编程资料 - 多线程

C#多线程编程实例实战

作者： 刘弹 www.ASPCool.com 时间:2003-5-17 上午 10:24:05 阅读次数:10996

单个写入程序/多个阅读程序在.Net 类库中其实已经提供了实现，即 System.Threading.ReaderWriterLock 类。本文通过对常见的单个写入/多个阅读程序的分析来探索 c# 的多线程编程。

问题的提出

所谓单个写入程序/多个阅读程序的线程同步问题，是指任意数量的线程访问共享资源时，写入程序（线程）需要修改共享资源，而阅读程序（线程）需要读取数据。在这个同步问题中，很容易得到下面二个要求：

1） 当一个线程正在写入数据时，其他线程不能写，也不能读。

2） 当一个线程正在读入数据时，其他线程不能写，但能够读。

在数据库应用程序环境中经常遇到这样的问题。比如说，有 n 个最终用户，他们都要同时访问同一个数据库。其中有 m 个用户要将数据存入数据库，n-m 个用户要读取数据库中的记录。

很显然，在这个环境中，我们不能让两个或两个以上的用户同时更新同一条记录，如果两个或两个以上的用户都试图同时修改同一记录，那么该记录中的信息就会被破坏。

我们也不让一个用户更新数据库记录的同时，让另一用户读取记录的内容。因为读取的记录很有可能同时包含了更新和没有更新的信息，也就是说这条记录是无效的记录。

实现分析

规定任一线程要对资源进行写或读操作前必须申请锁。根据操作的不同，分为阅读锁和写入锁，操作完成之后应释放相应的锁。将单个写入程序/多个阅读程序的要求改变一下，可以得到如下的形式：

一个线程申请阅读锁的成功条件是：当前没有活动的写入线程。

一个线程申请写入锁的成功条件是：当前没有任何活动（对锁而言）的线程。

因此，为了标志是否有活动的线程，以及是写入还是阅读线程，引入一个变量 m_nActive，如果 m_nActive＞0，则表示当前活动阅读线程的数目，如果 m_nActive=0，则表示没有任何活动线程，m_nActive<0，表示当前有写入线程在活动，注意 m_nActive<0，时只能取-1 的值，因为只允许有一个写入线程活动。

为了判断当前活动线程拥有的锁的类型，我们采用了线程局部存储技术（请参阅其它参考书籍），

将线程与特殊标志位关联起来。

申请阅读锁的函数原型为：public void AcquireReaderLock(int millisecondsTimeout)，其中的参数为线程等待调度的时间。函数定义如下：

```
public void AcquireReaderLock( int millisecondsTimeout )

{

// m_mutext 很快可以得到，以便进入临界区

m_mutex.WaitOne( );

// 是否有写入线程存在

bool bExistingWriter = ( m_nActive < 0 );

if( bExistingWriter )

{ //等待阅读线程数目加 1,当有锁释放时，根据此数目来调度线程

m_nWaitingReaders++;

}

else

{ //当前活动线程加 1

m_nActive++;

}

m_mutex.ReleaseMutex();

//存储锁标志为 Reader

System.LocalDataStoreSlot slot = Thread.GetNamedDataSlot(m_strThreadSlotName);

object obj = Thread.GetData( slot );

LockFlags flag = LockFlags.None;

if( obj != null )
```

```
flag = (LockFlags)obj ;

if( flag == LockFlags.None )

{

Thread.SetData( slot, LockFlags.Reader );

}

else

{

Thread.SetData( slot, (LockFlags)((int)flag | (int)LockFlags.Reader ) );

}


if( bExistingWriter )

{ //等待指定的时间

this.m_aeReaders.WaitOne( millisecondsTimeout, true );

}

}
```

它首先进入临界区（用以在多线程环境下保证活动线程数目的操作的正确性）判断当前活动线程的数目，如果有写线程(m_nActive<0)存在，则等待指定的时间并且等待的阅读线程数目加 1。如果当前活动线程是读线程(m_nActive>=0)，则可以让读线程继续运行。

申请写入锁的函数原型为：public void AcquireWriterLock(int millisecondsTimeout)，其中的参数为等待调度的时间。函数定义如下：

```
public void AcquireWriterLock( int millisecondsTimeout )

{

// m_mutext 很快可以得到，以便进入临界区

m_mutex.WaitOne( );
```

```
// 是否有活动线程存在

bool bNoActive = m_nActive == 0;

if( !bNoActive )

{

m_nWaitingWriters++;

}

else

{

m_nActive--;

}

m_mutex.ReleaseMutex();

//存储线程锁标志

System.LocalDataStoreSlot slot = Thread.GetNamedDataSlot( "myReaderWriterLockDataSlot" );

object obj = Thread.GetData( slot );

LockFlags flag = LockFlags.None;

if( obj != null )

flag = (LockFlags)Thread.GetData( slot );

if( flag == LockFlags.None )

{

Thread.SetData( slot, LockFlags.Writer );

}

else
```

```
{

Thread.SetData( slot, (LockFlags)((int)flag | (int)LockFlags.Writer ) );

}

//如果有活动线程，等待指定的时间

if( !bNoActive )

this.m_aeWriters.WaitOne( millisecondsTimeout, true );

}
```

它首先进入临界区判断当前活动线程的数目，如果当前有活动线程存在，不管是写线程还是读线程（m_nActive），线程将等待指定的时间并且等待的写入线程数目加 1，否则线程拥有写的权限。

释放阅读锁的函数原型为：public void ReleaseReaderLock()。函数定义如下：

```
public void ReleaseReaderLock()

{

System.LocalDataStoreSlot slot = Thread.GetNamedDataSlot(m_strThreadSlotName );

LockFlags flag = (LockFlags)Thread.GetData( slot );

if( flag == LockFlags.None )

{
return;

}

bool bReader = true;

switch( flag )

{

case LockFlags.None:

break;
```

```
case LockFlags.Writer:

bReader = false;

break;

}

if( !bReader )

return;

Thread.SetData( slot, LockFlags.None );

m_mutex.WaitOne();

AutoResetEvent autoresetevent = null;

this.m_nActive --;

if( this.m_nActive == 0 )

{

if( this.m_nWaitingReaders > 0 )

{

m_nActive ++ ;

m_nWaitingReaders --;

autoresetevent = this.m_aeReaders;

}

else if( this.m_nWaitingWriters > 0)

{

m_nWaitingWriters--;

m_nActive --;
```

```
autoresetevent = this.m_aeWriters ;

}

}

m_mutex.ReleaseMutex();

if( autoresetevent != null )

autoresetevent.Set();

}
```

释放阅读锁时，首先判断当前线程是否拥有阅读锁（通过线程局部存储的标志），然后判断是否有等待的阅读线程，如果有，先将当前活动线程加 1，等待阅读线程数目减 1，然后置事件为有信号。如果没有等待的阅读线程，判断是否有等待的写入线程，如果有则活动线程数目减 1，等待的写入线程数目减 1。释放写入锁与释放阅读锁的过程基本一致，可以参看源代码。

注意在程序中，释放锁时，只会唤醒一个阅读程序，这是因为使用 AutoResetEvent 的原历，读者可自行将其改成 ManualResetEvent，同时唤醒多个阅读程序，此时应令 m_nActive 等于整个等待的阅读线程数目。

测试

测试程序取自.Net FrameSDK 中的一个例子，只是稍做修改。测试程序如下，

```
using System;

using System.Threading;

using MyThreading;

class Resource {

myReaderWriterLock rwl = new myReaderWriterLock();

public void Read(Int32 threadNum) {

rwl.AcquireReaderLock(Timeout.Infinite);

try {
```

```
Console.WriteLine("Start Resource reading (Thread={0})", threadNum);

Thread.Sleep(250);

Console.WriteLine("Stop Resource reading (Thread={0})", threadNum);

}

finally {

rwl.ReleaseReaderLock();

}

}

public void Write(Int32 threadNum) {

rwl.AcquireWriterLock(Timeout.Infinite);

try {

Console.WriteLine("Start Resource writing (Thread={0})", threadNum);

Thread.Sleep(750);

Console.WriteLine("Stop Resource writing (Thread={0})", threadNum);

}

finally {

rwl.ReleaseWriterLock();

}

}

}

class App {

static Int32 numAsyncOps = 20;
```

```
static AutoResetEvent asyncOpsAreDone = new AutoResetEvent(false);

static Resource res = new Resource();



public static void Main() {

for (Int32 threadNum = 0; threadNum < 20; threadNum++) {

ThreadPool.QueueUserWorkItem(new WaitCallback(UpdateResource), threadNum);

}

asyncOpsAreDone.WaitOne();

Console.WriteLine("All operations have completed.");

Console.ReadLine();

}

// The callback method's signature MUST match that of a System.Threading.TimerCallback

// delegate (it takes an Object parameter and returns void)

static void UpdateResource(Object state) {

Int32 threadNum = (Int32) state;

if ((threadNum % 2) != 0) res.Read(threadNum);

else res.Write(threadNum);

if (Interlocked.Decrement(ref numAsyncOps) == 0)

asyncOpsAreDone.Set();

}

}
```

从测试结果中可以看出，可以满足单个写入程序\多个阅读程序的实现要求。

C# 一个多线程操作控件的例子.

```
//多线程间 控件的操作例子
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;
using System.Threading;
using System.Data.SqlClient;
using System.Collections;

namespace AutoMessager
{
    delegate void myDelegate();
    delegate void SetTextCallback(string text);

    public partial class frmAutoMsg : Form
    {
        event myDelegate myEvent;
        string connStr = string.Empty;
        Thread thd;
        //private Icon eyeIcon;

        //private NotifyIconEx notifyIconA;
        //private NotifyIconEx notifyIconB;
        private bool canClosed = false;

        public frmAutoMsg()
        {
            this.ShowInTaskbar = false;
            InitializeComponent();
            //eyeIcon = new Icon(GetType(), "EYE.ICO");
            notifyIcon1.ContextMenu = contextMenuB;
        }

        private void SetText(string text)
        {
            // InvokeRequired required compares the thread ID of the
            // calling thread to the thread ID of the creating thread.
            // If these threads are different, it returns true.
            if (this.txtMsgStatus.InvokeRequired)
            {
```

```
                SetTextCallback d = new SetTextCallback(SetText);
                this.Invoke(d, new object[] { text });
            }
            else
            {
                this.txtMsgStatus.Text += text;
            }
        }


        private void frmAutoMsg_Load(object sender, EventArgs e)
        {
            connStr = System.Configuration.ConfigurationManager.AppSe
ttings["ConnString"];
            thd = new Thread(new ThreadStart(doEvent));
            thd.IsBackground = true;
            thd.Start();
            //doEvent();
            //notifyIcon1.Visible = true;
        }

        /// <summary>
        /// 员工合同到期提醒
        /// </summary>
        void UpUserState()
        {
            SqlConnection conn = null;
            DateTime now = DateTime.Now;
            SqlTransaction tran = null;
            SqlDataReader dr = null;
            try
            {
               //数据库操作部分省略

                SetText(" 系统提示: 职员合同消息更新成功! ");
               SetText("执行时间:" + now.ToString("yyyy-MM-dd HH:mm:s
s") + " ");


            }
            catch (Exception ex)
            {
                dr.Close();
                tran.Rollback();
```

```
                SetText(" 系统错误: 职员合同消息更新错误:" + ex.Messa
ge + " ");
                SetText("执行时间:" + now.ToString("yyyy-MM-dd HH:mm:
ss") + " ");
            }
            finally
            {
                dr.Close();
                conn.Close();
                conn.Dispose();
            }
        }
        /// <summary>
        /// 采购及供货 到货提醒
        /// </summary>
        void UpCaiGou()
        {
            SqlConnection conn = null;
            DateTime now = DateTime.Now;
            SqlTransaction tran = null;
            SqlDataReader dr = null;
            try
            {
                //数据库操作部分省略

                SetText("系统提示: 合同采购消息更新成功! ");
                SetText("执行时间:" + now.ToString("yyyy-MM-dd HH:mm:
ss") + " ");


            }
            catch (Exception ex)
            {
                dr.Close();
                tran.Rollback();
                SetText("系统错误: 合同采购消息更新错误:" + ex.Messag
e + " ");
                SetText("执行时间:" + now.ToString("yyyy-MM-dd HH:mm:
ss") + " ");
            }
            finally
            {
                dr.Close();
                conn.Close();
```

```
                conn.Dispose();
            }
        }

        /// <summary>
        /// 供货收款情况提醒
        /// </summary>
        void GetMoney()
        {
            SqlConnection conn = null;
            DateTime now = DateTime.Now;
            try
            {
               //数据库操作部分省略
               SetText("系统提示: 供货付款消息更新成功! ");
               SetText("执行时间:" + now.ToString("yyyy-MM-dd HH:mm:s
s") + " ");
            }
            catch (Exception ex)
            {
                SetText("系统错误: 供货付款消息更新错误:" + ex.Messag
e + " ");
                SetText("执行时间:" + now.ToString("yyyy-MM-dd HH:mm:
ss") + " ");
            }
            finally
            {
                conn.Close();
                conn.Dispose();
            }
        }

        void doEvent()
        {
            //int weather = int.Parse(weatherTime.Text);
            //int del = int.Parse(fileTime.Text);
            //            if(weather < 1 || weather > 24 || del < 1 |
| del > 24)
            //            {
            //                MessageBox.Show("时间输入有错! ");
            //                button1.Enabled = true;
            //                return ;
            //            }
            while (true)
```

```
            {

                //DateTime now = DateTime.Now;
                int i = DateTime.Now.Hour;
                if (i > 2 && i < 4)
                {
                    myEvent = new myDelegate(UpUserState);
                    myEvent += new myDelegate(UpCaiGou);
                   // myEvent += new myDelegate(GetMoney);
                }
                //if (now.Hour == 3)
                //{
                //    myEventB = new myDelegate(deltemp);
                //}
                //if (myEventA != null) myEventA();
                //if (myEventB != null) myEventB();
                if (myEvent != null)
                {
                    myEvent();
                    myEvent = null;
                }
                Application.DoEvents();

                Thread.Sleep(6000000); //每 100 分钟检查一次时间
            }
        }

        private void frmAutoMsg_FormClosing(object sender, FormClosin
gEventArgs e)
        {
            if (canClosed == false)
            {
                e.Cancel = true;
                this.Hide();
                this.Visible = false;
                //this.
            }
        }

        private void menuItem2_Click(object sender, EventArgs e)
        {
            this.ShowInTaskbar = true;
            this.Show();
            this.Visible = true; //恢复主窗体
```

```
        }

        private void menuItem1_Click(object sender, EventArgs e)
        {
            canClosed = true;
            Application.Exit();
        }

        private void notifyIcon1_MouseDoubleClick(object sender, Mous
eEventArgs e)
        {
            this.ShowInTaskbar = true;
            this.Show();
            if (this.Visible == false)
            {
                this.Visible = true;
            }

        }

        private void btnClear_Click(object sender, EventArgs e)
        {
            this.txtMsgStatus.Text = "";
        }

        private void btnUpMsg_Click(object sender, EventArgs e)
        {
            myEvent = new myDelegate(UpUserState);
            myEvent += new myDelegate(UpCaiGou);
            //myEvent += new myDelegate(GetMoney);

            if (myEvent != null)
                myEvent();
        }
    }
}
```

.NET 事件模型教程（一）

目录

- 事件、事件处理程序概念
- 问题描述：一个需要较长时间才能完成的任务
- 高耦合的实现
- 事件模型的解决方案，简单易懂的 VB.NET 版本
- 委托（delegate）简介
- C# 实现
- 向"..NET Framework 类库设计指南"靠拢，标准实现

事件、事件处理程序概念

在面向对象理论中，一个对象（类的实例）可以有属性（property，获取或设置对象的状态）、方法（method，对象可以做的动作）等成员外，还有**事件**（**event**）。所谓事件，是对象内部状态发生了某些变化、或者对象做某些动作时（或做之前、做之后），向外界发出的通知。打个比方就是，对象"张三"肚子疼了，然后他站在空地上大叫一声"我肚子疼了！"事件就是这个通知。

那么，相对于对象内部发出的事件通知，外部环境可能需要应对某些事件的发生，而做出相应的反应。接着上面的比方，张三大叫一声之后，救护车来了把它接到医院（或者疯人院，呵呵，开个玩笑）。外界因应事件发生而做出的反应（具体到程序上，就是针对该事件而写的那些处理代码），称为**事件处理程序**（**event handler**）。

事件处理程序必须和对象的事件挂钩后，才可能会被执行。否则，孤立的事件处理程序不会被执行。另一方面，对象发生事件时，并不一定要有相应的处理程序。就如张三大叫之后，外界环境没有做出任何反应。也就是说，对象的事件和外界对该对象的事件处理之间，并没有必然的联系，需要你去挂接。

在开始学习之前，我希望大家首先区分"事件"和"事件处理程序"这两个概念。事件是隶属于对象（类）本身的，事件处理程序是外界代码针对对象的事件做出的反应。事件，是对象（类）的设计者、开发者应该完成的；事件处理程序是外界调用方需要完成的。简单的说，事件是"内"；事件处理程序是"外"。

了解以上基本概念之后，我们开始学习具体的代码实现过程。因为涉及代码比较多，限于篇幅，我只是将代码中比较重要的部分贴在文章里，进行解析，剩余代码还是请读者自己查阅，我已经把源代码打了包提供下载。我也建议你对照这些源代码，来学习教程。[下载本教程的源代码]

[TOP]

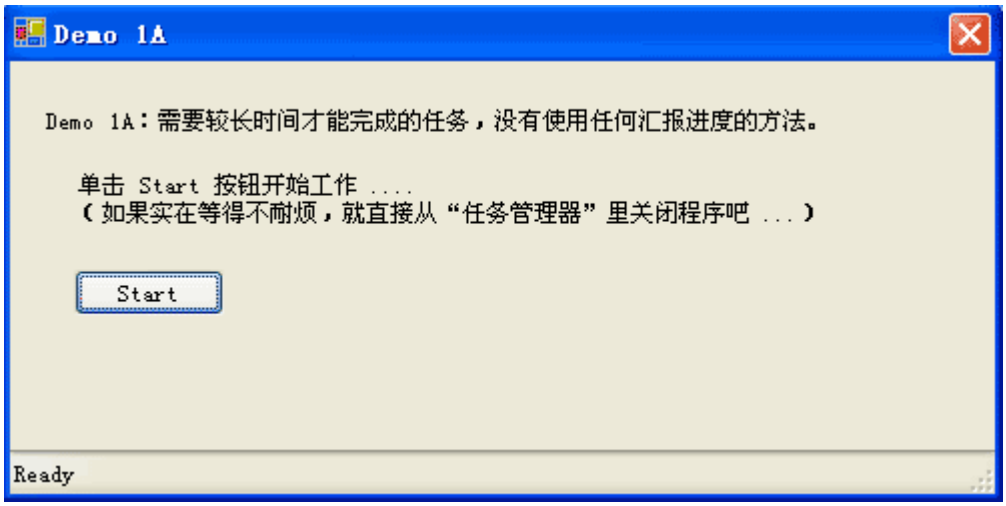
问题描述：一个需要较长时间才能完成的任务

Demo 1A，问题描述。这是一个情景演示，也是本教程中其他 Demo 都致力于解决的一个"实际问题"：Worker 类中有一个可能需要较长时间才能完成的方法 DoLongTimeTask：

```
using System;
using System.Threading;
```

```
namespace percyboy.EventModelDemo.Demo1A
{
    // 需要做很长时间才能完成任务的 Worker，没有加入任何汇报途径。
    public class Worker
    {
        // 请根据你的机器配置情况，设置 MAX 的值。
        // 在我这里（CPU: AMD Sempron 2400+, DDRAM 512MB）
        // 当 MAX = 10000，任务耗时 20 秒。
        private const int MAX = 10000;

        public Worker()
        {
        }

        public void DoLongTimeTask()
        {
            int i;
            bool t = false;
            for (i = 0; i <= MAX; i++)
            {
                // 此处 Thread.Sleep 的目的有两个：
                // 一个是不让 CPU 时间全部耗费在这个任务上：
                // 因为本例中的工作是一个纯粹消耗 CPU 计算资源的任务。
                // 如果一直让它一直占用 CPU,则 CPU 时间几乎全部都耗费
于此。
                // 如果任务时间较短，可能影响不大；
                // 但如果任务耗时也长，就可能会影响系统中其他任务的正
常运行。
                // 所以，Sleep 就是要让 CPU 有机会“分一下心”，
                // 处理一下来自其他任务的计算请求。
                //
                // 当然，这里的主要目的是为了让这个任务看起来耗时更长
一点。
                Thread.Sleep(1);

                t = !t;
            }
        }
    }
}
```

界面很简单（本教程中其他 Demo 也都沿用这个界面，因为我们主要的研究对象是 Worker.cs）：



单击“Start”按钮后，开始执行该方法。（具体的机器配置条件，完成此任务需要的时间也不同，你可以根据你的实际情况调整代码中的 MAX 值。）

在没有进度指示的情况下，界面长时间的无响应，往往会被用户认为是程序故障或者“死机”，而实际上，你的工作正在进行还没有结束。此次教程就是以解决此问题为实例，向你介绍 .NET 中事件模型的原理、设计与具体编码实现。

[TOP]

高耦合的实现

Demo 1B，高度耦合。有很多办法可以让 Worker 在工作的时候向用户界面报告进度，比如最容易想到的：

```
public void DoLongTimeTask()
{
    int i;
    bool t = false;
    for (i = 0; i <= MAX; i++)
    {
        Thread.Sleep(1);

        t = !t;

        在此处书写刷新用户界面状态栏的代码
    }
}
```

如果说 DoLongTimeTask 是用户界面（Windows 窗体）的一个方法，那么上面蓝色部分或许很简单，可能只不过是如下的两行代码：

```
            double rate = (double)i / (double)MAX;
            this.statusbar.Text = String.Format(@"已完成 {0:P2} ...",
rate);
```

不过这样的话，DoLongTimeTask 就是这个 Windows 窗体的一部分了，显然它不利于其他窗体调用这段代码。那么：Worker 类应该作为一个相对独立的部分存在。源代码 Demo1B 中给出了这样的一个示例（应该还有很多种、和它类似的方法）：

Windows 窗体 Form1 中单击"Start"按钮后，初始化 Worker 类的一个新实例，并执行它的 DoLongTimeTask 方法。但你应该同时看到，Form1 也赋值给 Worker 的一个属性，在 Worker 执行 DoLongTimeTask 方法时，通过这个属性刷新 Form1 的状态栏。Form1 和 Worker 之间相互粘在一起：Form1 依赖于 Worker 类（因为它单击按钮后要实例化 Worker），Worker 类也依赖于 Form1（因为它在工作时，需要访问 Form1）。这二者之间形成了高度耦合。

高度耦合同样不利于代码重用，你仍然无法在另一个窗体里使用 Worker 类，代码灵活度大为降低。正确的设计原则应该是努力实现低耦合：如果 Form1 必须依赖于 Worker 类，那么 Worker 类就不应该再反过来依赖于 Form1。

下面我们考虑使用 .NET 事件模型解决上述的"高度耦合"问题：

让 Worker 类在工作时，向外界发出"进度报告"的事件通知（**RateReport**）。同时，为了演示更多的情景，我们让 Worker 类在开始 DoLongTimeTask 之前发出一个"我要开始干活了！总任务数有 N 件。"的事件通知（**StartWork**），并在完成任务时发出"任务完成"的事件通知（**EndWork**）。

采用事件模型后，类 Worker 本身并不实际去刷新 Form1 的状态栏，也就是说 Worker 不依赖于 Form1。在 Form1 中，单击"Start"按钮后，Worker 的一个实例开始工作，并发出一系列的事件通知。我们需要做的是为 Worker 的事件书写事件处理程序，并将它们挂接起来。

[TOP]

事件模型的解决方案，简单易懂的 VB.NET 版本

Demo 1C，VB.NET 代码。虽然本教程以 C# 为示例语言，我还是给出一段 VB.NET 的代码辅助大家的理解。因为我个人认为 VB.NET 的事件语法，能让你非常直观的领悟到 .NET 事件模型的"思维方式"：

```
    Public Class Worker
        Private Const MAX = 10000

        Public Sub New()
        End Sub
```

```
        ' 注：此例的写法不符合 .NET Framework 类库设计指南中的约定，
        ' 只是为了让你快速理解事件模型而简化的。
        ' 请继续阅读，使用 Demo 1F 的 VB.NET 标准写法。
        '

        ' 工作开始事件，并同时通知外界需要完成的数量。
        Public Event StartWork(ByVal totalUnits As Integer)

        ' 进度汇报事件，通知外界任务完成的进度情况。
        Public Event RateReport(ByVal rate As Double)

        ' 工作结束事件。
        Public Event EndWork()

        Public Sub DoLongTimeTask()
            Dim i As Integer
            Dim t As Boolean = False
            Dim rate As Double

            ' 开始工作前，向外界发出事件通知
            RaiseEvent StartWork(MAX)

            For i = 0 To MAX
                Thread.Sleep(1)
                t = Not t

                rate = i / MAX
                RaiseEvent RateReport(rate)
            Next

            RaiseEvent EndWork()

        End Sub
```

首先是事件的声明部分：你只需写上 Public Event 关键字，然后写事件的名称，后面的参数部分写上需要发送到外界的参数声明。

然后请注意已标记为蓝色的 RaiseEvent 关键字，VB.NET 使用此关键字在类内部引发事件，也就是向外界发送事件通知。请注意它的语法，RaiseEvent 后接上你要引发的事件名称，然后是具体的事件参数值。

从这个例子中，我们可以加深对事件模型的认识：事件是对象（类）的成员，在对象（类）内部状态发生了一些变化（比如此例中 rate 在变化），或者对象做一些动作时（比如此例中，方法开始时，向外界 raise event；方法结束时，向外界 raise event），对象（类）发出的通知。并且，你也了解了事件参数的用法：事件参数是事件通知的相关内容，比如 RateReport 事件通知需要报告进度值 rate，StartWork 事件通知需要报告总任务数 MAX。

我想 RaiseEvent 很形象的说明了这些道理。

[TOP]

委托（delegate）简介。

在学习 C# 实现之前，我们首先应该了解一些关于"委托"的基础概念。

你可以简单的把"委托（delegate）"理解为 .NET 对函数的包装（这是委托的主要用途）。委托代表一"类"函数，它们都符合一定的规格，如：拥有相同的参数个数、参数类型、返回值类型等。也可以认为委托是对函数的抽象，是函数的"类"（类是具有某些相同特征的事物的抽象）。这时，委托的实例将代表一个具体的函数。

你可以用如下的方式**声明委托**：

```
        public delegate void MyDelegate(int integerParameter);
```

如上的委托将可以用于代表：有且只有一个整数型参数、且不带返回值的一组函数。它的写法和一个函数的写法类似，只是多了 delegate 关键字、而没有函数体。（注：本文中的函数（function），取了面向过程理论中惯用的术语。在完全面向对象的 .NET/C# 中，我用以指代类的实例方法或静态方法（method），希望不会因此引起误解。顺带地，既然完全面向对象，其实委托本身也是一种对象。）

委托的实例化：既然委托是函数的"类"，那么使用委托之前也需要实例化。我们先看如下的代码：

```
    public class Sample
    {
        public void DoSomething(int mode)
        {
            Console.WriteLine("test function.");
        }

        public static void Hello(int world)
        {
            Console.WriteLine("hello, world!");
        }
    }
```

我们看到 Sample 的实例方法 DoSomething 和静态方法 Hello 都符合上面已经定义了的 MyDelegate 委托的"规格"。那么我们可以使用 MyDelegate 委托来包装它们，以用于特殊的用途（比如下面要讲的事件模型，或者将来教程中要讲的多线程模型）。当然，包装的过程其实也是委托的实例化过程：

```
    Sample sp = new Sample();
    MyDelegate del = new MyDelegate(sp.DoSomething);
```

这是对上面的实例方法的包装。但如果这段代码写在 Sample 类内部，则应使用 this.DoSomething 而不用新建一个 Sample 实例。对 Sample 的 Hello 静态方法可以包装如下：

```
    MyDelegate del = new MyDelegate(Sample.Hello);
```

调用委托：对于某个委托的实例（其实是一个具体的函数），如果想执行它：

```
        del(12345);
```

直接写上委托实例的名字，并在括号中给相应的参数赋值即可。（如果函数有返回值，也可以像普通函数那样接收返回值）。

[TOP]

C# 实现

Demo 1D，C# 实现。这里给出 Demo 1C 中 VB.NET 代码的 C# 实现：是不是比 VB.NET 的代码复杂了一些呢？

```
using System;
using System.Threading;

namespace percyboy.EventModelDemo.Demo1D
{
    // 需要做很长时间才能完成任务的 Worker，这次我们使用事件向外界通知进度。
    public class Worker
    {
        private const int MAX = 10000;

        //  注：此例的写法不符合 .NET Framework 类库设计指南中的约定，
        // 只是为了让你快速理解事件模型而简化的。
        // 请继续阅读，使用 Demo 1E / Demo 1H 的 C# 标准写法。
        //

        public delegate void StartWorkEventHandler(int totalUnits);
        public delegate void EndWorkEventHandler();
        public delegate void RateReportEventHandler(double rate);

        public event StartWorkEventHandler StartWork;
        public event EndWorkEventHandler EndWork;
        public event RateReportEventHandler RateReport;
```

```
        public Worker()
        {
        }

        public void DoLongTimeTask()
        {
            int i;
            bool t = false;
            double rate;

            if (StartWork != null)
            {
                StartWork(MAX);
            }

            for (i = 0; i <= MAX; i++)
            {
                Thread.Sleep(1);
                t = !t;
                rate = (double)i / (double)MAX;

                if (RateReport != null)
                {
                    RateReport(rate);
                }
            }

            if (EndWork != null)
            {
                EndWork();
            }
        }
    }
}
```

这份代码和上面 VB.NET 代码实现一致的功能。通过 C# 代码，我们可以看到被 VB.NET 隐藏了的一些实现细节：

首先，这里一开始声明了几个委托（delegate）。然后声明了三个事件，这里请注意 C# 事件声明的方法：

```
        public event [委托类型] [事件名称];
```

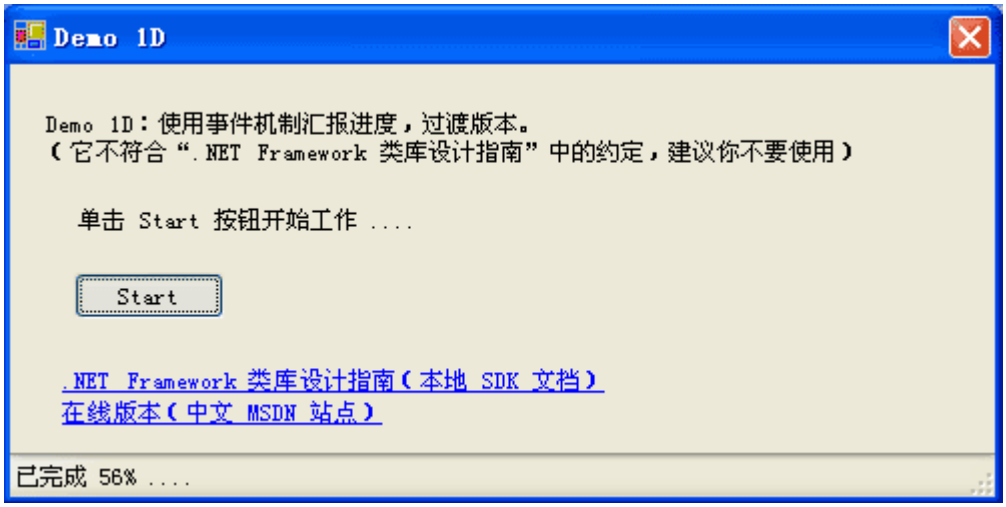
这里你可以看到 VB.NET 隐藏了声明委托的步骤。

另外提醒你注意代码中具体引发事件的部分：

```
        if (RateReport != null)
        {
            RateReport(rate);
        }
```

在调用委托之前，必须检查委托是否为 null，否则将有可能引发 NullReferenceException 意外；比较 VB.NET 的代码，VB.NET 的 RaiseEvent 语句实际上也隐藏了这一细节。

好了，到此为止，Worker 类部分通过事件模型向外界发送事件通知的功能已经有了第一个版本，修改你的 Windows 窗体，给它添加 RateReport 事件处理程序（请参看你已下载的源代码），并挂接到一起，看看现在的效果：



添加了进度指示之后的界面，极大的改善了用户体验，对用户更为友好。

[TOP]

向“.NET Framework 类库设计指南”靠拢，标准实现

Demo 1E，C# 的标准实现。上文已经反复强调了 Demo 1C, Demo 1D 代码不符合 CLS 约定。微软为 .NET 类库的设计与命名提出了一些指南，作为一种约定，.NET 开发者应当遵守这些约定。涉及事件的部分，请参看 事件命名指南（对应的在线网页），事件使用指南（对应的在线网页）。

```
using System;
using System.Threading;

namespace percyboy.EventModelDemo.Demo1E
{
    public class Worker
    {
        private const int MAX = 10000;
```

```
        public class StartWorkEventArgs : EventArgs
        {
            private int totalUnits;

            public int TotalUnits
            {
                get { return totalUnits; }
            }

            public StartWorkEventArgs(int totalUnits)
            {
                this.totalUnits = totalUnits;
            }
        }

        public class RateReportEventArgs : EventArgs
        {
            private double rate;

            public double Rate
            {
                get { return rate; }
            }

            public RateReportEventArgs(double rate)
            {
                this.rate = rate;
            }
        }

        public delegate void StartWorkEventHandler(object sender,
StartWorkEventArgs e);
        public delegate void RateReportEventHandler(object sender,
RateReportEventArgs e);

        public event StartWorkEventHandler StartWork;
        public event EventHandler EndWork;
        public event RateReportEventHandler RateReport;

        protected virtual void OnStartWork( StartWorkEventArgs e )
        {
            if (StartWork != null)
            {
```

```
                StartWork(this, e);
            }
        }

        protected virtual void OnEndWork( EventArgs e )
        {
            if (EndWork != null)
            {
                EndWork(this, e);
            }
        }

        protected virtual void OnRateReport( RateReportEventArgs e )
        {
            if (RateReport != null)
            {
                RateReport(this, e);
            }
        }

        public Worker()
        {
        }

        public void DoLongTimeTask()
        {
            int i;
            bool t = false;
            double rate;

            OnStartWork(new StartWorkEventArgs(MAX) );

            for (i = 0; i <= MAX; i++)
            {
                Thread.Sleep(1);
                t = !t;
                rate = (double)i / (double)MAX;

                OnRateReport( new RateReportEventArgs(rate) );
            }

            OnEndWork( EventArgs.Empty );
        }
    }
```

```
}
```

按照 .NET Framework 类库设计指南中的约定：

(1)事件委托名称应以 EventHandler 为结尾；

(2)事件委托的"规格"应该是两个参数：第一个参数是 object 类型的 sender，代表发出事件通知的对象（代码中一般是 this 关键字(VB.NET 中是 Me)）。第二个参数 e，应该是 EventArgs 类型或者从 EventArgs 继承而来的类型；

事件参数类型，应从 EventArgs 继承，名称应以 EventArgs 结尾。应该将所有想通过事件、传达到外界的信息，放在事件参数 e 中。

(3)一般的，只要类不是密封（C# 中的 sealed，VB.NET 中的 NotInheritable）的，或者说此类可被继承，应该为每个事件提供一个 protected 并且是可重写（C# 用 virtual，VB.NET 用 Overridable）的 OnXxxx 方法：该方法名称，应该是 On 加上事件的名称；只有一个事件参数 e；一般在该方法中进行 null 判断，并且把 this/Me 作为 sender 执行事件委托；在需要发出事件通知的地方，应调用此 OnXxxx 方法。

对于此类的子类，如果要改变发生此事件时的行为，应重写 OnXxxx 方法；并且在重写时，一般情况下应调用基类的此方法（C# 里的 base.OnXxxx，VB.NET 用 MyBase.OnXxxx）。

我建议你能继续花些时间研究一下这份代码的写法，它是 C# 的标准事件实现代码，相信你会用得着它！

在 Demo 1D 中我没有讲解如何将事件处理程序挂接到 Worker 实例的事件的代码，在这个 Demo 中，我将主要的部分列在这里：

```
        private void button1_Click(object sender, System.EventArgs e)
        {
            statusBar1.Text = "开始工作 ....";
            this.Cursor = Cursors.WaitCursor;

            long tick = DateTime.Now.Ticks;

            Worker worker = new Worker();

            // 将事件处理程序与 Worker 的相应事件挂钩
            // 这里我只挂钩了 RateReport 事件做示意
            worker.RateReport += new
Worker.RateReportEventHandler(this.worker_RateReport);

            worker.DoLongTimeTask();

            tick = DateTime.Now.Ticks - tick;
            TimeSpan ts = new TimeSpan(tick);
```

```
            this.Cursor = Cursors.Default;
            statusBar1.Text = String.Format("任务完成，耗时 {0} 秒。",
ts.TotalSeconds);
        }

        private void worker_RateReport(object sender,
Worker.RateReportEventArgs e)
        {
            this.statusBar1.Text = String.Format("已完成 {0:P0} ....",
e.Rate);
        }
```

请注意 C# 的挂接方式（"+="运算符）。

到这里为此，你已经看到了事件机制的好处：Worker 类的代码和这个 Windows Form 没有依赖关系。Worker 类可以单独存在，可以被重复应用到不同的地方。

VB.NET 的读者，请查看 Demo 1F 中的 VB.NET 标准事件写法，并参考这里的说明，我就不再赘述了。

[TOP]

2005 年 1 月 22 日 18:27 - (阅读:11953;评论:44)

评论

RE: .NET 事件模型教程（一）

2005-1-22 22:10 | 开心就好

有没有想法到 MSDN Webcast 来讲一次网络讲座，把你的这些心得传播给更多的朋友呢？如果有想法的话，请将你的文章，整理成一个 PPT，并且做一个十分钟左右的录音文件，发送到 msdnprc^_^microsoft.com(^_^变为@)。或者发给立楠也可以。

RE: .NET 事件模型教程（一）

2005-1-24 11:43 | BESTSKY

受益非浅,我以前从没有这样写过谢谢.

RE: .NET 事件模型教程（一）

2005-1-25 14:59 | TRULY

Good

RE: .NET 事件模型教程（一）

2005-1-27 17:10 | MOUSEINDARK

小弟愚笨，折腾了一个多小时才算是理解了
现在用.net 就感觉越学越是不懂，今天算是又张了见识了

RE: .NET 事件模型教程（一）

2005-2-2 17:32 | 生如夏花

前两天面试时碰到过一道关于事件代理机制的题，今天认识的更清楚了。

.NET 事件模型教程

2005-2-4 11:34 | MOREPOWER

Ping Back 来自：blog.csdn.net

RE: .NET 事件模型教程（一）

2005-3-7 15:27 | JUNO MAY

真的很感谢你写的这几篇文章，让我对 event driven 这些很模糊的概念变得清晰起来。

我是个网站设计师，喜欢用 php 和 function-oriented 的方法写应用程序，对 M$的东西都很感冒不太喜欢，但是.NET 的一些概念和方法确实对开发带来便利，尽管庞大的 framework 和 class 使得程序变得臃肿缓慢，但是大大加速开发进程和便于维护。

现在我们在用 Prado (一个借鉴了 asp.net 大部分思想的 php5 framework)开发应用程序，它山之石可以攻玉 ：）

RE: .NET 事件模型教程（一）

2005-3-17 16:44 | 冲浪

文章很不错,有没有想过出书哦....

RE: .NET 事件模型教程（一）

2005-3-21 1:25 | LIANGYJ

在《.net 框架程序设计》中，说到的"回调方法的原形应该有一个 void 返回值，并且接受两个参数，第一个参数为 object 类型，其指向发送通知的对象，第二个参数为一个继承自 EventArgs 的类型，其中包含所有通知接受者需要的附加信息"
而你在这里定义的 event 并没有符合这两个规则，那么究竟是你错？还是那本书的错呢？还是有另外的解释呢？请指点一下。

RE: .NET 事件模型教程（一）

2005-3-21 9:18 | 破宝

to Liangyj:

我相信如果你读完这第一篇教程全文的话，就不会认为我写的和你那本书有矛盾。你再看看最后一个小节"向".NET Framework 类库设计指南"靠拢，标准实现"里的内容？

RE: .NET 事件模型教程（一）

2005-4-1 4:01 | JAYE

xiexie 破宝，收藏一下不介意吧

RE: .NET 事件模型教程（一）

2005-4-9 21:29 | CQHYDZ

我是看 msdn 中自定义控件哪个录像知道事件模型的，你写的不错

RE: .NET 事件模型教程（一）

2005-4-12 17:55 | JERRY

好文啊！

我事件这一张翻来复去看了几遍都没看明白。听你这么一讲解，思路清晰了很多，多谢啊！

RE: .NET 事件模型教程（一）

2005-4-21 12:33 | 凉

写的真好，之前看的 msdn，一点也没看懂，现在总算有点明白了。

RE: .NET 事件模型教程（一）

2005-4-28 20:03 | DUR

写得很棒。:）

偶有一问。在你的 Worker.DoLongTimeTask 中写了 OnStartWork 来让 Worker 对象发出一个事件。

那么 Form 对象是怎么捕捉鼠标点了一下的呢？

如果想让某接口在收到一脉冲时产生一个事件，该怎么办呢？

RE: .NET 事件模型教程（一）

2005-5-9 3:08 | MYASPX

看了此文，对事件有了更深的了解！

RE: .NET 事件模型教程（一）

2005-5-12 19:49 | ERICZQWANG

对那个挂钩 RateReport 事件没有完全理解。EndReport 和 StartReport 没有挂钩，是不是就不执行了呢？

// 这里我只挂钩了 RateReport 事件做示意

worker.RateReport += new Worker.RateReportEventHandler(this.worker_RateReport);

RE: .NET 事件模型教程（一）

2005-5-12 20:02 | 破宝

to EricZQWang:

不知道你所谓"不执行"是指"谁"不执行？

正如文中一开始就说，事件是一个对象发出的消息，到了那个时候它就要发消息，无论是否有人注意这个消息。

如果一开始我们对某个事件挂接了一个 handler，则在这个事件发生时，handler 被执行。如果不挂钩，handler 不会执行。

RE: .NET 事件模型教程（一）

2005-5-13 8:53 | ERICZQWANG

谢谢破宝。 我 Share 一下自己新的理解："

// 这里挂钩了 RateReport 事件做示意

worker.RateReport += new Worker.RateReportEventHandler(this.worker_RateReport); "

他的作用就是使 RateReport != null.在 method:

worker.DoLongTimeTask()中，Worker 总共发出了三个消息

OnStartWork,OnRateReport,OnEndReport

在 method: button1_Click()中挂钩了 OnRateReport 消息，我觉得作用就是实例化了 Worker.RateReport,使 RateReport != null,OnStartReport 和 OnEndReport 没有挂钩，StartWork and EndStart ==null.
当 Worker 发出 OnRateReport 的消息时，会执行
this.statusBar1.Text = String.Format("已完成 {0:P0} ....", e.Rate);
但是：当 Worker 发出 OnStartWork 和 OnEndReport 消息时，
因为 StartWork 和 EndReport==null,所以在 Worker 的 Method:OnStartReport 和 OnEndReport 中什么都没有作

满分 100 的话，这个理解可以打多少分？:)

RE: .NET 事件模型教程（一）

2005-5-13 10:55 | 破宝

应该说从一个面向过程的观点来看，你的理解基本上没什么问题（当然指出一点："消息"是指 StartWork,RateRepport,EndWork，而不是 OnXxxx，后者只是 worker 的 protected 方法）。
但希望你能够上升到面向对象的角度再来领悟这个问题。祝你好运！

RE: .NET 事件模型教程（一）

2005-5-13 22:57 | GAOFAN

代码下不了啊，能不能给我发一份
gaofan628@yahoo.com.cn

谢谢先

RE: .NET 事件模型教程（一）

2005-5-28 20:26 | LHQ

虽然我还是个新手.
不过还是觉得应该更正一下 sender 和 e 不是类型而是对象
否则你怎么进行转换呢?
类本事是没有转换这个概念的
有了继承才有了转换这个概念
而转换本身又不是针对类的
因为 C#是一门面向对象的设计语言

RE: .NET 事件模型教程（一）

2005-5-29 2:13 | 破宝

to lhq:
不知道是哪句话中的措辞不严密，敬请指出。
按照.net 编码指南，事件处理程序的两个参数：sender 参数的类型应为 object 类型，e 参数的类型应为 EventArgs 类型或者 EventArgs 类型的子类。

RE: .NET 事件模型教程（一）

2005-5-31 11:32 | LHQ

(2)事件委托的"规格"应该是两个参数：第一个参数是 object 类型的 sender，代表发出事件通知的对象

（代码中一般是 this 关键字(VB.NET 中是 Me)）。第二个参数 e，应该是 EventArgs 类型或者从 EventArgs 继承而来的类型；
在最后几句里,第一句我没仔细看现在再看这句没有问题
不过第二个的e还有点问题 我认为e应该是EventArgs 类型或者从 EventArgs 继承而来的类型的对象
好比
class A{}
A a;
那 A 是类型,a 就是对象
e 就是 System.EventArgs 以及它所派生的类的对象
以上属个人意见,如有不当请指出

RE: .NET 事件模型教程（一）

2005-5-31 17:24 | 破宝

to lhq:
多谢指正，纯属文法上的考虑不周。

RE: .NET 事件模型教程（一）

2005-6-2 11:52 | JERRY

弱弱的问题：

在接收到事件后，我用 Label 来显示进度总是没办法显示，只有在事件完成后显示出一个 100%，而没办法在运行事件过程中显示现在运行到百分之多少了，用了 StatusBar 没这个问题。

StatusBar 的 Text 和 Label 的 Text 属性不一样吗？

RE: .NET 事件模型教程（一）

2005-6-2 13:53 | JERRY

另外，我用 progressbar 也不能正确显示当前进度。

好象正在运行的任务也会使窗体陷入无法响应的状态。

有没有办法使进程在运行，而窗体是有响应的：可以拖拽，可以最小话、最大化?

RE: .NET 事件模型教程（一）

2005-7-21 18:32 | LOVALING

看完了，总的来说感觉 C#的看起来最顺眼，个人观点，呵呵，
向"\.NET Framework 类库设计指南"靠拢，标准实现 这一节我不是十分认同这样的做法(虽然是设计指南)，本来一个简单的事情被搞得复杂了。第一个 sender 参数是必要的，使用过委托的都会有这样的需要，完全解除了设计上的藕合，结果丢失了一些参数，不得不以这样的方式来补偿。

没有必要把一个事件的参数包装到一个类里面，也许这样看起来所有的事件都有一样的外观，但我认为不如直接传递参数来得方便。如下：

namespace percyboy.EventModelDemo.Demo1E

```
{
public class Worker
{
private const int MAX = 10000;

public delegate void StartWorkEventHandler(object sender, int totalUnits);
public delegate void RateReportEventHandler(object sender, double rate);
public delegate void EndWorkEventHandler(object sender);

public event StartWorkEventHandler StartWork;
public event EndWorkEventHandler EndWork;
public event RateReportEventHandler RateReport;

public Worker()
{
}

public void DoLongTimeTask()
{
int i;
bool t = false;
double rate;

StartWork(this, MAX);

for (i = 0; i <= MAX; i++)
{
Thread.Sleep(1);
t = !t;
rate = (double)i / (double)MAX;

RateReport(this, rate);
}

EndWork(this);
}
}
}
```

sender 可以保留。不过派生一个类我觉得是一种过时的老的作法，既然有简单的，为何还要用那么复杂的呢，.net 里面这样也保证不会有问题。

可能是想通过一个类型来增强静态编译时的检错吧，防止有相同参数类型的事件接口绑定错了，猜想。

RE: .NET 事件模型教程（一）

2005-7-21 20:58 | 鐮村疊

to lovaling:

我的观点是"入乡随俗"。

很多从 Java 转 C# 的人在书写代码时，把类、方法、属性等等的命名，按照 Java 的命名规则去做，这样做虽然没什么坏处，但看起来总是感觉不伦不类的。所以我的观点是"入乡随俗"，做 .NET 就按微软给大家的约定，没什么不好；做 Java 就看 Sun 的编码规范。这样做出来的东西，感觉才是那个"味道"。

另外，关于事件参数都从 EventArgs 继承这一点，我认为是有好处的，不过好处不是定义事件的一方，而是调用事件的一方。

比如，我们在 VS.NET 中写代码时，如果是某事件 handler 的代码，你可以直接写 e ，然后一"点"就把所有跟此事件相关的参数成员点出来了，使用起来还是很方便。

我们在使用微软提供的标准类库时，形成了这样的习惯，那么在定义自己的事件时，没必要一定自创一套。标准类库的事件遵循一套标准，你自定义的遵循另一套标准，这样感觉还是会带来一定程度的混乱和迷惑。

RE: .NET 事件模型教程（一）

2005-9-22 15:32 | PUBLIC

在 委托（delegate）简介 一节中 "顺带地，既然完全面向对象，其实委托本身也是一种对象。" 一句话题出疑义，我好像记得： 委托（delegate）是一种类(class)。

RE: .NET 事件模型教程（一）

2005-11-5 17:07 | SUNW

感谢楼上的,终于弄懂了 C#里面的事件代理.

RE: .NET 事件模型教程（一）

2006-2-13 14:26 | YAO

好文章.以前不清楚的概念现在清楚了.

RE: .NET 事件模型教程（一）

2006-3-7 19:22 | WQXH

确实不错,以前比较模糊的概念现在越来越清楚了.
希望破宝写出更加好的文章.

RE: .NET 事件模型教程（一）

2006-7-2 12:59 | 野风

不错，很喜欢你的文章，有独道的见解...

.NET 事件模型教程（一）

2006-8-29 11:52 | AFTER_

.NET 事件模型教程（一）

目录

事件、事件处理程序概念
问题描述：一个需要较长时间才能完成的任务
高耦合的实现
事件模型的解决方案，简单易懂的 VB.NET 版本
委托（delegate）简介
C# 实现
向“.NET Framework 类库设计指南”靠拢，标准实现
事件、事件处理程序概念

RE: .NET 事件模型教程（一）

2006-9-8 8:30 | 蛋蛋

太好了！佩服

RE: .NET 事件模型教程（一）

2006-11-26 8:45 | ZHANG

```
public class Worker
{
private const int MAX = 10000;

public class StartWorkEventArgs : EventArgs
{
private int totalUnits;

public int TotalUnits
{
get { return totalUnits; }
}

public StartWorkEventArgs(int totalUnits)
{
this.totalUnits = totalUnits;
}
}

public class RateReportEventArgs : EventArgs
{
private double rate;

public double Rate
{
get { return rate; }
}
```

```
public RateReportEventArgs(double rate)
{
this.rate = rate;
}
}
```

感觉 public class StartWorkEventArgs：EventArgs 和 public class RateReportEventArgs：EventArgs
被嵌入 public class Worker 类内部了,是不是应该移出来，结构更清楚些。

回复：.NET 事件模型教程（一）

2006-12-27 17:51 | 我考百试通

.NET 事件模型和 Java 事件模型的对比

转载：.NET 事件模型教程（一）

2007-2-2 11:18 | 狂风

源文来源：http://blog.joycode.com/percyboy/archive/2005/01/22/43433.aspx.NET 事件模型教程（一） 目录 事件、事件...

FJTREQJJ

2007-2-9 21:39 | FJTREQJJ

jylxsuia btxcyibl http://alnqrxwg.com/ jsyeamnk hlqpsxzx [URL=http://tbrbhpom.com/]jewjulvr[/URL]

回复：.NET 事件模型教程（一）

2007-2-13 20:39 | LIUYUANBO

非常佩服！写的太好了

回复：.NET 事件模型教程（一）

2007-4-22 2:42 | 无情人

DD

回复：.NET 事件模型教程（一）

2007-4-25 15:09 | X2

受益匪浅！感动～～

回复：.NET 事件模型教程（一）

2007-4-25 15:10 | X2

受益匪浅！
强烈的感谢作者！
感动～～

.NET 事件模型教程（二）

目录

- 属性样式的事件声明
- 单播事件和多播事件
- 支持多播事件的改进

属性样式的事件声明

在第一节中，我们讨论了 .NET 事件模型的基本实现方式。这一部分我们将学习 C# 语言提供的高级实现方式：使用 add/remove 访问器声明事件。（注：本节内容不适用于 VB.NET。）

我们再来看看上一节中我们声明事件的格式：

```
        public event [委托类型] [事件名称];
```

这种声明方法，类似于类中的字段（field）。无论是否有事件处理程序挂接，它都会占用一定的内存空间。一般情况中，这样的内存消耗或许是微不足道的；然而，还是有些时候，内存开销会变得不可接受。比如，类似 System.Windows.Forms.Control 类型具有五六十个事件，这些事件并非每次都会挂接事件处理程序，如果每次都无端的多处这么多的内存开销，可能就无法容忍了。

好在 C# 语言提供了"属性"样式的事件声明方式：

```
        public event [委托类型] [事件名称]
        {
            add { .... }
            remove { .... }
        }
```

如上的格式声明事件，具有 add 和 remove 访问器，看起来就像属性声明中的 get 和 set 访问器。使用特定的存储方式（比如使用 Hashtable 等集合结构），通过 add 和 remove 访问器，自定义你自己的事件处理程序添加和移除的实现方法。

Demo 1G："属性"样式的事件声明。我首先给出一种实现方案如下（此实现参考了 .NET Framework SDK 文档中的一些提示）（限于篇幅，我只将主要的部分贴在这里）：

```
        public delegate void StartWorkEventHandler(object sender,
StartWorkEventArgs e);
        public delegate void RateReportEventHandler(object sender,
RateReportEventArgs e);

        // 注意：本例中的实现，仅支持“单播事件”。
        // 如需要“多播事件”支持，请参考 Demo 1H 的实现。
```

```
        // 为每种事件生成一个唯一的 object 作为键
        static readonly object StartWorkEventKey = new object();
        static readonly object EndWorkEventKey = new object();
        static readonly object RateReportEventKey = new object();

        // 使用 Hashtable 存储事件处理程序
        private Hashtable handlers = new Hashtable();

        // 使用 protected 方法而没有直接将 handlers.Add /
handlers.Remove
        // 写入事件 add / remove 访问器，是因为：
        // 如果 Worker 具有子类的话，
        // 我们不希望子类可以直接访问、修改 handlers 这个 Hashtable。
        // 并且，子类如果有其他的事件定义，
        // 也可以使用基类的这几个方法方便的增减事件处理程序。
        protected void AddEventHandler(object eventKey, Delegate
handler)
        {
            lock(this)
            {
                if (handlers[ eventKey ] == null)
                {
                    handlers.Add( eventKey, handler );
                }
                else
                {
                    handlers[ eventKey ] = handler;
                }
            }
        }

        protected void RemoveEventHandler(object eventKey)
        {
            lock(this)
            {
                handlers.Remove( eventKey );
            }
        }

        protected Delegate GetEventHandler(object eventKey)
        {
            return (Delegate) handlers[ eventKey ];
        }
```

```
        // 使用了 add 和 remove 访问器的事件声明
        public event StartWorkEventHandler StartWork
        {
            add { AddEventHandler(StartWorkEventKey, value); }
            remove { RemoveEventHandler(StartWorkEventKey); }
        }

        public event EventHandler EndWork
        {
            add { AddEventHandler(EndWorkEventKey, value); }
            remove { RemoveEventHandler(EndWorkEventKey); }
        }

        public event RateReportEventHandler RateReport
        {
            add { AddEventHandler(RateReportEventKey, value); }
            remove { RemoveEventHandler(RateReportEventKey); }
        }

        // 此处需要做些相应调整
        protected virtual void OnStartWork( StartWorkEventArgs e )
        {
            StartWorkEventHandler handler =
                (StartWorkEventHandler)
GetEventHandler( StartWorkEventKey );
            if (handler != null)
            {
                handler(this, e);
            }
        }

        protected virtual void OnEndWork( EventArgs e )
        {
            EventHandler handler =
                (EventHandler) GetEventHandler( EndWorkEventKey );

            if (handler != null)
            {
                handler(this, e);
            }
        }

        protected virtual void OnRateReport( RateReportEventArgs e )
        {
```

```
            RateReportEventHandler handler =
                (RateReportEventHandler)
GetEventHandler( RateReportEventKey );

            if (handler != null)
            {
                handler(this, e);
            }
        }

        public Worker()
        {
        }

        public void DoLongTimeTask()
        {
            int i;
            bool t = false;
            double rate;

            OnStartWork(new StartWorkEventArgs(MAX) );

            for (i = 0; i <= MAX; i++)
            {
                Thread.Sleep(1);
                t = !t;
                rate = (double)i / (double)MAX;

                OnRateReport( new RateReportEventArgs(rate) );
            }

            OnEndWork( EventArgs.Empty );
        }
```

细细研读这段代码，不难理解它的算法。这里，使用了名为 handlers 的 Hashtable 存储外部挂接上的事件处理程序。每当事件处理程序被"add"，就把它加入到 handlers 里存储；相反 remove 时，就将它从 handlers 里移除。这里取 event 的 key （开始部分为每一种 event 都生成了一个 object 作为代表这种 event 的 key）作为 Hashtable 的键。

[TOP]

单播事件和多播事件

在 Demo 1G 给出的解决方案中，你或许已经注意到：如果某一事件被挂接多次，则后挂接的事件处理程序，将改写先挂接的事件处理程序。这里就涉及到一个概念，叫"单播事件"。

所谓单播事件，就是对象（类）发出的事件通知，只能被外界的某一个事件处理程序处理，而不能被多个事件处理程序处理。也就是说，此事件只能被挂接一次，它只能"传播"到一个地方。相对的，就有"多播事件"，对象（类）发出的事件通知，可以同时被外界不同的事件处理程序处理。

打个比方，上一节开头时张三大叫一声之后，既招来了救护车，也招来了警察叔叔（问他是不是回不了家了），或许还有电视转播车（现场直播、采访张三为什么大叫，呵呵）。

多播事件会有很多特殊的用法。如果以后有机会向大家介绍 Observer 模式，可以看看 Observer 模式中是怎么运用多播事件的。（注：经我初步测试，字段形式的事件声明，默认是支持"多播事件"的。所以如果在事件种类不多时，我建议你采用上一节中所讲的字段形式的声明方式。）

[TOP]

支持多播事件的改进

Demo1H，支持多播事件。为了支持多播事件，我们需要改进存储结构，请参考下面的算法：

```
        public delegate void StartWorkEventHandler(object sender,
StartWorkEventArgs e);
        public delegate void RateReportEventHandler(object sender,
RateReportEventArgs e);

        // 为每种事件生成一个唯一的键
        static readonly object StartWorkEventKey = new object();
        static readonly object EndWorkEventKey = new object();
        static readonly object RateReportEventKey = new object();

        // 为外部挂接的每一个事件处理程序，生成一个唯一的键
        private object EventHandlerKey
        {
            get { return new object(); }
        }

        // 对比 Demo 1G，
        // 为了支持“多播”，
        // 这里使用两个 Hashtable：一个记录 handlers，
        // 另一个记录这些 handler 分别对应的 event 类型（event 的类型
用各自不同的 eventKey 来表示）。
        // 两个 Hashtable 都使用 handlerKey 作为键。
```

```
        // 使用 Hashtable 存储事件处理程序
        private Hashtable handlers = new Hashtable();
        // 另一个 Hashtable 存储这些 handler 对应的事件类型
        private Hashtable events = new Hashtable();

        protected void AddEventHandler(object eventKey, Delegate
handler)
        {
            // 注意添加时，首先取了一个 object 作为 handler 的 key，
            // 并分别作为两个 Hashtable 的键。

            lock(this)
            {
                object handlerKey = EventHandlerKey;
                handlers.Add( handlerKey, handler );
                events.Add( handlerKey, eventKey);
            }
        }

        protected void RemoveEventHandler(object eventKey, Delegate
handler)
        {
            // 移除时，遍历 events，对每一个符合 eventKey 的项，
            // 分别检查其在 handlers 中的对应项，
            // 如果两者都吻合，同时移除 events 和 handlers 中的对应项。
            //
            // 或许还有更简单的算法，不过我一时想不出来了 ： （

            lock(this)
            {
                foreach ( object handlerKey in events.Keys)
                {
                    if (events[ handlerKey ] == eventKey)
                    {
                        if ( (Delegate)handlers[ handlerKey ] ==
handler )
                        {
                            handlers.Remove( handlers[ handlerKey ] );
                            events.Remove( events[ handlerKey ] );
                            break;
                        }
                    }
                }
            }
```

```
        }

        protected ArrayList GetEventHandlers(object eventKey)
        {
            ArrayList t = new ArrayList();

            lock(this)
            {
                foreach ( object handlerKey in events.Keys )
                {
                    if ( events[ handlerKey ] == eventKey)
                    {
                        t.Add( handlers[ handlerKey ] );
                    }
                }
            }

            return t;
        }

        // 使用了 add 和 remove 访问器的事件声明
        public event StartWorkEventHandler StartWork
        {
            add { AddEventHandler(StartWorkEventKey, value); }
            remove { RemoveEventHandler(StartWorkEventKey, value); }
        }

        public event EventHandler EndWork
        {
            add { AddEventHandler(EndWorkEventKey, value); }
            remove { RemoveEventHandler(EndWorkEventKey, value); }
        }

        public event RateReportEventHandler RateReport
        {
            add { AddEventHandler(RateReportEventKey, value); }
            remove { RemoveEventHandler(RateReportEventKey, value); }
        }

        // 此处需要做些相应调整
        protected virtual void OnStartWork( StartWorkEventArgs e )
        {
            ArrayList handlers = GetEventHandlers( StartWorkEventKey );
```

```
            foreach(StartWorkEventHandler handler in handlers)
            {
                handler(this, e);
            }
        }

        protected virtual void OnEndWork( EventArgs e )
        {
            ArrayList handlers = GetEventHandlers( EndWorkEventKey );

            foreach(EventHandler handler in handlers)
            {
                handler(this, e);
            }
        }

        protected virtual void OnRateReport( RateReportEventArgs e )
        {
            ArrayList handlers =
GetEventHandlers( RateReportEventKey );

            foreach(RateReportEventHandler handler in handlers)
            {
                handler(this, e);
            }
        }
```

上面给出的算法，只是给你做参考，应该还有比这个实现更简单、更高效的方式。

为了实现“多播事件”，这次使用了两个 Hashtable：一个存储“handlerKey - handler”对，一个存储“handlerKey - eventKey”对。相信通过仔细研读，你可以读懂这段代码。我就不再赘述了。

[TOP]

2005 年 1 月 22 日 18:35 - (阅读:6119;评论:13)

评论

RE: .NET 事件模型教程（二）

2005-1-25 21:51 | HOO

good

对多播事件的一点意见。

2005-1-31 11:25 | WANG_SOLARIS

看了一下对多播事件的处理方式，总体思路值得肯定，但在此处对用 Hashtable 来存储键值对觉得有些不妥。

一般按照传统采用非静态成员来标识事件类型的方式，当在客户端为一个事件预定多个事件处理函数的时

候，是按照队列的方式来处理的(即先进先出原则)。
而在你的代码中采用了 Hashtable 就破坏了这个原则，因为对 Hashtable 的遍历并不是按照插入时的顺序进行的(见上面对 events 的遍历)。所以我建议换成其它支持按插入时顺序进行遍历的集合类型，比如 ListDictionary 是个选择，不过当事件很多而对性能要求又很高时，需考虑其它实现。(当然上面程序中的 handlers 仍然可以使用 Hashtable)

