 提供了与 synchronized 一样的可重入加锁的语义 (见 2.3.2 小节)。ReentrantLock 支持 Lock 接口定义的所有获取锁的模式。与 synchronized 相比, ReentrantLock 为处理不可用的锁提供了更多灵活性。

为什么要创建与内部锁如此相似的机制呢? 内部锁在大部分情况下都能很好地工作, 但是有一些功能上的局限——不能中断那些正在等待获取锁的线程, 并且在请求锁失败的情况下, 必须无限等待。内部锁必须在获取它们的代码块中被释放; 这很好地简化了代码, 与异常处理机制能够进行良好的互动, 但是在某些情况下, 一个更灵活的加锁机制提供了更好的活跃度和性能。

清单 13.2 提供了使用 Lock 接口的规范形式。这个模式在某种程度上比使用内部锁更加复杂: 锁**必须**在 finally 块中释放。另一方面, 如果锁守护的代码在 try 块之外抛出了异常, 它将永远都不会被释放了; 如果对象能够被置于不一致的状态, 可能需要额外的 try-catch, 或 try-finally 块。(当你在使用任何形式的锁时, 你总是应该关注异常带来的影响, 包括内部锁。)

忘记使用 finally 释放 Lock 是一个定时炸弹。当“不幸”发生的时候, 你将很难追踪到错误的发生点, 因为根本没有记录锁本应被释放的位置和时间。这就是 ReentrantLock 不能完全替代 synchronized 的原因: 它更加“危险”, 因为当程序的控制权离开了守护的块时, 不会自动清除锁。尽管记得在 finally 块中释放锁并不困难, 但忘记的可能性仍然存在¹。

清单 13.2 使用 ReentrantLock 保护对象状态

```
Lock lock = new ReentrantLock();
...
lock.lock();
try {
    // 更新对象的状态
    // 捕获异常, 必要时恢复到原来的不变约束
} finally {
    lock.unlock();
}
```

¹ FindBugs 拥有一个“未释放锁”的侦测器, 能够发现一个锁在获取它的代码路径之外未被释放。

13.1.1 可轮询的和可定时的锁请求

可定时的与可轮询的锁获取模式，是由 `tryLock` 方法实现，与无条件的锁获取相比，它具有更完善的错误恢复机制。在内部锁中，死锁是致命的——唯一的恢复方法是重新启动程序，唯一的预防方法是在构建程序时不要出错，所以不可能允许不一致的锁顺序。可定时的与可轮询的锁提供了另一个选择：可以规避死锁的发生。

如果你不能获得所有需要的锁，那么使用可定时的与可轮询的获取方式（`tryLock`）使你能够重新拿到控制权，它会释放你已经获得的这些锁，然后再重新尝试（或者至少会记录这个失败，抑或采取其他措施）。清单 13.3 展示了一种替代方法，能够解除 10.1.2 小节中动态的顺序死锁这个问题：使用 `tryLock` 试图获得两个锁，如果不能同时获得两个，就回退，并重新尝试。休眠时间由一个特定的组件管理，并由一个随机组件减小活锁发生的可能性。如果一定时间内，没有能获得所有需要的锁，`transferMoney` 返回一个失败状态，这样操作就能优雅地失败了。（更多使用可轮询锁避免死锁的例子参见 [CPJ 2.5.1.2] 和 [CPJ 2.5.1.3]。）

对于实现那些具有时间限制的活动，定时锁同样非常有用（参见 6.3.7 小节）。当这样的活动调用了阻塞的方法，定时锁能够在时间预算内设定相应的超时。如果活动在期待的时间内没能获得结果，这个机制使程序能够提前返回。使用内部锁一旦开始请求，锁就不能停止了，所以内部锁为实现具有时限的活动带来了风险。

第 134 页，清单 6.17 中旅游门户网站的例子，为请求出价的每一个汽车租赁公司创建了一个独立的任务。请求出价可能会包含一些基于网络的请求机制，比如 Web Service 请求。但是请求出价可能同样需要对稀缺资源的独占访问，比如直接通向公司的通信线路。

我们在 9.5 小节看到了一种确保串行化访问资源的方法：一个单线程化的 `Executor`。另一个方法是使用独占的锁来守护对资源的访问。清单 13.4 中的代码试图在 `Lock` 守护的共享线路中发送一条消息，但是如果它不能在预定时间内完成，就会优雅地失败。定时的 `tryLock` 与独占加锁相互配合，使它在这样具有时间限制的活动切实地发挥作用。

13.1.2 可中断的锁获取操作

正如定时锁的获得操作允许在限时活动内部使用独占锁，可中断的锁获取操作允许在可取消的活动中使用。7.1.6 节提出了几种机制，比如请求不响应中断的内部锁。这些不可中断的阻塞机制使得实现可取消的任务变得复杂了。当你正在响应中断的时候，`lockInterruptibly` 方法能够使你获得锁，并且由于它是内置于 `Lock` 的，你因此不必再创建其他种类不可中断的阻塞机制。

清单 13.3 使用 tryLock 避免锁顺序死锁

```
public boolean transferMoney(Account fromAcct,
                             Account toAcct,
                             DollarAmount amount,
                             long timeout,
                             TimeUnit unit)
    throws InsufficientFundsException, InterruptedException {
    long fixedDelay = getFixedDelayComponentNanos(timeout, unit);
    long randMod = getRandomDelayModulusNanos(timeout, unit);
    long stopTime = System.nanoTime() + unit.toNanos(timeout);

    while (true) {
        if (fromAcct.lock.tryLock()) {
            try {
                if (toAcct.lock.tryLock()) {
                    try {
                        if (fromAcct.getBalance().compareTo(amount)
                            < 0)
                            throw new InsufficientFundsException();
                        else {
                            fromAcct.debit(amount);
                            toAcct.credit(amount);
                            return true;
                        }
                    } finally {
                        toAcct.lock.unlock();
                    }
                }
            } finally {
                fromAcct.lock.unlock();
            }
        }
        if (System.nanoTime() < stopTime)
            return false;
        NANOSECONDS.sleep(fixedDelay + rnd.nextLong() % randMod);
    }
}
```

清单 13.4 具有预定时间的锁

```
public boolean trySendOnSharedLine(String message,
                                   long timeout, TimeUnit unit)
    throws InterruptedException {
    long nanosToLock = unit.toNanos(timeout)
        - estimatedNanosToSend(message);
    if (!lock.tryLock(nanosToLock, NANOSECONDS))
        return false;
    try {
        return sendOnSharedLine(message);
    } finally {
        lock.unlock();
    }
}
```

一个可中断的锁获取操作的规范形式，要比一个普通的锁稍微复杂一些，因为需要两个 `try` 块。（如果可中断的锁获取操作抛出了 `InterruptedException`，那么标准的 `try-finally` 加锁的模式就非常有效了。）清单 13.5 使用 `lockInterruptibly` 实现了清单 13.4 中的 `sendOnSharedLine`，这样我们就可在可取消的任务中调用它了。定时的 `tryLock` 同样响应中断，因此当你在需要获取定时和可中断的锁时可以使用 `tryLock` 这个方法。

清单 13.5 可中断的锁获取请求

```
public boolean sendOnSharedLine(String message)
    throws InterruptedException {
    lock.lockInterruptibly();
    try {
        return cancellableSendOnSharedLine(message);
    } finally {
        lock.unlock();
    }
}

private boolean cancellableSendOnSharedLine(String message)
    throws InterruptedException { ... }
```

13.1.3 非块结构的锁

在内部锁中，获取和释放这样成对的行为是块结构的——总是在其获得的相同的基本程序块中释放锁，而不考虑控制权是如何退出阻塞块的。自动释放锁简化了程序的分析，并避免了潜在的代码错误造成的麻烦，但是有时需要更灵活的加锁规则。

在第 11 章中，我们已经看到减小锁的粒度是如何能够提高可伸缩性的。分离锁时不同的哈希链在哈希容器中使用不同的锁。在链表中，我们可以通过为**每个链表节点**应用相似的原则来减小锁的粒度，从而允许不同的线程独立地操作链表的不同部分。给定节点的锁守护链接的指针，数据就存储在该节点中，所以如果要遍历或者修改链表，我们必须得到这个锁，并持有它直到我们获得了下一个节点的锁；只有在这之后，我们才能释放前一个节点的锁。这项技术的例子被称为连锁式加锁或者锁联接，将在 [CPJ2.5.1.4] 中介绍。

13.2 对性能的考量

当 `ReentrantLock` 被加入到 Java 5.0 中时，它提供的竞争上的性能要远远优于内部锁。对于同步原语而言，竞态时的性能是可伸缩性的关键：如果有越多的资源花费在锁的管理和调度上，那用留给应用程序的就会越少。更好的实现锁的方法会使用更少的系统调用，发生更少的上下文切换，在共享的内存总线上发起更少的内存同步通信。耗时的操作会占用本应用于程序的资源。

Java 6 中使用了经过改善的管理内部锁的算法，类似于 `ReentrantLock` 使用的算法，从而大大弥补了可伸缩性的不足。图 13.1 分别表示在 Java 5.0 和预发布的 Java 6 版本，内部锁和 `ReentrantLock` 之间性能的差别，程序跑在 4 路的 Solaris 操作系统中。曲线表现了同一 JVM 版本下 `ReentrantLock` 相对于内部锁速度的提升。在 Java 5.0 中，`ReentrantLock` 能够给吞吐量带来相当不错的提升，但是在 Java 6 中，这两者非常接近²。测试程序与 11.5 小节中的相同，这一次比较的是分别由内部锁和 `ReentrantLock` 守护的 `HashMap` 的吞吐量。

在 Java 5.0 中，内部锁的性能在从单线程（无竞争）到多线程的变化过程中，性能急剧下降；`ReentrantLock` 的性能下降要小得多，显示出了更好的可伸缩性。但是在 Java 6 中就完全不同了——内部锁不再因竞争导致“崩溃”，两者的伸缩比例基本相等。

图 13.1 中的曲线图提醒了我们，像“X 比 Y 更快”这样的陈述只不过是短暂的。性能和可伸缩性是基于平台的，比如 CPU，处理器数量，高速缓存大小，JVM 特性，所有这些因素都会随时间而改变³。

² 尽管这张曲线图没有能显示出来，但 Java 5.0 和 Java 6 可伸缩性的区别的确是来源于内部锁的改进，而不是 `ReentrantLock` 的衰退。

³ 当我们开始这本书的写作的时候，在锁的可伸缩性方面 `ReentrantLock` 看起来还是最新的词汇。不到一年，内部锁已经获得相当可观的性能。性能不仅仅是个不断变化的目标，而且变化得非常快。

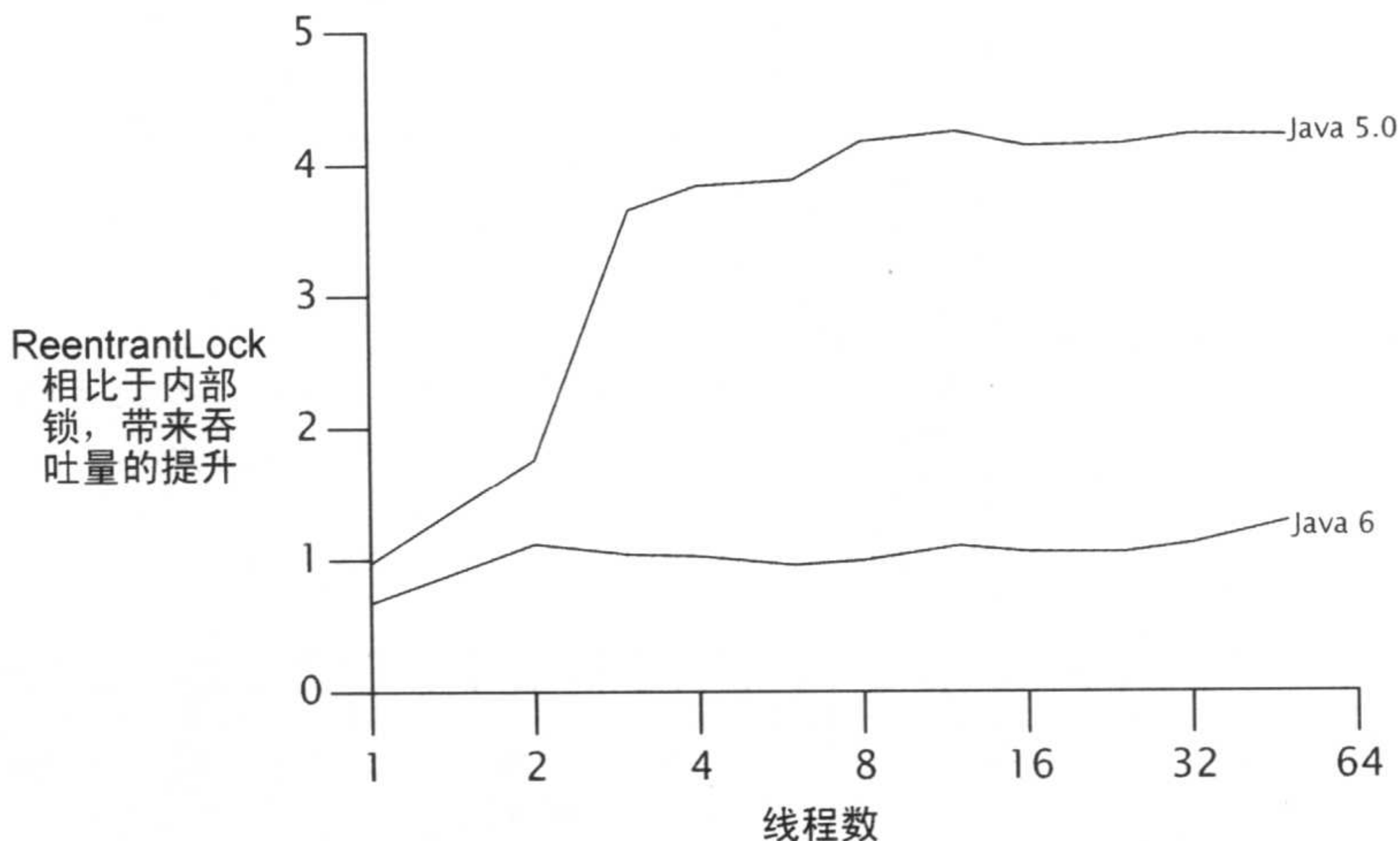


图 13.1 ReentrantLock 之于内部锁在 Java 5.0 和 Java 6 的性能提升的对比

性能是一个不断变化的目标；昨天的基准显示 X 比 Y 更快，这可能已经过时了。

13.3 公平性

ReentrantLock 构造函数提供了两种**公平性**的选择：创建**非公平锁**（默认）或者**公平锁**。线程按顺序请求获得公平锁，然而一个非公平锁允许“**闯入**”：当请求这样的锁时，如果锁的状态变为可用，线程的请求可以在等待线程的队列中向前跳跃，获得该锁。（Semaphore 同样提供了公平和非公平的获取顺序。）非公平的 ReentrantLock 并不是有意鼓励“**闯入**”——倘若遇到**闯入**的发生，它们不会有意避开。在公平锁中，如果锁已经被其他线程占有，新的请求线程会加入到等待队列，或者已经有一些线程在等待锁了；在非公平的锁中，线程只有当锁正在被占用时才会等待⁴。

我们难道不希望所有的锁都是公平的么？毕竟，公平是好的，不公平不好，对么？（去问问你的孩子。）当发生加锁的时候，公平会因为挂起和重新开始线程的代价带来巨大的性能开销。实践中，统计上的公平性保证——承诺一个阻塞的线程**最终**能够获得锁——通常已经够用了，比自由情况下的开销小得多。有一些算法依赖于公平的队列，确保它们的

⁴ 即使对于公平锁而言，可轮询的 tryLock 总会闯入。

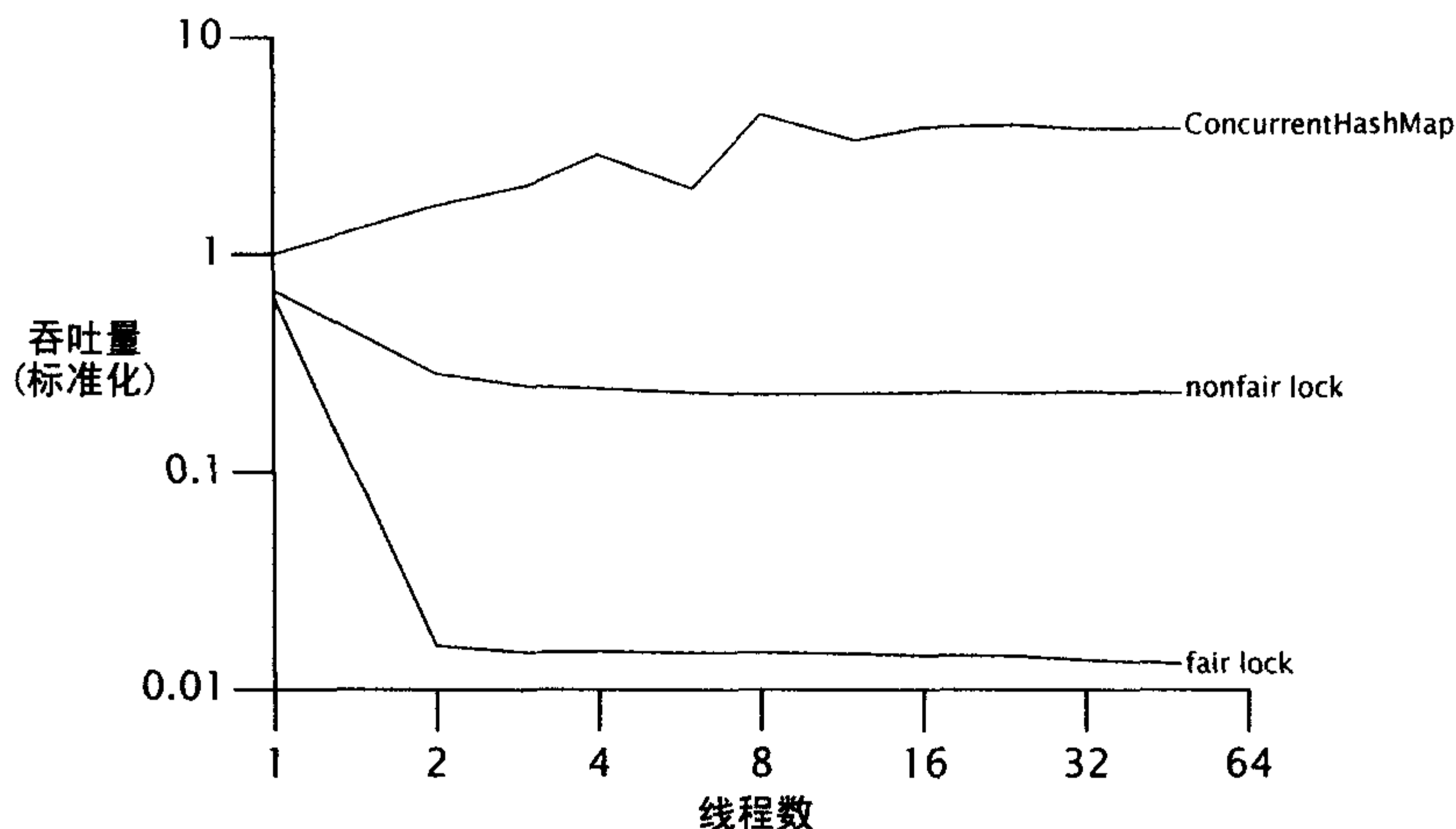


图 13.2 公平锁与非公平锁性能上的对比

正确性，但是有一些例外。在多数情况下，非公平锁性能的优势超过了公平地排队。

图 13.2 展现了另一个 Map 运行的测试，这一次比较由公平锁、非公平锁以及 ReentrantLock 包装的 HashMap，测试运行在 4 路的 Solaris 操作系统上，其结果绘制在比例图中⁵。公平性显示出相当差的性能，与其他两个相比几乎差了两个数量级。如果必要的话，不要为公平性付出代价。

在激烈竞争的情况下，闯入锁比公平锁性能好的原因之一是：挂起的线程重新开始，与它真正开始运行，两者之间会产生严重的延迟。我们假设线程 A 持有一个锁，线程 B 请求该锁。因为此时锁正在使用中，所以 B 会被挂起。当 A 释放锁后，B 重新开始。与此同时，如果 C 请求这个锁，那么 C 得到了很好的机会获得这个锁，使用它，并且甚至可能在 B 被唤醒前就已经释放该锁了。在这样的情况下，各方面都获得了成功：B 并没有比其他任何线程晚得到锁，C 更早地得到了锁，吞吐量得到了改进。

当持有锁的时间相对较长，或者请求锁的平均时间间隔比较长，那么使用公平锁是比较好的。在那些情况下，闯入带来的优势的情况——在线程正在唤醒的过程中，还没有得到锁——不太容易出现。

⁵ ConcurrentHashMap 的曲线在 4 线程和 8 线程数之间发生了摆动。这些变化几乎可以说来自于测量噪音，可能是由于一致的元素哈希码，线程调度，map 大小，垃圾回收或者其他内存系统的影响，也可能由于 OS 在测试用例运行期间，决定运行一些周期性任务回收。现实中，不值得为影响性能的因素而烦恼。我们不应该尝试人工的修改曲线，因为现实世界中性能评估会遇到各种各样的干扰。

正如默认的 `ReentrantLock` 一样，内部锁没有提供确定的公平性保证，但是大多数锁实现统计上的公平性保证，在大多数条件下已经足够好了。Java 语言规范并没有要求 JVM 公平地实现内部锁，JVM 也的确没有这样做。`ReentrantLock` 并没有减少锁的公平性——它只不过使一些存在的部分更显性化了。

13.4 在 synchronized 和 ReentrantLock 之间进行选择

`ReentrantLock` 与内部锁在加锁和内存语义上是相同的，在以下附加特性的语义上也相同，比如定时锁的等待，可中断锁的等待，公平性，以及实现非块结构的锁。`ReentrantLock` 的性能看起来胜过内部锁，Java 6 中时略微胜过，而 Java 5.0 中是大大超越。那么为什么不放弃使用 `synchronized`，鼓励大家都使用新的并发 `ReentrantLock` 呢？事实上的确有一些作者建议如此，把 `synchronized` 作为“遗留”结构。但是这会把好事情变坏。

内部锁相比于显式锁仍然具有很大的优势。这个标识更为人们所熟悉，也更简洁，而且很多现有的程序已经在使用内部锁了——混合这两者会造成混淆，反而并更易发生错误。`ReentrantLock` 绝对是最危险的同步工具；如果你忘记在 `finally` 块中调用 `unlock`，你的代码将很可能看起来能够正常运行，但是已经埋下定时炸弹，并很有可能伤及无辜。在内部锁不能满足需求，需要使用 `ReentrantLock` 的情况下才应该使用。

在内部锁不能够满足使用时，`ReentrantLock` 才被作为更高级的工具。当你需要以下高级特性时，才应该使用：可定时的、可轮询的与可中断的锁获取操作，公平队列，或者非块结构的锁。否则，请使用 `synchronized`。

在 Java 5.0 中，内部锁与 `ReentrantLock` 相比，还具有另外一个优点：线程转储能够显示哪些个调用框架获得了哪些锁，并能够识别发生了死锁的那些线程。JVM 并不知道哪个线程持有 `ReentrantLock`，因此在调试使用 `ReentrantLock` 的线程间存在的问题时，无法从中获得帮助。这个问题在 Java 6 中得到了解决，它提供了一个管理和调试接口，锁可以使用这个接口进行注册，并通过其他管理和调试接口，从线程转储中得到 `ReentrantLock` 的加锁信息。这些用于调试的消息的可用性对于 `synchronized` 是有优势的，即使它们大部分为即时消息；线程转储中的加锁信息解除了很多程序员的恐慌。`ReentrantLock` 的非块结构的特性仍然意味着获取锁不能依赖于特定的栈结构，这一点与内部锁是不同的。

未来的性能改进可能更倾向于 `synchronized` 而不是 `ReentrantLock`。因为 `synchronized` 是内置于 JVM 的，它能够进行优化，比如对线程限制的锁对象的锁省略（lock elision），粗化锁来减小内部锁的同步性（参见 11.3.2）；使用基于类库的锁来实现这些看起来可能性不大。除非你正发布在 Java 5.0 上，对于未来，除非你的平台对 `ReentrantLock` 的可伸缩性有切实的需要，否则就性能的原因选择 `ReentrantLock` 而不是 `synchronized`，这不是正确的决定。

13.5 读-写锁

`ReentrantLock` 实现了标准的互斥锁：一次最多只有一个线程能够持有相同 `ReentrantLock`。但是互斥通常作为保护数据一致性的很强的加锁约束，因此过分地限制了并发性。互斥是保守的加锁策略，避免了“写/写”和“写/读”的重叠，但是同样避开了“读/读”的重叠。在很多情况下，数据结构是“频繁被读取”的——它们是可变的，有的时候会被改变，但多数访问只进行读操作。此时，如果能够放宽，允许多个读者同时访问数据结构就非常好了。只要每个线程保证能够读到最新的数据，并且在读者读取数据的时候没有其他线程修改数据，就不会发生问题。这就是读-写锁允许的情况：一个资源能够被多个读者访问，或者被一个写者访问，两者不能同时进行。

清单 13.6 中所示的 `ReadWriteLock`，暴露了两个 `Lock` 对象，一个用来读，另一个用来写。读取 `ReadWriteLock` 锁守护的数据，你必须首先获得读取的锁，当需要修改 `ReadWriteLock` 守护的数据时，你必须首先获得写入的锁。尽管看起来是两个分离的锁，读取的锁和写入的锁只不过是统一的读-写锁对象的两个视角。

清单 13.6 `ReadWriteLock` 接口

```
public interface ReadWriteLock {  
    Lock readLock();  
    Lock writeLock();  
}
```

读-写锁实现的加锁策略允许多个同时存在的读者，但是只允许一个写者。与 `Lock` 一样，`ReadWriteLock` 允许多种实现，造成了性能、调度保证、获取优先、公平性、以及加锁语义等方面的不尽相同。

读-写锁的设计是用来进行性能改进的，使得特定情况下能够有更好的并发性。在实践中，当多处理器系统中，频繁的访问主要为读取数据结构的时候，读-写锁能够改进性

能；在其他情况下运行的情况比独占的锁要稍差一些，这归因于它更大的复杂性。使用它究竟能否带来改进，最好通过对系统进行剖析来判断；好在 `ReadWriteLock` 使用 `Lock` 作为读、写部分的锁，所以如果剖析的结果发现读-写锁没有能提高性能，把读-写锁替换为独占锁是比较容易的。

读取和写入锁之间的互动可以有很多种实现。`ReadWriteLock` 的一些实现选择如下：

释放优先。当写者释放写入锁，并且读者和写者都排在队列中，应该选择哪个——读者，写者，还是先请求的那个呢？

读者闯入。如果锁由读者获得，但是有写者正在等待，那么新到达的写者应该被授予读取的权力么？还是应该等待？允许读者闯入到写者之前提高了并发性，但是却带来了写者饥饿的风险。

重进入。读取锁和写入锁允许重入吗？

降级。如果线程持有写入的锁，它能够在不释放该锁的情况下获得读取锁么？这可能会造成写者“降级”为一个读取锁，同时不允许其他写者修改这个被守护的资源。

升级。读取锁能够优先于其他的读者和写者升级为一个写入锁么？大多数读-写锁的实现并不支持升级，因为在没有显式的升级操作的情况下，很容易造成死锁。（如果两个读者同时试图升级到同一个写入锁，并都不释放读取锁。）

`ReentrantReadWriteLock` 为两个锁提供了可重进入的加锁语义。与 `ReentrantLock` 相同，`ReentrantReadWriteLock` 能够被构造为非公平（默认）或者是公平的。在公平的锁中，选择权交给等待时间最长的线程；如果锁由读者获得，而一个线程请求写入锁，那么不再允许读者获得读取锁，直到写者被受理，并且已经释放了写入锁。在非公平的锁中，线程允许访问的顺序是不定的。由写者降级为读者是允许的；从读者升级为写者是不允许的（尝试这样的行为会导致死锁）。

与 `ReentrantLock` 相同，`ReentrantReadWriteLock` 的写入锁有一个唯一的所有者，并只能被获得了该锁的线程释放。Java 5.0 中，读取锁的行为更类似于一个 `Semaphore`，只维护活跃的读者数量，而不考虑它们的身份。这个行为在 Java 6 中获得了修改，现在也可以保持追踪哪些线程已经获得了读者锁⁶。

⁶ 改变Java 5.0中该实现的一个原因是：锁的实现不能区别一个线程是首次请求该读取锁，还是重新请求该锁，这有可能造成公平的读-写锁发生死锁。

当锁被持有的时间相对较长，并且大部分操作都不会改变锁守护的资源，那么读-写锁能够改进并发性。清单 13.7 中，ReadWriteMap 使用了 ReentrantReadWriteLock 来包装 Map，使得它能够在多线程间被安全地共享，并仍然能够避免“读-写”或者“写-写”冲突⁷。现实中，ConcurrentHashMap 的性能已经足够好了，所以你可以使用它，而不必使用这个新的解决方案，如果你需要并发的部分只有哈希 Map，但是如果你需要为 LinkedHashMap 这种可替换元素的 Map 提供更多的并发访问，那么这项技术是非常有用的。

清单 13.7 用读写锁包装的 Map

```
public class ReadWriteMap<K,V> {
    private final Map<K,V> map;
    private final ReadWriteLock lock = new ReentrantReadWriteLock();
    private final Lock r = lock.readLock();
    private final Lock w = lock.writeLock();

    public ReadWriteMap(Map<K,V> map) {
        this.map = map;
    }

    public V put(K key, V value) {
        w.lock();
        try {
            return map.put(key, value);
        } finally {
            w.unlock();
        }
    }
    // remove(), putAll(), clear() 也完全类似

    public V get(Object key) {
        r.lock();
        try {
            return map.get(key);
        } finally {
            r.unlock();
        }
    }
    // 对于其他的只读 Map 方法也完全类似
}
```

图 13.3 展示了 ReentrantLock 和 ReentrantReadWriteLock 包装的 ArrayList 之

⁷ ReadWriteMap 并没有实现 Map，因为实现视图的方法，比如 entrySet 和 values，是非常困难的，并且“简单”的方法通常已经足够了。

间吞吐量的比较，该测试运行在 4 路的 Solaris 操作系统上。这里使用的测试程序与 Map 的性能测试基本类似，我们在本书中已经大量使用——每一个操作随机选择一个值并在容器中查找，只有很少的操作会修改这个容器中的内容。

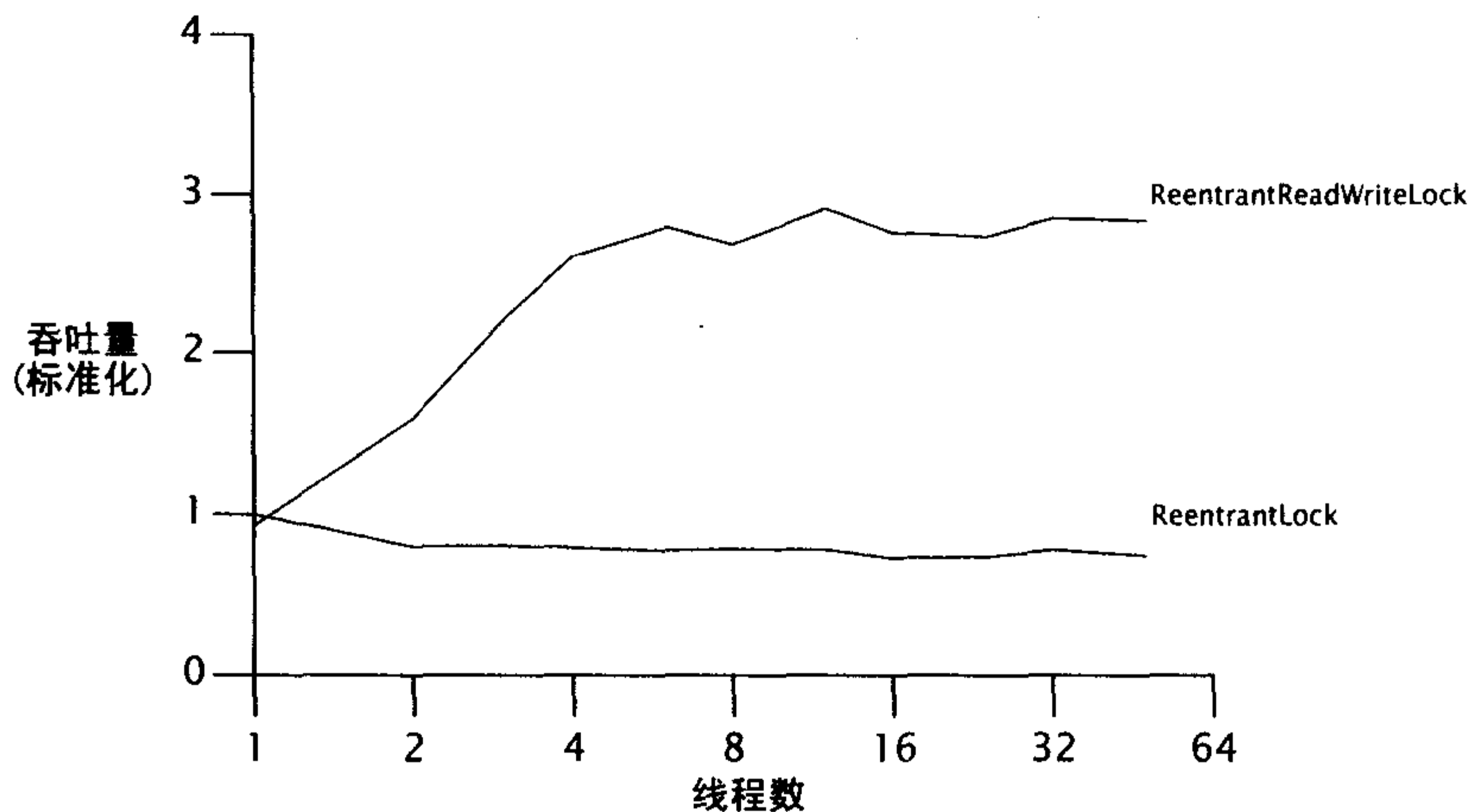


图 13.3 读-写锁的性能

总结

显式的 Lock 与内部锁相比提供了一些扩展的特性，包括处理不可用的锁时更好的灵活性，以及对队列行为更好的控制。但是 ReentrantLock 不能完全替代 synchronized；只有当你需要 synchronized 没能提供的特性时才应该使用。

读-写锁允许多个读者并发访问被守护的对象，当访问多为读取数据结构的时候，它具有改进可伸缩性的潜力。

构建自定义的同步工具

Building Custom Synchronizer

类库中包含了大量**依赖于状态**（state-dependent）的类——这些类拥有**基于状态的先验条件**——`FutureTask`、`Semaphore` 和 `BlockingQueue`。例如，你不能从一个空队列中移除条目，也不能从一个尚未结束的任务中获取结果；在这些操作可以执行前，你必须等到队列进入“非空”状态，任务进入“完成”状态。

创建状态依赖类最简单的方法通常是将其构建于已有的状态依赖库类之上；在 187 页的 `ValueLatch` 中，我们使用 `CountDownLatch` 来提供所需的阻塞行为。但是如果类库没有提供你需要的功能，你也可以使用语言和类库提供的底层机制，包括内部**条件队列**、显式的 `Condition` 对象和 `AbstractQueuedSynchronizer` 框架，构建属于自己的 `Synchronizer`。在这一章里，我们会探究一些实现状态依赖性时要面对的不同选择，以及使用平台提供的状态依赖性机制时需要遵守的不同规则。

14.1 管理状态依赖性

在单线程化的程序中，如果调用一个方法的时候，依赖于状态的先验条件尚未满足（比如“连接池非空”），那么这个先验条件永远无法变为真了。因此，在顺序程序中，如果类的先验条件无法满足，就将它标为失败。但是在并发程序中，基于状态的条件会在其他线程的活动中被改变：一个池在几条指令以前还是空的，现在却可以变为非空，因为另外的线程会归还一个元素。对于并发对象，依赖于状态的方法有时可以在不能满足先验条件的情况下选择失败，不过更好的选择是等待先验条件变为真。

有一种依赖于状态的操作，能够**被阻塞**直到可以继续执行，这与简单地让它们失败相比，更不易出问题，用起来也方便得多。内部条件队列的机制可以让线程一直阻塞，直到对象已经进入某个特定的状态，在该状态下进程可以继续执行，并且在阻塞线程可以进行

进一步执行的时候，对象会将它们唤醒。我们将在第 14.2 节介绍条件队列的细节。不过在展现高效的条件等待机制有何价值之前，我们先了解一下如何运用“轮询与休眠（polling and sleeping）”来（费力不讨好地）解决状态依赖性的问题。

一个可阻塞的状态依赖活动表现为清单 14.1 所示的形式。锁是在操作过程中被释放与重获的，这也让这种加锁的模式略显与众不同。组成先验条件的状态变量必须由对象的锁保护起来，这样它们能在测试先验条件的过程中保持不变。如果先验条件尚未满足，就必须释放锁，让其他线程可以修改对象的状态——否则，先验条件就永远无法变成真了。再一次测试先验条件之前，必须要重新获得锁。

清单 14.1 状态依赖的可阻塞行为的结构

```
void blockingAction() throws InterruptedException {
    acquire lock on object state
    while (precondition does not hold) {
        release lock
        wait until precondition might hold
        optionally fail if interrupted or timeout expires
        reacquire lock
    }
    perform action
    reacquire lock
}
```

