**WinForm多线程编程与Control.Invoke的应用浅谈**

2012-10-22 20:03 来源：博客园 作者：guozhijian 字号：T|T

[**摘要**]本文介绍在WinForm开发中如何使用多线程，以及在线程中如何通过Control.Invoke方法返回窗体主线程执行相关操作，并提供详细的示例代码供参考。

在WinForm开发中，我们通常不希望当窗体上点了某个按钮执行某个业务的时候，窗体就被卡死了，直到该业务执行完毕后才缓过来。一个最直接的方法便是使用多线程。多线程编程的方式在WinForm开发中必不可少。

本文介绍在WinForm开发中如何使用多线程，以及在线程中如何通过Control.Invoke方法返回窗体主线程执行相关操作。

**－. WinForm多线程编程**

**1. new Thread()**

新开一个线程，执行一个方法，没有参数传递：

[View Row Code](http://dotnet.9sssd.com/winform/art/850)

|  |
| --- |
| private void DoWork() { |
| Thread t = new Thread(new ThreadStart(this.DoSomething)); |
| t.Start(); |
| } |
| private void DoSomething() { |
| MessageBox.Show("thread start"); |
| } |

新开一个线程，执行一个方法，并传递参数：

[View Row Code](http://dotnet.9sssd.com/winform/art/850)

|  |
| --- |
| private void DoWork() { |
| Thread t = new Thread(new ParameterizedThreadStart(this.DoSomething)); |
| t.Start("guozhijian"); |
| } |
| private void DoSomething(object o) { |
| MessageBox.Show(o.ToString()); |
| } |

参数定义为object类型。

带参数的

**2. ThreadPool**

众所周知，新开一个线程代价是很高昂的，如果我们每个操作都新开一个线程，那么太浪费了，于是，下面使用线程池。

无参数传递：

[View Row Code](http://dotnet.9sssd.com/winform/art/850)

|  |
| --- |
| private void DoWork() { |
| ThreadPool.QueueUserWorkItem(new WaitCallback(this.DoSomething)); |
| } |
| private void DoSomething(object o) { |
| MessageBox.Show("thread start"); |
| } |

有参数传递：

[View Row Code](http://dotnet.9sssd.com/winform/art/850)

|  |
| --- |
| private void DoWork() { |
| ThreadPool.QueueUserWorkItem(new WaitCallback(this.DoSomething), "guozhijian"); |
| } |
| private void DoSomething(object o) { |
| MessageBox.Show(o.ToString()); |
| } |

使用匿名方法更灵活：

[View Row Code](http://dotnet.9sssd.com/winform/art/850)

|  |
| --- |
| private void DoWork() { |
| string name = "guozhijian"; |
| ThreadPool.QueueUserWorkItem(new WaitCallback(delegate(object o){ |
| MessageBox.Show(name); |
| })); |
| } |

在匿名代码段里面可以直接访问局部变量，不用在关心参数传递的问题

**二. Invoke**

**1. this.Invoke**

现在，在业务线程里面执行完毕，要改变窗体控件的值了，此时，如果直接通过this得到控件的句柄，然后对它进行操作是会抛异常的，.Net WinForm Application里面是不允许这样的操作的。这是，可以调用Invoke方法。

**2.Invoke方法签名：**

[View Row Code](http://dotnet.9sssd.com/winform/art/850)

|  |
| --- |
| object Control.Invoke(Delegate Method) |
| object Control.Invoke(Delegate Method, params object[] args) |

**3.使用自定义委托**

[View Row Code](http://dotnet.9sssd.com/winform/art/850)

|  |
| --- |
| private void DoWork() { |
| WaitCallback wc = new WaitCallback(this.DoSomething); |
| ThreadPool.QueueUserWorkItem(wc, "Guozhijian"); |
| } |
|  |
| private delegate void MyInvokeDelegate(string name); |
| private void DoSomething(object o) { |
| this.Invoke(new MyInvokeDelegate(this.ChangeText), o.ToString()); |
| } |
|  |
| private void ChangeText(string name) { |
| this.textBox1.Text = name; |
| } |

