第 10 章　线　　程

线程是一个单独程序流程。多线程是指一个程序可以同时运行多个任务，每个任务由一个单独的线程来完成。也就是说，多个线程可以同时在一个程序中运行，并且每一个线程完成不同的任务。程序可以通过控制线程来控制程序的运行，例如线程的等待、休眠、唤起线程等。本章将向读者介绍线程的机制、如何操作和使用线程以及多线程编程。

10.1　线程基本知识

线程是程序运行的基本单位，一个程序中可以同时运行多个线程。如果程序被设置为多线程，可以提高程序运行的效率和处理速度。Java 中线程的实现通常有两种方法：派生 Thread 类和实现 Runnable 接口。本节主要讲述线程的概念和创建线程的方法。

10.1.1　什么是线程

传统的程序设计语言同一时刻只能执行单任务操作，效率非常低，如果网络程序在接收数据时发生阻塞，只能等到程序接收数据之后才能继续运行。随着 Internet 的飞速发展，这种单任务运行的状况越来越不被接受。如果网络接收数据阻塞，后台服务程序就会一直处于等待状态而不能继续任何操作。这种阻塞情况经常发生，这时的 CPU 资源完全处于闲置状态。

多线程实现后台服务程序可以同时处理多个任务，并不发生阻塞现象。多线程是 Java 语言的一个很重要的特征。多线程程序设计最大的特点就是能够提高程序执行效率和处理速度。Java 程序可同时并行运行多个相对独立的线程。例如创建一个线程来接收数据，另一个线程发送数据，既使发送线程在接收数据时被阻塞，接受数据线程仍然可以运行。

线程（Thread）是控制线程（Thread of Control）的缩写，它是具有一定顺序的指令序列（即所编写的程序代码）、存放方法中定义局部变量的栈和一些共享数据。线程是相互独立的，每个方法的局部变量和其他线程的局部变量是分开的，因此，任何线程都不能访问除自身之外的其他线程的局部变量。如果两个线程同时访问同一个方法，那每个线程将各自得到此方法的一个拷贝。

Java 提供的多线程机制使一个程序可同时执行多个任务。线程有时也被称为小进程，它是从一个大进程里分离出来的小的独立的线程。由于实现了多线程技术，Java 显得更健壮。多线程带来的好处是更好的交互性能和实时控制性能。多线程是强大而灵巧的编程工具，但要用好它却不是件容易的事。在多线程编程中，每个线程都通过代码实现线程的行为，并将数据供给代码操作。编码和数据有时是相当独立的，可分别向线程提供。多个线程可以同时处理同一代码和同一数据，不同的线程也可以处理各自不同的编码和数据。

10.1.2 Thread 创建线程

了解了线程的概念后，现在向读者介绍创建线程的方法。Java 中有两种方法创建线程：一种是对 Thread 类进行派生并覆盖 run 方法；另一种是通过实现 runnable 接口创建。

本节先介绍如何通过 Thread 类派生线程的方法，下一节将向读者介绍另一种方法。继承 Thread 类并覆盖 Thread 类的 run 方法完成线程类的声明，通过 new 创建派生线程类的线程对象。run 中的代码实现了线程的行为。

前面的程序都是声明一个公共类，并在类内实现一个 main 方法。事实上，前面这些程序就是一个单线程程序。当它执行完 main 方法的程序后，线程正好退出，程序同时结束运行。下面演示一个系统创建单线程程序的例子。

```
//     程序：10.1   OnlyThread.java          描述：只有一个线程
public class OnlyThread{
      public static void main(String args[ ]){
            run();                                        //调用静态 run()方法
      }
      //实现 run()方法
      public static void run()
       {
            for (int count = 1,row = 1; row < 10; row++,count++)  //循环计算输出的*数目
            {
                  for (int i = 0; i < count; i++)                 //循环输出指定的 count 数目的*
                  {
                        System.out.print('*');          //输出*号
                  }
                  System.out.println();                 //输出换行符
            }
       }
}
```

编写完程序后，使用 javac 命令编译该文件产生 class 文件，然后使用 java 命令运行该 class 文件，运行结果如图 10-1 所示。

程序 10.1 只是建立了一个单一线程并执行的普通小程序，并没有涉及到多线程的概念。java.lang.Thread 类是一个通用的线程类，由于默认情况下 run 方法是空的，直接通过 Thread 类实例化的线程对象不能完成任何事，所以可以通过派生 Thread 类，并用具体程序代码覆盖 Thread 类中的 run 方法，实现具有各种不同功能的线程类。在程序中创建新的线程的方法之一是继承 Thread 类，并通过 Thread 子类声明线程对象。下面是通过 Thread 创建线程的例子。

```
//     文件：程序 10.2       ThreadDemo1.java          描述：产生一个新的线程
class ThreadDemo1 extends Thread{
      //声明 ThreadDemo1 构造方法
      ThreadDemo1(){}
      //声明 ThreadDemo1 带参数的构造方法
      ThreadDemo1(String szName)
      {
            super(szName);                                  //调用父类的构造方法
      }
      //重载 run 函数
      public void run()
```

```
    {
        for (int count = 1,row = 1; row < 10; row++,count++)  //循环计算输出的*数目
        {
            for (int i = 0; i < count; i++)                   //循环输出指定的 count 数目的*
            {
                System.out.print('*');                        //输出*
            }
            System.out.println();                             //输出换行符
        }
    }
    public static void main(String argv[ ]){
        ThreadDemo1 td = new ThreadDemo1();  //创建，并初始化 ThreadDemo1 类型对象 td
        td.start();                                           //调用 start()方法执行一个新的线程
        }
}
```

编写完程序后，使用 javac 命令编译该文件产生 class 文件，然后使用 java 命令运行该 class 文件，运行结果如图 10-2 所示。

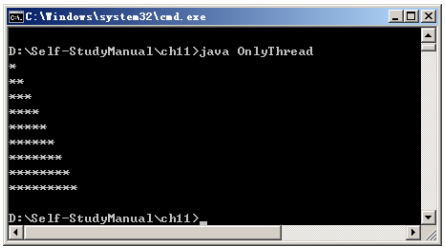

图 10-1　OnlyThread.java 运行结果

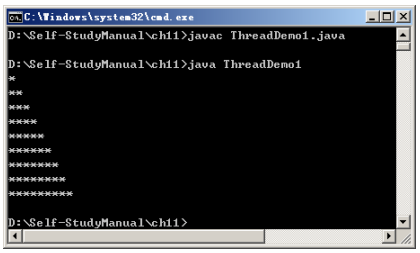

图 10-2　ThreadDemo1.java 运行结果

程序 10.2 与程序 10.1 表面上看运行结果相同，但是仔细对照会发现，程序 10.1 中对 run 方法的调用在程序 10.2 中变成了对 start 方法的调用，并且程序 10.2 明确派生 Thread 类，创建新的线程类。

10.1.3　Thread 创建线程步骤

通常创建一个线程的步骤如下。

（1）创建一个新的线程类，继承 Thread 类并覆盖 Thread 类的 run()方法。

```
class ThreadType extends Thread{
    public void run(){
        ……
    }
}
```

（2）创建一个线程类的对象，创建方法与一般对象的创建相同，使用关键字 new 完成。

```
ThreadType    tt = new ThreadType();
```