生产者-消费者的设计经常会使用 `ArrayBlockingQueue` 这种有限缓存。一个有限缓存提供的 `put` 和 `take` 操作，每一个都有先验条件：你不能从空缓存中获取元素，也不能把元素置入已满的缓存中。如果依赖于状态的操作在处理先验条件时失败，可以抛出异常或者返回错误状态（把问题留给调用者），也可以保持阻塞直到对象转入正确的状态。

我们接下来会实现几个有限缓存，其间会用到不同的方法处理先验条件失败。每种实现都扩展了清单 14.2 中的 `BaseBoundedBuffer`。`BaseBoundedBuffer` 实现了一个经典的基于队列的循环缓存，缓存的状态变量（`buf`、`head`、`tail` 和 `count`）是由缓存的内部锁保护的。它还提供了同步的 `doPut` 和 `doTake` 方法，用于在子类中实现 `put` 和 `take` 操作；底层的状态对于子类是隐藏的。

14.1.1 示例：将先验条件失败传给调用者

清单 14.3 的 `GrumpyBoundedBuffer` 是第一个直接的、简单的有限缓存实现。由于 `put` 和 `take` 方法都在访问缓存时采用“检查再运行（check-then-act）”逻辑，所以它们都是同步的，以确保独占访问缓存状态。

尽管这种方法实现起来足够简单，但用起来却令人厌烦。异常应该用于异常条件中[[EJ Item 39](#)]。与其把“缓存已满”称作是有限缓存的一个异常条件，还不如把“红灯”称作

清单 14.2 有限缓存不同实现的基类

```
@ThreadSafe
public abstract class BaseBoundedBuffer<V> {
    @GuardedBy("this") private final V[] buf;
    @GuardedBy("this") private int tail;
    @GuardedBy("this") private int head;
    @GuardedBy("this") private int count;

    protected BaseBoundedBuffer(int capacity) {
        this.buf = (V[]) new Object[capacity];
    }

    protected synchronized final void doPut(V v) {
        buf[tail] = v;
        if (++tail == buf.length)
            tail = 0;
        ++count;
    }

    protected synchronized final V doTake() {
        V v = buf[head];
        buf[head] = null;
        if (++head == buf.length)
            head = 0;
        --count;
        return v;
    }

    public synchronized final boolean isFull() {
        return count == buf.length;
    }

    public synchronized final boolean isEmpty() {
        return count == 0;
    }
}
```

清单 14.3 如果有限缓存不满足先验条件，会停滞不前

```
@ThreadSafe
public class GrumpyBoundedBuffer<V> extends BaseBoundedBuffer<V> {
    public GrumpyBoundedBuffer(int size) { super(size); }

    public synchronized void put(V v) throws BufferFullException {
        if (isFull())
            throw new BufferFullException();
        doPut(v);
    }

    public synchronized V take() throws BufferEmptyException {
        if (isEmpty())
            throw new BufferEmptyException();
        return doTake();
    }
}
```



是交通灯的异常条件。实现缓存时所作的简化（强迫调用者管理状态依赖性），无法抵消实际使用它时带来的麻烦，因为现在调用者必须时刻准备捕获异常，并且可能对每次缓存操作都要重试¹。一个结构正确的 take 调用如清单 14.4 所示——不够漂亮，尤其是当程序中很多地方都用到 put 和 take 操作时。

清单 14.4 调用 GrumpyBoundedBuffer 的客户端逻辑

```
while (true) {
    try {
        V item = buffer.take();
        // 这里使用 item
        break;
    } catch (BufferEmptyException e) {
        Thread.sleep(SLEEP_GRANULARITY);
    }
}
```

这种方法的一种变体是当缓存处在错误状态时返回一个错误值。上面的例子中，抛出一个异常并不只是意味着“抱歉，请再试一次”，这种方法只是在滥用异常上有一点点的

¹ 将状态依赖性推卸给调用者，还会使得做到像维持 FIFO 顺序这种事情变得希望渺茫；由于强迫调用者重试，你失去了“谁先到达”的信息。

改进，但是并没有解决根本问题：调用者必须自行处理先验条件失败²。

清单 14.4 的客户端代码不是实现重试逻辑的唯一方式。调用者可以不休眠（sleep），而直接重试 take 操作——一种被称作**忙等待**或**自旋等待**的方法。如果在相当长的一段时间内，缓存的状态都不会改变，那么使用这种方法就会消耗相当多的 CPU 时间。另一方面，调用者也可以决定休眠（sleep），以避免消耗过多的 CPU 时间，但是如果缓存的状态在调用 sleep 不久后，很快就发生了变化，那么它很容易会“睡过头”。所以，客户端代码身处于自旋产生的低 CPU 使用率和休眠产生的弱响应性之间的两难境地。每次迭代中，在忙等待与休眠之间的一种折中的选择是调用 Thread.yield，这给调度器一个提示：我现在可以让出一定的时间让另外的线程运行。假如你正在等待另一个线程的工作，那么如果你让出（yield）处理器，而不是把你的 CPU 调度配额全部消耗掉，会让有些工作可以更快地被处理。

14.1.2 示例：利用“轮询加休眠”实现拙劣的阻塞

清单 14.5 的 SleepyBoundedBuffer 尝试通过在 put 和 take 操作内部封装相同的“轮询和休眠”重试机制，为每次调用实现了重试逻辑，从而分担调用者的麻烦。如果缓存是空的，take 将休眠，直到另一个线程在缓存中置入了一些数据；如果缓存是满的，put 将休眠，直到另一个线程移除了一些数据，在缓存中腾出地方来。这个方法封装了对先验条件的管理，简化了缓存的使用——这向正确的方向上迈出了一步。

SleepyBoundedBuffer 的实现远比前面所做的尝试更复杂³。缓存代码必须在持有缓存的锁时才能测试相应的状态条件，因为表示状态条件的变量是由缓存的锁保护的。如果测试失败，执行线程会暂时休眠，不过首先会释放锁，这样其他线程才能够访问缓存⁴。一旦线程被唤醒，它会重新请求锁并再次尝试，在操作可以处理前，它可以休眠也可以检查状态条件。

从调用者的角度看，它的运行表现很好——如果操作可以立即执行，就去做，否则就阻塞——调用者不必处理失败和重试。选择休眠的时间间隔，是在响应性与 CPU 使用率之间作出的权衡；休眠的间隔越小，响应性越好，但是 CPU 的消耗也越高。图 14.1 演示了休眠间隔是如何影响响应性的：缓存空间变为可用的时刻与线程被唤醒并再次检查的时刻之间可能有延迟。

² Queue 提供了上述两种选择——poll 在队列为空时返回 null，remove 则抛出异常——但是不要打算把 Queue 用于生产者-消费者的设计。BlockingQueue 在队列处于可被处理的正确状态之前，会阻塞操作。因此当生产者和消费者并发执行时，它才是更好的选择。

³ 我们把白雪公主里面其他 5 个小矮人的实现留给了你，尤其是“SneezyBoundedBuffer”。（白雪公主身边的 7 个小矮人分别叫 Sleepy, Grumpy, Sneezy... 其中 Sleepy 和 Grumpy 对应的 BoundedBuffer 已经实现。译注）

⁴ 通常，让线程在休眠或者被阻塞的同时还持有锁，这是个坏主意，这种情况甚至会更糟，因为如果锁不能释放，期望的条件（缓存是满/是空）永远无法为真。

清单 14.5 有限缓存使用了拙劣的阻塞

```
@ThreadSafe
public class SleepyBoundedBuffer<V> extends BaseBoundedBuffer<V> {
    public SleepyBoundedBuffer(int size) { super(size); }

    public void put(V v) throws InterruptedException {
        while (true) {
            synchronized (this) {
                if (!isFull()) {
                    doPut(v);
                    return;
                }
            }
            Thread.sleep(SLEEP_GRANULARITY);
        }
    }

    public V take() throws InterruptedException {
        while (true) {
            synchronized (this) {
                if (!isEmpty())
                    return doTake();
            }
            Thread.sleep(SLEEP_GRANULARITY);
        }
    }
}
```



`SleepyBoundedBuffer` 也给调用者提出了新的要求：处理 `InterruptedException`。当一个方法因为等待某个条件变成真而阻塞时，为它提供一个取消机制是有意义的(参见第7章)。就像大多数行为良好的阻塞库的方法一样，`SleepyBoundedBuffer` 通过中断支持取消，如果被中断，它会提前返回，并抛出 `InterruptedException`。

上述这些将轮询与休眠组合成一个阻塞操作的尝试都不能令人非常满意。如果存在某种挂起线程的方法，能够保证当某个条件成为真时，线程可以及时地苏醒过来，这样就太好了。而这恰恰就是**条件队列**（condition queue）所做的工作。

14.1.3 让条件队列来解决这一切

条件队列就如同烤面包机上的“面包已好”的铃声。如果你正在听着它，当面包烤好后你可以立刻注意到，并且放下手头的事情（也可以先不理睬它，或许你想先看完报纸），

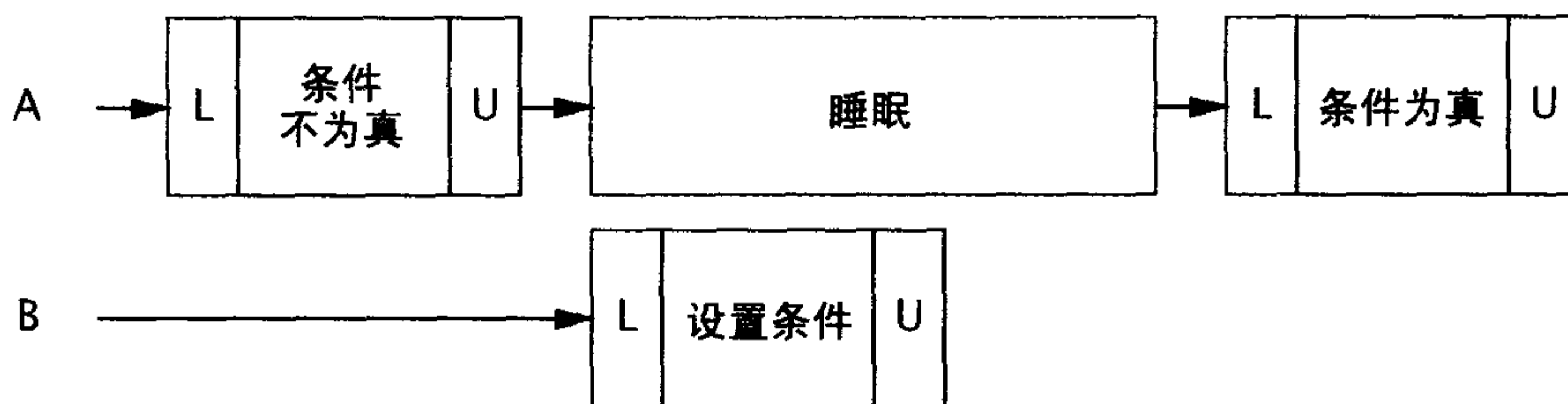


图 14.1 线程刚进入休眠，(唤醒)条件就成为真，这会引起线程“睡过头”

开始品尝面包。如果你没有听见它（可能你出去拿报纸了），你会错过通知信息，但是回到厨房后还是可以看到烤面包机的状态，如果已经烤完，就取出面包；如果未烤完，就再次监听提示铃声。

条件队列可以让一组线程——称作**等待集**——以某种方式等待相关条件变成真，它也因此得名。不同于传统的队列，它们的元素是数据项；条件队列的元素是等待相关条件的线程。

就像每个 Java 对象都能当作锁一样，每个对象也能当作条件队列，Object 中的 wait、notify、notifyAll 方法构成了内部条件队列的 API。一个对象的内部锁与它的内部条件队列是相关的：为了能够调用对象 X 中的任一个条件队列方法，你必须持有对象 X 的锁。这是因为“等待基于状态的条件”机制必须和“维护状态一致性”机制紧密地绑定在一起：除非你能检查状态，否则你不能等待条件；同时，除非你能改变状态，否则你不能从条件等待（队列）中释放其他的线程。

Object.wait 会自动释放锁，并请求 OS（操作系统）挂起当前线程，让其他线程获得该锁进而修改对象的状态。当它被唤醒时，它会在返回前重新获得锁。直观上看，调用 wait 意味着“我要去休息了，但是发生了需要关注的事情后叫醒我”，调用通知（notification）方法意味着“需要关注的事情发生了”。

清单 14.6 的 BoundedBuffer 使用 wait 和 notifyAll 实现了有限缓存。这个不仅比“休眠”版本更高效（如果缓存状态未变化，就不会频繁地“唤醒”），而且响应性更佳（当所关注的事情发生后及时唤醒）。这是一项很大的改进，不过要注意：相比于“休眠”版本，条件队列的引入并没有改变原有语意。它不过是在多方面进行优化后的结果：CPU 效率、上下文切换开销和响应性。你不能利用“轮询和休眠”完成的任何事情，用条件队列也无法完成⁵，但是它使得表达和管理状态的依赖性变得更加简单和高效。

⁵ 这并不完全正确；一个公平的条件队列可以确保线程从等待集中释放的相对顺序。像内部锁这种内部条件队列并不提供公平队列；不过显式的 Condition 可以提供公平/非公平队列的选择。

清单 14.6 有限缓存使用条件队列

```
@ThreadSafe
public class BoundedBuffer<V> extends BaseBoundedBuffer<V> {
    // 条件谓词: not-full (!isFull())
    // 条件谓词: not-empty (!isEmpty())

    public BoundedBuffer(int size) { super(size); }

    // 阻塞, 直到: not-full
    public synchronized void put(V v) throws InterruptedException {
        while (isFull())
            wait();
        doPut(v);
        notifyAll();
    }

    // 阻塞, 直到: not-empty
    public synchronized V take() throws InterruptedException {
        while (isEmpty())
            wait();
        V v = doTake();
        notifyAll();
        return v;
    }
}
```

BoundedBuffer 目前已经足够好了——简单易用, 而且把状态独立性管理得非常清晰⁶。用于生产环境的版本还应该包括限时的 put 和 take 版本, 这样如果阻塞操作不能在预计的时间内完成, 可以超时。限时版的 put 和 take 可一个通过 Object.wait 的限时版来实现, 这很简单。

14.2 使用条件队列

条件队列让构建有效且可响应的状态依赖类变得容易, 但是把它们用错也很容易; 关于如何正确使用它们, 存在着很多规则, 编译器和平台却并没有强制要求它们。(这是要求你尽量将程序构建于像 `LinkedBlockingQueue`、`Latch`、`Semaphore` 和 `FutureTask` 这些类之上的原因之一; 如果你能避免使用条件队列, 可以避免很多麻烦。)

⁶ 14.3 节的 `ConditionBoundedBuffer` 甚至做得更好: 因为它能使用单一通知而不是 `notifyAll`, 因此更高效。

14.2.1 条件谓词

正确使用条件队列的关键在于识别出对象可以等待的**条件谓词**。由于在 API 中没有关于条件谓词的具体实例，同时在语言规范或者 JVM 实现中也没有任何信息可以确保它的正确使用，因此这个**条件谓词**会引起大量的围绕 `wait` 和 `notify` 的混淆。事实上，在语言规范或者 Javadoc 中根本没有直接提到它。但是如果缺少了条件谓词，条件等待机制将无法工作。

条件谓词是先验条件的第一站，它在一个操作与状态之间建立起依赖关系。在有限缓存中，只有缓存不为空时 `take` 才能执行，否则它必须等待。就 `take` 而言，它的条件谓词是“缓存不空”，`take` 执行前必须先测试。类似地，`put` 的条件谓词是“缓存不满”。条件谓词是由类的状态变量构成的表达式；`BaseBoundedBuffer` 是通过比较 `count` 与零，测试是否“缓存不空”；并通过比较 `count` 与缓存大小，测试是否“缓存不满”。

将条件谓词和与之关联的条件队列，以及在条件队列中等待的操作，都写入文档。

在涉及了加锁、`wait` 方法和条件谓词的条件等待中，存在着一种非常重要的三元关系。条件谓词涉及状态变量，而状态变量是由锁保护的，所以在测试条件谓词之前，我们必须先持有锁。锁对象与条件队列对象（`wait` 和 `notify` 方法调用的对象）必须也是同一个对象。

在 `BoundedBuffer` 中，缓存的状态是由缓存的锁保护的，缓存对象则用作条件队列。`take` 方法会请求缓存锁，然后测试条件谓词（缓存是非空的）。如果缓存是非空的，它会移除第一个元素。之所以能这样做，是因为 `take` 此时仍然持有用于保护缓存状态的锁。

如果条件谓词不为真（缓存为空），那么 `take` 必须等待，直到另一个线程在缓存中置入一个对象为止。为此 `take` 需要调用缓存内部条件队列的 `wait` 方法，而这需要先持有条件队列对象的锁。仔细的设计就应该会做到这一点，`take` 已经持有了锁——正是测试条件谓词（并且如果条件谓词为真，在同一原子操作中修改缓存状态）时会用到的那个锁。`wait` 会释放锁，并阻塞当前线程，然后等待，直到特定时间的超时过后，线程被中断或者被通知唤醒。线程被唤醒后，`wait` 会在返回运行前重新请求锁。一个从 `wait` 方法中唤醒的线程，在重新请求锁的过程中没有任何特殊的优先级；它像任何其他尝试进入 `synchronized` 块的线程一样去争夺锁。

每次调用 `wait` 都会隐式地与特定的条件谓词相关联。当调用特定条件谓词的 `wait` 时，调用者必须已经持有了与条件队列相关的锁，这个锁必须同时还保护着组成条件谓词的状态变量。

14.2.2 过早地唤醒

锁、条件谓词和条件队列之间存在的三元关系看似并不错综复杂，但 `wait` 的返回并不一定意味着线程正在等待的条件谓词已经变成真了。

一个单独的**内部条件队列可以与多个条件谓词共同使用**。当有人调用 `notifyAll`，从而唤醒了你的线程时，并不意味着你正在等待条件谓词现在变成真了。（这就像让你的烤面包机和咖啡机共用一个振铃；当它响起后，你还必须查看是哪个设备发出的这个信号⁷。）另外，`wait` 甚至可以“假装”返回——不作为对任何线程调用 `notify` 的响应⁸。

当控制流重新进入调用 `wait` 的代码时，它会重新请求与条件队列相关联的锁。现在条件谓词就一定是真了么？不一定。它可能在通知线程（`notifying thread`）调用 `notifyAll` 的时刻已经变成真的，但是会在你重新请求锁的时刻又再次变为假。在你的线程被唤醒到 `wait` 重新请求锁的这段时间内，其他线程可能已经请求到锁，并改变了对象的状态。更有甚者，自从你调用了 `wait` 之后，条件谓词可能根本就没有变成过真。你无法知道另一个线程为什么调用 `notify` 或 `notifyAll`；也许是因为与同一条件队列相关的另一个条件谓词变成了真。“每条件队列-多条件谓词”是相当常见的——`BoundedBuffer` 就为“非满”与“非空”两个谓词使用了相同的条件队列⁹。

基于所有这些原因，当你从 `wait` 中唤醒后，都必须再次测试条件谓词，如果条件谓词尚未成真，就继续等待（或失败）。在条件谓词没有成真的情况下，你可以一次次地唤醒等待线程，因此你必须永远在循环内部调用 `wait`，在每次迭代中测试条件谓词。清单 14.7 所示的是条件等待的规范式。

⁷ 这个场景是 Tim（Tim Peierls，本书的合著者之一。译注）家餐厅的真实写照；当你听到一个响声时，其实会有很多设备的可能在蜂鸣，你必须检查烤面包机、微波炉、咖啡机和其他的设备，来判断信号的起源。

⁸ 就让我们还继续用“早餐”来作类比吧。这就好比烤面包机的线路连接有问题，导致面包尚未烤好，铃声就自己响起来了。

⁹ 对于线程来说，事实上有可能同时等待“非满”与“非空”！当生产者/消费者的数目超过了缓存的容量时，就会发生这种事情。

清单 14.7 状态依赖方法的规范式

```
void stateDependentMethod() throws InterruptedException {  
    // 条件谓词必须被锁守护  
    synchronized(lock) {  
        while (!conditionPredicate())  
            lock.wait();  
  
        // 现在，对象处于期望的状态中  
    }  
}
```

当使用条件等待时 (Object.wait 或者 Condition.await):

- 永远设置一个条件谓词——一些对象状态的测试，线程执行前必须满足它；
- 永远在调用 wait 前测试条件谓词，并且从 wait 中返回后再次测试；
- 永远在循环中调用 wait；
- 确保构成条件谓词的状态变量被锁保护，而这个锁正是与条件队列相关联的；
- 当调用 wait、notify 或者 notifyAll 时，要持有与条件队列相关联的锁；并且，
- 在检查条件谓词之后、开始执行被保护的逻辑之前，不要释放锁。

14.2.3 丢失的信号

我们在第 10 章曾经讨论过活跃度失败，比如死锁和活锁。另一种形式的活跃度失败是**丢失的信号** (missed signal)。当一个线程等待的特定条件已经为真，但是进入等待前检查条件谓词却返回了假，我们称这样就出现了一个丢失的信号。现在线程正在等待一个已经发过的事件给它通知。这就好像你启动了烤面包机，随后出屋去拿报纸，当你还在屋外时烤面包机的铃声响了，随后你坐在厨房的桌子前等着烤面包机的铃声。你可能会等上很长时间——潜在地可能永远等着¹⁰。通知不像你涂在面包上的果酱，它没有那么“粘”——如果线程 A 通知了一个条件队列，线程 B 随后在同一个条件队列上等待，那么线程 B **不会**立即被唤醒——需要另一个通知再唤醒它。违背了诸如上述清单中的警示而产生的编码错误，比如未能在调用 wait 之前先检测条件谓词，就会导致信号的丢失。如果你按照清单 14.7 那样来架构你的条件等待，你就不会再遇到信号丢失的问题。

¹⁰ 为了不必等待，其他人也会使用同一台烤面包机，但是这会让情况变得更糟；当铃声响起后，还要对面包的归属作一番争执。

14.2.4 通知

到目前为止，我们已经讨论过关于条件等待所发生的事情的前一半：等待。另一半是通知。在有限缓存中，如果缓存为空，调用 `take` 将会阻塞。在缓存变为非空时，为了能够让 `take` **解除阻塞**，我们必须确保**每一条**能够让缓存变为非空的代码路径都执行一个通知。在 `BoundedBuffer` 中，这种地方只有一处——`put` 之后。所以 `put` 在成功地向缓存中加入一个元素后，会调用 `notifyAll`。类似地，`take` 在移除一个元素后调用 `notifyAll`，向任何正在等待“不为满”条件的线程发出通知，缓存已经不为满了。

无论何时，当你在等待一个条件，一定要确保有人会在条件谓词变为真时通知你。

在条件队列 API 中有两个通知方法——`notify` 和 `notifyAll`。无论调用哪一个，你都必须持有与条件队列对象相关联的锁。调用 `notify` 的结果是：JVM 会从在这个条件队列中等待的众多线程中挑选出一个，并把它唤醒；而调用 `notifyAll` 会唤醒**所有**正在这个条件队列中等待的线程。由于你调用 `notify` 和 `notifyAll` 时必须持有条件队列对象的锁，这导致等待线程此时不能重新获得锁，无法从 `wait` 返回，因此该通知线程应该尽快释放锁，以确保等待线程尽可能快地解除阻塞。

由于会有多个线程因为不同的原因在同一个条件队列中等待，因此不用 `notifyAll` 而使用 `notify` 是危险的。这主要是因为单一的通知容易导致同类的线程丢失全部信号。

`BoundedBuffer` 提供了一个很好的示范，说明为什么 `notifyAll` 在大多数情况下都是优于 `notify` 的选择。这里条件队列用于两个不同的条件谓词：“非空”和“非满”。假设线程 *A* 因为谓词 P_A 而在条件队列中等待，同时线程 *B* 因为谓词 P_B 也在同一个条件队列中等待。现在，假设 P_B 变成真，线程 *C* 执行一个单一的 `notify`：JVM 将从它所拥有的众多线程中选择一个并唤醒。如果 *A* 被选中，它随后被唤醒，看到 P_A 尚未变成真，转而继续等待。其间，本应该可以执行的 *B* 却没有被唤醒。这不是严格意义上的“丢失信号”——它更像一个“被劫持的 (hijacked)”信号——不过问题是一样的：线程正在等待一个已经（或者本应该）发生过的信号。

只有同时满足下述条件后，才能用单一的 `notify` 取代 `notifyAll`：

相同的等待者。只有一个条件谓词与条件队列相关，每个线程从 `wait` 返回后执行相同的逻辑；并且，

一进一出。一个对条件变量的通知，至多只激活一个线程执行。

`BoundedBuffer` 满足“一进一出”条件，但是不满足“相同的等待者”条件，因为等待线程既可能在等待“非满”，也可能在等待“非空”的条件。像是 96 页中的 `TestHarness` 用到的“开始阀门”闭锁（`latch`）就不满足“一进一出”条件，因为一个单一的事件会释放一组线程，打开“开始阀门”后会有多个线程执行。

大多数类都不满足这些条件，因此普遍认可的做法是优先使用 `notifyAll`，而不是单一的 `notify`。尽管使用 `notifyAll` 而非 `notify` 可能有些低效，但是这样做更容易确保你的类的行为是正确的。

出于更理性的理由，这个“普遍认可的做法”让有些人感到不适。如果只有一个线程可以执行，那么使用 `notifyAll` 确实效率很低——有时微不足道，有时令人咋舌。如果有 10 个线程在条件队列中等待，调用 `notifyAll` 会唤醒每一个线程，让它们去竞争锁；然后它们中的大多数或者全部又回到休眠状态。这意味着每一个激活单一线程（或多个线程）执行的事件，都会带来大量上下文转换，和大量竞争锁的请求（最坏的情况是，当只有 n 个线程满足条件时，使用 `notifyAll` 会导致唤醒 $O(n^2)$ 数量的线程）。这是另一种¹¹“基于性能的考虑选择一套方案，而基于安全性的考虑支持另一套方案”的情况。

`BoundedBuffer` 中的 `put` 和 `take` 完成的通知是很保守的：每次向缓存置入对象或从中移出对象的时候，执行一次通知。我们可以对其进行优化：首先，观察只有当缓存从空转为非空，或者从满转为非满时，才需要从等待（队列）中释放一个线程；并且，只有当 `put` 或 `take` 影响到这些状态转换的某一种时，才发出通知。这叫做“**依据条件通知**（`conditional notification`）”。尽管“依据条件通知”可以提升性能，但它毕竟只是一种小技巧（而且还让子类的实现变得复杂），应该谨慎使用。清单 14.8 示范了在 `BoundedBuffer.put` 中如何使用“依据条件通知”。

单一的通知和“依据条件通知”都是优化行为。通常，进行优化时应该遵循“先让它跑起来，再让它快起来——**如果**它还没有足够快”的原则；错误地进行优化很容易给程序带来无法预料的活跃度失败。

¹¹ 译注：最坏的情况是指，条件 1 唤醒到第 n 个线程后才找到匹配的线程，条件 2 唤醒到第 $n-1$ 个线程后才找到匹配的线程，...，条件 n 唤醒最后一个线程，因此共需要唤醒 $n+(n-1)+\dots+2+1=(n^2+n)/2$ 次线程，其复杂度为 $O(n^2)$ 。

清单 14.8 在 BoundedBuffer.put 中使用“依据条件通知”

```
public synchronized void put(V v) throws InterruptedException {
    while (isFull())
        wait();
    boolean wasEmpty = isEmpty();
    doPut(v);
    if (wasEmpty)
        notifyAll();
}
```

14.2.5 示例：阀门类

96 页 TestHarness 中的“开始阀门闭锁（starting gate latch）”的构建，是通过将计数器初始化为 1，创建了一个**二元闭锁**：它只有两种状态，初始状态和终止状态。闭锁会阻止线程通过开始阀门，直到阀门被打开，此时所有的线程都可以通过。虽然闭锁机制通常都能够准确地满足我们的需要，但是在闭锁的行为下构建的“阀门”一旦被打开，就不能再重新关闭，有时这会成为一个缺陷。

使用条件等待开发一个可重关闭的 ThreadGate 很容易，正如清单 14.9 所示，ThreadGate 允许阀门打开与关闭。它提供的 await 方法会阻塞，直到阀门打开。open 方法之所以使用 notifyAll，是因为对于使用单一通知的“一出一进”检查会在这个类的语义中失败。

在 await 中使用条件谓词比简单地检查一下 isOpen 要复杂得多。不过这是必需的，因为如果有 N 个线程在阀门打开时等待它，它们都应该被允许执行。但是，如果阀门接连又快速地关闭了，而且 await 只检查 isOpen，那么所有线程可能无法被释放：在所有线程收到通知，重新请求锁，并退出 wait 的时刻，阀门可能已经再次关闭了。所以 ThreadGate 使用略微复杂一些的条件谓词：每次关闭阀门时，会递增 generation 计数器，如果阀门现在是打开的，或者如果线程到达阀门后，阀门就已经打开，那么线程就可以通过 await 了。

因为 ThreadGate 只支持等待阀门的打开，所以它只在 open 中执行通知；为了同时支持“等待打开”与“等待关闭”两个操作，ThreadGate 必须在 open 和 close 中都进行通知。这正说明了为什么维护依赖于状态的类非常脆弱——增加新的状态依赖操作可能需要修改很多代码路径，在这些路径上会修改对象状态，保证可以正确地执行通知。

14.2.6 子类的安全问题

使用依据条件的或者单一的通知会引入一些约束，导致子类化变得更加复杂 [CPJ 3.3.3.3]。如果你的基类在子类化时，会被某种方式破坏单一或条件通知的某个要求，这样

清单 14.9 使用 wait 和 notifyAll 实现可重关闭的阀门

```
@ThreadSafe
public class ThreadGate {
    // 条件谓词: opened-since(n) (isOpen || generation>n)
    @GuardedBy("this") private boolean isOpen;
    @GuardedBy("this") private int generation;

    public synchronized void close() {
        isOpen = false;
    }

    public synchronized void open() {
        ++generation;
        isOpen = true;
        notifyAll();
    }

    // 阻塞, 直到: opened-since(generation on entry)
    public synchronized void await() throws InterruptedException {
        int arrivalGeneration = generation;
        while (!isOpen && arrivalGeneration == generation)
            wait();
    }
}
```

你就必须重新构建基类, 以使子类可以代表基类发出适当的通知。

一个依赖于状态的类, 要么完全将它的等待和通知协议暴露 (并文档化) 给子类, 要么完全阻止子类参与其中。 (这是对“要么专门为继承而设计, 并给出文档说明, 要么禁止继承”原则的扩展 [EJ Item 15]。) 为继承设计一个依赖于状态的类, 至少需要将条件队列和锁暴露出来, 并且将条件谓词和同步策略都写入文档; 它可能还需要暴露底层的状态变量。 (最糟糕的事情是: 一个状态依赖的类可以把它的状态暴露给子类, 但是**没有**将他们的等待与通知的协议写入文档; 这就好比一个类暴露出它的状态变量, 但是没有把它的不变约束写入文档一样。)

还有另外一种选择就是直接禁止子类化, 可以通过把类声明为 final 类型的, 或者通过对子类隐藏条件队列、锁和状态变量来完成。否则, 如果子类做的一些事情破坏了基类运用 notify 的方式, 基类需要有能力和修复。考虑如果有一个无限的阻塞栈, 它的 pop 操作在栈为空的时候被阻塞, 但是它的 push 永远都可以执行。这符合使用单一通知的条件。如果这个类使用的正是单一通知, 而且在一个子类中添加了一个阻塞的“弹出两个连续元

素”方法，这样就有两种等待者了：等待弹出一个元素的和等待弹出两个元素的。但是如果基类暴露出条件队列，并且已把使用它的协议写入文档中，子类就会在覆写 `push` 方法时执行 `notifyAll`，以重新确保安全性。

14.2.7 封装条件队列

通常，最好可以把条件队列封装起来，这样在使用它的类层次结构之外，是不能访问它的。否则，调用者可能会禁不住诱惑，认为他们理解了关于等待于通知的协议，然后以一种与你的设计不相符行为使用它们。（不可能强迫统一的等待者请求单一的通知，除非条件队列对象对于你无法控制的代码是不可访问的；如果外部代码错误地在你的条件队列上等待，这会破坏你的通知协议，引起一个“被劫持的”信号。）

不幸的是，这条建议——使用封装的对象作为条件队列——并没有被一个最常见的用于线程安全类设计模式所遵循：使用对象的内部锁来保护对象自身的状态。`BoundedBuffer` 示范了这个常见的模式，即缓存对象自身既是锁，又是条件队列。但是，`BoundedBuffer` 可以很容易地被重新构建，并使用一个私有的锁对象和条件队列；唯一的不同在于，它将不再支持任何形式的客户端加锁。

14.2.8 入口协议和出口协议

Wellings (Wellings, 2004) 以“入口协议和出口协议 (entry and exit protocols)”的形式刻画了 `wait` 和 `notify` 的正确使用方法。对于每个依赖于状态的操作，以及每个修改了其他状态的操作（对于每一个修改状态的操作，并且其他操作对该状态有状态依赖），你都应该为其定义并文档化一个入口协议和出口协议。入口协议就是操作的条件谓词；出口协议涉及到要检查任何被操作改变的状态变量，确认它们是否引起其他一些条件谓词变为真，如果是，通知相关的条件队列。

`AbstractQueuedSynchronizer` 采用了出口协议的概念，位于 `java.util.concurrent` 包下的大部分状态依赖类都构建于它之上（参见 14.4 节）。它没有让 `Synchronizer` 类自己去执行通知，而是要求同步方法返回一个值，让这个值说明它的动作是否可能已经阻塞了一个或多个等待线程。这种显式 API 的要求，可以避免发生在某些状态转换的过程中“忘记”执行通知。

14.3 显式的 Condition 对象

正如我们在第 13 章看到的，在某些情况下，当内部锁非常不灵活时，显式锁就可以派上用场。正如 `Lock` 是广义的内部锁，`Condition`（参见清单 14.10）也是广义的内部条件队列。

内部条件队列有一些缺陷。每个内部锁只能有一个与之相关联的条件队列，这意味着在像 `BoundedBuffer` 这种类中，多个线程可能为了不同的条件谓词在同一个条件队列中等待，而且大多数常见的锁模式都会暴露条件队列对象。这些因素都导致不可能为了使用 `notifyAll`，而强迫等待线程统一。如果你想编写一个含有多个条件谓词的并发对象，或者你想获得比条件队列的可见性之外更多的控制权，那么显式的 `Lock` 和 `Condition` 的实现类提供了一个比内部锁和条件队列更加灵活的选择。

一个 `Condition` 和一个单独的 `Lock` 相关联，就像条件队列和单独的内部锁相关联一样；调用与 `Condition` 相关联的 `Lock` 的 `Lock.newCondition` 方法，可以创建一个 `Condition`。如同 `Lock` 提供了比内部加锁要丰富得多的特征集一样，`Condition` 也提供了比内部条件队列要丰富得多的特征集：每个锁可以有多个等待集、可中断/不可中断的条件等待、基于时限的等待以及公平/非公平队列之间的选择。

清单 14.10 Condition 接口

```
public interface Condition {
    void await() throws InterruptedException;
    boolean await(long time, TimeUnit unit)
        throws InterruptedException;
    long awaitNanos(long nanosTimeout) throws InterruptedException;
    void awaitUninterruptibly();
    boolean awaitUntil(Date deadline) throws InterruptedException;

    void signal();
    void signalAll();
}
```

不同于内部条件队列，你可以让每个 `Lock` 都有任意数量的 `Condition` 对象。`Condition` 对象继承了与之相关的锁的公平性特性；如果是公平的锁，线程会依照 FIFO 的顺序从 `Condition.await` 中被释放。

危险警告：`wait`、`notify` 和 `notifyAll` 在 `Condition` 对象中的对等体是 `await`、`signal` 和 `signalAll`。但是，`Condition` 继承于 `Object`，这意味着它也有 `wait` 和 `notify` 方法。一定要确保使用了正确的版本——`await` 和 `signal`！

清单 14.11 所示的是有限缓存的另一种实现，这次使用了两个 `Condition`，`notFull` 和 `notEmpty`，明确地表示“非满”与“非空”两个条件谓词。当缓存为空时，`take` 阻塞，它等待 `notEmpty`；`put` 向 `notEmpty` 发送信号，可以解除任何 `take` 所阻塞的线程。

`ConditionBoundedBuffer` 的行为和 `BoundedBuffer` 相同，但是它使用条件队列的方式，具有更好的可读性——分析使用多个 `Condition` 的类，要比分析一个使用单一内部队列加多个条件谓词的类简单得多。通过把两个条件谓词分离到两个等待集中，`Condition` 简化了使用单一通知的条件。使用更有效的 `signal`，而不是 `signalAll`，这就会减少相当数量的上下文转换，而且每次缓存操作都会触发对锁的请求。

就像内置的锁和条件队列一样，当使用显式的 `Lock` 和 `Condition` 时，也必须满足锁、条件谓词和条件变量之间的三元关系。涉及条件谓词的变量必须由 `Lock` 保护，检查条件谓词时以及调用 `await` 和 `signal` 时，必须持有 `Lock` 对象¹²。

在使用显式的 `Condition` 和内部条件队列之间作出选择，与你在 `ReentrantLock` 和 `synchronized` 之间进行选择是一样的：如果你需要使用需要一些高级特性，比如使用公平队列或者让每个锁对应多个等待集，这时使用 `Condition` 要好于使用内部条件队列。

