目录

1、C#的Timer解析 3
1.1、版权声明 3
1.2、内容详情 3
2、Timer范例教程 17
2.1、版权声明 17
2.2、内容详情 17
3、Socket网络编程大全 23
3.1、版权声明 23
3.2、内容详情 23
3.2.1 简单服务器端 23
3.2.2 简单客户端 24
3.2.3 获得本机IP 24
3.2.4 端对端通信 25
3.2.5 点对点通信 25
3.2.6 UDP对时服务器端 27
3.2.7 UDP对时客户端 28
3.2.8 点对点传输文件 30
3.2.9 发送邮件 33
3.2.10 接收邮件 33
4、多线程使用thread、threadpool、timer 44
4.1、版权声明 44
4.2、内容详情 44
5、谈谈多线程的思维方式 52
5.1、版权声明 52
5.2、内容详情 52
6、断点续传、多线程上载 59
6.1、版权声明 59
6.2、内容详情 59
7、C#多线程学习 66
7.1、版权声明 66
7.2、内容详情 66
7.2.1 多线程的相关概念 66
7.2.2 如何操纵一个线程 68
7.2.3 生产者和消费者 71
7.2.4 多线程的自动管理(线程池) 81
7.2.5 多线程的自动管理(定时器Timer) 85
7.2.6 互斥对象Mutex 87
8、C#多线程编程 91
8.1、版权声明 91
8.2、内容详情 91

8.2.1 lock使用注意事项.....91
8.2.2 集合类中Synchronized与SyncRoot属性原理分析.....93
8.2.3 Monitor使用示例及Mutex简介.....99
8.2.4 同步事件和等待句柄.....102
8.2.5 Timer（定时器）使用示例.....105
8.2.6 volatile关键字的原理探讨.....107
8.2.7 Interlocked类操作.....108
8.2.8 使用Semaphore类限制资源并发访问数.....109
8.2.9 用ReaderWriterLock类实现多用户读/单用户写同步.....111
8.2.10 异步方法调用.....113
8.2.11 异步事件调用.....116
8.2.12 BackgroundWorker.....117

1、C#的Timer解析

1.1、版权声明

文章出处：http://blog.csdn.net/BusyDonkey/article/details/5327665
文章作者：BusyDonkey

1.2、内容详情

在 C#里现在有 3 个 Timer 类：

- System.Windows.Forms.Timer
- System.Threading.Timer
- System.Timers.Timer

这三个 Timer 我想大家对 System.Windows.Forms.Timer 已经很熟悉了，唯一我要说的就是这个 Timer 在激发 Timer.Tick 事件的时候，事件的处理函数是在程序主线程上执行的，所以在 WinForm 上面用这个 Timer 很方便，因为在 From 上的所有控件都是在程序主线程上创建的，那么在 Tick 的处理函数中可以对 Form 上的所有控件进行操作，不会造成 WinForm 控件的线程安全问题。

1、Timer 运行的核心都是 System.Threading.ThreadPool

在这里要提到 ThreadPool（线程池）是因为，System.Threading.Timer 和 System.Timers.Timer 运行的核心都是线程池，Timer 每到间隔时间后就会激发响应事件，因此要申请线程来执行对应的响应函数，Timer 将获取线程的工作都交给了线程池来管理，每到一定的时间后它就去告诉线程池："我现在激发了个事件要运行对应的响应函数，麻烦你给我向操作系统要个线程，申请交给你了，线程分配下来了你就运行我给你的响应函数，没分配下来先让响应函数在这儿排队（操作系统线程等待队列）"，消息已经传递给线程池了，Timer 也就不管了，因为它还有其他的事要做（每隔一段时间它又要激发事件），至于提交的请求什么时候能够得到满足，要看线程池当前的状态：

1）如果线程池现在有线程可用，那么申请马上就可以得到满足，有线程可用又可以分为两种情况：

- <1>线程池现在有空闲线程，现在马上就可以用
- <2>线程池本来现在没有线程了，但是刚好申请到达的时候，有线程运行完毕释放了，那么申请就可以用别人释放的线程。
- 这两种情况情况就如同你去游乐园玩赛车，如果游乐园有 10 辆车，现在有 3 个人在玩，那么还剩 7 辆车，你去了当然可以选一辆开。另外还有一种情况就是你到达游乐园前 10 辆车都在开，但是你运气很好，刚到游乐园就有人不玩了，正好你坐