RE: .NET 事件模型教程（二）

2005-1-31 11:51 | 破宝

谢谢 wang_solaris 的建议！

我正在准备重写这一部分，因为已经有人给我指出了不确切的地方：
其实委托可以是多路的
这样的话，就没有必要分别存储多个委托，
而可以直接挂接在同一个委托实例上。

就是说，委托的一个实例可以同时挂接多个函数，
委托是具有 +=，-= 运算符的。
这一点，我写文章时不了解，给大家介绍的方法其实走了弯路。
所以我正在准备重写这一部分，暂时因为年关太忙无法马上动笔，请诸位见谅！

.NET 事件模型教程

2005-2-4 11:34 | MOREPOWER

Ping Back 来自：blog.csdn.net

RE: .NET 事件模型教程（二）

2005-5-11 14:37 | GAOFAN

代码下不了啊，谁有能否给我一份，感激不尽。。
gaofan628@yahoo.com.cn

RE: .NET 事件模型教程（二）

2005-8-12 15:35 | AYONGWUST

A 对象能不能伪装 B 对象发出 B 对象的事件通知。

.NET 事件模型教程（二）

2006-8-30 15:53 | JELINK

framework

回复: .NET 事件模型教程（二）

2006-12-27 17:13 | 我考百试通

.NET 事件模型和 Java 事件模型的对比

回复: .NET 事件模型教程（二）

2006-12-27 17:50 | 我考百试通

.NET 事件模型和 Java 事件模型的对比

回复: .NET 事件模型教程（二）

2007-5-10 10:55 | 座看云起

我的理解，未经证实：
挂接事件时将运算符由"+="改为"="，事件应该就由多播变成单播了。
也就是说.NET 代理机制本身是以某种方式实现了一个队列。

回复: .NET 事件模型教程（二）

2007-5-10 12:11 | 坐看云起

刚做了实验，挂接事件时使用"="是不允许的。呵呵。

.NET技术-.NET理论资料-.NET理论资料

2007-7-4 14:00 | JASONLI

综合：http://210.27.12.83/jingpin_mms.asp

回复: .NET 事件模型教程（二）

2007-7-11 13:33 | ILEX

谢谢，怎么一直未见你的改进版呢

.NET 事件模型教程（三）

通过前两节的学习，你已经掌握了 .NET 事件模型的原理和实现方式。这一节我将介绍两个替代方案，这些方案并不是推荐采用的，请尽量采用事件模型去实现。另外，在本节末尾，有一段适合熟悉 Java 语言的读者阅读，讨论了 .NET 和 Java 在"事件模型"方面的差异。

目录

- 使用接口实现回调
- .NET 事件模型和 Java 事件模型的对比

使用接口实现回调

事件模型其实是**回调函数**的一种特例。像前面的例子，Form1 调用了 Worker，Worker 反过来（通过事件模型）让 Form1 改变了状态栏的信息。这个操作就属于回调的一种。

在".NET Framework 类库设计指南"中提到了："委托、接口和事件允许提供回调功能。每个类型都有自己特定的使用特性，使其更适合特定的情况。"（参见 本地 SDK 版本，在线 MSDN 版本）

事件模型中，事实上也应用了委托来实现回调，可以说，事件模型是委托回调的一个特例。如果有机会，我会在关于多线程的教程中介绍委托回调在多线程中的应用。

这里我先来看看，如何使用接口实现回调功能，以达到前面事件模型实现的效果。

Demo 1I：使用接口实现回调。

```
using System;
using System.Threading;
using System.Collections;

namespace percyboy.EventModelDemo.Demo1I
{
    // 注意这个接口
    public interface IWorkerReport
    {
        void OnStartWork(int totalUnits);
        void OnEndWork();
        void OnRateReport(double rate);
    }

    public class Worker
    {
        private const int MAX = Consts.MAX;
        private IWorkerReport report = null;
```

```
        public Worker()
        {
        }

        // 初始化时同时指定 IWorkerReport
        public Worker(IWorkerReport report)
        {
            this.report = report;
        }

        // 或者初始化后，通过设置此属性指定
        public IWorkerReport Report
        {
            set { report = value; }
        }

        public void DoLongTimeTask()
        {
            int i;
            bool t = false;
            double rate;

            if (report != null)
            {
                report.OnStartWork( MAX );
            }

            for (i = 0; i <= MAX; i++)
            {
                Thread.Sleep(1);
                t = !t;
                rate = (double)i / (double)MAX;

                if (report != null)
                {
                    report.OnRateReport( rate );
                }
            }

            if ( report != null)
            {
                report.OnEndWork();
            }
```

```
        }
    }
}
```

你可以运行编译好的示例，它可以完成和前面介绍的事件模型一样的工作，并保证了耦合度没有增加。调用 Worker 的 Form1 需要做一个 IWorkerReport 的实现：

```
        private void button1_Click(object sender, System.EventArgs e)
        {
            statusBar1.Text = "开始工作 ....";
            this.Cursor = Cursors.WaitCursor;

            long tick = DateTime.Now.Ticks;

            Worker worker = new Worker();

            // 指定 IWorkerReport
            worker.Report = new MyWorkerReport(this);

            worker.DoLongTimeTask();

            tick = DateTime.Now.Ticks - tick;
            TimeSpan ts = new TimeSpan(tick);

            this.Cursor = Cursors.Default;
            statusBar1.Text = String.Format("任务完成，耗时 {0} 秒。",
ts.TotalSeconds);
        }

        // 这里实现 IWorkerReport
        private class MyWorkerReport : IWorkerReport
        {
            public void OnStartWork(int totalUnits)
            {
            }

            public void OnEndWork()
            {
            }

            public void OnRateReport(double rate)
            {
                parent.statusBar1.Text = String.Format("已完成
{0:P0} ....", rate);
```

```
            }

            private Form1 parent;

            public MyWorkerReport(Form1 form)
            {
                this.parent = form;
            }
        }
```

你或许已经觉得这种实现方式，虽然 Worker 类"里面"可能少了一些代码，却在调用时增加了很多代码量。从重复使用的角度来看，事件模型显然要更方便调用。另外，从面向对象的角度，我觉得理解了事件模型的原理之后，你会觉得"事件"会更亲切一些。

另外，IWorkerReport 中包含多个方法，而大多时候我们并不是每个方法都需要，就像上面的例子中那样，OnStartWork 和 OnEndWork 这两个都是空白。如果接口中的方法很多，也会给调用方增加更多的代码量。

下载的源代码中还包括一个 Demo 1J，它和 Worker 类一起，提供了一个 IWorkerReport 的默认实现 WorkerReportAdapter（每个方法都是空白）。这样，调用方只需要从 WorkerReportAdapter 继承，重写其中需要重写的方法，这样会减少一部分代码量。但我觉得仍然是很多。

注意，上述的代码，套用（仅仅是套用，因为它不是事件模型）"单播事件"和"多播事件"的概念来说，它只能支持"单播事件"。如果你想支持"多播事件"，我想你可以考虑加入 AddWorkerReport 和 RemoveWorkerReport 方法，并使用 Hashtable 等数据结构，存储每一个加入的 IWorkerReport。

[TOP]

.NET 事件模型和 Java 事件模型的对比

（我对 Java 语言的了解不是很多，如果有误，欢迎指正！）

.NET 的事件模型，对于 C#/VB.NET 两种主流语言来说，是在语言层次上实现的。C# 提供了 event 关键字，VB.NET 提供了 Event，RaiseEvent 关键字。像前面两节所讲的那样，它们都有各自的声明事件成员的语法。而 Java 语言本身是没有"事件"这一概念的。

从面向对象理论来看，.NET 的一个类（或类的实例：对象），可以拥有：字段、属性、方法、事件、构造函数、析构函数、运算符等成员类型。在 Java 中，类只有：字段、方法、构造函数、析构函数、运算符。Java 的类中没有属性和事件的概念。（虽然 Java Bean 中将 getWidth、setWidth 的两个方法，间接的转换为一个 Width 属性，但 Java 依然没有把"属性"作为一个语言层次的概念提出。）总之，在语言层次上，Java 不支持事件。

Java Swing 是 Java 世界中常用的制作 Windows 窗体程序的一套 API。在 Java Swing 中有一套事件模型，来让它的控件（比如 Button 等）拥有事件机制。

Swing 事件模型，有些类似于本节中介绍的接口机制。它使用的接口，诸如 ActionListener、KeyListener、MouseListener（注意：按照 Java 的命名习惯，接口命名不用前缀 I）等；它同时也提供一些接口的默认实现，如 KeyAdapter，MouseAdapter 等，使用方法大概和本节介绍的类似，它使用的是 addActionListener/removeActionListener，addKeyListener/removeKeyListener，addMouseListener/removeMouseListener 等方法，来增减这些接口的。

正像本节的例子那样，使用接口机制的 Swing 事件模型，需要书写很多的代码去实现接口或者重写 Adapter。而相比之下，.NET 事件模型则显得更为轻量级，所需的挂接代码仅一行足矣。

另一方面，我们看到 Swing 的命名方式，将这些接口都命名为 Listener，监听器；而相比之下，.NET 事件模型中，对事件的处理被称为 handler，事件处理程序。一个采用"监听"，一个是"处理"，我认为这体现了一种思维上的差异。

还拿张三大叫的例子来讲，"处理"模型是说：当张三大叫事件发生时，外界对它做出处理动作（handle this event）；监听，则是外界一直"监听"着张三的一举一动（listening），一旦张三大叫，监听器就被触发。处理模型是以张三为中心的思维，监听模型则是以外部环境为中心的思维。

[TOP]

2005 年 1 月 22 日 18:37 - (阅读:4524;评论:12)

评论

RE: .NET 事件模型教程（三）

2005-1-25 21:52 | HOO

不错，收藏先

.NET 事件模型教程

2005-2-4 11:34 | MOREPOWER

Ping Back 来自：blog.csdn.net

RE: .NET 事件模型教程（三）

2005-4-22 16:15 | AA

写得不错

RE: .NET 事件模型教程（三）

2005-7-20 13:26 | COOLLOVE

如果我在 PicBox 上绘几个矩形如何用自定义事件（例如在 mousedown 事件中）让它们自己区分，并且回应？？
估计没有人试过。

RE: .NET 事件模型教程（三）

2006-6-19 10:23 | AA

very good!thanks a lot!

RE: .NET 事件模型教程（三）

2006-6-30 10:30 | CREEKSUN

写的很好阿,就是看不懂.我想问您:我想在 asp.net 网页中调用一个别人编写的成熟的水质模型,该如何实现？求救!

.NET 事件模型教程（三）

2006-8-30 15:54 | JELINK

framework

回复: .NET 事件模型教程（三）

2006-12-27 17:13 | 我考百试通

.NET 事件模型和 Java 事件模型的对比

回复: .NET 事件模型教程（三）

2007-1-12 13:33 | MIKE_D

最近正好在做一个项目
用 C#，但是对 C#不是很熟悉，尤其是事件模型，怎么也不懂
今天看了觉得相当不错
现在就在我的项目中实验呢

回复: .NET 事件模型教程（三）

2007-3-15 17:27 | 飞扬跋扈

好文!!!

回复: .NET 事件模型教程（三）

2007-3-15 17:27 | 飞扬跋扈

好文!!! 好文!!! 好文!!! 好文!!! 好文!!!

.NET 2.0 中真正的多线程实例

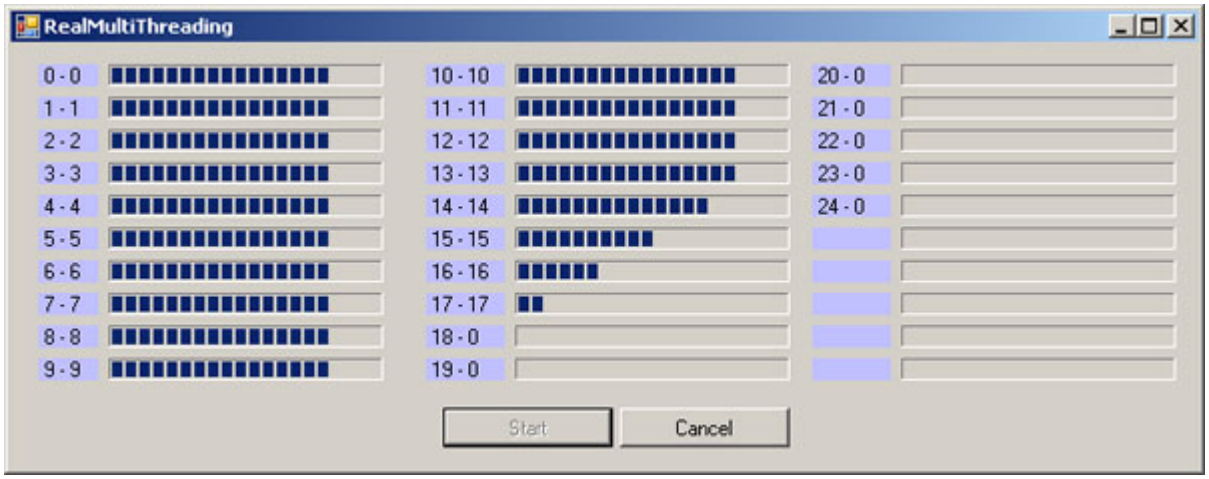
Real Multi-threading in .NET 2.0

remex1980 翻译于 2007-5-16 20:43:21

原作者： Jose Luis Latorre

原文地址： http://www.codeproject.com/useritems/RealMultiThreading.asp

多线程实例源代码（保留原文处链接）
http://www.codeproject.com/useritems/RealMultiThreading/RealMultiThreading_src.zip



简介

多线程总是那么让人振奋。大家都希望能够同时处理很多事情，不过如果我们没有正确的硬件的话，我们很难达到这点。到目前为止，我们所做的只是分开 CPU 使用较多的工作，使其为后台进程，这样可以使得界面上不被阻塞。

不过我希望能够得到更好的效果，并充分利用当前最新的多 CPU 效能。因此，我将写一个真正的多线程实例，将会有多个线程作为后台线程在运行。

这就是这篇文章将要写的，不得不说的是，最终的结果实在是让我很激动。希望你也能够发觉它的用处。

在有 4 个 CPU 的多 CPU 服务器上，我得到了 280％的效果（测试的是 CPU 型的任务），在一些非 CPU 占用较多的任务中，它可以提高到 500% 到 1000%的性能。

背景

网上也有不少介绍.Net 2.0 下的多线程的文章，应该说，我从它们中受益颇多。我正在使用的是 BackgroundWorker .Net 2.0 组件（不过也有实现在.net 1.1 下的代码）。

这里，我列出一些有用的文章链接：
来自Paul Kimmel的很好的介绍性文章
http://www.informit.com/articles/article.asp?p=459619&seqNum=5&rl=1
来自Juval Löwy的介绍性文章 http://www.devx.com/codemag/Article/20639/1954?pf=true
（必看）Joseph Albahari的C#中使用线程 http://www.albahari.com/threading/part3.html
Michael Weinhardt写的在Windows Forms 2.0 中一个简单安全的多线程，我使用了这个网页中的CPU密集型任务，这是他从Chris Sell的文章中引用的。
http://www.mikedub.net/mikeDubSamples/SafeReallySimpleMultithreadingInWindowsForms20/SafeReallySimpleMultithreadingInWindowsForms20.htm

如果你对多线程世界仍然不是特别熟悉或者希望了解最新的.Net 2.0 的 BackgroundWorker 组件，那么应该好好读读上面的文章。

提出的问题

任何一个任务……无论是 CPU 密集型还是普通的任务：

CPU 密集型：它可以分成一个、两个或多个线程，每个线程会占用一个 CPU（这样就使得程序的性能翻番）

普通任务：每一个顺序执行的普通任务，在进行数据存储或使用一个 web service 的时候都会有一些延迟。所有的这些，都意味着这些没有使用的时间对于用户或任务本身来说有了浪费。这些时间将被重新安排，并将被并行的任务使用，不会再丢失。也就是说，如果有 100 个 100ms 延迟的任务，它们在单线程模型和 20 个线程模型的性能差距会达到 1000％。

我们说，如果要处理一个创建一个网站多个块的任务，不是顺序的执行，而是花 1－4 秒钟把所有的 section 创建好；商标，在线用户，最新文章，投票工具等等…… 如果我们能够异步地创建它们，然后在发送给用户，会怎么样？我们就会节省很多 webservice 的调用，数据库的调用，许多宝贵的时间……这些调用会更快地执行。看上去是不是很诱人？

解决方案如下：

调用 BackgroundWorker，正如我们想要的那样，我们会继承它。后台 worker 会帮助我们建立一个“Worker”，用于异步地做一个工作。

我们想做的是建立一个工厂 Factory（只是为了面向对象的设计，于设计模式无关），任务会放在在这个 Factory 中执行。这意味着，我们将有一类的任务，一些进程，一些知道如何执行任务的 worker。

当然我们需要一个负责分配任务给这些 worker 的 manager，告诉这些 worker 当它们做完一步或全部时，做什么事情。当然，我们也需要 manager 能够告诉 worker 停止当前的任务。它们也需要休息啊：）当 manager 说停止的时候，它们就应该停止。

我们将会从底至上地解释这些，首先从 Worker 说起，然后再继续 Manager。

Worker

它是 Background worker 的继承类，我们构建一个构造函数，并分配两个 BackgroundWorker 的属性，分别是 WorkerReportsProgress 和 WorkerSupportsCancellation，它们的功能就向其名字的意义一样：报告进度，停止任务。每个 Worker 还有一个 id，Manager 将会通过这个 id 控制它们。

```
public class MTWorker : BackgroundWorker
  {
    #region Private members
    private int _idxLWorker = 0;
    #endregion

    #region Properties
    public int IdxLWorker
    {
      get { return _idxLWorker; }
      set { _idxLWorker = value; }
    }
    #endregion

    #region Constructor
    public MTWorker()
    {
      WorkerReportsProgress = true;
      WorkerSupportsCancellation = true;
    }
    public MTWorker(int idxWorker)
      : this()
    {
      _idxLWorker = idxWorker;
    }
    #endregion
```

另外，我们将重载 BackgroundWorker 的一些函数。事实上，最有意思的是，究竟谁在做真正的工作？它就是 OnDoWork，当我们 invoke 或者启动多线程的时候，它就会被调用。在这里，我们启动任务、执行任务、取消和完成这个任务。
我加了两个可能的任务，一个是普通型的，它会申请并等待文件系统、网络、数据库或 Webservices 的调用。另一个是 CPU 密集型的任务：计算 PI 值。你可以试试增加或减少线程数量后，增加或是减少的延迟（我的意思是增减 Worker 的数量）

OnDoWork 方法的代码

```
protected override void OnDoWork(DoWorkEventArgs e)
{
   //Here we receive the necessary data for doing the work...
   //we get an int but it could be a struct, class, whatever..
   int digits = (int)e.Argument;
   double tmpProgress = 0;
   int Progress = 0;
   String pi = "3";

   // This method will run on a thread other than the UI thread.
   // Be sure not to manipulate any Windows Forms controls created
   // on the UI thread from this method.
   this.ReportProgress(0, pi);
   //Here we tell the manager that we start the job..

   Boolean bJobFinished = false;
   int percentCompleteCalc = 0;
   String TypeOfProcess = "NORMAL"; //Change to "PI" for a cpu intensive task
   //Initialize calculations
   while (!bJobFinished)
   {
      if (TypeOfProcess == "NORMAL")
      {
         #region Normal Process simulation, putting a time
               delay to emulate a wait-for-something
         while (!bJobFinished)
         {
            if (CancellationPending)
            {
               e.Cancel = true;
               return; //break
            }
            //Perform another calculation step
            Thread.Sleep(250);
```

```
        percentCompleteCalc = percentCompleteCalc + 10;
        if (percentCompleteCalc >= 100)
          bJobFinished = true;
        else
          ReportProgress(percentCompleteCalc, pi);
      }
      #endregion
    }
    else
    {
      #region  Pi Calculation - CPU intensive job,
             beware of it if not using threading ;) !!
      //PI Calculation
      if (digits > 0)
      {
        pi += ".";
        for (int i = 0; i < digits; i += 9)
        {
          // Work out pi. Scientific bit :-)
          int nineDigits = NineDigitsOfPi.StartingAt(i + 1);
          int digitCount = System.Math.Min(digits - i, 9);
          string ds = System.String.Format("{0:D9}", nineDigits);
          pi += ds.Substring(0, digitCount);

          // Show progress
          tmpProgress = (i + digitCount);
          tmpProgress = (tmpProgress / digits);
          tmpProgress = tmpProgress * 100;
          Progress = Convert.ToInt32(tmpProgress);
          ReportProgress(Progress, pi);
          // Deal with possible cancellation
          if (CancellationPending) //If the manager says to stop, do so..
          {
            bJobFinished = true;
            e.Cancel = true;
            return;
          }
        }
      }
      bJobFinished = true;
      #endregion
    }

  }
```

```
  ReportProgress(100, pi); //Last job report to the manager ;)
  e.Result = pi; //Here we pass the final result of the Job
}
```

Manager

这是一个很有趣的地方，我确信它有很大的改进空间－欢迎任何的评论和改进！它所做的是给每个线程生成和配置一个 Worker，然后给这些 Worker 安排任务。目前，传给 Worker 的参数是数字，但是它能够传送一个包含任务定义的类或结构。一个可能的改进是选择如何做这些内部工作的策略模式。

调用 InitManager 方法配置任务，和它的数量等属性。然后，创建一个多线程 Worker 的数组，配置它们。
配置的代码如下：

```
private void ConfigureWorker(MTWorker MTW)
{
 //We associate the events of the worker
 MTW.ProgressChanged += MTWorker_ProgressChanged;
 MTW.RunWorkerCompleted += MTWorker_RunWorkerCompleted;
}
```

Like this, the Worker's subclassed thread management Methods are linked to the Methods held by the Manager. Note that with a Strategy pattern implemented we could assign these to the proper manager for these methods.
最主要的方法是 AssignWorkers，它会检查所有的 Worker，如果发现没有任务的 Worker，就分配一个任务给它。直到扫描一遍之后，没有发现任何有任务的 Worker，这样就意味这任务结束了。不需要再做别的了！

代码如下：

```
public void AssignWorkers()
{
  Boolean ThereAreWorkersWorking = false;
  //We check all workers that are not doing a job... and assign a new one
  foreach (MTWorker W in _arrLWorker)
  {
    if (W.IsBusy == false)
    {
      //If there are still jobs to be done...
      //we assign the job to the free worker
      if (_iNumJobs > _LastSentThread)
      {
```

```
        //We control the threads associated to a worker
        //(not meaning the jobs done) just 4 control.
        _LastSentThread = _LastSentThread + 1;
        W.JobId = _LastSentThread; //We assign the job number..
        W.RunWorkerAsync(_iPiNumbers); //We pass the parameters for the job.
        ThereAreWorkersWorking = true;
        //We have at least this worker we just assigned the job working..
       }
     }
     else
     {
       ThereAreWorkersWorking = true;
     }
   }

   if (ThereAreWorkersWorking == false)
   {
     //This means that no worker is working and no job has been assigned.
     //this means that the full package of jobs has finished
     //We could do something here...
     Button BtnStart = (Button)FormManager.Controls["btnStart"];
     Button BtnCancel = (Button)FormManager.Controls["btnCancel"];
     BtnStart.Enabled = true;
     BtnCancel.Enabled = false;
     MessageBox.Show("Hi, I'm the manager to the boss (user): " +
                   "All Jobs have finished, boss!!");
   }
}
```

只要有任务完成，这个方法就会被调用。从而，保证所有的任务能够完成。

我们还通过一个属性链接到 Form 上，这样我们就能向 UI 上输出我们想要的任何消息了。当然，你可能想链接到其它的一些类，不过这是最基本最通用的。

Well… improving it we could get a BackgroundManager for all our application needs..

界面

连接到界面上，并不是最主要的功能。这一部分的代码量非常少，也很简单：在 Manager 中添加一个引用，并在 form 的构造函数中配置它。
在一个按钮中，执行 Manager 类的 LaunchManagedProcess 方法。

```
private MTManager LM;
```

```
public Form1()
{
   InitializeComponent();
   LM = new MTManager(this, 25);
   LM.InitManager();
}

private void btnStart_Click(object sender, EventArgs e)
{
   btnStart.Enabled = false;
   btnCancel.Enabled = true;
   LM.LaunchManagedProcess();
}

private void btnCancel_Click(object sender, EventArgs e)
{
   LM.StopManagedProcess();
   btnCancel.Enabled = false;
   btnStart.Enabled = true;
}
```

（下面的自己看喽：）

Trying it!

This is the funniest part, changing the properties of how many threads to run simultaneously and how many Jobs to be processed and then try it on different CPU's… ah, and of course, change the calculation method from a CPU-intensive task to a normal task with a operation delay...

I would love to know your results and what have you done with this, any feedback would be great!!

Exercises For You…

This is not done! It could be a MultiThreadJob Framework if there is being done the following:

Implement a Strategy pattern that determines the kind of Worker to produce (with a factory pattern) so we will be able to do different kind of jobs inside the same factory.. what about migrating a database and processing each table in a different way… or integrating systems with this engine…

Implement -or extend- the strategy pattern for determining the treatment for the Input data and the result data of the jobs. We could too set-up a factory for getting the classes into a operating environment.

Optimize the AssignWorkers engine – I am pretty sure it can be improved.

Improve the WorkerManager class in order to be able to attach it to another class instead

to only a form.
Send me the code! I Would love to hear from you and what have you done.

About Jose Luis Latorre

Professional developer since 1991, having developed on multiple systems and languages since then, from Unix, as400, lotus notes, flash, javascript, asp, prolog, vb, c++, vb.Net, C#...

Now I'm focused on .Net development, both windows and web with two-three year experience on both and fully up-to date with 2.0 .Net in both Vb and C#
Also have experience with SQL server 2005 and Business Intelligence.

Jose Luis Lives in Barcelona, Spain, with his cat Pancho. To contact Jose Luis, email him at
joslat@gmail.com.

Click here to view Jose Luis Latorre's online profile.

Copyright MSProject 2006-2007. 转载本站文章必须经过作者本人或 管理员的同意

C#中在线程中访问主Form控件的问题

C#不允许直接从线程中访问 Form 里的控件，比如希望在线程里修改 Form 里的一个 TextBox 的内容等等，唯一的做法是使用 Invoke 方法，下面是一个 MSDN 里的 Example，很说明问题：

```
using System;

using System.Drawing;

using System.Windows.Forms;

using System.Threading;


   public class MyFormControl : Form

        {

      public delegate void AddListItem(String myString);

      public AddListItem myDelegate;

      private Button myButton;

      private Thread myThread;

      private ListBox myListBox;

      public MyFormControl()

            {

         myButton = new Button();

         myListBox = new ListBox();

         myButton.Location = new Point(72, 160);

         myButton.Size = new Size(152, 32);

         myButton.TabIndex = 1;
```

```
    myButton.Text = "Add items in list box";

    myButton.Click += new EventHandler(Button_Click);

    myListBox.Location = new Point(48, 32);

    myListBox.Name = "myListBox";

    myListBox.Size = new Size(200, 95);

    myListBox.TabIndex = 2;

    ClientSize = new Size(292, 273);

          Controls.AddRange(new Control[] {myListBox,myButton});

    Text = " 'Control_Invoke' example ";

    myDelegate = new AddListItem(AddListItemMethod);

  }

  static void Main()

       {

    MyFormControl myForm = new MyFormControl();

    myForm.ShowDialog();

  }

  public void AddListItemMethod(String myString)

       {

        myListBox.Items.Add(myString);

  }

  private void Button_Click(object sender, EventArgs e)

       {
```

```
            myThread = new Thread(new ThreadStart(ThreadFunction));

            myThread.Start();

          }

      private void ThreadFunction()
      {
         MyThreadClass myThreadClassObject  = new MyThreadClass(this);
         myThreadClassObject.Run();
      }
   }

   public class MyThreadClass
   {
      MyFormControl myFormControl1;
      public MyThreadClass(MyFormControl myForm)
      {
         myFormControl1 = myForm;
      }
      String myString;

      public void Run()
      {
         for (int i = 1; i <= 5; i++)
         {
            myString = "Step number " + i.ToString() + " executed";
            Thread.Sleep(400);
            // Execute the specified delegate on the thread that owns
            // 'myFormControl1' control's underlying window handle with
            // the specified list of arguments.
            myFormControl1.Invoke(myFormControl1.myDelegate,  new Object[] {myStrin
g});         }
      }
   }
```

BackgroudWorker 范例

在很多场合下，你需要在主(UI)线程中运行一些比较耗时间的任务,比如以下的任务

1 Image downloads

2 Web service invocations

3 File downloads and uploads (including for peer-to-peer applications)

4 Complex local computations

5 Database transactions

6 Local disk access, given its slow speed relative to memory access

这个时候 UI 就会陷入一种假死的状态,会给用户带来一种很不好的体验．如何在这里发挥多线程的优势以改善用户体验？.Net2.0 的 System.ComponentModel.BackgroundWorker 为我们提供了一个很方便的解决方法.

在 vs.net2005 101 sample 中提供了一个计算素数的例子，不过那个例子并没有全面演示 BackgroundWorker 的能力，尤其是没有对线程工作过程(ReportProgress)中的能力做比较好的演示.因此我重新做了一个 Demo.

这个例子很简单，就是将左边列表中的内容移至的右边，用一个进度条来显示移动的进度，当然既然是 BackgroundWorker 这个时候主界面可以进行其他操作.

本文的源代码提供下载，其中有详细注释，所以我在此简要介绍一下需要注意的地方.

BackgroundWorker 主要通过对 DoWork ProgressChanged RunWorkerCompleted 三个事件的处理来完成任务．需要注意在 DoWork 中不能直接操作主界面的元素.比如你在 MainForm 类中启动了一个 BackgroundWorker，在 DoWork 的处理方法中不能直接调用任何 MainForm 中的成员变量．但是在 ProgressChanged 和 RunWorkerCompleted 的事件处理中则无此限制，可以在后台线程中直接调用主线程中的元素，这是 BackgroundWorker 中最有亮点的地方．虽然在 DoWork 的处理方法中不能调用但是它也提供了参数传递的方法,可以间接调用.示例如下：

```
//If your background operation requires a parameter,
//call System.ComponentModel.BackgroundWorker.RunWorkerAsync
//with your parameter. Inside the System.ComponentModel.BackgroundWorker.DoWork
//event handler, you can extract the parameter from the
```

```
31      //System.ComponentModel.DoWorkEventArgs.Argument property.
32      worker.RunWorkerAsync(leftList);
```

```
27       private void worker_DoWork(object sender, DoWorkEventArgs e)
28           {
29               MoveList((BackgroundWorker)sender,e);
30           }
31
32           private void MoveList(BackgroundWorker worker,DoWorkEventArgs e)
33           {   //get leftList in Main UI Thread from arguments
34               IList<string> list = e.Argument as IList<string>;
35               //...
36           }
```

而在 ProgressChanged 和 RunWorkerCompleted 事件的处理方法中则更加简单.

```
27           private void worker_ProgressChanged(object sender, ProgressChangedEventArgs
e)
28           {
29               //Add string to the right listBox, we use rightList in Main UI Thread directly
30               rightList.Add(e.UserState as string);
31           }
```

上述原则可以说是 BackgroundWorker 最需要注意的地方.

另外一个容易被人粗心漏过的地方是有关属性的设置.

如果你要使 BackgroundWorker 支持进度汇报和取消功能别忘了在初始化的时候为下面两个属性赋值.

```
// Specify that the background worker provides progress notifications

        worker.WorkerReportsProgress = true;

        // Specify that the background worker supports cancellation

        worker.WorkerSupportsCancellation = true;
```

其它部分就让大家自己看代码吧.

BackgroundWorker 内部实现是基于 delegate 的异步调用.

dotnet 中一个重要组件－BackgroundWorker - Strive for perfection,Settle for excellence! - 博客园

dotnet 中一个重要组件－BackgroundWorker
最近一直在看 wse3.0，从一个例子中偶然的收获。虽然通过后台操作，从而减少用户交互时的“僵硬”体验一直是每个程序员的追求，在今天这样 ajax 的时代里面更加显的重要。一切为了用户，一切为了更丰富愉快的体验。本文并不是 ajax 相关的东东。伟大的 BackgroundWorker!

BackgroundWorker 类允许您在单独的专用线程上运行操作。耗时的操作（如下载和数据库事务）在长时间运行时可能会导致用户界面 (UI)
似乎处于停止响应状态。如果您需要能进行响应的用户界面，而且面临与这类操作相关的长时间延迟，则可以使用 BackgroundWorker 类方便地解决问题。

您必须非常小心，确保在 DoWork 事件处理程序中不操作任何用户界面对象。而应该通过 ProgressChanged 和
RunWorkerCompleted 事件与用户界面进行通信

使用方式：
1。给组件注册事件处理方法：

```
//正式做事情的地方
backgroundWorker1.DoWork +=
    new DoWorkEventHandler(backgroundWorker1_DoWork);

//任务完称时要做的，比如提示等等
backgroundWorker1.RunWorkerCompleted +=
    new
RunWorkerCompletedEventHandler(backgroundWorker1_RunWorkerCompleted);
```

```
            //任务进行时，报告进度
            backgroundWorker1.ProgressChanged +=
                new
ProgressChangedEventHandler(backgroundWorker1_ProgressChanged);
```

2。添加具体事件处理方法

DoWork 调用 RunWorkerAsync 时发生。

ProgressChanged 调用 ReportProgress 时发生。

RunWorkerCompleted 当后台操作已完成、被取消或引发异常时发生。

```
1//这个例子没有做什么事情，完全是看看效果而已，但同时有个大问题，我也不知道为什么，没有去除僵硬情况。
2namespace BackgroundWorkerTest
3{
4     public partial class Form1 : Form
5     {
6         public Form1()
7         {
8             InitializeComponent();
9             InitialzeBackgroundWorker();
10        }
11
12        private void InitialzeBackgroundWorker()
13        {
14            this.backgroundWorker1.DoWork+=new
DoWorkEventHandler(backgroundWorker1_DoWork);
15            this.backgroundWorker1.ProgressChanged+=new
ProgressChangedEventHandler(backgroundWorker1_ProgressChanged);
16            this.backgroundWorker1.RunWorkerCompleted+=new
RunWorkerCompletedEventHandler(backgroundWorker1_RunWorkerCompleted);
17        }
18
19
20        private void backgroundWorker1_RunWorkerCompleted(object sender,
RunWorkerCompletedEventArgs e)
21        {
22            MessageBox.Show("Completly");
23        }
24
25        private void backgroundWorker1_ProgressChanged(object sender,
ProgressChangedEventArgs e)
26        {
27            this.progressBar1.Value = e.ProgressPercentage;
```

```
28          }
29          private void backgroundWorker1_DoWork(object sender,DoWorkEventArgs e)
30          {
31              e.Result = ComputeFibonacci(this.backgroundWorker1, e);
32          }
33
34          private int ComputeFibonacci(object sender, DoWorkEventArgs e)
35          {
36
37              for (int i = 0; i < 100000; i++)
38              {
39                  if (this.backgroundWorker1.CancellationPending)
40                  {
41                      e.Cancel = true;
42
43                  }
44                  else
45                  {
46                      this.backgroundWorker1.ReportProgress(i);
47                  }
48
49              }
50              return 0;
51
52          }
53
54
55          private void button1_Click(object sender, EventArgs e)
56          {
57              this.backgroundWorker1.RunWorkerAsync();
58          }
59
60          private void button2_Click(object sender, EventArgs e)
61          {
62              this.backgroundWorker1.CancelAsync();
63          }
64      }
65}
```

给出另一种使用：继承 BackgroundWorker:

```
namespace UploadWinClient
{
    /**//// <summary>
    /// Contains common functionality for the upload and download classes
```

```
    /// This class should really be marked abstract but VS doesn't like that
because it can't draw it as a component then :(
    /// </summary>
    public class FileTransferBase : BackgroundWorker
    {
public FileTransferBase()
        {
            base.WorkerReportsProgress = true;
            base.WorkerSupportsCancellation = true;
        }
protected override void Dispose(bool disposing)
        {
            if(this.HashThread != null && this.HashThread.IsAlive)
                this.HashThread.Abort();
            base.Dispose(disposing);
        }

        protected override void OnRunWorkerCompleted(RunWorkerCompletedEventArgs
e)
        {
            if(this.HashThread != null && this.HashThread.IsAlive)
                this.HashThread.Abort();
            base.OnRunWorkerCompleted(e);
        }

        protected override void OnDoWork(DoWorkEventArgs e)
        {
            // make sure we can connect to the web service.   if this step is not
done here, it will retry 50 times because of the retry code
            this.WebService.Ping();
            base.OnDoWork(e);
        }
}//截取的部分代码。
//从中给出我们要 override 的一些方法
```

BackgroundWorker 在长时间的 webservices 中特别有用。

posted on 2006-07-05 10:25 flyingchen 阅读(708) 评论(1) 编辑 收藏 引用 网摘 所属分类: DotNet 技术

评论
关于 net2.0 里面新出现的类 backgroundworker 的应用[TrackBack] 2007-01-11 16:16 天龙
这是一个在.net2.0 里面新出现的类，用于执行后台比较长的任务而又想能和 UI 有点操作的应用里面。普通情况下，你点击一个按钮，去后台执行一个 process，如果你想得到结果，就得等这个 proces...

这是一个在.net2.0 里面新出现的类，用于执行后台比较长的任务而又想能和 UI 有点操作的应用里面。

普通情况下，你点击一个按钮，去后台执行一个 process，如果你想得到结果，就得等这个 process 结束。通常，可以使用异步执行回调来解决这个问题。现在，backgroundworker 给我们实现了这样一种简单的封装，可以把我们的复杂任务交给新的线程去处理，然后继续 UI 线程。等到我们的任务需要通知 UI 做什么事情的时候，可以 report 一下，在其事件里就可以直接使用 UI 控件，而不需要 Control.Invoke 去掉用之。

有这样一个应用：客户需要把大量数据（需要执行 3 天）copy 到另外一个机器，中间想能看到有多少数据被复制/失败等（实时报道）。

在这个例子里面，我们的界面可能非常简单：一个开始按钮，一个结束按钮，一个 richtextBox 来显示运行记录。但是后台执行可能就会比较棘手。如果简单的执行，并且报告，那么整个界面将失去响应（都在同一个线程里面，造成忙碌）。这时候，可以使用这个 backgroundworker 了。它可以在后台执行，并且报告给界面实时信息，界面不会失去响应。

先介绍一下 backgroundworker 的几个属性/方法

.WorkerReportsProgress：是否可以向外报告进度。

.WorkerSupportsCancellation ：是否可以暂停任务

. CancellationPending： 是否正在暂停中

. RunWorkerAsync() ： 开始执行任务。触发 DoWork 事件

. ReportProgress(int percentPrgress,object userState) ： 向外报告进度。触发 ProgressChanged 事件.其中，参数可以在 ProgressChangedEventArgs（worker_ProgressChanged(object sender, ProgressChangedEventArgs e)）中得到
. CancelAsync() ：取消（暂停）执行。

事件

worker.DoWork += new DoWorkEventHandler(worker_DoWork);//执行任务
worker.RunWorkerCompleted += new

```
RunWorkerCompletedEventHandler(worker_RunWorkerCompleted);//任务结束时
    worker.ProgressChanged += new
ProgressChangedEventHandler(worker_ProgressChanged)//报告状态
```

按照上边的资料，我们这个应用就可以这样处理之

formDisplay 是用于显示实时状态的窗口。有 DisplyMessage 方法来显示信息到界面

在 Hanlder 类（处理文件 copy 的）里面：

```
        static void worker_ProgressChanged(object sender, ProgressChangedEventArgs e)
        {
            //show the message on windows
           formDisplay.DisplyMessage("copy", e.UserState.ToString());//show message.
        }

        static void worker_RunWorkerCompleted(object sender, RunWorkerCompletedEventArgs
e)
        {
            string msg = "JOB copy : have been completed";
            formDisplay.DisplyMessage(msg);//show message
        }

        static void worker_DoWork(object sender, DoWorkEventArgs e)

        {
            for(…)
            {
                //copying
               (sender as BackgroundWorker). ReportProgress(0,"xxx complete");//report
}
        }
```

这样构造的程序里面，才不会出现 UI 失去响应。

当然，通过写自己的异步处理也可以实现，功能更强大。只不过这个用起来更简单。
至于 backgroundworker 是怎么实现的呢？这里有人已经做出了一些解答：

http://blog.joycode.com/sunmast/archive/2006/03/02/about_system_componentmodel_asynccoperation.aspx

在 UI 上使用 BackgroundWorker

凡是 WinForm 的应用程序，如果他执行了一个的非常冗长的处理操作（比如文件查询），它在执行时会锁定用户界面，虽然主活动窗口 一直在运行，但用户无法与程序交互，无法移动窗体或改变窗体大小，所以用户感觉很不爽。如何做才能使得这个程序有响应。答案就是在后台线程中执行这个操作。

在这里已经有了多种方法来做这个事情：

(一)委托异步调用

将具体耗时的操作作为一个委托，并用 BeginInvoke 来异步执行这个委托（Invoke 是同步调用），并且可以为这个操作传入参数并且通过 EndInvoke 方法获得返回返回值。

(二)使用 ThreadPool

新建.net FrameWork 中自带的 WaitCallback 委托，然后放到线程池中运行 ThreadPool.QueueUserWorkItem(callback); 根据 WaitCallback 委托的定义，可以传入一个 object 类型的参数。

但是不能精确的控制线程池中的线程。

(三)使用 Thread

和 ThreadPool 相比，使用 Thread 的开销会比较大。但是它有它的优势，使用 Thread 类可以显式管理线程。只要有可能，就应该使用 ThreadPool 类来创建线程。然而，在一些情况下，您还是需要创建并管理您自己的线程，而不是使用 ThreadPool 类。在.net 2.0 中，提供了一个新的委托 ParameterizedThreadStart 支持启动一个线程并传入参数，这是对原来的 ThreadStart 委托的改进。

说了这么多还没有说到今天的主角 BackgroundWorker，他也是一个在 2.0 中新增的类,可以用于启动后台线程,并在后台计算结束后调用主线程的方法.可以看出同样的功能使用委托的异步调用也可以实现，只是使用 BackgroundWorker 的话会更加的简便快捷，可以节省开发时间，并把你从创建自己的委托以及对它们的调用中解救出来。真是这样的吗看看下面这个例子。其实我也是从 101Samples 中看到的例子。

先看看 BackgroundWorker 中的主要概念。

第一：主要的事件及参数。

DoWork——当执行 BackgroundWorker.RunWorkerAsync 方法时会触发该事件，并且传递 DoWorkEventArgs 参数;

ProgressChanged——操作处理中获得的处理状态变化，通过 BackgroundWorker.ReportProgress(int)方法触发该事件，并且传递 ProgressChangedEventArgs,其中包含了处理的百分比;

RunWorkerCompleted——异步操作完成后会触发该事件，当然如果需要在操作过程中结束可以执行 BackgroundWorker.CancelAsync 方法要求异步调用中止，并且在异步委托操作中检测 BackgroundWorker.CancellationPending 属性如果为 true 的话，跳出异步调用，同时将 DoWorkEventArgs.Cancel 属性设为 true，这样当退出异步调用的时候，可以让处理 RunWorkerCompleted 事件的函数知道是正常退出还是中途退出。

第二：主要的方法。

BackgroundWorker.RunWorkerAsync——

“起动”异步调用的方法有两次重载 RunWorkerAsync(),RunWorkerAsync(object argument)，第二个重载提供了一个参数，可以供异步调用使用。(如果有多个参数要传递怎么办，使用一个类来传递他们吧)。调用该方法后会触发 DoWork 事件，并且为处理 DoWork 事件的函数 DoWorkEventArg 事件参数，其中包含

了 RunWorkerAsync 传递的参数。在相应 DoWork 的处理函数中就可以做具体的复杂操作。

BackgroundWorker.ReportProgress——

有时候需要在一个冗长的操作中向用户不断反馈进度，这样的话就可以调用的 ReportProgress(int percent),在调用 ReportProgress 方法时，触发 ProgressChanged 事件。提供一个在 0 到 100 之间的整数，它表示后台活动已完成的百分比。你也可能提供任何对象作为第二个参数，允许你 给事件处理程序传递状态信息。作为传递到此过程的 ProgressChangedEventArgs 参数属性，百分比和你自己的对象（如果提供的话）均要被传递到 ProgressChanged 事件处理程序。这些属性被分别命名为 ProgressPercentage 和 UserState，并且你的事件处理程序可以以任何需要的方式使用它们。(注意：只有在 BackgroundWorker.WorkerReportsProgress 属性被设置为 true 该方法才可用)。

BackgroundWorker.CancelAsync——

但需要退出异步调用的时候，就调用的这个方法。但是样还不够，因为它仅仅是将 BackgroudWorker.CancellationPending 属性设置为 true。你需要在具体的异步调用处理的时候，不断检查 BackgroudWorker.CancellationPending 是否为 true，如果是真的话就退出。(注意：只有在 BackgroundWorker.WorkerSupportsCancellation 属性被设置为 true 该方法才可用)。

贴出一段 101Samples 里面的代码，看一下就明白了：

```
public partial class MainForm : Form
{
private System.ComponentModel.BackgroundWorker backgroundCalculator;
public MainForm()
{
InitializeComponent();
backgroundCalculator = new BackgroundWorker();
backgroundCalculator.WorkerReportsProgress = true;
backgroundCalculator.WorkerSupportsCancellation = true;
backgroundCalculator.DoWork += new
DoWorkEventHandler(backgroundCalculator_DoWork);
backgroundCalculator.ProgressChanged += new
ProgressChangedEventHandler(backgroundCalculator_ProgressChanged);
backgroundCalculator.RunWorkerCompleted += new
RunWorkerCompletedEventHandler(backgroundCalculator_RunWorkerCompleted);
updateStatus(String.Empty);
}
private int getNextPrimeAsync(int start, BackgroundWorker worker, DoWorkEventArgs e)
{
int percentComplete = 0;
start++;
while (!isPrime(start))
{
// Check for cancellation
        if (worker.CancellationPending)
{
```

```
e.Cancel = true;
break;
}
else
{
start++;
percentComplete++;
worker.ReportProgress(percentComplete % 100);
}
}
return start;
}
void backgroundCalculator_RunWorkerCompleted(object sender, RunWorkerCompletedEventArgs
e)
{
if (e.Cancelled)
{
updateStatus("Cancelled.");
}
else if (e.Error != null)
{
reportError(e.Error);
}
else
{
reportPrime((int)e.Result);
}
calcProgress.Value = 0;
}
void backgroundCalculator_ProgressChanged(object sender, ProgressChangedEventArgs e)
{
updateProgress(e.ProgressPercentage);
}
void backgroundCalculator_DoWork(object sender, DoWorkEventArgs e)
{
int start = (int) e.Argument;
e.Result = getNextPrimeAsync(start, (BackgroundWorker)sender, e);
}
private void nextPrimeAsyncButton_Click(object sender, EventArgs e)
{
updateStatus("Calculating...");
int start;
Int32.TryParse(textBoxPrime.Text, out start);
if (start == 0)
```

```
{
reportError("The number must be a valid integer");
}
else
{
// Kick off the background worker process
                    backgroundCalculator.RunWorkerAsync(int.Parse(textBoxPrime.Text));
}
}
private void cancelButton_Click(object sender, EventArgs e)
{
if (backgroundCalculator.IsBusy)
{
updateStatus("Cancelling...");
backgroundCalculator.CancelAsync();
}
}
// Update the Status label
          private void updateStatus(string status)
{
calcStatus.Text = status;
}
// Indicate progress using progress bar
          private void updateProgress(int percentComplete)
{
calcProgress.Value = percentComplete;
}
}
```

BackgroundWorker 创建自己的委托并调用这个窗体的 Invoke 方法来运行它，BackgroundWorker 组件以一种优雅的方式来处理这个线程转换。BackgroundWorker 组件允许你从后台线程中调用它的 ReportProgress 方法，该方法触发其 ProgressChanged 事件处理例程返回到窗体的线程中。你不必使用 delegate/Invoke 方法自己处理这个线程转换，而是调用 ReportProgress，其余的事情交给组件来做。

DotNet 中异步编程的简单应用 - shenba - 博客园

DotNet 中异步编程的简单应用

这里说的异步编程并不是 AJAX 等的 Web 异步编程,而仅仅是 DotNet 中多线程的异步编程.这种多线程的异步编程主要用来解决某些受计算操作影响而引起主线程阻塞的问题.让程序(主要是窗体应用程序)看跑得更流畅.在 dotnet 的 CLR 以及 API 方法中有简单易用的方法供我们实现异步编程,并且都有相似的调用方法,诸如 BeginXXX,EndXXX,IAsyncResult 对象,同时也都涉及到回调,委托等操作.下面是一些简单的应用

1.异步的 IO 操作,基本上就是按参数传递

异步 IO

```
1// 值得注意的是最好给定 FileOptions.Asynchronous,相对效率会高些
2            FileStream fs = new FileStream("E:\\test.txt", FileMode.Open, FileAccess.Read, FileShare.Read, 1024, FileOptions.Asynchronous);
3            byte[] data = new byte[(int)fs.Length];
4
5            // 开始读取
6            fs.BeginRead(data, 0, data.Length, delegate (IAsyncResult ar)
7            {
8                // 这里是读取结束之后回调的委托方法内容
9                fs.EndRead(ar);
10               fs.Close();
11
12               string content = Encoding.Default.GetString(data);
13               Console.WriteLine(content);
14           }, fs);
```

2.读取数据库内容的异步操作,这里的代码是用在窗体程序中,涉及到一个跨线程改变窗体控件的问题,但是在窗体里,只有通过窗体的主线程来改变控件的行为.如下异步读取数据,读取数据的操作是在另外一个线程中,因此试图在这个线程中操作窗体控件是徒劳的,还会出异常.所以只能通过窗体的 invoke 方法,调用一个委托,这样才能实现.当然这里只是为了提出这个问题,其他的解决的方法还是有的,比如效率更低的轮询方法.

数据访问

```
1// 这个委托 用于在窗体的主线程中调用
2        private delegate void FillGridHandler(SqlDataReader reader);
3        private void AsnDataAccess()
4        {
5            // 对于异步操作的数据访问,连接字符串里必须设置 Async=True
6            string conStr = ConfigurationManager.ConnectionStrings["NorthWind"].ConnectionString;
7            SqlConnection con = new SqlConnection(conStr);
8            SqlCommand command = new SqlCommand("select [CompanyName] from [Suppliers]", con);
9
10           con.Open();
```

```
            command.BeginExecuteReader(delegate(IAsyncResult ar)
            {
                SqlDataReader reader = command.EndExecuteReader(ar);

                // 对于窗体应用程序的 GridView 的绑定
                FillGridHandler fillGridHandler = delegate(SqlDataReader
reader1)
                {
                    DataTable dt = new DataTable();
                    dt.Load(reader1);
                    reader.Close();
                    dataGridView1.DataSource = dt;
                };

                // 用窗体自身的线程去触发委托方法 才能改变窗体里控件属性
                this.Invoke(fillGridHandler, reader);
            }, command, System.Data.CommandBehavior.CloseConnection);
        }
```

3.异步触发委托的方法,异步编程离不开委托,其本身也就是调用了委托的异步方法,其内部就必定有一个委托对象

委托异步

```
delegate int CalSumHandler(int number);
        private static void DelegateAsyn()
        {
            CalSumHandler handler = delegate(int number)
            {
                int sum = 0;
                for (int i = 0; i < number; i++)
                {
                    sum += i;
                }
                return sum;
            };

            int n = 10;
            handler.BeginInvoke(n, delegate(IAsyncResult ar)
            {
                int res = handler.EndInvoke(ar);
                Console.WriteLine("result from asyndelegate,sum = {0}", res);
            }, n);
        }
```

posted on 2007-10-01 11:12 神八 阅读(102) 评论(0) 编辑 收藏

Delegate 比较全面的例子(原创) - 享受代码,享受人生 - 博客园

享受代码,享受人生

SOA is an integration solution. SOA is message oriented first.

The Key character of SOA is loosely coupled. SOA is enriched by

creating composite apps.

将 Delegate 理解为接口，只有一个方法的接口，这样最容易理解。这个方法只有声明，没有实现，实现在别的类。（实际上应该把它看作函数指针，不过接口更容易理解些。）

在你的类中有一个 Delegate 就相当于有一个接口。通过这个接口你可以调用一个方法，而这个方法在别的类定义，由别的类来干。

为了说的形象一点，举个例子：

学生考试完后成绩出来了，考的好了老师要表扬，考的不好了老师要批评。

使用接口的方法：

```
using System;
public class Student
{
    private IAdviser adviser;

    public void SetAdviser(IAdviser iadviser)
    {
        adviser = iadviser;
    }

    private int score;

    public void SetScore(int value)
    {
        if (value > 100 || value < 0)
        {
            Console.Out.WriteLine("分数不对");
        }
        else
        {
            score = value;
            if (adviser != null)
            {
                string result = adviser.Advise(score);
                Console.Out.WriteLine("学生收到老师返回的结果\t"+result);
            }
        }
    }
```

```
 }

public interface IAdviser
{
    string Advise(int score);
}

public class Teacher : IAdviser
{
    public string Advise(int score)
    {
        if (score < 60)
        {
            Console.Out.WriteLine(score+"老师说加油");
            return "不及格";
        }
        else
        {
            Console.Out.WriteLine(score+"老师说不错");
            return "及格";
        }
    }
}

class MainClass
{
    [STAThread]
    private static void Main(string[] args)
    {
        IAdviser teacher = new Teacher();
        Student s = new Student();
        s.SetAdviser(teacher);

        Console.Out.WriteLine("学生得到 50 分");
        s.SetScore(50);

        Console.Out.WriteLine("\n 学生得到 75 分");
        s.SetScore(75);

        Console.ReadLine();
    }
}
```

使用 Delegate 的方法：

```
using System;
using System.Threading;

public class Student
{

    private int score;

    public void SetScore(int value)
    {
        if (value > 100 || value < 0)
        {
            Console.Out.WriteLine("分数不对");
        }
        else
        {
            score = value;
            if (AdviseDelegateInstance!= null)
            {
                string result=AdviseDelegateInstance(score);
                Console.Out.WriteLine("学生收到老师返回的结果\t"+result);
            }
        }
    }

    public   delegate string AdviseDelegate(int score);

    public AdviseDelegate AdviseDelegateInstance;
}

public class Teacher
{
    public string Advise(int score)
    {
        if(score<60)
        {
            Console.Out.WriteLine(score+"老师说加油");
            return "不及格";
        }
        else
        {
            Console.Out.WriteLine(score+"老师说不错");
            return "及格";
        }
```

```
    }
}

class MainClass
{
    [STAThread]
    static void Main(string[] args)
    {
        Teacher teacher=new Teacher();
        Student s=new Student();

        s.AdviseDelegateInstance=new
Student.AdviseDelegate(teacher.Advise);

        Console.Out.WriteLine("学生得到 50 分");
        s.SetScore(50);

        Console.Out.WriteLine("\n 学生得到 75 分");
        s.SetScore(75);

        Console.ReadLine();
    }
}
```

如果老师很忙不能及时回复怎么办？比如这样：

```
public class Teacher
{
    public string Advise(int score)
    {
        Thread.Sleep(3000);
        if(score<60)
        {
            Console.Out.WriteLine(score+"老师说加油");
            return "不及格";
        }
        else
        {
            Console.Out.WriteLine(score+"老师说不错");
            return "及格";
        }
    }
}
```

总不能让学生一直等下去吧，采用多线程并发的办法。

Interface 的解决办法：

```
public void SetScore(int value)
{
    if (value > 100 || value < 0)
    {
        Console.Out.WriteLine("分数不对");
    }
    else
    {
        score = value;
        if (adviser != null)
        {
            Thread.adviserThread=new Thread(new
ThreadStart(adviser.Advise()));
            adviserThread.Start();
        }
    }
}
```

但是它不能使用带参数的函数，怎么办？（谁知道方法请指教）

.Net2.0 提供了新的方法 ParameterizedThreadStart

用 Delegate 解决（异步调用）：

```
public void SetScore(int value)
{
    if (value > 100 || value < 0)
    {
        Console.Out.WriteLine("分数不对");
    }
    else
    {
        score = value;
        if (AdviseDelegateInstance!= null)
        {
            AdviseDelegateInstance.BeginInvoke(score,null,null);

        }
    }
}
```

不过这样我们失去了老师的返回结果，不知道有没有及格了。

采用轮讯的方法去获得结果：

```
public void SetScore(int value)
{
    if (value > 100 || value < 0)
```

```
        {
            Console.Out.WriteLine("分数不对");
        }
        else
        {
            score = value;
            if (AdviseDelegateInstance!= null)
            {

                IAsyncResult res =
AdviseDelegateInstance.BeginInvoke(score,null, null);

                while( !res.IsCompleted )
System.Threading.Thread.Sleep(1);

                string result =
AdviseDelegateInstance.EndInvoke(res);
                Console.Out.WriteLine("学生收到老师返回的结果\t"+result);

            }
        }
    }
```

不过这样主线程又被阻塞了，采用回调的方式：（注：接口也可以采用回调的方式获得返回值）

```
    public void SetScore(int value)
    {
        if (value > 100 || value < 0)
        {
            Console.Out.WriteLine("分数不对");
        }
        else
        {
            score = value;
            if (AdviseDelegateInstance!= null)
            {
                IAsyncResult res =
AdviseDelegateInstance.BeginInvoke(score, new
System.AsyncCallback(CallBackMethod), null);
            }
        }
    }

    private void CallBackMethod(IAsyncResult asyncResult)
    {
```

```
        string result = AdviseDelegateInstance.EndInvoke(asyncResult);

        Console.Out.WriteLine("学生收到老师返回的结果\t" + result);
    }
```

这样就比较得到了一个比较好的解决方案了。我们再来看看 BeginInvoke 的第四个参数是干吗的呢？

```
    public void SetScore(int value)
    {
        if (value > 100 || value < 0)
        {
            Console.Out.WriteLine("分数不对");
        }
        else
        {
            score = value;
            if (AdviseDelegateInstance!= null)
            {
                AdviseDelegateInstance.BeginInvoke(score, new
System.AsyncCallback(CallBackMethod), "idior");
            }
        }
    }

    private void CallBackMethod(IAsyncResult asyncResult)
    {
        string result = AdviseDelegateInstance.EndInvoke(asyncResult);
        string stateObj=(string)asyncResult.AsyncState;

        Console.Out.WriteLine("学生{0}收到老师返回的结果\t" +
result,stateObj.ToString());
    }
```

哦，原来它可以用来标记调用者的一些信息。（这里采取的是硬编码的方式，你可以把它改为学生的 id 之类的信息）。

总结：Delegate 类似与 Interface 但是功能更加强大和灵活，它甚至还可以绑定到 Static 方法只要函数签名一致，而且由于+=操作符的功能，实现多播也是极为方便（即 Observer 模式），在此不再举例。

（补充：多播的时候改一下 SetScore 函数）

```
    public void SetScore(int value)
    {
        if (value > 100 || value < 0)
        {
```

```
                Console.Out.WriteLine("分数不对");
            }
            else
            {
                score = value;

                if (AdviseDelegateInstance!= null)
                {
                    foreach( AdviseDelegate ad in
AdviseDelegateInstance.GetInvocationList())
                    {
                        ad.BeginInvoke(score, new
System.AsyncCallback(CallBackMethod), "idior");
                    }
                }
            }
        }
```

本文没什么新的内容，就是自己练一下手，写个总结材料，希望对大家有帮助。.net2.0 提供了更好的线程模型。

完整源代码如下：

```
using System;
using System.Threading;

public class Student
{
    private int score;

    public void SetScore(int value)
    {
        if (value > 100 || value < 0)
        {
            Console.Out.WriteLine("分数不对");
        }
        else
        {
            score = value;
            if (AdviseDelegateInstance!= null)
            {
                AdviseDelegateInstance.BeginInvoke(score, new
System.AsyncCallback(CallBackMethod), "idior");
            }
```

```
        }
    }

    private void CallBackMethod(IAsyncResult asyncResult)
    {
        string result = AdviseDelegateInstance.EndInvoke(asyncResult);
        string stateObj=(string)asyncResult.AsyncState;

        Console.Out.WriteLine("学生{0}收到老师返回的结果\t" + result,stateObj);
    }

    public delegate string AdviseDelegate(int score);
    public AdviseDelegate AdviseDelegateInstance;
}

public class Teacher
{
    public string Advise(int score)
    {
        Thread.Sleep(3000);
        if (score < 60)
        {
            Console.Out.WriteLine(score + "老师说加油");
            return "不及格";
        }
        else
        {
            Console.Out.WriteLine(score + "老师说不错");
            return "及格";
        }
    }
}

class MainClass
{
    [STAThread]
    private static void Main(string[] args)
    {
        Teacher teacher = new Teacher();
        Student s = new Student();
        s.AdviseDelegateInstance= new
Student.AdviseDelegate(teacher.Advise);
        Console.Out.WriteLine("学生得到 50 分");
        s.SetScore(50);
```

```
            Console.Out.WriteLine("\n 学生得到 75 分");
            s.SetScore(75);
            Console.ReadLine();
        }
    }
```

参考资料： .NET Delegates: A C# Bedtime Story

Feedback

re: Delegate 比较全面的例子(原创) 回复 更多评论

2005-02-02 10:57 by KingofSC

不错啊，居然还说到多线程去了

re: Delegate 比较全面的例子(原创) 回复 更多评论

2005-02-02 11:27 by idior

@KingofSC

发现你总是潜水哦 :P

哪天看看你的大作啊?

re: Delegate 比较全面的例子(原创) 回复 更多评论

2005-02-02 14:05 by 吕震宇

不错！很全面。就是“Advise”出现得太多了，有时候分不清是 Advise 方法还是 Advise 委派了：）

re: Delegate 比较全面的例子(原创) 回复 更多评论

2005-02-02 15:22 by idior

@吕震宇

不好意思,是有点让人看不懂,已修改.谢谢指正.

re: Delegate 比较全面的例子(原创) 回复 更多评论

2005-03-11 00:56 by douhao_lale

很好啊，谢谢

re: Delegate 比较全面的例子(原创) 回复 更多评论

2005-07-25 15:46 by Boler

开始看的挺明白，后来太复杂了，看不懂了，放弃了~

re: Delegate 比较全面的例子(原创) 回复 更多评论

2005-07-25 20:46 by idior

需要用到多线程的时候再来看看吧

re: Delegate 比较全面的例子(原创) 回复 更多评论

2005-07-26 08:17 by yinh

不错，idior 你写的文章我都非常感兴趣。

re: Delegate 比较全面的例子(原创) 回复 更多评论

2005-07-30 17:06 by HNHLS99

上面的多线程的例子比较透彻，但是对多线程的机制涉及的很少啊，希望能补充以下。比如线程的轮询，线程的开始与结束，异步的调用的同步等等。

re: Delegate 比较全面的例子(原创) 回复 更多评论

2005-07-30 17:49 by idior

呵呵 别忘了本文的题目 “Delegate 比较全面的例子”

