华章专业开发者丛书

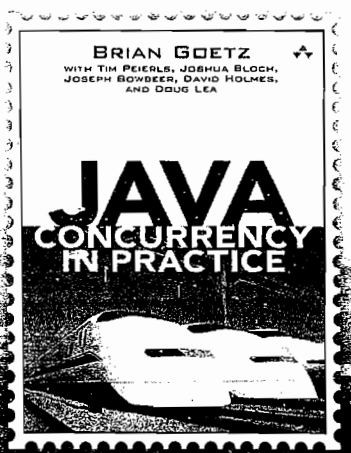
Java

并发编程实战

Java Concurrency in Practice

（美）Brian Goetz
Tim Peierls
Joshua Bloch
Joseph Bowbeer
David Holmes
Doug Lea
著

童云兰 等译



机械工业出版社
China Machine Press

本书深入浅出地介绍了Java线程和并发，是一本完美的Java并发参考手册。书中从并发性和线程安全性的基本概念出发，介绍了如何使用类库提供的基本并发构建块，用于避免并发危险、构造线程安全的类及验证线程安全的规则，如何将小的线程安全类组合成更大的线程安全类，如何利用线程来提高并发应用程序的吞吐量，如何识别可并行执行的任务，如何提高单线程子系统的响应性，如何确保并发程序执行预期任务，如何提高并发代码的性能和可伸缩性等内容，最后介绍了一些高级主题，如显式锁、原子变量、非阻塞算法以及如何开发自定义的同步工具类。

本书适合Java程序开发人员阅读。

本书版权登记号：图字：01-2011-1513

图书在版编目（CIP）数据

Java并发编程实战 /（美）盖茨（Goetz, B.）等著；童云兰等译 . —北京：机械工业出版社，2012.2
（华章专业开发者丛书）

书名原文：Java Concurrency in Practice

ISBN 978-7-111-37004-8

I. J… II. ①盖… ②童… III. JAVA语言－程序设计 IV. TP312

中国版本图书馆CIP数据核字（2011）第281977号

机械工业出版社（北京市西城区百万庄大街22号 邮政编码 100037）

责任编辑：关 敏

北京市荣盛彩色印刷有限公司印刷

2012年2月第1版第1次印刷

186mm×240mm • 19.5印张

标准书号：ISBN 978-7-111-37004-8

定价：69.00元

凡购本书，如有缺页、倒页、脱页，由本社发行部调换

客服热线：（010）88378991；88361066

购书热线：（010）68326294；88376949；68995259

投稿热线：（010）88379604

读者信箱：hzjsj@hzbook.com

对本书的赞誉

“我曾有幸在一个伟大的团队中工作，参与设计和实现在 Java 5.0 和 Java 6 等平台中新增的并发功能。现在，仍然是这个团队，将透彻地讲解这些新功能，以及关于并发的一般性概念。并发已不再只是高级用户谈论的话题，每一位 Java 开发人员都应该阅读这本书。”

——Martin Buchholz，Sun 公司的 JDK 并发大师

“在过去 30 多年时间里，计算机性能一直遵循着摩尔定律，但从现在开始，它将遵循 Amdahl 定律。编写能高效利用多处理器的代码非常具有挑战性。 在这本书中介绍的一些概念和技术，对于在当前（以及未来的）系统上编写安全的和可伸缩的代码来说都是非常有用的。”

——Doron Rajwan，Intel 公司研究人员

“如果你正在编写、设计、调试、维护以及分析多线程的 Java 程序，那么本书正是你所需要的。如果你曾对某个方法进行过同步，但却不理解其中的原因，那么你以及你的用户都有必要从头至尾仔细地读一读这本书。”

——Ted Neward，《Effective Enterprise Java》的作者

“Brian 非常清晰地阐述了并发的一些基本问题与复杂性。对于使用线程并关注程序执行性能的开发人员来说，这是一本必读的书。”

——Kirk Pepperdine，JavaPerformanceTuning.com 网站 CTO

“本书深入浅出地介绍了一些复杂的编程主题，是一本完美的 Java 并发参考手册。书中的每一页都包含了程序员日常需要应对的问题（以及相应的解决方案）。随着摩尔定律的发展趋势由提高处理器核的速度转向增加处理器核的数量，如何有效地利用并发性已变得越来越重要，本书正好介绍了这些方面的内容。”

——Cliff Click 博士，Azul Systems 公司高级软件工程师

“我对并发有着浓厚的兴趣，并且与大多数程序员相比，我或许写过更多存在线程死锁的代码，也在同步上犯了更多的错误。在介绍 Java 线程和并发等主题的众多书籍中，Brian 的这本书最具可读性，它通过循序渐进的方式将一些复杂的主题阐述得很清楚。我将本书推荐给 Java Specialists' Newsletter 的所有读者，因为它不仅有趣，而且很有用，它介绍了当前 Java 开发人员正在面对的许多问题。”

——Heinz Kabutz 博士，Java Specialists' Newsletter 的维护者

“我一直努力想使一些简单的问题变得更简单，然而本书已经简化了一个复杂但却关键的主

题：并发。这本书采用了创新的讲解方法、简单明了的风格，它注定会成为一本非常重要的书。

——Bruce Tate，《Beyond Java》的作者

这本书为 Java 开发人员在线程编程领域提供了不可多得的知识。我在读这本书时受到了极大的启发，部分原因在于它详细地介绍了 Java 中并发领域的 API，但更重要的却在于这本书以一种透彻并且易懂的方式来介绍复杂的并发知识，这是其他书籍很难媲美的。

——Bill Venners，《Inside the Java Virtual Machine》的作者

译 者 序

并发编程是Java语言的重要特性之一，在Java平台上提供了许多基本的并发功能来辅助开发多线程应用程序。然而，这些相对底层的并发功能与上层应用程序的并发语义之间并不存在一种简单而直观的映射关系。因此，如何在Java并发应用程序中正确且高效地使用这些功能就成了Java开发人员的关注重点。

本书正是为了解决这个问题而写的。书中采用循序渐进的讲解方式，从并发编程的基本理论入手，逐步介绍了在设计Java并发程序时各种重要的设计原则、设计模式以及思维模式，同时辅以丰富的示例代码作为对照和补充，使得开发人员能够更快地领悟Java并发编程的要领，围绕着Java平台的基础并发功能快速地构建大规模的并发应用程序。

全书内容由浅入深，共分为四个部分。第一部分介绍了Java并发编程的基础理论，包括线程安全性与状态对象的基础知识，如何构造线程安全的类并将多个小型的线程安全类构建成更大型的线程安全类，以及Java平台库中的一些基础并发模块；第二部分介绍了并发应用程序的构造理论，包括应用程序中并行语义的分解及其与逻辑任务的映射，任务的取消与关闭等行为的实现，以及Java线程池中的一些高级功能，此外还介绍了如何提高GUI应用程序的响应性；第三部分介绍了并发编程的性能调优，包括如何避免活跃性问题，如何提高并发代码的性能和可伸缩性以获得理想的性能，以及在测试并发代码正确性和性能时的一些实用技术；第四部分介绍了Java并发编程中的一些高级主题，包括显式锁、原子变量、非阻塞算法以及如何开发自定义的同步工具类等。

本书的特点在于注重阐述并发技术背后的理论知识，对于每种技术的介绍不仅使读者能做到“知其然”，更能做到“知其所以然”。对于希望深入研究和探索Java并发编程的读者来说，本书是非常合适的。

参与本书翻译工作的还有李杨、吴汉平、徐光景、童胜汉、陈军、胡凯、刘红、张玮、陈红、李斌、李勇涛、王海涛、周云波、彭敏才、张世锋、朱介秋、宗敬、李静、叶锦、高波、熊莉、程凤、陈娟、胡世娟、董敏、谢路阳、冯卓、李志勇、胡欢、王进等。由于译者的时间和水平有限，翻译中的疏漏和错误在所难免，还望读者和同行不吝指正。

童云兰

2011年11月于武汉

前　言

在写作本书时，对于中端桌面系统来说，多核处理器正变得越来越便宜。无独有偶，许多开发团队也注意到，在他们的项目中出现了越来越多与线程有关的错误报告。在 NetBeans 开发者网站上的最近一次公告中，一位核心维护人员注意到，为了修复与线程相关的问题，在某个类中竟然打了 14 次补丁。Dion Almaer，这位 TheServerSide 网站的前编辑，最近（在经过一番痛苦的调试过程并最终发现了一个与线程有关的错误之后）在其博客上写道，在大多数 Java 程序中充满了各种并发错误，使得程序只有在“偶然的情况下”才能正常工作。

确实，在开发、测试以及调试多线程程序时存在着巨大的困难，因为并发性错误通常并不会以某种确定的方式显现出来。当这些错误出现时，通常是在最糟糕的时刻，例如在正式产品中，或者在高负载的情况下。

当开发 Java 并发程序时，所要面对的挑战之一就是：平台提供的各种并发功能与开发人员在程序中需要的并发语义并不匹配。在 Java 语言中提供了一些底层机制，例如同步和条件等待，但在使用这些机制来实现应用级的协议与策略时必须始终保持一致。如果没有这些策略，那么在编写程序时，虽然程序看似能顺利地编译和运行，但却总会出现各种奇怪的问题。许多介绍并发的其他书籍更侧重于介绍一些底层机制和 API，而在设计级的策略和模式上叙述的不多。

Java 5.0 在 Java 并发应用程序的开发方面进展巨大，它不仅提供了一些新的高层组件，还补充了一些底层机制，从而使得无论是新手级开发人员还是专家级开发人员都能够更容易地构建并发应用程序。本书的作者都是 JCP 专家组的主要成员，也正是该专家组编写了这些新功能。本书不仅描述了这些新功能的行为和特性，还介绍了它们的底层设计模式和促使它们被添加到平台库中的应用场景。

我们的目标是向读者介绍一些设计规则和思维模式，从而使读者能够更容易也更乐意去构建正确的以及高性能的 Java 并发类和应用程序。

我们希望你能享受本书的阅读过程。

Brian Goetz
Williston，VT
2006 年 3 月

如何使用本书

为了解决在 Java 底层机制与设计级策略之间的不匹配问题，我们给出了一组简化的并发程序编写规则。专家看到这些规则会说：“嗯，这并不是完整的规则集。即使类 C 违背了规则 R，它仍然是线程安全的。”虽然在违背一些规则的情况下仍有可能编写出正确的并发程序，但这需

要对 Java 内存模型的底层细节有着深入的理解，而我们希望开发人员无须掌握这些细节就能编写出正确的并发程序。只要始终遵循这组简单的规则，就能编写出正确的并且可维护的并发程序。

我们假设读者对 Java 的基本并发机制已经有了一定程度的了解。本书并非是对并发的入门介绍——要了解这方面的内容，请参考其他书籍中有关线程的内容，例如《The Java Programming Language》（Arnold 等，2005）。此外，本书也不是介绍并发的百科全书——要了解这方面的内容，请参考《Concurrent Programming in Java》（Lea，2000）。事实上，本书提供了各种实用的设计规则，用于帮助开发人员创建安全的和高性能的并发类。在本书中相应的地方引用了以下书籍中的相关章节：《The Java Programming Language》、《Concurrent Programming in Java》、《The Java Language Specification》（Gosling 等，2005）以及《Effective Java》（Bloch，2001），并分别使用 [JPL n.m]、[CPJ n.m]、[JLS n.m] 和 [EJ Item n] 来表示它们。

在进行简要的介绍（第 1 章）之后，本书共分为四个部分：

基础知识。第一部分（第 2 章～第 5 章）重点介绍了并发性和线程安全性的基本概念，以及如何使用类库提供的基本并发构建块来构建线程安全类。在第一部分给出了一个清单，其中总结了这一部分中介绍的最重要的规则。

第 2 章与第 3 章构成了本书的基础。在这两章中给出了几乎所有用于避免并发危险、构造线程安全的类以及验证线程安全的规则。如果读者重“实践”而轻“理论”，那么可能会直接跳到第二部分，但在开始编写任何并发代码之前，一定要回来读一读这两章！

第 4 章介绍了如何将一些小的线程安全类组合成更大的线程安全类。第 5 章介绍了在平台库中提供的一些基础的并发构建模块，包括线程安全的容器类和同步工具类。

结构化并发应用程序。第二部分（第 6 章～第 9 章）介绍了如何利用线程来提高并发应用程序的吞吐量或响应性。第 6 章介绍了如何识别可并行执行的任务，以及如何在任务执行框架中执行它们。第 7 章介绍了如何使任务和线程在执行完正常工作之前提前结束。在健壮的并发应用程序与看似能正常工作的应用程序之间存在的重要差异之一就是，如何实现取消以及关闭等操作。第 8 章介绍了任务执行框架中的一些更高级特性。第 9 章介绍了如何提高单线程子系统的响应性。

活跃性、性能与测试。第三部分（第 10 章～第 12 章）介绍了如何确保并发程序执行预期的任务，以及如何获得理想的性能。第 10 章介绍了如何避免一些使程序无法执行下去的活跃性故障。第 11 章介绍了如何提高并发代码的性能和可伸缩性。第 12 章介绍了在测试并发代码的正确性和性能时可以采用的一些技术。

高级主题。第四部分（第 13 章～第 16 章）介绍了资深开发人员可能感兴趣的一些主题，包括：显式锁、原子变量、非阻塞算法以及如何开发自定义的同步工具类。

代码示例

虽然书中很多一般性的概念同样适用于 Java 5.0 之前的版本以及一些非 Java 的运行环境，但其中大多数示例代码（以及关于 Java 内存模型的所有描述）是基于 Java 5.0 或更高版本的，而且某些代码示例中还使用了 Java 6 的一些新增功能。

我们对书中的代码示例已经进行了压缩，以便减少代码量并重点突出与内容相关的部分。在本书的网站 http://www.javaconcurrencyinpractice.com 上提供了完整的代码示例、辅助示例以及勘误表。

代码示例可分为三类："好的"示例、"一般的"示例和"糟糕的"示例。"好的"示例是应该被效仿的技术。"糟糕的"示例是一定不能效仿的技术，而且还会用一个"Mr. Yuk"的图标[⊖]来表示该示例中的代码是"有害的"(参见程序清单 1)。"一般的"示例给出的技术并不一定是错的，但却是脆弱的、有风险的或是性能较差的，并且会用一个"Mr.Could Be Happier"图标来表示，如程序清单 2 所示。

程序清单 1　糟糕的链表排序方式（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);
}
```

有些读者会质疑这些"糟糕的"示例在本书中的作用，毕竟，在一本书中应该给出如何做正确的事，而不是错误的事。这些"糟糕的"示例有两个目的，它们揭示了一些常见的缺陷，但更重要的是它们示范了如何分析程序的线程安全性，而要实现这个目的，最佳的方式就是观察线程安全性是如何被破坏的。

程序清单 2　非最优方式的链表排序

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

致谢

本书诞生于 Java Community Process JSR 166 为 Java 5.0 开发 java.util.concurrent 包的过程中。还有许多人参与到 JSR 166 中，特别感谢 Martin Buchholz 将全部的工作融入到 JDK 中，并感谢 concurrency-interest 邮件列表中的所有读者对 API 草案提出的建议和反馈。

有不少来自各方面的人员都提出了建议和帮助，使得本书的内容得到了极大的充实。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、Berna 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

⊖ Mr. Yuk 是匹兹堡儿童医院的注册商标，本书获得了该商标的授权，因此可以在本书中使用。

式小组的所有成员，他们通过各种技术交流为本书提供了指导建议，使得本书更加完善。

特别感谢 Cliff Biffle、Barry Hayes、Dawid Kurzyniec、Angelika Langer、Doron Rajwan 和 Bill Venners，他们非常仔细地审阅了本书的全稿，指出了代码示例中的错误，并提出了大量的改进建议。

感谢 Katrina Avery 的编辑工作，以及 Rosemary Simpson 在非常短的时间里完成了索引生成工作。感谢 Ami Dewar 绘制的插图。

感谢 Addison-Wesley 的全体成员，他们使本书得以最终问世。Ann Sellers 启动了编写本书的项目，Greg Doench 监督并帮助本书有条不紊地完成，Elizabeth Ryan 负责本书的出版过程。

此外还要感谢许许多多的软件工程师，他们开发了本书得以依赖的各种软件，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 平台及其类库。

目　录

第三部分 活跃性、性能与测试

第四部分 高级主题

第①章

简　介

编写正确的程序很难，而编写正确的并发程序则难上加难。与串行程序相比，在并发程序中存在更多容易出错的地方。那么，为什么还要编写并发程序？线程是 Java 语言中不可或缺的重要功能，它们能使复杂的异步代码变得更简单，从而极大地简化了复杂系统的开发。此外，要想充分发挥多处理器系统的强大计算能力，最简单的方式就是使用线程。随着处理器数量的持续增长，如何高效地使用并发正变得越来越重要。

1.1　并发简史

在早期的计算机中不包含操作系统，它们从头到尾只执行一个程序，并且这个程序能访问计算机中的所有资源。在这种裸机环境中，不仅很难编写和运行程序，而且每次只能运行一个程序，这对于昂贵并且稀有的计算机资源来说也是一种浪费。

操作系统的出现使得计算机每次能运行多个程序，并且不同的程序都在单独的进程中运行：操作系统为各个独立执行的进程分配各种资源，包括内存，文件句柄以及安全证书等。如果需要的话，在不同的进程之间可以通过一些粗粒度的通信机制来交换数据，包括：套接字、信号处理器、共享内存、信号量以及文件等。

之所以在计算机中加入操作系统来实现多个程序的同时执行，主要是基于以下原因：

资源利用率。在某些情况下，程序必须等待某个外部操作执行完成，例如输入操作或输出操作等，而在等待时程序无法执行其他任何工作。因此，如果在等待的同时可以运行另一个程序，那么无疑将提高资源的利用率。

公平性。不同的用户和程序对于计算机上的资源有着同等的使用权。一种高效的运行方式是通过粗粒度的时间分片（Time Slicing）使这些用户和程序能共享计算机资源，而不是由一个程序从头运行到尾，然后再启动下一个程序。

便利性。通常来说，在计算多个任务时，应该编写多个程序，每个程序执行一个任务并在必要时相互通信，这比只编写一个程序来计算所有任务更容易实现。

在早期的分时系统中，每个进程相当于一台虚拟的冯·诺依曼计算机，它拥有存储指令和数据的内存空间，根据机器语言的语义以串行方式执行指令，并通过一组 I/O 指令与外部设备通信。对每条被执行的指令，都有相应的“下一条指令”，程序中的控制流是按照指令集的规则来确定的。当前，几乎所有的主流编程语言都遵循这种串行编程模型，并且在这些语言的规范中也都清晰地定义了在某个动作完成之后需要执行的“下一个动作”。

串行编程模型的优势在于其直观性和简单性，因为它模仿了人类的工作方式：每次只做一

件事情，做完之后再做另一件。例如，首先起床，穿上睡衣，然后下楼，喝早茶。在编程语言中，这些现实世界中的动作可以被进一步抽象为一组粒度更细的动作。例如，喝早茶的动作可以被进一步细化为：打开橱柜，挑选喜欢的茶叶，将一些茶叶倒入杯中，看看茶壶中是否有足够的水，如果没有的话加些水，将茶壶放到火炉上，点燃火炉，然后等水烧开等等。在最后一步等水烧开的过程中包含了一定程度的异步性。当正在烧水时，你可以干等着，也可以做些其他事情，例如开始烤面包（这是另一个异步任务）或者看报纸，同时留意茶壶水是否烧开。茶壶和面包机的生产商都很清楚：用户通常会采用异步方式来使用他们的产品，因此当这些机器完成任务时都会发出声音提示。但凡做事高效的人，总能在串行性与异步性之间找到合理的平衡，对于程序来说同样如此。

这些促使进程出现的因素（资源利用率、公平性以及便利性等）同样也促使着线程的出现。线程允许在同一个进程中同时存在多个程序控制流。线程会共享进程范围内的资源，例如内存句柄和文件句柄，但每个线程都有各自的程序计数器（Program Counter）、栈以及局部变量等。线程还提供了一种直观的分解模式来充分利用多处理器系统中的硬件并行性，而在同一个程序中的多个线程也可以被同时调度到多个 CPU 上运行。

线程也被称为轻量级进程。在大多数现代操作系统中，都是以线程为基本的调度单位，而不是进程。如果没有明确的协同机制，那么线程将彼此独立执行。由于同一个进程中的所有线程都将共享进程的内存地址空间，因此这些线程都能访问相同的变量并在同一个堆上分配对象，这就需要实现一种比在进程间共享数据粒度更细的数据共享机制。如果没有明确的同步机制来协同对共享数据的访问，那么当一个线程正在使用某个变量时，另一个线程可能同时访问这个变量，这将造成不可预测的结果。

1.2　线程的优势

如果使用得当，线程可以有效地降低程序的开发和维护等成本，同时提升复杂应用程序的性能。线程能够将大部分的异步工作流转换成串行工作流，因此能更好地模拟人类的工作方式和交互方式。此外，线程还可以降低代码的复杂度，使代码更容易编写、阅读和维护。

在 GUI（Graphic User Interface，图形用户界面）应用程序中，线程可以提高用户界面的响应灵敏度，而在服务器应用程序中，可以提升资源利用率以及系统吞吐率。线程还可以简化 JVM 的实现，垃圾收集器通常在一个或多个专门的线程中运行。在许多重要的 Java 应用程序中，都在一定程度上用到了线程。

1.2.1　发挥多处理器的强大能力

过去，多处理器系统是非常昂贵和稀少的，通常只有在大型数据中心和科学计算设备中才会使用多处理器系统。但现在，多处理器系统正日益普及，并且价格也在不断地降低，即使在低端服务器和中端桌面系统中，通常也会采用多个处理器。这种趋势还将进一步加快，因为通过提高时钟频率来提升性能已变得越来越困难，处理器生产厂商都开始转而在单个芯片上放置多个处理器核。所有的主流芯片制造商都开始了这种转变，而我们也已经看到了在一些机器上

出现了更多的处理器。

由于基本的调度单位是线程，因此如果在程序中只有一个线程，那么最多同时只能在一个处理器上运行。在双处理器系统上，单线程的程序只能使用一半的CPU资源，而在拥有100个处理器的系统上，将有99%的资源无法使用。另一方面，多线程程序可以同时在多个处理器上执行。如果设计正确，多线程程序可以通过提高处理器资源的利用率来提升系统吞吐率。

使用多个线程还有助于在单处理器系统上获得更高的吞吐率。如果程序是单线程的，那么当程序等待某个同步I/O操作完成时，处理器将处于空闲状态。而在多线程程序中，如果一个线程在等待I/O操作完成，另一个线程可以继续运行，使程序能够在I/O阻塞期间继续运行。（这就好比在等待水烧开的同时看报纸，而不是等到水烧开之后再开始看报纸）。

1.2.2 建模的简单性

通常，当只需要执行一种类型的任务（例如修改12个错误）时，在时间管理方面比执行多种类型的任务（例如，修复错误、面试系统管理员的接任者、完成团队的绩效考核，以及为下个星期的报告做幻灯片）要简单。当只有一种类型的任务需要完成时，只需埋头工作，直到完成所有的任务（或者你已经精疲力尽），你不需要花任何精力来琢磨下一步该做什么。而另一方面，如果需要完成多种类型的任务，那么需要管理不同任务之间的优先级和执行时间，并在任务之间进行切换，这将带来额外的开销。

对于软件来说同样如此：如果在程序中只包含一种类型的任务，那么比包含多种不同类型任务的程序要更易于编写，错误更少，也更容易测试。如果为模型中每种类型的任务都分配一个专门的线程，那么可以形成一种串行执行的假象，并将程序的执行逻辑与调度机制的细节，交替执行的操作，异步I/O以及资源等待等问题分离开来。通过使用线程，可以将复杂并且异步的工作流进一步分解为一组简单并且同步的工作流，每个工作流在一个单独的线程中运行，并在特定的同步位置进行交互。

我们可以通过一些现有的框架来实现上述目标，例如Servlet和RMI（Remote Method Invocation，远程方法调用）。框架负责解决一些细节问题，例如请求管理、线程创建、负载平衡，并在正确的时刻将请求分发给正确的应用程序组件。编写Servlet的开发人员不需要了解有多少请求在同一时刻要被处理，也不需要了解套接字的输入流或输出流是否被阻塞。当调用Servlet的service方法来响应Web请求时，可以以同步方式来处理这个请求，就好像它是一个单线程程序。这种方式可以简化组件的开发，并缩短掌握这种框架的学习时间。

1.2.3 异步事件的简化处理

服务器应用程序在接受来自多个远程客户端的套接字连接请求时，如果为每个连接都分配其各自的线程并且使用同步I/O，那么就会降低这类程序的开发难度。

如果某个应用程序对套接字执行读操作而此时还没有数据到来，那么这个读操作将一直阻塞，直到有数据到达。在单线程应用程序中，这不仅意味着在处理请求的过程中将停顿，而且还意味着在这个线程被阻塞期间，对所有请求的处理都将停顿。为了避免这个问题，单线程服

务器应用程序必须使用非阻塞 I/O，这种 I/O 的复杂性要远远高于同步 I/O，并且很容易出错。然而，如果每个请求都拥有自己的处理线程，那么在处理某个请求时发生的阻塞将不会影响其他请求的处理。

早期的操作系统通常会将进程中可创建的线程数量限制在一个较低的阈值内，大约在数百个（甚至更少）左右。因此，操作系统提供了一些高效的方法来实现多路 I/O，例如 Unix 的 select 和 poll 等系统调用，要调用这些方法，Java 类库需要获得一组实现非阻塞 I/O 的包（java.nio）。然而，在现代操作系统中，线程数量已得到极大的提升，这使得在某些平台上，即使有更多的客户端，为每个客户端分配一个线程也是可行的[⊖]。

1.2.4　响应更灵敏的用户界面

传统的 GUI 应用程序通常都是单线程的，从而在代码的各个位置都需要调用 poll 方法来获得输入事件（这种方式将给代码带来极大的混乱），或者通过一个“主事件循环（Main Event Loop）”来间接地执行应用程序的所有代码。如果在主事件循环中调用的代码需要很长时间才能执行完成，那么用户界面就会“冻结”，直到代码执行完成。这是因为只有当执行控制权返回到主事件循环后，才能处理后续的用户界面事件。

在现代的 GUI 框架中，例如 AWT 和 Swing 等工具，都采用一个事件分发线程（Event Dispatch Thread, EDT）来替代主事件循环。当某个用户界面事件发生时（例如按下一个按钮），在事件线程中将调用应用程序的事件处理器。由于大多数 GUI 框架都是单线程子系统，因此到目前为止仍然存在主事件循环，但它现在处于 GUI 工具的控制下并在其自己的线程中运行，而不是在应用程序的控制下。

如果在事件线程中执行的任务都是短暂的，那么界面的响应灵敏度就较高，因为事件线程能够很快地处理用户的动作。然而，如果事件线程中的任务需要很长的执行时间，例如对一个大型文档进行拼写检查，或者从网络上获取一个资源，那界面的响应灵敏度就会降低。如果用户在执行这类任务时触发了某个动作，那么必须等待很长时间才能获得响应，因为事件线程要先执行完该任务。更糟糕的是，不仅界面失去响应，而且即使在界面上包含了“取消”按钮，也无法取消这个长时间执行的任务，因为事件线程只有在执行完该任务后才能响应“取消”按钮的点击事件。然而，如果将这个长时间运行的任务放在一个单独的线程中运行，那么事件线程就能及时地处理界面事件，从而使用户界面具有更高的灵敏度。

1.3　线程带来的风险

Java 对线程的支持其实是一把双刃剑。虽然 Java 提供了相应的语言和库，以及一种明确的跨平台内存模型（该内存模型实现了在 Java 中开发“编写一次，随处运行”的并发应用程序），这些工具简化了并发应用程序的开发，但同时也提高了对开发人员的技术要求，因为在更多的

⊖ NPTL 线程软件包是专门设计用于支持数十万个线程的，在大多数 Linux 发布版本中都包含了这个软件包。非阻塞 I/O 有其自身的优势，但如果操作系统能更好地支持线程，那么需要使用非阻塞 I/O 的情况将变得更少。

程序中会使用线程。当线程还是一项鲜为人知的技术时，并发性是一个“高深的”主题，但现在，主流开发人员都必须了解线程方面的内容。

1.3.1 安全性问题

线程安全性可能是非常复杂的，在没有充足同步的情况下，多个线程中的操作执行顺序是不可预测的，甚至会产生奇怪的结果。在程序清单 1-1 的 UnsafeSequence 类中将产生一个整数值序列，该序列中的每个值都是唯一的。在这个类中简要地说明了多个线程之间的交替操作将如何导致不可预料的结果。在单线程环境中，这个类能正确地工作，但在多线程环境中则不能。

程序清单 1-1 非线程安全的数值序列生成器

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

    /** 返回一个唯一的数值。*/
    public int getNext() {
        return value++;
    }
}
```

