# 第 5 章 进程、线程与同步

接下来我们将讨论Windows NT/2000/XP操作系统架构下有关进程与线程的基本概念。必须牢记，上一章所介绍的CLR是在启动.NET程序集时，被运行时宿主载入进程的一个软件层。

## 5.1 简介

进程（process）是一块包含了某些资源的内存区域。操作系统利用进程把它的工作划分为一些功能单元。

进程中所包含的一个或多个执行单元称为线程（thread）。进程还拥有一个私有的虚拟地址空间，该空间仅能被它所包含的线程访问。

当运行.NET程序时，进程还会把被称为CLR的软件层包含到它的内存空间中。上一章曾经对CLR做了详细描述。该软件层是在进程创建期间由运行时宿主载入的（参见4.2.3节）。

线程只能归属于一个进程并且它只能访问该进程所拥有的资源。当操作系统创建一个进程后，该进程会自动申请一个名为主线程或首要线程的线程。主线程将执行运行时宿主，而运行时宿主会负责载入CLR。

应用程序（application）是由一个或多个相互协作的进程组成的。例如，Visual Studio开发环境就是利用一个进程编辑源文件，并利用另一个进程完成编译工作的应用程序。

在Windows NT/2000/XP操作系统下，我们可以通过任务管理器在任意时间查看所有的应用程序和进程。尽管只打开了几个应用程序，但是通常情况下将有大约30个进程同时运行。 事实上，为了管理当前的会话和任务栏以及其他一些任务，系统执行了大量的进程。

## 5.2 进程

### 5.2.1 简介

在运行于32位处理器上的32位Windows操作系统中，可将一个进程视为一段大小为4GB（$2^{32}$字节）的线性内存空间，它起始于0x00000000结束于0xFFFFFFFF。这段内存空间不能被其他进程所访问，所以称为该进程的私有空间。这段空间被平分为两块，2GB被系统所有，剩下2GB被用户所有。

如果有N个进程运行在同一台机器上，那么将需要N×4GB的海量RAM，还好事实并非如此。

- Windows是按需为每个进程分配内存的，4GB是32位系统中一个进程所占空间的上限。
- 将进程所需的内存划分为4KB大小的内存页，并根据使用情况将这些内存页存储在硬盘上或加载到RAM中，通过系统的这种虚拟内存机制，我们可以有效地减少对实际内存的需求量。当然这些对用户和开发者来说都是透明的。

### 5.2.2　System.Diagnostics.Process 类

System.Diagnostics.Process类的实例可以引用一个进程，被引用的进程包含以下几种。

- 该实例的当前进程。
- 本机上除了当前进程的其他进程。
- 远程机器上的某个进程。

通过该类所包含的方法和字段，可以创建或销毁一个进程，并且可以获得一个进程的相关信息。下面将讨论一些使用该类实现的常见任务。

### 5.2.3　创建和销毁子进程

下面的程序创建了一个称为*子进程*的新进程。在这种情况下，初始的进程称为*父进程*。子进程启动了一个记事本应用程序。父进程的线程在等待1秒后销毁该子进程。该程序的执行效果就是打开并关闭记事本。

例5-1

```
using System.Diagnostics;
using System.Threading;
class Program {
   static void Main() {
      // Create a child process which executes the 'notepad.exe'
      // application on the text file 'hello.txt'.
      Process process = Process.Start("notepad.exe", "hello.txt");
      // Sleep one second.
      Thread.Sleep(1000);
      // Destroy the child process.
      process.Kill();
   }
}
```

静态方法Start()可以使用已存在的Windows文件扩展名关联机制。例如，我们可以利用下面的代码执行同样的操作。

```
Process process = Process.Start("hello.txt");
```

默认情况下，子进程将继承其父进程的安全上下文。但还可以使用Process.Start()方法的一个重载版本在任意用户的安全上下文中启动该子进程，当然需要通过一个System.Diagnostics.ProcessStartInfo类的实例来提供该用户的用户名和密码。

### 5.2.4　避免在一台机器上同时运行同一应用程序的多个实例

有些应用程序需要这种功能。实际上，通常来说在同一台机器上同时运行一个应用程序的多个实例并没有意义。

直到现在，为了在Windows下满足上述约束，开发者最常用的方法仍然是使用*有名互斥体*（named mutex）技术（参见5.7.2节）。然而采用这种技术来满足上述约束存在以下缺点：

- 该技术具有使互斥体的名字被其他应用程序所使用的较小的、潜在的风险。在这种情况下该技术将不再有效并且会造成很难检测到的bug。
- 该技术不能解决我们仅允许一个应用程序产生N个实例这种一般的问题。

幸而在System.Diagnostics.Process类中拥有GetCurrentProcess()（返回当前进程）和GetProcesses()（返回机器上所有的进程）这样的静态方法。在下面的程序中我们为上述问题找到了一个优雅且简单的解决方案。

例5-2

```
using System.Diagnostics;
class Program {
   static void Main() {
      if ( TestIfAlreadyRunning() ) {
         System.Console.WriteLine("This app is already running!");
      }
      else{
         // Here, the entry point of the application.
      }
   }
   static bool TestIfAlreadyRunning() {
      Process processCurrent = Process.GetCurrentProcess();
      Process[] processes = Process.GetProcesses();
      foreach ( Process process in processes )
         if ( processCurrent.Id != process.Id )
            if ( processCurrent.ProcessName == process.ProcessName )
               return true;
      return false;
   }
}
```

通过方法参数指定了远程机器的名字后，GetProcesses()方法也可以返回远程机器上所有的进程。

### 5.2.5　终止当前进程

可以调用System.Environment类中的静态方法Exit(int exitCode)或FailFast(string message)终止当前进程。Exit()方法是最好的选择，它将彻底终止进程并向操作系统返回指定的退出代码值。之所以称为彻底终止是因为当前对象的所有清理工作以及finally块的执行都将由不同的线程完成。当然，终止进程将花费一定的时间。

顾名思义，FailFast()方法可以迅速终止进程。Exit()方法所做的预防措施将被它忽略。只有一个包含了指定信息的严重错误会被操作系统记录到日志中。你可能想要在探查问题的时候使用该方法，因为可以将该程序的彻底终止视为数据恶化的起因。

## 5.3　线程

### 5.3.1　简介

一个线程包含以下内容。

- 一个指向当前被执行指令的指令指针；
- 一个栈；
- 一个寄存器值的集合，定义了一部分描述正在执行线程的处理器状态的值；
- 一个私有的数据区。

所有这些元素都归于*线程执行上下文*的名下。处在同一个进程中的所有线程都可以访问该进程所包含的地址空间，当然也包含存储在该空间中的所有资源。

我们不准备讨论线程在*内核模式*或者*用户模式*执行的问题。尽管.NET以前的Windows一直使用这两种模式，并且依然存在，但是对.NET Framework来说它们是不可见的。

并行使用一些线程通常是我们在实现算法时的自然反应。实际上，一个算法往往由一系列可以并发执行的任务组成。但是需要引起注意的是，使用大量的线程将引起过多的*上下文切换*，最终反而影响了性能。

同样，几年前我们就注意到，预测每18个月处理器运算速度增加一倍的摩尔定律已不再成立。处

理器的频率停滞在3GHz~4GHz上下。这是由于物理上的限制，需要一段时间才能取得突破。同时，为了在性能竞争中不会落败，较大的处理器制造商如AMD和Intel目前都将目标转向多核芯片。因此我们可以预计在接下去的几年中这种类型的架构将广泛被采用。在这种情况下，改进应用性能的唯一方案就是合理地利用多线程技术。

### 5.3.2　受托管的线程与 Windows 线程

必须要了解，执行.NET应用的线程实际上仍然是Windows线程。但是，当某个线程被CLR所知时，我们将它称为受托管的线程。具体来说，由受托管的代码创建出来的线程就是受托管的线程。如果一个线程由非托管的代码所创建，那么它就是非托管的线程。不过，一旦该线程执行了受托管的代码它就变成了受托管的线程。

一个受托管的线程和非托管的线程的区别在于，CLR将创建一个`System.Threading.Thread`类的实例来代表并操作前者。在内部实现中，CLR将一个包含了所有受托管线程的列表保存在一个叫做ThreadStore地方。

CLR确保每一个受托管的线程在任意时刻都在一个AppDomain中执行，但是这并不代表一个线程将永远处在一个AppDomain中，它可以随着时间的推移转到其他的AppDomain中。关于AppDomain的概念参见4.1。

从安全的角度来看，一个受托管的线程的主用户与底层的非托管线程中的Windows主用户是无关的。

### 5.3.3　抢占式多任务处理

我们可以问自己下面这个问题：我的计算机只有一个处理器，然而在任务管理器中我们却可以看到数以百计的线程正同时运行在机器上！这怎么可能呢？

多亏了*抢占式多任务处理*，通过它对线程的调度，使得上述问题成为可能。*调度器*作为Windows内核的一部分，将时间切片，分成一段段的*时间片*。这些时间间隔以毫秒为精度且长度并不固定。针对每个处理器，每个时间片仅服务于单独一个线程。线程的迅速执行给我们造成了它们在同时运行的假象。我们在两个时间片的间隔中进行*上下文切换*。该方法的优点在于，那些正在等待某些Windows资源的线程将不会浪费时间片，直到资源有效为止。

之所以用抢占式这个形容词来修饰这种多任务管理方式，是因为在此种方式下线程将被系统强制性中断。那些对此比较好奇的人应该了解到，在上下文切换的过程中，操作系统会在下一个线程将要执行的代码中插入一条跳转到下一个上下文切换的指令。该指令是一个*软中断*，如果线程在遇到这条指令前就终止了（例如，它正在等待某个资源），那么该指定将被删除而上下文切换也将提前发生。

抢占式多任务处理的主要缺点在于，必须使用一种同步机制来保护资源以避免它们被无序访问。除此之外，还有另一种多任务管理模型，被称为*协调式多任务管理*，其中线程间的切换将由线程自己负责完成。该模型普遍认为太过危险，原因在于线程间的切换不发生的风险太大。如我们在4.2.8节中所解释的那样，该机制会在内部使用以提升某些服务器的性能，例如SQL Server2005。但Windows操作系统仅仅实现了抢占式多任务处理。

### 5.3.4　进程与线程的优先级

某些任务拥有比其他任务更高的优先级，它们需要操作系统为它们申请更多的处理时间。例如，某些由主处理器负责的外围驱动器必须不能被中断。另一类高优先级的任务就是图形用户界面。事实上，用户不喜欢等待用户界面被重绘。

那些从Win32世界来的用户知道在CLR的底层，也就是Windows操作系统中，可以为每个线程赋予一个0~31的优先级。但你无法在.NET的世界中也使用这些数值，因为：

- 它们无法描述自身的含义。
- 随着时间的流逝这些值是非常容易变化的。

1. 进程的优先级

可以使用Process类中的类型为ProcessPriorityClass的PriorityClass{get;set;}属性为进程赋予一个优先级。System.Diagnostics.ProcessPriorityClass枚举包含以下值：

| ProcessPriorityClass的值 | 对应的优先级 |
|---|---|
| Low | 4 |
| BelowNormal | 6 |
| Normal | 8 |
| AboveNormal | 10 |
| High | 13 |
| RealTime | 24 |

如果某个进程中属于Process类的PriorityBoostEnabled属性的值为true（默认值为true），那么当该进程占据前台窗口的时候，它的优先级将增加一个单位。只有当Process类的实例引用的是本机进程时，才能够访问该属性。

可以通过以下操作利用任务管理器来改变一个进程的优先级：在所选的进程上点击右键>设置优先级>从提供的6个值（和上图所述一致）中做出选择。

Windows操作系统有一个优先级为0的空闲进程。该进程不能被其他任何进程使用。根据定义，进程的活跃度用时间的百分比表示为：100%减去在空闲进程中所耗费时间的比率。

2. 线程的优先级

每个线程可以结合它所属进程的优先级，并使用System.Threading.Thread类中类型为ThreadPriority的Priority{get;set;}属性定义各自的优先级。System.Threading.ThreadPriority包含以下枚举值：

| ThreadPriority的值 | 对线程优先级的影响 |
|---|---|
| Lowest | –2 units from the priority of the process |
| BelowNormal | –1 unit from the priority of the process |
| Normal | Same priority as the process |
| AboveNormal | +1 unit from the priority of the process |
| Highest | +2 units from the priority of the process |