上去就可以接着开。

2）如果现在线程池现在没有线程可用，也分为两种情况：

- <1>线程池现有线程数没有达到设置的最大工作线程数，那么隔半秒钟.net framework 就会向操作系统申请一个新的线程（为避免向线程分配不必要的堆栈空间，线程池按照一定的时间间隔创建新的空闲线程。该时间间隔目前为半秒，但它在 .NET Framework 的以后版本中可能会更改）。
- <2>线程池现有工作线程数达到了设置的最大工作线程数，那么申请只有在等待队列一直等下去，直到有线程执行完任务后被释放。

那么上面提到了线程池有最大工作线程数，其实还有最小空闲线程数，那么这两个关键字是什么意思呢：

- 最大工作线程数：实际上就是指的线程池能够向操作系统申请的最大线程数，这个值在.net framework 中有默认值，这个默认值是根据你计算机的配置来的，当人你可以用 ThreadPool.GetMaxThreads 返回线程池当前最大工作线程数，你也可以同 ThreadPool.SetMaxThreads 设置线程池当前最大工作线程数。
- 最小空闲线程数：是指在程序开始后，线程池就默认向操作系统要最小空闲线程数个线程，另外这也是线程池维护的空闲线程数（如果线程池最小空闲线程数为 3，当前因为一些线程执行完任务被释放，线程池现在实际上有 10 个空闲线程，那么线程池会让操作系统释放多余的 7 个线程，而只维持 3 个空闲线程供程序使用），因为上面说了，在执行程序的时候在要线程池申请线程有半秒的延迟时间，这也会影响程序的性能，所以把握好这个值很重要，用样你可以用 ThreadPool.GetMinThreads 返回线程池当前最小空闲线程数，你也可以同 ThreadPool.SetMinThreads 设置线程池当前最小空闲线程数。

下面是我给的例子，这个例子让线程池申请 800 个线程，其中设置最大工作线程数为 500，800 个线程任务每个都要执行 100000000 毫秒目的是让线程不会释放，并且让用户选择，是否预先申请 500 个空闲线程免受那半秒钟的延迟时间，其结果可想而知当线程申请到 500 的时候，线程池达到了最大工作线程数，剩余的 300 个申请进入漫长的等待时间：

```
/*****************************************************
 * 项目:测试线程池
 * 描述:验证线程池的最大工作线程数和最小空闲线程数
 * 作者:@PowerCoder
 * 日期:2010-2-22
*****************************************************/

using System;
using System.Collections.Generic;
using System.Text;
using System.Threading;

namespace ConsoleApplication1
{
    class Program
```

```
    {
        static int i = 1;
        static int MaxThreadCount = 800;

        static void OutPut(object obj)
        {
            Console.Write("/r申请了:{0}个工作线程", i);
            i++;
            Thread.Sleep(100000000);//设置一个很大的等待时间，让每个申请的线程都一直执行
        }

        static void Main(string[] args)
        {
            int j;

            Console.Write("是否先申请500个空闲线程以保证前500个线程在线程池中开始就有线程用(Y/N)?");
//如果这里选择N，那么前两个任务是用的线程池默认空闲线程（可以用ThreadPool.GetMinThreads得到系统默认最小空闲线程数为2）申请立即得到满足，然而由于每个线程等待时间非常大都不会释放当前自己持有的线程，因此线程池中已无空闲线程所用，后面的任务需要在线程池中申请新的线程，那么新申请的每个线程在线程池中都要隔半秒左右的时间才能得到申请（原因请见下面的注释）
            string key = Console.ReadLine();
            if (key.ToLower() == "y")
                ThreadPool.SetMinThreads(500, 10);
//设置最大空闲线程为500，就好像我告诉系统给我预先准备500个线程我来了就直接用，因为这样就不用现去申请了，在线程池中每申请一个新的线程.NET Framework 会安排一个间隔时间，目前是半秒，以后的版本MS有可能会改

            int a, b;
            ThreadPool.GetMaxThreads(out a, out b);
            Console.WriteLine("线程池默认最大工作线程数：" + a.ToString() + "      默认最大异步 I/O 线程数：" + b.ToString());
            Console.WriteLine("需要向系统申请" + MaxThreadCount.ToString() + "个工作线程");

            for (j = 0; j <= MaxThreadCount - 1; j++)
//由于ThreadPool.GetMaxThreads返回的默认最大工作线程数为500（这个值要根据你计算机的配置来决定），那么向线程池申请大于500个线程的时候，500之后的线程会进入线程池的等待队列，等待前面500个线程某个线程执行完后来唤醒等待队列的某个线程
            {
                ThreadPool.QueueUserWorkItem(new WaitCallback(OutPut));
                Thread.Sleep(10);
            }

            Console.ReadLine();
        }
```

```
    }
}
```