（3）启动新线程对象，调用 start()方法。

```
tt.start();
```

（4）线程自己调用 run()方法。

```
void run();
```

下面演示一个创建多个线程的例子。

```
//文件：程序 10.3     ThreadDemo2.java          描述：产生三个新的线程
class ThreadDemo2 extends Thread{
      //声明无参数，空构造方法
      ThreadDemo2(){}
      //声明带有字符串参数的构造方法
      ThreadDemo2(String szName)
      {
            super(szName);                                    //调用父类的构造方法
      }
      //重载 run 函数
      public void run()
      {
            for (int count = 1,row = 1; row < 10; row++,count++)  //循环计算输出的*数目
            {
                  for (int i = 0; i < count; i++)             //循环输出指定的 count 数目的*
                  {
                        System.out.print('*');                //输出*
                  }
                  System.out.println();                       //输出*
            }
      }
      public static void main(String argv[ ]){
            ThreadDemo2 td1 = new ThreadDemo2();  //创建，并初始化 ThreadDemo2 类型对象 td1
            ThreadDemo2 td2 = new ThreadDemo2();  //创建，并初始化 ThreadDemo2 类型对象 td2
            ThreadDemo2 td3 = new ThreadDemo2();  //创建，并初始化 ThreadDemo2 类型对象 td3
            td1.start();                                      //启动线程 td1
            td2.start();                                      //启动线程 td2
            td3.start();                                      //启动线程 td3
            }
}
```

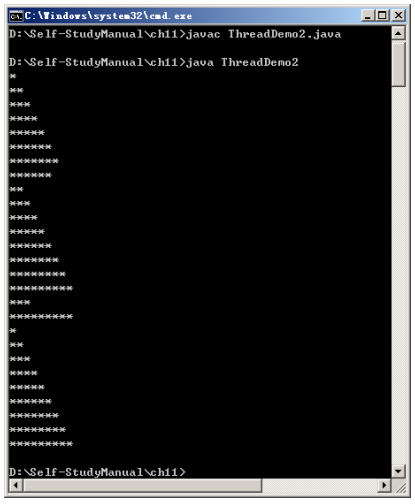

图 10-3　ThreadDemo2.java 运行结果

编写完程序后，使用 javac 命令编译该文件产生 class 文件，然后使用 java 命令运行该 class 文件，运行结果如图 10-3 所示。

程序 10.3 中创建了 3 个线程 td1、td2、td3，它们分别执行自己的 run 方法。在实际中运行的结果并不是想要的直角三角形，而是一些乱七八糟的*行，长短并没有一定的规律，这是因为线程并没有按照程序中调用的顺序来执行，而是产生了多个线程赛跑现象。

注意：Java 线程并不能按调用顺序执行，而是并行执行的单独代码。如果要想得到完整的直角三角形，需要在执行一个线程之前，判断程序前面的线程是否终止，如果已经终止，再来调用该线程。

10.1.4　Runnable 接口创建线程

通过实现 Runnable 接口的方法是创建线程类的第二种方法。利用实现 Runnable 接口来创建线程的方法可以解决 Java 语言不支持的多重继承问题。Runnable 接口提供了 run()方法的原型，因此创建新的线程类时，只要实现此接口，即只要特定的程序代码实现 Runnable 接口中的 run()方法，就可完成新线程类的运行。下面是一个使用 Runnable 接口并实现 run 方法创建线程的例子。

```
//     文件：程序 10.4      ThreadDemo3.java          描述：产生一个新的线程
class ThreadDemo3 implements Runnable{
      //重载 run 函数
      public void run()
      {
            for (int count = 1,row = 1; row < 10; row++,count++)  //循环计算输出的*数目
            {
                  for (int i = 0; i < count; i++)                 //循环输出指定的 count 数目的*
                  {
                        System.out.print('*');                    //输出*
                  }
                  System.out.println();                           //输出换行符
            }
      }
      public static void main(String argv[ ]){
            Runnable rb = new ThreadDemo3();        //创建，并初始化 ThreadDemo3 对象 rb
            Thread td = new Thread(rb);             //通过 Thread 创建线程
            td.start();                             //启动线程 td
            }
}
```

编写完程序后，使用 javac 命令编译该文件产生 class 文件，然后使用 java 命令运行该 class 文件，运行结果如图 10-4 所示。

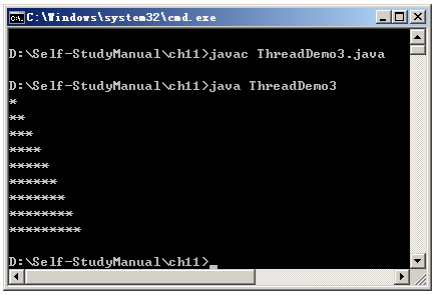

图 10-4　ThreadDemo3.java 运行结果

程序 10.4 的运行结果与程序 10.2 是相同的，但这里的线程是通过实现接口 Runnable 完成的。

10.1.5　Runnable 创建线程步骤

通常实现 Runnable 线程的步骤如下。

（1）创建一个实现 Runnable 接口的类，并且在这个类中重写 run 方法。

```
class ThreadType implements Runnable{
      public void run(){
            ……
      }
}
```

（2）使用关键字 new 新建一个 ThreadType 的实例。

```
Runnable rb = new ThreadType ();
```

（3）通过 Runnable 的实例创建一个线程对象，在创建线程对象时，调用的构造函数是 new Thread(ThreadType)，它用 ThreadType 中实现的 run()方法作为新线程对象的 run()方法。

```
Thread td = new Thread(rb);
```

（4）通过调用 ThreadType 对象的 start()方法启动线程运行。

```
td.start();
```

下面是一个通过 Runnable 创建多线程的例子。

```
//     文件：程序 10.5      ThreadDemo4.java          描述：产生三个新的线程
class ThreadDemo4 implements Runnable{
      //重载 run 函数
      public void run()
      {
            for (int count = 1,row = 1; row < 10; row++,count++)  //循环计算输出的*数目
            {
                  for (int i = 0; i < count; i++)                 //循环输出指定的 count 数目的*
                  {
                        System.out.print('*');                    //输出*
                  }
                  System.out.println();                           //输出换行符
            }
      }
      public static void main(String argv[ ]){
            Runnable rb1 = new ThreadDemo4();                     //创建，并初始化 ThreadDemo4 对象 rb1
            Runnable rb2 = new ThreadDemo4();                     //创建，并初始化 ThreadDemo4 对象 rb2
            Runnable rb3 = new ThreadDemo4();                     //创建，并初始化 ThreadDemo4 对象 rb3
            Thread td1 = new Thread(rb1);                         //创建线程对象 td1
            Thread td2 = new Thread(rb2);                         //创建线程对象 td2
            Thread td3 = new Thread(rb3);                         //创建线程对象 td3
            td1.start();                                          //启动线程 td1
            td2.start();                                          //启动线程 td2
            td3.start();                                          //启动线程 td3
            }
}
```

编写完程序后，使用 javac 命令编译该文件产生 class 文件，然后使用 java 命令运行该 class 文件，运行结果如图 10-5 所示。

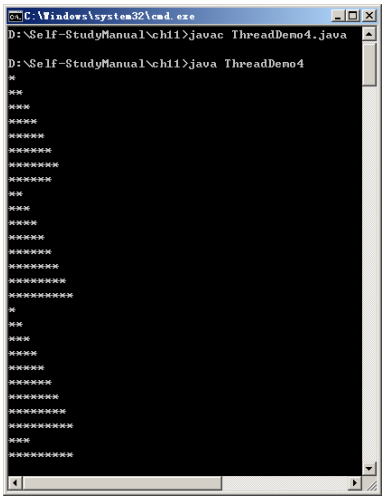

图 10-5　ThreadDemo4.java 运行结果

程序 10.5 与程序 10.3 相同，创建了 3 个线程 td1、td2、td3，且运行结果与 10.3 有些类似。两个程序都不是一个线程结束后再执行另外一个线程，而是线程之间并行运行。由于线程抢占资源，程序发生“线程赛跑”的现象。本书将在后续章节解决“线程赛跑”问题。

