前言

1995年冬天，所有程序员都注意到了Sun公司发布的第一个Java™ alpha版。Java有很多吸引程序员的特性（不仅仅限于Sun公司对Java的宣传）：Java是健壮的、安全的、与体系结构无关的、可移植的、面向对象的、简单的和多线程的。对于大多数程序员而言，最后两点看起来是矛盾的：一种多线程的语言能够简单吗？

实际上Java的线程系统是简单的，至少和其他的线程系统相比是这样的。Java线程系统的简单性使得它相当容易学习，即使不熟悉线程的程序员也可以轻松地掌握线程编程的基础知识。但是这种简单性是要付出代价的：Java中没有其他线程系统中的高级功能。但是程序员可以使用Java提供的简单功能来实现这些高级功能。这也正是本书的主题：如何使用Java提供的线程工具来进行简单的线程编程，以及如何在更复杂的程序中通过扩展这些工具来完成更高级的任务。

本书的读者对象

本书适合于那些希望学习在Java程序中使用线程的各种级别的程序员阅读。前几章讨论使用Java进行线程编程的问题，在这几章中，假设程序员不具有线程编程经验，因此处于基本级别。后面几章的级别要高一些，这是由此时使用的材料和对程序员经验所做的假设而决定的。那些没有线程编程经验的程序员应该按自然顺序阅读。

这种方法模拟了Java自身以及Java书籍的发展过程。早期的Java程序尽管有效，但是趋向于简单化：在Web页面上一个可以跳舞的动画效果是展示Java能力的最好广告，但是只表现了Java的皮毛。同样，关于Java的早期书籍也是趋向于提供对Java的全面说明，而只用一到两章来讲述Java的线程系统。

本书属于另一种类型的Java书籍：它仅仅关注一个主题，因此可以详细地解释Java线程系统的细节。本书读者的目标应该是编写更复杂的程序，以完全使用Java线程系统的能力。

尽管本书使用的材料不需要你原来就了解线程，但还是假设读者对于Java API方面的其他知识有一定的了解，并且可以编写简单的Java程序。

本书使用的版本

在Internet时代编写一本关于Java的书十分困难：书的基石是在不停地变化的。因此我们要选定一个基石，而这个基石就是Sun公司的JDK™ 2。也许在本书中没有包含Java 2以后的版本对于线程系统的一些改变。在本书中我们也会随时指出Java 2和以前版本的区别，因此使用Java 2以前版本的程序员也可以使用本书。

一些Java（包括嵌入浏览器中的和作为开发系统的）提供商都在为向Java线程系统提供一些附加的功能（这类似于我们在第五章到第八章通过Java线程系统的基本技术来提供一些附加功能）而互相竞争。这些扩展超出了本书的讨论范围：我们只关心Sun公司的JDK 2。只有当JDK在Unix平台和Windows平台上不一样时，我们才考虑这些平台的区别。我们将在第六章讨论在这些平台上Java线程调度的不一致性。

本书的组织结构

下面是本书的大纲，通过它可以了解本书的素材组织。在附录中讨论的问题要么是还不够成熟，要么是主要具有学术意义，只在极少数情况下才有用。

第一章，线程简介

本章介绍了线程的概念以及本书要使用的术语。

第二章，Java 线程 API

本章介绍了用来创建线程的 Java API。

第三章，同步技术

本章介绍了一种简单的锁定机制，Java程序员可以用它来同步对数据和代码的访问。

第四章，等待和通知

本章介绍另外一种机制，Java 程序员可以用它来同步对数据和代码的访问。

第五章，Java 线程编程的例子

本章总结了前面几章的技术。不同于上面几章的是，本章是面向解决方案的：提供的例子教你如何将已经学到的基本线程技术综合起来使用，并且告诉你如何使用线程来有效率地进行设计。

第六章，Java 线程调度

本章介绍了如何使用 Java API 来控制虚拟机对线程的调度，其中包括在虚拟机的不同实现中不同调度之间的区别。

第七章，Java 线程调度例子

本章提供了扩展Java调度模型的例子，其中包括提供循环调度和线程池的技术。

第八章，和同步相关的高级主题

本章讨论了一些和数据同步相关的高级主题，其中包括和死锁相关的设计以及其他一些同步类（包括 Java 不直接提供但是其他平台具有的同步方法）。

第九章，多处理器机器上的并行化

本章讨论了如何在具有多处理器的机器上设计程序，使得其可以充分利用该机器的能力。

第十章，线程组

本章讨论了 Java 的 ThreadGroup 类。该类允许程序员控制和操作一组线程。Java中和线程相关的安全机制也是基于这个类的，因此也在本章进行了讨论。

附录一，其他主题

本附录包括了Java API中很少有人感兴趣的一些方法：处理线程堆栈的方法以及ThreadDeath类。

附录二，异常和错误

本附录说明了由线程系统使用的异常和错误的细节信息。

排版约定

等宽字体（constant width）表示：

- 代码例子：

```
public void main(String args[]) {
    System.out.println("Hello, world");
}
```

- 正文中的方法、变量、参数名以及关键字。

等宽黑体（**bold constand width**）表示：

- 为解决问题而对程序做出的修改：

```
public void main(String args[]) {
    System.out.println("Hello, world");
}
```

- 在一大段代码中重点表示的部分。

斜体（*italic*）表示URL或者文件名，也表示新引入的术语。

例如，本书的例子可以从如下地址下载：

http://www.oreilly.com/catalog/jthreads2

作者联系方法

我们尽量保证了本书的完整性和精确性。但是Java规范的改变、在不同平台上厂商实现的不同以及底层操作系统的不同使得我们不可能保证所有的例子都是完全正确的（这并不包括可能的文字错误）。因为Java也是在不停变化的，所以本书也是在不停前进的。

作者欢迎你对此书提出反馈意见，特别是指出错误和遗漏之处。你可以通过以下的email地址与我们联系：

scott.oaks@sun.com

henry.wong@sun.com

建议与评论

本书的内容都经过测试，尽管我们做了最大的努力，但错误和疏忽仍然是在所难免的。如果你发现有什么错误，或者是对将来的版本有什么建议，请通过下面的地址告诉我们：

美国：

O'Reilly & Associates,Inc.
101 Morris Street
Sebastopol, CA 95472

中国：

100080 北京市海淀区知春路 49 号希格玛公寓 B 座 809 室
奥莱理软件（北京）有限公司

询问技术问题或对本书的评论，请发电子邮件到：

info@mail.oreilly.com.cn

最后，您可以在 WWW 上找到我们:

http://www.oreilly.com
http://www.oreilly.com.cn

致谢

正如读者所知，仅靠作者是不可能完成一本书的。在此我们对如下人员所提供的帮助和鼓励表示深深的感谢。

Michael Loukides，你肯定了我们的想法，并且带领我们走过构思的阶段。David Flanagan，感谢你对我们草稿的有价值的反馈。Hong Zhang，感谢你在 Windows 线程方面的帮助。Reynold Jabbour 和 Wendy Talmont，感谢你们的支持。

另外，还要感谢我们的家人。James，你在 Scott 完成本书的过程中给予了支持和鼓励；Nini，感谢你在 Henry 编写本书时给他 10% 的时间独处，并且在其他的时间里给予他鼓励：感谢你所做的一切。

最后，我们还要感谢那些对第一版提出建议的读者。我们已经尽力回答你们提出的每一个问题。

本章内容：

- Java 术语
- 线程概述
- 为什么要使用线程
- 总结

第一章

线程简介

本书介绍的是如何在 Java 编程语言和 Java 虚拟机中使用线程。在 Java 中，线程这一主题是很重要的。这种重要性从Java语言中内置了大量与线程相关的特性就可以看出，并且线程系统的另外一些特性也是Java虚拟机所必需的。线程是使用Java时一个不可分割的部分。

线程不是什么新概念：在一段时间里，许多操作系统都通过提供的库来为C语言程序员提供创建线程的机制。而其他一些语言（如 Ada）与 Java 一样，将对线程的支持内置到语言内部。但是，作为在特定的编程情况下才被用到的技术，线程一直被看成是一个非主流的编程主题。

在 Java 中，情况是完全不一样的。如果不引入有关线程的技术，则只能编写最简单的 Java 程序。许多原来用 C 和 C++ 编程的程序员可能从来没有考虑过学习线程，但随着 Java 的流行，他们也需要熟练地掌握线程编程了。

Java 术语

首先，我们要定义一些本书使用的术语。在不同的资料中，许多和Java相关的术语是不一致的；在本书中，我们将尽量一致地使用这些术语。

Java

第一个术语是*Java*本身。正如我们所知，Java最初是作为一种编程语言出现的，因此许多人现在还认为Java仅仅是一种编程语言。但是Java不仅仅是一种编程语言：它还是API（Application Programming Interface，应用编程接口）规范和虚拟机规范。所以当我们提到Java时，我们所指的是整个Java平台。它是由Java编程语言、Java API和Java虚拟机规范组成的一个完整的编程和运行时环境。在很多情况下，当提到Java时，我们可以通过上下文清楚地分辨出所指的是Java编程语言、Java API还是Java虚拟机。要注意的是，本书所讨论的线程特性直接依赖于完整Java平台的所有组成部分。尽管现在已经可能将Java程序直接编译成汇编语言并且脱离Java虚拟机运行，但本书中要讨论的程序与这种可以直接运行的程序不同。

虚拟机、解释程序和浏览器

Java虚拟机（virtual machine）是Java解释程序（interpreter）的另一种说法。Java解释程序是通过解释Java编程语言定义的中间字节码来运行Java程序的。Java解释程序的三种最常见的形式是：程序员使用的解释程序（*java*），它通过命令行或者文件管理器来运行程序；最终用户使用的解释程序（*jre*），它是编程环境的子集，同时也是构成Java插件的基础。内置到许多流行的浏览器［例如Netscape Navigator、Internet Explorer、HotJava™以及JDK(Java Developer's Kit,Java开发者工具包)中包含的appletviewer］中的解释程序。所有这些形式的Java解释程序都仅仅是Java虚拟机的实现，因此我们用Java虚拟机来任指它们中的一个。术语“Java解释程序”特指那些通过命令行独立运行的Java虚拟机(其中也包括那些实行即时编译的虚拟机)；而术语“支持Java的浏览器”（或者更简单地说是“浏览器”）专用于对那些内置于Web浏览器里的虚拟机的讨论中。

在大部分情况下，至少从理论上来说，这些虚拟机是一致的。但是实际上，在这些虚拟机的不同实现之间，还是存在着一些很重要的差别，其中一个差别就来自于线程。在某些情况下，这些差别是很重要的。我们将在第六章中讨论这些差别。

程序、应用程序和applet

这些术语都是用来定义用Java编写的程序的。一般而言，它们被统称为程序。但是Java程序可以分成两种类型：一种是直接在Java解释器中运行的，另一种是在支持Java的浏览器中运行的（注1）。在绝大部分时间里，这两种类型的程序的差异是无关紧要的。此时，我们就用程序来称呼它们。但是当它们之间的区别很重要的时候，我们用术语“应用程序”来称呼那些独立运行的Java程序，而将那些在支持Java的浏览器中运行的Java程序称为applet。对于线程而言，应用程序和applet的不同仅仅体现在它们的Java安全模型方面。我们将在第十章讨论Java安全模型和Java线程之间的交互。

线程概述

现在就剩下一个术语没有定义了：线程到底是什么？线程（thread）其实是控制线程（thread of control）的简写。而控制线程，简单地说，就是在一个程序中与其他控制线程无关的能够独立运行的代码片段。

控制线程

控制线程听起来是一个复杂的技术术语，但其实是一个简单的概念。它是程序运行时的路径。它决定将要执行什么代码：是if块还是else块？while循环到底要运行多少次？如果我们从一个“to do”列表中取出任务来执行，就像计算机运行一个应用程序那样，则我们的执行步骤以及执行的顺序就是执行路径。而这个执行路径就是控制线程运行的结果。

有多个控制线程就如同从两个“to do”列表中执行任务一样。我们仍然以正确的顺序运行每一个“to do”列表中的任务，但是当我们对其中一个列表中的任务感到厌烦时，可以切换到另外一个列表中去运行，并且当我们在以后返回到第一个列表中时，可以回到刚才离开的地方继续执行。

注1：尽管可以写一个可以同时在解释器和浏览器中运行的Java程序，但是当它们实际运行时，这些区别还是适用的。

多任务概述

我们都熟悉使用多任务操作系统同时运行多个程序。每一个这样的程序中都有至少一个线程，所以在某种程度上说，我们已经熟悉单个进程里的单线程的概念。单线程进程具有以下特性，这些特性同时也是多线程程序里的线程所具有的。

- 进程从一个众所周知的入口点开始执行。在C和C++这样的编程语言（Java就更不用说了）中，线程是从调用main()函数或方法的第一条语句开始执行的。
- 对于给定的输入，进程按照一个顺序的、预先定义好的次序来执行语句。对于一个单独的进程来说，任务很简单：执行程序的下一条语句。
- 在执行过程中，进程要访问某些数据。在Java中有三种类型的数据：局部（local）变量是从线程的堆栈中访问的，实例（instance）变量是通过对象引用来访问的，而静态（static）变量是通过类或者对象引用来访问的。

现在考虑一下当你坐在你的计算机旁，同时运行两个单线程程序（文本编辑器和文件管理器）时会发生什么。在你的计算机中有两个进程在运行，每一个进程都有一个线程，而这些线程都有我们上面所说的特性。每一个进程都不需要知道另一个进程的运行情况。根据你的计算机中运行的操作系统的不同，进程之间可以用不同的方法发送各种消息。例如，你会经常从文件管理器中拖动一个文件的图标到文本编辑器中去编辑它。尽管线程可以选择合作运行，但每一个进程都是独立运行的。图1-1显示了一个典型的多任务环境。

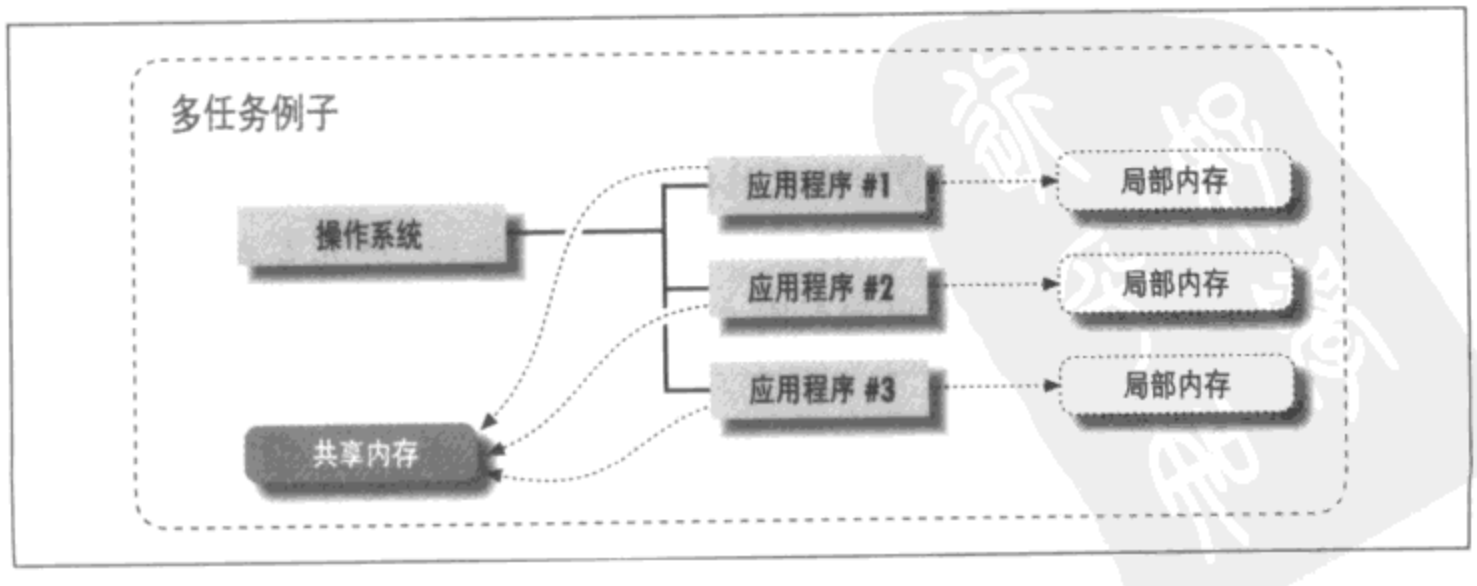

图1-1：多任务环境中的进程

从计算机使用者的角度来看，尽管会被很多变量影响，但这些进程看上去是在同时运行的。这些变量依赖于操作系统：例如，如果一个操作系统不支持多任务，那么就不可能有两个程序看上去是在同时运行的。或者用户认为某一个特定的进程比其他的更重要因此应该一直运行，那么通过关掉其他的运行进程也能影响这种同时性。

最后，这两个进程的数据在默认情况下是隔离的。每一个进程都有自己的堆栈来存储局部变量，也都有自己的数据空间来存放对象和其他数据元素。在许多操作系统中，程序员可以通过编程将数据对象存放在内存中并且允许其他进程访问它们，从而使得多个进程可以共享它们。

多线程概述

以上这些可以使我们做出如下的推论：可以将线程看成是进程，也可以将在一个Java虚拟机中运行的多线程程序看成是在一个操作系统中运行的多个进程，如图1-2所示：

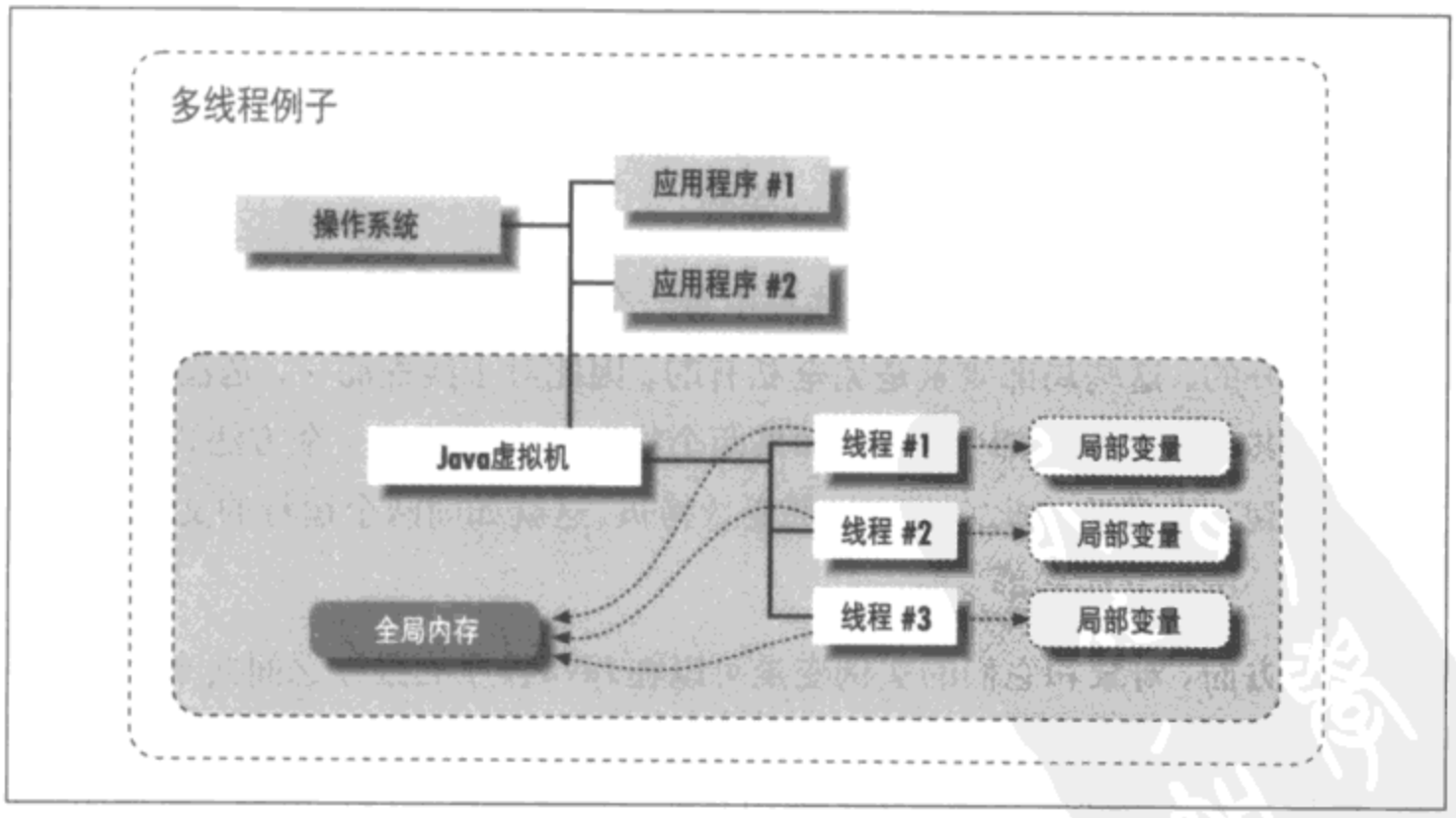

图 1-2：多任务和线程化

所以在Java程序中，多线程有如下的特性：

- 每一个线程都从一个预先定义好的、众所周知的地方开始执行。对于程序中的某个线程，这个地方就是main()方法；而对于程序中的其他线程，这个地方是由程序员在写代码时决定的。对于applet，浏览器也会从main()开始运行。
- 对于一个给定的输入，每一个线程从它的开始点按照一个顺序的、预先定义好的序列运行。线程的任务是简单的，就是执行序列中的下一条语句。
- 每一个线程都是独立于本程序内的其他线程而执行自己的代码的。如果线程希望与其他线程合作来完成某些任务，也有很多实现这种合作的机制。这种线程之间合作工作的技术使得线程编程成为很有用的技术。但是正如用户并不是一定要将文件从文件管理器中拖到编辑器中来编辑一样，线程之间的合作也是可选的。
- 线程看上去是在某种程度上并行运行的。正如第六章中所说的，并行的程度依赖于几个因素——程序中对于各种线程相对重要性的设定和操作系统对于各种特性的支持。所以当你在代码中使用线程时，它们并行运行的可能性是你必须要记住的关键性概念。
- 线程可以访问各种类型的数据。在这一点上，依据Java程序要访问的数据类型而与多进程进行类比是不太合适的。

  每一个线程都是独立的，因此线程中每一个方法的局部变量都是和其他线程隔离开的。这些局部变量是完全私有的，因此对于线程而言，是没有办法去访问其他线程的局部变量的。如果两个线程都去调用同一个方法，则每个线程得到一个单独的此方法的局部变量拷贝。这就如同两个运行的文本编辑器都有一份独立的局部变量一样。

  另一方面，对象和它们的实例变量可以在Java程序的线程之间共享。这种共享比大多数操作系统上进程之间的数据对象的共享要简单得多。实际上，方便地在线程间共享数据对象的能力是使得线程编程十分有用的一个原因。但是Java线程并不能随便访问其他线程的数据对象：它们首先要得到访问这些对象的权限、而且一个线程需要将对象的引用传递给另外一个线程。

> 静态变量是完全不适合这种类比的：在Java程序中，它们自动地在所有线程中共享。

请不要对此感到惊慌：在Java中进行线程编程并不意味着当你要编写一个可以运行多个程序的多任务操作系统时，必须进行系统级的编程。对于通常的任务而言，Java API设计得非常简单并且很少需要特别的技能。

为什么要使用线程？

在Java中，线程的概念是如此根深蒂固，以至于最简单的程序都要创建和使用线程。而且Java API中的许多类都是线程化的，所以有时候你可能都没有意识到自己已经在使用多线程了。

最初，使用线程是为了使得以下类型的程序更容易编写：这些程序可以被分成多个独立的任务，并且对于每一个独立的任务或线程而言，编写算法都是更容易的。这些类型的程序通常是专门的，而且是由多个独立任务组成的。因为这种类型的程序数量不多，所以对于这种类型程序的线程编程就成为了一种专门的技能。通常，这些程序都是作为独立的进程编写的，通过使用与操作系统相关的通信工具（如信号和共享内存空间）来在进程间通信。这种途径增加了系统的复杂性。

随着图形界面成为台式机的标准界面，由于线程系统可以使用户觉得程序有更好的性能，因此线程也慢慢流行起来了。在这些平台上使用线程并没有使程序运行得更快，但是通过一个专门处理用户的输入或者显示输出的线程，用户会觉得系统运行得更快了一些。

最近，线程编程又有了一些新的用法：利用那些越来越多的配置了多个处理器的计算机。那些需要大量计算的程序天生就是这一类的成员。例如，如果一个程序在单处理器机器上要运行一个小时，那么理论上在一个双处理器的机器上就只要半个小时，而在一个四个处理器的机器上只需要十五分钟。要达到这一点就要求使用线程编写程序来进行这些计算。

尽管具备多个处理器的机器已经存在很长的时间了，但是现在它们才足够便宜从而能够被广泛地使用。随着相对便宜的多处理器计算机和那些提供使用这些处理器的线程库的操作系统的出现，开发人员才能够利用这些新机器的每一点优势，这使得线程编程成为了一个热门话题。在Java出现以前，线程编程的兴趣主要集中于通过线程来使用单机上的多个处理器。

但是，Java中的线程通常与多处理器以及它们的能力没有关系。实际上，第一个Java虚拟机就不能利用机器上的多个处理器，而很多虚拟机的实现仍然遵循这个模型。但是，也有一些虚拟机的实现可以利用机器上的多个处理器。使用这些虚拟机，一个恰当编写的程序在一个有双处理器的机器上运行时可能会仅仅使用单处理器机器上的一半时间。如果你指望用Java将你的程序扩展到多个处理器上，使用恰当的虚拟机就可以达到这个目的。但是，即使你的Java程序是设计为在单处理器的机器上运行的，线程仍然是很重要的。

Java中没有异步行为的概念，这正是线程对于Java而言如此重要的主要原因。这就意味着在其他语言中使用的许多处理异步的编程技术都不可能在Java中使用了。实际上，你必须学习线程技术来处理这些异步的行为。

在其他时候，线程也是Java中一种便利的编程技术。特别是对于那些使用的算法本身就具有线程化特性的程序而言，使用Java更是容易。同时，很多Java程序也是由多个独立的行为组成的。在下面，我们列举了一些要求使用Java线程作为程序的一部分的情况。这些情况要么要求异步行为，要么由于使用线程而使程序更加优美。

非阻塞I/O

在Java和其他大多数编程语言中，当你需要从用户得到输入时，可以通过调用read()来指定从某个用户终端（在Java中是System.in）来获取输入。当程序执行到read()方法时，程序等待用户输入至少一个字符，然后才执行下一条语句。这种I/O被称为阻塞（blocking）I/O：在数据到达以满足read()方法前，程序的运行被阻塞住。

但是这种阻塞行为常常不是我们所期望的。当你从一个网络套接字中读取数据时，有时候会读不到数据，因为数据可能在网络传输过程中被延迟了，或者你是从一个周期性发送数据的网络服务器来读取数据的。如果程序阻塞在从网络套接字读取数据上，那么直到数据到达前它就什么都不能做。假如用户界面上有一个按钮，并且用户在程序阻塞在 read()时点击这个按钮，那么因为此时程序不能处理鼠标事件以及执行与按钮事件相关连的处理方法，所以什么都不会发生。这种使用户觉得程序被挂起的情况会让用户感到沮丧。

通常，有三种技术可以处理这种情况：

I/O 多路技术（multiplexing）

开发者经常会打开所有的输入源，并且通过 select()之类的系统调用来使得当某个输入源上有数据到来时通知程序。这使得程序可以像处理用户的鼠标事件一样处理输入［实际上，许多图形工具包都使用这种技术，用户先注册一个回调（callback）函数，当数据从某一个特定的源到来时，这些回调函数将被调用］。

轮询（polling）

轮询允许开发者测试某个源上是否有数据到来。如果有数据，就可以读取并且处理它了。如果没有数据，程序就执行其他的任务。轮询可以通过系统调用 poll()来进行，或者（在某些系统上）当 read()不能立刻读取到数据时，通过返回一个标志来表示。

信号

一个表示输入源的文件描述符可以被设定为当有数据到达数据源时，会有一个异步信号发送给程序。信号会中断当前程序的运行，当处理完数据后再返回到任务被打断的地方继续执行。

在 Java 中，这些技术都不能直接使用。类 FilterInputStream 的 available()方法只提供了对轮询的有限支持，并且没有大多数操作系统中轮询所提供的丰富语义。为了提供那些缺乏的特性，Java 开发者不得不建立一个独立的线程来读取数据。当数据没有到来时，这个线程是阻塞的，同时其他线程就可以处理用户事件以及其他的任务。

尽管阻塞可能发生在任何数据源上，但是最常发生的地方还是网络套接字。如果你过去习惯于套接字编程，你可能会用这些技术从套接字读取数据，但你可能没有向接套字写入过数据。许多程序员习惯于在本地局域网上编程，因此尽管模模糊糊地意识到向套接字写数据可能也会阻塞，但是因为这种阻塞仅仅在特定的情况下才会发生，所以大多数人可能会忽略这一点。网络数据的积压（backlog）就是这样一种情况。在快速的本地网络中，这是很少发生的。但是如果在Internet上进行套接字编程，数据积压发生的机会就会增加，因此试图将数据写入网络时程序被阻塞的机会也会同时增加。所以在Java中，你可能需要用两个线程来处理套接字，一个从套接字读取数据，一个向套接字写入数据。

警告和定时器

传统的操作系统都会提供某种形式的定时器或者警告调用：程序设置定时器然后继续运行。当时间到了时，某种形式的异步信号被发送到程序来通知它定时时间已到。

在Java中，程序员必须建立一个单独的线程来模拟定时器。这个线程睡眠一段指定的时间，然后通知其他线程指定的时间已经到了。

独立的任务

Java程序经常处理一些独立的任务。在最简单的情况下，一个applet可能会在一个页面上显示两个独立的动画；一个更复杂的情况可能是一个代替多个客户同时进行计算的运算服务器。在这些情况下，尽管可以通过一个单线程的程序来处理多个任务，但是用多个线程来分别处理每一个任务会使程序更简单优美。

对于问题“为什么要使用线程”的完整回答在很大程度上依赖于这个原因。作为程序员，我们都习惯于线性的思考，而且不适应于在程序中以并行路径来思考。但是没有理由保留我们所习惯的单线程思考方式：例如当字处理程序中的“保存”按钮被按下时，我们不得不等待一段时间后才能继续工作。更糟的是，字处理程序会周期性地进行自动保存的工作，这总会打断用户的键入以及思考。对于一个线程化的字处理程序，保存操作就可以用一个单独的线程来处理，这样它就不会

打搅用户的工作。如果你习惯于多线程编程，就会发现在很多情况下，增加一个独立的线程会使你的算法更优美，程序更易于使用。

并行算法

随着可以同时使用多个处理器的虚拟机的出现，Java成为了一个有用的开发并行程序的平台。通过将循环中的一个迭代放在一个处理器上而将另外一个迭代放在另外一个处理器上，可以将任何包含循环的程序并行化。但是循环中迭代之间数据的依赖关系可能使得这个循环不可能被并行化，当然还有其他的原因可能使得一个循环不能被并行化。但是对于很多程序来说，当在多处理器的机器上运行时，对那些占用CPU较多的循环进行并行化处理会极大地提高程序的远行速度。

许多语言都有可以自动对循环进行并行化处理的编译器，但是到目前为止，Java还没有这样的编译器。但是正如你会在第九章所看到的，并行化一个循环并不是一个困难的任务。

总结

在一个程序中使用多控制线程的想法看起来很新而且困难。但实际上并不是这样。每一个程序都有至少一个线程，而一个程序中有多个线程和在操作系统中有多个程序并没有什么不同。

一个Java程序可以包含多个线程，对于程序员而言，创建它们并不需要什么特别的知识。现在，你所需要知道的是当你写一个Java程序时，就已经有一个初始的线程从`main()`方法处开始执行了。当你写一个Java applet时，就有一个线程来调用`init()`、`actionPerformed()`等回调函数，这个线程被称为applet的主线程。在每一种情况下，你都可以认为你的程序是从一个单独的线程开始的。如果你想进行I/O处理（特别是如果I/O会被阻塞）、使用定时器或者在初始线程中进行并行处理，你一定要考虑使用一个新的线程来完成这些任务。我们将在下面的章节中介绍如何做到这些。

第二章

Java线程 API

本章内容:

- 通过Thread类创建线程
- 使用 Runable 接口的线程
- 线程的生命周期
- 线程命名
- 访问线程
- 线程的启动、停止和连接
- 总结

在本章中，我们将创建自己的线程。我们会逐步看到，Java线程不仅很容易使用，而且可以很好地和 Java 环境进行集成。

通过 Thread 类创建线程

在上一章中，我们将线程看成是以并行方式运行的单独任务。这些任务仅仅是通过线程执行的代码，而且这些代码实际上也是程序的一部分。这些代码可能用来从服务器上下载图像、通过扬声器来播放声音文件或者完成其他功能。作为程序的一部分，这些代码当然也可以在原始线程中运行。但是为了达到我们期望的并行性，必须创建和管理新的线程来执行恰当的代码。

我们先来看看在下面的例子中仅使用一个线程会有什么结果:

```
public class OurClass {
    public void run() {
        for (int I = 0; I < 100; I++) {
            System.out.println("Hello");
        }
    }
}
```

在这个例子中，有个名为 OurClass 的类。这个类仅有一个名为 run() 的公共方法，这个方法会在 Java 控制台或者标准输出上输出 100 次“Hello”。如果我们在一个 applet（小应用程序）中调用这个方法，它就会在该 applet 的线程中运行：

```
import java.applet.Applet;

public class OurApplet extends Applet {
    public void init() {
        OurClass oc = new OurClass();
        oc.run();
    }
}
```

如果我们实例化一个 OurClass 对象并且调用它的 run() 方法，不会出现什么特别的情况：一个 OurClass 对象会被创建，它的 run() 方法会被调用，同时“Hello”会被输出 100 次。像其他的方法调用一样，在 run() 运行结束以前，run() 方法的调用者会等待 run() 方法结束。我们用图 2-1 来表示代码的运行情况。

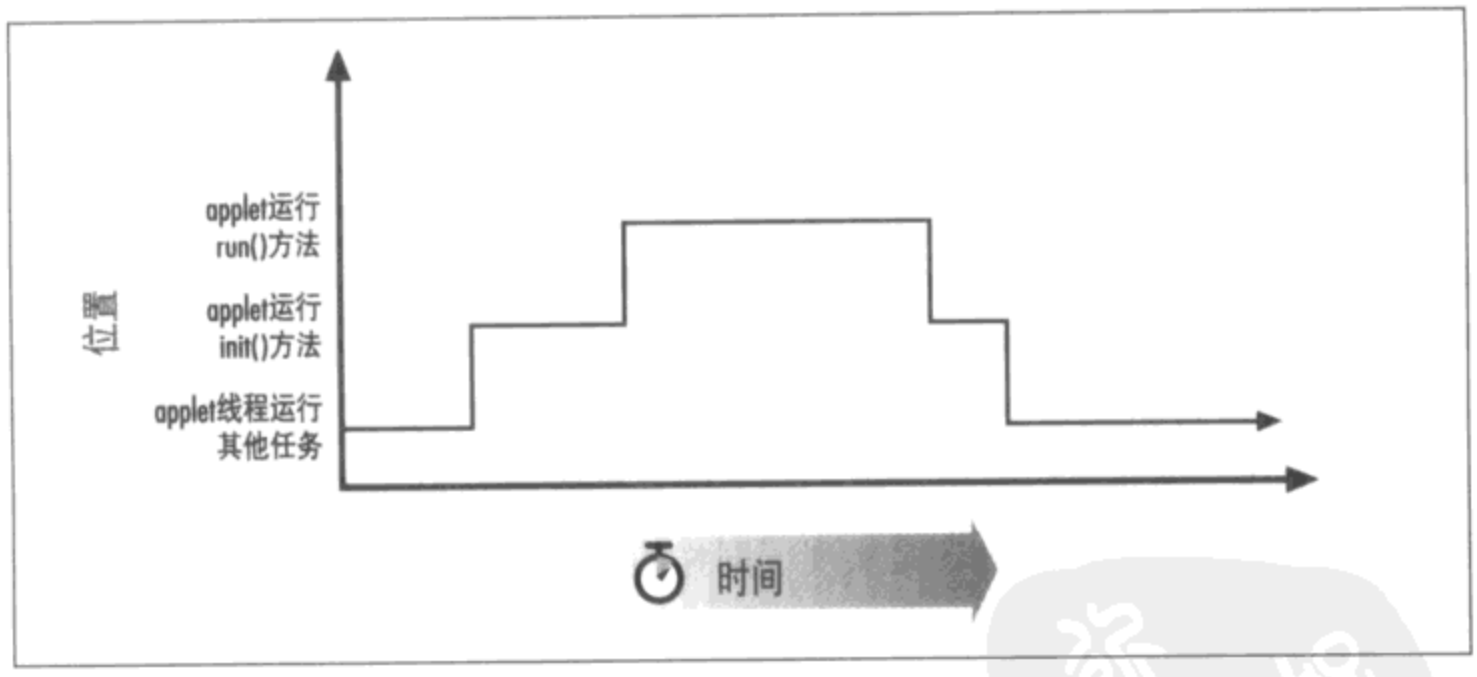

图 2-1：不使用线程时的方法调用

如果我们想让 OurClass 的 run()方法和 applet 的 init()方法以及其他方法一起并行运行，该怎么做呢？为了做到这一点，我们必须修改 OurClass，使它可以在一个新的线程中运行。因此我们要做的第一件事就是使得 OurClass 作为 Thread 类（java.lang.Thread）的子类：

```
public class OurClass extends Thread {
    public void run() {
        for (int I = 0; I < 100; I++) {
            System.out.println("Hello");
        }
    }
}
```

如果我们编译这段代码并且在我们的applet中运行，情况和原来一样：applet的init()方法调用OurClass对象的run()方法，并且会等待这个方法结束后才会继续运行。通过对该例的编译和运行，我们证实了Thread类的存在。该类是我们碰到的第一个与Java线程有关的API，也是我们运行/停止自己线程的编程接口。但是现在我们还没有创建一个新的线程；我们仅仅是创建了一个有run()方法的类而已。接下来，我们要修改applet:

```
import java.applet.Applet;

public class OurApplet extends Applet {
    public void init() {
        OurClass oc = new OurClass();
        oc.start();
    }
}
```

在applet的第二个版本中，我们改变了一行代码：对于run()方法的调用变成了对start()方法的调用。编译和执行这段代码证实了它仍然可以工作，同时对于用户而言，运行结果看上去和原来的版本一样。因为OurClass类没有包含start()方法，因此我们不能判断start()方法是Thread类的还是Thread的父类的。此外，因为该applet还是实现了同样的功能，我们可以得出这样的结论：无论是直接还是间接的，start()方法都调用了run()方法。

如果更仔细地研究，就会发现其实新版本的applet的行为和原来的是不一样的。尽管start()方法最终调用了run()方法，但是是在另外一个线程中做到这一点的。start()方法实际上创建了一个新的线程；这个新的线程，在进行了一些初始化操作后调用run()方法。在run()方法运行结束后，该线程还做了一些与线程终止相关的工作。因而，在新线程中运行的run()方法大概是在start()方法返回到第一个线程的同时开始运行的。如图2-2所示：

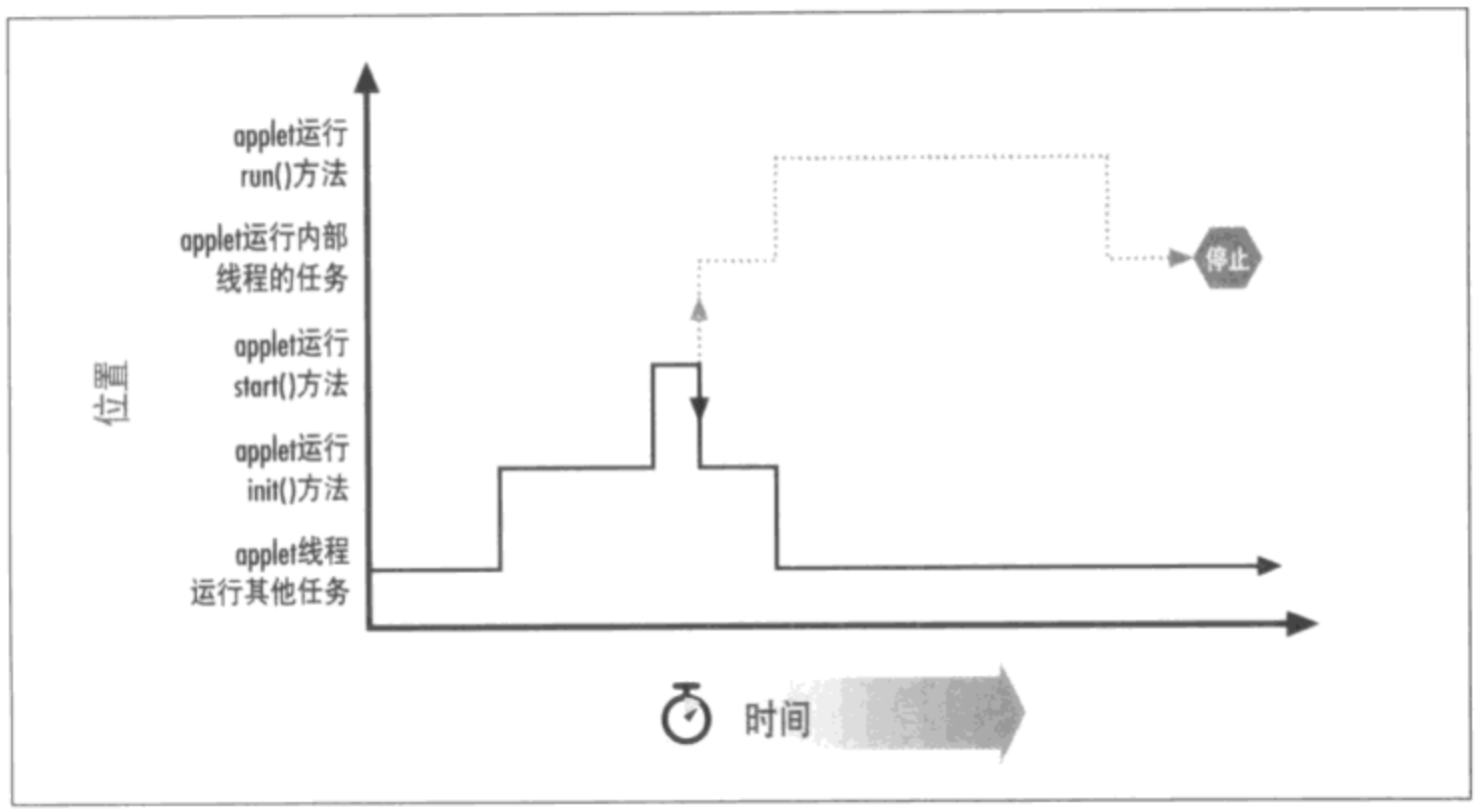

图 2-2: 线程化方法调用

下面是我们到目前为止讨论过的 Thread 类的方法:

Thread()

使用默认值来构造一个线程化的对象。

void run()

新创建的线程会执行这个方法。程序员应该用期望在新线程中执行的代码来覆盖这个方法。以后我们会介绍 run() 方法的默认实现，但是它在本质上是一个空方法。

void start()

创建一个新的线程并运行这个线程类中定义的 run() 方法。

创建一个线程有两个步骤: 首先，在我们的子类中使用自己希望在新的线程中运行的代码来覆盖run()方法;然后通过类的构造函数来创建我们子类的一个对象(在上面的例子中，是调用 Thread 类的默认构造函数)，并且通过调用子类的 start() 方法来执行 run() 方法。

run()和main()的比较

从本质上来说，对于一个新创建的线程，run()方法可以被看成是main()方法：如同程序是从main()方法开始运行一样，新线程就是从其run()方法开始运行的。

但是main()方法从argv来获取它的参数（argv一般是通过命令行来设定的），而新创建的线程是用编程的方式从原始线程获取它的参数的。因此，线程可以通过构造函数、静态实例变量或者程序员设计的其他技术来获得这些参数。

一个显示动画的applet

下面来看一个要创建线程的更具体一些的例子。当想在页面上显示动画时，一般是通过以一定时间间隔显示一系列的图像（帧）来做到这一点。定时器的这种用法在Java中要求使用一个单独线程的情况下很常见：Java中没有异步信号机制，因此必须创建一个单独的线程，使得其睡眠一段时间，然后再唤醒它来通知applet显示下一帧图片。

定时器的实现如下：

```
import java.awt.*;

public class TimerThread extends Thread {
    Component comp;                 // 需要重画的组件
    int timediff;                   // 组件重画的间隔时间
    volatile boolean shouldRun;     // 设为false以停止线程

    public TimerThread(Component comp, int timediff) {
        this.comp = comp;
        this.timediff = timediff;
        shouldRun = true;
    }

    public void run() {
        while (shouldRun) {
            try {
                comp.repaint();
                sleep(timediff);
```

```
                } catch (Exception e) {}
            }
        }
    }
```

这个例子中的TimerThread类和OurClass类一样，是从Thread类继承而来的，并且覆盖了run()方法。它的构造函数保存了要在其上调用repaint()方法的组件和两次调用repaint()方法之间的时间间隔。

到现在为止，我们还没有讨论过sleep()方法。

static void sleep (long milliseconds)

使得当前的执行线程睡眠指定的毫秒。这个方法是静态的，可以通过Thread类名来访问。

static void sleep (long milliseconds, int nanoseconds)

使得当前的执行线程睡眠指定的毫秒和指定的纳秒。这个方法是静态的，可以通过 Thread 类名来访问。

sleep()方法是 Thread 类的一部分，它使得当前线程（也就是调用 sleep()方法的线程）暂停运行指定毫秒长的时间。因为调用sleep()方法时会抛出某些异常，所以上面例子中的try语句是必需的。在附录二中我们会讨论这些异常。但是现在，我们仅仅是丢弃这些异常。

对于sleep()方法的最简单描述是:调用者会睡眠一段指定的时间。正是因为sleep()方法完成任务的方式使得它成为了Thread类的一部分：与那些等待I/O的线程的状态一样，当前线程（即调用sleep()方法的线程）在一段指定的时间内是处于阻塞状态的。有关volatile关键字的详细讨论参见附录一。

下面回到我们两步操作的第二步：看看使用了TimerThread类的applet Animate。

```
import java.applet.*;
import java.awt.*;

public class Animate extends Applet {
    int count, lastcount;
    Image pictures[];
    TimerThread timer;
```

sleep(long)和 sleep(long, int)

sleep()方法的这个版本允许程序员以纳秒级的精度来指定线程睡眠的时间。但不幸的是，大多数实现了Java虚拟机的操作系统都不支持纳秒级这么小的精度。在这些平台上，Java虚拟机会将指定的纳秒四舍五入为最接近的毫秒并以该值调用支持毫秒级的sleep()方法。实际上，大多数操作系统也并不支持一毫秒这样的精度，因此sleep()方法指定的毫秒数也被转换为平台支持的最小分辨率。

对于程序员而言，要牢记的是：并不是Java虚拟机的所有版本都提供纳秒级的支持。一个原则是，要使程序正确工作，就不能依赖于系统对纳秒级精度（甚至是精确的毫秒）的支持。

```
    public void init() {
        lastcount = 10; count = 0;
        pictures = new Image[10];
        MediaTracker tracker = new MediaTracker(this);
        for (int a = 0; a < lastcount; a++) {
            pictures[a] = getImage (
                getCodeBase(), new Integer(a).toString()+".jpeg");
            tracker.addImage(pictures[a], 0);
        }
        tracker.checkAll(true);
    }

    public void start() {
        timer = new TimerThread(this, 1000);
        timer.start();
    }
    public void stop() {
        timer.shouldRun = false;
        timer = null;
    }

    public void paint(Graphics g) {
        g.drawImage(pictures[count++], 0, 0, null);

        if (count == lastcount)
            count = 0;
    }
}
```

我们在applet的start()方法中创建并启动了一个新的线程。这个线程的功能就是通知applet什么时候开始显示下一帧图案，当applet的paint()方法被调用时，仍旧是applet的主线程完成实际的重画工作。在本例子中，init()方法仅仅是从服务器上下载图像。

停止线程

当applet的stop()方法被调用时，我们需要停止定时器线程，因为当applet不再运行的时候，就没有必要再产生repaint()请求。为了做到这一点，我们通过设置TimerThread类中的shouldRun变量来通知该类应该从run()方法中返回。当线程从其run()方法返回时，它已经结束了其运行。在这种情况下，我们也要将timer实例变量设为null，使得垃圾收集程序可以回收该线程对象。

线程从自己的run()方法中返回来结束运行是终止线程运行的最好方法。程序员可以决定线程如何知道自己何时从run()方法中返回；上例中设定了一个标志，这是一种最简单的实现方法。

设置标志就意味着线程要周期性地检查这个标志。有没有更简洁的方法呢？是否可以立即停止一个线程的运行而不是等待它去检查某些标志呢？答案是即肯定又否定的。Thread类包含的stop()方法可以立即停止线程的运行而不管线程正在做什么。在Java 2中，这个方法已经不被推荐使用了。但是导致不赞成使用stop()方法的原因不仅仅在于Java 2虚拟机，所有的Java平台上都不应该继续使用这个方法了。在第六章中，当我们对线程编程的细节有了更多了解后，会讨论不赞成使用stop()的具体原因。现在，我们只需记住：使用stop()方法是危险的。另外，在第十章也会提到使用这个方法导致的安全异常。所以我们不能总是依赖这个方法。

Applet类的start()和stop()方法

不幸的是，Applet类和Thread类都有start()和stop()方法，并且具有同样的签名（signature）。这可能是实现或调试多线程applet时使人困惑的原因之一。

其实这两个方法具有不同的功能，而且两者之间没有什么直接的联系。

下面是 stop() 方法的定义:

void stop()(Java 2 中已经不推荐使用了)

终止一个已经运行的线程。

当 run()方法结束后（或者是调用 stop()方法后），什么会返回给主线程呢？ 正如我们上面提到的，当 run() 方法结束时，线程会自动进行清理并处理一些与终止线程相关的事情。stop() 仅仅提供一个终止 run() 方法运行的途径。线程还是会自动进行清理并处理其他与终止线程相关的事情。stop() 方法的工作细节将在附录一中给出。

使用 Runable 接口的线程

到目前为止我们概要介绍的创建线程的技术，尽管很简单，但是存在一个问题：在Java中，类仅仅能够从另外一个类中继承它们的行为。这就意味着继承是很稀有的一种资源，并且对于程序员而言，继承是很“昂贵的”。

在上面的例子中，线程中仅仅包含一个简单的循环，因此没有什么要担心的。但是，对于一个有着详细的继承树关系的完整类结构而言，如果想让它在自己的线程中运行，就不能如我们上面那样使得它从Thread类继承而来。一个可行的方法就是创建一个新的类，该类从Thread继承而来，并且包含指向我们要运行的类的实例的引用。不言而喻，这样的解决方法是很麻烦的。

Java语言使用一种称为接口（interface）（注 1）的机制来解决其不支持多重继承的缺陷。Thread 类支持该机制。简单地说就是我们可以通过实现 Runnable 接口（java.lang.Runnable）来代替从 Thread 类继承。Runnable 接口的定义如下:

```
public interface Runnable {
    public abstract void run();
}
```

注 1: 有反对者认为接口不能完成多重继承的全部功能。但这种争论超出了本书的讨论范围。

Runable接口仅仅包含一个方法: run()方法。线程化的类实际上实现了Runnable接口。因此，当一个类继承自 Thread 类时，其子类也同时实现了 Runnable 接口。但是，此时我们并不想通过继承 Thread 类来实现 Runnable 接口。要达到这个目的，只需要将“extends Thread”替换为“implements Runnable”即可，步骤一中的其他部分都不需要改变。

```
public class OurClass implements Runnable {
    public void run() {
        for (int I = 0; I < 100; I++) {
            System.out.println("Hello, from another thread");
        }
    }
}
```

在创建线程的第二步中还需要做一些改变。因为OurClass类的实例不再是Thread对象，因此就不能作为Thread对象对待了。但是为了创建一个单独的线程，还是需要一个 Thread 类的实例。不过此时要通过 OurClass 对象的引用来创建新的线程对象。换句话说，就是现在的过程要稍微复杂一点。

```
import java.applet.Applet;

public class OurApplet extends Applet {
    public void init() {
        Runnable ot = new OurClass();
        Thread th = new Thread(ot);
        th.start();
    }
}
```

和前面一样，必须创建 OurClass 类的一个实例。但是，在这个新的版本中，还需要创建一个真正的线程对象。我们通过将可运行的OurClass对象的引用作为参数传递给 Thread 类的构造函数来创建一个 Thread 对象。

Thread(Runnable target)

构造与给出的 Runnable 对象相关联的新线程对象。

通过调用新的 Thread 对象的 start()方法来执行一个新的线程。

因为我们期望线程通过某种方法来运行我们的run()方法，所以需要将一个可运行对象传递给线程对象的构造函数。同时因为不再覆盖Thread类的run()方法，所以Thread类的默认run()方法将会运行；而默认的run()方法的代码如下：

```
public void run() {
    if (target != null) {
        target.run();
    }
}
```

在上面的代码中，target是传递给线程的构造函数的可运行对象。因此，线程通过执行Thread类的run()方法开始执行，该方法会立刻调用可运行对象的run()方法。

更有趣的是，因为现在使用的是Runnable接口而不是继承自Thread类的技术，所以OurClass类的代码可以和applet融和在一起了。这种技术经常用来在applet中创建单独的线程。因为applet本身现在也是可运行的，因此applet线程的实例变量和这个新线程的run()方法就是和原来相同的了。

```
import java.applet.Applet;

public class OurApplet extends Applet implements Runnable {
    public void init() {
        Thread th = new Thread(this);
        th.start();
    }

    public void run() {
        for (int I = 0; I < 100; I++) {
            System.out.println("Hello, from another thread");
        }
    }
}
```

这种方法同样也可以用来改写上面的Animate类：

```
import java.applet.*;
import java.awt.*;

public class Animate extends Applet implements Runnable {
    int count, lastcount;
```

```
Image pictures[];
Thread timer;

public void init() {
    lastcount = 10; count = 0;
    pictures = new Image[10];
    MediaTracker tracker = new MediaTracker(this);
    for (int a = 0; a < lastcount; a++) {
        pictures[a] = getImage (
            getCodeBase(), new Integer(a).toString()+".jpeg");
        tracker.addImage(pictures[a], 0);
    }
    tracker.checkAll(true);
}

public void start() {
    if (timer == null) {
        timer = new Thread(this);
        timer.start();
    }
}

public void paint(Graphics g) {
    g.drawImage(pictures[count++], 0, 0, null);
    if (count == lastcount) count = 0;
}

public void run() {
    while (isActive()) {
        try {
            repaint();
            Thread.sleep(1000);
        } catch (Exception e) {}
    }
    timer = null;
}
}
```

在将类代码合并后，我们拥有了对于applet的直接引用，因此可以直接调用repaint()方法。同时，因为Animate类不再是Thread类的子类了，所以run()方法不能够直接调用sleep()方法了。幸运的是，sleep()方法是一个静态方法，因此我们可以通过Thread类说明符来直接使用这个方法。

通过上面的例子，我们可以了解到，对于有固定继承结构的类而言，可以通过线

程化接口模型在不创建新类的情况下实现线程化。然而，什么时候应该使用Runnable()接口，而什么时候需要创建一个Thread类的子类呢？

isActive()方法

在上一个例子中，我们使用了isActive()方法，而不是显式地停止一个线程。这是另外一种停止线程的技术；使用isActive()方法的好处是run()方法可以正常地关闭一个线程，而不是通过使用stop()方法来硬性终止。这就使得run()方法在停止运行前可以进行一些清理工作。

isActive()方法是Applet类的一部分，它用来确定一个applet是否处于活动状态。从定义上说，一个applet在它的start()和stop()方法之间都是处于活动状态的。这个方法和Thread类的isAlive()方法是不同的，我们会在后面讨论isAlive()方法。请不要混淆这两个方法。

是不是通过Runnable接口的方式实现线程化就能够解决继承方法不能解决的问题呢？反过来是否也是这样呢？现在，这两种技术似乎没有任何重要的区别。对于某些任务使用一种技术是很容易的，而在其他任务中使用另外一种技术更容易。例如，在上面的Animate类中，通过在Applet的类中实现Runnable接口，节省了额外定义新类的工作。在更早的例子中，通过继承Thread类得到的单独的TimerThread类使得程序更容易理解和调试。但是它们之间的区别是非常小的，这使得所有的任务都可以用任何一种技术来解决。

现在，只要根据自己的喜好以及解决方案的简单程度来选择其中一种就可以了，而不需要担心它们之间的区别。我们期望通过本书的例子，使读者能够逐步掌握它们的用法。

要写一个简单的线程化Java程序是很容易的：编写一个类，其中定义的一个方法能够在某个独立线程中运行，该线程通过start()方法来启动，在执行完run()方法后结束运行。但是从上一章我们已经了解到，使得线程化系统具有强大能力的原因并不仅仅是创建不同线程的能力，真正的原因是不同的线程可以很容易地通过调用它们共享的对象方法进行通信。

使用继承还是使用接口?

如上所述，要根据我们的爱好和解决方案的简单程度来选择其中一种技术。那些喜欢纯粹面向对象技术的人会争辩说，如果不增强Thread类的话，就不应该使用从 Thread 类继承的方法。

理论学家可能会用整整一章来讨论这个问题。但是我们的兴趣是如何使代码清晰。关于是使用继承还是接口的其他原因都超出了本书的范围，因此不会在这里讨论。

线程的生命周期

到目前为止，我们对于线程的工作原理有了简单的了解：我们知道如何通过 start()方法来启动一个线程，如何通过安排 run()方法来终止线程的运行。下面我们将介绍两个可以提供线程在其生命周期中的信息的方法。

isAlive()方法

在调用 start()方法后，Java 虚拟机要经过一段时间才能真正启动线程。同样，当线程从其run()方法返回后，Java虚拟机也要用一段时间才能完成必需的清理工作。而如果使用 stop()方法的话，Java 虚拟机的清理时间会更长。

因为需要时间来启动和结束一个线程，所以导致了这种延迟；因此，如图 2-3 所示，在线程的运行阶段到其不运行阶段有一段过渡时间。在 run()方法返回后，在线程真正结束前也存在一小段时间。如果我们想知道 start()方法是不是被调用了，或者想知道一个线程是不是结束了(这也许更有用)，就必须使用isAlive()方法。这个方法可以告诉我们一个线程是不是已真正启动并且没有真正结束。

boolean isAlive()

用来判断一个线程是不是活动的。从定义上来说，一个线程从其真正启动前一段时间到其真正停止后一段时间之间都被认为是活动的。

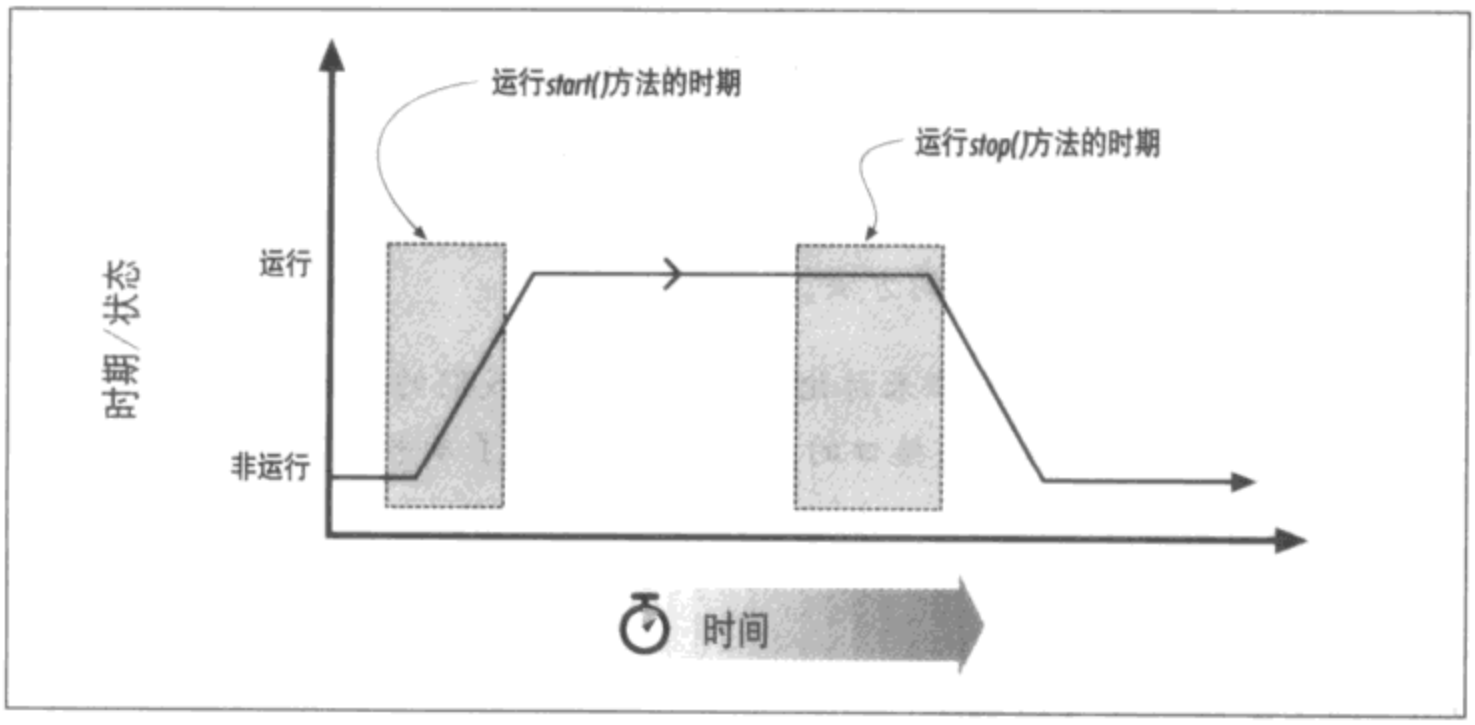

图 2-3: 线程状态的图形表示

下面修改 Animate 类，使其等待定时器线程结束后才结束自己的运行。

```
import java.applet.*;
import java.awt.*;

public class Animate extends Applet {
    int count, lastcount;
    Image pictures[];
    TimerThread timer;

    public void init() {
        lastcount = 10; count = 0;
        pictures = new Image[10];
        MediaTracker tracker = new MediaTracker(this);
        for (int a = 0; a < lastcount; a++) {
            pictures[a] = getImage(
                getCodeBase(), new Integer(a).toString()+".jpeg");
            tracker.addImage(pictures[a], 0);
        }
        tracker.checkAll(true);
    }

    public void start() {
        timer = new TimerThread(this, 1000);
        timer.start();
    }
```

```
        public void stop() {
            timer.shouldRun = false;
            while (timer.isAlive()) {
                try {
                    Thread.sleep(100);
                } catch (InterruptedException e) {}
            }
            timer = null;
        }

        public void paint(Graphics g) {
            g.drawImage(pictures[count++], 0, 0, null);

            if (count == lastcount) count = 0;
        }
    }
```

因为线程可能会阻塞(可能在等待I/O,也可能是处于start()方法的过渡期中),所以线程被启动并不一定意味着该线程真正在运行了或者是可以运行了。正是因为这些原因,isAlive()方法经常被用来检测一个线程是否停止运行了。以上面applet的stop()方法为例,和在Java原来的版本中一样,TimerThread对象会随着applet的启动和停止而启动和停止。但是在新版本中,applet的stop()方法不仅仅会停止TimerThread,它还通过检查来保证该线程是真正停止了。

在本例中,保证定时器线程真正停止并没有带来什么特别的益处。但是假如两个线程因为某种原因要访问相同的数据,并且要保证其中一个线程在另外一个线程停止运行后才能访问该数据,那么我们就可以在运行一个线程之前通过简单的循环和检查来确保另外一个线程已经结束运行。

在另外一种情况下,线程也被认为是不再活动的:在调用stop()方法后的一段时间内,线程被认为是不再活动的。这其实是相同的情况:不管是正常结束的还是由于调用stop()方法而导致的结果,都可以用isAlive()方法来检测run()方法是否真正结束了。

线程连接

isAlive()方法可以被看成是线程间的简单通信方法。我们等待另外一个线程的结束信息。下面看看另外一个例子:当启动好几个线程去进行某种长时间的运算

时，主线程就有时间完成其他的工作。假设主线程结束了其他的工作，需要对其子线程的运算结果进行处理，此时主线程就需要等运算过程结束后才能继续运行。

可以通过刚刚介绍过的在循环中使用isAlive()的技术来实现这一点。但是Java API中的另外一个技术更适合于这种情况。这种等待行为被称为线程连接(thread join)。我们将在创建线程时分裂出去的那些线程连接到一点上。因此，可以这样修改上面的例子:

```
import java.applet.Applet;

public class Animate extends Applet {
    ...
    public void stop() {
        t.shouldRun = false;
        try {
            t.join();
        } catch (InterruptedException e) {}
    }
}
```

Thread 类提供了如下的 join()方法:

void join()

等待指定的线程运行结束。根据定义，当线程不活动时（也包括线程并没有开始运行的情况），join()会立刻返回。

void join(long timeout)

等待指定的线程运行结束，但是不超过指定的超时值（以毫秒为单位）。实际上以毫秒为单位指定的超时值会根据底层平台的能力进行调整。

void join(long timeout, int nanos)

等待指定的线程运行结束，但是不超过指定的超时值（以毫秒和纳秒为单位）。实际上指定的超时值会根据底层平台的能力进行调整。

当调用 join()方法时，当前线程会等待其连接的线程不再处于活动状态。不活动可能是指以下三种情况: 线程根本就没有运行; 线程被其他线程停止了; 线程执行完毕。简单地说，join()方法实现了前面例子中综合使用 sleep()和

isAlive()方法所完成的功能。但是，使用join()只需要一次方法调用。超时控制机制还可以提供更好的控制能力，而且不会像轮询那样浪费CPU周期。

isActive()方法和join()方法的另外一个有趣的地方就是它们并不影响它们所作用的线程。无论join()方法是否作用于某个线程上，它的运行都不受任何影响。实际上，仅仅是调用join()方法的线程会受到影响。isActive()方法仅仅是返回线程的状态，而join()方法仅仅是在线程上等待某种状态。

join()、isAlive()和当前线程

一个线程自己调用isAlive()方法或者join()方法是没有意义的。因为如果一个线程不是处于活动状态，它就什么事情都不能做，所以没有任何理由去检查它是不是活动的。实际上，当用isAlive()检查自己时，仅仅返回true。如果线程在调用isAlive()方法期间被结束了，则isAlive()不可能返回。所以一个线程对自己调用isAlive()只会得到true。

线程自己调用join()也是没有意义的。但是我们可以检查一下这样做的结果。结果证实join()方法使用isAlive()来判断什么时候从join()方法返回。在现在的实现中，join()方法并不检查被连接的线程是不是自己。换句话说，join()方法当且仅当线程不活动时才会返回。这会使对自己调用join()方法的线程进入永久等待状态。

线程命名

下面我们将探讨一些主要用来对线程进行登记（bookkeeping）的方法。首先，可以给将一个字符串名字赋给一个线程对象。

void setName(String name)

分配一个名字给Thread实例。

String getName()

获取Thread实例的名字。

Thread 类提供了为线程分配一个名字以及获取这个名字的方法。尽管线程的 toString()方法的默认实现中会打印出这个名字，但是系统并不真正使用这个名字。那些为线程分配名字的程序员可以根据自己的需要自由地使用这个字符串。例如，我们为 TimerThread 类分配一个名字：

```
import java.awt.*;

public class TimerThread extends Thread {
    Component comp;                    // 需要重画的组件
    int timediff;                      // 重画组件的间隔时间
    volatile boolean shouldRun;        // 设为false以停止线程

    public TimerThread(Component comp, int timediff) {
        this.comp = comp;
        this.timediff = timediff;
        shouldRun = true;
        setName("TimerThread(" + timediff + " milliseconds)");
    }

    public void run() {
        while (shouldRun) {
            try {
                comp.repaint();
                sleep(timediff);
            } catch (Exception e) {}
        }
    }
}
```

在这个版本的 TimerThread 类中，我们为线程分配了一个名字。在此例中，该名字是“TimerThread”后跟定时器线程所使用的毫秒数。如果对该实例调用 getName()方法，就会返回这个字符串值。

Thread 类的构造函数也支持线程命名：

Thread(String name)

构造一个已经分配名字的线程对象。当通过继承方式来使用线程时可以使用这个构造函数。

Thread(Runnable target, String name)

该构造函数通过一个实现了 Runnable 接口的对象来创建一个线程对象，同

时分配一个名字给该线程对象。当通过接口方式来使用线程时可以使用这个构造函数。

线程名字的使用

使用线程名字来存储信息并不是很有用。对Thread类（使用继承方式来实现线程化）或者Runnable类型的类（通过接口来实现线程化）增加一个实例变量，也能够很容易地达到同样的效果。使用名字的最大好处可能就是便于调试。如果被赋予了一个名字，调试器和toString()方法就可以按照逻辑名字而不是数字来显示线程信息。

默认情况下，如果没有被赋予一个名字，Thread类会选择一个惟一的名字。该名字通常是在“Thread-”后面加上一个惟一的数字。

像使用setName()方法一样，通过线程构造函数来设定名字也是很简单的。对于继承和接口方式都提供了一个相应的构造函数。在下面的TimerThread例子中，我们通过构造函数来设定线程的名字，而不需要使用setName()方法。

```
import java.awt.*;

public class TimerThread extends Thread {
    Component comp;                    // 需要重画的组件
    int timediff;                      // 重画组件的间隔时间
    volatile boolean shouldRun;        // 设为false以停止线程

    public TimerThread(Component comp, int timediff) {
        super("TimerThread(" + timediff + " milliseconds)");
        this.comp = comp;
        this.timediff = timediff;
        shouldRun = true;
    }

    public void run() {
        while (shouldRun) {
            try {
                comp.repaint();
                sleep(timediff);
            } catch (Exception e) {}
        }
    }
}
```

访问线程

下面我们将介绍一些可以提供特定线程信息的方法。

当前线程

首先，来看看 currenThread()方法：

static Thread currentThread()

获取表示当前运行线程的 Thread 对象。该方法是静态的，可以通过 Thread 类名字进行调用。

作为 Thread 类的一个静态方法，它仅仅返回表示当前线程的 Thread 对象；而当前线程就是调用 currentThread()方法的线程。

为什么这是一个很有用的方法呢？原因就在于表示当前线程的Thread对象并没有保存在所有地方，即使保存了Thread对象，当前调用的方法也可能不能访问这个对象。例如，设想一个通过套接字进行I/O操作的类，它要将其读到的数据存储到一个内部的缓冲区中。在下一章中，我们会看到这个类的完整实现，但是现在，我们仅仅对其接口感兴趣：

```
public class AsyncReadSocket extends Thread {
    StringBuffer result;

    public AsyncReadSocket(String host, int port) {
    // 打开到给定主机的套接字
    }

    public void run() {
    // 从套接字读取数据，放入结果字符串缓冲区
    }

    // 获取已经从套接字读入的字符串
    // 只允许进行读操作的线程执行这个方法
    public String getResult() {
        String reader = Thread.currentThread().getName();
        if (reader.startsWith("Reader")) {
            String retval = result.toString();
```

```
                result = new StringBuffer();
                return retval;
            } else {
                return "";
            }
        }
    }
```

在该例中，只有通过getResult()方法才能得到读取的数据，但是getResult()的代码只允许进行读操作的线程获取存储的数据。在该例中，我们假设进行读操作的线程的名字是以“Reader”开头的。可以通过调用setName()或者构造函数将这样一个名字赋给读取线程。而通过调用getName()方法，可以很容易地得到线程名字。但是，在getResult()方法中，由于没有当前调用自己的Thread对象的引用，因此必须通过currentThread()方法来获取该引用。在该例中，获取线程对象的引用仅仅是为了获取线程的名字；但是只要有了该引用，其他的操作也是很容易实现的。例如，可以利用线程引用进行优先级控制或者是线程组的操作。在以后的章节中，我们会看到这些操作。

