第 14 章

C#多线程编程技术——多线程操作

学习指引

线程，有时被称为轻量进程（Lightweight Process，LWP），是程序执行流的最小单元。线程是程序中一个单一的顺序控制流程。在单个程序中同时运行多个线程完成不同的工作，称为多线程。

重点导读

- 熟悉线程与进程的概念。
- 掌握 Thread 类中的方法及属性。
- 掌握线程的基本操作。
- 掌握线程的优先级。

14.1 进程与线程

进程（Process）和线程（Thread）是操作系统的基本概念，但是它们比较抽象并且不容易掌握。读者可以将进程理解为程序在计算机上的一次执行活动，而线程则是进程的一个实体。执行线程就体现程序的真实执行情况。

14.1.1 进程的概念

1. 进程

进程是程序在计算机上的一次执行活动。运行一个程序就相当于启动一个进程。Windows 系统利用进程把工作划分为多个独立的区域，每个应用程序实例对应一个进程。进程是操作系统分配和使用系统资源的基本单位。进程包含正在运行时应用程序的所有资源。每个进程所占用的资源都是相互独立的。

进程资源包括：

（1）一个进程堆；

（2）一个或多个线程；
（3）一个虚拟地址空间，该空间独立于其他进程的地址空间；
（4）一个或多个代码段，包括.dll 中的代码；
（5）一个或多个包含全局变量的数据段；
（6）环境字符串，包含环境变量信息；
（7）其他资源，例如打开的句柄、其他的堆等。

2. 多进程

多进程就是在同一计算机系统中，同一个时刻允许两个或两个以上的进程处于运行状态。多进程具有以下特点：

（1）进程间互相独立，可靠性高。
（2）进程之间不共享数据，没有锁问题，结构简单。
（3）需要跨进程边界，多进程调度开销较大。

在 Windows 操作系统中，通过访问 Windows 任务管理器可以查看当前正在运行的进程。单击详细信息，可看到进程的 PID、CPU 使用率、内存使用率等信息，应用程序可以包含一个或多个进程，每个进程都有自己独立的数据、执行代码和系统资源，如图 14-1 所示。

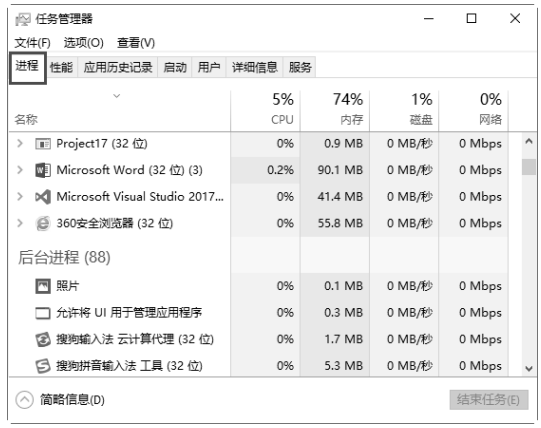

图 14-1　Windows 任务管理器

14.1.2　线程的概念

1. 线程

线程是程序中的一个执行流，每个线程都有自己的专有寄存器（栈、指针、程序计数器等），但代码区是共享的，即不同的线程可以执行同样的函数。

注意：一个进程可以有多个线程，一个线程必须有一个父进程，一个线程可以创建和撤销另一个线程，同一个进程中的多个线程之间可以并发执行。

2. 多线程

多线程是指程序中包含多个执行流，即在一个程序中可以同时运行多个不同的线程来执行不同的任务，也就是说允许单个程序创建多个并行执行的线程来完成各自的任务。

（1）多线程具有以下优点。

可以提高 CPU 的利用率。在多线程程序中，一个线程必须等待的时候，CPU 可以运行其他的线程而不是等待，这样就大大提高了程序的效率。

（2）多线程具有以下缺点。

① 线程也是程序，所以线程需要占用内存，线程越多占用内存也越多；
② 多线程需要协调和管理，所以需要 CPU 时间跟踪线程；
③ 线程之间对共享资源的访问会相互影响，必须解决竞用共享资源的问题；
④ 线程太多会导致控制太复杂，最终可能造成很多 Bug。