在大多数应用程序中，不需要修改进程和线程的优先级，它们的默认值为Normal。

### 5.3.5　System.Threading.Thread 类

CLR会自动将一个System.Threading.Thread类的实例与各个受托管的线程关联起来。可以使用该对象从线程自身或从其他线程来操纵线程。还可以通过System.Threading.Thread类的静态属性CurrentThread来获得当前线程的对象。

```
using System.Threading;
...
Thread threadCurrent = Thread.CurrentThread;
```

Thread类有一个功能使我们能够很方便的调试多线程应用程序，该功能允许我们使用一个字符串为线程命名：

```
threadCurrent.Name = “thread Foo”;
```

### 5.3.6　创建与 Join 一个线程

只需通过创建一个Thread类的实例，就可以在当前的进程中创建一个新的线程。该类拥有多个构造函数，它们将接受一个类型为System.Threading.ThreadStart或System.Threading.ParametrizedThreadStart的委托对象作为参数，线程被创建出来后首先执行该委托对象所引用的方法。使用ParametrizedThreadStart类型的委托对象允许用户为新线程将要执行的方法传入一个对象作为参数。Thread类的一些构造函数还接受一个整型参数用于设置线程要使用的最大栈的大小，该值至少为128KB（即131072字节）。创建了Thread类型的实例后，必须调用Thread.Start()方法以真正启动这个线程。

例5-3

```
using System.Threading;
class Program {
   static void f1() { System.Console.WriteLine("f1"); }
   void f2() { System.Console.WriteLine("f2"); }
   static void f3( object obj ) { System.Console.WriteLine(
                                  "f3 obj = {0}",obj); }
   static void Main() {
      // Explicitly specify the delegate class 'ThreadStart'.
      Thread t1 = new Thread( new ThreadStart(f1) );

      Program program = new Program();
      // Here, we use the C#2 facility to infer the delegate class
      // 'ThreadStart' from the definition of the 'Thread' class
      // constructor without argument.
      Thread t2 = new Thread( program.f2 );

      // Here, the C#2 compiler infer the delegate class
      // 'ParametrizedThreadStart' since the 'f3()' method
      // accept an 'object' parameter.
      Thread t3 = new Thread( f3 );

      t1.Start(); t2.Start(); t3.Start("hello");
      t1.Join();  t2.Join();  t3.Join();
   }
}
```

该程序输出：

```
f1
f2
f3 obj = hello
```

在这个例子中，我们使用Join()方法挂起当前线程，直到调用Join()方法的线程执行完毕。该方法还存在包含参数的重载版本，其中的参数用于指定等待线程结束的最长时间（即超时）所花费的毫秒数。如果线程中的工作在规定的超时时段内结束，该版本的Join()方法将返回一个布尔量True。

### 5.3.7　挂起一个线程

可以使用Thread类的Sleep()方法将一个正在执行的线程挂起一段特定的时间，还可以通过一个以毫秒为单位的整型值或者一个System.TimeSpan结构的实例设定这段挂起的时间。该结构的一个实例可以设定一个精度为1/10 ms（100 ns）的时间段，但是Sleep()方法的最高精度只有1ms。

```
// The current thread is suspended for two seconds.
Thread.Sleep( 2000 ) ;
```

我们也可以从将要挂起的线程自身或者另一个线程中使用Thread类的Suspend()方法将一个线程的活动挂起。在这两种情况中，线程都将被阻塞直到另一个线程调用了Resume()方法。相对于Sleep()方法，Suspend()方法不会立即将线程挂起，而是在线程到达下一个安全点之后，CLR才会将该线程挂起。安全点的概念参见4.7.11节。

### 5.3.8　终止一个线程

一个线程可以在以下场景中将自己终止。

- 从自己开始执行的方法（主线程中的Main()方法，其他线程中ThreadStart委托对象所引用的方法）中退出。
- 被自己终止。
- 被另一个线程终止。

第一种情况不太重要，我们将主要关注另两种情况。在这两种情况中，都可以使用Abort()方法（通过当前线程或从当前线程之外的一个线程）。使用该方法将在线程中引发一个类型为ThreadAbortException的异常。由于线程正处于一种被称为AbortRequested的特殊状态，该异常具有一个特殊之处：当它被异常处理所捕获后，将自动被重新抛出。只有在异常处理中调用Thread.ResetAbort()这个静态方法（如果我们有足够的权限）才能阻止它的传播。

例5-4　主线程的自杀

```
using System;
using System.Threading;
namespace ThreadTest{
   class Program {
      static void Main() {
         Thread t = Thread.CurrentThread;
         try{
            t.Abort();
         }
         catch( ThreadAbortException ) {
            Thread.ResetAbort();
         }
      }
   }
}
```

当线程A对线程B调用了Abort()方法，建议调用B的Join()方法，让A一直等待直到B终止。Interrupt()方法也可以将一个处于阻塞状态的线程（即由于调用了Wait()、Sleep()或者Join()其中一个方法而阻塞）终止。该方法会根据要被终止的线程是否处于阻塞状态而表现出不同的行为。

- 如果该方法被另一个线程调用时，要被终止的线程处于阻塞状态，那么会产生ThreadInterruptedException异常。
- 如果该方法被另一个线程调用时，要被终止的线程不处于阻塞状态，那么一旦该线程进入阻塞状态，就会引发异常。这种行为与线程对自己调用Interrupt()方法是一样的。

### 5.3.9　前台线程与后台线程

Thread类提供了IsBackground{get;set}的布尔属性。当前台线程还在运行时，它会阻止进程被终止。另一方面，一旦所指的进程中不再有前台线程，后台线程就会被CLR自动终止（调用Abort()方法）。IsBackground的默认值为false，这意味着所有的线程默认情况处于前台状态。

### 5.3.10　受托管线程的状态图

Thread类拥有一个System.Threading.ThreadState枚举类型的字段ThreadState，它包含以下枚举值：

```
Aborted              AbortRequested          Background
Running              Stopped                 StopRequested
Suspended            SuspendRequested        Unstarted
WaitSleepJoin
```

有关每个状态的具体描述可以在MSDN上一篇名为“ThreadState Enumeration”的文章中找到。该枚举类型是一个二进制位域，这表示一个该类型的实例可以同时表示多个枚举值。例如，一个线程可以同时处于Running、AbortRequested和Background这三种状态。二进制位域的概念参见10.11.3节。

根据我们在前面的章节中所了解的知识，我们定义了如图5-1所示的简化的状态图。

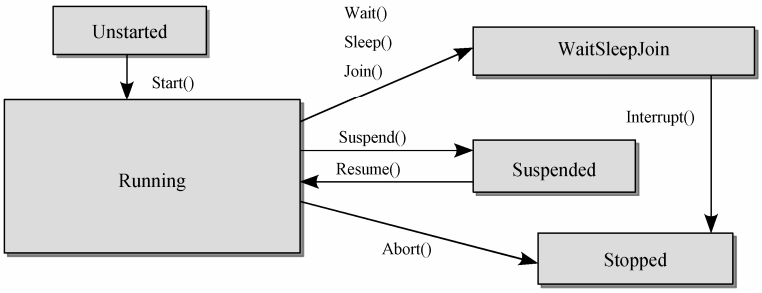


图5-1　简化的托管线程状态图

## 5.4　访问资源同步简介

在多线程应用（一个或多个处理器）的计算中会使用到同步这个词。实际上，这些应用程序的特点就是它们拥有多个执行单元，而这些单元在访问资源的时候可能会发生冲突。线程间会共享同步对象，而同步对象的目的在于能够阻塞一个或多个线程，直到另一个线程使得某个特定条件得到满足。

我们将看到，存在多种同步类与同步机制，每种制针对一个或一些特定的需求。如果要利用同步构建一个复杂的多线程应用程序，那么很有必要先掌握本章的内容。我们将在下面的内容中尽力区分他们，尤其要指出那些在各个机制间最微妙的区别。

合理地同步一个程序是最精细的软件开发任务之一，单这一个主题就足以写几本书。在深入到细节之前，应该首先确认使用同步是否不可避免。通常，使用一些简单的规则可以让我们远离同步问题。在这些规则中有线程与资源的亲缘性规则，我们将在稍后介绍。

应该意识到，对程序中资源的访问进行同步时，其难点来自于是使用细粒度锁还是粗粒度锁这个两难的选择。如果在访问资源时采用粗粒度的同步方式，虽然可以简化代码但是也会把自己暴露在争用瓶颈的问题上。如果粒度过细，代码又会变的很复杂，以至于维护工作令人生厌。然后又会遇上死锁和竞态条件这些在下面章节将要介绍的问题。

因此在我们开始谈论有关同步机制之前，有必要先了解一下有关竞态条件和死锁的概念。

### 5.4.1　竞态条件

竞态条件指的是一种特殊的情况，在这种情况下各个执行单元以一种没有逻辑的顺序执行动作，从而导致意想不到的结果。

举一个例子，线程T修改资源R后，释放了它对R的写访问权，之后又重新夺回R的读访问权再使用

它，并以为它的状态仍然保持在它释放它之后的状态。但是在写访问权释放后到重新夺回读访问权的这段时间间隔中，可能另一个线程已经修改了R的状态。

另一个经典的竞态条件的例子就是生产者/消费者模型。生产者通常使用同一个物理内存空间保存被生产的信息。一般说来，我们不会忘记在生产者与消费者的并发访问之间保护这个空间。容易被我们忘记的是生产者必须确保在生产新信息前，旧的信息已被消费者所读取。如果我们没有采取相应的预防措施，我们将面临生产的信息从未被消费的危险。

如果静态条件没有被妥善的管理，将导致安全系统的漏洞。同一个应用程序的另一个实例很可能会引发一系列开发者所预计不到的事件。一般来说，必须对那种用于确认身份鉴别结果的布尔量的写访问做最完善的保护。如果没有这么做，那么在它的状态被身份鉴别机制设置后，到它被读取以保护对资源的访问的这段时间内，很有可能已经被修改了。已知的安全漏洞很多都归咎于对静态条件不恰当的管理。其中之一甚至影响了Unix操作系统的内核。

### 5.4.2　死锁

死锁指的是由于两个或多个执行单元之间相互等待对方结束而引起阻塞的情况。例如：

一个线程T1获得了对资源R1的访问权。

一个线程T2获得了对资源R2的访问权。

T1请求对R2的访问权但是由于此权力被T2所占而不得不等待。

T2请求对R1的访问权但是由于此权力被T1所占而不得不等待。

T1和T2将永远维持等待状态，此时我们陷入了死锁的处境！这种问题比你所遇到的大多数的bug都要隐秘，针对此问题主要有三种解决方案：

- 在同一时刻不允许一个线程访问多个资源。
- 为资源访问权的获取定义一个关系顺序。换句话说，当一个线程已经获得了R1的访问权后，将无法获得R2的访问权。当然，访问权的释放必须遵循相反的顺序。
- 为所有访问资源的请求系统地定义一个最大等待时间（超时时间），并妥善处理请求失败的情况。几乎所有的.NET的同步机制都提供了这个功能。

前两种技术效率更高但是也更加难于实现。事实上，它们都需要很强的约束，而这点随着应用程序的演变将越来越难以维护。尽管如此，使用这些技术不会存在失败的情况。

大的项目通常使用第三种方法。事实上，如果项目很大，一般来说它会使用大量的资源。在这种情况下，资源之间发生冲突的概率很低，也就意味着失败的情况会比较罕见。我们认为这是一种乐观的方法。秉着同样的精神，我们在19.5节描述了一种乐观的数据库访问模型。

## 5.5　使用 volatile 字段与 Interlocked 类实现同步

### 5.5.1　volatile 字段

volatile字段可以被多个线程访问。我们假设这些访问没有做任何同步。在这种情况下，CLR中一些用于管理代码和内存的内部机制将负责同步工作，但是此时不能确保对该字段读访问总能读取到最新的值，而声明为volatile的字段则能提供这样的保证。在C#中，如果一个字段在它的声明前使用了volatile关键字，则该字段被声明为volatile。