要注意的是这里有一个微妙之处。因为getName()方法是Thread类的方法，所以在我们的代码中可以直接调用它。这会返回AsyncReadSocket线程的名字。但是我们实际想要的是调用getResult()方法的线程的名字，而不一定是AsyncReadSocket线程的名字。通常，我们可以这样使用AsyncReadSocket类：

```
public class TestRead extends Thread {
    AsyncReadSocket asr;
    public static void main(String args[]) {
        AsyncReadSocket asr = new AsyncReadSocket("myhost", 6001);
        asr.start();
        new TestRead(asr).start();
    }

    public TestRead(AsyncReadSocket asr) {
        super("ReaderThread");
        this.asr = asr;
    }

    public void run() {
        // 进行其他处理，允许asr读数据
        System.out.println("Data is " + asr.getResult());
    }
}
```

在该例中有三个线程是我们感兴趣的：由Java虚拟机启动来调用main()方法的线程、asr线程和TestRead线程。因为TestRead线程的名字也是以“Reader”开头的，所以它可以通过执行getResult()方法读取到数据。如果该例中的其他线程调用getResult()方法，则只会得到一个空串。

这是一个经常会引起混淆的原因：任何一个Thread类的子类，它的方法既可以被线程对象自己调用，也可以被其他线程对象调用（如同上面例子中的getResult()方法一样）。因此，不要假设一个线程对象的代码仅仅会被该线程对象所表示的特定线程调用。

枚举虚拟机中的线程

在Thread类中也提供了获取程序中所有线程列表的方法。

static int enumerate(Thread threadArray[])

获取程序中的所有线程对象，并且将它们存储在一个线程数组中。该方法的返回值是存储在该数组中的线程对象的个数。该方法是静态的，可以通过Thread类名来调用。

static int activeCount()

返回程序中的线程个数。该方法是静态的，可以通过Thread类名来调用。

该列表是通过enumerate()方法得到的。程序员只需创建一个线程数组并将它作为参数传递给enumerate()方法。enumerate()方法将线程对象的引用存储在该数组中，并返回所存储的线程对象的个数；该值是当前程序中的线程个数和该数组参数个数中较小的那个值。

为了确定一个合适大小的数组，需要在程序中动态确定线程的个数。可以用activeCount()方法来确定线程个数以及决定相应的线程数组的大小。例如，在我们的Animate applet中增加一个如下的方法，就可以打印出该applet中的所有线程：

```
import java.applet.*;
import java.awt.*;
```

```
public class Animate extends Applet {
// 实例变量和方法从略

    public void printThreads() {
        Thread ta[] = new Thread[Thread.activeCount()];

        int n = Thread.enumerate(ta);
        for (int i = 0; i < n; i++) {
            System.out.println("Thread " + i + " is " +
                    ta[i].getName());
        }
    }
}
```

在本例中，通过使用Thread类的activeCount()方法来确定线程数组的大小，然后实例化这个数组。一旦知道了活动线程的个数，我们就可以调用enumerate()方法来获取applet中所有线程对象的引用。该方法的其余部分仅仅是通过对每一个线程对象的引用调用getName()方法来获取并打印其名字。

注意我们上面用的是“程序中的所有线程”而不是“Java虚拟机中的所有线程”。这是因为在Thread类这一级别上，enumerate()方法仅仅返回程序创建的线程，加上（可能）Java虚拟机为应用程序或applet创建的主线程和GUI线程。它不会显示虚拟机中的其他线程（例如垃圾收集线程）。对于applet而言，该方法也不会显示其他applet的线程。在第十章中，我们会介绍如何与其他线程打交道。

线程的启动、停止和连接

看看Animate的最新版本：

```
import java.applet.Applet;

public class Animate extends Applet {
    TimerThread t;
    public void start() {
        if (t == null)
            t = new TimerThread(this, 500);
        t.start();
    }
```

什么时候线程是活动的？

什么时候线程是活动的？粗看上去，这是个简单的问题。对于一个线程来说，在start()方法和stop()方法之间（准确地说是在stop()方法调用结束后的一小段时间内），isAlive()方法认为其是活动的。我们可以认为一个“活着的”线程就是活动的。

但是，如果对于活动线程的定义是一个线程的引用会被activeCount()方法统计在内的话，我们对活动的定义就不同了。当一个线程对象被构造出来而不是被启动后，该对象的引用就会出现在enumerate()方法返回的线程数组中，并且也会被activeCount()方法统计在内。

当线程被停止或者其run()方法运行结束后才会从线程数组中删除。这就意味着如果一个线程被创建了但是还没有启动，那么即使在程序中没有保存该线程对象的引用，该线程对象还是会出现在enumerate()得到的列表中。

```
    public void stop() {
        t.shouldRun = false;
        try {
            t.join();
        } catch (InterruptedException e) {}
        // t = null;
    }
}
```

在最新的Animate例子中（参见本章前面的“线程的生命周期”一节），applet的start()方法每次都会创建一个TimerThread对象并且启动它。但是如果仅仅创建TimerThread一次呢？在上面的例子中，我们再次在applet的start()方法中创建了一个TimerThread线程。但是，由于我们知道该线程会在stop()方法中被停止，因此我们会在start()方法中试图重新启动它。换句话说，就是仅仅创建TimerThread线程一次，并且使用这个线程对象来控制动画的显示。通过启动/停止一个TimerThread对象，使得applet不需要在每一次启动时都去创建一个新的TimerThread实例，而且垃圾收集器也不需要在applet停止运行后，对于那个失去引用的TimerThread线程进行回收了。

但是这个想法能不能实现呢？可惜的是，答案是否定的。当一个线程被停止后，

线程对象的状态就被设置为不能重新启动。在该例中，如果我们试图通过调用TimerThread类的start()方法来重新启动线程，什么都不会发生。虽然start()方法不会返回异常，但是run()方法是不会被调用的。isAlive()方法也不会返回true。换句话说，不可能重新启动一个线程。一个线程实例能且只能使用一次。

关于重新启动线程的细节

当你试图重新启动一个线程时，到底发生了什么呢？其实结果是依赖于你在什么时候去重新启动该线程的。当一个线程的stop()方法被调用了或者其run()方法结束后，实际上要经过一段时间才能真正停止。因此，当调用start()方法时，到底发生什么依赖于一个竞态条件（第三章会讨论竞态条件）。

如果一个正在停止的线程被调用了start()方法，就会进入错误状态，并会抛出一个异常。对于一个没有被停止的线程调用start()方法也会得到同样的结果。

如果一个线程已经真正停止了，那么调用start()方法是不会有任何结果的：线程对象处于一个不可能重新启动的状态。

能不能再次停止一个已经停止的线程呢？粗看上去，这好像是一个奇怪的问题。但是答案是肯定的，而原因就是为了避免竞态条件的发生。我们知道有两种方式可以停止一个线程，所以我们有可能用stop()方法去停止一个其run()方法已经运行结束的线程。如果一个Thread类不允许对一个已停止的线程调用stop()方法，那就要求我们在停止一个线程前检查它是否是处于运行状态，因此我们必须避免因为run()方法在检查线程是否是运行和实际调用stop()方法之间运行结束而带来的竞态条件。这对Java程序员会是一个大负担。所以，stop()方法被设计为可以对一个已经停止的线程进行操作。

如果对于一个已经停止运行很久的线程调用join()方法，会有什么结果呢？到目前为止，我们假设join()方法要等待一个线程运行结束或者被停止运行。但是这种假设是不必要的，如果该线程是已经停止的，对它调用join()方法会立刻

返回。这看上去是显而易见的，但是同样要注意的是，如果join()方法要求线程是活动的话，同样也是会引起竞态条件的。

线程的停止和垃圾收集程序

线程对象和其他对象一样，当它不再被引用后，就成了垃圾收集程序的目标。作为程序员，我们只需知道垃圾收集程序能够很好地和线程系统合作，而不需要了解其细节。但是，对于那些对细节感兴趣的人，我们也简单解释一下垃圾收集程序是如何作用于线程系统的。

在上面的所有例子中，当线程对象已经运行结束或者被停止运行的时候，垃圾收集程序并不能回收它们。这是因为当我们停止这些线程的运行后，我们还是拥有对于TimerThread对象的引用。因此我们要手工地解除对线程对象的引用。但是，这样做仅仅会释放那些用于存储线程对象的内存。线程系统会在线程执行结束或被停止运行后，自动释放那些和线程相关的资源（包括和操作系统相关的资源）。

对于一个正在运行的线程，解除对线程对象的引用也不会引起什么副作用。为了对Thread类的currentThread()和enumerate()方法提供支持，线程系统会对所有正在系统中运行的线程保持一个引用。因此，当一个线程没有停止运行以前，线程系统是不会对线程对象解除引用的，因此垃圾收集程序也不会对这些线程进行回收。

如何对多个线程进行join()操作呢？先看看下面的代码：

```
import java.applet.Applet;

public class MyJoinApplet extends Applet {
    Thread t[] = new Thread[30];
    public void start() {
        for (int i=0; i<30; i++) {
            t[i] = new CalcThread(i);
            t[i].start();
        }
    }

    public void stop() {
        for (int i=0; i<30; i++) {
```

```
                try {
                    t[i].join();
                } catch (InterruptedException e) {}
            }
        }
    }
```

在该例中，启动了30个CalcThread对象。尽管没有定义CalcThread类，但是我们假设该类是一个大的数学算法的一部分。在applet的stop()方法中，我们使用一个循环来等待所有的线程结束。这是用来等待多个线程停止的最好方法吗？因为对于一个停止运行的线程调用join()方法是没有什么副作用的，所以尽管这些线程停止运行的顺序不同于它们启动的顺序，但将对它们的join()操作放在一个循环中是很好的方法。无论我们在循环中是如何使用join()的，完成全部join()所用的时间都是最后一个线程停止运行的时间。

当然，在某些情况下，使用特定的连接机制会好一些。但是那依赖于具体情况，而不是由线程系统决定的。如果不是按照启动顺序来调用join()，也不会对性能造成影响。

总结

下面是本章介绍过的Thread类的方法：

Thread()

使用默认值来创建一个线程对象。

Thread(Runnable target)

创建一个和给定Runnable对象相关联的线程对象。

Thread(String name)

创建一个线程对象，同时赋予其一个名字。该构造方法只适用于通过继承来创建线程对象。

Thread(Runnable target, String name)

创建一个线程对象，在将其与一个Runnable对象相关联的同时赋予其一个名字。该构造方法只适用于通过接口创建线程对象。

void run()

运行刚刚创建的线程。程序员应当用自己的代码来覆盖这个方法。稍后我们会看到 run() 方法的默认实现，但是这个默认实现的本质是一个空方法。

void start()

创建一个新线程并执行线程类中定义的 run() 方法。

void stop() (deprecated in Java 2)

停止一个运行的线程。

static void sleep (long milliseconds)

使得正在运行的线程睡眠指定的毫秒。该方法是静态的，因此可以通过 Thread 类名来使用。

static void sleep (long milliseconds, int nanoseconds)

使得正在运行的线程睡眠指定的毫秒和纳秒。该方法是静态的，因此可以通过 Thread 类名来使用。

boolean isAlive()

确定一个线程是不是活动的。其中，活动的定义是指从一个线程真正启动前一段时间到其真正停止后一段时间内的时间。

void join()

等待指定的线程执行结束。根据定义，当一个线程不活动时，join()会立刻返回。当一个线程没有启动时，join()也会立刻返回。

void join(long timeout)

等待指定的线程执行结束，但是不超过指定的毫秒。指定的超时值会被四舍五入到实际的平台能够支持的精度。

void join(long timeout, int nanos)

等待指定的线程执行结束，但是不超过指定的毫秒和纳秒。指定的时间会被四舍五入到实际的平台能够支持的精度。

void setName(String name)

为 Thread 实例指定名字。

String getName()

获取 Thread 实例的名字。

static Thread currentThread()

获取当前运行的 Thread 对象。该方法是静态的，因此可以通过 Thread 类名调用。

static int enumerate(Thread threadArray[])

获取程序中的线程对象，并且将它们存储在一个线程数组中。返回值是实际存储在线程数组中的线程个数。该方法是静态的，因此可以通过Thread类名调用。

static int activeCount()

返回程序中线程的个数。该方法是静态的，因此可以通过 Thread 类名调用。

在本章中，我们介绍了如何创建、启动和停止线程。这些是通过 Thread 类的方法实现的。同时，我们也了解到了如何获取线程的状态、名字和程序正在使用的线程。这使得我们可以编写简单的独立线程。

但是，和线程打交道时还需要考虑其他一些事情：最主要的是，与线程进行通信时要避免竞态条件的出现。这种通信问题，也称为同步，是我们下一章要讨论的主题。

第三章

同步技术

本章内容：

- 银行的例子
- 异步读取数据
- 一个进行同步操作的类
- 同步块
- 嵌套锁
- 死锁
- 返回到银行的例子
- 同步静态方法
- 总结

在上一章，我们介绍了很多基础知识：创建线程、启动/停止线程、给线程命名以及监视线程的生命周期等等。但是上一章中的例子所使用的线程或多或少都是独立运行的：它们之间没有共享数据。

在这一章中，我们要开始了解在线程间共享数据引起的问题。因为线程共享数据会或多或少地导致竞态条件（race condition）的发生（竞态条件是指多个线程试图同时访问同一数据），所以在线程间共享数据常常是有限制的。在这一章中，我们会了解到什么是竞态条件以及一种解决竞态条件的机制。利用这种机制，我们不仅可以使多个线程共享数据，同时可以解决由线程同步引起的其他问题。在开始以前，让我们介绍一些概念。

银行的例子

假设作为某家大银行的应用程序设计员，我们要设计一个 ATM（Automated Teller Machine，自动提款机）的程序。第一个任务，就是要设计和实现一个允许用户从 ATM 上提取现金的例程。一个最初的简单设计可能如下所示（流程参见图 3-1）：

1. 检查用户的账号里是否有足够的现金来支付这次提取，如果没有，则转到第四步。
2. 将用户账号里的钱减去其要提取的金额。
3. 将现金通过 ATM 付给用户。
4. 为用户打印结果。

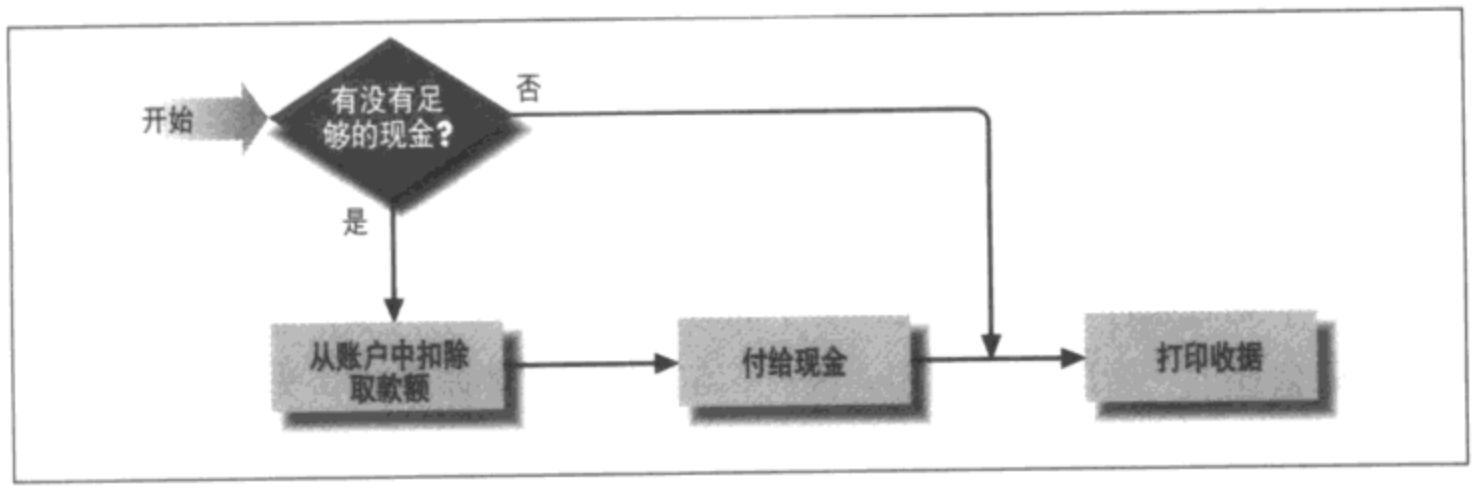

图 3-1：通过 ATM 取款的算法流程图

根据这个简单的算法，我们的实现如下：

```
public class AutomatedTellerMachine extends Teller {
    public void withdraw(float amount) {
        Account a = getAccount();
        if (a.deduct(amount))
            dispense(amount);
        printReceipt();
    }
}

public class Account {
    private float total;
    public boolean deduct(float t) {
        if (t <= total) {
            total -= t;
            return true;
        }
        return false;
    }
}
```

当然，我们假设Teller类以及getAccount()、dispense()和printReceipt()

方法都已经实现了。因为我们是想从总体上来对此算法进行研究的，所以在此就不提供这些方法的实现了。

在测试阶段，我们会运行一些简单的例子来测试上面的例程。这些例子都是从ATM中取款的，有些是提取少量的现金，有些则是提取大量的现金；有的例子中我们假设用户在账户中有足够的现金，有的例子中则假设没有足够的现金来完成交易。我们的代码看上去是通过了所有这些测试。因此我们在一台真正的ATM上进行Beta测试。

但是在实际生活中，允许两个人同时对同一个账号进行操作（例如使用夫妻卡）。所以可能发生如下的情况：两个人在同一个时刻决定清空同一个账户。这时，就出现了一个竞态条件：如果丈夫和妻子同时想提取同一个户头的全部现金，上面的例程就会在同一个时刻被调用，因此，每一个ATM就可能认为这两个用户在户头上有足够的现金，而同意对两个人都付给全部的现金。实际上，这两个用户是使用了两个线程来对存放账户信息的数据库进行访问的。

定义：原子性

原子性（atomic）的定义和原子是相关的：原子曾经被认为是最小的、不可继续分割的物质单位。因此一个原子性的操作是指在执行过程中不可被中断的操作。这种原子性可以通过硬件实现，也可以通过软件来模拟实现。通常，软件利用硬件提供的原子性指令来实现原子性的例程。

在该例中，我们定义的原子性例程就是那些不能停留在中间状态的例程。在上面的银行例子中，如果“检查账户”和“改变账户状态”动作是原子性的，那么另外一个线程就不能在第一个线程完成对账户的改变操作前去检查同一个账户的信息。

如果检查账户信息和改变账户状态不是原子性的，就会导致竞态条件。下面就是两个线程同时竞争使用同一个账户时可能产生的结果：

1. 丈夫的线程开始调用deduct()方法。
2. 丈夫的线程确认要取的现金小于或者等于账户中的现金数。

3. 妻子的线程开始调用deduct()方法。
4. 妻子的线程确认要取的现金小于或者等于账户中的现金数。
5. 妻子的线程通过deduct()方法在账户中减去要取出的钱数，返回true，因此ATM付给她现金。
6. 丈夫的线程通过deduct()方法在账户中减去要取出的钱数，返回true，因此ATM付给他现金。

Java规范提供了一些机制来特别处理这个问题。与其他的多线程系统相比，Java语言提供的synchronized关键字，允许程序员以一种类似于互斥的方式来访问某个资源。对于我们而言，这个关键字可以防止两个或者更多的线程同时调用deduct()方法。

```
public class Account {
    private float total;
    public synchronized boolean deduct(float t) {
        if (t <= total) {
            total -= t;
            return true;
        }
        return false;
    }
}
```

将deduct()方法定义为同步化（synchronized）后，如果两个用户想通过ATM在同一时刻取钱，那么第二个用户对deduct()的调用将会等待第一个用户对deduct()的调用结束后才会进行。通过保证一个时刻最多只有一个用户执行deduct()方法，可以消除竞态条件。

从表面看，同步的概念是简单的：当一个方法被定义为同步时，它就必须拥有一个被称为锁（lock）的令牌（token）。一旦获取该锁后，就可以执行这个方法了；当运行结束后，无论是正常返回还是因为发生异常而返回，都会释放该锁。因为一个对象只有一个锁，因此如果有两个线程试图调用同一个对象的某个被标记为同步的方法，那么只有一个线程能够立刻调用它，另外一个线程只有等待前一个线程对此方法的调用结束并释放对应的锁后才能调用该方法。

定义：互斥锁

互斥锁（mutex lock）也称为互斥独占锁（mutually exclusive lock）。许多线程系统都提供这种类型的锁作为一种同步方法。简单地说，它的功能就是在同一时间内只允许一个线程占有一个互斥体：如果两个线程试图获取同一个互斥体，那么只有一个会成功。另外一个线程只有等待第一个线程释放互斥体后才能获取互斥体、并继续其操作。

在Java中，系统中的每一个对象都会创建一个锁。当一个方法被声明为同步时，线程只有获取到该方法的锁后才能调用它。当调用结束时，锁机制会自动释放该锁。

异步读取数据

下面来看一个完整的例子。线程的一个主要用法就是异步地读取数据。因此，我们在这一节中要开发一个能够从网络套接字中异步读取数据的类。

为什么线程对I/O操作而言是很重要的？这是因为无论是对文件还是网络套接字进行读/写操作，都存在一个共同的问题，那就是这些操作是依赖于其他资源的。这些资源可能是其他的程序；也可能是磁盘或者网络；也可能是操作系统或者浏览器。但是这些资源可能因为各种各样的原因而暂时不可使用：从网络读取数据可能要等待数据到达；向文件中写入大量数据可能因为磁盘正在忙于处理其他的请求而要等待一段很长的时间，等等。但不幸的是，在Java API中不存在检查这些资源是否可用的方法。特别是对于网络套接字而言这更是一个大问题：因为数据可能要经过很长的时间才能通过网络传送完毕，使得通过套接字读取数据的程序可能在套接字上等待很长的时间。

为什么使用异步I/O？

有一种动力在背后推动着异步I/O技术，那就是人们期望程序在等待数据到来的同时还能够继续进行其他有用的工作。在applet中，如果不是使用一个单独的线程来进行异步的I/O操作，我们就会碰到前面章节中讨论的问题：鼠标和键盘事件会被延迟处理。在I/O处理结束以前，程序看上去是停止响应了。

InputStream()类的确包含available()方法。但是并不是所有的输入流都支持这个方法。同时，在慢速网络的情况下，向套接字写数据也可能要耗费很长时间。通常，使用available()方法来进行检查相对于创建一个新的线程来读取数据而言，效率更低，而且也更难于编程。

解决这个问题的方法是使用另外一个线程。假设我们在applet中使用一个新的线程：因为这个新的线程独立于applet的主线程，所以它的阻塞不会导致applet的主线程被阻塞。当然，这会带来一个新的问题：当该线程最终可以读取数据时，这些数据一定要返回给applet线程。下面我们来看一个通用的通过套接字读取数据的线程的可能实现：

```
import java.io.*;
import java.net.*;

public class AsyncReadSocket extends Thread {
    private Socket s;
    private StringBuffer result;

    public AsyncReadSocket(Socket s) {
        this.s = s;
        result = new StringBuffer();
    }

    public void run() {
        DataInputStream is = null;
        try {
            is = new DataInputStream(s.getInputStream());
        } catch (Exception e) {}
        while (true) {
            try {
                char c = is.readChar();
                appendResult(c);
            } catch (Exception e) {}
        }
    }

    // 获取已经从套接字读取的字符串
    // applet线程以同步的方式使用此方法获取数据
    public synchronized String getResult() {
        String retval = result.toString();
        result = new StringBuffer();
        return retval;
    }
```

```
        // 将新数据放入缓冲区
        // 这些数据将被 getResult 方法返回
        public synchronized void appendResult(char c) {
            result.append(c);
        }
    }
```

该例中的 AsyncReadSocket 类是一个 Thread 类，它的 run() 方法从套接字中读取字符。只要它读取到数据，就会将读取的字符添加到字符串缓冲区 result 中。即使该线程在读取数据的过程中发生阻塞，也不会影响程序中的其他线程。applet 可以通过 getResult() 方法来获得这个新线程读取的数据。如果调用 getResult() 时没有数据可返回，就返回空串。同时当 applet 的主线程忙于其他任务时，该线程将读取的数据存放起来。换句话说，通过套接字读取数据的线程随时将读取到的数据存储在某个变量中，而 applet 的主线程则可能随时通过 getResult() 来取得这些数据。这样，applet 就不必担心发生阻塞或者数据丢失的情况。这两个线程实际运行的结果可能如图 2-3 所示。

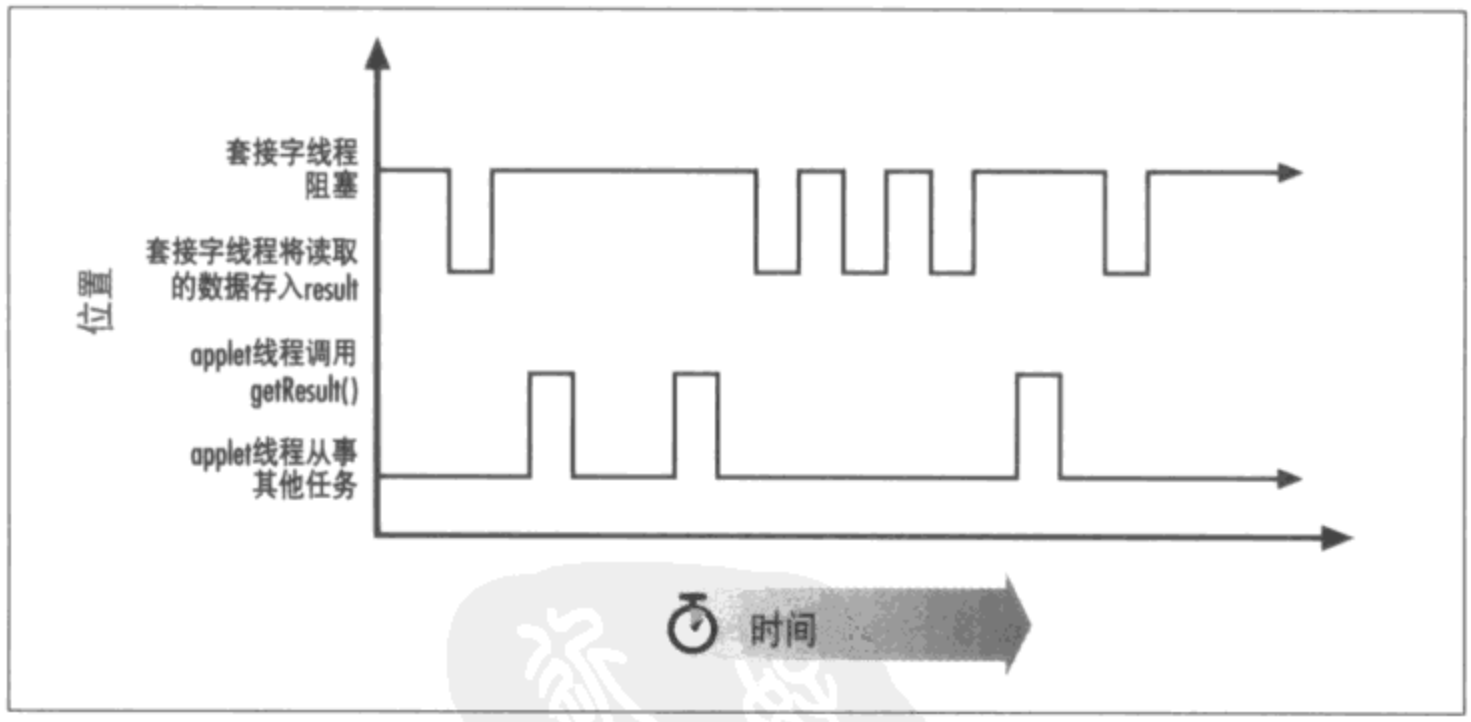

图 3-2: applet 的运行图示

正如我们所做的，线程编程的一个吸引人之处在于可以很容易地编写许多小的、独立的任务。同时因为这些线程都是存在于一个程序中的，所以它们之间的通信就和一个程序中两个方法间的通信一样简单。只需要一个每个线程都可以访问的公共引用，就可以使这些通信得以完成。本例中，该公共区域就是实例变量 result。

要注意的是，如果不使用synchronized关键字来防止applet的主线程和从套接字读取数据的线程同时访问result缓冲区，该例子就不能正确工作。那样做会发生竞态条件。特别是，如果getResult()和appendResult()方法不是同步的话，就可能发生下面的情况:

1. applet的线程进入getResult()方法。
2. applet将通过result来创建retval，并且将内容拷贝到retval中去。
3. 通过套接字读取数据的线程从readChar()方法返回。
4. 通过套接字读取数据的线程通过appendResult()方法将读取到的数据添加到result中。
5. applet的主线程将result清空。

这时，在第四步添加到字符串缓冲区中的数据就会丢失了：这些数据没有在第二步被applet所获取，同时在第五步随着原来的字符串缓冲区一起被释放了；同时，还存在另外一种竞态条件：如果两个线程同时调用getResult()方法，它们可能从同一个字符串缓冲区读取数据，这就使得同一个数据被读取和处理两次。

如果所有对result的操作都是原子性的，就可以解决这些竞态条件的问题。我们所要做的就是保证所有访问result变量的方法都是同步的。

既然我们提出了这么多的问题，那么在继续讨论以前，让我们先来回答其中一些问题。

为什么当一个对象的两个方法被定义为同步后，当一个线程调用其中一个方法时，另一个线程对另外一个方法的调用会被阻塞呢？我们已经在前面说过，同步一个方法使得对这个方法的调用只能以线性的方式进行。这就意味着当一个线程在调用此方法时，另外一个线程是不可能执行该方法的。但是，这种机制是通过对该对象加锁来实现的。一个方法被调用后，加在其上的锁就不可能被其他的线程获取，因此其他的线程就不可能同时调用该方法了。如果同一个对象中被定义为同步的两个方法也同时被多个线程调用，因为这两个方法是共享同一个对象锁的，所以也会有同样的结果。换句话说，即使有两个或者多个方法同时被不同的线程

什么时候竞态条件会引起问题？

当两个或者多个线程执行顺序的不同会影响到某些变量的值或者程序的输出时，我们就认为发生了竞态条件。也许一个应用程序的所有可能的线程运行顺序都会得到同样的结果，但是这并不表示没有发生竞态条件。可能竞态条件引起的影响是微不足道的，或者是根本没有被注意到。例如，在AsyncReadSocket类中丢失一个字符也许不会影响最终的结果；另一方面，系统的调度可能使得潜在的竞态条件根本就没有机会出现。

竞态条件只是潜伏的问题，因此对算法进行一些简单改变就可能使得竞态条件以一种破坏性的方式表现出来。另外，不同的虚拟机可能以不同的顺序来执行线程。所以，尽管程序中存在的竞态条件可能在开发环境中不产生问题，但并不能保证在其他情况下不会产生问题，因此程序员不应该让程序中存在竞态条件。

调用，它们也不可能在不同的线程中并行执行。如图3-3所示：当线程1和线程2试图获取同一个锁（L1）时，线程2必须等待线程1释放了该锁后才能继续运行。

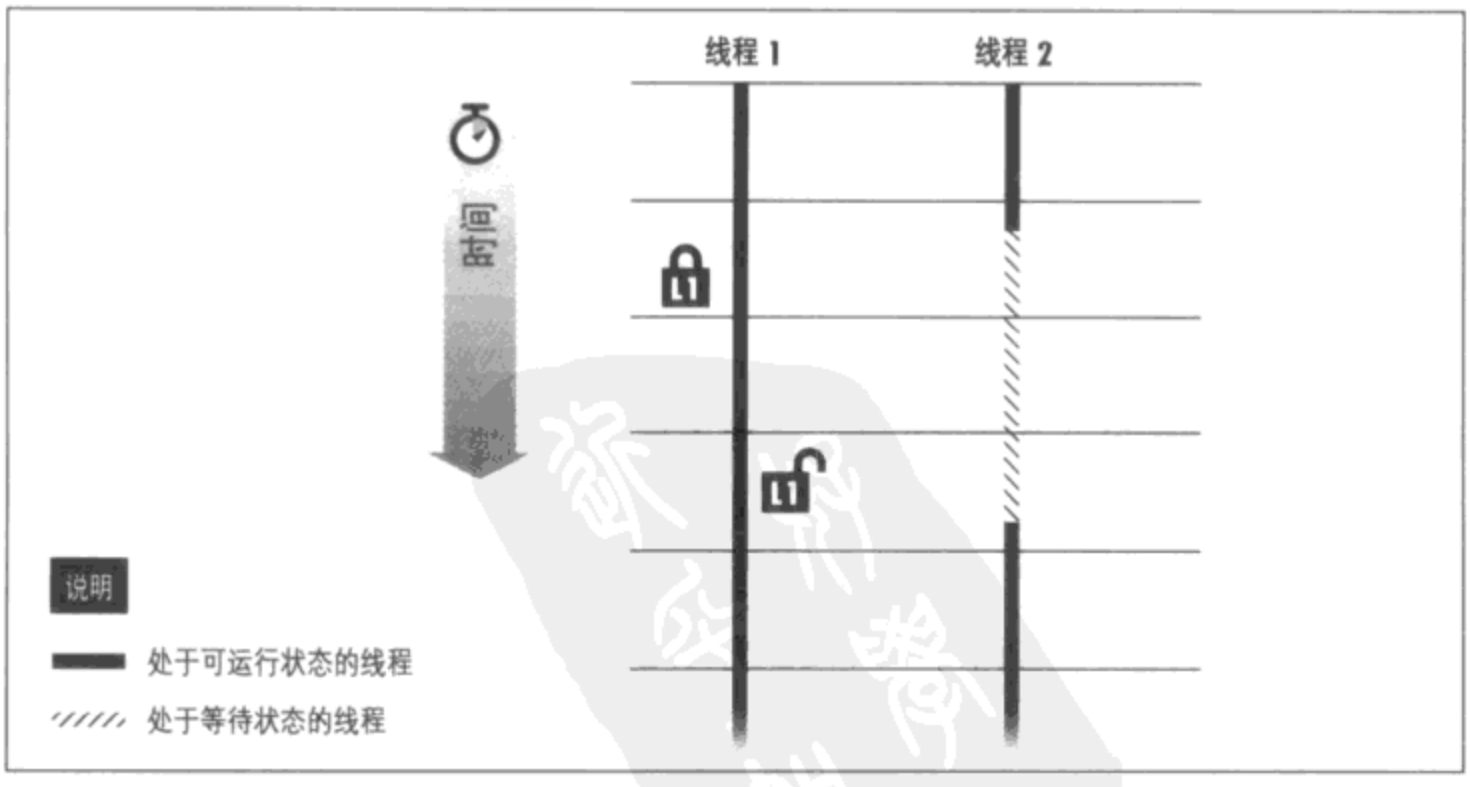

图 3-3：获取和释放锁

值得注意的是，锁是基于一个特定的对象，而不是特定的方法的。假设我们在不同的线程中创建了两个 AsyncReadSocket 对象：a 和 b 。一个线程执行 a.getResult()方法而另外一个执行 b.getResult()方法，则这两个调用是可以并行执行的。这是因为对于a.getResult()的调用是要获取实例对象 a 的锁，而对于 b.getResult()的调用是要获取实例对象 b 的锁。因为 a 和 b 是两个不同的对象，因此这两个线程获取的是不同的锁：两个线程的运行都不需要等待另外一个线程运行结束。

为什么我们需要 appendResult()方法？为什么我们不能简单地将这些代码放到 run()方法中去，而将 run()方法置为同步的呢？当然我们可以这样做，可是结果会是灾难性的。每一个锁都是有作用域的，该作用域就是该锁有效的代码范围。将run()方法同步会使得该锁的作用域过大，从而使得其他的方法没有办法运行。

定义：锁的作用域

锁的作用域（scope of a lock）就是从锁被获取到其被释放的时间。在上面的例子中，仅仅使用了同步方法。这意味着这些锁的作用域就是方法运行的时间。这称为方法的作用域（method scope）。

在本章的后面，我们会介绍如何在方法中对任意一块代码使用锁或者显式地获取和释放锁。这些锁有不同的作用域。随着对各种不同类型的锁的讨论，我们会逐步介绍它们的作用域。

因为 run()方法是在一个无限循环中执行的，所以 run()方法的作用域可以看成是无限的。如果 run()方法和 getResult()方法都被定义为同步的，它们就不能在不同的线程中并行执行。这是因为run()的一个任务就是打开套接字，并且在套接字关闭以前一直从中读取数据，这就使得在套接字被关闭以前run()方法拥有该对象锁。这也同样意味着，当套接字是打开时不能使用getResult()方法。这可不是我们对这个可以异步读取数据的类所期望的结果。

一个同步的方法如何与非同步的方法共同工作呢？简单地说，一个同步方法在运行前会试图去获取一个对象锁，而非同步方法不这么做。这意味着许多非同步方法可以并行地与一个同步方法一起运行。但是同一时刻只能有一个同步方法运行。

同步一个方法意味着在运行该方法时要获取一个锁。所以程序员的责任就是确保恰当的方法被同步。如果忘记同步一个应该同步的方法，就可能会产生竞态条件：例如，如果我们仅仅同步了getResult()方法，而没有同步appendResult()方法，那么getResult()方法可能和appendResult()方法同时被调用，所以还是没有解决竞态条件。

一个进行同步操作的类

为什么要使用新的关键字来解决竞态条件问题？可不可以通过重新设计算法来解决竞态条件问题？我们来看看能不能通过使用试错法来重写AsyncReadSocket类以解决竞态条件问题（很明显这不是最好的编程技术，但是对于我们的讨论是很有用的）。因为每一次试错都包含两个步骤：检查和设置变量，所以我们可以得出这样的结论——在虚拟机没有提供直接支持的情况下，竞态条件是不可避免的。如果虚拟机不能保证在对某个变量进行测试和设置新值之间不会出现对该变量的操作，竞态条件就会发生。但是对于如何避免竞态条件的研究有助于我们以后要开发的重要工具——BusyFlag类的实现。

乍看起来，避免两个线程同时改变result变量或同一个缓冲区的最简单方法就是使用忙标志。如果线程需要访问result变量，它就必须将标志设为忙状态。如果标志为忙，线程就必须等待直到该标志变为闲。当标志为闲时，该线程就可以将其设置为忙。然后该线程就可以安全地使用这个缓冲区了。一旦线程完成了它的任务，就可以将该标志设置为闲。

为什么要使用 BusyFlag 类？

使用 BusyFlag 类来解决竞态条件看起来更像是一个理论问题：为什么要使用 BusyFlag 类来代替同步机制？

对于我们到目前为止所碰到的问题而言，这种代替不是必要的。但是对于其他情况，其中的一个答案就是锁的作用域：因为同步机制不允许我们对特定作用域的代码加锁。后面我们会碰到同步机制不能解决锁的作用域问题的情况。另外，BusyFlag类对于实现本书其他部分要提到的一些机制也是有用的。

下面是BusyFlag类的一个可能实现。

```
public class BusyFlag {
    protected Thread busyflag = null;

    public void getBusyFlag () {
        while (busyflag != Thread.currentThread()) {
            if (busyflag == null)
                busyflag = Thread.currentThread();
            try {
                Thread.sleep(100);
            } catch (Exception e) {}
        }
    }

    public void freeBusyFlag () {
        if (busyflag == Thread.currentThread()) {
            busyflag = null;
        }
    }
}
```

该BusyFlag类包含两个方法。getBusyFlag()方法在当前线程中循环直到可以将busyflag设置为当前线程。如果busyflag被其他线程设置了，线程会等待100毫秒后再试。而只要该标志为null，线程就会立刻将busyflag设置为当前线程。另外一个方法freeBusyFlag()通过将该标志设置为null来释放该标志。这个实现看上去简单而优美地解决了我们面临的问题。但实际上并非如此。

为什么线程需要睡眠100毫秒？因为除了循环轮询外似乎没有其他方法来检测标志位的改变。但是，如果在循环轮询中不使用sleep()方法的话，就会浪费其他线程可以使用的CPU周期。从另一个极端来看，就算在一开始没有其他线程拥有该标志，也最少需要100毫秒来设置它。一种简单的处理方法是将线程睡眠的时间设置为一个变量，但是无论我们将线程睡眠的时间设定为多少，都不得不在设定标志所需要的时间和在循环中耗费的CPU周期之间取得平衡。

为什么线程在标志没有被设置时还是要睡眠100毫秒呢？这样做是有原因的：在检查标志是否为null和设置标志之间存在一个竞态条件。如果两个线程都发现标志为null，那么它们都可以设置该标志并且退出循环。但是通过调用sleep()方法，我们就允许两个线程在通过while循环再次检查该标志之前设置

busyflag。这样只有第二个线程才能够成功地设置该标志并且退出循环，从而可以退出 getBusyFlag()方法。

当然，这仍然存在着问题。有可能存在如下的执行顺序:

1. 线程 A 检测到 busyflag 为闲。
2. 线程 B 检测到 busyflag 为闲。
3. 线程 B 设置 busyflag。
4. 线程 B 睡眠 100 毫秒。
5. 线程 B 醒来，确定其拥有 busyflag，退出循环。
6. 线程 A 设置 busyflag，睡眠，醒来，确定其拥有 busyflag，退出循环。

虽然这种情况是极为罕见的，但还是有可能发生。因此，上面的代码不是大多数程序员所期望的。

我们可以在 Account 类中使用 BusyFlag 类来代替同步方法:

```
public class Account {
    private float total;
    private flag = new BusyFlag();

    public boolean deduct(float t) {
        boolean succeed = false;
        flag.getBusyFlag();
        if (t <= total) {
            total -= t;
            succeed = true;
        }
        flag.freeBusyFlag();
        return succeed;
    }
}
```

在绝大多数情况下，BusyFlag 类都可以很好地工作。也许该程序在 100 台 ATM 上进行一年的beta测试都没有问题，但是你愿意用你的职业生涯作为赌注，保证这个使用 BusyFlag 类的 AutomatedTeller 类不会出现问题吗?

如果有多个线程都来同时设置busyflag标志，将会有什么样的结果呢？对于busyflag变量的设置是不是原子操作呢？Java规范已经保证了，除了double和long类型以外，对于其他类型的变量进行赋值的操作都是原子性的。所以对于本例而言，多个线程试图同时对busyflag进行赋值操作是不会产生什么问题的。但是如果两个线程试图去同时改变一个long或者double类型的数据，就有可能得到错误的结果：该变量的某些位是由第一个线程设置的，而其他位却是由另外一个线程设置的。然而，原子性并不能确保线程通信；参见附录一中对volatile的讨论。

我们可不可以通过使用同步原语来解决BusyFlag类面临的问题呢？其实我们在BusyFlag类中所碰到的问题和我们试图用BusyFlag去解决的问题是同一个问题。这就意味着如果可以通过使用同步原语来解决BusyFlag类中的问题，也就可以用BusyFlag来解决其他的竞态条件问题，并且不用担心该解决方案会在某种条件下发生问题。下面就是我们对于该解决方案的实现（没有经过优化的）：

```
public class BusyFlag {
     protected Thread busyflag = null;
     public void getBusyFlag() {
          while (tryGetBusyFlag() == false) {
               try {
                    Thread.sleep(100);
               } catch (Exception e) {}
          }
     }

     public synchronized boolean tryGetBusyFlag() {
          if (busyflag == null) {
               busyflag = Thread.currentThread();
               return true;
          }
          return false;
     }

     public synchronized void freeBusyFlag() {
          if (busyflag == Thread.currentThread()) {
               busyflag = null;
          }
     }
}
```

在该BusyFlag类的实现中，我们引入了一个新的方法：tryGetBusyFlag()。它在本质上与getBusyFlag()方法是一致的，但是这个新方法并不等待标志被释

放。如果标志为闲，它就设置标志然后返回true。否则返回false。你可能已经注意到该方法被声明为同步的。这就意味着系统会保证任何线程在试图调用tryGetBusyFlag()方法前都必须获取该方法的对象锁。

同样，freeBusyFlag()方法也被声明为同步的：任何线程在试图调用该方法前都必须获取该方法对应的对象锁才能继续运行。因为对于该类的每一个实例而言，仅仅存在一个锁，所以freeBusyFlag()方法和tryGetBusyFlag()方法使用的是同一个锁。这就意味着不可能在线程试图获取和释放busyflag时产生竞态条件。

同步块

我们注意到原先的getBusyFlag()方法没有被声明为同步的。这是因为getBusyFlag()并不试图访问busyflag变量。实际上，getBusyFlag()通过调用tryGetBusyFlag()方法来访问busyflag变量。当然，tryGetBusyFlag()是被声明为同步的。让我们再来看看没有调用tryGetBusyFlag()方法的getBusyFlag()方法，该版本的getBusyFlag()方法直接访问busyflag变量：

```
public synchronized void getBusyFlag() {
    while (true) {
        if (busyflag == null) {
            busyflag = Thread.currentThread();
            break;
        }
        try {
            Thread.sleep(100);
        } catch (Exception e) {}
    }
}
```

假设我们不希望额外调用tryGetBusyFlag()而带来效率上的损失。因此在新的getBusyFlag()方法中，我们直接对busyflag进行操作。新版本的getBusyFlag()方法循环检测直到标志被释放，然后设置标志并且返回。因为我们现在是直接访问busyflag了，所以不得不将该方法设置为同步的，否则就会产生竞态条件。

但不幸的是，将该方法声明为同步的会引起另外一个问题。尽管将该方法设置为同步的可以防止getBusyFlag()方法和tryGetBusyFlag()方法同时被调用（这可以防止竞态条件的发生），但这样做也同时阻止了freeBusyFlag()方法的运行。如果getBusyFlag()被调用时，标志为忙状态，getBusyFlag()会循环等待该标志被释放。但是因为freeBusyFlag()方法只有等到getBusyFlag()释放了对象锁之后才能够运行，所以没有办法释放标志。这种两难的情况称为死锁(deadlock)。该例子中的死锁是锁和标志之间的问题。在更常见的情况下，死锁会发生在两个或者多个锁上，但是原理是一样的。

死锁的例子

我们在本章的后面和第八章中将会讲解有关死锁的细节知识。但是在我们继续以前，先来看一个例子。

假设我们在一家银行排队。我处于队首，并且是准备取钱的。假设此时银行的现金用完了，而我实际上愿意等其他人存入一些钱后再取出我所需要的钱。而同时还假设此时只有一个出纳员，并且该银行有这样一条规定：在当前交易没有完成以前，是不能进行下一个交易的。因为我还在等待取出自己的现金，所以我的交易是没有完成的。

假设你就在我后面，而且准备存一百万元钱。很明显，在我取完钱以前，你是不能存钱的。而你如果不将钱存到银行里去，我又没有办法取到钱。当然这个例子是一个非常牵强的例子，而且是通过常识就可以解决的问题。但是这就是在BusyFlag类中发生的问题。更进一步地说，这是一个很隐秘的问题，因此我们很可能在测试时没有发现它，就如同银行在有充足的现金储备时测试它的政策是不会发现这种死锁的。

在这个版本中，正是因为锁的作用域太大了，才会导致getBusyFlag()中的问题。我们所要做的就是将锁的作用域限定在我们要改变数据的范围内（即检查和获取busyflag）；并不需要在整个方法中一直持有该锁。幸运的是，Java提供了对代码块而不是整个方法进行同步的方法。通过仅仅对getBusyFlag()的部分代码使用同步块机制，就有了如下的程序：

```
public void getBusyFlag () {
    while (true) {
```

```
            synchronized (this) {
                if (busyflag == null) {
                    busyflag = Thread.currentThread();
                    break;
                }
            }
            try {
                Thread.sleep(100);
            } catch (Exception e) {}
        }
    }
```

在getBusyFlag()方法的新实现中，我们仅仅同步了那些对busyflag标志进行检测和设置（如果该标志为闲）的代码。这种用法与方法同步是很类似的，只是锁的作用域要小多了。

有趣的是，这种用法使我们不仅可以更加精确地控制对象锁，而且可以选择要获取哪个对象的对象锁。在本例中，因为是希望在方法tryGetBusyFlag()和freeBusyFlag()中使用同一个对象锁，因此使用this（对象本身）来获取对象锁。对于同步的方法而言，它们所拥有的锁就是该方法所属的类的对象锁，换句话说，也就是this对象。

嵌套锁

再次以BusyFlag类为例。假设我们增加一个方法来指明是哪个线程拥有锁。getBusyFlagOwner()方法仅仅返回busyflag的值。当然，该变量的值其实就是拥有该锁的线程对象的值。该方法的实现如下：

```
public synchronized Thread getBusyFlagOwner() {
    return busyflag;
}
```

更进一步，如果我们使用getBusyFlagOwner()方法来修改freeBusyFlag()方法的话：

```
public synchronized void freeBusyFlag () {
    if (getBusyFlagOwner() == Thread.currentThread()) {
        busyflag = null;
    }
}
```

对象或引用

随着同步块的引入，我们也能选择使用什么对象来同步一段代码。此时必须要注意一个物理对象和一个引用对象的实例变量之间的区别。

在BusyFlag类中，我们可以在getBusyFlag()、tryGetBusyFlag()和freeBusyFlag()方法中使用代码块同步机制。这允许我们使用任何对象作为锁对象。

但是变量busyflag不是一个好的选择。这三个方法在执行过程中改变它的值，其中就包括将其设置为null。而对一个null对象加锁会产生异常，并且对不同的对象加锁也违背了同步的初衷。

这看起来是很清楚的，但是一个经常发生的错误就是选用了错误的锁对象，因此再次重申：

> 同步是基于实际对象而不是对象引用的。多个变量可以引用同一个对象，变量也可以改变其值从而指向其他的对象。因此，当选择一个对象锁时，我们要根据实际对象而不是其引用来考虑。

作为一个原则，不要选择一个可能会在锁的作用域中改变值的实例变量作为锁对象。

在freeBusyFlag()方法的新版本中，我们通过调用getBusyFlagOwner()方法来判断拥有锁的线程是不是当前线程，如果是的话，才释放busyflag。有趣的是，freeBusyFlag()方法和getBusyFlagOwner()方法都是同步的，这会产生什么样的结果呢？线程会不会因为等待freeBusyFlag()方法来释放锁而阻塞在getBusyFlagOwner()方法中？如果不是，而且getBusyFlagOwner()方法也能够运行，那么当该方法运行完后会得到一个什么样的结果呢？会不会在freeBusyFlag()方法还需要这个锁的时候已经将该锁释放了？所有这些问题的答案都取决于你希望该程序是如何运行的。

一个同步区域（可能是同步块，也可能是同步方法）并不是盲目地在进入该区域时获取锁，在离开该区域时释放锁。如果当前线程已经拥有了该对象锁，就没有必要去等待该锁被释放或者是再次获取该锁。实际上，同步区域里的代码会直接

同步方法和同步代码块的对比

尽管有时候需要同步整个方法，但是实际上可能只需要使用同步块机制。在本书中，为了清楚起见，我们使用同步方法（synchronized method）机制来同步整个方法，而在其他情况下使用同步块（synchronized block）机制。但是在需要同步整个方法时，程序员可以根据自己的喜好选择同步方法机制或者是同步块机制。

同步方法机制是最简单的，但是正如我们前面所见的，因为其锁的作用域可能太大，有可能引起死锁。同时因为可能包含了不需要进行同步的代码块在内，也会降低程序的运行效率。

使用同步块机制时，如果使用过多的锁，也会引起问题。在后面我们会看到，如果要获取多个锁，也很容易引起死锁。同时获取和释放锁也有代价，因此在释放一个锁后的几行代码中就再次获取它也是低效的。

理论学家可能会使用整整一章来讨论这个问题。但是我们仅仅关心如何使代码清晰。因此我们是针对具体的例子来选择同步机制的。而有关如何在这两种机制之间进行选择的其他原因都超出了本书的范围。

运行。而且，系统也不会在退出该区域时释放锁（因为在进入该区域时并没有获取锁）。这意味着freeBusyFlag()方法可以安全地调用getBusyFlagOwner()方法。

但不幸的是，我们的BusyFlag类不是如此地聪明的。它会阻塞在自己拥有的锁上。为了解决这个问题。我们必须在BusyFlag类中实现一个计数器。对象必须检查自己是不是拥有锁，并且当已经拥有了锁时就仅仅对该计数器加一。在对应的freeBusyFlag()方法中，它会将计数器的值减一，当值为零时才真正释放该锁。这样，当一个线程直接或者间接地（通过方法调用）处在某个BusyFlag锁的作用域内时，就可以安全地进入由同一个BusyFlag锁实例所锁定的其他范围中去。

下面是修改过的BusyFlag类（也没有优化）：

```
public class BusyFlag {
    protected Thread busyflag = null;
```

```
protected int busycount = 0;
public void getBusyFlag() {
    while (tryGetBusyFlag() == false) {
        try {
            Thread.sleep(100);
        } catch (Exception e) {}
    }
}

public synchronized boolean tryGetBusyFlag() {
    if (busyflag == null) {
        busyflag = Thread.currentThread();
        busycount = 1;
        return true;
    }
    if (busyflag == Thread.currentThread()) {
        busycount++;
        return true;
     }
    return false;
}

public synchronized void freeBusyFlag () {
    if (getBusyFlagOwner() == Thread.currentThread()) {
        busycount--;
        if (busycount == 0)
            busyflag = null;
    }
}

public synchronized Thread getBusyFlagOwner() {
    return busyflag;
}
}
```

使用这个新版本的BusyFlag类，我们就可以锁定任何范围的代码而不需要担心已经拥有该锁了。同样释放锁时也是如此。同步机制和我们的BusyFlag类都可以作为嵌套锁使用。[BusyFlag类现在类似于另一个称为信号量（semaphore）的同步原语。]

死锁

虽然一个线程在获取一个锁以前检查它是不是拥有该锁不是一个很困难的问题，

但是这是否能够防止死锁的发生呢？在我们试图回答这个问题以前，先来看看到底是什么死锁。简单地说，死锁就是两个或者多个线程等待两个或者多个锁被释放，而程序又处于一个这些锁永远无法被释放的情况中。在将整个getBusyFlag()方法定义为同步时我们已经见过这种情况了。如果将 freeBusyFlag()方法定义为同步，也同样会导致在 getBusyFlag()返回后才能够将锁释放的情况。因为getBusyFlag()方法要等待 busyflag 被释放，而该锁又不可能被释放，因此就会永远处于等待状态中。

这种死锁是由于Java的同步原语要获取一个对象锁和BusyFlag类的锁机制造成的。仅仅使用Java的同步原语会不会也造成这样的死锁现象呢？答案是肯定的。更进一步说，预测和检测死锁的发生是很困难的。在测试时运行良好的代码也可能包含了死锁，而这种死锁仅仅在某种特定情况下或者在某个特定的Java虚拟机中才会发生。为了更好地说明这个问题，下面研究一些在任何数据库系统中都会存在的方法：

```
public void removeUseless(Folder file) {
     synchronized (file) {
          if (file.isUseless()) {
               Cabinet directory = file.getCabinet();
               synchronized (directory) {
                    directory.remove(file);
               }
          }
     }
}
```

假设某个数据库类有名为removeUseless()的方法。当程序清理数据库系统时会调用这个方法。它的参数是表示数据库系统中某些文件夹的对象。文件夹对象有一个名为isUseLess()的方法来判断该文件夹是否还有用。为了对某个文件夹进行操作，我们必须保证拥有该文件夹的对象锁。如果发现某个文件夹不再有用，我们可以通过 getCabinet()方法来获取对应的机柜（cabinet），然后通过调用remove()方法来删除该文件夹。和文件夹对象一样，也要先获取机柜的对象锁后才能对机柜对象进行操作。同时，假设我们还有另外一个名为updateFolder()的方法：

```
public void updateFolders(Cabinet dir) {
     synchronized (dir) {
```

```
                for (Folder f = dir.first(); f != null; f = dir.next(f)) {
                    synchronized (f) {
                        f.update();
                    }
                }
            }
        }
```

该方法的参数是表示数据库系统中一个机柜的对象。为了对该机柜进行操作，我们也必须先获取该对象的对象锁。假设对于一个机柜的更新是通过遍历该机柜所拥有的所有文件夹并且调用相应的update()操作来完成的。同样，在更新文件夹前也必须获取每个文件夹对象的对象锁。

这两个方法都没有什么特别的地方；在任何一个数据库系统中它们都可能以这样或者那样的形式存在。但是我们可能像图 3-4 那样运行它们。假设线程 1 调用updateFolders()方法，从而获取了机柜的锁（L1）。同时假设线程 2 调用removeUseless()方法，从而获取了文件夹的锁（L2），并且检测出该文件夹不再有用了，从而试图获取机柜的锁（L1）来删除该文件夹。在这个时候，线程 2 阻塞并在 L1 上等待，直到机柜对象锁被释放。

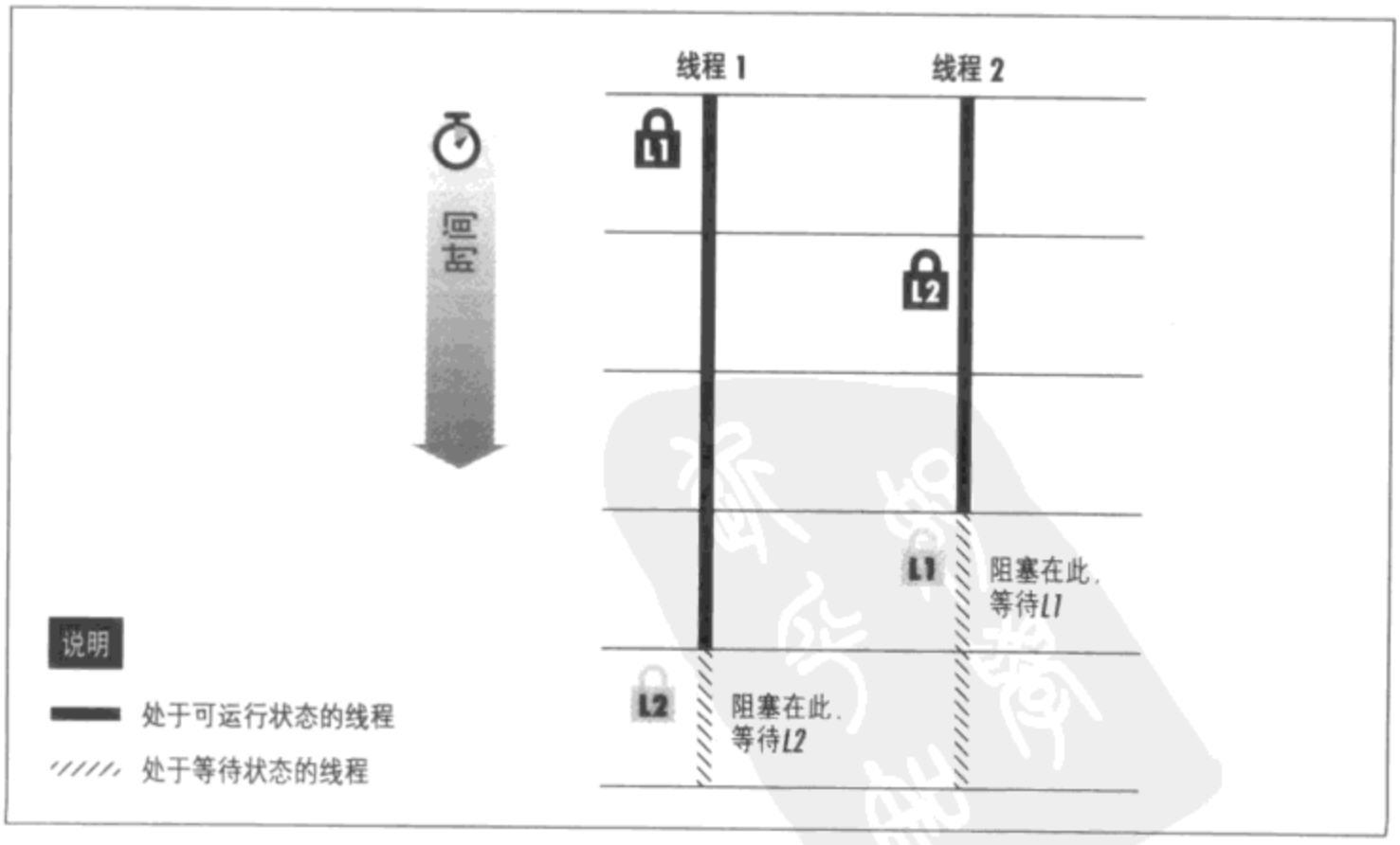

图 3-4：数据库系统中的死锁

但是如果 removeUseless()方法正在使用的文件夹正好是 updateFolers()方法所使用的，情况又会如何呢？当updateFolders()方法遍历到这个文件夹时，它试图去获取该文件夹的锁（L2）。现在，removeUseless()方法拥有了文件夹的锁，而等待机柜的锁被释放；updaeFolders()方法拥有机柜的锁而等待文件夹的锁被释放。这是一种经典的死锁情况。这种情况容易发生但是很难被检测到：每一个方法都使用简单而清晰的算法，而且代码中也没有明显会造成死锁的迹象。想想在一个由多个程序员开发的大系统中，如果开发这些代码的程序员互相不了解对方的代码，会出现什么情况。就算是设计得最好的程序也不能保证死锁不会发生。

系统能不能像阻止由于同一个线程试图重复获取同一个锁而导致的死锁那样来防止这种死锁的发生呢？不幸的是，这个问题是完全不同的。在嵌套锁的情况中，是一个线程试图两次获取同一个锁；而这是两个不同的线程试图去获取不同的锁。因为一个线程已经拥有了其中的一个锁，因此不可能释放该锁。为了解决这个问题，我们要么重新设计这个程序使得它不会进入出现死锁的情况，要么通过编程来避免死锁的发生。无论哪种方法，都需要对程序进行重新设计。根据原来设计的复杂程度，也许需要对原来的数据库系统进行重新设计。

如果程序员不重新设计程序，Java系统能不能自动解决死锁问题呢？答案是否定的。在第八章中我们会了解到如何通过程序设计来防止死锁发生。

返回到银行的例子

现在，我们解决了 ATM 的取款功能中的问题。我们证明了这个问题的出现是一个小概率事件，而且这次交易所涉及的钱只有几千块而已。更幸运的是，由于银行有很好的记录，因此多付的钱也要回来了。同样，我们的经理不想让这个小问题来打搅高层领导，同时我们对于该问题的穷追猛打也给她留下了很好的印象。因此，尽管她不是完全信任我们，我们还是获得了增强 ATM 系统其他部分的工作。

第一步就是看看我们现有的 ATM 代码：我们检查现有的代码来找出可能的竞态条件，而且使用我们已经知道的同步机制来解决这些问题。看起来每件事都是完

美的，但是有一天一位生气的顾客打来的电话却破了这一切。这位顾客通过ATM系统查询到他的账户中还有300元钱，但是当他试图取出300元时，系统却不允许他这么做。

后来证明了就在该顾客查询和取钱的间隔中，他的妻子通过另外一部ATM取出了100元。尽管系统的运行是正确的，但是顾客认为银行没有给予他正确的信息而威胁要将储存在该银行的一百万元取出。于是银行制定了一项新的政策：在同一时间，只能有一部ATM可以对同一个账户进行操作。

这就意味着我们需要对账户使用新的锁作用域：ATM类必须在用户会话期间对账户加锁。该会话可能包含多个交易并且会跨ATM类的多个方法。因为现在的锁作用域是跨多个方法的，因此原来的同步方法和同步代码块都不适用了。

幸运的是，我们已经有了BusyFlag类，所以只需要小小的改动就可以解决这个问题。

```
public class AutomatedTellerMachine extends Teller {
    Account a;

    public boolean synchronized login(String name, String password) {
        if (a != null)
            throw new IllegalArgumentException("Already logged in");
        a = verifyAccount(name, password);
        if (a == null)
            return false;
        a.lock();
        return true;
    }

    public void withdraw(float amount) {
        if (a.deduct(amount))
            dispense(amount);
        printReceipt();
    }

    public void balanceInquiry() {
        printBalance(a.balance());
    }
```

```
    public void synchronized logoff() {
        a.unlock();
        a = null;
    }
}

class Account {
    private float total;
    private BusyFlag flag = new BusyFlag();

    public synchronized boolean deduct(float t) {
        if (t <= total) {
            total -= t;
            return true;
        }
        else return false;
    }

    public synchronized float balance() {
        return total;
    }

    public void lock() {
        flag.getBusyFlag();
    }

    public void unlock() {
        flag.freeBusyFlag();
    }
}
```

通过使用BusyFlag锁，我们就可以在会话作用域使用锁了。在用户进入ATM时加上busyflag锁，而当用户离开ATM时释放busyflag锁。Java的同步原语是不能直接提供这种作用域的锁的。

现在我们可以自豪地将BusyFlag类放入我们的类库中了，这使得ATM开发组的其他程序员也都可以使用该类。这是一个很简单的类，但是在ATM系统中已被广泛使用。但是我们还是要指出，尽管现在的BusyFlag是正确的，但是并没有经过优化。在下一章中我们将讨论如何解决这个问题。

同步静态方法

在本章关于同步的讨论中，我们总是使用"获取对象锁"这句话。但是对于静态方法又该如何呢？当调用一个同步的静态方法时，引用的是哪个对象呢？一个静态方法没有this引用的概念，而我们又不能使用一个不存在的对象的对象锁。那么静态方法的同步是如何工作的呢？为了解决这个问题，我们首先引入类锁（class lock）的概念。就像可以对类的每一个实例（对象）获取一个对象锁一样，对于每一个类都可以获取一个锁。我们称之为类锁。对于实现而言，并不存在类锁这样的东西，但是这个概念有助于我们理解这是如何工作的。

当一个静态同步方法被调用时，程序会先试图获取类锁。除了是不同的锁以外，这种机制和调用非静态同步方法是一样的。它们也遵守同样的规则：如果一个同步静态方法调用同一个类中的其他同步静态方法，系统也能正确处理嵌套锁。

但是类锁和对象锁之间有什么关系呢？除了功能上的相似外，这两种锁之间没有任何其他的联系。这是两种不同的锁。类锁的获取和释放独立于对象锁。如果一个非静态同步方法调用一个静态同步方法，它就会拥有这两种锁。因为一个静态方法在没有对象引用的情况下不能调用一个非静态方法，所以在这两种锁之间发生死锁的可能性很小（但不是没有）。

如果一个同步静态方法拥有一个对象引用，它是否可以调用该对象的同步方法或者使用该对象来锁住一段同步代码？答案是肯定的。下面的例子就是通过调用一个同步静态方法来获取类锁，然后使用传递进来的对象来获取对象锁。

```
public class MyStatic {
    public synchronized static void staticMethod(MyStatic obj) {
        // 获得了类锁
        obj.nonStaticMethod();

        synchronized (obj) {
        // 获得了类锁和对象锁
        }
    }
    public synchronized void nonStaticMethod() {
        // 获得了对象锁
    }
}
```

一个非静态方法不通过调用同步静态方法是否也可以获取静态锁呢？换句话说，就是一个同步区域是不是可以使用类锁呢？例如：

```
public class ClassExample {
    synchronized void process() {
        synchronized (the class lock) {
        // 访问类的静态变量的代码
        }
    }
}
```

非静态方法试图去获取一个类锁的目的可能是为了防止在类数据(例如静态数据)上产生竞态条件。这可以通过调用该类的一个静态同步方法来实现。如果出于某种原因不愿意这样做，我们也可以通过一个公共的静态对象来使用同步块机制(使用一个静态的实例变量来存储这个公共对象可能是最好的方法)。例如，我们可以使用存储在某个类的所有对象都可以访问的公共地方的对象。

```
public class ClassExample {
    private static Object lockObject = new Object();
    synchronized void process() {
        synchronized (lockObject) {
            // 访问类的静态变量的代码
        }
    }
}
```

类锁和类对象

在该例中，我们使用Class对象的对象锁作为该类的通用锁。我们使用这个对象是因为类对象和系统中的类有一对一的关系。我们还提到当一个同步静态方法被调用时，系统会获取类锁。

已经证明了其实系统中并不存在什么类锁。当一个同步静态方法被调用时，系统其实是获取代表该类的类对象的对象锁。这就意味着类锁其实就是对应的类对象的对象锁。因此同时在程序中使用静态同步方法和使用类对象锁的同步块会造成混乱。

最后，如果不想创建一个新的对象，我们也可以获取表示类本身的类对象（也就是java.lang.Class类的实例）。该类的对象是用来在系统中表示类的。对我们

来说，使用这个类是因为系统中的每一个类都在类Class中有一个表示自己的对象。通过以下的方法可以获取该对象。

```
public class ClassExample {
    synchronized void process() {
        synchronized (Class.forName("ClassExample")) {
            // 访问类的静态变量的代码
        }
    }
}
```

调用Class类的ForName()方法会返回这个对象。我们可以使用这个对象来作为同步块的对象锁。

总结

在本章中，我们介绍了Java语言中的synchronized关键字。该关键字允许我们同步方法和代码块。

我们也开发了自己的同步原语BusyFlag。它可以帮助我们在多个方法之间对某个对象加锁，也可以根据需要来获取和释放锁。这种特性是Java的synchronized关键字所不能提供的，但在很多情况下很有用。

以上是我们对于同步的初步了解。正如我们以后要了解的，同步是多线程编程的一个重要部分。没有这些技术，我们就没有办法在不同的线程间正确地共享数据。尽管这些技术对于我们将要写的大部分程序而言已足够好，但在下一章中，我们还会介绍其他的技术。

第四章

等待和通知

本章内容：

- 返回到银行的例子
- 等待和通知
- wait()、notify()和notifyAll()
- wait()和sleep()
- 线程中断
- 静态方法（有关同步的细节）
- 总结

在上一章中，我们已经初步了解了同步的问题。使用介绍的同步工具，我们可以在线程之间进行互操作并且安全地共享数据而不会产生竞态条件。但是，正如我们将要看到的，同步不仅仅可以避免竞态条件，它还包括了基于线程的通知系统。而这正是我们这一章要讨论的。

返回到银行的例子

我们已经完成了ATM系统的全部代码，并且通过第三章学到的技术消除了系统中可能存在的问题。系统更加健壮了，原来经常出现的小问题再也没有出现。更重要的是，BusyFlag类使我们可以很快地完成一些行长所要求的修改。在这种情况下，BusyFlag类作为标准技术被采纳，并且在整个ATM系统中被广泛使用。

直到我们的老板意识到另外一个问题之前，我们一直都是英雄：ATM系统的一部分被发现有性能问题。这些地方是由其他人开发的，他们广泛地使用BusyFlag类。因为是我们开发的BusyFlag，所以我们必须解决这个问题。因此我们开始重新审视BusyFlag类：

```
public class BusyFlag {
    protected Thread busyflag = null;
    protected int busycount = 0;
```

```
public void getBusyFlag() {
    while (tryGetBusyFlag() == false) {
        try {
            Thread.sleep(100);
        } catch (Exception e) {}
    }
}

public synchronized boolean tryGetBusyFlag() {
    if (busyflag == null) {
        busyflag = Thread.currentThread();
        busycount = 1;
        return true;
    }
    if (busyflag == Thread.currentThread()) {
        busycount++;
        return true;
    }
    return false;
}

public synchronized void freeBusyFlag () {
    if (getBusyFlagOwner() == Thread.currentThread()) {
        busycount--;
        if (busycount == 0)
            busyflag = null;
    }
}

public synchronized Thread getBusyFlagOwner() {
    return busyflag;
}
}
```

在BusyFlag类中，我们注意到对sleep()方法的调用。最开始使用这个方法是为了避免浪费CPU周期。但是现在，这是一个值得讨论的问题了。如果睡眠时间过长，就会造成等待的时间太长，从而引起性能下降。反过来说，如果不使用sleep()方法，则会因为不停地轮询而使用过多的CPU周期从而同样引起性能下降。因此无论是哪种情况，都必须解决这个问题：最好有一个能够只等待到锁被释放为止的方法。我们期望只要锁被释放，getBusyFlag()方法就能够立刻获取busyflag，同时也不会在循环中消耗任何的CPU周期。这也正是我们在下一节中要讨论的问题。

等待和通知

如同任何对象都有一个可以被获取和释放的锁一样，每一个对象都提供一个等待该对象的区域。如同锁机制一样，这种机制的目的是帮助线程之间的通信（注1）。其实隐藏在这种机制后面的想法是很简单的：某个线程期望某种条件成立，并且假设另外一个线程会创造出这种条件。当其他线程创造出该条件后，就通知等待该条件的线程。这些是通过以下方法实现的：

void wait()

