华章程序员书库

[PACKT] PUBLISHING

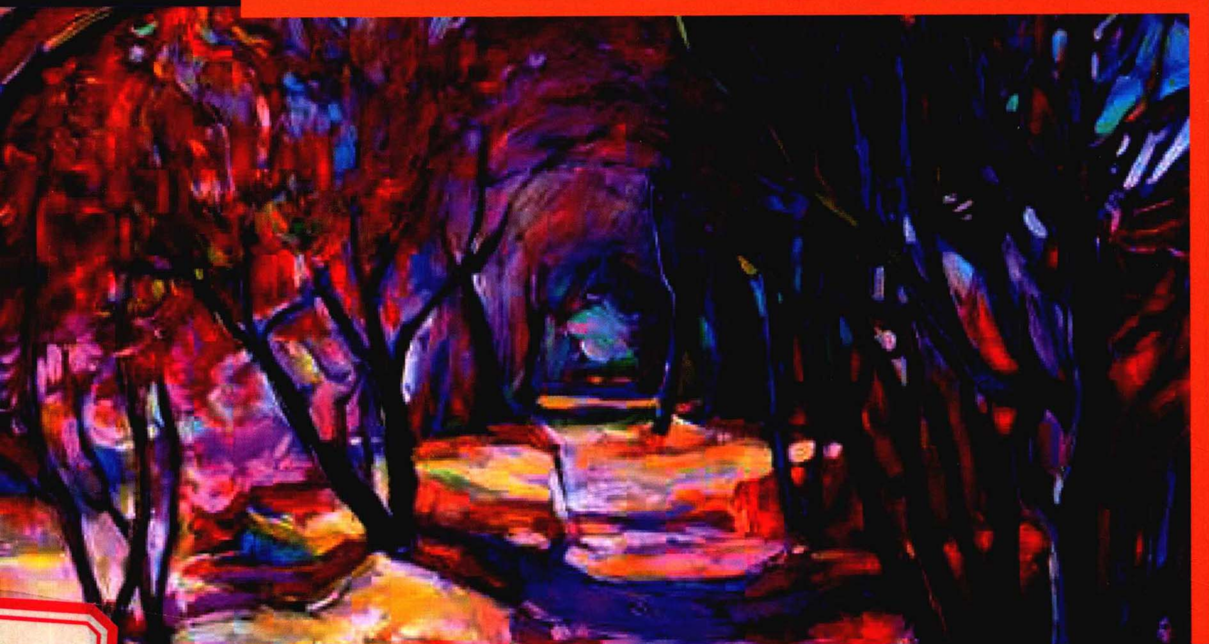
HZ BOOKS 华章IT

C#多线程编程实战

（原书第2版）

Multithreading with C# Cookbook

Second Edition



[美] 易格恩·阿格佛温（Eugene Agafonov） 著

黄博文 黄辉兰 译

机械工业出版社

China Machine Press

第1章 Chapter 1

线程基础

本章将涵盖C#中使用线程的基本操作。

你将学到以下内容：

- 使用C#创建线程
- 暂停线程
- 线程等待
- 终止线程
- 检测线程状态
- 线程优先级
- 前台线程和后台线程
- 向线程传参
- 使用C#中的lock关键字
- 使用Monitor类锁定资源
- 处理异常

1.1 简介

过去普通计算机只有一个计算单元，不能同时执行多个计算任务。然而操作系统却已经可以同时运行多个应用程序，即实现了多任务的概念。为了防止一个应用程序控制CPU而导致其他应用程序和操作系统本身永远被挂起这一可能情况，操作系统不得不使

用某种方式将物理计算单元分割为一些虚拟的进程，并给予每个执行程序一定量的计算能力。此外，操作系统必须始终能够优先访问 CPU，并能调整不同程序访问 CPU 的优先级。线程正是这一概念的实现。可以认为线程是一个虚拟进程，用于独立运行一个特定的程序。

请记住线程会消耗大量的操作系统资源。多个线程共享一个物理处理器将导致操作系统忙于管理这些线程，而无法运行程序。

因此，虽然有可能提高计算机的处理器能力，使得计算机每秒钟能执行越来越多的命令，但是使用线程通常是一个操作系统任务，试图在单核 CPU 上并行执行计算任务是没有意义的，因为这比顺序运行会花费更多的时间。然而，当处理器拥有多核时，过去的应用程序则不能使用这个优势，因为它们只使用了一个处理核心。

