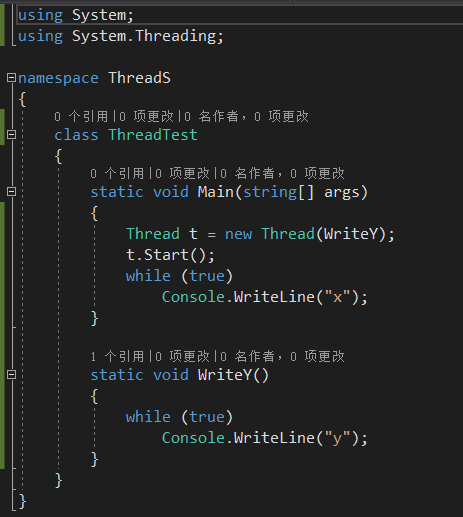
1. 概述与概念

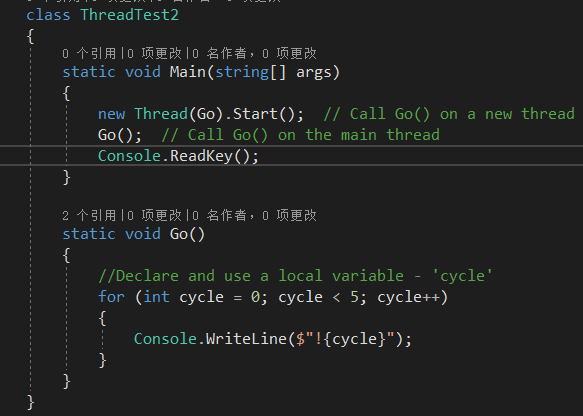
C#支持通过多线程并行地执行代码，一个线程有它独立的执行路径，能够与其他的线程同时地运行。一个C#程序开始于一个单线程，这个单线程是被CLR和操作系统（也称“主线程”）自动创建的，并具有多线程创建额外的线程。

简单例子如下：

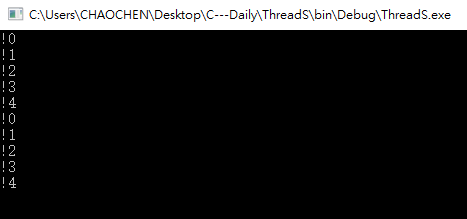


主线程创建了一个新线程“t”，它运行了一个重复打印字母“y”的方法，同时主线程重复打印字母“x”.**CLR分配每个线程到它自己的内存堆栈上，来保证局部变量的分离运行**。