10.2　线程周期

前面一节讨论了创建线程的两种实现方式，线程的

创建仅仅是线程生命周期中的一个内容。线程的整个周期由线程创建、可运行状态、不可运行状态和退出等部分组成，这些状态之间的转化是通过线程提供的一些方法完成的。本节将全面讨论线程周期之间的转化过程。

10.2.1　线程周期概念

一个线程有 4 种状态，任何一个线程都处于这 4 种状态中的一种状态。

- 创建（new）状态：调用 new 方法产生一个线程对象后、调用 start 方法前所处的状态。线程对象虽然已经创建，但还没有调用 start 方法启动，因此无法执行。当线程处于创建状态时，线程对象可以调用 start 方法进入启动状态，也可以调用 stop 方法进入停止状态。
- 可运行（runnable）状态：当线程对象执行 start()方法后，线程就转到可运行状态。进入此状态只是说明线程对象具有了可以运行的条件，但线程并不一定处于运行状态。因为在单处理器系统中运行多线程程序时，一个时间点只有一个线程运行，系统通过调度机制实现宏观意义上的运行线程共享处理器。因此一个线程是否在运行，除了线程必须处于 Runnable 状态之外，还取决于优先级和调度。
- 不可运行（non Runnable）状态：线程处于不可运行状态是由于线程被挂起或者发生阻塞，例如对一个线程调用 wait()函数后，它就可能进入阻塞状态；调用线程的 notify 或 notifyAll 方法后它才能再次回到可执行状态。
- 退出（done）状态：一个线程可以从任何一个状态中调用 stop 方法进入退出状态。线程一旦进入退出状态就不存在了，不能再返回到其他的状态。除此之外，如果线程执行完 run 方法，也会自动进入退出状态。

创建状态、可运行状态、不可运行状态、退出状态之间的转换关系如图 10-6 所示。

在图 10-6 中，通过 new 第一次创建线程时，线程位于创建状态，这时不能运行线程，只能等待进一步的方法调用改变其状态。然后，线程通过调用 start 方法启动线程，并进入可执行状态，或者调用方法 stop 进入退出状态。当程序位于退出状态时，线程已经结束执行，这是线程的最后一个状态，并且不能转化到其他状态。当程序的所有线程位于退出状态时，程序会强行终止。当线程位于可执行状态时，在一个特定的时间点上，每一个系统处理器只能运行一个线程。此时如果线程被挂起，执行就会被中断或者进入休眠状态，那么线程将进入不可执行状态，并且不可执行状态可以通过 resume、notify 等方法返回到可执行状态。表 10-1 列举了线程状态转换的函数。

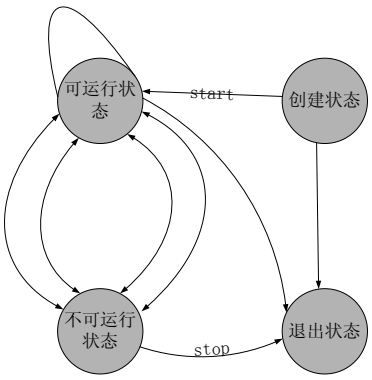

图 10-6　线程状态转换

表 10-1　线程状态转换函数

方法	描述	有效状态	目的状态
start()	开始执行一个线程	New	Runnable
stop()	结束执行一个线程	New 或 Runnable	Done
sleep(long)	暂停一段时间，这个时间为给定的毫秒	Runnable	NonRunnable
sleep(long,int)	暂停片刻，可以精确到纳秒	Runnable	NonRunnable
suspend()	挂起执行	Runnable	NonRunnable
resume()	恢复执行	NonRunnable	Runnable
yield()	明确放弃执行	Runnable	Runnable
wait()	进入阻塞状态	Runnable	NonRunnable
notify()	阻塞状态解除	NonRunnable	Runnable

注意：stop()、suspend()和 resume()方法现在已经不提倡使用，这些方法在虚拟机中可能引起“死锁”现象。suspend()和 resume()方法的替代方法是 wait()和 sleep()。线程的退出通常采用自然终止的方法，建议不要人工调用 stop()方法。

10.2.2　线程的创建和启动

Java 是面向对象的程序设计语言，设计的重点就是类的设计与实现。Java 利用线程类 Thread 来创建线程，线程的创建与普通类对象的创建操作相同。Java 通过线程类的构造方法创建一个线程，并通过调用 start 方法启动该线程。

实际上，启动线程的目的就是为了执行它的 run()方法，而 Thread 类中默认的 run()方法没有任何可操作代码，所以用 Thread 类创建的线程不能完成任何任务。为了让创建的线程完成特定的任务，必须重新定义 run()方法。在第一节中已经讲述过，Java 通常有两种重新定义 run()方法的方式。

- 派生线程类 Thread 的子类，并在子类中重写 run()方法。Thread 子类的实例对象是一个线程对象，并且该线程有专门定制的线程 run()方法，启动线程后就执行子类中重写的 run()方法。
- 实现 Runnable 接口并重新定义 run()方法。先定义一个实现 Runnable()接口的类，在该类中定义 run()方法，然后创建新的线程类对象，并以该对象作为 Thread 类构造方法的参数创建一个线程。

下面举一个通过上述两种方法创建并启动线程的例子。

```
//    文件：程序 10.6      ThreadStart.java          描述：线程启动例子
public class   ThreadStart
{
      public static void main(String[ ] args)
      {
            Runnable r = new RunnableThread();          //创建，并初始化 RunnableThread 对象
            Thread rt = new Thread(r);                  //创建，并初始化线程对象 rt
            rt.start();                                 //启动线程
            SubThread st = new SubThread("SubThread");//创建，并初始化 SubThread 对象
```

```
            st.start();                                    //启动线程
        }
}
class RunnableThread implements Runnable{
        //重载 run 函数
        public void run()
        {
            System.out.println("RunnableThread 启动");     //输出字符串信息
        }
}
class SubThread extends Thread{
        SubThread(){}                                      //声明，并实现 SubThread 构造方法
        //声明，并实现 SubThread 带参数的构造方法
        SubThread(String Name)
        {
            super(Name);                                   //调用父类的构造方法
        }
        //重载 run 函数
        public void run()
        {
            System.out.println("SubThread 启动");          //输出字符串信息
        }
}
```

编写完程序后，使用 javac 命令编译该文件产生 class 文件，然后使用 java 命令运行该 class 文件，运行结果如图 10-7 所示。

程序 10.6 中，类 RunnableThread 实现 Runnable 接口，并重写 run 方法；SubThread 继承 Thread，同样也实现了 run 方法。在 main 方法中，先通过 RunnableThread 类创建，并启动线程 rt；然后通过 SubThread 类创建，并启动另外一个线程 st。

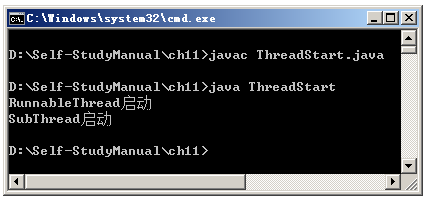

图 10-7　ThreadStart.java 运行结果

注意：调用线程的 run()方法是通过启动线程的 start()方法来实现的。因为线程在调用 start()方法之后，系统会自动调用 run()方法。与一般方法调用不同的地方在于一般方法调用另外一个方法后，必须等被调用的方法执行完毕才能返回，而线程的 start()方法被调用之后，系统会得知线程准备完毕并且可以执行 run()方法，start()方法就返回了，start()方法不会等待 run()方法执行完毕。