哦，太麻烦了，难道我每次都要定义一个委托啊，这样可不行。

**4.使用System.Action：**

[View Row Code](http://dotnet.9sssd.com/winform/art/850)

|  |
| --- |
| private void DoWork() { |
| WaitCallback wc = new WaitCallback(this.DoSomething); |
| ThreadPool.QueueUserWorkItem(wc, "Guozhijian"); |
| } |
|  |
| private void DoSomething(object o) { |
| this.Invoke(new Action<string>(this.ChangeText), o.ToString()); |
| } |
|  |
| private void ChangeText(string name) { |
| this.textBox1.Text = name; |
| } |

本例传递一个参数，System.Action有很多个重载，可以无参数（非泛型），而最多可以有四个参数，同样采用匿名方法，不使用泛型形式的System.Action，如下：

[View Row Code](http://dotnet.9sssd.com/winform/art/850)

|  |
| --- |
| private void DoWork() { |
| WaitCallback wc = new WaitCallback(this.DoSomething); |
| ThreadPool.QueueUserWorkItem(wc, "Guozhijian"); |
| } |
|  |
| private void DoSomething(object o) { |
| this.Invoke(new Action(delegate() { |
| this.textBox1.Text = o.ToString(); |
| })); |
| } |

**5.使用System.Func**

如果Invoke调用主窗体操作之后，还希望在调用完得到一个返回值：

[View Row Code](http://dotnet.9sssd.com/winform/art/850)

|  |
| --- |
| private void DoWork() { |
| WaitCallback wc = new WaitCallback(this.DoSomething); |
| ThreadPool.QueueUserWorkItem(wc, "Guozhijian"); |
| } |
|  |
| private void DoSomething(object o) { |
| System.Func<string, int> f = new Func<string, int>(this.GetId); |
| object result = this.Invoke(f,o.ToString()); |
| MessageBox.Show(result.ToString()); |
| } |
|  |
| private int GetId(string name) { |
| this.textBox1.Text = name; |
| if (name == "Guozhijian") { |
| return 999; |
| } |
| else { |
| return 0; |
| } |
| } |

result的值为 999。

System.Func同样有很多泛形重载，这里不赘述。

**6.关于Invoke的拥有者：Control**

本文例中都是用this来引用，这里this替换为窗体任何一个控件的句柄都是OK的，因为Control.Invoke含义是将方法委托给拥有该Control的线程去执行。

**C# WinForm多线程开发（一） Thread类库**

2012-10-22 20:11 来源：博客园 作者：joechen 字号：T|T

[**摘要**]本文介绍C# WinForm多线程开发之Thread类库，并提供简单的示例代码供参考。

Windows是一个多任务的系统，如果你使用的是windows 2000及其以上版本，你可以通过任务管理器查看当前系统运行的程序和进程。什么是进程呢？当一个程序开始运行时，它就是一个进程，进程所指包括运行中的 程序和程序所使用到的内存和系统资源。而一个进程又是由多个线程所组成的，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。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。

**一 关于Thread的说明**

在.net framework class library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。其中提供Thread类用于创建线 程，ThreadPool类用于管理线程池等等，此外还提供解决了线程执行安排，死锁，线程间通讯等实际问题的机制。如果你想在你的应用程序中使用多线 程，就必须包含这个类。Thread类有几个至关重要的方法，描述如下：

* **Start()**：启动线程
* **Sleep(int)：**静态方法，暂停当前线程指定的毫秒数
* **Abort()：**通常使用该方法来终止一个线程
* **Suspend()：**该方法并不终止未完成的线程，它仅仅挂起线程，以后还可恢复。
* **Resume()：**恢复被Suspend()方法挂起的线程的执行

线 程入口使程序知道该让这个线程干什么事，在C#中，线程入口是通过ThreadStart代理（delegate）来提供的，你可以把 ThreadStart理解为一个函数指针，指向线程要执行的函数，当调用 Thread.Start()方法后，线程就开始执行ThreadStart所代表或者说指向的函数。 ThreadState在各种情况下的可能取值如下：

* **Aborted：**线程已停止
* **AbortRequested：**线程的Thread.Abort()方法已被调用，但是线程还未停止
* **Background：**线程在后台执行，与属性Thread.IsBackground有关
* **Running：**线程正在正常运行
* **Stopped：**线程已经被停止
* **StopRequested：**线程正在被要求停止
* **Suspended：**线程已经被挂起（此状态下，可以通过调用Resume()方法重新运行）
* **SuspendRequested：**线程正在要求被挂起，但是未来得及响应
* **Unstarted：**未调用Thread.Start()开始线程的运行
* **WaitSleepJoin：**线程因为调用了Wait(),Sleep()或Join()等方法处于封锁状态