2、System.Threading.Timer

谈完了线程池，就可以开始讨论 Timer，这里我们先从 System.Threading.Timer 开始，System.Threading.Timer 的作用就是每到间隔时间后激发响应事件并执行相应函数，执行响应函数要向线程池申请线程，当然申请中会遇到一些情况在上面我们已经说了。值得注意的一点就是 System.Threading.Timer 在创建对象后立即开始执行，比如 System.Threading.Timer timer = new System.Threading.Timer(Excute, null, 0, 10);这句执行完后每隔 10 毫秒就执行 Excute 函数不需要启动什么的。下面就举个例子，我先把代码贴出来：

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Threading;
using System.Diagnostics;

namespace ConsoleApplication1
{
    class UnSafeTimer
    {
        static int i = 0;
        static System.Threading.Timer timer;
        static object mylock = new object();
        static int sleep;
        static bool flag;
        public static Stopwatch sw = new Stopwatch();

        static void Excute(object obj)
        {
            Thread.CurrentThread.IsBackground = false;
            int c;

            lock (mylock)
            {
                i++;
                c = i;
            }

            if (c == 80)
            {
                timer.Dispose();
//执行 Dispose 后 Timer 就不会再申请新的线程了,但是还是会给 Timmer 已经激发的事件申请线程
```

```
                sw.Stop();
            }

            if (c < 80)
                Console.WriteLine("Now:" + c.ToString());
            else
            {
                Console.WriteLine("Now:" + c.ToString() + "----------Timer 已经 Dispose 耗时:"
+ sw.ElapsedMilliseconds.ToString() + "毫秒");
            }

            if (flag)
            {
                Thread.Sleep(sleep);//模拟花时间的代码
            }
            else
            {
                if (i <= 80)
                    Thread.Sleep(sleep);//前 80 次模拟花时间的代码
            }
        }

        public static void Init(int p_sleep, bool p_flag)
        {
            sleep = p_sleep;
            flag = p_flag;
            timer = new System.Threading.Timer(Excute, null, 0, 10);
        }
    }

    class SafeTimer
    {
        static int i = 0;
        static System.Threading.Timer timer;

        static bool flag = true;
        static object mylock = new object();

        static void Excute(object obj)
        {
            Thread.CurrentThread.IsBackground = false;

            lock (mylock)
            {
```

```
                if (!flag)
                {
                    return;
                }

                i++;

                if (i == 80)
                {
                    timer.Dispose();
                    flag = false;
                }
                Console.WriteLine("Now:" + i.ToString());
            }

            Thread.Sleep(1000);//模拟花时间的代码
        }

        public static void Init()
        {
            timer = new System.Threading.Timer(Excute, null, 0, 10);
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            Console.Write("是否使用安全方法(Y/N)?");
            string key = Console.ReadLine();
            if (key.ToLower() == "y")
                SafeTimer.Init();
            else
            {
                Console.Write("请输入 Timmer 响应事件的等待时间(毫秒):");
//这个时间直接决定了前 80 个任务的执行时间，因为等待时间越短，每个任务就可以越快执行完，那么80 个任务中就有越多的任务可以用到前面任务执行完后释放掉的线程，也就有越多的任务不必去线程池申请新的线程避免多等待半秒钟的申请时间
                string sleep = Console.ReadLine();
                Console.Write("申请了 80 个线程后 Timer 剩余激发的线程请求是否需要等待时间(Y/N)?");
//这里可以发现选 Y 或者 N 只要等待时间不变，最终 Timer 激发线程的次数都相近，说明 Timer 的确在执行 80 次的 Dispose 后就不再激发新的线程了
                key = Console.ReadLine();
                bool flag = false;
```

