第 14 章 多线程

利用对象，可将一个程序分割成相互独立的区域。我们通常也需要将一个程序转换成多个独立运行的子任务。

象这样的每个子任务都叫作一个“线程”（Thread）。编写程序时，可将每个线程都想象成独立运行，而且都有自己的专用 CPU。一些基础机制实际会为我们自动分割 CPU 的时间。我们通常不必关心这些细节问题，所以多线程的代码编写是相当简便的。

这时理解一些定义对以后的学习狠有帮助。“进程”是指一种“自包容”的运行程序，有自己的地址空间。“多任务”操作系统能同时运行多个进程（程序）——但实际是由于 CPU 分时机制的作用，使每个进程都能循环获得自己的 CPU 时间片。但由于轮换速度非常快，使得所有程序好象是在“同时”运行一样。“线程”是进程内部单一的一个顺序控制流。因此，一个进程可能容纳了多个同时执行的线程。

多线程的应用范围很广。但在一般情况下，程序的一些部分同特定的事件或资源联系在一起，同时又不想为它而暂停程序其他部分的执行。这样一来，就可考虑创建一个线程，令其与那个事件或资源关联到一起，并让它独立于主程序运行。一个很好的例子便是“Quit”或“退出”按钮——我们并不希望在程序的每一部分代码中都轮询这个按钮，同时又希望该按钮能及时地作出响应（使程序看起来似乎经常都在轮询它）。事实上，多线程最主要的一个用途就是构建一个“反应灵敏”的用户界面。

14.1 反应灵敏的用户界面

作为我们的起点，请思考一个需要执行某些 CPU 密集型计算的程序。由于

CPU“全心全意”为那些计算服务，所以对用户的输入十分迟钝，几乎没有什么反应。在这里，我们用一个合成的 applet/application（程序片／应用程序）来简单显示出一个计数器的结果：

752-753 页程序

在这个程序中，AWT 和程序片代码都应是大家熟悉的，第 13 章对此已有很详细的交待。go()方法正是程序全心全意服务的对待：将当前的 count（计数）值置入 TextField（文本字段）t，然后使 count 增值。

go()内的部分无限循环是调用 sleep()。sleep()必须同一个 Thread（线程）对象关联到一起，而且似乎每个应用程序都有部分线程同它关联（事实上，Java 本身就是建立在线程基础上的，肯定有一些线程会伴随我们写的应用一起运行）。所以无论我们是否明确使用了线程，都可利用 Thread.currentThread()产生由程序使用的当前线程，然后为那个线程调用 sleep()。注意，Thread.currentThread()是 Thread 类的一个静态方法。

注意 sleep()可能“掷”出一个 InterruptException（中断违例）——尽管产生这样的违例被认为是中止线程的一种“恶意”手段，而且应该尽可能地杜绝这一做法。再次提醒大家，违例是为异常情况而产生的，而不是为了正常的控制流。在这里包含了对一个“睡眠”线程的中断，以支持未来的一种语言特性。

一旦按下 start 按钮，就会调用 go()。研究一下 go()，你可能会很自然地（就象我一样）认为它该支持多线程，因为它会进入“睡眠”状态。也就是说，尽管方法本身“睡着”了，CPU 仍然应该忙于监视其他按钮“按下”事件。但有一个问题，那就是 go()是永远不会返回的，因为它被设计成一个无限循环。这意味着 actionPerformed() 根本不会返回。由于在第一个按键以后便陷入 actionPerformed()中，所以程序不能再对其他任何事件进行控制（如果想出来，必须以某种方式“杀死”进程——最简便的方式就是在控制台窗口按 Ctrl＋C 键）。

这里最基本的问题是 go()需要继续执行自己的操作，而与此同时，它也需要返回，以便 actionPerformed()能够完成，而且用户界面也能继续响应用户的操作。但对象 go()这样的传统方法来说，它却不能在继续的同时将控制权返回给程序的其他部分。这听起来似乎是一件不可能做到的事情，就象 CPU 必须同时位于两个地方一样，但线程可以解决一切。“线程模型”（以及 Java 中的编程支持）是一种程序编写规范，可在单独一个程序里实现几个操作的同时进行。根据这一机制，CPU 可为每个线程都分配自己的一部分时间。每个线程都“感觉”自己好象拥有整个 CPU，但 CPU 的计算时间实际却是在所有线程间分摊的。

线程机制多少降低了一些计算效率，但无论程序的设计，资源的均衡，还是用户操作的方便性，都从中获得了巨大的利益。综合考虑，这一机制是非常有价值的。当然，如果本来就安装了多块 CPU，那么操作系统能够自行决定为不同的 CPU 分配哪些线程，程序的总体运行速度也会变得更快（所有这些都要求操作系统以及应用程序的支持）。多线程和多任务是充分发挥多处理机系统能力的一种最有效的方式。

14.1.1 从线程继承

为创建一个线程，最简单的方法就是从 Thread 类继承。这个类包含了创建和运行线程所需的一切东西。Thread 最重要的方法是 run()。但为了使用 run()，必

须对其进行过载或者覆盖，使其能充分按自己的吩咐行事。因此，run()属于那些会与程序中的其他线程“并发”或“同时”执行的代码。

下面这个例子可创建任意数量的线程，并通过为每个线程分配一个独一无二的编号（由一个静态变量产生），从而对不同的线程进行跟踪。Thread 的 run()方法在这里得到了覆盖，每通过一次循环，计数就减 1——计数为 0 时则完成循环（此时一旦返回 run()，线程就中止运行）。

755 页程序

run()方法几乎肯定含有某种形式的循环——它们会一直持续到线程不再需要为止。因此，我们必须规定特定的条件，以便中断并退出这个循环（或者在上述的例子中，简单地从 run()返回即可）。run()通常采用一种无限循环的形式。也就是说，通过阻止外部发出对线程的 stop()或者 destroy()调用，它会永远运行下去（直到程序完成）。

在 main()中，可看到创建并运行了大量线程。Thread 包含了一个特殊的方法，叫作 start()，它的作用是对线程进行特殊的初始化，然后调用 run()。所以整个步骤包括：调用构建器来构建对象，然后用 start()配置线程，再调用 run()。如果不调用 start()——如果适当的话，可在构建器那样做——线程便永远不会启动。

下面是该程序某一次运行的输出（注意每次运行都会不同）：

756 页程序

可注意到这个例子中到处都调用了 sleep()，然而输出结果指出每个线程都获得了属于自己的那一部分 CPU 执行时间。从中可以看出，尽管 sleep()依赖一个线程的存在来执行，但却与允许或禁止线程无关。它只不过是另一个不同的方法而已。

亦可看出线程并不是按它们创建时的顺序运行的。事实上，CPU 处理一个现有线程集的顺序是不确定的——除非我们亲自介入，并用 Thread 的 setPriority()方法调整它们的优先级。

main()创建 Thread 对象时，它并未捕获任何一个对象的句柄。普通对象对于垃圾收集来说是一种“公平竞赛”，但线程却并非如此。每个线程都会“注册”自己，所以某处实际存在着对它的一个引用。这样一来，垃圾收集器便只好对它“瞠目以对”了。

14.1.2 针对用户界面的多线程

现在，我们也许能用一个线程解决在 Counter1.java 中出现的问题。采用的一个技巧便是在一个线程的 run()方法中放置“子任务”——亦即位于 go()内的循环。一旦用户按下 Start 按钮，线程就会启动，但马上结束线程的创建。这样一来，尽管线程仍在运行，但程序的主要工作却能得以继续（等候并响应用户界面的事件）。下面是具体的代码：

757-759 页程序

现在，Counter2 变成了一个相当直接的程序，它的唯一任务就是设置并管理

用户界面。但假若用户现在按下 Start 按钮，却不会真正调用一个方法。此时不是创建类的一个线程，而是创建 SeparateSubTask，然后继续 Counter2 事件循环。注意此时会保存 SeparateSubTask 的句柄，以便我们按下 onOff 按钮的时候，能正常地切换位于 SeparateSubTask 内部的 runFlag（运行标志）。随后那个线程便可启动（当它看到标志的时候），然后将自己中止（亦可将 SeparateSubTask 设为一个内部类来达到这一目的）。

SeparateSubTask 类是对 Thread 的一个简单扩展，它带有一个构建器（其中保存了 Counter2 句柄，然后通过调用 start()来运行线程）以及一个 run()——本质上包含了 Counter1.java 的 go()内的代码。由于 SeparateSubTask 知道自己容纳了指向一个 Counter2 的句柄，所以能够在需要的时候介入，并访问 Counter2 的 TestField（文本字段）。