UnsafeSequence 的问题在于，如果执行时机不对，那么两个线程在调用 getNext 时会得到相同的值。在图 1-1 中给出了这种错误情况。虽然递增运算 someVariable++ 看上去是单个操作，但事实上它包含三个独立的操作：读取 value，将 value 加 1，并将计算结果写入 value。由于运行时可能将多个线程之间的操作交替执行，因此这两个线程可能同时执行读操作，从而使它们得到相同的值，并都将这个值加 1。结果就是，在不同线程的调用中返回了相同的数值。

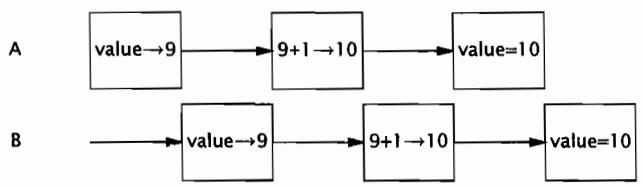


图 1-1 UnsafeSequence.getNext() 的错误执行情况

在图 1-1 中给出了不同线程之间的一种交替执行情况。在图中，执行时序按照从左到右的顺序递增，每行表示一个线程的动作。这些交替执行示意图给出的是最糟糕的执行情况[⊖]，目的是为了说明，如果错误地假设程序中的操作将按照某种特定顺序来执行，那么会存在各种可能的危险。

⊖ 事实上，在第 3 章中将看到，由于存在指令重排序的可能，因此实际情况可能会更糟糕。

在 UnsafeSequence 中使用了一个非标准的标注：@NotThreadSafe。这是在本书中使用的几个自定义标注之一，用于说明类和类成员的并发属性。（其他标注包括 @ThreadSafe 和 @Immutable，请参见附录 A 的详细信息）。线程安全性标注在许多方面都是有用的。如果用 @ThreadSafe 来标注某个类，那么开发人员可以放心地在多线程环境下使用这个类，维护人员也会发现它能保证线程安全性，而软件分析工具还可以识别出潜在的编码错误。

在 UnsafeSequence 类中说明的是一种常见的并发安全问题，称为竞态条件（Race Condition）。在多线程环境下，getValue 是否会返回唯一的值，要取决于运行时对线程中操作的交替执行方式，这并不是我们希望看到的情况。

由于多个线程要共享相同的内存地址空间，并且是并发运行，因此它们可能会访问或修改其他线程正在使用的变量。当然，这是一种极大的便利，因为这种方式比其他线程间通信机制更容易实现数据共享。但它同样也带来了巨大的风险：线程会由于无法预料的数据变化而发生错误。当多个线程同时访问和修改相同的变量时，将会在串行编程模型中引入非串行因素，而这种非串行性是很难分析的。要使多线程程序的行为可以预测，必须对共享变量的访问操作进行协同，这样才不会在线程之间发生彼此干扰。幸运的是，Java 提供了各种同步机制来协同这种访问。

通过将 getNext 修改为一个同步方法，可以修复 UnsafeSequence 中的错误，如程序清单 1-2 中的 Sequence[⊖]，这个类可以防止图 1-1 中错误的交替执行情况。（第 2 章和第 3 章将进一步分析这个类的工作原理。）

程序清单 1-2　线程安全的数值序列生成器

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

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

如果没有同步，那么无论是编译器、硬件还是运行时，都可以随意安排操作的执行时间和顺序，例如对寄存器或者处理器中的变量进行缓存，而这些被缓存的变量对于其他线程来说是暂时（甚至永久）不可见的。虽然这些技术有助于实现更优的性能，并且通常也是值得采用的方法，但它们也为开发人员带来了负担，因为开发人员必须找出这些数据在哪些位置被多个线程共享，只有这样才能使这些优化措施不破坏线程安全性。（第 16 章将详细介绍 JVM 实现了哪些顺序保证，以及同步将如何影响这些保证，但如果遵循第 2 章和第 3 章给出的指导原则，那么就可以绕开这些底层细节问题。）

⊖ 在 2.4 节中介绍了 @GuardedBy，这个标注说明了 Sequence 的同步策略。

1.3.2 活跃性问题

在开发并发代码时，一定要注意线程安全性是不可破坏的。安全性不仅对于多线程程序很重要，对于单线程程序同样重要。此外，线程还会导致一些在单线程程序中不会出现的问题，例如活跃性问题。

安全性的含义是“永远不发生糟糕的事情”，而活跃性则关注于另一个目标，即“某件正确的事情最终会发生”。当某个操作无法继续执行下去时，就会发生活跃性问题。在串行程序中，活跃性问题的形式之一就是无意中造成的无限循环，从而使循环之后的代码无法得到执行。线程将带来其他一些活跃性问题。例如，如果线程 A 在等待线程 B 释放其持有的资源，而线程 B 永远都不释放该资源，那么 A 就会永久地等待下去。第 10 章将介绍各种形式的活跃性问题，以及如何避免这些问题，包括死锁（10.1 节），饥饿（10.3.1 节），以及活锁（10.3.3 节）。与大多数并发性错误一样，导致活跃性问题的错误同样是难以分析的，因为它们依赖于不同线程的事件发生时序，因此在开发或者测试中并不总是能够重现。

1.3.3 性能问题

与活跃性问题密切相关的是性能问题。活跃性意味着某件正确的事情最终会发生，但却不够好，因为我们通常希望正确的事情尽快发生。性能问题包括多个方面，例如服务时间过长，响应不灵敏，吞吐率过低，资源消耗过高，或者可伸缩性较低等。与安全性和活跃性一样，在多线程程序中不仅存在与单线程程序相同的性能问题，而且还存在由于使用线程而引入的其他性能问题。

在设计良好的并发应用程序中，线程能提升程序的性能，但无论如何，线程总会带来某种程度的运行时开销。在多线程程序中，当线程调度器临时挂起活跃线程并转而运行另一个线程时，就会频繁地出现上下文切换操作（Context Switch），这种操作将带来极大的开销：保存和恢复执行上下文，丢失局部性，并且 CPU 时间将更多地花在线程调度而不是线程运行上。当线程共享数据时，必须使用同步机制，而这些机制往往会抑制某些编译器优化，使内存缓存区中的数据无效，以及增加共享内存总线的同步流量。所有这些因素都将带来额外的性能开销，第 11 章将详细介绍如何分析和减少这些开销。

1.4 线程无处不在

即使在程序中没有显式地创建线程，但在框架中仍可能会创建线程，因此在这些线程中调用的代码同样必须是线程安全的。这将给开发人员在设计和实现上带来沉重负担，因为开发线程安全的类比开发非线程安全的类要更加谨慎和细致。

每个 Java 应用程序都会使用线程。当 JVM 启动时，它将为 JVM 的内部任务（例如，垃圾收集、终结操作等）创建后台线程，并创建一个主线程来运行 main 方法。AWT（Abstract Window Toolkit，抽象窗口工具库）和 Swing 的用户界面框架将创建线程来管理用户界面事件。Timer 将创建线程来执行延迟任务。一些组件框架，例如 Servlet 和 RMI，都会创建线程池并调用这些线程中的方法。

如果要使用这些功能，那么就必须熟悉并发性和线程安全性，因为这些框架将创建线程并且在这些线程中调用程序中的代码。虽然将并发性认为是一种“可选的”或者“高级的”语言功能固然理想，但现实情况是，几乎所有的 Java 应用程序都是多线程的，因此在使用这些框架时仍然需要对应用程序状态的访问进行协同。

当某个框架在应用程序中引入并发性时，通常不可能将并发性仅局限于框架代码，因为框架本身会回调（Callback）应用程序的代码，而这些代码将访问应用程序的状态。同样，对线程安全性的需求也不能局限于被调用的代码，而是要延伸到需要访问这些代码所访问的程序状态的所有代码路径。因此，对线程安全性的需求将在程序中蔓延开来。

框架通过在框架线程中调用应用程序代码将并发性引入到程序中。在代码中将不可避免地访问应用程序状态，因此所有访问这些状态的代码路径都必须是线程安全的。

下面给出的模块都将在应用程序之外的线程中调用应用程序的代码。尽管线程安全性需求可能源自这些模块，但却不会止步于它们，而是会延伸到整个应用程序。

Timer。 Timer 类的作用是使任务在稍后的时刻运行，或者运行一次，或者周期性地运行。引入 Timer 可能会使串行程序变得复杂，因为 TimerTask 将在 Timer 管理的线程中执行，而不是由应用程序来管理。如果某个 TimerTask 访问了应用程序中其他线程访问的数据，那么不仅 TimerTask 需要以线程安全的方式来访问数据，其他类也必须采用线程安全的方式来访问该数据。通常，要实现这个目标，最简单的方式是确保 TimerTask 访问的对象本身是线程安全的，从而就能把线程安全性封装在共享对象内部。

Servlet 和 JavaServer Page（JSP）。Servlet 框架用于部署网页应用程序以及分发来自 HTTP 客户端的请求。到达服务器的请求可能会通过一个过滤器链被分发到正确的 Servlet 或 JSP。每个 Servlet 都表示一个程序逻辑组件，在高吞吐率的网站中，多个客户端可能同时请求同一个 Servlet 的服务。在 Servlet 规范中，Servlet 同样需要满足被多个线程同时调用，换句话说，Servlet 需要是线程安全的。

即使你可以确保每次只有一个线程调用某个 Servlet，但在构建网页应用程序时仍然必须注意线程安全性。Servlet 通常会访问与其他 Servlet 共享的信息，例如应用程序中的对象（这些对象保存在 ServletContext 中）或者会话中的对象（这些对象保存在每个客户端的 HttpSession 中）。当一个 Servlet 访问在多个 Servlet 或者请求中共享的对象时，必须正确地协同对这些对象的访问，因为多个请求可能在不同的线程中同时访问这些对象。Servlet 和 JSP，以及在 ServletContext 和 HttpSession 等容器中保存的 Servlet 过滤器和对象等，都必须是线程安全的。

远程方法调用（Remote Method Invocation，RMI）。RMI 使代码能够调用在其他 JVM 中运行的对象。当通过 RMI 调用某个远程方法时，传递给方法的参数必须被打包（也称为列集 [Marshaled]）到一个字节流中，通过网络传输给远程 JVM，然后由远程 JVM 拆包（或者称为散集 [Unmarshaled]）并传递给远程方法。

当 RMI 代码调用远程对象时，这个调用将在哪个线程中执行？你并不知道，但肯定不会在

你创建的线程中，而是将在一个由 RMI 管理的线程中调用对象。RMI 会创建多少个线程？同一个远程对象上的同一个远程方法会不会在多个 RMI 线程中被同时调用？⊖

远程对象必须注意两个线程安全性问题：正确地协同在多个对象中共享的状态，以及对远程对象本身状态的访问（由于同一个对象可能会在多个线程中被同时访问）。与 Servlet 相同，RMI 对象应该做好被多个线程同时调用的准备，并且必须确保它们自身的线程安全性。

Swing 和 AWT。GUI 应用程序的一个固有属性是异步性。用户可以在任意时刻选择一个菜单项或者按下一个按钮，应用程序就会及时响应，即使应用程序当时正在执行其他的任务。Swing 和 AWT 很好地解决了这个问题，它们创建了一个单独的线程来处理用户触发的事件，并对呈现给用户的图形界面进行更新。

Swing 的一些组件并不是线程安全的，例如 JTable。相反，Swing 程序通过将所有对 GUI 组件的访问局限在事件线程中以实现线程安全性。如果某个应用程序希望在事件线程之外控制 GUI，那么必须将控制 GUI 的代码放在事件线程中运行。

当用户触发某个 UI 动作时，在事件线程中就会有一个事件处理器被调用以执行用户请求的操作。如果事件处理器需要访问由其他线程同时访问的应用程序状态（例如编辑某个文档），那么这个事件处理器，以及访问这个状态的所有其他代码，都必须采用一种线程安全的方式来访问该状态。

⊖ 答案是：会的，但在 Javadoc 中并没有清楚地指出这一点，你需要阅读 RMI 规范。

第一部分
基础知识

第2章
线程安全性

你或许会感到奇怪，线程或者锁在并发编程中的作用，类似于铆钉和工字梁在土木工程中的作用。要建筑一座坚固的桥梁，必须正确地使用大量的铆钉和工字梁。同理，在构建稳健的并发程序时，必须正确地使用线程和锁。但这些终归只是一些机制。要编写线程安全的代码，其核心在于要对状态访问操作进行管理，特别是对共享的（Shared）和可变的（Mutable）状态的访问。

从非正式的意义上来说，对象的状态是指存储在状态变量（例如实例或静态域）中的数据。对象的状态可能包括其他依赖对象的域。例如，某个 HashMap 的状态不仅存储在 HashMap 对象本身，还存储在许多 Map.Entry 对象中。在对象的状态中包含了任何可能影响其外部可见行为的数据。

“共享”意味着变量可以由多个线程同时访问，而“可变”则意味着变量的值在其生命周期内可以发生变化。我们将像讨论代码那样来讨论线程安全性，但更侧重于如何防止在数据上发生不受控的并发访问。

一个对象是否需要是线程安全的，取决于它是否被多个线程访问。这指的是在程序中访问对象的方式，而不是对象要实现的功能。要使得对象是线程安全的，需要采用同步机制来协同对对象可变状态的访问。如果无法实现协同，那么可能会导致数据破坏以及其他不该出现的结果。

当多个线程访问某个状态变量并且其中有一个线程执行写入操作时，必须采用同步机制来协同这些线程对变量的访问。Java 中的主要同步机制是关键字 synchronized，它提供了一种独

占的加锁方式，但“同步”这个术语还包括 volatile 类型的变量，显式锁（Explicit Lock）以及原子变量。

在上述规则中并不存在一些想象中的“例外”情况。即使在某个程序中省略了必要同步机制并且看上去似乎能正确执行，而且通过了测试并在随后几年时间里都能正确地执行，但程序仍可能在某个时刻发生错误。

如果当多个线程访问同一个可变的状态变量时没有使用合适的同步，那么程序就会出现错误。有三种方式可以修复这个问题：

- 不在线程之间共享该状态变量。
- 将状态变量修改为不可变的变量。
- 在访问状态变量时使用同步。

如果在设计类的时候没有考虑并发访问的情况，那么在采用上述方法时可能需要对设计进行重大修改，因此要修复这个问题可谓是知易行难。如果从一开始就设计一个线程安全的类，那么比在以后再将这个类修改为线程安全的类要容易得多。

在一些大型程序中，要找出多个线程在哪些位置上将访问同一个变量是非常复杂的。幸运的是，面向对象这种技术不仅有助于编写出结构优雅、可维护性高的类，还有助于编写出线程安全的类。访问某个变量的代码越少，就越容易确保对变量的所有访问都实现正确同步，同时也更容易找出变量在哪些条件下被访问。Java 语言并没有强制要求将状态都封装在类中，开发人员完全可以将状态保存在某个公开的域（甚至公开的静态域）中，或者提供一个对内部对象的公开引用。然而，程序状态的封装性越好，就越容易实现程序的线程安全性，并且代码的维护人员也越容易保持这种方式。

当设计线程安全的类时，良好的面向对象技术、不可修改性，以及明晰的不变性规范都能起到一定的帮助作用。

在某些情况中，良好的面向对象设计技术与实际情况的需求并不一致。在这些情况中，可能需要牺牲一些良好的设计原则，以换取性能或者对遗留代码的向后兼容。有时候，面向对象中的抽象和封装会降低程序的性能（尽管很少有开发人员相信），但在编写并发应用程序时，一种正确的编程方法就是：首先使代码正确运行，然后再提高代码的速度。即便如此，最好也只是当性能测试结果和应用需求告诉你必须提高性能，以及测量结果表明这种优化在实际环境中确实能带来性能提升时，才进行优化。[⊖]

如果你必须打破封装，那么也并非不可以，你仍然可以实现程序的线程安全性，只是更困

⊖ 在编写并发代码时，应该始终遵循这个原则。由于并发错误是非常难以重现和调试的，因此如果只是在某段很少执行的代码路径上获得了性能提升，那么很可能被程序运行时存在的失败风险而抵消。

难，而且，程序的线程安全性将更加脆弱，不仅增加了开发的成本和风险，而且也增加了维护的成本和风险。第 4 章详细介绍了在哪些条件下可以安全地放宽状态变量的封装性。

到目前为止，我们使用了“线程安全类”和“线程安全程序”这两个术语，二者的含义基本相同。线程安全的程序是否完全由线程安全类构成？答案是否定的，完全由线程安全类构成的程序并不一定就是线程安全的，而在线程安全类中也可以包含非线程安全的类。第 4 章还将进一步介绍如何对线程安全类进行组合的相关问题。在任何情况中，只有当类中仅包含自己的状态时，线程安全类才是有意义的。线程安全性是一个在代码上使用的术语，但它只是与状态相关的，因此只能应用于封装其状态的整个代码，这可能是一个对象，也可能是整个程序。

2.1　什么是线程安全性

要对线程安全性给出一个确切的定义是非常复杂的。定义越正式，就越复杂，不仅很难提供有实际意义的指导建议，而且也很难从直观上去理解。因此，下面给出了一些非正式的描述，看上去令人困惑。在互联网上可以搜索到许多“定义”，例如：

…… 可以在多个线程中调用，并且在线程之间不会出现错误的交互。

……可以同时被多个线程调用，而调用者无须执行额外的动作。

看看这些定义，难怪我们会对线程安全性感到困惑。它们听起来非常像“如果某个类可以在多个线程中安全地使用，那么它就是一个线程安全的类”。对于这种说法，虽然没有太多的争议，但同样也不会带来太多的帮助。我们如何区分线程安全的类以及非线程安全的类？进一步说，“安全”的含义是什么？

在线程安全性的定义中，最核心的概念就是正确性。如果对线程安全性的定义是模糊的，那么就是因为缺乏对正确性的清晰定义。

正确性的含义是，某个类的行为与其规范完全一致。在良好的规范中通常会定义各种不变性条件（Invariant）来约束对象的状态，以及定义各种后验条件（Postcondition）来描述对象操作的结果。由于我们通常不会为类编写详细的规范，那么如何知道这些类是否正确呢？我们无法知道，但这并不妨碍我们在确信“类的代码能工作”后使用它们。这种“代码可信性”非常接近于我们对正确性的理解，因此我们可以将单线程的正确性近似定义为“所见即所知（we know it when we see it）”。在对“正确性”给出了一个较为清晰的定义后，就可以定义线程安全性：当多个线程访问某个类时，这个类始终都能表现出正确的行为，那么就称这个类是线程安全的。

> 当多个线程访问某个类时，不管运行时环境采用何种调度方式或者这些线程将如何交替执行，并且在主调代码中不需要任何额外的同步或协同，这个类都能表现出正确的行为，那么就称这个类是线程安全的。

由于单线程程序也可以看成是一个多线程程序，如果某个类在单线程环境[⊖]中都不是正

⊖ 如果你觉得这里对“正确性”的定义有些模糊，那么可以将线程安全类认为是一个在并发环境和单线程环境中都不会被破坏的类。

确的，那么它肯定不会是线程安全的。如果正确地实现了某个对象，那么在任何操作中（包括调用对象的公有方法或者对其公有域进行读 / 写操作）都不会违背不变性条件或后验条件。在线程安全类的对象实例上执行的任何串行或并行操作都不会使对象处于无效状态。

在线程安全类中封装了必要的同步机制，因此客户端无须进一步采取同步措施。

示例：一个无状态的 Servlet

我们在第 1 章列出了一组框架，其中每个框架都能创建多个线程并在这些线程中调用你编写的代码，因此你需要保证编写的代码是线程安全的。通常，线程安全性的需求并非来源于对线程的直接使用，而是使用像 Servlet 这样的框架。我们来看一个简单的示例——一个基于 Servlet 的因数分解服务，并逐渐扩展它的功能，同时确保它的线程安全性。

程序清单 2-1 给出了一个简单的因数分解 Servlet。这个 Servlet 从请求中提取出数值，执行因数分解，然后将结果封装到该 Servlet 的响应中。

程序清单 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);
    }
}
```

与大多数 Servlet 相同，StatelessFactorizer 是无状态的：它既不包含任何域，也不包含任何对其他类中域的引用。计算过程中的临时状态仅存在于线程栈上的局部变量中，并且只能由正在执行的线程访问。访问 StatelessFactorizer 的线程不会影响另一个访问同一个 StatelessFactorizer 的线程的计算结果，因为这两个线程并没有共享状态，就好像它们都在访问不同的实例。由于线程访问无状态对象的行为并不会影响其他线程中操作的正确性，因此无状态对象是线程安全的。

无状态对象一定是线程安全的。

大多数 Servlet 都是无状态的，从而极大地降低了在实现 Servlet 线程安全性时的复杂性。只有当 Servlet 在处理请求时需要保存一些信息，线程安全性才会成为一个问题。

2.2　原子性

当我们在无状态对象中增加一个状态时，会出现什么情况？假设我们希望增加一个“命中计数器”（Hit Counter）来统计所处理的请求数量。一种直观的方法是在 Servlet

中增加一个 long 类型的域，并且每处理一个请求就将这个值加 1，如程序清单 2-2 中的 UnsafeCountingFactorizer 所示。

程序清单 2-2　在没有同步的情况下统计已处理请求数量的 Servlet（不要这么做）

