5.1.2 用 Thread 类创建线程

Java 的线程是通过 java.lang.Thread 类来控制的，一个 Thread 类的对象代表一个线程，

而且只能代表一个线程，通过 Thread 类和它定义的对象，我们可以获得当前线程对象、获取某一线程的名称，可以实现控制线程暂停一段时间等功能，关于 Thread 类的具体应用与

程序清单：ThreadDemo1.java

```
public class ThreadDemo1
{
    public static void main(String args[])
    {
        new TestThread().run();
        while(true)
        {
            System.out.println("main thread is running");
        }
    }
}

class TestThread
{
    public void run()
    {
        while(true)
        {
            System.out.println(Thread.currentThread().getName() +
            "is running");
        }
    }
}
```

代码块 1

代码块 2

代码块 1 处的代码能否运行呢？编译 ThreadDemo1.java 文件，并运行一下，看看结果如何？

屏幕上不停地打印出 main is running，而不是 main thread is running，这说明代码块 1 处的程序没有运行，因为代码块 2 先于代码块 1 运行，且代码块 2 为无限循环，代码块 1 永远没有机会运行。同时，我们也能够看到当前线程的名称为 main。

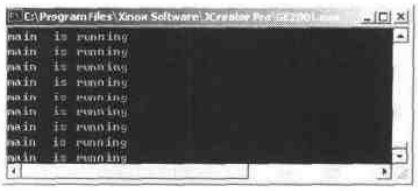

图 5.1

我们将代码进行如下修改（为了达到对比讲解，保持上下连贯性的效果，我们对修改过的地方进行注释，而不是彻底删除掉）：

程序清单：ThreadDemo2.java

```
public class ThreadDemo2
{
    public static void main(String args[])
    {
        new TestThread ().start(); /*run()*/

        while(true)
        {
            System.out.println("main thread is running");
        }
    }
}

    }
    class TestThread extends Thread
    {
        public void run()
        {
            while(true)
            {
                System.out.println(Thread.currentThread().getName() +
" is running");
            }
        }
    }
```

2. 调用 TestThread 类的 start 函数（从 Thread 类继承而来的）

1. 让 TestThread 类继承 Thread 类

上面的代码让 TestThread 类继承了 Thread 类，也就是 TestThread 类具有了 Thread 类的全部特点，程序没有直接调用 TestThread 类对象的 run 方法，而是调用了该类对象从 Thread 类继承来的 start 方法。运行一下，能够看到两个 while 循环处的代码同时交替运行，如图 5.2 所示。

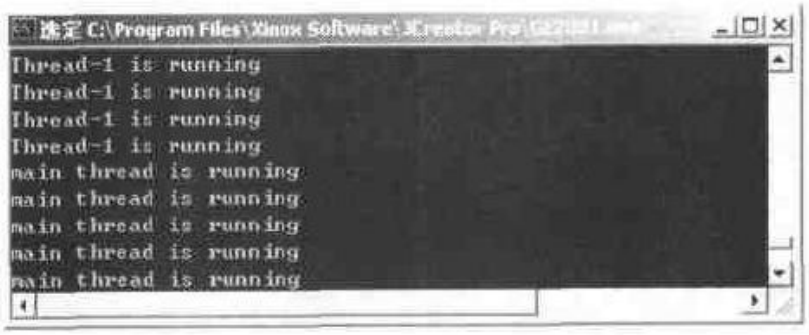

图　5.2

小结：

（1）要将一段代码在一个新的线程上运行，该代码应该在一个类的 run 函数中，并且 run 函数所在的类是 Thread 类的子类。倒过来看，要实现多线程，必须编写一个继承了 Thread 类的子类，子类要覆盖 Thread 类中的 run 函数，在子类的 run 函数中调用想在新线程上运行的程序代码。

（2）启动一个新的线程，不是直接调用 Thread 子类对象的 run 方法，而是调用 Thread 子类对象的 start（从 Thread 类中继承的）方法，Thread 类对象的 start 方法将产生一个新的线程，并在该线程上运行该 Thread 类对象中的 run 方法，根据面向对象的多态性，在该线程上实际运行的是 Thread 子类（也就是我们编写的那个类）对象中的 run 方法。