（如果你需要 `ReentrantLock` 的高级特性，并已经在使用它，那么你已经作出了选择。）

14.4 剖析 Synchronizer

`ReentrantLock` 和 `Semaphore` 这两个接口有很多共同点。这些类都扮演了“阀门”的角色，每次只允许有限数目的线程通过它；线程到达阀门后，可以允许通过（`lock` 或 `acquire` 成功返回），可以等待（`lock` 或 `acquire` 阻塞），也可以被取消（`tryLock` 或 `tryAcquire` 返回 `false`，指明在允许的时间内，锁或者“许可”不可用）。更进一步，它们都允许可中断的、不可中断的、可限时的请求尝试，它们也都允许选择公平、不公平的等待线程队列。

给出了这个共同点后，你可能认为 `Semaphore` 是实现于 `ReentrantLock` 之上的，或者可能 `ReentrantLock` 是作为只有一个许可的 `Semaphore` 实现的。这是完全可以的；一个很常见的练习就是使用 `lock` 实现一个计数信号量（就如清单 14.12 中的 `SemaphoreOnLock`），以及使用计数信号量实现一个 `lock`。

事实上，它们的实现都用到一个共同的基类，`AbstractQueuedSynchronizer` (AQS)——和其他很多的 `Synchronizer` 一样。AQS 是一个用来构建锁和 `Synchronizer` 的框架，令人惊讶的是，使用 AQS 能够简单且高效地构造出应用广泛的大量的 `Synchronizer`。不仅 `ReentrantLock` 和 `Semaphore` 是构建于 AQS 上的，其他的还有 `CountDownLatch`、`ReentrantReadWriteLock`、`SynchronousQueue`¹³ 和 `FutureTask`。

¹² `ReentrantLock` 要求在调用 `signal` 或者 `signalAll` 时应该持有 `Lock`。但是 `Lock` 的其他实现也允许创建不符合这一规则的 `Condition`。

¹³ Java 6 中，基于 AQS 的 `SynchronousQueue` 已经被一个（伸缩性更好的）非阻塞的版本所取代。

清单 14.11 有限缓存使用显式的条件变量

```
@ThreadSafe
public class ConditionBoundedBuffer<T> {
    protected final Lock lock = new ReentrantLock();
    // 条件谓词: notFull (count < items.length)
    private final Condition notFull = lock.newCondition();
    // 条件谓词: notEmpty (count > 0)
    private final Condition notEmpty = lock.newCondition();
    @GuardedBy("lock")
    private final T[] items = (T[]) new Object[BUFFER_SIZE];
    @GuardedBy("lock") private int tail, head, count;

    // 阻塞, 直到: notFull
    public void put(T x) throws InterruptedException {
        lock.lock();
        try {
            while (count == items.length)
                notFull.await();
            items[tail] = x;
            if (++tail == items.length)
                tail = 0;
            ++count;
            notEmpty.signal();
        } finally {
            lock.unlock();
        }
    }

    // 阻塞, 直到: notEmpty
    public T take() throws InterruptedException {
        lock.lock();
        try {
            while (count == 0)
                notEmpty.await();
            T x = items[head];
            items[head] = null;
            if (++head == items.length)
                head = 0;
            --count;
            notFull.signal();
            return x;
        } finally {
            lock.unlock();
        }
    }
}
```

清单 14.12 使用 lock 实现的计数信号量

```
// 这并不是 java.util.concurrent.Semaphore 的真正实现
@ThreadSafe
public class SemaphoreOnLock {
    private final Lock lock = new ReentrantLock();
    // 条件谓词: permitsAvailable (permits > 0)
    private final Condition permitsAvailable = lock.newCondition();
    @GuardedBy("lock") private int permits;

    SemaphoreOnLock(int initialPermits) {
        lock.lock();
        try {
            permits = initialPermits;
        } finally {
            lock.unlock();
        }
    }

    // 阻塞, 直到: permitsAvailable
    public void acquire() throws InterruptedException {
        lock.lock();
        try {
            while (permits <= 0)
                permitsAvailable.await();
            --permits;
        } finally {
            lock.unlock();
        }
    }

    public void release() {
        lock.lock();
        try {
            ++permits;
            permitsAvailable.signal();
        } finally {
            lock.unlock();
        }
    }
}
```

AQS 解决了实现一个 Synchronizer 的大量细节，比如等待线程的 FIFO 队列。单独的 Synchronizer 可以定义一个灵活的标准，用来描述线程是否应该允许通过，还是需要等待。

用 AQS 构建 Synchronizer 会有很多好处。不仅仅是它能极大地减少实现过程中消耗的精力，而且你再也不必像没有使用 AQS 构建 Synchronizer 时那样，去应付多个竞争点了。在 SemaphoreOnLock 中，请求许可的操作在两个地方可能会阻塞——一是信号量的状态正在被锁保护着，另外是当许可不可用时。使用 AQS 构建的 Synchronizer 只可能在一个点上发生阻塞，这样降低了上下文切换的开销，并提高了吞吐量。AQS 的设计充分考虑了可伸缩性，所有 `java.util.concurrent` 中构建于 AQS 的 Synchronizer 都会从中获益。

14.5 AbstractQueuedSynchronizer

大多数开发者可能永远不会直接用到 AQS；标准的 Synchronizer 集涵盖了范围极为广泛的情况。但是了解一些标准的 Synchronizer 是如何实现的，有助于理解它们的运作机理。

一个基于 AQS 的 Synchronizer 所执行的基本操作，是一些不同形式的**获取**(acquire)和**释放**(release)。获取操作是状态依赖的操作，总能够阻塞。借助锁和信号量，“获取”的含义变得相当直观——获取锁或者许可——并且调用者可能不得不去等待，直到 Synchronizer 处于可发生的状态。CountDownLatch 的请求意味着“等待，直到闭锁到达它的终止态”，FutureTask 则意味着“等待，直到任务已经完成”。“释放”不是一个可阻塞的操作；“释放”可以允许线程在请求执行前阻塞。

为了让一个类具有状态依赖性，它必须拥有一些状态。同步类中有一些状态需要管理，这项任务落在了 AQS 上：它管理一个关于状态信息的单一整数，状态信息可以通过 `protected` 类型的 `getState`、`setState` 和 `compareAndSetState` 等方法进行操作。这可以用于表现任何状态；例如，`ReentrantLock` 用它来表现拥有它的线程已经请求了多少次锁，`Semaphore` 用它来表现剩余的许可数，`FutureTask` 用它来表现任务的状态（尚未开始、运行、完成和取消）。Synchronizer 也可以自己管理一些额外的状态变量；例如，`ReentrantLock` 保存了当前锁的所有者的追踪信息，这样它就能区分出是重进入的（`reentrant`）还是竞争的（`contended`）条件锁。

AQS 中的获取与释放是以清单 14.13 所示的形式呈现。获取操作可能是**独占的**，就像 `ReentrantLock` 一样；也可能是**非独占的**，就像 `Semaphore` 和 `CountDownLatch` 一样。这取决于不同的 Synchronizer。

一个获取操作分为两步。第一步，Synchronizer 判断当前状态是否允许被获得；如果是，就让线程执行，如果不是，获取操作阻塞或失败。这个判断是由 Synchronizer 的

语意决定的；举例来说，如果想成功地获取锁，锁必须是未被占有的；而如果想成功地获取闭锁，闭锁必须未处于终止状态。

第二步包括了可能需要更新的状态；一个想获取 Synchronizer 的线程会影响到其他线程是否能够获取它。例如，获取锁的操作将锁的状态从“未被占有”改变为“已被占有”；从 Semaphore 中获取许可的操作会减少剩余许可的数量。另一方面，一个线程对闭锁的请求操作却不会影响到其他线程是否能够获取它，所以获取闭锁的操作不会改变闭锁的状态。

清单 14.13 AQS 中获取和释放操作的规范式

```
boolean acquire() throws InterruptedException {
    while (state does not permit acquire) {
        if (blocking acquisition requested) {
            enqueue current thread if not already queued
            block current thread
        }
        else
            return failure
    }
    possibly update synchronization state
    dequeue thread if it was queued
    return success
}

void release() {
    update synchronization state
    if (new state may permit a blocked thread to acquire)
        unblock one or more queued threads
}
```

支持独占获取的 Synchronizer 应该实现 tryAcquire、tryRelease 和 isHeldExclusively 这几个受保护的方法，而支持共享获取的 Synchronizer 应该实现 tryAcquireShared 和 tryReleaseShared。AQS 中的 acquire、acquireShared、release 和 releaseShared 这些方法，会调用在 Synchronizer 子类中这些方法的 try-版本（tryAcquire、tryAcquireShared 等等），以此决定是否执行该操作。Synchronizer 的子类会根据其 acquire 和 release 的语意，使用 getState、setState 以及 compareAndSetState 来检查并更新状态，然后通过返回的状态值告知基类这次“获取”或“释放”的尝试是否成功。举例来说，从 tryAcquireShared 返回一个负值，说明获取操作失败；返回零说明 Synchronizer 是被独占获取的；返回正值说明 Synchronizer 是被非独占获取的。对于 tryRelease 和 tryReleaseShared 方法来说，如果能够释放一些正在尝试获取 Synchronizer 的线程，解除这些线程的阻塞，那么这两个方法将返回 true。

为了简化锁的实现，以支持条件队列（就像 ReentrantLock），AQS 还提供了一些机制，用于创建与 Synchronizer 相关联的条件变量。

14.5.1 一个简单的闭锁

清单 14.14 中的 `OneShotLatch` 是一个使用 AQS 实现的二元闭锁。它包含两个公共方法：`await` 和 `signal`，相当于获取和释放操作。最初，闭锁是关闭的；任何调用 `await` 的线程都会阻塞，直到打开闭锁。一旦闭锁被一个 `signal` 调用打开，等待中的线程就会被释放，而且随后到达闭锁的线程也被允许执行。

清单 14.14 二元闭锁使用 `AbstractQueuedSynchronizer`

```
@ThreadSafe
public class OneShotLatch {
    private final Sync sync = new Sync();

    public void signal() { sync.releaseShared(0); }

    public void await() throws InterruptedException {
        sync.acquireSharedInterruptibly(0);
    }

    private class Sync extends AbstractQueuedSynchronizer {
        protected int tryAcquireShared(int ignored) {
            // 如果闭锁打开则成功 (state == 1), 否则失败
            return (getState() == 1) ? 1 : -1;
        }

        protected boolean tryReleaseShared(int ignored) {
            setState(1); // 闭锁现在已打开
            return true; // 现在, 其他线程可以获得闭锁
        }
    }
}
```

在 `OneShotLatch` 中，AQS 类型的状态管理着闭锁的状态——关闭 (0) 或打开 (1)。`await` 方法调用 AQS 的 `acquireSharedInterruptibly`，后者随后请求 `OneShotLatch` 中的 `tryAcquireShared` 方法。`tryAcquireShared` 的实现必须返回一个值，表明请求操作能否进行。如果闭锁已经事先打开，`tryAcquireShared` 会返回成功，并允许线程通过；否则它会返回一个值，表明获取请求的尝试失败。`acquireSharedInterruptibly` 方法处理失败的方式，是把线程置入一个队列中，该队列中的元素都是等待中的线程。类似地，`signal` 调用 `releaseShared`，进而导致 `tryReleaseShared` 被调用。`tryReleaseShared` 的实现无条件地把闭锁的状态设置为打开，（通过返回值）表明 `Synchronizer` 处于完全

被释放的状态。这让 AQS 要求所有等待中的线程尝试去重新请求 Synchronizer，并且，由于 `tryAcquireShared` 会返回成功，所以这次请求操作会成功。

`OneShotLatch` 是一个功能全面的、可用的、性能良好的 Synchronizer，大约用了 20 多行代码就实现了。当时，它还少了些有用的特性——比如限时的请求操作以及检查闭锁状态的能力——但是这些实现起来同样很容易，因为 AQS 也提供了限时版本的获取方法，以及用于常见的检查操作的工具函数。

不通过委托，直接扩展 AQS 本也是可以实现 `OneShotLatch` 的。但是由于若干个原因，这种做法并不合理 [EJ Item 14]。这样做将破坏 `OneShotLatch` 接口（只有两个方法）的简洁性，并且虽然 AQS 的公共方法不允许调用者破坏闭锁的状态，调用者仍然很容易误用它。`java.util.concurrent` 中没有一个 Synchronizer 是直接扩展 AQS 的——相反，它们都委托了 AQS 的私有内部子类。

14.6 java.util.concurrent 的 Synchronizer 类中的 AQS

`java.util.concurrent` 中很多可阻塞的类，比如 `ReentrantLock`、`Semaphore`、`ReentrantReadWriteLock`、`CountDownLatch`、`SynchronousQueue` 和 `FutureTask`，全部是用 AQS 构建的。让我们来快速地浏览一下这些类分别是如何使用 AQS 的，这不会过多地深入细节（源代码是作为 JDK 的一部分可下载的¹⁴）。

14.6.1 ReentrantLock

`ReentrantLock` 只支持独占的获取操作，因此它实现了 `tryAcquire`、`tryRelease` 和 `isHeldExclusively`；清单 14.15 是非公平版本的 `tryAcquire`。`ReentrantLock` 使用同步状态持有锁获取操作的计数，还维护一个 `owner` 变量来持有当前拥有的线程标识符。只有在当前线程刚刚获取到锁，或者刚刚释放了锁的时候，才会修改 `owner`¹⁵。在 `tryRelease` 中，它检查 `owner` 域以确保当前线程在执行一个 `unlock` 操作之前，已经拥有了锁；在 `tryAcquire` 中，它使用这个域来区分重进入的获取操作尝试与竞争的获取操作尝试。

当一个线程尝试去获取锁时，`tryAcquire` 会首先请求锁的状态。如果锁未被占有，它会尝试更新锁的状态，表明锁已被占有。因为状态可能在被观察后的几条指令中被修改了，所以 `tryAcquire` 使用 `compareAndSetState` 来尝试原子地更新状态，表明这个锁现

¹⁴ 或者也可以以少许的许可限制从 <http://gee.cs.oswego.edu/dl/concurrency-interest> 获得。

¹⁵ 由于受保护的操作状态的方法具有 `volatile` 读写的内存语意，同时，`ReentrantLock` 只是在调用 `getState` 之后才会谨慎地读取 `owner` 域，并且只是在调用 `setState` 之前才会谨慎地写入 `owner`，`ReentrantLock` 可以“驾驭”于同步状态的内存语意之上，因此免去了进一步的同步（参见 16.1.4 节）。

清单 14.15 非公平的 ReentrantLock 中 tryAcquire 的实现

```
protected boolean tryAcquire(int ignored) {
    final Thread current = Thread.currentThread();
    int c = getState();
    if (c == 0) {
        if (compareAndSetState(0, 1)) {
            owner = current;
            return true;
        }
    } else if (current == owner) {
        setState(c+1);
        return true;
    }
    return false;
}
```

在已经被占有，并确保状态自最后一次观察后没有被修改过（参见 15.3 节中关于 `compareAndSet` 的描述）。假设锁状态表明它已经被占有，如果当前线程是锁的拥有者，那么获取计数会递减；如果当前线程不是锁的拥有者，那么获取操作的尝试会失败。

`ReentrantLock` 还利用了 AQS 内置的对多条件变量和多等待集的支持。`Lock.newCondition` 返回一个 `ConditionObject` 的新实例，这是一个 AQS 的内部类。

14.6.2 Semaphore 和 CountdownLatch

`Semaphore` 使用 AQS 类型的同步状态持有当前可用许可的数量。`tryAcquireShared` 方法（参见清单 14.16）首先计算剩余许可的数量，如果没有足够的许可，会返回一个值，表明获取操作失败。如果还有充足的许可剩余，`tryAcquireShared` 会使用 `compareAndSetState`，尝试原子地递减许可的计数。如果成功（这意味着自从许可计数被最后一次看到后，没有被改变过），会返回一个值，表明获取操作成功。返回值同样加入了是否允许其他共享获取尝试能否成功的信息，如果可以的话，其他等待的线程同样会解除阻塞。

无论是没有足够的许可，还是 `tryAcquireShared` 可以原子地更新许可计数，以响应获取操作时，`while` 循环都会终止。尽管任何给定的 `compareAndSetState` 调用，都可能由于与另一个线程的竞争而失败（参见 15.3 节），这使它会重试。在重试过合理的次数后，两个终止条件中的一个会变成真。类似地，`tryReleaseShared` 会递增许可计数，这会潜在地解除等待中的线程的阻塞，不断地重试直到成功地更新。`tryReleaseShared` 的返回值表明，释放操作是否可以解除其他线程的阻塞。

`CountDownLatch` 使用 AQS 的方式与 `Semaphore` 相似：同步状态持有当前的计数。`countDown` 方法调用 `release`，后者会导致计数器递减，并且在计数器已经到达零的时候，

清单 14.16 Semaphore 的 tryAcquireShared 和 tryAcquireShared 方法

```
protected int tryAcquireShared(int acquires) {
    while (true) {
        int available = getState();
        int remaining = available - acquires;
        if (remaining < 0
            || compareAndSetState(available, remaining))
            return remaining;
    }
}

protected boolean tryReleaseShared(int releases) {
    while (true) {
        int p = getState();
        if (compareAndSetState(p, p + releases))
            return true;
    }
}
```

解除所有等待线程的阻塞；await 调用 acquire，如果计数器已经到达零，acquire 会立即返回，否则它会被阻塞。

14.6.3 FutureTask

初看 FutureTask，它甚至不像是一个 Synchronizer。但是 Future.get 的语意非常类似于闭锁——如果发生了某些事件（FutureTask 表现的任务的完成或取消），线程就可以执行，否则线程会留在队列中，直到有事件发生。FutureTask 使用 AQS 类型的同步状态来持有任务的状态——运行、完成或取消。FutureTask 也维护了一些额外的状态变量，来持有计算的结果或者抛出的异常。它还维护了一个引用，指向正在运行计算任务的线程（如果它当前正处于运行态），这样如果任务被取消，就可以中断该线程。

14.6.4 ReentrantReadWriteLock

ReadWriteLock 的接口要求了两个锁——一个读者锁和一个写者锁——但是在基于 AQS 的 ReentrantReadWriteLock 实现中，一个单独的 AQS 子类管理了读和写的加锁。ReentrantReadWriteLock 使用一个 16 位的状态为写锁（write-lock）计数，使用另一个 16 位的状态为读锁（read-lock）计数。对读锁的操作使用共享的获取与释放的方法；对写锁的操作使用独占的获取与释放的方法。

AQS 在内部维护一个等待线程的队列，持续追踪一个线程是否被独占请求，或者被共

享访问。在 `ReentrantReadWriteLock` 中，当锁可用时，如果位于队列头部的线程同时也正在准备写访问，线程会得到锁；如果位于队列头部的线程正在准备读访问，那么队列中所有首个写线程之前的线程都会得到锁¹⁶。

总结

如果你需要实现一个依赖于状态的类——如果不能满足依赖于状态的前提条件，类的方法必须阻塞——最佳的策略通常是将其构建于现有的库类之上，比如 `Semaphore`、`BlockingQueue` 或者 `CountDownLatch`，例如 187 页的 `ValueLatch` 那样。但是，有时现有的库类不能提供足够的功能；在这种情况下，你可以使用内部条件队列、显式 `Condition` 对象或者 `AbstractQueuedSynchronizer`，来构建属于自己的 `Synchronizer`。由于“管理状态的独立性”机制必须紧密依赖于“确保状态一致性”机制，所以内部条件队列与内部锁紧密地绑定到了一起。类似地，显式的 `Condition` 是与显式的 `Lock` 也是紧密地绑定到一起的，相比于内部条件队列，它还提供了一个可扩展的特征集，包括“多等待集每锁”，可中断或不可中断的条件等待，公平或非公平的队列，以及基于最终时限的等待。

¹⁶ 这种机制就像一些读写锁的实现那样，不会承诺读者优先或者写者优先策略的选择。为此，AQS 的等待队列本可以不同于先进先出队列，或者甚至使用两个队列。但是，在实际中是很少需要这么严格的排序策略的。如果非公平版本的 `ReentrantReadWriteLock` 无法提供可接受的活跃度，安全版本通常会转而提供令人满意的排序，而且保证读者和写者不会产生饥饿。

原子变量与非阻塞同步机制

Atomic Variables and Nonblocking Synchronization

`java.util.concurrent` 包中的许多类,比如 `Semaphore` 和 `ConcurrentLinkedQueue`,都提供了比使用 `synchronized` 更好的性能和可伸缩性。这一章,我们来学习这些性能提升的原始来源:原子变量和非阻塞的同步机制。

近来很多关于并发算法的研究都聚焦在**非阻塞算法**(nonblocking algorithms)上,这种算法使用底层原子化的机器指令取代锁,比如**比较并交换**(compare-and-swap),从而保证数据在并发访问下的一致性。非阻塞算法广泛用于操作系统和 JVM 中的线程和进程调度、垃圾回收以及实现锁和其他的并发数据结构。

与基于锁的方案相比,非阻塞算法的设计和实现都要复杂得多,但是它们在可伸缩性和活跃度上占有很大的优势。因为非阻塞算法可以让多个线程在竞争相同资源时不会发生阻塞,所以它能在更精化的层面上调整粒度,并能大大减少调度的开销。进一步而言,它们对死锁和其他活跃度问题具有免疫性。在基于锁的算法中,如果一个线程在持有锁的时候休眠,或者停滞不前,那么其他线程就都不可能前进了,而非阻塞算法不会受到单个线程失败的影响。在 Java 5.0 中,使用**原子变量类**(atomic variable classes),比如 `AtomicInteger` 和 `AtomicReference`,能够高效地构建非阻塞算法。

即使你不使用原子变量开发非阻塞算法,它也同样可以用作“更优的 `volatile` 变量”。原子变量提供了与 `volatile` 类型变量相同的内存语义,同时还额外支持原子更新——使它们能更加理想地用于计数器、序列发生器和统计数据收集等,另外也比基于锁的方案具有更加出色的可伸缩性。

15.1 锁的劣势

使用一致的加锁协议来协调对共享状态的访问,确保无论哪个线程持有守护变量的

锁，它们都能独占访问这些变量，并且对变量的任何修改对其他随后获得同一锁的线程都是可见的。

现代 JVM 能够对非竞争锁的获取和释放进行优化，让它们非常高效，但是如果多个线程同时请求锁，JVM 就需要向操作系统寻求帮助。倘若出现了这种情况，一些“不幸的”线程将会挂起，并稍后恢复运行¹。从线程开始恢复起，到它真正被调度前，可能必须等待其他线程完成它们的调度限额规定的时间。挂起和恢复线程会带来很大的开销，并通常伴有冗长的中断。对于基于锁，并且其操作过度细分的类（比如同步容器类，大多数方法只包含很少的操作），**当频繁地发生锁的竞争时**，调度与真正用于工作的开销间的比值会很可观。

volatile 变量与锁相比是更轻量的同步机制，因为它们不会引起上下文的切换和线程调度。然而，volatile 变量与锁相比有一些局限性：尽管它们提供了相似的可见性保证，但是它们不能用于构建原子化的复合操作。这意味着当一个变量依赖其他变量时，或者当变量的新值依赖于旧值时，是不能用 volatile 变量的。这些都限制了 volatile 变量的使用，因此它们不能用于实现可靠的通用工具，比如计数器，或互斥体（mutex）²。

例如，尽管自增操作（++i）**看起来**像是原子操作，事实上有 3 个独立操作——获取变量当前值，为该值加一，然后写回更新值。为了不丢失更新，整个的读-改-写操作必须是原子的。到目前为止，我们见过的唯一实现它的方法是加锁，如第 56 页中 Counter 所示。

Counter 是线程安全的，在几乎没有竞争的条件下会运行良好。但是在竞争下，性能会由于上下文切换的开销和调度延迟而受到损失。倘若临时占有锁的线程进入休眠，将成为又一个问题，因为会有线程在这个错误的时间里请求锁，

加锁还有其他的缺点。当一个线程正在等待锁时，它不能做任何其他事情。如果一个线程在持有锁的情况下发生了延迟（原因包括页错误、调度延迟，或者类似情况），那么其他所有需要该锁的线程都不能前进了。如果阻塞的线程是优先级很高的线程，持有锁的线程优先级较低，那么会造成严重问题——性能风险，被称为**优先级倒置**（priority inversion）。即使更高的优先级占先，它仍然需要等待锁被释放，这导致它的优先级会降至与优先级较低的线程相同的水平。如果持有锁的线程发生了永久性的阻塞（因为无限循环，死锁，活锁和其他活跃度失败），所有等待该锁的线程都不会前进了。

即使忽略这样的风险，加锁对于细分的操作而言，仍然是重量级（heavyweight）的机

¹ 明智的 JVM 在线程竞争锁时不需要挂起线程；它可以使用系统运行中的剖析数据提供的以前占用锁的时间长短数据，来适当决定挂起还是自旋等待。

² 虽然理论上存在可能性，但几乎完全不能使用 volatile 语义来构建互斥体和其他同步机制，见（Raynal, 1986）。

制，比如递增计数器。本应该有更好的技术用来管理线程之间的竞争——某些类似于 `volatile` 变量的机制，但是还要支持原子化更新。幸运的是，现代处理器为我们提供了这样的机制。

15.2 硬件对并发的支持

独占锁是一项**悲观**的技术——它假设最坏情况（如果你不锁门，捣蛋鬼就会闯入，并破坏物品的秩序），并且会通过获得正确的锁来避免其他线程的打扰，直到作出保证才能继续进行。

对于细粒度的操作，有另外一种选择通常更加有效——**乐观**的解决方法。凭借新的方法，我们可以指望不受打扰地完成更新。这个方法依赖于**冲突监测**，从而能判定更新过程中是否存在来自于其他成员的干涉，在冲突发生的情况下，操作失败，并会重试（也可能不重试）。这个乐观的方案就好比 we 常说的：“宽恕比准许更容易”，其中“更容易”意味着“更有效率”。

针对多处理器系统设计的处理器提供了特殊的指令，用来管理并发访问的共享数据。早期处理器具有原子化的**测试并设置**（test-and-set），**获取并增加**（fetch-and-increment）以及**交换**（swap）指令，这些对于实现互斥已经足够了，并能够用于实现更成熟的并发对象。如今，几乎所有现代的处理器的都具有一些形式的原子化的读-改-写指令，比如**比较并交换**（compare-and-swap）和**加载链接/存储条件**（load-linked/store-conditional）。操作系统和 JVM 使用这些指令来实现锁和并发的数据结构，但是直到 Java 5.0 以前这些还不能直接为 Java 类所使用。

15.2.1 比较并交换

包括 IA32 和 Sparc 在内的大多数处理器使用的架构方案都实现了**比较并交换**（CAS）**指令**。（其他处理器，比如 PowerPC，是用一对指令实现了相同的功能：**链接加载/存储条件**。）CAS 有 3 个操作数——内存位置 *V*、旧的预期值 *A* 和新值 *B*。当且仅当 *V* 符合旧预期值 *A* 时，CAS 用新值 *B* 原子化地更新 *V* 的值；否则它什么都不会做。在任何一种情况下，都会返回 *V* 的真实值。（这个变量称为 compare-and-set，无论操作是否成功都会返回。）CAS 的意思是：“我认为 *V* 的值应该是 *A*，如果是，那么将其赋值为 *B*，若不是，则不修改，并告诉我应该为多少”。CAS 是一项乐观技术——它抱着成功的希望进行更新，并且如果另一个线程在上次检查后更新了该变量，它能够发现错误。清单 15.1 中的 SimulatedCAS 阐释了 CAS 语义（但是并非真正的实现或者执行方式）。

当多个线程试图使用 CAS 同时更新相同的变量时，其中一个会胜出，并更新变量的值，而其他的都会失败。失败的线程不会被挂起（但是没有获取锁的线程就会被挂起）；它

清单 15.1 模拟 CAS 操作

```
@ThreadSafe
public class SimulatedCAS {
    @GuardedBy("this") private int value;

    public synchronized int get() { return value; }

    public synchronized int compareAndSwap(int expectedValue,
                                             int newValue) {
        int oldValue = value;
        if (oldValue == expectedValue)
            value = newValue;
        return oldValue;
    }

    public synchronized boolean compareAndSet(int expectedValue,
                                              int newValue) {
        return (expectedValue
                == compareAndSwap(expectedValue, newValue));
    }
}
```

们会被告知这一次赛跑失利，但允许再尝试。因为一个线程在竞争 CAS 时失败不会被阻塞，它可以决定是否重试，进行一些补救行动，或者什么都不做³。这样的灵活性大大减少了与锁相关的活跃度风险（尽管在一些极端的情况下会引入活锁风险——参见 10.3.3）。

使用 CAS 的典型模式是：首先从 *V* 中读取值 *A*，由 *A* 生成新值 *B*，然后使用 CAS 原子化地把 *V* 的值由 *A* 改成 *B*，并且期间不能有其他线程改变 *V* 的值。因为 CAS 能够发现来自其他线程的干扰，所以即使不使用锁，它也能够解决原子化地实现读-写-改的问题。

15.2.2 非阻塞计数器

清单 15.2 中 `CasCounter` 利用 CAS 实现了线程安全的计数器。自增操作遵循了经典形式——取得旧值，根据它计算出新值（加 1），并使用 CAS 设定新值。如果 CAS 失败，立即重试该操作。尽管在竞争十分激烈的情况下，更希望等待或者回退，以避免重试造成的活锁，但是，通常反复重试都是合理的策略。

³ 什么都不做可能是 CAS 响应失败时最明智的选择；在非阻塞算法中，比如 15.4.2 节中的链接队列算法，失败的 CAS 意味着其他线程已经完成了你所计划的这件事情。

CasCounter 不会发生阻塞, 如果其他线程同时更新计数器, 它会进行数次重试⁴。(实践中, 如果你仅仅需要一个计数器, 或者序列生成器, 直接使用 AtomicInteger 或者 AtomicLong 就可以了, 它们提供原子化的自增方法和其他算术方法。)

清单 15.2 使用 CAS 实现的非阻塞计数器

```
@ThreadSafe
public class CasCounter {
    private SimulatedCAS value;

    public int getValue() {
        return value.get();
    }

    public int increment() {
        int v;
        do {
            v = value.get();
        }
        while (v != value.compareAndSwap(v, v + 1));
        return v + 1;
    }
}
```

初看起来, 基于 CAS 的计数器看起来比基于锁的计数器性能差一些: 它具有更多的操作和更复杂的控制流, 表面看来还依赖于复杂的 CAS 操作。但是, 实际上基于 CAS 的计数器, 性能上远远胜过了基于锁的计数器, 即使只有很小的竞争, 或者不存在竞争。获取非竞争锁的快速路径, 通常至少需要一个 CAS 加上一个锁相关的细节琐事, 所以, 基于锁的计数器的最好情况也要比基于 CAS 的一般情况做更多的事情。因为 CAS 在大多数情况下都能成功(假设竞争程度中等偏下), 硬件将能够正确地预知隐藏在 while 循环中的分支, 使本应更复杂的控制逻辑的开销最小化。

加锁的语法可能比较简洁, 但是 JVM 和 OS 管理锁的工作却并不简单。加锁需要遍历 JVM 中整个复杂的代码路径, 并可能引起系统级的加锁、线程挂起以及上下文切换。在最优的情况下, 加锁需要至少一个 CAS, 所以使用锁时把 CAS 抛在一边, 几乎不能节省任何真正的执行开销。另一方面, 程序内部执行 CAS 不会调用到 JVM 的代码、系统调用或者调度活动。在应用级看起来越来越长的代码路径, 在考虑到 JVM 和 OS 的时候, 事实上会变

⁴ 理论上, 如果其他线程一直在 CAS 的竞争中处于优势, 执行线程本应会重试任意多次; 事实上, 这个类型的饥饿很少发生。

成更短的代码。CAS 最重要的缺点是：它强迫调用者处理竞争（通过重试、回退，或者放弃）；然而在锁被获得之前，却可以通过阻塞自动处理竞争⁵。

CAS 的性能随处理器数量变化很大。在单 CPU 系统中，CAS 通常的职责与一个脉冲周期相似，因为不需要处理器间的同步。到写作本书时为止，非竞争的 CAS 在多 CPU 系统中花费 10 到 150 个 CPU 周期的开销；CAS 的性能是一个不断改变的标准，不仅在不同的体系架构之间，就是在相同处理器的不同版本之间也会发生改变。厂商迫于竞争的压力，在近几年内还会持续提高 CAS 的性能。对此的一句金玉良言是，即使是在“快速路径”上，获取和释放**无竞争**锁的开销大约也是 CAS 的两倍。

15.2.3 JVM 对 CAS 的支持

如此说来，Java 代码如何能够从处理器执行 CAS 中确实获得收益呢？在 Java 5.0 之前，倘若不编写本机代码，是无法做到这一点的。在 Java 5.0 中，引入了底层的支持，将 `int`、`long` 和对象的引用暴露给 CAS 操作，并且 JVM 把它们编译为底层硬件提供的最有效的方法。在支持 CAS 的平台上，运行时把它们编排成恰当的（多条）机器指令；在最坏的情况下，如果 CAS 式的指令不可用，JVM 会使用自旋锁。这些底层的 JVM 支持，用于那些具有原子化变量的类（`java.util.concurrent.atomic` 中的 `AtomicXxx`），从而为数字类型和引用类型提供有效的 CAS 操作；而且，这些原子变量类还用于直接或间接地实现 `java.util.concurrent` 中大部分类。

15.3 原子变量类

原子变量比锁更精巧，更轻量，并且在多处理器系统中，对实现高性能的并发代码非常关键。原子变量限制了单个变量的竞争范围；这是你所能得到的最好结果了（假设你的算法能够以这么细的粒度来实现）。更新原子变量的快速（非竞争）路径，并不会比获取锁的快速路径差，并且通常会更快；而慢速路径绝对会比锁的慢速路径快，因为它不会引起线程的挂起和重新调度。在使用原子变量取代锁的算法中，线程更不易出现延迟，如果它们遇到竞争，也更容易恢复。

原子变量类，提供了广义的 `volatile` 变量，以支持原子的、条件的读-写-改操作。`AtomicInteger` 代表一个 `int` 值，并提供了 `get` 和 `set` 方法，它们与读取和写入可变的 `int` 有着相同的内存语义。它同样提供了一个 `compareAndSet` 方法（如果该方法成功返回，

⁵ 事实上，CAS 最大的缺陷在于难以正确地构建外围算法。

其内存效果和同时读取并写入一个 `volatile` 变量是一样的），以及原子化的插入、递增、递减等方法，这些是为了使用方便。`AtomicInteger` 与扩展的 `Counter` 表面上看有共同之处，但是在竞争条件下 `AtomicInteger` 提供了更好的可伸缩性，因为它可以直接利用硬件对并发的支持。

原子变量类共有 12 个，分成 4 组：计量器、域更新器（`field updater`）、数组以及复合变量。最常用的原子变量是计量器：`AtomicInteger`、`AtomicLong`、`AtomicBoolean` 以及 `AtomicReference`。他们都支持 CAS；`AtomicInteger` 和 `AtomicLong` 还支持算术运算。（也可以模拟其他基本类型的原子变量，对于 `short` 或者 `byte`，把它们的值强制转化为 `int`；对于浮点数，使用 `floatToIntBits` 或 `doubleToLongBits`。）

原子化的数组类（只有 `Integer`、`Long` 和 `Reference` 版本的可用），它的元素是可以被原子化地更新的。原子数组类为数组的元素提供了 `volatile` 的访问语义，这是普通数组所没有的特性——`volatile` 类型的数组只针对数组的引用具有 `volatile` 语义，而不是它的元素。（其他类型的原子变量在 15.4.3 节和 15.4.4 节中讨论。）

尽管原子化的计量器类扩展于 `Number`，它们并没有扩展基本类型的包装类，比如 `Integer` 或 `Long`。事实上，它们也不应该这样：基本类型的包装类是不可变的，而原子变量类是可变的。原子变量类同样没有重新定义 `hashCode` 或 `equals`；每一个实例都是独特的。与其他可变对象一样，它们也不适合作哈希容器的键。

15.3.1 原子变量是“更佳的 `volatile`”

在 3.4.2 节中，我们使用 `volatile` 引用一个不可变对象，用来原子化地更新多个状态变量。这个例子依赖于“检查再运行（`check-then-act`）”，但是在特殊情况下，竞争是无害的，因为我们并不关心是否偶尔发生更新失败。在大多数其他情况下，“检查再运行”不可能是无害的，并能够危及到数据的一致性。例如，如果 67 页上的 `NumberRange` 不能使用 `volatile` 引用一个不可变的 `holder` 对象，这不足以安全地实现对上下边界的更新的。也不能使用原子化的整数来存储边界。因为一个不变约束限制了两个数值，并且它们不能在遵循约束的情况下被同时更新，如果一个值域类使用 `volatile` 引用，或者使用多个原子化的整数，将会造成不安全的“检查再运行”顺序。

我们能够把 `OneValueCache` 的技术结合到原子化引用中，通过原子化地更新持有上下边界的不变类的引用，来缩小竞争条件。清单 15.3 中的 `CasNumberRange` 使用了 `AtomicReference` 和 `IntPair` 来保持状态；通过使用 `compareAndSet`，它能够避开 `NumberRange` 的竞争条件，更新上下界。

