

# 第14章 多线程

线程的概念

中断线程

线程状态

线程属性

同步

阻塞队列

线程安全的集合

Callable与Future

执行器

同步器

线程与Swing

读者可能已经很熟悉操作系统中的多任务 ( multitasking ) : 在同一时刻运行多个程序的能力。例如, 在编辑或下载邮件的同时可以打印文件。今天, 人们很可能有单台拥有多个CPU的计算机, 但是, 并发执行的进程数目并不是由CPU数目制约的。操作系统将CPU的时间片分配给每一个进程, 给人并行处理的感觉。

多线程程序在较低的层次上扩展了多任务的概念: 一个程序同时执行多个任务。通常, 每一个任务称为一个线程 ( thread ), 它是线程控制的简称。可以同时运行一个以上线程的程序称为多线程程序 ( multithreaded )。

然而, 多进程与多线程有哪些区别呢? 本质的区别在于每个进程拥有自己的一整套变量, 而线程则共享数据。这听起来似乎有些风险, 的确也是这样, 在本章稍后将可以看到这个问题。然而, 共享变量使线程之间的通信比进程之间的通信更有效、更容易。此外, 在有些操作系统中, 与进程相比较, 线程更“轻量级”, 创建、撤销一个线程比启动新进程的开销要小得多。

在实际应用中, 多线程非常有用。例如, 一个浏览器可以同时下载几幅图片。一个Web服务器需要同时处理几个并发的请求。图形用户界面 ( GUI ) 程序用一个独立的线程从宿主操作环境中收集用户界面的事件。本章将介绍如何为Java 应用程序添加多线程能力。

在Java SE 5.0中, 多线程发生了重大变化, 并增加了大量的类和接口, 为大多数应用程序员所需要的多线程机制提供了高质量的实现。本章将介绍Java SE 5.0新增的特性以及类的同步机制, 并帮助大家从中做出适当的选择。

温馨提示: 多线程可能会变得相当复杂。本章涵盖了应用程序可能需要的所有工具。尽管如此, 对于更复杂的系统级程序设计, 建议参看更高级的参考文献, 例如: Brian Goetz ( Addison-Wesley Professional, 2006 ) 的《Java Concurrency in Practice》。

## 14.1 线程的概念

这里从察看一个没有使用多线程的程序开始。用户很难让它执行多个任务。在对其进行剖析之后, 将展示让这个程序运行几个彼此独立的多个线程是很容易的。这个程序采用不断地移

动位置的方式实现球跳动的动画效果，如果发现球碰到墙壁，将进行刷新（见图14-1）。

当点击Start按钮时，程序将从屏幕的左上角弹出一个球，这个球便开始弹跳。Start按钮的处理程序将调用addBall方法。这个方法循环运行1000次move。每调用一次move，球就会移动一点，当碰到墙壁时，球将调整方向，并重新绘制面板。

```
Ball ball = new Ball();
panel.add(ball);
for (int i = 1; i <= STEPS; i++)
{
    ball.move(panel.getBounds());
    panel.paint(panel.getGraphics());
    Thread.sleep(DELAY);
}
```



图14-1 使用线程演示跳动的球

Thread类的静态sleep方法将暂停给定的毫秒数。

调用Thread.sleep不会创建一个新线程，sleep是Thread类的静态方法，用于暂停当前线程的活动。

sleep方法可以抛出一个InterruptedException异常。稍后将讨论这个异常以及对它的处理。现在，只是在发生异常时简单地终止弹跳。

如果运行这个程序，球就会自如地来回弹跳，但是，这个程序完全控制了整个应用程序。如果你在球完成1000次弹跳之前已经感到厌倦了，并点击Close按钮会发现球仍然还在弹跳。在球自己结束弹跳之前无法与程序进行交互。



注释：如果仔细地阅读本节末尾的代码会看到BounceFrame类的addBall方法中有调用

```
comp.paint(comp.getGraphics())
```

这一点很奇怪。一般来说，应该调用repaint方法让AWT获得图形上下文并负责绘制。但是，如果试图在这个程序中调用comp.repaint()会发现没有重画面板，这是因为addBall方法完全掌握着控制权。另外，还要注意ball组件扩展于JPanel；这会让擦除背景变得非常容易。接下来的程序将使用一个专门的线程计算球的位置，并会重新使用大家熟悉的repaint和JComponent。

显然，这个程序的性能相当糟糕。人们肯定不愿意让程序用这种方式完成一个非常耗时的工作。毕竟，当通过网络连接读取数据时，阻塞其他任务是经常发生的，有时确实想要中断读取操作。例如，假设下载一幅大图片。当看到一部分图片后，决定不需要或不想再看剩余的部分了，此时，肯定希望能够点击Stop按钮或Back按钮中断下载操作。下一节将介绍如何通过运行一个线程中的关键代码来保持用户对程序的控制权。

例14-1到例14-3给出了这个程序的代码。

## 例14-1 Bounce.java

```
1. import java.awt.*;
2. import java.awt.event.*;
3. import javax.swing.*;
4.
5. /**
6.  * Shows an animated bouncing ball.
7.  * @version 1.33 2007-05-17
8.  * @author Cay Horstmann
9.  */
10. public class Bounce
11. {
12.     public static void main(String[] args)
13.     {
14.         EventQueue.invokeLater(new Runnable()
15.         {
16.             public void run()
17.             {
18.                 JFrame frame = new BounceFrame();
19.                 frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
20.                 frame.setVisible(true);
21.             }
22.         });
23.     }
24. }
25.
26. /**
27.  * The frame with ball component and buttons.
28.  */
29. class BounceFrame extends JFrame
30. {
31.     /**
32.      * Constructs the frame with the component for showing the bouncing ball and Start and
33.      * Close buttons
34.      */
35.     public BounceFrame()
36.     {
37.         setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
38.         setTitle("Bounce");
39.
40.         comp = new BallComponent();
41.         add(comp, BorderLayout.CENTER);
42.         JPanel buttonPanel = new JPanel();
43.         addButton(buttonPanel, "Start", new ActionListener()
44.         {
45.             public void actionPerformed(ActionEvent event)
46.             {
47.                 addBall();
48.             }
49.         });
50.
51.         addButton(buttonPanel, "Close", new ActionListener()
52.         {
53.             public void actionPerformed(ActionEvent event)
54.             {
```

```

55.         System.exit(0);
56.     }
57. });
58.     add(buttonPanel, BorderLayout.SOUTH);
59. }
60.
61. /**
62.  * Adds a button to a container.
63.  * @param c the container
64.  * @param title the button title
65.  * @param listener the action listener for the button
66.  */
67. public void addButton(Container c, String title, ActionListener listener)
68. {
69.     JButton button = new JButton(title);
70.     c.add(button);
71.     button.addActionListener(listener);
72. }
73.
74. /**
75.  * Adds a bouncing ball to the panel and makes it bounce 1,000 times.
76.  */
77. public void addBall()
78. {
79.     try
80.     {
81.         Ball ball = new Ball();
82.         comp.add(ball);
83.
84.         for (int i = 1; i <= STEPS; i++)
85.         {
86.             ball.move(comp.getBounds());
87.             comp.paint(comp.getGraphics());
88.             Thread.sleep(DELAY);
89.         }
90.     }
91.     catch (InterruptedException e)
92.     {
93.     }
94. }
95.
96. private BallComponent comp;
97. public static final int DEFAULT_WIDTH = 450;
98. public static final int DEFAULT_HEIGHT = 350;
99. public static final int STEPS = 1000;
100. public static final int DELAY = 3;
101. }

```

#### 例14-2 Ball.java

```

1. import java.awt.geom.*;
2.
3. /**
4.  * A ball that moves and bounces off the edges of a rectangle
5.  * @version 1.33 2007-05-17
6.  * @author Cay Horstmann

```

```
7.  */
8. public class Ball
9. {
10.     /**
11.      * Moves the ball to the next position, reversing direction if it hits one of the edges
12.      */
13.     public void move(Rectangle2D bounds)
14.     {
15.         x += dx;
16.         y += dy;
17.         if (x < bounds.getMinX())
18.         {
19.             x = bounds.getMinX();
20.             dx = -dx;
21.         }
22.         if (x + XSIZE >= bounds.getMaxX())
23.         {
24.             x = bounds.getMaxX() - XSIZE;
25.             dx = -dx;
26.         }
27.         if (y < bounds.getMinY())
28.         {
29.             y = bounds.getMinY();
30.             dy = -dy;
31.         }
32.         if (y + YSIZE >= bounds.getMaxY())
33.         {
34.             y = bounds.getMaxY() - YSIZE;
35.             dy = -dy;
36.         }
37.     }
38.
39.     /**
40.      * Gets the shape of the ball at its current position.
41.      */
42.     public Ellipse2D getShape()
43.     {
44.         return new Ellipse2D.Double(x, y, XSIZE, YSIZE);
45.     }
46.
47.     private static final int XSIZE = 15;
48.     private static final int YSIZE = 15;
49.     private double x = 0;
50.     private double y = 0;
51.     private double dx = 1;
52.     private double dy = 1;
53. }
```

#### 例14-3 BallComponent.java

```
1. import java.awt.*;
2. import java.util.*;
3. import javax.swing.*;
4.
5. /**
6.  * The component that draws the balls.
```

```

7.  * @version 1.33 2007-05-17
8.  * @author Cay Horstmann
9.  */
10. public class BallComponent extends JPanel
11. {
12.     /**
13.      * Add a ball to the component.
14.      * @param b the ball to add
15.      */
16.     public void add(Ball b)
17.     {
18.         balls.add(b);
19.     }
20.
21.     public void paintComponent(Graphics g)
22.     {
23.         super.paintComponent(g); // erase background
24.         Graphics2D g2 = (Graphics2D) g;
25.         for (Ball b : balls)
26.         {
27.             g2.fill(b.getShape());
28.         }
29.     }
30.
31.     private ArrayList<Ball> balls = new ArrayList<Ball>();
32. }

```

#### **API** java.lang.Thread 1.0

- static void sleep(long millis)

休眠给定的毫秒数。

参数：millis      休眠的毫秒数

#### 使用线程给其他任务提供机会

可以将移动球的代码放置在一个独立的线程中，运行这段代码可以提高弹跳球的响应能力。实际上，可以发起多个球，每个球都在自己的线程中运行。另外，AWT的事件分派线程（event dispatch thread）将一直地并行运行，以处理用户界面的事件。由于每个线程都有机会得以运行，所以在球弹跳期间，当用户点击Close按钮时，事件调度线程将有机会关注到这个事件，并处理“关闭”这一动作。

这里用球弹跳代码作为示例，让大家对并发处理有一个视觉印象。通常，人们总会提防长时间的计算。这个计算很可能是某个大框架的一个组成部分，例如，GUI或web框架。无论何时框架调用自身的方法都会很快地返回一个异常。如果需要执行一个比较耗时的任务，应该使用独立的线程。

下面是在一个单独的线程中执行一个任务的简单过程：

1) 将任务代码移到实现了Runnable接口的类的run方法中。这个接口非常简单，只有一个方法：

```
public interface Runnable
```

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

可以如下所示实现一个类：

```
class MyRunnable implements Runnable  
{  
    public void run()  
    {  
        task code  
    }  
}
```

2) 创建一个类对象：

```
Runnable r = new MyRunnable();
```

3) 由Runnable创建一个Thread对象：

```
Thread t = new Thread(r);
```

4) 启动线程：

```
t.start();
```

要想将弹跳球代码放在一个独立的线程中，只需要实现一个类BallRunnable，然后，将动画代码放在run方法中，如同下面这段代码：

```
class BallRunnable implements Runnable  
{  
    ...  
    public void run()  
    {  
        try  
        {  
            for (int i = 1; i <= STEPS; i++)  
            {  
                ball.move(component.getBounds());  
                component.repaint();  
                Thread.sleep(DELAY);  
            }  
        }  
        catch (InterruptedException exception)  
        {  
        }  
    }  
    ...  
}
```

此外，需要捕获sleep方法可能抛出的异常InterruptedException。下一节将讨论这个异常。在一般情况下，线程在中断时被终止。因此，当发生InterruptedException异常时，run方法将结束执行。无论何时点击Start按钮，addBall方法都将启动一个新线程（见图14-2）：

```
Ball b = new Ball();  
panel.add(b);  
Runnable r = new BallRunnable(b, panel);  
Thread t = new Thread(r);  
t.start();
```

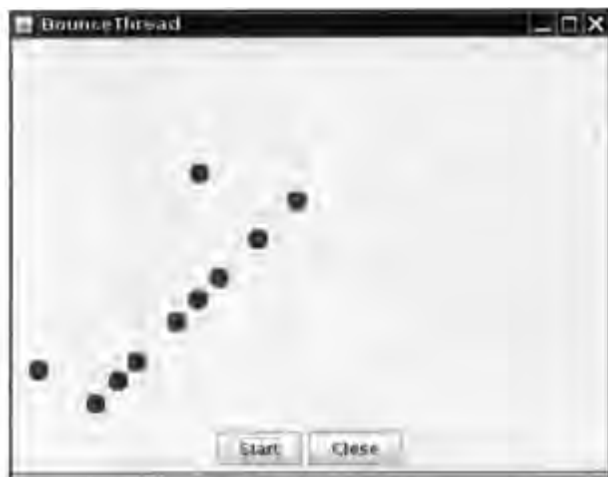


图14-2 运行多线程

这个问题已经讲得差不多了。现在应该知道如何并行运行多个任务了。本章其余部分将阐述如何控制线程之间的交互。

完整的代码在例14-4中。



注释：也可以通过构建一个Thread类的子类定义一个线程，如下所示：

```
class MyThread extends Thread
{
    public void run()
    {
        task code
    }
}
```

然后，构造一个子类的对象，并调用start方法。目前，这种方法已不再推荐。应该从运行机制上减少需要并行运行的任务数量。如果有很多任务，要为每个任务创建一个独立的线程所付出的代价太大了。可以使用线程池来解决这个问题，有关内容请参看第14.9节。



警告：不要调用Thread类或Runnable对象的run方法。直接调用run方法，只会执行同一个线程中的任务，而不会启动新线程。应该调用Thread.start方法。这个方法将创建一个执行run方法的新线程。

#### 例14-4 BounceThread.java

```
1. import java.awt.*;
2. import java.awt.event.*;
3. import javax.swing.*;
4.
5. /**
6.  * Shows animated bouncing balls.
7.  * @version 1.33 2007-05-17
8.  * @author Cay Horstmann
9.  */
10. public class BounceThread
```



```
11. {
12.     public static void main(String[] args)
13.     {
14.         EventQueue.invokeLater(new Runnable()
15.         {
16.             public void run()
17.             {
18.                 JFrame frame = new BounceFrame();
19.                 frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
20.                 frame.setVisible(true);
21.             }
22.         });
23.     }
24. }
25.
26. /**
27.  * A runnable that animates a bouncing ball.
28.  */
29. class BallRunnable implements Runnable
30. {
31.     /**
32.      * Constructs the runnable.
33.      * @aBall the ball to bounce
34.      * @aPanel the component in which the ball bounces
35.      */
36.     public BallRunnable(Ball aBall, Component aComponent)
37.     {
38.         ball = aBall;
39.         component = aComponent;
40.     }
41.
42.     public void run()
43.     {
44.         try
45.         {
46.             for (int i = 1; i <= STEPS; i++)
47.             {
48.                 ball.move(component.getBounds());
49.                 component.repaint();
50.                 Thread.sleep(DELAY);
51.             }
52.         }
53.         catch (InterruptedException e)
54.         {
55.         }
56.     }
57.
58.     private Ball ball;
59.     private Component component;
60.     public static final int STEPS = 1000;
61.     public static final int DELAY = 5;
62. }
63.
64. /**
65.  * The frame with panel and buttons.
66.  */
67. class BounceFrame extends JFrame
```

```

68. {
69.     /**
70.      * Constructs the frame with the component for showing the bouncing ball and Start and
71.      * Close buttons
72.      */
73.     public BounceFrame()
74.     {
75.         setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
76.         setTitle("BounceThread");
77.
78.         comp = new BallComponent();
79.         add(comp, BorderLayout.CENTER);
80.         JPanel buttonPanel = new JPanel();
81.         addButton(buttonPanel, "Start", new ActionListener()
82.             {
83.                 public void actionPerformed(ActionEvent event)
84.                 {
85.                     addBall();
86.                 }
87.             });
88.
89.         addButton(buttonPanel, "Close", new ActionListener()
90.             {
91.                 public void actionPerformed(ActionEvent event)
92.                 {
93.                     System.exit(0);
94.                 }
95.             });
96.         add(buttonPanel, BorderLayout.SOUTH);
97.     }
98.
99.     /**
100.      * Adds a button to a container.
101.      * @param c the container
102.      * @param title the button title
103.      * @param listener the action listener for the button
104.      */
105.     public void addButton(Container c, String title, ActionListener listener)
106.     {
107.         JButton button = new JButton(title);
108.         c.add(button);
109.         button.addActionListener(listener);
110.     }
111.
112.     /**
113.      * Adds a bouncing ball to the canvas and starts a thread to make it bounce
114.      */
115.     public void addBall()
116.     {
117.         Ball b = new Ball();
118.         comp.add(b);
119.         Runnable r = new BallRunnable(b, comp);
120.         Thread t = new Thread(r);
121.         t.start();
122.     }
123.
124.     private BallComponent comp;

```

```
125. public static final int DEFAULT_WIDTH = 450;
126. public static final int DEFAULT_HEIGHT = 350;
127. public static final int STEPS = 1000;
128. public static final int DELAY = 3;
129. }
```

#### **API** java.lang.Thread 1.0

- Thread(Runnable target)  
构造一个新线程，用于调用给定target的run ( ) 方法。
- void start()  
启动这个线程，将引发调用 run() 方法。这个方法将立即返回，并且新线程将并行运行。
- void run()  
调用关联Runnable 的run方法。

#### **API** java.lang.Runnable 1.0

- void run()  
必须覆盖这个方法，并在这个方法中提供所要执行的任务指令。

## 14.2 中断线程

当线程的run方法执行方法体中最后一条语句后，并经由执行return语句返回时，或者出现了在方法中没有捕获的异常时，线程将终止。在Java的早期版本中，还有一个stop方法，其他线程可以调用它终止线程。但是，这个方法现在已经被弃用了。稍后将讨论“为什么stop方法和suspend方法遭到弃用？”的缘由。

有一种可以强制线程终止的方法。然而，interrupt方法可以用来请求终止线程。

当对一个线程调用interrupt方法时，线程的中断状态将被置位。这是每一个线程都具有的boolean标志。每个线程都应该不时地检查这个标志，以判断线程是否被中断。

要想弄清中断状态是否被置位，首先调用静态的Thread.currentThread方法获得当前线程，然后调用isInterrupted method方法：

```
while (!Thread.currentThread().isInterrupted() && more work to do)
{
    do more work
}
```

但是，如果线程被阻塞，就无法检测中断状态。这是产生InterruptedException异常的地方。当在一个被阻塞的线程（调用sleep或wait）上调用interrupt方法时，阻塞调用将会被InterruptedException异常中断。（存在不能被中断的阻塞I/O调用，应该考虑选择可中断的调用。有关细节请参看卷II的第1章和第3章。）

没有任何语言方面的需求要求一个被中断的线程应该终止。中断一个线程不过是引起它的注意。被中断的线程可以决定如何响应中断。某些线程是如此重要以至于应该处理完异常后，继续执行，而不理会中断。但是，更普遍的情况是，线程将简单地将中断作为一个终止的请求。

这种线程的run方法具有如下形式：

```
public void run()
{
    try
    {
        . . .
        while (!Thread.currentThread().isInterrupted() && more work to do)
        {
            do more work
        }
    }
    catch (InterruptedException e)
    {
        // thread was interrupted during sleep or wait
    }
    finally
    {
        cleanup, if required
    }
    // exiting the run method terminates the thread
}
```

如果在每次工作迭代之后都调用sleep方法（或者其他可中断方法），isInterrupted检测既没有必要也没有用处。如果在中断状态被置位时调用sleep方法，它不会休眠。相反，它将清除这一状态并抛出InterruptedException。因此，如果你的循环调用sleep，不会检测中断状态。相反，要如下所示捕获InterruptedException异常：

```
public void run()
{
    try
    {
        . . .
        while (more work to do)
        {
            do more work
            Thread.sleep(delay);
        }
    }
    catch (InterruptedException e)
    {
        // thread was interrupted during sleep
    }
    finally
    {
        cleanup, if required
    }
    // exiting the run method terminates the thread
}
```



注释：有两个非常类似的方法，interrupted和isInterrupted。InterruptedException方法是一个静态方法，它检测当前的线程是否被中断。而且，调用interrupted方法会清除该线程的中断状态。另一方面，isInterrupted方法是一个实例方法，可用来检验是否有线程被中断。调用这个方法不会改变中断状态。