（3）由于线程的代码段在 run 方法中，那么该方法执行完以后，线程也就相应的结束了，因而可以通过控制 run 方法中的循环条件来控制线程的终止。

5.1.3　使用 Runnable 接口创建多线程

在 JDK 文档中，还看到了一个 Thread(Runnable target)构造方法，从 JDK 文档中查看 Runnable 接口类的帮助，该接口中只有一个 run()方法，当使用 Thread(Runnable target)方法创建线程对象时，需为该方法传递一个实现了 Runnable 接口的类对象，这样创建的线程将调用那个实现了 Runnable 接口的类对象中的 run()方法作为其运行代码，而不再调用 Thread 类中的 run 方法了。我们可以将上面的例子改写成下面这样：

程序清单：ThreadDemo3.java

```
public class ThreadDemo3
    {
        public static void main(String args[])
        {
            //new TestThread ().start();
            TestThread tt= new TestThread();//创建 TestThread 类的 一个实例

            Thread t= new Thread(tt);//创建一个 Thread 类的实例
            t.start();//使线程进入 Runnable 状态              while(true)
            {
                System.out.println("main thread is running");
            }
        }
```

```
}
class TestThread implements Runnable //extends Thread
{
    public void run()//线程的代码段，当执行 start()时，线程从此出开始执行
    {
        while(true)
        {
            System.out.println(Thread.currentThread().getName() +
            " is running");
        }
    }
}
```

可见，实现 Runnable 接口相对于继承 Thread 类来说，有如下显著的好处：

（1）适合多个相同程序代码的线程去处理同一资源的情况，把虚拟 CPU（线程）同程序的代码、数据有效分离，较好地体现了面向对象的设计思想。

（2）可以避免由于 Java 的单继承特性带来的局限。我们经常碰到这样一种情况，即当我们要将已经继承了某一个类的子类放入多线程中，由于一个类不能同时有两个父类，所以不能用继承 Thread 类的方式，那么，这个类就只能采用实现 Runnable 接口的方式了。

（3）有利于程序的健壮性，代码能够被多个线程共享，代码与数据是独立的。当多个线程的执行代码来自同一个类的实例时，即称它们共享相同的代码。多个线程可以操作相同的数据，与它们的代码无关。当共享访问相同的对象时，即它们共享相同的数据。当线程被构造时，需要的代码和数据通过一个对象作为构造函数实参传递进去，这个对象就是一个实现了 Runnable 接口的类的实例。

事实上，几乎所有多线程应用都可用第二种方式，即实现 Runnable 接口。

使用 runnable 的多线程代码

程序清单：ThreadDemo5.java

```
public class ThreadDemo5
{
    public static void main(String [] args)
    {
        ThreadTest t=new ThreadTest();
        new Thread(t).start();
        new Thread(t).start();
        new Thread(t).start();
        new Thread(t).start();
    }
}
class ThreadTest implements Runnable
{
    private int tickets=100;
    public void run()
    {
        while(true)
        {
            if(tickets>0)
                System.out.println(Thread.currentThread().getName()
                + " is saling ticket "  + tickets--);
        }
    }
}
```

在上面的程序中，创建了四个线程，每个线程调用的是同一个 ThreadTest 对象中的 run() 方法，访问的是同一个对象中的变量（tickets）的实例，这个程序满足了我们的需求。我们在 Windows 上可以启动多个记事本程序，也就是多个进程使用的是同一个记事本程序代码，明白了这个道理后，大家就应该对多个线程上运行完全相同的程序代码不再难以理解了。

2. 联合线程与 join 方法

在讲到联合线程之前，先来看下面的这段程序。

程序清单：JoinThread.java

```
public class JoinThread
{
    public static void main(String[] args)
    {
        ThreadTest t=new ThreadTest();
        Thread pp=new Thread(t);
        pp.start();
        int i=0;
        while(true)
        {
            if(i==100)
            {
                try
                {
                    pp.join();
                }
                catch(Exception e)
                {
                    System.out.println(e.getMessage());
                }
            }
            System.out.println("main Thread "+i++);
        }
    }
}
class ThreadTest implements Runnable
{
    public void run()
    {
        String str=new String();
        int i=0;
        while(true)
        {
            System.out.println(Thread.currentThread().getName()+" "+i++);
        }
    }
}
```