不是所有的字段都可以成为volatile，成为这种类型的字段有一个条件。如果一个字段要成为volatile，它的类型必须是以下所列的类型中的一种：

- 引用类型（这里只有访问该类型的引用是同步的，访问其成员并不同步）。
- 一个指针（在不安全的代码块中）。

- sbyte、byte、short、ushort、int、uint、char、float、bool（工作在64位处理器上时为double、long与ulong）。
- 一个使用以下底层类型的枚举类型：byte、sbyte、short、ushort、int、uint（工作在64位的处理器上时为double、long与ulong）。

你可能已经注意到了，只有值或者引用的位数不超过本机整型值的位数（4或8由底层处理器决定）的类型才能成为volatile。这意味着对更大的值类型进行并发访问必须进行同步，下面我们将会对此进行讨论。

### 5.5.2　System.Threading.Interlocked类

经验显示，那些需要在多线程情况下被保护的资源通常是整型值，而这些被共享的整型值最常见的操作就是增加/减少以及相加。.NET Framework利用System.Threading.Interlocked类提供了一个专门的机制用于完成这些特定的操作。这个类提供了Increment()、Decrement()与Add()三个静态方法，分别用于对int或者long类型变量的递增、递减与相加操作，这些变量以引用方式作为参数传入。我们认为使用Interlocked类让这些操作具有了*原子性*。

下面的程序显示了两个线程如何并发访问一个名为counter的整型变量。一个线程将其递增5次，另一个将其递减5次。

例5-5

```
using System.Threading;
class Program {
   static long counter = 1;
   static void Main() {
      Thread t1 = new Thread(f1);
      Thread t2 = new Thread(f2);
      t1.Start(); t2.Start(); t1.Join(); t2.Join();
   }
   static void f1() {
      for ( int i = 0; i < 5; i++ ) {
         Interlocked.Increment( ref counter );
         System.Console.WriteLine( “counter++ {0}” , counter );
         Thread.Sleep( 10 );
      }
   }
   static void f2() {
      for ( int i = 0; i < 5; i++ ) {
         Interlocked.Decrement( ref counter );
         System.Console.WriteLine( “counter-- {0}” , counter );
         Thread.Sleep( 10 );
      }
   }
}
```

该程序输出（以非确定方式输出，意味着每执行一次显示的结果都是不同的）：

```
counter++ 2
counter-- 1
counter++ 2
counter-- 1
counter++ 2
counter-- 1
counter++ 2
counter-- 1
```

```
counter-- 0
counter++ 1
```

如果我们不让这些线程在每次修改变量后休眠10毫秒，那么它们将有足够的时间在一个时间片中完成它们的任务，那样也就不会出现交叉操作，更不用说并发访问了。

### 5.5.3　Interlocked 类提供的其他功能

Interlocked类还允许使用Exchange()静态方法，以原子操作的形式交换某些变量的状态。还可以使用CompareExchange()静态方法在满足一个特定条件的基础上以原子操作的形式交换两个值。

## 5.6　使用 System.Threading.Monitor 类与 C#的 lock 关键字实现同步

以原子操作的方式完成简单的操作无疑是很重要的，但是这还远不能涵盖所有需要用到同步的事例。System.Threading.Monitor类几乎允许将任意一段代码设置为在某个时间仅能被一个线程执行。我们将这段代码称之为临界区。

### 5.6.1　Enter()方法和 Exit()方法

Monitor类提供了Enter(object)与Exit(object)这两个静态方法。这两个方法以一个对象作为参数，该对象提供了一个简单的方式用于唯一标识那个将以同步方式访问的资源。当一个线程调用了Enter()方法，它将等待以获得访问该引用对象的独占权（仅当另一个线程拥有该权力的时候它才会等待）。一旦该权力被获得并使用，线程可以对同一个对象调用Exit()方法以释放该权力。

一个线程可以对同一个对象多次调用Enter()，只要对同一对象调用相同次数的Exit()来释放独占访问权。

一个线程也可以在同一时间拥有多个对象的独占权，但是这样会产生死锁的情况。

绝不能对一个值类型的实例调用Enter()与Exit()方法。

不管发生了什么，必须在finally子句中调用Exit()以释放所有的独占访问权。

如果在例5-5中，一个线程非要将counter做一次平方而另一个线程非要将counter乘2，我们就不得不用Monitor类去替换对Interlocked类的使用。f1()与f2()的代码将变成下面这样：

例5-6[①]

```
using System.Threading;
class Program {
   static long counter = 1;
   static void Main() {
      Thread t1 = new Thread( f1 );
      Thread t2 = new Thread( f2 );
      t1.Start(); t2.Start(); t1.Join(); t2.Join();
   }
   static void f1() {
      for (int i = 0; i < 5; i++){
         try{
            Monitor.Enter( typeof( Program ) );
            counter *= counter;
         }
         finally{ Monitor.Exit( typeof( Program ) ); }
         System.Console.WriteLine("counter^2 {0}", counter);
         Thread.Sleep(10);
```

① 此段代码有误，参见勘误http://www.practicaldot.net/en2/Chapter 5/Listing_5_6.htm。——译者注

```
        }
    }
    static void f2() {
        for (int i = 0; i < 5; i++){
            try{
                Monitor.Enter( typeof( Program ) );
                counter *= 2;
            }
            finally{ Monitor.Exit( typeof( Program ) ); }
            System.Console.WriteLine("counter*2 {0}", counter);
            Thread.Sleep(10);
        }
    }
}
```

人们很容易想到用counter来代替typeof(Program)，但是counter是一个值类型的静态成员。需要注意平方和倍增操作是不满足交换律的，所以counter的最终结果是非确定性的。

### 5.6.2　C#的 lock 关键字

C#语言通过lock关键字提供了一种比使用Enter()和Exit()方法更加简洁的选择。我们的程序可以改写为下面这个样子：

例5-7

```
using System.Threading;
class Program {
    static long counter = 1;
    static void Main() {
        Thread t1 = new Thread(f1);
        Thread t2 = new Thread(f2);
        t1.Start(); t2.Start(); t1.Join(); t2.Join();
    }
    static void f1() {
        for (int i = 0; i < 5; i++){
            lock( typeof(Program) ) { counter *= counter; }
            System.Console.WriteLine("counter^2 {0}", counter);
            Thread.Sleep(10);
        }
    }
    static void f2() {
        for (int i = 0; i < 5; i++){
            lock( typeof(Program) ) { counter *= 2; }
            System.Console.WriteLine("counter*2 {0}", counter);
            Thread.Sleep(10);
        }
    }
}
```

和for以及if关键字一样，如果被lock关键字定义的块仅包含一条指令，就不再需要花括号。我们可以再次改写为：

```
...
lock( typeof( Program ) )
    counter *= counter;
...
```

使用lock关键字将引导C#编译器创建出相应的try/finally块，这样仍旧可以预期到任何可能引发的异常。可以使用Reflector或者ildasm.exe工具验证这一点。

### 5.6.3　SyncRoot 模式

和前面的例子一样，我们通常在一个静态方法中使用Monitor类配合一个Type类的实例。同样，我们往往会在一个非静态方法中使用this关键字来实现同步。在两种情况下，我们都是通过一个在类外部可见的对象对自身进行同步。如果其他部分的代码也利用这些对象来实现自身的同步，就会出现问题。为了避免这种潜在的问题，我们推荐使用一个类型为object的名为SyncRoot的私有成员，至于该成员是静态的还是非静态的则由需要而定。

例5-8

```
class Foo {
   private static object staticSyncRoot = new object();
   private object instanceSyncRoot = new object();
   public static void StaticFct() {
      lock ( staticSyncRoot ) { /*...*/ }
   }
   public void InstanceFct() {
      lock ( instanceSyncRoot ) { /*...*/ }
   }
}
```

System.Collections.ICollection接口提供了object类型的SyncRoot{get;}属性。大多数的集合类（泛型或非泛型）都实现了该接口。同样地，可以使用该属性同步对集合中元素的访问。不过在这里SyncRoot模式并没有被真正的应用，因为我们对访问进行同步所使用对象不是私有的。

例5-9

```
using System.Collections.Generic;
using System.Collections;
public class Program {
   public static void Main() {
      List<int> list = new List<int>();
      // ...
      lock ( ( (ICollection) list ).SyncRoot ) {
         foreach (int i in list) {
            /* ... */
         }
      }
   }
}
```

### 5.6.4　线程安全类

若一个类的每个实例在同一时间不能被一个以上的线程所访问，则该类称之为一个线程安全的类。为了创建一个线程安全的类，只需将我们见过的SyncRoot模式应用于它所包含的方法。如果一个类想变成线程安全的，而又不想为类中代码增加过多负担，那么有一个好方法就是像下面这样为其提供一个经过线程安全包装的继承类。

例5-10

```
class Foo {
   private class FooSynchronized : Foo {
      private object syncRoot = new object();
      private Foo m_Foo;
      public FooSynchronized( Foo foo ) { m_Foo = foo; }
      public override bool IsSynchronized { get { return true; } }
      public override void Fct1(){ lock( syncRoot ) { m_Foo.Fct1(); } }
```

```
      public override void Fct2(){ lock( syncRoot ) { m_Foo.Fct2(); } }
   }
   public virtual bool IsSynchronized { get { return false; } }
   public static Foo Synchronized(Foo foo){
      if( ! foo.IsSynchronized )
         return new FooSynchronized( foo );
      return foo;
   }
   public virtual void Fct1() { /*...*/ }
   public virtual void Fct2() { /*...*/ }
}
```

另一种方法就是使用System.Runtime.Remoting.Contexts.SynchronizationAttribute，这点我们将在本章稍后讨论。

### 5.6.5　Monitor.TryEnter()方法

```
public static bool TryEnter(object [,int] )
```

该方法与Enter()相似，只不过它是非阻塞的。如果资源的独占访问权已经被另一个线程占据，该方法将立即返回一个false返回值。我们也可以调用TryEnter()方法，让它以毫秒为单位阻塞一段有限的时间。因为该方法的返回结果并不确定，并且当获得独占访问权后必须在finally子句中释放该权力，所以建议当TryEnter()失败时立即退出正在调用的函数：

例5-11[①]

```
using System.Threading;
class Program {
   private static object staticSyncRoot = new object();
   static void Main() {
      // Comment this line to test the case where the
      // 'TryEnter()' method returns true.
      Monitor.Enter( staticSyncRoot );
      Thread t1 = new Thread(f1);
      t1.Start(); t1.Join();
   }
   static void f1() {
      bool bOwner = false;
      try {
         if( ! Monitor.TryEnter( staticSyncRoot ) )
            return;
         bOwner = true;
         // ...
      }
      finally {
         // Don't call the 'Monitor.Exit()' if you don't own the access.
         // Keep in mind that the finally bloc will be executed
         // whatever happen.
         if( bOwner )
            Monitor.Exit( staticSyncRoot );
      }
   }
}
```

### 5.6.6　Monitor类的Wait()方法, Pulse()方法以及PulseAll()方法

```
public static bool Wait( object [,int] )
```

① 此处代码有误，参见勘误http://www.practicaldot.net/en2/Chapter 5/Listing_5_11.htm。——译者注

```
public static void Pulse( object )
public static void PulseAll( object )
```

Wait()、Pulse()与PulseAll()方法必须在一起使用并且需要结合一个小场景才能被正确理解。我们的想法是这样的：一个线程获得了某个对象的独占访问权，而它决定等待（通过调用Wait()）直到该对象的状态发生变化。为此，该线程必须暂时失去对象独占访问权，以便让另一个线程修改对象的状态。修改对象状态的线程必须使用Pulse()方法通知那个等待线程修改完成。下面有一个小场景具体说明了这一情况。