**二 Winform中使用的thread**

首先可以看看最直接的方法，也是.net 1.0下支持的方法。但请注意的是，此方法在.net 2.0以后就已经是一种错误的方法了。

[View Row Code](http://dotnet.9sssd.com/winform/art/851)

|  |
| --- |
| public partial class Form1 : Form |
| { |
| public Form1() |
| { |
| InitializeComponent(); |
| } |
| private void Form1\_Load(object sender, EventArgs e) |
| { |
| Thread thread = new Thread(ThreadFuntion); |
| thread.IsBackground = true; |
| thread.Start(); |
| } |
| private void ThreadFuntion() |
| { |
| while (true) |
| { |
| this.textBox1.Text = DateTime.Now.ToString(); |
| Thread.Sleep(1000); |
| } |
| } |
| } |

这 段code 在vs2005或者2008上都抛出异常 ：Cross-thread operation not valid:Control 'textBox1' accessed from a thread other than the thread it was created on . 这是因为.net 2.0以后加强了安全机制，不允许在winform中直接跨线程访问控件的属性。那么怎么解决这个问题呢，下面提供几种方案。

第一种方案： 在Thread创建之气，将Control.CheckForIllegalCrossThreadCalls 设为 false。 此代码告诉编译器：在这个类中我们不检查跨线程的调用是否合法（如果没有加这句话运行也没有异常，那么说明系统以及默认的采用了不检查的方式）。然而，这 种方法不可取。我们查看CheckForIllegalCrossThreadCalls 这个属性的定义，就会发现它是一个static的，也就是说无论我们在项目的什么地方修改了这个值，他就会在全局起作用。而且像这种跨线程访问是否存在异 常，我们通常都会去检查。如果项目中其他人修改了这个属性，那么我们的方案就失败了，我们要采取另外的方案。

**第二种方案**

[View Row Code](http://dotnet.9sssd.com/winform/art/851)

|  |
| --- |
| namespace TestInvoker |
| { |
| public partial class Form1 : Form |
| { |
| public Form1() |
| { |
| InitializeComponent(); |
| } |
|  |
| private void button1\_Click(object sender, EventArgs e) |
| { |
| Thread thread = new Thread(new ThreadStart(StartSomeWorkFromUIThread)); |
| thread.IsBackground = true; |
| thread.Start(); |
| //StartSomeWorkFromUIThread(); |
| //label1.Text = "Set value through another thread!"; |
| } |
|  |
| private void StartSomeWorkFromUIThread() |
| { |
| if (this.InvokeRequired) |
| { |
| BeginInvoke(new EventHandler(RunsOnWorkerThread), null); |
| } |
| else |
| { |
| RunsOnWorkerThread(this, null); |
| } |
| } |
|  |
| private void RunsOnWorkerThread(object sender, EventArgs e) |
| { |
| Thread.Sleep(2000); |
| label1.Text = System.DateTime.Now.ToString(); |
| } |
| } |
| } |

通过上叙代码，可以看到问题已经被解决了，通过等待异步，我们就不会总是持有主线程的控制，这样就可以在不发生跨线程调用异常的情况下完成多线程对winform多线程控件的控制了。

**C# WinForm多线程开发（二） ThreadPool 与 Timer**

2012-10-22 20:16 来源：博客园 作者：joechen 字号：T|T

[**摘要**]本文介绍C# WinForm多线程开发之ThreadPool 与 Timer，并提供详细的示例代码供参考。

本文接上文，继续探讨WinForm中的多线程问题，再次主要探讨threadpool 和timer。

**一 ThreadPool**

线 程池（ThreadPool）是一种相对较简单的方法，它适应于一些需要多个线程而又较短任务（如一些常处于阻塞状态的线程），它的缺点是对创建的线程不 能加以控制，也不能设置其优先级。由于每个进程只有一个线程池，当然每个应用程序域也只有一个线程池（对线），所以你将发现 ThreadPool类的成员函数都为static！当你首次调用ThreadPool.QueueUserWorkItem、 ThreadPool.RegisterWaitForSingleObject等，便会创建线程池实例。下面我就线程池当中的两函数作一介绍：

