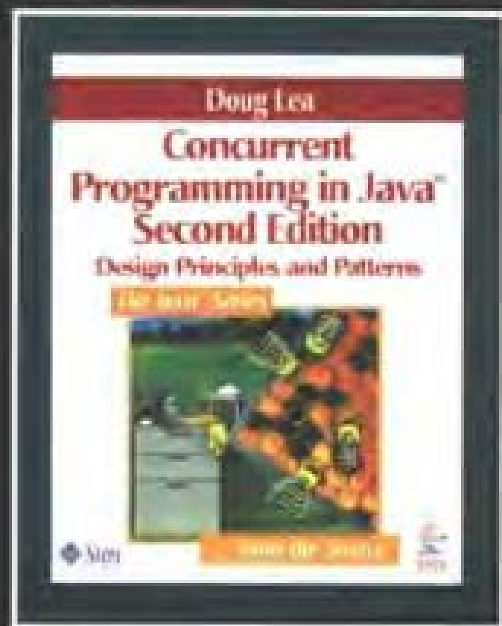
开发大师系列

Concurrent Programming in Java
Design Principles and Patterns Second Edition

Java并发编程
设计原则与模式
（第二版）

[美] Doug Lea 著
赵涌 齐科科 郑承豫 郭明亮 译



Sun公司Java核心技术图书之一 ■
适合希望掌握Java并发编程的中高级程序员 ■
涵盖了有关并发软件开发诸多方面的模式 ■

中国电力出版社
www.infopower.com.cn

Concurrent Programming in Java
Design Principles and Patterns Second Edition

Java并发编程
设计原则与模式（第二版）

本书全面介绍了如何使用Java 2平台进行并发编程，较上一版新增和扩展的内容包括：

- 存储模型
- 取消
- 可移植的并行编程
- 实现并发控制的工具类

Java平台提供了一套广泛而功能强大的API，工具和技术。内建支持线程是它的一个强大的功能。这一功能为使用Java编程语言的程序员提供了并发编程这一诱人但同时也非常具有挑战性的选择。

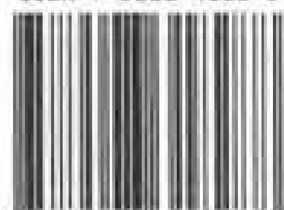
本书通过帮助读者理解有关并发编程的模式及其利弊，向读者展示了如何更精确地使用Java平台的线程模型。

这里，读者将通过使用java.lang.Thread类、synchronized和volatile关键字，以及wait、notify和notifyAll方法，学习如何初始化、控制和协调并发操作。此外，本书还提供了有关并发编程的全方位的详细内容，例如限制和同步、死锁和冲突、依赖于状态的操作控制、异步消息传递和控制流、协作交互，以及如何创建基于Web的服务和计算型服务。

本书的读者对象是那些希望掌握并发编程的中高级程序员。从设计模式的角度，本书提供了标准的设计技巧，以创建和实现用来解决一般性并发编程问题的组件。贯串全书的大量示例代码详细地阐述了在讨论中所涉及到的并发编程理念的细微之处。

Doug Lea 是面向对象技术和软件复用的前沿专家之一。他和Sun实验室开展合作研究长达五年之久。Lea是SUNY Oswego大学计算机科学系的教授。他是计算机应用纽约先进技术中心（New York Center for Advanced Technology in Computer Application）的软件工程实验室主任，也是Syracuse电气与计算机工程系的副教授。此外，他还是《Object-Oriented System Development》（Addison-Wesley, 1993）一书的作者之一。他在New Hampshire大学获得了学士、硕士和博士学位。

责任编辑 / 朱恩从
封面设计 / 王红柳



ISBN 7-5083-1828-5
定价：35.00元

开 发 大 师 系 列

Concurrent Programming in Java
Design Principles and Patterns Second Edition

Java并发编程
设计原则与模式
（第二版）

[美] Doug Lea 著
赵涌 齐科科 郑承豫 郭明亮 译

中国电力出版社

Concurrent Programming in Java : Design Principles and Patterns Second Edition (ISBN 0-201-31009-0)
Doug lea
Authorized translation from the English language edition, entitled Concurrent Programming in Java : Design Principles and Patterns Second Edition, published by Addison Wesley Longman, Inc.,
Copyright©2000
All rights reserved.
No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from the Publisher.
CHINESE SIMPLIFIED language edition published by China Electric Power Press Copyright©2003

本书由美国培生集团授权出版。
北京市版权局著作权合同登记号 图字：01-2002-4844 号

图书在版编目（CIP）数据
Java 并发编程：设计原则与模式 /（美）利著；赵涌等译．—2 版．—北京：中国电力出版社，2003
ISBN 7-5083-1828-5
Ⅰ.J... Ⅱ.①利...②赵... Ⅲ.JAVA 语言—程序设计 Ⅳ.TP312
中国版本图书馆 CIP 数据核字（2003）第 108178 号

书　　名：Java并发编程：设计原则与模式
编　　著：（美）道格•利
翻　　译：赵涌 等
责任编辑：朱恩从
出版发行：中国电力出版社
　　　　　地址：北京市三里河路6号　　邮政编码：100044
　　　　　电话：（010）88515918　　传　　真：（010）88518169
印　　刷：汇鑫印务有限公司
开　　本：787×1092　1/16　　印　　张：18　　字　　数：417千字
书　　号：ISBN 7-5083-1828-5
版　　次：2004 年 2 月北京第 1 版　　2004 年 2 月第 1 次印刷
定　　价：35.00 元
版权所有　翻印必究

译 者 序

自 20 世纪 90 年代以来，并发编程一直是软件设计的热门话题。随着大型企业软件的演化，以及 Java 与.net 的斗法，并发编程的概念正被不断地强化，其涵盖的内容也不断扩大，在所有有关软件和编程的杂志和论文中，它几乎独占鳌头。

由于 Java 语言在设计之初就考虑到了并发编程的因素，所以在和 C++的竞争中，它占了很大的优势。虽然 J2EE 规范了一些并发编程的底层设计，使得开发人员可以更专注于业务逻辑的实现，但作为大型软件的开发人员或架构师，对并发编程的理解和运用仍是必需的，而且可以说是永无止境的。

作为 Java 1.5 并发 JSR 的带头人，Doug Lea 编写的这本《Java 并发编程：设计原则与模式》可以称为这方面的经典之作。随着 util.concurrent 包的广泛商用化，以及加入 Java 1.5 标准库的临近，潜在的读者数量正在倍增。虽然国外读者对这本书的深度和广度赞誉甚高，但由于 Doug Lea 的严密论述而导致的语言相对深奥的问题也正日渐显露。因此，为了大家能更好地理解本书所述内容，我们在翻译的时候尽量使用意译，将英文中复杂的句法尽量简化。

翻译组的成员包括赵涌、齐科科、任文捷、关承豫和郭明亮等。具体负责的工作和个人介绍如下：

- 赵涌，负责总体审稿和校对。具有多年的编程经验，从 1998 年开始由 C/C++编程转为 Java 编程，一直在电信领域编写和设计分布式应用软件。现在对 Java 服务器端的模式和架构非常感兴趣。
- 齐科科，负责第 1 章的翻译工作。是开源软件爱好者，blogger。兴趣方向：并行与分布式计算。
- 任文捷，负责第 2 章的翻译工作。2001 年毕业于哈尔滨工业大学计算机系统结构研究室，硕士学历，主要研究方向为 Linux 操作系统健壮性分析。现在就职于联想研究院，主要研究方向是语音识别、自然语言理解和人机交互界面等。
- 关承豫，负责第 3 章的翻译工作。外资 Java 软件研发中心高级程序员，开源软件的爱好者。研究方向：面向对象设计、设计模式、企业级软件设计和 GUI 相关技术。
- 郭明亮，负责第 4 章的翻译工作。多年 Java 开发经验，具有丰富的实际项目经验，参与过多个大型 Java 项目的开发与设计。目前兴趣：分布式计算。

本书由 JavaResearch.org 组织翻译和总体协调，我们还在 JavaResearch.org 的论坛中专门开辟了“译作支持区”（http://www.javaresearch.org/forum/forum.jsp?column=481），大家在阅读本书过程中所遇到的任何问题或迸发的任何感想都可以在那里发表，直接和译者及更多本书的读者进行交流。有关本书的一些勘误，我们将在论坛中及时发布和更新。由于水平有限，书中可能还存在着错误和不足，敬请读者朋友批评指正。

Java 研究组织《Java 并发编程：设计原则与模式》翻译组

2003 年 10 月

java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

致　　谢

这本书开始于我在1995年春天编写的一小套Web页面。那时，作为实验性的开发阶段，我试图把自己早期使用Java并发特性的尝试变得更有实际意义。在演变过程中，我首先在World Wide Web上扩展、扩充以及删除了一些反映我和其他人有关Java并发编程的不断积累的经验模式，一直到现在的成书，书中涵盖了有关并发软件开发的诸多方面的模式。至今，那些网页还在，不过现在它只作为书中所阐述概念的补充材料。

在从页面到成书的过程中，许多有识之士给予了评注、建议、勘错报告以及意见交流。这些人包括：Ole Agesen、Tatsuya Aoyagi、Taranov Alexander、Moti Ben-Ari、Peter Buhr、Bruce Chapman、Il-Hyung Cho、Colin Cooper、Kelly Davis、Bruce Eckel、Yacov Eckel、Saleh Elmohamed、Ed Falis、Randy Farmer、Glenn Goldstein、David Hanson、Jyrki Heikkinen、Alain Hsiung、Jerry James、Johannes Johannsen、Istvan Kiss、Ross Knippel、Bil Lewis、Sheng Liang、Jonathan Locke、Steve MacDonald、Hidehiko Masuhara、Arnulf Mester、Mike Mills、Trevor Morris、Bill Pugh、Andrew Purshottam、Simon Roberts、John Rose、Rodney Ryan、Joel Rosi-Schwartz、Miles Sabin、Aamod Sane、Beverly Sanders、Doug Schmidt、Kevin Shank、Yukari Shirota、David Spitz、David Stoutamire、Henry Story、Sumana Srinivasan、Satish Subramanian、Jeff Swartz、Patrick Thompson、Volker Turau、Dennis Ulrich、Cees Vissar、Bruce Wallace、Greg Wilson、Grant Woodside、Steve Yen、Dave Yost，以及其他通过匿名电子邮件发送评注的人们。

Ralph Johnson模式研讨会的成员（尤其是Brian Foote和Ian Chai）通读了一些模式的早期形式，并提出了许多改进意见。纽约城模式组（New York City Patterns Group）的Raj Datta、Sterling Barrett和Philip Eskelin，硅谷模式组（Silicon Valley Patterns Group）的Russ Rufer、Ming Kwok、Mustafa Ozgen、Edward Anderson和Don Chin对第二版的早期版本也做出了宝贵贡献。

在时间很紧张的情况下，第一版和第二版书稿的正式和非正式的审阅人也都提出了有帮助的意见和建议。他们是Ken Arnold、Josh Bloch、Joseph Bowbeer、Patrick Chan、Gary Craig、Desmond D'Souza、Bill Foote、Tim Harrison、David Henderson、Tim Lindholm、Tom May、Oscar Nierstrasz、James Robins、Greg Travis、Mark Wales、Peter Welch和Deborra Zukowski。需要特别感谢Tom Cargill，他提出了很多见解和修正，而且容许在本书中阐述其特有的通知模式（Specific Notification pattern）。此外，还要特别感谢David Holmes，除了其他贡献，他还帮助开发和扩展了在第二版中所使用的教程的素材。

没有Sun实验室的慷慨支持，就不会有本书。我在这里要特别感谢Jos Marlowe和Steve Heller，他们为我提供了在有趣而激动人心的研究开发项目中协同工作的机会。

最后，还要感谢Kathy、Keith和Colin为这一切所付出的忍耐。

Doug Lea，1999年9月

java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

目　　录

译者序
致　谢

第 1 章　面向对象的并发编程 …… 1
1.1　使用并发构件 …… 4
1.2　对象和并发 …… 15
1.3　设计因素 …… 28
1.4　Before/After 模式 …… 42

第 2 章　独占 …… 51
2.1　不变性 …… 52
2.2　同步 …… 55
2.3　限制 …… 73
2.4　构造和重构类 …… 86
2.5　使用锁工具 …… 108

第 3 章　状态依赖 …… 118
3.1　处理失败 …… 119
3.2　受保护方法 …… 132
3.3　类的构建与重构 …… 148
3.4　使用并发控制工具类 …… 162
3.5　协同操作 …… 176
3.6　事务处理 …… 184
3.7　工具类的实现 …… 195

第 4 章　创建线程 …… 208
4.1　单向消息 …… 210
4.2　编写单向消息 …… 226
4.3　线程中的服务（Services in Thread） …… 241
4.4　并行分解（Parallel Decomposition） …… 255
4.5　活动对象 …… 274

java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

第 1 章

面向对象的并发编程

本书讨论了一些可以在Java™编程语言中使用的思考、设计和实现并发程序的方法。本书讨论的大多数内容都假定你是一个有经验的开发者，熟悉面向对象（object-oriented，OO）编程，但是对于并发编程的知识了解得不多。当然，对于有着相反经历的读者，即熟悉其他语言的并发编程，但是对 Java 语言中的相关部分知之甚少的读者，也可以从此书中得到不少帮助。

本书粗分为四章（也许称为部分更为合适）。第 1 章，我们从对一些常用构件的简要介绍开始，然后为并发的面向对象的编程建立一个概念基础：并发性是如何与对象结合起来的，设计需求的结果是如何影响对象和组件的构造的，如何使用一些常用的设计模式来构建解决方案。

接下来的三章主要是围绕着如何使用（和避免）Java 编程语言所提供的三种常用的并发构件进行阐述的：

独占（Exclusion）。可以通过阻止多个并发行为间的有害干扰来维护对象状态的一致性。通常使用同步（`synchronized`）的方法。

状态依赖（State dependence）。是否可以触发、阻止、延迟或是恢复某些行为是由一些对象是否处在这些行为可能成功或是已经成功的状态上决定的。通常，状态依赖关系使用***监视器（monitor）***方法实现，如 `Object.wait`、`Object.notify` 和 `Object.notifyAll`。

创建线程（Creating threads）。使用线程（`Thread`）对象来创建和管理并发操作。

每章都包含一系列主要的节，而每一节又都有各自独立的主题。它们分别讲述了高级的设计准则和策略，并发构件的技术细节，封装了常用方法的工具，以及用来解决特定并发问题的设计模式等一系列问题。大多数节的结尾都附带了一份有关进阶阅读的文章列表，它们包括了所讨论主题的更多信息。本书的线上支持包含了一些在线资料的链接，还有本书的更新、勘误和代码示例。可以通过如下网址访问：

`http://java.sun.com/Series`

或是

`http://gee.cs.oswego.edu/dl/cpj`

如果你已经熟悉了这些相关的基础知识，则可以按照章节的顺序深入学习每一个主题。虽然大多数读者可能更希望根据各自不同的顺序学习本书，但是，由于多数有关并发的概念和技术互相之间都是紧密关联的，因此很难完全独立地学习和理解某个单独的章节或是主题。然而，你可以使用一种广度优先的方法，在开始对某个部分进行详细深入地学习之前，

java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

先大致浏览一下每一章（包括本章）。在有选择地阅读某些材料之后，本书后面的很多内容都可以通过与这些材料上附带的交叉引用访问到。

你可以现在就开始尝试着使用这一技巧，跳过如下的预备知识。

术语（Terminology）。本书使用标准面向对象的术语约定：程序定义了***方法***（所实现的***操作***）和***成员变量***（所表示的***属性***），这些变量保存了特定***类***的所有***实例***（对象）。

面向对象程序中的交互通常与程序各个部分的职责相关。一个***客户（client）***对象通常期待某个动作的执行，而一个***服务（server）***对象通常包含着用来执行这个动作的代码。在这里，术语***客户***和***服务***指的是它们的通常含义，而不是特指分布式的客户/服务器结构。一个客户可以是任何对象，它向另一个对象发送请求。而一个服务也可以是任何对象，它接受这个请求。大多数对象都同时在扮演着客户和服务这两种角色。通常情况下，我们不必关心一个对象是客户或者还是服务，或是同时属于这两种类型，它通常被称作一个***主体（host）***，而那些由它调用的对象常常被称作是***辅助对象（helper）***或***对等体（peer）***。同时，当讨论到如同`obj.msg(arg)`形式的调用方式时，接受者（变量`obj`所代表的对象）则被称作***目标（target）***对象。

本书尽力避免讨论那些偶尔出现并且与并发操作没有直接关系的类和包，同时不涉及某些特殊框架下的有关并发控制的技术细节，诸如 Enterprise JavaBeans™ 和 Servlets。但有些时候，它确实使用了一些与 Java 平台相关的商业软件和产品。

代码（Code listing）。本书包含了一系列看上去有些恼人的、短小但可运行的例子，通过不断修改这些例子来说明书中涉及到的大多数技术和模式。这样做的目的是为了可以描述得更清晰，而不是想让本书变得枯燥。如果使用其他的具体例子，这些并发构件可能会由于过于细微而迷失在那些复杂的代码之中。通过重用相同的例子，并突出设计和实现方面的主要问题，可以使得细小但关键的差别变得更加明显。同时，本书的内容还包括了简要的代码和类的片断，描述了有关实现技术，但是它们也许并不完整甚至无法编译，这样的类都会在代码清单一开始的注释部分中被指明。

有时代码中会省略一些import语句、访问限定符，甚至某些方法和成员变量，这是因为，这些内容可以从上下文的关系中推断出来，或是不会对相应的功能产生影响。只要没有特殊的理由去限定子类的访问，`protected`限定符就被用来作为那些非公有元素的默认属性。这保证了并发类设计的可扩展性（参见 § 1.3.4和 § 3.3.3）。默认情况下，类没有访问限定符。有时，示例代码没有使用标准的格式，这样做的目的是为了使它们被安排在一页上或是突出我们感兴趣的某些主要内容。

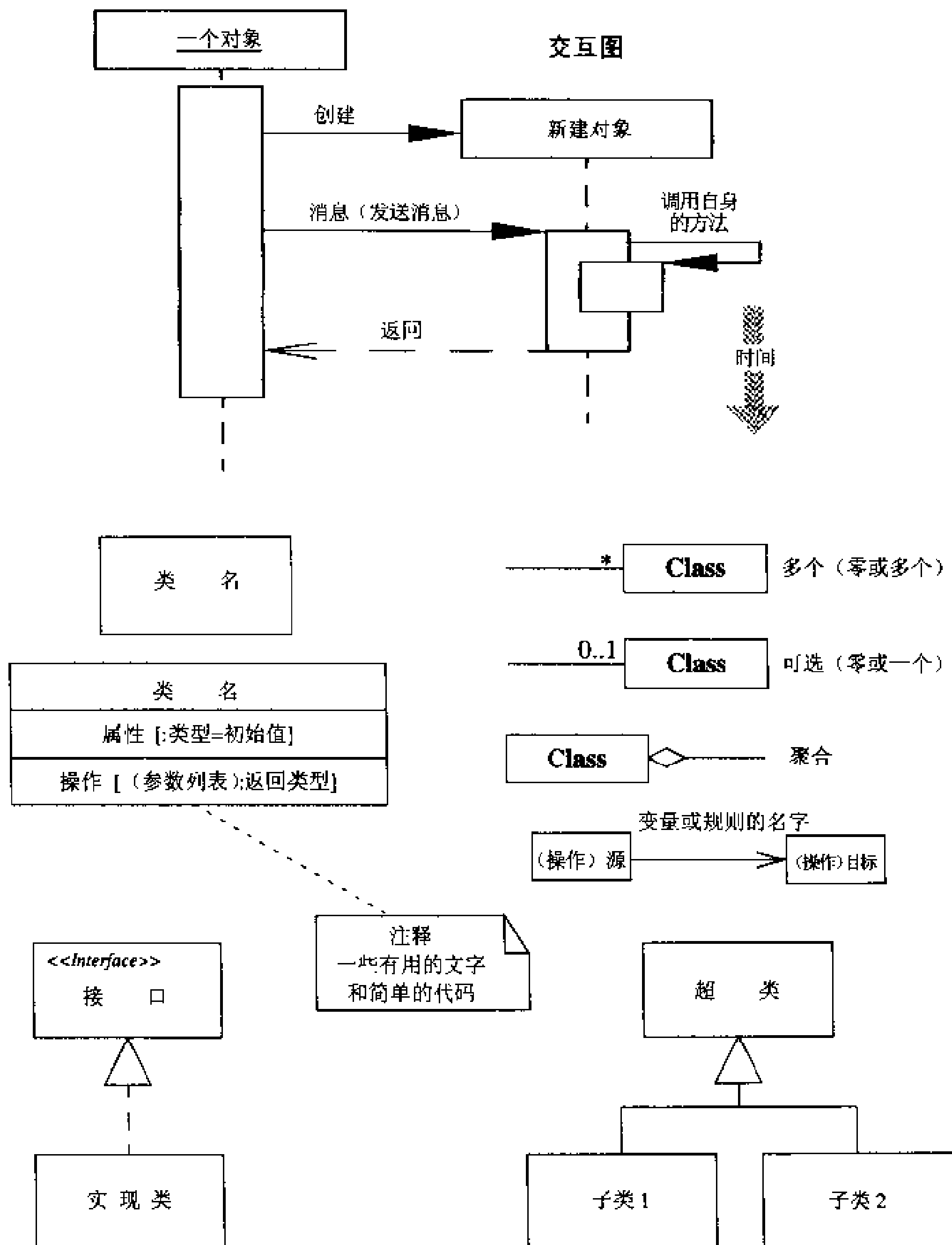
书中的所有例子的代码都可以从网上获得。书中的大多数技术和模式都用一个单独的代码实例加以说明，其中给出了它们的典型形式和用法。线上支持还包括了额外的例子，用于说明其他的一些细微变化，还有一些指向其他用法的链接。你还可以在网上找到一些更长的例子，它们更适合浏览和测试，而不是作为代码阅读。

线上支持还包括了一个链接，指向一个有用的包`util.concurrent`，其中包括一些书中讨论的实用类的产品级版本。这些代码运行在 Java 2 平台上，并且在 1.2.x 版本下进行了测试。线上支持中还包括了一些零散的讨论、说明和脚注，简要地记载了从之前版本到现有版本的修改，还有在写作本书时想到的、会在未来做出的修正，以及在实现时会遇到的一

java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

些奇怪但值得注意的问题。可以从在线支持中得到有关这些内容的最新数据。

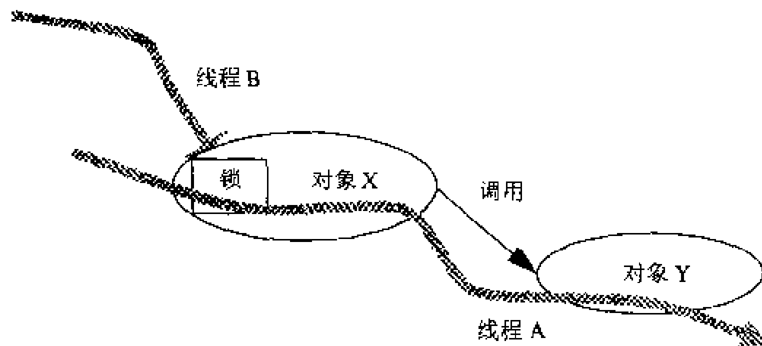
图例。我们使用标准的 UML 表示法来绘制交互图和类图（可以参考 § 1.1.3 的进一步说明）。本页的表示法图（经 Martin Fowler 允许）演示了图例在本书中的样子。除此之外，我们没有使用其他的 UML 表示法、方法和术语。



其他的大多数图例用来描绘线程时序关系（timethread），那些随意绘制的灰色曲线用来跟踪线程调用多个对象的轨迹。扁平的箭头表示阻塞。椭圆形的物体表示对象，有时当中也包含了一些有关内部特性的说明，比如锁、成员变量和一部分代码。在对象中间的那些细的线条（有时标以说明）指明了对象间的关系（通常是引用，或是潜在的调用关系）。下图使用了一个并无实际意义的例子来说明线程 A 已经取得了对象 X 上的锁，因而正在执行对象

java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

Y 上的某些方法，这里的 Y 就是 X 的一个辅助对象。同时，当线程 B 想执行对象 X 的某个方法时将被阻塞。



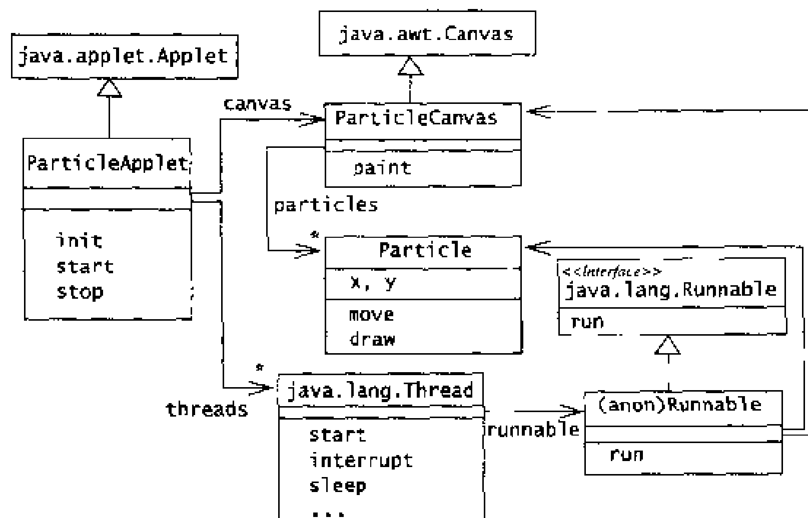
1.1 使用并发构件

本节将通过一个例子介绍几种基础的并发支持构件，然后大致地介绍 Thread 类的一些基本方法。其他的并发构件也会在引入它们的时候加以简要说明，完整的技术细节会在后面的章节中详细阐述（主要是 § 2.2.1 和 § 3.2.2）。通常情况下，并发程序也会使用一些 Java 编程语言所独有的特性，因此我们在使用的时候会对它们进行简要地回顾。

1.1.1 粒子 Applet

ParticleApplet 是一个用来显示随机移动粒子的 Applet。除了介绍相关的并发构件之外，这个例子也举例说明了一些在 GUI 程序中使用线程将遇到的问题。为了使得这个例子看上去更有吸引力、更真实，也许还需要进行一些润色工作。作为一个作业，你会在修改与添加的工作中得到乐趣的。

作为一个典型的GUI程序，ParticleApplet 使用多个辅助类来完成其中主要的工作。在讨论 ParticleApplet 之前，我们先大致介绍一下 Particle 和 ParticleCanvas 的结构。



java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

1.1.1.1 粒子

类 Particle 定义了一个用于移动物体的完全假想的模型。每个粒子（particle）用它的（x,y）位置表示。同时，每个粒子包括了一个用来随机移动它自身位置的方法和一个可以通过给定的 java.awt.Graphics 对象来绘制其自身（一个小方块）的方法。

虽然 Particle 对象内部并不包含任何并发操作，但是它的方法可能被多个并发行为调用。当一个操作正在执行一个 move 操作而几乎同时另一个操作正在调用 draw 方法时，我们就需要保证 draw 方法所描绘的是 Particle 所在的准确位置。这里，我们要求 draw 方法可以在 move 方法被调用之前或是之后使用粒子的当前位置值进行绘制。举例而言，当 draw 操作使用 move 方法调用前的 y 值和 move 方法调用后的 x 值绘制一个粒子的图形，就会产生一个概念性的错误。如果我们允许这种情况出现，那 draw 方法就可能把一个粒子绘制在其并没有出现过的位置上。

我们可以通过使用 synchronized 关键字来防止这种情况的发生，synchronized 可以修饰一个方法或是一个代码块。Object 类（和它的子类）的每一个实例都会在进入一个同步（synchronized）方法前加锁，并在离开这个方法时自动地释放这把锁。对于代码块的操作也是一样，只是同步代码块需要一个参数来指明对哪一个对象加锁。最常用的参数是 this，它意味着锁住当前正在执行的方法所属的对象。当一个线程持有了一把锁后，其他线程必须阻塞，等待这个线程释放这把锁。加锁对于非同步的方法不起作用，因此即便另一个线程持有这个锁，这个非同步方法也可以执行。

通过保证**同步**在同一个对象上的方法或代码块操作的***原子性（atomicity）***，加锁机制可以同时提供多种保护措施，包括对上层和底层冲突的保护。原子操作被作为一个整体来执行，这样它们就不会被插入的其他线程的操作打断。但是，就像将在 § 1.3.2 和第 2 章所提到的那样，过多的锁操作也会产生线程活跃性问题，最终导致程序停止。我们在这里并不过多地讨论此类问题，暂时使用一些简单的规则来排除这些干扰。

- 永远只是在更新对象的成员变量时加锁。
- 永远只是在访问有可能被更新对象的成员变量时才加锁。
- 永远不要在调用其他对象的方法时加锁。

在使用这些规则时有许多例外和细节需要注意，但是对于我们的 Particle 类而言，这些暂时已经足够了。

```
import java.util.Random;

class Particle {
  protected int x;
  protected int y;
  protected final Random rng = new Random();

  public Particle(int initialX, int initialY) {
    x = initialX;
    y = initialY;
  }
```


java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

```
    public synchronized void move() {
      x += rng.nextInt(10) - 5;
      y += rng.nextInt(20) - 10;
    }

    public void draw(Graphics g) {
      int lx, ly;
      synchronized (this) { lx = x; ly = y; }
      g.drawRect(lx, ly, 10, 10);
    }
  }
```

注意：

- 我们使用 final 关键字修饰随机数生成器 rng，这样做的目的是为了确保这个成员变量的引用不会被改变。这样一来，它就不会被我们的加锁规则所影响。许多并发程序中都大量地使用了 final 关键字，以帮助将减少同步需要的设计意图自动在文档中突出说明（参见 § 2.1）。
- draw 方法需要同时得到一对一致的 x 和 y 值。由于一个方法一次只能返回一个值，但是我们在这里却需要同时获得 x 和 y 的值，因此不能简单地把变量的访问封装在一个同步方法中。相反，我们使用一个同步代码块代替（可以查阅 § 2.4 来寻找其他的方法）。
- 可以看到 draw 方法遵守了我们上面提到的那个重要规则，在调用其他对象上的方法时释放了锁（这个方法是 g.drawRect）。而 move 方法看似违背了我们的规则，它调用了 rng.nextInt 方法。然而在这里，我们这样做是有它的原因的，因为每一个 Particle 都包含其自身的 rng 对象，因而 rng 对象是 Particle 的一部分，所以它应该不能被看成规则中所描述的"其他对象"。§ 2.3 描述了适用于这种情况下的更通用的条件，并且讨论了在这种情况下许多需要考虑的因素。

1.1.1.2　ParticleCanvas

ParticleCanvas 是 java.awt.Canvas 的一个简单子类，它为所有粒子提供了一个绘制的区域。它的主要职责是，当其 paint 方法被调用时，调用所有粒子的 draw 方法。

但是，ParticleCanvas 本身并不负责创建和管理这些粒子。ParticleCanvas 对象可以被动或是主动地得到这些粒子对象。这里，我们选择前一种方法。

实例变量 particles 是一个数组，保存了已存在的 Particle 对象。这个变量在需要的时候由 Applet 负责设置，被 paint 方法使用。我们同样应用了默认的规则，使用了两个简单的、同步的 get 和 set 方法 [也可以称为***访问方法***（accessor）和***指派*** 方法（assignment）] 来访问 particles 变量，而避免了直接访问 particles 变量本身。为了简化程序并使其被正确地使用，这个 particles 变量不允许被设为 null，而是被初始化为一个空数组。

java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

```
class ParticleCanvas extends Canvas {

  private Particle[] particles = new Particle[0];

  ParticleCanvas(int size) {
    setSize(new Dimension(size, size));
  }

  // intended to be called by applet
  protected synchronized void setParticles(Particle[] ps) {
    if (ps == null)
      throw new IllegalArgumentException("Cannot set null");

    particles = ps;
  }

  protected synchronized Particle[] getParticles() {
    return particles;
  }

  public void paint(Graphics g) { // override Canvas.paint
    Particle[] ps = getParticles();

    for (int i = 0; i < ps.length; ++i)
      ps[i].draw(g);

  }
}
```

1.1.1.3 ParticleApplet

类 Particle 和 ParticleCanvas 可以作为几个其他的程序的基础使用。但是对于 ParticleApplet 而言我们所要做的是：使每一个粒子以一种自治的"持续"的方式运动，并且更新显示这些粒子的位置。为了遵循标准 Applet 的编程约定，当外部程序调用（通常在 Web 浏览器内部被调用）Applet.start 方法之后，这些行为才会开始，直到 Applet.stop 被调用后才会结束（我们也可以通过添加按钮来允许用户来控制粒子动画的开始和结束）。

有多种方法可以实现这一切，最简单的一种是为每个粒子设置一个单独的循环，并且每一个循环都在一个单独的线程中执行。

在新线程中执行的操作必须被定义在实现了 java.lang.Runable 接口的类中。这个接口只包含了一个 run 方法。它没有参数，没有返回值，也不会抛出需要检查的异常：

```
public interface java.lang.Runnable {
  void run();
}
```

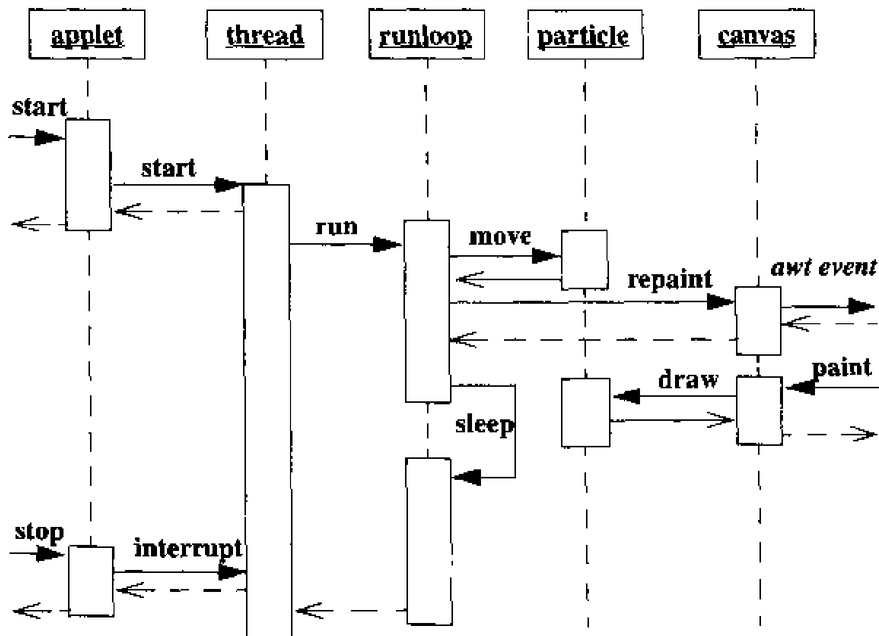
接口(interface)封装了一系列相一致的服务和属性［广义上，它们可以被称为一个***角色*** ***(role)***］，但在接口中并没有特别指明这些功能应该由哪个对象或是代码来实现。接口比类更为抽象，因为它没有包含任何有关实现的信息和代码。它们所做的只是定义一些公共操作的***签名*** ***(signature)***（包括名字、参数、结果类型和异常），没有对调用这些接口的对象

java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

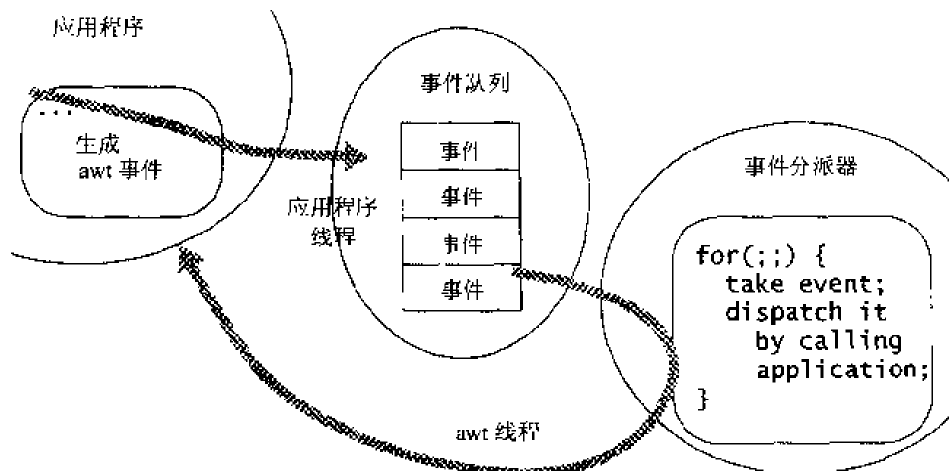
的类进行约束。除了包含一个 run 方法之外，实现 Runnable 接口的类和普通的类并无差别。

每一个 Thread 类的实例维护了执行和管理那些组成其动作的调用序列所需要的控制状态。Thread 类的最常用的构造函数接收一个 Runnable 对象作为它的参数，在线程被启动时会调用这个 Runnable 对象的 run 方法。由于任何类都可以实现 Runnable 接口，因此通常一种很方便且实用的做法就是把一个 Runnable 作为一个匿名内部类实现。

类 ParticleApplet 利用这种方法实现线程，从而使得粒子移动，并且在 Applet 结束时取消这些粒子的运动。我们可以通过覆盖标准 Applet 的 start 和 stop 方法来达到这一目的（虽然这两个方法与 Thread.start 和 Thread.stop 重名，但是两者并无任何关联）。



上面的交互图（interaction diagram）说明了 Applet 执行期间的主要消息序列。除了这些显式创建的线程之外，这个 Applet 还同 AWT 事件线程交互，这一点会在 §4.1.4 做更多的描述。交互图右边省略的部分是一个生产者-消费者模型，它大致采用如下的形式：



java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

```
public class ParticleApplet extends Applet {

  protected Thread[] threads = null; // null when not running

  protected final ParticleCanvas canvas
                                    = new ParticleCanvas(100);

  public void init() { add(canvas); }

  protected Thread makeThread(final Particle p) { // utility
    Runnable runloop = new Runnable() {
      public void run() {
        try {
          for(;;) {
            p.move();
            canvas.repaint();
            Thread.sleep(100); // 100msec is arbitrary
          }
        }
        catch (InterruptedException e) { return; }
      }
    };
    return new Thread(runloop);
  }

  public synchronized void start() {
    int n = 10; // just for demo

    if (threads == null) { // bypass if already started
      Particle[] particles = new Particle[n];
      for (int i = 0; i < n; ++i)
        particles[i] = new Particle(50, 50);
      canvas.setParticles(particles);

      threads = new Thread[n];
      for (int i = 0; i < n; ++i) {
        threads[i] = makeThread(particles[i]);
        threads[i].start();
      }
    }
  }

  public synchronized void stop() {
    if (threads != null) { // bypass if already stopped
      for (int i = 0; i < threads.length; ++i)
        threads[i].interrupt();
      threads = null;
    }
  }
}
```

注意：

- 我们在 makeThread 的操作中定义了一个“无限”循环（可能有人更喜欢使用“while(true)”语句），只有当前线程被中断时这个循环才会退出。在每一次

java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

循环中包含如下操作：粒子移动和告知画布进行重绘，这样这些粒子的移动才会被显示出来，在这之后的一段时间内什么都不做，从而降低工作的速度，以便配合人类的视觉感知速率。Thread.sleep 会使得当前的线程暂停，而之后它被系统时钟唤醒。

- 内部类之所以方便和实用的一个原因是：它们可以直接获取所有适当的上下文变量（在这里是 p 和 canvas）而不需要创建一个额外的类来保存它们。不过，在拥有这些便利的同时，内部类也带来一些小小的副作用：这些可以被内部类直接获得的方法参数和本地变量都必须被声明为 final，这样做的目的是为了保证这些变量的值都可以被无歧议地获得。否则，如果在 makeThread 方法内部生成了 Runnable 对象之后 p 被重新赋值，那么当 Runnable 执行的时候便会无法确定是应该使用 p 原先的值还是新赋的值。
- canvas.repaint 方法并不会直接调用 canvas.paint 方法，而是在 AWT 事件队列（java.awt.EventQueue）中添加了一个 UpdateEvent 对象。（在系统内部，这个操作会被优化，以便可以消除重复的事件。）java.awt.EventDispatchThread 的实例会异步地从队列中取出这些事件，然后分派给指定的接受者，（最终）调用 canvas.paint 方法。这个线程连同其他系统生成的线程也会存在于某些“名义上”的单线程程序中。
- 直到 Thread.start 方法被调用后，生成线程所包含的操作才会开始工作。
- 在§3.1.2 中我们会讨论到，有很多方法可以让一个线程的工作停止。最简单的莫过于让 run 方法自然终止。但是对于无限循环的方法而言，最好的选择是使用 Thread.interrupt 方法。（通过 InterruptedException 异常）一个被中断的线程会自动从下列方法中退出：Object.wait、Thread.join 和 Thread.sleep。调用者可以通过捕获这个异常并采取适当的措施来关闭线程。在这里，runloop 中的 catch 语句只是使得 run 方法退出，从而使得这个线程结束。
- 这里的 start 和 stop 方法都声明为 synchronized，从而可以避免并发的开始或停止操作。即便这些方法需要处理大量工作（包括调用其他的对象），这里的加锁机制也可以很好地完成开始到结束和结束再开始的状态转换工作。变量 threads 的空值（null）可以方便地用来表示当前的处理状态。

1.1.2 线程机制

线程是一个独立于其他线程执行的调用序列，它可以共享底层的系统资源，如文件，并且可以访问在同一个程序中构造的其他对象（参见§1.2.2）。java.lang.Thread 对象用来记录和控制这种行为。

每个程序至少包括一个线程——用来运行在 JVM（Java Virtual Machine，Java 虚拟机）启动时作为参数给出的类中的 main 方法。在初始化 JVM 的时候，其他的内部后台线程也会被启动。在不同的 JVM 实现中，这些线程的数目和特性并不相同。然而，所有的用户线

java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

程都需要被显式地构造，并且从这个主线程或是其他依次创建的线程中启动。

在这里，我们总结了有关 Thread 类的一些主要的方法和属性，并包括一些使用上的要点。在本节剩下的内容中，它们会被进一步地讨论和说明。你可以通过阅读《The Java™ Language Specification（"JLS"）》和公开的 API 文档来得到更为详细的信息和权威的解释。

1.1.2.1 构造方法（construction）

不同的线程构造方法接受下面所介绍对象中的一个或是多个组合作为参数：

- Runnable 对象，这样 Thread.start 方法就会调用这个 Runnable 对象的 run 方法。如果没有传递 Runnable 参数，Thread.run 的默认实现就会立即返回。
- String 对象，可以用来作为这个 Thread 对象的标识。这个参数只是用于跟踪和调试，别无他用。
- ThreadGroup 对象，用于放置新 Thread。如果该 ThreadGroup 不允许被访问，则会抛出一个 SecurityException 异常。

Thread 类本身也实现了 Runnable 接口。因此，除了构造 Runnable 对象并把它作为构造函数的参数传递给 Thread 类之外，你也可以生成 Thread 类的一个子类，通过覆盖这个 run 方法来执行相应的操作。不过，通常最好的策略是把 Runnable 接口当作一个单独的类来实现，并把它作为参数传递给一个 Thread 的构造函数。通过将代码隔离在单独的类中可以使你不必担心 Runnable 类中使用的同步方法和同步块与在相应线程类中所使用的其他任何方法之间的潜在操作所带来的影响。更一般地说，这种分离允许独立控制相关的操作和运行这些操作的上下文：同一个 Runnable 对象既可以传递给多个使用不同方式初始化的 Thread 对象，也可以传递给其他的轻量级执行者（executor）（见 § 4.1.4）。同样需要注意的是，继承了 Thread 类的对象不能再同时继承其他类了。

Thread 对象还拥有一个后台线程的（daemon）状态属性，这个属性不能通过 Thread 的构造函数来设置，只能在一个 Thread 实例被启动之前设置。如果 setDaemon 方法设置为真，则表明：当所有的非后台线程都终止后，JVM 就会退出，并且会立即终止这个线程。方法 isDaemon 用来返回相应的状态。后台线程的用处非常有限，因为它也需要在程序退出时做一些清除工作（daemon 通常读做"day-mon"，是系统编程传统的遗物。系统后台进程是持续的进程，比如打印队列管理程序，会"一直"存在于系统之中）。

1.1.2.2 启动线程

调用 Thread 类的 start 方法使得 Thread 类的一个实例将其 run 方法作为一个单独的操作启动。任何一个调用线程所拥有的同步锁都不会被这个新线程持有（见 § 2.2.1）。

不论是由于正常返回还是由于抛出了一个未经检查的异常（比如，RuntimeException、Error 或它们的子类）导致一个线程的 run 方法结束，这个线程都会终止。一旦线程被终止，它们便不能被重新启动。重复调用 start 方法，会得到一个 InvalidThreadState-Exception 异常。

java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

如果线程被启动并且没有终止，调用方法 isAlive 将返回 true。如果线程仅仅是因为某个原因阻塞，该方法也会返回 true。在由于线程被取消而 isAlive 方法返回 false 这个精确的执行点上，已知的 JVM 的实现并不相同（参见 § 3.1.2）。没有方法可以告诉你一个非 isAlive 的线程是否已经被启动了。同样，我们也无法很容易地获知一个线程究竟是被哪个线程启动的，尽管可以从它的 ThreadGroup（见 § 1.1.2.6）对象中得到处在同一线程组中的其他线程的标识。

1.1.2.3　优先级

为了能够在不同的硬件平台和操作系统上实现 JVM，Java 编程语言并不保证线程会被平等调度或是对待，甚至不严格保证线程的执行（参见 § 3.4.1.5）。但是线程的确支持优先级方法，这些方法可以影响线程的调度工作。

- 每一个线程都有一个优先级，它的范围在 Thread.MIN_PRIORITY 和 Thread.MAX_PRIORITY 之间（分别用 1 和 10 表示）。
- 默认情况下，每一个新线程的优先级大小与创建它的那个线程的优先级一致，与 main 方法相关的初始线程使用一个默认的优先级 Thread.NORM_PRIORITY(5)。
- 任何线程当前的优先级可以通过方法 getPriority 获得。
- 任何线程的优先级都可以通过调用 setPriority 方法动态地改变。一个线程所允许的最大优先级由它所处的 ThreadGroup 决定。

当活动（runnable）（参见 § 1.3.2）的线程数目多于系统所能够使用的 CPU 数目时，调度程序通常会偏向运行那些有着更高优先级的线程。具体的调度机制可能并且的确会由于平台的不同而不同。比如，某些 JVM 实现通常从当前的线程中选择优先级最高的线程运行（同等级别时则任意选择），而另一些 JVM 实现则将 10 个线程优先级别映射到略小的、系统所支持的级别中，因此不同优先级的线程可能被同样处理。还有一些系统，它们将优先级和时间机制或是其他调度策略混合考虑，这样可以确保即便拥有低优先级的线程也有机会运行。同样，改变优先级也是可以的，但不一定会像影响运行在同一个计算机系统中的其他程序一样，影响线程的调度。

优先级同线程的语义或正确性没有其他的联系（参见 § 1.3）。事实上，不能通过处理线程的优先级来替代锁的功能。优先级只能用来表示不同线程间的相对重要性或是紧急程度。当大量线程在争夺一个运行机会时，才可能考虑这些优先级的值。比如，使 ParticleApplet 例子中的粒子动画线程的优先级小于构造它们的 applet 线程的优先级，这样，可以提高在某些系统中对鼠标单击的响应，但在其他方面不会影响系统响应。不过，即便 setPriority 在某些系统中无效，程序也应该首先保证可以正确运行（虽然不一定响应及时）（对于 yield 方法也是一样，参见 § 1.1.2.5）。

以下的表格给出了一些通常的约定，用来指导如何针对不同的任务类型分配优先级。在很多并发应用程序中，通常，只有一小部分线程在运行（其他的都因为各种原因而阻塞），因此在这样的情况下不必设置线程的优先级。而在其他情况下，微调优先级可能会对并发系统的最后的性能优化有些许影响。

java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

优先级范围	适用情况
10	关键问题
7~9	交互，事件驱动
4~6	IO 相关
2~3	后台计算
1	在没有其他程序运行的情况下运行

1.1.2.4 控制方法

只有很少的方法可以用来与线程交互：

- 每一个线程都有一个对应的中断（interruption）状态，用布尔型变量表示（参见§3.1.2）。可以通过调用 Thread t 的 t.interrupt 方法来将这个状态设为 true，除非 Thread t 正在处理 Object.wait、Thread.sleep 或是 Thread.join 方法。在这种情况下，interrupt 方法会导致这些方法（在 t 中）抛出一个 InterruptdException，但是 t 的中断状态会被设为 false。
- 任何 Thread 的中断状态都可以通过 isInterrupted 方法获得。如果这个线程被 interrupt 方法中断，则该方法将返回 true。但是无论是通过调用 Thread.interrupted 方法（见§1.1.2.5），还是从 wait、sleep 或 join 方法中抛出 InterruptedException，这个状态值都不会被重置为 false。
- 通过调用线程 t 的 join 方法将调用者挂起（suspend），直到目标线程 t 结束运行：t.join 方法会在当 t.isAlive 方法的结果为 false（见§4.3.2）时返回。带有一个时间（以毫秒为单位）参数的版本，会在规定的时间后将运行控制权交还给调用者，而不管当前线程是否结束了。根据 isAlive 方法的定义，调用一个没有启动的线程的 join 方法是没有任何意义的。同样原因，尝试调用一个还未创建的线程的 join 方法也是不明智的。

最初，Thread 类也支持额外的控制方法 suspend、resume、stop 和 destroy。方法 suspend、resume 和 stop 已经被宣布为*不建议使用(deprecated)*的方法；方法 destroy 从没有在任何发布版本中被实现，并且也许永远不会实现。方法 suspend 和 resume 的功能可以通过将要在§3.2 中讨论的等待和通知（waiting and notification）技术实现，且更加安全可靠。有关 stop 方法的问题会在§3.1.2.3 中讨论。

1.1.2.5 静态方法

有一些 Thread 类的方法只能应用于当前正在运行的那个线程中（也就是，调用 Thread 方法的线程）。为了强制实施，这些方法都被声明为 static。

- Thread.currentThread 方法返回当前 Thread 的一个引用。该引用可以用来调用这个线程的其他（非静态）方法。比如，Thread.currentThread().getPriority()将返回这个调用线程的优先级。
- Thread.interrupted 清除当前线程的中断状态，并且返回调用之前的状态值（因此，一个线程的中断状态不会被其他线程清除）。

java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

- Thread.sleep(long msecs)使得当前线程挂起至少 msecs 毫秒（见 § 3.2.2）。
- Thread.yield 仅仅是一个建议，用来告知 JVM：如果系统中有其他未处在运行状态的活动线程，那么调度程序可以从这些线程中选择一个运行而放弃当前的线程。JVM 可以使用自己的方式理解这个建议。

尽管缺少保证，但 yield 方法仍旧可以在一些单 CPU 的 JVM 实现上起到相应的效果，只要这些实现不使用分时（time-sliced）抢占式（pre-emptive）的调用机制（参见 § 1.2.2）。在这种情况下，只有当一个线程阻塞时（比如，由于 IO 操作或是通过 sleep 方法）其余线程才会被重新调度。在这些系统中，由于执行耗时的、非阻塞的计算任务的线程会占有更多的 CPU 时间，因而降低了应用程序的响应。为了安全起见，当执行非阻塞的计算任务的方法时［这些方法有可能耗费过多的时间，而超出事件处理者（event handler）或其他反应线程（reactive thread）的可接受的相应时间］，则可以在执行过程中插入 yield 方法（甚至是 sleep 方法），并且在需要的时候为这个线程设置较低的优先级。为了减少不必要的影响，可以只在偶尔的情况下调用 yield 方法，比如一个包含如下语句的循环：

```
if (Math.random() < 0.01) Thread.yield();
```

使用抢占式调度机制的 JVM 实现，特别是在多处理器的情况下，才可能甚至是要求调度管理程序忽略 yield 方法所给出的提示。

1.1.2.6 ThreadGroup

每一个线程都会作为一个 ThreadGroup 的成员构造，默认情况下是使用该调用线程构造函数的线程的 ThreadGroup。ThreadGroup 内部使用类似树的结构。当一个对象创建了一个新的 ThreadGroup 时，这个 ThreadGroup 就会被加入它所在的组。方法 getThreadGroup 返回线程所在的组。ThreadGroup 类包含类似 enumerate（枚举）的方法，用于指出当前有哪些线程属于这个线程组。

使用类 ThreadGroup 的一个目的是用来支持某些安全机制，这些安全机制可以动态地限制对线程操作的访问，比如，不允许调用一个不属于线程组的线程的 interrupt 方法。这只是一系列针对可能发生的问题的保护措施的一部分，如防止一个 applet 试图关掉屏幕显示更新主线程。ThreadGroup 也可以用来限定其组内的线程可以设置的最大优先级。

注意，ThreadGroup 不应该在基于线程的程序中被直接使用。基于应用程序（application-dependent）的出发点，在多数应用程序中，普通的集合类（如 java.util.Vector）是用来跟踪一组 Thread 对象的更好选择。

在 ThreadGroup 提供的少数方法之中，并发程序里较为常用的是 uncaughtException。当一个线程组中的线程由于未被捕捉的异常（比如，NullPointerException）而导致终止时，这个方法就会被调用。这个方法通常打印出当前的堆栈跟踪信息（stack trace）。

1.1.3 进阶阅读

本书并不是一本 Java 语言的编程参考手册（当然它也不是一本 how-to 指导教程，有关并

java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

发的教科书，一份实验报告，或是有关设计方法、设计模式以及模式语言的书，然而它包括了上述每一个方面的相关讨论。大多数章节都包括了一份资源列表，里面包括了有关讨论主题的更多信息。如果你需要进行很多并发编程的工作，则肯定希望更多地了解有关它的知识。

你可以通过查阅 ***JLS*** 来得到有关书中提到的 Java 编程语言结构的各种特性的权威论述：Gosling, James, Bill Joy 和 Guy Steele。《The Java™ Language Specification》，Addison-Wesley, 1996 在写作本书的时候，它的第二版正在计划之中，其中包含了有关 Java 2 平台的说明和更新内容。

有关 Java 语言的介绍：

Arnold, Ken 和 James Gosling。《The Java™ Programming Language, Second Edition》，Addison-Wesley，1998。

如果你从未使用线程编写过程序，也许会从下面这本书在线或发行版本中的 “Threads” 一节得到帮助：

Campione, Mary 和 Kathy Walrath。《The Java™ Tutorial, Second Edition》, Addison-Wesley, 1998。

一份简明的 UML 表示法的指南：

Fowler, Martin 和 Kendall Scott。《UML Distilled，Second Edition》，Addison-Wesley 1999。经过允许，本书的 3～4 页从中摘录了一些 UML 图例。

想要获得有关 UML 的更详细介绍，可以阅读：

Rumbaugh, James, Ivar Jacobson 和 Grady Booch。《The Unified Modeling Language Reference Manual》，Addison-Wesley，1999。

1.2 对象和并发

有很多方法可以用来描述对象、并发以及它们之间的关系。本节从多个方面进行了讨论（它们的定义，基于系统的，有关格式的和基于模型的各个方面），这些内容可以帮助你建立一个有关并发面向对象编程的概念基础。

1.2.1 并发

同大多数计算术语一样，为“并发”下一个定义是件非常困难的事情。通常而言，一个并发程序是一个可以同时处理多个任务的程序。例如，一个 Web 浏览器可以在处理一个用来得到一个 HTML 页面的 HTTP GET 方法的同时，播放一个音频片段，显示已接收的图片的字节数，以及处理用户的输入框。然而，这种同时性有时只是一种错觉。在某些计算机系统中，这些不同操作的确是由不同 CPU 完成的。而在另一些系统中，它们却是由一个单一的分时 CPU 处理的，这个 CPU 会在不同的操作中快速切换，以至于它们看上去是在同一时刻被处理，至少对于人类的感知而言，这些切换不会被察觉。

以下是一个更准确但有些抽象的对于并发程序的定义：一个 JVM 和其底层操作系统提供了从表象的同时性到物理的并行性（通过多 CPU）的映射。默认情况下，这种映射可以通过

java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

在适当的时候并行地执行独立的操作来实现，或通过分时来实现。并发程序由实现了这种并发方式的程序成分组成。Java 编程语言中的并发编程需要使用 Java 语言中相应的构件来完成，而不是像系统级的构件那样用来生成新的操作系统进程。通常，这个概念只是局限于在单个 JVM 上运行的相关构件，而不是应用在诸如使用远程方法调用（Remote Method Invocation，RMI）的分布式编程中。这种分布式编程包含了分布在不同计算机系统中的多个 JVM。

通过考量几种常见并发应用程序的特性，我们可以更好地理解并发和使用并发的原因：

Web 服务（Web Service）。多数基于套接字（socket）的 Web 服务（如 HTTP daemon、Servlet 引擎和应用服务器）都是多线程的。通常，支持多个并发的连接其主要目的是为了保证当有新的连接到来时可以被立即处理，而不需要等待其他连接结束。这种方式可以将服务的延迟减至最小，因而提高了可访问性。

数值计算。许多计算密集型的任务都可以被并行化处理，因此如果拥有多个 CPU 的话，这些任务就可以被更快地处理。这里是通过并行处理来使得程序的吞吐量（throughput）最大化。

I/O 处理。即便是在一个名义上的串行计算机中，那些处理有关输入输出的设备，如磁盘或是网络设备等，也都是独立于 CPU 运行的。并发程序可以利用这段时间进行其他工作，而不是把时间浪费在等待缓慢的 I/O 操作上。这样可以更高效地利用计算机的资源。

模拟程序。并发程序可以用来模拟物理世界中有着独立自主行为的对象，通常，这些行为很难被单纯的串行程序捕获。

GUI 程序。即便大多数用户界面（程序）本身都是单线程的，它们通常也会和多线程的服务交互和通信。并发处理可以使用户的操控响应及时，即便在处理许多耗时的工作中。

基于组件的软件。大粒度的软件组件（诸如排版、编辑之类的设计工具）都在内部生成了多个线程，以帮助程序记录、提供多媒体支持、获得更高的自主性或是提高性能等。

移动代码。类似`java.applet`的框架，可以在单独的线程中执行下载的代码。作为一系列策略的一部分，这样做可以用来隔离、监视和控制这些未知代码所带来的影响。

嵌入式系统。大多数运行在小型专用设备上的程序用来进行有关实时控制的工作。它们的每一个组件都不断对来自外界，如传感器或是其他设备的输入做出反应，并且及时地向外界返回相应的输出。《Java™ Language Specification》中规定，Java 平台并不支持严格的实时系统，这类系统的正确性依赖于是否在限定的时间段内执行完了有关的操作。对于一些以安全为第一要素的严格实时系统的要求，某些特殊的运行时系统提供了十分严格的保证。但是所有的 JVM 实现只支持不严格的实时控制，此时时间和性能仅作为服务质量（QoS）的因素考虑而不是正确性因素（参见 § 1.3.3）。这种可移植性保证了 JVM 可以实现在现今的多种不同目的的硬件和软件系统上。

1.2.2 并发执行构件

线程只是用来并发执行代码的多种可用构件中的一种。按注重自治性还是注重开销，创建一个操作可以有多种不同的方式。基于线程的设计并不一定是有关并发问题的最好解决方案。根据各自开销的不同，下列讨论的方法可以提供不同程度的安全性、保护、容错和管理控制。这些方案（和它们与之相关的编程支持构件）之间的区别更多地影响着设计策略，而

java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

不是各自方案的相关细节。

1.2.2.1 计算机系统

如果你有一群可用的计算机系统，则可以将每一个逻辑运算单元映射到一个单独的计算机上。每个计算机系统都可以拥有一个或是多个处理器，甚至可以是作为一个单独单元管理并且共享一个通用的操作系统的计算机集群。这种方式可以带来无限的自治性和独立性。每个系统都可以被单独地管理和操控，而与其他系统单元毫不相干。

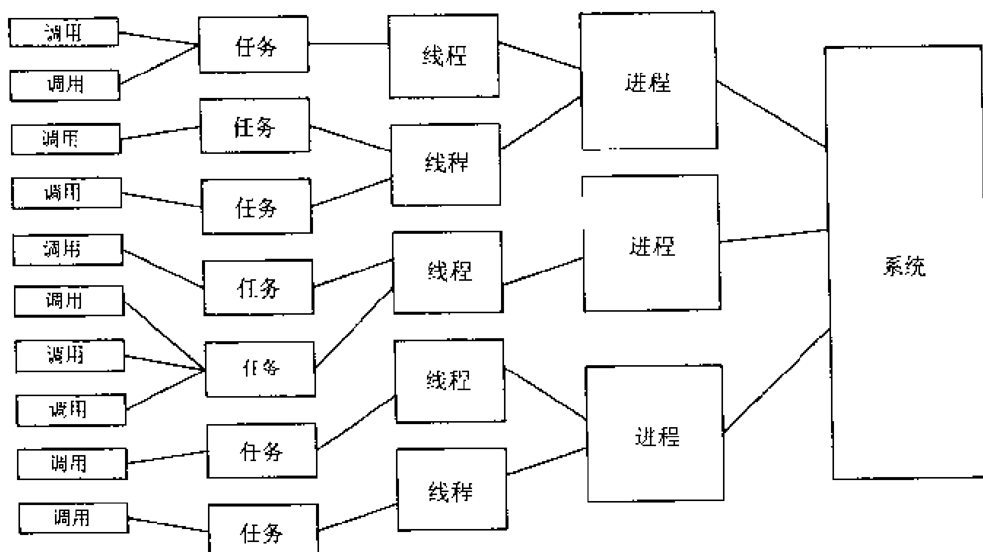
但是在这些实体间进行有关构造、定位、回收和传递消息的处理开销会非常昂贵，并且在它们之间共享本地资源的可能性也不存在了。与那些普通的并发程序相比，要在计算机系统中解决有关命名、安全、容错、恢复和可访问性的问题要困难许多。所以这种方式只适用于那些从本质上需要使用分布式解决方案的系统。并且，除了最小型的嵌入式系统之外，所有的系统都可以拥有多个进程。

1.2.2.2 进程

进程是一个操作系统级的抽象，它可以允许一个计算机系统同时支持多个运算单元。通常，每个进程表示一个单独运行的程序，比如，一个运行着的 JVM。同“计算机系统”的概念一样，进程也是一个逻辑的抽象概念而不是一个物理概念。因此，进程与 CPU 绑定的过程可能会动态地变化。

操作系统确保多个并发的、运行的进程之间具有某种程度的独立性和安全性，并且较少互相干扰。通常，进程不允许访问另一个进程的存储空间（虽然也会有特例），必须通过进程间的通信机制相互调用，比如使用管道。大多数系统至少会尽力保证进程的创建和调度。通常这种机制使用抢占型时间片调度方法——定期挂起进程以便可以给其他进程运行的机会。

在进程间的创建、管理和通信的开销相对于上一节提到的有关计算机系统的解决方式而言较小一些。但是，由于进程共享底层的计算资源（CPU、存储器及 IO 通道等），因此它们拥有较小的自治性。比如，一台机器会因为一个进程将其他进程都关闭而崩溃。



java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

1.2.2.3 线程

为了减少操作的开销，各种形式的线程都付出了更多的自治性作为代价。主要包括如下方面：

共享。线程可以共享内存、打开的文件和其他与单独进程相关的资源。Java编程语言中的线程可以共享所有类似的资源。有些操作系统同时支持一些中间意义上的构件，如“轻量进程”和“内核线程”，通过额外的请求或是其他限制条件，它们之间可以共享某些特定的资源。

调度。为了进一步减少调度机制的开销，独立性的保证可能会被进一步削弱。在一种极端的情况下，所有的线程都会被当作一个单线程进程一同处理。它们会一同竞争结果，任何时刻只有一个线程在运行，如果这个线程不被阻塞，其他线程就没有运行的机会（参见§1.3.2）。而对于另一种极端情况而言，通过抢先式调度规则，底层调度程序允许系统中所有的线程直接互相竞争。Java编程语言中的线程可以使用介于这两个极端之间的任何一种机制进行调度。

通信。系统可以通过有线或是无线的通道通信，比如通过套接字（socket）。进程可以使用这种方式，但是也可以使用例如管道和进程间的通信等更轻量的机制。除了上述提到的方法之外，线程还可以使用更“廉价”的方法：多个线程之间可以依赖共享的内存区间，并使用基于内存的同步机制（比如加锁和等待唤醒机制）来进行通信。使用这种方式进行通信更为高效，但是有时也会由于过于复杂而造成更多的程序错误。

1.2.2.4 任务和轻量级执行框架（lightweight executable framework）

虽然牺牲自治性来减少开销可以保证线程机制在多种应用程序中得到支持，但是它不能保证可以完美地满足某种特定操作的需求。尽管不同平台上线程的性能表现不同，但是花费在线程创建上的开销仍旧明显地大于那种最廉价的方式（但具有最低的独立性），即在当前线程中直接调用代码。

当线程的创建和管理的开销已经成为影响程序性能的因素之一时，通过创建你自己的轻量级执行框架，并加以使用上的限制（比如禁止使用某种方式的阻塞），提供更少的调度保证，或是提供更少的同步和通信手段，可以进一步通过减少自治性来提升性能。在§4.1.4中我们会讨论到，这些任务可以通过类似线程映射到进程和计算系统的方式映射到线程上。

与轻量级执行框架最为类似的系统是基于事件的系统和子系统（参见§1.2.3、§3.6.4和§4.1），在这些系统中，对于方法的调用会“触发”一个异步操作（这个操作并不是一个实际意义上的操作），它们会像事件一样被保存在队列中，并且由后台线程处理。其他几种轻量级执行框架会在第4章中讨论。构造和使用这些框架可以改善并发程序的结构和提升性能。这些框架的使用减少了有关（参见§1.3.3）并发执行技术不适合用来表示逻辑上的异步行为和逻辑上的自治对象的疑虑（参见§1.2.4）。

1.2.3 并发和面向对象编程

自从对象和并发的早期阶段，这两个概念就被联系在了一起。第一个并发的面向对象语

java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

言（大概诞生在 1966 年），***Simula***，同样也是第一个面向对象语言和最早的并发语言之一。早期的 Simula 语言中的面向对象和并发的构件相对比较原始和笨拙。比如 Simula 语言中的并发性基于 ***coroutines（协同程序）***——一种类似线程的成分，它们需要程序员定义从一个任务到另一个任务间的切换。之后的许多语言也同样支持了并发和面向对象构件——事实上，甚至是几种早期的 C++语言的原型版本也包括了几种并发支持类库。并且在 Ada 语言（虽然在其早期版本时，它几乎不是一种面向对象语言）的帮助下，把并发编程从特殊的、低级的语言和系统的世界中解放了出来。

面向对象设计在 20 世纪 70 年代出现的多线程系统编程的实践中并没有扮演什么实质性的角色。同样，对于出现在 20 世纪 80 年代的世界范围的面向对象编程热潮而言，并发也不占主要地位。但是对于面向对象的并发性的兴趣一致活跃在一些研究实验室和高级开发小组中，并且再度成为一种关键的编程方面，其中一部分的因素要归功于 Java 平台的流行和普及。

并发的面向对象编程享有这两种技术的大多数特性。但是它在许多关键地方与你所熟悉的编程方式不同，我们将在下面讨论这些差别。

1.2.3.1 串行面向对象编程

并发面向对象程序通常使用与串行面向对象程序同样的编程技术和设计模式（见 § 1.4 的例子）。但是从本质上而言，它们更为复杂。当同一时刻可以有多个行为发生时，程序的执行必然变得不可确定。代码可以按某种不可思议的顺序执行。只要没有额外的规定，任何顺序都是允许的（参见 § 2.2.7）。所以仅仅通过顺序阅读代码，你通常无法理解一个并发程序。比如，在没有进一步预先警告的情况下，一个成员变量在某行代码中被赋以一个值，而在执行下一行代码的时候这个变量可能被赋予了另一个值（由于某个其他的并发操作所致）。为了避免各种形式的干扰，通常在设计时需要更加严格和谨慎地考虑。

1.2.3.2 基于事件的编程

一些并发编程技术与那些在 GUI 工具包中应用的事件框架有许多的共同点，这些工具包通常包括在 `java.awt`、`javax.swing` 和其他语言中，比如 Tcl/Tk 和 Visual Basic。在 GUI 框架中，类似鼠标单击这样的事件被封装在 `Event` 对象中，并且被放入一个 `EventQueue` 对象里。然后，这些事件被一个单独的***事件循环（event loop）***一个一个地分派和处理，通常这个***循环***在一个独立的线程中运行。在 § 4.1 我们会讨论到，我们可以扩展这种设计以使可以支持额外的并发性，比如通过创建多个事件循环线程，每一个线程并发地处理事件，或是使用一个独立的线程分派每一个事件（或是通过其他策略）。同样，虽然这种方法增加了新的设计思路，但是也随之带来了有关并发操作中干扰和协调的新问题。

1.2.3.3 并发系统编程

面向对象的并发编程不同于 C 语言中的多线程系统编程，这主要是因为 C 语言中缺乏面向对象中的封装性、模块性、扩展性和安全性等特性。另外，并发支持是由 Java 编程语言本身提供的，而不需要额外的程序库支持。这样可以避免某些常见错误的发生，并且可以让编译器自动和安全地进行有关优化的工作，而不需要像在 C 语言中那样手工处理。

java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

虽然 Java 编程语言中的并发支持构件通常类似于标准 POSIX pthread 线程库和在 C 语言中使用的包，但是它们之间有着很重要的区别，特别是在有关等待和通知的细节上（参见 § 3.2.2）。当然，你也可以使用与 POSIX 程序（routine）类似的工具类（参见 § 3.4.4）。但是通常使用语言直接提供的机制会比使用额外的类库更为有效。

1.2.3.4 其他并发编程语言

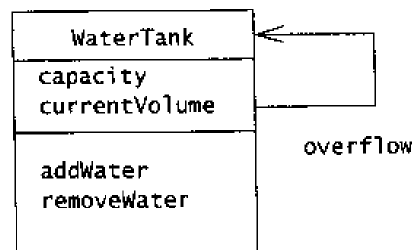
人们普遍地认为所有的并发编程语言都没有对并发特性进行全面的定义，因此从本质上来说，所有的并发编程语言从某种程度上看都是相同的。然而，通过开发包、类、实用程序、工具和编码约定来模拟其他语言中的特性，使一种语言所编写的程序看上去和另一种语言或是其他的并发构件开发出来的程序相同并不是十分的困难。书中讲述的构件可以提供如下功能和编程方式：基于信号机的系统（§ 3.4.1）；future（§ 4.3.3）；基于关卡的并行处理（§ 4.4.3）；CSP（Communicating Sequential Processes）（§ 4.5.1），以及其他内容。如果在符合要求的前提下，只使用上述提到的多种方式中的一种便可以实现你的程序当然是一个完美的解决之道。但是，许多并发设计、模式、框架和系统都有其渊源，并且在孜孜不倦地从其他各种可能的系统中吸取和继承各种优秀的思想。

1.2.4 对象模型和映射方式

在串行和并发的面向对象编程中，通常有关对象的概念是不同的，甚至即便在不同形式的并发面向对象程序中也是不同的。通过仔细研究底层的对象模型和映射方式可以展示在上一节中提到的多种编程方式之间的本质区别。

大多数人喜欢将软件对象看作真实对象的模型，他们可以根据需要任意地模拟真实世界中的对象。当然，这里的“真实”的含义是相对于观测者的理解而言，并且通常包含了只是在计算机领域中才有意义的技巧。

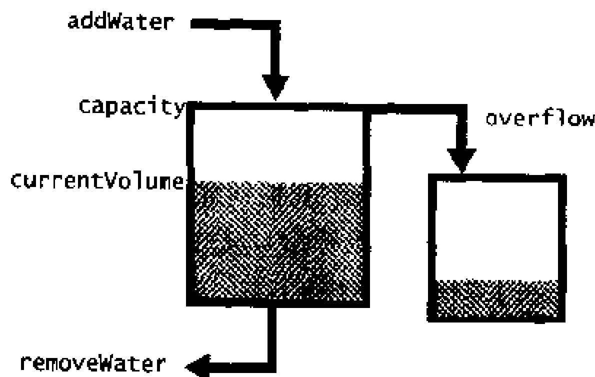
举一个简单的例子，考虑以下类 WaterTank 的框架 UML 类图和代码片断：




```
class WaterTank {                              // Code sketch
  final float capacity;
  float currentVolume = 0.0f;
  WaterTank overflow;

  WaterTank(float cap) { capacity = cap; ... }

  void addWater(float amount) throws OverflowException;
  void removeWater(float amount) throws UnderflowException;
}
```


java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

这里，使用下面介绍的内容来表示或模拟一个水槽：



- ***属性（Attribute）***，如 capacity 和 currentVolume。这些属性都由 WaterTank 对象的成员变量表示。我们可以从上下文中选择我们感兴趣的那些属性。比如，所有真实的水槽都有位置、形状、颜色和其他属性。这个类只与容积有关。
- ***不变约束（Invariant）***，用来声明约束关系。比如 currentVolume 通常大于 0 而小于容积（capacity）。capacity 是一个非负的值，并且在构造之后就不会改变。
- ***操作（Operation）***。用来描述类似 addWater（注水）和 removeWater（放水）这样的行为。有关操作的选择同样反映了一些隐含的设计思想，它们与程序所要反映的正确性、颗粒度和精确性相关。比如，我们可以对一个有着阀门和开关的水槽建模，也可以为每一个水分子建模，并使用相关的操作改变它们的位置。
- 使用类似水管或是其他水槽，与其他对象建立***联系（connection）***（或是***潜在的***联系），比如，每个水槽都附带的溢出水槽，这样在 addWater 操作时，溢出的水便可以注入这个水槽。
- 影响操作的***前提条件（Precondition）***和***结束条件（Postcondition）***。类似的规则包括：不可以从一个空水槽中取水，也不能向一个已经蓄满水且没有溢出水的水槽中注水。
- ***协议（Protocol）***。用来约束如何并且何时处理消息（操作的请求），比如，我们可以设定一条规则，规定任何时候最多只有一个 addWater 或 removeWater 消息可以被处理；相反，我们也可以规定一条规则，规定可以在调用 addWater 操作之中处理 removeWater 消息。

1.2.4.1 对象模型

WaterTank 类使用对象技术对真实世界进行建模。对象模型提供了一系列规则和框架，它们可以用来更普遍地定义对象。这些规则和框架包括：

静态（Statics）。每一个对象的结构（通常通过一个类）使用内部属性（状态）、与其他对象的联系、本地（内部）方法和用来接受来自其他对象的消息的方法或是入口来描述。

封装（Encapsulation）。对象有一层膜，将对象的内部和外部分隔开来。内部状态只能由这个对象本身修改（我们不考虑允许这种规则被破坏的一些语言特性）。

java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

***通信*（Communication）**。对象之间通过传递消息来进行通信。对象通过消息来触发其他对象中的行为。消息可以有多种形式，从简单的程序调用到通过任意通信协议传输。

***标识*（Identity）**。新的对象可以在任何时刻（需要符合系统资源的限制）被任何对象（受到访问控制的限制）构造。一旦被构造成功，每个对象在其生存期间会一直维护一个惟一的标识。

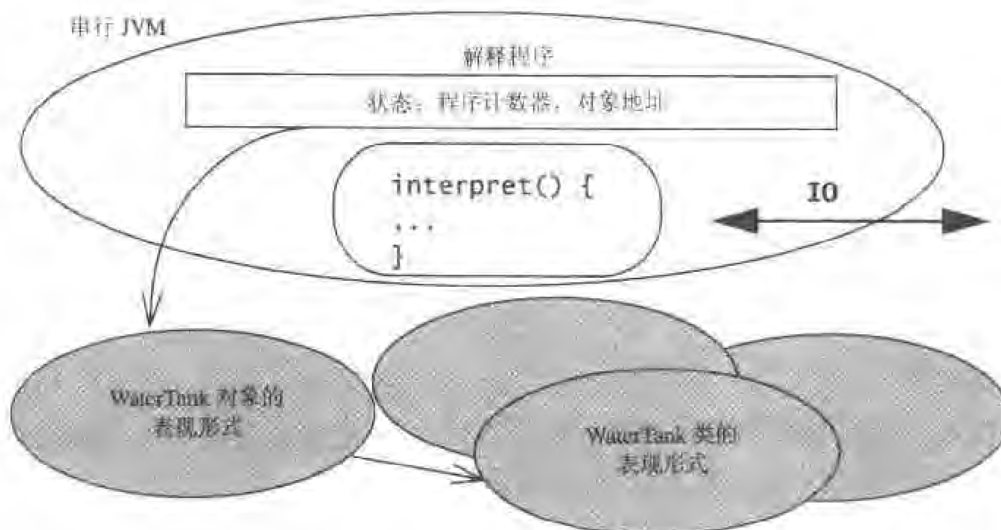
连接*（Connection）**。如果一个对象知道其他对象的标识，则可以向这些对象发送消息。一些模型使用通道***标识而不是普通的对象标识，或者在对象标识的基础上使用***通道***标识。从抽象的角度来看，通道就是用来传送消息的工具。两个共享同一通道的对象可以通过这个通道互相传递消息而不需要知道对方的标识。典型的面向对象模型和语言依赖基于对象的原语来进行直接方法调用，依赖基于通道的抽象元素进行 IO 和相关的连线通信。类似事件通道的机制可以同时用在上面两种情况之中。

***计算*（Computation）**。对象可以用来执行 4 种基本的计算：

- 接收一个消息。
- 更新内部状态。
- 发送一个消息。
- 创建一个新消息。

有多种方式可以用来解释和定义这种抽象性质。比如，一种实现 `waterTank` 对象的方法就是通过制造一个有着特殊目的的小型硬件装置来维护这些指定的状态、指令和连接。但是，由于本书并不是一本有关硬件设计的书，所以忽略了这种可能性，只把我们的注意力放在基于软件的解决方法上来。

1.2.4.2 串行映射方式



我们可以使用一台普通的计算机（包括一个 CPU、一条总线、内存和 IO 端口）来模拟任何对象，比如 `waterTank`。通过将一个有关 `waterTank` 的描述（通常是一个`.class` 文件）载入一个 JVM 即可以实现这个目的。这样 JVM 便可以构造一个实例的***被动*（passive）**

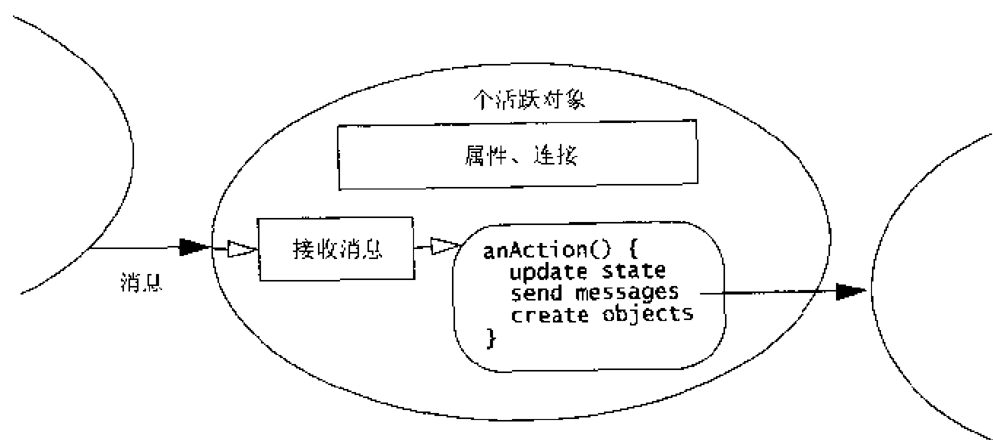
java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

表现形式，并且解释处理相关的操作。这种映射策略同样也可以在 CPU 层次上完成，这时操作被编译成为本地代码而不是由 JVM 解释的字节码。这种方式也可以扩展到包含多个不同类的对象的程序中，每一个类都根据需要被加载和实例化，只要 JVM 可以一直保存它正在模拟的对象的标识（“this”）。

从另一个角度看，JVM 本身也是一个对象，虽然这是一个十分特殊的对象，它可以表现为任何对象。[更规范的说法是，它是一个通用图灵机（Universal Turing Machine）]。Java 语言中的 Class 对象和反射机制可以更容易地描述将其他对象作为数据处理的反射对象，当然，类似的映射关系也同样存在于其他大多数语言中。

在一个纯粹的串行环境中，我们可讨论的内容就到此为止了。但是在进行下一部分的讨论之前，考虑一下使用这种方式实现通用对象模型的限制。在一个串行 JVM 中，不可能直接模拟多个对 waterTank 对象的并发操作。由于所有的消息传递都是通过串行的程序调用操作的，所以不必指定相关规则来约定是否可以并发地处理多个消息——这种情况不可能发生。这样，串行的面向对象处理方式限制了你希望表现的高级设计问题。

1.2.4.3 活跃对象（active object）



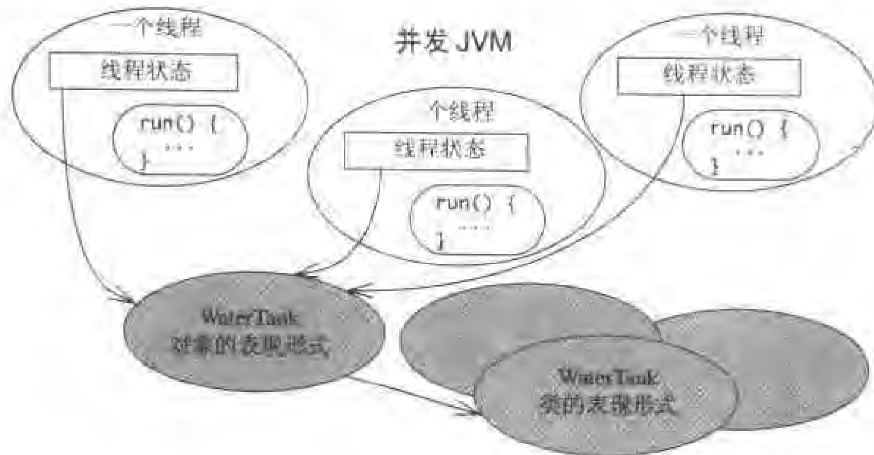
在多种映射方式中的另一个极端方式是活跃对象模型［通常也称为*行为*模型（actor model）］，这里每个对象都是自治的。每个都可能和一个串行 JVM 一样功能强大。模型内部的类和对象的表现形式可以和上一节中的被动框架使用同样的形式。比如在这里，每个 waterTank 都可以被映射到一个单独的主动对象之中，然后再被一个独立的 JVM 加载，并且可以让它永远地模拟这些定义的操作。

在分布式面向对象系统中，主动对象模型形成了一个通用的高级对象概念：不同的对象可以分布在不同的机器上，因此一个对象的位置和管理域通常是重要的编程问题。所有的消息传递工作都是由远程通信完成的（比如通过套接字），这些远程通信可以遵循多种协议，包括单向消息（这时，发送的消息不要求得到响应）、广播（同时将一个消息发送到多个接收处）和类似程序调用的请求-响应交换。

这种模型也可当作是从面向对象角度审视大多数操作系统级的进程，每一个进程之间都互相独立，并且互相之间尽量少地共享资源（参见 § 1.2.2）。

java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

1.2.4.4 混合模型

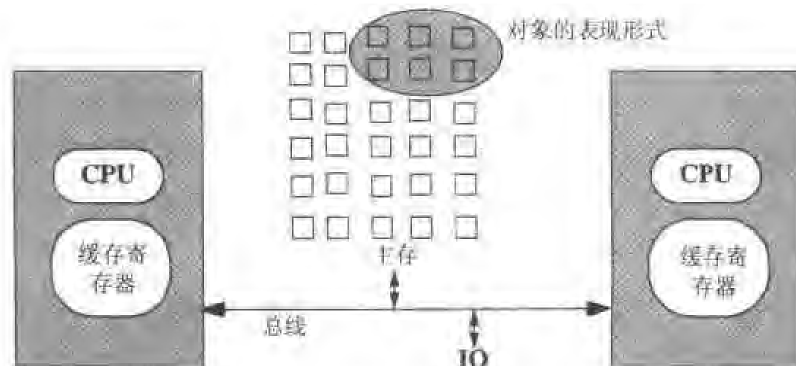


在 Java 编程语言中，支持并发的底层模型和映射方式介于被动模型和主动模型这两种极端形式之间。一个完整的 JVM 可能会由多个线程组成，每个线程使用类似于单独的串行 JVM 的方式活动。但是，同纯粹的主动对象不同，所有这些线程都可能共享同一套底层的被动表现形式。

这种映射方式可以模拟这两种极端形式的任何一种。可以只使用一个线程来模拟纯粹的被动串行模型。根据活跃对象的数目来创建线程，同时避免多个线程访问同一个被动对象的表现形式（参见 § 2.3），并且使用同远程消息传递（参见 § 4.1）相同的语义构件，便可以模拟纯粹的活跃模型。但是多数并发程序都使用一种折中的方案。

基于线程的并发面向对象模型从概念上将“普通”被动对象与活跃对象（线程）分离出来。但是被动对象通常表现出在串行编程中不存在的线程相关特征，比如通过锁来保护这些被动对象。相对于活跃模型中的活跃对象而言，这里所说的活跃对象则较为简单，只支持较少的方法（如 run 方法）。然而使用这两种方法中的任何一种都可以进行有关并发面向对象系统的设计（使被动对象适应多线程环境）或是限制活跃对象，这样它们就可以较为简单地由线程构件表示。

使用这种对象模型的一个原因是：对象模型可以使用一种直接并且有效的方式映射到单处理器或是 SMP（共享内存的多处理器）的硬件和操作系统上。可以根据需要，随时将线程绑定在 CPU 上，或是使用分时的方式：本地线程状态保存在寄存器和 CPU 中；使用共享主存储器的方式实现共享对象。



java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

程序员如何控制这些不同的映射，是一个可以用来区分多种并行编程与并发编程的度量手段。典型的并行编程需要明确的设计步骤来将线程、任务或是进程连同数据一起映射到物理处理器和它们的本地存储区域中去。而并发编程将这些问题留给 JVM（连同底层操作系统）来处理。这种方法增加了程序的可移植性，但需要考虑这些映射的不同实现的差异。

通过在线程上使用同样的映射策略可以实现分时：维护 Thread 对象的表现形式，并且使用一个调度程序***切换上下文状态***，将与一个线程相关的 CPU 状态值使用与其相关的存储形式保存起来，然后恢复另一个线程的运行。

可以在这种模型和映射方式的基础上进行修改和扩展。比如，***持久化***对象应用程序和系统通常依赖数据库来保存对象的表现形式而不是直接使用主存储器。

1.2.5 进阶阅读

有一系列丰富的关于并发技术的阅读资料，包括从理论基础的内容到有关并发应用程序的使用的实践指导。

1.2.5.1 并发编程

下列教科书包括了一些额外的并发算法的细节、编程策略和本书没有讨论到的标准方法：

Andrews, Gregory.《Foundations of Multithreaded, Parallel, and Distributed Programming》, Addison-Wesley, 1999. 它是在 Andrew 的《Concurrent Programming: Principle and Practice》（Benjamin Cummings,1991）的基础上的扩充和更新。

Ben-Ari, M.《Principles of Concurrent and Distributed Programming》， Prentice Hall, 1990。

Bernstein, Arthur 和 Philip Lewis.《Concurrency in Programming and Database Systems》, Jones and Bartlett, 1993。