10.2.3　线程状态转换

1．线程进入可执行状态

当以下几种情况发生时，线程进入可执行状态。

（1）其他线程调用 notify()或者 notifyAll()方法，唤起处于不可执行状态的线程。

```
public final void notify()
public final void notifyAll()
```

notify 仅仅唤醒一个线程并允许它获得锁，notifyAll 唤醒所有等待这个对象的线程，并允许它们获得锁。wait 和 notify 是 Java 同步机制的重要内容，所以留在后面的线程同步部分

做进一步讲解。

（2）线程调用 sleep(millis)方法，millis 毫秒之后线程会进入可执行状态。

```
static void sleep(long millis) throws InterruptedException
```

在 millis 毫秒数内让当前正在执行的线程进入休眠状态，等到时间过后，该线程会自动苏醒并继续执行。sleep 方法的精确度受到系统计数器的影响。

```
static void sleep(long millis, int nanos) throws InterruptedException
```

在毫秒数（millis）加纳秒数（nanos）内让当前正在执行的线程进入休眠状态，此操作的精确度也受到系统计数器的影响。

（3）线程对 I/O 操作的完成。

2．线程进入不可执行状态

当以下几种情况发生时，线程进入不可执行状态。

（1）线程自动调用 wait()方法，等待某种条件的发生。

```
public final void wait() throws InterruptedException
```

当其他线程调用 notify()方法或 notifyAl()方法后，处于等待状态的线程获得锁之后才会被唤醒，然后该线程一直等待重新获得对象锁才继续运行。

（2）线程调用 sleep()方法进入不可执行状态，在一定时间后会进入可执行状态。

（3）线程等待 I/O 操作的完成。

下面举一个线程阻塞的例子。

```
//     文件：程序 10.7       ThreadSleep.java          描述：线程阻塞例子
public class   ThreadSleep
{
      public static void main(String[ ] args)
      {
            SubThread st = new SubThread("SubThread");       //创建，并初始化 SubThread 对象 st
            st.start();                                      //启动线程 st
      }
}

class SubThread extends Thread{
      SubThread(){}                                          //声明，实现 SubThread 无参数构造方法
      //声明，实现 SubThread 带字符串参数构造方法
      SubThread(String Name)
      {
            super(Name);                                     //调用父类的构造方法
      }
      //重载 run 函数
      public void run()
      {
            for (int count = 1,row = 1; row < 10; row++,count++) //循环计算输出的*数目
            {
                  for (int i = 0; i < count; i++)            //循环输出指定的 count 数目的*
                  {
                        System.out.print('*');               //输出*
                  }
                  try                                        //try-catch 块，用于捕获异常
                  {
```

```
                Thread.sleep(1000);                    //线程休眠 1 秒钟
                System.out.print("\t wait.....");
            }
            catch (InterruptedException e)             //捕获异常 InterruptedException
            {
                e.printStackTrace();                   //异常抛出信息
            }
            System.out.println();                      //输出换行符
        }
    }
}
```

编写完程序后，使用 javac 命令编译该文件产生 class 文件，然后使用 java 命令运行该 class 文件，运行结果如图 10-8 所示。

程序 10.7 中，每输出一行*就要休息 1 秒钟。当执行 sleep()语句后，线程进入不可执行状态等待 1 秒钟之后，线程 st 会自动苏醒并继续执行。由于 sleep 方法抛出 InterruptedException 异常，所以在调用时必须捕获异常。

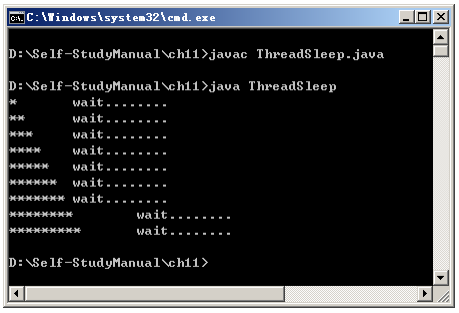

图 10-8　ThreadSleep.java 运行结果

10.2.4　等待线程结束

isAlive()方法用来判断一个线程是否存活。当线程处于可执行状态或不可执行状态时，isAlive()方法返回 true；当线程处于创建状态或退出状态时，则返回 false。也就是说，isAlive()方法如果返回 true，并不能判断线程是处于可运行状态还是不可运行状态。isAlive()方法的原型如下所示。

```
public final boolean isAlive()
```

该方法用于测试线程是否处于活动状态。活动状态是指线程已经启动（调用 start 方法）且尚未退出所处的状态，包括可运行状态和不可运行状态。可以通过该方法解决程序 10.3 中的问题，先判断第一个线程是否已经终止，如果终止再来调用第二个线程。这里提供两种方法。

第一种是不断查询第一个线程是否已经终止，如果没有，则让主线程睡眠一直到它终止即“while/isAlive/sleep”，格式如下。

```
线程 1.start();
while(线程 1.isAlive())
{
    Thread.sleep(休眠时间);
}
线程 2.start();
```

第二种是利用 join()方法。

```
public final void join(long millis) throws InterruptedException
```

等待该线程终止的时间最长为毫秒（millis），超时为 0 意味着要一直等下去。

public final void join(long millis,int nanos) throws InterruptedException

等待该线程终止的时间最长为毫秒（millis）加纳秒（nanos）。

public final void join() throws InterruptedException

等待该线程终止。

下面举一个等待线程结束并执行另外一个线程的例子。

```
//     文件：程序 10.8 WaitThreadStop.java            描述：等待一个线程的结束的两种方法
class WaitThreadStop extends Thread{
      //声明，并实现 WaitThreadStop 无参数构造方法
      WaitThreadStop(){}
      //声明，并实现带有一个字符串参数的构造方法
      WaitThreadStop(String szName)
      {
            super(szName);                        //调用父类的构造方法
      }
      //重载 run 函数
      public void run()
      {
            for (int count = 1,row = 1; row < 10; row++,count++)
            {
                  for (int i = 0; i < count; i++)
                  {
                        System.out.print('*');        //输出*
                  }
                  System.out.println();               //输出换行符
            }
      }
}
public class WaitThreadStopMain{
      public static void main(String argv[ ]){
            WaitThreadStopMain test = new WaitThreadStopMain();                    //创建，初始化
WaitThreadStopMain 对象 test
            test.Method1();        //调用 Method1 方法
            //test.Method2();
      }
            //第一种方法：while/isAlive/sleep
      public void Method1(){
            WaitThreadStop th1 = new WaitThreadStop();  //创建，并初始化 WaitThreadStop 对象 th1
            WaitThreadStop th2 = new WaitThreadStop();  //创建，并初始化 WaitThreadStop 对象 th2
            //执行第一个线程
            th1.start();
            //查询第一个线程的状态
            while(th1.isAlive()){
                  try{
                        Thread.sleep(100);          //休眠 100 毫秒
                  }catch(InterruptedException e)
                  {
                        e.printStackTrace();        //异常信息输出
                  }
                  }
            //     当第一个线程终止后，运行第二个线程
            th2.start();                              //启动线程 th2
            }
      //第二种方法，使用 join 方法实现等待其他线程结束
```

```
    public void Method2(){
        WaitThreadStop th1 = new WaitThreadStop();  //创建，并初始化 WaitThreadStop 对象 th1
        WaitThreadStop th2 = new WaitThreadStop();  //创建，并初始化 WaitThreadStop 对象 th2
        //执行第一个线程
        th1.start();
        try{
            th1.join();                             //th1 调用 join 方法
        }catch(InterruptedException e)
        {
            e.printStackTrace();                    //异常信息输出
        }
        //      执行第二个线程
        th2.start();
    }
}
```