等待某种条件的发生。这是Object类的方法，而且必须在被同步的方法或者代码块中被调用。

void notify()

通知线程其等待的条件已经发生了。这是Object类的方法，而且必须在被同步的方法或者代码块中被调用。

wait()、notify()和Object类

有趣的是，和同步方法一样，等待和通知机制存在于Java系统中每一个对象里。但是等待和通知机制要通过方法调用才能实现，而同步机制是通过增加关键字来实现的。

wait()/notify()机制能够工作是因为它们是Object类的方法。每一个Java对象都是直接或者间接地从Object类继承来的，所以它们也都是Object对象，因此也就支持这种机制。

等待和通知机制的目的是什么？它们是如何工作的？等待和通知机制也是同步机制；但它不仅仅是一个通信机制：它允许一个线程通知另外一个线程某种特定的条件发生了。等待和通知机制并不指定该条件到底是什么。

注1：对于Solaris或POSIX线程，这被称为条件变量（condition variable）；而在Windows 95/NT中，它们被称为事件变量（event variable）。

等待和通知机制能不能用来代替同步机制？实际上，答案是否定的。等待和通知机制不能解决同步机制可以解决的竞态条件问题。事实上，等待和通知机制必须和同步锁机制配合使用才能够避免在等待和通知机制本身中可能出现的竞态条件。

现在就用这个技术来解决 BusyFlag 类中的等待时间问题。在原来的设计中，getBusyFlag()方法要调用tryGetBusyFlag()来获取busyflag。如果未能获得该标志，则睡眠100毫秒后再试。但是其实我们真正要做的是等待一个条件的发生（busyflag为闲）。因此我们可以使用这样一种机制：如果条件（busyflag为闲）不满足，则等待该条件的发生，并且在该标志释放时通知等待该条件的线程。这使得我们可以提供最终的、优化后的 BusyFlag 类。

```
public class BusyFlag {
    protected Thread busyflag = null;
    protected int busycount = 0;

    public synchronized void getBusyFlag() {
        while (tryGetBusyFlag() == false) {
            try {
                wait();
            } catch (Exception e) {}
        }
    }

    public synchronized boolean tryGetBusyFlag() {
        if (busyflag == null) {
            busyflag = Thread.currentThread();
            busycount = 1;
            return true;
        }
        if (busyflag == Thread.currentThread()) {
            busycount++;
            return true;
        }
        return false;
    }

    public synchronized void freeBusyFlag() {
        if (getBusyFlagOwner() == Thread.currentThread()) {
            busycount--;
            if (busycount == 0) {
                busyflag = null;
                notify();
```

```
                }
            }
        }

        public synchronized Thread getBusyFlagOwner() {
            return busyflag;
        }
    }
```

在新版本的getBusyFlag()方法中，对wait()方法的调用取代了100毫秒的睡眠。这用来等待要求的条件发生。而freeBusyFlag()方法也包含了对于notify()方法的调用。这用来发送被要求的条件已经发生了的通知。实践证明，新的实现比原来的好得多。我们现在只是等待busyflag进入闲状态（等待的时间不会多也不会少），同时也不用浪费CPU周期每隔100毫秒唤醒一次来检查busyflag是不是被释放了。

等待、通知和同步机制

前面已经提到，等待和通知机制需要用同步锁机制来解决其中包含的竞态条件。不幸的是，如果在等待和通知机制中不结合使用同步锁机制的话，就不可能解决竞态条件问题。这就是为什么waint()方法和notify()方法都必须拥有一个它们进行等待/通知的对象的对象锁的原因。

wait()方法在进入等待前会释放锁，但在从wait()方法返回前会再次获取该锁。这样做是为了消除竞态条件。你可能记得在Java API中没有关于释放和重新获得锁的概念。实际上，wait()方法使用了Java的同步机制中所不直接提供的特性，因此wait()方法与同步机制是紧密结合的。换句话说，wait()方法是一个固有的方法，是不可能用纯Java来实现的。

将等待和通知机制和同步方法结合使用的方法实际上是一种标准用法。在其他的系统上（如Solaris或者POSIX线程）中，条件变量也要求拥有一个互斥锁才能工作。

程序中还有其他一些改变：getBusyFlag()方法被同步了。在前面的例子中，如果将getBusyFlag()定义为同步，就会导致锁的作用域过大。但是wait()方法现在的工作方式使我们不必担心死锁了。wait()方法会释放锁，从而使得其他线

程可以运行 freeBusyFlag()方法。而当 wait()方法返回时，它会重新获取锁。这使得从程序员的角度来看是一直拥有锁的。

如果调用 notify()方法时没有线程在等待该条件，会有什么结果呢？这也不会有什么错误。正如在 BusyFlag 类中，如果在释放 busyflag时没有其他线程在等待该标志，也是可行的。因为等待和通知机制在发送通知时并不知道该通知会发送到何处去，因此它们假设一个没有线程在等待的通知就是不用发送的通知。换句话说，如果调用 notify()时没有任何线程在等待该事件，notify()方法就简单地返回。

在等待和通知中存在的竞态条件到底是什么呢？简而言之，一个调用 wait()的线程是在确定了它所要求的条件不能满足后（典型的方法是检查变量值）才调用 wait()方法的。而当其他线程设定了条件后（通常是设置同一个变量的值），它调用 notify()方法。在以下情况下，会出现竞态条件：

1. 第一个线程检查条件，确定需要等待。
2. 第二个线程设定条件。
3. 第二个线程调用 notify()方法，这个调用因为没有等待的线程而立刻返回。
4. 第一个线程调用 wait()方法。

如何解决这个潜在的竞态条件呢？可以使用我们前面介绍的同步锁机制来解决它。在调用 wait()方法或 notify()方法前，我们必须获取 wait()方法或 notify()方法要使用的对象的对象锁。这个步骤是必需的，如果没有该锁，wait()方法和 notify()方法就不会正确地工作，并且会抛出异常。更进一步，wait()方法也会在进入等待前释放锁，而在返回时又会再次获取锁。程序员必须使用这个锁来保证检查条件和设置条件值的操作都是原子性操作。一般而言，这是通过将条件变量作为锁对象的一个实例变量来实现的。

在 wait()方法的释放和重新获取锁的过程中是否会产生竞态条件呢？ wait()和锁机制是紧密地结合在一起的，这使得只有当处于等待状态的线程接收到通知后才会真正释放对象锁。因此如果我们想自己去实现 wait()方法和 notify()方法，

不能说是不可能，但也是很困难的。对于我们来说，这些都是实现的细节。我们只需知道它们可以工作，而且工作的很好就可以了。系统会保证等待和通知机制不会产生任何竞态条件。

等待、通知和同步块

在到目前为止的等待和通知例子中，我们已经使用过了同步方法。但是没有理由说不能改为使用同步块。惟一的要求就是我们进行同步的对象必须是wait()方法notify()方法使用的同一个对象。如下例所示：

```
public class ExampleBlockLock {
    private StringBuffer sb = new StringBuffer();
    public void getLock() {
        doSomething(sb);
        synchronized (sb) {
            try {
                sb.wait();
            } catch (Exception e) {}
        }
    }
    public void freeLock() {
        doSomethingElse(sb);
        synchronized (sb) {
            sb.notify();
        }
    }
}
```

为什么getBusyFlag()方法要循环检测tryGetBusyFlag()方法是否返回false？当wait()方法返回时，不是要释放标志吗？答案是否定的。当wait()方法返回时，标志并不一定为闲。用来解决等待和通知机制竞态条件的内部方法仅仅可以防止通知不会丢失，但并不能解决下面的问题：

1. 第一个线程获取busyflag。
2. 第二个线程调用tryGetBusyFlag()，返回false。
3. 第二个线程调用wait()方法，该方法会释放同步锁。
4. 第一个线程进入freeBusyFlag()方法，获取锁。

5. 第一个线程调用 notify()方法。
6. 第三个线程试图调用 getBusyFlag()，阻塞并等待同步锁。
7. 第一个线程从 freeBusyFlag()方法中返回，释放同步锁。
8. 第三个线程获取同步锁并且进入 getBusyFlag()方法。因为 busyflag 为闲，它就获取了该标志，然后从 getBusyFlag()方法退出，释放同步锁。
9. 第二个线程接收到通知，从 wait()方法中返回，重新获取同步锁。
10. 第二个线程再次调用 tryGetBusyFlag()方法，确定该标志位还是忙，因此再次调用 wait()方法

如果在 getBusyFlag()方法中没有该循环:

```
public synchronized void getBusyFlag() {
    if (tryGetBusyFlag() == false) {
        try {
            wait();
            tryGetBusyFlag();
        } catch (Exception e) {}
    }
}
```

则在第十步中，尽管线程2没有通过 tryGetBusyFlag()方法获取 busyflag，它还是会从 getBusyFlag()中返回。所以当 wait()方法返回时，仅仅表示在过去的某个时候条件被满足了，并且另一个线程调用了 notify()方法；但是我们不能假设现在条件仍然是满足的，而不再次去检查条件。因此，还是需要将 wait()方法的调用放到一个循环中。

wait()、notify()和 notifyAll()

当有多个线程在等待某个通知时会有什么情况发生呢？当调用 notify()时，到底哪个线程会收到通知呢？答案是不确定的: Java规范并没有定义哪个线程会收到通知。到底哪个线程会收到通知依赖于Java虚拟机的实现、调度程序和程序运行时的计时等因素。就算是在同一个平台上，也没有办法来确定到底哪个线程会收到通知。

Object 类提供另外一个方法来帮助我们处理有多个线程等待同一个条件的情况。

void notifyAll()

当条件满足时，通知所有等待的线程。这是Object类的方法，而且必须在被同步的方法或者是代码块中调用。

当有多个线程在等待同一个对象时，Object类提供的notifyAll()方法可以帮助我们避免只有一个线程得到通知。该方法类似于notify()方法，但是它是通知所有在该对象上等待的线程而不是只随便通知一个。和notify()方法一样，notifyAll()方法并不让我们决定到底哪些线程应该得到通知：实际上所有线程都得到通知。正是因为所有的等待线程都得到通知，我们才有可能用自己的选择机制来决定哪个线程可以运行，而哪个（哪些）线程应该再次调用wait()方法。

notifyAll()方法真的同时唤醒所有的线程吗？

答案是即对又不对。所有的等待线程都会被唤醒，但是它们还需要重新获取对象锁。因此这些线程不可能并行运行：它们必须串行运行来依次获取对象锁。因此当某个线程调用了notifyAll()方法释放锁后，在同一时刻只有一个线程可以运行。

为什么我们期望唤醒所有的线程呢？有很多可能的原因，其中之一就是线程有可能在多于一个的条件上等待。因为我们不能控制到底是哪一个线程会获得通知，因此完全有可能被唤醒的线程是在完全不同的条件上等待的。通过唤醒全部的等待线程，我们就可以在程序中决定这些线程中到底哪一个应该继续运行下去。

另外一个原因是该通知可能会满足多个等待线程，请看下面的例子：

```
public class ResourceThrottle {
    private int resourcecount = 0;
    private int resourcemax = 1;

    public ResourceThrottle (int max) {
        resourcecount = 0;
        resourcemax = max;
    }
```

```
    public synchronized void getResource (int numberof) {
        while (true) {
            if ((resourcecount + numberof) <= resourcemax) {
                resourcecount += numberof;
                break;
            }
            try {
                wait();
            } catch (Exception e) {}
        }
    }

    public synchronized void freeResource (int numberof) {
        resourcecount -= numberof;
        notifyAll();
    }
}
```

我们定义了一个名为ResourceThrottle的新类。该类有两个方法: getResource()和freeResource()。这两个方法都有一个指示多少个资源要被获取/释放的参数。而资源的最大数是由ResourceThrottle类的构造函数定义的。这个类类似于我们的BusyFlag类、它的getResource()方法在期望获取的资源数量不满足时也会进入等待状态。而freeResource()方法也是通过调用notify()方法来通知等待的线程系统资源数量增加了。

而该类与BusyFlag类的不同之处就是调用notifyAll()方法而不是notify()方法。原因如下:

- 当一定数量的资源被释放时,完全有可能被系统唤醒的那个线程所要求的资源多于系统能够提供的资源。如果我们使用notify()方法,就可能唤醒一个不恰当的线程、导致系统中其他可能满足条件的等待线程不被唤醒。
- 当资源被释放时、也许有多于一个的等待线程会满足条件。例如,如果我们释放十个资源、就可以满足分别是在等待三个、四个、一个和两个资源的四个线程。释放的资源和等待线程期望获取的资源间不存在一一对应关系。

通过唤醒所有的线程、就可以用很少量的工作解决上面两个问题。但是,现在我们仅仅是模拟了有目标性的通知的系统。我们并不能真正控制到底哪一个线程会被唤醒;实际上,我们是在将所有线程都唤醒后才来决定到底哪个线程能够继续

运行的。如果有大量线程处于等待状态，那么可能许多线程在被唤醒后发现它们期望的条件并没有满足，而只好再次进入等待状态，因此导致系统低效。

如果我们的确想控制到底哪一个线程能够接收到通知，就要实现一个对象数组。数组中保存的是每个线程等待的条件，而该数组就作为通知到达的目标。这意味着每个线程将等待在数组中的不同对象上。通过在调用notify()的线程中决定到底哪些线程应该收到通知，我们就避免了唤醒大量的线程然后再使其进入等待状态的代价。而使用这种对象数组方法的不利之处在于我们不得不在不同的对象上加锁。而使用多个锁很容易使人困惑甚至导致死锁。这种方法的实现相对比较复杂；我们也许不得不编写一个新的类来实现这个功能：

```
public class TargetNotify {
    private Object Targets[] = null;

    public TargetNotify (int numberOfTargets) {
        Targets = new Object[numberOfTargets];
        for (int i = 0; i < numberOfTargets; i++) {
            Targets[i] = new Object();
        }
    }

    public void wait (int targetNumber) {
        synchronized (Targets[targetNumber]) {
            try {
                Targets[targetNumber].wait();
            } catch (Exception e) {}
        }
    }

    public void notify (int targetNumber) {
        synchronized (Targets[targetNumber]) {
            Targets[targetNumber].notify();
        }
    }
}
```

该实现的概念是很简单的：在TargetNotify类中，对象数组的目的就是为了使用等待和通知机制。我们不是让所有线程都等待在this对象上，而是选择一个对象来等待（这可能有一点使人胡涂：因为我们提供的wait()方法有不同的参数，所以并没有覆盖Object类的wait()方法）。然后，当我们决定到底哪一个线程可以被唤醒后，就可以有针对性地通知该线程了。

对于这种有针对性的通知方式而言，程序员可以在效率高的优点和其复杂性之间做出选择。换句话说，每种技术都有缺点，因此是由开发者来权衡到底哪种机制是最合适的。

wait()和 sleep()

Object类还重载了wait()方法，可以在等待一定毫秒后进入超时状态（当然，正如我们在第二章了解到的，真正的超时精度可能达不到一毫秒的级别）。

void wait(long timeout)

等待一个条件的发生。但是，如果在设定的时间内没有收到通知，就返回。这是 Object 类的方法，而且必须是在被同步的方法或者代码块中调用。

void wait(long timeout, int nanos)

等待一个条件的发生。但是，如果在设定的毫秒和纳秒内没有收到通知，就返回。这是Object类的方法，而且必须是在被同步的方法或者代码块中调用。

这些方法是用来支持外部事件的。当我们仅仅关注通知是否到达时，通常不使用这些方法。但是，有时候通知是依赖于外部条件的，在这种情况下，我们还要关心通知何时到达。因此，当条件可能不发生时，超时机制就很有用了。假如我们有一个与某个股票行情服务器连接的程序。它期望在30秒内能够连接服务器（就是满足连接这个条件）；如果在30秒内连接还没有成功，程序也许就会试图去连接一个后备服务器。因此我们可以在程序中使用wait(30000)来实现这一点。

当我们知道条件最终会被满足时，也可以使用超时机制，使得还可以同时完成其他的任务。例如，假设我们在getBusyFlag()方法中要处理其他一些任务。

```
public synchronized void getBusyFlag() {
    while (tryGetBusyFlag() == false) {
        wait(100);
        doSomethingElse();
    }
}
```

在该例中，我们用100毫秒来等待通知。如果在100毫秒内通知没有到达，则被唤醒。当然这个例子显得有些做作，因为我们可以很容易地通过创建另外一个线程来处理其他的任务实现这一点。

如果我们知道通知不会到来，那么wait(long)和sleep(long)之间有什么区别呢？

假设不对一个对象调用notify()方法，那么在理论上就不会对该对象调用wait()方法。但是，wait(long)方法有着如下的优点：除了其行为和Thread类的sleep(long)一样外，它还能释放然后再次获取锁。这就意味着即使不使用等待和通知机制，仍然可以使用wait(long)方法来达到使得线程睡眠但是又不保持锁的特性。例如，假如有如下的类：

```
public class WaitExample {
    public synchronized void ProcessLoop() {
        processOne();
        try {
            wait(1000);
        } catch (Exception e) {}
        processTwo();
    }
}
```

WaitExample是一个需要在两个不同的操作间睡眠一秒钟的类，在这段时间内，它必须释放其锁。如果我们不使用wait(long)方法，代码就会增加额外的复杂性：

```
public class WaitExample {
    public void ProcessLoop() {
        synchronized (this) {
            processOne();
        }
        try {
            Thread.sleep(1000);
        } catch (Exception e) {}
        synchronized (this) {
            processTwo();
        }
    }
}
```

正如我们说过的，这仅仅是一个简单的例子：对于如下的类，如果不使用wait(long)方法，会有什么结果呢：

```
public class WaitExample {
    public synchronized void ProcessLoop() {
        processOne();
        for (int i=0; i<50; i++) {
            processTwo();
            try {
                wait(1000);
            } catch (Exception e) {}
        }
    }
}
```

线程中断

和我们在第二章中讨论的sleep()和join()方法一样，wait()方法在某种情况下会抛出InterruptedException异常。当其他线程调用interrupt()方法中断了线程的运行时，这些方法都会抛出这个异常。

void interrup()(java 1.1及以上版本)

向一个指定的线程发出中断信号。如果目标线程是阻塞在与线程相关的方法（如sleep()、join()或者wait()）中，则目标线程会转到一个非阻塞状态。否则会设置一个布尔标志指明该线程已经被中断。

Java 1.0中的线程中断

如果你正在使用Java 1.0，你会发现这一节中讨论的与中断相关的方法都不起作用：它们只会抛出NoSuchMethodError异常。

在Java 1.0.2以及Netscape 3.0和IE 3.0之类的游览器（它们是基于Java 1.0.2的）中，这些与线程相关的方法也不能正确运行。特别是，interrupt()方法不能够中断一个处于睡眠状态的线程。虽然可以使用isInterrupted()方法来判断interrupt()方法是否被调用过，但是这些方法的Java 1.0.2实现不能处理那些被阻塞的线程。

在Java 1.1及以上版本中，线程中断才能很好地工作，但是在许多基于1.1的浏览器中，这些方法还是有问题。所以我们建议目前只在Java应用程序和在Java插件上运行的applet中使用这些方法。

interrupt()方法的作用依赖于那些被中断的线程上正在运行的方法是否会根据该中断信号抛出相应的异常。当线程在执行sleep()、wait()和join()方法时，这些方法会抛出InterruptedException异常。否则，会设置一个标志，线程可以在以后通过检查该标志来确定interrupt()方法是否被调用过（注2）。

interrupt()方法是Thread类的一个方法，是一个线程用来向另外一个线程发信号的：它也可以（尽管没有什么意义）中断自己。如果目标线程运行的方法抛出InterruptedException异常，它就知道自己被中断了。否则，线程必须使用如下的方法来检查自己是否已被中断：

static boolean interrupted()（java 1.1及以上版本）

返回一个指示当前线程是否已被中断的布尔值。这个方法是Thread类的一个静态方法，因此可以通过类说明符来调用。该方法简单地返回被interrupt()方法设置的标志的值。

boolean isInterrupted()（Java 1.1及以上版本）

返回一个表示指定的线程是否已被中断的布尔值。该方法简单地返回interrupt()方法设置的标志的值。

这两个方法的惟一区别在于interrupted()方法是静态的，而且是作用于当前线程上，而isInterrupted()方法是动态的，并且必须在一个线程对象上执行。interrupted()方法和下面的代码等价：

```
Thread.currentThread().is Interrupted();
```

这两个方法后面隐藏的是：interrupt()方法的内部实现会在目标线程对象的某处设置一个标志，用来表示interrupt()方法已经被调用过了，而interrupted()和isInterrupted()方法只是简单地返回该标志。

我们可以使用interrupt()方法来使一个处于等待状态的线程得到一个不同的输出。等待和通知机制常常用在生产者/消费者情况中：一个或者多个线程负责生

注2：在一些虚拟机中，还有另外一种可能性，稍后我们后检查这种可能性。

产数据（例如，从某些服务器读取数据），而另外一些线程负责消耗数据（例如解析数据）。当没有数据可以消耗时，消费者线程就会进入等待之中。

```
import java.util.*;

public class Consumer extends Thread {
    Vector data;
    public Consumer(Vector data) {
        this.data = data;
    }
    public void run() {
          Object o;
          while (true) {
                synchronized(data) {
                      if (isInterrupted())
                           return;
                      while (data.size() == 0) {
                            try {
                                 data.wait();
                            } catch (InterruptedException ie) {
                                  return;
                            }
                      }
                      o = data.elementAt(0);
                      data.removeElementAt(0);
                }
                process(o);
          }
    }
}
```

现在不是通过在其run()方法中设置一个标志来停止一个消费者线程，而是通过另外一个线程来中断消费者线程。这样就会得到两个可能的结果：如果消费者线程正在执行wait()方法，则在抛出异常的同时run()方法也返回。但是如果消费者线程正在执行process()方法，则process()方法会执行完毕，而消费者线程会在下一次检查中断标志时从run()中退出。

注意，如果线程在执行sleep、wait()和join()方法时被中断，Java虚拟机的内部实现不会设置中断标志。如下面的代码：

```
boolean done = false;
synchronized (lock) {
```

```
        while (!done) {
            try {
                lock.wait();
            } catch (InterruptedException ie) {
                done = isInterrupted();
            }
        }
    }
```

在catch子句中，变量done会被设置为false，从而保证循环不会停止。

在某些情况下，直接在目标线程中设置一个布尔标志来停止目标线程比发送中断信号更有用。但是，interrupt()方法可以中断sleep()和wait()方法，同时它对我们将在第十章学习的某些线程管理技术也是很有帮助的。

可中断的I/O

interrupt()方法令人疑惑的地方在于I/O方面：interrupt()方法能否影响那些阻塞在I/O操作上的线程？就目前来说答案是否定的，而且你也不能依赖于这种中断的能力。当然，在以后的虚拟机实现中，情况可能会改变。

但是，我们已经知道，有一些Java虚拟机实现（注3）（特别是Solaris本地线程实现）允许interrupt()方法中断任何挂起的I/O操作。因此，如果一个阻塞在read()方法上的线程被interrupt()方法中断了，则read()方法会抛出IOException异常；而具体是什么IOException则不确定：在Java 2中，是InterruptedIOException，而在1.1中，会抛出其他异常（例如SocketException）。在以后的版本中这也会有所改变：以后Solaris本地线程实现也许不允许I/O被中断。在一些使用绿色线程的虚拟机实现中，有些I/O方法会抛出InterruptedIOException异常，而有些I/O方法不会。因为Windows平台上的I/O操作是不能被中断的，所以Windows平台上的虚拟机不支持这一点。

那么，程序员应该如何去做呢？安全的方法是不要用interrupt()方法使得一个等待I/O完成的线程从阻塞状态中退出：如果你真的需要这样做，应当关闭这个

注3：这些虚拟机的不同实现将在第六章中讨论。

线程阻塞在其上的输入/输出流。即使可中断I/O操作以后被Java虚拟机所采纳，它也会使用一个不同的接口。如果你确实要依赖于可中断I/O，则应该明白被中断的I/O因为不能确定I/O的状态，也不知道从何处再启动该I/O，因此该I/O是不可再启动的。正是因为重新启动一个被中断的I/O是很困难的，所以不同虚拟机的实现是不一致的。

如果我们想不管线程是否是在I/O操作上阻塞，都使用interrupt()方法来停止它，该怎么做呢？在上一个例子中，我们利用wait()方法可以抛出异常的特性来得知没有数据了。如果我们想在I/O操作上也这么做，可以编辑如下：

```
import java.util.*;
import java.io.*;
import java.net.*;

class StockObservable extends Observable {
    String lastTick;

    void setTick(String s) {
        lastTick = s;
        setChanged();
        notifyObservers();
    }
}

public class StockHandler extends Thread {
    private BufferedReader br;
    private InputStream is;
    private Socket sock;
    private StockObservable stock;
    private volatile boolean done = false;
    private Object lock = new Object();

    class StockHandlerThread extends Thread {
        public void run() {
            String s;
            try {
                while ((s = br.readLine()) != null)
                    stock.setTick(s);
            } catch (IOException ioe) {}
            done = true;
            synchronized(lock) {
                lock.notify();
            }
```

```
        }
    }

    public StockHandler(StockObservable o, String host, int port)
                    throws IOException, UnknownHostException {
        sock = new Socket(host, port);
        is = sock.getInputStream();
        stock = o;
    }

    public void run() {
        br = new BufferedReader(new InputStreamReader(is));
        Thread t = new StockHandlerThread();
        t.start();
        synchronized(lock) {
            while (!done) {
                try {
                    lock.wait(Integer.MAX_VALUE);
                } catch (InterruptedException ie) {
                    done = true;
                    try {
                        t.interrupt();
                        is.close();
                        sock.close();
                    } catch (IOException ioe) {}
                }
            }
        }
    }
```

我们原来经常提到，在Java中经常通过一个独立的线程来处理I/O。这就是该技术的一个例子。该类建立了一个到股票服务器的套接字，从服务器中读取数据，然后通过给定对象发布数据。此时的read()方法经常会被阻塞。而当我们想停止该线程时，就不得不通过某些方法来终止read()方法。这可以通过关闭read()方法正在读取的套接字来实现。

但是，该例子中所做的是启动两个线程：一个读取数据，另外一个等待中断信号的到来（因为设置的超时时间是如此之长）。当第二个线程被中断后，它就关闭第一个线程正在其上阻塞的输入流，这样两个线程就都可以退出了。这允许我们通过向第二个线程发送中断信号来关闭全部线程（以及与它们相关联的套接字）。

```
Thread t = new StockHandler(...);
... Do other stuff until we need to shut down the handler ...
t.interrupt();
```

当然，我们也可以简单地暴露该线程的套接字和输入流实例变量，这样任何一个线程都可以直接关闭它们。因为最好是将属于一个类的信息都包装到一起，所以我们很少这样做。同样，我们也可以提供一个类似于 shutdown()这样的方法来关闭套接字和输入流。这样做可以少用一个线程：StockHandler 类在 run()方法中读取数据，而外部线程可以执行 shutdown()方法。

你可能赞同或者反对将这些方法加到StockHandler的接口中去。我们在第十章中会再次讨论这个问题。

最后，要注意在通过关闭输入流来使得处理股票信息的线程 t 不再阻塞前，也对 t 调用了 interrupt()方法。这是因为系统中存在一个 bug：在使用 Java 虚拟机的本地线程实现的Solaris 2.6及以前版本中，该例中的 close()方法会阻塞直到在另外的线程中执行的read()方法不再阻塞为止。虽然这个问题在solaris 2.7中已经解决了，但是为了保证该例在早期的Solaris中也可以正常运行，我们还是加上了一个对 interrupt()的调用。而且这种做法在任何 Solaris 版本上、使用绿色线程实现的虚拟机上以及 Windows 平台上都不会带来副作用。更通用的说法是：因为该处理股票信息的线程可能调用wait()、sleep()方法，所以在这些情况下向它发出中断信号也是有必要的。

静态方法（有关同步的细节）

在一个静态方法中调用 wait()方法和 notify()方法会有什么结果？ wait()方法和 notify()方法都是 Object 类的非静态方法。因为静态方法在没有一个对象引用的情况下不能调用非静态方法，因此静态方法不能直接调用 wait()方法和 notify()方法。但这不能阻止我们实例化一个Object对象来作为等待的对象。这和我们曾经用过的在静态方法中获取对象锁的技术一样。

在静态方法和非静态方法中同时使用一个实际对象来调用 wait()方法和 notify()方法可以使它们能够互相通信，这和在静态和非静态方法中在一个公共

对象上使用同步块机制是一样的道理。下面的staticWait()方法和static-Notify()方法可以从静态或者是非静态方法中调用:

```
public class MyStaticClass {
    static private Object obj = new Object();

    public static void staticWait() {
        synchronized (obj) {
            try {
                obj.wait();
            } catch (Exception e) {}
        }
    }

    public static void staticNotify() {
        synchronized (obj) {
            obj.notify();
        }
    }
}
```

运行静态方法的线程很少会和运行非静态方法的线程这样通信。不过，通过这两个静态版本的wait()方法和notify()方法，我们使得这种通信成为可能。因为这两个方法的签名与wait()方法和notify()方法的签名一样，因此我们使用了不同的名字。

总结

本章介绍了以下方法:

void wait()

等待某种条件的发生。这是Object类的方法，而且必须是在被同步的方法或者代码块中调用。

void wait(long timeout)

等待一个条件的发生。但是，如果在设定的时间内没有收到通知，就返回。这是Object类的方法，而且必须是在被同步的方法或者代码块中调用。

void wait(long timeout, int nanos)

等待一个条件的发生。但是，如果在设定的毫秒和纳秒内没有收到通知，就返回。这是Object类的方法，而且必须是在被同步的方法或者代码块中调用。

void notify()

通知线程其等待条件已经发生了。这是Object类的方法，而且必须是在被同步的方法或者代码块中调用。

void notifyAll()

当条件满足时，通知所有等待的线程。这是Object类的方法，而且必须是在被同步的方法或者代码块中调用。

void interrupt()(Java 1.1 及以上版本)

向一个特定的线程发出中断信号。如果目标线程是阻塞在与线程相关的方法（如sleep()、join()或者是wait()）中，则目标线程会转到一个非阻塞状态。否则会设置一个标志来指明该线程已经被中断。

static boolean interrupted() (Java 1.1 及以上版本)

返回一个指示当前线程是否已被中断的布尔值。这个方法是Thread类的一个静态方法，因此可以通过类说明符来调用。该方法简单地返回被interrupt()方法设置的标志的值。

boolean isInterrupted() (Java 1.1 及以上版本)

返回一个指示指定的线程是否已被中断的布尔值。该方法简单地返回interrupt()方法设置的标志的值。

使用这些方法，我们就能够在线程间进行有效率的通信了。除了可以防止竞态条件外，这些机制还使得线程不必求助于轮询或者超时就可以在线程间通过事件或者条件来交互。虽然等待和通知机制是本章最广泛使用的技术，但我们还是可以中断一个线程，而不用考虑它正在做什么。

Java中对数据和线程进行同步的两个默认技术——synchronized关键字以及等待和通知方法，使得线程可以使用一种简单、健壮的方法进行通信和合作。尽管在第八章中我们将学习到一些数据同步的高级技术，但这些基本的技术对于大多数Java程序而言已经足够了。

第五章

Java线程编程的例子

本章内容：

- 数据结构和容器
- 简单的同步例子
- 一个网络服务器类
- AsyncInputStream 类
- 使用 TCPServer 和 AsyncInputStream
- 总结

在上面的几章中，我们学习了一些用来在线程间进行数据同步的工具。通过使用这些工具，线程之间就可以以一种安全的方式来使用、修改共享的数据而不发生竞态条件，因此我们的线程就可以和其他线程、系统线程以及标准Java库启动的线程来共同合作了。这种能够安全处理数据的能力就使得线程间能够交换数据，从而使得我们可以安全地使用多个线程来完成任务，最终实现我们的目标。

换句话说，我们现在可以认为线程编程仅仅是编程中的细节问题。在理想状态下，线程可以被看成是完成某项工作的对象。虽然线程本身是一个强有力的编程工具，但是最终我们想要的也就是用它来播放音乐或者是通过套接字来读取数据。

在本章中，我们要看看如何使用线程。我们将要看到对于特定的问题，如何使用线程来解决它，同时要讨论解决方案、线程本身以及对线程进行控制和同步的机制的实现细节。我们从使用线程来解决问题而不是学习线程系统特性的角度来学习线程。

数据结构和容器

有趣的是，我们最初的几个例子并不需要创建线程。第一个主题是可以在线程间使用和传递的数据类型。在创建一个数据对象时，并不知道到底有多少线程会使

用它：尽管这些对象可能会被多个线程访问，但也可能只被一个线程访问（在这种情况下，就不需要同步对该对象的访问）。因此，我们先来看看用来在进程间传递数据的操作系统机制。

在Unix系统中，最初的IPC（interprocess communication，进程间通信）技术是消息队列、信号量和共享内存。尽管Unix已经增加了许多其他的机制，但这三种机制还是最为常用而且也在很多应用程序中大量使用。Java中的IPC机制（同步锁以及等待和通知机制）是同步机制，它们不同于消息队列和共享内存，在线程间不进行真正的数据传递。Java中的IPC关心的是同步，而不是通信（注1）。这样做是因为如果有了线程间的同步工具，实现线程间通信也就容易了。现在，让我们看看消息队列和共享内存技术对于线程间通信是否有用。

消息队列

我们从消息队列开始：

```
import java.util.*;

public class MsgQueue {
    Vector queue = new Vector();
    public synchronized void send(Object obj) {
        queue.addElement(obj);
    }

    public synchronized Object recv() {
        if (queue.size() == 0) return null;

        Object obj = queue.firstElement();
        queue.removeElementAt(0);
        return obj;
    }
}
```

注1：这可以应用于大多数线程系统。在Solaris或POSIX线程中，主要工具是互斥锁、读－写锁、信号量和条件变量，它们都不传送实际数据。

如果有了恰当的同步工具，消息队列的实现就会很简单。在多任务情况下，操作系统不得不通过一个队列将数据从一个应用程序传送到另一个。当然，操作系统要同步这些通信过程。因为线程是共享同一个数据对象，因此数据传递是通过使用对公共数据对象的引用来实现的。一旦我们能够实现对这些数据对象的同步访问，通过消息队列来发送和接收这些数据就不是困难的事。在我们消息队列实现的中，是通过使用Java系统提供的Vector（矢量）类来实现队列的。我们只需要保证对于该队列增减元素的动作都是安全的；而这可以通过保证对于队列的访问是同步的来实现。这个实现是如此的简单，以至于我们甚至都不需要实现一个MsgQueue类。实际上，可以通过将对Vector类的操作（增加/删除一个消息）放在一个同步块中来直接达到这一目的。换句话说，消息队列和任何一个容器类（也就是能够存放一系列对象的类）一样容易实现。

共享内存

一个共享内存的实现如下：

```
public class ShareMemory extends BusyFlag {
    byte memory[];
    public ShareMemory (int size) {
        memory = new byte[size];
    }

    public synchronized byte[] attach() {
        getBusyFlag();
        return memory;
    }

    public synchronized void detach() {
        freeBusyFlag();
    }
}
```

和MsgQueue类一样，ShareMemory类也很简单，而且也没有必要单独实现。直接对一个Byte数组进行同步操作就可以得到同样的效果，而不需要实现ShareMemory类。该类的惟一好处在于，因为它是BusyFlag的子类，所以我们可以在任何范围内（包括方法间）对该共享内存调用attach()和detach()方法。

尽管线程在很多方面类似于进程，但是我们还是要明白在进程间和线程间对于数据的处理是不同的。在进程间共享数据要求操作系统提供一些特别的方法。但是Java程序中的数据通常都是在线程间共享的。实际上，不需要特别的操作就可以共享数据，这个事实证明了上面所说的IPC技术实际上是不需要的。其实，我们应该时刻从线程的角度来思考。因此每当开发一个新类时，不管程序到底是不是包含多个线程，都应当从可能有多个线程来使用该类的角度进行考虑。

循环链表

到底我们要开发什么容器类呢？链表，B树，图，还是我们发明的其他数据结构？当我们实现这些类的其中之一时，是不是都要假定它们会被多个线程并行访问？答案取决于你个人的喜好或者是公司的策略。但是保证容器类是线程安全的并不困难。因此毫无疑问最好是使得它们是线程安全的，从而不用在以后某个时候还担心它会在多个线程间使用。现在先看一个我们应该都实现过的容器类 —— 循环链表：

```
class CircularListNode {
    Object o;
    CircularListNode next;
    CircularListNode prev;
}
```

像原来编写的其他链表一样，我们也是使用一个简单的数据结构来存储对象。实际上我们现在并不关心链表中所存放对象的类型，因此只需要一个Object类对象的引用。这使得我们可以在容器中存放任何类型的对象［对于简单类型（例如int），需要使用它们对应的包装类］。还有一个实现中的细节，那就是通过保存前一节点和后一节点的引用来提高搜索时的效率。

```
public class CircularList {
    private CircularListNode current;

    public synchronized void insert(Object o) {
        CircularListNode tn = new CircularListNode();
        tn.o = o;
        if (current == null) {
            tn.next = tn.prev = tn;
            current = tn;
```

```
        } else {                    // 在当前节点前添加
            tn.next = current;
            tn.prev = current.prev;
            current.prev.next = tn;
            current.prev = tn;
        }
    }

    public synchronized void delete(Object o) {
        CircularListNode p = find(o);
        CircularListNode next = p.next;
        CircularListNode prev = p.prev;
        if (p == p.next) {          // 列表中的最后一个对象
            current = null;
            return;
        }
        prev.next = next;
        next.prev = prev;
        if (current == p) current = next;
    }

    private CircularListNode find(Object o) {
        CircularListNode p = current;
        if (p == null)
            throw new IllegalArgumentException();
        do {
            if (p.o == o) return p;
            p = p.next;
        } while (p != current);
        throw new IllegalArgumentException();
    }

    public synchronized Object locate(Object o) {
        CircularListNode p = current;
        do {
            if (p.o.equals(o)) return p.o;
            p = p.next;
        } while (p != current);
        throw new IllegalArgumentException();
    }

    public synchronized Object getNext() {
        if (current == null)
            return null;
        current = current.next;
        return current.o;
    }
}
```

CircularList 类的实现与原来实现过的其他循环链表并没有什么两样。可以通过 insert()方法和delete()方法来对链表进行操作；如果该链表存放对象，我们就可以使用getNext()方法来遍历该链表或者使用locate()方法来搜索某个特定的对象。

如何保证多个线程也能安全地使用 CircularList 类呢？可以简单地将所有的方法都定义为同步方法。通过使用synchronized关键字，就可以保证CircularList类能够同时在多个线程间安全地使用。换句话说，只要几分钟的时间就可以使得该类是线程安全的，从而可以将它作为线程间通信的标准方法。只要经过足够的练习，我们就可以毫不费力地使用同步工具了。

我们注意到 find()方法没有被定义为同步方法：find()仅仅是一个被同步方法所调用的私有方法，因此尽管将它定义为同步方法没有副作用，但还是没有理由将它也定义为同步方法。还要注意的是find()和locate()方法之间的一些细微不同之处：作为一个内部方法，find()方法返回CircularList对象，而locate()方法返回的是插入到链表中的实际对象。

同步和效率

有人认为仅仅因为存在被多个线程使用的可能性就同步一个类会导致低效：同步要使用一些额外的时间来获取同步锁。这是在设计一个大型程序时，程序员必须意识到的一个折衷。

在本书中，我们认为解决性能问题比查找因为没有正确同步数据而产生的bug要容易得多。在大多数情况下，Java API 也是这么设计的：像 Hashtable 和 Vector 这样的类都是正确同步的，从而保证它们在多线程程序中也可以正确地运行。

在Java 2中，一些新的容器类更关注于效率而不是线程安全。这些类被称为集合（collection）类。它们都是java.util.Collection 类或者java.util.Map类的子类。例如，HashMap类有着和Hashtable类同样的语义，但HashMap类不是同步的。同样，ArrayList 类提供与 Vector 类同样的语义，等等。

在基于Java 2的程序中使用这些类，可以获得性能上的一点提高。但是这种用法是存在争议的。因为从1.1.6开始，Java虚拟机的参考实现中的同步代码就被全部重写了，从而极大地降低了获取一个同步锁的代价。因此，对大多数程序而言，只有频繁地使用上面提到的这些类时才会获得一些性能上的提高。但另一方面，如果在使用HashMap类时没有意识到有两个线程会同时使用它，而由此带来的竞态条件会使得你的程序每运行100天就会发生错误。那么，稍微运行得快一点的代码到底有什么好处呢？

在第八章中，我们将说明如何安全地使用这些类以及其他非线程安全类。

简单的同步例子

在这一节中，我们将看到两个使用同步机制来完成复杂同步任务的例子。

屏障

在所有类型的线程同步工具中，屏障（barrier）可能是最容易理解和最少使用的。在我们想到同步时，最开始想到的就是一群线程执行某个任务的一部分，然后在某一点处同步它们的结果。屏障就是这样一个简单的等待点：所有的线程在这一点同步，它们要么是合并各自的结果，要么是运行到整个任务的下一阶段。而到现在为止所讨论的同步机制都是关心更加复杂的方面，如防止竞态条件发生、处理线程间数据的传送和交互以及线程间的信号等等。

既然屏障如此简单，那么为什么到现在为止我们还没有提到过它？答案是：其实我们已经使用过这种技术；我们使用Thread类来同步各个线程。通过使用join()方法，我们等待全部线程结束从而可以处理结果或者为下一个任务启动新的线程。

但是join()方法存在一些问题。第一，不停地创建和停止线程会使线程丢失它们在上一个操作中存放的状态信息。第二，如果必须要创建新的线程，逻辑操作就不能放在一起：因为对于每一个新的子任务，都必须创建一个新的线程，因此每一个子任务的实现都必须放在单独的run()方法中。如果子任务都是很小的，则将所有这些代码放在一个方法中更容易。

```
public class Barrier {
    private int threads2Wait4;
    private InterruptedException iex;

    public Barrier (int nThreads) {
        threads2Wait4 = nThreads;
    }

    public synchronized int waitForRest()
        throws InterruptedException {
        int threadNum = --threads2Wait4;

        if (iex != null) throw iex;
        if (threads2Wait4 <= 0) {
            notifyAll();
            return threadNum;
        }
        while (threads2Wait4 > 0) {
            if (iex != null) throw iex;
            try {
                wait();
            } catch (InterruptedException ex) {
                iex = ex;
                notifyAll();
            }
            return threadNum;
        }

        public synchronized void freeAll() {
            iex = new InterruptedException("Barrier Released by freeAll");
            notifyAll();
        }
    }
```

使用前面章节中的技术来实现Barrier类是简单的。每一个到达屏障的线程都调用wait()方法，而当最后一个线程到达时，就要通知所有等待的线程。如果其中任何一个等待线程收到中断信号，则所有的线程都应当收到同样的中断信号。另一个方法freeAll()用来对所有的线程产生一个中断信号。为了方便起见，每一个线程都被分配了一个线程号。最后一个到达屏障的线程将被分配零。在屏障已经释放后，如果还有线程到来，就会被赋予一个负值。这表示出现了错误：如果对一个设计为同步四个线程的屏障发送了五个线程，则第五个线程将会收到负值。

上面屏障的实现也是一个单用户的实现。一旦屏障达到其构造函数设定的线程上限，或者出现错误，屏障就不会阻塞任何线程。在了解了这个简单的实现后，实现这个类的单一用途实例也就不是一件困难的事情了。

下面是使用 Barrier 类的例子：

```
public class ProcessIt implements Runnable {
    String is[];
    Barrier bpStart, bp1, bp2, bpEnd;

    public ProcessIt (String sources[]) {
        is = sources;
        bpStart = new Barrier(sources.length);
        bp1 = new Barrier(sources.length);
        bp2 = new Barrier(sources.length);
        bpEnd = new Barrier(sources.length);

        for (int i=0; i < is.length; i++) {
            (new Thread(this)).start();
        }
    }
public void run() {
    try {
        int i = bpStart.waitForRest();
        doPhaseOne(is[i]);
        bp1.waitForRest();
        doPhaseTwo(is[i]);
        bp2.waitForRest();
        doPhaseThree(is[i]);
        bpEnd.waitForRest();
    } catch (InterruptedException ex) {};
}

public void doPhaseOne(String ps) {
}

public void doPhaseTwo(String ps) {
}

public void doPhaseThree(String ps) {
}

static public void main(String args[]) {
    ProcessIt pi = new ProcessIt(args);
```

```
        }
    }
```

使用 Barrier 类并不表示不需要创建线程。在 ProcessIt 类中，还是要创建线程并且实现其 run() 方法；但是我们只需实现该方法一次。在同一个 run() 方法中已经包含了全部三个阶段的代码。线程在进入下一阶段前会等待其他线程同步。通过使用Barrier类，我们还可以允许所有线程在同一个时间启动并且被赋予一个线程号。

该例子的运行流程如图 5-1 所示，该图同时显示了使用 join() 方法来创建新线程时的执行流程。在使用屏障技术和创建新的线程之间有点微妙的不同。第一点是屏障技术不要求线程系统销毁和创建大量的线程，这是屏障技术的一个优势。第二点是使用屏障类时所有的线程都是活动的，这使得应用程序一直处于多线程状态（注 2）。使用 Barrier 类是一个一阶段过程，而使用 Thread 类是一个两阶段过程：先用 join() 方法来使其他线程进入单线程状态，然后再创建新的线程。

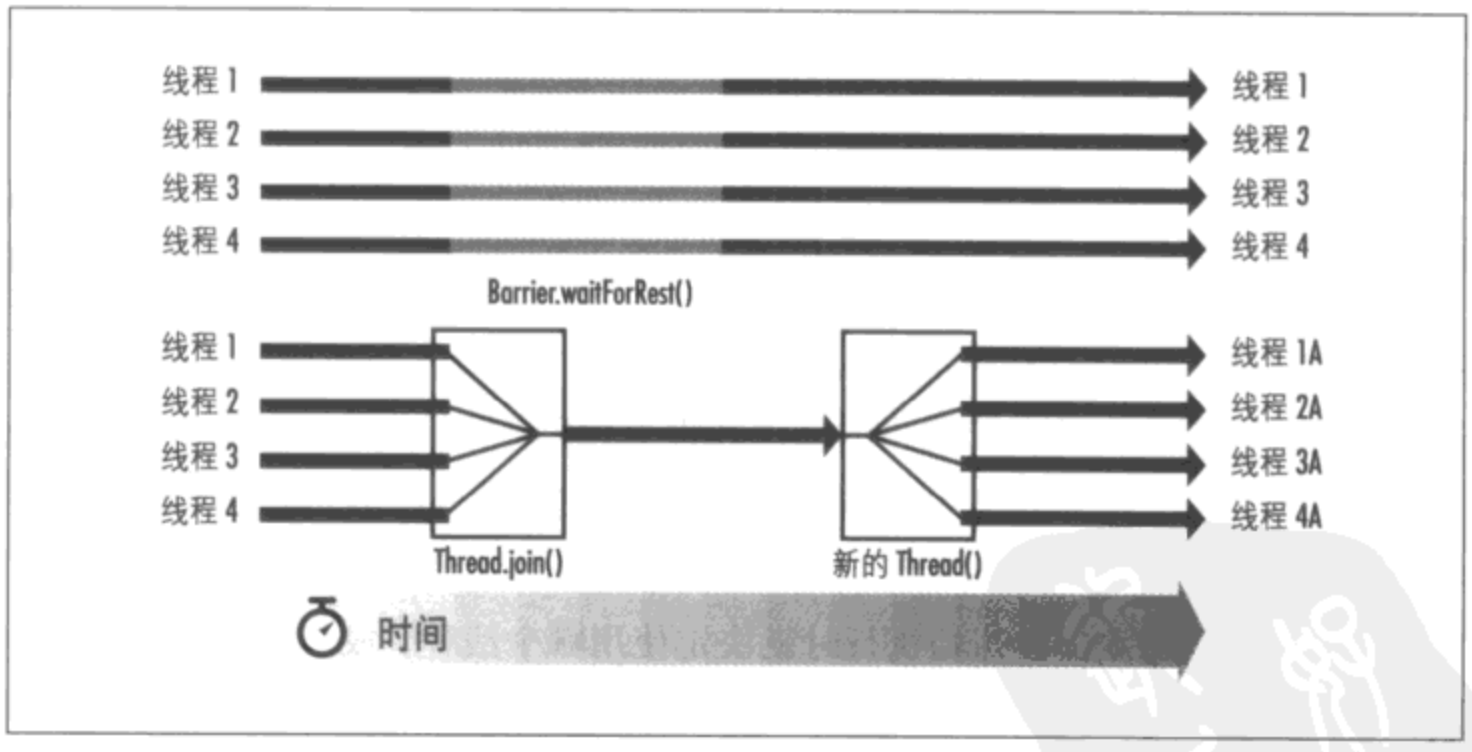

图 5-1：Barrier 类和连接线程的比较

这个两阶段过程允许在创建新线程以前执行其他任务，因为在阶段间只存在一个

注 2：这里存在一些争议，认为既然只有一个线程处于等待状态，那么系统实际上是单线程的。

线程。对于Barrier类而言，这是不可能的，因为对要被释放的线程的惟一要求就是线程计数。因此在某些复杂的情况下，如果要完成初始化和清理之类的任务，这就会成为一个问题。一个解决方法就是修改Barrier类，使得其在执行下一阶段任务前运行一些设置代码。但是，这使得Barrier类不能将全部代码放在一个方法中，而这正是Barrier的一个突出优点。我们将通过Barrier来保护不同阶段的设置代码，而不是将各个阶段的代码放到不同的run()方法中。下面是在不修改Barrier类的情况下解决这个问题的方法：

```
public class ProcessIt implements Runnable {
    public void run() {
        try {
            int i = bpStart.waitForRest();
            doPhaseOne(is[i]);
            if (bp1.waitForRest() == 0)
                doPhaseTwoSetup();
            bp1b.waitForRest();
            doPhaseTwo(is[i]);
            if (bp2.waitForRest() == 0)
                doPhaseThreeSetup();
            bp2b.waitForRest();
            doPhaseThree(is[i]);
            bpEnd.waitForRest();
        } catch (InterruptedException ex) {};
    }
}
```

在这个例子中，我们没有在不同的阶段间使用一个屏障，而是使用了两个屏障。这样是为了模拟在两阶段代码中的清理和设置代码。实际上，只有一个线程会执行这些代码，所以这部分代码就是在单线程情况下运行的。我们使用屏障返回的线程号来判断到底是哪个线程要执行这些代码。实际上，还可以用其他的方法来选择这个线程：例如，在开始执行时就决定使用哪个线程，或者就使用运行这些设置代码的线程。更进一步，因为我们现在可以运行清理和设置代码，所以没有必要在一开始的时候就声明所有的屏障。屏障的定义和清理都可以包含在设置代码中。

屏障技术对多阶段算法很有用。例如，编译器通常是由对代码的预处理、解析代码、将代码转换为内部格式以及优化代码等阶段组成的。每一个阶段都可以通过

多个线程来实现，而在阶段间，每一个线程都要等待其他线程完成它们在该阶段中的工作。

条件变量

条件变量是POSIX线程提供的一种同步类型。条件变量非常类似于Java中的等待和通知机制，实际上，它们的功能是一致的。条件变量的四个基本操作——wait()、timed_wait()、signal()和broadcast()与Java中的wait()、wait(long)、notify()和notifyAll()是一一对应的。它们的实现逻辑也是一致的。条件变量的wait()操作要求拥有一个互斥锁。在其等待时会释放锁，而在wait()方法返回前，会重新获取锁。signal()方法唤醒一个线程，而broadcast()方法唤醒所有的等待线程。这些方法都要求在运行过程中拥有一个互斥锁。条件变量中存在的竞态条件是用和Java中的等待和通知机制一样的方式解决的。

但是它们之间也有细微的差别。在Java中，等待和通知机制是和相关联的锁紧密集成的。这使得它比条件变量更容易使用。在同步块中调用wait()和notify()方法是一种很自然的使用方式。但是，如果使用条件变量，就要求先创建一个互斥锁，存储该互斥锁并且在不使用该锁时销毁它。

不幸的是，Java的便利也要求付出一些小小的代价。条件变量和与之相对应的互斥锁是两种不同的互斥体。这使得可以对两个不同的条件变量使用同一个互斥体，对一个条件变量使用两个不同的互斥体，或者对条件变量和互斥体进行任意组合。尽管等待和通知机制比条件变量更容易使用，而且也能够解决以信号为基础的同步问题中的绝大部分，但是它不能将任意同步锁赋予任意通知对象。如果想在用相同的同步锁保护公共数据的同时对两个不同的通知对象发送信号，则条件变量会更有效。

下面是条件变量的实现。

```
public class CondVar {
private BusyFlag SyncVar;
```

```
public CondVar() {
    this(new BusyFlag());
}

public CondVar(BusyFlag sv) {
    SyncVar = sv;
}

public void cvWait() throws InterruptedException {
    cvTimedWait(SyncVar, 0);
}

public void cvWait(BusyFlag sv) throws InterruptedException {
    cvTimedWait(sv, 0);
}

public void cvTimedWait(int millis) throws InterruptedException {
    cvTimedWait(SyncVar, millis);
}

public void cvTimedWait(BusyFlag sv, int millis)
                            throws InterruptedException {
    int i = 0;
    InterruptedException errex = null;

    synchronized (this) {
        // 使用此方法必须拥有锁
        if (sv.getBusyFlagOwner() != Thread.currentThread()) {
            throw new IllegalMonitorStateException(
                                "current thread not owner");
        }

        // (完全地)释放锁
        while (sv.getBusyFlagOwner() == Thread.currentThread()) {
            i++;
            sv.freeBusyFlag();
        }

        // 使用wait()方法
        try {
            if (millis == 0) {
                wait();
            } else {
                wait(millis);
            }
        } catch (InterruptedException iex) {
            errex = iex;
```

```
            }
        }

        // 获得锁（返回到最初状态）
        for (; i>0; i--) {
            sv.getBusyFlag();
        }

        if (errex != null) throw errex;
        return;
    }

    public void cvSignal() {
        cvSignal(SyncVar);
    }

    public synchronized void cvSignal(BusyFlag sv) {
        // 使用此方法必须拥有锁
        if (sv.getBusyFlagOwner() != Thread.currentThread()) {
            throw new IllegalMonitorStateException(
                                "current thread not owner");
        }
        notify();
    }

    public void cvBroadcast() {
        cvBroadcast(SyncVar);
    }

    public synchronized void cvBroadcast(BusyFlag sv) {
        // 使用此方法必须拥有锁
        if (sv.getBusyFlagOwner() != Thread.currentThread()) {
            throw new IllegalMonitorStateException(
                                "current thread not owner");
        }
        notifyAll();
    }
}
```

在上面的代码中，我们反向使用了等待和通知机制，使用BusyFlag类作为同步锁而不是绑定对象的对象锁。线程间的信号是通过等待和通知机制来实现的。同时为了解决 BusyFlag 类和 CondVar 类间存在的竞态条件，我们还使用了标准同步机制与等待和通知机制。

cvWait()机制是通过三个部分来实现的。首先，通过 freeBusyFlag()方法来释放 BusyFlag 锁。因为 BusyFlag 类是可以嵌套的，所以必须释放 busyflag 上的所有锁。为了在以后能够重新获得这些锁，我们必须记录下被释放的所有锁的个数。

第二步是通过调用 wait()方法来映射 Java 的内部实现。最后一步是重新获取先前步骤中释放的锁。在 cvWait()方法的前两个步骤中存在的竞态条件是通过包含这两个步骤的同步块来解决的。因为在第三步中信号已经被接收了，所以不用将第三步也包含到同步块中去。同时，如果此时我们还接收到另外一个信号，线程将会忽略该信号（这类似于 wait()方法同时收到两个通知时的行为）。

cvSingal()和cvBroadcast()方法类似于notify()和notifyAll()方法。这些方法同样被同步以避免与 cvWait()方法产生竞态条件。

大多数时候，当使用条件变量而不是Java的等待和通知机制时，都是希望建立由一个锁控制的两个信号通道（例如，两个变量）。在这种情况下，应该创建一个 BusyFlag 类，同时用该 BusyFlag 类来控制全部的条件变量。这样就可以使用 CondVar 类（该类不需要 BusyFlag 类作为参数）的方法。在更复杂的情况下，如果你需要对同一个变量使用不同的锁，就可以创建一个不使用 BusyFlag 参数的 CondVar 类，然后将 BusyFlag 传递给 wait()和 signal()方法。

经常使用CondVar类的地方是缓冲区管理。当缓冲区满时，如果线程试图将数据写入缓冲区，线程就会阻塞。其他使用缓冲区的线程也必须等待缓冲区中有数据时才能进行读操作。这样，我们就有了一个锁（和缓冲区相关联的锁）和两个条件：空或者满。在这种情况下，使用条件变量比使用纯粹的等待和通知机制更清晰。在本章的后面，我们将给出它的实现。

一个网络服务器类

在套接字联网模型下，服务器端上有很多和客户相连接的套接字，服务器要对它们进行读或写操作。我们已经知道，通过使用一个单独的线程来从套接字读取数

据，就可以解决因为等待数据到来而造成的阻塞现象。在服务器端使用线程还有其他的好处：对于每一个连接的客户都使用一个线程，在该线程中不需要考虑与其他客户相连接的其他线程的运行情况。这大大简化了服务器端的编程：我们可以像在一个时刻只处理一个用户连接那样进行编码。

在本节中，我们将开发这样一个服务器。但是在实际开始前，还是让我们复习一些关于网络的基本知识。

图5-2展示了服务器和客户间的数据连接。在服务器端建立套接字分为两个步骤。第一步是创建一个在某个端口上进行监听的套接字。客户端通过这个端口和服务器协商建立一个私有的连接。

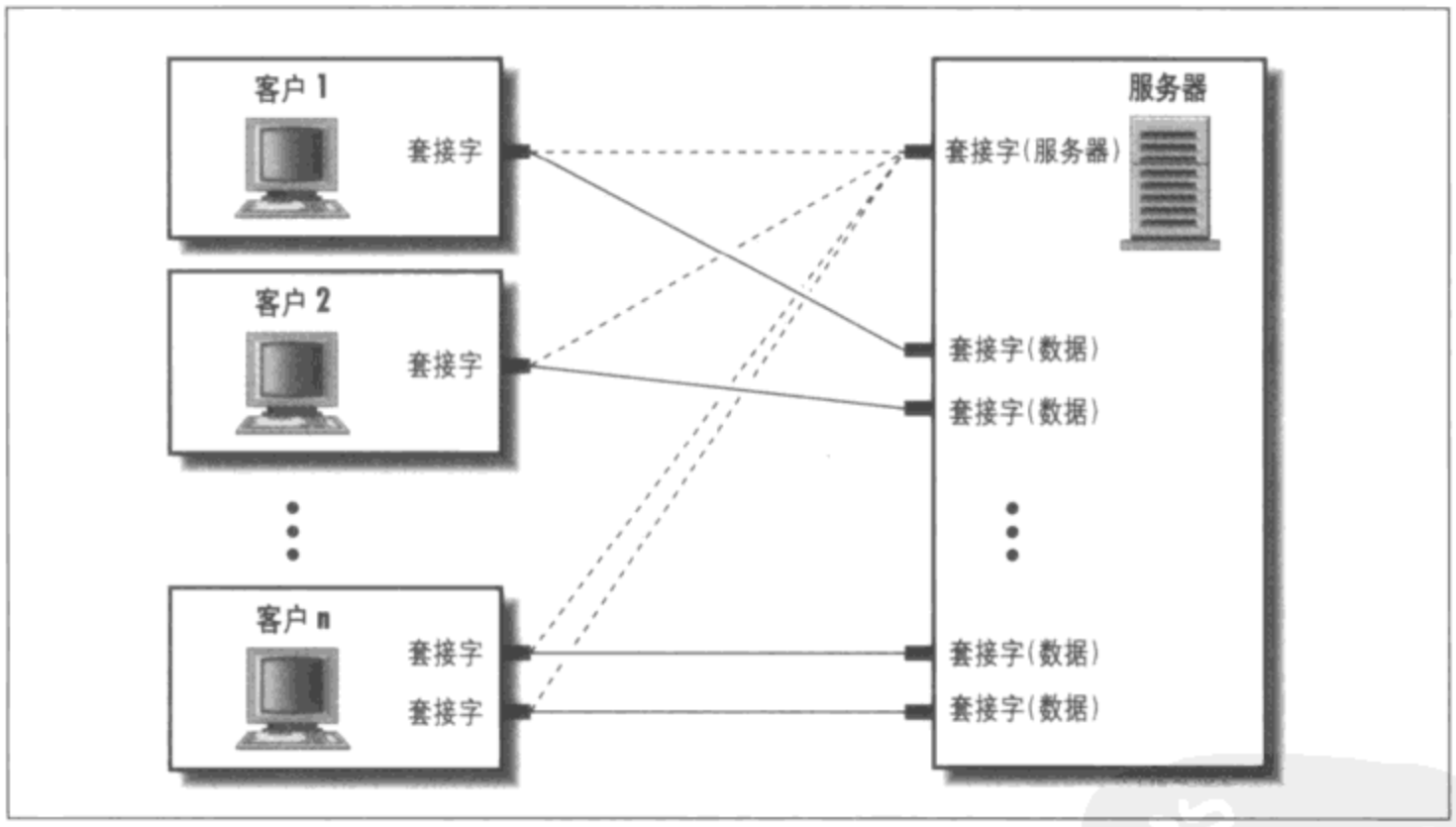

图 5-2：服务器和客户之间的网络连接

一旦建立了该数据连接，服务器和客户就可以通过这个私有的连接进行通信。一般来说，这个步骤是通用的：大多数程序员所关心的仅仅是数据套接字（私有连接）。更进一步，服务器端的数据套接字是仅仅对应于某个特定客户的。尽管可能同时用不同的机制处理许多数据套接字，但是一般而言，每一个套接字都使用同样的代码，而且其运行是与其他套接字无关的。

因为建立连接的过程是通用的，所以可以将它放到TCPSerer中去，而不必每一次都重复实现这些代码。TCPServer 类主要创建一个 ServerSocket，并接收用户的连接请求。这都在一个单独的线程中实现。一旦连接建立了，服务器就对自己进行克隆（拷贝）操作，这样就可以在新的线程中处理新的客户连接。

```
import java.net.*;
import java.io.*;

public class TCPServer implements Cloneable, Runnable {
    Thread runner = null;
    ServerSocket server = null;
    Socket data = null;
    volatile boolean shouldStop = false;

    public synchronized void startServer(int port) throws IOException {
        if (runner == null) {
            server = new ServerSocket(port);
            runner = new Thread(this);
            runner.start();
        }
    }

    public synchronized void stopServer() {
        if (server != null) {
            shouldStop = true;
            runner.interrupt();
            runner = null;
            try {
                server.close();
            } catch (IOException ioe) {}
            server = null;
        }
    }

    public void run() {
        if (server != null) {
            while (!shouldStop) {
                try {
                    Socket datasocket = server.accept();
                    TCPServer newSocket = (TCPServer) clone();

                    newSocket.server = null;
                    newSocket.data = datasocket;
                    newSocket.runner = new Thread(newSocket);
                    newSocket.runner.start();
```

```
                } catch (Exception e) {}
            }
        } else {
            run(data);
        }
    }

    public void run(Socket data) {

    }
}
```

考虑到TCPServer类启动的线程个数，该类的实现是很简单的。首先，TCPServer类实现了 Runnable 接口；我们将创建该类要执行的线程。其次，该类是可克隆的，因此对于每一个连接都可以创建一个该类的新拷贝。同样，因为这个拷贝也是可运行的，因此对每一个客户连接都可以创建另一个线程。因为原始的TCPServer对象必须处理服务器套接字，而拷贝出来的新线程要处理数据套接字，因此 TCPServer 类必须能够同时处理服务器套接字和数据套接字。

一开始，一旦实例化了 TCPServer 对象，就调用 startServer() 方法：

```
public synchronized void startServer(int port) throws IOException {
    if (runner == null) {
        server = new ServerSocket(port);
        runner = new Thread(this);
        runner.start();
    }
}
```

该方法创建一个 ServerSocket 对象和一个处理该对象的独立线程。通过使用一个独立的线程来处理 ServerSocket，startServer() 方法就可以立刻返回。这样同一个程序就可以为作为多个服务器运行。当然，我们可以将这些初始化代码都放到TCPServer 类的构造函数中去；没有什么特别的原因让我们将它们放到一个独立的方法中。

stopServer() 方法是 TCPServer 类的清理方法。

```
public synchronized void stopServer() {
    if (server != null) {
        shouldStop = true;
        runner.interrupt();
```

```
            runner = null;
            try {
                server.close();
            } catch (IOException ioe) {}
            server = null;
        }
    }
```

该方法清理 startServer() 方法所做过的工作。在该例中，我们通过设置要在 run() 方法中检查的标志来停止已启动的线程；另外，为了防止运行的线程阻塞在 accetp() 方法中，我们中断该线程的运行。最后，我们关闭该线程使用的套接字。

我们还将 runner 变量设置为 null，以允许对象被重用：如果 runner 变量是 null，就可以再次调用 startServer() 方法，使得 ServerSocket 可以再次在同一个或者是其他的端口上启动。

还要注意，stopServer() 方法在试图停止服务器前也要检查变量 server 是不是 null。这样做的原因就是 TCPServer 对象会被克隆以处理数据套接字，而对于这些克隆出来的 TCPServer，我们将变量 server 设置为 null，为了防止我们对一个处理数据套接字的 TCPServer 调用 stopServer() 方法，就需要在该方法中检查 server 变量的值。

这些逻辑就组成了 run() 方法：

```
public void run() {
    if (server != null) {
        while (!shouldStop) {
            try {
                Socket datasocket = server.accept();
                TCPServer newSocket = (TCPServer) clone();

                newSocket.server = null;
                newSocket.data = datasocket;
                newSocket.runner = new Thread(newSocket);
                newSocket.runner.start();
            } catch (Exception e) {}
        }
    } else {
        run(data);
    }
```

```
}
```

在 run() 方法中最有趣的就是其中所包含的 if 条件。因为在 startServer() 方法中设置了服务器实例变量，因此 run() 方法中的 if 条件总是为 true。在后面我们还会克隆这个TCPServer对象，并且使用这个克隆的对象来启动更多的线程。在克隆出来的对象中条件代码就不一样了。

对于 ServerSocket 的处理是简单的。我们只需调用 accept() 方法来接收客户的连接请求。ServerSocket 类会处理绑定套接字和设置监听器数目的细节。一旦我们已经接收了客户端的网络连接，我们就可以再次享受使用线程编程带来的好处了。

但是，在该例中，我们使用 TCPServer 类而不是其他 Runnable 类：更进一步，我们在新创建的线程中使用克隆出来的 TCPServer 对象，并且将这个对象配置为可运行对象。这就是为什么 TCPServer 中的 run() 方法要检查 ServerSocket 对象是否有效的原因。我们克隆 TCPServer 对象的原因是为了拥有每个线程的私有数据。通过建立对象的拷贝，我们建立了实例变量的拷贝，这些拷贝可以在以后设置为新创建的线程所需要的值。

在 run() 方法中的 while 循环中包含了所有对 ServerSocket 的处理。run() 方法中的其他部分用来处理客户数据套接字。

```
public void run() {
    if (server != null) {
        ...
    } else {
        run(data);
    }
}

public void run(Socket data) {
}
```

新创建的线程运行一个新克隆出来的可运行对象，它在一开始调用 run() 方法；对于一个数据套接字，run() 方法仅仅是调用被重载的 run(data) 方法。从代码中我们可以了解到，run(data) 方法什么也没有做；TCPServer 类本身是不会

对数据套接字进行任何操作的，因此要实现一个真正有用的TCPServer，我们必须扩展该方法：

```
import java.net.*;
import java.io.*;

public class ServerHandler extends TCPServer {
    public void run(Socket data) {
        try {
            InputStream is = data.getInputStream();
            OutputStream os = data.getOutputStream();

            // 处理数据套接字，从略
        } catch (Exception e) {}
    }
}
```

在子类中要做的就是覆盖run(data)方法；在run(data)方法中只需处理一个数据套接字，而不需要考虑ServerSocket或者其他的数据套接字。当run(data)方法被调用时，它具有一份自己的TCPServer对象的拷贝，并且运行在自己的线程上。TCPServer类的实例隐藏了ServerSocket和其他数据套接字的所有细节。一旦已经开发出了我们自己的TCPServer类版本（在该例中，是ServerHandler类），就可以创建该类的一个实例并且启动它。下面就是一个使用ServerHandler类的例子：

```
import java.net.*;
import java.io.*;

public class MyServer {
    public static void main(String args[]) throws Exception {
        TCPServer serv = new ServerHandler();

        serv.startServer(300);
    }
}
```

使用ServerHandler很简单。我们只需实例化一个TCPServer对象，并且调用它的startServer()方法。因为ServerHandler也是TCPServer对象，因此它能像TCPServer对象一样运行；区别仅仅在于ServerHandler类有自己的对于每一个数据套接字进行处理的代码。

在 TCPServer 中是否也存在着和线程相关的问题，特别是同步问题？基本上来说，我们还没有见到过这种问题。因为要检查可能改变的公共实例变量，所以 startServer()和 stopServer()方法都是同步的。因为 startServer()方法已经保证了 run()方法仅仅被调用一次，所以 run()方法并不需要被同步。

TCPServer 类和 applet

在对 TCPServer 类的使用中，我们实现了一个独立的提供服务的应用程序。这种服务对于应用程序或者是 applet（或用其他语言编写的程序）而言都是有效的。

当然我们也可以想象出一些使用 applet 来提供网络服务的情况。这些 applet 被浏览器下载然后向用户提供服务。这种服务是根据需要提供的，并且可以在任意时刻被停止。但是在这种临时性的环境中，没有什么服务对网络上的其他客户是有用的。

对于每一个套接字连接而言，对 run()方法的调用都是在一个被克隆出来的 TCPServer 对象中进行的，而每一个数据套接字线程都将改变和检查 TCPServer 类的不同实例，因此没有理由来同步这些处理数据套接字的线程。那些单独的处理数据套接字的线程之间并没有共享数据，因此也没有必要被同步。同时，如果 ServerHandler 类需要共享数据的话，则这个同步操作应该在 ServerHandler 或者某个支持类中实现。

在该例中，我们使用了 Runnable 接口技术。我们能不能直接从 Thread 类派生而不使用 Runnable 接口呢？答案是肯定的。但是使用 Runnable 接口使得 TCPServer 类可以用一个新的线程来运行自身的一个拷贝。而通过 Thread 类派生的话就需要另外一个不同的实现了。这个实现可能要求实例化一个新的 TCPServer 类而不是简单地克隆出一个来。

我们并没有在系统中保存“数据套接字”的线程对象的引用；这会引起问题吗？答案是否定的。前面已经说过，线程系统会在其内部保存所有活动线程的引用。只要 stop()方法没有被调用或者该线程的 run()方法还没有结束，线程就被认

为是活动的，线程系统就会在某处保存一个该活动线程的引用。虽然删除一个线程对象的所有引用使得TCPServer不能停止这个数据套接字线程，但是垃圾收集系统还是不能回收这个线程对象，因为线程系统仍然保存着对它的引用。

不知你是否注意到了，其实很难分辨出ServerHandler类和MyServer类到底有没有被线程化。其实这也是我们试图去达到的目标。线程仅仅是一个工具，线程系统仅仅是提供服务的。我们的最终目的是解决问题。一个正确设计的类是不必苛求使用什么工具的。ServerHandler类只需处理一个数据套接字，而MyServer类只需启动ServerHandler。是否使用线程是实现的细节问题。这就是面向对象编程的好处。

AsyncInputStream类

前面开发的AsyncReadSocket类有着如下的问题：

- 该类是特定于网络套接字的。但是我们也许要对文件、管道或者任何数据流使用异步I/O类。最理想的是可以使得任何数据源都具有异步的能力，而不仅仅是网络套接字。

  对于从流中进行输入已经有了一个类结构。输入流的顶层是InputStream类，因此我们应当使用该类的子类。我们同样可以从嵌套的FilterInputStream和其子类中获益。

