



“十四五”职业教育国家规划教材

工业和信息化“十三五”
人才培养规划教材

Java Basic Case Tutorial
2nd Edition

Java 基础

案例教程

第2版



黑马程序员 编著



中国工信出版集团



人民邮电出版社
POSTS & TELECOM PRESS

6.6.3 泛型方法	159	8.2 线程的创建	183
6.6.4 泛型接口	160	8.2.1 继承 Thread 类创建多线程	183
6.6.5 类型通配符	161	8.2.2 实现 Runnable 接口创建多线程	185
6.7 JDK 8 新特性——Lambda 表达式	162	8.2.3 两种实现多线程方式的对比分析	186
6.8 本章小结	163	8.3 线程的生命周期及状态转换	188
6.9 本章习题	163	8.4 线程的调度	189
第7章 I/O (输入/输出)	164	8.4.1 线程的优先级	189
7.1 File 类	164	8.4.2 线程休眠	190
7.1.1 创建 File 对象	164	【案例 8-1】 龟兔赛跑	191
7.1.2 File 类的常用方法	165	8.4.3 线程让步	192
7.1.3 遍历目录下的文件	167	8.4.4 线程插队	192
7.1.4 删除文件及目录	169	【案例 8-2】 Svip 优先办理服务	193
【案例 7-1】 批量操作文件管理器	170	8.5 多线程同步	193
7.2 字节流	170	8.5.1 线程安全问题	193
7.2.1 字节流的概念	170	8.5.2 同步代码块	194
7.2.2 InputStream 读文件	172	8.5.3 同步方法	195
7.2.3 OutputStream 写文件	173	8.5.4 死锁问题	197
7.2.4 文件的复制	175	【案例 8-3】 模拟银行存取钱	198
7.2.5 字节缓冲流	176	【案例 8-4】 工人搬砖	198
【案例 7-2】 商城进货交易记录	177	【案例 8-5】 小朋友就餐	198
【案例 7-3】 日记本	177	8.6 本章小结	198
7.3 字符流	177	8.7 本章习题	198
7.3.1 字符流定义及基本用法	177	第9章 网络编程	199
7.3.2 字符流操作文件	178	9.1 网络通信协议	199
7.3.3 转换流	180	9.1.1 IP 地址和端口号	200
【案例 7-4】 升级版日记本	181	9.1.2 InetAddress	201
【案例 7-5】 微信投票	181	9.1.3 UDP 与 TCP	201
7.4 本章小结	181	9.2 UDP 通信	202
7.5 本章习题	181	9.2.1 DatagramPacket	203
第8章 多线程	182	9.2.2 DatagramSocket	203
8.1 线程概述	182	9.2.3 UDP 网络程序	204
8.1.1 进程	182	9.2.4 多线程的 UDP 网络程序	206
8.1.2 线程	183	【案例 9-1】 模拟微信聊天	207

第 8 章

多线程

学习目标

- ★ 了解线程与进程的区别
- ★ 掌握创建多线程的两种方式
- ★ 了解线程的生命周期及状态转换
- ★ 掌握线程的调度
- ★ 掌握多线程的同步



拓展阅读

多线程是提升程序性能非常重要的一种方式，也是学习 Java 编程必须要掌握的技术。使用多线程可以让程序充分利用 CPU 的资源，提高 CPU 的使用效率，从而解决高并发带来的负载均衡问题。本章将针对 Java 中的多线程知识进行详细地讲解等。

8.1 线程概述

计算机能够同时完成多项任务，例如，一边访问浏览器，一边使用 QQ 进行聊天，这就是多线程技术。计算机中的中央处理器（Central Processing Unit，CPU）即使是单核也可以同时运行多个任务，因为操作系统执行多个任务时就是让 CPU 对多个任务轮流交替执行。Java 是支持多线程的语言之一，它对多线程编程提供了内置的支持，可以使程序同时执行多个执行片段。本章将对 Java 多线程的相关知识进行详细讲解。

8.1.1 进程

在学习线程之前，需要先了解什么是进程。在一个操作系统中，每个独立执行的程序都可称为一个进程，也就是“正在运行的程序”。目前，大部分计算机上安装的都是多任务操作系统，即能够同时执行多个应用程序，最常见的有 Windows、Linux、UNIX 等。在本教材使用的 Windows 操作系统下，右键单击任务栏、选择“启动任务管理器”选项可以打开“Windows 任务管理器”窗口，在窗口的“进程”选项卡中可以看到当前正在运行的程序，也就是系统所有的进程，如 chrome.exe、QQ.exe 等，如图 8-1 所示。