按下 onOff 按钮，几乎立即能得到正确的响应。当然，这个响应其实并不是“立即”发生的，它毕竟和那种由“中断”驱动的系统不同。只有线程拥有 CPU 的执行时间，并注意到标记已发生改变，计数器才会停止。

1. 用内部类改善代码

下面说说题外话，请大家注意一下 SeparateSubTask 和 Counter2 类之间发生的结合行为。SeparateSubTask 同 Counter2“亲密”地结合到了一起——它必须持有指向自己“父”Counter2 对象的一个句柄，以便自己能回调和操纵它。但两个类并不是真的合并为单独一个类（尽管在下一节中，我们会讲到 Java 确实提供了合并它们的方法），因为它们各自做的是不同的事情，而且是在不同的时间创建的。但不管怎样，它们依然紧密地结合到一起（更准确地说，应该叫“联合”），所以使程序代码多少显得有些笨拙。在这种情况下，一个内部类可以显著改善代码的“可读性”和执行效率：

759-761 页程序

这个 SeparateSubTask 名字不会与前例中的 SeparateSubTask 冲突——即使它们都在相同的目录里——因为它已作为一个内部类隐藏起来。大家亦可看到内部类被设为 private（私有）属性，这意味着它的字段和方法都可获得默认的访问权限（run()除外，它必须设为 public，因为它在基础类中是公开的）。除 Counter2i 之外，其他任何方面都不可访问 private 内部类。而且由于两个类紧密结合在一起，所以很容易放宽它们之间的访问限制。在 SeparateSubTask 中，我们可看到 invertFlag()方法已被删去，因为 Counter2i 现在可以直接访问 runFlag。

此外，注意 SeparateSubTask 的构建器已得到了简化——它现在唯一的用外就是启动线程。Counter2i 对象的句柄仍象以前那样得以捕获，但不再是通过人工传递和引用外部对象来达到这一目的，此时的内部类机制可以自动照料它。在 run()中，可看到对 t 的访问是直接进行的，似乎它是 SeparateSubTask 的一个字段。父类中的 t 字段现在可以变成 private，因为 SeparateSubTask 能在未获任何特殊许可的前提下自由地访问它——而且无论如何都该尽可能地把字段变成“私有”属性，以防来自类外的某种力量不慎地改变它们。

无论在什么时候，只要注意到类相互之间结合得比较紧密，就可考虑利用内部类来改善代码的编写与维护。

14.1.3 用主类合并线程

在上面的例子中，我们看到线程类（Thread）与程序的主类（Main）是分隔开的。这样做非常合理，而且易于理解。然而，还有另一种方式也是经常要用到的。尽管它不十分明确，但一般都要更简洁一些（这也解释了它为什么十分流行）。通过将主程序类变成一个线程，这种形式可将主程序类与线程类合并到一起。由于对一个 GUI 程序来说，主程序类必须从 Frame 或 Applet 继承，所以必须用一个接口加入额外的功能。这个接口叫作 Runnable，其中包含了与 Thread 一致的基本方法。事实上，Thread 也实现了 Runnable，它只指出有一个 run()方法。

对合并后的程序／线程来说，它的用法不是十分明确。当我们启动程序时，会创建一个 Runnable（可运行的）对象，但不会自行启动线程。线程的启动必须明确进行。下面这个程序向我们演示了这一点，它再现了 Counter2 的功能：

762-763 页程序 1

现在 run()位于类内，但它在 init()结束以后仍处在“睡眠”状态。若按下启动按钮，线程便会用多少有些暧昧的表达方式创建（若线程尚不存在）：

```
new Thread(Counter3.this);
```

若某样东西有一个 Runnable 接口，实际只是意味着它有一个 run()方法，但不存在与之相关的任何特殊东西——它不具有任何天生的线程处理能力，这与那些从 Thread 继承的类是不同的。所以为了从一个 Runnable 对象产生线程，必须单独创建一个线程，并为其传递 Runnable 对象；可为其使用一个特殊的构建器，并令其采用一个 Runnable 作为自己的参数使用。随后便可为那个线程调用 start()，如下所示：

```
selfThread.start();
```

它的作用是执行常规初始化操作，然后调用 run()。

Runnable 接口最大的一个优点是所有东西都从属于相同的类。若需访问什么东西，只需简单地访问它即可，不需要涉及一个独立的对象。但为这种便利也是要付出代价的——只可为那个特定的对象运行单独一个线程（尽管可创建那种类型的多个对象，或者在不同的类里创建其他对象）。

注意 Runnable 接口本身并不是造成这一限制的罪魁祸首。它是由于 Runnable 与我们的主类合并造成的，因为每个应用只能主类的一个对象。

14.1.4 制作多个线程

现在考虑一下创建多个不同的线程的问题。我们不可用前面的例子来做到这一点，所以必须倒退回去，利用从 Thread 继承的多个独立类来封装 run()。但这是一种更常规的方案，而且更易理解，所以尽管前例揭示了我们经常都能看到的编码样式，但并不推荐在大多数情况下都那样做，因为它只是稍微复杂一些，而且灵活性稍低一些。

下面这个例子用计数器和切换按钮再现了前面的编码样式。但这一次，一个特定计数器的所有信息（按钮和文本字段）都位于它自己的、从 Thread 继承的对象内。Ticker 中的所有字段都具有 private（私有）属性，这意味着 Ticker 的具体实现方案可根据实际情况任意修改，其中包括修改用于获取和显示信息的数据组件的数量及类型。创建好一个 Ticker 对象以后，构建器便请求一个 AWT 容器（Container）的句柄——Ticker 用自己的可视组件填充那个容器。采用这种方式，

以后一旦改变了可视组件，使用 Ticker 的代码便不需要另行修改一道。

764-766 页程序

Ticker 不仅包括了自己的线程处理机制，也提供了控制与显示线程的工具。可按自己的意愿创建任意数量的线程，毋需明确地创建窗口化组件。

在 Counter4 中，有一个名为 s 的 Ticker 对象的数组。为获得最大的灵活性，这个数组的长度是用程序片参数接触 Web 页而初始化的。下面是网页中长度参数大致的样子，它们嵌于对程序片（applet）的描述内容中：

```
<applet code=Counter4 width=600 height=600>
<param name=size value="20">
</applet>
```

其中，param，name 和 value 是所有 Web 页都适用的关键字。name 是指程序中对参数的一种引用称谓，value 可以是任何字串（并不仅仅是解析成一个数字的东西）。

我们注意到对数组 s 长度的判断是在 init()内部完成的，它没有作为 s 的内嵌定义的一部分提供。换言之，不可将下述代码作为类定义的一部分使用（应该位于任何方法的外部）：

```
inst size = Integer.parseInt(getParameter("Size"));
Ticker[] s = new Ticker[size]
```

可把它编译出来，但会在运行期得到一个空指针违例。但若将 getParameter()初始化移入 init()，则可正常工作。程序片框架会进行必要的启动工作，以便在进入 init()前收集好一些参数。

此外，上述代码被同时设置成一个程序片和一个应用（程序）。在它是应用程序的情况下，size 参数可从命令行里提取出来（否则就提供一个默认的值）。

数组的长度建好以后，就可以创建新的 Ticker 对象；作为 Ticker 构建器的一部分，用于每个 Ticker 的按钮和文本字段就会加入程序片。

按下 Start 按钮后，会在整个 Ticker 数组里遍历，并为每个 Ticker 调用 start()。记住，start()会进行必要的线程初始化工作，然后为那个线程调用 run()。

ToggleL 监视器只是简单地切换 Ticker 中的标记，一旦对应线程以后需要修改这个标记，它会作出相应的反应。

这个例子的一个好处是它使我们能够方便地创建由单独子任务构成的大型集合，并以监视它们的行为。在这种情况下，我们会发现随着子任务数量的增多，机器显示出来的数字可能会出现更大的分歧，这是由于为线程提供服务的方式造成的。

亦可试着体验一下 sleep(100)在 Ticker.run()中的重要作用。若删除 sleep()，那么在按下一个切换按钮前，情况仍然会进展良好。按下按钮以后，那个特定的线程就会出现一个失败的 runFlag，而且 run()会深深地陷入一个无限循环——很难在多任务处理期间中止退出。因此，程序对用户操作的反应灵敏度会大幅度降低。

14.1.5 Daemon 线程

“Daemon”线程的作用是在程序的运行期间于后台提供一种“常规”服务，但它并不属于程序的一个基本部分。因此，一旦所有非 Daemon 线程完成，程序

也会中止运行。相反，假若有任何非 Daemon 线程仍在运行（比如还有一个正在运行 main()的线程），则程序的运行不会中止。

通过调用 isDaemon()，可调查一个线程是不是一个 Daemon，而且能用 setDaemon()打开或者关闭一个线程的 Daemon 状态。如果是一个 Daemon 线程，那么它创建的任何线程也会自动具备 Daemon 属性。