Burns, Alan 和 Geoff Davis.《Concurrent programming》， Addison-Wesley, 1993。

Bustard, David, John Elder 和 Jim Welsh.《Concurrent Program Structures》。Prentice Hall, 1988。

Schneider, Fred.《On Concurrent Programming》，Springer-Verlag, 1997。

Java 编程语言的并发构件的由来可以追溯到 C.A.R.Hoare 和 Per Brinch Hansen 首次提出的类似的构件，可以从他们与其他人的论文中读到更多相关内容：

Dahl, Ole-Johan, Edsger Dijkstra 和 C.A.R. Hoare(eds.).《Structured Programming》, Academic Press, 1972。

Gehani, Narain 和 Andrew McGettrick (eds.).《Concurrent Programming》，Addison-Wesley, 1988。

下面是一个有关这些构件是如何在多种不同的语言和系统中被定义和被实现的比较调查：

Buhr, Peter, Michel Fortier 和 Michael Coffin. “Monitor Classification” ,《ACM Computing Surveys》， 1995。

并发面向对象，基于对象或是基于模块语言包括 Simula、Modula-3、Mesa、Ada、Orca、Sather 和 Euclid。有关这些语言的更详细的信息可以在它们相应的手册中找到，也可以参考以下内容：

java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

Birtwistle 和 Graham, Ole-Johan Dahl, Bjorn Myhrtag 和 Kristen Nygaard.《Simula Begin》, Auerbach Press, 1973。

Burns, Alan 和 Andrew Wellings.《Concurrency in Ada》, Cambridge University Press, 1995。

Holt 和 R.C.《Concurrent Euclid, the Unix System, and Tunis》, Addison-Wesley, 1983。

Nelson 和 Greg(ed.).《Systems Programming with Modula-3》, Prentice Hall, 1991。

Stoutamire, David 和 Stephen Omohundro.《The Sather/pSather 1.1 Specification》, 技术报告, University of California at Berkeley, 1996。

有关使用不同方法在 Java 编程语言中实现并发操作的书籍，包括：

Hartley 和 Stephen.《Concurrent Programming using Java》, Oxford University Press, 1998。本书使用一种操作系统的方法来实现并发。

Holub, Allen.《Taming Java Threads》, Apress, 1999。本书收集了作者在《JavaWorld》在线杂志上有关线程的专栏中发表的一系列文章。

Lewis 和 Bil.《Multithreaded programming in Java》。Prentice Hall, 1999。这本书包括本书中的一些主题的简要介绍，并且详细介绍了如何与 POSIX 线程集成。

Magee, Jeff 和 Jeff Kramer,《Concurrency: State Models and Java Programs》, Wiley, 1999。本书强调了有关建模和分析的内容。

更多的有关书、文章和使用线程进行系统编程的手册关注于那些特殊的操作系统和线程包的细节：

Butenhof 和 David.《Programming with POSIX Threads》, Addison-Wesley, 1997。本书对 POSIX 线程库及其使用方面进行了最详尽地讨论。

Lewis, Bil 和 Daniel Berg.《Multithreaded Programming with Pthreads Prentice Hall》, 1998。

Norton, Scott 和 Mark Dipasquale.《Thread Time》, Prentice Hall, 1997。

大多数有关操作系统和系统编程的文章描述了有关语言级线程和同步构件的底层支持机制的设计和构造的内容：

Hanson 和 David.《C Interfaces and Implementations》, Addison-Wesley,1996。

Silberschatz, Avi 和 Peter Galvin.《Operating Systems Concepts》, Addison-Wesley, 1994。

Tanenbaum 和 Andrew.《Modern Operating Systems》, Prentice Hall, 1992。

1.2.5.2 模型

考虑一下在软件中可以看见的各种不同形式的并发，你就不会对有如此之多与基础并发理论相关的方法而感到吃惊。过程计算、事件结构、线性逻辑、Petri 网，以及时序逻辑等理论说明都与理解有关并发面向对象的系统有着潜在的联系。你可以通过阅读如下资料对多数与并发理论有关的方法做一个初步的了解：

van Leeuwen 和 Jan(ed.).《Handbook of Theoretical Computer Science Volume B》, MIT Press, 1990。

一本有关模型、相关的编程技巧和设计模式的选集（里面的内容至今仍没有过时），并且使用了多种语言和系统做了示例：

java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

Filman, Robert 和 Daniel Friedman.《Coordinated Computing》，McGraw-Hill, 1984。

有很多基于主动模型的实验性质的并发面向对象语言，其中最著名的是 ***Actor*** 语言家族：

Agha 和 Gul.《ACTORS: A Model of Concurrent Computation in Distributed Systems》，MIT Press, 1986。

针对在并发中使用面向对象方法的广泛调查可以在下面找到：

Briot, Jean-Pierre, Rachid Guerraoui 和 Klaus-Peter Lohr. "Concurrency and Distribution in Object-Oriented Programming",《Computing Surveys》，1998。

有关面向对象模型、系统和语言的研究论文可以在下列有关面向对象的会议录中找到，包括《*ECOOP*》、《*OOPSLA*》、《*COOTS*》、《*TOOLS*》和《*ISCOPE*》。同样也可以在有关并发的会议文献，比如《*CONCUR*》和类似于《*IEEE Concurrency*》的刊物中找到。另外，下面一系列丛书中的章节包括了许多相关方法和问题的研究：

Agha, Gul, Peter Wegner 和 Aki Yonezawa(eds.).《Research Directions in Concurrent Object-Oriented Programming》，MIT Press, 1993。

Briot, Jean-Pierre, Jean-Marc Geib 和 Akinori Yonezawa (eds.).《Object Based Parallel and Distributte Computing, LNCS 1107, Springer Velrlag, 1996。

Guerraoui, Rachid, Oscar Nierstrasz, and Michel Riveill (eds.).《Object-Based Distributed Processing》，LNCS 791, Springer-Verlag, 1993。

Nierstrasz, Oscar 和 Dennis Tsichritzis(eds.).《Object-Oriented Software Composition》，Prentice Hall, 1995。

1.2.5.3 分布式系统

有关分布式算法、协议和系统设计的文章包括：

Barbosa 和 Valmir.《An Introduction to Distributed Algorithms》. Kaufman, 1996。

Birman, Kenneth 和 Robbert von Renesse.《Reliable Distributed Computing with the Isis Toolkit》，IEEE Press, 1994。

Coulouris, George, Jean Dollimore 和 Tim Kindberg.《Distributed Systems: Concepts and Design》，Addison-Wesley, 1994。

Lynch 和 Nancy.《Distributed Algorithms》，Morgan Kaufman, 1996。

Mullender 和 Sape(ed.),《Distributed Systems》，Addison-Wesley, 1993。

Raynal 和 Michel.《Distributed Algorithms and Protocols》，Wiley, 1988。

有关使用 RMI 进行分布式编程的细节，见：

Arnold, Ken, Bryan O'Sullivan, Robert Scheifler, Jim Waldo 和 Ann Wollrath.《The Jini™ Specification》，Addison-Wesley, 1999。

1.2.5.4 实时编程

大多数有关实时编程的资料都关注于***硬件实时***系统，它们的主要目的是为了保证系统的正确性，某些行为必须在一定的时间内执行。Java 编程语言本身不支持这种保证性，因此本书并不涉及有关基于期限的调度、优先级分派算法和相关的问题。有关实时设计的资料包括：

java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

Burns, Alan 和 Andy Wellings.《Real-Time Systems and Programming Languages》, Addison-Wesley, 1997。这本书描述了如何使用 Ada、occam 和 C 语言进行实时编程，并且包括了有关优先级倒置问题和解决方案说明。

Gomaa 和 Hassan.《Software Design Methods for Concurrent and Real-Time Systems》, Addison-Wesley, 1993。

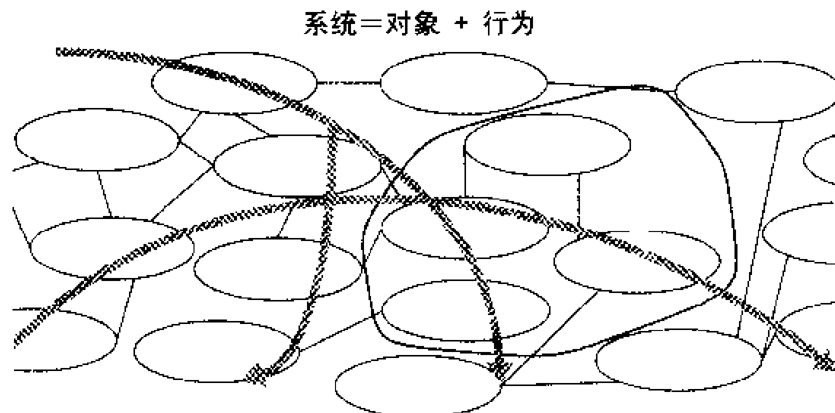
Levi, Shem-Tov 和 Ashok Agrawala.《Real-Time System Design》, McGraw-Hill, 1990。

Selic, Bran, Garth Gullekson 和 Paul Ward,《Real-time Object-Oriented Modeling》, Wiley, 1995。

1.3 设计因素

本节审视了一些有关设计方面的问题，它们大多出现在并发软件的开发过程里，而很少出现在串行程序中。本书后面将要讨论到的大多数有关构造和设计模式的内容包括了它们如何解决这里讨论到的有关应用的问题（也包括其他与并发联系较少的内容，比如准确性及可测试性等等）。

我们可以同时从两个互补的角度来观察任何面向对象系统：以对象为中心和以行为为中心。



从以对象为中心的视角出发，一个系统就是一系列互相联系的对象的集合。但是这是一个结构化的集合，而不是一个随机的对象大杂烩。多个对象先聚集成组，比如组成一个 ParticleApplet 的对象，然后再组成更大的组件和子系统。

从以行为为中心的视角而言，一个系统是一组可能发生的并发行为的集合。在最小粒度的层次上，它们只是独立的消息传递（通常是方法调用），它们依次组成调用链、事件序列、任务、会话、事务和线程。一个逻辑行为（比如运行 ParticleApplet）可能包括多个线程。从更高的层次上看，部分行为代表着系统范围的用例。

这两幅视图的任何一个都不能单独构成一个完整的系统，因为一个特定的对象可能参与多个行为，而一个特定的行为也可能与多个对象相关。然而，从这两个视图中会产生两个互相依赖的有关***正确性***的问题，一个是与对象为中心的，而另一个是以行为为中心的：

安全（***Safety***）。没有什么比无法保证一个对象的安全更糟糕的事情了。

活跃性（***Liveness***）。一个行为最终总是应该会发生的。

java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

安全方面的错误会导致在运行时刻发生无法预期的行为——程序会朝着错误的方向发展。而活跃性方面的错误会导致没有任何行为发生——最终程序会停止运行。让人沮丧的是，有很多你可能认为能够提高活跃性的简单措施反而会破坏安全属性，反之亦然。想要把它们同时调理好，是一个巨大的挑战。

因此在你自己的程序中，你不得不在这些不同类型错误而造成的影响中寻找平衡。但是将安全问题作为基本的设计重点是一个标准工程（而不只是软件工程）实践。你的代码越重要，你的程序就越必须保证安全性，不作任何事情总比危险的行为要好。

从另一个方面来说，在实际的调试并发设计的工作中，我们将大部分时间都花费在与活跃性和与活跃性相关的效率问题上。有很多理由值得我们用牺牲安全性的方式来换取活跃性。例如，如果由于不协调的并发操作而引起一个显示器只是暂时地显示了一些无意义的图像（比如绘制了偏离的像素、不正确的过程指示器或是失真的图像），并且如果你确信这种状态很快会被纠正，那么这些错误情况就是可以接受的。

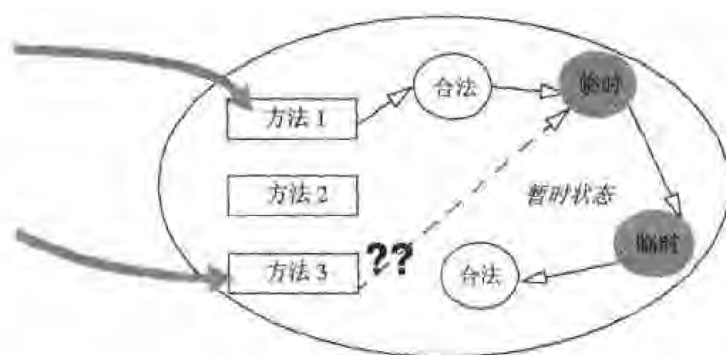
安全和活跃性的问题可以被进一步扩展，这样它们会包含两类与***软件质量***有关的问题，一个主要是以对象为中心，而另一个主要是以活跃性为中心。它们有时处在完全对立的位置上：

可重用性（Reusability）。在多个不同的上下文中使用对象和类。

性能（Performance）。一个操作/行为执行的快慢程度。

本节的余下部分会进一步对并发程序中有关安全、活跃性、性能和可重用性的问题进行讨论。其中的内容包括了基本的术语和定义，以及一些核心问题和策略的简要介绍，有关这些问题和策略的讨论会一直贯穿本书。

1.3.1 安全



安全的并发编程实践是从安全的串行编程实践中发展而来的。并发设计中的安全因素在普通的***类型***安全概念的基础上增加了一个时间维。一个具有类型检查功能的程序可能不一定正确，但是至少它不会造成危害，比如将一个表示 `float` 类型的比特串当作一个对象引用。同样，一个安全的并发设计也许不一定能够达到预想的效果，但它至少不应该由于竞争的线程而造成表现形式的崩溃。

从实践上看，类型安全和多线程安全之间的一个区别是多数类型安全的问题可以由编译器进行自动的检查来解决。一个没有通过编译期检查的程序是不能运行的。而多数有关线程安全的问题通常不能被自动地检查，而必须依赖程序员的经验和相关的训练才能够避免。有

java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

关如何证明一个设计是否正确的方法的讨论超出了本书的范畴（见进阶阅读）。这里讨论的用来确保安全的技术是经过仔细的工程实践（包括一些基于形式方法的实践）的验证的，而不仅仅是一种正式的方法。

同样，多线程安全在有关***保密***（*security*）的设计和编程技术的基础上增加了一个时间维。保密编程实践禁止有关调用者、程序及主体访问某些对象上的特定方法或是资源。并发控制引入了一个称为***瞬时***（*transient*）的概念，用来表示由于某些操作当前正在由其他线程执行，因此禁止访问。

保证安全的主要目的是为了确保一个系统中的所有对象都能够维护***一致***的状态：所有的类成员变量和它们所依赖的所有其他类成员变量的状态都维护其合法且有意义的值。有时需要花费很多努力来明确“合法”和“有意义”对一个特定的类所代表的含义。其中一种方法是先从概念上建立某种***不变约束***，比如水槽的容积通常介于零和它的最大容量之间。这些通常可以表述为具体的相关类的成员变量之间的关系。

如果一个对象的所有成员变量都遵从它们的不变约束，那么这个对象就是***一致的***。每个类的每个公有方法的职责应该都是将对象从一个一致性状态转移到另一个一致性状态。安全对象有可能会在方法执行的中间偶尔处于暂时的不一致性状态。但是当它们在这种不一致的状态下，不应该试图调用一个新的操作。如果每个对象的操作都完全按照其逻辑功能所设计的那样，并且这些功能都被正确地实现，你便可以确信，使用这些对象的程序不会由于对象的不一致性而出错。

在并发程序中需要更加关注不变约束的一个理由是，相对于大多数串行程序而言，并发中的不变约束更容易被破坏。在串行上下文中，同样需要防止由于不一致性而造成的错误，比如处理异常和回调时，或是从类的一个方法对另一个方法的自我调用。这些问题在并发程序中会变得更为突出。在 § 2.2 中我们会讨论到，确保一致性的最通用的方法是使用独占技术来确保公有操作的***原子性***——每个操作在完成之前不会被其他操作打断。没有这些措施的保护，***竞争条件***造成的***存储冲突***会导致并发程序不一致性问题的出现：

读/写冲突（*Read/Write conflict*）。一个线程在对一个变量的值进行读操作时，另一个线程正在对它执行写操作。这样很难预测读线程所看到的变量值——哪个线程赢得了这场“竞赛”就能第一个对这个成员变量操作。在 § 2.2 中我们会讨论到，这个读到的值甚至可能不是由线程写入的。

写/写冲突（*Write/Write conflict*）。两个线程同时试图对同一个变量执行写操作。同样，我们很难或是不可能预见下一个读操作所看见的值。

同样原因，当一个对象处于不一致状态时，相应的操作后果也不可能预见。这样的例子包括：

- 一个图形的程序（比如一个 `Particle` 程序）将一个对象绘制在该对象没有出现的位置上。
- 由于在自动转账过程中取钱，而造成了银行账目的不平衡。
- 一个链表的 next 指针指向了一个并不属于这个链表的节点。
- 两个并发的传感器的更新导致实时控制器执行了一个不正确的响应操作。

java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

1.3.1.1 属性和约束

安全的编程技术依赖于对对象表现相关的需求属性和约束性问题有着清晰地理解。没有意识到这些问题的开发人员很少能够在自己的工作中很好地体现出这些因素。有很多形式方法可以用来精确地描述这些需求所说明的约束关系（大多数有关并发设计方法的文章可以在进阶阅读中给出的列表中找到）。这些内容虽然很有用，但是在本书中我们只关注内容的准确而不引入这些相关的形式方法。

一致性的需求经常来自于在进行有关类的初始设计过程中所做出的对高层概念属性的定义。这些约束通常并不关心这些属性是如何被具体地表示和如何被成员变量和方法来访问的。这可以从本章先前讨论的 `WaterTank` 和 `Particle` 的例子中看出。这里还有一些其他的例子，这些例子的大多数会在本书的教程中再次详细地讨论：

- 一份 BankAccount（银行账户）的结余（balance）应该等于存款及利息的总和减去支取数目和服务收费。
- 一个 Packet（IP 包）有一个合法的 IP 地址作为它的目的地址。
- 一个 Counter（计数器）有一个非负的整型计数值。
- 一张 Invoice（发票）有一个付款期限属性，表明了支付系统的规则。
- 一个 Thermostat（自动调温器）维护着一个温度值，它应该等于传感器测量到的最近的温度值。
- 一个 Shape（图形）有位置、维数和颜色属性，它们都遵守一个特定的 GUI 工具的样式规则。
- 一个 BoundedBuffer（有界缓冲）有一个元素计数，它的大小通常处于零和缓冲区上限之间。
- 一个 Stack（堆栈）有它的大小，并且当堆栈非空的时候，有一个顶层元素。
- 一个 Window（窗口）有一个属性集，用来维护当前有关字体、背景颜色等属性的映射。
- 一个 Interval（时间间隔）对象有一个开始时间（startDate），该时间不能晚于它的结束时间（endDate）。

从本质上而言，这些属性最终会通过某种方法映射到对象的成员变量上，但是这种对应并不需要是直接的。比如，处在堆栈顶部的元素并不需要由一个变量维护，而可以作为一个数组元素或是链表节点保存。同样，某些属性可以通过对其他属性的计算（“衍生”）而得到；例如，BankAccount 拥有的一个布尔型（`boolean`）变量 `overdrawn`（透支），可以通过判断账目结余是否等于零来确定这个变量的值。

1.3.1.2 表现约束（representational constraint）

进一步的约束和不变约束通常会作为一个类的额外的实现决策而出现。定义一个成员变量的目的可以是为了维护一个特殊的数据结构、提高系统的性能或是为了其他的内部维护需要，它通常需要遵守一系列不变约束规则。主要的成员变量及其约束的类型包括：

直接值表示（Direct value representations）。直接使用成员变量实现具体的属性。比如，

java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

一个 Buffer（缓冲）对象会使用一个 putIndex 成员记录数组的索引，用来提示下一个添加元素的位置。

缓存值表示（Cached value representations）。成员变量用来避免或是减少计算和方法调用的需要。比如，一个 BankAccount 可以维护一个 overdrawn（透支）变量，这个变量只有当前结余小于零的时候为真。这样便不需要每次使用这个属性的时候都计算它的值。

逻辑状态表示（Logical state representations）。用来反应逻辑控制状态。比如，一个 BankCardReader（银行卡阅读机）可以维护一个 card 变量，用来指示当前正在被读取的卡，validPIN 变量记录这个 PIN 访问码是否已经被验证过。这个 CardReader 的 validPIN 变量可以在一个协议中记录卡是否被成功地读取和验证的信息。有些状态表示采用***角色变量（role variable）***的形式，控制着一系列相关的方法（有时这些方法在一个单独的接口中定义）的响应。比如，一个 game-playing 对象会在主动和被动角色间切换，这种切换规则依赖于 whoseTurn 变量的值。

执行状态变量（Execution state variable）。成员变量记录着对象最小粒度的动态状态，比如记录一个特定操作是否正处在处理过程中。执行状态变量可以表示一个指定的消息是否已经被接受，相应的响应操作是否被触发，操作是否结束，或者是否发出了返回消息。执行状态变量通常是枚举类型，并且这些变量的名字通常以*-ing* 结尾；比如 CONNECTING、UPDATING 和 WAITING。另一种经常使用的执行状态变量用于记录某些方法的进入次数或离开次数。正如我们将在 §3.2 中所讨论的，并发程序中的对象总是比串行上下文中的对象需要更多的此类变量，它们可以用来跟踪和管理这些异步处理方法的执行过程。

历史变量（History variable）。用来表示一个对象的历史信息或是历史状态。最完善的方式是***历史日志（history log）***，它记录了曾经接受和发送过的所有消息，连同所有开始和完成的相应的内部操作及状态的改变。另外一种方法，虽然不是非常完善，但是更为常见。比如，BankAccount 类可以维护一个 lastSavedBalance 变量，该变量用来保存最近检查点的值，在恢复一个已取消的事务时使用。

版本跟踪变量（Version tracking variable）。用来记录版本信息的变量，通常是用整数、时间戳、对象引用、签名代码（signature code）或是其他表现形式来表示相关的时间、顺序或与一个对象最近改变的状态相关的属性。比如，一个 Thermostat 对象可以在更新其 temperature（温度）的值时，递增一个 readingNumber 变量或是记录一个 lastReadingTime 变量。

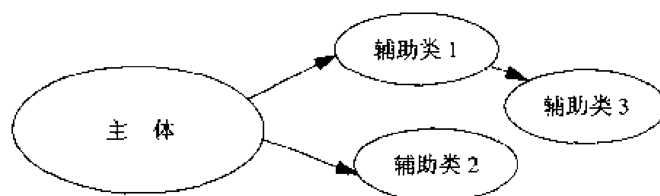
所需对象的引用（Reference to acquaintance）。用来指向与一个主体（host）交互的对象的成员变量，但是这些成员变量本身并非代表这个主体的逻辑状态：比如，一个 EventDispatcher 的 callback（回调）目标对象，或是一个代理 requestHandler 对象的 WebServer 对象。

表示对象的引用（References to representation object）。在概念上由主对象体持有的属性，但是实际上却由其他辅助类负责管理。引用变量可以指向其他对象，它们协助主体对象表示主体对象的某些状态。所以，任何对象的逻辑状态都可以包括它所拥有的这些引用所指向对象的状态。另外，这些引用变量本身也是主对象的具体状态的一部分（参见 §2.3.3）。任何

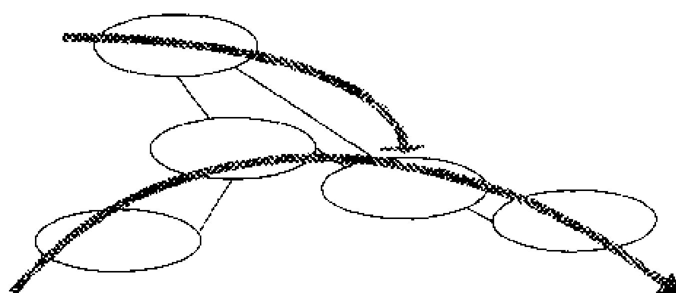
java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

想要确保安全的方法都必须仔细考虑这些对象之间的关系。比如：

- 一个 Stack（堆栈）有一个 headOfLinkedList 变量，用于记录这个堆栈列表的第一个节点。
- 一个 Person（人）对象会维护一个 homePageURL 变量，它是一个 java.net.URL 对象。
- 一个 BankAccount 的结余信息（balance）可能会由一个集中的系统维护，在这种情况下，这个 BankAccount 需要保存一个指向这个系统的变量（以便可以从它那里得到当前的结余信息）。这时，该 BankAccount 的某些逻辑状态事实上是由这个系统所管理。
- 对象可能只有通过访问由其他对象维护的属性列表（property list）才能得知它自身的属性。



1.3.2 活跃性



构造一个可靠的安全系统的一种方法是保证没有对象执行任何方法，这样便永远不会遇到任何冲突。但这似乎并不是一个有效的编程模式。安全方面的问题必须与活跃性方面的问题[1]保持平衡。

在一个运行的系统中，每个行为最终都会结束；每一个被调用的方法最终会开始执行。但是一个行为可能（也许只是暂时的）会由于下列互相关联的原因而不能继续运行：

锁（Locking）。由于另一个线程持有了一个同步方法所需要的锁，因此阻塞了这个线程的执行。

等待（Waiting）。由于另一个线程没有产生所需的事件、消息或是条件，所以导致一个方法因为等待而阻塞（通过 `Object.wait` 及其相关引申方法）。

输入（Input）。因为所需的来自另一个进程或是设备的输入没有到来，因此一个基于 IO 的方法由于等待而阻塞。

CPU 竞争（CPU contention）。即便一个线程处在可运行状态，也可能因为其他线程或

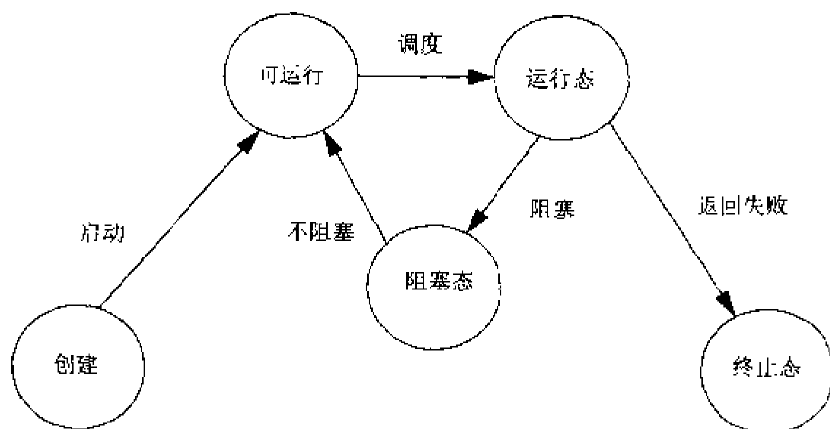
1　一些有关“活跃性”的属性可能被作为一系列线程对象的安全属性对待。比如，设定 deadlock-freedom（无死锁）属性可以用来避免一系列进程因无休止地互相等待而造成的错误状态。

java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

是运行在同一个计算机上的另一个完全独立的程序占用了 CPU 或是其他计算资源而失败。

失败（Failure）。线程中正在运行的方法遇到一个永久的异常、错误或故障。

线程操作过程中的瞬间的阻塞通常是可以接受的。事实上，频繁的短暂阻塞是许多并发编程模型的固有特性。典型线程的生命周期由许多瞬时的阻塞状态和重新调度组成：



但是，***永久性*** 或是无限制的休眠通常是严重的问题。潜在的永久性失败的例子会在本书的其他部分给予更深的讨论，包括：

死锁（Deadlock）。由于对锁的循环依赖而造成死锁现象。最常见的情况是，线程 A 持有了对象 X 的锁，并且正在试图获取对象 Y 的锁。同时，线程 B 已经拥有了对象 Y 的锁，并在试图获取对象 X 的锁。因此没有哪个线程能够执行进一步的操作（参见 § 2.2.5）。

错过唤醒通知（Missed signal）。如果一个线程在用于唤醒它的通知发出之后才进入等待（参见 § 3.2.2），它就会一直停留在静止状态。

嵌套的监视器互锁（Nested monitor lockout）。一个正在等待的线程持有一把锁，并且其他的用来唤醒这个线程的线程需要使用这把锁（参见 § 3.3.4）。

活锁（Livelock）。对于某个操作的持续尝试，却持续失败（参见 § 2.4.4.2）。

无 CPU 资源（Starvation）。JVM/OS 没有能够成功地为一个线程申请到 CPU 时间。这种情况可能是由于调度策略或主机遭到有敌意的拒绝服务攻击（DOS）而造成的（参见 § 1.1.2.3 和 § 3.4.1.5）。

资源耗尽（Resource exhaustion）。一组线程共同拥有所有当前有限数目的资源。其中一个需要额外的资源，但是没有其他的线程愿意放弃其所持有的资源（参见 § 4.5.1）。

分布式错误（Distributed failure）。通过套接字（socket）连接的、被看作是输入流 `InputStream` 的远程机器崩溃或是变得无法访问（参见 § 3.1）。

1.3.3 性能

基于性能的要求扩展了有关活跃性方面的问题。除了需要保证每个被调用的方法最终都能够得到执行之外，为了满足性能的要求，还要求这些方法能够被迅速、及时地执行。虽然我们在本书中并不考虑有关硬件实时系统中的问题（在那种系统中，某个方法在某段指定的时间内没有成功地执行会导致灾难性的系统错误），但是几乎所有的并发程序都有内含的或

java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

是明确的性能要求。

有意义的性能需求通常可以通过一些可测量的性质描述，包括如下的度量标准。性能的需求可以由多次测量的总体趋势（比如平均值和中值）和它们的变化（比如，范围及标准方差）来表示。

吞吐量（Throughput）。每个单元时间内执行的操作数目。我们感兴趣的操作类型包括从单独的方法到完整的运行的程序。多数情况下，吞吐量并不是用一个比率值来描述，而是使用执行一个操作所要耗费的时间来描述的。

延迟（Latency）。发出一个消息（通过比如一个鼠标单击、方法调用或是一个收到的套接字联结）和服务这个消息的时间间隔。在操作均匀、单线程执行且持续请求的方法的上下文中，延迟只是吞吐量的倒数。但是在更通常的情况下，延迟反映了响应时间——直到***某些事情***发生之前的程序延迟，而并不一定需要等到一个方法或是服务的完成。

容量（Capacity）。在规定的一个最小吞吐量或是最大延迟的情况下，系统所支持的同时发生的操作数目。特别是在一个网络应用程序中，这个量度值可以用作一个有关全局***可用性***的有用的指导参数，因为它反映了在没有由于超时或是网络队列溢出的原因而造成断线的情况下，系统可以服务的客户的数目。

效率（Efficiency）。等于吞吐量除以其所需要支持的计算的资源数目（比如 CPU、内存和 IO 设备）。

可扩展性（Scalability）。当一个系统中有新的资源（通常包括 CPU、内存和 IO 设备）加入时，延迟性或吞吐量增加的比率。相关的度量方式包括***可利用性（utilization）***——一个任务可以使用的资源的百分比。

劣化（Degradation）。在没有增加资源的情况下随着客户、操作和行为增加，系统延迟或吞吐量下降的比率。

大多数多线程设计都毫无疑问地通过牺牲一小部分计算效率来得到更好的延迟性和可扩展性。并发支持引入了如下几种可能降低程序执行速度的程序开销和竞争类型：

锁（Lock）。synchronized 方法通常比非同步方法需要更多的调用开销。同样，一个经常由于等待锁（或是因为其他任何原因）而阻塞的方法比没有这些阻塞的方法执行得慢。

监视器（Monitor）。`Object.wait`，`Object.notify`，`Object.notifyAll` 和从这些方法引出的方法（比如 `Thread.join`）会比普通的 JVM 运行时操作需要更多的开销。

线程（Thread）。创建和启动一个线程通常比创建一个普通对象和调用其中的一个方法开销更大。

上下文切换（Context-switching）。将线程映射到 CPU 上去需要面对上下文切换的开销，这时，一个 JVM/OS 将与一个线程相关的 CPU 状态保存起来，并选择另一个线程来执行，以加载相关的 CPU 状态。

调度（Scheduling）。用来挑选一个适合的线程来执行时所需要执行的计算和相应的底层机制所增加的额外开销。这些操作可能会进一步与其他系统事务（比如处理异步事件和垃圾收集）进行交互。

位置（Locality）。在多处理器的环境下，当运行在不同 CPU 上的多个线程想要访问相

java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

同的一些对象时，用来维护缓存一致性的硬件和底层系统软件必须在多个处理器之间交换这些数据的值。

***算法*（Algorithmic）**。一些高效的串行算法并不能在并发设置下使用。比如，某些依赖缓存的数据结构只有在所有的操作都是由同一个线程执行时才能够正常工作。但是，有许多有效的并发算法能够选择，包括那些可以通过并行机制来提升速度的算法。

与并发构件相关的开销随着 JVM 的更新而逐渐减少。比如，在编写本书的时候，在最新开发的 JVM 上面运行一个不包含任何操作，且不参与竞争的同步方法的开销同几个非同步空操作的方法的调用相差无几（由于不同类型方法的调用需要花费不同的时间，并且会使用不同的优化手段，比如静态方法与实例方法，因此在这里并不值得对此进行更详细的讨论）。

但是这些开销的递减方式是非线性的。比如，10 个线程频繁地竞争同一把锁，与让其中的每一个线程经过 10 个不参与竞争的锁相比，前者具有更差的全局性能表现。并且，由于并发支持所需的底层资源管理部分通常只是根据指定的目标负载而进行优化，因此当在系统中过多地使用了锁、监视器操作或线程时会导致性能急剧下降。

接下来的章节包括如何根据需求减少使用并发构件的讨论。但是应该牢记的是，任何有关性能的问题只有在这些性能要素可以被度量和分离之后才可以被解决。在没有事实论据的情况下，多数有关性能问题的本质和问题来源的猜想都是错误的。最常用的度量方法是通过比较不同的设计、负载和配置情况来展示性能的差别和变化的趋势。

1.3.4 重用性

类或是对象可以被重用是指它可以轻易地在不同的上下文中被使用，而不论这种使用方式是作为一个黑盒组件重用，还是通过子类化或相关技术作为一个白盒扩展的基础而使用。

安全和活跃性方面的问题之间的相互作用会明显地影响重用性。通常，设计一个在所有可能场景中都是安全的组件是可以实现的。比如，无论一把同步锁是被如何使用的，一个被阻塞的同步方法只有在它获得了该同步锁后才会继续工作。但是在某些情况下，使用这个安全组件的程序也许会遇到活跃性的错误（比如，死锁）。相反，一个只使用非同步方法的组件可以保证其功能会*一直*保持有效（至少相对于被锁住），但当它允许被多个并发的操作访问时则可能会遇到安全的问题。

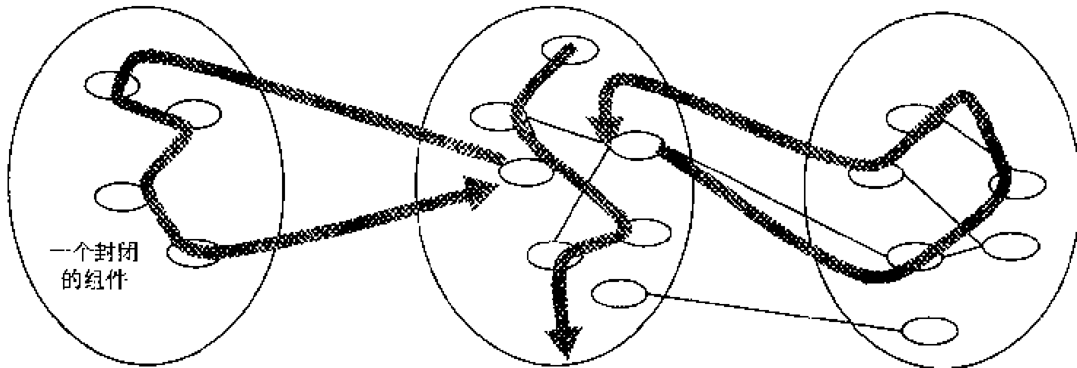
安全和活跃性的双重性可以在一些极端的设计方法中表现出来。一些自顶向下的设计策略使用一种纯粹的、以安全为首要目的的方法：首先确保每个类和方法的安全，之后再试图通过性能优化的工作提升活跃性；相反，自底向上的方法多在多线程系统编程中使用：它首先考虑代码的活跃性，然后再试图增加安全方面的控制，比如通过添加锁。在实际应用中，这两种极端方式都不会取得成功。自顶向下的设计很容易产生一个运行缓慢、经常出现死锁的系统；而对于自底向上的设计而言，出现未预期的安全错误的代码是常有的事。

多数实用并且高效的组件通常不是，并且不需要是绝对安全的，而有些由多个组件组成的有用服务也不是任何时候都是活跃的，理解了这一点对我们的工作会非常有帮助。其实，它们只有处在某些应用约束的特定上下文中才会正常地工作。因此，如何建立、描述、记录和理解这些上下文便成为了并发软件设计中的一个核心问题。

java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

有两个常用的方法（和一系列介于其中的选择）可以用来处理有关上下文依赖的问题：（1）通过封闭部分系统来减少不确定性；（2）建立一系列机制和协议允许组件具有开放性或是维持这种开放性。许多实际的设计工作在这两种方法的基础上做了多种努力。

1.3.4.1 封闭子系统



一个理想的封闭系统是一个你可以获得其所有行为的、完整静态（设计时）信息的系统。通常情况下，这是难以实现的，并且是没有必要的。但是，通过使用可能的极端面向对象的封装技术，封闭部分系统通常还是可以做到的，封闭的单位可以从单独的类到产品级的组件：

限制外部通信 ***(Restricted external communication)***。所有的交互，不论是向内还是向外的交互操作，都必须经过一个狭窄的接口。较简单的方法是，这个子系统是***通信封闭的*** ***(communication-closed)***，即永远不会从内部发起一个向子系统的外部对象的方法调用。

可确定的内部结构 ***(Deterministic internal structure)***。组成一个子系统的所有对象和线程的具体特性（理想情况下是这些构件的数量）是静态可知的。通过使用 `final` 和 `private` 关键字可以进一步加强这种特性。

至少在某些这类系统中，你可以从理论上证明——使用非正式的、正式的、甚至机制上的方法——在一个封闭组件中可以保证没有***内部***安全或是活跃性冲突问题出现。或者，即便这些问题出现了，你也可以继续通过修改设计和实现方面的工作进一步加强和完善你的组件，直到这些问题消失。在最好的情况下，你可以综合应用这些知识来进一步分析系统中依赖这个组件的其他部分的相关情况。

有关对象、线程和交互行为、完整的静态信息不仅可以告诉你在一个程序的运行过程中会有什么事情发生，还可以告诉你什么不会发生。比如，即使存在两个对象中的两个同步方法互相调用的情况，只要它们不会同时被这个子系统的不同线程调用，那么就不会发生死锁现象。

同样，封闭性可以为手工或是编译器驱动的优化提供进一步的可能性：比如从方法定义中删去不必要的同步修饰符，或是在消除了不希望的交互可能性后使用先进的特殊目的的算法。嵌入式系统同样由许多封闭模块组成，部分原因是为了提高可预测性（predictability）、调度性和进行相关的性能分析。

虽然封闭的系统非常容易处理，但它们也非常脆弱。一旦用来管理它们内部结构的相关

java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

约束和假设条件被改变，这些组件就将变得毫无用处，而必须重新开发。

1.3.4.2 开放系统

一个理想的开放系统可以允许系统在多个方向上进行无限的扩展。它可以动态加载未知的类，允许子类覆盖几乎任何方法，在不同的子系统的对象间使用回调（*进行通信*），允许在多个线程之间共享通用的资源，可以使用反射机制来发现和调用属于其他未知对象的方法等。如同完全封闭的系统一样，无限的开放性通常也是无法实现的，并且是没有必要的。如果系统中的任何事情都可以改变，你便无法编写任何代码。然而大多数系统都需要具有一定的灵活性。

想要对一个开放系统进行完整的静态分析几乎是不可能的，因为它们的特性和结构会随着时间而变化。因此，开放系统必须依赖每一个组件所对应的、有文档记录的***策略***和***协议***。

Internet 是一个开放系统的最好的范例。它可以不断地进化，比如通过添加新的主机、网页和服务，并且只需要其中的参与者互相遵守某些网络机制和协议便可以互相协作。同其他开放系统一样，很好地遵守这些 Internet 策略和协议总是比较困难的。然而，JVM 本身可以通过某种机制来确保那些不遵守规则的组件不会对系统的完整性造成灾难性的毁坏。

策略驱动的设计比较适合应用在一些较小的典型并发系统上，在这些系统中，策略和协议通常作为设计规则进行定义。有关策略域（policy domain）的一些例子会在接下来的章节中进行更深入的讨论：

程序流程（Flow）。比如一个如下形式的规则：类型 A 的组件可以向类型 B 的组件发送消息，但是反之不行。

阻塞（Blocking）。比如一个如下形式的规则：如果资源 R 无法获得的话，那么类型 A 的方法通常会立刻抛出异常，而不是阻塞直到等到这个资源可用。

通知（Notification）。比如一个如下形式的规则：在任何时刻，如果类型 A 的对象被更新，则会向它的监听器发送有关更新的通知。

通过尽量减少不一致问题发生的可能性，使用相对少量的策略可以简化设计。组件开发者也许可以借助代码审核（code review）的方法和相关工具的帮助，只需要检查这些代码是否遵守相关的设计规则即可，因而能够将主要的精力集中在手边的任务上。这样，开发者就可以在着眼于局部的同时，还能够纵观全局。

然而，随着系统中的协议数目的不断增多，并且因这些协议引入的***编程职责***使得开发者不堪重负时，策略驱动的设计就会变得无法控制。一旦为了遵守设计策略的要求，即便是类似更新一个账户结余或打印“Hello, world”这样简单的方法，也需要编写许多行笨拙且容易出错的代码的时候，我们便不得不考虑实施一些补救工作了：简化或是减少规则的数目；或是开发一些工具来自动生成代码并且/或者自动检查这些代码是否遵循规则；或是创建与特殊领域相关的语言来加强某个规则；或是通过创建一些框架或是工具库来减少每个方法中所需的相关支持代码的数量。

对于策略的选择倒是不需要为了考虑系统的效率而过多地进行“优化”，但是在开发中这些机制必须被完全地遵守，并且越严格越好。这些机制的选择原则组成了本书所描述的几

java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

个框架和设计模式的基础。有可能其中的某些框架或是设计模式会无法在你的项目中使用，并且很多现象和结果很可能使你变得非常迷惑（“我从没有做过**那个**”），因为这些框架和设计模式的底层策略很可能与你所使用的其他策略产生冲突。

虽然使用封闭的系统使你能够对系统性能进行优化，但是使用开放的系统使你能够为将来的变更进行优化。这两种调整和重构的方法通常在软件过程中具有同等难度，但是却有着相反的结果。为了性能目的而优化通常需要使用特殊的方式进行“硬编码”式的设计决策。为了扩展性目的而优化则需要将类似的“硬编码”的设计需求从系统中删除，相反，应该允许改变这些需求，比如通过将它们定义为可覆盖的方法，通过使用回调钩子（callback hook），或将相关功能抽象为接口，从而可以通过动态加载组件的方法来使用完全不同的方法重新实现这些接口。

因为并发程序通常比串行程序包含更多细小的策略，并且它们更多地依赖于有关的特定表现形式的不变约束，因此为了使得包含了并发构件的类能够被更容易地扩展，它们通常需要更多的特殊关照。这种现象已经非常地常见，被称为***继承变异*** ***(the inheritance anomaly)***，相关内容会在 § 3.3.3.3 中进行详细讨论。

但是，也有其他一些编程技术在提升性能的同时并不会对可扩展性进行限制。随着编译器和 JVM 技术的进步，上面提到的那些策略将会变得不再那么可靠。比如，动态编译可以使得多个可扩展的组件好像是在非常接近的时间内被加载，因此，在特定的运行上下文中可以进行更有效的优化和特殊化，这种效果是任何程序员都无法通过编程技术能够达到的。

1.3.4.3　文档

一旦系统的组成依赖于环境的上下文，对于与组件相关的上下文用法和约束进行很好地理解和记录是非常重要的。然而，如果没有提供这些信息，那么使用、重用、维护、测试、配置管理、系统增值和与之相关的软件工程的问题会变得十分困难。

有很多用户可以使用文档来帮助他们加强对系统的理解，这类用户包括将类作为黑盒组件使用的开发者，子类开发者，维护、编辑或是修复代码的人员，测试人员和审核代码的人员，以及系统用户。面对这些使用者，我们的首要目的是尽量通过减少非预期的情况来减少对于大量文档的需求，从而减少概念的复杂性：

标准化 ***(Standardization)***。使用通用的机制、协议和接口来书写文档。比如：

- 通过使用标准的设计模式和参考书、网页或设计文档来更完整地描述相关内容。
- 使用标准的工具库和框架。
- 使用标准的编码习惯和命名约定。
- 使用列举通常错误的标准审核列表，使得文档更清晰。

清晰化 ***(Clarity)***。使用最简单的、自证明的编码方式，比如：

- 使用异常机制来声明需要检查的条件。
- 通过访问修饰符（比如 `private`）来声明内部约束。
- 采用通用的默认程序命名和方法签名约定，比如：在没有特殊需要的情况下，一个会被阻塞的方法会在其方法定义中抛出一个 InterruptedException 异常。

java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

***辅助代码*（*Auxiliary code*）**。使用一些代码，用来示范系统的用法。比如：

- 添加示例或是推荐的用法实例。
- 提供记录了那些会造成非显见效果的代码片段。
- 包括一些用来作为自测试之用的方法。

在通过使用这些方法来避免对一些显而易见的内容进行说明之外，文档更主要是用来阐明设计***目的***。最关键的细节内容可以使用一种系统的样式描述，比如使用下面的表格中列出的那些半形式化的注释。这些注释会在本书中一直出现，并且在需要的时候会给予详细的解释。

PRE	前提条件（不必经过检查）。 `/** PRE: Caller holds synch lock...`
WHEN	守护条件（通常需要被检查）。 `/** WHEN: not empty return oldest...`
POST	结束条件（通常不会被检查）。 `/** POST: Resource r is released...`
OUT	保证消息被发送（比如，通过回调）。 `/** OUT: c.process(buff) called after read...`
RELY	其他对象和方法所需要的属性（通常不被检查）。 `/** RELY: Must be awakened by x.signal()...`
INV	在每个公共方法的起始处和结束处都需要保持为真。 `/** INV: x,y are valid screen coordinates...`
INIT	在对象的构造时刻，必须被满足的约束。 `/** INIT: bufferCapacity greater than zero...`

另外，可以使用较少结构化的文档来说明那些不明显的约束、上下文的限制、假设和设计目的，这些内容会影响系统在一个并发环境中的使用情况。在这里不可能给出该类文档的完整列表，但是通常包括如下几种：

- 有关状态和方法约束的高层设计信息。
- 在需要锁的情况下，由于缺少锁而会引起的、已知的安全局限性。
- 一个方法由于等待一个条件、事件或资源，而被不确定地阻塞。
- 某些特殊的方法，只能被其他方法调用，也许是被其他类的方法调用。

同其他的书一样，本书并不可以作为一个有关这类文档实践的特别好的榜样，因为大多数此类问题都是使用文字描述而不是使用代码文档示例。

1.3.5 进阶阅读

有关高层面向对象软件的分析和设计的说明，至少包括了一些相关的并发问题的论述：

Atkinson 和 Colin.《Object-Oriented Reuse, Concurrency and Distribution》, Addision-Wesley, 1991.

Booch 和 Grady.《Object Oriented Analysis and Design》, Benjamin Cummings, 1994。

java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

Buhr, Ray J. A.和 Ronald Casselman. 《Use Case Maps for Object-Oriented Systems》, Prentice Hall, 1996。Buhr 和 Casselman 使用类似本书中使用的时序线程图来进行与用例的映射。

Cook, Steve 和 John Daniels. 《Designing Object Systems: Object-Oriented Modelling With Syntropy》, Prentice Hall,1994。

de Champeaux, Dennis, Doug Lea 和 Penelope Faure. 《Object Oriented System Development》, Addison-Wesley, 1993。

D'Souza, Desmond 和 Alan Wills. 《Objects, Components, and Frameworks with UML》, Addison-Wesley,1999。

Reenskaug 和 Trygve. 《Working with Objects》, Prentice Hall, 1995。

Rumbaugh, James, Michael Blaha, William Premerlani, Frederick Eddy 和 William Lorensen. 《Object-Oriented Modeling and Design》, Prentice Hall, 1991。

有关并发软件的规范、分析、设计和验证的内容：

Apt, Krzysztof 和 Ernst-Rudiger Olderog. 《Verification of Sequential and Concurrent Programs》, Springer-Verlag, 1997。

Carriero, Nicholas 和 David Gelernter. 《How to Write Parallel Programs》, MIT Press, 1990。

Chandy, K.Mani 和 Jayedev Misra. 《Parallel Program Design》, Addison-Wesley, 1989。

Jackson 和 Michael. 《Principles of Program Design》, Academic Press, 1975。

Jensen, Kurt 和 Grzegorz Rozenberg (eds.). 《High-level Petri Nets: Theory and Application》, Springer-Verlag, 1991。

Lamport 和 Leslie. 《The Temporal Logic of Actions》, SRC Research Report 79, Digital Equipment Corp, 1991。

Leveson 和 Nancy. 《Sofeware: System Safety and Computers》, Addison-Wesley, 1995。

Manna, Zohar 和 Amir Pneuli. 《The Temporal Logic of Reactive and Concurrent Systems》, Springer-Verlag, 1991。

许多软件开发的特定领域都非常依赖于并发机制。比如，许多模拟系统、电信系统和多媒体系统都是高度多线程化的。由于类似系统设计的绝大多数基础内容都是由这些基本的并发技术组成，因此本书不再对大规模的软件构架和与特定的并发程序相关的特殊编程技术进行过多讨论。可以参考下面的资料可获得相关的知识：

Fishwick, Paul. 《Simulation Model Design and Execution》, Prentice Hall, 1995。

Gibbs, Simon 和 Dennis Tsichritzis. 《Multimedia Programming》, Addison-Wesley, 1994。

Watkins 和 Kevin. 《Discrete Event Simulation in C》, McGraw-Hill, 1993。

技术问题只是并发软件开发的一部分，还包括有关测试、组织、管理、人力因素、维护、工具和软件工程准则等问题。以下是一些有关基础的软件工程方法的介绍，这些方法可以同时适用于每日的编程工作和大型工程：

Humphrey, Watts. 《A Discipline for Software Engineering》, Addison-Wesley, 1995。

下面是一种完全不同的软件工程观点：

Beck, Kent. 《Extreme Programming Explained: Embrace Change》, Addison-Wesley,

java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

1999。

Jain, Raj.《The Art of Computer Systems Performance Analysis》, Wiley, 1991。

Wegner, Peter. “Why Interacton Is More Powerful Than Algorithms”,《Communications of the ACM》, May 1997。

1.4 Before/After 模式

许多并发设计都能够很好地使用模式来描述。一个模式封装了一个成功和通用的设计形式，通常是由一个或多个接口、类和/或者遵循特定的静态和动态约束，以及关系的对象组成的***对象结构***［也常被称作***微体系（micro-architecture）***］。使用模式来描述有关的设计和技术，可以使得这些内容不必在各自不同的上下文中被以完全相同的方式重复实现，并且这些内容通常无法很好地被封装在一个可重用的组件当中。对于上述情况而言，模式是一个十分理想的工具。可重用的组件和框架在并发软件开发中扮演着重要的角色。但是也有很多并发面向对象编程享受着这种可重复的设计方式所带来的种种好处，包括重用、适应性和扩展性，而不是重用特定的某些类。

与那本由 Gamma、Helm、Johnson 和 Vlissides（见进阶阅读 § 1.4.5）四人撰写的先驱之书《Design Patterns》中讨论的设计模式不同，本书中讨论的模式与所在章节的内容紧密相关，这些章节讨论了一系列与这些模式相关的上下文和软件的设计原则，这些上下文和原则说明了这个模式的主要设计目的和相关约束。本书的许多模式都是通常的面向对象分层和组合模式的小型扩展或不同变体。本节会对一些后续章节常用的相关内容进行一些介绍。其他的内容会在首次遇到时进行简要的描述。

1.4.1 分层

在各种类型的系统中，使用分层（layering）策略来控制程序的行为机制是一种常见的结构化法则。许多面向对象的分层和组合技术都使用一种类似“三明治”的方式，即将一些方法调用和代码包含在一个 ***before*** 操作和一个 ***after*** 操作之间。所有形式的 before/after 控制方法都包括了一个***基础（ground）***方法，我们在这里称之为 method，并且通常将其放在如下的调用序列中：

```
before(); method(); after();
```

如果这个基础方法会抛出异常，那么为了确保 ***after*** 操作的执行也可以使用如下形式：

```
before();
try { method(); }
finally { after(); }
```

当然，绝大多数本书中的例子都与并发控制相关。比如，一个 syncronized 方法在被调用***之前***需要先获得一把锁，并且在方法执行结束***之后***释放这把锁。但是可以结合另一个十分有用的面向对象编程实践——自检查代码（self-checking code），来阐明这个 before/after

java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

模式的基本的思想：任何一个对象的成员变量都应该在这个对象没有处在被公共方法访问的时候保持不变（参见§1.3.1）。即使这些方法抛出已声明的异常，不变约束也应该得以确保，除非这些异常是由于崩溃或程序的错误而引起的（比如 RuntimeExceptions 和 Errors）。

通过在类的每一个公共方法的入口和出口同时对不变约束进行检查，可以动态地检查程序是否遵循了这些可计算的不变约束。类似的技术也可以用在对前提条件和结束条件的检查上，但是为了简单起见，我们在这里只讨论不变约束。

举例而言，假设我们想要创建一个水槽类，并且在其中包含了对容量是否处在零和最大容积之间这个不变约束的自我检查代码。为了实现这个目的，我们可以定义一个 checkVolumeInvariant 方法并且把它作为 ***before*** 和 ***after*** 操作。我们可以首先定义一个异常，并且在不变约束检查失败的时候抛出它：

```
class AssertionError extends java.lang.Error {
  public AssertionError() { super(); }
  public AssertionError(String message) { super(message); }
}
```

如果只是在每个方法中手工插入相关的检查代码，很可能会造成程序的错误。因此，使用三种 before/after 设计模式中的任何一种便可以将这些检查代码与基础方法分隔开来：它们是适配器类（adapter class），基于子类的设计和方法适配器类（method adapter class）。

所有的方法中，最好的方式就是将基础功能放在一个接口中进行定义。当你希望为实现的改变留出足够的余地时，接口通常是非常有用的。相反，当我们在以后应用 before/after 模式时，缺少已有接口的定义通常会对我们的选择造成某种局限性。

在以下关于接口的例子中，我们对于在§1.2.4 中讨论的水槽类进行了一些微小的修改。before/after 技术可以用来对 transferWater 操作进行有关不变约束的检查。

```
interface Tank {
  float getCapacity();
  float getVolume();
  void  transferWater(float amount)
          throws OverflowException, UnderflowException;
}
```

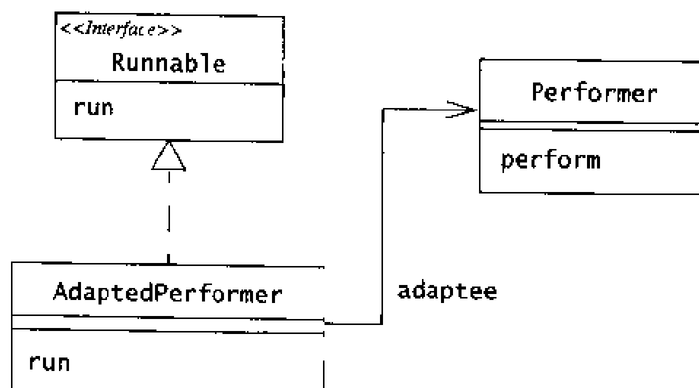
1.4.2 适配器（adapter）

当在设计了一个或多个具体的类之后再进行有关标准接口的定义工作时，这些类很可能没有很好的实现这个接口。比如，这些类中的某些方法的名字很可能与那些在接口中定义的方法有许多不同。如果你无法通过修改这些具体类来改正这些问题的话，则可以通过创建一个适配器类来消除这种不兼容性。

假设你拥有一个 Performer 类，该类包括了一个名为 perform 的方法，并且满足了作为一个 Runnable 对象运行所需要的所有条件（除了方法名的差别之外），这时，你可以

java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

通过创建一个适配器使其他的类可以在一个线程中使用它。




```
class AdaptedPerformer implements Runnable {
  private final Performer adaptee;

  public AdaptedPerformer(Performer p) { adaptee = p; }
  public void run() { adaptee.perform(); }
}
```

这只是许多用来创建适配器的常用方式中的一种，这种方式也是《Design Patterns》中提到的几种相关模式的基础。一个 ***Proxy*** 是一个和它的代理有着一致接口的适配器。一个 ***Composite*** 维护了一组代理对象，这些代理对象有着同样的接口。

在这种基于代理的组合方式中，这个可以被公开访问的主体类会把对所有方法的访问转送给它所代理的一个或是多个对象，并且会将得到的返回再转交给调用者。当然，在这个过程中也许会根据代理的调用做一些微小的转换（改变方法名称、强制转换参数类型以及过滤结果等）。

仅仅通过将这个被代理的调用包装在这个控制操作中，适配器便可以用来提供 before/after 控制。比如，假设我们拥有一个实现类，称为 `TankImpl`，则可以实现如下的 `AdaptedTank` 类，并且把原先类的构造函数：

```
new TankImpl (...)
```

替换为：

```
new AdaptedTank(new TankImpl(...))
```

这样，这个类就可以代替原来的类在应用程序中使用了。

```
class AdaptedTank implements Tank {
  protected final Tank delegate;

  public AdaptedTank(Tank t) { delegate = t; }

  public float getCapacity() { return delegate.getCapacity(); }

  public float getVolume() { return delegate.getVolume(); }

  protected void checkVolumeInvariant() throws AssertionError {
```


java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

```
      float v = getVolume();
      float c = getCapacity();
      if ( !(v >= 0.0 && v <= c) )
        throw new AssertionError();
    }

    public synchronized void transferWater(float amount)
     throws OverflowException, UnderflowException {

      checkVolumeInvariant();  // before-check

      try {
        delegate.transferWater(amount);
      }

      // The re-throws will be postponed until after-check
      //   in the finally clause

      catch (OverflowException ex)  { throw ex; }
      catch (UnderflowException ex) { throw ex; }

      finally {
        checkVolumeInvariant(); // after-check
      }
    }
  }
```

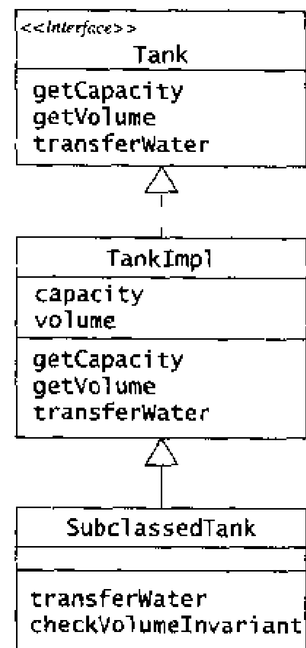
1.4.3 子类化

在通常情况下，当我们所使用的插入了 before/after 的方法与其基础版本的方法的名字和用法相同时，相比于适配器方法而言，使用子类化方法更为简单。某个方法的子类化版本可以在调用它的超类方法的前后，插入一些进行相应的检查代码。比如：

```
class SubclassedTank extends TankImpl {

  protected void checkVolumeInvariant() throws AssertionError {
    // ... identical to AdaptedTank version ...
  }

  public synchronized void transferWater(float amount)
   throws OverflowException, UnderflowException {
    // identical to AdaptedTank version except for inner call:

    // ...
    try {
      super.transferWater(amount);
    }
    // ...
  }
}
```


java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

有时选择子类化方法或是适配器方法只是习惯而已，但另外一些时候这种选择也体现了代理和继承这两种方式的差异。



适配器允许我们不必遵守与子类化相关的规则。比如，你无法通过将一个 `public` 方法覆盖为一个 `private` 方法来阻止对这个方法的访问，但是在一个适配器中，你只要轻而易举地删去这个方法即可。不同形式的代理甚至可以作为子类化技术的一种替代形式使用，让每一个“子”类（适配器：Adapter）拥有一个指向其“父”类（被适配者：Adaptee）的引用，并且将这个子类“继承”的所有方法的调用都转交给它的“父”类处理。通常，这些适配器和那些被适配者拥有同样的接口，在这种情况下它们可以被看作是一种简单的 ***Proxy*** 模式。并且代理方法比子类化方法更为灵活，因为这些“子类”对象甚至可以动态地改变它们的“父”类（通过重置这个代理引用）。

通过代理方法同样可以满足多继承的需要。比如，如果一个类必须实现两个互不相关的接口，如 `Tank` 和 `java.awt.event.ActionListener`，并且已有两个已知的超类可以提供这些所需的功能，我们就可以继承其中的一个类，并且把另一个类作为我们的被代理者实现。

但是在某些方面，代理方法却不如子类方法功能强大。比如，在基于代理的“子类”实现中，对于那些“父类方法”的调用并不会自动绑定到它所对应的那些“覆盖的”方法上。同样，基于适配器的设计有时会由于适配器对象和被适配者对象的类型不同而遇到麻烦。比如，在比较两个对象的类型是否相同时必须加倍的小心，因为如果在需要一个被适配者类型的对象时使用了一个适配器对象的话，便会导致程序错误的产生，反之也是一样。

大多数这些问题可以通过使用一些极端的措施加以避免，如在适配器类中对所有被适配器者的类中的方法进行定义，维护一个类似“`self`”的引用指向这个被适配者对象，并且在所有需要使用 `this` 的时候用它代替，甚至是在自我调用和进行有身份性检查（比如，通过覆盖 `Object.equals` 方法）的时候。有人使用专门的术语 ***delegation*** 来描述用这种方式实现的对象和类，而不是转发技术（forwarding technique）——这是一个常用来实现简单适配器的方法。

1.4.3.1 模板方法

一旦你确信需要在一系列互相关联的类中使用 before/after 控制的话，就可以创建一个抽象类，并且使用***模板方法*** 模式使得这个抽象类可以自动地实现这个控制序列（这里的模板与 C++的范型是两个不同的概念）。

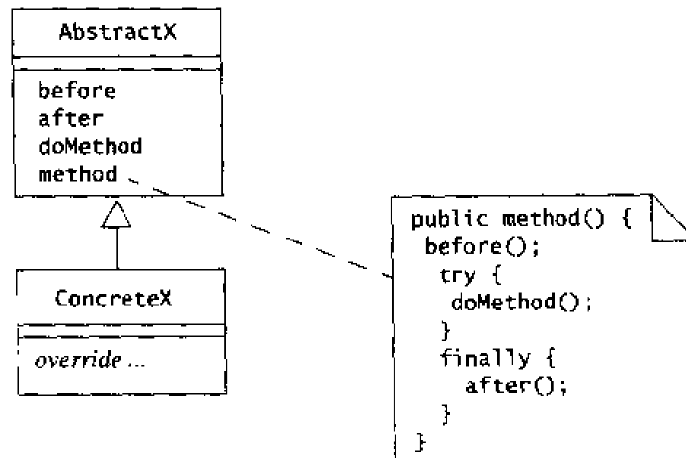
一个支持模板方法的抽象类定义了一种框架，它可以使得构造那些覆盖了基础方法、before/after 操作，或者同时覆盖这两者的子类更加容易。

- 把基本的基础操作代码定义在非公共方法中（根据约定，我们一律把 method 方法的这类非公共版本命名为 `doMethod`）。虽然缺少灵活性，但是当这些方法需要被

java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

子类覆盖时，不用声明为非公共方法。

- 将 before 和 after 操作也被定义为非公共方法。
- 在公共方法中，在 before 和 after 方法之间调用这个基础方法。



我们将这种方式应用在 Tank 例子上：

```
abstract class AbstractTank implements Tank {
  protected void checkVolumeInvariant() throws AssertionError {
    // ... identical to AdaptedTank version ...
  }

  protected abstract void doTransferWater(float amount)
   throws OverflowException, UnderflowException;

  public synchronized void transferWater(float amount)
   throws OverflowException, UnderflowException {
    // identical to AdaptedTank version except for inner call:

    // ...
    try {
      doTransferWater(amount);
    }
    // ...
  }
}

class ConcreteTank extends AbstractTank {
  protected final float capacity;
  protected float volume;
  // ...
  public float getVolume() { return volume; }
  public float getCapacity() { return capacity; }

  protected void doTransferWater(float amount)
   throws OverflowException, UnderflowException {
    // ... implementation code ...
  }
}
```


java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

1.4.4 方法适配器

用来实现 before/after 控制的最灵活、有时也是最笨拙的方法是定义一个类，这个类的全部目的只是为了调用特定对象上的某个特殊的方法。在 ***Command Object*** 模式和它的许多变种中，这些类的实例被作为参数传递、处理并且最终被执行（在我们的讨论中，是夹在 before/after 操作当中）。

由于静态类型规则的限制，每种需要被包装的方法都一定会有一个不同的适配器类。为了避免产生大量的这些类型，许多应用程序都将注意力集中在一个或是少量接口上，每个接口都定义了一个方法。比如，线程类和大多数其他执行框架中都只接受一个 `Runnable` 接口的实例，用来调用这些实例中所定义的那个无参、没有返回值并且不抛出异常的 `run` 方法。同样在 § 4.3.3.1 中，我们定义和使用了一个只包含一个 `call` 方法的 `Callable` 接口，这个方法使用一个 `Object` 对象作为参数，返回一个 `Object` 对象，并且可以抛出任何异常。

在更加特殊的应用程序中，你可以定义任何合适的带有单个方法的接口，并生成一个实现类（通常都是使用匿名内部类的方式），然后将它作为参数传递，以便可以保证这个类最终能够被调用。这种技术被广泛地应用于 `java.awt` 和 `javax.swing` 包中，其中定义了许多与不同类型的事件处理方法有关的接口和抽象类［在其他一些语言当中，使用***方法指针*** ***(function pointer)*** 和***闭合*** ***(closure)*** 来达到同样的目的］。

在这里，我们可以基于方法适配器来实现一个 before/after 的分层处理，首先我们定义一个 `TankOp` 接口。

```
interface TankOp {
  void op() throws OverflowException, UnderflowException;
}
```

在接下来的示例代码中，对于所有方法适配器的使用都只局限在 `TankWithMethod-Adapter` 类的内部。在这个简单的例子中，只有一个可供包装的方法。但是，同样的框架可以在这个类及其子类中定义的其他 `Tank` 方法中使用。方法适配器在那些对象实例使用之前需要被注册，并且（或者）这些实例经常在被执行之前会在多个对象之间传递，它可以用来处理额外的设置开销和编程职责。

```
class TankWithMethodAdapter {
  // ...
  protected void checkVolumeInvariant() throws AssertionError {
    // ... identical to AdaptedTank version ...
  }

  protected void runWithinBeforeAfterChecks(TankOp cmd)
   throws OverflowException, UnderflowException {
    // identical to AdaptedTank.transferWater
    //   except for inner call:

    // ...
    try {
      cmd.op();
```


java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

```
        }
        // ...
      }

      protected void doTransferWater(float amount)
       throws OverflowException, UnderflowException {
        // ... implementation code ...
      }

      public synchronized void transferWater(final float amount)
       throws OverflowException, UnderflowException {

        runWithinBeforeAfterChecks(new TankOp() {
          public void op()
           throws OverflowException, UnderflowException {
             doTransferWater(amount);
          }
        });
      }
    }
```

方法适配器的一些应用可以通过反射机制半自动地实现。一个通用的构造器可以在一个类中寻找一个特定的 `java.lang.reflect.Method` 对象，为其设置参数，调用它，并且返回结果。然而使用这种方法缺少静态保证性，会增加额外的处理消耗，并且可能会有许多异常需要处理。所以这种方法通常只是在需要处理未知的、需要动态加载的代码时才会被使用。

如果抛开具体使用的语言来看，则有许多极端和奇特的反射技术可以利用。比如，可以通过创建和使用特殊的工具，把用来表示 before 和 after 操作的字节码嵌入到已编译的类中去，或者在加载类的时候完成这个工作。

1.4.5 进阶阅读

除了上述介绍的那些模式之外，还有许多有用的设计模式可以在并发编程中使用，但是这其中的许多并没有包括在本书中。有关模式和模式相关问题的其他书籍包括：

Buschmann, Frank, Regine Meunier, Hans Rohnert, Peter Sommerlad 和 Michael Stal.《Pattern-Oriented Software Architecture: A System of Patterns》, Wiley, 1996。

Coplien 和 James.《Advanced C++: Programming Styles and Idioms》, Addison-Wesley, 1992。

Fowler 和 Martin.《Analysis Patterns》, Addison-Wesley,1997。

Gamma, Erich, Richard Helm, Ralph Johnson 和 John Vlissides.《Design Patterns》, Addison-Wesley, 1994。（著名的“四人帮”之书。）

Rising 和 Linda.《The Patterns Handbook》, Cambridge University Press, 1998。

Shaw, Mary 和 David Garlan.《Software Architecture》, Prentice Hall, 1996。

（多人编著）《Pattern Languages of Program Design》, Addison-Wesley。本丛书包含了在一年一度的模式编程语言年会（the annual Pattern Languages of Programming，PLoP）上讨论

java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

的许多模式。

面向对象语言 Self 是少数可以直接支持一种纯粹的、基于代理的编程语言之一，它不需要处理显式的消息传递工作。相关内容可参见：

Ungar, David. "The Self Papers",《Lisp and Symbolic Computation》，1991。

使用反射方法实现 before/after 技术在 Lisp，Scheme 和 CLOS（通用 Lisp 对象系统）中十分常见。如：

Abelson, Harold 和 Gerald Sussman.《Structure and Interpretation of Computer Programs》，MIT Press, 1993。

Kiczales, Gregor, Jim des Rivieres 和 Daniel Bobrow.《The Art of the Metaobject Protocol》，MIT Press, 1993。

其他分层的同步设计模式在以下文献中有讨论：

Rito Silva, António, João Pereira 和 José Alves Marques. "Object Synchronizer"，发表在 Neil Harrison, Brian Foote 和 Hans Rohnert（eds.），《Pattern Languages of Program Design》，第四卷，Addison-Wesley, 1999。

下面的文献中描述了一种组合方法，可以用来对并发控制分层：

Holmes, David.《Synchronisation Rings: Composable Synchronisation for Concurrent Object Oriented Systems》，PhD Thesis, Macquarie University, 1999。

将多种 before/after 方法组合在一起来处理不同的功能（比如，将同步控制和持久化控制结合起来），需要设计更精细和更复杂的框架。其中一种方法是创建一个***元类（metaclass）***框架，通过类对象可以半自动地完成方法截取和方法包装的工作。下面是有关这种组合技术的一份广泛的分析和讨论资料：

Forman, Ira 和 Scott Danforth.《Putting Metaclasses to Work》，Addison-Wesley,1999。

面向方面的编程（Aspect-oriented Programming）将分层的 before/after 技术用一系列工具代替，这些工具可以把用来处理不同控制问题的代码结合起来。有关 AspectJ 语言的报告包含了许多本书的例子，但是它们都使用了同一种面向方面的方式描述。

Kiczales, Gregor, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier 和 John Irwin. "Aspect-Oriented Programming",《Proceedings of the European Conference on Object-Oriented Programming（ECOOP）》，1997。

有几种工具可以进行有关不变约束的半自动化的测试工作。比如：

Beck, Kent 和 Erich Gamma. "Test Infected: Programmers Love Writing Tests",《The Java Report》，July 1998。

java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

第 2 章

独　　占

在一个安全的系统中，每一个对象都会保持自己的完整性。想做到这一点有时候也需要其他对象及其方法的配合。

独占技术是用来保证对象稳定不变，并且避免即使是瞬间的状态冲突所带来的影响。编程技术和设计模式是通过避免多线程并发地修改或影响一个对象表现形式来达到独占目的的。所有的方法都基于下面三个基本策略：

- 通过确保所有的方法从不同时修改一个对象表现形式，即对象永远不会进入不一致的状态来消除部分或者所有的独占控制的需要。
- 通过加锁和相关的机制来动态保证，一个对象在同一时刻只能被一个线程访问。
- 通过隐藏或者限制对象的使用权限，来结构性地保证只能有一个线程（或者每个时刻一个线程）可以使用该对象。

本章的前三节围绕着这三个基本策略描述其基本特性和使用模式——不变性（参见§2.1）、同步（参见§2.2）及限制（参见§2.3）。§2.4 讨论用这几种方法的组合来增强安全机制、活跃性、性能和一些语义上的保证。§2.5 介绍如何使用工具类来达到使用内置结构无法达到的效果。另外，第 3 章描述的某些类、技术和工具也可以实现独占技术（特别参见§3.3.2）。

串行编程和并发编程的主要区别在于是否***强制***使用了这些方法。为了保证并发系统的安全性，程序员必须确保被多线程访问的所有对象，或者是不变的，或者是通过同步机制控制的，此外还要保证没有其他的对象在其属主域以外被并发访问。实际上，提供这些保障的技术从某种意义上说只不过是一些面向对象技术的扩展，但并发编程对错误更加敏感。

正如在§1.3.1 中说过的，这些事情从本质上来讲不能由编译器和运行时的系统来保证。分析和测试工具可以发现部分错误，但是每一个类、组件、子系统、应用程序和整个系统的安全机制主要靠开发者来保证。另外，和独占相关的一些策略和设计原则也必须被明确和广泛地采纳。

如果使用本来不是在多线程环境下使用的代码，则一定要格外小心。java.*中的很多类在设计的时候都是线程安全的（对于那些例外情况，当其在本书中出现时会进行注释，其他限制参见类 API 文档）。无论如何，在构建多线程应用程序的时候都应该重新组织、重新包装（见§2.3.3.1）那些本来应用于单线程环境中的类和包。

java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

2.1 不变性

如果一个对象的状态不能改变，那么它永远不会遇到由于多个操作以不同的方式改变其状态而导致的冲突和不一致现象。

当前对象永远不会被改变，在执行中只是不断地创建新的对象，用这种方法保证不变性很容易理解。不幸的是，这种方法一般不能控制界面的交互及线程之间的协作等功能。尽管如此，选择性地使用不变性是并发面向对象编程中的一个基本方法。

具有不变性的最简单的对象，是对象中根本没有数据。因此，它们的方法都是没有状态的——也就是说这些方法不依赖于任何对象的任何数据。例如：下面的 StateLessAdder 类和它的 add 方法显然总是安全和有效的。

```
class StatelessAdder {
  public int add(int a, int b) { return a + b; }
}
```

同样的安全和活跃性在只具有 final 数据的类中也适用。这样类的实例不会面临底层的读-写冲突和写-写冲突（见 § 1.3.1），这是因为其值不会被改写。并且，只要它们的初始值是以一种一致的、合法的方式创建的，那么这些对象在更高的层面上也不会出现不变性方面的错误，例如：

```
class ImmutableAdder {
  private final int offset;

  public ImmutableAdder(int a) { offset = a; }

  public int addOffset(int b) { return offset + b; }
}
```

2.1.1 应用程序

创建一些包含比 ImmutableAdder 中更有意义的结构和功能的不变对象是没问题的。应用程序提供了抽象数据类型（Adstract Data Type，ADT）、数据容器和共享状态的表示法。

2.1.1.1 抽象数据类型（ADT）

不变对象可以作为表示数值的简单抽象数据类型的实例。一些应用较为普遍的不变对象已经定义在了 java.*包中。其中包括 java.awt.Color、java.lang.Integer、java.math.BigDecimal、java.lang.String 及其他一些[1]。定义自己的 ADT 类很容易，例如：Fraction、Interval 和 Complex-Float 等。这种类的实例永远不会改变其数据的值，但却有可能提供创建表示新值的对象的方法，例如：

1 注意：在 java.*包中的一些抽象数据类是不可变的，例如：java.awt.Point。

java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

```
class Fraction {                                          // Fragments
  protected final long numerator;
  protected final long denominator;

  public Fraction(long num, long den) {
    // normalize:
    boolean sameSign = (num >= 0) == (den >= 0);
    long n = (num >= 0)? num : -num;
    long d = (den >= 0)? den : -den;
    long g = gcd(n, d);
    numerator = (sameSign)? n / g : -n / g;
    denominator = d / g;
  }

  static long gcd(long a, long b) {
    // ... compute greatest common divisor ...
  }

  public Fraction plus(Fraction f) {
    return new Fraction(numerator * f.denominator +
                        f.numerator * denominator,
                      denominator * f.denominator);
  }

  public boolean equals(Object other) { // override default
    if (! (other instanceof Fraction) ) return false;
    Fraction f = (Fraction)(other);
    return numerator * f.denominator ==
           denominator * f.numerator;
  }

  public int hashCode() {                    // override default
    return (int) (numerator ^ denominator);
  }
}
```

代表不变数据抽象的类的实例，如果其仅仅作为封装数据来使用，那么这些实例的标识就不重要了。例如：如果两个 java.awt.Color 对象都代表黑色（通过设置 RGB 为 0），那么它们就被认为是相等的。这就是为什么 ADT 型的类都要通过覆盖 Object.equals()和 Object.ashcode()方法来反映两个抽象数据是否相等，Fraction 类就是一个说明。这些方法的默认实现，是判断这些包含数据的对象的标识是否相同。通过覆盖 Object.equals()以屏蔽标识，使得多个 ADT 对象可以表示相同的数据，或者/并且实现相同的功能。客户不需知道，或不用关心某时刻到底使用的是其中的哪个 ADT 对象。

程序员也不必在这个程序中始终秉承 ADT 的不变性表示。有时候定义不同的类以支持不同的功能，为一些概念创造不变版本和可更新版本是很有帮助的。例如：java.lang.String 是不变类，而 java.lang.StringBuffer 却是可变的，其改变要依赖于同步方法。

2.1.1.2　数据容器

如果一次性建立一个对象的状态，并且永远使用这个状态，那么使用不变对象就很方便并且很必要。例如：一个不变的 ProgramConfiguration 对象可以反映一个应用程序整个运行

java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

期间的所有配置信息。

另外，当偶尔使用通过部分拷贝的方法来创建新对象的变量、版本及状态的时候，使用不变数据容器也很有用。在这种情况下，数据拷贝的开销比起因此而避免的数据同步的代价，就变得微不足道了（见 § 2.4.4）。一个不变对象的状态改变类似于产生一个在某方面不同于原来那个对象的新对象。

2.1.1.3　共享

当程序员想要共享对象以有效地使用存储空间，并且还想保持对这些对象的高效访问时，使用不变性是一种很好的技术策略。一个不变的对象可以被任意多个对象引用，而不用担心同步或访问冲突等问题。例如：许多单独的字符（或者字母）可以共享同一个不变字体对象的引用。这是《Design Patterns》一书中的享元（Flyweight）模式的应用。很多享元设计就是通过保证共享表示的不变性得以简单实现的。

用于并发设置中的很多工具类的实例，其内部都不可变，并且由其他多个对象共享。例如：

```
class Relay {
  protected final Server server;

  Relay(Server s) { server = s; }

  void doIt() { server.doIt(); }
}
```

尽管纯粹的不变对象是简单、方便且有效的，但是很多并发面向对象编程也使用部分不变性——部分数据是常量，或者只有调用某个方法之后才是常量，或只是在一段时间内是常量等。当使用可改变的对象来实现设计感觉非常困难时，尝试不变特性的设计是一个很有用的策略。本书中会介绍很多这样的设计，尤其是在 § 2.4 中。

2.1.2　结构

为了提高效率，所有依赖于不变性的设计原则都必须恰当地使用 `final` 关键字。另外，在初始化不变对象的时候要格外小心（参见 § 2.2.7）。尤其当一个不变对象还没有初始化之前就被其他对象使用会适得其反。要把下面这一点作为设计任何类的总体原则：

- **在构造函数执行结束之前，不能访问对象的数据！**

和串行编程相比，在并发编程中这一点更难保证。构造函数应该只执行与初始化数据相关的操作。如果一个方法依赖于对象初始化的完成，那么构造函数就不应该调用该方法。如果一个对象是在其他类可存取的成员变量或表中创建的，那么构造函数应该避免使用该对象的引用，避免用 `this` 参数来调用其他的方法，概括地说，要避免用 this 产生的泄露错误（参见 § 2.3）。如果不注意这些问题，运行在其他线程上的其他的对象和方法就可以访问默认的初始化零值（标量数据）或者空指针（用来引用对象的成员变量），这是由 JVM 创建对象时在构造函数之前自动执行的。

有的时候，概念上不可改变的成员变量的值在构造函数中不能被完全初始化。例如：这些数据是根据某些文件逐渐创建的，或者同时创建的对象间有相互依赖关系。这时一定要确

java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

保在这些值被初始化之前，对象一定不能被访问。因而几乎总要用到同步（可以参见 §2.2.4 和 §3.4.2 的例子）。

2.2 同步

使用锁可以避免底层的存储冲突和相应高层面上的不变约束冲突。例如，下面的类：

```
class Even {                                          //  Do not use
  private int n = 0;
  public int next(){ // POST?: next is always even
    ++n;
    ++n;
    return n;
  }
}
```

如果没有锁，当两个或多个线程执行同一个 Even 对象的 next 方法时，可能会因为系统冲突而得不到预想的结果。下面的表格是一个可能的执行路线，表明只是调用 putfields 和 getfields 的编译代码来读写变量 n 可能导致的结果。

线 程 A	线 程 B
read 0	
write 1	
	read 1
	write 2
read 2	read 2
	write 3
write 3	return 3
return 3	

对于一个典型的并发程序，很有可能有两个线程同时调用 Even.next 方法，看起来并没有违背安全原则。使用 Even 这个版本的程序也许可以通过某些测试，但是最后一定会导致崩溃。这种违背安全原则的现象很难测试，却会导致灾难性的后果。这使得程序员为实现可靠的并发编程要更加小心谨慎。

把 Even.next 方法声明为 synchronized 就可以避免这种冲突的执行路线。锁可以把声明为 synchronized 的方法按照次序执行，这样，或者是线程 A 的 next 方法彻底执行完之后再执行线程 B 的 next 方法，或者相反。

2.2.1 机制

作为进一步讨论锁方面策略的基础，我们先总结一下通用机制和一些围绕 synchronized 关键字的使用注意事项。

java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

2.2.1.1　对象和锁

每一个 Object 类及其子类的实例都拥有一把锁。而 int 及 float 等基本类型都不是 Object 类。基本类型只能通过包含它们的对象被锁住。每一个单独的成员变量都不能标记为 synchronized。锁只能在使用成员变量的方法中应用。但是，可以像 § 2.2.7.4 中讨论的那样，将成员变量声明为 volatile 类型，这将影响成员变量的原子性（atomicity）、可见性和顺序性。

同样，包含基本类型元素的数组对象也是拥有锁的对象，但是它的每个基本元素却没有锁（不能把数组元素声明为 volatile）。锁住 Object 类型的数组，不会自动地锁住数组中的每一个元素。没有在一个原子操作中同时锁住多个对象的构件。

Class 的实例是 Object。下面我们会讨论和 Class 对象相关的锁可以用在以 static synchronized 声明的方法中。

2.2.1.2　同步方法和阻塞

synchronized 关键字在语法上有两种形式：作用于程序块或方法。块同步需要一个参数来表明锁住的是哪一个对象。这种方式使得任何一个方法都可以锁住任何一个对象。同步块最常用的参数就是 this。

块同步比方法同步更基础一些，下面这样的声明：

```
synchronized void f() { /* body */ }
```

和

```
void f() { synchronized(this) { /* body */ } }
```

是等价的。

synchronized 关键字不属于方法签名的一部分。所以当子类覆盖父类方法时，synchronized 修饰符不会被继承。因此接口中的方法不能被声明为 synchronized。同样地，构造函数不能被声明为 synchronized（尽管构造函数中的程序块可以被声明为 synchronized）。

子类和父类的方法使用同一个锁，但是内部类的锁和它的外部类无关。然而，一个非静态的内部类可以锁住它的外部类，就像下面这个样子：

```
synchronized(OuterClass.this) { /* body */ }.
```

2.2.1.3　申请和释放锁

锁的申请和释放是在使用 synchronized 关键字时根据内部的申请释放协议来使用的。所有的锁都是块结构。当进入 synchronized 方法或者块的时候得到锁，退出的时候释放锁，即使因异常退出也会释放锁。不会有忘了释放锁的情况发生。

锁操作基于“每线程”而不是“每调用”。如果锁是空闲的，或者某个线程已经拥有了锁，那么该线程就可以通过获取锁的操作继续处于活动状态，否则该线程将处于阻塞状态。[这里的可以多次进入（reentrant 或 recursive）使用的锁和 POSIX 的线程使用锁的原则有所不同。]一般来说，允许一个 synchronized 方法在不释放锁的情况下直接调用需要同一个锁的另一个 synchronized 方法。

synchronized 方法或者块只需要和另一个需要同一个锁的 synchronized 方法或者块遵守

java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

锁原则就可以。非同步的方法在任何时候都可以执行，即使同步方法正在执行。换句话说，synchronized 和原子操作（atomic）不是等价的，但是同步可以实现原子操作。

如果一个线程释放了锁，另一个线程就可以得到它（当然，如果这个线程还有其他的 synchronized 方法的话，得到锁的另一个线程也可以是原来的线程）。但是没有办法确定到底哪个线程在什么时候得到这个锁（这里没有**公平**可言，详见 § 3.4.1.5），也没有办法发现某个线程是否拥有某个锁。

在 § 2.2.7 中我们会讨论 synchronized 在对底层存储方面还有些副作用。

2.2.1.4 静态

锁住一个对象并不代表不可访问这个对象或者其任何父类的静态数据。可以通过 synchronized static 方法或块来实现静态数据的保护。静态同步方法使用该静态方法所在的类相关的 Class 对象拥有的锁。C 类的静态锁可以按下面的方式在实例对象内部访问：

```
synchronized(C.class) { /* body */ }
```

和每个类相关的静态锁与任何其他类的锁都没有关系，包括它的父类。如果想在子类中增加一个静态同步方法来达到保护父类的静态数据的目的是不可能的。应该用明确的块版本。

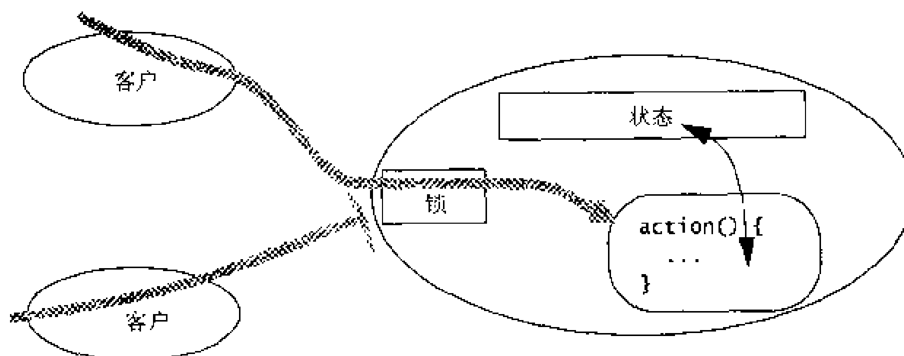
用下面这种方式也不好：

```
synchronized(getClass()) { /* body */ }        // Do not use
```

这里实际锁住的类（或子类）和声明了需要保护的静态数据的类可能根本不同。

JVM 在类装载和初始化的时候为 Class 类自动申请和释放锁。除非程序员自己编写了一个 ClassLoader 类，在静态初始化过程中拥有多个锁，否则这种内部机制不会影响作用于 Class 对象的普通方法和块的同步机制。除此之外，JVM 再没有什么行为会使用到程序员创建和使用的锁。尽管如此，当程序员创建 java.*的子类时还是要格外小心这些类中的锁原则。