```
                if (key.ToLower() == "y")
                {
                    flag = true;
                }

                UnSafeTimer.sw.Start();
                UnSafeTimer.Init(Convert.ToInt32(sleep), flag);
            }

            Console.ReadLine();
        }
    }
}
```

这个例子包含了两个 Timer 的类 UnSafeTimer 和 SafeTimer，两个类的代码的大致意思就是使用 Timer 每隔 10 毫秒就执行 Excute 函数，Excute 函数会显示当前执行的次数，在 80 次的时候通过 timer.Dispose()让 Timer 停止不再激发响应事件。

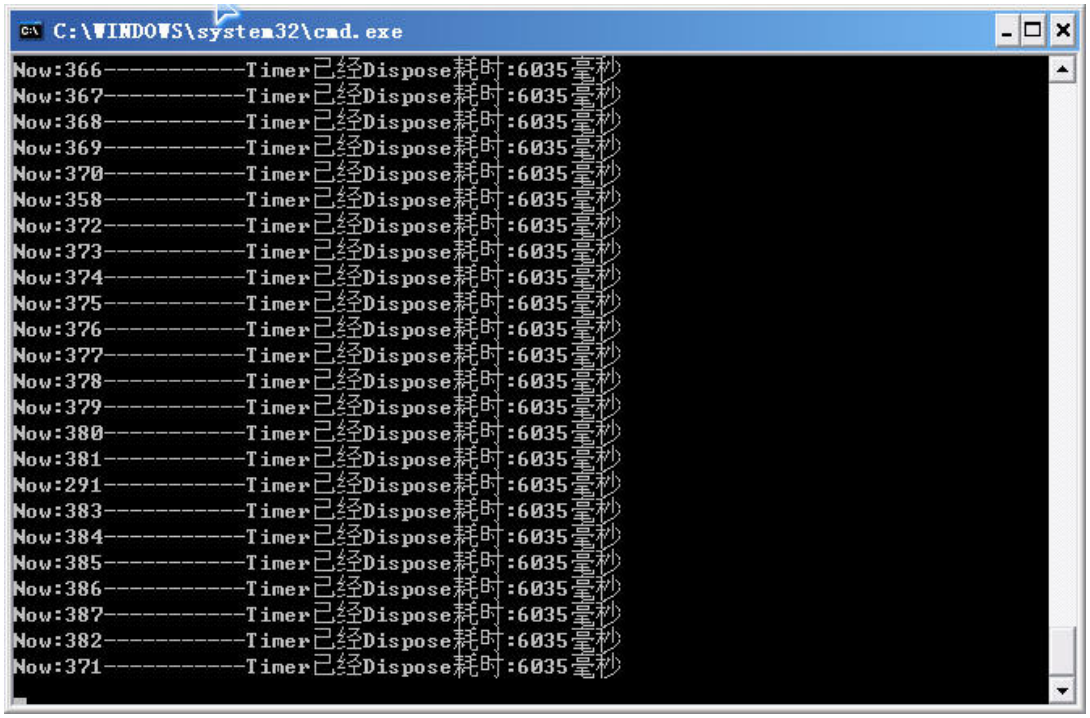
首先我们来分析下 UnSafeTimer

```
class UnSafeTimer
{
    static int i = 0;
    static System.Threading.Timer timer;
    static object mylock = new object();
    static int sleep;
    static bool flag;
    public static Stopwatch sw = new Stopwatch();

    static void Excute(object obj)
    {
        Thread.CurrentThread.IsBackground = false;
        int c;