[View Row Code](http://dotnet.9sssd.com/winform/art/852)

|  |
| --- |
| public static bool QueueUserWorkItem( //调用成功则返回true |
| WaitCallback callBack,//要创建的线程调用的委托 |
| object state //传递给委托的参数 |
| )//它的另一个重载函数类似,只是委托不带参数而已 |

此函数的作用是把要创建的线程排队到线程池，当线程池的可用线程数不为零时（线程池有创建线程数的限制，缺身值为25），便创建此线程，否则就排队到线程池等到它有可用的线程时才创建。

[View Row Code](http://dotnet.9sssd.com/winform/art/852)

|  |
| --- |
| public static RegisteredWaitHandle RegisterWaitForSingleObject( |
| WaitHandle waitObject,// 要注册的 WaitHandle |
| WaitOrTimerCallback callBack,// 线程调用的委托 |
| object state,//传递给委托的参数 |
| int TimeOut,//超时，单位为毫秒， |
| bool executeOnlyOnce //是否只执行一次 |
| ); |
|  |
| public delegate void WaitOrTimerCallback( |
| object state,//也即传递给委托的参数 |
| bool timedOut//true表示由于超时调用，反之则因为waitObject |
| ); |

此 函数的作用是创建一个等待线程，一旦调用此函数便创建此线程，在参数waitObject变为终止状态或所设定的时间TimeOut到了之前，它都处于 “阻塞”状态，值得注意的一点是此“阻塞”与Thread的WaitSleepJoin状态有很大的不同：当某Thread处于 WaitSleepJoin状态时CPU会定期的唤醒它以轮询更新状态信息，然后再次进入WaitSleepJoin状态，线程的切换可是很费资源的；而 用此函数创建的线程则不同，在触发它运行之前，CPU不会切换到此线程，它既不占用CPU的时间又不浪费线程切换时间，但CPU又如何知道何时运行它？实 际上线程池会生成一些辅助线程用来监视这些触发条件，一旦达到条件便启动相应的线程，当然这些辅助线程本身也占用时间，但是如果你需创建较多的等待线程 时，使用线程池的优势就越加明显。

更详细内容demo：

[View Row Code](http://dotnet.9sssd.com/winform/art/852)

|  |
| --- |
| namespace TestMethodInvoker |
| { |
| public partial class Form2 : Form |
| { |
| public Form2() |
| { |
| InitializeComponent(); |
| } |
|  |
| private void button1\_Click(object sender, EventArgs e) |
| { |
| //ThreadPool.RegisterWaitForSingleObject( |
| // ev, |
| // new WaitOrTimerCallback(WaitThreadFunc), |
| // 4, |
| // 2000, |
| // false//表示每次完成等待操作后都重置计时器，直到注销等待 |
| // ); |
| ThreadPool.QueueUserWorkItem(new WaitCallback(ThreadFunc), "test1"); |
| //Thread.Sleep(10000); |
| } |
|  |
|  |
| private delegate void MyInvokeDelegate(string name); |
|  |
| private void Test(object o) |
| { |
| richTextBox1.Text += string.Format("the object is {0} \n", o); |
| } |
|  |
| public void ThreadFunc(object b) |
| { |
| this.Invoke(new MyInvokeDelegate(Test), b); |
| } |
| public void WaitThreadFunc(object b, bool t) |
| { |
| richTextBox1.Text += string.Format("the object is {0},t is {1}\n", b, t); |
| } |
|  |
|  |
| } |
| } |

一个很值得扩展的地方时，这里的invoke 用的是代理，其实还有其他的方法，比如 action 和func。实例代码如下：

[View Row Code](http://dotnet.9sssd.com/winform/art/852)

|  |
| --- |
| this.Invoke(new Action<string>(this.ChangeText), o.ToString()); |
|  |
| this.Invoke(new Action(delegate() { this.textBox1.Text = o.ToString();})); |
|  |
|  |
| private void DoSomething(object o) |
| { |
| System.Func<string, int> f = new Func<string, int>(this.GetId); |
| object result = this.Invoke(f, o.ToString()); |
| MessageBox.Show(result.ToString()); |
| } |
|  |
| private int GetId(string name) |
| { |
| this.textBox1.Text = name; |
| if (name == "Y") |
| { |
| return 999; |
| } |
| else |
| { |
| return 0; |
| } |
| } |