下面这个例子演示了 Daemon 线程的用法：

768-769 页程序

Daemon 线程可将自己的 Daemon 标记设置成“真”，然后产生一系列其他线程，而且认为它们也具有 Daemon 属性。随后，它进入一个无限循环，在其中调用 yield()，放弃对其他进程的控制。在这个程序早期的一个版本中，无限循环会使 int 计数器增值，但会使整个程序都好象陷入停顿状态。换用 yield()后，却可使程序充满“活力”，不会使人产生停滞或反应迟钝的感觉。

一旦 main()完成自己的工作，便没有什么能阻止程序中断运行，因为这里运行的只有 Daemon 线程。所以能看到启动所有 Daemon 线程后显示出来的结果，System.in 也进行了相应的设置，使程序中断前能等待一个回车。如果不进行这样的设置，就只能看到创建 Daemon 线程的一部分结果（试试将 readLine()代码换成不同长度的 sleep()调用，看看会有什么表现）。

14.2 共享有限的资源

可将单线程程序想象成一种孤立的实体，它能遍历我们的问题空间，而且一次只能做一件事情。由于只有一个实体，所以永远不必担心会有两个实体同时试图使用相同的资源，就象两个人同时都想停到一个车位，同时都想通过一扇门，甚至同时发话。

进入多线程环境后，它们则再也不是孤立的。可能会有两个甚至更多的线程试图同时同一个有限的资源。必须对这种潜在资源冲突进行预防，否则就可能发生两个线程同时访问一个银行帐号，打印到同一台计算机，以及对同一个值进行调整等等。

14.2.1 资源访问的错误方法

现在考虑换成另一种方式来使用本章频繁见到的计数器。在下面的例子中，每个线程都包含了两个计数器，它们在 run()里增值以及显示。除此以外，我们使用了 Watcher 类的另一个线程。它的作用是监视计数器，检查它们是否保持相等。这表面是一项无意义的行动，因为如果查看代码，就会发现计数器肯定是相同的。但实际情况却不一定如此。下面是程序的第一个版本：

770-773 页程序

和往常一样，每个计数器都包含了自己的显示组件：两个文本字段以及一个标签。根据它们的初始值，可知道计数是相同的。这些组件在 TwoCounter 构建器加入 Container。由于这个线程是通过用户的一个“按下按钮”操作启动的，所以 start()可能被多次调用。但对一个线程来说，对 Thread.start()的多次调用是非法的（会产生违例）。在 started 标记和过载的 start()方法中，大家可看到针对

这一情况采取的防范措施。

在 run()中，count1 和 count2 的增值与显示方式表面上似乎能保持它们完全一致。随后会调用 sleep()；若没有这个调用，程序便会出错，因为那会造成 CPU 难于交换任务。

synchTest()方法采取的似乎是没有意义的行动，它检查 count1 是否等于 count2；如果不等，就把标签设为“Unsynched”（不同步）。但是首先，它调用的是类 Sharing1 的一个静态成员，以便增值和显示一个访问计数器，指出这种检查已成功进行了多少次（这样做的理由会在本例的其他版本中变得非常明显）。

Watcher 类是一个线程，它的作用是为处于活动状态的所有 TwoCounter 对象都调用 synchTest()。其间，它会对 Sharing1 对象中容纳的数组进行遍历。可将 Watcher 想象成它掠过 TwoCounter 对象的肩膀不断地“偷看”。

Sharing1 包含了 TwoCounter 对象的一个数组，它通过 init()进行初始化，并在我们按下“start”按钮后作为线程启动。以后若按下“Observe”（观察）按钮，就会创建一个或者多个观察器，并对毫不设防的 TwoCounter 进行调查。

注意为了让它作为一个程序片在浏览器中运行，Web 页需要包含下面这几行：

774 页上程序

可自行改变宽度、高度以及参数，根据自己的意愿进行试验。若改变了 size 和 observers，程序的行为也会发生变化。我们也注意到，通过从命令行接受参数（或者使用默认值），它被设计成作为一个独立的应用程序运行。

下面才是最让人“不可思议”的。在 TwoCounter.run()中，无限循环只是不断地重复相邻的行：

```
t1.setText(Integer.toString(count1++));
t2.setText(Integer.toString(count2++));
```

（和“睡眠”一样，不过在这里并不重要）。但在程序运行的时候，你会发现 count1 和 count2 被“观察”（用 Watcher 观察）的次数是不相等的！这是由线程的本质造成的——它们可在任何时候挂起（暂停）。所以在上述两行的执行时刻之间，有时会出现执行暂停现象。同时，Watcher 线程也正好跟随着进来，并正好在这个时候进行比较，造成计数器出现不相等的情况。

本例揭示了使用线程时一个非常基本的问题。我们跟无从知道一个线程什么时候运行。想象自己坐在一张桌子前面，桌上放有一把叉子，准备叉起自己的最后一块食物。当叉子要碰到食物时，食物却突然消失了（因为这个线程已被挂起，同时另一个线程进来“偷”走了食物）。这便是我们要解决的问题。

有的时候，我们并不介意一个资源在尝试使用它的时候是否正被访问（食物在另一些盘子里）。但为了让多线程机制能够正常运转，需要采取一些措施来防止两个线程访问相同的资源——至少在关键的时期。

为防止出现这样的冲突，只需在线程使用一个资源时为其加锁即可。访问资源的第一个线程会其加上锁以后，其他线程便不能再使用那个资源，除非被解锁。如果车子的前座是有限的资源，高喊“这是我的!”的孩子会主张把它锁起来。

14.2.2 Java 如何共享资源

对一种特殊的资源——对象中的内存——Java 提供了内建的机制来防止它

们的冲突。由于我们通常将数据元素设为从属于 private（私有）类，然后只通过方法访问那些内存，所以只需将一个特定的方法设为 synchronized（同步的），便可有效地防止冲突。在任何时刻，只可有一个线程调用特定对象的一个 synchronized 方法（尽管那个线程可以调用多个对象的同步方法）。下面列出简单的 synchronized 方法：

synchronized void f() { /* ... */ }

synchronized void g() { /* ... */ }

每个对象都包含了一把锁（也叫作“监视器”），它自动成为对象的一部分（不必为此写任何特殊的代码）。调用任何 synchronized 方法时，对象就会被锁定，不可再调用那个对象的其他任何 synchronized 方法，除非第一个方法完成了自己的工作，并解除锁定。在上面的例子中，如果为一个对象调用 f()，便不能再为同样的对象调用 g()，除非 f()完成并解除锁定。因此，一个特定对象的所有 synchronized 方法都共享着一把锁，而且这把锁能防止多个方法对通用内存同时进行写操作（比如同时有多个线程）。

每个类也有自己的一把锁（作为类的 Class 对象的一部分），所以 synchronized static 方法可在一个类的范围内被相互间锁定起来，防止与 static 数据的接触。

注意如果想保护其他某些资源不被多个线程同时访问，可以强制通过 synchronized 方访问那些资源。

1. 计数器的同步

装备了这个新关键字后，我们能够采取的方案就更灵活了：可以只为 TwoCounter 中的方法简单地使用 synchronized 关键字。下面这个例子是对前例的改版，其中加入了新的关键字：

775-778 页程序

我们注意到无论 run()还是 synchTest()都是“同步的”。如果只同步其中的一个方法，那么另一个就可以自由忽视对象的锁定，并可无碍地调用。所以必须记住一个重要的规则：对于访问某个关键共享资源的所有方法，都必须把它们设为 synchronized，否则就不能正常地工作。

现在又遇到了一个新问题。Watcher2 永远都不能看到正在进行的事情，因为整个 run()方法已设为“同步”。而且由于肯定要为每个对象运行 run()，所以锁永远不能打开，而 synchTest()永远不会得到调用。之所以能看到这一结果，是因为 accessCount 根本没有变化。

为解决这个问题，我们能采取的一个办法是只将 run()中的一部分代码隔离出来。想用这个办法隔离出来的那部分代码叫作“关键区域”，而且要用不同的方式来使用 synchronized 关键字，以设置一个关键区域。Java 通过“同步块”提供对关键区域的支持；这一次，我们用 synchronized 关键字指出对象的锁用于对其中封闭的代码进行同步。如下所示：

779 页中程序

在能进入同步块之前，必须在 synchObject 上取得锁。如果已有其他线程取得了这把锁，块便不能进入，必须等候那把锁被释放。

可从整个 run()中删除 synchronized 关键字，换成用一个同步块包围两个关键行，从而完成对 Sharing2 例子的修改。但什么对象应作为锁来使用呢？那个对象已由 synchTest()标记出来了——也就是当前对象（this）！所以修改过的 run()方法象下面这个样子：