2.2.2 完全同步对象



锁是最基本的信息接收控制机制。如果一个客户想要调用一个对象的方法，而另一个方法或代码块正在（另一个线程中）执行，那么锁可以阻塞这个客户。

基于锁的最安全的（但不一定是最好的）并发面向对象设计策略是，把注意力限制在***完全同步***对象（也就是原子对象）。因为在完全同步对象中：

java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

- 所有的方法都是同步的。
- 没有公共成员变量，或者其他的封装问题。
- 所有的方法都是有限的（没有无限循环，或者无休止的递归），所以最终都会释放锁。
- 所有成员变量在构造函数中已经初始化为稳定一致的状态。
- 在一个方法开始和结束的时候，对象的状态都应该稳定一致（遵守不变性），即使出现了异常情况也应如此。

例如：下面的例子是一个简化版本的 java.util.Vector：

```
class ExpandableArray {

  protected Object[] data;  // the elements
  protected int size = 0;   // the number of array slots used
  // INV: 0 <= size <= data.length

  public ExpandableArray(int cap) {
    data = new Object[cap];
  }

  public synchronized int size() {
    return size;
  }

  public synchronized Object get(int i) // subscripted access
   throws NoSuchElementException {
    if (i < 0 || i >= size )
      throw new NoSuchElementException();

    return data[i];
  }

  public synchronized void add(Object x) { // add at end
    if (size == data.length) {          // need a bigger array
      Object[] olddata = data;
      data = new Object[3 * (size + 1) / 2];
      System.arraycopy(olddata, 0, data, 0, olddata.length);
    }
    data[size++] = x;
  }

  public synchronized void removeLast()
   throws NoSuchElementException {
    if (size == 0)
      throw new NoSuchElementException();

    data[--size] = null;
  }
}
```

如果没有同步机制，这段代码在并发程序中就是不可靠的。例如：在 removeLast 操作过程中，试图处理访问方法 get 就会产生读-写冲突。如果同时执行两个 add 操作，就会出现写-写冲突，这时出现的结果就很难预料。

java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

2.2.3 遍历

在完全同步类中，可以通过把一个操作封装在 synchronized 方法中来增加新的原子操作。鉴于方便性和重用性，程序员最好把少量此类操作增加到具有普遍意义的类或其子类中。这样可以避免客户一次次地从小组件中创建具有普遍使用意义的原子操作。例如：定义 removeLast 和 prepend 的 synchronized 版本很有用，在 java.util.Vector 中的 ExpandableArray 和其他类似的集合类中的类似方法也应该这样处理。

但是这种策略对另一种集合的通用方法不起作用：***遍历***。遍历就是对集合中的每一个元素执行一些相应的操作或逐个使用。因为对集合元素的操作可能无限多，所以把集合中的每个方法都定义为 synchronized 方法是没有意义的。

对这个设计问题一般有三种解决方法：聚集操作、索引化遍历和版本化迭代变量，每种方法都有设计的利弊（更多的使用其他集合类的策略参见 § 2.4.1.3、 § 2.4.4 和 § 2.5.1.4）。这些问题和利弊在设计使用锁的类中出现得更加普遍。

2.2.3.1 同步聚合操作

一种安全使用枚举的方法就是把作用于每个元素的操作抽取出来，这样可以把它作为 synchronized applyToAll 方法的参数。例如：

```
interface Procedure {
  void apply(Object obj);
}

class ExpandableArrayWithApply extends ExpandableArray {

  public ExpandableArrayWithApply(int cap) { super(cap); }

  synchronized void applyToAll(Procedure p) {
    for (int i = 0; i < size; ++i)
      p.apply(data[i]);
  }
}
```

比如说，这可以用来打印集合 v 中的所有元素：

```
v.applyToAll(new Procedure() {
  public void apply(Object obj) {
    System.out.println(obj)
  }
} );
```

这个方法消除了在遍历过程中其他线程试图增加或减少元素可能带来的干扰，但是代价是拥有集合的锁的时间太长。这种代价有时是可以接受的，但是这种方法会引发性能和活跃性问题，处理的方法正如 § 1.1.1.1 中所说的默认规则，当调用操作方法（这里指 apply 方法）时释放锁。

java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

2.2.3.2 索引化遍历和客户端锁

对 ExpandableArray 使用的另一种遍历策略是要求客户端使用索引的访问方法来遍历，例如：

```
for (int i = 0; i < v.size(); ++i)              // Do not use
  System.out.println(v.get(i));
```

这样可以避免对每个元素操作的时候都使用锁，其代价是对每个元素都要进行两个同步操作（size 和 get）。更重要的是，为了处理由细锁粒度产生的潜在冲突问题，必须要重写循环。像 i<v.size()这样的操作可能成功，但在之后，另一个线程可能删除了当前的最后一个元素，如果这时再调用 v.get(i)可能就会出错。解决这个问题的一个办法是使用***客户端锁*** 来保证大小检查和访问的原子性。

```
for (int i = 0; true; ++i) {                   // Limited utility
  Object obj = null;
  synchronized(v) {
    if (i < v.size())
      obj = v.get(i);
    else
      break;
  }
  System.out.println(obj);
}
```

既便这样，还可能会有问题。例如：如果 ExpandableArray 类支持重新设置元素位置的方法，那么在遍历过程中，如果 v 被这样修改了，那么同样的元素就可能被打印两次。

作为一个极端的手段，客户端可以将全部的遍历包含在 synchronized（v）语句中。同理，这种方法通常是可以接受的，但是会引起讨论使用同步聚合方法时所见的长时间被锁的问题。如果对元素的操作很费时，则可以先拷贝数组用来做遍历：

```
Object[] snapshot;
synchronized(v) {
  snapshot = new Object[v.size()];
  for (int i = 0; i < snapshot.length, ++i)
    snapshot[i] = v.get(i);
}

for (int i = 0; snapshot.length; ++i) {
  System.out.println(snapshot[i]);
}
```

客户端锁在多线程非面向对象的编程中使用得很普遍，这种方式通常比较灵活。并且，当类的实例由于被放入其他（参见 § 2.4.5）而无法在内部实现同步策略时，客户端锁在面向对象的系统中就非常有用了。

在以破坏封装为代价的前提下，客户端锁可以解决潜在的冲突问题。此时正确与否完全依赖于对 ExpandableArray 的内部实现的了解程度，这个类如果被修改了，就可能无法正常工作。在一个闭合子系统中，这一点也许可以接受。如果有文档明确地指出使用这些类的条件，客户端锁就可能是一个很好的选择。当然，文档中还要规定所有未来可以进行的修改和

java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

子类中相应的支持。

2.2.3.3 版本化迭代变量

第三种遍历方法是涉及的集合类支持***失败即放弃*** (*fast-fail*) 的迭代变量，如果在遍历过程中集合元素被修改，迭代操作就会抛出一个异常。实现这种策略的最简单的方法就是维护一个迭代操作的版本号，这个版本号在每次更新集合时都会增长。每当迭代变量访问下一个元素时，都会先看一下这个版本号，如果它已经改变了，则会抛出一个异常。这个版本号应该足够大，使得在一次遍历过程中版本号不会循环。一般来讲，整型（int）就足够了。

集合框架中的 java.util.Iterator 使用的就是这种策略。我们可以把这种策略应用于 ExpandableArray 的子类，作为 after 操作来更新版本号（参见 § 1.4.3）：

```
class ExpandableArrayWithIterator extends ExpandableArray {
  protected int version = 0;

  public ExpandableArrayWithIterator(int cap) { super(cap); }

  public synchronized void removeLast()
   throws NoSuchElementException {
    super.removeLast();
    ++version;                 // advertise update
  }

  public synchronized void add(Object x) {
    super.add(x);
    ++version;
  }

  public synchronized Iterator iterator() {
    return new EAIterator();
  }

  protected class EAIterator implements Iterator {
    protected final int currentVersion;
    protected int currentIndex = 0;

    EAIterator() { currentVersion = version; }

    public Object next() {
      synchronized(ExpandableArrayWithIterator.this) {
        if (currentVersion != version)
          throw new ConcurrentModificationException();
        else if (currentIndex == size)
          throw new NoSuchElementException();
        else
          return data[currentIndex++];
      }
    }

    public boolean hasNext() {
      synchronized(ExpandableArrayWithIterator.this) {
        return (currentIndex < size);
      }
```


java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

```
      }

      public void remove() {
        // similar
      }
    }
  }
```

这里的打印循环应该这样设计：

```
for (Iterator it = v.iterator(); it.hasNext();) {
  try {
    System.out.println(it.next());
  }
  catch (NoSuchElementException ex) { /* ... fail ... */ }
  catch (ConcurrentModificationException ex) {
    /* ... fail ... */
  }
}
```

即使这样，处理失效的方法还是很有限的。ConcurrentModificationException 经常说明了在线程之间存在无计划且不希望看到的交互，这些问题需要修正代码而不是靠处理异常解决。

版本化迭代变量封装了基于数据结构的设计选择，但有时显得过于保守。例如：一个同时被调用的 add 方法本来不会和通常的遍历操作有冲突，但是在这里却会导致抛出异常。然而，对于集合类来说，版本化迭代变量还是一个比较好的选择，部分因为可以在这些迭代变量之上使用聚合遍历或客户端锁，反之却不可以。

2.2.3.4 访问者

在《Design Patterns》一书中，访问者（visitor）模式扩展了迭代遍历的含意，支持客户对按任意方式相互连接的对象集执行操作，这时，可以用某种树或图的形式来组织节点，而不是像 ExpandableArray 那样的串行列表（另外，访问者模式也支持每个节点的多态操作）。

访问者和其他一些具有广义的、遍历概念的实现策略和考虑与最简单的迭代变量的实现思路相似，或可以简化为这样的思路。例如：程序员可以首先创建一个具有节点的列表，然后使用任何一种上面提到的遍历技术来遍历该列表。然而，这里的锁只能锁住列表，而不能锁住节点本身。通常，这是最好的方法。但是程序员要想确保在遍历过程中，所有的节点都被锁住，则应该考虑容器锁（参见 § 2.4.5）和限制（参见 § 2.3.3）的方式。

相反，如果遍历通过每个节点都支持 nextNode 来实现，并且程序员不想同时占用所有节点的所有锁，那么在处理下一个节点之前，每一个节点的同步（其实指的是锁——译者注）都要被释放，这部分内容可以参见 § 2.4.1 和 § 2.5.1.4。

2.2.4 静态和单态（singleton）

正如《Design Patterns》一书中所说，单态类的目的是为了只支持一个实例。把这个类

java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

实例声明为 static，这样类和实例方法就可以使用同一个锁。

下面介绍一种完全同步的单态类的创建，这种方法可以推迟实例的创建，直到通过 instance 方法第一次调用这个类。这种类实现了给在一个应用中的不同的类中的对象、事务、消息等分配全局惟一序列号时使用的计数器（为了演示初始化过程中的计算过程，初始值是 0~2^{62} 中的一个随机正数）。

```
class LazySingletonCounter {
  private final long initial;
  private long count;

  private LazySingletonCounter() {
    initial = Math.abs(new java.util.Random().nextLong() / 2);
    count = initial;
  }

  private static LazySingletonCounter s = null;

  private static final Object classLock =
                              LazySingletonCounter.class;

  public static LazySingletonCounter instance() {
    synchronized(classLock) {
      if (s == null)
        s = new LazySingletonCounter();
      return s;
    }
  }

  public long next() {
    synchronized(classLock) { return count++; }
  }

  public void reset() {
    synchronized(classLock) { count = initial; }
  }
}
```

这里看到的锁机制（以及其他一些略微改变的变体）防止了在某些条件下由于两个不同的线程同时调用 instance 方法而创建两个实例的情况出现。在例子中只有一个实例和 s 绑定，并且在下一次调用 instance 时被返回。正如 §2.4.1 中所说，也许在某些情况下，这段代码可行；但是在多数情况下这段代码都会引起严重的错误。

避免这种错误的较简单的方法是避免使用延迟的初始化。因为 JVM 执行类的动态装载，所以没有必要使用单态类的延迟初始化。static 成员变量直到类装载之后才初始化。除了不知道类装载的准确时间（我们只知道当执行代码需要访问这个类时会加载它），与其他语言相比较，完全初始化静态变量并不会增加很多系统的启动开销。所以，除非初始化工作既耗费资源，又很少需要，否则简单的方式就是把单态数据声明为 static final 类型。例如：

java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

```
class EagerSingletonCounter {
  private final long initial;
  private long count;

  private EagerSingletonCounter() {
    initial = Math.abs(new java.util.Random().nextLong() / 2);
    count = initial;
  }

  private static final EagerSingletonCounter s =
                              new EagerSingletonCounter();

  public static EagerSingletonCounter instance() { return s; }
  public synchronized long next() { return count++; }
  public synchronized void reset() { count = initial; }
}
```

尽管很简单，但是在没有特殊原因的时候一定要依赖于实例，程序员可以定义和使用有静态方法的版本，就像：

```
class StaticCounter {
  private static final long initial =
              Math.abs(new java.util.Random().nextLong() / 2);
  private static long count = initial;
  private StaticCounter() { } // disable instance construction
  public static synchronized long next() { return count++; }
  public static synchronized void reset() { count = initial; }
}
```

有时，每个线程创建一个实例比每个程序创建一个实例更加恰当，这时用 ThreadLocal（参见 § 2.3.2）比用单态类更好。

2.2.5 死锁

尽管完全同步的原子操作很安全，但线程却因此失去了灵活性。试想：Cell 类中的一个方法和另一个 Cell 对象交换数据：

```
class Cell {                                        // Do not use
  private long value;
  synchronized long getValue() { return value; }
  synchronized void setValue(long v) { value = v; }

  synchronized void swapValue(Cell other) {
    long t = getValue();
    long v = other.getValue();
    setValue(v);
    other.setValue(t);
  }
}
```

SwapValue 是一个多方（multiparty）同步操作——也就是操作本身需要多个对象的锁。如果不小心，就会出现一个线程调用 a.swapValue(b)，另一个线程调用 b.swapValue(a)，当按

java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

照下面的路线执行时，就会导致死锁：

线　程　1	线　程　2
进入 a. swapValue(b)时获取 a 的锁	
在执行 t = getValue()时，顺利获得 a 的锁（因为已经持有）	进入 b. swapValue(a)时获取 b 的锁
执行 v = other.getValue()时，由于需要 b 的锁而处于等待状态	在执行 t = getValue()时，顺利获得 b 的锁
	执行 v = other.getValue()时，由于需要 a 的锁而处于等待状态

至此，两个线程就被永远锁住了。

更普遍地说，死锁是在两个或多个线程都有权限访问两个或多个对象，并且每个线程都在已经得到一个锁的情况下等待其他线程已经得到的锁。

2.2.6 顺序化资源

为了避免死锁或其他活跃性失败，或者从这样的情况中恢复过来，我们还需要使用本章中的其他独占技术。其中一个比较简单的技术就是顺序化资源（resource ordering）。顺序化资源可以应用在像 Cell 这样的类中，而不需要改变类的结构。

顺序化资源的思想是把每一个嵌套的 synchronized 方法或块中使用的对象和数字标签（或者其他可以排序的数据类型）关联起来。如果同步操作是根据对象标签的最小最先（least-first）的原则，那么下面的现象就永远不会发生：一个线程拥有了 x 的同步锁正等待着 y 的同步锁；而另一个线程拥有了 y 的同步锁正等待着 x 的同步锁。它们都会以同样的顺序得到锁，从而避免了发生死锁。更普遍一些，在开发设计中，为了打破对称或者强行设置优先次序，都可以使用顺序化资源的方式。

有些情况下（参见 § 2.4.5），我们会对使用的一系列锁制定特殊的顺序化原则。但有的时候，程序员也可以用常规的标签实现锁的顺序化。例如：程序员可以使用 System.identityHashCode 的返回值。即使类本身已经覆盖了 hashCode 方法，这个方法在默认情况下还是直接调用 hashCode。尽管没有什么机制可以保证 identityHashCode 方法的惟一性，但在实际的运行系统中，这个方法的惟一性在很大程度上得到了保证。为了进一步确保安全，程序员应该覆盖 hashCode 方法，或者在任何使用顺序化资源的类中使用其他的标签方法来保证惟一性。例如：程序员可以使用 § 2.2.4 中介绍的任何一个类为每一个对象分配一个串行数。

java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

作为更深层的检查方法，***别名检测（alias detection）***可以应用在使用了嵌套同步的方法中，用以处理绑定同一对象的两个或者多个引用情况。例如：在 swapValue 中，可以检查一下 Cell 是否在和自己进行交换。严格来说，这种检查方式在这里是可选的（参见 § 2.5.1）。同步锁访问是每线程而不是每调用的。对于已经获得其锁的对象再进行同步操作是可行的。但是，这种别名检测方法可以进一步预防后续功能、效率的下降和基于同步的复杂性。一般在对两个或者多个对象同步之前使用这种方法，除非两个对象是独立、毫不相关的类型（因为两个不相关类型对象的引用无论如何也不会是同一类对象，所以无需检查）。

swapValue 的一个更好的版本如下，这里既使用了顺序化资源，又使用了别名检查：

```
public void swapValue(Cell other) {
  if (other == this) // alias check
    return;
  else if (System.identityHashCode(this) <
           System.identityHashCode(other))
    this.doSwapValue(other);
  else
    other.doSwapValue(this);
}

protected synchronized void doSwapValue(Cell other) {
  // same as original public version:
  long t = getValue();
  long v = other.getValue();
  setValue(v);
  other.setValue(t);
}
```

作为提高效率的小小修改，我们可以进一步把 doSwapValue 简化，其方法是：首先得到必要的锁，然后直接访问成员变量。这样可以避免在已经获得锁的情况下再调用自己的 synchronized 方法，但是当成员变量的访问属性发生变化时，需要做相应的代码改动。

```
// slightly faster version
protected synchronized void doSwapValue(Cell other) {
  synchronized(other) {
    long t = value;
    value = other.value;
    other.value = t;
  }
}
```

注意，this 的锁是通过 synchronized 方法限定符得到的，但是 other 的锁却是显式地得到的。我们还可以把 doSwapValue 的代码合并到 swapValue 中，然后在 swapValue 中显式获取这两个锁，这样做可以有一点小小（可能根本没有）的性能改进。

获得锁的顺序问题无疑在使用嵌套同步的方法中非常突出。当一个同步方法获得一个对象的锁后，再调用另外一个对象的同步方法的时候，就会产生问题。但是在级联式调用（cascaded calls）的情况下，使用顺序化资源并没有什么优势。总的来说，对象不确定接下来哪个对象会被调用，也不知道它们是否需要同步。这就是为什么在开放系统中（参见 § 2.5），

java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

当同步调用不释放锁（参见§2.4.1）时，死锁如此难以解决的原因之一。

2.2.7 Java 存储模型

我们先看一个不用同步机制定义的小型类：

```
final class SetCheck {
  private int  a = 0;
  private long b = 0;

  void set() {
    a =  1;
    b = -1;
  }

  boolean check() {
    return ((b ==  0) ||
            (b == -1 && a == 1));
  }
}
```

在纯串行化语言里，check 方法永远不会返回 false。即使编译器、运行系统及硬件可以用任何方式（甚至你想都想不到的方式）调用代码，情况还是如此。例如，下面就是几种 set 执行的路线：

- 编译器可以重新安排语句的执行顺序，这样 b 就可以在 a 之前赋值。如果方法是内嵌的（inline），编译器还可以把其他语句重新排列。
- 处理器可以改变这些语句的机器指令的执行顺序，甚至同时执行这些语句。
- 存储系统（由于被缓存控制单元控制）也可以重新安排对应存储单元的写操作顺序。这些写操作可能与其他计算和存储操作同时发生。
- 编译器、处理器和/或存储系统都可以把这两条语句的机器指令交叉执行。例如：在一台 32 位的机器上，可能先写 b 的高位，然后再写 a，最后写 b 的低位。
- 编译器、处理器和/或存储系统都可以使对应于变量的存储单元一直保留着原来的值，以某种方式维护着相应的值（例如，在 CPU 的寄存器中）以保证代码正常运行，直到一个 check 调用才更新。

在串行编程语言中，这些都不是问题，因为程序的执行在语义上遵循着“***就像顺序执行（as-if-serial）***”的规则[2]。因为在串行编程中不必担心语句执行的具体细节，所以这段代码可以按上面的情况任意操做。这为编译器和机器提供了重要的灵活性。在过去的十年里，利用这个特点（通过管道超标量 CPU、多层缓存、装载/存储平衡，和过程间寄存器分配策略等等），计算机在执行速度上得到了突飞猛进的发展。这个特性把串行语言保护了起来，使得人们无需知道其在底层是怎么执行的。不需要创建自己线程的程序员从来不会受这些因素

2 更确切地说，*就像顺序执行*（被称为编程顺序）的语义可以被定义为对一个图的节点的遍历执行，这个图由在值或控制上有相互依赖关系的操作按顺序组成，这些操作的顺序表示了基于编程语言的基本表达式和语句在语义上的依赖关系。

java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

的影响。

但在并发编程语言中，这一切就都不同了。在这里，完全可能一个线程在调用 check 而另一个线程在执行 set，这样，check 的执行可能被优化执行的 set 打断。这时如果发生上面提到的任何一种情况，check 方法都有可能返回 false。例如，check 可以读到一个长整型的 b，这个 b 既不是 0 也不是-1，而是被写了一半、介于两者之间的值。当然，语句的乱序执行也可以导致 check 读到 b 的值是-1，而读到 a 的值是 0。

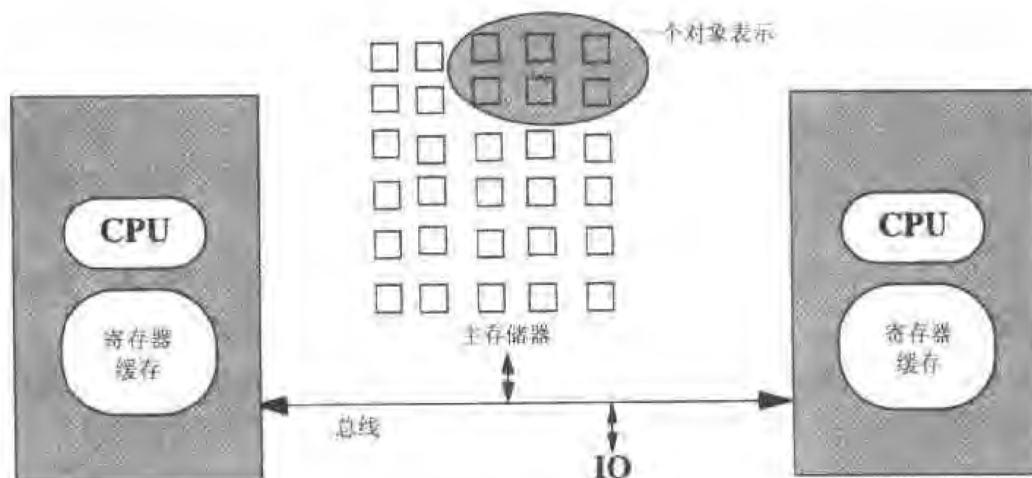
换一句话说，在并发编程中不仅可以有多条语句交叉执行，而且还可以打乱顺序执行，或者在被优化得面目全非后再执行。随着编译器和运行时技术的成熟，多处理器变得越来越普遍，而这种现象也就变得越来越普遍。但是对于习惯于串行编程的程序员来说，这会导致奇怪的结果（换句话说，几乎对所有的程序员都是这样），因为他们从来不了解串行执行代码底层的执行特性。这也许就是并发编程错误的根源吧。

几乎在所有情况下，都可以使用一种简单明显的办法来避免由于优化执行而在并发编程中引发的复杂性：***使用同步***。例如：其在 SetCheck 类中的两个方法都被声明为同步的，那么程序员就可以确定不会有具体的执行细节影响到这段代码的执行结果。

但有时，程序员不能或者不想这样做，或者需要理解别人的没有使用同步的代码。在这些情况下，程序员必须依靠 ***Java 存储模型*** 的语法规则提供的最基本的保证。这个模型允许像上面列举的那样操作，但是其潜在的影响依赖于执行语法，并且该模型要求程序员掌握用来从某个方面控制执行语法的一些技巧（在 § 2.4 中会讨论多数语法控制技巧）。

Java 存储模型是《Java Language Specification》的一部分，在 JLS 的第 17 章有详细描述。这里只简单地讨论一下该模型的动机、特性和编程理念。这里也给出了一些 JSP 第一版中没有的声明和更新[3]。

我们假设可以把该模型看成是 § 1.2.4 中描述的标准 SMP 机器的理想化模型。



为了建造这样的模型，我们假设每个线程都运行在不同的 CPU 上。当然，这种假设即使是在多处理器的情况下也很少见。这种每线程每 CPU 来实现多线程的合理性说明了模型

3 截至到本书编写时，JLS 中的存储模型和其他相关章节都正在更新，以包含 Java 2 平台。请查阅在线文档，以获得和本章相关资料的更新。

java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

的一些特殊属性。例如：由于CPU控制着其他CPU无法访问的寄存器，所以模型必须容许一个线程不知道正在被另一个线程处理的数据的值的情况。但是，模型的应用并不局限于多处理器。可以应用此模型考察在单CPU系统中多个编译器和多个处理器的行为。

模型不会明确地指定上面的种种执行策略到底是由编译器、CPU、缓存控制器，还是其他机制来执行。也不会以程序员熟悉的类、对象、方法等方式来讨论。模型只定义线程和主存的抽象关系。每一个线程都有一个工作存储空间（缓存和寄存器的抽象）用来存储数据。模型保证了与方法相关的指令顺序以及与数据相关的存储单元这两者之间的一些交互的特性。很多规则都是根据何时主存和每线程工作存储空间之间传送数据来描述的。主要围绕以下三个相关问题：

***原子性*（Atomicity）**。指令必须有不可分割的特性。为了建模的目的，规则只需要阐述对代表成员变量的存储单元的简单读写操作。这里的成员变量可以是实例对象和静态变量，还包括数组元素，但是不包括方法中的局部变量。

***可见性*（Visibility）**。在什么情况下一个线程的效果对另一个线程是可见的。这里的效果是指写入成员变量的值对于这个成员变量的读出操作是可见的。

***顺序化*（Ordering）**。在什么情况下对一个线程来说操作可以是无序的。主要的顺序化问题围绕着和读写有关的赋值语句的顺序。

如果一致地应用同步机制，那么每个属性都有一个简单的特征：在一个synchronized方法或者块中所做的改变，对使用同一个锁的另一个synchronized方法或者块来说，是原子的和可见的。并且在一个线程里，是按照程序指定的顺序来处理synchronized的方法或者块的。即使在方法或者块内执行的语句打乱了顺序，也不会影响到其他使用了同步的线程。

如果没有使用同步机制或者同步机制不一致，结果就变得很复杂。存储模型所做的保证比程序员直觉上期待的弱了很多，也比那些典型的JVM的实现所提供的保证弱。要想保证独占在实践中核心的对象一致性关系，程序员还必须额外地尽一些义务。对象必须对所有依赖于自己的线程维护不变约束，而不是只对修改该对象某个状态的线程保持不变约束。

模型的最重要规则和属性将在下面依次讨论。

2.2.7.1 原子性

存取和更新除了long和double之外的任何类型的成员变量所对应的存储单元，它们都是原子的。其中包括引用其他对象的成员变量。另外，原子性还可以扩展到volatile long和double类型（尽管非volatile的long和double类型不能保证原子性，但它们在某些实现中也可以具有这种特性）。

当在表达式中使用非long和double类型时，原子性保证程序员得到的数据或者是初始值或者是某线程修改之后的值，绝不是两个或多个线程同时修改而产生的混乱字节。但是，就像下面将要看到的，原子性本身并不能保证程序获得的值是线程最近修改的值。由于这个原因，原子性在本质上对并发程序的设计没有多大影响。

2.2.7.2 可见性

只有在下面的情况中，线程对数据的修改对于另一个线程而言才***确保***是可见的：

java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

1. 写线程释放了同步锁，而读线程获得了该同步锁。

其精髓是：释放的时候强制地把线程所使用的工作存储单元的值刷新到主存，获得锁的时候要装载（或者重新装载）可访问成员变量的值。锁只为同步方法或块中的操作提供独占，而它对存储的影响却包括了执行操作的线程使用的所有成员变量。

注意同步的双重意义：它既通过锁操作支持高层同步协议，同时也支持存储系统[有时是通过底层的存储关卡（memory barrier）机器指令实现的]，从而保证多个线程中的数据表示的同步。这说明，相对串行编程来说，分布式编程和并发编程更具相似性。synchronized 的第二个含意可以视为一种机制，它使得一个线程中的方法可以发送和/或接收在另一个线程的方法中对数据的修改信息。从这个角度看，使用锁和发消息只是语法不同而已。

2. 如果一个成员变量被声明为 volatile，那么在写线程做下一步存储操作之前，写入这个 volatile 成员变量的数据在主存中刷新，并使其对其他线程可见（也就是说，为了此目的，成员变量会被立即刷新）。读线程在每次使用 volatile 成员变量之前都要重新读入数据。

3. 如果一个线程访问一个对象的成员变量，那么线程读到的或者是初始值[4]，或者是被另一个线程修改过的值。

还有，对没有完全创建好的对象进行引用很不好（参见 § 2.1.2）。在构造函数内部启动一个线程也很危险，尤其是当一个类可能被子类化的时候。Thread.start 方法可以达到这样的存储效果：调用 start 方法的线程释放它的锁，而刚刚启动的线程再获得这个锁。如果在子类的构造函数执行之前，实现了 Runnable 的父类就调用了 new Thread(this).start()，那么当 run 方法执行时，对象很可能没有被完全初始化。同样的，如果创建并且启动了一个新的线程 T，然后又创建了由 T 使用的对象 X，那么除非对所有使用到对象 X 引用的地方都应用同步，否则无法知道 X 的成员变量对线程是否可见。或者，如果可行的话，可以在启动线程 T 之前创建 X。

4. 当一个线程结束的时候，所有的写入数据都将被刷新到主存中。

例如：一个线程用 Thread.join 方法和另一个线程的结束同步，那么第一个线程肯定能看到第二个线程的执行结果（参见 § 4.3.2）。

注意，在***同一个***线程之内的不同方法之间传递对象的引用，***永远不会***引起可见性问题。

存储模型保证：如果上面这些操作最终一定会发生，那么一个线程对一个成员变量的更新最终对另一个线程是可见的。但是，***最终***可能是一段很长的时间。很难期盼一个不使用同步的线程和另一个线程能对成员变量的值一直保持同步。尤其需要注意的是，如果成员变量不是 volatile 的或者通过同步访问的，那么千万不要企图通过执行循

4 在写作本书的时候，JLS 还没有明确地声明，对一个初始的 final 字段来说，读到的值就是在其初始方法或创建方法中分配的值。但是在本书中，我们假设有这样的声明。对于非 final 字段来说，标量的默认初始可见值是 0，引用的默认初始可见值是 null。

java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

环来等待另一个线程的写入操作（参见§3.2.6）。

在没有同步的时候，模型还支持不一致的可见性。例如：得到某个对象的一个成员变量刷新之后的值，而同时读取另一个成员变量的刷新之前的值。同样，也可能读取一个刷新后的、更新的引用变量的值，但是同时正在引用这个引用对象的一个成员变量的旧值。

模型中并不要求跨越线程的可见性失效一定发生，而仅仅是容许这样的情况发生。这就是在不使用同步的多线程中也不能保证一定会发生违反安全性的现象的原因之一，这种情况只是有可能而已。在当前的很多 JVM 的实现和操作系统上，甚至于在使用多处理器的操作平台上，都很少出现可见性失效问题。共享一个 CPU 的多个线程使用共用的缓存，缺乏强大的编译优化和强壮的、保证缓存一致性的硬件，这都使得数据看起来可以在线程之间即时传递。这样就不可能测试基于可见性的错误了，因为这样的错误很少出现，或者只有在没有测试过的平台或将来的某个操作系统下才会发生错误。这在有关多线程的安全问题方面更加普遍。如果不使用同步，并发编程会在很多方面出现问题，包括存储的一致问题。

2.2.7.3 排序

排序规则在两种情况下应用，线程内部和线程之间：

- 从线程执行方法的角度来说，指令的处理就像是以通常的串行编程语言一样的串行执行。
- 从其他线程的角度看，它们可能通过同时运行不同步的方法来“监视”本线程，那么几乎什么结果都可能发生。惟一有用的约束条件是：总是保证同步方法或者块的相关顺序，以及对 volatile 成员变量的操作。

再强调一遍，这里说的只是最小的保证。几乎在任何一个程序或者操作系统中，都会有更加严格的排序。但是却不能完全依靠这些排序，因为很快你会发现，很难在其他不同的 JVM 上测试运行失败的代码，这是因为 JVM 的属性各不相同，但又都遵从了上面的规则。

注意，从线程内部看的规则，适应所有有关 JLS 中语义的讨论。例如：算术表达式中的等式，对于本线程来说是从左向右地（JLS 中的 15.6 节）执行，但是对于其他线程来说并不一定是这样。

只有在使用同步、独占，或者纯巧合，使得在某时刻只有一个线程控制变量的时候，线程内部的看似同步的属性才有用。如果多个线程运行非同步的代码，且它们读写相同的数据，那么任何的交叉执行，原子性失效、条件竞争及可见性失效等就都会出现，这样对于任何一个线程看似顺序的概念都没有意义。

尽管 JLS 对于可能出现的某些合法与不合法的重新排序制订了规则，但上述的其他因素大大降低了规则在实践中所能提供的保证，运行结果可能反映了任何可能的重新排序的任意组合。因而没有必要给代码的排序属性解释原因。

2.2.7.4 volatile

在原子性、可见性及排序方面，把一个数据声明为 volatile 几乎等价于使用一个小的完全同步类来保护通过 get/set 方法操作的成员变量，就像：

java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

```
final class VFloat {
  private float value;

  final synchronized void  set(float f) { value = f; }
  final synchronized float get()        { return value; }
}
```

把一个数据声明为 volatile 的区别（和同步相比——译者注）只是没有使用锁。尤其是对于复合读写操作，例如对 volatile 变量的++操作在执行时***并不具有*** 原子性。

并且，排序和可见性只能影响 volatile 成员变量本身，把一个引用成员变量声明为 volatile，并不能保证这个成员变量所引用的非 volatile 数据的可见性。同样的，把一个数组成员变量声明为 volatile 也不能保证数组元素的可见性。对于数组来说，volatile 特性不能手工传递，这是因为数组元素本身不能声明为 volatile。

因为没有锁的参与，所以使用 volatile 比同步要划算一些，或者至少开销是相同的。然而，如果在方法内部频繁使用 volatile 成员变量，则整个性能就会降下来。这时，把整个方法声明为同步会更好一些。

如果仅仅需要保证在多线程之间正确地访问成员变量的值，而没有其他需要使用锁的理由，就不如把成员变量声明为 volatile 类型。在如下情况中，这种现象会发生：

- 不需要和其他成员变量之间遵循不变约束。
- 不需要根据当前的值重写成员变量。
- 没有线程会使用正常语法写入非法值。
- 读线程的行为不依赖于任何非 volatile 成员变量的值。

如果通过某种方式得知，只有一个线程可以改变成员变量的值，而其余很多线程随时可以读这个数据，那么使用 volatile 类型就很有意义。例如：Thermometer 类将 temperature 成员变量声明为 volatile 类型。正如 § 3.4.2 中要讨论的那样，volatile 类型可以作为数据完备的标志。在 § 4.4 中还有一些额外的例子，使用轻量级可执行框架自动实现同步的某些特性，但是需要使用 volatile 的声明以确保结果成员变量的值在多个任务中是可见的。

2.2.8 进阶阅读

关于对多线程程序有影响的计算机体系结构的特性描述可参见：

Schimmel 和 Curt.《UNIX Systems for Modern Architectures Symmetric Multiprocessing and Caching for Kernel Programmers》，Addison-Wesley, 1994.

Patterson, David 和 John Hennessy.《Computer Organization and Design: The Hardware/Software Interface》，Morgan Kaufmann, 1997。请参阅联机补充材料中有关特定机器的体系结构下更多资源的相关链接。

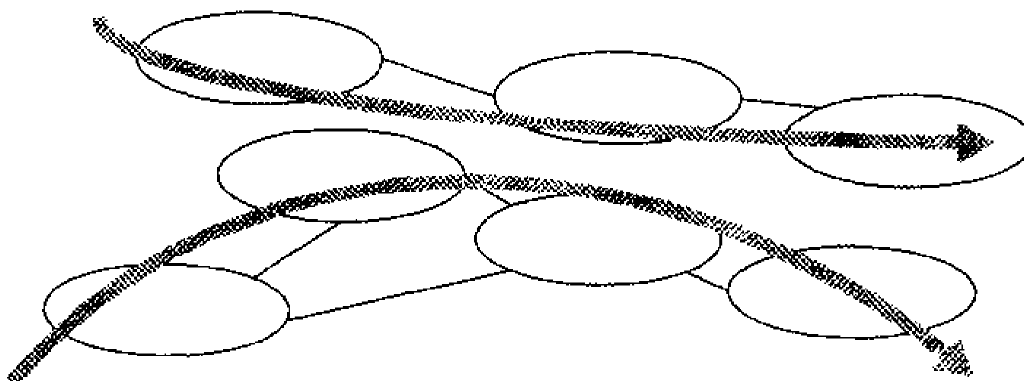
由于多处理器和多线程编程的推广和它们之间的相关性的更多研究，存储的一致模型受到更多的关注。至少在锁方面，Java 的存储模型和***释放一致性模型家族*** 很相近，作为概述，可参见：

Adve, Sarita 和 K. Gharachorloo. "Shared Memory Consistency Models: A Tutorial",《IEEE

java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

Computer》, December 1996, 66~76. 请参阅跟进文章，包括："Recent Advances in Memory Consistency Models for Hardware Shared-Memory Systems"*Proceedings of the EEE*, special issue on distributed shared memory, 1999.

2.3 限制



限制利用封装技术，从结构上保证某一时刻最多只有一个活动访问某个对象。这样可以静态地保证，对某个对象的访问是只在一个线程中惟一的存在，而不需要在每次访问的时候都使用动态的锁。其主要策略是，定义类和方法建立防泄漏的（leak-proof）的***所有者*** 区域，保证只有一个线程，或者一个时刻只有一个线程可以访问被限制的对象。

限制和很多其他安全策略很相似，都是为了保护敏感信息不会泄露到某个区域以外。这里的敏感信息的泄露是指对象的访问，这些对象几乎都是通过引用的方式来访问的。这个问题面临着和安全方面的其他问题的同样挑战：有的时候很难证明一个遗漏都没有，而除非设计是防泄漏的，否则限制策略很不可靠。尽管如此，因为有备份的策略的存在，所以限制比起安全方面的其他问题而言没有那么严重。因此，如果程序员不能使用限制的话，则可以使用本章提供的其他独占策略。

限制依赖于编程语言所提供的作用域、存取控制和安全特征，以支持数据隐藏和数据封装。然而限制的意义是确保数据的惟一性，这不能被语言完全支持。有四种方法可以保证在某个活动中，对象 x 的引用 r 是否可能在方法 m 中"***泄露*** "：

- 在方法 m 中调用方法或者进行对象初始化的时候，把 r 作为参数。
- 在方法 m 中把 r 作为方法调用的返回值。
- m 在一些可以被其他活动访问的成员变量中记录 r（在最坏的情况下，在任何地方都可以访问 static 类变量）。
- m 释放（通过上面任意一种方法）另外一个引用，通过这个引用可以访问 r。

如果泄露只发生在不会引起我们感兴趣的对象之状态的改变（通过成员变量赋值）的方法中，那么对这种部分遗漏是可以忍受的。（参见 § 2.4.3）

在一些封闭的类和子系统中（参见 § 1.3.4），可以对此做彻底的检查。在开放系统中，很多约束只能作为设计原则，通过工具和审核来支持。

java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

这部分讨论四种限制。第一种，也是最简单的一种，方法限制，包括和局部变量相关的普通编程。第二种线程限制，介绍将访问控制在线程内部的技巧。第三种对象限制，利用面向对象的封装技术，为方法访问对象封提供一种更强的保证访问惟一性的机制。第四种，***组***限制，把这几种技术应用到跨越线程的对象集合的协作上。

2.3.1 跨方法的限制

如果一个方法调用并且创建一个对象，但又不允许这个对象有遗漏，那么其他线程不会干扰（甚至根本就不知道）这个线程对该对象的使用。在所有的编程策略中，用局部作用域隐藏可见性是一个普遍的封装策略。

需要稍加注意的是，这个方法可以扩展到串行调用方法的情况中。例如，看一下下面这个使用 java.awt.Point 的类。Point 类只是一个简单的记录类型，因为它拥有公共的 x 和 y 成员变量，所以在多个线程之间共享 Point 类的实例很不明智。

```
class Plotter {                                          // Fragments
  // ...

  public void showNextPoint() {
    Point p = new Point();
    p.x = computeX();
    p.y = computeY();
    display(p);
  }

  protected void display(Point p) {
    // somehow arrange to show p.
  }
}
```

这里的 showNextPoint 方法创建了局部的 Point 类型的变量。这样，在 showNextPoint 保证不再访问 p 的情况下，可以允许 Point 只能***通过尾部调用(tail call)***的方式泄露到 display(p) 中，在这之后，别的线程可能访问 Point 类（别的线程后来访问 Point 类是有可能的。尤其是所有基于图形的程序都依赖于 AWT 事件线程。参见 § 1.1.1.3 和 § 4.1.4，尽管在这里线程不大可能修改 Point 对象）。

这里是一个***传递（hand-off）***协议的例子，这种协议保证在某个时刻上最多只有一个活动的执行方法可以访问某对象。这种尾部调用的方式是最简单，通常也是最好的方法。在***工厂（factory）***方法中也有同样的使用方法。工厂方法创建和初始化对象，并在最后返回这个对象，可以参见 § 1.1.1.3 中的 ParticleApplet.makeThread 方法。

2.3.1.1 会话

很多传递序列按会话构建。在会话中，某个公共的入口方法负责创建对象，这个对象将被限制在组成某种服务的一组操作序列中。在完成操作之后，这样的入口方法还应该负责相应的清除工作。例如：

java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

```
class SessionBasedService {                         // Fragments
  // ...
  public void service() {
    OutputStream output = null;
    try {
      output = new FileOutputStream("...");
      doService(output);
    }
    catch (IOException e) {
      handleIOFailure();
    }
    finally {
      try { if (output != null) output.close(); }
      catch (IOException ignore) {} // ignore exception in close
    }
  }

  void doService(OutputStream s) throws IOException {
    s.write(...);
    // ... possibly more handoffs ...
  }
}
```

如果程序员可以选择的话，一般都会在 finally 语句中进行清除工作，而不是依靠析构函数（也就是覆盖 Object.finalize 方法）。使用 finally 语句可以很清楚地知道清除工作是在什么时候进行的，这一点可用于保护像文件这样的稀有资源。相反地，析构函数一般都是在可能的情况下由垃圾收集器异步地触发。

2.3.1.2 替代协议

如果一个方法必须在做一次调用之后才能访问某个对象，或者必须进行多次调用，那么尾部调用的传递协议就派不上用场了。必须要增加新的设计原则来覆盖这些情况，例如如下修改 Plotter 类中的方法。

```
public void showNextPointV2() {
  Point p = new Point();
  p.x = computeX();
  p.y = computeY();
  display(p);
  recordDistance(p); // added
}
```

选项包括：

调用者拷贝：有一些类，例如：Point，在传递中只是表示数据值，其对象身份无关紧要，因而，调用者可以首先将这个对象做一个拷贝，然后再将该拷贝拿给接收者用，这里是一个例子：

```
display(p);
```

可以由

```
display(new Point(p.x, p.y));
```


java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

接收者拷贝：如果一个方法对作为参数传递给它的引用对象的使用约束一无所知（同样，如果对象身份无关紧要）。那么这个方法可以做一个本地拷贝，留着自己使用。这里是一个例子，display 方法的第一行可以为：

```
Point localPoint = new Point(p.x, p.y);
```

使用标量参数：通过发送标量参数，提供接收者需要的创建对象的信息，而不是发送引用，这可以消除在调用者和接收者之间职责的不确定性。例如，我们可以重新制定 dislpay 方法的参数：

```
protected void display(int xcoord, int ycoord) { ... }
```

并通过以下形式调用：

```
display(p.x, p.y);
```

信任：接收者（或者类的作者）可以许诺不通过引用参数的方式修改或传递对象。而且，它也要确保在后续的调用中尽量减少不希望的访问。

如果上面提到的几种方式都无法做到，那么纯粹的限制就不能保证成功，应该使用本章提供的其他方法。例如，如果这里不需要使用 java.awt.Point 类，那么程序员就可以使用 ImmutablePoint 类来确保对象不被修改（参见§2.4.4）。

2.3.2 线程内限制

基于线程的限制技巧[5]是从方法串行调用扩展来的。实际上，最简单也是很有效的一种方法是使用每线程一个会话（thread-per-session）的设计策略（参见§4.1），其实这就等于基于会话的限制。例如：可以在 run 中初始化传递：

```
class ThreadPerSessionBasedService { // fragments
  // ...
  public void service() {
    Runnable r = new Runnable() {
      public void run() {
        OutputStream output = null;
        try {
          output = new FileOutputStream("...");
          doService(output);
        }
        catch (IOException e) {
          handleIOFailure();
        }
        finally {
          try { if (output != null) output.close(); }
          catch (IOException ignore) {}
        }
      }
    };
```

5 显然这里假设读者只有第一章所说的基本的线程使用知识，但读者简单地看一下第四章还是很有帮助的。

java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

```
    new Thread(r).start();
  }

  void doService(OutputStream s) throws IOException {
    s.write(...);
    // ... possibly more hand-offs ...
  }
}
```

一些并行软件的设计（例如 CSP——参见 § 4.5.1）把一个线程可以访问的所有成员变量都严格地限制在这个线程里。这种方法是与基于并发编程（参见 § 1.2.2）中保证在进程间（在这里是线程间）的地址空间相分离的方法相类似的。

但是，通常限制线程中的每个对象的访问权限是不可能的。运行在 JVM 上的所有线程最后至少要共享底层的资源，例如，由 java.lang.System 中的方法控制的资源。

2.3.2.1 线程私有（Thread-specific）成员变量

在一系列的调用中，线程中执行的方法调用除了要接收受限制的引用之外，还可以访问代表它们正在运行线程的 Thread 对象，以及由此可以访问到的更多信息。静态方法 Thread.currentThread()可以被任何方法调用，并且返回调用者的 Thread 对象。

程序员可以利用这个特性，在 Thread 的子类中增加成员变量，并提供只在本线程内部访问这些成员变量的方法。例如：

```
class ThreadWithOutputStream extends Thread {
  private OutputStream output;

  ThreadWithOutputStream(Runnable r, OutputStream s) {
    super(r);
    output = s;
  }

  static ThreadWithOutputStream current()
   throws ClassCastException {
    return (ThreadWithOutputStream) (currentThread());
  }

  static OutputStream getOutput() { return current().output; }

  static void setOutput(OutputStream s) { current().output = s;}
}
```

这个类可以如下使用：

```
class ServiceUsingThreadWithOutputStream {          // Fragments
  // ...
  public void service() throws IOException {
    OutputStream output = new FileOutputStream("...");
    Runnable r = new Runnable() {
      public void run() {
        try { doService(); } catch (IOException e) { ... }
      }
    };
```


java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

```
      new ThreadWithOutputStream(r, output).start();
    }

    void doService() throws IOException {
      ThreadWithOutputStream.current().getOutput().write(...);
      // ...
    }
}
```

2.3.2.2 ThreadLocal

java.lang.ThreadLocal 工具类排除了使用私有线程技术的一个障碍，这个障碍就是对特定 Thread 子类的依赖。java.lang.ThreadLocal 工具类使得线程私有变量可以以特殊的形式增加到任意一段代码中。

ThreadLocal 类内部维护了一张相关数据（Object 引用）和 Thread 实例的表。ThreadLocal 类中的 set 和 get 方法可以存取当前 Thread 控制的数据。从 ThreadLocal 类继承来的 java.lang.InheriableThreadLocal 类可以自动把本线程的变量传递给其创建的任何一个线程。

很多使用 ThreadLocal 的设计都被视为单态（参见 § 2.2.4）的扩展。多数的 ThreadLocal 应用程序为每个线程创建了一个资源的实例，而不是为每个程序创建一个。ThreadLocal 的变量通常被声明为静态的，并且是在包范围内可见，所以这些变量可以在运行于某个线程的一组方法中访问。

ThreadLocal 可以以如下这种方式运行在我们的程序中：

```
class ServiceUsingThreadLocal {                        // Fragments
  static ThreadLocal output = new ThreadLocal();

  public void service() {
    try {
      final OutputStream s = new FileOutputStream("...");
      Runnable r = new Runnable() {
        public void run() {
          output.set(s);
          try { doService(); }
          catch (IOException e) { ... }
          finally {
            try { s.close(); }
            catch (IOException ignore) {}
          }
        }
      };
      new Thread(r).start();
    }
    catch (IOException e) { ...}
  }

  void doService() throws IOException {
    ((OutputStream)(output.get())).write(...);
    // ...
  }
}
```

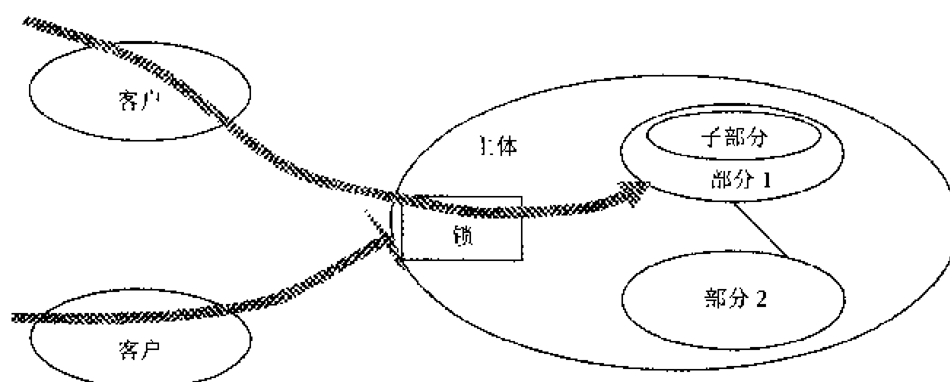

java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

2.3.2.3 应用和结果

拥有线程私有数据的ThreadLocal和Thread子类，一般只在没有更好选择的情况下选用。与其他设计相比（例如，基于会话的设计），其优缺点包括：

- 把对象引用放在Thread对象内部（或者与其相关联），使得运行于同一个线程的方法可以共享这些引用，而不需要以参数的形式传递。在维护诸如当前线程的AccessControlContext（类似于java.security包中那样）等上下文信息，或者为打开相关文件而保存的当前工作目录时，使用这种线程局部的方式是一个很好的选择。ThreadLocal还可以用来创建每个线程的资源池（参见§3.4.1.2）。
- 使用线程私有变量会隐藏影响行为的参数，这使得更加难于进行错误或遗漏检查。从这个意义上说，线程私有数据存在着和静态全局变量同样的可追踪性问题，尽管没有静态全局变量那么严重。
- 保证线程私有数据状态的改变（例如：关闭一个输出文件，打开另一个）会影响所有相关的代码。这一点实现起来很简单，但是保证所有这些改变间的相互协调却是很难的。
- 在线程内部对线程私有数据进行读写时不需要同步。但是通过currentThread或内部的ThreadLocal表的访问路径不比没有竞争时的同步方法所要付出的代价低。所以，只有在对象需要共享且在多个线程间竞争使用该对象的情况下，改用线程私有数据技术才有可能提高系统的整体性能。
- 使用线程私有数据会增加代码的依赖关系，从而降低了代码的重用性。在利用Thread子类的时候，该问题更加突出。例如：doService只有运行在ThreadWithOutputStream类型的线程中的时候才是可用的。如果不在这种情况下使用，调用current方法会引起ClassCastException异常。
- 通过ThreadLocal增加上下文信息有时是惟一可以使组件和不在调用序列间传递信息的现有代码协调工作的方法（参见§3.6.2）。
- 轻量级可执行程序框架只是间接的基于Thread类，尤其是工作者线程池（参见§4.1.4）。所以，在轻量级可执行程序框架中很难把相关数据和执行上下文联合起来。

2.3.3 对象内限制



java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

即使程序员由于不能在一个特定的方法或者线程中限制对对象的访问而必须使用***动态***锁时，程序员也可以对所有在对象***内部***、对不同部分的访问进行限制。这样，一旦一个线程进入到这个对象的一个方法后，就不再需要***额外***的锁。用这种方式，对外部 Host 容器对象的独占控制，可以自动传递给内部的 Part（指的是 Part1 和 Part2——译者注）。为了能这样工作，对 Part 的引用必须是防漏的（参见 § 2.4.5 介绍的在容器泄漏无法避免的情况下使用的其他策略）。

在各种面向对象的设计程序中都可以找到对象限制技术。在并发上下文中增加的主要要求就是要保证 Host 对象的各个入口点都要应用同步。这种方法与建立含有像 double 这样的标量类型数据的完全同步对象（参见 § 2.2.2）时所使用的技术相同。但是在这里，这项技术用于包含对其他对象的引用的类中。

在基于限制的设计中，Host 对象被视为***拥有*** 内部 Part 对象的对象。反过来，也可以认为 Part 对象“物理上”包含在 Host 对象中：

- Host 对象在其构造函数中创建每个 Part 对象的实例，并将对 Part 对象的引用赋值给非公有的成员变量。这样的构建保证对于 Part 对象的引用不会被其他对象所共享。还有一种做法就是把构造函数作为一个传递点。
- 和在其他各种限制技术中一样，Host 对象不能泄漏出任何对 Part 对象的引用：不能传递作为函数的参数或者返回值的引用，必须保证保存引用的成员变量对外是不可访问的。而且，Part 对象也不能泄漏出自己的标识，例如，将 this 作为一个回调参数（参见 § 4.3.1）来传给一个外部方法。只有这样，才能保证 Part 对象对于外部来说只能通过 Host 对象的方法来访问。
- 在最保守的限制设计——***固定包含（fixed containment）***中，宿主对象不会对已经指向内部 Part 对象的引用成员变量重新赋值，这样就避免了在 Host 对象中更改成员变量时所需的同步。固定包含实现了在《Design Patterns》一书中所说的聚合(aggregation)的主要思想，在 UML 中用一个菱形符号来代表。
- 除非 Host 对象也被限制在其他对象中，否则 Host 对象中的相应方法都应该是同步的（参见 § 2.3.3.2，一种定义类的同步版本和非同步版本的方法）。这样就保证了所有对于 Part 对象（以及所有在它们内部创建的对象）的访问都是独占的。注意，在单个限制域中的不同 Part 对象之间，不需要同步就可以互相访问其他对象的方法，只有外部访问才需要同步。

尽管以上约束必须保证（有时候很难检查），对象限制技术还是应用得很广泛，部分原因是因为这项技术可用于创建适配器以及其他的基于代理（delegation）的设计。

2.3.3.1 适配器

在完全同步的主体对象中，适配器（参见 § 1.4.2）可以用于封装原始的没有经过同步的基本（ground）对象。这种方式可以引发最简单的代理式设计：就是那种把所有信息直接发送给其代理的适配器。同步适配器可以将原来用于串行机制的代码，以及动态装载的不能在安全方面得到保证的代码嵌入到多线程的上下文中。

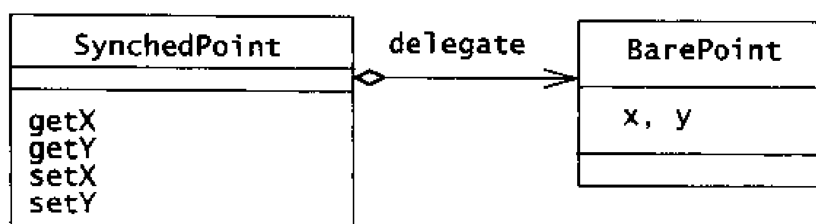
java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

有些被极度优化的（也许会用 native 代码），具有密集计算功能的代码，为了效率起见，不执行内部的并行控制。适配器可以为这样的代码提供惟一的安全入口。值得注意的是，无论怎么包装，都没有办法解决由于使用 native 代码而在多线程之间不安全地访问内部成员变量的问题。

对于一个或者多个非保护的基本类，程序员可以定义一个同步的适配器类，用一个 delegate 成员变量保存一个对基本对象的引用，通过引用可以向基本对象发送请求，并且返回应答（注意：如果某个基本方法以 return this 的方法给出应答，那么适配器也应该以 return this 的形式传递）。代理的引用不需要是 final 类型的，但是如果它们是可重新分配的，一定要注意适配器要独占访问。例如，一个适配器可能偶尔会将引用指向一个内部新建的代理。

正如在 § 1.4 中提到的，有些时候，适配器和其内部维护的基本对象必须保证等价，程序员可以通过覆盖相应的 equals 和 hashCode 方法来满足要求。但是在基于限制的设计中没有理由这么做，因为内部对象不会被漏掉，所以也没有进行比较的可能性。

作为一个简单的例子，同步适配器可以对包含 public 实例变量的诸如完全开放的 Point 类设置同步访问和更新方法：



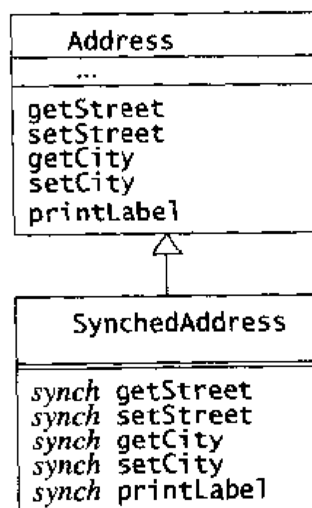

```
class BarePoint {
  public double x;
  public double y;
}

class SynchedPoint {

  protected final BarePoint delegate = new BarePoint();

  public synchronized double getX() { return delegate.x;}
  public synchronized double getY() { return delegate.y; }
  public synchronized void setX(double v) { delegate.x = v; }
  public synchronized void setY(double v) { delegate.y = v; }
}
```

java.util.Collection 框架就是使用基于适配器的策略组织起层次化的集合类同步。除了 Vector 和 HashTable 之外，基本的集合类（例如 java.util.ArrayList）都是非同步的。但是匿名的同步适配器可以如下面这样在基类基础上建立起来：

```
List l = Collections.synchronizedList(new ArrayList());
```


java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

2.3.3.2　子类化



当一个类的实例总是限制在其他类中的时候，就没有理由把它的方法同步。但是，如果一个类的部分实例是受限制的，而另一些不是，那么最安全的办法就是：即使不是在所有的使用环境中都需要锁，也要恰当地使用同步（对于使用其他策略的情况，参见 § 2.4.5 和 § 3.3.4）。

随着编译器、工具和运行时系统持续不断的发展，它们可以在很大程度上优化或者最小化多余的锁的开销。但是如果必要，或者值得的话，程序员可以手工定义多个版本的类，并且根据不同环境使用不同版本的类。最简单的方式之一就是子类化（参见 § 1.4.3）：创建无保护的基类，然后将每一个子类中的 m 方法覆盖为 synchronized 方法，最后再在 m 中调用 super.m。例如：

```
class Address {                                        // Fragments
  protected String street;
  protected String city;

  public String getStreet() { return street; }
  public void setStreet(String s) { street = s; }
  // ...
  public void printLabel(OutputStream s) { ... }
}

class SynchronizedAddress extends Address {
  // ...
  public synchronized String getStreet() {
    return super.getStreet();
  }
  public synchronized void setStreet(String s) {
    super.setStreet(s);
  }

  public synchronized void printLabel(OutputStream s) {
    super.printLabel(s);
  }
}
```

2.3.4　组限制

多线程中对对象组的访问能够保证在某一个时间只有一个线程可以访问指定的***资源***对象。这里的每个资源都由一个对象所有，但是所有关系随着时间在改变。维护独占所有权的协议，和 § 2.3.1 中讨论的在方法调用之间传递引用时相似，只不过前者需要更多的结构来操纵对象组和线程间的一致性。

我们把独占资源和物理对象作以下的类比：

- 如果你已经拥有了其中之一（指的是独占资源或者物理对象——译者注），你便可以

java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

用它做一些事情，否则就不能。

- 如果你拥有了它，别人就不能拥有。
- 如果你把它给了别人，你就不再拥有它。
- 如果你把它损坏了，那么没有人会再拥有它。

如果遵循这种方式，对象就可以被看作是资源。我们可以更具体一些把这条原则特征化：在一段时间内，最多只有一个对象的一个成员变量可以指向某个独占资源。可以利用这个事实来保证在某个活动中的限制，从而降低对资源对象的动态同步。

在某些语境或上下文中，包含独占资源的协议被称作：***令牌、指挥棒、线性对象、能力***等，有时也叫***资源***。一些并行和分布式算法依赖于下面的思想：一个时刻只有一个对象拥有令牌。作为基于硬件的例子，令牌网络维护着一个令牌，它在所有节点之间持续不断地传递。每个节点只有在获得令牌的时候才可以发送信息。

尽管多数传递协议都比较简单，但实现起来难免会出错。当引用对象的成员变量获得拥有权的时候，它有时和物理对象的表现行为不太一样。例如：表达式 x.r = y.s 不会使包含成员变量 s 的拥有者 y 在做了这个操作之后失去所有权。相反，这个操作实现了 r 和 s 的绑定（这可以和现实生活中的知识产权以及其他权限问题做类比，它们本质上不需要物理的传递操作）。这个问题引发了一系列的解决方案，从非正式的协议到各种权重的法律机制。

为了增强可靠性，程序员可以把协议封装在执行下面操作的方法中：不同资源对象 r 和 s，与在成员变量 ref 中持有它们的所有者对象 x 和 y 可以表现为如下关系。需要强调的是，所需的锁是用同步块的方式体现的。

获取：拥有者 x 建立对 r 的初始所有权。通常是执行构造函数，或者是初始化 r 并且通过以下方法赋值：

```
synchronized(this) { ref = r; }
```

丢弃：拥有者 x 使得资源 r 不属于任何拥有者。这通常是通过当前的拥有者执行下面的操作实现的：

```
synchronized(this) { ref = null; }
```

放入（给予）：拥有者 y 向拥有者 x 发送一个消息，消息中把资源 r 的引用作为参数传递，在此操作之后，y 不再对 r 有拥有权，而 x 获得此拥有权。

```
                 x                                    y
void put(Resource s) {              void anAction(Owner x) { //...
  synchronized(this) {                Resource s;
    ref = s;                          synchronized(this) {
  }                                     s = ref;
}                                       ref = null;
                                      }
                                      x.put(s);
                                    }
```

拿出：拥有者 y 请求从拥有者 x 获得资源，x 把 r 作为返回值送出，并释放所有权。

java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

```
                  x                                       y
Resource take() {                        void anAction(Owner x) { //...
  synchronized(this) {                     Resource r = x.take();
    Resource r = ref;                      synchronized(this) {
    ref = null;                              ref = r;
    return r;                              }
  }                                      }
}
```

交换: 拥有者 y 用资源 s 与拥有者 x 的资源 r 做交换。这个操作可以按 s = exchange(null) 的方式执行拿出操作，或者通过 exchange(r)执行，并且忽略返回结果以实现放入操作。

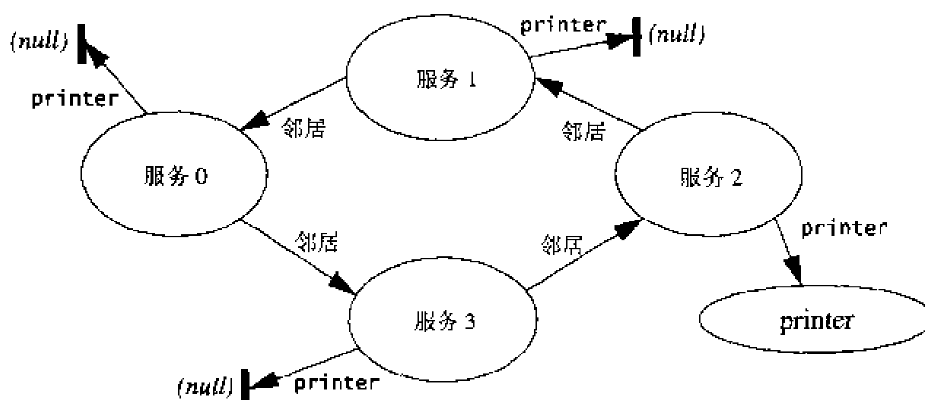
```
                  x                                       y
Resource exchange(Resource s){           void anAction(Owner x) { //...
  synchronized(this) {                     synchronized(this) {
    Resource r = ref;                        ref = x.exchange(ref);
    ref = s;                               }
    return r;                            }
  }
}
```

在下面的情况中使用这个协议：一个 OutputStream 对象，*几乎* 完全限制在主体对象内，但偶尔会被其他客户使用。这种情况下，程序员可以允许客户拿出内部的对象，对它进行操作，然后再把它放入。虽然主体对象的控制功能将被减弱，但至少保证了不会有完整性冲突问题。

2.3.4.1 环

一般情况下，资源的管理可能要维护一个池（参见 § 3.4.1），使用消息传递网络，这个网络可能采用某种交换（参见 § 3.4.3）或者流（参见 § 4.2）策略，或者采用可以避免死锁或资源耗尽的策略（参见 § 4.5.1）。但是当只需要确保一组相互交互的对象一起严格地限制一个资源的时候，可以使用更简单的传递协议。做到这点的一个简单办法就是把一组平等的类放到一个环（ring）中，在环中，每个节点都只和其惟一的临节点通信。

作为一个不太真实的简化例子，我们可以想象有一组 PrintService 对象作为节点放到环中，在它们中间传递着对 Printer 的使用权。如果一个节点申请打印却没有这个权限，那么它会尝试从其邻居那里得到这个权限。这个请求会顺序传递到有打印权限的节点。把相关的方法定义为同步可以保证节点在完成打印之前不会释放打印机。下面是这种结构的一个快照：



java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

只有在所有的节点都遵守传送协议，所有连接都完好地建立，并且至少有一个节点有打印机的时候，这种设计才能产生预想的结果。例子中的初始方法展示了一种建立所需结构的方法。还需要一些扩展来支持新的 PrintService 对象的动态加入，以支持多个 Printer 和处理没有 Printer 的情况。

```
class Printer {
  public void printDocument(byte[] doc) { /* ... */ }
  // ...
}

class PrintService {

  protected PrintService neighbor = null; // node to take from
  protected Printer printer = null;

  public synchronized void print(byte[] doc) {
    getPrinter().printDocument(doc);
  }

  protected Printer getPrinter() {     // PRE: synch lock held
    if (printer == null)  // need to take from neighbor
      printer = neighbor.takePrinter();
    return printer;
  }

  synchronized Printer takePrinter() { // called from others
    if (printer != null) {
      Printer p = printer; // implement take protocol
      printer = null;
      return p;
    }
    else
      return neighbor.takePrinter(); // propagate
  }

  // initialization methods called only during start-up

  synchronized void setNeighbor(PrintService n) {
    neighbor = n;
  }

  synchronized void givePrinter(Printer p) {
    printer = p;
  }
  // Sample code to initialize a ring of new services

  public static void startUpServices(int nServices, Printer p)
   throws IllegalArgumentException {

    if (nServices <= 0 || p == null)
      throw new IllegalArgumentException();

    PrintService first = new PrintService();
    PrintService pred = first;
```


java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

```
      for (int i = 1; i < nServices; ++i) {
        PrintService s = new PrintService();
        s.setNeighbor(pred);
        pred = s;
      }

      first.setNeighbor(pred);
      first.givePrinter(p);
    }
  }
```

2.3.5 进阶阅读

Hermes 编程语言是一些搭建并发和分布式程序的语言构件和技术的前身，其中包括引用传递原语，参见：

Strom, Robert, David Bacon, Arthur Goldberg, Andy Lowry, Daniel Yellin 和 Shaula Yemini.《Hermes: A Language for Distributed Computing》，Prentice Hall, 1991.

Spring 操作系统接口定义语言把转接策略作为方法的参数限定符，参见：

《A Spring Collection》，SunSoft Press, 1994.

基于惟一引用的技术在其他面向对象的设计和分析的方法中也扮演着很重要的角色，例如，参见：

Hogg, John, Doug Lea, R.C. Holt, Alan Wills 和 Dennis de Champeaux. "The Geneva Convention on the Treatment of Object Aliasing"，《OOPS Messenger》，April 1992.

在分布式系统中使用限制的正式方法，参见：

Gardelli, Luca 和 Andrew Gordon. "Mobile Ambients", in Maurice Nivat(ed)，《Foundations of Software Science and Computational Structures》，Springer LNCS 1378, 1998.

2.4 构造和重构类

在类设计的初期，围绕独占访问控制很难权衡各种设计元素。在并行程序中的很多类都经历过反复的重构过程来处理下面这些因素：

- 像很多基于限制的设计一样，少量的几个入口点锁（entry-point lock）在只有很少几个线程的情况下可以很好地工作，因为这时的开销并不大。但是在竞争的环境下，尤其是在多处理器的环境下，系统性能会很快下降。如果多个线程竞争同一个入口点锁，那么多数线程会把大部分时间浪费在等待锁上，因而增加了系统延迟，限制了并行的优越性。多数系统都随着系统的增长而开始使用细粒度锁。最有名的一个案例是，曾经有一个操作系统只使用了一个*巨型*锁作为访问内核的入口点，（但是为了更好的支持多进程，增加了一些小范围、持有时间很短的锁的使用）。
- 使用太多的锁会增加系统负担，以及不可预料的活跃性失败的几率。
- 只用一把锁来保护一个功能的多个方面会导致不必要的竞争。

java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

- 长时间持有锁会引起性能和活跃性问题，使异常处理变得复杂。
- 锁住单一一个方法有时不会达到预想的目的。例如：我们要通过调用两个不同的、加锁的访问方法来获取两个相关的属性，如果在两个调用之间有状态转换，就可能不会得到预想的关系。

这里没有一个最佳的策略，但是可以通过几种技术和模式来在这几种因素之间权衡利弊。本节就介绍这些相关策略，用以除去不必要的同步，用同步分解来匹配功能，通过适配器对外支持只读操作，通过分离状态表示从而降低访问开销或者增强潜在的并行机制，将对象分组，通过对组使用共同的锁来映射层次化设计。上面任何一种方法都可以在类的初始设计时使用，但是有些方法的实现要依赖于一些在设计初期很难使用（或者很不明智）的技术。

2.4.1 减少同步

对于一个类或一段程序来说，如果锁呈现出活跃性或性能问题，那么最好的方法就是用本章中的其他方法重新设计。事实是，在取消一些 synchronized 方法或块的同步时（尽管有时这么做会弱化语义保证），基于同步设计的基本逻辑仍可以被保存。

2.4.1.1 *存取*（accessor）

对一个用来存取成员变量的方法同步，有时（但并不总是）会有明显的开销。关于是否对一个用来存取成员变量的方法进行同步，要考虑两点：

合法性：相应成员变量永远不会是非法值。也就是，程序员要保证成员变量要永远不能（即使是瞬间也不可以）违背一致性原则（从定义上讲，这种成员变量不包括 long 和 double 类型的成员变量）。

陈旧性：客户不是必须要得到成员变量最近更新的值，偶尔用过时的数据也可以（参见§2.2.7）。

如果成员变量*不*总是合法的，那么程序员的选择可以是：

- 同步所有存取方法（和所有更新数据的方法）。
- 确保客户在得到非法值的时候能够意识到，并且采取行动［例如：通过双重检查（double-check），参见§2.4.1.2］。
- 省略存取方法。这种方法应用得出奇的多。自问一下，为什么每个客户都想知道成员变量的值以及如何处理这个值？因为在并行编程中，对象的属性可以异步地改变，某个客户通过一行代码得到的值可能在执行下一行代码之前就已经改变了。因此，在并发编程中存取方法不总是很有用。正因为其不总是很有用，所以即使程序员在这种情况下不去掉存取方法，同步也不会涉及到性能问题。

如果一个成员变量总是合法的，但不能是陈旧的数据，那么你可以有下面的选择：

- 从存取方法中去掉同步，把实例变量标记为 volatile。但这种方法只是在对标量使用时可以得到预想的结果，对数组或 Object 的引用得不到预想的结果，这些数组或 Object 的引用只是用来帮助维护对象的标识：访问一个 volatile 类型的引用，不会自

java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

动保证引用所代表的成员变量或元素本身的可见性（对于 Object 对象，如果必要，可以进一步把被引用对象本身声明为 volatile）。这种方法的主要缺陷是，volatile 类型的声明妨碍了编译器对使用这个成员变量的方法的优化，从而导致了纯粹的性能损失。

如果一个成员变量总是合法的，并且陈旧数据是可以接受的，那么你可以有下面的选择：

- 从存取方法中去掉同步，而不必考虑进一步的改变。
- 如果不怕危险的话，还可以把这个成员变量声明为 public。

作为一个例子，考虑一下 ExpandableArray 类的 size()方法，它返回 size 的值。通过对所有方法的检查发现 size 成员变量总是合法的——它从来不会是 0~data.length 之外的值（也许这不一定保证为真，例如，当 size 在 add 方法中设置成-1 以表示需要重新设置尺寸时）。假设这种约束作为对所有子类和未来修改的内部要求已经文档化，那么同步可以从存取方法中去掉。

决定数据陈旧与否，或者是否需要声明为 volatile 类型依赖于在使用环境中的遍历策略的选择，如果客户主要用索引循环来遍历元素：

```
for (int i = 0; i < v.size(); ++i)      // questionable
  System.out.println(v.get(i));
```

那么使用陈旧的 size 的值是不能接受的。例如：某个客户获得 size 的值是 0，这样它就会跳过循环，即使此时数组中有很多元素也是如此。注意，如果一旦运行循环，在执行第一个同步的 get 之后便会刷新 size，作为下一次调用 size()的值（可以回顾一下 § 2.2.3，关于客户在任何情况下都要准备好处理在索引检查和元素访问之间 size 改变的情况，所以这种遍历方法是有问题的）。

但是如果使用聚合或者迭代变量，因为它们都执行内部的同步操作，所以程序员可以使用非同步的 size()方法和非 volatile 的 size，并且声明这个方法只是大致估计了当前元素的个数。然而，这一点对于使用通常的类（例如 ExpandableArray）的客户来说也是不能接受的。但是对另一些客户来说，他们获得的数据只要是上次同步的线程读或写之后的最新数据就可以了，这时，我们可以做如上的处理。

2.4.1.2 双重检查

在对一个非同步成员变量进行访问时，如果调用者发现读到了非法成员变量的时候，可以采取相应的措施。这样的措施之一就是在同步的环境下重新访问这个成员变量，判断一下它的最新值，然后采取恰当的行动。这就是双重检查思想的精髓。

在闭锁（参见 § 3.4.2）、SpinLock 类（参见 § 3.2.6）和缓存协议中都可以看见双重检查及其变体（包括被称为***测试-然后-测试-然后-设置***的循环版本）。但是双重检查最普遍的应用是有条件地放宽初始化检查的同步操作。如果遇到一个没有初始化的值（对于标量，默认值是 0），访问方法需要一把锁来同步检查一下初始化工作是否必须（而不是仅仅读取陈旧的值），并且在同步锁下执行初始化操作，以防止多次初始化。例如：

java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

```
class AnimationApplet extends Applet {              // Fragments
  // ...
  int framesPerSecond; // default zero is illegal value

  void animate() {
    // ...
    if (framesPerSecond == 0) { // the unsynchronized check
      synchronized(this) {
        if (framesPerSecond == 0) { // the double-check
          String param = getParameter("fps");
          framesPerSecond = Integer.parseInt(param);
        }
      }
    }

    // ... actions using framesPerSecond ...
  }
}
```

尽管双重检查有一些合理的应用，但做起来要十分的细致：

- 一般来说，对对象或者数组的引用使用双重检查是不明智的。对于没有同步的读操作来说，引用的可见性不能保证从引用访问的非 volatile 对象的可见性。即使引用非空，在没有同步时通过引用访问的数据也可能是陈旧的值。
- 只使用一个标志成员变量来指示所有必须被初始化的成员变量是很难的。在 § 2.2.7 中讨论的顺序化重新排序会导致其他成员变量在还没有被初始化前，标志本身就已经被设置为已初始化了。

对这两个问题的补救通常需要使用某种锁。这种思想通常导致避免使用双重检查这种偷懒的初始化方法。取而代之的是或者使用尽早初始化的机制，或者是基于完全同步的检查（正如在 § 2.2.4 中单态类的例子）。

但是在另外一些情况下，程序员还可以用更弱的方法——***单次检查***。这里的检查、初始化和成员变量绑定都在***没有***同步的情况下执行。这样就要依赖于访问非同步成员变量的具体情况，同时会导致多实例化出现的可能性。所以这种情况只有在初始化操作没有负面影响，不需要同步操作，并且很少使用多线程的时候才是合理的。

2.4.1.3　开放调用（Open Call）

正如在 § 2.1 中讨论的，如果一个方法不对任何可变成员变量进行访存操作，或者不依赖于任何可变成员变量，那么这个方法就是无状态的。操纵完全不变对象的方法应该是无状态的，但是无状态的方法也可以出现在其他类型的类中，例如：在纯计算的工具方法中，或对相关对象（acquaintances）做方法调用的时候（可以把这种对象和 § 1.3.1.2 中的支持状态表示的对象做相反的对照）。

程序员不需要同步一个方法中无状态的部分，这样做使得在非同步部分中可以进行 synchronized 方法的调用，从而增强性能并减少锁的干扰。但是，同步操作只有在方法的各部分无论如何都不会有依赖关系的时候才可以分解，这样，在方法完全结束之前“看见”或者使用对象才是可以接受的。

java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

为了举例说明，我们不妨考虑下面这种普遍的服务类。如果 helper.operation 操作需要很长时间，那么调用像 getState 这样的 synchronized 操作就需要阻塞一段让人无法接受的时间，来等待这个方法可用。

```
class ServerWithStateUpdate {
  protected double state;
  protected final Helper helper = new Helper();

  public synchronized void service() {
    state = ...; // set to some new value
    helper.operation();
  }

  public synchronized double getState() { return state; }
}
```

如果 helper 代表了主机状态的某些方面，或者对 helper.operation 的调用依赖于，或者会修改主机状态，那么整个 service 方法都必须使用同步机制。但是，如果 helper.operation 的调用和主机无关，那么 service 方法就可以使用 § 1.1.1.1 中建议的默认原则来组织。

- 首先，更新状态（控制锁）。
- 然后，发送消息（不需要控制锁）。

非同步的发送消息也被称为***开放调用***。正如在 § 4.1 和 § 4.2 中讨论的那样，拥有这种形式方法的类表现良好，适合于在并行和事件驱动的系统中作为部件。例如，假设 helper.operation 满足我们的需要，那么上面的类就可以重写为：

```
class ServerWithOpenCall {
  protected double state;
  protected final Helper helper = new Helper();

  protected synchronized void updateState() {
    state =   ...; // set to some new value
  }

  public void service() {
    updateState();
    helper.operation();
  }
}
```

即使 helper 引用成员变量是可变的，这里也可能用到开放调用。例如，使用同步块：

```
class ServerWithAssignableHelper {
  protected double state;
  protected Helper helper = new Helper();
  synchronized void setHelper(Helper h) { helper = h; }

  public void service() {
    Helper h;
    synchronized(this) {
      state = ...
```


java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

```
      h = helper;
    }
    h.operation();
  }

  public synchronized void synchedService() { // see below
    service();
  }
}
```

synchedService 方法揭示了使用开放调用方法的弱点。在 synchedService 方法内部调用 Service，会导致在整个 service 执行过程中锁都被占用，包括对 h. operation 的调用。这违反了重新创建该方法的目的。要避免这样的问题，需要将并行设置中类里不使用同步的现象文档化。

通过不变引用连接成的数据结构通常适用于这种操作。例如，有一个 LinkedCell 类，类中的每个单元（cell）都包含有对它后续单元的引用，并且对于这个类来说，我们要求在创建时就固定了对后续单元的引用。对于如 Lisp 样式的表格来说，这是一个普遍需求。有关后续单元的方法或者方法的一部分不需要同步，这样做可以加快遍历的速度。为了清晰和强调，下面的方法使用了递归，实际上，程序员也可以使用迭代。

```
class LinkedCell {
  protected int value;
  protected final LinkedCell next;

  public LinkedCell(int v, LinkedCell t) {
    value = v;
    next = t;
  }

  public synchronized int value() { return value; }
  public synchronized void setValue(int v) { value = v; }

  public int sum() {                  // add up all element values
    return (next == null) ? value() : value() + next.sum();
  }

  public boolean includes(int x) { // search for x
    return (value() == x) ? true:
             (next == null)? false : next.includes(x);
  }
}
```

在这里需要再次强调，当一个 synchronized 方法调用一个非同步的方法时，对象还处在被锁的状态。因此，在下面的代码中，在使用 sum 方法时仍然处于同步状态：

```
synchronized int ineffectivelyUnsynchedSum() {    // bad idea
  return value + nextSum();      // synch still held on call
}
int nextSum() { return (next == null)? 0: next.sum(); }
```


java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

2.4.2 分解同步

如果一个类的表示和行为可以分解为互相独立、互不干扰，或者只是不冲突的子部分，那么就值得用细粒度的辅助对象来重新创建类，这些辅助对象的行为由主机来代理。

这条规则是面向对象设计中的普遍规则，但在并行面向对象编程中有更多的要求。如果有一组 synchronized 操作都在等待和某个对象相关的一个同步锁，那么就会出现死锁，或者活跃性及其他基于锁的性能问题。但如果它们等待的是不同的锁，就可能不会出现死锁，并且/或者运行会效率提高。普遍的原则是，把类内部的同步操作分得越细，在大多数情况下，它的活跃性属性就越高。但这一点是以更加复杂和潜在的错误为代价的。

2.4.2.1 分解类

我们来看一个非常简单的例子——Shape 类。Shape 类保存着位置和维度信息，而耗时的 adjustLocation 和 adjustDimensions 分别对它们进行独立地修改：

```
class Shape {                                    // Incomplete
  protected double x = 0.0;
  protected double y = 0.0;
  protected double width = 0.0;
  protected double height = 0.0;

  public synchronized double x()       { return x;}
  public synchronized double y()       { return y; }
  public synchronized double width()   { return width;}
  public synchronized double height()  { return height; }

  public synchronized void adjustLocation() {
    x = longCalculation1();
    y = longCalculation2();
  }

  public synchronized void adjustDimensions() {
    width = longCalculation3();
    height = longCalculation4();
  }

  // ...
}
```

我们假设 adjustLocation 从来不处理维度信息，并且 adjustDimensions 从来不处理位置信息，因而可以通过重新设计类来提高其性能，即 adjustLocation 的调用者不需要等待 adjustDimensions 调用的完成，反之亦然。

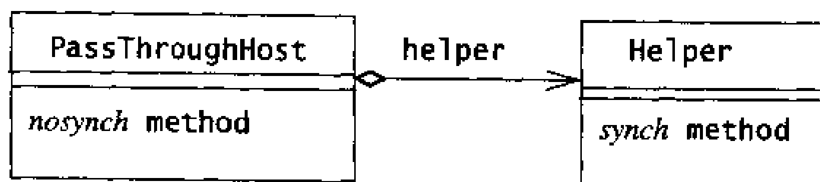
在重新创建类时，通过分解类来降低粒度是最直接的尝试：

- 把 Host 类的一些功能分解成其他一些类，我们称为 Helper。
- 在 Host 类里，声明惟一的 final 成员变量，该成员变量作为构造函数中初始化的新 Helper 类的引用（换句话说，就是将每个 helper 严格地限制在 host 中）。
- Host 类用非同步的方法，以公开调用的方式直接把相应的调用传递给 Helper 类。这

java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

种做法可行是因为对于 Host 类来说，这样的方法是无状态的。

这几步的极端结果就是***通过主机*** ***(Pass-Through Host)*** 的设计。在这种设计中，所有的信息都以简单的、非同步的公开调用方式被传递：



例如：下面就是通过主机版本的 Shape 类：

```
class PassThroughShape {

  protected final AdjustableLoc loc = new AdjustableLoc(0, 0);
  protected final AdjustableDim dim = new AdjustableDim(0, 0);

  public double x()              { return loc.x(); }
  public double y()              { return loc.y(); }

  public double width()          { return dim.width(); }
  public double height()         { return dim.height(); }

  public void adjustLocation()   { loc.adjust(); }
  public void adjustDimensions() { dim.adjust(); }
}
class AdjustableLoc {
  protected double x;
  protected double y;

  public AdjustableLoc(double initX, double initY) {
    x = initX;
    y = initY;
  }

  public synchronized double x() { return x;}
  public synchronized double y() { return y; }

  public synchronized void adjust() {
    x = longCalculation1();
    y = longCalculation2();
  }

  protected double longCalculation1() { /* ... */ }
  protected double longCalculation2() { /* ... */ }

}


class AdjustableDim {
  protected double width;
  protected double height;

  public AdjustableDim(double initW, double initH) {
```


java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

```
    width = initW;
    height = initH;
  }

  public synchronized double width() { return width;}
  public synchronized double height() { return height; }

  public synchronized void adjust() {
    width = longCalculation3();
    height = longCalculation4();
  }

  protected double longCalculation3() { /* ... */ }
  protected double longCalculation4() { /* ... */ }

}
```

2.4.2.2　分解锁

如果你不能或者不想分解类，则可以分解与每个子功能相关的同步锁。这种操作与刚才提到的把类分解成辅助类，然后再集合所有辅助类的表示和方法的思想类似，只是辅助类的同步锁重新归属于主体类。实际上没有必要严格使用这种方法。

当把一个类的所有其他内容都去掉而只剩下同步锁时，这个类都退化成java.lang.Object。这个事实解释了为什么人们喜欢用 Object 的实例作为同步辅助手段。

每当用 Object 作为同步锁来实现相关设计的时候，程序员也许会自问，某一个锁是为什么样的辅助类对象所使用的。在分解锁的情况下，每个 Object 都控制着一组子方法集的访问，所以每组方法集的每个方法都是基于同一个锁对象的块同步。

分解锁的基本步骤和分解类的很相似：

- 为每个***独立的*** 功能集声明一个 final 对象，我们称为 lock。并且在 Host 的构造函数中初始化，之后就再也不改变了。
  - 锁对象可以是任何 Object 的子类。如果没有什么其他的目的，也可以是 Object 本身。
  - 如果一个子方法集和一个被某成员变量惟一引用的已有对象惟一关联，那么程序员也可以把这个对象作为锁对象。
  - 其中的一个锁对象可以和 host 对象（this）本身相关联。
- 把所有子方法集中的方法声明为非同步的，但是在方法中使用 synchronized(lock){...}的方式把所有代码包起来。

固定大小的哈希表是分解锁的一个应用，表中的每一个桶都拥有它自己的锁（这种策略不能很容易地应用于动态设置大小的哈希表，例如：java.util.Hashtable 中的哈希表，因为它们不能依赖于锁对象不变性）。在管理等待和通知操作的类中也可以看到锁分解，参见 § 3.7.2。