为了有效地利用现代处理器的计算能力，使用某种方式让程序能使用不止一个处理核心是非常重要的。这需要组织多个线程间的通信和相互同步。

本章中的内容将关注于使用 C# 语言执行一些非常基本的线程操作。我们将介绍线程的生命周期，其包括创建线程、挂起线程、线程等待，以及中止线程。然后我们会介绍一些基本的线程同步技术。

1.2 使用 C# 创建线程

在接下来的内容中，我们将使用 Visual Studio 2015 作为主要的工具来使用 C# 编写多线程程序。本节将展示如何创建一个新的 C# 程序，并在该程序中使用线程。

从微软官方网站可以下载免费的 Visual Studio 社区版 2015。我们需要它来运行示例代码。

1.2.1 准备工作

为了学习本节，你需要安装 Visual Studio 2015。除此之外无需其他准备。本节的源代码放置在 BookSamples\Chapter1\Recipe1 目录中。

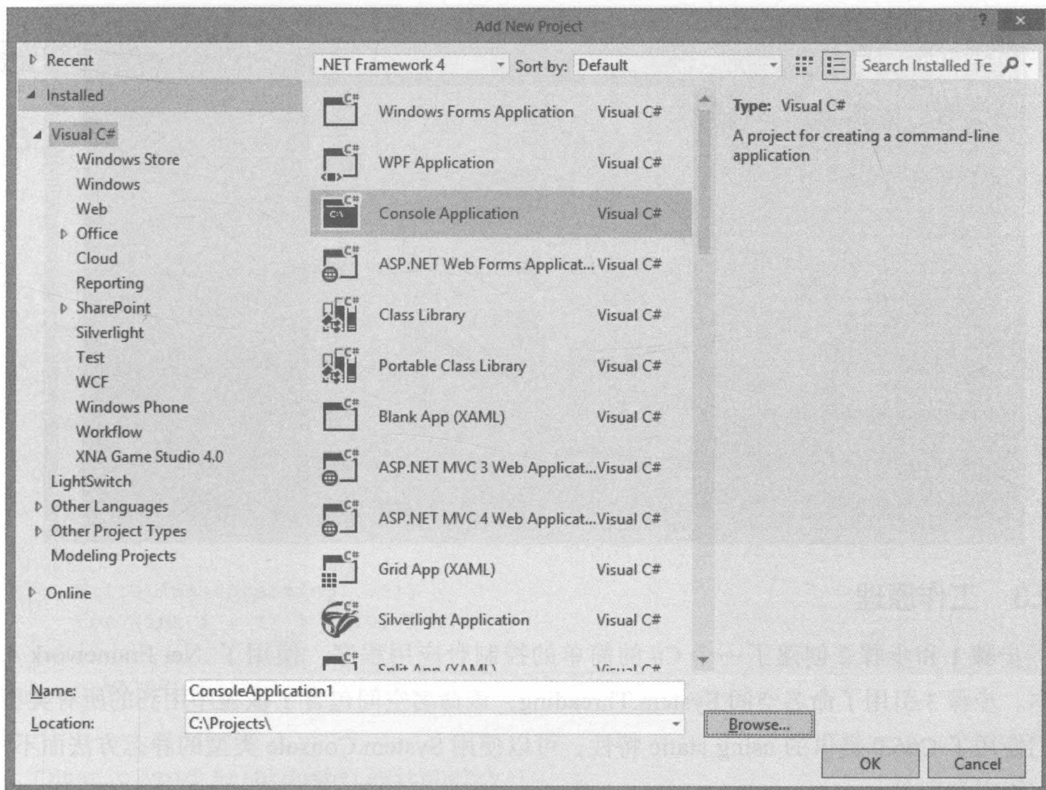
1.2.2 实现方式

请执行以下步骤来了解如何创建一个新的 C# 程序，并在其中使用线程：

1. 启动 Visual Studio 2015。新建一个 C# 控制台应用程序项目。

2. 确保该工程使用 .NET Framework 4.6 或以上版本。不过本节的代码在之前的版本中也

能正常运行。



3. 在 Program.cs 文件中加入以下 using 指令：

```
using System;
using System.Threading;
using static System.Console;
```