在很多发布的代码中会发现InterruptedException异常被抑制在很低的层次上，像这样：

```
void mySubTask()
{
    . . .
    try { sleep(delay); }
    catch (InterruptedException e) {} // DON'T IGNORE!
    . . .
}
```

不要这样做！如果不认为在catch子句中做这一处理有什么好处的话，仍然有两种合理的选择：

- 在catch子句中调用Thread.currentThread().interrupt()来设置中断状态。于是，调用者可以对其进行检测。

```
void mySubTask()
{
    . . .
    try { sleep(delay); }
    catch (InterruptedException e) { Thread().currentThread().interrupt(); }
    . . .
}
```

- 或者，更好的选择是，用throws InterruptedException标记你的方法，不采用try语句块捕获异常。于是，调用者（或者，最终的run方法）可以捕获这一异常。

```
void mySubTask() throws InterruptedException
{
    . . .
    sleep(delay);
    . . .
}
```

#### java.lang.Thread 1.0

- void interrupt()  
向线程发送中断请求。线程的中断状态将被设置为true。如果目前该线程被一个sleep调用阻塞，那么，InterruptedException异常被抛出。
- static boolean interrupted()  
测试当前线程（即正在执行这一命令的线程）是否被中断。注意，这是一个静态方法。这一调用会产生副作用——它将当前线程的中断状态重置为false。
- boolean isInterrupted()  
测试线程是否被终止。不像静态的中断方法，这一调用不改变线程的中断状态。
- static Thread currentThread()  
返回代表当前执行线程的 Thread 对象。

### 14.3 线程状态

线程可以有如下6种状态：

- New ( 新生 )
- Runnable ( 可运行 )
- Blocked ( 被阻塞 )
- Waiting ( 等待 )
- Timed waiting ( 计时等待 )
- Terminated ( 被终止 )

下一节对每一种状态进行解释。

要确定一个线程的当前状态，可调用getState方法。

### 14.3.1 新生线程

当用new操作符创建一个新线程时，如new Thread(r)，该线程还没有开始运行。这意味着它的状态是new。当一个线程处于新生状态时，程序还没有开始运行线程中的代码。在线程运行之前还有一些簿记工作要做。

### 14.3.2 可运行线程

一旦调用start方法，线程处于runnable状态。一个可运行的线程可能正在运行也可能没有运行，这取决于操作系统给线程提供运行的时间。(Java的规范说明没有将它作为一个单独状态。一个正在运行中的线程仍然处于可运行状态。)

一旦一个线程开始运行，它不必始终保持运行。事实上，运行中的线程被中断，目的是为了其他线程获得运行机会。线程调度的细节依赖于操作系统提供的服务。抢占式调度系统给每一个可运行线程一个时间片来执行任务。当时间片用完，操作系统剥夺该线程的运行权，并给另一个线程运行机会(见图14-4)。当选择下一个线程时，操作系统考虑线程的优先级——更多的内容见第14.4.1节。

现在所有的桌面以及服务器操作系统都使用抢占式调度。但是，像手机这样的小型设备可能使用协作式调度。在这样的设备中，一个线程只有在调用yield方法、或者被阻塞或等待时，线程才失去控制权。

在具有多个处理器的机器上，每一个处理器运行一个线程，可以有多个线程并行运行。当然，如果线程的数目多于处理器的数目，调度器依然采用时间片机制。

记住，在任何给定时刻，一个可运行的线程可能正在运行也可能没有运行(这就是为什么将这个状态称为可运行而不是运行)。

### 14.3.3 被阻塞线程和等待线程

当线程处于被阻塞或等待状态时，它暂时不活动。它不运行任何代码且消耗最少的资源。直到线程调度器重新激活它。细节取决于它是怎样达到非活动状态的。

- 当一个线程试图获取一个内部的对象锁(而不是java.util.concurrent库中的锁)，而该锁被其他线程持有，则该线程进入阻塞状态(我们在第14.5.3节讨论java.util.concurrent锁，在第14.5.5节讨论内部对象锁)。当所有其他线程释放该锁，并且线程调度器允许本线程持有它的时候，该线程将变成非阻塞状态。

- 当线程等待另一个线程通知调度器一个条件时，它自己进入等待状态。我们在第14.5.4节来讨论条件。在调用Object.wait方法或Thread.join方法，或者是等待java.util.concurrent库中的Lock或Condition时，就会出现这种情况。实际上，被阻塞状态与等待状态是有很大的不同的。
- 有几个方法有一个超时参数。调用它们导致线程进入计时等待（timed waiting）状态。这一状态将一直保持到超时期满或者接收到适当的通知。带有超时参数的方法有Thread.sleep和Object.wait、Thread.join、Lock.tryLock以及Condition.await的计时版。

图14-3展示了线程可以具有的状态以及从一个状态到另一个状态可能的转换。当一个线程被阻塞或等待时（或终止时），另一个线程被调度为运行状态。当一个线程被重新激活（例如，因为超时期满或成功地获得了一个锁），调度器检查它是否具有比当前运行线程更高的优先级。如果是这样，调度器从当前运行线程中挑选一个，剥夺其运行权，选择一个新的线程运行。

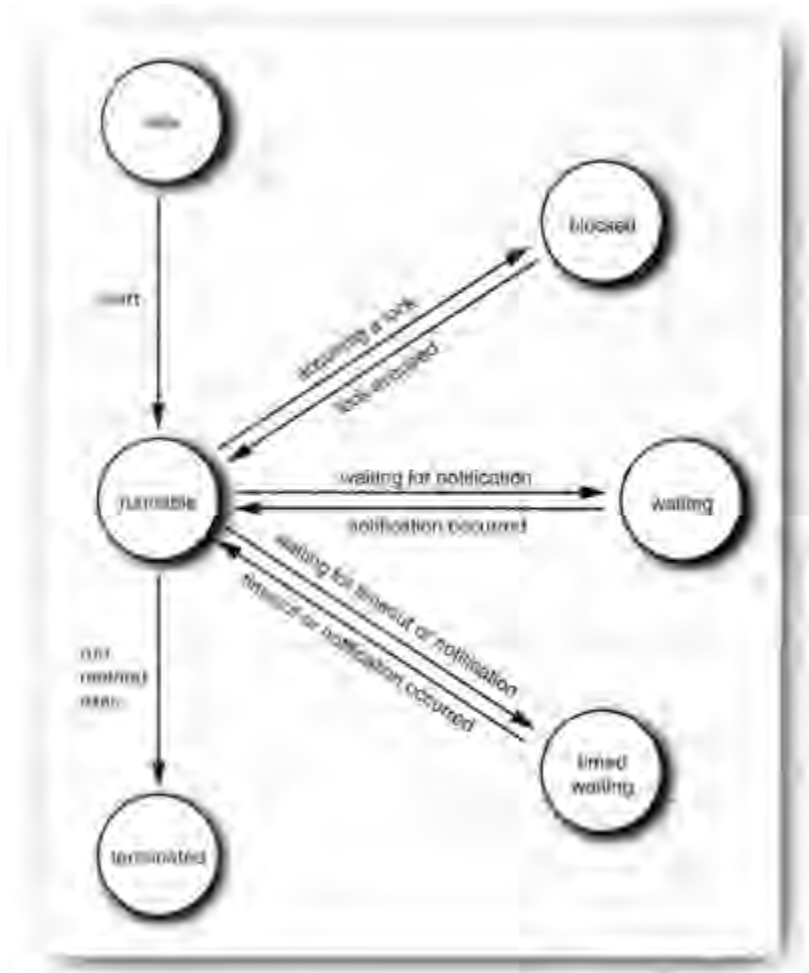


图14-3 线程状态

#### 14.3.4 被终止的线程

线程因如下两个原因之一而被终止：

- 因为run方法正常退出而自然死亡。
- 因为一个没有捕获的异常终止了run方法而意外死亡。

特别是，可以调用线程的stop方法杀死一个线程。该方法抛出ThreadDeath 错误对象，由此杀死线程。但是，stop方法已过时，不要在自己的代码中调用它。

#### java.lang.Thread 1.0

- void join()  
等待终止指定的线程。
- void join(long millis)  
等待指定的线程死亡或者经过指定的毫秒数。
- Thread.State getState() 5.0  
得到这一线程的状态；NEW、RUNNABLE、BLOCKED、WAITING、TIMED\_WAITING 或TERMINATED之一。
- void stop()  
停止该线程。这一方法已过时。
- void suspend()  
暂停这一线程的执行。这一方法已过时。
- void resume()  
恢复线程。这一方法仅仅在调用suspend()之后调用。这一方法已过时。

## 14.4 线程属性

下面将讨论线程的各种属性，其中包括：线程优先级、守护线程、线程组以及处理未捕获异常的处理程序。

### 14.4.1 线程优先级

在Java程序设计语言中，每一个线程有一个优先级。默认情况下，一个线程继承它的父线程的优先级。可以用setPriority方法提高或降低任何一个线程的优先级。可以将优先级设置为在MIN\_PRIORITY（在Thread类中定义为1）与MAX\_PRIORITY（定义为10）之间的任何值。NORM\_PRIORITY被定义为5。

每当线程调度器有机会选择新线程时，它首先选择具有较高优先级的线程。但是，线程优先级是高度依赖于系统的。当虚拟机依赖于宿主机平台的线程实现机制时，Java线程的优先级被映射到宿主机平台的优先级上，优先级个数也许更多，也许更少。

例如，Windows有7个优先级别。一些Java优先级将映射到相同的操作系统优先级。在Sun为Linux提供的Java虚拟机，线程的优先级被忽略——所有线程具有相同的优先级。

初级程序员常常过度使用线程优先级。为优先级而烦恼是事出有因的。不要将程序构建为功能的正确性依赖于优先级。





警告：如果确实要使用优先级，应该避免初学者常犯的一个错误。如果有几个高优先级的线程没有进入非活动状态，低优先级的线程可能永远也不能执行。每当调度器决定运行一个新线程时，首先会在具有高优先级的线程中进行选择，尽管这样会使低优先级的线程完全饿死。



java.lang.Thread 1.0

- void setPriority(int newPriority)  
设置线程的优先级。优先级必须在 Thread.MIN\_PRIORITY 与 Thread.MAX\_PRIORITY 之间。一般使用 Thread.NORM\_PRIORITY 优先级。
- static int MIN\_PRIORITY  
线程的最小优先级。最小优先级的值为1。
- static int NORM\_PRIORITY  
线程的默认优先级。缺省优先级为5。
- static int MAX\_PRIORITY  
线程的最高优先级。最高优先级的值为10。
- static void yield()  
导致当前执行线程处于让步状态。如果有其他的可运行线程具有至少与此线程同样高的优先级，那么这些线程接下来会被调度。注意，这是一个静态方法。

#### 14.4.2 守护线程

可以通过调用

```
t.setDaemon(true);
```

将线程转换为守护线程（daemon thread）。这样一个线程没有什么神奇。守护线程的惟一用途是为其他线程提供服务。计时线程就是一个例子，它定时地发送“时间嘀嗒”信号给其他线程或清空过时的高速缓存项的线程。当只剩下守护线程时，虚拟机就退出了，由于如果只剩下守护线程，就没必要继续运行程序了。

守护线程有时会被初学者错误地使用，他们不打算考虑关机（shutdown）动作。但是，这是很危险的。守护线程应该永远不去访问固有资源，如文件、数据库，因为它会在任何时候甚至在一个操作的中间发生中断。



java.lang.Thread 1.0

- void setDaemon(boolean isDaemon)  
标识该线程为守护线程或用户线程。这一方法必须在线程启动之前调用。

#### 14.4.3 未捕获异常处理器

线程的run方法不能抛出任何被检测的异常，但是，不被检测的异常会导致线程终止。在这种情况下，线程就死亡了。

但是，不需要任何catch子句来处理可以被传播的异常。相反，就在线程死亡之前，异常被传递到一个用于未捕获异常的处理器。

该处理器必须属于一个实现Thread.UncaughtExceptionHandler接口的类。这个接口只有一个方法。

```
void uncaughtException(Thread t, Throwable e)
```

从Java SE 5.0起，可以用setUncaughtExceptionHandler方法为任何线程安装一个处理器。也可以用Thread类的静态方法setDefaultUncaughtExceptionHandler为所有线程安装一个默认的处理器。替换处理器可以使用日志API发送未捕获异常的报告到日志文件。

如果不安装默认的处理器，默认的处理器为空。但是，如果不为独立的线程安装处理器，此时的处理器就是该线程的ThreadGroup对象。



注释：线程组是一个可以统一管理的线程集合。默认情况下，创建的所有线程属于相同的线程组，但是，也可能会建立其他的组。从Java SE 5.0起引入了更好的特性用于线程集合的操作。不要在自己的程序中使用线程组。

ThreadGroup类实现Thread.UncaughtExceptionHandler接口。它的uncaughtException方法做如下操作：

- 1) 如果该线程组有父线程组，那么父线程组的uncaughtException方法被调用。
  - 2) 否则，如果Thread.getDefaultExceptionHandler方法返回一个非空的处理器，则调用该处理器。
  - 3) 否则，如果Throwable是ThreadDeath的一个实例，什么都不做。
  - 4) 否则，线程的名字以及Throwable的栈踪迹被输出到System.err上。
- 这是你在程序中肯定看到过许多次的栈踪迹。



java.lang.Thread 1.0

- static void setDefaultUncaughtExceptionHandler(Thread.UncaughtExceptionHandler handler) 5.0
- static Thread.UncaughtExceptionHandler getDefaultUncaughtExceptionHandler() 5.0  
设置或获取未捕获异常的默认处理器。
- void setUncaughtExceptionHandler(Thread.UncaughtExceptionHandler handler) 5.0
- Thread.UncaughtExceptionHandler getUncaughtExceptionHandler() 5.0  
设置或获取未捕获异常的处理器。如果没有安装处理器，则将线程组对象作为处理器。



java.lang.Thread.UncaughtExceptionHandler 5.0

- void uncaughtException(Thread t, Throwable e)  
当一个线程因未捕获异常而终止，按规定要将客户报告记录到日志中。
- 参数：t                      由于未捕获异常而终止的线程  
         e                      未捕获的异常对象

**API** java.lang.ThreadGroup 1.0

- void uncaughtException(Thread t, Throwable e)

如果有父线程组，调用父线程组的这一方法；或者，如果Thread类有默认处理器，调用该处理器，否则，输出栈踪迹到标准错误流上（但是，如果e是一个ThreadDeath对象，栈踪迹是被禁用的。ThreadDeath对象由stop方法产生，而该方法已经过时）。

## 14.5 同步

在大多数实际的多线程应用中，两个或两个以上的线程需要共享对同一数据的存取。如果两个线程存取相同的对象，并且每一个线程都调用了修改该对象状态的方法，将会发生什么呢？可以想像，线程彼此踩了对方的脚。根据各线程访问数据的次序，可能会产生讹误的对象。这样一个情况通常称为竞争条件（race condition）。

### 14.5.1 竞争条件的一个例子

为了避免多线程引起的对共享数据的讹误，必须学习如何同步存取。在本节中，你会看到如果没有使用同步会发生什么。在下一节中，将会看到如何同步数据存取。

在下面的测试程序中，模拟一个有若干账户的银行。随机地生成在这些账户之间转移钱款的交易。每一个账户有一个线程。每一笔交易中，会从线程所服务的账户中随机转移一定数目的钱款到另一个随机账户。

模拟代码非常直观。我们有具有transfer方法的Bank类。该方法从一个账户转移一定数目的钱款到另一个账户（还没有考虑负的账户余额）。如下是Bank类的transfer方法的代码。

```
public void transfer(int from, int to, double amount)
// CAUTION: unsafe when called from multiple threads
{
    System.out.print(Thread.currentThread());
    accounts[from] -= amount;
    System.out.printf(" %10.2f from %d to %d", amount, from, to);
    accounts[to] += amount;
    System.out.printf(" Total Balance: %10.2f\n", getTotalBalance());
}
```

这里是TransferRunnable类的代码。它的run方法不断地从一个固定的银行账户取出钱款。在每一次迭代中，run方法随机选择一个目标账户和一个随机账户，调用bank对象的transfer方法，然后睡眠。

```
class TransferRunnable implements Runnable
{
    ...
    public void run()
    {
        try
        {
            int toAccount = (int) (bank.size() * Math.random());
            double amount = maxAmount * Math.random();
            bank.transfer(fromAccount, toAccount, amount);
        }
    }
}
```

```

        Thread.sleep((int) (DELAY * Math.random()));
    }
    catch (InterruptedException e) {}
}
}

```

当这个模拟程序运行时，不清楚在某一时刻某一银行账户中有多少钱。但是，知道所有账户的总金额应该保持不变，因为所做的一切不过是从一个账户转移钱款到另一个账户。

在每一次交易的结尾，transfer方法重新计算总值并打印出来。

本程序永远不会结束。只能按 CTRL+C 来终止这个程序。

下面是典型的输出：

```

...
Thread[Thread-11,5,main] 588.48 from 11 to 44 Total Balance: 100000.00
Thread[Thread-12,5,main] 976.11 from 12 to 22 Total Balance: 100000.00
Thread[Thread-14,5,main] 521.51 from 14 to 22 Total Balance: 100000.00
Thread[Thread-13,5,main] 359.89 from 13 to 81 Total Balance: 100000.00
...
Thread[Thread-36,5,main] 401.71 from 36 to 73 Total Balance: 99291.06
Thread[Thread-35,5,main] 691.46 from 35 to 77 Total Balance: 99291.06
Thread[Thread-37,5,main] 78.64 from 37 to 3 Total Balance: 99291.06
Thread[Thread-34,5,main] 197.11 from 34 to 69 Total Balance: 99291.06
Thread[Thread-36,5,main] 85.96 from 36 to 4 Total Balance: 99291.06
...
Thread[Thread-4,5,main]Thread[Thread-33,5,main] 7.31 from 31 to 32 Total Balance:
99979.24
    627.50 from 4 to 5 Total Balance: 99979.24
...