- 拥有OBJ对象独占访问权的T1线程，调用Wait(OBJ)方法将它自己注册到OBJ对象的被动等待列表中。
- 由于以上的调用，T1失去了对OBJ的独占访问权。因此，另一个线程T2通过调用Enter(OBJ)获得OBJ的独占访问权。
- T2最终修改了OBJ的状态并调用Pulse(OBJ)通知了这次修改。该调用将导致OBJ被动等待列表中的第一个线程（在这里是T1）被移到OBJ的主动等待列表的首位。而一旦OBJ的独占访问权被释放，OBJ主动等待列表中的第一个线程将被确保获得该权力。然后它就从Wait(OBJ)方法中退出等待状态。
- 在我们的场景中，T2调用Exit(OBJ)以释放对OBJ的独占访问权，接着T1恢复访问权并从Wait(OBJ)方法中退出。
- PulseAll()将使得被动等待列表中的线程全部转移到主动等待列表中。注意这些线程将按照它们调用Wait()的顺序到达非阻塞态。

如果Wait(OBJ)被一个调用了多次Enter(OBJ)的线程所调用，那么该线程将需要调用相同次数的Exit(OBJ)以释放对OBJ的访问权。即使在这种情况下，另一个线程调用一次Pulse(OBJ)就足以将第一个线程变成非阻塞态。

下面的程序通过ping与pong两个线程以交替的方式使用一个ball对象的访问权来演示该功能。

例5-12

```
using System.Threading;
public class Program {
   static object ball = new object();
   public static void Main() {
      Thread threadPing = new Thread( ThreadPingProc );
      Thread threadPong = new Thread( ThreadPongProc );
      threadPing.Start(); threadPong.Start();
      threadPing.Join();  threadPong.Join();
   }
   static void ThreadPongProc() {
      System.Console.WriteLine("ThreadPong: Hello!");
      lock ( ball )
         for (int i = 0; i < 5; i++){
            System.Console.WriteLine("ThreadPong: Pong");
            Monitor.Pulse( ball );
            Monitor.Wait( ball );
         }
      System.Console.WriteLine("ThreadPong: Bye!");
   }
   static void ThreadPingProc() {
      System.Console.WriteLine("ThreadPing: Hello!");
      lock ( ball )
```

```
            for(int i=0; i< 5; i++){
               System.Console.WriteLine("ThreadPing: Ping");
               Monitor.Pulse( ball );
               Monitor.Wait( ball );
            }
         System.Console.WriteLine("ThreadPing: Bye!");
      }
   }
```

该程序输出（以不确定的方式）：

```
ThreadPing: Hello!
ThreadPing: Ping
ThreadPong: Hello!
ThreadPong: Pong
ThreadPing: Ping
ThreadPong: Pong
ThreadPing: Ping
ThreadPong: Pong
ThreadPing: Ping
ThreadPong: Pong
ThreadPing: Ping
ThreadPong: Pong
ThreadPing: Bye!
```

pong线程没有结束并且仍然阻塞在Wait()方法上。由于pong线程是第二个获得ball对象的独占访问权的，所以才导致了该结果。

## 5.7　使用 Win32 对象同步：互斥体、事件与信号量

System.Threading.WaitHandle抽象基类提供了三个继承类，那些在Win32环境下使用过同步技术的用户都十分了解它们的用途。

- Mutex类（mutex这个词是MUTual Exclusion的缩写）。
- AutoResetEvent类定义了一个会自动重置的事件。ManualResetEvent类定义了一个需要手动重置的事件。这两个类继承于代表一般事件的EventWaitHandle类。
- Semaphore类。

WaitHandle类和它的继承类有一个特点，它们都重写了WaitOne()方法并实现了WaitAll()、WaitAny()与SignalAndWait()静态方法。它们分别允许等待对象收到信号或者数组中的所有对象收到信号，以等待数组中至少一个对象收到信号以及向一个对象发出信号并等待另一个对象。与Monitor类和Interlocked类相反，以上这些类必须被实例化后才能使用。因此这里所要考虑的对象不是被同步的对象，而是用于同步的对象。也就意味着作为参数传递给WaitAll()、WaitAny()以及SignalAndWait()静态方法的对象本身全部都是互斥体、事件或者信号量。

### 5.7.1　共享 Win32 同步对象

Monitor类与WaitHandle的继承类各自所提倡的同步模型，还存在一个大的差别值得注意。使用Monitor类仅限于一个单独的进程中，而WaitHandle类的继承类调用的是非托管的Win32对象，它们可以被处在同一机器上的多个进程所访问。因此，WaitHandle类的继承类提供了某些接受一个字符串参数的构造函数，该字符串用于命名同步对象。这些类还提供了OpenExisting(string)方法用于获得一个已存在的某个特定名称的同步对象的引用。

由于WaitHandle类继承于MarshalByRefObject，所以我们还可以做得更多。利用.NET的Remoting

技术，这样一个同步对象甚至可以被不同机器上的多个进程所共享。

在6.10.1节，我们解释了如何使用System.Security.AccessControl命名空间下的MutexSecurity、EventWaitHandleSecurity以及SemaphoreSecurity类型来操纵Windows赋予同步对象的访问权。

### 5.7.2　互斥体

就功能而言，互斥体（mutex）的使用和管程的使用很接近，不过有一点区别。

- 我们可以在同一台机器甚至远程机器上的多个进程中使用同一个互斥体。
- 使用管程无法等待多个对象。
- Mutex类没有Monitor类中的Wait()、Pulse()以及PulseAll()方法所提供的功能。

我们知道，如果需要同步的访问处在一个进程中，应该考虑使用Monitor类代替互斥体因为它的效率更高。

下面的程序展示了使用一个*有名互斥体*来保护对资源的访问，该互斥体被同一台机器的多个进程共享。被共享的资源是一个文件，每个程序的实例将在文件中写入10行文字。

例5-13

```
using System.Threading;
using System.IO;
class Program {
   static void Main() {
      // The mutex created is named 'MutexTest'.
      Mutex mutexFile = new Mutex(false, "MutexTest");
      for (int i = 0; i < 10; i++){
         mutexFile.WaitOne();
         // Open the file, write 'Hello i' and then close it.
         FileInfo fi = new FileInfo("tmp.txt");
         StreamWriter sw = fi.AppendText();
         sw.WriteLine("Hello {0}", i);
         sw.Flush();
         sw.Close();
         System.Console.WriteLine("Hello {0}", i);
         // Wait one second to make obvious the sharing of mutex.
         Thread.Sleep(1000);
         mutexFile.ReleaseMutex();
      }
      mutexFile.Close();
   }
}
```

注意，使用WaitOne()方法会阻塞当前线程，直到获得互斥体，使用ReleaseMutex()方法则可以释放互斥体。

在本程序中new Mutex不一定意味着创建一个互斥体，可能是创建一个对互斥体"MutexTest"的引用。只有当该互斥体不存在的时候，操作系统才会创建它。同样，只有当没有其他进程引用互斥体时，Close()方法才会真正销毁它。

对于那些习惯在Win32环境中使用命名互斥体技术来避免在同一台机器上运行两个相同程序的用户，应该知道在.NET中提供了更好的方法，参见5.2.4节。

### 5.7.3　事件

相比目前所看到的同步机制，事件（event）并没有显式定义一个在给定时间属于某个线程的某个

资源的概念。事件被用于在线程间传递通知，该通知表示“某个事件发生了”。被关注的事件会与System.Threading.AutoResetEvent和System.Threading.ManualResetEvent这两个事件类的一个实例关联。这两个类都继承于System.Threading.EventWaitHandle类。一个EventWaitHandle的实例可以被System.Threading.EventResetMode枚举类型中AutoReset或ManualReset两者之一的值初始化，这也意味着你不是非要直接创建其中的一个继承类[①]。

具体说来，一个线程通过调用代表事件的对象的WaitOne()这个阻塞方法[②]来等待一个事件被触发。另一个线程调用事件对象的Set()方法以触发事件从而使得第一个线程继续执行。

自动重置事件与手动重置事件的区别在于，后者需要调用Reset()方法将事件返回到非活动状态。

> 手动重置与自动重置的区别可能远比你想象的要大。如果几个线程在等待同一个自动重置的事件，那么该事件需要为每个线程都触发一次。而在手动重置的情况下只需要简单的触发一次事件就可以让所有的阻塞线程继续执行。

下面的程序创建了两个线程t0和t1，它们以不同的速度增加自己的计数器。当t0的计数器达到2，它就触发events[0]事件；当t1的计数器达到3，它就触发events[1]事件。主线程等待两个事件都被触发以显式一条消息。

例5-14

```
using System;
using System.Threading;
class Program {
   static EventWaitHandle[] events;
   static void Main() {
      events = new EventWaitHandle[2];
      // Initial event state : false
      events[0] = new EventWaitHandle( false ,EventResetMode.AutoReset );
      events[1] = new EventWaitHandle( false ,EventResetMode.AutoReset );
      Thread t0 = new Thread( ThreadProc0 );
      Thread t1 = new Thread( ThreadProc1 );
      t0.Start(); t1.Start();
      AutoResetEvent.WaitAll( events );
      Console.WriteLine( “MainThread: Thread0 reached  2”   +
                         “ and Thread1 reached 3.”  );
      t0.Join();t1.Join();
   }
   static void ThreadProc0() {
      for ( int i = 0; i < 5; i++ ) {
         Console.WriteLine( “Thread0: {0}”, i );
         if ( i == 2 ) events[0].Set();
         Thread.Sleep( 100 );
      }
   }
   static void ThreadProc1() {
      for ( int i = 0; i < 5; i++ ) {
         Console.WriteLine( “Thread1: {0}”, i );
         if ( i == 3 ) events[1].Set();
         Thread.Sleep( 60 );
      }
```

① 这样就可以避免依赖于一个具体类。——译者注

② 所谓阻塞方法是指WaitOne()方法调用后，如果事件没有被触发，线程将陷入阻塞状态。——译者注

```
    }
}
```

该程序输出：

```
Thread0: 0
Thread1: 0
Thread1: 1
Thread0: 1
Thread1: 2
Thread1: 3
Thread0: 2
MainThread: Thread0 reached 2 and Thread1 reached 3
Thread1: 4
Thread0: 3
Thread0: 4
```

### 5.7.4　信号量

一个System.Threading.Semaphore类的实例可用来限制对一个资源的并发访问数。你可以想象一个停车场的入口大门只包含一定数量的停车位。只有当停车场中还有空位的时候它才会打开。同样，当调用WaitOne()方法试图进入一个信号量（semaphore）的时候，如果当时已经进入的线程的数量达到某个最大值时，该线程就会被阻塞。这个最大的入口的数量由Semaphore类的构造函数的第二个参数设定，而第一个参数则定义了初始时的入口数量。如果第一个参数的值小于第二个，线程在调用构造函数时会自动地占据二者差值数量的入口。最后这点也说明了同一个线程可以占据同一个信号量的多个入口。

下面这个例子通过启动3个线程让它们定时的尝试进入一个最大入口数为5的信号量，来举例说明上述内容。主线程占据了该信号量的三个入口，迫使另外3个线程去共享剩下的2个入口。

例5-15

```
using System;
using System.Threading;
class Program {
   static Semaphore semaphore;
   static void Main() {
      // Initial number of free slots                : 2.
      // Maximal number of slots used simulteanously : 5.
      // Number of slot owned by the main thread     : 3 (5-2).
      semaphore = new Semaphore( 2, 5 );
      for ( int i = 0; i < 3; ++i ) {
         Thread t = new Thread( WorkerProc );
         t.Name = “Thread” + i;
         t.Start();
         Thread.Sleep( 30 );
      }
   }
   static void WorkerProc() {
      for ( int j = 0; j < 3; j++ ) {
         semaphore.WaitOne();
         Console.WriteLine( Thread.CurrentThread.Name + “: Begin” );
         Thread.Sleep( 200 );  // Simulate a 200 milliseconds task.
         Console.WriteLine( Thread.CurrentThread.Name + “: End” );
         semaphore.Release();
      }
```

```
    }
}
```

下面是程序的输出。我们看到其中并发工作的子线程从未超过2个。

```
Thread0: Begin
Thread1: Begin
Thread0: End
Thread2: Begin
Thread1: End
Thread1: Begin
Thread2: End
Thread2: Begin
Thread1: End
Thread1: Begin
Thread2: End
Thread2: Begin
Thread1: End
Thread0: Begin
Thread2: End
Thread0: End
Thread0: Begin
Thread0: End
```