3. 线程的生命周期

线程生命周期开始于 System.Threading.Thread 类的对象被创建时，结束于线程被终止或完成执行时。下面列出了线程生命周期中的各种状态。

（1）未启动状态：当线程实例被创建，但 Start 方法未被调用时的状况。

（2）就绪状态：当线程准备好运行并等待 CPU 周期时的状况。

（3）不可运行状态：已经调用 Sleep 方法、Wait 方法或者通过 I/O 操作阻塞时，线程是不可运行的。

（4）死亡状态：当线程已完成执行或已中止时的状况。

在 C#中，Thread 类用于线程的工作。它允许创建并访问多线程应用程序中的单个线程。进程中第一个被执行的线程称为主线程。

【例 14-1】编写程序，获取当前的主线程，并为其命名。

（1）在 Visual Studio 2017 中，新建名称为“Project1”的文件。

（2）在代码编辑区域输入以下代码。

```
using System;
using System.Threading;
namespace Project1
{
    class Program
    {
        static void Main(string[] args)
        {
            Thread th = Thread.CurrentThread;
            th.Name = "MainThread";  //给当前线程命名为"MainThread"
            Console.WriteLine("This is {0}", th.Name);
        }
    }
}
```

【程序分析】在本例中，用户可以通过 Thread 类的静态属性 CurrentThread 获取当前执行的线程，然后通过 Name 属性赋值“MainThread”。

在 Visual Studio 2017 中的运行结果如图 14-2 所示。

图 14-2　主线程

14.1.3　Thread 类

在.NET Framework 中，所有与多线程机制应用相关的类都是放在 System.Threading 命名空间中的。如果用户想在应用程序中使用多线程，就必须包含这个类。

表 14-1 列出了 Thread 类的一些常用的属性。

表 14-1　Thread 类的属性及说明

属　　性	说　　明
CurrentContext	获取线程正在其中执行的当前上下文
CurrentCulture	获取或设置当前线程的区域性
CurrentPrinciple	获取或设置线程的当前负责人（对基于角色的安全性而言）
CurrentThread	获取当前正在运行的线程
CurrentUICulture	获取或设置资源管理器使用的当前区域性以便在运行时查找区域性特定的资源
ExecutionContext	获取一个 ExecutionContext 对象，该对象包含有关当前线程的各种上下文的信息
IsAlive	获取一个值，该值指示当前线程的执行状态
IsBackground	获取或设置一个值，该值指示某个线程是否为后台线程
IsThreadPoolThread	获取一个值，该值指示线程是否属于托管线程池
ManagedThreadId	获取当前托管线程的唯一标识符

续表

属　性	说　明
Name	获取或设置线程的名称
Priority	获取或设置一个值，该值指示线程的调度优先级
ThreadState	获取一个值，该值包含当前线程的状态

表 14-2 列出了 Thread 类的一些常用方法。

表 14-2　Thread 类的方法及说明

方　法	说　明
Abort	在调用此方法的线程上引发 ThreadAbortException，以开始终止此线程的过程。调用此方法通常会终止线程
GetApartmentState	返回表示单元状态的 ApartmentState 值
GetDomain	返回当前线程正在其中运行的当前域
GetDomainID	返回唯一的应用程序域标识符
Interrupt	中断处于 WaitSleepJoin 线程状态的线程
Join	在此实例表示的线程终止前，阻止调用线程
ResetAbort	取消当前线程所请求的 Abort（Object）
SetApartmentState	在线程启动前设置其单元状态
Sleep	将当前线程挂起指定的时间
SpinWait	导致线程等待由 iterations 参数定义的时间量
Start	使线程得以按计划执行
Suspend	挂起线程，或者如果线程已挂起，则不起作用
VolatileRead	读取字段值。无论处理器的数目或处理器缓存的状态如何，该值都是由计算机的任何处理器写入的最新值
VolatileWrite	立即向字段写入一个值，以使该值对计算机中的所有处理器都可见