接下来我们定义一个局部变量，然后在主线程和新建的线程上同时调用这个方法。

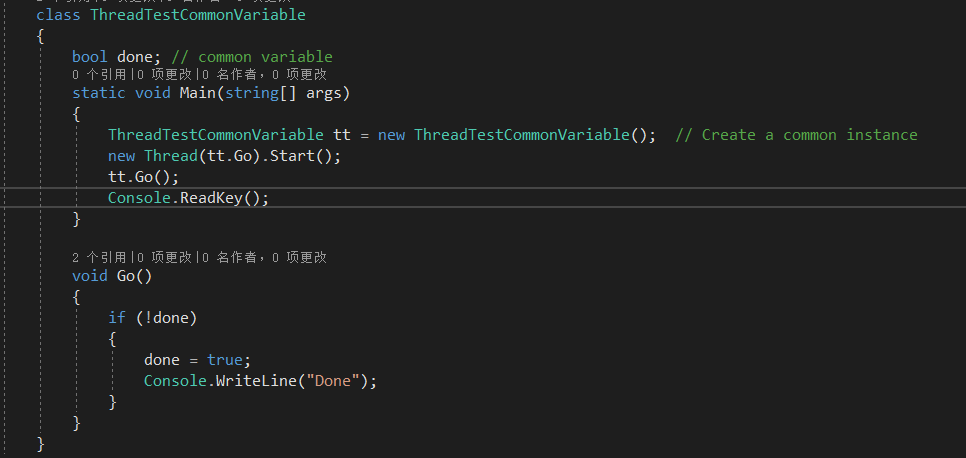


输出如下：

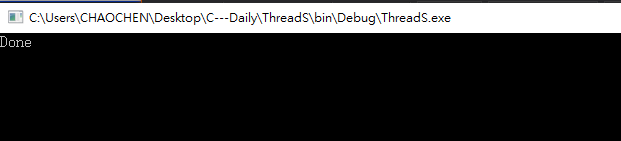


由上可知局部变量cycle的副本在各自的内存堆栈中创建，输出也一样。

当线程们引用一些公用的目标**实例**时，它们会共享数据

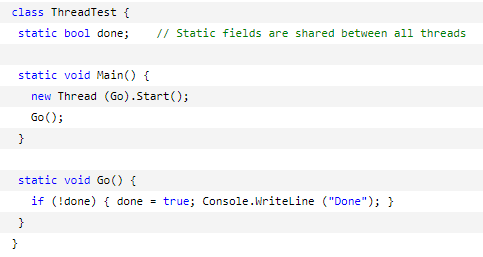


运行结果显示只有一个Done打印出来



由此可以看到这两个线程都调用了Go()方法，但是只打印出一个Done，这就说明其中一个线程修改了done值后，另一个线程继续使用了这个值，即两个线程共享了这个变量。

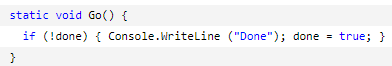
静态字段提供了另一种在线程间共享数据的方式，下面以done为静态字段的例子：



**总结**：

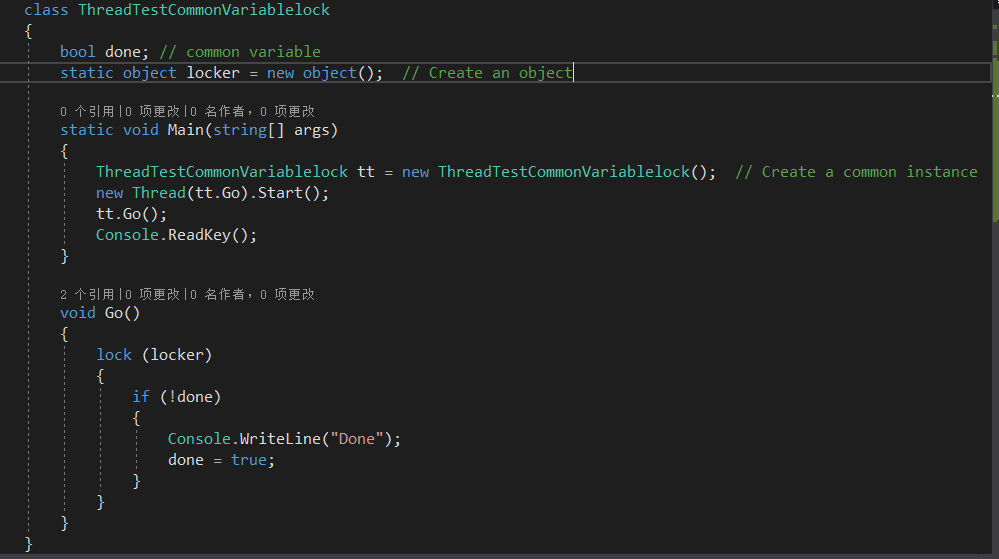
上面两个实例基本类似，done都是class中的全局变量，只不过一个是静态的一个是非静态的的。静态的就用静态方法调用，非静态的就用实例方法调用。

另外，上面两个例子还说明了一个概念，那就是多线程的线程安全输出实际上是不确定的：它存在一种可能性，就是“Done”会被打印两次。。如果我们调换指令的顺序，“Done”被打印两次的机会会大幅提升。



出现的原因就是一个线程在判断if块的时候，正好另一个线程正在执行Writeline语句（在它将done设置为true之前）。（两个线程的执行没有先后顺序）

补救措施是当读写公共字段的时候，提供一个排它锁（C#中的lock语句）

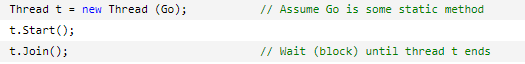


当两个线程争夺一个锁的时候（locker），一个线程等待，直到那个锁打开。在这种情况下，就保证了同一时刻只有一个线程能获得内存空间，完成运行。代码在如此不确定的多线程环境中运行被称为线程安全。