- 不同于TCPServer类，AsyncReadSocket类没有很好地隐藏其线程的细节。

我们是否需要为此开发一个新的类？难道InputStream类没有支持异步I/O的方法吗？尽管在开发AsyncReadSocket类的过程中很少提到，但InputStream类的`available()`方法其实可以不被阻塞地返回在该流上可以读取的字节数。尽管这个方法听起来很有用，但是因为它返回的是已经被读取而等待处理的字节数，因此并不总是符合我们的需要。在有些操作系统上，返回的字节数可能包括那些已经被机器读取但被操作系统保有的字节。不过并不是所有的操作系统都这样（微软、苹果、Sun和其他Unix厂商提供的操作系统都是这样的）。

因此，available()方法返回0并不意味着read()方法会阻塞。因为我们开发该类的主要原因就是为了避免阻塞，因此available()方法不适合我们的要求。

另外，在自己的程序中缓存数据比依赖操作系统缓存数据要好。因为数据已经从操作系统拷贝到我们的程序中了，所以当程序不是很忙时（例如用户在思考时），这样操作就比直接从输入流中读取要稍微快一点。

因此我们所需要的就是一个 InputSteam 类，它的 available()方法能够正确报告在不阻塞的情况下可以实际读取的字节数，同时自己对数据进行缓存。这个新类AsyncInputStream的实现类似于AsyncReadSocket类。它创建另一个线程来从输入流中读取数据。因为是使用其他线程来读取数据，所以在数据不可用时 read()方法被阻塞也不会造成什么影响。我们可以像使用 InputStream 对象一样来使用 AsyncInputStream 类。如图 5-3 所示，AsyncInputStream 实际上是从 FilterInputStream 派生出来的，而 FilterInputStream 是包含 InputStream 实例的 InputStream 类的基类。

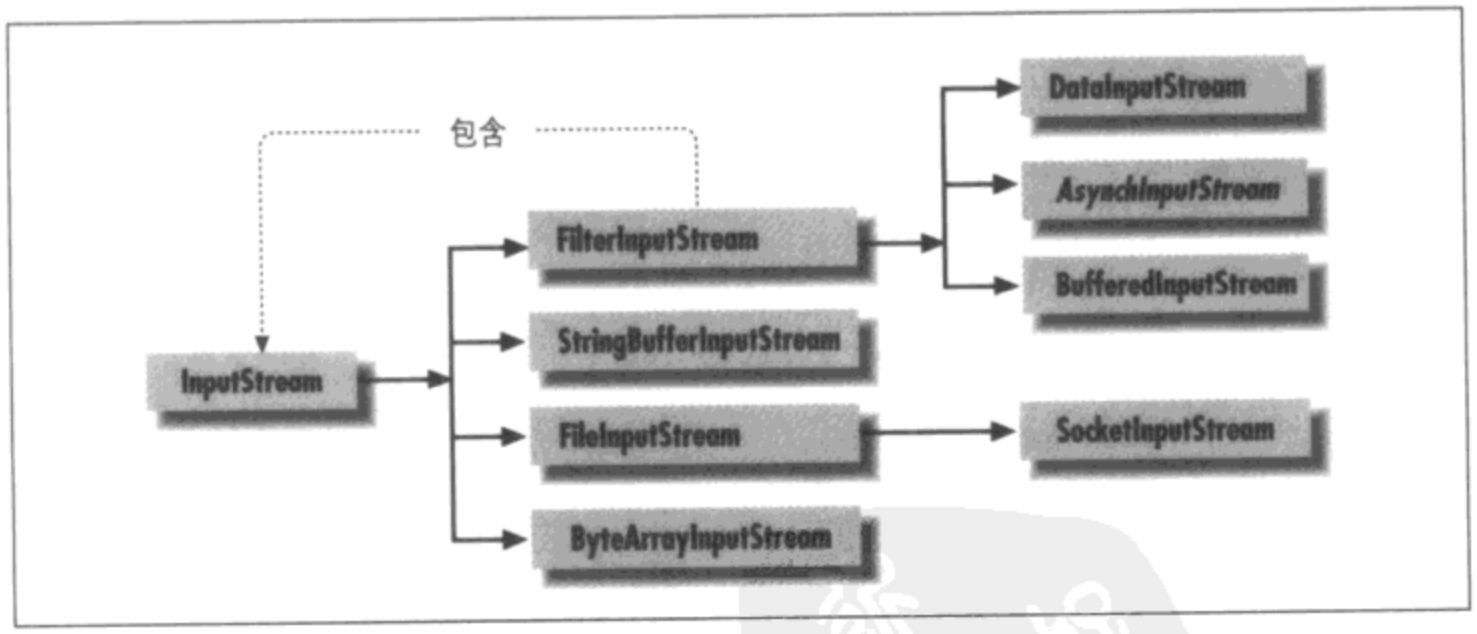

图 5-3: Java InputStream 类的层次关系

使用另外一个线程来读取数据是实现的细节。在我们研究AsyncInputStream类的细节前，先来看看 AsyncInputStream 类本身：

```
import java.net.*;
import java.io.*;
```

```
public class AsyncInputStream extends FilterInputStream
                              implements Runnable {
    private Thread runner;                    // 同步的读取线程
    private volatile byte result[];           // 缓冲区
    private volatile int reslen;              // 缓冲区大小
    private volatile boolean EOF;             // 文件尾指示符
    private volatile IOException IOError;     // I/O异常

    BusyFlag lock;                            // 数据锁
    CondVar empty, full;                      // 信号变量

    protected AsyncInputStream(InputStream in, int bufsize) {
        super(in);

        lock = new BusyFlag();                // 分配同步变量
        empty = new CondVar(lock);
        full = new CondVar(lock);

        result = new byte[bufsize];           // 分配存储区域
        reslen = 0;                           // 同时初始化变量
        EOF = false;
        IOError = null;
        runner = new Thread(this);            // 启动读取线程
        runner.start();
    }

    protected AsyncInputStream(InputStream in) {
        this(in, 1024);
    }

    public int read() throws IOException {
        try {
            lock.getBusyFlag();
            while (reslen == 0) {
                try {
                    if (EOF) return(-1);
                    if (IOError != null) throw IOError;
                    empty.cvWait();
                } catch (InterruptedException e) {}
            }
            return (int) getChar();
        } finally {
            lock.freeBusyFlag();
        }
    }

    public int read(byte b[]) throws IOException {
```

```
        return read(b, 0, b.length);
    }

    public int read(byte b[], int off, int len) throws IOException {
        try {
            lock.getBusyFlag();
            while (reslen == 0) {
                try {
                    if (EOF) return(-1);
                    if (IOError != null) throw IOError;
                    empty.cvWait();
                } catch (InterruptedException e) {}
            }

            int sizeread = Math.min(reslen, len);
            byte c[] = getChars(sizeread);
            System.arraycopy(c, 0, b, off, sizeread);
            return(sizeread);
        } finally {
            lock.freeBusyFlag();
        }
    }

    public long skip(long n) throws IOException {
        try {
            lock.getBusyFlag();
            int sizeskip = Math.min(reslen, (int) n);
            if (sizeskip > 0) {
                byte c[] = getChars(sizeskip);
            }
            return((long)sizeskip);
        } finally {
            lock.freeBusyFlag();
        }
    }

    public int available() throws IOException {
        return reslen;
    }

    public void close() throws IOException {
        try {
            lock.getBusyFlag();
            reslen = 0;                          // 清空缓冲区
            EOF = true;                          // 标记文件尾
            empty.cvBroadcast();                 // 向所有线程发出信号
            full.cvBroadcast();
```

```
        } finally {
            lock.freeBusyFlag();
        }
    }

    public void mark(int readlimit) {
    }

    public void reset() throws IOException {
    }

    public boolean markSupported() {
        return false;
    }

    public void run() {
        try {
            while (true) {
                int c = in.read();
                try {
                    lock.getBusyFlag();
                    if ((c == -1) || (EOF)) {
                        EOF = true;                    // 标记文件尾
                        in.close();                    // 关闭输入流
                        return;                        // 结束I/O线程
                    } else {
                        putChar((byte)c);              // 存放读取的字节
                    }
                    if (EOF) {
                        in.close();                    // 关闭输入流
                        return;                        // 结束I/O线程
                    }
                } finally {
                    lock.freeBusyFlag();
                }
            }

        } catch (IOException e) {
            IOError = e;                               // 保存发生的异常
            return;
        } finally {
            try {
                lock.getBusyFlag();
                empty.cvBroadcast();                   // 向所有的线程发出信号
            } finally {
                lock.freeBusyFlag();
            }
        }
```

```
    }

    private void putChar(byte c) {
        try {
            lock.getBusyFlag();
            while ((reslen == result.length) && (!EOF)) {
                try {
                    full.cvWait();
                } catch (InterruptedException ie) {}
            }
            if (!EOF) {
                result[reslen++] = c;
                empty.cvSignal();
            }
        } finally {
            lock.freeBusyFlag();
        }
    }

    private byte getChar() {
        try {
            lock.getBusyFlag();
            byte c = result[0];
            System.arraycopy(result, 1, result, 0, --reslen);
            full.cvSignal();
            return c;
        } finally {
            lock.freeBusyFlag();
        }
    }

    private byte[] getChars(int chars) {
        try {
            lock.getBusyFlag();
            byte c[] = new byte[chars];
            System.arraycopy(result, 0, c, 0, chars);
            reslen -= chars;
            System.arraycopy(result, chars, result, 0, reslen);
            full.cvSignal();
            return c;
        } finally {
            lock.freeBusyFlag();
        }
    }
}
```

我们并不关心如何使用线程来进行 I/O 操作的细节；该类中使用线程的代码和

AsyncReadSocket类没有什么区别。新的线程只是在输入流上进行阻塞方式的读取，并提供了使得主线程以非阻塞方式获取数据的方法。该类中关于InputStream的部分是很有趣的，但是对Java数据输入流的讨论超出了本书的范围。

为什么对这个类的讨论很重要呢？该类与AsyncReadSocket类有什么不同呢？
尽管该类和AsyncReadSocket类一样实现了异步读取，但它同时也是一个FilterInputStream类，而且我们也关心多线程的I/O操作和InputStream类之间的关系。因为该类的行为必须和InputStream类一样，所以我们就不能像原来那样，由于只关注如何同I/O线程进行通信而选取最优的接口。这也是我们在实现多线程类时向现实世界所做的妥协吧。

为了使该类的方法能够工作正常，我们需要使用已知的所有同步技术。让我们看看该类的实例变量和构造函数：

```
public class AsyncInputStream extends FilterInputStream
                              implements Runnable {
    private Thread runner;                    // 同步的读取线程
    private volatile byte result[];           // 缓冲区
    private volatile int reslen;              // 缓冲区大小
    private volatile boolean EOF;             // 文件尾指示符
    private volatile IOException IOError;     // I/O异常

    BusyFlag lock;                            // 数据锁
    CondVar empty, full;                      // 信号变量

    protected AsyncInputStream(InputStream in, int bufsize) {
        super(in);

        lock = new BusyFlag();                // 分配同步变量
        empty = new CondVar(lock);
        full = new CondVar(lock);

        result = new byte[bufsize];           // 分配存储缓冲区
        reslen = 0;                           // 并且初始化变量
        EOF = false;
        IOError = null;
        runner = new Thread(this);            // 启动读取线程
        runner.start();
    }

    protected AsyncInputStream(InputStream in) {
```

```
        this(in, 1024);
    }
```

最开始的三个实例变量：runner、result 和reslen，是该类最重要的数据。runner存放由该类启动的I/O线程的引用，result和reslen存放从I/O线程取回的数据以及数据长度。这和AsyncReadSocket类不同，AsyncReadSocket并不提供对数据长度的支持：AsyncReadSocket的getResult()方法不允许调用者指定要读取的字节数。因为InputStream类可以读取任意长度的数据，所以我们必须跟踪缓冲区中的可用数据。

为了能够像InputStream类一样报告EOF（End-Of-File，文件尾）和抛出I/O异常，我们定义了EOF和IOError实例变量。EOF条件和I/O异常是由AsyncInputStream类中包含的InputStream对象产生的。我们需要在I/O线程中保存EOF事件并捕获I/O异常，然后在调用线程中指示EOF发生或者是抛出I/O异常。如果AsyncInputStream类不必和InputStream类具有同样的行为，则完全可以使用更简单的错误报告系统。

缓冲区result中的数据是由实例变量lock保护的，并且我们还将两个条件变量（empty和full）与锁相关联。这是我们曾经讨论过的管理缓冲区的CondVar类的一个实例：我们可以让线程在与一个锁相关联的两个条件上等待。

AsyncInputStream类的第一个构造函数很简单。首先，我们为缓冲区分配和初始化空间，然后对那些用来和I/O线程通信的变量进行初始化。然后，我们实例化并且启动I/O线程。另外一个构造函数有着和FilterInputStream类相同的函数签名，对于这个函数，我们使用默认的缓冲区大小。通过提供这个构造函数，我们的接口才类似于FilterInputStream。

下面来看看数据是如何传回给用户的：

```
public int read() throws IOException {
    try {
        lock.getBusyFlag();
        while (reslen == 0) {
            try {
                if (EOF) return(-1);
```

```
                    if (IOError != null) throw IOError;
                    empty.cvWait();
                } catch (InterruptedException e) {}
            }
            return (int) getChar();
        } finally {
            lock.freeBusyFlag();
        }
    }

    private byte getChar() {
        try {
            lock.getBusyFlag();
            byte c = result[0];
            System.arraycopy(result, 1, result, 0, --reslen);
            full.cvSignal();
            return c;
        } finally {
            lock.freeBusyFlag();
        }
    }
```

在InputStream类中，read()方法从输入流中读取单个字节。如果I/O线程碰到了EOF或者是IOException，就应该分别用实例变量EOF或者IOError来代替。read()方法返回-1表示EOF，或者代表I/O线程抛出一个IOException。

InputStream和文件尾

很明显，对于一个FileInputStream而言，EOF表示读到了文件的最后一个字节。但是对于其他的数据源这又是什么意思呢？

EOF可以由多种原因产生，例如StringBufferInputStream报告的字符串尾、ByteArrayInputStream报告的数组尾或者是SocketInputStream报告的网络连接关闭。

无论是哪种情况，我们都应当将该指示符当成是不可能从数据源中读取更多数据的意思。这个实际的数据源是什么则不是我们所关心的。

另外，我们只是在缓冲区中没有数据时才检查EOF或者I/O异常。因为I/O线程是先进行读操作的，所以我们只有等待用户将缓冲区中的数据都读取完后才能在

read()方法中表示EOF或者抛出异常：用户应该在EOF或者是异常实际发生的地方碰到它们。当I/O线程碰到EOF或者IOException时，就会停止读取数据，因此我们可以保证缓冲区中的数据都在它们发生之前被读取。

最后，为了保护数据缓冲区result和长度指示符reslen，我们使用BusyFlag锁。getChar()方法返回下一个字符，它也同样使用BusyFlag。你可能会问，我们为什么使用一个锁来保护四个不同的实例变量？这是一个设计上的问题；因为result和reslen两个变量是相关的，并且不可能只改变其中的一个。EOF和IOError变量在I/O线程的生命周期中只会使用一次。因此在一个合适的锁已经存在的情况下，再去创建一个新的BusyFlag来保护EOF和IOError是没有什么意义的。

如果调用read()方法的时候没有数据会发生什么情况呢？ read()方法必须正确处理这种情况。这意味着read()方法在这种情况下会阻塞。换句话说，read()方法所做的违背了它的设计初衷。

```
        public int read() throws IOException {
            try {
                lock.getBusyFlag();
                while (reslen == 0) {
                    try {
                        if (EOF) return(-1);
                        if (IOError != null) throw IOError;
                        empty.cvWait();
                    } catch (InterruptedException e) {}
                }
                return (int) getChar();
            } finally {
                lock.freeBusyFlag();
            }
        }

        private void putChar(byte c) {
            try {
                lock.getBusyFlag();
                while ((reslen == result.length) && (!EOF)) {
                    try {
                        full.cvWait();
                    } catch (InterruptedException ie) {}
                }
```

```
            if (!EOF) {
                result[reslen++] = c;
                empty.cvSignal();
            }
        } finally {
            lock.freeBusyFlag();
        }
    }
```

很显然，read()方法不应该阻塞在从InputStream中读取数据上；InputStream处于I/O线程的控制之下，因此不能直接被read()方法访问。为了模拟这种阻塞，我们使用了条件变量empty。read()方法只是等待更多数据的到来。如果I/O线程读取到了数据，那么在将数据放入缓冲区中时会产生一个信号，这是通过调用putChar()方法中的cvSignal()方法来完成的。像在run()方法中看到的那样，I/O线程调用putChar()方法来将数据放入缓冲区。

```
public void run() {
    try {
        while (true) {
            int c = in.read();
            try {
                lock.getBusyFlag();
                if ((c == -1) || (EOF)) {
                    EOF = true;              // 标记文件尾
                    in.close();              // 关闭输入流
                    return;                  // 结束I/O线程
                } else {
                    putChar((byte)c);        // 存储读取的字节
                }
                if (EOF) {
                    in.close();              // 关闭输入流
                    return;                  // 结束I/O线程
                }
            } finally {
                lock.freeBusyFlag();
            }
        }
    } catch (IOException e) {
        IOError = e;                         // 保存异常
        return;
    } finally {
        try {
            lock.getBusyFlag();
            empty.cvBroadcast();             // 向所有的线程发出信号
```

```
            } finally {
                lock.freeBusyFlag();
            }
        }
    }
```

用于I/O线程的代码类似于AsyncReadSocket类中的代码。我们从输入流中读取数据，如果有必要就进入阻塞状态。当我们接收到数据时，使用putChar()方法将其放入缓冲区中。另外，如果碰到EOF或者是截获一个IOException，就将相关的信息放到恰当的实例变量中去。为了使这些活动在与其他线程交互时也能安全地进行，我们在线程中使用了同一个锁：BusyFlag。

当EOF或者IOException发生时，所有被阻塞的读线程将会有什么结果呢？正如我们已经提到的那样，通过使用条件变量可以使read()方法按阻塞方式进行。但是，当EOF或者是IOException条件发生时，因为没有数据到来，所以不会有通知发生。为了解决这个问题，我们必须在这些条件发生时使用cvBroadcast()方法。那些等待的线程将会依次被唤醒，然后处理缓冲区中存在的数据。

```
public void run() {
    try {
        while (true) {
            int c = in.read();
            try {
                lock.getBusyFlag();
                if ((c == -1) || (EOF)) {
                    EOF = true;                    // 标记文件尾
                    in.close();                    // 关闭输入流
                    return;                        // 结束I/O线程
                } else {
                    putChar((byte)c);              // 存储读取的字节
                }
                if (EOF) {
                    in.close();                    // 关闭输入流
                    return;                        // 结束I/O线程
                }
            } finally {
                lock.freeBusyFlag();
            }
        }

    } catch (IOException e) {
        IOError = e;                               // 保存异常
```

```
            return;
        } finally {
            try {
                lock.getBusyFlag();
                empty.cvBroadcast();                    // 向所有的线程发出信号
            } finally {
                lock.freeBusyFlag();
            }
        }
    }

    public void close() throws IOException {
        try {
            lock.getBusyFlag();
            reslen = 0;                                 // 清空缓冲区
            EOF = true;                                 // 标记到达文件尾
            empty.cvBroadcast();                        // 向所有的线程发出信号
            full.cvBroadcast();
        } finally {
            lock.freeBusyFlag();
        }
    }
```

当缓冲区中没有数据时，剩下的从 InputStream 中进行读取操作的线程将会从它们的 read()方法中返回 EOF 或者 IOError。不必担心以后可能发生的对 read()方法的调用，因为它们仅仅返回 EOF 或者 IOError。

available()方法的实现和我们所期望的一样(这也是我们设计AsyncInputStream类的动机)，它的实现是很简单的。

```
public int available() throws IOException {
    return reslen;
}
```

它只是简单地返回缓冲区中的字节个数。因为I/O线程实际上是从InputStream中读取数据的，并且在没有数据可以读取时会进入阻塞状态，因此我们就会知道此时在网络上没有可以读取的数据。但是，缓冲区中数据的最大数目是由用户配置的，因此有可能出现缓冲区满以及读取的数据无法存放到缓冲区中去的情况。

最后，还有三个与 AsyncInputStream 类的设计相关的结论。尽管它们对于AsyncInputStream类很重要，但因为与线程没有关系，所以我们不会在此研究它们，只是为了完整起见而给出一个简单的概述：

- read(byte[])方法，类似于read()方法，当数据不可用时会阻塞。但是，如果存在可以使用的数据，但是不足以填充该字节数组，read(byte[])方法仅仅读取这部分数据，而且返回值是实际读取的字节数。我们是根据AsyncInputStream类的设计原则（以异步方式工作）来选择这个实现方法的，而且该实现也是最符合这个设计精神的。
- skip()方法非阻塞地跳过指定的字节数。如果skip()方法要跳过的字节数多于当前可用的字节数，该方法仅仅跳过当前可用的数据并且返回实际跳过的字节数。再次说明，该实现同样符合AsyncInputStream类的设计思想。
- 尽管InputStream类支持标记和重置功能，但AsyncInputStream类不支持它。这是因为没有什么实际的理由要求一个异步流来支持这个功能。如果用户的确需要这个功能，他们可以实例化一个BufferedInputStream对象来包含AsyncInputStream对象，从而获得这个功能。

AsyncOutputStream类？

不实现AsyncWriteSocket类的原因是：这样做没有用处。数据在网络连接两端的很多地方都实现了缓冲，因此没有理由去担心write()方法会被阻塞很长一段时间。但是，尽管很罕见，write()方法还是有可能被阻塞。

对于AsyncOutputStream类，还有另外一个复杂性：如果OutPutStream没有被正确地传递，其write()方法就会抛出I/O异常。调用AsyncOutputStream的write()方法可能要在很长时间后才能够返回。这可以通过在随后对write()方法或者close()方法进行调用时抛出异常来处理。这不是一个好的解决方法，但是在许多将数据写到缓冲区的例子中是很常见的。

那些希望实现一个真正健壮的程序的程序员可能会根据我们的AsyncInputStream类来实现他们自己的AsyncOutputStream类。

为什么要使用两个条件变量而不是等待和通知机制？这是为了效率。在该例中，我们有一个数据源（缓冲区result），它可以有两个条件：空（此时如果线程试图读取数据，就会进入等待数据到来的状态）或满（当线程试图存储数据到缓冲区时也会进入等待数据被消耗的状态）。如果使用等待和通知机制，则无论哪种情

况发生，都必须要调用notifyAll()方法，这会导致唤醒太多的线程。当然，这也是可以工作的，但是唤醒所有的线程来重新检查条件不如条件变量更有效率。

AsyncInputStream类的实例类似于任何一个InputStream对象。它们可以毫无问题地在使用InputStream类的地方使用。虽然AsyncInputStream类也是Runnable类型的，但这只是实现的细节。使用AsyncInputStream类的用户甚至不需要了解到当一个AsyncInputStream对象被实例化时会创建一个新的线程。

使用 TCPServer 和 AsyncInputStream

下面我们修改ServerHandler类来使用异步的方式从客户端读取请求。

```
import java.net.*;
import java.io.*;

public class ServerHandler extends TCPServer {
    public void run(Socket data) {
        try {
            InputStream is =
                new AsyncInputStream(data.getInputStream());
            OutputStream os = data.getOutputStream();

            // 在此对数据套接字进行处理
        } catch (Exception e) {}
    }
}
```

在ServerHandler类中只改动一行代码，就变成了以异步的方式从客户端读取数据。同时我们也将提供服务的线程个数翻了一番。虽然从代码上看不出有什么线程被创建了，但是实际上对于每一个客户端都有两个线程在为其服务。

总结

在本章中，我们了解了一些有关线程和相关同步问题的实际例子。从上一章开始，我们就单纯地将线程作为一个工具来使用。我们启动了新线程，并在这些线程间进行通信，但是使用这些类的客户可能并不关心甚至不知道这些线程的存在。

我们还研究了在没有使用线程的情况下与同步相关的一些问题。在设计一个简单的像容器这样的例子时都要牢记线程。这是因为，尽管我们没有使用任何线程，但是程序已经被线程化了。我们不仅要在类的实现细节上考虑到线程的存在，还要考虑到整个系统的其他类。不管我们的类是否使用了线程，对于由线程引起的死锁和竞态条件问题，都要在设计中给予考虑。

第六章

Java线程调度

本章内容：

- 线程调度概述
- 何时调度是重要的
- 调度和线程优先级
- 常见的调度实现
- 本地调度支持
- 其他线程调度方法
- 总结

到现在为止，我们已经了解了Java线程系统的基本知识，也能够使用Java线程技术来编写复杂的程序了。现在，我们将开始进入和线程系统相关的一些专门领域。下面几章所要介绍的技术和问题在日常编程中是不会遇到的，只有当你需要对线程行为进行一些直接的控制时，它们的重要性才会显示出来。

首先，我们要在本章中讨论线程调度这一主题。在大多数Java程序中，线程个数都多于该程序运行的机器所配置的CPU个数。因为在同一时刻，一个CPU上只能够运行一个线程，所以不是程序中所有的线程都能同时运行。因此Java线程调度的主要任务就是决定在某一时刻运行程序中的哪一个线程。

理解Java线程调度的关键是认识到CPU是一个有限的资源。当两个或者更多的线程在一个单CPU的机器上运行时，它们会竞争CPU一直到运行结束。因此需要第三方——程序、Java虚拟机或者是操作系统来保证这些线程共享CPU。对于一个多CPU的机器而言，当一个程序中的线程个数多于CPU个数时，道理也是同样的。因此，本章的关键就是如何在试图使用CPU的线程间分配CPU。

在先前的例子中，我们并不关心调度问题，这是因为在那些例子中，线程调度的细节对我们并不重要。我们关心的那些线程并非经常竞争CPU：它们都有特定的任务去完成，但是这些线程要么运行时间很短，要么只是周期性地使用CPU来完

成它们的任务。以调用getImage()方法时自动创建的加载一个图像的线程为例。在大多数时间内，该线程都是在等待从网络传来的数据而不使用CPU。当一定数量的数据到达后，该线程很快就可以处理完这些数据，然后继续等待更多的数据。因为该线程并不是经常需要使用CPU，所以也就没有必要关心Java虚拟机的调度机制了。

Java 规范并没有规定 Java 虚拟机的实现要用什么样的方式来调度线程。尽管在 Java 规范中规定调度方针是基于线程优先级的，但是该指导方针也不是绝对的，而且不同的虚拟机以不同的方式来遵循这个指导方针。另外，Thread 类中和线程调度相关的方法 suspend()和resume()，在 Java 2 中已经不建议使用了（并且不应该在所有的Java版本中使用）。因此，我们没有办法保证不同 Java 虚拟机上线程的执行顺序。

许多人，特别是那些刚刚开始使用Java线程的新手，会对此感到很吃惊：对这个主题的讨论所用的时间大大超过其应该占有的比例。实际上本章介绍的技术很少被用来影响Java线程调度。在很大程度上，只有当一个或者多个线程在很长一段时间内都是占用 CPU 较多时才需要考虑 Java 线程调度。而前面提到的加载图像的线程，尽管也是占用CPU较多的，但是因为它们仅仅需要很短的CPU时间，所以我们并不关心这些线程是如何被调度的。

线程调度概述

我们从线程调度的基本原则开始。任何一个虚拟机都可能不是完全遵循这些原则，但是这些原则是我们了解线程调度的基础。

我们先以一个占用 CPU 较多的线程为例：下面的程序会输出什么呢？

```
class TestThread extends Thread {
    String id;

 public TestThread(String s) {
        id = s;
    }
```

程序分类

用 Java 或者其他语言编写的程序可以分为以下三类：

占用 CPU 较多型

指那些需要很多 CPU 周期来完成任务的程序。它们在很长的时间内要使用大量的CPU 周期来完成数学或者符号计算（例如，对字符串或图像进行操作），但是不需要或者是很少需要从用户或者其他外部数据源得到输入。

占用 I/O 较多型

那些有大半时间在等待I/O操作结束的程序：对磁盘中的文件进行读写操作 、通过网络套接字进行读写操作或者与另外一个程序进行通信。

交互型

对于用户的输入进行相应处理的程序。当用户执行一个特定的动作时，程序就可能运行一个占用 CPU 较多或者是占用 I/O 较多的任务，当任务结束后，程序再次等待用户的下一个命令。第五章讨论的TCPServer 属于这一类，尽管交互是来自于其他（客户）程序而不是用户输入。

一个程序的运行过程在不同的阶段可能属于不同类别，从而可能属于所有这三个类别。

```
    public void doCalc(int i) {
        // 根据 i 的值进行复杂的处理过程
    }

    public void run() {
        int i;
        for (i = 0; i < 10; i++) {
            doCalc(i);
            System.out.println(id);
        }
    }
}

public class Test {
    public static void main(String args[]) {
        TestThread t1, t2, t3;
        t1 = new TestThread("Thread 1");
        t1.start();
```

```
            t2 = new TestThread("Thread 2");
            t2.start();
            t3 = new TestThread("Thread 3");
            t3.start();
        }
    }
```

假设 doCalc()方法的运算是很复杂的，每一次调用都要运行 3 到 5 秒钟，在这段时间内不可能调用其他方法。显然，当该程序运行结束后，我们会有 10 行的“Thread 1”， 10 行的“Thread 2”，10 行的“Thread 3”，但是它们的顺序如何呢？

通常都是假设输出结果的顺序随机的，可能如下所示:

```
Thread 1
Thread 2
Thread 2
Thread 3
Thread 1
Thread 2
Thread 3
```

但是这证明了Java并没有明确地指定线程如何被调度，它并不要求能够产生随机输出的调度种类。如果我们看见 10 行的“Thread 1”后面跟着 10 行的“Thread 2”、再后面跟着 10 行的“Thread 3”，就意味着线程 1 在线程 2 开始运行前结束，而线程 2 在线程 3 开始前结束。

为了理解为什么会这样，我们需要了解Java线程机制的一些内部组成。从概念上来说，Java 虚拟机中的每一个线程都处于下面四种状态之一:

初始状态（initial）

一个线程对象从创建（就是其构造函数被调用）时起到start()方法被调用之间都是处于初始状态。

可运行状态（runnable）

从线程的start()方法被调用开始，它就进入可运行状态了。有很多种方法可以使线程离开可运行状态，但是可运行状态可以看成是线程的默认状态:如果一个线程不是处于其他状态，就认为它是在可运行状态。

阻塞状态（blocked）

当一个线程等待某个特定的事件发生时即处于阻塞状态。一个简单的例子就是当一个线程试图通过一个套接字从远程数据服务器读取数据，却没有数据可以读取时，它就处于阻塞状态。处于睡眠或者是等待一个对象锁的线程也被认为是处于阻塞状态。

退出状态（exiting）

当一个线程的`run()`方法返回或者`stop()`方法被调用时就是处于退出状态。

在Java程序中，经常有多个线程同时处于可运行状态。在这种情况下，会从处于可运行状态的线程池中挑选出一个作为当前运行线程。池中的其他线程尽管是在等待运行的机会（也就是成为当前运行线程），但仍然处于可运行状态。因此问题的关键就是池中哪一个线程会被挑选出来作为当前运行线程。

为了简化我们的讨论，我们认为当前运行线程仅有一个。在一个多处理器的机器和特定的虚拟机实现中，可能存在多个当前运行线程 —— 有可能和机器拥有的处理器一样多。但是对于一个特定的CPU，如何选择一个线程也遵循我们现在正在讨论的这些原则。

Java实现的是抢占式的、基于优先级的调度程序。这就意味着Java程序中的每一个线程都被赋予了一个优先级，该优先级是一个特定范围内的非负整数。无论是线程改变其可运行状态还是线程运行了很长的时间，Java虚拟机都不会改变线程的优先级，优先级的值只能由程序员改变。因此一个优先级为5的线程从它被创建时开始，虽然会在运行和阻塞状态之间进行多次变迁，但是直到该线程进入退出状态为止，都会一直保持该优先级。

Java虚拟机遵循的调度原则是：当前运行线程应当是所有处于可运行状态的线程中具有最高优先级的线程，因此线程优先级的值对于调度很重要。这就是我们说Java实现了一个以优先级为基础的调度程序的原因。Java虚拟机实现的调度程序还是抢占式的，也就是说，当一个高优先级的线程进入可运行状态时，Java虚拟机会中断当前正在运行的低优先级线程，以使得高优先级的线程可以成为当前运行线程。

调度例子：具有不同优先级的线程

通过一个例子可以更清楚地说明这一点。让我们看看下面的代码：

```
public class SchedulingExample implements Runnable {
    public static void main(String args[]) {
        Thread calcThread = new Thread(this);
        calcThread.setPriority(4);
        calcThread.start();

        AsyncReadSocket reader;
        reader = new AsyncReadSocket(new Socket(host, port));
        reader.setPriority(6);
        reader.start();

        doDefault();
    }

    public void run() {
        doCalc();
    }
}
```

该例子有三个线程：第一个是在main()方法中运行的主线程，它在创建其他两个线程后运行doDefault()方法。第二个线程是一个计算线程（calcThread），它调用doCalc()方法。第三个是读取套接字的AsyncReadSocket线程（见第三章）reader。

在下面的讨论中，尽管Java虚拟机中还有很多其他的线程，但还是假设Java虚拟机中只存在我们创建的线程。为了简单起见，我们忽略那些经常处于阻塞状态而对我们的讨论没有什么影响的线程。图6-1显示了例子中的线程在不同状态间变迁的过程。

我们在时间T1处由默认的主线程开始执行main()方法。此时这个优先级为5的初始线程是Java虚拟机中惟一的活动线程。因此它是处于可运行状态的，而且也是当前运行线程。在时间T2，该默认线程创建calcThread，将它的优先级设定为4，并且调用它的start()方法。现在Java虚拟机中就有两个同时处于可运行状态的线程，但是因为默认线程具有更高的优先级，因此它仍然是当前运行线程。calcThread线程是处于可运行状态的，但是在等待CPU。

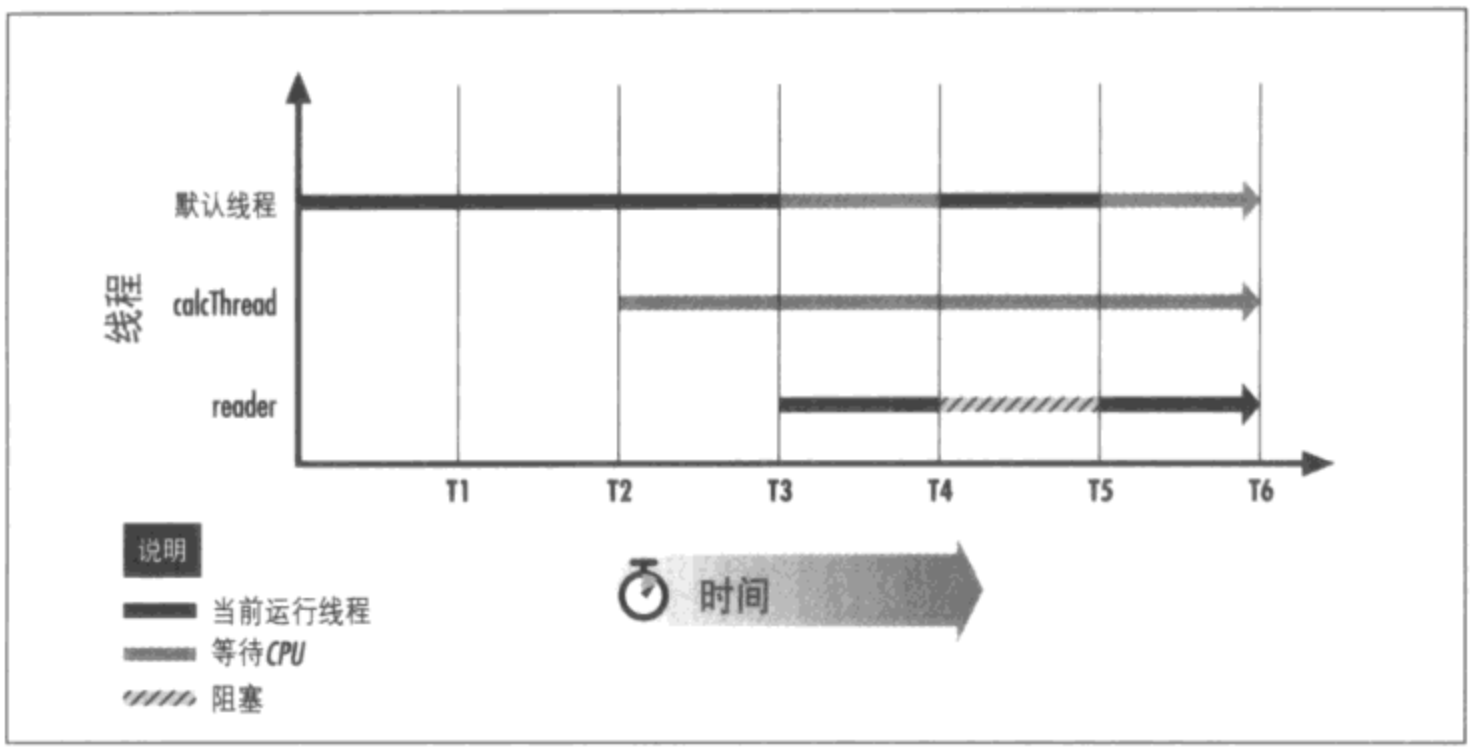

图 6-1：线程状态图

默认线程依旧执行：它创建reader线程，将它的优先级设定为6，然后调用该线程的start()方法。当默认线程调用了reader线程的start()方法后，reader线程就进入了可运行状态。因为reader线程的优先级高于默认线程，所以reader线程就成为了当前运行线程（这使得默认线程不再运行了，尽管它是处于可运行状态的）。这个改变在图的T3处表示。此时reader线程就在套接字上执行readChar()方法。但是如果没有数据可以读取，reader线程就进入阻塞状态（如图中T4处所示）。当这种情况发生时，默认线程就从原来被中断处开始继续执行（实际上，默认线程完全没有意识到它曾经被中断过）。默认线程一直作为当前运行线程运行，直到readChar()方法可以从套接字中读取数据。当套接字中有数据到来时（在T5时刻），Java虚拟机将reader线程的状态改变为可运行状态。当Java虚拟机改变了reader线程的状态后，它发现该线程的优先级是所有处于可运行状态的线程中最高的，于是它会中断默认线程并且将reader线程作为当前运行线程。

同时，calcThread一直耐心地等待运行的时机，它只有等到默认线程以及reader线程都阻塞或者是退出（或者是其他的线程提高了calcThread的优先级）时才能运行。calcThread是很难成为当前运行线程的，这也被称为CPU饥饿。通常，因为Java虚拟机从来都不会因为某个线程缺少运行的机会而调整它的优先级（尽

管有些底层操作系统会这样做），所以Java程序员要负责保证程序中的每一个线程都不会产生CPU饥饿。

调度具有同样优先级的线程

在大多数Java程序中，很多线程都具有同样的优先级值，因此我们来详细讨论一下这种情况。我们关注的是在Java虚拟机中的概念层面上到底发生了什么。要再次说明的是，我们只想提供Java虚拟机中线程调度系统是如何工作的一个例子，而不是提供一个特定的Java虚拟机中如何实现调度线程的蓝图。

我们可以设想Java虚拟机通过链表来跟踪Java程序中的所有线程；Java虚拟机中的每一个线程都处于一个表示其状态的链表中。因为线程的优先级可以有11级，因此我们可以设想有14个链表：一个链表用来存放所有处于初始状态的线程，一个链表用来存放所有处于阻塞状态的线程，一个链表用来存放所有处于退出状态的线程，另外对于每一个优先级都有一个链表。在某个特定优先级链表上的线程都是处于可运行状态的：一个具有优先级7的处于可运行状态的线程处于优先级为7的链表上，但当该线程被阻塞后，就会被移动到阻塞链表中去。

为了简单起见，假设这些线程是在一个有序链表上的；实际上，它们可能存放在简单的池中。将这些线程放在一个链表中暗示了它们在链表中的顺序就是它们被选择成为当前运行线程的顺序。虽然这样做可以帮助我们理解选取当前运行线程的过程，但是这并不表示该方式就是某个实现使用的方法。

现在让我们再来看看上一个例子，这次我们将calcThread的优先级改为和默认线程的优先级一样。如果这两个线程具有相同的优先级，则我们的状态图就可能像图6-2一样。这次，我们从T2时刻开始讨论，因为从此时开始，事情才变得有趣起来。

现在的区别就在于默认线程和calcThread具有相同的优先级，因此当reader线程被阻塞时，Java虚拟机会以和原来不一样的方式来选择当前运行线程。在该例中，我们只考虑3个Java内部链表：优先级为5的线程链表（默认线程和

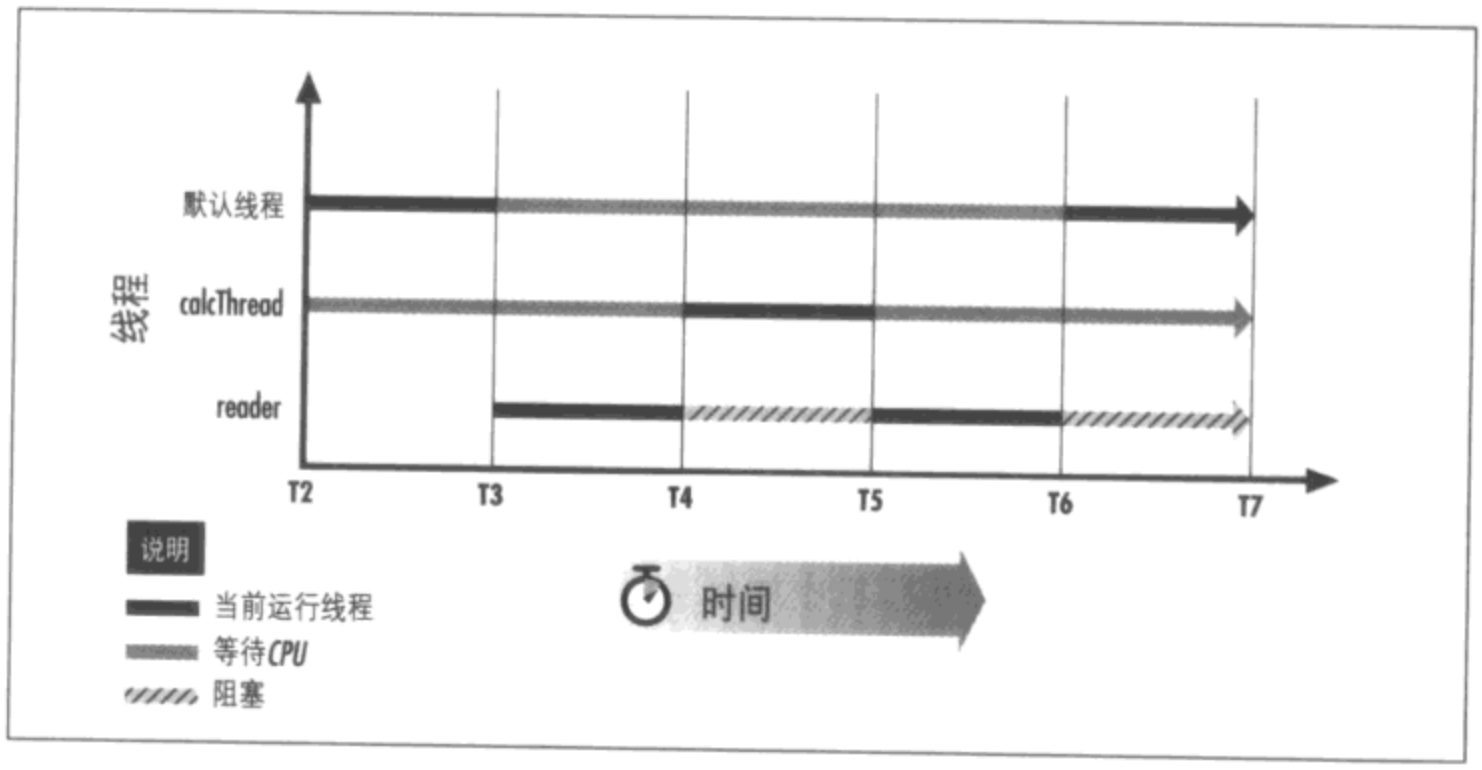

图 6-2: 具有相同优先级的线程的运行状态图

calcThread)、优先级为 6 的线程链表（reader）以及阻塞链表。在 Java 虚拟机进入 T2 时刻时，当 calcThread 启动后，这些链表的状态如下所示（注 1）:

```
优先级 5:  默认线程 -> calcThread -> NULL
优先级 6:  NULL
   阻塞:  NULL
```

因为默认线程是非空并且优先级最高的链表上的第一个元素，因此Java虚拟机会选择它作为当前运行线程。Java 虚拟机也会在退出T2时改变优先级为5的链表，如下所示:

```
优先级 5:  calcThread -> 默认线程 -> NULL
```

在时刻T3，默认线程启动reader线程，reader线程会抢占默认线程。因此Java虚拟机的内部链表如下所示:

```
优先级 5:  默认->线程 calcThread -> NULL
优先级 6:  reader -> NULL
   阻塞:  NULL
```

注 1: 在这些数据图中，当前运行线程总是非空且优先级最高的链表中的最后一个线程：当位于链表头部的线程被选择作为当前运行线程时，它便被移到链表的尾部。

在T4时刻，当reader线程进入阻塞状态时，Java虚拟机搜索非空的链表，发现优先级为5的链表非空。因此该链表的第一个元素（calcThread）成为了当前运行线程，并同时将它移到该链表的尾部。因此在离开T4时刻时，内部链表如下所示：

```
优先级 5: 默认线程 -> calcThread -> NULL
优先级 6: NULL
    阻塞: reader -> NULL
```

让我们继续：每当reader线程进入阻塞状态时，默认线程和calcThread就会改变它们在优先级为5的链表中的位置，从而轮流成为当前运行线程。

在该例子中，我们认为如果一个线程成为了当前运行线程，就会移到链表的尾部。这就导致了当reader线程被阻塞时，优先级为5的链表中的不同线程会成为当前运行线程。尽管这是到目前为止Java虚拟机中最常用的实现方法，但并不是所有虚拟机都是这样做的：某个实时系统中的线程在被中断后是不会被重新排序的。在那种实现（以及类似的实现）中，calcThread和reader线程会交替运行，而默认线程成为了饥饿线程。

优先级倒置和优先级继承

在一个典型的以优先级为基础的线程系统中，当一个线程试图去获取一个被低优先级线程拥有的锁时，会有不寻常的事情发生：因为高优先级的线程被阻塞，使得它暂时以低优先级来运行。例如当一个优先级为8的线程试图去获取一个被优先级为2的线程所拥有的锁时，因为该线程是在等待这个优先级为2的线程释放该锁，因此它会以优先级2来运行。这就是优先级倒置。

优先级倒置经常通过优先级继承来解决。通过优先级继承，如果一个高优先级的线程希望获取一个低优先级的线程所拥有的锁，则该低优先级的线程就会暂时提高它的优先级；它的优先级会和那个高优先级的线程一样。当该线程释放锁时，它的优先级就会降到原来的级别上。

让我们来看一个例子。如果我们有三个线程：线程2、线程5和线程8，它们的优先级分别是2、5和8。我们假设线程2是当前线程而其他两个都是阻塞的。

```
优先级 2: 线程2 -> NULL
优先级 5: NULL
优先级 8: NULL
    阻塞: 线程5 -> 线程8 -> NULL
```

此时，线程2获取了一个锁，因为没有其他线程试图来获取该锁，它的优先级不会改变。现在，假设线程5不阻塞了；它会成为当前运行线程:

```
优先级 2: 线程2 -> NULL
优先级 5: 线程5 -> NULL
优先级 8: NULL
    阻塞: 线程8 -> NULL
```

当线程8不阻塞时它就成为了当前运行线程。如果它试图获取由线程2拥有的锁，它会再次阻塞，但是线程2的优先级会被调整为8:

```
优先级 2: NULL
优先级 5: 线程5 -> NULL
优先级 8: 线程2 -> NULL
    阻塞: 线程8 -> NULL
```

因此尽管线程2在程序中设定的优先级比其他可运行线程的优先级要低，但还是会被选择成为当前运行线程。当线程2释放该锁后，它的优先级会被改回为2，同时线程8会从阻塞状态退出（因为线程8是在等待该锁):

```
优先级 2: 线程2 -> NULL
优先级 5: 线程5 -> NULL
优先级 8: 线程8 -> NULL
    阻塞: NULL
```

当然了，我们可以预见到线程8会成为当前运行线程。

使用优先级继承技术是为了使得高优先级的线程能够尽快地运行结束。如果在上面的例子中没有使用优先级继承技术，则线程8不得不等待线程5成为阻塞状态或者运行结束，然后线程2能够获得运行权来释放该锁。这会使得线程8暂时和线程2的优先级一样。优先级继承技术更加有利于高优先级线程的运行。

优先级继承是一种普遍使用的技术，但并不是Java虚拟机必须具有的特性。

循环调度

在上面的例子中，calcThread和默认线程在没有reader线程干涉的情况下不会改变它们在优先级链表中的位置。也就是说，calcThread从来都不会抢占默认线程，反之亦然。这可能会使很多人感到迷惑，他们会认为一个抢占式多线程系统也意味着同样优先级的线程应该是分时间片运行的，也就是说它们会周期性地互相抢占。

具有同样优先级的线程互相抢占的情况称为循环调度(round-robin scheduling)，这也是Java虚拟机中最受争议的一个部分。在Java语言规范中并没有要求虚拟机使用循环调度，但是也没有禁止这样做。有些Java虚拟机因为操作系统的原因而使用了循环调度，但是也有很多虚拟机，特别是那些非Windows平台上的虚拟机，并没有使用这种调度算法。

这使得我们的讨论中出现了一个不确定的因素。在一个使用了循环调度的平台上，具有同样优先级的线程会周期性地将控制权交给其他的线程。这个过程同样遵循我们前面说到的概念：被选中的线程同时也被移动到它所在的优先级链表的尾部。而循环调度算法意味着系统会周期性地中断当前运行线程，而选择同一个优先级链表中的下一个线程作为当前运行线程。而在一个没有使用循环调度算法的平台上，当前运行线程直到其被阻塞住或者有更高优先级线程要运行以前，都是一直在运行的。

线程模型

是否使用优先级继承和是否在具有同样优先级值的线程间进行循环调度，是不同Java虚拟机实现中线程调度之间的一个不同点。正是因为Java语言规范中对于线程调度说之甚少，而且不同的实现也要借助主机平台来对Java线程提供支持，才产生了这样的差异。

在Java的初期，基于优先级的线程调度被认为是绝对的：最高优先级的线程就一定是当前运行线程。许多Java编程书籍（包括本书的第一版）都是基于这个假设来讨论多线程编程的。但是新的Java规范是这样描述线程调度的：“具有高优先

级的线程一般而言会比低优先级的线程更容易得到CPU，但并不能保证一个高优先级的线程一定会运行，同时也不能依赖线程优先级来实现互斥（注2）。”

这个说明是对真实世界的工作方式的承认。有些操作系统不能确切地断定一个被阻塞线程转换为可运行状态的时间，因此，在一个高优先级的线程由阻塞状态转换为可运行状态和它真正成为当前运行线程之间会有一小段时间。实际上，因为你不能绝对地预计一个线程何时变成非阻塞状态，因此这个时间差在大多数情况下是不重要的。如果在数据到达套接字端口和在该端口读取数据的线程成为非阻塞状态之间有一点微小的延迟，程序是不会觉察的；它会简单地假设对该数据有一点点延迟而已。毕竟，Java不是一个实时操作系统。

更重要的是，许多Java虚拟机的实现都是由操作系统来直接调度Java线程，而不是由虚拟机自己来进行调度的。虽然操作系统一般都遵循我们上面所说的基于优先级的调度策略，但是它们一般会给线程调度增加额外的复杂性，而这对我们上面说过的一些基本概念是有影响的。

因此，想了解Java线程最终是如何被调度的，就需要理解所涉及的特定虚拟机。Java虚拟机可以分成两个基本的类别：

绿色线程（green-thread）模型

在这种模型中，线程是由虚拟机来调度的。它们大多数都严格遵循我们上面说过的以优先级为基础的理想化调度策略。

本地线程（native-thread）模型

在这种模型中，线程是由运行虚拟机的操作系统进行调度的。尽管它们都遵循我们在此讨论的调度策略，但是因为操作系统的不同，这种模型导致了Java线程调度之间的很多细微差别。

在本章的后面，我们将讨论在不同的平台上这些线程模型的实现，并从线程调度的角度来看看这些实现之间的细微差别。

注2：　参见《Java Language Specification》，P.415（Addison-Wesley，1996）。

何时调度是重要的

线程调度的细节是很深奥的，但好消息是：在大多数情况下，本章所讨论的调度细节对Java程序没有什么实际上的影响。无论使用哪种操作系统和线程库，这一点对于多线程程序而言都是真的，而对Java编程来说更是如此。

在Java程序中，经常是因为程序可能要调用一个会被阻塞的方法而创建线程：如对于一个慢速的输入流（例如SocketInputStream）调用read()方法，用Thread.sleep()方法来模拟一个周期性的定时器，或者调用wait()方法来等待一个特定的事件。这就导致了Java虚拟机中的线程经常在阻塞和可运行状态之间变迁。同时，只要Java虚拟机中的每一个线程都发生周期性的阻塞，它们就会得到运行的机会：每一个线程成为当前运行线程，进入阻塞状态，退出阻塞状态并被插入其所在优先级链表的尾部，它会和其他线程经过同样的周期而慢慢在链表中向前运动。

即使是在虚拟机中的所有线程都不是周期性阻塞的情况下，也可以忽略调度的问题。Java程序是用来完成一个特定任务的，并且能够完成该任务才是最重要的。一个负责计算并且显示四个连续GIF图像的Java程序必须等到四个图像全部计算完了才能够显示最终的图像。如果将每个图像都放在一个单独的线程中进行处理，则会方便得多。但是不论这四个线程是顺序调度还是采用某种循环调度算法，这些线程计算全部图像所用的时间都是一样的。当一个Java程序被分割成多个子任务，并且每一个子任务都是用一个单独的线程来完成时，因为我们最终关心的是能否完成任务，所以我们经常忽略这些线程的调度过程。

那么在什么情况下，我们关心线程的调度机制呢？就是当我们关注的不是上面那些通常的情况时，特别是当：

- 程序中有一个或者多个占用CPU较多的线程。
- 对计算的中间结果感兴趣时（例如，如果我们希望尽快看见那四个图像中的一个）。
- 线程之间不是合作完成一个任务；它们希望公平地完成单独的任务：例如可

能希望被循环调度（例如，一个完成多个不同用户请求的服务程序）或者期望顺序调度（例如，使用遵循先来先服务原则的服务器程序）。

当我们讨论用不同的机制来实现这些要求时，会对上面的这些情况进行更加深入的探究。

循环调度和公平性

具有同样优先级的Java线程并不是自动地由循环调度算法来实行时间片调度的，这会让许多程序员感到吃惊。这种吃惊的部分原因在于，人们习惯认为一个程序中的线程在理论上应该是平等地在操作系统中运行的：长时间以来，人们根深蒂固地认为时间片调度对于多进程是最公平的。在一个多用户的交互环境中，这是对的。但是，在某些场合中，循环调度并不是最公平的算法，而且程序员也要求保证线程间不会发生时间片调度。设想一个计算服务器的例子，它同时接收从不同客户端发出的连接请求，并且对每一个连接都使用一个新的线程。这是一个很完美的服务器体系结构。但是对于这种体系结构，什么是最好的调度机制其实是一个很困难的问题。

假设CalcServer对于每一个客户的连接都会进行某种复杂的分析计算工作；同样假设这个计算对于每一个客户要耗时5秒钟。当有五个用户同时连接到服务器上时，CalcServer就会创建五个单独的线程。如果这些线程是分时间片的，它们要使用25秒来完成所有的计算工作，因为这五个线程是被平等对待的，我们可以认为这五个线程都是大概在第25秒后才完成它们各自的计算任务的。因此每一个客户都是在25秒后才得到其计算结果的。如果我们的CalcServer不使用循环调度算法，就会有不同的结果：客户还是在同一时刻连接上服务器，但是某个客户（可能是任意选择的一个）得到了完成其计算工作的机会；因此第一个客户在第5秒就得到了结果，而不是第25秒。接着第二个客户的计算开始了；第二个客户会在第10秒得到结果，依次类推。只有第五个客户会在第25秒得到运算结果。

到底上面的哪个调度机制更合理些？答案依赖于在这5秒钟内服务器的行为。如果仅仅是输出一个结果，很显然非时间片调度算法更公平：平均而言，每一个客户要等待15秒钟来得到结果而不是时间片算法中的25秒钟。但是如果服务器在

每一秒钟的计算中都会给客户一个结果的话，则时间片算法更公平：每一个客户在5秒钟后都有了一个结果，而在非时间片算法中，第五个客户到了第21秒钟才会得到第一个结果。

换句话说，对于“显示中间结果”这样的要求而言，循环调度算法提供了公平的结果。但是如果我们所关心的是最终的计算答案，那么在一个单CPU的机器上循环调度算法并不合适：即使在最好的情况下，它也没有体现出什么优越性，而对于CalcServer这样的例子而言，它实际上是降低了系统的吞吐量。

对于一个有多个CPU的系统而言，情况就更加复杂了。如果在一个有四个CPU的机器上运行五个线程，假设系统不是使用循环调度算法，客户得到结果的平均时间是6秒钟：头四个将在第5秒钟得到结果，而最后一个会在第10秒钟得到结果。另一方面，如果使用循环调度算法，平均响应时间是6.2秒钟。但是，每一个回答的时间都非常接近于6.2秒钟：实际上，我们可以认为每一个客户都会在6.2秒钟后得到结果。因此，尽管循环调度算法的平均计算时间稍微多了一点，但是它提供了更高的公平性。而且在这个例子中，如果我们最关心的是何时得到这五个线程的最终结果，则循环调度算法更快：6.2秒对10秒。

调度和线程优先级

让我们开始深入探讨影响线程调度的编程技术；我们将开始了解如何控制Java线程的优先级。这是Java程序员能够最有效地影响线程调度的机制；当然，对线程优先级的调整应该符合我们对于程序行为的要求。

Java API中与线程优先级有关的调用

在Java的Thread类中，有三个静态的最终变量用来定义线程优先级的允许范围：

Thread.MIN_PRIORITY

最小的线程优先级。

Thread.MAX_PRIORITY

最大的线程优先级。

Thread.NORM_PRIORITY

Java 解释程序中定义的默认优先级。

每一个线程都有一个处于MIN_PRIORITY（1）和MAX_PRIORITY（10）之间的优先级值。但是，不是所有的线程都可以拥有这个范围里的每一个值：每一个线程都是属于一个线程组的，而每一个线程组都有其最大优先级（小于或者等于MAX_PRIORITY），这个值是该组中线程的最大优先级值。我们将在第十章讨论线程组。同时我们还应当知道，applet 中的线程的优先级值最大为 NORM_PRIORITY + 1。另外，虚拟机还可以创建优先级为 0 的内部线程。因此对于一个虚拟机而言，有效的线程优先级级别是 11 级。

符号化的线程优先级值

使用符号化的优先级常量并不一定是有用的。我们通常认为，这些基于符号名字的常量值可以使人们认为它们的实际值是不相关的。使用符号名字还可以允许我们改变变量并且让这种改变影响到全部代码。

不幸的是，对于线程优先级而言，这种逻辑是不适用的：如果要控制线程的每一个单独的优先级，那么我们有时候必须知道这些值的实际范围是什么。如果最大和最小优先级的范围是 20，我们可以有 20 个在不同优先级上的线程。但是，如果范围是 5，则这 20 个线程就必须共享这些优先级了（平均而言，每一个优先级级别上有 4 个线程）。因此只知道这些常量的存在是不够的；我们还要知道Java线程的最小优先级是1，最大优先级是10（对于applet而言是6），默认优先级是5。

使用本地线程库的虚拟机使得这个问题更加复杂，因为后台的操作系统可能不能支持十个不同的线程优先级，这意味着实际上，不同的Java线程优先级会映射到操作系统上的同一个线程优先级上。

默认的线程优先级就是线程创建时的优先级。它一般应该是（但也不一定）：NORM_PRIORITY（5）。

在 Java 的 Thread 类中有两个和线程优先级相关的方法：

void setPriority(int priority)

设置线程的优先级。如果该值超过了允许的范围，则抛出一个异常。但是如果该值是在允许的范围以内但是又超过了该线程所在的线程组所允许的最大值，则设定的优先级为允许的最大值。

int getPriority()

获取给定线程的优先级。

使用与优先级有关的调用

让我们看一个使用这些调用的例子。通常，通过设置线程的优先级就可以得到所期望的调度结果。假设你的程序中有两个线程，其中一个经常会被阻塞，你只需将这个线程的优先级设为比另外一个高一些来阻止其发生CPU饥饿。我们使用如下的代码来说明这一点，下面这个例子进行计算并显示不规则的图像。对于不规则图像的计算是一个占用CPU较多的任务，但是对于不规则图像的每一块都可以分别进行计算并显示。因此我们将这个计算过程放到一个单独的低优先级的线程中去。在计算完了图像的每一块后，通过调用repaint()方法来绘制图像。同时，applet的初始线程大部分时间都在等待用户的输入事件或者是重画事件的发生。

下面是applet的框架代码：

```
import java.applet.*;
import java.awt.*;

public class Fractal extends Applet implements Runnable {
    Thread calcThread;
    boolean sectionsToCalculate;
    static int nSections = 10;

    public void start() {
        Thread current = Thread.currentThread();
        calcThread = new Thread(this);
        calcThread.setPriority(current.getPriority() - 1);
        calcThread.start();
    }

    public void stop() {
        sectionsToCalculate = false;
    }
```

```
    void doCalc(int i) {
        // 计算不规则图像的第 i 部分
    }

    public void run() {
        for (int i = 0; i < nSections && sectionsToCalculate; i++) {
            doCalc(i);
            repaint();
        }
    }

    public void paint(Graphics g) {
        // 绘制计算后的结果
    }
}
```

如果我们不将计算线程的优先级降低会有什么结果？在该例中，applet经过了其init()和start()方法后，就在系统中有了两个具有NORM_PRIORITY优先级的线程：applet的主线程和计算线程。因为applet的主线程阻塞在等待事件上，计算线程就是惟一的可运行线程，因此也就成了当前运行线程。计算线程完成了对于不规则图像中某一块图像的计算，然后调用repaint()方法。这会创建一个将applet的主线程唤醒的事件，使得applet的主线程转换为可运行状态。

但是，计算线程仍然处于可运行状态。这就使得计算线程仍然是当前运行线程。applet的主线程仅仅是加在NORM_PRIORITY链表的尾部，如果我们的Java虚拟机不是使用循环调度算法的话，计算线程就会一直是当前运行线程。尽管不规则图像的好多块都被计算出来了，但因为applet不可能得到成为当前运行线程的机会来重画屏幕，所以对于repaint()的调用没有任何效果。

但是，如果我们将计算线程的优先级设定为低于applet线程的优先级，则当计算线程调用repaint()方法时，因为applet的主线程具有高的优先级而且是处于可运行状态，它就成为了当前运行线程。applet执行paint()方法然后进入阻塞状态，使得计算线程再次成为当前运行线程。

如果applet在计算图像时，用户要与其进行交互，这个技术也是重要的。如果计算线程和applet的主线程具有同样的优先级，则当计算线程运行时，applet不能对用户的输入进行响应。

何时使用这些基于优先级的调用

在什么情况下使用这些设置特定线程优先级的技术是恰当的？当下面两个条件都成立时，应该使用该技术：

- 有且只有一个占用CPU较多的线程（或者目标机器的每一个CPU上都有一个这样的线程）。
- 对于用户而言，可以得到中间结果是有意义的。

对于上面的例子而言，很明显，这两点都是满足的：有一个计算不规则图像部分的线程，而且不规则图像每一个部分的显示都可以认为是中间结果。通过连续的修改可以更好地表现数学模型。

中间结果对于用户而言很重要的另外一个重要领域是加载图像：当部分图像可以显示时，可以只显示这部分，使用户可以在屏幕上看到图像慢慢展开。但是要记住：一般情况下，Java程序是通过网络来加载图像的，这意味着读取图像的线程经常会被阻塞，因此没有必要来调整任何线程的优先级。但是如果Java程序是通过计算来获取图像的，降低该线程的优先级会是个好主意。

如果有多于一个的占用CPU较多的线程，会出现什么结果呢？在上面的不规则图像例子中，如果我们对于图像的每一部分都创建一个线程来进行计算会有什么结果？从程序的角度看是优美的，但是也存在危险。如果你的程序中有多个占用CPU较多的线程，你应当将每个占用CPU较多线程的优先级都降低。在上面的例子中，只有在每一个计算线程的优先级都低于applet线程的优先级时，才会得到你期望的部分结果。

在对于具有同样优先级的线程使用循环调度算法的平台上，占用CPU较多的线程会互相竞争CPU，这使得对于不规则图像的每一个部分分别进行计算需要的时间要长一些。这也意味着显示不规则图像的每个部分（也就是显示中间结果）比仅仅使用一个单独的计算线程的情况要慢。

另外一方面，如果程序中占用CPU较多的线程的个数和处理器个数相同的话，使用这个技术就可以最大可能地利用机器的资源并最快地得到中间结果。

常见的调度实现

我们来看看这些技术在多个通用平台上的Java虚拟机实现中是如何发挥作用的。Java是与平台无关的语言，提供和平台相关的实现细节会违反这一规则，因此我们通常没有必要来说明这些地方。但是有时候这些实现的细节是很重要的。

另外一方面，真实世界中的Java有一个特点，就是不同的Java虚拟机厂商是互相竞争的：比赛谁最快，谁能够运行更多的线程等等。只要一个虚拟机的实现遵循Java语言的规范并且通过了Java兼容包的测试，就是一个有效的Java虚拟机了。由于Java规范中对于线程调度的灵活性，我们将要讨论的所有实现都是有效的Java虚拟机（至少在线程调度支持方面）。

绿色线程

第一个模型是最简单的。在这种模型中，操作系统根本就不知道线程的存在，由虚拟机处理线程API的细节。从操作系统的角度看，只有一个进程和一个线程。该模型中的每一个线程都是虚拟机中的一个抽象表示：虚拟机必须处理与该线程相关的线程对象的所有信息。这包括线程的堆栈、用来表示线程正在运行的Java指令的程序计数器，以及其他与线程相关的统计信息。虚拟机将这些信息装载到内存中并且操作它们：通过程序计数器来运行指令，获取下一条指令再来执行这样的循环。

当虚拟机要运行另外一个线程时，它将当前运行线程的状态信息都保存起来，然后用目标线程的相关信息进行替换。目标线程的堆栈成为虚拟机要操作的堆栈，虚拟机要运行的下一条指令将会是目标线程的程序计数器指示的指令。

当然，所有这些都发生在逻辑层面上：特定的虚拟机实现可能会有些不同。但是这个模型最突出的一个特点就是操作系统不知道虚拟机在此时进行线程切换。对于操作系统而言，虚拟机仅仅是在执行代码而已；在虚拟机外，这些不同的线程都是不可见的。

在Java中，这种模型被称为绿色线程模型。“绿色”这个词并没有什么特别的意义的——并不是表示这种线程模型比起其他的模型而言是不成熟的（没有其他模型健壮或有用）。有时候，因为线程是仅仅存在于用户级别的应用程序中的，而不需要进行系统调用来处理线程的细节，所以这些线程又被称为用户级线程(user-level thread)。在Solaris1（SunOS 4.1.3）中，这种线程模型被称为*lwp*。但是它与Solaris 2中的LWP模型是不一样的，我们将会在后面讨论LWP模型。

用户级别和系统级别线程

在大多数现代操作系统中，操作系统在逻辑上都可以被分成两块：用户级和系统级。操作系统本身（即操作系统内核）是系统级的。内核是负责处理用户级别的系统调用的。

例如，如果一个运行在用户级别的程序想读取一个文件，它就必须调用（或者陷入）操作系统的内核，而内核会读取文件并且将数据返回给用户程序。这样划分的好处是保证系统更加可靠：一个进行了非法操作的程序可以被系统终止而不会影响其他的程序或者内核。仅仅当内核执行了非法操作时才可能导致整个机器崩溃。

正是因为这种划分，才有可能在用户级、系统级或者是两个级别都对线程提供支持。

因为这种模型并不依赖于操作系统提供线程专有的功能，所以绿色线程是完全可移植的。实际上，尽管线程模型需要一些用汇编语言写的代码（例如，某些代码必须能够在CPU上执行原子性的检查和设置指令），但是该模型本身是很容易移植的。通常只有在汇编代码中才能访问这个指令。不过，虽然线程库本身是可以移植的，但是使用绿色线程意味着虚拟机的其他实现细节会更加复杂：例如，虚拟机必须以非阻塞的方式处理所有的I/O操作。这使得编写虚拟机的工作难度更大。

因为绿色线程模型比起我们将要介绍的其他模型而言要更容易移植，所以它一直被作为Java参考实现的标准。实际上，将这个模型移植到大多数操作系统上都不是一件困难的事。人们经常因为Window 3.1缺乏对线程的支持而认为将Java移植到Windows 3.1上是很困难的。但实际上，在Windows 3.1上有着许多用户级

的线程库，因而将绿色线程库移植到该平台上也是容易的。而其他的移植问题，例如缺少对32位的支持和对AWT的移植，仍然是难以克服的。

尽管Unix平台经常也支持本地线程模型，但是绿色线程模型在大多数Unix平台上也很常见。在Unix平台上启用Java的浏览器通常都是使用绿色线程模型的（虽然Java插件可能使用其他模型）。

因为操作系统不知道绿色模型中的线程，所以即使在一个多CPU的机器上，用绿色线程模型实现的Java虚拟机在同一时刻也只能运行一个线程。

绿色线程的调度

大多数情况下，绿色线程的调度和我们上面讨论的是一致的。大多数绿色线程的实现中都没有循环调度这个概念，因此绿色线程库不会自动进行分时间片调度。虚拟机负责全部的线程调度，而虚拟机仅仅在高优先级的线程成为运行状态或者是当前运行线程被阻塞时才进行线程切换。但是，这不是Java规范中的要求，并且绿色线程实现也可以通过包含一个定时器来实现循环调度。

作为绿色线程模型的参考实现中使用了优先级继承技术，这使得拥有被某个高优先级线程等待的锁的线程会暂时以高优先级运行。

根据操作系统的不同，绿色线程可能在调度上不够精确：当一个高优先级的线程期望运行时，低优先级的当前运行线程还可能继续运行一小段时间。

例如，假设一个Java线程试图通过套接字来读取数据，而端口上没有数据可以被读取。我们期望的结果是该线程会被阻塞——但是虚拟机自身不能被阻塞。因此，当有数据到达端口时，虚拟机应该通过某种方法得到通知，从而使得线程可以通过套接字来读取数据。在大多数操作系统中，可以通过使用异步I/O的方法来实现这一点：当套接字上有数据到来时，虚拟机会收到一个信号，该信号会中断虚拟机的当前运行。通过响应信号，虚拟机就可以运行那个期望通过套接字读取数据的线程了。

但是在某些操作系统上，是没有异步I/O机制的。在这种情况下，虚拟机就只能

通过周期性地检查套接字来确认数据是否到达。通常是每隔几毫秒才进行一次这样的轮询。当端口上有数据时，虚拟机就可以调度那个期望读取数据的线程。

现在，假设一个高优先级的线程在等待套接字上的数据，而同时一个低优先级的线程在执行某个计算任务。如果虚拟机是依赖于轮询套接字来检查是否有数据可以被读取的，那么在数据实际到来和虚拟机下一次检查该套接字之间就会有一个非常小的时间窗口。在这段时间内，尽管高优先级的线程应当是正在运行的线程，但低优先级的线程仍然是在运行的。

因为这个延迟是很短的，类似这样的情况可能不会影响实际程序。同时还存在好多其他的因素来影响程序：如果数据的等待时间大于它通过网络传输的时间会如何？如果同一个机器上的其他进程使得Java应用程序不能够保证每隔几毫秒就轮询一次会有什么结果？因为Java不是一个实时平台，所以我们刚刚讨论的问题是不会对 Java 程序运行造成什么影响的。