## 5.8　利用 System.Threading.ReaderWriterLock 类实现同步

System.Threading.ReaderWriterLock类实现了多用户读/单用户写的同步访问机制。与Monitor类或Mutex类提供的独占访问模型相反，该机制考虑到一个线程在访问资源的时候可能仅仅需要读或者写这一事实。如果某一时刻尚未获得写访问权，则此时可以获得读访问权。如果某一时刻没有任何对资源进行访问的迹象，则此时可以获得写访问权。此外，如果一个写访问权已被申请而尚未获得，那么所有新的读访问将暂时搁置。

如果该模型适用，那么相对于Monitor类或者Mutex类，它总是一个更好的选择。事实上，独占访问模型不允许任何形式的并发访问以致于效率总是不高。此外，我们应该注意到，应用程序通常在访问资源的时候更多的是读而不是写。

和互斥体以及事件一致，但与Monitor类不同的是，ReaderWriterLock类必须在使用前被初始化。必须从用于同步的对象的角度去考虑，而不是被同步的对象。

下面是一个展示使用该类的例子，不过这里并没有使用到它的全部功能。实际上，DowngradeFromWriterLock()以及UpgradeToWriterLock()方法允许我们请求更换访问权而不释放当前的访问。

例5-16

```
using System;
using System.Threading;
class Program {
   static int theResource   = 0;
   static ReaderWriterLock rwl = new ReaderWriterLock();
   static void Main() {
      Thread tr0 = new Thread( ThreadReader );
      Thread tr1 = new Thread( ThreadReader );
      Thread tw = new Thread( ThreadWriter );
      tr0.Start(); tr1.Start(); tw.Start();
      tr0.Join();  tr1.Join();  tw.Join();
   }
```

```
    static void ThreadReader() {
       for ( int i = 0; i < 3; i++ )
          try{
             // AcquireReaderLock() raises an
             // ApplicationException when timed out.
             rwl.AcquireReaderLock(1000);
             Console.WriteLine("Begin Read  theResource = {0}",theResource);
             Thread.Sleep(10);
             Console.WriteLine("End   Read  theResource = {0}",theResource);
             rwl.ReleaseReaderLock();
          }
          catch ( ApplicationException ) { /* ... */ }
    }
    static void ThreadWriter() {
       for (int i = 0; i < 3; i++)
          try{
             // AcquireReaderLock() raises an
             // ApplicationException when timed out.
             rwl.AcquireWriterLock(1000);
             Console.WriteLine("Begin Write theResource = {0}",theResource);
             Thread.Sleep(100);
             theResource  ++;
             Console.WriteLine("End   Write theResource = {0}",theResource);
             rwl.ReleaseWriterLock();
          }
          catch ( ApplicationException ) { /* ... */ }
    }
}
```

该程序输出：

```
Begin Read  theResource = 0
Begin Read  theResource = 0
End   Read  theResource = 0
End   Read  theResource = 0
Begin Write theResource = 0
End   Write theResource = 1
Begin Read  theResource = 1
Begin Read  theResource = 1
End   Read  theResource = 1
End   Read  theResource = 1
Begin Write theResource = 1
End   Write theResource = 2
Begin Read  theResource = 2
Begin Read  theResource = 2
End   Read  theResource = 2
End   Read  theResource = 2
Begin Write theResource = 2
End   Write theResource = 3
```

## 5.9　利用 System.Runtime.Remoting.Contexts.SynchronizationAttribute 实现同步

将System.Runtime.Remoting.Contexts.SynchronizationAttribute应用于某个类后，该类的实例无法被多个线程同时访问。我们说，这样的类是线程安全的。

为了获得上述的特性，被应用该attribute的类必须是上下文绑定的。换句话说，它必须继承于System.ContextBoundObject类。上下文绑定的含义参见22.15节。

下面这个例子演示了如何应用该特性。

例5-17

```
using System.Runtime.Remoting.Contexts;
using System.Threading;
[Synchronization(SynchronizationAttribute.REQUIRED)]
public class Foo : System.ContextBoundObject {
   public void DisplayThreadId() {
      System.Console.WriteLine( "Begin: ManagedThreadId = " +
                                Thread.CurrentThread.ManagedThreadId );
      Thread.Sleep( 1000 );
      System.Console.WriteLine( "End:   ManagedThreadId = " +
                                Thread.CurrentThread.ManagedThreadId );
   }
}
public class Program {
   static Foo m_Objet = new Foo();
   static void Main() {
      Thread t0 = new Thread( ThreadProc );
      Thread t1 = new Thread( ThreadProc );
      t0.Start(); t1.Start();
      t0.Join();  t1.Join();
   }
   static void ThreadProc() {
      for ( int i = 0; i < 2; i++ )
         m_Objet.DisplayThreadId();
   }
}
```

该程序输出：

```
Begin: ManagedThreadId = 27
End:   ManagedThreadId = 27
Begin: ManagedThreadId = 28
End:   ManagedThreadId = 28
Begin: ManagedThreadId = 27
End:   ManagedThreadId = 27
Begin: ManagedThreadId = 28
End:   ManagedThreadId = 28
```

### 5.9.1　同步域简介

为了更好地阅读下面的内容，必须理解以下概念：

- AppDomain（参见4.1节），
- .NET的上下文以及上下文绑定/上下文灵活对象（参见22.15节），
- 消息接收器（参见13.5节）。

同步域是一个完全由CLR接管的实体。该领域包含了一个或多个.NET上下文，因而也将这些上下文中的对象包含其中。一个.NET上下文最多属于一个同步域。此外，同步域的概念要比AppDomain

的概念窄。图5-2展示了进程、AppDomain、同步上下文[①]、.NET上下文和.NET对象的关系。

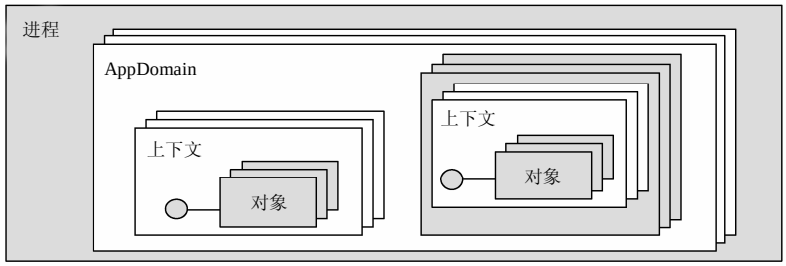


图5-2　进程、AppDomain、同步域、.NET上下文和.NET对象

由于我们正在讨论同步以及多线程应用程序，所以有必要牢记AppDomain与进程中的线程这两个概念是正交的。事实上，一个线程可以自由穿越两个AppDomain之间的边界，例如，只需让处在DB AppDomain的对象B调用处在DA AppDomain的对象A就实现了穿越。此外，一个AppDomain的代码可以被零个，一个或多个线程同时执行。

同步域之所以有意义就在于它不能被多个线程所共享。换句话说，一个处在同步域中的对象的方法是不能被多个线程同时执行的。这也意味着在任一时刻，最多只有一个线程处于同步域中。我们接着谈到对同步域的独占访问权。这种独占访问权的管理还是由CLR负责的。

### 5.9.2　System.Runtime.Remoting.Contexts.Synchronization与同步域

你可能已经猜到，System.Runtime.Remoting.Contexts.SynchronizationAttribute实际上是用来告诉CLR何时创建一个同步域以及如何定义它的边界。这些设置是在声明System.Runtime.Remoting.Contexts.SynchronizationAttribute的时候完成的。声明attribute时使用下面四个值其中之一：NOTSUPPORTED、SUPPORTED、REQUIRED或REQUIRES_NEW。注意REQUIRED是默认值。

AppDomain的成员在一个对象创建另一个对象的时候发生联系。这里我们谈到了创建者对象，但是要考虑到一个对象也可以由一个静态方法创建，而静态方法的执行可能是由一个处在同步域的对象的调用引起的。我们知道在这种情况下，该静态方法为AppDomain增加了新的成员，因此也扮演了创建者对象的角色。下表是对可能出现的四种特性的解释。

| | |
|---|---|
| NOT_SUPPORTED | 该参数确保应用SynchronizationAttribute的类的实例不会属于任何一个同步域（无论它的创建者对象是否属于一个同步域。） |
| SUPPORTED | 该参数说明应用SynchronizationAttribute的类的实例不需要属于某个同步域。尽管如此，如果该实例的创建者对象属于某个同步域，那么它也将属于该同步域。该特性几乎没什么用，因为开发者还必须在类方法中提供另一种同步机制。尽管如此，它最终允许通过一个不需要被同步的对象增加同步域中的成员（这种类型的对象非常少见因为它必须没有任何状态。） |
| REQUIRED | 该参数确保应用SynchronizationAttribute 的类的实例一定属于某个同步域。如果它的创建者对象属于某个同步域，那它也将属于该同步域，否则会为它的创建者对象新建一个同步域 |
| REQUIRES_NEW | 该参数确保应用SynchronizationAttribute 的类的实例将属于一个新的同步域（无论它的创建者对象是否属于一个同步域。） |

① 这里的同步上下文疑为同步域，而且图中似乎缺少同步域的标识。——译者注

这些特性可归纳如下：

| 被应用的值 | 创建者对象是否属于某个同步域 | 被创建的对象将属于…… |
|---|---|---|
| NOT_SUPPORTED | 否 | ……所有的同步域之外 |
| | 是 | |
| SUPPORTED | 否 | ……所有的同步域之外 |
| | 是 | ……创建者对象的同步域 |
| REQUIRED | 否 | ……新的同步域 |
| | 是 | ……创建者对象的同步域 |
| REQUIRES_NEW | 否 | ……新的同步域 |
| | 是 | |

### 5.9.3　重入与同步域

在同步域D中，当拥有独占访问权的T1线程调用了一个处在D外的对象的方法后，CLR将会有两种表现：

- 要么CLR允许另一个线程T2获得对D的独占访问权。在这种情况下，T1可能需要在它调用D外的对象之后进入等待状态，直到T2释放了对D的独占访问权。在这里出现了重入（Reentrancy），因为T1请求再次进入D，如图5-3所示。
- 要么T1仍然占据对D的独占访问权。在这种情况下，另一个希望调用D中对象的方法的线程T2需要等待T1释放它对D的独占访问权。

调用静态方法不会被认为是同步域外的调用。

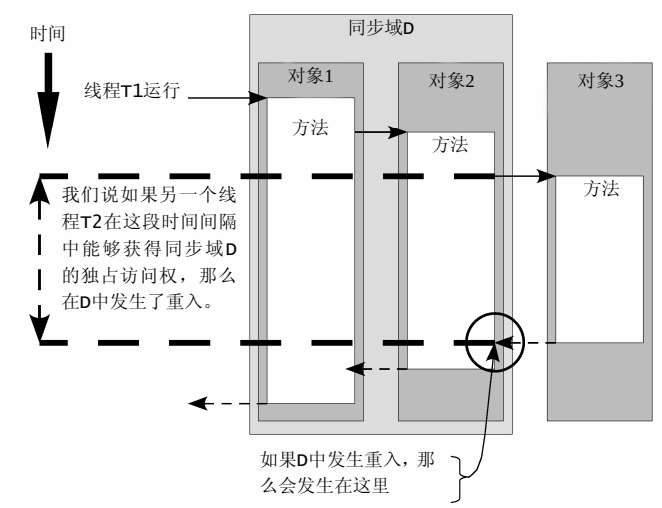


图5-3　同步域中的重入

某些System.Runtime.Remoting.Contexts.Synchronization类的某些构造函数接受一个布尔值作为参数。该布尔值定义了当调用当前同步域外的对象时是否允许发生重入。当该布尔值设为真时，允许发生重入。如果所有的对象都允许重入，只需要将同步域中遇到的第一个对象的synchronizationAttribute的重入参数设为真。