14.2　线程的基本操作

通过使用 Thread 类，可以对线程进行创建、休眠、挂起、恢复、终止及设置优先权等操作。

14.2.1　创建线程

在 C#中创建线程时，首先需要创建一个 ThreadStart 委托实例，再以这个 ThreadStart 委托作为参数，来构造 Thread 实例。

注意：Thread 类拥有四种重载的构造函数，常用的一个函数接收一个 ThreadStart 类型的参数，而 ThreadStart 是一个委托，其语法格式如下：

```
public delegate void ThreadStart()
```

【例 14-2】编写程序，启动创建好的线程。

（1）在 Visual Studio 2017 中，新建名称为“Project2”的文件。

（2）在代码编辑区域输入以下代码。

```
using System;
using System.Threading;
namespace Project2
{
    class Program
    {
        //创建线程的方法,输出 0 到 10
        public static void ThreadMethod()
        {
            Console.WriteLine("辅助线程开始...");
            for (int i = 0; i < 10; i++)
            {
                Console.WriteLine("辅助线程: {0}", i);
                Thread.Sleep(2000);                          //调用 Sleep 方法，使辅助线程休眠 2 秒
            }
            Console.WriteLine("辅助线程结束.");
        }
        static void Main(string[] args)
        {
            Console.WriteLine("主线程开始");
            //创建委托实例
     ThreadStart ts = new ThreadStart(ThreadMethod);      //注册 ThreadMethod 方法
             //通过委托实例来构造 Thread 类
            Thread th = new Thread(ts);
            th.Start();                                      //启动线程
            for (char i = 'A'; i < 'K'; i++)
            {
                Console.WriteLine("主线程: {0}", i);
                Thread.Sleep(1000);                          //调用 Sleep 方法，使主线程休眠 1 秒
            }
            th.Join();                                       //主线程等待辅助线程结束
            Console.WriteLine("主线程结束");
        }
    }
}
```

【程序分析】本例演示了启动创建好的线程。在代码中，首先用户可以自定义一个静态的 void 方法 ThreadMethod；然后在 Main 方法中，创建 ThreadStart 委托的实例 ts；接着通过 ts 构造 Thread 类的实例 th，这样就创建好一个线程。由于 Main 方法是程序的入口点，优先执行 Main 方法，所以 Main 方法是主线程，ThreadMethod 方法为辅助线程；接着调用 Start 方法启动线程，此时主线程中的 for 循执行每一次后就会休眠 1 秒，而辅助线程每执行一次则休眠 2 秒；最后调用了 Join 方法，在辅助线程中的 for 循环执行完之后停止执行主线程中的 for 循环。

在 Visual Studio 2017 中的运行结果如图 14-3 所示。

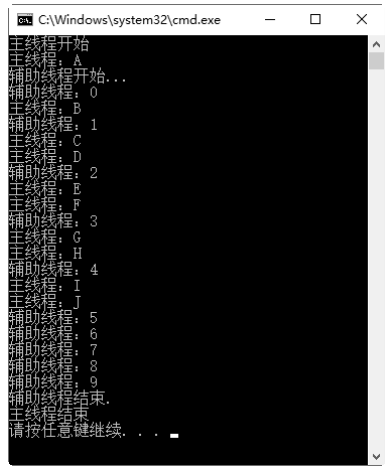

图 14-3　线程的创建

14.2.2　线程休眠

线程的休眠是通过 Thread 类的 Sleep 方法实现的，而 Thread 类的实例的 IsAlive 属性可以判断线程是

否执行完毕。

Sleep 方法有以下两种重载形式。

（1）将当前线程挂起指定的毫秒数，语法格式如下：

```
public static void Sleep (int millisecondsTimeout)
```

millisecondsTimeout：线程被阻止的毫秒数。如果该参数的值为零，则该线程会将其时间的剩余部分让给任何已经准备好运行的、具有同等优先级的线程，否则会无限期阻止线程。