**二 Timer**

它适用于需周期性调用的方法，它不在创建计时器的线程中运行，它在由系统自动分配的单独线程中运行。这和Win32中的SetTimer方法类似。它的构造为：

[View Row Code](http://dotnet.9sssd.com/winform/art/852)

|  |
| --- |
| public Timer( |
| TimerCallback callback,//所需调用的方法 |
| object state,//传递给callback的参数 |
| int dueTime,//多久后开始调用callback |
| int period//调用此方法的时间间隔 |
| );// |

如 果 dueTime 为0，则 callback 立即执行它的首次调用。如果 dueTime 为 Infinite，则 callback 不调用它的方法。计时器被禁用，但使用 Change 方法可以重新启用它。如果 period 为0或 Infinite，并且 dueTime 不为 Infinite，则 callback 调用它的方法一次。计时器的定期行为被禁用，但使用 Change 方法可以重新启用它。如果 period 为零 (0) 或 Infinite，并且 dueTime 不为 Infinite，则 callback 调用它的方法一次。计时器的定期行为被禁用，但使用 Change 方法可以重新启用它。

在创建计时器之后若想改变它的period和dueTime，我们可以通过调用Timer的Change方法来改变：

[View Row Code](http://dotnet.9sssd.com/winform/art/852)

|  |
| --- |
| public bool Change( |
| int dueTime, |
| int period |
| );// |

显然所改变的两个参数对应于Timer中的两参数。

**C# WinForm多线程开发（三） Control.Invoke**

2012-10-22 20:23 来源：博客园 作者：joechen 字号：T|T

[**摘要**]本文介绍C# WinForm多线程开发之Control.Invoke，并提供详细的示例代码供参考。

下面我们就把在Windows Form软件中使用Invoke时的多线程要注意的问题给大家做一个介绍。

首 先，什么样的操作需要考虑使用多线程？总的一条就是，负责与用户交互的线程(以下简称为UI线程)应该保持顺畅，当UI线程调用的API可能引起阻塞时间 超过30毫秒时(比如访问CD-ROM等速度超慢的外设、进行远程调用等等)就应该考虑使用多线程。为什么是30毫秒？30毫秒的概念是人眼可以察觉到的 一个迟滞，大约等同于电影里的一帧停留的时间，最长不要超过100毫秒。

第二，最方便和简单的多线程是使用线程池。通过线程池里的线程运行 代码的最简便方法则是使用异步委托调用。注意委托调用通常是同步完成的，请使用BeginInvoke方法，这样就可以把要调用的方法排队到线程池里等候 处理，而程序的流程会立刻返回到调用方(此处是UI线程)，而调用方因此不会出现阻塞。

看看下面的例子我们就发现要使用线程池异步执行代码 也并非十分复杂，这里我们利用System.Windows.Forms.MethodInvoker委托进行异步调用。注意MethodInvoker 委托不接受方法参数，如果需要向异步执行的方法传递参数，请使用其他委托，或者需要自己定义。

[View Row Code](http://dotnet.9sssd.com/winform/art/853)

|  |
| --- |
| private void StartSomeWorkFromUIThread () { |
| // 我们要做的工作相对UI线程而言台慢了，用下面的方法异步进行处理 |
| MethodInvoker mi = new MethodInvoker(RunsOnWorkerThread);//这是入口方法 |
| mi.BeginInvoke(null, null); // 这样就不会阻塞 |
| } |
|  |
| // 缓慢的工作在此方法内进行处理，使用线程池里的线程 |
| private void RunsOnWorkerThread() { |
| DoSomethingSlow(); |
| } |

归纳上述方法，对UI线程而言实际上就是：1、发出调用，2、立刻返回，具体运行过程不理了，这样UI线程就不会被阻塞。这种方法很重要，下面我们会深入介绍。除了上面的方法，还有其他使用线程池的方法，当然如果你高兴也可以自己创建线程。

第 三，在Windows Form中使用多线程的，最重要的一条注意事项是，除了创建控件的线程以外，绝对不要在任何其他线程里面调用控件的成员(只有极个别情况例外)，也就是说 控件属于创建它的线程，不能从其他线程里面访问。这一条适用于所有从System.Windows.Forms.Control派生的控件(因此可以说是 几乎所有控件)，包括Form控件本身也是。举一反三，我们很容易得出这样的结论，控件的子控件必须由创建控件的线程来创建，比如一个表单上的按钮，比如 由创建表单的线程来创建，因此，一个窗口中的所有控件实际上都活在同一个线程之中。在实际编程时，大多数的软件的做法都是让同一线程负责全部的控件，这就 是我们所说的UI线程。看下面的例子：