在多任务操作系统中，表面上看是支持进程并发执行的，例如可以一边听音乐一边聊天。但实际上这些进程并不是同时运行的。在计算机中，所有的应用程序都是由 CPU 执行的，对于一个 CPU 而言，在某个时间点只能运行一个程序，也就是说，只能执行一个进程。操作系统会为每一个进程分配一段有限的 CPU 使用时间，CPU 在这段时间中执行某个进程，然后会在下一段时间去执行另一个进程。由于 CPU 运行速度很快，能在极短的时间内在不同的进程之间进行切换，所以给人同时执行多个程序的感觉。

8.1.2 线程

通过 8.1.1 小节的学习可以知道，每个运行的程序都是一个进程，在一个进程中还可以有多个执行单元同时运行，这些执行单元可以看作程序执行的一条条线索，称为线程。操作系统中的每一个进程中至少存在一个线程。例如，当一个 Java 程序启动时，就会产生一个进程，该进程中会默认创建一个线程，在这个线程上会运行 `main()` 方法中的代码。

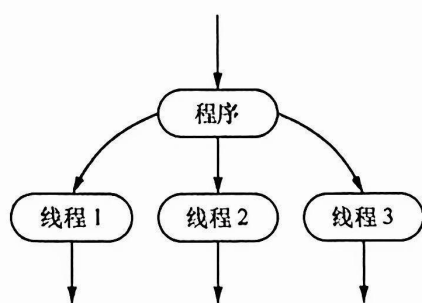


图8-2 多线程程序的执行过程

在前面章节所接触过的程序中，代码都是按照调用顺序依次往下执行的，没有出现两段程序代码交替运行的效果，这样的程序称为单线程程序。如果希望程序中实现多段程序代码交替运行的效果，则需要创建多个线程，即多线程程序。多线程，是指一个进程在执行过程中可以产生多个单线程，这些单线程程序在运行时是相互独立的，它们可以并发执行。多线程程序的执行过程如图 8-2 所示。

图 8-2 所示的多条线程看似是同时执行的，其实不然，它们与进程一样，也是由 CPU 轮流执行的，只不过 CPU 运行速度很快，故给人同时执行的感觉。

8.2 线程的创建

8.1 节介绍了什么是多线程，下面为读者讲解在 Java 程序中如何实现多线程。在 Java 中提供了两种多线程实现方式：一种是继承 `java.lang` 包下的 `Thread` 类，覆写 `Thread` 类的 `run()` 方法，在 `run()` 方法中实现运行在线程上的代码；另一种是实现 `java.lang.Runnable` 接口，同样是在 `run()` 方法中实现运行在线程上的代码。下面就对创建多线程的两种方式分别进行讲解，并比较它们的优缺点。

8.2.1 继承 Thread 类创建多线程

在学习多线程之前，先来看看我们所熟悉的单线程程序，如文件 8-1 所示。

文件 8-1 Example01.java

```

1 public class Example01 {
2     public static void main (String[] args) {
3         MyThread myThread = new MyThread (); // 创建 MyThread 实例对象
4         myThread.run (); // 调用 MyThread 类的 run () 方法
5         while (true) { // 该循环是一个死循环，打印输出语句
6             System.out.println ("Main 方法在运行");
7         }
8     }
}

```



图8-1 任务管理器窗口


```

9  }
10 class MyThread {
11     public void run () {
12         while (true) { // 该循环是一个死循环, 打印输出语句
13             System.out.println ("MyThread类的 run () 方法在运行");
14         }
15     }
16 }

```

文件 8-1 的运行结果如图 8-3 所示。

从图 8-3 所示的运行结果可以看出, 程序一直打印 “MyThread 类的 run () 方法在运行”, 这是因为该程序是一个单线程程序, 在文件 8-1 的第 4 行代码调用 MyThread 类的 run () 方法时, 执行到 MyThread 类中第 12~14 行代码定义的死循环, 循环会一直进行。因此, MyThread 类的打印语句将被无限执行, 而 main () 方法中的打印语句无法得到执行。