（2）将当前线程挂起指定的时间，语法格式如下：

```
public static void Sleep (TimeSpan timeout)
```

timeout：线程被阻止的时间量的 TimeSpan。

【例 14-3】编写程序，创建线程，并在运行时休眠 5 秒。

（1）在 Visual Studio 2017 中，新建名称为“Project3”的文件。

（2）在代码编辑区域输入以下代码。

```
using System;
using System.Threading;
namespace Project3
{
    class Program
    {
        //下面的实例演示了 sleep() 方法的使用，用于在一个特定的时间暂停线程
        public static void Method()
        {
            Console.WriteLine("启动线程");
            //线程暂停 5000 毫秒
            int t = 5000;
            Console.WriteLine("线程暂停 {0} 秒",t / 1000);
            Thread.Sleep(t);
            Console.WriteLine("线程恢复");
        }
        static void Main(string[] args)
        {
            ThreadStart ts = new ThreadStart(Method);
            Thread th = new Thread(ts);   //创建线程
            th.Start();                   //启动线程
        }
    }
}
```

【程序分析】本例演示了线程的休眠。在代码中，首先用户可以自定义一个静态的 void 方法 Method；然后在 Main 方法中，创建 ThreadStart 委托的实例 ts；接着通过 ts 构造 Thread 类的实例 th，这样就创建好一个线程；最后通过 Start 方法，启动线程，由于在 Method 方法中调用了 Sleep 方法，使线程休眠 5 秒后再运行。

在 Visual Studio 2017 中的运行结果如图 14-4 所示。

图 14-4　线程的休眠

14.2.3　线程的挂起与恢复

Suspend 方法用于挂起线程，Resume 方法用于继续执行已经挂起的线程。可以使用这两个方法进行线程的同步，和 Start 方法有些类似的是，在调用 Suspend 方法后不会立即停止，而是执行到一个安全点后挂起。

1. Suspend 方法

挂起线程，或者如果线程已挂起，则不起作用，语法格式如下：

```
public void Suspend ();
```

2. Resume 方法

继续已挂起的线程，语法格式如下：

```
public void Resume ();
```

【例 14-4】编写程序，自定义两个线程，主线程（MainThread 方法）和工作线程（WorkThread 方法），主线程倒序输出，工作线程正序输出。先完成主线程，再完成工作线程。

（1）在 Visual Studio 2017 中，新建名称为“Project4”的文件。

（2）在代码编辑区域输入以下代码。

```
using System;
using System.Threading;
namespace Project4
{
    class Program
    {
        static void Main(string[] args)
        {
            ThreadStart work = new ThreadStart(WorkThread);     //创建工作线程
            Thread th = new Thread(work);
            th.Start();                                         //启动线程
            th.Suspend();                                       //挂起线程
            MainThread();                                       //主线程
            th.Resume();                                        //恢复线程
        }
        static void WorkThread()
        {
            for (long i = 1; i < 1000000000; i++)
            {
                if (i % 100000000 == 0 && i != 0)
                {
                    Console.WriteLine("工作线程 WorkThread-->i={0}", i);
                }
            }
        }
        static void MainThread()
        {
            long gap = 0;
            for (long i = 1000000000; i >= 0; i--)
            {
                gap = i - 1;
                if (i % 100000000 == 0 )
                {
                    Console.WriteLine("主线程 MainThread-->i={0}", i);
                }
            }
            Console.WriteLine();
        }
    }
}
```

【程序分析】本例演示了线程的挂起与恢复。在代码中，用户定义了两个方法 WorkThread 和 MainThread。MainThread 方法用于倒序输出 0～10；WorkThread 方法用于正序输出 1～9；然后在 Main 方法中，创建 ThreadStart 委托的实例 work；接着通过 work 构造 Thread 类的实例 th，这样就创建好一个线程；最后通过 Start 方法，启动线程。

挂起线程使用 Suspend 方法。线程被挂起后，操作被停止或进入休眠状态。因此，从结果中可以看出，此时主线程正常执行，但是工作线程 WorkThread 没有被执行。那么要想工作线程能继续执行，就需要使用