在上面的程序中用到了 Thread 类的 join 方法，即 pp.join();语句，它的作用是把 pp 所对应的线程合并到调用 pp.join();语句的线程中。在 main 线程中的循环计数达到 100 之前，

看到 main 线程和 Thread-1 线程交替执行的情况，如图 5.7 中打印出来的结果。

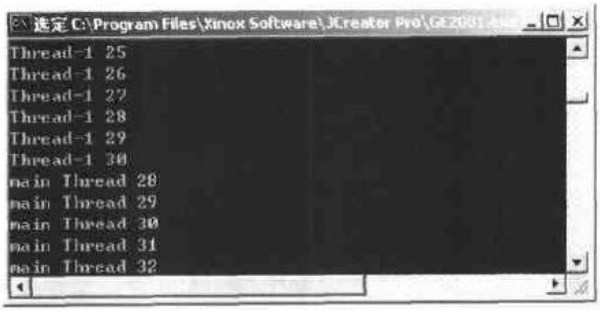

图 5.7

在 main 线程中的循环计数达到 100 之后，看到只有 Thread-1 线程执行的情况，如图 5.8 中打印出来的结果。

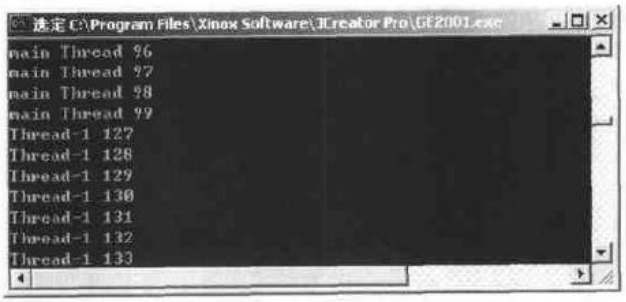

图 5.8

可见，Thread-1 线程中的代码被并入到了 main 线程中，也就是 Thread-1 线程中的代码不执行完，main 线程中的代码就只能一直等待。查看 JDK 文档，我们发现，除了有无参数的 join 方法外，还有两个带参数的 join 方法，分别是 join(long millis)和 join(long millis,int nanos)，它们的作用是指定合并时间，前者精确到毫秒，后者精确到纳秒，意思是两个线程合并指定的时间后，又开始分离，回到合并前的状态。读者可以把上面的程序中的 join 方法修改成为有参数的，再看看程序运行的结果。

时结束复制过程，具体的程序代码见本章最后的一节（如何控制线程的生命）。

3．多线程中的另外一个比较典型的例子就是 WWW 服务器，我们都知道，WWW 的服务器是可以同时为若干个浏览者服务的，这就需要它为每一个来访者都创建一个线程。如果它是单线程的话，在一个时间段就只能为一个人服务，其他人只有干等的份了。

5.2 多线程的同步

5.2.1 线程安全问题

在 5.14 节卖车票的程序代码中，极有可能碰到一种意外，就是同一张票号被打印两次或多次，也可能出现打印出 0 甚至负数的票号。这个意外发生在下面这部分代码处：

```
if(tickets>0)
      System.out.println(Thread.currentThread().getName() +
      " is saling ticket "  + tickets--);
```

假设 tickets 的值为 1 的时候，线程 1 刚执行完 if(tickets>0)这行代码，正准备执行下面的代码，就在这时，操作系统将 CPU 切换到了线程 2 上执行，此时 tickets 的值仍为 1，线程 2 执行完上面两行代码，tickets 的值变为 0 后，CPU 又切回到了线程 1 上执行，线程 1 不会再执行 if(tickets>0)这行代码，因为先前已经比较过了，并且比较的结果为真，线程 1 将直接往下执行这行代码：

```
System.out.println(Thread.currentThread().getName() +
 " is saling ticket "  + tickets--);
```