但是，在Java虚拟机中的某个线程使得另外一个线程成为非阻塞状态后，高优先级的线程仍然要经过一段时间后才能真正运行。看看下面的运行于低优先级线程中的代码片断：

```
public class LockTest {
    Object someObject = new Object();
    class ThreadA extends Thread {
        ThreadA() {
            setPriority(Thread.MAX_PRIORITY);
        }
        public void run() {
            synchronized(someObject) {
                someObject.wait();
            }
            someObject.methodA();
        }
    }
    class ThreadB extends Thread {
        ThreadB() {
            setPriority(Thread.NORM_PRIORITY);
        }
        public void run() {
            synchronized(someObject) {
                someObject.notify();
            }
```

```
                someObject.methodB();
            }
        }
        static void main(String args[]) {
            new ThreadA().start();
            new ThreadB().start();
        }
    }
```

在该例中，我们启动了两个线程：ThreadA的优先级是10，ThreadB具有优先级5。因为我们先启动ThreadA，我们希望它会开始运行其run()方法然后阻塞在其wait()方法上。接着ThreadB开始运行并且通知ThreadA。在一个严格按照优先级调度的模型中，ThreadA会被唤醒，抢占ThreadB的运行，执行methodA()方法。ThreadB会接着执行methodB()方法。虽然在大多数情况下methodA()是在methodB()前被调用的，但是我们不能保证绿色线程模型的优先级调度程序都是严格地按照这种顺序运行。

Windows本地线程

在Window 95和Window NT（更一般的说就是任何32位的Windows操作系统）上的本地线程模型中，操作系统完全看得见虚拟机里的每一个线程，同时Java线程是和操作系统里的线程一一对应的。所以，Java线程调度是由操作系统的底层线程调度决定的。

这种模型是容易理解的：我们可以将线程都理解成为进程。在这种情况下操作系统不区分线程和进程：它认为线程就是进程。当然，对于操作系统而言，线程和进程是有一些微小的区别的，但是这都不是调度程序所关心的。

因为在Windows操作系统上没有流行的Java虚拟机的绿色线程实现，所以Window平台上的所有虚拟机实际上都是使用本地Window线程模型。启用Java的浏览器也是使用这种模型。但是，因为Windows平台有太多的Java虚拟机厂，因此他们的细节都是不同的。

因为操作系统知道线程，Windows的本地线程可以看成是重量级的线程。如果系统中有太多的线程，会导致操作系统无法响应的问题，这限制了Windows平台上可并发运行的线程数目。在下一章我们将使用缓冲池技术来解决这个问题。

但是这种实现允许在机器的多个CPU上同时运行多个线程。每一个CPU将根据下面说明的原则来选择一个当前运行线程来运行:

Windows 本地线程的调度

在Windows本地线程模型中，操作系统在线程调度中扮演重要的角色。实际上，操作系统就如同调度进程一样对线程进行调度。这意味着线程调度是在抢占的、基于优先级的机制的基础上进行的。虽然这也是我们希望的，但是除了我们上面说明的通常原则以外，还有一些复杂的因素要考虑。

首先，Windows操作系统仅仅承认七个优先级级别；而这七个优先级级别要映射到Java线程的十一个优先级级别上去。因为Java线程中优先级0是保留给虚拟机内部的线程的，这就导致了可编程的十个优先级要映射到Windows平台上的六个优先级上。不同的虚拟机会以不同的方式做到这一点，但是一个完成映射的常见实现如表6-1所示:

表6-1: Win32平台上Java线程优先级的映射

Java优先级	Windows 95/NT优先级
0	THREAD_PRIORITY_IDLE
1 (Thread.MIN_PRIORITY)	THREAD_PRIORITY_LOWEST
2	THREAD_PRIORITY_LOWEST
3	THREAD_PRIORITY_BELOW_NORMAL
4	THREAD_PRIORITY_BELOW_NORMAL
5 (Thread.NORM_PRIORITY)	THREAD_PRIORITY_NORMAL
6	THREAD_PRIORITY_ABOVE_NORMAL
7	THREAD_PRIORITY_ABOVE_NORMAL
8	THREAD_PRIORITY_HIGHEST
9	THREAD_PRIORITY_HIGHEST
10 (Thread.MAX_PRIORITY)	THREAD_PRIORITY_TIME_CRITICAL

在这个实现中，一个具有优先级3的线程和一个具有优先级4的线程和两个有优先级4的线程是一样的。

除了七个优先级级别外，Windows操作系统还有五个调度类，因此在Windows中的线程实际上是根据优先级和调度类进行调度的。但是，因为线程的调度类是不容易被改变的，所以在一个优先级可以被程序员动态改变的系统中它们是不需要被考虑的。

在Windows平台上，另外一个复杂因素是：程序员设定给一个线程的优先级（七个平台专有的优先级之一）仅仅是操作系统用来决定线程的绝对优先级的因素之一。其他可以影响线程优先级的因素是：

- Windows本地线程会使用优先级继承技术。
- 线程的实际优先级是编程设定的优先级减去一个值，这个值反映该线程最近运行的时间。这个值是连续调整的：运行时间越长，该值就越接近于零。这种调整是主要针对具有同样优先级的线程，这实际上是在同样优先级线程间实现的循环调度算法。因此，在Windows平台上，具有同样优先级的线程是分时间片运行的：所有的事情都是平等的，具有同样优先级的线程得到大致相同的CPU运行时间。
- 在另一方面，一个长期没有得到运行机会的线程会暂时性地得到一个提高了的优先级。当线程得到运行机会后，这个提高值会随着该线程的运行时间而慢慢减小。这即可以防止出现线程饥饿的情况，又可以保证高优先级线程比低优先级线程得到更多的运行机会。这种优先级提高的效果依赖于最开始的初始线程优先级值：要和一个优先级 为5的线程竞争，优先级为3的线程就比优先级为1的线程能够得到更多的运行机会。

上面所说的这些仅仅表明，在Windows平台上很难保证线程的明确运行顺序。但是，因为操作系统保证了不会出现线程饥饿以及同样优先级的线程之间是分时间片运行的，通常这也就不是一个问题了。

Solaris本地线程

我们最后看到的线程模型也是最复杂的。在操作系统层面上看，Solaris本地线程提供了最为灵活的线程模型。但是因为Java API中没有利用这些特性的接口，所

以在Java程序中不能直接使用这些灵活性。当然，Java程序员可以通过本地方法调用的方式来直接调用本地线程库，从而获得那些Java API不提供的特性。有些时候，面对那些无可奈何的情况，我们不得不这样做。

Solaris本地线程提供两个级别的编程模型：一种是用户级线程库，它们对于操作系统而言是看不见的，它们的调度方式和我们在前面所说过的绿色线程模型一样。另外，还有系统级线程（也被称为轻量级进程 —— LWP），操作系统可以看见它们，而且也使用类似于我们刚刚讨论的Windows平台上线程模型的调度方法。这两个级别的线程模型之间的相互影响使得Solaris本地线程库具有很大的灵活性（以及复杂性）。

Sun公司的产品版虚拟机使用Solaris本地线程模型。从Java 2开始，Sun公司的参考版虚拟机开始使用Solaris本地线程模型。在写本书时，Solaris平台上的Netscape和Internet Explorer（版本4.0及以前版本）都是使用绿色线程模型，不知道在当两个浏览器使用与Java 2兼容的虚拟机时是否会使用本地线程模型。HotJava可以在两种模型中的任意一个上运行。

如果使用本地线程模型，因为操作系统知道这些线程，所以Java程序就可以在一个多CPU的机器上同时运行多个线程。

Solaris本地线程的调度

Solaris本地线程的调度有点复杂，但是还是遵循我们已经讨论过的原则。从每一个LWP的角度看，调度都是遵循绿色线程模型的。在任何时刻，每个LWP都选择具有最高优先级的线程运行。LWP不会在可选的线程间进行分时间片调度：这一点和绿色线程模型一样，一旦LWP选择一个线程来运行，它就一直运行它到一个更高优先级的线程出现为止。但是在如何处理线程阻塞的问题上，LWP和绿色线程是不同的，在后面会看到这一点。LWP可以看到的线程和Java程序中的线程之间有着一对一的映射关系，因此，在只有一个LWP的Java虚拟机中，调度非常类似于绿色线程模型。

但是，使用Solaris本地线程模型的程序一般是使用多个LWP的，每一个LWP都是由操作系统进行调度的。尽管LWP自己也有优先级，但是这个优先级是Java

程序看不见的，同时也不受LWP选择运行的Java线程优先级的影响。因此虚拟机中的所有LWP在本质上都有着同样的优先级，并且该优先级会根据该LWP最近运行的时间来进行调整（类似于Windows平台上的优先级调整）。因此，在LWP之间是分时间片运行的。

当一个LWP运行时，它是选择一个最高优先级的线程运行。当该LWP最终用完它的时间片后，另外一个LWP会开始运行。这个LWP在剩下的线程中选择一个最高优先级的线程运行。这个过程会在该虚拟机的整个生存期内持续进行。

假设一个使用两个LWP的虚拟机上创建了两个线程，并且每一个的优先级都是5。当第一个LWP开始运行时，它随机选择一个线程开始运行。当最终该LWP使用完其时间片后，第二个LWP开始运行了，并且选择剩下的那个线程开始运行。当这个LWP也用完其时间片后，第一个LWP又开始运行，因而第一个线程也就开始运行了。因为每一个LWP是通过操作系统进行分时间片的，所以这两个线程也就是分时间片运行的。

好，现在假设仅仅有两个LWP但是有三个具有同样优先级的线程。在该例中，前两个线程是分时间片的，而第三个线程则根本不会运行（至少是在前两个中的一个被阻塞或者运行结束以前）。同样，这和我们在前面学到的调度模型是一致的：只有其他具有同样或者更高优先级的线程都阻塞了，线程才能获得运行权。一个LWP一旦选择了一个线程，就会在该线程阻塞或者具有更高优先级的线程出现前一直运行该线程。

现在，我们假设有两个线程和两个LWP。但是这次一个线程的优先级是5，而另一个是4。有趣的是，这两个线程之间也是分时间片运行的。优先级为4的线程在优先级为5的线程没有阻塞的情况下还是会运行的。当第一个LWP开始运行时，它选择优先级为5的线程来运行。当第二个LWP开始运行时，它必须要选择一个线程，因为优先级为5的线程已经被第一个LWP所占有，所以此时只有优先级为4的线程可供选择。因此，尽管优先级为5的线程没有阻塞，第二个LWP还是开始运行优先级为4的线程。在该例中，这两个线程是分时间片的。

我们上面的讲述并没有暗示因为优先级为5的线程在一个特定的LWP中运行了，它就永远限制在该LWP中了。这种情况（也被称为束缚线程）可能在Solaris程

序中存在，但是在大多数的Java虚拟机实现中都是使用非束缚线程技术的。优先级为5的线程可能会被其LWP用一个高优先级的线程替换掉，而自己再次进入等待运行线程的池中。当运行优先级为4的LWP开始再次运行时，因为优先级为5的线程不属于任何一个LWP，它会用优先级为5的线程来置换它运行的优先级为4的线程。

根据我们的经验，如果Java虚拟机中的LWP数目为N，则该虚拟机中具有最高优先级的N个线程将会分时间片运行（即使这些线程的优先级不同）。

最后，我们还要提到的是，Solaris本地线程是使用优先级继承技术的。

虚拟机中的LWP

一个虚拟机中可以有多少个LWP？答案依赖于我们将要讨论的一些准则而不是一个确定的值。为了回答这个问题，我们必须稍微深入探究一下Solaris线程库。

Solaris线程模型严格遵循Java线程API。在线程库本身中，是有线程优先级概念的。这意味着对于特定的LWP，Solaris线程库是使用基于优先级的调度算法来进行实际的线程调度的。对于程序员而言，这种调度模式和虚拟机使用绿色线程模型时是一样的。

Solaris线程库是根据如下的指导方针来控制LWP个数的。

- 虚拟机开始只有一个LWP。当虚拟机创建线程时，这些线程都是运行在这个LWP上的。该LWP根据基于优先级的调度算法来选择一个优先级最高的线程在其上运行。
- 当线程执行一个系统调用时，它就暂时束缚在该LWP上。一个系统调用就是一个要使用内核来完成某项工作的调用。在Java中，这包括对大多数流进行读/写操作（除了基于字符串的流以外），以及创建套接字。一个束缚在LWP上的线程是不可能被抢占的。
- 如果从系统调用中很快返回，则LWP继续执行该线程。但是此时该线程是非束缚的，这也意味着如果有一个高优先级的线程被创建，LWP会运行那个高优先级的线程。

- 如果系统调用被阻塞了，则相应的LWP也阻塞。如果所有的LWP都阻塞，而此时有等待运行的线程，则操作系统会自动创建一个新的LWP。这个新的LWP会从等待运行的线程池中选择一个具有最高优先级的线程来运行。

Java虚拟机本身从不创建新的LWP；这项工作是由操作系统和Solaris线程库来完成的。因此Java虚拟机中的LWP数目就等于该虚拟机中曾经被同时阻塞的最大线程个数加上一。实际上，这也意味着一个典型的Java应用程序会启动五个到七个LWP，因为在Java虚拟机的初始化过程中，大致会有四个到六个LWP会同时阻塞。

在一个Java程序中，通常至少有两个LWP可用。其他那些LWP都会在Java虚拟机运行内部线程期间发生阻塞。因而，尽管你的程序中可能有两个不同优先级的线程，你还是可以依赖它们通过这两个LWP而分时间片运行。

在一个单处理器的机器上，这个数目的LWP是足够的。如果这些LWP不够用（也就是如果你创建的线程因为阻塞而耗尽全部LWP），则新的LWP会根据需要而被创建。如果Java程序中有一个或者多个非阻塞线程，则至少会有一个LWP来运行这些线程。

在一个多处理器机器上，这个数目的LWP就可能不够了。特别是当Java程序中的线程都是占用CPU较多的并且是很少阻塞时。如果在一个有八个处理器的机器上仅仅有两个LWP，而且该机器是作为计算服务器来设计的，则不管你创建多少个线程，都不可能得到所期望的结果。为了最大限度地利用这台机器的能力，你至少需要八个LWP —— 每一个CPU上都有一个，这样你才可能同时运行八个占用CPU较多的线程。

在这个例子中，你必须调用操作系统专有的库才能够使得操作系统创建八个（或者你所希望的）LWP。我们在本节的结束部分会通过一个例子来说明如何实现这一点的，但是这要通过本地方法来调用Solaris线程库中的thr_etconcurrency()方法来创建足够的LWP。从不同的角度看，这样做可能是非常复杂或者是非常酷的。

这看起来太复杂了。我们是否真的要这样做呢？可能不需要。如果使用一个单CPU的机器，这样做不会给你带来什么好处。实际上，因为过多的LWP会互相竞争CPU，还会稍微降低机器的性能。如果是一台多CPU的机器，仅仅当你的多个线程都会在非阻塞情况下进行长时间的运算时，才会带来性能提高。如果写的是一个聊天程序，则多个LWP不会对你有什么帮助：尽管此时每一个连接在服务器上客户的都有一个线程，但是这些线程在大多数时候都是处于阻塞状态（等待用户输入）。在这种情况下，线程库会创建足够多的LWP：足以包含所有同时被阻塞的线程；而当一个线程非阻塞时，它就会处在一个LWP中，从而就可以开始处理用户的请求了。

因此，除了编写包含多个占用CPU较多线程的程序外，我们是不需要担心可用的LWP个数的。

这听起来很酷，但是如何知道我们到底需要多少个LWP呢？这是一个很难回答的问题。假设你的线程很多，则答案是可能会同时被阻塞的线程个数再加上一个LWP来同时运行其他的线程。当机器上的CPU个数和同时运行的线程个数相同时我们会得到最大的吞吐量。如果LWP的个数过少，则CPU会无所事事。如果LWP个数过多，则LWP间会竞争CPU时间。

当然，机器也可能还要处理其他的事情，因此我们可能会希望使用比CPU个数少一点的LWP，使得其他的程序也可以得到足够的CPU时间。但是，使用比CPU个数多的LWP是不可能带来任何好处的。即使你的程序希望几百个线程进行分时间片运行，通过在程序中使用其他的调度算法来达到这一点比使用几百个LWP要好。使用LWP是要消耗一些系统资源的，而且LWP比线程更加对资源敏感的。Solaris上的Java虚拟机可以很轻易地处理几百个线程，但是却不能使用几百个LWP。

本地调度支持

到目前为止，我们对于Java线程API的介绍是不完全的，还没有包含一些对调度的高级用法。例如，还不能知道机器上有多少个CPU，也不能设定Solaris上Java

虚拟机中LWP的个数，或者通过设置处理器相似掩码（affinity mask）使得特定的线程在特定的进程上运行。不幸的是，要克服这些限制的惟一方法是在程序中使用本地方法调用。

在本节中，我们将简单介绍如何做到这一点。我们会给出一个完整的例子，但是对于Windows线程、Solaris或POSIX线程以及Java本地接口（JNI）的介绍则不在本书的范围之内。

我们通过设计一个可以完成如下三个操作的类开始：获取机器上的CPU个数、获取和设置我们期望虚拟机可以同时运行的线程个数：

```
public class CPUSupport {
    static boolean loaded = false;
    static {
        try {
            System.loadLibrary("CPUSupportWin");
            loaded = true;
        } catch (Error e) {
            try {
                System.loadLibrary("CPUSupportSolaris");
                loaded = true;
            } catch (Error err) {
                System.err.println(
                        "Warning: No platform library for CPUSupport");
            }
        }
    }

    private static native int getConcurrencyN();
    private static native void setConcurrencyN(int i);
    private static native int getNumProcessorsN();

    public static int getConcurrency() {
        if (!loaded)
            // 假设是绿色线程
            return 1;
        return getConcurrencyN();
    }

    public static void setConcurrency(int n) {
        if (loaded)
            setConcurrencyN(n);
    }
```

```
        public static int getNumProcessors() {
            if (!loaded)
                // 假设是绿色线程
                return 1;
            return getNumProcessorsN();
        }
    }
```

我们计划使得该类可以在所有的平台上运行；如果不存在平台专有的本地库，则我们假设是使用绿色线程模型。当然，如果需要，也可以很容易地包含对于其他操作系统的支持。现在我们要做的就是编写我们希望支持的操作系统的本地库。

在 Window 平台上实现 CPUSupport

下面是在 Windows 平台上本地库的实现代码：

```
#include <jni.h>
#include <windows.h>

JNIEXPORT jint JNICALL Java_CPUSupport_getNumProcessorsN
                            (JNIEnv *env, jobject cls)
{
    static DWORD numCPU = 0;
    SYSTEM_INFO process_info;

    if (numCPU == 0) {
        GetSystemInfo(&process_info);
        numCPU = process_info.dwNumberOfProcessors;
    }
    return numCPU;
}

JNIEXPORT void JNICALL Java_CPUSupport_setConcurrencyN
                        (JNIEnv *env, jobject cls, jint kthreads)
{
    // 对于Windows而言，该返回值（concurrency）是无限大
    return;
}

JNIEXPORT jint JNICALL Java_CPUSupport_getConcurrencyN
                            (JNIEnv *env, jobject cls)
{
    // 对于Windows而言，该返回值（concurrency）是无限大
```

```
        // 但是我们返回处理器的个数
        return Java_CPUSupport_getNumProcessorsN(env, cls);
    }
```

在Windows环境中，为了得到CPU的数目，我们仅仅调用操作系统的GetSystemInfo()函数并且获取需要的信息就可以了。但是，我们不能改变Windows平台上可以并行运行的线程个数：每一个Java线程都和一个Windows线程相关联。这导致了所有的线程都是可以并行运行的（如果有无限的内存和无限快的CPU）。因此我们仅仅返回处理器的个数，这使得我们可以了解到底有多少个线程可以同时运行。

为了在Microsoft C/C++ 5.0中编译这段代码，需要执行下面的命令：

```
cl -Ic:\java\include -Ic:\java\include\win32 -LD CPUSupportWin.c
```

你需要用你的JDK安装目录来替换上面命令中的*c:\java*。生成的DLL文件（*CPUSupportWin.dll*）要放在虚拟机可以通过环境变量PATH找到的路径中。

在Solaris平台上实现CPUSupport

下面是Solaris平台上对CPUSupport的实现：

```
#include <jni.h>
#include <thread.h>

JNIEXPORT jint JNICALL Java_CPUSupport_getConcurrencyN
            (JNIEnv * env, jobject class)
{
    return thr_getconcurrency();
}

JNIEXPORT void JNICALL Java_CPUSupport_setConcurrencyN
            (JNIEnv * env, jobject class, jint n)
{
    thr_setconcurrency(n);
}

JNIEXPORT jint JNICALL Java_CPUSupport_getNumProcessorsN
            (JNIEnv * env, jobject class)
{
```

```
    int num_threads;
    num_threads = sysconf(_SC_NPROCESSORS_ONLN);
    return num_threads;
}
```

因为这个实现仅仅是映射操作系统的调用，所以很简单。在该例中，getConcurrency()方法返回当前的LWP个数，而setConcurrency()方法设定当前的LWP个数。

通过下面的命令可以在Sun Workshop 4.2中编译：

```
cc -I/usr/java/include -I/usr/java/include/solaris -mt -G -o \
libCPUSupportSolaris.so CPUSupportSolaris.c
```

如果JDK不是安装在*/usr/java*，则修改上面命令中的相应路径。一旦该库编译成功了，你一定要将它加入到LD_ LIBRARY_PATH环境变量中去，使得虚拟机可以找到该库。

其他线程调度方法

在Thread类中还有其他可以影响线程调度的方法。正如我们将要看到的，正是因为有各种不同的本地线程调度模型以及它们对于分时间片调度的使用产生的复杂性，使得从Java调度的角度看这些方法都不是最有用的技术。另外，这些方法中的两个从Java 2开始已经不赞成使用了，而且也不应该在任何版本的Java中使用。但是，我们还是要全面地了解一下这些与线程调度有关的API。

suspend()和resume()方法

有两个方法可以直接影响线程的状态：

void suspend()(在Java 2中不赞成使用)

阻止一个线程运行。

void resume()(在Java 2中不赞成使用)

允许一个被挂起的线程继续允许。

suspend()方法将一个特定的线程从可运行状态转换为阻塞状态。在该例中，线程不是因为等待某个资源而阻塞，它是等待另外的线程来恢复它的运行。resume()方法将一个阻塞线程转换为可运行状态。

在本章前面的“线程调度综述”中，我们提到过线程有四个状态。实际上，尽管挂起状态和阻塞状态在概念上没有实际区别，但是两者是不同的。严格地说，suspend()方法会将一个处于任何状态的线程转换为挂起状态。就算是原来处于阻塞状态的线程也会像其他状态的线程一样被挂起。同样，resume()方法会将一个处于挂起状态的线程恢复成其在挂起以前的状态。因此被resume()方法恢复的线程可能还是处于阻塞状态。尽管有这些微小的区别，但是我们还是坚持认为阻塞和挂起状态是相同的状态。

使用suspend()和resume()方法的一个通常原则是在applet中使用它们。当applet不活动时，你也不希望它的线程是活动的。根据这个原则，我们来修订绘制不规则图像的applet:

```
import java.applet.Applet;
import java.awt.*;

public class Fractal extends Applet implements Runnable {
    Thread t;
    public void start() {
        if (t == null) {
            t = new Thread(this);
            t.setPriority(Thread.currentThread().getPriority() - 1);
            t.start();
        }
        else t.resume();
    }

    public void stop() {
        t.suspend();
    }

    public void run() {
        //进行计算，不时调用repaint()
    }

    public void paint(Graphics g) {
        //完全重画整个不规则图像
```

```
        }
    }
```

这个例子比我们原来的那个要好：在原来的例子中，当用户重新访问页面时，计算线程要从头开始重新计算而且要重新绘制所有的结果。现在，applet 可以保存原来的结果，而仅仅从上一次被中断的地方开始重新计算。

suspend()和 resume()的方法的替换方法

尽管在该例中有对 suspend()和 resume()方法的使用，但是因为这两个方法中潜伏有安全漏洞，所以这两个方法在Java 2中已经不推荐使用了。这个安全漏洞存在于所有的Java虚拟机中，所以在没有将这两个方法标记为不推荐使用的Java 1.0/1.1 虚拟机中，使用这两个方法也是不安全的。实际上，suspend()和 resume()方法就不应该被使用。stop()方法在Java 2中也被标记为不推荐使用，其原因是也是一样的。

suspend()方法的问题就是它会导致“锁饥饿”，甚至会出现由于“锁饥饿”导致整个虚拟机停机的情况。如果一个拥有某个锁的线程被挂起，则该线程依然拥有该锁。只要该线程是挂起的，其他的线程就没有办法获取该锁。根据这个被挂起锁的情况，可能会导致全部的线程最终因为等待该锁而被阻塞。

你可能认为只要仔细编程，就可以避免一个被挂起线程拥有锁的情况出现。但是，在Java API和虚拟机内部有着很多你所不知道的锁，因此你不知道一个将要被挂起的线程是不是拥有锁。更严重的是，如果一个正在堆上分配空间来创建对象的线程被挂起，则该线程将会拥有一个保护整个堆的锁，这使得其他线程都无法分配任何空间了。这是多么可怕的事情。

这也不是不能克服的问题：也可以这样实现虚拟机，使得一个拥有锁，或者是至少拥有一个虚拟机的内部锁的线程不能被挂起。但是 Java 虚拟机不是这样编写的，而且 Java 规范也没有这样要求。因此，suspend()就被标记为不推荐使用了。resume()方法是没有什么危险的。但是既然 resume()仅仅是对 suspend()方法有用，它也就只好也被标记为不推荐使用了。

stop()方法也有同样的问题。在stop()方法中，问题不是锁会被无限期拥有——实际上，当线程被终止时，其锁会被释放（参见附录一）。问题是复杂的数据结构可能会处于一个不稳定状态：例如，如果线程在更新一个链表的过程中被终止了，则链表的链接就可能处于不一致状态。我们对一个链表操作时，一开始要获取一个锁就是为了保证对于其他线程而言，链表不会处于不一致状态；如果我们终止线程对链表的操作，则获取锁所保证的好处就没有了。因此stop()方法也不被推荐使用。

所以我们的结论是一个线程不应当终止或者是停止另外一个线程：线程仅仅可以终止自己（通过从run()方法返回）或者是挂起自己（通过调用wait()方法）。它可以通过响应其他线程设定的标志或者使用自己设计的其他方法来做到这一点。

在前面的章节中，我们看到了不用stop()方法如何达到同样的结果。这里我们也用其他方法得到与suspend()同样的效果：

```
import java.applet.Applet;
import java.awt.*;

public class Fractal extends Applet implements Runnable {
    Thread t;
    volatile boolean shouldRun = false;
    Object runLock = new Object();
    int nSections;
    public void start() {
        if (t == null) {
            shouldRun = true;
            t = new Thread(this);
            t.setPriority(Thread.currentThread().getPriority() - 1);
            t.start();
        }
        else {
            synchronized(runLock) {
                shouldRun = true;
                runLock.notify();
            }
        }
    }

    public void stop() {
        shouldRun = false;
```

```
    }

    void doCalc(int i) {
        //计算不规则图像的第 i 块
    }

    public void run() {
        for (int i = 0; i < nSections; i++) {
            doCalc(i);
            repaint();
            synchronized(runLock) {
                while (shouldRun == false)
                    try {
                        runLock.wait();
                    } catch (InterruptedException ie) {}
            }
        }
    }

    public void paint(Graphics g) {
        //画出不规则图像的全部部分
    }
}
```

applet 的 start() 方法仍旧是负责创建和启动计算线程；另外，它还负责将 shouldRun标志设置为 true。计算线程会检查这个标志来判断它是否应当继续计算。当 run() 方法检查到该标志为 false时就等待该标志为 true。这种对于 wait() 方法的调用和对计算线程调用 suspend() 有一样的效果。同样，start() 方法提供的通知也有着和 resume() 方法同样的效果。

现在挂起一个线程需要两个步骤：applet的 stop() 方法设定标志位；计算线程的 run() 方法检查这个标志位。因此，在 applet 对用户不可见后会有一段时间计算线程还在计算不规则图像的片断。一般而言，使用这个技术的话，在你希望线程停止或者是挂起和线程通过检查标志位来判断是否该挂起自己之间是有一段时间的。但这比使用 suspend() 方法要安全（当然，我们也不保证在 Java 平台的以后版本中不会再次出现 suspend() 方法）。

为什么在applet的stop()方法中没有对访问shouldRun标志进行同步呢？ 因为对于布尔变量的测试和设置操作已经是一个原子性操作了，而 stop() 方法仅仅是

进行这样一个原子性操作，因此就不需要对stop()方法进行同步。其他的方法因为不仅仅包含了对shouldRun标志的操作，所以要定义为同步的。另外，因为它们要调用wait()或者notify()方法，所以必须要拥有一个锁。

yield()方法

最后一个可以影响当前运行线程的方法是yield()方法。它的有用性体现在它允许同样优先级的其他线程运行。

static void yield()

产生当前线程，同时允许Java虚拟机运行具有同样优先级的另外一个线程。

yield()方法有很多值得注意的地方。第一，注意这是一个静态方法，并且仅仅影响当前运行线程。如下面的代码片断：

```
public class YieldApplet extends Applet implements Runnable {
    Thread t;
    public void init() {
        t = new Thread(this);
    }

    public void paint(Graphics g) {
        t.yield();
    }
}
```

当applet线程执行paint()方法并调用yield()方法时，尽管我们使用对象t来调用yield()方法，但是yield()方法产生applet线程，而不是计算线程t。

当一个线程调用yield()时到底发生了什么？对于线程的状态而言，什么都没有发生：线程仍然是处于可运行状态。但是逻辑上，该线程已经移动到其优先级链表的尾部，因此Java虚拟机就根据同样的选择原则来选取一个新的线程作为当前运行线程。很清楚，系统中不存在比刚产生的线程优先级更高的线程，因此新的线程就是和刚产生的线程具有相同优先级的线程。如果没有相同优先级的线程，则调用yield()方法不起任何作用，刚产生的线程会立刻被再次选择成为当前运行线程。在这种情况下，调用yield()方法就如同调用sleep(0)一样。

如果系统中有和调用yield()方法产生的线程同样优先级的线程，则其中的一个就会成为当前运行线程。因此，yield()方法可以使你知道系统中是否有同样优先级的线程。但是，这并不保证到底哪一个线程会被选中：尽管在系统中有其他具有同样优先级的线程，调用yield()方法产生的线程也可能被调度程序选中。

下面让我们看看在绘制不规则图像的例子中使用yield()方法代替优先级调用的结果：

```
import java.applet.Applet;
import java.awt.*;

public class Fractal extends Applet implements Runnable {
    Thread t;
    volatile boolean shouldRun = false;
    Object runLock = new Object();
    int nSections;

    public void start() {
        if (t == null) {
            shouldRun = true;
            t = new Thread(this);
            t.start();
        }
        else {
            synchronized(runLock) {
                shouldRun = true;
                runLock.notify();
            }
        }
    }

    public void stop() {
        shouldRun = false;
    }

    void doCalc(int i) {
        // 计算不规则图像的第 i 块部分
    }

    public void run() {
        for (int i = 0; i < nSections; i++) {
            doCalc(i);
            repaint();
```

```
                Thread.yield();
                synchronized(runLock) {
                    while (shouldRun == false)
                        try {
                            runLock.wait();
                        } catch (InterruptedException ie) {}
                }
            }
        }

        public void paint(Graphics g) {
            // 绘制不规则图像的全部部分
        }
    }
```

在该例中，我们不再设置计算线程的优先级低于applet中的其他线程。现在当我们的计算线程得到结果时，它就仅仅调用yield()方法。当计算线程调用repaint()方法时，applet线程会进入可运行状态。因此Java虚拟机就选择applet线程作为当前运行线程，applet重新绘制自己然后阻塞，计算线程再次成为当前运行线程来计算不规则图像的下一个部分。

这个例子中还是存在一些问题。第一，因为applet线程和计算线程具有相同的优先级，因此在产生计算线程之前，用户无法和applet进行交互。例如，如果用户选择GUI中的一个选择框，程序可能在计算线程产生前无法采取恰当的行动。在一个使用本地线程调度的平台上，因为applet线程和计算线程都是分时间片的，这一般不会发生。但是在一个使用绿色线程库的平台上这就是一个大问题。

第二，在该例中存在竞态条件——其实在所有依赖于yield()方法的例子中都是存在的。这仅仅在对于具有同样优先级的线程进行分时间片操作的本地线程平台上才会发生，并且如同其他竞态条件一样，它发生的概率是很小的。在上面的例子中，当计算线程刚刚产生后，可能操作系统正好也在对另一个线程或者是LWP进行调度。这意味着尽管计算线程刚刚产生，它还是又成为了下一个要运行的线程。如果发生了这种情况，好的方面是程序还是会继续运行，而且当计算线程下一次产生或者操作系统下一次调度applet线程时，这次计算出的不规则图像部分还是会绘制上去。

从坏的方面来说，产生的线程仍然成为要运行的线程。但是，这种情况仅仅在操

作系统会对线程进行调度，而且是对具有同样优先级的线程进行分时间片调度时才会发生。在上面这种情况中，其实是没有必要使用yield()方法的。如果程序是希望通过yield()方法来通知Java虚拟机现在正是改变当前运行线程的时机的话，则无论是在使用绿色线程实现（在这种环境中，yield()方法总是可以得到期望的结果）还是在使用本地线程实现的Java虚拟机上使用yield()方法都是没有问题的。

yield()方法和基于优先级的调度之间的比较

如果希望对线程调度进行控制，则对于是使用yield()方法还是调整单个线程的优先级这个问题是仁者见仁、智者见智的。原因就在于这两个方法有着相似的作用。在我们的例子中，很清楚我是喜欢使用基于优先级的方法来控制线程调度的。这些方法使得Java程序员具有足够的灵活性。

我很少认为yield()方法是有用的。这可能使得那些主要使用yield()方法影响线程调度的平台上编程的程序员感到吃惊。但是在使用本地线程的Java虚拟机上对于具有同样优先级线程调度的不确定性，这使得yield()方法不是那么可靠：一个调用yield()方法的线程可能因为操作系统进行分时间片调度而又被选中成为当前运行线程。但是另一方面，如果你的线程经常使用yield()方法，那么从长远来看竞态条件也不是一个问题了，而且yield()方法也是进行线程调度的有效方法。同时，yield()方法相对于基于优先级的方法而言也是容易理解的，这一切使得不少程序员也很喜欢使用它。

守护线程

最后一个和线程调度有关的问题是和守护线程相关的。在Java系统中有两种类型的线程：守护线程和用户线程。从字面上看，守护线程是由Java API在内部创建的，而用户线程是由用户创建的。但其实并非如此。任何线程都可以是守护线程或者是用户线程。所有线程在一开始都是用户线程。所以我们到现在为止所看到的所有线程都是用户线程。

一些由虚拟机代替你创建的线程是守护线程。一个守护线程在很多方面都和用户线程一样：有优先级，有同样的方法，经过同样的状态转换。对于调度而言，守护线程是和用户线程一样进行调度的：没有一种线程比另外一种优先级高。在程序的运行过程中，守护线程和用户线程的行为是一样的。

仅仅是当一个用户线程结束后，Java虚拟机才检查系统中的线程是否是守护线程。当一个用户线程结束后，Java虚拟机会检查是否有其他的用户线程存在。如果有其他线程存在，则Java虚拟机根据我们上面讨论过的原则，调度下一个线程运行（可能是用户线程，也可能是守护线程）。如果系统中仅仅有守护线程了，Java虚拟机会退出，程序也会终止。

在Java虚拟机的参考实现中，最典型的守护线程是垃圾收集线程（在其他的虚拟机实现中，垃圾收集程序可能不需要使用一个单独的线程）。对于Java程序不再引用的对象，垃圾收集程序不定期地进行回收，这也就是为什么Java程序不需要担心内存管理的原因。因此，垃圾收集程序是一个很有用的线程。如果系统中没有其他的线程运行，则垃圾收集程序也就没有什么东西可以回收了：毕竟垃圾收集程序不是Java程序自己创建的。因此，如果垃圾收集程序是Java虚拟机中惟一运行的线程，则很清楚，它是没有什么工作要去做的，并且Java虚拟机也可以退出了。因此，垃圾收集程序是一个守护线程。

在Thread类中有两个关于守护线程的方法：

void setDaemon(boolean on)

设置一个线程是守护线程（将on设定为true）或者是一个用户线程（将on设定为false）

boolean isDaemon()

如果线程是一个守护线程则返回true，如果是一个用户线程则返回false。

setDaemon()方法仅仅在线程对象已经被创建但是还没有运行前才能被调用。如果一个线程已经运行了，你就不能将一个用户线程变为一个守护线程（反之亦然）；试图这样做会产生异常。为了绝对的正确，在任何时候对一个活动的线程调用setDaemon()方法都会产生异常，即使该线程就是一个守护线程。

默认情况下，一个由用户线程创建的线程是用户线程；一个守护线程创建的线程是守护线程。仅仅是当一个线程创建另一个线程并且希望改变它的状态时才使用 setDaemon()方法。

但不幸的是，用户线程和守护线程之间的区别不很清晰。尽管一般而言，守护线程是用来对用户线程提供服务的，但守护线程完成这种服务的时间可能比提出要求的用户线程的生命周期要长。更进一步，可能有些关键服务不希望因为Java虚拟机的退出而被中断。例如，一个进行数据备份的线程在没有用户线程生成数据的情况下是没有什么用处的。但是，在其进行数据备份的过程中，数据库所处的状态是不允许该线程终止的。对于该备份线程而言，当没有其他线程生成数据时它就没有什么用处，从这个角度看它应当作为一个守护线程，但是，为了保护数据库的一致性，你还是要将它作为一个用户线程。

最理想的是，我们可以允许该线程在用户线程和守护线程之间进行切换。因为这不是 Java API 所允许的，我们可以通过实现一个锁来保护守护线程。一个 DaemonLock 类的实现如下所示：

```
public class DaemonLock implements Runnable {
    private int lockCount = 0;

    public synchronized void acquire() {
        if (lockCount++ == 0) {
            Thread t = new Thread(this);
            t.setDaemon(false);
            t.start();
        }
    }

    public synchronized void release() {
        if (--lockCount == 0) {
            notify();
        }
    }

    public synchronized void run() {
        while (lockCount != 0) {
            try {
                wait();
            } catch (InterruptedException ex) {};
```

```
            }
        }
    }
```

该类的实现是简单的：通过保证存在一个用户线程来保护守护线程。只要系统中存在一个用户线程，虚拟机就不会退出，这也就允许守护线程来完成其关键任务。当该关键任务完成后，守护线程就可以释放该Daemon锁，从而导致用户线程的终止。如果系统中没有其他用户线程，守护线程就会退出。但是不同之处在于，我们现在可以保证程序退出到代码的关键部分之外。

在第七章我们会看到一个使用这个类的例子。

总结

下面是我们在本章中使用的Thread类的方法：

void setPriority(int priority)

设置给定线程的优先级。如果该值超过了允许的范围，则抛出一个异常。但是如果该值是在允许的范围以内但是又超过了该线程所在的线程组所允许的最大值，则设定的优先级为最大允许的值。

int getPriority()

获取给定线程的优先级。

void suspend()(在Java 2中不赞成使用)

阻止一个线程运行。

void resume()(在Java 2中不赞成使用)

允许一个被挂起的线程继续运行。

static void yield()

产生当前线程，同时允许Java虚拟机运行具有同样优先级的另一个线程。

void setDaemon(boolean on)

设置一个线程是守护线程（将on设定为true）或者是一个用户线程（将on设定为false）。

boolean isDaemon()

如果线程是一个守护线程则返回true，如果是一个用户线程则返回false。

在本章中，我们在线程优先级和调度上花费了大量时间。因为在Java规范中没有定义线程调度模型，所以线程调度在不同的平台上可以有不同的行为，这使得调度问题在Java编程中是一个灰色地带。调度模型的不同很少会影响程序的最终结果或有用性，因此Java对调度采取了简单化的原则。对于那些线程调度会很重要的情况，Java将由于使用显式调度而产生的复杂性留给了程序员来处理。

因此，不同的Java虚拟机实现对于调度的处理方式是不同的。最简单的模型（绿色线程模型）使用相当严格的基于优先级的调度算法；而使用操作系统专有线程库的模型也基本遵循这个算法，但是在决定选择哪一个线程来运行时又考虑了其他一些因素。

在下一章中，我们将了解一些基于优先级调度的调度技术。

第七章

Java线程调度例子

本章内容：

- 线程池
- 循环调度
- 作业调度
- 总结

当程序中有固定数目的线程并且能够事先分析它们的行为时，上一章中的线程方法非常有用。而当用户对于中间结果的显示感兴趣时，基于优先级的调度方法也是很有用的。但是，有时候你需要在任何平台上都可以对一些独立的线程进行循环时间片调度。也有时候，创建多个线程是很方便的，但是你不想使用某些平台上提供的循环时间片调度算法。另外，因为某些平台不能处理太多数目的线程，所以有时候你也需要限制程序中使用的线程数目。

在本章中我们将考虑这些问题，而且会提供四个如何使用Java程序实现通用调度的例子。我们从介绍线程池（thread pool）开始：线程池中拥有有限数目的线程，但是其中的每一个线程都可以依次运行多个对象。当某个特定的虚拟机不能支持程序需要的线程个数时，线程池是很有用的。

接着，我们会介绍两个基于循环调度的调度程序：一个是非常简单的例子（主要适合于绿色线程实现），另一个是更通用的例子，它既可以用来实现循环调度，也可以用来屏蔽循环调度。使用该调度例子是因为：

- 只在有限的时候才会需要使用这样的调度程序。
- 通过对该调度程序的开发，可以说明编写多线程程序时需要考虑的问题。

最后，我们会展示一个作业调度程序。当一个作业需要在某个特定的时间运行时，该调度程序是很有用的。

线程池

首先我们来介绍线程池。创建ThreadPool类的原始想法是创建很多线程，这些线程处于空闲状态，等待处理某些任务。但可能与你想象的不一样，使用该类的基本原理不是想通过预先创建线程来节省时间：在很多平台上，启动一个线程所需要的代价并不比让一个线程等待运行某些工作所需要的代价小。

实际上，这个类的设计目的是：为了在程序中更有效地使用机器的资源而限制程序中使用的线程个数。例如，Java API中就使用这个技术来限制加载图像的线程个数。如果你创建一个 MediaTracker 对象，则 Java API 会创建四个线程来同时取出MediaTracker对象中注册的图像资源。这就限制了程序所使用的提供图像的服务器的工作负载，同时也限制了程序用来加载图像而使用的带宽。

这个技术经常用在计算服务器上。如果你的计算服务器同时接收到很多请求，那么对每个请求都创建一个不同的线程可能不是一种有效的方法。如果服务器是在一个使用某种循环调度算法的本地线程平台上运行，则这些请求就会相互竞争CPU资源。在这些情况下，最好是只创建和机器上处理器个数一样多的线程（如果该机器还要处理其他工作，则可能使用比处理器个数更少的线程）。在一个不能够同时处理太多线程的平台上，这种技术也可以用来限制程序中活动线程的个数。

下面是一个线程池类的实现：

```
import java.util.*;

public class ThreadPool {

    class ThreadPoolRequest {
        Runnable target;
        Object lock;

        ThreadPoolRequest(Runnable t, Object l) {
            target = t;
```

```
            lock = 1;
        }
    }

    class ThreadPoolThread extends Thread {
        ThreadPool parent;
        volatile boolean shouldRun = true;
        ThreadPoolThread(ThreadPool parent, int i) {
            super("ThreadPoolThread " + i);
            this.parent = parent;
        }

        public void run() {
            ThreadPoolRequest obj = null;
            while (shouldRun) {
                try {
                    parent.cvFlag.getBusyFlag();
                    while (obj == null && shouldRun) {
                        try {
                            obj = (ThreadPoolRequest)
                                    parent.objects.elementAt(0);
                            parent.objects.removeElementAt(0);
                        } catch (ArrayIndexOutOfBoundsException aiobe) {
                            obj = null;
                        } catch (ClassCastException cce) {
                            System.err.println("Unexpected data");
                            obj = null;
                        }
                        if (obj == null) {
                            try {
                                parent.cvAvailable.cvWait();
                            } catch (InterruptedException ie) {
                                return;
                            }
                        }
                    }
                } finally {
                    parent.cvFlag.freeBusyFlag();
                }
                if (!shouldRun)
                    return;
                obj.target.run();
                try {
                    parent.cvFlag.getBusyFlag();
                    nObjects--;
                    if (nObjects == 0)
                        parent.cvEmpty.cvSignal();
                } finally {
```

```
                parent.cvFlag.freeBusyFlag();
            }
            if (obj.lock != null) {
                synchronized(obj.lock) {
                    obj.lock.notify();
                }
            }
            obj = null;
        }
    }
}

Vector objects;
int nObjects = 0;
CondVar cvAvailable, cvEmpty;
BusyFlag cvFlag;
ThreadPoolThread poolThreads[];
boolean terminated = false;

public ThreadPool(int n) {
    cvFlag = new BusyFlag();
    cvAvailable = new CondVar(cvFlag);
    cvEmpty = new CondVar(cvFlag);
    objects = new Vector();
    poolThreads = new ThreadPoolThread[n];
    for (int i = 0; i < n; i++) {
        poolThreads[i] = new ThreadPoolThread(this, i);
        poolThreads[i].start();
    }
}

private void add(Runnable target, Object lock) {
    try {
        cvFlag.getBusyFlag();
        if (terminated)
            throw new
                IllegalStateException("Thread pool has shut down");
        objects.addElement(new ThreadPoolRequest(target, lock));
        nObjects++;
        cvAvailable.cvSignal();
    } finally {
        cvFlag.freeBusyFlag();
    }
}

public void addRequest(Runnable target) {
    add(target, null);
}
```

```
public void addRequestAndWait(Runnable target)
                        throws InterruptedException {
    Object lock = new Object();
    synchronized(lock) {
        add(target, lock);
        lock.wait();
    }
}

public void waitForAll(boolean terminate)
                        throws InterruptedException {
    try {
        cvFlag.getBusyFlag();
        while (nObjects != 0)
            cvEmpty.cvWait();
        if (terminate) {
            for (int i = 0; i < poolThreads.length; i++)
                poolThreads[i].shouldRun = false;
            cvAvailable.cvBroadcast();
            terminated = true;
        }
    } finally {
        cvFlag.freeBusyFlag();
    }
}
```

```
    public void waitForAll() throws InterruptedException {
        waitForAll(false);
    }
}
```

在该例中，内部类完成了绝大部分的工作。每一个线程都在等待工作；当线程得到通知后，就从存放对象的向量中取出第一个对象来执行。当该对象的执行完成后，线程必须通知与该对象相关联的锁（如果存在的话），使得addRequestAndWait()方法知道何时可以返回；同时，线程也要通知线程池本身，使得waitForAll()方法会检查是否应该返回。

因此，在上面的代码中有三个等待点：

- 一些请求对象有一个相关联的锁对象（在addRequestAndWait()方法中创建的Object对象）。addRequestAndWait()方法使用标准的等待和通知机制在该对象上进行等待；当某个ThreadPoolThread对象执行完run()方法后，它会接收到通知信号。

- 一个与cvBusyFlag相关联的条件变量cvAvailable。这个变量用来通知有工作在等待处理。只要nObjects变量增加，就是有等待处理的工作，因此add()方法就可以向一个线程发出表示有新工作存在的信号。同样，当向量中没有等待处理的对象时，ThreadPoolThreads会在该条件变量上等待。
- 与cvBusyFlag相关联的条件变量cvEmpty。该条件变量用来表示所有的任务都已经完成了，即nObjects变量为零。waitForAll()方法是等待该条件，而一个ThreadPoolThread对象在将nObjects设置为零时会发出该信号。

因为后面两种情况共享一个锁（用来保护对nObjects访问的cvBusyFlag），所以尽管在不同的情况下条件变量有不同的值，我们还是使用两个条件变量。如果也使用标准的等待和通知机制来通知对nObjects值感兴趣的线程，则我们不能使通知受到很好的控制：只要nObjects变为零，我们就不得不通知所有的ThreadPoolThreads对象，同时也不得不通知运行waitForAll()方法的线程。

同时，ThreadPool类自己也仅仅提供了创建线程池（同时设定线程池中线程的个数）、向池中增加对象、等待池中所有对象执行完毕的方法。毫不奇怪，该类有着和MediaTracker类相似的地方；它是对该类的简单扩展，通过对向量中的每一个对象都赋予一个ID来模似MediaTracker的接口。另外，addRequestAndWait()方法也类似于JFC（Java™ Foundation Class，Java基础类）中使用的技术，这种技术使得任意线程（可能不是线程安全的）都可以在JFC中运行。我们会在第八章中对此进行简单的讨论。

要注意的是，在线程池中运行的对象应当是实现了Runnable接口的。因为我们经常认为一个实现了Runnable接口的对象会在它自己的线程中运行（Thread类本身也实现Runnable接口），因此这一点可能会使人觉得胡涂。在该例中，如果创建一个线程对象，然后使用addRequest()方法将其加入到线程池中，再显式地运行该线程对象是一种错误的做法：在这种情况下，我们期望线程池来运行该对象。但是与创建一个新的类或者接口的做法相比，使用Runnable接口更加清晰。使用Runnable接口还允许在线程池中运行那些使用线程技术的已存在代码。

有趣的是，没有办法来自动关闭线程池。如果线程池对象超出了作用域，垃圾收集程序也不会对其进行回收：线程池中的线程对象（和其他所有的线程对象一样）

在虚拟机中都有一个内部数据结构，因此在它们结束前是不会被垃圾收集程序回收的。而因为这些线程在线程池中都保有一个对自己的引用，所以在线程池中所有的线程都被回收之前，线程池是不会被回收的。因此，我们必须要有某种方式来通知线程池结束运行：这通过向 waitForAll()方法传递一个参数 true 来实现。然后，当线程池运行完全部作业后，waitForAll()方法安排线程池中的线程结束并标记线程池不会再接收新的任务。线程池中的线程会退出运行，然后线程池就可以被垃圾收集程序回收。

下面看一个使用线程池的例子。因为想通过线程池来限制同时运行的线程个数，所以会使用第六章中开发的CPUSupport类来得到CPU的个数，从而保证线程池中的线程个数和机器上的CPU个数相同。我们用TCPServer类作为整个例子的基础。

```
import java.io.*;
import java.net.*;

public class TCPCalcServer extends TCPServer {
    class CalcObject implements Runnable {
        OutputStream os;
        InputStream is;

        CalcObject(InputStream is, OutputStream os) {
            this.os = os;
            this.is = is;
        }

        public void run() {
            // 进行计算
        }
    }

    ThreadPool pool;

    TCPCalcServer() {
        int numThreads = CPUSupport.getNumProcessors();
        CPUSupport.setConcurrency(numThreads + 5);
        pool = new ThreadPool(numThreads);
    }

    public static void main(String args[]) {
        try {
```

```
                new TCPCalcServer().startServer(3535);
            } catch (IOException ioe) {
                // 省略错误处理过程
            }
        }

        public void run(Socket data) {
            try {
                pool.addRequest(new CalcObject(data.getInputStream(),
                                               data.getOutputStream()));
            } catch (IOException ioe) {
                // 省略错误处理过程
            }
        }
    }
```

TCPServer类中真正的工作是在run(data)方法中完成的。在该例中，我们仅仅创建一个新的计算对象然后将该对象加入到线程池中。尽管忽略了实际的计算过程，但还是提供了该计算对象的代码框架的一个简单实现。要再次说明的是，尽管CalcObject类实现了Runnable接口，但是它没有和任何一个指定的线程联系起来。

run(data)方法难道不是在一个单独的线程中运行吗？这是否和原先设计的限制线程个数的目标相冲突呢？我们原先的目标是限制程序中同时活动的线程个数。而对于该服务器程序而言，调用run(data)的线程其运行时间是很短的。因此，尽管对每一个用户连接，都会创建一个新的线程，但是在任意时刻，线程个数还是相当少的。如果你愿意，也可以重写TCPServer类（而不是将它子类化），但是这个例子已经运行得很好了。

因为该服务器是一个计算服务器，我们应当在Solaris系统上设置LWP的个数，从而保证可以充分利用所有CPU的能力。我们将并发线程的个数设置为CPU的个数（加上了虚拟机中进行I/O绑定的5个线程）。而在其他所有的平台上，我们将线程池中线程的个数设置为CPU的个数。这样我们就可以最大限度地利用机器的能力。

循环调度

下面的例子使用了两种实现循环调度的方法。因为本地线程实现都使用某种形式的循环调度，所以对于那些在使用绿色线程实现的Java虚拟机上运行的程序而言，该技术是很有用处的。如果你不清楚最终程序会在什么样的实现上运行，也可以使用其中一种技术来保证在程序中使用循环调度，即使底层实现会为你完成这个工作。在使用本地线程实现的平台上，尽管这些技术不能提供任何好处，但是它们也不会和本地线程实现冲突。

还记得当Java程序中包含一个或者多个占用CPU较多的线程时产生的矛盾吗？一个Java程序中的几百个线程可能只是周期性地需要CPU，或者在生命周期的绝大部分时间里是处于阻塞状态的。在这种情况下，不可能产生对CPU的激烈竞争：此时，每一个希望得到CPU的线程都可以得到运行的机会。而只有当存在至少一个占用CPU较多的线程，并且该线程可能会阻止其他所有线程运行时，才会导致CPU饥饿现象的发生。

如果只有一个占用CPU较多的线程，就没有必要使用复杂的调度算法：只需在Java程序中将该占用CPU较多的线程的优先级降低到低于其他所有线程的优先级就可以了。这使得其他线程可以在需要时得到CPU，而在其他线程阻塞时，占用CPU较多的线程可以继续运行。我们要实现的调度类也会遵循这一原则：相对于那些经常阻塞的线程而言，占用CPU较多的线程的优先级要低。

在本节中，我们会看到两个调度程序。这两个调度程序都遵守同样一个基本原则：每一个在其控制下的线程在运行过程中，都会被分配固定长度的时间片。当给定的时间片使用完后，另外一个线程就会运行。

简单的循环调度程序

如何创建一个循环调度程序？很显然，需要某种周期性的定时器。当定时器的时间到了时，我们就可以选择一个不同的线程成为当前运行线程。那么如何做到这一点呢？

最简单的回答是：什么也不需要做。这个简单的调度程序仅仅是一个具有高优先级的定时器，它被周期性地唤醒，然后再次立刻进入睡眠状态。这实际上等于创建了一个基于时间的调度事件：当定时器线程醒来时，它就成为当前运行线程，从而导致原来运行线程所处的链表被重新调整。

```
public class SimpleScheduler extends Thread {
    int timeslice;

    public SimpleScheduler(int t) {
        timeslice = t;
        setPriority(Thread.MAX_PRIORITY);
        setDaemon(true);
    }

    public void run() {
        while (true)
            try {
                sleep(timeslice);
            } catch (Exception e) {}
    }
}
```

我们使用第六章中用过的例子来进行演示：

```
class TestThread extends Thread {
    String id;

    public TestThread(String s) {
        id = s;
    }

    public void doCalc(int i) {
    }
    public void run() {
        int i;
        for (i = 0; i < 10; i++) {
            doCalc(i);
            System.out.println(id);
        }
    }
}

public class Test {
    public static void main(String args[]) {
```

```
            new SimpleScheduler(100).start();
            TestThread t1, t2, t3;
            t1 = new TestThread("Thread 1");
            t1.start();
            t2 = new TestThread("Thread 2");
            t2.start();
            t3 = new TestThread("Thread 3");
            t3.start();
        }
    }
```

在该程序中，三个线程（t1、t2 和 t3）都具有 Java 的默认线程优先级 NORM_PRIORITY。而SimpleScheduler线程是在MAX_PRIORITY优先级上运行的。SimpleScheduler 线程通常是阻塞的，因此一开始的线程链表如下:

```
优先级 5: t2 -> t3 -> t1 -> NULL
   阻塞: SimpleScheduler -> NULL
```

此时，t1 是当前运行线程，因而我们一开始会看见输出“Thread 1”。当 SimpleScheduler 醒来时，它变成可运行状态。因为它具有 Java 虚拟机中最高的优先级，所以成为当前运行线程:

```
优先级 5: t2 -> t3 -> t1 -> NULL
优先级 10: SimpleScheduler -> NULL
```

SimpleScheduler 会立刻执行 sleep()方法而重新变成阻塞状态；于是 Java 虚拟机从链表中选择下一个线程(t2)作为当前运行线程，并且将它移到链表的尾部:

```
优先级 5: t3 -> t1 -> t2 -> NULL
   阻塞: SimpleScheduler -> NULL
```

周而复始，在链表中优先级为 5 的每一个线程都依次成为当前运行线程。

这个调度程序要求每次当虚拟机从优先级链表中选取一个线程运行后，都会对该链表重新排序。正如在上一章中所提到的，尽管这不是Java规范所要求的，但通常都是这样实现的。但是我们也知道，在某个实时操作系统上，这种调度机制不能工作。

注意，该机制在本地线程实现上也可以工作。在Windows平台上，当前运行线程改变的速度比 SimpleScheduler 中指定的时间要快得多。这是因为当调度程序睡眠时，操作系统会在某些时刻改变当前运行线程。在Solaris平台上，线程的重新排序依赖于LWP的数目，但是对于一个LWP而言，中断已经足以使它调度另外一个线程了，这也就达到了我们的预期目的。

一个更复杂的调度程序

现在介绍一个更加完善的调度程序，它可以用循环的方式来调度线程。即使是在将分时间片调度作为默认调度方式的本地线程平台上，也可以使用它来限制循环调度，通过让该调度程序为一个特定的线程分配很大的时间片，可以实现这种限制。但是，在某些情况下，本地线程平台上具有最高优先级的线程并不一定是当前运行线程，因此我们不能够完全阻止在这些平台上发生的循环调度：我们所能做的仅仅是通过调度程序促使操作系统更加偏爱某个线程。

本节中的例子都是假设系统中只有一个 CPU。如果你希望在具有多个 CPU 的机器上使用这种技术，就需要创建 *N* 个当前运行线程（*N* 是机器中 CPU 的个数）而不是我们例子中的一个当前运行线程来调整该调度程序的设置。正如我们说过的，这种技术可以在具有多个处理器的机器上运行，它可以有效地防止任何CPU饥饿现象的出现，但是对于全部线程进行调度的效果要差一些。

我们通过创建三个具有不同优先级的线程来建立调度程序：

级别 6

调度程序本身是一个运行在级别6上的单独线程。这个级别既有利于Java虚拟机和 Java API 创建的默认线程的运行，也有利于调度程序所控制的线程的运行。该线程在绝大多数时间内是睡眠的（也就是阻塞的），因此它不会经常成为当前运行线程。

级别 4

调度程序从其控制的线程中选择一个并将它的优先级设定为4。在大部分时间内，该线程是Java虚拟机中优先级最高的非阻塞线程，因此它会被选择出来成为当前运行线程。

优先级 2

由调度程序控制的其他所有线程都运行在优先级2上。因为总有一个线程运行在优先级 4 上，所以在优先级 2 上的这些线程是不会在该级别上运行的；它们会停留在该线程优先级上，直到被调度程序所选中，并且被赋予优先级 4 为止。那时，这个被选中的线程就很可能会成为当前运行线程。

该调度程序背后的想法是程序员让调度程序来控制某些线程的运行。调度程序从这些线程中选择一个，并且将它的优先级设定为4。其他的线程优先级保持为2。这个优先级为4的线程会成为当前运行线程。调度程序会不时被自己唤醒来选择一个不同的线程作为惟一具有优先级 4 的线程。在使用绿色线程实现的平台上，具有优先级4的线程会被选择成为当前运行线程，而在使用本地线程实现的平台上，该线程一般也会被选择成为当前运行线程。

对于该调度系统中的所有线程而言(包括调度程序本身的线程以及它控制的所有线程)，很显然不存在CPU饥饿现象：调度程序本身的线程会在需要时运行，而当调度程序线程在调整其所控制的线程优先级时，所有其他的线程都有成为当前运行线程的机会。

为了管理所有的线程，我们要使用在第五章中开发的CircularList类。这个类使我们可以通过一个队列来管理所有被调度程序控制的线程：通过 insert()方法可以将一个线程加入到链表中，也可以通过 delete()方法来删除它们。更重要的是，可以通过循环调用 getNext()方法来遍历链表。

下面是调度程序的实现：

```
public class CPUScheduler extends Thread {
    private int timeslice;                          // 线程应当运行的毫秒数
    private CircularList threads;                   // 所有被调度的线程

    public volatile boolean shouldRun = false; // 当该标志被设置时退出

    public CPUScheduler(int t) {
        threads = new CircularList();
        timeslice = t;
    }
```

```
    public void addThread(Thread t) {
        threads.insert(t);
        t.setPriority(2);
    }

    public void removeThread(Thread t) {
        t.setPriority(5);
        threads.delete(t);
    }

    public void run() {
        Thread current;
        setPriority(6);
        while (shouldRun) {
            current = (Thread) threads.getNext();
            if (current == null)
                return;
            current.setPriority(4);
            try {
                Thread.sleep(timeslice);
            } catch (InterruptedException ie) {};
            current.setPriority(2);
        }
    }
}
```

尽管在本章的后面还要对该调度程序进行一些必要的调整，但是上面的代码就是调度程序的精髓。后面要增加的代码仅仅是为了使调度程序更加健壮和实现线程安全，并没有增加什么基本功能：在了解那些细节问题之前，还是先来理解这些基本功能吧。

程序员使用两个方法来和调度程序打交道：addThread()用来将一个线程添加到调度程序控制的线程链表中；removeThread()方法将一个线程对象从链表中删除（注 1）。

通过这个接口，就可以在本节前面介绍的 ThreadTest 类中使用 CPUScheduler 类了。

注 1：这里有一个小小的错误，当线程从调度程序中删除时，我们为它分配默认的线程优先级，而不是它添加到调度程序中时的优先级。正确的做法是将线程的优先级保存在对 addThread()方法的调用中，并在 removeThread()方法中恢复该优先级。我们将这个实现留给读者去考虑。

```
class TestThread extends Thread {
    String id;

    public TestThread(String s) {
        id = s;
    }

    public void doCalc(int i) {
    }

    public void run() {
        int i;
        for (i = 0; i < 10; i++) {
            doCalc(i);
            System.out.println(id);
        }
    }
}

public class Test {
    public static void main(String args[]) {
        CPUScheduler c = new CPUScheduler(100);
        TestThread t1, t2, t3;
        t1 = new TestThread("Thread 1");
        t2 = new TestThread("Thread 2");
        t3 = new TestThread("Thread 3");
        c.addThread(t1);
        c.addThread(t2);
        c.addThread(t3);
        t1.start();
        t2.start();
        t3.start();
        c.start();
    }
}
```

当程序调用c.start()时，CPUScheduler的run()方法被调用；正是这个run()方法通过一个分时间片循环调度算法来操作所有线程。从本质上来说，该调度程序的原理是很简单的：在一个无限循环中，遍历链表中的所有线程并且调整它们的优先级。在循环之间会睡眠指定毫秒长的时间片。当调度程序被唤醒前，当前运行线程一直运行，直到调度程序再次调整线程的优先级。当程序员使用removeThread()方法将原先加入链表的所有线程删除后，就没有线程需要进行调度了，此时调度程序会从run()方法中返回。

现在让我们研究一下程序中的t1、t2、t3、t4和CPUScheduler线程是如何运行的。在调用了c.start()方法后，程序中线程的状态是：

```
优先级 2: t1 -> t2 -> t3 -> NULL
优先级 6: CPUScheculer -> NULL
```

CPUScheduler线程是程序中优先级最高的线程，因此是当前运行线程。它开始执行其run()方法，该方法的第一件事就是设置线程t1的优先级为4：

```
优先级 2: t2 -> t3 -> NULL
优先级 4: t1 -> NULL
优先级 6: CPUScheduler -> NULL
```

当前运行线程仍然是CPUScheduler，它调用sleep()方法使自己进入阻塞状态。这使得t1成为当前运行线程：

```
优先级 2: t2 -> t3 -> NULL
优先级 4: t1 -> NULL
   阻塞: CPUScheduler -> NULL
```

当CPUScheduler线程被唤醒后，它将t1的优先级改回到2，同时将t2的优先级改为4：

```
优先级 2: t3 -> t1 -> NULL
优先级 4: t2 -> NULL
优先级 6: CPUScheduler -> NULL
```

系统就这样一直循环下去。

调整1：对CPUScheduler中的数据进行同步

现在CPUScheduler的基本逻辑是正确的，因此我们要保证的是CPUScheduler是线程安全的，并且不会因为没有正确地进行同步操作而造成竞态条件。我们会通过一系列的步骤来逐步实现这个目的。这样做的原因是：这些步骤也是以后设计任何要与多个线程一起工作的类所要采取的步骤。

初看起来，似乎没有什么变量是需要被同步的：惟一一个需要被保护的实例变量就是threads，而所有会改变该变量的操作都是通过CircularList类来进行的，而

这些方法也已经是同步的了。但是如果我们调用removeThread()方法来删除一个被CPUScheduler标记为当前运行线程的线程，会有什么结果呢？CPUScheduler在试图改变一个已经从线程链表中删除的线程的优先级时会产生错误。因此，removeThread()方法必须通过某种方法通知CPUScheduler当前线程已经被删除了。

这就意味着变量current应当成为一个实例变量，这样run()和removeThread()方法就都可以访问它了。我们可以同步对这个变量的访问。下面就是新的CPUScheduler类：

```
public class CPUScheduler extends Thread {
    ...
    private Thread current;
    public void removeThread(t) {
        t.setPriority(5);
        threads.delete(t);
        synchronized(this) {
            if (current == t)
                current = null;
        }
    }
    ...
    public void run() {
        ...
        try {
            Thread.sleep(timeslice);
        } catch (InterruptedException ie) {};
        synchronized(this) {
            if (current != null)
                current.setPriority(2);
        }
    }
}
```

另外的方法是可以将run()和removeThread()方法定义为同步的：

```
    public synchronized void run() {
        ...
    }

    public synchronized void removeThread(Thread t) {
        ...
    }
```

正如我们已经知道的，将run()方法定义为一个同步方法通常不是一个好主意。因此现在我们不会采用这个方法，但是很快我们会再次考虑这个决定。

调整 2：保证 CPUScheduler 线程安全

我们已经同步了 CPUScheduler 中的所有变量，但是仍然不能防止那些仍然处于 CPUScheduler 控制中的线程退出运行。

特别是，只有在该线程处于运行状态时，用run()方法改变它的优先级才是有效的操作。因此，如果 CPUScheduler 处于睡眠状态，而处于优先级 4 的线程从其run()方法中退出了，这会有什么结果呢？当CPUScheduler醒来时，它试图将这个处于退出状态的线程的优先级设为2（这会产生异常）。同样，如果运行的线程调用了一个优先级为2的某个线程的stop()方法，而下一次CPUScheduler选择了这个线程并且设置它的优先级，也会产生一个异常。

因此我们应当将所有这些对setPrioprity()方法的调用放在一个try/catch子句中，使得在这些情况发生时我们可以截获这些异常。这意味着要修改所有调用 setPriority()方法的代码：

```
public void removeThread(Thread t) {
    try {
        t.setPriority(5);
    } catch(Exception e) {}
    threads.delete(t);
    synchronized(this) {
        if (current == t)
            current = null;
    }
}

public void run() {
    while (shouldRun) {
        ...
        try {
            current.setPriority(4);
        } catch (Exception e) {
            removeThread(current);
        }
        ...
```

```
                synchronized(this) {
                    if (current != null)
                        try {
                            current.setPriority(2);
                        } catch (Exception e) {
                            removeThread(current);
                        }
                }
                ...
            }
        }
```

run()方法的第一个try子句使我们在运行于优先级4的线程退出时得到保护；第二个try子句使我们在一个线程从另外一个线程中退出时得到保护。要注意的是在两种情况下，我们都需要将该线程从链表中删除，这也意味着我们还要在removeThread()方法中使用catch子句。

调整3：与线程安全相关的更多修改

我们已经使得CPUScheduler类的方法是线程安全的了，但是类本身呢？如果两个线程试图创建CPUScheduler会有什么结果？结果是令人胡涂的：我们会有两个调度线程，它们之间会竞争对其他线程的调度。因此我们要确保只能创建一个CPUScheduler实例。这是通过在CPUScheduler中创建一个静态变量来实现的，在创建CPUScheduler实例时要检查该变量来确保没有其他的实例存在。因为不能使得构造函数同步，所以需要引入一个同步方法来访问这个静态变量。该类的构造函数和相关代码如下所示：

```
public class CPUScheduler extends Thread {
    private static boolean initialized = false;
    private synchronized static boolean isInitialized() {
        if (initialized)
            return true;
        initialized = true;
        return false;
    }

    public CPUScheduler(int t) {
        if (isInitialized())
            throw new SecurityException("Already initialized");
        threads = new CircularList();
```

```
            timeslice = t;
        }
    }
```

调整 4：设计一个退出机制

如果 CPUScheduler 控制下的所有线程都结束了，它也会退出。如果所有的程序都像 TestThread 类一样，在运行的一开头就定义了所有要执行的任务，CPUScheduler 的这种实现是没有什么问题的。但是如果想在 TCPServer 中使用 CPUScheduler会怎么样呢？当前的CPUScheduler实现在那种情况下是不能正常运行的：当没有客户连接到TCPServer时，CPUScheduler就会退出，这使得以后连接到 TCPServer 的客户都不会进行分时间片调度了。

实际上，我们需要将 CPUScheduler 作为一个守护线程来运行，并且要调整其 run() 方法的逻辑。这样改是有意义的：当程序中有能够被调度的线程时，CPUScheduler是有用的。对TCPServer这个例子来说，是要保证系统中至少有一个其他线程：TCPServer 的监听线程。当用户连接到 TCPServer 时，该监听线程会创建线程然后让 CPUScheduler 进行调度。实现了分时间片调度的 TCPServer 的实现如下所示：

```
import java.net.*;
import java.io.*;

public class CalcServer {
    public static void main(String args[]) {
        CalcRequest r = new CalcRequest();
        try {
            r.startServer(3535);
        } catch (Exception e) {
            System.out.println("Unable to start server");
        }
    }
}

class CalcRequest extends TCPServer {
    CPUScheduler scheduler;
    CalcRequest() {
        scheduler = new CPUScheduler(100);
        scheduler.start();
    }
```

```
    void doCalc(Socket s) {
    }

    public void run(Socket s) {
        scheduler.addThread(Thread.currentThread());
        doCalc(s);
    }
}
```

每次当CalcRequest类的run()方法被调用时，就会有一个新的线程，因此我们需要将它加入到在该类的构造函数中创建的CPUScheduler中去。当没有线程需要被调度时（简单地说就是不存在客户连接时），只要能够保证CPUScheduler不会退出，我们就拥有了一个分时间片计算服务器。在CalcServer的一个活动会话期间，存在如下线程:

一个监听线程

该线程等待客户连接并且创建客户线程。

零个或者多个客户线程

这些线程代表连接的客户进行计算。

CPUScheduler线程

这个守护线程进行线程调度。

我们可以通过将服务器的shouldRun标志设置为false来安全地关闭CalcServer，最后所有的客户线程也会完成它们的计算任务并且结束。当所有的客户线程结束后，只有守护线程CPUScheduler存在于程序中，此时程序就会结束。

我们需要改变CPUScheduler，使得它在没有线程需要被调度时不是退出，而是等待更多的线程到来。下面就是修改过的全部代码（因为我们已经完成了所有的实现，因此这里是全部的代码）:

```
public class CPUScheduler extends Thread {
    private CircularList threads;
    private Thread current;
    private int timeslice;
    private static boolean initialized = false;
    private boolean needThreads;
```

```
private static synchronized boolean isInitialized() {
    if (initialized)
        return true;
    initialized = true;
    return false;
}

public CPUScheduler(int t) {
    if (isInitialized())
        throw new SecurityException("Already initialized");
    threads = new CircularList();
    timeslice = t;
    setDaemon(true);
}

public synchronized void addThread(Thread t) {
    t.setPriority(2);
    threads.insert(t);
    if (needThreads) {
        needThreads = false;
        notify();
    }
}
```

```
public void removeThread(Thread t) {
    threads.delete(t);
    synchronized(this) {
        if (t == current)
            current = null;
    }
}

public synchronized void run() {
    setPriority(6);
    while (true) {
        current = (Thread) threads.getNext();
        while (current == null) {
            needThreads = true;
            try {
                wait();
            } catch (Exception e) {}
            current = (Thread) threads.getNext();
        }
        try {
            current.setPriority(4);
        } catch (Exception e) {
            removeThread(current);
            continue;
        }
```

```
                try {
                    wait(timeslice);
                } catch (InterruptedException ie) {};
                if (current != null) {
                    try {
                        current.setPriority(2);
                    } catch (Exception e) {
                        removeThread(current);
                    }
                }
            }
        }
    }
```