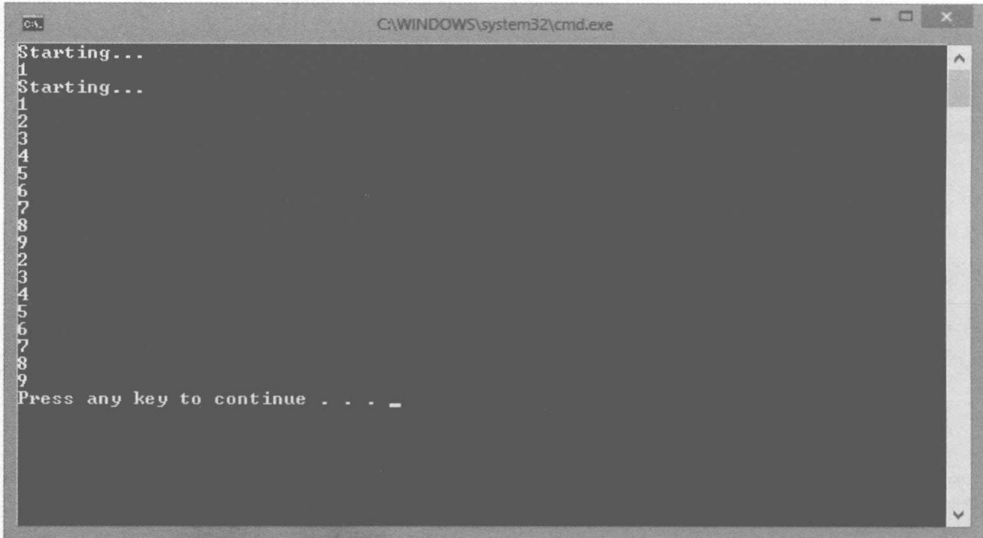
4. 在 Main 方法下面加入以下代码片段：

```
static void PrintNumbers()
{
  WriteLine("Starting...");
  for (int i = 1; i < 10; i++)
  {
    WriteLine(i);
  }
}
```

5. 在 Main 方法中加入以下代码片段：

```
Thread t = new Thread(PrintNumbers);
t.Start();
PrintNumbers();
```

6. 运行程序。输出如下所示：



1.2.3 工作原理

步骤 1 和步骤 2 创建了一个 C# 的简单的控制台应用程序，使用了 .Net Framework 4.0 版本。步骤 3 引用了命名空间 System.Threading，该命名空间包含了该程序用到的所有类型。我们使用了 C#6.0 提供的 using static 特性，可以使用 System.Console 类型的静态方法而不用我们指定类型名。

正在执行中的程序实例可被称为一个进程。进程由一个或多个线程组成。这意味着当运行程序时，始终有一个执行程序代码的主线程。

步骤 4 中定义了方法 PrintNumbers，该方法会被主程序和新创建的线程使用。在步骤 5 中创建了一个线程来运行 PrintNumbers 方法。当我们构造一个线程时，ThreadStart 或 ParameterizedThreadStart 的实例委托会传给构造函数。我们只需指定在不同线程运行的方法名，而 C# 编译器则会在后台创建这些对象。然后我们在主线程中以通常的方式启动了一个线程来运行 PrintNumbers 方法。

结果两组范围为 1 到 10 的数字会随机交叉输出。这说明 PrintNumbers 方法同时运行在主线程和另一个线程中。

1.3 暂停线程

本节将展示如何让一个线程等待一段时间而不用消耗操作系统资源。

试读结束：需要全本请在线购买： www.ertongbook.com

1.3.1 准备工作

为了学习本节，你需要安装 Visual Studio 2015。除此之外无需其他准备。本节的源代码放置在 BookSamples\Chapter1\Recipe2 目录中。

1.3.2 实现方式

请执行以下步骤来了解如何暂停线程而不消耗操作系统资源：

1. 启动 Visual Studio 2015。创建一个新的 C# 控制台应用程序项目。
2. 在 Program.cs 文件中加入以下 using 指令：

```
using System;
using System.Threading;
using static System.Console;
using static System.Threading.Thread;
```

3. 在 Main 方法下面加入以下代码片段：

```
static void PrintNumbers()
{
  WriteLine("Starting...");
  for (int i = 1; i < 10; i++)
  {
    WriteLine(i);
  }
}
static void PrintNumbersWithDelay()
{
  WriteLine("Starting...");
  for (int i = 1; i < 10; i++)
  {
    Sleep(TimeSpan.FromSeconds(2));
    WriteLine(i);
  }
}
```