[View Row Code](http://dotnet.9sssd.com/winform/art/853)

|  |
| --- |
| // 这是由UI线程定义的Label控件 |
| private Label lblStatus; |
| . |
| // 以下方法不在UI线程上执行 |
| private void RunsOnWorkerThread() { |
| DoSomethingSlow(); |
| lblStatus.Text = "Finished!"; // 这是错的 |
| } |

我 们要特别提醒大家，很多人刚开始的时候都会使用以上的方法来访问不在同一个线程里的控件(包括笔者本人)，而且在1.0版.Net 框架上似乎没有发现问题，但是这根本就是错的，更糟糕的是，程序员在这里不会得到任何错误提示，一开始就上当受骗，之后会莫明其妙地发现其他错误，这就是 Windows Form多线程编程的痛苦所在。笔者试过花很多时间来Debug自己写的Splash窗口突然消失的问题，结果还是失败了：笔者在软件的引导过程中，用另 外一个线程里创建了一个Splash窗口来显示欢迎信息，然后尝试把主线程里引导的状态直接写入到Splash窗口上的控件中，开始还OK，可是过一会 Splash窗口就莫明其妙消失了。

理解了这一点，我们应该留意到，有时候即使没有用System.Threading.Thread来显式创建一个线程，我们也可能因为使用了异步委托的BeginInvoke方法来隐式创建了线程(从线程池里)，在这种线程里也同样不能调用UI线程所创建的控件的成员。

第 四，由于上述限制，我们可能会感到很不方便，的确，当我们利用一个新创建的线程来执行某些花时间的运算时，怎样知道运算进度如何并通过UI反映给用户呢？ 解决方法很多！比如熟悉多线程编程的用户很快会想到，我们采用一些低级的同步方法，工作者线程把状态保存到一个同步对象中，让UI线程轮询 (Polling)该对象并反馈给用户就可以了。不过，这还是挺麻烦的，实际上不用这样做，Control类(及其派生类)对象有一个Invoke方法很 特别，这是少数几个不受线程限制的成员之一。我们前面说到，绝对不要在任何其他线程里面调用非本线程创建的控件的成员时，也说了“只有极个别情况例外”， 这个Invoke方法就是极个别情况之一----Invoke方法可以从任何线程里面调用。下面我们来讲解Invoke方法。

Invoke 方法的参数很简单，一个委托，一个参数表(可选)，而Invoke方法的主要功能就是帮助你在UI线程(即创建控件的线程)上调用委托所指定的方法。 Invoke方法首先检查发出调用的线程(即当前线程)是不是UI线程，如果是，直接执行委托指向的方法，如果不是，它将切换到UI线程，然后执行委托指 向的方法。不管当前线程是不是UI线程，Invoke都阻塞直到委托指向的方法执行完毕，然后切换回发出调用的线程(如果需要的话)，返回。注意，使用 Invoke方法时，UI线程不能处于阻塞状态。以下MSDN里关于Invoke方法的说明：

“控件上有四种方法可以安全地从任何线程进行调用：Invoke、BeginInvoke、EndInvoke 和 CreateGraphics。对于所有其他方法调用，则应使用调用 (invoke) 方法之一封送对控件的线程的调用。  
委 托可以是 EventHandler 的实例，在此情况下，发送方参数将包含此控件，而事件参数将包含 EventArgs.Empty。委托还可以是 MethodInvoker 的实例或采用 void 参数列表的其他任何委托。调用 EventHandler 或 MethodInvoker 委托比调用其他类型的委托速度更快。”

好了，说完Invoke，顺便说说BeginInvoke，毫无疑问这是Invoke的异步版本 (Invoke是同步完成的)，不过大家不要和上面的System.Windows.Forms.MethodInvoker委托中的 BeginInvoke混淆，两者都是利用不同线程来完成工作，但是控件的BeginInvoke方法总是使用UI线程，而其他的异步委托调用方法则是利 用线程池里的线程。相对Invoke而言，使用BeginInvoke稍稍麻烦一点，但还是那句话，异步比同步效果好，尽管复杂些。比如同步方法可能出现 这样一种死锁情况：工作者线程通过Invoke同步调用UI线程里的方法时会阻塞，而万一UI线程正在等待工作者线程做某件事时怎么办？因此，能够使用异 步方法时应尽量使用异步方法。