如果希望文件 8-1 中两个 while 循环中的打印语句能够并发执行, 就需要实现多线程。为此 Java 提供了一个线程类 Thread, 通过继承 Thread 类, 并重写 Thread 类中的 run () 方法便可实现多线程。在 Thread 类中, 提供了一个 start () 方法用于启动新线程, 线程启动后, 虚拟机会自动调用 run () 方法, 如果子类重写了该方法便会执行子类中的方法。下面通过修改文件 8-1 中的案例来演示如何通过继承 Thread 类的方式来实现多线程, 如文件 8-2 所示。

文件 8-2 Example02.java

```

1  public class Example02 {
2      public static void main (String[] args) {
3          MyThread myThread = new MyThread (); // 创建线程 MyThread 的线程对象
4          myThread.start (); // 开启线程
5          while (true) { // 通过死循环语句打印输出
6              System.out.println ("main () 方法在运行");
7          }
8      }
9  }
10 class MyThread extends Thread {
11     public void run () {
12         while (true) { // 通过死循环语句打印输出
13             System.out.println ("MyThread类的 run () 方法在运行");
14         }
15     }
16 }

```

文件 8-2 的运行结果如图 8-4 所示。

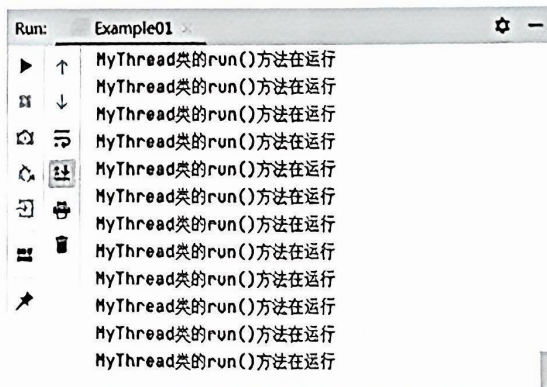


图8-3 文件8-1的运行结果

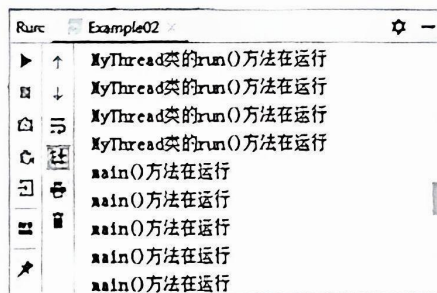


图8-4 文件8-2的运行结果

在文件 8-2 中, 第 5~7 行代码定义了一个 while 死循环, 并在循环中打印 “main () 方法在运行”; 第 12~14 行代码也定义了一个 while 死循环, 并在循环中打印 “MyThread 类的 run () 方法在运行”。利用两个 while 来模拟多线程环境, 从图 8-4 所示的运行结果可以看到, 两个循环中的语句都有输出, 说明该文件实现了多线程。为了使读者更好地理解单线程和多线程的执行过程, 下面通过一个图例分析单线程和多线程的区别, 如图 8-5 所示。

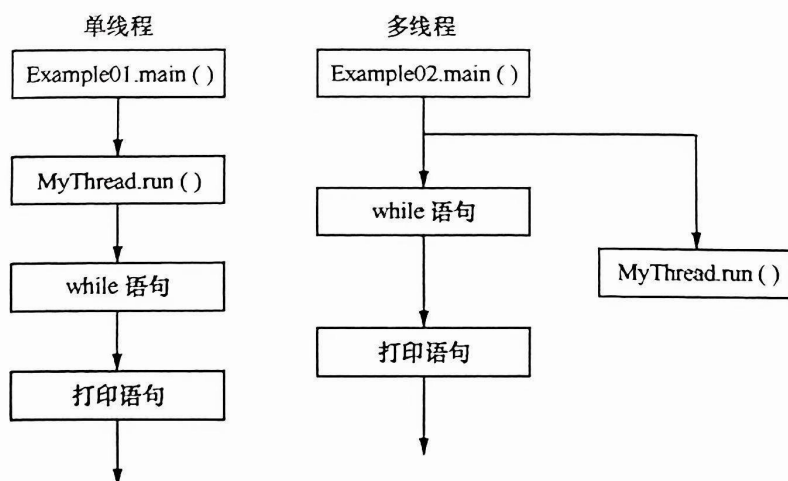


图8-5 单线程和多线程的区别

从图 8-5 可以看出，单线程的程序在运行时，会按照代码的调用顺序执行，而在多线程中，main（）方法和 MyThread 类的 run（）方法却可以同时运行，互不影响，这正是单线程和多线程的区别。