我们对构造函数进行了一些调整：将该线程作为守护线程来运行。要注意的是run()方法也做了一点改变：当我们试图从链表中取出一个线程时，是通过循环来取出一个可用线程的。如果链表中没有线程，就会进入等待状态直到链表中有一个为止。这要求我们在addThread()方法中增加一个标志，通过它来表示当一个线程被加入时是否需要向CPUScheduler线程发出通知信号。

另外，我们还将run()方法本身声明为一个同步方法，并且调用wait()方法而不是sleep()方法。这个例子违反了不要将run()方法声明为同步方法的原则，但这样做也是有道理的：实际上在该方法中等待的时间要多于运行时间。因为处于等待状态时会释放所拥有的锁，因此同步该方法是没有什么问题的。

调整5：占用CPU较少的线程

如果当前运行线程阻塞了会有什么结果？以我们的TestThread程序为例，如果当前运行线程进入阻塞状态，会有什么事情发生？在一开始，线程的状态如下：

```
优先级 2: t2 -> t3 -> NULL
优先级 4: t2 -> NULL
    阻塞: CPUScheduler -> NULL
```

线程t2是当前运行线程。当CPUScheduler睡眠时，t2就会运行。如果t2因为某种原因进入阻塞状态，则线程状态如下：

```
优先级 2: t3- > t1 -> NULL
优先级 4: NULL
    阻塞: t2-> CPUScheduler -> NULL
```

这意味着尽管 t 3 处于优先级为 2 的状态，它也会成为当前运行线程。当CPUScheduler被唤醒时，它将t2的优先级重新设置为2，将t3的优先级设置为4，然后进入睡眠状态。这时线程的状态是:

```
优先级 2: t1->NULL
优先级 4: t3->NULL
    阻塞: t2 -> CPUScheduler -> NULL
```

因为CPUScheduler没有办法检测到t2是否是阻塞的，所以有时候它会将t2的优先级设置为4。除此以外，事情都很正常。而此时，Java的调度程序会再次从优先级为2的线程中选取一个作为当前运行线程。

因为每一个线程都会得到一些时间片来运行，所以我们的代码没有问题。但是当CPUScheduler睡眠时，会有一段很短的时间使得优先级为4的线程阻塞而优先级为2的线程成为当前运行线程。实际上，因为在优先级为4的线程阻塞到优先级为6的线程唤醒之间有一段时间，使得这个优先级为2的线程偷窃了一些CPU时间。

因为一旦CPUScheduler被唤醒，我们又会进入我们所希望的线程状态了，所以这种情况的发生不是一个大的问题。当然如果我们能够知道优先级为4的线程进入阻塞状态，就可以防止这种偷窃CPU时间的情况发生。但是，在一个本地线程平台上，在某些时候我们不能够阻止一个低优先级的线程运行。这与我们现在正在讨论的现象有某种相似性。因此，就算是解决这个问题，也不能完全避免这种情况的发生。

在一个绿色线程平台上，可以想到在CPUScheduler中创建一个优先级为3的线程。当优先级为4的线程阻塞时，该线程就成为了当前运行线程；该线程可以通知优先级为6的线程醒来并且进行调度工作。要注意的是，对一个本地线程平台，这个方案不起作用：当优先级为4的线程还没有阻塞时，优先级为3的线程就有可能运行了。并且在Windows平台上，优先级3和优先级4共享同一个底层操作系统优先级。当然我们可以修改这些线程的优先级来避免优先级的重叠，例如可以让调度程序运行在优先级8上，而让目标线程运行在优先级6上。但是我们已经看到，将一个占用CPU较多的线程的优先级设定为高于系统的默认优先级（特

别是将其优先级设定为系统GUI线程运行的级别）并不是一个好的方法，并且这样做也不能阻止优先级为3的线程在目标线程没有阻塞时运行。

就算是在绿色线程平台上，该问题也不可能被完全解决。如果所有被调度的线程都被阻塞，则优先级为3的线程会不停地运行，消耗了大量的CPU资源而没有做什么实际工作。在本书的第一版中，我们通过挂起优先级为3的线程来解决这一问题，但是现在suspend()方法已经被废弃了，因此就不能再使用这种解决方法了。因为这种解决方法带来的好处并不大，所以不使用它也没有什么问题。

这个例子再次说明了我们一贯的观点：Java调度程序允许你对线程调度进行一些控制，但是不能进行绝对的控制。

作业调度

我们通过研究一个作业调度来总结我们的例子。与循环调度不同，作业调度和防止线程饥饿或者进行公平调度之间是没有关系的。作业调度关心的是何时运行一个可运行对象，而不是怎样运行一个可运行对象。

有很多进行作业调度的应用程序。字处理程序可能每五分钟要保存一下文档来防止数据丢失。一个备份程序可能每天都要进行增量备份；而且每个星期都要进行一次完全备份。在第二章中的Animate applet中，需要每秒钟都重新生成一个重画请求。当时，是通过一个定时器线程重复调用sleep()方法来实现这一点的。在那个例子中，对于重画请求的调度是很容易实现的，而且也只有这样一个简单的循环作业要调度。

对于更加复杂的作业调度，或者当一个程序有很多的作业需要调度时，使用作业调度程序就比为程序中的每一个作业都实现调度要容易得多。更进一步，在使用定时器线程的例子中，我们需要创建一个只处理作业的线程。如果有很多作业，则使用作业调度程序可能比让很多线程来调度它们自己要更好一些。这个专门进行作业调度的调度程序可以使得所有这些作业在其自身的线程中运行，也可以为了更好地利用底层平台提供的线程资源而使得这些作业在线程池中运行。

下面就是一个作业调度程序的实现:

```
import java.util.*;

public class JobScheduler implements Runnable {
    final public static int ONCE = 1;
    final public static int FOREVER = -1;
    final public static long HOURLY = (long)60*60*1000;
    final public static long DAILY = 24*HOURLY;
    final public static long WEEKLY = 7*DAILY;
    final public static long MONTHLY = -1;
    final public static long YEARLY = -2;

    private class JobNode {
        public Runnable job;
        public Date executeAt;
        public long interval;
        public int count;
    }
    private ThreadPool tp;
    private DaemonLock dlock = new DaemonLock();
    private Vector jobs = new Vector(100);

    public JobScheduler(int poolSize) {
        tp = (poolSize > 0) ? new ThreadPool(poolSize) : null;
        Thread js = new Thread(this);
        js.setDaemon(true);
        js.start();
    }

    private synchronized void addJob(JobNode job) {
        dlock.acquire();
        jobs.addElement(job);
        notify();
    }

    private synchronized void deleteJob(Runnable job) {
        for (int i=0; i < jobs.size(); i++) {
            if (((JobNode) jobs.elementAt(i)).job == job) {
                jobs.removeElementAt(i);
                dlock.release();
                break;
            }
        }
    }
```

```
private JobNode updateJobNode(JobNode jn) {
    Calendar cal = Calendar.getInstance();
    cal.setTime(jn.executeAt);
    if (jn.interval == MONTHLY) {
            // 这有一个小问题 (参见 java.util.calendar)
            cal.add(Calendar.MONTH, 1);
            jn.executeAt = cal.getTime();
    } else if (jn.interval == YEARLY) {
            cal.add(Calendar.YEAR, 1);
            jn.executeAt = cal.getTime();
    } else {
            jn.executeAt =
                    new Date(jn.executeAt.getTime() + jn.interval);
    }
    jn.count = (jn.count == FOREVER) ? FOREVER : jn.count - 1;
    return (jn.count != 0) ? jn : null;
}

private synchronized long runJobs() {
    long minDiff = Long.MAX_VALUE;
    long now = System.currentTimeMillis();

    for (int i=0; i < jobs.size();) {
        JobNode jn = (JobNode) jobs.elementAt(i);
        if (jn.executeAt.getTime() <= now) {
            if (tp != null) {
                tp.addRequest(jn.job);
            } else {
                Thread jt = new Thread(jn.job);
                jt.setDaemon(false);
                jt.start();
            }
            if (updateJobNode(jn) == null) {
                jobs.removeElementAt(i);
                dlock.release();
            }
        } else {
            long diff = jn.executeAt.getTime() - now;
            minDiff = Math.min(diff, minDiff);
            i++;
        }
    }
    return minDiff;
}

public synchronized void run() {
    while (true) {
        long waitTime = runJobs();
```

```
            try {
                wait(waitTime);
            } catch (Exception e) {};
        }
    }

    public void execute(Runnable job) {
        executeIn(job, (long)0);
    }

    public void executeIn(Runnable job, long millis) {
        executeInAndRepeat(job, millis, 1000, ONCE);
    }

    public void executeInAndRepeat(Runnable job,
                                    long millis, long repeat) {
        executeInAndRepeat(job, millis, repeat, FOREVER);
    }
    public void executeInAndRepeat(Runnable job, long millis,
                                    long repeat, int count) {
        Date when = new Date(System.currentTimeMillis() + millis);
        executeAtAndRepeat(job, when, repeat, count);
    }

    public void executeAt(Runnable job, Date when) {
        executeAtAndRepeat(job, when, 1000, ONCE);
    }

    public void executeAtAndRepeat(Runnable job, Date when,
                                    long repeat) {
        executeAtAndRepeat(job, when, repeat, FOREVER);
    }

    public void executeAtAndRepeat(Runnable job, Date when,
                                    long repeat, int count) {
        JobNode jn = new JobNode();
        jn.job = job;
        jn.executeAt = when;
        jn.interval = repeat;
        jn.count = count;
        addJob(jn);
    }

    public void cancel(Runnable job) {
        deleteJob(job);
    }
}
```

该作业调度程序的实现非常简单：它仅仅是遍历全部的请求作业（也就是存放在jobs向量中的元素），并且要么将那些要运行的作业放入一个线程池中去处理，要么启动一个新的线程来执行这个作业。另外，我们还需要算出该作业下次要运行的时间，并且等待那个时间的到达。这个处理过程就是这样循环往复。

为了完整起见，我们还对JobScheduler类增加了一点点复杂性。除了可以接受一个可运行对象以及何时执行该作业的时间外，我们还可以指定作业要运行的次数以及执行作业的时间间隔。因此，当运行完一个作业后，我们还要计算是否有必要执行另一次计算以及何时要执行下一次计算。

在JobScheduler类中，所有这些都是通过一个线程调用runJobs()方法来完成的。而updateJobNode()方法是用来确定该作业是否需要重复执行的。addJob()和deleteJob()方法用来向作业向量中增加/删除作业。JobScheduler类的大部分逻辑实际上是实现这些通过接口提供给程序员的选项和方法。

在JobScheduler类中有八个方法可供程序员使用：

public void execute(Runnable job)

用于只运行一次的作业。只是简单地运行该作业。

public void executeIn(Runnable job, long millis)

用于只运行一次的作业。在经过指定的时间后运行该作业。

public void executeAt(Runnable job, Date when)

用于只运行一次的作业。在指定的时间运行该作业。

public void executeInAndRepeat(Runnable job, long millis, long repeat)

public void executeInAndRepeat(Runnable job, long millis, long repeat, int count)

用来多次执行一个作业。这些方法在经过了参数millis指定的时间（以毫秒计）后运行作业。然后在经过了以参数repeat指定的时间（以毫秒计）后再次运行该作业。这一过程将重复运行以参数count指定的次数。如果参数count没有指定，则该过程会一直循环下去。

常量 HOURLY、DAILY、WEEKLY、MONTHLY 和 YEARLY 也可以作为 repeat 参数来使用的。其中 HOURLY、DAILY 和 WEEKLY 是为了方便而提供的。但是，因为每一个月的日数不一样，并且要考虑到闰年的问题，所以 MONTHLY 和 YEARLY 在我们的作业调度程序中是以不同的方法处理的。

public void executeAtAndRepeat(Runnable job, Date when, long repeat)

public void executeAtAndRepeat(Runnable job, Date when, long repeat, int count)

用来多次执行一个作业。这些方法在一个指定的时间运行作业。然后在经过了指定的时间间隔后再次运行该作业。这一过程将重复指定的次数。如果没有指定次数，该过程会一直循环继续下去。

这些方法也支持 HOURLY、DAILY、WEEKLY、MONTHLY 和 YEARLY 这些常量作为参数。

public void cancel(Runnable job)

删除指定的作业。因为一个作业可能已经执行完毕从而在调用 cancel() 方法前就已经从作业向量中删除了，所以当调用该方法时如果该作业并不在请求作业向量中，也不会产生错误。如果同一个作业多次被放到作业向量中，该方法只会删除第一个。

尽管这个方法集提供的方法已经很丰富了，但是对于某些操作系统上提供的作业调度程序来说，这些方法还是不够的。在那些系统上，程序员可以通过指定在星期几、几号、一年的第几个星期这样一些条件来来调度作业。

定义作业运行时间的条件一般也是这样指定的。例如，我们一般不会指定备份程序在某个特定的日子和时间运行，而是指定在每个星期的某一天运行（例如每个星期天的上午两点）。同样，我们是在每个月的一号和十五号发工资。因为程序设计应该符合真实世界的要求，所以我们需要修改作业调度程序对这些要求提供支持。

如何加强作业调度程序来支持要求的实现就作为读者的练习。但是，如果使用 Calendar 类，则这个要求不难实现。例如，通过使用这个类，我们可以很容易地从某一特定天开始，在每个星期的特定日子来执行一个作业：

```
public void executeAtNextDOW(Runnable job, Date when, int DOW) {
    Calendar target = Calendar.getInstance();
    target.setTime(when);
    while (target.get(Calendar.DAY_OF_WEEK) != DOW)
        target.add(Calendar.DATE, 1);
    executeAt(job, target.getTime());
}
```

有了这个增强的功能，我们就可以通过如下的方式来在星期天执行一个作业：

```
executeAtNextDOW(job, new Date(), Calendar.SUNDAY);
```

作业调度程序是否需要作为一个守护线程来运行呢？粗看起来，这是一个好主意。毕竟，如果没有用户线程就没有需要调度的作业。但是可能在作业向量中有曾经执行过的作业存在并且在等待下一次执行。因为这些作业是不会自己对自己进行调度的，所以当它们在作业向量中等待时，没有线程与它们相关联。此时，就可能出现还有作业在等待调度，而所有的用户线程都已经退出的情况。在这种情况下，如果我们的作业调度程序是作为一个守护线程来配置的，它就会在还有作业等待执行的情况下退出。

如果使用我们在第六章中开发的DaemonLock类，情况会好一点：我们可以使作业调度程序作为一个守护线程运行，也可以保证只有在没有作业要调度并且没有用户线程存在的情况下作业调度程序才会退出。我们所要做的就是当作业加入到调度程序中去的时候，要获取一个守护锁。而当作业从调度程序中删除时才释放该锁。但是，因为线程池中的线程不是守护线程，所以这个方法只是在作业调度程序没有使用线程池时才有用（也就是说，每一个作业都是使用一个新的线程来运行的）。

总结

在本章中，我们展示了四种调度技术。其中最有用的是关于线程池的概念：很多线程处于空闲状态，等待有任务可以运行。当虚拟机中只有有限数目的线程，并且我们想最大限度地使用机器的CPU能力时，线程池技术是很有用的。

我们还展示了两种可以用来使用（或者限制）循环调度的技术。这两种技术都不是完全令人满意的：尽管 SimpleScheduler 可以在大多数平台上运行，但并不能保证可以在所有的平台上运行。而 CPUScheduler 在其控制的线程进入阻塞状态时会有一些反常的行为。但是，对于占用 CPU 较多的线程而言，如果你想影响 Java 虚拟机和后台操作系统提供的调度行为，这些技术还是很有用的。

最后，我们还展示了当不对每一个线程都使用定时器线程时，如何进行批处理性质的作业调度。这不仅仅是一个有用的作业调度机制，它还展示了如何使用我们曾经开发的其他技术来编写一个真正的线程工具。

第八章

和同步相关的高级主题

本章内容：

- 同步术语
- 预防死锁
- 锁饥饿
- 非线程安全的类
- 总结

在这一章中，我们将了解一些和同步相关的高级主题。当编写一个要使用多个线程的Java程序时，和数据同步相关的问题是程序设计中最困难的部分，而产生数据同步错误的原因与程序中事件发生的顺序相关，因此很难被检查出来。我们经常会发现由数据同步造成的错误会因为计时的不同而被掩盖。程序正常运行时可能碰到的数据错误，当在调试器中运行时或者是在程序中加上一些调试语句时，由于程序的计时是完全不一样的，原来出现的数据错误可能不再出现了。

同步术语

一个有在某种线程系统上的工作经历的程序员，一般习惯于使用该系统的特定术语来描述我们在本章讨论的某些概念，而那些没有在这些线程系统上工作过的程序员不会了解这些术语。因此我们先将大家习惯使用的术语和本章中使用的术语进行对比。

屏障（barrier）

屏障是多个线程的集合点：在所有的线程都到达这一点以前，任何一个线程都不能越过这一点。Java没有屏障类，但是我们已经在第五章中实现了这个类。

条件变量（condition variable）

条件变量实际上并不是锁：它是和锁相关联的变量。条件变量经常在数据同步的情况下使用。条件变量一般会提供功能类似于Java的等待和通知机制的API；在这种机制中，条件变量实际上就是被锁保护的对象。我们在第五章中实现了条件变量。

临界区（critical section）

临界区和同步方法或者同步块是一样的。但是临界区不能像同步方法或同步块那样进行嵌套。

事件变量（event variable）

事件变量就是条件变量。

锁（lock）

这个术语指一个特定的线程进入一个同步方法或者同步块的权力。我们说进入了这样一个方法或者代码块的线程就是获取了锁。正如我们在第三章中讨论的那样，锁是和一个特定的对象实例或者类相关联的。

监视器（monitor）

在不同的线程系统中有着不同意义的一个通用同步术语。在某些系统中，监视器就是锁；但是在其他系统中，监视器类似于等待和通知机制。

互斥（mutex）

锁的另外一种称呼。互斥不能像同步块或者同步方法那样嵌套，而且一般而言，在操作系统级别上互斥可以在不同的进程间使用。

读－写锁（reader-writer lock）

一个可以被多个只对共享数据进行读操作的线程同时拥有，或者是只被一个要写这些共享数据的线程拥有的锁。在Java中没有读－写锁的概念，但是我们会在本章的后面开发出一个读－写锁类。

信号量（semaphore）

在不同的计算机系统中对于信号量的使用是不一致的。许多程序员像使用Java中的锁一样使用信号量来锁定一个对象，这种方法使得信号量等同于互斥。另外一个更加复杂的使用方法是利用和信号量相连的计数器来实现对于临界区中代码的嵌套进入。在这种使用方法中，Java中的锁等同于信号量。

信号量还用来获取资源（而不是代码）的访问权限；在第四章的例子中，ResourceThrottle 类就实现了这种类型的信号量。

预防死锁

在任何线程系统中，因为线程竞争同一个锁集合而造成的死锁都是最难解决的问题。实际上，这是一个我们没有办法解决或者说是没有方法来试图解决的问题。因此我们只是试图使大家对死锁有更深刻的理解，并且提供一些防止死锁发生的原则。但是防止死锁发生是Java程序员的责任。Java虚拟机是不能代替你进行死锁预防或者是死锁检测的。

我们通过下面一个模拟厨房如何工作的例子来看看死锁。当厨师要做饼干时，他要通过量杯来将合适份量的配料放入碗中；当厨师要做煎鸡蛋时，他要拿一个碗来打鸡蛋，然后用量杯来将打完的鸡蛋分到每一个煎蛋器中。这是厨师工作的典型过程。如果只有一个顾客，这个流程是没有一点问题的。如果他有两个顾客，一个希望要饼干，而另外一个要煎蛋，我们就会发现自己进入死锁了：煎蛋所需要的量杯被用来做饼干，而做饼干所需要的量杯又在做煎蛋（注 1）：

```
public class Kitchen {
    static MeasuringCup theCup;
    static Bowl theBowl;

    public void makeCookie() {
        synchronized(theCup) {
            theCup.measureOut(1, theFlour);
            synchronized(theBowl) {
                theBowl.putIngredients(theCup);
                theBowl.mix();
            }
        }
    }
```

注 1：显然，本节中的代码例子并不是一个完整的例子，除了缺少我们所提到的所有方法和类以外，还缺少其他一些有用的方法。例如，我们的类不包括汤的配料，因为多线程的配料会糟蹋了肉汤。

```
        public void makeOmelette() {
            synchronized(theBowl) {
                Eggs e[] = getBrokenEggs();
                theBowl.putIngredients(e);
                theBowl.mix();
                synchronized(theCup) {
                    theCup.measureOut(theBowl);
                }
            }
        }
    }
```

和上面的死锁例子一样，这个例子也很简单。但是更加复杂的死锁也遵循同样的原则：尽管它们更加难以检测，但都是两个或者多个线程试图获取对方的锁。

死锁是由多个类之间以一种隐秘的顺序来相互调用它们的同步部分（包括同步方法或者同步块）而造成的，所以是很难检测的。一个例子是：假设有从A到Z的26个类，并且类A的同步方法调用类B的同步方法，类B的方法调用类C的方法，依次类推，直到类Z调用类A的方法。这种情况造成的死锁与上面makeCookie()和makeOmelette()方法造成的死锁是同一种类型的。但是程序员很难通过检查代码来检测到这个死锁。

但是，仔细地检查代码还是目前惟一可能用来检测死锁的方法。Java虚拟机在运行时是不会进行死锁检测的。尽管有通过检查源代码来检测潜在死锁的工具，但是到目前为止对于Java而言，还没有这样的工具。

避免死锁的最简单方法是遵循这样一个原则：一个同步方法永远不要调用另外一个同步方法。尽管这是一个经常被推荐的方法，但是因为下面的两个原因，这也不是一个理想的方法：

- 这是不切实际的：许多有用的Java方法都是同步的，并且你会希望在你的同步方法中调用这些方法。例如，我们会在同步方法中调用Java的Vector类的addElement()方法。
- 要求过于严格。如果你要调用的一个同步方法不会调用另外一个同步方法，就不会产生死锁（这就是我们为什么可以在同步方法中调用addElement()

方法，因为addElement()方法不会调用其他同步方法)。一般来说，同步方法可以根据我们后面要阐述的原则来调用其他同步方法。

尽管如此，如果你能设法遵循这个原则，你的程序中就一定不会存在死锁。

还有一种经常使用的技术可以避免死锁，就是对与我们要使用的对象相关联但是更高一级的对象加锁。在上面的例子中，这就意味着锁住厨房而不是我们要使用的每一个工具。这使得我们的方法被这样同步：

```
public class Kitchen {
    public synchronized void makeCookie() { ... }
    public synchronized void makeOmelette() { ... }
}
```

当然，我们也可以不锁住所有的东西。我们可以对量杯和碗创建一个BusyFlag，而在希望获取其中任何一个时先获取这个锁。我们也可以在程序中规定如下的原则：无论是使用量杯还是碗，都要先获取碗所对应的锁。这些根据锁住多个对象的理论而使用的不同方法都会碰到锁粒度这个问题。在后面我们会讨论锁粒度。该技术产生的问题是因为锁粒度大小不合适。如果用kitchen类来同步方法，就会使得不可能有一个以上的厨师同时使用厨房：而使用多线程的目的就是为了允许多个厨师同时使用厨房。如果我们的设计是正确的，则一定有原因使得我们试图去获取多个锁而不是一个全局的锁。如果为了避免死锁而违反这个设计方案，就有点矫枉过正了。

避免死锁的最有用原则是确保以同样的顺序来获取锁。在上面的例子中，这就意味着要保证获取碗对应的锁一定要发生在获取量杯对应的锁之前（或者相反，只要保持一致即可）。这暗示了在类之间存在着一种锁的层次结构。锁层次和Java类的层次无关：它是对象的层次关系而不是类的层次关系。

更进一步说，锁的层次关系与类的层次关系是完全无关的：类MeasuringCup和类Boul可能在类层次关系中是兄弟关系，但是在锁层次关系中，它们必须是上下级关系。锁的层次关系是一个队列而不是一个树结构：锁层次关系中的每一个对

象都必须有且只有一个父对象（如同Java的类层次关系），而且必须有且只有一个儿子。

如果你正在开发一个复杂的Java程序，则在定义类层次关系的同时也定义锁层次关系不失为一个好主意；图8-1显示了一个层次关系的例子。但是因为对于锁层次关系没有什么强制的机制，所以它是依赖于程序员的编程经验来保证其层次关系的好坏的。

在上面的例子中，通过保证在获取量杯对应的锁之前一定要获取碗对应的锁，我们就可以防止死锁的产生。这样，即使我们只想获取量杯，也需要先获取碗对应的锁。下面就是对上面例子修改后的结果：

```
public void makeCookie() {
    synchronized(theBowl) {
        synchronized(theCup) {
            theCup.measureOut(1, theFlour);
            theBowl.putIngredients(theCup);
            theBowl.mix();
        }
    }
```

如果你的Java程序中使用了标准的Java同步技术，则遵循锁的获取层次关系来编程是防止死锁发生的最好方法。

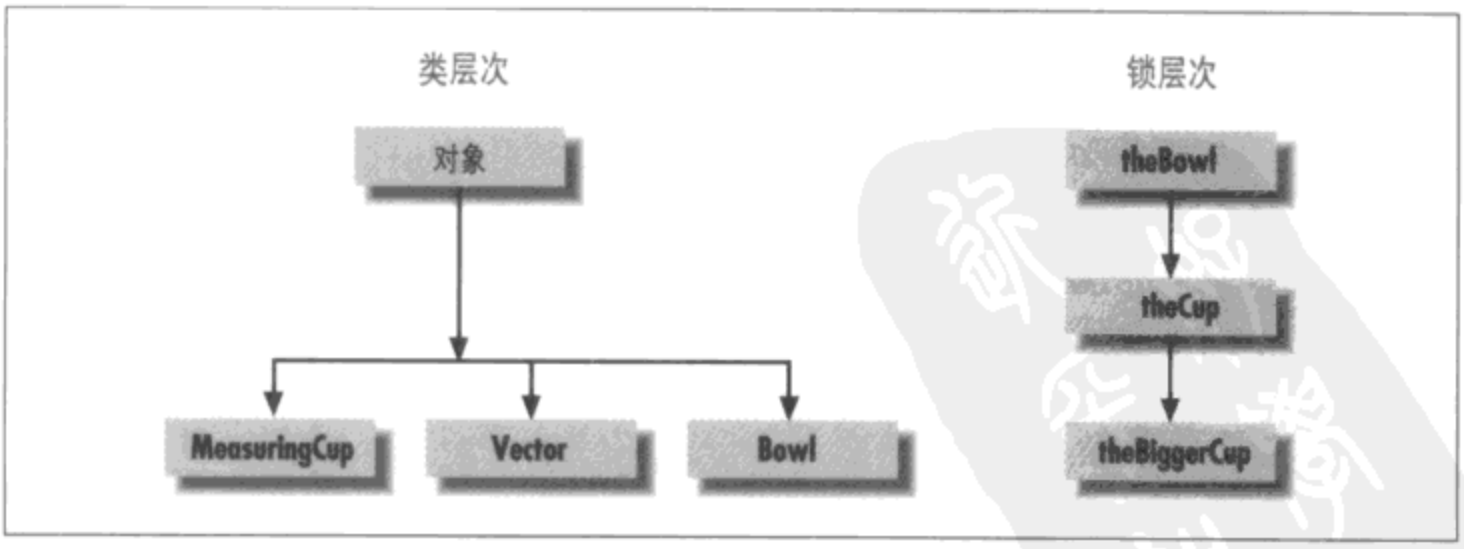

图8-1：类和锁的层次关系图

如果使用我们开发的BusyFlag类会有什么结果呢？它能不能防止死锁的发生？从某一点上来说，答案是肯定的。使用 BusyFlag 类使得 Java 程序增加了某种复杂性，并且会引入标准的 Java 同步技术不会产生的新类型的死锁。但是 BusyFlag 类也使我们可以建立更复杂的抵御死锁的机制，这在某种情况下是有用的。

BusyFlag 类中帮助我们防止死锁的是 tryGetBusyFlag()方法。在标准的 Java 同步调用中，没有类似的测试锁状态的概念：标准的Java线程仅仅试图去获取锁，而在获取该锁之前会一直处于阻塞状态。BusyFlag 类允许我们测试可否获取锁，并且可以在标志是忙时试图进行恢复。

下面是用 BusyFlag 重写的厨房的例子：

```
public class Kitchen {
    static MeasuringCup theCup;
    static Bowl theBowl;
    static BusyFlag theCupFlag, theBowlFlag;
    public void makeCookie() {
        theCupFlag.getBusyFlag();
        theCup.measureOut(1, theFlour);
        theBowlFlag.getBusyFlag();
        theBowl.putIngredients(theCup);
        theBowl.mix();
        theBowlFlag.freeBusyFlag();
        theCupFlag.freeBusyFlag();
    }

    public void makeOmelette() {
        theBowlFlag.getBusyFlag();
        Eggs e[] = getBrokenEggs();
        theBowl.putIngredients(e);
        theBowl.mix();
        theCupFlag.getBusyFlag();
        theCup.measureOut(theBowl);
        theCupFlag.freeBusyFlag();
        theBowlFlag.freeBusyFlag();
    }
}
```

到目前为止，我们用BusyFlag类代替了Java的标准同步块。这还是会造成死锁。但是，如果我们重写 makeCookie()方法，就会更进一步：

```
public void makeCookie() {
    theCupFlag.getBusyFlag();
    theCup.measureOut(1, theFlour);
    if (theBowlFlag.tryGetBusyFlag()) {
        theBowl.putIngredients(theCup);
        theBowl.mix();
        theBowlFlag.freeBusyFlag();
    }
    else {
        // 进行其他的处理
    }
    theCupFlag.freeBusyFlag();
}
```

此时，我们在试图获取与碗对应的BusyFlag时通过测试其标志值是否为闲来防止死锁的发生。如果标志为闲，我们就可以获取它并且继续运行。甚至，即使这时又有一个厨师线程开始煎鸡蛋，因为直到我们释放了碗和杯子对应的锁之前该线程是一直阻塞的，所以也不会发生死锁。

在makeCookie()方法的else子句中实现的逻辑决定我们是否可以真正防止死锁的发生。也许在else子句中我们可以使用另外一个碗，但是这实际上并没有什么好处：如果那个碗是被某个执行makeTrifle()方法的线程使用会怎么样呢？else子句必须做如下两件事情之一：它要么做一些不需要锁住任何器具的事情，要么释放量杯的BusyFlag。假设有一大块蜡纸，则我们可以将面粉放在蜡纸上，然后再等待碗。

```
public void makeCookie() {
    theCupFlag.getBusyFlag();
    theCup.measureOut(1, theFlour);
    if (theBowlFlag.tryGetBusyFlag()) {
        theBowl.putIngredients(theCup);
        theBowl.mix();
        theBowlFlag.freeBusyFlag();
        theCupFlag.freeBusyFlag();
    }
    else {
        WaxedPaper thePaper = new WaxedPaper();
        thePaper.emptyOnto(theCup);
        theCupFlag.freeBusyFlag();
        theBowlFlag.getBusyFlag();
        theBowl.putIngredients(thePaper);
        theBowl.mix();
```

```
            theBowlFlag.freeBusyFlag();
        }
    }
```

因为关键字synchronized不允许我们随心所欲地释放锁，所以这种解决方法不能和它一起工作。要想使用关键字synchronized，我们就不得不一直使用蜡纸对象：

```
public void makeCookie() {
    WaxedPaper thePaper = new WaxedPaper();
    synchronized(theCup) {
        theCup.measureOut(1, theFlour);
        thePaper.emptyOnto(theCup);
    }

    synchronized(theBowl) {
        theBowl.putIngredients(thePaper);
        theBowl.mix();
    }
}
```

使用synchronized关键字的代码是很清楚的，也很容易理解和维护。但是在一个蜡纸是珍稀资源的世界中，使用BusyFlag的代码就比使用蜡纸的代码要好。在真实世界的代码中，这些稀有资源可能是指某种运行速度慢但是容易实现的算法、占用内存较多的操作以及类似的东西。

使用BusyFlag比使用锁层次关系更复杂。但是BusyFlag也有其优势：使用BusyFlag比使用有序锁更能增加程序的并行度。在使用BusyFlag的例子中，在做饼干的线程使用量杯的同时另外一个线程可以打鸡蛋。但是在使用有序锁的例子中，想煎蛋的线程必须等待做饼干的线程释放全部锁后才能够打鸡蛋。

因此在设计Java程序时，你要考虑到使用BusyFlag带来的好处是否大于它带来的复杂性。如果你使用锁层次关系，则好处是代码简单，而坏处是会失去一些并行性。我们建议先用最简单的方法来写出代码，然后修改那些成为运行效率瓶颈的地方，以解决其并行性问题。

另外一种类型的死锁

在上面最后一个使用BusyFlag的厨房例子中，我们提到过一种新的死锁类型，该死锁在使用Java关键字synchronized时是不会发生的。这是一个由于拥有锁的线程非正常退出而引起的问题。

下面简化我们的例子，使得该类只有一个同步化的方法。类的定义可能如下所示：

```
public class Kitchen {
    public synchronized void makeCookie() { ... }
}
```

现在我们有两个线程，一个正在执行makeCookie()方法，而另外一个阻塞在试图进入makeCookie()方法的过程中。在一般情况下，第一个线程完成makeCookie()方法然后退出这个方法，此时，第二个线程就获得进入makeCookie()方法的机会。

但是如果第一个线程在运行时碰到一个运行时异常并且退出，会产生什么结果呢？在大多数线程系统中，这会导致死锁。因为那个被停止运行的线程并不会自动释放其拥有的锁。在这些系统中，第二个线程因为不能获取锁来继续运行，将不得不永远等待下去。但是在Java中，当线程离开同步块的作用域后，即使是因为发生异常而离开，也会自动释放锁。因此在Java中，这种类型的死锁是不会发生的。

但是如果是使用BusyFlag类而不是Java的synchronized关键字，就有可能导致这种类型的死锁。在本例中，该方法的代码类似于：

```
public void makeCookie() {
    flag.getBusyFlag();
    // ... 做某些工作 ...
    flag.freeBusyFlag();
}
```

如果在运行过程中碰到一个运行时异常，则BusyFlag不会被释放。这就意味着第二个线程没有机会来进入做饼干的流程。要注意的是，Java要求你捕获除了运行时异常以外的全部异常，所以只有在发生了运行时异常时才会产生这种死锁问题。

运行时异常通常是没有办法进行恢复的灾难性错误，因此在其发生时是否释放BusyFlag是无关紧要的，但是我们现在并不采用这样的假设。

还有一个解决这个问题的方法：我们可以使用Java的finally子句来保证BusyFlag在任何情况下都会被释放。为了使得BusyFlag和关键字synchronized具有同样的锁语义，可以用如下的方式：

```
public void makeCookie() {
    try {
        flag.getBusyFlag();
        // ... 做某些工作 ...
    } finally {
        flag.freeBusyFlag();
    }
}
```

这样，对BusyFlag的作用就和synchronized相同了。很显然，在本章使用的例子中，我们都能够使用try/finally子句来保证当异常发生时，锁会被释放。但是在我们看过的其他例子中，这种方法就不一定奏效了。使用BusyFlag的一个好处就是可以在不是获取该锁的另外一个方法中释放该锁。如果使用了这种技术，我们必须意识到这种新类型的死锁仍然有可能发生。

顺便说一句，Java关键字synchronized不会导致这种类型的死锁发生不见得是一件好事情。当一个拥有锁的线程发生运行时异常时，其使用的数据就会处于不一致的状态。如果其他线程接着获取该锁并对这些不一致的数据进行处理，就会得到错误的结果。在我们的例子中，如果第一个线程在做巧克力饼干时碰到了一个运行时异常，它就会在碗中留下一些原料。在正常的情况下，makeCookie()退出前会将碗清理干净，但是如果碰到异常，就不会进行这些清理工作。这时如果第二个线程试图来做果仁饼干，则结果会是巧克力果仁饼干。

我们可以将清理碗的工作放在finally子句中，以防止这种情况的发生。但是如果这个方法也抛出一个异常，会出现什么结果呢？对于Java语言而言，这是没有办法解决的。实际上，正是这个问题导致了stop()方法不被推荐使用：stop()方法是通过抛出异常来工作的，这可能导致Java虚拟机中的某些关键资源处于不一致的状态。

因此，我们不可能完全解决这个问题。在很多情况下，当使用BusyFlag时，如果线程是以一种非预期的方式结束的，则我们情愿冒死锁的危险也不愿意让第二个线程来使用不一致的数据。假设在股票交易系统中有一个更新当前股票价格的线程碰到了运行时异常：如果另外一个线程使用这些错误的数据进行交易，则公司可能会损失几百万美元。在类似于上面例子的情况下，使用具有交易完整性功能的数据库系统可以防止线程不正常退出所带来的麻烦。在数据库系统中是通过两阶段提交协议来保证交易完整性的。当然也可以在你的Java程序中直接实现这样的功能，但是这种实现是比较困难的。

锁饥饿

只要有多个线程竞争一个稀有的资源，就会有发生饥饿的危险。在前面讨论CPU饥饿时曾提到过这个概念：如果使用了错误的调度选项，某些线程就永远不会有成为当前运行线程的机会，这会导致CPU饥饿。

同样，当使用关键字synchronized来获取锁时，理论上也会出现这种情况。锁饥饿就是指一个特定的线程试图去获取一个锁，但是因为该锁一直被另外一个线程拥有，导致它永远无法成功地得到该锁。很显然，导致这种情况发生的一个简单例子就是某个线程获取了一个锁并且再也不释放它：这使得所有其他试图获取该锁的线程都会失败，并且发生锁饥饿。但是锁饥饿还可能更加难以发现：如果有六个试图获取同一个锁的线程，而其中的五个锁各以20%的时间获取锁，那么第六个线程就会发生锁饥饿。

像CPU饥饿一样，锁饥饿不是大多数多线程的Java程序要考虑的问题。如果我们的Java程序可以运行无限长的时间，则当程序中的线程逐渐运行结束时，最终所有的线程都会获取其所需的锁。但是也正如CPU饥饿一样，锁饥饿会导致不公平：有时候，我们想保证每一个线程都会以合理的顺序获取锁，使得一个线程不必等到其他线程都运行结束后才有获取锁的机会。

下面考虑两个线程来竞争一个锁的例子。假设线程A周期性地获取该对象锁，如图8-2所示：

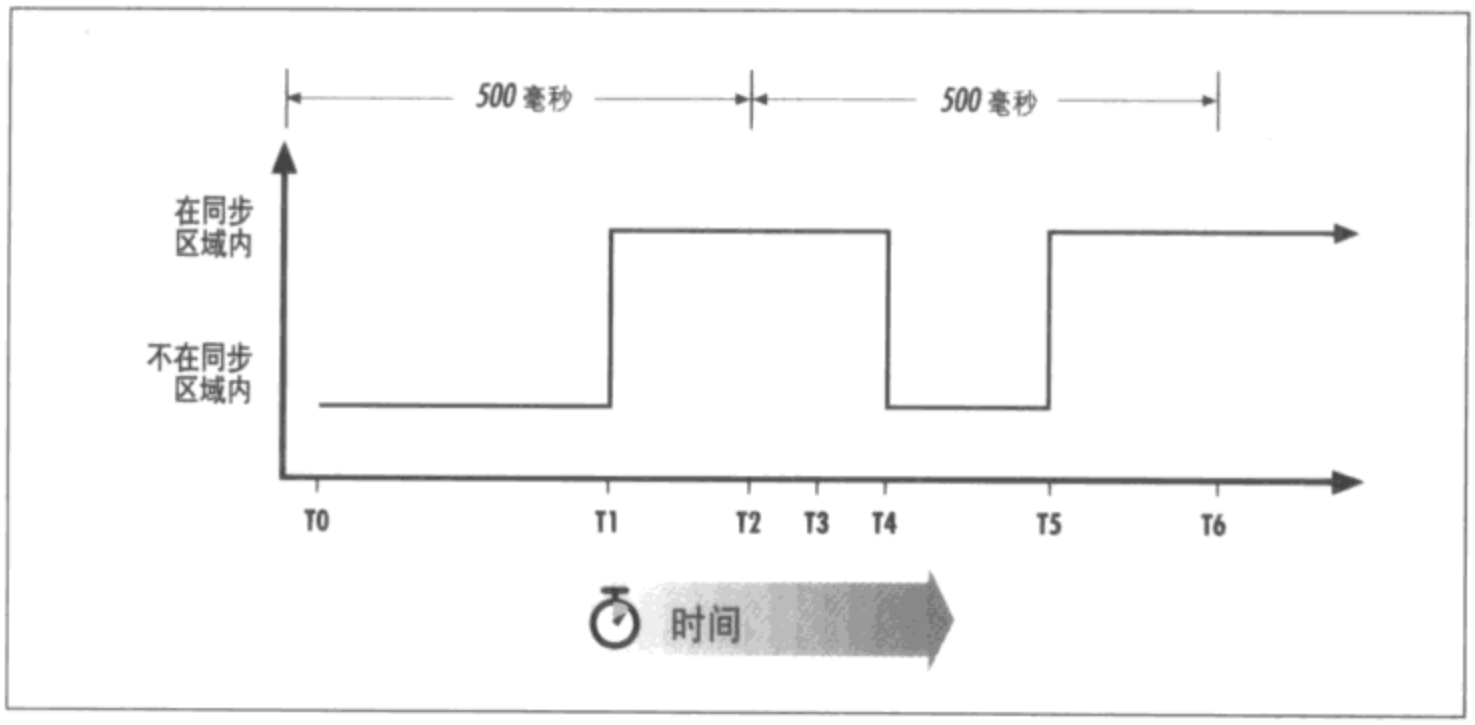

图 8-2：调用同步方法的图示；线程 A 周期性地调用一个同步方法

我们还假设这两个线程是在一个分时间片调度程序的调度下运行，该调度程序会每隔500毫秒选择一个新的线程作为当前运行线程。下面是在图上的不同时间点处发生的事情：

T0

在T0时刻，线程A和线程B都处于可运行状态，并且线程A成为当前运行线程。

T1

线程A仍旧是当前运行线程，并且当它进入同步块后就获取了对象锁。

T2

时间片到；这使得线程B成为当前运行线程。

T3

在线程B成为当前运行线程后不久就试图进入同步块。因为锁由线程A拥有，所以线程B进入阻塞状态，这也使线程A成为当前运行线程。线程A继续在其同步块中运行。

T4

线程A退出同步块。这使得线程B进入可运行状态，但是因为调度程序的时间片还没有到，因此线程A还是当前运行线程。

T5

线程 A 再次进入同步块并且获取对象锁。线程 B 保持在可运行状态。

T6

线程B再次成为当前运行线程。它立刻试图进入同步块，但是因为该同步块的锁还是被线程A所拥有，所以线程B又立刻进入阻塞状态。线程A再次成为当前运行线程。此时我们又处于和T3同样的状态。

这种情况有可能一直持续下去。因此尽管线程B经常会处于可运行状态，但是它从来都不会获取锁，而且也不会做什么实际工作。

很显然，这是一个极端的例子：每次调度都是在线程A拥有同步块锁的时刻进行的。对于这两个线程而言，就是线程A一直拥有锁，这是不可能的。但是如果有多个线程，就有可能会发生这样的情况：当某个特定线程被调度时，总会发现有其他的线程已经拥有该线程最想拥有的锁。

下面的代码经常会产生锁饥饿：

```
public class MyThread extends Thread {
    public void run() {
        while (true) {
            synchronized(someObject) {
                // ... 进行一些计算 ...
            }
        }
    }
}

public class Test {
    public static void main(String args[]) {
        MyThread t1, t2;
        t1 = new MyThread();
        t2 = new MyThread();
        t1.start();
        t2.start();
    }
}
```

乍看起来，我们可能会认为这段代码应该工作得很好：当线程1退出同步块后，线程2会立刻获得someObject的对象锁，因此这两个线程会轮换获得该锁。但是

正如我们上面所见到的，实际情况并不是这样的：除非正好是在同步块退出（释放锁）和下一次循环开始（获取锁）这短短的时间内进行了线程调度，否则线程2永远都不会获取到someObject的锁，也就永远不可能成为当前运行线程。当然，在其中加入对yield()的调用可以解决这个问题，但这不是一个通用的解决方法。

造成这个问题的原因有两个：

获取锁的过程不是排队的

当线程试图去获取一个锁时，它并不检查是否有其他的线程已经试图来获取该锁。更简单地说，就是是否有另外一个线程已经试图来获取该锁并且因为该锁被其他线程拥有而阻塞在该锁上。用伪代码来表示就是：

```
while (lock is held)
    wait for a while
acquire lock
```

当多个线程具有同样的优先级时，尽管有一个线程已经在等待某个锁，但该过程不能防止将锁给予另外一个线程的情况发生。

释放锁并不会发生线程调度

当一个锁被释放时，所有在该锁上等待的线程都由等待转为可运行状态。但是，这并不会发生线程调度。因此没有一个由等待状态转到可运行状态的线程成为当前运行线程；那个释放锁的线程仍然是当前运行线程（当然，我们还是假设所有的线程都处于同一个线程优先级上）。

尽管我们通过例子展示了锁饥饿的可能性，但是锁饥饿只在极少数情况下才会发生。实际上，只有当以下的条件都满足时才会发生锁饥饿：

多个线程竞争同一个锁

对于那些可能发生锁饥饿的线程而言，该锁就是一个珍稀资源。

而且在某个期间内，没有足够的CPU时间来运行全部的线程。在这段时间内，至少有两个处于可运行状态的线程，或者是有一个拥有该锁的线程进入了阻塞状态（这是最坏的情况）。

如果有足够的CPU时间来运行全部线程，并且拥有锁的线程并没有进入阻塞状态，则一个希望拥有锁的线程会在某个时候（只要它成为系统中惟一处于可运行状态的线程）拥有该锁。

我们关注于这段时间内对锁的竞争

假如我们正在对一个大矩阵进行计算，则在计算开始的时候可能会出现多个线程竞争同一个锁和CPU的情况。但是因为我们关心的是计算的最终结果，所以就算一些线程暂时处于锁饥饿状态，我们也不会关心：反正最重要的是可以在同样的时间内得到最终结果。

像CPU饥饿一样，只有在关心某段时间内是否会不公平地分配锁时才会关注锁饥饿。

这些线程都具有相同的优先级

在上面讨论过的例子中，如果线程B具有比线程A高的优先级，那么在时间T4会发生什么呢？因为线程A释放锁而导致线程B从阻塞状态转换为可运行状态，线程B就会因为其优先级高而成为当前运行线程。

当然，如果线程A具有比线程B高的优先级，则线程B仍然不会得到成为当前运行线程的机会，但是，此时线程B是处于CPU饥饿而不是锁饥饿状态。

这些线程都必须处于循环调度程序的控制之下

如果这些具有同样优先级的线程不是处于循环调度程序的控制之下，则它们是处于CPU饥饿而不是锁饥饿状态。还要注意的是该循环调度程序不会调整线程的优先级，否则就会违背上一个规则。由第七章中的SimpleScheduler控制的线程和本地线程一样，都容易发生锁饥饿。

所有这些导致锁饥饿产生的条件都是因为线程在试图获取一个锁时仅仅会检查该锁是否被其他线程所拥有，而不是检查是否有其他线程已经在该锁上等待。因此，当我们发现可能会发生锁饥饿时，就需要一个和某个队列相关联的锁来保证将锁公平地分配给每一个等待获取该锁的线程。

该类的编写很简单：可以用Vector类来实现队列，然后我们只需编写允许其他线程获取/释放锁的方法。getBusyFlag()方法将请求放在队列中，freeBusyFlag()方法在锁被释放时通知队列中的第一个元素。

QueuedBusyFlag 类的实现如下:

```
import java.util.Vector;

public class QueuedBusyFlag extends BusyFlag {
    protected Vector waiters;

    public QueuedBusyFlag() {
        waiters = new Vector();
    }

    public synchronized void getBusyFlag() {
        Thread me = Thread.currentThread();
        if (me == busyflag) {
            busycount++;
            return;
        }
        waiters.addElement(me);
        while ((Thread) waiters.elementAt(0) != me) {
            try {
                wait();
            } catch (Exception e) {}
        }
        busyflag = me;
        busycount = 0;
    }

    public synchronized void freeBusyflag() {
        if (Thread.currentThread() != busyflag)
            throw new IllegalArgumentException(
                                    "QueuedBusyflag not held");
        if (busycount == 0) {
            waiters.removeElementAt(0);
            notifyAll();
            busyflag = null;
        }
        else busycount--;
    }

    public synchronized boolean tryGetBusyflag() {
        if (waiters.size() != 0 && busyflag != Thread.currentThread())
            return false;
        getBusyFlag();
        return true;
    }
}
```

尽管QueuedBusyFlag有着和BusyFlag类一样的接口，但是我们还是需要实现一些方法。如果一个线程试图获取一个锁，它就进入getBusyFlag()方法并且将自己放入waiters向量中。它在该向量中等待一直到成为该向量的第一个元素。类似地，当线程释放该锁时，它就将自己从waiters向量中删除，并且通知在向量中等待的其他线程来检查自己是不是第一个元素。

因为该实现依赖于notifyAll()方法来唤醒等待获取该锁的全部线程，所以这种实现不是很有效率。如果有30个线程在等待该锁，尽管其中只有一个线程会得到锁而其他29个线程将再次调用wait()方法，但是30个线程都会被唤醒。因此你可能只会在那些锁饥饿是一个大问题的情况下才使用该技术。我们也可以使用在第四章中讨论过的有目的性的通知方法来实现一个更有效率的方案，但是具体的实现还是留给读者作为练习吧。

因为它是一个BusyFlag，所以我们可以像下面这样来使用这个新的类：

```
public class DBAccess {
    private QueuedBusyFlag lock;

    public DBAccess() {
        lock = new QueuedBusyFlag();
    }
    public Object read() {
        Object o;
        try {
            lock.getBusyFlag();
            o = someMethodThatReturnsData();
            return o;
        } finally {
            lock.freeBusyFlag();
        }
    }

    public void write(Object o) {
        try {
            lock.getBusyFlag();
            someMethodThatSendsData(o);
        } finally {
            lock.freeBusyFlag();
        }
    }
```

这些代码没有什么可令人惊奇的：该例子的代码和使用标志BusyFlag的代码的惟一区别就是在该例子中对数据库的请求是确保线性的，而如果使用标准BusyFlag的话，请求是被随机满足的（依赖于底层平台）。

读－写锁

有时候，我们可能会花很长的时间从一个对象中读取信息。我们需要锁住该对象来保证读取的数据是一致的，但是我们并不需要阻止其他线程来同时读取该对象：如果所有的线程都是进行读操作的，而读操作并不改变数据，因此没有理由不让它们并行地进行读取。

实际上，只有在改变数据的时候才真正需要对数据加锁；也就是要对数据进行写操作的时候。对数据的改变可能使得那些读取数据的线程得到不一致的数据。到现在为止，基于每一个锁都只是被拥有一段不长的时间这样的理论，我们使用的锁都只允许一个线程来访问数据而不管线程是要读数据还是要写数据。

如果在很长的一段时间内线程都拥有锁，那么允许多个要读取数据的线程并行地进行读取操作，从而避免它们之间的竞争是很有意义的。当然，我们还是只允许一个线程来写数据，同时我们还必须保证在一个要进行写操作的线程对数据进行改变时不会有读线程正在读取数据。

我们以一个B树为例。该B树用来存放供多个线程进行查询的工作数据。依据在该树中存放的数据量的多少，在其中查询一个特定条目可能会需要很长的时间。该B树的接口可能如下所示：

```
public class BTree {
    public synchronized boolean find(Object o) {
        // 进行长时间的搜索工作，如果对象存在，则返回该对象
        // 否则返回空
    }

    public synchronized void insert(Object o) {
        // 进行长时间的插入操作
    }
}
```

该接口的问题是，如果两个线程同时调用 find()方法来查询数据，那么其中一个就会阻塞在等待获取锁上；该线程在第一个线程进行查询时一直是阻塞的。如果这两个线程是在一个分时间片调度环境中运行的话，则它们会因为是在竞争同一个锁而不能分时间片运行。如果它们是在一个具有多个CPU的机器上运行，则它们两个不可能同时在不同的处理器上运行。如果这个B树是在一个供多个客户使用的服务器上，我们就会期望线程以并行的方式调用 find()方法。

这就是为什么会引入读写－锁的原因。如果我们有一个允许多个线程同时读取数据的锁，就可以使用这样的接口：

```
public class BTree {
    RWLock lock;
    public boolean find(Object o) {
        try {
            lock.lockRead();
            // 进行长时间的搜索工作，如果对象存在，则返回该对象
            // 否则返回空
            return answer;
        } finally {
            lock.unlock();
        }
    }
    public void insert(Object o) {

        try {
            lock.lockWrite();
            // 进行长时间的插入操作
        } finally {
            lock.unlock();
        }
    }
}
```

现在，尽管该B树在同一时刻只能被一个线程改变，但是可以允许多个线程同时读取 B 树中的数据。

坏消息是Java API并没有提供读－写锁；好消息是编写自己的读－写锁并不是一件困难的事情。下面就来看看一个简单的读－写锁的实现：

```
import java.util.*;
```

```
class RWNode {
    static final int READER = 0;
    static final int WRITER = 1;
    Thread t;
    int state;
    int nAcquires;
    RWNode(Thread t, int state) {
        this.t = t;
        this.state = state;
        nAcquires = 0;
    }
}

public class RWLock {
    private Vector waiters;

    private int firstWriter() {
        Enumeration e;
        int index;
        for (index = 0, e = waiters.elements();
                                e.hasMoreElements(); index++) {
            RWNode node = (RWNode) e.nextElement();
            if (node.state == RWNode.WRITER)
                return index;
        }
        return Integer.MAX_VALUE;
    }

    private int getIndex(Thread t) {
        Enumeration e;
        int index;
        for (index = 0, e = waiters.elements();
                                e.hasMoreElements(); index++) {
            RWNode node = (RWNode) e.nextElement();
            if (node.t == t)
                return index;
        }
        return -1;
    }

    public RWLock() {
        waiters = new Vector();
    }

    public synchronized void lockRead() {
        RWNode node;
        Thread me = Thread.currentThread();
```

```
        int index = getIndex(me);
        if (index == -1) {
            node = new RWNode(me, RWNode.READER);
            waiters.addElement(node);
        }
        else node = (RWNode) waiters.elementAt(index);
        while (getIndex(me) > firstWriter()) {
            try {
                wait();
            } catch (Exception e) {}
        }
        node.nAcquires++;
    }
    public synchronized void lockWrite() {
        RWNode node;
        Thread me = Thread.currentThread();
        int index = getIndex(me);
        if (index == -1) {
            node = new RWNode(me, RWNode.WRITER);
            waiters.addElement(node);
        }
        else {
            node = (RWNode) waiters.elementAt(index);
            if (node.state == RWNode.READER)
                throw new IllegalArgumentException("Upgrade lock");
            node.state = RWNode.WRITER;
        }
        while (getIndex(me) != 0) {
            try {
                wait();
            } catch (Exception e) {}
        }
        node.nAcquires++;
    }

    public synchronized void unlock() {
        RWNode node;
        Thread me = Thread.currentThread();
        int index;
        index = getIndex(me);
        if (index  > firstWriter())
            throw new IllegalArgumentException("Lock not held");
        node = (RWNode) waiters.elementAt(index);
        node.nAcquires--;
        if (node.nAcquires == 0) {
            waiters.removeElementAt(index);
            notifyAll();
        }
```

```
        }
    }
```

该读－写锁的接口很简单：方法lockRead()用来获取读锁；方法lockWrite()用来获取写锁；而方法unlock()用来释放锁（只能够提供一个unlock()方法，具体的原因在后面解释）。和QueuedBusyFlag类一样，试图获取锁的线程会先存放在waiters向量中直到成为该向量中的第一个元素，但是此时对于“第一个”的定义和QueuedBusyFlag中的有些不同。

读－写锁就是一个锁

你可能会认为读－写锁是两个独立但是相关联的锁：一个读锁、一个写锁。可能是因为我们使用的称呼使你产生了这样的误解：我们一直使用读锁和写锁这两个概念，就好象有两个单独的锁一样。从逻辑层面上说这是对的，并且我们也会继续使用这两个词。但是实际上我们所使用的是一个锁。

因为我们需要跟踪每一个线程到底要获取什么类型的锁——是读锁还是写锁，所以就需要创建一个类来封装线程和其希望获取的锁的类型。这就是RWNode类；waiters队列存放RWNode类型的元素，而不是像QueuedBusyFlag类那样存放Thread元素。

除了对于“第一”的定义不同以外，获取读锁的逻辑和QueuedBusyFlag中的逻辑是一样的。对于读锁而言，“第一”就是指在waiters队列中在其前面没有其他请求写锁的请求。如果某个请求前面的所有请求都是希望获取读锁的，则这个请求可以获取锁。否则，就只能等待该请求成为队列的第一个元素。

对于写锁的请求则要严格得多：只有队列的第一个元素才能够获取该锁，这和在QueuedBusyFlag类中是一样的。

记录一个特定的线程已经有多少次获取锁的逻辑有一点点改变。在QueuedBusyFlag类中，我们用一个实例变量就可以记录该值。但是因为读锁可以被多个线程同时获取，我们就不能仅仅使用一个实例变量了。我们将nAcquires和每一个线程相关联。这也正是为什么两个获取锁的方法都会检查调用者是否已经和一个节点关联的原因。

我们的读－写锁类并没有“锁升级”的概念；如果你拥有一个读锁，就不能获取一个写锁。你必须显式地释放读锁后再去申请获取写锁；否则就会产生IllegalArgumentException异常。如果提供了“锁升级”的特性，那么类应该在申请写锁之前释放读锁。真正的锁升级是不可能的。

最后，我们的读－写锁还有一些辅助方法：firstWrite()方法用来在waiters队列中搜索第一个试图获取写锁的节点，getIndex()方法用来找出与调用线程相关联的节点对应的索引号。因为Vector类的indexOf()方法需要我们传递一个类型为RWNode的对象，但是我们所有的仅仅是一个线程，因此我们不能直接使用向量类的indexof()方法。

图8-3显示了通过waiters队列试图获取锁的状态。已经拥有锁的线程用白色背景表示，等待获取锁的线程用黑色背景表示。每一个节点都在试图获取一个读锁或者写锁。

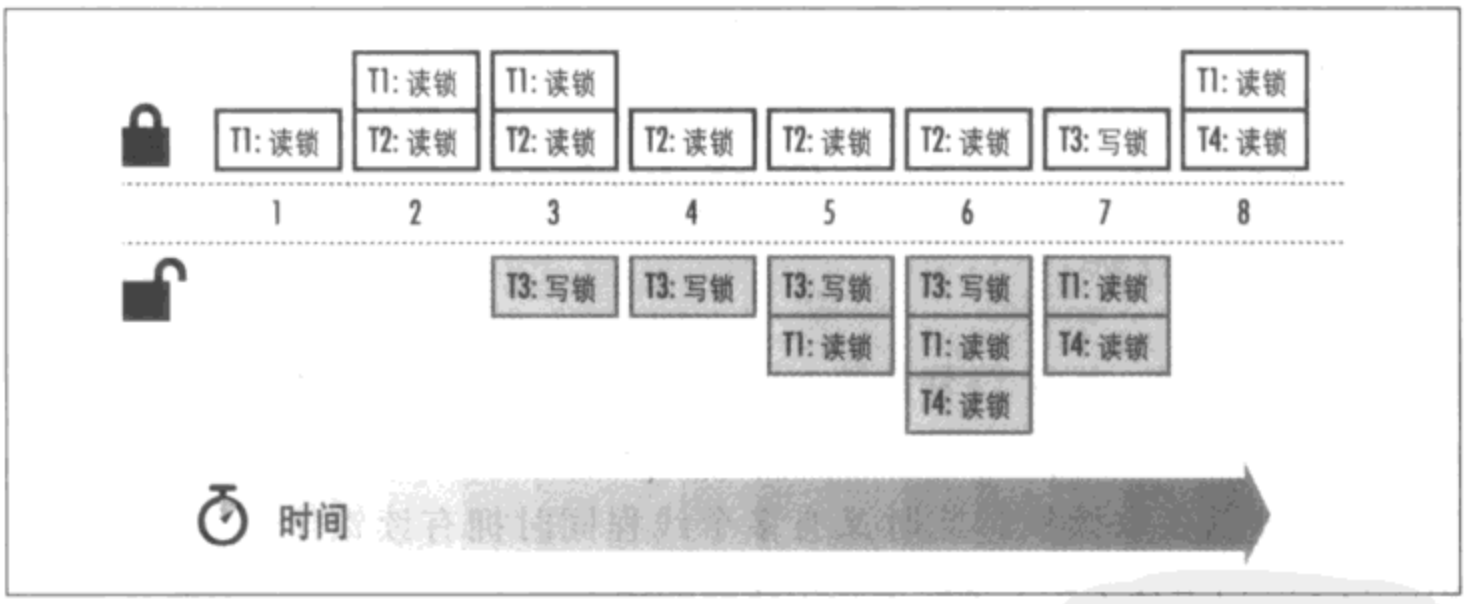

图8-3：读－写锁队列

在位置1处，线程T1已经获取了读锁。因为T1是waiters队列中的惟一一个线程，所以getIndex()方法返回0，而firstWriter()方法返回MAX_VALUE。因为该索引值小于第一个请求写锁的节点的索引值，所以对于读锁的请求被准许了。在位置2处，基于同样的逻辑，线程T2的请求也被准许了。此时两个线程同时获取了读锁。

在3位置处，线程T3试图去获取一个写锁。因为T3在waiters队列中的索引值是2，因此它不能获取该锁，而是执行lockWrite()中的wait()方法。接着在位置4处，线程T1释放其读锁，unlock()方法会调用notifyAll()，这会唤醒T3，但是因为T3在waiters队列中的索引值是1，它不得不继续执行wait()方法。

在位置5处，线程T1再次试图获取读锁。但是，因为它在队列中的索引值（2）大于第一个请求写锁的节点的索引值（1），线程T1不会立刻获取该锁，而是执行lockRead()中的wait()方法。此时我们可能想到因为T2拥有读锁，是否可以给予线程T1读锁，从而使得有多个读锁并行存在。但是如果我们实现这样的逻辑，就会使那些试图获取写锁的线程饥饿：可能有多个试图获取读锁的线程，并且尽管它们可能会经常地释放各自的锁，但是它们中的一个就可以阻止希望获取写锁的线程运行。这就是我们要将请求线程放在一个waiters队列中并且将它的索引值和队列中其他线程的索引值进行比较的基本原理。正如位置6处发生的情况一样。

在位置7处，线程T2释放锁，并且通知其他线程该锁被释放了。因为T3是请求写锁，并且索引值是0，lockWrite()方法就会将锁给予它，而其他在lockRead()中的线程继续执行wait()。

最后，在位置8处，线程T3释放锁。此时当剩下的两个线程被通知该锁为闲时，因为它们的索引值小于MAX_VALUE（当没有线程试图去获取写锁时就返回该值），它们就都可以获取读锁。此时又有多个线程同时拥有读锁了。这也是通过notifyAll()方法来轻易地唤醒多个线程的例子。

优先级倒置锁

在本节中我们要看到的最后一个例子是和优先级倒置相关联的饥饿。在我们曾经看到过的虚拟机中，优先级倒置是通过优先级继承来解决的。

但是如果我们希望使用BusyFlag来锁住程序中大块范围的代码呢？优先级继承会如何影响BusyFlag类呢？毫不奇怪，因为BusyFlag仅仅是模拟一个锁，并且只

是使用Java的同步锁来避免任务中的竞态条件，所以优先级继承对于BusyFlag的运行是没有任何影响的。一旦获取了BusyFlag，而且从getBusyFlag()方法退出了，则保护getBusyFlag()方法的同步锁也就释放了。对于Java虚拟机而言，此时是没有同步锁的。

一个拥有BusyFlag的低优先级线程不会因为一个高优先级的线程试图获取该标志而被Java虚拟机提升其优先级：因为它们不会试图在同一个时刻执行同一个同步方法，因此虚拟机并不知道它们在相互竞争。

可以很容易地实现一个支持优先级继承的BusyFlag类：

```
public class PriorityBusyFlag extends BusyFlag {
    protected int currentPriority;

    public synchronized void getBusyFlag() {
        while (tryGetBusyFlag() == false) {
            Thread prevOwner = getBusyFlagOwner();
            try {
                int curP = Thread.currentThread().getPriority();
                if (curP > prevOwner.getPriority()) {
                    prevOwner.setPriority(curP);
                }
                wait();
            } catch (Exception e) {}
        }
    }

    public synchronized boolean tryGetBusyFlag() {
        boolean succeed = super.tryGetBusyFlag();
        if (succeed)
            currentPriority = Thread.currentThread().getPriority();
        return succeed;
    }

    public synchronized void freeBusyFlag() {
        if (getBusyFlagOwner() == Thread.currentThread()) {
            super.freeBusyFlag();
            if (getBusyFlagOwner() == null) {
                Thread.currentThread().setPriority(currentPriority);
                notifyAll();
            }
        }
    }
}
```

PriorityBusyFlag的使用类似于BusyFlag类。区别在于如果请求一个BusyFlag的线程的优先级高于拥有该标志的线程，则拥有该标志的线程的优先级会提高；而当拥有该标志的线程释放BusyFlag时，它的优先级会被复原。

这个方法在功能上和支持优先级继承的本地线程系统是一样的。但是，虚拟机是在内部实现这些细节的。而我们所能做的最多就是在PriorityBusyFlag类中使用setPriority()方法来改变优先级。但是如果其他线程也改变这个线程的优先级，或者是线程自己也改变其优先级的话，这种做法就没有什么作用。

非线程安全的类

一个完美的世界可能不需要本节的内容：在那个世界中，每一个类都正确地进行了同步操作，使得它们可以被多个线程同时运行。因此你在使用这些Java类的时候可以不考虑同步的问题。

但是我们还是要回到真实的世界。在这里，你经常要使用一些非线程安全的类——那些由于没有考虑到会被多个线程同时使用而没有进行正确同步的类。存在非线程安全的类并不意味着在你的工作中不能创建线程安全的类：通过保证你自己的类都被正确地同步了，可以使这个世界更好一些。

在本节中，我们要了解两种和非线程安全类打交道的技术。

显式同步

在开始的时候，Java有这样一些集合类：Hashtable类、Vector类和其他一些提供对象集合的类。这些类的优点就是它们都是线程安全的：它们的方法都被正确地同步了，例如当两个线程同时向一个向量中插入对象时，不会导致向量内部的状态不一致。

Java 2通过引入很多集合类形成了集合的概念：这些类要么实现了Collection接口，要么实现了Map接口。有很多这样的类：例如HashMap和ArrayList类就提供了和原来的Hashtable和Vector类相似的语法。但是它们之间的区别是：大多数新的集合类都是非线程安全的。

实际上，这些类并没有一定的规则：虽然它们中的大多数都是非线程安全的，但是也有一些（例如原来的Hashtable类，它也实现了Map接口）是线程安全的。并且大多数非线程安全的类都有能力提供线程安全的实现。因此当你和一个由通用类型（例如Map）标识的对象打交道时，你就不清楚该对象是不是线程安全的。

同步了的集合

另一方面，Collection类都有这些方法：synchronizedCollection()、synchronizedMap()、synchronizedList()和synchronizedSet()。这可以将一个非线程安全的集合对象变成一个线程安全的集合对象。我们在此讨论的技术仅仅适用于集合类的非线程安全版本；我们就是使用这些非线程安全的集合类来作为例子的。

这对程序员而言是一个极大的负担，他们必须搞清楚一个特定的Map对象是否是线程安全的，同时，如果不是线程安全的，他们就必须在使用多个线程的情况下保证该对象能够被正确使用。最简单的做法就是显式地同步对该对象的所有访问操作。

```
import java.util.*;

public class ArrayTest {
    private ArrayList al;

    public ArrayTest() {
        al = new ArrayList();
    }

    public void addItems(Object first, Object second) {
        synchronized(al) {
            al.add(first);
            al.add(second);
        }
    }

    public Object get(int index) {
        synchronized(al) {
            return al.get(index);
        }
```

```
        }
    }
```

在该例子中所有对 ArrayList 的访问都是同步的；现在多个线程可以调用 ArrayTest的addItem()和get()方法，而不用担心会破坏数组列表的内部状态。

要注意的是我们使得数组列表是私有的。为了使得这种技术能够工作，我们必须保证没有人会偷偷摸摸地在没有进行同步的情况下使用该列表，而实现这一点的最简单方法就是将列表隐藏在使用它的对象中。这样，我们就只需考虑在我们的 ArrayTest 类中对数组列表的访问了。

addItem()方法显示了集合类的一个好处：我们可以在一个同步块中向集合加入多个项目。这比 ArrayList 中的同步 add() 方法更加有效。在我们的测试类中，我们仅仅需要获取同步锁一次；而在传统的Vector类中，我们不得不获取该锁两次。但是这种效率也是有很高代价的，如果你忘记对映射进行正确的同步，就会碰到很多竞态条件，而这是很难被发现的。使用哪一种集合类可以由你自己选择。

这种方法可以用在那些其访问可以被同步的任何非线程安全类上。但是还是有一些非线程安全类（例如我们后面要提到的JFC类）不能使用这种技术，因为其内部会以非同步的方式调用其他的非线程安全类。但是对于非线程安全的数据结构类，应当使用显式同步技术。

显式同步和本地代码

当需要调用一个非线程安全的本地库时，一定要使用显式同步技术。因为使用C或者是其他语言的程序员不会考虑到他的库会在一个多线程环境中使用，因此在这种情况下经常要使用该技术。

但是，此时还是有一点点不同。我们不能在对象级别上进行同步（像上面的例子那样），因为虚拟机有一个共享的本地库实例，所以每一个对象都共享相同的本地代码。更进一步，我们必须在类级别进行同步操作，因此每一个使用本地库的对象都会共享同一个锁。

实现这个任务很简单:

```
public class AccessNative {
    static {
        System.loadLibrary("myLibrary");
    }
    public static synchronized native void function1();
    public static synchronized native void function2();
    ...
}
```

这里我们使得每一个调用本地库的方法都是静态的和同步的。因为对于这些方法的调用都要求获取一个和AccessNative类相关联的锁，这就保证了在一个时刻只有虚拟机中的一个线程能够调用本地方法。

同时还有一个警告：如果另外一个类也加载了myLibrary库，则该类中的线程可以和AccessNative类中的线程同时调用本地库中的方法。

这种技术和在JDBC-ODBC桥中使用的技术是一样的：在JDBC-ODBC桥的早期版本中，假设底层ODBC驱动程序是非线程安全的，因此采取了串行访问本地库的方式。但是这样做使得线程不能够并行地访问数据库，这极大地降低了效率。这是大多数使用数据库的应用程序所面临的问题，要访问数据库的线程不得不经常等待在I/O处理上。

在Java 2中，JDBC-ODBC桥假设底层ODBC驱动程序是线程安全的。如果你使用的是非线程安全的驱动程序，则应该保证对该驱动的访问被正确地同步。要做到这一点，只需对第一种技术进行一点修改即可：只要保证对Connection对象的访问是同步的就可以了。但是在该例中，因为我们使用的是本地代码，所以还要保证在虚拟机中只有一个Connection对象在使用ODBC驱动程序。

单线程访问

另外一个可以用于非线程安全的技术就是保证只有一个线程能够访问这些类。这是一个困难的任务，但这项技术的优势就是它一定是可以工作的，而不管这些类在内部做什么，如果要在使用JFC作为其GUI的程序中使用线程，则一定要使用

这个技术。我们将首先说明如何和JFC一起工作，然后概述如何利用这项技术和其他类一起工作（特别是如何和你自己开发的类一起工作）。

使用 JFC

JFC是Java平台上最大的类集，而且也因为其属于为数不多的几个非线程安全的类集而著名。因此，无论何时使用这些类，都要注意只能从一个线程中访问JFC对象；特别是我们要保证只能够从虚拟机中的一个事件调度线程来访问JFC对象。这个线程通过执行任何一个监听器方法（例如actionPerformed()）来响应用户的事件。