        lock (mylock)
        {
            i++;
            c = i;
        }

        if (c == 80)
        {
            timer.Dispose();
//执行 Dispose 后 Timer 就不会再申请新的线程了,但是还是会给 Timmer 已经激发的事件申请线程
```

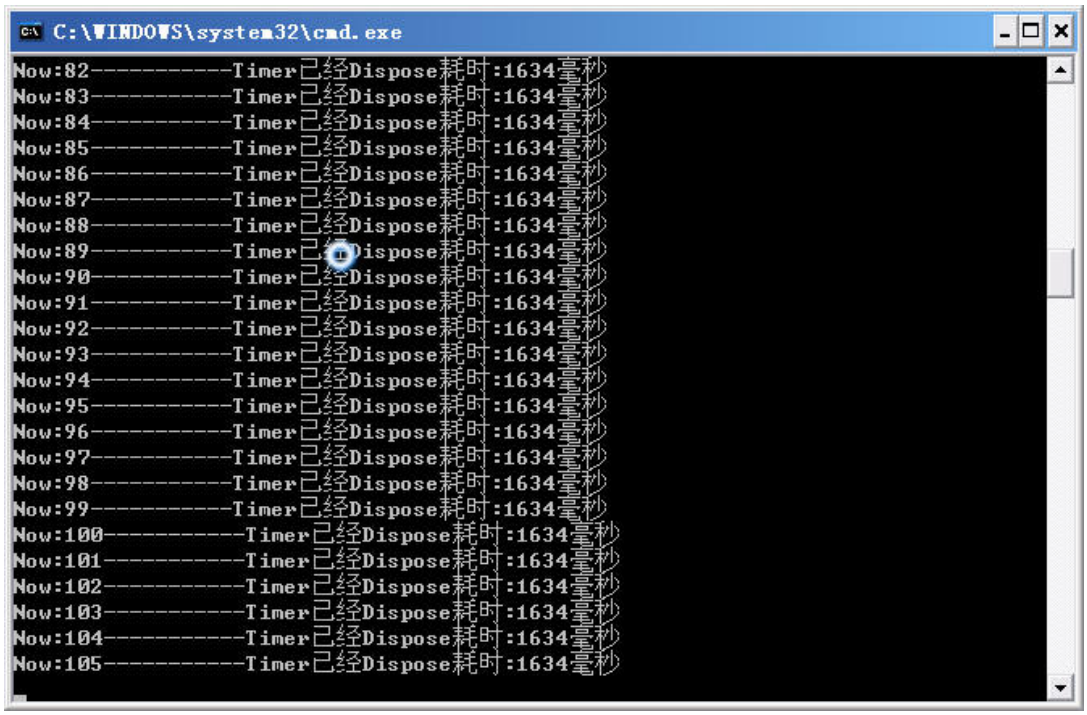
```
            sw.Stop();
        }

        if (c < 80)
            Console.WriteLine("Now:" + c.ToString());
        else
        {
            Console.WriteLine("Now:" + c.ToString() + "-----------Timer 已经 Dispose 耗时:" +
sw.ElapsedMilliseconds.ToString() + "毫秒");
        }

        if (flag)
        {
            Thread.Sleep(sleep);//模拟花时间的代码
        }
        else
        {
            if (i <= 80)
                Thread.Sleep(sleep);//前 80 次模拟花时间的代码
        }
    }

    public static void Init(int p_sleep, bool p_flag)
    {
        sleep = p_sleep;
        flag = p_flag;
        timer = new System.Threading.Timer(Excute, null, 0, 10);
    }
}
```

你可以执行试一试，在输入是否执行安全方法的时候选 N，等待时间 1000，申请了 80 个线程后 Timer 剩余激发的线程选 N，本来想在 80 次的时候停下来，可是你会发现直到执行到 660 多次之后才停下来（具体看机器配置），申请前 80 个线程的时间为 10532 毫秒，反正执行的次数大大超出了限制的 80 次，回头想想让 Timer 不在激发事件的方法是调用 timer.Dispose()，难不成是 Dispose 有延迟？延迟的过程中多执行了 500 多次？那么我们再来做个试验，我们在申请了 80 个线程后 Timer 剩余激发的线程选 y，请耐心等待结果，在最后你会发现执行时间还是 660 次左右，这很显然是不合理的，如果 Dispose 有延迟时间造成所执行 500 多次，那么加长 80 次后面每个线程的申请时间在相同的延迟时间内申请的线程数应该减少，因为后面 500 多个线程每个线程都要执行 1000 毫秒，那么势必有些线程会去申请新的线程有半秒钟的等待时间（你会发现申请了 80 个线程后 Timer 剩余激发的线程选 y 明显比选 n 慢得多，就是因为这个原因），所以看来不是因为 Dispose 造成的。

那么会是什么呢？我们这次这样选在输入是否执行安全方法的时候选 N，等待时间 500，申请了 80 个线程后 Timer 剩余激发的线程选 N。



那么会是什么呢？我们这次这样选在输入是否执行安全方法的时候选 N，等待时间 50，申请了 80 个线程后 Timer 剩余激发的线程选 N



我们发现随着每次任务等待时间的减少多执行的次数也在减少，最关键的一点我们从图中可以看到，前 80 次任务申请的时间也在减少，这是最关键的，根据上面线程池所讲的内容我们可以归纳出：每次任务的等待时间直接决定了前 80 个任务的执行时间，因为等待时

间越短，每个任务就可以越快执行完，那么 80 个任务中就有越多的任务可以用到前面任务执行完后释放掉的线程，也就有越多的任务不必去线程池申请新的线程避免多等待半秒钟的申请时间，而 Timer 并不会去关心线程池申请前 80 个任务的时间长短，只要它没有执行到 timer.Dispose(),它就会每隔 10 毫秒激发一次响应时间，不管前 80 次任务执行时间是长还是短，timer 都在第 80 次任务才执行 Dispose，执行 Dispose 后 timer 就不会激发新的事件了，但是如果前 80 次任务申请的时间越长，那么 timer 就会在前 80 次任务申请的时间内激发越多响应事件，那么线程池中等待队列中就会有越多的响应函数等待申请线程，System.Threading.Timer 没有机制取消线程池等待队列中多余的申请数，所以导致等待时间越长，80 次后执行的任务数越多。

由此只用 timer.Dispose()来终止 Timer 激发事件是不安全的，所以又写了个安全的执行机制：