4. 在 Main 方法中加入以下代码片段：

```
Thread t = new Thread(PrintNumbersWithDelay);
t.Start();
PrintNumbers();
```

5. 运行程序。

1.3.3 工作原理

当程序运行时，会创建一个线程，该线程会执行 PrintNumbersWithDelay 方法中的代码。

然后会立即执行 PrintNumbers 方法。关键之处在于在 PrintNumbersWithDelay 方法中加入了 Thread.Sleep 方法调用。这将导致线程执行该代码时，在打印任何数字之前会等待指定的时间（本例中是 2 秒钟）。当线程处于休眠状态时，它会占用尽可能少的 CPU 时间。结果我们会发现通常后运行的 PrintNumbers 方法中的代码会比独立线程中的 PrintNumbersWithDelay 方法中的代码先执行。

1.4 线程等待

本节将展示如何让程序等待另一个线程中的计算完成，然后在代码中使用该线程的计算结果。使用 Thread.Sleep 行不通，因为并不知道执行计算需要花费的具体时间。

1.4.1 准备工作

为了学习本节，你需要安装 Visual Studio 2015。除此之外无需其他准备。本节的源代码放置在 BookSamples\Chapter1\Recipe3 目录中。

1.4.2 实现方式

请执行以下步骤来了解如何让一个程序等待另一个线程中的计算完成，并随后使用该线程的计算结果：

1. 启动 Visual Studio 2015。创建一个新的 C# 控制台应用程序项目。
2. 在 Program.cs 文件中加入以下 using 指令：

```
using System;
using System.Threading;
using static System.Console;
using static System.Threading.Thread;
```

3. 在 Main 方法下面加入以下代码片段：

```
static void PrintNumbersWithDelay()
{
  WriteLine("Starting...");
  for (int i = 1; i < 10; i++)
  {
    Sleep(TimeSpan.FromSeconds(2));
    WriteLine(i);
  }
}
```

4. 在 Main 方法中加入以下代码片段：

```
WriteLine("Starting...");
Thread t = new Thread(PrintNumbersWithDelay);
t.Start();
t.Join();
WriteLine("Thread completed");
```

5. 运行程序。

1.4.3 工作原理

当程序运行时，启动了一个耗时较长的线程来打印数字，打印每个数字前要等待两秒。但我们在主程序中调用了 t.Join 方法，该方法允许我们等待直到线程 t 完成。当线程 t 完成时，主程序会继续运行。借助该技术可以实现在两个线程间同步执行步骤。第一个线程会等待另一个线程完成后再继续执行。第一个线程等待时是处于阻塞状态（正如 1.3 节中调用 Thread.Sleep 方法一样）。

1.5 终止线程

本节将讲述如何终止线程的执行。

1.5.1 准备工作

为了学习本节，你需要安装 Visual Studio 2015。除此之外无需其他准备。本节的源代码放置在 BookSamples\Chapter1\Recipe4 目录中。

1.5.2 实现方式

请执行以下步骤来了解如何终止线程的执行：

1. 启动 Visual Studio 2015。创建一个新的 C# 控制台应用程序项目。

2. 在 Program.cs 文件中加入以下 using 指令：

```
using System;
using System.Threading;
using static System.Console;
```

3. 在 Main 方法下面加入以下代码片段：

```
static void PrintNumbersWithDelay()
{
  WriteLine("Starting...");
  for (int i = 1; i < 10; i++)
  {
```

```
        Sleep(TimeSpan.FromSeconds(2));
        WriteLine(i);
    }
}
```

4. 在 Main 方法中加入以下代码片段：

```
WriteLine("Starting program...");
Thread t = new Thread(PrintNumbersWithDelay);
t.Start();
Thread.Sleep(TimeSpan.FromSeconds(6));
t.Abort();
WriteLine("A thread has been aborted");
Thread t = new Thread(PrintNumbers);
t.Start();
PrintNumbers();
```

5. 运行程序。

1.5.3 工作原理

当主程序和单独的数字打印线程运行时，我们等待 6 秒后对线程调用了 t.Abort 方法。这给线程注入了 ThreadAbortException 方法，导致线程被终结。这非常危险，因为该异常可以在任何时刻发生并可能彻底摧毁应用程序。另外，使用该技术也不一定总能终止线程。目标线程可以通过处理该异常并调用 Thread.ResetAbort 方法来拒绝被终止。因此并不推荐使用 Abort 方法来关闭线程。可优先使用一些其他方法，比如提供一个 CancellationToken 方法来取消线程的执行。在第 3 章中我们会讨论该方法。