但此刻 tickets 的值已变为 0，屏幕打印出的将是 0。

要想立即见到这种意外，可用在程序中调用 Thread.sleep()静态方法来刻意造成线程间的这种切换，Thread.sleep()方法迫使线程执行到该处后暂停执行，让出 CPU 给别的线程，在指定的时间（这里是毫秒）后，CPU 回到刚才暂停的线程上执行。修改完的 ThreadTest 代码如下：

```
class ThreadTest implements Runnable
```

```
{
    private int tickets=100;
    public void run()
    {
        while(true)
        {
            if(tickets>0)
            {
                try
                {
                Thread.sleep(10);
                }
                catch(Exception e)
                {
                    System.out.println(e.getMessage());
                }
                System.out.println(Thread.currentThread().getName() +
                " is saling ticket "  + tickets--);
            }
        }
    }
}
```

在上面的程序代码中，我们故意造成线程执行完 if(tickets>0)语句后，执行 Thread.sleep(10)，以让出 CPU 给别的线程，让读者直接看到在这一时刻发生线程切换的情况。查看 JDK 文档中关于 Thread.sleep()方法的定义如下：

编译运行上面的程序，屏幕上打出的最后几行结果如下：

```
Thread-2 is saling ticket 3
Thread-3 is saling ticket 2
Thread-4 is saling ticket 1
Thread-1 is saling ticket 0
```

```
Thread-2 is saling ticket -1
Thread-3 is saling ticket  2
```

票号被打印出来了负数，这就显示了同一张票被卖了 4 次的意外发生。

这种意外问题，就是我们有时听到专业人士谈到的“线程安全”问题，也许某天就有人问你，你写的类是线程安全的吗？就是说你编写的那个类的同一个实例对象的方法在多个线程被调用，是否会出现类似上面的意外，不要到时听不明白人家的意思，那会给人很业余的印象。

你编写的线程代码 符合 线程安全吗

5.2.2 同步代码块

如何避免上面的这种意外？如何让我们的程序是线程安全的呢？这就是我们要为大家讲解的如何实现线程间的同步问题。要解决上面的问题，我们必须保证下面这段代码的原子性：

```
if(tickets>0)
        System.out.println(Thread.currentThread().getName() +
        " is saling ticket "  + tickets--);
```

即当一个线程运行到 if(tickets>0)后，CPU 不去执行其他线程中的、可能影响当前线程中的下一句代码的执行结果的代码块，必须等到下一句执行完后才能去执行其他线程中的有关代码块。这段代码就好比一座独木桥，任一时刻，只能有一个人在桥上行走，程序中不能有多个线程同时在这两句代码之间执行，这就是线程同步。

我们修改一下 ThreadTest 类，使其具有线程同步效果，代码如下：

程序中不能有多个线程同时对某段代码作用 这一段代码就线程同步了

我们修改一下 ThreadTest 类，使其具有线程同步效果，代码如下：

```
class ThreadTest implements Runnable
{
    private int tickets=100;
    String str = new String ("");
    public void run()
    {
        while(true)
        {
        synchronized(str)
            {
            if(tickets>0)
               {
                 try
                 {
```

```
                    Thread.sleep(10);
                   }
                   catch(Exception e)
                    {
                      System.out.println(e.getMessage());
                   }
                   System.out.println(Thread.currentThread().getName()+
                      " is saling ticket " + tickets--);
                 }
          }
       }
       }
}
```

在上面的代码中，我们将这些需要具有原子性的代码，放入 synchronized 语句内，形成了同步代码块。在同一时刻只能有一个线程可以进入同步代码块内运行，只有当该线程离开同步代码块后，其他线程才能进入同步代码块内运行。synchronized 语句的格式为：

```
synchronized(object){代码段} //object 可以是任意的一个对象
```

所以，程序中用 String str = new String ("")随便产生了一个对象，用在后面的同步代码块。编译运行后，程序在屏幕上打印出的内容如下：

```
Thread-2 is saling ticket 3
Thread-3 is saling ticket 2
Thread-4 is saling ticket 1
```