```
@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);
    }
}
```

不幸的是，UnsafeCountingFactorizer 并非线程安全的，尽管它在单线程环境中能正确运行。与前面的 UnsafeSequence 一样，这个类很可能会丢失一些更新操作。虽然递增操作 ++count 是一种紧凑的语法，使其看上去只是一个操作，但这个操作并非原子的，因而它并不会作为一个不可分割的操作来执行。实际上，它包含了三个独立的操作：读取 count 的值，将值加 1，然后将计算结果写入 count。这是一个“读取 - 修改 - 写入”的操作序列，并且其结果状态依赖于之前的状态。

图 1-1 给出了两个线程在没有同步的情况下同时对一个计数器执行递增操作时发生的情况。如果计数器的初始值为 9，那么在某些情况下，每个线程读到的值都为 9，接着执行递增操作，并且都将计数器的值设为 10。显然，这并不是我们希望看到的情况，如果有一次递增操作丢失了，命中计数器的值就将偏差 1。

你可能会认为，在基于 Web 的服务中，命中计数器值的少量偏差或许是可以接受的，在某些情况下也确实如此。但如果该计数器被用来生成数值序列或者唯一的对象标识符，那么在多次调用中返回相同的值将导致严重的数据完整性问题[⊖]。在并发编程中，这种由于不恰当的执行时序而出现不正确的结果是一种非常重要的情况，它有一个正式的名字：竞态条件（Race Condition）。

2.2.1　竞态条件

在 UnsafeCountingFactorizer 中存在多个竞态条件，从而使结果变得不可靠。当某个计算的正确性取决于多个线程的交替执行时序时，那么就会发生竞态条件。换句话说，就是正确的结

⊖ 在 UnsafeSequence 和 UnsafeCountingFactorizer 中还存在其他一些严重的问题，例如可能出现失效数据（Stale Data）问题（3.1.1 节）。

果要取决于运气[⊖]。最常见的竞态条件类型就是“先检查后执行（Check-Then-Act）”操作，即通过一个可能失效的观测结果来决定下一步的动作。

在实际情况中经常会遇到竞态条件。例如，假定你计划中午在 University Avenue 的星巴克与一位朋友会面。但当你到达那里时，发现在 University Avenue 上有两家星巴克，并且你不知道说好碰面的是哪一家。在 12:10 时，你没有在星巴克 A 看到朋友，那么就会去星巴克 B 看看他是否在那里，但他也不在那里。这有几种可能：你的朋友迟到了，还没到任何一家星巴克；你的朋友在你离开后到了星巴克 A；你的朋友在星巴克 B，但他去星巴克 A 找你，并且此时正在去星巴克 A 的途中。我们假设是最糟糕的情况，即最后一种可能。现在是 12:15，你们两个都去过了两家星巴克，并且都开始怀疑对方是否失约了。现在你会怎么做？回到另一家星巴克？来来回回要走多少次？除非你们之间约定了某种协议，否则你们整天都在 University Avenue 上走来走去，倍感沮丧。

在“我去看看他是否在另一家星巴克”这种方法中，问题在于：当你在街上走时，你的朋友可能已经离开了你要去的星巴克。你首先看了看星巴克 A，发现“他不在”，并且开始去找他。你可以在星巴克 B 中做同样的选择，但不是同时发生。两家星巴克之间有几分钟的路程，而就在这几分钟的时间里，系统的状态可能会发生变化。

在星巴克这个示例中说明了一种竞态条件，因为要获得正确的结果（与朋友会面），必须取决于事件的发生时序（当你们到达星巴克时，在离开并去另一家星巴克之前会等待多长时间……）。当你迈出前门时，你在星巴克 A 的观察结果将变得无效，你的朋友可能从后门进来了，而你却不知道。这种观察结果的失效就是大多数竞态条件的本质——基于一种可能失效的观察结果来做出判断或者执行某个计算。这种类型的竞态条件称为“先检查后执行”：首先观察到某个条件为真（例如文件 X 不存在），然后根据这个观察结果采用相应的动作（创建文件 X），但事实上，在你观察到这个结果以及开始创建文件之间，观察结果可能变得无效（另一个线程在这期间创建了文件 X），从而导致各种问题（未预期的异常、数据被覆盖、文件被破坏等）。

2.2.2　示例：延迟初始化中的竞态条件

使用“先检查后执行”的一种常见情况就是延迟初始化。延迟初始化的目的是将对象的初始化操作推迟到实际被使用时才进行，同时要确保只被初始化一次。在程序清单 2-3 中的 LazyInitRace 说明了这种延迟初始化情况。getInstance 方法首先判断 ExpensiveObject 是否已经被初始化，如果已经初始化则返回现有的实例，否则，它将创建一个新的实例，并返回一个引用，从而在后来的调用中就无须再执行这段高开销的代码路径。

⊖ 竞态条件这个术语很容易与另一个相关术语“数据竞争（Data Race）”相混淆。数据竞争是指，如果在访问共享的非 final 类型的域时没有采用同步来进行协同，那么就会出现数据竞争。当一个线程写入一个变量而另一个线程接下来读取这个变量，或者读取一个之前由另一个线程写入的变量时，并且在这两个线程之间没有使用同步，那么就可能出现数据竞争。在 Java 内存模型中，如果在代码中存在数据竞争，那么这段代码就没有确定的语义。并非所有的竞态条件都是数据竞争，同样并非所有的数据竞争都是竞态条件，但二者都可能使并发程序失败。在 UnsafeCountingFactorizer 中既存在竞态条件，又存在数据竞争。参见第 16 章了解数据竞争的更详细内容。

程序清单 2-3　延迟初始化中的竞态条件（不要这么做）

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

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

在 LazyInitRace 中包含了一个竞态条件，它可能会破坏这个类的正确性。假定线程 A 和线程 B 同时执行 getInstance。A 看到 instance 为空，因而创建一个新的 ExpensiveObject 实例。B 同样需要判断 instance 是否为空。此时的 instance 是否为空，要取决于不可预测的时序，包括线程的调度方式，以及 A 需要花多长时间来初始化 ExpensiveObject 并设置 instance。如果当 B 检查时，instance 为空，那么在两次调用 getInstance 时可能会得到不同的结果，即使 getInstance 通常被认为是返回相同的实例。

在 UnsafeCountingFactorizer 的统计命中计数操作中存在另一种竞态条件。在“读取 − 修改 − 写入”这种操作（例如递增一个计数器）中，基于对象之前的状态来定义对象状态的转换。要递增一个计数器，你必须知道它之前的值，并确保在执行更新的过程中没有其他线程会修改或使用这个值。

与大多数并发错误一样，竞态条件并不总是会产生错误，还需要某种不恰当的执行时序。然而，竞态条件也可能导致严重的问题。假定 LazyInitRace 被用于初始化应用程序范围内的注册表，如果在多次调用中返回不同的实例，那么要么会丢失部分注册信息，要么多个行为对同一组注册对象表现出不一致的视图。如果将 UnsafeSequence 用于在某个持久化框架中生成对象的标识，那么两个不同的对象最终将获得相同的标识，这就违反了标识的完整性约束条件。

2.2.3　复合操作

LazyInitRace 和 UnsafeCountingFactorizer 都包含一组需要以原子方式执行（或者说不可分割）的操作。要避免竞态条件问题，就必须在某个线程修改该变量时，通过某种方式防止其他线程使用这个变量，从而确保其他线程只能在修改操作完成之前或之后读取和修改状态，而不是在修改状态的过程中。

假定有两个操作 A 和 B，如果从执行 A 的线程来看，当另一个线程执行 B 时，要么将 B 全部执行完，要么完全不执行 B，那么 A 和 B 对彼此来说是原子的。原子操作是指，对于访问同一个状态的所有操作（包括该操作本身）来说，这个操作是一个以原子方式执行的操作。

如果 UnsafeSequence 中的递增操作是原子操作，那么图 1-1 中的竞态条件就不会发生，并

且递增操作在每次执行时都会把计数器增加 1。为了确保线程安全性，“先检查后执行”（例如延迟初始化）和“读取 - 修改 - 写入”（例如递增运算）等操作必须是原子的。我们将“先检查后执行”以及“读取 - 修改 - 写入”等操作统称为复合操作：包含了一组必须以原子方式执行的操作以确保线程安全性。在 2.3 节中，我们将介绍加锁机制，这是 Java 中用于确保原子性的内置机制。就目前而言，我们先采用另一种方式来修复这个问题，即使用一个现有的线程安全类，如程序清单 2-4 中的 CountingFactorizer 所示。

程序清单 2-4　使用 AtomicLong 类型的变量来统计已处理请求的数量

```
@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);
    }
}
```

在 java.util.concurrent.atomic 包中包含了一些原子变量类，用于实现在数值和对象引用上的原子状态转换。通过用 AtomicLong 来代替 long 类型的计数器，能够确保所有对计数器状态的访问操作都是原子的。[⊖]由于 Servlet 的状态就是计数器的状态，并且计数器是线程安全的，因此这里的 Servlet 也是线程安全的。

我们在因数分解的 Servlet 中增加了一个计数器，并通过使用线程安全类 AtomicLong 来管理计数器的状态，从而确保了代码的线程安全性。当在无状态的类中添加一个状态时，如果该状态完全由线程安全的对象来管理，那么这个类仍然是线程安全的。然而，在 2.3 节你将看到，当状态变量的数量由一个变为多个时，并不会像状态变量数量由零个变为一个那样简单。

在实际情况中，应尽可能地使用现有的线程安全对象（例如 AcomicLong）来管理类的状态。与非线程安全的对象相比，判断线程安全对象的可能状态及其状态转换情况要更为容易，从而也更容易维护和验证线程安全性。

2.3　加锁机制

当在 Servlet 中添加一个状态变量时，可以通过线程安全的对象来管理 Servlet 的状态以维护 Servlet 的线程安全性。但如果想在 Servlet 中添加更多的状态，那么是否只需添加更多的线程安全状态变量就足够了？

⊖ CountingFactorizer 调用 incrementAndGet 来递增计数器，同时会返回递增后的值。这里忽略了返回值。

假设我们希望提升 Servlet 的性能：将最近的计算结果缓存起来，当两个连续的请求对相同的数值进行因数分解时，可以直接使用上一次的计算结果，而无须重新计算。（这并非一种有效的缓存策略，5.6 节将给出一种更好的策略。）要实现该缓存策略，需要保存两个状态：最近执行因数分解的数值，以及分解结果。

我们曾通过 AtomicLong 以线程安全的方式来管理计数器的状态，那么，在这里是否可以使用类似的 AtomicReference[⊖]来管理最近执行因数分解的数值及其分解结果吗？在程序清单 2-5 中的 UnsafeCachingFactorizer 实现了这种思想。

程序清单 2-5 该 Servlet 在没有足够原子性保证的情况下对其最近计算结果进行缓存（不要这么做）

```
@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);
        }
    }
}
```

然而，这种方法并不正确。尽管这些原子引用本身都是线程安全的，但在 UnsafeCachingFactorizer 中存在着竞态条件，这可能产生错误的结果。

在线程安全性的定义中要求，多个线程之间的操作无论采用何种执行时序或交替方式，都要保证不变性条件不被破坏。UnsafeCachingFactorizer 的不变性条件之一是：在 lastFactors 中缓存的因数之积应该等于在 lastNumber 中缓存的数值。只有确保了这个不变性条件不被破坏，上面的 Servlet 才是正确的。当在不变性条件中涉及多个变量时，各个变量之间并不是彼此独立的，而是某个变量的值会对其他变量的值产生约束。因此，当更新某一个变量时，需要在同一个原子操作中对其他变量同时进行更新。

在某些执行时序中，UnsafeCachingFactorizer 可能会破坏这个不变性条件。在使用原子引用的情况下，尽管对 set 方法的每次调用都是原子的，但仍然无法同时更新 lastNumber 和 lastFactors。如果只修改了其中一个变量，那么在这两次修改操作之间，其他线程将发现不变性条件被破坏了。同样，我们也不能保证会同时获取两个值：在线程 A 获取这两个值的过程中，

⊖ AtomicLong 是一种替代 long 类型整数的线程安全类，类似地，AtomicReference 是一种替代对象引用的线程安全类。在第 15 章将介绍各种原子变量（Atomic Variable）及其优势。

线程 B 可能修改了它们，这样线程 A 也会发现不变性条件被破坏了。

要保持状态的一致性，就需要在单个原子操作中更新所有相关的状态变量。

2.3.1　内置锁

Java 提供了一种内置的锁机制来支持原子性：同步代码块（Synchronized Block）。（第 3 章将介绍加锁机制以及其他同步机制的另一个重要方面：可见性）同步代码块包括两部分：一个作为锁的对象引用，一个作为由这个锁保护的代码块。以关键字 synchronized 来修饰的方法就是一种横跨整个方法体的同步代码块，其中该同步代码块的锁就是方法调用所在的对象。静态的 synchronized 方法以 Class 对象作为锁。

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

每个 Java 对象都可以用做一个实现同步的锁，这些锁被称为内置锁 (Intrinsic Lock) 或监视器锁 (Monitor Lock)。线程在进入同步代码块之前会自动获得锁，并且在退出同步代码块时自动释放锁，而无论是通过正常的控制路径退出，还是通过从代码块中抛出异常退出。获得内置锁的唯一途径就是进入由这个锁保护的同步代码块或方法。

Java 的内置锁相当于一种互斥体（或互斥锁），这意味着最多只有一个线程能持有这种锁。当线程 A 尝试获取一个由线程 B 持有的锁时，线程 A 必须等待或者阻塞，直到线程 B 释放这个锁。如果 B 永远不释放锁，那么 A 也将永远地等下去。

由于每次只能有一个线程执行内置锁保护的代码块，因此，由这个锁保护的同步代码块会以原子方式执行，多个线程在执行该代码块时也不会相互干扰。并发环境中的原子性与事务应用程序中的原子性有着相同的含义——一组语句作为一个不可分割的单元被执行。任何一个执行同步代码块的线程，都不可能看到有其他线程正在执行由同一个锁保护的同步代码块。

这种同步机制使得要确保因数分解 Servlet 的线程安全性变得更简单。在程序清单 2-6 中使用了关键字 synchronized 来修饰 service 方法，因此在同一时刻只有一个线程可以执行 service 方法。现在的 SynchronizedFactorizer 是线程安全的。然而，这种方法却过于极端，因为多个客户端无法同时使用因数分解 Servlet，服务的响应性非常低，无法令人接受。这是一个性能问题，而不是线程安全问题，我们将在 2.5 节解决这个问题。

程序清单 2-6　这个 Servlet 能正确地缓存最新的计算结果，但并发性却非常糟糕（不要这么做）

```
@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　重入

当某个线程请求一个由其他线程持有的锁时，发出请求的线程就会阻塞。然而，由于内置锁是可重入的，因此如果某个线程试图获得一个已经由它自己持有的锁，那么这个请求就会成功。“重入”意味着获取锁的操作的粒度是“线程”，而不是“调用”[⊖]。重入的一种实现方法是，为每个锁关联一个获取计数值和一个所有者线程。当计数值为 0 时，这个锁就被认为是没有被任何线程持有。当线程请求一个未被持有的锁时，JVM 将记下锁的持有者，并且将获取计数值置为 1。如果同一个线程再次获取这个锁，计数值将递增，而当线程退出同步代码块时，计数器会相应地递减。当计数值为 0 时，这个锁将被释放。

重入进一步提升了加锁行为的封装性，因此简化了面向对象并发代码的开发。在程序清单 2-7 的代码中，子类改写了父类的 synchronized 方法，然后调用父类中的方法，此时如果没有可重入的锁，那么这段代码将产生死锁。由于 Widget 和 LoggingWidget 中 doSomething 方法都是 synchronized 方法，因此每个 doSomething 方法在执行前都会获取 Widget 上的锁。然而，如果内置锁不是可重入的，那么在调用 super.doSomething 时将无法获得 Widget 上的锁，因为这个锁已经被持有，从而线程将永远停顿下去，等待一个永远也无法获得的锁。重入则避免了这种死锁情况的发生。

程序清单 2-7　如果内置锁不是可重入的，那么这段代码将发生死锁

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

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

⊖ 这与 pthread(POSIX 线程) 互斥体的默认加锁行为不同，pthread 互斥体的获取操作是以“调用”为粒度的。

2.4　用锁来保护状态

由于锁能使其保护的代码路径以串行形式[⊖]来访问，因此可以通过锁来构造一些协议以实现对共享状态的独占访问。只要始终遵循这些协议，就能确保状态的一致性。

访问共享状态的复合操作，例如命中计数器的递增操作（读取－修改－写入）或者延迟初始化（先检查后执行），都必须是原子操作以避免产生竞态条件。如果在复合操作的执行过程中持有一个锁，那么会使复合操作成为原子操作。然而，仅仅将复合操作封装到一个同步代码块中是不够的。如果用同步来协调对某个变量的访问，那么在访问这个变量的所有位置上都需要使用同步。而且，当使用锁来协调对某个变量的访问时，在访问变量的所有位置上都要使用同一个锁。

一种常见的错误是认为，只有在写入共享变量时才需要使用同步，然而事实并非如此（3.1节将进一步解释其中的原因）。

> 对于可能被多个线程同时访问的可变状态变量，在访问它时都需要持有同一个锁，在这种情况下，我们称状态变量是由这个锁保护的。

在程序清单2-6的SynchronizedFactorizer中，lastNumber和lastFactors这两个变量都是由Servlet对象的内置锁来保护的，在标注@GuardedBy中也已经说明了这一点。

对象的内置锁与其状态之间没有内在的关联。虽然大多数类都将内置锁用做一种有效的加锁机制，但对象的域并不一定要通过内置锁来保护。当获取与对象关联的锁时，并不能阻止其他线程访问该对象，某个线程在获得对象的锁之后，只能阻止其他线程获得同一个锁。之所以每个对象都有一个内置锁，只是为了免去显式地创建锁对象。[⊜]你需要自行构造加锁协议或者同步策略来实现对共享状态的安全访问，并且在程序中自始至终地使用它们。

> 每个共享的和可变的变量都应该只由一个锁来保护，从而使维护人员知道是哪一个锁。

一种常见的加锁约定是，将所有的可变状态都封装在对象内部，并通过对象的内置锁对所有访问可变状态的代码路径进行同步，使得在该对象上不会发生并发访问。在许多线程安全类中都使用了这种模式，例如Vector和其他的同步集合类。在这种情况下，对象状态中的所有变量都由对象的内置锁保护起来。然而，这种模式并没有任何特殊之处，编译器或运行时都不会强制实施这种（或者其他的）模式[⊝]。如果在添加新的方法或代码路径时忘记了使用同步，那

⊖ 对象的串行访问（Serializing Access）与对象的序列化（Serialization，即将对象转化为字节流）操作毫不相干。串行访问意味着多个线程依次以独占的方式访问对象，而不是并发地访问。

⊜ 回想起来，这种设计决策或许比较糟糕：不仅会引起混乱，而且还迫使JVM需要在对象大小与加锁性能之间进行权衡。

⊝ 如果某个变量在多个位置上的访问操作中都持有一个锁，但并非在所有位置上的访问操作都如此时，那么通过一些代码核查工具，例如FindBugs，就可以发现这种情况，并报告可能出现了一个错误。

么这种加锁协议会很容易被破坏。

并非所有数据都需要锁的保护，只有被多个线程同时访问的可变数据才需要通过锁来保护。第 1 章曾介绍，当添加一个简单的异步事件时，例如 TimerTask，整个程序都需要满足线程安全性要求，尤其是当程序状态的封装性比较糟糕时。考虑一个处理大规模数据的单线程程序，由于任何数据都不会在多个线程之间共享，因此在单线程程序中不需要同步。现在，假设希望添加一个新功能，即定期地对数据处理进度生成快照，这样当程序崩溃或者必须停止时无须再次从头开始。你可能会选择使用 TimerTask，每十分钟触发一次，并将程序状态保存到一个文件中。

由于 TimerTask 在另一个（由 Timer 管理的）线程中调用，因此现在就有两个线程同时访问快照中的数据：程序的主线程与 Timer 线程。这意味着，当访问程序的状态时，不仅 TimerTask 代码必须使用同步，而且程序中所有访问相同数据的代码路径也必须使用同步。原本在程序中不需要使用同步，现在变成了在程序的各个位置都需要使用同步。

当某个变量由锁来保护时，意味着在每次访问这个变量时都需要首先获得锁，这样就确保在同一时刻只有一个线程可以访问这个变量。当类的不变性条件涉及多个状态变量时，那么还有另外一个需求：在不变性条件中的每个变量都必须由同一个锁来保护。因此可以在单个原子操作中访问或更新这些变量，从而确保不变性条件不被破坏。在 SynchronizedFactorizer 类中说明了这条规则：缓存的数值和因数分解结果都由 Servlet 对象的内置锁来保护。

> 对于每个包含多个变量的不变性条件，其中涉及的所有变量都需要由同一个锁来保护。

如果同步可以避免竞态条件问题，那么为什么不在每个方法声明时都使用关键字 synchronized？事实上，如果不加区别地滥用 synchronized，可能导致程序中出现过多的同步。此外，如果只是将每个方法都作为同步方法，例如 Vector，那么并不足以确保 Vector 上复合操作都是原子的：

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

虽然 contains 和 add 等方法都是原子方法，但在上面这个“如果不存在则添加 (put-if-absent)”的操作中仍然存在竞态条件。虽然 synchronized 方法可以确保单个操作的原子性，但如果要把多个操作合并为一个复合操作，还是需要额外的加锁机制（请参见 4.4 节了解如何在线程安全对象中添加原子操作的方法）。此外，将每个方法都作为同步方法还可能导致活跃性问题 (Liveness) 或性能问题（Performance），我们在 SynchronizedFactorizer 中已经看到了这些问题。

2.5　活跃性与性能

在 UnsafeCachingFactorizer 中，我们通过在因数分解 Servlet 中引入了缓存机制来提升性能。在缓存中需要使用共享状态，因此需要通过同步来维护状态的完整性。然而，如果使用 SynchronizedFactorizer 中的同步方式，那么代码的执行性能将非常糟糕。SynchronizedFactorizer

中采用的同步策略是，通过 Servlet 对象的内置锁来保护每一个状态变量，该策略的实现方式也就是对整个 service 方法进行同步。虽然这种简单且粗粒度的方法能确保线程安全性，但付出的代价却很高。

由于 service 是一个 synchronized 方法，因此每次只有一个线程可以执行。这就背离了 Serlvet 框架的初衷，即 Serlvet 需要能同时处理多个请求，这在负载过高的情况下将给用户带来糟糕的体验。如果 Servlet 在对某个大数值进行因数分解时需要很长的执行时间，那么其他的客户端必须一直等待，直到 Servlet 处理完当前的请求，才能开始另一个新的因数分解运算。如果在系统中有多个 CPU 系统，那么当负载很高时，仍然会有处理器处于空闲状态。即使一些执行时间很短的请求，比如访问缓存的值，仍然需要很长时间，因为这些请求都必须等待前一个请求执行完成。

图 2-1 给出了当多个请求同时到达因数分解 Servlet 时发生的情况：这些请求将排队等待处理。我们将这种 Web 应用程序称之为不良并发 (Poor Concurrency) 应用程序：可同时调用的数量，不仅受到可用处理资源的限制，还受到应用程序本身结构的限制。幸运的是，通过缩小同步代码块的作用范围，我们很容易做到既确保 Servlet 的并发性，同时又维护线程安全性。要确保同步代码块不要过小，并且不要将本应是原子的操作拆分到多个同步代码块中。应该尽量将不影响共享状态且执行时间较长的操作从同步代码块中分离出去，从而在这些操作的执行过程中，其他线程可以访问共享状态。

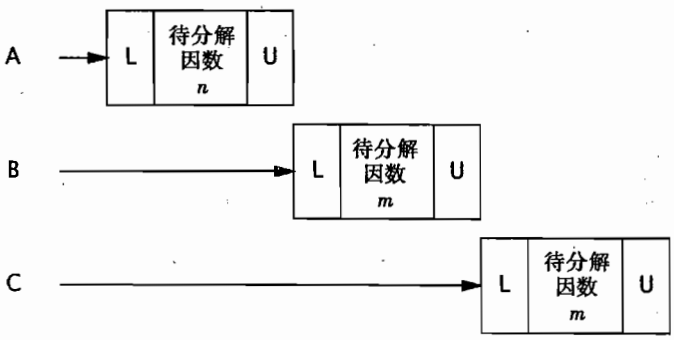


图 2-1　SynchronizedFactorizer 中的不良并发

程序清单 2-8 中的 CachedFactorizer 将 Servlet 的代码修改为使用两个独立的同步代码块，每个同步代码块都只包含一小段代码。其中一个同步代码块负责保护判断是否只需返回缓存结果的“先检查后执行”操作序列，另一个同步代码块则负责确保对缓存的数值和因数分解结果进行同步更新。此外，我们还重新引入了“命中计数器”，添加了一个“缓存命中”计数器，并在第一个同步代码块中更新这两个变量。由于这两个计数器也是共享可变状态的一部分，因此必须在所有访问它们的位置上都使用同步。位于同步代码块之外的代码将以独占方式来访问局部（位于栈上的）变量，这些变量不会在多个线程间共享，因此不需要同步。

程序清单 2-8　缓存最近执行因数分解的数值及其计算结果的 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);
    }
}
```

在 CachedFactorizer 中不再使用 AtomicLong 类型的命中计数器，而是使用了一个 long 类型的变量。当然也可以使用 AtomicLong 类型，但使用 CountingFactorizer 带来的好处更多。对在单个变量上实现原子操作来说，原子变量是很有用的，但由于我们已经使用了同步代码块来构造原子操作，而使用两种不同的同步机制不仅会带来混乱，也不会在性能或安全性上带来任何好处，因此在这里不使用原子变量。

重新构造后的 CachedFactorizer 实现了在简单性（对整个方法进行同步）与并发性（对尽可能短的代码路径进行同步）之间的平衡。在获取与释放锁等操作上都需要一定的开销，因此如果将同步代码块分解得过细（例如将 ++hits 分解到它自己的同步代码块中），那么通常并不好，尽管这样做不会破坏原子性。当访问状态变量或者在复合操作的执行期间，CachedFactorizer 需要持有锁，但在执行时间较长的因数分解运算之前要释放锁。这样既确保了线程安全性，也不会过多地影响并发性，而且在每个同步代码块中的代码路径都“足够短”。

要判断同步代码块的合理大小，需要在各种设计需求之间进行权衡，包括安全性（这个需求必须得到满足）、简单性和性能。有时候，在简单性与性能之间会发生冲突，但在 CachedFactorizer 中已经说明了，在二者之间通常能找到某种合理的平衡。

通常，在简单性与性能之间存在着相互制约因素。当实现某个同步策略时，一定不要盲目地为了性能而牺牲简单性（这可能会破坏安全性）。

当使用锁时，你应该清楚代码块中实现的功能，以及在执行该代码块时是否需要很长的时间。无论是执行计算密集的操作，还是在执行某个可能阻塞的操作，如果持有锁的时间过长，那么都会带来活跃性或性能问题。

当执行时间较长的计算或者可能无法快速完成的操作时（例如，网络I/O或控制台I/O），一定不要持有锁。

第3章

对象的共享

第2章的开头曾指出，要编写正确的并发程序，关键问题在于：在访问共享的可变状态时需要进行正确的管理。第2章介绍了如何通过同步来避免多个线程在同一时刻访问相同的数据，而本章将介绍如何共享和发布对象，从而使它们能够安全地由多个线程同时访问。这两章合在一起，就形成了构建线程安全类以及通过 java.util.concurrent 类库来构建并发应用程序的重要基础。

我们已经知道了同步代码块和同步方法可以确保以原子的方式执行操作，但一种常见的误解是，认为关键字 synchronized 只能用于实现原子性或者确定“临界区（Critical Section）”。同步还有另一个重要的方面：内存可见性（Memory Visibility）。我们不仅希望防止某个线程正在使用对象状态而另一个线程在同时修改该状态，而且希望确保当一个线程修改了对象状态后，其他线程能够看到发生的状态变化。如果没有同步，那么这种情况就无法实现。你可以通过显式的同步或者类库中内置的同步来保证对象被安全地发布。

3.1 可见性

可见性是一种复杂的属性，因为可见性中的错误总是会违背我们的直觉。在单线程环境中，如果向某个变量先写入值，然后在没有其他写入操作的情况下读取这个变量，那么总能得到相同的值。这看起来很自然。然而，当读操作和写操作在不同的线程中执行时，情况却并非如此，这听起来或许有些难以接受。通常，我们无法确保执行读操作的线程能适时地看到其他线程写入的值，有时甚至是根本不可能的事情。为了确保多个线程之间对内存写入操作的可见性，必须使用同步机制。

在程序清单 3-1 中的 NoVisibility 说明了当多个线程在没有同步的情况下共享数据时出现的错误。在代码中，主线程和读线程都将访问共享变量 ready 和 number。主线程启动读线程，然后将 number 设为 42，并将 ready 设为 true。读线程一直循环直到发现 ready 的值变为 true，然后输出 number 的值。虽然 NoVisibility 看起来会输出 42，但事实上很可能输出 0，或者根本无法终止。这是因为在代码中没有使用足够的同步机制，因此无法保证主线程写入的 ready 值和 number 值对于读线程来说是可见的。

程序清单 3-1 在没有同步的情况下共享变量（不要这么做）