```

正如前面所示，出现了错误。在最初的交易中，银行的余额保持在\$100 000，这是正确的，因为共100个账户，每个账户\$1 000。但是，过一段时间，余额总量有轻微的变化。当运行这个程序的时候，会发现有时很快就出错了，有时很长的时间后余额发生混乱。这样的状态不会带来信任感，人们很可能不愿意将辛苦挣来得钱存到这个银行。

例14-5到例14-7中的程序提供了完全的源代码。看看是否可以从代码中找出问题。下一节将解说其中神秘。

#### 例14-5 UnsynchBankTest.java

```

1. /**
2.  * This program shows data corruption when multiple threads access a data structure.
3.  * @version 1.30 2004-08-01
4.  * @author Cay Horstmann
5.  */
6. public class UnsynchBankTest
7. {
8.     public static void main(String[] args)
9.     {
10.         Bank b = new Bank(NACCOUNTS, INITIAL_BALANCE);
11.         int i;
12.         for (i = 0; i < NACCOUNTS; i++)
13.         {
14.             TransferRunnable r = new TransferRunnable(b, i, INITIAL_BALANCE);

```

```
15.         Thread t = new Thread(r);
16.         t.start();
17.     }
18. }
19.
20. public static final int NACCOUNTS = 100;
21. public static final double INITIAL_BALANCE = 1000;
22. }
```

**例14-6 Bank.java**

```
1. /**
2.  * A bank with a number of bank accounts.
3.  * @version 1.30 2004-08-01
4.  * @author Cay Horstmann
5.  */
6. public class Bank
7. {
8.     /**
9.      * Constructs the bank.
10.     * @param n the number of accounts
11.     * @param initialBalance the initial balance for each account
12.     */
13.     public Bank(int n, double initialBalance)
14.     {
15.         accounts = new double[n];
16.         for (int i = 0; i < accounts.length; i++)
17.             accounts[i] = initialBalance;
18.     }
19.
20.     /**
21.      * Transfers money from one account to another.
22.      * @param from the account to transfer from
23.      * @param to the account to transfer to
24.      * @param amount the amount to transfer
25.      */
26.     public void transfer(int from, int to, double amount)
27.     {
28.         if (accounts[from] < amount) return;
29.         System.out.print(Thread.currentThread());
30.         accounts[from] -= amount;
31.         System.out.printf(" %10.2f from %d to %d", amount, from, to);
32.         accounts[to] += amount;
33.         System.out.printf(" Total Balance: %10.2f\n", getTotalBalance());
34.     }
35.
36.     /**
37.      * Gets the sum of all account balances.
38.      * @return the total balance
39.      */
40.     public double getTotalBalance()
41.     {
42.         double sum = 0;
43.
44.         for (double a : accounts)
45.             sum += a;
```

```

46.
47.     return sum;
48. }
49.
50. /**
51.  * Gets the number of accounts in the bank.
52.  * @return the number of accounts
53.  */
54. public int size()
55. {
56.     return accounts.length;
57. }
58.
59. private final double[] accounts;
60. }

```

#### 例14-7 TransferRunnable.java

```

1. /**
2.  * A runnable that transfers money from an account to other accounts in a bank.
3.  * @version 1.30 2004-08-01
4.  * @author Cay Horstmann
5.  */
6. public class TransferRunnable implements Runnable
7. {
8.     /**
9.      * Constructs a transfer runnable.
10.     * @param b the bank between whose account money is transferred
11.     * @param from the account to transfer money from
12.     * @param max the maximum amount of money in each transfer
13.     */
14. public TransferRunnable(Bank b, int from, double max)
15. {
16.     bank = b;
17.     fromAccount = from;
18.     maxAmount = max;
19. }
20.
21. public void run()
22. {
23.     try
24.     {
25.         while (true)
26.         {
27.             int toAccount = (int) (bank.size() * Math.random());
28.             double amount = maxAmount * Math.random();
29.             bank.transfer(fromAccount, toAccount, amount);
30.             Thread.sleep((int) (DELAY * Math.random()));
31.         }
32.     }
33.     catch (InterruptedException e)
34.     {
35.     }
36. }
37.
38. private Bank bank;

```

```
39. private int fromAccount;  
40. private double maxAmount;  
41. private int DELAY = 10;  
42. }
```

### 14.5.2 详解竞争条件

上一节中运行了一个程序，其中有几个线程更新银行账户余额。一段时间之后，错误不知不觉地出现了，总额要么增加，要么变少。当两个线程试图同时更新同一个账户的时候，这个问题就出现了。假定两个线程同时执行指令

```
accounts[to] += amount;
```

问题在于这不是原子操作。该指令可能被处理如下：

- 1) 将accounts[to]加载到寄存器。
- 2) 增加amount。
- 3) 将结果写回accounts[to]。

现在，假定第1个线程执行步骤1和2，然后，它被剥夺了运行权。假定第2个线程被唤醒并修改了accounts数组中的同一项。然后，第1个线程被唤醒并完成其第3步。

这样，这一动作擦去了第二个线程所做的更新。于是，总金额不再正确。（见图14-4。）

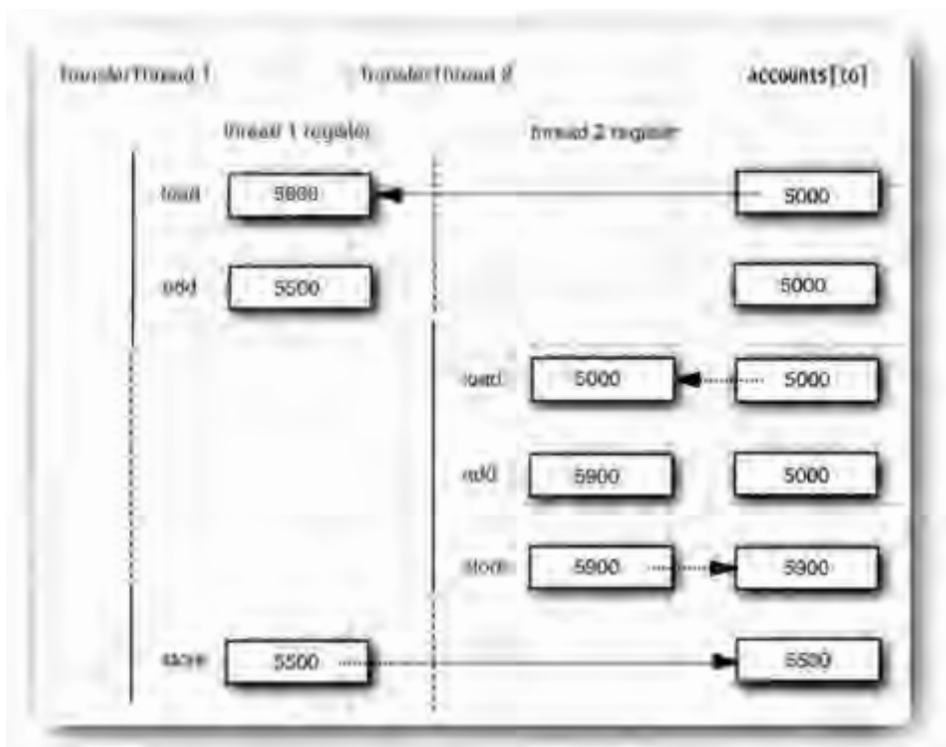


图14-4 同时被两个线程访问

我们的测试程序检测到这一讹误。（当然，如果线程在运行这一测试时被中断，也有可能不会出现失败警告！）



注释：可以具体看一下执行我们的类中的每一个语句的虚拟机的字节码。运行命令

```
javap -c -v Bank
```

对Bank.class文件进行反编译。例如，代码行

```
accounts[to] += amount;
```

被转换为下面的字节码：

```
aload_0
getfield      #2; //Field accounts:[D
iload_2
dup2
daload
dload_3
dadd
dastore
```

这些代码的含义无关紧要。重要的是增值命令是由几条指令组成的，执行它们的线程可以在任何一条指令点上被中断。

出现这一讹误的可能性有多大呢？这里通过将打印语句和更新余额的语句交织在一起执行，增加了发生这种情况的机会。

如果删除打印语句，讹误的风险会降低一点，因为每个线程在再次睡眠之前所做的工作很少，调度器在计算过程中剥夺线程的运行权可能性很小。但是，讹误的风险并没有完全消失。如果在负载很重的机器上运行许多线程，那么，即使删除了打印语句，程序依然会出错。这种错误可能会几分钟、几小时或几天出现一次。坦白地说，对程序员而言，很少有比无规律出现错误更糟的事情了。

真正的问题是transfer方法的执行过程中可能会被中断。如果能够确保线程在失去控制之前方法运行完成，那么银行账户对象的状态永远不会出现讹误。

### 14.5.3 锁对象

从 Java SE 5.0开始，有两种机制防止代码块受并发访问的干扰。Java语言提供一个synchronized关键字达到这一目的，并且Java SE 5.0引入了ReentrantLock类。synchronized关键字自动提供一个锁以及相关的“条件”，对于大多数需要显式锁的情况，这是很便利的。但是，我们相信在读者分别阅读了锁和条件的内容之后，理解synchronized关键字是很轻松的事情。java.util.concurrent框架为这些基础机制提供独立的类，在此以及第14.5.4节加以解释这个内容。读者理解了这些构建块之后，将讨论第14.5.5节。

用ReentrantLock保护代码块的基本结构如下：

```
myLock.lock(); // a ReentrantLock object
try
{
    critical section
}
finally
{
    myLock.unlock(); // make sure the lock is unlocked even if an exception is thrown
}
```



这一结构确保任何时刻只有一个线程进入临界区。一旦一个线程封锁了锁对象，其他任何线程都无法通过lock语句。当其他线程调用lock时，它们被阻塞，直到第一个线程释放锁对象。



**警告：**把解锁操作括在finally子句之内是至关重要的。如果在临界区的代码抛出异常，锁必须被释放。否则，其他线程将永远阻塞。

让我们使用一个锁来保护Bank类的 transfer 方法。

```
public class Bank
{
    public void transfer(int from, int to, int amount)
    {
        bankLock.lock();
        try
        {
            System.out.print(Thread.currentThread());
            accounts[from] -= amount;
            System.out.printf(" %10.2f from %d to %d", amount, from, to);
            accounts[to] += amount;
            System.out.printf(" Total Balance: %10.2f\n", getTotalBalance());
        }
        finally
        {
            bankLock.unlock();
        }
    }
    . . .
    private Lock bankLock = new ReentrantLock(); // ReentrantLock implements the Lock interface
}
```

假定一个线程调用transfer，在执行结束前被剥夺了运行权。假定第二个线程也调用transfer，由于第二个线程不能获得锁，将在调用lock方法时被阻塞。它必须等待第一个线程完成transfer方法的执行之后才能再度被激活。当第一个线程释放锁时，那么第二个线程才能开始运行（见图14-5）。

尝试一下。添加加锁代码到transfer方法并且再次运行程序。你可以永远运行它，而银行的余额不会出现讹误。

注意每一个Bank对象有自己的ReentrantLock对象。如果两个线程试图访问同一个Bank对象，那么锁以串行方式提供服务。但是，如果两个线程访问不同的Bank对象，每一个线程得到不同的锁对象，两个线程都不会发生阻塞。本该如此，因为线程在操纵不同的Bank实例的时候，线程之间不会相互影响。

锁是可重入的，因为线程可以重复地获得已经持有的锁。锁保持一个持有计数（hold count）来跟踪对lock方法的嵌套调用。线程在每一次调用lock都要调用unlock来释放锁。由于这一特性，被一个锁保护的代码可以调用另一个使用相同的锁的方法。

例如，transfer方法调用getTotalBalance方法，这也会封锁bankLock对象，此时bankLock对象的持有计数为2。当getTotalBalance方法退出的时候，持有计数变回1。当transfer方法退出的时候，持有计数变为0。线程释放锁。

通常，可能想要保护需若干个操作来更新或检查共享对象的代码块。要确保这些操作完成

后，另一个线程才能使用相同对象。

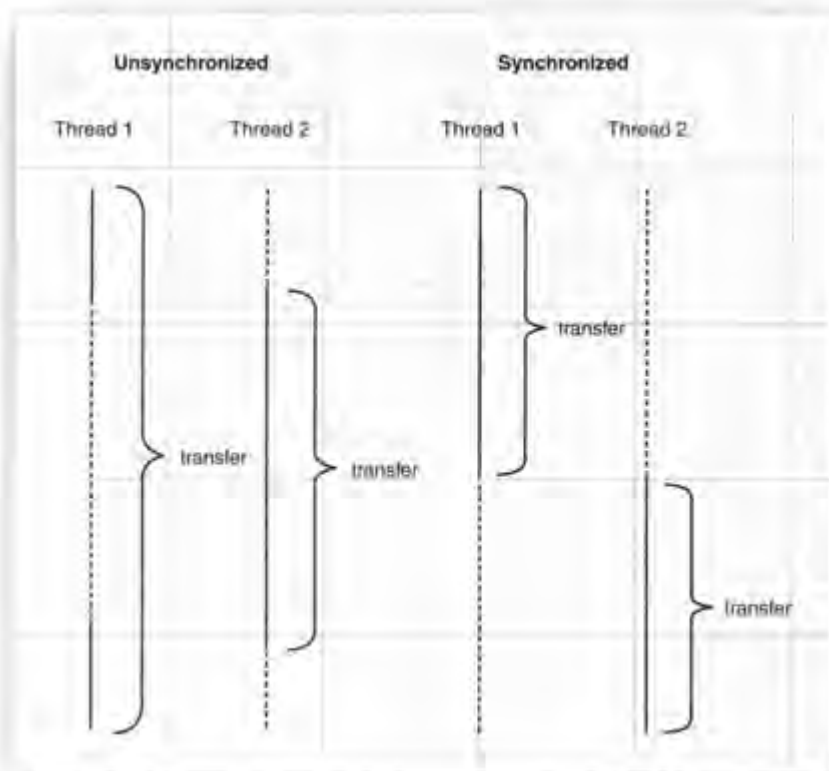


图14-5 非同步线程与同步线程的比较


**X** 警告：要留心临界区中的代码，不要因为异常的抛出而跳出了临界区。如果在临界区代码结束之前抛出了异常，finally子句将释放锁，但会使对象可能处于一种受损状态。

**API** java.util.concurrent.locks.Lock 5.0

- void lock()  
获取这个锁；如果锁同时被另一个线程拥有则发生阻塞。
- void unlock()  
释放这个锁。

**API** java.util.concurrent.locks.ReentrantLock 5.0

- ReentrantLock()  
构建一个可以被用来保护临界区的可重入锁。
- ReentrantLock(boolean fair)  
构建一个带有公平策略的锁。一个公平锁偏爱等待时间最长的线程。但是，这一公平的保证将大大降低性能。所以，默认情况下，锁没有被强制为公平的。

 警告：听起来公平锁更合理一些，但是使用公平锁比使用常规锁要慢很多。只有当你确实了解自己要做什么并且对于你要解决的问题有一个特定的理由必须使用公平锁的时候，才可以使用公平锁。即使使用公平锁，也无法确保线程调度器是公平的。如果线程调度器选择忽略一个线程，而该线程为了这个锁已经等待了很长时间，那么就没有机会公平地处理这个锁了。

#### 14.5.4 条件对象

通常，线程进入临界区，却发现在某一条件满足之后它才能执行。要使用一个条件对象来管理那些已经获得了一个锁但是却不能做有用工作的线程。在这一节里，我们介绍Java库中条件对象的实现。（由于历史的原因，条件对象经常被称为条件变量（conditional variable）。）

现在来细化银行的模拟程序。我们避免选择没有足够资金的账户作为转出账户。注意不能使用下面这样的代码：

```
if (bank.getBalance(from) >= amount)
    bank.transfer(from, to, amount);
```

当前线程完全有可能在成功地完成测试，且在调用transfer方法之前将被中断。

```
if (bank.getBalance(from) >= amount)
    // thread might be deactivated at this point
    bank.transfer(from, to, amount);
```

在线程再次运行前，账户余额可能已经低于提款金额。必须确保没有其他线程在本检查余额与转账活动之间修改余额。通过使用锁来保护检查与转账动作来做到这一点：

```
public void transfer(int from, int to, int amount)
{
    bankLock.lock();
    try
    {
        while (accounts[from] < amount)
        {
            // wait
            ...
        }
        // transfer funds
        ...
    }
    finally
    {
        bankLock.unlock();
    }
}
```

现在，当账户中没有足够的余额时，应该做什么呢？等待直到另一个线程向账户中注入了资金。但是，这一线程刚刚获得了对bankLock的排它性访问，因此别的线程没有进行存款操作的机会。这就是为什么我们需要条件对象的原因。

一个锁对象可以有一个或多个相关的条件对象。你可以用newCondition方法获得一个条件对象。习惯上给每一个条件对象命名为可以反映它所表达的条件的名字。例如，在此设置一个

条件对象来表达“余额充足”条件。

```
class Bank
{
    public Bank()
    {
        . . .
        sufficientFunds = bankLock.newCondition();
    }
    . . .
    private Condition sufficientFunds;
}
```

如果transfer方法发现余额不足，它调用

```
sufficientFunds.await();
```

当前线程现在被阻塞了，并放弃了锁。我们希望这样可以使得另一个线程可以进行增加账户余额的操作。

等待获得锁的线程和调用await方法的线程存在本质上的不同。一旦一个线程调用await方法，它进入该条件的等待集。当锁可用时，该线程不能马上解除阻塞。相反，它处于阻塞状态，直到另一个线程调用同一条件上的signalAll方法时为止。当另一个线程转账时，它应该调用

```
sufficientFunds.signalAll();
```

这一调用重新激活因为这一条件而等待的所有线程。当这些线程从等待集中移出时，它们再次成为可运行的，调度器将再次激活它们。同时，它们将试图重新进入该对象。一旦锁成为可用的，它们中的某个将从await调用返回，获得该锁并从被阻塞的地方继续执行。

此时，线程应该再次测试该条件。由于无法确保该条件被满足——signalAll方法仅仅是通知正在等待的线程：此时有可能已经满足条件，值得再次去检测该条件。



注释：通常，对await的调用应该在如下形式的循环体中

```
while (!(ok to proceed))
    condition.await();
```

至关重要的是最终需要某个其他线程调用signalAll方法。当一个线程调用await时，它没有办法重新激活自身。它寄希望于其他线程。如果没有其他线程来重新激活等待的线程，它就永远不再运行了。这将导致令人不快的死锁（deadlock）现象。如果所有其他线程被阻塞，最后一个活动线程在解除其他线程的阻塞状态之前就调用await方法，那么它也被阻塞。没有任何线程可以解除其他线程的阻塞，那么该程序就挂起了。


应该何时调用signalAll呢？经验上讲，在对象的状态有利于等待线程的方向改变时调用signalAll。例如，当一个账户余额发生改变时，等待的线程会应该有机会检查余额。在例子中，当完成了转账时，调用signalAll方法。

```
public void transfer(int from, int to, int amount)
{
    bankLock.lock();
    try
    {
        while (accounts[from] < amount)
```

```
        sufficientFunds.await();
        // transfer funds
        ...
        sufficientFunds.signalAll();
    }
    finally
    {
        bankLock.unlock();
    }
}
```

注意调用`signalAll`不会立即激活一个等待线程。它仅仅解除等待线程的阻塞，以便这些线程可以在当前线程退出同步方法之后，通过竞争实现对对象的访问。

另一个方法`signal`，则是随机解除等待集中某个线程的阻塞状态。这比解除所有线程的阻塞更加有效，但也存在危险。如果随机选择的线程发现自己仍然不能运行，那么它再次被阻塞。如果没有其他线程再次调用`signal`，那么系统就死锁了。

 **警告：**当一个线程拥有某个条件的锁时，它仅仅可以在该条件上调用`await`、`signalAll`或`signal`方法。

如果你运行例14-8中的程序，会注意到没有出现任何错误。总余额永远是\$100 000。没有任何账户曾出现负的余额（但是，你还是需要按下CTRL+C键来终止程序）。你可能还注意到这个程序运行起来稍微有些慢——这是为同步机制中的簿记操作所付出的代价。

实际上，正确地使用条件是富有挑战性的。在开始实现自己的条件对象之前，应该考虑使用“同步器”中描述的结构。

#### 例14-8 Bank.java

```
1. import java.util.concurrent.locks.*;
2.
3. /**
4.  * A bank with a number of bank accounts that uses locks for serializing access.
5.  * @version 1.30 2004-08-01
6.  * @author Cay Horstmann
7.  */
8. public class Bank
9. {
10.     /**
11.      * Constructs the bank.
12.      * @param n the number of accounts
13.      * @param initialBalance the initial balance for each account
14.      */
15.     public Bank(int n, double initialBalance)
16.     {
17.         accounts = new double[n];
18.         for (int i = 0; i < accounts.length; i++)
19.             accounts[i] = initialBalance;
20.         bankLock = new ReentrantLock();
21.         sufficientFunds = bankLock.newCondition();
22.     }
23.
24.     /**
```

```
25.  * Transfers money from one account to another.
26.  * @param from the account to transfer from
27.  * @param to the account to transfer to
28.  * @param amount the amount to transfer
29.  */
30.  public void transfer(int from, int to, double amount) throws InterruptedException
31.  {
32.      bankLock.lock();
33.      try
34.      {
35.          while (accounts[from] < amount)
36.              sufficientFunds.await();
37.          System.out.print(Thread.currentThread());
38.          accounts[from] -= amount;
39.          System.out.printf(" %10.2f from %d to %d", amount, from, to);
40.          accounts[to] += amount;
41.          System.out.printf(" Total Balance: %10.2f\n", getTotalBalance());
42.          sufficientFunds.signalAll();
43.      }
44.      finally
45.      {
46.          bankLock.unlock();
47.      }
48.  }
49.
50.  /**
51.   * Gets the sum of all account balances.
52.   * @return the total balance
53.   */
54.  public double getTotalBalance()
55.  {
56.      bankLock.lock();
57.      try
58.      {
59.          double sum = 0;
60.
61.          for (double a : accounts)
62.              sum += a;
63.
64.          return sum;
65.      }
66.      finally
67.      {
68.          bankLock.unlock();
69.      }
70.  }
71.
72.  /**
73.   * Gets the number of accounts in the bank.
74.   * @return the number of accounts
75.   */
76.  public int size()
77.  {
78.      return accounts.length;
79.  }
80.
```

```
81. private final double[] accounts;
82. private Lock bankLock;
83. private Condition sufficientFunds;
84. }
```

#### **API** java.util.concurrent.locks.Lock 5.0

- `Condition newCondition()`  
返回一个与该锁相关的条件对象。

#### **API** java.util.concurrent.locks.Condition 5.0

- `void await()`  
将该线程放到条件的等待集中。
- `void signalAll()`  
解除该条件的等待集中的所有线程的阻塞状态。
- `void signal()`  
从该条件的等待集中随机地选择一个线程，解除其阻塞状态。

#### 14.5.5 synchronized关键字

在前面一节中，介绍了如何使用Lock和Condition对象。在进一步深入之前，总结一下有关锁和条件的关键之处：

- 锁用来保护代码片段，任何时刻只能有一个线程执行被保护的代码。
- 锁可以管理试图进入被保护代码段的线程。
- 锁可以拥有一个或多个相关的条件对象。
- 每个条件对象管理那些已经进入被保护的代码段但还不能运行的线程。