作为一个简单的例子，这里有一个类分解版本的 Shape 类：

```
class LockSplitShape {                          // Incomplete
  protected double x = 0.0;
  protected double y = 0.0;
```


java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

```
  protected double width = 0.0;
  protected double height = 0.0;

  protected final Object locationLock = new Object();
  protected final Object dimensionLock = new Object();

  public double x() {
    synchronized(locationLock) {
      return x;
    }
  }

  public double y() {
    synchronized(locationLock) {
      return y;
    }
  }

  public void adjustLocation() {
    synchronized(locationLock) {
      x = longCalculation1();
      y = longCalculation2();
    }
  }

  // and so on

}
```

2.4.2.3 隔离成员变量

有一些类用于管理相互间独立的属性和特性集合，每一个这样的集合都可以独立于其他的集合使用。例如：Person 类可以有 age、income 和 isMarried 成员变量，这些成员变量的改变和 Person 对象作为整体所做的任何其他操作无关。至于这种做法是否可以接受当然要看在语法上如何使用这个类了。

如果程序员需要同步保护以避免对数据的并发修改，那么不能只通过简单地把成员变量声明为 volatile 来达到这个目的。但是程序员可以使用一种简单的分解方法，把同步保护分担给用来保护对简单类型进行简单操作的对象。这样的类扮演着和 java.lang.Double 及 java.lang.Integer 类相似的角色，只是这样的类不是用于保证不变性，而是用于保证原子性。例如，可以自己创建一个下面的类：

```
class SynchronizedInt {
  private int value;

  public SynchronizedInt(int v) { value = v; }

  public synchronized int get() { return value; }

  public synchronized int set(int v) { // returns previous value
    int oldValue = value;
    value = v;
    return oldValue;
```


java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

```
    }
    public synchronized int increment() { return ++value; }
    // and so on
  }
```

（在线提供的 util.concurrent 包中包含了一组这样的类，每个对应一个基本类型，并且支持其他一些辅助操作，例如，§2.4.4.2 描述的 commit 方法。）

这样的类可以按下面例子中的方式使用：

```
  class Person {                                    // Fragments
    // ...

    protected final SynchronizedInt age = new SynchronizedInt(0);

    protected final SynchronizedBoolean isMarried =
                              new SynchronizedBoolean(false);

    protected final SynchronizedDouble income =
                              new SynchronizedDouble(0.0);

    public int getAge() { return age.get(); }

    public void birthday() { age.increment(); }

    // ...
  }
```

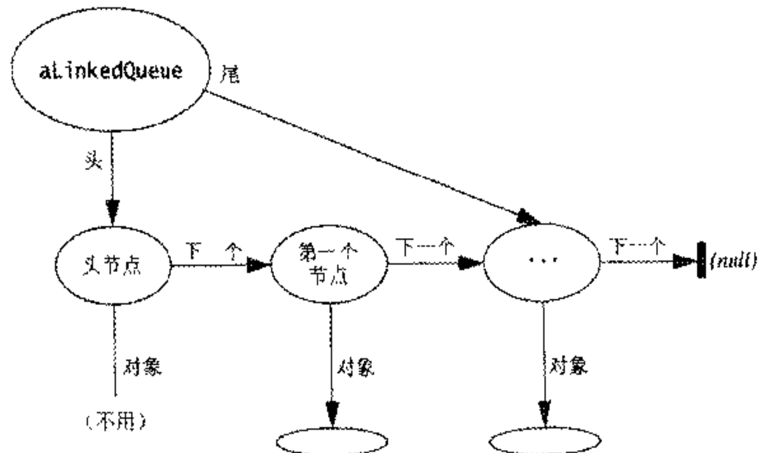
2.4.2.4　链表数据结构（Linked data structure）

对于作为链表数据结构入口点的对象来说，类分解策略可以通过下面的方法使得对该对象的访问竞争达到最小：在入口类的完全同步化（这样会限制并发量）和链表中所有节点完全同步化（这种方法效率低，而且会引起活跃性问题）两个极端策略之间寻找一个中间点。

所有锁分解技术的主要目的都是把不同的锁和不同的方法集联系起来。但是在链表数据结构中，通常会导致对数据结构和算法本身的进一步调整。对于控制链表数据结构访问的类，没有进行同步分解的普遍应用的方法，但下面的类列举了一些通用的技巧。

下面的 LinkedQueue 类可以作为普通的无界先进先出队列（first-in-first-out，FIFO）。它对 put 和 poll 操作分别维持同步。putLock 的锁保证一个时候只可以进行一个 put 操作。pollLock 的锁同样保证一个时候只可以进行一个 poll 操作。在这种应用中，通常有一个头节点，使得 put 和 poll 操作可以独立地进行。而在每一个 poll 操作之后，以前的第一个节点变成新的头节点。另外，当 put 和 poll 同时对一个空队列或即将成为空的队列操作的时候，对于被访问的节点本身必须加锁以防止冲突，因为在这种情况下，head 和 last 指向同一个头节点。

java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！




```
class LinkedQueue {
  protected Node head = new Node(null);
  protected Node last = head;

  protected final Object pollLock = new Object();
  protected final Object putLock = new Object();

  public void put(Object x) {
    Node node = new Node(x);
    synchronized (putLock) {       // insert at end of list
      synchronized (last) {
        last.next = node;          // extend list
        last = node;
      }
    }
  }

  public Object poll() {           // returns null if empty
    synchronized (pollLock) {
      synchronized (head) {
        Object x = null;
        Node first = head.next;    // get to first real node
        if (first != null) {
          x = first.object;
          first.object = null;     // forget old object
          head = first;            // first becomes new head
        }
        return x;
      }
    }
  }

  static class Node {              // local node class for queue
    Object object;
    Node next = null;

    Node(Object x) { object = x; }
  }
}
```


java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

在线文档中包含一些队列类，它们对这个基本的设计进行了提炼、扩展和优化。

2.4.3 只读适配器

在基于限制的设计中（参见 § 2.3.3），Host 对象不能泄露它的任何 Part 对象的标识。这一点使得对于任何存取访问或属性检查方法来说，都不能返回对 Part 对象的引用。

一个替代办法是返回 Part 对象的拷贝。例如在 SynchedPoint（参见 § 2.3.3）类中可以增加一个方法：

```
public synchronized BarePoint getPoint() {
  return new BarePoint(delegate.x, delegate.y);
}
```

当 Part 对象是某个实现了 clone 方法的类的实例时，程序员可以返回 part.clone()。如果需要返回数据集，则可以使用一个特殊的数组，例如：

```
public synchronized double[] getXY() {
  return new double[] { delegate.x, delegate.y } ;
}
```

但是，在处理一些对象的时候，进行拷贝可能很昂贵，而在处理其他对象的时候进行拷贝又没有意义。例如，保存本身不应该被拷贝的文件、线程或者其他资源的引用。很多情况下，程序员可以通过创建和返回 Part 对象的适配器来选择性地适当允许一些遗漏。适配器只向外提供客户需要使用且不会导致潜在干扰的相关操作——一般来说都是只读操作。除非这些方法只处理不变状态，否则就需要同步。

这种策略最安全的版本可以如下建立：

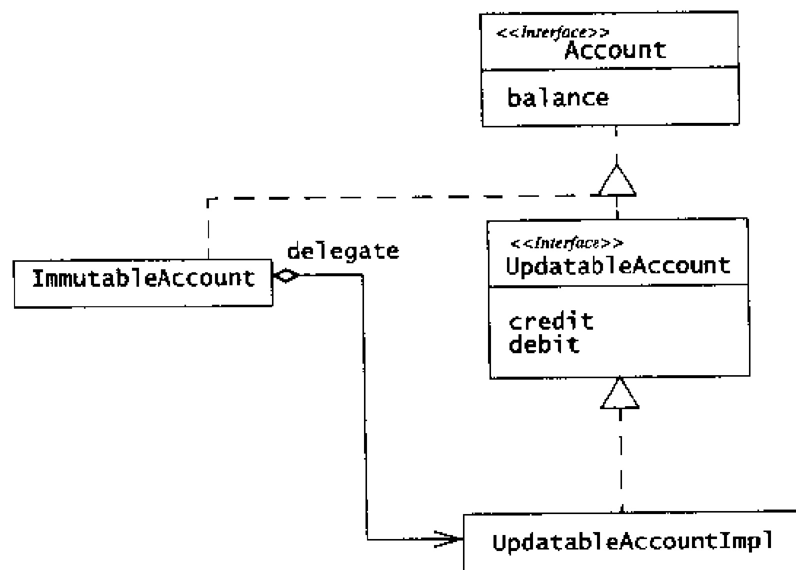
- 定义一个基本的 interface，用其来描述一些不变性的功能。
- 作为可选功能，可以定义一个子接口来支持在通常可变的实现类中用于更新的方法。
- 定义一个只读适配器，该适配器只传递接口中定义的操作。为了增加安全性，可以把不变类声明为 final。使用 final 成员变量表明，当你认为拥有一个不变对象的时候，你确实就拥有了一个（它不会是一个支持可变操作的子类）。

这些步骤可以应用于下面简单的 Account 类中。在这个例子中，尽管账号信息只保存在 AccountHolders 中，但是在可修改的 UpdatableAccount 的实现中使用了同步。

```
class InsufficientFunds extends Exception {}

interface Account {
  long balance();
}

interface UpdatableAccount extends Account {
  void credit(long amount) throws InsufficientFunds;
  void debit(long amount) throws InsufficientFunds;
}

// Sample implementation of updatable version
```


java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

```
class UpdatableAccountImpl implements UpdatableAccount {
  private long currentBalance;

  public UpdatableAccountImpl(long initialBalance) {
    currentBalance = initialBalance;
  }

  public synchronized long balance() {
    return currentBalance;
  }

  public synchronized void credit(long amount)
   throws InsufficientFunds {
    if (amount >= 0 || currentBalance >= -amount)
      currentBalance += amount;
    else
      throw new InsufficientFunds();
  }

  public synchronized void debit(long amount)
   throws InsufficientFunds {
    credit(-amount);
  }
}

final class ImmutableAccount implements Account {
  private Account delegate;

  public ImmutableAccount(long initialBalance) {
    delegate = new UpdatableAccountImpl(initialBalance);
  }

  ImmutableAccount(Account acct) { delegate = acct; }

  public long balance() { // forward the immutable method
    return delegate.balance();
  }
}
```



例如，这些类可以应用于：

java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

```
class AccountRecorder { // A logging facility
  public void recordBalance(Account a) {
    System.out.println(a.balance()); // or record in file
  }
}

class AccountHolder {
  private UpdatableAccount acct = new UpdatableAccountImpl(0);
  private AccountRecorder recorder;

  public AccountHolder(AccountRecorder r) {
    recorder = r;
  }

  public synchronized void acceptMoney(long amount) {
    try {
      acct.credit(amount);
      recorder.recordBalance(new ImmutableAccount(acct));//(*)
    }
    catch (InsufficientFunds ex) {
      System.out.println("Cannot accept negative amount.");
    }
  }
}
```

在有*号的一行使用只读方式的包装类看起来好像多余。但是这样做可以保证当某人编写了下面这样的子类，并将其和 AccountHolder 连接到一起后，该程序仍然可用：

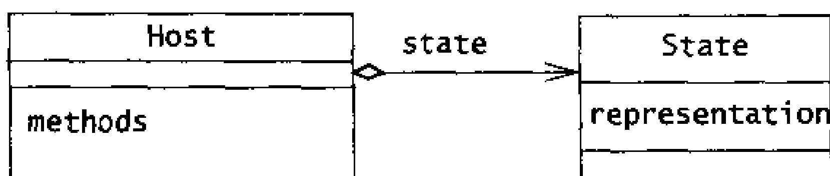
```
class EvilAccountRecorder extends AccountRecorder {
  private long embezzlement;
  // ...
  public void recordBalance(Account a) {
    super.recordBalance(a);

    if (a instanceof UpdatableAccount) {
      UpdatableAccount u = (UpdatableAccount)a;
      try {
        u.debit(10);
        embezzlement += 10;
      }
      catch (InsufficientFunds quietlyignore) {}
    }
  }
}
```

在 java.util.Collection 框架结构中使用了这种方法的一个变体。由于主 Collection 接口允许可变方法抛出 UnsupportedOperationExceptions 异常，所以这里没有声明单独的接口。对于所有试图进行数据更新的操作，匿名只读适配器都会抛出这些异常。下面是它的使用示例：

```
List l = new ArrayList();
// ...
untrustedObject.use(Collections.unmodifiableList(l));
```


java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

2.4.4 写拷贝（Copy-on-Write）



如果一组包含对象状态的成员变量间需要维护相互的不变约束，那么程序员可以将这些成员变量单独放在一个对象中，从而继续保证语义上的不变约束。

这样做需要基于这样一个事实，即：不变表示对象一直维护着合法对象状态的一致快照。依赖不变约束避免了对相关属性的多个读操作间的协调。这样做也避免了对客户隐藏这些表示的需要。

例如，在§1.1.1.1 中，我们必须特别小心地使用同步块技术来保证 Particles 类的(x,y)坐标被正确地显示。而在§2.4.2 中描述的 Shape 类甚至没有提供用来保证这一点的机制。一种解决方法就是利用单独的 ImmutablePoint 类来保存位置信息，而这些信息在任何时候都是一致的。

```
class ImmutablePoint {
  private final int x;
  private final int y;

  public ImmutablePoint(int initX, int initY) {
    x = initX;
    y = initY;
  }

  public int x() { return x; }
  public int y() { return y; }
}
```

ImmutablePoint 类可以用在下面的 Dot 类中，否则 Dot 类就和§1.1.1.1 中的 Particle 类类似。这个类展示了与使用写拷贝更新数据（即：状态改变不直接更新成员变量，而是创建并添加一个新表示对象）相关的常用技术。

注意：这里需要某种形式的同步。即使点的表示对象是不变的，loc 引用也是可变的。尽管根据§2.4.1 中的讨论，对访问方法 Location 的同步可以不必太严格，但 shiftX 必须声明为 synchronized 方法（否则的话，对这个方法就要进行相应的修改），从而防止在多个并发操作访问 loc.x()和 loc.y()的时候使用不同版本的 loc。

```
class Dot {
  protected ImmutablePoint loc;

  public Dot(int x, int y) {
    loc = new ImmutablePoint(x, y);
  }
```


java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

```
  public synchronized ImmutablePoint location() { return loc; }

  protected synchronized void updateLoc(ImmutablePoint newLoc) {
    loc = newLoc;
  }

  public void moveTo(int x, int y) {
    updateLoc(new ImmutablePoint(x, y));
  }

  public synchronized void shiftX(int delta) {
    updateLoc(new ImmutablePoint(loc.x() + delta,
                                 loc.y()));
  }
}
```

2.4.4.1 内部写拷贝

如果状态表现形式严格限制在一个对象的内部，那就不必为加强不变访问而创建新的类。每当需要快速轻易地获得一致的表现形式，同时又不希望有创建对象的开销时，就可以使用写拷贝技术。使用写拷贝，最多使用一次同步操作就可以访问到不可变的对象表现形式的所有状态。另外，有些时候仅使用单一的快照，要比获得在此快照使用过程中所有的状态修改情况方便得多。

例如，写拷贝集合对象在事件和多点传送框架（参见 § 4.1）中用来维护监听器的集合。这里的对象必须要维护一个监听器或其句柄的列表，这些监听器必须接收状态改变的通知和其他感兴趣的事件。这个列表很少改变，但是会被经常访问。并且，当对象接收通知需要对监听器列表进行修改时，这种改变通常都是在下一次通知发生时才生效，而不是本次就生效。

尽管对下面的数据结构来说，有很多其他的好方法实现（包括在 java.awt.EventMulticaster 中使用的特殊目的、基于树的结构和维护对在一个共同的基础结构上编辑的记录的更加精细的结构），但是基于数组的写拷贝集合类还是适合众多的应用。通过迭代变量来遍历不仅速度快，还可以避免用其他方式进行遍历时发生的 ConcurrentModificationExceptions 异常（参见 § 2.2.3）。

```
class CopyOnWriteArrayList {                          // Incomplete
  protected Object[] array = new Object[0];

  protected synchronized Object[] getArray() { return array; }

  public synchronized void add(Object element) {
    int len = array.length;
    Object[] newArray = new Object[len+1];
    System.arraycopy(array, 0, newArray, 0, len);
    newArray[len] = element;
    array = newArray;
  }

  public Iterator iterator() {
    return new Iterator() {
      protected final Object[] snapshot = getArray();
      protected int cursor = 0;
```


java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

```
        public boolean hasNext() {
          return cursor < snapshot.length;
        }

        public Object next() {
          try {
            return snapshot[cursor++];
          }
          catch (IndexOutOfBoundsException ex) {
            throw new NoSuchElementException();
          }
        }
      };
    }
  }
```

（在线提供的 util.concurrent 包中包含了这个类的一个版本，它和 java.util.List 接口一致）如果把这个类应用在对大集合进行频繁修改的情况下，其效率可能低得惊人，但是对于在 § 3.5.2 和 § 3.6.4 中列举的多数多点广播应用却很适合。

2.4.4.2 乐观更新

乐观更新（optimistic update）采用比其他写拷贝技术更弱的协议：它并不是把整个状态更新方法锁住，而只是在状态更新的开始和结束的时候使用同步。每个方法通常都有下面的形式：

1. 得到当前状态表示的一个拷贝（在持有锁的时候）。
2. 创建一个新的状态表示（不持有任何锁）。
3. 只有在老状态在被获取后且没有被更新的情况下，才能被***转换*** 成新的状态。

乐观更新技术把同步限制在很短的一瞬间——只是足够用来访问和随后对状态表示更新的时间。这在多处理器上可以提供很好的性能，至少在被恰当使用的时候是这样。

在写拷贝技术基础上增加的主要需求是在处理步骤三中可能出现的失败。因为在当前线程有机会更新状态表示之前，其他线程就已经擅自把它更新了。这种潜在的失败导致了限制乐观更新技术使用范围的两点考虑（在 § 3.1.1 中做详细讨论）：

失败协议。失败的时候，或者选择对整个方法序列重试，或者把失败返回给客户端，这样客户端可以采取相应的操作。最普通的操作就是重试，但是这种方法可能会导致***活锁***——类似于无限的挂起，方法不断循环却不做进一步的操作。尽管出现活锁的可能性很小，但是活锁会导致操作永远不会结束，并会为此持续不断地消耗 CPU 资源。由于这个原因，当在任意长的时间里可能出现很多线程竞争时，使用乐观更新技术并不是好的选择。但是，在一些特殊的***无等待*** 乐观算法中，无论是否有竞争，在尝试过一定的次数后，就被认为操作成功而继续往下运行（参见“进阶阅读”）。

副作用。因为会导致失败，所以在创建新状态表示过程中执行的操作，不能带来任何不能取消的副作用。例如：不能向写入文件，创建线程，或者在 GUI 中画图，除非这些操作在出错的情况下可以有效地被***取消***（参见 § 3.1.2）。

java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

2.4.4.3 原子性提交

所有乐观技术的核心都是使用原子性提交（atomic commitment）来代替赋值语句。只有现有的状态表示是调用者所期望的才可以有条件地转换到新的状态表示。有很多方法都可以用来区分和跟踪不同的状态表示，例如：使用版本号、事务标识符、时间戳和签名代码等，但是至今为止最简单也是最常用的方法就是简单依赖于状态对象的引用标识符。下面是一个普通的例子：

```
class Optimistic {                            // Generic code sketch

  private State state; // reference to representation object

  private synchronized State getState() { return state; }

  private synchronized boolean commit(State assumed,
                                      State next) {
    if (state == assumed) {
      state = next;
      return true;
    }
    else
      return false;
  }
}
```

在 commit 方法如何定义方面通常有一些较小的改变，例如：compareAndSwap 命名的版本返回当前的值，这个值可以是新值或旧值，取决于操作提交是否成功。在系统级的并发编程中，乐观技术的日益流行要部分归功于（也部分导致）这样一个事实：很多现代处理器都包含有一个高效的内嵌 compareAndSwap 指令或其变体。尽管从 Java 编程语言不能直接访问这些指令，但是原则上可以优化编译器，通过映射构件来使用这些指令（即使不能这样，优化算法还是有效的）。

在一个纯乐观类中，很多更新方法都使用标准形式：得到初始状态，创建新的状态表示，然后如果可能就提交，否则就循环或者抛出异常。但是不依赖于任何初始状态的方法可以实现得更为简单，即无条件地变换成新状态。例如，下面是 Dot 类的一个乐观版本：

```
class OptimisticDot {
  protected ImmutablePoint loc;

  public OptimisticDot(int x, int y) {
    loc = new ImmutablePoint(x, y);
  }

  public synchronized ImmutablePoint location() { return loc; }

  protected synchronized boolean commit(ImmutablePoint assumed,
                                        ImmutablePoint next) {
    if (loc == assumed) {
      loc = next;
      return true;
    }
```


java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

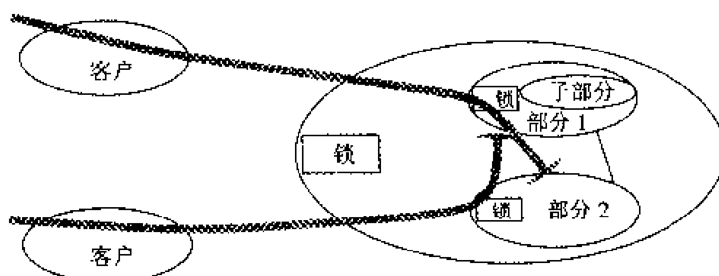
```
      else
        return false;
    }

    public synchronized void moveTo(int x, int y) {
      // bypass commit since the operation is unconditional
      loc = new ImmutablePoint(x, y);
    }

    public void shiftX(int delta) {
      boolean success = false;
      do {
        ImmutablePoint old = location();
        ImmutablePoint next = new ImmutablePoint(old.x() + delta,
                                                 old.y());
        success = commit(old, next);
      } while (!success);
    }

  }
```

如果潜在的长时间干扰是一个需要考虑的因素，那么在 shiftX 中的循环就可以使用§3.1.1.5 中讨论的指数回退策略。

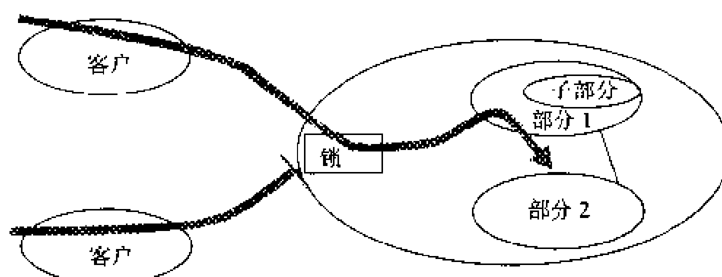
2.4.5 开放容器

如果使用分层包含设计（参见§2.3.3），但是又不能，或不想把所有 Part 对象都对其他客户隐藏，就可以使用排序的***层次式***锁技术。

如果 Part 对象对客户可见，就必须使用同步。但是如果部分对象频繁调用其他部分对象的方法，那么这种设计就很容易导致死锁。例如：若一个线程持有 part1 的锁，同时想对 part2 做调用，而另一个线程做着相反的操作：



可以使用严格的对象制设计中的策略来消除这种死锁现象：Part 对象依赖 Host 上的锁来实现它们的同步控制。如果客户端必须先获得 Host 对象的锁，这种形式的死锁就不会发生。



java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

这种方法可以满足大多数包含可见组件的包容设计（其他的设计变体可以参见 § 2.5.1.3）。在对部分对象进行操作之前首先要获得最外层容器的锁，这种思想代表了在 § 2.2.6 中讨论的有关资源顺序化技术结构化的应用。当然，如果没有限制，也就没有保障这种实现方案的简单策略。类（和其创造者）必须了解并且遵守这些规则。策略选择主要考虑谁应该知道这些规则，是内部的部分对象，还是外部客户。虽然这两种选择都不是完美的，但是必须选择一个。

- 使用内部锁会使得对已经存在类的更新比较难，并且会增加类对其上下文的依赖。
- 当任何一个客户忘记使用协议的时候，外部锁都会失败。

2.4.5.1　内部规则

在使用内部包容锁的情况下，每一个 Part 对象都对其需要动态独占控制的方法使用其容器的同步锁。在效率最高的情况下，每一个 Part 对象都有一个 final 成员变量，它在 Part 对象创建的时候初始化并且在需要使用锁的情况下都使用它。也可以接受 Part 对象方法内其他不相关的锁（参见 § 3.3.4）。

例如：

```
class Part {                                    // Code sketch
  protected final Object lock;
  // ...

  public Part(Object owner) { lock = owner; }

  public Part() { lock = this; } // if no owner, use self

  public void anAction() {
    synchronized(lock) {
      anotherPart.help();
      // ...
    }
  }
}
```

作为一种设计策略，程序员可以把大多数或者所有的类都定义成这个样子，以便在各种各样的基于容器的框架中使用。但是，如果一个 Part 对象的所有关系动态都被改变时，这种设计策略将很难使用。这种情况下，程序员必须先对作为锁使用的成员变量做额外的同步访问控制（通常使用 synchronized(this)），然后再使用它来对方法体做访问控制。

如果可以把每个 Part 类都声明为 Host 类的***内部*** 类，那么有一种比较简单的用于实现内部包容锁定的结构。这种情况下可以使用 Host.this 作为 synchronized 同步块的参数：

```
class Host {                                    // code sketch
  // ...

  class Part {
    // ...
    public void anAction() {
      synchronized(Host.this) {
        anotherPart.help();
```


java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

```
            // ...
          }
        }
      }
    }
```

2.4.5.2 外部规则

在极端条件下，对于非结构化的外部锁来说，对每个 Part 对象的每个方法的每次调用，调用者都必须在调用之前知道自己要得到哪个锁：

```
synchronized(someLock) { aPart.anAction(); }
```

在有限闭合系统中，开发者甚至可以创建一个列表，用于定义哪个锁和哪个对象相关联，并且要求代码的编写者遵循这些规则。这种方法对于小型的、不可扩展的嵌入系统来说可以防止死锁。

当然，这种方法的扩展性并不好。在规模稍大一些的情况下，客户端代码就必须使用编程来决定自己要获得哪个锁。通过创建一个表来保存对象和锁之间的关系，可以解决这个问题。更结构化一些的方法是：在每个 Part 对象中包含一个方法，称为 getlock 方法，该方法返回同步控制所需要的锁。客户端可以像下面这样进行调用：

```
synchronized(aPart.getLock()) { aPart.anAction(); }
```

java.awt 包中就使用了这种方法（至少到 § 1.2 版本之前是这样）。每一个 java.awt.Component 都支持 getTreeLock 方法，该方法返回对当前容器（例如：一个 Frame）进行同步控制所需要的锁。何时和如何使用这个锁留由客户端代码决定。这样就引入了特殊的扩展性的可能，和内部锁原则相比，这还会带来一点点的性能提高。例如：一个客户得知自己已经持有了锁就不需要重新获取。但是这种外部锁控制方法很容易产生错误，并且需要大量的文档支持——客户必须对操作有足够了解后才能决定是否可以使用锁，以及如何使用。

2.4.5.3 多层包含

两种层次式包容锁方法都可以扩展到大于两层的情况，每层都有一个相关的锁。代码在调用更新方法之前，必须以从外到内的顺序持有所有层的锁。用 synchronized 块或方法来建立任意层数的嵌套锁显得十分笨拙，但是用锁工具类或者管理器也可能实现。

2.4.6 进阶阅读

有关重构类的一般说明可以参见：

Fowler 和 Martin.《Refactoring》，Addison-Wesley, 1999.

总可以成功继续运行的乐观更新算法包括实现了***无等待***算法的类，这时，线程不会总是等待需要依靠其他线程的某些操作才能够变为真的条件判断。在无等待算法中，每个线程在经过有限次尝试之后都会成功，而不管其他线程做了或没做什么。很多算法都使用***帮助***的概念：如果一个线程自己不能继续，它会采取一些行动来帮助另一个线程完成任务。无等待算法理论的描述请参见：

java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

Herlihy 和 Maurice. "Wait-free synchronization",《ACM Transactions on Programming Languages and Systems》, vol. 13, no. 1, 1991.

实际上，对于通常使用的数据结构来说，只有很少的几种可以使用无等待更新算法，例如：队列和列表。但是，这些算法在操作系统内核和 JVM 对底层运行时的支持中应用得越来越多。在线帮助包含了下面文章中的无等待队列的一个改进版本，以及用其他语言实现的自由等待算法的链接。

Michael, Maged 和 Michael Scott. "Simple, fast, and practical non-blocking and blocking concurrent queue algorithms", 《Proceedings, 15th ACM Symposium on Principles of Distributed Computing》, ACM, 1996.

2.5 使用锁工具

内部的 synchronized 方法和块可以满足很多基于锁的应用，但是有下面的限制：

- 如果某线程试图得到锁，而这个锁已经被其他线程持有，那么就没有办法回退，也没有办法在等待一段时间之后放弃等待，或者在某个中断之后取消获取锁的企图。这些使得线程很难从活跃性问题中恢复。
- 没有办法改变锁的语义形式，例如重入性、读和写保护、或公平性等方面的改变。
- 没有对同步的访问控制。任何一个方法都可以对其可访问的对象执行 synchronized (obj)操作，这样导致由于所需要的锁已被占用引起的拒绝服务问题。
- 方法和块内的同步，使得只能够对严格的块结构使用锁。例如：不能在一个方法中获得锁，而在另一个方法中释放锁。

这些问题可以通过使用工具类来控制锁的方式克服，这样的类可以用 § 3.7 中描述的技巧来创建。这里，我们把目标限制在使用这些工具类实现基于锁的设计。尽管可以创建锁类，使其提供任何想要的语法和使用属性，但我们在这里只讨论两个通用类，互斥独占锁和读写锁。为了更加具体，我们使用了从在线文档中得到的 util.concurrent 包中的类的版本。当然，这里的讲述对任何可能被创建的锁工具类都适用。

本章前面提到的所有基于锁的设计方法，（如果希望的话）都可以重新用锁工具创建，而不是使用内建的 synchronized 方法或者块（在 § 1.2.5 中列出的有关并行系统编程的文献中，可以发现很多其他的例子）。本节的重点是讲述使用中的难点。

锁工具类提供的解决方案是以笨拙的代码模式和更少的对使用正确性的自动保证为代价的。比起 synchronized 方法或者块来说，使用锁工具时需要更加小心，并且要严格遵守相应的规则。这些结构可能需要更大的开销，因为它们比内建的同步更难优化。

2.5.1 Mutex

一个 Mutex 类［互斥独占锁（mutual exclusion lock）的缩写］可以定义为（省略了方法实现代码）：

java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

```
public class Mutex implements Sync {
  public void acquire() throws InterruptedException;
  public void release();
  public boolean attempt(long msec) throws InterruptedException;
}
```

（在 util.concurrent 包中的版本，Mutex 实现了 Sync 接口。它是为遵循获得-释放协议的类而定义的标准接口。）

正如你所希望的，acquire 和同步块的入口操作相似，release 和同步块的释放锁操作相似。Attempt 操作只有在规定的时间内得到锁才会返回 true（在实现中至少需要尽量准确地测量这个时间，并及时做出反应——参见 § 3.2.5）。零是合法的参数，这表明如果得不到锁的话根本不需要等待。

和内建的同步机制不同的是，如果当前的线程在试图获得锁的过程中被中断，acquire 和 release 方法会抛出 InterruptedException 异常。这一点增加了使用的复杂性，但是提供了编写响应性良好的健壮代码来处理取消操作的机制。我们将在 § 3.1.2 中详细讨论对 InterruptedException 异常合理的处理范围。这里只列举几个最基本的选择。

可以替换下面的形式，Mutex 可以像内建锁（built-in）一样使用：

```
synchronized(lock) { /* body */ }
```

替换为下面的更冗余、更笨拙的 before/after 结构：

```
try {
  mutex.acquire();
  try {
    /* body */
  }
  finally {
    mutex.release();
  }
}
catch (InterruptedException ie) {
  /* response to thread cancellation during acquire */
}
```

和 synchronized 块不同的是，标准 Mutex 类中的锁不能重入。如果锁已经被执行 acquire 操作的线程所持有，这个线程还会在 acquire 操作上被阻塞。当然，再定义并使用一个 ReentrantLock 类也是可能的，但是一个简单的 Mutex 类就可以满足很多的锁应用。例如，我们可以使用它重新实现 § 1.1.1.1 中的 Particle 类：

```
class ParticleUsingMutex {
  protected int x;
  protected int y;
  protected final Random rng = new Random();
  protected final Mutex mutex = new Mutex();

  public ParticleUsingMutex(int initialX, int initialY) {
    x = initialX;
    y = initialY;
  }
```


java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

```
    public void move() {
      try {
        mutex.acquire();
        try {
          x += rng.nextInt(10) - 5;
          y += rng.nextInt(20) - 10;
        }
        finally { mutex.release(); }
      }
      catch (InterruptedException ie) {
        Thread.currentThread().interrupt();
      }
    }

    public void draw(Graphics g) {
      int lx, ly;

      try {
        mutex.acquire();
        try {
          lx = x; ly = y;
        }
        finally { mutex.release(); }
      }
      catch (InterruptedException ie) {
        Thread.currentThread().interrupt();
        return;
      }

      g.drawRect(lx, ly, 10, 10);
    }
  }
```

操作体周围的 try/catch 结构模拟了 synchronized 块中的下列行为：即无论操作体如何退出都会释放锁，即使是因为一个没有捕捉到的异常而退出也不例外。作为一个设计原则，要经常使用 try/catch 语句，即使不会抛出什么异常也要这么做，这是一个好习惯。

如果线程在获得锁的过程中被中断，move 和 draw 方法都会不执行任何操作而立即返回。对于取消操作来说，这是一个简单有效的响应。尽管如此，正如 § 3.1.2 中将要讨论的，catch 语句也必须通过 Thread.currentThread().interrupt() 方式传递取消状态。

和原来的类相比，ParticleUsingMutex 类对恶意的拒绝服务现象更具抵抗力。因为不使用内建的 synchronized 块，所以谁持有锁都无所谓（注意：在 ParticleApplet 类中不会出现这种问题，因为所有的引用都限制在 applet 内部）。如果做进一步的假设，我们甚至可以把 Mutex 声明为 private，但是这么做对扩展性并没什么好处。因为任何一个合理的子类都需要访问锁，把 mutex 声明为 private 就几乎等价于把这个类本身声明为 final。

2.5.1.1 方法适配器

更好的有关锁的结构和规则可以通过 § 1.4 中讨论的任何一个 before/after 模式来实现。例如，使用方法适配器来支持通用的包装器的定义，实现在任何锁内运行任何代码。一个包

java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

装器或者定义为使用锁的类的方法，或者定义为一个单独的工具类。作为后者的例子：

```
class WithMutex {
  private final Mutex mutex;
  public WithMutex(Mutex m) { mutex = m; }

  public void perform(Runnable r) throws InterruptedException {
    mutex.acquire();
    try     { r.run(); }
    finally { mutex.release(); }
  }
}
```

这种方式可以被下面的类使用：它们以内部方法的形式把纯操作分离出来，再在公有方法实现的包装器内部调用这些纯操作，例如：

```
class ParticleUsingWrapper {                          // Incomplete
  // ...
  protected final WithMutex withMutex =
                                  new WithMutex(new Mutex());

  protected void doMove() {
    x += rng.nextInt(10) - 5;
    y += rng.nextInt(20) - 10;
  }

  public void move() {
    try {
      withMutex.perform(new Runnable() {
        public void run() { doMove(); }
      });
    }
    catch (InterruptedException ie) {
      Thread.currentThread().interrupt();
    }
  }
  // ...
}
```

这种设计从某种程度上来说开销更大，所以只在被保护的操作相对耗时的类中才使用。同时，这里给出的版本只能应用于可以表示为没有参数，且没有返回值的 Runnables 的操作。所以，不能应用于像 draw 这样的方法中。但是这种方法可以通过定义额外的方法适配器来扩展，用方法适配器处理、接收其他的参数和/或返回值的操作，这方面的内容参见 § 1.4.4。

2.5.1.2 回退（Back-off）

attempt 方法在从死锁以及和多个锁相关的其他活跃性问题中恢复的时候很有用。在无法避免死锁的时候（至少，很多开放系统中的一些组件都是这样），可以使用 attempt 而不用 acquire，并提供一个估计的超时时间（例如几秒）来指明可能的挂起时间，然后在失败时采取相应的措施（参见 § 3.2.5）。

attempt 操作也可以按更特殊的方式来处理某种容易死锁的创建。例如：这里有 § 2.2.5 中 Cell 的另一个版本，当发现潜在的死锁现象时，它就将回退重试。它在重试之间尝试着给

java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

出了一个短延迟。因为它采用的是“重试”，所以可能会导致活锁。如果程序员认为无限活锁现象比随机的硬件失效出现的概率低，这么做就是可以接受的。

注意，这里和很多类似的与非重入锁有关的创建中都使用了别名检查，以避免因为要获得已经获得的锁而产生的死锁现象（参见 §2.2.6）。

```
class CellUsingBackoff {
  private long value;
  private final Mutex mutex = new Mutex();

  void swapValue(CellUsingBackoff other) {
    if (this == other) return;   // alias check required here
    for (;;) {
      try {
        mutex.acquire();
        try {
          if (other.mutex.attempt(0)) {
            try {
              long t = value;
              value = other.value;
              other.value = t;
              return;
            }
            finally {
              other.mutex.release();
            }
          }
        }
        finally {
          mutex.release();
        };
        Thread.sleep(100);
      }
      catch (InterruptedException ie) {
        Thread.currentThread().interrupt();
        return;
      }
    }
  }
}
```

2.5.1.3 重新排序（Reordering）

如果在使用排序锁技术（参见 §2.2.6 和 §2.4.5）的设计中，某一种层次结构的排序锁会有最少的异常发生，那么回退操作是一个很好的用来找出这种特定层次结构的方法。在这种情况下，需要获得多个锁的代码可以以某种顺序获得锁，如果失败了就释放所有的锁，然后再以另一种顺序重试。这种方法可以扩展基于容器的锁机制的使用范围。如果程序员有更复杂的回退策略来对付异常事件，那么可以不必保证一定是按照某种希望的顺序获取锁。例如：在应用了回调或者集合遍历的层次化包容设计中，就可能出现这种情况。因为在这些情况下不能保证锁的访问顺序一定是按照给定的一组锁顺序规则集。

为了演示最基本的技术，下面的 Cell 类在 swapValue 中使用了避免死锁而排列组合锁的访问顺序的策略。当失败时，它会以相反的顺序锁住对象：

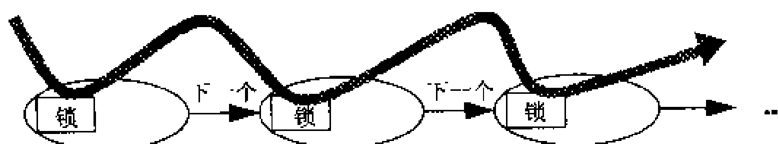
java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

```
class CellUsingReorderedBackoff {
  private long value;
  private final Mutex mutex = new Mutex();

  private static boolean trySwap(CellUsingReorderedBackoff a,
                                 CellUsingReorderedBackoff b)
   throws InterruptedException {
    boolean success = false;

    if (a.mutex.attempt(0)) {
      try {
        if (b.mutex.attempt(0)) {
          try {
            long t = a.value;
            a.value = b.value;
            b.value = t;
            success = true;
          }
          finally {
            b.mutex.release();
          }
        }
      }
      finally {
        a.mutex.release();
      }
    }
    return success;
  }

  void swapValue(CellUsingReorderedBackoff other) {
    if (this == other) return;      // alias check required here
    try {
      while (!trySwap(this, other) &&
             !trySwap(other, this))
        Thread.sleep(100);
    }
    catch (InterruptedException ex) {
      Thread.currentThread().interrupt();
    }
  }
}
```

2.5.1.4 非阻塞结构的锁

当获得/释放锁操作不能在同一个方法或者代码块中进行时，不能使用 synchronized 块，这时可以使用 Mutex。

例如，在链表的各节点之间进行遍历的时候，程序员可以使用 Mutex 实现锁传递（也称锁连接）。这种情况下，要求在持有当前节点锁的同时，获得下一个节点的锁。但是，在获得下一个节点的锁之后，就可以释放当前的锁了。



java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

锁传递遍历允许非常细粒度的锁，这样也就增加了潜在的并发性，但这是以额外的复杂性和负载为代价的，而这些代价只有在极度竞争资源的情况下才有意义。

```
class ListUsingMutex {

  static class Node {
    Object item;
    Node next;
    Mutex lock = new Mutex(); // each node keeps its own lock
    Node(Object x, Node n) { item = x; next = n; }
  }

  protected Node head; // pointer to first node of list

  // Use plain synchronization to protect head field.
  //  (We could instead use a Mutex here too but there is no
  //  reason to do so.)

  protected synchronized Node getHead() { return head; }

  public synchronized void add(Object x) { // simple prepend

    // for simplicity here, do not allow null elements
    if (x == null) throw new IllegalArgumentException();

    // The use of synchronized here protects only head field.
    // The method does not need to wait out other traversers
    // that have already made it past head node.

    head = new Node(x, head);
  }

  boolean search(Object x) throws InterruptedException {
    Node p = getHead();

    if (p == null || x == null) return false;

    p.lock.acquire();  // Prime loop by acquiring first lock.

    // If above acquire fails due to interrupt, the method will
    //   throw InterruptedException now, so there is no need for
    //   further cleanup.

    for (;;) {
      Node nextp = null;
      boolean found;

      try {
        found = x.equals(p.item);
        if (!found) {
          nextp = p.next;
          if (nextp != null) {
            try {          // Acquire next lock
                           //   while still holding current
              nextp.lock.acquire();
            }
```


java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

```
              catch (InterruptedException ie) {
                throw ie;      // Note that finally clause will
                               //   execute before the throw
              }
            }
          }
        }
        finally {          // release old lock regardless of outcome
          p.lock.release();
        }

        if (found)
          return true;
        else if (nextp == null)
          return false;
        else
          p = nextp;
      }
    }

    // ...  other similar traversal and update methods ...
  }
```

Mutex 的另外一个实现内建机制所不支持的、阻塞结构的应用是条件变量对象的构建，参见 § 3.4.4。

2.5.1.5 锁的顺序化管理器（Lock Ordering Manager）

当必须按照某个顺序获得锁时（例如 § 2.4.5 中讨论的层次化包容策略和 § 2.2.6 中讨论的通用顺序化资源技术），可以通过把顺序化方法集中在一个锁管理类中来保证与规则一致。

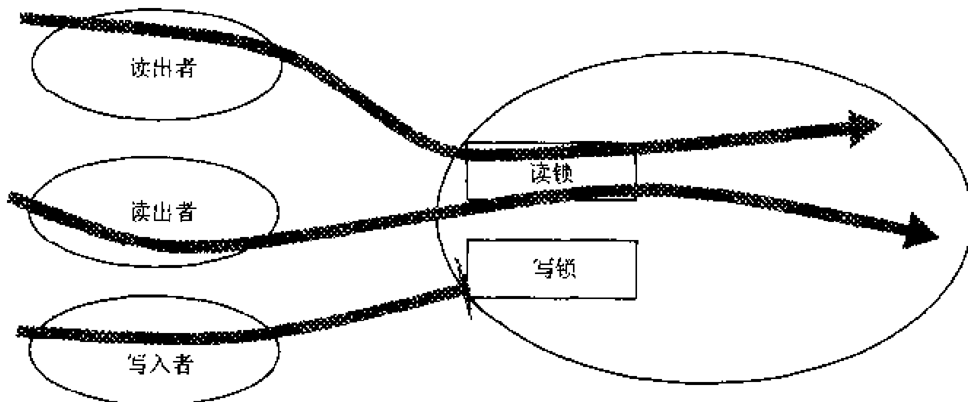
有特别多的方法可以组织使用的多种锁，定义顺序化规则，以及规定管理器类的责任。但是几乎所有的设计都包含有下列形式的方法，这种形式表明，程序员必须非常认真地确保无论在什么异常情况下，都要释放锁。

```
class LockManager {                         // Code sketch
  // ...
  protected void sortLocks(Sync[] locks) { /* ... */ }

  public void runWithinLocks(Runnable op, Sync[] locks)
   throws InterruptedException {

    sortLocks(locks);

    // for help in recovering from exceptions
    int lastlocked = -1;
    InterruptedException caught = null;

    try {
      for (int i = 0; i < locks.length; ++i) {
        locks[i].acquire();
        lastLocked = i;
      }

      op.run();
```


java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

```
        }

        catch (InterruptedException ie) {
          caught = ie;
        }

        finally {
          for (int j = lastlocked; j >= 0; --j)
            locks[j].release();

          if (caught != null)
            throw caught;
        }
      }
    }
```

2.5.2 读-写锁



ReadWriteLocks 中保存着一对相关的锁，下面是它们的一种定义：

```
    interface ReadWriteLock {
      Sync readLock();
      Sync writelock();
    }
```

这里的两个方法所返回的锁和 Mutex（参见 § 2.5.1）遵守着同样的 Sync 接口，支持 acquire、release 和 attempt 方法。

正如 § 3.3.3 中要讨论的那样，有很多方法都可以实现这个接口，到底使用哪种实现方法要根据具体的使用策略而定。为了说明问题，我们可以定义一个一般的实现类 RWLock。

读-写锁的设计思想是，只要没有写线程，readLock 就可以同时被多个读线程控制，writeLock 是独占的。在下面的情况下，读-写锁相对普通的锁来说是更好的选择：

- 类中的方法可以彻底地分为只访问（读）内部数据的方法，和只修改（写）内部数据的方法。
- 当写方法运行的时候，不允许读操作进行（如果写的时候还允许读，那么程序员可以使用非同步读方法，或者复制写作为更新数据的方法，参见 § 2.4）。
- 在应用的场景中，一般读操作多于写操作。

java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

- 由于这样的操作相对来说比较耗时，所以相比其他简单技术来说，读-写锁有一些额外的开销，但却可以允许多个线程同时读。

读-写锁通常应用于对大型集合数据进行访问的类中，这些类的方法结构类似于：

```
class DataRepository {                         // Code sketch

  protected final ReadWriteLock rw = new RWLock();

  public void access() throws InterruptedException {
    rw.readLock().acquire();
    try {
      /* read data */
    }
    finally {
      rw.readLock().release();
    }
  }

  public void modify() throws InterruptedException {
    rw.writeLock().acquire();
    try {
      /* write data */
    }
    finally {
      rw.writeLock().release();
    }
  }

}
```

读-写锁在普通的集合类应用中很有用。在线文档中提供的 util.concurrent 包中包含了一组适配器类，这些类可以和 java.util.Collection 类一起使用，在纯访问的方法中（例如 contains）放置读锁，在更新数据的方法中（例如 add）放置写锁。

2.5.3 进一步讨论

在下面的书中可以找到使用不同方式锁的一些模式：

McKenney 和 Paul. "Selecting Locking Primitives for Parallel Programming", 《Communications of the ACM》, 39(10): 75-82, 1996.

java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

第3章

状态依赖

执行任何操作通常都有两个必要条件：

外部条件：对象收到了执行操作的请求。

内部条件：对象处于可进行操作的状态。

举一个编程之外的例子：有人让你去做电话记录，为此，你必须有纸和铅笔（或者其他记录设备）。

独占技术主要关注于维持不变约束，而状态依赖的并发控制更加关注前提条件和结束条件。当客户调用主体对象的方法时，这个操作不一定具有需要的基于状态的前提条件。反过来说，当主体对象并未处于合适的可操作状态，或者其依赖的其他对象的操作不能满足自身的结束条件时，或者其他线程的操作改变了主体对象所依赖对象的状态时，主体对象的操作就可能会无法满足结束条件。

大多数有状态依赖操作的类的设计问题都要考虑如下因素：如何顾及到所有可能出现的消息和状态的组合，从而使得设计具有完整性。比如在这个例子中：

	有铅笔	没有铅笔
电话铃响	接电话	接电话
记录消息	写下消息	?

正如表格所示，这些设计通常都要考虑当对象并不处在可正常执行操作的状态时的情形。在一个理想的系统中，所有方法都不应有基于状态的前提条件，并且总是能够满足结束条件。只要有可能，我们就应该按这种方式来编写类和方法，这样，几乎能避免本章讨论的所有问题。但是很多行为的本质就是依赖于状态的，并且无法被编写成在所有状态下都能满足结束条件。

要设计基于状态的操作并实现它通常有两种方法，它们分别源自于活跃性优先和安全性优先的设计角度。

乐观的“先试再看”的方法在被调用时总是可以被试着执行的，但其并不能保证操作成功，因此调用者有可能需要处理调用失败时的状况。

保守的“先测再做”的方法则只有在满足了前提条件时候，才执行相应操作。因此，一旦满足了前提条件，操作就一定能够成功。

java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

如果方法既不检查前提条件也不检查结束条件，那么它们只能在肯定满足了前提条件的上下文中被调用。至少在并发系统中，采用这个做法是有问题的。

乐观法和保守法都使用得相当普遍，并且对于不同的设计而言，只要采用了恰当的形式，上述两种方法所达到的效果可能同样有效。但是，由于这两种方法的一般形式的决定因素往往难以控制，所以，它们很难互换。乐观法依赖于异常机制以及当结束条件无法满足时的相关提示机制的存在。保守法依赖于那些提供保障的构件，这些构件能够在操作所需的前提条件满足时发出提示，并保证在执行过程中，该条件一直满足。当然，这两种方法的混合形式也是可行的，而且在实际操作中相当普遍。要特别指出的是，很多使用保守法的设计中也包含了一些可能会遭遇异常的代码，因此，保守法也需要处理失败的情况。

在并发编程中，那些处理和状态依赖的操作相关的方法非常耗费精力，且需格外注意。本章所讨论的内容分为以下几部分：

- § 3.1　讨论异常及取消。
- § 3.2　介绍用于在保守法中提供保障的构件以及实现它们的机制。
- § 3.3　为使用并发控制的类提供了一些结构模式。
- § 3.4　展示了工具类是如何在减少复杂性的同时提高程序的可靠性、性能，以及灵活程度的。
- § 3.5　讲述了与协同操作有关的问题及解决方案——那些依赖多个参与者状态的操作。
- § 3.6　对事务性并发控制进行简要介绍。
- § 3.7　总结了一些构建并发控制工具类的常见技巧。

3.1　处理失败

纯粹的乐观法源自乐观的更新和事务协议。但有时某些被称为乐观法的程序也仅仅是因为在它们的代码中调用了可能会失败的方法。“先试再看”的设计在尝试执行操作时并不先确保这些操作会成功，这通常是由于那些保证操作成功的约束条件很难检测造成的。但是，乐观法一定会去检测结束条件的（通常是通过捕捉失败异常来实现），当发现结束条件无法满足时，就执行相应的失败处理策略。

对“先试再看”的方法的需求往往是因为我们无法或者不愿去检测前提条件及其相关约束。这通常是由以下这几种原因造成的：

- 某些条件无法通过使用给定的语言或执行环境中的构件来计算。比如，无法去检查得到的锁是不是正在被别人使用或者得到的引用是不是惟一的（参见 § 2.3）。
- 在并发程序中，前提条件可能会有临时范围（有时也被称作活动约束）。如果一个约束不在主体对象的控制之下，即使这个约束是暂时满足的，但是，在依赖于这个约束的操作的执行过程中，它未必能一直能被满足。比如，你在写下消息的时候铅笔突然断了；某个方法在写入文件之前得知文件系统的空间是足够的，但是在写入的过程中却发现磁盘空间已满（可能是其他程序的操作所导致），类似的还有，一

java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

台远程机器当前可用并不意味着在执行一个依赖于它的方法的过程中，这台机器不会出现死机或者无法访问的情况。

- 某些条件的变化是由其他线程所发出的信号导致的。最常见的例子就是取消操作的状态，它在任何线程执行任何操作的过程中，这个状态都有可能异步地变为真（参见 § 3.1.2）。
- 检测某些约束所需的计算量过大。比如，要求将矩阵转换为标准的上三角的形式的约束。当操作很容易取消或者其失败的几率很低时，计算这些简单的前提条件可能就没有必要了，取而代之的是，在以后发现失败时应用相应的撤销策略。

在所有这些例子中，由于缺乏对那些保证成功的条件的检测，导致了这些方法必须检测并处理那些潜在的无法满足结束条件的情况。

3.1.1 异常

在多线程的程序设计中，容错处理无处不在。并发带来这样可能性：程序的一部分失败了而其余的部分还在正常运行。但是，如果不小心的话，一个失败的操作可能使对象处于某种状态，而这种状态会导致其他线程也无法成功运行下去。

当方法检测到无法达到预计的效果或者结束条件时，它们就会抛出异常（同等的操作有：设置相应的状态值或发出失败通知）。通常，处理这些失败操作的方法有以下六种：突然中断、继续（忽略失败）、回滚、前滚、重试以及委托给其他异常处理器。突然中断和继续是最为极端的两个方法。回滚或者前滚是通常选择，因为它们可以保持对象状态的一致性。重试会在局部包含失败点。委托则能允许多个对象和活动协同处理这些失败的操作。

选择哪一种方法来处理失败必须被统一。但有时同时支持多种方法并让客户代码去选择也是可能的。例如，通过一个对话框来让用户选择是否重新从磁盘上读取文件。除了在本章中，这六种方法的其他例子还可以在本书的其他章节中找到。

3.1.1.1 突然中断

一个处理失败操作的极端做法就是让这个方法立即结束并返回（通常是抛出异常），无论当前对象或者活动处于什么状态。当你确信局部的失败会导致整个操作的失败，并且在此操作中的对象不会再被重用时（例如，它们完全被限制在一个会话之中，参见 § 2.3.1），这种方法就是可行的。比如，当一个文件转换组件无法打开被转换文件时，就可使用这种方法。

突然中断也是处理未捕获（及未声明）的 RuntimeException 的默认策略，如 NullPointerException，因为这通常表明程序中含有错误。当一个通常可恢复的错误无法被处理时，你可以将其作为一个 RuntimeException 或者 Error 来抛出，以便执行更极端处理。

因为突然中断并不属于正常的程序终结（通过 System.exit），所以从这种失败中恢复的方法是很少的。当对象被各种操作所共享，无法在失败后重建，并且不可能（或者不切实际）撤销这个失败的操作时，惟一的方法就是在遭遇失败的这个对象中设置一个已被破坏的标记并突然中断。这个标记应该使以后的所有操作都失败，直到那个对象以某种方式被修复，例如，通过错误处理对象来修复。

java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

3.1.1.2 继续

如果一个失败的调用并没有影响调用者的状态，且对当前操作的所有功能需求也没有什么影响的话，那么忽略这个异常并继续执行的做法就是可接受的。虽然这种做法初看起来有些不负责任，但是其在事件框架和单向消息协议（参见 § 4.1）中还是可以使用的。例如，调用一个监听器对象的改变通知的方法失败了，其最坏的结果也仅仅是使一个动画中的几帧被跳过而已，并不会对整体造成长期的影响。

继续的策略在一些错误处理程序中可以看到（在大多数的 finally 语句中），它们在处理错误时往往会忽略处理过程中伴随出现的异常，比如在关闭文件时忽略关闭文件所可能造成的异常。这种处理策略也可以被用于那些永远不能停止的线程中，因为当遇到异常时，它们要尽最大的可能去保证自身继续运行。

3.1.1.3 回滚

在乐观法中，最让人满意设计的就是失败恢复（clean-fail）所提供的担保：要么操作完全成功，如果失败，则把所有对象恢复到操作执行前的状态。在 § 2.4.4.2 中所介绍的乐观更新法就是这种方法的一种演示，在那个方法中，更改得以成功的前提是没有其他线程试图对同一个对象更新。

在回滚中，为了维护对象的状态，通常有以下两种互补的方法。

临时性操作：在试图更新对象状态之前，先构建一套临时的对象来执行操作，如果操作成功，则用临时的对象状态替代原来的对象状态。方法在执行过程中，仅仅更新临时对象的状态，直到可以确保操作成功时，才将临时对象的状态更新到原来的对象中。采用这种方法，可以使我们在失败时无需撤销任何操作。

设置检查点：在试图更新对象的状态之前，使用历史变量按时间顺序记录下对象的状态，可以采用 Memento 模式（参见《Design Patterns》）。方法可以直接更新当前对象的状态。一旦操作失败，对象就返回到以前检查点所记录的状态。

当操作并不是完全同步时，通常必须采用临时性操作这种方法。临时性操作能够避免其他线程看到不一致的、部分更新的对象状态。并且，在读操作多于写操作的情况下，这种方法的效率也是很高的。在其他情况中，由于设置检查点的方法更容易操作，所以应该优先选用。但是，无论采用哪一种方法，通过完全复制整个对象来记录状态通常都是没有必要的，一般来说，在对象中添加一些额外的成员变量或者在方法中添加一些临时变量就足够了。

除了上述更新对象状态的操作需要回滚方法之外，我们还需要别的依据具体情况而定的回滚方法来支持其他种类的操作，比如那些通过向别的对象发送消息的操作。在这些操作中，每个发出的消息都有对应的执行相反操作的反消息（antimessage）。比如，一个贷方（credit）操作应该由借方（debit）操作来恢复。我们可以通过维护与一系列操作相关的撤销消息列表来扩展这个概念，用以支持回滚到任何一个记录点上。

有些操作即不能通过临时性操作也无法通过反消息来恢复，因此，它们无法采用回滚方法。这其中包括了那些由于执行了 IO 操作或者启动其他物理设备而产生了外部可视效果，

java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

且无法毫无影响的恢复的操作。在 IO 操作中，我们通常也可以接受概念上等效的回滚操作。比如，一个方法执行了记录日志的操作，如果这个日志系统支持“请忽略日志记录 XYZ”这种功能，那么在操作失败时，我们就可以使用这个功能来回滚。

然而，正如在§3.1.2.2 中所讨论的那样，大多数 IO 对象（比如 InputStream）都是无法回滚的。大多数 IO 对象都没有可以使它们的内部缓冲区或者成员变量返回到某种状态的控制方法。通常，你的最佳选择就是关闭这个 IO 对象，新建一个并将其绑定到相同的文件、设备，或者网络链接上对象。

3.1.1.4 前滚

当程序无法或不能执行回滚操作，而正常继续运行也已不可能时，可以尽量保守地继续向前前滚，从而重新到达能够正常运行的状态，这种状态可能和出错前的状态有所不同。相对于其他的对象、方法和线程而言，前滚（有时也称“恢复”）这一方法通常特别适合；因为大多数情况下，它们很难区分前滚和回滚。

某些类似的操作可以放到 finally 语句中来执行一些最终的清除任务（比如，关闭文件和取消其他操作），这些任务对达到继续正常执行的安全点是必要的。大多数的前滚方法和回滚方法都十分类似，但由于前滚不需要保存的或临时备份对象的状态，因此该方法通常更易使用。

一些方法从概念上可以分为下列两个部分：一部分很容易通过回滚来恢复的操作（比如可以通过立刻返回或重新抛出异常来回滚），另一部分操作从恢复临界点（point of no return）开始，一旦执行便无法恢复，必须继续运行，无论失败与否，都要运行直至到达安全点。例如，在某个协议的执行过程中，一个方法达到了一个临界点，在这一点上，它必须发出或收到某个应答（参见§3.4.1.4）。

3.1.1.5 重试

当确信重新尝试执行某个操作能够成功时，就可以将局部的错误控制在方法内部，而不向客户抛出异常。重试只有在局部的回滚可行时才可使用，只有这样，对象和操作的状态才能在每次重试前保持正确。

当操作失败的原因是由于其他对象临时处于错误或非期望状态而造成时，则基于重试的策略通常是可行的；例如，当处理和 IO 或者远程机器相关的操作时。正如在§2.4.4.2 中所示，乐观状态更新法通常是依赖于重试来实现的，因为干扰模式一般不会一直持续。在轮询设计中，重试这个方法也是经常被采用，例如§4.1.5 中讨论的那些例子。级联算法中包含了数种重试方法，它首先尝试最优的几种操作，如果没有成功，就执行其他操作，直到操作成功。

如果不小心，不停的重试可能会大量地消耗 CPU 时间（参见§3.2.6）。你可以试探性地在重试中间插入延时，这样不但可以减少重复争用而导致的操作失败，对 CPU 时间浪费的现象也可以有所缓解。一个常用的策略就是指数让步（exponential backoff）法（见以太网协议的例子），每次重试的间隔时间都指数级的长于上一次间隔的时间。

例如，你可以用下面的方法来连接一台在过载时会拒绝连接的服务器。重试的循环在每次失败后都比上一次多延时一会。但是，循环会在线程被中断时退出，因为其没有理由在当

java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

前线程被取消后还继续执行（如 § 3.1.2.2 所提示的，在某些版本的 JDK 中，必须把以下的程序改为先捕获 InterruptedIOException，然后再重新抛出 InterruptedException 才可以）。

```
class ClientUsingSocket {                          // Code sketch
  // ...
  Socket retryUntilConnected() throws InterruptedException {
    // first delay is randomly chosen between 5 and 10secs
    long delayTime = 5000 + (long)(Math.random() * 5000);
    for (;;) {
      try {
        return new Socket(server, portnumber);
      }
      catch (IOException ex) {
        Thread.sleep(delayTime);
        delayTime = delayTime * 3 / 2 + 1; // increase 50%
      }
    }
  }
}
```

3.1.1.6 异常处理器

有时，你可能会希望把错误处理统一交给某些集中的异常处理器，因为在某个线程或者系统的某部分中出现的异常可能需要其他线程的操作来解决；或者当异常出现时，不通过这种方法，系统的其他部分就无法得知这个情况，此时，调用、回调或者通知错误处理对象的方法就非常有用了。当客户程序不知如何去处理异常时，使用这些方法可以使得代码具有更好的扩展性和灵活性。但是，在使用回调及事件通知等相关技巧来替代异常去处理失败时，还是需要要特别仔细地考虑的。一旦脱离了基于堆栈的、流程控制的异常机制，在系统的不同部分中预测和处理异常就变得更加困难了。

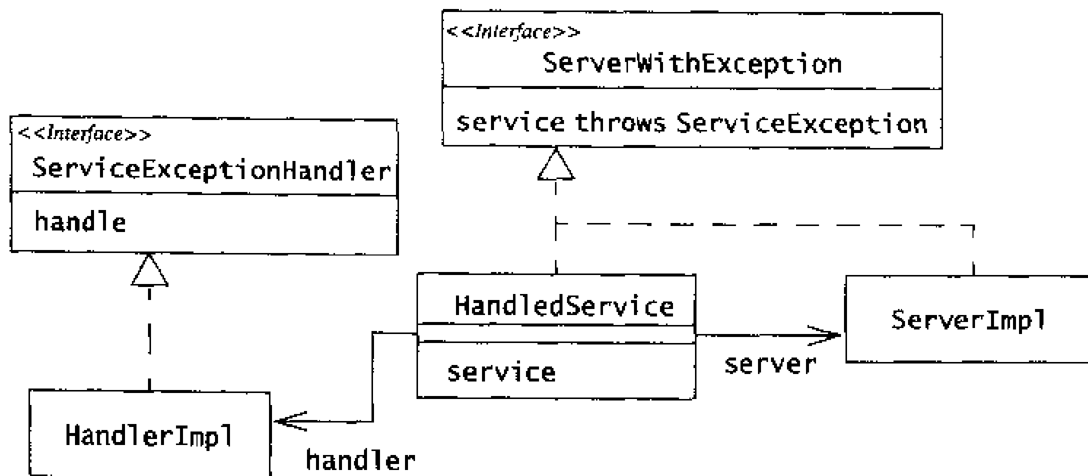
我们可以通过创建一个 before/after 类（参见 § 1.4）来构建一个异常处理器，这个类在之后操作中用于处理异常。例如，假设你用一个接口来定义某个抛出 ServiceException 异常的服务，用另一个接口来定义处理这个异常的处理器。ServiceExceptionHandler 的实现在这里作为 Strategy 对象，详见《Design Patterns》。随后你可以创建一个调用代理给客户程序使用，这样客户程序就不用再处理异常了。例如：

```
interface ServerWithException {
  void service() throws ServiceException;
}

interface ServiceExceptionHandler {
  void handle(ServiceException e);
}

class HandledService implements ServerWithException {
  final ServerWithException server = new ServerImpl();
  final ServiceExceptionHandler handler = new HandlerImpl();

  public void service() { // no throw clause
    try {
      server.service();
```


java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

```
        }
        catch (ServiceException e) {
          handler.handle(e);
        }
      }
    }
```

需要注意的一点是，虽然像这样让 HandledService 实现 ServerWithException 接口是合法的，但是所有依赖于异常处理器的代码都需要直接使用 HandledService 这个类，而不是更抽象的 ServerWithException 类型。



在异常捕捉代码段中可以执行的操作都可以在处理器中执行，包括停止一个或多个线程的操作，以及启动清除处理线程等。异常处理器也可以把错误通过某种交互协议传送给运行在其他线程中的错误处理工具，把异常以 RuntimeException 或 Error 形式重新抛出，把异常封装在 InvocationTargetException 中以表明这是一个级联式错误（参见 § 4.3.3.1）等等。

你也可以这样构建服务，让客户程序在调用 service 方法时，把异常处理器作为回调的参数。当服务本身不知道错误发生后应抛出什么异常时，这种基于回调的处理方法也是可以采用的。这种方法可以如下构建：

```
interface ServerUsingCallback {
  void anotherservice(ServiceFailureHandler handler);
}
```

如代码所示，调用程序必须提供一个在异常出现时调用的回调对象（可能就是调用对象本身）。更多的细节、替换方案和变化可参见 § 4.3.1。

把一种消息传送的协议转换为其他类型的协议时，我们也可以采用异常处理器的方法（参见 § 4.1.1）。例如，在使用基于事件的框架时，一个服务可能会生成一个由 Exception-EventListener 处理的 ExceptionEvent。下面所示的 ServiceIssuingExceptionEvent 的实现方式就是一个例子，它使用了 § 2.4.4 中介绍的 CopyOnWriteArrayList 来管理处理器的列表。除了例子中的方法，也可以异步地触发这些事件（参见 § 4.1）。

java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

```
class ExceptionEvent extends java.util.EventObject {
  public final Throwable theException;

  public ExceptionEvent(Object src, Throwable ex) {
    super(src);
    theException = ex;
  }
}


class ExceptionEventListener {                       // Incomplete
  public void exceptionOccured(ExceptionEvent ee) {
    // ... respond to exception...
  }
}

class ServiceIssuingExceptionEvent {                 // Incomplete
  // ...
  private final CopyOnWriteArrayList handlers =
                                  new CopyOnWriteArrayList();

  public void addHandler(ExceptionEventListener h) {
    handlers.add(h);
  }

  public void service() {
    // ...
    if ( /* failed */ ) {
      Throwable ex = new ServiceException();
      ExceptionEvent ee = new ExceptionEvent(this, ex);

      for (Iterator it = handlers.iterator(); it.hasNext();) {
        ExceptionEventListener l =
                        (ExceptionEventListener)(it.next());
        l.exceptionOccured(ee);
      }
    }
  }

}
```

java.beans 包中使用了一个相反的转换以把事件转换为异常，详细的讨论可以参见 § 3.6.4。

3.1.2 取消

当某个线程中的操作失败了或者改变执行流程时，无论其他相关线程在做什么，可能必须或希望取消这些线程的执行。取消会给运行中的线程带来根本无法预料的失败状态。由取消（Cancellation[1]）的异步特性而导致的设计技巧会使我们联想起在设计分布式系统时，要随时准备处理那些任何时候都有可能因为连接失败或者死机而造成的错误。除此之外，并发程序还需保证其他线程中参与操作的对象处于一致的状态。

在多线程程序中，取消的出现是自然的。例如以下情形：

1 这种两个 l 的 cancellation 拼写方法在并发编程中最常使用。

java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

- 几乎所有图形界面上都有取消按钮的操作。
- 可以被正常停止的媒体演示（比如说动画）。
- 不再需要某些线程产生的结果时。比如，当使用多个线程在数据库中搜索时，一旦有线程返回了结果，其他的线程就可以被取消了。
- 一批操作由于其中的一个或者多个遭遇到未知的失败或者异常而无法继续时。

3.1.2.1 中断

实现取消最常见的方法都依赖于每个线程的中断状态[2]，中断状态可以通过调用 Thread.interrupt 来设置，调用 Thread.isInterrupted 来查询，调用 Thread.interrupted 来查询并清除，以及有时通过抛出 InterruptedException 来响应。

可以使用线程中断来作为取消操作的请求。虽然并没有什么可以阻止你使用中断来做其他的事情，但用来取消操作是其通常的目的。基于中断的取消操作依赖于取消者和被取消者之间的某种协议来确保共享于多个线程中的对象在被取消线程结束时还处于正确的状态。在 java.*包中的大多数类（基本上全部）都遵循这个协议。

在大多数的情况下，取消某线程的操作会导致该线程终止。但这并不保证在调用 interrupt 方法后线程就会立即停止。这样可以让被中断线程在终止之前有机会去做清除操作，不过这也会使得代码必须要查询线程的中断状态并适时地采取操作。

这种推迟甚至忽略取消请求的能力使得编写反应迅速同时又很健壮的程序成为可能。在执行任何很难或者不可能恢复操作之前的安全点，作为前提条件之一，应检测线程的中断状态。检查后的处理包含了 § 3.1.1 中所讨论的大部分方案。

- 继续（忽略或者清除中断状态）可以用于那些被设计为一直运行的线程；例如那些执行数据库管理的服务，它们是实现程序基本功能的基础。当收到中断请求后，线程正在执行的操作可以被终止，这样就可使其继续执行其他的任务。然而，即使在这种情况下，重新启动一个新的处于良好初始状态的线程来替换当前线程也可能是更好的选择。
- 突然终止（例如，抛出 Error）通常用于那些提供独立服务的线程中，这些服务除了在 run 方法的 finally 语句中执行清除操以外不再需要其他操作。然而，当线程执行的服务依赖于其他线程时（参见 § 4.3），它必须对其他线程发出警告或者设置相应的状态（异常本身并不能自动传播到其他线程中）。
- 当线程所使用的对象被其他线程所共享时，必须使用回滚或者前滚的方法。

通过改变对线程中断状态检测（调用 Thread.currentThread().isInterrupted()方法）的频率，部分控制代码对中断的响应速度。为了程序的效率，也不必过于频繁地检测。举例来说，如果执行一个取消操作共需 10,000 条指令，且每执行 10,000 条正常指令就做一次是否被取消的检测，那么，从发出取消的请求到线程终止，平均共需 15,000 条指令。对于大多数的程序来说，只要对继续执行不会造成什么影响，这种频率就已经足够了。由此可知，你可以仅把中断检测代码放在程序中最合适且最重要的位置。但对于对性能要求特别严格的程序而

2 中断在 JDK 1.0 中并没有被支持。由不同版本的策略和机制都不同，导致了各种不规范的取消方法。

java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

言，我们就需要通过构建一套分析模型或者收集试验数据来更准确地确定这个检测频率，以达到最快地反映速度和最佳性能之间地平衡（请参阅 § 4.4.1.7）。

在 Object.wait、Thread.join、Thread.sleep，以及它们的派生方法中，中断检测是自动执行的。这些方法在遇到中断时，会抛出 InterruptedException 异常以中断当前的操作，使得线程可以被唤醒并执行取消操作。

按照规范，当 InterruptedException 异常被抛出的同时，中断的状态也应该被重置。这种特性对清除操作常常是有用的，但也是错误和混淆的源头。如果你需要在处理完 Interruption-Exception 异常后保持并传递该中断的状态，则需重新抛出异常或者调用 Thread.current-Thread().interrupt()来重新设置中断的状态。如果在你所创建的线程中的代码需要调用其他并不能正常维护中断状态的代码，则可以通过这样来避免程序出现问题：使用一个成员变量来记录取消状态，无论何时，在调用 interrupt 方法时都去设置它，且无论何时从那些无法维护取消状态的调用中返回前都检测这个变量。

有两种情况会导致线程处于睡眠状态而无法去检查中断的状态或者捕捉 Interrupted-Exception 异常：被同步锁或者 IO 操作所阻塞。当线程在等待获得某个同步方法或者代码段中的锁时，是无法响应中断请求的。不过，正如 § 2.5 所讨论的那样，当你需要彻底解决在取消过程中由于等待获得锁而造成的等待问题时，你可以使用锁工具类。在使用这些类时，代码只会在访问锁对象自身时才会造成阻塞，而不会在锁对象用来保护其他代码的同步访问时造成阻塞。这种阻塞是非常短暂的（虽然其具体的阻塞时间无法被保证）。

3.1.2.2 IO 和资源失效

某些 IO 类（比如著名的 java.net.Socket 及其相关的类）对于那些会被阻塞的读操作，提供了设置其超时时间的方法，这样你就可以在这些操作超时后检测中断的状态。

在 java.io 包中，还有一种方法也被采用——一种特殊的资源失效法（resource revocation）。如果某线程对一个 IO 对象（比如说 InputStream）s 执行了 s.close()操作，那么以后任何其他线程对 s 的操作（比如 s.read()）都会导致 IOException 异常。资源失效会影响所有使用被关闭的 IO 对象的线程，并且会导致这些被关闭的对象不再可用。如果必要，可以通过创建新的对象来代替这些失效的对象。

在其他场合也非常适合使用这种资源失效法（比如用于安全方面）。它可以使得程序免于碰到这种状况：当多个线程共享一个对象时，只要取消其中的一个线程，就会导致这个对象不可用。绝大多数在 java.io 包中的类没有，也无法在 IO 异常时执行错误恢复操作。例如，如果在 StreamTokenizer 或者 ObjectInputStream 的操作中间出现了一个底层的 IO 异常，没有什么可行方法能够使操作继续成功完成。因此，JVM 所采用的处理策略就是不自动中断 IO 操作。

这也给执行处理取消操作的代码带来了额外的任务。如果一个线程可能正在执行 IO 操作，则在这个操作过程中，任何试图取消此线程的操作都必须知道该线程正在操作的 IO 对象是什么，并愿意关闭这个对象。如果这点可以接受，那么你就可以通过关闭 IO 对象且中断这个线程来完成取消操作。例如：

java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

```
class CancellableReader {                          // Incomplete
  private Thread readerThread; // only one at a time supported
  private FileInputStream dataFile;

  public synchronized void startReaderThread()
   throws IllegalStateException, FileNotFoundException {
    if (readerThread != null) throw new IllegalStateException();
    dataFile = new FileInputStream("data");
    readerThread = new Thread(new Runnable() {
      public void run() { doRead(); }
    });
    readerThread.start();
  }

  protected synchronized void closeFile() { // utility method
    if (dataFile != null) {
      try { dataFile.close(); }
      catch (IOException ignore) {}
      dataFile = null;
    }
  }

  protected void doRead() {
    try {
      while (!Thread.interrupted()) {
        try {
          int c = dataFile.read();
          if (c == -1) break;
          else process(c);
        }
        catch (IOException ex) {
          break; // perhaps first do other cleanup
        }
      }
    }
    finally {
      closeFile();
      synchronized(this) { readerThread = null; }
    }
  }

  public synchronized void cancelReaderThread() {
    if (readerThread != null) readerThread.interrupt();
    closeFile();
  }
}
```

大多数其他取消 IO 操作的例子[3]都源于这种需要：中断那些等待 IO 输入的线程，而那

3 某些 JDK 版本也支持 InterruptedIOException，但只是部分支持，并且只在一部分平台上可用。当写作这本书时，未来版本已经计划取消这种支持，部分原因就是由于它会导致 IO 对象失效。但既然 InterruptedIOException 被定义为 IOException 的子类，这里所讨论的模式在支持 InterruptedIOException 的版本中应该大都可行，但其中也会有不确定的因素：中断时可能会抛出 InterruptedIOException 异常，也可能会抛出 InterruptedException 异常。一个可选的解决方案是，捕捉 InterruptedIOException 异常并把它作为 InterruptedException 异常抛出。

java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

些被等待的输入内容要么是已知不可能达到或者不可能及时达到的。对于大多数基于套接字的流来说，你可以通过设置套接字的超时时间参数来解决。对于其他的流，你可以依赖于InputStream.available，并通过自己写的定时轮询机制来避免IO阻塞超时（参见§4.1.5）。这种机制可以使用一个类似于§3.1.1.5中的定时让步重试（timed back-off retry）协议来解决。例如：

```
class ReaderWithTimeout {                      // Generic code sketch
  // ...
  void attemptRead(InputStream stream, long timeout) throws... {
    long startTime = System.currentTimeMillis();
    try {
      for (;;) {
        if (stream.available() > 0) {
          int c = stream.read();
          if (c != -1) process(c);
          else break; // eof
        }
        else {
          try {
            Thread.sleep(100); // arbitrary fixed back-off time
          }
          catch (InterruptedException ie) {
            /* ... quietly wrap up and return ... */
          }
          long now = System.currentTimeMillis();
          if (now - startTime >= timeout) {
            /* ... fail ...*/
          }
        }
      }
    }
    catch (IOException ex) { /* ... fail ... */ }
  }
}
```

3.1.2.3 异步终止

stop方法原来是包括在Thread类中，但是后来，这个方法就被标记为不鼓励使用的方法了。无论什么情况下，Thread.stop方法都会导致线程突然抛出ThreadDeath异常（同interrupt方法一样，stop并不会中断那些等待获得锁或者IO操作的线程，但和Interrupt方法不同的是，它并不保证能够中断那些调用了wait、sleep或者join方法的线程）。

这是一种武断且危险的操作。由于Thread.stop生成的是异步信号，程序有可能在执行操作时被中止，而为了保证程序的安全和对象状态的一致性，这些操作必须被回滚或者前滚。来看下面这个很简单的例子：

```
class C {                                          // Fragments
  private int v;  // invariant: v >= 0

  synchronized void f() {
    v = -1  ;   // temporarily set to illegal value as flag
    compute();  // possible stop point (*)
```


java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

```
        v = 1;        // set to legal value
      }

      synchronized void g() {
        while (v != 0) {
          --v;
          something();
        }
      }
    }
```

如果 Thread.stop 碰巧在（*）行被调用，那么这个对象就将崩溃：当线程终止时，由于变量 v 被赋了一个非法的值，该对象将处于错误状态。任何其他线程在调用这个对象时，都会导致其执行非期望的或者危险的操作。例如，在这个例子中的 g 方法中的循环将会执行 2*Integer.MAX_VALUE 次直到 v 值重新变为正值。

使用 stop 方法会使回滚或者前滚变得非常困难。乍看起来，这个问题看上去并不是太严重，毕竟所有在调用 compute 过程所抛出的任何未被捕捉的异常都会导致对象处于错误的状态。但是，Thread.stop 所产生的效果是更加隐蔽而危险的，因为你无法在传递取消请求的同时清除 ThreadDeath 异常（由 Thread.stop 抛出）。更加严重的是，除非你每行代码后都添加一个 catch 语句（捕捉 ThreadDeath 异常），你将无法把对象准确地构建为可以恢复的状态，而这就会使程序遭遇无法预测的错误。与之相反的是，你可以用预防代码来清除或者处理其他类型的运行时异常而不会出现这糟糕的结果。

换句话说，将 Thread.stop 方法标记为不鼓励使用并不是为了修复其错误的逻辑，而是用以更正大家对其功能的错误判断。把每个方法都编写成可以允许在每个字节码处出现取消请求一般是无法实现的（这个是开发操作系统的底层代码的开发人员所众所皆知的。编写那些即使很少、很短的但可安全地被异步取消的代码也是很难的）。

要注意的是，任何执行中的代码都被允许去捕捉然后忽略由 stop 方法所抛出的 ThreadDeath 异常。因此，stop 方法同 Interrupt 一样，并不能保证线程一定会被终止，它用起来更加危险一点而已。任何使用了 stop 方法的程序都暗示着：比起不用它，突然中断操作可能会造成的危害会更小。

3.1.2.4　资源控制

在任何要载入并执行外部代码的系统设计中，都可能会用到取消。试图取消那些不遵守标准协议的代码会遇到许多困难。那些代码很有可能完全忽略中断，甚至捕捉并丢弃 ThreadDeath 异常，在这种情况下，调用 Thread.interrupt 和 Thread.stop 将毫无效果。

外部代码做什么或者做多长时间是无法控制的。但是可以并且应该使用标准的安全机制来减少这些意料之外的结果。一种可行的方法是通过创建并使用一个 SecurityManager 及其相关的类来拒绝那些运行时间过长的线程对被检查资源的请求（这个方法的细节超出了本书所讨论的范围，可参见进阶阅读）。这种资源拒绝（resource denial）配合 § 3.1.2.2 中所讨论的资源失效可以防止外部代码和那些应该继续运行下去的线程争夺资源。这些方法所带来的副作用是：常常最终会导致线程因异常而失败。

java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

此外，可以调用某个线程的 setPriority(Thread.MIN_PRIORITY)方法来减少它对 CPU 资源的争用，并可以使用 SecurityManager 来防止线程再次提升它的优先级。

3.1.2.5 多阶段取消

有时，即使最普通的代码也要采用远比预期还要极端的方法来取消。为了应对这种可能性，可以构建一个通用的多阶段取消工具，它在取消任务时，首先采用破坏性最小的方法，如果没有成功，那么再采用破坏性稍大的方法。

多段取消法这个模式可以在很多操作系统的进程中看到。例如，在 Unix 系统的关机操作中就采用了这一模式，它先试图用 kill -1 来终止任务，如果未成功，再执行 kill -2，直至 kill -9。大多数窗口系统中的任务管理器也采用了类似的策略。

以下是一个简单版本的梗概（更详细的使用 Thread.join 的例子可以在 § 4.3.2 中找到）。

```
class Terminator {

  // Try to kill; return true if known to be dead

  static boolean terminate(Thread t, long maxWaitToDie) {

    if (!t.isAlive()) return true;  // already dead

    // phase 1 -- graceful cancellation

    t.interrupt();
    try { t.join(maxWaitToDie); }
    catch(InterruptedException e){} //  ignore

    if (!t.isAlive()) return true;  // success

    // phase 2 -- trap all security checks

    theSecurityMgr.denyAllChecksFor(t); // a made-up method
    try { t.join(maxWaitToDie); }
    catch(InterruptedException ex) {}

    if (!t.isAlive()) return true;

    // phase 3 -- minimize damage

    t.setPriority(Thread.MIN_PRIORITY);
    return false;
  }

}
```

要注意的是，这个例子中的 terminate 方法本身也忽略了中断。这表明该方法所采取的策略就是一旦取消操作开始了，就要一直运行下去。对正在执行中的取消任务执行取消操作，会给为终止而执行清除操作的代码带来很多问题。

由于在不同 JVM 上 Thread.isAlive 行为不尽相同（参见 § 1.1.2），这个 terminate 方法有可能在被中止的线程完全消失之前就返回 true。

java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

3.1.3 进阶阅读

基于模式的异常处理报告可以参考：

Renzel 和 Klaus. "Error Detection", in Frank Buschmann and Dirk Riehle(eds.)《Proceedings of the 1997 European Pattern Language of Programming Conference》, Irsee, Germany, Siemens Technical Report 120/SW1/FB, 1997.

某些保护代码中免于异步取消或者中断的底层技巧（比如，使用硬件中断）在 Java 语言中不存在或者无法使用。但是很多系统软件的开发人员却不惜一切代价地来避免异步取消。可以参考 § 1.2.5 所列出的 Butenhof 的书。关于面向对象并发编程的相关讨论可见：

Fleiner, Claudio, Jerry Feldman 和 David Stoutamire. "Killing Thread Considered Dangerous", 《Proceedings of the POOMA '96 Conference》, 1996.

在比大多数并发编程结构更松散的环境中检测并处理一组线程的终止需要更复杂的协议。通用的终止检测算法的讨论可参见 § 1.2.5 所列出的并发和分布式编程的资料。

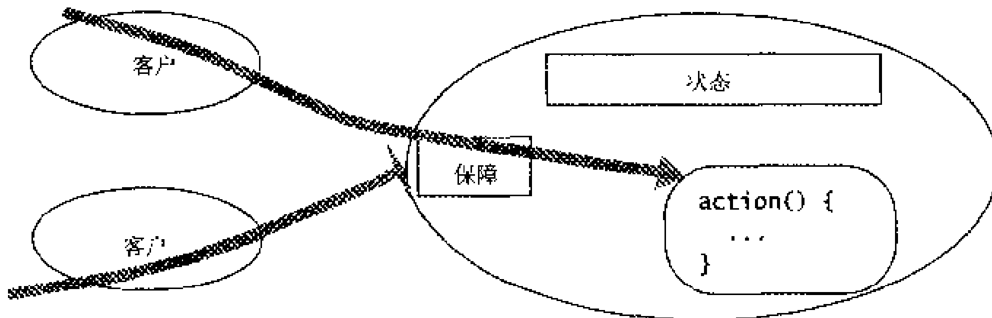
安全管理器可见：

Gong 和 Li. 《Inside Java™ 2 Platform Security》, Addison-Wesley, 1999.

资源控制框架在以下文献中有所介绍：

Czajkowski, Grzegorz 和 Thorsten von Eicken. "JRes: A Resource Accounting Interface for Java", 《Proceedings of 1998 ACM OOPSLA Conference》, ACM, 1998.