编写完程序后，使用 javac 命令编译该文件产生 class 文件，然后使用 java 命令运行该 class 文件，运行结果如图 10-9 所示。

程序 10.8 先等待线程 th1 执行结束之后，再执行 th2。上面程序提供了两种实现等待 th1 线程终止的方法。第一种方法是通过 isAlive 不断查询第一线程的状态，等待第一个线程的终止，然后执行第二个线程；第二种方法是通过 join()方法等待线程终止。

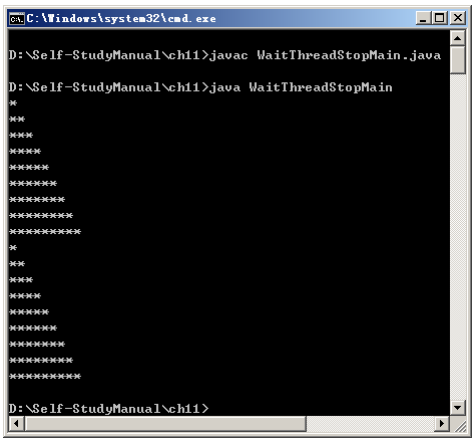

图 10-9　WaitThreadStop.java 运行结果

10.3　线程调度

多线程应用程序的每一个线程的重要性和优先级可能不同，例如有多个线程都在等待获得 CPU 的时间片，那么优先级高的线程就能抢占 CPU 并得以执行；当多个线程交替抢占 CPU 时，优先级高的线程占用的时间应该多。因此，高优先级的线程执行的效率会高些，执行速度也会快些。

在 Java 中，CPU 的使用通常是抢占式调度模式不需要时间片分配进程。抢占式调度模式是指许多线程同时处于可运行状态，但只有一个线程正在运行。当线程一直运行直到结束，或者进入不可运行状态，或者具有更高优先级的线程变为可运行状态，它将会让出 CPU。线程与优先级相关的方法如下。

```
public final void setPriority(int newPriority)
```

设置线程的优先级为 newPriority。newPriority 的值必须在 MIN_PRIORITY 到 MAX_PRIORITY 范围内，通常它们的值分别是 1 和 10。目前 Windows 系统只支持 3 个级别的优先级，它们分别是 Thread.MAX_PRIORITY、Thread.MIN_PRIORITY 和 Thread.NORM_PRIORITY。

```
public final int getPriority()
```

获得当前线程的优先级。

下面举一个线程优先级的例子。

```
//     文件：程序 10.9       ThreadPriority.java          描述：设置线程优先级
class   InheritThread extends Thread {
            //自定义线程的 run()方法
            public void run(){
                  System.out.println("InheritThread is running…");       //输出字符串信息
                  for(int i=0;i<10;i++){
                        System.out.println(" InheritThread: i="+i);       //输出信息
                        try{
                              Thread.sleep((int)Math.random()*1000);    //线程休眠
                        }
                        catch(InterruptedException e)                     //捕获异常
                        {}
                  }
            }
}
//通过 Runnable 接口创建的另外一个线程
class RunnableThread implements Runnable{
      //自定义线程的 run()方法
      public void run(){
            System.out.println("RunnableThread is running…");       //输出字符串信息
            for(int i=0;i<10;i++){
                  System.out.println("RunnableThread : i="+i);          //输出 i
                  try{
                        Thread.sleep((int)Math.random()*1000);          //线程休眠
                  }
                  catch(InterruptedException e){                        //捕获异常
                  }
          }
    }
}
public class ThreadPriority{
      public static void main(String args[ ]){
            //用 Thread 类的子类创建线程
            InheritThread itd=new InheritThread();
            //用 Runnable 接口类的对象创建线程
            Thread rtd=new Thread(new RunnableThread());
            itd.setPriority(5);                                           //设置 myThread1 的优先级 5
            rtd.setPriority(5);                                           //设置 myThread2 的优先级 5
            itd.start();                                                  //启动线程 itd
            rtd.start();                                                  //启动线程 rtd
    }
}
```

编写完程序后，使用 javac 命令编译该文件产生 class 文件，然后使用 java 命令运行该 class 文件，运行结果如图 10-10 所示。

在程序 10.9 中，线程 rtd 和 itd 具有相同的优先级，所以它们交互占用 CPU，宏观上处于并行运行状态。重新设定优先级，如下所示。

```
itd.setPriority(1);      //设置 myThread1 的优先级 1
rtd.setPriority(10);     //设置 myThread2 的优先级 10
```

运行程序结果如图 10-11 所示。

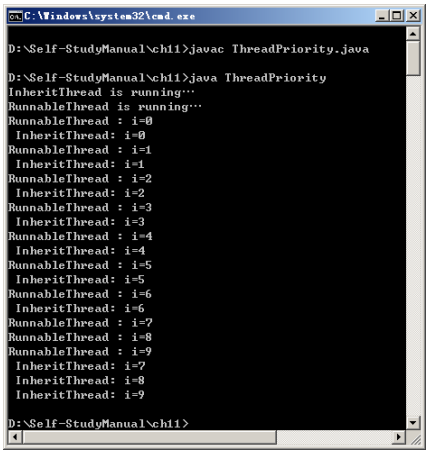

图 10-10　ThreadPriority.java 运行结果

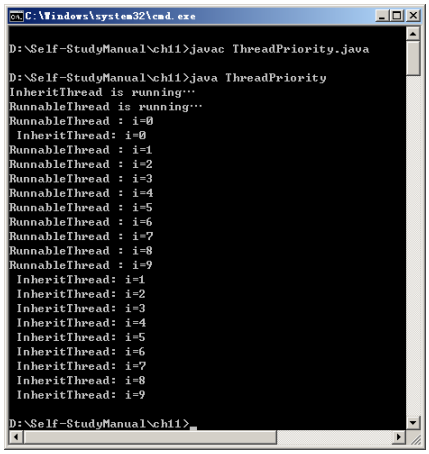

图 10-11　ThreadPriority.java 运行结果

从运行结构可以看出程序 10.9 修改后，由于设置了线程 itd 和 rtd 的优先级，并且 rtd 的优先级较高，基本上是 rtd 都优先抢占 CPU 资源。

10.4　线程同步

Java 应用程序中的多线程可以共享资源，例如文件、数据库、内存等。当线程以并发模式访问共享数据时，共享数据可能会发生冲突。Java 引入线程同步的概念，以实现共享数据的一致性。线程同步机制让多个线程有序的访问共享资源，而不是同时操作共享资源。

10.4.1　同步概念

在线程异步模式的情况下，同一时刻有一个线程在修改共享数据，另一个线程在读取共享数据，当修改共享数据的线程没有处理完毕，读取数据的线程肯定会得到错误的结果。如果采用多线程的同步控制机制，当处理共享数据的线程完成处理数据之后，读取线程读取数据。

通过分析多线程出售火车票的例子，可以更好得理解线程同步的概念。线程 Thread1 和线程 Thread2 都可以出售火车票，但是这个过程中会出现数据与时间信息不一致的情况。线程 Thread1 查询数据库，发现某张火车票 T 可以出售，所以准备出售此票；此时系统切换到线程 Thread2 执行，它在数据库中查询存票，发现上面的火车票 T 可以出售，所以线程 Thread2 将这张火车票 T 售出；当系统再次切换到线程 Thread1 执行时，它又卖出同样的票 T。这是一个典型的由于数据不同步而导致的错误。