Lock和Condition接口被添加到Java SE 5.0中，这也向程序设计人员提供了高度的封锁控制。然而，大多数情况下，并不需要那样的控制，并且可以使用一种嵌入到Java语言内部的机制。从1.0版开始，Java中的每一个对象都有一个内部锁。如果一个方法用synchronized 关键字声明，那么对象的锁将保护整个方法。也就是说，要调用该方法，线程必须获得内部的对象锁。

换句话说，

```
public synchronized void method()
{
    method body
}
```

等价于

```
public void method()
{
    this.intrinsicLock.lock();
    try
    {
        method body
    }
}
```

```
        finally { this.intrinsicLock.unlock(); }  
    }
```

例如，可以简单地声明Bank类的transfer方法为synchronized，而不是使用一个显式的锁。

内部对象锁只有一个相关条件。wait方法添加一个线程到等待集中，notifyAll /notify方法解除等待线程的阻塞状态。换句话说，调用wait或notifyAll等价于

```
intrinsicCondition.await();  
intrinsicCondition.signalAll();
```



注释：wait、notifyAll以及notify方法是Object类的final方法。Condition方法必须被命名为await、signalAll和signal以便它们不会与那些方法发生冲突。

例如，可以用Java实现Bank类如下：

```
class Bank  
{  
    public synchronized void transfer(int from, int to, int amount) throws InterruptedException  
    {  
        while (accounts[from] < amount)  
            wait(); // wait on intrinsic object lock's single condition  
        accounts[from] -= amount;  
        accounts[to] += amount;  
        notifyAll(); // notify all threads waiting on the condition  
    }  
    public synchronized double getTotalBalance() { . . . }  
    private double[] accounts;  
}
```

可以看到，使用synchronized关键字来编写代码要简洁得多。当然，要理解这一代码，你必须了解每一个对象有一个内部锁，并且该锁有一个内部条件。由锁来管理那些试图进入synchronized方法的线程，由条件来管理那些调用wait的线程。



提示：Synchronized方法是相对简单的。但是，初学者常常对条件感到困惑。在使用wait/notifyAll之前，应该考虑使用第14.10节描述的结构之一。

将静态方法声明为synchronized也是合法的。如果调用这种方法，该方法获得相关的类对象的内部锁。例如，如果Bank类有一个静态同步的方法，那么当该方法被调用时，Bank.class对象的锁被锁住。因此，没有其他线程可以调用同一个类的这个或任何其他的同步静态方法。

内部锁和条件存在一些局限。包括：

- 不能中断一个正在试图获得锁的线程。
- 试图获得锁时不能设定超时。
- 每个锁仅有单一的条件，可能是不够的。

在代码中应该使用哪一种？Lock和Condition对象还是同步方法？下面是一些建议：

- 最好既不使用Lock/Condition也不使用synchronized关键字。在许多情况下你可以使用java.util.concurrent包中的一种机制，它会为你处理所有的加锁。例如，在第14.6节，你会看到如何使用阻塞队列来同步完成一个共同任务的线程。



- 如果synchronized关键字适合你的程序，那么请尽量使用它，这样可以减少编写的代码数量，减少出错的几率。例14-9给出了用同步方法实现的银行实例。
- 如果特别需要Lock/Condition结构提供的独有特性时，才使用Lock/Condition。

**例14-9 Bank.java**

```
1. /**
2.  * A bank with a number of bank accounts that uses synchronization primitives.
3.  * @version 1.30 2004-08-01
4.  * @author Cay Horstmann
5.  */
6. public class Bank
7. {
8.     /**
9.      * Constructs the bank.
10.     * @param n the number of accounts
11.     * @param initialBalance the initial balance for each account
12.     */
13.     public Bank(int n, double initialBalance)
14.     {
15.         accounts = new double[n];
16.         for (int i = 0; i < accounts.length; i++)
17.             accounts[i] = initialBalance;
18.     }
19.
20.     /**
21.     * Transfers money from one account to another.
22.     * @param from the account to transfer from
23.     * @param to the account to transfer to
24.     * @param amount the amount to transfer
25.     */
26.     public synchronized void transfer(int from, int to, double amount) throws InterruptedException
27.     {
28.         while (accounts[from] < amount)
29.             wait();
30.         System.out.print(Thread.currentThread());
31.         accounts[from] -= amount;
32.         System.out.printf(" %10.2f from %d to %d", amount, from, to);
33.         accounts[to] += amount;
34.         System.out.printf(" Total Balance: %10.2f\n", getTotalBalance());
35.         notifyAll();
36.     }
37.
38.     /**
39.     * Gets the sum of all account balances.
40.     * @return the total balance
41.     */
42.     public synchronized double getTotalBalance()
43.     {
44.         double sum = 0;
45.
46.         for (double a : accounts)
47.             sum += a;
48.
49.         return sum;
```

```
50. }
51.
52. /**
53.  * Gets the number of accounts in the bank.
54.  * @return the number of accounts
55.  */
56. public int size()
57. {
58.     return accounts.length;
59. }
60.
61. private final double[] accounts;
62. }
```

#### API java.lang.Object 1.0

- `void notifyAll()`  
解除那些在该对象上调用`wait`方法的线程的阻塞状态。该方法只能在同步方法或同步块内部调用。如果当前线程不是对象锁的持有者，该方法抛出一个`IllegalMonitorStateException`异常。
- `void notify()`  
随机选择一个在该对象上调用`wait`方法的线程，解除其阻塞状态。该方法只能在一个同步方法或同步块中调用。如果当前线程不是对象锁的持有者，该方法抛出一个`IllegalMonitorStateException`异常。
- `void wait()`  
导致线程进入等待状态直到它被通知。该方法只能在一个同步方法中调用。如果当前线程不是对象锁的持有者，该方法抛出一个`IllegalMonitorStateException`异常。
- `void wait(long millis)`
- `void wait(long millis, int nanos)`  
导致线程进入等待状态直到它被通知或者经过指定的时间。这些方法只能在一个同步方法中调用。如果当前线程不是对象锁的持有者该方法抛出一个`IllegalMonitorStateException`异常。

参数：millis          毫秒数  
          nanos          纳秒数，<1 000 000

#### 14.5.6 同步阻塞

正如刚刚讨论的，每一个Java对象有一个锁。线程可以通过调用同步方法获得锁。还有另一种机制可以获得锁，通过进入一个同步阻塞。当线程进入如下形式的阻塞：

```
synchronized (obj) // this is the syntax for a synchronized block
{
    critical section
}
```

于是它获得obj的锁。

有时会发现“特殊的”锁，例如：

```
public class Bank
{
    public void transfer(int from, int to, int amount)
    {
        synchronized (lock) // an ad-hoc lock
        {
            accounts[from] -= amount;
            accounts[to] += amount;
        }
        System.out.println(. . .);
    }
    . . .
    private double[] accounts;
    private Object lock = new Object();
}
```

在此，lock对象被创建仅仅是用来使用每个Java对象持有的锁。

有时程序员使用一个对象的锁来实现额外的原子操作，实际上称为客户端锁定（client-side locking）。例如，考虑Vector类，一个列表，它的方法是同步的。现在，假定在Vector<Double>中存储银行余额。这里有一个transfer方法的原始实现：

```
public void transfer(Vector<Double> accounts, int from, int to, int amount) // ERROR
{
    accounts.set(from, accounts.get(from) - amount);
    accounts.set(to, accounts.get(to) + amount);
    System.out.println(. . .);
}
```

Vector类的get和set方法是同步的，但是，这对于我们并没有什么帮助。在第一次对get的调用已经完成之后，一个线程完全可能在transfer方法中被剥夺运行权。于是，另一个线程可能在相同的存储位置存入不同的值。但是，我们可以截获这个锁：

```
public void transfer(Vector<Double> accounts, int from, int to, int amount)
{
    synchronized (accounts)
    {
        accounts.set(from, accounts.get(from) - amount);
        accounts.set(to, accounts.get(to) + amount);
    }
    System.out.println(. . .);
}
```

这个方法可以工作，但是它完全依赖于这样一个事实，Vector类对自己的所有可修改方法都使用内部锁。然而，这是真的吗？Vector类的文档没有给出这样的承诺。不得不仔细研究源代码并希望将来的版本能介绍非同步的可修改方法。如你所见，客户端锁定是非常脆弱的，通常不推荐使用。

#### 14.5.7 监视器概念

锁和条件是线程同步的强大工具，但是，严格地讲，它们不是面向对象的。多年来，研究

人员努力寻找一种方法，可以在不需要程序员考虑如何加锁的情况下，就可以保证多线程的安全性。最成功的解决方案之一是监视器（monitor），这一概念最早是由Per Brinch Hansen和Tony Hoare 在20世纪70年代提出的。用Java的术语来讲，监视器具有如下特性：

- 监视器是只包含私有域的类。
- 每个监视器类的对象有一个相关的锁。
- 使用该锁对所有的方法进行加锁。换句话说，如果客户端调用obj.method()，那么obj对象的锁是在方法调用开始时自动获得，并且当方法返回时自动释放该锁。因为所有的域是私有的，这样的安排可以确保一个线程在对对象操作时，没有其他线程能访问该域。
- 该锁可以有任意多个相关条件。

监视器的早期版本只有单一的条件，使用一种很优雅的句法。可以简单地调用await  
accounts[from] >= balance而不使用任何显式的条件变量。然而，研究表明盲目地重新测试条件是低效的。显式的条件变量解决了这一问题。每一个条件变量管理一个独立的线程集。

Java设计者以不是很精确的方式采用了监视器概念，Java中的每一个对象有一个内部的锁和内部的条件。如果一个方法用synchronized关键字声明，那么，它表现的就像是一个监视器方法。通过调用wait/notifyAll/notify来访问条件变量。

然而，在下述的3个方面Java对象不同于监视器，从而使得线程的安全性下降：

- 域不要求必须是private。
- 方法不要求必须是synchronized。
- 内部锁对客户是可用的。

这种对安全性的轻视激怒了Per Brinch Hansen。他在一次对原始Java中的多线程的严厉评论中，写到：“这实在是令我震惊，在监视器和并发Pascal出现四分之一世纪后，Java的这种不安全的并行机制被编程社区接受。这没有任何益处。” [Java's Insecure Parallelism, ACM SIGPLAN Notices 34:38-45, April 1999.]

#### 14.5.8 Volatile域

有时，仅仅为了读写一个或两个实例域就使用同步，显得开销过大了。毕竟，什么地方能出错呢？遗憾的是，使用现代的处理器和编译器，出错的可能性很大。

- 多处理器的计算机能够暂时在寄存器或本地内存缓冲区中保存内存中的值。结果是，运行在不同处理器上的线程可能在同一个内存位置取到不同的值。
- 编译器可以改变指令执行的顺序以使吞吐量最大化。这种顺序上的变化不会改变代码语义，但是编译器假定内存的值仅仅在代码中有显式的修改指令时才会改变。然而，内存的值可以被另一个线程改变！

如果你使用锁来保护可以被多个线程访问的代码，那么可以不考虑这种问题。编译器被要求通过在必要的时候刷新本地缓存来保持锁的效应，并且不能不正当地重新排序指令。详细的解释见JSR 133的Java内存模型和线程规范（参看 <http://www.jcp.org/en/jsr/detail?id=133>）。该规范的大部分很复杂而且技术性强，但是文档中也包含了很多解释得很清晰的例子。在<http://www-106.ibm.com/developerworks/java/library/j-jtp02244.html>有Brian Goetz写的一个更

易懂的概要介绍。

☑ 注释：Brian Goetz给出了下述“同步格言”：“如果向一个变量写入值，而这个变量接下来可能会被另一个线程读取，或者，从一个变量读值，而这个变量可能是之前被另一个线程写入的，此时必须使用同步”。

`volatile`关键字为实例域的同步访问提供了一种免锁机制。如果声明一个域为`volatile`，那么编译器和虚拟机就知道该域是可能被另一个线程并发更新的。

例如，假定一个对象有一个布尔标记`done`，它的值被一个线程设置却被另一个线程查询，如同我们讨论过的那样，你可以使用锁：

```
public synchronized boolean isDone() { return done; }
public synchronized void setDone() { done = true; }
private boolean done;
```

或许使用内部锁不是个好主意。如果另一个线程已经对该对象加锁，`isDone`和`setDone`方法可能阻塞。如果注意到这个方面，一个线程可以为这一变量使用独立的`Lock`。但是，这也会带来许多麻烦。

在这种情况下，将域声明为`volatile`是合理的：

```
public boolean isDone() { return done; }
public void setDone() { done = true; }
private volatile boolean done;
```

✗ 警告：`Volatile`变量不能提供原子性。例如，方法

```
public void flipDone() { done = !done; } // not atomic
```

不能确保改变域中的值。

在这样一种非常简单的情况下，存在第3种可能性，使用 `AtomicBoolean`。这个类有方法 `get`和`set`，且确保是原子的（就像它们是同步的一样）。该实现使用有效的机器指令，在不使用锁的情况下确保原子性。在`java.util.concurrent.atomic`中有许多包装器类用于原子的整数、浮点数、数组等。这些类是为编写并发实用程序的系统程序员提供使用的，而不是应用程序员。

总之，在以下3个条件下，域的并发访问是安全的：

- 域是`final`，并且在构造器调用完成之后被访问。
- 对域的访问由公有的锁进行保护。
- 域是`volatile`的。

☑ 注释：Java SE 5.0之前，`volatile`的语义是允许的。语言的设计者试图在优化使用`volatile`域的代码的性能方面给实现人员留有余地。但是，旧规范太复杂，实现人员难以理解，这带来了混乱和非预期的行为。例如，不可变对象不是真的不可变。

#### 14.5.9 死锁

锁和条件不能解决多线程中的所有问题。考虑下面的情况：

账户1：\$200

账户2：\$300  
线程1：从账户1转移\$300到账户2  
线程2：从账户2转移\$400到账户1

如图14-6所示，线程1和线程2都被阻塞了。因为账户1以及账户2中的余额都不足以进行转账，两个线程都无法执行下去。

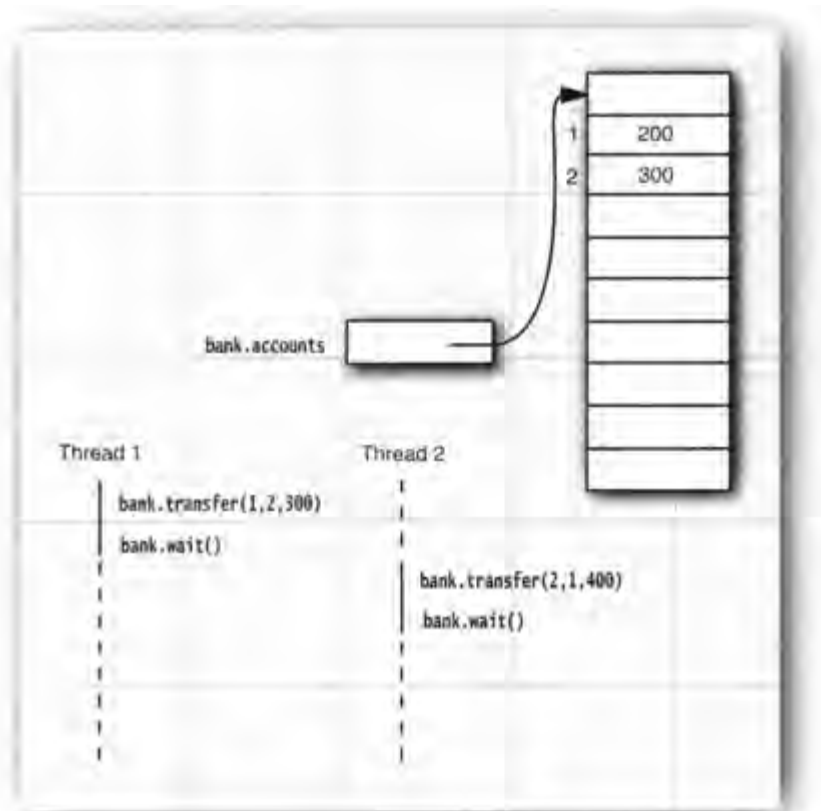


图14-6 发生死锁的情况

是否会因为每一个线程要等待更多的钱款存入而导致所有线程都被阻塞呢？这样的状态称为死锁（deadlock）。

在这个程序里，死锁不会发生，原因很简单。每一次转账至多\$1 000。因为有100个账户，而且所有账户的总金额是\$100 000，在任意时刻，至少有一个账户的余额高于\$1 000。从该账户取钱的线程可以继续运行。

但是，如果修改run方法，把每次转账至多\$1 000的限制去掉，死锁很快就会发生。试试看。将NACCOUNTS设为10。每次交易的金额上限设置为 $2 * \text{INITIAL\_BALANCE}$ ，然后运行该程序。程序将运行一段时间后就会挂起。



提示：当程序挂起时，键入`CTRL+\`，将得到一个所有线程的列表。每一个线程有一个栈踪迹，告诉你线程被阻塞的位置。像第11章叙述的那样，运行jconsole并参考线程面板（见图14-7）。

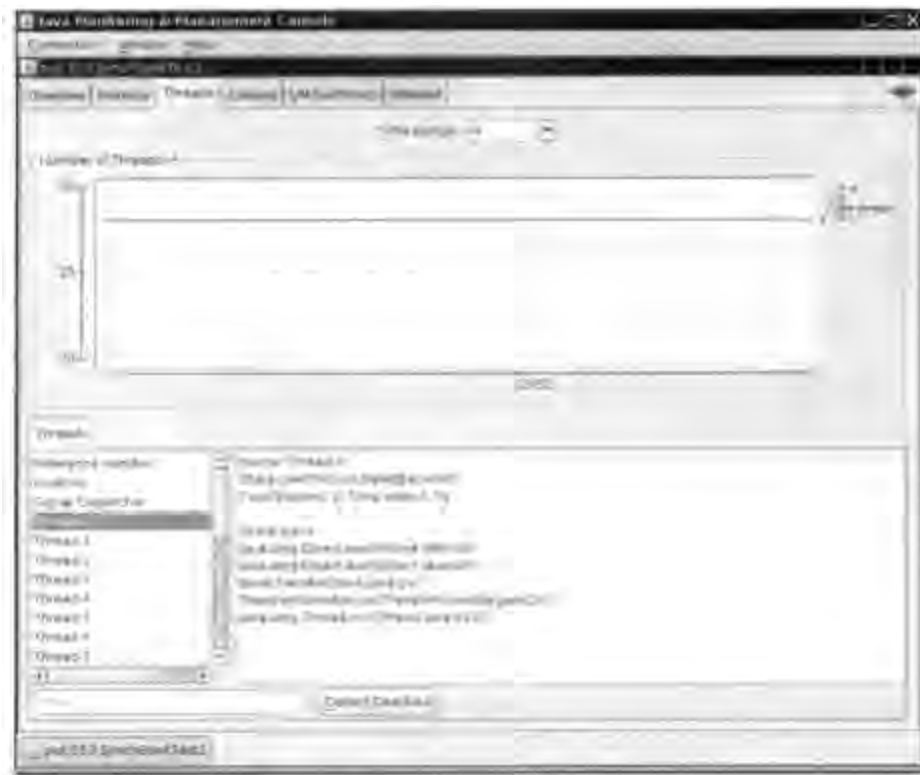


图14-7 jconsole中的线程面板

导致死锁的另一种途径是让第 $i$ 个线程负责向第 $i$ 个账户存钱，而不是从第 $i$ 个账户取钱。这样一来，有可能将所有的线程都集中到一个账户上，每一个线程都试图从这个账户中取出大于该账户余额的钱。试试看。在SynchBankTest程序中，转用TransferRunnable类的run方法。在调用transfer时，交换fromAccount和toAccount。运行该程序并查看它为什么会立即死锁。

还有一种很容易导致死锁的情况：在SynchBankTest程序中，将signalAll方法转换为signal，会发现该程序最终会挂起（将NACCOUNTS设为10可以更快地看到结果）。signalAll通知所有等待增加资金的线程，与此不同的是signal方法仅仅对一个线程解锁。如果该线程不能继续运行，所有的线程可能都被阻塞。考虑下面这个会发生死锁的例子。