Resume 方法恢复线程。

在 Visual Studio 2017 中的运行结果如图 14-5 所示。

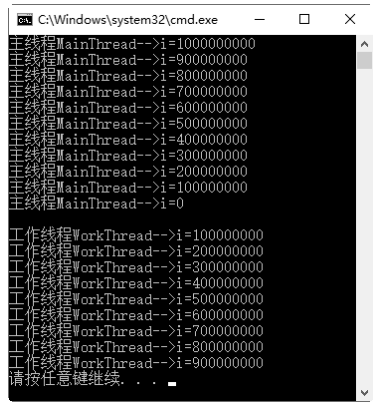

图 14-5　线程的挂起与恢复

14.2.4　终止线程

线程的终止是通过 Thread 类的 Abort 方法和 Join 方法来实现的。

1. Abort 方法

当一个线程执行时间太长时，用户有可能要终止这个线程，这就要使用 Abort 方法。该方法有两种重载方式：

```
public void Abort()//终止进程
public void Abort(Object stateInfo)
//终止线程并提供有关线程终止的异常信息
```

参数 stateInfo 是一个对象，包含应用程序特定的信息（如状态），该信息可供正被终止的线程使用

注意：在线程调用 Abort 方法时，会引发 ThreadAbortException 异常。如果没有捕获异常，线程将会终止通过。

【例 14-5】编写程序，启动线程，while 循环 5 次后终止线程。

（1）在 Visual Studio 2017 中，新建名称为“Project5”的文件。

（2）在代码编辑区域输入以下代码。

```
using System;
using System.Threading;
namespace Project5
{
    class Program
    {
        static void Main(string[] args)
        {
            ThreadStart work = new ThreadStart(WorkThread);      //创建工作线程
            Thread th = new Thread(work);
            th.Start();                                          //启动线程
            int i = 0;
            while (th.IsAlive)
            {
                i++;
                Thread.Sleep(500);                               //休眠 0.5 秒
                if (i == 5)
                {
                    th.Abort();                                  //线程被终止
                    Console.WriteLine("\r\n 线程被终止");
                }
            }
        }
        static void WorkThread()
        {
            for (long i = 0; i < 1000000000; i++)
            {
                if (i % 100000000 == 0 && i != 0)
                {
                    Console.WriteLine("工作线程 WorkThread-->i={0}", i);
                }
            }
        }
```

```
        }
    }
```

【程序分析】本例演示了线程的终止。在代码中，用户首先自定义方法 WorkThread，用于输出一组数据；然后在 Main 方法中，创建 ThreadStart 委托的实例 work；接着通过 work 构造 Thread 类的实例 th，这样就创建好一个线程；最后通过 Start 方法，启动工作线程。

中止线程使用 Abort 方法。线程被中止，就停止运行，是无法恢复的，因为 Windows 会永久地删除被中止线程的所有数据。跟挂起工作线程时的结果一样，中止工作线程后，工作线程自然不会被执行。

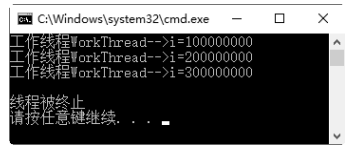

图 14-6　Abort 方法终止线程

在 Visual Studio 2017 中的运行结果如图 14-6 所示。

2. Join 方法

Join 方法用于等待线程中止，如果后续的处理依赖于另一个已经终止的线程，可以调用 Join 方法，等待线程中止。该方法有三种重载形式：

```
public void Join()
public bool Join(int millisecondsTimeout)
public bool Join(TimeSpan timeout)
```

参数说明：

millisecondsTimeout 表示等待线程终止的毫秒数。如果线程已终止，则返回值为 true，如果线程经过了 millisecondsTimeout 指定时间后未终止，返回值为 false。

timeout 表示等待线程终止的时间量 TimeSpan。如果线程已终止，则返回值为 true，如果线程经过 timeout 时间量之后未终止，则返回值为 false。