779 页下程序

这是必须对 Sharing2.java 作出的唯一修改，我们会看到尽管两个计数器永远不会脱离同步（取决于允许 Watcher 什么时候检查它们），但在 run()执行期间，仍然向 Watcher 提供了足够的访问权限。

当然，所有同步都取决于程序员是否勤奋：要访问共享资源的每一部分代码都必须封装到一个适当的同步块里。

2. 同步的效率

由于要为同样的数据编写两个方法，所以无论如何都不会给人留下效率很高的印象。看来似乎更好的一种做法是将所有方法都设为自动同步，并完全消除 synchronized 关键字（当然，含有 synchronized run()的例子显示出这样做是很不通的）。但它也揭示出获取一把锁并非一种“廉价”方案——为一次方法调用付出的代价（进入和退出方法，不执行方法主体）至少要累加到四倍，而且根据我们的具体现方案，这一代价还有可能变得更高。所以假如已知一个方法不会造成冲突，最明智的做法便是撤消其中的 synchronized 关键字。

14.2.3 回顾 Java Beans

我们现在已理解了同步，接着可换从另一个角度来考察 Java Beans。无论什么时候创建了一个 Bean，就必须假定它要在一个多线程的环境中运行。这意味着：

(1) 只要可行，Bean 的所有公共方法都应同步。当然，这也带来了“同步”在运行期间的开销。若特别在意这个问题，在关键区域中不会造成问题的方法就可保留为“不同步”，但注意这通常都不是十分容易判断。有资格的方法倾向于规模很小（如下例的 getCircleSize()）以及 / 或者“微小”。也就是说，这个方法调用在如此少的代码片里执行，以至于在执行期间对象不能改变。如果将这种方法设为“不同步”，可能对程序的执行速度不会有明显的影响。可能也将一个 Bean 的所有 public 方法都设为 synchronized，并只有在保证特别必要、而且会造成一个差异的情况下，才将 synchronized 关键字删去。

(2) 如果将一个多造型事件送给一系列对那个事件感兴趣的“听众”，必须假在列表中移动的时候可以添加或者删除。

第一点很容易处理，但第二点需要考虑更多的东西。让我们以前一章提供的 BangBean.java 为例。在那个例子中，我们忽略了 synchronized 关键字（那时还没有引入呢），并将造型设为单造型，从而回避了多线程的问题。在下面这个修改过的版本中，我们使其能在多线程环境中工作，并为事件采用了多造型技术：

781-784 页程序

很容易就可以为方法添加 synchronized。但注意在 addActionListener()和 removeActionListener()中，现在添加了 ActionListener，并从一个 Vector 中移去，所以能够根据自己愿望使用任意多个。

我们注意到，notifyListeners()方法并未设为“同步”。可从多个线程中发出对这个方法的调用。另外，在对 notifyListeners()调用的中途，也可能发出对 addActionListener()和 removeActionListener()的调用。这显然会造成问题，因为它否定了 Vector actionListeners。为缓解这个问题，我们在一个 synchronized 从句中“克隆”了 Vector，并对克隆进行了否定。这样便可在不影响 notifyListeners()的前提下，对 Vector 进行操纵。

paint()方法也没有设为“同步”。与单纯地添加自己的方法相比，决定是否对过载的方法进行同步要困难得多。在这个例子中，无论 paint()是否“同步”，它似乎都能正常地工作。但必须考虑的问题包括：

(1) 方法会在对象内部修改“关键”变量的状态吗？为判断一个变量是否“关键”，必须知道它是否会被程序中的其他线程读取或设置（就目前的情况看，读取或设置几乎肯定是通过“同步”方法进行的，所以可以只对它们进行检查）。对 paint()的情况来说，不会发生任何修改。

(2) 方法要以这些“关键”变量的状态为基础吗？如果一个“同步”方法修改了一个变量，而我们的方法要用到这个变量，那么一般都愿意把自己的方法也设为“同步”。基于这一前提，大家可观察到 cSize 由“同步”方法进行了修改，所以 paint()应当是“同步”的。但在这里，我们可以问：“假如 cSize 在 paint()执行期间发生了变化，会发生的最糟糕的事情是什么呢？”如果发现情况不算太坏，而且仅仅是暂时的效果，那么最好保持 paint()的“不同步”状态，以避免同步方法调用带来的额外开销。

(3) 要留意的第三条线索是 paint()基础类版本是否“同步”，在这里它不是同步的。这并不是一个非常严格的参数，仅仅是一条“线索”。比如在目前的情况下，通过同步方法（好 cSize）改变的一个字段已合成到 paint()公式里，而且可能已改变了情况。但请注意，synchronized 不能继承——也就是说，假如一个方法在基础类中是“同步”的，那么在衍生类过载版本中，它不会自动进入“同步”状态。

TestBangBean2 中的测试代码已在前一章的基础上进行了修改，已在其中加入了额外的“听众”，从而演示了 BangBean2 的多造型能力。

14.3 堵塞

一个线程可以有四种状态：

(1) 新（New）：线程对象已经创建，但尚未启动，所以不可运行。

(2) 可运行（Runnable）：意味着一旦时间分片机制有空闲的 CPU 周期提供给一个线程，那个线程便可立即开始运行。因此，线程可能在、也可能不在运行当中，但一旦条件许可，没有什么能阻止它的运行——它既没有“死”掉，也未被“堵塞”。

(3) 死（Dead）：从自己的 run()方法中返回后，一个线程便已“死”掉。亦可调用 stop()令其死掉，但会产生一个违例——属于 Error 的一个子类（也就是说，我们通常不捕获它）。记住一个违例的“掷”出应当是一个特殊事件，而不是正常程序运行的一部分。所以不建议你使用 stop()（在 Java 1.2 则是坚决反对）。

另外还有一个 destroy()方法（它永远不会实现），应该尽可能地避免调用它，因为它非常武断，根本不会解除对象的锁定。

(4) 堵塞（Blocked）：线程可以运行，但有某种东西阻碍了它。若线程处于堵塞状态，调度机制可以简单地跳过它，不给它分配任何 CPU 时间。除非线程再次进入“可运行”状态，否则不会采取任何操作。

14.3.1 为何会堵塞

堵塞状态是前述四种状态中最有趣的，值得我们作进一步的探讨。线程被堵塞可能是由下述五方面的原因造成的：

(1) 调用 sleep(毫秒数)，使线程进入“睡眠”状态。在规定的时间内，这个线程是不会运行的。

(2) 用 suspend()暂停了线程的执行。除非线程收到 resume()消息，否则不会返回“可运行”状态。

(3) 用 wait()暂停了线程的执行。除非线程收到 nofify()或者 notifyAll()消息，否则不会变成“可运行”（是的，这看起来同原因 2 非常相象，但有一个明显的区别是我们马上要揭示的）。

(4) 线程正在等候一些 IO（输入输出）操作完成。

(5) 线程试图调用另一个对象的“同步”方法，但那个对象处于锁定状态，暂时无法使用。

亦可调用 yield()（Thread 类的一个方法）自动放弃 CPU，以便其他线程能够运行。然而，假如调度机制觉得我们的线程已拥有足够的时间，并跳转到另一个线程，就会发生同样的事情。也就是说，没有什么能防止调度机制重新启动我们的线程。线程被堵塞后，便有一些原因造成它不能继续运行。

下面这个例子展示了进入堵塞状态的全部五种途径。它们全都存在于名为 Blocking.java 的一个文件中，但在这儿采用散落的片断进行解释（大家可注意到片断前后的“Continued”以及“Continuing”标志。利用第 17 章介绍的工具，可将这些片断连结到一起）。首先让我们看看基本的框架：

786-787 页程序

Blockable 类打算成为本例所有类的一个基础类。一个 Blockable 对象包含了一个名为 state 的 TextField（文本字段），用于显示出对象有关的信息。用于显示这些信息的方法叫作 update()。我们发现它用 getClass.getName()来产生类名，而不是仅仅把它打印出来；这是由于 update(0 不知道自己为其调用的那个类的准确名字，因为那个类是从 Blockable 衍生出来的。

在 Blockable 中，变动指示符是一个 int i；衍生类的 run()方法会为其增值。

针对每个 Bloackable 对象，都会启动 Peeker 类的一个线程。Peeker 的任务是调用 read()方法，检查与自己关联的 Blockable 对象，看看 i 是否发生了变化，最后用它的 status 文本字段报告检查结果。注意 read()和 update()都是同步的，要求对象的锁定能自由解除，这一点非常重要。