账户1：\$1 990

所有其他账户：每一个\$990

线程1：从账户1转移\$995到账户2

所有其他线程：从他们的账户转移\$995到另一个账户

显然，除了线程1，所有的线程都被阻塞，因为他们的账户中没有足够的余额。

线程1继续执行，运行后出现如下状况：

账户1：\$995

账户2：\$1 985

所有其他账户：每个\$990

然后，线程1调用signal。signal方法随机选择一个线程为它解锁。假定它选择了线程3。该

线程被唤醒，发现在它的账户里没有足够的金额，它再次调用`await`。但是，线程1仍在运行，将随机地产生一个新的交易，例如，

线程1：从账户1转移\$997到账户2

现在，线程1也调用`await`，所有的线程都被阻塞。系统死锁。

问题的起因在于调用 `signal`。它仅仅为一个线程解锁，而且，它很可能选择一个不能继续运行的线程（在我们的例子中，线程2必须把钱从账户2中取出）。

遗憾的是，Java编程语言中没有任何东西可以避免或打破这种死锁现象。必须仔细设计程序，以确保不会出现死锁。

#### 14.5.10 锁测试与超时

线程在调用`lock`方法来获得另一个线程所持有的锁的时候，很可能发生阻塞。应该更加谨慎地申请锁。`tryLock`方法试图申请一个锁，在成功获得锁后返回`true`，否则，立即返回`false`，而且线程可以立即离开去做其他事情。

```
if (myLock.tryLock())
    // now the thread owns the lock
    try { . . . }
    finally { myLock.unlock(); }
else
    // do something else
```

可以调用`tryLock`时，使用超时参数，像这样：

```
if (myLock.tryLock(100, TimeUnit.MILLISECONDS)) . . .
```

`TimeUnit`是一个枚举类型，可以取的值包括`SECONDS`、`MILLISECONDS`、`MICROSECONDS`和`NANOSECONDS`。

`lock`方法不能被中断。如果一个线程在等待获得一个锁时被中断，中断线程在获得锁之前一直处于阻塞状态。如果出现死锁，那么，`lock`方法就无法终止。

然而，如果调用带有用超时参数的`tryLock`，那么如果线程在等待期间被中断，将抛出`InterruptedException`异常。这是一个非常有用的特性，因为允许程序打破死锁。

也可以调用`lockInterruptibly`方法。它就相当于一个超时设为无限的`tryLock`方法。

在等待一个条件时，也可以提供一个超时：

```
myCondition.await(100, TimeUnit.MILLISECONDS))
```

如果一个线程被另一个线程通过调用`signalAll`或 `signal`激活，或者超时时限已达到，或者线程被中断，那么`await`方法将返回。

如果等待的线程被中断，`await`方法将抛出一个`InterruptedException`异常。在你希望出现这种情况时线程继续等待（可能不太合理），可以使用`awaitUninterruptibly`方法代替`await`。

**API** `java.util.concurrent.locks.Lock 5.0`

- `boolean tryLock()`

尝试获得锁而没有发生阻塞；如果成功返回真。这个方法会抢夺可用的锁，即使该锁有



公平加锁策略，即便其他线程已经等待很久也是如此。

- `boolean tryLock(long time, TimeUnit unit)`  
尝试获得锁，阻塞时间不会超过给定的值；如果成功返回true。
- `void lockInterruptibly()`  
获得锁，但是会不确定地发生阻塞。如果线程被中断，抛出一个`InterruptedException`异常。

#### `java.util.concurrent.locks.Condition 5.0`

- `boolean await(long time, TimeUnit unit)`  
进入该条件的等待集，直到线程从等待集中移出或等待了指定的时间之后才解除阻塞。如果因为等待时间到了而返回就返回false，否则返回true。
- `void awaitUninterruptibly()`  
进入该条件的等待集，直到线程从等待集移出才解除阻塞。如果线程被中断，该方法不会抛出`InterruptedException`异常。

#### 14.5.11 读/写锁

`java.util.concurrent.locks`包定义了两个锁类，我们已经讨论的`ReentrantLock`类和`ReentrantReadWriteLock`类。如果很多线程从一个数据结构读取数据而很少线程修改其中数据的话，后者是十分有用的。在这种情况下，允许对读者线程共享访问是合适的。当然，写者线程依然必须是互斥访问的。

下面是使用读/写锁的必要步骤：

1) 构造一个`ReentrantReadWriteLock`对象：

```
private ReentrantReadWriteLock rwl = new ReentrantReadWriteLock();
```

2) 抽取读锁和写锁：

```
private Lock readLock = rwl.readLock();  
private Lock writeLock = rwl.writeLock();
```

3) 对所有的访问者加读锁：

```
public double getTotalBalance()  
{  
    readLock.lock();  
    try { . . . }  
    finally { readLock.unlock(); }  
}
```

4) 对所有的修改者加写锁：

```
public void transfer(. . .)  
{  
    writeLock.lock();  
    try { . . . }  
    finally { writeLock.unlock(); }  
}
```

**API** java.util.concurrent.locks.ReentrantReadWriteLock 5.0

- Lock readLock()  
得到一个可以被多个读操作共用的读锁，但会排斥所有写操作。
- Lock writeLock()  
得到一个写锁，排斥所有其他的读操作和写操作。

#### 14.5.12 为什么弃用 stop 和 suspend 方法

初始的Java版本定义了一个stop方法用来终止一个线程，以及一个suspend方法用来阻塞一个线程直至另一个线程调用resume。stop和suspend方法有一些共同点：都试图控制一个给定线程的行为。

从Java SE 1.2起就弃用了这两个方法。stop方法天生就不安全，经验证明suspend方法会经常导致死锁。在本节，将看到这些问题所在，以及怎样避免这些问题的出现。

首先来看看stop方法，该方法终止所有未结束的方法，包括run方法。当线程被终止，立即释放被它锁住的所有对象的锁。这会导致对象处于不一致的状态。例如，假定TransferThread在从一个账户向另一个账户转账的过程中被终止，钱款已经转出，却没有转入目标账户，现在银行对象就被破坏了。因为锁已经被释放，这种破坏会被其他尚未停止的线程观察到。

当线程要终止另一个线程时，无法知道什么时候调用stop方法是安全的，什么时候导致对象被破坏。因此，该方法被弃用了。在希望停止线程的时候应该中断线程，被中断的线程会在安全的时候停止。



注释：一些作者声称stop方法被弃用是因为它会导致对象被一个已停止的线程永久锁定。但是，这一说法是错误的。从技术上讲，被停止的线程通过抛出ThreadDeath异常退出所有它所调用的同步方法。结果是，该线程释放它持有的内部对象锁。

接下来，看看suspend方法有什么问题。与stop不同，suspend不会破坏对象。但是，如果用suspend挂起一个持有一个锁的线程，那么，该锁在恢复之前是不可用的。如果调用suspend方法的线程试图获得同一个锁，那么程序死锁：被挂起的线程等着被恢复，而将其挂起的线程等待获得锁。

在图形用户界面中经常出现这种情况。假定我们有一个图形化的银行模拟程序。Pause按钮用来挂起转账线程，而Resume按钮用来恢复线程。

```
pauseButton.addActionListener(new
    ActionListener()
    {
        public void actionPerformed(ActionEvent event)
        {
            for (int i = 0; i < threads.length; i++)
                threads[i].suspend(); // Don't do this
        }
    });
resumeButton.addActionListener(. . .); // calls resume on all transfer threads
```

假设有一个paintComponent方法，通过调用getBalances方法获得一个余额数组，从而为每

一个账户绘制图表。

就像在第14.11节所看到的，按钮动作和重绘动作出现在同一个线程中——事件分配线程（event dispatch thread）。考虑下面的情况：

- 1) 某个转账线程获得bank对象的锁。
- 2) 用户点击Pause按钮。
- 3) 所有转账线程被挂起；其中之一仍然持有bank对象上的锁。
- 4) 因为某种原因，该账户图表需要重新绘制。
- 5) paintComponent方法调用getBalances方法。
- 6) 该方法试图获得bank对象的锁。

现在程序被冻结了。

事件分配线程不能继续运行，因为锁由一个被挂起的线程所持有。因此，用户不能点击Resume按钮，并且这些线程无法恢复。

如果想安全地挂起线程，引入一个变量suspendRequested并在run方法的某个安全的地方测试它，安全的地方是指该线程没有封锁其他线程需要的对象的地方。当该线程发现suspendRequested变量已经设置，将会保持等待状态直到它再次获得为止。

下面的代码框架实现这一设计：

```
public void run()
{
    while (...)
    {
        ...
        if (suspendRequested)
        {
            suspendLock.lock();
            try { while (suspendRequested) suspendCondition.await(); }
            finally { suspendLock.unlock(); }
        }
    }
}

public void requestSuspend() { suspendRequested = true; }
public void requestResume()
{
    suspendRequested = false;
    suspendLock.lock();
    try { suspendCondition.signalAll(); }
    finally { suspendLock.unlock(); }
}

private volatile boolean suspendRequested = false;
private Lock suspendLock = new ReentrantLock();
private Condition suspendCondition = suspendLock.newCondition();
```

## 14.6 阻塞队列

现在，读者已经看到了形成Java并发程序设计基础的底层构建块。然而，对于实际编程来说，应该尽可能远离底层结构。使用由并发处理的专业人士实现的较高层次的结构要方便得多、要安全得多。

对于许多线程问题，可以通过使用一个或多个队列以优雅且安全的方式将其形式化。生产者线程向队列插入元素，消费者线程则取出它们。使用队列，可以安全地从一个线程向另一个线程传递数据。例如，考虑银行转账程序，转账线程将转账指令对象插入一个队列中，而不是直接访问银行对象。另一个线程从队列中取出指令执行转账。只有该线程可以访问该银行对象的内部。因此不需要同步。（当然，线程安全的队列类的实现者不能不考虑锁和条件，但是，那是他们的问题而不是你的问题。）

当试图向队列添加元素而队列已满，或是想从队列移出元素而队列为空的时候，阻塞队列（blocking queue）导致线程阻塞。在协调多个线程之间的合作时，阻塞队列是一个有用的工具。工作者线程可以周期性地中间结果存储在阻塞队列中。其他的工作者线程移出中间结果并进一步加以修改。队列会自动地平衡负载。如果第一个线程集运行得比第二个慢，第二个线程集在等待结果时会阻塞。如果第一个线程集运行得快，它将等待第二个队列集赶上来。表14-1给出了阻塞队列的方法。

表14-1 阻塞队列方法

方 法	正 常 动 作	特殊情况下的动作
add	添加一个元素	如果队列满，则抛出IllegalStateException异常
element	返回队列的头元素	如果队列空，抛出NoSuchElementException异常
offer	添加一个元素并返回true	如果队列满，返回false
peek	返回队列的头元素	如果队列空，则返回null
poll	移出并返回队列的头元素	如果队列空，则返回null
put	添加一个元素	如果队列满，则阻塞
remove	移出并返回头元素	如果队列空，则抛出NoSuchElementException异常
take	移出并返回头元素	如果队列空，则阻塞

阻塞队列方法分为以下3类，这取决于当队列满或空时它们的响应方式。如果将队列当作线程管理工具来使用，将要用到put和take方法。当试图向满的队列中添加或从空的队列中移出元素时，add、remove和element操作抛出异常。当然，在一个多线程程序中，队列会在任何时候空或满，因此，一定要使用offer、poll和peek方法作为替代。这些方法如果不能完成任务，只是给出一个错误提示而不会抛出异常。



注释：poll和peek方法返回空来指示失败。因此，向这些队列中插入null值是非法的。

还有带有超时的offer方法和poll方法的变体。例如，下面的调用：

```
boolean success = q.offer(x, 100, TimeUnit.MILLISECONDS);
```

尝试在100毫秒的时间内在队列的尾部插入一个元素。如果成功返回true；否则，达到超时时间，返回false。类似地，下面的调用：

```
Object head = q.poll(100, TimeUnit.MILLISECONDS)
```

尝试用100毫秒的时间移除队列的头元素；如果成功返回头元素，否则，达到在超时时间，返回null。

如果队列满，则put方法阻塞；如果队列空，则take方法阻塞。在不带超时参数时，offer和

poll方法等效。

java.util.concurrent包提供了阻塞队列的几个变种。默认情况下，LinkedBlockingQueue的容量是没有上边界的，但是，也可以选择指定最大容量。LinkedBlockingDeque是一个双端的版本。ArrayBlockingQueue在构造时需要指定容量，并且有一个可选的参数来指定是否需要公平性。若设置了公平参数，则那么等待了最长时间的线程会优先得到处理。通常，公平性会降低性能，只有在确实非常需要时才使用它。

PriorityBlockingQueue是一个带优先级的队列，而不是先进先出队列。元素按照它们的优先级顺序被移出。该队列是没有容量上限，但是，如果队列是空的，取元素的操作会阻塞。（有关优先级队列的详细内容参看第13章。）

最后，DelayQueue包含实现Delayed接口的对象：

```
interface Delayed extends Comparable<Delayed>
{
    long getDelay(TimeUnit unit);
}
```

getDelay方法返回对象的残留延迟。负值表示延迟已经结束。元素只有在延迟用完的情况下才能从DelayQueue移除。还必须实现compareTo方法。DelayQueue使用该方法对元素进行排序。

例14-10中的程序展示了如何使用阻塞队列来控制线程集。程序在一个目录及它的所有子目录下搜索所有文件，打印出包含指定关键字的行。

#### 例14-10 BlockingQueueTest.java

```
1. import java.io.*;
2. import java.util.*;
3. import java.util.concurrent.*;
4.
5. /**
6.  * @version 1.0 2004-08-01
7.  * @author Cay Horstmann
8.  */
9. public class BlockingQueueTest
10. {
11.     public static void main(String[] args)
12.     {
13.         Scanner in = new Scanner(System.in);
14.         System.out.print("Enter base directory (e.g. /usr/local/jdk1.6.0/src): ");
15.         String directory = in.nextLine();
16.         System.out.print("Enter keyword (e.g. volatile): ");
17.         String keyword = in.nextLine();
18.
19.         final int FILE_QUEUE_SIZE = 10;
20.         final int SEARCH_THREADS = 100;
21.
22.         BlockingQueue<File> queue = new ArrayBlockingQueue<File>(FILE_QUEUE_SIZE);
23.
24.         FileEnumerationTask enumerator = new FileEnumerationTask(queue, new File(directory));
25.         new Thread(enumerator).start();
26.         for (int i = 1; i <= SEARCH_THREADS; i++)
27.             new Thread(new SearchTask(queue, keyword)).start();
28.     }
```

```
29. }
30.
31. /**
32.  * This task enumerates all files in a directory and its subdirectories.
33.  */
34. class FileEnumerationTask implements Runnable
35. {
36.     /**
37.      * Constructs a FileEnumerationTask.
38.      * @param queue the blocking queue to which the enumerated files are added
39.      * @param startingDirectory the directory in which to start the enumeration
40.      */
41.     public FileEnumerationTask(BlockingQueue<File> queue, File startingDirectory)
42.     {
43.         this.queue = queue;
44.         this.startingDirectory = startingDirectory;
45.     }
46.
47.     public void run()
48.     {
49.         try
50.         {
51.             enumerate(startingDirectory);
52.             queue.put(DUMMY);
53.         }
54.         catch (InterruptedException e)
55.         {
56.         }
57.     }
58.
59.     /**
60.      * Recursively enumerates all files in a given directory and its subdirectories
61.      * @param directory the directory in which to start
62.      */
63.     public void enumerate(File directory) throws InterruptedException
64.     {
65.         File[] files = directory.listFiles();
66.         for (File file : files)
67.         {
68.             if (file.isDirectory()) enumerate(file);
69.             else queue.put(file);
70.         }
71.     }
72.
73.     public static File DUMMY = new File("");
74.
75.     private BlockingQueue<File> queue;
76.     private File startingDirectory;
77. }
78.
79. /**
80.  * This task searches files for a given keyword.
81.  */
82. class SearchTask implements Runnable
83. {
84.     /**
85.      * Constructs a SearchTask.
```

```
86.  * @param queue the queue from which to take files
87.  * @param keyword the keyword to look for
88.  */
89.  public SearchTask(BlockingQueue<File> queue, String keyword)
90.  {
91.      this.queue = queue;
92.      this.keyword = keyword;
93.  }
94.
95.  public void run()
96.  {
97.      try
98.      {
99.          boolean done = false;
100.         while (!done)
101.         {
102.             File file = queue.take();
103.             if (file == FileEnumerationTask.DUMMY)
104.             {
105.                 queue.put(file);
106.                 done = true;
107.             }
108.             else search(file);
109.         }
110.     }
111.     catch (IOException e)
112.     {
113.         e.printStackTrace();
114.     }
115.     catch (InterruptedException e)
116.     {
117.     }
118. }
119.
120. /**
121.  * Searches a file for a given keyword and prints all matching lines.
122.  * @param file the file to search
123.  */
124.  public void search(File file) throws IOException
125.  {
126.      Scanner in = new Scanner(new FileInputStream(file));
127.      int lineNumber = 0;
128.      while (in.hasNextLine())
129.      {
130.          lineNumber++;
131.          String line = in.nextLine();
132.          if (line.contains(keyword)) System.out.printf("%s:%d:%s\n", file.getPath(),
133.              lineNumber, line);
134.      }
135.      in.close();
136.  }
137.
138.  private BlockingQueue<File> queue;
139.  private String keyword;
140. }
```

生产者线程枚举在所有子目录下的所有文件并把它们放到一个阻塞队列中。这个操作很快，如果没有上限的话，很快就包含了所有找到的文件。

我们同时启动了大量搜索线程。每个搜索线程从队列中取出一个文件，打开它，打印所有包含该关键字的行，然后取出下一个文件。我们使用一个小技巧在工作结束后终止这个应用程序。为了发出完成信号，枚举线程放置一个虚拟对象到队列中（这就像在行李输送带上放一个写着“最后一个包”的虚拟包）。当搜索线程取到这个虚拟对象时，将其放回并终止。

注意，不需要显式的线程同步。在这个应用程序中，我们使用队列数据结构作为一种同步机制。

**API** `java.util.concurrent.ArrayBlockingQueue<E> 5.0`

- `ArrayBlockingQueue(int capacity)`
- `ArrayBlockingQueue(int capacity, boolean fair)`

构造一个带有指定的容量和公平性设置的阻塞队列。该队列用循环数组实现。

**API** `java.util.concurrent.LinkedBlockingQueue<E> 5.0`

**API** `java.util.concurrent.LinkedBlockingDeque<E> 6`

- `LinkedBlockingQueue()`
- `LinkedBlockingDeque()`
- `LinkedBlockingQueue(int capacity)`
- `LinkedBlockingDeque(int capacity)`

构造一个无上限的阻塞队列或双向队列，用链表实现。

根据指定容量构建一个有限的阻塞队列或双向队列，用链表实现。

**API** `java.util.concurrent.DelayQueue<E extends Delayed> 5.0`

- `DelayQueue()`

构造一个包含`Delayed`元素的无界的阻塞时间有限的阻塞队列。只有那些延迟已经超过时间的元素可以从队列中移出。

**API** `java.util.concurrent.Delayed 5.0`

- `long getDelay(TimeUnit unit)`

得到该对象的延迟，用给定的时间单位进行度量。

**API** `java.util.concurrent.PriorityBlockingQueue<E> 5.0`

- `PriorityBlockingQueue()`
- `PriorityBlockingQueue(int initialCapacity)`
- `PriorityBlockingQueue(int initialCapacity, Comparator<? super E> comparator)`



构造一个无边界阻塞优先队列，用堆实现。

参数：`initialCapacity`      优先队列的初始容量。默认值是11。  
`comparator`              用来对元素进行比较的比较器，如果没有指定，则元素必须实现Comparable接口。

#### **API** `java.util.concurrent.BlockingQueue<E> 5.0`

- `void put(E element)`  
添加元素，在必要时阻塞。
- `E take()`  
移除并返回头元素，必要时阻塞。
- `boolean offer(E element, long time, TimeUnit unit)`  
添加给定的元素，如果成功返回true，如果必要时阻塞，直至元素已经被添加或超时。
- `E poll(long time, TimeUnit unit)`  
移除并返回头元素，必要时阻塞，直至元素可用或超时用完。失败时返回null。

#### **API** `java.util.concurrent.BlockingDeque<E> 6`

- `void putFirst(E element)`
- `void putLast(E element)`  
添加元素，必要时阻塞。
- `E takeFirst()`
- `E takeLast()`  
移除并返回头元素或尾元素，必要时阻塞。
- `boolean offerFirst(E element, long time, TimeUnit unit)`
- `boolean offerLast(E element, long time, TimeUnit unit)`  
添加给定的元素，成功时返回true，必要时阻塞直至元素被添加或超时。
- `E pollFirst(long time, TimeUnit unit)`
- `E pollLast(long time, TimeUnit unit)`  
移动并返回头元素或尾元素，必要时阻塞，直至元素可用或超时。失败时返回空。

## 14.7 线程安全的集合

如果多线程要并发地修改一个数据结构，例如散列表，那么很容易会破坏这个数据结构（有关散列表的详细信息见第13章）。例如，一个线程可能要开始向表中插入一个新元素。假定在调整散列表各个桶之间的链接关系的过程中，被剥夺了控制权。如果另一个线程也开始遍历同一个链表，可能使用无效的链接并造成混乱，会抛出异常或者陷入死循环。