【例 14-6】编写程序，使用 Join 方法等待线程终止。

（1）在 Visual Studio 2017 中，新建名称为“Project6”的文件。

（2）在代码编辑区域输入以下代码。

```
using System;
using System.Threading;
namespace Project6
{
    class Program
    {
        static void Main(string[] args)
        {
            ThreadStart work = new ThreadStart(WorkThread);     //创建工作线程
            Thread th = new Thread(work);
            th.Start();                                         //启动线程
            th.Join();                                          //等待工作线程中止
         //th.Join(1000);
            MainThread();
        }
        static void WorkThread()
        {
            for (long i = 1; i < 1000000000; i++)
            {
                if (i % 100000000 == 0 && i != 0)
                {
                    Console.WriteLine("工作线程 WorkThread-->i={0}", i);
                }
            }
            Console.WriteLine("工作线程执行完毕");
        }
        static void MainThread()
```

```
        {
            long gap = 0;
            for (long i = 1000000000; i >= 0; i--)
            {
                gap = i - 1;
                if (i % 100000000 == 0)
                {
                    Console.WriteLine("主线程  MainThread-->i={0}", i);
                }
            }
            Console.WriteLine("主线程执行完毕");
        }
    }
}
```

【程序分析】本例演示了线程的等待终止。在 Main 方法中，工作线程调用了 Join 方法，因此，需待工作线程中止后，主线程才会被执行。

Join 的其他重载方法可以指定等待的时间期限，超过了这个时间期限，程序也会继续执行。因此，可以将“th.Join();”语句修改为“th.Join(1000);”。

在 Visual Studio 2017 中的运行结果如图 14-7 和图 14-8 所示。

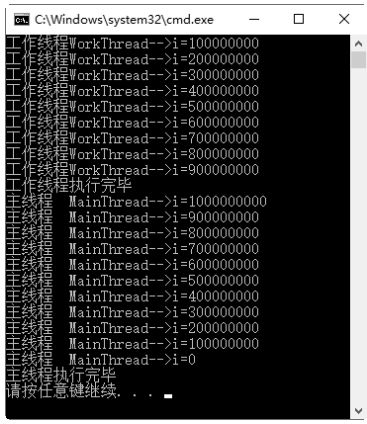

图 14-7　等待线程终止

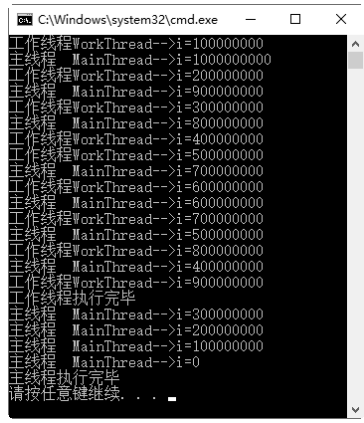

图 14-8　限时等待线程终止

14.2.5　线程的优先级

线程的优先级可以通过 Thread 类的 Priority 属性设置，Priority 属性是一个 ThreadPriority 型枚举，列举了 5 个优先等级：AboveNormal、BelowNormal、Highest、Lowest、Normal。

普通线程的优先级默认为 Normal；如果想有更高的优先级，可设置为 AboveNormal 或 Highest；如果想有较低的优先级，可设置为 BelowNormal 或 Lowest。线程优先级值，从高到低按顺序如表 14-3 所示。

表 14-3　线程的优先级值及说明

优 先 级 值	说　　明
Highest	在具有任何其他优先级的线程之前
AboveNormal	可以将 Thread 安排在具有 Highest 优先级线程之后，在 Normal 之前
Normal	在 AboveNormal 之后，BelowNormal 之前。默认值
BelowNormal	在 Normal 之后，Lowest 之前
Lowest	在具有其他任何优先级的线程之后