下面的例子用代码说明了图5-3：

例5-18

```
using System;
using System.Runtime.Remoting.Contexts;
using System.Threading;
[Synchronization( SynchronizationAttribute.REQUIRES_NEW, true )]
public class Foo1 : ContextBoundObject {
   public void DisplayThreadId() {
      Console.WriteLine( "Foo1 Begin: ManagedThreadId = " +
                         Thread.CurrentThread.ManagedThreadId );
      Thread.Sleep( 1000 );
      Foo2 obj2 = new Foo2();
      obj2.DisplayThreadId();
      Console.WriteLine( "Foo1 End:   ManagedThreadId = " +
                         Thread.CurrentThread.ManagedThreadId );
   }
}
[Synchronization( SynchronizationAttribute.REQUIRED )]
public class Foo2 : ContextBoundObject {
   public void DisplayThreadId() {
      Console.WriteLine( "Foo2 Begin: ManagedThreadId = " +
                         Thread.CurrentThread.ManagedThreadId );
      Thread.Sleep( 1000 );
      Foo3 obj3 = new Foo3();
      obj3.DisplayThreadId();
      Console.WriteLine( "Foo2 End:   ManagedThreadId = " +
                         Thread.CurrentThread.ManagedThreadId );
   }
}
// We can be sure that instances of the 'Foo3' class will never reside
// in any synchronization domain.
[Synchronization( SynchronizationAttribute.NOT_SUPPORTED )]
public class Foo3 : ContextBoundObject {
   public void DisplayThreadId() {
      Console.WriteLine( "Foo3 Begin: ManagedThreadId = " +
                         Thread.CurrentThread.ManagedThreadId );
      Thread.Sleep( 1000 );
      Console.WriteLine( "Foo3 End:   ManagedThreadId = " +
                         Thread.CurrentThread.ManagedThreadId );
   }
}
public class Program {
   static Foo1 m_Objet = new Foo1();
   static void Main() {
      Thread t1 = new Thread( ThreadProc );
      Thread t2 = new Thread( ThreadProc );
      t1.Start(); t2.Start();
      t1.Join(); t2.Join();
   }
   static void ThreadProc() {
      m_Objet.DisplayThreadId();
   }
}
```

该程序输出如下：

```
Foo1 Begin: ManagedThreadId = 3
Foo2 Begin: ManagedThreadId = 3
Foo1 Begin: ManagedThreadId = 4
Foo2 Begin: ManagedThreadId = 4
Foo3 Begin: ManagedThreadId = 4
Foo3 Begin: ManagedThreadId = 3
Foo3 End:   ManagedThreadId = 4
Foo2 End:   ManagedThreadId = 4
Foo1 End:   ManagedThreadId = 4
Foo3 End:   ManagedThreadId = 3
Foo2 End:   ManagedThreadId = 3
Foo1 End:   ManagedThreadId = 3
```

显然如果我们对Foo1禁用重入，程序将输出如下：

```
Foo1 Begin: ManagedThreadId = 3
Foo2 Begin: ManagedThreadId = 3
Foo3 Begin: ManagedThreadId = 3
Foo3 End:   ManagedThreadId = 3
Foo2 End:   ManagedThreadId = 3
Foo1 End:   ManagedThreadId = 3
Foo1 Begin: ManagedThreadId = 4
Foo2 Begin: ManagedThreadId = 4
Foo3 Begin: ManagedThreadId = 4
Foo3 End:   ManagedThreadId = 4
Foo2 End:   ManagedThreadId = 4
```

重入用于优化对资源的管理，因为它可以全面缩短线程在同步域中使用独占访问权的时间。尽管如此，默认情况下重入功能并没有被激活。事实上，如果发生重入，我们对同步域的概念的理解会有很大的改变。更糟的是，没适当的调整就激活重入功能可能会产生死锁现象。由于以上原因，建议不要使用重入，除非特别需要。

### 5.9.4　另一个名为 Synchronization 的 attribute

.NET Framework在另一个命名空间提供了另外一个同名的attribute。`System.EnterpriseServices.SynchronizationAttribute`拥有同样的目的只不过在内部使用了COM+中用于同步的企业服务。基于以下两个原因，优先选择使用`System.Runtime.Remoting.Contexts.SynchronizationAttribute`。

- 它的使用更加高效。
- 相较于COM+的版本，该机制支持异步调用。

COM+企业服务的概念参见8.6节。

## 5.10　CLR 的线程池

### 5.10.1　简介

线程池的概念并不是什么新概念。尽管如此，在`System.Threading.ThreadPool`类的帮助下，.NET Framework使我们能够比其他技术更方便地使用线程池。

在一个多线程的应用程序中，大多数的线程将它们的时间花费在等待事件上。通常情况下，线程并没有充分地发挥作用。此外，事实上你必须在设计应用程序的时候考虑如何管理线程，而如果没有这个额外的任务，你就可以工作得轻松点了。

线程池的使用为这些问题提供了一个更加优雅且高效的解决方案。可以把任务发送到线程池中，而由它负责将这些任务分配到线程中。这个池完全负责：

- 创建与销毁线程；
- 分配任务；
- 优化对线程的使用。

这样开发者就从上面这些责任中解脱出来。尽管有这些好处，但是在下面的情况中最好还是不要使用线程池。

- 你的任务必须由优先权系统管理；
- 你的任务需要长时间执行（几秒钟）；
- 你的任务必须在STA（Single Apartment Thread，单套间线程）中处理。事实上，线程池中的线程都属于MTA（Multiple Apartment Thread，多套间线程）类型。这个与COM技术有内在联系的线程套间的概念在8.4.5节有详细介绍。

### 5.10.2　使用线程池

在.NET中，每个进程拥有一个线程池。这意味者ThreadPool类的所有方法都是静态的，因为它们都应用于唯一一个池。.NET Framework使用这个池实现异步调用、异步输入/输出的需求以及定时器。异步调用和定时器都将在本章稍后介绍。

线程池中的最大线程数默认为每个进程25个工作线程以及25个处理异步操作的线程（完成端口线程或I/O线程）。这两个限制可以在任意时刻通过调用静态方法ThreadPool.SetMaxThreads()来修改。如果池中的线程数达到最大将不会再创建出新的线程。如果在所有线程都处于活动状态的时候有额外的任务被提交，它们将进入等待队列直到有线程空闲出来。尽管如此，创建任务的线程并不需要等到任务被处理完毕才能继续执行。可以以下面两种方式使用线程池。

- 使用ThreadPool.QueueUserWorkItem()静态方法提交你自己的任务以及它们的处理方法。一旦任务被提交将无法撤销。
- 创建一个定时器，利用它周期性的向池中提交预先定义好的任务以及它们的处理方法。这种情况下，必须使用ThreadPool.RegisterWaitForSingleObject()这一静态方法。

注意处理方法的引用是通过委托对象来实现的。以上的两种方法都分别存在一个不安全的版本（UnsafeQueueUserWorkItem()和UnsafeRegisterWaitForSingleObject()）。这两个不安全版本的方法允许工作线程与提交任务的线程不必处在同一个安全上下文。使用这些方法可以提升性能，因为在管理安全上下文的时候不再需要检验提交任务的栈。

下面这个例子演示了用户任务（以一个数字作为参数并由ThreadTask()方法处理）以及被周期提交的任务（没有参数并由ThreadTaskWait()方法处理）的使用。这里故意使用长时间的用户任务以迫使线程池创建新的线程。

例5-19

```
using System.Threading;
public class Program {
   public static void Main() {
      ThreadPool.RegisterWaitForSingleObject(
         new AutoResetEvent( false ),
         // Method which represents the periodic task.
         new WaitOrTimerCallback( ThreadTaskWait ),
         null,   // No parameter for the periodic task.
         2000,   // Period = 2 seconds.
         false); // Trigger the periodic task for ever.

      // Post 3 user tasks with parameters 0,1,2.
```

```
        for ( int count = 0; count < 3; ++count )
           ThreadPool.QueueUserWorkItem(new WaitCallback(ThreadTask), count);

        // Wait 12 seconds before shutting down the process.
        // Keep in mind that threadpool threads are background.
        Thread.Sleep( 12000 );
     }
     static void ThreadTask( object obj ) {
        System.Console.WriteLine( "Thread#{0}  Task#{1}  Begin",
                      Thread.CurrentThread.ManagedThreadId , obj.ToString());
        Thread.Sleep( 5000 );
        System.Console.WriteLine( "Thread#{0}  Task#{1}  End",
                      Thread.CurrentThread.ManagedThreadId , obj.ToString());
     }
     static void ThreadTaskWait( object obj, bool signaled ) {
        System.Console.WriteLine("Thread#{0}  TaskWait",
                      Thread.CurrentThread.ManagedThreadId );
     }
  }
```

该程序输出如下（以非确定方式）：

```
Thread#4  Task#0  Begin
Thread#5  Task#1  Begin
Thread#6  Task#2  Begin
Thread#7  TaskWait
Thread#7  TaskWait
Thread#4  Task#0  End
Thread#5  Task#1  End
Thread#6  Task#2  End
Thread#7  TaskWait
Thread#7  TaskWait
```

## 5.11　定时器

很多应用程序都含有需要被周期性执行的任务。例如，你可能想要周期性检验服务器的有效性。首先进入脑中的解决方案就是为该任务创建一个线程，并在两次检验间隔中调用Thread.Sleep()。不过这种实现方法没有很好地发挥线程的作用，因为在两次执行期间线程什么也没有做。我们将展示一些.NET Framework提供的更高效的实现方法。

更具体地说，框架提供了System.Timers.Timer、System.Threading.Timer和System.Windows.Forms.Timer这些类。其中，Forms命名空间下的类必须用于执行周期性的图像任务，比如更新窗体上的数据或是处理基于帧的动画。而在Timers与Threading命名空间下的类的选择则更加细致，它们将由你的需求所决定。

### 5.11.1　System.Timers.Timer 类

System.Timers.Timer在实现中使用线程池中的线程去执行任务。与使用线程池一样，必须在访问任务所用到的资源时，对线程进行同步。

这个类提供了一个double类型的Interval{get;set;}属性，可以通过它设置精度为毫秒的执行时间。Start()与Stop()方法允许开始或停止某个定时器，也可以通过bool类型的Enabled{get;set;}属性改变定时器的活动状态。

一个ElapsedEventHandler类型的委托对象引用代表将被执行的任务的方法。该引用被表示为ElapsedEventHandler类型的Elapsed事件。ElapsedEventHandler委托对象的签名式如下所示。

```
void ElapsedEventHandler(object sender, ElapsedEventArgs e)
```

第一个参数引用了触发任务的定时器。因此，几个定时器可以触发同一个方法而通过该参数来区别它们。此外，由于一个委托对象可以引用多个方法，同一个定时器也就可以在每次被触发后连续调用多个方法。第二个参数包含了定时器被触发的时间，可以通过该对象中类型为DateTime的ElapsedEvenArgs.SignalTime{get;}属性获得该值。下面的程序演示了System.Timers.Timer类的使用：

例5-20

```
using System.Timers;
class Program {
   static Timer Timer1 = new Timer();
   static Timer Timer2 = new Timer();
   static void Main() {
      Timer1.Interval = 1000;  // Period = 1 second.
      Timer1.Elapsed += new ElapsedEventHandler( PeriodicTaskHandler );
      Timer2.Interval = 2000;  // Period = 2 seconds.
      Timer2.Elapsed += new ElapsedEventHandler( PeriodicTaskHandler );
      Timer2.Elapsed += new ElapsedEventHandler( PeriodicTaskHandler );
      Timer1.Start(); Timer2.Start();
      System.Threading.Thread.Sleep( 5000 ); // Sleep 5 seconds.
      Timer1.Stop(); Timer2.Stop();
   }
   static void PeriodicTaskHandler( object sender, ElapsedEventArgs e ) {
      string str = ( sender == Timer1 ) ? “Timer1 “ : “Timer2 “;
      str += e.SignalTime.ToLongTimeString();
      System.Console.WriteLine( str );
   }
}
```

该程序输出如下：

```
Timer1 19:42:49
Timer1 19:42:50
Timer2 19:42:50
Timer2 19:42:50
Timer1 19:42:51
Timer1 19:42:52
Timer2 19:42:52
Timer2 19:42:52
Timer1 19:42:53
```