可以通过提供锁来保护共享数据结构，但是选择线程安全的实现作为替代可能更容易些。当然，前一节讨论的阻塞队列就是线程安全的集合。在下面各小节中，将讨论Java类库提供的另一种线程安全的集合。

### 14.7.1 高效的映像、集合和队列

java.util.concurrent包提供了映像、有序集和队列的高效实现：ConcurrentHashMap、ConcurrentSkipListMap、ConcurrentSkipListSet和ConcurrentLinkedQueue。

这些集合使用复杂的算法，通过允许并发地访问数据结构的不同部分来使竞争极小化。

与大多数集合不同，size方法不必在常量时间内操作。确定这样的集合当前的大小通常需要遍历。

集合返回弱一致性（weakly consistent）的迭代器。这意味着迭代器不一定能反映出它们被构造之后的所有的修改，但是，它们不会将同一个值返回两次，也不会抛出ConcurrentModificationException异常。



注释：与之形成对照的是，集合如果在迭代器构造之后发生改变，java.util包中的迭代器将抛出一个ConcurrentModificationException异常。

并发的散列映像表，可高效地支持大量的读者和一定数量的写者。默认情况下，假定可以有多达16个写者线程同时执行。可以有更多的写者线程，但是，如果同一时间多于16个，其他线程将暂时被阻塞。可以指定更大数目的构造器，然而，恐怕没有这种必要。

ConcurrentHashMap和ConcurrentSkipListMap类有相应的方法用于原子性的关联插入以及关联删除。putIfAbsent方法自动地添加新的关联，前提是原来没有这一关联。对于多线程访问的缓存来说这是很有用的，确保只有一个线程向缓存添加项：

```
cache.putIfAbsent(key, value);
```

相反的操作是删除（或许应该叫作removeIfPresent）。调用

```
cache.remove(key, value)
```

将原子性地删除键值对，如果它们在映像表中出现的话。最后，

```
cache.replace(key, oldValue, newValue)
```

原子性地用新值替换旧值，假定旧值与指定的键值关联。



java.util.concurrent.ConcurrentLinkedQueue<E> 5.0

- ConcurrentLinkedQueue<E>()
- 构造一个可以被多线程安全访问的无边界非阻塞的队列。



java.util.concurrent.ConcurrentLinkedQueue<E> 6

- ConcurrentSkipListSet<E>()
  - ConcurrentSkipListSet<E>(Comparator<? super E> comp)
- 构造一个可以被多线程安全访问的有序集。第一个构造器要求元素实现Comparable接口。



java.util.concurrent.ConcurrentHashMap<K, V> 5.0



java.util.concurrent.ConcurrentSkipListMap<K, V> 6

- ConcurrentHashMap<K, V>()

- `ConcurrentHashMap<K, V>(int initialCapacity)`
- `ConcurrentHashMap<K, V>(int initialCapacity, float loadFactor, int concurrencyLevel)`

构造一个可以被多线程安全访问的散列映像表。

参数：initialCapacity      集合的初始容量。默认值为16。

loadFactor      控制调整：如果每一个桶的平均负载超过这个因子，表的大小会被重新调整。默认值为0.75。

concurrencyLevel      并发写者线程的估计数目。

- `ConcurrentSkipListMap<K, V>()`
- `ConcurrentSkipListSet<K, V>(Comparator<? super K> comp)`

构造一个可以被多线程安全访问的有序的映像表。第一个构造器要求键实现Comparable接口。

- `V putIfAbsent(K key, V value)`

如果该键没有在映像表中出现，则将给定的值同给定的键关联起来，并返回null。否则返回与该键关联的现有值。

- `boolean remove(K key, V value)`

如果给定的键与给定的值关联，删除给定的键与值并返回真。否则，返回false。

- `boolean replace(K key, V oldValue, V newValue)`

如果给定的键当前与oldvalue相关联，用它与newValue关联。否则，返回false。

#### 14.7.2 写数组的拷贝

`CopyOnWriteArrayList`和`CopyOnWriteArraySet`是线程安全的集合，其中所有的修改线程对底层数组进行复制。如果在集合上进行迭代的线程数超过修改线程数，这样的安排是很有用的。当构建一个迭代器的时候，它包含一个对当前数组的引用。如果数组后来被修改了，迭代器仍然引用旧数组，但是，集合的数组已经被替换了。因而，旧的迭代器拥有一致的（可能过时的）视图，访问它无须任何同步开销。

#### 14.7.3 旧的线程安全的集合

从Java的初始版本开始，`Vector`和`Hashtable`类就提供了线程安全的动态数组和散列表的实现。在Java SE 1.2中，这些类被弃用了，取而代之的是`ArrayList`和`HashMap`类。这些类不是线程安全的，而集合库中提供了不同的机制。任何集合类通过使用同步包装器（`synchronization wrapper`）变成线程安全的：

```
List<E> synchArrayList = Collections.synchronizedList(new ArrayList<E>());  
Map<K, V> synchHashMap = Collections.synchronizedMap(new HashMap<K, V>());
```

结果集合的方法使用锁加以保护，提供了线程的安全访问。

应该确保没有任何线程通过原始的非同步方法访问数据结构。最便利的方法是确保不保存任何指向原始对象的引用，简单地构造一个集合并立即传递给包装器，像我们的例子中所做的

那样。

如果在另一个线程可能进行修改时需要对集合进行迭代，仍然需要使用“客户端”封锁：

```
synchronized (synchHashMap)
{
    Iterator<K> iter = synchHashMap.keySet().iterator();
    while (iter.hasNext()) . . . ;
}
```

如果使用“for each”循环必须使用同样的代码，因为循环使用了迭代器。注意：如果在迭代过程中，别的线程修改集合，迭代器会失效，抛出ConcurrentModificationException异常。同步仍然是需要的，因此并发的修改可以被可靠地检测出来。

最好使用java.util.concurrent包中定义的集合，不使用同步包装器中的。特别是，假如它们访问的是不同的桶，由于ConcurrentHashMap已经精心地实现了，多线程可以访问它而且不会彼此阻塞。有一个例外是经常被修改的数组列表。在那种情况下，同步的ArrayList可以胜过CopyOnWriteArrayList。

#### java.util.Collections 1.2

- static <E> Collection<E> synchronizedCollection(Collection<E> c)
- static <E> List synchronizedList(List<E> c)
- static <E> Set synchronizedSet(Set<E> c)
- static <E> SortedSet synchronizedSortedSet(SortedSet<E> c)
- static <K, V> Map<K, V> synchronizedMap(Map<K, V> c)
- static <K, V> SortedMap<K, V> synchronizedSortedMap(SortedMap<K, V> c)

构建集合视图，该集合的方法是同步的。

## 14.8 Callable与Future

Runnable封装一个异步运行的任务，可以把它想像成为一个没有参数和返回值的异步方法。Callable与Runnable类似，但是有返回值。Callable接口是一个参数化的类型，只有一个方法call。

```
public interface Callable<V>
{
    V call() throws Exception;
}
```

类型参数是返回值的类型。例如，Callable<Integer> 表示一个最终返回Integer对象的异步计算。

Future保存异步计算的结果。可以启动一个计算，将Future对象交给某个线程，然后忘掉它。Future对象的所有者在结果计算好之后就可以获得它。

Future接口具有下面的方法：

```
public interface Future<V>
{
    V get() throws . . . ;
    V get(long timeout, TimeUnit unit) throws . . . ;
}
```

```
void cancel(boolean mayInterrupt);  
boolean isCancelled();  
boolean isDone();  
}
```

第一个get方法的调用被阻塞，直到计算完成。如果在计算完成之前，第二个方法的调用超时，抛出一个TimeoutException异常。如果运行该计算的线程被中断，两个方法都将抛出InterruptedException。如果计算已经完成，那么get方法立即返回。

如果计算还在进行，isDone方法返回false；如果完成了，则返回true。

可以用cancel方法取消该计算。如果计算还没有开始，它被取消且不再开始。如果计算处于运行之中，那么如果mayInterrupt参数为true，它就被中断。

FutureTask包装器是一种非常便利的机制，可将Callable转换成Future和Runnable，它同时实现二者的接口。例如：

```
Callable<Integer> myComputation = . . . ;  
FutureTask<Integer> task = new FutureTask<Integer>(myComputation);  
Thread t = new Thread(task); // it's a Runnable  
t.start();  
. . .  
Integer result = task.get(); // it's a Future
```

例14-11中的程序使用了这些概念。这个程序与前面那个寻找包含指定关键字的文件的例子相似。然而，现在我们仅仅计算匹配的文件数目。因此，我们有了一个需要长时间运行的任务，它产生一个整数值，一个Callable<Integer>的例子。

```
class MatchCounter implements Callable<Integer>  
{  
    public MatchCounter(File directory, String keyword) { . . . }  
    public Integer call() { . . . } // returns the number of matching files  
}
```

然后我们利用MatchCounter创建一个FutureTask 对象，并用来启动一个线程。

```
FutureTask<Integer> task = new FutureTask<Integer>(counter);  
Thread t = new Thread(task);  
t.start();
```

最后，我们打印结果。

```
System.out.println(task.get() + " matching files.");
```

当然，对get的调用会发生阻塞，直到有可获得的结果为止。

在call方法内部，使用相同的递归机制。对于每一个子目录，我们产生一个新的MatchCounter并为它启动一个线程。此外，把FutureTask对象隐藏在ArrayList<Future<Integer>>中。最后，把所有结果加起来：

```
for (Future<Integer> result : results)  
    count += result.get();
```

每一次对get的调用都会发生阻塞直到结果可获得为止。当然，线程是并行运行的，因此，很可能在大致相同的时刻所有的结果都可获得。

## 例14-11 FutureTest.java

```
1. import java.io.*;
2. import java.util.*;
3. import java.util.concurrent.*;
4.
5. /**
6.  * @version 1.0 2004-08-01
7.  * @author Cay Horstmann
8.  */
9. public class FutureTest
10. {
11.     public static void main(String[] args)
12.     {
13.         Scanner in = new Scanner(System.in);
14.         System.out.print("Enter base directory (e.g. /usr/local/jdk5.0/src): ");
15.         String directory = in.nextLine();
16.         System.out.print("Enter keyword (e.g. volatile): ");
17.         String keyword = in.nextLine();
18.
19.         MatchCounter counter = new MatchCounter(new File(directory), keyword);
20.         FutureTask<Integer> task = new FutureTask<Integer>(counter);
21.         Thread t = new Thread(task);
22.         t.start();
23.         try
24.         {
25.             System.out.println(task.get() + " matching files.");
26.         }
27.         catch (ExecutionException e)
28.         {
29.             e.printStackTrace();
30.         }
31.         catch (InterruptedException e)
32.         {
33.         }
34.     }
35. }
36.
37. /**
38.  * This task counts the files in a directory and its subdirectories that contain a given keyword.
39.  */
40. class MatchCounter implements Callable<Integer>
41. {
42.     /**
43.      * Constructs a MatchCounter.
44.      * @param directory the directory in which to start the search
45.      * @param keyword the keyword to look for
46.      */
47.     public MatchCounter(File directory, String keyword)
48.     {
49.         this.directory = directory;
50.         this.keyword = keyword;
51.     }
52.
53.     public Integer call()
54.     {
55.         count = 0;
```

```
56.     try
57.     {
58.         File[] files = directory.listFiles();
59.         ArrayList<Future<Integer>> results = new ArrayList<Future<Integer>>();
60.
61.         for (File file : files)
62.             if (file.isDirectory())
63.             {
64.                 MatchCounter counter = new MatchCounter(file, keyword);
65.                 FutureTask<Integer> task = new FutureTask<Integer>(counter);
66.                 results.add(task);
67.                 Thread t = new Thread(task);
68.                 t.start();
69.             }
70.         else
71.         {
72.             if (search(file)) count++;
73.         }
74.
75.         for (Future<Integer> result : results)
76.             try
77.             {
78.                 count += result.get();
79.             }
80.             catch (ExecutionException e)
81.             {
82.                 e.printStackTrace();
83.             }
84.     }
85.     catch (InterruptedException e)
86.     {
87.     }
88.     return count;
89. }
90.
91. /**
92.  * Searches a file for a given keyword.
93.  * @param file the file to search
94.  * @return true if the keyword is contained in the file
95.  */
96. public boolean search(File file)
97. {
98.     try
99.     {
100.         Scanner in = new Scanner(new FileInputStream(file));
101.         boolean found = false;
102.         while (!found && in.hasNextLine())
103.         {
104.             String line = in.nextLine();
105.             if (line.contains(keyword)) found = true;
106.         }
107.         in.close();
108.         return found;
109.     }
110.     catch (IOException e)
111.     {
112.         return false;
113.     }
```

```
113.     }  
114. }  
115.  
116. private File directory;  
117. private String keyword;  
118. private int count;  
119. }
```

#### **API** java.util.concurrent.Callable<V> 5.0

- `V call()`  
运行一个将产生结果的任务。

#### **API** java.util.concurrent.Future<V> 5.0

- `V get()`
- `V get(long time, TimeUnit unit)`  
获取结果，如果没有结果可用，则阻塞直到真正得到结果超过指定的时间为止。如果不成功，第二个方法会抛出 `TimeoutException` 异常。
- `boolean cancel(boolean mayInterrupt)`  
尝试取消这一任务的运行。如果任务已经开始，并且 `mayInterrupt` 参数值为 `true`，它就会被中断。如果成功执行了取消操作，返回 `true`。
- `boolean isCancelled()`  
如果任务在完成前被取消了，则返回 `true`。
- `boolean isDone()`  
如果任务结束，无论是正常结束、中途取消或发生异常，都返回 `true`。

#### **API** java.util.concurrent.FutureTask<V> 5.0

- `FutureTask(Callable<V> task)`
- `FutureTask(Runnable task, V result)`  
构造一个既是 `Future<V>` 又是 `Runnable` 的对象。

## 14.9 执行器

构建一个新的线程是有一定代价的，因为涉及与操作系统的交互。如果程序中创建了大量的生命期很短的线程，应该使用线程池（thread pool）。一个线程池中包含许多准备运行的空闲线程。将 `Runnable` 对象交给线程池，就会有一个线程调用 `run` 方法。当 `run` 方法退出时，线程不会死亡，而是在池中准备为下一个请求提供服务。

另一个使用线程池的理由是减少并发线程的数目。创建大量线程会大大降低性能甚至使虚拟机崩溃。如果有一个会创建许多线程的算法，应该使用一个线程数“固定的”线程池以限制并发线程的总数。

执行器（`Executor`）类有许多静态工厂方法用来构建线程池，表14-2中对这些方法进行



了汇总。

表14-2 执行者工厂方法

方 法	描 述
<code>newCachedThreadPool</code>	必要时创建新线程；空闲线程会被保留60秒
<code>newFixedThreadPool</code>	该池包含固定数量的线程；空闲线程会一直被保留
<code>newSingleThreadExecutor</code>	只有一个线程的“池”，该线程顺序执行每一个提交的任务（类似于Swing事件分配线程）
<code>newScheduledThreadPool</code>	用于预定执行而构建的固定线程池，替代 <code>java.util.Timer</code>
<code>newSingleThreadScheduledExecutor</code>	用于预定执行而构建的单线程“池”

### 14.9.1 线程池

先来看一下表14-2中的3个方法。在第14.9.2节中，我们讨论其余的方法。`newCachedThreadPool`方法构建了一个线程池，对于每个任务，如果有空闲线程可用，立即让它执行任务，如果没有可用的空闲线程，则创建一个新线程。`newFixedThreadPool`方法构建一个具有固定大小的线程池。如果提交的任务数多于空闲的线程数，那么把得不到服务的任务放置到队列中。当其他任务完成以后再运行它们。`newSingleThreadExecutor`是一个退化了的大小为1的线程池：由一个线程执行提交的任务，一个接着一个。这3个方法返回实现了`ExecutorService`接口的`ThreadPoolExecutor`类的对象。

可用下面的方法之一将一个`Runnable`对象或`Callable`对象提交给`ExecutorService`：

```
Future<?> submit(Runnable task)
Future<T> submit(Runnable task, T result)
Future<T> submit(Callable<T> task)
```

该池会在方便的时候尽早执行提交的任务。调用`submit`时，会得到一个`Future`对象，用来查询该任务的状态。

第一个`submit`方法返回一个奇怪样子的`Future<?>`。可以使用这样一个对象来调用`isDone`、`cancel`或`isCancelled`。但是，`get`方法在完成的时候只是简单地返回`null`。

第二个版本的`Submit`也提交一个`Runnable`，并且`Future`的`get`方法在完成的时候返回指定的`result`对象。

第三个版本的`Submit`提交一个`Callable`，并且返回的`Future`对象将在计算结果准备好的时候得到它。

当用完一个线程池的时候，调用`shutdown`。该方法启动该池的关闭序列。被关闭的执行器不再接受新的任务。当所有任务都完成以后，线程池中的线程死亡。另一种方法是调用`shutdownNow`。该池取消尚未开始的所有任务并试图中断正在运行的线程。

下面总结了在使用连接池时应该做的事：

- 1) 调用`Executors`类中静态的方法`newCachedThreadPool`或`newFixedThreadPool`。
- 2) 调用`submit`提交`Runnable`或`Callable`对象。
- 3) 如果想要取消一个任务，或如果提交`Callable`对象，那就要保存好返回的`Future`对象。
- 4) 当不再提交任何任务时，调用`shutdown`。

例如，前面的程序例子产生了大量的生命期很短的线程，每个目录产生一个线程。例14-12中的程序使用了一个线程池来运行任务。

出于信息方面的考虑，这个程序打印出执行中池中最大的线程数。但是不能通过 `ExecutorService` 这个接口得到这一信息。因此，必须将该 `pool` 对象转型为 `ThreadPoolExecutor` 类对象。

#### 例14-12 ThreadPoolTest.java

```
1. import java.io.*;
2. import java.util.*;
3. import java.util.concurrent.*;
4.
5. /**
6.  * @version 1.0 2004-08-01
7.  * @author Cay Horstmann
8.  */
9. public class ThreadPoolTest
10. {
11.     public static void main(String[] args) throws Exception
12.     {
13.         Scanner in = new Scanner(System.in);
14.         System.out.print("Enter base directory (e.g. /usr/local/jdk5.0/src): ");
15.         String directory = in.nextLine();
16.         System.out.print("Enter keyword (e.g. volatile): ");
17.         String keyword = in.nextLine();
18.
19.         ExecutorService pool = Executors.newCachedThreadPool();
20.
21.         MatchCounter counter = new MatchCounter(new File(directory), keyword, pool);
22.         Future<Integer> result = pool.submit(counter);
23.
24.         try
25.         {
26.             System.out.println(result.get() + " matching files.");
27.         }
28.         catch (ExecutionException e)
29.         {
30.             e.printStackTrace();
31.         }
32.         catch (InterruptedException e)
33.         {
34.         }
35.         pool.shutdown();
36.
37.         int largestPoolSize = ((ThreadPoolExecutor) pool).getLargestPoolSize();
38.         System.out.println("largest pool size=" + largestPoolSize);
39.     }
40. }
41.
42. /**
43.  * This task counts the files in a directory and its subdirectories that contain a given keyword.
44.  */
45. class MatchCounter implements Callable<Integer>
46. {
```

```
47.  /**
48.   * Constructs a MatchCounter.
49.   * @param directory the directory in which to start the search
50.   * @param keyword the keyword to look for
51.   * @param pool the thread pool for submitting subtasks
52.   */
53.  public MatchCounter(File directory, String keyword, ExecutorService pool)
54.  {
55.      this.directory = directory;
56.      this.keyword = keyword;
57.      this.pool = pool;
58.  }
59.
60.  public Integer call()
61.  {
62.      count = 0;
63.      try
64.      {
65.          File[] files = directory.listFiles();
66.          ArrayList<Future<Integer>> results = new ArrayList<Future<Integer>>();
67.
68.          for (File file : files)
69.              if (file.isDirectory())
70.              {
71.                  MatchCounter counter = new MatchCounter(file, keyword, pool);
72.                  Future<Integer> result = pool.submit(counter);
73.                  results.add(result);
74.              }
75.          else
76.          {
77.              if (search(file)) count++;
78.          }
79.
80.          for (Future<Integer> result : results)
81.              try
82.              {
83.                  count += result.get();
84.              }
85.              catch (ExecutionException e)
86.              {
87.                  e.printStackTrace();
88.              }
89.      }
90.      catch (InterruptedException e)
91.      {
92.      }
93.      return count;
94.  }
95.
96.  /**
97.   * Searches a file for a given keyword.
98.   * @param file the file to search
99.   * @return true if the keyword is contained in the file
100.   */
101.  public boolean search(File file)
102.  {
103.      try
```

```
104.     {
105.         Scanner in = new Scanner(new FileInputStream(file));
106.         boolean found = false;
107.         while (!found && in.hasNextLine())
108.         {
109.             String line = in.nextLine();
110.             if (line.contains(keyword)) found = true;
111.         }
112.         in.close();
113.         return found;
114.     }
115.     catch (IOException e)
116.     {
117.         return false;
118.     }
119. }
120.
121. private File directory;
122. private String keyword;
123. private ExecutorService pool;
124. private int count;
125. }
```