清单 15.3 使用 CAS 避免多元的不变约束

```
public class CasNumberRange {
    @Immutable
    private static class IntPair {
        final int lower; // 不变约束: lower <= upper
        final int upper;
        ...
    }
    private final AtomicReference<IntPair> values =
        new AtomicReference<IntPair>(new IntPair(0, 0));

    public int getLower() { return values.get().lower; }
    public int getUpper() { return values.get().upper; }

    public void setLower(int i) {
        while (true) {
            IntPair oldv = values.get();
            if (i > oldv.upper)
                throw new IllegalArgumentException(
                    "Can't set lower to " + i + " > upper");
            IntPair newv = new IntPair(i, oldv.upper);
            if (values.compareAndSet(oldv, newv))
                return;
        }
    }
    // setUpper 完全类似
}
```

15.3.2 性能比较：锁与原子变量

为了证明锁和原子变量之间可伸缩性的差异，我们建立了基准，比较几个不同的伪随机数字生成器（PRNG）实现。在 PRNG 中，下一个随机数字是上一个数字确定的函数值，所以 PRNG 必须记得前一个数值，作为其状态的一部分。

清单 15.4 和 15.5 列出了线程安全的 PRNG 的两种实现，一个使用 `ReentrantLock`，另一个使用 `AtomicInteger`。测试驱动程序分别重复地调用它们；每个迭代生成一个随机数字（它会取得，然后修改共享的 `seed` 状态），并执行一些严格针对线程本地数据的操作迭代。这个模拟典型的操作，包括操作共享状态的部分，和操作线程本地状态的部分。

图 15.1 和图 15.2 展示了模拟低级和中级工作强度下，每个迭代的吞吐量。在低级强度的线程本地计算中，锁和原子变量都经历了激烈的竞争；在强化了线程本地计算后，

清单 15.4 使用 ReentrantLock 实现随机数字生成器

```
@ThreadSafe
public class ReentrantLockPseudoRandom extends PseudoRandom {
    private final Lock lock = new ReentrantLock(false);
    private int seed;

    ReentrantLockPseudoRandom(int seed) {
        this.seed = seed;
    }

    public int nextInt(int n) {
        lock.lock();
        try {
            int s = seed;
            seed = calculateNext(s);
            int remainder = s % n;
            return remainder > 0 ? remainder : remainder + n;
        } finally {
            lock.unlock();
        }
    }
}
```

清单 15.5 使用 AtomicInteger 实现随机数字生成器

```
@ThreadSafe
public class AtomicPseudoRandom extends PseudoRandom {
    private AtomicInteger seed;

    AtomicPseudoRandom(int seed) {
        this.seed = new AtomicInteger(seed);
    }

    public int nextInt(int n) {
        while (true) {
            int s = seed.get();
            int nextSeed = calculateNext(s);
            if (seed.compareAndSet(s, nextSeed)) {
                int remainder = s % n;
                return remainder > 0 ? remainder : remainder + n;
            }
        }
    }
}
```

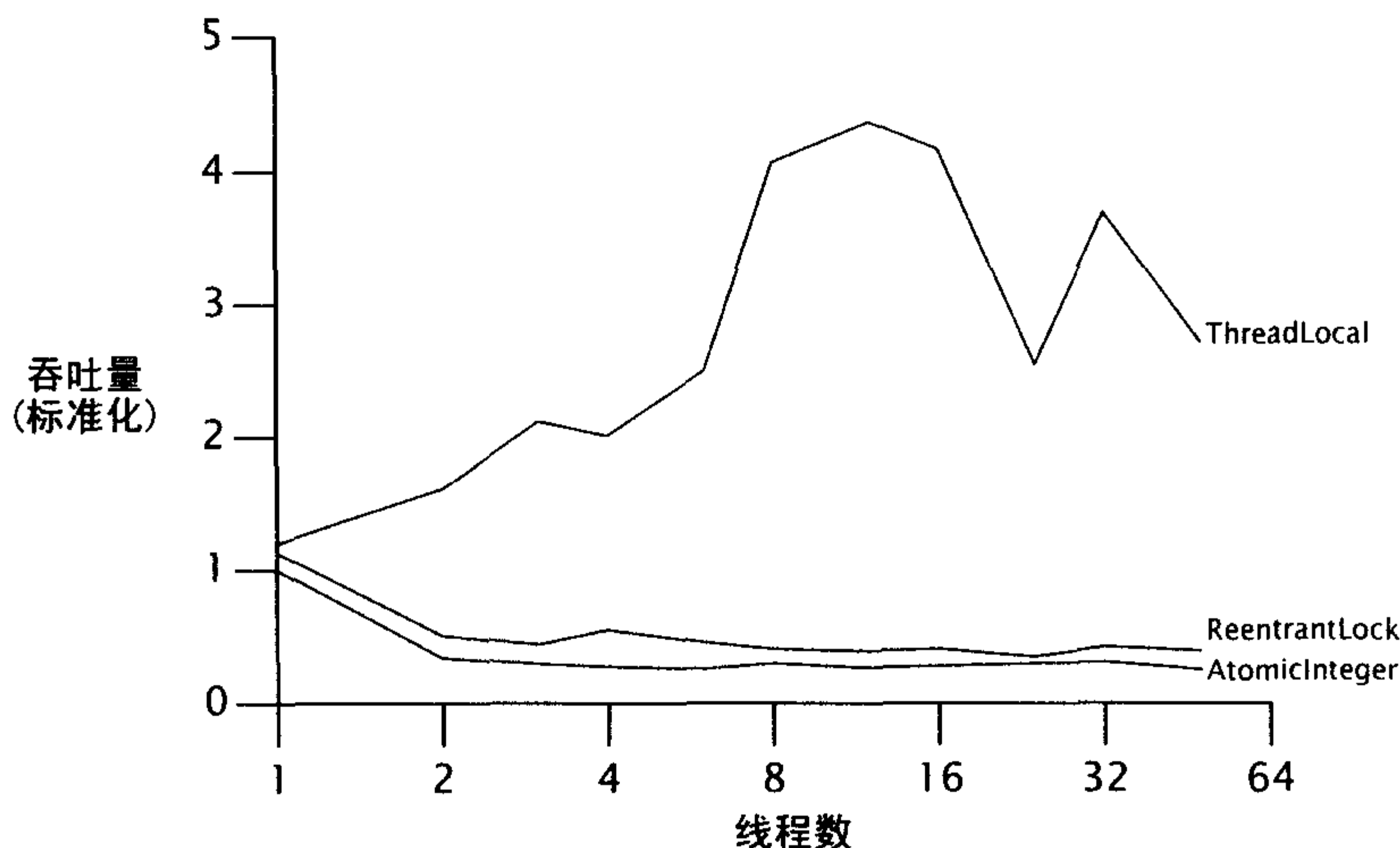


图 15.1 Lock 和 AtomicInteger 在激烈竞争下的性能比较

线程访问锁和原子变量的频率减小了，因而它们的竞争压力也变小减小了。

如图所示，在激烈竞争下，锁胜过原子变量，但是在真实的竞争条件下，原子变量会胜过锁⁶。这是因为锁通过挂起线程来响应竞争，减小了 CPU 的利用和共享内存总线上的同步通信量。（这与在生产者-消费者线程中，以阻塞生产者来减小消费者负荷，使它们能够赶上进度的设计是类似的。）从另一方面来看，使用原子变量把竞争管理推回给调用类。与大多数基于 CAS 的算法相比，AtomicPseudoRandom 通过不断反复尝试来响应竞争，这通常是正确的，但是在激烈竞争环境下，会带来更多竞争。

我们会批评 AtomicPseudoRandom 写得太差了，会声讨原子变量与锁相比是错误的选择，但在这之前，我们应该实现图 15.1 中实现的竞争级别，这很难做到：不存在任何一个程序，除了竞争锁或原子变量，其他什么都不做。在实践中，原子化的伸缩性比锁更好，因为在典型的竞争级别中，原子性会带来更好的效率。

锁与原子化随竞争度的不同，性能发生的改变阐明了各自的优势和劣势。在中低程度的竞争下，原子化提供更好的可伸缩性；在高强度的竞争下，锁能够更好地帮助我们避免竞争。（基于 CAS 的算法在单 CPU 系统中，胜过基于锁的算法，因为 CAS 在单 CPU 的

⁶ 在其他领域同样能够得到印证：交通信号灯在拥挤的交通状况下能够带来更好的吞吐量，但是环岛在低拥堵的交通状况下带来更好的吞吐量；以太网使用的竞争设计在低通信量的情况下运行良好，但是在高通信量下，使用令牌环的标记传送设计更适宜。

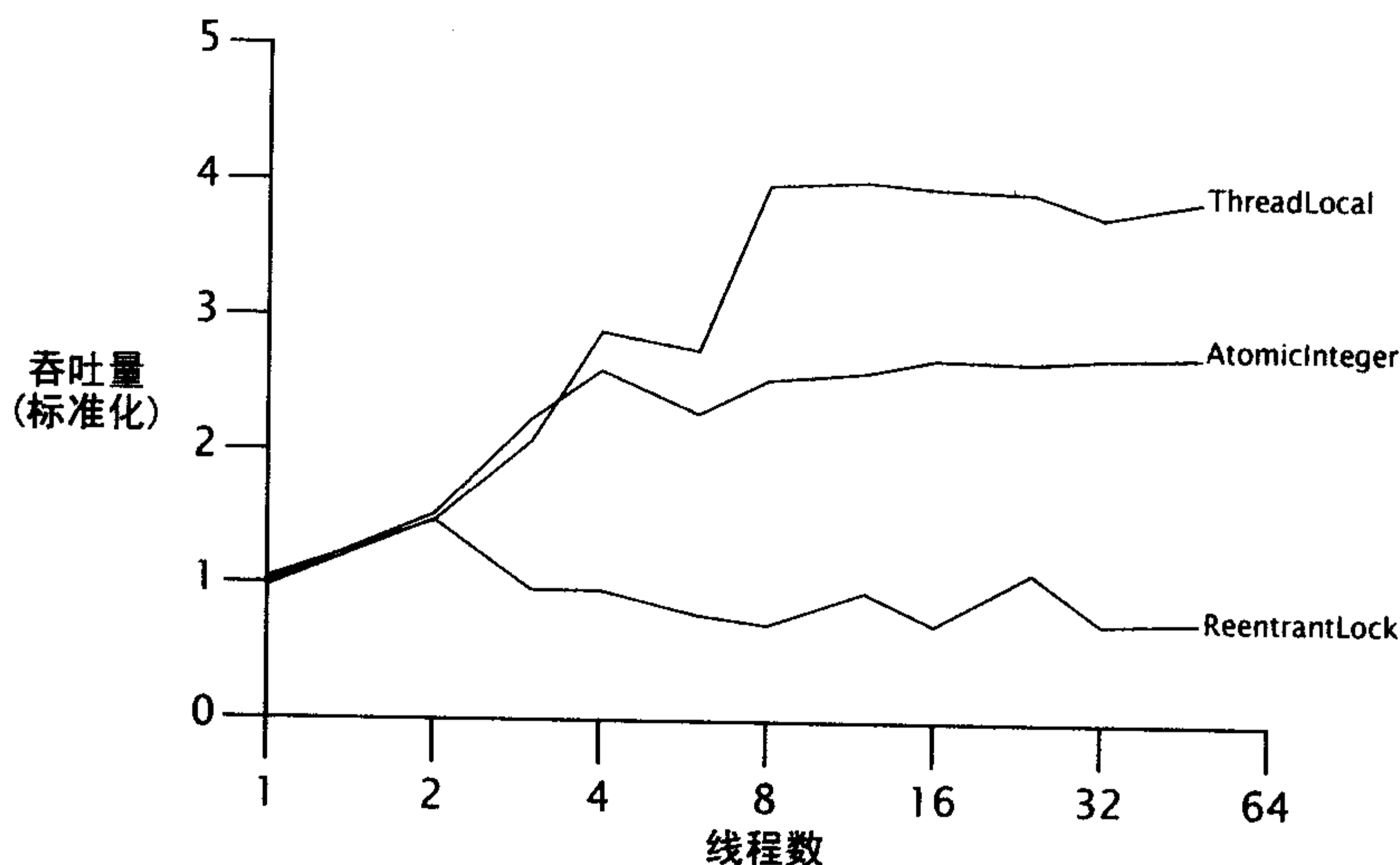


图 15.2 在中等强度的竞争下 Lock 和 AtomicInteger 的性能对比

系统下总是成功的，除非遇到偶发情况，线程在读-改-写操作的中途被抢占。)

图 15.1 和图 15.2 都具有第三条曲线；使用 ThreadLocal 的 PRNG 状态实现 PseudoRandom。这个实现方案改变了类的行为——每个线程查看自己私有的伪随机数字序列，而不是所有线程共享同一序列——但是如果能够避免的话，不共享状态的开销会更小。我们能够通过更有效地处理竞争改进可伸缩性，但是真正的可伸缩性完全是通过减少竞争实现的。

15.4 非阻塞算法

基于锁的算法会带来一些活跃度失败的风险。如果线程在持有锁的时候因为阻塞 I/O，页面错误，或其他原因发生延迟，很可能所有线程都不能前进了。一个线程的失败或挂起不应该影响其他线程的失败或挂起，这样的算法被称为**非阻塞**（nonblocking）算法；如果算法的每一步骤中都有一些线程能够继续执行，那么这样的算法称为**锁自由**（lock-free）算法。在线程间使用 CAS 进行协调，这样的算法如果能构建正确的话，它既是**非阻塞**的，又是**锁自由**的。非竞争的 CAS 总是能够成功，如果多个线程以一个 CAS 竞争，总会有一个胜出并前进。非阻塞算法对死锁和优先级倒置有“免疫性”（但它们可能会出现饥饿和活锁，因为它们允许重进入）。到目前为止，我们已经见到一个非阻塞算法：CasCounter。好的非阻塞算法已经在多种常见的数据结构上现身，包括栈、队列、优先级队列、哈希

表——设计新的数据结构的任务最好还是留给专家们。

15.4.1 非阻塞栈

实现同等功能前提下，非阻塞算法被认为比基于锁的算法更加复杂。创建非阻塞算法的前提是为维护数据的一致性，解决如何把原子化范围缩小到一个**唯一**变量。在链式容器类（比如队列）中，有时你可以不必再进行状态转换了；而把它变为对单独链接的修改；进而，你可以使用一个 `AtomicReference` 来表达每一个必须被原子更新的链接。

栈是最简单的链式数据结构：每个元素仅仅引用唯一的元素，并且每个元素只被一个元素引用。清单 15.6 中 `ConcurrentStack` 显示了如何使用原子引用来构建栈。栈是 `Node` 元素的一个链表，栈顶作为根，每个元素都包含一个值和一个指向下一个元素的链接。`push` 方法创建一个新的链接节点，该节点的 `next` 域指向当前的栈顶，然后使用 CAS 把这个新节点加入到栈中。如果我们使用栈的时候栈顶元素恰为新加入的节点，那么 CAS 成功了；如果栈顶元素变化了（因为其他线程在我们开始前插入或移除了元素），CAS 就失败了，`push` 方法会根据当前栈的状态更新节点，反复尝试。在这两种情况下，栈在 CAS 操作后仍然能够保持一致性。

`CasCounter` 和 `ConcurrentStack` 阐释了非阻塞算法的所有特性：一些任务的完成具有不确定性，并可能必须重做。在 `ConcurrentStack` 中，当我们构建表示新节点的 `Node` 时，我们希望 `next` 的引用值在把该节点装入栈的时候仍然有效，但是同时仍然为重试作好了准备。

在 `ConcurrentStack` 等中用到的非阻塞算法，其线程安全性源于：`compareAndSet` 既能提供原子性，又能提供可见性保证，加锁也同样如此。当一个线程改变栈的状态时，它使用具有与写入 `volatile` 变量相同的内存效应的 `compareAndSet`。当线程检查栈的时候，通过调用同一个 `AtomicReference` 的 `get` 方法来实现，它具有与读取 `volatile` 变量相同的内存效应。所以任何线程修改都能够安全地发布给其他正在检查列表状态的线程。并且这个列表通过 `compareAndSet` 进行修改，更新 `top` 的引用或者因发现其他线程的干扰而失败，这些都是原子化地进行的。

15.4.2 非阻塞链表

到目前为止我们已经见到了两个非阻塞算法，即计数器和栈，它们阐释了使用 CAS “投机地”更新值这一最基本的模式，如果更新失败就重试。构建非阻塞算法的窍门是：缩小原子化的范围到唯一的变量。在计数器中，这很简单，在栈中，实现这一点也非常明显，但是对于一些更复杂的数据结构，比如队列、哈希表，或者树，它可能变得更加微妙。

一个链接队列比栈更加复杂，因为它需要支持首尾的快速访问。为了实现，它会维护

清单 15.6 使用 Treiber 算法 (Treiber, 1986) 的非阻塞栈

```
@ThreadSafe
public class ConcurrentStack <E> {
    AtomicReference<Node<E>> top = new AtomicReference<Node<E>>();

    public void push(E item) {
        Node<E> newHead = new Node<E>(item);
        Node<E> oldHead;
        do {
            oldHead = top.get();
            newHead.next = oldHead;
        } while (!top.compareAndSet(oldHead, newHead));
    }

    public E pop() {
        Node<E> oldHead;
        Node<E> newHead;
        do {
            oldHead = top.get();
            if (oldHead == null)
                return null;
            newHead = oldHead.next;
        } while (!top.compareAndSet(oldHead, newHead));
        return oldHead.item;
    }

    private static class Node <E> {
        public final E item;
        public Node<E> next;

        public Node(E item) {
            this.item = item;
        }
    }
}
```

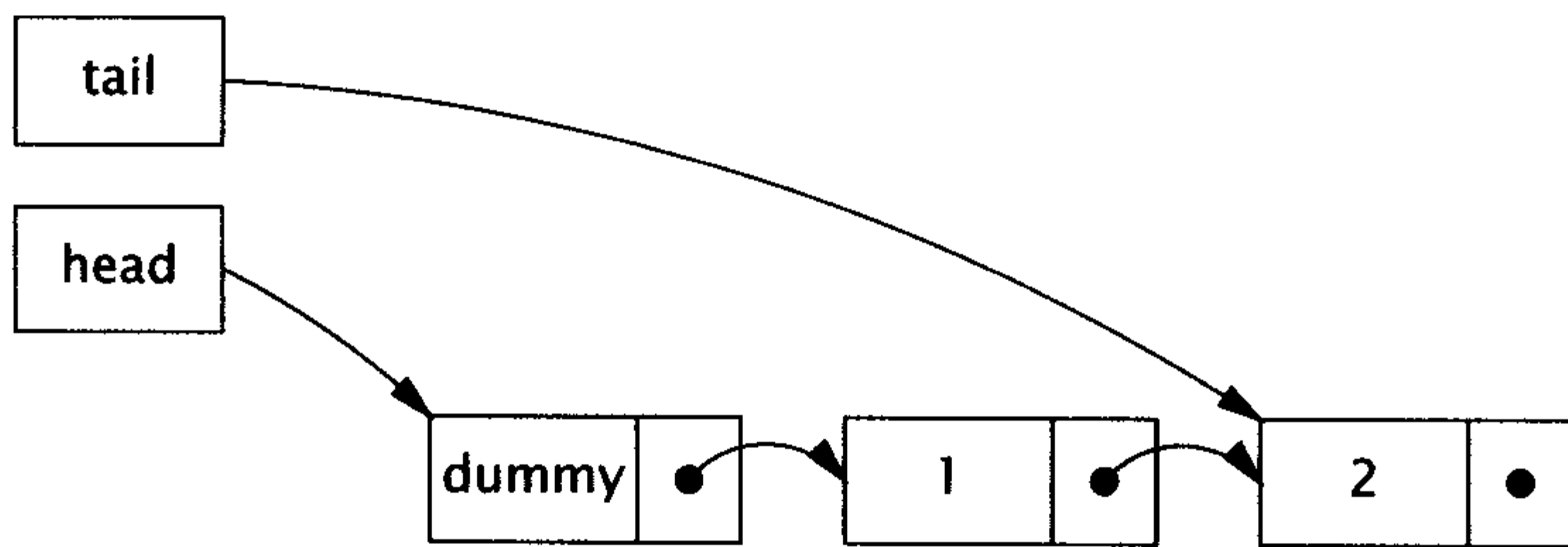


图 15.3 队列中有两个元素处于静止状态

独立的队首指针和队尾指针。两个指针初始时都指向队列的末尾节点：当前最后一个元素的 `next` 指针，即队尾指针。在成功加入新元素时，两个节点都需要更新——原子化更新。乍看起来，这不能通过原子变量实现；彼此分离的 CAS 操作需要更新两个指针，并且如果第一个成功，第二个失败，队列将处于不一致的境地。并且即使两个都成功了，另一个线程也可能在两个操作之间访问队列。所以为链接队列构建非阻塞算法需要考虑到这两种情况。

我们需要使用几个窍门来完成这项任务。第一个是即使在多步更新中，也要确保数据结构总能处于一致状态。这样，如果线程 *B* 到达时发现线程 *A* 在更新中，*B* 可以分辨操作已部分完成，并且知道不能立即开始自己的更新。那么 *B* 就开始等待（通过反复检查队列状态）直到 *A* 完成更新，这样两个线程就不会相互影响了。

尽管这个方法本身能够使得线程“轮流”访问数据结构，不会造成破坏，但如果一个线程在更新中失败，就没有线程能够再访问队列了。为了使其成为非阻塞算法，我们必须保证一个线程的失败，不会阻止其他线程继续前进。因此，第二个诀窍是，确保如果 *B* 到达时发现数据结构正在被 *A* 修改，在数据结构中应该有足够多的信息，说明需要 *B* 来**替代 *A* 完成更新**。如果 *B* “帮助” *A* 完成其操作，那么 *B* 可以进行自己的操作，而不用等待 *A* 的操作完成。当 *A* 恢复后试图完成其操作，会发现 *B* 已经替它完成了。

清单 15.7 中的 `LinkedQueue` 展示了 Michael-Scott 的非阻塞链接队列算法的插入部分（Michael and Scott, 1996），它已经用在了 `ConcurrentLinkedQueue` 中。在很多队列算法中，一个空队列都有一个“哨兵（sentinel）节点”或者“虚（dummy）节点”，并且队首指针和队尾指针的初始化都指向哨兵节点。队尾指针永远指向哨兵节点（如果队列为空），也就是队列的最后一个元素，或者（当操作正在更新队列时）指向倒数第二个元素。图 15.3 阐释了包含两个元素的队列的正常状态，或者说**稳定状态**。

插入新的元素涉及到两个指针的更新。首先通过更新当前队尾元素的 `next` 指针把新节点链接到列表队尾；然后释放队尾指针，指向新的最末元素。在这两个操作之间，队列

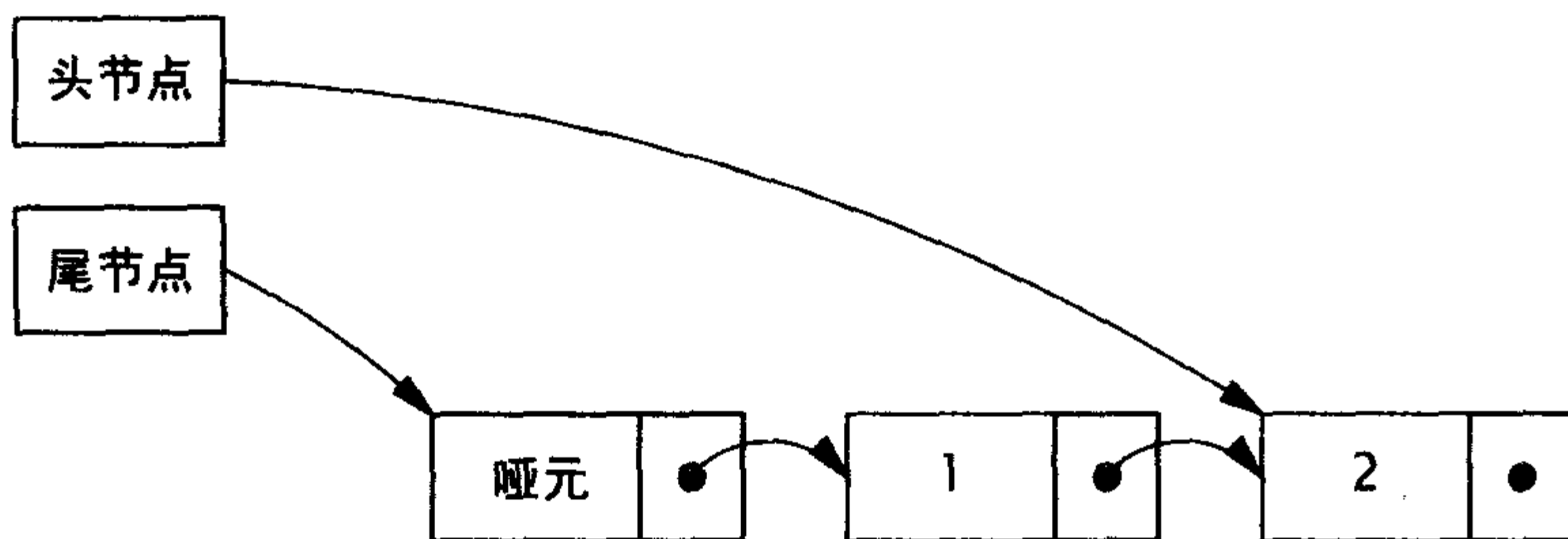


图 15.4 插入期间，队列处于中间的状态

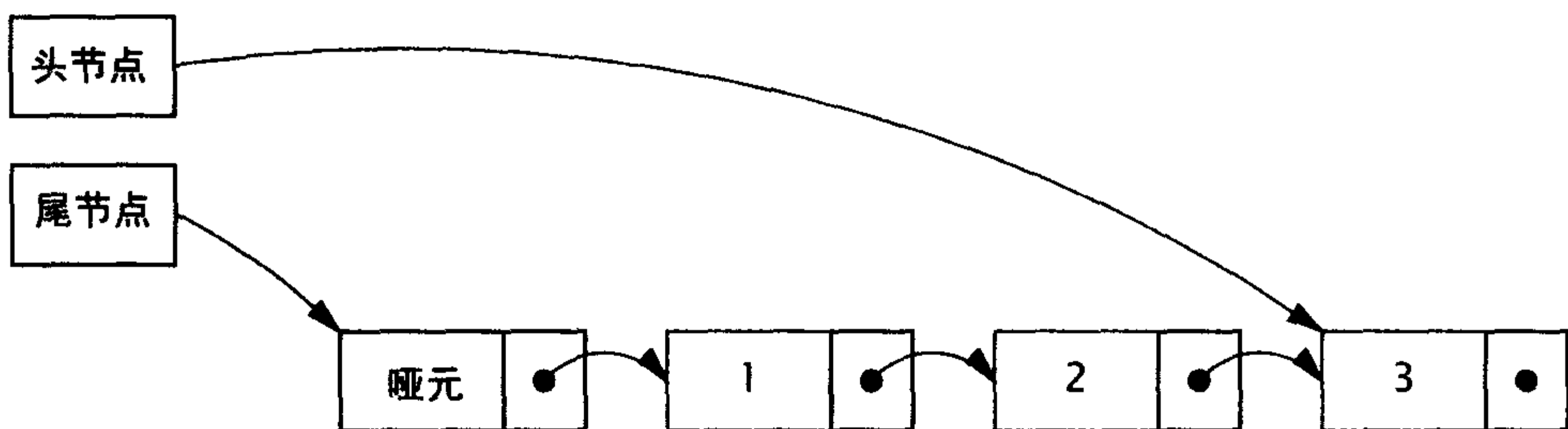


图 15.5 插入完成后，队列再一次回到稳定状态

处于中间状态，如图 15.4。在第二次更新后，队列再一次处于稳定状态，如图 15.5 所示。

要想同时实现两个窍门的方法是：假设队列处于稳定状态，则尾节点的 `next` 域指向 `null`，如果队列处于中间状态，`tail.next` 为非空。所以任何线程都能够通过检查 `tail.next` 即时地了解队列状态。进一步而言，如果队列处于中间状态，它能够通过推进队尾指针向前移动一个节点把状态恢复为稳定状态，结束任意线程正在插入元素的操作⁷。

`LinkedQueue.put` 在尝试插入前首先检查队列是否处于中间状态（步骤 A）。如果是，那么其他线程正在插入元素（在步骤 C 和 D 之间）。除了等待线程完成，当前线程还能够帮助结束操作，推进队尾指针（步骤 B）。在这之后，如果另一个线程已经开始插入新元素了，它会进行重复检查，继续推进队尾指针直到它发现队列处于稳定状态，可以开始自己的插入了。

步骤 C 中的 CAS，在队尾链接了新的节点，如果两个线程试图同时插入元素，会造成失败。在这样的情况下，并不会造成危害：没有发生改变，当前的线程只需要重载队尾指针并重试。C 一旦成功，插入就生效了；第二个 CAS（步骤 D）被用作“清除”，因为

⁷ 关于这个算法正确性的分析说明，参见（Michael and Scott, 1996），或者（Herlihy and Shavit, 2006）。

清单 15.7 Michael-Scott 非阻塞队列算法中的插入 (Michael 与 Scott, 1996)

```

@ThreadSafe
public class LinkedQueue <E> {
    private static class Node <E> {
        final E item;
        final AtomicReference<Node<E>> next;

        public Node(E item, Node<E> next) {
            this.item = item;
            this.next = new AtomicReference<Node<E>>(next);
        }
    }

    private final Node<E> dummy = new Node<E>(null, null);
    private final AtomicReference<Node<E>> head
        = new AtomicReference<Node<E>>(dummy);
    private final AtomicReference<Node<E>> tail
        = new AtomicReference<Node<E>>(dummy);

    public boolean put(E item) {
        Node<E> newNode = new Node<E>(item, null);
        while (true) {
            Node<E> curTail = tail.get();
            Node<E> tailNext = curTail.next.get();
            if (curTail == tail.get()) {
                if (tailNext != null) {
                    // 队列处于静止状态, 推进尾节点
                    tail.compareAndSet(curTail, tailNext);
                } else {
                    // 处于静止状态, 尝试插入新节点
                    if (curTail.next.compareAndSet(null, newNode)) {
                        // 插入成功, 尝试推进尾节点
                        tail.compareAndSet(curTail, newNode);
                        return true;
                    }
                }
            }
        }
    }
}

```

它可以由插入的线程执行，也可以由其他任何线程。如果 *D* 失败，插入线程无论如何都会返回，而不是重新尝试 CAS，因为不再需要重试了——另一个线程已经在步骤 *B* 就完成了这个工作！这样做不会有问题，因为在所有线程尝试向队列链接新节点之前，首先通过检查 `tail.next` 是否为非空来判定队列是否需要清理。如果是，它首先会推进队尾指针（可能多次），直到队列处于稳定状态。

15.4.3 原子化的域更新器

清单 15.7 中阐释了 `ConcurrentLinkedQueue` 的算法，但是真正的实现与它略有区别。`ConcurrentLinkedQueue` 并未使用原子化的引用，而是使用普通的 `volatile` 引用来代表每一个节点，并通过基于反射的 `AtomicReferenceFieldUpdater` 来进行更新，如清单 15.8 所示。

清单 15.8 在 `ConcurrentLinkedQueue` 中使用原子化的域更新器

```
private class Node<E> {
    private final E item;
    private volatile Node<E> next;

    public Node(E item) {
        this.item = item;
    }
}

private static AtomicReferenceFieldUpdater<Node, Node> nextUpdater
    = AtomicReferenceFieldUpdater.newUpdater(
        Node.class, Node.class, "next");
```

原子化的域更新器类（在 `Integer`、`Long`，以及 `Reference` 版本中可用），代表着已存在的 `volatile` 域基于反射的“视图”，使得 CAS 能够用于已有的 `volatile` 域。更新器类没有构造函数；为了创建，你可以调用 `newUpdater` 的工厂方法，声明类和域的名称。域更新器类并不依赖特定的实例；它可以用于更新目标类任何实例的目标域。更新器类提供的原子性保护比普通的原子类差一些，因为你不能保证底层的域不被直接修改——`compareAndSet` 和算术方法只在其他线程使用原子化的域更新器方法时保证其原子性。

在 `ConcurrentLinkedQueue` 中，更新 `Node` 的 `next` 域是通过使用 `nextUpdater` 的 `compareAndSet` 方法实现的。这个有点迂回的方案完全是因性能原因使用的。对于频繁分配的、生命周期短暂的对象，比如队列的链接节点，减少每个 `Node` 的 `AtomicReference` 创建，对于减小插入操作的开销是非常有效的。

然而，几乎在所有的情况下，普通原子变量表现已经相当不错了——仅仅在很少的情况下才下需要使用原子化的域更新器。（当你想要保存现有类的串行化形式时，原子化的域更新器会非常有用。）

15.4.4 ABA 问题

ABA 问题是因为在算法中误用比较并交换而引起的反常现象，节点被循环使用（主要存在于不能被垃圾回收的环境中）。CAS 有效地请求“V 的值仍为 A 么？”，并且如果成立就继续处理更新。在大多数情况下，包括这一章展示的例子，已经足够使用了。但是有时我们还想询问“V 在我上次观察过后发生了改变么？”在某些算法中，把 V 的值由 A 转换为 B，再转换为 A 仍然记为一次改变，需要我们重新进行算法中的某些步骤。

算法中如果进行自身链接节点对象的内存管理，那么可能出现 **ABA 问题**。在这种情况下，即使列表的头仍然指向之前观察到的节点，这也不足以说明列表的内容没有发生改变。如果让垃圾回收器为你管理链表的节点，仍然不能避免 ABA 问题，还有一个相对简单的解决方案：更新一对值，包括引用和版本号，而不是仅更新该值的引用。即使值由 A 改为 B，又再改回 A，版本号也是不同的。AtomicStampedReference（以及它的同系 AtomicMarkableReference）提供了一对变量原子化的条件更新。AtomicStampedReference 更新对象引用的整数对，允许“版本化”引用是能够避免⁸ ABA 问题的。类似地，AtomicMarkableReference 更新一个对象引用的布尔对，它能够用于一些算法，使节点在被标记为“deleted”后仍保留在列表中⁹。

总结

非阻塞算法通过使用低层级并发原语，比如比较并交换，取代了锁。原子变量类向用户提供了这些低层级原语，也能够当作“更佳的 volatile 变量”使用，同时提供了整数类和对象引用的原子化更新操作。

非阻塞算法在设计和实现中很困难，但是在典型条件下能够提供更好的可伸缩性，并能更好地预防活跃度失败。从 JVM 的一个版本到下一个版本间并发性能的提升很大程度上来源于非阻塞算法的使用，包括在 JVM 内部以及平台类库。

⁸ 实践中，无论如何，理论上计数器都应该如此包装。

⁹ 很多处理器提供双重CAS（CAS2或者CASX）操作，可用于操控整形指针对，使得这个操作的效率得到很大提高。在Java 6中，AtomicStampedReference并没有使用双重CAS，即使平台支持这样做。（双向CAS与DCAS不同，控制内存中互不相关的位置；在写作本书时，还没有通用处理器实现DCAS。）

Java 存储模型

The Java Memory Model

纵观全书，我们已经最大程度地避开了 Java 存储模型（JMM）的底层细节，一直在关注高层的设计问题，比如安全发布、规约，以及遵守同步策略等等。它们的安全性得益于 JMM，当你理解了**为什么**这些机制会如此工作后，能发现可以更容易有效地使用它们。本章中，我们会揭开 Java 存储模型的神秘面纱，看一看它的底层条件和它提供的保证，其间还会涉及一些本书讲过的高层设计原则背后的原理。

16.1 什么是存储模型，要它何用

假设一个线程为变量 `aVariable` 赋值：

```
aVariable = 3;
```

存储模型要回答这个问题：“在什么条件下，读取 `aVariable` 的线程会看到 3 这个值？”这听起来像是个愚蠢的问题，但是如果缺少了同步，就会有很多因素会导致线程可能无法立即——甚至永远——看到另一个线程的操作所产生的结果。编译器生成指令的次序，可以不同于源代码所暗示的“显然”的版本，而且编译器还会把变量存储在寄存器，而不是内存中；处理器可以乱序或者并行地执行指令；缓存会改变写入提交到主内存的变量的次序；最后，存储在处理器本地缓存中的值，对于其他处理器并不可见。这些因素都会妨碍一个线程看到一个变量的最新值，而且会引起内存活动在不同的线程中表现出不同的发生次序——如果你没有适当同步的话。

在单线程环境中，这些伎俩的发生，对于我们来说是隐藏的，它除了能提高程序执行速度外，不会产生其他的影响。Java 语言规范规定了 JVM 要维护**内部线程类似顺序化语义**（within-thread as-if-serial semantics）：只要程序的最终结果等同于它在严格的顺序化环境中执行的结果，那么上述所有的行为都是允许的。这倒也是件好事情，因为在最近几年

中，重新排序后的指令使得程序在计算性能上得到了很大的提升。当然了，对性能的提升作出贡献的，除了越来越高的时钟频率，还有不断提升的并行性——管道超标量体系结构执行单元，动态指令调度，试探性执行以及成熟的多级存储缓存。随着处理变得越来越成熟，编译器也在不断地改进。重新排序后的指令，对于优化执行以及成熟的全局寄存器分配算法的使用，都是大有裨益的。处理器的制造商纷纷转向生产多核（multicore）处理器，很大的原因在于，时钟频率正在变得难以经济地获得提高，可以提升的只有硬件并行性。

在多线程的环境中，为维护正确的顺序性不得不产生很大的性能开销。因为在大部分时间里，同步的应用程序中的线程都在做属于自己的工作，额外的线程间协调只会降低程序的运行效率，不会带来任何好处。只有当多个线程要共享数据时，才必须协调它们的活动；协调是通过使用同步完成的，JVM 依赖于程序明确地指出何时需要协调线程的活动。