临时暂停，或阻止 是多线程的协同工作，同步活动的本质特征。等待一个排它锁被释放是一个线程被阻止的原因，另一个原因是线程想要暂停或sleep一段时间。

Thread.Sleep(TimeSpan.FromSeconds(30)); // Block for 30 seconds

一个线程也可以使用它的join方法来等待另一个线程结束：



一个线程一旦被阻止，就不再消耗CPU资源了。

线程是如何工作的

线程被一个线程协调程序管理着——一个CLR委托给操作系统的函数。线程协调程序确保将所有活动的线程被分配适当的执行时间；并且那些等待或阻止的线程——比如说在排它锁中、或在用户输入——都是不消耗CPU时间的。

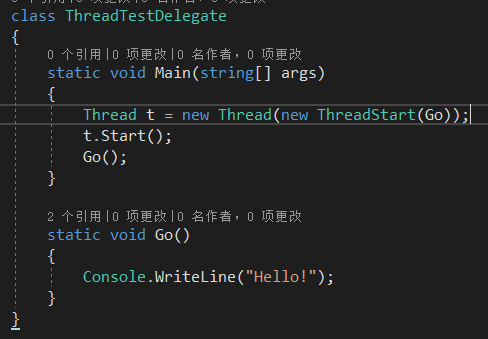
在单核处理器的电脑中，线程协调程序完成一个时间片之后迅速地在活动的线程之间进行切换执行。这就导致“波涛汹涌”的行为，例如在第一次例子中，每次重复x和y。

在多核的电脑中，多线程被实现成混合时间片和真实的并发——不同的线程在不同的CPU上运行。

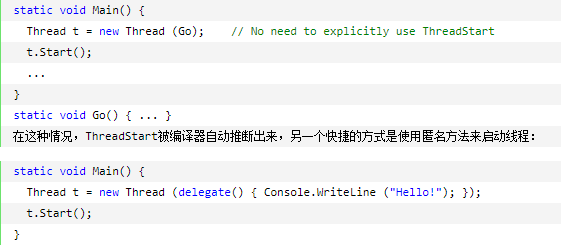
创建和开始使用多线程

所有线程都用Thread类来创建，通过ThreadStart委托来指明方法从哪里开始运行，

ThreadStart存在于System.Threading.ThreadStart



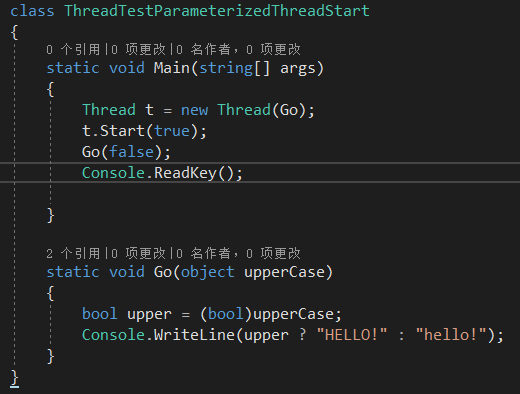
一个线程可以通过C#堆委托简短的语法更便利地创建出来：



线程有一个isAlive属性，在调用Start()之后直到线程结束之前一直为true。一个线程一旦结束便不能重新开始了。

**将数据传入ThreadStart中**

在上面的例子中，我们想更好的区分每个线程的输出结果，让其中一个线程输出大写字母。我们传入一个状态到Go中来实现，但我们不能使用**Threadstart委托，因为他不能接收参数**，但是.NET Framework定义了另一个版本的委托叫做**ParameterizedThreadStart，它可以接收一个单独的object类型参数**。

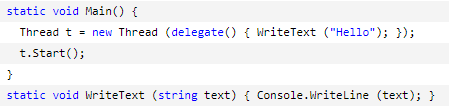


在整个例子中，编译器**自动**推断出ParameterizedThreadStart委托，因为Go方法接受一个单独的object参数。



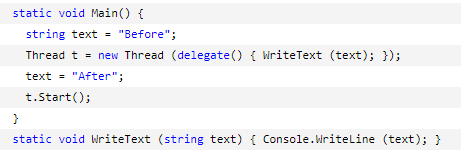
ParameizedThreadStart特性是在使用之前我们必须对我们想要的类型进行装箱操作（只接受object类型），并且只能接收一个参数。