下面举一个线程异步模式访问数据的例子。

```
//     文件：程序 10.10     ThreadNoSynchronized.java          描述：多线程不同步的原因
class ShareData{
      public static String szData = "";        //声明，并初始化字符串数据域，作为共享数据
```

```
}
class ThreadDemo extends Thread{
    private ShareData oShare;              //声明，并初始化 ShareData 数据域
    ThreadDemo(){}                         //声明，并实现 ThreadDemo 构造方法
    //声明，并实现 ThreadDemo 带参数的构造方法
    ThreadDemo(String szName,ShareData oShare){
        super(szName);                     //调用父类的构造方法
        this.oShare = oShare;              //初始化 oShare 域
    }
    public void run(){
        for (int i = 0; i < 5; i++){
            if (this.getName().equals("Thread1")){
                oShare.szData = "这是第 1 个线程";
                //为了演示产生的问题，这里设置一次睡眠
                try{
                Thread.sleep((int)Math.random() * 100);          //休眠
                }
                catch(InterruptedException e){                   //捕获异常
                }
                System.out.println(this.getName() + ":" + oShare.szData); //输出字符串信息
                }else if(this.getName().equals("Thread2"))
                {
                oShare.szData = "这是第 2 个线程";
                //为了演示产生的问题，这里设置一次睡眠
                try{
                    Thread.sleep((int)Math.random() * 100);   //线程休眠
                }catch(InterruptedException e)                   //捕获异常
                {}
                System.out.println(this.getName() + ":" + oShare.szData); //输出字符串信息
            }
        }
    }
}
public class ThreadNoSynchronized {
    public static void main(String argv[ ]){
        ShareData oShare = new ShareData();          //创建，初始化 ShareData 对象 oShare
        ThreadDemo th1 = new ThreadDemo("Thread1",oShare);       //创建线程 th1
        ThreadDemo th2 = new ThreadDemo("Thread2",oShare);       //创建线程 th2
        th1.start();       //启动线程 th1
        th2.start();       //启动线程 th2
    }
}
```

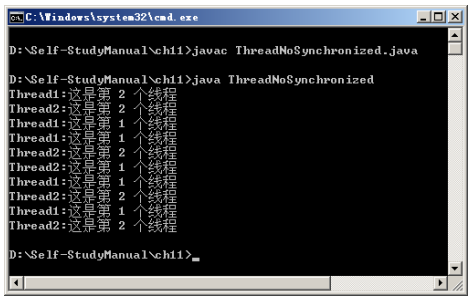

图 10-12　ThreadNoSynchronized.java 运行结果

编写完程序后，使用 javac 命令编译该文件产生 class 文件，然后使用 java 命令运行该 class 文件，运行结果如图 10-12 所示。

程序 10.10 中预想的结果是“Thead1：这是第 1 个线程”或“Thead2：这是第 2 个线程”，但是线程对数据的异步操作导致运行结果出现了差错。

上面程序是由于线程不同步而导致错误。为了解决此类问题，Java 提供了“锁”机制实现线程的同步。锁机制的原理是每个线程进入共享代码之前

获得锁，否则不能进入共享代码区，并且在退出共享代码之前释放该锁，这样就解决了多个线程竞争共享代码的情况，达到线程同步的目的。Java 中锁机制的实现方法是共享代码之前加入 synchronized 关键字。

在一个类中，用关键字 synchonized 声明的方法为同步方法。Java 有一个专门负责管理线程对象中同步方法访问的工具——同步模型监视器，它的原理是为每个具有同步代码的对象准备惟一的一把“锁”。当多个线程访问对象时，只有取得锁的线程才能进入同步方法，其他访问共享对象的线程停留在对象中等待，如果获得锁的线程调用 wait 方法放弃锁，那么其他等待获得锁的线程将有机会获得锁。当某一个等待线程取得锁，它将执行同步方法，而其他没有取得锁的线程仍然继续等待获得锁。

Java 程序中线程之间通过消息实现相互通信，wait()、notify()及 notifyAll()方法可完成线程间的消息传递。例如，一个对象包含一个 synchonized 同步方法，同一时刻只能有一个获得锁的线程访问该对象中的同步方法，其他线程被阻塞在对象中等待获得锁。当线程调用 wait()方法可使该线程进入阻塞状态，其他线程调用 notify()或 notifyAll()方法可以唤醒该线程。

10.4.2　同步格式

当把一语句块声明为 synchornized，在同一时间，它的访问线程之一才能执行该语句块。用关键字 synchonized 可将方法声明为同步，格式如下。

```
class 类名{
        public synchonized 类型名称 方法名称(){
                ......
        }
}
```

对于同步块，synchornized 获取的是参数中的对象锁。

```
synchornized(obj)
{
   //……
}
```

当线程执行到这里的同步块时，它必须获取 obj 这个对象的锁才能执行同步块；否则线程只能等待获得锁。必须注意的是 obj 对象的作用范围不同，控制情况不尽相同。示例如下。

```
public void method()
{
   Object obj= new Object();      //创建局部 Object 类型对象 obj
   synchornized(obj)              //同步块
   {
        //……
   }
}
```

上面的代码创建了一个局部对象 obj。由于每一个线程执行到 Object obj = new Object()时都会产生一个 obj 对象，每一个线程都可以获得创建的新的 obj 对象的锁，不会相互影响，因此这段程序不会起到同步作用。如果同步的是类的属性，情况就不同了。同步类的成员变量的一般格式如下。

```
class method
{
    Object o = new Object();    //创建 Object 类型的成员变量 o
public void test()
{
synchornized(o)              //同步块
{
            //……
        }
    }
}
```

当两个并发线程访问同一个对象的 synchornized(o)同步代码块时，一段时间内只能有一个线程运行。另外的线程必须等到当前线程执行完同步代码块释放锁之后，获得锁的线程将执行同步代码块。有时可以通过下面的格式声明同步块。

```
public void method()
{
synchornized(this)           //同步块
{
    //……
    }
}
```

当有一个线程访问某个对象的 synchornized(this)同步代码块时，另外一个线程必须等待该线程执行完此代码块，其他线程可以访问该对象中的非 synchornized(this)同步代码。如果类中包含多个 synchornized(this)同步代码块，如果同步线程有一个访问其中一个代码块，则其他线程不能访问该对象的所有 synchornized(this)同步代码块。对于下面形式的同步块而言，调用 ClassName 对象实例的并行线程中只有一个线程能够访问该对象。

```
synchornized(ClassName.class)
{
    //……
}
```