1. 睡眠

这个程序的第一项测试是用 sleep()作出的：

788-789 页程序

在 Sleeper1 中，整个 run()方法都是同步的。我们可看到与这个对象关联在一起的 Peeker 可以正常运行，直到我们启动线程为止，随后 Peeker 便会完全停止。这正是“堵塞”的一种形式：因为 Sleeper1.run()是同步的，而且一旦线程启动，它就肯定在 run()内部，方法永远不会放弃对象锁定，造成 Peeker 线程的堵塞。

Sleeper2 通过设置不同步的运行，提供了一种解决方案。只有 change()方法才是同步的，所以尽管 run()位于 sleep()内部，Peeker 仍然能访问自己需要的同步方法——read()。在这里，我们可看到在启动了 Sleeper2 线程以后，Peeker 会持续运行下去。

2. 暂停和恢复

这个例子接下来的一部分引入了“挂起”或者“暂停”（Suspend）的概述。Thread 类提供了一个名为 suspend()的方法，可临时中止线程；以及一个名为 resume()的方法，用于从暂停处开始恢复线程的执行。显然，我们可以推断出 resume()是由暂停线程外部的某个线程调用的。在这种情况下，需要用到一个名为 Resumer（恢复器）的独立类。演示暂停 / 恢复过程的每个类都有一个相关的恢复器。如下所示：

789-790 页程序

SuspendResume1 也提供了一个同步的 run()方法。同样地，当我们启动这个线程以后，就会发现与它关联的 Peeker 进入“堵塞”状态，等候对象锁被释放，但那永远不会发生。和往常一样，这个问题在 SuspendResume2 里得到了解决，它并不同步整个 run()方法，而是采用了一个单独的同步 change()方法。

对于 Java 1.2，大家应注意 suspend()和 resume()已获得强烈反对，因为 suspend()包含了对象锁，所以极易出现“死锁”现象。换言之，很容易就会看到许多被锁住的对象在傻乎乎地等待对方。这会造成整个应用程序的“凝固”。尽管在一些老程序中还能看到它们的踪迹，但在你写自己的程序时，无论如何都应避免。本章稍后就会讲述正确的方案是什么。

3. 等待和通知

通过前两个例子的实践，我们知道无论 sleep()还是 suspend()都不会在自己被调用的时候解除锁定。需要用到对象锁时，请务必注意这个问题。在另一方面，wait()方法在被调用时却会解除锁定，这意味着可在执行 wait()期间调用线程对象中的其他同步方法。但在接着的两个类中，我们看到 run()方法都是“同步”的。在 wait()期间，Peeker 仍然拥有对同步方法的完全访问权限。这是由于 wait()在挂起内部调用的方法时，会解除对象的锁定。

我们也可以看到 wait()的两种形式。第一种形式采用一个以毫秒为单位的参数，它具有与 sleep()中相同的含义：暂停这一段规定时间。区别在于在 wait()中，对象锁已被解除，而且能够自由地退出 wait()，因为一个 notify()可强行使时间流逝。

第二种形式不采用任何参数，这意味着 wait()会持续执行，直到 notify()介入

为止。而且在一段时间以后，不会自行中止。

wait()和 notify()比较特别的一个地方是这两个方法都属于基础类 Object 的一部分，不象 sleep()，suspend()以及 resume()那样属于 Thread 的一部分。尽管这表面看有点儿奇怪——居然让专门进行线程处理的东西成为通用基础类的一部分——但仔细想想又会释然，因为它们操纵的对象锁也属于每个对象的一部分。因此，我们可将一个 wait()置入任何同步方法内部，无论在那个类里是否准备进行涉及线程的处理。事实上，我们能调用 wait()的唯一地方是在一个同步的方法或代码块内部。若在一个不同步的方法内调用 wait()或者 notify()，尽管程序仍然会编译，但在运行它的时候，就会得到一个 IllegalMonitorStateException（非法监视器状态违例），而且会出现多少有点莫名其妙的一条消息：“current thread not owner”（当前线程不是所有人”。注意 sleep()，suspend()以及 resume()都能在不同步的方法内调用，因为它们不需要对锁定进行操作。

只能为自己的锁定调用 wait()和 notify()。同样地，仍然可以编译那些试图使用错误锁定的代码，但和往常一样会产生同样的 IllegalMonitorStateException 违例。我们没办法用其他人的对象锁来愚弄系统，但可要求另一个对象执行相应的操作，对它自己的锁进行操作。所以一种做法是创建一个同步方法，令其为自己的对象调用 notify()。但在 Notifier 中，我们会看到一个同步方法内部的 notify()：

792 页上程序

其中，wn2 是类型为 WaitNotify2 的对象。尽管并不属于 WaitNotify2 的一部分，这个方法仍然获得了 wn2 对象的锁定。在这个时候，它为 wn2 调用 notify()是合法的，不会得到 IllegalMonitorStateException 违例。

792-793 页程序

若必须等候其他某些条件（从线程外部加以控制）发生变化，同时又不想在线程内一直傻乎乎地等下去，一般就需要用到 wait()。wait()允许我们将线程置入“睡眠”状态，同时又“积极”地等待条件发生改变。而且只有在一个 notify()或 notifyAll()发生变化的时候，线程才会被唤醒，并检查条件是否有变。因此，我们认为它提供了在线程间进行同步的一种手段。

4. IO 堵塞

若一个数据流必须等候一些 IO 活动，便会自动进入“堵塞”状态。在本例下面列出的部分中，有两个类协同通用的 Reader 以及 Writer 对象工作（使用 Java 1.1 的流）。但在测试模型中，会设置一个管道化的数据流，使两个线程相互间能安全地传递数据（这正是使用管道流的目的）。

Sender 将数据置入 Writer，并“睡眠”随机长短的时间。然而，Receiver 本身并没有包括 sleep()，suspend()或者 wait()方法。但在执行 read()的时候，如果没有数据存在，它会自动进入“堵塞”状态。如下所示：

793-794 页程序

这两个类也将信息送入自己的 state 字段，并修改 i 值，使 Peeker 知道线程仍

在运行。

5. 测试

令人惊讶的是，主要的程序片（Applet）类非常简单，这是大多数工作都已置入 Blockable 框架的缘故。大概地说，我们创建了一个由 Blockable 对象构成的数组。而且由于每个对象都是一个线程，所以在按下“start”按钮后，它们会采取自己的行动。还有另一个按钮和 actionPerformed()从句，用于中止所有 Peeker 对象。由于 Java 1.2“反对”使用 Thread 的 stop()方法，所以可考虑采用这种折衷形式的中止方式。

为了在 Sender 和 Receiver 之间建立一个连接，我们创建了一个 PipedWriter 和一个 PipedReader。注意 PipedReader in 必须通过一个构建器参数同 PipedWriterout 连接起来。在那以后，我们在 out 内放进去的所有东西都可从 in 中提取出来——似乎那些东西是通过一个“管道”传输过去的。随后将 in 和 out 对象分别传递给 Receiver 和 Sender 构建器；后者将它们当作任意类型的 Reader 和 Writer 看待（也就是说，它们被“上溯”造型了）。

Blockable 句柄 b 的数组在定义之初并未得到初始化，因为管道化的数据流是不可在定义前设置好的（对 try 块的需要将成为障碍）：

795-796 页程序

在 init()中，注意循环会遍历整个数组，并为页添加 state 和 peeker.status 文本字段。

首次创建好 Blockable 线程以后，每个这样的线程都会自动创建并启动自己的 Peeker。所以我们会看到各个 Peeker 都在 Blockable 线程启动之前运行起来。这一点非常重要，因为在 Blockable 线程启动的时候，部分 Peeker 会被堵塞，并停止运行。弄懂这一点，将有助于我们加深对“堵塞”这一概念的认识。

14.3.2 死锁

由于线程可能进入堵塞状态，而且由于对象可能拥有“同步”方法——除非同步锁定被解除，否则线程不能访问那个对象——所以一个线程完全可能等候另一个对象，而另一个对象又在等候下一个对象，以此类推。这个“等候”链最可怕的情形就是进入封闭状态——最后那个对象等候的是第一个对象！此时，所有线程都会陷入无休止的相互等待状态，大家都动弹不得。我们将这种情况称为“死锁”。尽管这种情况并非经常出现，但一旦碰到，程序的调试将变得异常艰难。

就语言本身来说，尚未直接提供防止死锁的帮助措施，需要我们通过谨慎的设计来避免。如果有谁需要调试一个死锁的程序，他是没有任何窍门可用的。

1. Java 1.2 对 stop()，suspend()，resume()以及 destroy()的反对

为减少出现死锁的可能，Java 1.2 作出的一项贡献是“反对”使用 Thread 的 stop()，suspend()，resume()以及 destroy()方法。