或许可以由你来介绍多线程啊。

re: Delegate 比较全面的例子(原创) 回复 更多评论

2005-08-18 11:18 by 阿新
牛，呕像
re: Delegate 比较全面的例子(原创)　回复　更多评论
2005-11-22 10:45 by luyu
前面看的不错，越看越混乱了。
re: Delegate 比较全面的例子(原创)　回复　更多评论
2005-12-14 17:05 by giogio
呵呵，其实写到“如果老师很忙不能及时回复怎么办？比如这样：”之前比较好。
后面就和多线程结合的更紧密了。

我觉得线程本身比 delegate 要更大更基本，delegate 可以算一节，线程就应该算一章。

类似于讲蒸气机的时候讲到一半开始用量子力学解释……这个确实容易让人晕……
re: Delegate 比较全面的例子(原创)　回复　更多评论
2005-12-19 09:41 by giogio
“将 Delegate 理解为接口，只有一个方法的接口，这样最容易理解。”

这么说不好吧……
我觉得这俩玩意不是一回事啊，只是看起来比较想像而已。
re: Delegate 比较全面的例子(原创)　回复　更多评论
2005-12-19 22:16 by idior
@ giogio

记住这句话，或许有一天你会突然觉得有道理的。
这是 delegate 的本质，不过需要你对面向对象有一定的理解。
你可以参考一下这篇文章。

http://idior.cnblogs.com/archive/2005/02/03/101510.aspx

http://linkcd.cnblogs.com/archive/2005/07/19/196087.html
re: Delegate 比较全面的例子(原创)　回复　更多评论
2005-12-19 22:46 by giogio
其实是这样的。
老师讲完 delegate 后，我就说：这玩意和 interface 起的作用一样。

老师就说：从表面上看是这样的，但是两者从根本上不同。
他似乎很像让我牢牢记住这是两种东西，强调这两个不能混淆。

老师还让我准备一下，用大约 15 分钟给大家讲 delegate 和 event 呢……
re: Delegate 比较全面的例子(原创)　回复　更多评论
2006-03-18 17:03 by qq:86562467
拜托各位高手，谁哪儿有关于 Delegate Event WebService 方面的例子
我想实现如下的功能

假设我有三个类 分别是 Class S , Class A , Class B

现在想用 代理 Delegate 和 事件 Event 实现类 A 和类 B 的通信功能

但是 类 A 和类 B 不能直接通信 必须通过 类 S 实现

类 S 就是 起一个 中转或服务器的作用

谁哪儿有这样的例子 拜托 给一份 学习一下

谢谢了

re: Delegate 比较全面的例子(原创) 回复 更多评论

2006-04-26 10:48 by anchky

收藏了！

re: Delegate 比较全面的例子(原创) 回复 更多评论

2006-08-10 09:28 by 匿名

貌似老大很习惯用 JAVA，相信 public void SetScore(int value)

不能通过编译，似乎 s.AdviseDelegateInstance=new

Student.AdviseDelegate(teacher.Advise)也是不能通过编译的，因为你没有实现 Advise 的 get 方法。但老大确实很有想法，

把一些问题说得比较清楚

re: Delegate 比较全面的例子(原创) 回复 更多评论

2006-09-13 16:20 by lxinxuan

Console.Out.WriteLine("学生{0}收到老师返回的结果\t" + result,stateObj); 改为

Console.Out.WriteLine(string.Format("学生{0}收到老师返回的结果\t" + result,

stateObj));

re: Delegate 比较全面的例子(原创) 回复 更多评论

2006-09-13 17:11 by lxinxuan

@qq:86562467:

我知道以下是没有满足你的要求，所以，我请求 idior 帮忙解答：

```
public class A
{
public delegate void SetValueDelegate(string v);
public SetValueDelegate SetValueInstance;

public void SetValue(string v)
{
SetValueInstance(v);
}
}

public class S
{
B b = new B();
public void SetValue(string v)
{
b.SetValue(v);
}
```

```
}

public class B
{
public string bValue;
public B()
{
bValue = "b";
}

public void SetValue(string v)
{
this.bValue = v;
}

public string GetValue()
{
return this.bValue;
}
}

class MainClass
{
[STAThread]
private static void Main(string[] args)
{
S s = new S();
A a = new A();
B b = new B();
a.SetValueInstance = new A.SetValueDelegate(s.SetValue);
a.SetValue("a");
MessageBox.Show(b.GetValue());//
}
}
```

re: Delegate 比较全面的例子(原创)　回复　更多评论

2006-09-19 00:53 by huangyi_

<pre>

```
class Delegate(object):
'模拟.net 的 delegate'
def __init__(self):
self.handlers = []
def __call__(self,*args,**kw):
for h in self.handlers:
```

```
h(*args,**kw)
def __iadd__(self,handler):
self.handlers.append(handler)
return self

d = Delegate()
def handler1(a,b):print a,b
def handler2(a,b):print a+b
d(1,2)
</pre>
```

也许这个可以帮助理解？ 呵呵

re: Delegate 比较全面的例子(原创)　回复　更多评论

2006-10-12 16:59 by wthorse

但是它不能使用带参数的函数，怎么办？

可以在调用的 target 所在的类中定义属性，调用多线程之前先赋值。

re: Delegate 比较全面的例子(原创)　回复　更多评论

2007-02-11 16:03 by 臭石头

多线程传递参数，我经常用，呵呵。

写一个类来包装，写一个没有参数的方法，方法内部调用带参数的方法，这些参数，作为这个类的公共成员。

声明这个类的一个实例，然后……剩下的不说了

对 Delegate 的理解[TrackBack]　回复　更多评论

2007-03-20 18:06 by 9q

下面这篇文章论述的比较有意思：Delegate 比较全面的例子(原创)　查看原文

System.ComponentModel.AsyncOperation 类 - 这个类很特别昨天在尝试使用 System.ComponentModel.BackgroundWorker 时，发现这个类的行为和我预料的大不一样，可以说是惊喜。原来以为这个类只是一个线程的简单包装，用多线程模拟了异步调用而已；但是看下面的这段代码：

```
Thread.CurrentThread.Name = "Main Thread";
backgroundWorker1.RunWorkerAsync();

...
```

```
private void backgroundWorker1_DoWork(object sender, DoWorkEventArgs e)
{
    int i = 0;
    while (i++ < 100)
    {
        backgroundWorker1.ReportProgress(i);
        Thread.Sleep(50);
    }
}

private void backgroundWorker1_ProgressChanged(object sender, ProgressChangedEventArgs e)
{
    this.Text = e.ProgressPercentage + "% - " + Thread.CurrentThread.Name;
}

private void backgroundWorker1_RunWorkerCompleted(object sender,
RunWorkerCompletedEventArgs e)
{
    this.Text = "DONE - " + Thread.CurrentThread.Name;
}
```

毫无疑问，_DoWork 方法是运行在另一个不同线程之上的（很容易验证这一点，这也符合 BackgroundWorker 的设计），而这个方法又调用了 backgroundWorker1.ReportProgress 方法，触发了 ProgressChanged 事件。在通常的异步实现，_ProgressChanged 方法应该运行于事件触发者相同的线程中；但在这里，它运行于主线程（名为 Main Thread 的线程）。_RunWorkerCompleted 方法也是一样。

在我看来，这个行为非常特别，实际上它也非常有用。这样_ProgressChanged 方法体中操作 UI 控件的代码都无需使用 Control.Invoke 包装了，让程序的编写大为简化。而我真正感兴趣的是这个类究竟是怎么实现的，我用 Reflector 打开它的源码之后，原来关键在于它用到了一个名为 AsyncOperation 的类（System.ComponentModel.AsyncOperation）。

AsyncOperation 类有个 Post 方法，可以用来把一个委托（作为方法指针/列表）提交给另一个线程执行。继续反编译下去，又查到了 System.Threading.SynchronizationContext 类。不过具体怎么实现是无从得知了，因为追踪到最后，停在了一个[MethodImpl(MethodImplOptions.InternalCall)]的方法，它由 CLR 本身实现。（我个人猜测，其中很可能利用了 Windows API：Get/SetThreadContext，和结构体 CONTEXT，改掉了线程上下文。）

退一步说，它怎么实现的并不是那么重要，重要的是我们可以用这个 AsyncOperation 类实现自己的 BackgroundWorker。这里是我写的和上面代码基本等价的实现：

```
AsyncOperation asyncOperation;
SendOrPostCallback progressReporter;
Thread workerThread;
```

```
public MainForm()
{
    InitializeComponent();

    asyncOperation = AsyncOperationManager.CreateOperation(null);
    progressReporter = new SendOrPostCallback(ReportProgress);
    workerThread = new Thread(new ThreadStart(WorkOnWorkerThread));
}

private void MainForm_Load(object sender, EventArgs e)
{
    Thread.CurrentThread.Name = "Main Thread";
    workerThread.Name = "Worker Thread";
    workerThread.IsBackground = true;
    workerThread.Start();
}

void ReportProgress(object obj)
{
    this.Text = obj.ToString() + "% - " + Thread.CurrentThread.Name;
}

void WorkOnWorkerThread()
{
    int i = 0;
    while (i++ < 100)
    {
        asyncOperation.Post(progressReporter, i);
        Thread.Sleep(50);
    }
}
```

C#事件编程 - 五年 - 博客园　　wxy

1.定义一个 "代表"

如:

```
public delegate void DisconnectedEventHandler(object
sender,ClientEventArgs e);
```

2.定义事件参数

如:

```
public class ClientEventArgs : EventArgs
{
```

```
        public IPAddress IP
        {
            get { return ( (IPEndPoint)this.socket.RemoteEndPoint
).Address; }
        }
        public int Port
        {
            get{return ((IPEndPoint)this.socket.RemoteEndPoint).Port;}
        }
        public ClientEventArgs(Socket clientManagerSocket)
        {
            this.socket = clientManagerSocket;
        }
    }
```

3.使用"代表"定义一个事件

```
        public event DisconnectedEventHandler Disconnected;
```

4.触发事件

```
        protected virtual void OnDisconnected(ClientEventArgs e)
        {
            if ( Disconnected != null )
                Disconnected(this , e);
        }

         this.OnDisconnected(new ClientEventArgs(this.socket));
```

5.使用事件

```
            ClientManager newClientManager = new ClientManager(socket);
            newClientManager.Disconnected += new
DisconnectedEventHandler(ClientDisconnected);
```

6.定义事件处理方法

```
        void ClientDisconnected(object sender , ClientEventArgs e)
        {
            if ( this.RemoveClientManager(e.IP) )
                this.UpdateConsole("Disconnected." , e.IP , e.Port);
        }
```

关于.NET 异步调用的初步总结
最近看了看.NET 异步调用方面的资料，现择重点总结，若有纰漏敬请指正。

异步调用的实质：
异步调用通过委托将所需调用的方法置于一个新线程上运行，从而能够使一个可能需要较长时间的任务在后台执行而不影响调用方的其他行为。

异步调用的实现：
前面已经讲道，异步调用通过委托实现。委托支持同步和异步调用。在同步调用中，一个委托的实例可记录多个目标方法；在异步调用中，一个委托实例中有且只能包含一个目标方法。异步调用使用委托实例的 BeginInvoke 方法和 EndInvoke 方法分别开始调用和检索返回值，这两个方法在编译期生成。调用 BeginInvoke 后委托立即返回；调用 EndInvoke 时倘若委托方法未执行完毕，则阻塞当前线程至调用完毕。
假设有一个委托

```
public delegate int ASyncHandler(int a,string b,ref string c);
```

那么，其 BeginInvoke 与 EndInvoke 的形式如下：

```
public IAsyncResult BeginInvoke(int a,string b,ref string c,AsyncCallback callback,object asyncState);

public int EndInvoke(ref string c,IAsyncResult asyncResult);
```

也就是说，BeginInvoke 与 EndInvoke 的参数列表与当前委托签名有关，可以总结为：

```
public IAsyncResult BeginInvoke(委托所具有的全部参数，AsyncCallback callback,object asyncState);

public 委托返回值 EndInvoke(委托参数中 ref/out 部分,IAsyncResult asyncResult);
```

BeginInvoke 返回一个 IAsyncResult，其实质是实现 IAsyncResult 的 System.Runtime.Remoting.Messaging.AsyncResult 类。该对象相当于一个“凭证”，在调用 EndInvoke 时用于确认应等待返回的方法（猜测如此）。就像去银行，存钱时拿到一份存折（凭证），取款时依据存折（凭证）取款。
EndInvoke 检索委托返回值，并返回标有 ref/out 的参数值。
IAsyncResult 接口声明：

```
public interface IAsyncResult
{
    object AsyncState{get;}
    WaitHandle AsyncWaitHandle{get;}
    bool CompletedSynchronously{get;}
    bool IsCompleted{get;}
}
```

等待调用结束的三种方法：
1、使用 EndInvoke 主动等待异步调用结束。这是最简单的一种方法，适用于非用户界面程序及一些 IO 操作，

因为在调用 EndInvoke 之后当前线程被阻塞，除了等待什么都不能做。
2、使用 WaitHandle 等待异步调用结束。IAsyncResult 中有 WaitHandle 成员，获取用于等待异步操作完成的 WaitHandle，即调用结束信号。使用 WaitHandle.WaitOne()可以阻塞当前线程至异步调用完成。这样做的好处是：在调用 WaitOne 之后、EndInvoke 之前，可以执行其他处理。
3、主动轮询。使用 IAsyncResult 中有 IsCompleted 成员检索当前异步调用情况。该方法适用于用户界面程序，想象可在一个循环内做到既等待委托完成，又可以更新用户界面。
4、使用回调，在异步调用结束时执行一个操作。前面的 BeginInvoke 方法签名的最后两个参数用于回调。需要用到 AsyncCallback 委托：

```
public delegate void AsyncCallback(IAsyncResult asyncResult);
```

回调方法在系统线程池中执行。BeginInvoke 的最后一个参数（object asyncState）可以传递包含回调方法将要使用的信息的对象。在回调方法中调用 EndInvoke 可以通过取得 System.Runtime.Remoting.Messaging.AsyncResult.AsyncDelegate 实现。

个人认为方法 1、2 相差不算太大。

先写这么些，以后再补上其他的一些东西。

Mcad 学习笔记之异步编程(AsyncCallback 委托,IAsyncResult 接口,BeginInvoke 方法,EndInvoke 方法的使用小总结) - aierong 原创技术随笔(.Net 方向应用) - 博客园

Mcad 学习笔记之异步编程(AsyncCallback 委托,IAsyncResult 接口,BeginInvoke 方法,EndInvoke 方法的使用小总结)

Posted on 2005-05-25 17:47 aierong 阅读(4162) 评论(3) 编辑 收藏 引用 网摘 所属分类: MCAD 学习

让我们来看看同步异步的区别：

同步方法调用在程序继续执行之前需要等待同步方法执行完毕返回结果

异步方法则在被调用之后立即返回以便程序在被调用方法完成其任务的同时执行其它操作

.NET 框架基类库中有好几种类都可以提供同步和异步的方法调用。

因为同步方法调用会导致程序流程中途等待，所以采用同步方法的情况下往往会导致程序执行的延迟

相比来说，在某些条件下选择异步方法调用就可能更好一些

例如，有的时候程序需要给多个 Web 服务发出请求，还有远程处理信道（HTTP、TCP）和代理，这时就最好采用异步方法

.NET Framework 允许异步调用任何方法，定义与需要调用的方法具有相同签名的委托

CLR 将自动为该委托定义添加适当签名的 BeginInvoke 虚方法和 EndInvoke 虚方法和 Invoke 方法。

关于委托的这 3 个方法的详细说明可以参考这文章

http://www.cnblogs.com/aierong/archive/2005/05/25/162181.html

我们先来了解这 2 个方法和一个委托和一个接口：

(1)

BeginInvoke 方法用于启动异步调用

它与您需要异步执行的方法具有相同的参数，只不过还有两个额外的参数,将 AsyncCallback 和 AsyncState（可通过

IAsyncResult 接口的

AsyncState 属性获得）作为最后两个参数,如没有可以为空.

BeginInvoke 立即返回，不等待异步调用完成。

BeginInvoke 返回 IasyncResult，可用于监视调用进度。

结果对象 IAsyncResult 是从开始操作返回的，并且可用于获取有关异步开始操作是否已完成的状态。

结果对象被传递到结束操作，该操作返回调用的最终返回值。

在开始操作中可以提供可选的回调。如果提供回调，在调用结束后，将调用该回调；并且回调中的代码可以调用结束操作。

(2)

EndInvoke 方法用于检索异步调用结果。

在调用 BeginInvoke 后可随时调用 EndInvoke 方法,注意：始终在异步调用完成后调用 EndInvoke.

如果异步调用未完成，EndInvoke 将一直阻塞到异步调用完成。

EndInvoke 的参数包括需要异步执行的方法的 out 和 ref 参数以及由 BeginInvoke 返回的 IAsyncResult。

要注意的是,始终在异步调用完成后调用 EndInvoke

(3)

AsyncCallback 委托用于指定在开始操作完成后应被调用的方法

AsyncCallback 委托被作为开始操作上的第二个到最后一个参数传递

代码原型如下：

```
[Serializable]
public delegate void AsyncCallback(IAsyncResult ar);
```

(4)

IAsyncResult 接口

它表示异步操作的状态.

该接口定义了 4 个公用属性

实际上，发起和完成.NET 异步调用有 4 种方案可供你选择

1.方案 1-自己调用 EndInvoke 方法

异步执行方法的最简单方式是以 BeginInvoke 开始，对主线程执行一些操作，然后调用 EndInvoke,EndInvoke 直到异步调用完成后才返回

还是先来段自己喜欢的控制台代码：

```
1using System;
2
3namespace ConsoleApplication1
4{
5     class Class1
6     {
7          public delegate void AsyncEventHandler();
8
9          void Event1()
10         {
11             Console.WriteLine("Event1 Start");
12             System.Threading.Thread.Sleep(2000);
13             Console.WriteLine("Event1 End");
14         }
15
16         void Event2()
17         {
18             Console.WriteLine("Event2 Start");
19             int i=1;
20             while(i<1000)
21             {
22                 i=i+1;
23                 Console.WriteLine("Event2 "+i.ToString());
24             }
25             Console.WriteLine("Event2 End");
```

```
26          }
27
28          void CallbackMethod(IAsyncResult ar)
29          {
30              ((AsyncEventHandler) ar.AsyncState).EndInvoke(ar);
31          }

34          [STAThread]
35          static void Main(string[] args)
36          {
37              long start=0;
38              long end=0;
39              Class1 c = new Class1();
40              Console.WriteLine("ready");
41              start=DateTime.Now.Ticks;
42
43              AsyncEventHandler asy = new AsyncEventHandler(c.Event1);
44              IAsyncResult ia=asy.BeginInvoke(null,null);
45              c.Event2();
46              asy.EndInvoke(ia);
47
48              end =DateTime.Now.Ticks;
49              Console.WriteLine("时间刻度差="+ Convert.ToString(end-start) );
50              Console.ReadLine();
51          }
52      }
53}
54
```

此程序简单,异步的处理过程在代码 43-46 这几行

结果如下：

现在让我们来看看同步处理

修改代码 43-46 这几行代码：

c.Event1();

c.Event2();

结果如下：

前者的时间刻度大大小于后者

我们可以明显地看到异步运行的速度优越性

2.方案 2-采用查询（IsCompleted 属性）

IAsyncResult.IsCompleted 属性获取异步操作是否已完成的指示,发现异步调用何时完成.

再次修改代码 43-46 这几行代码：

AsyncEventHandler asy = new AsyncEventHandler(c.Event1);

```
IAsyncResult ia=asy.BeginInvoke(null,null);
c.Event2();
while(!ia.IsCompleted)
{
}
asy.EndInvoke(ia);
```

3.方案 3-采用 AsyncWaitHandle 来等待方法调用的完成

IAsyncResult.AsyncWaitHandle 属性获取用于等待异步操作完成的 WaitHandle

WaitHandle.WaitOne 方法阻塞当前线程，直到当前的 WaitHandle 收到信号

使用 WaitHandle，则在异步调用完成之后，但在通过调用 EndInvoke 结果之前，可以执行其他处理

再次修改代码 43-46 这几行代码：

```
AsyncEventHandler asy = new AsyncEventHandler(c.Event1);
IAsyncResult ia=asy.BeginInvoke(null,null);
c.Event2();
ia.AsyncWaitHandle.WaitOne();
```

4.方案 4-利用回调函数

如果启动异步调用的线程不需要处理调用结果，则可以在调用完成时执行回调方法

要使用回调方法，必须将代表该方法的 AsyncCallback 委托传递给 BeginInvoke

再次修改代码 43-46 这几行代码：

```
AsyncEventHandler asy = new AsyncEventHandler(c.Event1);
asy.BeginInvoke(new AsyncCallback(c.CallbackMethod),asy);
c.Event2();
```

希望上面提到的知识对你有所提示

当然欢迎交流和指正

blog:

http://www.cnblogs.com/aierong

author:aierong

email:aierong@126.com

Feedback

re:

Mcad 学习笔记之异步编程(AsyncCallback 委托,IAsyncResult 接口,BeginInvoke 方法,EndInvoke 方法的使用小总结)

回复　更多评论

2005-05-26 09:14 by eric

问个题外话，考 Mcsd for .net 怎么报名?准备呢？微软中文认证主页上，找不到相关的信息啊；能给些相关的资源吗？谢谢。

re:

Mcad 学习笔记之异步编程(AsyncCallback 委托,IAsyncResult 接口,BeginInvoke 方法,EndInvoke 方法的使用小总结)

回复　更多评论

2005-05-26 09:59 by aierong

to:eric

我也没考过

具体的我也不知道

http://www.examlink.com/

你去看看

使用委托进行异步编程[TrackBack] 回复 更多评论

2007-04-19 10:50 by 落叶

1.什么是异步编程使用.NET 异步编程，在程序继续执行的同时对.NET 类方法进行调用，直到进行指定的回调为止；如果没有提供回调，则直到对调用的阻塞、轮询或等待完成为止。例如，一个程序可以调用一...

查看原文

UI 设计注意点

软件的智能和记忆功能

1.用户登录界面最好有用户名和 ID 的记忆，焦点直接定位到密码输入框

2.单据录入界面最好有保存和载入默认值的功能

3.单据搜索界面可以保存用户自定义的各种搜索条件组合

4.用户调整过的 GRID 的列宽，窗口的位置可以自动记忆

5.系统可以根据用户的使用频度对相关功能进行自动的优先级排序

6.系统能够记忆不同用户的使用偏好，使用系统的固有模式和常用的自定义设置

减少不必要的重复交互

1.减少不必要的各种操作，能够点一次鼠标或敲一次键盘完成的绝不作出两次或多次。

2.提示信息要适度，太多不好，太少也不好。

3.数据项完整性校验问题要注意光标焦点自动定位到错误处

4.完整业务功能不要让用户在多个窗口切换多次才能够完成。尽量减少这种切换。

5.为了方便用户切换窗口，相关的表单最好都作为非模式的形式。

6.相同的信息不要让用户在系统中多处或多次录入，保证入口的唯一性

7.系统要尽可能根据用户已经录入信息自动获取其它附属信息，而不需要用户重复的选择或录入。

导航和界面跳转

1.表单新弹出对话框，对话框再弹出对话框的这种层次要控制在 3 层以内。

2.所有的非模式活动窗口最好有类似桌面任务栏一样的停靠方式，方便切换窗口

3.系统可以支持用户自己定义常用功能和菜单

4.对于常用功能应该提供便捷的快捷键和工具栏按钮

5.对于系统中提供的各种业务和表单功能能够让用户便捷挑转到帮助信息上

6.对表单和界面联动和交互的时候要注意相关界面数据的自动刷新

7.一个窗口中最多不要出现超过三个的 GRID 控件

8.BS 方式不要左右滚屏。CS 模式既要避免左右滚屏也要避免上下滚屏

9.需要根据业务查看需求和数据的展现需求来选择合适的界面控件

系统性能和健壮性方面的

1.系统中相关的耗时操作都必须必须转变鼠标为等待状态

2.系统耗时操作超过 30 秒的最好能够提供给用户相关的进度条功能

3.系统耗时功能超过 2 分钟的最好能够设计为异步多线程的方式进行处理

4.系统应用有友好的完整性和约束校验的提示信息，方便用户修改录入数据

5.在系统出现异常情况下应该有友好的统一的提示信息，同时后台应该记录详细的异常日志

界面友好性和易用性方面的

1.表单应该能够根据屏幕分辩率自动适应。在界面上让用户一次能够看到足够多的信息

2.表单应该支持 Tab 键功能，顺序为从左到右，从上到下。

3.常用的表单应该同时支持键盘操作和鼠标操作。

4.界面上控件的布局应该间距适当，标签和控件对齐，有适当的录入提示信息。

5.界面的配色应该尽量简单，尽量少使用各种刺眼的颜色

6.用户看到表单后应该就基本清楚相关功能，表单要尽量自我解释，不要设计过多的隐含在界面里面功能

数据的录入和检索

1.根据业务需要选择适合的数据录入控件

2.数据录入控件应该有完备的数据完整性和一致性校验功能

3.系统应该提供用户暂时保存录入数据的功能

4.能够自动获取数据不要让用户再去录入，能够选择录入数据不要让用户手工录入

5.数据检索条件应该适中，不应太多也不应太少。检索支持组合条件检索。

6.为了满足不同需求检索可以提供简单检索和高级检索多种方式。

7.应该在第一时间提供给用户检索数据，因此检索功能存在性能问题时候要考虑分页。

8.在检索功能较耗时的时候应该提供给用户相关的进度条显示进度

9.表格最好能够提供行显示和列显示等多种显示模式，方面用户查看数据

C#是一门支持多线程的语言，因此线程的使用也是比较常见的。由于线程的知识在 Win32 编程的时候已经说得过多，所以在.Net 中很少介绍这部分（可能.Net 不觉得这部分是它所特有的）。

那么线程相关的问题大致有如下四类（这篇文章只讨论单线程、单线程与 UI 线程这两方面的问题）。

问题一，线程的基本操作，例如：暂停、继续、停止等；

问题二，如何向线程传递参数或者从中得到其返回值；

问题三，如何使线程所占用的 CPU 不要老是百分之百；

最后一个，也是问题最多的，就是如何在子线程来控制 UI 中的控件，换句话说，就是在线程中控制窗体某些控件的显示。

对于问题一，我不建议使用 Thread 类提供的 Suspend、Resume 以及 Abort 这三个方法，前两个有问题，好像在 VS05 已经屏蔽这两个方法；对于 Abort 来说，除了资源没有得到及时释放外，有时候会出现异常。如何做呢，通过设置开关变量来完成。

对于问题二，我不建议使用静态成员来完成，仅仅为了线程而破坏类的封装有些得不偿失。那如何做呢，通过创建单独的线程类来完成。

对于问题三来说，造成这个原因是由于线程中进行不间断的循环操作，从而使 CPU 完全被子线程占有。那么处理此类问题，其实很简单，在适当的位置调用 Thread.Sleep(20)来释放所占有 CPU 资源，不要小看这 20 毫秒的睡眠，它的作用可是巨大的，可以使其他线程得到 CPU 资源，从而使你的 CPU 使用效率降下来。

看完前面的三个问题的解释，对于如何做似乎没有给出一个明确的答案，为了更好地说明如何解决这三个问题，我用一个比较完整的例子展现给大家，代码如下。

```
//--------------------------- Sub-thread class ---------------------------------------

//-------------------------------------------------------------------------------------

//---File:            clsSubThread
```

```
//---Description:    The sub-thread template class file

//---Author:          Knight

//---Date:            Aug.21, 2006

//-------------------------------------------------------------------------------------

//---------------------------{Sub-thread class}--------------------------------------

namespace ThreadTemplate

{

    using System;

    using System.Threading;

    using System.IO;

    /// <summary>

    /// Summary description for clsSubThread.

    /// </summary>

    public class clsSubThread:IDisposable

    {

        private Thread thdSubThread = null;

        private Mutex mUnique = new Mutex();



        private bool blnIsStopped;

        private bool blnSuspended;

        private bool blnStarted;

        private int nStartNum;
```

```
public bool IsStopped

{

    get{ return blnIsStopped; }

}

public bool IsSuspended

{

    get{ return blnSuspended; }

}

public int ReturnValue

{

    get{ return nStartNum;}

}




public clsSubThread( int StartNum )

{

    //

    // TODO: Add constructor logic here

    //

    blnIsStopped = true;

    blnSuspended = false;
```

```
        blnStarted = false;


        nStartNum = StartNum;
    }


    /// <summary>
    /// Start sub-thread
    /// </summary>
    public void Start()
    {
        if( !blnStarted )
        {
            thdSubThread = new Thread( new ThreadStart( SubThread ) );
            blnIsStopped = false;
            blnStarted = true;
            thdSubThread.Start();
        }
    }


    /// <summary>
    /// Thread entry function
    /// </summary>
```

```
private void SubThread()
{
    do
    {
        // Wait for resume-command if got suspend-command here
        mUnique.WaitOne();
        mUnique.ReleaseMutex();


        nStartNum++;


        Thread.Sleep(1000); // Release CPU here
    }while( blnIsStopped == false );
}


/// <summary>
/// Suspend sub-thread
/// </summary>
public void Suspend()
{
    if( blnStarted && !blnSuspended )
    {
        blnSuspended = true;
```

```
            mUnique.WaitOne();
        }
    }

    /// <summary>
    /// Resume sub-thread
    /// </summary>
    public void Resume()
    {
        if( blnStarted && blnSuspended )
        {
            blnSuspended = false;
            mUnique.ReleaseMutex();
        }
    }

    /// <summary>
    /// Stop sub-thread
    /// </summary>
    public void Stop()
    {
        if( blnStarted )
```

```
        {
            if( blnSuspended )
                Resume();


            blnStarted = false;
            blnIsStopped = true;
            thdSubThread.Join();
        }
    }
    #region IDisposable Members
    /// <summary>
    /// Class resources dispose here
    /// </summary>
    public void Dispose()
    {
        // TODO:   Add clsSubThread.Dispose implementation
        Stop();//Stop thread first
        GC.SuppressFinalize( this );
    }


    #endregion
}
```

```
}
```

那么对于调用呢，就非常简单了，如下：

```
        // Create new sub-thread object with parameters
        clsSubThread mySubThread = new clsSubThread( 5 );

        mySubThread.Start();//Start thread

        Thread.Sleep( 2000 );
        mySubThread.Suspend();//Suspend thread

        Thread.Sleep( 2000 );
        mySubThread.Resume();//Resume thread

        Thread.Sleep( 2000 );
        mySubThread.Stop();//Stop thread

        //Get thread's return value
        Debug.WriteLine( mySubThread.ReturnValue );

        //Release sub-thread object
        mySubThread.Dispose();
```

在回过头来看看前面所说的三个问题。

对于问题一来说，首先需要局部成员的支持，那么

```
private Mutex mUnique = new Mutex();



private bool blnIsStopped;

private bool blnSuspended;

private bool blnStarted;
```

光看成员名称，估计大家都已经猜出其代表的意思。接下来需要修改线程入口函数，要是这些开关变量能发挥作用，那么看看 SubThread 这个函数。

```
/// <summary>

/// Thread entry function

/// </summary>

private void SubThread()

{

    do

    {

        // Wait for resume-command if got suspend-command here

        mUnique.WaitOne();

        mUnique.ReleaseMutex();



        nStartNum++;
```

```
            Thread.Sleep(1000);

        }while( blnIsStopped == false );

    }
```

函数比较简单，不到十句，可能对于“blnIsStopped == false”这个判断来说，大家还比较好理解，这是一个普通的判断，如果当前 Stop 开关打开了，就停止循环；否则一直循环。

大家比较迷惑的可能是如下这两句：

```
            mUnique.WaitOne();

            mUnique.ReleaseMutex();
```

这两句的目的是为了使线程在 Suspend 操作的时候能发挥效果，为了解释这两句，需要结合 Suspend 和 Resume 这两个方法，它俩的代码如下。

```
    /// <summary>

    /// Suspend sub-thread

    /// </summary>

    public void Suspend()

    {

        if( blnStarted && !blnSuspended )

        {

            blnSuspended = true;

            mUnique.WaitOne();

        }

    }
```

```
/// <summary>

/// Resume sub-thread

/// </summary>

public void Resume()
{
    if( blnStarted && blnSuspended )
    {
        blnSuspended = false;

        mUnique.ReleaseMutex();
    }

}
```

为了更好地说明，还需要先简单说说 Mutex 类型。对于此类型对象，当调用对象的 WaitOne 之后，如果此时没有其他线程对它使用的时候，就立刻获得信号量，继续执行代码；当再调用 ReleaseMutex 之前，如果再调用对象的 WaitOne 方法，就会一直等待，直到获得信号量的调用 ReleaseMutex 来进行释放。这就好比卫生间的使用，如果没有人使用则可以直接使用，否则只有等待。

明白了这一点后，再来解释这两句所能出现的现象。

```
mUnique.WaitOne();
mUnique.ReleaseMutex();
```

当在线程函数中，执行到“mUnique.WaitOne();”这一句的时候，如果此时外界没有发送 Suspend 消息，也就是信号量没有被占用，那么这一句可以立刻返回。那么为什么要紧接着释放呢，因为不能总占着信号量，立即释放信号量是避免在发送 Suspend 命令的时候出现等待；如果此时外界已经发送了 Suspend 消息，也就是说信号量已经被占用，此时“mUnique.WaitOne();”不能立刻返回，需要等到信号量被释放才能继续进行，也就是需要调用 Resume 的时候，“mUnique.WaitOne();”才能获得信号量进行继续执行。这样才能达到真正意义上的 Suspend 和 Resume。

至于线程的 Start 和 Stop 来说，相对比较简单，这里我就不多说了。

现在再来分析一下问题二，其实例子比较明显，是通过构造函数和属性来完成参数和返回值，这一点我也不多说了。如果线程参数比较多的话，可以考虑属性来完成，类似于返回值。

问题三，我就更不用多说了。有人说了，如果子线程中的循环不能睡眠怎么办，因为睡眠的话，有时会造成数据丢失，这方面的可以借鉴前面 Suspend 的做法，如果更复杂，则牵扯到多线程的同步问题，这部分我会稍后单独写一篇文章。

前三个问题解决了，该说说最常见的问题，如何在子线程中控制窗体控件。这也是写线程方面程序经常遇到的，其实我以前写过两篇文章，都对这方面做了部分介绍。那么大家如果有时间的话，不妨去看看。

http://blog.csdn.net/knight94/archive/2006/03/16/626584.aspx

http://blog.csdn.net/knight94/archive/2006/05/27/757351.aspx

首先说说，为什么不能直接在子线程中操纵 UI 呢。原因在于子线程和 UI 线程属于不同的上下文，换句比较通俗的话说，就好比两个人在不同的房间里一样，那么要你直接操作另一个房间里的东西，恐怕不行罢，那么对于子线程来说也一样，不能直接操作 UI 线程中的对象。

那么如何在子线程中操纵 UI 线程中的对象呢，.Net 提供了 Invoke 和 BeginInvoke 这两种方法。简单地说，就是子线程发消息让 UI 线程来完成相应的操作。

这两个方法有什么区别，这在我以前的文章已经说过了，Invoke 需要等到所调函数的返回，而 BeginInvoke 则不需要。

用这两个方法需要注意的，有如下三点：

第一个是由于 Invoke 和 BeginInvoke 属于 Control 类型的成员方法，因此调用的时候，需要得到 Control 类型的对象才能触发，也就是说你要触发窗体做什么操作或者窗体上某个控件做什么操作，需要把窗体对象或者控件对象传递到线程中。

第二个，对于 Invoke 和 BeginInvoke 接受的参数属于一个 delegate 类型，我在以前的文章中使用的是 MethodInvoker，这是.Net 自带的一个 delegate 类型，而并不意味着在使用 Invoke 或者 BeginInvoke 的时候只能用它。参看我给的第二篇文章（《如何弹出一个模式窗口来显示进度条》），会有很多不同的 delegate 定义。

最后一个，使用 Invoke 和 BeginInvoke 有个需要注意的，就是当子线程在 Form_Load 开启的时候，会遇到异常，这是因为触发 Invoke 的对象还没有完全初始化完毕。处理此类问题，在开启线程之前显式的调用“this.Show();”，来使窗体显示在线程开启之前。如果此时只是开启线程来初始化显示数据，那我建议你不要使用子线程，用 Splash 窗体的效果可能更好。这方面可以参看如下的例子。

http://www.syncfusion.com/FAQ/WindowsForms/FAQ_c95c.aspx#q621q

线程的四个相关问题已经说完了，这篇文章只说了单线程，以及单线程与 UI 线程交互的问题。其中涉及到的方法不一定是唯一的，因为.Net 还提供了其他类来扶助线程操作，这里就不一一罗列。至于多线程之间的同步，我会稍后专门写篇文章进行描述。

VS2005 中更新其他线程访问界面线程控件的方法

作者：Depraved_Survival | 录入时间：2007-9-4 | 点击：17 次　　打印此文章 | 字体：大 中 小

VS2005 中，界面线程中的控件如果由其他线程进行更新时，编译器会自动抛出异常来避免这种不安全的跨线程访问方式。解决这个问题的一种方法是在界面线程中编写控件内容更新程序，并声明委托，利用 Invoke 方法进行实现。具体实现方法如下。

假设我们的 From1 中有一个 TextBox 空间，名为 txtBox，From1 在 Load 的时候启动一个线程 thread，线程处理函数为 threadProc，在该线程中将 Hello the world 写入 txtBox 控件中。

对于上面的问题，在以往 VS2003 中我们只需在 thread 线程的处理函数中加入 this.txtBox.Text="Hello the world!"即可，而在 VS2005 中同样的写法编译器会抛出异常。在 VS2005 中正确的调用方法是先声明用于更新空间的委托，在定义一个更新控件用的方法，然后在线程函数中通过 From1 的 InvokeRequired 方法判断是否需要采用 Invoke 方式，如果需要使用 Invoke 方式访问，否则采用 VS2003 中的方式更新控件内容，代码如下：

```
//声明更新控件用的代理委托
protected delegate void UpdateControlText(string strText);
//定义更新控件的方法
protected void updateControlText(string strText)
{
    txtBox.Text = strText;
    return;
}
```

需要说明的是，上述委托和方法为 From1 对应类对象的成员，委托的参数必须与方法的参数完全相同。

在线程中需要更新控件的位置中添加如下代码：

```
if (this.InvokeRequired)
 {
     UpdateControlText update = new UpdateControlText(updateControlText);
     this.Invoke(update, "Hello the world!");
 }
 else
 {
     this.txtBox.Text="Hello the world!";
 }
```

Invoke 方法中第一个参数为委托类型的对象，构造该对象的时候采用 updateControlText 作为构造参数。从第二个参数开始为一个 params object [] objParams 的队列，可以容纳任意多任意类型的参数，但是针对每次调用来说必须要指定类型个数与 UpdateControlText 完全相同的参数表。虽然参数不同时编译器不会报错，但是会在运行中弹出异常。

借助 WebService 实现多线程上传文件 - 愚翁专栏 - CSDNBlog 正在处理您的请求...

借助 WebService 实现多线程上传文件

在 WebService 的帮助下，进行多线程上传文件是非常简单。因此我只做个简单的例子，那么如果想要实现此功能的朋友，可以在我的基础上进行扩展。

首先说说服务器端，只需要提供一个能允许多线程写文件的函数即可，具体代码如下。

```
[WebMethod]
public bool UploadFileData( string FileName, int StartPosition, byte[] bData )
{
    string strFullName = Server.MapPath( "Uploads" ) + @"\" + FileName;
    FileStream fs = null;
    try
    {
        fs = new FileStream( strFullName, FileMode.OpenOrCreate,
            FileAccess.Write, FileShare.Write );
    }
    catch( IOException err )
    {
        Session["ErrorMessage"] = err.Message;
        return false;
    }

    using( fs )
    {
        fs.Position = StartPosition;
        fs.Write( bData, 0, bData.Length );
    }
    return true;
}
```

其中“Uploads”是在服务程序所在目录下的一个子目录，需要设置 ASPNET 用户对此目录具有可写权限。

相对于服务器端来说，客户端要稍微复杂一些，因为要牵扯到多线程的问题。为了更好的传递参数，我用一个线程类来完成。具体如下。

```
    public delegate void UploadFileData( string FileName, int StartPos, byte[]
bData );

    /// <summary>
    /// FileThread: a class for sub-thread
    /// </summary>
    sealed class FileThread
    {
        private int nStartPos;
```

```
        private int nTotalBytes;
        private string strFileName;
        public static UploadFileData UploadHandle;

        /// <summary>
        /// Constructor
        /// </summary>
        /// <param name="StartPos"></param>
        /// <param name="TotalBytes"></param>
        /// <param name="FileName"></param>
        public FileThread( int StartPos, int TotalBytes, string FileName )
        {
            //Init thread variant
            nStartPos = StartPos;
            nTotalBytes = TotalBytes;
            strFileName = FileName;

            //Only for debug
            Debug.WriteLine( string.Format( "File name:{0} position: {1} total
byte:{2}",
                strFileName, nStartPos, nTotalBytes ) );
        }

        /// <summary>
        /// Sub-thread entry function
        /// </summary>
        /// <param name="stateinfo"></param>
        public void UploadFile( object stateinfo )
        {
            int nRealRead, nBufferSize;
            const int BUFFER_SIZE = 10240;

            using( FileStream fs = new FileStream( strFileName,
                        FileMode.Open, FileAccess.Read,
                        FileShare.Read ) )
            {
                string sName = strFileName.Substring( strFileName.LastIndexOf(
"\\" ) + 1 );
                byte[] bBuffer = new byte[BUFFER_SIZE];//Init 10k buffer
                fs.Position = nStartPos;
                nRealRead = 0;
                do
                {
                    nBufferSize = BUFFER_SIZE;
```

```
                if( nRealRead + BUFFER_SIZE > nTotalBytes )
                    nBufferSize = nTotalBytes - nRealRead;

                nBufferSize = fs.Read( bBuffer, 0, nBufferSize );
                if( nBufferSize == BUFFER_SIZE )
                    UploadHandle( sName,
                        nRealRead + nStartPos,
                        bBuffer );
                else if( nBufferSize > 0 )
                {
                    //Copy data
                    byte[] bytData = new byte[nBufferSize];
                    Array.Copy( bBuffer,0, bytData, 0, nBufferSize );

                    UploadHandle( sName,
                        nRealRead + nStartPos,
                        bytData );
                }

                nRealRead += nBufferSize;
            }
            while( nRealRead < nTotalBytes );
        }
        //Release signal
        ManualResetEvent mr = stateinfo as ManualResetEvent;
        if( mr != null )
            mr.Set();
    }
}
```

那么在执行的时候，要创建线程类对象，并为每一个每个线程设置一个信号量，从而能在所有线程都结束的时候得到通知，大致的代码如下。

```
FileInfo fi = new FileInfo( txtFileName.Text );
if( fi.Exists )
{
    btnUpload.Enabled = false;//Avoid upload twice

    //Init signals
    ManualResetEvent[] events = new ManualResetEvent[5];

    //Devide blocks
    int nTotalBytes = (int)( fi.Length / 5 );
    for( int i = 0; i < 5; i++ )
    {
```

```
            events[i] = new ManualResetEvent( false );
            FileThread thdSub = new FileThread(
                i * nTotalBytes,
                ( fi.Length - i * nTotalBytes ) > nTotalBytes ?
nTotalBytes:(int)( fi.Length - i * nTotalBytes ),
                fi.FullName );
            ThreadPool.QueueUserWorkItem( new WaitCallback( thdSub.UploadFile ),
events[i] );
        }

        //Wait for threads finished
        WaitHandle.WaitAll( events );

        //Reset button status
        btnUpload.Enabled = true;
    }
```

总体来说，程序还是相对比较简单，而我也只是做了个简单例子而已，一些细节都没有进行处理。

本来想打包提供给大家下载，没想到 CSDN 的 Blog 对于这点做的太差，老是异常。
如下是客户端的完整代码。

```
//--------------------------- Multi-thread Upload Demo
---------------------------------------
//-----------------------------------------------------------------------------------------------
//---File:             frmUpload
//---Description:    The multi-thread upload form file to demenstrate howto use
multi-thread to
//                    upload files
//---Author:          Knight
//---Date:            Oct.12, 2006
//-----------------------------------------------------------------------------------------------
//---------------------------{Multi-thread Upload
Demo}---------------------------------------
using System;
using System.Drawing;
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;
using System.Data;

namespace CSUpload
{
    using System.IO;
    using System.Diagnostics;
```

```
using System.Threading;
using WSUploadFile;//Web-service reference namespace

/// <summary>
/// Summary description for Form1.
/// </summary>
public class frmUpload : System.Windows.Forms.Form
{
    private System.Windows.Forms.TextBox txtFileName;
    private System.Windows.Forms.Button btnBrowse;
    private System.Windows.Forms.Button btnUpload;
    /// <summary>
    /// Required designer variable.
    /// </summary>
    private System.ComponentModel.Container components = null;

    public frmUpload()
    {
        //
        // Required for Windows Form Designer support
        //
        InitializeComponent();

        //
        // TODO: Add any constructor code after InitializeComponent call
        //
    }

    /// <summary>
    /// Clean up any resources being used.
    /// </summary>
    protected override void Dispose( bool disposing )
    {
        if( disposing )
        {
            if (components != null)
            {
                components.Dispose();
            }
        }
        base.Dispose( disposing );
    }

    #region Windows Form Designer generated code
```

```
        /// <summary>
        /// Required method for Designer support - do not modify
        /// the contents of this method with the code editor.
        /// </summary>
        private void InitializeComponent()
        {
            this.txtFileName = new System.Windows.Forms.TextBox();
            this.btnBrowse = new System.Windows.Forms.Button();
            this.btnUpload = new System.Windows.Forms.Button();
            this.SuspendLayout();
            //
            // txtFileName
            //
            this.txtFileName.Location = new System.Drawing.Point(16, 24);
            this.txtFileName.Name = "txtFileName";
            this.txtFileName.Size = new System.Drawing.Size(248, 20);
            this.txtFileName.TabIndex = 0;
            this.txtFileName.Text = "";
            //
            // btnBrowse
            //
            this.btnBrowse.Location = new System.Drawing.Point(272, 24);
            this.btnBrowse.Name = "btnBrowse";
            this.btnBrowse.TabIndex = 1;
            this.btnBrowse.Text = "&Browse...";
            this.btnBrowse.Click += new
System.EventHandler(this.btnBrowse_Click);
            //
            // btnUpload
            //
            this.btnUpload.Location = new System.Drawing.Point(272, 56);
            this.btnUpload.Name = "btnUpload";
            this.btnUpload.TabIndex = 2;
            this.btnUpload.Text = "&Upload";
            this.btnUpload.Click += new
System.EventHandler(this.btnUpload_Click);
            //
            // frmUpload
            //
            this.AutoScaleBaseSize = new System.Drawing.Size(5, 13);
            this.ClientSize = new System.Drawing.Size(370, 111);
            this.Controls.Add(this.btnUpload);
            this.Controls.Add(this.btnBrowse);
            this.Controls.Add(this.txtFileName);
```

```
            this.FormBorderStyle =
System.Windows.Forms.FormBorderStyle.FixedSingle;
            this.MaximizeBox = false;
            this.Name = "frmUpload";
            this.Text = "Upload";
            this.Load += new System.EventHandler(this.frmUpload_Load);
            this.ResumeLayout(false);

        }
        #endregion

        /// <summary>
        /// The main entry point for the application.
        /// </summary>
        static void Main()
        {
            Application.Run(new frmUpload());
        }

        private FileUpload myUpload = new FileUpload();
        private void UploadData( string FileName, int StartPos, byte[] bData )
        {
            //Call web service upload
            myUpload.UploadFileData( FileName, StartPos, bData );
        }

        private void btnUpload_Click(object sender, System.EventArgs e)
        {
            FileInfo fi = new FileInfo( txtFileName.Text );
            if( fi.Exists )
            {
                btnUpload.Enabled = false;//Avoid upload twice

                //Init signals
                ManualResetEvent[] events = new ManualResetEvent[5];

                //Devide blocks
                int nTotalBytes = (int)( fi.Length / 5 );
                for( int i = 0; i < 5; i++ )
                {
                    events[i] = new ManualResetEvent( false );
                    FileThread thdSub = new FileThread(
                        i * nTotalBytes,
                        ( fi.Length - i * nTotalBytes ) > nTotalBytes ?
```

```
nTotalBytes:(int)( fi.Length - i * nTotalBytes ),
                            fi.FullName );
                        ThreadPool.QueueUserWorkItem( new WaitCallback(
thdSub.UploadFile ), events[i] );
                    }

                    //Wait for threads finished
                    WaitHandle.WaitAll( events );

                    //Reset button status
                    btnUpload.Enabled = true;
                }

            }

            private void frmUpload_Load(object sender, System.EventArgs e)
            {
                FileThread.UploadHandle = new UploadFileData( this.UploadData );
            }

            private void btnBrowse_Click(object sender, System.EventArgs e)
            {
                if( fileOpen.ShowDialog() == DialogResult.OK )
                    txtFileName.Text = fileOpen.FileName;
            }
            private OpenFileDialog fileOpen = new OpenFileDialog();



        }

        public delegate void UploadFileData( string FileName, int StartPos, byte[]
bData );

        /// <summary>
        /// FileThread: a class for sub-thread
        /// </summary>
        sealed class FileThread
        {
            private int nStartPos;
            private int nTotalBytes;
            private string strFileName;
            public static UploadFileData UploadHandle;

            /// <summary>
```

```
        /// Constructor
        /// </summary>
        /// <param name="StartPos"></param>
        /// <param name="TotalBytes"></param>
        /// <param name="FileName"></param>
        public FileThread( int StartPos, int TotalBytes, string FileName )
        {
            //Init thread variant
            nStartPos = StartPos;
            nTotalBytes = TotalBytes;
            strFileName = FileName;

            //Only for debug
            Debug.WriteLine( string.Format( "File name:{0} position: {1} total
byte:{2}",
                strFileName, nStartPos, nTotalBytes ) );
        }

        /// <summary>
        /// Sub-thread entry function
        /// </summary>
        /// <param name="stateinfo">></param>
        public void UploadFile( object stateinfo )
        {
            int nRealRead, nBufferSize;
            const int BUFFER_SIZE = 10240;

            using( FileStream fs = new FileStream( strFileName,
                        FileMode.Open, FileAccess.Read,
                        FileShare.Read ) )
            {
                string sName = strFileName.Substring( strFileName.LastIndexOf(
"\\" ) + 1 );
                byte[] bBuffer = new byte[BUFFER_SIZE];//Init 10k buffer
                fs.Position = nStartPos;
                nRealRead = 0;
                do
                {
                    nBufferSize = BUFFER_SIZE;
                    if( nRealRead + BUFFER_SIZE > nTotalBytes )
                        nBufferSize = nTotalBytes - nRealRead;

                    nBufferSize = fs.Read( bBuffer, 0, nBufferSize );
                    if( nBufferSize == BUFFER_SIZE )
```

```
                    UploadHandle( sName,
                        nRealRead + nStartPos,
                        bBuffer );
                else if( nBufferSize > 0 )
                {
                    //Copy data
                    byte[] bytData = new byte[nBufferSize];
                    Array.Copy( bBuffer,0, bytData, 0, nBufferSize );

                    UploadHandle( sName,
                        nRealRead + nStartPos,
                        bytData );
                }

                nRealRead += nBufferSize;
            }
            while( nRealRead < nTotalBytes );
        }
        //Release signal
        ManualResetEvent mr = stateinfo as ManualResetEvent;
        if( mr != null )
            mr.Set();
    }
  }

}
```

Trackback: http://tb.blog.csdn.net/TrackBack.aspx?PostId=1332160

[收藏到我的网摘]　Knight94 发表于 2006 年 10 月 12 日 19:47:00

相关文章：
用 WebClient.UploadData 方法 上载文件数据 2005-04-17 sunsnow8

<<《Effective C#》Item 19：推荐在继承中使用接口 | 《Effective C#》Item 2：定义常量的两种方法 >>

小玉 发表于 2006-10-26 09:36:00 IP: 218.24.228.*

问个问题:
水晶报表可以动态创建吗?或者
.NET 里什么报表可以实现动态创建报表?
我现在要做在程序动行时设计报表,可以给我些提示吗
谢谢了

Night 发表于 2006-11-25 20:08:00 IP: 219.131.239.*

请问

我在运行时候

这句//Wait for threads finished

WaitHandle.WaitAll(events);

出现报错

未处理 NotSupportedException

不支持一个 STA 线程上针对多个句柄的 WaitAll

这个是什么引起的

但是注释了这句也可以运行

去掉 受不受 影响？

Night 发表于 2006-11-26 10:11:00 IP: 219.131.240.*

你好！

我按照你的方法 去做了

去掉了 Program.cs 内的[STAThread]

但是 if (fileOpen.ShowDialog() == DialogResult.OK)

这里显示错误

在可以调用 OLE 之前，必须将当前线程设置为单线程单元(STA)模式。请确保您的 Main 函数带有 STAThreadAttribute 标记。

只有将调试器附加到该进程才会引发此异常。

这种该怎么改？

谢谢

knight94 发表于 2006-11-26 10:55:00 IP: 218.71.239.*

你的窗体上引用了什么特殊的 com。

knight94 发表于 2006-11-26 09:15:00 IP: 218.71.239.*

to Night

你这个错误，只要把 main 入口函数上的[STAThread]属性标示去掉就行了。

bj20082005 发表于 2007-01-23 17:11:56 IP:

你这个 FileUpload myUpload = new FileUpload();是引用哪个组件呀

FlyBird2004 发表于 2007-04-04 11:30:05 IP: 222.82.221.*

可以将这个程序的打包帮忙发一份嘛？因为调试了半天也没有成功。不胜感激！myg_mail@sina.com

新手老问题---------跨线程的控件访问

电子科技大学 03 级 02 班 周银辉

新手经常会遇到这样的问题: a 线程去访问 b 线程的控件,编译器报错(.net1.0 编译时好像不会报,.net2.0 是肯定会的).

解决方法有 3 种:
1, 不安全的方法: 将 Control.CheckForIllegalCrossThreadCalls 设置为 false (.net1.0 中没有)
2,安全的方法: 异步委托
3, 安全的方法: 就是使用 BackgroundWorker 来替代你自己创建的线程(.net1.0 中没有)

以下是示例代码

```
using System;
using System.ComponentModel;
using System.Threading;
using System.Windows.Forms;

namespace CrossThreadDemo
{
    public class Form1 : Form
    {
        // This delegate enables asynchronous calls for setting
        // the text property on a TextBox control.
        delegate void SetTextCallback(string text);
        // This thread is used to demonstrate both thread-safe and
        // unsafe ways to call a Windows Forms control.
        private Thread demoThread = null;
        // This BackgroundWorker is used to demonstrate the
        // preferred way of performing asynchronous operations.
        private BackgroundWorker backgroundWorker1;
        private TextBox textBox1;
        private Button setTextUnsafeBtn;
        private Button setTextSafeBtn;
        private Button setTextBackgroundWorkerBtn;
        private System.ComponentModel.IContainer components = null;

        public Form1()
        {
            InitializeComponent();
        }

        protected override void Dispose(bool disposing)
```

```
{
    if (disposing && (components != null))
    {
        components.Dispose();
    }
    base.Dispose(disposing);
}

// This event handler creates a thread that calls a
// Windows Forms control in an unsafe way.
private void setTextUnsafeBtn_Click(
object sender,
EventArgs e)
{
    this.demoThread =
    new Thread(new ThreadStart(this.ThreadProcUnsafe));

    this.demoThread.Start();
}

// This method is executed on the worker thread and makes
// an unsafe call on the TextBox control.
private void ThreadProcUnsafe()
{
    this.textBox1.Text = "This text was set unsafely.";
}

// This event handler creates a thread that calls a
// Windows Forms control in a thread-safe way.
private void setTextSafeBtn_Click(
object sender,
EventArgs e)
{
    this.demoThread =
    new Thread(new ThreadStart(this.ThreadProcSafe));

    this.demoThread.Start();
}

// This method is executed on the worker thread and makes
// a thread-safe call on the TextBox control.
private void ThreadProcSafe()
{
    this.SetText("This text was set safely.");
```

```
}

// This method demonstrates a pattern for making thread-safe
// calls on a Windows Forms control.
//
// If the calling thread is different from the thread that
// created the TextBox control, this method creates a
// SetTextCallback and calls itself asynchronously using the
// Invoke method.
//
// If the calling thread is the same as the thread that created
// the TextBox control, the Text property is set directly.

private void SetText(string text)
{
    // InvokeRequired required compares the thread ID of the
    // calling thread to the thread ID of the creating thread.
    // If these threads are different, it returns true.
    if (this.textBox1.InvokeRequired)
    {
        SetTextCallback d = new SetTextCallback(SetText);
        this.Invoke(d, new object[] { text });
    }
    else
    {
        this.textBox1.Text = text;
    }
}

// This event handler starts the form's
// BackgroundWorker by calling RunWorkerAsync.
//
// The Text property of the TextBox control is set
// when the BackgroundWorker raises the RunWorkerCompleted
// event.
private void setTextBackgroundWorkerBtn_Click(
object sender,
EventArgs e)
{
    this.backgroundWorker1.RunWorkerAsync();
}

// This event handler sets the Text property of the TextBox
// control. It is called on the thread that created the
```

```
        // TextBox control, so the call is thread-safe.
        //
        // BackgroundWorker is the preferred way to perform asynchronous
        // operations.

        private void backgroundWorker1_RunWorkerCompleted(
        object sender,
        RunWorkerCompletedEventArgs e)
        {
            this.textBox1.Text =
            "This text was set safely by BackgroundWorker.";
        }

        Windows Form Designer generated code#region Windows Form Designer generated code

        private void InitializeComponent()
        {
            this.textBox1 = new System.Windows.Forms.TextBox();
            this.setTextUnsafeBtn = new System.Windows.Forms.Button();
            this.setTextSafeBtn = new System.Windows.Forms.Button();
            this.setTextBackgroundWorkerBtn = new System.Windows.Forms.Button();
            this.backgroundWorker1 = new System.ComponentModel.BackgroundWorker();
            this.SuspendLayout();
            //
            // textBox1
            //
            this.textBox1.Location = new System.Drawing.Point(12, 12);
            this.textBox1.Name = "textBox1";
            this.textBox1.Size = new System.Drawing.Size(240, 20);
            this.textBox1.TabIndex = 0;
            //
            // setTextUnsafeBtn
            //
            this.setTextUnsafeBtn.Location = new System.Drawing.Point(15, 55);
            this.setTextUnsafeBtn.Name = "setTextUnsafeBtn";
            this.setTextUnsafeBtn.TabIndex = 1;
            this.setTextUnsafeBtn.Text = "Unsafe Call";
            this.setTextUnsafeBtn.Click += new
System.EventHandler(this.setTextUnsafeBtn_Click);
            //
            // setTextSafeBtn
            //
            this.setTextSafeBtn.Location = new System.Drawing.Point(96, 55);
            this.setTextSafeBtn.Name = "setTextSafeBtn";
```

```
            this.setTextSafeBtn.TabIndex = 2;
            this.setTextSafeBtn.Text = "Safe Call";
            this.setTextSafeBtn.Click += new System.EventHandler(this.setTextSafeBtn_Click);
            //
            // setTextBackgroundWorkerBtn
            //
            this.setTextBackgroundWorkerBtn.Location = new System.Drawing.Point(177, 55);
            this.setTextBackgroundWorkerBtn.Name = "setTextBackgroundWorkerBtn";
            this.setTextBackgroundWorkerBtn.TabIndex = 3;
            this.setTextBackgroundWorkerBtn.Text = "Safe BW Call";
            this.setTextBackgroundWorkerBtn.Click += new
System.EventHandler(this.setTextBackgroundWorkerBtn_Click);
            //
            // backgroundWorker1
            //
            this.backgroundWorker1.RunWorkerCompleted += new
System.ComponentModel.RunWorkerCompletedEventHandler(this.backgroundWorker1_RunWork
erCompleted);
            //
            // Form1
            //
            this.ClientSize = new System.Drawing.Size(268, 96);
            this.Controls.Add(this.setTextBackgroundWorkerBtn);
            this.Controls.Add(this.setTextSafeBtn);
            this.Controls.Add(this.setTextUnsafeBtn);
            this.Controls.Add(this.textBox1);
            this.Name = "Form1";
            this.Text = "Form1";
            this.ResumeLayout(false);
            this.PerformLayout();

        }

        #endregion

        [STAThread]
        static void Main()
        {
            Application.EnableVisualStyles();
            Application.Run(new Form1());
        }
    }
}
```

在使用方法 2 时的注意事项：不要将除了控件访问外其他逻辑代码放到委托的回调方法中.