JMM 规定了 JVM 的一种最小保证：什么时候写入一个变量会对其他线程可见。这样的设计，可以在对可预言性的需要和开发程序的简易性之间取得平衡。程序员会在高性能的 JVM 实现上开发程序，这些 JVM 广泛地覆盖了当今流行的处理器体系架构。现代的处理器和编译器为了从你的程序中榨取性能，会用尽手段。如果不了解这些话，你在刚刚接触 JMM 的某些方面时会十分困惑。

16.1.1 平台的存储模型

在可共享内存的多处理器体系架构中，每个处理器都有它自己的缓存，并且周期性地与主内存协调一致。处理器架构提供了不同级别的**缓存一致性**（cache coherence）；有些只提供最小的保证，几乎在任何时间内，都允许不同的处理器在相同的存储位置上看到不同的值。无论是操作系统、编译器、运行时（有时甚至包括应用程序），都要将就这些硬件与线程安全需求之间的不同。

想要保证每个处理器能在任意时间内获知其他处理器正在进行的工作，其代价非常高昂。大多数时间里这些信息都是没用的，所以处理器会牺牲存储一致性的保证，来换取性能的提升。一种架构的**存储模型**告诉了应用程序可以从它的存储系统中获得何种担保，同时详细定义了一些特殊的指令称为**存储关卡**（memory barriers）或**栅栏**（fences），用以在需要共享数据时，得到额外的存储协调保证。为了帮助 Java 开发者屏蔽这些跨架构的存储模型之间的不同，Java 提供了自己的存储模型，JVM 会通过适当的位置上插入存储关卡，来解决 JMM 与底层平台存储模型之间的差异化。

关于程序执行，一个简约的理想模型（mental model）是去想象：操作执行的顺序是唯一的，那就是它们出现在程序中的顺序，这与执行它们的处理器无关；另外，变量每一次读操作，都能得到执行序列上这个变量最新的写入值，无论这个值是哪个处理器写入的。如果这件不切实际的事情在某个模型上出现，那么那个模型叫作**顺序化一致性**模型。软件

开发者经常错误地假设顺序化一致性的存在，但是没有哪个现代的多处理器架构会提供这一点，JMM 也如此。冯·诺伊曼模型，这个经典的顺序化计算模型，仅仅是现代多处理行为的模糊近似而已。

最后的结论是，跨线程共享数据时，现代可共享内存的多处理器（和编译器）架构会做出一些令人惊讶的事情来；除非你已经使用存储关卡，通知它们不要这样做。幸运的是，不需要在 Java 程序中指明存储关卡的放置位置，只要在访问共享状态时能够识别到它们就可以了。通过正确地使用同步，可以做到这些。

16.1.2 重排序

在第 2 章描述竞争条件和原子性失败时，我们用到过交互图，说明调度器交互调用存取操作的“偶发时序”，会在没有正确同步的程序中引发错误的结果。更糟的是，JMM 还允许从不同的角度观看一个动作，此时该动作会以不同的次序执行。这使得在没有同步的情况下，推断动作的执行次序变得更加复杂。各种能够引起操作延迟或者错续执行的不同原因，都可以归结为一类**重排序**（reordering）。

清单 16.1 的 PossibleReordering 阐释了在没有正确同步的情况下，即使是最简单的并发程序，推断它的行为也是很困难的。可以很简单地想到 PossibleReordering 是如何打印出 (1, 0) 或 (0, 1) 或 (1, 1) 的：线程 A 可以在 B 开始前完成，B 也可以在 A 开始前完成，或者它们的动作可以交替进行。但奇怪的是，PossibleReordering 竟然还可以打印 (0, 0)！由于每个线程中的动作都没有依赖其他线程的数据流，因此这些动作可以乱序执行。（即使按照次序执行，缓存刷新至主内存的时序也会导致这种现象的出现，从线程 B 的角度看，赋值操作可能在 A 中以相反的次序发生。）图 16.1 显示了一种可能的交替操作，这种重排序会打印 (0, 0)。

PossibleReordering 只是一个极为简单的程序，但是罗列出它所有可能的结果后仍然令人惊讶。内存级的重排序会让程序的行为变得不可预期。没有同步，推断执行次序的难度令人望而却步；只需要确保你的程序已正确同步，事情就会变得简单些。同步抑制了编译器、运行时和硬件对存储操作的各种方式的重排序，否则这些重排序将会破坏 JMM 提供的可见性保证¹。

16.1.3 Java 存储模型的简介

Java 存储模型的定义是通过**动作**（actions）的形式进行描述的，所谓动作，包括变量的读和写、监视器加锁和释放锁、线程的启动和拼接（join）。

¹ 在大多数流行的处理器架构中，存储模型都足够强大，以应付读取 volatile 变量与非 volatile 变量之间的性能差异。

清单 16.1 没有充分同步的程序可以产生令人惊讶的结果（不要这样做）

```

public class PossibleReordering {
    static int x = 0, y = 0;
    static int a = 0, b = 0;
    /
    public static void main(String[] args)
        throws InterruptedException {
        Thread one = new Thread(new Runnable() {
            public void run() {
                a = 1;
                x = b;
            }
        });
        Thread other = new Thread(new Runnable() {
            public void run() {
                b = 1;
                y = a;
            }
        });
        one.start(); other.start();
        one.join(); other.join();
        System.out.println("(" + x + "," + y + ")");
    }
}

```

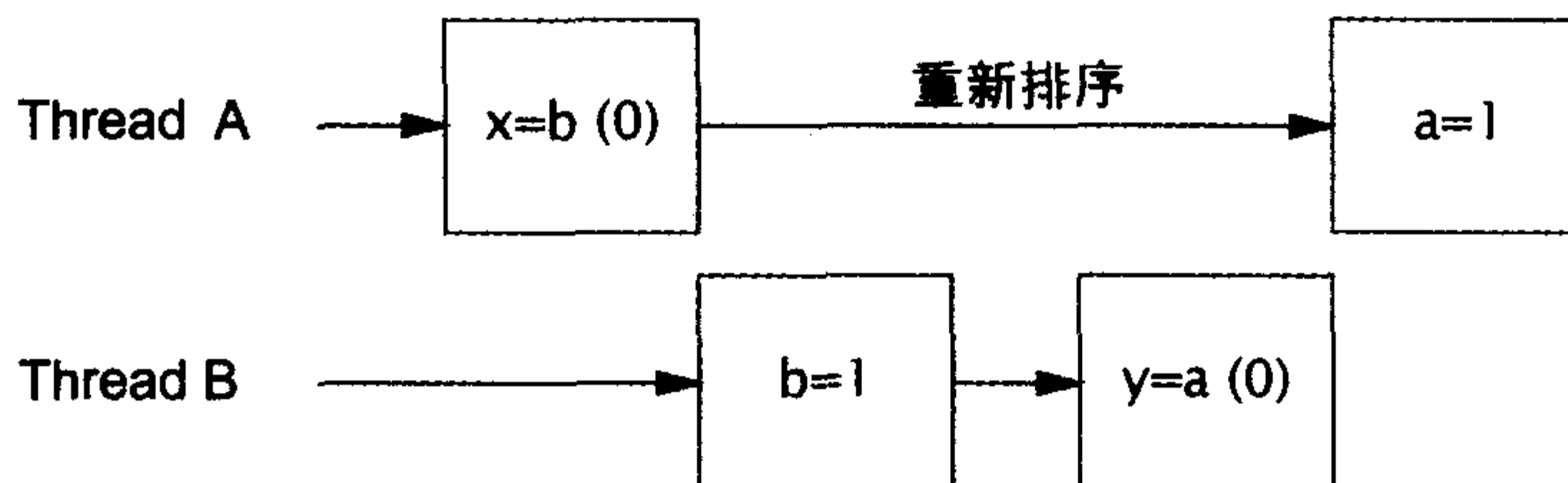


图 16.1 PossibleReordering 中显示重排序的交替操作

JMM 为所有程序内部的动作定义了一个偏序关系²，叫作 *happens-before*。要想保证执行动作 *B* 的线程看到动作 *A* 的结果（无论 *A* 和 *B* 是否发生在同一个线程中），*A* 和 *B* 之间就必须满足 *happens-before* 关系。如果两个操作之间并未依照 *happens-before* 关系排序，JVM 可以对它们随意地重排序。

² 偏序关系 $<$ 是集合上的一种反对称的、自反的和传递的关系，不过并不是任意两个元素 x, y 都必须满足 $x < y$ 或者 $y < x$ 。我们每天都在应用偏序关系来表达我们的喜好；我们可以喜欢寿司胜过干酪三明治，可以喜欢莫扎特胜过马勒（Mahler），但是我们不必在干酪三明治和莫扎特之间作出一个明确的喜好选择。

当一个变量被多个线程读取，且至少被一个线程写入时，如果读写操作并未依照 *happens-before* 排序，就会产生**数据竞争**（data race）。一个**正确同步的程序**（correctly synchronized program）是没有数据竞争的程序；正确同步的程序会表现出顺序的一致性，这就是说所有程序内部的动作会以固定的、全局的顺序发生。

happens-before 的法则包括：

程序次序法则：线程中的每个动作 *A* 都 *happens-before* 于该线程中的每一个动作 *B*，其中，在程序中，所有的动作 *B* 都出现在动作 *A* 之后。

监视器锁法则：对一个监视器锁的解锁 *happens-before* 于每一个后续对同一监视器锁的加锁³。

volatile 变量法则：对 volatile 域的写入操作 *happens-before* 于每一个后续对同一域的读操作⁴。

线程启动法则：在一个线程里，对 Thread.start 的调用会 *happens-before* 于每一个启动线程中的动作。

线程终结法则：线程中的任何动作都 *happens-before* 于其他线程检测到这个线程已经终结、或者从 Thread.join 调用中成功返回，或者 Thread.isAlive 返回 false。

中断法则：一个线程调用另一个线程的 interrupt *happens-before* 于被中断的线程发现中断（通过抛出 InterruptedException，或者调用 isInterrupted 和 interrupted）。

终结法则：一个对象的构造函数的结束 *happens-before* 于这个对象 finalizer 的开始。

传递性：如果 *A happens-before* 于 *B*，且 *B happens-before* 于 *C*，则 *A happens-before* 于 *C*。

虽然动作仅仅需要满足偏序关系，但是同步动作——锁的获取与释放，以及 volatile 变量的读取与写入——却是满足全序关系（译注：当偏序集中的任意两个元素都可比时，称该偏序集满足全序关系。）的。这样，在描述 *happens-before* 关系时，就可以使用“后续”锁的获取和 volatile 变量的读取这种形式了。

图 16.2 演示了两个线程同步使用一个公共锁时，它们之间的 *happens-before* 关系。线程 A 内部的所有动作都是依照“程序次序法则”进行排序的。线程 B 的内部动作也一样。

³ 显式锁的加锁和解锁有着与内置锁相同的存储语意。

⁴ 原子变量的读写操作有着与 volatile 变量相同的语意。

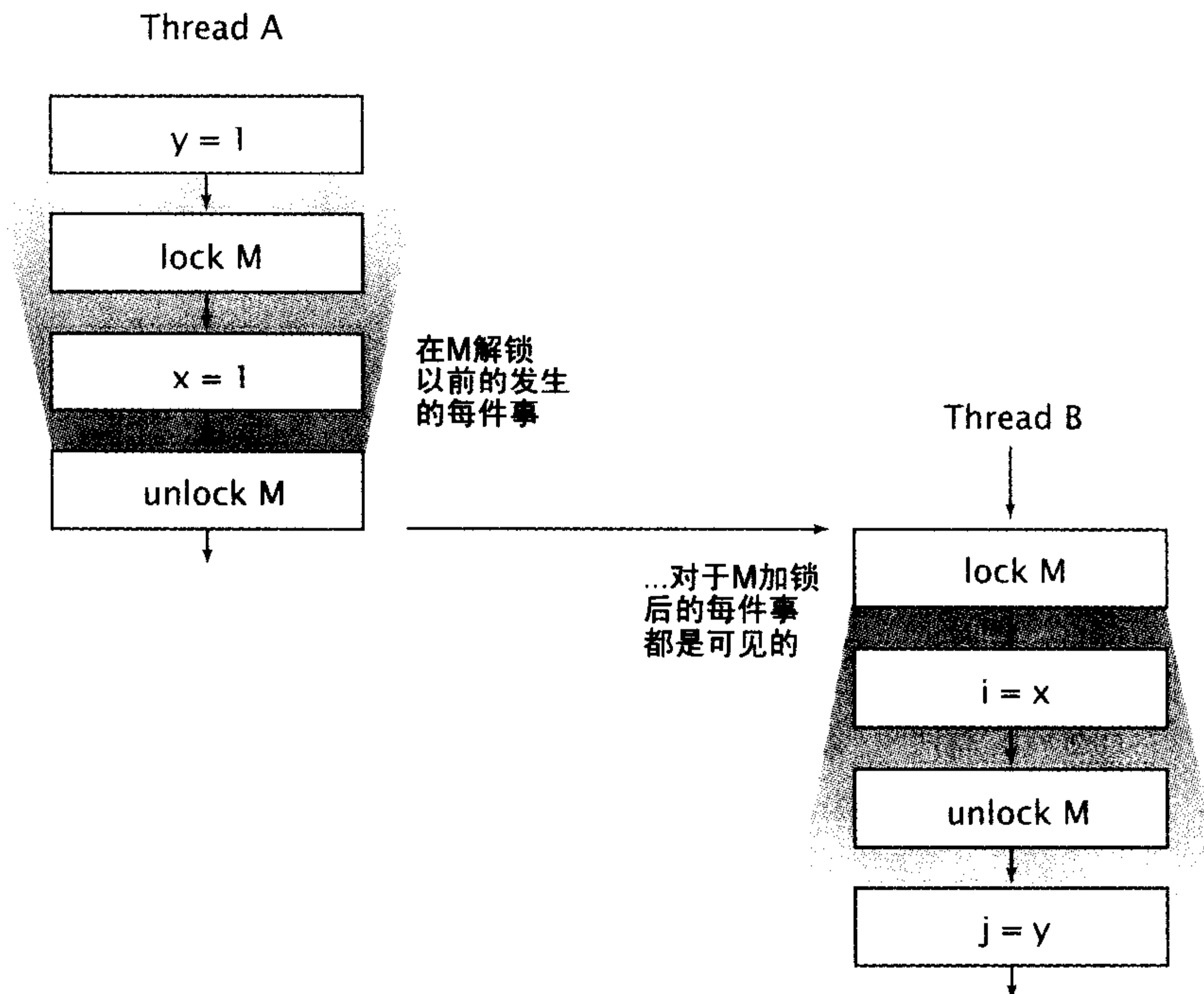


图 16.2 Java 存储模型中 *happens-before* 的图例

因为 A 释放了锁 *M*, B 随后获得了锁 *M*, A 中的所有释放锁之前的动作, 也就因此排到了 B 中请求到锁后动作的前面。如果两个线程是在不同的锁上进行同步的, 我们就不能对它们之间的动作如何排序做任何断言——两个线程的动作之间并不存在 *happens-before* 关系。

16.1.4 “驾驭”在同步之上

得益于 *happens-before* 顺序的强大, 有时你可以直接“驾驭”在现有同步提供的可见性属性之上。为了满足 *happens-before*, 这需要结合“程序次序法则”与另外一种次序法则（通常是“监视器锁法则”或“volatile 变量法则”）来对访问变量的操作进行排序, 否则就用锁来保护它。这种技术由于对语句出现的顺序是敏感的, 因此很容易出错; 作为一项高端的技术, 它应该被留作用来榨取性能关键类（performance-critical class, 比如 `ReentrantLock`）的最后一滴性能。

在 `FutureTask` 中, `AbstractQueuedSynchronizer` 的受保护方法的实现阐释了何为“驾驭”。AQS 维护了一个整数, 代表 `Synchronizer` 的状态。`FutureTask` 使用它存储

任务的状态：运行中，完成和取消。FutureTask 同时还维护着其他的变量，比如计算的结果。当一个线程调用 set 保存结果，而另一个线程调用 get 重获结果，这两个线程已经按照 *happens-before* 很好地进行了排序。将结果的引用声明为 volatile 类型就可以做到这一点。但是也可以通过采用现有的同步，以较低开销取得相同的结果。

FutureTask 是经过精心设计的，它确保对 tryReleaseShared 的成功调用总是 *happens-before* 于后续的对 tryAcquireShared 的调用；tryReleaseShared 总会写入一个 volatile 变量，tryAcquireShared 会读取该变量。清单 16.2 中的 innerSet 和 innerGet 方法会在保存和重获 result 时被调用到；因此，innerSet 写入 result 的操作，会发生在 releaseShared 的调用之前（它会调用 tryReleaseShared），同样 innerGet 读取 result 的操作，会发生在 acquireShared 的调用之后（它会调用 tryAcquireShared），程序次序法则与 volatile 变量法则相结合，确保了在 innerSet 中写入 result 能够 *happens-before* 于在 innerGet 中读取 result。

清单 16.2 FutureTask 的内部类示范了如何“驾驭”同步

```
// FutureTask 的内部类
private final class Sync extends AbstractQueuedSynchronizer {
    private static final int RUNNING = 1, RAN = 2, CANCELLED = 4;
    private V result;
    private Exception exception;

    void innerSet(V v) {
        while (true) {
            int s = getState();
            if (ranOrCancelled(s))
                return;
            if (compareAndSetState(s, RAN))
                break;
        }
        result = v;
        releaseShared(0);
        done();
    }

    V innerGet() throws InterruptedException, ExecutionException {
        acquireSharedInterruptibly(0);
        if (getState() == CANCELLED)
            throw new CancellationException();
        if (exception != null)
            throw new ExecutionException(exception);
        return result;
    }
}
```



这种技术用到了一个已有的 *happens-before* 排序，这个排序是因为其他原因创建的，而不是专门创建用来发布 X 的。因此我们把这种技术称为“驾驭（同步）”。

在 `FutureTask` 中用到的“驾驭”技术是相当容易出错的，不应该武断地拿来就用。但是，在有些情况下，使用“驾驭”技术是相当合情合理的。比如，当一个类的规约中规定，它必须把 *happens-before* 融入到方法之间。例如，使用 `BlockingQueue` 进行安全发布就是“驾驭”的一种方式。一个线程将对象置入队列，另一个线程随后获取它，这就构成了安全发布。这是因为，`BlockingQueue` 的实现可以保证有足够的内部同步，它能确保进队 *happens-before* 于出队。

由类库担保的其他 *happens-before* 排序包括：

- 将一个条目置入线程安全容器 *happens-before* 于另一个线程从容器中获取条目。
- 执行 `CountDownLatch` 中的倒计时 *happens-before* 于线程从闭锁 (latch) 的 `await` 中返回。
- 释放一个许可给 `Semaphore` *happens-before* 于从同一 `Semaphore` 里获得一个许可。
- `Future` 表现的任务所发生的动作 *happens-before* 于另一个线程成功地从 `Future.get` 中返回。
- 向 `Executor` 提交一个 `Runnable` 或 `Callable` *happens-before* 于开始执行任务。
- 最后，一个线程到达 `CyclicBarrier` 或 `Exchanger` *happens-before* 于相同关卡 (barrier) 或 `Exchange` 点中的其他线程被释放。如果 `CyclicBarrier` 使用一个关卡 (barrier) 动作，到达关卡 *happens-before* 于关卡动作，依照次序，关卡动作 *happens-before* 于线程从关卡中释放。

16.2 发布

第 3 章我们探究了一个对象是如何被正确或不正确地发布出去的。当时描述的安全发布技术之所以是安全的，正是得益于 JMM 提供的保证；而为不正确发布带来风险的真正原因，是在“发布共享对象”与从“另一个线程访问它”之间，缺少 *happens-before* 排序。

16.2.1 不安全的发布

在缺少 *happens-before* 关系的情况下，存在重排序的可能性。这就解释了为什么如果在没有充分同步的情况下就发布一个对象，会导致另外的线程看到一个**部分创建对象**（参见 3.5 节）。新对象的初始化涉及到写入变量——新对象的域。类似地，引用的发布涉及

到写入另一个变量——新对象的引用。如果你不能保证共享引用 *happens-before* 于另外的线程加载这个共享引用，那么写入新对象的引用与写入对象域（从消费该对象的线程的角度看）可以被重排序。在这种情况下，另一个线程可以看到对象引用的最新值，**但也看到一些或全部对象状态的过期值**——一个部分创建对象。

错误的情性初始化会导致不正确的发布，正如清单 16.3 所示。初看起来，程序中仅有的问题似乎是 2.2.2 节描述的竞争条件。在某些特定的条件下，比如当 `Resource` 的所有实例都是一样的（以及可能多次创建 `Resource` 实例时带来的低效率），你可能并不介意竞争条件问题。不幸的是，即使忽略掉这些瑕疵，另外的线程还是可以观察到一个部分构建的 `Resource` 引用，所以 `UnsafeLazyInitialization` 仍然不安全。

清单 16.3 不安全的情性初始化（不要这样做）

```
@NotThreadSafe
public class UnsafeLazyInitialization {
    private static Resource resource;

    public static Resource getInstance() {
        if (resource == null)
            resource = new Resource(); // 不安全的发布
        return resource;
    }
}
```



假设第一次调用 `getInstance` 的是线程 *A*。它会看到 `resource` 是 `null`，接着初始化一个新的 `Resource`，然后设置 `resource` 引用这个新实例。随后线程 *B* 调用 `getInstance`，它可能看到 `resource` 已经有了一个非空的值，于是就使用这个已经创建的 `Resource`。这最初看起来可能无害，但是线程 *A* 写入 `resource` 与线程 *B* 读取 `resource` 之间，不是按照 *happens-before* 进行排序的。发布对象时存在数据竞争，因此 *B* 并不能保证可以看到 `Resource` 的正确状态。

在新分配一个 `Resource` 时，`Resource` 的构造函数会把实例的域从（由 `Object` 的构造函数写入的）默认值转变到它们的初始值。由于两个线程都没有用到同步，因此 *B* 看到的 *A* 的动作，可能与 *A* 执行它们时的顺序并不相同。所以即使 *A* 是先初始化 `Resource`，再设置 `resource` 引用它的，*B* 也可以看到写入 `resource` 的动作会先于写入 `Resource` 域的发生。*B* 可能因此看到一个部分构建的 `Resource`，它可能处于非法的状态——在后期，它的状态可能出现无法预料的改变。

除了不可变对象以外，使用被另一个线程初始化的对象，是不安全的，除非对象的发布是 *happens-before* 于对象的消费线程使用它。

16.2.2 安全发布

第 3 章描述的安全发布的常用模式，可以确保发布的对象对于其他线程是可见的，因为它们保证发布对象是 *happens-before* 于消费线程加载已发布对象的引用。如果线程 *A* 将 *X* 置入 `BlockingQueue`（并且随后没有线程修改它），线程 *B* 从队列中获取 *X*，可以保证 *B* 看到的 *X* 就是 *A* 留下的那个。这是因为在 `BlockingQueue` 的实现中，*B* 进行了充足的内部同步，确保了 `put` *happens-before* 于 `take`。类似地，使用锁保护共享变量，或者使用共享的 `volatile` 类型变量，也可以保证对该变量的读取和写入是按照 *happens-before* 排序的。

happens-before 事实上可以比安全发布承诺更强的可见性与排序性。如果 *X* 从 *A* 到 *B* 是安全发布的，安全发布可以保证 *X* 的状态的可见性，但不包括 *A* 可能触及到的其他变量的状态。但是，如果 *A* 将 *X* 置入队列 *happens-before* 于 *B* 从队列中获取 *X*。*B* 不仅仅能看到 *X* 的状态就是 *A* 留下的（假设 *A* 或者其他线程没有对 *X* 再做修改），而且 *B* 还能看到 *A* 移交 *X* 前所作的每件事情（我们又遇到了同样的警告）⁵。

既然 JMM 已经提供了更为强大的 *happens-before*，我们为什么还要强烈关注 `@GuardedBy` 和安全发布呢？因为从移交对象所有权和发布的角度考虑问题，相比于从独立的内存写入的可见性的角度考虑问题，能更好地适合大多数的程序设计。*happens-before* 排序操作就是处于独立内存访问级别的操作——它是一种“并发汇编语言”。安全发布的操作更加贴近你的程序设计。

16.2.3 安全初始化技巧

有时候，如果对象的初始化很昂贵，那么将它们的初始化延迟到真正需要它们的时刻，是很有意义的。不过我们已经看到一些关于惰性初始化的误用是如何引起麻烦的。可以像清单 16.4 那样修改 `UnsafeLazyInitialization`，将 `getResource` 方法声明为 `synchronized`。贯穿 `getInstance` 的代码路径相当简短（一个检查和一个预测分支），所以如果 `getInstance` 不会被多个线程频繁调用的话，人们也不会争论 `SafeLazyInitialization` 加锁的方法是否能提供令人满意的性能。

初始阶段对待静态域（在静态初始块中初始化的域 [PL 2.2.1 and 2.5.3]）的方式，

⁵ JMM 保证 *B* 至少可以看到 *A* 写入的最新值，后续的写入可能也可能不会被看见。

清单 16.4 线程安全的情性初始化

```
@ThreadSafe
public class SafeLazyInitialization {
    private static Resource resource;

    public synchronized static Resource getInstance() {
        if (resource == null)
            resource = new Resource();
        return resource;
    }
}
```

略微有些独特，它提供了额外的线程安全性保证。静态初始化是 JVM 完成的，发生在类的初始阶段，即类被加载后到类被任意线程使用之前。JVM 要在初始化期间获得一个锁 [JLS 12.4.2]，这个锁每个线程都至少会用到一次，来确保一个类是否已被加载；这个锁也保证了静态初始化期间，内存写入的结果自动地对所有线程是可见的。所以静态初始化的对象，无论是构造期间还是被引用的时候，都不需要显式地进行同步。然而，这仅仅适用于**构造当时**（as-constructed）的状态——如果对象是可变的，为了保证后续修改的可见性，避免数据变脏，读者和写者仍然需要同步。

清单 16.5 主动初始化

```
@ThreadSafe
public class EagerInitialization {
    private static Resource resource = new Resource();

    public static Resource getResource() { return resource; }
}
```

像清单 16.5 那样，使用主动的初始化，避免了每次调用 SafeLazyInitialization 的 getInstance 时引发的同步开销。这项技术可以和 JVM 的情性类加载相结合，创建一种情性初始化技术，使得在通常的代码路径中都不需要同步。清单 16.6 的“**情性初始化 holder 类技巧**”使用一个专门用来初始化 Resource 的类。JVM 将 ResourceHolder 的初始化被延迟到真正使用它的时刻 [JLS 12.4.1]。因为 Resource 是在静态初始阶段进行初始化的，所以不再需要额外的同步。线程第一次调用 getResource，引起 ResourceHolder 的加载和初始化，这个时候，正是在静态初始阶段 Resource 完成初始化发生的时间。

清单 16.6 惰性初始化 holder 类技巧

```
@ThreadSafe
public class ResourceFactory {
    private static class ResourceHolder {
        public static Resource resource = new Resource();
    }

    public static Resource getResource() {
        return ResourceHolder.resource ;
    }
}
```

16.2.4 双检查锁 (double-checked locking)

没有哪本关于并发的书会对双检查锁 (DCL) 反模式只字不提。请看清单 16.7。在很早以前的 JVM 中, 同步, 甚至是无竞争的同步, 都存在惊人的性能开销。结果是, 很多聪明的 (或者至少看上去聪明) 的小技巧被发明出来——有的好, 有的坏, 有的丑——以降低同步的影响。DCL 属于“丑”的那一类。

因为早期 JVM 的性能问题又一次遗留下很多事情需要考虑到。而惰性初始化常被用来避免潜在必要的昂贵操作, 降低程序启动时间。正确编写的惰性初始化方法需要同步。但是, 同步不仅很慢, 更重要的是, 同步的含义还没有完全被理解: “独占性”方面很好地被理解了, 但是“可见性”方面还没有。

DCL 声称是“鱼与熊掌可兼得”的最佳典范——惰性初始化在通常的代码路径下, 不需要在同步上花费时间。它运作的方式是, 首先检查在没有同步的情况下检查是否需要初始化, 如果 resource 不等于 null, 就用它。否则, 就进行同步, 并再次检查 Resource 是否需要同步, 以保证只有唯一的线程真正地初始化了共享的 Resource。通常的代码路径——获取一个已经构建的 Resource 的引用——并没有用到同步。这就是问题所在: 就像 16.2.1 节中描述的, 线程可能看到一个部分创建的 Resource。

DCL 的真正问题在于, 它是基于这样的假设的: 当没有使用同步时读取一个共享变量, 可能发生的最坏的事情不过是错误地看到过期值 (具体而言是 null); 这种情况下, DCL 方法通过占有锁后再检查一次, 希望这样能够避免风险。但是, 最坏的情况事实上比这还糟糕——线程可能看到引用的当前值, 但是对象的状态值却是过期的。这意味着对象可以被观察到, 但却处于无效或错误的状态。

JMM 后续的修订 (Java 5.0 或更高) 要做到: 如果把 resource 声明为 volatile 类型, DCL 就能正确地工作, 同时由于读取 volatile 变量通常比读取非 volatile 变量的性能开销只有轻微的增长, 所以这种方法的性能开销也很低。

清单 16.7 双检查锁反模式（不要这样做）

```
@NotThreadSafe
public class DoubleCheckedLocking {
    private static Resource resource;

    public static Resource getInstance() {
        if (resource == null) {
            synchronized (DoubleCheckedLocking.class) {
                if (resource == null)
                    resource = new Resource();
            }
        }
        return resource;
    }
}
```



无论如何，DCL 这个技巧的使用已经被广泛地废弃了——催生它的动力（缓慢的无竞争同步和缓慢的 JVM 启动）已经不复存在，这使得优化的效果越来越不明显了。惰性初始化容器（lazy initialization holder）的模式提供了同样的好处，而且更易理解。

16.3 初始化安全性

保证了**初始化安全性**，就可以让正确创建的**不可变**对象在没有同步的情况下，可以被安全地跨线程地共享，而不管它们是如何发布的——甚至发布时也存在数据竞争。（这就是说，**如果** Resource 是不可变的，UnsafeLazyInitialization 事实上也可以是安全的。）

假若没有初始化安全性，就会发生这样的事情：像 String 这样的不可变对象，没有发布或消费线程中用到同步，可能表现出它们的值被改变。安全的（系统）架构依赖于 String 的不可变性；缺少了初始化安全性，可能会形成一个安全漏洞，让恶意代码绕过安全检查。

初始化安全可以保证，对于正确创建的对象，无论它是如何发布的，所有线程都将看到构造函数设置的 final 域的值。更进一步，一个正确创建的对象中，任何可以通过其 final 域触及到的变量（比如一个 final 数组中的元素，或者一个 final 域引用的 HashMap 里面的内容），也可以保证对其他线程都是可见的⁶。

⁶ 这只被用于下述的对象：它们只有通过正在被构建的对象的 final 域，才能触及到。

对于含有 `final` 域的对象，初始化安全性可以抑制重排序，否则这些重排序会发生在对象的构造期间以及内部加载对象引用的时刻。所有构造函数要写入值的 `final` 域，以及任何通过这些域可以到达任何变量，都会在构造函数完成后被“冻结”，而且可以保证任何获得该引用的线程，至少可以看到和冻结值一样新的值。用于向通过 `final` 域可到达的初始变量写入值的操作，不会和构造后的操作一起被重排序。

初始化安全性意味着，像清单 16.8 的 `SafeStates`，即使存在着不安全的惰性初始化，或者在没有同步的公共静态域中隐藏 `SafeStates` 的引用，即使它没有使用同步，而且依赖于非线程安全的 `HashSet`，它都可以被安全地发布。

清单 16.8 不可变对象的初始化安全性

```
@ThreadSafe
public class SafeStates {
    private final Map<String, String> states;

    public SafeStates() {
        states = new HashMap<String, String>();
        states.put("alaska", "AK");
        states.put("alabama", "AL");
        ...
        states.put("wyoming", "WY");
    }

    public String getAbbreviation(String s) {
        return states.get(s);
    }
}
```

但是，有很多对 `SafeStates` 细小的修改都可能破坏它的线程安全性。如果 `state` 不是 `final` 类型的，或者如果任何构造函数以外的方法修改 `state` 的内容，初始化安全性还不足以在缺少同步的情况下保证安全地访问 `SafeStates`。如果 `SafeStates` 还有其他的非 `final` 域，其他线程仍然可能看到这些域上的不正确的值。这也放任对象会在构造期间逸出，让初始化安全性的保证成为空谈。

初始化安全性保证只有以通过 `final` 域触及的值，在构造函数完成时才是可见的。对于通过非 `final` 域触及的值，或者创建完成后可能改变的值，必须使用同步来确保可见性。

总结

Java 存储模型明确地规定了在什么时机下，操作存储器的线程的动作可以保证被另外的动作看到。规范还规定了要保证操作是按照一种偏序关系进行排序。这种关系称为 *happens-before*，它是规定在独立存储器和同步操作的级别之上的。如果缺少充足的同步，线程在访问共享数据时就会发生非常无法预期的事情。然而，使用第 2 章与第 3 章中提到的高层规则，比如 `@GuardedBy` 和安全发布，可以在不考虑 *happens-before* 的底层细节的情况下，也能确保线程安全性。

同步 Annotation

Annotation for Concurrency

我们已经使用@GuardedBy 和@ThreadSafe 等 Annotation，来展现如何将线程安全性的保证和同步策略进行文档化。本附录记录了这些 Annotation；你可以从本书的网站上下载它们的源代码。（当然，还有更多应该被文档化的线程安全性保证以及实现细节，并没有包含在这个 Annotation 的最小集合里。）

A.1 类 Annotation

我们用到了 3 个类级 Annotation 来描述类的可预期的线程安全性保证：@Immutable、@ThreadSafe 和@NotThreadSafe。@Immutable 自然是意味着类是不可变的，并包含了@ThreadSafe 的意义。@NotThreadSafe 是可选的——如果类没有被标明是线程安全的，就无法肯定它是不是线程安全的，但是如果你想明确地表示出它不是线程安全的，就标注为@NotThreadSafe。

这些 Annotation 相对是非侵入的，这对用户和维护者都是有益的。用户可以立即看出一个类是否是线程安全的，维护者也可以直接检查类是否遵守了线程安全性保证。Annotation 对于第三个利益既得者也是有用的：工具。静态的代码分析工具可以有能力对代码进行验证，看它是否遵守了由 Annotation 指定的契约，比如标明为@Immutable 的类是否真是不可变的。

A.2 域 Annotation 和方法 Annotation

上面的类级 Annotation 只是类的公共文档的一部分。类的线程安全性策略的其他方面，对于维护者来说是完全有必要的，不过并没有作为公共文档的一部分。

使用加锁的类应该写入文档的信息包括：哪个状态变量被哪个锁保护着，以及哪个锁用来保护这些变量的信息。有时，一个线程安全类的状态始终由锁保护着，但是后来发生的修改，比如可能添加新的变量而没有充分地被锁保护，也可能添加一个新方法，却没有正确地使用锁来保护现有的状态变量。这些都是常见的可能在无意中导致非线程安全性的源头。把哪个锁保护哪个变量的信息文档化，有利于避免所有这些疏忽。

- `@GuardedBy(lock)` 记录的是：线程只有在持有了一个特定的锁后，才能访问某个域或方法。`lock` 参数代表一个锁，只有持有了该锁，才能访问被标注的域或方法。`lock` 的可能值如下：
- `@GuardedBy("this")`，是指包含在对象中的内部锁（方法或域是这个对象的一个成员）；
- `@GuardedBy("fieldName")`，是指与 `fieldName` 引用的对象相关联的锁，它或者是一个隐式锁（`fieldName` 没有引用一个 `Lock`），或者是一个显式锁（`fieldName` 引用了一个 `Lock`）；
- `@GuardedBy("ClassName.fieldName")`，类似于 `@GuardedBy("fieldName")`，不过所引用的锁对象是存储在另一个类中的静态域；
- `@GuardedBy("methodName()")`，是指锁对象是 `methodName()` 方法的返回值；
- `@GuardedBy("ClassName.class")`，是指 `ClassName` 类的直接量对象。

使用 `@GuardedBy` 标识出每一个需要加锁的状态变量，这个锁确保它会有助于维护和代码审查，而且能够帮助自动化分析工具发现潜在的线程安全性错误。

参考文献

Bibliography

Ken Arnold, James Gosling, and David Holmes. The Java Programming Language, Fourth Edition. Addison–Wesley, 2005.

David F. Bacon, Ravi B. Konuru, ChetMurthy, andMauricio J. Serrano. Thin Locks: Featherweight Synchronization for Java. In SIGPLAN Conference on Programming Language Design and Implementation, pages 258–268, 1998. URL <http://citeseer.ist.psu.edu/bacon98thin.html>.

Joshua Bloch. Effective Java Programming Language Guide. Addison–Wesley, 2001.

Joshua Bloch and Neal Gafter. Java Puzzlers. Addison–Wesley, 2005.

Hans Boehm. Destructors, Finalizers, and Synchronization. In POPL '03: Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pages 262–272. ACM Press, 2003. URL <http://doi.acm.org/10.1145/604131.604153>.

Hans Boehm. Finalization, Threads, and the Java Memory Model. JavaOne presentation, 2005. URL <http://developers.sun.com/learning/javaoneonline/2005/coreplatform/TS-3281.pdf>.

Joseph Bowbeer. The Last Word in Swing Threads, 2005. URL <http://java.sun.com/products/jfc/tsc/articles/threads/threads3.html>.