下面我们利用所学到的知识来改写上面那个简单的例子：

[View Row Code](http://dotnet.9sssd.com/winform/art/853)

|  |
| --- |
| // 这是由UI线程定义的Label控件 |
| private Label lblStatus; |
| . |
| // 以下方法不在UI线程上执行 |
| private void RunsOnWorkerThread() { |
| DoSomethingSlow(); |
| // Do UI update on UI thread |
| object[] pList = { this, System.EventArgs.Empty }; |
| lblStatus.BeginInvoke( |
| new System.EventHandler(UpdateUI), pList); |
| } |
|  |
| // 切换回UI线程执行的入口 |
| private void UpdateUI(object o, System.EventArgs e) { |
| //现在没问题了，使用Invoke使得线程总是回到UI线程，所以我们可以放心大胆地调用控件的成员了 |
| lblStatus.Text = "Finished!"; |
| } |

第五，关于多线程编程还要考虑线程之间的同步问题、死锁和争用条件，有关这类问题的文章很多，我们就不赘述了。

**WinForm UI线程机制简介**

2012-10-22 20:42 来源：博客园 作者：joechen 字号：T|T

[**摘要**]Windows Form的UI线程既form程序运行的主线程，它负责界面的创建，更新等与界面直接有关的工作。那么，这个线程的实现机制是什么呢，本文介绍WinForm UI线程机制。

Windows Form的UI线程既form程序运行的主线程，它负责界面的创建，更新等与界面直接有关的工作。那么，这个线程的实现机制是什么呢，它与windows的消息机制有没有关联呢。本文从这里出手，来浅谈一下ui线程机制。

每 一个从Control类中派生出来的WinForm类（包括Control类）都是依靠底层Windows消息和一个消息泵循环（message pump loop）来执行的。消息循环都必须有一个相对应的线程，因为发送到一个window的消息实际上只会被发送到创建该window的线程中去。其结果是， 即使提供了同步（synchronization），你也无法从多线程中调用这些处理消息的方法。大多数plumbing是掩藏起来的，因为 WinForm是用代理（delegate）将消息绑定到事件处理方法中的。WinForm将Windows消息转换为一个基于代理的事件，但你还是必须 注意，由于最初消息循环的缘故，只有创建该form的线程才能调用其事件处理方法。如果你在你自己的线程中调用这些方法，则它们会在该线程中处理事件，而 不是在指定的线程中进行处理。你可以从任何线程中调用任何不属于消息处理的方法。

Control类（及其派生类）实现了一个定义在System.ComponentModel命名空间下的接口 -- ISynchronizeInvoke，并以此来处理多线程中调用消息处理方法的问题：

[View Row Code](http://dotnet.9sssd.com/winform/art/854)

|  |
| --- |
| public interface ISynchronizeInvoke |
| { |
| object Invoke(Delegate　method,object[] args); |
| IAsyncResult BeginInvoke(Delegate　method,object[] args); |
| object EndInvoke(IAsyncResult result); |
| bool InvokeRequired {get;} |
| } |

ISynchronizeInvoke 提供了一个普通的标准机制用于在其他线程的对象中进行方法调用。例如，如果一个对象实现了ISynchronizeInvoke，那么在线程T1上的客户 端可以在该对象中调用ISynchronizeInvoke的Invoke()方法。Invoke()方法的实现会阻塞（block）该线程的调用，它将 调用打包发送（marshal）到 T2，并在T2中执行调用，再将返回值发送会T1，然后返回到T1的客户端。Invoke()方法以一个代理来定位该方法在T2中的调用，并以一个普通的 对象数组做为其参数。

调用者还可以检查InvokeRequired属性，因为你既可以在同一线程中调用 ISynchronizeInvoke也可以将它重新定位（redirect）到其他线程中去。如果InvokeRequired的返回值是false的 话，则调用者可以直接调用该对象的方法。