1.6 检测线程状态

本节将描述一个线程可能会有哪些状态。获取线程是否已经启动或是否处于阻塞状态等相应信息是非常有用的。请注意由于线程是独立运行的，所以其状态可以在任何时候被改变。

1.6.1 准备工作

为了学习本节，你需要安装 Visual Studio 2015。除此之外无需其他准备。本节的源代码放置在 BookSamples\Chapter1\Recipe5 目录中。

1.6.2 实现方式

请执行以下步骤来了解如何确定线程状态及获取线程相关的信息。

1. 启动 Visual Studio 2015。创建一个新的 C# 控制台应用程序项目。
2. 在 Program.cs 文件中加入以下 using 指令：

```
using System;
using System.Threading;
using static System.Console;
using static System.Threading.Thread;
```

3. 在 Main 方法下面加入以下代码片段：

```
static void DoNothing()
{
  Sleep(TimeSpan.FromSeconds(2));
}

static void PrintNumbersWithStatus()
{
  WriteLine("Starting...");
  WriteLine(CurrentThread.ThreadState.ToString());
  for (int i = 1; i < 10; i++)
  {
    Sleep(TimeSpan.FromSeconds(2));
    WriteLine(i);
  }
}
```

4. 在 Main 方法中加入以下代码片段：

```
WriteLine("Starting program...");
Thread t = new Thread(PrintNumbersWithStatus);
Thread t2 = new Thread(DoNothing);
WriteLine(t.ThreadState.ToString());
t2.Start();
t.Start();
for (int i = 1; i < 30; i++)
{
  WriteLine(t.ThreadState.ToString());
}
Sleep(TimeSpan.FromSeconds(6));
t.Abort();
WriteLine("A thread has been aborted");
WriteLine(t.ThreadState.ToString());
WriteLine(t2.ThreadState.ToString());
```

5. 运行程序。

1.6.3 工作原理

当主程序启动时定义了两个不同的线程。一个将被终止，另一个则会成功完成运行。线

程状态位于 Thread 对象的 ThreadState 属性中。ThreadState 属性是一个 C# 枚举对象。刚开始线程状态为 ThreadState.Unstarted。然后我们启动线程，并估计在一个周期为 30 次迭代的区间中，线程状态会从 ThreadState.Running 变为 ThreadState.WaitSleepJoin。

请注意始终可以通过 Thread.CurrentThread 静态属性获得当前 Thread 对象。

如果实际情况与以上不符，请增加迭代次数。终止第一个线程后，会看到现在该线程状态为 ThreadState.Aborted。程序也有可能会打印出 ThreadState.AbortRequested 状态。这充分说明了同步两个线程的复杂性。请记住不要在程序中使用线程终止。我在这里使用它只是为了展示相应的线程状态。

最后可以看到第二个线程 t2 成功完成并且状态为 ThreadState.Stopped。另外还有一些其他的线程状态，但是要么已经被弃用，要么没有我们实验过的几种状态有用。

1.7 线程优先级

本节将描述线程优先级的几种不同的可能选项。线程优先级决定了该线程可占用多少 CPU 时间。

1.7.1 准备工作

为了学习本节，你需要安装 Visual Studio 2015。除此之外无需其他准备。本节的源代码放置在 BookSamples\Chapter1\Recipe6 目录中。

1.7.2 实现方式

请执行以下步骤来了解线程优先级的工作方式：

1. 启动 Visual Studio 2015。新建一个 C# 控制台应用程序项目。

2. 在 Program.cs 文件中加入以下 using 指令：

```
using System;
using System.Threading;
using static System.Console;
using static System.Threading.Thread;
using static System.Diagnostics.Process;
```

3. 在 Main 方法下面加入以下代码片段：

```
static void RunThreads()
{
```

试读结束：需要全本请在线购买： www.ertongbook.com

```
  var sample = new ThreadSample();

  var threadOne = new Thread(sample.CountNumbers);
  threadOne.Name = "ThreadOne";
  var threadTwo = new Thread(sample.CountNumbers);
  threadTwo.Name = "ThreadTwo";

  threadOne.Priority = ThreadPriority.Highest;
  threadTwo.Priority = ThreadPriority.Lowest;
  threadOne.Start();
  threadTwo.Start();

  Sleep(TimeSpan.FromSeconds(2));
  sample.Stop();
}

class ThreadSample
{
  private bool _isStopped = false;

  public void Stop()
  {
    _isStopped = true;
  }

  public void CountNumbers()
  {
    long counter = 0;

    while (!_isStopped)
    {
      counter++;
    }

    WriteLine($"{CurrentThread.Name} with " +
      $"{CurrentThread.Priority,11} priority " +
      $"has a count = {counter,13:N0}");
  }
}
```