8.2.2 实现 Runnable 接口创建多线程

在文件 8-2 中通过继承 Thread 类实现了多线程，但是这种方式有一定的局限性。因为 Java 只支持单继承，一个类一旦继承了某个父类就无法再继承 Thread 类，例如学生类 Student 继承了 Person 类，就无法通过继承 Thread 类创建线程。

为了克服这种弊端，Thread 类提供了另外一个构造方法 Thread（Runnable target），其中 Runnable 是一个接口，它只有一个 run（）方法。当通过 Thread（Runnable target）构造方法创建线程对象时，只需为该方法传递一个实现了 Runnable 接口的实例对象，这样创建的线程将调用实现了 Runnable 接口的类中的 run（）方法作为运行代码，而不需要调用 Thread 类中的 run（）方法。

下面通过一个案例来演示如何通过实现 Runnable 接口的方式来创建多线程，如文件 8-3 所示。

文件 8-3 Example03.java

```

1 public class Example03 {
2     public static void main (String[] args) {
3         MyThread myThread = new MyThread (); // 创建 MyThread 的实例对象
4         Thread thread = new Thread (myThread); // 创建线程对象
5         thread.start (); // 开启线程，执行线程中的 run () 方法
6         while (true) {
7             System.out.println ("main () 方法在运行");
8         }
9     }
10 }
11 class MyThread implements Runnable {
12     public void run () { // 线程的代码段，当调用 start () 方法时，线程从此处开始执行
13         while (true) {
14             System.out.println ("MyThread 类的 run () 方法在运行");
15         }
16     }
17 }

```

文件 8-3 的运行结果如图 8-6 所示。

文件 8-3 中，第 11~17 行代码定义的 MyThread 类实现了 Runnable 接口，并在第 12~16 行代码中重写了 Runnable 接口中的 run（）方法；第 4 行代码中通过 Thread 类的构造方法将 MyThread 类的实例对象作为参数传入，第 5 行代码中使用 start（）方法开启 MyThread 线程，最后在第 6~8 行代码中定义了一个 while 死循环。从图 8-6 的运行结果可以看出，main（）方法和 run（）方法中的打印语句都执行了，说明文件 8-3 实现了多线程。

8.2.3 两种实现多线程方式的对比分析

既然直接继承 Thread 类和实现 Runnable 接口都能实现多线程,那么这两种实现多线程的方式在实际应用中又有什么区别呢?下面通过一种应用场景来分析。

假设售票厅有 4 个窗口可发售某日某次列车的 100 张车票,这时,100 张车票可以看作共享资源、4 个售票窗口需要创建 4 个线程。为了更直观地显示窗口的售票情况,可以通过 Thread 的 currentThread() 方法得到当前线程的实例对象,然后调用 getName() 方法可以获取到线程的名称。

首先通过继承 Thread 类的方式创建多线程,如文件 8-4 所示。

文件 8-4 Example04.java

```
1 public class Example04 {
2     public static void main (String[] args) {
3         new TicketWindow ().start (); // 创建第一个线程对象 TicketWindow 并开启
4         new TicketWindow ().start (); // 创建第二个线程对象 TicketWindow 并开启
5         new TicketWindow ().start (); // 创建第三个线程对象 TicketWindow 并开启
6         new TicketWindow ().start (); // 创建第四个线程对象 TicketWindow 并开启
7     }
8 }
9 class TicketWindow extends Thread {
10     private int tickets = 100;
11     public void run () {
12         while (true) { // 通过死循环语句打印语句
13             if (tickets > 0) {
14                 Thread th = Thread.currentThread (); // 获取当前线程
15                 String th name = th.getName (); // 获取当前线程的名字
16                 System.out.println ("线程名称 " + th name + " 正在发售票 " + tickets + " 张票");
17             }
18         }
19     }
20 }
```