如何弹出一个模式窗口来显示进度条

最近看了好多人问这方面的问题，以前我也写过一篇 blog，里面说了如何在子线程中控制进度条。但目前大多数环境，需要弹出模式窗口，来显示进度条，那么只需要在原先的基础上稍作修改即可。

首先是进度条窗体，需要在上面添加进度条，然后去掉 ControlBox。除此外，还要增加一个方法，用来控制进度条的增加幅度，具体如下：

```
/// <summary>
/// Increase process bar
/// </summary>
/// <param name="nValue">the value increased</param>
/// <returns></returns>
public bool Increase( int nValue )
{
    if( nValue > 0 )
    {
        if( prcBar.Value + nValue < prcBar.Maximum )
        {
            prcBar.Value += nValue;
            return true;
        }
        else
        {
            prcBar.Value = prcBar.Maximum;
            this.Close();
            return false;
        }
    }
    return false;
}
```

接着就是主窗体了，如何进行操作了，首先需要定义两个私有成员，一个委托。其中一个私有成员是保存当前进度条窗体对象，另一个是保存委托方法（即增加进度条尺度），具体如下：

```
private frmProcessBar myProcessBar = null;

private delegate bool IncreaseHandle( int nValue );

private IncreaseHandle myIncrease = null;
```

接着要在主窗体中提供函数来打开进度条窗体，如下：

```
/// <summary>
/// Open process bar window
/// </summary>
private void ShowProcessBar()
{
    myProcessBar = new frmProcessBar();
    // Init increase event
    myIncrease = new IncreaseHandle( myProcessBar.Increase );
    myProcessBar.ShowDialog();
    myProcessBar = null;

}
```

那么现在就可以开始创建线程来运行，具体如下：

```
/// <summary>
/// Sub thread function
/// </summary>
private void ThreadFun()
{
    MethodInvoker mi = new MethodInvoker( ShowProcessBar );
    this.BeginInvoke( mi );

    Thread.Sleep( 1000 );//Sleep a while to show window
    bool blnIncreased = false;
    object objReturn = null;
    do
    {
        Thread.Sleep( 50 );
        objReturn = this.Invoke( this.myIncrease,
            new object[]{ 2 } );
       blnIncreased = (bool)objReturn ;
    }
    while( blnIncreased );
}
```

注意以上，在打开进度条窗体和增加进度条进度的时候，一个用的是 BeginInvoke，一个是 Invoke，这里的区别是 BeginInvoke 不需要等待方法运行完毕，而 Invoke 是要等待方法运行完毕。还有一点，此处用返回值来判断进度条是否到头了，如果需要有其他的控制，可以类似前面的方法来进行扩展。

启动线程，可以如下：

```
Thread thdSub = new Thread( new ThreadStart( ThreadFun ) );
thdSub.Start();
```

这样，一个用模式打开进度条窗体就做完了。

用户不喜欢反应慢的程序。在执行耗时较长的操作时，使用多线程是明智之举，它可以提高程序 UI 的响应速度，使得一切运行显得更为快速。在 Windows 中进行多线程编程曾经是 C++开发人员的专属特权，但是现在，可以使用所有兼容 Microsoft .NET 的语言来编写。

不过 Windows 窗体体系结构对线程使用制定了严格的规则。如果只是编写单线程应用程序，则没必要知道这些规则，这是因为单线程的代码不可能违反这些规则。然而，一旦采用多线程，就需要理解 Windows 窗体中最重要的一条线程规则：除了极少数的例外情况，否则都不要在它的创建线程以外的线程中使用控件的任何成员。本规则的例外情况有文档说明，但这样的情况非常少。这适用于其类派生自 System.Windows.Forms.Control 的任何对象，其中几乎包括 UI 中的所有元素。所有的 UI 元素（包括表单本身）都是从 Control 类派生的对象。此外，这条规则的结果是一个被包含的控件（如，包含在一个表单中的按钮）必须与包含它控件位处于同一个线程中。也就是说，一个窗口中的所有控件属于同一个 UI 线程。实际中，大部分 Windows 窗体应用程序最终都只有一个线程，所有 UI 活动都发生在这个线程上。这个线程通常称为 UI 线程。这意味着您不能调用用户界面中任意控件上的任何方法，除非在该方法的文档说明中指出可以调用。该规则的例外情况（总有文档记录）非常少而且它们之间关系也不大。请注意，以下代码是非法的：

```
        private Thread myThread;

        private void Form1_Load(object sender, EventArgs e)

        {

            myThread = new Thread(new ThreadStart(RunsOnWorkerThread));

            myThread.Start();

        }

        private void RunsOnWorkerThread()

        {

            label1.Text = "myThread 线程调用 UI 控件";

    }
```

如果您在 .NET Framework 1.0 版本中尝试运行这段代码，也许会侥幸运行成功，或者初看起来是如此。这就是多线程错误中的主要问题，即它们并不会立即显现出来。甚至当出现了一些错误时，在第一次演示程序之前一切看起来也都很正常。但不要搞错 —我刚才显示的这段代码明显违反了规则，并且可以预见，任何抱希望于“试运行时良好，应该就没有问题”的人在即将到来的调试期是会付出沉重代价的。

下面我们来看看有哪些方法可以解决这一问题。

一、System.Windows.Forms.MethodInvoker 类型是一个系统定义的委托，用于调用不带参数的方法。

```
private Thread myThread;
private void Form1_Load(object sender, EventArgs e)
{
    myThread = new Thread(new ThreadStart(RunsOnWorkerThread));
    myThread.Start();
}
private void RunsOnWorkerThread()
{
    MethodInvoker mi = new MethodInvoker(SetControlsProp);
    BeginInvoke(mi);
}
private void SetControlsProp()
{
    label1.Text = "myThread 线程调用 UI 控件";
}
```

二、直接用 System.EventHandle（可带参数）

```
private Thread myThread;
private void Form1_Load(object sender, EventArgs e)
{
    myThread = new Thread(new ThreadStart(RunsOnWorkerThread));
    myThread.Start();
}
```

```
private void RunsOnWorkerThread()
{
    //DoSomethingSlow();
    string pList = "myThread 线程调用 UI 控件";
    label1.BeginInvoke(new System.EventHandler(UpdateUI), pList);
}
//直接用 System.EventHandler,没有必要自定义委托
private void UpdateUI(object o, System.EventArgs e)
{
   //UI 线程设置 label1 属性
    label1.Text = o.ToString() + "成功!";
}
```

三、包装 Control.Invoke

虽然第二个方法中的代码解决了这个问题，但它相当繁琐。如果辅助线程希望在结束时提供更多的反馈信息，而不是简单地给出“Finished!”消息，则 BeginInvoke 过于复杂的使用方法会令人生畏。为了传达其他消息，例如“正在处理”、“一切顺利”等等，需要设法向 UpdateUI 函数传递一个参数。可能还需要添加一个进度栏以提高反馈能力。这么多次调用 BeginInvoke 可能导致辅助线程受该代码支配。这样不仅会造成不便，而且考虑到辅助线程与 UI 的协调性，这样设计也不好。对这些进行分析之后，我们认为包装函数可以解决这两个问题。

```
private Thread myThread;
private void Form1_Load(object sender, EventArgs e)
{
    myThread = new Thread(new ThreadStart(RunsOnWorkerThread));
    myThread.Start();
}
```

```
private void RunsOnWorkerThread()

{

    ////DoSomethingSlow();

    for (int i = 0; i < 100; i++)

    {

        ShowProgress( Convert.ToString(i)+"%", i);

        Thread.Sleep(100);

    }

}

public void ShowProgress(string msg, int percentDone)

{

    // Wrap the parameters in some EventArgs-derived custom class:

    System.EventArgs e = new MyProgressEvents(msg, percentDone);

    object[] pList = { this, e };



    BeginInvoke(new MyProgressEventsHandler(UpdateUI), pList);

}

private delegate void MyProgressEventsHandler(object sender, MyProgressEvents e);

private void UpdateUI(object sender, MyProgressEvents e)

{

    lblStatus.Text = e.Msg;

    myProgressControl.Value = e.PercentDone;
```

```
        }

    public class MyProgressEvents : EventArgs

    {

        public string Msg;

        public int PercentDone;

        public MyProgressEvents(string msg, int per)

        {

            Msg = msg;

            PercentDone = per;

        }

 }
```

ShowProgress 方法对将调用引向正确线程的工作进行封装。这意味着辅助线程代码不再担心需要过多关注 UI 细节，而只要定期调用 ShowProgress 即可。

如果我提供一个设计为可从任何线程调用的公共方法，则完全有可能某人会从 UI 线程调用这个方法。在这种情况下，没必要调用 BeginInvoke，因为我已经处于正确的线程中。调用 Invoke 完全是浪费时间和资源，不如直接调用适当的方法。为了避免这种情况，Control 类将公开一个称为 InvokeRequired 的属性。这是“只限 UI 线程”规则的另一个例外。它可从任何线程读取，如果调用线程是 UI 线程，则返回假，其他线程则返回真。这意味着我可以按以下方式修改包装：

```
        public void ShowProgress(string msg, int percentDone)

        {

            if (InvokeRequired)

            {

                // As before

                //...
```

```
        }
        else
        {
            // We're already on the UI thread just
            // call straight through.
            UpdateUI(this, new MyProgressEvents(msg,PercentDone));
        }
    }
```

在当前线程中访问主线程的控件

在做广州移动集团短信平台的时候，需要定时刷新 SP 的告警信息。于是创建了一个 Timer 对象，在 Timer 对象的 Elapsed 事件中需要访问主窗体中的一个 LinkLabel。刚开始直接用 this.lnklblMsg.Text="你有"+intMsg+"条新告警信息";结果运行出错：System.InvalidOperationException:线程间操作无效：从不是创建控件“lnklblMsg”的线程访问它。主要原因是控件“lnklblMsg”是由主窗体创建的，而访问它是在 Timer 创建的线程中访问。解决这个问题的方法是用 Invoke 来执行委托，间接调用 lnklblMsg 控件。

首先创建委托和访问 LinkLabel 控件的方法：

```
    //创建委托，用于访问主线程的控件
     public delegate void delegateNewMsg(int msgs);
     void dgtNewMsgMethod(int intMsgs)
     {
         if (intMsgs > 0)
         {
             this.lnklblNewWarn.Text = "有" + intMsgs.ToString() + "条新告警信息";
         }
         else
         {
             this.lnklblNewWarn.Text = "";
         }
     }
```

然后再在 Timer 的 Elapsed 方法中执行委托：

```
      delegateNewMsg d = dgtNewMsgMethod;
```

```
        Invoke(d,intMsgs);//访问主线程资源
```

C#中在线程中访问主 Form 控件的问题

C#不允许直接从线程中访问 Form 里的控件，比如希望在线程里修改 Form 里的一个 TextBox 的内容等等，唯一的做法是使用 Invoke 方法，下面是一个 MSDN 里的 Example，很说明问题：

```
using System;
using System.Drawing;
using System.Windows.Forms;
using System.Threading;

    public class MyFormControl : Form
    ...{
        public delegate void AddListItem(String myString);
        public AddListItem myDelegate;
        private Button myButton;
        private Thread myThread;
        private ListBox myListBox;
        public MyFormControl()
        ...{
            myButton = new Button();
            myListBox = new ListBox();
            myButton.Location = new Point(72, 160);
            myButton.Size = new Size(152, 32);
            myButton.TabIndex = 1;
            myButton.Text = "Add items in list box";
            myButton.Click += new EventHandler(Button_Click);
            myListBox.Location = new Point(48, 32);
            myListBox.Name = "myListBox";
            myListBox.Size = new Size(200, 95);
            myListBox.TabIndex = 2;
            ClientSize = new Size(292, 273);
            Controls.AddRange(new Control[] ...{myListBox,myButton});
            Text = " 'Control_Invoke' example ";
            myDelegate = new AddListItem(AddListItemMethod);
        }
        static void Main()
        ...{
            MyFormControl myForm = new MyFormControl();
            myForm.ShowDialog();
        }
        public void AddListItemMethod(String myString)
```

```
    ...{
            myListBox.Items.Add(myString);
    }
    private void Button_Click(object sender, EventArgs e)
    ...{
        myThread = new Thread(new ThreadStart(ThreadFunction));
        myThread.Start();
    }
    private void ThreadFunction()
    ...{
        MyThreadClass myThreadClassObject   = new MyThreadClass(this);
        myThreadClassObject.Run();
    }
}
public class MyThreadClass
...{
    MyFormControl myFormControl1;
    public MyThreadClass(MyFormControl myForm)
    ...{
        myFormControl1 = myForm;
    }
    String myString;

    public void Run()
    ...{



        for (int i = 1; i <= 5; i++)
        ...{
            myString = "Step number " + i.ToString() + " executed";
            Thread.Sleep(400);
            // Execute the specified delegate on the thread that owns
            // 'myFormControl1' control's underlying window handle with
            // the specified list of arguments.
            myFormControl1.Invoke(myFormControl1.myDelegate,
                                  new Object[] ...{myString});
        }
    }
}
```

如何在子线程中操作窗体上的控件 - 愚翁专栏 - CSDNBlog 正在处理您的请求...

如何在子线程中操作窗体上的控件

一般来说，直接在子线程中对窗体上的控件操作是会出现异常，这是由于子线程和运行窗体的线程是不同的空间，因此想要在子线程来操作窗体上的控件，是不可能简单的通过控件对象名来操作，但不是说不能进行操作，微软提供了 Invoke 的方法，其作用就是让子线程告诉窗体线程来完成相应的控件操作。

现在用一个用线程控制的进程条来说明，大致的步骤如下：

1. 创建 Invoke 函数，大致如下：

```
/// <summary>
/// Delegate function to be invoked by main thread
/// </summary>
private void InvokeFun()
{
    if( prgBar.Value < 100 )
        prgBar.Value = prgBar.Value + 1;
}
```

2. 子线程入口函数：

```
/// <summary>
/// Thread function interface
/// </summary>
private void ThreadFun()
{
    //Create invoke method by specific function
    MethodInvoker mi = new MethodInvoker( this.InvokeFun );

    for( int i = 0; i < 100; i++ )
    {
        this.BeginInvoke( mi );
        Thread.Sleep( 100 );
    }
}
```

3. 创建子线程：

```
Thread thdProcess = new Thread( new ThreadStart( ThreadFun ) );
thdProcess.Start();
```

备注：

```
using System.Threading;
private System.Windows.Forms.ProgressBar prgBar;
```

运行后的效果如下图所示：

Trackback: http://tb.blog.csdn.net/TrackBack.aspx?PostId=626584

[收藏到我的网摘]　Knight94 发表于 2006 年 03 月 16 日 19:25:00

<<如何在 MDI 程序中把子窗体菜单合并到主窗体上 |

knight94 发表于 2006-04-21 09:21:00 IP: 218.71.239.*

首先，你在创建这个类的对象时，要把当前的窗体对象传进去（为了到时候能通过它调用其的方法）。

然后把

Thread thdProcess = new Thread(new ThreadStart(ThreadFun));

改成

Thread thdProcess = new Thread(new ThreadStart(yourObj.ThreadFun));

即可。

User 发表于 2006-04-29 16:03:00 IP: 219.238.164.*

我想实现批量注册用户的功能，首先在文本域里录入用户信息（按照约定的规则），然后执行。

现在我希望在界面上有一个文本域滚动提示每个用户导入是否成功，失败的原因，另外可以有一个滚动条显示进度。

麻烦 knight94 ，指教。

knight94 发表于 2006-05-01 13:17:00 IP: 218.71.239.*

to User

对于你的问题，参看我的例子就可以了，我的例子是如何滚动进度条，那么你可以在上面的基础上进行扩展，而且 Invoke 调用方法，也可以加参数，例如：

```
-------in your form class ------
public void ShowText( string sData )
{
//Handle "sData" here
}
--------in thread fun ----------
MethodInvoker mi = new MethodInvoker( this.ShowText );
this.BeginInvoke( mi, new object[]{ yourText } );
```

aicsharp 发表于 2006-05-26 17:25:00 IP: 60.216.137.*

不错，msdn 的解释太不中国了，光看那个根本不知道干什么的。他是这么解释的

“MethodInvoker 提供一个简单委托，该委托用于调用含 void 参数列表的方法。在对控件的 Invoke

方法进行调用时或需要一个简单委托又不想自己定义时可以使用该委托。

”

这是 msdn for vs2005 的解释。

DisonWorld 发表于 2006-05-30 13:51:00 IP: 61.186.185.*

I have a question as the following:

```
foreach(int i in ...)
{
//the order is important, that is:
//DoSomthing(2) must be done after the DoSomething(1) is completed

DoSomething(i);
}

private void DoSomething(int i)
{
//In here, a function to upload file will be called,
//and it will be implemeted by using class Upload, who is inherited from
BackgroundWorker and will call a WSE
// to upload the file, so the speed is a bit slow
}
```

So my question is how to make the DoSomthing(2) only begins when the DoSomthing(1) is finished.

knight94 发表于 2006-05-30 14:04:00 IP: 218.71.239.*

to DisonWorld

you can reprace "BeginInvoke" method with "Invoke" method.

See difference between them in:
http://blog.csdn.net/Knight94/archive/2006/05/27/757351.aspx

DisonWorld 发表于 2006-05-31 12:00:00 IP: 61.186.185.*

Hello Knight94,

I have tried the Invoke, but the ui seem to be dead, and the import order seems to wrong.
For example:
PackageA: no files to import
PackageB: need to import files
PackageC: no files to import
After the PackageA is finished, and the PackageB will start and begin import file, but even the PackageB has not been finished, the PackageC will start.

```
private delegate bool ImportSinglePackage();
private ImportSinglePackage importSinglePackage = null;
```

```
//The UI will call the BeginImport to start the importing
public BeginImport()
{
Cursor.Current = Cursors.WaitCursor;
this.importSinglePackage = new
ImportSinglePackage(this.ImportBySinglePackage);

mImportingDataThread = new Thread(new
ThreadStart(this.ImportPackages));
mImportingDataThread.Start();
}

private void ImportPackages()
{
bool IsAllPackagesHaveBeenImported = false;
do{
this.currentImportRecordKey =
recordKeysToImport[recordsHasBeenImported];
Thread.Sleep(1000);
IsAllPackagesHaveBeenImported =
(bool)this.Invoke(this.importSinglePackage, null);
}while (!IsAllPackagesHaveBeenImported);
}

private bool ImportBySinglePackage()
{
//ImportFiles will call the a class inherited from the BackgroundWorker to
download/upload file

Thread threadImportFiles = new Thread(new ThreadStart(this.ImportFiles));
threadImportFiles.Start();
threadImportFiles.Join();
```

knight94 发表于 2006-05-31 13:09:00 IP: 218.71.239.*

As the upper code, there is no return value in your function named "ImportBySinglePackage".

By the way, you should check the invoke's return value in your "ImportPackages" function.

tianjj 发表于 2006-08-17 17:10:00 IP: 203.86.72.*

to:knight94

--------in thread fun ----------

MethodInvoker mi = new MethodInvoker(this.ShowText);
这个调用带有参数，好像有误。
提示 showText 重载与 MethodInvoker 不匹配。

tianjj 发表于 2006-08-17 17:12:00 IP: 203.86.72.*
to:knight94

--------in thread fun ----------
MethodInvoker mi = new MethodInvoker(this.ShowText);
这个调用带有参数，好像有误。
提示 showText 重载与 MethodInvoker 不匹配。

tianjj 发表于 2006-08-17 17:27:00 IP: 203.86.72.*
to:knight94
MethodInvoker mi = new MethodInvoker(this.ShowText);

这个调用带参数方法，好像有误，
显示 showText 重载与 MethodInvoker 不匹配。

knight94 发表于 2006-08-17 18:07:00 IP: 218.71.239.*
to tianjj

参看：
http://blog.csdn.net/knight94/archive/2006/05/27/757351.aspx

knight94 发表于 2006-08-17 18:10:00 IP: 218.71.239.*
to tianjj

MethodInvoker 是一个 delegate 类型，你可以自己定义，例如：
public delegate int MyInvoker(string strValue);

MyInvoder mi = new MyInvoker(yourMethod);
this.Invoke(mi, new object[]{ "test" });

菜鸟 发表于 2006-08-23 16:29:00 IP: 219.140.184.*
进度条为什么不用 TIMER 控件来做？

knight94 发表于 2006-08-23 17:48:00 IP: 218.71.239.*
Timer 的交互性太差。

大漠狂沙 发表于 2006-11-02 14:00:00 IP: 219.142.3.*
工作线程类：
public class CommissionHelper
{

```
private System.Windows.Forms.ProgressBar proBar;
public CommissionHelper(System.Windows.Forms.ProgressBar proBar)
{
this.proBar=proBar;
}
public void Run()
{
int i=1;
for()
{
proBar.Value=i;
i=i+1;
}
}
}
```

主线程窗体按钮调用：

```
private void btnTransfer_Click(object sender, System.EventArgs e)
{
CommissionHelper helper=new CommissionHelper(proBar);

Thread tdh=new Thread( new ThreadStart( helper.Run ) );

tdh.Start();
}
```

这样也可以实现。我没明白为什么要用 Invoke？！
谢谢！

knight94 发表于 2006-11-02 14:49:00 IP: 218.71.239.*

to 大漠狂沙

问题是很对情况下，在子线程中不能直接操纵 UI 线程上的控件，这时候就需要用 Invoke 或者 BeginInvoke 来完成。

也就是说现在有些程序是可以跨线程互相操作，有些程序是不允许的。
前者，按照你所说的没问题；后者按照你所写的，则是有问题的。

大漠狂沙 发表于 2006-11-03 11:16:00 IP: 219.142.3.*

嗯，明白了，谢谢你的回复，学到很多东西！

小鲁 发表于 2006-11-04 14:16:00 IP: 58.61.133.*

愚翁老师,非常感谢,我钻研了阻塞线程中打开新窗体,窗体定死的问题,一天一夜都没有答案,你的文章个给了我启发,现在解决了非常感谢.

wc_king 发表于 2006-11-06 00:44:00 IP: 222.90.59.*

knight94 发表于 2006-11-02 14:49:00 IP: 218.71.239.*

to 大漠狂沙

问题是很对情况下，在子线程中不能直接操纵 UI 线程上的控件，这时候就需要用 Invoke 或者 BeginInvoke 来完成。

也就是说现在有些程序是可以跨线程互相操作，有些程序是不允许的。

前者，按照你所说的没问题；后者按照你所写的，则是有问题的。

————————————————————————————————

请问，能举个例子说明什么时候会出现跨线程互相操作有问题？那么应该根据什么标准来判断会出现这种问题？

knight94 发表于 2006-11-06 17:56:00 IP: 218.71.239.*

to wc_king

对于你所说的，可以通过 Control.InvokeRequired 来进行判断是否需要 Invoke 或者 BeginInvoke 来操作。

老实和尚 发表于 2006-11-24 14:25:00 IP: 211.162.77.*

楼主：

我新学 C#，以前一直在 unix 下面混的，现在我根据您的介绍写一个界面，需求是：当我按了“begin”button 以后，listview 上不停的打印东西出来，按了“stop”button 以后，停止打印。源代码如下所示：

```
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;

using System.Threading;
namespace WindowsApplication1
{
public partial class Form1 : Form
{
private volatile bool flag = false;
private Thread th1 = null;
public Form1()
{
InitializeComponent();
}
```

```
private void button1_Click(object sender, EventArgs e)
{
flag = true;
th1 = new Thread(new ThreadStart(th_fun));
th1.Start();

}
private void fun()
{
while (flag)
{
listView1.Items.Add("ssss");
listView1.Refresh();
Thread.Sleep(1000);
}
Console.WriteLine("stopppppp");
}

private void th_fun()
{
MethodInvoker me = new MethodInvoker(this.fun);
this.BeginInvoke(me);
}
private void button2_Click(object sender, EventArgs e)
{
Console.WriteLine("stop clicked");
```

knight94 发表于 2006-11-24 16:58:00 IP: 218.71.239.*

to 老实和尚

你的循环放错位置了，你这样写，也就是线程只执行一次函数，而这个函数永远在执行。

你如下去修改

```
private void fun()
{
listView1.Items.Add("ssss");
listView1.Refresh();
}

private void th_fun()
{
MethodInvoker me = new MethodInvoker(this.fun);
while (flag)
```

```
{
this.BeginInvoke(me);
Thread.Sleep(1000);
}
Console.WriteLine("stopppppp");
}
```

老实和尚 发表于 2006-11-24 17:37:00 IP: 211.162.77.*

谢谢楼主。

thx～～～～

atealxt 发表于 2006-12-12 11:31:06 IP: 221.216.0.*

谢谢你的几篇 Thread 的文章让我解决了和上面"小鲁"一样的问题

dingsea 发表于 2007-01-09 16:50:37 IP: 219.74.131.*

如果把 InvokeFun()的实现改为

label1.Text+="1";

这样一句的话,在程序没有运行完之前退出会有异常:

Cannot call Invoke or InvokeAsync on a control until the window handle has been created.

superwat 发表于 2007-01-19 16:01:17 IP: 218.104.7.*

老大,我郁闷了,救救我啊.

```
using System.Threading;
private Boolean bTasking = false;
private Thread tr_Task;

public void UF_Task(Boolean bstart)
{
bTasking = bstart;
if (bstart) sBarP2.Value = 0;
if (tr_Task == null)
{
tr_Task = new Thread(new ThreadStart(UF_DoProcess));
tr_Task.IsBackground = true;
tr_Task.Name = "QBSProcess";
tr_Task.Start();
}
if (!bstart) sBarP2.Value = 100;
}

private void UF_DoProcess()
```

```
{
try
{
MethodInvoker mip = new MethodInvoker(this.UF_ProcStep);
while (bTasking)
{
this.BeginInvoke(mip);
Thread.Sleep(100);
}
}
catch { }
}

private void UF_ProcStep()
{
if (!bTasking) return;
if (sBarP2.Value == 100) sBarP2.Value = 0;
else sBarP2.PerformStep();
}
```

假设我在调用一个任务 A 时 ,大约需要 1 秒,调用
UF_Task(true);
A 任务;
UF_Task(false);

可是进度条没有中间状态啊,进度条只有 0 和 100....
怎么回事啊???
在线等你,老大.

Knight94 发表于 2007-01-20 11:40:05 IP: 210.77.27.*

to superwat

你出现的问题是由于用 Invoke 是提交给 UI 所在线程来完成,而 UI 线程此时忙于处理 A 任务，因此得不到及时响应。

因此你可以把这部分
UF_Task(true);
A 任务;
UF_Task(false);
放到一个单独的线程去处理，这样就避免了 UI 线程被占用。

superwat 发表于 2007-01-22 16:33:00 IP: 218.104.7.*

:)
谢谢老大,我也明白其中的道理了.现在想到另外一个办法.我试试效果，行的通的话,回来

献丑,呵呵

sunrobust 发表于 2007-01-26 11:50:23 IP: 221.122.45.*

愚翁,您好!这篇文章使我收益匪浅,致谢!

我是新手,有几个问题向你咨询.

1. 如果我将子线程单独写在一个类定义里面,应如何安排 InvokeFun() 和 ThreadFun()两个函数?

即谁在子线程定义中,谁应该写在主窗体定义中?

2. 是不是在子线程中仍然要引入 System.Windows.Forms 命名控件?

sunrobust 发表于 2007-01-26 11:51:31 IP: 221.122.45.*

3. 是否要修改 ProcessBar 的属性为 Public?

Knight94 发表于 2007-01-31 10:13:11 IP: 210.77.27.*

to sunrobust

不一定，既然是通过委托来操作，没必要把 processbar 属性设为 public，只需要给出相应的修改方法就行。

你可以参看这篇文章

http://blog.csdn.net/Knight94/archive/2006/08/24/1111267.aspx

委托

[转]如何智能客户端应用程序性能

智能客户端应用程序性能

发布日期： 08/20/2004 | 更新日期： 08/20/2004

patterns & practices
proven practices for predictable results

智能客户端体系结构与设计指南

David Hill、Brenton Webster、Edward A. Jezierski、Srinath Vasireddy、Mohammad Al-Sabt，Microsoft Corporation，Blaine Wastell Ascentium Corporation，Jonathan Rasmusson 和 Paul Gale ThoughtWorks 和 Paul Slater Wadeware LLC

相关链接

Microsoft® *patterns & practices* 库 http://www.microsoft.com/resources/practices/default.mspx

.NET 的应用程序体系结构：设计应用程序和服务 http://msdn.microsoft.com/library/en-us/dnbda/html/distapp.asp

摘要：本章讨论如何优化您的智能客户端应用程序。本章分析您可以在设计时采取的步骤，并介绍如何调整智能客户端应用程序以及诊断任何性能问题。

本页内容

针对性能进行设计

性能调整和诊断

小结

参考资料

智能客户端应用程序可以提供比 Web 应用程序更丰富和响应速度更快的用户界面，并且可以利用本地系统资源。如果应用程序的大部分驻留在用户的计算机上，则应用程序不需要到 Web 服务器的持续的往返行程。这有利于提高性能和响应性。然而，要实现智能客户端应用程序的全部潜能，您应该在应用程序的设计阶段仔细考虑性能问题。通过在规划和设计您的应用程序时解决性能问题，可以帮助您及早控制成本，并减小以后陷入性能问题的可能性。

注改善智能客户端应用程序的性能并不仅限于应用程序设计问题。您可以在整个应用程序生存期中采取许多个步骤来使 .NET 代码具有更高的性能。虽然 .NET 公共语言运行库 (CLR) 在执行代码方面非常有效，但您可以使用多种技术来提高代码的性能，并防止在代码级引入性能问题。有关这些问题的详细信息，请参阅 http://msdn.microsoft.com/perf。

在应用程序的设计中定义现实的性能要求并识别潜在的问题显然是重要的，但是性能问题通常只在编写代码之后对其进行测试时出现。在这种情况下，您可以使用一些工具和技术来跟踪性能问题。

本章分析如何设计和调整您的智能客户端应用程序以获得最佳性能。它讨论了许多设计和体系结构问题（包括线程处理和缓存注意事项），并且分析了如何增强应用程序的 Windows 窗体部分的性能。本章还介绍了您可以用来跟踪和诊断智能客户端应用程序性能问题的一些技术和工具。

针对性能进行设计

您可以在应用程序设计或体系结构级完成许多工作，以确保智能客户端应用程序具有良好的性能。您应该确保在设计阶段尽可能早地制定现实的且可度量的性能目标，以便评估设计折衷，并且提供最划算的方法来解决性能问题。只要可能，性能目标就应该基于实际的用户和业务要求，因为这些要求受到应用程序所处的操作环境的强烈影响。性能建模是一种结构化的且可重复的过程，您可以使用该过程来管理您的应用程序并确保其实现性能目标。有关详细信息，请参阅 *Improving .NET Application Performance and Scalability* 中的第 2 章"Performance Modeling"，网址为：http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnpag/html/scalenetchapt02.asp。

智能客户端通常是较大的分布式应用程序的组成部分。很重要的一点是在完整应用程序的上下文中考虑智能客户端应用程序的性能，包括该客户端应用程序使用的所有位于网络中的资源。微调并优化应用程序中的每一个组件通常是不必要或不可能的。相反，性能调整应该基于优先级、时间、预算约束和风险。一味地追求高性能通常并不是一种划算的策略。

智能客户端还将需要与用户计算机上的其他应用程序共存。当您设计智能客户端应用程序时，您应该考虑到您的应用程序将需要与客户端计算机上的其他应用程序共享系统资源，例如，内存、CPU 时间和网络利用率。

注有关设计可伸缩的高性能远程服务的信息，请参阅*Improving .NET Performance and Scalability*，网址为：http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnpag/html/scalenet.asp。本指南包含有关如何优化 .NET 代码以获得最佳性能的详细信息。

要设计高性能的智能客户端，请考虑下列事项：

- **在适当的位置缓存数据**。数据缓存可以显著改善智能客户端应用程序的性能，使您可以在本地使用数据，而不必经常从网络检索数据。但是，敏感数据或频繁更改的数据通常不适合进行缓存。
- **优化网络通讯**。如果通过"健谈的"接口与远程层服务进行通讯，并且借助于多个请求/响应往返行程来执行单个逻辑操作，则可能消耗系统和网络资源，从而导致低劣的应用程序性能。
- **有效地使用线程**。如果您使用用户界面 (UI) 线程执行阻塞 I/O 绑定调用，则 UI 似乎不对用户作出响

应。因为创建和关闭线程需要系统开销，所以创建大量不必要的线程可能导致低劣的性能。

- **有效地使用事务**。如果客户端具有本地数据，则使用原子事务可帮助您确保该数据是一致的。因为数据是本地的，所以事务也是本地的而不是分布式的。对于脱机工作的智能客户端而言，对本地数据进行的任何更改都是暂时的。客户端在重新联机时需要同步更改。对于非本地数据而言，在某些情况下可以使用分布式事务（例如，当服务位于具有良好连接性的同一物理位置并且服务支持它时）。诸如 Web 服务和消息队列之类的服务不支持分布式事务。
- **优化应用程序启动时间**。较短的应用程序启动时间使用户可以更为迅速地开始与应用程序交互，从而使用户立刻对应用程序的性能和可用性产生好感。应该对您的应用程序进行适当的设计，以便在应用程序启动时仅加载那些必需的程序集。因为加载每个程序集都会引起性能开销，所以请避免使用大量程序集。
- **有效地管理可用资源**。低劣的设计决策（例如，实现不必要的完成器，未能在 **Dispose** 方法中取消终止，或者未能释放非托管资源）可能导致在回收资源时发生不必要的延迟，并且可能造成使应用程序性能降低的资源泄漏。如果应用程序未能正确地释放资源，或者应用程序显式强制进行垃圾回收，则可能会妨碍 CLR 有效地管理内存。
- **优化 Windows 窗体性能**。智能客户端应用程序依靠 Windows 窗体来提供内容丰富且响应迅速的用户界面.您可以使用多种技术来确保 Windows 窗体提供最佳性能。这些技术包括降低用户界面的复杂性，以及避免同时加载大量数据。

在许多情况下，从用户角度感受到的应用程序性能起码与应用程序的实际性能同样重要。您可以通过对设计进行某些特定的更改来创建在用户看来性能高得多的应用程序，例如：使用后台异步处理（以使 UI 能作出响应）；显示进度栏以指示任务的进度；提供相应的选项以便用户取消长期运行的任务。

本节将专门详细讨论这些问题。

数据缓存原则

缓存是一种能够改善应用程序性能并提供响应迅速的用户界面的重要技术。您应该考虑下列选项：

- **缓存频繁检索的数据以减少往返行程**。如果您的应用程序必须频繁地与网络服务交互以检索数据，则应该考虑在客户端缓存数据，从而减少通过网络重复获取数据的需要。这可以极大地提高性能，提供对数据的

近乎即时的访问，并且消除了可能对智能客户端应用程序性能造成不利影响的网络延迟和中断风险。

- **缓存只读引用数据**。只读引用数据通常是理想的缓存对象。此类数据用于提供进行验证和用户界面显示所需的数据，例如，产品说明、ID 等等。因为客户端无法更改此类数据，所以通常可以在客户端缓存它而无须进行任何进一步的特殊处理。
- **缓存要发送给位于网络上的服务的数据**。您应该考虑缓存要发送给位于网络上的服务的数据。例如，如果您的应用程序允许用户输入由在多个窗体中收集的一些离散数据项组成的定单信息，则请考虑允许用户输入全部数据，然后在输入过程的结尾在一个网络调用中发送定单信息。
- **尽量少地缓存高度不稳定的数据**。在缓存任何不稳定的数据之前，您需要考虑在其变得陈旧或者由于其他原因变得不可用之前，能够将其缓存多长时间。如果数据高度不稳定并且您的应用程序依赖于最新信息，则或许只能将数据缓存很短一段时间（如果可以缓存）。
- **尽量少地缓存敏感数据**。您应该避免在客户端上缓存敏感数据，因为在大多数情况下，您无法保证客户端的物理安全。但是，如果您必须在客户端上缓存敏感数据，则您通常将需要加密数据，该操作本身也会影响性能。

有关数据缓存的其他问题的详细信息，请参阅本指南的第 2 章。另请参阅 *Improving .NET Application Performance and Scalability* 的第 3 章"Design Guidelines for Application Performance"(http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnpag/html/scalenetchapt03.asp) 的"Caching"一节以及 *Improving .NET Application Performance and Scalability* 的第 4 章"Architecture and Design Review of .NET Application for Performance and Scalability"(http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnpag/html/scalenetchapt04.asp)。

网络通讯原则

您将面临的另一个决策是如何设计和使用网络服务，例如，Web 服务。特别地，您应该考虑与网络服务交互的粒度、同步性和频率。要获得最佳的性能和可伸缩性，您应该在单个调用中发送更多的数据，而不是在多个调用中发送较少量的数据。例如，如果您的应用程序允许用户在定单中输入多个项，则较好的做法是为所有项收集数据，然后将完成的采购定单一次性发送给服务，而不是在多个调用中发送单个项的详细信息。除了降低与进行大量网络调用相关联的系统开销以外，这还可以减少服务和/或客户端内的复杂状态管理的需要。

应该将您的智能客户端应用程序设计为尽可能地使用异步通讯，因为这将有助于使用户界面快速响应以及并行执行任务。有关如何使用 **BeginInvoke** 和 **EndInvoke** 方法异步启动调用和检索数据的详细信息，请参阅"Asynchronous Programming Overview"(http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpovrasynchronousprogrammingoverview.asp)。

注有关设计和构建偶尔连接到网络的智能客户端应用程序的详细信息，请参阅 第 3 章"建立连接"和 第 4 章"偶尔连接的智能客户端"。

线程处理原则

在应用程序内使用多个线程可能是一种提高其响应性和性能的好方法。特别地，您应该考虑使用线程来执行可以在后台安全地完成且不需要用户交互的处理。通过在后台执行此类工作，可以使用户能够继续使用应用程序，并且使应用程序的主用户界面线程能够维持应用程序的响应性。

适合于在单独的线程上完成的处理包括：

- **应用程序初始化**。请在后台线程上执行漫长的初始化，以便用户能够尽快地与您的应用程序交互，尤其是在应用程序功能的重要或主要部分并不依赖于该初始化完成时。
- **远程服务调用**。请在单独的后台线程上通过网络进行所有远程调用。很难（如果不是无法）保证位于网络上的服务的响应时间。在单独的线程上执行这些调用可以减少发生网络中断或延迟的风险，从而避免对应用程序性能造成不利影响。
- **IO 绑定处理**。应该在单独的线程上完成诸如在磁盘上搜索和排序数据之类的处理。通常，这种工作要受到磁盘 I/O 子系统而不是处理器可用性的限制，因此当该工作在后台执行时，您的应用程序可以有效地维持其响应性。

尽管使用多个线程的性能好处可能很显著，但需要注意，线程使用它们自己的资源，并且使用太多的线程可能给处理器（它需要管理线程之间的上下文切换）造成负担。要避免这一点，请考虑使用线程池，而不是创建和管理您自己的线程。线程池将为您有效地管理线程，重新使用现有的线程对象，并且尽可能地减小与线程创建和处置相关联的系统开销。

如果用户体验受到后台线程所执行的工作的影响，则您应该总是让用户了解工作的进度。以这种方式提供反馈可以增强用户对您的应用程序的性能的感觉，并且防止他或她假设没有任何事情发生。请努力确保用户可以随时取消漫长的操作。

您还应该考虑使用 **Application** 对象的 **Idle** 事件来执行简单的操作。**Idle** 事件提供了使用单独的线程来进行后台处理的简单替代方案。当应用程序不再有其他用户界面消息需要处理并且将要进入空闲状态时，该事件将激发。您可以通过该事件执行简单的操作，并且利用用户不活动的情况。例如：

[C#]

```
public Form1()
{
InitializeComponent();
Application.Idle += new EventHandler( OnApplicationIdle );
}
private void OnApplicationIdle( object sender, EventArgs e )
{
}
```

[Visual Basic .NET]

```
Public Class Form1
Inherits System.Windows.Forms.Form
Public Sub New()
MyBase.New()
InitializeComponent()
AddHandler Application.Idle, AddressOf OnApplicationIdle
End Sub
```

```
Private Sub OnApplicationIdle(ByVal sender As System.Object, ByVal e
As System.EventArgs)
End Sub
End Class
```

注有关在智能客户端中使用多个线程的详细信息，请参阅 第 6 章"使用多个线程"。

事务原则

事务可以提供重要的支持，以确保不会违反业务规则并维护数据一致性。事务可以确保一组相关任务作为一个单元成功或失败。您可以使用事务来维护本地数据库和其他资源（包括消息队列的队列）之间的一致性。

对于需要在网络连接不可用时使用脱机缓存数据的智能客户端应用程序，您应该将事务性数据排队，并且在网络连接可用时将其与服务器进行同步。

您应该避免使用涉及到位于网络上的资源的分布式事务，因为这些情况可能导致与不断变化的网络和资源响应时间有关的性能问题。如果您的应用程序需要在事务中涉及到位于网络上的资源，则应该考虑使用补偿事务，以便使您的应用程序能够在本地事务失败时取消以前的请求。尽管补偿事务在某些情况下可能不适用，但它们使您的应用程序能够按照松耦合方式在事务的上下文内与网络资源交互，从而减少了不在本地计算机控制之下的资源对应用程序的性能造成不利影响的可能性。

注有关在智能客户端中使用事务的详细信息，请参阅 第 3 章"建立连接"。

优化应用程序启动时间

快速的应用程序启动时间几乎可以使用户立即开始与应用程序交互，从而使用户立刻对应用程序的性能和可用性产生好感。

当应用程序启动时，首先加载 CLR，再加载应用程序的主程序集，随后加载为解析从应用程序的主窗体中引用的对象的类型所需要的所有程序集。CLR 在该阶段*不会* 加载所有相关程序集；

它仅加载包含主窗体类上的成员变量的类型定义的程序集。在加载了这些程序集之后，实时 (JIT) 编译器将在方法运行时编译方法的代码（从 **Main** 方法开始）。同样，JIT 编译器*不会*编译您的程序集中的所有代码。相反，将根据需要逐个方法地编译代码。

要尽可能减少应用程序的启动时间，您应该遵循下列原则：

- **尽可能减少应用程序主窗体类中的成员变量**。这将在 CLR 加载主窗体类时尽可能减少必须解析的类型数量。
- **尽量不要立即使用大型基类程序集（XML 库或 ADO.NET 库）中的类型**。这些程序集的加载很费时间。使用应用程序配置类和跟踪开关功能时将引入 XML 库。如果要优先考虑应用程序启动时间，请避免这一点。
- **尽可能使用惰性加载**。仅在需要时获取数据，而不是提前加载和冻结 UI。
- **将应用程序设计为使用较少的程序集**。带有大量程序集的应用程序会招致性能开销增加。这些开销来自加载元数据、访问 CLR 中的预编译映像中的各种内存页以加载程序集（如果它是用本机映像生成器工具 Ngen.exe 预编译的）、JIT 编译时间、安全检查等等。您应该考虑基于程序集的使用模式来合并程序集，以便降低相关联的性能开销。
- **避免设计将多个组件的功能组合到一个组件中的单一类**。将设计分解到多个只须在实际调用时进行编译的较小类。
- **将应用程序设计为在初始化期间对网络服务进行并行调用**。通过在初始化期间调用可以并行运行的网络服务，可以利用服务代理提供的异步功能。这有助于释放当前执行的线程并且并发地调用服务以完成任务。

- **使用 NGEN.exe 编译和试验 NGen 和非 NGen 程序集，并且确定哪个程序集保存了最大数量的工作集页面**。NGEN.exe（它随附在 .NET Framework 中）用于预编译程序集以创建本机映像，该映像随后被存储在全局程序集缓存的特殊部分，以便应用程序下次需要它时使用。通过创建程序集的本机映像，可以使程序集更快地加载和执行，因为 CLR 不需要动态生成程序集中包含的代码和数据结构。有关详细信息，请参阅 *Improving .NET Application Performance and Scalability* 的第 5 章“Improving Managed Code Performance”中的“Working Set Considerations”和“NGen.exe Explained”部分，网址为：http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnpag/html/scalenetchapt05.asp。

注如果您使用 NGEN 预编译程序集，则会立即加载它的所有依赖程序集。

管理可用资源

公共语言运行库 (CLR) 使用垃圾回收器来管理对象生存期和内存使用。这意味着无法再访问的对象将被垃圾回收器自动回收，并且自动回收内存。由于多种原因无法再访问对象。例如，可能没有对该对象的任何引用，或者对该对象的所有引用可能来自其他可作为当前回收周期的一部分进行回收的对象。尽管自动垃圾回收使您的代码不必负责管理对象删除，但这意味着您的代码不再对对象的确切删除时间具有显式控制。

请考虑下列原则，以确保您能够有效地管理可用资源：

- **确保在被调用方对象提供 Dispose 方法时该方法得到调用**。如果您的代码调用了支持 **Dispose** 方法的对象，则您应该确保在使用完该对象之后立即调用此方法。调用 **Dispose** 方法可以确保抢先释放非托管资源，而不是等到发生垃圾回收。除了提供 **Dispose** 方法以外，某些对象还提供其他管理资源的方法，例如，**Close** 方法。在这些情况下，您应该参考文档资料以了解如何使用其他方法。例如，对于 **SqlConnection** 对象而言，调用 **Close** 或 **Dispose** 都足可以抢先将数据库连接释放回连接池中。一种可以确保您在对象使用完毕之后立即调用 **Dispose** 的方法是使用 Visual C# .NET 中的 **using** 语句或 Visual Basic .NET 中的 **Try/Finally** 块。

下面的代码片段演示了 **Dispose** 的用法。

C# 中的 **using** 语句示例：

```
using( StreamReader myFile = new StreamReader("C:\\ReadMe.Txt")){
            string contents = myFile.ReadToEnd();
            //... use the contents of the file
            } // dispose is called and the StreamReader's resources
released
```

Visual Basic .NET 中的 **Try/Finally** 块示例：

```
Dim myFile As StreamReader
            myFile = New StreamReader("C:\\ReadMe.Txt")
            Try
            String contents = myFile.ReadToEnd()
            '... use the contents of the file
            Finally
            myFile.Close()
            End Try
```

注在 C# 和 C++ 中，Finalize 方法是作为析构函数实现的。在 Visual Basic .NET 中，**Finalize** 方法是作为 **Object** 基类上的 **Finalize** 子例程的重写实现的。

- **如果您在客户端调用过程中占据非托管资源，则请提供 Finalize 和 Dispose 方法**。如果您在公共或受保护的方法调用中创建访问非托管资源的对象，则应用程序需要控制非托管资源的生存期。在图 8.1 中，第一种情况是对非托管资源的调用，在此将打开、获取和关闭资源。在此情况下，您的对象无须提供 **Finalize** 和 **Dispose** 方法。在第二种情况下，在方法调用过程中占据非托管资源；因此，您的对象应该提供 **Finalize** 和 **Dispose** 方法，以便客户端在使用完该对象后可以立即显式释放资源。

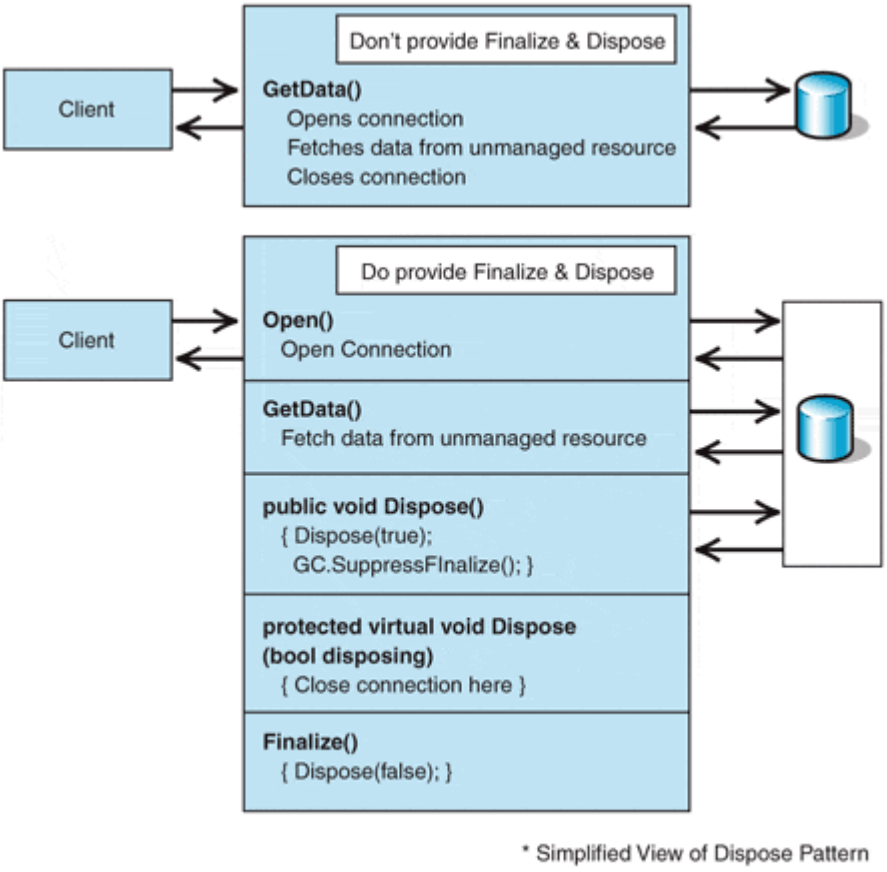


图 8.1：Dispose 和 Finalize 方法调用的用法

垃圾回收通常有利于提高总体性能，因为它将速度的重要性置于内存利用率之上。只有当内存资源不足时，才需要删除对象；否则，将使用所有可用的应用程序资源以使您的应用程序受益。但是，如果您的对象保持对非托管资源（例如，窗口句柄、文件、GDI 对象和网络连接）的引用，则程序员通过在这些资源不再使用时显式释放它们可以获得更好的性能。如果您要在客户端方法调用过程中占据非托管资源，则对象应该允许调用方使用 **IDisposable** 接口（它提供 **Dispose** 方法）显式管理资源。通过实现 **IDisposable**，对象将通知它可被要求明确进行清理，而不是等待垃圾回收。实现 **IDisposable** 的对象的调用方在使用完该对象后将简单地调用 **Dispose** 方法，以便它可以根据需要释放资源。

有关如何在某个对象上实现 **IDisposable** 的详细信息，请参阅 *Improving .NET Application Performance and Scalability* 中的第 5 章"Improving Managed Code Performance"，网址为：http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnpag/html/scalenetchapt05.asp。

注如果您的可处置对象派生自另一个也实现了 IDisposable 接口的对象，则您应该调用基类的 **Dispose** 方法以使其可以清理它的资源。您还应该调用实现了 **IDisposable** 接口的对象所拥有的所有对象上的 **Dispose**。

Finalize 方法也使您的对象可以在删除时显式释放其引用的任何资源。由于垃圾回收器所具有的非确定性，在某些情况下，**Finalize** 方法可能长时间不会被调用。实际上，如果您的应用程序在垃圾回收器删除对象之前终止，则该方法可能永远不会被调用。然而，需要使用 **Finalize** 方法作为一种后备策略，以防调用方没有显式调用 **Dispose** 方法（**Dispose** 和 **Finalize** 方法共享相同的资源清理代码）。通过这种方式，可能在某个时刻释放资源，即使这发生在最佳时刻之后。

注要确保 Dispose 和 Finalize 中的清理代码不会被调用两次，您应该调用 **GC.SuppressFinalize** 以通知垃圾回收器不要调用 **Finalize** 方法。

垃圾回收器实现了 **Collect** 方法，该方法强制垃圾回收器删除所有对象挂起删除。不应该从应用程序内调用该方法，因为回收周期在高优先级线程上运行。回收周期可能冻结所有 UI 线程，从而使得用户界面停止响应。

有关详细信息，请参阅 *Improving .NET Application Performance and Scalability* 中的"Garbage Collection Guidelines"、"Finalize and Dispose Guidelines"、"Dispose Pattern"和"Finalize and Dispose Guidelines"，网址为：http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnpag/html/scalenetchapt05.asp。

优化 Windows 窗体性能

Windows 窗体为智能客户端应用程序提供了内容丰富的用户界面，并且您可以使用许多种技术来帮助确保 Windows 窗体提供最佳性能。在讨论特定技术之前，对一些可以显著提高 Windows 窗体性能的高级原则进行回顾是有用的。

- **小心创建句柄**。Windows 窗体将句柄创建虚拟化（即，它动态创建和重新创建窗口句柄对象）。创建句柄对象的系统开销可能非常大；因此，请避免进行不必要的边框样式更改或者更改 MDI 父对象。
- **避免创建带有太多子控件的应用程序**。Microsoft? Windows? 操作系统限制每个进程最多有 10,000 个控件，但您应该避免在窗体上使用成百上千个控件，因为每个控件都要消耗内存资源。

本节的其余部分讨论您可以用来优化应用程序用户界面性能的更为具体的技术。

使用 BeginUpdate 和 EndUpdate

许多 Windows 窗体控件（例如，**ListView** 和 **TreeView** 控件）实现了 **BeginUpdate** 和 **EndUpdate** 方法，它们在操纵基础数据或控件属性时取消了控件的重新绘制。通过使用 **BeginUpdate** 和 **EndUpdate** 方法，您可以对控件进行重大更改，并且避免在应用这些更改时让控件经常重新绘制自身。此类重新绘制会导致性能显著降低，并且用户界面闪烁且不反应。

例如，如果您的应用程序具有一个要求添加大量节点项的树控件，则您应该调用 **BeginUpdate**，添加所有必需的节点项，然后调用 **EndUpdate**。下面的代码示例显示了一个树控件，该控件用于显示许多个客户的层次结构表示形式及其定单信息。

[C#]

```
// Suppress repainting the TreeView until all the objects have been c
reated.
TreeView1.BeginUpdate();
// Clear the TreeView.
TreeView1.Nodes.Clear();
// Add a root TreeNode for each Customer object in the ArrayList.
```

```
foreach( Customer customer2 in customerArray )
{
TreeView1.Nodes.Add( new TreeNode( customer2.CustomerName ) );
// Add a child TreeNode for each Order object in the current Custome
r.
foreach( Order order1 in customer2.CustomerOrders )
{
TreeView1.Nodes[ customerArray.IndexOf(customer2) ].Nodes.Add(
new TreeNode( customer2.CustomerName + "." + order1.OrderID ) );
}
}
// Begin repainting the TreeView.
TreeView1.EndUpdate();
```

[Visual Basic .NET]

```
        ' Suppress repainting the TreeView until all the objects have
been created.
TreeView1.BeginUpdate()
' Clear the TreeView
TreeView1.Nodes.Clear()
' Add a root TreeNode for each Customer object in the ArrayList
For Each customer2 As Customer In customerArray
TreeView1.Nodes.Add(New TreeNode(customer2.CustomerName))
' Add a child TreeNode for each Order object in the current Customer.
For Each order1 As Order In customer2.CustomerOrders
TreeView1.Nodes(Array.IndexOf(customerArray, customer2)).Nodes.Add( _
New TreeNode(customer2.CustomerName & "." & order1.OrderID))
Next
```

```
Next
' Begin repainting the TreeView.
TreeView1.EndUpdate()
```

即使在您不希望向控件添加许多对象时，您也应该使用 **BeginUpdate** 和 **EndUpdate** 方法。在大多数情况下，您在运行之前将不知道要添加的项的确切个数。因此，为了妥善处理大量数据以及应付将来的要求，您应该总是调用 **BeginUpdate** 和 **EndUpdate** 方法。

注调用 Windows 窗体控件使用的许多 **Collection** 类的 **AddRange** 方法时，将自动为您调用 **BeginUpdate** 和 **EndUpdate** 方法。

使用 SuspendLayout 和 ResumeLayout

许多 Windows 窗体控件（例如，**ListView** 和 **TreeView** 控件）都实现了 **SuspendLayout** 和 **ResumeLayout** 方法，它们能够防止控件在添加子控件时创建多个布局事件。

如果您的控件以编程方式添加和删除子控件或者执行动态布局，则您应该调用 **SuspendLayout** 和 **ResumeLayout** 方法。通过 **SuspendLayout** 方法，可以在控件上执行多个操作，而不必为每个更改执行布局。例如，如果您调整控件的大小并移动控件，则每个操作都将引发单独的布局事件。

这些方法按照与 **BeginUpdate** 和 **EndUpdate** 方法类似的方式操作，并且在性能和用户界面稳定性方面提供相同的好处。

下面的示例以编程方式向父窗体中添加按钮：

[C#]

```
private void AddButtons()
{
// Suspend the form layout and add two buttons.
this.SuspendLayout();
```

```
Button buttonOK = new Button();
buttonOK.Location = new Point(10, 10);
buttonOK.Size = new Size(75, 25);
buttonOK.Text = "OK";
Button buttonCancel = new Button();
buttonCancel.Location = new Point(90, 10);
buttonCancel.Size = new Size(75, 25);
buttonCancel.Text = "Cancel";
this.Controls.AddRange(new Control[]{buttonOK, buttonCancel});
this.ResumeLayout();
}
```

[Visual Basic .NET]

```
Private Sub AddButtons()
' Suspend the form layout and add two buttons
Me.SuspendLayout()
Dim buttonOK As New Button
buttonOK.Location = New Point(10, 10)
buttonOK.Size = New Size(75, 25)
buttonOK.Text = "OK"
Dim buttonCancel As New Button
buttonCancel.Location = New Point(90, 10)
buttonCancel.Size = New Size(75, 25)
buttonCancel.Text = "Cancel"
Me.Controls.AddRange(New Control() { buttonOK, buttonCancel } )
Me.ResumeLayout()
End Sub
```

每当您添加或删除控件、执行子控件的自动布局或者设置任何影响控件布局的属性（例如，大小、位置、定位点或停靠属性）时，您都应该使用 **SuspendLayout** 和 **ResumeLayout** 方法。

处理图像

如果您的应用程序显示大量图像文件（例如，.jpg 和 .gif 文件），则您可以通过以位图格式预先呈现图像来显著改善显示性能。

要使用该技术，请首先从文件中加载图像，然后使用 PARGB 格式将其呈现为位图。下面的代码示例从磁盘中加载文件，然后使用该类将图像呈现为预乘的、Alpha 混合 RGB 格式。例如：

[C#]

```
if ( image != null && image is Bitmap )
{
Bitmap bm = (Bitmap)image;
Bitmap newImage = new Bitmap( bm.Width, bm.Height,
System.Drawing.Imaging.PixelFormat.Format32bppPArgb );
using ( Graphics g = Graphics.FromImage( newImage ) )
{
g.DrawImage( bm, new Rectangle( 0,0, bm.Width, bm.Height ) );
}
image = newImage;
}
```

[Visual Basic .NET]

```
        If Not(image Is Nothing)  AndAlso (TypeOf image Is Bitmap) Th
en
Dim bm As Bitmap = CType(image, Bitmap)
Dim newImage As New Bitmap(bm.Width, bm.Height, _
```

```
System.Drawing.Imaging.PixelFormat.Format32bppPArgb)
Using g As Graphics = Graphics.FromImage(newImage)
g.DrawImage(bm, New Rectangle(0, 0, bm.Width, bm.Height))
End Using
image = newImage
End If
```

使用分页和惰性加载

在大多数情况下，您应该仅在需要时检索或显示数据。如果您的应用程序需要检索和显示大量信息，则您应该考虑将数据分解到多个页面中，并且一次显示一页数据。这可以使用户界面具有更高的性能，因为它无须显示大量数据。此外，这可以提高应用程序的可用性，因为用户不会同时面对大量数据，并且可以更加容易地导航以查找他或她需要的确切数据。

例如，如果您的应用程序显示来自大型产品目录的产品数据，则您可以按照字母顺序显示这些项，并且将所有以“A”开头的产品显示在一个页面上，将所有以“B”开头的产品显示在下一个页面上。然后，您可以让用户直接导航到适当的页面，以便他或她无须浏览所有页面就可以获得他或她需要的数据。

以这种方式将数据分页还使您可以根据需要获取后台的数据。例如，您可能只需要获取第一页信息以便显示并且让用户与其进行交互。然后，您可以获取后台中的、已经准备好供用户使用的下一页数据。该技术在与数据缓存技术结合使用时可能特别有效。

您还可以通过使用惰性加载技术来提高智能客户端应用程序的性能。您无须立即加载可能在将来某个时刻需要的数据或资源，而是可以根据需要加载它们。您可以在构建大型列表或树结构时使用惰性加载来提高用户界面的性能。在此情况下，您可以在用户需要看到数据时（例如，在用户展开树节点时）加载它。

优化显示速度

根据您用于显示用户界面控件和应用程序窗体的技术，您可以用多种不同的方式来优化应用程序的显示速度。

当您的应用程序启动时，您应该考虑尽可能地显示简单的用户界面。这将减少启动时间，并且向用户呈现整洁且易于使用的用户界面。而且，您应该努力避免引用类以及在启动时加载任何不会立刻需要的数据。这将减少应用程序和 .NET Framework 初始化时间，并且提高应用程序的显示速度。

当您需要显示对话框或窗体时，您应该在它们做好显示准备之前使其保持隐藏状态，以便减少需要的绘制工作量。这将有助于确保窗体仅在初始化之后显示。

如果您的应用程序具有的控件含有覆盖整个客户端表面区域的子控件，则您应该考虑将控件背景样式设置为不透明。这可以避免在发生每个绘制事件时重绘控件的背景。您可以通过使用 **SetStyle** 方法来设置控件的样式。使用 **ControlsStyles.Opaque** 枚举可以指定不透明控件样式。

您应该避免任何不必要的控件重新绘制操作。一种方法是在设置控件的属性时隐藏控件。在 **OnPaint** 事件中具有复杂绘图代码的应用程序能够只重绘窗体的无效区域，而不是绘制整个窗体。**OnPaint** 事件的 **PaintEventArgs** 参数包含一个 **ClipRect** 结构，它指示窗口的哪个部分无效。这可以减少用户等待查看完整显示的时间。

使用标准的绘图优化，例如，剪辑、双缓冲和 **ClipRectangle**。这还将通过防止对不可见或要求重绘的显示部分执行不必要的绘制操作，从而有助于改善智能客户端应用程序的显示性能。有关增强绘图性能的详细信息，请参阅 *Painting techniques using Windows Forms for the Microsoft .NET Framework*，网址为：http://windowsforms.net/articles/windowsformspainting.aspx。

如果您的显示包含动画或者经常更改某个显示元素，则您应该使用双缓冲或多缓冲，在绘制当前图像的过程中准备下一个图像。**System.Windows.Forms** 命名空间中的 **ControlStyles** 枚举适用于许多控件，并且 **DoubleBuffer** 成员可以帮助防止闪烁。启用 **DoubleBuff**

er 样式将使您的控件绘制在离屏缓冲中完成，然后同时绘制到屏幕上。尽管这有助于防止闪烁，但它的确为分配的缓冲区使用了更多内存。

返回页首

性能调整和诊断

在设计和实现阶段处理性能问题是实现应用程序性能目标的最划算的方法。但是，您只有在开发阶段经常且尽早测试应用程序的性能，才能真正有效地优化应用程序的性能。

尽管针对性能进行设计和测试都很重要，但在这些早期阶段优化每个组件和所有代码不是有效的资源用法，因此应该予以避免。所以，应用程序可能存在您在设计阶段未预料到的性能问题。例如，您可能遇到由于两个系统或组件之间的无法预料的交互而产生的性能问题，或者您可能使用原来存在的、未按希望的方式执行的代码。在此情况下，您需要追究性能问题的根源，以便您可以适当地解决该问题。

本节讨论一些将帮助您诊断性能问题以及调整应用程序以获得最佳性能的工具和技术。

制定性能目标

当您设计和规划智能客户端应用程序时，您应该仔细考虑性能方面的要求，并且定义合适的性能目标。在定义这些目标时，请考虑您将如何度量应用程序的实际性能。您的性能度量标准应该明确体现应用程序的重要性能特征。请努力避免无法准确度量的模糊或不完整的目标，例如，"应用程序必须快速运行"或"应用程序必须快速加载"。您需要了解应用程序的性能和可伸缩性目标，以便您可以设法满足这些目标并且围绕它们来规划您的测试。请确保您的目标是可度量的和可验证的。

定义良好的性能度量标准使您可以准确跟踪应用程序的性能，以便您可以确定应用程序是否能够满足它的性能目标。这些度量标准应该包括在应用程序测试计划中，以便可以在应用程序的测试阶段度量它们。

本节重点讨论与智能客户端应用程序相关的特定性能目标的定义。如果您还要设计和生成客户端应用程序将消耗的网络服务，则您还需要为这些服务定义适当的性能目标。在此情况下，您应该确保考虑整个系统的性能要求，以及应用程序各个部分的性能与其他部分以及整个系统之间存在怎样的关系。

考虑用户的观点

当您为智能客户端应用程序确定合适的性能目标时，您应该仔细考虑用户的观点。对于智能客户端应用程序而言，性能与可用性和用户感受有关。例如，只要用户能够继续工作并且获得有关操作进度的足够反馈，用户就可以接受漫长的操作。

在确定要求时，将应用程序的功能分解为多个使用情景或使用案例通常是有用的。您应该识别对于实现特定性能目标而言关键且必需的使用案例和情景。应该将许多使用案例所共有且经常执行的任务设计得具有较高性能。同样，如果任务要求用户全神贯注并且不允许用户从其切换以执行其他任务，则需要提供优化的且有效的用户体验。如果任务不太经常使用且不会阻止用户执行其他任务，则可能无须进行大量调整。

对于您识别的每个性能敏感型任务，您都应该精确地定义用户的操作以及应用程序的响应方式。您还应该确定每个任务使用的网络和客户端资源或组件。该信息将影响性能目标，并且将驱动对性能进行度量的测试。