10.4.3 同步应用

下面举一个使用 synchornized 解决 10.10 中遇到“线程赛跑”问题的例子。该程序先创建一个共享数据 oShare，然后分别创建两个线程访问共享数据。

```
//    文件：程序 10.11    ThreadSynchronizedMain.java  描述：多线程不同步的解决方法——锁
class ShareData{
    public static String szData = ""; //声明，并初始化字符串数据域 szData
}
class ThreadDemo extends Thread{
    private ShareData oShare;       //声明 ShareData 数据域
    ThreadDemo(){}                  //声明，实现无参数构造方法
    //声明，实现带参数构造方法
    ThreadDemo(String szName,ShareData oShare){
        super(szName);              //调用父类构造方法
        this.oShare = oShare;       //初始化 oShare 域
    }
    public void run(){
```

```
            //同步块，并指出同步数据 oShare
            synchronized (oShare){    //指定同步块，给 oShare 加锁
            for (int i = 0; i < 5; i++){    //循环执行
                  if (this.getName().equals("Thread1"))                    //当前线程是 Thread1
                  {
                        oShare.szData = "这是第 1 个线程";
                        //为了演示产生的问题，这里设置一次睡眠
                        try{
                        Thread.sleep((int)Math.random() * 50);             //线程休眠
                        }
                        catch(InterruptedException e){                     //捕获异常
                        }
                        System.out.println(this.getName() + ":" + oShare.szData); //输出字符串信息
                  }else if(this.getName().equals("Thread2"))               //当前线程为 Thread2
                  {

                        oShare.szData = "这是第 2 个线程";
                        //为了演示产生的问题，这里设置一次睡眠
                        try{
                              Thread.sleep((int)Math.random() * 50);       //线程休眠
                        }catch(InterruptedException e)                     //捕获异常
                        {}
                        System.out.println(this.getName() + ":" + oShare.szData); //输出字符串信息
                        }
                  }
            }
      }
}
public class ThreadSynchronizedMain {
      public static void main(String argv[ ]){
            ShareData oShare = new ShareData();            //创建，并初始化 ShareData 对象 oShare
            ThreadDemo th1 = new ThreadDemo("Thread1",oShare); //创建线程 th1
            ThreadDemo th2 = new ThreadDemo("Thread2",oShare); //创建线程 th2
            th1.start();                                                   //启动线程 th1
            th2.start();                                                   //启动线程 th2
      }
}
```

编写完程序后，使用 javac 命令编译该文件产生 class 文件，然后使用 java 命令运行该 class 文件，运行结果如图 10-13 所示。

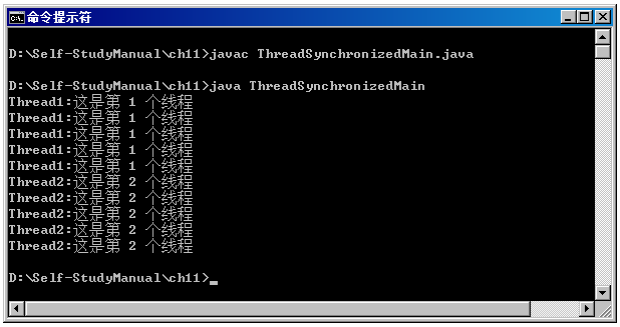

图 10-13　ThreadSynchronizedMain.java 运行结果

在程序 10.11 中，利用同步块实现两个线程的同步问题，声明了两个线程 th1 和 th2，并且在 run 方法中包括同步块，当程序启动线程 th1 之后，th1 获得对象的锁，直到 th1 执行结束之后，th2 才能获得对象的锁，并继续运行。

注意：由于两个线程都在等待对方释放各自拥有的锁的现象称为死锁。这种现象往往是由于相互嵌套的 synchronized 代码段而造成，因此，在程序中尽量少用嵌套的 synchronized 代码块。

10.5 线程通信

多线程之间可以通过消息通信，以达到相互协作的目的。Java 中线程之间的通信是通过 Object 类中的 wait()、notify()、notifyAll()等几种方法实现的。Java 中每个对象内部不仅有一个对象锁之外，还有一个线程等待队列，这个队列用于存放所有等待对象锁的线程。

10.5.1 生产者/消费者

生产者与消费者是一个很好的线程通信的例子。生产者在一个循环中不断生产共享数据，而消费者则不断地消费生产者生产的共享数据。二者之间的关系可以很清楚地说明，必须先有生产者生产共享数据，才能有消费者消费共享数据。因此程序必须保证在消费者消费之前，必须有共享数据，如果没有，消费者必须等待产生新的共享数据。生产者和消费者之间的数据关系如下。

- 生产者生产前，如果共享数据没有被消费，则生产者等待；生产者生产后，通知消费者消费。
- 消费者消费前，如果共享数据已经被消费完，则消费者等待；消费者消费后，通知生产者生产。

为了解决生产者和消费者的矛盾，引入了等待/通知（wait/notify)机制。等待通知使用 wait 方法，通知消费生产使用 notifyAll()或者 notify()方法，程序 10.12 将举一个多线程通信即消费者和生产者的例子。

```
//文件：程序 10.12    Producer.java        描述：    生产者消费者线程
class Producer extends Thread        //实现生产者线程
{
    Queue q;                         //声明队列 q

    Producer(Queue q)                //生产者构造方法
    {
        this.q = q;                  //队列 q 初始化
    }
    public void run()
    {
        for(int i=1;i<5;i++)         //循环添加元素
        {
            q.put(i);                //给队列中添加新的元素
```

```
        }
    }
}
class Consumer extends Thread
{
    Queue q;                        //声明队列 q

    Consumer(Queue q)               //消费者构造方法
    {
        this.q = q;                 //队列 q 初始化
    }
    public void run()
    {
        while(true)                 //循环消费元素
        {
            q.get();                //获取队列中的元素
        }
    }
}
```

Producer 是一个生产者类，该生产者类提供一个以共享队列作为参数的构造方法，它的 run 方法循环产生新的元素，并将元素添加于共享队列；Consumer 是一个消费者类，该消费者类提供一个以共享队列作为参数的构造方法，它的 run 方法循环消费元素，并将元素从共享队列删除。

10.5.2　共享队列

共享队列类是用于保存生产者生产、消费者消费的共享数据。共享队列有两个域：value（元素的数目）、isEmpty（队列的状态）。共享队列提供了 put 和 get 两个方法。

```
class Queue
{
    int value = 0;                  //声明，并初始化整数类型数据域 value
    boolean isEmpty = true;         //声明，并初始化布尔类型数据域 isEmpty，用于判断队列的状态
     //生产者生产方法
    public synchronized void put(int v)
    {
        //如果共享数据没有被消费，则生产者等待
        if (!isEmpty)
        {
            try
            {
                System.out.println("生产者等待");
                wait();                     //进入等待状态
            }
            catch(Exception e)              //捕获异常
            {
                e.printStackTrace();        //异常信息输出
            }
        }
        value += v;      //value 值加 v
        isEmpty = false;                    //isEmpty 赋值为 false
```

```
            System.out.println("生产者共生产数量："+v);      //输出字符串信息
            notify();                                       //生产之后通知消费者消费
}
    //消费者消费的方法
    public synchronized int get()
{
            //消费者消费前，如果共享数据已经被消费完，则消费者等待
            if (isEmpty)
            {
                try
                {
                   System.out.println("消费者等待");      //输出字符串信息
                      wait();                               //进入等待状态
                }
                catch(Exception e)                          //捕获异常
                {
                      e.printStackTrace();                  //异常信息输出
                }
            }
              value--;                                      //value 值-1
              if (value < 1)
              {
                    isEmpty = true;                         //isEmpty 赋值 true
              }
              System.out.println("消费者消费一个,剩余："+value);         //输出信息
            notify();                                       //消费者消费后,通知生产者生产
            return value;                                   //返回 value
    }
}
```

生产者调用 put 方法生产共享数据，如果共享数据不为空，生产者线程进入等待状态；否则将生成新的数据，然后调用 notify()方法唤起消费者线程进行消费；消费者调用 get 方法消费共享数据，如果共享数据为空，消费者线程进入等待状态，否则将消费共享数据，然后调用 notify()方法唤起生产者线程进行生产。

10.5.3 运行生产者/消费者

下面的程序是生产者/消费者程序的主程序，该程序创建了一个共享队列、一个生产者线程、一个消费者线程，分别调用线程的 start 方法启动两个线程。

```
//      文件：      ThreadCommunication.java            描述：多线程之间通信
public class ThreadCommunication
{
    public static void main(String[ ] args)
    {
        Queue q = new Queue();                          //创建，并初始化一个队列
        Producer p = new Producer(q);                   //创建，并初始化一个生产者
        Consumer c = new Consumer(q);                   //创建，并初始化一个消费者

        c.start();                                      //消费者线程启动
        p.start();                                      //生产者线程启动
    }
```

```
}
```