3.2 受保护方法



保守的先测再做的方法只有在确保操作会成功时才执行操作，这种保证是部分地依靠检测前提条件来达到的。当前提条件无法达到时，通常有以下三种处理策略。

阻碍（Balking）：当前提条件检测失败时抛出异常。从概念上来说，这种异常和在乐观法中所见到的异常是不同的：在这里，异常意味着拒绝，而不是失败。但通常对客户程序来说，这两种异常所产生效果往往是一样。

保护性挂起（Guarded suspension）：把当前的方法调用（包括其相关线程）挂起，直到前提条件为真。

超时（Time-out）：这是介于阻碍和保护性挂起之间的一种处理策略，它将设置一个等待前提条件为真的时间上限。

java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

在这些方法中，没有一种可以作为通用的最佳处理策略。正如 § 3.4.1 中所示的那样，通常可以提供多个支持不同处理策略方法，最后让客户程序来选择。

阻碍在串行执行的程序和并发程序中都是很常见的。当前提条件不为真且以后也不一定为真时，拒绝执行就是惟一的选择。例如，在线程已启动后再调用 Thread.start 方法会导致该方法抛出 IllegalThreadStateException 异常（参见 § 1.1.2），这是因为线程在启动后就无法返回到未启动状态了。几乎对于所有基于参数的前提条件来说，拒绝也是最好的选择。例如，当一个绘图方法收到的尺寸参数为负值时，它就会抛出 IllegalArgumentException 异常。无论什么时候，当一个方法在获取资源过程中应用"要么现在拥有且执行，要么就退出不再执行"的策略时，阻碍法也是很有用的。当拒绝执行并不被视为异常行为时，使用阻碍法的方法就没必要抛出异常。§ 1.1.1.3 中的 ParticleApplet.stop 方法就是一个例子，这个方法在 applet 停止后忽略后面的停止请求。

3.2.1 保护性挂起

保护性挂起和超时在串行执行的程序中没有类似的对等体，但在并发程序中，这两种方法却起着至关重要的作用。在很多方法中都应用了保护的概念，还有很多说明和构件用来设计那些使用保障的并发软件。在开始讨论如何实现之前，我们先来考虑用来构成基于保护性挂起的设计的总体方法和构件。

下面给出的 BoundedCounter 接口可以作为我们讨论的素材，其含义就是任何实现了 BoundedCounter 的类都要保证计数器的 count 值在 MIN 和 MAX 之间。

```
interface BoundedCounter {
  static final long MIN = 0;  // minimum allowed value
  static final long MAX = 10; // maximum allowed value


  long count();          // INV:  MIN <= count() <= MAX
                         // INIT: count() == MIN

  void inc();            // only allowed when count() < MAX

  void dec();            // only allowed when count() > MIN

}
```

3.2.1.1 保障

从某种意义上来说，受保护方法是 synchronized 类型的方法的一种可定制扩展，它提供了独占的一种扩展形态。对一个普通的同步方法来说，保障的含意就是一个处于就绪执行状态的对象。也就是说，它不参与任何操作。从实现的层面来看，这意味着当前线程拥有这个对象的同步锁。受保护方法通过添加基于状态的条件（如 count() < MAX），进一步细分了这种就绪状态，而这些条件从逻辑上保证了操作可以继续执行。

保障也可以被看作是一种特殊形式的条件。在串行执行的程序中，一个 if 语句就可以检测执行方法所需的条件是否已满足。当条件没有被满足时，没有必要去等待条件为真。因为

java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

既然没有其他并发的操作能够使条件有所改变，那么这个条件自然就永远不会为真了。但是在并发程序中，异步的状态变化会在任何时候发生。

因此，受保护方法带来了那些在简单条件下不会出现的活跃性问题。任何保障隐含着这个断言：最终，某些线程会使需要的状态改变出现，或者，如果这些状态改变不出现，那么最好的选择就是不去执行当前的操作。超时法是对这种断言的一种折中，如果等待状态的时间过长，那么就使用备用的阻碍法终止当前操作的执行。

某些高阶的设计方法使用一种类似 if 的构件 WHEN（有时也称为 AWAIT）来表示有条件的等待，这个构件对于设计受保护方法是很有用的。例如，下列计数器类的伪代码就使用了 WHEN 这个构件：

```
pseudoclass BoundedCounterWithWhen {                    // Pseudocode
  protected long count = MIN;

  public long count() { return count; }

  public void inc() {
    WHEN (count < MAX) {
      ++count;
    }
  }

  public void dec()
    WHEN (count > MIN) {
      --count;
    }
  }
}
```

这里的 WHEN 构件表明了 BounderCounter 要保证 count 的值保持在 MIN 和 MAX 之间。如果接收到 dec 消息，但此时 count 值已经为 MIN 而无法继续减少，则该线程就会被阻塞，直到在其他线程中调用了 inc 方法而使得 count 值大于 MIN 时才会继续执行。

3.2.1.2 基于状态的消息接收

在受保护方法中的操作只有在收到某种消息且对象处于某个状态时才会被触发。由于消息和状态的重要性是相等的，所以在你设计这些方法时，可以交换这两部分的顺序。当类的几个不同方法在同一种状态下被触发时，基于状态的形式可能更加易于使用，例如，当对象作为某种角色使用时。这种形式同时也更能清楚地表达某些流行的高阶 OO 分析及其设计中使用的基于状态的表示法。

Ada 语言中的并发结构可以用来按这种形式来定义方法。如果使用 Ada 的伪代码，那么 BoundedCounter 就可以被表示为：

```
pseudoclass BoundedCounterWithAccept {                  // Pseudocode
  protected long count = MIN;

  WHEN (true) ACCEPT public long count() {
    return count;
  }
```


java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

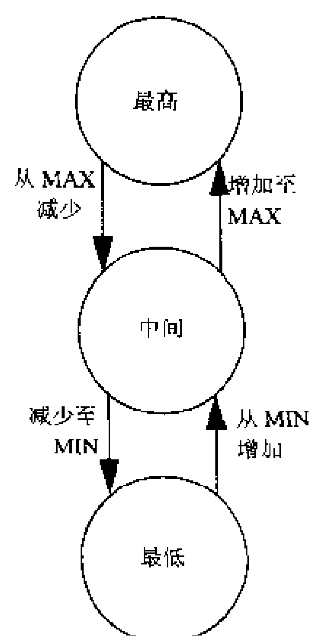
```
    WHEN (count < MAX) ACCEPT public void inc()  {
      ++count;
    }

    WHEN (count > MIN) ACCEPT public void dec()  {
      --count;
    }
  }
```

从极端的角度来看，对于某些设计，如果你假设操作一直处在被请求中，但是只有当对象状态发生了特定的转变后，才会被触发，这些设计可能理解起来更加容易一些。某些循环的方法就是采用了这种形式。例如，你可以设计一个特殊的计数器，每当计数器的值达到某个门限时，就把计数器的值置为零，通过这个机制使得计数器持续可用。这种形式有时也被称为并发约束编程（Concurrent Constrain Programming），此时由于并没有消息调用，操作都是由于对象状态的改变而被触发的。

3.2.1.3 定义逻辑控制状态

多个对象所维护的多个变量一起组成了一个庞大的（或者对于实际情况来说是无穷大的）状态空间，但是只有一小部分逻辑状态是用于保证操作的执行的。例如，对于 inc 和 dec 方法，BounderCounter 只有三个逻辑状态是有效的，而不是 count 的每一个值都有一个状态。

状　　态	条　　件	inc	dec
最高	count == MAX	不可操作	可操作
中间	MIN < count < MAX	可操作	可操作
最低	Count == MIN	可操作	不可操作



在标记这些状态时要注意一些特殊情况。比如，如果 MAX 等于 MIN＋1，那么就没有明显的中间状态了。如果 MIN 等于 MAX，那么就没有办法区分最高和最低状态，此时，inc 和 dec 都无法执行。

如表中所示，逻辑状态通常用谓词来定义——即布尔表达式，用于区别不同的成员变量的范围、值和其他可计算的属性。它们可以是程序中单独的返回布尔值的方法，或者直接就是方法中的布尔条件。如果对于这种方法中的状态分析过于复杂且笨拙的话，则可以使用以下方法来设计并编码状态：状态图、表、决策树、自动机，以及处理状态机的相关方法（见进阶阅读 § 1.3.5）。

除了使用谓词表达式以外，还可以使用变量来明确表示逻辑状态。每种不同的状态都可以用一个整数或者其他离散的数据类型来表示。为了保证这个变量所表示的状态保持准确，每次更新操作都需要重新计算它的值（参见 § 3.3.1）。而且，并不是只能使

java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

用一个变量，如果一个对象的状态可以被分为相互独立的几维，那么就可以使用多个变量来表示对象的状态。几种特殊的例子包括：

- 角色变量（Role Variable）控制着对一系列相关方法（这些方法通常定义在一个单独的接口中）的所有应答。当对象可以在角色间切换时，一个单独的变量就足够用来指示如何采取合适的操作了。例如，一个对象可能在生产者和消费者的角色中切换。当它处于某一种角色中时，可能会忽略或者延时应答那些和其他角色相关的消息。
- 除了用值来表示状态外，你还可以使用一个状态对象（state object）的引用来表示状态。对于每一种状态，你可以写一个类来描述当对象处于这种状态时的行为。然后，主类中包含有一个引用变量，比如 stateObject，它总是指向一个合适的代理对象。这是对《Design Patterns》一书中状态对象（State as Object）模式的一个应用；§ 3.7.2 中描述了该模式的一个变体。

3.2.2 监控机制

实现受保护方法的途径很多，并不止于设计它的方法。但是，基本上所有的实现方法都可以被视为以下几种使用了 Object.wait、Object.notify 和 Object.notifyAll 策略的特殊形式：

- 对于每一个需要等待的条件，写一段受保护的 wait 循环：如果当前受保护条件不为真，则使当前线程阻塞。
- 确保任何会改变被等待条件的方法都会通知那些正在等待这些条件的线程，使得它们在被唤醒以重新检测当前的受保护条件。

作为讨论这些方法的预备知识，下面列出了等待和通知方法特性的概要。

正如每个 Object 都有一个锁（参见 § 2.2.1），每个 Object 也有一套等待集合（wait set），它由 wait、notify、notifyAll 和 Thread.interrupt 方法来操作。同时拥有锁和等待集合的实体通常被称作监视器（Monitor）（虽然每种语言所定义的细节都会有所不同）。任何 Object 都可以用作监视器。

每个对象的等待集合都是维护在 JVM 内部的。等待集合一直保存着那些因调用对象 wait 方法而被阻塞的线程，直到接受到相应的通知或者该等待集合被释放。

由于等待集合和锁之间的交互机制，只有获得目标对象的同步锁时，才可以调用它的 wait、notify 和 notifyAll 的方法。这种要求通常无法靠编译来检查。如果条件不能被满足，那么在运行时刻调用以上方法就会导致其抛出 IllegalMonitorStateException 异常。

每种方法在被调用后所执行的操作如下所示：

Wait。wait 方法被调用后，会执行如下操作：

- 如果当前线程已被中断，那么该方法会立即退出，然后抛出一个 InterruptedException 异常。否则，当前线程被阻塞。
- JVM 把该线程放入目标对象内部且无法访问的等待集合中。
- 目标对象的同步锁被释放，但是这个线程所拥有的其他锁依然被这个线程保留着。即使由于嵌套的同步调用而使线程重新获得目标对象的同步锁，这个锁也会被完全释放。当线程重新恢复执行时，它会重新获得目标对象的同步锁。

java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

Notify。notify 方法被调用后，会执行如下操作：

- 如果存在的话，JVM 从目标对象内部的等待集合中任意地移除一个线程 T。如果等待集合中的线程数大于 1，那么哪个线程被选中完全是随机的。参见 § 3.4.1.5。
- T 必须重新获得目标对象的同步锁，这必然导致它将被阻塞直到调用 Thread.notify 的线程释该同步锁。如果其他线程在 T 获得此锁之前就获得它，那么 T 就要被一直阻塞下去。
- T 从执行 wait 的那点恢复执行。

NotifyAll。notifyAll 方法被调用后的操作和 notify 类似，不同的只是等待集合中所有的线程（同时）都要执行那些操作。然而，由于它们必须要先获得目标对象的同步锁，所以只有一个线程可以继续操作。

Interrupt。如果对一个因调用了 wait 而被挂起的对象调用 Thread.interrupt 方法，那么这个方法的执行机制就和 notify 类似，只是在重新获得对象锁后，该方法就会抛出 InterruptedException 异常，并且该线程的中断状态将被置为 false。如果 interrupt 和 notify 同时发生，那么哪个操作会被先执行完全是随机的，所以两种结果都是有可能的（JLS 的后续版本可能会对这种结果给出确定性的保证）。

定时的 Wait。那些可以设置等待时间的 wait 方法，wait(long msecs)和 wait(long msecs, int nanosecs)，可以通过调用参数来设定线程在等待集合中的最长等待时间。它们所执行的操作是和没定时的 wait 方法是一致的，当等待的时间超过了预定的时间后，该等待会被自动释放。没有办法可以区分 wait 方法是因通知还是超时而返回的。与直觉相反的是，wait(0)及 wait(0,0)和普通不定时的 wait 方法有着相同的效果。

由于线程间的竞争，线程的调度策略以及时间的粒度问题（对于时间的粒度，没有任何担保。对于那些小于 1ms 的参数，大多数 JVM 的实现都会有 1～20 毫秒的可察觉的反应时间），一个定时的等待可能会在设定的超时时间前后的任意时刻被恢复。

Thread.sleep(long msecs)方法使用了一个定时的等待，但是其并不和当前对象的同步锁相关，它的操作类似于下面的代码：

```
if (msecs != 0)  {
  Object s = new Object();
  synchronized(s) { s.wait(msecs); }
}
```

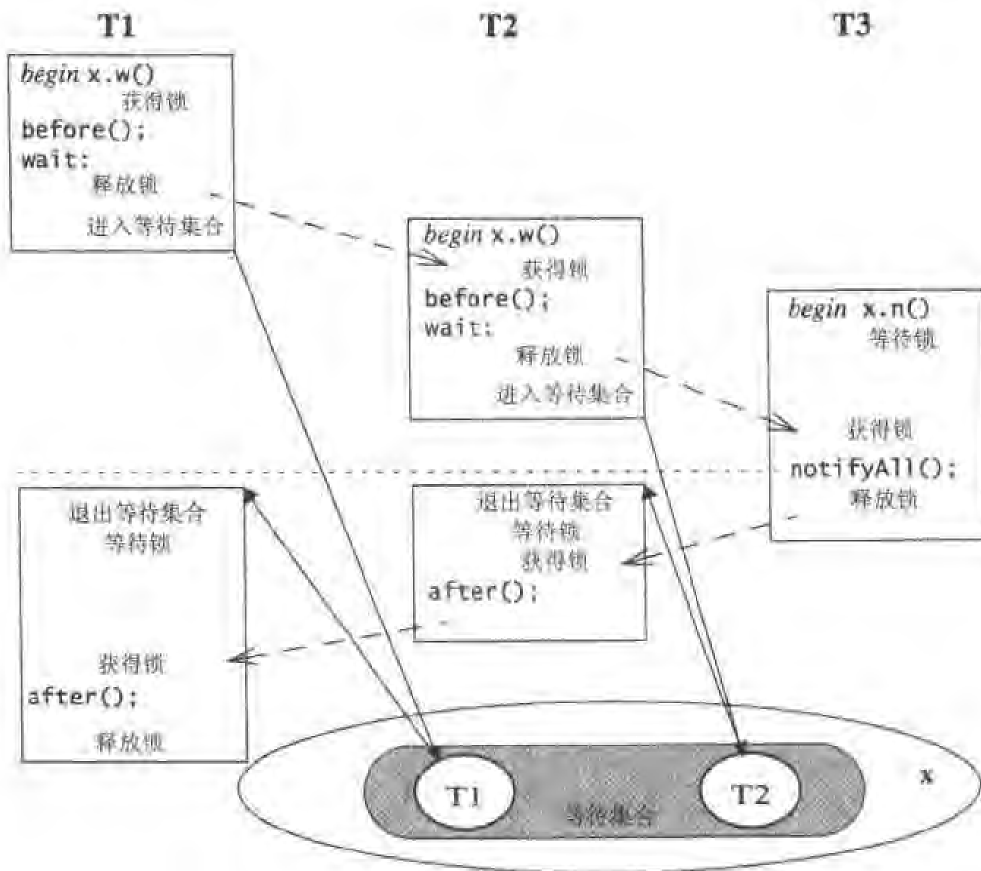
当然，没有必要非得如此实现 sleep 方法。要注意的一点是，无论其含义如何，sleep(0)几乎不会造成暂停。

为了阐明这些方法的内部机制，可以考虑以下这个使用 wait 和 notifyAll 的类。

```
class X {
  synchronized void w() throws InterruptedException {
    before(); wait(); after();
  }
  synchronized void n() { notifyAll(); }
  void before() {}
  void after() {}
}
```


java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

以下是当 3 个线程调用一个 x 的方法时可能的结果。注意，虽然 T1 在 T2 之前开始等待，但是 T2 却在 T1 之前恢复。当然结果也可能不是这样，这完全是随机的。



3.2.3 受保护的等待

实现受保护方法的标准模式就是一个简单的调用 wait 的 while 循环。例如，BoundedCounter 的 inc 方法可以按如下方法来实现：

```
synchronized void inc() throws InterruptedException {
  while (count >= MAX) wait();
  ++count;
  // ...
}
```

有时为了确保保障能被正确实现，把每个保障封装在其自己的方法中可能会有所帮助。例如下面这个通用的例子：

```
class GuardedClass {                        // Generic code sketch
  protected boolean cond = false;

  // PRE: lock held
  protected void awaitCond() throws InterruptedException {
    while (!cond) wait();
  }
```


java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

```
    public synchronized void guardedAction() {
      try {
        awaitCond();
      }
      catch (InterruptedException ie) {
        // fail
      }

      // actions
    }
  }
```

条件检测必须放在 while 循环中。当一个处于等待中的操作被恢复时，它并不知道其所等待的条件是否真的被满足了，它仅仅知道它被唤醒了。因此，为了确保程序状态正确，必须再次检测所等待的条件。

作为一个编程的习惯，即使类中只包含一个 wait 且其只等待一个条件时，也应该使用这种形式。那种假设对象在从某个 wait 中恢复后会处于某种状态中的代码是永远不可接受的。其中一个原因就是：如果其他不相关的对象在目标对象上误调用了 notify 或者 notifyAll 方法（因为所有对象的这些方法都是公共的），那么这种代码就会出错。此外，这种形式的等待也可以避免假醒（系统在未调用任何通知方法前就释放了当前的等待[4]）所造成的破坏。不过，最重要的一点是，如果没有这个重新检测的步骤，当别人添加了用于其他目的的等待和通知的方法后（可能在该类的子类中），这个程序就可能会出现非常奇怪错误。

设计拥有受保护的等待的对象比设计简单的完全同步对象（参见 § 2.2.2）难度更大。拥有受保护的等待的方法并不完全是原子的。那些等待中的方法在挂起时不会保留对象同步锁，这就使得其他线程可以开始执行该对象的任何 synchronized 方法（通常的情况是，其他非同步的方法仍然可以在任何时刻被执行）。

3.2.3.1　中断的等待

如果等待中的线程已被中断了，那么 wait 操作就会失败（或者根本就不执行）。这样，阻塞中的线程就可以响应线程取消的请求。从这点上看，受保护方法同“先试再看”的方法是类似的（首先尝试那些可能会失败的前提条件）如果尝试失败，那么 § 3.1 中所介绍的错误处理策略和实现方法都是可以应用的。

到目前为止，在受保护方法中最常用的处理策略就是通过重新抛出 InterruptedException 异常来把错误通报给客户程序，然后由客户程序来处理它。如果这个受保护的等待是在方法的开始部分，那么就不需要执行任何清除方法了。

重新抛出 InterruptedException 异常（或者，通常是不捕捉它）使得方法签名中要包括 throws InterruptedException，而这可以作为一个方法使用受保护的等待或者其衍生形式的简单声明。这对该类的潜在用户可是一个重要的信息（参见 § 3.3.4）。

4　当撰写本书时，JLS 并未特别指出假醒是否可以出现。然而，大多数 JVM 的实现都使用了操作系统的程序（如 POSIX 线程库），而这些程序是允许假醒的，且假醒是可能发生的。

java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

3.2.4 通知

基于等待的这种结构确保了实现保障时的安全性。为了保证线程的活跃性，类中还必须包含那些当等待的条件为真时，唤醒等待这些条件的线程的代码。任何时候，当保障中的任何一个变量发生了变化，而这个变化可能导致受保护条件为真时，等待中的线程都应被唤醒，并重新检测受保护条件。

让被阻塞的线程最终能够重新检测条件的最简单方法就是：在那些会导致相应的状态变化的方法中插入 notifyAll。实现这种方法的最简单做法就是定义相应的工具方法，这些工具方法封装了赋值及当变量的值有变化时发出通知的操作。这可能会使那些有很多赋值操作的类产生无用的信号并会导致比较差的性能（由于上下文切换所导致）。然而，作为一个设计的惯例，可以先在赋值方法中使用一个通知方法，然后再按照本章后面所提供的方法来优化，这通常是一个好主意。以下的 BoundedCounter 就是一个例子：

```
class SimpleBoundedCounter {
  protected long count = MIN;

  public synchronized long count() { return count; }

  public synchronized void inc() throws InterruptedException {
    awaitUnderMax();
    setCount(count + 1);
  }

  public synchronized void dec() throws InterruptedException {
    awaitOverMin();
    setCount(count - 1);
  }

  protected void setCount(long newValue) { // PRE: lock held
    count = newValue;
    notifyAll(); // wake up any thread depending on new value
  }

  protected void awaitUnderMax() throws InterruptedException {
    while (count == MAX) wait();
  }

  protected void awaitOverMin() throws InterruptedException {
    while (count == MIN) wait();
  }
}
```

3.2.4.1 错误状态与丢失信号

在 SimpleBoundedCounter 中，于 inc 方法中对 awaitUnderMax 和 setCount 的调用是在同一个同步锁的范围中。仅仅分别地同步 awaitUnderMax 方法和 setCount 方法，而不去同步整个 inc 方法是不够的。这会导致安全问题。例如：

java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

```
void badInc() throws InterruptedException {   // Do not use
  synchronized(this) { while (count >= MAX) wait(); }
  // (*)
  synchronized(this) { ++count; notifyAll(); }
}
```

这个版本的 inc 方法可能会有错误状态，即由其他线程在（*）点（在等待结束并释放了锁之后和为增加 count 值而重新获得锁之前这段时间）所执行的操作而导致的变化了的条件。这种状态可能会导致在受保护条件不为真的情况下，操作也会被执行，这很有可能会破坏对象的不变约束，从而使对象失效。

此外，如果 setCount 操作不具有原子性，那么就有可能出现线程的活跃性问题。例如：

```
void badSetCount(long newValue) {             // Do not use
  synchronized(this) { notifyAll(); }
  // (**)
  synchronized(this) { count = newValue; }
}
```

在这个例子中，该方法首先获得锁以执行 notifyAll，然后释放它，随后重新获得锁以改变 count 的值。这就可能会导致出现丢失信号的现象：执行到**点的线程可能会在试图唤醒它的信号发出之后，但是在其等待的条件改变之前就开始等待。随后，这个线程要么一直处于等待状态中，要么一直等待，直到下一个通知被发出。

需要注意的是，在 synchronized 方法中，notifyAll 放置的位置并不重要。只有当这个同步锁被释放时，其他被唤醒的线程才可以继续。作为编程的风格，大多数人都把通知放在方法的最后部分。

这里所示的导致错误状态和丢失信号的代码可能看起来比较牵强。但是，在更复杂的等待与通知的设计中，它们可能就是常见错误的源头（例如在 § 3.7.2 中的例子）。

3.2.4.2 单一通知

SimpleBoundedCounter 类使用了 notifyAll 方法，因为其他线程既可能在等待计数器的数值大于 MIN，也可能在等待其值小于 MAX。所以，使用 notify 是不能满足需要的，这是因为它仅仅只能唤醒一个线程（如果存在的话）。JVM 很有可能选择了某个条件尚未满足的线程，而不是那些等待条件已被满足并可以继续执行的线程。这种情况是很可能发生的，比如，如果当有多个线程正在试图增加计数器值而其他几个正在减少时（考虑当 MAX==MIN+1 时）。

但是，在某些其他类中，你可以通过使用 notify 代替 notifyAll 来减少通知操作所带来的上下文切换的开销。当确认最多只有一个线程需要被唤醒时，你就可以使用单一通知来提高程序的性能。当以下这几种条件满足时，就可以应用这种策略：

- 所有等待中的线程所等待的条件都依赖于同一个通知，通常它们所等待的条件也都是相同的。
- 每个通知本来就只能允许最多一个线程继续运行。这时唤醒其他线程时就没有必要。

java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

- 你可以接收 interrupt 和 notify 发生在同一时刻所带来的不确定因素。在这种情况下，一个被唤醒的线程很有可能要被终止。此时，你可能会希望其他线程去替它接收这个通知，但是这并不会自动完成（在使用 notifyAll 时，这个问题就不会存在，因为所有的线程都会被唤醒）。

为了说明 notify 和 notifyAll 之间的关系，下面的 GuardedClassUsingNotify 类通过在那些封装了保障的辅助方法中添加相应的指令，以此来用 notify 去模拟 notifyAll。在其中，我们添加了一个执行状态（executation state）变量来跟踪等待的线程数量，并用其构建了一个循环，以把通知广播给所有的等待线程，这样就实现了对 notifyAll 的模拟（但这种模拟仅仅是近似的，notifyAll 才是最基本的内建操作）。

在这个类中，看上去比较奇怪的 catch 语句是为了确保：如果一个被取消的线程接收到通知后，能够把这个通知重新发给其他处于等待中的线程（如果存在的话）。在这里，这种安全保护并不是必须的，毕竟所有等待中的线程都会被唤醒，但是，在任何使用 notify 的代码中，如果中断并不应该让整个程序都终止，那么就应该使用该技巧。

要注意的一点是，在 catch 语句中对 notify 额外的调用可能会导致计算所得的需要通知的线程数量大于实际等待中的线程数量。因此，这会使得调用 notify 次数多于所需要的次数。这点也证明了在使用 notify 时，把 wait 方法放在保障循环中的必要性。

```
class GuardedClassUsingNotify {
  protected boolean cond = false;
  protected int nWaiting = 0; // count waiting threads

  protected synchronized void awaitCond()
   throws InterruptedException {
    while (!cond) {
      ++nWaiting;     // record fact that a thread is waiting
      try {
        wait();
      }
      catch (InterruptedException ie) {
        notify();     // relay to non-cancelled thread
        throw ie;
      }
      finally {
        --nWaiting;  // no longer waiting
      }
    }
  }

  protected synchronized void signalCond() {
    if (cond) {                    // simulate notifyAll
      for (int i = nWaiting; i > 0; --i) notify();
    }
  }
}
```

在开放的、可扩展的设计中（参见 § 1.3.4），使用 notify 所需的条件通常是相当特殊且脆弱的。使用 notify 来优化提供保障的构件，通常是错误的源头。作为一个常用的设计策略，

java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

可以把 notify 隔离在并发控制工具类（参见 § 3.4）中，而这些工具类是可以被单独优化并仔细审查及测试的。在本章后面的部分将采用这个做法。

在自闭合的设计中，使用 notify 的条件通常比较容易达到，因为你可以完全控制所有参与的线程。例如，以下这个自闭合系统的两人游戏的代码片断在轮流等待时使用了等待。在这种情况下，使用单一的 notify 就足够了，因为被唤醒的线程只可能是另一个等待的线程。从另一个角度来说，由于只有一个线程在等待，所以这种版本和使用 notifyAll 的版本之间的性能差距可能非常的小，这是因为和 notifyAll 相关的开销主要来自于上下文切换，而非调用 notifyAll 本身。

要注意的一点是，在 GamePlayer.releaseTurn 中，giveTurn 是以开放调用（参见 § 2.4.1.3）的形式执行的。在执行通知操作时，尽量释放同步是一个好习惯（参见 § 3.7.2）。

```
class GamePlayer implements Runnable {              // Incomplete
  protected GamePlayer other;
  protected boolean myturn = false;

  protected synchronized void setOther(GamePlayer p) {
    other = p;
  }

  synchronized void giveTurn() { // called by other player
    myturn = true;
    notify();                    // unblock thread
  }

  void releaseTurn() {
    GamePlayer p;
    synchronized(this) {
      myturn = false;
      p = other;
    }
    p.giveTurn(); // open call
  }

  synchronized void awaitTurn() throws InterruptedException {
    while (!myturn) wait();
  }

  void move() { /*... perform one move ... */ }

  public void run() {
    try {
      for (;;) {
        awaitTurn();
        move();
        releaseTurn();
      }
    }
    catch (InterruptedException ie) {} // die
  }

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


java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

```
        GamePlayer one = new GamePlayer();
        GamePlayer two = new GamePlayer();
        one.setOther(two);
        two.setOther(one);
        one.giveTurn();
        new Thread(one).start();
        new Thread(two).start();
    }
}
```

3.2.5 定时的等待

为了不在受保护方法中永无止境地等待一个条件变为真，基于超时的设计指定了 wait 操作处于挂起状态的时间上限。

当然，对于超时的响应是和环境相关的。当用超时法来试探性执行某些操作时，未满足这个谓词的情况对相应程序来说，可能仅仅是一个信息而已。而在其他情况中，超时可能会导致那些试探性执行的操作都被取消，在这种情况下，定义一个 TimeOutException 异常作为 InterruptedException 的子类通常是合适的。

当检测无法预知的活跃性问题（比如说死锁[5]）时，超时法通常比其他方法更为有效。因为与其他方法相比，它们对上下文做的假设较少。任何暂停所导致阻塞的时间一旦超过了可接受的范围，超时法都可以将其检测出来并触发相应的失败响应（参见 § 3.1.1）。由于对各种延时的响应大部分都是相同的，所以它们都可以被超时异常或者其他通知方法所触发，控制等待时间以重新判断等待条件的参数有时是完全任意的，且常常需要通过反复地试验来决定。通常，提供合适的参数并不困难，这些参数能使超时捕捉到真正的活跃性问题，而不是在等待时间刚开始变长时就发出假警告。由于大多情况下，对这种失败的处理都需要用户介入，我们可以通过一些机制来向用户询问补救的措施，以此作为备份的策略。

超时有时候很难用 wait(msec)来表达。在下面的 TimeoutBoundedCounter 类中，wait 方法被放在循环中，以避免可能出现的不相关通知所带来的问题。这个循环中的代码看上去有一点乱，但是其原理和没有使用超时的那个版本基本是一致的。当线程从等待中被唤醒后，在检测超时前，它总是要去重新检测等待的条件是否已经满足。这可以避免以下这种情况所造成的编程错误：当因超时而使等待中的线程唤醒时，其他的竞争线程先于这个超时的线程得到了恢复执行的机会。这些竞争线程很有可能已改变了等待的条件，此时，这个因超时而被唤醒的线程可能就不必或不应该再出失败的信号了。如果条件尚未达到，那么将重新检测超时时间，并为在下次循环中使用而做相应的调整。

当线程因超时而恢复后，如果不论等待的条件是否满足，只要超时了，就意味着失败。那么在这种情况下，你可以修改代码，改变其中等待条件检测和超时检测的顺序。

---

5 对检测死锁的算法的讨论有很多，例如，在§1.2.5 进阶阅读中列出的 Andrews 和 Bernstein、Lewis 的文章。它的实现需要使用特殊的锁。然而，一些运行系统和调试器具有可以检测内建同步机制等导致的死锁的功能。

java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

```
class TimeoutException extends InterruptedException { ... }

class TimeOutBoundedCounter {
  protected long count = 0;

  protected long TIMEOUT = 5000; // for illustration

  // ...
  synchronized void inc() throws InterruptedException {

    if (count >= MAX) {
      long start = System.currentTimeMillis();
      long waitTime = TIMEOUT;

      for (;;) {
        if (waitTime <= 0)
          throw new TimeoutException();
        else {
          try {
            wait(waitTime);
          }
          catch (InterruptedException ie) {
            throw ie;  // coded this way just for emphasis
          }
          if (count < MAX)
            break;
          else {
            long now = System.currentTimeMillis();
            waitTime = TIMEOUT - (now - start);
          }
        }
      }
    }

    ++count;
    notifyAll();
  }

  synchronized void dec() throws InterruptedException {
    // ... similar ...
  }

}
```

3.2.6 忙等待

通常，使用等待与通知的方法所实现的保障总是比以下这种乐观重试类型（optimistic-retry-style）的忙等待“反复循环（spinloop）”要好。

```
protected void busyWaitUntilCond() {
  while (!cond)
    Thread.yield();
}
```

由于忙等待的缺陷，使得其对于大多数的受保护操作的实现都不太合适。通过对比，可

java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

以说明为什么基于挂起的等待与通知方法是更优秀的。

3.2.6.1 效率

忙等待法可能会浪费很多 CPU 时间来不断地做无用的循环检测工作。而基于 wait 的方法仅仅在其他线程发出对象状态改变的通知后，即保障条件可能发生变化时，才重新检测等待条件。即使有时当条件并未满足时也会发出通知，但其造成的无效检测次数也远远少于那种持续而盲目的反复循环造成的次数。

最主要的例外情况就是：已知所等待的条件一定会在非常短且有限的时间内达到。此时，浪费在不停循环检测上的时间可能会少于挂起并恢复线程所需要的时间。在和设备控制相关的操作中常应用这种方法。有限制的反复循环与挂起一起常被用于运行时（run-time）系统中，作为那些通常只被保持很短时间的“自适应”锁的优化的一种手段。

3.2.6.2 调度

在反复循环中的 yield 方法仅仅是个对 JVM 的一个提示（参见§1.1.2），JVM 并不担保这个方法能有效地使其他线程获得执行时间以改变等待条件。因此，这种忙等待法是更加依赖于特定 JVM 的，并可能会与线程调度的其他方面相互影响。例如，如果反复循环的线程的优先级较高，而改变等待条件的线程优先级比较低时，这个反复循环的线程就有可能一直运行而把其他线程都排挤在一边。在基于 wait 的版本中，处于等待中的线程根本就不会运行，因此也就不可能会遭遇到这种线程调度的问题了（当然，其他的线程调度问题还是存在的）。

某些最有可能去使用反复循环法的情形就是当循环中还可以进行其他操作，而不仅仅是 yield 操作时。这些其他的操作有助于减少反复循环法对 CPU 时间的浪费，并可以更好地适应通常的线程调度策略。如果反复循环法是必须的且没有其他方法可以替换时，则可以通过使用§3.1.1.5 中所介绍的定时让步（timed back-off）技巧来减少 CPU 的开销和避免线程调度中的不确定问题。

3.2.6.3 触发

与基于 wait 的结构不同，使用了反复循环法的方法并不需要对应的通知方法来触发对条件的检测。有时，在没有或者无法编写这种信号通知的方法的绝望情况下，反复循环法也会被使用。但是，如果碰巧当条件暂时被满足时，线程却没有被调度到执行状态，这种忙等待法就有可能会错失运行的时机。

不过，要注意的是，相似的现象也会在基于 wait 的结构中出现：在被通知的线程有机会继续执行之前，由于其他线程所执行的操作，notify 或者 notifyAll 发出信号时的条件可能会在信号发出后有所改变。这也是为什么要保护所有类型的等待的原因之一。

此外，没有一种方法可以单独且自动地保证公平性——即每个潜在的可执行线程最终都会执行。即使在基于 wait 的结构中，也可能发生这种情况：某个特殊的，不断因受保护方法而挂起的循环线程，却总被最先恢复执行，因此，其他线程根本没有机会执行（参见§3.4.1.5）。

java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

3.2.6.4 同步操作

按理想的方式同步反复循环是很困难的。比如，如果把 busyWaitUntilCond 方法声明为 synchronized，那么这个方法很有可能就不工作了，因为这会导致没有其他 synchronized 方法可以去改变等待条件。至少，cond 应该被定义为 voliate 并且在其自己的 synchronized 方法中进行存取。但是，如果没有对整个“检测一执行”的过程进行同步的话，你就无法保证在条件检测和执行操作时，对象的状态一直保持一致。

如 § 3.4.2.1 中所示，只有在那些担保一旦为真就永远为真的闭锁断言中，才可安全地在受保护方法中使用非同步的忙等待法。与之相反的是，基于 wait 的版本在等待时可以自动地释放同步锁（只是对主体对象而言），并且在被唤醒时自动获得该锁。只要保障和相应操作是被包围在同一个锁之内，且保障所引用的变量也被这个锁保护着，那么就不会有出现错误状态的危险。这就是为何 wait 语句只可以被用在同步环境之中的原因之一。可是，等待任务拥有其他锁也会导致维护上的困难，包括 § 3.3.4 所讨论的嵌套的监视器的问题。

3.2.6.5 实现

在那些你不得不采用忙等待的情况下，你可以使用类似如下 SpinLock 的类。没有任何理由去为了实现锁（参见 § 2.5.1）而使用这个类，但这个类所展示的结构在其他情况中是可以使用的。

这个例子中的 release 方法是同步的，这是为了确保当锁被释放时，会执行内存同步操作，这种同步是任何使用这个类的程序都需要的（参见 § 2.2.7）。当条件检测失败后，应该如何增加延时的时间是与平台及应用相关的（例如，纯粹的、不使用 yield 的循环阶段，只在多处理器平台上才是有意义的），而且即使有很多的经验，这个时间也是很难有效地调节。

```
class SpinLock {                     // Avoid needing to use this

  private volatile boolean busy = false;

  synchronized void release() { busy = false; }

  void acquire() throws InterruptedException {
    int spinsBeforeYield = 100;      // 100 is arbitrary
    int spinsBeforeSleep = 200;      // 200 is arbitrary
    long sleepTime = 1;              // 1msec is arbitrary
    int spins = 0;
    for (;;) {
      if (!busy) {                   // test-and-test-and-set
        synchronized(this) {
          if (!busy) {
            busy = true;
            return;
          }
        }
      }

      if (spins < spinsBeforeYield) {        // spin phase
        ++spins;
      }
```


java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

```
          else if (spins < spinsBeforeSleep) {  // yield phase
            ++spins;
            Thread.yield();
          }
          else {                                // back-off phase
            Thread.sleep(sleepTime);
            sleepTime =  3 * sleepTime / 2 + 1; // 50% is arbitrary
          }
        }
      }
    }
```

3.3 类的构建与重构

§3.2 所讨论的基本的等待与通知方法可以和其他设计策略进行整合，以提高其重用性与/或性能，同时可以对操作进行更好地控制。这一节研究了在跟踪逻辑状态和执行状态时，包装可覆盖方法中的控制策略时，以及与构建基于限制的设计时，常见的一些通用模式、技巧及问题：

3.3.1 跟踪状态

编写受保护方法最保守的策略就是在每次更改一个实例变量时，都去调用 notifyAll 方法，这个策略的扩展性很强。如果所有实例变量的变化都会导致 notifyAll 被调用，那么在这个类及其子类的任何方法中都可以定义一段等待相应状态的保障语句。但从另一方面来说，当生成的通知不能导致任何线程所等待的受保护条件发生变化时，这种方法的效率就比较低了。通常，这种无用通知可以部分或者全部地通过对逻辑状态的分析来避免。为了避免在任何实例变量发生变化时都发出通知，你可以仅在线程所等待的逻辑状态的发生转变时再发出通知。以下的例子说明这些技巧。

3.3.1.1 通道与有界缓冲区

在多种并发软件的设计中，通道都扮演了非常重要的角色（参见§1.2.4 和§4.1）。通道的接口可以定义为：

```
interface Channel {
  void   put(Object x)  throws InterruptedException;
  Object take()         throws InterruptedException;
}
```

take 与 put 方法作为数据传输方法可以被视为类似于 Sync 的 acquire 与 release 的操作（参见§2.5.1），或不基于 IO 的流 read 和 write 操作，或传递操作封装版本（参见§2.3.4），或当通道中的基本单位是消息时，消息的 receive 与 send 操作（参见§4.1.1）。

有界缓冲区可以被用作为通道（其他可用作通道的类可参见§3.4.1）。有界缓冲区和有界计数器的总体结构是类似的。除了可用大小之外（计数器值），缓冲区还维护着一个固定

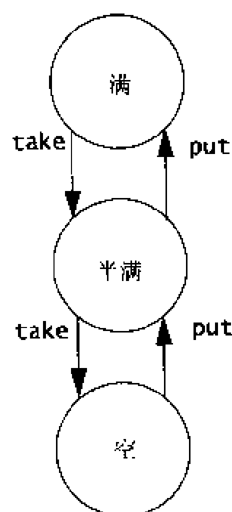
java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

大小的元素数组。缓冲区用 put 及 take 方法分别替代了计数器 inc 和 dec 方法。并且，MIN 就是零，MAX 就是容量（被定义为 int 类型以简化数组的索引工作）。

```
interface BoundedBuffer extends Channel {
  int capacity();     // INV: 0 < capacity
  int size();         // INV: 0 <= size <= capacity
}
```

如任何一本数据结构教科书所言，BoundedBuffer 可以用这种方法实现：构建一个固定长度的数组和两个索引指针，这两个指针循环地遍历这个数组，并且分别指向下一个 put 和下一个 take 的位置。BoundedBuffer 所定义的逻辑状态和状态转变与 BoundedCounter 类似：

状　态	条　件	put	take
满	size == capacity	不行	行
半满	0 < size < capacity	行	行
空	size == 0	行	不行



我们可以看到，只有从满状态和空状态离开的状态转变才会影响等待中的线程。这也就是当缓冲区的可用大小从零开始增加时或者从容量值开始减小时。

通过以上分析，BoundedBuffer 可以仅在离开满状态和空状态时才发出通知（这个代码简洁的原因部分是因为使用了后递增算子和后递减算子的编码模式所致）。

这个版本的 BoundedBuffer 所发出的通知次数远远少于那种每次缓冲区可用大小发生改变后都发出消息（这可能会导致毫无必要的线程唤醒）的版本。在那些重新校验保障的开销很大的情形中，通过这种方法来减少重新校验的次数可以极大地提高程序的性能。

```
class BoundedBufferWithStateTracking {
  protected final Object[]  array;     // the elements
  protected int putPtr = 0;            // circular indices
  protected int takePtr = 0;
  protected int usedSlots = 0;         // the count

  public BoundedBufferWithStateTracking(int capacity)
   throws IllegalArgumentException {
    if (capacity <= 0) throw new IllegalArgumentException();
    array = new Object[capacity];
  }

  public synchronized int size() { return usedSlots; }

  public int capacity() { return array.length; }

  public synchronized void put(Object x)
   throws InterruptedException {
```


java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

```
    while (usedSlots == array.length) // wait until not full
      wait();

    array[putPtr] = x;
    putPtr = (putPtr + 1) % array.length; // cyclically inc

    if (usedSlots++ == 0)             // signal if was empty
      notifyAll();
  }

  public synchronized Object take()
   throws InterruptedException{

    while (usedSlots == 0)          // wait until not empty
      wait();

    Object x = array[takePtr];
    array[takePtr] = null;
    takePtr = (takePtr + 1) % array.length;

    if (usedSlots-- == array.length) // signal if was full
      notifyAll();
    return x;
  }

}
```

3.3.1.2 状态变量

有时可以使用状态变量，即那些使用单一的成员变量来表示整个对象的逻辑状态的变量，来简化或更好地封装状态跟踪（参见 § 3.2.1.3）。通常，状态变量的值都为枚举类型。每次相关的成员变量更新过后，都要重新计算状态变量的值。这个重新计算的过程可以被封装在一个单独的方法中，如 updateState，然后在每次成员变量更新后调用它。在重新计算状态后，updateState 就会发出相关状态变化所关联的通知。例如，在 BoundedCounter 中（BoundedBuffer 也和此类似）使用状态变量的代码如下：

```
class BoundedCounterWithStateVariable  {
  static final int BOTTOM = 0, MIDDLE = 1, TOP = 2;
  protected int state = BOTTOM;  // the state variable
  protected long count = MIN;

  protected void updateState() { // PRE: synch lock held
    int oldState = state;
    if      (count == MIN) state = BOTTOM;
    else if (count == MAX) state = TOP;
    else                   state = MIDDLE;
    if (state != oldState && oldState != MIDDLE)
       notifyAll();                // notify on transition
  }

  public synchronized long count() { return count; }
```


java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

```
    public synchronized void inc() throws InterruptedException {
      while (state == TOP) wait();
      ++count;
      updateState();
    }

    public synchronized void dec() throws InterruptedException {
      while (state == BOTTOM) wait();
      --count;
      updateState();
    }
  }
```

除了使用 updateState 去重新计算对象状态，你还可以让每个执行更新操作的方法去决定下一个正确的状态，并把这个状态作为调用 updateState 的参数，随后，如果状态变化了，updateState 仍然会发出相应的状态变化通知（然而，这可能会带来 § 3.3.3.3 中所讨论的易碎性问题）。随着状态数量的增加，你可以使用更加复杂的机制，比如有穷状态机或者决策表（见进阶阅读）。

3.3.2 冲突集合

那些跟踪底层操作执行状态的类可以使用这些信息来决定如何处理新的请求。其最主要的应用之一便是构建定制的独占策略，相对于第 2 章中所介绍的，这种策略提供了更加细化的独占控制。

举个例子来说，设想一个 Inventory 类，它有 store 和 retrieve 方法来存取对象，每个对象都有不同的描述。假设这些操作都需要消耗时间，但是它们都被实现为不需要底层同步机制就可以正常工作了。在这种情况下，我们可以允许那些语义上互不冲突的操作同时执行，这就使得该类的并发性能好于那种完全同步的版本。

通常，应用这种策略的控件，其基本功能都是通过数据库的事务处理来实现的，但是在这里，我们将用 java.util.Hashtable 来演示。虽然这种完全同步的 Hashtable 可以允许 Inventory 的实现无需去考虑任何底层的同步细节，但是，我们仍然想要给 store 和 retrieve 方法添加一些语义上的约束。如下所示：

- retrieve 操作不应该和 store 操作并发执行，因为 store 方法正在存入的对象可能就是正在被请求取出的对象，而你并不想为此返回错误提示。
- 两个或者两个以上的 retrieve 操作不应该同时执行，因为其中一个操作正在取出的对象可能正是另一个操作所请求的。

我们还可以制定其他决策，例如，允许所有的操作都并发执行，并允许操作失败。同样，我们可以基于这些操作内部实现的细节来制定策略。例如，如果 retrieve 方法被编写成需要独占的，而 store 却不需要的话，那么上述的约束还是成立的。

某些正规或者半正规的符号被设计出来，用以帮助表示这种的信息。被最广泛采用的，同时这种并发问题也是足够了的方法就是冲突集合——不能同时出现的方法对的集合。比

java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

如，在这个例子中，这个冲突集合就是：

{ (store, retrieve), (retrieve, retrieve) }.

这种信息可以用在类的语义文档中，当通过跟踪执行状态来实现这些语义时，它们还可以作为实现的指南。

3.3.2.1 实现

基于冲突集合的类可以使用 before/after 这种设计（参见 § 1.4），即基本操作被那些维护着独占关系的代码所环绕。下面的机制可以通过任何的 before/after 模式来实现：

- 对于每个方法，定义一个计数变量，用以表示该方法是否在执行中。
- 把每个基本操作都隔离入非公共的方法中。
- 编写那些基本操作的公共版本，即在那些基本操作的前后添加上 before/after 的控制。
  - 每个同步的 before 操作都必须先等待所有非冲突的方法结束，这可从计数变量得知。随后，before 操作增加与该方法相关的计数变量的值。
  - 每个同步的 after 操作减少该方法的计数变量的值，并发出通知以唤醒等待中的线程。

在 Inventory 类上应用了以上的步骤后，其代码如下：

```
class Inventory {

  protected final Hashtable items = new Hashtable();
  protected final Hashtable suppliers = new Hashtable();

  // execution state tracking variables:

  protected int storing = 0;     // number of in-progress stores
  protected int retrieving = 0; // number of retrieves

  // ground actions:

  protected void doStore(String description, Object item,
                         String supplier) {
    items.put(description, item);
    suppliers.put(supplier, description);
  }

  protected Object doRetrieve(String description) {
    Object x = items.get(description);
    if (x != null)
      items.remove(description);
    return x;
  }

  public void store(String description,
                    Object item,
                    String supplier)
   throws InterruptedException {

    synchronized(this) {                    // before-action
      while (retrieving != 0) // don't overlap with retrieves
```


java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

```
      wait();
    ++storing;                              // record exec state
  }

  try {
    doStore(description, item, supplier); // ground action
  }

  finally {                                // after-action
    synchronized(this) {                   // signal retrieves
      if (--storing == 0) // only necessary when hit zero
        notifyAll();
    }
  }
}

public Object retrieve(String description)
 throws InterruptedException {

  synchronized(this) {                     // before-action
    // wait until no stores or retrieves
    while (storing != 0 || retrieving != 0)
      wait();
    ++retrieving;
  }

  try {
    return doRetrieve(description);        // ground action
  }

  finally {
    synchronized(this) {                   // after-action
      if (--retrieving == 0)
        notifyAll();
    }
  }
}
```

（由于这个冲突集合的特性，调用在 retrieve 方法中的 notifyAll 的条件总是满足的。然而，通常来说，通知只在条件满足时才应被发出。）

3.3.2.2 变种及扩展

在以上 Inventory 例子中所采用的方法也可以应用于乐观方法里，在其中，冲突常称为无效关系（invalidation relation）。它们通常这样被实现：在事务提交前中断冲突的操作，而不是一直等待到安全时才继续执行这些操作（参见 § 3.6）。

更加具扩展性的说明符号可被用于在更加详细的层次上表现冲突，它可以表达如下的冲突关系：某个方法 methodA 仅在 methodC 执行后才会同 methodB 冲突。类似的，在 Inventory 类中，我们可能想用一种更加准确的符号来描述这种情况：当 retrieve 方法正在执行中时，store 方法可以开始执行，但是反过来却不行。很多用于这种目的的说明符号已经被设计出来了，（参见 § 1.2.5 和 § 3.3.5 中的进阶阅读），这就使得我们能够更加详细地描述冲突，并通过使用执行状态跟踪变量来半自动地实现它。然而，从极端的角度来说，可能没有什么比

java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

一套完整历史日志能更好地实现给定的策略了。

在大多数依赖于冲突集合的类中，也可以使用 § 3.4 和 § 3.7 中所介绍的减少发出通知次数及减少上下文切换的技巧。

基于执行状态跟踪和冲突集合的实现可能是脆弱的且难以扩展。由于冲突集合是基于那些在类中已定义好的方法的，而不是基于其语义的逻辑表示或者底层的不变约束的，所以在子类中添加或者改变方法都是困难的。例如，如果加入一个按某种顺序进行排序的 sort 方法或者一个检测一项是否存在的 search 方法，它们就有可能与当前的处理有不同的冲突，因此需要重新设计。

在 § 3.3.3 中所描述的读出者和写入者模式，以及相关的构件通过把操作按照是否可以扩展进行分类，来部分地解决上述的问题。读出者和写入者模式同时解决了冲突集合无法解决的优先级与调度的问题。例如，在 Inventory 中，我们可能想添加这样一种规定：如果有多个等待的线程，那么等待执行 retrieve 方法的线程将优先于等待 store 方法的线程被执行，或者与此相反。

3.3.3 构建子类

可以通过构建子类来在已有机制上，添加不同层面的控制策略。或者与之相反。这种应用扩展了 § 2.3.3.2 中所见到的、在基本功能上添加锁定功能的子类用法。

3.3.3.1 读出者与写入者

读出者与写入者模式是这样一组并发控制构件：它们有着相同的基础，但是，一个负责控制那些执行读出操作的线程（读出者），而另一个负责控制那些执行改写或者改变状态操作的线程（写入者），它们并发控制的策略不一样。

在 § 2.5.2，我们看到了这种模式的一个版本。在那里，它被封装为一个工具类。下面，我们要介绍一个可被子类化的、应用了 before/after 设计的版本，这个版本使用了模板方法模式（参见 § 1.4.3）。除了作为工具类以外，对于那些混合了基于子类化的 before/after 的并发控制，以及使用计数器来记录的消息和操作的控制策略来说，这种设计无疑是个很好的模式。例如，非常类似的技术可以应用于那些需要按照特定顺序、成对地接收消息的类（例如，要求按这个顺序，read、write、read、write 的顺序，等等）。这种技术也可以应用于有目的的锁（intention lock），对于给定容器的可访问的所有对象，这种锁为它们在以后获取（或升级为）读/写锁做好了准备（参见 § 2.4.5）。

在添加控制机制之前，必须先制定一套如何管理它们的策略。读出者与写入者是一种通用并发控制策略，例如，§ 3.3.2 中的 Inventory 类就使用了类似的策略。但是与之不同是，读出者和写入者策略不是仅用于控制那些特定的方法的，而是控制所有具有读出与写入语义的方法的。但是，这些控制策略的细节还是和具体的情况相关。需要考虑以下几点：

- 如果当前已经存在一个或者多个活动的（执行中的）读出者，且已有一个写入者在等待的时候，一个新的读出者是否能够立即加入？如果答案是肯定的话，那么不断增加的读出者将会使得写入者无法执行；如果答案为否，那么读出者的吞吐量就会下降。

java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

- 如果某些读出者与写入者同时在等待一个活动的写入者完成操作，那么你的处理策略会偏向于读出者还是写入者？先到者优先？随机？轮流？类似的选择也会在发生读出者终止后。
- 是否允许写入者在不释放锁的同时降级成为读出者？

虽然这些策略上的问题没有明确的答案，但是一些标准的解决方案和相关的实现还是存在的。我们将举例说明这种常见的策略：如果当前有等待中的写入者，那么读出者线程将被阻塞，等待中的写入者的执行顺序是任意的（仅仅依赖于底层 JVM 调度程序来排列这些线程的恢复顺序），并且写入者不可以降级为读出者。

这种同步控制策略的实现需要跟踪执行状态。同其他策略一样，这可以通过记录当前正参与读出或者写入操作的线程数量，以及那些正在等待的线程数量的计数器来实现。在这里，跟踪等待中的线程数量，是对冲突集合实现的主要扩展。

为了构建相应的实现，控制代码应该在围绕在真正的读写代码的方法对中，而那些真正的读写方法，则需在子类中实现。before/after 的设计（参见 § 1.4.3）允许构建任意数量公共的读出类型或者写入类型的方法，每个公共方法都在 before/after 方法对的中间调用那些非公共的方法。

下面这个版本是通用的，其子类基本不需要做太多的修改。值得一提的是，对等待读出者的计数在这个版本中实际上是没有必要的，因为实际上并没有依赖于它的策略。但是，由于它的存在，你可以通过让 allowReader 和 allowWriter 方法中的谓词依赖于这个值，来调整控制策略。例如，你可以修改这些方法中条件，来决定优先处理等待队列中计数器值更大的线程。

```
abstract class ReadWrite {
  protected int activeReaders = 0;  // threads executing read
  protected int activeWriters = 0;  // always zero or one

  protected int waitingReaders = 0; // threads not yet in read
  protected int waitingWriters = 0; // same for write

  protected abstract void doRead(); // implement in subclasses
  protected abstract void doWrite();

  public void read() throws InterruptedException {
    beforeRead();
    try     { doRead(); }
    finally { afterRead(); }
  }

  public void write() throws InterruptedException {
    beforeWrite();
    try     { doWrite(); }
    finally { afterWrite(); }
  }
  protected boolean allowReader() {
    return waitingWriters == 0 && activeWriters == 0;
  }
```


java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

```
  protected boolean allowWriter() {
    return activeReaders == 0 && activeWriters == 0;
  }

  protected synchronized void beforeRead()
   throws InterruptedException {
    ++waitingReaders;
    while (!allowReader()) {
      try { wait(); }
      catch (InterruptedException ie) {
        --waitingReaders; // roll back state
        throw ie;
      }
    }
    --waitingReaders;
    ++activeReaders;
  }

  protected synchronized void afterRead()  {
    --activeReaders;
    notifyAll();
  }

  protected synchronized void beforeWrite()
   throws InterruptedException {
    ++waitingWriters;
    while (!allowWriter()) {
      try { wait(); }
      catch (InterruptedException ie) {
        --waitingWriters;
        throw ie;
      }
    }
    --waitingWriters;
    ++activeWriters;
  }

  protected synchronized void afterWrite() {
    --activeWriters;
    notifyAll();
  }
}
```

这个类或它的子类也可以被重新包装一下以支持 § 2.5.2 中所介绍的 ReadWriteLock 接口。这可以通过内部类来实现（类似的策略也被用在 util.concurrent 的 ReadWriteLock 中，不过其中也包含了 § 3.7 中所讨论的一些优化技巧来避免不必要的通知）。例如：

```
class RWLock extends ReadWrite implements ReadWriteLock {
  class RLock implements Sync {
    public void acquire() throws InterruptedException {
      beforeRead();
    }

    public void release() {
```


java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

```
      afterRead();
    }

    public boolean attempt(long msecs)
     throws InterruptedException{
       return beforeRead(msecs);
    }
  }

  class WLock implements Sync {
    public void acquire() throws InterruptedException {
      beforeWrite();
    }

    public void release() {
      afterWrite();
    }

    public boolean attempt(long msecs)
     throws InterruptedException{
       return beforeWrite(msecs);
    }
  }

  protected final RLock rlock = new RLock();
  protected final WLock wlock = new WLock();

  public Sync readLock()  { return rlock; }
  public Sync writeLock() { return wlock; }

  public boolean beforeRead(long msecs)
   throws InterruptedException {
    // ... time-out version of beforeRead ...
  }

  public boolean beforeWrite(long msecs)
   throws InterruptedException {
    // ... time-out version of beforeWrite ...
  }
}
```

3.3.3.2 将保障分层

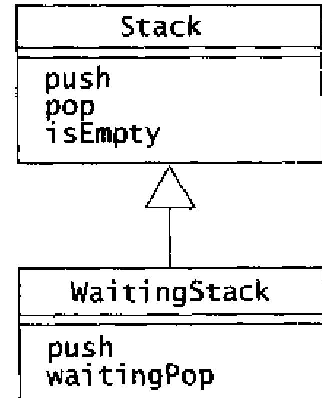
保障可以被添加到那些被编写为阻碍模式的基本数据结构类中。例如，考虑一下这个简单的 Stack:

```
class StackEmptyException extends Exception { }

class Stack {                                          // Fragments

  public synchronized boolean isEmpty() { /* ... */ }

  public synchronized void push(Object x) { /* ... */ }

  public synchronized Object pop() throws StackEmptyException {
    if (isEmpty())
```


java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

```
        throw new StackEmptyException();
      // else ...
    }
  }
```

在单线程的情况下，当从空 Stack 中 pop 出元素时，操作被阻碍并返回是非常有用的，因为在串行执行的程序中，由于不会有其他的线程在 Stack 中添加元素，所以等待 pop 的操作是毫无必要的，否则程序将无休止地等待下去。但是换一个角度来看，在并发的环境中，Stack 的客户程序可能希望暂停并等待 Stack 中出现一个元素。一个低效的方法是当执行 pop 时，一旦捕捉到了 StackEmptyException 异常，就重试 pop 操作。这是一种拙劣的忙等待方式。



直接支持保障的版本可以通过构建子类，并在子类中添加相应的控制方法来实现。不过，在这个类中，直接覆盖 pop 方法并不是一个好主意。而且，从另一个角度来考虑，阻碍与等待的不同控制策略反映在不同的方法签名上：使用阻碍的 pop 方法会抛出 StackEmptyException 异常，而使用等待的 pop 方法却不会；反过来说，使用等待的 pop 方法会抛出 InterruptedException 异常，而使用阻塞的版本就不会抛出这个异常。虽然在某些接口中，这两种方法也可以被合并，但是，如果将这两个方法分开来处理，程序会更易于维护。

虽然如此，不去修改 pop 方法而在子类中添加 waitingPop 方法还是可以的。然而要注意的是，push 方法还是必须被覆盖以通知那些被 waitingPop 所阻塞的线程（这里对 notifyAll 的调用是可以被优化的）。

```
class WaitingStack extends Stack {

  public synchronized void push(Object x) {
    super.push(x);
    notifyAll();
  }

  public synchronized Object waitingPop()
   throws InterruptedException {

    while (isEmpty()) {
      wait();
    }

    try {
      return super.pop();
    }
    catch (StackEmptyException cannothappen) {
      // only possible if pop contains a programming error
      throw new Error("Internal implementation error");
    }
  }
}
```


java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

3.3.3.3 继承变异

某些并发 OO 编程语言（参见进阶阅读）在语法上就对方法的定义有这样的要求：由非公共方法来定义功能，由公共方法来定义并发控制的策略；换句话说，它们要求程序中的方法按照类似于采用了模板方法的 ReadWrite 类的形式进行区分。即使在这种区分没有被严格时，这也是一种非常值得采用的方法：

- 这可以使子类能够独立地对执行功能的代码或者执行并发控制代码进行修改，从而使得子类无需修改所有父类的方法就可以得到所需的功能。
- 这可以避免为并发功能而存在的成员变量与为实现基本功能而存在的逻辑状态变量之间的混淆。它们可以在子类中再出现。
- 这可以避免不使用这种区分方法时，那些围绕在独占控制、访问内部变量或者方法、对象身份、嵌套的监视器（参见 § 3.3.4），以及接口改写周围的问题。构建子类是对已有对象的扩展而非组合。例如，没有必要去保证子类对象中的超类数据是被子类对象完全所拥有的。

只要所有的相关成员变量和方法都被声明为 protected，子类就可以对基类代码进行必要的修改以获得想要的控制策略。尽管基类作者在初始设计中可能有非常好的思路，但有时在子类中对基类方法进行修改是实现想要的控制策略的惟一途径。虽然从设计角度来说，protected 的存取控制的确存在问题，但是在并发环境中，这可以使得子类有能力去改变并发控制的策略，因此，其带来的好处是多于那些因滥用超类成员变量所带来的问题。

为了使子类能够正确继承父类，必须在文档中对类中的不变约束进行详细说明。相对于那些把所有成员变量都公共化（即使通过 set/get 方法），并指望外部客户程序能够找出如何保持对象的一致性、如何保持语义上的不变约束，以及如何使新的操作或者策略满足原子性需求的类来说，那些依赖于 protected 的成员变量、方法及在文档中详细说明的约束的父类是更容易被正确扩展的。

但是，这种类型的继承也有其局限性。当刚开始使用那些实验性的并发 OO 语言时，一些研究人员注意到，通过定义子类来在父类中添加或者扩展通用的功能或控制策略有时是非常困难甚至于不可能的。类似的问题也出现在高阶的 OO 分析及设计方法中。

同样，对于很多串行执行的类，其结构也是很难扩展的，例如，那些被毫无目的地声明为 final 类型的方法。但是，在并发 OO 程序中所遇到的阻碍已经多到可以用一个单独的术语来描述它了，即继承变异（inheritance anomaly）。这个术语所含盖的问题的相互之间的联系比较松散。例如：

- 对于那些在父类方法中未提供相应通知的条件，如果子类中包含了和其相关的受保护等待，那么父类中的那些方法必须被重写。这可以从 WaitingStack 类中看到（§ 3.3.3.2），其中的 push 方法就是因为要为新方法 waitingPop 提供通知而被改写。
- 类似的，如果父类使用了 notify 而非 notifyAll，且由于子类添加的功能导致使用 notify 的必要条件不再被满足时，父类中所有发出通知的方法都必须被重写。
- 如果父类没有明确地描述或者跟踪那些子类方法所需要的逻辑或执行状态，那么所有需要跟踪或检测那些状态的方法都要被重写。

java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

- 使用状态变量（§3.3.1.2）会使了类的扩展被限制在父类里所定义的逻辑状态之中。因此，子类必须遵守与父类相同的逻辑状态的抽象规范。这种做法在某些上层的 OO 分析与设计中被推荐，但这也会影响构建子类的效果。例如，假设你想这样来扩展 BoundedCounterWithStateVariable 类：添加一个 disable 方法来阻塞 inc 和 dec 方法，以及一个 enable 方法使它们继续执行。对这两个方法的支持就会引入新的逻辑状态，并使得基本方法中的保障与通知的条件发生改变。

总体来说，以上的问题给我们提出了如下警告：在设计并发程序时所需花费的精力通常比串行执行的程序更多，否则，设计出的并发程序很可能是很难以使用或者扩展的。虽然这些问题没有好记的名字，但是相同的问题也会在试图聚集、组合或者委托对象时遇到。

避免某些最常见扩展性问题的方法之一就是把保障和通知都封装在可覆盖的方法中，然后按如下的形式构建公共方法：

```
public synchronized void anAction() {
  awaitGuardsForThisAction();
  doAction();
  notifyOtherGuardsAffectedByThisAction();
}
```

然而，与串行执行的 OO 程序设计一样，没有通用的方法可以使得父类能够适合各种扩展情况并在任何扩展情况下都不需要修改。大多数编写类时避免这些问题的指导方针可以归结为一下两点：1. 避免过早的优化；2. 封装设计决策。

令人吃惊的是，这两点规则是如此地难以遵守。通常情况下，对已知情况不优化比优化需要更多的抽象，且会对设计有更多的束缚。类似的，你无法在设计决策还未做出的时候就封装这个设计决策。这就需要你在写一个类时考虑到当前尚未出现的情况。

类似于这样的规则通常都是在重新回顾代码、清除无用代码并使代码复用性更高时才得以应用。理想情况下，你可以预测到为了使一个类可重用并扩展所需要的各个方面。但是，实际往往并非如此。回顾时的重构及迭代的修改是软件开发中非常重要和基本的一部分。

3.3.4 限制及嵌套的监视器

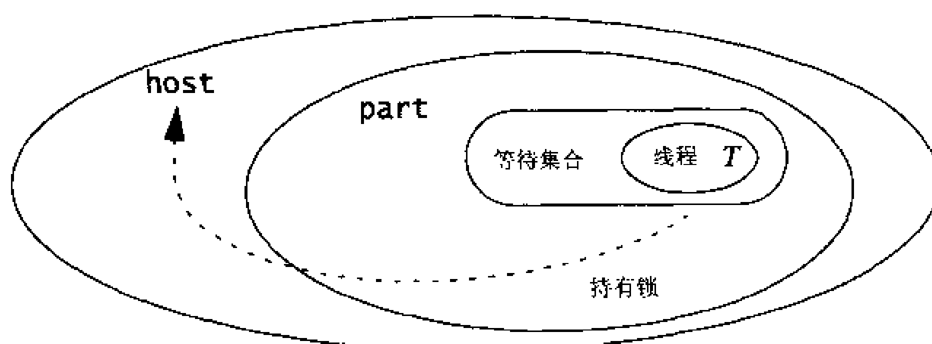
如 §2.3.3 和 §2.4.5 中所讨论的那样，通常来说，将同步的 Part 对象限制在同步的 Host 对象中是可行的，最多也就是带来过多锁的开销而已。然而，当 Part 对象使用了等待和通知方法后，情况就变得非常复杂了。与这种情况相关的问题通常被称为嵌套的监视器问题（the nested monitor problem）。为了说明其中潜在的死锁问题，考虑一下这个例子：

```
class PartWithGuard {
  protected boolean cond = false;

  synchronized void await() throws InterruptedException {
    while (!cond)
      wait();
    // any other code
  }
```


java学习群：72030155, 每天20:30-23:00都有大神视频教学, 想学习的同学可以进群免费听课！

```
    synchronized void signal(boolean c) {
      cond = c;
      notifyAll();
    }
  }

  class Host {
    protected final PartWithGuard part = new PartWithGuard();

    synchronized void rely() throws InterruptedException {
      part.await();
    }

    synchronized void set(boolean c) {
      part.signal(c);
    }
  }
```

受保护的挂起只有在你确信其他线程最终能够解除 wait 时才是有意义的。但是，在这个例子中，Host 类却阻止了其他线程做这样的工作。该问题是这样造成的：某个对象的等待集合中的所有线程，除了这个对象的同步锁以外，它们一直保持着其他对象的同步锁。例如，假设一个线程 ***T*** 调用了 host.rely 方法而导致其在 part 中被阻塞。而当 ***T*** 被阻塞时，它所拥有的 host 的锁还是保持着的，因此，没有其他的线程能够通过执行 host.set 来解除这个锁。



当普通的同步方法调用其他一样普通但却使用了 wait 的同步方法时，这种嵌套的问题就可能会导致意想不到的死锁。如同处理其他状态依赖行为的策略一样，你需要用详细的文档去说明类中所使用的等待策略，这样当别人试图去使用这个类时，就可以正确地解决这个潜在的问题。在受保护方法签名上添加 InterruptedException 就是一个好的开始。

有两个方法可以避免嵌套的监视器所带来的死锁问题。第一，也是最简单的（事实上就是 § 1.1.1.1 所介绍的默认规则的一个应用）一个方法就是在那些调用 Part 对象方法的 Host 对象方法中不使用同步。当对 Host 对象的调用是无状态的时候，就可以使用这种方法（参见 § 2.4.1）。

在其他情况中，如果 Part 对象的方法必须要访问被锁定的 Host 对象的状态，则可以重新定义 Part 的类，让其使用一种扩展形式的分层的容器锁（参见 § 2.4.5），这种方法将把 Host 对象作为监视器。例如：

java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

```
class OwnedPartWithGuard {                              // Code sketch
  protected boolean cond = false;
  final Object lock;
  OwnedPartWithGuard(Object owner) { lock = owner; }

  void await() throws InterruptedException {
    synchronized(lock) {
      while (!cond)
        lock.wait();
      // ...
    }
  }

  void signal(boolean c) {
    synchronized(lock) {
      cond = c;
      lock.notifyAll();
    }
  }
}
```

3.3.5 进阶阅读

对继承变异更多的讨论和例子可以在 § 1.2.5 中所列的 Agha、Wegner 及 Yonezawa 所编辑的集锦中找到，也可以在最近的 OO 会议，例如 ECOOP 中的论文、§ 1.4.5 中所列的 David Holmes 的论文和下面的文章中找到：

McHale 和 Ciaran.《Synchronization in Concurrent Object-Oriented Languages, PhD Thesis, Trinity College》, Ireland, 1994.

组合过滤器（Composition-Filter）系统是需要将同步控制和业务功能分离的 OO 开发框架的例子，其中包含了一套比冲突集合更加具有扩展性的表示法用以描述并发控制约束。可以参见 Mehmet 的论文以及 § 1.2.5 中所列出的由 Guerraoui，Nierstrasz 和 Riveill 所编辑的集锦。

表示状态和状态转移（例如，使用有限状态机）技术的描述可以在 § 1.3.5 列出的 OO 和并发软件设计中找到。其他的模式可以参见：

Dyson, Paul 和 Bruce Anderson. "State Patterns", in Robert Martin, Dirk Riehle, and Frank Buschmann(eds.) 《Pattern Languages of Program Design, Volume 3》, Addison-Wesley, 1998.

RWLock 中所使用的内部类是 Extension Object 模式的一个简单的、不支持查询的版本。详见：

Gamma 和 Erich. "Extension Object", in Robert Martin, Dirk Riehle, and Frank Buschmann(eds.), 《Pattern Languages of Program Design, Volume 3》, Addison-Wesley, 1998.

3.4 使用并发控制工具类

内建的等待与通知方法可以用来实现任何依赖状态的协调机制。但是，它们也带来了相

java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

关的三个难题：

- 等待与通知方法的需求与特性往往涉及类设计的不同方面，这就导致了概念上和代码上的复杂性。例如，虽然§3.3.3中所介绍的使用了模板方法的读出者与写入者的版本是可靠且灵活的，但是，相对于§2.5.2中那个实现了ReaderWriteLock的版本来说，使用它需要对底层的设计有更多了解。
- 虽然对监视器方法的简单应用十分简单，但是当其他因素添加进来后，尤其是在处理线程取消中的性能和稳定性问题时，出现错误（例如错误状态）的几率就会大大增加。当这些解决方案被封装为工具类时，处理所有情况的复杂工作就只需做一次了。即使构造的工具类会给使用者有带来一些编程约束，但只要使用它们相对于重新实现来说，不会带来更多的困难、不易出错，那么就是值得的。为了提高软件的质量，工具类应该封装那些不是很容易实现的简单概念，并尽可能地便于使用。
- 虽然无数的设计在理论上都可以通过使用受保护方法来实现，但是，实际使用中，它们大部分都可以归入几种通用的类别之中。大多数这些代码都可以被重用，而不必在每次使用是都从头写起。同时，这也把并发控制策略的选择及其实现清楚地分开来了。

这一节讨论了工具类及其应用的四个典型的例子，包括使用简单的工具类来构造更加复杂工具类的例子。还有一些其他的例子将在本书的后面部分介绍。为了具体一些，本节介绍的都是util.concurrent包中的版本，但是基本上所有的讨论都可以应用于其他的你可以创建的实现上。这些类的大多数实现细节将在§3.7中进行介绍（可能只有编写这些类的定制版本的开发人员对此感兴趣）。

3.4.1 信号机

信号机（更确切地说，计数信号机counting semaphore）是并发控制中的经典构件。同其他工具类一样，它们也遵守获得一释放协议，因此也同样支持§2.5里介绍的Mutex类中的Sync接口。

从概念上说，一个信号机维护着一组在构造方法中初始化了的许可证。如果必要的话，每次acquire操作都会阻塞直到有一个许可证可用，然后占用这个许可证。attempt方法执行类似的操作，但是它可以在超时的时候失败并退出。每一次release都会添加一个许可证。不过，事实上并没有使用真实的许可证对象；信号机只需知道当前可用的许可证的数量并执行相应的操作即可。

也有其他方法来描述信号机，其中包括它们最初始动机的寓意：用于防止铁路碰撞的信号旗帜。

3.4.1.1 互斥锁

信号机可以被用于实现互斥锁，只需将初始化的许可证数置为1即可。例如，一个Mutex类可以这样定义：

java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

```
class Mutex implements Sync {
  private Semaphore s = new Semaphore(1);

  public void acquire() throws InterruptedException {
    s.acquire();
  }

  public void release(); {
    s.release();
  }

  public boolean attempt(long ms) throws InterruptedException {
    return s.attempt(ms);
  }
}
```

这种类型的锁也被称为二元信号机（binary semaphore），因为其许可证的计数器的值只能取 0 或者 1。这里所没有（但是可以）提到的一个微小细节是，按照一般惯例，释放一个并没有被占有的 Mutex 是不会产生任何效果的（一个不是很常用的处理方法是抛出一个异常）。另外，实际上定义一个 Mutex 并不是完全必要的，一个初始化参数为 1 的 Semaphore 可以被直接用作为锁，在这种情况下，额外的释放操作会被记录下来并因此允许额外的获取操作。虽然这并不是所期望的特性，但是，在那些与锁定无关的环境中，这种特性可以被用于解决丢失信号的问题（参见 § 3.2.4.1）。

由于信号机可以被用作锁或者其他类型的并发控制构件，它们足以被用作一个并发构件的基础构件。例如，可以使用信号机来实现 synchronized 方法锁、wait、notify 以及 notifyAll 操作，但是反过来却不行（详情参见 § 1.2.5 的进阶阅读中所列的 Andrew 书中的例子）。

某些系统和语言事实上就将信号机作为它们所提供的惟一并发控制构件。不过，相对于那些块状结构的锁定，例如通过 synchronized 方法和 synchronized 程序块来实现的锁定，以及使用 Mutex 构造的 before/after 构件来实现的锁定，过度依赖信号机来实现互斥会更复杂且容易出错。相对于被用作锁，信号机在用于计数和发信号时更有价值。

3.4.1.2 资源池

信号机是一种特定的计数器，因此在大多数和计数器相关的类中，它也是并发控制构件的自然选择。例如，各种的资源池类通常都要记录其资源的数目（例如，文件描述符、打印机、缓冲及大型图像对象）以便用户取出并在用后放回。

以下的这个 Pool 类展示了大多数资源池的基本构架。这个类只包含了一个通用且有效的保护机制，用于确保被放回资源池中的资源确实被取出过。此外还可以添加其他保护机制，例如，检测调用者是否有权获取资源。

为了符合这种取出/放回的协议，使用资源池的程序通常应该采用 before/after 模式，就像这样：

```
try {
  Object r = pool.getItem();
  try { use(r); }
```


java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

```
      finally { pool.returnItem(r); }
    }
    catch (InterruptedException ie) {
      // deal with interrupt while trying to obtain item
    }
```

Pool 类显示了绝大部分使用并发工具类的类中都存在的分层结构特性：在公共的非同步控制方法中包涵着内部 synchronized 的辅助类的方法。独占在 doGet 和 doReturn 方法中是必须的，因为可能有多个客户线程通过了对 available.acquire 的调用。如果没有锁定的话，很有可能同时会有多个线程并发地操作内部的列表。从另一方面来看，如果将 getItem 和 returnItem 定义成 synchronized 的，那么将会造成一个错误。因为这样做不仅仅是没有意义的，而且，当一个在 acquire 中等待的线程拥有其他执行 release 操作的线程所需要的锁时，就会造成嵌套监视器错误（参见 § 3.3.4）。

```
class Pool {                                           // Incomplete
  protected java.util.ArrayList items = new ArrayList();
  protected java.util.HashSet busy = new HashSet();

  protected final Semaphore available;

  public Pool(int n) {
    available = new Semaphore(n);
    initializeItems(n);
  }

  public Object getItem() throws InterruptedException {
    available.acquire();
    return doGet();
  }

  public void returnItem(Object x) {
    if (doReturn(x))
      available.release();
  }

  protected synchronized Object doGet() {
    Object x = items.remove(items.size()-1);
    busy.add(x); // put in set to check returns
    return x;
  }

  protected synchronized boolean doReturn(Object x) {
    if (busy.remove(x)) {
      items.add(x); // put back into available item list
      return true;
    }
    else return false;
  }

  protected void initializeItems(int n) {
    // Somehow create the resource objects
    //   and place them in items list.
  }
}
```


java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

需要注意的是，由于在这里使用了 HashSet，所以，这些资源类在改写 equals 方法时不能破坏其基于对象标识的比较算法（参见 § 2.1.1），因为这种算法是维护资源所必需的。

3.4.1.3 有界缓冲区

只要你的设计从概念上是采用许可证机制的，那么信号机就是非常有用的工具。例如，我们可以基于以下的思想来设计一个 BoundedBuffer：

- 最初，对于一个大小为 *n* 的缓冲区来说，有 *n* 个放入许可证和 0 个取出许可证。
- 一个 take 操作必须先获得一个取出许可证，随后释放一个放入许可证。
- 一个 put 操作必须先获得一个放入许可证，随后释放一个取出许可证。

为了更加便利地使用信号机，可以把底层数组的操作封装到一个简单的 BufferArray 辅助类中（事实上，如 § 4.3.4 中所示，一个底层数据结构完全不同的类，例如链表，可以在不对这个设计的逻辑做任何修改的情况下被采用）。BufferArray 类使用了 synchronized 方法，当多个客户在获得许可证后并发地执行插入或者取出操作时，这可以保证操作的独占性。

```
class BufferArray {
  protected final Object[]  array;      // the elements
  protected int putPtr = 0;             // circular indices
  protected int takePtr = 0;
  BufferArray(int n) { array = new Object[n]; }

  synchronized void insert(Object x) {  // put mechanics
    array[putPtr] = x;
    putPtr = (putPtr + 1) % array.length;
  }

  synchronized Object extract() {       // take mechanics
    Object x = array[takePtr];
    array[takePtr] = null;
    takePtr = (takePtr + 1) % array.length;
    return x;
  }
}
```

对应的 BoundedBufferWithSemaphore 类在缓冲区操作的周围添加上信号机操作以实现 put 和 take。虽然每个方法都由 acquire 方法开始并由 release 方法结束，但它们采用的使用模式与 § 2.5 中所使用的锁的模式不同。释放操作使用的信号机是与获得操作所使用的信号机不同，并且仅仅当元素被成功地插入或取出后才会被执行。释放操作并未放在 finally 语句中的原因如下：如果对缓冲区的操作在某种情况下失败了，某些恢复操作是需要的，但最后的释放语句却是不需要的。

```
class BoundedBufferWithSemaphores {
  protected final BufferArray buff;
  protected final Semaphore putPermits;
  protected final Semaphore takePermits;

  public BoundedBufferWithSemaphores(int capacity) {
    if (capacity <= 0) throw new IllegalArgumentException();
    buff = new BufferArray(capacity);
```


java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

```
    putPermits = new Semaphore(capacity);
    takePermits = new Semaphore(0);
  }

  public void put(Object x) throws InterruptedException {
    putPermits.acquire();
    buff.insert(x);
    takePermits.release();
  }

  public Object take() throws InterruptedException {
    takePermits.acquire();
    Object x = buff.extract();
    putPermits.release();
    return x;
  }

  public Object poll(long msecs) throws InterruptedException {
    if (!takePermits.attempt(msecs)) return null;
    Object x = buff.extract();
    putPermits.release();
    return x;
  }

  public boolean offer(Object x, long msecs)
   throws InterruptedException {
    if (!putPermits.attempt(msecs)) return false;
    buff.insert(x);
    takePermits.release();
    return true;
  }
}
```

这个类中也包含有 put 和 take 方法的变体，offer 和 poll 方法，它们支持阻碍（当 msecs 为 0 时）或者超时策略。这些方法用 Semaphore.attempt 来实现，它们负责处理那些 § 3.2.5 中所描述的、与超时相关的问题。offer 和 poll 方法允许客户程序选择它们所需的保护策略。但是，客户程序必须指定相互兼容的操作策略。例如，如果一个生产者仅使用 offer(x, 0)，但是只有使用 poll(0)的消费者，那么缓冲区中的内容不会被传送。

当有多个线程同时使用缓冲区时，BoundedBufferWithSemaphores 类应该比 § 3.3.1 中所介绍的 BoundedBufferWithStateTracking 类的效率更高。BoundedBufferWithSemaphores 类依赖于两套底层的等待集合，而 BoundedBufferWithStateTracking 类只有一个。所以，任何从空到有或者从有到空的状态转变都会唤醒所有的等待线程，包括那些等待其他逻辑条件的线程，以及由于其他线程取走了惟一的内容或者填满了最后一个空位而将在唤醒后立即重新进入等待状态的线程。

BoundedBufferWithSemaphores 类对这两种条件使用了不同的监视器。底层的 Semaphore 实现可以利用这一点来通过使用 notify 代替 notifyAll 以避免无需的上下文切换（参见 § 3.7.1）。这就使得最坏情况下被唤醒的线程数与实际调用的线程数之间的关系从二次方下降为线性关系。更一般地说，无论何时，相对于使用基于 notifyAll 的方案，使用信号

java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

机来隔离状态都会提高程序性能。

3.4.1.4 同步通道

如 § 3.3.1 中所提到的那样，BoundedBuffer 的接口可以被扩展以描述任何支持 put 和 take 操作的 Channel。

```
interface Channel {                                 // Repeated
  void   put(Object x) throws InterruptedException;
  Object take()         throws InterruptedException;
}
```

（util.concurrent 包中这个接口的版本也包含了支持超时的 offer 和 poll 方法，并且是继承了 Puttable 和 Takable 接口的，这使得该类可以被用来仅支持单 操作。）

你可以给 Channel 添加许多语义。例如，§ 2.4.2 中的队列类拥有无限的容量（至少从概念上是无限的——直到用完系统的所有内存后才会失败），而有界缓冲区的容量却只有有限的预定义的容量。一个极端的例子就是没有容量的同步通道。对于同步通道来说，每个试图执行 put 操作的线程都要先等待某个试图执行 take 操作的线程执行完后才可执行，反之亦然。这样，§ 4.1.4 和 § 4.5.1 中所介绍的框架和模式对线程交互进行精确控制的需求就可以被满足了。

信号机可以被用来实现同步通道。在这里，我们可以使用同有界缓冲区相同的方法，再加上一个信号机以确保只有当 put 操作所提供的某项内容被取走后，这个 put 操作才可以继续执行。到目前为止，我们仅使用了在方法的第一行抛出 InterruptedException 异常的阻塞模式，以允许中断时执行某些简单的清除工作。但是，我们需要在 put 方法的最后执行第二个 acquire 操作。如果在这个 acquire 在操作时被中断，那么该协议就会被破坏。虽然定义这个类的一个可以完全回滚的版本是可能的，但是在这里，最简单的方案就是前滚（参见 § 3.1.1.4），即忽略任何中断操作直到第二个 acquire 操作完成。

```
class SynchronousChannel implements Channel {

  protected Object item = null; // to hold while in transit

  protected final Semaphore putPermit;
  protected final Semaphore takePermit;
  protected final Semaphore taken;

  public SynchronousChannel() {
    putPermit = new Semaphore(1);
    takePermit = new Semaphore(0);
    taken = new Semaphore(0);
  }

  public void put(Object x) throws InterruptedException {
    putPermit.acquire();
    item = x;
    takePermit.release();

    // Must wait until signaled by taker
    InterruptedException caught = null;
```


java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

```
      for (;;) {
        try {
          taken.acquire();
          break;
        }
        catch(InterruptedException ie) { caught = ie; }
      }

      if (caught != null) throw caught; // can now rethrow
    }

    public Object take() throws InterruptedException {
      takePermit.acquire();
      Object x = item;
      item = null;
      putPermit.release();
      taken.release();
      return x;
    }
  }
```

3.4.1.5　公平性与调度

内置的等待与通知的方法并未提供任何形式的公平性保证。它们不保证哪一个在等待集合中的线程会在 notify 的操作中被选中，或者在 notifyAll 的操作中，那个线程会先获得锁（这样就排斥了其他线程）并继续执行下去。

JLS 所允许的由 JVM 实现的灵活性使得我们无法精确证明一个系统的活跃性特征。但在大多数的情况下，这都不是实际操作中的主要问题。例如，在大多数缓冲区的应用中，究竟是哪个线程获得执行 take 的机会是无关紧要的。在资源池的管理类中，必须明确保证正在等待所需资源的线程，不会因为底层的 notify 操作恢复线程的不公平性，而不断被其他线程所排挤以致没有机会运行。类似的问题在很多同步通道的应用中也会碰到。

虽然改变 notify 的语义是不可能的，但是改变 Semaphore 类（子类）的 acquire 操作以提供更好的公平性却是可以的。这种方法可以支持一系列的策略，它们根据公平性定义的不同而不同。

最为人所熟知的策略就是先入先出（FIFO），即等待时间最长的线程最先被选中。这在直觉上是让人满意的，但是，在多个处理器系统中，如果多个线程（近似）同时在不同的处理器上开始等待，这种策略也可能是毫无必要的甚至是武断的。然而，各种 FIFO 的弱化或者类似的解决方案为应用程序提供了避免不确定性延迟问题的方法。

不过，这种保证还是有其本质上的限制的：没有办法可以确保底层的系统确实会执行一个给定的进程或者线程，除非系统提供了 JLS 所需的最小要求之外的额外保证。不过，这在实际应用中并不是一个特别突出的问题。即使不是全部也是大多数的 JVM 实现都尽力提供了一个在最小需求上适当扩展的合理调度机制。虽然在它们执行可运行的线程时，可能会有某些弱点（限制或者随机的公平性问题），但是，让一个语言规范去规定所有可能出现的情况的解决方案是非常困难的。所以这被留作了 JLS 实现质量的问题。

类似于信号机的工具类是用以构建不同的公平性策略及解决调度问题的利器。例如，在

java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

§3.7.3中描述的FIFOSemaphore的实现，它维护了FIFO的通知顺序。类似于Pool类的应用可以使用这种或者其他类型的、提供了公平性功能的信号机，不过这可能会增加额外的消耗。

3.4.1.6 优先级

除了解决公平性问题以外，信号机的实现类也可以处理线程优先级的问题。notify方法并未保证按优先级顺序去唤醒线程，但这样做显然是允许的，并且在某些JVM中得到了实现。

优先级的设定（参见§1.1.2.3）往往仅在可执行的线程数多于CPU的个数，并且这些线程中的任务之间存在着本质的紧急性或重要性区别时，才是有价值的。这种情况往往出现在嵌入式（软件）实时系统中。在这种系统中，一个微处理器必须执行很多与其环境相交互的任务。

对优先级设定的依赖会使得通知策略变得更加复杂。即使发出的通知会使阻塞的线程按照线程优先级的顺序来恢复（执行），系统仍然可能会遭遇优先级颠倒的现象。当高优先级的线程因为要等待低优先级的线程先完成，并随后释放锁或者改变高优先级的线程所需要的条件时，优先级颠倒的现象就会发生。在精确的优先级级调度系统中，如果低优先级的线程没有机会运行的话，这就会导致高优先级的线程处于“饥饿”状态。

解决这个问题的方案之一就是使用特定的信号机类或者使用这种信号机所构建的锁。在这里，并发控制对象自己处理优先级问题。当高优先级的线程被阻塞后，并发控制对象会临时提高那些会使得高优先级线程恢复执行的低优先级线程的优先级。这个现象揭示了这样一个事实，执行到一个释放点的操作是一个高优先级的操作（参见§1.2.5的进阶阅读）。为了实现这个方案，所有相关的同步和锁定都必须依赖于这种优先级调整的工具类。

进一步说，仅在那些使用精确的优先级调度系统的特定JVM实现上，这种策略才能保证实现这种期望的特性。在实际情况中，任何支持精确的优先级调度的JVM都会对内建的锁和监视器操作使用优先级的调整。在其他情况下，无法保证使用精确优先级调度系统的正确性。

在实际情况中，完全依赖于精确优先级调度的程序将会牺牲很多可移植性。因为它们需要额外的、与JVM实现相关的保证，这些保证可能是通过使用额外的并发控制工具或构件来实现的。在其他更易于移植的程序中，那些“偏爱”高优先级线程的信号机类和其他相关的工具依然被用作提高程序响应的工具之一。

3.4.2 闭锁

闭锁（latch）变量或者闭锁条件是指那些一旦获得某个值就再也不会变化的变量或者条件。二元闭锁变量或者条件（通常就被称为闭锁，也叫一次性变量）的值只能改变一次，即从其初始化状态到其最终状态。

和闭锁相关的并发控制技术可以被封装在一个简单的Latch类中，并遵循通用的获取—释放的接口，但是它的语义为：一个release操作将使得所有之前或者之后的acquire操作恢复执行。

java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

闭锁可以帮助解决以下的初始化问题（参见 § 2.4.1）：直到所有的对象和线程都完全构造完成之后，某些操作才可以执行。例如，一个比 § 3.2.4 中更具有挑战性的游戏程序可能会需要保证所有的游戏者都必须等到游戏开始才能进行游戏。这可以通过类似下面的代码来实现：