4. 在 Main 方法中加入以下代码片段：

```
WriteLine($"Current thread priority: {CurrentThread.Priority}");
WriteLine("Running on all cores available");
RunThreads();
Sleep(TimeSpan.FromSeconds(2));
WriteLine("Running on a single core");
GetCurrentProcess().ProcessorAffinity = new IntPtr(1);
RunThreads();
```

5. 运行程序。

1.7.3 工作原理

当主程序启动时定义了两个不同的线程。第一个线程优先级为 ThreadPriority.Highest，即具有最高优先级。第二个线程优先级为 ThreadPriority.Lowest，即具有最低优先级。我们先打印出主线程的优先级值，然后在所有可用的 CPU 核心上启动这两个线程。如果拥有一个以上的计算核心，将在两秒钟内得到初步结果。最高优先级的线程通常会计算更多的迭代，但是两个值应该很接近。然而，如果有其他程序占用了所有的 CPU 核心运行负载，结果则会截然不同。

为了模拟该情形，我们设置了 ProcessorAffinity 选项，让操作系统将所有的线程运行在单个 CPU 核心（第一个核心）上。现在结果完全不同，并且计算耗时将超过 2 秒钟。这是因为 CPU 核心大部分时间在运行高优先级的线程，只留给剩下的线程很少的时间来运行。

请注意这是操作系统使用线程优先级的一个演示。通常你无需使用这种行为编写程序。

1.8 前台线程和后台线程

本节将描述前台线程和后台线程，及如何设置该选项来影响程序的行为。

1.8.1 准备工作

为了学习本节，你需要安装 Visual Studio 2015。除此之外无需其他准备。本节的源代码放置在 BookSamples\Chapter1\Recipe7 目录中。

1.8.2 实现方式

请执行以下步骤来了解程序中前台线程和后台线程的效果：

1. 启动 Visual Studio 2015。新建一个 C# 控制台应用程序项目。
2. 在 Program.cs 文件中加入以下 using 指令：

```
using System;
using System.Threading;
using static System.Console;
using static System.Threading.Thread;
```

3. 在 Main 方法下面加入以下代码片段：

```
class ThreadSample
{
```

```
  private readonly int _iterations;

  public ThreadSample(int iterations)
  {
    _iterations = iterations;
  }
  public void CountNumbers()
  {
    for (int i = 0; i < _iterations; i++)
    {
      Sleep(TimeSpan.FromSeconds(0.5));
      WriteLine($"{CurrentThread.Name} prints {i}");
    }
  }
}
```

4. 在 Main 方法中加入以下代码片段：

```
var sampleForeground = new ThreadSample(10);
var sampleBackground = new ThreadSample(20);

var threadOne = new Thread(sampleForeground.CountNumbers);
threadOne.Name = "ForegroundThread";
var threadTwo = new Thread(sampleBackground.CountNumbers);
threadTwo.Name = "BackgroundThread";
threadTwo.IsBackground = true;

threadOne.Start();
threadTwo.Start();
```

5. 运行程序。

1.8.3 工作原理

当主程序启动时定义了两个不同的线程。默认情况下，显式创建的线程是前台线程。通过手动的设置 threadTwo 对象的 IsBackground 属性为 true 来创建一个后台线程。通过配置来实现第一个线程会比第二个线程先完成。然后运行程序。

第一个线程完成后，程序结束并且后台线程被终结。这是前台线程与后台线程的主要区别：进程会等待所有的前台线程完成后再结束工作，但是如果只剩下后台线程，则会直接结束工作。

一个重要注意事项是如果程序定义了一个不会完成的前台线程，主程序并不会正常结束。

1.9 向线程传递参数

本节将描述如何提供一段代码来使用要求的数据运行另一个线程。我们将介绍不同的方式来满足此任务，并且回顾常见的错误。

1.9.1 准备工作

为了学习本节，你需要安装 Visual Studio 2015。除此之外无需其他准备。本节的源代码放置在 BookSamples\Chapter1\Recipe8 目录中。