#### **API** java.util.concurrent.Executors 5.0

- `ExecutorService newCachedThreadPool()`  
返回一个带缓存的线程池，该池在必要的时候创建线程，在线程空闲60秒之后终止线程。
- `ExecutorService newFixedThreadPool(int threads)`  
返回一个线程池，该池中的线程数由参数指定。
- `ExecutorService newSingleThreadExecutor()`  
返回一个执行器，它在一个单独的线程中依次执行各个任务。

#### **API** java.util.concurrent.ExecutorService 5.0

- `Future<T> submit(Callable<T> task)`
- `Future<T> submit(Runnable task, T result)`
- `Future<?> submit(Runnable task)`  
提交指定的任务去执行。
- `void shutdown()`  
关闭服务，会先完成已经提交的任务而不再接收新的任务。

#### **API** java.util.concurrent.ThreadPoolExecutor 5.0

- `int getLargestPoolSize()`  
返回线程池在该执行器生命周期中的最大尺寸。

### 14.9.2 预定执行

`ScheduledExecutorService`接口具有为预定执行（Scheduled Execution）或重复执行任务而设计的方法。它是一种允许使用线程池机制的`java.util.Timer`的泛化。`Executors`类的`newScheduledThreadPool`和`newSingleThreadScheduledExecutor`方法将返回实现了`ScheduledExecutorService`接口的对象。

可以预定`Runnable`或`Callable`在初始的延迟之后只运行一次。也可以预定一个`Runnable`对象周期性地运行。详细内容见API文档。

**API** `java.util.concurrent.Executors 5.0`

- `ScheduledExecutorService newScheduledThreadPool(int threads)`  
返回一个线程池，它使用给定的线程数来调度任务。
- `ScheduledExecutorService newSingleThreadScheduledExecutor()`  
返回一个执行器，它在一个单独线程中调度任务。

**API** `java.util.concurrent.ScheduledExecutorService 5.0`

- `ScheduledFuture<V> schedule(Callable<V> task, long time, TimeUnit unit)`
- `ScheduledFuture<?> schedule(Runnable task, long time, TimeUnit unit)`  
预定在指定的时间之后执行任务。
- `ScheduledFuture<?> scheduleAtFixedRate(Runnable task, long initialDelay, long period, TimeUnit unit)`  
预定在初始的延迟结束后，周期性地运行给定的任务，周期长度是`period`。
- `ScheduledFuture<?> scheduleWithFixedDelay(Runnable task, long initialDelay, long delay, TimeUnit unit)`  
预定在初始的延迟结束后周期性地给定的任务，在一次调用完成和下一次调用开始之间有长度为`delay`的延迟。

### 14.9.3 控制任务组

你已经了解了如何将一个执行器服务作为线程池使用，以提高执行任务的效率。有时，使用执行器有更有实际意义的原因，控制一组相关任务。例如，可以在执行器中使用`shutdownNow`方法取消所有的任务。

`invokeAny`方法提交所有对象到一个`Callable`对象的集合中，并返回某个已经完成了的任务的结果。无法知道返回的究竟是哪个任务的结果，也许是最先完成的那个任务的结果。对于搜索问题，如果你愿意接受任何一种解决方案的话，你就可以使用这个方法。例如，假定你需要对一个大整数进行因数分解计算来解码RSA密码。可以提交很多任务，每一个任务使用不同范围内的数来进行分解。只要其中一个任务得到了答案，计算就可以停止了。

`invokeAll`方法提交所有对象到一个`Callable`对象的集合中，并返回一个`Future`对象的列表，代表所有任务的解决方案。当计算结果可获得时，可以像下面这样对结果进行处理：

```
List<Callable<T>> tasks = . . .;
List<Future<T>> results = executor.invokeAll(tasks);
for (Future<T> result : results)
    processFurther(result.get());
```

这个方法的缺点是如果第一个任务恰巧花去了很多时间，则可能不得不进行等待。以结果按可获得的顺序保存起来更有实际意义。可以用ExecutorCompletionService来进行排列。

用常规的方法获得一个执行器。然后，构建一个ExecutorCompletionService，提交任务给完成服务（completion service）。该服务管理Future对象的阻塞队列，其中包含已经提交的任务的执行结果（当这些结果成为可用时）。这样一来，相比前面的计算，一个更有效的组织形式如下：

```
ExecutorCompletionService service = new ExecutorCompletionService(executor);
for (Callable<T> task : tasks) service.submit(task);
for (int i = 0; i < tasks.size(); i++)
    processFurther(service.take().get());
```

#### **API** java.util.concurrent.ExecutorService 5.0

- T invokeAny(Collection<Callable<T>> tasks)
- T invokeAny(Collection<Callable<T>> tasks, long timeout, TimeUnit unit)  
执行给定的任务，返回其中一个任务的结果。第二个方法若发生超时，抛出一个TimeoutException异常。
- List<Future<T>> invokeAll(Collection<Callable<T>> tasks)
- List<Future<T>> invokeAll(Collection<Callable<T>> tasks, long timeout, TimeUnit unit)  
执行给定的任务，返回所有任务的结果。第二个方法若发生超时，抛出一个TimeoutException异常。

#### **API** java.util.concurrent.ExecutorCompletionService 5.0

- ExecutorCompletionService(Executor e)  
构建一个执行器完成服务来收集给定执行器的结果。
- Future<T> submit(Callable<T> task)
- Future<T> submit(Runnable task, T result)  
提交一个任务给底层的执行器。
- Future<T> take()  
移除下一个已完成的结果，如果没有任何已完成的结果可用则阻塞。
- Future<T> poll()
- Future<T> poll(long time, TimeUnit unit)  
移除下一个已完成的结果，如果没有任何已完成结果可用则返回null。第二个方法将等待给定的时间。

## 14.10 同步器

java.util.concurrent包包含了几个能帮助人们管理相互合作的线程集类见表14-3。这些机制具有为线程之间的共用集结点模式（common rendezvous patterns）提供的“预置功能”（canned functionality）。如果有一个相互合作的线程集满足这些行为模式之一，那么应该直接重用合适的库类而不要试图提供手工的锁与条件的集合。

表14-3 同步器

类	它能做什么	何时使用
CyclicBarrier	允许线程集等待直至其中预定数目的线程到达一个公共障栅（barrier），然后可以选择执行一个处理障栅的动作	当大量的线程需要在它们的结果可用之前完成时
CountDownLatch	允许线程集等待直到计数器减为0	当一个或多个线程需要等待直到指定数目的事件发生
Exchanger	允许两个线程在要交换的对象准备好时交换对象	当两个线程工作在同一数据结构两个实例上的时候，一个向实例添加数据而另一个从实例清除数据
Semaphore	允许线程集等待直到被允许继续运行为止	限制访问资源的线程总数。如果许可数是1，常常阻塞线程直到另一个线程给出许可为止
SynchronousQueue	允许一个线程把对象交给另一个线程	在没有显式同步的情况下，当两个线程准备好将一个对象从一个线程传递到另一个时

### 14.10.1 信号量

概念上讲，一个信号量管理许多的许可证（permits）。为了通过信号量，线程通过调用acquire请求许可。许可的数目是固定的，由此限制了通过的线程数量。其他线程可以通过调用release释放许可。其实没有实际的许可对象，信号量仅维护一个计数。而且，许可不是必须由获取它的线程释放。事实上，任何线程都可以释放任意数目的许可。如果释放的许可多于可用许可的最大数目，信号量只是被设置为可用许可的最大数目。这种随意性使得信号量既具有灵活性又容易带来混乱。

信号量在1968年由Edsger Dijkstra发明，作为同步原语（synchronization primitive）。Dijkstra指出信号量可以被有效地实现，并且有足够的的能力解决许多常见的线程同步问题。在几乎任何一本操作系统教科书中，都能看到使用信号量实现的有界队列。当然，应用程序员不必自己实现有界队列。建议在信号量的行为能适合你面对的同步问题时才使用它，否则你会陷入思维的混乱。

一个简单的例子，一个许可数为1的信号量作为一个可由其他线程打开或关闭的门很有用。在第14.10.6节有这样的例子，工作器线程创建动画。偶尔，工作器线程等待用户按下一个按钮。工作器线程试图获得一个许可，并且它不得不等待直到按钮点击释放一个许可。

### 14.10.2 倒计时门栓

一个倒计时门栓（CountDownLatch）让一个线程集等待直到计数变为0。倒计时门栓是一

次性的。一旦计数为0，就不能再重用了。

一个有用的特例是计数值为1的门栓。实现一个只能通过一次的门。线程在门外等候直到另一个线程将计数器值置为0。

举例，假定一个线程集需要一些初始的数据来完成工作。工作器线程被启动并在门外等候。另一个线程准备数据。当数据准备好的时候，调用countDown，所有工作器线程就可以继续运行了。

然后，可以使用第二个门栓检查什么时候所有工作器线程完成工作。用线程数初始化门栓。每个工作器线程在结束前将门栓计数减1。另一个获取工作结果的线程在门外等待，一旦所有工作器线程终止该线程继续运行。

### 14.10.3 障栅

CyclicBarrier类实现了一个集结点（rendezvous）称为障栅（barrier）。考虑大量线程运行在一次计算的不同部分的情形。当所有部分都准备好时，需要把结果组合在一起。当一个线程完成了它的那部分任务后，我们让它运行到障栅处。一旦所有的线程都到达了这个障栅，障栅就撤销，线程就可以继续运行。

下面是其细节。首先，构造一个障栅，并给出参与的线程数：

```
CyclicBarrier barrier = new CyclicBarrier(nthreads);
```

每一个线程做一些工作，完成后在障栅上调用await：

```
public void run()
{
    doWork();
    barrier.await();
    . . .
}
```

await方法有一个可选的超时参数：

```
barrier.await(100, TimeUnit.MILLISECONDS);
```

如果任何一个在障栅上等待的线程离开了障栅，那么障栅就被破坏了（线程可能离开是因为它调用await时设置了超时，或者因为它被中断了）。在这种情况下，所有其他线程的await方法抛出BrokenBarrierException异常。那些已经在等待的线程立即终止await的调用。

可以提供一个可选的障栅动作（barrier action），当所有线程到达障栅的时候就会执行这一动作。

```
Runnable barrierAction = . . . ;
CyclicBarrier barrier = new CyclicBarrier(nthreads, barrierAction);
```

该动作可以收集那些单个线程的运行结果。

障栅被称为是循环的（cyclic），因为可以在所有等待线程被释放后被重用。在这一点上，有别于CountDownLatch，CountDownLatch只能被使用一次。

### 14.10.4 交换器

当两个线程在同一个数据缓冲区的两个实例上工作的时候，就可以使用交换器（Exchanger）。



典型的情况是，一个线程向缓冲区填入数据，另一个线程消耗这些数据。当它们都完成以后，相互交换缓冲区。

#### 14.10.5 同步队列

同步队列是一种将生产者与消费者线程配对的机制。当一个线程调用SynchronousQueue的put方法时，它会阻塞直到另一个线程调用take方法为止，反之亦然。与Exchanger的情况不同，数据仅仅沿一个方向传递，从生产者到消费者。

即使SynchronousQueue类实现了BlockingQueue接口，概念上讲，它依然不是一个队列。它没有包含任何元素，它的size方法总是返回0。

#### 14.10.6 例子：暂停动画与恢复动画

考虑一个要做某项工作的程序，更新屏幕的显示，在等待用户查看结果之后按下按钮继续，然后完成工作的下一步。

许可计数为1的信号量可以用来处理工作器线程和事件分配线程之间的同步。工作器线程在准备好要暂停的时候调用acquire。只要用户点击Continue按钮，GUI线程就调用release。

如果在工作器线程准备好的时候，用户多次点击该按钮会发生什么呢？毕竟因为只有一个许可证可用，许可计数为1。

例14-13中的程序使用这种思路工作。程序以动画形式展示排序算法。工作器线程对数组进行排序，周期性地停止，等待用户继续给出许可。用户看到算法当前状态的绘图会感到满意，并按下Continue按钮允许工作器线程进行下一步处理。

这里不打算让读者为排序算法烦恼，于是，调用Arrays.sort，它实现归并算法。要暂停该算法，我们提供了一个等待信号量的Comparator对象。因此，每当算法比较两个元素的时候，暂停动画。绘制数组的当前值并加亮显示被比较的元素（见图14-8）。



图14-8 动画演示排序算法



注释：动画展示了较小的有序段归并到较大的有序段，但是它并不那么精确。mergesort算法使用第二个数组存放我们无法看到的临时值。这个例子的要点是不要过于用心地钻研排序算法，而是展示如何使用信号量用来暂停工作器线程。

#### 例14-13 AlgorithmAnimation.java

```
1. import java.awt.*;
2. import java.awt.geom.*;
3. import java.awt.event.*;
4. import java.util.*;
5. import java.util.concurrent.*;
6. import javax.swing.*;
7.
8. /**
9.  * This program animates a sort algorithm.
10.  * @version 1.01 2007-05-18
```

```
11. * @author Cay Horstmann
12. */
13. public class AlgorithmAnimation
14. {
15.     public static void main(String[] args)
16.     {
17.         EventQueue.invokeLater(new Runnable()
18.         {
19.             public void run()
20.             {
21.                 JFrame frame = new AnimationFrame();
22.                 frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
23.                 frame.setVisible(true);
24.             }
25.         });
26.     }
27. }
28.
29. /**
30.  * This frame shows the array as it is sorted, together with buttons to single-step the
31.  * animation or to run it without interruption.
32.  */
33. class AnimationFrame extends JFrame
34. {
35.     public AnimationFrame()
36.     {
37.         ArrayComponent comp = new ArrayComponent();
38.         add(comp, BorderLayout.CENTER);
39.
40.         final Sorter sorter = new Sorter(comp);
41.
42.         JButton runButton = new JButton("Run");
43.         runButton.addActionListener(new ActionListener()
44.         {
45.             public void actionPerformed(ActionEvent event)
46.             {
47.                 sorter.setRun();
48.             }
49.         });
50.
51.         JButton stepButton = new JButton("Step");
52.         stepButton.addActionListener(new ActionListener()
53.         {
54.             public void actionPerformed(ActionEvent event)
55.             {
56.                 sorter.setStep();
57.             }
58.         });
59.
60.         JPanel buttons = new JPanel();
61.         buttons.add(runButton);
62.         buttons.add(stepButton);
63.         add(buttons, BorderLayout.NORTH);
64.         setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
65.
66.         Thread t = new Thread(sorter);
67.         t.start();
```

```
68.     }
69.
70.     private static final int DEFAULT_WIDTH = 300;
71.     private static final int DEFAULT_HEIGHT = 300;
72. }
73.
74. /**
75.  * This runnable executes a sort algorithm. When two elements are compared, the algorithm
76.  * pauses and updates a component.
77.  */
78. class Sorter implements Runnable
79. {
80.     /**
81.      * Constructs a Sorter.
82.      * @param values the array to be sorted
83.      * @param comp the component on which to display the sorting progress
84.      */
85.     public Sorter(ArrayComponent comp)
86.     {
87.         values = new Double[VALUES_LENGTH];
88.         for (int i = 0; i < values.length; i++)
89.             values[i] = new Double(Math.random());
90.         this.component = comp;
91.         this.gate = new Semaphore(1);
92.         this.run = false;
93.     }
94.
95.     /**
96.      * Sets the sorter to "run" mode. Called on the event dispatch thread.
97.      */
98.     public void setRun()
99.     {
100.         run = true;
101.         gate.release();
102.     }
103.
104.     /**
105.      * Sets the sorter to "step" mode. Called on the event dispatch thread.
106.      */
107.     public void setStep()
108.     {
109.         run = false;
110.         gate.release();
111.     }
112.
113.     public void run()
114.     {
115.         Comparator<Double> comp = new Comparator<Double>()
116.         {
117.             public int compare(Double i1, Double i2)
118.             {
119.                 component.setValues(values, i1, i2);
120.                 try
121.                 {
122.                     if (run) Thread.sleep(DELAY);
123.                     else gate.acquire();
124.                 }
```

```

125.         catch (InterruptedException exception)
126.         {
127.             Thread.currentThread().interrupt();
128.         }
129.         return i1.compareTo(i2);
130.     }
131. };
132.     Arrays.sort(values, comp);
133.     component.setValues(values, null, null);
134. }
135.
136. private Double[] values;
137. private ArrayComponent component;
138. private Semaphore gate;
139. private static final int DELAY = 100;
140. private volatile boolean run;
141. private static final int VALUES_LENGTH = 30;
142. }
143.
144. /**
145.  * This component draws an array and marks two elements in the array.
146.  */
147. class ArrayComponent extends JComponent
148. {
149.     /**
150.      * Sets the values to be painted. Called on the sorter thread.
151.      * @param values the array of values to display
152.      * @param marked1 the first marked element
153.      * @param marked2 the second marked element
154.      */
155.     public synchronized void setValues(Double[] values, Double marked1, Double marked2)
156.     {
157.         this.values = values.clone();
158.         this.marked1 = marked1;
159.         this.marked2 = marked2;
160.         repaint();
161.     }
162.
163.     public synchronized void paintComponent(Graphics g) // Called on the event dispatch thread
164.     {
165.         if (values == null) return;
166.         Graphics2D g2 = (Graphics2D) g;
167.         int width = getWidth() / values.length;
168.         for (int i = 0; i < values.length; i++)
169.         {
170.             double height = values[i] * getHeight();
171.             Rectangle2D bar = new Rectangle2D.Double(width * i, 0, width, height);
172.             if (values[i] == marked1 || values[i] == marked2) g2.fill(bar);
173.             else g2.draw(bar);
174.         }
175.     }
176.
177.     private Double marked1;
178.     private Double marked2;
179.     private Double[] values;
180. }

```

**API** java.util.concurrent.CyclicBarrier 5.0

- CyclicBarrier(int parties)  
构建一个线程数目是parties的循环障碍。当所有的线程都在障碍上调用await之后，执行barrierAction。
- CyclicBarrier(int parties, Runnable barrierAction)
- int await()  
等待直到所有的线程在障碍上调用await或者时间超时为止，在这种情况下会抛出TimeoutException异常。成功时，返回这个线程的序号。第一个线程的序号为parties - 1，最后一个线程是0。
- int await(long time, TimeUnit unit)

**API** java.util.concurrent.CountDownLatch 5.0

- CountDownLatch(int count)  
用给定的计数构建一个倒计时门栓。
- void await()  
等待这个门栓的计数降为0。
- boolean await(long time, TimeUnit unit)  
等待这个门栓的计数降为0或者时间超时。如果计数为0返回true，如果超时返回false。
- public void countDown()  
递减这个门栓的计数值。

**API** java.util.concurrent.Exchanger<V> 5.0

- V exchange(V item)
- V exchange(V item, long time, TimeUnit unit)  
阻塞直到另一个线程调用这个方法，然后，同其他线程交换item，并返回其他线程的item。第二个方法时间超时时，抛出TimeoutException异常。

**API** java.util.concurrent.SynchronousQueue<V> 5.0

- SynchronousQueue()
- SynchronousQueue(boolean fair)  
构建一个允许线程提交item的同步队列。如果fair为true，队列优先照顾等待了最长时间的线程。
- void put(V item)  
阻塞直到另一个线程调用take来获取item。
- V take()  
阻塞直到另一个线程调用put。返回另一个线程提供的item。

**API** java.util.concurrent.Semaphore 5.0

- Semaphore(int permits)
- Semaphore(int permits, boolean fair)  
用给定的许可数目为最大值构建一个信号量。如果 fair为true，队列优先照顾等待了最长时间的线程。
- void acquire()  
等待获得一个许可。
- boolean tryAcquire()  
尝试获得一个许可，如果没有许可是可用的，返回false。
- boolean tryAcquire(long time, TimeUnit unit)  
尝试在给定时间内获得一个许可；如果没有许可是可用的，返回false。
- void release()  
释放一个许可。

### 14.11 线程与Swing

在有关本章的介绍里已经提到，在程序中使用线程的理由之一是提高程序的响应性能。当程序需要做某些耗时的工作时，应该启动另一个工作器线程而不是阻塞用户接口。

但是，必须认真考虑工作器线程在做什么，因为这或许令人惊讶，Swing不是线程安全的。如果你试图在多个线程中操纵用户界面的元素，那么用户界面可能崩溃。

要了解这一问题，运行例14-14的测试程序。当你点击Bad按钮时，一个新的线程将启动，它的run方法操作一个组合框，随机地添加值和删除值。

```
public void run()
{
    try
    {
        while (true)
        {
            int i = Math.abs(generator.nextInt());
            if (i % 2 == 0)
                combo.insertItemAt(new Integer(i), 0);
            else if (combo.getItemCount() > 0)
                combo.removeItemAt(i % combo.getItemCount());
            sleep(1);
        }
        catch (InterruptedException e) {}
    }
}
```