```
class Player implements Runnable {                    // Code sketch
  // ...
  protected final Latch startSignal;

  Player(Latch l) { startSignal = l; }

  public void run() {
    try {
      startSignal.acquire();
      play();
    }
    catch(InterruptedException ie) { return; }
  }
  // ...
}

class Game {
  // ...
  void begin(int nplayers) {
    Latch startSignal = new Latch();

    for (int i = 0; i < nplayers; ++i)
      new Thread(new Player(startSignal)).start();

    startSignal.release();
  }
}
```

闭锁的扩展之一就是倒数计数器（countdown），其 acquire 操作在 release 操作执行了固定的次数，而不仅仅是一次后恢复执行。闭锁，倒数计数器以及其他建立在它们基础之上的简单工具类可以被用于处理以下这些条件的响应操作：

完成指示器。例如，强制某些线程直到某些操作执行完毕后才可以继续执行。

定时阈值。例如，在某个时期触发一组线程。

事件指示。例如，触发那些只有收到特定报文或者特定按钮被按下后才能继续的操作。

错误指示。例如，触发在全局性的关闭任务执行时才可以运行的一组线程。

3.4.2.1 闭锁变量与闭锁谓词

虽然闭锁工具类对大多数一次性触发应用来说都是很方便的，但在其他环境中，使用闭锁变量（也被称为永久变量）和闭锁谓词同样可以提高成员可靠性、易用性及效率。

闭锁谓词（包括常见的阈值指示器）的特性之一就是：它属于很少的几个可以（虽然很少用）用非同步忙等待循环（参见 § 3.2.6）来实现受保护方法的选择之一。如果已知一个谓词是闭锁，那么就不会有错误状态的危险（参见 § 3.2.4.1）。在检测这个谓词是否为真的操作和依赖于这个条件为真才可以执行的操作之间，这个谓词的值是不会改变的。例如：

java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

```
class LatchingThermometer {                    // Seldom useful
  private volatile boolean ready; // latching
  private volatile float temperature;

  public double getReading() {
    while (!ready)
      Thread.yield();
    return temperature;
  }

  void sense(float t) { // called from sensor
    temperature = t;
    ready = true;
  }
}
```

要注意的是，这种结构只有在下面的情况下才可以使用：类中所有相关的变量都被定义为 volatile 或者它们的读写都在同步下执行（参见 § 2.2.7）。

3.4.3 交换器

交换器可以用作同步通道（参见 § 3.4.1.4），不过它用一个 rendezvous 方法（有时也叫做 exchange）取代了 put 和 take 这两个方法，并且合并它们的效果（参见 § 2.3.4）。这个方法以某个线程提供给另一个线程的对象作为参数，并返回另一个线程所提供的对象。

交换器可以被扩展以支持两个以上的参与者使用，或者被进一步增强，以在参数上执行其他操作而非仅仅交换它们。util.concurrent 包中的 Rendezvous 类支持了这些功能。但是大部分程序的应用都仅限于在两个线程之间交换资源对象（通过使用 Rendezvous 默认的只有两个参与者的构造函数）。

扩展了 § 2.3.4 中所描述的机制的、基于交换器的协议可以用作资源池的替换方案（参见 § 3.4.1.2）。它可以被用于这种情况：不同线程中的两个或者两个以上的任务，每个任务在任何时间只能操作一个资源。当一个线程使用完一个资源而需要获得其他资源时，它就和其他的线程交换。这个协议最常见的应用就是缓冲区交换。在这种应用中，一个线程向一个缓冲区中填充数据（例如，通过读入数据），当这个缓冲区被填满时，这个线程就把它交换给处理缓冲区的线程，由这个线程来清空缓冲区。在这种方法中，只使用两个缓冲区且不需要拷贝，因此不再需要资源管理池了。

以下的这个 FillAndEmpty 类粗略地展示了在使用交换器时所需要的额外异常处理。由于这个协议是对称的，在试图交换的过程中，如果一个参与者被取消或者超时，则必然在对应的参与者中导致出现一个异常（在这个例子中是 BrokenBarrierException）。在下面这个例子中，当异常出现时，程序只是从 run 中返回。更实际一点的版本可能要执行相应的清除操作，包括执行额外的操作以处理在中断时尚未填满或者尚未清空的缓冲区，同时，还需处理 readByte 方法所带来的 IO 异常和文件结束条件。

```
class FillAndEmpty {                              // Incomplete
  static final int SIZE = 1024; // buffer size, for demo
  protected Rendezvous exchanger = new Rendezvous(2);
```


java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

```
    protected byte readByte() { /* ... */; }
    protected void useByte(byte b) { /* ... */ }

    public void start() {
      new Thread(new FillingLoop()).start();
      new Thread(new EmptyingLoop()).start();
    }

    class FillingLoop implements Runnable { // inner class
      public void run() {
        byte[] buffer = new byte[SIZE];
        int position = 0;

        try {
          for (;;) {

            if (position == SIZE) {
              buffer = (byte[])(exchanger.rendezvous(buffer));
              position = 0;
            }

            buffer[position++] = readByte();
          }
        }
        catch (BrokenBarrierException ex) {} // die
        catch (InterruptedException ie) {} // die
      }
    }

    class EmptyingLoop implements Runnable { // inner class
      public void run() {
        byte[] buffer = new byte[SIZE];
        int position = SIZE;  // force exchange first time through

        try {
          for (;;) {

            if (position == SIZE) {
              buffer = (byte[])(exchanger.rendezvous(buffer));
              position = 0;
            }

            useByte(buffer[position++]);
          }
        }
        catch (BrokenBarrierException ex) {} // die
        catch (InterruptedException ex) {} // die
      }
    }

  }
```

在这里通过对交换器的使用展示了工具类的设计优势，它将与对象成员变量相关的问题用与传递与消息相关的问题来替代。当需要大规模的协调机制时，这种设计比较易于处理（见第4章）。

java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

3.4.4 条件变量

在 Java 编程语言中，对于每个对象，监视器操作只和其一个等待集合相关。有些语言或者线程库（例如 POSIX 的 pthreads）支持通过一个公共对象或者锁来管理与多个条件变量（condition variable）相关的多个等待集合。

虽然任何需要多个等待集合的设计都可以通过类似于信号机的构件来实现，但通过创建工具类来模仿其他系统中的条件变量还是可行的。事实上，对 pthreads 风格的 condvars 的支持使得我们可以使用那些和并发的 C 或者 C++程序中的模式基本相同模式。

CondVar 类可以用于表示一个与给定的 Mutex 一同被管理的条件变量，这个 Mutex 被（并不一定）所有相关类中的独占锁使用。因此，使用 CondVar 的类必须依赖于 § 2.5.1 中所描述的“手动”锁定技巧。多个 CondVar 可以使用同一个 Mutex[6]。

下面这个类支持类似于标准的等待和通知的方法，其方法的命名基于它们在 pthreads 中类似方法的名字。

```
class CondVar {                              // Implementation omitted
  protected final Sync mutex;
  public CondVar(Sync lock) { mutex = lock; }

  public void await() throws InterruptedException;
  public boolean timedwait(long ms) throws InterruptedException;
  public void signal();                 // analog of notify
  public void broadcast();                // analog of notifyAll
}
```

（在 util.concurrent 包中的版本中，这些操作的细微差别也是源自 pthreads 的。例如，和 notify 不同，signal 方法并不需要线程先持有锁。）

这种类的主要应用通常不同于其原始的设计目的，它们往往被用于移植那些采用其他语言或系统的代码。另一方面，CondVar 同样可以被用于 § 3.3 是所描述设计模式中，并且也会遇到相同的设计问题。例如，以下是另一个有界缓冲区类。除了异常处理的结构以外，这个类看上去似乎就是 pthreads 编程书中的程序（参见 § 1.2.5 中的进阶阅读）。

```
class PThreadsStyleBuffer {
  private final Mutex mutex = new Mutex();
  private final CondVar notFull = new CondVar(mutex);
  private final CondVar notEmpty = new CondVar(mutex);
  private int count = 0;
  private int takePtr = 0;
  private int putPtr = 0;
  private final Object[] array;

  public PThreadsStyleBuffer(int capacity) {
    array = new Object[capacity];
  }
```

6 与之相反的，一个条件变量使用多个 Mutex 在逻辑上是可以的，但这种情况通常都反应了程序中的错误，因此在这个类中并不支持。

java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

```
    public void put(Object x) throws InterruptedException {
      mutex.acquire();
      try {
        while (count == array.length)
          notFull.await();

        array[putPtr] = x;
        putPtr = (putPtr + 1) % array.length;
        ++count;
        notEmpty.signal();
      }
      finally {
        mutex.release();
      }
    }

    public Object take() throws InterruptedException {
      Object x = null;
      mutex.acquire();
      try {
        while (count == 0)
          notEmpty.await();

        x = array[takePtr];
        array[takePtr] = null;
        takePtr = (takePtr + 1) % array.length;
        --count;
        notFull.signal();
      }
      finally {
        mutex.release();
      }
      return x;
    }
  }
```

3.4.5 进阶阅读

更多对信号机和条件变量讨论可以在很多并发编程的书中找到（参见§1.2.5）。

资源池可以被扩展为更通用的对象管理类。参见：

Sommerlad 和 Peter. “Manager”, in Robert Martin, Dirk Riehle, and Frank Buschmann (eds.), 《Pattern Languages of Program Design, Volume 3》, Addison-Wesley, 1998.

交换器的更多细节可见：

Sane, Aamod 和 Roy Campbell. “Resource Exchanger”, in John Vlissides, James Coplien, and Norman Kerth (eds.), 《Pattern Languages of Program Design, Volume 2》, Addison-Wesley, 1996.

经常使用的一些调度策略的近似公平性在下文中有所讨论：

Epema 及 Dick H.J. “Decay-Usage Scheduling in Multiprocessors”, 《ACM Transactions on Computer Systems》, Vol. 16, 367-415, 1998.

java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

3.5 协同操作

到目前为止，本章所讨论的内容还仅仅局限于那些依赖于对象状态的受保护操作。协同操作（Joint Action）框架提供了处理更通用的设计问题的方法。从高层设计角度来看，协同操作是这样一种原子（atomic）的受保护方法，它包含的条件与操作是和多个不同的且相互独立的参与对象相关的，它们可以被抽象地描述为一个包含两个或者两个以上对象的原子方法：

```
void jointAction(A a, B b) {                          // Pseudocode
  WHEN (canPerformAction(a, b))
    performAction(a, b);
}
```

这种类型的问题通常出现在分布式协议的开发、数据库，以及并发约束编程中。如 § 3.5.2 中所示，当不同且独立的对象中的不同且独立的操作需要相互协调时，即使那些看上去很普通的、依赖于委托的设计模式也需要这种处理方法。

除非你有某些特定目的的解决方案，处理协同操作的第一件事就是把含糊不清的目的和定义好的规范转换成可以实际编程的东西。具体需要考虑以下几点：

职责分配。哪个对象负责执行这个协同操作？其中的一个参与者？所有参与者？某个协调对象？

判断条件。如何去判断参与者何时处于可以执行操作的状态？通过调用它们的读取方法？当它们处在可执行状态时是否通知你？当它们可能处在可执行状态时都会通知你吗？

编写操作。多个对象中的操作如何组织？它们需要成为原子操作吗？如果它们中的某个或者多个操作失败了又如何处理？

连接条件和操作。如何确保操作只在条件满足时才会执行？假的警告能够接受吗？在检测条件和执行操作之间，需要禁止一个或者多个参与者改变状态吗？是仅当参与者进入了合适的状态时，还是只要收到条件已被满足的通知操作就会被执行？需要避免多个对象同时试图执行这个操作吗？

3.5.1 通用解决方案

不可能只通过很少的解决方案就能解决所有情况下的所有问题。被最广泛使用的通用解决方案是这样的：当参与者处于（或者可能处于）适合执行协同操作的状态时，它们彼此之间会相互通知，并且直到操作执行完，它们才会改变自己的状态。

这种设计提供了处理协同操作问题的高效解决方案。但是，其结构可能是易碎的且无法扩展的，并且可能会使参与者相互之间耦合得非常紧密。但当你可以通过构建参与类的子类来添加特定的通知或者操作，并且可以避免很多协同操作的设计中所存在的本质性的死锁问题，或者重中恢复时，这种方法就适于使用了。

这个设计的主要目标就是在被同步的代码中定义相关的通知和操作，这些通知和操作正确地嵌入了原有的方法中，这种方式很容易使人想起 double-dispatching 和 Visitor 模式（参

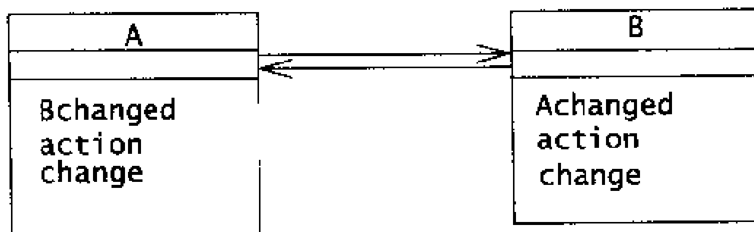
java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

见《Design Patterns》)。通常，好的解决方案都利用了参与者以及它们之间交互的特性。由于协同操作需要参与者之间直接耦合，并且，必须利用所有可用的约束以避免活跃性问题，所以，大多数协同操作都存在对上下文高度依赖的问题。这随即也导致了含有这些特殊用途代码的类必须被标记为 final。

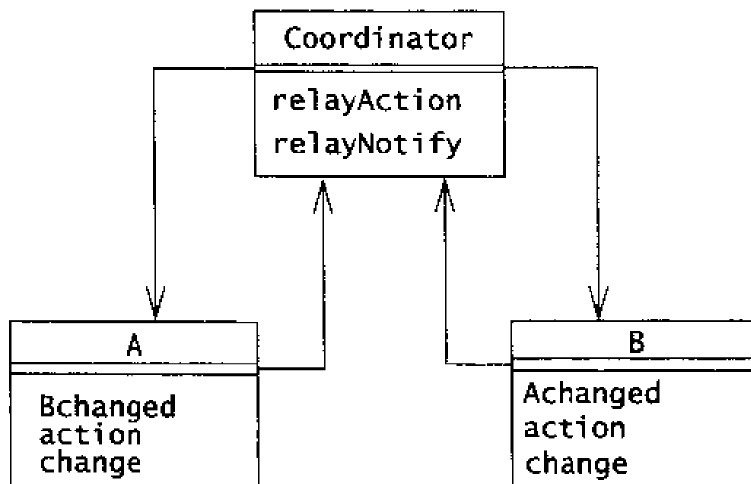
3.5.1.1 结构

为了更具体一些，以下的描述都是针对于两个参与者的情况的（即类 A 和类 B)，但也可以推广到多于两个参与者的情况中。在这里，任何一个参与者的状态变化都会引发对另一个参与者的通知。这些通知随即就会触发某一个或者全部参与者的相应协同操作。

可以采用的有两种典型的设计：平展版本（flat version)，即直接将参与者相互耦合在一起：



外部协调版本（explicitly coordinated version)，通过第三方对象（Mediator 的一种，见《Design Patterns》）中转某些或者全部的消息，这个对象也有可能参与相关的操作。尽管通过第三方对象来协调很少是必须的，但是使用它可以增加灵活性，并且它可以用于初始化对象和连接：



3.5.1.2 类和方法

在构建相应的类和方法时可以采用以下通用步骤：

- 定义 A 和 B 类（通常是子类)，他们都有指向对方的引用，同时也维护着其他值与引用，用以检测自己是否处于触发条件下并/或者执行相关操作。或者，通过第三方对象来间接地连接参与者。
- 编写一个或者多个方法来执行主操作。可以这样来实现：选择一个类包含主操作方法，该方法调用另一个参与者中的辅助方法。或者，这个主操作方法可以被定义在

java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

第三方协调类中，它随后再调用 A 和 B 中的辅助方法。

- 在每个类中，编写当另一个对象状态改变时被调用的 synchronized 方法。例如，在 A 类中，编写 Bchanged 方法，在 B 类中，编写 Achanged 方法。在每个方法中，也要包含检测自己是否处于正确状态的代码。如果最终的操作与这两个参与者都有关，那么在执行时就一定要保持着它们两者的同步锁。
- 在每个类中，当每个会触发相应操作的状态改变时，调用另一个对象的 changed 方法。如果必要，应确保导致通知的状态改变代码被正确地同步，以保证在所有的测试－执行序列被执行后，才释放在开始改变状态时所持有的所有参与者的锁。
- 确保在 A 和 B 的实例在被允许接受消息以相互交互之前，它们之间的连接和相应状态都已经被初始化了。这可以很容易地通过第三方的协调对象来实现。

通常，可以利用那些和应用环境相关的约束来简化或者合并这些步骤。例如，当通知和/或操作总是基于一个参与者的话，那么某些步骤就可以省略。类似的，如果改变的条件包含简单的闭锁断言（参见 § 3.4.2），那么就没有必要使用同步保证通知和操作的连续性。而且，如果你可以在协调者类中构建一个公用锁，并让类 A 和类 B 中的所有方法都使用该锁（参见 § 2.4.5），那么其他的同步就都可以去掉了，并且可以把这当成一个单对象并发控制问题，使用 § 3.2～ § 3.4 中的技巧来解决它。

3.5.1.3 活跃性

当所有的通知和操作都对称地分布在参与者之中，那么按上述步骤所设计的系统就很有可能进入死锁状态。一个由调用 Achanged 方法开始的操作序列可能会和由调用 Bchanged 方法开始的操作序列相互锁死。虽然没有通用的解决方案，但针对死锁的冲突解决策略（conflict-resolution strategie）包含以下几种方法。其中有些方法需要对程序进行大面积的改写和进行迭代式的优化。

强制定向性。例如，让所有的改变都通过其中一个参与者来执行。这可能只在你能够修改参与者接口的情况下可用。

先后顺序。例如，使用资源排序（参见 § 2.2.6）来避免顺序冲突的问题。

让步（Back-off）。例如，如果已经有其他更新任务正在执行，那么就忽略这个更新任务。如下面的例子所演示的那样，更新竞争往往都可以被简单地检测到并被安全地忽略。在其他情况下，可能需要使用支持超时的工具类来检测冲突，并且需要参与者在遇到失败时重试更新操作。

传递令牌。例如，仅当某个参与者获得某个资源时，才允许其执行操作，这可以通过所有权传递（ownership-transfer）协议来控制（参见 § 2.3.4）。

弱化语义。例如，如果不影响主要功能的话，可以适当放松原子性保证。

外部调度。例如，通过把活动按任务来进行管理和表示，参见 § 4.3.4。

3.5.1.4 范例

为了说明某些常用的技巧，考虑一下这个例子，当支票账户的余额低于某个值时，服务会自动将钱从储蓄账户转移到支票账户，但是这只能发生在储蓄账户没有透支的情况下。这

java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

个操作可以用以下由伪代码所编写的协同操作来表示：

```
void autoTransfer(BankAccount checking,       // Pseudocode
                  BankAccount savings,
                  long threshold,
                  long maxTransfer) {
  WHEN (checking.balance() < threshold &&
        savings.balance() >= 0) {
    long amount = savings.balance();
    if (amount > maxTransfer) amount = maxTransfer;
    savings.withdraw(amount);
    checking.deposit(amount);
  }
}
```

我们的解决方案要依赖以下这个简单的 BankAccount 类：

```
class BankAccount {
  protected long balance = 0;

  public synchronized long balance() {
    return balance;
  }

  public synchronized void deposit(long amount)
   throws InsufficientFunds {
    if (balance + amount < 0)
      throw new InsufficientFunds();
    else
      balance += amount;
  }

  public void withdraw(long amount) throws InsufficientFunds {
    deposit(-amount);
  }
}
```

以下的思考构成了一个解决方案：

- 没有必要去添加一个额外的协调者类。所需的交互操作可以被定义在 BankAccount 的子类中。
- 操作可以在支票账户的余额减少时或者储蓄账户余额增加时被执行。由于导致这两者改变的惟一操作就是 deposit（因为这个例子中的 withdraw 调用的也是 deposit），所以每个类中的 deposit 方法都会引发转账操作。
- 只有支票账户需要知道 threshold，且只有储蓄账户需要了解 maxTransfer 值（其他因素可能会导致略微不同的实现）。
- 在储蓄账户这方，条件检测和操作的代码可以被定义在一个 transferOut 方法中，如果不需要转账，那么其返回 0，否则，它会扣除相应的金额并返回转账的金额数。
- 在支票账户这方，一个 tryTransfer 方法可以处理支票账户和储蓄账户所引起的转账操作。

如果不加以特别的注意，所得的代码可能是非常容易死锁的。这是对称的协同操作中的

java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

一个本质性问题，因为每个参与对象的状态改变都可能导致相应操作被执行。在这个例子中，支票账户和储蓄账户可以同时开始执行它们的转帐操作。我们需要一个方法来避免出现由于双方都试图调用对方的方法而导致相互死锁的问题（注意，如果我们仅仅在支票账户的余额减少时才执行相应操作，那么死锁就不会发生。这样就可以得到一个简单得多的方案）。

为了演示，这里用一个简单的非定时让步（untimed back-off）协议解决了潜在的死锁问题，这是一种常用的解决方案（当然不可能解决所有的问题）。tryTransfer 方法使用了一个布尔工具类，其包含有一个 testAndSet 方法，这个方法首先将当前的值设为 true，并返回前一个值，这是一个原子操作（另一种方法是使用 Mutex 的 attempt 方法）。

```
class TSBoolean {
  private boolean value = false;

  // set to true; return old value
  public synchronized boolean testAndSet() {
    boolean oldValue = value;
    value = true;
    return oldValue;
  }

  public synchronized void clear() {
    value = false;
  }
}
```

这个类的实例用于在支票帐户这方的 tryTransfer 方法中控制其 synchronized 部分，这就是该设计中潜在的死锁点。当某个由储蓄账户发起的转账操作试图执行时，如果已经有一个转账操作正在执行（在这个例子中，该操作一定是由支票账户发起的），那么这个试图转账的操作就会被忽略，从而避免了死锁。由于 tryTransfer 和 transferOut 操作都是基于最新的存入余额值进行操作的，所以这种忽略是可以接受的。

以上的讨论使我们得到了下面的 BankAccount 的子类，它们只在以上给定的上下文中工作。每个类都需要在初始化的过程中建立相互的联接（未在类中显示）。

是否将这些类标记为 final 的确是非常难以定夺的。然而，在这些类中，子类的构建者在修改它们的方法和协议时，还是一定的自由度的，例如，改变 shouldTry 中的转账条件或 transferOut 中的转账金额。

```
class ATCheckingAccount extends BankAccount {
  protected ATSavingsAccount savings;
  protected long threshold;
  protected TSBoolean transferInProgress = new TSBoolean();

  public ATCheckingAccount(long t) { threshold = t; }

  // called only upon initialization
  synchronized void initSavings(ATSavingsAccount s) {
    savings = s;
  }

  protected boolean shouldTry() { return balance < threshold; }
```


java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

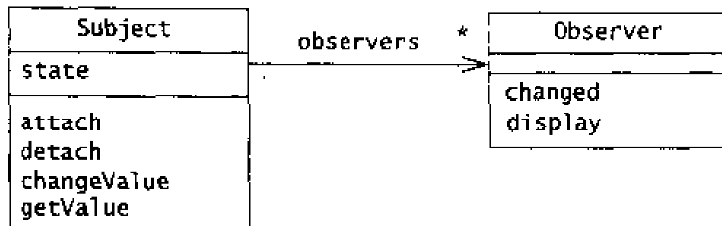
```
  void tryTransfer() { // called internally or from savings
    if (!transferInProgress.testAndSet()) { // if not busy ...
      try {
        synchronized(this) {
          if (shouldTry()) balance += savings.transferOut();
        }
      }
      finally { transferInProgress.clear(); }
    }
  }

  public synchronized void deposit(long amount)
   throws InsufficientFunds {
    if (balance + amount < 0)
       throw new InsufficientFunds();
    else {
      balance += amount;
      tryTransfer();
    }
  }
}

class ATSavingsAccount extends BankAccount {

  protected ATCheckingAccount checking;
  protected long maxTransfer;

  public ATSavingsAccount(long max) {
    maxTransfer = max;
  }

  // called only upon initialization
  synchronized void initChecking(ATCheckingAccount c) {
    checking = c;
  }

  synchronized long transferOut() { // called only from checking
    long amount = balance;
    if (amount > maxTransfer)
      amount = maxTransfer;
    if (amount >= 0)
      balance -= amount;
    return amount;
  }

  public synchronized void deposit(long amount)
   throws InsufficientFunds {
    if (balance + amount < 0)
      throw new InsufficientFunds();
    else {
      balance += amount;
      checking.tryTransfer();
    }
  }

}
```


java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

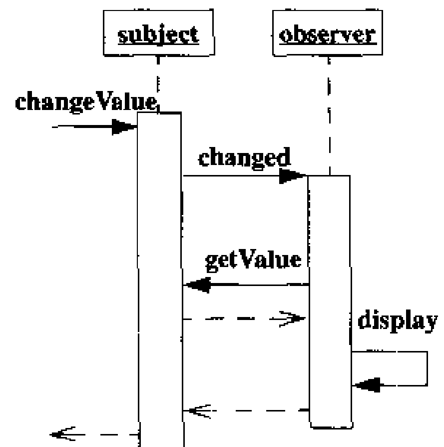
3.5.2 分离观察者（Decoupling Observer）

解决有关完全协同操作的设计和实现问题的最佳方法就是：不坚持要求跨越多个独立对象的操作为原子性操作。完全原子化很少是必要的，并可能会导致其他后续的设计问题，使类更难使用或者复用。

为了说明，我们来看一下《Design Patterns》中的 Observer 模式：

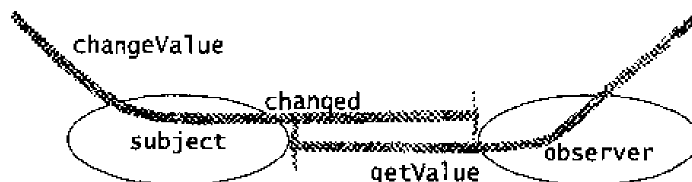


在 Observer 模式中，主题（Subject）（有时也被称作被观察者 Observable）表示了它所模拟的对象（比如温度）的状态，它具有获得和改变这个状态的方法。观察者（Observer）以某种方式显示或者使用主题所表示的状态（例如，描绘各种样式的温度计）。当一个主题的状态改变后，它仅仅通知其观察者，告诉它们自已的改变。而观察者负责通过回调方法来检测主题的变化并且决定是否执行相应的操作，例如，在屏幕上重绘主题。



可以在一些 GUI 框架、发布－预定系统，以及基于约束的程序中看到观察者模式。这个模式的一个版本定义在 java.util.Observable 和 java.util.Observer 中，但直到编写这本书时，它们还未被用在 AWT 和 Swing 中（参见 § 4.1.4）。

很容易错误地把观察者模式设计为同步的协同操作，而没有注意到这会导致潜在的活跃性问题。例如，如果每个类中的所有方法都被定义为 synchronized，并且 Observer.changed 可以在 Subject.changeValue 方法以外调用，则这些调用就可能相互死锁：



这个问题可以通过 § 3.5.1 中所讨论的技巧来解决。然而，最好的也是最容易的方法就是避免它。同步那些与改变通知相关的操作是没有必要的，除非你真的需要 Observer 操作和 Subject 的改变操作在同一个原子操作中。事实上，这种需求首先就否决了大部分使用观察者模式的原因。

作为替代，你可以应用 § 1.1.1.1 中的默认规则，并且，当从 Subject 中调用 Object 方法

java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

时释放掉不需要的锁，用以解除它们之间的耦合。不过，这使得 Observer 在得到状态改变的通知之前，可以多次改变 Subject 的状态，同时，Observer 在调用 getValue 时可能不会有任何变化。通常，这种语义弱化是完全可以接受的，甚至是所期望的。

这里有一个简单的实现，Subject 仅使用一个双精度数来代表其模拟的状态。它使用 § 2.4.4 中的 CopyOnWriteArrayList 类来维护 Observer 的列表。这就避免了在遍历观察者时的锁定操作，同时也满足了设计的目的。为了简化示例，这里的 Observer 被定义为一个具体类（而不是一个可以有多个实现的接口），且只能处理一个 Subject。

```
class Subject {

  protected double val = 0.0;        // modeled state
  protected final CopyOnWriteArrayList observers =
                                  new CopyOnWriteArrayList();

  public synchronized double getValue() { return val; }
  protected synchronized void setValue(double d) { val = d; }

  public void attach(Observer o) { observers.add(o); }
  public void detach(Observer o) { observers.remove(o); }

  public void changeValue(double newstate) {
    setValue(newstate);
    for (Iterator it = observers.iterator(); it.hasNext();)
      ((Observer)(it.next())).changed(this);
  }

}

class Observer {

  protected double cachedState;       // last known state
  protected final Subject subj;       // only one allowed here

  public Observer(Subject s) {
    subj = s;
    cachedState = s.getValue();
    display();
  }

  public synchronized void changed(Subject s){
    if (s != subj) return;      // only one subject

    double oldState = cachedState;
    cachedState = subj.getValue(); // probe
    if (oldState != cachedState)
      display();
  }

  protected void display() {
    // somehow display subject state; for example just:
    System.out.println(cachedState);
  }

}
```


java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

3.5.3 进阶阅读

在 DisCo 建模与规范语言中，协同操作用作为统一的表现多方操作的框架：

Jarvinen, Hannu-Matti, Reino Kurki-Suonio, Markku Sakkinnen 和 Kari Systa. "Object-Oriented Specification of Reactive Systems", 《Proceedings, 1990 International Conference on Software Engineering》, IEEE. 1990.

在交互过程中，它们在一个略微不同的上下文中被进一步研究，同时也提出了应用于协同操作设计中的几种不同的公平性问题。例如，为避免恶意的参与者占据资源而排挤其他的参与者而做的相应的设计策略。见：

Francez, Nissim 和 Ira Forman.《Interacting Processes》, ACM Press, 1996.

正对对象和操作过程中任务协调的更广泛的调研，可见：

Malone, Thomas 和 Kevin Crowston. "The Interdisciplinary Study of Coordination", 《ACM Computing Surveys》, March 1994.

协同操作框架可以用作实现分布式协议的内部机制的基础。有关并发和分布式对象的协议的前瞻性表述和分析可见：

Rosenschein, Jeffrey 和 Gilad Zlotkin.《Rules of Encounter: Designing Conventions for Automated Negotiation Among Computers》, MIT Press, 1994.

Fagin, Ronald, Joseph Halpern. Yoram Moses 和 Moshe Vardi.《Reasoning about Knowledge》, MIT Press, 1995.

有关处理参与者失败操作的协同操作框架可参见以下的描述：

Stround, Robert 和 Avelino Zorzo. "A Distributed Object-Oriented Framework for Depend able Multiparty Interactions", 《Proceedings of OOPSLA》, ACM, 1999.

3.6 事务处理

在并发 OO 编程中，事务是指某客户对象在任意一组参与对象上，通过调用一系列方法执行的某个操作，它不会受其他外界干扰。

参与者和操作序列的任意性使得我们需要扩展§3.5 所讨论的协同操作的控制策略。事务处理技巧将基于委托的同步与控制延伸到这样的环境中：每个参与者都可能不清楚其操作中的原子约束，且无法依赖更有效率的架构方案。在事务处理框架中，每个参与者（以及每个客户对象）都放弃了其自主决定如何执行并发控制的权利。取而代之的是，参与者必须在如何以及何时执行操作和/或提交操作的结果上达成一致。

通常，在实现某些有价值的通用功能的组件时，也会对程序员提出更多的要求，事务处理框架就是很好的一个例子。支持事务处理协议的类是非常有用并具有极高重用性的。事务处理框架可以被用于处理书中讨论的大部分并发问题。但是，它们依赖于这样的设计：每个类，在每个功能层面，都要支持标准化的事务处理协议，以将控制在连续的层中传递。由于事务处理框架过于笨重，当你需要保证构建在某些对象上的代码序列的原子性时，事务处理

java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

框架往往是不合适的。

例如，如果已知在一个组件或者应用程序中的所有调用序列，你可能就可以不使用事务控制。在这种情况下，你可以对每个调用都使用其专用的设计（使用任何可以使用的技巧）而无需考虑通用性的问题。这可能是在可重用的同步对象中包含常用方法的原子版本的（参见§2.2.2）极端方式。从使用最简单且最安全的方法这个角度来说，有时，这种方案似乎是可用。类似的，当客户程序知道如何获得执行某个操作所需要的所有的锁，且知道如何避免潜在死锁时，则可以完全依赖客户端锁定（参见§2.2.3）。

这一节对并发编程中通用的事务处理技巧进行了概要地描述。这里所列举的设计仅仅处理内部并发问题，并不包括数据库或者分布处理中的问题。因为，即使轻量级（至少相对来说）的内部事务处理框架通常也会与应用相关的约束和特性联系在一起，你很少会直接使用这里所介绍的接口和类（虽然它们大多数都是 net.jini 中的简化版本）。如果你使用标准的事务处理框架，例如 JDBC 或者 JTS，那么就有可能碰到与持久性支持或相关服务有关的问题，而这些问题不在本书讨论的范围之内。不过，这一节最后的例子（§3.6.4）向我们展示了事务处理采用的想法是如何帮助构建更通用的并发 OO 设计的。因此，这一节的主要目的就是简要地概括事务处理系统是如何扩展并发控制的方法的，并提出了几种可以按需要进行调整以用于其他并发问题的处理技巧。

紧接上一个例子，重新考虑如何实现§3.5.1 中的 BankAccount 类的 transfer 方法。从事务处理的角度看，一个独立的转账操作（没有任何自动转账的措施）如下所示：

```
pseudoclass AccountUser {                              // Pseudocode

  TransactionLogger log; // any kind of logging facility
  // ...

  // Attempt transfer; return true if successful
  public boolean transfer(long amount,
                          BankAccount source,
                          BankAccount destination) {
    TRANSACTIONALLY {
      if (source.balance() >= amount) {
        log.logTransfer(amount, source, destination);
        source.withdraw(amount);
        destination.deposit(amount);
        return true;
      }
      else
        return false;
    }
  }
}
```

伪修饰词 TRANSACTIONALLY 表明，我们希望这段代码要么一次完全执行完毕，要么就完全不执行，且不会和其他操作有任何的冲突。一旦此点被实现后，这个操作就可以被用于§3.5.1 中所描述的自动转帐方案中。此外，这种事务处理方法还允许比我们例子中更加灵活的方案，但是也需要更多的开销。一旦这些类被赋予事务处理的功能，它们就可以使任

java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

何和银行帐户相关的操作序列具有事务性。

3.6.1 事务处理协议

事务处理框架依赖于大多数并发控制策略都使用的 before/after 策略的一种扩展。在事务处理中，before 操作通常被称为参与（有时也被称为开始），after 操作常被称为提交。参与/提交操作与获得/释放锁操作之间最主要的区别在于：参与/提交需要所有参与对象达成一致，所有参与者都必须同意开始或结束事务。这就形成了参与和/或提交的两阶段协议——首先达成一致，然后执行。如果任何参与者不同意参与或者提交，那么这个试图执行的事务就要被中止。这个协议的最直接版本如下所示：

1. 对于每个参与者 p，如果 p 不能参与，那么中止。
2. 对于每个参与者 p，试验性地执行 p 的操作。
3. 对于每个参与者 p，如果 p 无法提交，则中止。
4. 对于每个参与者 p，提交 p 在这个事务中所产生的结果。

在大多数并发控制的上下文中，有两个补充策略可以应用于这个协议。在最纯乐观事务处理（optimistic transaction）中，参与者总是可以参与，但是并不是总可以提交。在最保守的事务处理（conservative transaction）中，参与者并不是总可以参与，而一旦它们参与了，就一定能够提交。当竞争的可能性低到足以忽略回滚所带来的开销时，乐观法是最适用的。当很难或者无法撤销事务中所执行的操作时，保守法就是最适用的。然而，很少有这种纯粹的情况，并且构建允许这两种情况相互混合的框架也并不太难。

只有在执行任何操作之前，所有参与者的身份都已知的情况下，这种最经典的保守事务处理才能实现。这种条件并不是总能达到的。在 OO 系统中，参与者对象是指这样的对象：它们的方法会在事务的调用序列中被调用。由于多态、动态加载等其他机制，我们通常无法预先识别它们；往往只有在操作被执行时，才可以知道它们的身份。尽管如此，在很多情况下，至少有一些参与者是可以被预知的，并且可以在它们参与任何无法恢复的操作之前对它们进行侦测。

然而，在大多数实现保守的 OO 事务处理的方法中，参与者只是试验性地参与。它们可以在以后拒绝提交，使用同乐观事务处理中几乎完全相同的策略。反过来说，在乐观法中，完全回滚也不一定是必要的。如果不影响整体的功能的话，某些前滚操作也是允许的。

3.6.2 事务参与者

除了要有支持参与、提交、中止，以及（如果必要的话）创建事务的方法，对于每个在结构化事务处理框架中的类，其每个公共方法中，除了正常的参数外，都还需要添加一个事务控制参数。

当调用方法时，所提供的事务参数是用于要求这个方法在给定的事务中执行，并且只在被要求提交操作结果时才能提交。这些方法都采用以下这种形式：

```
ReturnType aMethod(Transaction t, ArgType args) throws...
```


java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

例如，BankAccount.deposit 可以被这样定义：

```
void deposit(Transaction t, long amount) throws ...
```

Transaction 用于提供必要的控制信息，可以是任何类型的类。该事务信息必须被传递到参与事务的每个方法中，包括那些嵌入的对辅助对象的调用。事务参数的最简单形式就是惟一标识每个事务的事务键（key）。随后，每个参与对象负责在所有方法中按照相应的事务策略使用这个键去管理和隔离操作。此外，事务参数也可以是某个特定的控制或协调对象的引用，这个对象拥有帮助参与者扮演它们在事务中角色的方法。

然而，在这里还是可以舞弊的，并且很多事务处理框架都这么做了。例如，事务标识符可以被隐藏为线程相关的数据（参见 § 2.3.2）。before/after 操作可以应用在执行组件所提供服务的会话的截获入口处。可以通过反射或者扫描字节码来测定参与者。并且，回滚可以通过把整个组件的状态序列化并/或在进入服务会话时获取锁来半自动实现。这种技巧可以用于对应用程序的开发者隐藏事务控制的细节，但这同时也造成了在一般的执行内部并发控制的轻量级事务处理框架所不值得的消耗和使用限制。

3.6.2.1 接口

参与者类必须实现一些接口，这些接口它们定义了并发控制中所需要的方法。以下是一个简单但很有代表性的接口：

```
class Failure extends Exception {}

interface Transactor {

  // Enter a new transaction and return true, if possible
  public boolean join(Transaction t);

  // Return true if this transaction can be committed
  public boolean canCommit(Transaction t);

  // Update state to reflect current transaction
  public void commit(Transaction t) throws Failure;

  // Roll back state (No exception; ignore if inapplicable)
  public void abort(Transaction t);

}
```

在很多变化中都可以把参与阶段的操作按照提交阶段那样进行拆分——一个预备的 canJoin 方法，以及一个必需的 join 方法。这里的 canCommit 方法通常在事务处理框架中被命名为 prepare。

为了简化起见，这些操作只使用了一个 Failure 异常，这一系列的例子都这样处理。参与对象在遭遇事实的或者潜在的冲突时，或者它们被要求参与未知的事务时，可以抛出异常。当然，在实际使用中，你可能需要子类化这些异常以在调用失败时为客户提供额外的信息。

第二个接口或者类是用于描述 Transaction 自身的。在讨论简单的操作时，我们可以使用以下这个无操作版本：

java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

```
class Transaction {
  // add anything you want here
}
```

同样，事务不一定要和一个对象相关。一个简单的惟一 long transactionKey 参数就可以满足所有对 Transaction 的需要了。从另一个极端来说，你可能需要 TransactionFactory 类创建所有的 Transaction。这可以使得不同类型的事务处理使用不同的 Transaction 对象。

3.6.2.2　实现

事务中的参与者必须同时支持事务参与者接口和描述其基本操作的接口。例如：

```
interface TransBankAccount extends Transactor {

  public long balance(Transaction t) throws Failure;

  public void deposit(Transaction t, long amount)
   throws InsufficientFunds, Failure;

  public void withdraw(Transaction t, long amount)
   throws InsufficientFunds, Failure;

}
```

不过，在纯粹的存取方法上添加事务签名的确不是必要的，例如这里的 balance 方法。取而代之的是（或者除了使用事务的版本外），当不参与事务的客户程序调用这些方法时，它们可以返回最近一次提交的内容。或者，一个空事务类型（或者就传入 null 作为 Transaction 参数）可以被用于指示这是一次对这个事务方法的快捷调用。

对于实现事务性类的最常用方法来说，首先需要分离出底层的状态对象，使用临时操作法或者检查点方法（参见 §2.4.4 和 §3.1.1.3）将其构建为独立的辅助类。这有助于简化执行那些虚拟状态变化的操作，这些变化仅在 commit 操作之后才更新，并/或在 abort 操作时被恢复。这种方法尤其适用于那些支持持久性的事务框架，因为它们通常都需要分离出表示层，以使从磁盘或者其他媒体读出与写入更有效率。

虽然目前这种方法是最好驾驭的，但有时它也会导致不如意的编码风格，即对象必须假设自己正处在由某个事务对象来维护的状态。仅当其状态和给定的事务相关联时，普通的公共方法才可以执行相应的操作，并调用类似对象的方法。

事务性方法（包括控制方法和其基本操作）的实现可以使用从乐观到保守的各种方法。下面这张表格概略地描述了那些使用 tx 这个 Transaction 参数的方法的主要步骤。

方　法	乐　观　法	保　守　法
join	• 拷贝当前的状态，并将其和 tx 关联（例如使用一个散列表来保存），同时复制该状态初始时的一些值。 • 返回 true	• 当已经处于某个与当前事务相冲突的事务中时，返回 false，也可以先尝试使用定时等待法来等待当前事务完成。 • 询问所有参与操作的对象是否可以参与；如果任何一个无法参与，返回 false。

java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

续表

方 法	乐 观 法	保 守 法
join		• 对当前状态作一个备份拷贝，用以在取消时恢复。 • 记录 tx 为当前事务；返回 true
action methods	• 如果 tx 不是已知的事务，那么先尝试参与 tx，如果参与失败，那么这个操作失败。 • 在当前的状态上执行相应的基本操作，并/或使用 tx 作为参数，调用其他对象上的其他方法，记录所有这些对象的标识符。 • 如果出现任何失败操作，将 tx 标记为不可以提交。	• 如果 tx 不是当前的事务，那么操作失败。 • 在当前的状态上执行相应的基本操作，并/或使用 tx 作为参数，调用其他对象上的其他方法。 • 一旦失败，标记当前 tx 为不可提交。
abort	• 丢弃任何与 tx 相关的记录。 • 向所有参与者传播这个操作。	• 如果 tx 是当前事务，用备份的内容恢复状态并标明无当前事务。 • 向所有参与者传播这个操作。
commit	• 将 tx 相关的状态保存为当前状态。 • 向所有参与者传播这个操作。	• 抛弃说有的备份；标明无当前事务。 • 向所有参与者传播这个操作
canCommit	• 如果自参与了 tx 后，发生了任何有冲突的提交，或者其他有冲突的事务已经约定要提交时，返回 false。 • 询问所有与当前操作相关的所有对象是否可以提交，如果任何一个对象无法提交，返回 false。 • 表明 tx 已约定要提交；返回 true。	• 询问其他参与者，如果任何一个无法提交，那么返回 false。 • 如果操作中没有发生局部的失败，就返回 true。

当把这些应用在 BankAccount 类上时，由于在每一步上使用最简单的方法，以至于最后所得的使用混合策略的版本基本上是无法真正商用的。为了简化，它只维护了一份状态拷贝（作为类的一个成员变量），因此，这个类只可以用于非相互重叠的事务处理中。不过，对于演示事务处理类的通用结构这个目的来说，这个版本还是足够了，并且它也暗示了如果要实现一个更有用的版本需要多编写多少代码。

```
class SimpleTransBankAccount implements TransBankAccount {

  protected long balance = 0;
  protected long workingBalance = 0; // single shadow copy
  protected Transaction currentTx = null; // single transaction

  public synchronized long balance(Transaction t) throws Failure {
    if (t != currentTx) throw new Failure();
    return workingBalance;
  }

  public synchronized void deposit(Transaction t, long amount)
   throws InsufficientFunds, Failure {
    if (t != currentTx) throw new Failure();
    if (workingBalance < -amount)
      throw new InsufficientFunds();
    workingBalance += amount;
  }
```


java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

```
public synchronized void withdraw(Transaction t, long amount)
 throws InsufficientFunds, Failure {
  deposit(t, -amount);
}

public synchronized boolean join(Transaction t) {
  if (currentTx != null) return false;
  currentTx = t;
  workingBalance = balance;
  return true;
}

public synchronized boolean canCommit(Transaction t) {
  return (t == currentTx);
}

public synchronized void abort(Transaction t) {
  if (t == currentTx)
    currentTx = null;
}

public synchronized void commit(Transaction t) throws Failure{
  if (t != currentTx) throw new Failure();
  balance = workingBalance;
  currentTx = null;
}

}
```

实现了 Transactor 接口的类也可以在参与者中共享引用。例如，你可以构建一个代理账户，这个账户会将消息发给其他不相关或者无法控制的账户。

```
class ProxyAccount implements TransBankAccount {
  private TransBankAccount delegate;

  public boolean join(Transaction t) {
    return delegate.join(t);
  }


  public long balance(Transaction t) throws Failure {
    return delegate.balance(t);
  }

  // and so on...
}
```

3.6.3 创建事务

使用那些实现了 Transactor 接口的参与者的事务都采用了标准的形式，并按下列步骤执行：

- 创建一个新的 Transaction。
- 调用所有（初始时已知的）参与者的 join 方法，如果有无法参与的参与者，那么立即失败。
- 尝试执行整个操作，如果其中有任何操作失败了，就中止所有的参与者，并回滚所

java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

有从属操作。

- 当操作完成时，执行所有的 canCommit 操作，再根据返回的结果执行 commit 或者 abort。

在大多数的应用中，如果初始化事务的类也实现了 Transactor 接口，就可以简化不少。它们也可以支持其他的方法以执行构建日志和相关记录的操作。

这个协议的很多部分都是可以自动化完成的，重新分配或者集中功能，以及添加其他特征也是可行的。例如，可以花费更多的努力去计算一个事务是否可以参与和/或提交，以最大限度地减少中止所带来的开销。可以分析（例如，可以使用冲突集合，参见 § 3.3.2）潜在的冲突事务的操作和参与者结构以进一步处理，当没有可能的冲突时，就允许操作的重叠。类似的，锁定策略也可以通过使用读写锁来优化，甚至于优化以支持锁定升级和有目的的锁（参见 § 3.3.3.1）。

3.6.3.1 范例

本节的这个版本 transfer 操作处理了下列几种潜在的失败：

语义失败。如果账户中可能没有足够的资金，那么该方法就会返回 false。在这个例子中，甚至没有对操作源是否有足够的余额进行检查。甚至于，当它返回真时，试图执行 withdraw 的操作还是会失败。类似的，既然 amount 允许为负值，当 source.withdraw 成功时，destination.deposit 也可能会失败（对于一个负金额，存款操作就等同于取款操作，反之亦然）。在这里还可以通过捕捉其他异常来处理这些方法中所遭遇的其他错误。

冲突。当由于和当前某个事务冲突，账户无法参与或者提交当前事务时，便抛出一个异常以表示这个操作是可以重试的。

事务错误。如果参与对象在表示其可以提交后未能完成提交操作，那么它所造成的后果就是无法恢复且灾难性的。当然，这些对象的方法应该尽其所能地去避免提交失败，因为这种内部错误一旦发生就无药可救了。在这个例子中，我们把异常重新转发给客户程序。在更实际的版本中，这个失败可能会触发一个恢复操作，该操作会将上一次保存在持久性记录中的对象取出用于恢复操作。

在这个例子中，这些情况下的恢复操作碰巧都相同（并被放入同一个辅助方法中）。abort 语句执行状态回滚操作，日志必须被独立回滚。

```
class FailedTransferException extends Exception {}
class RetryableTransferException extends Exception {}

class AccountUser {
  TransactionLogger log;                  // a made-up class

  // helper method called on any failure
  void rollback(Transaction t, long amount,
                TransBankAccount src, TransBankAccount dst) {
    log.cancelLogEntry(t, amount, src, dst);
    src.abort(t);
    dst.abort(t);
  }
```


java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

```
public boolean transfer(long amount,
                        TransBankAccount src,
                        TransBankAccount dst)
  throws FailedTransferException, RetryableTransferException {

    if (src == null || dst == null)       // screen arguments
      throw new IllegalArgumentException();
    if (src == dst) return true;          // avoid aliasing

    Transaction t = new Transaction();
    log.logTransfer(t, amount, src, dst); // record

    if (!src.join(t) || !dst.join(t)) {   // cannot join
      rollback(t, amount, src, dst);
      throw new RetryableTransferException();
    }

    try {
      src.withdraw(t, amount);
      dst.deposit(t, amount);
    }
    catch (InsufficientFunds ex) {        // semantic failure
      rollback(t, amount, src, dst);
      return false;
    }
    catch (Failure k) {                   // transaction error
      rollback(t, amount, src, dst);
      throw new RetryableTransferException();
    }

    if (!src.canCommit(t) || !dst.canCommit(t)) { // interference
      rollback(t, amount, src, dst);
      throw new RetryableTransferException();
    }

    try {
      src.commit(t);
      dst.commit(t);
      log.logCompletedTransfer(t, amount, src, dst);
      return true;
    }
    catch(Failure k) {                    // commitment failure
      rollback(t, amount, src, dst);
      throw new FailedTransferException();
    }

  }
}
```

3.6.4 可否决（vetoable）改变

事务处理框架可能会变得异常复杂，这个事实常常使开发人员忽视了那些简单有效的小规模并发设计中的事务解决方案。我们将用一个更常见的、可以很容易地由事务来解决的设计问题来总结这一节。

java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

在 JavaBean 框架中，组件对象会有一系列的属性（propertie）——即支持 get 和 set 方法的成员变量。受约束（constrained）属性可以支持可否决的 set 方法。一个组件可以拥有一组事件监听器，当一个可否决的 set 方法执行时，就会向这些监听器发送可否决的改变事件。只要有一个监听器对这个事件发出 PropertyVetoException，那么试图改变的属性操作就必须被取消。

很多组件都用这种可否决类型的属性改变来实现相应的操作。例如，试图退出一个编辑器程序的操作可以被实现为一个 set 方法，这个方法可以被任何尚未保存的文档和确认对话框所否决。

可否决的改变使用的是一种简化版本的事务协议，该协议中只有一个活跃的参与者，但是可能会有多个被动的参与者在等待投票以决定操作的结果。这种投票的操作既可以用保守形式（比如，在执行更新操作之前），也可以用乐观形式（比如，在试验性的执行了更新操作以后）来实现。

对于构建类似的解决方案，以下是一些 java.beans 包支持相关的背景知识：

- 监听器的结构基本上与 § 3.5.2 中所介绍的观察者相同，惟一的不同点在于它们是由包含了改变信息的事件所触发。在这里讨论的一般例子中，当事件发生时，我们是直接调用每个监听器的 vetoableChange(PropertyChangeEvent evt)方法，而不是像 § 4.1 中介绍的那样，将调用放入队列中。
- java.beans 包中的 VetoableChangeSupport 和 PropertyChangeSupport 类可以被用于向所有的监听器广播事件。不过，通常我们都使用支持无锁多播（lock-free multicasting）的复制写（copy-on-write）版本。下面的例子使用了 util.concurrent 包中的 VetableChangeMulticaster 和 PropertyChangeMulticaster 类，它们的接口和 java.beans 包中对应的类的接口相同。它们提供了 § 2.4.4 中所描述的添加监听器和移除监听器的方法。
- VetoableChangeMulticaster.fireVetoableChange 方法构建并广播 PropertyChangeEvent 事件，这些事件对象的成员变量含有属性名称、旧值和计划更新值。

作为一个演示那些基本技巧的比较简单的例子，现在来看一下这个含有一个可否决的 color 属性的 ColoredThing 组件。每一个 ColoredThing 都可以拥有多个否决者，同时也可以有一些普通的监听器，它们在每次属性更新时被通知。我们将使用一个简单的保守型方案。

当 ColoredThing 接收到 setColor(Color newColor)的请求后，便执行以下操作步骤：

1. 检查当前是否有其他的 setColor 操作正在执行，如果是的话，就抛出一个 PropertyVetoException 异常。为此，这个类维护了一个描述当前是否有正在进行的改变的布尔型执行状态变量。另一个更好（但也许不一定受欢迎）的版本是基于 changePending 使用 wait/notifyAll 构件以等待其他事务结束。

2. 检查调用参数是否为空，如果是的话，则拒绝执行这个改变。这也在某种意义上演示了组件如何否决自己的改变。

3. 调用 fireVetoableChange 方法，它将这个改变事件广播给所有的监听器。

4. 如果这个改变事件导致了某个 PropertyVetoException 异常，那么就中止当前操作并重新抛出这个异常。否则，更新 color 这个成员变量，重置事务进行的标志，并向所有的属性

java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

监听器发送改变事件。作为额外的保护手段，这个方法维护了一个 completed 变量，它是用于侦测运行时异常的。如果遇到这种运行时异常，finally 代码段会确保 changePending 这个标志被正确重置。

```
class ColoredThing {

   protected Color myColor = Color.red; // the sample property
   protected boolean changePending;

   // vetoable listeners:
   protected final VetoableChangeMulticaster vetoers =
     new VetoableChangeMulticaster(this);

   // also some ordinary listeners:
   protected final PropertyChangeMulticaster listeners =
     new PropertyChangeMulticaster(this);

   // registration methods, including:
   void addVetoer(VetoableChangeListener l) {
     vetoers.addVetoableChangeListener(l);
   }

   public synchronized Color getColor() { // property accessor
     return myColor;
   }

   // internal helper methods
   protected synchronized void commitColor(Color newColor) {
     myColor = newColor;
     changePending = false;
   }

   protected synchronized void abortSetColor() {
     changePending = false;
   }

   public void setColor(Color newColor)
    throws PropertyVetoException {
     Color oldColor = null;
     boolean completed = false;

     synchronized (this) {

       if (changePending) { // allow only one transaction at a time
         throw new PropertyVetoException(
           "Concurrent modification", null);
       }
       else if (newColor == null) {    // Argument screening
         throw new PropertyVetoException(
           "Cannot change color to Null", null);
       }
       else {
         changePending = true;
         oldColor = myColor;
       }
     }
```


java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

```
        try {
          vetoers.fireVetoableChange("color", oldColor, newColor);
          // fall through if no exception:
          commitColor(newColor);
          completed = true;
          // notify other listeners that change is committed
          listeners.firePropertyChange("color", oldColor, newColor);
        }
        catch(PropertyVetoException ex) { // abort on veto
          abortSetColor();
          completed = true;
          throw ex;
        }
        finally {                        // trap any unchecked exception
          if (!completed) abortSetColor();
        }
      }
    }
```

3.6.5 进阶阅读

对数据库系统中的事务处理更加详细的讨论可以在以下文献中找到：

Bacon 和 Jean. 《Concurrent Systems》，Addison-Wesley, 1993.

Cellary, Wojciech, E. Gelenbe 和 Tadeusz Morzy.《Concurrency Control in Distributed Database Systems》，North-Holland, 1988.

Gray, Jim 和 Andreas Reuter. 《Transaction Processing: Concepts and Techniques》，Morgan Kaufmann, 1993.

Khoshafian 和 Setrag. 《Object-Oriented Databases》，Wiley, 1993

Lynch, Nancy, Michael Merritt, William Weihl 和 Alan Fekete. 《Atomic Transactions》，Morgan Kaufmann, 1994.

使用 JDBC 进行数据库编程可见：

White, Seth, Maydene Fisher, Rick Cattell, Graham Hamilton 和 Mark Hapner. 《JDBC™ API Tutorial and Reference, Second Edition》，Addison-Wesley, 1999.

3.7 工具类的实现

工具类和工具方法可以将高效、可信赖、通用的并发控制构件的实现封装起来，几乎可以使它们像语言中的一部分那样被使用。这些类可以捕获那些灵巧的、复杂的、易于出错的构件又高效的利用特殊情况，并将最后结果打包以使使用它们的程序可以更容易编写、更可靠且常常性能更佳。当实际的设计证明是必要的时候，为实现这些类一次性地花费一些开发精力是值得的。

这一节描述了一些用于构建通用工具类的技巧。这些技巧都依赖于本书前面所讨论的设计和实现的策略，但是它们还包含有一些仅在构建支持类时才会出现的特殊结构。

这节首先描述了如何在普通接口上添加获得一释放（acquire-release）协议。随后，通过

java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

一个例子演示了如何使用协同操作的技巧来将类分割以获得必要的并发控制，并在随后将其合并以提高性能。最后，讨论了如何将等待的线程隔离以管理通知。

3.7.1 获得－释放协议

如 § 2.5.1 和 § 3.4.1 中所讨论的那样，很多并发控制构件都遵守获得－释放协议，这个协议可以由以下这个简单接口来实现：

```
interface Sync {
  void acquire() throws InterruptedException;
  void release();
  boolean attempt(long msec) throws InterruptedException;
}
```

如果要以某种给定语义（例如，锁、信号机、闭锁）来实现这个接口，那么就需要由 Sync 对象来管理那些会导致等待和通知的内部状态，而不是由使用它们的类来管理。此外，所有的控制都必须被放在对外的方法之内；不能将这些控制方法散播到客户类中，而且，不应该让客户程序必须通过使用一些要特别对待，非标准的方法才能达到期望的效果。

通过实现 § 3.4.1 所介绍的 Semaphore 类，可以遇到大多数实现中都会的问题。其他基于 Sync 的实现也同其类似（事实上，如 § 3.4.1 所示，类似于 Mutex 的类可以通过使用信号机来实现）。

无论从概念角度还是从其表现角度，信号机都维护着一个计数器，这个计数器的值是它所管理的许可证数量。基本的概念是，acquire 操作一直要等待到（如果有必要）至少有一个许可证时才可执行，并且，每次 release 操作都应该增加许可证的数量并通知所有处于等待中的线程。以下是在实现这个类之前所做的分析：

- 由于所有等待中的线程都在等待许可证，而每个 release 操作都会添加一个许可证，所以我们可以使用 notify 来取代 notifyAll，这会减少通知所造成的开销。此外，还可以使用 § 3.2.4.2 中所描述的中断时发出额外通知（extra-notify-on-interrupt）的方法来避免当线程在不恰当的时间中断时所造成的错误。
- 由于这个类需要被编写得尽量通用，所以，为了安全起见，计数器使用了 long 来计数。这可避免所有实际使用中可能会遇到的溢出问题，而相对于监视器的消耗来说，由于使用 long 所造成的开销几乎可以忽略不计。
- 为了保证程序的响应率，必须检测以确保当前线程在获得任何锁之前没有被中断。这样能将那些应该自我取消，却一直在等待获得锁的线程所造成的损失降低到最低（参见 § 3.1.2）。这同时也能统一地保证当线程进入中断的状态时，一定会有 InterruptedException 异常被抛出，而不是仅仅在线程碰巧由于 wait 而阻塞时才抛出异常。

```
class Semaphore implements Sync  {

  protected long permits; // current number of available permits

  public Semaphore(long initialPermits) {
```


java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

```
    permits = initialPermits;
  }

  public synchronized void release() {
    ++permits;
    notify();
  }
  public void acquire() throws InterruptedException {
    if (Thread.interrupted()) throw new InterruptedException();
    synchronized(this) {
      try {
        while (permits <= 0) wait();
        --permits;
      }
      catch (InterruptedException ie) {
        notify();
        throw ie;
      }
    }
  }

  public boolean attempt(long msecs)throws InterruptedException{
    if (Thread.interrupted()) throw new InterruptedException();
    synchronized(this) {
      if (permits > 0) {     // same as acquire but messier
        --permits;
        return true;
      }
      else if (msecs <= 0)   // avoid timed wait if not needed
        return false;
      else {
        try {
          long startTime = System.currentTimeMillis();
          long waitTime = msecs;

          for (;;) {
            wait(waitTime);
            if (permits > 0) {
              --permits;
              return true;
            }
            else {                    // check for time-out
              long now = System.currentTimeMillis();
              waitTime = msecs - (now - startTime);
              if (waitTime <= 0)
                return false;
            }
          }
        }
        catch(InterruptedException ie) {
          notify();
          throw ie;
        }
      }
    }
  }
}
```


java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

3.7.2 委托操作

协同操作可以用于解决受保护方法，由于同一个等待集合中的不同线程等待不同的逻辑条件所造成的低效问题。当 notifyAll 试图去通知那些等待某个条件变化的线程时，会同时唤醒那些等待与这个条件完全无关的线程。这些无用的信号及其导致的大量上下文切换现象可以通过将不同的等待条件委托给不同的辅助对象来减轻。

在 § 3.4.1 中，我们使用信号机很容易地实现了这个效果。在这里，我们将从最基本的地方开始，并试图通过利用特定问题的特殊属性来获得更好的性能。当设计问题可以进行优化处理，并且这些优化只能应用于纯等待与通知机制时，这里介绍的技巧才是有使用价值的。

这种依照状态依赖操作来分割类的做法，扩展了 § 2.4.2 中所描述的按锁来分割类的思想，同时也借鉴了 State as Object 模式（见《Design Patterns》）。然而，由于下列的一些约束，这种设计也仅限在比较狭小范围的构件中：

- 由于辅助类必须访问公共的状态，所以，无法完全把每个辅助类按自包含方式隔离起来。在辅助类之间，对公共状态的不同访问必须被正确地同步。
- 每个可能影响其他辅助类的受保护条件的辅助类必须提供对这种影响的有效通知，以避免活跃性问题。
- 和 wait 机制有关的辅助方法的同步，必须注意避免出现嵌套的监视器问题（参见 § 3.3.4）。

3.7.2.1 设计步骤

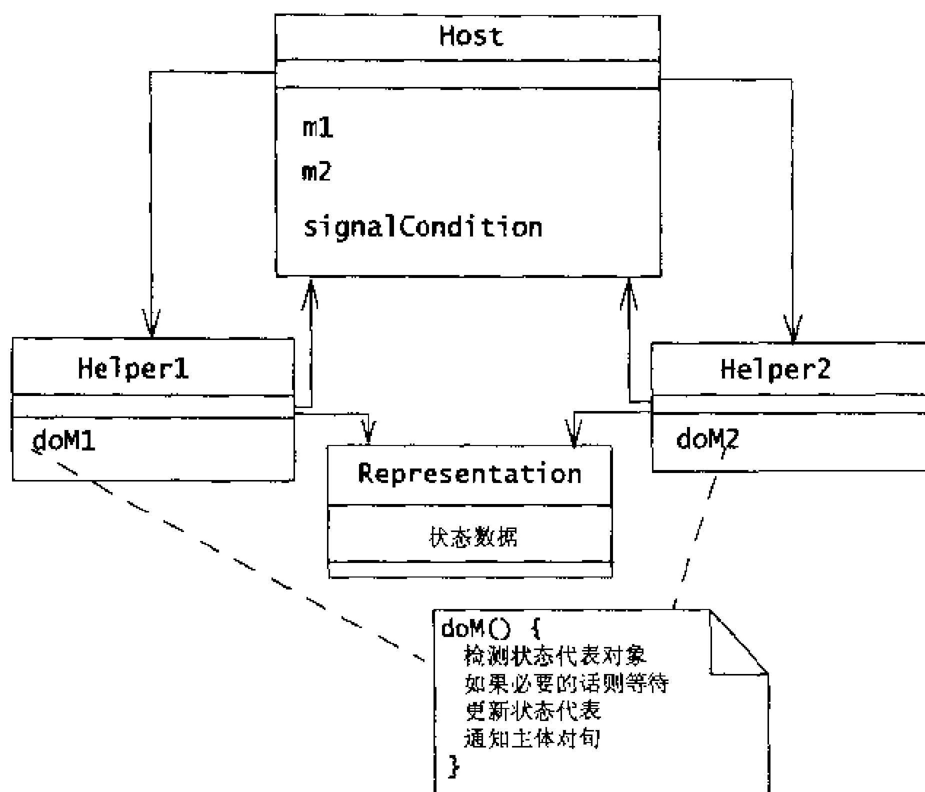
解决这些约束的常用方法首先就是将 Host 类尽可能地分解成小块：一个类用作被共享的状态的代表，以及各种类型的辅助类。随后，处理剩余的协同操作的协调问题。最后，将这些小块组合成有用的类：

- 定义一个类，比如说 Representation，用以存放多个方法中都要使用的成员变量，这仅仅是一个记录类型的类，没有私有的成员变量，且允许在特定的同步代码段中进行无限制的访问与更新操作。
- 对每一组共用相同的等待条件的功能，定义一个辅助类。每个辅助类的实例都要有指向主体对象和状态代表对象的（也可以间接地通过主体对象的引用来引用状态代表对象）引用变量。
- 将 Host 类设计为一个中转点，每个 Host 的公共方法都应该是一个非同步的转发方法。同时，定义一些非同步方法，这些方法是给辅助类在改变那些可能会影响其他辅助类的状态时调用的。分发相关的 notify 和 notifyAll 的调用（或者，这些通知可以直接在辅助对象间传递）。这个主体对象也应该在其构造函数中初始化所有的辅助对象。
- 每个辅助对象都必须在保证安全的前提下避免出现活跃性问题，特别需要注意的是：
    - 如果条件检测包含了对共享状态代表对象的检测，那么这个操作就必须在共享

java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

状态代表和辅助对象都已锁定的情况下进行。

- 在进入任何 wait 之前，共享状态代表对象的锁都必须被释放，但是辅助对象上的锁仍可以被保持，以避免丢失信号（参见§3.2.4），在那种情况下，等待在通知发出后才开始。
- 为避免潜在的死锁，通知的分发不需要同步。



一个通用的 helper 方法可以采用如下的形式：

```
void doM() throws InterruptedException {
  for(;;) {                    // wait loop
    synchronized(this) {       // check->wait must lock this
      synchronized(rep) {      // check->act must lock rep
        boolean canAct = inRightState(rep);
        if (canAct) {
          update(rep);         // the guarded action
          break;
        }
      }                        // release rep lock before wait
      wait();                  // fall-through if !canAct
    }                          // release lock before signal
  }
  host.signalChange();
}
```

3.7.2.2 有界缓冲区

作为最后一个 BoundedBuffer 的例子，我们将构建一个委托版本，这个版本利用实际使

java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

用的数据结构和算法的一些特点提升性能。最终，这个版本会比前一个版本快一些，不过它也只是用于演示前面介绍的技巧的。

首先，我们需要将辅助对象分为执行 put 和执行 take 操作的。委托的设计方案通常对每个方法都需要一个辅助类。但是在这里，通过利用观察到的所有权传递现象，我们可以只使用一个辅助类（两个实例）。如 § 2.3.4 中所提到的，一个单独的 exchange 操作就可以用于 put 类型和 take 类型的传递操作。例如 exchange(null)，就完成了 take 操作。基于缓冲区的 exchange 版本使用当前的数组存取槽的值作为参数替换旧值，并在随后循环地向后移动到下一个位置。

如果将辅助类 Exchanger 定义为内部类的话，那它就可以很便捷地直接访问主体和作为共享状态代表的数组。我们还需要一个数组存取槽的计数器，用于当没有存取槽可用时指示 exchange 操作必须停止。对执行 put 操作的辅助类，计数器的值从数组的容量开始；而对于执行 take 操作的辅助类，计数器的值是从 0 开始的。Exchange 操作只有在存储槽的数量大于 0 的时候才可以执行。

每个成功的 exchange 操作都会减少这个计数器的值。由计数器的值为 0 所造成的等待只有在执行增加操作的辅助对象发出了相应通知时才会被释放。其具体实现如下：向其他的 exchanger 发出 addedSlotNotification，并通过主体来分发。

在这个特定的设计中，有另外一个特殊的考虑使得这个设计的性能有了微小地提高。虽然保存数据的数组必须被两个辅助对象所共享，但当仅需支持 put 和 take 方法，这个数组就不需要同步的保护。这可以通过将主体类定义为 final 来保证。没有同步锁保护的情况下还能正确运行的原因是，在这里的算法中，任何执行中的 put 操作所能操作的数组存储槽必须和当前正被 take 操作所访问的存储槽不同。此外，外部的同步足以排除那些内存可见性问题（参见 § 2.2.7）。与之相反，BoundedBufferWithSemaphores 类需要对数组操作进行锁定，因为它不限制在什么时候只有一个 put 或 take 方法执行。

为了进一步提高性能，这里使用了 notify 来通知，因为这个类满足了使用 notify 的条件（在 § 3.2.4.2 有所讨论）：(1)在每个辅助类中的每个等待任务都在等待相同的逻辑条件（take 等待非空的条件，put 等待非满的条件）；（2）每次通知至少使得一个线程继续执行——每次 put 操作使得一个 take 操作可行，每次 take 操作使得一个 put 操作可行；（3）可以通过重新通知来处理中断问题。

为了再榨取一点性能，在这个类中，由于很容易跟踪当前是否有线程在等待，因此，可以仅在有需要被通知的线程时才发出 notify。这个策略对性能的影响和不同的 JVM 实现相关。由于 notify 操作所造成的消耗正变得越来越小，所以在这里通过记录线程来避免 notify 调用的改进也渐渐变得没有必要了。

```
final class BoundedBufferWithDelegates {
  private Object[] array;
  private Exchanger putter;
  private Exchanger taker;

  public BoundedBufferWithDelegates(int capacity)
   throws IllegalArgumentException {
    if (capacity <= 0) throw new IllegalArgumentException();
```


java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

```
    array = new Object[capacity];
    putter = new Exchanger(capacity);
    taker = new Exchanger(0);
  }


  public void put(Object x) throws InterruptedException {
    putter.exchange(x);
  }

  public Object take() throws InterruptedException {
    return taker.exchange(null);
  }

  void removedSlotNotification(Exchanger h) { // relay
    if (h == putter)  taker.addedSlotNotification();
    else              putter.addedSlotNotification();
  }
  protected class Exchanger {                    // Inner class
    protected int ptr = 0;             // circular index
    protected int slots;               // number of usable slots
    protected int waiting = 0;         // number of waiting threads

    Exchanger(int n) { slots = n; }

    synchronized void addedSlotNotification() {
      ++slots;
      if (waiting > 0) // unblock a single waiting thread
        notify();
    }

    Object exchange(Object x) throws InterruptedException {
      Object old = null; // return value

      synchronized(this) {
        while (slots <= 0) { // wait for slot
          ++waiting;
          try {
            wait();
          }
          catch(InterruptedException ie) {
            notify();
            throw ie;
          }
          finally {
            --waiting;
          }
        }

        --slots;             // use slot
        old = array[ptr];
        array[ptr] = x;
        ptr = (ptr + 1) % array.length; // advance position
      }

      removedSlotNotification(this); // notify of change
```


java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

```
            return old;
          }
        }
      }
```

3.7.2.3 合并类

所有种类的同步分割都可以通过两种方法来实现。在以锁来分割的情况中（参见§2.4.2），可以创建 helper 类并在主体对象中转发操作，或者你可以把操作方法留在主体对象中，但在调用它们时，通过概念上代表不同辅助者的 Object 来同步。

当分割状态依赖的操作时，同样要遵守相同的原则。除了将操作委托给辅助者，也可以将方法保留在主体对象中，并添加一些概念上代表辅助者的 Object。Object 仅仅作为锁在同步中使用。那些用于等待和通知的对象可以被用作监视器。即，存放需要等待和通知的线程的地方。

将辅助者合并到主体类中会使得主体对象变得更加复杂，但是，由于短路（short-circuited）方法调用以及类似的原因，这同时也会提高性能。依此方法来进一步简化，我们可以定义一个更简明的、效率更高，当然也是更复杂的 BoundedBuffer。

```
final class BoundedBufferWithMonitorObjects {
  private final Object[] array;    // the elements

  private int putPtr = 0;          // circular indices
  private int takePtr = 0;

  private int emptySlots;          // slot counts
  private int usedSlots = 0;

  private int waitingPuts = 0;     // counts of waiting threads
  private int waitingTakes = 0;

  private final Object putMonitor = new Object();
  private final Object takeMonitor = new Object();

  public BoundedBufferWithMonitorObjects(int capacity)
   throws IllegalArgumentException {
    if (capacity <= 0)
      throw new IllegalArgumentException();

    array = new Object[capacity];
    emptySlots = capacity;
  }
  public void put(Object x) throws InterruptedException {
    synchronized(putMonitor) {
      while (emptySlots <= 0) {
        ++waitingPuts;
        try { putMonitor.wait(); }
        catch(InterruptedException ie) {
          putMonitor.notify();
          throw ie;
        }
        finally { --waitingPuts; }
```


java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

```
      }
      --emptySlots;
      array[putPtr] = x;
      putPtr = (putPtr + 1) % array.length;
    }
    synchronized(takeMonitor) { // directly notify
      ++usedSlots;
      if (waitingTakes > 0)
        takeMonitor.notify();
    }
  }

  public Object take() throws InterruptedException {
    Object old = null;
    synchronized(takeMonitor) {
      while (usedSlots <= 0) {
        ++waitingTakes;
        try { takeMonitor.wait(); }
        catch(InterruptedException ie) {
          takeMonitor.notify();
          throw ie;
        }
        finally { --waitingTakes; }
      }
      --usedSlots;
      old = array[takePtr];
      array[takePtr] = null;
      takePtr = (takePtr + 1) % array.length;
    }
    synchronized(putMonitor) {
      ++emptySlots;
      if (waitingPuts > 0)
        putMonitor.notify();
    }
    return old;
  }

}
```

3.7.3 特定通知

除了将类似于BoundedBufferWithMonitorObjects类中的辅助Object看作是设计工作中的顶点，你还可以把它们当作是任何需要分离监视器的设计的实现工具。Tom Cargill所发明的特定通知模式正是使用了这个策略。

这个策略的基本思想是通过监视器中的waits来使任务线程进入睡眠状态，这里的监视器就是普通的Object（更常见的是，那些帮助记录的类的实例），我们仅仅使用了它们的等待集合。每个任务使用一个监视器，或者一组必须被单独通知的任务使用一个监视器。在大多数的情况下，需要每个线程中都有一个监视器，而一组需要同时被唤醒的线程可以使用相同的监视器。这些监视器的用途和条件队列（condition queue）类似，这种条件队列在某些基于监视器的并发编程语言中有直接的支持（参见§3.4.4）。监视器与它们的主要区别是，由于没有

java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

了语言的直接支持，当使用这些辅助的监视器时，必须更加小心以避免嵌套监视器的问题。

无论何时，当你需要线程 wait 并且通知策略并没有动态地依赖于线程的属性时，特定通知的方法可能都是有用的。一旦某个线程被放入其等待集合中，除了唤醒它以外，没有任何方法可以访问到它。可以应用这种约束的情况包括以下几种：

- 通过使用明确的队列（例如，FIFO、LIFO 及优先级队列）来支持特定的调度策略。
- 把即将开始的任务按照它们所等待执行的方法来划分到不同的队列中。这可以用于扩展那些基于冲突集合（参见 § 3.3.2）的技巧。

然而，虽然将调度约束（例如 FIFO）和对基于逻辑状态或者执行状态的约束合并在一起十分诱人。但是，这两种应用间的交互通常会导致概念和逻辑上的问题。例如，你可能需要考虑这种情况：由于线程 A 比线程 B 先到达，所以它应该比线程 B 先执行，但线程 B 在逻辑上是可以执行的，而线程 A 却不能。这就必须通过精心构建相应的机制来重新将线程排入队列，管理锁定顺序以及处理其他复杂的情况。

3.7.3.1 设计步骤

主要的设计步骤是 § 3.7.2.1 中所描述步骤的特殊化版本。如下创建或修改一个类，比如 Host：

- 对每个或者每组需要特定通知的线程，创建一个用作监视器的对象，这些监视器可以存于数组或者其他集合中，或者在执行中动态创建。
- 在作为监视器的类中执行维护工作，用以管理等待和通知的操作，以及这些操作的超时和中断策略。如 § 3.7.3.2 中的 WaitNode 所示，这通常需要维护一个 released 成员变量，用以记录等待的线程是否已经因为通知、中断或者超时而释放。这些类可能有例如 doWait、doTimeWait、doNotify 和 doNotifyAll 的方法，以执行可信赖的等待与通知操作，并按照需要处理中断和超时。如果你无法在作为监视器的记录对象中加入这些维护工作，那么这些问题就需要在主体类中被解决。
- 在每个会使任务挂起的主体方法中使用其所对应的监视器的 monitor.doWait()方法。必须通过确保 wait 方法不在同步主体对象的代码段中被调用，来避免嵌套的监视器问题。最简单同时也最令人满意的形式如下所示：

```
boolean needToWait;    // to remember value after synch exit
synchronized (this) {
  needToWait = ...;
  if (needToWait)
    enqueue(monitor); // or any similar bookkeeping
}
if (needToWait) monitor.doWait();
```

- 在每个会使任务恢复执行的方法中使用 monitor.doNotify()，同时还要处理超时或者中断所造成的问题。

3.7.3.2 FIFO 信号机

特定通知可以被用于实现 § 3.4.1.5 中讨论的 FIFO 信号机类。FIFO 信号机随后即可以被

java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

用在那些依赖于 FIFO 属性的工具类中。

下面的这个 FIFOSemaphore 类（util.concurrent 中对应类的一个简化版本）被定义为 § 3.7.1 中的 Semaphore 的子类。这个 FIFOSemaphore 类维护着一个 WaitQueue 的链表，其中保存的是用作监视器的 WaitNode。不能立刻获得许可证的 acquire 操作将会使一个等待监视器对象进入队列。release 操作从队列中取出最早的等待节点并通知它。

WaitNode 中的 released 成员变量将会帮助其管理通知和中断之间的交互。在 release 操作过程中，任何由于中断而中止的监视器都会被跳过。与之相反，一个被中断的 wait 会先检测这个变量，以判断除了被中断以外，自己是否已被通知过。如果是的话，它必须执行前滚操作，忽略异常，但需通过重置中断状态（参见 § 3.1.2）来保持取消的状态（通过在超时的时候设定 released 的状态，一个在这里未给出的 doTimedWait 方法可以按类似的方法来实现）。在不恰当时间出现的潜在的中断导致了 release 中的重试循环。

FIFOSemaphore、WaitQueue 以及 WaitNode 之间的交互在确保必要的原子性的同时也避免了嵌入的监视器问题。同时它们还演示了某些支持 FIFO 策略中的不确定性。我们仅能保证在某个起点和终点之间，这个信号机是 FIFO 的。这个起点开始于 acquire 操作的 synchronized (this)。终点通常出现在由于 notify 而释放 wait 时。两个进入 acquire 操作的线程可能会按照与进入不同的顺序获得锁，例如，第一个进入的线程可能在运行至 synchronized(this)之前就被 JVM 调度暂停了。类似的，某个线程先于另一线程被释放，但它可能在另一个线程之后返回至调用者。尤其是在多处理器系统中，这个类只提供了其使用者期望的一些担保。

调度规则的改变可以通过使用一个不同种类的队列来实现：例如，某个基于 Thread.getPriority 的队列。然而，通过改变这个类来处理那些基于执行或逻辑状态的语义限制（semantic restriction）是非常困难的。大多数语义限制需要被通知或者被中断的线程获得其他的锁。这样就会增加这里所使用方案的复杂度，因为这里的方案利用了被唤醒的线程无需存取主锁的特点。这些问题需要针对不同的应用采取不同的方案来解决。

```
class FIFOSemaphore extends Semaphore {

  protected final WaitQueue queue = new WaitQueue();

  public FIFOSemaphore(long initialPermits) {
    super(initialPermits);
  }

  public void acquire() throws InterruptedException {
    if (Thread.interrupted()) throw new InterruptedException();

    WaitNode node = null;

    synchronized(this) {
      if (permits > 0) {     // no need to queue
        --permits;
        return;
      }
      else {
        node = new WaitNode();
```


java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

```
        queue.enq(node);
      }
    }

    // must release lock before node wait

    node.doWait();
  }

  public synchronized void release() {
    for (;;) {                    // retry until success
      WaitNode node = queue.deq();

      if (node == null) {       // queue is empty
        ++permits;
        return;
      }
      else if (node.doNotify())
        return;

      // else node was already released due to
      //   interruption or time-out, so must retry
    }
  }
  // Queue node class. Each node serves as a monitor.

  protected static class WaitNode {
    boolean released = false;
    WaitNode next = null;        // to arrange in linked list

    synchronized void doWait() throws InterruptedException {
      try {
        while (!released)
          wait();
      }
      catch (InterruptedException ie) {

        if (!released) {         // interrupted before notified
          // Suppress future notifications:
          released = true;
          throw ie;
        }
        else {                   // interrupted after notified
          // ignore exception but propagate status:
          Thread.currentThread().interrupt();
        }

      }
    }

    synchronized boolean doNotify() { // return true if notified

      if (released)              // was interrupted or timed out
        return false;
      else {
        released = true;
```


java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

```
            notify();
            return true;
          }
        }

        synchronized boolean doTimedWait(long msecs)
         throws InterruptedException {
           // similar
        }
      }
      // Standard linked queue class.
      // Used only when holding Semaphore lock.

      protected static class WaitQueue {
        protected WaitNode head = null;
        protected WaitNode last = null;

        protected void enq(WaitNode node) {
          if (last == null)
            head = last = node;
          else {
            last.next = node;
            last = node;
          }
        }

        protected WaitNode deq() {
          WaitNode node = head;
          if (node != null) {
            head = node.next;
            if (head == null) last = null;
            node.next = null;
          }
          return node;
        }
      }
    }
```

3.7.4 进阶阅读

使用例如 Dekker 算法和基于令牌的算法来实现锁的技巧可以在 Andrew 和其他列在§1.2.5 中的并发编程书中找到。不过，没有任何理由去使用基于这些技巧的通用并发控制类，而不去使用内建的 synchronized 方法和代码段。

特定通知模式最先被介绍在下列文献中：

Cargill 和 Thomas, "Specific Notification for Java Thread Synchronization",《Proceedings of the Pattern Languages of Programming Conference》, 1996.

另一个使用特定通知来精炼 notifyAll 的结构可见：

Mizuno 和 Masaaki. "A Structured Approach for Developing Concurrent Programs in Java",《Information Processing Letters》, 1999.

有关本节所讨论技巧的更多例子和扩展可以在本书的在线补充中找到。

java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

第4章

创建线程

要对所有使用与线程有关的功能的方法进行分类是一件很困难的事情。但是有两种常用的方式可以从它们对下面这行代码的不同观点上进行区分：

```
new Thread(aRunnable).start();
```

这是一种特别的方法调用（即调用 Runnable 接口的 run 方法），还是创建了一个特别的对象（即一个新的 Thread 类的实例）呢？显然，这里是两者兼而有之。但是，基于上述的两种观点，就有了两种使用线程的方式。在第 1 章的讨论中也涉及了对这两种方式的描述：

基于任务（Task-based）。这种情况下，使用线程的主要原因就是能够异步地调用执行某个任务的方法。这里的任务可以是一个简单的方法，或是一个会话过程。基于线程的技术能够支持消息传递模式，这种模式消除了纯粹的过程调用的局限。基于任务的设计可以在事件框架、并行计算，以及有大量 IO 交互的系统中看到。

基于参与者（Actor-based）。这种情况下，使用线程的主要原因是创建并启动一个新的、自治的、活动的、类似进程的对象。这个对象可能反过来作用于外部事件、与其他参与者交互等等。基于参与者的设计能在交互、控制和分布式系统中见到。它们也是大多数实现并发的常规方法所关注的焦点。（***任务*** 与***参与者*** 这两个术语有许多重叠、相似的含义。我们把对它们的使用按上面的方式进行界定。）

在基于任务的系统中，被动对象有时会发送主动（线程推进的）消息，而在基于参与者的系统中，主动对象通常发送被动消息。正如通常由于人为差别所造成的情况，这两种方式没有哪种永远是最好的，并且会有一个巨大的中间范围能够使用两种方式中的一种或两种同时进行设计。

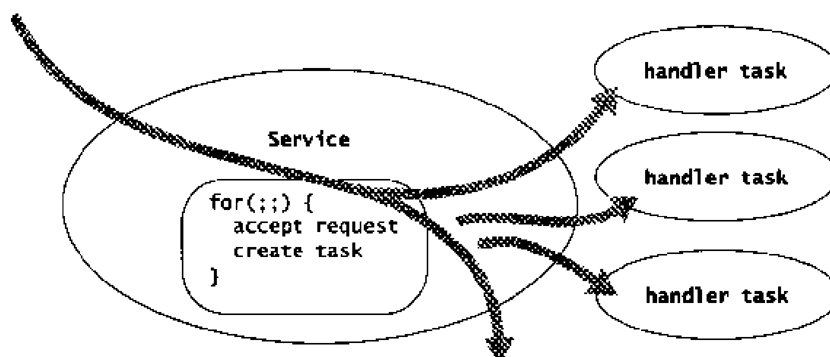
基于参与者的方法通常用于与其他系统交互的后台线程的构造。它们也用于定义有内部活动的实体，例如在 § 3.2.4 介绍的 GamePlayer。它们的主要方法经常使用一个循环体，其中有处理方法：

```
for(;;) { acceptAndProcessCommand(); }
```

在为了某种概念或基于性能的原因来执行一个给定任务、服务或异步计算，而不是依赖直接的过程调用的情况下，通常会使用基于任务的方式。基于任务的设计提供了逻辑异步地映射到线程和基于线程的构造之间的区分。在本章会大量讨论这两个方面的问题。

java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

作为一个初始的例子，一个 Web 服务介绍了实现普通的基于线程的设计。这里，一个运行的 WebService 是一个参与者风格的后台线程——它通过监听新的输入请求与周围环境持续交互。但是对 handler.process 的调用则是基于任务的方式——一个新的任务开始处理每一个输入的请求。这里，为了能够简明地说明问题，输入请求只是一个简单的数字，并且处理只是把数字的负值返回给客户。




```
class WebService implements Runnable {
  static final int PORT = 1040;   // just for demo
  Handler handler = new Handler();

  public void run() {
    try {
      ServerSocket socket = new ServerSocket(PORT);
      for (;;) {
        final Socket connection = socket.accept();
        new Thread(new Runnable() {
          public void run() {
            handler.process(connection);
          }}).start();
      }
    }
    catch(Exception e) { } // die
  }

  public static void main(String[] args) {
    new Thread(new WebService()).start();
  }
}
```

```
class Handler {

  void process(Socket s) {
    DataInputStream in = null;
    DataOutputStream out = null;
    try {
      in = new DataInputStream(s.getInputStream());
      out = new DataOutputStream(s.getOutputStream());
      int request = in.readInt();
      int result = -request;     // return negation to client
      out.writeInt(result);
```


java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

```
      }
      catch(IOException ex) {}      // fall through

      finally {                     // clean up
        try { if (in != null) in.close(); }
        catch (IOException ignore) {}
        try { if (out != null) out.close(); }
        catch (IOException ignore) {}
        try  { s.close(); }
        catch (IOException ignore) {}
      }
    }