一个替代方案是使用一个匿名方法调用一个普通的方法：



在匿名方法中调用了我们需要运行的方法。就可以不使用ParameterizedThreadStart。

这个方案的优点是目标方法（这里是WriteText），可以接收任意数量的参数，并且没有装箱操作。但是这个被传入的参数可以是外部变量放入到匿名。如下所示：

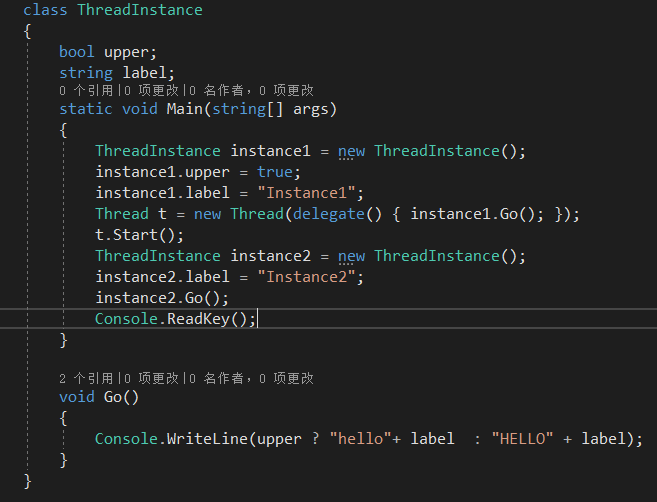


输出的结果是：After

匿名方法打开了一种怪异的现象，当外部变量被后来的部分修改了值的时候，可能会透过外部变量进行无意的互动。

一旦线程开始运行，外部变量最好被处理成只读的——除非有人愿意使用适当的锁。

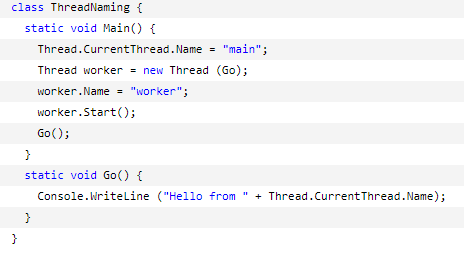
另一种较常见的方式是将对象实例的方法而 不是静态方法传入到线程中，对象实例的属性可以告诉线程要做什么：



命名线程：

线程可以通过它的name属性进行命名，这非常有利于调试：可以用console.writeline打印出线程的名字，VS可以将线程的名字显示在调试工具栏的位置上。线程的名字可以在被任何时间设置——但是只能设置一次，重命名会引发异常。

程序的主线程也可以被命名，主线程通过CurrentThread命名：



**前台线程和后台线程**：

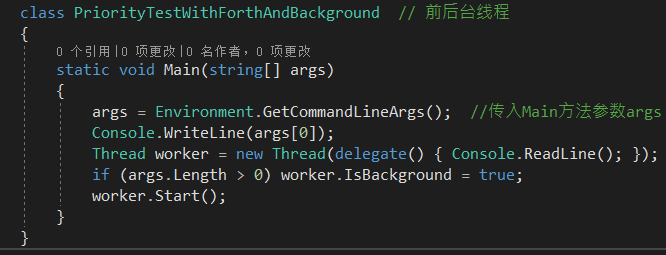
线程默认为前台线程，这意味着任何前台线程在运行时都会保持程序的存活。C#也支持后台线程，当所有前台线程结束后，它们不能维持程序的存活。

前台线程：当它结束时，整个程序就结束了。

后台线程：后台程序不能维持程序的存活，即使后台线程仍在运行，只要前台结束，整个程序就结束了。

以任何方式将线程从前台改变为后台都**不会改变它在CPU协调程序中的优先级和状态**。

线程的**isBackground**属性控制它的前后台状态。



如果程序被调用的时候没有任何参数，工作线程为前台线程，并且等待ReadLine语句来等待用户触发回车，这期间，主线程退出，但是程序保持运行，因为一个前台线程仍然存活。

当有参数传入Main（），工作线程被赋值为后台线程，当主线程结束程序立即退出，终止了ReadLine。

后台线程的这种特性，使任何最后操作都被规避了。好的方式是明确等待任何后台工作线程完成后再结束程序，大多用ThreadJoin。如果因为某种原因某个工作线程无法完成，可以试图终止它的对应工作，如果失败了，则抛弃整个线程，允许它与进程一起消亡。