编写完程序后，使用 javac 命令编译该文件产生 class 文件，然后使用 java 命令运行该 class 文件，运行结果如图 10-14 所示。

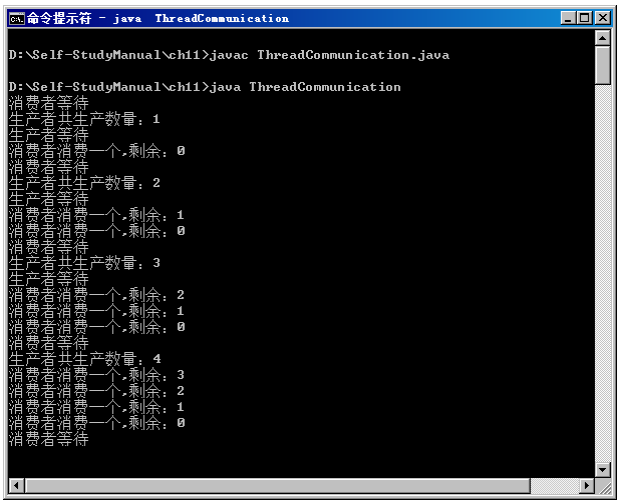

图 10-14　ThreadCommunication.java 运行结果

程序 10.12 中，模拟了生产者和消费者的关系。开始消费者调用消费方法时处于等待状态，此时唤起生产者线程。生产者开始生产共享数据之后，消费者进行消费，但是当共享数据为空，所有消费者必须等待，生产者继续生产，然后消费者再次消费，如此循环直到程序运行最后，可以看到线程一直等待。注意这个线程进入等待后没有其他线程唤醒，除非强行退出 JVM 环境，否则它一直等待。

注意：考虑到程序的安全性，多数情况下使用 notifiAll()，除非明确可以知道唤醒哪一个线程。wait 方法调用的前提条件是当前线程获取了这个对象的锁，也就是说 wait 方法必须放在同步块或同步方法中。

10.6　死锁

前面已经讲过，为了保证数据安全使用 synchronized 同步机制，当线程进入堵塞状态（不可运行状态和等待状态）时，其他线程无法访问那个加锁对象（除非同步锁被解除），所以一个线程会一直处于等待另一个对象的状态，而另一个对象又会处于等待下一个对象的状态，以此类推，这个线程“等待”状态链会发生很糟糕的情形，即封闭环状态（也就是说最后那个对象在等待第一个对象的锁）。此时，所有的线程都陷入毫无止境的等待状态中，无法继续运行，这种情况就称为“死锁”。虽然这种情况发生的概率很小，一旦出现，程序的调试变得困难而且查错也是一件很麻烦的事情。下面举一个死锁的例子。

```
//      文件：程序 10.13      ThreadLocked.java          描述：多线程不同步的原因
public class ThreadLocked implements Runnable {
        public static boolean flag = true;                  //起一个标志作用
        private static Object A = new Object();             //声明，并初始化静态 Object 数据域 A
```

```
private static Object B = new Object();            //声明，并初始化静态 Object 数据域 B
public static void main(String[ ] args) throws InterruptedException {
    Runnable r1 = new ThreadLocked();              //创建，并初始化 ThreadLocked 对象 r1
    Thread t1 = new Thread(r1);                    //创建线程 t1
    Runnable r2 = new ThreadLocked();              //创建，并初始化 ThreadLocked 对象 r2
    Thread t2 = new Thread(r2);                    //创建线程 t2
    t1.start();  //启动线程 t1
    t2.start();  //启动线程 t2
}
public void AccessA()
{
    flag = false;                                  //初始化域 flag
    //同步代码快
    synchronized (A) {                             //声明同步块，给对象 A 加锁
        System.out.println("线程 t1： 我得到了 A 的锁");          //输出字符串信息
        try {
            //让当前线程睡眠,从而让另外一个线程可以先得到对象 B 的锁
            Thread.sleep(1000);                    //休眠
        } catch (InterruptedException e) {         //捕获异常
            e.printStackTrace();                   //异常信息输出
        }
        System.out.println("线程 t1： 我还想要得到 B 的锁");
        //在得到 A 锁之后,又想得到 B 的锁
        //同步块内部嵌套同步块
        synchronized (B) {                         //声明内部嵌套同步块，指定对象 B 的锁
            System.out.println("线程 t1： 我得到了 B 的锁");      //输出字符串信息
        }
    }
}
public void AccessB()
{
    flag = true;                                   //修改 flag 的值
    //同步代码块
    synchronized (B) {                             //指定同步块，给 B 加锁
        System.out.println("线程 t2： 我得到了 B 的锁");      //输出字符串信息
        try {
            //让当前线程睡眠,从而让另外一个线程可以先得到对象 A 的锁
            Thread.sleep(1000);                    //休眠
        } catch (InterruptedException e) {         //捕获异常 InterruptedException
                e.printStackTrace();               //异常信息输出
        }
        System.out.println("线程 t2： 我还想要得到 A 的锁");      //字符串信息输出
        //在得到 B 锁之后,又想得到 A 的锁
        //同步块内部嵌套内部快
        synchronized (A) {                         //指定同步块，给 A 加锁
            System.out.println("线程 t2： 我得到了 A 的锁");      //输出字符串信息
        }
    }
}
public void run() {
    if (flag)                                      //当 flag 为 true，执行下面语句
    {
        AccessA();                                 //调用 AccessA 方法
    }else
```

```
        {
            AccessB();                          //调用 AccessB 方法
        }
    }

}
```

编写完程序后，使用 javac 命令编译该文件产生 class 文件，然后使用 java 命令运行该 class 文件，运行结果如图 10-15 所示。

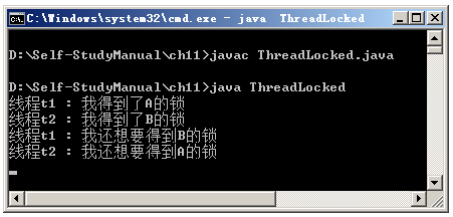

图 10-15　ThreadLocked.java 运行结果

程序 10.13 中创建了两个线程 t1 和 t2，并且声明两个方法：AccessA 和 AccessB。在运行过程中，线程 t1 先获得了 A 的锁，然后又要求获得 B 的锁；而 t2 先获得 B 的锁，然后又要求获得 A 的锁，此时便进入了无休止的相互等待状态，即死锁。

Java 语言本身并没有提供防止死锁的具体方法，但是在具体程序设计时必须要谨慎，以防止出现死锁现象。通常在程序设计中应注意，不要使用 stop()、suspend()、resume()以及 destroy()方法。

stop()方法不安全，它会解除由该线程获得的所有对象锁，而且可能使对象处于不连贯状态，如果其他线程此时访问对象，而导致的错误很难检查出来。suspend()/resume ()方法也极不安全，调用 suspend()方法时，线程会停下来，但是该线程并没有放弃对象的锁，导致其他线程并不能获得对象锁。调用 destroy()会强制终止线程，但是该线程也不会释放对象锁。

10.7　小结

Java 应用程序通过多线程技术共享系统资源，线程之间的通信与协同通过简单的方法调用完成。可以说，Java 语言对多线程的支持增强了 Java 作为网络程序设计语言的优势，为实现分布式应用系统中多用户并发访问，提高服务器效率奠定了基础。多线程编程是编写大型软件必备的技术，读者应该作为重点和难点学习。