  }
```

本章把线程的构造和组织技术划分为下面的章节：

§4.1 介绍一系列为了实现概念性的单向消息的可选方式，有时通过使用线程或基于线程的轻量级执行框架来异步地初始化任务。

§4.2 讨论使用单向消息策略的组件网络的系统设计。

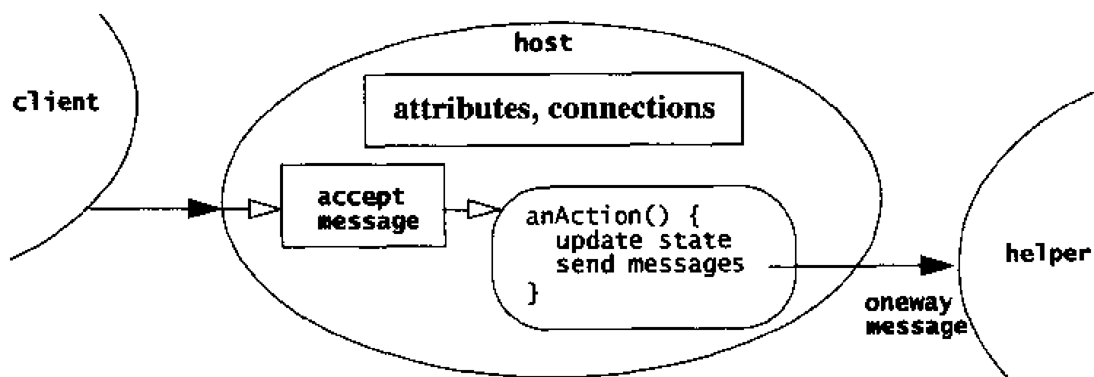
§4.3 介绍可选的用于计算结果或为初始化该线程的客户提供服务的线程的构造方式。

§4.4 检查在使用多处理器的情况下，能够用来提高性能的问题分解技术。

§4.5 提供一个活动对象系统的设计构造和框架的概览，其中使用了 CSP 进行阐述。

本章中的许多设计跨越了并发、分布式与并行编程的范围。这些介绍着重于并发的单 JVM 的解决方案。但是其中的一些架构在开发支持多处理器或多计算机的系统和框架中也经常采用。

4.1 单向消息



一个主体对象可以向一个或多个接收者（recipient）发送一个逻辑的单向消息，而不用依赖消息的结果。发送一个单向消息会以某种方式导致某个任务被执行。这个任务可能只由一行代码组成，也可能代表一个必须获得许多资源和成小时计算时间的会话。但是发布消息的线程并不依赖消息导致的任务的结果、任务结束的时间，或者（通常情况下）任务是否曾经结束过。常见的例子包括：

java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

事件 ***(Event)***：如鼠标单击等。

通知 ***(Notification)***：如状态改变警告。

记录 ***(Postings)***：邮件消息、股票报价等。

激活 ***(Activation)***：创建小程序（applet）、后台线程等。

命令 ***(Command)***：打印请求等。

中继 ***(Relay)***：消息转发和分派。

在发送者和接收者之间的单向交互不需要严格地异步。例如：发送者可能负责保证接收者实际上收到了消息。另外，发送者或其他对象可能在后来希望能够取消或回滚因发送消息而出现的任务效果［当然这（任务取消或回滚）也不总是可以做到的，例如任务已经完成的情况，参见§3.1.2］。

如果每个任务都能立刻运行，你便可以通过过程调用的方式触发单向消息。在这种方式下，无论如何调用者都要等到消息触发的任务结束为止，即使它没有理由这样做。但是经常由于基于性能、设计理念或逻辑方面的原因而通过基于线程的构造来发送消息，在这种方式下，线程中相关的任务独立执行。

4.1.1 消息格式

在单向消息传送的方式中，消息中封装了许多不同风格的调用，虽然它们中的一些与分布式或多进程程序关联得更紧密一些（参见§1.2.2），但是它们中的任何一个都能够与在本节所讨论的构造方式一起使用。除了直接方法调用外，消息格式可能还包括：

命令字串 ***(Command string)***。接收者必须对收到的命令字串进行解析、解码，然后再将相关的任务分派出去。命令字串消息广泛应用于基于套接字和管道的通信中，尤其是在Web服务中。

事件对象 ***(Event object)***。消息中包含了一个对事件的结构化描述。接收者将与事件关联的操作任务转发。在包括java.awt的GUI框架，以及由java.beans支持的组件框架中，事件对象都得到了广泛应用。

请求对象 ***(Request object)***。消息中包含了一个（排列过或串行化过的）方法名和参数的编码。接收者将相关的方法调用发送给一个辅助对象，由这个辅助对象来执行这个方法。请求对象这种方式常被用在分布式对象支持系统中，如同在java.rmi与org.omg.corba中使用的那样。它的变体也被用在Ada的任务中。

类对象 ***(Class object)***。消息是一个类的代表（例如通过一个.class文件），接收者可以通过它将类实例化。这种模式应用在java.applet框架中，同时也应用在远程激活协议中。

可运行对象 ***(Runnable object)***。消息是由一些接收者执行的代码组成的。可运行事件的混合形态（它既包括了一个事件的描述，也包括了相关的动作）被应用于一些事件框架中。使用序列化(serialized)的可运行对象的扩展形式会在移动代理框架(mobile agent framework)中见到。

任意对象 ***(Arbitrary object)***。一个发送者可以把所有的对象都当作消息来处理，在消息中，把对象作为方法参数或通过一个Channel传递（参见§4.2.1）。例如在JavaSpaces™ 框

java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

架中，发送者可以把任何序列化对象作为消息发送［也被称为“条目”（entry)]。接收者只接收那些类型和字段值与一个指定的匹配准则相一致的条目，并以一种合适的方式处理这些对象。

这些格式之间的差别反映了调用者对接收者需要用来执行任务的代码了解的程度，也包括了其他的事情。通常情况下，使用可运行对象既是最有效的，也是最方便的，尤其是在基于线程的框架中，线程构造函数使用 **Runnable** 类的实例作为参数的情况。我们将着重介绍这种方式，但也会在某些时候介绍其他的方式。

4.1.2 开放调用（Open Call）

考虑一下中心主体对象在一个调用链中的情况，主体对象从任意数量的客户那里接收到 req 请求，在处理该请求的过程中，主体对象必须向一个或多个辅助对象发送逻辑的单向处理消息。此外，我们经常会忽视这样一个事实，在处理请求之前，我们需要花费大量的努力来对请求进行解码，这些请求可能是从套接字中读出的，例如在 WebService 类中，等等。另外，通过使用在§2.4.4 和§3.5.2 介绍的构造方式。所有在这一节中讨论的类都能够进行扩展，从而向多个辅助类发送多点传送消息。

这里的设计动力是响应时间。如果一个主机在繁忙地处理请求，那么它就不能再去接收一个新的客户请求。这就增加了从客户端来的新请求的响应时间，并从整体上降低了服务的可用性。

一些响应时间的特性能够简单地通过使用在§2.4 介绍的通过（pass-through）和开放调用来处理。如：

```
class OpenCallHost {                          // Generic code sketch
  protected long localState;
  protected final Helper helper = new Helper();

  protected synchronized void updateState(...) {
    localState = ...;
  }

  public void req(...) {
    updateState(...);
    helper.handle(...);
  }
}
```

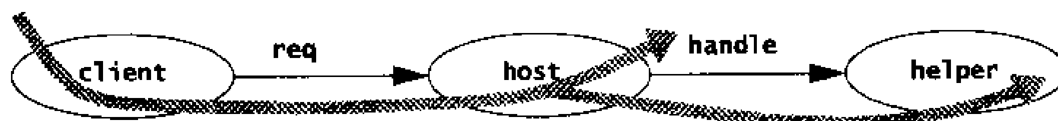
这里，即使对 helper.handle 方法的调用是比较耗时的，主体对象仍然能够从运行不同线程里的客户那里接收到新的请求。请求的接收率只与主体对象用于更新本地状态所花费的时间有关。

通常，使用开放调用消除了与给定主机相关的瓶颈点，但是它并没有提到如何开始向一

java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

个系统引入并发这样一个广泛的问题。开放调用只在下面这种情况中是有用的，即客户端已经知道如何在他们期望或需要的时候使用其他的允许独立执行的方法。

4.1.3 每消息一线程（Thread-Per-Message）



通过将发送的消息在单独线程里运行的方式可以在单向消息设计中引入并发，如下面的代码所示：

```
class ThreadPerMessageHost {                    // Generic code sketch
  protected long localState;
  protected final Helper helper = new Helper();

  protected synchronized void updateState() {
    localState = ...;
  }

  public void req(...) {
    updateState(...);
    new Thread(new Runnable() {
      public void run() {
        helper.handle(...);
      }
    }).start();
  }
}
```

在多个并发任务能够比同样多的任务串行执行运行得更快时，这种策略便能提高系统的吞吐量，这种情况下，通常是由于这些任务是与 IO 资源或计算资源有关，并且运行在一个多处理器的系统上。由于客户端无需等待彼此的任务完成，所以这种方式还可以增强公正性并提高可用性。

决策是否创建或启动线程来执行任务并不比决策是否创建其他类型的对象或发送其他类型的消息要困难得多：关键是要物有所值。

由于创建线程远比直接方法调用开销多，所以每消息一线程的设计会增加请求的反应时间。但是，在任务的执行与线程构造相比非常耗时，或者任务是基于会话的，或者需要将它与其他独立的活动隔离开，或者能使用 IO 或 CPU 的并行性的情况下，这种牺牲（每任务一线程）通常是值得的。但是即使是构造的开销是可接受的，性能的问题也会显现出来。JVM 的实现和（或）操作系统可能不会很好地对太多线程的做出响应。例如：它们可能耗尽了所有与线程有关的系统资源。另外，随着线程数量的增加，线程调度和上下文切换会占用大量的处理时间。

4.1.3.1 执行者

由于直接依赖于 Thread 类，ThreadPerMessage 类的代码风格也将成为一个问题。在这

java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

种用法下，很难调整线程的初始化参数以及应用中用到的线程相关的数据（参见 § 2.3.2）。我们可以通过创建一个接口来避免这种情况，如下所示：

```
interface Executor {
  void execute(Runnable r);
}
```

上面的接口可以由下面的类来实现：

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

这个实现可以使用在如下的类中：

```
class HostWithExecutor {                    // Generic code sketch
  protected long localState;
  protected final Helper helper = new Helper();
  protected final Executor executor;

  public HostWithExecutor(Executor e) { executor = e; }

  protected synchronized void updateState(...) {
    localState = ...;
  }

  public void req(...) {
    updateState(...);
    executor.execute(new Runnable() {
      public void run() {
        helper.handle(...);
      }
    });
  }
}
```

对这种接口的使用使得我们还可以用轻量级可执行框架替代线程。

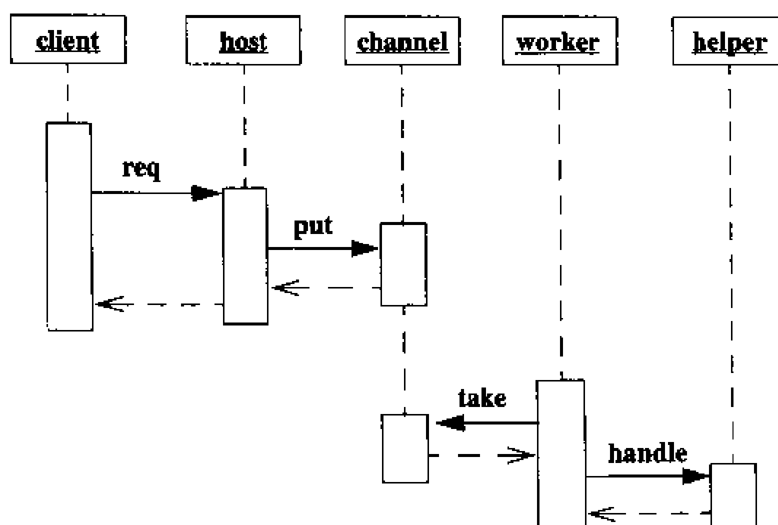
4.1.4 工作者线程（Worker Thread）

轻量级可执行框架填充了开放调用设计与每消息一线程设计之间的空隙。在以一定的使用限制为代价的情况下，为了最大化（或至少是提高）吞吐量并最小化平均反应时间，你可以在需要引入有限并发的情况下使用它。

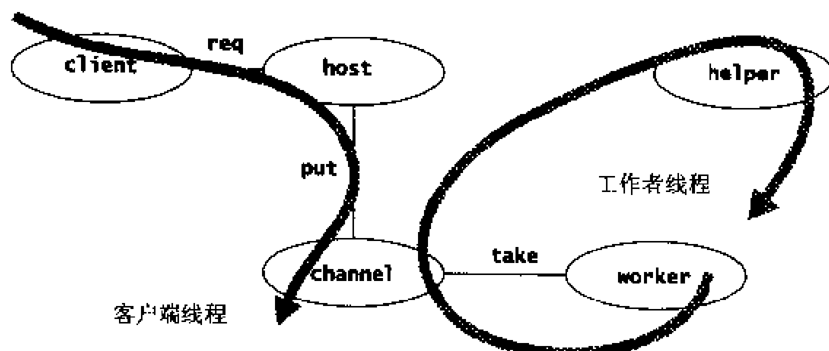
轻量级可执行框架可以通过很多种方式构造，但是它们的基本设计思路都是使用一个线程执行许多不相关的任务（这里是连续地执行）。这些线程被称为工作者线程、后台线程，或者在超过一个线程以上的情况时被称为线程池。

每个工作者从主体那里持续地接收新的 Runnable 命令，并以某种通道的方式（如一个队列、缓冲等，参见 § 3.4.1）持有它，直到它能被运行。这种设计有着经典的产生者-消费者关系的形式：主体产生任务，工作者通过运行它们将它们消费。

java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！



通过允许你将许多小的、逻辑上异步的执行单元作为任务打包，轻量级的可执行框架能够改善一些基于任务的并发程序的结构，而不用过多考虑性能上的影响：因为将一个 Runnable 对象放到队列里一般来说比创建一个新的 Thread 对象要快。同时，由于你可以控制工作者线程的数量，所以能将资源耗尽的机会降到最小，并减少上下文切换的开销。显式的排队等候在调整执行语义时还会带来更大的灵活性。例如，你可以使用优先级队列实现通道，这样对排序任务的控制会比使用 Thread.setPriority 方法控制更有保证（参见 §4.3.4）。



为了与纯粹的基于线程的版本进行互操作，工作者线程要被打包成一个***执行者(Executor)***。下面是一个能够用在 HostWithExecutor 类中替代每消息-线程版本的常用的实现方式。

```
class PlainWorkerPool implements Executor {
  protected final Channel workQueue;

  public void execute(Runnable r) {
    try {
      workQueue.put(r);
    }
    catch (InterruptedException ie) { // postpone response
      Thread.currentThread().interrupt();
    }
  }
```


java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

```
    public PlainWorkerPool(Channel ch, int nworkers) {
      workQueue = ch;
      for (int i = 0; i < nworkers; ++i) activate();
    }

    protected void activate() {
      Runnable runLoop = new Runnable() {
        public void run() {
          try {
            for (;;) {
              Runnable r = (Runnable)(workQueue.take());
              r.run();
            }
          }
          catch (InterruptedException ie) {} // die
        }
      };
      new Thread(runLoop).start();
    }
  }
```

4.1.4.1 设计选择

围绕着基于工作者线程的可执行框架要下的第一个决定就是要不要创建或使用它们。主要的问题在于，在基本的线程中是否有你不需要或想放弃的属性？如果不是这样，那么找到一个比 JVM 产品实现中所带的内置线程支持是更好的解决方案是不太可能的。

获得工作者线程性能优势的代价就是要引入一些会影响工作者线程类的设计，以及使用额外的可调整参数、应用结果和编程约定（包括那些能从在线附录中得到的包含在 util.concurrent 包中的线程类）。

标识（Identity）。绝大多数的工作者线程必须被“匿名地”处理。因为同一个工作者线程会被多个任务重用，而对 ThreadLocal 和线程相关的上下文控制技术（参见 § 2.3.2）的使用就显得笨拙了。为了处理这个，你需要知道所有的这些上下文数据，并且在执行每一个任务时，在需要的情况下通过某种方式将它重置（这包括了由运行时支持的类所维护的安全上下文信息）。但是，大多数轻量级可执行框架都避免依赖于任何线程相关的技术。

如果标识是线程中你想放弃的惟一属性，那么应通过重用已经存在的线程来执行多个 Runnable 任务，工作者线程惟一潜在的性能价值就是将线程的启动开销最小化，但是它也可能导致资源消耗。

队列（Queuing）。停留在队列中的 Runnable 任务并不运行。这也是在大多数工作者线程设计中带来性能优势的一个原因——如果每个操作都与一个线程相关，那么它就需要由 JVM 来独立地调度。但是作为一个结果，队列执行通常不能用于任务之间存在相互依赖的情况。如果一个正在运行的任务阻塞在那里，等待由另外一个等候在队列中的任务产生的条件，那么系统就相当于被冻结了。这里的解决方案包括：

- 有多少个同时执行的任务就使用多少个工作者线程。在这种情况下，Channel 不必执行任何的排队等候，所以你可以使用 SynchronousChannels（参见 § 3.4.1.4），即每一个非排队等候的通道都需要等候一段时间，并且反之亦然。这里，主体对象

java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

简单地将任务分发给工作者线程，而工作者线程立即启动并执行它们。为了能让这种方式更好地工作，工作者线程池应该是可以动态扩展的。

- 限制使用存在任务依赖的上下文是不可能的，例如在 HTTP 服务器中，每一个消息都是通过不相关的外部客户请求一个文件的形式发出的。当它们不能保证任务相互独立时，就需要由辅助对象创建实际的线程。
- 创建自定义队列，这些队列理解正在被工作者线程处理的特殊种类任务之间的依赖关系。例如，大多数处理代表事务的任务（参见 § 3.6）的池必须清楚事务的依赖关系。在 § 4.4.1 描述的轻量级并行框架依赖于特殊的排队策略，这些策略只应用在分而治之算法中创建的子任务里。

饱和性（Saturation）。随着请求比率的增长，一个工作者线程池最终会达到饱和。这时，线程池中的所有工作者线程会都在处理任务，并且使用这个池的主体对象不能再分配任务。对于这种情况，可能的处理方式包括：

- 增加池的大小。在很多应用程序中，边界是渐进地估计出来的。如果边界只是一个对基于展示一个在测试工作负荷下能在某个特殊平台上良好工作的价值的猜测，那么它就是可以增加的。可是，在某一点上，你必须接受下面的几个选项之一，除非在 JVM 资源耗尽，以至于不能构造一个新的线程的情况下，你能够忍受失败。
- 如果服务的特性允许，那么可以使用一个没有边界的缓冲通道并让请求堆积。但是，这种存在潜在风险的系统最终会因耗尽内存而失效，不过和由于创建线程而导致的资源耗尽相比，它可以坚持更长的时间。
- 建立一个后端压力通知方案来让客户端停止发送这么多的请求。如果最终的客户端是一个分布式系统中的一部分，那么它们可以转而使用其他的服务器作替代。
- 在达到饱和时丢弃新的请求。如果你知道客户端无论如何都会重试的话，这会是一个很好的选择。但是，除非重试是自动的，否则你需要加入回调、事件或通知给客户端来警告它们请求被丢弃了，这样才能保证它们会知道去重试（参见 § 4.3.1）。
- 通过丢弃已经排列在队列中但还没有运行的旧请求为新的请求腾出空间，或者甚至取消一个或多个正在执行的任务。这种在达到饱和时，对新的请求的优先级超过旧的请求的情况，在某些时候与使用模式吻合得很好。例如，在某些电信系统中，旧的无人处理的任务通常是客户端已经放弃或断开的请求。
- 阻塞直到某个线程可用。在操作者是可预测的、持续较短时间的情况下，这是一个很好的选择。这种情况下，你可以确定等待可以消除而不会有不可接受的延迟。
- 主体可以在自己的线程内直接运行任务，这经常也是最佳的默认选择。在这种情况下，主体实质上暂时地成为一个单线程。处理请求的行为限制了它能够接收新请求的比率，这样就能阻止进一步的本地故障。

线程管理（Thread management）。PlainWorkerPool 类在某种程度上是一种浪费，因为不管需不需要，它都在启动时创建所有的工作者线程，并且在没有人使用服务时，也让这些线程都无限制地存活下去。可以通过使用一个管理类来减轻这些问题，这个管理类需要支持以下特性：

java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

- ***延迟构造***：即只在没有空闲服务线程处理请求时，才激活一个新的线程。延迟构造允许用户提供一个合适的池的大小限制，通过它来避免当运行的线程比计算机的处理能力少的情况下发生的线程利用不足的问题。但是，当一个新的请求导致一个新线程被创建的情况会出现较长的等待时间，这样延迟构造也会带来一定的开销。通过在构造线程池时创建少量的“预热”线程，可以降低延迟构造的启动效应。
- ***空闲超时***：允许线程暂停等待工作并在超时时中止。如果线程池长时间没有被使用，将最终导致所有的工作者线程退出。当与延迟构造一起使用时，如果后来请求的比例提高，这些中止的线程就会被新的线程替代。

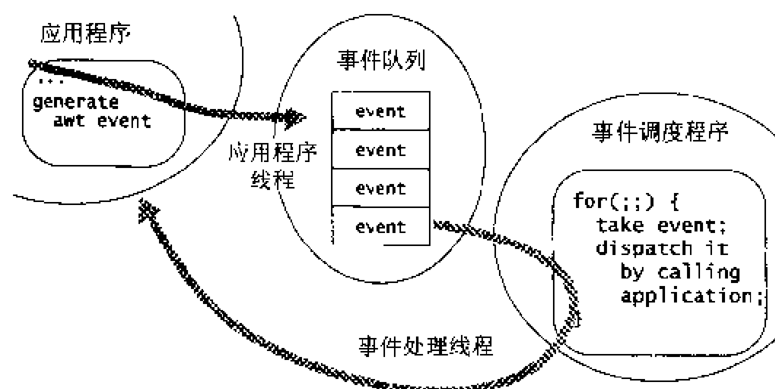
在资源紧张的应用程序中，你可能也将其他资源（诸如成套的可重用图形对象）与每一个工作者线程关联起来，这样就将资源池（参见 § 3.4.1.2）与线程池结合起来了。

取消（Cancellation）。需要区分任务的取消（参见 § 3.1.2）与执行任务的工作者线程的取消。一种区分方法是：

- 在被中断的情况下，允许当前的工作者线程死掉，但是如果工作队列非空或当一个新的任务到达时，如果需要的话，可用一个新的工作者线程替换它。
- 在工作者线程类中提供一个 shutdown 方法，它会使存在的工作者线程死掉并且不会创建额外的工作者线程。

另外，如果一个主体线程在派发任务的过程中被取消了，你就可能需要激发一些错误处理的手段。不是像在 PlainWorkerPool 中见到的那样把任务加到队列中，而是悄悄地处理掉 InterruptedException 异常。这虽然符合单向消息传递框架的最小要求，但大多数的应用程序都需要采取其他的补救措施。

4.1.4.2 事件队列



许多基于事件的框架（包括 java.awt 和 javax.swing 包支持的那些）依赖于其中一个工作者线程正确地操作一个无限制的队列的设计。在这个队列中包含着必须被发送给由应用程序定义的 ***listener*** 对象的 EventObject 对象的实例（而不像 Runnable 对象那样能够独立运行）。监听者与那些最初产生事件对象的对象经常是同一个对象。

使用操作在一个单一的事件队列的单线程与通常的工作者线程的设计相比，在用法上有所简化，但是这也带来了事件框架特性的一些局限性。

java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

- 队列的排序属性可以用来优化处理技术。例如，同一个屏幕区域内重复的重画事件在被从队列的前端选中并被工作者线程接收之前，自动事件过滤技术能够删除或组合这些重复的事件。
- 你可以要求某些对象上的所有方法操作只能通过向队列中发送消息的方式被激活，并且这些方法最终被一个单一的工作者线程执行。这会导致这些对象的线程限制（参见§2.3.2）的形态。如果很好地使用，这会消除在对这些对象的操作内对动态锁定技术的需要，从而提高性能。而且这也能降低应用程序的复杂度，因为它不需要再另外构造线程。

这是 Swing 的单线程原则的基础：所有对 Swing 对象的处理都必须通过事件处理线程来执行，除了只在很少的几个地方例外。虽然在 AWT 中没有说明，但是也遵循了这样的设计规范。

- 在事件的处理者没有完全构造好并准备好处理事件之前，事件不应该被启用。在其他基于线程的设计中，这条原则也同样有效（参见§2.2.7）。这也是一个更常见的导致错误的原因，因为在自己的构造函数内注册一个事件处理器或监听器并不像构造线程一样是一个明显的、过早启用并发执行的方式。
- 事件框架的用户永远都不要派发只有在对未来事件处理结束后才能解锁的、会阻塞通路的操作。在大多数事件框架中，在实现模式对象框时都会遇到这个问题，并且需要一个业界的解决方案。通过为交互组件设置一个“禁用”的状态，可以简单地获得局部的解决方案。在这种情况下，交互组件直到接收到一个重新启用的事件后才能被使用。这就通过禁止不希望的操作被激发而避免了事件队列的阻塞。
- 更进一步地，为了维持事件框架的响应性，操作根本就不应阻塞，而且也不应该执行耗时的操作。

这套设计原则会让事件框架设计比每事件一线程的设计带来更好的性能，并且会让不使用线程的开发者的编程工作更简单。但是，这种用法的局限对的确需要构造其他线程的程序带来了更大的冲击。例如，由于单线程规则，即使是对 GUI 组件最小的操作（诸如改变一个标签的文本）也必须通过发送 runnable 事件对象来执行，在事件对象中封装了通过事件操作线程执行的动作。

在 Swing 和 AWT 应用程序中，javax.swing.SwingUtilities.invokeLater 方法和 java.awt.EventQueue.invokeLater 方法能够在事件处理线程中执行与显示相关的命令。这些方法创建在从队列中取出时被执行的 runnable 事件对象。在线附录中包含了到 SwingWorker 工具类的链接，这个工具类在一定程度上为产生屏幕更新结果的线程应用了上述规则。

4.1.4.3 定时器（Timer）

在工作者线程设计中，Runnable 任务可能只是呆在队列中而没有运行，在某些应用程序中发生这种情况就会成为问题。但是在某些时候，如果希望操作被延迟，这又会成为一个有用的特点。

对工作者线程的使用既可以提高效率，也可以简化特意延迟或者周期执行的操作的用

java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

法——经过一定的延迟或者固定的间隔时间（例如每天中午），这些操作会在某个特定时间被触发。一个标准的定时程序既可以自动计算复杂的时间，也可以通过重用工作者线程来避免过度的线程构造。这种情况的主要损失是，如果一个工作者线程阻塞或者长时间处理一个任务，那么触发其他的任务可能比由基本的 JVM 创建并调度分离的线程会延迟更长的时间。

基于时间的后台进程可以通过在 § 4.1.4.1 描述的基本工作者线程设计的变体方式构造。例如，下面是依赖一个未被显示的优先级队列的类（它的形式可能类似于在 4.3.4 所展示的调度队列）的版本的主要部分，并且它被设置成只支持一个工作线程。

```
class TimerDaemon {                                    // Fragments

  static class TimerTask implements Comparable { // ...
    final Runnable command;
    final long execTime;       // time to run at
    public int compareTo(Object x) {
      long otherExecTime = ((TimerTask)(x)).execTime;
      return (execTime < otherExecTime) ? -1 :
                   (execTime == otherExecTime)? 0 : 1;
    }
  }

  // a heap or list with methods that preserve
  // ordering with respect to TimerTask.compareTo

  static class PriorityQueue {
    void put(TimerTask t);
    TimerTask least();
    void removeLeast();
    boolean isEmpty();
  }

  protected final PriorityQueue pq = new PriorityQueue();

  public synchronized void executeAfterDelay(Runnable r,long t){
    pq.put(new TimerTask(r, t + System.currentTimeMillis()));
    notifyAll();
  }
  public synchronized void executeAt(Runnable r, Date time) {
    pq.put(new TimerTask(r, time.getTime()));
    notifyAll();
  }

  // wait for and then return next task to run
  protected synchronized Runnable take()
   throws InterruptedException {
    for (;;) {
      while (pq.isEmpty())
        wait();
      TimerTask t = pq.least();
      long now = System.currentTimeMillis();
      long waitTime = now - t.execTime;
      if (waitTime <= 0) {
```


java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

```
          pq.removeLeast();
          return t.command;
        }
        else
          wait(waitTime);
      }
    }

    public TimerDaemon() { activate(); } // only one

    void activate() {
      // same as PlainWorkerThread except using above take method
    }
  }
```

在§3.7讨论的技术可以用在这里以提高等待和通知操作的效率。

通过在运行任务之前将任务重新排列的附加手段来扩展这个类，使它还能处理周期性的任务。然而，这也需要处理这样一个事实，就是周期性的操作实际上无法很精确地周期执行，部分原因是由于在指定的延迟时间里，定时等待不一定会被准确地唤醒。可以选择忽略时钟时间的延迟并根据时钟时间重新调度，或者忽略时钟，在当前任务开始之后的固定时间延迟之后重新调度执行下一个任务。在多媒体同步中，特别需要 Fancier scheme，参见进阶阅读§1.3.5。

另外，定时器后台程序[1]提供了取消延时或周期性操作的方法。一种方法是使用 executeAt 和其他调度方法，它们接收或返回一个适当的可以修改的 TimerTask 对象，这个 TimerTask 对象提供了一个由工作者线程设置状态位的 cancel 方法。

4.1.5 轮询和事件驱动的IO

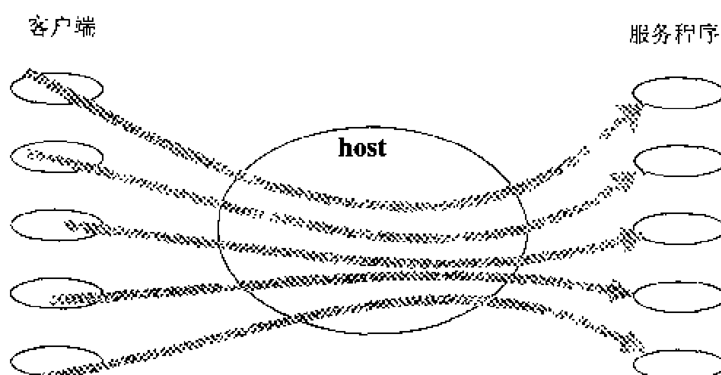
大多数工作者线程设计都依赖于阻塞通道（blocking channel），工作者线程等待在阻塞通道中要运行的输入命令。然而，在有些场景下，通过使用优化风格的循环重试可以提供更好的解决方案。在大多数情况下，它们都包括执行从 IO 流中接收到的消息中所分析出的命令。

在高负荷的与 IO 相关的系统中要实现低等待时间和高吞吐量是个很大的挑战。创建一个线程来执行基于 IO 的任务所花费的时间增加了延迟的时间，但是大多数运行时系统都进行了调优，一旦线程被创建，它们对通过 IO 流到达的新输入的响应会非常快。在输入时，可以比你通过其他技术实现的系统在更短的延迟时间内解锁。尤其是在基于套接字的 IO 环境中，通常要求每个 IO 会话需要一个线程的设计。在这种情况下，每一个依赖于从不同连接来的输入的会话使用（或重用）一个不同的线程。

然而，随着同时活动的连接数量的增长，（只）在这种情况下，别的方法就显得更有吸引力了。考虑一下这个例子，一个多用户的游戏服务程序或者一个交易服务程序，它具备以下特点：

1　在本书写作的过程中，即将发布的 SDK 中计划支持一个类似的类。

java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！



- 成千上万个同时发出的套接字连接，以一个稳定的比率加入和离开系统。例如，人们启动或者结束一个游戏。
- 在任意给定的时间内，任意指定的套接字上都保持相对较低的输入比例，但是，累计所有的连接，总的 IO 比率可能非常高。
- 至少与某些输入相关的大量计算，例如在游戏中可能导致全局状态改变的计算。

在大型主机系统中，这类问题有时可以通过创建一个特殊目的的前端机器来处理，由这个机器将所有的输入多路传输到一个数据流中，然后通过主服务程序来处理这个数据流。主服务程序通常是多线程的，但是由于不需要同时处理那么多的客户端，主服务程序结构得到了简化并且更高效。

轮询和事件驱动族的设计能无需专用的前台就能解决这些问题。虽然（在写本书时）它们还没有被 java.io 和 java.net 包中的类明确地支持，但是它已经提供了足够的机制，使我们能够构造获得更好的性能设计（这些设计类似于那些在其他系统和语言中使用套接字的 select 和 poll 操作的设计）。我们将会用套接字的输入来说明它，但是这个方法也可以应用到输出、文件以及更多的外来设备的 IO 上，如传感器等等。

4.1.5.1 事件驱动的任务

许多基于 IO 的任务最初是以基于会话的风格编写的（参见 § 2.3.1），程序持续地从套接字中取回命令并处理它们。例如：

```
class SessionTask implements Runnable {  // Generic code sketch
  protected final Socket socket;
  protected final InputStream input;
  SessionTask(Socket s) throws IOException {
    socket = s; input = socket.getInputStream();
  }

  public void run() {            // Normally run in a new thread
    byte[] commandBuffer = new byte[BUFFSIZE];
    try {
      for (;;) {
        int bytes = input.read(commandBuffer, 0, BUFFSIZE);
        if (bytes != BUFFSIZE) break;
        processCommand(commandBuffer, bytes);
      }
    }
```


java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

```
        catch (IOException ex) {
          cleanup();
        }
        finally {
          try { input.close(); socket.close(); }
          catch(IOException ignore) {}
        }
      }
    }
```

为了处理多个会话，而不用使用很多的线程，任务首先需要被重构为事件驱动的风格，在这种情况下，事件代表着IO的可用性。在这种风格下，会话可能由许多任务的执行组成，而这些任务又都是由事件激发的，当输入变成可用时，每一个任务都将被调用。事件驱动的IO任务在形式上类似于GUI中的事件处理器。一个基于会话的设计可以通过以下方式转换为一个事件驱动的设计：

- 隔离基本命令的功能，将其放在重新设计的任务的run方法中，每次先读取命令，然后再执行相关操作。
- 定义run方法，无论什么时候，当输入可读的时候（或发生IO异常的情况下），它都可以被重复地触发。
- 手动维护完整的状态，从而在每个会话完成之后，因为输入已经被耗尽或连接已经被关闭，而使事件操作不再被触发。

例如：

```
class IOEventTask implements Runnable {  // Generic code sketch
  protected final Socket socket;
  protected final InputStream input;
  protected volatile boolean done = false; // latches true

  IOEventTask(Socket s) throws IOException {
    socket = s; input = socket.getInputStream();
  }

  public void run() { // trigger only when input available
    if (done) return;

    byte[] commandBuffer = new byte[BUFFSIZE];
    try {
      int bytes = input.read(commandBuffer, 0, BUFFSIZE);
      if (bytes != BUFFSIZE) done = true;
      else processCommand(commandBuffer, bytes);
    }
    catch (IOException ex) {
      cleanup();
      done = true;
    }
    finally {
      if (!done) return;
      try { input.close(); socket.close(); }
      catch(IOException ignore) {}
    }
  }
```


java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

```
    // Accessor methods needed by triggering agent:
    boolean done()      { return done; }
    InputStream input() { return input; }
  }
```

4.1.5.2　触发

当驱动任务的事件相对比较少时，少量的工作者线程就可以处理大量的任务。最简单的情况发生在工作者线程的数量只有一个的情况下。在这种情况下，工作者线程重复地轮询一系列打开的套接字连接，检查是否有可用的输入（通过 InputStream.available 方法）或是否遇到其他与 IO 相关的状态改变。如果有，那么工作者线程就执行相关的 run 方法。

这种工作者线程的风格与在 § 4.1.4.1 介绍的有所不同，它并不是从一个阻塞的队列中找出任务并盲目地运行任务，工作者线程必须通过重复地检查注册任务的列表来判断是否有任务可以被执行。它只在任务已经完成的情况下才把任务从任务列表中清除。

一种常见的形式如下所示：

```
class PollingWorker implements Runnable {        // Incomplete
  private List tasks = ...;
  private long sleepTime = ...;

  void register(IOEventTask t)   { tasks.add(t); }
  void deregister(IOEventTask t) { tasks.remove(t); }

  public void run() {
    try {
      for (;;) {
        for (Iterator it = tasks.iterator(); it.hasNext();) {
          IOEventTask t = (IOEventTask)(it.next());
          if (t.done())
            deregister(t);
          else {
            boolean trigger;
            try {
              trigger = t.input().available() > 0;
            }
            catch (IOException ex) {
              trigger = true; // trigger if exception on check
            }
            if (trigger)
              t.run();
          }
        }
        Thread.sleep(sleepTime); // pause between sweeps
      }
    }
    catch (InterruptedException ie) {}
  }
}
```

下面列出了一些设计上需要关注的问题：

- 轮询在本质上要依赖一个忙等待的循环（参见 § 3.2.6），而这个循环在本质上是一

java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

种浪费行为（但是有些时候仍然比上下文切换要经济一些）。要处理好这个问题，需要依靠经验来决策怎样插入 sleep、yield 或其他操作，从而在节省 CPU 时间和维持一个可接受的平均响应延迟之间达到某种平衡。

- 性能对维护注册任务列表的底层数据结构的特性是非常敏感的。如果新任务有规律地来往，那么任务列表可以相当频繁地改变。在这种情况下，诸如复制写（copy-on-write）的方案（参见 § 2.4.4）就不能很好地工作了。但是会有很多原因要求我们尽可能廉价地遍历列表。一种方法是为遍历维护一个缓存的列表，并只在每一次扫描列表结束时才更新它（如果需要的话）。
- 事件驱动的任务应该只在有足够数据来执行它们相关的操作时才能被激发。但是，在许多应用程序中（例如那些使用自由形态的、基于字符串的命令的应用程序），并不能预先得知激发任务的最小的数据量。实际上（如同在这里举例说明的），它经常只是检查到至少有一个字节可读就开始执行。它利用了基于套接字的客户端发送数据包（packet）的事实——通常情况下每个数据包中都包含了一个完整的命令。但是，在命令没有完全到达的情况下，工作者线程只能停止，这样，除非加入缓冲方案，否则就会增加其他任务的延迟。
- 如果某些输入会带来耗时计算或 IO 阻塞，那么单一的工作者线程是不能接受的。一种解决方案是把这样的耗时计算放在一个新线程里执行或者由线程池里的一个单独的线程执行。不过，有时使用多个轮询的工作者线程是一种更有效的方式；如果线程足够，平均算来总是会有一个轮询线程在等待输入。
- 对多个轮询工作者线程的使用需要额外的协调，以保证在同一时间内不会有两个工作者线程同时去执行同一个任务，导致彼此访问任务列表时相互阻碍。一种方法就是把任务类设置为“忙”状态，例如，通过 testAndSet（参见 § 3.5.1.4）。

从给出的这些关注点和与设计决策相关的上下文依赖中可以看出，大多数框架都需要通过定制以适合特殊应用程序的需要就不令人惊奇了。不过，可以从在线附录中得到 util.concurrent 包中的一些实用工具，它们可以帮助你构建一些标准的解决方案。

4.1.6 进阶阅读

大多数关于消息、格式、传输等的细节，在实际使用过程中都与特定的开发包和系统相关，所以最好的资源就是它们自带的手册和文档。

在分布式系统中，关于消息传递的讨论可以在 § 1.2.5 列出的资源中找到。这几个开发包和框架中的任意一个都可以扩展在这里讨论的技术，并应用到分布式环境中。例如，这些设计（也包括在 § 4.2 和本书其他地方介绍的大多数设计）中的大多数能够在调整后用于 JavaSpaces。反过来，许多分布式的消息传递技术也可以简化后用于并发、非分布式的设置中。

下面这本书讨论了使用 JavaSpaces 的设计和实现：

Freeman, Eric, Susan Hupfer 和 Ken Arnold.《JavaSpaces™: Principles, Patterns, and Practice》, Addison-Wesley, 1999.

对于不同的方法，参见 Aleph、JMS 和 Ninja 开发包中的例子，可以通过在线附录中提

java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

供的链接访问这些包。许多商业的分布式系统都基于 CORBA 和相关的框架，它们也包括对单向消息传递的一些支持。参见：

Henning, Michi 和 Steve Vinoski.《Advanced CORBA Programming with C++》, Addison-Wesley, 1999.

Pope 和 Alan.《The CORBA Reference Guide》, Addison-Wesley, 1998.

某些类似于本书所提到的系统级的单向消息策略在下面列出的书中有所描述：

Langendoen, Koen, Raoul Bhoedjang 和 Henri Bal. "Models for Asynchronous Message Handling",《IEEE Concurrency》, April-June 1997.

对于应用程序编程而言，与基于线程的框架相比，单队列和单线程的事件框架是一种更好的选择，这个主题可以在下文中看到：

Ousterhout 和 John. "Why Threads Are a Bad Idea (For Most Purposes) ",《USENIX Technical Conference》, 1996.

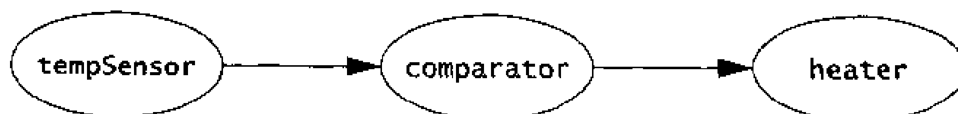
4.2 编写单向消息

许多跨进程和分布式的设计都包含了交换单向消息的对象组（参见 § 1.2 和 § 4.5）。类似的技术也可以应用到单个并发程序中。事实上，如同在 § 4.1 所讨论的那样，同分布式系统相比，在并发编程中存在更多可用的设计选项。不必限制消息使用的格式，例如基于套接字的命令的形式。并发编程也可以使用更轻量级的选择，包括直接调用和基于事件通信的方式。

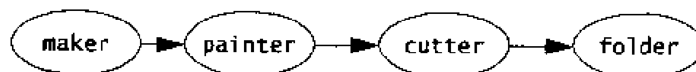
但是，太多的选择同样也会带来产生混乱、难于理解的设计问题。这一节将介绍一些简单的程序级（或子系统级的）的构造技术，它会帮助你产生行为良好、易于理解并容易扩展的设计。

流网络 ***(Flow network)***。一系列传递单向消息和/或对象的对象集合，这些消息沿着从消息源到消息接收者的路径传递信息。在支持一个或多个连接步骤或连接阶段（stage）的任意类型的系统或子系统中，都可能存在流模式。在这种模式中，每一个阶段都扮演着***生产者***或***消费者*** 的角色。主要的分类包括：

控制系统 ***(Control system)***。外部传感器的输入最终会导致控制系统产生特殊的效应输出。诸如航空控制系统这样的应用系统，包含了许多种类的输入与输出。作为一个简单点的例子，考虑一下恒温加热系统的主要控制过程：

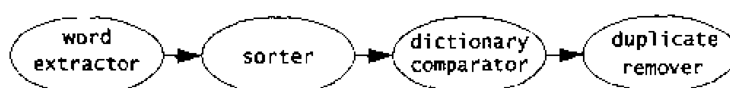


组装系统 ***(Assembly systems)***。在最后为了某种目的使用之前，新创建的对象要经历一系列的改变和（或）与其他新对象进行集成；例如，一个纸盒的组装线：

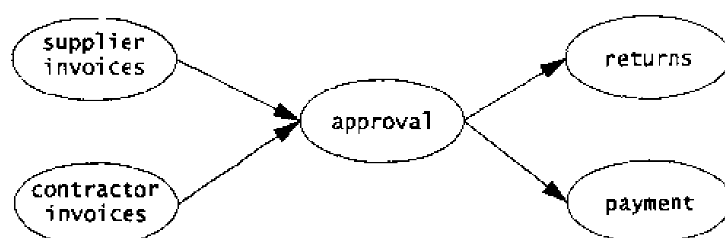


java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

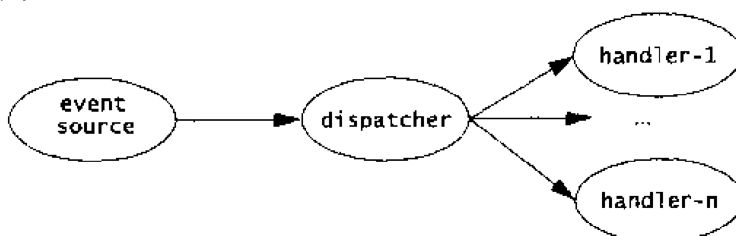
数据流系统*（*Dataflow system*）**。每个阶段转换或以另外的方式处理数据。例如，在流水线的多媒体系统中，需要经过多个步骤处理音频和（或）视频数据。在发布-订阅*（*publish-subscribe*）**系统中，可能有许多数据源发送消息给许多可能的消费者。在 Unix 的管道过滤脚本程序中，每个阶段发送字符数据，如同在下面所示的一个简单的拼写检查程序中所作的那样：



***工作流系统*（*Workflow system*）**。每一个阶段代表一个需要根据一些业务策略或其他需求来执行的操作；例如，一个简单的支付系统：



***事件系统*（*Event system*）**。阶段发送并最终执行与代表着消息、用户输入或模拟的物理现象的对象相关的代码。许多事件系统的早期都采用下面的形式：



4.2.1 组合（Composition）

开发流网络需要关注两方面内容：被传递的数据的设计与负责传递工作的阶段的设计。

4.2.1.1 消息表现（Representation）

流网络传递表现组件——一系列代表着流所涉及的事情的对象或值。在介绍性的例子里，温度、纸板片、单词、发票，以及事件都是在连接阶段间传送的值和对象的基本类型。这些组件常常是一些有趣的对象，它们可以自己执行服务、与其他对象通信，等等。但是当把它们看作是流的原始材料时，它们仅仅被当作是提供数据或信息而不是行为的被动代表。

然而在流系统的整个设计中，它们扮演着相似的角色，对表现类型的不同分类会影响到系统其他部分的细节：

- 代表着世界状态（例如，将读到的温度值作为标量或不可改变的抽象数据类型进行维护）的信息类型与大多数其他类型不同，区别在于在需要的情况下，这种类型往往可以接收重用的旧值或当前的最佳估计值。在本质上，生产者会提供无穷无尽的这样的值。

java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

- 事件指示器通常最多可以使用一次，虽然在被使用之前它们可以被传递很多次。
- 可变化的资源类型（例如纸板）可以从一个阶段传输（参见 § 2.3.4）到下一个阶段，从而保证在任何给定的时间内，每个对象至多在一个阶段被操作。
- 作为选择，如果可变代表对象的标识并不重要，那么就可以根据需要在不同阶段之间拷贝这些对象。基于拷贝的方法更常用于分布式流网络中，在分布式流网络中，并不能简单地通过引用在不同阶段间传递对象的所有权。
- 为了控制的目的可以使用人造的数据类型。例如，可以把一个特殊的 null 标记作为一个终结符，通过它触发取消或关闭操作。类似地，可以发送一个特殊的 keepalive 信号通知一个阶段另一个阶段仍然存在。另外，一套独特的边带（sideband）控制方法能够用于很多阶段，边带控制可以将阶段设置到会影响主要数据处理过程的不同的模式。例如，一个恒温器比较器中可能有一个单独的控制器用来改变它的阈值。

4.2.1.2 阶段（Stage）

表现良好的流网络中的阶段都遵守一套约束，这些约束会让你想起在电子电路设计中所见到的那些约束。下面给出了一套保守的、会产生少量基本阶段类型的组合规则的集合：

方向性（Directionality）。流维护一个从发生源到接收点的单一的方向性。从消费者到生产者之间没有循环或向后的分支。这就导致了一个信息或对象流的有向非循环图（DAG）。

互操作性（Interoperability）。通常，通过保证一套接口的一致性可以保证组件之间方法和消息的格式是标准的。

连接性（Connectivity）。阶段维护固定的连接性：消费者可能只从已知的生产者那里接收消息，反之亦然。因此，例如，虽然一个 Web 服务可能有任意数量的匿名客户，但是可以把一个给定的 TemperatureComparator 对象设计成只能从一个指定的 TemperatureSensor 对象那里接收温度变化的消息。

通常，按照从生产者到消费者或相反的方向维护对象的直接引用的方式，或者通过让它们共享访问一个 Channel 对象的方式来安排连接性；另一方面，整个网络可以在一定的约束下使用黑板、多点广播通道，或 JavaSpaces，生产者可以向其中添加带有目标消费者标记的消息。（参见 § 4.1.6）。

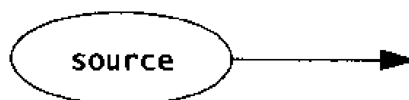
传输协议（Transfer protocol）。每个消息传输信息或对象。一旦一个阶段已经传输了一个可变的对象，它就永远不会再操纵那个对象。在必要的时候，如果一个阶段传输出去的元素仍然不能被其他阶段所接受，则可以放入特殊的缓存阶段以保存这些元素。

典型情况下，传输协议依赖于在 § 2.3.4 描述的基本的放入（put）和取出（take）操作。当所有的消息都使用基于放入（put-based）的传输时，这种网络通常被称为推动流（push flow）；当消息使用基于取出（take-based）的传输时，该网络通常被标注为拉动流（pull flow）；当消息所使用的通道既支持放入又支持取出（也可能是交换）时，网络可以采用各种混合形式。

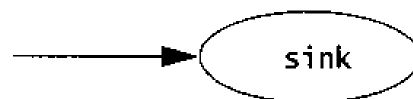
线程（Thread）。只要每一个（潜在地）从给定生产者到给定消费者的并发活动连接使用不同的线程或基于线程的消息发送结构，阶段就可以通过使用在 § 4.1 中描述的任何一种模式来实现单向消息传递。

java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

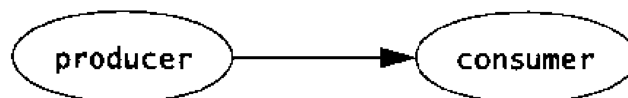
发送源没有前级。



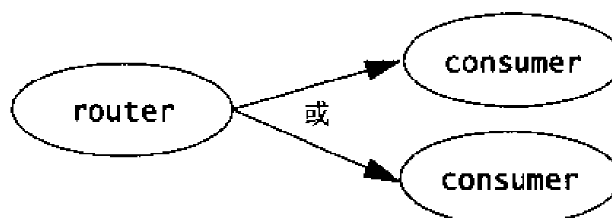
接收者没有后级。



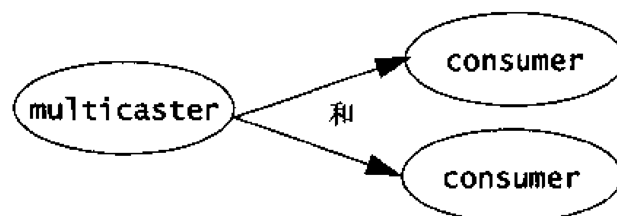
线性阶段至多有一个前级和一个后级。



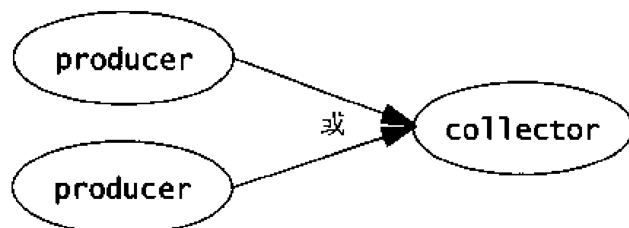
路由器给它的其中一个后级发送消息。



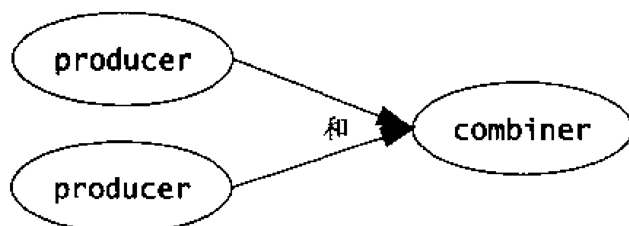
广播给它的所有后级发送消息。



收集器每次从它的一个前级中接收消息。

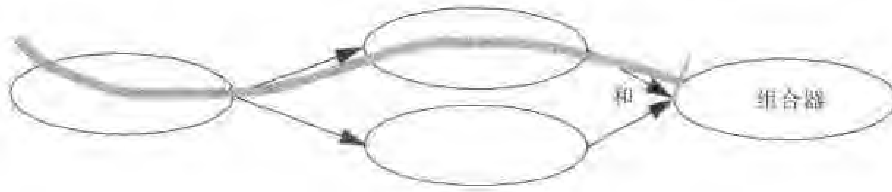


组合器从它的所有前级那里接收消息。



通常情况下很少使用将生产者发送给消费者的每个消息和每个消息流在一个单独的线程中处理来满足需求。但可以在需要的时候通过进一步规定连接性规则来使用线程实现。在推系统（push-based system）中，大多数发送源内在地使用线程。另外，如果一个推动阶段具有多个最终可能会到达一个组合器阶段的后级阶段，那么这个推动阶段必须在独立的线程中发送不同的消息。否则，如果某个线程在一个组合点上被阻塞，那么组合器就有可能永远也看不到需要用来解锁这个线程的其他输入。

java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！



反过来，大多数拉系统中的接收器在其内部都使用了基于线程的消息结构，比如上图描述的分开/汇聚连接，不过，其触发方向是反向的。

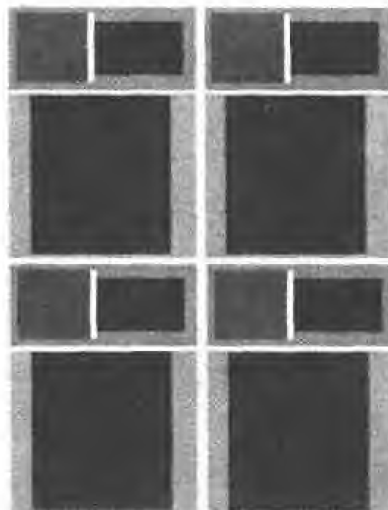
可以通过很多种方式放宽这些规则。事实上，你可以选择任何一套你喜欢的组合规则。但是列出的这些约束适合消除一大类安全和活跃性的问题，同时也能够满足普通的重用性和性能的目标：单向流避免了死锁；连接性管理避免了在不同流之间不必要的交叉；传输协议避免了由于不当的共享而导致的安全问题，而且不需要广泛地动态同步；接口的一致性保证了类型安全，同时仍然支持组件之间的互操作性。

4.2.1.3 脚本（Scripting）

选用标准的组合规则集使我们可以构造能够安排阶段协同操作的高级工具，而同时不必强迫使用集中化的动态同步控制。可以将流网络的组合看作脚本的一种形式，如同这个单词（*scripting*）的通常含义——能够将已有的对象类型的实例粘和到一起的半自动化的编程方式。这是一种与某些语言相关的编程方法，这些语言包括 JavaScript、Visual Basic、Unix shells，以及 FlowMark（一种工作流工具）。对脚本工具的开发，或对已有工具的集成，在构建基于流的系统中是一个可选的步骤。

这种架构类似于 GUI 构建器的架构，GUI 构建器由一套基本的窗口部件、包装器和布局管理器、实例化某种 GUI 的代码，以及用于集成这些的可视化脚本编辑器组成。或者，通过直接操作工具来编写流脚本也是可能的，例如，一旦某个组件通过拖放（dragged-and-dropped）与其他组件相连，组件间立刻就可以进行通信了。

4.2.2 装配线（Assembly Line）



java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

这一节剩下的部分将通过一个装配线 Applet 的例子说明流系统的设计与实现，这个例子会创建一系列可能会让你隐约想起艺术大师皮特·蒙德里安（Piet Mondrian）和马克·罗思科（Mark Rothko）画风的“图画”。在这里只给出了一些主要的类，其中还有些还包含了一些没有实现的方法说明。你可以在在线附录上找到完整的代码，那里还包含了其他应用级的基于流的系统的例子。

4.2.2.1 消息表现（Representation）

为了开始这个例子，我们需要定义一些基本的表现类型。在这个系统中，所有的元素都可以定义为抽象类 Box 的子类，在 Box 类中，每一个 Box 具有一个颜色（color）和一个大小（size）属性，Box 类在被请求时能够显示自己，和深度（duplicate）克隆自己。Box 类默认实现了颜色（color）机制，而其他部分则被设置为抽象的，需要在不同的子类中不同地定义。

```
abstract class Box {
  protected Color color = Color.white;

  public synchronized Color getColor()        { return color; }
  public synchronized void  setColor(Color c) { color = c; }
  public abstract java.awt.Dimension size();
  public abstract Box duplicate();                 // clone
  public abstract void show(Graphics g, Point origin);// display
}
```

这个例子的主题就是从产生简单的基本盒子的代码开始，然后把它们推向不同的阶段：绘制、连接、交换，最终形成图画。基本盒子（BasicBox）只是未加工的原材料：

```
class BasicBox extends Box {
  protected final Dimension size;

  public BasicBox(int xdim, int ydim) {
    size = new Dimension(xdim, ydim);
  }

  public synchronized Dimension size() { return size; }

  public void show(Graphics g, Point origin) {
    g.setColor(getColor());
    g.fillRect(origin.x, origin.y, size.width, size.height);
  }

  public synchronized Box duplicate() {
    Box p =  new BasicBox(size.width, size.height);
    p.setColor(getColor());
    return p;
  }
}
```

可以通过下面的方式制成两种更奇特的盒子：把两个已存在的盒子边靠边地连接在一起，通过一个用线做成的边框围住它们，从而形成一个新的盒子。连接在一起的盒子也可以互相交换。所有这些排列方式都可以是水平的也可以是垂直的。两种由此产生的类可以通过

java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

继承 JoinedPair 类共享一些通用的代码：

```
abstract class JoinedPair extends Box {
  protected Box fst; // one of the boxes
  protected Box snd; // the other one

  protected JoinedPair(Box a, Box b) {
    fst = a;
    snd = b;
  }

  public synchronized void flip() { // swap fst/snd
    Box tmp = fst; fst = snd; snd = tmp;
  }

  //  other internal helper methods
}
class HorizontallyJoinedPair extends JoinedPair {

  public HorizontallyJoinedPair(Box l, Box r) {
    super(l, r);
  }

  public synchronized Box duplicate() {
    HorizontallyJoinedPair p =
      new HorizontallyJoinedPair(fst.duplicate(),
                                 snd.duplicate());
    p.setColor(getColor());
    return p;
  }

  // ... other implementations of abstract Box methods
}

class VerticallyJoinedPair extends JoinedPair {
  // similar
}
```

最后一种盒子将一个 Box 包装在一个边框内：

```
class WrappedBox extends Box {
  protected Dimension wrapperSize;
  protected Box inner;

  public WrappedBox(Box innerBox, Dimension size) {
    inner = innerBox;
    wrapperSize = size;
  }

  // ... other implementations of abstract Box methods
}
```

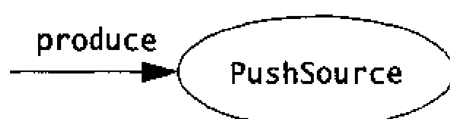
4.2.2.2 接口（Interface）

再往下，我们希望知道怎样才能够将各个阶段连贯起来，那么将接口标准化就是非常值得做的了。我们希望能够将任意阶段与任意其他有意义的阶段相连，所以要求主要方法具有

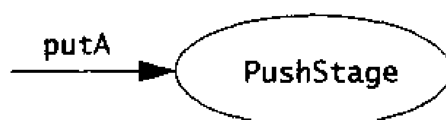
java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

通用的名字。

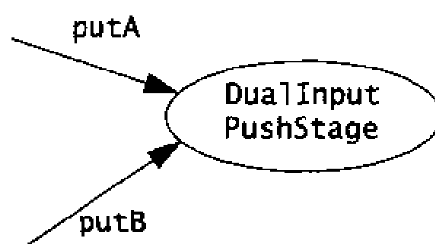
由于我们正在做的是基于推动的单向流，所以这些接口主要描述放入风格（put-style）的方法。事实上，除了它会在两个输入阶段的情况下工作得不好外，我们可以只是把它命名为 put。例如，一个 VerticalJoiner 需要两个放入方法，一个供放入顶端的盒子使用，另一个供放入底部的盒子使用。我们可以通过设计一个交替地把输入作为顶端或底部的连接器（Joiner）来避免这种设计，但是这会导致更难于控制。作为替代，我们使用在某种程度上有些笨拙，但是却易于扩展的名字，如 putA、putB 等等：




```
interface PushSource {
  void produce();
}
```




```
interface PushStage {
  void putA(Box p);
}
```

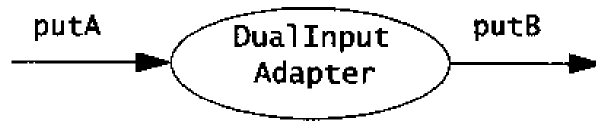



```
interface DualInputPushStage extends PushStage {
  void putB(Box p);
}
```

4.2.2.3 适配器

我们可以通过定义一个简单的适配器类（Adapter），使 DualInputPushStage 的“B”通道对其他阶段完全透明。在这种情况下，当适配器类接收到 putA 方法后，将它传递给原定的接收者的 putB 方法。通过这种方式，大多数阶段都可以构建成调用 putA 方法，而不用知道或不关心盒子被传递给某个后继者的 B 通道中：

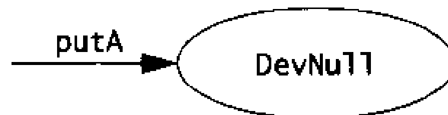
java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！




```
class DualInputAdapter implements PushStage {
  protected final DualInputPushStage stage;

  public DualInputAdapter(DualInputPushStage s) { stage = s; }

  public void putA(Box p) { stage.putB(p); }

}
```

4.2.2.4 接收器（Sink）

接收器没有任何后级。最简单的接收器甚至不处理它自己的输入，并以此作为丢弃消息的方式。在 Unix 管道和过滤器的思想中，我们可以把它称作：




```
class DevNull implements PushStage {
  public void putA(Box p) { }
}
```

更有趣的接收器需要更有趣的代码。例如，在本节开始时展示的用于产生图像的 applet 中，applet 子类被定义为实现 PushStage 接口。通过显示组装的对象，它成为最终的接收器。

4.2.2.5 连接（Connection）

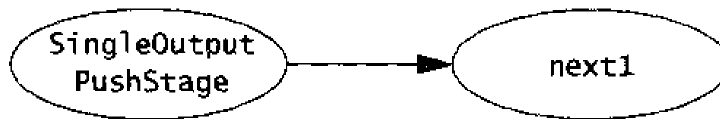
接口可以使不同阶段的方法名标准化，但是必须在每个阶段对象上使用某些类型的实例变量来维护与后级的连接关系。除了像 DevNull 这样的接收器，每一个阶段至少有一个后继者。有几种实现的选项，包括：

- 每个对象维护一个容纳所有其后级的集合对象。
- 使用一个主连接注册表，每一个阶段通过与注册表交互找到它的后级。
- 创建最小表示：为确定只有一个后继的阶段定义一个基类，为确定只有两个后继的阶段定义另一个基类。

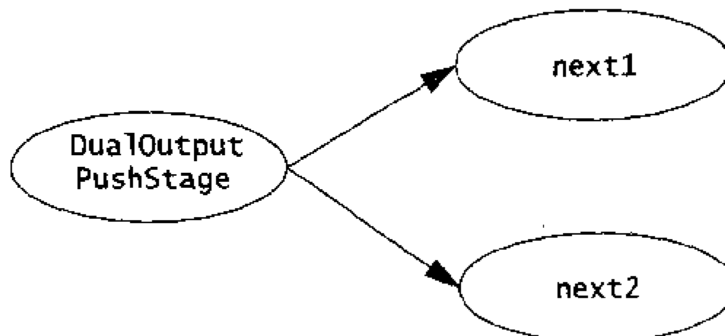
第三种方法是最简单的并能够在这里很好地工作［事实上，它也一直是一个有效的选项，具有三个或更多输出的阶段可以通过把它的输出联结成只有两个的方式创建。当然，如果大多数阶段都具有大量和（或）可变数量的后继器，你就可能不会这么做了］。

这会导致支持一个或两个连接并具有一个或两个相关连接方法的基类，并需要使用有些笨拙的后缀命名约定（如 attach1 和 attach2）来命名这些方法。因为连接是动态分配的，所以只能在同步的方式下访问它们。

java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！



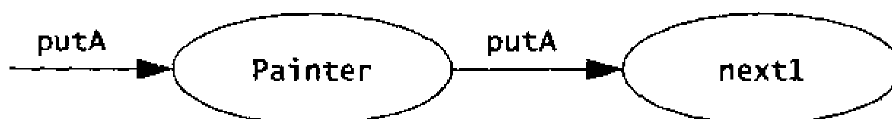

```
class SingleOutputPushStage {
  private PushStage next1 = null;
  protected synchronized PushStage next1() { return next1; }
  public synchronized void attach1(PushStage s) { next1 = s; }
}
```




```
class DualOutputPushStage extends SingleOutputPushStage {
  private PushStage next2 = null;
  protected synchronized PushStage next2() { return next2; }
  public synchronized void attach2(PushStage s) { next2 = s; }
}
```

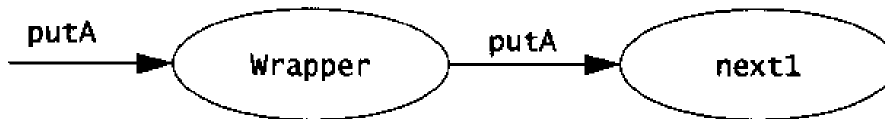
4.2.2.6　线性阶段（Linear stage）

现在，我们就可以构建继承任意一种基类，同时实现任意一种标准接口的类。最简单的变换阶段是线性、单输入/单输出的阶段。Painter 类、Wrapper 类和 Flipper 类仅仅是如下所示的情况：

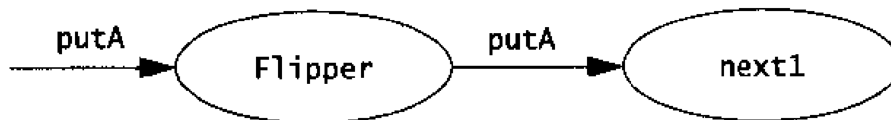



```
class Painter extends SingleOutputPushStage
              implements PushStage {
  protected final Color color; // the color to paint boxes

  public Painter(Color c) { color = c; }

  public void putA(Box p) {
    p.setColor(color);
    next1().putA(p);
  }
}
```


java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！




```
class Wrapper extends SingleOutputPushStage
                implements PushStage {
  protected final int thickness;

  public Wrapper(int t) { thickness = t; }

  public void putA(Box p) {
    Dimension d = new Dimension(thickness, thickness);
    next1().putA(new WrappedBox(p, d));
  }
}
```

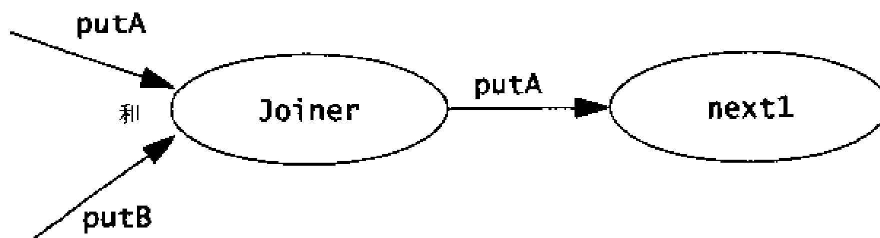



```
class Flipper extends SingleOutputPushStage
                implements PushStage {
  public void putA(Box p) {
    if (p instanceof JoinedPair)
      ((JoinedPair) p).flip();
    next1().putA(p);
  }
}
```

Painter 和 Wrapper 阶段可以应用于任意种类的盒子中。但是 Flipper 只对 JoinedPair 有意义：如果一个 Flipper 接收到的不是 JoinedPair 的对象，它只是把该对象传递出去，而不做任何处理。在一个更“强类型”的版本中，我们可能会通过把它们送到 DevNull 的方式，将不是 JoinedPair 类型的盒子去掉。

4.2.2.7 组合器（Combiner）

我们有两种类型的组合器：水平的和垂直的 Joiner。与消息表现类相似，这些类具有足够多的共性以至于可以根据它们抽取出一个超类。在能够把从 putA 和 putB 中得到的元素组合起来之前，Joiner 阶段会阻塞进一步的输入。这可以通过保护机制实现，这种机制暂停从 putA 中接收额外的内容，直到已经存在的那些与从 putB 中得到的组成一对为止，反之亦然：



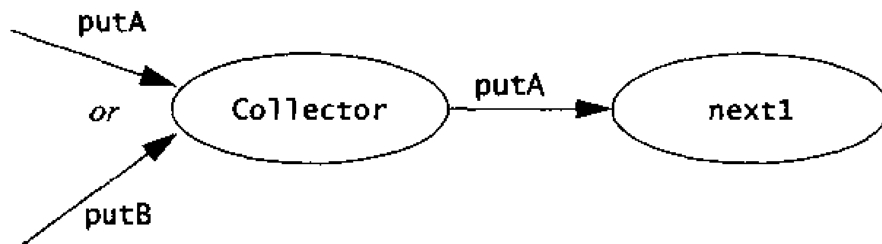
java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

```
abstract class Joiner extends SingleOutputPushStage
                      implements DualInputPushStage {
  protected Box a = null;  // incoming from putA
  protected Box b = null;  // incoming from putB

  protected abstract Box join(Box p, Box q);

  protected synchronized Box joinFromA(Box p) {
    while (a != null)            // wait until last consumed
      try { wait(); }
      catch (InterruptedException e) { return null; }
    a = p;
    return tryJoin();
  }
  protected synchronized Box joinFromB(Box p) { // symmetrical
    while (b != null)
      try { wait(); }
      catch (InterruptedException ie) { return null; }
    b = p;
    return tryJoin();
  }

  protected synchronized Box tryJoin() {
    if (a == null || b == null) return null; // cannot join
    Box joined = join(a, b);               // make combined box
    a = b = null;                          // forget old boxes
    notifyAll();                           // allow new puts
    return joined;
  }

  public void putA(Box p) {
    Box j = joinFromA(p);
    if (j != null) next1().putA(j);
  }

  public void putB(Box p) {
    Box j = joinFromB(p);
    if (j != null) next1().putA(j);
  }
}
class HorizontalJoiner extends Joiner {
  protected Box join(Box p, Box q) {
    return new HorizontallyJoinedPair(p, q);
  }
}
class VerticalJoiner extends Joiner {
  protected Box join(Box p, Box q) {
    return new VerticallyJoinedPair(p, q);
  }
}
```

4.2.2.8 收集器（Collector）

Collector 可以在任意一个通道上接收消息，并把它们转发给一个单一的后继器：

java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！



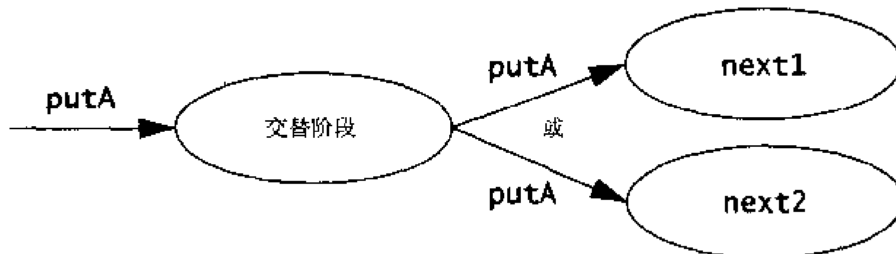

```
class Collector extends SingleOutputPushStage
                    implements DualInputPushStage {
  public void putA(Box p) { next1().putA(p);}
  public void putB(Box p) { next1().putA(p); }
}
```

如果我们为了某些原因而需要在这里强加一个瓶颈，则可以定义收集器的另外一种形式。在这种形式里，这些方法都被声明成 synchronized 的。这也可以用于保证对一个给定的收集器，在任意时间内至多有一个活动在进行。

4.2.2.9　双输出阶段（Dual output stage）

我们的多输出阶段将产生线程，或使用在 § 4.1 讨论的其他选项之一来驱动它们的每一个（它并不关心是哪一个）输出。当元素最终被传递给 Combiner 阶段时（在这里，指的是 Joiner），这可以维持系统的活跃性。为了说明的简单性，下面的类创建了新的 Thread。另外，我们可以通过建立一个简单的工作者线程池来处理这些消息：

交替阶段（Alternator）交替地把输入传递给它的不同后级：

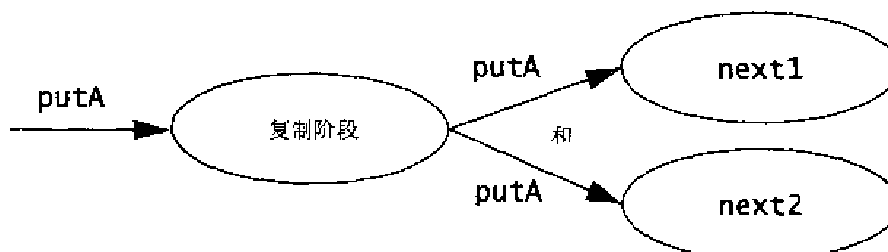



```
class Alternator extends DualOutputPushStage
                      implements PushStage {
  protected boolean outTo2 = false; // control alternation

  protected synchronized boolean testAndInvert() {
    boolean b = outTo2;
    outTo2 = !outTo2;
    return b;
  }

  public  void putA(final Box p) {
    if (testAndInvert())
      next1().putA(p);
    else {
      new Thread(new Runnable() {
        public void run() { next2().putA(p); }
```


java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

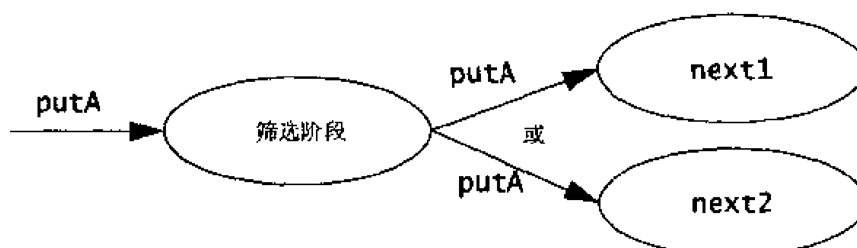
```
            }).start();
        }
    }
}
```

复制阶段（Cloner）把同一个输入元素广播给它的两个后级：




```
class Cloner extends DualOutputPushStage
                   implements PushStage {

  public void putA(Box p) {
    final Box p2 = p.duplicate();
    next1().putA(p);
    new Thread(new Runnable() {
      public void run() { next2().putA(p2); }
    }).start();
  }

}
```

筛选阶段（Screener）是这样一个阶段，它把所有符合某些条件的输入都定向到一个通道，而把其他的输入送到另外的通道。



通过将 BoxPredicate 封装为一个接口用以检查并实现它，我们可以构建一个通用的 Screener，例如，一个保证盒子与给定边界相适合（这里是方形的边界）的类。Screener 自己接收一个 BoxPredicate 并用它来控制输出：

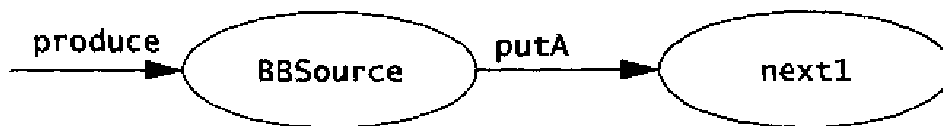
```
interface BoxPredicate {
  boolean test(Box p);
}

class MaxSizePredicate implements BoxPredicate {

  protected final int max; // max size to let through

  public MaxSizePredicate(int maximum) { max = maximum; }
```


java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

```
    public boolean test(Box p) {
      return p.size().height <= max && p.size().width <= max;
    }
  }

  class Screener extends DualOutputPushStage
                 implements PushStage {

    protected final BoxPredicate predicate;
    public Screener(BoxPredicate p) { predicate = p; }

    public void putA(final Box p) {
      if (predicate.test(p)) {
        new Thread(new Runnable() {
          public void run() { next1().putA(p); }
        }).start();
      }
      else
        next2().putA(p);
    }
  }
```

4.2.2.10 发送源（Source）

这里是一个例子代码，它产生随机大小的基本盒子（BasicBox）。为方便起见，它配备了一个自主循环的 run 方法，以重复调用 produce 方法，并使用随机的生产延迟时间。




```
  class BasicBoxSource extends SingleOutputPushStage
                       implements PushSource, Runnable {

    protected final Dimension size;      // maximum sizes
    protected final int productionTime; // simulated delay

    public BasicBoxSource(Dimension s, int delay) {
      size = s;
      productionTime = delay;
    }

    protected Box makeBox() {
      return new BasicBox((int)(Math.random() * size.width) + 1,
                          (int)(Math.random() * size.height) + 1);
    }

    public void produce() {
      next1().putA(makeBox());
    }

    public void run() {
      try {
        for (;;) {
```

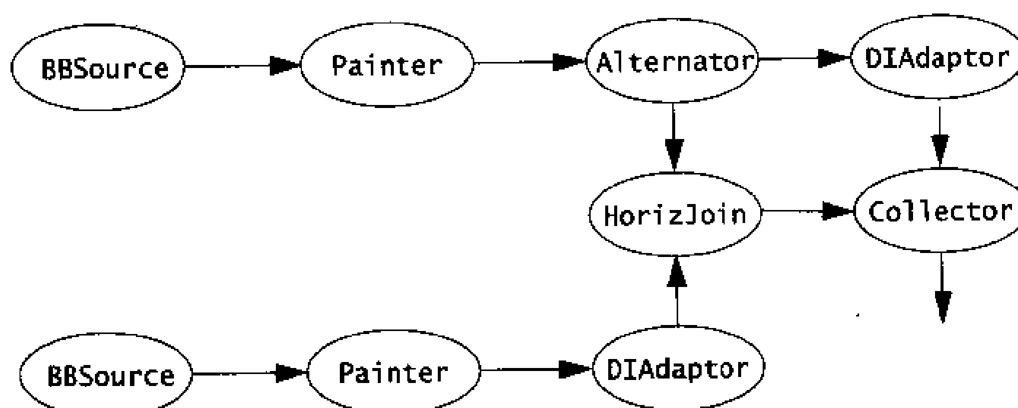

java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

```
            produce();
            Thread.sleep((int)(Math.random() * 2* productionTime));
          }
        }
        catch (InterruptedException ie) { } // die
      }

    }
```

4.2.2.11 协调（Coordination）

由于没有基于这些类的脚本工具，我们不得不通过手动地创建所需阶段的实例并把它们连接起来的方式编写组装线。这在理论上是容易的，但是由于对哪个阶段会连接到哪个阶段缺乏可视化的向导，所以在实际操作中是繁琐并容易出错的。

这里是一个在本节开始时介绍的生产图像的 applet 中用到的流的片段：



可以在在线附录中找到实现的代码，它的主要结构主要是由许多行如下所示的代码所组成的：

```
Stage aStage = new Stage();
aStage.attach(anotherStage);
```

然后运行所有发送源线程的 start 方法。

4.2.3 进阶阅读

流模式常常作为用例、场景、脚本，以及从高级面向对象分析（high-level object-oriented analysis）得来的相关概念的计算版本。在 § 1.3.5 和 § 1.4.5 列出的大多数关于 OO 设计和设计模式的书中都描述了与基于流的系统有关的分析、设计和实现的问题。与包传输网络（packet networking）、电信技术以及多媒体系统这样经常需要更精心的基于流的设计的领域相关的问题，可以在 § 1.2.5 并发和分布式系统的内容中找到。

4.3 线程中的服务（Services in Thread）

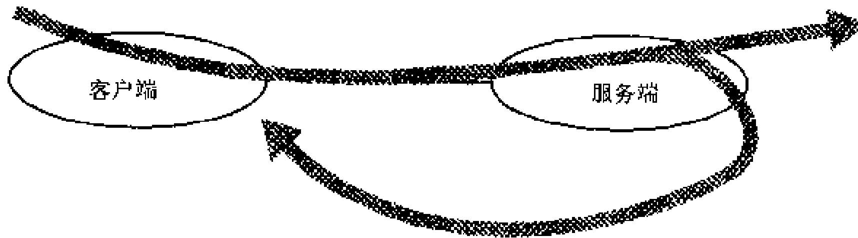
许多任务的计算结果或提供的服务虽然不会被它们的客户立刻使用，但是最终会被客户使用。在这些情况下，不像那些涉及单向消息的情况，在某种程度上，客户的操作依赖于特

java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

定任务的完成。

这一节描述了一些别的可用的设计方法：向单向消息中加入一个回调（callback）、依靠 Thread.join 方法、基于 Futures 构建实用工具以及创建工作者线程。结合如何在并行处理的机器上提高密集计算的性能的话题，§4.4 重温并扩展了这些技术。

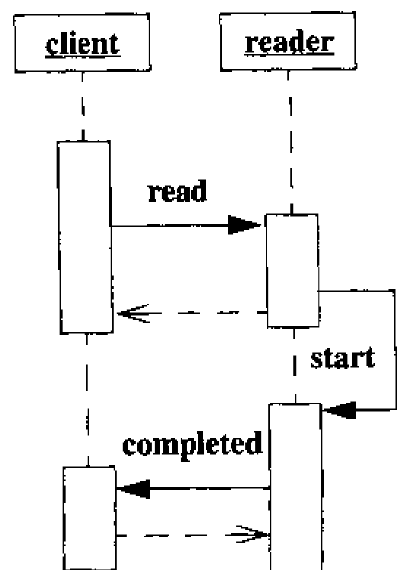
4.3.1 完成回调（Completion Callback）



从纯粹的单向消息传递的角度来说，最自然的处理任务完成的方式是：客户端通过给服务端发送一个单向消息的方式激活一个任务，服务器端稍后通过给调用者发送一个单向回调消息通知任务已经完成。这种有效、异步、基于通知的风格的设计，最适用于松耦合设计的情况，这种情况下，服务的完成将触发客户端一些独立的操作的。完成回调设计有时在结构上和观察者（Observer）的设计是完全一致的（参见§3.5.2）。

例如，假设某个应用程序可以提供几种功能，其中一个或多个功能需要首先读入一个特定的文件。由于 IO 操作相对比较慢，并且在这个文件读取的过程中，你并不希望其他的功能被禁止——这样会降低系统的响应能力。一种解决方法就是创建一个文件读取（FileReader）服务来异步地读取文件，当服务完成之后，向应用程序发送一条信息，而此后的应用程序就可以执行相应的功能了。

4.3.1.1 接口（Interface）



要构建这样一个 FileReader 或任何其他使用完成回调的服务，必须先为回调信息定义一个客户接口。接口中的方法必须能够完成在面向方法的服务版本中对应的对返回类型和异常的处理。通常需要两种方法，一种是与任务正常完成有关的，另一种是与调用过程中由异常引发的错误相关的。

另外，回调方法通常需要一个参数来指明哪个操作已经完成了，从而当存在多个调用的情况下，客户端可以对它们排序。大多数情况下，可以通过向客户端返回几个调用参数的方式简单地实现。在更常见的设计方案中，服务端返回一个惟一的标识符（通常称作 cookie），这个标识符既作为最初请求的返回值，也可以作为任何一个回调方法的参数。这个技术的变体被用于远程调用框架的幕后，它

java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

通过跨网络的异步消息实现过程调用。

在 FileReader 的例子里，我们可以像下面这样使用接口：

```
interface FileReader {
  void read(String filename, FileReaderClient client);
}

interface FileReaderClient {
  void readCompleted(String filename, byte[] data);
  void readFailed(String filename, IOException ex);
}
```

4.3.1.2 实现（Implementation）

根据你是倾向于由客户端还是由服务端创建执行服务的线程，存在两种实现这些接口的风格。通常，如果没有运行在自身的线程里时服务仍然有用，那么就应该将控制权分配给客户端。

在更典型的情况下，对线程的使用本质上仍然是完成回调设计，控制权被分配给服务方法。需要注意的是，这会导致回调方法在由服务端构造的线程中执行，而不是在客户端直接构造的线程中执行。如果代码依赖特定线程的属性，这还会带来意外的结果。这些属性例如：ThreadLocal 和 java.security.AccessControlContext（参见 § 2.3.2）。

这里我们可以通过使用服务创建线程的方式，实现一个客户端和服务端，如下所示：

```
class FileReaderApp implements FileReaderClient {  // Fragments
  protected FileReader reader = new AFileReader();

  public void readCompleted(String filename, byte[] data) {
    // ... use data ...
  }

  public void readFailed(String filename, IOException ex){
    // ... deal with failure ...
  }

  public void actionRequiringFile() {
    reader.read("AppFile", this);
  }

  public void actionNotRequiringFile() { ... }
}

class AFileReader implements FileReader {

  public void read(final String fn, final FileReaderClient c) {
    new Thread(new Runnable() {
      public void run() { doRead(fn, c); }
    }).start();
  }

  protected void doRead(String fn, FileReaderClient client) {
```


java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

```
        byte[] buffer = new byte[1024]; // just for illustration
        try {
          FileInputStream s = new FileInputStream(fn);
          s.read(buffer);
          if (client != null) client.readCompleted(fn, buffer);
        }
        catch (IOException ex) {
          if (client != null) client.readFailed(fn, ex);
        }
      }
    }
```

这里的服务类是用空的 client 参数来处理不需要回调的客户端。虽然在这里并不是十分可能，但是在许多服务里回调都是可选的。另一种方法是，你可以定义和使用一个 NullFileReaderClient 类，这个类里包含了一个没有任何操作的回调函数的版本（参见进阶阅读）。并且，从服务的角度考虑，回调的目标也可能是任意的对象，例如可能是某些请求服务的对象的辅助对象。你也可以使用在 § 3.1.1.6 介绍的技术，用事件通知来代替回调。

4.3.1.3 保护回调方法（Guarding callback method）

在某些应用程序中，客户端只在它们处于特殊状态时才可以处理完成时回调（completion callback）。在这里，回调方法自己可以包含保护机制，它可以挂起对每个输入的回调的处理，直到客户端可以处理它。

例如，假设我们有一个 FileReaderClient，它启动了一系列异步文件读取操作，并需要按照给定的顺序处理它们。这种结构模拟了通常情况下远程调用是怎样被处理的：典型地，为每一个请求分配一个顺序号，并且回复也是按照顺序号的顺序处理的。这种策略可能具有一定的风险，因为，如果回调中的一个或多个永远不会结束，那么它就会导致其他回调被无限期地挂起。不过，可以通过加入等待超时的方式解决这个缺点。

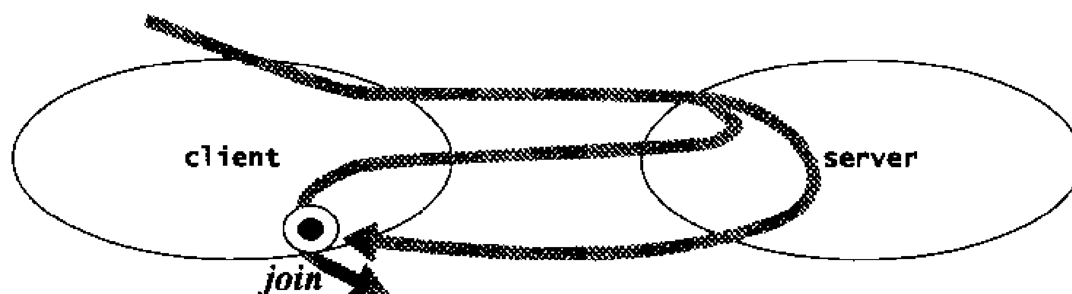
```
class FileApplication implements FileReaderClient { // Fragments
  private String[] filenames;
  private int currentCompletion; // index of ready file

  public synchronized void readCompleted(String fn, byte[] d) {
    // wait until ready to process this callback
    while (!fn.equals(filenames[currentCompletion])) {
      try { wait(); }
      catch(InterruptedException ex) { return; }
    }
    // ... process data...
    // wake up any other thread waiting on this condition:
    ++currentCompletion;
    notifyAll();
  }

  public synchronized void readFailed(String fn, IOException e){
    // similar...
  }

  public synchronized void readfiles() {
```


java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