```
class SafeTimer
{
    static int i = 0;
    static System.Threading.Timer timer;

    static bool flag = true;
    static object mylock = new object();

    static void Excute(object obj)
    {
        Thread.CurrentThread.IsBackground = false;

        lock (mylock)
        {
            if (!flag)
            {
                return;
            }

            i++;

            if (i == 80)
            {
                timer.Dispose();
                flag = false;
            }
            Console.WriteLine("Now:" + i.ToString());
        }

        Thread.Sleep(1000);//模拟花时间的代码
    }
```

```
    public static void Init()
    {
        timer = new System.Threading.Timer(Excute, null, 0, 10);
    }
}
```

安全类中我们用了个 bool 类型的变量 flag 来判断当前是否执行到 80 次了，执行到 80 次后将 flag 置为 false，然后 timer.Dispose，这时虽然任务还是要多执行很多次但是由于 flag 为 false，Excute 函数一开始就做了判断 flag 为 false 会立即退出，Excute 函数 80 次后相当于就不执行了。

3、System.Timers.Timer

在上面的例子中我们看到 System.Threading.Timer 很不安全，即使在安全的方法类，也只能让事件响应函数在 80 次后立刻退出让其执行时间近似于 0，但是还是浪费了系统不少的资源。

所以本人更推荐使用现在介绍的 System.Timers.Timer,System.Timers.Timer 大致原理和 System.Threading.Timer 差不多，唯一几处不同的就是：

- 构造函数不同，构造函数可以什么事情也不做，也可以传入响应间隔时间：System.Timers.Timer timer = new System.Timers.Timer(10);
- 响应事件的响应函数不在构造函数中设置：timer.Elapsed += new ElapsedEventHandler(timer_Elapsed);
- 声明 System.Timers.Timer 对象后他不会自动执行，需要调用 timer.Start() 或者 timer.Enabled = true 来启动它， timer.Start()的内部原理还是设置 timer.Enabled = true
- 调用 timer.Stop()或者 timer.Enabled = false 来停止引发 Elapsed 事件， timer.Stop()的内部原理还是设置 timer.Enabled = false，最重要的是 timer.Enabled = false 后会取消线程池中当前等待队列中剩余任务的执行。

那么我们来看个例子：

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Timers;
using System.Threading;

namespace ConsoleApplication2
{
    class UnSafeTimer
    {
        static int i = 0;
```

```
        static System.Timers.Timer timer;
        static object mylock = new object();

        public static void Init()
        {
            timer = new System.Timers.Timer(10);
            timer.Elapsed += new ElapsedEventHandler(timer_Elapsed);
            timer.Start();
        }

        static void timer_Elapsed(object sender, ElapsedEventArgs e)
        {
            Thread.CurrentThread.IsBackground = false;
            int c;

            lock (mylock)
            {
                i++;
                c = i;
            }

            Console.WriteLine("Now:" + i.ToString());

            if (c == 80)
            {
                timer.Stop();
//可应看到System.Timers.Timer的叫停机制比System.Threading.Timer好得多，就算在不安全的代码下Timer也最多多执行一两次(我在试验中发现有时会执行到81或82)，说明Stop方法在设置Timer的Enable为false后不仅让Timer不再激发响应事件，还取消了线程池等待队列中等待获得线程的任务，至于那多执行的一两次任务我个人认为是Stop执行过程中会耗费一段时间才将Timer的Enable设置为false，这段时间多余的一两个任务就获得了线程开始执行