程序打印完最后的票号 1 后，就再也没有继续在屏幕上打印输出了，相当于这四个线程卖完最后一张票后，就停止了卖票的操作，这也向读者说明了改写后的程序是线程安全的了。

任意类型的对象都有一个标志位，该标志位具有 0、1 两种状态，其开始状态为 1，当执行 synchronized(object)语句后，object 对象的标志位变为 0 状态，直到执行完整个 synchronized 语句中的代码块后又回到 1 状态。一个线程执行到 synchronized(object)语句处时，先检查 object 对象的标志位，如果为 0 状态，表明已经有另外的线程的执行状态正在有关的同步代码块中，这个线程将暂时阻塞，让出 CPU 资源，直到另外的线程执行完有关的同步代码块，将 object 对象的标志位恢复到 1 状态，这个阻塞就被取消，线程能够继续往下执行，并将 object 对象的标志位变为 0 状态，防止其他线程再进入有关的同步代码块中。如果有多个线程因等待同一对象的标志位而处于阻塞状态时，当对象的标志位恢复到 1 状态时，只会有一个线程能够继续运行，其他线程仍然处于阻塞等待状态。我们反复提到有关的同步代码块，是指不仅同一个代码块在多个线程间可以实现同步（像上面例子一样），若干个不同的代码块也可以实现相互之间的同步，只要各 synchronized(object)语句中的 object 完全是同一个对象就可以。上面的讲解主要是为了达到通俗易懂的目的，但有时与同行交流，或是参看相关书籍时，我们也不得不掌握一些专业术语，下面是对刚才的内容用专业术语进行的陈述。

当线程执行到 [illegible] 的时候，检查传入的实参对象，并创建该对象的锁标记（即

大家也看到同步处理后，程序的运行速度比原来没有使用同步处理前更慢了，因为系统要不停地对同步监视器进行检查，需要更多的开销。同步是以牺牲程序的性能为代价的，如果我们能够确定程序没有安全性的问题，就没必要使用同步控制。

我们将程序代码略作修改，改变 String str = new String ("")这行代码的位置，将 str 对象放到 run 方法中定义：

```
class ThreadTest implements Runnable
{
    private int tickets=100;
    public void run()
    {
        String str = new String ("");
        while(true)
        {
        synchronized(str)
            {
            if(tickets>0)
               {
                 try
```

如果改变 str 的位置，放于 run 方法中，线程安全会出问题

编译运行后，发现结果又不正常了，问题出在什么地方呢？在这个程序中，run 方法被四个线程所调用，相当于 run 方法被调用了四次，对每一次调用，程序都产生一个不同的 str 局部对象，这四个线程使用的同步监视器完全是四个不同的对象，所以彼此之间不能同步。

5.2.3 同步函数

除了可以对代码块进行同步外，也可以对函数实现同步，只要在需要同步的函数定义前加上 synchronized 关键字即可，我们按下面的代码修改 ThreadTest 类：

```
class ThreadTest implements Runnable
{
    private int tickets=100;
    public void run()
    {
        while(true)
        {
       sale();
    }
    }
    public synchronized void sale()
    {
    if(tickets>0)
        {
        try
        {


            Thread.sleep(10);
          }
          catch(Exception e)
          {
              System.out.println(e.getMessage());
          }
          System.out.println(Thread.currentThread().getName()+
              " is saling ticket " + tickets--);
          }
      }
  }
```

编译运行后的结果同上面同步代码块方式的运行结果完全一样，可见，在函数定义前使用 synchronized 关键字也能够很好地实现线程间的同步。

在同一类中，使用 synchronized 关键字定义的若干方法，可以在多个线程之间同步，当有一个线程进入了 synchronized 修饰的方法（获得监视器），其他线程就不能进入同一个对象的所有使用了 synchronized 修饰的方法，直到第一个线程执行完它所进入的 synchronized 修饰的方法为止（离开监视器）。