可用性研究提供了非常有价值的信息源，并且可能大大影响性能目标的定义。正式的可用性研究在确定用户如何执行他们的工作、哪些使用情景是共有的以及哪些不是共有的、用户经常执行哪些任务以及从性能观点看来应用程序的哪些特征是重要的等方面可能非常有用。如果您要生成新的应用程序，您应该考虑提供应用程序的原型或模型，以便可以执行基本的可用性测试。

考虑应用程序操作环境

对应用程序的操作环境进行评估是很重要的，因为这可能对应用程序施加必须在您制定的性能目标中予以反映的约束。

位于网络上的服务可能对您的应用程序施加性能约束。例如，您可能需要与您无法控制的 Web 服务进行交互。在这种情况下，需要确定该服务的性能，并且确定这是否将对客户端应用程序的性能产生影响。

您还应该确定任何相关服务和组件的性能如何随着时间的变化而变化。某些系统会经受相当稳定的使用，而其他系统则会在一天或一周的特定时间经受变动极大的使用。这些区别可能在关键时间对应用程序的性能造成不利影响。例如，提供应用程序部署和更新服务的服务可能会在星期一早上 9 点缓慢响应，因为所有用户都在此时升级到应用程序的最新版本。

另外，还需要准确地对所有相关系统和组件的性能进行建模，以便可以在严格模拟应用程序的实际部署环境的环境中测试您的应用程序。对于每个系统，您都应该确定性能概况以及最低、平均和最高性能特征。然后，您可以在定义应用程序的性能要求时根据需要使用该数据。

您还应该仔细考虑用于运行应用程序的硬件。您将需要确定在处理器、内存、图形功能等方面的目标硬件配置，或者至少确定一个如果得不到满足则无法保证性能的最低配置。

通常，应用程序的业务操作环境将规定一些更为苛刻的性能要求。例如，执行实时股票交易的应用程序将需要执行这些交易并及时显示所有相关数据。

性能调整过程

对应用程序进行性能调整是一个迭代过程。该过程由一些重复执行直至应用程序满足其性能目标的阶段组成。（请参见**图 8.2**。）

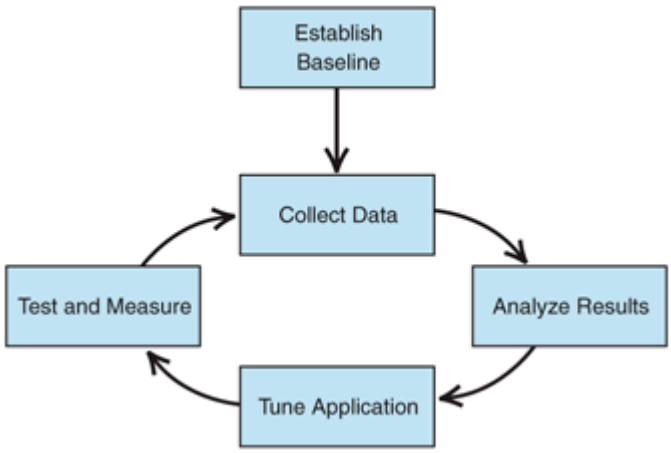


图 **8.2**：性能调整过程

正如图 **8.2** 所阐明的，性能调整要求您完成下列过程：

- **建立基准**。在您开始针对性能调整应用程序时，您必须具有与性能目标、目标和度量标准有关的定义良好的基准。这可能包括应用程序工作集大小、加载数据（例如，目录）的时间、事务持续时间等等。
- **收集数据**。您将需要通过针对您已经定义的性能目标度量应用程序的性能，来对应用程序性能进行评价。性能目标应该体现特定的且可度量的度量标准，以使您可以在任何时刻量化应用程序的性能。要使您可以收集性能数据，您可能必须对应用程序进行规范，以便可以发布和收集必需的性能数据。下一节将详细讨论您可以用来完成这一工作的一些选项。
- **分析结果**。在收集应用程序的性能数据之后，您将能够通过确定哪些应用程序功能要求最多的关注，来区分性能调整工作的轻重缓急。此外，您可以使用该数据来确定任何性能瓶颈的位置。通常，您将只能够通过收集更详细的性能数据来确定瓶颈的确切位置：例如，通过使用应用程序规范。性能分析工具可能帮助您识别瓶颈。
- **调整应用程序**。在已经识别瓶颈之后，您可能需要修改应用程序或其配置，以便尝试解决问题。您应该致力于将更改降低至最低限度，以便可以确定更改对应用程序性能的影响。如果您同时进行多项更改，可能难以确定每项更改对应用程序的总体性能的影响。
- **测试和度量**。在更改应用程序或其配置之后，您应该再次测试它以确定更改具有的效果，并且使新的性能数据得以收集。性能工作通常要求进行体系结构或其他具有较高影响的更改，因此彻底的测试是很关键的。

您的应用程序测试计划应该针对预料到的所有情况，在配置了适当硬件和软件的客户计算机上演习应用程序所实现的完整范围的功能。如果您的应用程序使用网络资源，则应该加载这些资源，以便您可以获得有关应用程序在此类环境中所具有的性能的准确度量。

上述过程将使您可以通过针对特定目标度量应用程序的总体性能，来重点解决特定的性能问题。

性能工具

您可以使用许多工具来帮助您收集和分析应用程序的性能数据。本节中介绍的每种工具都具有不同的功能，您可以使用这些功能来度量、分析和查找应用程序中的性能瓶颈。

注除了这里介绍的工具以外，您还可以使用其他一些选项和第三方工具。有关其他日志记录和异常管理选项的说明，请参阅 *Exception Management Architecture Guide*，网址为：

http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnbda/html/exceptdotnet.asp

在决定哪些工具最适合您的需要之前，您应该仔细考虑您的确切要求。

使用性能日志和警报

性能日志和警报是作为 Windows 操作系统的一部分发行的一种管理性能监控工具。它依靠由各种 Windows 组件、子系统和应用程序发布的性能计数器，使您可以跟踪资源使用情况以及针对时间以图形方式绘制它们。

您可以使用 Performance Logs and Alerts 来监控标准的性能计数器（例如，内存使用情况或处理器使用情况），或者您可以定义您自己的自定义计数器来监控应用程序特定的活动。

.NET CLR 提供了许多有用的性能计数器，它们使您可以洞察应用程序性能的好坏。关系比较大的一些性能对象是：

- **.NET CLR 内存**。提供有关托管 .NET 应用程序内存使用情况的数据，包括应用程序正在使用的内存数量以及对未使用的对象进行垃圾回收所花费的时间。

- **.NET CLR 加载**。提供有关应用程序正在使用的类和应用程序域的数量的数据，并且提供有关它们的加载和卸载速率的数据。
- **.NET CLR 锁和线程**。提供与应用程序内使用的线程有关的性能数据，包括线程个数以及试图同时对受保护的资源进行访问的线程之间的争用率。
- **.NET CLR 网络**。提供与通过网络发送和接收数据有关的性能计数器，包括每秒发送和接收的字节数以及活动连接的个数。
- **.NET CLR 异常**。提供有关应用程序所引发和捕获的异常个数的报告。

有关上述计数器、它们的阈值、要度量的内容以及如何度量它们的详细信息，请参阅 *Improving .NET Application Performance and Scalability* 的第 15 章"Measuring .NET Application Performance"中的"CLR and Managed Code"部分，网址为：http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnpag/html/scalenetchapt15.asp。

您的应用程序还可以提供您可以通过使用性能日志和警报轻松监控的、应用程序特定的性能计数器。您可以像以下示例所显示的那样，定义自定义性能计数器：

[C#]

```
PerformanceCounter counter = new PerformanceCounter( "Category", "CounterName", false );
```

[Visual Basic .NET]

```
Dim counter As New PerformanceCounter("Category", "CounterName", False)
```

在创建性能计数器对象之后，您可以为您的自定义性能计数器指定类别，并将所有相关计数器保存在一起。**PerformanceCounter** 类在 **System.Diagnostics** 命名空间中定义，该命名空间中还定义了其他一些可用于读取和定义性能计数器和类别的类。有关创建自定义性能计数器的详细信息，请参阅知识库中编号为 317679 的文章"How to create and make chang

es to a custom counter for the Windows Performance Monitor by using Visual Basic .NET”，网址为：http://support.microsoft.com/default.aspx?scid=kb;en-us;317679。

注要注册性能计数器，您必须首先注册该类别。您必须具有足够的权限才能注册性能计数器类别（它可能影响您部署应用程序的方式）。

规范

您可以使用许多工具和技术来帮助您对应用程序进行规范，并且生成度量应用程序性能所需的信息。这些工具和技术包括：

- **Event Tracing for Windows (ETW)**。该 ETW 子系统提供了一种系统开销较低（与性能日志和警报相比）的手段，用以监控具有负载的系统的性能。这主要用于必须频繁记录事件、错误、警告或审核的服务器应用程序。有关详细信息，请参阅 Microsoft Platform SDK 中的“Event Tracing”，网址为：http://msdn.microsoft.com/library/default.asp?url=/library/en-us/perfmon/base/event_tracing.asp。
- **Enterprise Instrumentation Framework (EIF)**。EIF 是一种可扩展且可配置的框架，您可以使用它来对智能客户端应用程序进行规划。它提供了一种可扩展的事件架构和统一的 API — 它使用 Windows 中内置的现有事件、日志记录和跟踪机制，包括 Windows Management Instrumentation (WMI)、Windows Event Log 和 Windows Event Tracing。它大大简化了发布应用程序事件所需的编码。如果您计划使用 EIF，则需要通过使用 EIF .msi 在客户计算机上安装 EIF。如果您要在智能客户端应用程序中使用 EIF，则需要在决定应用程序的部署方式时考虑这一要求。有关详细信息，请参阅“How To:Use EIF”，网址为：http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnpag/html/scalenethowto14.asp。
- **Logging Application Block**。Logging Application Block 提供了可扩展且可重用的代码组件，以帮助您生成规范化的应用程序。它建立在 EIF 的功能基础之上，以提供某些功能，例如，针对事件架构的增强功能、多个日志级别、附加的事件接收等等。有关详细信息，请参阅“Logging Application Block”，网址为“http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnpag/html/Logging.asp。

•**Windows Management Instrumentation (WMI)**。WMI 组件是 Windows 操作系统的一部分，并且提供了用于访问企业中的管理信息和控件的编程接口。系统管理员常用它来自动完成管理任务（通过使用调用 WMI 组件的脚本）。有关详细信息，请参阅 Windows Management Information，网址为：http://msdn.microsoft.com/library/default.asp?url=/library/en-us/wmisdk/wmi/wmi_start_page.asp。

•**调试和跟踪类**。.NET Framework 在 **System.Diagnosis** 下提供了 **Debug** 和 **Trace** 类来对代码进行规范。**Debug** 类主要用于打印调试信息以及检查是否有断言。**Trace** 类使您可以对发布版本进行规范，以便在运行时监控应用程序的完好状况。在 Visual Studio .NET 中，默认情况下启用跟踪。在使用命令行版本时，您必须为编译器添加 **/d:Trace** 标志，或者在 Visual C# .NET 源代码中添加 **#define TRACE**，以便启用跟踪。对于 Visual Basic .NET 源代码，您必须为命令行编译器添加 **/d:TRACE=True**。有关详细信息，请参阅知识库中编号为 815788 的文章"HOW TO:Trace and Debug in Visual C# .NET"，网址为：http://support.microsoft.com/default.aspx?scid=kb;en-us;815788。

CLR Profiler

CLR Profiler 是 Microsoft 提供的一种内存分析工具，并且可以从 MSDN 下载。它使您能够查看应用程序进程的托管堆以及调查垃圾回收器的行为。使用该工具，您可以获取有关应用程序的执行、内存分配和内存消耗的有用信息。这些信息可以帮助您了解应用程序的内存使用方式以及如何优化应用程序的内存使用情况。

CLR Profiler 可从 http://msdn.microsoft.com/netframework/downloads/tools/default.aspx 获得。有关如何使用 CLR Profiler 工具的详细信息，另请参阅"How to use CLR Profiler"，网址为：http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnpag/html/scalenethowto13.asp?frame=true。

CLR Profiler 在日志文件中记录内存消耗和垃圾回收器行为信息。然后，您可以使用一些不同的图形视图，通过 CLR Profiler 来分析该数据。一些比较重要的视图是：

•**Allocation Graph**。显示有关对象分配方式的调用堆栈。您可以使用该视图来查看方法进行的每个分配的系统开销，隔离您不希望发生的分配，以及查看方法可能进行的过度分配。

- **Assembly, Module, Function, and Class Graph**。显示哪些方法造成了哪些程序集、函数、模块或类的加载。
- **Call Graph**。使您可以查看哪些方法调用了其他哪些方法以及相应的调用频率。您可以使用该图表来确定库调用的系统开销，以及调用了哪些方法或对特定方法进行了多少个调用。
- **Time Line**。提供了有关应用程序执行的基于文本的、按时间顺序的层次结构视图。使用该视图可以查看分配了哪些类型以及这些类型的大小。您还可以使用该视图查看方法调用使得哪些程序集被加载，并且分析您不希望发生的分配。您可以分析完成器的使用情况，并且识别尚未实现或调用 **Close** 或 **Dispose** 从而导致瓶颈的方法。

您可以使用 CLR Profiler.exe 来识别和隔离与垃圾回收有关的问题。这包括内存消耗问题（例如，过度或未知的分配、内存泄漏、生存期很长的对象）以及在执行垃圾回收时花费的时间的百分比。

注有关如何使用 CLR Profiler 工具的详细信息，请参阅"Improving .NET Application Performance and Scalability"，网址为：http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnpag/html/scalenethowto13.asp?frame=true。

返回页首

小结

要完全实现智能客户端应用程序的潜能，您需要在应用程序的设计阶段仔细考虑性能问题。通过在早期阶段解决这些性能问题，您可以在应用程序设计过程中控制成本，并减小在开发周期的后期陷入性能问题的可能性。

本章分析了许多不同的技术，您可以在规划和设计智能客户端应用程序时使用这些技术，以确保优化它们的性能。本章还考察了您可以用来确定智能客户端应用程序内的性能问题的一些工具和技术。

返回页首

参考资料

有关详细信息，请参阅以下内容：

- http://msdn.microsoft.com/perf
- http://www.windowsforms.net/Default.aspx
- http://msdn.microsoft.com/vstudio/using/understand/perf/
- http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnnetcomp/html/netcfimproveformloadperf.asp
- http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dndotnet/html/highperfmanagedapps.asp
- http://msdn.microsoft.com/msdnmag/issues/02/08/AdvancedBasics/default.aspx
- http://msdn.microsoft.com/library/default.asp?url=/msdnmag/issues/04/01/NET/toc.asp?frame=true
- http://msdn.microsoft.com/library/default.asp?url=/msdnmag/issues/03/02/Multithreading/toc.asp?frame=true

posted on 2007-12-10 17:04 寒蝉 阅读(288) 评论(0) 编辑 收藏 所属分类: Smart Client(智能客户端)系列学习笔记

今天看 MSDN 里头的 C# 委托示例,代码中的注释是英文的,看着不舒服,我把它翻译了一下,示例中提到的书也换成了中文的,

```
using System;

// 书店
namespace Bookstore
{
    using System.Collections;

    //  描述图书列表中的书
    public struct Book
    {
        public string Title;          // 书名
        public string Author;         // 作者
        public decimal Price;         // 价格
        public bool Paperback;        // 是不是平装

        public Book(string title, string author, decimal price, bool paperBack)
        {
            Title = title;
            Author = author;
            Price = price;
            Paperback = paperBack;
        }
    }



    // 声明一个委托类型,用来处理图书:
    public delegate void ProcessBookDelegate(Book book);

    // 维护一个图书数据库
    public class BookDB
    {
        // 数据库中所有图书的列表
        ArrayList list = new ArrayList();

        // 向数据库添加一本书
        public void AddBook(string title, string author, decimal price, bool paperBack)
        {
            list.Add(new Book(title, author, price, paperBack));
        }
```

```
        // 调用传递进来的委托,处理每一本书
        public void ProcessPaperbackBooks(ProcessBookDelegate processBook)
        {
            foreach (Book b in list)
            {
                if (b.Paperback)
                    // 调用委托:
                    processBook(b);
            }
        }
    }
}


// 使用 Bookstore 中的类:
namespace BookTestClient
{
    using Bookstore;

    // 计算图书总价和平均价格
    class PriceTotaller
    {
        int countBooks = 0;// 图书本数
        decimal priceBooks = 0.0m;// 图书总价

        // 添加图书
        internal void AddBookToTotal(Book book)
        {
            countBooks += 1;
            priceBooks += book.Price;
        }


        //  平均价格
        internal decimal AveragePrice()
        {
            return priceBooks / countBooks;
        }
    }


    // 这个类用来测试图书数据库
    class Test
```

```
    {
        //  打印书名
        static void PrintTitle(Book b)
        {
            Console.WriteLine("    {0}", b.Title);
        }

        // 主函数,程序从这里开始执行
        static void Main()
        {
            BookDB bookDB = new BookDB();

            // 用几本书初始化数据库
            AddBooks(bookDB);

            // 打印所有平装书的书名
            Console.WriteLine("平装书:");
            // 创建一个与静态方法 Test.PrintTitle 相关联的委托:
            bookDB.ProcessPaperbackBooks(new ProcessBookDelegate(PrintTitle));

            // 用一个 PriceTotaller 对象获取平装书的平均价格
            PriceTotaller totaller = new PriceTotaller();

            // 创建一个与对象 totaller 的实例方法 AddBookToTotal 相关联的委托实例
            bookDB.ProcessPaperbackBooks(new
ProcessBookDelegate(totaller.AddBookToTotal));

            Console.WriteLine("平装书的平均价格是: ${0:#.##}",
totaller.AveragePrice());
        }

        // 用几本书初始化数据库
        static void AddBooks(BookDB bookDB)
        {
            bookDB.AddBook("呐喊",          "鲁迅",    19.95m,true);
            bookDB.AddBook("我的人生哲学","王蒙",    39.95m,true);
            bookDB.AddBook("三重门",        "韩寒",    129.95m,false);
            bookDB.AddBook("那小子真帅",  "可爱淘", 12.00m,true);
        }
    }
}
```

C# 异步调用机制的理解（一） - 思索的秋天 - 博客园

通常，在进行一个对象的方法调用时，对象执行期间客户端通常都是堵塞的，只有等方法执行完毕返回控制权时才会回到客户端，然而某些时候，我们需要异步调用方法，即对象在后台执行方法调用，控制权可以立即返回到客户端，随后能以某种方式通知客户端已经执行完毕，这种执行模式就是所谓的异步调用方法，而操作就是大家所知的异步调用。异步操作通常用于执行完成时间可能较长的任务，如打开大文件、连接远程计算机或查询数据库。异步操作在主应用程序线程以外的线程中执行。应用程序调用方法异步执行某个操作时，应用程序可在异步方法执行其任务时继续执行。

.NET Framework 为异步操作提供两种设计模式：

使用 IAsyncResult 对象的异步操作。

使用事件的异步操作

今天主要讲的是使用 IAsyncResult 对象的异步操作，利用一个委托来进行异步编程。

一： 异步调用编程模型

支持异步调用使用多线程肯定是必须的，但是如果每一次异步调用都创建一个新线程的话肯定是一个资源的浪费。在.Net 中，为我们提供了线程池，线程池在我们每次进行异步调用的时候都会给调用方分配一个线程，然后控制权会在很短的时间内返回到客户端。

总的来说 BeginInvoke()方法发起一个异步调用，EndInvoke()管理方法的完成，包括获取输出参数和返回值。要进行异步调用的说明还需要理解很多其他的概念，包括委托，事件，多线程等，这些概念我就不一一道来了。

下面是整个说明过程中用到的一个类（摘抄而来）：

```
public class CalCulator
{
    public int Add(int argument1, int argument2)
    {
        return argument1 + argument2;
    }
    public int Subtract(int argument1, int argument2)
    {
        return argument1 - argument2;
    }
}
```

一个委托： `public delegate int BinaryOperation(int argument1, int argument2);`

二：使用 beginInvoke()和 EndInvoke() 方法

异步委托提供以异步方式调用同步方法的能力。当同步调用一个委托时，“Invoke”方法直接对当前线程调用目标方法。如果编译器支持异步委托，则它将生成“Invoke”方法以及“BeginInvoke”和“EndInvoke”方法。如果调用“BeginInvoke”方法，则公共语言运行库

(CLR)
将对请求进行排队并立即返回到调用方。将对来自线程池的线程调用该目标方法。提交请求的原始线程自由地继续与目标方法并行执行，该目标方法是对线程池线程运行的。如果在对“BeginInvoke”方法的调用中指定了回调方法，则当目标方法返回时将调用该回调方法。在回调方法中，“EndInvoke”方法获取返回值和所有输入/输出参数。如果在调用“BeginInvoke”时未指定任何回调方法，则可以从调用“BeginInvoke”的线程中调用“EndInvoke”。

对于上面的文字意思用代码表示如下：

1.编译器生成的 BinaryOperaion 定义：

```
public sealed class BinaryOperaion : MulticastDelgate
{
    public BinaryOperaion(object target, int methodPtr)
    {
        ...
    }
    public virtual int Invoke(int argument1, int argument2)
    { ...}
    public virtual IAsyncResult BeginInvoke(int argument1, int
argument2, AsyncCallback callback, object asyncState)
    {
        ...
    }
    public virtual int EndInvoke(IAsyncResult result)
    { ...}
}
```

2.同步调用：

```
public delegate int BinaryOperation(int argument1, int argument2);
CalCulator calculator = new CalCulator();
BinaryOperaion oppDel = calculator.Add;
oppDel(2,3);
```

3 .异步调用：

首先说明下 BeginInvoke（）方法返回的 IAsyncResult 接口对象，在.Net 中 IAsyncResult 的定义如下：

```
    // 摘要:
//      表示异步操作的状态。
[ComVisible(true)]
public interface IAsyncResult
{
    // 摘要:
    //      获取用户定义的对象，它限定或包含关于异步操作的信息。
    //
    // 返回结果:
    //      用户定义的对象，它限定或包含关于异步操作的信息。
```

```
        object AsyncState { get; }
        //
        // 摘要:
        //     获取用于等待异步操作完成的 System.Threading.WaitHandle。
        //
        // 返回结果:
        //     用于等待异步操作完成的 System.Threading.WaitHandle。
        WaitHandle AsyncWaitHandle { get; }
        //
        // 摘要:
        //     获取异步操作是否同步完成的指示。
        //
        // 返回结果:
        //     如果异步操作同步完成，则为 true；否则为 false。
        bool CompletedSynchronously { get; }
        //
        // 摘要:
        //     获取异步操作是否已完成的指示。
        //
        // 返回结果:
        //     如果操作完成则为 true，否则为 false。
        bool IsCompleted { get; }
    }
}
```

从它的定义我们可以看出很多东西， 如获取异步操作是否同步完成的指示，我们可以把 IAsyncResult 对象传递给 EndInvoke()
方法，来标识你希望检索的那个特定的异步方法；如下所示：

```
        CalCulator calculator = new CalCulator();
        BinaryOperaion oppDel = calculator.Add;
        IAsyncResult asynResult = oppDel.BeginInvoke(2, 3, null, null);
        int iResult=oppDel.EndInvoke(asynResult);
        System.Diagnostics.Debug.Assert(iResult==5);
```

虽然例子很简单，但是要运用到实际的程序编写中的话，还是要多下点苦功夫的，在上面有几个关键点，其中使用 EndInvoke()方法我们可以获取所有输出参数和方法的返回值，它堵塞了调用者一直到它等待的对象返回。这里有几个需要注意的地方：

首先：EndInvoke()方法在每次调用异步操作时只能执行一次；
其次：当使用委托进行异步调用时，委托内部列表之允许一个目标方法；
最后，在将 IAsynResult 对象传递给 EndInvoke()方法时，委托和 IAsynResult 对象必须要匹配。

三：轮询或等待完成

有的时候客户端只是想知道某个操作是否完成了，或者想等待一段时间然后再做一些有限的处理，然后再继续等待或处理。这个时候我们可以使用 IAsynResult 的 AsyncWaitHandle 属性进行堵塞，一直到方法完成
，方式如下：asyncResult.AsyncWaitHandle.WaitOne();当然也可以指定超时
;其实在有的时候我们有很多方法需要在异步环境下进行的时候我们可以这样来等待多个方法的完成，我们可以获取到一个 WaitHandle 数组，然后调用它的 WaitAll（）方法等待多个异步方法的完成，这个在管理多个异步方法尤其有优势。

总结：
由于篇幅有限，还有些没讲的怎么详细，希望在下篇能分开慢慢道来，在进行异步的过程中，其实对于怎样完成回调方法也是很有用的，我们可以在回调方法中进行我们需要的一些处理，如发出通知给客户端（这个过程应该是事件驱动的）。
自己其实也只是在一个自动更新模块中用到了一些，理解肯定还不深，还是那句话，多多指教。
posted on 2007-09-15 12:58 寒蝉 阅读(339) 评论(0) 编辑 收藏 所属分类: C#读书笔记

Copyright ©2007 寒蝉 Powered by: 博客园 模板提供：沪江博客

C#的多线程能力

楼主 celineshi（）2006-11-08 14:40:47 在 专题开发/技术/项目 / 英特尔多核计算技术提问

线程是允许进行并行计算的一个抽象概念：在另一个线程完成计算任务的同时，一个线程可以对图像进行更新，二个线程可以同时处理同一个进程发出的二个网络请求。我们在这篇文章中将重点讨论 Java 和 C#在线程方面的不同之处，并将一些 Java 中线程的常用模式转换为 C#。

从概念上讲，线程提供了一种在一个软件中并行执行代码的方式——每个线程都“同时”在一个共享的内存空间中执行指令，（当然是在一个处理器上，这是通过处于运行状态的线程的交替执行完成的。），因此，每个线程都可以访问一个程序内的数据结构。由于这种原因，多线程编程的难度就可想而知了，因为一个程序内有许多不同的线程需要安全地共享数据。

线程的创建和运行

Java 在 java.lang.Thread 和 java.lang.Runnable 类中提供了大部分的线程功能。创建一个线程非常简单，就是扩展 Thread 类，并调用 start()。通过创建一个执行 Runnable()的类，并将该类作为参数传递给 Thread()，也可以定义一个线程。仔细地阅读下面这个简单的 Java 程序，其中有 2 个线程同时在从 1 数到 5，并将结果打印出来。

```
public class ThreadingExample
extends Object {
public static void main( String args[] ) {
Thread[] threads = new Thread[2];
for( int count=1;count<=threads.length;count ) {
threads[count] = new Thread( new Runnable() {
public void run() {
count();

}

} );
threads[count].start();

}

}

public static void count() {
for( int count=1;count<=5;count )
System.out.print( count ″ ″ );

}
```

```
}
```

我们可以使用 System.Threading.Thread 和 System.Threading.ThreadStart 二个类将上述的 Java 程序转换为 C#语言：

```
using System.Threading;
public class ThreadingExample : Object {
public static void Main() {
Thread[] threads = new Thread[2];
for( int count=1;count<=threads.Length;count ) {
threads[count] = new Thread( new ThreadStart( Count ) );

threads[count].Start();

}

}

public static void Count() {
for( int count=1;count<=5;count )
Console.Write( count " " );

}

}
```

这个例子中有一些小技巧。Java 允许扩展 java.lang.Thread 类和执行 java.lang.Runnable 接口，C#则没有为我们提供这些便利。一个 C#中的 Thread 对象是不可知的，必须通过 ThreadStart 进行创建，这意味着不能使用内部的类模式，而必须创建一个对象，而且必须传递给线程一个对象的方法供线程执行用。

线程的使用

Java 中存在许多编程人员希望能够对线程使用的标准操作：例如，测试线程是否存在、加入一个线程直到它死亡、杀死一个线程等。

表 1：线程管理的函数

Java 中 java.lang.Thread 中的方法和 C#中 System.Threading.Thread 对象的对比。

setDaemon(boolean on) 方法

IsBackground 设置属性值

使一个存在的进程成为一个新线程（如果剩下的所有进程都成了新线程，程序将停止运行）。

isDaemon()方法

IsBackground 获取属性

如果该线程是一个后台线程，则返回真值。

isAlive()　方法
IsAlive　获取属性
如果该线程处于活动状态，则返回真值。

interrupt()　方法
Interrupt()　方法
尽管在 Java 中这一方法可以用来设置线程的中断状态，而且可以用来检查线程是否被中断。在 C#中没有相应的方法，对一个没有处于阻塞状态的线程执行 Interrupt 方法将使下一次阻塞调用自动失效。

isInterrupted()　方法
n/a
如果该线程处于阻塞状态，则返回真值。

sleep(long millis)和 sleep(long millis, int nanos)
Sleep(int millisecondTimeout) and Sleep(System.TimeSpan) 方法
使正在执行的线程暂停一段给定的时间，或直到它被中断。这一方法将在 Java 中将产生一个 java.lang.InterruptedException 状态，在 C#中将产生 System.Threading. ThreadInterruptedException 状态。

join()、join(long millis)和 join(long millis, int nanos) 方法
Join()、Join(int millisecondTimeout)和 Join(System.TimeSpan) 方法　与 Java 中仅依靠超时设定不同的是，在 C#语言中则依据线程停止运行是由于线程死亡（返回真）或是超时（返回假）而返回一个布尔型变量。

suspend()　方法
Suspend()　方法
二者的功能相同。这一方法容易引起死循环，如果一个占有系统关健资源的线程被挂起来，则在这一线程恢复运行之前，其他的线程不能访问该资源。

resume()　方法
Resume()　方法
恢复一个被挂起的线程。

stop()　方法
Abort()　方法
参见下面的“线程停止”部分。
（特别说明，在上面的表中，每个小节的第一行是 java 中的方法，第二行是 C#中的方法，第三行是有关的注释，由于在文本文件中不能组织表格，请编辑多费点心组织表格，原文中有表格的格式。）

线程的中止

由于能够在没有任何征兆的情况下使运行的程序进入一种混乱的状态，Java 中的 Thread.stop 受到了普遍的反对。根据所调用的 stop()方法，一个未经检查的 java.lang.ThreadDeath 错误将会破坏正在运行着的程序的栈，随着它的不断运行，能够解除任何被锁定的对象。由于这些锁被不分青红皂白地被打开，由它们所保护的数据就非常可能陷入混乱状态中。

根据当前的 Java 文档，推荐的中止一个线程的方法是让运行的线程检查一个由其他的线程能够改变的变量，该变量代表一个“死亡时间”条件。下面的程序就演示了这种方法。

```
//  条件变量
private  boolean  timeToDie  =  false;

//  在每次迭代中对条件变量进行检查。
class  StoppableRunnable
extends  Runnable  {
public  void  run()  {
while(  !timeToDie  )  {
//  进行相应的操作

}

}

}
```

上述的讨论对 C#中的 Abort 方法也适合。根据调用的 Abort 方法，令人捉摸不定的 System.Threading.ThreadAbortException 可能会破坏线程的栈，它可能释放线程保持的一些变量，使处于保护状态中的数据结构出现不可预测的错误。我建议使用与上面所示的相似的方法来通知一个应该死亡的线程。

线程的同步

从概念上来看，线程非常易于理解，实际上，由于他们可能交互地对同一数据结构进行操作，因此它们成为了令编程人员头疼的一种东西。以本文开始的 ThreadingExample 为例，当它运行时，会在控制台上输出多种不同的结果。从 1 2 3 4 5 1 2 3 4 5 到 1 1 2 2 3 3 4 4 5 5 或 1 2 1 2 3 3 4 5 4 5 在内的各种情况都是可能出现的，输出结果可能与操作系统的线程调度方式之间的差别有关。有时，需要确保只有一个线程能够访问一个给定的数据结构，以保证数据结构的稳定，这也是我们需要线程同步机制的原因所在。

为了保证数据结构的稳定，我们必须通过使用“锁”来调整二个线程的操作顺序。二种语言都通过对引用的对象申请一个“锁”，一旦一段程序获得该“锁”的控制权后，就可以保证只有它获得了这个“锁”，能够对该对象进行操作。同样，利用这种锁，一个线程可以一直处于等待状态，直到有能够唤醒它信号通过变量传来为止。

表 2：线程同步

需要对线程进行同步时需要掌握的关健字

synchronized

lock

C#中的 lock 命令实际上是为使用 System.Threading.Monitor 类中的 Enter 和 Exit 方法的语法上的准备

Object.wait()
Monitor.Wait(object obj)
C#中没有等待对象的方法，如果要等待一个信号，则需要使用 System.Threading.Monitor 类，这二个方法都需要在同步的程序段内执行。

Object.notify()
Monitor.Pulse(object obj)
参见上面的 Monitor.Wait 的注释。

Object.notify()
Monitor.PulseAll(object obj)
参见上面的 Monitor.Wait 的注释。
（特别说明，在上面的表中，每个小节的第一行是 java 中的方法，第二行是 C#中的方法，第三行是有关的注释，由于在文本文件中不能组织表格，请编辑多费点心组织表格，原文中有表格的格式。）

我们可以对上面的例子进行一些适当的修改，通过首先添加一个进行同步的变量，然后对 count()方法进行如下的修改，使变量在“锁”中被执行加 1 操作。

```
public static Object synchronizeVariable = "locking variable";

public static void count() {
synchronized( synchronizeVariable ) {
for( int count=1;count<=5;count ) {
System.out.print( count " " );
synchronizeVariable.notifyAll();
if( count < 5 )
try {
synchronizeVariable.wait();

} catch( InterruptedException error ) {

}

}

}

}
```

作了上述的改变后，每次只有一个线程（因为一次只能有一个线程获得 synchronizeVariable）能够执行 for loop 循环输出数字 1；然后，它会唤醒所有等待

synchronizeVariable 的线程（尽管现在还没有线程处于等待状态。），并试图获得被锁着的变量，然后等待再次获得锁变量；下一个线程就可以开始执行 for loop 循环输出数字 1，调用 notifyAll()唤醒前面的线程，并使它开始试图获得 synchronizeVariable 变量，使自己处于等待状态，释放 synchronizeVariable，允许前面的线程获得它。这个循环将一直进行下去，直到它们都输出完从 1 到 5 的数字。

通过一些简单的语法变化可以将上述的修改在 C#中实现：

```
public static Object synchronizeVariable = "locking variable";

public static void count() {
lock( synchronizeVariable ) {
for( int count=1;count<=5;count ) {
System.out.print( count " " );
Monitor.PulseAll( synchronizeVariable );
if( count < 5 )
Monitor.Wait( synchronizeVariable );

}

}

}
```

C#中特有的线程功能

象我们一直对 C#所抱的期望那样，C#中确实有一些 Java 不支持的方法、类和函数，对于铁杆的 Java 线程编程人员而言，这可是一件好事，因为他们可以用 C#编写代码，然后在 Java 代码中引用。

Enter/TryEnter/Exit

要在 Java 中获得某一变量的锁，必须在代码的首尾二端加上 synchronized 关健字，指明需要获得锁的对象。一旦线程开始执行 synchronized 块中的代码，它就获得了对这一对象的锁的控制权。同样，一旦线程已经离开了 synchronized 块，它也将释放这一对象的锁。我们已经知道，C#也有一个相似的被称作 lock 的关健字。除了 lock 这个关健字外，C#还提供了内置的获得和释放锁的方法：Monitor.Enter(object obj) 和 Monitor.Exit(object obj)，通过使用这些方法，编程人员可以获得与使用 lock 相同的作用，但提供了更精确的控制方法。例如，可以在一个方法中锁定几个变量，而不同时或在代码中的不同部分释放它们。

对一个需要进行同步的对象执行 System.Threading.Monitor.Enter 操作将使线程获得该对象的锁，或者在由其他线程控制着该对象的锁时进行阻塞。通过执行 Monitor.Exit 方法就可以释放锁，如果线程已经不控制着该对象的锁了，这一方法将会产生一个 System.Threading.SynchronizationLockException 异常信号。

C#中的 Monitor 类不但包括 Enter 方法，还包括 TryEnter 方法，如果执行该方法，就会或者获得一个锁，或者返回一个表明它不能获得锁的返回值。

原子操作

System.Threading.Interlocked 类提供了程序对由几个线程共享的变量进行同步访问的能力，C#把一些操作抽象为“原子”操作或“不可分割”的操作。为了说明这一问题是如何

解决的，我们来看一下下面的 Java 代码：

```
public static int x = 1;
        public static void increment() {
x = x 1;
        }
```

如果有二个不同的线程同时调用 increment()，x 最后的值可能是 2 或 3，发生这种情况的原因可能是二个进程无序地访问 x 变量，在没有将 x 置初值时对它执行加 1 操作；在任一线程有机会对 x 执行加 1 操作之前，二个线程都可能将 x 读作 1，并将它设置为新的值。

在 Java 和 C#中，我们都可以实现对 x 变量的同步访问，所有进程都可以按各自的方式运行。但通过使用 Interlocked 类，C#提供了一个对这一问题更彻底的解决方案。Interlocked 类有一些方法，例如 Increment(ref int location)、Decrement(ref int location)，这二个方法都取得整数型参数，对该整数执行加或减 1 操作，并返回新的值，所有这些操作都以“不可分割的”方式进行，这样就无需单独创建一个可以进行同步操作的对象，如下例所示：

```
public static Object locker = ...
public static int x = 1;

public static void increment() {
synchronized( locker ) {
x = x 1;
      }
        }
```

C#中的 Interlocked 类可以用下面的代码完成相同的操作：

```
public static int x = 1;

public static void Increment() {
Interlocked.Increment( ref x );
        }
```

Interlocked 中还包括一个名字为 Exchange 的方法，可以“不可分割”地将一个变量的值设置为另一个变量的值。

线程池

如果许多利用了线程的应用软件都创建线程，这些线程将会因等待某些条件（键盘或新的 I/O 输入等）而在等待状态中浪费大部分的时间，C#提供的 System.Threading.ThreadPool 对象可以解决这一问题。使用 ThreadPool 和事件驱动的编程机制，程序可以注册一个 System.Threading.WaitHandle 对象（WaitHandle 是 C#编程中等待和通知机制的对象模型。）和 System.Threading.WaitOrTimerCallback 对象，所有的线程无需自己等待 WaitHandle 的释放，ThreadPool 将监控所有向它注册的 WaitHandle，然后在 WaitHandle 被释放后调用相应 WaitOrTimerCallback 对象的方法。

结束语

在本篇文章中我们简单地讨论了 C#提供的用于线程和并行操作的机制，其中的大部分与 Java 相似——C#提供了可以运行提供的方法的 Thread 对象，同时提供了对代码访问进行同步的方法。与在其他方面一样，C#在线程方面也提供了一些 Java 不支持的语法（在一定程度上，揭示了同步操作的一些底层的内容。），Java 编程人员可能会发现这一部分非常有用。

C#异步编程

同步方法和异步方法的区别

同步方法调用在程序继续执行之前需要等待同步方法执行完毕返回结果
异步方法则在被调用之后立即返回以便程序在被调用方法完成其任务的同时执行其它操作

异步编程概览

.NET Framework 允许您异步调用任何方法。定义与您需要调用的方法具有相同签名的委托；公共语言运行库将自动为该委托定义具有适当签名

的 BeginInvoke 和 EndInvoke 方法。

BeginInvoke 方法用于启动异步调用。它与您需要异步执行的方法具有相同的参数，只不过还有两个额外的参数（将在稍后描述）。

BeginInvoke 立即返回，不等待异步调用完成。
BeginInvoke 返回 IasyncResult，可用于监视调用进度。

EndInvoke 方法用于检索异步调用结果。调用 BeginInvoke 后可随时调用 EndInvoke 方法；如果异步调用未完成，EndInvoke 将一直阻塞到

异步调用完成。EndInvoke 的参数包括您需要异步执行的方法的 out 和 ref 参数（在 Visual Basic 中为 <Out> ByRef 和 ByRef）以及由

BeginInvoke 返回的 IAsyncResult。

四种使用 BeginInvoke 和 EndInvoke 进行异步调用的常用方法。调用了 BeginInvoke 后，可以：

1.进行某些操作，然后调用 EndInvoke 一直阻塞到调用完成。
2.使用 IAsyncResult.AsyncWaitHandle 获取 WaitHandle,使用它的 WaitOne 方法将执行一直阻塞到发出 WaitHandle 信号，然后调用

EndInvoke。这里主要是主程序等待异步方法，等待异步方法的结果。
3.轮询由 BeginInvoke 返回的 IAsyncResult，IAsyncResult.IsCompeted 确定异步调用何时完成，然后调用 EndInvoke。此处理个人认为与
相同。
4.将用于回调方法的委托传递给 BeginInvoke。该方法在异步调用完成后在 ThreadPool 线程上执行，它可以调用 EndInvoke。这是在强制装

换回调函数里面 IAsyncResult.AsyncState(BeginInvoke 方法的最后一个参数)成委托，然后用委托执行 EndInvoke。
警告　始终在异步调用完成后调用 EndInvoke。

以上有不理解的稍后可以再理解。

例子

1）先来个简单的没有回调函数的异步方法例子

请再运行程序的时候，仔细看注释，对理解很有帮助。还有，若将注释的中的两个方法都同步，你会发现异步运行的速度优越性。

```
using System;


namespace ConsoleApplication1
    {
  class Class1
      {
    //声明委托
    public delegate void AsyncEventHandler();

    //异步方法
    void Event1()
        {
      Console.WriteLine("Event1 Start");
      System.Threading.Thread.Sleep(4000);
      Console.WriteLine("Event1 End");
```

```
}

// 同步方法
void Event2()
        {
    Console.WriteLine("Event2 Start");
    int i=1;
    while(i<1000)
            {
        i=i+1;
        Console.WriteLine("Event2 "+i.ToString());
    }
    Console.WriteLine("Event2 End");
}

[STAThread]
static void Main(string[] args)
        {
    long start=0;
    long end=0;
    Class1 c = new Class1();
    Console.WriteLine("ready");
    start=DateTime.Now.Ticks;
```

```
            //实例委托
            AsyncEventHandler asy = new AsyncEventHandler(c.Event1);
            //异步调用开始，没有回调函数和 AsyncState,都为 null
            IAsyncResult ia = asy.BeginInvoke(null, null);
            //同步开始，
            c.Event2();
            //异步结束，若没有结束，一直阻塞到调用完成，在此返回该函数的 return，若有返回值。
            asy.EndInvoke(ia);

            //都同步的情况。
            //c.Event1();
            //c.Event2();


            end =DateTime.Now.Ticks;
            Console.WriteLine("时间刻度差="+ Convert.ToString(end-start) );
            Console.ReadLine();
        }
    }
}
```

2）下面看有回调函数的 WebRequest 和 WebResponse 的异步操作。

```
using System;

using System.Net;

using System.Threading;

using System.Text;

using System.IO;


// RequestState 类用于通过

// 异步调用传递数据

public class RequestState

        {

    const int BUFFER_SIZE = 1024;

    public StringBuilder RequestData;

    public byte[] BufferRead;

    public HttpWebRequest Request;

    public Stream ResponseStream;

    // 创建适当编码类型的解码器

    public Decoder StreamDecode = Encoding.UTF8.GetDecoder();


    public RequestState()

            {

        BufferRead = new byte[BUFFER_SIZE];

        RequestData = new StringBuilder("");

        Request = null;
```

```
        ResponseStream = null;
    }
}

// ClientGetAsync 发出异步请求
class ClientGetAsync
        {
    public static ManualResetEvent allDone = new ManualResetEvent(false);
    const int BUFFER_SIZE = 1024;
    public static void Main(string[] args)
            {
        if (args.Length < 1)
                {
            showusage();
            return;
        }

       // 从命令行获取 URI
        Uri HttpSite = new Uri(args[0]);
        // 创建请求对象
        HttpWebRequest wreq = (HttpWebRequest)WebRequest.Create(HttpSite);
        // 创建状态对象
```

```
            RequestState rs = new RequestState();

            // 将请求添加到状态，以便它可以被来回传递

            rs.Request = wreq;

            // 发出异步请求

            IAsyncResult r = (IAsyncResult)wreq.BeginGetResponse(new AsyncCallback(RespCal
lback), rs);

            // 将 ManualResetEvent 设置为 Wait，

            // 以便在调用回调前，应用程序不退出

            allDone.WaitOne();

        }


        public static void showusage()

            {

            Console.WriteLine("尝试获取 (GET) 一个 URL");

            Console.WriteLine("\r\n 用法：:");

            Console.WriteLine("ClientGetAsync URL");

            Console.WriteLine("示例：:");

            Console.WriteLine("ClientGetAsync http://www.microsoft.com/net/");

        }


        private static void RespCallback(IAsyncResult ar)

            {

            // 从异步结果获取 RequestState 对象
```

```
            RequestState rs = (RequestState)ar.AsyncState;

            // 从 RequestState 获取 HttpWebRequest

            HttpWebRequest req = rs.Request;

            // 调用 EndGetResponse 生成 HttpWebResponse 对象

            // 该对象来自上面发出的请求

            HttpWebResponse resp = (HttpWebResponse)req.EndGetResponse(ar);


            // 既然我们拥有了响应，就该从

            // 响应流开始读取数据了

            Stream ResponseStream = resp.GetResponseStream();


            // 该读取操作也使用异步完成，所以我们

            // 将要以 RequestState 存储流

            rs.ResponseStream = ResponseStream;


            // 请注意，rs.BufferRead 被传入到 BeginRead。

            // 这是数据将被读入的位置。

            IAsyncResult iarRead = ResponseStream.BeginRead(rs.BufferRead, 0, BUFFER_SIZ
E, new AsyncCallback(ReadCallBack), rs);

        }



        private static void ReadCallBack(IAsyncResult asyncResult)

            {

            // 从 asyncresult 获取 RequestState 对象
```

```
RequestState rs = (RequestState)asyncResult.AsyncState;

// 取出在 RespCallback 中设置的 ResponseStream
Stream responseStream = rs.ResponseStream;

// 此时 rs.BufferRead 中应该有一些数据。
// 读取操作将告诉我们那里是否有数据
int read = responseStream.EndRead(asyncResult);

if (read > 0)
        {
    // 准备 Char 数组缓冲区，用于向 Unicode 转换
    Char[] charBuffer = new Char[BUFFER_SIZE];

    // 将字节流转换为 Char 数组，然后转换为字符串
    // len 显示多少字符被转换为 Unicode
    int len = rs.StreamDecode.GetChars(rs.BufferRead, 0, read, charBuffer, 0);
    String str = new String(charBuffer, 0, len);


    // 将最近读取的数据追加到 RequestData stringbuilder 对象中，
    // 该对象包含在 RequestState 中
    rs.RequestData.Append(str);
    // 现在发出另一个异步调用，读取更多的数据
    // 请注意，将不断调用此过程，直到
```

```
            // responseStream.EndRead 返回 -1
            IAsyncResult ar = responseStream.BeginRead(rs.BufferRead, 0, BUFFER_SIZE, new AsyncCallback(ReadCallBack), rs);
        }
        else
            {
            if (rs.RequestData.Length > 1)
                {
                // 所有数据都已被读取，因此将其显示到控制台
                string strContent;
                strContent = rs.RequestData.ToString();
                Console.WriteLine(strContent);
            }

            // 关闭响应流
            responseStream.Close();

            // 设置 ManualResetEvent，以便主线程可以退出
            allDone.Set();
        }
        return;
    }
```

```
}



在这里有回调函数，且异步回调中又有异步操作。
```

首先是异步获得 ResponseStream，然后异步读取数据。

这个程序非常经典。从中可以学到很多东西的。我们来共同探讨。

总结

上面说过，.net framework 可以异步调用任何方法。所以异步用处广泛。

在.net framework 类库中也有很多异步调用的方法。一般都是已 Begin 开头 End 结尾构成一对，异步委托方法，外加两个回调函数和 AsyncState 参数，组成异步操作的宏观体现。所以要做异步编程，不要忘了委托 delegate、Begin，End，AsyncCallBack 委托，AsyncState 实例(在回调函数中通过 IAsyncResult.AsyncState 来强制转换)，IAsycResult(监控异步)，就足以理解异步真谛了。

C#异步数据处理及进度显示

对于 C#的事件,指代(Delegate)总是感觉理解不太深刻。这几天正好有机会学习了一下。从一个程序中改了一部分代码，实现了一个异步数据处理基本构架。

```
using System;

using System.Collections.Generic;

using System.ComponentModel;

using System.Data;

using System.Drawing;

using System.Text;

using System.Windows.Forms;

using System.Threading;



namespace testManager

    {

    //progressbar 窗体
    //有一个 cancel 按钮，和一个进度条

    public class ProgressForm : Form

        {



        private System.ComponentModel.IContainer components = nul
l;



        protected override void Dispose(bool disposing)

            {
```

```
        if (disposing && (components != null))
            {
            components.Dispose();
        }
        base.Dispose(disposing);
    }

        Windows InitializeComponent

    private System.Windows.Forms.ProgressBar poProgressBar;
    private System.Windows.Forms.Button btnCancel;

    //定义进度状态的事件。
    public delegate void StatusHandler();
    public event StatusHandler StatusChanged;

    private int iStatus = 1;//1:normal 0:cancel -1:exception

    //返回进度状态
    public int GetStatus
        {
            get { return iStatus; }
    }
```

```
        public ProgressForm()
            {
            InitializeComponent();
            this.poProgressBar.Value = 0;
        }

        //进度条加 1
        public void PerformStep()
            {
            try
                {
                this.poProgressBar.PerformStep();
                this.poProgressBar.Refresh();
                if (this.poProgressBar.Value == 100)
                    {
                    iStatus = 1;
                    this.StatusChanged();
                }
            }
            catch
                {
```

```
                iStatus = -1;

                StatusChanged();

            }



        }



        private void btnCancel_Click(object sender, EventArgs e)

            {

            this.iStatus = 0;

            StatusChanged();

        }

    }



    //逻辑控制类，相当于主控部分
    public class ProgressManager

        {
        //实际进行运算处理的类实例

        private IRunAble poRun;
        //进度条类实例

        private ProgressForm poProgress;



        public delegate void CompletedHandler();

        public event CompletedHandler ProgressCompleted;
```

```
            public int GetStatus
                {
                    get { return poProgress.GetStatus; }
            }


            public ProgressManager(IRunAble RunAbleInstance)
                {
                this.poRun = RunAbleInstance;
                poProgress = new ProgressForm();
                this.poProgress.StartPosition = FormStartPosition.Cen
terScreen;
                this.poProgress.Show();
                poProgress.StatusChanged += new ProgressForm.StatusHa
ndler(poProgress_StatusChanged);
            }
            public void Start()
                {
                //异步调用，进行运算。
                AsyncDelegate oDelegate = new AsyncDelegate(DoFunctio
n);
                oDelegate.BeginInvoke(myCallBack, new object());
            }


            private delegate void AsyncDelegate();
```

```
        private void DoFunction()
            {
            this.poRun.Run(this);
        }

        //回调函数
        private void myCallBack(IAsyncResult iResult)
            {
            this.poProgress.Close();
        }

        public void PerformStep()
            {
            this.poProgress.PerformStep();
        }

        void poProgress_StatusChanged()
            {
            if (poProgress.GetStatus == 0 || poProgress.GetStatus
== 1)
                {
                poProgress.Close();
```

```
            }

            ProgressCompleted();

        }

    }



    //实际运算类接口
    public interface IRunAble

        {

        void Run(ProgressManager aoManager);

    }



    //实际运算外理类
    public class CalRun : IRunAble

        {

        public void Run(ProgressManager aoManager)

            {

            double p = 3.1415926;

            for (int i = 1; i <= 100000; i++)

                {

                if (aoManager.GetStatus == 0)

                    {

                    break;

                }
```

```
                    System.Windows.Forms.Application.DoEvents();

                    double dd = i * i * p;

                    double dd2 = Math.Abs(dd * 456) * Math.Abs(dd * 4
56);

                    double dd3 = Math.Pow(dd, dd2) * Math.Pow(dd, dd2
);



                    if (i % 1000 == 0)

                        {

                        aoManager.PerformStep();

                    }

                }

            }

        }



    }
```

调用方式如下：

```
    private CalRun cr;

            private ProgressManager pm;



            private void button1_Click(object sender, EventArgs e)

                {

                cr = new CalRun();
```

```
                pm = new ProgressManager(cr);

                pm.ProgressCompleted += new ProgressManager.Completed
Handler(pm_ProgressCompleted);

                pm.Start();

            }



            void pm_ProgressCompleted()

                {

                switch (pm.GetStatus)

                    {

                    case 1:

                        MessageBox.Show("OK");

                        break;

                    case 0:

                        MessageBox.Show("Cancel");

                        break;

                    case -1:

                        MessageBox.Show("Error");

                        break;

                }

            }
```

英文原版地址：http://www.sellsbrothers.com/writing/default.aspx?content=delegates.htm

.NET 委托：一个 C#睡前故事

英文版原作者：Chris Sells（http://www.sellsbrothers.com/）

翻译：袁晓辉（http://www.farproc.com/ http://dev.csdn.net/uoyevoli）

紧耦合

从前，在南方一块奇异的土地上，有个工人名叫彼得，他非常勤奋，对他的老板总是百依百顺。但是他的老板是个吝啬的人，从不信任别人，坚决要求随时知道彼得的工作进度，以防止他偷懒。但是彼得又不想让老板呆在他的办公室里站在背后盯着他，于是就对老板做出承诺：无论何时，只要我的工作取得了一点进展我都会及时让你知道。彼得通过周期性地使用“带类型的引用”(原文为：“typed reference” 也就是 delegate？？)“回调”他的老板来实现他的承诺，如下：

```
class Worker {
    public void Advise(Boss boss) { _boss = boss; }
    public void DoWork() {
        Console.WriteLine(“工作: 工作开始”);
        if( _boss != null ) _boss.WorkStarted();

        Console.WriteLine(“工作: 工作进行中”);
        if( _boss != null ) _boss.WorkProgressing();

        Console.WriteLine(”“工作: 工作完成””);
        if( _boss != null ) {
            int grade = _boss.WorkCompleted();
            Console.WriteLine(“工人的工作得分＝” + grade);
        }
    }
    private Boss _boss;
}

class Boss {
    public void WorkStarted() { /* 老板不关心。 */ }
    public void WorkProgressing() { /*老板不关心。 */ }
    public int WorkCompleted() {
        Console.WriteLine(“时间差不多！”);
        return 2; /* 总分为 10 */
    }
}
```

```
class Universe {
    static void Main() {
        Worker  peter = new Worker();
        Boss        boss = new Boss();
        peter.Advise(boss);
        peter.DoWork();

        Console.WriteLine(“Main: 工人工作完成”);
        Console.ReadLine();
    }
}
```

接口

现在，彼得成了一个特殊的人，他不但能容忍吝啬的老板，而且和他周围的**宇宙**也有了密切的联系，以至于他认为**宇宙**对他的工作进度也感兴趣。不幸的是，他必须也给**宇宙**添加一个特殊的回调函数 Advise 来实现同时向他老板和**宇宙**报告工作进度。彼得想要把潜在的通知的列表和这些通知的实现方法分离开来，于是他决定把方法分离为一个接口：

```
interface IWorkerEvents {
    void WorkStarted();
    void WorkProgressing();
    int WorkCompleted();
}

class Worker {
    public void Advise(IWorkerEvents events) { _events = events; }
    public void DoWork() {
        Console.WriteLine(“工作: 工作开始”);
        if( _events != null ) _events.WorkStarted();

        Console.WriteLine(“工作: 工作进行中”);
        if(_events != null ) _events.WorkProgressing();

        Console.WriteLine(″“工作: 工作完成”″);
        if(_events != null ) {
            int grade = _events.WorkCompleted();

            Console.WriteLine(“工人的工作得分＝” + grade);
        }
    }
    private IWorkerEvents _events;
}
```

```
class Boss : IWorkerEvents {
    public void WorkStarted() { /* 老板不关心。 */ }
    public void WorkProgressing() { /* 老板不关心。 */ }
    public int WorkCompleted() {
        Console.WriteLine(“时间差不多！”);
        return 3; /* 总分为 10 */
    }
}
```

委托

不幸的是，每当彼得忙于通过接口的实现和老板交流时，就没有机会及时通知**宇宙**了。至少他应该忽略身在远方的老板的引用，好让其他实现了 IWorkerEvents 的对象得到他的工作报告。（”At least he’d abstracted the reference of his boss far away from him so that others who implemented the IWorkerEvents interface could be notified of his work progress” 原话如此，不理解到底是什么意思:)）

他的老板还是抱怨得很厉害。“彼得！”他老板吼道，“你为什么在工作一开始和工作进行中都来烦我？！我不关心这些事件。你不但强迫我实现了这些方法，而且还在浪费我宝贵的工作时间来处理你的事件，特别是当我外出的时候更是如此！你能不能不再来烦我？”

于是，彼得意识到接口虽然在很多情况都很有用，但是当用作事件时，“粒度”不够好。他希望能够仅在别人想要时才通知他们，于是他决定把接口的方法分离为单独的委托，每个委托都像一个小的接口方法：

```
delegate void WorkStarted();
delegate void WorkProgressing();
delegate int WorkCompleted();

class Worker {
    public void DoWork() {
        Console.WriteLine(“工作: 工作开始”);
        if( started != null ) started();

        Console.WriteLine(“工作: 工作进行中”);
        if( progressing != null ) progressing();

        Console.WriteLine(”“工作: 工作完成””);
        if( completed != null ) {
            int grade = completed();
            Console.WriteLine(“工人的工作得分＝” + grade);
        }
    }
```

```
    public WorkStarted started;
    public WorkProgressing progressing;
    public WorkCompleted completed;
}

class Boss {
    public int WorkCompleted() {
        Console.WriteLine("Better...");
        return 4; /* 总分为 10 */
    }
}

class Universe {
    static void Main() {
        Worker  peter = new Worker();
        Boss        boss = new Boss();
        peter.completed = new WorkCompleted(boss.WorkCompleted);
        peter.DoWork();

        Console.WriteLine(“Main: 工人工作完成”);
        Console.ReadLine();
    }
}
```

静态监听者

这样，彼得不会再拿他老板不想要的事件来烦他老板了，但是他还没有把**宇宙**放到他的监听者列表中。因为**宇宙**是个包涵一切的实体，看来不适合使用实例方法的委托（想像一下，实例化一个“**宇宙**”要花费多少资源…..），于是彼得就需要能够对静态委托进行挂钩，委托对这一点支持得很好：

```
class Universe {
    static void WorkerStartedWork() {
        Console.WriteLine("Universe notices worker starting work");
    }

    static int WorkerCompletedWork() {
        Console.WriteLine("Universe pleased with worker's work");
        return 7;
    }

    static void Main() {
        Worker  peter = new Worker();
        Boss        boss = new Boss();
        peter.completed = new WorkCompleted(boss.WorkCompleted);
```

```
        peter.started = new WorkStarted(Universe.WorkerStartedWork);
        peter.completed = new WorkCompleted(Universe.WorkerCompletedWork);
        peter.DoWork();

        Console.WriteLine(“Main: 工人工作完成”);
        Console.ReadLine();
    }
}
```

事件

不幸的是，**宇宙**太忙了，也不习惯时刻关注它里面的个体，它可以用自己的委托替换了彼得老板的委托。这是把彼得的 Worker 类的的委托字段做成 public 的一个无意识的副作用。同样，如果彼得的老板不耐烦了，也可以决定自己来激发彼得的委托（真是一个粗鲁的老板）：

```
         // Peter’s boss taking matters into his own hands
        if( peter.completed != null ) peter.completed();
```

彼得不想让这些事发生，他意识到需要给每个委托提供“注册”和“反注册”功能，这样监听者就可以自己添加和移除委托，但同时又不能清空整个列表也不能随意激发彼得的事件了。彼得并没有来自己实现这些功能，相反，他使用了 event 关键字让 C#编译器为他构建这些方法：

```
class Worker {
...
    public event WorkStarted started;
    public event WorkProgressing progressing;
    public event WorkCompleted completed;
}
```

彼得知道 event 关键字在委托的外边包装了一个 property，仅让 C#客户通过+= 和 -=操作符来添加和移除，强迫他的老板和**宇宙**正确地使用事件。

```
static void Main() {
        Worker  peter = new Worker();
        Boss        boss = new Boss();
        peter.completed += new WorkCompleted(boss.WorkCompleted);
        peter.started += new WorkStarted(Universe.WorkerStartedWork);
        peter.completed += new WorkCompleted(Universe.WorkerCompletedWork);
        peter.DoWork();

        Console.WriteLine(“Main: 工人工作完成”);
        Console.ReadLine();
    }
```

“收获”所有结果

到这时，彼得终于可以送一口气了，他成功地满足了所有监听者的需求，同时避免了与特定实现的紧耦合。但是他注意到他的老板和**宇宙**都为它的工作打了分，但是他仅仅接收了一个分数。面对多个监听者，他想要“收获”所有的结果，于是他深入到代理里面，轮询监听者列表，手工一个个调用：

```
public void DoWork() {
        ...
        Console.WriteLine("“工作: 工作完成”");
        if( completed != null ) {
            foreach( WorkCompleted wc in completed.GetInvocationList() ) {
                int grade = wc();
                Console.WriteLine(“工人的工作得分＝” + grade);
            }
        }
    }
```

异步通知：激发 & 忘掉

同时，他的老板和**宇宙**还要忙于处理其他事情，也就是说他们给彼得打分所花费的事件变得非常长：

```
class Boss {
    public int WorkCompleted() {
        System.Threading.Thread.Sleep(3000);
        Console.WriteLine("Better..."); return 6; /* 总分为 10 */
    }
}

class Universe {
    static int WorkerCompletedWork() {
        System.Threading.Thread.Sleep(4000);
        Console.WriteLine("Universe is pleased with worker's work");
        return 7;
    }
    ...
}
```

很不幸，彼得每次通知一个监听者后必须等待它给自己打分，现在这些通知花费了他太多的工作事件。于是他决定忘掉分数，仅仅异步激发事件：

```
public void DoWork() {
        ...
        Console.WriteLine("“工作: 工作完成”");
```

```
        if( completed != null ) {
            foreach( WorkCompleted wc in completed.GetInvocationList() )
            {
                wc.BeginInvoke(null, null);
            }
        }
    }
```

异步通知：轮询

这使得彼得可以通知他的监听者，然后立即返回工作，让进程的线程池来调用这些代理。随着时间的过去，彼得发现他丢失了他工作的反馈，他知道听取别人的赞扬和努力工作一样重要，于是他异步激发事件，但是周期性地轮询，取得可用的分数。

```
public void DoWork() {
        ...
        Console.WriteLine("“工作: 工作完成”");
        if( completed != null ) {
            foreach( WorkCompleted wc in completed.GetInvocationList() ) {
                IAsyncResult res = wc.BeginInvoke(null, null);
                while( !res.IsCompleted ) System.Threading.Thread.Sleep(1);
                int grade = wc.EndInvoke(res);
                Console.WriteLine(“工人的工作得分＝” + grade);
            }
        }
    }
```