```
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 可能会持续循环下去，因为读线程可能永远都看不到 ready 的值。一种更奇怪的现象是，NoVisibility 可能会输出 0，因为读线程可能看到了写入 ready 的值，但却没有看到之后写入 number 的值，这种现象被称为“重排序（Reordering）”。只要在某个线程中无法检测到重排序情况（即使在其他线程中可以很明显地看到该线程中的重排序），那么就无法确保线程中的操作将按照程序中指定的顺序来执行。[⊖]当主线程首先写入 number，然后在没有同步的情况下写入 ready，那么读线程看到的顺序可能与写入的顺序完全相反。

在没有同步的情况下，编译器、处理器以及运行时等都可能对操作的执行顺序进行一些意想不到的调整。在缺乏足够同步的多线程程序中，要想对内存操作的执行顺序进行判断，几乎无法得出正确的结论。

NoVisibility 是一个简单的并发程序，只包含两个线程和两个共享变量，但即便如此，在判断程序的执行结果以及是否会结束时仍然很容易得出错误结论。要对那些缺乏足够同步的并发程序的执行情况进行推断是十分困难的。

这听起来有点恐怖，但实际情况也确实如此。幸运的是，有一种简单的方法能避免这些复杂的问题：只要有数据在多个线程之间共享，就使用正确的同步。

3.1.1　失效数据

NoVisibility 展示了在缺乏同步的程序中可能产生错误结果的一种情况：失效数据。当读线程查看 ready 变量时，可能会得到一个已经失效的值。除非在每次访问变量时都使用同步，否则很可能获得该变量的一个失效值。更糟糕的是，失效值可能不会同时出现：一个线程可能获得某个变量的最新值，而获得另一个变量的失效值。

通常，当食物过期（即失效）时，还是可以食用的，只不过味道差了一些。但失效的

⊖ 这看上去似乎是一种失败的设计，但却能使 JVM 充分地利用现代多核处理器的强大性能。例如，在缺少同步的情况下，Java 内存模型允许编译器对操作顺序进行重排序，并将数值缓存在寄存器中。此外，它还允许 CPU 对操作顺序进行重排序，并将数值缓存在处理器特定的缓存中。更多细节请参阅第 16 章。

数据可能导致更危险的情况。虽然在 Web 应用程序中失效的命中计数器可能不会导致太糟糕的情况[⊖]，但在其他情况中，失效值可能会导致一些严重的安全问题或者活跃性问题。在 NoVisibility 中，失效数据可能导致输出错误的值，或者使程序无法结束。如果对象的引用（例如链表中的指针）失效，那么情况会更复杂。失效数据还可能导致一些令人困惑的故障，例如意料之外的异常、被破坏的数据结构、不精确的计算以及无限循环等。

程序清单 3-2 中的 MutableInteger 不是线程安全的，因为 get 和 set 都是在没有同步的情况下访问 value 的。与其他问题相比，失效值问题更容易出现：如果某个线程调用了 set，那么另一个正在调用 get 的线程可能会看到更新后的 value 值，也可能看不到。

程序清单 3-2 非线程安全的可变整数类

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

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

在程序清单 3-3 的 SynchronizedInteger 中，通过对 get 和 set 等方法进行同步，可以使 MutableInteger 成为一个线程安全的类。仅对 set 方法进行同步是不够的，调用 get 的线程仍然会看见失效值。

程序清单 3-3 线程安全的可变整数类

```
@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

当线程在没有同步的情况下读取变量时，可能会得到一个失效值，但至少这个值是由之前某个线程设置的值，而不是一个随机值。这种安全性保证也被称为最低安全性（out-of-thin-air safety）。

最低安全性适用于绝大多数变量，但是存在一个例外：非 volatile 类型的 64 位数值变量（double 和 long，请参见 3.1.4 节）。Java 内存模型要求，变量的读取操作和写入操作都必须是原子操作，但对于非 volatile 类型的 long 和 double 变量，JVM 允许将 64 位的读操作或写操作

⊖ 在没有同步的情况下读取数据，类似于在数据库中使用 READ_UNCOMMITTED 隔离级别，在这种级别上将牺牲准确性以获取性能的提升。然而，在非同步的读取操作中则牺牲了更多的准确度，因为线程看到的共享变量值很容易失效。

分解为两个 32 位的操作。当读取一个非 volatile 类型的 long 变量时，如果对该变量的读操作和写操作在不同的线程中执行，那么很可能会读取到某个值的高 32 位和另一个值的低 32 位[⊖]。因此，即使不考虑失效数据问题，在多线程程序中使用共享且可变的 long 和 double 等类型的变量也是不安全的，除非用关键字 volatile 来声明它们，或者用锁保护起来。

3.1.3　加锁与可见性

内置锁可以用于确保某个线程以一种可预测的方式来查看另一个线程的执行结果，如图 3-1 所示。当线程 A 执行某个同步代码块时，线程 B 随后进入由同一个锁保护的同步代码块，在这种情况下可以保证，在锁被释放之前，A 看到的变量值在 B 获得锁后同样可以由 B 看到。换句话说，当线程 B 执行由锁保护的同步代码块时，可以看到线程 A 之前在同一个同步代码块中的所有操作结果。如果没有同步，那么就无法实现上述保证。

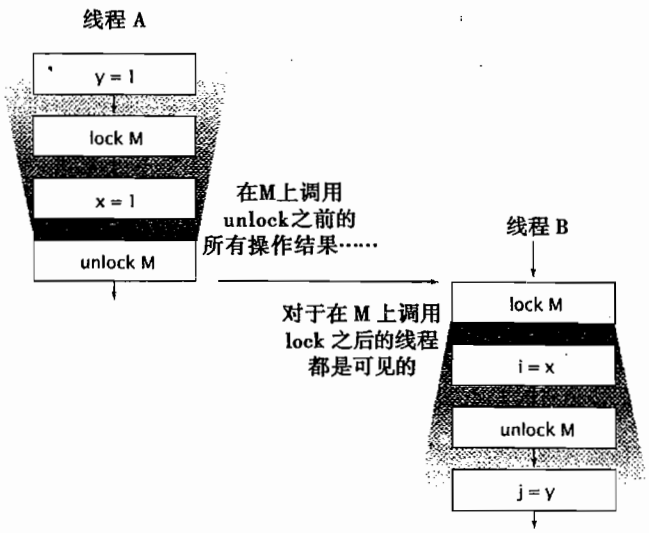


图 3-1　同步的可见性保证

现在，我们可以进一步理解为什么在访问某个共享且可变的变量时要求所有线程在同一个锁上同步，就是为了确保某个线程写入该变量的值对于其他线程来说都是可见的。否则，如果一个线程在未持有正确锁的情况下读取某个变量，那么读到的可能是一个失效值。

加锁的含义不仅仅局限于互斥行为，还包括内存可见性。为了确保所有线程都能看到共享变量的最新值，所有执行读操作或者写操作的线程都必须在同一个锁上同步。

3.1.4　Volatile 变量

Java 语言提供了一种稍弱的同步机制，即 volatile 变量，用来确保将变量的更新操作通知

⊖ 在编写 Java 虚拟机规范时，许多主流处理器架构还不能有效地提供 64 位数值的原子操作。

到其他线程。当把变量声明为 volatile 类型后，编译器与运行时都会注意到这个变量是共享的，因此不会将该变量上的操作与其他内存操作一起重排序。volatile 变量不会被缓存在寄存器或者对其他处理器不可见的地方，因此在读取 volatile 类型的变量时总会返回最新写入的值。

理解 volatile 变量的一种有效方法是，将它们的行为想象成程序清单 3-3 中 SynchronizedInteger 的类似行为，并将 volaLile 变量的读操作和写操作分别替换为 get 方法和 set 方法[⊖]。然而，在访问 volatile 变量时不会执行加锁操作，因此也就不会使执行线程阻塞，因此 volatile 变量是一种比 sychronized 关键字更轻量级的同步机制。[⊜]

volatile 变量对可见性的影响比 volatile 变量本身更为重要。当线程 A 首先写入一个 volatile 变量并且线程 B 随后读取该变量时，在写入 volatile 变量之前对 A 可见的所有变量的值，在 B 读取了 volatile 变量后，对 B 也是可见的。因此，从内存可见性的角度来看，写入 volatile 变量相当于退出同步代码块，而读取 volatile 变量就相当于进入同步代码块。然而，我们并不建议过度依赖 volatile 变量提供的可见性。如果在代码中依赖 volatile 变量来控制状态的可见性，通常比使用锁的代码更脆弱，也更难以理解。

> 仅当 volatile 变量能简化代码的实现以及对同步策略的验证时，才应该使用它们。如果在验证正确性时需要对可见性进行复杂的判断，那么就不要使用 volatile 变量。volatile 变量的正确使用方式包括：确保它们自身状态的可见性，确保它们所引用对象的状态的可见性，以及标识一些重要的程序生命周期事件的发生（例如，初始化或关闭）。

程序清单 3-4 给出了 volatile 变量的一种典型用法：检查某个状态标记以判断是否退出循环。在这个示例中，线程试图通过类似于数绵羊的传统方法进入休眠状态。为了使这个示例能正确执行，asleep 必须为 volatile 变量。否则，当 asleep 被另一个线程修改时，执行判断的线程却发现不了[⊝]。我们也可以用锁来确保 asleep 更新操作的可见性，但这将使代码变得更加复杂。

程序清单 3-4 数绵羊

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

⊖ 这种类比并不准确，SynchronizedInteger 在内存可见性上的作用比 volatile 变量更强。请参见第 16 章。

⊜ 在当前大多数处理器架构上，读取 volatile 变量的开销只比读取非 volatile 变量的开销略高一些。

⊝ 调试小提示：对于服务器应用程序，无论在开发阶段还是在测试阶段，当启动 JVM 时一定都要指定 -server 命令行选项。server 模式的 JVM 将比 client 模式的 JVM 进行更多的优化，例如将循环中未被修改的变量提升到循环外部，因此在开发环境（client 模式的 JVM）中能正确运行的代码，可能会在部署环境（server 模式的 JVM）中运行失败。例如，如果在程序清单 3-4 中“忘记”把 asleep 变量声明为 volatile 类型，那么 server 模式的 JVM 会将 asleep 的判断条件提升到循环体外部（这将导致一个无限循环），但 client 模式的 JVM 不会这么做。在解决开发环境中出现无限循环问题时，解决这个问题的开销远小于解决在应用环境出现无限循环的开销。

虽然 volatile 变量很方便，但也存在一些局限性。volatile 变量通常用做某个操作完成、发生中断或者状态的标志，例如程序清单 3-4 中的 asleep 标志。尽管 volatile 变量也可以用于表示其他的状态信息，但在使用时要非常小心。例如，volatile 的语义不足以确保递增操作(count++) 的原子性，除非你能确保只有一个线程对变量执行写操作。(原子变量提供了“读－改－写”的原子操作，并且常常用做一种“更好的 volatile 变量”。请参见第 15 章)。

加锁机制既可以确保可见性又可以确保原子性，而 volatile 变量只能确保可见性。

当且仅当满足以下所有条件时，才应该使用 volatile 变量：

- 对变量的写入操作不依赖变量的当前值，或者你能确保只有单个线程更新变量的值。
- 该变量不会与其他状态变量一起纳入不变性条件中。
- 在访问变量时不需要加锁。

3.2　发布与逸出

“发布 (Publish)”一个对象的意思是指，使对象能够在当前作用域之外的代码中使用。例如，将一个指向该对象的引用保存到其他代码可以访问的地方，或者在某一个非私有的方法中返回该引用，或者将引用传递到其他类的方法中。在许多情况中，我们要确保对象及其内部状态不被发布。而在某些情况下，我们又需要发布某个对象，但如果在发布时要确保线程安全性，则可能需要同步。发布内部状态可能会破坏封装性，并使得程序难以维持不变性条件。例如，如果在对象构造完成之前就发布该对象，就会破坏线程安全性。当某个不应该发布的对象被发布时，这种情况就被称为逸出 (Escape)。3.5 节介绍了如何安全发布对象的一些方法。现在，我们首先来看看一个对象是如何逸出的。

发布对象的最简单方法是将对象的引用保存到一个公有的静态变量中，以便任何类和线程都能看见该对象，如程序清单 3-5 所示。在 initialize 方法中实例化一个新的 HashSet 对象，并将对象的引用保存到 knownSecrets 中以发布该对象。

程序清单 3-5　发布一个对象

```
public static Set<Secret> knownSecrets;

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

当发布某个对象时，可能会间接地发布其他对象。如果将一个 Secret 对象添加到集合 knownSecrets 中，那么同样会发布这个对象，因为任何代码都可以遍历这个集合，并获得对这个新 Secret 对象的引用。同样，如果从非私有方法中返回一个引用，那么同样会发布返回的对象。程序清单 3-6 中的 UnsafeStates 发布了本应为私有的状态数组。

程序清单 3-6 使内部的可变状态逸出（不要这么做）

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

如果按照上述方式来发布 states，就会出现问题，因为任何调用者都能修改这个数组的内容。在这个示例中，数组 states 已经逸出了它所在的作用域，因为这个本应是私有的变量已经被发布了。

当发布一个对象时，在该对象的非私有域中引用的所有对象同样会被发布。一般来说，如果一个已经发布的对象能够通过非私有的变量引用和方法调用到达其他的对象，那么这些对象也都会被发布。

假定有一个类 C，对于 C 来说，“外部（Alien）方法”是指行为并不完全由 C 来规定的方法，包括其他类中定义的方法以及类 C 中可以被改写的方法（既不是私有 [private] 方法也不是终结 [final] 方法）。当把一个对象传递给某个外部方法时，就相当于发布了这个对象。你无法知道哪些代码会执行，也不知道在外部方法中究竟会发布这个对象，还是会保留对象的引用并在随后由另一个线程使用。

无论其他的线程会对已发布的引用执行何种操作，其实都不重要，因为误用该引用的风险始终存在[⊖]。当某个对象逸出后，你必须假设有某个类或线程可能会误用该对象。这正是需要使用封装的最主要原因：封装能够使得对程序的正确性进行分析变得可能，并使得无意中破坏设计约束条件变得更难。

最后一种发布对象或其内部状态的机制就是发布一个内部的类实例，如程序清单 3-7 的 ThisEscape 所示。当 ThisEscape 发布 EventListener 时，也隐含地发布了 ThisEscape 实例本身，因为在这个内部类的实例中包含了对 ThisEscape 实例的隐含引用。

程序清单 3-7 隐式地使 this 引用逸出（不要这么做）

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

⊖ 如果有人窃取了你的密码并发布到 alt.free-passwords 新闻组上，那么你的信息将“逸出”：无论是否有人会（或者尚未）恶意地使用这些个人信息，你的账户都已经不再安全了。发布一个引用同样会带来类似的风险。

安全的对象构造过程

在 ThisEscape 中给出了逸出的一个特殊示例，即 this 引用在构造函数中逸出。当内部的 EventListener 实例发布时，在外部封装的 ThisEscape 实例也逸出了。当且仅当对象的构造函数返回时，对象才处于可预测的和一致的状态。因此，当从对象的构造函数中发布对象时，只是发布了一个尚未构造完成的对象。即使发布对象的语句位于构造函数的最后一行也是如此。如果 this 引用在构造过程中逸出，那么这种对象就被认为是不正确构造[⊖]。

不要在构造过程中使 this 引用逸出。

在构造过程中使 this 引用逸出的一个常见错误是，在构造函数中启动一个线程。当对象在其构造函数中创建一个线程时，无论是显式创建（通过将它传给构造函数）还是隐式创建（由于 Thread 或 Runnable 是该对象的一个内部类），this 引用都会被新创建的线程共享。在对象尚未完全构造之前，新的线程就可以看见它。在构造函数中创建线程并没有错误，但最好不要立即启动它，而是通过一个 start 或 initialize 方法来启动（请参见第 7 章了解更多关于服务生命周期的内容）。在构造函数中调用一个可改写的实例方法时（既不是私有方法，也不是终结方法），同样会导致 this 引用在构造过程中逸出。

如果想在构造函数中注册一个事件监听器或启动线程，那么可以使用一个私有的构造函数和一个公共的工厂方法（Factory Method），从而避免不正确的构造过程，如程序清单 3-8 中 SafeListener 所示。

程序清单 3-8　使用工厂方法来防止 this 引用在构造过程中逸出

```
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;
    }
}
```

⊖ 具体来说，只有当构造函数返回时，this 引用才应该从线程中逸出。构造函数可以将 this 引用保存到某个地方，只要其他线程不会在构造函数完成之前使用它。在程序清单 3-8 的 SafeListener 中就使用了这种技术。

3.3 线程封闭

当访问共享的可变数据时，通常需要使用同步。一种避免使用同步的方式就是不共享数据。如果仅在单线程内访问数据，就不需要同步。这种技术被称为线程封闭（Thread Confinement），它是实现线程安全性的最简单方式之一。当某个对象封闭在一个线程中时，这种用法将自动实现线程安全性，即使被封闭的对象本身不是线程安全的 [CPJ 2.3.2]。

在 Swing 中大量使用了线程封闭技术。Swing 的可视化组件和数据模型对象都不是线程安全的，Swing 通过将它们封闭到 Swing 的事件分发线程中来实现线程安全性。要想正确地使用 Swing，那么在除了事件线程之外的其他线程中就不能访问这些对象（为了进一步简化对 Swing 的使用，Swing 还提供了 invokeLater 机制，用于将一个 Runnable 实例调度到事件线程中执行）。Swing 应用程序的许多并发错误都是由于错误地在另一个线程中使用了这些被封闭的对象。

线程封闭技术的另一种常见应用是 JDBC(Java Database Connectivity) 的 Connection 对象。JDBC 规范并不要求 Connection 对象必须是线程安全的[⊖]。在典型的服务器应用程序中，线程从连接池中获得一个 Connection 对象，并且用该对象来处理请求，使用完后再将对象返还给连接池。由于大多数请求（例如 Servlet 请求或 EJB 调用等）都是由单个线程采用同步的方式来处理，并且在 Connection 对象返回之前，连接池不会再将它分配给其他线程，因此，这种连接管理模式在处理请求时隐含地将 Connection 对象封闭在线程中。

在 Java 语言中并没有强制规定某个变量必须由锁来保护，同样在 Java 语言中也无法强制将对象封闭在某个线程中。线程封闭是在程序设计中的一个考虑因素，必须在程序中实现。Java 语言及其核心库提供了一些机制来帮助维持线程封闭性，例如局部变量和 ThreadLocal 类，但即便如此，程序员仍然需要负责确保封闭在线程中的对象不会从线程中逸出。

3.3.1 Ad-hoc 线程封闭

Ad-hoc 线程封闭是指，维护线程封闭性的职责完全由程序实现来承担。Ad-hoc 线程封闭是非常脆弱的，因为没有任何一种语言特性，例如可见性修饰符或局部变量，能将对象封闭到目标线程上。事实上，对线程封闭对象（例如，GUI 应用程序中的可视化组件或数据模型等）的引用通常保存在公有变量中。

当决定使用线程封闭技术时，通常是因为要将某个特定的子系统实现为一个单线程子系统。在某些情况下，单线程子系统提供的简便性要胜过 Ad-hoc 线程封闭技术的脆弱性。[⊜]

在 volatile 变量上存在一种特殊的线程封闭。只要你能确保只有单个线程对共享的 volatile 变量执行写入操作，那么就可以安全地在这些共享的 volatile 变量上执行“读取－修改－写入”的操作。在这种情况下，相当于将修改操作封闭在单个线程中以防止发生竞态条件，并且 volatile 变量的可见性保证还确保了其他线程能看到最新的值。

⊖ 应用程序服务器提供的连接池是线程安全的。连接池通常会由多个线程同时访问，因此非线程安全的连接池是毫无意义的。

⊜ 使用单线程子系统的另一个原因是为了避免死锁，这也是大多数 GUI 框架都是单线程的原因。第 9 章将进一步介绍单线程子系统。

由于 Ad-hoc 线程封闭技术的脆弱性，因此在程序中尽量少用它，在可能的情况下，应该使用更强的线程封闭技术（例如，栈封闭或 ThreadLocal 类）。

3.3.2 栈封闭

栈封闭是线程封闭的一种特例，在栈封闭中，只能通过局部变量才能访问对象。正如封装能使得代码更容易维持不变性条件那样，同步变量也能使对象更易于封闭在线程中。局部变量的固有属性之一就是封闭在执行线程中。它们位于执行线程的栈中，其他线程无法访问这个栈。栈封闭（也被称为线程内部使用或者线程局部使用，不要与核心类库中的 ThreadLocal 混淆）比 Ad-hoc 线程封闭更易于维护，也更加健壮。

对于基本类型的局部变量，例如程序清单 3-9 中 loadTheArk 方法的 numPairs，无论如何都不会破坏栈封闭性。由于任何方法都无法获得对基本类型的引用，因此 Java 语言的这种语义就确保了基本类型的局部变量始终封闭在线程内。

程序清单 3-9　基本类型的局部变量与引用变量的线程封闭性

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

    // animals被封闭在方法中，不要使它们逸出！
    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;
}
```

在维持对象引用的栈封闭性时，程序员需要多做一些工作以确保被引用的对象不会逸出。在 loadTheArk 中实例化一个 TreeSet 对象，并将指向该对象的一个引用保存到 animals 中。此时，只有一个引用指向集合 animals，这个引用被封闭在局部变量中，因此也被封闭在执行线程中。然而，如果发布了对集合 animals（或者该对象中的任何内部数据）的引用，那么封闭性将被破坏，并导致对象 animals 的逸出。

如果在线程内部（Within-Thread）上下文中使用非线程安全的对象，那么该对象仍然是线程安全的。然而，要小心的是，只有编写代码的开发人员才知道哪些对象需要被封闭到执行线程中，以及被封闭的对象是否是线程安全的。如果没有明确地说明这些需求，那么后续的维护人员很容易错误地使对象逸出。

3.3.3 ThreadLocal 类

维持线程封闭性的一种更规范方法是使用 ThreadLocal，这个类能使线程中的某个值与保存值的对象关联起来。ThreadLocal 提供了 get 与 set 等访问接口或方法，这些方法为每个使用该变量的线程都存有一份独立的副本，因此 get 总是返回由当前执行线程在调用 set 时设置的最新值。

ThreadLocal 对象通常用于防止对可变的单实例变量（Singleton）或全局变量进行共享。例如，在单线程应用程序中可能会维持一个全局的数据库连接，并在程序启动时初始化这个连接对象，从而避免在调用每个方法时都要传递一个 Connection 对象。由于 JDBC 的连接对象不一定是线程安全的，因此，当多线程应用程序在没有协同的情况下使用全局变量时，就不是线程安全的。通过将 JDBC 的连接保存到 ThreadLocal 对象中，每个线程都会拥有属于自己的连接，如程序清单 3-10 中的 ConnectionHolder 所示。

程序清单 3-10 使用 ThreadLocal 来维持线程封闭性

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

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

当某个频繁执行的操作需要一个临时对象，例如一个缓冲区，而同时又希望避免在每次执行时都重新分配该临时对象，就可以使用这项技术。例如，在 Java 5.0 之前，Integer.toString() 方法使用 ThreadLocal 对象来保存一个 12 字节大小的缓冲区，用于对结果进行格式化，而不是使用共享的静态缓冲区（这需要使用锁机制）或者在每次调用时都分配一个新的缓冲区[⊖]。

当某个线程初次调用 ThreadLocal.get 方法时，就会调用 initialValue 来获取初始值。从概念上看，你可以将 ThreadLocal<T> 视为包含了 Map< Thread,T> 对象，其中保存了特定于该线程的值，但 ThreadLocal 的实现并非如此。这些特定于线程的值保存在 Thread 对象中，当线程终止后，这些值会作为垃圾回收。

假设你需要将一个单线程应用程序移植到多线程环境中，通过将共享的全局变量转换为 ThreadLocal 对象（如果全局变量的语义允许），可以维持线程安全性。然而，如果将应用程序范围内的缓存转换为线程局部的缓存，就不会有太大作用。

在实现应用程序框架时大量使用了 ThreadLocal。例如，在 EJB 调用期间，J2EE 容器需要将一个事务上下文（Transaction Context）与某个执行中的线程关联起来。通过将事务上下文保存在静态的 ThreadLocal 对象中，可以很容易地实现这个功能：当框架代码需要判断当前运行的是哪

⊖ 除非这个操作的执行频率非常高，或者分配操作的开销非常高，否则这项技术不可能带来性能提升。在 Java 5.0 中，这项技术被一种更直接的方式替代，即在每次调用时分配一个新的缓冲区，对于像临时缓冲区这种简单的对象，该技术并没有什么性能优势。

一个事务时，只需从这个 ThreadLocal 对象中读取事务上下文。这种机制很方便，因为它避免了在调用每个方法时都要传递执行上下文信息，然而这也将使用该机制的代码与框架耦合在一起。

开发人员经常滥用 ThreadLocal，例如将所有全局变量都作为 ThreadLocal 对象，或者作为一种“隐藏”方法参数的手段。ThreadLocal 变量类似于全局变量，它能降低代码的可重用性，并在类之间引入隐含的耦合性，因此在使用时要格外小心。

3.4　不变性

满足同步需求的另一种方法是使用不可变对象（Immutable Object）[EJ Item 13]。到目前为止，我们介绍了许多与原子性和可见性相关的问题，例如得到失效数据，丢失更新操作或者观察到某个对象处于不一致的状态等等，都与多线程试图同时访问同一个可变的状态相关。如果对象的状态不会改变，那么这些问题与复杂性也就自然消失了。

如果某个对象在被创建后其状态就不能被修改，那么这个对象就称为不可变对象。线程安全性是不可变对象的固有属性之一，它们的不变性条件是由构造函数创建的，只要它们的状态不改变，那么这些不变性条件就能得以维持。

不可变对象一定是线程安全的。

不可变对象很简单。它们只有一种状态，并且该状态由构造函数来控制。在程序设计中，一个最困难的地方就是判断复杂对象的可能状态。然而，判断不可变对象的状态却很简单。

同样，不可变对象也更加安全。如果将一个可变对象传递给不可信的代码，或者将该对象发布到不可信代码可以访问它的地方，那么就很危险——不可信代码会改变它们的状态，更糟的是，在代码中将保留一个对该对象的引用并稍后在其他线程中修改对象的状态。另一方面，不可变对象不会像这样被恶意代码或者有问题的代码破坏，因此可以安全地共享和发布这些对象，而无须创建保护性的副本 [EJ Item 24]。

虽然在 Java 语言规范和 Java 内存模型中都没有给出不可变性的正式定义，但不可变性并不等于将对象中所有的域都声明为 final 类型，即使对象中所有的域都是 final 类型的，这个对象也仍然是可变的，因为在 final 类型的域中可以保存对可变对象的引用。

当满足以下条件时，对象才是不可变的：

- 对象创建以后其状态就不能修改。
- 对象的所有域都是 final 类型[⊖]。
- 对象是正确创建的（在对象的创建期间，this 引用没有逸出）。

⊖ 从技术上来看，不可变对象并不需要将其所有的域都声明为 final 类型，例如 String 就是这种情况，这就要对类的良性数据竞争（Benign Data Race）情况做精确分析，因此需要深入理解 Java 内存模型。（注意：String 会将散列值的计算推迟到第一次调用 hash Code 时进行，并将计算得到的散列值缓存到非 final 类型的域中，但这种方式之所以可行，是因为这个域有一个非默认的值，并且在每次计算中都得到相同的结果 [因为基于一个不可变的状态]。自己在编写代码时不要这么做。）

在不可变对象的内部仍可以使用可变对象来管理它们的状态，如程序清单3-11中的ThreeStooges所示。尽管保存姓名的Set对象是可变的，但从ThreeStooges的设计中可以看到，在Set对象构造完成后无法对其进行修改。stooges是一个final类型的引用变量，因此所有的对象状态都通过一个final域来访问。最后一个要求是“正确地构造对象”，这个要求很容易满足，因为构造函数能使该引用由除了构造函数及其调用者之外的代码来访问。

程序清单3-11 在可变对象基础上构建的不可变类