            }


            Thread.Sleep(1000);
//等待1000毫秒模拟花时间的代码，注意：这里的等待时间直接决定了80(由于是不安全模式有时会是81或82、83)个任务的执行时间，因为等待时间越短，每个任务就可以越快执行完，那么80个任务中就有越多的任务可以用到前面任务执行完后释放掉的线程，也就有越多的任务不必去线程池申请新的线程避免多等待半秒钟的申请时间
        }
    }

    class SafeTimer
    {
```

```
        static int i = 0;
        static System.Timers.Timer timer;

        static bool flag = true;
        static object mylock = new object();

        public static void Init()
        {
            timer = new System.Timers.Timer(10);
            timer.Elapsed += new ElapsedEventHandler(timer_Elapsed);
            timer.Start();
        }

        static void timer_Elapsed(object sender, ElapsedEventArgs e)
        {
            Thread.CurrentThread.IsBackground = false;

            lock (mylock)
            {
                if (!flag)
                {
                    return;
                }
                i++;

                Console.WriteLine("Now:" + i.ToString());

                if (i == 80)
                {
                    timer.Stop();
                    flag = false;
                }
            }

            Thread.Sleep(1000);//同 UnSafeTimer
        }

        class Program
        {
            static void Main(string[] args)
            {
                Console.Write("是否使用安全 Timer>(Y/N)?");
                string Key = Console.ReadLine();
```

```
                if (Key.ToLower() == "y")
                    SafeTimer.Init();
                else
                    UnSafeTimer.Init();

                Console.ReadLine();
            }
        }
    }
}
```

这个例子和 System.Threading.Timer 差不多，这里也分为:安全类 SafeTimer 和不安全类 UnSafeTimer,原因是 timer.Stop()有少许的延迟时间有时任务会执行到 81~83，但是就算是不安全方法也就最多多执行几次，不像 System.Threading.Timer 多执行上百次...

所以我这里还是推荐大家使用 System.Timers.Timer。

2、Timer范例教程

2.1、版权声明

文章出处：http://www.cnblogs.com/codingsilence/articles/1205734.html
文章作者：codingsilence

2.2、内容详情

这是一个关于 Timer 的例子，我们将创建一个简单的应用程序，使用 Timer 对象来倒计时一个由自己设定的时间并循环播放一段音乐，直到重设 Timer 控件。

Timer 对象基础

首先你要知道的是，使用 Timer 对象你需要访问如下命名空间：

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

接下来，介绍一下创建一个 Timer 的要点以及为这个 timer 对象的 Elapsed 事件设定事件委派。

先创建一个 Timer 对象，这里定义使用的 timer 为 timerClock。接下来设定 Elapsed 事件委派，当事件被触发时，指定的委派将被调用，这里定义使用的委派名称为 OnTimer()。接着，设定 Interval 属性，使用毫秒数值指示希望 Elapsed 事件被调用的间隔，这意味着，当定义 Interval 属性为 1000 毫秒时，定义的委派 OnTimer()将每隔 1000 毫秒被调用一次，或者说是每隔 1 秒。

最后，需要设定 Enabled 属性为 true,以使这个 timer 对象开始工作。接下来，剩下的只是一个小问题—创建一个委派，在这个 timer 对象的 Elapsed 属性被触发时调用。如果以前没有使用过委派，不用担心，它们很容易使用，只需要创建一个方法，用来接收适合你捕获事件的一些变量。