比如，假设你想要从另一个线程中调用某个form中的Close方法，那么你可以使用预先定义好的的MethodInvoker代理，并调用Invoke方法：

[View Row Code](http://dotnet.9sssd.com/winform/art/854)

|  |
| --- |
| Form form; |
| /\* obtain a reference to the form, then: \*/ |
| ISynchronizeInvoke synchronizer; |
| synchronizer = form; |
| if(synchronizer.InvokeRequired) |
| { |
| MethodInvoker invoker = new |
| MethodInvoker(form.Close); |
| synchronizer.Invoke(invoker,null); |
| } |
| else |
| form.Close(); |

ISynchronizeInvoke 不仅仅用于WinForm中。例如，一个Calculator类提供了将两个数字相加的Add()方法，它就是通过ISynchronizeInvoke 来实现的。用户必须确定ISynchronizeInvoke.Invoke()方法的调用是执行在正确的线程中的。

**C# 在正确的线程中写入调用**

列表A. Calculator类的Add()方法用于将两个数字相加。如果用户直接调用Add()方法，它会在该用户的线程中执行调用，而用户可以通过ISynchronizeInvoke.Invoke()将调用写入正确的线程中。

列表A：

[View Row Code](http://dotnet.9sssd.com/winform/art/854)

|  |
| --- |
| public class Calculator : ISynchronizeInvoke |
| { |
| public int Add(int arg1,int arg2) |
| { |
| int threadID = Thread.CurrentThread.GetHashCode(); |
| Trace.WriteLine( "Calculator thread ID is " + threadID.ToString()); |
| return arg1 + arg2; |
| } |
| //ISynchronizeInvoke implementation |
| public object Invoke(Delegate method,object[] args) |
| { |
| public IAsyncResult BeginInvoke(Delegate method,object[] args) |
| { |
| public object EndInvoke(IAsyncResult result) |
| { |
| public bool InvokeRequired |
| { |
| } |
| } |
| //Client-side code |
| public delegate int AddDelegate(int arg1,int arg2); |
| int threadID = Thread.CurrentThread.GetHashCode(); |
| Trace.WriteLine("Client thread ID is " + threadID.ToString()); |
| Calculator calc; |
| /\* Some code to initialize calc \*/ |
| AddDelegate addDelegate = new AddDelegate(calc.Add); |
| object[] arr = new object[2]; |
| arr[0] = 3; |
| arr[1] = 4; |
| int sum = 0; |
| sum = (int) calc.Invoke(addDelegate,arr); |
| Debug.Assert(sum ==7); |
|  |
| /\* Possible output: |
| Calculator thread ID is 29 |
| Client thread ID is 30 |
| \*/ |
| } |
| } |
| } |

或 许你并不想进行同步调用，因为它被打包发送到另一个线程中去了。你可以通过BeginInvoke()和EndInvoke()方法来实现它。你可以依照 通用的.NET非同步编程模式（asynchronous programming model）来使用这些方法：用BeginInvoke()来发送调用，用EndInvoke()来实现等待或用于在完成时进行提示以及收集返回结果。

还值得一提的是ISynchronizeInvoke方法并非安全类型。 类型不符会导致在执行时被抛出异常，而不是编译错误。所以在使用ISynchronizeInvoke时要格外注意，因为编辑器无法检查出执行错误。

实 现ISynchronizeInvoke要求你使用一个代理来在后期绑定（late binding）中动态地调用方法。每一种代理类型均提供DynamicInvoke()方法： public object DynamicInvoke(object[] args);

理论上来说，你必须将一个方法代理放到一个需要提供对象运行的真实的线程中 去，并使Invoke() 和BeginInvoke()方法中的代理中调用DynamicInvoke()方法。ISynchronizeInvoke的实现是一个非同一般的编程 技巧，本文附带的源文件中包含了一个名为Synchronizer的帮助类（helper class）和一个测试程序，这个测试程序是用来论证列表A中的Calculator类是如何用Synchronizer类来实现 ISynchronizeInvoke的。Synchronizer是ISynchronizeInvoke的一个普通实现，你可以使用它的派生类或者将 其本身作为一个对象来使用，并将ISynchronizeInvoke实现指派给它。

用来实现Synchronizer的一个重要元素是使 用一个名为WorkerThread的嵌套类（nested class）。WorkerThread中有一个工作项目（work item）查询。WorkItem类中包含方法代理和参数。Invoke()和BeginInvoke()用来将一个工作项目实例加入到查询里。 WorkerThread新建一个.NET worker线程，它负责监测工作项目的查询任务。查询到项目之后，worker会读取它们，然后调用DynamicInvoke()方法。