```
@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);
    }
}
```

由于程序的状态总在不断地变化，你可能会认为需要使用不可变对象的地方不多，但实际情况并非如此。在“不可变的对象”与“不可变的对象引用”之间存在着差异。保存在不可变对象中的程序状态仍然可以更新，即通过将一个保存新状态的实例来“替换”原有的不可变对象。下一节将给出使用这项技术的示例。⊖

3.4.1 Final域

关键字final可以视为C++中const机制的一种受限版本，用于构造不可变性对象。final类型的域是不能修改的（但如果final域所引用的对象是可变的，那么这些被引用的对象是可以修改的）。然而，在Java内存模型中，final域还有着特殊的语义。final域能确保初始化过程的安全性，从而可以不受限制地访问不可变对象，并在共享这些对象时无须同步。

即使对象是可变的，通过将对象的某些域声明为final类型，仍然可以简化对状态的判断，因此限制对象的可变性也就相当于限制了该对象可能的状态集合。仅包含一个或两个可变状态的“基本不可变”对象仍然比包含多个可变状态的对象简单。通过将域声明为final类型，也相当于告诉维护人员这些域是不会变化的。

⊖ 许多开发人员都担心这种方法会带来性能问题，但这是没有必要的。内存分配的开销比你想象的还要低，并且不可变对象还会带来其他的性能优势，例如减少了对加锁或者保护性副本的需求，以及降低对基于“代”的垃圾收集机制的影响。

正如“除非需要更高的可见性，否则应将所有的域都声明为私有域”[EJ Item 12]是一个良好的编程习惯，“除非需要某个域是可变的，否则应将其声明为final域”也是一个良好的编程习惯。

3.4.2　示例：使用 Volatile 类型来发布不可变对象

在前面的 UnsafeCachingFactorizer 类中，我们尝试用两个 AtomicReferences 变量来保存最新的数值及其因数分解结果，但这种方式并非是线程安全的，因为我们无法以原子方式来同时读取或更新这两个相关的值。同样，用 volatile 类型的变量来保存这些值也不是线程安全的。然而，在某些情况下，不可变对象能提供一种弱形式的原子性。

因式分解 Servlet 将执行两个原子操作：更新缓存的结果，以及通过判断缓存中的数值是否等于请求的数值来决定是否直接读取缓存中的因数分解结果。每当需要对一组相关数据以原子方式执行某个操作时，就可以考虑创建一个不可变的类来包含这些数据，例如程序清单 3-12 中的 OneValueCache[⊖]。

程序清单 3-12　对数值及其因数分解结果进行缓存的不可变容器类

```
@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);
    }
}
```

对于在访问和更新多个相关变量时出现的竞争条件问题，可以通过将这些变量全部保存在一个不可变对象中来消除。如果是一个可变的对象，那么就必须使用锁来确保原子性。如果是一个不可变对象，那么当线程获得了对该对象的引用后，就不必担心另一个线程会修改对象的状态。如果要更新这些变量，那么可以创建一个新的容器对象，但其他使用原有对象的线程仍然会看到对象处于一致的状态。

⊖ 如果在 OneValueCache 和构造函数中没有调用 copyOf，那么 OneValueCache 就不是不可变的。Arrays.copyOf 是在 Java 6 中引入的，同样还可以使用 clone。

程序清单 3-13 中的 VolatileCachedFactorizer 使用了 OneValueCache 来保存缓存的数值及其因数。当一个线程将 volatile 类型的 cache 设置为引用一个新的 OneValueCache 时，其他线程就会立即看到新缓存的数据。

程序清单 3-13　使用指向不可变容器对象的 volatile 类型引用以缓存最新的结果

```
@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);
    }
}
```

与 cache 相关的操作不会相互干扰，因为 OneValueCache 是不可变的，并且在每条相应的代码路径中只会访问它一次。通过使用包含多个状态变量的容器对象来维持不变性条件，并使用一个 volatile 类型的引用来确保可见性，使得 Volatile Cached Factorizer 在没有显式地使用锁的情况下仍然是线程安全的。

3.5　安全发布

到目前为止，我们重点讨论的是如何确保对象不被发布，例如让对象封闭在线程或另一个对象的内部。当然，在某些情况下我们希望在多个线程间共享对象，此时必须确保安全地进行共享。然而，如果只是像程序清单 3-14 那样将对象引用保存到公有域中，那么还不足以安全地发布这个对象。

程序清单 3-14　在没有足够同步的情况下发布对象（不要这么做）

```
// 不安全的发布
public Holder holder;

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

你可能会奇怪，这个看似没有问题的示例何以会运行失败。由于存在可见性问题，其他线程看到的 Holder 对象将处于不一致的状态，即便在该对象的构造函数中已经正确地构建了不变性条件。这种不正确的发布导致其他线程看到尚未创建完成的对象。

3.5.1　不正确的发布：正确的对象被破坏

你不能指望一个尚未被完全创建的对象拥有完整性。某个观察该对象的线程将看到对象处于不一致的状态，然后看到对象的状态突然发生变化，即使线程在对象发布后还没有修改过它。事实上，如果程序清单 3-15 中的 Holder 使用程序清单 3-14 中的不安全发布方式，那么另一个线程在调用 assertSanity 时将抛出 AssertionError。[⊖]

程序清单 3-15　由于未被正确发布，因此这个类可能出现故障

```
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.");
    }
}
```

由于没有使用同步来确保 Holder 对象对其他线程可见，因此将 Holder 称为“未被正确发布”。在未被正确发布的对象中存在两个问题。首先，除了发布对象的线程外，其他线程可以看到的 Holder 域是一个失效值，因此将看到一个空引用或者之前的旧值。然而，更糟糕的情况是，线程看到 Holder 引用的值是最新的，但 Holder 状态的值却是失效的[⊜]。情况变得更加不可预测的是，某个线程在第一次读取域时得到失效值，而再次读取这个域时会得到一个更新值，这也是 assertSainty 抛出 AssertionError 的原因。

如果没有足够的同步，那么当在多个线程间共享数据时将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

由于不可变对象是一种非常重要的对象，因此 Java 内存模型为不可变对象的共享提供了一种特殊的初始化安全性保证。我们已经知道，即使某个对象的引用对其他线程是可见的，也并不意味着对象状态对于使用该对象的线程来说一定是可见的。为了确保对象状态能呈现出一致的视图，就必须使用同步。

另一方面，即使在发布不可变对象的引用时没有使用同步，也仍然可以安全地访问该对象。为了维持这种初始化安全性的保证，必须满足不可变性的所有需求：状态不可修改，所有域都是 final 类型，以及正确的构造过程。（如果程序清单 3-15 中的 Holder 对象是不可变的，那么即使 Holder 没有被正确地发布，在 assertSanity 中也不会抛出 AssertionError。）

⊖ 问题并不在于 Holder 类本身，而是在于 Holder 类未被正确地发布。然而，如果将 n 声明为 final 类型，那么 Holder 将不可变，从而避免出现不正确发布的问题。请参见 3.5.2 节。

⊜ 尽管在构造函数中设置的域值似乎是第一次向这些域中写入的值，因此不会有“更旧的”值被视为失效值，但 Object 的构造函数会在子类构造函数运行之前先将默认值写入所有的域。因此，某个域的默认值可能被视为失效值。

任何线程都可以在不需要额外同步的情况下安全地访问不可变对象，即使在发布这些对象时没有使用同步。

这种保证还将延伸到被正确创建对象中所有 final 类型的域。在没有额外同步的情况下，也可以安全地访问 final 类型的域。然而，如果 final 类型的域所指向的是可变对象，那么在访问这些域所指向的对象的状态时仍然需要同步。

3.5.3 安全发布的常用模式

可变对象必须通过安全的方式来发布，这通常意味着在发布和使用该对象的线程时都必须使用同步。现在，我们将重点介绍如何确保使用对象的线程能够看到该对象处于已发布的状态，并稍后介绍如何在对象发布后对其可见性进行修改。

要安全地发布一个对象，对象的引用以及对象的状态必须同时对其他线程可见。一个正确构造的对象可以通过以下方式来安全地发布：

- 在静态初始化函数中初始化一个对象引用。
- 将对象的引用保存到 volatile 类型的域或者 AtomicReferance 对象中。
- 将对象的引用保存到某个正确构造对象的 final 类型域中。
- 将对象的引用保存到一个由锁保护的域中。

在线程安全容器内部的同步意味着，在将对象放入到某个容器，例如 Vector 或 synchronizedList 时，将满足上述最后一条需求。如果线程 A 将对象 X 放入一个线程安全的容器，随后线程 B 读取这个对象，那么可以确保 B 看到 A 设置的 X 状态，即便在这段读 / 写 X 的应用程序代码中没有包含显式的同步。尽管 Javadoc 在这个主题上没有给出很清晰的说明，但线程安全库中的容器类提供了以下的安全发布保证：

- 通过将一个键或者值放入 Hashtable、synchronizedMap 或者 ConcurrentMap 中，可以安全地将它发布给任何从这些容器中访问它的线程（无论是直接访问还是通过迭代器访问）。
- 通过将某个元素放入 Vector、CopyOnWriteArrayList、CopyOnWriteArraySet、synchronizedList 或 synchronizedSet 中，可以将该元素安全地发布到任何从这些容器中访问该元素的线程。
- 通过将某个元素放入 BlockingQueue 或者 ConcurrentLinkedQueue 中，可以将该元素安全地发布到任何从这些队列中访问该元素的线程。

类库中的其他数据传递机制（例如 Future 和 Exchanger）同样能实现安全发布，在介绍这些机制时将讨论它们的安全发布功能。

通常，要发布一个静态构造的对象，最简单和最安全的方式是使用静态的初始化器：

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

静态初始化器由 JVM 在类的初始化阶段执行。由于在 JVM 内部存在着同步机制，因此通过这种方式初始化的任何对象都可以被安全地发布 [JLS 12.4.2]。

3.5.4　事实不可变对象

如果对象在发布后不会被修改，那么对于其他在没有额外同步的情况下安全地访问这些对象的线程来说，安全发布是足够的。所有的安全发布机制都能确保，当对象的引用对所有访问该对象的线程可见时，对象发布时的状态对于所有线程也将是可见的，并且如果对象状态不会再改变，那么就足以确保任何访问都是安全的。

如果对象从技术上来看是可变的，但其状态在发布后不会再改变，那么把这种对象称为“事实不可变对象（Effectively Immutable Object）”。这些对象不需要满足 3.4 节中提出的不可变性的严格定义。在这些对象发布后，程序只需将它们视为不可变对象即可。通过使用事实不可变对象，不仅可以简化开发过程，而且还能由于减少了同步而提高性能。

> 在没有额外的同步的情况下，任何线程都可以安全地使用被安全发布的事实不可变对象。

例如，Date 本身是可变的[⊖]，但如果将它作为不可变对象来使用，那么在多个线程之间共享 Date 对象时，就可以省去对锁的使用。假设需要维护一个 Map 对象，其中保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 对象的值在被放入 Map 后就不会改变，那么 synchronizedMap 中的同步机制就足以使 Date 值被安全地发布，并且在访问这些 Date 值时不需要额外的同步。

3.5.5　可变对象

如果对象在构造后可以修改，那么安全发布只能确保“发布当时”状态的可见性。对于可变对象，不仅在发布对象时需要使用同步，而且在每次对象访问时同样需要使用同步来确保后续修改操作的可见性。要安全地共享可变对象，这些对象就必须被安全地发布，并且必须是线程安全的或者由某个锁保护起来。

> 对象的发布需求取决于它的可变性：
> - 不可变对象可以通过任意机制来发布。
> - 事实不可变对象必须通过安全方式来发布。
> - 可变对象必须通过安全方式来发布，并且必须是线程安全的或者由某个锁保护起来。

3.5.6　安全地共享对象

当获得对象的一个引用时，你需要知道在这个引用上可以执行哪些操作。在使用它之前是否需要获得一个锁？是否可以修改它的状态，或者只能读取它？许多并发错误都是由于没有理

⊖ 这或许是类库设计中的一个错误。

解共享对象的这些“既定规则”而导致的。当发布一个对象时，必须明确地说明对象的访问方式。

在并发程序中使用和共享对象时，可以使用一些实用的策略，包括：

线程封闭。线程封闭的对象只能由一个线程拥有，对象被封闭在该线程中，并且只能由这个线程修改。

只读共享。在没有额外同步的情况下，共享的只读对象可以由多个线程并发访问，但任何线程都不能修改它。共享的只读对象包括不可变对象和事实不可变对象。

线程安全共享。线程安全的对象在其内部实现同步，因此多个线程可以通过对象的公有接口来进行访问而不需要进一步的同步。

保护对象。被保护的对象只能通过持有特定的锁来访问。保护对象包括封装在其他线程安全对象中的对象，以及已发布的并且由某个特定锁保护的对象。

第④章

对象的组合

到目前为止，我们已经介绍了关于线程安全与同步的一些基础知识。然而，我们并不希望对每一次内存访问都进行分析以确保程序是线程安全的，而是希望将一些现有的线程安全组件组合为更大规模的组件或程序。本章将介绍一些组合模式，这些模式能够使一个类更容易成为线程安全的，并且在维护这些类时不会无意中破坏类的安全性保证。

4.1 设计线程安全的类

在线程安全的程序中，虽然可以将程序的所有状态都保存在公有的静态域中，但与那些将状态封装起来的程序相比，这些程序的线程安全性更难以得到验证，并且在修改时也更难以始终确保其线程安全性。通过使用封装技术，可以使得在不对整个程序进行分析的情况下就可以判断一个类是否是线程安全的。

在设计线程安全类的过程中，需要包含以下三个基本要素：

- 找出构成对象状态的所有变量。
- 找出约束状态变量的不变性条件。
- 建立对象状态的并发访问管理策略。

要分析对象的状态，首先从对象的域开始。如果对象中所有的域都是基本类型的变量，那么这些域将构成对象的全部状态。程序清单 4-1 中的 Counter 只有一个域 value，因此这个域就是 Counter 的全部状态。对于含有 n 个基本类型域的对象，其状态就是这些域构成的 n 元组。例如，二维点的状态就是它的坐标值 (x，y)。如果在对象的域中引用了其他对象，那么该对象的状态将包含被引用对象的域。例如，LinkedList 的状态就包括该链表中所有节点对象的状态。

程序清单 4-1 使用 Java 监视器模式的线程安全计数器

```
@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;
    }
}
```

同步策略（Synchronization Policy）定义了如何在不违背对象不变条件或后验条件的情况下对其状态的访问操作进行协同。同步策略规定了如何将不可变性、线程封闭与加锁机制等结合起来以维护线程的安全性，并且还规定了哪些变量由哪些锁来保护。要确保开发人员可以对这个类进行分析与维护，就必须将同步策略写为正式文档。

4.1.1 收集同步需求

要确保类的线程安全性，就需要确保它的不变性条件不会在并发访问的情况下被破坏，这就需要对其状态进行推断。对象与变量都有一个状态空间，即所有可能的取值。状态空间越小，就越容易判断线程的状态。final 类型的域使用得越多，就越能简化对象可能状态的分析过程。（在极端的情况中，不可变对象只有唯一的状态。）

在许多类中都定义了一些不可变条件，用于判断状态是有效的还是无效的。Counter 中的 value 域是 long 类型的变量，其状态空间为从 Long.MIN_VALUE 到 Long.MAX_VALUE，但 Counter 中 value 在取值范围上存在着一个限制，即不能是负值。

同样，在操作中还会包含一些后验条件来判断状态迁移是否是有效的。如果 Counter 的当前状态为 17，那么下一个有效状态只能是 18。当下一个状态需要依赖当前状态时，这个操作就必须是一个复合操作。并非所有的操作都会在状态转换上施加限制，例如，当更新一个保存当前温度的变量时，该变量之前的状态并不会影响计算结果。

由于不变性条件以及后验条件在状态及状态转换上施加了各种约束，因此就需要额外的同步与封装。如果某些状态是无效的，那么必须对底层的状态变量进行封装，否则客户代码可能会使对象处于无效状态。如果在某个操作中存在无效的状态转换，那么该操作必须是原子的。另外，如果在类中没有施加这种约束，那么就可以放宽封装性或序列化等需求，以便获得更高的灵活性或性能。

在类中也可以包含同时约束多个状态变量的不变性条件。在一个表示数值范围的类（例如程序清单 4-10 中的 NumberRange）中可以包含两个状态变量，分别表示范围的上界和下界。这些变量必须遵循的约束是，下界值应该小于或等于上界值。类似于这种包含多个变量的不变性条件将带来原子性需求：这些相关的变量必须在单个原子操作中进行读取或更新。不能首先更新一个变量，然后释放锁并再次获得锁，然后再更新其他的变量。因为释放锁后，可能会使对象处于无效状态。如果在一个不变性条件中包含多个变量，那么在执行任何访问相关变量的操作时，都必须持有保护这些变量的锁。

> 如果不了解对象的不变性条件与后验条件，那么就不能确保线程安全性。要满足在状态变量的有效值或状态转换上的各种约束条件，就需要借助于原子性与封装性。

4.1.2　依赖状态的操作

类的不变性条件与后验条件约束了在对象上有哪些状态和状态转换是有效的。在某些对象的方法中还包含一些基于状态的先验条件（Precondition）。例如，不能从空队列中移除一个元素，在删除元素前，队列必须处于“非空的”状态。如果在某个操作中包含有基于状态的先验条件，那么这个操作就称为依赖状态的操作。

在单线程程序中，如果某个操作无法满足先验条件，那么就只能失败。但在并发程序中，先验条件可能会由于其他线程执行的操作而变成真。在并发程序中要一直等到先验条件为真，然后再执行该操作。

在 Java 中，等待某个条件为真的各种内置机制（包括等待和通知等机制）都与内置加锁机制紧密关联，要想正确地使用它们并不容易。要想实现某个等待先验条件为真时才执行的操作，一种更简单的方法是通过现有库中的类（例如阻塞队列 [Blocking Queue] 或信号量 [Semaphore]）来实现依赖状态的行为。第 5 章将介绍一些阻塞类，例如 BlockingQueue、Semaphore 以及其他的同步工具类。第 14 章将介绍如何使用在平台与类库中提供的各种底层机制来创建依赖状态的类。

4.1.3　状态的所有权

4.1 节曾指出，如果以某个对象为根节点构造一张对象图，那么该对象的状态将是对象图中所有对象包含的域的一个子集。为什么是一个“子集”？在从对象可以达到的所有域中，需要满足哪些条件才不属于对象状态的一部分？

在定义哪些变量将构成对象的状态时，只考虑对象拥有的数据。所有权（Ownership）在 Java 中并没有得到充分的体现，而是属于类设计中的一个要素。如果分配并填充了一个 HashMap 对象，那么就相当于创建了多个对象：HashMap 对象，在 HashMap 对象中包含的多个对象，以及在 Map.Entry 中可能包含的内部对象。HashMap 对象的逻辑状态包括所有的 Map.Entry 对象以及内部对象，即使这些对象都是一些独立的对象。

无论如何，垃圾回收机制使我们避免了如何处理所有权的问题。在 C++ 中，当把一个对象传递给某个方法时，必须认真考虑这种操作是否传递对象的所有权，是短期的所有权还是长期的所有权。在 Java 中同样存在这些所有权模型，只不过垃圾回收器为我们减少了许多在引用共享方面常见的错误，因此降低了在所有权处理上的开销。

许多情况下，所有权与封装性总是相互关联的：对象封装它拥有的状态，反之也成立，即对它封装的状态拥有所有权。状态变量的所有者将决定采用何种加锁协议来维持变量状态的完整性。所有权意味着控制权。然而，如果发布了某个可变对象的引用，那么就不再拥有独占的控制权，最多是“共享控制权”。对于从构造函数或者从方法中传递进来的对象，类通常并不拥有这些对象，除非这些方法是被专门设计为转移传递进来的对象的所有权（例如，同步容器封装器的工厂方法）。

容器类通常表现出一种“所有权分离”的形式，其中容器类拥有其自身的状态，而客户代码则拥有容器中各个对象的状态。Servlet 框架中的 ServletContext 就是其中一个示

例。ServletContext为Servlet提供了类似于Map形式的对象容器服务，在ServletContext中可以通过名称来注册（setAttribute）或获取（getAttribute）应用程序对象。由Servlet容器实现的ServletContext对象必须是线程安全的，因为它肯定会被多个线程同时访问。当调用setAttribute和getAttribute时，Servlet不需要使用同步，但当使用保存在ServletContext中的对象时，则可能需要使用同步。这些对象由应用程序拥有，Servlet容器只是替应用程序保管它们。与所有共享对象一样，它们必须安全地被共享。为了防止多个线程在并发访问同一个对象时产生的相互干扰，这些对象应该要么是线程安全的对象，要么是事实不可变的对象，或者由锁来保护的对象。⊖

4.2 实例封闭

如果某对象不是线程安全的，那么可以通过多种技术使其在多线程程序中安全地使用。你可以确保该对象只能由单个线程访问（线程封闭），或者通过一个锁来保护对该对象的所有访问。

封装简化了线程安全类的实现过程，它提供了一种实例封闭机制（Instance Confinement），通常也简称为“封闭”[CPJ 2.3.3]。当一个对象被封装到另一个对象中时，能够访问被封装对象的所有代码路径都是已知的。与对象可以由整个程序访问的情况相比，更易于对代码进行分析。通过将封闭机制与合适的加锁策略结合起来，可以确保以线程安全的方式来使用非线程安全的对象。

> 将数据封装在对象内部，可以将数据的访问限制在对象的方法上，从而更容易确保线程在访问数据时总能持有正确的锁。

被封闭对象一定不能超出它们既定的作用域。对象可以封闭在类的一个实例（例如作为类的一个私有成员）中，或者封闭在某个作用域内（例如作为一个局部变量），再或者封闭在线程内（例如在某个线程中将对象从一个方法传递到另一个方法，而不是在多个线程之间共享该对象）。当然，对象本身不会逸出——出现逸出情况的原因通常是由于开发人员在发布对象时超出了对象既定的作用域。

程序清单4-2中的PersonSet说明了如何通过封闭与加锁等机制使一个类成为线程安全的（即使这个类的状态变量并不是线程安全的）。PersonSet的状态由HashSet来管理的，而HashSet并非线程安全的。但由于mySet是私有的并且不会逸出，因此HashSet被封闭在PersonSet中。唯一能访问mySet的代码路径是addPerson与containsPerson，在执行它们时都要获得PersonSet上的锁。PersonSet的状态完全由它的内置锁保护，因而PersonSet是一个线

⊖ 需要注意的是，虽然HttpSession对象在功能上类似于Servlet框架，但可能有着更严格的要求。由于Servlet容器可能需要访问HttpSession中的对象，以便在复制操作或者钝化操作（Passivation，指的是将状态保存到持久性存储）中对它们序列化，因此这些对象必须是线程安全的，因为容器可能与Web Application程序同时访问它们。（之所以说“可能”，是因为在Servlet的规范中并没有明确定义复制与钝化等操作，这只是大多数Servlet容器的一个常见功能。）