最后应该知道类型为ISynchronizeInvoke的Timer.SynchronizingObject属性允许指定线程池中的某个线程必须去执行任务。ISynchronizeInvoke接口将在本章的稍后讨论。

### 5.11.2　System.Threading.Timer 类

System.Threading.Timer类和System.Timers.Timer很相似。该类同样使用线程池中的线程去执行任务只不过它不允许你指定使用哪个线程。

另一个区别在于，该类允许你提供一个到期时间。到期时间在定时器被构造的时候指定，它定义了调用回调方法的延迟时间。你可以在任意时刻调用Change()方法修改到期时间。通过将该值设为0，可以立即启动定时器。设为System.Threading.Timer.Infinite常量，可以停止定时器。当不再需要重启定时器时，也可以调用Dispose()方法停止它。

### 5.11.3　System.Windows.Forms.Timer 类

System.Windows.Forms.Timer 类背后的原理与 System.Threading.Timer 类相比更接近于 System.Timers.Timer类。该类的特点在于它启动任务的线程是专属于与任务相关的窗口的。这是因为在它的内部实现中用到了WM_TIMER窗口消息。绝不能使用该类去启动那种执行时间超过几分之一秒的任务，以免造成用户界面无响应。一个使用该类的例子可以在专门介绍Windows窗体的章节中找到，参见18.8.7节。

## 5.12　异步方法调用

理解委托的概念（参见10.13节）对于开始本节内容十分重要。

当一个线程调用方法后，直到方法完成后才继续执行，这样的方法调用被认为是同步的。这种行为比较耗费资源，因为线程在等待方法完成的时间内是阻塞的。如果是对远程对象的调用，那么这段时间可能非常长，因为网络访问将花去几千个时钟周期。但是，如果调用所获取的信息在方法调用后没有立即被用到，那么这种等待是没有必要的。在通常的程序中尤其在分布式体系结构中，经常发生这样的事情：调用一个方法，方法完成一个动作并返回一个仅仅表示该动作是否成功的信息。在这种情况下，程序并不需要立即知道动作是否成功。如果它失败了，我们可以在稍后的时间重启该动作。

我们可以使用异步调用来处理这类情况。我们的想法是线程在调用方法后立即返回而不去等待调用完成。在内部处理的过程中，该调用会立即被线程池中的一个线程接管。过些时候，程序可以获取异步调用的返回信息。这种异步调用技术完全由CLR管理。

我们在上面所描述的机制可以用在你的体系结构中。同样它也被.NET Framework中的很多类所使用，尤其是在以异步方式管理数据流的时候或者在管理对处在另一个AppDomain中的远程对象的异步调用的时候。

### 5.12.1　异步委托

在异步调用期间，不用创建或管理用于执行方法的线程。该线程由之前已经介绍过的线程池管理。

在使用一个异步委托前，应该注意到所有的委托类型自动提供了两个名为BeginInvoke()和EndInvoke()的方法。这两个方法的签名式是由包含它们的委托类型的签名式所决定的。例如，下面这个委托类型：

```
delegate int Deleg( int a, int b );
```

提供了下面两个方法：

```
IAsyncResult BeginInvoke( int a,int b ,AsyncCallback callback, object o );
int EndInvoke( IAsyncResult result );
```

这两个方法由C#编译器产生。尽管如此，Visual Studio的智能感知技术能够从委托类型中推导出它们从而将它们展示出来。

为了以异步方式调用一个方法，必须先使用一个具有同样签名式的委托对象引用该方法。然后，只需要调用该委托对象的BeginInvoke()方法。你可能已经注意到，编译器将确保BeginInvoke()的头几个参数与被调用方法的参数一致。该方法的最后两个类型为AsyncCallback与object的参数将在稍后介绍。

异步调用的返回值可以通过调用EndInvoke()方法获得。这里，同样由编译器保证EndInvoke()

的返回值类型与委托类型的返回值类型一致（在我们的例子中是int类型）。对EndInvoke()的调用是阻塞式的，这意味着该调用只有当异步执行有效结束后它才会返回。

下面的程序演示了对WriteSum()方法的异步调用。

例5-21

```
using System.Threading;
class Program {
   public delegate int Deleg( int a, int b );
   static int WriteSum( int a, int b ) {
      int sum = a + b;
      System.Console.WriteLine( "Thread#{0}: WriteSum() sum = {1}",
                                Thread.CurrentThread.ManagedThreadId, sum);
      return sum;
   }
   static void Main() {
      Deleg proc = WriteSum;
      System.IAsyncResult async = proc.BeginInvoke( 10, 10, null, null );
      // You can do some work here...
      int sum = proc.EndInvoke( async );
      System.Console.WriteLine( "Thread#{0}: Main()     sum = {1}",
                                Thread.CurrentThread.ManagedThreadId, sum);
   }
}
```

该程序输出：

```
Thread 15: WriteSum() Sum = 20
Thread 18: Main()     Sum = 20
```

一个异步调用是由一个实现了System.IAsyncResult接口的类的对象实现的。在本例中，这个底层实现类是System.Runtime.Remoting.Messaging.AsyncResult。BeginInvoke()方法的返回值是一个AsyncResult类型的对象，它将作为EndInvoke()方法的一个输入参数用于标识之前的异步调用。

如果在异步调用中引发了异常，它将自动被CLR截获并保存。CLR将在调用EndInvoke()的时候重新抛出该异常。

### 5.12.2　回调方法

我们可以为异步调用指定一个方法，该方法将在异步调用结束的时候自动被调用。这样的方法被称为*回调方法*。调用回调方法的线程与调用异步方法的线程是同一个线程。

为了使用回调方法，只需要用一个System.AsyncCallback类型的委托对象引用该方法，将该对象作为倒数第二个参数传入到BeginInvoke()方法中。回调方法必须符合委托的类型定义，也就是说它必须以一个IAsyncResult类型的变量作为唯一参数并且返回void类型的值。如下例所示，回调方法必须调用EndInvoke()。

如果线程池中用于处理异步调用的线程是后台线程，那么可能会出现问题。如下例所示，我们必须实现一个事件管理机制，用来确保应用程序不会在异步调用的执行尚未完成的时候就终止了。IAsyncResult接口代表了一个继承于WaitHandle类的同步对象，但是该对象在异步处理完成后就立即被触发，而此时回调方法还未被调用。因此，该对象没有等到回调方法执行完毕。

例5-22

```
using System;
using System.Threading;
using System.Runtime.Remoting.Messaging;
class Program {
   public delegate int Deleg( int a, int b );
   // Initial state of the event: false.
   static AutoResetEvent ev = new AutoResetEvent( false );
   static int WriteSum( int a, int b ) {
      Console.WriteLine( " Thread#{0}: Sum = {1}",
                         Thread.CurrentThread.ManagedThreadId , a + b );
      return a + b;
   }
   static void SumDone(IAsyncResult async) {
      // Wait a second to simulate some work.
      Thread.Sleep( 1000 );
      Deleg proc = ( (AsyncResult) async ).AsyncDelegate as Deleg;
      int sum = proc.EndInvoke( async );
      Console.WriteLine( "Thread#{0}: Callback method sum = {1}",
                         Thread.CurrentThread.ManagedThreadId , sum );
      ev.Set();
   }
   static void Main() {
      Deleg proc = WriteSum;
      // The C#2 compiler infer a delegate object of type AsyncCallback
      // to reference the SumDone() method.
      IAsyncResult async = proc.BeginInvoke( 10, 10, SumDone , null );
      Console.WriteLine(
        "Thread#{0}: BeginInvoke() called! Wait for SumDone() completion.",
        Thread.CurrentThread.ManagedThreadId );
      ev.WaitOne();
      Console.WriteLine(
        "{0}: Bye... ", Thread.CurrentThread.ManagedThreadId );
   }
}
```

该程序输出：

```
Thread#12: BeginInvoke() called! Wait for SumDone() completion.
Thread#14: Sum = 20
Thread#14: Callback method sum = 20
Thread#12: Bye...
```

如果去掉事件机制，该例子输出如下：

```
Thread#12: BeginInvoke() called! Wait for SumDone() completion.
Thread#12: Bye...
```

应用程序没有等待异步处理结束也没有等待回调方法执行。

### 5.12.3　向回调方法传递状态

如果BeginInvoke()方法中的最后一个参数没有被设为空值，那么可以使用它引用一个对象，触发异步调用的线程以及回调方法都可以使用该对象。在IAsyncResult接口中的AsyncState属性包含了该对象的另一个引用。它可以表示一个由回调方法设置的状态。例如，上一节例子中的事件可以被视为一个状态。下面就是前一个例子重写后得到的使用该功能的例子：

例5-23

```
using System;
using System.Threading;
using System.Runtime.Remoting.Messaging;
class Program {
   public delegate int Deleg( int a, int b );
   static int WriteSum( int a, int b ) {
      Console.WriteLine( "Thread#{0}: Sum = {1}",
                         Thread.CurrentThread.ManagedThreadId , a+b);
      return a + b;
   }
   static void SumDone( IAsyncResult async ) {
      // Wait a second to simulate some work.
      Thread.Sleep( 1000 );
      Deleg proc = ( (AsyncResult) async ).AsyncDelegate as Deleg;
      int sum = proc.EndInvoke( async );
      Console.WriteLine( "Thread#{0}: Callback method sum = {1}",
                         Thread.CurrentThread.ManagedThreadId , sum );
      ( (AutoResetEvent)async.AsyncState ).Set();
   }
   static void Main() {
      Deleg proc = WriteSum;
      AutoResetEvent ev = new AutoResetEvent(false);
      IAsyncResult async = proc.BeginInvoke( 10, 10, SumDone, ev );
      Console.WriteLine(
        "Thread#{0}: BeginInvoke() called! Wait for SumDone() completion.",
        Thread.CurrentThread.ManagedThreadId );
      ev.WaitOne();
      Console.WriteLine(
        "{0}: Bye... ", Thread.CurrentThread.ManagedThreadId );
   }
}
```

### 5.12.4　One-Way 调用

可以将`System.Runtime.Remoting.Messaging.OneWayAttribute`应用于任何一个方法，无论该方法是否是静态的。这个attribute会告诉CLR该方法不返回任何信息。即便某个方法返回一些数据（通过返回值或使用由`out`或`ref`关键字定义的参数），但是一旦它被标记了该attribute，那么它就不会再返回任何东西。

一个被标记了`OneWayAttribute`的方法既可以以同步的方式调用，也可以以异步的方式调用。如果一个方法被标记了`OneWayAttribute`，并且在它的执行过程中引起了一个异常却没有被捕获，那么在同步方式下，该异常会向上传播。但是如果是在异步方式下，该异常将不会被传播。在大多数情况下，被标记为one-way的方法是以异步方式工作的。

One-way的异步调用通常用于完成一些辅助性的任务，这些任务的成功与否对应用程序的正常运行不会产生影响。大多数情况下，我们使用它们在应用程序的执行中交流一些信息。

## 5.13　线程–资源亲缘性

使用线程与资源之间亲缘性（affinity）的概念，可以极大地简化对资源访问的同步工作。它的主要思想是，总使用同一个线程去访问某个资源。由于该资源不会被共享，也就不再需要保护它免受并发访问的不良影响。.NET Framework提供了几种机制用于实现亲缘性的概念。

### 5.13.1　`System.ThreadStaticAttribute`

默认情况下，一个静态字段将被一个进程中的所有线程所共享。这种特性迫使开发者去同步所有对这种字段的访问。对静态字段应用System.ThreadStaticAttribute，可以强迫CLR为进程中的每个线程创建一个该静态字段的实例。因此，使用这种机制是一种在线程和资源间实现亲缘性概念的好方法。

最好避免在声明静态字段的时候直接对它进行初始化。事实上，在这种情况下，只有载入该类的线程会完成它自己版本的字段的初始化。下面的程序说明了这种特性：

例5-24

```
using System.Threading;
class Program {
   [System.ThreadStatic]
   static string str = “Initial value “;
   static void DisplayStr() {
      System.Console.WriteLine(“Thread#{0} Str={1}”,
                               Thread.CurrentThread.ManagedThreadId , str);
   }
   static void ThreadProc() {
      DisplayStr();
      str = “ThreadProc value”;
      DisplayStr();
   }
   static void Main() {
      DisplayStr();
      Thread thread = new Thread( ThreadProc );
      thread.Start();
      thread.Join();
      DisplayStr();
   }
}
```