可以通过调用线程的 Priority 属性来获取和设置其优先级。Priority 属性用来获取或设置一个值，该值指示线程的调度优先级。

语法格式如下：

```
public ThreadPriority Priority{get;set;}
```

属性值是 ThreadPriority 枚举值之一，默认值为 Normal。

【例 14-7】编写程序，分别创建两个线程，然后通过设定不同的优先级来显示线程的名称和优先级。

（1）在 Visual Studio 2017 中，新建名称为“Project7”的文件。

（2）在代码编辑区域输入以下代码。

```
using System;
using System.Threading;
namespace Project7
{
    class Program
    {
        public static void method()
        {
            //输出线程的名称
            Console.WriteLine("线程名称: {0}",Thread.CurrentThread.Name.ToString());
            //输出线程的优先级
            Console.WriteLine("线程的优先级: {0}",Thread.CurrentThread.Priority.ToString());
        }
        static void Main(string[] args)
        {
            ThreadStart ts1 = new ThreadStart(method);
            ThreadStart ts2 = new ThreadStart(method);
            Thread t1 = new Thread(ts1);
            Thread t2 = new Thread(ts2);
            //为两个线程命名
            t1.Name = "学习 C#线程一";
            t2.Name = "学习 Java 线程二";
            //指定线程 t1 的优先级为 Highest
            t1.Priority = ThreadPriority.Highest;
            //启动两个线程
            t1.Start();
            t2.Start();
        }
    }
}
```

【程序分析】本例演示了线程的优先级。在代码中用户自定义方法 method，用于输出线程的名称和优先级；然后在 Main 方法中，创建两个线程 t1 和 t2，首先使用 Name 属性为两个线程命名，再使用 Priority 属性设置线程 t1 的优先级为 Highest，最后启动两个线程。通过最后的输出结果发现，t1 的优先级值为 Highest，t2 的优先级值为 Normal，这是因为 Normal 是线程的默认值。

在 Visual Studio 2017 中的运行结果如图 14-9 所示。

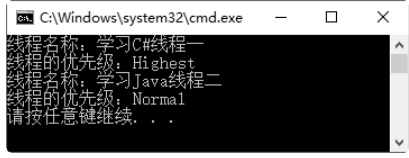

图 14-9　线程的优先级

14.3 就业面试技巧与解析

本章对 C#中进行线程编程的主要类 Thread 进行了介绍，并对进程的基本操作进行了详细讲解。通过本章的学习，读者应熟练掌握使用 C#进行线程编写的基础知识，并能在实际开发中应用线程解决各种多任务的问题。

14.3.1 面试技巧与解析（一）

面试官：使用多线程的优点？

应聘者：可以同时完成多个任务；可以使程序的响应速度更快；可以让占用大量处理时间的任务或当前没有进行处理的任务定期将处理时间让给别的任务；可以随时停止任务；可以设置每个任务的优先级以优化程序性能。

面试官：实现多线程的原因是什么？

应聘者：总结起来有以下两方面的原因：

（1）CPU 运行速度太快，硬件处理速度跟不上，所以操作系统进行分时间片管理。这样，从宏观角度来说是多线程并发的，因为 CPU 速度太快，察觉不到，看起来是同一时刻执行了不同的操作。但是从微观角度来讲，同一时刻只能有一个线程在处理。

（2）目前计算机都是多核多 CPU 的，一个 CPU 在同一时刻只能运行一个线程，但是多个 CPU 在同一时刻就可以运行多个线程。

14.3.2 面试技巧与解析（二）

面试官：多线程虽然有很多优点，但是也必须认识到多线程可能存在影响系统性能的不利方面，才能正确使用线程。简述一下多线程的缺点。

应聘者：使用多线程的缺点有以下几个方面：

（1）线程也是程序，所以线程需要占用内存，线程越多，占用内存也越多。

（2）多线程需要协调和管理，所以需要占用 CPU 时间以便跟踪线程。

（3）线程之间对共享资源的访问会相互影响，必须解决争用共享资源的问题。

（4）线程太多会导致控制太复杂，最终可能造成很多程序缺陷。

面试官：创建一个多线程都有哪些步骤？

应聘者：当启动一个可执行程序时，将创建一个主线程。在默认的情况下，C#程序具有一个线程，此线程执行程序中以 Main 方法开始和结束的代码，Main 方法直接或间接执行的每一个命令都有默认线程（主线程）执行，当 Main 方法返回时此线程也将终止。

创建多线程的步骤：

（1）编写线程所要执行的方法。

（2）实例化 Thread 类，并传入一个指向线程所要执行方法的委托。（这时线程已经产生，但还没有运行。）

（3）调用 Thread 实例的 Start 方法，标记该线程可以被 CPU 执行了，但具体执行时间由 CPU 决定。