1.9.2 实现方式

请执行以下步骤来了解如何给一个线程传递参数：

1. 启动 Visual Studio 2015。新建一个 C# 控制台应用程序项目。
2. 在 Program.cs 文件中加入以下 using 指令：

```
using System;
using System.Threading;
using static System.Console;
using static System.Threading.Thread;
```

3. 在 Main 方法下面加入以下代码片段：

```
static void Count(object iterations)
{
  CountNumbers((int)iterations);
}

static void CountNumbers(int iterations)
{
  for (int i = 1; i <= iterations; i++)
  {
    Sleep(TimeSpan.FromSeconds(0.5));
    WriteLine($"{CurrentThread.Name} prints {i}");
  }

}

static void PrintNumber(int number)
{
  WriteLine(number);
}

class ThreadSample
{
```

```
  private readonly int _iterations;

  public ThreadSample(int iterations)
  {
    _iterations = iterations;
  }
  public void CountNumbers()
  {
    for (int i = 1; i <= _iterations; i++)
    {
      Sleep(TimeSpan.FromSeconds(0.5));
         WriteLine($"{CurrentThread.Name} prints {i}");
    }
  }
}
```

4. 在Main方法中加入以下代码片段:

```
var sample = new ThreadSample(10);

var threadOne = new Thread(sample.CountNumbers);
threadOne.Name = "ThreadOne";
threadOne.Start();
threadOne.Join();

WriteLine("--------------------------");

var threadTwo = new Thread(Count);
threadTwo.Name = "ThreadTwo";
threadTwo.Start(8);
threadTwo.Join();

WriteLine("--------------------------");

var threadThree = new Thread(() => CountNumbers(12));
threadThree.Name = "ThreadThree";
threadThree.Start();
threadThree.Join();
WriteLine("--------------------------");

int i = 10;
var threadFour = new Thread(() => PrintNumber(i));
i = 20;
var threadFive = new Thread(() => PrintNumber(i));
threadFour.Start();
threadFive.Start();
```

5. 运行程序。

1.9.3 工作原理

当主程序启动时，首先创建了 ThreadSample 类的一个对象，并提供了一个迭代次数。然后使用该对象的 CountNumbers 方法启动线程。该方法运行在另一个线程中，但是使用数字 10，该数字是通过 ThreadSample 对象的构造函数传入的。因此，我们只是使用相同的间接方式将该迭代次数传递给另一个线程。

1.9.4 更多信息

另一种传递数据的方式是使用 Thread.Start 方法。该方法会接收一个对象，并将该对象传递给线程。为了应用该方法，在线程中启动的方法必须接受 object 类型的单个参数。在创建 threadTwo 线程时演示了该方式。我们将 8 作为一个对象传递给了 Count 方法，然后 Count 方法被转换为整型。

接下来的方式是使用 lambda 表达式。lambda 表达式定义了一个不属于任何类的方法。我们创建了一个方法，该方法使用需要的参数调用了另一个方法，并在另一个线程中运行该方法。当启动 threadThree 线程时，打印出了 12 个数字，这正是我们通过 lambda 表达式传递的数字。

使用 lambda 表达式引用另一个 C# 对象的方式被称为闭包。当在 lambda 表达式中使用任何局部变量时，C# 会生成一个类，并将该变量作为该类的一个属性。所以实际上该方式与 threadOne 线程中使用的一样，但是我们无须定义该类，C# 编译器会自动帮我们实现。

这可能会导致几个问题。例如，如果在多个 lambda 表达式中使用相同的变量，它们会共享该变量值。在前一个例子中演示了这种情况。当启动 threadFour 和 threadFive 线程时，它们都会打印 20，因为在这两个线程启动之前变量被修改为 20。

1.10 使用 C# 中的 lock 关键字

本节将描述如何确保当一个线程使用某些资源时，同时其他线程无法使用该资源。我们将了解该情况的必要性及整个线程安全概念都包含什么。

1.10.1 准备工作

为了学习本节，你需要安装 Visual Studio 2015。除此之外无需其他准备。本节的源代码放置在 BookSamples\Chapter1\Recipe9 目录中。

1.10.2 实现方式

请执行以下步骤来了解如何使用 C# 中的 lock 关键字：

试读结束：需要全本请在线购买： www.ertongbook.com