程安全的类。

程序清单 4-2　通过封闭机制来确保线程安全

```
@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);
    }
}
```

这个示例并未对 Person 的线程安全性做任何假设，但如果 Person 类是可变的，那么在访问从 PersonSet 中获得的 Person 对象时，还需要额外的同步。要想安全地使用 Person 对象，最可靠的方法就是使 Person 成为一个线程安全的类。另外，也可以使用锁来保护 Person 对象，并确保所有客户代码在访问 Person 对象之前都已经获得正确的锁。

实例封闭是构建线程安全类的一个最简单方式，它还使得在锁策略的选择上拥有了更多的灵活性。在 PersonSet 中使用了它的内置锁来保护它的状态，但对于其他形式的锁来说，只要自始至终都使用同一个锁，就可以保护状态。实例封闭还使得不同的状态变量可以由不同的锁来保护。（后面章节的 ServerStatus 中就使用了多个锁来保护类的状态。）

在 Java 平台的类库中还有很多线程封闭的示例，其中有些类的唯一用途就是将非线程安全的类转化为线程安全的类。一些基本的容器类并非线程安全的，例如 ArrayList 和 HashMap，但类库提供了包装器工厂方法（例如 Collections.synchronizedList 及其类似方法），使得这些非线程安全的类可以在多线程环境中安全地使用。这些工厂方法通过“装饰器（Decorator）”模式 (Gamma et al., 1995) 将容器类封装在一个同步的包装器对象中，而包装器能将接口中的每个方法都实现为同步方法，并将调用请求转发到底层的容器对象上。只要包装器对象拥有对底层容器对象的唯一引用（即把底层容器对象封闭在包装器中），那么它就是线程安全的。在这些方法的 Javadoc 中指出，对底层容器对象的所有访问必须通过包装器来进行。

当然，如果将一个本该被封闭的对象发布出去，那么也能破坏封闭性。如果一个对象本应该封闭在特定的作用域内，那么让该对象逸出作用域就是一个错误。当发布其他对象时，例如迭代器或内部的类实例，可能会间接地发布被封闭对象，同样会使被封闭对象逸出。

封闭机制更易于构造线程安全的类，因为当封闭类的状态时，在分析类的线程安全性时就无须检查整个程序。

4.2.1　Java 监视器模式

从线程封闭原则及其逻辑推论可以得出 Java 监视器模式[⊖]。遵循 Java 监视器模式的对象会把对象的所有可变状态都封装起来，并由对象自己的内置锁来保护。

在程序清单 4-1 的 Counter 中给出了这种模式的一个典型示例。在 Counter 中封装了一个状态变量 value，对该变量的所有访问都需要通过 Counter 的方法来执行，并且这些方法都是同步的。

在许多类中都使用了 Java 监视器模式，例如 Vector 和 Hashtable。在某些情况下，程序需要一种更复杂的同步策略。第 11 章将介绍如何通过细粒度的加锁策略来提高可伸缩性。Java 监视器模式的主要优势就在于它的简单性。

Java 监视器模式仅仅是一种编写代码的约定，对于任何一种锁对象，只要自始至终都使用该锁对象，都可以用来保护对象的状态。程序清单 4-3 给出了如何使用私有锁来保护状态。

程序清单 4-3　通过一个私有锁来保护状态

```
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
        }
    }
}
```

使用私有的锁对象而不是对象的内置锁（或任何其他可通过公有方式访问的锁），有许多优点。私有的锁对象可以将锁封装起来，使客户代码无法得到锁，但客户代码可以通过公有方法来访问锁，以便（正确或者不正确地）参与到它的同步策略中。如果客户代码错误地获得了另一个对象的锁，那么可能会产生活跃性问题。此外，要想验证某个公有访问的锁在程序中是否被正确地使用，则需要检查整个程序，而不是单个的类。

4.2.2　示例：车辆追踪

程序清单 4-1 中的 Counter 是一个简单但用处不大的 Java 监视器模式示例。我们来看一个更有用处的示例：一个用于调度车辆的“车辆追踪器”，例如出租车、警车、货车等。首先使用监视器模式来构建车辆追踪器，然后再尝试放宽某些封装性需求同时又保持线程安全性。

每台车都由一个 String 对象来标识，并且拥有一个相应的位置坐标（x，y）。在 VehicleTracker 类中封装了车辆的标识和位置，因而它非常适合作为基于 MVC (Model-View-Controller，模型－视图－控制器）模式的 GUI 应用程序中的数据模型，并且该模型将由一个视图线程和多

⊖ 虽然 Java 监视器模式来自于 Hoare 对监视器机制的研究工作（Hoare, 1974），但这种模式与真正的监视器类之间存在一些重要的差异。进入和退出同步代码块的字节指令也称为 monitorenter 和 monitorexit，而 Java 的内置锁也称为监视器锁或监视器。

个执行更新操作的线程共享。视图线程会读取车辆的名字和位置，并将它们显示在界面上：

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

类似地，执行更新操作的线程通过从 GPS 设备上获取的数据或者调度员从 GUI 界面上输入的数据来修改车辆的位置。

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

视图线程与执行更新操作的线程将并发地访问数据模型，因此该模型必须是线程安全的。程序清单 4-4 给出了一个基于 Java 监视器模式实现的“车辆追踪器”，其中使用了程序清单 4-5 中的 MutablePoint 来表示车辆的位置。

程序清单 4-4　基于监视器模式的车辆追踪

```
@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 { /*  程序清单 4-5  */ }
```

虽然类 MutablePoint 不是线程安全的，但追踪器类是线程安全的。它所包含的 Map 对象和可变的 Point 对象都未曾发布。当需要返回车辆的位置时，通过 MutablePoint 拷贝构造函数或者 deepCopy 方法来复制正确的值，从而生成一个新的 Map 对象，并且该对象中的值与原有 Map 对象中的 key 值和 value 值都相同。[⊖]

程序清单 4-5　与 Java.awt.Point 类似的可变 Point 类（不要这么做）

```
@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;
    }
}
```

在某种程度上，这种实现方式是通过在返回客户代码之前复制可变的数据来维持线程安全性的。通常情况下，这并不存在性能问题，但在车辆容器非常大的情况下将极大地降低性能[⊖]。此外，由于每次调用 getLocation 就要复制数据，因此将出现一种错误情况——虽然车辆的实际位置发生了变化，但返回的信息却保持不变。这种情况是好还是坏，要取决于你的需求。如果在 location 集合上存在内部的一致性需求，那么这就是优点，在这种情况下返回一致的快照就非常重要。然而，如果调用者需要每辆车的最新信息，那么这就是缺点，因为这需要非常频繁地刷新快照。

4.3　线程安全性的委托

大多数对象都是组合对象。当从头开始构建一个类，或者将多个非线程安全的类组合为一个类时，Java 监视器模式是非常有用的。但是，如果类中的各个组件都已经是线程安全的，会是什么情况呢？我们是否需要再增加一个额外的线程安全层？答案是“视情况而定”。在某些情况下，通过多个线程安全类组合而成的类是线程安全的（如程序清单 4-7 和程序清单 4-9 所示），而在某些情况下，这仅仅是一个好的开端（如程序清单 4-10 所示）。

⊖ 注意，deepCopy 并不只是用 unmodifiableMap 来包装 Map 的，因为这只能防止容器对象被修改，而不能防止调用者修改保存在容器中的可变对象。基于同样的原因，如果只是通过拷贝构造函数来填充 deepCopy 中的 HashMap，那么同样是不正确的，因为这样做只复制了指向 Point 对象的引用，而不是 Point 对象本身。

⊖ 由于 deepCopy 是从一个 synchronized 方法中调用的，因此在执行时间较长的复制操作中，tracker 的内置锁将一直被占有，当有大量车辆需要追踪时，会严重降低用户界面的响应灵敏度。

在前面的CountingFactorizer类中，我们在一个无状态的类中增加了一个AtomicLong类型的域，并且得到的组合对象仍然是线程安全的。由于CountingFactorizer的状态就是AtomicLong的状态，而AtomicLong是线程安全的，因此CountingFactorizer不会对counter的状态施加额外的有效性约束，所以很容易知道CountingFactorizer是线程安全的。我们可以说CountingFactorizer将它的线程安全性委托给AtomicLong来保证：之所以CountingFactorizer是线程安全的，是因为AtomicLong是线程安全的。⊖

4.3.1　示例：基于委托的车辆追踪器

下面将介绍一个更实际的委托示例，构造一个委托给线程安全类的车辆追踪器。我们将车辆的位置保存到一个Map对象中，因此首先要实现一个线程安全的Map类，ConcurrentHashMap。我们还可以用一个不可变的Point类来代替MutablePoint以保存位置，如程序清单4-6所示。

程序清单4-6　在DelegatingVehicleTracker中使用的不可变Point类

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

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

由于Point类是不可变的，因而它是线程安全的。不可变的值可以被自由地共享与发布，因此在返回location时不需要复制。

在程序清单4-7的DelegatingVehicleTracker中没有使用任何显式的同步，所有对状态的访问都由ConcurrentHashMap来管理，而且Map所有的键和值都是不可变的。

程序清单4-7　将线程安全委托给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);
    }
```

⊖ 如果count不是final类型，那么要分析CountingFactorizer的线程安全性将变得更复杂。如果CountingFactorizer将count修改为指向另一个AtomicLong域的引用，那么必须确保count的更新操作对于所有访问count的线程都是可见的，并且还要确保在count的值上不存在竞态条件。这也是尽可能使用final类型域的另一个原因。

```
    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);
    }
}
```

如果使用最初的 MutablePoint 类而不是 Point 类，就会破坏封装性，因为 getLocations 会发布一个指向可变状态的引用，而这个引用不是线程安全的。需要注意的是，我们稍微改变了车辆追踪器类的行为。在使用监视器模式的车辆追踪器中返回的是车辆位置的快照，而在使用委托的车辆追踪器中返回的是一个不可修改但却实时的车辆位置视图。这意味着，如果线程 A 调用 getLocations，而线程 B 在随后修改了某些点的位置，那么在返回给线程 A 的 Map 中将反映出这些变化。在前面提到过，这可能是一种优点（更新的数据），也可能是一种缺点（可能导致不一致的车辆位置视图），具体情况取决于你的需求。

如果需要一个不发生变化的车辆视图，那么 getLocations 可以返回对 locations 这个 Map 对象的一个浅拷贝 (Shallow Copy)。由于 Map 的内容是不可变的，因此只需复制 Map 的结构，而不用复制它的内容，如程序清单 4-8 所示（其中只返回一个 HashMap，因为 getLocations 并不能保证返回一个线程安全的 Map）。

程序清单 4-8 返回 locations 的静态拷贝而非实时拷贝

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

4.3.2 独立的状态变量

到目前为止，这些委托示例都仅仅委托给了单个线程安全的状态变量。我们还可以将线程安全性委托给多个状态变量，只要这些变量是彼此独立的，即组合而成的类并不会在其包含的多个状态变量上增加任何不变性条件。

程序清单 4-9 中的 VisualComponent 是一个图形组件，允许客户程序注册监控鼠标和键盘等事件的监听器。它为每种类型的事件都备有一个已注册监听器列表，因此当某个事件发生时，就会调用相应的监听器。然而，在鼠标事件监听器与键盘事件监听器之间不存在任何关联，二者是彼此独立的，因此 VisualComponent 可以将其线程安全性委托给这两个线程安全的监听器列表。

程序清单 4-9　将线程安全性委托给多个状态变量

```
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 来保存各个监听器列表。它是一个线程安全的链表，特别适用于管理监听器列表（参见 5.2.3 节）。每个链表都是线程安全的，此外，由于各个状态之间不存在耦合关系，因此 VisualComponent 可以将它的线程安全性委托给 mouseListeners 和 keyListeners 等对象。

4.3.3　当委托失效时

大多数组合对象都不会像 VisualComponent 这样简单：在它们的状态变量之间存在着某些不变性条件。程序清单 4-10 中的 NumberRange 使用了两个 AtomicInteger 来管理状态，并且含有一个约束条件，即第一个数值要小于或等于第二个数值。

程序清单 4-10　NumberRange 类并不足以保护它的不变性条件（不要这么做）

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

    public void setLower(int i) {
    // 注意——不安全的“先检查后执行”
        if (i > upper.get())
            throw new IllegalArgumentException(
                    "can't set lower to " + i + " > upper");
        lower.set(i);
    }

    public void setUpper(int i) {
```

```
        // 注意 —— 不安全的“先检查后执行”
        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，没有维持对下界和上界进行约束的不变性条件。setLower 和 setUpper 等方法都尝试维持不变性条件，但却无法做到。setLower 和 setUpper 都是“先检查后执行”的操作，但它们没有使用足够的加锁机制来保证这些操作的原子性。假设取值范围为 (0，10)，如果一个线程调用 setLower(5)，而另一个线程调用 setUpper(4)，那么在一些错误的执行时序中，这两个调用都将通过检查，并且都能设置成功。结果得到的取值范围就是 (5，4)，那么这是一个无效的状态。因此，虽然 AtomicInteger 是线程安全的，但经过组合得到的类却不是。由于状态变量 lower 和 upper 不是彼此独立的，因此 NumberRange 不能将线程安全性委托给它的线程安全状态变量。

NumberRange 可以通过加锁机制来维护不变性条件以确保其线程安全性，例如使用一个锁来保护 lower 和 upper。此外，它还必须避免发布 lower 和 upper，从而防止客户代码破坏其不变性条件。

如果某个类含有复合操作，例如 NumberRange，那么仅靠委托并不足以实现线程安全性。在这种情况下，这个类必须提供自己的加锁机制以保证这些复合操作都是原子操作，除非整个复合操作都可以委托给状态变量。

> 如果一个类是由多个独立且线程安全的状态变量组成，并且在所有的操作中都不包含无效状态转换，那么可以将线程安全性委托给底层的状态变量。

即使 NumberRange 的各个状态组成部分都是线程安全的，也不能确保 NumberRange 的线程安全性，这种问题非常类似于 3.1.4 节介绍的 volatile 变量规则：仅当一个变量参与到包含其他状态变量的不变性条件时，才可以声明为 volatile 类型。

4.3.4 发布底层的状态变量

当把线程安全性委托给某个对象的底层状态变量时，在什么条件下才可以发布这些变量从而使其他类能修改它们？答案仍然取决于在类中对这些变量施加了哪些不变性条件。虽然 Counter 中的 value 域可以为任意整数值，但 Counter 施加的约束条件是只能取正整数，此外递增操作同样约束了下一个状态的有效取值范围。如果将 value 声明为一个公有域，那么客户代码可以将它修改为一个无效值，因此发布 value 会导致这个类出错。另一方面，如果某个变量

表示的是当前温度或者最近登录用户的 ID，那么即使另一个类在某个时刻修改了这个值，也不会破坏任何不变性条件，因此发布这个变量也是可以接受的。（这或许不是个好主意，因为发布可变的变量将对下一步的开发和派生子类带来限制，但不会破坏类的线程安全性。）

> 如果一个状态变量是线程安全的，并且没有任何不变性条件来约束它的值，在变量的操作上也不存在任何不允许的状态转换，那么就可以安全地发布这个变量。

例如，发布 VisualComponent 中的 mouseListeners 或 keyListeners 等变量就是安全的。由于 VisualComponent 并没有在其监听器链表的合法状态上施加任何约束，因此这些域可以声明为公有域或者发布，而不会破坏线程安全性。

4.3.5　示例：发布状态的车辆追踪器

我们来构造车辆追踪器的另一个版本，并在这个版本中发布底层的可变状态。我们需要修改接口以适应这种变化，即使用可变且线程安全的 Point 类。

程序清单 4-11 中的 SafePoint 提供的 get 方法同时获得 x 和 y 的值，并将二者放在一个数组中返回[⊖]。如果为 x 和 y 分别提供 get 方法，那么在获得这两个不同坐标的操作之间，x 和 y 的值发生变化，从而导致调用者看到不一致的值：车辆从来没有到达过位置（x，y）。通过使用 SafePoint，可以构造一个发布其底层可变状态的车辆追踪器，还能确保其线程安全性不被破坏，如程序清单 4-12 中的 PublishingVehicleTracker 类所示。

程序清单 4-11　线程安全且可变的 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;
    }
}
```

⊖ 如果将拷贝构造函数实现为 this (p.x，p.y)，那么会产生竞态条件，而私有构造函数则可以避免这种竞态条件。这是私有构造函数捕获模式（Private Constructor Capture Idiom，Bloch and Gafter，2005）的一个实例。

程序清单 4-12 安全发布底层状态的车辆追踪器

```
@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 将其线程安全性委托给底层的 ConcurrentHashMap，只是 Map 中的元素是线程安全的且可变的 Point，而并非不可变的。getLocation 方法返回底层 Map 对象的一个不可变副本。调用者不能增加或删除车辆，但却可以通过修改返回 Map 中的 SafePoint 值来改变车辆的位置。再次指出，Map 的这种“实时”特性究竟是带来好处还是坏处，仍然取决于实际的需求。PublishingVehicleTracker 是线程安全的，但如果它在车辆位置的有效值上施加了任何约束，那么就不再是线程安全的。如果需要对车辆位置的变化进行判断或者当位置变化时执行一些操作，那么 PublishingVehicleTracker 中采用的方法并不合适。

4.4 在现有的线程安全类中添加功能

Java 类库包含许多有用的“基础模块”类。通常，我们应该优先选择重用这些现有的类而不是创建新的类：重用能降低开发工作量、开发风险（因为现有的类都已经通过测试）以及维护成本。有时候，某个现有的线程安全类能支持我们需要的所有操作，但更多时候，现有的类只能支持大部分的操作，此时就需要在不破坏线程安全性的情况下添加一个新的操作。

例如，假设需要一个线程安全的链表，它需要提供一个原子的“若没有则添加（Put-If-Absent）”的操作。同步的 List 类已经实现了大部分的功能，我们可以根据它提供的 contains 方法和 add 方法来构造一个“若没有则添加”的操作。

“若没有则添加”的概念很简单，在向容器中添加元素前，首先检查该元素是否已经存在，如果存在就不再添加。（回想“先检查再执行”的注意事项。）由于这个类必须是线程安全的，因此就隐含地增加了另一个需求，即“若没有则添加”这个操作必须是原子操作。这意味着，如果在链表中没有包含对象 X，那么在执行两次“若没有则添加”X 后，在容器中只能包含一个 X 对象。然而，如果“若没有则添加”操作不是原子操作，那么在某些执行情况下，有两个线程都将看到 X 不在容器中，并且都执行了添加 X 的操作，从而使容器中包含两个相同的 X 对象。

要添加一个新的原子操作，最安全的方法是修改原始的类，但这通常无法做到，因为你可能无法访问或修改类的源代码。要想修改原始的类，就需要理解代码中的同步策略，这样增加的功能才能与原有的设计保持一致。如果直接将新方法添加到类中，那么意味着实现同步策略的所有代码仍然处于一个源代码文件中，从而更容易理解与维护。

另一种方法是扩展这个类，假定在设计这个类时考虑了可扩展性。程序清单 4-13 中的 BetterVector 对 Vector 进行了扩展，并添加了一个新方法 putIfAbsent。扩展 Vector 很简单，但并非所有的类都像 Vector 那样将状态向子类公开，因此也就不适合采用这种方法。

程序清单 4-13　扩展 Vector 并增加一个“若没有则添加”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;
    }
}
```

“扩展”方法比直接将代码添加到类中更加脆弱，因为现在的同步策略实现被分布到多个单独维护的源代码文件中。如果底层的类改变了同步策略并选择了不同的锁来保护它的状态变量，那么子类会被破坏，因为在同步策略改变后它无法再使用正确的锁来控制对基类状态的并发访问。（在 Vector 的规范中定义了它的同步策略，因此 BetterVector 不存在这个问题。）

4.4.1　客户端加锁机制

对于由 Collections.synchronizedList 封装的 ArrayList，这两种方法在原始类中添加一个方法或者对类进行扩展都行不通，因为客户代码并不知道在同步封装器工厂方法中返回的 List 对象的类型。第三种策略是扩展类的功能，但并不是扩展类本身，而是将扩展代码放入一个“辅助类”中。

程序清单 4-14 实现了一个包含“若没有则添加”操作的辅助类，用于对线程安全的 List 执

行操作，但其中的代码是错误的。

程序清单 4-14　非线程安全的“若没有则添加”（不要这么做）

```
@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;
    }
}
```

为什么这种方式不能实现线程安全性？毕竟，putIfAbsent 已经声明为 synchronized 类型的变量，对不对？问题在于在错误的锁上进行了同步。无论 List 使用哪一个锁来保护它的状态，可以确定的是，这个锁并不是 ListHelper 上的锁。ListHelper 只是带来了同步的假象，尽管所有的链表操作都被声明为 synchronized，但却使用了不同的锁，这意味着 putIfAbsent 相对于 List 的其他操作来说并不是原子的，因此就无法确保当 putIfAbsent 执行时另一个线程不会修改链表。

要想使这个方法能正确执行，必须使 List 在实现客户端加锁或外部加锁时使用同一个锁。客户端加锁是指，对于使用某个对象 X 的客户端代码，使用 X 本身用于保护其状态的锁来保护这段客户代码。要使用客户端加锁，你必须知道对象 X 使用的是哪一个锁。

在 Vector 和同步封装器类的文档中指出，它们通过使用 Vector 或封装器容器的内置锁来支持客户端加锁。程序清单 4-15 给出了在线程安全的 List 上执行 putIfAbsent 操作，其中使用了正确的客户端加锁。

程序清单 4-15　通过客户端加锁来实现“若没有则添加”

```
@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;
        }
    }
}
```

通过添加一个原子操作来扩展类是脆弱的，因为它将类的加锁代码分布到多个类中。然

而，客户端加锁却更加脆弱，因为它将类 C 的加锁代码放到与 C 完全无关的其他类中。当在那些并不承诺遵循加锁策略的类上使用客户端加锁时，要特别小心。

客户端加锁机制与扩展类机制有许多共同点，二者都是将派生类的行为与基类的实现耦合在一起。正如扩展会破坏实现的封装性 [EJ Item 14]，客户端加锁同样会破坏同步策略的封装性。

4.4.2 组合

当为现有的类添加一个原子操作时，有一种更好的方法：组合（Composition）。程序清单 4-16 中的 ImprovedList 通过将 List 对象的操作委托给底层的 List 实例来实现 List 的操作，同时还添加了一个原子的 putIfAbsent 方法。（与 Collections.synchronizedList 和其他容器封装器一样，ImprovedList 假设把某个链表对象传给构造函数以后，客户代码不会再直接使用这个对象，而只能通过 ImprovedList 来访问它。）

程序清单 4-16　通过组合实现“若没有则添加”