之所以反对使用 stop()，是因为它不安全。它会解除由线程获取的所有锁定，而且如果对象处于一种不连贯状态（“被破坏”），那么其他线程能在那种状态下检查和修改它们。结果便造成了一种微妙的局面，我们很难检查出真正的问题所在。所以应尽量避免使用 stop()，应该采用 Blocking.java 那样的方法，用一个标

志告诉线程什么时候通过退出自己的 run()方法来中止自己的执行。

如果一个线程被堵塞，比如在它等候输入的时候，那么一般都不能象在 Blocking.java 中那样轮询一个标志。但在这些情况下，我们仍然不该使用 stop()，而应换用由 Thread 提供的 interrupt()方法，以便中止并退出堵塞的代码。

797-798 页程序

Blocked.run()内部的 wait()会产生堵塞的线程。当我们按下按钮以后，blocked（堵塞）的句柄就会设为 null，使垃圾收集器能够将其清除，然后调用对象的 interrupt()方法。如果是首次按下按钮，我们会看到线程正常退出。但在没有可供“杀死”的线程以后，看到的便只是按钮被按下而已。

suspend()和 resume()方法天生容易发生死锁。调用 suspend()的时候，目标线程会停下来，但却仍然持有在这之前获得的锁定。此时，其他任何线程都不能访问锁定的资源，除非被“挂起”的线程恢复运行。对任何线程来说，如果它们想恢复目标线程，同时又试图使用任何一个锁定的资源，就会造成令人难堪的死锁。所以我们不应该使用 suspend()和 resume()，而应在自己的 Thread 类中置入一个标志，指出线程应该活动还是挂起。若标志指出线程应该挂起，便用 wait()命其进入等待状态。若标志指出线程应当恢复，则用一个 notify()重新启动线程。我们可以修改前面的 Counter2.java 来实际体验一番。尽管两个版本的效果是差不多的，但大家会注意到代码的组织结构发生了很大的变化——为所有“听众”都使用了匿名的内部类，而且 Thread 是一个内部类。这使得程序的编写稍微方便一些，因为它取消了 Counter2.java 中一些额外的记录工作。

799-801 页程序

Suspendable 中的 suspended（已挂起）标志用于开关“挂起”或者“暂停”状态。为挂起一个线程，只需调用 fauxSuspend()将标志设为 true（真）即可。对标志状态的侦测是在 run()内进行的。就象本章早些时候提到的那样，wait()必须设为“同步”（synchronized），使其能够使用对象锁。在 fauxResume()中，suspended 标志被设为 false（假），并调用 notify()——由于这会在一个“同步”从句中唤醒 wait()，所以 fauxResume()方法也必须同步，使其能在调用 notify()之前取得对象锁（这样一来，对象锁可由要唤醒的那个 wait()使用）。如果遵照本程序展示的样式，可以避免使用 wait()和 notify()。

Thread 的 destroy()方法根本没有实现；它类似一个根本不能恢复的 suspend()，所以会发生与 suspend()一样的死锁问题。然而，这一方法没有得到明确的“反对”，也许会在 Java 以后的版本（1.2 版以后）实现，用于一些可以承受死锁危险的特殊场合。

大家可能会奇怪当初为什么要实现这些现在又被“反对”的方法。之所以会出现这种情况，大概是由于 Sun 公司主要让技术人员来决定对语言的改动，而不是那些市场销售人员。通常，技术人员比搞销售的更能理解语言的实质。当初犯下了错误以后，也能较为理智地正视它们。这意味着 Java 能够继续进步，即便这使 Java 程序员多少感到有些不便。就我自己来说，宁愿面对这些不便之处，也不愿看到语言停滞不前。

14.4 优先级

线程的优先级（Priority）告诉调试程序该线程的重要程度有多大。如果有大量线程都被堵塞，都在等候运行，调试程序会首先运行具有最高优先级的那个线程。然而，这并不表示优先级较低的线程不会运行（换言之，不会因为存在优先级而导致死锁）。若线程的优先级较低，只不过表示它被准许运行的机会小一些而已。

可用 getPriority()方法读取一个线程的优先级，并用 setPriority()改变它。在下面这个程序片中，大家会发现计数器的计数速度慢了下来，因为它们关联的线程分配了较低的优先级：

802-805 页程序

Ticker 采用本章前面构造好的形式，但有一个额外的 TextField（文本字段），用于显示线程的优先级；以及两个额外的按钮，用于人为提高及降低优先级。

也要注意 yield()的用法，它将控制权自动返回给调试程序（机制）。若不进行这样的处理，多线程机制仍会工作，但我们会发现它的运行速度慢了下来（试试删去对 yield()的调用）。亦可调用 sleep()，但假若那样做，计数频率就会改由 sleep()的持续时间控制，而不是优先级。

Counter5 中的 init()创建了由 10 个 Ticker2 构成的一个数组；它们的按钮以及输入字段（文本字段）由 Ticker2 构建器置入窗体。Counter5 增加了新的按钮，用于启动一切，以及用于提高和降低线程组的最大优先级。除此以外，还有一些标签用于显示一个线程可以采用的最大及最小优先级；以及一个特殊的文本字段，用于显示线程组的最大优先级（在下一节里，我们将全面讨论线程组的问题）。最后，父线程组的优先级也作为标签显示出来。

按下“up”（上）或“down”（下）按钮的时候，会先取得 Ticker2 当前的优先级，然后相应地提高或者降低。

运行该程序时，我们可注意到几件事情。首先，线程组的默认优先级是 5。即使在启动线程之前（或者在创建线程之前，这要求对代码进行适当的修改）将最大优先级降到 5 以下，每个线程都会有一个 5 的默认优先级。

最简单的测试是获取一个计数器，将它的优先级降低至 1，此时应观察到它的计数频率显著放慢。现在试着再次提高优先级，可以升高回线程组的优先级，但不能再高了。现在将线程组的优先级降低两次。线程的优先级不会改变，但假若试图提高或者降低它，就会发现这个优先级自动变成线程组的优先级。此外，新线程仍然具有一个默认优先级，即使它比组的优先级还要高（换句话说，不要指望利用组优先级来防止新线程拥有比现有的更高的优先级）。

最后，试着提高组的最大优先级。可以发现，这样做是没有效果的。我们只能减少线程组的最大优先级，而不能增大它。

14.4.1 线程组

所有线程都隶属于一个线程组。那可以是一个默认线程组，亦可是一个创建线程时明确指定的组。在创建之初，线程被限制到一个组里，而且不能改变到一个不同的组。每个应用都至少有一个线程从属于系统线程组。若创建多个线程而不指定一个组，它们就会自动归属于系统线程组。

线程组也必须从属于其他线程组。必须在构建器里指定新线程组从属于哪个

线程组。若在创建一个线程组的时候没有指定它的归属，则同样会自动成为系统线程组的一名属下。因此，一个应用程序中的所有线程组最终都会将系统线程组作为自己的“父”。

之所以要提出“线程组”的概念，很难从字面上找到原因。这多少为我们讨论的主题带来了一些混乱。一般地说，我们认为是由于“安全”或者“保密”方面的理由才使用线程组的。根据 Arnold 和 Gosling 的说法：“线程组中的线程可以修改组内的其他线程，包括那些位于分层结构最深处的。一个线程不能修改位于自己所在组或者下属组之外的任何线程”（注释①）。然而，我们很难判断“修改”在这儿的具体含义是什么。下面这个例子展示了位于一个“叶子组”内的线程能修改它所在线程组树的所有线程的优先级，同时还能为这个“树”内的所有线程都调用一个方法。

①：《The Java Programming Language》第 179 页。该书由 Arnold 和 Jams Gosling 编著，Addison-Wesley 于 1996 年出版

807-808 页程序

在 main()中，我们创建了几个 ThreadGroup（线程组），每个都位于不同的“叶”上：x 没有参数，只有它的名字（一个 String），所以会自动进入“system”（系统）线程组；y 位于 x 下方，而 z 位于 y 下方。注意初始化是按照文字顺序进行的，所以代码合法。

有两个线程创建之后进入了不同的线程组。其中，TestThread1 没有一个 run()方法，但有一个 f()，用于通知线程以及打印出一些东西，以便我们知道它已被调用。而 TestThread2 属于 TestThread1 的一个子类，它的 run()非常详尽，要做许多事情。首先，它获得当前线程所在的线程组，然后利用 getParent()在继承树中向上移动两级（这样做是有道理的，因为我想把 TestThread2 在分级结构中向下移动两级）。随后，我们调用方法 activeCount()，查询这个线程组以及所有子线程组内有多少个线程，从而创建由指向 Thread 的句柄构成的一个数组。enumerate()方法将指向所有这些线程的句柄置入数组 gAll 里。然后在整个数组里遍历，为每个线程都调用 f()方法，同时修改优先级。这样一来，位于一个“叶子”线程组里的线程就修改了位于父线程组的线程。