文件 8-4 的运行结果如图 8-7 所示。

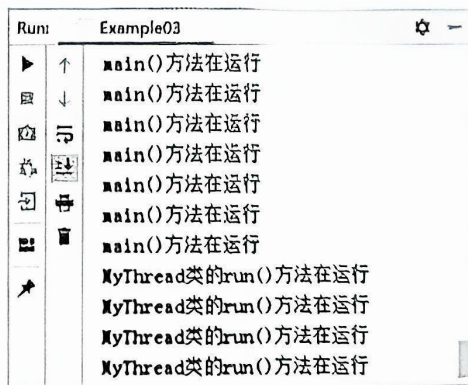


图8-6 文件8-3的运行结果



图8-7 文件8-4的运行结果

从图 8-7 所示的运行结果可以看出,每张票都被打印了 4 次。出现这种现象的原因是 4 个线程没有共享 100 张票,而是各自出售了 100 张票。在程序中创建了 4 个 TicketWindow 对象,就等于创建了 4 个售票程序,每个程序中都有 100 张票,每个线程在独立地处理各自的资源。需要注意的是,文件 8-4 中每个线程都有自己的名字,主线程默认的名字是“main”,用户创建的第一个线程的名字默认为“Thread-0”,第二个线程的名字默认为“Thread-1”,依次类推。如果希望指定线程的名称,可以通过调用 setName (String name) 方法为线程设置名称。

由于现实中铁路系统的票资源是共享的,因此上面的运行结果显然不合理。为了保证资源共享,在程序中只能创建一个售票对象,然后采用开启多个线程去运行同一个售票对象的售票方法。简单来说,就是 4 个线程运行同一个售票程序,这时就需要用到多线程的第二种实现方式。

下面通过实现 Runnable 接口的方式来创建多线程。对文件 8-4 进行修改,并使用构造方法 Thread

(Runnable target, String name) 在创建线程对象时指定线程的名称, 如文件 8-5 所示。

文件 8-5 Example05.java

```

1 public class Example05 {
2     public static void main (String[] args) {
3         TicketWindow tw = new TicketWindow ();           // 创建 TicketWindow 实例对象 tw
4         new Thread (tw, "窗口 1").start ();               // 创建线程对象并命名为窗口 1, 开启线程
5         new Thread (tw, "窗口 2").start ();               // 创建线程对象并命名为窗口 2, 开启线程
6         new Thread (tw, "窗口 3").start ();               // 创建线程对象并命名为窗口 3, 开启线程
7         new Thread (tw, "窗口 4").start ();               // 创建线程对象并命名为窗口 4, 开启线程
8     }
9 }
10 class TicketWindow implements Runnable {
11     private int tickets = 100;
12     public void run () {
13         while (true) {
14             if (tickets > 0) {
15                 Thread th = Thread.currentThread (); // 获取当前线程
16                 String th_name = th.getName ();      // 获取当前线程的名字
17                 System.out.println (th_name + " 正在发售第 " + tickets-- + " 张票 ");
18             }
19         }
20     }
21 }

```

文件 8-5 的运行结果如图 8-8 所示。

在文件 8-5 中, 第 10~21 行代码创建了一个 TicketWindow 对象并实现了 Runnable 接口, 然后在 main 方法中创建了 4 个线程, 每个线程都去调用这个 TicketWindow 对象中的 run () 方法, 这样就可以确保 4 个线程访问的是同一个 tickets 变量, 共享 100 张车票。

通过文件 8-4 继承 Thread 类创建多线程和文件 8-5 实现 Runnable 接口创建多线程可以看出, 实现 Runnable 接口相对于继承 Thread 类来说, 具有以下优势。

(1) 适合多个相同程序代码的线程去处理同一个资源的情况, 把线程同程序代码、数据有效分离, 很好地体现了面向对象的设计思想。

(2) 可以避免由于 Java 的单继承带来的局限性。在开发中经常碰到这样一种情况, 即使用一个已经继承了某一个类的子类创建线程, 由于一个类不能同时有两个父类, 因此不能使用继承 Thread 类的方式, 只能采用实现 Runnable 接口的方式。

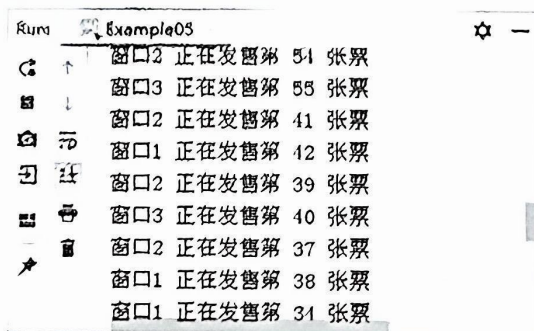


图8-8 文件8-5的运行结果

小提示:

JDK 8 简化了多线程的创建方法, 在创建线程时指定线程要调用的方法, 格式如下:

```

Thread t = new Thread ( () -> {
    //main 方法代码
} );

```

下面通过一个案例来讲解, 具体代码如下:

```

1 public class Main {
2     public static void main (String[] args) {
3         Thread t = new Thread ( () -> {
4             while (true) {
5                 System.out.println ("start new thread!");
6             }
7         });
8         t.start (); // 启动新线程
9     }
10 }
11 class MyThread extends Thread {
12     public void run () {
13         while (true) { // 通过死循环语句打印输出

```



```

14         System.out.println("MyThread 类的 run () 方法在运行");
15     }
16 }
17 )

```

上述代码第 3~7 行代码使用了 JDK 8 中新增的多线程创建方法, 其运行结果与文件 8-3 类似。

8.3 线程的生命周期及状态转换

在 Java 中, 任何对象都有生命周期, 线程也不例外, 它也有自己的生命周期。当 Thread 对象创建完成时, 线程的生命周期便开始了。当 run () 方法中代码正常执行完毕或者线程抛出一个未捕获的异常 (Exception) 或者错误 (Error) 时, 线程的生命周期便会结束。线程的整个生命周期可以分为 5 个阶段, 分别是新建状态 (New)、就绪状态 (Runnable)、运行状态 (Running)、阻塞状态 (Blocked) 和死亡状态 (Terminated), 线程的不同状态表明了线程当前正在进行的活动。在程序中, 通过一些操作可以使线程在不同状态之间转换, 如图 8-9 所示。

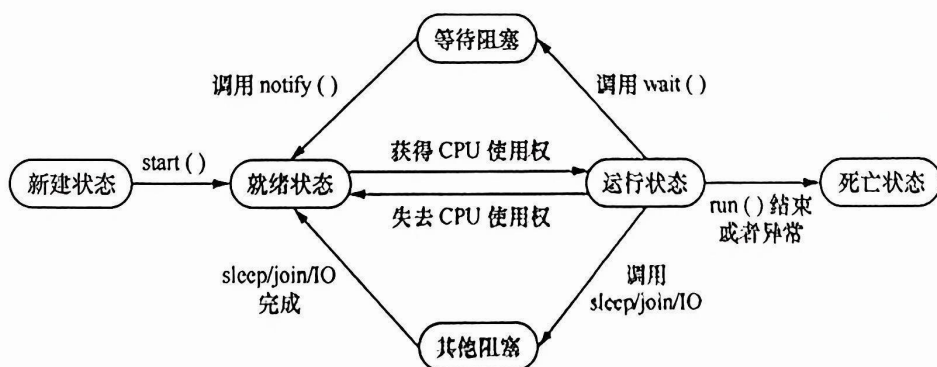


图8-9 线程状态转换图

图 8-9 中展示了线程各种状态的转换关系, 箭头表示可转换的方向, 其中, 单箭头表示状态只能单向的转换, 例如, 线程只能从新建状态转换到就绪状态, 反之则不能; 双箭头表示两种状态可以互相转换, 例如, 就绪状态和运行状态可以互相转换。通过一张图还不能完全描述清楚线程各状态之间的区别, 接下来针对线程生命周期中的五种状态分别进行详细讲解, 具体如下。

1. 新建状态 (New)

创建一个线程对象后, 该线程对象就处于新建状态, 此时它不能运行, 与其他 Java 对象一样, 仅仅由 Java 虚拟机为其分配了内存, 没有表现出任何线程的动态特征。

2. 就绪状态 (Runnable)

当线程对象调用了 start () 方法后, 该线程就进入就绪状态。处于就绪状态的线程位于线程队列中, 此时它只是具备了运行的条件, 能否获得 CPU 的使用权并开始运行, 还需要等待系统的调度。

3. 运行状态 (Running)

如果处于就绪状态的线程获得了 CPU 的使用权, 并开始执行 run () 方法中的线程执行体, 则该线程处于运行状态。一个线程启动后, 它可能不会一直处于运行状态, 当运行状态的线程使用完系统分配的时间后, 系统就会剥夺该线程占用的 CPU 资源, 让其他线程获得执行的机会。需要注意的是, 只有处于就绪状态的线程才可能转换到运行状态。