5.3 线程间的通信

5.3.1 问题的引出

我们通过这样的一个应用来讲解线程间的通信。有一个数据存储空间，划分为两部分，一部分用于存储人的姓名，另一部分用于存储人的性别。我们的应用包含两个线程，一个线程向数据存储空间添加数据（生产者），另一个线程从数据存储空间中取出数据（消费者）。这个程序有两种意外需要我们考虑：

第一个意外，假设生产者线程刚向数据存储空间中添加了一个人的姓名，还没有加入这个人的性别，CPU 就切换到了消费者线程，消费者线程将把这个人的姓名和上一个人的性别联系到了一起。这个过程可用图 5.12 表示。

5.3.2 问题如何解决

我们先来构思这个程序例子，程序中的生产者线程和消费者线程运行的是不同的程序代码，因此我们需要编写两个包含有 run 方法的类来完成这两个线程，一个是生产者类 Producer，一个是消费者类 Consumer。

```
class Producer implements Runnable
{
    public void run()
{
    while(true)
    {
        编写往数据存储空间中放入数据的代码
    }
}
}
class Consumer implements Runnable
{
    public void run()
```

```
{
    while(true)
    {
        编写从数据存储空间中读取数据的代码
    }
}
}
```

当程序写到这里，我们发现还需要定义一个新的数据结构来作为数据存储空间。

```
class Q
{
    String name;
    String sex;
}
```

Producer 和 Consumer 中的 run 函数都需要操作类 Q 的同一个对象实例，接下来，对 Producer 和 Consumer 这两个类作如下修改，顺便也写出程序的主调用类 ThreadCommunation。

当程序写到这里，我们发现还需要定义一个新的数据结构来作为数据存储空间。

```
class Q
{
    String name;
    String sex;
}
```

Producer 和 Consumer 中的 run 函数都需要操作类 Q 的同一个对象实例，接下来，对 Producer 和 Consumer 这两个类作如下修改，顺便也写出程序的主调用类 ThreadCommunation。

两个函数放在同一个类中编写，是不是容易和清晰得多呢？我们再将代码修改成下面这样：

死　锁　的　解　决　办　法

```
class C
{
    private String name = "陈琼";
    private String sex = "女";
    public synchronized void put(String name,String sex)
    {
        this.name=name;
    try
    {
        Thread.sleep(10);
    }
    catch(Exception e)
    {
        System.out.println(e.getMessage());
    }
        this.sex=sex;
    }
```

5.4.2 如何控制线程的生命

程序清单：ThreadLife.java

```
public class ThreadLife
{
    public static void main(String[] args)
    {
        ThreadTest t=new ThreadTest();
        new Thread(t).start();
        for(int i=0;i<100;i++)
        {
            if(i == 50)
                t.stopMe();
            System.out.println("mainThread is running");
        }
    }
}
```

```
class ThreadTest implements Runnable
{
    private boolean bFlag = true;
    public void stopMe()
    {
        bFlag = false;
    }
    public void run()
    {
        while(bFlag)
        {
            System.out.println(Thread.currentThread().getName()+
                " is running ");
        }
    }
}
```

运行结果：

```
......
Thread-1 is running
Thread-1 is running
Thread-1 is running
Thread-1 is running
mainThread is running
mainThread is running
mainThread is running
```



```
mainThread is running
......
```

.....

上面的程序中定义了一个计数器 i，用来控制 main 线程的循环打印次数，在 i 的值从 0 到 50 的这段时间内，两个线程是交替运行的，但当计数器 i 的取值变为 50 的时候，程序调用了 ThreadTest 类的 stopMe 方法，而在 stopMe 方法中，将 bFlag 变量赋值为 false，也就是终止了 while 循环，run 方法结束，Thread-1 线程随之结束。main 线程在计数器 i 等于 50 的时候，调用了 ThreadTest 类的 stopMe 方法后，CPU 不一定会马上切换到 Thread-1 线程上，也就是说 Thread-1 线程不一定会马上终止，main 线程的计数器 i 可能到达五十几甚至六十几后，Thread-1 线程才真正结束。

综上所述，我们推荐使用控制 run 方法中循环条件的方式来结束一个线程，这也是实际情况中用的最多的。

控制 run 方法中的的循环条件来决定一个线程的生死