Cliff Click. Performance Myths Exposed. JavaOne presentation, 2003.

Cliff Click. Performance Myths Revisited. JavaOne presentation, 2005. URL <http://developers.sun.com/learning/javaoneonline/2005/coreplatform/TS-3268.pdf>.

Martin Fowler. Presentation Model, 2005. URL <http://www.martinfowler.com/eaDev/>

PresentationModel.html.

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison-Wesley, 1995.

Martin Gardner. The fantastic combinations of John Conway's new solitaire game 'Life'. *Scientific American*, October 1970.

James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification, Third Edition*. Addison-Wesley, 2005.

Tim Harris and Keir Fraser. *Language Support for Lightweight Transactions*.

In *OOPSLA '03: Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 388–402. ACM Press, 2003. URL <http://doi.acm.org/10.1145/949305.949340>.

Tim Harris, Simon Marlow, Simon Peyton-Jones, and Maurice Herlihy. *Composable Memory Transactions*. In *PPoPP '05: Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 48–60. ACM Press, 2005. URL <http://doi.acm.org/10.1145/1065944.1065952>.

Maurice Herlihy. *Wait-Free Synchronization*. *ACM Transactions on Programming Languages and Systems*, 13(1):124–149, 1991. URL <http://doi.acm.org/10.1145/114005.102808>.

Maurice Herlihy and Nir Shavit. *Multiprocessor Synchronization and Concurrent Data Structures*. Morgan-Kaufman, 2006.

C. A. R. Hoare. *Monitors: An Operating System Structuring Concept*. *Communications of the ACM*, 17(10):549–557, 1974. URL <http://doi.acm.org/10.1145/355620.361161>.

David Hovemeyer and William Pugh. *Finding Bugs is Easy*. *SIGPLAN Notices*, 39(12):92–106, 2004. URL <http://doi.acm.org/10.1145/1052883.1052895>. Ramnivas Laddad. *AspectJ in Action*. Manning, 2003.

Doug Lea. *Concurrent Programming in Java, Second Edition*. Addison-Wesley, 2000.

Doug Lea. *JSR-133 Cookbook for Compiler Writers*. URL <http://gee.cs.oswego.edu/dl/jmm/cookbook.html>.

J. D. C. Little. A proof of the Queueing Formula $L = \lambda W$. *Operations Research*, 9:383–387, 1961.

Jeremy Manson, William Pugh, and Sarita V. Adve. The Java Memory Model. In POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pages 378–391. ACM Press, 2005. URL <http://doi.acm.org/10.1145/1040305.1040336>.

George Marsaglia. XorShift RNGs. Journal of Statistical Software, 8(13), 2003. URL <http://www.jstatsoft.org/v08/i14>.

Maged M. Michael and Michael L. Scott. Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms. In Symposium on Principles of Distributed Computing, pages 267–275, 1996. URL <http://citeseer.ist.psu.edu/michael96simple.html>.

Mark Moir and Nir Shavit. Concurrent Data Structures, In Handbook of Data Structures and Applications, chapter 47. CRC Press, 2004.

William Pugh and Jeremy Manson. Java Memory Model and Thread Specification, 2004. URL <http://www.cs.umd.edu/~pugh/java/memoryModel/jsr133.pdf>.

M. Raynal. Algorithms for Mutual Exclusion. MIT Press, 1986.

William N. Scherer, Doug Lea, and Michael L. Scott. Scalable Synchronous Queues. In 11th ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming (PPoPP), 2006.

R. K. Treiber. Systems Programming: Coping with Parallelism. Technical Report RJ 5118, IBM Almaden Research Center, April 1986.

Andrew Wellings. Concurrent and Real-Time Programming in Java. John Wiley & Sons, 2004.

索引

Index

Symbols

64-bit operations

nonatomic nature of; 36

A

ABA problem; 336

abnormal thread termination

handling; 161–163

abort saturation policy; 174

See also lifecycle; termination;

abrupt shutdown

limitations; 158–161

triggers for; 164

vs. graceful shutdown; 153

AbstractExecutorService

task representation use; 126

abstractions

See models/modeling; representation;

AbstractQueuedSynchronizer

See AQS framework;

access

See also encapsulation; sharing; visibility;

exclusive

and concurrent collections; 86

integrity

nonblocking algorithm use; 319

mutable state

importance of coordinating; 110

remote resource

as long-running GUI task; 195

serialized

WorkerThread example; 227_{li}

vs. object serialization; 27_{fn}

visibility role in; 33

AccessControlContext

custom thread factory handling; 177

acquisition of locks

See locks, acquisition;

action(s)

See also compound actions; condition, predicate; control flow; task(s);

barrier; 99

JMM specification; 339–342

listener; 195–197

activity(s)

See also task(s);

cancellation; 135, 135–150

tasks as representation of; 113

ad-hoc thread confinement; 43

See also confinement;

algorithm(s)

See also design patterns; idioms; representation;

comparing performance; 263–264

design role of representation; 104

lock-free; 329

Michael-Scott nonblocking queue; 332

nonblocking; 319, 329, 329–336

backoff importance for; 231_{fn}

synchronization; 319–336

SynchronousQueue; 174_{fn}

parallel iterative

barrier use in; 99

recursive

parallelizing; 181–188

Treiber's

nonblocking stack; 331_{li}

work stealing

dequeues and; 92

alien method; 40

See also untrusted code behavior;

deadlock risks posed by; 211

publication risks; 40

allocation

- advantages vs. synchronization; 242
- immutable objects and; 48_{fn}
- object pool use
 - disadvantages of; 241
- scalability advantages; 263

Amdahl's law; 225–229

- See also* concurrent/concurrency; performance; resource(s); throughput; utilization;
- lock scope reduction advantage; 234
- qualitative application of; 227

analysis

- See also* instrumentation; measurement; static analysis tools;
- deadlock
 - thread dump use; 216–217
- escape; 230
- for exploitable parallelism; 123–133
- lock contention
 - thread dump use; 240
- performance; 221–245

annotations; 353–354

- See also* documentation;
- for concurrency documentation; 6
- @GuardedBy; 28, 354
 - synchronization policy documentation use; 75
- @Immutable; 353
- @NotThreadSafe; 353
- @ThreadSafe; 353

AOP (aspect-oriented programming)

- in testing; 259, 273

application(s)

- See also* frameworks(s); service(s); tools;
- scoped objects
 - thread safety concerns; 10
- frameworks, and ThreadLocal; 46
- GUI; 189–202
 - thread safety concerns; 10–11
- parallelizing
 - task decomposition; 113
- shutdown
 - and task cancellation; 136

AQS (AbstractQueuedSynchronizer)

- framework; 308, 311–317
- exit protocol use; 306
- FutureTask implementation
 - piggybacking use; 342

ArrayBlockingQueue; 89

- as bounded buffer example; 292
- performance advantages over BoundedBuffer; 263

ArrayDeque; 92**arrays**

- See also* collections; data structure(s);
- atomic variable; 325

asymmetric two-party tasks

- Exchanger management of; 101

asynchrony/asynchronous

- events, handling; 4
- I/O, and non-interruptable blocking; 148
- sequentiality vs.; 2
- tasks

- execution, Executor framework use; 117

- FutureTask handling; 95–98

atomic variables; 319–336

- and lock contention; 239–240
- classes; 324–329
- locking vs.; 326–329
- strategies for use; 34
- volatile variables vs.; 39, 324–326

atomic/atomicity; 22

- See also* invariant(s); synchronization; visibility;

64-bit operations

- nonatomic nature of; 36
- and compound actions; 22–23
- and multivariable invariants; 57, 67–68
- and open call restructuring; 213
- and service shutdown; 153
- and state transition constraints; 56
- caching issues; 24–25
- client-side locking support for; 80
- field updaters; 335–336
- immutable object use for; 48
- in cache implementation design; 106
- intrinsic lock enforcement of; 25–26
- loss
 - risk of lock scope reduction; 234
- Map operations; 86
- put-if-absent; 71–72
- statistics gathering hooks use; 179
- thread-safety issues
 - in servlets with state; 19–23

- AtomicBoolean**; 325
 - AtomicInteger**; 324
 - nonblocking algorithm use; 319
 - random number generator using; 327_{li}
 - AtomicLong**; 325
 - AtomicReference**; 325
 - nonblocking algorithm use; 319
 - safe publication use; 52
 - AtomicReferenceFieldUpdater**; 335
 - audit(ing)**
 - See also* instrumentation;
 - audit(ing) tools**; 28_{fn}
 - AWT (Abstract Window Toolkit)**
 - See also* GUI;
 - thread use; 9
 - safety concerns and; 10–11
 - B**
 - backoff**
 - and nonblocking algorithms; 231_{fn}
 - barging**; 283
 - See also* fairness; ordering; synchronization;
 - and read-write locks; 287
 - performance advantages of; 284
 - barrier(s)**; 99, 99–101
 - See also* latch(es); semaphores; synchronizers;
 - based timer; 260–261
 - action; 99
 - memory; 230, 338
 - point; 99
 - behavior**
 - See also* activities; task(s);
 - bias**
 - See* testing, pitfalls;
 - bibliography**; 355–357
 - binary latch**; 304
 - AQS-based; 313–314
 - binary semaphore**
 - mutex use; 99
 - Bloch, Joshua**
 - (bibliographic reference); 69
 - block(ing)**; 92
 - bounded collections
 - semaphore management of; 99
 - testing; 248
 - context switching impact of; 230
 - interruptible methods and; 92–94
 - interruption handling methods; 138
 - methods
 - and interruption; 143
 - non-interruptable; 147–150
 - operations
 - testing; 250–252
 - thread pool size impact; 170
 - queues; 87–94
 - See also* Semaphore;
 - and thread pool management; 173
 - cancellation, problems; 138
 - cancellation, solutions; 140
 - Executor functionality combined with; 129
 - producer-consumer pattern and; 87–92
 - spin-waiting; 232
 - state-dependent actions; 291–308
 - and polling; 295–296
 - and sleeping; 295–296
 - condition queues; 296–308
 - structure; 292_{li}
 - threads, costs of; 232
 - waits
 - timed vs. unbounded; 170
- BlockingQueue**; 84–85
 - and state-based preconditions; 57
 - safe publication use; 52
 - thread pool use of; 173
- bound(ed)**
 - See also* constraints; encapsulation;
 - blocking collections
 - semaphore management of; 99
 - buffers
 - blocking operations; 292
 - scalability testing; 261
 - size determination; 261
 - queues
 - and producer-consumer pattern; 88
 - saturation policies; 174–175
 - thread pool use; 172
 - thread pool use of; 173
 - resource; 221
- boundaries**
 - See also* encapsulation;
 - task; 113
 - analysis for parallelism; 123–133
- broken multi-threaded programs**
 - strategies for fixing; 16

BrokenBarrierException

parallel iterative algorithm use; 99

buffer(s)

See also cache/caching;

bounded

blocking state-dependent operations with; 292

scalability testing; 261

size determination; 261

BoundedBuffer example; 249_{li}

condition queue use; 297

test case development for; 248

BoundedBufferTest example; 250_{li}

capacities

comparison testing; 261–263

testing; 248

bug pattern(s); 271, 271

See also debugging; design patterns; testing;

detector; 271

busy-waiting; 295

See also spin-waiting;

C**cache/caching**

See also performance;

atomicity issues; 24–25

flushing

and memory barriers; 230

implementation issues

atomic/atomicity; 106

safety; 104

misses

as cost of context switching; 229

result

building; 101–109

Callable; 126_{li}

FutureTask use; 95

results handling capabilities; 125

callbacks

testing use; 257–259

caller-runs saturation policy; 174**cancellation; 135–150**

See also interruption; lifecycle; shutdown;

activity; 135

as form of completion; 95

Future use; 145–147

interruptible lock acquisition; 279–281

interruption relationship to; 138

long-running GUI tasks; 197–198

non-standard

encapsulation of; 148–150

reasons and strategies; 147–150

points; 140

policy; 136

and thread interruption policy; 141

interruption advantages as implementation strategy; 140

reasons for; 136

shutdown and; 135–166

task

Executor handling; 125

in timed task handling; 131

timed locks use; 279

CancellationException

Callable handling; 98

CAS (compare-and-swap) instructions;

321–324

See also atomic/atomicity, variables;

Java class support in Java 5.0; 324

lock-free algorithm use; 329

nonblocking algorithm use; 319, 329

cascading effects

of thread safety requirements; 28

cellular automata

barrier use for computation of; 101

check-then-act operation

See also compound actions;

as race condition cause; 21

atomic variable handling; 325

compound action

in collection operations; 79

multivariable invariant issues; 67–68

service shutdown issue; 153

checkpoint

state

shutdown issues; 158

checksums

safety testing use; 253

class(es)

as instance confinement context; 59

extension

strategies and risks; 71

with helper classes; 72–73

synchronized wrapper

client-side locking support; 73

thread-safe

and object composition; 55–78

cleanup

See also lifecycle;
and interruption handling
protecting data integrity; 142
in end-of-lifecycle processing; 135
JVM shutdown hooks use for; 164

client(s)

See also server;
requests
as natural task boundary; 113

client-side locking; 72–73, 73

See also lock(ing);
and compound actions; 79–82
and condition queues; 306
class extension relationship to; 73
stream class management; 150_{fn}

coarsening

See also lock(ing);
lock; 231, 235_{fn}, 286

code review

as quality assurance strategy; 271

collections

See also hashtables; lists; set(s);
bounded blocking
semaphore management of; 99
concurrent; 84–98
building block; 79–110
copying
as alternative to locking; 83
lock striping use; 237
synchronized; 79–84
concurrent collections vs.; 84

Collections.synchronizedList

safe publication use; 52

Collections.synchronizedXxx

synchronized collection creation; 79

communication

mechanisms for; 1

compare-and-swap (CAS) instructions

See CAS;

comparison

priority-ordered queue use; 89

compilation

dynamic
and performance testing; 267–
268
timing and ordering alterations
thread safety risks; 7

completion; 95

See also lifecycle;
notification

of long-running GUI task; 198

service

Future; 129

task

measuring service time variance;
264–266
volatile variable use with; 39

CompletionService

in page rendering example; 129

composition; 73

See also delegation; encapsulation;
as robust functionality extension
mechanism; 73
of objects; 55–78

compound actions; 22

See also atomic/atomicity; concur-
rent/concurrency, collec-
tions; race conditions;
atomicity handling of; 22–23
concurrency design rules role; 110
concurrent collection support for; 84
examples of

See check-then-act operation;
iteration; navigation; put-
if-absent operation; read-
modify-write; remove-if-
equal operation; replace-if-
equal operation;

in cache implementation; 106

in synchronized collection class use
mechanisms for handling; 79–82

synchronization requirements; 29

computation

compute-intensive code
impact on locking behavior; 34
thread pool size impact; 170

deferred

design issues; 125

thread-local

and performance testing; 268

Concurrent Programming in Java; 42,

57, 59, 87, 94, 95, 98, 99, 101,
124, 201, 211, 279, 282, 304

concurrent/concurrency

See also parallelizing/parallelism;
safety; synchroniza-
tion/synchronized;
and synchronized collections; 84
and task independence; 113
annotations; 353–354
brief history; 1–2

- building blocks; 79–110
- cache implementation issues; 103
- collections; 84–98
- ConcurrentHashMap** locking strategy
 - advantages; 85
- debugging
 - costs vs. performance optimization value; 224
- design rules; 110
- errors
 - See* deadlock; livelock; race conditions; starvation;
- fine-grained
 - and thread-safe data models; 201
- modifying
 - synchronized collection problems with; 82
- object pool disadvantages; 241
- poor; 30
- prevention
 - See also* single-threaded;
 - single-threaded executor use; 172, 177–178
- read-write lock advantages; 286–289
- testing; 247–274
- ConcurrentHashMap**; 84–86
 - performance advantages of; 242
- ConcurrentLinkedDeque**; 92
- ConcurrentLinkedQueue**; 84–85
 - algorithm; 319–336
 - reflection use; 335
 - safe publication use; 52
- ConcurrentMap**; 84, 87_{li}
 - safe publication use; 52
- ConcurrentModificationException**
 - avoiding; 85
 - fail-fast iterators use; 82–83
- ConcurrentSkipListMap**; 85
- ConcurrentSkipListSet**; 85
- Condition**; 307_{li}
 - explicit condition object use; 306
 - intrinsic condition queues vs. performance considerations; 308
- condition**
 - predicate; 299, 299–300
 - lock and condition variable relationship; 308
 - queues; 297
 - See also* synchronizers;
 - AQS support for; 312
 - blocking state-dependent operations use; 296–308
 - explicit; 306–308
 - intrinsic; 297
 - intrinsic, disadvantages of; 306
 - using; 298
- variables
 - explicit; 306–308
- waits
 - and condition predicate; 299
 - canonical form; 301_{li}
 - interruptible, as feature of Condition; 307
 - uninterruptable, as feature of Condition; 307
 - waking up from, condition queue handling; 300–301
- conditional**
 - See also* blocking/blocks;
 - notification; 303
 - as optimization; 303
 - subclassing safety issues; 304
 - use; 304_{li}
 - read-modify-writer operations
 - atomic variable support for; 325
- configuration**
 - of **ThreadPoolExecutor**; 171–179
 - thread creation
 - and thread factories; 175
 - thread pool
 - post-construction manipulation; 177–179
- confinement**
 - See also* encapsulation; single-thread(ed);
 - instance; 59, 58–60
 - stack; 44, 44–45
 - thread; 42, 42–46
 - ad-hoc; 43
 - and execution policy; 167
 - in Swing; 191–192
 - role, synchronization policy specification; 56
 - serial; 90, 90–92
 - single-threaded GUI framework use; 190
 - ThreadLocal**; 45–46
- Connection**
 - thread confinement use; 43
 - ThreadLocal** variable use with; 45

- consistent/consistency**
 - copy timeliness vs.
 - as design tradeoff; 62
 - data view timeliness vs.
 - as design tradeoff; 66, 70
 - lock ordering
 - and deadlock avoidance; 206
 - weakly consistent iterators; 85
- constraints**
 - See also* invariant(s); post-conditions;
pre-conditions;
 - state transition; 56
 - thread creation
 - importance of; 116
- construction/constructors**
 - See also* lifecycle;
 - object
 - publication risks; 41–42
 - thread handling issues; 41–42
 - partial
 - unsafe publication influence; 50
 - private constructor capture idiom;
69_{fn}
 - starting thread from
 - as concurrency bug pattern; 272
 - ThreadPoolExecutor; 172_{li}
 - post-construction customization;
177
- consumers**
 - See also* blocking, queues; producer-consumer pattern;
 - blocking queues use; 88
 - producer-consumer pattern
 - blocking queues and; 87–92
- containers**
 - See also* collections;
 - blocking queues as; 94
 - scoped
 - thread safety concerns; 10
- contention/contented**
 - as performance inhibiting factor; 263
 - intrinsic locks vs. ReentrantLock
 - performance considerations;
282–286
 - lock
 - costs of; 320
 - measurement; 240–241
 - reduction impact; 211
 - reduction, strategies; 232–242
 - scalability impact; 232
 - signal method reduction in; 308
 - locking vs. atomic variables; 328
 - resource
 - and task execution policy; 119
 - deque advantages; 92
 - scalability under
 - as AQS advantage; 311
 - scope
 - atomic variable limitation of; 324
 - synchronization; 230
 - thread
 - collision detection help with; 321
 - latches help with; 95
 - throughput impact; 228
 - unrealistic degrees of
 - as performance testing pitfall;
268–269
- context switching; 229**
 - See also* performance;
 - as cost of thread use; 229–230
 - condition queues advantages; 297
 - cost(s); 8
 - message logging
 - reduction strategies; 243–244
 - performance impact of; 221
 - reduction; 243–244
 - signal method reduction in; 308
 - throughput impact; 228
- control flow**
 - See also* event(s); lifecycle; MVC
(model-view-controller) pattern;
 - coordination
 - in producer-consumer pattern;
94
 - event handling
 - model-view objects; 195_{fg}
 - simple; 194_{fg}
 - latch characteristics; 94
 - model-view-controller pattern
 - and inconsistent lock ordering;
190
 - vehicle tracking example; 61
- convenience**
 - See also* responsiveness;
 - as concurrency motivation; 2
- conventions**
 - annotations
 - concurrency documentation; 6
 - Java monitor pattern; 61

cooperation/cooperating

See also concurrent/concurrency;
synchronization;
end-of-lifecycle mechanisms
interruption as; 93, 135
model, view, and controller objects
in GUI applications
inconsistent lock ordering; 190
objects
deadlock, lock-ordering; 212_{li}
deadlock, possibilities; 211
livelock possibilities; 218
thread
concurrency mechanisms for; 79

coordination

See also synchronization/synchronized;
control flow
producer-consumer pattern,
blocking queues use; 94
in multithreaded environments
performance impact of; 221
mutable state access
importance of; 110

copying

collections
as alternative to locking; 83
data
thread safety consequences; 62

CopyOnWriteArrayList; 84, 86–87

safe publication use; 52
versioned data model use
in GUI applications; 201

CopyOnWriteArraySet

safe publication use; 52
synchronized Set replacement; 86

core pool size parameter

thread creation impact; 171, 172_{fn}

correctly synchronized program; 341**correctness; 17**

See also safety;
testing; 248–260
goals; 247
thread safety defined in terms of; 17

corruption

See also atomic/atomicity; encapsulation; safety; state;
data
and interruption handling; 142
causes, stale data; 35

cost(s)

See also guidelines; performance;
safety; strategies; tradeoffs;
thread; 229–232
context switching; 8
locality loss; 8
tradeoffs
in performance optimization
strategies; 223

CountDownLatch; 95

AQS use; 315–316
puzzle-solving framework use; 184
TestHarness example use; 96

counting semaphores; 98

See also Semaphore;
permits, thread relationships; 248
SemaphoreOnLock example; 310_{li}

coupling

See also dependencies;
behavior
blocking queue handling; 89
implicit
between tasks and execution
policies; 167–170

CPU utilization

See also performance;
and sequential execution; 124
condition queues advantages; 297
impact on performance testing; 261
monitoring; 240–241
optimization
as multithreading goal; 222
spin-waiting impact on; 295

creation

See also copying; design; policy(s);
representation;
atomic compound actions; 80
class
existing thread-safe class reuse
advantages over; 71
collection copy
as immutable object strategy; 86
of immutable objects; 48
of state-dependent methods; 57
synchronizer; 94
thread; 171–172
explicitly, for tasks; 115
thread factory use; 175–177
unbounded, disadvantages; 116
thread pools; 120
wrappers

- during memoization; 103
- customization**
 - thread configuration
 - ThreadFactory use; 175
 - thread pool configuration
 - post-construction; 177–179
- CyclicBarrier**; 99
 - parallel iterative algorithm use; 102_{li}
 - testing use; 255_{li}, 260_{li}

D

- daemon threads**; 165
- data**
 - See also* state;
 - contention avoidance
 - and scalability; 237
 - hiding
 - thread-safety use; 16
 - nonatomic
 - 64-bit operations; 36
 - sharing; 33–54
 - See also* page renderer examples;
 - access coordination; 277–290, 319
 - advantages of threads; 2
 - shared data models; 198–202
 - synchronization costs; 8
 - split data models; 201, 201–202
 - stale; 35–36
 - versioned data model; 201
- data race**; 341
 - race condition vs.; 20_{fn}
- data structure(s)**
 - See also* collections; object(s);
 - queue(s); stack(s); trees;
 - handling
 - See* atomic/atomicity; confinement; encapsulation; iterators/iteration; recursion;
 - protection
 - and interruption handling; 142
 - shared
 - as serialization source; 226
 - testing insertion and removal handling; 248
- database(s)**
 - deadlock recovery capabilities; 206
 - JDBC Connection
 - thread confinement use; 43
 - thread pool size impact; 171

- Date**
 - effectively immutable use; 53
- dead-code elimination**
 - and performance testing; 269–270
- deadline-based waits**
 - as feature of Condition; 307
- deadlock(s)**; 205, 205–217
 - See also* concurrent/concurrency,
 - errors; liveness; safety;
 - analysis
 - thread dump use; 216–217
 - as liveness failure; 8
 - avoidance
 - and thread confinement; 43_{fn}
 - nonblocking algorithm advantages; 319, 329
 - strategies for; 215–217
 - cooperating objects; 211
 - diagnosis
 - strategies for; 215–217
 - dynamic lock order; 207–210
 - in GUI framework; 190
 - lock splitting as risk factor for; 235
 - locking during iteration risk of; 83
 - recovery
 - database capabilities; 206
 - polled and timed lock acquisition use; 279, 280
 - timed locks use; 215
 - reentrancy avoidance of; 27
 - resource; 213–215
 - thread starvation; 169, 168–169, 215
- deadly embrace**
 - See* deadlock;
- death, thread**
 - abnormal, handling; 161–163
- debugging**
 - See also* analysis; design; documentation; recovery; testing;
 - annotation use; 353
 - concurrency
 - costs vs. performance optimization value; 224
 - custom thread factory as aid for; 175
 - JVM optimization pitfalls; 38_{fn}
 - thread dump use; 216_{fn}
 - thread dumps
 - intrinsic lock advantage over ReentrantLock; 285–286
 - unbounded thread creation risks;

decomposition

See also composition; delegation;
encapsulation;
producer-consumer pattern; 89
tasks-related; 113–134

Decorator pattern

collection class use for wrapper factories; 60

decoupling

of activities
as producer-consumer pattern
advantage; 87
task decomposition as representation of; 113
of interrupt notification from handling in Thread interruption handling methods; 140
task submission from execution and Executor framework; 117

delayed tasks

See also time/timing;
handling of; 123

DelayQueue

time management; 123

delegation

See also composition; design; safety;
advantages
class extension vs.; 314
for class maintenance safety; 234
thread safety; 234
failure causes; 67–68
management; 62

dependencies

See also atomic/atomicity; invariant(s); postconditions; preconditions; state;
code
as removal, as producer-consumer pattern advantage; 87
in multiple-variable invariants
thread safety issues; 24
state
blocking operations; 291–308
classes; 291
classes, building; 291–318
managing; 291–298
operations; 57
operations, condition queue handling; 296–308

task freedom from, importance of; 113

task

and execution policy; 167
thread starvation deadlock; 168

task freedom from

importance; 113

Deque; 92**deques**

See also collections; data structure(s);
queue(s);
work stealing and; 92

design

See also documentation; guidelines;
policies; representation;
strategies;

class

state ownership as element of; 57–58

concurrency design rules; 110

concurrency testing; 250–252

condition queue encapsulation; 306

condition queues

and condition predicate; 299

control flow

latch characteristics; 94

execution policy

influencing factors; 167

GUI single-threaded use

rationale for; 189–190

importance

in thread-safe programs; 16

of thread-safe classes

guidelines; 55–58

parallelism

application analysis for; 123–133

parallelization criteria; 181

performance

analysis, monitoring, and improvement; 221–245

performance tradeoffs

evaluation of; 223–225

principles

simplicity of final fields; 48

producer-consumer pattern

decoupling advantages; 117

Executor framework use; 117

program

and task decomposition; 113–134

result-bearing tasks

representation issues; 125

- strategies
 - for InterruptedException; 93
- thread confinement; 43
- thread pool size
 - relevant factors for; 170
- timed tasks; 131–133
- tradeoffs
 - collection copying vs. locking during iteration; 83
 - concurrent vs. synchronized collections; 85
 - copy-on-write collections; 87
 - synchronized block; 34
 - timeliness vs. consistency; 62, 66, 70
- design patterns**
 - antipattern example
 - double-checked locking; 348–349
 - examples
 - See Decorator pattern; MVC (model-view-controller) pattern; producer-consumer pattern; Singleton pattern;
- destruction**
 - See teardown;
- dining philosophers problem**; 205
 - See also deadlock;
- discard saturation policy**; 174
- discard-oldest saturation policy**; 174
- documentation**
 - See also debugging; design; good practices; guidelines; policy(s);
 - annotation use; 6, 353
 - concurrency design rules role; 110
 - critical importance for conditional notification use; 304
 - importance
 - for special execution policy requirements; 168
 - stack confinement usage; 45
 - of synchronization policies; 74–77
 - safe publication requirements; 54
- double-checked locking (DCL)**; 348–349
 - as concurrency bug pattern; 272
- downgrading**
 - read-write lock implementation strategy; 287
- driver program**
 - for TimedPutTakeTest example; 262
- dynamic**
 - See also responsiveness;
 - compilation
 - as performance testing pitfall; 267–268
 - lock order deadlocks; 207–210
- E**
- EDT (event dispatch thread)**
 - GUI frameworks use; 5
 - single-threaded GUI use; 189
 - thread confinement use; 42
- Effective Java Programming Language Guide**; 46–48, 73, 166, 257, 292, 305, 314, 347
- efficiency**
 - See also performance;
 - responsiveness vs.
 - polling frequency; 143
 - result cache, building; 101–109
- elision**
 - lock; 231_{fn}
 - JVM optimization; 286
- encapsulation**
 - See also access; atomic/atomicity; confinement; safety; state; visibility;
 - breaking
 - costs of; 16–17
 - code
 - as producer-consumer pattern advantage; 87
 - composition use; 74
 - concurrency design rules role; 110
 - implementation
 - class extension violation of; 71
 - instance confinement relationship with; 58–60
 - invariant management with; 44
 - locking behavior
 - reentrancy facilitation of; 27
 - non-standard cancellation; 148–150
 - of condition queues; 306
 - of lifecycle methods; 155
 - of synchronization
 - hidden iterator management through; 83
 - publication dangers for; 39
 - state

- breaking, costs of; 16–17
 - invariant protection use; 83
 - ownership relationship with; 58
 - synchronizer role; 94
 - thread-safe class use; 23
- synchronization policy
 - and client-side locking; 71
- thread ownership; 150
- thread-safety role; 55
- thread-safety use; 16
- end-of-lifecycle**
 - See also* thread(s);
 - management techniques; 135–166
- enforcement**
 - locking policies, lack of; 28
- entry protocols**
 - state-dependent operations; 306
- Error**
 - Callable handling; 97
- error(s)**
 - as cancellation reason; 136
 - concurrency
 - See* deadlock; livelock; race conditions;
- escape; 39**
 - analysis; 230
 - prevention
 - in instance confinement; 59
 - publication and; 39–42
 - risk factors
 - in instance confinement; 60
- Ethernet protocol**
 - exponential backoff use; 219
- evaluation**
 - See also* design; measurement; testing;
 - of performance tradeoffs; 223–225
- event(s); 191**
 - as cancellation reason; 136
 - dispatch thread
 - GUI frameworks use; 5
 - handling
 - control flow, simple; 194_{fg}
 - model-view objects; 195_{fg}
 - threads benefits for; 4
 - latch handling based on; 99
 - main event loop
 - vs. event dispatch thread; 5
 - notification
 - copy-on-write collection advantages; 87
 - sequential processing
 - in GUI applications; 191
 - timing
 - and liveness failures; 8
- example classes**
 - AtomicPseudoRandom; 327_{li}
 - AttributeStore; 233_{li}
 - BackgroundTask; 199_{li}
 - BarrierTimer; 261_{li}
 - BaseBoundedBuffer; 293_{li}
 - BetterAttributeStore; 234_{li}
 - BetterVector; 72_{li}
 - Big; 258_{li}
 - BoundedBuffer; 248, 249_{li}, 297, 298_{li}
 - BoundedBufferTest; 250_{li}
 - BoundedExecutor; 175
 - BoundedHashSet; 100_{li}
 - BrokenPrimeProducer; 139_{li}
 - CachedFactorizer; 31_{li}
 - CancellableTask; 151_{li}
 - CasCounter; 323_{li}
 - CasNumberRange; 326_{li}
 - CellularAutomata; 102_{li}
 - Computable; 103_{li}
 - ConcurrentPuzzleSolver; 186_{li}
 - ConcurrentStack; 331_{li}
 - ConditionBoundedBuffer; 308, 309_{li}
 - Consumer; 256_{li}
 - Counter; 56_{li}
 - CountingFactorizer; 23_{li}
 - CrawlerThread; 157_{li}
 - DelegatingVehicleTracker; 65_{li}, 201
 - DemonstrateDeadlock; 210_{li}
 - Dispatcher; 212_{li}, 214_{li}
 - DoubleCheckedLocking; 349_{li}
 - ExpensiveFunction; 103_{li}
 - Factorizer; 109_{li}
 - FileCrawler; 91_{li}
 - FutureRenderer; 128_{li}
 - GrumpyBoundedBuffer; 292, 294_{li}
 - GuiExecutor; 192, 194_{li}
 - HiddenIterator; 84_{li}
 - ImprovedList; 74_{li}
 - Indexer; 91_{li}
 - IndexerThread; 157_{li}
 - IndexingService; 156_{li}
 - LazyInitRace; 21_{li}
 - LeftRightDeadlock; 207_{li}
 - LifecycleWebServer; 122_{li}
 - LinkedQueue; 334_{li}

- ListHelper; 73, 74_{li}
- LogService; 153, 154_{li}
- LogWriter; 152_{li}
- Memoizer; 103_{li}, 108_{li}
- Memoizer2; 104_{li}
- Memoizer3; 106_{li}
- MonitorVehicleTracker; 63_{li}
- MutableInteger; 36_{li}
- MutablePoint; 64_{li}
- MyAppThread; 177, 178_{li}
- MyThreadFactory; 177_{li}
- Node; 184_{li}
- NoVisibility; 34_{li}
- NumberRange; 67_{li}
- OneShotLatch; 313_{li}
- OneValueCache; 49_{li}, 51_{li}
- OutOfTime; 124_{li}, 161
- PersonSet; 59_{li}
- Point; 64_{li}
- PossibleReordering; 340_{li}
- Preloader; 97_{li}
- PrimeGenerator; 137_{li}
- PrimeProducer; 141_{li}
- PrivateLock; 61_{li}
- Producer; 256_{li}
- PutTakeTest; 255_{li}, 260
- Puzzle; 183_{li}
- PuzzleSolver; 188_{li}
- QueueingFuture; 129_{li}
- ReaderThread; 149_{li}
- ReadWriteMap; 288_{li}
- ReentrantLockPseudoRandom; 327_{li}
- Renderer; 130_{li}
- SafeListener; 42_{li}
- SafePoint; 69_{li}
- SafeStates; 350_{li}
- ScheduledExecutorService; 145_{li}
- SemaphoreOnLock; 310_{li}
- Sequence; 7_{li}
- SequentialPuzzleSolver; 185_{li}
- ServerStatus; 236_{li}
- SimulatedCAS; 322_{li}
- SingleThreadRenderer; 125_{li}
- SingleThreadWebServer; 114_{li}
- SleepyBoundedBuffer; 295, 296_{li}
- SocketUsingTask; 151_{li}
- SolverTask; 186_{li}
- StatelessFactorizer; 18_{li}
- StripedMap; 238_{li}
- SwingUtilities; 191, 192, 193_{li}
- Sync; 343_{li}
- SynchronizedFactorizer; 26_{li}
- SynchronizedInteger; 36_{li}
- TaskExecutionWebServer; 118_{li}
- TaskRunnable; 94_{li}
- Taxi; 212_{li}, 214_{li}
- TestHarness; 96_{li}
- TestingThreadFactory; 258_{li}
- ThisEscape; 41_{li}
- ThreadDeadlock; 169_{li}
- ThreadGate; 305_{li}
- ThreadPerTaskExecutor; 118_{li}
- ThreadPerTaskWebServer; 115_{li}
- ThreeStooges; 47_{li}
- TimedPutTakeTest; 261
- TimingThreadPool; 180_{li}
- TrackingExecutorService; 159_{li}
- UEHLogger; 163_{li}
- UnsafeCachingFactorizer; 24_{li}
- UnsafeCountingFactorizer; 19_{li}
- UnsafeLazyInitialization; 345_{li}
- UnsafeStates; 40_{li}
- ValueLatch; 184, 187_{li}
- VisualComponent; 66_{li}
- VolatileCachedFactorizer; 50_{li}
- WebCrawler; 160_{li}
- Widget; 27_{li}
- WithinThreadExecutor; 119_{li}
- WorkerThread; 227_{li}
- exceptions**
 - See also error(s); interruption; lifecycle;
 - and precondition failure; 292–295
 - as form of completion; 95
 - Callable handling; 97
 - causes
 - stale data; 35
 - handling
 - Runnable limitations; 125
 - logging
 - UEHLogger example; 163_{li}
 - thread-safe class handling; 82
 - Timer disadvantages; 123
 - uncaught exception handler; 162–163
 - unchecked
 - catching, disadvantages; 161
- Exchanger**
 - See also producer-consumer pattern;
 - as two-party barrier; 101
 - safe publication use; 53

execute

submit vs., uncaught exception handling; 163

execution

policies

design, influencing factors; 167
Executors factory methods; 171
implicit couplings between tasks and; 167–170
parallelism analysis for; 123–133

task; 113–134

policies; 118–119

sequential; 114

ExecutionException

Callable handling; 98

Executor framework; 117_{li}, 117–133

and GUI event processing; 191, 192

and long-running GUI tasks; 195

as producer-consumer pattern; 88

execution policy design; 167

FutureTask use; 97

GuiExecutor example; 194_{li}

single-threaded

deadlock example; 169_{li}

ExecutorCompletionService

in page rendering example; 129

Executors

factory methods

thread pool creation with; 120

ExecutorService

and service shutdown; 153–155

cancellation strategy using; 146

checkMail example; 158

lifecycle methods; 121_{li}, 121–122

exhaustion

See failure; leakage; resource exhaustion;

exit protocols

state-dependent operations; 306

explicit locks; 277–290

interruption during acquisition; 148

exponential backoff

and avoiding livelock; 219

extending

existing thread-safe classes

and client-side locking; 73

strategies and risks; 71

ThreadPoolExecutor; 179

external locking; 73**F****factory(s)**

See also creation;

methods

constructor use with; 42

newTaskFor; 148

synchronized collections; 79, 171

thread pool creation with; 120

thread; 175, 175–177

fail-fast iterators; 82

See also iteration/iterators;

failure

See also exceptions; liveness, failure;

recovery; safety;

causes

stale data; 35

graceful degradation

task design importance; 113

management techniques; 135–166

modes

testing for; 247–274

precondition

bounded buffer handling of; 292

propagation to callers; 292–295

thread

uncaught exception handlers;