拥有一个后台工作线程是有益的。最直接的理由是当提到结束程序**它总是可能有最后的发言权**。交织以不会消亡的前台线程，保证程序的正常退出。

一般前台线程用于需要长时间等待的任务。后台线程一般用于处理时间较短的任务。比如处理客户端发过来的请求信息。

一般主线程和使用Tread构造的线程默认为前台线程

线程池线程也就是使用ThreadPool.QueueUserWorkItem()和Task工厂创建的线程都默认为后台线程。

对于程序失败退出的普遍原因是存在“被忘记”的前台线程。

抛弃一个前台线程是十分危险的，尤其是对Windows Form程序，因为程序直到主线程结束时才退出（对于用户是这样的），但是它的进程仍然运行着。在任务管理器它将从应用程序栏消失不见，但却可以在进程栏中找到它。除非用户找到并结束它，他将继续消耗资源。

线程优先级：

线程的priority属性确定了线程相对于其他同一进程的活动线程拥有多少**执行时间**。

以下是级别：



只有当多个同一个进程中的线程同时活动时，优先级才有用。

设置一个线程的优先级为高一些，并不意味着他能执行实时的工作（实时工作一般都是最高优先级，需要最先进行处理），因为它受限于程序的进程的级别。要执行实时的工作，必须提升到System.Diagnostics命名空间下Process的级别。如下所示：

设置当前进程的优先级为high。

ProcessPriorityClass.High其实是一个短暂缺口的过程中的最高优先级别.

RealTime.

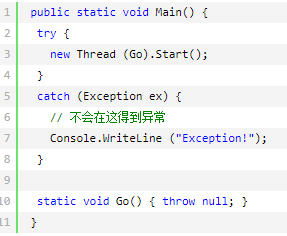
当设置进程级别到RealTime通知操作系统:你不想你的进程被抢占了.如果你的程序进入一个偶然的死循环,那么整个操作系统会被锁住.除了关机已经没有别的方法了.这样看来这种级别弊端太明显,所以High大体上被认为最高的有用进程级别.

如果一个实时的程序有一个用户界面,提升进程的级别是不太好的,因为当用户界面UI过于复杂的时候,界面的更新耗费过多的CPU时间,拖慢了整台电脑.降低主线程的级别、提升进程的级别、确保实时线程不进行界面刷新，但这样并不能避免电脑越来越慢。因为操作系统仍会拨出过多的CPU给整个进程。最理想的方案是使实时工作和用户界面在不同的进程运行，通过Remoting或共享内存方式进行通信，共享内存需要Win32 API中的P/Invoking。

异常处理

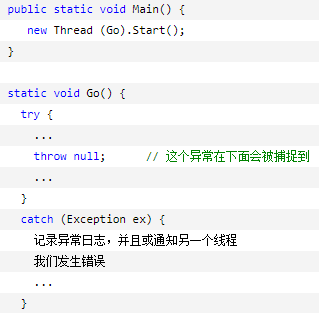
任何线程创建范围内，try\catch\finally块中当线程开始执行便不再与其有任何关系。

如下例所示：



这里的try\catch根本不会作用,但新创建的线程确实引发了NullReferenceException异常.

补救方法如下所示:



从.NET 2.0开始,任何线程内的未处理的异常都将导致整个程序关闭,这意味着忽略异常不再是一个选项了.因此为了避免由于未处理异常而引起的程序崩溃,try\catch块需要出现在每个线程进入的方法内。

但是

对于经常使用“全局”异常处理的Windows Forms 程序员来说，这可能有点麻烦（针对线程）。像下面这样：



Application.ThreadException事件在异常被抛出时触发，以一个Windows信息的方法。简而言之，一个WindowsForms程序几乎所有代码都可以被触发。虽然看起来很完美，它使人产生一种虚假的安全感——所有的异常都被中央异常处理捕捉到了。由工作线程抛出的异常便是一个没有被Application.ThreadException捕捉到的很好的例子。

.NET framework为全局异常处理提供了一个更低级别的事件:APPDomain.UnhandledException,这个事件在任何类型的程序(不管有没有用户界面)的任何线程有未处理的异常都会被触发.尽管它提供了好的不得了的异常处理解决机制,但是这不意味着这程序不崩溃,也不意味着能取消.NET异常对话框.