```
@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(); }
    // ... 按照类似的方式委托 List 的其他方法
}
```

ImprovedList 通过自身的内置锁增加了一层额外的加锁。它并不关心底层的 List 是否是线程安全的，即使 List 不是线程安全的或者修改了它的加锁实现，ImprovedList 也会提供一致的加锁机制来实现线程安全性。虽然额外的同步层可能导致轻微的性能损失[⊖]，但与模拟另一个对象的加锁策略相比，ImprovedList 更为健壮。事实上，我们使用了 Java 监视器模式来封装现有的 List，并且只要在类中拥有指向底层 List 的唯一外部引用，就能确保线程安全性。

4.5 将同步策略文档化

在维护线程安全性时，文档是最强大的（同时也是最未被充分利用的）工具之一。用户可以通过查阅文档来判断某个类是否是线程安全的，而维护人员也可以通过查阅文档来理解其

⊖ 性能损失很小，因为在底层 List 上的同步不存在竞争，所以速度很快，请参见第 11 章。

中的实现策略，避免在维护过程中破坏安全性。然而，通常人们从文档中获取的信息却是少之又少。

在文档中说明客户代码需要了解的线程安全性保证，以及代码维护人员需要了解的同步策略。

synchronized、volatile 或者任何一个线程安全类都对应于某种同步策略，用于在并发访问时确保数据的完整性。这种策略是程序设计的要素之一，因此应该将其文档化。当然，设计阶段是编写设计决策文档的最佳时间。这之后的几周或几个月后，一些设计细节会逐渐变得模糊，因此一定要在忘记之前将它们记录下来。

在设计同步策略时需要考虑多个方面，例如，将哪些变量声明为 volatile 类型，哪些变量用锁来保护，哪些锁保护哪些变量，哪些变量必须是不可变的或者被封闭在线程中的，哪些操作必须是原子操作等。其中某些方面是严格的实现细节，应该将它们文档化以便于日后的维护。还有一些方面会影响类中加锁行为的外在表现，也应该将其作为规范的一部分写入文档。

最起码，应该保证将类中的线程安全性文档化。它是否是线程安全的？在执行回调时是否持有一个锁？是否有某些特定的锁会影响其行为？不要让客户冒着风险去猜测。如果你不想支持客户端加锁也是可以的，但一定要明确地指出来。如果你希望客户代码能够在类中添加新的原子操作，如 4.4 节所示，那么就需要在文档中说明需要获得哪些锁才能实现安全的原子操作。如果使用锁来保护状态，那么也要将其写入文档以便日后维护，这很简单，只需使用标注 @GuardedBy 即可。如果要使用更复杂的方法来维护线程安全性，那么一定要将它们写入文档，因为维护者通常很难发现它们。

甚至在平台的类库中，线程安全性方面的文档也是很难令人满意。当你阅读某个类的 Javadoc 时，是否曾怀疑过它是否是线程安全的？[⊖]大多数类都没有给出任何提示。许多正式的 Java 技术规范，例如 Servlet 和 JDBC，也没有在它们的文档中给出线程安全性的保证和需求。

尽管我们不应该对规范之外的行为进行猜测，但有时候出于工作需要，将不得不面对各种糟糕的假设。我们是否应该因为某个对象看上去是线程安全的而就假设它是安全的？是否可以假设通过获取对象的锁来确保对象访问的线程安全性？（只有当我们能控制所有访问该对象的代码时，才能使用这种带风险的技术，否则，这只能带来线程安全性的假象。）无论做出哪种选择都难以令人满意。

更糟糕的是，我们的直觉通常是错误的：我们认为“可能是线程安全“的类通常并不是线程安全的。例如，java.text.SimpleDateFormat 并不是线程安全的，但 JDK 1.4 之前的 Javadoc 并没有提到这点。许多开发人员都对这个类不是线程安全的而感到惊讶。有多少程序已经错误地生成了这种非线程安全的对象，并在多线程中使用它？这些程序没有意识到这将在高负载的情况下导致错误的结果。

如果某个类没有明确地声明是线程安全的，那么就不要假设它是线程安全的，从而有

⊖ 如果你从未考虑过这些问题，那么你确实比较乐观。

效地避免类似于SimpleDateFormat的问题。而另一方面，如果不对容器提供对象（例如HttpSession）的线程安全性做某种有问题的假设，也就不可能开发出一个基于Servlet的应用程序。不要使你的客户或同事也做这样的猜测。

解释含糊的文档

许多Java技术规范都没有（或者至少不愿意）说明接口的线程安全性，例如ServletContext、HttpSession或DataSource[⊖]。这些接口是由容器或数据库供应商来实现的，而你通常无法通过查看其实现代码来了解细节功能。此外，你也不希望依赖于某个特定JDBC驱动的实现细节——你希望遵从标准，这样代码可以基于任何一个JDBC驱动工作。但在JDBC的规范中从未出现“线程”和“并发”这些术语，同样在Servlet规范中也很少提到。那么你该做些什么呢？

你只能去猜测。一个提高猜测准确性的方法是，从实现者（例如容器或数据库的供应商）的角度去解释规范，而不是从使用者的角度去解释。Servlet通常是在容器管理的（Container-Managed）线程中调用的，因此可以安全地假设：如果有多个这种线程在运行，那么容器是知道这种情况的。Servlet容器能生成一些为多个Servlet提供服务的对象，例如HttpSession或ServletContext。因此，Servlet容器应该预见到这些对象将被并发访问，因为它创建了多个线程，并且从这些线程中调用像Servlet.service这样的方法，而这个方法很可能会访问ServletContext。

由于这些对象在单线程的上下文中很少是有用的，因此我们不得不假设它们已被实现为线程安全的，即使在规范中没有明确地说明。此外，如果它们需要客户端加锁，那么客户端代码应该在哪个锁上进行同步？在文档中没有说明这一点，而要猜测的话也不知从何猜起。在规范和正式手册中给出的如何访问ServletContext或HttpSession的示例中进一步强调了这种“合理的假设”，并且没有使用任何客户端同步。

另一方面，通过把setAttribute放到ServletContext中或者将HttpSession的对象由Web应用程序拥有，而不是Servlet容器拥有。在Serviet规范中没有给出任何机制来协调对这些共享属性的并发访问。因此，由容器代替Web应用程序来保存这些属性应该是线程安全的，或者是不可变的。如果容器的工作只是代替Web应用程序来保存这些属性，那么当从servlet应用程序代码访问它们时，应该确保它们始终由同一个锁保护。但由于容器可能需要序列化HttpSession中的对象以实现复制或钝化等操作，并且容器不可能知道你的加锁协议，因此你要自己确保这些对象是线程安全的。

可以对JDBC DataSource接口做出类似的推断，该接口表示一个可重用的数据库连接池。DataSource为应用程序提供服务，它在单线程应用程序中没有太大意义。我们很难想象不在多线程情况下使用getConnection。并且，与Servlet一样，在使用DataSource的许多示例代码中，JDBC规范并没有说明需要使用任何客户端加锁。因此，尽管JDBC规范没有说明DataSource是否是线程安全的，或者要求生产商提供线程安全的实现，但同样由于“如果不

⊖ 令我们失望的是，在多次对规范的修订中一直都忽略了这些问题。

这么做将是不可思议的”，所以我们只能假设 DataSource.getConnection 不需要额外的客户端加锁。

另一方面，在 DataSource 分配 JDBC Connection 对象上没有这样的争议，因为在它们返回连接池之前，不会有其他操作将它们共享。因此，如果某个获取 JDBC Connection 对象的操作跨越了多个线程，那么它必须通过同步来保护对 Connection 对象的访问。（大多数应用程序在实现使用 JDBC Connection 对象的操作时，通常都会把 Connection 对象封闭在某个特定的线程中。）

第5章

基础构建模块

第4章介绍了构造线程安全类时采用的一些技术，例如将线程安全性委托给现有的线程安全类。委托是创建线程安全类的一个最有效的策略：只需让现有的线程安全类管理所有的状态即可。

Java平台类库包含了丰富的并发基础构建模块，例如线程安全的容器类以及各种用于协调多个相互协作的线程控制流的同步工具类（Synchronizer）。本章将介绍其中一些最有用的并发构建模块，特别是在Java 5.0和Java 6中引入的一些新模块，以及在使用这些模块来构造并发应用程序时的一些常用模式。

5.1 同步容器类

同步容器类包括Vector和Hashtable，二者是早期JDK的一部分，此外还包括在JDK 1.2中添加的一些功能相似的类，这些同步的封装器类是由Collections.synchronizedXxx等工厂方法创建的。这些类实现线程安全的方式是：将它们的状态封装起来，并对每个公有方法都进行同步，使得每次只有一个线程能访问容器的状态。

5.1.1 同步容器类的问题

同步容器类都是线程安全的，但在某些情况下可能需要额外的客户端加锁来保护复合操作。容器上常见的复合操作包括：迭代（反复访问元素，直到遍历完容器中所有元素）、跳转（根据指定顺序找到当前元素的下一个元素）以及条件运算，例如“若没有则添加”（检查在Map中是否存在键值K，如果没有，就加入二元组(K,V)）。在同步容器类中，这些复合操作在没有客户端加锁的情况下仍然是线程安全的，但当其他线程并发地修改容器时，它们可能会表现出意料之外的行为。

程序清单5-1给出了在Vector中定义的两个方法：getLast和deleteLast，它们都会执行“先检查再运行”操作。每个方法首先都获得数组的大小，然后通过结果来获取或删除最后一个元素。

程序清单5-1 Vector上可能导致混乱结果的复合操作

```
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);
}
```

这些方法看似没有任何问题，从某种程度上来看也确实如此——无论多少个线程同时调用它们，也不破坏 Vector。但从这些方法的调用者角度来看，情况就不同了。如果线程 A 在包含 10 个元素的 Vector 上调用 getLast，同时线程 B 在同一个 Vector 上调用 deleteLast，这些操作的交替执行如图 5-1 所示，getLast 将抛出 ArrayIndexOutOfBoundsException 异常。在调用 size 与调用 getLast 这两个操作之间，Vector 变小了，因此在调用 size 时得到的索引值将不再有效。这种情况很好地遵循了 Vector 的规范——如果请求一个不存在的元素，那么将抛出一个异常。但这并不是 getLast 的调用者所希望得到的结果（即使在并发修改的情况下也不希望看到），除非 Vector 从一开始就是空的。

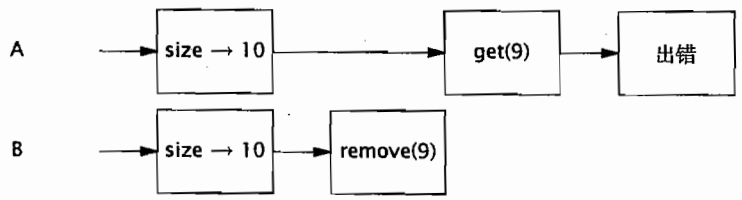


图 5-1　交替调用 getLast 和 deleteLast 时将抛出 ArrayIndexOutOfBoundsException

由于同步容器类要遵守同步策略，即支持客户端加锁[⊖]，因此可能会创建一些新的操作，只要我们知道应该使用哪一个锁，那么这些新操作就与容器的其他操作一样都是原子操作。同步容器类通过其自身的锁来保护它的每个方法。通过获得容器类的锁，我们可以使 getLast 和 deleteLast 成为原子操作，并确保 Vector 的大小在调用 size 和 get 之间不会发生变化，如程序清单 5-2 所示。

程序清单 5-2　在使用客户端加锁的 Vector 上的复合操作

```
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);
    }
}
```

⊖ 这只在 Java 5.0 的 Javadoc 中作为迭代示例简要地提了一下。

在调用 size 和相应的 get 之间，Vector 的长度可能会发生变化，这种风险在对 Vector 中的元素进行迭代时仍然会出现，如程序清单 5-3 所示。

程序清单 5-3　可能抛出 ArrayIndexOutOfBoundsException 的迭代操作

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

这种迭代操作的正确性要依赖于运气，即在调用 size 和 get 之间没有线程会修改 Vector。在单线程环境中，这种假设完全成立，但在有其他线程并发地修改 Vector 时，则可能导致麻烦。与 getLast 一样，如果在对 Vector 进行迭代时，另一个线程删除了一个元素，并且这两个操作交替执行，那么这种迭代方法将抛出 ArrayIndexOutOfBoundsException 异常。

虽然在程序清单 5-3 的迭代操作中可能抛出异常，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，而抛出的异常也与其规范保持一致。然而，像在读取最后一个元素或者迭代等这样的简单操作中抛出异常显然不是人们所期望的。

我们可以通过在客户端加锁来解决不可靠迭代的问题，但要牺牲一些伸缩性。通过在迭代期间持有 Vector 的锁，可以防止其他线程在迭代期间修改 Vector，如程序清单 5-4 所示。然而，这同样会导致其他线程在迭代期间无法访问它，因此降低了并发性。

程序清单 5-4　带有客户端加锁的迭代

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

5.1.2　迭代器与 ConcurrentModificationException

为了将问题阐述清楚，我们使用了 Vector，虽然这是一个“古老”的容器类。然而，许多“现代”的容器类也并没有消除复合操作中的问题。无论在直接迭代还是在 Java 5.0 引入的 for-each 循环语法中，对容器类进行迭代的标准方式都是使用 Iterator。然而，如果有其他线程并发地修改容器，那么即使是使用迭代器也无法避免在迭代期间对容器加锁。在设计同步容器类的迭代器时并没有考虑到并发修改的问题，并且它们表现出的行为是“及时失败”（fail-fast）的。这意味着，当它们发现容器在迭代过程中被修改时，就会抛出一个 ConcurrentModificationException 异常。

这种“及时失败”的迭代器并不是一种完备的处理机制，而只是“善意地”捕获并发错误，因此只能作为并发问题的预警指示器。它们采用的实现方式是，将计数器的变化与容器关联起来：如果在迭代期间计数器被修改，那么 hasNext 或 next 将抛出 ConcurrentModificationException。然而，这种检查是在没有同步的情况下进行的，因此可能会看到失效的计数值，而迭代器可能并没有意识到已经发生了修改。这是一种设计上的权

衡，从而降低并发修改操作的检测代码[⊖]对程序性能带来的影响。

程序清单 5-5 说明了如何使用 for-each 循环语法对 List 容器进行迭代。从内部来看，javac 将生成使用 Iterator 的代码，反复调用 hasNext 和 next 来迭代 List 对象。与迭代 Vector 一样，要想避免出现 ConcurrentModificationException，就必须在迭代过程持有容器的锁。

程序清单 5-5　通过 Iterator 来迭代 List

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

然而，有时候开发人员并不希望在迭代期间对容器加锁。例如，某些线程在可以访问容器之前，必须等待迭代过程结束，如果容器的规模很大，或者在每个元素上执行操作的时间很长，那么这些线程将长时间等待。同样，如果容器像程序清单 5-4 中那样加锁，那么在调用 doSomething 时将持有一个锁，这可能会产生死锁（请参见第 10 章）。即使不存在饥饿或者死锁等风险，长时间地对容器加锁也会降低程序的可伸缩性。持有锁的时间越长，那么在锁上的竞争就可能越激烈，如果许多线程都在等待锁被释放，那么将极大地降低吞吐量和 CPU 的利用率（请参见第 11 章）。

如果不希望在迭代期间对容器加锁，那么一种替代方法就是“克隆”容器，并在副本上进行迭代。由于副本被封闭在线程内，因此其他线程不会在迭代期间对其进行修改，这样就避免了抛出 ConcurrentModificationException（在克隆过程中仍然需要对容器加锁）。在克隆容器时存在显著的性能开销。这种方式的好坏取决于多个因素，包括容器的大小，在每个元素上执行的工作，迭代操作相对于容器其他操作的调用频率，以及在响应时间和吞吐量等方面的需求。

5.1.3　隐藏迭代器

虽然加锁可以防止迭代器抛出 ConcurrentModificationException，但你必须要记住在所有对共享容器进行迭代的地方都需要加锁。实际情况要更加复杂，因为在某些情况下，迭代器会隐藏起来，如程序清单 5-6 中的 HiddenIterator 所示。在 HiddenIterator 中没有显式的迭代操作，但在粗体标出的代码中将执行迭代操作。编译器将字符串的连接操作转换为调用 StringBuilder.append(Object)，而这个方法又会调用容器的 toString 方法，标准容器的 toString 方法将迭代容器，并在每个元素上调用 toString 来生成容器内容的格式化表示。

程序清单 5-6　隐藏在字符串连接中的迭代操作（不要这么做）

```
public class HiddenIterator {
    @GuardedBy("this")
```

⊖ 在单线程代码中也可能抛出 ConcurrentModificationException 异常。当对象直接从容器中删除而不是通过 Iterator.remove 来删除时，就会抛出这个异常。

```
    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);
    }
}
```

addTenThings 方法可能会抛出 ConcurrentModificationException，因为在生成调试消息的过程中，toString 对容器进行迭代。当然，真正的问题在于 HiddenIteracor 不是线程安全的。在使用 println 中的 set 之前必须首先获取 HiddenIterator 的锁，但在调试代码和日志代码中通常会忽视这个要求。

这里得到的教训是，如果状态与保护它的同步代码之间相隔越远，那么开发人员就越容易忘记在访问状态时使用正确的同步。如果 HiddenIterator 用 synchronizedSet 来包装 HashSet，并且对同步代码进行封装，那么就不会发生这种错误。

正如封装对象的状态有助于维持不变性条件一样，封装对象的同步机制同样有助于确保实施同步策略。

容器的 hashCode 和 equals 等方法也会间接地执行迭代操作，当容器作为另一个容器的元素或键值时，就会出现这种情况。同样，containsAll、removeAll 和 retainAll 等方法，以及把容器作为参数的构造函数，都会对容器进行迭代。所有这些间接的迭代操作都可能抛出 ConcurrentModificationException。

5.2　并发容器

Java 5.0 提供了多种并发容器类来改进同步容器的性能。同步容器将所有对容器状态的访问都串行化，以实现它们的线程安全性。这种方法的代价是严重降低并发性，当多个线程竞争容器的锁时，吞吐量将严重减低。

另一方面，并发容器是针对多个线程并发访问设计的。在 Java 5.0 中增加了 ConcurrentHashMap，用来替代同步且基于散列的 Map，以及 CopyOnWriteArrayList，用于在遍历操作为主要操作的情况下代替同步的 List。在新的 ConcurrentMap 接口中增加了对一些常见复合操作的支持，例如“若没有则添加”、替换以及有条件删除等。

通过并发容器来代替同步容器，可以极大地提高伸缩性并降低风险。

Java 5.0 增加了两种新的容器类型：Queue 和 BlockingQueue。Queue 用来临时保存一组等

待处理的元素。它提供了几种实现，包括：ConcurrentLinkedQueue，这是一个传统的先进先出队列，以及 PriorityQueue，这是一个（非并发的）优先队列。Queue 上的操作不会阻塞，如果队列为空，那么获取元素的操作将返回空值。虽然可以用 List 来模拟 Queue 的行为——事实上，正是通过 LinkedList 来实现 Queue 的，但还需要一个 Queue 的类，因为它能去掉 List 的随机访问需求，从而实现更高效的并发。

BlockingQueue 扩展了 Queue，增加了可阻塞的插入和获取等操作。如果队列为空，那么获取元素的操作将一直阻塞，直到队列中出现一个可用的元素。如果队列已满（对于有界队列来说），那么插入元素的操作将一直阻塞，直到队列中出现可用的空间。在“生产者 - 消费者”这种设计模式中，阻塞队列是非常有用的，5.3 节将会详细介绍。

正如 ConcurrentHashMap 用于代替基于散列的同步 Map，Java 6 也引入了 ConcurrentSkipListMap 和 ConcurrentSkipListSet，分别作为同步的 SortedMap 和 SortedSet 的并发替代品（例如用 synchronizedMap 包装的 TreeMap 或 TreeSet）。

5.2.1　ConcurrentHashMap

同步容器类在执行每个操作期间都持有一个锁。在一些操作中，例如 HashMap.get 或 List.contains，可能包含大量的工作：当遍历散列桶或链表来查找某个特定的对象时，必须在许多元素上调用 equals（而 equals 本身还包含一定的计算量）。在基于散列的容器中，如果 hashCode 不能很均匀地分布散列值，那么容器中的元素就不会均匀地分布在整个容器中。某些情况下，某个糟糕的散列函数还会把一个散列表变成线性链表。当遍历很长的链表并且在某些或者全部元素上调用 equals 方法时，会花费很长的时间，而其他线程在这段时间内都不能访问该容器。

与 HashMap 一样，ConcurrentHashMap 也是一个基于散列的 Map，但它使用了一种完全不同的加锁策略来提供更高的并发性和伸缩性。ConcurrentHashMap 并不是将每个方法都在同一个锁上同步并使得每次只能有一个线程访问容器，而是使用一种粒度更细的加锁机制来实现更大程度的共享，这种机制称为分段锁（Lock Striping，请参见 11.4.3 节）。在这种机制中，任意数量的读取线程可以并发地访问 Map，执行读取操作的线程和执行写入操作的线程可以并发地访问 Map，并且一定数量的写入线程可以并发地修改 Map。ConcurrentHashMap 带来的结果是，在并发访问环境下将实现更高的吞吐量，而在单线程环境中只损失非常小的性能。

ConcurrentHashMap 与其他并发容器一起增强了同步容器类：它们提供的迭代器不会抛出 ConcurrentModificationException，因此不需要在迭代过程中对容器加锁。ConcurrentHashMap 返回的迭代器具有弱一致性（Weakly Consistent），而并非“及时失败”。弱一致性的迭代器可以容忍并发的修改，当创建迭代器时会遍历已有的元素，并可以（但是不保证）在迭代器被构造后将修改操作反映给容器。

尽管有这些改进，但仍然有一些需要权衡的因素。对于一些需要在整个 Map 上进行计算的方法，例如 size 和 isEmpty，这些方法的语义被略微减弱了以反映容器的并发特性。由于 size 返回的结果在计算时可能已经过期了，它实际上只是一个估计值，因此允许 size 返回一个近似值而不是一个精确值。虽然这看上去有些令人不安，但事实上 size 和 isEmpty 这样的方法在并

发环境下的用处很小，因为它们的返回值总在不断变化。因此，这些操作的需求被弱化了，以换取对其他更重要操作的性能优化，包括 get、put、containsKey 和 remove 等。

在 ConcurrentHashMap 中没有实现对 Map 加锁以提供独占访问。在 Hashtable 和 synchronizedMap 中，获得 Map 的锁能防止其他线程访问这个 Map。在一些不常见的情况中需要这种功能，例如通过原子方式添加一些映射，或者对 Map 迭代若干次并在此期间保持元素顺序相同。然而，总体来说这种权衡还是合理的，因为并发容器的内容会持续变化。

与 Hashtable 和 synchronizedMap 相比，ConcurrentHashMap 有着更多的优势以及更少的劣势，因此在大多数情况下，用 ConcurrentHashMap 来代替同步 Map 能进一步提高代码的可伸缩性。只有当应用程序需要加锁 Map 以进行独占访问[⊖]时，才应该放弃使用 ConcurrentHashMap。

5.2.2　额外的原子 Map 操作

由于 ConcurrentHashMap 不能被加锁来执行独占访问，因此我们无法使用客户端加锁来创建新的原子操作，例如 4.4.1 节中对 Vector 增加原子操作“若没有则添加”。但是，一些常见的复合操作，例如“若没有则添加”、“若相等则移除（Remove-If-Equal）”和“若相等则替换（Replace-If-Equal）”等，都已经实现为原子操作并且在 ConcurrentMap 的接口中声明，如程序清单 5-7 所示。如果你需要在现有的同步 Map 中添加这样的功能，那么很可能就意味着应该考虑使用 ConcurrentMap 了。

程序清单 5-7　ConcurrentMap 接口

```
public interface ConcurrentMap<K,V> extends Map<K,V> {
//仅当 K 没有相应的映射值时才插入
    V putIfAbsent(K key, V value);

//仅当 K 被映射到 V 时才移除
    boolean remove(K key, V value);

//仅当 K 被映射到 oldValue 时才替换为 newValue
    boolean replace(K key, V oldValue, V newValue);

//仅当 K 被映射到某个值时才替换为 newValue
    V replace(K key, V newValue);
}
```

5.2.3　CopyOnWriteArrayList

CopyOnWriteArrayList 用于替代同步 List，在某些情况下它提供了更好的并发性能，并且在迭代期间不需要对容器进行加锁或复制。（类似地，CopyOnWriteArraySet 的作用是替代同步 Set。）

“写入时复制（Copy-On-Write）”容器的线程安全性在于，只要正确地发布一个事实不可

⊖ 或者需要依赖于同步 Map 带来的一些其他作用。

变的对象，那么在访问该对象时就不再需要进一步的同步。在每次修改时，都会创建并重新发布一个新的容器副本，从而实现可变性。“写入时复制”容器的迭代器保留一个指向底层基础数组的引用，这个数组当前位于迭代器的起始位置，由于它不会被修改，因此在对其进行同步时只需确保数组内容的可见性。因此，多个线程可以同时对这个容器进行迭代，而不会彼此干扰或者与修改容器的线程相互干扰。“写入时复制”容器返回的迭代器不会抛出 ConcurrentModificationException，并且返回的元素与迭代器创建时的元素完全一致，而不必考虑之后修改操作所带来的影响。

显然，每当修改容器时都会复制底层数组，这需要一定的开销，特别是当容器的规模较大时。仅当迭代操作远远多于修改操作时，才应该使用“写入时复制”容器。这个准则很好地描述了许多事件通知系统：在分发通知时需要迭代已注册监听器链表，并调用每一个监听器，在大多数情况下，注册和注销事件监听器的操作远少于接收事件通知的操作。（关于“写入时复制”的更多信息请参见 [CPJ 2.4.4]。）

5.3　阻塞队列和生产者－消费者模式

阻塞队列提供了可阻塞的 put 和 take 方法，以及支持定时的 offer 和 poll 方法。如果队列已经满了，那么 put 方法将阻塞直到有空间可用；如果队列为空，那么 take 方法将会阻塞直到有元素可用。队列可以是有界的也可以是无界的，无界队列永远都不会充满，因此无界队列上的 put 方法也永远不会阻塞。

阻塞队列支持生产者－消费者这种设计模式。该模式将“找出需要完成的工作”与“执行工作”这两个过程分离开来，并把工作项放入一个“待完成”列表中以便在随后处理，而不是找出后立即处理。生产者－消费者模式能简化开发过程，因为它消除了生产者类和消费者类之间的代码依赖性，此外，该模式还将生产数据的过程与使用数据的过程解耦开来以简化工作负载的管理，因为这两个过程在处理数据的速率上有所不同。

在基于阻塞队列构建的生产者－消费者设计中，当数据生成时，生产者把数据放入队列，而当消费者准备处理数据时，将从队列中获取数据。生产者不需要知道消费者的标识或数量，或者它们是否是唯一的生产者，而只需将数据放入队列即可。同样，消费者也不需要知道生产者是谁，或者工作来自何处。BlockingQueue 简化了生产者－消费者设计的实现过程，它支持任意数量的生产者和消费者。一种最常见的生产者－消费者设计模式就是线程池与工作队列的组合，在 Executor 任务执行框架中就体现了这种模式，这也是第 6 章和第 8 章的主题。

以两个人洗盘子为例，二者的劳动分工也是一种生产者－消费者模式：其中一个人把洗好的盘子放在盘架上，而另一个人从盘架上取出盘子并把它们烘干。在这个示例中，盘架相当于阻塞队列。如果盘架上没有盘子，那么消费者会一直等待，直到有盘子需要烘干。如果盘架放满了，那么生产者会停止清洗直到盘架上有更多的空间。我们可以将这种类比扩展为多个生产者（虽然可能存在对水槽的竞争）和多个消费者，每个工人只需与盘架打交道。人们不需要知道究竟有多少生产者或消费者，或者谁生产了某个指定的工作项。

“生产者”和“消费者”的角色是相对的，某种环境中的消费者在另一种不同的环境中可能会成为生产者。烘干盘子的工人将“消费”洗干净的湿盘子，而产生烘干的盘子。第三个人

把洗干净的盘子整理好，在这种情况中，烘干盘子的工人既是消费者，也是生产者，从而就有了两个共享的工作队列（每个队列都可能阻塞烘干工作的运行）。

阻塞队列简化了消费者程序的编码，因为 take 操作会一直阻塞直到有可用的数据。如果生产者不能尽快地产生工作项使消费者保持忙碌，那么消费者就只能一直等待，直到有工作可做。在某些情况下，这种方式是非常合适的（例如，在服务器应用程序中，没有任何客户请求服务），而在其他一些情况下，这也表示需要调整生产者线程数量和消费者线程数量之间的比率，从而实现更高的资源利用率（例如，在“网页爬虫 [Web Crawler]”或其他应用程序中，有无穷的工作需要完成）。

如果生产者生成工作的速率比消费者处理工作的速率快，那么工作项会在队列中累积起来，最终耗尽内存。同样，put 方法的阻塞特性也极大地简化了生产者的编码。如果使用有界队列，那么当队列充满时，生产者将阻塞并且不能继续生成工作，而消费者就有时间来赶上工作处理进度。

阻塞队列同样提供了一个 offer 方法，如果数据项不能被添加到队列中，那么将返回一个失败状态。这样你就能够创建更多灵活的策略来处理负荷过载的情况，例如减轻负载，将多余的工作项序列化并写入磁盘，减少生产者线程的数量，或者通过某种方式来抑制生产者线程。

> 在构建高可靠的应用程序时，有界队列是一种强大的资源管理工具：它们能抑制并防止产生过多的工作项，使应用程序在负荷过载的情况下变得更加健壮。

虽然生产者 – 消费者模式能够将生产者和消费者的代码彼此解耦开来，但它们的行为仍然会通过共享工作队列间接地耦合在一起。开发人员总会假设消费者处理工作的速率能赶上生产者生成工作项的速率，因此通常不会为工作队列的大小设置边界，但这将导致在之后需要重新设计系统架构。因此，应该尽早地通过阻塞队列在设计中构建资源管理机制——这件事情做得越早，就越容易。在许多情况下，阻塞队列能使这项工作更加简单，如果阻塞队列并不完全符合设计需求，那么还可以通过信号量 (Semaphore) 来创建其他的阻塞数据结构（请参见 5.5.3 节）。

在类库中包含了 BlockingQueue 的多种实现，其中，LinkedBlockingQueue 和 ArrayBlockingQueue 是 FIFO 队列，二者分别与 LinkedList 和 ArrayList 类似，但比同步 List 拥有更好的并发性能。PriorityBlockingQueue 是一个按优先级排序的队列，当你希望按照某种顺序而不是 FIFO 来处理元素时，这个队列将非常有用。正如其他有序的容器一样，PriorityBlockingQueue 既可以根据元素的自然顺序来比较元素（如果它们实现了 Comparable 方法），也可以使用 Comparator 来比较。

最后一个 BlockingQueue 实现是 SynchronousQueue，实际上它不是一个真正的队列，因为它不会为队列中元素维护存储空间。与其他队列不同的是，它维护一组线程，这些线程在等待着把元素加入或移出队列。如果以洗盘子的比喻为例，那么这就相当于没有盘架，而是将洗好的盘子直接放入下一个空闲的烘干机中。这种实现队列的方式看似很奇怪，但由于可以直接交付工作，从而降低了将数据从生产者移动到消费者的延迟。（在传统的队列中，在一个工作单元可以交付之前，必须通过串行方式首先完成入列 [Enqueue] 或者出列 [Dequeue] 等操作。）直

接交付方式还会将更多关于任务状态的信息反馈给生产者。当交付被接受时，它就知道消费者已经得到了任务，而不是简单地把任务放入一个队列——这种区别就好比将文件直接交给同事，还是将文件放到她的邮箱中并希望她能尽快拿到文件。因为 SynchronousQueue 没有存储功能，因此 put 和 take 会一直阻塞，直到有另一个线程已经准备好参与到交付过程中。仅当有足够多的消费者，并且总是有一个消费者准备好获取交付的工作时，才适合使用同步队列。

5.3.1　示例：桌面搜索

有一种类型的程序适合被分解为生产者和消费者，例如代理程序，它将扫描本地驱动器上的文件并建立索引以便随后进行搜索，类似于某些桌面搜索程序或者 Windows 索引服务。在程序清单 5-8 的 DiskCrawler 中给出了一个生产者任务，即在某个文件层次结构中搜索符合索引标准的文件，并将它们的名称放入工作队列。而且，在 Indexer 中还给出了一个消费者任务，即从队列中取出文件名称并对它们建立索引。

程序清单 5-8　桌面搜索应用程序中的生产者任务和消费者任务

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

生产者－消费者模式提供了一种适合线程的方法将桌面搜索问题分解为更简单的组件。将文件遍历与建立索引等功能分解为独立的操作，比将所有功能都放到一个操作中实现有着更高的代码可读性和可重用性：每个操作只需完成一个任务，并且阻塞队列将负责所有的控制流，因此每个功能的代码都更加简单和清晰。

生产者－消费者模式同样能带来许多性能优势。生产者和消费者可以并发地执行。如果一个是 I/O 密集型，另一个是 CPU 密集型，那么并发执行的吞吐率要高于串行执行的吞吐率。如果生产者和消费者的并行度不同，那么将它们紧密耦合在一起会把整体并行度降低为二者中更小的并行度。

在程序清单 5-9 中启动了多个爬虫程序和索引建立程序，每个程序都在各自的线程中运行。前面曾讲，消费者线程永远不会退出，因而程序无法终止，第 7 章将介绍多种技术来解决这个问题。虽然这个示例使用了显式管理的线程，但许多生产者－消费者设计也可以通过 Executor 任务执行框架来实现，其本身也使用了生产者－消费者模式。

程序清单 5-9　启动桌面搜索