所有的JFC对象都不是线程安全的，这意味着我们不能够直接通过自己的线程来调用这些对象中的方法。例如线程不能直接试图从滑动块对象中读取数据。这是因为当线程从滑动块中读取数据时，用户可能正在改变滑动块的值。因为对滑动块的访问不是同步的，所以这两个线程都可以同时访问滑动块对象，这就会破坏滑动块的内部数据并且产生错误。因此我们的线程一定要通过虚拟机中的事件分派线程来读取滑动块的数据。

这个例子也说明了为什么前面讨论的用于对象访问的显式同步技术不能够用在JFC上：我们的线程可以同步对滑动块的访问，但是事件分派线程并不会同步其内部的访问。要记住锁都是要互相合作的；如果所有的线程都不去获取锁，则竞态条件还是会发生。

因此要安全地和Swing组件交互，就要保证只从事件分派线程来访问它们，因为只有这样才能有效地保证对这些组件的访问是单线程的，从而杜绝竞态条件的发生。JFC包含了很多在事件分派线程中执行的方法。

- 在java.awt.event包的监听接口中定义的方法。
- invokeAndWait()。
- invokeLater()。
- repaint()。

我们将依次来了解它们。

事件分派线程和与事件相关的方法

首先，让我们看看什么是事件分派线程。当Java虚拟机开始执行时，它会启动一个初始线程。然后，当第一个与AWT相关的类（包括JFC类）被实例化时，在JVM中的GUI工具包也会被初始化。根据底层操作系统的不同，这会导致一个或者多个附加的线程被创建，它们负责与本地的窗口系统交互。

不管有多少个线程被创建，其中一个被称为事件分派线程。该线程负责获取用户输入的事件；当用户敲入一个字符时，该事件分派线程就会从底层窗口系统中收到一个消息。当用户移动鼠标或者是按下鼠标的键时，该事件分派线程也会接收到消息。当它接收到消息后，就开始处理该事件的分派：它指出该事件发生在哪个AWT组件上，并且调用在该组件上注册的事件处理方法。

因此，任何被调用来对这些消息进行响应的方法都是在消息分派线程中被调用的。在一般情况下，任何与消息相关的方法（actionPerformed()、focusGained()和itemStateChanged()）以及作为java.awt.event包中的监听接口的一部分的方法都是在事件分派线程中被调用的。

这意味着大多数对Swing组件的访问都是通过事件分派线程来调用的，所以这是个好消息。因此对于大多数的GUI代码都不需要使用下面列出的方法：只有在通过自己的线程（而不是事件分派线程）来访问Swing组件时才需要调用invokeAndWait()和invokeLater()方法。换句话说，如果你将自己的线程加入到一个Swing程序中，并且该线程需要直接访问Swing组件，就需要使用invokeAndWait()或者invokeLater()方法。否则，就应当和正常编程一样编写与事件相关的方法。

在事件分派过程中有两个微妙的地方。第一个是JApplet类中那些与事件相关的方法好像不是由事件分派线程调用的。实际上，JApplet的start()和stop()方法是由程序中的另外一个线程调用的，并且不应该在这两个方法中直接访问任何Swing组件。从技术上讲，该警告也适用于init()方法。因为init()方法通常会进行Swing调用（例如add()方法），这看上去是很危险的。但是，浏览器会保证对init()方法的调用只有一次，并且在调用过程中也保证对Swing组件的访问是安全的。如果要在你自己的程序中使用JApplet，就必须保证做到同样的

事情：在调用JApplet类的init()方法前不要调用任何JFrame的show()方法（或者是使用invokeAndWait()方法来保证init()方法是在事件分派线程中运行的）。同时，如果你的程序调用了init()方法，你也要保证这不会在事件分派线程中进行。

第二点要复杂一些。第二点产生的原因是因为可以从非事件分派线程来调用与事件相关的方法。假设有一个从某个套接字中读取数据的线程；在读取过程中发生了I/O错误，因此你希望停止程序的运行。在这种情况下，你可能会试图调用actionPerformed()方法，该方法对用户按下“关闭”按钮的事件进行响应。毕竟，该方法中已经包含了全部的关闭程序代码，而且你也不想再重写这些代码了。因此，actionPerformed()方法可能被两个不同的线程调用：事件分派线程（用来响应用户事件）和通过套接字读取数据的线程（用来响应I/O错误）。为了适应这两个线程的要求，必须使用下面要讨论的技术，以保证在actionPerformed()方法中对任何Swing组件的访问都是安全的。

关键在于actionPerformed()方法（以及任何与Swing事件相关的方法）不能够保证可以安全地操作Swing组件：当该方法被事件分派线程调用时是安全的，而被其他线程调用时是不安全的。对Swing组件的操作是否安全是由上下文而不是方法本身决定的。

使用哪个invokeAndWait()方法？

在Java 2中，EventQueue类引入了三个新的静态方法：isEventDispatchThread()、invokeLater()和invokeAndWait()。这些方法在功能上和SwingUtilities类中对应的方法是相同的。你可以根据自己的喜好来任意选用其中的一个；使用SwingUtilities的方法可以保证你的程序与Java 1.1兼容。

invokeAndWait()方法

要保证对于Swing组件的访问是通过事件分派线程进行的，最简单方法就是使用invokeAndWait()方法。如果一个线程执行该方法，它就阻塞直到事件分派线程执行完某些代码为止。

下面我们来看一个例子。invokeAndWait()方法经常用来获取GUI中某个元素的值。在下面的代码中，我们使用invokeAndWait()来获取一个滑动块的值。

```
import javax.swing.*;
import java.awt.*;

public class SwingTest extends JApplet {
    JSlider slider;
    int val;

    class SwingCalcThread extends Thread {
        public void run() {
            Runnable getVal = new Runnable() {
                public void run() {
                    val = slider.getValue();
                }
            };

            for (int i = 0; i < 10; i++) {
                try {
                    Thread.sleep(2000);
                    SwingUtilities.invokeAndWait(getVal);
                    System.out.println("Value is " + val);
                } catch (Exception e) {}
            }
        }

        public void init() {
            slider = new JSlider();
            getContentPane().setLayout(new BorderLayout());
            getContentPane().add("North", slider);
        }

        public void start() {
            new SwingCalcThread().start();
        }
    }
```

尽管这只是一个简单的程序框架，但是该applet还是创建了一个滑动块并且启动了一个进行计算的线程。下面让我们看看该applet是如何执行的：

1. applet初始化自己（通过init()方法），创建一个仅有一个滑动块的GUI。

2. 在applet的start()方法中，生成一个计算线程。

3. 该计算线程会进行计算工作（一开始它是处于睡眠状态的，但是不久就要进行一些有用的工作了）。该计算线程需要周期性地获取滑动块的当前值。它通过创建一个可运行对象（getVal实例变量）并且将它传递给invokeAndWait()方法来做到这一点。计算线程随后会进入阻塞状态，直到invokeAndWait()方法返回。

4. 同时，invokeAndWait()方法就会安排GUI事件分派线程来调用getVal对象的run()方法。当run()方法被调用后，滑动块的值就会被存放到val实例变量中。

5. 一旦getVal对象的run()方法返回了，invokeAndWait()方法就会返回，而计算线程就可以进行下一轮循环了。

但这里也有一点复杂之处，即不能在事件分派线程内部调用invokeAndWait()方法；这样做会抛出一个异常。但是如果想和上面的套接字例子一样，在一个事件回调方法和一个用户线程中执行相同的代码，则不能在actionPerformed()方法中随便地使用invokeAndWait()方法来调用使用Swing组件的方法；你必须通过SwingUtilities.isEventDispatchThread()方法来检查一下当前是否是在事件分派线程中，并且在actionPerformed()方法中有相应的代码。该例子的代码框架如下：

```
public class TestSwing extends JApplet implements ActionListener {
class ReaderThread extends Thread {
    public void run() {
        try {
            //... 通过套接字读取数据，处理数据 ...
        } catch (IOException ioe) {
            actionPerformed(null);
        }
    }
}

public void init() {
    JButton jb = new JButton("Close");
    getContentPane().add(jb);
    jb.addActionListener(this);
}
```

```
        public void actionPerformed(ActionEvent ae) {
            class doClose implements Runnable {
                public void run() {
                    //... 访问Swing组件 ...
                    //... 这些代码是actionPerformed方法的主体 ...
                }
            };
            doClose dc = new doClose();
            if (SwingUtilities.isEventDispatchThread())
                dc.run();
            else {
                try {
                    SwingUtilities.invokeAndWait(dc);
                } catch (Exception e) {}
            }
        }
    }
```

对于invokeLater()方法而言，是没有这种限制的。

invokeLater()方法

invokeLater()方法类似于invokeAndWait()方法，但是它是不阻塞的。因为它并不等待目标对象的run()方法执行完毕，所以此方法不适用于那些需要从JFC对象中获取数据的情况。但是，该方法可以用来在JFC对象中设置数据。

```
import javax.swing.*;
import java.awt.*;

public class SwingTest extends JApplet {
    JSlider slider;
    JLabel label;
    int val;

    class SwingCalcThread extends Thread {
        public void run() {
            Runnable getVal = new Runnable() {
                public void run() {
                    val = slider.getValue();
                }
            };
            Runnable setVal = new Runnable() {
                public void run() {
                    label.setText("Last calc is " + val);
```

```
            }
        };

        for (int i = 0; i < 10; i++) {
            try {
                Thread.sleep(2000);
                SwingUtilities.invokeAndWait(getVal);
                SwingUtilities.invokeLater(setVal);
            } catch (Exception e) {}
        }
    }
}

public void init() {
    slider = new JSlider();
    label = new JLabel("Last calc is 0");
    getContentPane().setLayout(new BorderLayout());
    getContentPane().add("North", slider);
    getContentPane().add("Center", label);
}

public void start() {
    new SwingCalcThread().start();
}
```

在该例中，没有理由让计算线程等待标签中的数据被设置完成；计算线程仅仅调度操作，然后继续其计算工作。在某些情况下，这种做法是不合适的。在本例中，当invokeLater()被调用时，标签的值并不能立刻更新。这有可能导致其中一次循环的反馈信息没有显示给用户。但是，总体而言，在线程不是特别关心run()方法运行的结果时，invokeLater()方法还是很有用的。

repaint()方法

即使是在JFC中，repaint()方法仍然是线程安全的。因此，任何线程都可以在任意时刻调用特定组件的repaint()方法。因为有很多Java应用程序都依赖于周期性的重画行为，所以这一点很重要。

repaint()方法能够这样工作的原因是其自身并不做很多的工作：它仅仅是安排事件分派线程调用相对应的paint()方法。因此，一个applet可以让一个线程将

数据保存到实例数据中然后再调用applet的repaint()方法；当applet下次再次刷新自己时，就可以使用新的数据了。

还有其他可以用来处理JFC和线程的技术。在JFC中有一个定时器类可以用来隐藏invokeLater()方法的细节；将ActionListener对象传递给该定时器，当时间到时，定时器就会安排事件分派线程调用该对象的actionPerformed()方法。

另外，在Sun公司的网站上还有一个名为SwingWorker的类。它可以用和上面相反的方式工作：它创建一个新的目标线程，而在事件分派线程中编写获取该目标线程结果的代码。但是以我的观点看来，这种做法是一个后退：事件分派线程怎么知道何时应该检查目标线程的结果呢？当然，如果你感兴趣，也可以通过Sun公司的网站来获得更多的技术细节。

那么Swing类到底有多么不安全的呢？在我们曾经使用过的例子中，我们实际上是在JSlider类中设置和获取一个整型值的。在Java中读写整型值是一个原子操作，那么是否有必要使用这些方法呢？的确，在有些例子中，答案是否定的。但是这些例子是很难被清楚地描述的。因此在非事件分派线程中调用所有的Swing方法要比使用事件分派线程更安全。即使是在上面那些只进行简单赋值操作的例子中，也有很多我们并没有注意到的方法被调用了：getValue()方法调用了getModel()方法，同时可能正在安装一个新的模式。这可能不会导致什么问题，也可能使得getModel()方法返回null对象引用，从而引发运行时异常；如果没有对Swing代码进行仔细的研究，就不能保证程序的健壮性。同时因为我们不能预计以后的实现会是什么样的，所以最好是使用上面说明的调用方法。

其他非线程安全类

invokeAndWait()方法(以及上面讨论过的其他类似的方法)提供了一个在简单的外部同步机制无能为力时处理非线程安全类的方法。对这些非线程安全类，我们应该实现一个类似的机制。

通过队列来使得有且只有一个线程可以对发生的动作进行响应是一种典型的解决方法。invokeAndWait()方法本身就是建立在虚拟机内部的一个事件队列的基础

上的：虚拟机仅仅创建新的事件，将它们放到队列中去然后等待事件分派线程处理它们（invokeLater()方法不用等待即可返回）。事件分派线程则负责处理传递给invokeAndWait()方法的对象的run()方法。更有趣的是，invokeAndWait()方法并不会自己创建新的线程，也不会导致新的线程被创建：run()方法是在当前存在的线程（事件分派线程）中运行的，这一点和第七章中讨论的线程池例子是一样的。

这种类似性告诉我们如何保证只有一个线程可以访问非线程安全类：将所有这些访问封装在一个在线程池中执行的对象中，而该线程池在初始化时就只创建一个线程。此时我们就可以像使用invokeLater()和invokeAndWait()方法一样使用该线程池的addRequest()和addRequestAndWait()方法了。

总结

将锁机制与Java语言和API进行紧密结合对于多线程的Java编程是非常有用的。但是，尽管Java的锁机制是很强大的，但在某些复杂的Java程序中，它也可能不能提供我们所需要类型的同步机制。幸运的是，Java语言内置的同步机制是帮助我们在那些特殊情况下创建更复杂、更有效的锁的基石。

如同Java语言中的其他部分一样，Java中内置的锁机制是依照简单性的原则来设计的，这降低了Java程序中出错的可能性。但是像Java语言的其他部分一样，这种简单性又可以满足除了最复杂编程需求外的其他要求。因此除非真正需要本章所介绍的这些复杂的同步机制，否则应当尽可能地使用那些内置的同步机制。

最后，当你要使用那些非线程安全的代码时，Java的锁定工具会保证这些代码在多线程的程序中也能安全地运行，这是通过显式地锁定这些代码或者保证这些代码只在一个线程中执行来实现的。

本章内容：

- 单线程程序并行化
- 内层循环线程化
- 循环输出
- 多处理器扩展
- 总结

第九章

多处理器机器上的并行化

在此之前，我们只是把线程化当作一种简化编程的技术：线程被用来获取异步行为或是执行独立的任务。尽管我们曾讨论过线程是如何在多处理器机器上被调度的，但总的说来，我们先前讨论的技术与机器的处理器数目并没有什么关系，我们也没有试图充分利用机器的处理器数目以获得更快的执行速度。

多线程应用程序和多处理器系统有着特别的联系。线程的分离为处理器的分配提供了简单明了的方案。由于操作系统可以把不同的线程分配到不同的处理器上，所以应用程序可以运行得更快。

在这一章中，我们将讨论如何并行化Java程序以使它们在多CPU机器上更快地运行。我们所考察的方法不仅适用于新开发的程序，而且适用于那些占用CPU较多的现有程序。

Java线程系统在多处理器机器上是如何运转的呢？ 一个程序是运行在单处理器机器上还是多处理器机器上在概念上其实并没有多大差别；线程的行为在两种情况下基本相同。尽管如此，就像我们在第六章中讨论过的一样，关键的差别在于，对于多处理器系统来说，可能宿主平台的每一个CPU上都有正在运行的线程。这一点使得当Java程序运行在多处理器机器上时，下面的假设变得十分重要：

- 我们不能再认定当前运行的线程具有最高的优先级。更高优先级的线程可能运行在另一个处理器上。
- 我们不能再认定低优先级的线程没法运行。有可能存在着充足的处理器让它运行。
- 我们不能再认定不同优先级的线程无法同时运行。
- 我们不能再认定因为某些特定场景不可能发生，从而一些竞态条件可以被忽视。虽然单处理器系统上的竞态条件更依赖于Java虚拟机的调度引擎，多处理器系统的竞态条件却是实实在在存在的。

这里要理解的一点就是以前的那些假设现在并不能得到保证。虽然在单处理器机器（尤其是绿色线程模型）上，很少有违反这些假设的情况，但在多处理器系统中，违反它们的情况会经常发生。

单线程程序并行化

如果不重新设计程序，并行化的最佳区域（也就是可以引入多线程以提高程序性能的区域）是应用程序受CPU限制的地方。毕竟，如果连一个CPU都不是始终在忙的话，就没什么理由再引入新的处理器。在很多情况下，只有在受CPU限制的地方（此时，进程使用了全部的处理器周期，但没有充分利用磁盘或网络等资源），使用更多的处理器才会提高性能。这种处理过程可能是一个大的数学计算，但更可能是对一些短小的数学计算的多次迭代。而且，这些计算可能会使用一个大的循环，甚至是大量的嵌套循环。我们马上将要研究的算法通常就是这种类型的。考虑下面的计算：

```
public class SinTable {
    private float lookupValues[] = null;

    public synchronized float[] getValues() {
        if (lookupValues == null) {
            lookupValues = new float [360 * 100];
            for (int i = 0; i < (360*100); i++) {
                float sinValue = (float)Math.sin(
                                        (i % 360)*Math.PI/180.0);
```

```
                lookupValues[i] = sinValue * (float)i / 180.0f;
            }
        }
        return lookupValues;
    }
}
```

我们将使用这段代码作为本章后面示例的基础。单线程程序（从而也只使用单处理器）将执行代码指定循环，把结果存入lookupValues数组。假定对sinValue变量的计算很耗时，整个循环就会执行很长时间。这种情况在有些时候可以接受。但假定你有一台有12个处理器的计算机，而且没有其他的应用程序需要运行，现在只有一个CPU工作而其他11个处理器都闲着。考虑到12路机器的昂贵费用，这种情况让人难以接受。

在开始讨论之前，让我们先定义几个术语（注1）。变量sinValue有几个特别的属性。很明显，它只在循环的运行期内存在。它是用来帮助查找表计算的临时变量；它不在循环的不同迭代之间传递值，并且在每次迭代中，会被重新赋值。我们定义sinValue为一个循环私有变量（loop-private variable），即：完全在循环的单个迭代中初始化、计算和使用的变量。

进一步的考察会发现，索引变量i也是循环私有的：它也是完全用在循环的一个迭代中。它还可以被认为是一种特殊类型的循环私有变量。因为它在一个迭代中从来不改变自身的值，而是完全和迭代索引联系在一起，我们实际上可以把它看成是迭代中的常量。不过现在，简单地把它看成循环私有变量已经足够了。

我们可以像下面这样把循环的不同部分分解到多个线程中去：

```
public class SinTable implements Runnable {
    private class SinTableRange {
        public int start, end;
    }

    private float lookupValues[];
    private Thread lookupThreads[];
```

注1：本节使用的术语在某种程度上都是基于对Solairs操作系统可用的自动线程化MPC编译器的。

```
    private int startLoop, endLoop, curLoop, numThreads;

    public SinTable() {
        lookupValues = new float [360 * 100];
        lookupThreads = new Thread[12];
        startLoop = curLoop = 0;
        endLoop = (360 * 100);
        numThreads = 12;
    }

    private synchronized SinTableRange loopGetRange() {
        if (curLoop >= endLoop)
            return null;
        SinTableRange ret = new SinTableRange();
        ret.start = curLoop;
        curLoop += (endLoop-startLoop)/numThreads+1;
        ret.end = (curLoop<endLoop)?curLoop:endLoop;
        return ret;
    }

    private void loopDoRange(int start, int end) {
        for (int i = start; i < end; i += 1) {
            float sinValue = (float)Math.sin((i % 360)*Math.PI/180.0);
            lookupValues[i] = sinValue * (float)i / 180.0f;
        }
    }

    public void run() {
        SinTableRange str;
        while ((str = loopGetRange()) != null) {
            loopDoRange(str.start, str.end);
        }
    }

    public float[] getValues() {
        for (int i = 0; i < numThreads; i++) {
            lookupThreads[i] = new Thread(this);
            lookupThreads[i].start();
        }
        for (int i = 0; i < numThreads; i++) {
            try {
                lookupThreads[i].join();
            } catch (InterruptedException iex) {}
        }
        return lookupValues;
    }
}
```

新版本中的代码和先前版本的代码在功能上是相同的，不过，它在逻辑上进行了一些修改。首先，现在没有使用一个大循环进行计算，而是使用一个循环去启动12个（numThreads）不同的子线程，并且给每个子线程提供数学循环的不同部分去进行运算。原先的数学计算被移到一个新的方法loopDoRange()中。在这个方法中，循环被修改为只在查找表的一部分上，而不是在整个查找表上工作。各个线程只负责计算自己的那部分表。这些线程通过调用lookGetRange()方法来确定自己计算表的哪一部分。而那个启动12个子线程的初始线程只是简单地等待12个子线程运行结束。由于这个长计算现在使用12个而不是1个线程来完成，因此基于多处理器的操作系统现在就有可能把不同的线程放到不同的处理器上。

这种计算能够行之有效是因为下面几个原因。首先，循环索引变量i和sinValue变量是循环私有变量，现在在各个工作线程中成为堆栈变量。各线程的loopDoRange()方法使用两个变量的不同拷贝。这意味着12个子线程在计算时都拥有这些变量的拷贝。

其次，虽然lookupTable数组不是循环私有的，但它的个体成员却可以被认为是循环私有的。数组的每个个体变量只被特定的迭代访问。因为每个线程影响且只影响数组的一个成员，所以，虽然不同的子线程处理循环中的多个迭代，但一个迭代不会被多个线程处理。

惟一需要同步的是对不同范围的分配。为了避免子线程在分配中相互重叠，loopGetRange()方法需要同步。在这个例子中，loop只被分解成12个区域，所以执行这个方法的时间与循环本身的计算相比就微不足道了。

新版本的代码比第一个版本更加复杂。新的代码必须启动和跟踪12个单独的线程。子线程必须被修改为能够处理循环中指定的一部分。虽然例子中的同步工作很少，但依据数学计算中的算法，我们很可能会遇到同步需求很复杂的情况。

在处理这么简单的循环时都引入了这么多的复杂性，在面对更复杂的循环时，我们很可能会无从下手。为了控制复杂性，我们把所有和循环管理有关的逻辑都转移到一个单独的类中。然后，我们就可以通过使用这个类提供的服务而方便地实现循环：

```
public class LoopHandler implements Runnable {
    protected class LoopRange {
        public int start, end;
    }
    protected Thread lookupThreads[];
    protected int startLoop, endLoop, curLoop, numThreads;

    public LoopHandler(int start, int end, int threads) {
        startLoop = curLoop = start;
        endLoop = end;
        numThreads = threads;
        lookupThreads = new Thread[numThreads];
    }

    protected synchronized LoopRange loopGetRange() {
        if (curLoop >= endLoop)
            return null;
        LoopRange ret = new LoopRange();
        ret.start = curLoop;
        curLoop += (endLoop-startLoop)/numThreads+1;
        ret.end = (curLoop<endLoop) ? curLoop : endLoop;
        return ret;
    }

    public void loopDoRange(int start, int end) {
    }

    public void loopProcess() {
        for (int i = 0; i < numThreads; i++) {
            lookupThreads[i] = new Thread(this);
            lookupThreads[i].start();
        }
        for (int i = 0; i < numThreads; i++) {
            try {
                lookupThreads[i].join();
            } catch (InterruptedException iex) {}
        }
    }

    public void run() {
        LoopRange str;
        while ((str = loopGetRange()) != null) {
            loopDoRange(str.start, str.end);
        }
    }
}
```

在新的LoopHandler类中，我们实现了先前应用到sinTable上的逻辑。产生、跟踪以及与初始线程汇合的逻辑现在用新的loopProcess()方法来完成。确定范围以及处理循环的逻辑（原来在SinTable类的run()和loopGetRange()方法中实现）都几乎没有改变。循环处理程序同样被修改以便能处理更通用的循环，而且由一个构造函数为循环的开始、结束和线程的数目赋值。和前面的例子一样，算法调用loopDoRange()方法去处理计算。不过，在这个例子中，LoopHandler类对这个方法有一个空的实现。

现在，SinTable类的实现就简单多了：

```
public class SinTable extends LoopHandler {
    private float lookupValues[];
    public SinTable() {
        super(0, 360*100, 12);
        lookupValues = new float [360 * 100];
    }

    public void loopDoRange(int start, int end) {
        for (int i = start; i < end; i++) {
            float sinValue = (float)Math.sin((i % 360)*Math.PI/180.0);
            lookupValues[i] = sinValue * (float)i / 180.0f;
        }
    }

    public float[] getValues() {
        loopProcess();
        return lookupValues;
    }
}
```

在这个例子中，我们只需要配置循环处理程序需要的范围，提供循环的逻辑给loopDoRange()方法，并且调用loopProcess()方法，以在多线程方式下处理循环。虽然，它还是比SinTable类的第一个实现复杂，但却比刚才那个实现更简单和容易控制。

循环调度和负载平衡

循环调度（loop schedling）指的是把循环中的迭代分配到单个线程中的过程。在LoopHandler类中，它是用loopGetRange()方法实现的。为了最大程度地利用

处理器，我们应该在尽少引入分配决策额外负载的情况下，把工作尽量均匀地分配到不同的线程上。这个过程被定义为负载平衡（load balancing）。

我们要处理的循环调度有以下几种基本类型：

静态或组块调度（static or chunk scheduling）

在静态调度中，根据线程的数量，给每个线程分配相同数目的迭代。假定循环中待分配的迭代数目为1000，用来处理任务的线程数目为10，那么每个线程会被分配100次迭代。这也就是LoopHandler类使用的算法。算法在数目上加了1以保证分配能够整除。否则，可能会剩下1个迭代，也就必须有一个子线程在完成原先的组块后再去处理它。

这个算法的问题在于它假定循环的每个迭代花费相同的时间。如果这一点不成立，那么会有一个线程比其他线程晚完成。而由于分配是在循环开始时进行的，其他的线程也只能在它完成剩余工作的时候袖手旁观。

自调度（self-scheduling）

在自调度中，每个子线程首先抓取小的迭代组块来执行。完成分配的范围后，它再抓取另外的小块。假定循环中待分配的迭代数目为1000，用来处理任务的线程数目为10，那么每个子线程会一直处理很小的组块（比如说20个迭代），直到1000个迭代被处理完毕。

虽然这种方式和静态调度一样，不同的子线程可能也不会同时完成工作，但由于自调度模型中的组块很小，进程等候最后线程处理的空闲时间也会很少。为了让这个空闲时间更小，我们可以把组块分得更小。然而，在获取处理区域的过程中会有额外的开销，组块越小，开销就越大。

下面是这个模型的实现：

```
public class SelfLoopHandler extends LoopHandler {
    protected int groupSize;

    public SelfLoopHandler(int start, int end, int size, int threads) {
        super(start, end, threads);
        groupSize = size;
    }

    protected synchronized LoopRange loopGetRange() {
```

```
            if (curLoop >= endLoop)
                return null;
            LoopRange ret = new LoopRange();
            ret.start = curLoop;
            curLoop += groupSize;
            ret.end = (curLoop<endLoop)?curLoop:endLoop;
            return ret;
        }
    }
```

自调度循环处理程序的设计很简单。现有的LoopHandler类已经有了工作到循环完成的逻辑。我们只需要修改构造函数，使之处理被请求的组块大小，并修改loopGetRange()方法来返回确定的组块大小。在自调度程序的实现中，我们只需继承原有的循环处理程序，并实现其中的变化就行了。

引导型自调度（guided self-scheduling）

引导型自调度程序是静态调度程序和自调度程序的折衷。开始时，引导型调度程序抓取循环的很多迭代；在接近尾部的过程中，迭代越取越少。算法还会用到一个最小组块大小。因而，它的行为基本上像一个慢慢转变成自调度程序的静态调度程序。假定循环中待分配的迭代数目为1000，用来处理任务的线程数目为10，那么第一个子线程获取任务的1/10（100个迭代）。第2个线程获取剩余工作的1/10（90个迭代）。迭代次数越来越少直到达到最小组块大小，比如10；然后一直分配最小大小的组块直到1000个迭代全部结束。

这个算法看起来问题最少。与自调度程序不同，它的额外负担只出现在循环尾部。而且，除非单个迭代与开始时的长期迭代相比在执行周期上差异很大，否则不会出现静态调度程序所具有的问题。

下面是引导型自调度程序的实现：

```
public class GuidedLoopHandler extends LoopHandler {
    protected int minSize;

    public GuidedLoopHandler(int start, int end, int min, int threads){
        super(start, end, threads);
        minSize = min;
    }

    protected synchronized LoopRange loopGetRange() {
```

```
            if (curLoop >= endLoop)
                return null;
            LoopRange ret = new LoopRange();
            ret.start = curLoop;
            int sizeLoop = (endLoop-curLoop)/numThreads;
            curLoop += (sizeLoop>minSize)?sizeLoop:minSize;
            ret.end = (curLoop<endLoop)?curLoop:endLoop;
            return ret;
        }
    }
```

实现引导型自调度程序同样也很简单。我们只需要修改构造函数让它能处理所需的最小组块大小，并修改loopGetRange()方法来返回剩余循环的一部分。在引导型自调度程序的实现中，我们同样继承了原先的循环处理程序，只实现了其中的变化部分。

用户自定义调度程序（User-defined scheduler）

前面，我们有意让自调度程序和引导型自调度程序的实现很简单。我们把最初的循环处理程序设计得很适合继承，其中的调度算法可以被方便地修改。尽管这样实现引导型自调度程序已经很不错了，它仍然是为一般的循环而设计的。虽然每种调度程序都有比其他调度程序工作得更好的情况，但如果知道关于循环的充分信息，而且又花费了足够的努力，我们还是会发现可能使用其他的调度程序会更好一些。这需要找到合适的处理逻辑，并编写一个新的loopGetRange()方法。

下面来看看如何修改我们最初的例子，使它运用某种我们刚才看到的调度技术：

```
public class SinTable extends GuidedLoopHandler {
    private float lookupValues[];

    public SinTable() {
        super(0, 360*100, 100, 12);
        lookupValues = new float [360 * 100];
    }

    public void loopDoRange(int start, int end) {
        for (int i = start; i < end; i++) {
            float sinValue = (float)Math.sin((i % 360)*Math.PI/180.0);
            lookupValues[i] = sinValue * (float)i / 180.0f;
        }
    }
```

```
        public float[] getValues() {
            loopProcess();
            return lookupValues;
        }
    }
```

为了在SinTable类中使用引导型自调度程序算法，我们只需简单地继承GuidedLoopHandler类，并修改其构造函数以传递最小组块大小。我们还可以重写GuidedLoopHandler类，对构造函数进行重载，使它获取默认最小值，这样，它就可以有和静态循环调度程序签名相同的构造函数。

自动并行化编译器

这个词来自于Solaris平台上的自动并行化MP C编译器。自动并行化所用的技术和我们这一章前面所讲的相同，只是它是由编译器而非程序员来完成的。虽然很早就有一些语言（比如FORTRAN）能提供自动并行化技术，但对C语言来说，它还比较新。这是因为C语言中指针以及其他一些机制引发了一些混淆问题，从而难以给变量和循环分类。即使是在现在，也还需要使用#pragmas帮助编译器区分循环中的变量。

在这一点上，Java看起来更像FORTRAN，而不是C。在Java中，所有的变量引用都被跟踪（为了垃圾收集的目的），不允许使用指针算法，变量类型也被强制。因此，其中的混淆问题要比C少得多。这也意味着为Java开发自动并行化编译器要比为C开发简单多了。当然，除非你已经有了一个自动并行化编译器，否则你必须像我们前面做的那样去使用这些技术。

变量分类

在实现SinTable类时，我们曾经把原来非线程化循环中的变量归类为循环私有变量。其实，除了循环私有变量之外，还有着其他变量类别。对变量进行分类的原因在于循环中不同迭代间可能存在着数据依赖关系，不同类型的变量需要在线程内和线程间进行不同类型的处理。通过变量分类，我们就能够正确地更改它们而不引起竞态条件。变量的分类依据是它们的用法，这些分类也决定了它们在多线程循环处理程序中被实现和处理的方式。

循环私有变量

前面已经提到，循环私有变量是那种不会在循环的迭代间传值的变量。它实际上可能是循环本身定义的变量，也可能是只被循环的某个迭代访问的实例或公共变量。lookupValues数组变量就是这种情况，数组的每个成员只被循环的一个迭代访问。所以尽管整个数组不是对任一迭代循环私有的，但数组的特定成员对特定迭代来说却是循环私有的。

如SinTable类中所示，对循环私有变量的处理通常是利用变量在该线程中的局部拷贝来进行的。因为每个线程都有拷贝，所以线程之间不会互相干扰。在lookupValues数组的例子中，线程只访问数组中私有的部分，而不会干扰其他线程的工作。

只读变量

只读（read-only）变量指的是其值在循环过程中不发生变化的变量。它们可能是常量，也可能是循环前被初始化而到循环处理后才被改动的变量。

对只读变量不必进行特殊处理，子线程不需要拥有它们的局部拷贝，对它们的访问也不需要任何同步。

回存变量

回存（storeback）变量基本上是循环结束后需要使用的循环私有变量。比如，假定我们在循环结束后还需要对 lookupValues 数组进行一些额外的处理：

```
public float[] getValues() {
    if (lookupValues == null) {
        float sinValue = 0;
        lookupValues = new float [360 * 100];
        for (int i = 0; i < (360*100); i++) {
            sinValue = (float)Math.sin((i % 360)*Math.PI/180.0);
            lookupValues[i] = sinValue * (float)i / 180.0f;
        }
        lookupValues[0] += sinValue;
    }
    return lookupValues;
}
```

在这个被改动了一点的SinTable版本中，sinValue变量和lookupValue数组的个体成员都仍是循环私有变量。它们在循环的不同迭代间并不存在数据依赖关系。然而，sinValue在这里同时也是回存变量。由于它在循环结束后还需使用，所以必须保证它的结果和循环以正确顺序执行时的结果一致。lookupValues数组的成员也总是被看成回存变量，只是由于并没有保留个体拷贝，所以没有必要做专门区分。

下面说明回存变量是如何被处理的:

```
public class SinTable extends GuidedLoopHandler {
    private float lookupValues[];
    private float sinValue;

    public SinTable() {
        super(0, 360*100, 100, 12);
        lookupValues = new float [360 * 100];
    }

    public void loopDoRange(int start, int end) {
        float sinValue = 0;
        for (int i = start; i < end; i++) {
            sinValue = (float)Math.sin((i % 360)*Math.PI/180.0);
            lookupValues[i] = sinValue * (float)i / 180.0f;
        }
        if (end == endLoop)
            this.sinValue = sinValue;
    }

    public float[] getValues() {
        loopProcess();
        lookupValues[0] += sinValue;
        return lookupValues;
    }
}
```

sinValue仍是循环私有变量。然而，它的确是个回存变量，因此我们需要保存它“最后”的值。由于算法现在是在多线程方式下执行的，所以，最后进行的迭代未必就会产生最后的值。

线程必须检查确认自己执行的是循环的最后组块，才能把自己的循环私有变量拷

贝到全局拷贝。还要注意的是，这里并不需要同步，因为只有最后迭代中的值会被拷贝，所以只会有一个线程执行这段代码，不可能产生竞态条件。

归约变量（reduction variable）

显然，循环的不同迭代间确实存在实际的数据依赖关系，所以我们无法把每个变量都处理为循环私有变量。这些数据依赖关系使得执行不同迭代的不同线程在执行期间可能会相互干扰。我们把这些被循环的多个迭代共享的变量称为共享变量（shared variable）。

共享变量会产生一些问题。首先，不同线程同时访问它们会产生竞态条件。其次，共享变量的值可能依赖于处理顺序。第一种情况可以通过同步简单地解决。但第二种情况会产生很大的问题。

我们先来考虑：如果顺序不产生什么影响，情况会是怎样的。这时，我们可以用任何顺序处理循环，只需要在访问它们时进行同步。例如，假定我们还需要计算SinTable的和：

```
public float[] getValues() {
    for (int i = 0; i < (360*100); i++) {
        sinValue = (float)Math.sin((i % 360)*Math.PI/180.0);
        lookupValues[i] = sinValue * (float)i / 180.0f;
        sumValue += lookupValues[i];
    }
    return lookupValues;
  }
```

在这种情况下，sumValue变量显然不是循环私有变量。它的值从一个迭代被传递到另一个迭代，要想得到正确的处理结果就需要维持这种依赖关系。然而，sumValue变量只在循环结束后才会有用。迭代只是把值加到总计中去，并没有处理小计或者其他基于顺序的要求。而且，加法本身是和顺序无关的：把一组数以任意顺序相加，结果总会相同。

sumValue变量是个归约变量。它仍然要被多个线程共享，但由于处理顺序不重要，所以这种共享只需要同步以避免竞态条件：

```
public class SinTable extends GuidedLoopHandler {
    private float lookupValues[];
    public float sumValue;
```

有时候，顺序确实会起作用

在本节的例子中，我们假定可以按任意顺序进行加法操作。由于加法符合结合律，这当然可行。

但是，在计算机中，加法未必一定符合结合律。因为计算机内部使用固定数目的位存储无穷精确数，在每次数学计算中都会发生一些取整错误。一般来说，这些错很小，有时候还会相互抵消，我们没有必要关心它。但是，也有一些情况，当操作的顺序改变时，这些错误的传播会产生差别很大的结果。

比如，在进行敏感数字分析时，就应该注意，本节使用的技巧可能会导致一些无法接受的错误传播和错误结果。

```
    public SinTable() {
        super(0, 360*100, 100, 12);
        lookupValues = new float [360 * 100];
    }

    public void loopDoRange(int start, int end) {
        float sinValue = 0;
        for (int i = start; i < end; i++) {
            sinValue = (float)Math.sin((i % 360)*Math.PI/180.0);
            lookupValues[i] = sinValue * (float)i / 180.0f;
            synchronized (this) {
                sumValue += lookupValues[i];
            }
        }
    }

    public float[] getValues() {
        loopProcess();
        return lookupValues;
    }
}
```

例子使用SinTable实例上的同步锁避免了竞态条件。如果有多个彼此不相关的归约变量，而且它们不能同时保存，那么，为每个回归变量设置一个同步锁（或者BusyFlags）是个不错的想法。

而且，我们现在对循环的每个迭代都做了同步。这样做效率不高。更好的方法是把值赋给循环私有变量，只在把每个范围的和加到规约变量时进行同步。这么做消除了大部分同步的需要，从而大幅度提高了线程的并行化。

```
public class SinTable extends GuidedLoopHandler {
    private float lookupValues[];
    public float sumValue;

    public SinTable() {
        super(0, 360*100, 100, 12);
        lookupValues = new float [360 * 100];
    }

    public void loopDoRange(int start, int end) {
        float sinValue = 0.0f;
        float sumValue = 0.0f;
        for (int i = start; i < end; i++) {
            sinValue = (float)Math.sin((i % 360)*Math.PI/180.0);
            lookupValues[i] = sinValue * (float)i / 180.0f;
            sumValue += lookupValues[i];
        }
        synchronized (this) {
            this.sumValue += sumValue;
        }
    }

    public float[] getValues() {
        loopProcess();
        System.out.println(sumValue);
        return lookupValues;
    }
}
```

在这个新例子中，我们对这个值做了两阶段归约：先把每个迭代的值归约到变量sumValue的局部拷贝，然后把这个局部拷贝归约到实际的归约变量。由于变量sumValue的局部拷贝是循环私有的，所以没有必要进行同步。虽然在计算归约变量的时候还必须同步，但只是对每个范围才做一次，而不是每个迭代都要做一次了。

最后，所有的归约变量都是回存变量。没有必要为归约变量设计专门的回存处理逻辑。

共享变量

从根本上说，循环中所有的变量都是共享变量，因为它们都可以被执行循环的所有线程访问。在并行化循环的过程中，我们很快从共享变量分出只读变量和循环私有变量。对于剩下的共享变量，我们试图把它们转化为循环私有变量，或者把它们分类为归约变量。

不幸的是，有时候一个共享变量不属于上面所说的任何类别，它只是一个共享变量。这时我们的技术就没法工作。尽管我们十分希望把任何循环都变为能在多线程环境中执行，但并不是所有的算法都能够被设计成如此。

共享变量的另一个问题是副作用。比如，如果我们需要保存变量sumValue的每个小计，它就不能被看成是归约变量，因为变量的改变也很重要。如果我们必须在循环过程中输出小计，那么把它定为归约变量不仅可能导致中间结果乱序，而且还可能使中间结果出错。

当使用变量分类的方法不能帮助我们并行化的时候，还有其他一些方法可用。这些方法可能无法解决所有情况，但是随着经验的增加，我们能把越来越多的循环改造为能够在多线程环境下运行。

循环分析和变形

为了对并行化提供帮助，除了对循环变量进行分析以外，我们还可以分析循环本身的算法。虽然在大多数情况下，如果不重新设计算法，我们说什么事情也做不了；但还是有几种情况，我们可以在不完全重新设计的情况下对代码做快速的修改。通过对原始代码进行简单的变形，我们就可以对循环的线程运用我们前面讨论过的技术了。

循环分配

在许多情况下，一个大的复杂的循环中只有一小部分必须顺序执行，而且还有可能把这个大的复杂的循环分解成两个单独的循环。如果复杂循环被分解成为两个

循环 —— 一个包含可以并行执行的代码，另一个包含顺序代码，我们就可以并行化原先循环的一部分。甚至还可以在运行并行循环的同时运行顺序循环。

现在回到我们的SinTable例子，假定我们不仅需要一个总和，而且希望得到待生成的表的运行小计：

```
public float[] getValues() {
    for (int i = 0; i < (360*100); i++) {
        sinValue = (float)Math.sin((i % 360)*Math.PI/180.0);
        lookupValues[i] = sinValue * (float)i / 180.0f;
        if (i == 0) {
            sumValues[0] = lookupValues[0];
        } else {
            sumValues[i] = lookupValues[i] + lookupValues[i-1];
        }
    }
    return lookupValues;
}
```

sumValues数组变量显然是个共享变量。sumValues变量的成员也是共享的，因为它们中的一些也被两个不同的线程访问。而且，处理顺序也会造成影响。这样，如果操作前面块的线程还没有结束，其他的线程就不可能启动新的组块。

可以用下面的方法解决这个问题：

```
public class SinTable extends GuidedLoopHandler {
    private float lookupValues[];
    public float sumValues[];

    public SinTable() {
        super(0, 360*100, 100, 12);
        lookupValues = new float [360 * 100];
        sumValues = new float [360 * 100];
    }

    public void loopDoRange(int start, int end) {
        float sinValue = 0.0f;
        for (int i = start; i < end; i++) {
            sinValue = (float)Math.sin((i % 360)*Math.PI/180.0);
            lookupValues[i] = sinValue * (float)i / 180.0f;
        }
    }
```

```
        public float[] getValues() {
            loopProcess();
            sumValues[0] = lookupValues[0];
            for (int i = 1; i < (360*100); i++) {
                sumValues[i] = lookupValues[i] + lookupValues[i-1];
            }
            return lookupValues;
        }
    }
```

虽然无法在不对算法做很大改动的情况下并行化运行小计，但我们可以快速地把循环转化为两个单独的循环。第一个包含可线程化的代码，第二个处理小计。当这些完成之后，我们就可以先线程化第一个循环，而不改动第二个。在新的SinTable类中，我们已经把计算运行小计的代码移到了单独的循环中。这个单独的循环在第一个线程处理完毕后在一个单线程中运行。

我们对使用这种技术的前后进行一些比较。由于循环的很大部分是在单线程情况下运行的，所以性能收益并不能补偿所花费的努力。但在大多数情况下，小计计算在整个计算中只占很小的比例，所费代价也会很小。

循环隔离

一些应用程序并不包含单个的大循环。即使某个特定的循环注定没办法并行化，程序中可能还有着其他的循环。即使其他这些循环也不能被并行化，我们仍然可以用不同的线程运行不同的循环。

虽然一些循环可能很复杂，在迭代之间存在着大量数据依赖关系，但是有可能不同循环间的数据依赖关系很少。可以隔离个体线程，使它们分别运行在不同的线程中。使用这种技术，就无法再进行负载平衡。毕竟，即使应用程序有4个主要的循环，而且可以完全分离，你还是不可能把这4个循环分配到12个处理器上。

循环交换

多层循环是受CPU限制的应用程序的主要动机。这有可能是循环直接包含在其他循环里，更多的情况可能是循环调用了含有循环的方法。由于这种情况十分常见，

我们将在本章后面部分专门研究内层循环线程化。现在，先让我们来看一个简单的例子：

```
public float[][] getValues() {
    for (int i = 0; i < 360; i++) {
        lookupValues[0][i] = 0;
    }
    for (int j = 1; j < 1000; j++) {
        for (int i = 0; i < 360; i++) {
            float sinValue = (float)Math.sin((i % 360)*Math.PI/180.0);
            lookupValues[j][i] = sinValue * (float)i / 180.0f;
            lookupValues[j][i]+=lookupValues[j-][i]*(float)j/180.0f;
        }
    }
    return lookupValues;
}
```

对多层循环来说，通常对外层循环线程化比对内层循环线程化更为有利。我们没有必要对内层循环和外层循环都线程化，因为无论线程化哪个都会用到全部处理器。如果外层循环被线程化，那么再对内层循环线程化并不能提供更快的速度，因为没有更多的处理器去运行额外的线程(反之亦然)。我们喜欢线程化外层循环的原因在于在生成、销毁和同步多个线程的过程中都需要一定的额外负载。要线程化外层循环，我们只需生成和销毁这些线程一次，而且只在粗粒度上进行同步，而不需要更多的同步。

在表计算的这个新版本中，我们在一个两维表上工作。计算中使用了3个循环。第一个循环只是用来设置第一行值为0，另外两个循环实际上是一对多层循环。算法从一行到另一行循环操作，执行内层循环来处理要存到不同列上的值。

例子的问题在于行之间有着数据依赖关系。因为任何行的计算都依赖于前一行的计算，所以数组中任何列的成员都不是而且没法转变成循环私有的。内层循环的并行化却没有任何问题，因为它们在迭代间不存在数据依赖关系。惟一的要求是内层循环必须认定外层循环以正确的顺序运行；因为我们不曾对外层循环线程化，这个要求是可以满足的。

不过，让我们看看把最初的代码改写为下面这样会怎么样：

```
public float[][] getValues() {
    for (int i = 0; i < 360; i++) {
        lookupValues[0][i] = 0;
    }
    for (int i = 0; i < 360; i++) {
        for (int j = 1; j < 1000; j++) {
            float sinValue = (float)Math.sin((i % 360)*Math.PI/180.0);
            lookupValues[j][i] = sinValue * (float)i / 180.0f;
            lookupValues[j][i]+=lookupValues[j-][i]*(float)j/180.0f;
        }
    }
    return lookupValues;
}
```

在这个例子中，循环被进行了交换。我们不再一行行地处理，而是一列列地计算。这样，内层循环就可以一行行地处理。通过交换循环，内层循环现在就不能再进行线程化，因为loopupValues数组列的成员间存在着数据依赖关系。然而，外层循环现在可以线程化了。而只要外层循环可以线程化，就没有理由再去线程化内层循环。线程化外层循环比线程化内层循环更加有利，在进行多线程化之前的这个简单转换对我们的开发投资给予了更好的回报。

不幸的是，虽然循环嵌套很常见，但这样的例子却不多。通常内层循环是设置代码，而在外层循环中有多个循环顺序执行，或者内层循环可能在外层循环所调用的方法之中。复杂的数据依赖关系使得没法用循环交换的方法解决问题。

内层循环可线程化而外层循环不可以线程化的情况很常见。我们将在本章后面的部分更为详细地考察内循环线程化的问题。

循环的重新实现

你可能已经注意到了，前面所设计的循环处理程序具有相当的局限性。它只适用于循环，而且在执行前必须知道循环的范围。它只能以整数作为索引，且迭代间隔只能是1。虽然其中一些限制是因为我们没有实现对线程处理程序一些功能的支持，但最大的原因却在于很难实现能够处理通用循环的算法（如果这不是不可能完成的工作）。

如果循环变形中其他的方法都失败了，编程经验依然十分有用。while循环和do

循环可以转变成for循环。开始和结束迭代可以在循环执行之前计算。代码可以移出/移入循环，或者在不同循环中移动，从而使得其他的循环变形可以发生。代码变化还可能导致变量分类的变化。共享变量可能还会因为它在循环中的使用方式的变化而变成循环私有变量或者归约变量。

不幸的是，我们并没有办法来担保成功。我们的目标是平衡开发的努力和获取的加速。结果却可能是花费很多天去实现一个改动，而从改动获得的加速只有百分之一二。毕竟，如果要进行无限制的努力，还不如从头开始重新设计整个应用程序。

内层循环线程化

当循环为嵌套时，我们先前讨论的问题都没有改变：如果你只把这些方法应用到内层循环，它们是起作用的。然而，还有另外一些很微妙的话题可以应用到内层循环。现在让我们回到两维的SinTable上。前面提到，循环交换应该允许外层循环被线程化。但现在，让我们不使用循环变形的方法，而试图去线程化内层循环：

```
public float[][] getValues() {
    for (int i = 0; i < 360; i++) {
        lookupValues[0][i] = 0;
    }
    for (int j = 1; j < 1000; j++) {
        for (int i = 0; i < 360; i++) {
            float sinValue = (float)Math.sin((i % 360)*Math.PI/180.0);
            lookupValues[j][i] = sinValue * (float)i / 180.0f;
            lookupValues[j][i]+=lookupValues[j-][i]*(float)j/180.0f;
        }
    }
    return lookupValues;
}
```

我们第一个要分类的变量是外层循环索引变量j。必须要对它分类的原因在于它被用于内层循环。在本例中，j是只读变量。乍一看，这样做并没有什么意义：索引变量怎么可能是只读的呢？我们必须把注意力只集中在我们试图线程化的范围内。在整个内层循环的执行过程中，这个变量只有一个不变的值。

虽然lookupValues数组变量是共享变量，但其元素却是循环私有的。因为循环的不同迭代依据循环索引和只读变量j来访问数组的不同成员，所以它的成员可以被看成是循环私有的。lookupValues数组的成员同时也是回存变量。不过，因为我们不产生这些变量的局部拷贝，所以没有必要把这些变量存放回去。

最后两个变量sinValue和i被简单地定为循环私有变量，每个线程都具有它们的单独拷贝。这些变量在循环结束后不会再被使用，所以也不必进行回存处理。

我们通过观察内层循环本身的算法来选择循环调度程序。在这个例子中，没有哪个迭代会比其他的迭代运行更长的时间。选择默认的静态或组块调度程序可能最好。当然，选择自调度或者引导型自调度也没有什么坏处。

完成了这些工作，就可以像以前一样使用循环处理程序对循环进行线程化。然而，这里有一点儿复杂：和外层循环相比，内层循环将运行更多的次数。这意味着线程的产生和销毁会执行更多次。而且，循环处理程序被设计成一次性对象。对外层循环的每个迭代都需要生成一个新的循环处理程序。虽然使用循环处理程序的工作不存在任何问题，但额外的负担比线程化外层循环大得多。

可以使用下面的方法来部分地解决这个问题：

```
public class PoolLoopHandler implements Runnable {
    protected class LoopRange {
        public int start, end;
    }
    protected ThreadPool poolThreads;
    protected int startLoop, endLoop, curLoop, numThreads;

    public PoolLoopHandler(int start, int end, int threads) {
        numThreads = threads;
        poolThreads = new ThreadPool(numThreads);
        setRange(start, end);
    }

    public synchronized void setRange(int start, int end) {
        startLoop = start;
        endLoop = end;
        reset();
    }
```

```
public synchronized void reset() {
    curLoop = startLoop;
}

protected synchronized LoopRange loopGetRange() {
    if (curLoop >= endLoop)
        return null;
    LoopRange ret = new LoopRange();
    ret.start = curLoop;
    curLoop += (endLoop-startLoop)/numThreads+1;
    ret.end = (curLoop<endLoop)?curLoop:endLoop;
    return ret;
}

public void loopDoRange(int start, int end) {
}

public void loopProcess() {
    reset();
    for (int i = 0; i < numThreads; i++) {
        poolThreads.addRequest(this);
    }
    try {
            poolThreads.waitForAll();
        } catch (InterruptedException iex) {}
    }

    public void run() {
        LoopRange str;
        while ((str = loopGetRange()) != null) {
            loopDoRange(str.start, str.end);
        }
    }
}
```

我们最初的loopHandler类只能使用一次的原因只是一个设计缺陷。循环索引从不被设置回循环的起始值，循环的范围也没法改变。为了解决这个问题，我们只需加入两个新方法：reset()和setRange()，用它们可以把索引值设置回循环的起始值，及指定循环的新范围。为了避免多次进行线程创建和销毁，可以使用我们在第七章中实现的ThreadPool类。这个方法现在不在loopProcess()方法中创建线程，而是把任务分配给了线程池中的线程。我们现在就只需等待池中所有的线程完成自己当前的任务了。这么做会有一些帮助，但是随之而来的同步将会对程序最终的加速性能产生负面影响。

关于内层循环的警告

在线程化任何循环前，我们应该检查这个循环。如果循环只是执行很短的时间，那就没有理由去线程化这个循环。在这些情况下，设置和拆卸线程化的循环带来的额外负载可能远远大于线程化所能带来的加速收益。

在从外层循环转到内层循环时，我们必须检查内层循环。因为外层循环可线程化并不代表内层循环也可以线程化。如果外层循环的迭代数目远高于内层循环，内层循环可能只会运行很短的时间。也有可能是外层循环中的方法调用执行很长的时间。

我们可以十分简单地在池处理程序中实现其他调度模型：

```
public class PoolSelfLoopHandler extends PoolLoopHandler {
    private int groupSize;
    public PoolSelfLoopHandler(int start, int end,
                               int size, int threads) {
        super(start, end, threads);
        setSize(size);
    }

    public synchronized void setSize(int size) {
        groupSize = size;
        reset();
    }

    protected synchronized LoopRange loopGetRange() {
        if (curLoop >= endLoop)
            return null;
        LoopRange ret = new LoopRange();
        ret.start = curLoop;
        curLoop += groupSize;
        ret.end = (curLoop<endLoop)?curLoop:endLoop;
        return ret;
    }
}
```

这里有意思的是它和我们最初的 SelfLoopHandler 类很相似。不过，为了提高它的可配置性，我们已经修改了循环处理程序以允许修改一些额外的参数，比如组块的大小。

下面看看如何使用新的循环处理程序:

```
public class SinTable extends PoolLoopHandler {
    private float lookupValues[][];
    private int j;
    public SinTable() {
        super(0, 360, 12);
        lookupValues = new float[1000][];
        for (int j = 0; j < 1000; j++) {
            lookupValues[j] = new float[360];
        }
    }

    public void loopDoRange(int start, int end) {
        float sinValue = 0.0f;
        for (int i = start; i < end; i++) {
            sinValue = (float)Math.sin((i % 360)*Math.PI/180.0);
            lookupValues[j][i] = sinValue * (float)i / 180.0f;
            lookupValues[j][i] += lookupValues[j-1][i]*(float)j/180.0f;
        }
    }

    public float[][] getValues() {
        for (int i = 0; i < 360; i++) {
            lookupValues[0][i] = 0;
        }
        for (j = 1; j < 1000; j++) {
            loopProcess();
        }
        return lookupValues;
    }
}
```

为了实现 SinTable 类，我们把内层循环的代码放在了 loopDoRange() 方法中，然后调用 loopProcess() 方法处理内层循环。由于索引变量 j 是只读共享变量，所以它现在是 SinTable 类的一个实例变量。

很重要的一点是要有一个可以多次使用的循环处理程序。如果使用早期的循环处理程序，我们就不得不为每个执行的内层循环生成一个新的循环处理程序实例。这意味着外层循环的代码和内层循环的代码不在同一个类中。而且，我们不得不向每个实例传递一个变量 j 和数组 lookupValues 的引用，因为它们在不同的内层循环处理程序中被共享。

循环输出

向文件或显示器写字符串的工作是I/O绑定的。在输出循环上使用多线程技术并没有太大意义。因为操作和I/O有关，线程的大部分时间被用来等待，与其让12个处理器空等，还不如只用1个。而且，输出的顺序也很重要。写入文件或显示器的字符串最终会被人或其他应用程序处理，不管计算是用一个线程还是多个线程完成，输出都应该是一样的。

可是，如果循环中输出部分远远少于数学计算部分，会怎么样呢？如果循环中有相当的部分是占用CPU较多的，却仅仅因为其中有一个println()方法调用就放弃对它进行并行化，可能是很愚蠢的。这时惟一要处理的问题只是输出顺序。它可以通过两步输出的方法进行解决：不再把数据直接写到显示器或文件，而先把它写入一个基于内存的虚拟输出，并为数据附上用来排序的索引；在循环结束时，再把输出送到显示器或文件，这时候，使用索引信息保证数据以正确的顺序输出。

让我们再次看看SinTable循环：

```
public synchronized float[] getValues() {
    if (lookupValues == null) {
        for (int i = 0; i < (360*100); i++) {
            float sinValue = (float)Math.sin((i % 360)*Math.PI/180.0);
            lookupValues[i] = sinValue * (float)i / 180.0f;
            System.out.println(" " + i + "    " + lookupValues[i]);
        }
    }
    return lookupValues;
}
```

在getValues()方法的新版本中，我们还是把表格输出到标准输出。显然，这个例子很简单，可以通过循环变形把计算分配到两个单独的循环中。但现在，姑且让我们假定输出过程和算法密切结合，没有办法对循环进行变形。

为了解决这个问题，我们将使用下面的类：

```
import java.util.*;
```

```
import java.io.*;

public class LoopPrinter {
    private Vector pStorage[];
    private int growSize;

    public LoopPrinter(int initSize, int growSize) {
        pStorage = new Vector[initSize];
        this.growSize = growSize;
    }

    public LoopPrinter() {
        this(100, 0);
    }

    private synchronized void enlargeStorage(int minSize) {
        int oldSize = pStorage.length;
        if (oldSize < minSize) {
            int newSize = (growSize > 0) ?
                oldSize + growSize : 2 * oldSize;
            if (newSize < minSize) {
                newSize = minSize;
            }
            Vector newVec[] = new Vector[newSize];
            System.arraycopy(pStorage, 0, newVec, 0, oldSize);
            pStorage = newVec;
        }
    }

    public synchronized void print(int index, Object obj) {
        if (index >= pStorage.length) {
            enlargeStorage(index+1);
        }
        if (pStorage[index] == null) {
            pStorage[index] = new Vector();
        }
        pStorage[index].addElement(obj.toString());
    }

    public synchronized void println(int index, Object obj) {
        print(index, obj);
        print(index, "\n");
    }

    public synchronized void send2stream(PrintStream ps) {
        for (int i = 0; i < pStorage.length; i++) {
            if (pStorage[i] != null) {
```

```
                    Enumeration e = pStorage[i].elements();
                    while (e.hasMoreElements()) {
                        ps.print(e.nextElement());
                    }
                }
            }
        }
    }
```

循环输出程序的实现中使用了一个两维向量。第一维用来分离输出。这个输出索引可能和实际循环的索引有关，也可能和循环组块有关，再或者，它可能是多个循环索引的组合。无论是哪种情况，一个输出索引都不能被分配给多个线程，因为索引化向量内部的排序基于其上。第二维用来保存要输出的字符串。因为索引已经给待输出的串排序，这一维只用来存储要被送到索引的字符串（注 2）。

输出对象到虚拟显示是用print()和println()方法完成的。在提供待输出的对象时，应用程序还必须同时提供一个索引作为输出顺序的参考。这些方法简单地保存字符串的索引以便它们能够在以后输出。第二阶段的输出过程用send2stream()方法完成。循环一结束，此方法就被调用，把结果输出到指定输出。

下面是 LoopPrinter 类的使用方法：

```
public class SinTable extends GuidedLoopHandler {
    private float lookupValues[];
    private LoopPrinter lp;

    public SinTable() {
        super(0, 360*100, 100, 12);
        lookupValues = new float [360 * 100];
        lp = new LoopPrinter(360*100, 0);
    }

    public void loopDoRange(int start, int end) {
        for (int i = start; i < end; i++) {
            float sinValue = (float)Math.sin((i % 360)*Math.PI/180.0);
            lookupValues[i] = sinValue * (float)i / 180.0f;
            lp.println(i, " " + i + " " + lookupValues[i]);
        }
```

注 2：　通常，我们可以用一维字符串缓冲区数组做用样的事情。

```
        }

        public float[] getValues() {
            loopProcess();
            lp.send2stream(System.out);
            return lookupValues;
        }
    }
```

循环输出程序要在循环之前创建，先前被送到文件或显示器的输出现在被送到循环输出程序，而且 send2stream()方法在循环结束时就被调用。由于循环输出程序将把所有的信息都发送到一个目标，所以如果循环要输出到不同流，就需要创建多个循环输出程序。

还要注意：当我们构造循环输出程序时，用索引大小作为它的初始大小。这里，循环输出程序可以扩展到任意大小，因此，不需要对它进行额外的定义。不过，我们希望避免扩大尺寸，因为这个操作不仅需要对该方法进行同步，而且还会根据大小的不同花费一些时间。print()方法和println()方法为此也必须同步。这是为了两个目的：第一，可以在不引起竞态条件的情况下增加数组大小；第二，允许方法在一个索引被分配给两个线程时还能工作，虽然此时输出顺序得不到保证。如果循环输出程序被修改为不允许增大数组，而且假定开发者不会把一个索引分配给两个线程，那么这个层次的同步就不需要了。

多处理器扩展

扩展这个术语有时被过度使用。它可以用于一台机器可以同时执行多少个应用程序，可以同时写多少个磁盘，或者一个百吉饼店工人同时可以处理多少份奶油干酪百吉饼订单。当不论增加多少资源，输出都无法再增加时，所得的极限值常被用来指明事物可扩展的程度。如果烤箱一个小时不能生产更多的百吉饼，那么生产线上无论增加多少个工人都没有用：百吉饼生产效率不可能超过烤箱烤饼的效率。可扩展极限也常被许多其他因素控制，比如奶油干酪的生产率、冰箱的大小，甚至百吉饼店的供应商供货的效率。

在这一章中，我们提到的多线程应用程序的可扩展性指的是通过增加处理器数目

可以获得加速的处理器数目的上限。在此极限上增加任何处理器，也无法让应用程序运行得更快。显然，应用程序的扩展程度如何依赖于很多因素：操作系统、Java虚拟机的实现、浏览器或者应用服务器以及Java应用本身。应用程序的最佳扩展性能将基于所有这些因素的扩展极限。

对理想世界中的理想CPU绑定程序，我们可能也期望得到理想的扩展性能：增加第二个CPU会让程序运行时间减半，再增加一个，运行时间就会减少到1/3，以次类推。然而，即使是对本章中研究过的那些基于循环的程序，我们也会看到扩展量依然受着以下这些重要约束的影响：

建立时间

执行并行化循环之外的代码需要花费一定的时间。这部分时间与可提供的线程和处理器数目无关，因为只有一个线程执行这些代码。

新的同步要求

在并行化本章的循环中，我们已经引入一些附加的簿记代码，其中一些是同步的。由于获取同步锁要消耗时间，这增加了执行代码需要的时间。

方法串行化

并行化代码中的一些方法是同步的，必须顺序执行。对与这些方法相关的锁的争夺同样影响待并行化的程序的可扩展性。

虚拟机的影响

影响特定程序可扩展性的因素之一是虚拟机本身的实现。比如，获取同步锁需要花费一定的时间，而且虚拟机中实际实现同步的代码本身也常常需要同步。两个试图获取不同同步锁的线程还是可能竞争虚拟机内相同的资源。此外，还有其他一些虚拟机或操作系统影响程序的可扩展性的例子。

本章中我们演示的结果基于Sun公司的Solaris 2.6 VM的1.1.6版。在其他的虚拟机和操作系统上将出现不同的结果：事实上，Solaris上的1.2 beta版就显示出更好的可扩展性，主要原因在于它提高了获取同步锁的效率（假定`loopGetRange()`方法要同步，这一点就十分重要）。Java 2 Solaris的产品发行版应该也能得到这样的结果。

如果我们把建立时间、同步时间和执行串行化方法的时间看成总执行时间的一部分，剩下的时间就是并行化代码运行的时间。我们要看到的最大可扩展量可以用Amdahl法则给出：

$$S = (1 - F) + \frac{F}{N}$$

这里，S是我们要看到的可扩展性，假定F%的代码被N个处理器并行处理。如果95%的代码被并行化，而我们有8个处理器，代码运行的时间将是原先时间的16.8%（0.05+0.95/8）。然而，我们引入了计算循环范围的代码（或者其他代码），实际上我们增加了串行化代码的数量，因此F有可能是个负数。在这种情况下，我们的并行化代码会比原来运行得更久。

那么，我们从本章讨论的技术中究竟能得到什么类型的扩展呢？为了回答这个问题，我们将测试双重循环例子的几种实现：

```
public float[][] getValues() {
    for (int i = 0; i < 360; i++) {
        lookupValues[0][i] = 0;
    }
    for (int j = 1; j < 1000; j++) {
        for (int i = 0; i < 360; i++) {
            float sinValue = (float)Math.sin((i % 360)*Math.PI/180.0);
            lookupValues[j][i] = sinValue * (float)i / 180.0f;
            lookupValues[j][i] += lookupValues[j-1][i]*(float)j/180.0f;
        }
    }
    return lookupValues;
}
```

为了方便测试，我们使用下面的类和接口创建一个系统，通过它，我们可以测试不同的循环处理程序。因为我们是在占用CPU较多的线程上工作的，所以我们已经包含了Solaris专用代码去设置LWP的数目，但是这段代码可以在任何操作系统下运行。

```
public interface ScaleTester {
    public void init(int nRows, int nCols, int nThreads);
    public float[][] doCalc();
}
```

```
import java.util.*;
import java.text.*;
import java.io.*;

public class ScaleTest {
    private int nIter = 200;
    private int nRows = 2000;
    private int nCols = 200;
    private int nThreads = 8;
    Class target;

    ScaleTest(int nIter, int nRows, int nCols, int nThreads,
              String className) {
        this.nIter = nIter;
        this.nRows = nRows;
        this.nCols = nCols;
        this.nThreads = nThreads;
        try {
            target = Class.forName(className);
        } catch (ClassNotFoundException cnfe) {
            System.out.println(cnfe);
            System.exit(-1);
        }
    }

    void chart() {
        long sumTime = 0;
        long startLoop = System.currentTimeMillis();
        try {
            ScaleTester st = (ScaleTester) target.newInstance();
            for (int i = 0; i < nIter; i++) {
                st.init(nRows, nCols, nThreads);
                long then = System.currentTimeMillis();
                float ans[][] = st.doCalc();
                long now = System.currentTimeMillis();
                sumTime += (now - then);
            }
        } catch (Exception e) {
            e.printStackTrace();
            System.exit(-1);
        }
        long endLoop = System.currentTimeMillis();
        long calcTime = endLoop - startLoop;
        System.err.println("Loop time " + sumTime +
                           " (" + ((sumTime * 100) / calcTime) + "%)");
        System.err.println("Calculation time  " + calcTime);
    }
```

```
    public static void main(String args[]) {
        if (args.length != 5) {
            System.out.println(
    "Usage: java ScaleTester nIter nRows nCols nThreads className");
            System.exit(-1);
        }
        ScaleTest sc = new ScaleTest(Integer.parseInt(args[0]),
                                     Integer.parseInt(args[1]),
                                     Integer.parseInt(args[2]),
                                     Integer.parseInt(args[3]),
                                     args[4]);
        CPUSupport.setConcurrency(Integer.parseInt(args[3]) + 5);
        sc.chart();
    }
}
```

在使用ScaleTest类时，我们得到两个数值：执行整个程序需要的毫秒数（包括初始化部分，这部分是单线程运行的）以及只执行计算循环的时间。这样，我们就可以比较这些数值，以确定循环处理类的不同实现的可扩展性。

我们将使用对下面这个类的度量作为比较的基线：

```
public class Basic implements ScaleTester {
    private float lookupValues[][];
    int nCols, nRows;

    public void init(int nRows, int nCols, int nThreads) {
        this.nCols = nCols;
        this.nRows = nRows;
        lookupValues = new float[nRows][];
        for (int j = 0; j < nRows; j++) {
            lookupValues[j] = new float[nCols];
        }
    }

    public float[][] doCalc() {
        for (int i = 0; i < nCols; i++) {
            lookupValues[0][i] = 0;
        }
        for (int j = 1; j < nRows; j++) {
            for (int i = 0; i < nCols; i++) {
                float sinValue =
                                (float)Math.sin((i % 360)*Math.PI/180.0);
                lookupValues[j][i] = sinValue * (float)i / 180.0f;
                lookupValues[j][i] +=
```

```
                                    lookupValues[j-1][i]*(float)j/180.0f;
            }
        }
        return lookupValues;
    }
}
```

这个类没有线程化；它是实现我们要测试的基本计算的通常方法。下面的循环处理程序类是我们要与之进行比较的一个实现：

```
public class GuidedLoopInterchanged implements ScaleTester {
    private float lookupValues[][];
    private int nRows, nCols, nThreads;

    private class GuidedLoopInterchangedHandler
                                        extends GuidedLoopHandler {
        GuidedLoopInterchangedHandler(int nc, int nt) {
            super(0, nc, 10, nt);
        }

        public void loopDoRange(int start, int end) {
            for (int i = start; i < end; i++) {
                lookupValues[0][i] = 0;
            }
            for (int i = start; i < end; i++) {
                for (int j = 1; j < nRows; j++) {
                    float sinValue =
                                (float)Math.sin((i % 360)*Math.PI/180.0);
                    lookupValues[j][i] = sinValue * (float)i / 180.0f;
                    lookupValues[j][i] +=
                                lookupValues[j-1][i]*(float)j/180.0f;
                }
            }
        }
    }

    public void init(int nRows, int nCols, int nThreads) {
        this.nRows = nRows;
        this.nCols = nCols;
        this.nThreads = nThreads;
        lookupValues = new float[nRows][];
        for (int j = 0; j < nRows; j++) {
            lookupValues[j] = new float[nCols];
        }
    }
```

```
        public float[][] doCalc() {
            GuidedLoopInterchangedHandler loop =
                        new GuidedLoopInterchangedHandler(nCols, nThreads);
            loop.loopProcess();
            return lookupValues;
        }
    }
```

这个类使用简单的循环处理程序处理循环；不过，要注意的是，我们已经交换了循环使得外层循环可以线程化。

表9-1列出了ScaleTest程序运行已交换循环的不同实现的结果：我们将组块、自调度、引导型自调度循环处理程序和我们刚才演示的代码结合在一起。这些测试在有8个CPU的机器上运行，使用的迭代次数是200。我们已把基线运行时间规范化为100，其他的数据可以用百分比来表示：我们得到的最佳结果是原先时间的20.6%。每个单元格的第一个数表示运行500行和1000列的结果，第二个数表示运行1000行和500列的情况。

表9-1：简单循环处理程序的可扩展性

	线程数目	总时间	循环时间
基本	1	100/100	96/96
组块调度	1	124.6/123.4	120.8/119.7
	2	64.5/63.1	61.2/59.3
	4	34.7/35.3	31.5/31.8
	8	23.7/23.0	20.3/19.3
	12	24.0/24.0	20.6/20.2
自调度	1	129.7/127.6	125.8/123.8
	2	71.9/70.3	69.0/66.8
	4	39.3/39.6	36.1/36.1
	8	23.1/24.1	19.8/20.5
	12	22.7/23.5	19.2/19.8

表9-1：简单循环处理程序的可扩展性（续）

	线程数目	总时间	循环时间
引导型自调度	1	124.7/122.5	120.9/118.9
	2	64.0/63.6	60.8/60.5
	4	34.4/34.2	31.3/30.8
	8	20.6/21.8	17.3/18.1
	12	22.3/23.1	18.9/19.1