该程序输出如下：

```
Thread#1 Str=Initial value
Thread#2 Str=
Thread#2 Str=ThreadProc value
Thread#1 Str=Initial value
```

### 5.13.2　线程本地存储（TLS）

线程与资源之间的亲缘性的概念也可以通过线程本地存储（常称之为TLS）的概念来实现。这并不是什么新概念，它存在于Win32的级别。事实上，.NET Framework在内部也是使用的这个实现。

TLS中用到了数据槽的概念。一个数据槽是System.LocalDataStoreSlot类的一个实例。一个数据槽也可以被视为一个对象数组。该数组的大小总是与当前进程中的线程数相当。因此，每个线程都在数据槽中拥有自己的位置。该位置对其他线程来说是不可见的。它可以用于引用一个对象。CLR负责为每一个数据槽建立线程与对象之间的对应关系。Thread类提供了下面两个方法用于读写存储在数据槽中的数据。

```
static public object GetData( LocalDataStoreSlot slot );
static public void SetData( LocalDataStoreSlot slot, object obj );
```

#### 1. 有名数据槽

可以命名一个数据槽用于以后对它进行标识。Thread类提供了以下方法用于创建、获取或销毁一个有名数据槽：

```
static public LocalDataStoreSlot AllocateNamedDataSlot( string slotName );
static public LocalDataStoreSlot GetNamedDataSlot( string slotName );
static public void FreeNamedDataSlot( string slotName );
```

垃圾回收器不会销毁有名数据槽。这个工作由程序员自己负责。

下面的程序使用了一个有名数据槽为进程中的每个线程提供一个计数器。该计数器每调用一次fServer()方法就增加一次。该程序利用TLS使得fServer()方法不用将对计数器的引用作为其参数。另一个好处在于开发者不需要为每个线程维护一个计数器。

例5-25

```
using System;
using System.Threading;
class Program {
   static readonly int NTHREAD = 3;    // 3 threads to create.
   // 2 calls to fServer() for each thread created.
   static readonly int MAXCALL = 2;
   static readonly int PERIOD = 1000; // 1 second between calls.
   static bool fServer() {
      LocalDataStoreSlot dSlot = Thread.GetNamedDataSlot( “Counter” );
      int counter = (int) Thread.GetData( dSlot );
      counter++;
      Thread.SetData( dSlot, counter );
      return !( counter == MAXCALL );
   }
   static void ThreadProc() {
      LocalDataStoreSlot dSlot = Thread.GetNamedDataSlot( “Counter” );
      Thread.SetData( dSlot, (int) 0 );
      do{
         Thread.Sleep( PERIOD );
         Console.WriteLine(
            “Thread#{0} I’ve called fServer(), Counter = {1}”,
            Thread.CurrentThread.ManagedThreadId ,
            (int)Thread.GetData(dSlot));
      } while ( fServer() );
      Console.WriteLine(“Thread#{0} bye”,
         Thread.CurrentThread.ManagedThreadId );
   }
   static void Main() {
      Console.WriteLine( “Thread#{0} I’m the main thread, hello world”,
                         Thread.CurrentThread.ManagedThreadId );
      Thread.AllocateNamedDataSlot( “Counter” );
      Thread thread;
      for ( int i = 0; i < NTHREAD; i++ ) {
         thread = new Thread( ThreadProc );
         thread.Start();
      }
      // We don’t implement a mechanism to wait for child threads
      // completion, thus, we are waiting for a while.
      Thread.Sleep( PERIOD * (MAXCALL + 1) );
      Thread.FreeNamedDataSlot( “Counter” );
      Console.WriteLine( “Thread#{0} I’m the main thread, bye.”,
                         Thread.CurrentThread.ManagedThreadId );
   }
}
```

该程序输出如下：

```
Thread#1 I'm the main thread, hello world
Thread#3 I've called fServer(), Counter = 0
Thread#4 I've called fServer(), Counter = 0
Thread#5 I've called fServer(), Counter = 0
Thread#3 I've called fServer(), Counter = 1
Thread#3 bye
Thread#4 I've called fServer(), Counter = 1
Thread#4 bye
Thread#5 I've called fServer(), Counter = 1
Thread#5 bye
Thread#1 I'm the main thread, bye
```

2. 匿名数据槽

可以调用Thread类的静态方法AllocateDataSlot()来创建一个匿名数据槽。开发者不需要负责匿名数据槽的销毁。但是必须确保LocalDataStoreSlot类的一个实例对所有的线程都可见。下面是用匿名数据槽的概念重写前一个程序的代码。

例5-26

```
using System;
using System.Threading;
class Program {
   static readonly int NTHREAD = 3;    // 3 threads to create.
   // 2 calls to fServer() for each thread created.
   static readonly int MAXCALL = 2;
   static readonly int PERIOD = 1000; // 1 second between calls.
   static LocalDataStoreSlot dSlot;
   static bool fServer() {
      int counter = (int) Thread.GetData( dSlot );
      counter++;
      Thread.SetData( dSlot, counter );
      return !( counter == MAXCALL );
   }
   static void ThreadProc() {
      Thread.SetData( dSlot, (int) 0 );
      do{
         Thread.Sleep(PERIOD);
         Console.WriteLine(
            "Thread#{0} I've called fServer(), Counter = {1}",
            Thread.CurrentThread.ManagedThreadId ,
            (int)Thread.GetData(dSlot));
      } while ( fServer() );
      Console.WriteLine( "Thread#{0} bye",
         Thread.CurrentThread.ManagedThreadId );
   }
   static void Main() {
      Console.WriteLine( "Thread#{0} I'm the main thread, hello world",
                         Thread.CurrentThread.ManagedThreadId );
      dSlot = Thread.AllocateDataSlot();
      for ( int i = 0; i < NTHREAD; i++ ) {
         Thread thread = new Thread( ThreadProc );
         thread.Start();
      }
      Thread.Sleep( PERIOD * (MAXCALL + 1) );
```

```
        Console.WriteLine( "Thread#{0} I'm the main thread, bye.",
            Thread.CurrentThread.ManagedThreadId );
    }
}
```

### 5.13.3　System.ComponentModel.ISynchronizeInvoke 接口

System.ComponentModel.ISynchronizeInvoke接口定义如下所示。

```
public object System.ComponentModel.ISynchronizeInvoke {
        public object       Invoke( Delegate method, object[] args );
        public IAsyncResult BeginInvoke( Delegate method, object[] args );
        public object       EndInvoke( IAsyncResult result );
        public bool         InvokeRequired{ get; }
}
```

该接口的一个实现将保证某些方法总是被同一个线程以同步或异步的方式执行。

- 在同步方式中，一个线程T1调用对象OBJ的方法M()。事实上，T1通过指定一个引用OBJ.M()的委托对象以及一个含有其参数的数组来调用ISynchronizeInvoke.Invoke()方法。另一个线程T2去执行OBJ.M()方法。T1等待调用结束并在调用返回后获取相应的信息。
- 异步方式与同步方式的不同之处在于T1将调用ISynchronizeInvoke.BeginInvoke()方法。当T2执行OBJ.M()方法的时候T1将不再处于阻塞状态。当T1需要调用所返回的信息时，它会调用ISynchronizeInvoke.EndInvoke()方法，此时如果T2完成了OBJ.M()的执行，该方法将提供这一信息。

ISynchronizeInvoke接口主要被框架用于强制Windows Form技术使用专用于窗体的线程去执行窗体的方法。这个限制是来自于Windows Form技术构建在Windows消息管道之上这一事实。同类型的问题在System.ComponentModel.BackgroundWorker类中也有处理，参见18.3.4节。

可以从Juval Lowy提供的实现**ISynchronizeInvoke**的例子中得到启发，去开发你自己的ISynchronizeInvoke接口的实现。该例子可以从URL中找到。http://docs.msdnaa.net/ark_new3.0/cd3/content/Tech_System%20Programming.htm.

## 5.14　执行上下文简介

.NET 2 Framework提供了一些新类用于捕获当前线程的执行上下文，并将该上下文传播到另一个线程。

- System.Security.SecurityContext
  该类的一个实例包含了底层Windows用户的身份信息以及线程栈的状态。其中身份信息以一个System.Security.Principal.WindowsIdentity类的实例的形式表示，而线程栈的状态则通过一个System.Threading.CompressedStack类的实例表现。线程栈的状态主要用于CAS机制遍历栈。
- System.Threading.SynchronizationContext，能够针对不同的同步模型自动处理某些兼容性约束。
- System.Threading.HostExecutionContext，能够在当前线程的执行上下文中访问运行时宿主。
- System.Runtime.Remoting.Messaging.LogicalCallContext，.NET Remoting能够利用该类的实例传播.NET Remoting的上下文。关于此话题的更多信息参见15.6节。
- System.Threading.ExecutionContext，该类型的一个实例包含了以上所提到的上下文的并集。

重新以例6-11为例，这次我们通过在当前线程上模拟Windows用户guest来修改安全上下文。

例5-27

```
...
   static void Main() {
      System.IntPtr pToken;
      if( LogonUser(
            "guest" ,     // login
            string.Empty, // Windows domain
            "guest_pwd" , // password
            2,            // LOGON32_LOGON_INTERACTIVE
            0,            // LOGON32_PROVIDER_DEFAUT
            out pToken)) {
         WindowsIdentity.Impersonate( pToken );
         DisplayContext( "Main" );
         ThreadPool.QueueUserWorkItem( Callback, null );
         CloseHandle( pToken );
      }
   }
   static void Callback(object o) { DisplayContext("Callback"); }
   static void DisplayContext( string s ) {
      System.Console.WriteLine( s+" Thread#{0} Current user is {1}",
         Thread.CurrentThread.ManagedThreadId,
         WindowsIdentity.GetCurrent().Name );
   }
...
```

该程序输出如下：

```
Main Thread#1 Current user is PSMACCHIA\guest
Callback Thread#3 Current user is PSMACCHIA\guest
```

在.NET 1.1中，该程序输出如下：

```
Main Thread#1 Current user is PSMACCHIA\guest
Callback Thread#3 Current user is PSMACCHIA\pat
```

事实上，在.NET 2中线程池在默认情况下会把提交任务的线程的上下文传递给执行任务的线程。不过在.NET 1中并不是这样，使用ExecutionContext.SuppressFlow()方法可以在.NET 2中重现.NET 1的功能。

例5-28

```
...
         DisplayContext( "Main" );
         ExecutionContext.SuppressFlow();
         ThreadPool.QueueUserWorkItem( Callback, null );
...
```

该示例输出如下：

```
Main Thread#1 Current user is PSMACCHIA\guest
Callback Thread#3 Current user is PSMACCHIA\pat
```

下面的例子展示了如何通过程序传播执行上下文。首先，使用ExecutionContext.Capture()捕获当前上下文。然后，为其创建一个拷贝并传递给从线程池中请求获得的线程。该线程通过调用ExecutionContext.Run()方法传播上下文。此方法以一个执行上下文和委托对象作为参数，在设置

好当前线程的上下文后，它将在当前线程上调用委托对象所引用的方法。

例5-29

```
...
   static void Main() {
      ...
         WindowsIdentity.Impersonate( pToken );
         DisplayContext( "Main" );
         ExecutionContext ctx = ExecutionContext.Capture();
         ExecutionContext.SuppressFlow();
         ThreadPool.QueueUserWorkItem(
            SetContextAndThenCallback, ctx.CreateCopy() );
         CloseHandle( pToken );
      }
   }
   static void SetContextAndThenCallback( object o ) {
      ExecutionContext ctx = o as ExecutionContext;
      ExecutionContext.Run( ctx, Callback, null );
   }
   static void Callback( object o ) {
      DisplayContext( "Callback" );
   }
...
```

不出所料，该示例输出如下：

```
Main Thread#1 Current user is PSMACCHIA\guest
Callback Thread#3 Current user is PSMACCHIA\guest
```