调试方法 list()打印出与一个线程组有关的所有信息，把它们作为标准输出。在我们对线程组的行为进行调查的时候，这样做是相当有好处的。下面是程序的输出：

808 页下程序

list()不仅打印出 ThreadGroup 或者 Thread 的类名，也打印出了线程组的名字以及它的最高优先级。对于线程，则打印出它们的名字，并接上线程优先级以及所属的线程组。注意 list()会对线程和线程组进行缩排处理，指出它们是未缩排的线程组的“子”。

大家可看到 f()是由 TestThread2 的 run()方法调用的，所以很明显，组内的所有线程都是相当脆弱的。然而，我们只能访问那些从自己的 system 线程组树分支出来的线程，而且或许这就是所谓“安全”的意思。我们不能访问其他任何人

的系统线程树。

1. 线程组的控制

抛开安全问题不谈，线程组最有用的一个地方就是控制：只需用单个命令即可完成对整个线程组的操作。下面这个例子演示了这一点，并对线程组内优先级的限制进行了说明。括号内的注释数字便于大家比较输出结果：

809-810 页程序

下面的输出结果已进行了适当的编辑，以便用一页能够装下（java.lang.已被删去），而且添加了适当的数字，与前面程序列表中括号里的数字对应：

811 页程序

所有程序都至少有一个线程在运行，而且 main()采取的第一项行动便是调用 Thread 的一个 static（静态）方法，名为 currentThread()。从这个线程开始，线程组将被创建，而且会为结果调用 list()。输出如下：

812 页上程序

我们可以看到，主线程组的名字是 system，而主线程的名字是 main，而且它从属于 system 线程组。

第二个练习显示出 system 组的最高优先级可以减少，而且 main 线程可以增大自己的优先级：

812 页中上程序

第三个练习创建一个新的线程组，名为 g1；它自动从属于 system 线程组，因为并没有明确指定它的归属关系。我们在 g1 内部放置了一个新线程，名为 A。随后，我们试着将这个组的最大优先级设到最高的级别，并将 A 的优先级也设到最高一级。结果如下：

812 页中程序

可以看出，不可能将线程组的最大优先级设为高于它的父线程组。

第四个练习将 g1 的最大优先级降低两级，然后试着把它升至 Thread.MAX_PRIORITY。结果如下：

812 页中下程序

同样可以看出，提高最大优先级的企图是失败的。我们只能降低一个线程组的最大优先级，而不能提高它。此外，注意线程 A 的优先级并未改变，而且它现在高于线程组的最大优先级。也就是说，线程组最大优先级的变化并不能对现有线程造成影响。

第五个练习试着将一个新线程设为最大优先级。如下所示：

812 页下程序

因此，新线程不能变到比最大线程组优先级还要高的一级。

这个程序的默认线程优先级是 6；若新建一个线程，那就是它的默认优先级，而且不会发生变化，除非对优先级进行了特别的处理。练习六将把线程组的最大优先级降至默认线程优先级以下，看看在这种情况下新建一个线程会发生什么事情：

813 页上程序

尽管线程组现在的最大优先级是 3，但仍然用默认优先级 6 来创建新线程。所以，线程组的最大优先级不会影响默认优先级（事实上，似乎没有办法可以设置新线程的默认优先级）。

改变了优先级后，接下来试试将其降低一级，结果如下：

813 页中程序

因此，只有在试图改变优先级的时候，才会强迫遵守线程组最大优先级的限制。

我们在(8)和(9)中进行了类似的试验。在这里，我们创建了一个新的线程组，名为 g2，将其作为 g1 的一个子组，并改变了它的最大优先级。大家可以看到，g2 的优先级无论如何都不可能高于 g1：

813 页中下程序

也要注意在 g2 创建的时候，它会被自动设为 g1 的线程组最大优先级。

经过所有这些实验以后，整个线程组和线程系统都会被打印出来，如下所示：

813-814 页程序

所以由线程组的规则所限，一个子组的最大优先级在任何时候都只能低于或等于它的父组的最大优先级。

本程序的最后一个部分演示了用于整组线程的方法。程序首先遍历整个线程树，并启动每一个尚未启动的线程。例如，system 组随后会被挂起（暂停），最后被中止（尽管用 suspend()和 stop()对整个线程组进行操作看起来似乎很有趣，但应注意这些方法在 Java 1.2 里都是被“反对”的）。但在挂起 system 组的同时，也挂起了 main 线程，而且整个程序都会关闭。所以永远不会达到让线程中止的那一步。实际上，假如真的中止了 main 线程，它会“掷”出一个 ThreadDeath 违例，所以我们通常不这样做。由于 ThreadGroup 是从 Object 继承的，其中包含了 wait()方法，所以也能调用 wait(秒数×1000)，令程序暂停运行任意秒数的时间。当然，事前必须在一个同步块里取得对象锁。

ThreadGroup 类也提供了 suspend()和 resume()方法，所以能中止和启动整个

线程组和它的所有线程，也能中止和启动它的子组，所有这些只需一个命令即可（再次提醒，suspend()和 resume()都是 Java 1.2 所“反对”的）。

从表面看，线程组似乎有些让人摸不着头脑，但请注意我们很少需要直接使用它们。

14.5 回顾 runnable

在本章早些时候，我曾建议大家在将一个程序片或主 Frame 当作 Runnable 的实现形式之前，一定要好好地想一想。若采用那种方式，就只能在自己的程序中使用其中的一个线程。这便限制了灵活性，一旦需要用到属于那种类型的多个线程，就会遇到不必要的麻烦。

当然，如果必须从一个类继承，而且想使类具有线程处理能力，则 Runnable 是一种正确的方案。本章最后一个例子对这一点进行了剖析，制作了一个 RunnableCanvas 类，用于为自己描绘不同的颜色（Canvas 是“画布”的意思）。这个应用被设计成从命令行获得参数值，以决定颜色网格有多大，以及颜色发生变化之间的 sleep()有多长。通过运用这些值，大家能体验到线程一些有趣而且可能令人费解的特性：

815-816 页程序

ColorBoxes 是一个典型的应用（程序），有一个构建器用于设置 GUI。这个构建器采用 int grid 的一个参数，用它设置 GridLayout（网格布局），使每一维里都有一个 grid 单元。随后，它添加适当数量的 CBox 对象，用它们填充网格，并为每一个都传递 pause 值。在 main()中，我们可看到如何对 pause 和 grid 的默认值进行修改（如果用命令行参数传递）。

CBox 是进行正式工作的地方。它是从 Canvas 继承的，并实现了 Runnable 接口，使每个 Canvas 也能是一个 Thread。记住在实现 Runnable 的时候，并没有实际产生一个 Thread 对象，只是一个拥有 run()方法的类。因此，我们必须明确地创建一个 Thread 对象，并将 Runnable 对象传递给构建器，随后调用 start()（在构建器里进行）。在 CBox 里，这个线程的名字叫作 t。

请留意数组 colors，它对 Color 类中的所有颜色进行了列举（枚举）。它在 newColor()中用于产生一种随机选择的颜色。当前的单元（格）颜色是 cColor。

paint()则相当简单——只是将颜色设为 cColor，然后用那种颜色填充整张画布（Canvas）。

在 run()中，我们看到一个无限循环，它将 cColor 设为一种随机颜色，然后调用 repaint()把它显示出来。随后，对线程执行 sleep()，使其“休眠”由命令行指定的时间长度。

由于这种设计方案非常灵活，而且线程处理同每个 Canvas 元素都紧密结合在一起，所以在理论上可以生成任意多的线程（但在实际应用中，这要受到 JVM 能够从容对付的线程数量的限制）。

这个程序也为我们提供了一个有趣的评测基准，因为它揭示了不同 JVM 机制在速度上造成的戏剧性的差异。

14.5.1 过多的线程

有些时候，我们会发现 ColorBoxes 几乎陷于停顿状态。在我自己的机器上，

这一情况在产生了 10×10 的网格之后发生了。为什么会这样呢？自然地，我们有理由怀疑 AWT 对它做了什么事情。所以这里有一个例子能够测试那个猜测，它产生了较少的线程。代码经过了重新组织，使一个 Vector 实现了 Runnable，而且那个 Vector 容纳了数量众多的色块，并随机挑选一些进行更新。随后，我们创建大量这些 Vector 对象，数量大致取决于我们挑选的网格维数。结果便是我们得到比色块少得多的线程。所以假如有一个速度的加快，我们就能立即知道，因为前例的线程数量太多了。如下所示：