```
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();
}
```

5.3.2　串行线程封闭

在 java.util.concurrent 中实现的各种阻塞队列都包含了足够的内部同步机制，从而安全地将对象从生产者线程发布到消费者线程。

对于可变对象，生产者－消费者这种设计与阻塞队列一起，促进了串行线程封闭，从而将对象所有权从生产者交付给消费者。线程封闭对象只能由单个线程拥有，但可以通过安全地发布该对象来“转移”所有权。在转移所有权后，也只有另一个线程能获得这个对象的访问权限，并且发布对象的线程不会再访问它。这种安全的发布确保了对象状态对于新的所有者来说是可见的，并且由于最初的所有者不会再访问它，因此对象将被封闭在新的线程中。新的所有

者线程可以对该对象做任意修改，因为它具有独占的访问权。

对象池利用了串行线程封闭，将对象“借给”一个请求线程。只要对象池包含足够的内部同步来安全地发布池中的对象，并且只要客户代码本身不会发布池中的对象，或者在将对象返回给对象池后就不再使用它，那么就可以安全地在线程之间传递所有权。

我们也可以使用其他发布机制来传递可变对象的所有权，但必须确保只有一个线程能接受被转移的对象。阻塞队列简化了这项工作。除此之外，还可以通过 ConcurrentMap 的原子方法 remove 或者 AtomicReference 的原子方法 compareAndSet 来完成这项工作。

5.3.3 双端队列与工作密取

Java 6 增加了两种容器类型，Deque（发音为“deck”）和 BlockingDeque，它们分别对 Queue 和 BlockingQueue 进行了扩展。Deque 是一个双端队列，实现了在队列头和队列尾的高效插入和移除。具体实现包括 ArrayDeque 和 LinkedBlockingDeque。

正如阻塞队列适用于生产者－消费者模式，双端队列同样适用于另一种相关模式，即工作密取（Work Stealing）。在生产者－消费者设计中，所有消费者有一个共享的工作队列，而在工作密取设计中，每个消费者都有各自的双端队列。如果一个消费者完成了自己双端队列中的全部工作，那么它可以从其他消费者双端队列末尾秘密地获取工作。密取工作模式比传统的生产者－消费者模式具有更高的可伸缩性，这是因为工作者线程不会在单个共享的任务队列上发生竞争。在大多数时候，它们都只是访问自己的双端队列，从而极大地减少了竞争。当工作者线程需要访问另一个队列时，它会从队列的尾部而不是从头部获取工作，因此进一步降低了队列上的竞争程度。

工作密取非常适用于既是消费者也是生产者问题——当执行某个工作时可能导致出现更多的工作。例如，在网页爬虫程序中处理一个页面时，通常会发现有更多的页面需要处理。类似的还有许多搜索图的算法，例如在垃圾回收阶段对堆进行标记，都可以通过工作密取机制来实现高效并行。当一个工作线程找到新的任务单元时，它会将其放到自己队列的末尾（或者在工作共享设计模式中，放入其他工作者线程的队列中）。当双端队列为空时，它会在另一个线程的队列队尾查找新的任务，从而确保每个线程都保持忙碌状态。

5.4 阻塞方法与中断方法

线程可能会阻塞或暂停执行，原因有多种：等待 I/O 操作结束，等待获得一个锁，等待从 Thread.sleep 方法中醒来，或是等待另一个线程的计算结果。当线程阻塞时，它通常被挂起，并处于某种阻塞状态（BLOCKED、WAITING 或 TIMED_WAITING）。阻塞操作与执行时间很长的普通操作的差别在于，被阻塞的线程必须等待某个不受它控制的事件发生后才能继续执行，例如等待 I/O 操作完成，等待某个锁变成可用，或者等待外部计算的结束。当某个外部事件发生时，线程被置回 RUNNABLE 状态，并可以再次被调度执行。

BlockingQueue 的 put 和 take 等方法会抛出受检查异常（Checked Exception）InterruptedException，这与类库中其他一些方法的做法相同，例如 Thread.sleep。当某方法抛出 Interrupted-

Exception 时，表示该方法是一个阻塞方法，如果这个方法被中断，那么它将努力提前结束阻塞状态。

Thread 提供了 interrupt 方法，用于中断线程或者查询线程是否已经被中断。每个线程都有一个布尔类型的属性，表示线程的中断状态，当中断线程时将设置这个状态。

中断是一种协作机制。一个线程不能强制其他线程停止正在执行的操作而去执行其他的操作。当线程 A 中断 B 时，A 仅仅是要求 B 在执行到某个可以暂停的地方停止正在执行的操作——前提是如果线程 B 愿意停止下来。虽然在 API 或者语言规范中并没有为中断定义任何特定应用级别的语义，但最常使用中断的情况就是取消某个操作。方法对中断请求的响应度越高，就越容易及时取消那些执行时间很长的操作。

当在代码中调用了一个将抛出 InterruptedException 异常的方法时，你自己的方法也就变成了一个阻塞方法，并且必须要处理对中断的响应。对于库代码来说，有两种基本选择：

传递 InterruptedException。避开这个异常通常是最明智的策略——只需把 InterruptedException 传递给方法的调用者。传递 InterruptedException 的方法包括，根本不捕获该异常，或者捕获该异常，然后在执行某种简单的清理工作后再次抛出这个异常。

恢复中断。有时候不能抛出 InterruptedException，例如当代码是 Runnable 的一部分时。在这些情况下，必须捕获 InterruptedException，并通过调用当前线程上的 interrupt 方法恢复中断状态，这样在调用栈中更高层的代码将看到引发了一个中断，如程序清单 5-10 所示。

程序清单 5-10　恢复中断状态以避免屏蔽中断

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

还可以采用一些更复杂的中断处理方法，但上述两种方法已经可以应付大多数情况了。然而在出现 InterruptedException 时不应该做的事情是，捕获它但不做出任何响应。这将使调用栈上更高层的代码无法对中断采取处理措施，因为线程被中断的证据已经丢失。只有在一种特殊的情况中才能屏蔽中断，即对 Thread 进行扩展，并且能控制调用栈上所有更高层的代码。第 7 章将进一步介绍取消和中断等操作。

5.5　同步工具类

在容器类中，阻塞队列是一种独特的类：它们不仅能作为保存对象的容器，还能协调生产者和消费者等线程之间的控制流，因为 take 和 put 等方法将阻塞，直到队列达到期望的状态

(队列既非空，也非满)。

同步工具类可以是任何一个对象，只要它根据其自身的状态来协调线程的控制流。阻塞队列可以作为同步工具类，其他类型的同步工具类还包括信号量（Semaphore）、栅栏（Barrier）以及闭锁（Latch）。在平台类库中还包含其他一些同步工具类的类，如果这些类还无法满足需要，那么可以按照第 14 章中给出的机制来创建自己的同步工具类。

所有的同步工具类都包含一些特定的结构化属性：它们封装了一些状态，这些状态将决定执行同步工具类的线程是继续执行还是等待，此外还提供了一些方法对状态进行操作，以及另一些方法用于高效地等待同步工具类进入到预期状态。

5.5.1　闭锁

闭锁是一种同步工具类，可以延迟线程的进度直到其到达终止状态 [CPJ 3.4.2]。闭锁的作用相当于一扇门：在闭锁到达结束状态之前，这扇门一直是关闭的，并且没有任何线程能通过，当到达结束状态时，这扇门会打开并允许所有的线程通过。当闭锁到达结束状态后，将不会再改变状态，因此这扇门将永远保持打开状态。闭锁可以用来确保某些活动直到其他活动都完成后才继续执行，例如：

- 确保某个计算在其需要的所有资源都被初始化之后才继续执行。二元闭锁（包括两个状态）可以用来表示“资源 R 已经被初始化”，而所有需要 R 的操作都必须先在这个闭锁上等待。
- 确保某个服务在其依赖的所有其他服务都已经启动之后才启动。每个服务都有一个相关的二元闭锁。当启动服务 S 时，将首先在 S 依赖的其他服务的闭锁上等待，在所有依赖的服务都启动后会释放闭锁 S，这样其他依赖 S 的服务才能继续执行。
- 等待直到某个操作的所有参与者（例如，在多玩家游戏中的所有玩家）都就绪再继续执行。在这种情况中，当所有玩家都准备就绪时，闭锁将到达结束状态。

CountDownLatch 是一种灵活的闭锁实现，可以在上述各种情况中使用，它可以使一个或多个线程等待一组事件发生。闭锁状态包括一个计数器，该计数器被初始化为一个正数，表示需要等待的事件数量。countDown 方法递减计数器，表示有一个事件已经发生了，而 await 方法等待计数器达到零，这表示所有需要等待的事件都已经发生。如果计数器的值非零，那么 await 会一直阻塞直到计数器为零，或者等待中的线程中断，或者等待超时。

在程序清单 5-11 的 TestHarness 中给出了闭锁的两种常见用法。TestHarness 创建一定数量的线程，利用它们并发地执行指定的任务。它使用两个闭锁，分别表示“起始门（Starting Gate）”和“结束门（Ending Gate）”。起始门计数器的初始值为 1，而结束门计数器的初始值为工作线程的数量。每个工作线程首先要做的值就是在启动门上等待，从而确保所有线程都就绪后才开始执行。而每个线程要做的最后一件事情是将调用结束门的 countDown 方法减 1，这能使主线程高效地等待直到所有工作线程都执行完成，因此可以统计所消耗的时间。

程序清单 5-11　在计时测试中使用 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;
    }
}
```

为什么要在 TestHarness 中使用闭锁，而不是在线程创建后就立即启动？或许，我们希望测试 n 个线程并发执行某个任务时需要的时间。如果在创建线程后立即启动它们，那么先启动的线程将“领先”后启动的线程，并且活跃线程数量会随着时间的推移而增加或减少，竞争程度也在不断发生变化。启动门将使得主线程能够同时释放所有工作线程，而结束门则使主线程能够等待最后一个线程执行完成，而不是顺序地等待每个线程执行完成。

5.5.2　FutureTask

FutureTask 也可以用做闭锁。（FutureTask 实现了 Future 语义，表示一种抽象的可生成结果的计算 [CPJ 4.3.3]）。FutureTask 表示的计算是通过 Callable 来实现的，相当于一种可生成结果的 Runnable，并且可以处于以下 3 种状态：等待运行（Waiting to run），正在运行（Running）和运行完成（Completed）。“执行完成”表示计算的所有可能结束方式，包括正常结束、由于取消而结束和由于异常而结束等。当 FutureTask 进入完成状态后，它会永远停止在这个状态上。

Future.get 的行为取决于任务的状态。如果任务已经完成，那么 get 会立即返回结果，否则 get 将阻塞直到任务进入完成状态，然后返回结果或者抛出异常。FutureTask 将计算结果从执行计算的线程传递到获取这个结果的线程，而 FutureTask 的规范确保了这种传递过程能实现结果的安全发布。

FutureTask 在 Executor 框架中表示异步任务，此外还可以用来表示一些时间较长的计算，这些计算可以在使用计算结果之前启动。程序清单 5-12 中的 Preloader 就使用了 FutureTask 来执行一个高开销的计算，并且计算结果将在稍后使用。通过提前启动计算，可以减少在等待结果时需要的时间。

程序清单 5-12　使用 FutureTask 来提前加载稍后需要的数据

```
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，其中包含从数据库加载产品信息的任务，以及一个执行运算的线程。由于在构造函数或静态初始化方法中启动线程并不是一种好方法，因此提供了一个 start 方法来启动线程。当程序随后需要 ProductInfo 时，可以调用 get 方法，如果数据已经加载，那么将返回这些数据，否则将等待加载完成后再返回。

Callable 表示的任务可以抛出受检查的或未受检查的异常，并且任何代码都可能抛出一个 Error。无论任务代码抛出什么异常，都会被封装到一个 ExecutionException 中，并在 Future.get 中被重新抛出。这将使调用 get 的代码变得复杂，因为它不仅需要处理可能出现的 ExecutionException（以及未检查的 CancellationException），而且还由于 ExecutionException 是作为一个 Throwable 类返回的，因此处理起来并不容易。

在 Preloader 中，当 get 方法抛出 ExecutionException 时，可能是以下三种情况之一：Callable 抛出的受检查异常，RuntimeException，以及 Error。我们必须对每种情况进行单独处理，但我们将使用程序清单 5-13 中的 launderThrowable 辅助方法来封装一些复杂的异常处理逻辑。在调用 launderThrowable 之前，Preloader 会首先检查已知的受检查异常，并重新抛出它们。剩下的是未检查异常，Preloader 将调用 launderThrowable 并抛出结果。如果 Throwable 传递给

launderThrowable 的是一个 Error，那么 launderThrowable 将直接再次抛出它；如果不是 RuntimeException，那么将抛出一个 IllegalStateException 表示这是一个逻辑错误。剩下的 RuntimeException，launderThrowable 将把它们返回给调用者，而调用者通常会重新抛出它们。

程序清单 5-13　强制将未检查的 Throwable 转换为 RuntimeException

```
    /** 如果Throwable是Error，那么抛出它；如果是RuntimeException，那么返回它，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）用来控制同时访问某个特定资源的操作数量，或者同时执行某个指定操作的数量 [CPJ 3.4.1]。计数信号量还可以用来实现某种资源池，或者对容器施加边界。

Semaphore 中管理着一组虚拟的许可 (permit)，许可的初始数量可通过构造函数来指定。在执行操作时可以首先获得许可（只要还有剩余的许可），并在使用以后释放许可。如果没有许可，那么 acquire 将阻塞直到有许可（或者直到被中断或者操作超时）。release 方法将返回一个许可给信号量。[⊖]计算信号量的一种简化形式是二值信号量，即初始值为 1 的 Semaphore。二值信号量可以用做互斥体 (mutex)，并具备不可重入的加锁语义：谁拥有这个唯一的许可，谁就拥有了互斥锁。

Semaphore 可以用于实现资源池，例如数据库连接池。我们可以构造一个固定长度的资源池，当池为空时，请求资源将会失败，但你真正希望看到的行为是阻塞而不是失败，并且当池非空时解除阻塞。如果将 Semaphore 的计数值初始化为池的大小，并在从池中获取一个资源之前首先调用 acquire 方法获取一个许可，在将资源返回给池之后调用 release 释放许可，那么 acquire 将一直阻塞直到资源池不为空。在第 12 章的有界缓冲类中将使用这项技术。（在构造阻塞对象池时，一种更简单的方法是使用 BlockingQueue 来保存池的资源。）

同样，你也可以使用 Semaphore 将任何一种容器变成有界阻塞容器，如程序清单 5-14 中的 BoundedHashSet 所示。信号量的计数值会初始化为容器容量的最大值。add 操作在向底层容器中添加一个元素之前，首先要获取一个许可。如果 add 操作没有添加任何元素，那么会立刻

⊖ 在这种实现中不包含真正的许可对象，并且 Semaphore 也不会将许可与线程关联起来，因此在一个线程中获得的许可可以在另一个线程中释放。可以将 acquire 操作视为是消费一个许可，而 release 操作是创建一个许可，Semaphore 并不受限于它在创建时的初始许可数量。

释放许可。同样，remove 操作释放一个许可，使更多的元素能够添加到容器中。底层的 Set 实现并不知道关于边界的任何信息，这是由 BoundedHashSet 来处理的。

程序清单 5-14　使用 Semaphore 为容器设置边界

```
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;
    }
}
```

5.5.4　栅栏

我们已经看到通过闭锁来启动一组相关的操作，或者等待一组相关的操作结束。闭锁是一次性对象，一旦进入终止状态，就不能被重置。

栅栏 (Barrier) 类似于闭锁，它能阻塞一组线程直到某个事件发生 [CPJ 4,4.3]。栅栏与闭锁的关键区别在于，所有线程必须同时到达栅栏位置，才能继续执行。闭锁用于等待事件，而栅栏用于等待其他线程。栅栏用于实现一些协议，例如几个家庭决定在某个地方集合："所有人 6:00 在麦当劳碰头，到了以后要等其他人，之后再讨论下一步要做的事情。"

CyclicBarrier 可以使一定数量的参与方反复地在栅栏位置汇集，它在并行迭代算法中非常有用：这种算法通常将一个问题拆分成一系列相互独立的子问题。当线程到达栅栏位置时将调用 await 方法，这个方法将阻塞直到所有线程都到达栅栏位置。如果所有线程都到达了栅栏位置，那么栅栏将打开，此时所有线程都被释放，而栅栏将被重置以便下次使用。如果对 await 的调用超时，或者 await 阻塞的线程被中断，那么栅栏就被认为是打破了，所有阻塞的 await 调

用都将终止并抛出 BrokenBarrierException。如果成功地通过栅栏，那么 await 将为每个线程返回一个唯一的到达索引号，我们可以利用这些索引来“选举”产生一个领导线程，并在下一次迭代中由该领导线程执行一些特殊的工作。CyclicBarrier 还可以使你将一个栅栏操作传递给构造函数，这是一个 Runnable，当成功通过栅栏时会（在一个子任务线程中）执行它，但在阻塞线程被释放之前是不能执行的。

在模拟程序中通常需要使用栅栏，例如某个步骤中的计算可以并行执行，但必须等到该步骤中的所有计算都执行完毕才能进入下一个步骤。例如，在 n-body 粒子模拟系统中，每个步骤都根据其他粒子的位置和属性来计算各个粒子的新位置。通过在每两次更新之间等待栅栏，能够确保在第 k 步中的所有更新操作都已经计算完毕，才进入第 k+1 步。

在程序清单 5-15 的 CellularAutomata 中给出了如何通过栅栏来计算细胞的自动化模拟，例如 Conway 的生命游戏 (Gardner，1970)。在把模拟过程并行化时，为每个元素（在这个示例中相当于一个细胞）分配一个独立的线程是不现实的，因为这将产生过多的线程，而在协调这些线程上导致的开销将降低计算性能。合理的做法是，将问题分解成一定数量的子问题，为每个子问题分配一个线程来进行求解，之后再将所有的结果合并起来。CellularAutomata 将问题分解为 N_{cpu} 个子问题，其中 N_{cpu} 等于可用 CPU 的数量，并将每个子问题分配给一个线程。⊖ 在每个步骤中，工作线程都为各自子问题中的所有细胞计算新值。当所有工作线程都到达栅栏时，栅栏会把这些新值提交给数据模型。在栅栏的操作执行完以后，工作线程将开始下一步的计算，包括调用 isDone 方法来判断是否需要进行下一次迭代。

程序清单 5-15　通过 CyclicBarrier 协调细胞自动衍生系统中的计算

```
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;
```

⊖ 在这种不涉及 I/O 操作或共享数据访问的计算问题中，当线程数量为 N_{cpu} 或 $N_{cpu}+1$ 时将获得最优的吞吐量。更多的线程并不会带来任何帮助，甚至在某种程度上会降低性能，因为多个线程将会在 CPU 和内存等资源上发生竞争。

```
        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();
        }
    }
```

另一种形式的栅栏是 Exchanger，它是一种两方（Two-Party）栅栏，各方在栅栏位置上交换数据 [CPJ 3.4.3]。当两方执行不对称的操作时，Exchanger 会非常有用，例如当一个线程向缓冲区写入数据，而另一个线程从缓冲区中读取数据。这些线程可以使用 Exchanger 来汇合，并将满的缓冲区与空的缓冲区交换。当两个线程通过 Exchanger 交换对象时，这种交换就把这两个对象安全地发布给另一方。

数据交换的时机取决于应用程序的响应需求。最简单的方案是，当缓冲区被填满时，由填充任务进行交换，当缓冲区为空时，由清空任务进行交换。这样会把需要交换的次数降至最低，但如果新数据的到达率不可预测，那么一些数据的处理过程就将延迟。另一个方法是，不仅当缓冲被填满时进行交换，并且当缓冲被填充到一定程度并保持一定时间后，也进行交换。

5.6　构建高效且可伸缩的结果缓存

几乎所有的服务器应用程序都会使用某种形式的缓存。重用之前的计算结果能降低延迟，提高吞吐量，但却需要消耗更多的内存。

像许多“重复发明的轮子”一样，缓存看上去都非常简单。然而，简单的缓存可能会将性能瓶颈转变成可伸缩性瓶颈，即使缓存是用于提升单线程的性能。本节我们将开发一个高效且可伸缩的缓存，用于改进一个高计算开销的函数。我们首先从简单的 HashMap 开始，然后分析它的并发性缺陷，并讨论如何修复它们。

在程序清单 5-16 的 Computable<A，V> 接口中声明了一个函数 Computable，其输入类型为 A，输出类型为 V。在 ExpensiveFunction 中实现的 Computable，需要很长的时间来计算结果，我们将创建一个 Computable 包装器，帮助记住之前的计算结果，并将缓存过程封装起来。（这项技术被称为“记忆 [Memoization]”。）

程序清单 5-16　使用 HashMap 和同步机制来初始化缓存

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

public class ExpensiveFunction
        implements Computable<String, BigInteger> {
    public BigInteger compute(String arg) {
        // 在经过长时间的计算后
        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;
    }
}
```

在程序清单 5-16 中的 Memoizer1 给出了第一种尝试：使用 HashMap 来保存之前计算的结果。compute 方法将首先检查需要的结果是否已经在缓存中，如果存在则返回之前计算的值。否则，将把计算结果缓存在 HashMap 中，然后再返回。

HashMap 不是线程安全的，因此要确保两个线程不会同时访问 HashMap，Memoizer1 采用了一种保守的方法，即对整个 compute 方法进行同步。这种方法能确保线程安全性，但会带来一个明显的可伸缩性问题：每次只有一个线程能够执行 compute。如果另一个线程正在计算结果，那么其他调用 compute 的线程可能被阻塞很长时间。如果有多个线程在排队等待还未计算出的结果，那么 compute 方法的计算时间可能比没有“记忆”操作的计算时间更长。在图 5-2 中给出了当多个线程使用这种方法中的“记忆”操作时发生的情况，这显然不是我们希望通过缓存获得的性能提升结果。