162–163

timeout

deadlock detection use; 215

fairness

See also responsiveness; synchronization;

as concurrency motivation; 1

fair lock; 283

nonfair lock; 283

nonfair semaphores vs. fair

performance measurement; 265

queuing

intrinsic condition queues; 297_{fn}

ReentrantLock options; 283–285

ReentrantReadWriteLock; 287

scheduling

thread priority manipulation

risks; 218

'fast path' synchronization

CAS-based operations vs.; 324

costs of; 230

feedback

See also GUI;

user

in long-running GUI tasks; 196_{li}

fields

atomic updaters; 335–336

hot fields

avoiding; 237

updating, atomic variable advantages; 239–240

initialization safety

final field guarantees; 48

FIFO queues

BlockingQueue implementations; 89

files

See also data; database(s);

as communication mechanism; 1

final

and immutable objects; 48

concurrency design rules role; 110

immutability not guaranteed by; 47

safe publication use; 52

volatile vs.; 158_{fn}

finalizers

JVM orderly shutdown use; 164

warnings; 165–166

finally block

See also interruptions; lock(ing);

importance with explicit locks; 278

FindBugs code auditing tool

See also tools;

as static analysis tool example; 271

locking failures detected by; 28_{fn}

unreleased lock detector; 278_{fn}

fire-and-forget event handling strategy

drawbacks of; 195

flag(s)

See mutex;

cancellation request

as cancellation mechanism; 136

interrupted status; 138

flexibility

See also responsiveness; scalability;

and instance confinement; 60

decoupling task submission from execution, advantages for;

119

immutable object design for; 47

in CAS-based algorithms; 322

interruption policy; 142

resource management

as blocking queue advantage; 88

task design guidelines for; 113

task design role; 113

flow control

communication networks, thread

pool comparison; 173_{fn}

fragility

See also debugging; guidelines; robustness; safety; scalability; testing;

issues and causes

as class extension; 71

as client-side locking; 73

interruption use for non-

standard purposes; 138

issue; 43

piggybacking; 342

state-dependent classes; 304

volatile variables; 38

solutions

composition; 73

encapsulation; 17

stack confinement vs. ad-hoc

thread confinement; 44

frameworks

See also AQS framework; data structure(s); Executor framework; RMI framework; Servlets framework;

application

and ThreadLocal; 46

serialization hidden in; 227

thread use; 9

thread use impact on applications; 9

threads benefits for; 4

functionality

extending for existing thread-safe classes

strategies and risks; 71

tests

vs. performance tests; 260

Future; 126_{li}

cancellation

of long-running GUI task; 197

strategy using; 145–147

characteristics of; 95

encapsulation of non-standard cancellation use; 148

results handling capabilities; 125

safe publication use; 53

task lifecycle representation by; 125

- task representation
 - implementation strategies; 126
- FutureTask**; 95
 - AQS use; 316
 - as latch; 95–98
 - completion notification
 - of long-running GUI task; 198
 - efficient and scalable cache implementation with; 105
 - example use; 97_{li}, 108_{li}, 151_{li}, 199_{li}
 - task representation use; 126
- G**
- garbage collection**
 - as performance testing pitfall; 266
- gate**
 - See also* barrier(s); conditional; latch(es);
 - as latch role; 94
 - ThreadGate example; 304
- global variables**
 - ThreadLocal variables use with; 45
- good practices**
 - See* design; documentation; encapsulation; guidelines; performance; strategies;
- graceful**
 - degradation
 - and execution policy; 121
 - and saturation policy; 175
 - limiting task count; 119
 - task design importance; 113
 - shutdown
 - vs. abrupt shutdown; 153
- granularity**
 - See also* atomic/atomicity; scope;
 - atomic variable advantages; 239–240
 - lock
 - Amdahl's law insights; 229
 - reduction of; 235–237
 - nonblocking algorithm advantages; 319
 - serialization
 - throughput impact; 228
 - timer
 - measurement impact; 264
- guarded**
 - objects; 28, 54
 - state
 - locks use for; 27–29
- @GuardedBy**; 353–354
 - and documenting synchronization policy; 7_{fn}, 75
- GUI (Graphical User Interface)**
 - See also* event(s); single-thread(ed); Swing;
 - applications; 189–202
 - thread safety concerns; 10–11
 - frameworks
 - as single-threaded task execution example; 114_{fn}
 - long-running task handling; 195–198
 - MVC pattern use
 - in vehicle tracking example; 61
 - response-time sensitivity
 - and execution policy; 168
 - single-threaded use
 - rationale for; 189–190
 - threads benefits for; 5
- guidelines**
 - See also* design; documentation; policy(s); strategies;
 - allocation vs. synchronization; 242
 - atomicity
 - definitions; 22
 - concurrency design rules; 110
 - Condition methods
 - potential confusions; 307
 - condition predicate
 - documentation; 299
 - lock and condition queue relationship; 300
 - condition wait usage; 301
 - confinement; 60
 - deadlock avoidance
 - alien method risks; 211
 - lock ordering; 206
 - open call advantages; 213
 - thread starvation; 169
 - documentation
 - value for safety; 16
 - encapsulation; 59, 83
 - value for safety; 16
 - exception handling; 163
 - execution policy
 - design; 119
 - special case implications; 168
 - final field use; 48
 - finalizer precautions; 166
 - happens-before use; 346
 - immutability

- effectively immutable objects; 53
- objects; 46
- requirements for; 47
- value for safety; 16
- initialization safety; 349, 350
- interleaving diagrams; 6
- interruption handling
 - cancellation relationship; 138
 - importance of interruption policy knowledge; 142, 145
 - interrupt swallowing precautions; 143
- intrinsic locks vs. `ReentrantLock`; 285
- invariants
 - locking requirements for; 29
 - thread safety importance; 57
 - value for safety; 16
- lock
 - contention, reduction; 233
 - contention, scalability impact; 231
 - holding; 32
 - ordering, deadlock avoidance; 206
- measurement
 - importance; 224
- notification; 303
- objects
 - stateless, thread-safety of; 19
- operation ordering
 - synchronization role; 35
- optimization
 - lock contention impact; 231
 - premature, avoidance of; 223
- parallelism analysis; 123–133
- performance
 - optimization questions; 224
 - simplicity vs.; 32
- postconditions; 57
- private field use; 48
- publication; 52, 54
- safety
 - definition; 18
 - testing; 252
- scalability; 84
 - attributes; 222
 - locking impact on; 232
- sequential loops
 - parallelization criteria; 181
- serialization sources; 227

- sharing
 - safety strategies; 16
- sharing objects; 54
- simplicity
 - performance vs.; 32
- starvation avoidance
 - thread priority precautions; 218
- state
 - consistency preservation; 25
 - managing; 23
 - variables, independent; 68
- stateless objects
 - thread-safety of; 19
- synchronization
 - immutable objects as replacement for; 52
 - shared state requirements for; 28
- task cancellation
 - criteria for; 147
- testing
 - effective performance tests; 270
 - timing-sensitive data races; 254
- this reference
 - publication risks; 41
- threads
 - daemon thread precautions; 165
 - handling encapsulation; 150
 - lifecycle methods; 150
 - pools; 174
 - safety; 18, 55
- volatile variables; 38

H

- hand-over-hand locking; 282**
- happens-before**
 - JMM definition; 340–342
 - piggybacking; 342–344
 - publication consequences; 244–249
- hardware**
 - See also* CPU utilization;
 - concurrency support; 321–324
 - JVM interaction
 - reordering; 34
 - platform memory models; 338
 - timing and ordering alterations by thread safety risks; 7
- hashcodes/hashtables**
 - See also* collections;
 - `ConcurrentHashMap`; 84–86
 - performance advantages of; 242
 - `Hashtable`; 79

- safe publication use; 52
- inducing lock ordering with; 208
- lock striping use; 237
- heap inspection tools**
 - See also* tools;
 - measuring memory usage; 257
- Heisenbugs**; 247_{fn}
- helper classes**
 - and extending class functionality;
72–73
- heterogeneous tasks**
 - parallelization limitations; 127–129
- hijacked signal**
 - See* missed signals;
- Hoare, C. A. R.**
 - Java monitor pattern inspired by
(bibliographic reference); 60_{fn}
- hoisting**
 - variables
 - as JVM optimization pitfall; 38_{fn}
- homogeneous tasks**
 - parallelism advantages; 129
- hooks**
 - See also* extending;
 - completion
 - in `FutureTask`; 198
 - shutdown; 164
 - JVM orderly shutdown; 164–165
 - single shutdown
 - orderly shutdown strategy; 164
 - `ThreadPoolExecutor` extension; 179
- hot fields**
 - avoidance
 - scalability advantages; 237
 - updating
 - atomic variable advantages; 239–
240
- HotSpot JVM**
 - dynamic compilation use; 267
- 'how fast'; 222**
 - See also* GUI; latency; responsive-
ness;
 - vs. 'how much'; 222
- 'how much'; 222**
 - See also* capacity; scalability;
 - throughput;
 - importance for server applications;
223
 - vs. 'how fast'; 222

HttpSession

- thread-safety requirements; 58_{fn}

I

I/O

- See also* resource(s);
- asynchronous
 - non-interruptable blocking; 148
- message logging
 - reduction strategies; 243–244
- operations
 - thread pool size impact; 170
- sequential execution limitations; 124
- server applications
 - task execution implications; 114
- synchronous
 - non-interruptable blocking; 148
 - threads use to simulate; 4
- utilization measurement tools; 240

idempotence

- and race condition mitigation; 161

idioms

- See also* algorithm(s); conventions;
- design patterns; documen-
tation; policy(s); protocols;
- strategies;
- double-checked locking (DCL)
 - as bad practice; 348–349
- lazy initialization holder class; 347–
348
- private constructor capture; 69_{fn}
- safe initialization; 346–348
- safe publication; 52–53

IllegalStateException

- `Callable` handling; 98

@Immutable; 7, 353

immutable/immutability; 46–49

- See also* atomic/atomicity; safety;
- concurrency design rules role; 110
- effectively immutable objects; 53
- initialization safety guarantees; 51
- initialization safety limitation; 350
- objects; 46
 - publication with volatile; 48–49
 - requirements for; 47
- role in synchronization policy; 56
- thread-safety use; 16

implicit coupling

- See also* dependencies;
- between tasks and execution poli-
cies; 167–170

- improper publication**; 51
 - See also* safety;
- increment operation (++)**
 - as non-atomic operation; 19
- independent/independence**; 25
 - See also* dependencies; encapsulation; invariant(s); state;
 - multiple-variable invariant lack of thread safety issues; 24
 - parallelization use; 183
 - state variables; 66, 66–67
 - lock splitting use with; 235
 - task
 - concurrency advantages; 113
- inducing lock ordering**
 - for deadlock avoidance; 208–210
- initialization**
 - See also* construction/constructors;
 - lazy; 21
 - as race condition cause; 21–22
 - safe idiom for; 348_{li}
 - unsafe publication risks; 345
 - safety
 - and immutable objects; 51
 - final field guarantees; 48
 - idioms for; 346–348
 - JMM support; 349–350
- inner classes**
 - publication risks; 41
- instance confinement**; 59, 58–60
 - See also* confinement; encapsulation;
- instrumentation**
 - See also* analysis; logging; monitoring; resource(s), management; statistics; testing;
 - of thread creation
 - thread pool testing use; 258
 - potential
 - as execution policy advantage; 121
 - service shutdown use; 158
 - support
 - Executor framework use; 117
 - thread pool size requirements determination use of; 170
 - ThreadPoolExecutor hooks for; 179
- interfaces**
 - user
 - threads benefits for; 5
- interleaving**
 - diagram interpretations; 6
 - generating
 - testing use; 259
 - logging output
 - and client-side locking; 150_{fn}
 - operation; 81_{fg}
 - ordering impact; 339
 - thread
 - dangers of; 5–8
 - timing dependencies impact on race conditions; 20
 - thread execution
 - in thread safety definition; 18
- interrupted (Thread)**
 - usage precautions; 140
- InterruptedException**
 - flexible interruption policy advantages; 142
 - interruption API; 138
 - propagation of; 143_{li}
 - strategies for handling; 93
 - task cancellation
 - criteria for; 147
- interruption(s)**; 93, 135, 138–150
 - See also* completion; errors; lifecycle; notification; termination; triggering;
 - and condition waits; 307
 - blocking and; 92–94
 - blocking test use; 251
 - interruption response strategy
 - exception propagation; 142
 - status restoration; 142
 - lock acquisition use; 279–281
 - non-cancellation uses for; 143
 - non-interruptable blocking
 - handling; 147–150
 - reasons for; 148
 - policies; 141, 141–142
 - preemptive
 - deprecation reasons; 135_{fn}
 - request
 - strategies for handling; 140
 - responding to; 142–150
 - swallowing
 - as discouraged practice; 93
 - bad consequences of; 140
 - when permitted; 143
 - thread; 138
 - volatile variable use with; 39

intransitivity

encapsulation characterized by; 150

intrinsic condition queues; 297

disadvantages of; 306

intrinsic locks; 25, 25–26

See also encapsulation; lock(ing);

safety; synchronization;

thread(s);

acquisition, non-interruptable blocking reason; 148

advantages of; 285

explicit locks vs.; 277–278

intrinsic condition queue relationship to; 297

limitations of; 28

recursion use; 237_{fn}

ReentrantLock vs.; 282–286

visibility management with; 36

invariant(s)

See also atomic/atomicity; post-conditions; pre-conditions; state;

and state variable publication; 68

BoundedBuffer example; 250

callback testing; 257

concurrency design rules role; 110

encapsulation

state, protection of; 83

value for; 44

immutable object use; 49

independent state variables requirements; 66–67

multivariable

and atomic variables; 325–326

atomicity requirements; 57, 67–68

locking requirements for; 29

preservation of, as thread safety requirement; 24

thread safety issues; 24

preservation of

immutable object use; 46

mechanisms and synchronization policy role; 55–56

publication dangers for; 39

specification of

thread-safety use; 16

thread safety role; 17

iostat application

See also measurement; tools;

I/O measurement; 240

iterators/iteration

See also concurrent/concurrency;

control flow; recursion;

as compound action

in collection operations; 79

atomicity requirements during; 80

fail-fast; 82

ConcurrentModificationException exception with; 82–83

hidden; 83–84

locking

concurrent collection elimination of need for; 85

disadvantages of; 83

parallel iterative algorithms

barrier management of; 99

parallelization of; 181

unreliable

and client-side locking; 81

weakly consistent; 85

J

Java Language Specification, The; 53, 218_{fn}, 259, 358

Java Memory Model (JMM); 337–352

See also design; safety; synchronization; visibility;

initialization safety guarantees for immutable objects; 51

Java monitor pattern; 60, 60–61

composition use; 74

vehicle tracking example; 61–71

Java Programming Language, The; 346

java.nio package

synchronous I/O

non-interruptable blocking; 148

JDBC (Java Database Connectivity)

Connection

thread confinement use; 43

JMM (Java Memory Model)

See Java Memory Model (JMM);

join (Thread)

timed

problems with; 145

JSPs (JavaServer Pages)

thread safety requirements; 10

JVM (Java Virtual Machine)

See also optimization;

deadlock handling limitations; 206

escape analysis; 230–231

lock contention handling; 320_{fn}

- nonblocking algorithm use; 319
 - optimization pitfalls; 38_{fn}
 - optimizations; 286
 - service shutdown issues; 152–153
 - shutdown; 164–166
 - and daemon threads; 165
 - orderly shutdown; 164
 - synchronization optimization by; 230
 - thread timeout interaction
 - and core pool size; 172_{fn}
 - thread use; 9
 - uncaught exception handling; 162_{fn}
- K**
- keep-alive time**
 - thread termination impact; 172
- L**
- latch(es); 94, 94–95**
 - See also* barriers; blocking; semaphores; synchronizers;
 - barriers vs.; 99
 - binary; 304
 - AQS-based; 313–314
 - FutureTask; 95–98
 - puzzle-solving framework use; 184
 - ThreadGate example; 304
 - layering**
 - three-tier application
 - as performance vs. scalability illustration; 223
 - lazy initialization; 21**
 - as race condition cause; 21–22
 - safe idiom for; 348_{li}
 - unsafe publication risks; 345
 - leakage**
 - See also* performance;
 - resource
 - testing for; 257
 - thread; 161
 - Timer problems with; 123
 - UncaughtExceptionHandler
 - prevention of; 162–163
 - lexical scope**
 - as instance confinement context; 59
 - library**
 - thread-safe collections
 - safe publication guarantees; 52
- Life cellular automata game**
 - barrier use for computation of; 101
 - lifecycle**
 - See also* cancellation; completion; construction/constructors; Executor; interruption; shutdown; termination; thread(s); time/timing;
 - encapsulation; 155
 - Executor
 - implementations; 121–122
 - management strategies; 135–166
 - support
 - Executor framework use; 117
 - task
 - and Future; 125
 - Executor phases; 125
 - thread
 - performance impact; 116
 - thread-based service management; 150
 - lightweight processes**
 - See* threads;
 - linked lists**
 - LinkedBlockingDeque; 92
 - LinkedBlockingQueue; 89
 - performance advantages; 263
 - thread pool use of; 173–174
 - LinkedList; 85
 - Michael-Scott nonblocking queue; 332–335
 - nonblocking; 330
 - List**
 - CopyOnWriteArrayList as concurrent collection for; 84, 86
 - listeners**
 - See also* event(s);
 - action; 195–197
 - Swing
 - single-thread rule exceptions; 190
 - Swing event handling; 194
 - lists**
 - See also* collections;
 - CopyOnWriteArrayList
 - safe publication use; 52
 - versioned data model use; 201
 - LinkedList; 85
 - List
 - CopyOnWriteArrayList as concurrent replacement; 84, 86

Little's law

lock contention corollary; 232_{fn}

livelock; 219, 219

See also concurrent/concurrency,
errors; liveness;
as liveness failure; 8

liveness

See also performance; responsiveness
failure;

causes

See deadlock; livelock; missed
signals; starvation;

failure

avoidance; 205–220

improper lock acquisition risk of; 61

nonblocking algorithm advantages;
319–336

performance and

in servlets with state; 29–32

safety vs.

See safety;

term definition; 8

testing

criteria; 248

thread safety hazards for; 8

local variables

See also encapsulation; state; vari-
ables;

for thread confinement; 43

stack confinement use; 44

locality, loss of

as cost of thread use; 8

Lock; 277_{li}, 277–282

and Condition; 307

interruptible acquisition; 148

timed acquisition; 215

lock(ing); 85

See also confinement; encapsulation;
@GuardedBy; safety; synchro-
nization;

acquisition

AQS-based synchronizer opera-
tions; 311–313

improper, liveness risk; 61

interruptible; 279–281

intrinsic, non-interruptible

blocking reason; 148

nested, as deadlock risk; 208

polled; 279

protocols, instance confinement
use; 60

reentrant lock count; 26

timed; 279

and instance confinement; 59

atomic variables vs.; 326–329

avoidance

immutable objects use; 49

building

AQS use; 311

client-side; 72–73, 73

and compound actions; 79–82

condition queue encapsulation

impact on; 306

stream class management; 150_{fn}

vs. class extension; 73

coarsening; 231

as JVM optimization; 286

impact on splitting synchronized

blocks; 235_{fn}

concurrency design rules role; 110

ConcurrentHashMap strategy; 85

ConcurrentModificationException

avoidance with; 82

condition variable and condition

predicate relationship; 308

contention

measurement; 240–241

reduction, guidelines; 233

reduction, impact; 211

reduction, strategies; 232–242

scalability impact of; 232

coupling; 282

cyclic locking dependencies

as deadlock cause; 205

disadvantages of; 319–321

double-checked

as concurrency bug pattern; 272

elision; 231_{fn}

as JVM optimization; 286

encapsulation of

reentrancy facilitation; 27

exclusive

alternative to; 239–240

alternatives to; 321

inability to use, as Concurrent-
HashMap disadvantage; 86

timed lock use; 279

explicit; 277–290

interruption during lock acqui-
sition use; 148

granularity

Amdahl's law insights; 229

- reduction of; 235–237
 - hand-over-hand; 282
 - in blocking actions; 292
 - intrinsic; 25, 25–26
 - acquisition, non-interruptable
 - blocking reason; 148
 - advantages of; 285
 - explicit locks vs.; 277–278
 - intrinsic condition queue relationship to; 297
 - limitations of; 28
 - private locks vs.; 61
 - recursion use; 237_{fn}
 - ReentrantLock vs., performance considerations; 282–286
 - iteration
 - concurrent collection elimination of need for; 85
 - disadvantages of; 83
 - monitor
 - See intrinsic locks;
 - non-block-structured; 281–282
 - nonblocking algorithms vs.; 319
 - open calls
 - for deadlock avoidance; 211–213
 - ordering
 - deadlock risks; 206–213
 - dynamic, deadlocks resulting from; 207–210
 - inconsistent, as multithreaded GUI framework problem; 190
 - private
 - intrinsic locks vs.; 61
 - protocols
 - shared state requirements for; 28
 - read-write; 286–289
 - implementation strategies; 287
 - reentrant
 - semantics; 26–27
 - semantics, ReentrantLock capabilities; 278
 - ReentrantLock fairness options; 283–285
 - release
 - in hand-over-hand locking; 282
 - intrinsic locking disadvantages; 278
 - preference, in read-write lock implementation; 287
 - role
 - synchronization policy; 56
 - scope
 - See also lock(ing), granularity; narrowing, as lock contention reduction strategy; 233–235
 - splitting; 235
 - Amdahl's law insights; 229
 - as lock granularity reduction strategy; 235
 - ServerStatus examples; 236_{li}
 - state guarding with; 27–29
 - striping; 237
 - Amdahl's law insights; 229
 - ConcurrentHashMap use; 85
 - stripping; 237
 - thread dump information about; 216
 - thread-safety issues
 - in servlets with state; 23–29
 - timed; 215–216
 - unreleased
 - as concurrency bug pattern; 272
 - visibility and; 36–37
 - volatile variables vs.; 39
 - wait
 - and condition predicate; 299
 - lock-free algorithms; 329**
 - logging**
 - See also instrumentation; exceptions
 - UEHLogger example; 163_{li}
 - service
 - as example of stopping a thread-based service; 150–155
 - thread customization example; 177
 - ThreadPoolExecutor hooks for; 179
 - logical state; 58**
 - loops/looping**
 - and interruption; 143
- M**
- main event loop**
 - vs. event dispatch thread; 5
 - Map**
 - ConcurrentHashMap as concurrent replacement; 84
 - performance advantages; 242
 - atomic operations; 86
 - maximum pool size parameter; 172**
 - measurement**
 - importance for effective optimization; 224
 - performance; 222

- profiling tools; 225
 - lock contention; 240
 - responsiveness; 264–266
 - strategies and tools
 - profiling tools; 225
 - ThreadPoolExecutor hooks for; 179
- memoization**; 103
 - See also* cache/caching;
- memory**
 - See also* resource(s);
 - barriers; 230, 338
 - depletion
 - avoiding request overload; 173
 - testing for; 257
 - thread-per-task policy issue; 116
 - models
 - hardware architecture; 338
 - JMM; 337–352
 - reordering
 - operations; 339
 - shared memory multiprocessors; 338–339
 - synchronization
 - performance impact of; 230–231
 - thread pool size impact; 171
 - visibility; 33–39
 - ReentrantLock effect; 277
 - synchronized effect; 33
- Michael-Scott nonblocking queue**; 332–335
- missed signals**; 301, 301
 - See also* liveness;
 - as single notification risk; 302
- model(s)/modeling**
 - See also* Java Memory Model (JMM); MVC (model-view-controller) design pattern; representation; views;
 - event handling
 - model-view objects; 195_{fg}
 - memory
 - hardware architecture; 338
 - JMM; 337–352
 - model-view-controller pattern
 - deadlock risk; 190
 - vehicle tracking example; 61
 - programming
 - sequential; 2
 - shared data
 - See also* page renderer examples; in GUI applications; 198–202
 - simplicity
 - threads benefit for; 3
 - split data models; 201, 201–202
 - Swing event handling; 194
 - three-tier application
 - performance vs. scalability; 223
 - versioned data model; 201
- modification**
 - concurrent
 - synchronized collection problems with; 82
 - frequent need for
 - copy-on-write collection not suited for; 87
- monitor(s)**
 - See also* Java monitor pattern;
 - locks
 - See* intrinsic locks;
- monitoring**
 - See also* instrumentation; performance; scalability; testing; tools;
 - CPU utilization; 240–241
 - performance; 221–245
 - ThreadPoolExecutor hooks for; 179
 - tools
 - for quality assurance; 273
- monomorphic call transformation**
 - JVM use; 268_{fn}
- mpstat application**; 240
 - See also* measurement; tools;
- multiple-reader, single-writer locking**
 - and lock contention reduction; 239
 - read-write locks; 286–289
- multiprocessor systems**
 - See also* concurrent/concurrency;
 - shared memory
 - memory models; 338–339
 - threads use of; 3
- multithreaded**
 - See also* safety; single-thread(ed); thread(s);
 - GUI frameworks
 - issues with; 189–190
- multivariable invariants**
 - and atomic variables; 325–326
 - atomicity requirements; 57, 67–68
 - dependencies, thread safety issues; 24
 - locking requirements for; 29

- preservation of, as thread safety requirement; 24
- mutable; 15**
 - objects
 - safe publication of; 54
 - state
 - managing access to, as thread safety goal; 15
- mutexes (mutual exclusion locks); 25**
 - binary semaphore use as; 99
 - intrinsic locks as; 25
 - ReentrantLock capabilities; 277
- MVC (model-view-controller) pattern**
 - deadlock risks; 190
 - vehicle tracking example use of; 61
- N**
- narrowing**
 - lock scope
 - as lock contention reduction strategy; 233–235
- native code**
 - finalizer use and limitations; 165
- navigation**
 - as compound action
 - in collection operations; 79
- newTaskFor; 126_{li}**
 - encapsulating non-standard cancellation; 148
- nonatomic 64-bit operations; 36**
- nonblocking algorithms; 319, 329, 329–336**
 - backoff importance for; 231_{fn}
 - synchronization; 319–336
 - SynchronousQueue; 174_{fn}
 - thread-safe counter use; 322–324
- nonfair semaphores**
 - advantages of; 265
- notification; 302–304**
 - See also* blocking; condition, queues; event(s); listeners; notify; notifyAll; sleeping; wait(s); waking up;
 - completion
 - of long-running GUI task; 198
 - conditional; 303
 - as optimization; 303
 - use; 304_{li}
 - errors
 - as concurrency bug pattern; 272
 - event notification systems
 - copy-on-write collection advantages; 87
- notify**
 - as optimization; 303
 - efficiency of; 298_{fn}
 - missed signal risk; 302
 - notifyAll vs.; 302
 - subclassing safety issues
 - documentation importance; 304
 - usage guidelines; 303
- notifyAll**
 - notify vs.; 302
- @NotThreadSafe; 6, 353**
- NPTL threads package**
 - Linux use; 4_{fn}
- nulling out memory references**
 - testing use; 257
- O**
- object(s)**
 - See also* resource(s);
 - composing; 55–78
 - condition
 - explicit; 306–308
 - effectively immutable; 53
 - guarded; 54
 - immutable; 46
 - initialization safety; 51
 - publication using volatile; 48–49
 - mutable
 - safe publication of; 54
 - pools
 - appropriate uses; 241_{fn}
 - bounded, semaphore management of; 99
 - disadvantages of; 241
 - serial thread confinement use; 90
 - references
 - and stack confinement; 44
 - sharing; 33–54
 - state; 55
 - components of; 55
 - Swing
 - thread-confinement; 191–192
- objects**
 - guarded; 28
- open calls; 211, 211–213**
 - See also* encapsulation;
- operating systems**
 - concurrency use
 - historical role; 1

operations

64-bit, nonatomic nature of; 36
state-dependent; 57

optimistic concurrency management

See atomic variables; CAS; nonblocking algorithms;

optimization**compiler**

as performance testing pitfall;
268–270

JVM

pitfalls; 38_{fn}
strategies; 286

lock contention

impact; 231
reduction strategies; 232–242

performance

Amdahl's law; 225–229
premature, avoidance of; 223
questions about; 224
scalability requirements vs.; 222

techniques

See also atomic variables; nonblocking synchronization;
condition queues use; 297
conditional notification; 303

order(ing)

See also reordering; synchronization;
acquisition, in `ReentrantReadWriteLock`; 317_{fn}

checksums

safety testing use; 253

FIFO

impact of caller state dependence handling on; 294_{fn}

lock

deadlock risks; 206–213
dynamic deadlock risks; 207–210
inconsistent, as multithreaded
GUI framework problem; 190

operation

synchronization role; 35

partial; 340_{fn}

happens-before, JMM definition;
340–342
happens-before, piggybacking;
342–344
happens-before, publication consequences; 244–249

performance-based alterations in
thread safety risks; 7

total

synchronization actions; 341

orderly shutdown; 164**OutOfMemoryError**

unbounded thread creation risk; 116

overhead

See also CPU utilization; measurement; performance;

impact of

See performance; throughput;

reduction

See nonblocking algorithms; optimization; thread(s), pools;

sources

See blocking/blocks; contention;
context switching; multi-threaded environments;
safety; suspension; synchronization; thread(s), lifecycle;

ownership

shared; 58

split; 58

state

class design issues; 57–58

thread; 150

P**page renderer examples**

See also model(s)/modeling, shared data;

heterogeneous task partitioning; 127–129

parallelism analysis; 124–133

sequential execution; 124–127

parallelizing/parallelism

See also concurrent/concurrency;

Decorator pattern;

application analysis; 123–133

heterogeneous tasks; 127–129

iterative algorithms

barrier management of; 99

puzzle-solving framework; 183–188

recursive algorithms; 181–188

serialization vs.

Amdahl's law; 225–229

task-related decomposition; 113

thread-per-task policy; 115

partial ordering; 340_{fn}

happens-before

and publication; 244–249

JMM definition; 340

- piggybacking; 342–344
 - partitioning**
 - as parallelizing strategy; 101
 - passivation**
 - impact on HttpSession thread-safety requirements; 58_{fn}
 - perfbat application**
 - See also measurement; tools;
 - CPU performance measure; 261
 - performance measurement use; 225
 - perfmom application; 240**
 - See also measurement; tools;
 - I/O measurement; 240
 - performance measurement use; 230
 - performance; 8, 221, 221–245**
 - See also concurrent/concurrency;
 - liveness; scalability; throughput; utilization;
 - and heterogeneous tasks; 127
 - and immutable objects; 48_{fn}
 - and resource management; 119
 - atomic variables
 - locking vs.; 326–329
 - cache implementation issues; 103
 - composition functionality extension mechanism; 74_{fn}
 - costs
 - thread-per-task policy; 116
 - fair vs. nonfair locking; 284
 - hazards
 - See also overhead; priority(s), inversion;
 - JVM interaction with hardware reordering; 34
 - liveness
 - in servlets with state; 29–32
 - locking
 - during iteration impact on; 83
 - measurement of; 222
 - See also capacity; efficiency; latency; scalability; service time; throughput;
 - locks vs. atomic variables; 326–329
 - memory barrier impact on; 230
 - notifyAll impact on; 303
 - optimization
 - See also CPU utilization; piggybacking;
 - Amdahl's law; 225–229
 - bad practices; 348–349
 - CAS-based operations; 323
 - reduction strategies; 232–242
 - page renderer example with CompletionService
 - improvements; 130
 - producer-consumer pattern advantages; 90
 - read-write lock advantages; 286–289
 - ReentrantLock vs. intrinsic locks; 282–286
 - requirements
 - thread-safety impact; 16
 - scalability vs.; 222–223
 - issues, three-tier application model as illustration; 223
 - lock granularity reduction; 239
 - object pooling issues; 241
 - sequential event processing; 191
 - simplicity vs.
 - in refactoring synchronized blocks; 34
 - synchronized block scope; 30
 - SynchronousQueue; 174_{fn}
 - techniques for improving
 - atomic variables; 319–336
 - nonblocking algorithms; 319–336
 - testing; 247–274
 - criteria; 248
 - goals; 260
 - pitfalls, avoiding; 266–270
 - thread pool
 - size impact; 170
 - tuning; 171–179
 - thread safety hazards for; 8
 - timing and ordering alterations for thread safety risks; 7
 - tradeoffs
 - evaluation of; 223–225
- permission**
 - codebase
 - and custom thread factory; 177
- permits; 98**
 - See also semaphores;
- pessimistic concurrency management**
 - See lock(ing), exclusive;
- piggybacking; 344**
 - on synchronization; 342–344
- point(s)**
 - barrier; 99
 - cancellation; 140

poison

message; 219
 See also livelock;
 pill; 155, 155–156
 See also lifecycle; shutdown;
 CrawlerThread; 157_{li}
 IndexerThread; 157_{li}
 IndexingService; 156_{li}
 unbounded queue shutdown
 with; 155

policy(s)

See also design; documentation;
 guidelines; protocol(s);
 strategies;

application

thread pool advantages; 120

cancellation; 136

for tasks, thread interruption
 policy relationship to; 141
 interruption advantages as im-
 plementation strategy; 140

execution

design, influencing factors; 167
 Executors, for ThreadPoolExec-
 utor configuration; 171
 implicit couplings between tasks
 and; 167–170
 parallelism analysis for; 123–133
 task; 118–119
 task, application performance
 importance; 113

interruption; 141, 141–142

saturation; 174–175

security

custom thread factory handling;
 177

sequential

task execution; 114

sharing objects; 54

synchronization; 55

requirements, impact on class
 extension; 71

requirements, impact on class
 modification; 71

shared state requirements for; 28

task scheduling

sequential; 114

thread pools; 117

thread pools advantages over
 thread-per-task; 121

thread-per-task; 115

thread confinement; 43

polling

blocking state-dependent actions;

295–296

for interruption; 143

lock acquisition; 279

pool(s)

See also resource(s);

object

appropriate uses; 241_{fn}

bounded, semaphore use; 99

disadvantages of; 241

serial thread confinement use; 90

resource

semaphore use; 98–99

thread pool size impact; 171

size

core; 171, 172_{fn}

maximum; 172

thread; 119–121

adding statistics to; 179

application; 167–188

as producer-consumer design; 88

as thread resource management
 mechanism; 117

callback use in testing; 258

combined with work queues, in
 Executor framework; 119

configuration post-construction
 manipulation; 177–179

configuring task queue; 172–174

creating; 120

deadlock risks; 215

factory methods for; 171

sizing; 170–171

uncaught exception handling;
 163

portal

timed task example; 131–133

postconditions

See also invariant(s);

preservation of

mechanisms and synchroniza-
 tion policy role; 55–56

thread safety role; 17

precondition(s)

See also dependencies, state; invari-
 ant(s);

condition predicate as; 299

failure

bounded buffer handling of; 292

- propagation to callers; 292–295
- state-based
 - in state-dependent classes; 291
 - management; 57
- predictability**
 - See also* responsiveness;
 - measuring; 264–266
- preemptive interruption**
 - deprecation reasons; 135_{fn}
- presentation**
 - See* GUI;
- primitive**
 - local variables, safety of; 44
 - wrapper classes
 - atomic scalar classes vs.; 325
- priority(s)**
 - inversion; 320
 - avoidance, nonblocking algorithm advantages; 329
 - thread
 - manipulation, liveness hazards; 218
 - when to use; 219
- PriorityBlockingQueue**; 89
 - thread pool use of; 173–174
- PriorityQueue**; 85
- private**
 - constructor capture idiom; 69_{fn}
 - locks
 - Java monitor pattern vs.; 61
- probability**
 - deadlock avoidance use with timed and polled locks; 279
 - determinism vs.
 - in concurrent programs; 247
- process(es); 1**
 - communication mechanisms; 1
 - lightweight
 - See* threads;
 - threads vs.; 2
- producer-consumer pattern**
 - and Executor functionality
 - in CompletionService; 129
 - blocking queues and; 87–92
 - bounded buffer use; 292
 - control flow coordination
 - blocking queues use; 94
 - Executor framework use; 117
 - pathological waiting conditions; 300_{fn}
 - performance testing; 261
 - safety testing; 252
 - work stealing vs.; 92
- profiling**
 - See also* measurement;
 - JVM use; 320_{fn}
 - tools
 - lock contention detection; 240
 - performance measurement; 225
 - quality assurance; 273
- programming**
 - models
 - sequential; 2
- progress indication**
 - See also* GUI;
 - in long-running GUI task; 198
- propagation**
 - of interruption exception; 142
- protocol(s)**
 - See also* documentation; policy(s); strategies;
 - entry and exit
 - state-dependent operations; 306
 - lock acquisition
 - instance confinement use; 60
 - locking
 - shared state requirements for; 28
 - race condition handling; 21
 - thread confinement
 - atomicity preservation with open calls; 213
- pthreads (POSIX threads)**
 - default locking behavior; 26_{fn}
- publication; 39**
 - See also* confinement; documentation; encapsulation; sharing;
 - escape and; 39–42
 - improper; 51, 50–51
 - JMM support; 244–249
 - of immutable objects
 - volatile use; 48–49
 - safe; 346
 - idioms for; 52–53
 - in task creation; 126
 - of mutable objects; 54
 - serial thread confinement use; 90
 - safety guidelines; 49–54
 - state variables
 - safety, requirements for; 68–69
 - unsafe; 344–346

put-if-absent operation

See also compound actions;
as compound action
atomicity requirements; 71
concurrent collection support for; 84

puzzle solving framework

as parallelization example; 183–188

Q**quality assurance**

See also testing;
strategies; 270–274

quality of service

measuring; 264
requirements
and task execution policy; 119

Queue; 84–85**queue(s)**

See also data structures;
blocking; 87–94
cancellation, problems; 138
cancellation, solutions; 140
CompletionService as; 129
producer-consumer pattern and;
87–92
bounded
saturation policies; 174–175
condition; 297
blocking state-dependent operations use; 296–308
intrinsic; 297
intrinsic, disadvantages of; 306
FIFO; 89
implementations
serialization differences; 227
priority-ordered; 89
synchronous
design constraints; 89
thread pool use of; 173
task
thread pool use of; 172–174
unbounded
poison pill shutdown; 156
using; 298
work
in thread pools; 88, 119