817-819 页程序

在 ColorBoxes2 中，我们创建了 CBoxVector 的一个数组，并对其初始化，使其容下各个 CBoxVector 网格。每个网格都知道自己该“睡眠”多长的时间。随后为每个 CBoxVector 都添加等量的 Cbox2 对象，而且将每个 Vector 都告诉给 go()，用它来启动自己的线程。

CBox2 类似 CBox——能用一种随机选择的颜色描绘自己。但那就是 CBox2 能够做的全部工作。所有涉及线程的处理都已移至 CBoxVector 进行。

CBoxVector 也可以拥有继承的 Thread，并有一个类型为 Vector 的成员对象。这样设计的好处就是 addElement()和 elementAt()方法可以获得特定的参数以及返回值类型，而不是只能获得常规 Object（它们的名字也可以变得更短）。然而，这里采用的设计表面上看需要较少的代码。除此以外，它会自动保留一个 Vector 的其他所有行为。由于 elementAt()需要大量进行“封闭”工作，用到许多括号，所以随着代码主体的扩充，最终仍有可能需要大量代码。

和以前一样，在我们实现 Runnable 的时候，并没有获得与 Thread 配套提供的所有功能，所以必须创建一个新的 Thread，并将自己传递给它的构建器，以便正式“启动”——start()——一些东西。大家在 CBoxVector 构建器和 go()里都可以体会到这一点。run()方法简单地选择 Vector 里的一个随机元素编号，并为那个元素调用 nextColor()，令其挑选一种新的随机颜色。

运行这个程序时，大家会发现它确实变得更快，响应也更迅速（比如在中断它的时候，它能更快地停下来）。而且随着网格尺寸的壮大，它也不会经常性地陷于“停顿”状态。因此，线程的处理又多了一项新的考虑因素：必须随时检查自己有没有“太多的线程”（无论对什么程序和运行平台）。若线程太多，必须试着使用上面介绍的技术，对程序中的线程数量进行“平衡”。如果在一个多线程的程序中遇到了性能上的问题，那么现在有许多因素需要检查：

(1) 对 sleep，yield()以及／或者 wait()的调用足够多吗？

(2) sleep()的调用时间足够长吗？

(3) 运行的线程数是不是太多？

(4) 试过不同的平台和 JVM 吗？

象这样的一些问题是造成多线程应用程序的编制成为一种“技术活”的原因之一。

14.6 总结

何时使用多线程技术，以及何时避免用它，这是我们需要掌握的重要课题。骼它的主要目的是对大量任务进行有序的管理。通过多个任务的混合使用，可以更有效地利用计算机资源，或者对用户来说显得更方便。资源均衡的经典问题是

在 IO 等候期间如何利用 CPU。至于用户方面的方便性，最经典的问题就是如何在一个长时间的下载过程中监视并灵敏地反应一个“停止”（stop）按钮的按下。

多线程的主要缺点包括：

(1) 等候使用共享资源时造成程序的运行速度变慢。

(2) 对线程进行管理要求的额外 CPU 开销。

(3) 复杂程度无意义的加大，比如用独立的线程来更新数组内每个元素的愚蠢主意。

(4) 漫长的等待、浪费精力的资源竞争以及死锁等多线程症状。

线程另一个优点是它们用“轻度”执行切换（100 条指令的顺序）取代了“重度”进程场景切换（1000 条指令）。由于一个进程内的所有线程共享相同的内存空间，所以“轻度”场景切换只改变程序的执行和本地变量。而在“重度”场景切换时，一个进程的改变要求必须完整地交换内存空间。

线程处理看来好象进入了一个全新的领域，似乎要求我们学习一种全新的程序设计语言——或者至少学习一系列新的语言概念。由于大多数微机操作系统都提供了对线程的支持，所以程序设计语言或者库里也出现了对线程的扩展。不管在什么情况下，涉及线程的程序设计：

(1) 刚开始会让人摸不着头脑，要求改换我们传统的编程思路；

(2) 其他语言对线程的支持看来是类似的。所以一旦掌握了线程的概念，在其他环境也不会有太大的困难。尽管对线程的支持使 Java 语言的复杂程度多少有些增加，但请不要责怪 Java。毕竟，利用线程可以做许多有益的事情。

多个线程可能共享同一个资源（比如一个对象里的内存），这是运用线程时面临的最大的一个麻烦。必须保证多个线程不会同时试图读取和修改那个资源。这要求技巧性地运用 synchronized（同步）关键字。它是一个有用的工具，但必须真正掌握它，因为假若操作不当，极易出现死锁。

除此以外，运用线程时还要注意一个非常特殊的问题。由于根据 Java 的设计，它允许我们根据需要创建任意数量的线程——至少理论上如此（例如，假设为一项工程方面的有限元素分析创建数以百万的线程，这对 Java 来说并非实际）。然而，我们一般都要控制自己创建的线程数量的上限。因为在某些情况下，大量线程会将场面变得一团糟，所以工作都会几乎陷于停顿。临界点并不象对象那样可以达到几千个，而是在 100 以下。一般情况下，我们只创建少数几个关键线程，用它们解决某个特定的问题。这时数量的限制问题不大。但在较常规的一些设计中，这一限制确实会使我们感到束手束脚。

大家要注意线程处理中一个不是十分直观的问题。由于采用了线程“调度”机制，所以通过在 run()的主循环中插入对 sleep()的调用，一般都可以使自己的程序运行得更快一些。这使它对编程技巧的要求非常高，特别是在更长的延迟似乎反而能提高性能的时候。当然，之所以会出现这种情况，是由于在正在运行的线程准备进入“休眠”状态之前，较短的延迟可能造成“sleep()结束”调度机制的中断。这便强迫调度机制将其中止，并于稍后重新启动，以便它能做完自己的事情，再进入休眠状态。必须多想一想，才能意识到事情真正的麻烦程度。

本章遗漏的一件事情是一个动画例子，这是目前程序片最流行的一种应用。然而，Java JDK 配套提供了解决这个问题的一整套方案（并可播放声音），大家可到 java.sun.com 的演示区域下载。此外，我们完全有理由相信未来版本的 Java 会提供更好的动画支持——尽管目前的 Web 涌现出了与传统方式完全不同的非 Java、非程序化的许多动画方案。如果想系统学习 Java 动画的工作原理，可参考

《Core Java——核心 Java》一书，由 Cornell&Horstmann 编著，Prentice-Hall 于 1997 年出版。若欲更深入地了解线程处理，请参考《Concurrent Programming in Java——Java 中的并发编程》，由 Doug Lea 编著，Addison-Wiseley 于 1997 年出版；或者《Java Threads——Java 线程》，Oaks&Wong 编著，O'Reilly 于 1997 年出版。

14.7 练习

(1) 从 Thread 继承一个类，并（过载）覆盖 run()方法。在 run()内，打印出一条消息，然后调用 sleep()。重复三遍这些操作，然后从 run()返回。在构建器中放置一条启动消息，并覆盖 finalize()，打印一条关闭消息。创建一个独立的线程类，使它在 run()内调用 System.gc()和 System.runFinalization()，并打印一条消息，表明调用成功。创建这两种类型的几个线程，然后运行它们，看看会发生什么。

(2) 修改 Counter2.java，使线程成为一个内部类，而且不需要明确保存指向 Counter2 的一个。

(3) 修改 Sharing2.java，在 TwoCounter 的 run()方法内部添加一个 synchronized（同步）块，而不是同步整个 run()方法。

(4) 创建两个 Thread 子类，第一个的 run()方法用于最开始的启动，并捕获第二个 Thread 对象的句柄，然后调用 wait()。第二个类的 run()应在过几秒后为第一个线程调用 modifyAll()，使第一个线程能打印出一条消息。

(5) 在 Ticker2 内的 Counter5.java 中，删除 yield()，并解释一下结果。用一个 sleep()换掉 yield()，再解释一下结果。

(6) 在 ThreadGroup1.java 中，将对 sys.suspend()的调用换成对线程组的一个 wait()调用，令其等候 2 秒钟。为了保证获得正确的结果，必须在一个同步块内取得 sys 的对象锁。

(7) 修改 Daemons.java，使 main()有一个 sleep()，而不是一个 readLine()。实验不同的睡眠时间，看看会有什么发生。

(8) 到第 7 章（中间部分）找到那个 GreenhouseControls.java 例子，它应该由三个文件构成。在 Event.java 中，Event 类建立在对时间的监视基础上。修改这个 Event，使其成为一个线程。然后修改其余的设计，使它们能与新的、以线程为基础的 Event 正常协作。