4. 阻塞状态 (Blocked)

一个正在执行的线程在某些特殊情况下, 如被人为挂起或执行耗时的输入/输出操作时, 会让出 CPU 的使用权并暂时中止自己的执行, 进入阻塞状态。线程进入阻塞状态后, 就不能进入排队队列。只有当引起阻塞的原因被消除后, 线程才可以转入就绪状态。

下面就列举一下线程由运行状态转换成阻塞状态的原因, 以及如何从阻塞状态转换成就绪状态。

- 当线程试图获取某个对象的同步锁时，如果该锁被其他线程所持有，则当前线程会进入阻塞状态，如果想从阻塞状态进入就绪状态就必须获取到其他线程所持有的锁。

- 当线程调用了一个阻塞式的 I/O 方法时，该线程就会进入阻塞状态，如果想进入就绪状态就必须等到这个阻塞的 I/O 方法返回。

- 当线程调用了某个对象的 wait () 方法时，也会使线程进入阻塞状态，如果想进入就绪状态就需要使用 notify () 方法唤醒该线程。

- 当线程调用了 Thread 的 sleep (long millis) 方法时，也会使线程进入阻塞状态，在这种情况下，只需等到线程睡眠的时间到了后，线程就会自动进入就绪状态。

- 当在一个线程中调用了另一个线程的 join () 方法时，会使当前线程进入阻塞状态，在这种情况下，需要等到新加入的线程运行结束后才会结束阻塞状态，进入就绪状态。

需要注意的是，线程从阻塞状态只能进入就绪状态，而不能直接进入运行状态，也就是说，结束阻塞的线程需要重新进入可运行池中，等待系统的调度。

5. 死亡状态 (Terminated)

如果线程调用 stop () 方法或 run () 方法正常执行完毕，或者线程抛出一个未捕获的异常 (Exception)、错误 (Error)，线程就进入死亡状态。一旦进入死亡状态，线程将不再拥有运行的资格，也不能再转换到其他状态。

8.4 线程的调度

在前文介绍过，程序中的多个线程是并发执行的，某个线程若想被执行必须要得到 CPU 的使用权。Java 虚拟机会按照特定的机制为程序中的每个线程分配 CPU 的使用权，这种机制称为线程的调度。

在计算机中，线程调度有两种模型，分别是分时调度模型和抢占式调度模型。分时调度模型，是指让所有的线程轮流获得 CPU 的使用权，并且平均分配每个线程占用 CPU 的时间片。抢占式调度模型，是指让可运行池中优先级高的线程优先占用 CPU，而对于优先级相同的线程，随机选择一个线程使其占用 CPU，当它失去了 CPU 的使用权后，再随机选择其他线程获取 CPU 使用权。Java 虚拟机默认采用抢占式调度模型，通常情况下程序员不需要去关心它，但在某些特定的需求下需要改变这种模式，由程序自己来控制 CPU 的调度。本节将围绕线程调度的相关知识进行详细讲解。

8.4.1 线程的优先级

在应用程序中，如果要对线程进行调度，最直接的方式就是设置线程的优先级。优先级越高的线程获得 CPU 执行的机会越大，而优先级越低的线程获得 CPU 执行的机会越小。线程的优先级用 1~10 的整数来表示，数字越大优先级越高。除了可以直接使用数字表示线程的优先级外，还可以使用 Thread 类中提供的 3 个静态常量表示线程的优先级，如表 8-1 所示。

表 8-1 Thread 类的优先级常量

Thread 类的静态常量	功能描述
static int MAX_PRIORITY	表示线程的最高优先级，值为 10
static int MIN_PRIORITY	表示线程的最低优先级，值为 1
static int NORM_PRIORITY	表示线程的普通优先级，值为 5

程序在运行期间，处于就绪状态的每个线程都有自己的优先级，例如，main 线程具有普通优先级。然而线程优先级不是固定不变的，可以通过 Thread 类的 setPriority (int newPriority) 方法进行设置，setPriority () 方法中的参数 newPriority 接收的是 1~10 的整数或者 Thread 类的 3 个静态常量。下面通过一个案例演示不同优先级的两个线程在程序中的运行情况，如文件 8-6 所示。