试试看。点击Bad按钮。点击几次组合框，移动滚动条，移动窗口，再次点击Bad按钮，不断点击组合框。最终，你会看到一个异常报告（见图14-9）。

发生了什么？当把一个元素插入组合框时，组合框将产生一个事件来更新显示。然后，显示代码开始运行，读取组合框的当前大小并准备显示这个值。但是，工作器线程保持运行，有



要解决这一问题，在任何线程中，可以使用两种有效的方法向事件队列添加任意的动作。例如，假定想在一个线程中周期性地更新标签来表明进度。不可以从自己的线程中调用 `label.setText`，而应该使用 `EventQueue` 类的 `invokeLater` 方法和 `invokeAndWait` 方法使所调用的方法在事件分配线程中执行。

应该将 `Swing` 代码放置到实现 `Runnable` 接口的类的 `run` 方法中。然后，创建该类的一个对象，将其传递给静态的 `invokeLater` 或 `invokeAndWait` 方法。例如，下面是如何更新标签内容的代码：

```
EventQueue.invokeLater(new
    Runnable()
    {
        public void run()
        {
            label.setText(percentage + "% complete");
        }
    });
```

当事件放入事件队列时，`invokeLater` 方法立即返回，而 `run` 方法被异步执行。`invokeAndWait` 方法等待直到 `run` 方法确被实执行过为止。

在更新进度标签时，`invokeLater` 方法更适宜。用户更希望让工作器线程有更快完成工作而不是得到更加精确的进度指示器。

这两种方法都是在事件分配线程中执行 `run` 方法。没有新的线程被创建。

例14-14演示了如何使用 `invokeLater` 方法安全地修改组合框的内容。如果点击 `Good` 按钮，线程插入或删除数字。但是，实际的修改是发生在事件分配线程中。

#### 例14-14 `SwingThreadTest.java`

```
1. import java.awt.*;
2. import java.awt.event.*;
3. import java.util.*;
4. import javax.swing.*;
5.
6. /**
7.  * This program demonstrates that a thread that runs in parallel with the event dispatch thread
8.  * can cause errors in Swing components.
9.  * @version 1.23 2007-05-17
10.  * @author Cay Horstmann
11.  */
12. public class SwingThreadTest
13. {
14.     public static void main(String[] args)
15.     {
16.         EventQueue.invokeLater(new Runnable()
17.         {
18.             public void run()
19.             {
20.                 SwingThreadFrame frame = new SwingThreadFrame();
21.                 frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
22.                 frame.setVisible(true);
23.             }
24.         });
25.     }
```



```
26. }
27.
28. /**
29.  * This frame has two buttons to fill a combo box from a separate thread. The "Good" button
30.  * uses the event queue; the "Bad" button modifies the combo box directly.
31.  */
32. class SwingThreadFrame extends JFrame
33. {
34.     public SwingThreadFrame()
35.     {
36.         setTitle("SwingThreadTest");
37.
38.         final JComboBox combo = new JComboBox();
39.         combo.insertItemAt(Integer.MAX_VALUE, 0);
40.         combo.setPrototypeDisplayValue(combo.getItemAt(0));
41.         combo.setSelectedIndex(0);
42.
43.         JPanel panel = new JPanel();
44.
45.         JButton goodButton = new JButton("Good");
46.         goodButton.addActionListener(new ActionListener()
47.         {
48.             public void actionPerformed(ActionEvent event)
49.             {
50.                 new Thread(new GoodWorkerRunnable(combo)).start();
51.             }
52.         });
53.         panel.add(goodButton);
54.         JButton badButton = new JButton("Bad");
55.         badButton.addActionListener(new ActionListener()
56.         {
57.             public void actionPerformed(ActionEvent event)
58.             {
59.                 new Thread(new BadWorkerRunnable(combo)).start();
60.             }
61.         });
62.         panel.add(badButton);
63.
64.         panel.add(combo);
65.         add(panel);
66.         pack();
67.     }
68. }
69.
70. /**
71.  * This runnable modifies a combo box by randomly adding and removing numbers. This can result
72.  * in errors because the combo box methods are not synchronized and both the worker thread
73.  * and the event dispatch thread access the combo box.
74.  */
75. class BadWorkerRunnable implements Runnable
76. {
77.     public BadWorkerRunnable(JComboBox aCombo)
78.     {
79.         combo = aCombo;
80.         generator = new Random();
81.     }
82.
```

```
83.     public void run()
84.     {
85.         try
86.         {
87.             while (true)
88.             {
89.                 int i = Math.abs(generator.nextInt());
90.                 if (i % 2 == 0) combo.insertItemAt(i, 0);
91.                 else if (combo.getItemCount() > 0) combo.removeItemAt(i % combo.getItemCount());
92.                 Thread.sleep(1);
93.             }
94.         }
95.         catch (InterruptedException e)
96.         {
97.         }
98.     }
99.
100.     private JComboBox combo;
101.     private Random generator;
102. }
103.
104. /**
105.  * This runnable modifies a combo box by randomly adding and removing numbers. In order to
106.  * ensure that the combo box is not corrupted, the editing operations are forwarded to the
107.  * event dispatch thread.
108.  */
109. class GoodWorkerRunnable implements Runnable
110. {
111.     public GoodWorkerRunnable(JComboBox aCombo)
112.     {
113.         combo = aCombo;
114.         generator = new Random();
115.     }
116.
117.     public void run()
118.     {
119.         try
120.         {
121.             while (true)
122.             {
123.                 EventQueue.invokeLater(new Runnable()
124.                 {
125.                     public void run()
126.                     {
127.                         int i = Math.abs(generator.nextInt());
128.                         if (i % 2 == 0) combo.insertItemAt(i, 0);
129.                         else if (combo.getItemCount() > 0) combo.removeItemAt(i
130.                             % combo.getItemCount());
131.                     }
132.                 });
133.                 Thread.sleep(1);
134.             }
135.         }
136.         catch (InterruptedException e)
137.         {
138.         }
139.     }
```

```

140.
141.     private JComboBox combo;
142.     private Random generator;
143. }

```

#### API java.awt.EventQueue 1.1

- static void invokeLater(Runnable runnable) 1.2  
在待处理的线程被处理之后，让runnable对象的run方法在事件分配线程中执行。
- static void invokeAndWait(Runnable runnable) 1.2  
在待处理的线程被处理之后，让runnable对象的run方法在事件分配线程中执行。该调用会阻塞，直到run方法终止。
- static boolean isDispatchThread() 1.2  
如果执行这一方法的线程是事件分配线程，返回true。

#### 14.11.2 使用Swing工作器

当用户发布一条处理过程很耗时的命令时，你可能打算启动一个新的线程来完成这个工作。如同上一节介绍的那样，线程应该使用EventQueue.invokeLater方法来更新用户界面。

有人已经创建了很方便的类，让人们轻松地完成这样的任务，并且，这些类中的一个已经编入Java SE 6。本节我们介绍SwingWorker类。

例14-15中的程序有加载文本文件的命令和取消加载过程的命令。应该用一个长的文件来测试这个程序，例如The Count of Monte Cristo的全文，它在本书的附赠代码的gutenberg目录下。该文件在一个单独的线程中加载。在读取文件的过程中，Open菜单项被禁用，Cancel菜单项为可用（见图14-10）。读取每一行后，状态条中的线性计数器被更新。读取过程完成之后，Open菜单项重新变为可用，Cancel项被禁用，状态行文本置为Done。

这个例子展示了后台任务的典型UI活动：

- 在每一个工作单位完成之后，更新UI来显示进度。
- 整个工作完成之后，对UI做最后的更新。

SwingWorker类使得实现这一任务轻而易举。覆盖doInBackground方法来完成耗时的任务，不时地调用publish来报告工作进度。这一方法在工作器线程中执行。publish方法使得process方法在事件分配线程中执行来处理进度数据。当工作完成时，done方法在事件分配线程中被调用以便完成UI的更新。

每当要在工作器线程中做一些工作时，构建一个新的工作器（每一个工作器对象只能被使用一次）。然后调用execute方法。典型的方式是在事件分配线程中调用execute，但没有这样的需求。



图14-10 在独立线程中加载文件

假定工作器产生某种类型的结果；因此，`SwingWorker<T, V>`实现`Future<T>`。这一结果可以通过`Future`接口的`get`方法获得。由于`get`方法阻塞直到结果成为可用，因此不要在调用`execute`之后马上调用它。只在已经知道工作完成时调用它，是最为明智的。典型地，可以从`done`方法调用`get`。（有时，没有调用`get`的需求，处理进度数据就是你所需要的。）

中间的进度数据以及最终的结果可以是任何类型。`SwingWorker`类有3种类型作为类型参数。`SwingWorker<T, V>`产生类型为`T`的结果以及类型为`V`的进度数据。

要取消正在进行的工作，使用`Future`接口的`cancel`方法。当该工作被取消的时候，`get`方法抛出`CancellationException`异常。

正如前面已经提到的，工作器线程对`publish`的调用会导致在事件分配线程上的`process`的调用。为了提高效率，几个对`publish`的调用结果，可用对`process`的一次调用成批处理。`process`方法接收一个包含所有中间结果的列表`<V>`。

把这一机制用于读取文本文件的工作中。正如所看到的，`JTextArea`相当慢。在一个长的文本文件（比如，*The Count of Monte Cristo*）中追加行会花费相当可观的时间。

为了向用户展示进度，要在状态行中显示读入的行数。因此，进度数据包含当前行号以及文本的当前行。将它们打包到一个普通的内部类中：

```
private class ProgressData
{
    public int number;
    public String line;
}
```

最后的结果是已经读入`StringBuilder`的文本。因此，需要一个`SwingWorker<StringBuilder, ProgressData>`。

在`doInBackground`方法中，读取一个文件，每次一行。在读取每一行之后，调用`publish`方法发布行号和当前行的文本。

```
@Override public StringBuilder doInBackground() throws IOException, InterruptedException
{
    int lineNumber = 0;
    Scanner in = new Scanner(new FileInputStream(file));
    while (in.hasNextLine())
    {
        String line = in.nextLine();
        lineNumber++;
        text.append(line);
        text.append("\n");
        ProgressData data = new ProgressData();
        data.number = lineNumber;
        data.line = line;
        publish(data);
        Thread.sleep(1); // to test cancellation; no need to do this in your programs
    }
    return text;
}
```

在读取每一行之后休眠1毫秒，以便不使用重读就可以检测取消动作，但是，不要使用休眠来减慢程序的执行速度。如果对这一行加注解，会发现*The Count of Monte Cristo* 的加载相

当快，只有几批用户接口更新。



注释：从工作器线程来更新文本区可以使这个程序的处理相当顺畅，但是，对大多数 Swing 组件来说不可能做到这一点。这里，给出一种通用的方法，其中所有组件的更新都出现在事件分配线程中。

在这个 process 方法中，忽略除最后一行行号之外的所有行号，然后，我们把所有的行拼接在一起用于文本区的一次更新。

```
@Override public void process(List<ProgressData> data)
{
    if (isCancelled()) return;
    StringBuilder b = new StringBuilder();
    statusLine.setText("" + data.get(data.size() - 1).number);
    for (ProgressData d : data) { b.append(d.line); b.append("\n"); }
    textArea.append(b.toString());
}
```

在 done 方法中，文本区被更新为完整的文本，并且 Cancel 菜单项被禁用。

在 Open 菜单项的事件监听器中，工作器是如何启动的。这一简单的技术允许人们在保持对用户界面的正常响应的同时，执行耗时的任务。

#### 例14-15 SwingWorkerTest.java

```
1. import java.awt.*;
2. import java.awt.event.*;
3. import java.io.*;
4. import java.util.*;
5. import java.util.List;
6. import java.util.concurrent.*;
7.
8. import javax.swing.*;
9.
10. /**
11.  * This program demonstrates a worker thread that runs a potentially time-consuming task.
12.  * @version 1.1 2007-05-18
13.  * @author Cay Horstmann
14.  */
15. public class SwingWorkerTest
16. {
17.     public static void main(String[] args) throws Exception
18.     {
19.         EventQueue.invokeLater(new Runnable()
20.         {
21.             public void run()
22.             {
23.                 JFrame frame = new SwingWorkerFrame();
24.                 frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
25.                 frame.setVisible(true);
26.             }
27.         });
28.     }
29. }
30.
```

```

31. /**
32.  * This frame has a text area to show the contents of a text file, a menu to open a file and
33.  * cancel the opening process, and a status line to show the file loading progress.
34.  */
35. class SwingWorkerFrame extends JFrame
36. {
37.     public SwingWorkerFrame()
38.     {
39.         chooser = new JFileChooser();
40.         chooser.setCurrentDirectory(new File("."));
41.
42.         textArea = new JTextArea();
43.         add(new JScrollPane(textArea));
44.         setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
45.
46.         statusLine = new JLabel(" ");
47.         add(statusLine, BorderLayout.SOUTH);
48.
49.         JMenuBar menuBar = new JMenuBar();
50.         setJMenuBar(menuBar);
51.
52.         JMenu menu = new JMenu("File");
53.         menuBar.add(menu);
54.
55.         openItem = new JMenuItem("Open");
56.         menu.add(openItem);
57.         openItem.addActionListener(new ActionListener()
58.         {
59.             public void actionPerformed(ActionEvent event)
60.             {
61.                 // show file chooser dialog
62.                 int result = chooser.showOpenDialog(null);
63.
64.                 // if file selected, set it as icon of the label
65.                 if (result == JFileChooser.APPROVE_OPTION)
66.                 {
67.                     textArea.setText("");
68.                     openItem.setEnabled(false);
69.                     textReader = new TextReader(chooser.getSelectedFile());
70.                     textReader.execute();
71.                     cancelItem.setEnabled(true);
72.                 }
73.             }
74.         });
75.
76.         cancelItem = new JMenuItem("Cancel");
77.         menu.add(cancelItem);
78.         cancelItem.setEnabled(false);
79.         cancelItem.addActionListener(new ActionListener()
80.         {
81.             public void actionPerformed(ActionEvent event)
82.             {
83.                 textReader.cancel(true);
84.             }
85.         });
86.     }

```

```
87.
88. private class ProgressData
89. {
90.     public int number;
91.     public String line;
92. }
93.
94. private class TextReader extends SwingWorker<StringBuilder, ProgressData>
95. {
96.     public TextReader(File file)
97.     {
98.         this.file = file;
99.     }
100.
101.     // the following method executes in the worker thread; it doesn't touch Swing components
102.
103.     @Override
104.     public StringBuilder doInBackground() throws IOException, InterruptedException
105.     {
106.         int lineNumber = 0;
107.         Scanner in = new Scanner(new FileInputStream(file));
108.         while (in.hasNextLine())
109.         {
110.             String line = in.nextLine();
111.             lineNumber++;
112.             text.append(line);
113.             text.append("\n");
114.             ProgressData data = new ProgressData();
115.             data.number = lineNumber;
116.             data.line = line;
117.             publish(data);
118.             Thread.sleep(1); // to test cancellation; no need to do this in your programs
119.         }
120.         return text;
121.     }
122.
123.     // the following methods execute in the event dispatch thread
124.
125.     @Override
126.     public void process(List<ProgressData> data)
127.     {
128.         if (isCancelled()) return;
129.         StringBuilder b = new StringBuilder();
130.         statusLine.setText("" + data.get(data.size() - 1).number);
131.         for (ProgressData d : data)
132.         {
133.             b.append(d.line);
134.             b.append("\n");
135.         }
136.         textArea.append(b.toString());
137.     }
138.
139.     @Override
140.     public void done()
141.     {
142.         try
143.         {
```

```

144.         StringBuilder result = get();
145.         textArea.setText(result.toString());
146.         statusLine.setText("Done");
147.     }
148.     catch (InterruptedException ex)
149.     {
150.     }
151.     catch (CancellationException ex)
152.     {
153.         textArea.setText("");
154.         statusLine.setText("Cancelled");
155.     }
156.     catch (ExecutionException ex)
157.     {
158.         statusLine.setText("" + ex.getCause());
159.     }
160.
161.     cancelItem.setEnabled(false);
162.     openItem.setEnabled(true);
163. }
164.
165. private File file;
166. private StringBuilder text = new StringBuilder();
167. };
168.
169. private JFileChooser chooser;
170. private JTextArea textArea;
171. private JLabel statusLine;
172. private JMenuItem openItem;
173. private JMenuItem cancelItem;
174. private SwingWorker<StringBuilder, ProgressData> textReader;
175.
176. public static final int DEFAULT_WIDTH = 450;
177. public static final int DEFAULT_HEIGHT = 350;
178. }

```

#### **API** javax.swing.SwingWorker<T, V> 6

- `abstract T doInBackground()`  
覆盖这一方法来执行后台的任务并返回这一工作的结果。
- `void process(List<V> data)`  
覆盖这一方法来处理事件分配线程中的中间进度数据。
- `void publish(V... data)`  
传递中间进度数据到事件分配线程。从`doInBackground`调用这一方法。
- `void execute()`  
为工作器线程的执行预定这个工作器。
- `SwingWorker.StateValue getState()`  
得到这个工作器线程的状态，值为PENDING、STARTED或DONE之一。



### 14.11.3 单一线程规则

每一个Java应用程序都开始于主线程中的main方法。在Swing程序中，main方法的生命期是很短的。它在事件分配线程中规划用户界面的构造然后退出。在用户界面构造之后，事件分配线程会处理事件通知，例如调用actionPerformed或paintComponent。其他线程在后台运行，例如将事件放入事件队列的进程，但是那些线程对应用程序员是不可见的。

本章前面介绍了单一线程规则：“除了事件分配线程，不要在任何线程中接触Swing组件。”本节进一步研究此规则。

对于单一线程规则存在一些例外情况。

- 可在任一个线程里添加或移除事件监听器。当然该监听器的方法会在事件分配线程中被触发。
- 只有很少的Swing方法是线程安全的。在API文档中用这样的句子特别标明：“尽管大多数Swing方法不是线程安全的，但这一方法是。”在这些线程安全的方法中最有用的是：

```
JTextComponent.setText  
JTextArea.insert  
JTextArea.append  
JTextArea.replaceRange  
JComponent.repaint  
JComponent.revalidate
```



注释：在本书中多次使用repaint方法，但是，revalidate方法不怎么常见。这样做的目的是在内容改变之后强制执行组件布局。传统的AWT有一个validate方法强制执行组件布局。对于Swing组件，应该调用revalidate方法。（但是，要强制执行JFrame的布局，仍然要调用validate方法，因为JFrame是一个Component不是一个JComponent。）

历史上，单一线程规则是更加随意的。任何线程都可以构建组件，设置优先级，将它们添加到容器中，只要这些组件没有一个是已经被实现的（realized）。如果组件可以接收paint事件或validation事件，组件被实现。一旦调用组件的setVisible(true)或pack(!)方法或者组件已经被添加到已经被实现的容器中，就出现这样的情况。

单一线程规则的这一版本是便利的，它允许在main方法中创建GUI，然后，在应用程序的顶层框架调用setVisible(true)。在事件分配线程上没有令人讨厌的Runnable的安排。

遗憾的是，一些组件的实现者没有注意原来的单一线程规则的微妙之处。他们在事件分配线程启动活动，而没有检查组件是否是被实现的。例如，如果在JTextComponent上调用setSelectionStart或setSelectionEnd，在事件分配线程中安排了一个插入符号的移动，即使该组件不是可见的。

检测并定位这些问题可能会好些，但是Swing的设计者没有走这条轻松的路。他们认定除了使用事件分配线程之外，从任何其他线程访问组件永远都是不安全的。因此，你需要在事件分配线程构建用户界面，像程序示例中那样调用EventQueue.invokeLater。

当然，有不少程序使用旧版的单一线程规则，在主线程初始化用户界面。那些程序有一定

的风险，某些用户界面的初始化会引起事件分配线程的动作与主线程的动作发生冲突。如同我们在第7章讲到的，不要让自己成为少数不幸的人之一，为时有时无的线程bug烦恼并花费时间。因此，一定要遵循严谨的单一线程规则。

现在读者已经读到《Java核心技术 卷I》的末尾。这一卷涵盖了Java程序设计语言的基础知识以及大多数编程项目所需要的标准库中的部分内容。希望读者在学习Java基础知识的过程中感到愉快并得到了有用的信息。有关高级知识内容，如网络、高级的AWT/Swing、安全性以及国际化，请阅读卷II。