从表中我们可以得出一些结论：

- 建立线程和循环处理类的开销很明显：在单线程情况下，将多花费22%到29%的时间。因此在单处理器机器上不应使用这种技术。
- 循环计算的可扩展性很好。在初始代码中，循环占96%，根据Amdahl法则，我们预期得到的最大可扩展性是16.8%。而实际上我们得到了20.6%，即90%的代码得到了并行化。6%的差别是因为对于loopGetRange()方法需要串行化调用，而且不同线程完成的工作量可能并不相等。
- 运行8个以上的线程效果并不好。这既是因为存在多个线程竞争一个CPU（处理器的数目为8），也是因为调用loopGetRange()方法会引起同步。
- 在这个例子中，引导型自适应调度方法具有最好的效果。这一点也不奇怪：对正弦值的计算并不总是花费相同的时间，所以让某个线程执行太长的时间会使组块调度程序受到损害。因为线程没有执行相同量的工作，所以扩展性受到了损害。

不过，总体来说，我们获得了很好的扩展性。

在测试中，回存变量起了什么作用呢？我们可以重写测试程序，以便每次计算lookup值的时候，就把它加到sumValue实例变量上去。使用前面提到的归约技术进行修改后，测试产生了表9-2中给出的数值。

表 9-2：有回存变量的循环处理程序的可扩展性

	线程数目	总时间	循环时间
基本	1	100/100	97/96
组块调度	1	123.3/121.9	119.6/118.3
	2	64.1/62.7	61.5/59.5
	4	36.4/35.2	33.4/32.0
	8	22.5/22.7	19.3/19.3
	12	24.1/23.7	20.9/20.1
引导型自调度	1	123.3/121.6	119.6/117.9
	2	64.6/63.2	62.0/60.0
	4	36.0/34.3	33.1/31.
	8	20.2/21.5	17.1/18.0
	12	22.1/22.3	19.0/18.7

由于回存变量只有一个，所以对可扩展性的影响很小。事实上，结果在某些情况下更好的原因在于基线现在执行了更长的时间。尽管如此，多个回存变量累积起来可能会产生更明显的影响。

只线程化内层循环又会怎样呢？这个问题很有意思，因为它示范出内层循环较小时的同步开销和我们所节省的时间的比较。重写我们的第一个测试（没有回存变量），也不再交换循环，而对内层循环线程化，得到的结果如表 9-3 所示：

表 9-3：内层循环处理程序的可扩展性

	线程数目	总时间	循环时间
基本	1	100/100	97/96
引导型自调度	1	138.0/159.7	133.8/155.0
	2	82.2/138.3	77.2/131.4
	4	66.7/164.1	60.0/154.2
	8	104.3/515.3	92.8/499.9
	12	1318.9/4466.3	1292.5/4421.7

现在的情况如何呢？首先，我们简单地改动了测试的参数：第一个数值是运行100行5000列产生的结果，第二个是运行500行1000列产生的结果。在第一种情况下，我们在有4个CPU时获得了一些扩展性，它允许我们在每个CPU上运行250次计算。而当CPU数目增加到8个时，每个内层循环只有125次计算，重复调用同步的loopGetRange()方法所增加的负担已经超过了我们并行化小循环所获得的收益。而继续增加线程数目就使情况变得极为糟糕了。

在第二种情况下，内层循环十分小，而我们频繁地调用loopGetRange()方法，结果没有任何的扩展性可言。在最好的情况下（2个线程），我们就比所并行化的代码增加了相当于43%的代码量。

所以，就像我们在前面提到的，线程化小循环（尤其是小的内层循环）是没有必要的。

最后，如果我们向循环中加入输出计算结果的代码又会怎样呢？我们依然使用前面设计的LoopPrinter类对这种情况进行线程化。而且，让我们回想一下在结束LoopPrinter类一节时对从LoopPrinter类中去除同步的讨论。在现在这个测试中，我们总是知道输出数组的大小，而且可以保证同一个索引不会用于不同的线程，因此，我们可以重写LoopPrinter，如下所示：

```
import java.util.*;
import java.io.*;

// 循环输出程序的非线程安全版本
public class LoopPrinter {
    private Vector pStorage[];

    public LoopPrinter(int size) {
        pStorage = new Vector[size];
    }

    public void print(int index, Object obj) {
        if (pStorage[index] == null) {
            pStorage[index] = new Vector();
        }
        pStorage[index].addElement(obj.toString());
    }
```

```
    public void println(int index, Object obj) {
        print(index, obj);
        print(index, "\n");
    }

    public void send2stream(PrintStream ps) {
        for (int i = 0; i < pStorage.length; i++) {
            if (pStorage[i] != null) {
                Enumeration e = pStorage[i].elements();
                while (e.hasMoreElements()) {
                    ps.print(e.nextElement());
                }
            }
        }
    }
}
```

在循环输出程序的这个新版本中，不再包含同步代码，因此它在扩展方面少了很多问题。然而，由于对Vector类的调用，即使是这个循环输出程序版本也给我们的多线程程序增加了大量的额外负担。而且，它还花费更多的时间向这些向量写字符串，然后把它们转储出来，而不是简单地调用System.out.println()方法。尽管如此，这个类的线程安全版本和非线程安全版本之间的差别还是很明显的。表9-4列出了我们从两种情况获得的结果。

表9-4：两个LoopPrinter类的可扩展性

	线程数目	总时间	循环时间
基本	1	100/100	96/98
线程安全循环处理程序	1	125.4/126.0	116.7/119.7
	2	79.0/97.8	70.3/91.9
	4	55.5/82.5	47.2/76.7
	8	46.6/84.2	38.2/78.3
	12	48.2/86.9	39.5/80.0
非线程安全循环处理程序	1	125.1/121.0	116.3/111.3
	2	77.9/92.7	69.4/85.3
	4	55.3/79.0	47.0/67.1
	8	45.6/78.2	37.0/64.9
	12	47.7/78.2	39.1/64.9

表中的第一组数是对200行1000列的表运行200次迭代，而且每100行输出一次。第二组数则是每20行输出一次结果。在第二种情况中，额外代码使得根本不可能获得扩展性。这个例子清楚地说明：审慎的设计和使用非同步化类可以带来很大的好处。

可以看到这种方法和我们先前对生成线程安全代码的警告是相互矛盾的。我们依然建议总是从线程安全代码开始。不过，在和现在类似的情况下，如果你花费了足够的精力，能保证你能正确使用非线程安全代码，那么在扩展性方面所得到的好处可能会超过为避免竞态条件而认真编码所花费的额外努力。

总结

在本章中，我们讨论了让Java程序在多处理器机器上更快运行的技术。我们考察了循环（最常见的占用CPU较多型代码源），而且开发了让这些循环在多线程方式下运行的类。在此过程中，我们对变量进行分类，使用不同的调度算法，并应用简单的循环变形以获得并行化。

本章的目标是从头开始编写更快的程序，在不重新设计的情况下提高已有算法的性能以及为要求高性能的场合提供丰富的选择方法。

第十章

线程组

本章内容：

- 线程组概念
- 创建线程组
- 线程组方法
- 操作线程组
- 线程组、线程和安全
- 总结

在这一章中，我们将讨论Java的ThreadGroup类。顾名思义，这个类被用来处理一组线程。它被用于以下两个方面：一，允许通过调用单个方法操作多个线程；二，提供Java安全机制和线程交互的基础。在Java 1.0中，线程组的实际应用只限于Java应用程序：对于applet而言，事实上并不存在对线程组的操作。造成这种情况的原因一方面是由于安全限制，另一方面也是因为API中存在着bug。在以后的Java版本中，这种情况得到了改变，线程组现在可以用于任何Java程序。

线程组概念

假定你要使用我们在第五章中设计的TCPServer类编写一个服务器。每个连到服务器的客户都要作为一个单独的线程运行。而且服务器要为每个客户创建多个其他线程，比如定时器线程、读取客户数据的线程、向客户写数据的线程或者一些用于计算的线程。那么，服务器就要管理很多的线程。

这种场合正是ThreadGroup类发挥作用的地方。线程组允许你在一次调用中修改多个线程，这样既方便了对线程的操作，也降低了出现遗漏的可能性。

虽然我们在前面没有提到过线程组，但它们其实一直不离我们左右：Java虚拟机中的所有线程都属于线程组。我们创建的任何线程都自动属于Java虚拟机为我们设定的默认线程组。也就是说：前面所考察的所有线程都属于一个已有线程组。

多个线程组之间的关系并不是杂乱无章的，它们彼此关联。每个线程组（显然，第一个线程组例外）都有一个父线程组，因此线程组是以一种树状层次关系存在的。树根就是系统线程组。

你可以创建自己的线程组；每个线程组都是一个已有线程组的儿子。在刚才讨论的 TCPServer 的例子中，线程的层次关系可能如图 10-1 所示：

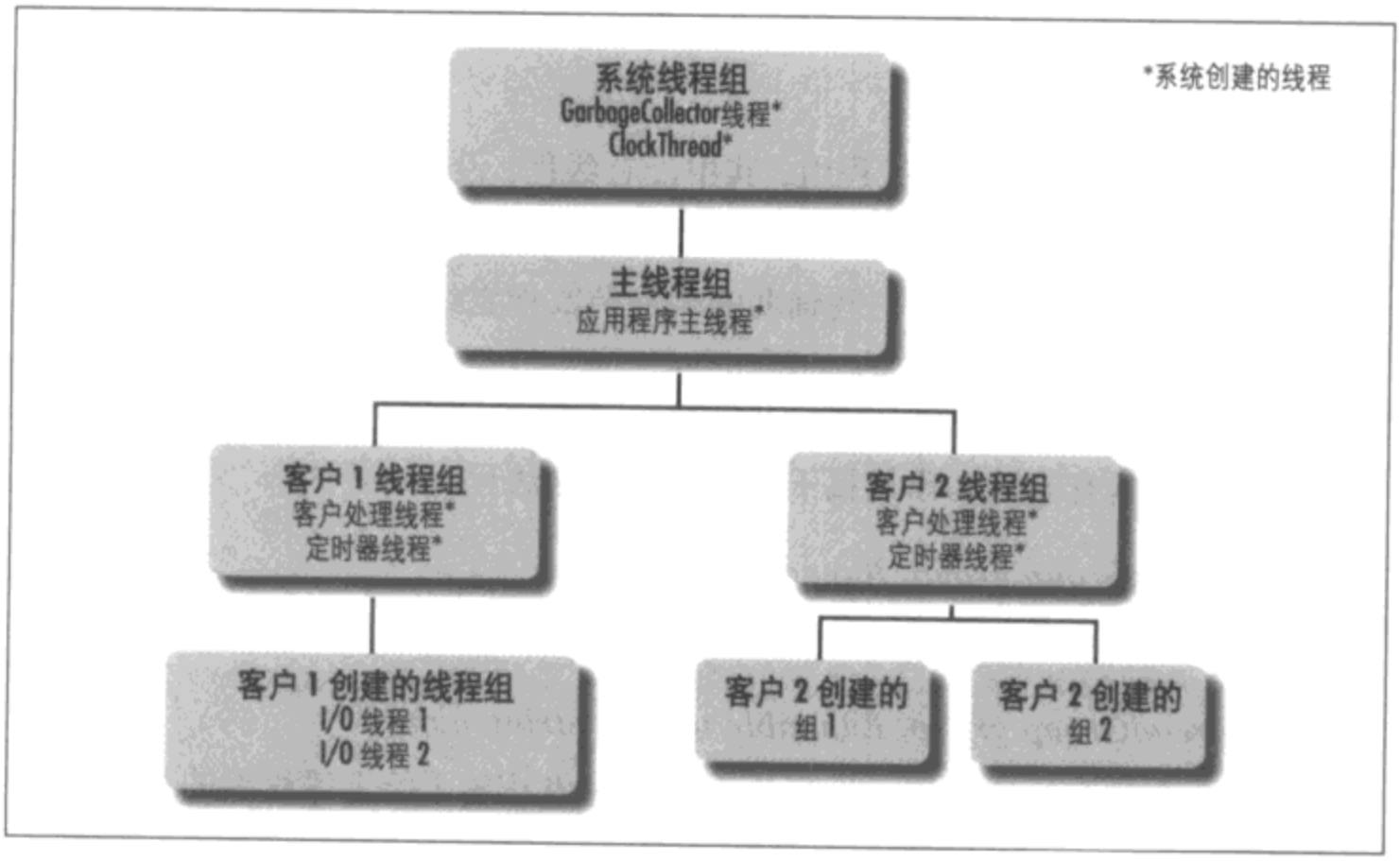

图 10-1：线程组层次图

我们最终为每个连接的客户创建了至少一个线程组；注意可以在线程组之下创建其他线程组。还要注意线程是分散到整个层次关系的各个组中的：一个线程组既包括了线程，也（可能）包括其他线程组。

创建线程组

有两个构造函数可以用来创建新的线程组：

ThreadGroup(String name)

用给定名称创建线程组。

ThreadGroup(ThreadGroup parent, String name)

用给定名称创建给定父线程组的子线程组。

在第一个构造函数中，新线程组是当前线程所在线程组的儿子；而在第二种情况下，新线程组被插到线程层次关系中，作为给定线程组的儿子（但是这么做可能不大好，在默认情况下，线程组可以被插到Java应用程序中线程层次关系的任何地方）。在Java 1.0中，只有Java应用程序可以创建线程组；现在该限制被取消了。

这些构造函数创建一个空的线程组，其中没有线程。没有什么方法能把线程移入指定线程组；只有在创建线程对象的时候才能把线程放到线程组里。从这个限制可以想到，Thread类中会有一些辅助的构造函数指定线程应该属于哪个组：

Thread(ThreadGroup group, String name)

构造一个具有给定名称且属于给定线程组的线程。

Thread(ThreadGroup group, Runnable target)

构造一个运行给定目标对象且属于给定线程组的新线程。

Thread(ThreadGroup group, Runnable target, String name)

构造一个具有给定名称、运行给定目标对象且属于给定线程组的线程。

要注意到没有只用ThreadGroup做参数的构造函数，这看起来好像是个疏忽。我们在第二章中学习的构造函数会使线程成为当前线程所属线程组的成员。

类似地，也没有方法用来从线程组中删除线程：线程在生命周期中一直属于某个线程组。当线程结束的时候，会被自动从线程组中删除。

我们可以使用这些构造函数修改TCPServer类，使得每个客户都既在不同的线程中运行，也被放入不同的线程组。这么做很简单：只要紧挨在创建客户线程之前创建线程组就行了，这样当客户线程启动时，它就是新的线程组的成员了：

```
import java.net.*;
import java.io.*;
```

```
public class TCPServer implements Cloneable, Runnable {
    Thread runner = null;
    ServerSocket server = null;
    Socket data = null;
    volatile boolean shouldStop = false;
    ThreadGroup group = null;
    int groupNo = 0;

    public synchronized void startServer(int port) throws IOException {
        if (runner == null) {
            server = new ServerSocket(port);
            runner = new Thread(this);
            runner.start();
        }
    }

    public synchronized void stopServer() {
        if (server != null) {
            shouldStop = true;
            runner.interrupt();
            runner = null;
            try {
                server.close();
            } catch (IOException ioe) {}
            server = null;
        }
    }

    public void run() {
        if (server != null) {
            while (!shouldStop) {
                try {
                    Socket datasocket = server.accept();
                    TCPServer newSocket = (TCPServer) clone();

                    newSocket.server = null;
                    newSocket.data = datasocket;
                    newSocket.group =
                        new ThreadGroup("Client Group " + groupNo++);
                    newSocket.runner =
                        new Thread(newSocket.group, newSocket);
                    newSocket.runner.start();
                } catch (Exception e) {}
            }
        } else {
            run(data);
        }
    }
```

```
        public void run(Socket data) {
        }
    }
```

在第五章中，我们用TCPServer的子类向客户提供服务；在下一节中，我们将会看到如何用线程组来简化处理客户的编码。

线程组方法

不同于下一节中要考察的那些不被推荐使用的方法，ThreadGroup类的方法通常可以提供丰富的信息。在这一节中，我们将介绍ThreadGroup类的所有方法。

查找线程组

经常会有这种情况：你希望调用线程组方法，但手头却没有线程组对象。线程类提供了一个方法用以返回线程对象所在线程组的引用：

ThreadGroup getThreadGroup()

返回线程的线程组引用，比如：

```
// 找到当前运行线程所属的线程组
ThreadGroup tg= Thread.currentThread().getThreadGroup();
```

还可以用ThreadGroup类的getParent()方法找到一个已有线程组的父线程组：

ThreadGroup getParent()

返回线程组的父线程组的ThreadGroup引用。

最后，还可以使用ThreadGroup类的parentOf()方法检验某个特定的线程组是否是另一个线程组的祖先：

boolean parentOf(ThreadGroup g)

如果g是此线程组的祖先，则返回true。

注意：parentOf()方法的命名并不准确；不管g是调用线程组、调用线程组的父亲，还是调用线程组的祖父，以及线程组在层次关系中的某个祖先，它都会返回true。

枚举线程组

下面讲解的一组方法允许你得到线程组中所有线程的列表。枚举线程的确是线程组类的职责：虽然Thread类也包含一些枚举线程的方法，但这些方法只是简单地调用ThreadGroup类的对应方法。

ThreadGroup类有两个基本的方法可以返回线程列表：

int enumerate(Thread list[])

把此线程组及其子孙线程组中所有线程的引用填入list数组。

int enumerate(Thread list[], boolean recurse)

把此线程组中所有线程的引用填入list数组；如果recurse为true，则此线程组及其子孙线程组中所有线程的引用都被填入list数组。

这些调用把每个符合条件的线程的引用填入list数组，并返回插入到数组中的线程的数目。线程是否符合条件取决于recurse参数：如果recurse为true，不仅指定线程组的线程被返回，而且该线程组的子孙线程组中的线程也会被返回。当recurse参数为false时，显然只会返回该线程组的成员线程。

对系统线程组调用enumerate()方法，并把recurse设置为true，将会返回虚拟机上的所有线程。你可以通过刚才讲到的getParent()方法找到系统线程组（当然，这可能要受到安全模型的限制）。

由于Java数组是定长数组，所以在调用enumerate()方法前，必须确定list参数的大小（否则可能得不到完整的列表）。我们可以使用activeCount()方法，得到list数组的准确大小：

int activeCount()

返回本线程组及子孙线程组中活动线程的数目。

这个方法没有递归选项；activeCount()总是返回当前线程组及其所有子孙线程组中的线程数目。

下面的代码段演示了如何使用这些方法显示当前线程组的线程。修改enumerate()方法的参数可以显示当前线程组及其子孙线程组中的线程：

```
ThreadGroup tg = Thread.currentThread().getThreadGroup();
int n = tg.activeCount();
Thread list[] = new Thread[n];
int count = tg.enumerate(list, false);
System.out.println("Threads in thread group " + tg);
for (int i = 0; i < count; i++)
    System.out.println(list[i]);
```

通过使用具有下面签名的enumerate()方法可以得到ThreadGroup对象列表，而不是Thread对象的列表：

int enumerate(ThreadGroup list[])

获取给定线程组的所有子孙线程组的引用。此方法在线程组层次上进行递归操作。

int enumerate(ThreadGroup list[], boolean recurse)

获取给定线程组的所有直接子孙的线程组引用；如果recurse为true，则返回线程组所有子孙的线程组引用。

这些方法在概念上和我们刚才讨论的方法相同。为了确定list参数的大小，需要使用activeGroupCount()方法：

int activeGroupCount()

返回给定线程组（在任何层次上）的线程组子孙的数目。

以前提到过，Thread类也有一个enumerate()方法。它总是进行递归搜索；它实际上是下面代码的简写：

```
Thread.currentThread().getThreadGroup().enumerate(list, true);
```

与之类似，Thread类的activeCount()方法实际上是下面代码的简写：

```
Thread.currentThread().getThreadGroup().activeCount();
```

最后，还有一个只用于测试的方法：

void list()

把当前线程组的所有线程列表发送到标准输出。

线程组中与优先级相关的调用

Java线程组中有一个最大优先级的概念。这个最大优先级和Thread类的优先级方法相互作用：不能把线程的优先级设得比所属线程组的最大优先级高。默认情况下，线程组的最大优先级与其父线程组的最大优先级相同。系统线程组的最大优先级是10（Thread.MAX_PRIORITY）。applet线程组（applet中所有线程所属的线程组）的最大优先级只有6。

有两个方法可以处理线程组的优先级：

void setMaxPriority(int priority)

设置线程组的最大优先级。

int getMaxPriority()

得到线程组的最大优先级。

在Java虚拟机的参考实现版本中，线程组的最大优先级不动声色地起着强制作用：如果线程所属的线程组的最大优先级是6，而你想把线程的优先级提高到8，最终线程优先级还是会被悄悄地设置为6。在一些浏览器（以及Java 1.0）中，如果试图把个体线程的优先级设为超过线程组的最大优先级，就会抛出SecruityException异常。

线程组的最大优先级一旦被降低，就不能再被提高。

不过，只有在线程优先级被实际改变的时候才会检查线程组的最大优先级。因此，如果一个最大优先级为10的线程组里包含一个优先级为8的线程，把线程组最大优先级改为6并不影响该线程：它的优先级还是8，除非调用了这个线程的set-

Priority()方法。不过，所有嵌套线程组的最大优先级都会立刻被修改：目标线程组包含的任何子线程组的最大优先级都会被降到请求值。这种变动在线程组层次上递归传播。

销毁线程组

可以用 destroy()方法销毁线程组：

void destroy()

清理线程组并把它从线程组层次关系中删除。

destroy()方法的使用是受到限制的：只有当线程组不包含线程时，才可以调用它。destroy()方法是递归操作的，它不仅销毁目标线程组，而且会销毁目标线程组的子孙。如果这些线程组中存在着活动的线程，destroy()方法会产生 IllegalThreadStateException 异常。

可以用下面的方法来检查是否已经调用 destroy()方法销毁指定线程组：

boolean isDestroyed()(只用于 Java 1.1 及以上版本)

返回一个指示线程组是否已被销毁的标志。

这个方法看起来有些奇怪：既然线程组都已经被销毁了，又怎么能在其上执行一个方法呢？答案在于：destroy()方法只是把线程组从组层次关系上删除，实际的线程组对象存在，直到不再存在对它的引用时才会被垃圾收集程序回收。

守护线程组

ThreadGroup 类上也有守护线程组的概念，它有点类似于守护线程。不过，两者并不相关：守护线程可以属于非守护线程组，守护线程组也可以包含非守护线程。守护线程组的好处在于：当其中不再包含线程和线程组时，它会被自动销毁。和线程不同，线程组的守护状态在任何时候都不能被修改：

void setDaemon(boolean on)

修改线程组的守护状态。

boolean isDaemon()

如果线程组是守护线程组则返回 `true`。

这里要强调一点：守护线程组只有在其中所有的线程都实际退出后才能被销毁：如果守护线程组中还有守护线程，守护线程组也只能等守护线程被停止后才能被销毁。这是因为守护线程一直在虚拟机中服务用户线程，而不只是特定线程组的用户线程。

当然，守护线程的最大好处就是程序员不需要费力显式地去终止它。因而，虽然只要包含守护线程，守护线程组就会还自动存在的概念很吸引人，但实际上却不能这样编程。

其他方法

到现在为止，ThreadGroup 类还剩下 3 个方法：

String getName()

返回线程组的名字。

void uncaughtException(Thread t, Throwable e)

当线程由于未捕获的异常而退出时会调用这个方法；它的默认行为是向 `System.err`打印线程的堆栈轨迹。在附录一中我们将会对这个方法进行更多的说明。

boolean allowThreadSuspension(boolean b)(只用于 Java 1.1)

设置线程组的 `vmAllowSuspension`标志为 `b`，并返回原先的值。如果虚拟机在低内存时运行，一些虚拟机实现会试图挂起 `vmAllowSuspension`标志为 `true`的线程组中的线程以获取更多内存。

不过，由于 `suspend()`方法本身在版本 2 中已经不推荐使用，所以虚拟机不再会挂起 `vmAllowSuspension`标志为 `true`的线程组中的线程，这个方法也就没什么用处了。

操作线程组

线程组真正有用的地方就是可以一次操纵所有线程的能力。ThreadGroup类提供了4个这样的方法；不过，其中3个现在都已经不被推荐使用，这个想法也已经不像它过去那么有用了：

void suspend()(Java 2 不推荐使用)

挂起源自该线程组的所有线程。

void resume()(Java 2 不推荐使用)

恢复源自该线程组的所有线程。

void stop()(Java 2 不推荐使用)

停止源自该线程组的所有线程。

void interrupt()(只在Java 2及以上版本中使用)

中断源自该线程组的所有线程。

这些方法都和它们在Thread类中的对应函数以相同的方式工作，但它们不仅影响该线程组中的所有线程，而且影响该线程组所有子孙线程组中的线程。换句话说，这些方法对所有源自指定线程组的线程组递归操作。在我们TCPServer线程组层次的例子中，这意味着如果我们中断了Client1线程组，那么也会中断Client1创建的线程组中的I/O线程（注1）。

在创建TCPServer的子类时，我们可以使用这些调用来简化编程。在ServerHandler子类中，我们省略那些为客户进行的处理。这次，我们假定服务器从客户读取一系列命令，而且把各个命令放到不同的线程中运行；这样，客户可以异步传送命令，而不用等待服务器完成前面的命令。通过把所有这些线程放到一个组中，我们可以利用线程组机制使用一个调用去修改所有为用户运行的线程。

注1：我们知道你肯定渴望自己去试一试。不过，如果你在Java应用程序中挂起了系统线程组，那么虚拟机中的所有线程都会被挂起，从而挂起了虚拟机。但对于Java applet来说情况就不同了，这主要是由于我们将在后面讨论的安全限制。

在这个例子中，我们使用下面这种机制处理客户关闭连接的情况：通过一个调用，我们可以中断所有为这个客户运行的线程（这里假定线程会像第四章中的例子那样周期性的检查它们的中断状态，发现状态为 true 时退出）。

我们另外还创建了一个线程组，把所有客户线程都加入其中。最后，就形成了下面这些线程组：

- TCPServer 线程组，包括监听客户请求的线程。
- 和客户进行通信的线程的线程组（每个客户都有一个）。这就是我们前面在 TCPServer 例子中创建的线程组。
- 客户的计算线程组，包括为客户进行计算的所有线程。这是我们将在下面代码中创建的线程组。

另外一个有用的技术是：创建一个在线程组外的线程来实际操作线程组。虽然这不是绝对的要求，但是它确实很方便，比如：你可以用它来中断你所属的线程组。

下面是修改后的 ServerHandler 类，已经添加了额外的线程组逻辑：

```
import java.net.*;
import java.io.*;

class CalculateThread extends Thread {
    OutputStream os;
    CalculateThread(ThreadGroup tg, OutputStream os) {
        super(tg, "Client Calculate Thread");
        this.os = os;
    }
    public void run() {
        // 进行计算，将结果返回给 OutputStream os
        // 确保经常检查 isInterrupted()标志
    }
}

public class ServerHandler extends TCPServer {
    public static final int INTERRUPT = 0;
    public static final int CALCULATE = 1;
    ThreadGroup tg;

    public volatile boolean shouldRun;
```

```
private int getCommand(InputStream is) {
    // 从输入流读取命令数据并且返回命令
}

public void run(Socket data) {
    tg = new ThreadGroup("Client Thread Group");
    try {
        InputStream is = data.getInputStream();
        OutputStream os = data.getOutputStream();
        while (shouldRun) {
            switch(getCommand(is)) {
                case INTERRUPT:
                    tg.interrupt();
                    break;
                case CALCULATE:
                    new CalculateThread(tg, os).start();
                    break;
            }
        }
    } catch (Exception e) {
        tg.interrupt();
    }
}

public static void main(String args[]) throws Exception {
        TCPServer serv = new ServerHandler();
        serv.startServer(300);
    }
}
```

线程组、线程和安全

在本章前面提到的对applet的多种限制来自于Java的安全机制。Java在很多方面都有安全机制：在语言本身中，在虚拟机中，并内置于Java API中。在讨论线程时，我们只需考虑API中的安全机制，这一节就将考察这些机制如何影响线程和线程组。ThreadGroup类的一个主要动机就是用来加强安全。

Java线程安全是通过SecurityManager类实施的；Java程序中的安全策略在类被实例化和安装到虚拟机中时建立。当试图对线程和线程组执行特定的操作时，API将询问安全管理器以决定是否允许这些操作。在Java 2之前，Java应用程序中并没有安全管理器，除非你自己动手编写并安装一个；这也是我们刚才讨论的

操作在Java应用程序中都合法的原因。不过，在Java applet中，通常都有一个安全管理器去实施特定的限制。

浏览器和安全管理器

在编写Java applet时，你并没有机会去改变安全管理器：它是被浏览器本身初始化和安装的，而且，一旦安装，就不能改变。

Java规范并没有指明安全管理器应该实施哪种政策。这一级别的安全策略是由特定浏览器决定的。不同的浏览器可能实施不同级别的安全：比如，Netscape浏览器不允许Java程序从用户的本地磁盘上读取任何文件，而Sun公司的HotJava浏览器允许用户指定一个目录列表，applet可以从中读取文件。

这里的法则就是由Java应用程序的作者最终决定使用什么安全策略；对于浏览器而言，浏览器的作者就是应用程序的作者。因此，不同的浏览器就有着不同的安全模型和策略。

在Java 2中，applet依然是被安全管理器所限制，但是现在也有一些新的方法来在默认安全管理器的管理下运行Java应用程序。当然，应用程序还是可以用传统方式安装它们自己的安全管理器（或者在没有安全管理器的情况下运行）。

SecurityManager类中对于Thread类和ThreadGroup类分别有处理安全策略的方法，它们有相同的名称，不过方法签名不同：

void checkAccess(Thread t)

检查当前线程是否可以修改线程t的状态。

void checkAccess(ThreadGroup tg)

检查当前线程组是否可以修改线程组tg的状态。

和SecurityManager类的其他方法一样，如果发现要执行的操作和安全策略冲突，这些方法就会抛出SecurityException异常。例如，下面是Thread类的interrupt()方法的实现代码（这实际上是对Thread类包含代码的精简）：

```
public void interrupt() {
    SecurityManager s = System.getSecurityManager();
    if (s != null)
        s.checkAccess(this);     // this 就是Thread.currentThread();
    interrupt0();
}
```

通过调用 checkAccess()方法来保证线程安全是一个规范的编程方式：如果操作违反了线程策略，该方法会产生一个运行时异常。假定没有抛出异常，就会调用一个内部方法来实际执行这个方法的逻辑。

安全和 checkAccess()方法

Thread 和 ThreadGroup 类都有名为 checkAccess()的内部方法。默认情况下，这个方法以线程或者线程组对象作为参数来调用安全管理器的 checkAccess()方法。

Thread 和 ThreadGroup 类的 checkAccess()方法都是公共的，如果想要检查什么安全策略在起作用，可以从任何线程和线程组对象直接调用它。

ThreadGroup 类的 CheckAccess()方法是最终的；不能被覆盖。而 Thread 类的 checkAccess()方法不是最终的，你可以覆盖它，改变特定线程的安全策略（需要记住的是：这样只会影响你自己的线程类，而不会影响系统中的其他线程）。

因为SecurityManager类中只为Thread类和ThreadGroup类分别提供了一个方法，所以线程的安全策略就只会是“全是”/“全否”的命题。如果安全管理器确定特定的线程不可以中断其他线程，那么这个线程也不能设置其他线程的优先级。不过，安全管理器可以（而且通常）考虑线程的上下文信息（包括它的线程组）以确定这个线程的策略。

表 10-1 给出了 Thread 和 ThreadGroup 类中调用安全管理器来确定操作是否合法的方法列表。注意这组方法包括所有创建以及其他改变特定线程或线程组的状态的方法，却不包括任何提供线程信息的方法（比如 enumerate()方法或 get-

Priority()方法)。因此，无论应用安装了什么安全管理器，任何线程都能够检查虚拟机上所有其他的线程；线程只(可能)被限制不能去修改其他线程的状态。

表 10-1：受安全管理器影响的线程和线程组方法

Thread 方法	ThreadGroup 方法
Thread()[包括所有的签名]	ThreadGroup()[包括所有签名]
stop()[包括两种签名]	stop()
suspend()	suspend()
suspend()	suspend()
resume()	resume()
interrupt()	interrupt()
setPriority(int priority)	setPriority(int priority)
setDaemon(boolean on)	setDaemon()
setName(String s)	destroy()

安全管理器建立的控制完全由Java应用程序或者启用Java的浏览器的作者决定，所以，我们无法准确预知线程上的哪些操作是允许的。虽然如此，我们还是在此列出一些最常见的情况：

Java 1.0.2 和 1.1 应用程序

默认情况下，这些版本的应用程序没有安全管理器，所有线程都可以对其他任何线程执行任意操作。当然，如果应用程序的作者决定安装安全管理器，情况会有所不同。

基于 Java 1.0.2 的浏览器

这包括 1.0.2 版的 appletview、Internet Explorer 3.0 和 Netscape 3.0。在这些浏览器中，每个 applet 都具有自己的线程组。applet 可以在自己的线程组中创建线程，但是，尽管 applet 可以创建其他线程组，它却不能向该线程组中加入线程。因此，图 10-2 中的 applet 1 可以创建子组 1、子组 2，但无法创建线程 C、线程 D。applet 无法向其他线程组添加线程的事实也使得它创建线程组的能力没什么用处。

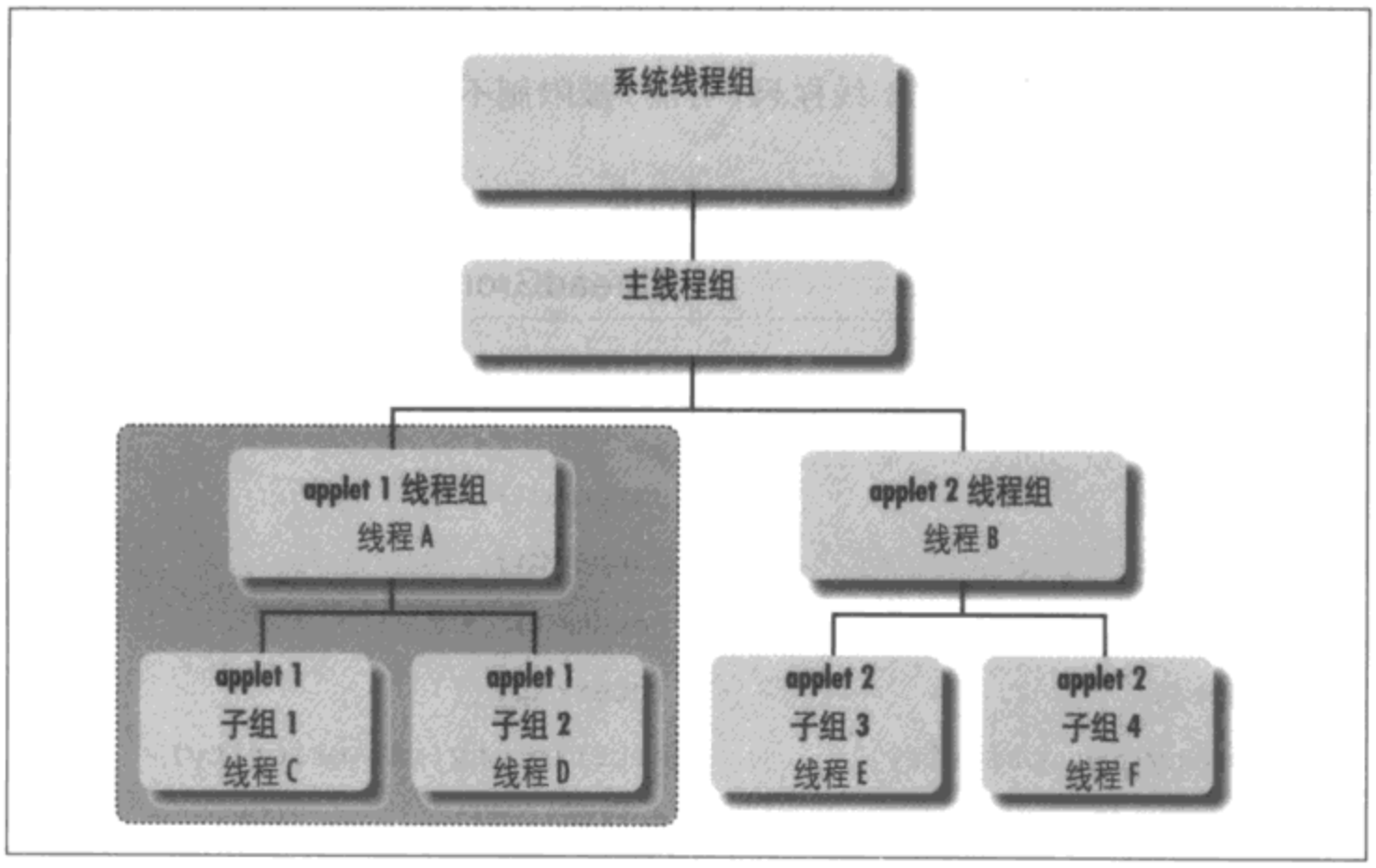

图 10-2: 启用 Java 的浏览器中可能的线程

在这个线程层次关系中，applet线程可以修改其他任何线程组和其他任何线程，包括不相关 applet 中的线程（比如，线程 A 可以修改线程 B）。

基于 Java 1.1 的浏览器

基于 Java1.1 的浏览器包括 1.1 版的 appletviewer、Internet Explorer 4.0 和 Netscape 4.0. 虽然这些浏览器有一个通用的参考基础，但是在线程安全的实现方式上却存在差别。在 appletviewer 的情况下，浏览器中的每个 applet 被给予一个惟一的线程组，applet 可以创建线程组，并把它安装到 applet 线程组之下的线程组层次关系中。在图 10-2 中，浏览器可以创建 applet 1 线程组，applet 本身可以去创建子组 1、子组 2，阴影框描绘了属于 applet 1 的线程组。

图 10-2 阴影框中的任何线程都能够访问框中的其他线程。因此，线程 A 可以操作线程 C 和 D，线程 C 也可以操作它的父线程（线程 A）以及兄弟线程组中的线程（线程 D）。但是，applet 线程不可以访问系统线程或者主线程，也不可以访问任何不属于自己线程组集合的线程（线程 C 和线程 D）。然而，在 Netscape 中，applet 线程可以访问它们父亲（即主线程组）的线程。更让

人觉得奇怪的是，applet能够访问和操作任意线程组，包括系统线程组和主线程组。

在Internet Explorer 4.0中，线程安全的基本思想被稍微做了修改。首先，IE 4.0不允许applet调用getParent()方法去获得系统线程组和主线程组。这是对核心API的改变，我们在前面说过，核心API是不做这种安全检查的。所以，IE 4.0中的applet线程可以操作任何它能够访问的线程，但是，这种访问被限制在了applet本身（比如，图10-2中的阴影框）。

在Netscape 4.0中，除了浏览器为applet创建的默认线程组外，applet不能在其他线程组中创建线程。而且Netscape 4.0中的enumerate()方法并不返回applet线程组以外其他线程组的线程集合，因此想捕获applet外的其他线程是不可能的。

Java 2 应用程序

默认情况下，Java 2应用程序和基于1.0.2与1.1的应用程序以同样的方式工作：没有安全管理器，任何线程都能够访问其他线程。

不过，如果在启动Java 2应用程序时，开启了-Djava.security.manager选项，就会为应用程序安装默认的安全管理器。在这个安全管理器中，访问其他线程的授权是严格地基于线程层次的：任何线程都能操纵层次中比它低的其他线程。兄弟线程不能够相互操作，儿子线程不能操作父亲线程。

Java 2还允许通过一系列安全文件来配置默认的安全管理器；这些文件通常包括 *${JAVAHOME}/lib/security/java.policy* 和 *${HOME}/.java.policy*。应用程序所使用的策略文件包含代码对应的URL与从那些URL获取的代码应被授予的权限之间的映射关系。因此，从某个特定URL加载的代码可能被授予如下权限：

```
permission java.security.AllPermission
```

或者是如下的权限：

```
permission java.security.RuntimePermission "thread"
```

被授予这些权限之一的代码可以访问虚拟机上的所有其他线程。

而且，Java 2中，Thread类的stop()方法现在进行一项额外的安全检查。为了能够对任何线程调用stop()方法，必须对给定的URL授予下面的权限：

```
permission java.lang.RuntimePermission "stopThread"
```

默认情况下，所有的代码都被授予全部的权限。但是，一个终端用户或者系统管理员也有可能改变策略文件，以使 stop()方法不能被任意调用。

基于Java 2的浏览器

截止到本书写作的时候，还没有基于Java 2的浏览器可用，因此它们将会采用何种安全策略并不清楚。不过，Java 2 appletviewer 和 1.1 appletviewer 遵循相同的策略。这种策略也有可能会通过策略文件进行辅助设置，以使得从特定 URL 加载的代码能获得访问虚拟机上任意线程的权限。

总结

下面是本章介绍的 ThreadGroup 类的方法列表：

ThreadGroup(String name)

用给定名称创建线程组。

ThreadGroup(ThreadGroup parent, String name)

用给定名称创建给定父亲的子线程组。

void suspend()(Java 2 中不推荐使用)

挂起所有源自此线程组的线程。

void resume()(Java 2 中不推荐使用)

重新开始所有源自此线程组的线程。

void stop()(Java 2 中不推荐使用)

终止所有源自此线程组的线程。

void destroy()

清理线程组并把它从线程组层次关系中删除。

void interrupt()(只用于Java 2 及以上版本)

中断所有源自此线程组的线程。

ThreadGroup getParent()

返回线程组的父线程组引用。

boolean parentOf(ThreadGroup g)

如果组 g 是此线程组的祖先，则返回 true。

int enumerate(Thread list[])

把此线程组及其子孙线程组中所有线程的引用填入 list 数组。

int enumerate(Thread list[], boolean recurse)

把此线程组中所有线程的引用填入 list 数组；如果 recurse 为 true，则此线程组及其子孙线程组中所有线程的引用都被填入 list 数组。

int activeCount()

返回此线程组及其子孙线程组中活动线程的数目。

int enumerate(ThreadGroup list[])

检索给定线程组的所有子孙线程组的引用。此方法对线程组层次递归操作。

int enumerate(ThreadGroup list[], boolean recurse)

检索给定线程组的所有直接子孙的线程组引用；如果 recurse 为 true，则返回线程组所有子孙的线程组引用。

int activeGroupCount()

返回给定线程组（任何层次上）的线程组子孙的数目。

void setMaxPriority(int priority)

设置线程组的最大优先级。

int getMaxPriority()

得到线程组的最大优先级。

void setDaemon(boolean on)

改变线程组的守护状态。

boolean isDaemon()

如果线程组是守护线程组，则返回 true。

boolean isDestroyed()(只用于Java 1.1及以上版本)

返回一个指示线程组是否已被销毁的标志。

String getName()

返回线程组的名称。

void list()

把当前线程组的所有线程列表发送到标准输出。

boolean allowThreadSuspension(boolean b)(只用于Java 1.1)

设置线程组的vmAllowSuspension标志为b，并返回原先的值。如果虚拟机在低内存情况下运行，一些虚拟机实现会试图挂起vmAllowSuspension为true的线程组中的线程以获取更多内存。

void uncaughtException(Thread t, Throwable e)

这个方法在线程因为未捕获的异常而退出时被调用；它的默认行为是向System.err输出线程的堆栈轨迹。

另外，我们介绍了Thread类的一些新方法：

Thread(ThreadGroup group, String name)

构造一个具有给定名称且属于给定线程组的线程。

Thread(ThreadGroup group, Runnable target)

构造一个运行给定目标对象且属于给定线程组的新的线程。

Thread(ThreadGroup group, Runnable target, String name)

构造一个具有给定名称、运行给定目标对象且属于给定线程组的线程。

ThreadGroup getThreadGroup()

返回线程的ThreadGroup引用。

最后，我们介绍了SecurityManager类的一些操作线程的方法：

void checkAccess(Thread t)

检查当前线程是否可以修改线程t的状态。

void checkAccess(ThreadGroup tg)

检查当前线程组是否可以修改线程组 tg 的状态。

在本章中，我们讲述了Java线程机制的最后一部分内容：如何把线程聚合在一起以及如何操作组中的全部线程。ThreadGroup类形成的线程层次关系是Java线程机制中安全策略的基础。

和最后几章的其他主题一样，大多数程序都不会用到ThreadGroup类；它只是一个方便你对线程组进行额外控制的专用类。它也是帮助你理解如何使用Java线程的最后一个专用类。虽然我们在附录中又介绍了一些让你开阔眼界的主题，但本书中所讨论的内容已经足以使你用Java写出有效而复杂的（如果有必要的话）多线程程序。

附录一

其他主题

到目前为止，我们已经讨论了线程系统的多个方面。这些考察是基于在实际程序开发中常会遇到的不同例子和问题的。还有一些相当晦涩的问题没有被讨论；它们就是本附录要讨论的内容。

线程堆栈信息

Thread 类用下面的方法向程序员提供线程堆栈的信息：

int countStackFrames() （*在 Java 2 中不推荐使用*）

返回指定线程的堆栈帧数目。要让这个方法工作，指定线程必须被挂起。该方法是Thread类的方法，并不把本地方法的帧计算在内。由于线程必须被挂起，所以，无法直接得到当前线程的堆栈帧计数。

static void dumpStack()

把当前线程的堆栈轨迹输出到 System.err。它是Thread类的静态方法，可以通过 Thread 类来直接使用。使用它，只能得到当前运行线程的堆栈轨迹。

有意思的是，我们可能会认为：使用上面两个方法，既可以得到堆栈帧的数目，又可以把这些帧实际输出出来。然而，这两个方法不能一起工作。因为只有挂起

了线程才能够计算堆栈帧的数目，这样就无法计算当前线程的帧数。而dumpStack()方法只能够输出当前线程的堆栈信息。

dumpStack()方法输出的信息和Throwable类的printStackTrace()方法提供的信息相同。dumpStack()方法实际上只是个方便途径；它实例化了一个异常对象，并调用printStaceTrace()方法。

通用线程信息

为了输出线程或线程组的信息，需要用到下面的方法：

String toString()

返回描述Thread对象的字符串。toString()本来是Object类的方法，被Thread类覆盖以提供线程名字、线程优先级和线程所属线程组的名字。

String toString()

返回描述ThreadGroup对象的字符串。toString()本来是Object类的方法，被ThreadGroup类覆盖以提供线程组名字和线程组的最大优先级。

toString()方法被线程类覆盖以提供从对象到字符串的转换。因此，代码：

```
Thread t = new TimerThread(this, 500);
System.out.println(t);
```

产生如下输出：

```
Thread[TimerThread-500,6,group applet-TimerApplet]
```

void list()

输出线程组继承关系的当前布局，从list()方法所在的线程组开始。list()是ThreadGroup类的方法，只是简单地把信息输出到System.out。这个方法对线程组进行递归操作。

list()方法输出的信息是toString()方法返回的信息，下面是一个applet中的list()示例：

```
java.lang.ThreadGroup[name=system,maxpri=10]
    Thread[clock handler,11,system]
    Thread[Idle thread,0,system]
    Thread[Async Garbage Collector,1,system]
    Thread[Finalizer thread,1,system]
    java.lang.ThreadGroup[name=main,maxpri=10]
        Thread[main,5,main]
        Thread[AWT-Input,5,main]
        Thread[AWT-Motif,5,main]
        Thread[Screen Updater,4,main]
        AppletThreadGroup[name=group applet-Ticker,maxpri=6]
            Thread[thread applet-Ticker,6,group applet-Ticker]
            Thread[SUNW stock reader,5,group applet-Ticker]
            Thread[APPL stock reader,5,group applet-Ticker]
            Thread[NINI stock reader,5,group applet-Ticker]
            Thread[JRA stock reader,5,group applet-Ticker]
            Thread[ticker timer thread,4,group applet-Ticker]
```

默认异常处理程序

我们以前在考察start()方法的时候，简单地说“start()方法间接调用了run()方法”，现在让我们看看在此过程中究竟发生了什么。start()方法并不启动另一个控制线程，run()方法也不是这个新线程的“main”例程。有其他一些必须首先考虑的簿记细节。线程在run()方法可以执行之前，必须在Java虚拟机上设置。这个过程如图A-1所示。

在线程结束前，所有没被捕获的异常都是用run()方法外面的代码处理的。我们现在就来看看这个异常处理。

为什么这个处理会引起我们的兴趣呢？因为默认的异常处理程序是一个Java方法；它可以被覆盖。也就是说：我们可以为应用程序写一个新的默认异常处理程序。这个方法如下所示：

void uncaughtException(Thread t, Throwable o)

默认的异常处理程序方法被作为一个最终处理者调用来处理线程在run()方法中没有捕获的异常。它是ThreadGroup类的方法。

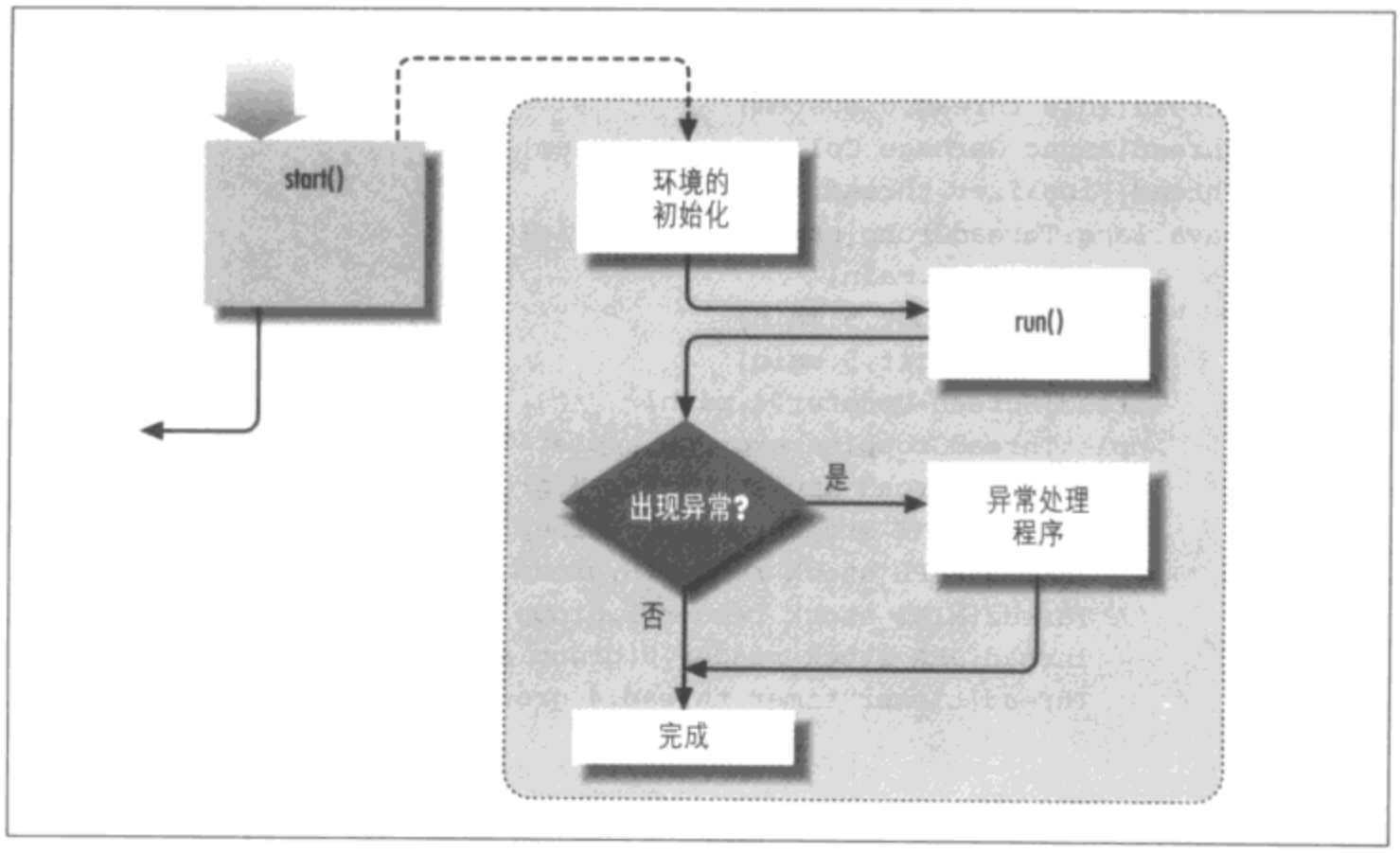

图 A-1: 主线程的流程图

默认异常处理程序是ThreadGroup类的方法。它只在异常从run()方法抛出时被调用。从技术上讲，线程在run()方法返回的时候就结束了，即使异常处理程序仍在此线程中运行。

那么，默认的异常处理程序做了些什么呢？实际上，它什么也没做。默认异常处理程序要完成的惟一工作就是输出Throwable对象记录的堆栈轨迹。被记录轨迹的线程就是首先抛出这个对象的线程（惟一的例外就是当Throwable对象是ThrowDeath对象的时候，这时，就什么都不做。我们将在后面讨论这种情况）

让我们回到第三章中的银行例子。我们知道在我们的ATM系统中，任何异常都必须被捕获，因此我们必须处理每一个异常。但有些问题，比如ATM缺钱了，可能会在算法中的多个地方碰到。用默认异常处理程序处理诸如此类的问题可能是最好的选择。

下面看看一种可能的默认异常处理程序实现：

```
public class ATMOutOfMoneyException extends RuntimeException {
    public ATMOutOfMoneyException() {
```

```
            super();
        }

        public ATMOutOfMoneyException(String s) {
            super(s);
        }
    }

    public class ATMThreadGroup extends ThreadGroup {
        public ATMThreadGroup(String name) {
            super(name);
        }

        public void uncaughtException(Thread t, Throwable e) {
            if (e instanceof ATMOutOfMoneyException) {
                AlertAdminstrator(e);
            } else {
                super.uncaughtException(t, e);
            }
        }
    }
```

可以通过覆盖uncaughtException()方法来实现默认异常处理程序。这样做需要子类化ThreadGroup类，并且实例化一个实例，还要创建所有的线程使它们属于这个实例。我们向方法传递的参数除了被抛出的实际对象，还有抛出对象的Thread类实例。在银行例子中，我们只关心缺钱的情况。抛出的其他各种对象都被传递给原始的默认处理程序。

ThreadDeath类

ThreadDeath类是Throwable类的一种，用来终止线程。它扩展了Error类，因而不应被程序捕获。从理论上说，如果一个Throwable对象不是Exception类对象，就没有理由去捕捉和处理它。这一点通常也适用于ThreadDeath类。

实际上抛出一个对象是怎样终止线程的呢？我们前面已经提到，线程在run()方法完成时要执行一些清理代码。显然，有两种方法让run()方法结束：简单地返回来自行结束，或者抛出或者没能捕捉某个异常（包括Error和Throwable对象）。

在默认情况下，如果run()抛出异常，线程打印一个错误信息以及异常的堆栈轨迹。只是，ThreadDeath对象是一个特例。如果run()方法抛出了ThreadDeath对象，uncaughtException()方法只是简单返回，并不做其他事情。

ThreadDeath对象通常只是和stop()方法一起使用。对某个线程调用stop()方法时，目标线程会产生并抛出一个ThreadDeath对象。由于stop()方法已被废用，这种方法也就没什么用了。

有没有可能捕捉到ThreadDeath对象呢？虽然捕捉任何Throwable对象都是可能的；但我们并不建议使用这种方法去阻止线程的死亡。毕竟，如果我们不想让线程死亡的话，又为什么要去调用stop()方法呢？而且还可能会影响到希望目标线程终止的其他线程。调用目标线程的stop()方法的线程可能跟着会试图连接目标线程；如果把ThreadDeath捕捉了，连接就永远不会完成。

这种方法的一个可能用途是当线程被停止时进行清理处理。在这种情况下，我们捕捉ThreadDeath对象，执行清理代码，然后重新抛出对象。即便如此，在这种情况也很难认为捕捉ThreadDeath对象是理所当然的；我们可以通过使用finally子句完成同样的事情。不过，finally子句总会被执行，而你可能希望代码只在线程被停止的时候执行。

有趣的是，ThreadDeath类恰恰是stop()方法遭到抨击的首要原因：如果异常在同步方法或者同步块的中间被抛出，线程会立即从方法中返回，(可能)使得对象的临界数据处于不一致的状态。你可能会明智地捕获ThreadDeath异常，并且正确地清理你的代码使得stop()方法更加安全，但这只能保护你自己的代码，而不能保护虚拟机的临界区代码或是Java API本身的代码。

尽管如此，ThreadDeath类在一种受限制的场合作为stop()方法的替代品还是有用的。比如，线程遇到错误，希望终止自己，但错误并没有严重到它希望让用户看到的地步。做这件事的一个途径是让线程简单地从run()方法返回，但让线程能够放开所有方法是比较困难的。第二种方法是让线程对自身调用stop()方法。最后的方法就是让线程抛出ThreadDeath错误。这样就可以放开线程的堆栈，使得线程退出自己的run()方法，但由于ThreadDeath错误是虚拟机在特定方式下

处理的，用户并不知道线程已经退出了：因为并没有堆栈轨迹被打印到Java控制台。

即便如此，一个想终止自身的线程也不能莽莽撞撞地抛出ThreadDeath对象：抛出时，线程必须在它确认不会使任何数据处于可能导致的不一致状态（比如，它当前没有持有任何锁）。如果你很细心地编制线程，而且确信线程使所有数据都一致，那么抛出ThreadDeath对象来立即结束线程是安全的。这实际上和自己调用stop()方法是一个道理：惟一的差别就是编译器在你调用stop()方法的时候会发出警告（即使是在确信安全的情况下线程对自己调用该方法），而抛出ThreadDeath对象的时候，编译器就不会如此。不管怎样，你必须谨慎小心，在绝对安全的情况下才这么做。

继承ThreadDeath类

ThreadDeath对象和一个新的stop()方法一起使用：

void stop(Throwable o)(在Java 2中不推荐使用)

终止一个已经运行的线程。通过抛出指定的对象，可以终止线程。

该重载的stop()方法使用同一个参数表，以便开发者能用任何Throwable对象放开堆栈。然而，直到目前，都没有理由用ThreadDeath之外的任何对象停止线程。但我们现在可以覆盖默认异常处理程序；如果我们希望线程由于特定原因死亡，而且可以处理特定原因，我们可以像下面这样生成一个新的Throwable类型和处理程序：

```
public class ATMThreadDeath extends ThreadDeath {
    public int reason;
    public ATMThreadDeath(int reason) {
        this.reason = reason;
    }
}

public class ATMThreadGroup extends ThreadGroup {
    public ATMThreadGroup(String name) {
        super(name);
```

```
        }

        public void uncaughtException(Thread t, Throwable e) {
            if (e instanceof ATMThreadDeath) {
                HandleSpecialExit(e);
            }
            super.uncaughtException(t, e);
        }
    }
```

假定有需要注意的特定的退出处理情况，我们可以创建包含死亡原因的ThreadDeath类的新版本。有了这个新版的ThreadDeath类，我们就可以创建专门的处理程序去处理退出条件。当然，我们现在必须使用其他的stop()方法去发送我们的ATMThreadDeath对象：

```
runner.stop(new ATMThreadDeath(3));
```

我们可以使用stop()方法向其他线程递送通用异常吗？这样其实是可以做到的，不过，我们并不建议这么做。有一些原因反对这样传递参数。根据异常和stop()方法被调用的时机，我们可能会抛出一些违反了throws关键字语义的异常。编译器需要你处理一些它知道会被抛出的异常，但在这种情况下，编译器并不知道你要其他线程抛出的通用异常是什么。如果你执行下面的代码：

```
runner.stop(new IOException());
```

而runner线程可能正执行那些不准备处理IOException的代码。这就会引起混淆。

我们还可以列出更多反对使用这种方法的原因，但这也无法阻止某些开发者使用这种技术作为信号传递系统（注1）。我们只能简单说一句，stop()并不是被设计出来作为信号传递系统的，这么使用可能会产生一些不可预料或与平台有关的结果。

注1：　或者使用异常系统作为回调机制。

关于线程销毁的更多讨论

通过调用stop()方法和退出run()方法的异常机制，我们使得run()方法提早退出，从而结束线程。我们还可以使用destroy()方法，杀死线程，它反过来也会终止run()方法的执行。差别在于run()方法退出的方式：前者允许run()方法终止，从而杀死了线程。而后者杀死了线程，从而终止了run()方法。

通过允许run()方法结束，线程的堆栈也可以放开。这意味着finally子句被允许在堆栈放开的时候执行。这使得线程结束时允许一个更好的状态存在；它还允许在堆栈放开的时候释放同步锁。因为这些好处，线程通常被允许放开而不是仅仅终止。当然，问题在于，由于线程死亡异常可能在任何时候被抛出，那儿可能并没有finally子句去执行，这又把我们带到了要求废止stop()方法的问题上来。

我们现在考察destoy()方法来结束我们的讨论，它允许在不放开堆栈的情况下销毁线程。这种方法可能会作为最后一手来使用：

void destroy()(没有实现)

立刻销毁线程。Thread类的这个方法不执行任何清理代码，在此方法前锁住的任何锁仍然保持锁住状态。

为什么会在线程结束后不进行清理呢？实际上，应该没有什么在线程结束后不想进行清理的情况。然而，却可能存在着清理代码无法工作的情况。比如，使用等待和通知机制，由于难以获得锁可能无法立刻放开堆栈：被停止的线程如果恰好在执行wait()方法，那它可能需要等候一些时间才能结束。如果在试图重新获得锁的时候，它死锁了，那么线程永远也不会结束。所以，对放开的等待期是无法被接受的。

然而，我们可以把这种情况看成程序的bug，通过修改代码去解决，而不是留下可能不释放的锁。在目前的情况下，实际上并没有什么问题：因为destroy()方法在参考JDK中，并没有真正实现，只会简单抛出NoSuchMethodError。

关键字 volatile

我们在第三章提到过，设置变量值的操作（除了long和double类型的变量）是原子操作。也就是说，对读/写变量值的简单访问没有必要进行同步。

然而，Java内存模型比这种说法要复杂。线程可以在本地内存（比如，机器寄存器）中保存变量，这时候，如果一个线程改变变量值，那么另外的线程可能看不到值的变化。这种情况在用变量（比如我们用来终止线程的shoudRun变量）控制的循环中尤其明显：循环线程可能已经把变量值装到寄存器中了，而当另一个线程把变量设置为假时，它可能没有看到。

处理这种情况有多种方法。可以对包含控制变量的对象进行同步，或者更进一步地提供一个对控制变量的存取器方法（就像我们在BusyFlag类中对busyflag变量所做的一样）。或者，你可以把变量标记为volatile，这意味着每次用到这个变量的时候，都必须从主存读取。

在VM 1.2及其之前的各个VM版本中，Java内存模型的实际实现总是从主存读取变量，从而使得对这一点的讨论也没有多大意义。但随着VM越来越成熟，Java会引入新的内存模型和优化方法，注意到这一点也会日渐重要。

附录二

异常和错误

迄今为止，我们在讨论线程及其相关类时很少提及出错情况。原因之一是线程系统不依赖于外部硬件，从而缺少实际的错误情形。处理磁盘和网络的类必须处理那些硬件失败可能引起的错误情况。数据库和窗口系统也需要一个错误系统使程序员可以更好地控制应用、数据结构和用户的交互。

那么，对线程来说，哪些工作是必要的呢？线程是一种处理器资源。启动另一个线程，只需要简单的设立允许处理器运行代码及配置处理器使其在不同线程间切换的数据结构。就像我们在第六章讨论的，线程可能涉及到操作系统；也可能涉及到多个处理器。但无论在哪种情况下，它所可能涉及到的硬件只会是处理器和可能的附加存储。同步系统也只会涉及内存。牵涉的硬件很少，错误自然也很少发生。当然，我们会遇到一些处理器和线程错误，但这些错误通常影响整个虚拟机，而非个体线程。

现在，我们惟一需要考虑的错误就是程序员的错误。他们可能意外地错误配置了线程或者错误使用了线程或同步机制。

如何报告错误情况呢？就像Java系统提供的其他类一样，线程使用抛出异常和错误的概念。下面我们考察线程系统所抛出的一些异常和错误。

InterruptedException

InterruptedException可能是本书中最常见的异常。它表明方法比预期时间提前返回。尽管在大多数的例子中，我们选择捕捉或者忽视这些异常，我们其实不必这样去做。因为有可能依赖程序处理这种异常（解决的方案可能像重新调用方法一样简单）。

下边来看看在本书中遇到过的中断异常:

join()方法

线程类提供join()方法使线程可以等待另外一个线程结束或被终止（参见第二章）。抛出这种异常意味着其他线程可能没有结束。join()方法被用另两个方法签名重载，使得程序可以指出一个超时时间（timeout）。这些方法抛出异常就表明其他线程没有结束，等候时间也没有被满足。

sleep()方法

线程类提供sleep()方法，允许线程等候指定的时间段（参见第二章）。抛出这种异常意味着sleep()方法还没睡眠到指定时间。

wait()方法

Object类提供wait()方法，允许线程等候某种通知情况（参见第四章）。抛出异常意味着wait()方法还没有接到通知。Wait()方法也用两个其他的方法签名覆盖以使程序指定超时时间。异常被这些方法抛出意味着既没收到通知，超时条件也没满足。

InterruptedException是通过Thread类的interrupt()方法产生的。

InterruptedIOException

不同I/O类的多个方法会抛出InterruptedIOException回应interrupt()方法:如果目标线程因为一个I/O操作被阻塞，InterruptedIOException就会抛出。在绿色线程实现上，这一点实现得并不完全: 一些I/O方法可以被中断，另一些则不可以。这个特征在Windows系统上则是完全没有实现。在Solaris本地线程虚拟

机上，它实现得有些不一致：在Java 1.1中，一些操作抛出标准异常（比如SocketException），而在Java 2中，它们抛出InterruptedIOException。

将来，这些实现将会一致，但目前还不清楚发展趋势如何；也有可能这种异常会被忽视掉。其间，需要中断I/O的程序员应该关闭I/O操作所在的流，被中断的I/O不能再被认为是可重启的，即使是在支持这么做的平台上。

NoSuchMethodError

Thread类被设计时，某些方法没有得到立即的支持。为了避免改变Thread类的接口，这些方法的大多数被简单写作抛出NoSuchMethodError。随着更多的功能的加入，抛出这种错误的方法越来越少。截至这本书写作的时候，抛出这种错误对象的方法只有Thread类的`destroy()`方法了（参见附录一）。

异常或错误

异常和错误之间有什么区别呢？就虚拟机来说，两者并没有什么区别：它们都是被抛出来报告某种情况的对象。可以像捕获异常对象一样捕获错误对象。然而，在实践上，两种类型的用法存在不同。

错误情况指的是Java虚拟机的错误。一般来说，它们报出的问题是不能被程序解决的。引发原因可能是内存越界、堆栈溢出或者装载和解析程序中的类时产生的问题。它们被分离开的原因是为了允许捕获所有一般异常。程序可能通过捕获异常对象去捕获所有的异常，但程序没理由捕获错误对象。

RuntimeException

RuntimeException并不直接被线程类的任何方法抛出，它只是一个基类，用来指定一组特定的异常。运行时异常十分的基本，以致于要检测每个可能抛出的运行时异常非常繁琐（另一个不检测的原因在于这些异常通常是程序的bug）。和其他异常不同，编译器不要求你处理RuntimeException。

下列异常都是运行时异常。

IllegalThreadStateException

当线程处在无法实现请求的状态时，会抛出IllegalThreadStateException。这是由程序的非法请求引起的，通常意味着程序中的bug。下列是线程系统中可能抛出IllegalThreadStateException的情形：

start()方法

Thread类提供start()方法来启动新线程（参见第二章）。就像我们前面所说的，线程只能启动一次。如果程序对已运行的线程调用start()方法，就会抛出IllegalThreadStateException。

setDaemon()方法

Thread类提供setDaemon()方法用来指定线程为守护线程（参见第六章）。线程的守护状态必须在线程被启动前设置。如果对已运行的线程调用setDaemon()方法，将会抛出IllegalThreadStateException。

countStackFrame()方法

Thread类提供countStackFrame()方法来确定线程正在调用堆栈中多深处（参见附录一）。要生成这个计数，线程必须被挂起。如果调用这个方法时，线程没有挂起，将会引发IllegalThreadStateException。

destroy()方法

ThreadGroup类提供destroy()方法以允许线程组被销毁（参见第十章）。一个ThreadGroup实例只在它不包含任何线程而且不包含任何包含线程的线程组时才可以被销毁。如果调用destroy()方法销毁包含线程的线程组，或者调用它去销毁已经销毁了的线程组，将会引发IllegalThreadStateException。

线程构造函数

Thread类包含某些构造函数以允许线程被放入特定的线程组（参见第十章）。不能向这些构造函数传递已被销毁的线程组；否则，会引发IllegalThreadStateException。

IllegalArgumentException

可能存在这样的情况：调用方法时所给的参数不正确。当出现这种情况时，会引发IllegalArgumentException。和Thread类有关的方法只有一个会抛出这种异常：

SetPriority()方法

Thread类提供setPriority()方法控制线程的优先级（参见第六章）。优先级必须取在系统最大优先级和最小优先级之间。如果请求的优先级不在这个范围，会引发IllegalArgumentException（setPriority()方法还可能抛出安全异常；参见本附录后面的“SecurityException”）。

IllegalThreadStateException实际上是IllegalArgumentException类的子类；如果试图捕捉IllegalArgumentException类型的对象，同时还会捕捉IllegalThread-StateException类型的对象。

IllegalMonitorStateException

当尝试对等待监视器进行操作而监视器当时的状态不允许进行此操作时，Thread系统会抛出IllegalMonitorStateException。目前，涉及到这种异常的惟一操作是等待和通知机制；抓取和释放锁本身不是方法调用，因此不会抛出异常。

wait()方法

Object类提供wait()方法，供线程等待通知条件（参见第四章）。wait()方法必须在持有对对象的同步锁时才能被调用。wait()方法同样被用两个不同的方法签名重载，以允许程序指定超时时间。这几个方法如果在没有占有同步锁的时候被调用，IllegalMonitorStateException会被抛出。

notify()和notifyAll()方法

Object类提供notify()方法，允许线程传送通知信号给任何等待的线程（参见第四章）。notify()方法必须在持有对对象的同步锁时才能被调用。Object类同样提供了notifyAll()方法，该方法用来唤醒所有等待的线程。如果在没有获得同步锁的情况下调用这些方法，会抛出IllegalMonitorState-Exception异常。

NullPointerException

Thread 类在下列情况下抛出这种异常:

stop()方法

Thread类提供一个版本的stop()方法去允许用户指定用来停止线程的对象(参见附录一)。一般情况下,程序不使用这些方法;然而,如果程序使用了这个方法,而且传送空对象去停止线程,就会抛出 NullPointerException。

ThreadGroup 构造函数

ThreadGroup类提供一个版本的构造函数以允许应用指定其父线程组(参见第十章)。如果为父线程组指定的是 null,会抛出 NullPointerException。

此外,Java 虚拟机本身可能会在执行线程类内部的代码时抛出 NullPointer-Exception。

SecurityException

Thread 类和 ThreadGroup 的大多数方法可能会抛出 SecurityException。抛出 SecurityException 的情况有以下几种:

checkAccess()方法

Thread 类提供了 checkAccess()方法,它简单地调用安全管理器来确定线程是否可以被当前线程组访问(参见第十章)。如果访问不被允许,会抛出一个 SecurityException 异常。表 10-1 给出了完整的调用 checkAccess()方法的方法列表。

checkAccess()方法

ThreadGroup类提供了checkAccess()方法,它简单地调用安全管理器来确定线程组是否可以被当前线程组访问(参见第十章)。如果访问不被允许,就会抛出一个 SecurityException 异常。表 10-1 给出了完整的调用 checkAccess()方法的方法列表。

setPriority()方法

Thread类提供setPriority()方法，设置线程的调度优先级。被请求的优先级必须小于线程所属线程组的最大优先级。否则，会抛出SecurityException（参见第十章）。

stop()方法

在Java 2及以上版本中，如果在代码没被授予stopThread许可的情况下调用Thread类的stop()方法，stop()方法将会抛出安全异常。

任意异常

下面的方法可能会抛出任意运行时异常：

run()方法

Thread类的run()方法执行用户指定的代码，因此可能抛出任何用户代码没有捕捉的运行时异常。run()方法抛出的异常是用附录一中讲述的方法捕获的。

词汇表

API（Application Programming Interface）
应用编程接口

blocking
阻塞

dead lock
死锁

interpreter
解释器

lock
锁

mutex lock
互斥锁

mutually exclusive lock
互斥独占锁

polling
轮询

race condition
竞态条件

thread
线程

thread join
线程连接

token
令牌

virtual machine
虚拟机