异步通知：委托

不幸地，彼得有回到了一开始就想避免的情况中来，比如，老板站在背后盯着他工作。于是，他决定使用自己的委托作为他调用的异步委托完成的通知，让他自己立即回到工作，但是仍可以在别人给他的工作打分后得到通知：

```
public void DoWork() {
        ...
        Console.WriteLine("“工作: 工作完成”");
        if( completed != null ) {
            foreach( WorkCompleted wc in completed.GetInvocationList() ) {
                wc.BeginInvoke(new AsyncCallback(WorkGraded), wc);
            }
        }
    }

    private void WorkGraded(IAsyncResult res) {
```

```
        WorkCompleted wc = (WorkCompleted)res.AsyncState;
        int grade = wc.EndInvoke(res);
        Console.WriteLine("工人的工作得分＝" + grade);
    }
```

宇宙中的幸福

彼得、他的老板和**宇宙**最终都满足了。彼得的老板和**宇宙**可以收到他们感兴趣的事件通知，减少了实现的负担和非必需的往返“差旅费”。彼得可以通知他们，而不管他们要花多长时间来从目的方法中返回，同时又可以异步地得到他的结果。彼得知道，这并不***十分***简单，因为当他异步激发事件时，方法要在另外一个线程中执行，彼得的目的方法完成的通知也是一样的道理。但是，迈克和 彼得是好朋友，他很熟悉线程的事情，可以在这个领域提供指导。

他们永远幸福地生活下去……<完>

2005 年 9 月 2 日

作者Blog：http://blog.csdn.net/uoyevoli/

衔接UI线程和管理后台工作线程的类（多线程、异步调用）

一、引言

在编写Windows form时，如果直接在UI线程要运行一个费时方法的话（如从数据库查询大量数据时），会引起程序“假死”，从而导致用户不满。这个时候就需要通过多线程技术来解决，提高界面交互性能，方便用户使用。

一般通过三种方式解决：

1.通过System.Threading.Thread类，创建新的线程，Thread.Start运行费时方法。

2.通过System.Threading.ThreadPool类，将费时任务提交到线程池中，等待运行。

以上两种方法，基本思路是在UI界面中控制线程的启动和中止，在线程中回调用UI界面方法，更新界面。在线程中回调UI界面方法时，特别是涉及更新控件属性时，如果不注意，存在很大的隐患。这两种办法，编码和控制结构较为复杂，需要启动和管理额外的线程占用资源。

3.通过异步委托调用，将该方法排队到系统线程池的线程中运行，而在费时方法中也通过Control.BeginInvoke异步回调，达到"启动后不管"的目的。

这种方法，编码简单，程序结构较为清晰，充分利用.NET框架的异步委托功能，但要对异步调用知识较熟悉。

相关知识点参见

现利用.NET异步委托调用功能，编写Task抽象类，以方便管理后台工作线程，衔接后台线程与UI线程的联系。该抽象类提供了调用和管理的框架，没有方法的实现细节，通过继承类、重写方法，可以实现想要的功能。主要功能如下：

1.利用异步委托调用，实际多线程，不需要单独后台线程。

2.通过委托、事件驱动，实际后台与前台UI线程的联系，实现事件广播。

3.支持正常取消后台工作方法（费时方法）运行，也可以强制中止线程。

4.能够捕获取消、强制中止和方法出错三种情况，并突发相关事件，以便进行释放资源等操作。

5.通过异步调用，在工作方法中安全调用涉及UI控件的方法。

6.自行管理工作进程状态，提供状态变化事件。

7.只要工作方法调用签名，符合定义的TaskDelegate委托接口，可通过StartTask(TaskDelegate worker ,params object[] args)方便调用。在实际使用时，可在继承类中定义多个相同调用接口的方法，避免重复编码，较为方便。

给大家作个参考，而大牛呢，多点指正。当是扔个砖头，想砸块玉吧。

二、代码

```
1   using System;
2   using System.Windows.Forms;
3
4   namespace Net66.AsynchThread
5   {
6       /// <summary>
7       /// 任务工作状态
8       /// </summary>
9       public enum TaskStatus
10      {...
37
38      /// <summary>
39      /// 任务状态消息
40      /// </summary>
41      public class TaskEventArgs : EventArgs
42      {...
137
138     /// <summary>
139     /// 任务的工作方法(Work)的委托接口
140     /// 传入值:对象数组(object[])
141     /// 返回值:对象(object)
142     /// </summary>
143     public delegate object TaskDelegate( params object[] args );
144
145     /// <summary>
```

```
146     /// 任务事件的委托接口
147     /// </summary>
148     public delegate void TaskEventHandler( object sender, TaskEventArgs e );
149
150     abstract public class Task
151     {
152         内部属性
178
179         事件
201
202         属性
267
268         触发事件
358
359         工作进程管理
497
498         工作方法的基础
520     }
521 }
522
523 使用 Task 类/*
525
526 使用 Task 类
527
528 一.在 UI 线程中创建 Task 类
529
530 Task 类负责管理后台线程。要使用 Task 类，必须做的事情就是创建一个 Task 对象，注册它激发的事件，并且实现这些事件的处理。因为事件是在 UI 线程上激发的，所以您根本不必担心代码中的线程处理问题。
531
```

下面的示例展示了如何创建 Task 对象。现假设 UI 有两个按钮，一个用于启动运算，一个用于停止运算，还有一个进度栏显示当前的计算进度。

```
// 创建任务管理对象
_Task = new Task();
// 挂接任务管理对象工作状态变化事件
_Task.TaskStatusChanged += new TaskEventHandler( OnTaskStatusChanged );
// 挂接任务管理对象工作进度变化事件
_Task.TaskProgressChanged += new TaskEventHandler( OnTaskProgressChanged );
```

(1)

用于计算状态和计算进度事件的事件处理程序相应地更新 UI，例如通过更新状态栏控件。

```
private void OnTaskProgressChanged( object sender,TaskEventArgs e )
{
    _progressBar.Value = e.Progress;
}
```

(2)

下面的代码展示的 TaskStatusChanged 事件处理程序更新进度栏的值以反映当前的计算进度。假定进度栏的最小值和最大值已经初始化。

```
private void OnTaskStatusChanged( object sender, TaskEventArgs e )
{
    switch ( e.Status )
    {
        case TaskStatus.Running:
            button1.Enabled = false;
            button2.Enabled = true;
            break;
        case TaskStatus.Stop:
```

```
            button1.Enabled = true;
            button2.Enabled = false;
            break;
        case TaskStatus.CancelPending:
            button1.Enabled = false;
            button2.Enabled = false;
            break;
    }
}

在这个示例中，TaskStatusChanged 事件处理程序根据计算状态启用和禁用启动和停止按钮。这可以防止用户尝试启动一个已经在进行的计算，并且向用户提供有关计算状态的反馈。

通过使用 Task 对象中的公共方法，UI 为每个按钮单击实现了窗体事件处理程序，以便启动和停止计算。例如，启动按钮事件处理程序调用 StartTask 方法，如下所示。

private void startButton_Click( object sender, System.EventArgs e )
{
    _Task.StartTask( new object[] {} );
}

类似地，停止计算按钮通过调用 StopTask 方法来停止计算，如下所示。

private void stopButton_Click( object sender, System.EventArgs e )
{
    _Task.StopTask();
}

二.可能在非 UI 线程中使用 Task 类时
(1)和(2)应作如下改变
```

(1)

用于计算状态和计算进度事件的事件处理程序相应地更新 UI，例如通过更新状态栏控件。

```
private void OnTaskProgressChanged( object sender,TaskEventArgs e )
{
    if (InvokeRequired )        //不在 UI 线程上,异步调用
    {
        TaskEventHandler TPChanged = new TaskEventHandler( OnTaskProgressChanged );
        this.BeginInvoke(TPChanged,new object[] {sender,e});
    }
    else                        //更新
    {
        _progressBar.Value = e.Progress;
    }
}
```

(2)

下面的代码展示的 TaskStatusChanged 事件处理程序更新进度栏的值以反映当前的计算进度。假定进度栏的最小值和最大值已经初始化。

```
private void OnTaskStatusChanged( object sender, TaskEventArgs e )
{
    if (InvokeRequired )        //不在 UI 线程上,异步调用
    {
        TaskEventHandler TSChanged = new TaskEventHandler( OnTaskStatusChanged );
        this.BeginInvoke(TSChanged,new object[] {sender,e});
    }
    else                        //更新
    {
        switch ( e.Status )
```

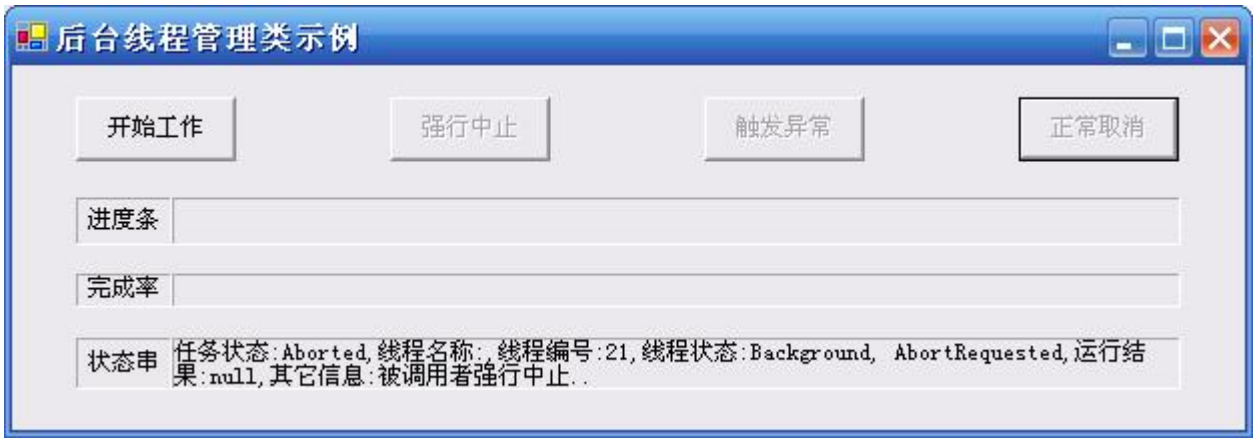
```
        {
            case TaskStatus.Running:
                button1.Enabled = false;
                button2.Enabled = true;
                break;
            case TaskStatus.Stop:
                button1.Enabled = true;
                button2.Enabled = false;
                break;
            case TaskStatus.CancelPending:
                button1.Enabled = false;
                button2.Enabled = false;
                break;
        }
    }
}

*/
```

三、示例

1.启动时的UI界面

2.后台工作方法(费用方法)运行后,任务状态为Running

3.强制中止工作方法，运行任务状态Aborted



4.工作方法突发错误时，任务状态ThrowErrorStoped

5.工作方法正常结束或正常取消而结束时，任务状态Stopped

示例代码下载

posted on 2005-08-03 00:19 Net66 阅读(4810) 评论(25) 编辑 收藏 所属分类: C#

.

评论

#4 楼 2005-08-03 08:46 Nineteen@newsmth [未注册用户]

System.Threading.ThreadPool 还是不要用的好，这东西牵扯进程的异步调用，这东西微软就该设置成 internal

回复 引用 查看

#5 楼 2005-08-03 08:49 学习 [未注册用户]

http://www.vckbase.com/document/viewdoc/?id=1126

回复 引用 查看

#6 楼 2005-08-03 08:54 James

这些 BeginXXXX 的方法都是使用了 ThreadPool 的。

回复 引用 查看

#7 楼 2005-08-03 09:06 张老三 [未注册用户]

刚完成了一个图库管理系统，使用的就是这种方式．感觉还不错．

回复 引用 查看

#8 楼 2005-08-03 09:08 win [未注册用户]

文章的意思应该是不用显式使用 ThreadPool,通过系统默认调用吧

回复 引用 查看

#10 楼 2005-08-03 09:39 妖居

异步委托也是刚刚用过，做一个分析目录的工具的时候用的。

回复 引用 查看

#11 楼 2005-08-03 09:41 James

我的意思是说，这些异步委托调用什么的，其内部都是使用了 ThreadPool 的，这样当然不用你自己去直接使用 ThreadPool 了。

回复 引用 查看

#12 楼 2005-08-03 18:48 yinh

最近写的项目中因为用得很多网络交互，频繁的异步调用，但对控制这些线程却没有一点主意，希望能从你的文章中得到一些启示。

回复 引用 查看

#14 楼 2005-09-14 16:10 双鱼座

看了下楼主的代码，大致上不错。不过在封装性方面做得不够...其实封装性是需要早一些构思的，比功能的实现还要再早一些。

先搞清楚你的 Task 的派生类依赖什么，也就是说可供派生类调用的方法，例如 Fire***方法；接下来搞清楚你的 Task 的派生类不依赖什么，例如必须在重写方法中调用 base.Work 这样的问题；最后是参数的传递问题。由此我想到了在 Delphi3 中对 Thread 的封装，有一个同步执行方法的方法，在调用者不知道任何细节的情况下可以进行安全的调用。当然，那个封装代价有点大了。

我作了如下修改：

1.定义一个抽象方法：

protected abstract object Execute(params object [] args);

这个方法才是真正需要继承的方法。

2.基类的 Work 方法改成私有方法(Work 这个名字取得不是太好)，代码这样写：

```
System.Threading.Thread.CurrentThread.IsBackground = true;
_workThread = System.Threading.Thread.CurrentThread;
return Execute(args);
```

3.加一个线程方法：

```
void start()
{
StartTask(new TaskDelegate(Work), tempArgs);
}
```

4.定义一个私有字段用来向线程传递参数：

```
private object[] tempArgs;
```

5.最后加一个启动方法：

```
public void Start(params object[] args)
{
tempArgs = args;
Thread thread = new Thread(new ThreadStart(start));
thread.Start();
}
```

改造完成了。客户端代码由此变得非常简洁：

1.当按下"开始执行"时的代码：

```
_Task.Start(new object[] {});
```

2.Concrete 类型 newasynchui 的代码也简单了，只需要重写一个 Execute 方法而不是重载一个 Work 和另外写一个 Work2，在方法中也不必调用 base.Work，你想调用也调用不了，因为是那是私有方法。

当然还有其它一些更好的建议，例如不必定义那么多的事件，其实一两个就足够了，也不需要派生新的 EventArgs 类型，因为从 sender 中可以获得所有的信息。

我的解决方案可能更邪一点了，因为我是用反射和 Emit 实现的，原理和你的差不多，只不过客户端代码可以非常简单而已。

回复 引用 查看

#20 楼 2006-10-07 16:37 可乐[匿名] [未注册用户]

看了例程非常激动，也许是还有很多东西要学没有看的很彻底，希望楼主能解释，就是在 UI 中启动一个辅助线程后往往会应为需要在 UI 启动的这个辅助线程中在启动其他线程，这个时候该怎样使用 Task 抽象类，该如何将辅助线程启动的其他线程中的信息显示在 UI 中，还有怎么才能利用 AsynchThread 在 UI 中停止辅助线程的时候能同时停止辅助线程启动的其他线程，希望楼主能解答，谢谢!!

回复 引用 查看

在.NET 客户端程序中使用多线程

admin 发表于：04-06-08

通常认为在编写程序中用到多线程是一个高级的编程任务，容易发生错误。在本月的栏目中，我将在一个 Windows 窗体应用程序中使用多线程，它具有实际的意义，同时尽量使事情简单。我的目标是在一个普通的需求描述中用最好的办法讲解多线程；客户仍然比较喜欢使用户交互方式的应用程序。

多线程通常和服务器端软件，可扩展性及性能技术联系在一起。 然而，在微软.NET 框架中，许多服务器端应用程序都驻留在 ASP.NET 体系结构中。同样，这些应用程序在逻辑上是单线程的， 因为 IIS 和 ASP.NET 在 ASP.NET Web Form 或 Web 服务程序中执行了许多或所有的多线程。 在 ASP.NET 应用程序中你一般可以忽略线程性。 这就是为什么在.NET 框架中，多线程更倾向于在客户端使用的一个原因，比如在保证同用户交互的同时而执行一个很长的操作。

线程背景

线程执行代码。它们由操作系统实现，是 CPU 本身的一种抽象。许多系统都只有一个 CPU， 线程是把 CPU 快速的处理能力分开而执行多个操作的一种方法，使它们看起来好像同步似的。即使一个系统由多个 CPU， 但运行的线程一般要比处理器多。

在一个 Windows 为基础的应用程序中，每一个进程至少要有一个线程，它能够执行机器语言指令。 一旦一个进程的所有线程都中止了，进程本身和它所占用的资源将会被 Windows 清除。

许多应用程序都被设计为单线程程序，这意味着该程序实现的进程从来不会有超过一个线程在执行，即使在系统中有多个同样的处理在进行。一般一个进程不会关心系统中其他进程的线程的执行。

然而，在单个进程里的所有线程不仅共享虚拟地址空间，而且许多进程级的资源也被共享, 比如文件和窗口句柄等。由于进程资源共享的特征，一个线程必须考虑同一进程中其它线程正在做什么。线程同步是在多线程的进程中保持各线程互不冲突的一门艺术。这也使得多线程比较困难。

最好的方式是只有在需要时才使用多线程，尽量保持事情简单。而且要避免线程同步的情况。在本栏目中，我将向你展示如何为一个普通的客户应用程序做这些事情。

为什么使用多个线程？

已经有许多单线程的客户端应用程序，而且每天还有许多正在被写。在许多情况下，单线程的行为已经足够了。

然而，在某些特定的应用程序中加入一些异步行为可以提高你的经验。典型的数据库前端程序是一个很好的例子。

数据库查询需要花费大量时间完成。在一个单线程的应用程序里，这些查询会导致 window 消息处理能力阻塞，导致程序的用户交互被冻结。解决办法就是，这个我将要详细描述，用一个线程处理来自操作系统的消息，而另外一个线程做一个很长的工作。在你的代码中使用第二个线程的重

要原因就是即使在幕后有一个繁忙的工作在进行，也要保证你的程序的用户交互有响应。

我们首先看一下执行一长串操作的单线程的 GUI 程序。然后我们将用额外的线程整理该程序。

Figure 1 是用 C＃写的一个程序的完整源代码。它创建了一个带有文本框和按钮的窗体。如果你在文本框中键入了一个数字，然后按下按钮，这个程序将处理你输入的那个数字，它表示秒数，每秒钟响铃一次代表后台的处理。除了 Figure 1 的代码外，你可以从本文开头的链接中下载完整的代码。下载或键入 Figure 1 所示的代码，在读之前编译运行它，（编译前，在 Visual Studio.NET 中右击你的工程，加入 Microsoft Visual Basic 运行时引用）当你试着运行 Figure 1 中的
SingleThreadedForm.cs 应用程序时，你马上就会看到几个问题。

Figure 1 SingleThreadedForm.cs

```
using System;
using System.Drawing;
using System.Threading;
using System.Windows.Forms;
using Microsoft.VisualBasic;

class App {

  // Application entry point
  public static void Main() {

    // Run a Windows Forms message loop
    Application.Run(new SingleThreadedForm());
  }
}

// A Form-derived type
class SingleThreadedForm : Form {

  // Constructor method
  public SingleThreadedForm() {

    // Create a text box
    text.Location = new Point(10, 10);
    text.Size = new Size(50, 20);
    Controls.Add(text);

    // Create a button
    button.Text = "Beep";
    button.Size = new Size(50, 20);
    button.Location = new Point(80, 10);

    // Register Click event handler
```

```
    button.Click += new EventHandler(OnClick);

    Controls.Add(button);
  }

  // Method called by the button's Click event
  void OnClick(Object sender, EventArgs args) {
    // Get an int from a string
    Int32 count = 0;
    try { count = Int32.Parse(text.Text); } catch (FormatException) {}

    // Count to that number
    Count(count);
  }

  // Method beeps once per second
  void Count(Int32 seconds) {

    for (Int32 index = 0; index < seconds; index++) {
      Interaction.Beep();
      Thread.Sleep(1000);
    }
  }

  // Some private fields by which to reference controls
  Button button = new Button();
  TextBox text = new TextBox();
}
```

在你第一次测试运行时，在文本框中输入 20，按下按钮。你将看到程序的用户交互变得完全没有响应了。你不能单击按钮或者编辑文本框，程序也不能被从容的关闭，如果你覆盖该窗体接着会显示一个窗口的部分区域，它将不再重绘自己（见 Figure 2），这个程序被锁定足足 20 秒， 然而它还可以继续响铃，证明它还没有真正的死掉。这个简单的程序解释了单线程 GUI 程序的问题。

我将用多线程解决第一个问题：未响应的用户交互，但首先我将解释是什么导致了这种现象。

线程和 Windows 用户界面

Windows Forms 类库建立在大家所熟知的 User32 Win32 API 基础上。User32 实现了 GUI 的基本元素，例如窗体，菜单及按钮之类等。所有由 User32 实现的窗体和控件都使用了事件驱动型结构。

这里简单的讲讲它们如何工作。发生在窗体上的事情，例如鼠标单击，坐标变化，大小变化和重绘请求，都称作事件。在 User32 API 模型中的事件是由窗体消息表示的。每一个窗体有一个函数，叫做窗口过程或 WndProc，它由应用程序实现。WndProc 为窗体负责处理窗体消息。

但是 WndProc 不是神奇的被系统调用。相反，应用程序必须调用 GetMessage 主动地从系统中得到窗体消息。该消息被应用程序调用 DispatchMethod API 方法分配到它们的目标窗体的 WndProc 方法中。应用程序只是简单的循环接收和分配窗口消息，一般叫做消息泵或消息循环。线程拥有所有窗体，这样它就可以提取消息，WndProc 函数也被同样的线程所调用。

现在回到 Windows Forms 类来。Windows Forms 在应用程序中对 User32 的消息结构进行了大约 95%的抽象。代替了 WndProc 函数，Windows Forms 程序定义了事件处理器和虚拟函数重载来处理与窗体（窗口）或控件有关的不同系统事件。然而消息提取必须要运行，它在 Windows Forms API 的 Application.Run 方法里面实现。

Figure 1 所示的代码似乎仅仅调用了 Application.Run 接着就退出了。 然而这缺少了透明性：应用程序的主线程在其生命周期里只对 Application.Run 进行一次调用进行消息提取，其结果却为用应用程序其它部分创造了不同事件处理器的调用。当窗体上的按钮被单击时，在 Figure 1 中的 OnClick 方法被主线程调用，该线程同样要负责在 Application.Run 中提取消息。

这解释了为什么在一个长操作发生时，用户交互没有响应。如果在一个事件处理器中一个很长的操作 （如数据库查询）发生了，那么主线程就被占用，它又需要不断提取消息。没有能力提取消息并发送到窗口或窗体上， 就没有能力响应调整大小，重绘自己，处理单击或响应用户的任何交互。

在接下来的部分为了执行长操作我将使用公共语言运行时的线程池来修改 Figure 1 所示的例子代码，这样主线程仍然可以提取消息。

托管线程池

CLR 为每一个托管进程维护了一个线程池，这意味着当你的应用程序主线程需要进行某些异步处理时，你可以很容易的从线程池中借助某个线程实现特定的处理。一旦处理工作完成，线程被归还到线程池以便以后使用。让我们看一个例子，修改使用线程池。

注意 Figure 3 中 FlawMultiThreadForm.cs 中红色部分表示的行；它们是由 Figure 1 中的单线程变为多线程程序 时唯一要修改的代码。如果你编译 Figure 3 所示的代码，并设置运行 20 秒，你将看到当处理 20 个响铃的请求时，仍然能够响应用户的交互。在客户端程序中使用多线程来响应用户交互是一个吸引人的原因。

Figure 3 FlawedMultiThreadedForm.cs

```
using System;
using System.Drawing;
using System.Threading;
using System.Windows.Forms;
using Microsoft.VisualBasic;

class App {

  // Application entry point
  public static void Main() {

  // Run a Windows Forms message loop
```

```
  Application.Run(new FlawedMultiThreadedForm());
  }
}

// A Form-derived type
class FlawedMultiThreadedForm : Form {

  // Constructor method
  public FlawedMultiThreadedForm() {

    // Create a text box
    text.Location = new Point(10, 10);
    text.Size = new Size(50, 20);
    Controls.Add(text);

    // Create a button
    button.Text = "Beep";
    button.Size = new Size(50, 20);
    button.Location = new Point(80, 10);

    // Register Click event handler
    button.Click += new EventHandler(OnClick);

    Controls.Add(button);
  }

  // Method called by the button's Click event
  void OnClick(Object sender, EventArgs args) {
    // Get an int from a string
    Int32 count = 0;
    try { count = Int32.Parse(text.Text); } catch (FormatException) {}

      // Count to that number
      WaitCallback async = new WaitCallback(Count);
      ThreadPool.QueueUserWorkItem(async, count);
    }

    // Async method beeps once per second
    void Count(Object param) {

      Int32 seconds = (Int32) param;
      for (Int32 index = 0; index < seconds; index++) {
        Interaction.Beep();
        Thread.Sleep(1000);
```

```
    }
  }

  // Some private fields by which to reference controls
  Button button = new Button();
  TextBox text = new TextBox();
}
```

然而，在 Figure 3 中所做的变化，却引入了一个新问题（如 Figure 3 的名字一样）；现在用户可以启动多个同时响铃的长操作。在许多实时应用中这会导致线程间的冲突。为了修正这个线程同步请求，我将讲述这些，但首先熟悉一下 CLR''''s 线程池。

类库中的 System.Threading.ThreadPool 类提供了一个访问 CLR''''s 线程池的 API 接口， ThreadPool 类型不能被实例化，它由静态成员组成。ThreadPool 类型最重要的方法是对 ThreadPool.QueueUserWorkItem 的两个重载。这两种方法让你定义一个你愿意被线程池中的一个线程进行回调的函数。通过使用类库中的 WaitCallback 委托类型的一个实例来定义你的方法。一种重载让你对异步方法定义一个参数；这是 Figure 3 所使用的版本。

下面的两行代码创建一个委托实例，代表了一个 Count 方法，接下来的调用排队等候让线程池中的方法进行回调。

```
WaitCallback async = new WaitCallback(Count);
ThreadPool.QueueUserWorkItem(async, count);
```

ThreadPool.QueueUserWorkItem 的两个方法让你在队列中定义一个异步回调方法，然后立即返回。 同时线程池监视这个队列，接着出列方法，并使用线程池中的一个或多个线程调用该方法。这是 CLR''''s 线程池的主要用法。

CLR''''s 线程池也被系统的其它 APIs 所使用。例如， System.Threading.Timer 对象在定时间隔到来时将会在线程池中排队等候回调。ThreadPool.RegisterWaitForSingleObject 方法当响应内核系统同步对象有信号时会在线程池中排队等候调用。最后，回调由类库中的不同异步方法执行，这些异步方法又由 CLR''''s 线程池来执行。

一般来说，一个应用程序仅仅对于简单的异步操作需要使用多线程时毫无疑问应该使用线程池。相比较手工创建一个线程对象，这种方法是被推荐的。调用 ThreadPool.QueueUserWorkItem 执行简单，而且相对于重复的手动创建线程来说能够更好的利用系统资源。

最简单的线程同步

在本栏目开始我就称保持线程同步而不互相冲突是一门艺术。Figure 3 所示的 FlawedMultiThreadForm.cs 应用程序有一个问题：用户可以通过单击按钮引发一个很长的响铃操作，他们可以继续单击按钮而引发更多的响铃操作。如果不是响铃，该长操作是数据库查询或者在进程的内存中进行数据结构操作，你一定不想在同一时间内，有一个以上的线程做同样的工作。最好的情况下这是系统资源的一种浪费，最坏的情况下会导致数据毁灭。

最容易的解决办法就是禁止按钮一类的用户交互元素；两个进程间的通信稍微有点难度。过一会我将给你看如何做这些事情。但首先，让我指出所有线程同步使用的一些线程间通信的形式－从一个线程到另一个线程通信的一种手段。稍后我将讨论大家所熟知的 AutoResetEvent 对象类型，它仅用在线程间通信。

现在让我们首先看一下为 Figure 3 中 FlawedMultiThreadedForm.cs 程序中加入的线程同步代码。再一次的，Figure 4 CorrectMultiThreadedForm.cs 程序中红色部分表示的是其先前程序的较小的改动部分。 如果你运行这个程序你将看到当一个长响铃操作在进行时用户交互被禁止了（但没有挂起），响铃完成的时候又被允许了。这次这些代码的变化已经足够了，我将逐个运行他们。

Figure 4 CorrectMultiThreadedForm.cs

```
using System;
using System.Drawing;
using System.Threading;
using System.Windows.Forms;
using Microsoft.VisualBasic;

class App {

  // Application entry point
  public static void Main() {

    // Run a Windows Forms message loop
    Application.Run(new CorrectMultiThreadedForm());
  }
}

// A Form-derived type
class CorrectMultiThreadedForm : Form{

  // Constructor method
  public CorrectMultiThreadedForm() {

    // Create a textbox
    text.Location = new Point(10, 10);
    text.Size = new Size(50, 20);
    Controls.Add(text);

    // Create a button
    button.Text = "Beep";
    button.Size = new Size(50, 20);
    button.Location = new Point(80, 10);

    // Register Click event handler
    button.Click += new EventHandler(OnClick);

    Controls.Add(button);

    // Cache a delegate for repeated reuse
```

```
    enableControls = new BooleanCallback(EnableControls);
  }

  // Method called by the button's Click event
  void OnClick(Object sender, EventArgs args) {
    // Get an int from a string
    Int32 count = 0;
    try { count = Int32.Parse(text.Text); } catch (FormatException) {}

    // Count to that number
    EnableControls(false);
    WaitCallback async = new WaitCallback(Count);
    ThreadPool.QueueUserWorkItem(async, count);
  }

  // Async method beeps once per second
  void Count(Object param) {

    Int32 seconds = (Int32) param;
    for (Int32 index = 0; index < seconds; index++) {
      Interaction.Beep();
      Thread.Sleep(1000);
    }

    Invoke(enableControls, new Object[]{true});
  }

  void EnableControls(Boolean enable) {
    button.Enabled = enable;
    text.Enabled = enable;
  }

  // A delegate type and matching field
  delegate void BooleanCallback(Boolean enable);
  BooleanCallback enableControls;

  // Some private fields by which to reference controls
  Button button = new Button();
  TextBox text = new TextBox();
}
```

在 Figure 4 的末尾处有一个 EnableControls 的新方法，它允许或禁止窗体上的文本框和按钮控件。在 Figure 4 的开始我加入了一个 EnableControls 调用，在后台响铃操作排队等候之前立即禁止文本框和按钮。到这里线程的同步工作已经完成了一半，因为禁止了用户交互，所以用户不能引发更多的后台冲突操作。在 Figure 4 的末尾你将看到一个名为 BooleanCallback 的委托类型被定义，其签名是同 EnableControls 方法

兼容的。在那个定义之前，一个名为 EnableControls 的委托域被定义（见例子），它引用了该窗体的 EnableControls 方法。这个委托域在代码的开始处被分配。

你也将看到一个来自主线程的回调，该主线程为窗体和其控件拥有和提取消息。这个调用通过向 EnableControls 传递一个 true 参数来使能控件。这通过后台线程调用窗体的 Invoke 方法来完成，当其一旦完成其长响铃操时。代码传送的委托引用 EnableControls 去 Invoke，该方法的参数带有一个对象数组。Invoke 方法是线程间通信的一个非常灵活的方式，特别是对于 Windows Forms 类库中的窗口或窗体。在这个例子中，Invoke 被用来告诉主 GUI 线程通过调用 EnableControls 方法重新使能窗体上的控件。

Figure 4 中的 CorrectMultiThreadedForm.cs 的变化实现了我早先的建议－－当响铃操作在执行时你不想运行，就禁止引发响铃操作的用户交互部分。当操作完成时，告诉主线程重新使能被禁止的部分。对 Invoke 的调用是唯一的，这一点应该注意。

Invoke 方法在 System.Windows.Forms.Controls 类型中定义，包含 Form 类型让类库中的所有派生控件都可使用该方法。Invoke 方法的目的是配置了一个从任何线程对为窗体或控件实现消息提取线程的调用。

当访问控件派生类时，包括 Form 类，从提取控件消息的线程来看你必须这样做。这在单线程的应用程序中是很自然的事情。但是当你从线程池中使用多线程时，要避免从后台线程中调用用户交互对象的方法和属性是很重要的。相反，你必须使用控件的 Invoke 方法间接的访问它们。Invoke 是控件中很少见的一个可以安全的从任何线程中调用的方法，因为它是用 Win32 的 PostMessage API 实现的。

使用 Control.Invoke 方法进行线程间的通信有点复杂。但是一旦你熟悉了这个过程，你就有了在你的客户端程序中实现多线程目标的工具。本栏目的剩余部分将覆盖其它一些细节，但是 Figure 4 中的 CorrectMultiThreadedForm.cs 应用程序是一个完整的解决办法：当执行任意长的操作时仍然能够响应用户的其它操作。尽管大多数的用户交互被禁止，但用户仍然可以重新配置和调整窗口，也可以关闭程序。然而，用户不能任意使用程序的异步行为。这个小细节能够让你对你的程序保持自信心。

在我的第一个线程同步程序中，没有使用任何传统的线程结构，例如互斥或信号量，似乎一钱不值。然而，我却使用了禁止控件的最普通的方法。

细节－实现一个取消按钮

有时你想为你的用户提供一种取消长操作的方法。你所需要的就是你的主线程同后台线程之间的一些通信方法，通知后台线程操作不再被需要，可以停止。System.Threading 名字空间为这个方法提供了一个类：AutoResetEvent。

AutoResetEvent 是线程间通信的一种简单机制。一个 AutoResetEvent 对象可以有两种状态中的一个：有信号的和无信号的。当你创建一个 AutoResetEvent 实例时，你可以通过构造函数的参数来决定其初始状态。然后感知该对象的线程通过检查 AutoResetEvent 对象的状态，或者用 AutoResetEvent 对象的 Set 或 Reset 方法调整其状态，进行相互通信。

在某种程度上 AutoResetEvent 很像一个布尔类型，但是它提供的特征使其更适合于在线程间进行通信。这样的一个例子就是它有这种能力：一个线程可以有效的等待直到一个 AutoResetEvent 对象从一个无信号的状态变为有信号的状态。它是通过在该对象上调用 WaitOne 实现的。任何一个线程对一个无信号的 AutoResetEvent 对象调用了 WaitOne，就会被有效的阻塞直到其它线程使该对象有信号。使用布尔变量线程必须在一个循环中登记该变量，这是无效率的。一般来说没有必要使用 Reset 来使一个 AutoResetEvent 变为无信号，因为当其它线程感知到该对象为有信号时，它会被立即自动的设为无信号的。

现在你需要一种让你的后台线程无阻塞的测试 AutoResetEvent 对象的方法，你会有许多工具实现线程的取消。为了完成这些，调用带有 WaitOne

的重载窗体并指出一个零毫秒的超出时间，以零毫秒为超出时间的 WaitOne 会立即返回，而不管 AutoResetEvent 对象的状态是否为有信号。如果返回值为 true，这个对象是有信号的；否则由于时间超出而返回。

我们整理一下实现取消的特点。如果你想实现一个取消按钮，它能够取消后台线程中的一个长操作，按照以下步骤：

在你的窗体上加入 AutoResetEvent 域类型

通过在 AutoResetEvent 的构造函数中传入 false 参数，设置该对象初始状态为无信号的。 接着在你的窗体上保 存该对象的引用域，这是为了能够在窗体的整个生命周期内可以对后台线程的后台操作实现取消操作。

在你窗体上加入一个取消按钮。

在取消按钮的 Click 事件处理器中，通过调用 AutoResetEvent 对象的 Set 方法使其有信号。

同时，在你的后台线程的逻辑中周期性地在 AutoResetEvent 对象上调用 WaitOne 来检查用户是否取消了。

```
if(cancelEvent.WaitOne(0, false)){
// cancel operation
}
```

你必须记住使用零毫秒参数，这样可以避免在后台线程操作中不必要的停顿。

如果用户取消了操作，通过主线程 AutoResetEvent 会被设为有信号的。 当 WaitOne 返回 true 时你的后台线程会 得到警告，并停止操作。同时在后台线程中由于调用了 WaitOne 该事件会被自动的置为无信号状态。

为了能够看到取消长操作窗体的例子，你可以下载 CancelableForm.cs 文件。这个代码是一个完整的程序，它与 Figure 4 中的 CorrectMultiThreadedForm.cs 只有稍微的不同。

注意在 CancelableForm.cs 也采用了比较高级的用法 Control.Invoke, 在那里 EnableControls 方法被设计用来调用它自己如果当它被一个错误的线程所调用时。在它使用窗体上的任何 GUI 对象的方法或属性时要先做这个检查。 这样能够使得 EnableControls 能够从任何线程中直接安全的调用，在方法的实现中有效的隐藏了 Invoke 调用的复杂性。这些可以使应用程序更加有维护性。注意在这个例子中同样使用了 Control.BeginInvoke, 它是 Control.Invoke 的异步版本。

你也许注意到取消的逻辑依赖于后台线程通过 WaitOne 调用周期性的取消检查的能力。但是如果正在讨论的问题不能被取消怎么办？如果后台操作是一个单个调用，像 DataAdapter.Fill，它会花很长时间？有时会有解决办法的，但并不总是。

如果你的长操作根本不能取消，你可以使用一个伪取消的方法来完成你的操作，但在你的程序中不要影响你的操作结果。这不是技术上的取消操作，它把一个可忍受的操作帮定到一个线程池中，但这是在某种情况下的一种折中办法。如果你实现了类似的解决办法，你应该从你的取消按钮事件处理器中直接使能你已禁止的 UI 元素，而不要还依赖于被绑定的后台线程通过 Invoke 调用使能你的控件。同样重要的使设计你的后台操作线程，当其返回时测试一下它是否被取消，以便它不影响现在被取消的操作的结果。

这种长操作取消是比较高级的方法，它只在某些情况下才可行。例如，数据库查询的伪取消就是这样，但是一个数据库的更新，删除，插入伪取消是一个滞后的操作。有永久的操作结果或与反馈有关的操作，像声音和图像，就不容易使用伪取消方法，因为操作的结果在用户取消以后是非常明显的。

更多细节－有关定时器

在应用程序中需要一个定时器来引发一个定期的任务一定不一般。例如，如果你的程序在窗体的状态条上显示当前时间，你可能每 5 秒钟更新一次时间。System.Threading 名字空间包括了一个名为 Timer 多线程定时器类。

当你创建一个定时器类的实例时，你为定时器回调指明了一个以毫秒为单位的周期，而且你也传递给该对象一个委托用来每过一个时钟周期调用你。回调发生在线程池中的线程上。事实上，每次时钟周期到来时真正发生的是一个工作条目在线程池中排队；一般来说一个调用会马上发生的，但是如果线程池比较忙，这个回调也许会在稍后的一个时间点发生。

如果你考虑在你的程序中使用多线程，你也许会考虑使用定时器类。然而，如果你的程序使用了 Windows 窗体，你不必使用多线程的行为，在 System.Windows.Forms 名字空间中有另外一个也叫 Timer 的定时器类。

System.Windows.Forms.Timer 与其多线程的同伴比起来有一个明显的好处：因为它不是多线程的，所以不会在其它线程中对你进行回调，而且更适合为应用程序提取窗口消息的主线程。实际上 System.Windows.Forms.Timer 的实现是在系统中使用了 WM_TIMER 的一个窗口消息。这种方法在你的 System.Windows.Forms.Timer 的事件处理器中不必担心线程同步，线程间通信之类的问题。

对于 Windows 窗体类程序，作为一个很好的技巧就是使用 System.Windows.Forms.Timer 类， 除非你特别需要线程池中的线程对你进行回调。既然这种要求很少见，为了使事情简单，把使用 System.Windows.Forms.Timer 作为一个规则，即使在你的程序的其它地方使用了多线程。

展望将来

微软最近展示了一个即将出现的 GUI API，代号为“Avalon”，本期 MSDN 杂志的问题列表中（见 70 页）Charles Petzold''''s 的文章描述了其特点。在 Avalon 框架中用户接口元素没有被系与一个特殊的线程；作为更换每个用户接口元素与一个单独的逻辑线程上下文相关联，在 UIContext 类中实现。但是当你发现 UIContext 类中包含了 Invoke 方法，及其姊妹 BeginInvoke 时，你就不会惊奇了，在名字上与窗体类中的控件类上名称一样的目的是说明他们在逻辑作用上是一致的。

作者简介

Jason Clark 为微软和 Wintellect 公司提供培训和咨询，他是 Windows NT 和 Windows 2000 服务器团队的开发前辈。他是 Windows 2000 服务器程序编程一书的合著者。与 Jason 的联系方式：JClark@Wintellect.com

.net WinForm 控件的事件委托剖析 选择自 flashvan 的 Blog

关键字: Delegate, MulticastDelegate, EventHandler, EventHandlerList, EventHandlerList.ListEntry, Control, Component

首先从 controlInstance.Click 事件开始. 用 Reflector 反编译 System.Windows.Forms.Control 类可以看到对 Click 事件的定义:

```
        [System.Windows.Forms.SRCategory("CatAction"), System.Windows.Forms.SRDescription("ControlOnClickDescr")]
        public event EventHandler Click
        {
          add
          {
            base.Events.AddHandler(Control.EventClick, value);
          }
          remove
          {
            base.Events.RemoveHandler(Control.EventClick, value);
          }
        }
```

这里的 Control.EventClick 是一个只读的静态私有属性,它是以后事件查找委托的键(key),请记住这个.

```
private static readonly object EventClick = new object();
```

Control 的 Events 属性是由 System.ComponentModel.Component 继承而来,它是 EventHandlerList 的实例.

```
private EventHandlerList events;
```

```
protected EventHandlerList Events
{
    get
    {
        if (this.events == null)
        {
            this.events = new EventHandlerList();
        }
        return this.events;
    }
}
```

EventHandlerList 类有三个重要的方法:

```
    public void AddHandler(object key, Delegate value);
    public void RemoveHandler(object key, Delegate value);
    private ListEntry Find(object key);
```

AddHandler 的作用是插入一个键和一个委托类型的值，插入之前通过 Find 方法检查一下同样的委托对象是否存在,如果存在,则合并；如果不存在,以要插入的委托对象(value)为头.

```
public void AddHandler(object key, Delegate value)
{
    EventHandlerList.ListEntry entry1 = this.Find(key);
    if (entry1 != null)
    {
        entry1.handler = Delegate.Combine(entry1.handler, value);
    }
    else
```

```
    {
        this.head = new EventHandlerList.ListEntry(key, value, this.head);
    }
}
```

如果是一个按钮的 Click 事件,我们一般定义为:

```
button1.Click += new EventHandler(OnButtonClick);

protected void OnButtonClick(object sender, System.EventArgs e)
{
    // 你的处理函数
}
```

则通过了 button1.Events.AddHandler(Control.EventClick, EventHandler handler),而这个 handler 却是一个 MulticastDelegate 的实例。看 MS 公布的.net framework 部分源码就知道了:

```
// ==++==
//
//
//    Copyright (c) 2002 Microsoft Corporation.  All rights reserved.
//
//    The use and distribution terms for this software are contained in the file
//    named license.txt, which can be found in the root of this distribution.
//    By using this software in any fashion, you are agreeing to be bound by the
//    terms of this license.
//
//    You must not remove this notice, or any other, from this software.
```

```
//
//
// ==--==
namespace System {

    using System;
    /// <include file='doc\EventHandler.uex' path='docs/doc[@for="EventHandler"]/*' />
    [Serializable()]
    public delegate void EventHandler(Object sender, EventArgs e);}
```

现在我们转到对委托(Delegate)和多播委托(MulticastDelegate)的研究了。

Delegate 类已经封装好产生委托，消除委托和执行委法的方法，它是一个不能实例化的抽象类。但.net 的编译器支持由 delegate 定义的类型来实例化一个 Delegate 对象，它能让这个对象的执行委托方法像普通函数一样调用(具体的可以看 C#高级编程里面的实例)，所以很多时候，delegate 类型会被认为是函数指针。Delegate 还有两个很重要的方法，组合委托 Combine 和删除委托 Remove。在单播委托 Delegate 中使用这组合委托方法会抛出多播不支持异常(MulticastNotSupportedException)。而使用删除委托方法时，如果这个单播委托和要删除的委托是同一个值时，则返回 null，证明已经删除；如果不是，则原封不动返回原来的单播委托。

EventHandler 实际上是一个多播委托实例，所以它支持组合委托和删除委托的方法。这个实例，实际上是一个委托实例链，它是这个链的链尾。每次像调用普通函数调用这个委托的时候，这个委托会执行完委托的代理函数，并查找链表中上一个委托实例，执行这个委托的代理函数，再查找链表中上上个委托实例，执行当前委托的代理函数。。。　一直到链表被遍历完。

```
        protected override sealed Object DynamicInvokeImpl(Object[] args)
        {
            if (_prev != null)
                _prev.DynamicInvokeImpl(args);
            return base.DynamicInvokeImpl(args);
        }
```

好了。那可以想象，一个用户点击按钮 button1，首先执行的函数是 OnClick 函数

```
[EditorBrowsable(EditorBrowsableState.Advanced)]
protected virtual void OnClick(EventArgs e)
{
  if (this.CanRaiseEvents)
  {
    EventHandler handler1 = (EventHandler) base.Events[Control.EventClick];
    if (handler1 != null)
    {
      handler1(this, e);
    }
  }
}
```

handler1 就是一个多播委托，　如果不为空，则执行它，而且这个执行就是执行所有的代理函数。这样就明白了WinForm控件Click事件所有的始终了！

参考文章: http://blog.sunmast.com/Sunmast/archive/2005/04/21/1769.aspx
以上是个人观点，由于时间仓促及个人水平有限，难免错误，敬请斧正！

c#中使用多线程（图）　　选择自 iuhxq 的 Blog

```
using System;
using System.Drawing;
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;
using System.Data;
using System.Threading;
namespace student
{
 /// <summary>
 /// Form1 的摘要说明。
 /// </summary>
 public class Form1 : System.Windows.Forms.Form
 {
  private System.Windows.Forms.Button button1;
  private System.Windows.Forms.Button button2;
  private System.Windows.Forms.Button button3;
  ArrayList threads = new ArrayList();
  private System.Windows.Forms.ListView listView1;
  private System.Windows.Forms.ColumnHeader columnHeader1;
  private System.Windows.Forms.ColumnHeader columnHeader2;
  /// <summary>
  /// 必需的设计器变量。
  /// </summary>
  private System.ComponentModel.Container components = null;
  public Form1()
  {
   //
   // Windows 窗体设计器支持所必需的
   //
   InitializeComponent();
   //
   // TODO: 在 InitializeComponent 调用后添加任何构造函数代码
   //
  }
  /// <summary>
  /// 清理所有正在使用的资源。
  /// </summary>
  protected override void Dispose( bool disposing )
  {
   if( disposing )
   {
```

```
  if (components != null)
  {
   components.Dispose();
  }
 }
 base.Dispose( disposing );
}
#region Windows 窗体设计器生成的代码
/// <summary>
/// 设计器支持所需的方法 - 不要使用代码编辑器修改
/// 此方法的内容。
/// </summary>
private void InitializeComponent()
{
 this.button1 = new System.Windows.Forms.Button();
 this.button2 = new System.Windows.Forms.Button();
 this.button3 = new System.Windows.Forms.Button();
 this.listView1 = new System.Windows.Forms.ListView();
 this.columnHeader1 = new System.Windows.Forms.ColumnHeader();
 this.columnHeader2 = new System.Windows.Forms.ColumnHeader();
 this.SuspendLayout();
 //
 // button1
 //
 this.button1.Location = new System.Drawing.Point(24, 216);
 this.button1.Name = "button1";
 this.button1.TabIndex = 1;
 this.button1.Text = "Add";
 this.button1.Click += new System.EventHandler(this.button1_Click);
 //
 // button2
 //
 this.button2.Location = new System.Drawing.Point(104, 216);
 this.button2.Name = "button2";
 this.button2.TabIndex = 2;
 this.button2.Text = "Del";
 this.button2.Click += new System.EventHandler(this.button2_Click);
 //
 // button3
 //
 this.button3.Location = new System.Drawing.Point(184, 216);
 this.button3.Name = "button3";
 this.button3.TabIndex = 3;
 this.button3.Text = "DelAll";
```

```
 this.button3.Click += new System.EventHandler(this.button3_Click);
 //
 // listView1
 //
 this.listView1.Columns.AddRange(new System.Windows.Forms.ColumnHeader[] {
                    this.columnHeader1,
                    this.columnHeader2});
 this.listView1.FullRowSelect = true;
 this.listView1.GridLines = true;
 this.listView1.Location = new System.Drawing.Point(0, 0);
 this.listView1.Name = "listView1";
 this.listView1.Size = new System.Drawing.Size(288, 208);
 this.listView1.TabIndex = 5;
 this.listView1.View = System.Windows.Forms.View.Details;
 //
 // columnHeader1
 //
 this.columnHeader1.Text = "线程编号";
 this.columnHeader1.Width = 81;
 //
 // columnHeader2
 //
 this.columnHeader2.Text = "value";
 this.columnHeader2.Width = 180;
 //
 // Form1
 //
 this.AutoScaleBaseSize = new System.Drawing.Size(6, 14);
 this.ClientSize = new System.Drawing.Size(292, 266);
 this.Controls.Add(this.listView1);
 this.Controls.Add(this.button3);
 this.Controls.Add(this.button2);
 this.Controls.Add(this.button1);
 this.Name = "Form1";
 this.Text = "Form1";
 this.Load += new System.EventHandler(this.Form1_Load);
 this.ResumeLayout(false);
}
#endregion
/// <summary>
/// 应用程序的主入口点。
/// </summary>
[STAThread]
static void Main()
```

```
  {
   Application.Run(new Form1());
  }
  private void Form1_Load(object sender, System.EventArgs e)
  {

  }
  private void button1_Click(object sender, System.EventArgs e)
  {
   Add();
  }
  private void button2_Click(object sender, System.EventArgs e)
  {
   Del();
  }
  private void button3_Click(object sender, System.EventArgs e)
  {
   while(threads.Count>0)
   {
    Thread t1 = (Thread)threads[0];
    if(t1.IsAlive)
    {
     t1.Abort();
    }
    threads.RemoveAt(0);
    lock(listView1)
    {
     listView1.Items.RemoveAt(0);
    }
   }
  }
  private void Add()
  {
   int count = threads.Count;
   if(count<10)
   {
    Thread t = new Thread(new ThreadStart(Process));
    t.Start();
    threads.Add(t);
    lock(listView1)
    {
     listView1.Items.Insert(count, new ListViewItem(new
string[]{count.ToString(),"0"}));
    }
```

```
    }
   }
   private void Del()
   {
    int count = threads.Count;
    if(count>0)
    {
     Thread t1 = (Thread)threads[count-1];
     if(t1.IsAlive)
     {
      t1.Abort();
     }
     threads.RemoveAt(count-1);
     lock(listView1)
     {
      listView1.Items.RemoveAt(count-1);
     }
    }
   }

   private void Process()
   {
    int i = 1;
    while(true)
    {
     i++;
     int j = threads.IndexOf(Thread.CurrentThread);
     lock(listView1)
     {
      listView1.Items[j].SubItems[1].Text = i.ToString();
     }
     Thread.Sleep(0);
    }
   }
  }
}
```

作者Blog： http://blog.csdn.net/iuhxq/

相关文章

[原创]c#中使用多线程（图）二

SQL 存储过程&算法

取汉字拼音首字母的存储过程

DataGrid自定义分页存储过程

利用Visual C#打造一个平滑的进度条　　选择自 swazn_yj 的 Blog

利用 Visual C#打造一个平滑的进度条

概述

本文描述了如何建立一个简单的、自定义的用户控件——一个平滑的进度条。

在早先的进度条控件版本中，例如在 Microsoft Windows Common Controls ActiveX 控件中提供的版本，您可以看到进度条有两种不同的视图。您可以通过设定 Scrolling 属性来设定 Standard 视图或是 Smooth 视图。 Smooth 视图提供了一个区域来平滑的显示进度，Standard 试图则看上去是由一个一个方块来表示进度的。

在 Visual C# .NET 中提供的进度条控件只支持 Standard 视图。

本文的代码样例揭示了如何建立一个有如下属性的控件：

Minimum。该属性表示了进度条的最小值。默认情况下是 0 ；您不能将该属性设为负值。

Maximum。该属性表示了进度条的最大值。默认情况下是 100 。

Value。该属性表示了进度条的当前值。该值必须介于 Minimum 和 Maximum 之间。

ProgressBarColor。该属性表示了进度条的颜色。

建立一个自定义的进度条控件

1、按着下面的步骤，在 Visual C# .NET 中建立一个 Windows Control Library 项目：

a、打开 Microsoft Visual Studio .NET。

b、点击 File 菜单，点击 New ，再点击 Project 。

c、在 New Project 对话框中，在 Project Types 中选择 Visual C# Projects，然后在 Templates 中选择 Windows Control Library 。

d、在 Name 框中，填上 SmoothProgressBar ，并点击 OK 。

e、在 Project Explorer 中，重命名缺省的 class module ，将 UserControl1.cs 改为 SmoothProgressBar.cs 。

f、在该 UserControl 对象的 Property 窗口中，将其 Name 属性从 UserControl1 改为 SmoothProgressBar 。

2、此时，您已经从 control 类继承了一个新类，并可以添加新的功能。但是，ProgressBar 累是密封(sealed)的，不能再被继承。因此，您必须从头开始建立这个控件。

将下面的代码添加到UserControl 模块中，就在“Windows Form Designer generated code”之后：

```
int min = 0; // Minimum value for progress range
int max = 100; // Maximum value for progress range
int val = 0; // Current progress
Color BarColor = Color.Blue; // Color of progress meter

protected override void OnResize(EventArgs e)
{
  // Invalidate the control to get a repaint.
  this.Invalidate();
}

protected override void OnPaint(PaintEventArgs e)
{
  Graphics g = e.Graphics;
  SolidBrush brush = new SolidBrush(BarColor);
  float percent = (float)(val - min) / (float)(max - min);
  Rectangle rect = this.ClientRectangle;

  // Calculate area for drawing the progress.
  rect.Width = (int)((float)rect.Width * percent);

  // Draw the progress meter.
  g.FillRectangle(brush, rect);

  // Draw a three-dimensional border around the control.
  Draw3DBorder(g);

  // Clean up.
  brush.Dispose();
  g.Dispose();
}

public int Minimum
```

```
{
  get
  {
    return min;
  }

  set
  {
    // Prevent a negative value.
    if (value < 0)
    {
      min = 0;
    }

    // Make sure that the minimum value is never set higher than
the maximum value.
    if (value > max)
    {
      min = value;
      min = value;
    }

    // Ensure value is still in range
    if (val < min)
    {
      val = min;
    }

    // Invalidate the control to get a repaint.
    this.Invalidate();
  }
}

public int Maximum
{
  get
  {
    return max;
  }

  set
  {
    // Make sure that the maximum value is never set lower than
the minimum value.
```

```
    if (value < min)
    {
      min = value;
    }

    max = value;

    // Make sure that value is still in range.
    if (val > max)
    {
      val = max;
    }

    // Invalidate the control to get a repaint.
    this.Invalidate();
  }
}

public int Value
{
  get
  {
    return val;
  }

  set
  {
    int oldValue = val;

    // Make sure that the value does not stray outside the valid
range.
    if (value < min)
    {
      val = min;
    }
    else if (value > max)
    {
      val = max;
    }
    else
    {
      val = value;
    }
```

```
    // Invalidate only the changed area.
    float percent;

    Rectangle newValueRect = this.ClientRectangle;
    Rectangle oldValueRect = this.ClientRectangle;

    // Use a new value to calculate the rectangle for progress.
    percent = (float)(val - min) / (float)(max - min);
    newValueRect.Width = (int)((float)newValueRect.Width *
percent);

    // Use an old value to calculate the rectangle for progress.
    percent = (float)(oldValue - min) / (float)(max - min);
    oldValueRect.Width = (int)((float)oldValueRect.Width *
percent);

    Rectangle updateRect = new Rectangle();

    // Find only the part of the screen that must be updated.
    if (newValueRect.Width > oldValueRect.Width)
    {
      updateRect.X = oldValueRect.Size.Width;
      updateRect.Width = newValueRect.Width -
oldValueRect.Width;
    }
    else
    {
      updateRect.X = newValueRect.Size.Width;
      updateRect.Width = oldValueRect.Width -
newValueRect.Width;
    }

    updateRect.Height = this.Height;

    // Invalidate the intersection region only.
    this.Invalidate(updateRect);
  }
}

public Color ProgressBarColor
{
  get
  {
    return BarColor;
```

```
  }

  set
  {
    BarColor = value;

    // Invalidate the control to get a repaint.
    this.Invalidate();
  }
}

private void Draw3DBorder(Graphics g)
{
  int PenWidth = (int)Pens.White.Width;

  g.DrawLine(Pens.DarkGray, new
Point(this.ClientRectangle.Left, this.ClientRectangle.Top),
new Point(this.ClientRectangle.Width - PenWidth,
this.ClientRectangle.Top));
  g.DrawLine(Pens.DarkGray, new
Point(this.ClientRectangle.Left, this.ClientRectangle.Top), new
Point(this.ClientRectangle.Left, this.ClientRectangle.Height -
PenWidth));
  g.DrawLine(Pens.White, new Point(this.ClientRectangle.Left,
this.ClientRectangle.Height - PenWidth),
new Point(this.ClientRectangle.Width - PenWidth,
this.ClientRectangle.Height - PenWidth));
g.DrawLine(Pens.White, new Point(this.ClientRectangle.Width -
PenWidth, this.ClientRectangle.Top),
new Point(this.ClientRectangle.Width - PenWidth,
this.ClientRectangle.Height - PenWidth));
}
```

3、在 Build 菜单中，点击 Build Solution 来编译整个项目。

建立一个简单的客户端应用

1、在 File 菜单中，点击 New ，再点击 Project。

2、在 Add New Project 对话框中，在 Project Types 中点击 Visual C# Projects，在 Templates 中点击 Windows Application，并点击 OK。

3、按照下面的步骤，在 Form 上添加两个 SmoothProgressBar 实例：

a、在 Tools 菜单上，点击 Customize Toolbox。

b、点击 .NET Framework Components 页。

c、点击 Browse，然后选中你在 Create a Custom ProgressBar Control 段中建立的 SmoothProgressBar.dll 文件。

d、点击 OK。您可以看到在 toolbox 中已经有 SmoothProgressBar 控件了。

e、从 toolbox 中拖两个 SmoothProgressBar 控件的实例到该 Windows Application 项目中的默认 form 上。

4、从 toolbox 页中拖一个 Timer 控件到 form 上。

5、将下面的代码添加到 Timer 控件的 Tick 事件中：

```
if (this.smoothProgressBar1.Value > 0)
{
  this.smoothProgressBar1.Value--;
  this.smoothProgressBar2.Value++;
}
else
{
  this.timer1.Enabled = false;
}
```

6、从 toolbox 页中拖一个 Button 控件到 form 上。

7、将下面的代码添加到 Button 控件的 Click 事件中：

```
this.smoothProgressBar1.Value = 100;
this.smoothProgressBar2.Value = 0;

this.timer1.Interval = 1;
this.timer1.Enabled = true;
```

8、在 Debug 菜单中，点击 Start 来运行样例项目。

9、点击 Button。注意观察那两个进度指示器。一个逐渐减小，另一个逐渐增加。

作者Blog：http://blog.csdn.net/swazn_yj/

转 Windows 窗体多线程

Windows 窗体多线程

当我们在编写一个需要长时间运行的程序时(如数学计算,执行数据库命令,访问 WebService)常常将它们写在一个组件中,让他们在后台运行.从而不影响 Windows 界面的显示和界面上的交互操作.但我们有时还是感到不怎方便,如我们不能直接应用 winForm 里定义的变量等.那么在 UI 进程中能否直接执行长时间运行的程序,而不影响 UI 进程呢?

下面的示例将解决这个问题.