R**race conditions; 7, 20–22**

See also concurrent/concurrency,
errors; data, race; time/timing;
avoidance
immutable object use; 48
in thread-based service shutdown; 153
in GUI frameworks; 189
in web crawler example
idempotence as mitigating circumstance; 161

random(ness)

livelock resolution use; 219
pseudorandom number generation
scalability; 326–329
test data generation use; 253

reachability

publication affected by; 40

read-modify-write operation

See also compound actions;
as non-atomic operation; 20

read-write locks; 286–289**ReadWriteLock; 286_{li}**

exclusive locking vs.; 239

reaping

See termination;

reclosable thread gate; 304**recovery, deadlock**

See deadlock, recovery;

recursion

See also control flow; iterators/iteration;

intrinsic lock acquisition; 237_{fm}

parallelizing; 181–188

See also Decorator pattern;

reentrant/reentrancy; 26

and read-write locks; 287

locking semantics; 26–27

ReentrantLock capabilities; 278

per-thread lock acquisition; 26–27

ReentrantLock; 277–282

ReentrantLock

AQS use; 314–315

intrinsic locks vs.

performance; 282–286

Lock implementation; 277–282

random number generator using;

327_{li}

Semaphore relationship with; 308

ReentrantReadWriteLock

AQS use; 316–317
reentrant locking semantics; 287

references

stack confinement precautions; 44

reflection

atomic field updater use; 335

rejected execution handler

ExecutorService post-termination
task handling; 121
puzzle-solving framework; 187

RejectedExecutionException

abort saturation policy use; 174
post-termination task handling; 122
puzzle-solving framework use; 187

RejectedExecutionHandler

and saturation policy; 174

release

AQS synchronizer operation; 311
lock
in hand-over-hand locking; 282
intrinsic locking disadvantages;
278
preferences in read-write lock
implementation; 287
unreleased lock bug pattern; 271

permit

semaphore management; 98

remote objects

thread safety concerns; 10

remove-if-equal operation

as atomic collection operation; 86

reordering; 34

See also deadlock; optimization; or-
der(ing); ordering; synchro-
nization; time/timing;

initialization safety limitation; 350

memory

barrier impact on; 230

operations; 339

volatile variables warning; 38

replace-if-equal operation

as atomic collection operation; 86

representation

See also algorithm(s); design; docu-
mentation; state(s);

activities

tasks use for; 113

algorithm design role; 104

result-bearing tasks; 125

task

lifecycle, Future use for; 125

Runnable use for; 125

with Future; 126

thread; 150

request**interrupt**

strategies for handling; 140

requirements

See also constraints; design; docu-
mentation; performance;

concrete

importance for effective perfor-
mance optimization; 224

concurrency testing

TCK example; 250

determination

importance of; 223

independent state variables; 66–67

performance

Amdahl's law insights; 229

thread-safety impact; 16

synchronization

synchronization policy compo-
nent; 56–57

synchronization policy documenta-
tion; 74–77

resource exhaustion, preventing

bounded queue use; 173

execution policy as tool for; 119

testing strategies; 257

thread pool sizing risks; 170

resource(s)

See also CPU; instrumentation; mem-
ory; object(s); pool(s); utiliza-
tion;

accessing

as long-running GUI task; 195

bound; 221

consumption

thread safety hazards for; 8

deadlocks; 213–215

depletion

thread-per-task policy issue; 116

increase

scalability relationship to; 222

leakage

testing for; 257

management

See also instrumentation; testing;
dining philosophers prob-
lem;

- blocking queue advantages; 88
- execution policy as tool for; 119
- Executor framework use; 117
- finalizer use and limitations; 165
- graceful degradation, saturation
 - policy advantages; 175
- long-running task handling; 170
- saturation policies; 174–175
- single-threaded task execution
 - disadvantages; 114
- testing; 257
- thread pools; 117
- thread pools, advantages; 121
- thread pools, tuning; 171–179
- thread-per-task policy disadvantages; 116
- threads, keep-alive time impact
 - on; 172
- timed task handling; 131
- performance
 - analysis, monitoring, and improvement; 221–245
- pools
 - semaphore use; 98–99
 - thread pool size impact; 171
- utilization
 - Amdahl's law; 225
 - as concurrency motivation; 1
- response-time-sensitive tasks**
 - execution policy implications; 168
- responsiveness**
 - See also* deadlock; GUI; livelock; liveness; performance;
 - as performance testing criteria; 248
 - condition queues advantages; 297
 - efficiency vs.
 - polling frequency; 143
 - interruption policy
 - InterruptedException advantages; 142
 - long-running tasks
 - handling; 170
 - measuring; 264–266
 - page renderer example with CompletionService
 - improvements; 130
 - performance
 - analysis, monitoring, and improvement; 221–245
 - poor
 - causes and resolution of; 219
 - safety vs.
 - graceful vs. abrupt shutdown; 153
 - sequential execution limitations; 124
 - server applications
 - importance of; 113
 - single-threaded execution disadvantages; 114
 - sleeping impact on; 295
 - thread
 - pool tuning, ThreadPoolExecutor use; 171–179
 - request overload impact; 173
 - safety hazards for; 8
- restoring interruption status; 142**
- result(s)**
 - bearing latches
 - puzzle framework use; 184
 - cache
 - building; 101–109
 - Callable handling of; 125
 - Callable use instead of Runnable; 95
 - dependencies
 - task freedom from, importance of; 113
 - Future handling of; 125
 - handling
 - as serialization source; 226
 - irrelevancy
 - as cancellation reason; 136, 147
 - non-value-returning tasks; 125
 - Runnable limitations; 125
- retry**
 - randomness, in livelock resolution; 219
- return values**
 - Runnable limitations; 125
- reuse**
 - existing thread-safe classes
 - strategies and risks; 71
- RMI (Remote Method Invocation)**
 - thread use; 9, 10
 - safety concerns and; 10
 - threads benefits for; 4
- robustness**
 - See also* fragility; safety;
 - blocking queue advantages; 88
 - InterruptedException advantages; 142
 - thread pool advantages; 120

rules

See also guidelines; policy(s); strategies;

happens-before; 341

Runnable

handling exceptions in; 143

task representation limitations; 125

running

ExecutorService state; 121

FutureTask state; 95

runtime

timing and ordering alterations by
thread safety risks; 7

RuntimeException

as thread death cause; 161

Callable handling; 98

catching

disadvantages of; 161

S**safety**

See also encapsulation; immutable
objects; synchronization;
thread(s), confinement;

cache implementation issues; 104
initialization

guarantees for immutable ob-
jects; 51

idioms for; 346–348

JMM support; 349–350

liveness vs.; 205–220

publication

idioms for; 52–53

in task creation; 126

of mutable objects; 54

responsiveness vs.

as graceful vs. abrupt shutdown;
153

split ownership concerns; 58

subclassing issues; 304

testing; 252–257

goals; 247

tradeoffs

in performance optimization
strategies; 223–224

untrusted code behavior

protection mechanisms; 161

saturation

policies; 174–175

scalability; 222, 221–245

algorithm

comparison testing; 263–264

Amdahl's law insights; 229

as performance testing criteria; 248

client-side locking impact on; 81

concurrent collections vs. synchro-
nized collections; 84

ConcurrentHashMap advantages; 85,
242

CPU utilization monitoring; 240–241

enhancement

reducing lock contention; 232–
242

heterogeneous task issues; 127

hot field impact on; 237

intrinsic locks vs. ReentrantLock

performance; 282–286

lock scope impact on; 233

locking during iteration risk of; 83

open call strategy impact on; 213

performance vs.; 222–223

lock granularity reduction; 239

object pooling issues; 241

three-tier application model as
illustration; 223

queue implementations

serialization differences; 227

result cache

building; 101–109

serialization impact on; 228

techniques for improving

atomic variables; 319–336

nonblocking algorithms; 319–336

testing; 261

thread safety hazards for; 8

under contention

as AQS advantage; 311

ScheduledThreadPoolExecutor

as Timer replacement; 123

scheduling

overhead

performance impact of; 222

priority manipulation risks; 218

tasks

sequential policy; 114

thread-per-task policy; 115

threads as basic unit of; 3

work stealing

dequeues and; 92

scope/scoped*See also* granularity;

containers

thread safety concerns; 10

contention

atomic variable limitation of; 324

escaping

publication as mechanism for; 39

lock

narrowing, as lock contention

reduction strategy; 233–235

synchronized block; 30

search

depth-first

breadth-first search vs.; 184

parallelization of; 181–182

security policies

and custom thread factory; 177

Selector

non-interruptable blocking; 148

semantics*See also* documentation; representation;

atomic arrays; 325

binary semaphores; 99

final fields; 48

of interruption; 93

of multithreaded environments

ThreadLocal variable considerations; 46

reentrant locking; 26–27

ReentrantLock capabilities; 278

ReentrantReadWriteLock capabilities; 287

undefined

of Thread.yield; 218

volatile; 39

weakly consistent iteration; 85

within-thread-as-if-serial; 337

Semaphore; 98

AQS use; 315–316

example use; 100_{li}, 176_{li}, 249_{li}

in BoundedBuffer example; 248

saturation policy use; 175

similarities to ReentrantLock; 308

state-based precondition management with; 57

semaphores; 98, 98–99

as coordination mechanism; 1

binary

mutex use; 99

counting; 98

permits, thread relationships;

248_{fn}SemaphoreOnLock example; 310_{li}

fair vs. nonfair

performance comparison; 265

nonfair

advantages of; 265

sendOnSharedLine example; 281_{li}**sequential/sequentiality***See also* concurrent/concurrency;

asynchrony vs.; 2

consistency; 338

event processing

in GUI applications; 191

execution

of tasks; 114

parallelization of; 181

orderly shutdown strategy; 164

page renderer example; 124–127

programming model; 2

task execution policy; 114

tests, value in concurrency testing;
250

threads simulation of; 4

serialized/serialization

access

object serialization vs.; 27_{fn}

timed lock use; 279

WorkerThread; 227_{li}

granularity

throughput impact; 228

impact on HttpSession thread-

safety requirements; 58_{fn}

parallelization vs.

Amdahl's law; 225–229

scalability impact; 228

serial thread confinement; 90, 90–92

sources

identification of, performance
impact; 225**server***See also* client;

applications

context switch reduction; 243–
244

design issues; 113

service(s)*See also* applications; frameworks;

logging

- as thread-based service example;
150–155
- shutdown
 - as cancellation reason; 136
- thread-based
 - stopping; 150–161
- servlets**
 - framework
 - thread safety requirements; 10
 - threads benefits for; 4
 - stateful, thread-safety issues
 - atomicity; 19–23
 - liveness and performance; 29–32
 - locking; 23–29
 - stateless
 - as thread-safety example; 18–19
- session-scoped objects**
 - thread safety concerns; 10
- set(s)**
 - See also* collection(s);
 - BoundedHashSet example; 100_{ii}
 - CopyOnWriteArraySet
 - as synchronized Set replacement; 86
 - safe publication use; 52
 - PersonSet example; 59_{ii}
 - SortedSet
 - ConcurrentSkipListSet as concurrent replacement; 85
 - TreeSet
 - ConcurrentSkipListSet as concurrent replacement; 85
- shared/sharing; 15**
 - See also* concurrent/concurrency;
publication;
 - data
 - See also* page renderer examples;
access coordination, explicit lock
use; 277–290
 - models, GUI application handling; 198–202
 - synchronization costs; 8
 - threads advantages vs. processes; 2
 - data structures
 - as serialization source; 226
 - memory
 - as coordination mechanism; 1
 - memory multiprocessors
 - memory models; 338–339
 - mutable objects
 - guidelines; 54
 - objects; 33–54
 - split data models; 201–202
 - state
 - managing access to, as thread
safety goal; 15
 - strategies
 - ExecutorCompletionService
use; 130
 - thread
 - necessities and dangers in GUI
applications; 189–190
 - volatile variables as mechanism for;
38
- shutdown**
 - See also* lifecycle;
 - abrupt
 - JVM, triggers for; 164
 - limitations; 158–161
 - as cancellation reason; 136
 - cancellation and; 135–166
 - ExecutorService state; 121
 - graceful vs. abrupt tradeoffs; 153
 - hooks; 164
 - in orderly shutdown; 164–165
 - JVM; 164–166
 - and daemon threads; 165
 - of thread-based services; 150–161
 - orderly; 164
 - strategies
 - lifecycle method encapsulation;
155
 - logging service example; 150–
155
 - one-shot execution service exam-
ple; 156–158
 - support
 - LifecycleWebServer example;
122_{ii}
- shutdown; 121**
 - logging service shutdown alterna-
tives; 153
- shutdownNow; 121**
 - limitations; 158–161
 - logging service shutdown alterna-
tives; 153
- side-effects**
 - as serialization source; 226
 - freedom from
 - importance for task indepen-
dence; 113

- synchronized Map implementations
 - not available from Concurrent-HashMap; 86
- signal**
 - ConditionBoundedBuffer example; 308
- signal handlers**
 - as coordination mechanism; 1
- simplicity**
 - See also* design;
 - Java monitor pattern advantage; 61
 - of modeling
 - threads benefit for; 3
 - performance vs.
 - in refactoring synchronized blocks; 34
- simulations**
 - barrier use in; 101
- single notification**
 - See* notify; signal;
- single shutdown hook**
 - See also* hook(s);
 - orderly shutdown strategy; 164
- single-thread(ed)**
 - See also* thread(s); thread(s), confinement;
 - as Timer restriction; 123
 - as synchronization alternative; 42–46
 - deadlock avoidance advantages; 43_{fn}
 - subsystems
 - GUI implementation as; 189–190
 - task execution
 - disadvantages of; 114
 - executor use, concurrency prevention; 172, 177–178
- Singleton pattern**
 - ThreadLocal variables use with; 45
- size(ing)**
 - See also* configuration; instrumentation;
 - as performance testing goal; 260
 - bounded buffers
 - determination of; 261
 - heterogeneous tasks; 127
 - pool
 - core; 171, 172_{fn}
 - maximum; 172
 - task
 - appropriate; 113
 - thread pools; 170–171
- sleeping**
 - blocking state-dependent actions
 - blocking state-dependent actions; 295–296
- sockets**
 - as coordination mechanism; 1
 - synchronous I/O
 - non-interruptable blocking reason; 148
- solutions**
 - See also* interruption; results; search; termination;
- SortedMap**
 - ConcurrentSkipListMap as concurrent replacement; 85
- SortedSet**
 - ConcurrentSkipListSet as concurrent replacement; 85
- space**
 - state; 56
- specification**
 - See also* documentation;
 - correctness defined in terms of; 17
- spell checking**
 - as long-running GUI task; 195
- spin-waiting; 232, 295**
 - See also* blocking/blocks; busy-waiting;
 - as concurrency bug pattern; 273
- split(ing)**
 - data models; 201, 201–202
 - lock; 235
 - Amdahl's law insights; 229
 - as lock granularity reduction strategy; 235
 - ServerStatus examples; 236_{li}
 - ownership; 58
- stack(s)**
 - address space
 - thread creation constraint; 116_{fn}
 - confinement; 44, 44–45
 - See also* confinement; encapsulation;
 - nonblocking; 330
 - size
 - search strategy impact; 184
 - trace
 - thread dump use; 216
- stale data; 35–36**
 - improper publication risk; 51
 - race condition cause; 20_{fn}

- starvation**; 218, 218
 - See also* deadlock; livelock; liveness; performance;
 - as liveness failure; 8
 - locking during iteration risk of; 83
 - thread starvation deadlock; 169, 168–169
 - thread starvation deadlocks; 215
- state(s)**; 15
 - See also* atomic/atomicity; encapsulation; lifecycle; representation; safety; visibility;
 - application
 - framework threads impact on; 9
 - code vs.
 - thread-safety focus; 17
 - dependent
 - classes; 291
 - classes, building; 291–318
 - operations; 57
 - operations, blocking strategies; 291–308
 - operations, condition queue handling; 296–308
 - operations, managing; 291
 - task freedom from, importance of; 113
 - encapsulation
 - breaking, costs of; 16–17
 - invariant protection use; 83
 - synchronizer role; 94
 - thread-safe class use; 23
 - lifecycle
 - ExecutorService methods; 121
 - locks control of; 27–29
 - logical; 58
 - management
 - AQS-based synchronizer operations; 311
 - managing access to
 - as thread safety goal; 15
 - modification
 - visibility role; 33
 - mutable
 - coordinating access to; 110
 - object; 55
 - components of; 55
 - remote and thread safety; 10
 - ownership
 - class design issues; 57–58
 - servlets with
 - thread-safety issues, atomicity; 19–23
 - thread-safety issues, liveness and performance concerns; 29–32
 - thread-safety issues, locking; 23–29
 - space; 56
 - stateless servlet
 - as thread-safety example; 18–19
 - task
 - impact on Future.get; 95
 - intermediate, shutdown issues; 158–161
 - transformations
 - in puzzle-solving framework example; 183–188
 - transition constraints; 56
 - variables
 - condition predicate use; 299
 - independent; 66, 66–67
 - independent, lock splitting; 235
 - safe publication requirements; 68–69
- stateDependentMethod** example; 301_{ii}
- static**
 - initializer
 - safe publication mechanism; 53, 347
- static analysis** tools; 271–273
- statistics gathering**
 - See also* instrumentation;
 - adding to thread pools; 179
 - ThreadPoolExecutor hooks for; 179
- status**
 - flag
 - volatile variable use with; 38
 - interrupted; 138
 - thread
 - shutdown issues; 158
- strategies**
 - See also* design; documentation; guidelines; policy(s); representation;
 - atomic variable use; 34
 - cancellation
 - Future use; 145–147
 - deadlock avoidance; 208, 215–217
 - delegation
 - vehicle tracking example; 64
 - design

- interruption policy; 93
 - documentation use
 - annotations value; 6
 - end-of-lifecycle management; 135–166
 - InterruptedException handling; 93
 - interruption handling; 140, 142–150
 - Future use; 146
 - lock splitting; 235
 - locking
 - ConcurrentHashMap advantages; 85
 - monitor
 - vehicle tracking example; 61
 - parallelization
 - partitioning; 101
 - performance improvement; 30
 - program design order
 - correctness then performance; 16
 - search
 - stack size impact on; 184
 - shutdown
 - lifecycle method encapsulation; 155
 - logging service example; 150–155
 - one-shot execution service example; 156–158
 - poison pill; 155–156
 - split ownership safety; 58
 - thread safety delegation; 234–235
 - thread-safe class extension; 71
- stream classes**
 - client-side locking with; 150_{fn}
 - thread safety; 150
- String**
 - immutability characteristics; 47_{fn}
- striping**
 - See also* contention;
 - lock; 237, 237
 - Amdahl's law insights; 229
 - ConcurrentHashMap use; 85
- structuring**
 - thread-safe classes
 - object composition use; 55–78
- subclassing**
 - safety issues; 304
- submit, execute vs.**
 - uncaught exception handling; 163
- suspension, thread**
 - costs of; 232, 320
 - elimination by CAS-based concurrency mechanisms; 321
 - Thread.suspend, deprecation reasons; 135_{fn}
- swallowing interrupts**
 - as discouraged practice; 93
 - bad consequences of; 140
 - when permitted; 143
- Swing**
 - See also* GUI;
 - listeners
 - single-thread rule exceptions; 192
 - methods
 - single-thread rule exceptions; 191–192
 - thread
 - confinement; 42
 - confinement in; 191–192
 - use; 9
 - use, safety concerns and; 10–11
 - untrusted code protection mechanisms in; 162
- SwingWorker**
 - long-running GUI task support; 198
- synchronization/synchronized; 15**
 - See also* access; concurrent/concurrency; lock(ing); safety;;
 - allocation advantages vs.; 242
 - bad practices
 - double-checked locking; 348–349
 - blocks; 25
 - Java objects as; 25
 - cache implementation issues; 103
 - collections; 79–84
 - concurrent collections vs.; 84
 - problems with; 79–82
 - concurrent building blocks; 79–110
 - contended; 230
 - correctly synchronized program; 341
 - data sharing requirements for; 33–39
 - encapsulation
 - hidden iterator management through; 83
 - requirement for thread-safe classes; 18
 - 'fast path'
 - CAS-based operations vs.; 324
 - costs of; 230

- immutable objects as replacement;
52
- inconsistent
 - as concurrency bug pattern; 271
- memory
 - performance impact of; 230–231
- memory visibility use of; 33–39
- operation ordering role; 35
- piggybacking; 342–344
- policy; 55
 - documentation requirements;
74–77
 - encapsulation, client-side lock-
ing violation of; 71
 - race condition prevention with; 7
 - requirements, impact on class
extension; 71
 - requirements, impact on class
modification; 71
 - shared state requirements for; 28
- ReentrantLock capabilities; 277
- requirements
 - synchronization policy compo-
nent; 56–57
- thread safety need for; 5
- types
 - See barriers; blocking, queues;
FutureTask; latches;
semaphores;
- uncontended; 230
- volatile variables vs.; 38
- wrapper
 - client-side locking support; 73
- synchronizedList (Collections)**
 - safe publication use; 52
- synchronizer(s); 94, 94–101**
 - See also Semaphore; CyclicBarrier;
FutureTask; Exchanger;
CountDownLatch;
 - behavior and interface; 308–311
 - building
 - with AQS; 311
 - with condition queues; 291–318
- synchronous I/O**
 - non-interruptable blocking; 148
- SynchronousQueue; 89**
 - performance advantages; 174_{fn}
 - thread pool use of; 173, 174

T

- task(s); 113**
 - See also activities; event(s); lifecycle;
- asynchronous
 - FutureTask handling; 95–98
- boundaries; 113
 - parallelism analysis; 123–133
 - using ThreadLocal in; 168
- cancellation; 135–150
 - policy; 136
 - thread interruption policy rela-
tionship to; 141
- completion
 - as cancellation reason; 136
 - service time variance relation-
ship to; 264–266
- dependencies
 - execution policy implications;
167
 - thread starvation deadlock risks;
168
- execution; 113–134
 - in threads; 113–115
 - policies; 118–119
 - policies and, implicit couplings
between; 167–170
 - policies, application perfor-
mance importance; 113
 - sequential; 114
- explicit thread creation for; 115
- GUI
 - long-running tasks; 195–198
 - short-running tasks; 192–195
- heterogeneous tasks
 - parallelization limitations; 127–
129
- homogeneous tasks
 - parallelism advantages; 129
- lifecycle
 - Executor phases; 125
 - ExecutorService methods; 121
 - representing with Future; 125
- long-running
 - responsiveness problems; 170
- parallelization of
 - homogeneous vs. heteroge-
neous; 129
- post-termination handling; 121
- queues
 - management, thread pool con-
figuration issues; 172–174

- thread pool use of; 172–174
 - representation
 - Runnable use for; 125
 - with Future; 126
 - response-time sensitivity
 - and execution policy; 168
 - scheduling
 - thread-per-task policy; 115
 - serialization sources
 - identifying; 225
 - state
 - effect on Future.get; 95
 - intermediate, shutdown issues; 158–161
 - thread(s) vs.
 - interruption handling; 141
 - timed
 - handling of; 123
 - two-party
 - Exchanger management of; 101
- TCK (Technology Compatibility Kit)**
 - concurrency testing requirements; 250
- teardown**
 - thread; 171–172
- techniques**
 - See also* design; guidelines; strategies;
- temporary objects**
 - and ThreadLocal variables; 45
- terminated**
 - ExecutorService state; 121
- termination**
 - See also* cancellation; interruption; lifecycle;
 - puzzle-solving framework; 187
 - safety test
 - criteria for; 254, 257
 - thread
 - abnormal, handling; 161–163
 - keep-alive time impact on; 172
 - reasons for deprecation of; 135_{fn}
 - timed locks use; 279
- test example method; 262_{li}**
- testing**
 - See also* instrumentation; logging; measurement; monitoring; quality assurance; statistics;
 - concurrent programs; 247–274
 - deadlock risks; 210_{fn}
 - functionality
 - vs. performance tests; 260
 - liveness
 - criteria; 248
 - performance; 260–266
 - criteria; 248
 - goals; 260
 - pitfalls
 - avoiding; 266–270
 - dead code elimination; 269
 - dynamic compilation; 267–268
 - garbage collection; 266
 - progress quantification; 248
 - proving a negative; 248
 - timing and synchronization artifacts; 247
 - unrealistic code path sampling; 268
 - unrealistic contention; 268–269
 - program correctness; 248–260
 - safety; 252–257
 - criteria; 247
 - strategies; 270–274
 - testPoolExample example; 258_{li}**
 - testTakeBlocksWhenEmpty example; 252_{li}**
 - this reference**
 - publication risks; 41
 - Thread**
 - join
 - timed, problems with; 145
 - getState
 - use precautions; 251
 - interruption methods; 138, 139_{li}
 - usage precautions; 140
 - thread safety; 18, 15–32**
 - and mutable data; 35
 - and shutdown hooks; 164
 - characteristics of; 17–19
 - data models, GUI application handling; 201
 - delegation; 62
 - delegation of; 234
 - in puzzle-solving framework; 183
 - issues, atomicity; 19–23
 - issues, liveness and performance; 29–32
 - mechanisms, locking; 23–29
 - risks; 5–8
 - thread(s); 2**
 - See also* concurrent/concurrency; safety; synchronization;

- abnormal termination of; 161–163
- as instance confinement context; 59
- benefits of; 3–5
- blocking; 92
- confinement; 42, 42–46
 - See also* confinement; encapsulation;
 - ad-hoc; 43
 - and execution policy; 167
 - in GUI frameworks; 190
 - in Swing; 191–192
 - role, synchronization policy specification; 56
 - stack; 44, 44–45
 - ThreadLocal; 45–46
- cost
 - context locality loss; 8
 - context switching; 8
- costs; 229–232
- creation; 171–172
 - explicit creation for tasks; 115
 - unbounded, disadvantages; 116
- daemon; 165
- dumps; 216
 - deadlock analysis use; 216–217
 - intrinsic lock advantage over ReentrantLock; 285
 - lock contention analysis use; 240
- factories; 175, 175–177
- failure
 - uncaught exception handlers; 162–163
- forced termination
 - reasons for deprecation of; 135_{fn}
- interleaving
 - dangers of; 5–8
- interruption; 138
 - shutdown issues; 158
 - status flag; 138
- leakage; 161
 - testing for; 257
 - Timer problems with; 123
 - UncaughtExceptionHandler prevention of; 162–163
- lifecycle
 - performance impact; 116
 - thread-based service management; 150
- overhead
 - in safety testing, strategies for mitigating; 254
- ownership; 150
- pools; 119–121
 - adding statistics to; 179
 - and work queues; 119
 - application; 167–188
 - as producer-consumer design; 88
 - as thread resource management mechanism; 117
 - callback use in testing; 258
 - creating; 120
 - deadlock risks; 215
 - factory methods for; 171
 - post-construction configuration; 177–179
 - sizing; 170–171
 - task queue configuration; 172–174
- priorities
 - manipulation, liveness risks; 218
- priority
 - when to use; 219
- processes vs.; 2
- queued
 - SynchronousQueue management of; 89
- risks of; 5–8
- serial thread confinement; 90, 90–92
- services that own
 - stopping; 150–161
- sharing
 - necessities and dangers in GUI applications; 189–190
- single
 - sequential task execution; 114
- sources of; 9–11
- starvation deadlock; 169, 168–169
- suspension
 - costs of; 232, 320
 - Thread.suspend, deprecation reasons; 135_{fn}
- task
 - execution in; 113–115
 - scheduling, thread-per-task policy; 115
 - scheduling, thread-per-task policy disadvantages; 116
 - vs. interruption handling; 141
- teardown; 171–172
- termination
 - keep-alive time impact on; 172
- thread starvation deadlocks; 215

thread-local

See also stack, confinement;
computation
 role in accurate performance
 testing; 268

Thread.stop

deprecation reasons; 135_{fn}

Thread.suspend

deprecation reasons; 135_{fn}

ThreadFactory; 176_{li}

customizing thread pool with; 175

ThreadInfo

and testing; 273

ThreadLocal; 45–46

and execution policy; 168
for thread confinement; 43
risks of; 46

ThreadPoolExecutor

and untrusted code; 162
configuration of; 171–179
constructor; 172_{li}
extension hooks; 179
newTaskFor; 126_{li}, 148

@ThreadSafe; 7, 353**throttling**

as overload management mechanism; 88, 173
saturation policy use; 174
Semaphore use in BoundedExecutor
 example; 176_{li}

throughput

See also performance;
as performance testing criteria; 248
locking vs. atomic variables; 328
producer-consumer handoff
 testing; 261
queue implementations
 serialization differences; 227
server application
 importance of; 113
server applications
 single-threaded task execution
 disadvantages; 114
thread safety hazards for; 8
threads benefit for; 3

Throwable

FutureTask handling; 98

time/timing

See also deadlock; lifecycle; order/ordering; race conditions;

-based task

handling; 123
management design issues; 131–133

barrier handling based on; 99

constraints

as cancellation reason; 136
in puzzle-solving framework;
 187
interruption handling; 144–145

deadline-based waits

as feature of Condition; 307

deferred computations

design issues; 125

dynamic compilation

as performance testing pitfall;
 267

granularity

measurement impact; 264

keep-alive

thread termination impact; 172

LeftRightDeadlock example; 207_{fg}**lock acquisition**; 279**lock scope**

narrowing, as lock contention
 reduction strategy; 233–235

long-running GUI tasks; 195–198**long-running tasks**

responsiveness problem handling; 170

measuring

in performance testing; 260–263
ThreadPoolExecutor hooks for;
 179

performance-based alterations in

thread safety risks; 7

periodic tasks

handling of; 123

progress indication

for long-running GUI tasks; 198

relative vs. absolute

class choices based on; 123_{fn}

response

task sensitivity to, execution
 policy implications; 168

short-running GUI tasks; 192–195**thread timeout**

core pool size parameter impact
 on; 172_{fn}

timed locks; 215–216

- weakly consistent iteration semantics; 86
 - TimeoutException**
 - in timed tasks; 131
 - task cancellation criteria; 147
 - Timer**
 - task-handling issues; 123
 - thread use; 9
 - timesharing systems**
 - as concurrency mechanism; 2
 - tools**
 - See also* instrumentation; measurement;
 - annotation use; 353
 - code auditing
 - locking failures detected by; 28_{fn}
 - heap inspection; 257
 - measurement
 - I/O utilization; 240
 - importance for effective performance optimization; 224
 - performance; 230
 - monitoring
 - quality assurance use; 273
 - profiling
 - lock contention detection; 240
 - performance measurement; 225
 - quality assurance use; 273
 - static analysis; 271–273
 - transactions**
 - See also* events;
 - concurrent atomicity similar to; 25
 - transformations**
 - state
 - in puzzle-solving framework example; 183–188
 - transition**
 - See also* state;
 - state transition constraints; 56
 - impact on safe state variable publication; 69
 - travel reservations portal example**
 - as timed task example; 131–133
 - tree(s)**
 - See also* collections;
 - models
 - GUI application handling; 200
 - traversal
 - parallelization of; 181–182
 - TreeMap**
 - ConcurrentSkipListMap as concurrent replacement; 85
 - TreeSet**
 - ConcurrentSkipListSet as concurrent replacement; 85
 - Treiber's nonblocking stack algorithm;** 331_{li}
 - trigger(ing)**
 - See also* interruption;
 - JVM abrupt shutdown; 164
 - thread dumps; 216
 - try-catch block**
 - See also* exceptions;
 - as protection against untrusted code behavior; 161
 - try-finally block**
 - See also* exceptions;
 - and uncaught exceptions; 163
 - as protection against untrusted code behavior; 161
 - tryLock**
 - barging use; 283_{fn}
 - deadlock avoidance; 280_{li}
 - trySendOnSharedLine example;** 281_{li}
 - tuning**
 - See also* optimization;
 - thread pools; 171–179
- U**
- unbounded**
 - See also* bounded; constraints;
 - queue(s);
 - blocking waits
 - timed vs., in long-running task management; 170
 - queues
 - nonblocking characteristics; 87
 - poison pill shutdown use; 155
 - thread pool use of; 173
 - thread creation
 - disadvantages of; 116
 - uncaught exception handlers;** 162–163
 - See also* exceptions;
 - UncaughtExceptionHandler;** 163_{li}
 - custom thread class use; 175
 - thread leakage detection; 162–163
 - unchecked exceptions**
 - See also* exceptions;
 - catching
 - disadvantages of; 161

uncontended

synchronization; 230

unit tests

for BoundedBuffer example; 250

issues; 248

untrusted code behavior

See also safety;

ExecutorService code protection

strategies; 179

protection mechanisms; 161

updating

See also lifecycle;

atomic fields; 335–336

immutable objects; 47

views

in GUI tasks; 201

upgrading

read-write locks; 287

usage scenarios

performance testing use; 260

user

See also GUI;

cancellation request

as cancellation reason; 136

feedback

in long-running GUI tasks; 196_{li}

interfaces

threads benefits for; 5

utilization; 225

See also performance; resource(s);

CPU

Amdahl's law; 225, 226_{fg}

optimization, as multithreading

goal; 222

sequential execution limitations;

124

hardware

improvement strategies; 222

V**value(s)**

See result(s);

variables

See also encapsulation; state;

atomic

classes; 324–329

locking vs.; 326–329

nonblocking algorithms and;

319–336

volatile variables vs.; 39, 325–326

condition

explicit; 306–308

hoisting

as JVM optimization pitfall; 38_{fm}

local

stack confinement use; 44

multivariable invariant requirements

for atomicity; 57

state

condition predicate use; 299

independent; 66, 66–67

independent, lock splitting use

with; 235

object data stored in; 15

safe publication requirements;

68–69

ThreadLocal; 45–46

volatile; 38, 37–39

atomic variable class use; 319

atomic variable vs.; 39, 325–326

multivariable invariants prohib-

ited from; 68

variance

service time; 264

Vector

as safe publication use; 52

as synchronized collection; 79

check-then-act operations; 80_{li}, 79–

80

client-side locking management of

compound actions; 81_{li}

vehicle tracking example

delegation strategy; 64

monitor strategy; 61

state variable publication strategy;

69–71

thread-safe object composition de-

sign; 61–71

versioned data model; 201**views**

event handling

model-view objects; 195_{fg}

model-view-controller pattern

deadlock risks; 190

vehicle tracking example; 61

reflection-based

by atomic field updaters; 335

timeliness vs. consistency; 66, 70

updating

in long-running GUI task han-

dling; 201

with split data models; 201

visibility

See also encapsulation; safety; scope;

condition queue

control, explicit Condition and

Lock use; 306

guarantees

JMM specification of; 338

lock management of; 36–37

memory; 33–39

ReentrantLock capabilities; 277

synchronization role; 33

volatile reference use; 49

vmstat application

See also measurement; tools;

CPU utilization measurement; 240

performance measurement; 230

thread utilization measurement; 241

Void

non-value-returning tasks use; 125

volatile

cancellation flag use; 136

final vs.; 158_{fn}

publishing immutable objects with;

48–49

safe publication use; 52

variables; 38, 37–39

atomic variable class use; 319

atomic variable vs.; 39, 325–326

atomicity disadvantages; 320

multivariable invariants prohib-

ited from; 68

thread confinement use with; 43

W**wait(s)**

blocking

timed vs. unbounded; 170

busy-waiting; 295

condition

and condition predicate; 299

canonical form; 301_{li}

errors, as concurrency bug pat-

tern; 272

interruptible, as feature of Con-

dition; 307

uninterruptable, as feature of

Condition; 307

waking up from, condition

queue handling; 300–301

sets; 297

multiple, as feature of Condi-
tion; 307

spin-waiting; 232

as concurrency bug pattern; 273

waiting to run

FutureTask state; 95

waking up

See also blocking/blocks; condition,

queues; notify; sleep; wait;

condition queue handling; 300–301

weakly consistent iterators; 85

See also iterators/iteration;

web crawler example; 159–161**within-thread usage**

See stack, confinement;

within-thread-as-if-serial semantics;

337

work

queues

and thread pools, as producer-

consumer design; 88

in Executor framework use; 119

thread pool interaction, size tun-

ing requirements; 173

sharing

deque advantages for; 92

stealing scheduling algorithm; 92

deque and; 92

tasks as representation of; 113

wrapper(s)

factories

Decorator pattern; 60

synchronized wrapper classes

as synchronized collection

classes; 79

client-side locking support; 73