```
      currentCompletion = 0;
      for (int i = 0; i < filenames.length; ++i)
        reader.read(filenames[i],this);
    }
  }
```

4.3.2 协作线程（Joining Thread）

虽然完成时回调（completion callback）非常灵活，但是当一个调用者需要等待由它启动的一个特殊任务超时结束时，这种方式就显得极为笨拙了。



如果线程 A 中的一个操作必须要等到某个线程 B 完成后才能继续执行，那么你就可以使用在第 3 章介绍过的任何一种等待和通知技术来阻塞线程 A。例如，假设存在一种从线程 A 和线程 B 都可以访问到的称为 terminated 的闭锁（Latch，参见 § 3.4.2），线程 A 可以通过 terminated.acquire()方法进行等待，而线程 B 也可以在它完成任务时执行 terminated.release()方法。

但是，通常并没有必要自己创建等待和通知机制，因为 Thread.join 方法已经提供了这种功能。当目标对象处于 isAlive 状态时，join 方法将调用者阻塞。终止线程会自动进行通知。在这种机制内部，用于等待和通知的监视对象可以是任意的，并且可能会根据 JVM 实现的不同而变化。但是在大多数情况下，都把目标线程对象自己作为监视对象（这也是为什么不扩展线程类以加入调用等待或通知的 run 方法的原因）。当 Thread.join 方法的这些特点不适合某个特殊应用程序需求的情况下，可以返回去使用手工方式。

无论是 Thread.join 方法还是明确地编码变量的方式都可以用于下面的设计中：客户端需要一个服务被执行，但它并不立即依赖这个服务的结果或效果［在某些情况下被称为***延迟同步（deferred-synchronous）***调用］。通常，如果服务任务是耗时的，则可以使用这种方法从使用 CPU 或 IO 的并行机制中得益，通过把它运行在一个单独的线程中可以提高整体的吞吐量。

一个常见的应用程序是图像处理。从一个文件或套接字中获得图像的原始数据并把它转换成某种可以显示的形式是一个非常耗时的操作，这种耗时既包括 CPU 处理又包括 IO 处理。通常，这些处理可以与其他显示相关的操作并行进行。

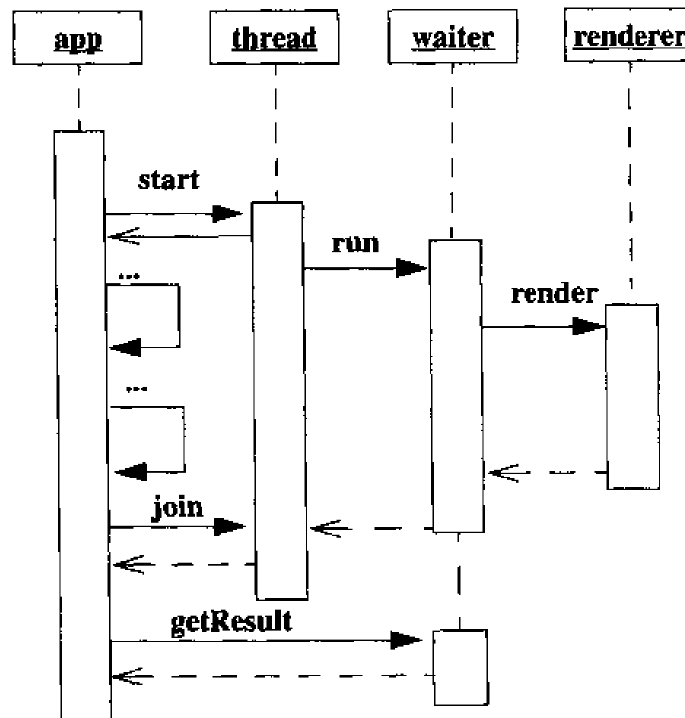
这种策略的一个版本被用于 java.awt.MediaTracker 和相关的类中，在合适的时候应该使用这种策略。这里，我们会展示一个更通常的版本，可以通过各种方式扩展和细化它从而支

java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

持特定的应用程序。

为了把它建立起来，假定存在一个通用的针对图像的 Pic 接口，以及一个 Renderer 接口，这个接口描述了一个接收指向图像数据的 URL 并最终返回 Pic 接口（在一个更现实的环境下，render 方法当然也会抛出各种各样的失败异常。在这里，我们假定在遇到任何错误时它只是简单地返回 null）。当然，也假定存在一个实现了 Renderer 接口的 StandardRenderer 类。

Thread.join 方法可以用于编写接下来的客户端程序，如 PictureApp 类（为了展示的目的，它只调用了几个伪造的方法）。它创建了一个实现了 Runnable 的 waiter 对象，这个对象既负责启动渲染线程也负责跟踪渲染的结果。



虽然这只是一个普通的练习，但是，如同在 waiter 对象中见到的那样，其中对非同步（或直接）访问内部的 result 字段的使用是比较精细的。由于访问是非同步的，所以正确性就依赖于线程终止和 join 方法都内在地使用了 synchronized 方法或块（参见 § 2.2.7）。

```
interface Pic {
  byte[] getImage();
}

interface Renderer {
  Pic render(URL src);
}

class PictureApp {                                    // Code sketch
  // ...
  private final Renderer renderer = new StandardRenderer();

  public void show(final URL imageSource) {
```


java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

```
      class Waiter implements Runnable {
        private Pic result = null;
        Pic getResult() { return result; }
        public void run() {
          result = renderer.render(imageSource);
        }
      };

      Waiter waiter = new Waiter();
      Thread t = new Thread(waiter);
      t.start();

      displayBorders();  // do other things
      displayCaption();  //  while rendering

      try {
        t.join();
      }
      catch(InterruptedException e) {
        cleanup();
        return;
      }

      Pic pic = waiter.getResult();
      if (pic != null)
        displayImage(pic.getImage());
      else
        // ... deal with assumed rendering failure
    }
  }
```

不管线程是成功完成还是异常终止，Thread.join 方法都将控制权返回给调用者。为了展示的简单性，在这里，使用 result 字段的空值指示任何一种失败，包括渲染器的退出。§ 4.3.3.1 的版本展示了一个更有效的方法。

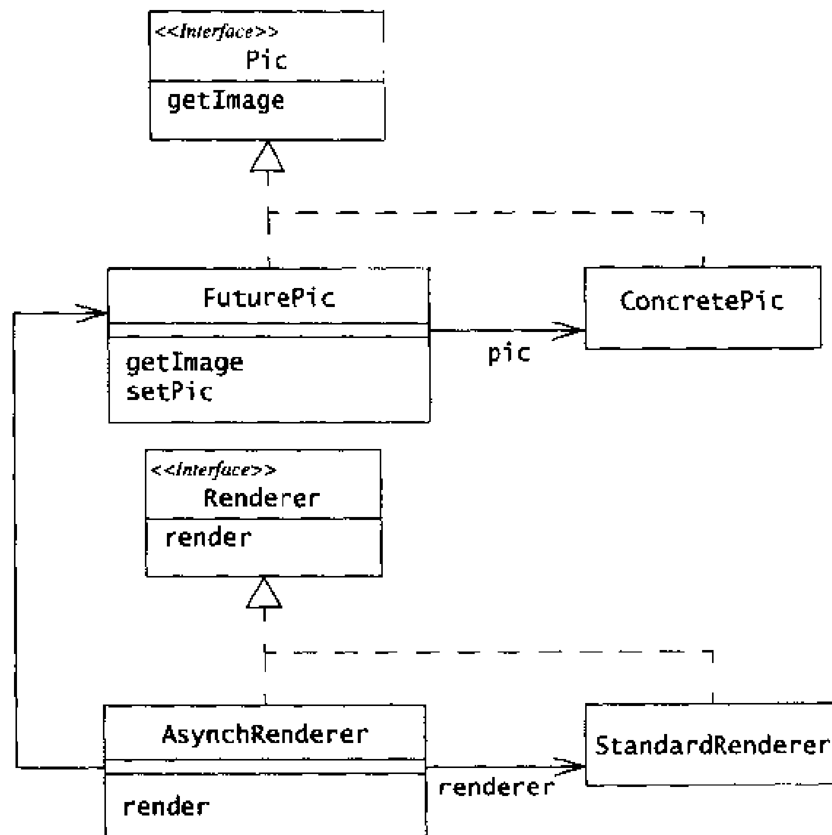
4.3.3 Future

通过下面列出的方式，可以把底层基于协作的构造（join-based construction）方法组织成一个更方便、更结构化的方式：

- 创建 ***Future***——当客户端在"虚拟"数据对象的计算还没有完成之前尝试调用它们的字段存取方法时，它们就会自动地阻塞。对于一个给定的数据对象，Future 作为"借据"（IOU）。
- 创建服务方法的不同版本，这些方法启动一个或多个线程并返回在计算完成时会解除阻塞的 Future 对象。

由于在数据访问和数据服务方法中内建了 Future 相关的机制，所以只有在数据对象和服务方法都使用接口，而不是使用类定义时，它们才可以以通常的方式应用。但是，如果定义了相关的接口，那么 Future 对象就可以很容易地建立了。例如，一个基于 Future 的 AsynchRenderer 类可以使用代理机制包装实际的实现类（参见 § 1.4.2）：

java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！




```
class AsynchRenderer implements Renderer {
  private final Renderer renderer = new StandardRenderer();

  static class FuturePic implements Pic { // inner class
    private Pic pic = null;
    private boolean ready = false;
    synchronized void setPic(Pic p) {
      pic = p;
      ready = true;
      notifyAll();
    }

    public synchronized byte[] getImage() {
      while (!ready)
        try { wait(); }
        catch (InterruptedException e) { return null; }
      return pic.getImage();
    }
  }

  public Pic render(final URL src) {
    final FuturePic p = new FuturePic();
    new Thread(new Runnable() {
      public void run() { p.setPic(renderer.render(src)); }
    }).start();
    return p;
  }
}
```


java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

为了介绍的目的，AsynchRenderer 类使用了显示的、基于 ready 标志的等待和通知操作，而没有使用 Thread.join 方法。

依赖于这个类的应用程序可以以一种简单的方式写成：

```
class PicturAppWithFuture {                        // Code sketch
  private final Renderer renderer = new AsynchRenderer();

  public void show(final URL imageSource) {
    Pic pic = renderer.render(imageSource);

    displayBorders();  // do other things ...
    displayCaption();

    byte[] im = pic.getImage();
    if (im != null)
      displayImage(im);
    else  // deal with assumed rendering failure
  }
}
```

4.3.3.1 Callable 接口

大多数基于 Future 的设计都明确地采用在 AsynchRenderer 类中展示的形式。对这些类的构造和使用可以通过逐步采用一个更一般的接口的方式进行进一步地标准化和自动化。

Runnable 接口描述了任何纯粹的操作，以同样的方式，一个 Callable 接口可以用于描述任何的服务方法，这个方法接收一个 Object 参数，返回一个 Object 结果，并可能抛出一个 Exception 异常。

```
interface Callable {
  Object call(Object arg) throws Exception;
}
```

这里对 Object 类的（笨拙的）使用是为了适应不同的情况，例如，可以通过把参数放到一个对象数组中，使方法可以接收多个的参数。

虽然还有其他的选择，但是通过一个协调应用的单一的类来包装支持机制是最方便的。接下来的 FutureResult 类展示了一系列的选择（它是可以从在线附录中得到的 util.concurrent 包中的一个类的简化版本）。

FutureResult 类维护着一些方法以获得返回的结果 Object，或被 Callable 对象抛出的异常。与我们介绍过的 Pic 版本不同，在那个版本里，所有的错误都只是通过 null 的方式指示。这里对中断的处理则更忠实一些，它会把抛出的异常返回给想要得到结果的客户端。

为了能够恰当地将在服务中碰到的异常与那些在尝试执行服务时碰到的异常区分开，被 Callable 对象抛出的异常会使用 java.lang.reflect.InvocationTargetException 重新包装，这个类是将一个异常封装在另一个异常中的通用类。

当然，为了通用性的目的，FutureResult 类自身并不创建线程。作为替代，它提供能够返回 Runnable 接口的 setter 方法，这样，用户就可以在线程内部或任何其他代码 Executor

java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

（参见 § 4.1.4）内执行这个 Runnable 对象。这也可以使 Callable 在轻量级的执行框架中可用，这个执行框架用来处理通过单向消息启动的任务。作为一个替代性的策略，你可以建立一个与 Executor 类似的 Caller 框架，但是它更针对服务类型任务的需要，例如，调度执行、状态检查，以及控制异常响应的支持方法：

```
class FutureResult {                              // Fragments
  protected Object value = null;
  protected boolean ready = false;
  protected InvocationTargetException exception = null;

  public synchronized Object get()
   throws InterruptedException, InvocationTargetException {

    while (!ready) wait();

    if (exception != null)
      throw exception;
    else
      return value;
  }

  public Runnable setter(final Callable function) {
    return new Runnable() {
      public void run() {
        try {
          set(function.call());
        }
        catch(Throwable e) {
          setException(e);
        }
      }
    };
  }

  synchronized void set(Object result) {
    value = result;
    ready = true;
    notifyAll();
  }

  synchronized void setException(Throwable e) {
    exception = new InvocationTargetException(e);
    ready = true;
    notifyAll();
  }

  // ... other auxiliary and convenience methods ...

}
```

FutureResult 类可以用于直接支持通用的 Future，或者作为一个构造更特殊版本的工具。下面是一个直接使用方式的例子：

java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

```
class PictureDisplayWithFutureResult {            // Code sketch

  private final Renderer renderer = new StandardRenderer();
  // ...

  public void show(final URL imageSource) {

    try {
      FutureResult futurePic = new FutureResult();
      Runnable command = futurePic.setter(new Callable() {
        public Object call() {
          return renderer.render(imageSource);
        }
      });
      new Thread(command).start();

      displayBorders();
      displayCaption();

      displayImage(((Pic)(futurePic.get())).getImage());
    }

    catch (InterruptedException ex) {
      cleanup();
      return;
    }
    catch (InvocationTargetException ex) {
      cleanup();
      return;
    }
  }
}
```

这个例子由于使用了通用类来标准化应用协议，而显得不够灵活。这也可能是你想用 FutureResult 来构造一个更标准化、更容易使用的，与 AsynchRenderer 类具有相同的方法和结构的版本的原因。

4.3.4 调度服务（Scheduling Service）

如同在 §4.1.4 所讨论过的，在某些时候，工作者线程设计同每任务一线程（thread-per-task）设计相比，可以提高性能。它们也能够用于调度和优化执行不同客户发出的服务请求。

作为一个著名的例子，考虑一下控制磁盘读写访问的类，磁盘中包含了许多柱面，但是却只有一个读写磁头。服务的接口只包含读和写两种方法。实际上，它当然可以使用块标识符（block indicator）来替代原始的柱面数，并处理掉或抛出各种 IO 异常，在这里，我们只是简单地把它作为一个单一的 Failure 异常。

```
interface Disk {
  void read(int cylinderNumber, byte[] buffer) throws Failure;
  void write(int cylinderNumber, byte[] buffer) throws Failure;
}
```

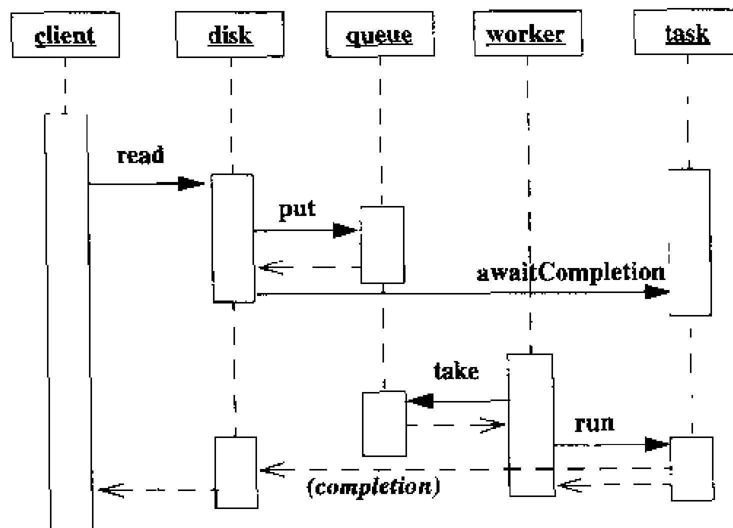

java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

在柱面间进行扫描时，与按照服务产生的顺序维护访问请求相比，按照升序访问柱面，并且在每次扫描结束后将磁头重新设置到开始的位置，这样平均起来会更快（根据磁盘类型的不同，按照升序再降序扫描的方式可能更好，但是这里仍然坚持采用升序方式）。

如果不使用某些辅助性数据结构，这种策略可能是很复杂的。允许一个请求执行的条件是：

> 一直等到当前的请求柱面号比当前的磁头位置柱面号大，但这个柱面号又是所有等待服务的请求中柱面号最小的一个，或者如果当前磁头的柱面号比所有请求的柱面号都大时，那么取请求中柱面号最小的一个处理。

这个条件实在是太笨拙、太低效了，并且在一系列不相关的客户端之间进行协调时，它甚至可能会导致死锁。但是，通过使用一个由工作者线程维护的排序队列的帮助，可以相当容易地实现它。任务可以按照柱面号排序加入到队列里，并在轮到它时被执行。通过使用两个队列可以很容易地实现这种"电梯算法"（elevator algorithm），其中一个队列用于保存当前的扫描，而另一个用于保存下一次扫描。



这样产生的框架是将类似 Future 的结构与 § 4.1.4 的工作者线程设计结合起来。为了建立这样的一个框架，我们需要定义一个 Runnable 类以包含与 DiskTask 相关的额外维护。队列类使用在 § 3.4.1 讨论过的基于信号机的方法，但是在这里它将用于排序链表。服务类构造一个运行队列里任务的工作者线程。公有的服务方法负责创建任务，把它们放到队列之中，在任务执行后，服务方法才结束，并通知客户端。

```
abstract class DiskTask implements Runnable {
  protected final int cylinder;          // read/write parameters
  protected final byte[] buffer;
  protected Failure exception = null;         // to relay out
  protected DiskTask next = null;             // for use in queue
  protected final Latch done = new Latch(); // status indicator

  DiskTask(int c, byte[] b) { cylinder = c; buffer = b; }

  abstract void access() throws Failure; // read or write

  public void run() {
```


java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

```
    try  { access(); }
    catch (Failure ex) { setException(ex); }
    finally { done.release(); }
  }

  void awaitCompletion() throws InterruptedException {
    done.acquire();
  }

  synchronized Failure getException() { return exception; }
  synchronized void setException(Failure f) { exception = f; }
}
```

```
class DiskReadTask extends DiskTask {
  DiskReadTask(int c, byte[] b) { super(c, b); }
  void access() throws Failure { /* ... raw read ... */ }
}

class DiskWriteTask extends DiskTask {
  DiskWriteTask(int c, byte[] b) { super(c, b); }
  void access() throws Failure { /* ... raw write ... */ }
}
```

```
class ScheduledDisk implements Disk {
  protected final DiskTaskQueue tasks = new DiskTaskQueue();

  public void read(int c, byte[] b) throws Failure {
    readOrWrite(new DiskReadTask(c, b));
  }

  public void write(int c, byte[] b) throws Failure {
    readOrWrite(new DiskWriteTask(c, b));
  }

  protected void readOrWrite(DiskTask t) throws Failure {
    tasks.put(t);
    try {
      t.awaitCompletion();
    }
    catch (InterruptedException ex) {
      Thread.currentThread().interrupt(); // propagate
      throw new Failure(); // convert to failure exception
    }
    Failure f = t.getException();
    if (f != null) throw f;
  }

  public ScheduledDisk() {     // construct worker thread
    new Thread(new Runnable() {
      public void run() {
        try {
          for (;;) {
            tasks.take().run();
```


java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

```
            }
          }
          catch (InterruptedException ie) {} // die
        }
      }).start();
    }
  }
```

```
class DiskTaskQueue {
  protected DiskTask thisSweep = null;
  protected DiskTask nextSweep = null;
  protected int currentCylinder = 0;

  protected final Semaphore available = new Semaphore(0);

  void put(DiskTask t) {
    insert(t);
    available.release();
  }

  DiskTask take() throws InterruptedException {
    available.acquire();
    return extract();
  }

  synchronized void insert(DiskTask t) {
    DiskTask q;
    if (t.cylinder >= currentCylinder) {   // determine queue
      q = thisSweep;
      if (q == null) { thisSweep = t; return; }
    }
    else {
      q = nextSweep;
      if (q == null) { nextSweep = t; return; }
    }
    DiskTask trail = q;              // ordered linked list insert
    q = trail.next;
    for (;;) {
      if (q == null || t.cylinder < q.cylinder) {
        trail.next = t; t.next = q;
        return;
      }
      else {
        trail = q; q = q.next;
      }
    }
  }
  synchronized DiskTask extract() { // PRE: not empty
    if (thisSweep == null) {          // possibly swap queues
      thisSweep = nextSweep;
      nextSweep = null;
    }
    DiskTask t = thisSweep;
    thisSweep = t.next;
    currentCylinder = t.cylinder;
```


java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

```
        return t;
    }
}
```

4.3.5 进阶阅读

ABCL 是最早把 Future 作为语言的固件的面向对象的并发编程语言之一。参见：

Yonezawa, Akinori 和 Mario Tokoro. 《Object-Oriented Concurrent Programming》, MIT Press, 1988.

在 Eiffel//语言（对 Eiffel 语言的并行扩展）中，Future 被称为 ***wait-by-necessity*** 构件。见：

Caromel, Denis, and Yves Roudier. "Reactive Programming in Eiffel//", in Jean-Pierre Briot, Jean-Marc Geib 和 Akinori Yonezawa (eds.) 《Object Based Parallel and Distributed Computing》, LNCS 1107, Springer Verlag, 1996.

在 Scheme 和 Multilisp 编程语言中，Future 和相关结构的描述见：

Dybvig, R. Kent 和 Robert Hieb. "Engines from Continuations", 《Computer Languages》, 14(2):109-123, 1989.

Feeley 和 Marc. 《An Efficient and General Implementation of Futures on Large Scale Shared-Memory Multiprocessors》, PhD Thesis, Brandeis University, 1993.

在网络应用中，与完成时间调相关的额外技术的介绍见：

Pyarali, Irfan, Tim Harrison 和 Douglas C. Schmidt. "Asynchronous Completion Token", in Robert Martin, Dirk Riehle, and Frank Buschmann (eds.), 《Pattern Languages of Program Design, Volume 3》, Addison-Wesley, 1999.

在客户端并不总是需要回调消息的情况下，空对象模式（Null Object pattern）对简化回调设计常常是非常有用的。见：

Woolf 和 Bobby. "Null Object", in Robert Martin, Dirk Riehle, and Frank Buschmann (eds.), 《Pattern Languages of Program Design, Volume 3》, Addison-Wesley, 1999.

4.4 并行分解（Parallel Decomposition）

通过特别地设计以利用多个 CPU 的运算能力，并行编程可以解决密集计算的问题。它的主要性能指标通常是吞吐量和可伸缩性，即在每个单位时间内，可以被执行的运算数量，以及当出现额外可用计算资源时，性能提高的潜力。但是，这些目标与其他性能目标常常纠缠在一起。例如，对于一个将工作分发到并行执行程序的服务，并行性也可以改善它的响应延迟。

在 Java 编程语言中，并行性的主要挑战是，在存在多个 CPU 的情况下，创建能够利用这些 CPU 的***可移植性（portable）***的程序，同时在单处理器、以及常常是处理不相关程序的分时多处理器（time-shared multiprocessor）的情况下，也能够正常运行。

但是，一些经典的并行性的方法并不能很好地满足这些目标。需要假定特定的体系结构、

java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

拓扑结构、处理器的处理能力，或其他固定的环境约束的方法，都不适用于通常可用的 JVM 实现。虽然通过特殊的扩展以适应特殊的并行计算机，针对特定的计算机编写特定的并行程序来创建运行时系统（run-time system）并不是一种错误；但是与这些相关的编程技巧已经超出了本书的范围。当然，RMI 和其他的分布式框架也可以通过远程计算机来获得并行性。事实上，在这里所讨论的大多数设计方式都可以通过调整成在本地网络上使用串行化和远程调用的方式来获得并行性。而且，这种方法也在成为一个常见而且有效的粗粒度的并行处理方式。但是，这些机制也超出了本书的范围。

作为替代，我们着重于三个基于任务的设计的系列：分支/合并（fork/join）的并行性、计算树（computation tree），以及关卡（barrier）。在存在多个处理器的情况下，通过这些技术可以得到非常有效的、可以利用多个处理器的程序，同时还保持着可移植性和串行效率。根据经验，它们至少可以在几十个 CPU 之间自由伸缩。甚至，当这些种基于任务的并行程序被调优到最大限度地利用给定的硬件平台的情况下，它们只需要较小的调整就可以最大限度地利用其他的平台。

在本书写作的过程中，这些技术最常见的目标就是应用服务器和计算服务器，它们常常，但并不总是使用多处理器。无论在哪种情况下，我们假定有充足的 CPU 周期资源，所以利用它们的主要目标就是提高对计算问题的解决。换句话说，如果程序运行在接近饱和的计算机上，这些技术未必很有帮助。

4.4.1 分支/合并

分支/合并分解依赖于与串行算法设计（sequential algorithm design）中常用的分而治之（divide-and-conquer）技术的并行版本。这种方案采用下面的形式：

```
pseudoclass Solver {                                   // Pseudocode
  // ...
  Result solve(Param problem) {
    if (problem.size <= BASE_CASE_SIZE)
      return directlySolve(problem);
    else {
      Result l, r;
      IN-PARALLEL {
        l = solve(lefthalf(problem));
        r = solve(rightHalf(problem));
      }
      return combine(l, r);
    }
  }
}
```

这里花费了大量的工作和精力来设计一个分而治之算法。但是许多常见的计算密集问题已经有已知的类似于这种形式的解决方案。当然，可能会有不止两层递归调用、多个基本操作以及与这些操作相关的任意数量的前题处理和结束处理操作。

熟悉的串行处理的例子包括：快速排序、归并分类，以及许多对数据结构、矩阵和图像进行处理的算法等。在递归的任务是相互独立的情况下，串行递归的分而治之的设计是易于

java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

并行化的；在这种情况下，这些任务操作在数据集上的不同部分（例如一个数组的不同部分）或解决不同的子问题，并且不需要通信或协调的操作。这些经常体现于递归算法中，甚至是那些最初并不是有意为实现并行的算法。

另外，有些在串行化环境中并不常用的算法（例如，矩阵乘法）也有递归的版本，但是由于它们已有的易于并行化的形式，它们更常用于多处理器的环境中。而其他的并行算法，通过执行大量的变换和预处理以把问题转换成可以使用 fork/join 技术解决的形式（参见 § 4.4.4 的进阶阅读）。

IN-PARALLEL 的伪码通过 forking 和随后调用 joining 执行递归的任务来实现。但是，在讨论怎么做之前，我们首先分析一些可以对由递归产生的任务进行有效地并行执行的讨论和框架。

4.4.1.1 任务的颗粒度与结构

在实现 fork/join 设计时遇到的设计问题大多与任务粒度有关：

最大化并行性（Maximizing parallelism）。总体来说，任务越小，就越容易使用并行性。在所有事情都是相同的情况下，通过使用大量细粒度（fine-grained）的任务而不只是少量粗粒度（coarse-grained）的任务会使等多的 CPU 处于繁忙状态，同时可以提高负载平衡、本地化和可伸缩性，降低 CPU 空闲，等待其他任务的时间比率，从而达到更高的吞吐量。

最小化开销（Minimizing overhead）。在并行的情况下，构造和管理对象来处理任务，而不只是顺序地调用一个方法来处理任务，是与串行解决方法相比基于任务的编程方式不可避免的主要开销。在本质上，创建和使用任务对象比使用堆栈结构的代价更昂贵。另外，对任务对象的使用也会增加必须被传递的参数和返回结果数据的数量，从而影响垃圾收集。在所有事情都是相同的情况下，且在只有少量粗粒度的任务的情况下，总的开销是最小的。

最小化争用（Minimizing contention）。如果每个任务都需要经常和其他任务通信，或者必须要等待其他任务所拥有的资源，那么并行性不会带来太大的速度提高。任务的大小和结构应该保证尽可能多的独立性。它们应该尽可能少地使用（在大多数情况下，不要使用）共享资源、全局（静态）的变量，以及所有的相关性。理想情况下，每个任务只包含简单、直接的，从任务开始一直运行到结束的代码。但是，fork/join 的设计至少需要少量的同步。触发处理的主要对象通常要等待所有的子任务结束后再继续处理。

最大化本地化（Maximizing locality）。每个子任务都应该处理问题中一小部分的惟一任务的，这不仅是在理念上，还应在底层的资源和内存访问模式的级别上。在具有大量缓存的现代处理器上，将引用本地化的重构可以显著地提高性能。在处理大量的数据集合时，甚至在并行性并不是主要目的的情况下，将计算分割成被良好本地化的计算单元也是很正常的。递归分解常常是实现这个目标的有效方式。并行性强调本地化的作用。当所有的并行任务都访问数据集的不同部分时（例如，一个普通矩阵的不同区域），通常，可以减少在缓存间传递更新数据的分区策略可以得到更好的性能。

4.4.1.2 框架

对于颗粒度和相关任务结构的问题没有什么通用的最佳解决方案。任何一种选择都代表

java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

了一种能够最好地解决当前问题的折衷方案。但是，也有可能构建一个支持大多数选择的轻量级的执行框架。

线程对象没有必要成为支持纯粹计算性的 fork/join 任务的重型运输车。例如，有些任务永远都不需要在 IO 上阻塞，并且永远也不需要处于 sleep 状态。它们只需要一个用来同步子任务的操作。可以通过扩展在 § 4.1.4 讨论过的工作者线程技术来构造可以有效支持所需结构的框架。虽然存在不同的方法，但是为了讨论得具体，我们的讨论限制在 util.concurrent 包中的一个框架中，它限制所有的任务都必须是 FJTask 类的子类。下面是主要方法的一个简略骨架。更详细的细节会从 § 4.4.1.4～ § 4.4.1.7 随着例子仔细讨论。

```
abstract class FJTask implements Runnable {
  boolean isDone();                  // True after task is run
  void cancel();                     // Prematurely set as done
  void fork();                       // Start a dependent task
  void start();                      // Start an arbitrary task
  static void yield();               // Allow another task to run
  void join();                       // Yield caller until done
  static void invoke(FJTask t);      // Directly run t
  static void coInvoke(FJTask t,
                       FJTask u);    // Fork and join t and u
  static void coInvoke(FJTask[] tasks); // coInvoke all
  void reset();                      // Clear to allow reuse
}
```

一个相关的 FJTaskRunnerGroup 类提供了这个框架的控制和入口点。FJTaskRunnerGroup 创建给定数量的工作者线程。通常情况下，工作者线程的数量应该与系统上 CPU 数量相同。这个类的 invoke 方法可以启动一个主任务，这个主任务又可以依次创建其他任务。

FJTasks 必须只使用这些任务控制方法，而不是任意的 Thread 或监控方法。虽然这些操作名与 Thread 类的操作名相同或类似，但是它们的实现却截然不同。特别地，这里没有任何常见的挂起机制。例如，join 操作只是通过下面的方式简单地实现：让它的工作者线程运行其他任务到完成，直到被通知目标任务已经完成（通过 isDone 方法）。如果使用普通的线程，这种方式可能根本不会奏效，但是在所有的任务都被组织成 fork/join 方法的形式时，它就非常有效而且高效了。

这些折衷的方式使 FJTask 的构造和调用实质上可能比任何支持完整的 Thread 接口的类更廉价。在本书写作的过程中，至少在某些平台上，对于在下面展示的各种例子中，创建、运行以及管理一个 FJTask 的开销只是执行同样串行方法调用的开销的 4～10 倍之间。

这种方法的主要效果就是：在任务分割和任务颗粒度之间做出选择时，最终降低开销因素的影响。任务的颗粒度的阈值可以是相当小的，即使在最保守的情况下，对于几千个指令的任务，与在单处理器上执行相比也不会有显著的性能降低。即使在最大的平台上，程序也会利用所有可用的 CPU，而不需要任何特殊的工具来提取或管理并行性。但是，成功也依赖于任务类和方法的构造，它们需要尽可能地减小开销、避免争用，以及保持本地化。

4.4.1.3 定义任务

通过下面的步骤，串行化的分而治之算法可以表示成基于 fork/join 的类：

java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

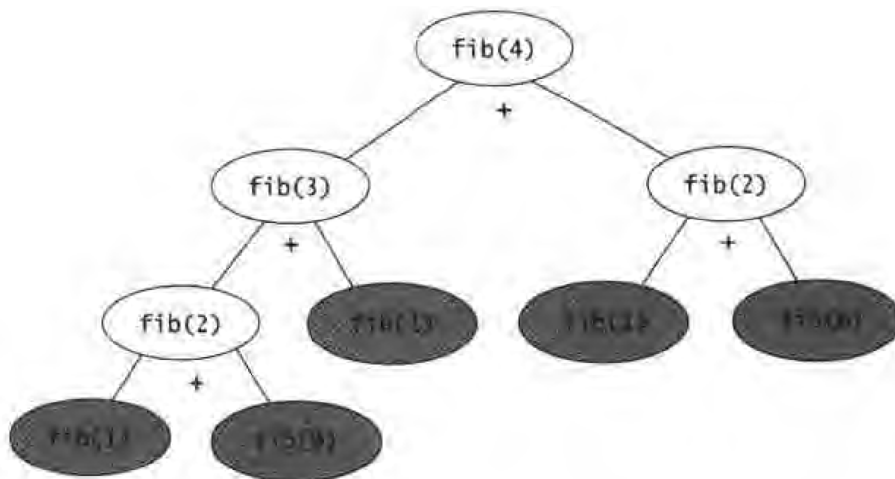
1. 使用下面的属性创建一个任务类：
   - 保存参数和结果字段。它们中的大多数应该被严格控制在一个任务的内部，永远不要从其他任务访问这些字段。这可以消除在使用它们的过程中对同步的需要。但是，典型情况下，其他任务会访问它的结果变量，这些变量要么被声明为volatile，要么只能通过同步（synchronized）方法访问。
   - 一个初始化参数变量的构造函数。
   - 一个运行通过改写方法代码而得到的 run 方法。
2. 用下面的代码替换最初的递归：
   - 创建子任务对象。
   - 派生（fork）每一个子任务，并让它们并行运行。
   - 主任务连接（join）每一个子任务。
   - 通过访问每一个子任务的结果变量组合最后的结果。

3. 用阈值检查替代（或扩展）原来的判断是否为基本操作的检查。小于阈值的问题应该使用原来的串行化代码。当问题的规模使得并行执行的开销超过了它带来的潜在收益时，这样的检查就可以保证效率。对于可能的问题设置一个好的阈值，对性能调优是非常有益的。

4. 用一个创建相关任务的方法替代原来的方法，等它运行结束后返回结果（在 FJTask 框架中，最外面的调用通过 FJTaskRunnerGroup.invoke 方法执行）。

4.4.1.4 斐波那契（Fibonacci）

我们会使用一个单独的、不现实，但是非常经典的例子演示这些基本步骤：递归计算的fib，Fibonacci 函数。可以采用串行化的方式编制这个函数，如下所示：




```
int seqFib(int n) {
   if (n <= 1)
      return n;
   else
      return seqFib(n-1) + seqFib(n-2);
}
```

这个例子是不现实的，因为对于这个特殊的问题，有更快的非递归的方法，但是对于展

java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

示递归和并行性的目的，这是一个非常受欢迎的例子。因为它只做了很少的其他计算，而且更容易看到 fork/join 设计的基本结构，虽然它产生了很多递归调用——至少 fib(n)个调用用来计算 fib(n)。最初的几个顺序值是：0, 1, 1, 2, 3, 5, 8；fib(10)是 55；fib(20)是 6,765；fib(30)是 832,040；fib(40)是 102,334,155。

seqFib 函数可以被转换成如下所示的一个任务类：

```
class Fib extends FJTask {
  static final int sequentialThreshold = 13; // for tuning
  volatile int number;                       // argument/result

  Fib(int n) { number = n; }

  int getAnswer() {
    if (!isDone())
      throw new IllegalStateException("Not yet computed");
    return number;
  }

  public void run() {
    int n = number;

    if (n <= sequentialThreshold)          // base case
      number = seqFib(n);
    else {
      Fib f1 = new Fib(n - 1);             // create subtasks
      Fib f2 = new Fib(n - 2);

      coInvoke(f1, f2);                    // fork then join both

      number = f1.number + f2.number;      // combine results
    }
  }


  public static void main(String[] args) { // sample driver
    try {
      int groupSize = 2;     // 2 worker threads
      int num = 35;          // compute fib(35)
      FJTaskRunnerGroup group = new FJTaskRunnerGroup(groupSize);
      Fib f = new Fib(num);
      group.invoke(f);
      int result = f.getAnswer();
      System.out.println("Answer: " + result);
    }
    catch (InterruptedException ex) {} // die
  }
}
```

注意：

- 这个类中有一个用来保存计算斐波那契函数的参数的字段。当然，我们也需要另外用来保存计算结果的变量。但是，如同在这些类里相当典型的情况，没有必要保存两个变量。为了节约（请记住，在一个计算过程中可能会创建上百万个 Fib 对象），我们可以为了优化的目的只使用一个变量，在计算后用计算结果覆盖原来的参数

java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

（这是几种令人不舒服的手工优化方式中的第一个，它在这里出现的目的是为了在构造有效的并行程序时，示范一些重要的实用小技巧）。

- 把 number 字段声明为 volatile 是为了保证在它被计算后，可以从其他任务或线程读到（参见 § 2.2.7）它的新值。在这里和接下来的例子里，在每个任务执行的过程中，volatile 字段都只会被读和/或写一次，否则的话它就应该被保存为局部变量。这种方式避免了由于潜在的编译器优化带来的影响，在不使用 volatile 时，编译器可能会做一些优化。
- 另外，我们也可以使用同步的方式访问 number 字段。但是这样做并不是一个好主意。通常，在轻量级的并行任务框架中，对 volatile 的使用要比在通常目的的并发编程中用的多得多。任务除了经常通过访问成员变量返回结果以外，通常并不需要其他的同步和控制机制。用 synchronized 替代 volatile 的最常见原因是在处理数组的时候。独立的数组元素不能被声明成 volatile。即使是在不需要锁的例子里，在 synchronized 的方法或代码块中处理数组也是保证数组更新的可见性的最简单的方法。一个偶尔会具有吸引力的替代方法是，数组中的每个元素都是有 volatile 成员变量的传递对象。
- 在通过 invoke 或 coInvoke 执行 run 方法完成后，isDone 方法会返回 true。它在 getAnswer 方法中用作一种保护手段，用于检查由于最终的客户程序在计算没有完成时尝试访问结果时导致的程序错误（在这里，这种可能没有机会发生，但是这种防护措施可以帮助我们避免无意识的错误使用）。
- sequentialThreshold 常量设立了颗粒度。它代表了是不是值得创建任务的开销的平衡点，并反映了维护良好串行化性能的设计目标。例如，对于一个运行在 4 个 CPU 的系统上，在使用一个工作者线程的情况下，当 seqFib 函数的参数值非常大时，把 sequentialThreshold 值设置为 13 会导致 4%的性能下降。但是在使用 4 个工作者线程的情况下，每秒处理几百万个 Fib 任务，它就会带来至少 3.8 倍的性能提高。
- 与其使用一个编译时常量，不如把阈值定义为一个运行期变量，并根据可用的 CPU 数量或其他的平台特性给它设置一个值。在性能不是线性增长的基于任务的程序中，甚至是这个例子中，都是非常有用的。随着 CPU 数量的增长，通信和资源管理的开销也会相应地增长，但是可以通过增加阀值的方式平衡 CPU 增长带来的开销。
- 通过一个方便的方法可以并行地模拟递归，例如 coInvoke(FJTask t, FJTask u)，它完成如下操作：

```
t.fork();invoke(u);t.join();
```

- fork 方法是对 Thread.start 方法的某种模拟。任何一个派生的任务，在创建它的工作者线程中总是按照基于栈的 LIFO（后进先出）的顺序被处理。但是对于由其他并行工作者线程运行的其他任务而言，它们之间是按照基于队列的 FIFO（先进先出）的顺序。这体现了通常基于栈的串行调用方式与通常基于队列的线程调度方式的交叉排序。对于递归的基于任务的并行性而言，这种（通过双端队列实现的）策略是非常理想的（见进阶阅读），并且当处理具有严格依赖的任务时，这种策略是很常

java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

见的——这些任务由最终将它们相互连接起来的任务或子任务创建。

- 与之相比，FJTask.start 方法的行为更像 Thread.start 方法。它使用了基于 FIFO 的队列以调度所有的工作者线程。例如，FJTaskRunnerGroup.invoke 方法会使用它启动执行一个新的主任务。
- Join 方法应该只在通过 fork 启动任务时使用。它利用 fork/join 子任务的终止依赖模式优化任务的执行。
- FJTask.invoke 方法在另一个任务内运行一个任务的主体，并等待它完成。这种方式看起来很特别，它是 coInvoke 的单任务版本，是对 u.fork(),u.join()的优化。

如同使用最初的 Thread 类编程那样，对任何轻量级可执行框架的有效使用都需要对所提供的方法和语义的深入理解。FJTask 框架利用了在递归和并行分解之间的相互依赖，并因此促成在 Fib 中见到的分而治之的编程风格。但是，符合这种通用风格的编程准则和设计模式的范围是相当广泛的，如同在接下来的例子中所展示的。

4.4.1.5 链接子任务（Linking subtask）

即使在派生（forked）的子任务的数量是动态变化的情况下，也可以使用 fork/join 技术。在实现这种方式的几种相关的策略中，你可以加入链接字段，从而可以通过列表维护子任务。在产生所有的任务之后，累加（accumulate）[也被称为归纳（reduction）] 操作使可以串行地遍历列表，连接并使用每一个子任务的结果。

现在，进一步扩展 Fib 这个例子，FibVL 类展示了实现它的一种方式。这种风格的解决方案在这里并不是非常有用，但是它比较适用于子任务的数量是动态创建，甚至可能在不同的方法中创建的情况下。这里需要注意子任务按照它们被创建相反的顺序被连接起来。由于结果的处理顺序并不重要，所以可以使用最简单的链接算法 [在当前元素前插入（prepending)]，这种方法在遍历的过程中反转了创建任务的顺序。无论什么时候，累加步骤都是可交换的而且是与计算结果相关的，应用这种策略，任务可以按照任何顺序处理。如果需要考虑顺序，那么我们可能需要相应地调整列表的结构或遍历方式。

```
class FibVL extends FJTask {
  volatile int number; // as before
  final FibVL next;    // embedded linked list of sibling tasks

  FibVL(int n, FibVL list) { number = n; next = list; }

  public void run() {
    int n = number;
    if (n <= sequentialThreshold)
      number = seqFib(n);
    else {
      FibVL forked = null;                    // list of subtasks

      forked = new FibVL(n - 1, forked); // prepends to list
      forked.fork();

      forked = new FibVL(n - 2, forked);
      forked.fork();
```


java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

```
        number = accumulate(forked);
      }
    }

    // Traverse list, joining each subtask and adding to result
    int accumulate(FibVL list) {
      int sum = 0;
      for (FibVL f = list; f != null; f = f.next) {
        f.join();
        sum += f.number;
      }
      return sum;
    }
  }
```

4.4.1.6 回调

可以扩展基于任务的递归的 fork/join 并行算法，从而在使用其他局部同步条件替代 join 的情况下，可以使用它。在 FJTask 框架中，t.join()方法是作为下面代码的优化版本实现的：

```
while (!t.isDone()) yield();
```

在这里，yield 方法允许底层的工作者线程处理其他的任务（更特殊地，在 FJTask 框架中，如果存在其他任务，那么这个线程至少会处理一个其他任务）。

除了 isDone 方法之外，任何其他的条件都可以用在这种结构中，只要你确定：由于任务的子任务（或子任务之一，等等）的操作，正在等待的判定条件最终一定会变成 true。例如，除了依赖于 join，任务控制也可以依赖跟踪任务创建和任务完成的计数器。在每一次派生新任务时计数器便递增，而在派生的任务产生一个结果时计数器递减。如果子任务通过回调而不是通过访问结果字段的方式返回任务结果，那么这个方案，以及与计数器相关的其他方案就是一个很有吸引力的选择了。这种形式的计数器只是在 § 4.4.3 讨论的关卡的小尺度、本地化的版本。

基于回调的 for/join 设计可以在诸如问题解决算法（problem-solving algorithm）、游戏、搜索，以及逻辑编程中看到。在许多这样的应用中，派生的子任务的数量是可以动态变化的，而且子任务的结果最好通过方法调用的形式获取，而不要直接使用字段获取。

而且，与诸如在 § 4.4.1.5 讨论的链接任务技术相比，基于回调的方法允许更大的异步。当子任务具有不同的生存期时，这可以带来更好的性能，因为在某些时候，对快速完成的子任务结果的处理可以与仍在进行计算的、较长的子任务重叠地进行。不过，这种设计放弃了对运行结果的任何顺序保证，因此只适用于子任务结果的处理过程与结果产生的顺序是完全独立的情况。

在接下来的 FibVCB 类中用到了回调计数器，虽然它并不是非常适合这个问题，但在这里我们只是为了举例说明这个技术。这段代码展示了一个典型的对任务的局部变量（task-local variable）、volatiles 以及锁的巧妙的组合使用，来保证任务控制开销的最小化：

java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

```
class FibVCB extends FJTask {
  // ...
  volatile int number = 0;        // as before
  final FibVCB parent;            // is null for outermost call
  int callbacksExpected = 0;
  volatile int callbacksReceived = 0;

  FibVCB(int n, FibVCB p) { number = n; parent = p; }

  // Callback method invoked by subtasks upon completion
  synchronized void addToResult(int n) {
    number += n;
    ++callbacksReceived;
  }

  public void run() {  // same structure as join-based version
    int n = number;
    if (n <= sequentialThreshold)
      number = seqFib(n);
    else {
      // Clear number so subtasks can fill in
      number = 0;
      // Establish number of callbacks expected
      callbacksExpected = 2;

      new FibVCB(n - 1, this).fork();
      new FibVCB(n - 2, this).fork();

      // Wait for callbacks from children
      while (callbacksReceived < callbacksExpected) yield();
    }

    // Call back parent
    if (parent != null) parent.addToResult(number);
  }
}
```

注意：

- 所有互斥的独占锁只限于保护字段访问的小代码段，在一个轻量级的任务框架中，任何一个类都必须遵循这一点。而且，除非可以保证任务可以很快继续，否则绝对不允许任务阻塞。特别地，这种框架最好保证同步块不能跨越 forks 以及随后的 joins 或 yield。
- 为了帮助消除一些同步，回调计数被分割成两个计数器，callbacksExpected 和 callbacksReceived。当这两个计数器的值一样时，任务就结束了。
- callbacksExpected 计数器仅被当前的线程使用，所以访问方法不必是 synchronized，当然也不需要是 volatile。事实上，由于在递归过程中，正好总是需要两个回调，而且在 run 方法之外永远也不需要这个值，所以可以简单地修改这个类以消除所有对这个变量的需要。但是在更典型的基于回调的设计中，仍然需要这样一个变量，在这种设计中，派生的任务量可能会动态地变化，并且可能在多个方法内产生。

java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

- addToResult 回调方法必须是 synchronized，以避免子任务在差不多相同的时间里回调产生的干扰问题。
- 只要 number 与 callbacksReceived 都被声明成 volatile，而且 callbacksReceived 随着 addToResult 方法中的最后一条语句被更新，那么 yield 循环测试就不必使用同步机制了，因为它在等待的是一个锁存的阈值，一旦达到阈值，它就不会再改变。
- 我们也可以定义一个改写的使用这些机制的 getAnswer 方法，从而在已经接收到所有的回调时，由它返回一个结果。但是，由于这个方法设计成由外部的（非任务的）客户程序在所有的计算都已经完成的情况下调用，所以就没有什么强制性的理由来这样做了。最初的 Fib 类的版本就已经足够用了。
- 尽管使用了上述技巧，在这个版本里，与任务控制有关的开销要比使用 coInvoke 的最初的版本要高一些。如果一定要使用它，则最好选择一个稍微大一些的串行化阈值，从而较少地利用并行性。

4.4.1.7 取消（Cancellation）

在某些设计里，并不需要保存回调计数或费尽一切力气去遍历整个子任务列表。相反，当任务的任意一个子任务（或子任务的子任务之一，等等）达到一个合适的结果时，任务就可以结束了。在这些情况下，你可以取消所有那些正在运算过程中，会产生不需要的运算结果的子任务，从而避免无用的计算。

这里的选项与在其他包含取消（参见 §3.1.2）的情况很类似。例如，子任务可以有规律地调用其父对象的一个方法（可能是 isDone），这个方法可以指出结果是不是已经找到，如果已经找到它就可以尽早返回。同时，它们也必须设置自己的状态，从而使它的任何一个子任务都可以同样地处理。这种方式可以通过使用 FJTask.cancel 实现，FJTask.cancel 只是预先设定了 isDone 的状态。这种方式可以消除还没有启动的任务的执行，但是对于正在运行过程中的任务并没有作用，除非任务的 run 方法自己检测更新的状态，并进行相应的处理。

当一整套任务都尝试去计算一个单一的结果时，一个更简单的策略便可以满足这种需要：任务可以有规律地检查一个标识任务结束的全局（static）变量。但是，在有许多任务、许多 CPU 的情况下，相对于产生这么多对同一块内存区域的访问，尤其是必须通过同步方式的访问，从而对底层系统施加了很大压力的情况，还是倾向于选择更本地化的策略。另外，需要记住，与任务取消相关的全部开销都应该少于让小任务继续运行的开销，即使这些任务产生的结果是无用的。

例如，这里有一个可以解决经典的 N 皇后（N-Queens）问题的类，在一个 NxN 的国际象棋棋盘上查找放置 N 个互相不会攻击的皇后的位置。为了展示的简单性，我们使用了一个静态的 Result 变量。这里，任务只在进入方法的时候检查取消。所以，即使已经找到了答案，它们也会继续在可能的执行路线上循环。但是，刚刚被产生的任务会立即退出。这可能有一点浪费，但是与在任务每一次迭代时都检查是否完成的方案相比，这样可以得到一个更快的解决方案。

这里同样需要注意的是，任务没有连接（join）它们的子任务，因为并没有理由如此做。

java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

只有最外部（main 中）的调用会需要等待一个结果；在这里，是通过在 Result 类里加入一个标准的等待、通知方法，从而可以支持这种方式的（另外，为了简洁的目的，这个版本并没有使用任何形式的颗粒度阈值。但是要加入一个非常容易，例如当行数接近于边界尺寸时，可以通过直接搜索移动，而不必派生子任务）。

```
class NQueens extends FJTask {
  static int boardSize; // fixed after initialization in main
  // Boards are arrays where each cell represents a row,
  // and holds the column number of the queen in that row

  static class Result {           // holder for ultimate result
    private int[] board = null;   // non-null when solved

    synchronized boolean solved() { return board != null; }

    synchronized void set(int[] b) { // Support use by non-Tasks
      if (board == null) { board = b; notifyAll(); }
    }

    synchronized int[] await() throws InterruptedException {
      while (board == null) wait();
      return board;
    }
  }
  static final Result result = new Result();
  public static void main(String[] args) {
    boardSize = ...;
    FJTaskRunnerGroup tasks = new FJTaskRunnerGroup(...);
    int[] initialBoard = new int[0]; // start with empty board
    tasks.execute(new NQueens(initialBoard));
    int[] board = result.await();
    // ...
  }

  final int[] sofar;              // initial configuration

  NQueens(int[] board) { this.sofar = board;  }

  public void run() {
    if (!result.solved()) {       // skip if already solved
      int row = sofar.length;

      if (row >= boardSize)       // done
        result.set(sofar);

      else {                      // try all expansions

        for (int q = 0; q < boardSize; ++q) {

          // Check if queen can be placed in column q of next row
          boolean attacked = false;
          for (int i = 0; i < row; ++i) {
            int p = sofar[i];
            if (q == p || q == p - (row-i) || q == p + (row-i)) {
              attacked = true;
```


java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

```
            break;
          }
        }

        // If so, fork to explore moves from new configuration
        if (!attacked) {
          // build extended board representation
          int[] next = new int[row+1];
          for (int k = 0; k < row; ++k) next[k] = sofar[k];
          next[row] = q;
          new NQueens(next).fork();
        }
      }
    }
  }
}
}
```

4.4.2 计算树

许多计算密集的算法都包含下列形式的任务：

```
对于数量固定的步骤，或直到收敛（convergence），执行{
  更新问题的一个区域;
  等待其他任务完成更新它们的区域;
}
```

大多数情况下，这样的算法在分区的数组、矩阵或图像表示上执行更新操作。例如，许多物理上的力学问题都包含对矩阵单元的重复地局部更新。Jacobi 算法和相关的张弛技术重复地重算邻近单元的估计值，典型情况下，这些技术使用一个如下所示的求平均公式。

```
void oneStep(double[][] oldMatrix,
             double[][] newMatrix, int i, int j) {
   newMatrix[i][j] = 0.25 * (oldMatrix[i-1][j] +
                             oldMatrix[i][j-1] +
                             oldMatrix[i+1][j] +
                             oldMatrix[i][j+1]);
 }
```


java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

通常，为了节省空间，会把两个不同的矩阵 newMatrix 和 OldMatrix 在连续的步骤中反复交换。

需要所有任务周期性地等待所有其他任务完成的算法，并不会像更加低耦合的 fork/join 设计那样可以缩放自如。即使这样，这些算法也是常见的、有效的而且经得住考验的、能够显著提高并行计算速度的方法。

4.4.2.1 创建和使用树

为了并行地更新区域，而在迭代设计中重复地使用 fork/join 分解技术可能是很没有效率的。因为在迭代的过程中，区域是相同的，它们可以只被构造一次，并重复地被调用，这样，在每一次迭代过程中，相关的更新就可以按照与由一个递归方法产生的相同的顺序执行。

计算树清晰地表示了在 fork/join 递归过程中隐含地出现的树结构的计算。相对于递归操作和基本操作，这些树有两种节点，内部节点与叶子节点。通过下面的步骤，它们可以被构造和用于迭代更新问题：

1. 创建一个代表递归分区的任务对象的树，在这棵树上：
   - 每一个内部节点都包含对子分区的引用，并且有一个执行每一个子分区的 fork/join 处理的更新方法。
   - 每一个叶子节点代表了一个最细颗粒度的分区，并具有一个能够直接操作它的更新方法。
2. 对于固定数量的步骤，或直到叶子节点，做下面的操作：
   - 执行处理根分区更新方法的任务。

例如，通过使用上面介绍过的求平均公式，接下来的代码展示了一系列执行 Jacobi 迭代的类的重点。除了更新之外，这个版本还跟踪在两次迭代之间单个元素值的差别，如果最大的差别值小于常量 EPSILON 时，计算就停止了。另外，与许多这个形式的程序类似，这个代码也假定矩阵的外部有不被修改的边界单元，所以永远都不需要检查边界条件（其他的选择包括，在边界处使用特殊的边界公式重新计算边界值，并把边界值作为矩阵周围的元素的值）。

这里使用的递归分解策略是将矩阵分割到不同的象限，当单元的数量至多达到 leafCells 时，分割就停止。在这里，leafCells 作为颗粒度的阈值。如果矩阵中行和列的数量差不多相等时，这个策略就可以工作得很好。如果不是这样、那么就需要定义额外的类和方法，每次只在一个维度上进行分割。这里的方法假定作为一个整体矩阵已经存在，因此，与其实际地分割矩阵的单元，还不如让任务节点只是跟踪矩阵中正在工作的每个分区的行和列的偏移。

这里使用的基于子类的设计反映了内部节点与叶子节点不同的结构和行为。它们两个都是抽象类 JTree 的子类：

```
abstract class JTree extends FJTask {
  volatile double maxDiff; // for convergence check
}

class Interior extends JTree {
```


java学习群：72030155, 每天20:30-23:00都有大神视频教学, 想学习的同学可以进群免费听课！

```
    private final JTree[] quads;

    Interior(JTree q1, JTree q2, JTree q3, JTree q4) {
      quads = new JTree[] { q1, q2, q3, q4 };
    }

    public void run() {
      coInvoke(quads);
      double md = 0.0;
      for (int i = 0; i < quads.length; ++i) {
        md = Math.max(md,quads[i].maxDiff);
        quads[i].reset();
      }
      maxDiff = md;
    }
  }

  class Leaf extends JTree {
    private final double[][] A; private final double[][] B;
    private final int loRow;     private final int hiRow;
    private final int loCol;     private final int hiCol;
    private int steps = 0;

    Leaf(double[][] A, double[][] B,
         int loRow, int hiRow, int loCol, int hiCol) {
      this.A = A;    this.B = B;
      this.loRow = loRow; this.hiRow = hiRow;
      this.loCol = loCol; this.hiCol = hiCol;
    }

    public synchronized void run() {
      boolean AtoB = (steps++ % 2) == 0;
      double[][] a = (AtoB)? A : B;
      double[][] b = (AtoB)? B : A;
      double md = 0.0;
      for (int i = loRow; i <= hiRow; ++i) {
        for (int j = loCol; j <= hiCol; ++j) {
          b[i][j] = 0.25 * (a[i-1][j] + a[i][j-1] +
                            a[i+1][j] + a[i][j+1]);
          md = Math.max(md, Math.abs(b[i][j] - a[i][j]));
        }
      }
      maxDiff = md;
    }
  }
```

driver 类首先构建了一个代表其参数矩阵的分区的树。构建方法自己可以被并行化。但是由于这里的基本操作只是节点构造，所以，应该把颗粒度阈值设置得足够高，从而在对于巨大矩形的问题时，才值得使用并行化。

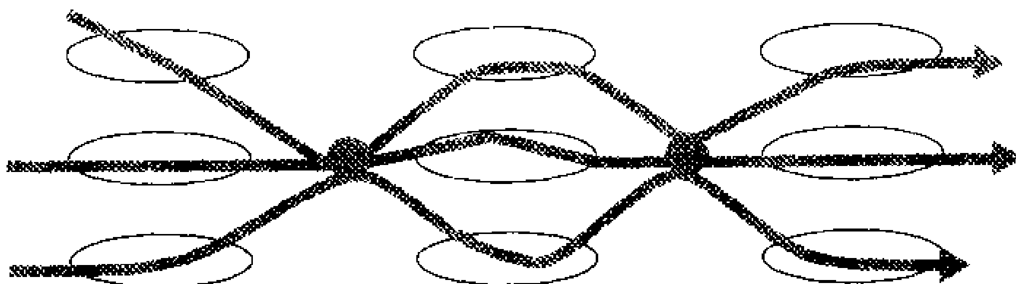
Run 方法重复地启动根任务，并等待它结束。为了展示的简单性，在这里直到收敛时它才会结束。在将这个例子转变成一个实际的程序时，除了可能会用到的其他所需改变中，你可能还需要初始化矩阵，以及限定最大迭代次数，以处理运算不收敛的情况。由于每一次迭代都需要一个等待根任务结束的完整的同步点，因此，插入一个维护或报告迭代之间的全局状态的额外操作，相对来说是比较简单的。

java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

```
class Jacobi extends FJTask {
  static final double EPSILON = 0.001; // convergence criterion
  final JTree root;
  final int maxSteps;
  Jacobi(double[][] A, double[][] B,
         int firstRow, int lastRow, int firstCol, int lastCol,
         int maxSteps, int leafCells) {
    this.maxSteps = maxSteps;
    root = build(A, B, firstRow, lastRow, firstCol, lastCol,
                 leafCells);
  }

  public void run() {
    for (int i = 0; i < maxSteps; ++i) {
      invoke(root);
      if (root.maxDiff < EPSILON) {
        System.out.println("Converged");
        return;
      }
      else root.reset();
    }
  }

  static JTree build(double[][] a, double[][] b,
                     int lr, int hr, int lc, int hc, int size) {
    if ((hr - lr + 1) *  (hc - lc + 1) <= size)
      return new Leaf(a, b, lr, hr, lc, hc);
    int mr = (lr + hr) / 2; // midpoints
    int mc = (lc + hc) / 2;
    return new Interior(build(a, b, lr,   mr, lc,   mc, size),
                        build(a, b, lr,   mr, mc+1, hc, size),
                        build(a, b, mr+1, hr, lc,   mc, size),
                        build(a, b, mr+1, hr, mc+1, hc, size));
  }
}
```

4.4.3 关卡（barrier）

递归分解是一种比较强大、灵活的技术，但是它并不总是适用于迭代问题的结构，为了得到高效的实现，通常需要采用轻量级的执行框架。解决大多数迭代问题的更直接的方式是，首先将问题分解成不同的段（*segment*），与每个段相关联的任务执行一个循环，该循环必须周期性地等待其他段完成。从基于树的方法的角度看，这种设计消除了所有内部节点，从而只需要处理叶子节点。



java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

在递归的任务中，我们可以根据需要处理线程，从而使它更适应并行迭代的要求（参见进阶阅读）。然而，这样做通常不会有什么收益，部分原因是，所有的线程构造的开销都集中在启动阶段。在这里，我们举例说明使用每个只执行一个 Runnable 对象的标准线程的基本机制。总的来说，在使用线程时，颗粒度阈值必须要比使用轻量级可执行类时要高很多（但是仍然要比在分布式并行设计方式中所需的要低得多）。但是在不考虑粒度的情况下，迭代算法的基本逻辑是完全相同的。在许多迭代问题中，由于使用了粗粒度阈值，一些潜在的并行性被浪费了。当所有的线程在近似相同的时间里执行类似相同的操作时，只创建与 CPU 个数相同多的任务，或者是 CPU 个数的很小倍数的任务，系统可以工作得很好。

虽然，通过使用等待和通知结构，手工地创建必要的控制机制也是可行的，但是，依赖于封装关卡机制的标准同步辅助方法会更加方便，而且不易出错。在迭代设计中选择的同步设备是一个周期性关卡（cyclic barrier）。周期性关卡在初始化时使用固定数量的需要重复的被同步的成员。它只提供一个方法，barrier，这将强迫每一个调用程序必须等待到它所有的成员都调用了这个方法，然后复位进行下一次迭代。一个基本的 CyclicBarrier 类可以定义成如下形式：

```
class CyclicBarrier {

  protected final int parties;
  protected int count;      // parties currently being waited for
  protected int resets = 0;  // times barrier has been tripped

  CyclicBarrier(int c) { count = parties = c; }

  synchronized int barrier() throws InterruptedException {
    int index = --count;
    if (index > 0) {         // not yet tripped
      int r = resets;        // wait until next reset

      do { wait(); } while (resets == r);

    }
    else {                   // trip
      count = parties;       // reset count for next time
      ++resets;
      notifyAll();           // cause all other parties to resume
    }

    return index;
  }
}
```

（可以从在线附录中得到这个类的 util.concurrent 版本，它可以更可靠地处理中断和超时。在具有大量处理器的系统上构造可以减少在锁和成员变量上的内存争用的更复杂的版本，这是非常值得的。）

这里定义的 CyclicBarrier.barrier 方法，将进入 barrier 方法时，其他仍然在等待的线程的数量返回。在一些算法中，这个方法可能是很有用的。作为另外一个副产品，barrier 方法本

java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

质上是同步的，因此也可以作为一个内存的关卡来确保在大多数典型用法的环境里正确地刷新和载入数组元素的值（参见 § 2.2.7）。

barrier 也可以构造成一个简单的共识操作符（参见 § 3.6）。它收集多个线程关于是否继续进行到下一个迭代的“投票”。当收集到所有的投票并达成一致时，就会释放同步（但是，不像事务处理框架，使用 CyclicBarrier 类的线程是不允许投“反对票”的）。

通过使用 barriers，可以很容易地表示许多迭代算法。在最简单的情况下，程序可以直接套用下面的格式（这里省略了所有特殊问题的细节）。

```
class Segment implements Runnable  {            // Code sketch
  final CyclicBarrier bar; // shared by all segments
  Segment(CyclicBarrier b, ...) { bar = b; ...; }

  void update() { ... }

  public void run() {
    // ...
    for (int i = 0; i < iterations; ++i) {
      update();
      bar.barrier();
    }
    // ...
  }
}

class Driver {
  // ...
  void compute(Problem problem) throws ... {
    int n = problem.size / granularity;
    CyclicBarrier barrier = new CyclicBarrier(n);
    Thread[] threads = new Thread[n];

    // create
    for (int i = 0; i < n; ++i)
      threads[i] = new Thread(new Segment(barrier, ...));

    // trigger
    for (int i = 0; i < n; ++i) threads[i].start();

    // await termination
    for (int i = 0; i < n; ++i) threads[i].join();
  }
}
```

这个结构可以用来解决大多数已知迭代次数的问题。然而，许多问题需要检查收敛性或者迭代之间的一些全局属性［相反，在一些混乱松弛算法（chaotic relaxation algorithm）中，在每次迭代之后，你甚至不需要用到关卡，但是你可以让程序段在多个关卡和/或者条件检查之间自由运行一段时间］。

一种提供收敛性检查的方法是修改 CyclicBarrier 类，使得无论在什么时候需要重置一个关卡时，都可以可选地运行一个补充的 Runnable 命令。更经典的方法是依赖 barrier 返回的索引值，这种方法展示的技术在其他情况下也有效。只有获得索引值为 0 的调用程序（这个

java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

值可以任意指定，但必须是合法的）能够执行检查，而所有其他调用者只能等待第二个关卡。

例如，这里有一个基于关卡的类的版本节选，它可以用来解决§4.4.2 所描述的 Jacobi 问题。前面给出的通用形式的分割者可以初始化和运行 JacobiSegment 对象的集合。

```
class JacobiSegment implements Runnable {        // Incomplete
  // These are same as in Leaf class version:
  double[][] A;       double[][] B;
  final int firstRow;  final int lastRow;
  final int firstCol;  final int lastCol;
  volatile double maxDiff;
  int steps = 0;
  void update() { /* Nearly same as Leaf.run */ }

  final CyclicBarrier bar;
  final JacobiSegment[] allSegments; // for convergence check
  volatile boolean converged = false;

  JacobiSegment(double[][] A, double[][] B,
                int firstRow, int lastRow,
                int firstCol, int lastCol,
                CyclicBarrier b, JacobiSegment[] allSegments) {
    this.A = A;   this.B = B;
    this.firstRow = firstRow; this.lastRow = lastRow;
    this.firstCol = firstCol; this.lastCol = lastCol;
    this.bar = b;
    this.allSegments = allSegments;
  }

  public void run() {
    try {
      while (!converged) {
        update();
        int myIndex = bar.barrier(); // wait for all to update
        if (myIndex == 0) convergenceCheck();
        bar.barrier();               // wait for convergence check
      }
    }
    catch(Exception ex) {
      // clean up ...
    }
  }

  void convergenceCheck() {
    for (int i = 0; i < allSegments.length; ++i)
      if (allSegments[i].maxDiff > EPSILON) return;
    for (int i = 0; i < allSegments.length; ++i)
      allSegments[i].converged = true;
  }
}
```

4.4.4 进阶阅读

要全面了解高性能并行处理的方法，请阅读：

Skillicorn, David 和 Domenico Talia, “Models and Languages for Parallel Computation”,

java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

《Computing Surveys》, June 1998.

虽然大多数关于并行编程的文字都着重强调为细粒度的并行机结构所设计的算法，但是它们也覆盖了使用支持一个 JVM 的多处理器来实现的设计技巧和算法，例如，见：

Foster 和 Ian.《Designing and Building Parallel Programs》, Addison Wesley, 1995.

Roosta 和 Seyed.《Parallel Processing and Parallel Algorithms》, Springer-Verlag, 1999.

Wilson 和 Gregory.《Practical Parallel Programming》, MIT Press, 1995.

Zomaya 和 Albert (ed.).《Parallel and Distributed Computing Handbook》, McGraw-Hill, 1996.

基于模式的并行编程书籍包括：

Massingill, Berna, Timothy Mattson 和 Beverly Sanders.《A Pattern Language for Parallel Application Programming》, Technical report, University of Florida, 1999.

MacDonald, Steve, Duane Szafron 和 Jonathan Schaeffer. "Object-Oriented Pattern-Based Parallel Programming with Automatically Generated Frameworks", in《Proceedings of the 5th USENIX Conference on Object-Oriented Tools and Systems (COOTS) 》, 1999.

FJTask 框架内部依赖一种称为"工作窃取"(work-stealing) 的任务调度者，它基于 Clik 中的任务调度者。Clik 是一个基于 C 语言的并行编程框架。在"工作偷窃"调度者中，正常情况下，每一个工作者线程只运行（按照 LIFO 的顺序）它自己构造的任务，但是在空闲时，它会偷窃（按 FIFO 的顺序）其他工作者线程构造的任务。更详细一些内容，包括为什么对于递归 fork/join 程序而言，这种方案是最佳的解释，可以在下面的读物中找到：

Frigo, Matteo, Charles Leiserson 和 Keith Randall. "The Implementation of the Cilk-5 Multithreaded Language",《Proceedings of 998 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI) 》, 1998.

在线附录中包括了一些这一节里讨论过的技术的更实际的例子。它也提供了到 Clik 包以及相关框架的链接，包括 Hood（一个 C++实现的 Clik）和 Filaments（一个包含支持基于关卡的迭代计算的特定框架的 C 语言的包）。

4.5 活动对象

在遍及几乎整个这一章所讲述的基于任务的框架中，线程主要用于在被动对象之间激发活动消息。但是，如果从相反的角度上解决某些设计问题的话——由活动对象彼此发送被动消息，这种方式也是非常有效的。

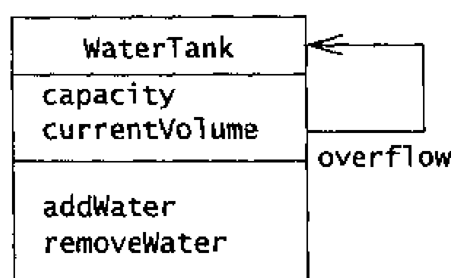
为了展示这个，考虑一个符合在第 1 章中描述的 WaterTank 的活动对象：

```
pseudoclass ActiveWaterTank extends Thread {        // Pseudocode
  // ...
  public void run() {
    for (;;) {
      accept message;
      if (message is of form addWater(float amount)) {
        if (currentVolume >= capacity) {
```


java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

```
            if (overflow != null) {
              send overflow.addWater(amount);
              accept response;
              if (response is of form OverflowException)
                reply response;
              else ...
            else ...
          else ...
        }
        else if (message is of form removeWater(float amount)) {
          ...
        }
      }
    }
  }
```

在这里使用伪码，是因为对于从一个活动对象向另一个活动对象发送消息，没有内建的语法支持，而只有在被动对象之间直接调用的语法。但是，如同在§4.1.1讨论过的，即使在使用被动对象时，也会遇到类似的问题。在那一节所描述的任何一种解决方案在这里也同样有效：从各种消息格式中选择一种，通过流、通道、事件队列、管道及套接字等方式传输消息。事实上，如同本章开始时的 WebService 例子中展示的，向设计中添加基于任务的构造要比添加基于活动对象的构造要容易。相反地，当某些对象是活动的而不是被动的时，在本章讨论的大多数基于任务的设计都可以同样工作得很好。



进而，把 Runnable 作为消息使用会带来一种烦人但是通用（至少在某种程度上）的活动对象的形式：一个普通的工作者线程设计的较小变体，这种设计符合在§1.2.4 解释过的活动对象的初始抽象特性，如下所示：

```
class ActiveRunnableExecutor extends Thread {
  Channel me = ... //  used for all incoming messages

  public void run() {
    try {
      for (;;) {
        ((Runnable)(me.take())).run();
      }
    }
    catch (InterruptedException ie) {} // die
  }
}
```

当然，这样的类并不十分有用，除非它也包括内部方法，可以产生 Runnable 来执行和（或）发送给其他的活动对象。以这种方式编写整个程序是可能的，但并不自然。

但是，在响应系统中的许多组件可以被有效地构造成在更受限的规则和消息传递约定下工作的活动对象。特别地，这也包括了那些与其他计算机或设备交互的对象，通常，它们是

java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

一个程序中主要的外部可见对象。

在诸如 CORBA 和 RMI 的分布式框架中，外部可见的活动对象本书就列出了它可以接收的消息的接口。它们内部通常有一个比 ActiveWaterTank 更统一的结构。它们通常包含了一个主要的 run 循环，重复地接收外部请求，并将它们分发给内部提供相应服务的被动对象，然后构造返回给客户端的回应消息（当使用 CORBA 和 RMI 时，内部的被动对象在编程时要明确地编写代码。而活动对象，有时候也被称为"skeleton"，通常是由工具自动产生的）。

采用活动的、参与者风格的方法设计其他组件也是非常可能的。采用这种方式设计整个系统的一个原因是：可以利用与活动实体和它们的消息有关的特定规则集相关的已经开发得很好的理论和设计技巧。这一节剩余的部分给出了一个最著名、最有影响力的这样的框架——CSP 的简短概述。

4.5.1 CSP

C.A.R. Hoare 的"通信串行过程"（CSP）理论，既提供了处理并发的常规方法，也提供了相关的设计技巧。如同在 § 4.5.2 的进阶阅读中所讨论的那样，在并发设计和编程中，虽然存在大量紧密相关的方法，但是 CSP 具有最大的影响。CSP 已经成为某些编程语言的基础（包括 occam），并且在其他编程语言（包括 Ada）的设计中也是很有影响力的，在 Java 编程语言中可以通过使用一些库来支持 CSP。

接下来的部分展示了由 Peter Welch 和他的同事们开发的 JCSP 包。可以通过在线附录上的链接得到这个包。这一节只是提供了一个简短的纲要。有兴趣的读者可能想要得到开发包的拷贝、文档以及相关的文字介绍。

4.5.1.1 过程和通道（Process and channel）

CSP 过程可以以一种特殊的参与者风格的对象来创建，在其中：

- 过程没有任何方法接口和外部可调用的方法。由于没有任何可调用的方法，所以就不可能让其他的线程调用方法。这样就不需要明确的锁。
- 过程只通过通道读写数据进行通信。
- 过程没有标识，所以不能被显示地引用。但是，可以使用通道作为引用的模拟（参见 § 1.2.4），以允许与通道另一端的任何过程进行通信。
- 过程不需要在一个接收消息的循环中一直运行（尽管很多情况下是这样做的）。它们可以在所需的不同通道上读写消息。

CSP ***通道（channel）***以一种特殊的 Channel 来创建，在其中：

- 所有的通道都是同步的（参见 § 3.4.1.4），并且因此不包含任何内部的缓冲（但是，你也可以构造执行缓冲的过程）。
- 通道只支持携带数据值的读（"?"）和写（"!"）操作。操作行为同 take 和 put 类似。
- 最基本的通道是一对一的。它只能被连接到一对过程上，一个是读过程，一个是写过程。当然，也可以定义连接多个读过程和多个写过程的通道。

java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

4.5.1.2 组合（Composition）

CSP 的优雅主要来源于它简单、容易分析和处理的组合规则。CSP 中的“S”代表串行化的，所以基本的过程在内部数据上执行串行化计算（例如，添加数字、条件测试、赋值，以及循环）。高级的过程是通过组合创建的；对于一个通道 c、变量 x，以及过程 P 和 Q：

c?x -> P	从 C 中读取数据使 P 可以运行。
c!x -> P	向 C 中写入数据使 P 可以运行。
P ; Q	P 跟随在 Q 之后。
P || Q	P 和 Q 并行。
P [ ] Q	P 或 Q（但不是两者都有）。

选择操作符 P[]Q 要求 P 和 Q 两者都是可以通信的过程（具有 d?y -> R 或 d!y -> R 的形式）。选择运行哪一个过程依赖于哪一个通信准备就绪。如果其中之一已经（或即将）准备就绪，那么就选择这条分支。如果两个分支都已经（或即将）准备就绪，那么两者都有可能（这是不可确定的）。

4.5.1.3 JCSP

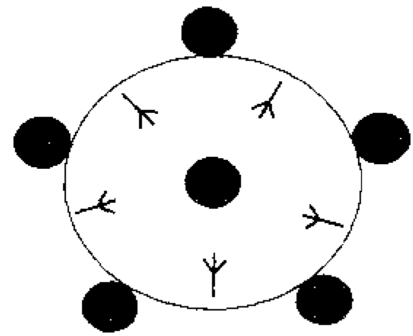
JCSP 包以一种很直接的方式支持基于 CSP 的设计。它由一个执行框架组成，通过接口、类和方法所代表的构造，这个框架可以有效地支持 CSP，这些接口、类和方法包括：

- 接口 ChannelInput（支持 read），ChannelOutput（支持 write），以及 Channel（既支持读又支持写）使用 Object 作为参数，但是也提供使用 int 参数的特殊版本。主要的实现类是 One2OneChannel，它支持一个单一读过程和一个单一写过程的用法。但是也有提供多路通道的不同的类。
- 接口 CSProcess 描述只提供 run 方法的过程。实现类 Parallel 和 Sequence（以及其他的类）的构造器可以接收其他 CSProcess 对象数组并创建组合。
- 通过 Alternative 类支持选择操作符[]。它的构造器接收元素类型为 Guard 的数组。Alternative 类提供一个 select 方法，这个方法返回哪一个元素可以被选择（随后必须要选择）的索引标志。fairSelect 方法以同样的方式工作，但是它提供额外的公正性保证——在多次选择的过程中，它会从所有可选的选择中选择，而不总是固定地选择其中的某一个。下面展示了 Alternative 类的惟一用法使用了 guard 类型 AltingChannelInput，它通过 One2OneChannel 类实现。
- 额外的工具包括各种 CSProcess 的实现，例如 Timer（执行延迟的写，并可以在 Alternative 类超时处理）、Generate（产生顺序数字），Skip（实际上并不做什么事情——只是 CSP 的基本元素之一），以及通过 AWT 实现迭代和显示的类。

4.5.1.4 哲学家进餐问题（Dining philosophers）

作为一个经典的例子，考虑一下著名的“哲学家进餐问题”（Dining philosophers）问题。一张桌子上放着五把叉子（如图所排列）和一碗意大利面条。桌子边上坐着五个哲学家，他

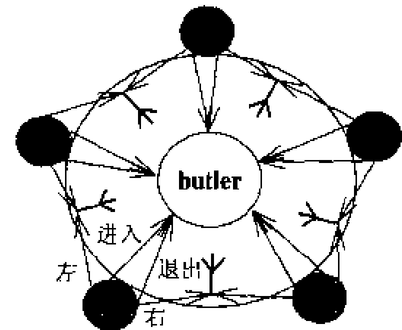
java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！



们中的每一个人都只能吃一会儿，然后思考一会儿，然后再吃，依此类推。在吃面条时，每个哲学家需要两把叉子，左边一把，右边一把（没有人知道为什么：这只是故事中的一部分），而当他们开始思考时就必须放下叉子。

这里要解决的主要问题是：如果不采用一些协调的手段，当所有的哲学家都拿起他们的左手叉子时，他们就会永远阻塞在那里等待被别的哲学家拿在手里的叉子，这样他们就会饿死。

有许多解决这个问题的方法（然而也有更多无解的方法）。我们会演示一个 Hoare 描述的例子，这个例子加入了这样一个要求［通过 Butler（注：巴特勒，1862—1947 年，美国教育家、政治家，曾获 1931 年诺贝尔和平奖）强加的］：在任何给定的时间，至多有四位哲学家允许就座。这个要求足以保证，在任何时候至多有一个哲学家可以吃——如果只有四个哲学家，他们中至少有一个人可以拿到两只叉子。这种做法并不是通过方法本身保证五位哲学家最终可以吃到东西。但是可以通过使用 Butler 类中的 Alternative.fairSelect 方法得到这种保证，这个方法可以保证公正地处理就座消息。



我们会使用一个简单的、纯粹 CSP 的风格，其中，所有的通道都是一对一的，并且消息中没有任何内容（使用 null 作为消息体）。这种方式需要着重注意同步和过程构造问题。系统由具有五个 Philosopher 的 College、五个 Fork，以及一个 Butler（站在意大利面条的碗中）组成，使用 One2OneChannel 进行连接。

由于所有东西都必须是过程或通道，因此，叉子必须是过程。fork 持续地循环等待从它的使用者（它左边的哲学家或右边的哲学家）那里传来的消息。当它得到一个指示叉子拿起来的消息时，它就等待另一个指示叉子落下去的消息（虽然通过不同种类的消息或消息内容指示叉子起来与落下，显得更真实些，但是这里只是使用 null 消息就足够满足需要了）。

在 JCSP 中，可以写成如下所示的样子：

```
class Fork implements CSProcess {

  private final AltingChannelInput[] fromPhil;

  Fork(AltingChannelInput l, AltingChannelInput r) {
    fromPhil = new AltingChannelInput[] { l, r };
  }

  public void run() {
    Alternative alt = new Alternative(fromPhil);

    for (;;) {
      int i = alt.select();   // await message from either
      fromPhil[i].read();     // pick up
```


java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

```
        fromPhil[i].read();      // put down
      }

    }
  }
```

Butler 过程确保在任何给定时间里，至多有 N-1 个哲学家就座（在这里就是 4 个）。它通过下面的方式做到这一点：如果有足够的位子，就使 enter（进入）和 exit（退出）消息都可用，如果不是，则只有 exit 消息可用。由于 Alternative 类负责处理选项数组，因此它需要一些处理来做到这一点（JCSP 中的某些工具可以用来简化这些操作）。在通道数组 chans 中的 exit 通道放置在 enter 通道之前，这样不管使用哪一个 Alternative，都可以从正确的通道中读。这里使用 fairSelect 是为了保证不会是同样的四个哲学家一直坐在那里吃面条，而让第五个哲学家一直在那里尝试进入。

```
class Butler implements CSProcess {

  private final AltingChannelInput[] enters;
  private final AltingChannelInput[] exits;

  Butler(AltingChannelInput[] e, AltingChannelInput[] x) {
    enters = e;
    exits = x;
  }

  public void run() {
    int seats = enters.length;
    int nseated = 0;

    // set up arrays for select
    AltingChannelInput[] chans = new AltingChannelInput[2*seats];
    for (int i = 0; i < seats; ++i) {
      chans[i] = exits[i];
      chans[seats + i] = enters[i];
    }

    Alternative either = new Alternative(chans);
    Alternative exit = new Alternative(exits);

    for (;;) {
      // if max number are seated, only allow exits
      Alternative alt = (nseated <  seats-1)? either : exit;

      int i = alt.fairSelect();
      chans[i].read();

      // if i is in first half of array, it is an exit message
      if (i < seats) --nseated; else ++nseated;
    }
  }
}
```

Philosopher 过程在循环里一直运行，在思考和吃面条之间交替。在吃面条之前，哲学家

java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

必须首先进入他们的座位，然后拿到两只叉子。在吃完之后，则必须做相反的事情。eat（吃）和 think（思考）方法在这里都是空操作，但是可以填充实现它，通过将 JCSP 的通道和过程的状态通过接口报告给 AWT 可以实现一个动态演示的版本。

```
class Philosopher implements CSProcess {

  private final ChannelOutput leftFork;
  private final ChannelOutput rightFork;
  private final ChannelOutput enter;
  private final ChannelOutput exit;

  Philosopher(ChannelOutput l, ChannelOutput r,
              ChannelOutput e, ChannelOutput x) {
    leftFork = l;
    rightFork = r;
    enter = e;
    exit = x;
  }

  public void run() {

    for (;;) {

      think();

      enter.write(null);          // get seat
      leftFork.write(null);       // pick up left
      rightFork.write(null);      // pick up right

      eat();

      leftFork.write(null);       // put down left
      rightFork.write(null);      // put down right
      exit.write(null);           // leave seat

    }

  }

  private void eat() {}
  private void think() {}
}
```

最后，我们创建一个 College 类，由它代表叉子、哲学家与 Butler 的并行组合。可以使用由 JCSP 提供的一个方便的函数 create 来构造通道，create 函数可以创建通道数组。Parallel 构造器接收一个 CSProcess 的数组，它在开始时载入所有的参与者。

```
class College implements CSProcess {
  final static int N = 5;

  private final CSProcess action;

  College() {
    One2OneChannel[] lefts = One2OneChannel.create(N);
```


java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

```
    One2OneChannel[] rights = One2OneChannel.create(N);
    One2OneChannel[] enters = One2OneChannel.create(N);
    One2OneChannel[] exits = One2OneChannel.create(N);

    Butler butler = new Butler(enters, exits);

    Philosopher[] phils = new Philosopher[N];
    for (int i = 0; i < N; ++i)
      phils[i] = new Philosopher(lefts[i], rights[i],
                                 enters[i], exits[i]);

    Fork[] forks = new Fork[N];
    for (int i = 0; i < N; ++i)
      forks[i] = new Fork(rights[(i + 1) % N], lefts[i]);

    action = new Parallel(
      new CSProcess[] {
        butler,
        new Parallel(phils),
        new Parallel(forks)
      });
  }

  public void run() { action.run(); }

  public static void main(String[] args) {
    new College().run();
  }
}
```

4.5.2 进阶阅读

对于可以被有效地表达成无标识、无接口的通过同步通道进行通信的过程的有限集合的系统，CSP 已经被证明是一种系统设计和分析的成功方式，CSP 是在下面这本书中引入的：

Hoare 和 C. A. R.《Communicating Sequential Processes》, Prentice Hall, 1985.

一个更新后的版本见：

Roscoe 和 A. William.《The Theory and Practice of Concurrency》, Prentice Hall, 1997.

第 1 章（包括在 § 1.2.5.4 中 Burns and Welling 所著的书）中列出的几段文字在描述 occam 和 Ada 语言中的构造的过程中讨论过 CSP。

其他相关的形式方法、设计技巧、语言，以及框架采用了不同的基本假设，这些基本假设与其他并发系统的特性和（或）不同的分析风格紧密相关。这些方法包括 Milner 的 CCS 和π-calculus，以及 Berry 的 Esterel，见：

Milner 和 Robin.《Communication and Concurrency》, Prentice Hall, 1989.

Berry 和 Gerard. "The Foundations of Esterel", in Gordon Plotkin, Colin Stirling, and Mads Tofte (eds.),《Proof, Language and Interaction》, MIT Press, 1998.

对这些以及与并发系统设计相关的方法，随着包支持变得可用，在从概念上被看作是活

java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

动对象的集合的系统的开发中，相对于直接使用基于线程的构造而言，这些方法也成为一种有吸引力的选择。例如：***Triveni*** 是一个部分基于 Esterel 的方法，在下面的书中有这方面的描述：

Colby, Christopher, Lalita Jategaonkar Jagadeesan, Radha Jagadeesan, Konstantin Läufer 和 Carlos Puchol. "Objects and Concurrency in Triveni: A Telecommunication Case Study in Java", 《USENIX Conference on Object-Oriented Technologies and Systems (COOTS)》，1998.

Triveni 通过 Java 编程语言包支持（请见在线附录）。它与 CSP 的主要差别是：Triveni 中的活动对象通过发送事件进行通信。Triveni 也提供在接收到事件时的与活动中断和挂起相关的计算和组合规则，尤其是在实时设计环境中，这些规则提高了 Triveni 的表达能力。

java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！