本例利用多线程从长时间运行的操作(计算 fbnc 数列(n>36))中分离出用户界面 (UI)，以将用户的后续输入传递给辅助线程(CalHandler,showDel)以调节其行为与用户界面元素进行交互，从而实现稳定而正确的多线程处理的消息传递方案。

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;
using System.Threading;
using System.Runtime.Remoting;
```

```
using System.Runtime.Remoting.Messaging;


namespace AsynchCalcPi
{
    public partial class Form2 : Form
    {
        public Form2()
        {
            InitializeComponent();
            Form2.calComplete += new CalHandler(Form2_calComplete);
        }

        void Form2_calComplete(string strTemp)
        {

            //为了对调用者屏蔽与此 UI 线程有关的线程安全通信信息，
            //ShowCalcResult 方法在此 UI 线程上通过 Control.BeginInvoke 方法使
用 showDel 给自己发送消息。
            //Control.BeginInvoke 异步队列为 UI 线程提供服务，并且不等待结果就继续运
行。
            if(!bClose )
            this.BeginInvoke(new showDel(showRes ),strTemp );


        }

        int times = 1;
        private void showRes(string strTemp)
        {
            times += 1;
            this.richTextBox1.AppendText("," + strTemp);
            this.progressBar1.Value = iStep * times%100;
            if (FinishFlag)
            {

                //timer1.Enabled = false;
                MessageBox.Show(strTemp);
            }
        }
        private delegate void showDel(string stemp);
```

```
        private void button1_Click(object sender, EventArgs e)
        {
            try
            {

                j = Int32.Parse(this.textBox_Num.Text.Trim());
                iStep = 100 / j;
                if (j < 1)
                    return;
            }
            catch
            {
                MessageBox.Show("请在文本框内输入数字字符");
                return;
            }
            for (int i = 0; i < j; i++)
                this.richTextBox1.AppendText(this.ComputeFibonacci(i).ToString() +
",");

        }


        private long ComputeFibonacci(int n)
        {
            //' The parameter n must be >= 0 and <= 91.
            //' Fib(n), with n > 91, overflows a long.
            if (n < 0 || n > 91)
            {
                MessageBox.Show("value must be >= 0 and <= 91", "n");
            }

            long result = 0;

            if (n < 2)
                result = 1;
            else
            {

                result = ComputeFibonacci(n - 1) + ComputeFibonacci(n - 2);
            }
            return result;
        }

        public int AddInterlink(int i)
```

```
        {
            if (i <= 0)
                return 0;
            else if (i > 0 && i <= 2)
                return 1;
            else return AddInterlink(i - 1) + AddInterlink(i - 2);
        }

        private void button2_Click(object sender, EventArgs e)
        {
            try
            {

                j = Int32.Parse(this.textBox_Num.Text.Trim());
                iStep = 100 / j;
                if (j < 1)
                    return;
            }
            catch
            {
                MessageBox.Show("请在文本框内输入数字字符");
                return;
            }

            for (int i = 0; i < j; i++)
                this.richTextBox1.AppendText(this.AddInterlink(i).ToString() + ",");
        }

        private void button3_Click(object sender, EventArgs e)
        {
            try
            {

                j = Int32.Parse(this.textBox_Num.Text.Trim());
                iStep = 100 / j;
                if (j < 1)
                    return;
            }
            catch
            {
                MessageBox.Show("请在文本框内输入数字字符");
                return;
            }
            ComputeFibonacciDel calcFbnc = new ComputeFibonacciDel(this.Compu
```

```
teFibonacci);

            calcFbnc.BeginInvoke(j, callBack, null);

        }

        //实时显示通知服务
        private long ShowCalcResult(int n)
        {
            long result1 = 0;
            for (int i = 0; i < n; i++)
            {
                result1 = this.ComputeFibonacci(i);
                //委托 calComplete 由辅助线程用于向 UI 线程回传消息，通常是有关长时间运
行的操作的最新进度。

                calComplete(result1.ToString() );
            }
            return result1;
        }
        //定义计算过程中用于传递消息的委托
        public delegate void CalHandler(string strTemp);
        //定义事件
        public static event CalHandler calComplete;
        //定义委托 进行异步计算 Fibonacci 数列
        private delegate long ComputeFibonacciDel(int n);

        //定义引用在异步操作完成时调用的回调方法.用以在计算完成后取得返回值和当前状
态.
        AsyncCallback callBack = new AsyncCallback(ShowResult);
        private  static  bool  FinishFlag = false;
        static void ShowResult(IAsyncResult ar)
        {
            // Asynchronous Callback method.
            // Obtains the last parameter of the delegate call.
            int value = Convert.ToInt32(ar.AsyncState);
            // Obtains return value from the delegate call using EndInvoke.
            AsyncResult aResult = (AsyncResult)ar;
            ComputeFibonacciDel temp = (ComputeFibonacciDel)aResult.AsyncDele
gate;
            long result = temp.EndInvoke(ar);
            FinishFlag = true;

            calComplete("当前状态代号:" + value.ToString() + "  " + "计算后的返回结
```

```
果:" + result.ToString());

        }
        int i = 0;
        private void timer1_Tick(object sender, EventArgs e)
        {
          i += 1;
          i = i % 100;
          this.progressBar1.Value = i;

        }

        int j = 0;
        int iStep = 1;
        ComputeFibonacciDel calcFbnc;
        private void button4_Click(object sender, EventArgs e)
        {
          FinishFlag = false;
          //停止进度条的自动滚动.让进度条根据当前进度显示
          this.timer1.Enabled = false;
          this.progressBar1.Value = 0;
          try
          {

            j= Int32.Parse(this.textBox_Num.Text.Trim());
            iStep = 100 / j ;
             if (j < 1)
                 return;
          }
          catch
          {
             MessageBox.Show("请在文本框内输入数字字符");
             return;
          }

          //ComputeFibonacciDel，用于捆绑要传递给（从线程池中分配的）辅助线程上的
ShowCalcResult 的参数。
          //当用户决定要计算 fbnc 数列 时，事件处理程序将创建此委托的一个实例。
          //此工作通过调用 BeginInvoke 在线程池中进行排队。该委托实际上是由 UI 线程
用于向辅助线程传递消息。
         calcFbnc = new ComputeFibonacciDel(this.ShowCalcResult );

          IAsyncResult aResult= calcFbnc.BeginInvoke(j,callBack , null);

```

```
        //已在 callBack 方法中写出,此处不再写此方法.
          ////Wait for the call to complete
        //aResult.AsyncWaitHandle.WaitOne();
        //long callResult = calcFbnc.EndInvoke(aResult);
    }
    bool bClose = false;
    private void Form2_FormClosing(object sender, FormClosingEventArgs e)
      {
        bClose = true;

    }
```

.NET Framework 异步调用

.NET Framework 异步调用

.NET Framework 允许您异步调用任何方法。定义与您需要调用的方法具有相同签名的委托；公共语言运行库将自动为该委托定义具有适当签名的 **BeginInvoke** 和 **EndInvoke** 方法。

BeginInvoke 方法用于启动异步调用。它与您需要异步执行的方法具有相同的参数，只不过还有两个额外的参数（将在稍后描述）。**BeginInvoke** 立即返回，不等待异步调用完成。**BeginInvoke** 返回 IasyncResult，可用于监视调用进度。

EndInvoke 方法用于检索异步调用结果。调用 **BeginInvoke** 后可随时调用 **EndInvoke** 方法；如果异步调用未完成，**EndInvoke** 将一直阻塞到异步调用完成。**EndInvoke** 的参数包括您需要异步执行的方法的 **out** 和 **ref** 参数（在 Visual Basic 中为 <Out> **ByRef** 和 **ByRef**）以及由 **BeginInvoke** 返回的 **IAsyncResult**。

注意 Visual Studio .NET 中的智能感知功能会显示 **BeginInvoke** 和 **EndInvoke** 的参数。如果您没有使用 Visual Studio 或类似的工具，或者您使用的是 C# 和 Visual Studio .NET，请参见 异步方法签名获取有关运行库为这些方法定义的参数的描述。

本主题中的代码演示了四种使用 **BeginInvoke** 和 **EndInvoke** 进行异步调用的常用方法。调用了 **BeginInvoke** 后，可以：

- 进行某些操作，然后调用 **EndInvoke** 一直阻塞到调用完成。
- 使用 IAsyncResult.AsyncWaitHandle 获取 WaitHandle，使用它的 WaitOne 方法将执行一直阻塞到发出 **WaitHandle** 信号，然后调用 **EndInvoke**。
- 轮询由 **BeginInvoke** 返回的 **IAsyncResult**，确定异步调用何时完成，然后调用 **EndInvoke**。
- 将用于回调方法的委托传递给 **BeginInvoke**。该方法在异步调用完成后在 ThreadPool 线程上执行，它可以调用 **EndInvoke**。

警告 始终在异步调用完成后调用 **EndInvoke**。

测试方法和异步委托

四个示例全部使用同一个长期运行的测试方法 **TestMethod**。该方法显示一个表明它已开始处理的控制台信息，休眠几秒钟，然后结束。**TestMethod** 有一个 **out** 参数（在 Visual Basic 中为 <Out> **ByRef**），它演示了如何将这些参数添加到 **BeginInvoke** 和 **EndInvoke** 的签名中。您可以用类似的方式处理 **ref** 参数（在 Visual Basic 中为 **ByRef**）。

下面的代码示例显示 **TestMethod** 以及代表 **TestMethod** 的委托；若要使用任一示例，请将示例代码追加到这段代码中。

注意 为了简化这些示例，**TestMethod** 在独立于 Main() 的类中声明。或者，**TestMethod** 可以是包含 Main() 的同一类中的 **static** 方法（在 Visual Basic 中为 **Shared**）。

```
using System;
using System.Threading;

public class AsyncDemo {
    // The method to be executed asynchronously.
    //
    public string TestMethod(int callDuration, out int threadId) {
        Console.WriteLine("Test method begins.");
        Thread.Sleep(callDuration);
        threadId = AppDomain.GetCurrentThreadId();
        return "MyCallTime was " + callDuration.ToString();
    }
}

// The delegate must have the same signature as the method
// you want to call asynchronously.
public delegate string AsyncDelegate(int callDuration, out int threadId);
```


```
using System;
using System.Threading;

public class AsyncDemo {
    // The method to be executed asynchronously.
    //
    public string TestMethod(int callDuration, out int threadId) {
        Console.WriteLine("Test method begins.");
        Thread.Sleep(callDuration);
        threadId = AppDomain.GetCurrentThreadId();
        return "MyCallTime was " + callDuration.ToString();
    }
}

// The delegate must have the same signature as the method
// you want to call asynchronously.
public delegate string AsyncDelegate(int callDuration, out int threadId);
```

使用 EndInvoke 等待异步调用

异步执行方法的最简单方式是以 BeginInvoke 开始，对主线程执行一些操作，然后调用 EndInvoke。EndInvoke 直到异步调用完成后才返回。这种技术非常适合文件或网络操作，但是由于它阻塞 EndInvoke，所以不要从用户界面的服务线程中使用它。

```
public class AsyncMain {
    static void Main(string[] args) {
        // The asynchronous method puts the thread id here.
        int threadId;

        // Create an instance of the test class.
        AsyncDemo ad = new AsyncDemo();

        // Create the delegate.
        AsyncDelegate dlgt = new AsyncDelegate(ad.TestMethod);

        // Initiate the asychronous call.
        IAsyncResult ar = dlgt.BeginInvoke(3000,
            out threadId, null, null);

        Thread.Sleep(0);
        Console.WriteLine("Main thread {0} does some work.",
            AppDomain.GetCurrentThreadId());

        // Call EndInvoke to Wait for the asynchronous call to complete,
        // and to retrieve the results.
        string ret = dlgt.EndInvoke(out threadId, ar);

        Console.WriteLine("The call executed on thread {0}, with return value \"{1}\".", threadId, ret);
    }
}
```

使用 WaitHandle 等待异步调用

等待 WaitHandle 是一项常用的线程同步技术。您可以使用由 BeginInvoke 返回的 IAsyncResult 的 AsyncWaitHandle 属性来获取 WaitHandle。异步调用完成时会发出 WaitHandle 信号，而您可以通过调用它的 WaitOne 等待它。

如果您使用 WaitHandle，则在异步调用完成之后，但在通过调用 EndInvoke 检索结果之前，可以执行其他处理。

```
public class AsyncMain {
    static void Main(string[] args) {
        // The asynchronous method puts the thread id here.
        int threadId;

        // Create an instance of the test class.
        AsyncDemo ad = new AsyncDemo();

        // Create the delegate.
        AsyncDelegate dlgt = new AsyncDelegate(ad.TestMethod);

        // Initiate the asychronous call.
        IAsyncResult ar = dlgt.BeginInvoke(3000,
            out threadId, null, null);

        Thread.Sleep(0);
        Console.WriteLine("Main thread {0} does some work.",
            AppDomain.GetCurrentThreadId());

        // Wait for the WaitHandle to become signaled.
        ar.AsyncWaitHandle.WaitOne();

        // Perform additional processing here.
        // Call EndInvoke to retrieve the results.
        string ret = dlgt.EndInvoke(out threadId, ar);

        Console.WriteLine("The call executed on thread {0}, with return value \"{1}\".", threadId, ret);
    }
}
```

轮询异步调用完成

您可以使用由 BeginInvoke 返回的 IAsyncResult 的 IsCompleted 属性来发现异步调用何时完成。从用户界面的服务线程中进行异步调用时可以执行此操作。轮询完成允许用户界面线程继续处理用户输入。

```
public class AsyncMain {
    static void Main(string[] args) {
        // The asynchronous method puts the thread id here.
        int threadId;

        // Create an instance of the test class.
        AsyncDemo ad = new AsyncDemo();

        // Create the delegate.
        AsyncDelegate dlgt = new AsyncDelegate(ad.TestMethod);

        // Initiate the asychronous call.
        IAsyncResult ar = dlgt.BeginInvoke(3000,
            out threadId, null, null);

        // Poll while simulating work.
        while(ar.IsCompleted == false) {
            Thread.Sleep(10);
        }

        // Call EndInvoke to retrieve the results.
        string ret = dlgt.EndInvoke(out threadId, ar);

        Console.WriteLine("The call executed on thread {0}, with return value \"{1}\".", threadId, ret);
    }
}
```

异步调用完成时执行回调方法

如果启动异步调用的线程不需要处理调用结果，则可以在调用完成时执行回调方法。回调方法在 ThreadPool 线程上执行。

要使用回调方法，必须将代表该方法的 AsyncCallback 委托传递给 BeginInvoke。也可以传递包含回调方法将要使用的信息的对象。例如，可以传递启动调用时曾使用的委托，以便回调方法能够调用 EndInvoke。

```
public class AsyncMain {
    // Asynchronous method puts the thread id here.
    private static int threadId;

    static void Main(string[] args) {
        // Create an instance of the test class.
        AsyncDemo ad = new AsyncDemo();

        // Create the delegate.
        AsyncDelegate dlgt = new AsyncDelegate(ad.TestMethod);

        // Initiate the asychronous call.  Include an AsyncCallback
        // delegate representing the callback method, and the data
        // needed to call EndInvoke.
        IAsyncResult ar = dlgt.BeginInvoke(3000,
            out threadId,
            new AsyncCallback(CallbackMethod),
            dlgt );

        Console.WriteLine("Press Enter to close application.");
        Console.ReadLine();
    }

    // Callback method must have the same signature as the
    // AsyncCallback delegate.
    static void CallbackMethod(IAsyncResult ar) {
        // Retrieve the delegate.
        AsyncDelegate dlgt = (AsyncDelegate) ar.AsyncState;

        // Call EndInvoke to retrieve the results.
        string ret = dlgt.EndInvoke(out threadId, ar);

        Console.WriteLine("The call executed on thread {0}, with return value \"{1}\".", threadId, ret);
    }
}
```

C#的多线程机制探索

注：本文中出现的代码均在.net Framework RC3 环境中运行通过

一.多线程的概念

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。浏览器就是一个很好的多线程的例子,在浏览器中你可以在下载 JAVA 小应用程序或图象的同时滚动页面,在访问新页面时,播放动画和声音,打印文件等。

多线程的好处在于可以提高 CPU 的利用率——任何一个程序员都不希望自己的程序很多时候没事可干，在多线程程序中，一个线程必须等待的时候，CPU 可以运行其它的线程而不是等待，这样就大大提高了程序的效率。

然而我们也必须认识到线程本身可能影响系统性能的不利方面，以正确使用线程：

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

基 于以上认识，我们可以一个比喻来加深理解。假设有一个公司，公司里有很多各司其职的职员，那么我们可以认为这个正常运作的公司就是一个进程，而公司里的职 员就是线程。一个公司至少得有一个职员吧，同理，一个进程至少包含一个线程。在公司里，你可以一个职员干所有的事，但是效率很显然是高不起来的，一个人的 公司也不可能做大；一个程序中也可以只用一个线程去做事，事实上，一些过时的语言如 fortune,basic 都是如此，但是象一个人的公司一样，效率很低，如果做大程序，效率更低——事实上现在几乎没有单线程的商业软件。公司的职员越多，老板就得发越多的薪水给他们，还得耗费大量精力去管理他们，协调他们之间的矛盾和利益；程序也是如此，线程越多耗费的资源也越多，需要 CPU 时间去跟踪线程，还得解决诸如死锁，同步等问题。总之，如果你不想你的公司被称为“皮包公司”，你就得多几个员工；如果你不想让你的程序显得稚气，就在你的程序里引入多线程吧！

本文将对 C#编程中的多线程机制进行探讨，通过一些实例解决对线程的控制，多线程间通讯等问题。为了省去创建 GUI 那些繁琐的步骤，更清晰地逼近线程的本质，下面所有的程序都是控制台程序，程序最后的 Console.ReadLine()是为了使程序中途停下来，以便看清楚执行过程中的输出。

好了，废话少说，让我们来体验一下多线程的 C#吧！

二.操纵一个线程

任何程序在执行时，至少有一个主线程，下面这段小程序可以给读者一个直观的印象：

```
//SystemThread.cs
using System;
using System.Threading;

namespace ThreadTest
{
    class RunIt
    {
        [STAThread]
        static void Main(string[] args)
        {
            Thread.CurrentThread.Name="System Thread";//给当前线程起名为"System
Thread"
Console.WriteLine(Thread.CurrentThread.Name+"'Status:"+Thread.CurrentThread.ThreadState);
            Console.ReadLine();
        }
    }
}
```

编译执行后你看到了什么？是的，程序将产生如下输出：

System Thread's Status:Running

在这里，我们通过 Thread 类的静态属性 CurrentThread 获取了当前执行的线程，对其 Name 属性赋值“System Thread”，最后还输出了它的当前状态（ThreadState）。所谓静态属性，就是这个类所有对象所公有的属性，不管你创建了多少个这个类的实例，但是类的静态属性在内存中只有一个。很容易理解 CurrentThread 为什么是静态的——虽然有多个线程同时存在，但是在某一个时刻，CPU 只能执行其中一个。

就像上面程序所演示的，我们通过 Thread 类来创建和控制线程。注意到程序的头部，我们使用了如下命名空间：

```
using System;
using System.Threading;
```

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

下面我们就动手来创建一个线程，使用 Thread 类创建线程时，只需提供线程入口即可。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。

打开你的 VS.net，新建一个控制台应用程序（Console Application），下面这些代码将让你体味到完全控制一个线程的无穷乐趣！

```
//ThreadTest.cs

using System;
using System.Threading;

namespace ThreadTest
{
public class Alpha
    {
        public void Beta()
        {
            while (true)
            {
                Console.WriteLine("Alpha.Beta is running in its own thread.");
            }
        }
    };

    public class Simple
    {
        public static int Main()
        {
```

```
                    Console.WriteLine("Thread Start/Stop/Join Sample");

                    Alpha oAlpha = new Alpha();
                    file://这里创建一个线程，使之执行 Alpha 类的 Beta()方法
                    Thread oThread = new Thread(new ThreadStart(oAlpha.Beta));
                    oThread.Start();
                    while (!oThread.IsAlive);
                        Thread.Sleep(1);
                    oThread.Abort();
                    oThread.Join();
                    Console.WriteLine();
                    Console.WriteLine("Alpha.Beta has finished");
                    try
                    {
                        Console.WriteLine("Try to restart the Alpha.Beta thread");
                        oThread.Start();
                    }
                    catch (ThreadStateException)
                    {
                        Console.Write("ThreadStateException trying to restart
Alpha.Beta. ");
                        Console.WriteLine("Expected since aborted threads cannot be
restarted.");
                        Console.ReadLine();
                    }
                    return 0;
                }
            }
        }
```

这段程序包含两个类 Alpha 和 Simple，在创建线程 oThread 时我们用指向 Alpha.Beta()方法的初始化了 ThreadStart 代理（delegate）对象，当我们创建的线程 oThread 调用 oThread.Start()方法启动时，实际上程序运行的是 Alpha.Beta()方法：

```
Alpha oAlpha = new Alpha();
Thread oThread = new Thread(new ThreadStart(oAlpha.Beta));
oThread.Start();
```

然后在 Main()函数的 while 循环中，我们使用静态方法 Thread.Sleep()让主线程停了 1ms，这段时间 CPU 转向执行线程 oThread。然后我们试图用 Thread.Abort()方法终止线程 oThread，注意后面的 oThread.Join()，Thread.Join()方法使主线程等待，直到 oThread 线程结束。你可以给 Thread.Join()方法指定一个 int 型的参数作为等待的最

长时间。之后，我们试图用 Thread.Start()方法重新启动线程 oThread，但是显然 Abort()方法带来的后果是不可恢复的终止线程，所以最后程序会抛出 ThreadStateException 异常。

程序最后得到的结果将如下图：

```
F:\Visual Studio Projects\ConsoleApplication4\bin\Debug\ThreadTest.exe
Alpha.Beta is running in its own thread.
Alpha.Beta is running in its own thread.
Alpha.Beta is running in its own thread.
Alpha.Beta is running in its own thread.
Alpha.Beta is running in its own thread.
Alpha.Beta is running in its own thread.
Alpha.Beta is running in its own thread.
Alpha.Beta is running in its own thread.
Alpha.Beta is running in its own thread.
Alpha.Beta is running in its own thread.
Alpha.Beta is running in its own thread.
Alpha.Beta is running in its own thread.
Alpha.Beta is running in its own thread.
Alpha.Beta is running in its own thread.
Alpha.Beta is running in its own thread.
Alpha.Beta is running in its own thread.
Alpha.Beta is running in its own thread.
Alpha.Beta is running in its own thread.
Alpha.Beta is running in its own thread.
Alpha.Beta is running in its own thread.

Alpha.Beta has finished
Try to restart the Alpha.Beta thread
ThreadStateException trying to restart Alpha.Beta. Expected since aborted thread
s cannot be restarted.
```

在这里我们要注意的是其它线程都是依附于 Main()函数所在的线程的，Main()函数是 C#程序的入口，起始线程可以称之为主线程，如果所有的前台线程都停止了，那么主线程可以终止，而所有的后台线程都将无条件终止。而所有的线程虽然在微观上是串行执行的，但是在宏观上你完全可以认为它们在并行执行。

读者一定注意到了 Thread.ThreadState 这个属性，这个属性代表了线程运行时状态，在不同的情况下有不同的值，于是我们有时候可以通过对该值的判断来设计程序流程。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

上面提到了 Background 状态表示该线程在后台运行，那么后台运行的线程有什么特别的地方呢？其实后台线程跟前台线程只有一个区别，那就是后台线程不妨碍程序的终止。一旦一个进程所有的前台线程都终止后，CLR（通用语言运行环境）将通过调用任意一个存活中的后台进程的 Abort()方法来彻底终止进程。

当线程之间争夺 CPU 时间时，CPU 按照是线程的优先级给予服务的。在 C#应用程序中，用户可以设定 5 个不同的优先级，由高到低分别是 Highest，AboveNormal，Normal，BelowNormal，Lowest，在创建线程时如果不指定优先级，那么系统默认为 ThreadPriority.Normal。给一个线程指定优先级
，我们可以使用如下代码：

```
//设定优先级为最低
myThread.Priority=ThreadPriority.Lowest;
```

通过设定线程的优先级，我们可以安排一些相对重要的线程优先执行，例如对用户的响应等等。

现在我们对怎样创建和控制一个线程已经有了一个初步的了解，下面我们将深入研究线程实现中比较典型的的问题，并且探讨其解决方法。

三.线程的同步和通讯——生产者和消费者

假 设这样一种情况，两个线程同时维护一个队列，如果一个线程对队列中添加元素，而另外一个线程从队列中取用元素，那么我们称添加元素的线程为生产者，称取用元素的线程为消费者。生产者与消费者问题看起来很简单，但是却是多线程应用中一个必须解决的问题，它涉及到线程之间的同步和通讯问题。

前面说过，每个线程都有自己的资源，但是代码区是共享的，即每个线程都可以执行相同的函数。但是多线程环境下，可能带来的问题就是几个线程同时执行一个函数，导致数据的混乱，产生不可预料的结果，因此我们必须避免这种情况的发生。C#提供了一个关键字 lock，它可以把一段代码定义为互斥段（critical section），互斥段在一

个时刻内只允许一个线程进入执行，而其他线程必须等待。在 C#中，关键字 lock 定义如下：

```
lock(expression) statement_block
```

expression 代表你希望跟踪的对象，通常是对象引用。一般地，如果你想保护一个类的实例，你可以使用 this；如果你希望保护一个静态变量（如互斥代码段在一个静态方法内部），一般使用类名就可以了。而 statement_block 就是互斥段的代码，这段代码在一个时刻内只可能被一个线程执行。

下面是一个使用 lock 关键字的典型例子，我将在注释里向大家说明 lock 关键字的用法和用途：

```
//lock.cs
using System;
using System.Threading;

internal class Account
{
int balance;
Random r = new Random();
internal Account(int initial)
{
	balance = initial;
}

internal int Withdraw(int amount)
{
	if (balance < 0)
	{
	file://如果 balance 小于 0 则抛出异常
	throw new Exception("Negative Balance");
	}
	//下面的代码保证在当前线程修改 balance 的值完成之前
	//不会有其他线程也执行这段代码来修改 balance 的值
	//因此，balance 的值是不可能小于 0 的
	lock (this)
	{
	Console.WriteLine("Current Thread:"+Thread.CurrentThread.Name);
	file://如果没有 lock 关键字的保护，那么可能在执行完 if 的条件判断之后
	file://另外一个线程却执行了 balance=balance-amount 修改了 balance 的值
	file://而这个修改对这个线程是不可见的，所以可能导致这时 if 的条件已经不成立了
	file://但是，这个线程却继续执行 balance=balance-amount，所以导致 balance 可能小于 0
	if (balance >= amount)
	{
		Thread.Sleep(5);
		balance = balance - amount;
		return amount;
	}
	else
	{
		return 0; // transaction rejected
	}
	}
}
```

```
internal void DoTransactions()
{
    for (int i = 0; i < 100; i++)
    Withdraw(r.Next(-50, 100));
}
}

internal class Test
{
static internal Thread[] threads = new Thread[10];
public static void Main()
{
    Account acc = new Account (0);
    for (int i = 0; i < 10; i++)
    {
    Thread t = new Thread(new ThreadStart(acc.DoTransactions));
    threads[i] = t;
    }
    for (int i = 0; i < 10; i++)
    threads[i].Name=i.ToString();
    for (int i = 0; i < 10; i++)
    threads[i].Start();
    Console.ReadLine();
}
}
```

而多线程公用一个对象时，也会出现和公用代码类似的问题，这种问题就不应该使用 lock 关键字了，这里需要用到 System.Threading 中的一个类 Monitor，我们可以称之为监视器，Monitor 提供了使线程共享资源的方案。

Monitor 类可以锁定一个对象，一个线程只有得到这把锁才可以对该对象进行操作。对象锁机制保证了在可能引起混乱的情况下一个时刻只有一个线程可以访问这个对象。Monitor 必须和一个具体的对象相关联，但是由于它是一个静态的类，所以不能使用它来定义对象，而且它的所有方法都是静态的，不能使用对象来引用。下面代码说明了使用 Monitor 锁定一个对象的情形：

```
.....
Queue oQueue=new Queue();
.....
Monitor.Enter(oQueue);
.....//现在 oQueue 对象只能被当前线程操纵了
Monitor.Exit(oQueue);//释放锁
```

如上所示，当一个线程调用 Monitor.Enter()方法锁定一个对象时，这个对象就归它所有了，其它线程想要访问这个对象，只有等待它使用 Monitor.Exit()方法释放锁。为了保证线程最终都能释放锁，你可以把 Monitor.Exit()方法写在 try-catch-finally 结构中的 finally 代码块里。对于任何一个被 Monitor 锁定的对象，内存中都保存着与它相关的一些信息，其一是现在持有锁的线程的引用，其二是一个预备队列，队列中保存了已经准备好获取锁的线程，其三是一个等待队列，队列中保存着当前正在等待这个对象状态改变的队列的引用。当拥有对象锁的线程准备释放锁时，它使用 Monitor.Pulse()方法通知等待队列中的第一个线程，于是该线程被转移到预备队列中，当对象锁被释放时，在预备队列中的线程可以立即获得对象锁。

下面是一个展示如何使用 lock 关键字和 Monitor 类来实现线程的同步和通讯的例子，也是一个典型的生产者与消费者问题。这个例程中，生产者线程和消费者线程是交替进行的，生产者写入一个数，消费者立即读取并且显示，我将在注释中介绍该程序的精要所在。用到的系统命名空间如下：

```
using System;
using System.Threading;
```

首先，我们定义一个被操作的对象的类 Cell，在这个类里，有两个方法：ReadFromCell()和 WriteToCell。消费者线程将调用 ReadFromCell()读取 cellContents 的内容并且显示出来，生产者进程将调用 WriteToCell()方法向 cellContents 写入数据。

```
public class Cell
{
int cellContents; // Cell 对象里边的内容
bool readerFlag = false; //  状态标志，为 true 时可以读取，为 false 则正在写入
public int ReadFromCell( )
{
    lock(this) // Lock 关键字保证了什么，请大家看前面对 lock 的介绍
    {
```

```
        if (!readerFlag)//如果现在不可读取
        {
            try
            {
            file://等待 WriteToCell 方法中调用 Monitor.Pulse()方法
            Monitor.Wait(this);
            }
            catch (SynchronizationLockException e)
            {
            Console.WriteLine(e);
            }
            catch (ThreadInterruptedException e)
            {
            Console.WriteLine(e);
            }
        }
        Console.WriteLine("Consume: {0}",cellContents);
        readerFlag = false; file://重置 readerFlag 标志，表示消费行为已经完成
        Monitor.Pulse(this); file://通知 WriteToCell()方法（该方法在另外一个线程中执
行，等待中）
        }
        return cellContents;
    }

    public void WriteToCell(int n)
    {
        lock(this)
        {
        if (readerFlag)
        {
            try
            {
            Monitor.Wait(this);
            }
            catch (SynchronizationLockException e)
            {
            file://当同步方法（指 Monitor 类除 Enter 之外的方法）在非同步的代码区
被调用
            Console.WriteLine(e);
            }
            catch (ThreadInterruptedException e)
            {
            file://当线程在等待状态的时候中止
            Console.WriteLine(e);
```

```
            }
        }
        cellContents = n;
        Console.WriteLine("Produce: {0}",cellContents);
        readerFlag = true;
        Monitor.Pulse(this); file://通知另外一个线程中正在等待的 ReadFromCell()方法
        }
    }
    }
```

下面定义生产者 CellProd 和消费者类 CellCons，它们都只有一个方法 ThreadRun()，以便在 Main()函数中提供给线程的 ThreadStart 代理对象，作为线程的入口。

```
    public class CellProd
    {
    Cell cell; //  被操作的 Cell 对象
    int quantity = 1; //  生产者生产次数，初始化为 1

    public CellProd(Cell box, int request)
    {
        //构造函数
        cell = box;
        quantity = request;
    }
    public void ThreadRun( )
    {
        for(int looper=1; looper<=quantity; looper++)
        cell.WriteToCell(looper); file://生产者向操作对象写入信息
    }
    }

    public class CellCons
    {
    Cell cell;
    int quantity = 1;

    public CellCons(Cell box, int request)
    {
        cell = box;
        quantity = request;
    }
```

```
public void ThreadRun( )
{
    int valReturned;
    for(int looper=1; looper<=quantity; looper++)
    valReturned=cell.ReadFromCell( );//消费者从操作对象中读取信息
}
}
```

然后在下面这个类 MonitorSample 的 Main()函数中我们要做的就是创建两个线程分别作为生产者和消费者，使用 CellProd.ThreadRun()方法和 CellCons.ThreadRun()方法对同一个 Cell 对象进行操作。

```
public class MonitorSample
{
public static void Main(String[] args)
{
    int result = 0; file://一个标志位，如果是 0 表示程序没有出错，如果是 1 表明有错误发生
    Cell cell = new Cell( );

    //下面使用 cell 初始化 CellProd 和 CellCons 两个类，生产和消费次数均为 20 次
    CellProd prod = new CellProd(cell, 20);
    CellCons cons = new CellCons(cell, 20);

    Thread producer = new Thread(new ThreadStart(prod.ThreadRun));
    Thread consumer = new Thread(new ThreadStart(cons.ThreadRun));
    //生产者线程和消费者线程都已经被创建，但是没有开始执行

    try
    {
```

```
        producer.Start( );
        consumer.Start( );

        producer.Join( );
        consumer.Join( );
        Console.ReadLine();
        }
        catch (ThreadStateException e)
        {
        file://当线程因为所处状态的原因而不能执行被请求的操作
        Console.WriteLine(e);
        result = 1;
        }
        catch (ThreadInterruptedException e)
        {
        file://当线程在等待状态的时候中止
        Console.WriteLine(e);
        result = 1;
        }
        //尽管 Main()函数没有返回值，但下面这条语句可以向父进程返回执行结果
        Environment.ExitCode = result;
    }
    }
```

大家可以看到，在上面的例程中，同步是通过等待 Monitor.Pulse()来完成的。首先生产者生产了一个值，而同一时刻消费者处于等待状态，直到收到生产者的“脉冲(Pulse)”通知它生产已经完成，此后消费者进入消费状态，而生产者开始等待消费者完成操作后将调用 Monitor.Pulese()发出的“脉冲”。它的执行结果很简单：

```
Produce: 1
Consume: 1
Produce: 2
Consume: 2
Produce: 3
Consume: 3
...
...
Produce: 20
Consume: 20
```

事实上，这个简单的例子已经帮助我们解决了多线程应用程序中可能出现的大问题，只要领悟了解决线程间冲突的基本方法，很容易把它应用到比较复杂的程序中去。

四、线程池和定时器——多线程的自动管理

在多线程的程序中，经常会出现两种情况。一种情况下，应用程序中的线程把大部分的时间花费在等待状态，等待某个事件发生，然后才能给予响应；而另外一种情况则是线程平常都处于休眠状态，只是周期性地被唤醒。在.net framework 里边，我们使用 ThreadPool 来对付第一种情况，使用 Timer 来对付第二种情况。

ThreadPool 类提供一个由系统维护的线程池——可以看作一个线程的容器，该容器需要 Windows 2000 以上版本的系统支持，因为其中某些方法调用了只有高版本的 Windows 才有的 API 函数。你可以使用 ThreadPool.QueueUserWorkItem()方法将线程安放在线程池里，该方法的原型如下：

```
//将一个线程放进线程池，该线程的 Start()方法将调用 WaitCallback 代理对象代表的函数
public static bool QueueUserWorkItem(WaitCallback);
//重载的方法如下，参数 object 将传递给 WaitCallback 所代表的方法
public static bool QueueUserWorkItem(WaitCallback, object);
```

要注意的是，ThreadPool 类也是一个静态类，你不能也不必要生成它的对象，而且一旦使用该方法在线程池中添加了一个项目，那么该项目将是没有办法取消的。在这里你无需自己建立线程，只需把你要做的工作写成函数，然后作为参数传递给 ThreadPool.QueueUserWorkItem()方法就行了，传递的方法就是依靠 WaitCallback 代理对象，而线程的建立、管理、运行等等工作都是由系统自动完成的，你无须考虑那些复杂的细节问题，线程池的优点也就在这里体现出来了，就好像你是公司老板——只需要安排工作，而不必亲自动手。

下面的例程演示了 ThreadPool 的用法。首先程序创建了一个 ManualResetEvent 对象，该对象就像一个信号灯，可以利用它的信号来通知其它线程，本例中当线程池中所有线程工作都完成以后，ManualResetEvent 的对象将被设置为有信号，从而通知主线程继续运行。它有几个重要的方法：Reset()，Set()，WaitOne()。初始化该对象时，用户可以指定其默认的状态（有信号/无信号），在初始化以后，该对象将保持原来的状态不变直到它的 Reset()或者 Set()方法被调用，Reset()方法将其设置为无信号状态，Set()方法将其设置为有信号状态。WaitOne()方法使当前线程挂起直到 ManualResetEvent 对象处于有信号状态，此时该线程将被激活。然后，程序将向线程池中添加工作项，这些以函数形式提供的工作项被系统用来初始化自动建立的线程。当所有的线程都运行完了以后，ManualResetEvent.Set()方法被调用，因为调用了 ManualResetEvent.WaitOne()方法而处在等待状态的主线程将接收到这个信号，于是它接着往下执行，完成后边的工作。

```
using System;
using System.Collections;
using System.Threading;

//这是用来保存信息的数据结构，将作为参数被传递
```

```
public class SomeState
{
public int Cookie;
public SomeState(int iCookie)
{
    Cookie = iCookie;
}
}

public class Alpha
{
public Hashtable HashCount;
public ManualResetEvent eventX;
public static int iCount = 0;
public static int iMaxCount = 0;
public Alpha(int MaxCount)
{
    HashCount = new Hashtable(MaxCount);
    iMaxCount = MaxCount;
}

file://线程池里的线程将调用 Beta()方法
public void Beta(Object state)
{
    //输出当前线程的 hash 编码值和 Cookie 的值
    Console.WriteLine(" {0} {1} :", Thread.CurrentThread.GetHashCode(),
    ((SomeState)state).Cookie);
    Console.WriteLine("HashCount.Count={0},
Thread.CurrentThread.GetHashCode()={1}", HashCount.Count,
Thread.CurrentThread.GetHashCode());
    lock (HashCount)
    {
    file://如果当前的 Hash 表中没有当前线程的 Hash 值，则添加之
    if (!HashCount.ContainsKey(Thread.CurrentThread.GetHashCode()))
        HashCount.Add (Thread.CurrentThread.GetHashCode(), 0);
    HashCount[Thread.CurrentThread.GetHashCode()] =
((int)HashCount[Thread.CurrentThread.GetHashCode()])+1;
    }

    int iX = 2000;
    Thread.Sleep(iX);
    //Interlocked.Increment()操作是一个原子操作，具体请看下面说明
    Interlocked.Increment(ref iCount);
    if (iCount == iMaxCount)
```

```
            {
            Console.WriteLine();
            Console.WriteLine("Setting eventX ");
            eventX.Set();
            }
        }
        }

        public class SimplePool
        {
        public static int Main(string[] args)
        {
            Console.WriteLine("Thread Pool Sample:");
            bool W2K = false;
            int MaxCount = 10;//允许线程池中运行最多 10 个线程
            //新建 ManualResetEvent 对象并且初始化为无信号状态
            ManualResetEvent eventX = new ManualResetEvent(false);
            Console.WriteLine("Queuing {0} items to Thread Pool", MaxCount);
            Alpha oAlpha = new Alpha(MaxCount); file://创建工作项
            //注意初始化 oAlpha 对象的 eventX 属性
            oAlpha.eventX = eventX;
            Console.WriteLine("Queue to Thread Pool 0");
            try
            {
            file://将工作项装入线程池
            file://这里要用到 Windows 2000 以上版本才有的 API，所以可能出现
NotSupportException 异常
            ThreadPool.QueueUserWorkItem(new WaitCallback(oAlpha.Beta),
            new SomeState(0));
            W2K = true;
            }
            catch (NotSupportedException)
            {
            Console.WriteLine("These API's may fail when called on a non-Windows
2000 system.");
            W2K = false;
            }
            if (W2K)//如果当前系统支持 ThreadPool 的方法.
            {
            for (int iItem=1;iItem < MaxCount;iItem++)
            {
                //插入队列元素
                Console.WriteLine("Queue to Thread Pool {0}", iItem);
                ThreadPool.QueueUserWorkItem(new WaitCallback(oAlpha.Beta),new
```

```
SomeState(iItem));
            }
            Console.WriteLine("Waiting for Thread Pool to drain");
            file://等待事件的完成，即线程调用 ManualResetEvent.Set()方法
            eventX.WaitOne(Timeout.Infinite,true);
            file://WaitOne()方法使调用它的线程等待直到 eventX.Set()方法被调用
            Console.WriteLine("Thread Pool has been drained (Event fired)");
            Console.WriteLine();
            Console.WriteLine("Load across threads");
            foreach(object o in oAlpha.HashCount.Keys)
            Console.WriteLine("{0} {1}", o, oAlpha.HashCount[o]);
            }
            Console.ReadLine();
            return 0;

        }
        }
```

程序中有些小地方应该引起我们的注意。SomeState 类是一个保存信息的数据结构，在上面的程序中，它作为参数被传递给每一个线程，你很容易就能理解这个，因为你需要把一些有用的信息封装起来提供给线程，而这种方式是非常有效的。程序出现的 InterLocked 类也是专为多线程程序而存在的，它提供了一些有用的原子操作，所谓原子操作就是在多线程程序中，如果这个线程调用这个操作修改一个变量，那么其他线程就不能修改这个变量了，这跟 lock 关键字在本质上是一样的。

我们应该彻底地分析上面的程序，把握住线程池的本质，理解它存在的意义是什么，这样我们才能得心应手地使用它。下面是该程序的输出结果：

```
Thread Pool Sample:
Queuing 10 items to Thread Pool
Queue to Thread Pool 0
Queue to Thread Pool 1
...
...
Queue to Thread Pool 9
Waiting for Thread Pool to drain
98 0 :
HashCount.Count==0, Thread.CurrentThread.GetHashCode()==98
100 1 :
HashCount.Count==1, Thread.CurrentThread.GetHashCode()==100
98 2 :
...
...
```

```
Setting eventX
Thread Pool has been drained (Event fired)
Load across threads
101 2
100 3
98 4
102 1
```

与 ThreadPool 类不同，Timer 类的作用是设置一个定时器，定时执行用户指定的函数，而这个函数的传递是靠另外一个代理对象 TimerCallback，它必须在创建 Timer 对象时就指定，并且不能更改。定时器启动后，系统将自动建立一个新的线程，并且在这个线程里执行用户指定的函数。下面的语句初始化了一个 Timer 对象：

```
Timer timer = new Timer(timerDelegate, s,1000, 1000);
```

第一个参数指定了 TimerCallback 代理对象；第二个参数的意义跟上面提到的 WaitCallback 代理对象的一样，作为一个传递数据的对象传递给要调用的方法；第三个参数是延迟时间——计时开始的时刻距现在的时间，单位是毫秒；第四个参数是定时器的时间间隔——计时开始以后，每隔这么长的一段时间，TimerCallback 所代表的方法将被调用一次，单位也是毫秒。这句话的意思就是将定时器的延迟时间和时间间隔都设为 1 秒钟。

定时器的设置是可以改变的，只要调用 Timer.Change()方法，这是一个参数类型重载的方法，一般使用的原型如下：

```
public bool Change(long, long);
```

下面这段代码将前边设置的定时器修改了一下：

```
timer.Change(10000,2000);
```

很显然，定时器 timer 的时间间隔被重新设置为 2 秒，停止计时 10 秒后生效。

下面这段程序演示了 Timer 类的用法。

```
using System;
using System.Threading;
class TimerExampleState
```

```
    {
    public int counter = 0;
    public Timer tmr;
    }

    class App
    {
    public static void Main()
    {
        TimerExampleState s = new TimerExampleState();

        //创建代理对象 TimerCallback，该代理将被定时调用
        TimerCallback timerDelegate = new TimerCallback(CheckStatus);

        //创建一个时间间隔为 1s 的定时器
        Timer timer = new Timer(timerDelegate, s,1000, 1000);
        s.tmr = timer;

        //主线程停下来等待 Timer 对象的终止
        while(s.tmr != null)
        Thread.Sleep(0);
        Console.WriteLine("Timer example done.");
        Console.ReadLine();
    }
    file://下面是被定时调用的方法

    static void CheckStatus(Object state)
    {
        TimerExampleState s =(TimerExampleState)state;
        s.counter++;
        Console.WriteLine("{0} Checking Status {1}.",DateTime.Now.TimeOfDay,
s.counter);
        if(s.counter == 5)
        {
        file://使用 Change 方法改变了时间间隔
        (s.tmr).Change(10000,2000);
        Console.WriteLine("changed...");
        }
        if(s.counter == 10)
        {
        Console.WriteLine("disposing of timer...");
        s.tmr.Dispose();
        s.tmr = null;
        }
```

```
    }
    }
```

程序首先创建了一个定时器，它将在创建 1 秒之后开始每隔 1 秒调用一次 CheckStatus()方法，当调用 5 次以后，在 CheckStatus()方法中修改了时间间隔为 2 秒，并且指定在 10 秒后重新开始。当计数达到 10 次，调用 Timer.Dispose()方法删除了 timer 对象，主线程于是跳出循环，终止程序。程序执行的结果如下：

```
F:\Visual Studio Projects\ConsoleApplication4\bin\Debug\ThreadTest
Mutex Sample ...
 - Main Owns gM1 and gM2
t1Start started,  Mutex.WaitAll(Mutex[])
t2Start started,  gM1.WaitOne( )
t3Start started,  Mutex.WaitAny(Mutex[])
t4Start started,  gM2.WaitOne( )
 - Main releases gM1
t3Start finished, Mutex.WaitAny(Mutex[])
t2Start finished, gM1.WaitOne( ) satisfied
 - Main releases gM2
t4Start finished, gM2.WaitOne( )
t1Start finished, Mutex.WaitAll(Mutex[]) satisfied
... Mutex Sample
```

上面就是对 ThreadPool 和 Timer 两个类的简单介绍，充分利用系统提供的功能，可以为我们省去很多时间和精力——特别是对很容易出错的多线程程序。同时我们也可以看到.net Framework 强大的内置对象，这些将对我们的编程带来莫大的方便。

、互斥对象——更加灵活的同步方式

有 时候你会觉得上面介绍的方法好像不够用，对，我们解决了代码和资源的同步问题，解决了多线程自动化管理和定时触发的问题，但是如何控制多个线程相互之间的联系呢？例如我要到餐厅吃饭，在吃饭之前我先得等待厨师把饭菜做好，之后我开始吃饭，吃完我还得付款，付款方式可以是现金，也可以是信用卡，付款之后我才 能离开。分析一下这个过程，我吃饭可以看作是主线程，厨师做饭又是一个线程，服务员用信用卡收款和收现金可以看作另外两个线程，大家可以很清楚地看到其中 的关系——我吃饭必须等待厨师做饭，然后等待两个收款线程之中任意一个的完成，然后我吃饭这个线程可以执行离开这个步骤，于是我吃饭才算结束了。事实上，现实中有着比这更复杂的联系，我们怎样才能很好地控制它们而不产生冲突和重复呢？

这种情况下，我们需要用到互斥对象，即 System.Threading 命名空间中的 Mutex 类。大家一定坐过出租车吧，事实上我们可以把 Mutex 看作一个出租车，那么乘客就是

线程了，乘客首先得等车，然后上车，最后下车，当一个乘客在车上时，其他乘客就只有等他下车以后才可以上车。而线程与 Mutex 对象的关系也正是如此，线程使用 Mutex.WaitOne()方法等待 Mutex 对象被释放，如果它等待的 Mutex 对象被释放了，它就自动拥有这个对象，直到它调用 Mutex.ReleaseMutex()方法释放这个对象，而在此期间，其他想要获取这个 Mutex 对象的线程都只有等待。

下面这个例子使用了 Mutex 对象来同步四个线程，主线程等待四个线程的结束，而这四个线程的运行又是与两个 Mutex 对象相关联的。其中还用到 AutoResetEvent 类的对象，如同上面提到的 ManualResetEvent 对象一样，大家可以把它简单地理解为一个信号灯，使用 AutoResetEvent.Set()方法可以设置它为有信号状态，而使用 AutoResetEvent.Reset()方法把它设置为无信号状态。这里用它的有信号状态来表示一个线程的结束。

```
// Mutex.cs
using System;
using System.Threading;

public class MutexSample
{
static Mutex gM1;
static Mutex gM2;
const int ITERS = 100;
static AutoResetEvent Event1 = new AutoResetEvent(false);
static AutoResetEvent Event2 = new AutoResetEvent(false);
static AutoResetEvent Event3 = new AutoResetEvent(false);
static AutoResetEvent Event4 = new AutoResetEvent(false);

public static void Main(String[] args)
{
    Console.WriteLine("Mutex Sample ...");
    //创建一个 Mutex 对象，并且命名为 MyMutex
    gM1 = new Mutex(true,"MyMutex");
    //创建一个未命名的 Mutex  对象.
    gM2 = new Mutex(true);
    Console.WriteLine(" - Main Owns gM1 and gM2");

    AutoResetEvent[] evs = new AutoResetEvent[4];
    evs[0] = Event1; file://为后面的线程 t1,t2,t3,t4 定义 AutoResetEvent 对象
    evs[1] = Event2;
    evs[2] = Event3;
    evs[3] = Event4;

    MutexSample tm = new MutexSample( );
```

```
        Thread t1 = new Thread(new ThreadStart(tm.t1Start));
        Thread t2 = new Thread(new ThreadStart(tm.t2Start));
        Thread t3 = new Thread(new ThreadStart(tm.t3Start));
        Thread t4 = new Thread(new ThreadStart(tm.t4Start));
        t1.Start( );//  使用 Mutex.WaitAll()方法等待一个 Mutex 数组中的对象全部被释放
        t2.Start( );//  使用 Mutex.WaitOne()方法等待 gM1 的释放
        t3.Start( );//  使用 Mutex.WaitAny()方法等待一个 Mutex 数组中任意一个对象被释放
        t4.Start( );//  使用 Mutex.WaitOne()方法等待 gM2 的释放


        Thread.Sleep(2000);
        Console.WriteLine(" - Main releases gM1");
        gM1.ReleaseMutex( ); file://线程 t2,t3 结束条件满足

        Thread.Sleep(1000);
        Console.WriteLine(" - Main releases gM2");
        gM2.ReleaseMutex( ); file://线程 t1,t4 结束条件满足

        //等待所有四个线程结束
        WaitHandle.WaitAll(evs);
        Console.WriteLine("... Mutex Sample");
        Console.ReadLine();
    }

    public void t1Start( )
    {
        Console.WriteLine("t1Start started, Mutex.WaitAll(Mutex[])");
        Mutex[] gMs = new Mutex[2];
        gMs[0] = gM1;//创建一个 Mutex 数组作为 Mutex.WaitAll()方法的参数
        gMs[1] = gM2;
        Mutex.WaitAll(gMs);//等待 gM1 和 gM2 都被释放
        Thread.Sleep(2000);
        Console.WriteLine("t1Start finished, Mutex.WaitAll(Mutex[]) satisfied");
        Event1.Set( ); file://线程结束，将 Event1 设置为有信号状态
    }

    public void t2Start( )
    {
        Console.WriteLine("t2Start started, gM1.WaitOne( )");
        gM1.WaitOne( );//等待 gM1 的释放
        Console.WriteLine("t2Start finished, gM1.WaitOne( ) satisfied");
        Event2.Set( );//线程结束，将 Event2 设置为有信号状态
```

```
    }

    public void t3Start( )
    {
        Console.WriteLine("t3Start started, Mutex.WaitAny(Mutex[])");
        Mutex[] gMs = new Mutex[2];
        gMs[0] = gM1;//创建一个 Mutex 数组作为 Mutex.WaitAny()方法的参数
        gMs[1] = gM2;
        Mutex.WaitAny(gMs);//等待数组中任意一个 Mutex 对象被释放
        Console.WriteLine("t3Start finished, Mutex.WaitAny(Mutex[])");
        Event3.Set( );//线程结束，将 Event3 设置为有信号状态
    }

    public void t4Start( )
    {
        Console.WriteLine("t4Start started, gM2.WaitOne( )");
        gM2.WaitOne( );//等待 gM2 被释放
        Console.WriteLine("t4Start finished, gM2.WaitOne( )");
        Event4.Set( );//线程结束，将 Event4 设置为有信号状态
    }
    }
```

下面是该程序的执行结果：

```
F:\Visual Studio Projects\ConsoleApplication4\bin\Debug\ThreadTest
20:15:00.8090640 Checking Status 1.
20:15:01.8004896 Checking Status 2.
20:15:02.8019296 Checking Status 3.
20:15:03.8133840 Checking Status 4.
20:15:04.8148240 Checking Status 5.
changed...
20:15:14.8292240 Checking Status 6.
20:15:14.9293680 Checking Status 7.
20:15:15.0295120 Checking Status 8.
20:15:15.1296560 Checking Status 9.
20:15:15.2298000 Checking Status 10.
disposing of timer...
Timer example done.
```

从执行结果可以很清楚地看到，线程 t2,t3 的运行是以 gM1 的释放为条件的，而 t4 在 gM2 释放后开始执行，t1 则在 gM1 和 gM2 都被释放了之后才执行。Main()函数最后，使用 WaitHandle 等待所有的 AutoResetEvent 对象的信号，这些对象的信号代表相应线程的结束。

六、小结

多线程程序设计是一个庞大的主题，而本文试图在.net Framework 环境下，使用最新的 C#语言来描述多线程程序的概貌。希望本文能有助于大家理解线程这种概念，理解多线程的用途，理解它的 C#实现方法，理解线程将为我们带来的好处和麻烦。C#是一种新的语言，因此它的线程机制也有许多独特的地方，希望大家能通过本文清楚地看到这些，从而可以对线程进行更深入的理解和探索。

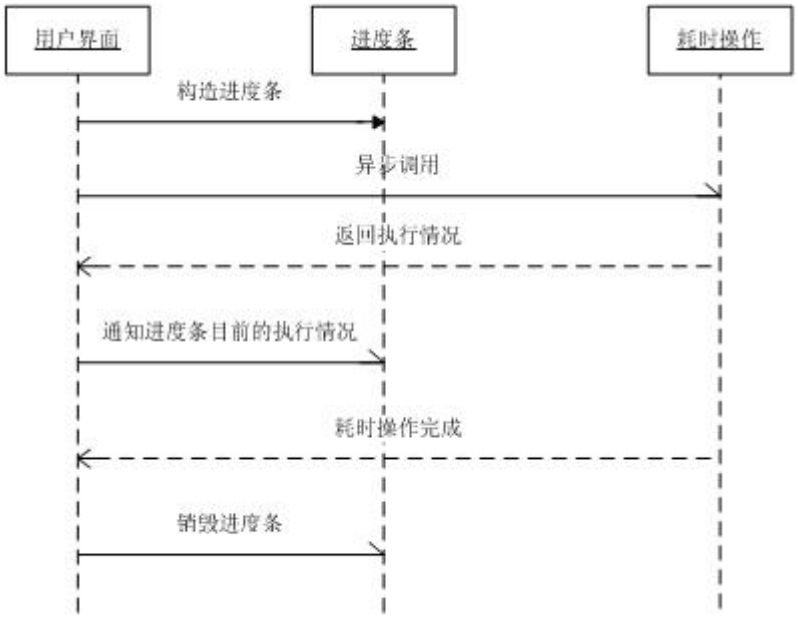
打造迅速响应的用户界面

背景

最近抽时间开发了一个生成SQL脚本和执行脚本的 小工具，基本已经完成，但由于生成脚本和执行脚本相对而言是比较耗时的操作，所以原先的单线程模式会短暂的冻结用户界面，由此，为了更好的用户体验，针对这两个耗时的操作引进了多线程模式。我的目标是给这两个操作添加对应的进度条（与用户界面处在不同的Form），显示目前的进度情况，及时把脚本的执行情况反馈给用户。下面，我就依托这个小小的背景，谈一下自己的见解吧。

需要做的工作

首先，需要做的事情是，搞清楚用户界面、进度条以及耗时的操作之间的关系，可以用 UML 的序列图表示，如下图所示：



从图中已经可以清晰地看到它们三者之间的关系，不过我还是要简单的叙述一下：

1) 用户界面构造并显示进度条。

2) 用户界面异步调用耗时操作。

3) 耗时操作需要及时地把执行情况反馈给用户界面。

4) 用户界面把目前的执行情况通知给进度条。

5) 耗时操作完成时，通知用户界面。

6) 用户界面销毁进度条，提示任务执行完毕。

明确了做什么之后，下一步就可以思考有哪些方法可以帮助我们解决这个问题了。

有哪些方法

不管采取哪些方法，你都需要思考如下几个问题：

1) 用户界面和耗时操作如何异步调用？

2) 耗时操作如果需要参数，那么如何传递参数？

3) 耗时操作如何把当前的执行情况发送给用户界面？

4) 如果耗时操作有返回值（分为每一个小的耗时操作的返回值和整个耗时操作的返回值），那么如何把执行的结果返回给用户界面？

5) 用户界面如何把执行的情况反馈给进度条？

如果这些问题你都已经考虑清楚了，那么这里要解决的问题也就不再是问题了。让我们逐个解答吧。

用户界面和耗时操作如何异步调用？

针对这个问题，我想解决办法有两种，一是自己利用多线程的相关类构建工作者线程，把耗时操作分配给工作者线程，然后在 UI 线程里开启工作者线程。第二种方法，使用异步执行委托，这是在 UI 线程里处理此类问题所特有的方法，使用此方法免去了很多代码，简单好用。

耗时操作如果需要参数，那么如何传递参数？

这个问题的答案就跟你上面选择的方式有关系了，如果你采用的是自己构建多线程，并且需要传递给工作者线程一些参数，你可以采取的办法有：

- 通过耗时操作的公开属性给其传值。
- 通过耗时操作类的构造函数的参数传值。
- 借助于第三方给其传值，如：你可以把耗时操作需要的数据预先存储到一个文件里或者数据库里。
- 混合使用以上的方法。

如果你采用的是异步执行委托的方式，除了以上的办法外，你还可以采取的办法有：

- 通过 **BeginInvoke** 传给耗时操作所需要的数据。

借助这些方法均可以帮助你顺利地把参数传递给耗时操作，我想说的是，你的需求决定了你的办法，你的办法决定了你的代码量以及性能，所以，要三思而后行。

耗时操作如何把当前的执行情况发送给用户界面？

由于耗时操作与 UI 处于不同的线程，所以 UI 不能直接得到耗时操作的执行情况，在这里，我们可以利用的办法有：

- 利用耗时操作的事件，把执行情况发送给事件处理者（用户界面）。
- 借助于第三方，把耗时操作的执行情况存储到一个文件里或者数据库里，供用户界面获取。
- 构建消息通道，通过消息通道，把消息发送给对消息感兴趣的接收者。

如何把执行的结果返回给用户界面？

这个问题与上一问题的解决方式差不多，只不过，如果用户界面采用的是异步委托执行的方式与耗时操作衔接的话，可以使用另外一种接收返回的执行结果，设置 **BeginInvoke** 的回调函数，然后在回调函数里调用 **EndInvoke** 接收耗时操作返回的结果。

用户界面如何把执行的情况反馈给进度条？

当用户界面得到执行的情况后，它需要把这些信息传给进度条，通过进度条，向用户展示目前耗时操作的执行情况，那么，用户界面是如何把执行的情况反馈给进度条呢？

在这里，我们需要判断用户界面与进度条，是否处在同一个线程里（使用 InvokeRequired

判断），决定是采用 Invoke 的方式，还是直接访问。

到这里，我们已经把所有面临的问题逐步剖析了一遍，相信你已经对类似的问题有了自己的解决办法。剩下的事情就是如何在实际的代码中运用这些方法了。

如何解决我的问题？

接下来，针对我的脚本辅助工具中遇到的问题，做一个阐述。由于我的工具里有两个比较耗时的操作，而解决它们的办法都较类似，所以，我就以脚本的产生为例子，说明一下我的解决之道。

首先，定义一个委托，用于异步执行耗时操作即产生脚本，代码如下：

```
private delegate string CreateSql(ArrayList list,BuildSQL.DataBase.SQLServer server);
```

然后，我需要做的是：构建进度条，启动耗时操作。在这里，我采用的是异步执行委托的方式启动耗时操作的，而我的耗时操作：产生脚本会在结束的时候返回产生的脚本。代码如下：

```
privatevoid StartCreateSql(ArrayList list,BuildSQL.DataBase.SQLServer server)
        {
            //禁用产生脚本按钮
            this.btnCreateSql.Enabled = false;
            //启动进度条
            bar = new ProgressBar();
            bar.Title = "正在生成脚本…";
            bar.StartPosition = FormStartPosition.CenterParent;
            bar.Show();
            bar.ShowProgress(list.Count,0);
            //开始产生脚本
```

```
            BuildSQL.DTO.SQLServer2000.SQLScript sqlScript = new BuildSQL.DTO.SQLSer
ver2000.SQLScript();
//绑定事件，获取耗时操作执行的情况
            sqlScript.ProgressChangeEventHandler+=new BuildSQL.DTO.SQLServer2000.SQ
LScript.ProgressDelegate(sqlScript_ProgressChangeEventHandler);
            CreateSql sql = new CreateSql(sqlScript.CreateSQLScript);
//异步执行委托，并绑定回调函数 CreateSqlFinish，目的是在回调函数里获取产生的脚本
            sql.BeginInvoke(list,server,new AsyncCallback(CreateSqlFinish),sql);
        }
```

为了获取耗时操作的执行情况，我采用了事件机制，封装在耗时操作类里的代码为：

```
//定义进度的委托和事件
public  delegate  void ProgressDelegate(object sender,ProgressEventArgs e);
public  event ProgressDelegate ProgressChangeEventHandler;

//触发事件
public  int Now
        {
            get{return m_now;}
            set
            {
                m_now = value;
                //触发事件
                if(ProgressChangeEventHandler != null)
                {
                    ProgressEventArgs e = new ProgressEventArgs(m_max,value);
                    ProgressChangeEventHandler(this,e);
                }
                //任务完成事件
```

```
            if(value == m_max)
            {
                if(ProgressFinishEventHandler != null)
                    ProgressFinishEventHandler(this, new EventArgs());
            }
        }
    }
```

用户界面接收耗时操作的执行情况，并把这些信息发送给进度条，代码如下：

```
 privatevoid sqlScript_ProgressChangeEventHandler(object sender, BuildSQL.DTO.SQLSe
rver2000.ProgressEventArgs e)
    {
    if(InvokeRequired)
      {
      ShowProgressDelegate show = new ShowProgressDelegate(bar.ShowProgress);
      show.BeginInvoke(e.Max, e.Now, null, null);
    }
    else
      {
      bar.ShowProgress(e.Max, e.Now);
    }
  }
```

最后，当耗时操作结束的时候，获取其执行的结果，在这里获取的是产生的脚本，如下：

```
 private void CreateSqlFinish(IAsyncResult result)
    {
    if(bar != null)
      {
```

```
            bar.Close();
        }
        if(MessageBox.Show("脚本产生成功！") == DialogResult.OK)
        {
            this.txtScript.Text = ((CreateSql)result.AsyncState).EndInvoke(result);
            //启用产生脚本按钮
            this.btnCreateSql.Enabled = true;
        }
    }
```

到此，我的这个工具中所要解决的问题已经解决了。在这里，我想提醒的是，你不必关注代码的细节，而应该关注代码的骨架。当然这些代码如果想运用在其它地方，可能需要重新编写，我想，如果你愿意，那么下一步考虑的事情应该是如何把这些代码抽取到一个类中，更容易的应用到类似这样的需求里。

最后，给大家推荐一下我的这个 脚本辅助工具，希望它也可以给你的工作带来便利。文中如果有不对的地方，还请大家批评指正。

备注：如果你需要了解更多的信息，你可以阅读msdn的相关文章：通过多线程为基于 .NET 的应用程序实现响应迅速的用户

多线程在Visual C#网络编程中的应用

admin 发表于：04-05-18

网络应用程序的一般都会或多或少的使用到线程，甚至可以说，一个功能稍微强大的网络应用程序总会在其中开出或多或少的线程，如果应用程序中开出的线程数目大于二个，那么就可以把这个程序称之为多线程应用程序。那么为什么在网络应用程序总会和线程交缠在一起呢？这是因为网络应用程序在执行的时候，会遇到很多意想不到的问题，其中最常见的是网络阻塞和网络等待等。

程序在处理这些问题的时候往往需要花费很多的时间，如果不使用线程，则程序在执行时的就会表现出如运行速度慢，执行时间长，容易出现错误、反应迟钝等问题。而如果把这些可能造成大量占用程序执行时间的过程放在线程中处理，就往往能够大大提高应用程序的运行效率和性能和获得更优良的可伸缩性。那么这是否就意味着应该在网络应用程序中广泛的使用线程呢？情况并非如此，线程其实是一把双刃剑，如果不分场合，在不需要使用的地方强行使用就可能会产生许多程序垃圾，或者在程序结束后，由于没有能够销毁创建的进程而导致应用程序挂起等问题。

所以如果你认为自己编写的代码足够快，那我给你的建议还是别使用线程或多线程。这里要提醒诸位的是如果您对在Windows下的线程和其执行原理和机制还不十分清楚，可以先参阅一下介绍Windows操作系统方面的书籍，它们一般都会对其进行比较详细的阐述。然后再阅读本文。

一.简介在Visual C#中创建和使用线程：

Visual C#中使用的线程都是通过自命名空间System.Threading中的Thread类经常实例化完成的。通过Thread类的构造函数来创建可供Visual C#使用的线程，通过Thread中的方法和属性来设定线程属性和控制线程的状态。以下Thread类中的最典型的构造函数语法，在Visual C#中一般使用这个构造函数来创建、初始化Thread实例。

```
public Thread (
  ThreadStart start
) ;
```

参数

start ThreadStart 委托，它将引用此线程开始执行时要调用的方法。

Thread还提供了其他的构造函数来创建线程，这里就不一一介绍了。表01是Thread类中的一些常用的方法及其简要说明：

方法	说明
Abort	调用此方法通常会终止线程，但会引起ThreadAbortException类型异常。
	中断处于WaitSleepJoin 线程状态的线程。

Interrupt	
Join	阻塞调用线程，直到某个线程终止时为止。
ResetAbort	取消当前线程调用的 Abor 方法。
Resume	继续已挂起的线程。
Sleep	当前线程阻塞指定的毫秒数。
Start	操作系统将当前实例的状态更改为 ThreadState.Running。
Suspend	挂起线程，或者如果线程已挂起，则不起作用。

表 01：Thread 类的常用方法及其说明

这里要注意的是在.Net 中执行一个线程，当线程执行完毕后，一般会自动销毁。如果线程没有自动销毁可通过 Thread 中的 Abort 方法来手动销毁，但同样要注意的是如果线程中使用的资源没有完全销毁，Abort 方法执行后，也不能保证线程被销毁。在 Thread 类中还提供了一些属性用以设定和获取创建的 Thread 实例属性，表 02 中是 Thread 类的一些常用属性及其说明：

属性	说明
CurrentCulture	获取或设置当前线程的区域性。
CurrentThread	获取当前正在运行的线程。
IsAlive	获取一个值，该值指示当前线程的执行状态。
	获取或设置一个值，该值指示某个线程是否为后台线程。

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

表 02：Thread 类的常用属性及其说明

二.本文的主要内容及程序调试和运行环境：

本文的主要内容是介绍多线程给用 Visual C＃编写网络应用程序带来的更高性能提高。具体的做法是在 Visual C＃用二种不同的方法，一种采用了多线程，另一种不是，来实现同一个具体网络应用示例，此示例的功能是获取网络同一网段多个 IP 地址对应的计算机的在线状态和对应的计算机名称，通过比较这二种方法的不同执行效率就可知多线程对提高网络应用程序的执行效率是多么的重要了。以下是本文中设计到程序的调试和运行的基本环境配置：

（1）.微软公司视窗 2000 服务器版。

（2）.Visual Studio .Net 2002 正式版，.Net FrameWork SDK 版本号 3705。

三．扫描网络计算机的原理：

下面介绍的这个示例的功能是通过扫描一个给定区间 IP 地址，来判断这些 IP 地址对应的计算机是否在线，如果在线则获得 IP 地址对应的计算机名称。程序判断计算机是否在线的是采用对给定 IP 地址的计算机进行 DNS 解析，如果能够根据 IP 地址解析出对应的计算机名称，则说明此 IP 地址对应的计算机在线；反之，如果解析不出，则会产生异常出错，通过对异常的捕获，得到此 IP 地址对应的计算机并不在线。

为了更清楚地说明问题和便于掌握在 Visual C＃编写多线程网络应用程序的方法，本文首先介绍的是不基于多线程的网络计算机扫描程序的编写步骤，然后再在其基础上，把它修改成多线程的计算机扫描程序，最后比较这二个程序的执行效率，你就会发现线程在网络编程中的重要作用了。

四．Visual C＃实现不基于多线程的网络计算机扫描程序

以下是在 Visual C＃实现不基于多线程的网络计算机扫描程序步骤：

1. 启动 Visual Studio .Net，并新建一个 Visual C＃项目，项目名称为【扫描网络计算机】。

2. 把 Visual Studio .Net 的当前窗口切换到【Form1.cs（设计）】窗口，并从【工具箱】中的【Windows 窗体组件】选项卡中往 Form1 窗体中拖入下列组件，并执行相应操作：

四个 NumericUpDown 组件，用以组合成一个 IP 地址区间。

一个 ListBox 组件，用以显示扫描后的结果。

一个 ProgressBar 组件，用以显示程序的运行进度。

四个 Label 组件，用以显示提示信息。

一个 GroupBox 组件。

一个 Button 组件，名称为 button1，并在这组件拖入窗体后，双击 button1，这样 Visual Studio .Net 就会产生这 button1 组件 Click 事件对应的处理代码。

界面设置如下图：

图 01：【扫描网络计算机】项目的设计界面

3. 把 Visual Studio .Net 的当前窗口切换到【Form1.cs】，进入 Form1.cs 文件的编辑界面。在 Form1.cs 头部，用下列代码替换系统缺省的导入命名空间代码：

```
using System ;
using System.Drawing ;
using System.Collections ;
using System.ComponentModel ;
using System.Windows.Forms ;
using System.Data ;
using System.Net.Sockets ;
using System.Net ;
```

4. 用下列代码替换 Form1.cs 中的 button1 的 Click 时间对应的处理代码，下列代码的功能是扫描给定的 IP 地址区间，并把扫描结果显示出来。

```
private void button1_Click ( object sender , System.EventArgs e )
{
  listBox1.Items.Clear ( ) ;
  //清楚扫描结果显示区域
  DateTime StartTime = DateTime.Now ;
  //获取当前时间
  string mask = numericUpDown1.Value.ToString ( ) + "." + numericUpDown2.Value.ToString ( ) +
"." + numericUpDown3.Value.ToString ( ) + "." ;
  int Min = ( int ) numericUpDown4.Value ;
```

```
  int Max = ( int ) numericUpDown5.Value ;
  if ( Min > Max )
  {
    MessageBox.Show ( "输入的 IP 地址区间不合法，请检查！" , "错误！" ) ;
    return ;
  }
  //判断输入的 IP 地址区间是否合法
  progressBar1.Minimum = Min ;
  progressBar1.Maximum = Max ;
  int i ;
  for ( i = Min ; i <= Max ; i++ )
  {
    string ip= mask + i.ToString ( ) ;
    IPAddress myIP = IPAddress.Parse ( ip ) ;
    //根据给定的 IP 地址字符串，处境 IPAddress 实例
    try
    {
      IPHostEntry myHost = Dns.GetHostByAddress ( myIP ) ;
      string HostName = myHost.HostName.ToString ( ) ;
      listBox1.Items.Add ( ip + "名称为：" + HostName ) ;
    }
    catch
    {
      listBox1.Items.Add ( ip + "主机没有响应!" ) ;
    }
    progressBar1.Value = i ;
  }
  //扫描给定 IP 地址对应的计算机是否在线
  DateTime EndTime = DateTime.Now ;
  TimeSpan ts = EndTime-StartTime ;
  //获得扫描网络计算机所使用的时间
  label4.Text = ts.Seconds.ToString ( ) + "秒" ;
  MessageBox.Show ( "成功完成检测！" , "提示" ) ;
  progressBar1.Value = Min ;
}
```

由于上述代码比较简单，并且在代码中的注释也比较详细，这里就不加以解释了，但请注意上面代码中对时间日期类型数据的处理方法。因为有很多人曾经向我讯问过类似问题。

5. 至此，不基于多线程的【扫描网络计算机】项目的全部工作就完成了，程序的执行是很机械的，其方法是对每一个 IP 按照顺序进行 DNS 解析，并得到解析结果，所以程序的执行时间和扫描的 IP 地址区间段大小成正比。图 02 是此程序运行后，扫描"10.138.198.1"至"10.138.198.10"这个 IP 地址区间计算机后的运行界面。整个程序的运行时间为 43 秒：

图 02：不基于多线程的【扫描网络计算机】项目的运行界面

五.把【扫描网络计算机】程序修改成基于多线程的程序：

在修改成多线程程序之前，必须面对并解决下面几个问题：

1. 线程是无返回值的，所以在线程中处理、调用的应是一个过程，所以要把扫描 IP 地址对应的计算机的代码给包装成一个过程。

2. 放在线程中处理的过程，因为没有返回值，从而无法向主程序（进程）传递数值。但扫描 IP 地址对应的计算机的过程却要向主程序（进程）传递 IP 地址是否在线的数据，所以在修改成多线程程序之前，必须从线程往主程序（进程）传递数据的问题。

下面是在【扫描网络计算机】项目的基础上，把它修改成基于多线程程序的具体实现步骤：

1. 由于程序中使用到线程，所以在 Form1.cs 代码首部，导入命名空间代码区中加入下列代码，下列代码是导入 Thread 类所在的命名空间。

```
using System.Threading ;
```

2. 在 Form1.cs 代码的 namespace 代码区加入下列语句，下列语句是定义一个 delegate：

```
public delegate void UpdateList ( string sIP , string sHostName ) ;
```

3. 在 Form1.cs 中的定义 Form1 的 class 代码区定义加入下列代码，下列代码是定义一个变量，用以存放程序执行的时间：

```
private System.DateTime StartTime ;
```

4. 在 Form1.cs 代码的 Main 函数之后，添加下列代码，下列代码是创建一个名称为 ping 的 Class，这个 Class 能够通过其设定的属性接收给定的 IP 地址字符串，并据此来判断此 IP 地址字符串对应的计算机是否在线，并通过其设定的 HostName 属性接收从线程传递来的数据。

```
public class ping
{
  public UpdateList ul ;
  public string ip ;
```

```
  //定义一个变量，用以接收传送来的 IP 地址字符串
  public string HostName ;
  //定义一个变量，用以向主进展传递对应 IP 地址是否在线数据
  public void scan ( )
  {
    IPAddress myIP = IPAddress.Parse ( ip ) ;
    try
    {
      IPHostEntry myHost = Dns.GetHostByAddress ( myIP );
      HostName = myHost.HostName.ToString ( ) ;
    }
    catch
    {
      HostName = "" ;
    }
    if (HostName == "")
      HostName = " 主机没有响应！";
    if ( ul != null)
      ul ( ip , HostName ) ;
  }
  //定义一个过程（也可以看出为方法），用以判断传送来的 IP 地址对应计算机是否在线
}
```

5. 在 Form1.cs 中添加完上述代码后，再添加下列代码：

```
void UpdateMyList ( string sIP , string sHostName )
{
  lock ( listBox1 )
  {
    listBox1.Items.Add ( sIP + " " + sHostName ) ;
    if ( progressBar1.Value != progressBar1.Maximum )
    {
      progressBar1.Value++ ;
    }
    if ( progressBar1.Value == progressBar1.Maximum )
    {
      MessageBox.Show ( "成功完成检测！" , "提示" ) ;
      DateTime EndTime = DateTime.Now ;
      TimeSpan ts = EndTime-StartTime ;
```

```
      label4.Text = ts.Seconds.ToString ( ) + "秒" ;
      //显示扫描计算机所需要的时间
      progressBar1.Value = progressBar1.Minimum ;
    }
  }
}
```

6. 用下列代码替换Form1.cs中button1的Click事件对应的处理代码,下列代码功能是创建多个扫描给定IP地址区间对应的计算机线程实例,并显示扫描结果。

```
private void button1_Click(object sender, System.EventArgs e)
{
  listBox1.Items.Clear ( ) ;
  //清楚扫描结果显示区域
  StartTime = DateTime.Now ;
  //获取当前时间
  string mask = numericUpDown1.Value.ToString ( ) + "." + numericUpDown2.Value.ToString ( ) +
"." + numericUpDown3.Value.ToString ( ) + "." ;
  int Min = ( int ) numericUpDown4.Value ;
  int Max = ( int ) numericUpDown5.Value ;
  if ( Min > Max )
  {
    MessageBox.Show ( "输入的IP地址区间不合法，请检查！" , "错误！" ) ;
    return ;
  }
  //判断输入的IP地址区间是否合法
  int _ThreadNum = Max - Min + 1 ;
  Thread[] mythread = new Thread [ _ThreadNum ] ;
  //创建一个多个Thread实例
  progressBar1.Minimum = Min ;
  progressBar1.Maximum = Max + 1 ;
  progressBar1.Value = Min ;
  int i ;
  for (i = Min ; i <= Max ; i++ )
  {
    int k = Max - i ;
    ping HostPing = new ping ( ) ;
    //创建一个ping实例
    HostPing.ip = mask + i.ToString ( ) ;
    HostPing.ul = new UpdateList ( UpdateMyList ) ;
```

```
    //向这个 ping 实例中传递 IP 地址字符串
    mythread[k] = new Thread ( new ThreadStart ( HostPing.scan ) ) ;
    //初始化一个线程实例
    mythread[k].Start ( ) ;
    //启动线程
  }
}
```

至此，【扫描网络计算机】项目已经被修改成一个多线程的程序了，此时在运行程序，并且同样再扫描上面给定 IP 地址区间对应的计算机，就会惊奇的发现程序执行时间所建为 10 秒了，并且不论要扫描的计算机数目有多少，程序的运行时间也是 10 秒左右，这是因为程序为扫描每一个 IP 都分配一个线程，这样程序的执行时间就不与要扫描的 IP 地址段中的 IP 地址数目有关联了，这样也就大大减少了程序的运行时间，提高了程序的运行效率，这也充分体现出多线程给网络编程带来的好处。图 03 也是程序扫描"10.138.198.1"至"10.138.198.10"这个 IP 地址区间计算机后的运行界面所示：

图 03：基于多线程的【扫描网络计算机】项目的运行界面

通过对二个程序的比较可见，在编写网络应用程序中，正确的使用线程的确能够大大提高程序的运行效率。

六.总结：

至此，本节要介绍的内容就全部结束了，不知道诸位通过上面的介绍是否了解、掌握了下面几点：

1. 如何获取系统当前时间，和实现时间日期类型数据的加减。

2. 在编写网络应用程序时候，使用线程（多线程）的原因，以及线程（多线程）会给网络应用程序带来什么好处。

3. 如何在应用程序中创建多个线程实例。

4. 如何实现带"返回值"的线程。

如果上述要点你都能够掌握，那是再好不过的了。但如果你对线程及其使用方法还感觉模糊，那也不要紧，毕竟线程在编程技术中是一个内容丰富，使用复杂的东东，要立马掌握的确是很困难的事情。在以后的文章中也将再介绍这方面的内容。

基于.NET 的多线程编程入门

作者： 维维 出处： 天极网 责任编辑： 方舟 [2005-09-14 08:30]

简介

多线程在构建大型系统的时候是需要重点关注的一个重要方面，特别是在效率（系统跑得多快？）和性能（系统工作正常？）之间做一个权衡的时候。恰当的使用多线程可以极大的提高系统性能。

什么是线程？

每个正在系统上运行的程序都是一个进程。每个进程包含一到多个线程。进程也可能是整个程序或者是部分程序的动态执行。线程是一组指令的集合，或者是程序的特殊段，它可以在程序里独立执行。也可以把它理解为代码运行的上下文。所以线程基本上是轻量级的进程，它负责在单个程序里执行多任务。通常由操作系统负责多个线程的调度和执行。

什么是多线程？

多线程是为了使得多个线程并行的工作以完成多项任务，以提高系统的效率。线程是在同一时间需要完成多项任务的时候被实现的。

使用线程的好处有以下几点：

- 使用线程可以把占据长时间的程序中的任务放到后台去处理

- 用户界面可以更加吸引人，这样比如用户点击了一个按钮去触发某些事件的处理，可以弹出一个进度条来显示处理的进度

- 程序的运行速度可能加快

• 在一些等待的任务实现上如用户输入、文件读写和网络收发数据等，线程就比较游泳了。在这种情况下我们可以释放一些珍贵的资源如内存占用等等。

还有其他很多使用多线程的好处，这里就不一一说明了。

一些线程模型的背景

我们可以重点讨论一下在Win32 环境中常用的一些模型。

• 单线程模型

在这种线程模型中，一个进程中只能有一个线程，剩下的进程必须等待当前的线程执行完。这种模型的缺点在于系统完成一个很小的任务都必须占用很长的时间。

• 块线程模型（单线程多块模型STA）

这种模型里，一个程序里可能会包含多个执行的线程。在这里，每个线程被分为进程里一个单独的块。每个进程可以含有多个块，可以共享多个块中的数据。程序规定了每个块中线程的执行时间。所有的请求通过Windows消息队列进行串行化，这样保证了每个时刻只能访问一个块，因而只有一个单独的进程可以在某一个时刻得到执行。这种模型比单线程模型的好处在于，可以响应同一时刻的多个用户请求的任务而不只是单个用户请求。但它的性能还不是很好，因为它使用了串行化的线程模型，任务是一个接一个得到执行的。

• 多线程块模型（自由线程块模型）

多线程块模型（MTA）在每个进程里只有一个块而不是多个块。这单个块控制着多个线程而不是单个线程。这里不需要消息队列，因为所有的线程都是相同的块的一个部分，并且可以共享。这样的程序比单线程模型和STA的执行速度都要块，因为降低了系统的负载，因而可以优化来减少系统idle的时间。这些应用程序一般比较复杂，因为程序员必须提供线

程同步以保证线程不会并发的请求相同的资源，因而导致竞争情况的发生。这里有必要提供一个锁机制。但是这样也许会导致系统死锁的发生。

多线程在.NET里如何工作?

在本质上和结构来说，.NET是一个多线程的环境。有两种主要的多线程方法是.NET所提倡的：使用ThreadStart来开始你自己的进程，直接的（使用ThreadPool.QueueUserWorkItem）或者间接的（比如Stream.BeginRead，或者调用BeginInvoke）使用ThreadPool类。一般来说，你可以"手动"为长时间运行的任务创建一个新的线程，另外对于短时间运行的任务尤其是经常需要开始的那些，进程池是一个非常好的选择。进程池可以同时运行多个任务，还可以使用框架类。对于资源紧缺需要进行同步的情况来说，它可以限制某一时刻只允许一个线程访问资源。这种情况可以视为给线程实现了锁机制。线程的基类是System.Threading。所有线程通过CLI来进行管理。

• 创建线程：

创建一个新的Thread对象的实例。Thread的构造函数接受一个参数：

```
Thread DummyThread = new Thread( new ThreadStart(dummyFunction) );
```

• 执行线程：

使用 Threading 命名空间里的 start 方法来运行线程：

```
DummyThread.Start ();
```

• 组合线程：

经常会出现需要组合多个线程的情况，就是当某个线程需要其他线程的结束来完成自己的任务。假设 DummyThread 必须等待 DummyPriorityThread 来完成自己的任务，我们只需要这样做：

```
DummyPriorityThread.Join() ;
```

• 暂停线程：

使得线程暂停给定的秒

```
DummyPriorityThread.Sleep(<Time in Second>);
```

• 中止线程：

如果需要中止线程可以使用如下的代码：

```
DummyPriorityThread.Abort();
```

• 同步

经常我们会遇到需要在线程间进行同步的情况，下面的代码给出了一些方法：

```
using System;using System.Threading;namespace SynchronizationThreadsExample{
  class SynchronizationThreadsExample{
    private int counter = 0;
    static void Main( ) {
      SynchronizationThreadsExample STE = new SynchronizationThreadsExample();
      STE.ThreadFunction( );
    }
    public void ThreadFunction ( ) {
      Thread DummyThread = new Thread( new ThreadStart(SomeFunction) ;
      DummyThread.IsBackground=true;
      DummyThread.Name = "First Thread";
      DummyThread.Start( );
      Console.WriteLine("Started thread {0}", DummyThread.Name);
      Thread DummyPriorityThread = new Thread( new ThreadStart(SomeFunction) );
      DummyPriorityThread.IsBackground=true;
      DummyPriorityThread.Name = "Second Thread";
      DummyPriorityThread.Start( );
      Console.WriteLine("Started thread {0}", DummyPriorityThread.Name);
      DummyThread.Join( );
```

```
        DummyPriorityThread.Join( );
      }
      public void SomeFunction( ) {
        try {
          while (counter < 10) {
            int tempCounter = counter;
            tempCounter ++;
            Thread.Sleep(1);
            counter = tempCounter;
            Console.WriteLine( "Thread {0}. SomeFunction: {1}",Thread.CurrentThread.Name,
counter);
          }
        }
        catch (ThreadInterruptedException Ex) {
          Console.WriteLine( "Exception in thread {0} ", Thread.CurrentThread.Name);
        }
        finally {
          Console.WriteLine( "Thread {0} Exiting. ",Thread.CurrentThread.Name);
        }
      }
    }
}
```

- 使用 Interlock

C#提供了一个特殊的类叫做 interlocked，就是提供了锁机制的实现，我们可以加入如下的代码实现锁机制：

```
Interlocked.SomeFunction (ref counter);
```

- 使用锁

这是为了锁定代码关键区域以进行同步，锁定代码如下：

```
lock (this){ Some statements ;}
```

- 使用 Monitor

当有需要进行线程管理的时候我们可以使用：

```
Monitor.Enter(this);
```

其他也有一些方法进行管理，这里就不一一提及了。

线程的缺点

线程自然也有缺点，以下列出了一些：

- 如果有大量的线程，会影响性能，因为操作系统需要在他们之间切换；
- 更多的线程需要更多的内存空间
- 线程会给程序带来更多的 bug，因此要小心使用
- 线程的中止需要考虑其对程序运行的影响
- 通常块模型数据是在多个线程间共享的，需要一个合适的锁系统替换掉数据共享

浅析.Net 下的多线程编程

admin 发表于：04-04-24

多线程是许多操作系统所具有的特性，它能大大提高程序的运行效率，所以多线程编程技术为编程者广泛关注。目前微软的.Net 战略正进一步推进，各种相关的技术正为广大编程者所接受，同样在.Net 中多线程编程技术具有相当重要的地位。本文我就向大家介绍在.Net 下进行多线程编程的基本方法和步骤。

开始新线程

在.Net 下创建一个新线程是非常容易的，你可以通过以下的语句来开始一个新的线程：

```
Thread thread = new Thread (new ThreadStart (ThreadFunc));

thread.Start ();
```

第一条语句创建一个新的 Thread 对象，并指明了一个该线程的方法。当新的线程开始时，该方法也就被调用执行了。该线程对象通过一个 System..Threading.ThreadStart 类的一个实例以类型安全的方法来调用它要调用的线程方法。

第二条语句正式开始该新线程，一旦方法 Start（）被调用，该线程就保持在一个"alive"的状态下了，你可以通过读取它的 IsAlive 属性来判断它是否处于"alive"状态。下面的语句显示了如果一个线程处于"alive"状态下就将该线程挂起的方法：

```
if (thread.IsAlive) {

thread.Suspend ();

}
```

不过请注意，线程对象的 Start（）方法只是启动了该线程，而并不保证其线程方法 ThreadFunc

（）能立即得到执行。它只是保证该线程对象能被分配到 CPU 时间，而实际的执行还要由操作系统根据处理器时间来决定。

一个线程的方法不包含任何参数，同时也不返回任何值。它的命名规则和一般函数的命名规则相同。它既可以是静态的（static）也可以是非静态的（nonstatic）。当它执行完毕后，相应的线程也就结束了，其线程对象的 IsAlive 属性也就被置为 false 了。下面是一个线程方法的实例：

```
public static void ThreadFunc()
  {
  for (int i = 0; i <10; i++) {
  Console.WriteLine("ThreadFunc {0}", i);
  }
  }
```

前台线程和后台线程

.Net 的公用语言运行时（Common Language Runtime，CLR）能区分两种不同类型的线程：前台线程和后台线程。这两者的区别就是：应用程序必须运行完所有的前台线程才可以退出；而对于后台线程，应用程序则可以不考虑其是否已经运行完毕而直接退出，所有的后台线程在应用程序退出时都会自动结束。

一个线程是前台线程还是后台线程可由它的 IsBackground 属性来决定。这个属性是可读又可写的。它的默认值为 false，即意味着一个线程默认为前台线程。我们可以将它的 IsBackground 属性设置为 true，从而使之成为一个后台线程。

下面的例子是一个控制台程序，程序一开始便启动了 10 个线程，每个线程运行 5 秒钟时间。由于线程的 IsBackground 属性默认为 false，即它们都是前台线程，所以尽管程序的主线程

很快就运行结束了，但程序要到所有已启动的线程都运行完毕才会结束。示例代码如下：

```
using System;

using System.Threading;

class MyApp

{

public static void Main ()

{

for (int i=0; i<10; i++) {

   Thread thread = new Thread (new ThreadStart (ThreadFunc));

   thread.Start ();

}

}

private static void ThreadFunc ()

{

DateTime start = DateTime.Now;

while ((DateTime.Now - start).Seconds <5)

;

}

}
```

接下来我们对上面的代码进行略微修改，将每个线程的 IsBackground 属性都设置为 true，则每个线程都是后台线程了。那么只要程序的主线程结束了，整个程序也就结束了。示例代码如下：

```
using System;
using System.Threading;
class MyApp
{
public static void Main ()
{
for (int i=0; i<10; i++) {
Thread thread = new Thread (new ThreadStart (ThreadFunc));
thread.IsBackground = true;
thread.Start ();
}
}
private static void ThreadFunc ()
{
DateTime start = DateTime.Now;
while ((DateTime.Now - start).Seconds <5)
;
}
}
```

既然前台线程和后台线程有这种差别，那么我们怎么知道该如何设置一个线程的IsBackground 属性呢？下面是一些基本的原则：对于一些在后台运行的线程，当程序结束时这些线程没有必要继续运行了，那么这些线程就应该设置为后台线程。比如一个程序启动了一个进行大量运算的线程，可是只要程序一旦结束，那个线程就失去了继续存在的意义，那么那个线程就该是作为后台线程的。而对于一些服务于用户界面的线程往往是要设置为前台线程的，因为即使程序的主线程结束了，其他的用户界面的线程很可能要继续存在来显示相关的信息，所以不能立即终止它们。这里我只是给出了一些原则，具体到实际的运用往往需要编程者的进一步仔细斟酌。

线程优先级

一旦一个线程开始运行，线程调度程序就可以控制其所获得的 CPU 时间。如果一个托管的应用程序运行在 Windows 机器上，则线程调度程序是由 Windows 所提供的。在其他的平台上，线程调度程序可能是操作系统的一部分，也自然可能是.Net 框架的一部分。不过我们这里不必考虑线程的调度程序是如何产生的，我们只要知道通过设置线程的优先级我们就可以使该线程获得不同的 CPU 时间。

线程的优先级是由 Thread.Priority 属性控制的，其值包含：ThreadPriority.Highest、ThreadPriority.AboveNormal 、 ThreadPriority.Normal 、 ThreadPriority.BelowNormal 和 ThreadPriority.Lowest。从它们的名称上我们自然可以知道它们的优先程度，所以这里就不多作介绍了。

线程的默认优先级为 ThreadPriority.Normal。理论上，具有相同优先级的线程会获得相同的 CPU 时间，不过在实际执行时，消息队列中的线程阻塞或是操作系统的优先级的提高等原因会导致具有相同优先级的线程会获得不同的 CPU 时间。不过从总体上来考虑仍可以忽略这种差异。你可以通过以下的方法来改变一个线程的优先级。

```
thread.Priority = ThreadPriority.AboveNormal;
```

或是：

thread.Priority = ThreadPriority.BelowNormal;

通过上面的第一句语句你可以提高一个线程的优先级，那么该线程就会相应的获得更多的 CPU 时间；通过第二句语句你便降低了那个线程的优先级，于是它就会被分配到比原来少的 CPU 时间了。你可以在一个线程开始运行前或是在它的运行过程中的任何时候改变它的优先级。理论上你还可以任意的设置每个线程的优先级，不过一个优先级过高的线程往往会影响到其他线程的运行，甚至影响到其他程序的运行，所以最好不要随意的设置线程的优先级。

挂起线程和重新开始线程

Thread 类分别提供了两个方法来挂起线程和重新开始线程，也就是 Thread.Suspend 能暂停一个正在运行的线程，而 Thread.Resume 又能让那个线程继续运行。不像 Windows 内核，.Net 框架是不记录线程的挂起次数的，所以不管你挂起线程过几次，只要一次调用 Thread.Resume 就可以让挂起的线程重新开始运行。

Thread 类还提供了一个静态的 Thread.Sleep 方法，它能使一个线程自动的挂起一定的时间，然后自动的重新开始。一个线程能在它自身内部调用 Thread.Sleep 方法，也能在自身内部调用 Thread.Suspend 方法，可是一定要别的线程来调用它的 Thread.Resume 方法才可以重新开始。这一点是不是很容易想通的啊？下面的例子显示了如何运用 Thread.Sleep 方法：

```
while (ContinueDrawing) {

DrawNextSlide ();

Thread.Sleep (5000);

}
```

终止线程

在托管的代码中，你可以通过以下的语句在一个线程中将另一个线程终止掉：

```
thread.Abort ();
```

下面我们来解释一下 Abort（）方法是如何工作的。因为公用语言运行时管理了所有的托管

的线程，同样它能在每个线程内抛出异常。Abort（）方法能在目标线程中抛出一个ThreadAbortException 异常从而导致目标线程的终止。不过 Abort（）方法被调用后，目标线程可能并不是马上就终止了。因为只要目标线程正在调用非托管的代码而且还没有返回的话，该线程就不会立即终止。而如果目标线程在调用非托管的代码而且陷入了一个死循环的话，该目标线程就根本不会终止。不过这种情况只是一些特例，更多的情况是目标线程在调用托管的代码，一旦 Abort（）被调用那么该线程就立即终止了。

在实际应用中，一个线程终止了另一个线程，不过往往要等那个线程完全终止了它才可以继续运行，这样的话我们就应该用到它的 Join（）方法。示例代码如下：

```
thread.Abort ();   // 要求终止另一个线程

thread.Join (); // 只到另一个线程完全终止了，它才继续运行
```

但是如果另一个线程一直不能终止的话（原因如前所述），我们就需要给 Join（）方法设置一个时间限制，方法如下：

```
thread.Join (5000); // 暂停 5 秒
```

这样，在 5 秒后，不管那个线程有没有完全终止，本线程就强行运行了。该方法还返回一个布尔型的值，如果是 true 则表明那个线程已经完全终止了，而如果是 false 的话，则表明已经超过了时间限制了。

时钟线程

.Net 框架中的 Timer 类可以让你使用时钟线程，它是包含在 System.Threading 名字空间中的，它的作用就是在一定的时间间隔后调用一个线程的方法。下面我给大家展示一个具体的实例，该实例以 1 秒为时间间隔，在控制台中输出不同的字符串，代码如下：

```
using System;

using System.Threading;

class MyApp

{

private static bool TickNext = true;

public static void Main ()

{

Console.WriteLine ("Press Enter to terminate...");
```

```
TimerCallback callback = new TimerCallback (TickTock);

Timer timer = new Timer (callback, null, 1000, 1000);

Console.ReadLine ();

}

private static void TickTock (object state)

{

Console.WriteLine (TickNext ? "Tick" : "Tock");

TickNext = ! TickNext;

}

}
```

从上面的代码中，我们知道第一个函数回调是在 1000 毫秒后才发生的，以后的函数回调也是在每隔 1000 毫秒之后发生的，这是由 Timer 对象的构造函数中的第三个参数所决定的。程序会在 1000 毫秒的时间间隔后不断的产生新线程，只到用户输入回车才结束运行。不过值得注意的是，虽然我们设置了时间间隔为 1000 毫秒，但是实际运行的时候往往并不能非常精确。因为 Windows 操作系统并不是一个实时系统，而公用语言运行时也不是实时的，所以由于线程调度的千变万化，实际的运行效果往往是不能精确到毫秒级的，但是对于一般的应用来说那已经是足够的了，所以你也不必十分苛求。

小结

本文介绍了在.Net 下进行多线程编程所需要掌握的一些基本知识。从文章中我们可以知道在.Net 下进行多线程编程相对以前是有了大大的简化，但是其功能并没有被削弱。使用以上的一些基本知识，读者就可以试着编写.Net 下的多线程程序了。不过要编写出功能更加强大而且 Bug 少的多线程应用程序，读者需要掌握诸如线程同步、线程池等高级的多线程编程技术。读者不妨参考一些操作系统方面或是多线程编程方面的技术丛书。

通过多线程为基于 .NET 的应用程序实现响应迅速的用户

通过多线程为基于 .NET 的应用程序实现响应迅速的用户

Ian Griffiths

摘要

如果应用程序在控制用户界面的线程上执行非 UI 处理，则会使应用程序的运行显得缓慢而迟钝，让用户难以忍受。但是长期以来，编写适用于 Windows 的多线程应用程序只限于 C++ 开发人员。现在有了 .NET Framework，您就可以充分利用 C# 中的多线程来控制程序中的指令流，并使 UI 线程独立出来以便用户界面能够迅速响应。本文将向您介绍如何实现这一目标。此外，本文还将讨论多线程的缺陷并提供一个框架来保护并发线程执行的安全。

用户不喜欢反应慢的程序。程序反应越慢，就越没有用户会喜欢它。在执行耗时较长的操作时，使用多线程是明智之举，它可以提高程序 UI 的响应速度，使得一切运行显得更为快速。在 Windows 中进行多线程编程曾经是 C++ 开发人员的专属特权，但是现在，可以使用所有兼容 Microsoft .NET 的语言来编写，其中包括 Visual Basic.NET。不过，Windows 窗体对线程的使用强加了一些重要限制。本文将对这些限制进行阐释，并说明如何利用它们来提供快速、高质量的 UI 体验，即使是程序要执行的任务本身速度就较慢。

为什么选择多线程？

多线程程序要比单线程程序更难于编写，并且不加选择地使用线程也是导致难以找到细小错误的重要原因。这就自然会引出两个问题：为什么不坚持编写单线程代码？如果必须使用多线程，如何才能避免缺陷呢？本文的大部分篇幅都是在回答第二个问题，但首先我要来解释一下为什么确实需要多线程。

多线程处理可以使您能够通过确保程序"永不睡眠"从而保持 UI 的快速响应。大部分程序都有不响应用户的时候：它们正忙于为您执行某些操作以便响应进一步的请求。也许最广为人知的例子就是出现在"打开文件"对话框顶部的组合框。如果在展开该组合框时，CD-ROM 驱动器里恰好有一张光盘，则计算机通常会在显示列表之前先读取光盘。这可能需要几秒钟的时间，在此期间，程序既不响应任何输入，也不允许取消该操作，尤其是在确实并不打算使用光驱的时候，这种情况会让人无法忍受。

执行这种操作期间 UI 冻结的原因在于，UI 是个单线程程序，单线程不可能在等待 CD-ROM 驱动器读取操作的同时处理用户输入，如图 1 所示。"打开文件"对话框会调用某些阻塞 (blocking) API 来确定 CD-ROM 的标题。阻塞 API 在未完成自己的工作之前不会返回，因此这期间它会阻止线程做其他事情。

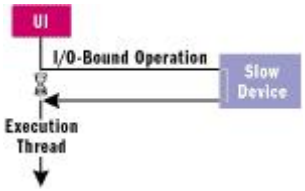


图 1 单线程

在多线程下，像这样耗时较长的任务就可以在其自己的线程中运行，这些线程通常称为辅助线程。因为只有辅助线程受到阻止，所以阻塞操作不再导致用户界面冻结，如图 2 所示。应用程序的主线程可以继续处理用户的鼠标和键盘输入的同时，受阻的另一个线程将等待 CD-ROM 读取，或执行辅助线程可能做的任何操作。

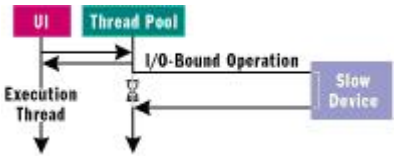


图 2 多线程

其基本原则是，负责响应用户输入和保持用户界面为最新的线程（通常称为 UI 线程）不应该用于执行任何耗时较长的操作。惯常做法是，任何耗时超过 30ms 的操作都要考虑从 UI 线程中移除。这似乎有些夸张，因为 30ms 对于大多数人而言只不过是他们可以感觉到的最短的瞬间停顿，实际上该停顿略短于电影屏幕中显示的连续帧之间的间隔。

如果鼠标单击和相应的 UI 提示（例如，重新绘制按钮）之间的延迟超过 30ms，那么操作与显示之间就会稍显不连贯，并因此产生如同影片断帧那样令人心烦的感觉。为了达到完全高质量的响应效果，上限必须是 30ms。另一方面，如果您确实不介意感觉稍显不连贯，但也不想因为停顿过长而激怒用户，则可按照通常用户所能容忍的限度将该间隔设为 100ms。

这意味着如果想让用户界面保持响应迅速，则任何阻塞操作都应该在辅助线程中执行 — 不管是机械等待某事发生（例如，等待 CD-ROM 启动或者硬盘定位数据），还是等待来自网络的响应。

异步委托调用

在辅助线程中运行代码的最简单方式是使用异步委托调用（所有委托都提供该功能）。委托通常是以同步方式进行调用，即，在调用委托时，只有包装方法返回后该调用才会返回。要以异步方式调用委托，请调用 BeginInvoke 方法，这样会对该方法排队以在系统线程池的线程中运行。调用线程会立即返回，而不用等待该方法完成。这比较适合于 UI 程序，因为可以用它来启动耗时较长的作业，而不会使用户界面反应变慢。

例如，在以下代码中，System.Windows.Forms.MethodInvoker 类型是一个系统定义的委托，用于调用不带参数的方法。

```
private void StartSomeWorkFromUIThread () {

    // The work we want to do is too slow for the UI

    // thread, so let's farm it out to a worker thread.

    MethodInvoker mi = new MethodInvoker(RunsOnWorkerThread);
```

```
    mi.BeginInvoke(null, null); // This will not block.
}

// The slow work is done here, on a thread
// from the system thread pool.
private void RunsOnWorkerThread() {
    DoSomethingSlow();
}
```

如果想要传递参数，可以选择合适的系统定义的委托类型，或者自己来定义委托。MethodInvoker 委托并没有什么神奇之处。和其他委托一样，调用 BeginInvoke 会使该方法在系统线程池的线程中运行，而不会阻塞 UI 线程以便其可执行其他操作。对于以上情况，该方法不返回数据，所以启动它后就不用再去管它。如果您需要该方法返回的结果，则 BeginInvoke 的返回值很重要，并且您可能不传递空参数。然而，对于大多数 UI 应用程序而言，这种"启动后就不管"的风格是最有效的，稍后会对原因进行简要讨论。您应该注意到，BeginInvoke 将返回一个 IAsyncResult。这可以和委托的 EndInvoke 方法一起使用，以在该方法调用完毕后检索调用结果。

还有其他一些可用于在另外的线程上运行方法的技术，例如，直接使用线程池 API 或者创建自己的线程。然而，对于大多数用户界面应用程序而言，有异步委托调用就足够了。采用这种技术不仅编码容易，而且还可以避免创建并非必需的线程，因为可以利用线程池中的共享线程来提高应用程序的整体性能。

线程和控件

Windows 窗体体系结构对线程使用制定了严格的规则。如果只是编写单线程应用程序，则没必要知道这些规则，这是因为单线程的代码不可能违反这些规则。然而，一旦采用多线程，就需要理解 Windows 窗体中最重要的一条线程规则：除了极少数的例外情况，否则都不要在它的创建线程以外的线程中使用控件的任何成员。

本规则的例外情况有文档说明，但这样的情况非常少。这适用于其类派生自 System.Windows.Forms.Control 的任何对象，其中几乎包括 UI 中的所有元素。所有的 UI 元素（包括表单本身）都是从 Control 类派生的对象。此外，这条规则的结果是一个被包含的控件（如，包含在一个表单中的按钮）必须与包含它控件位处于同一个线程中。也就是说，一个窗口中的所有控件属于同一个 UI 线程。实际中，大部分 Windows 窗体应用程序最终都只有一个线程，所有 UI 活动都发生在这个线程上。这个线程通常称为 UI 线程。这意味着您不能调用用户界面中任意控件上的任何方法，除非在该方法的文档说明中指出可以调用。该规则的例外情况（总有文档记录）非常少而且它们之间关系也不大。请注意，以下代码是非法的：

```
// Created on UI thread

private Label lblStatus;

...

// Doesn't run on UI thread

private void RunsOnWorkerThread() {

    DoSomethingSlow();

    lblStatus.Text = "Finished!";     // BAD!!

}
```

图 3 多线程

如果您在 .NET Framework 1.0 版本中尝试运行这段代码，也许会侥幸运行成功，或者初看起来是如此。这就是多线程错误中的主要问题，即它们并不会立即显现出来。甚至当出现了一些错误时，在第

一次演示程序之前一切看起来也都很正常。但不要搞错 — 我刚才显示的这段代码明显违反了规则，并且可以预见，任何抱希望于"试运行时良好，应该就没有问题"的人在即将到来的调试期是会付出沉重代价的。

要注意，在明确创建线程之前会发生这样的问题。使用委托的异步调用实用程序（调用它的 BeginInvoke 方法）的任何代码都可能出现同样的问题。委托提供了一个非常吸引人的解决方案来处理 UI 应用程序中缓慢、阻塞的操作，因为这些委托能使您轻松地让此种操作运行在 UI 线程外而无需自己创建新线程。但是由于以异步委托调用方式运行的代码在一个来自线程池的线程中运行，所以它不能访问任何 UI 元素。上述限制也适用于线程池中的线程和手动创建的辅助线程。

在正确的线程中调用控件

有关控件的限制看起来似乎对多线程编程非常不利。如果在辅助线程中运行的某个缓慢操作不对 UI 产生任何影响，用户如何知道它的进行情况呢？至少，用户如何知道工作何时完成或者是否出现错误？幸运的是，虽然此限制的存在会造成不便，但并非不可逾越。有多种方式可以从辅助线程获取消息，并将该消息传递给 UI 线程。理论上讲，可以使用低级的同步原理和池化技术来生成自己的机制，但幸运的是，因为有一个以 Control 类的 Invoke 方法形式存在的解决方案，所以不需要借助于如此低级的工作方式。

Invoke 方法是 Control 类中少数几个有文档记录的线程规则例外之一：它始终可以对来自任何线程的 Control 进行 Invoke 调用。Invoke 方法本身只是简单地携带委托以及可选的参数列表，并在 UI 线程中为您调用委托，而不考虑 Invoke 调用是由哪个线程发出的。实际上，为控件获取任何方法以在正确的线程上运行非常简单。但应该注意，只有在 UI 线程当前未受到阻塞时，这种机制才有效 — 调用只有在 UI 线程准备处理用户输入时才能通过。从不阻塞 UI 线程还有另一个好理由。Invoke 方法会进行测试以了解调用线程是否就是 UI 线程。如果是，它就直接调用委托。否则，它将安排线程切换，并在 UI 线程上调用委托。无论是哪种情况，委托所包装的方法都会在 UI 线程中运行，并且只有当该方法完成时，Invoke 才会返回。

Control 类也支持异步版本的 Invoke，它会立即返回并安排该方法以便在将来某一时间在 UI 线程上运行。这称为 BeginInvoke，它与异步委托调用很相似，与委托的明显区别在于，该调用以异步方式在线程池的某个线程上运行，然而在此处，它以异步方式在 UI 线程上运行。实际上，Control 的 Invoke、BeginInvoke 和 EndInvoke 方法，以及 InvokeRequired 属性都是 ISynchronizeInvoke 接口的成员。该接口可由任何需要控制其事件传递方式的类实现。

由于 BeginInvoke 不容易造成死锁，所以尽可能多用该方法；而少用 Invoke 方法。因为 Invoke 是同步的，所以它会阻塞辅助线程，直到 UI 线程可用。但是如果 UI 线程正在等待辅助线程执行某操作，情况会怎样呢？应用程序会死锁。BeginInvoke 从不等待 UI 线程，因而可以避免这种情况。

现在，我要回顾一下前面所展示的代码片段的合法版本。首先，必须将一个委托传递给 Control 的 BeginInvoke 方法，以便可以在 UI 线程中运行对线程敏感的代码。这意味着应该将该代码放在它自己的方法中，如图 3 所示。一旦辅助线程完成缓慢的工作后，它就会调用 Label 中的 BeginInvoke，以便在其 UI 线程上运行某段代码。通过这样，它可以更新用户界面。

包装 Control.Invoke

虽然**图 3** 中的代码解决了这个问题，但它相当繁琐。如果辅助线程希望在结束时提供更多的反馈信息，而不是简单地给出"Finished!"消息，则 BeginInvoke 过于复杂的使用方法会令人生畏。为了传达其他消息，例如"正在处理"、"一切顺利"等等，需要设法向 UpdateUI 函数传递一个参数。可能还需要添加一个进度栏以提高反馈能力。这么多次调用 BeginInvoke 可能导致辅助线程受该代码支配。这样不仅会造成不便，而且考虑到辅助线程与 UI 的协调性，这样设计也不好。对这些进行分析之后，我们认为包装函数可以解决这两个问题，如**图 4** 所示。

ShowProgress 方法对将调用引向正确线程的工作进行封装。这意味着辅助线程代码不再担心需要过多关注 UI 细节，而只要定期调用 ShowProgress 即可。请注意，我定义了自己的方法，该方法违背了"必须在 UI 线程上进行调用"这一规则，因为它进而只调用不受该规则约束的其他方法。这种技术

会引出一个较为常见的话题：为什么不在控件上编写公共方法呢（这些方法记录为 UI 线程规则的例外）？

刚好 Control 类为这样的方法提供了一个有用的工具。如果我提供一个设计为可从任何线程调用的公共方法，则完全有可能某人会从 UI 线程调用这个方法。在这种情况下，没必要调用 BeginInvoke，因为我已经处于正确的线程中。调用 Invoke 完全是浪费时间和资源，不如直接调用适当的方法。为了避免这种情况，Control 类将公开一个称为 InvokeRequired 的属性。这是"只限 UI 线程"规则的另一个例外。它可从任何线程读取，如果调用线程是 UI 线程，则返回假，其他线程则返回真。这意味着我可以按以下方式修改包装：

```
public void ShowProgress(string msg, int percentDone) {

    if (InvokeRequired) {

        // As before

        ...

    } else {

        // We're already on the UI thread just

        // call straight through.

        UpdateUI(this, new MyProgressEvents(msg,

            PercentDone));

    }

}
```

图 4

ShowProgress 现在可以记录为可从任何线程调用的公共方法。这并没有消除复杂性 — 执行 BeginInvoke 的代码依然存在，它还占有一席之地。不幸的是，没有简单的方法可以完全摆脱它。

锁定

任何并发系统都必须面对这样的事实，即，两个线程可能同时试图使用同一块数据。有时这并不是问题 — 如果多个线程在同一时间试图读取某个对象中的某个字段，则不会有问题。然而，如果有线程想要修改该数据，就会出现问题。如果线程在读取时刚好另一个线程正在写入，则读取线程有可能会看到虚假值。如果两个线程在同一时间、在同一个位置执行写入操作，则在同步写入操作发生之后，所有从该位置读取数据的线程就有可能看到一堆垃圾数据。虽然这种行为只在特定情况下才会发生，读取操作甚至不会与写入操作发生冲突，但是数据可以是两次写入结果的混加，并保持错误结果直到下一次写入值为止。为了避免这种问题，必须采取措施来确保一次只有一个线程可以读取或写入某个对象的状态。

防止这些问题出现所采用的方式是，使用运行时的锁定功能。C# 可以让您利用这些功能、通过锁定关键字来保护代码（Visual Basic 也有类似构造，称为 SyncLock）。规则是，任何想要在多个线程中调用其方法的对象在每次访问其字段时（不管是读取还是写入）都应该使用锁定构造。例如，请参见**图 5**。

```
// This field could be modified and read on any thread, so all access

// must be protected by locking the object.

private double myPosition;

•••

public double Position {

    get {
```

```
        // Position could be modified from any thread, so we need to lock
        // this object to make sure we get a consistent value.
        lock (this) {
            return myPosition;
        }
    }
    set {
        lock (this) {
            myPosition = value;
        }
    }
}

public void MoveBy(double offset) {
    // Here we are reading, checking and then modifying the value. It is
    // vitally important that this entire sequence is protected by a
    // single lock block.
    lock (this) {
        double newPos = Position + offset;
```

```
        // Check within range - MINPOS and MAXPOS

        // will be const values defined somewhere in

        // this class

        if (newPos > MAXPOS) newPos = MAXPOS;

        else if (newPos < MINPOS) newPos = MINPOS;

        Position = newPos;

    }

}
```

锁定构造的工作方式是：公共语言运行库 (CLR) 中的每个对象都有一个与之相关的锁，任何线程均可获得该锁，但每次只能有一个线程拥有它。如果某个线程试图获取另一个线程已经拥有的锁，那么它必须等待，直到拥有该锁的线程将锁释放为止。C# 锁定构造会获取该对象锁（如果需要，要先等待另一个线程利用它完成操作），并保留到大括号中的代码退出为止。如果执行语句运行到块结尾，该锁就会被释放，并从块中部返回，或者抛出在块中没有捕捉到的异常。

请注意，MoveBy 方法中的逻辑受同样的锁语句保护。当所做的修改比简单的读取或写入更复杂时，整个过程必须由单独的锁语句保护。这也适用于对多个字段进行更新 — 在对象处于一致状态之前，一定不能释放该锁。如果该锁在更新状态的过程中释放，则其他线程也许能够获得它并看到不一致状态。如果您已经拥有一个锁，并调用一个试图获取该锁的方法，则不会导致问题出现，因为单独线程允许多次获得同一个锁。对于需要锁定以保护对字段的低级访问和对字段执行的高级操作的代码，这非常重要。MoveBy 使用 Position 属性，它们同时获得该锁。只有最外面的锁阻塞完成后，该锁才会恰当地释放。

对于需要锁定的代码，必须严格进行锁定。稍有疏漏，便会功亏一篑。如果一个方法在没有获取对象锁的情况下修改状态，则其余的代码在使用它之前即使小心地锁定对象也是徒劳。同样，如果一个线程在没有事先获得锁的情况下试图读取状态，则它可能读取到不正确的值。运行时无法进行检查来确保多线程代码正常运行。

死锁

锁是确保多线程代码正常运行的基本条件，即使它们本身也会引入新的风险。在另一个线程上运行代码的最简单方式是，使用异步委托调用（请参见 图 6）。

```
public class Foo {

    public void CallBar() {

        lock (this) {

            Bar myBar = new Bar ();

            myBar.BarWork(this);

        }

    }

    // This will be called back on a worker thread

    public void FooWork() {

        lock (this) {

            // do some work

            •••

        }

    }
```

```
}

public class Bar {

    public void BarWork(Foo myFoo) {

        // Call Foo on different thread via delegate.

        MethodInvoker mi = new MethodInvoker(

            myFoo.FooWork);

        IAsyncResult ar = mi.BeginInvoke(null, null);

        // do some work

        •••

        // Now wait for delegate call to complete (DEADLOCK!)

        mi.EndInvoke(ar);

    }

}
```

如果曾经调用过 Foo 的 CallBar 方法，则这段代码会慢慢停止运行。CallBar 方法将获得 Foo 对象上的锁，并直到 BarWork 返回后才释放它。然后，BarWork 使用异步委托调用，在某个线程池线程中调用 Foo 对象的 FooWork 方法。接下来，它会在调用委托的 EndInvoke 方法前执行一些其他操作。EndInvoke 将等待辅助线程完成，但辅助线程却被阻塞在 FooWork 中。它也试图获取 Foo 对象的锁，但锁已被 CallBar 方法持有。所以，FooWork 会等待 CallBar 释放锁，但 CallBar 也在等待 BarWork 返回。不幸的是，BarWork 将等待 FooWork 完成，所以 FooWork 必须先完成，它才能开始。结果，没有线程能够进行下去。

这就是一个死锁的例子，其中有两个或更多线程都被阻塞以等待对方进行。这里的情形和标准死锁情况还是有些不同，后者通常包括两个锁。这表明如果有某个因果性（过程调用链）超出线程界限，就会发生死锁，即使只包括一个锁！Control.Invoke 是一种跨线程调用过程的方法，这是个不争的重要事实。BeginInvoke 不会遇到这样的问题，因为它并不会使因果性跨线程。实际上，它会在某个线程池线程中启动一个全新的因果性，以允许原有的那个独立进行。然而，如果保留 BeginInvoke 返回的 IAsyncResult，并用它调用 EndInvoke，则又会出现问题，因为 EndInvoke 实际上已将两个因果性合二为一。避免这种情况的最简单方法是，当持有一个对象锁时，不要等待跨线程调用完成。要确保这一点，应该避免在锁语句中调用 Invoke 或 EndInvoke。其结果是，当持有一个对象锁时，将无需等待其他线程完成某操作。要坚持这个规则，说起来容易做起来难。

在检查代码的 BarWork 时，它是否在锁语句的作用域内并不明显，因为在该方法中并没有锁语句。出现这个问题的唯一原因是 BarWork 调用自 Foo.CallBar 方法的锁语句。这意味着只有确保正在调用的函数并不拥有锁时，调用 Control.Invoke 或 EndIn-voke 才是安全的。对于非私有方法而言，确保这一点并不容易，所以最佳规则是，根本不调用 Control.Invoke 和 EndInvoke。这就是为什么"启动后就不管"的编程风格更可取的原因，也是为什么 Control.BeginInvoke 解决方案通常比 Control.Invoke 解决方案好的原因。

有时候除了破坏规则别无选择，这种情况下就需要仔细严格地分析。但只要可能，在持有锁时就应该避免阻塞，因为如果不这样，死锁就难以消除。

使其简单

如何既从多线程获益最大，又不会遇到困扰并发代码的棘手错误呢？如果提高的 UI 响应速度仅仅是使程序时常崩溃，那么很难说是改善了用户体验。大部分在多线程代码中普遍存在的问题都是由要一次运行多个操作的固有复杂性导致的，这是因为大多数人更善于思考连续过程而非并发过程。通常，最好的解决方案是使事情尽可能简单。

UI 代码的性质是：它从外部资源接收事件，如用户输入。它会在事件发生时对其进行处理，但却将大部分时间花在了等待事件的发生。如果可以构造辅助线程和 UI 线程之间的通信，使其适合该模型，则未必会遇到这么多问题，因为不会再有新的东西引入。我是这样使事情简单化的：将辅助线程视为另一个异步事件源。如同 Button 控件传递诸如 Click 和 MouseEnter 这样的事件，可以将辅助线程视为传递事件（如 ProgressUpdate 和 WorkComplete）的某物。只是简单地将这看作一种类比，还是真正将辅助对象封装在一个类中，并按这种方式公开适当的事件，这完全取决于您。后一种选择可能需要更多的代码，但会使用户界面代码看起来更加统一。不管哪种情况，都需要 Control.BeginInvoke 在正确的线程上传递这些事件。

对于辅助线程，最简单的方式是将代码编写为正常顺序的代码块。但如果想要使用刚才介绍的"将辅助线程作为事件源"模型，那又该如何呢？这个模型非常适用，但它对该代码与用户界面的交互提出了限制：这个线程只能向 UI 发送消息，并不能向它提出请求。

例如，让辅助线程中途发起对话以请求完成结果需要的信息将非常困难。如果确实需要这样做，也最好是在辅助线程中发起这样的对话，而不要在主 UI 线程中发起。该约束是有利的，因为它将确保有一个非常简单且适用于两线程间通信的模型 — 在这里简单是成功的关键。这种开发风格的优势在于，在等待另一个线程时，不会出现线程阻塞。这是避免死锁的有效策略。

```
private void btnRead_Click(object o, EventArgs e)

{

    btnRead.Enabled = false;

    txtDir.Enabled = false;

    Cursor = Cursors.AppStarting;

    // Clear the list view

    lvwDirList.Clear();

    // This could be slow, so do this via an async delegate.
```

```
    ReadDelegate dlg = new ReadDelegate(ReadDirectory);

    dlg.BeginInvoke(txtDir.Text, null, null);

}

// Worker function (and associated delegate)

private delegate void ReadDelegate(string dirName);

private void ReadDirectory(string dirName)

{

    try

    {

        string[] names = Directory.GetFileSystemEntries(dirName);

        // Marshal the results back onto the right thread.

        object[] pList = { names };

        BeginInvoke(new DirRead(readDir_DirEntries), pList);

    }

    catch (DirectoryNotFoundException)

    {

        // Raise the directory not found 'event'

        BeginInvoke(new MethodInvoker(readDir_DirNotFound));

    }
```

```
    catch (Exception e)
    {
        // Raise the miscellaneous error 'event'.
        object[] pList = { e };
        BeginInvoke(new DirError(readDir_Error), pList);
    }

}
// Functions called via Invoke by worker thread.
private delegate void DirRead(string[] names);
private void readDir_DirEntries(string[] names)
{
    Cursor = Cursors.WaitCursor;
    foreach(string name in names)
    {
        lvwDirList.Items.Add(name);
    }
    ResetCtls();
}
```

```
private void readDir_DirNotFound()

{

    MessageBox.Show(this, "Directory not found", "Error");

    ResetCtls();

}

private delegate void DirError(Exception e);

private void readDir_Error(Exception e)

{

    MessageBox.Show(this, string.Format("Error reading directory:\n{0}",

        e.Message), "Error");

    ResetCtls();

}

// Set the relevant controls and cursors back to their default

// 'ready' states after the operation has completed.

private void ResetCtls()

{

    // Re-enable the Read button and put the cursor back to normal.

    btnRead.Enabled = true;

    txtDir.Enabled = true;
```

```
    Cursor = Cursors.Default;

}
```

图 7 显示了使用异步委托调用以在辅助线程中执行可能较慢的操作（读取某个目录的内容），然后将结果显示在 UI 上。它还不至于使用高级事件语法，但是该调用确实是以与处理事件（如单击）非常相似的方式来处理完整的辅助代码。

取消

前面示例所带来的问题是，要取消操作只能通过退出整个应用程序实现。虽然在读取某个目录时 UI 仍然保持迅速响应，但由于在当前操作完成之前程序将禁用相关按钮，所以用户无法查看另一个目录。如果试图读取的目录是在一台刚好没有响应的远程机器上，这就很不幸，因为这样的操作需要很长时间才会超时。

要取消一个操作也比较困难，尽管这取决于怎样才算取消。一种可能的理解是"停止等待这个操作完成，并继续另一个操作。"这实际上是抛弃进行中的操作，并忽略最终完成时可能产生的后果。对于当前示例，这是最好的选择，因为当前正在处理的操作（读取目录内容）是通过调用一个阻塞 API 来执行的，取消它没有关系。但即使是如此松散的"假取消"也需要进行大量工作。如果决定启动新的读取操作而不等待原来的操作完成，则无法知道下一个接收到的通知是来自这两个未处理请求中的哪一个。

支持取消在辅助线程中运行的请求的唯一方式是，提供与每个请求相关的某种调用对象。最简单的做法是将它作为一个 Cookie，由辅助线程在每次通知时传递，允许 UI 线程将事件与请求相关联。通过简单的身份比较（参见**图 8**），UI 代码就可以知道事件是来自当前请求，还是来自早已废弃的请求。

```
// Member variable holding the current request—gets updated with the
```

```
// current call object every time we start a new operation. (This call

// object would also be made available to the worker function—either as a

// parameter, or possibly by making the worker function a member of the

// call object.)

private object currReq;

•••

private void workerNotify(object callObj) {

    // Only process this event if it's from the last request we made.

    // (I.e. ignore belated notifications from requests we've long since

    // abandoned).

    if (object.ReferenceEquals(callObj, currReq)) {

        // This is for an operation we still care about, so handle it

        •••

    }

}
```

如果简单抛弃就行，那固然很好，不过您可能想要做得更好。如果辅助线程执行的是进行一连串阻塞操作的复杂操作，那么您可能希望辅助线程在最早的时机停止。否则，它可能会继续几分钟的无用操作。在这种情况下，调用对象需要做的就不止是作为一个被动 Cookie。它至少还需要维护一个标记，指明请求是否被取消。UI 可以随时设置这个标记，而辅助线程在执行时将定期测试这个标记，以确定是否需要放弃当前工作。

对于这个方案，还需要做出几个决定：如果 UI 取消了操作，它是否要等待直到辅助线程注意到这次取消？如果不等待，就需要考虑一个争用条件：有可能 UI 线程会取消该操作，但在设置控制标记之前辅助线程已经决定传递通知了。因为 UI 线程决定不等待，直到辅助线程处理取消，所以 UI 线程有可能会继续从辅助线程接收通知。如果辅助线程使用 BeginInvoke 异步传递通知，则 UI 甚至有可能收到多个通知。UI 线程也可以始终按与"废弃"做法相同的方式处理通知 — 检查调用对象的标识并忽略它不再关心的操作通知。或者，在调用对象中进行锁定并决不从辅助线程调用 BeginInvoke 以解决问题。但由于让 UI 线程在处理一个事件之前简单地对其进行检查以确定是否有用也比较简单，所以使用该方法碰到的问题可能会更少。

程序关闭

某些情况下，可以采用"启动后就不管"的异步操作，而不需要其他复杂要求来使操作可取消。然而，即使用户界面不要求取消，有可能还是需要实现这项功能以使程序可以彻底关闭。

当应用程序退出时，如果由线程池创建的辅助线程还在运行，则这些线程会被终止。终止是简单粗暴的操作，因为关闭甚至会绕开任何还起作用的 Finally 块。如果异步操作执行的某些工作不应该以这种方式被打断，则必须确保在关闭之前这样的操作已经完成。此类操作可能包括对文件执行的写入操作，但由于突然中断后，文件可能被破坏。

一种解决办法是创建自己的线程，而不用来自辅助线程池的线程，这样就自然会避开使用异步委托调用。这样，即使主线程关闭，应用程序也会等到您的线程退出后才终止。System.Threading.Thread 类有一个 IsBackground 属性可以控制这种行为。它默认为 false，这种情况下，CLR 会等到所有非背景线程都退出后才正常终止应用程序。然而，这会带来另一个问题，因为应用程序挂起时间可能会比您预期的长。窗口都关闭了，但进程仍在运行。这也许不是个问题。如果应用程序只是因为要进行一些清理工作才比正常情况挂起更长时间，那没问题。另一方面，如果应用程序在用户界面关闭后还挂起几分钟甚至几小时，那就不可接受了。例如，如果它仍然保持某些文件打开，则可能妨碍用户稍后重启该应用程序。

最佳方法是，如果可能，通常应该编写自己的异步操作以便可以将其迅速取消，并在关闭应用程序之前等待所有未完成的操作完成。这意味着您可以继续使用异步委托，同时又能确保关闭操作彻底且及时。

错误处理

在辅助线程中出现的错误一般可以通过触发 UI 线程中的事件来处理，这样错误处理方式就和完成及进程更新方式完全一样。因为很难在辅助线程上进行错误恢复，所以最简单的策略就是让所有错误都为致命错误。错误恢复的最佳策略是使操作完全失败，并在 UI 线程上执行重试逻辑。如果需要用户干涉来修复造成错误的问题，简单的做法是给出恰当的提示。

AsyncUtils 类处理错误以及取消。如果操作抛出异常，该基类就会捕捉到，并通过 Failed 事件将异常传递给 UI。

小结

谨慎地使用多线程代码可以使 UI 在执行耗时较长的任务时不会停止响应，从而显著提高应用程序的反应速度。异步委托调用是将执行速度缓慢的代码从 UI 线程迁移出来，从而避免此类间歇性无响应的最简单方式。

Windows Forms Control 体系结构基本上是单线程，但它提供了实用程序以将来自辅助线程的调用封送返回至 UI 线程。处理来自辅助线程的通知（不管是成功、失败还是正在进行的指示）的最简单策略是，以对待来自常规控件的事件（如鼠标单击或键盘输入）的方式对待它们。这样可以避免在 UI 代码中引入新的问题，同时通信的单向性也不容易导致出现死锁。

有时需要让 UI 向一个正在处理的操作发送消息。其中最常见的是取消一个操作。通过建立一个表示正在进行的调用的对象并维护由辅助线程定期检查的取消标记可实现这一目的。如果用户界面线程需要等待取消被认可（因为用户需要知道工作已确实终止，或者要求彻底退出程序），实现起来会有些复杂，但所提供的示例代码中包含了一个将所有复杂性封装在内的基类。派生类只需要执行一些必要的工作、周期性测试取消，以及要是因为取消请求而停止工作，就将结果通知基类。

相关文章请参阅：

Applied Microsoft .NET Framework Programming by Jeffrey Richter (Microsoft Press, 2002)

Essential .NET, Volume 1: The Common Language Runtime by Don Box (Addison Wesley Professional, 2002)

有关背景信息，请参阅：

Safe, Simple Multithreading in Windows Forms

A Second Look at Windows Forms Multithreading

Ian Griffiths 是一位专门从事 .NET Windows 窗体应用程序的独立顾问，现居英国。他是 DevelopMentor 的一名讲师，并与他人合著有 *NET Windows Forms in a Nutshell* (O'Reilly, 2002) 和 *Mastering Visual Studio .NET* (O'Reilly, February 2003) 一书。