针对 Elapsed 事件，这个委派需要接收一个普通对象和一个 ElapsedEventArgs 对象。

```
private System.Timers.Timer timerClock = new System.Timers.Timer();
timerClock.Elapsed += new ElapsedEventHandler(OnTimer);
timerClock.Interval = 1000;
timerClock.Enabled = true;
```

```
public void OnTimer( Object source, ElapsedEventArgs e )
{
    //Your code here
}
```

在报警程序中使用 Timer 控件

介绍了这些基础，现在来看在实际应用中的代码。注意，这里并不包括播放音乐和显示最小化图标的代码。

在下面的代码中，可以看到，将实例化 Timer 对象的方法放在初始化方法 InitializeTimer()中，这个方法将被类构造调用。并且创建了两个方法，inputToSeconds()和 secondsToTime()用来将字符串格式的时间格式转换为正型，以及一个反处理过程。这些方法只是用来帮助我们在 TextBox 控件中显示日期格式，这在整个应用的结构中，并不十分重要。其他的那些代码，是标准的 Visual Studio.NET 为 Win Form 程序生成的样板文件。

```
using System;
using System.Drawing;
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;
using System.Data;
using System.Threading;
using System.Timers;
using System.IO;
using System.Reflection;

namespace timerAlarm
{
    public class TimerForm : System.Windows.Forms.Form
    {
        //Controls and Components
        private System.Windows.Forms.TextBox timerInput;
        private System.Windows.Forms.Button StartButton;
        private System.Windows.Forms.Button ResetButton;
        private System.ComponentModel.IContainer components;
        //Timer and associated variables
        private System.Timers.Timer timerClock = new System.Timers.Timer();
        private int clockTime = 0;
        private int alarmTime = 0;

        public TimerForm()
        {
```

```
            InitializeComponent();
            InitializeTimer();
        }

        protected override void Dispose(bool disposing)
        {
            if (disposing)
            {
                if (components != null)
                {
                    components.Dispose();
                }
            }
            base.Dispose(disposing);
        }

        #region Windows Form Designer generated code
        #endregion

        public void InitializeTimer()
        {
            this.timerClock.Elapsed += new ElapsedEventHandler(OnTimer);
            this.timerClock.Interval = 1000;
            this.timerClock.Enabled = true;
        }

        [STAThread]
        static void Main()
        {
            Application.Run(new TimerForm());
        }

        private void TimerForm_Resized(object sender, System.EventArgs e)
        {
            if (this.WindowState == FormWindowState.Minimized)
            {
                this.Hide();
            }
        }

        private void StartButton_Click(object sender, System.EventArgs e)
        {
            this.clockTime = 0;
            inputToSeconds(this.timerInput.Text);
```

```
        }

        private void ResetButton_Click(object sender, System.EventArgs e)
        {
            try
            {
                this.clockTime = 0;
                this.alarmTime = 0;
                this.timerInput.Text = "00:00:00";
            }
            catch (Exception ex)
            {
                MessageBox.Show("ResetButton_Click(): " + ex.Message);
            }
        }

        public void OnTimer(Object source, ElapsedEventArgs e)
        {
            try
            {
                this.clockTime++;
                int countdown = this.alarmTime - this.clockTime;
                if (this.alarmTime != 0)
                {
                    this.timerInput.Text = secondsToTime(countdown);
                }

                //Sound Alarm
                if (this.clockTime == this.alarmTime)
                {
                    MessageBox.Show("Play Sound");
                }
            }
            catch (Exception ex)
            {
                MessageBox.Show("OnTimer(): " + ex.Message);
            }
        }

        private void inputToSeconds(string timerInput)
        {
            try
            {
                string[] timeArray = new string[3];
```

```
            int minutes = 0;
            int hours = 0;
            int seconds = 0;
            int occurence = 0;
            int length = 0;

            occurence = timerInput.LastIndexOf(":");
            length = timerInput.Length;

            //Check for invalid input
            if (occurence == -1 || length != 8)
            {
                MessageBox.Show("Invalid Time Format.");
                ResetButton_Click(null, null);
            }
            else
            {
                timeArray = timerInput.Split(':');

                seconds = Convert.ToInt32(timeArray[2]);
                minutes = Convert.ToInt32(timeArray[1]);
                hours = Convert.ToInt32(timeArray[0]);

                this.alarmTime += seconds;
                this.alarmTime += minutes * 60;
                this.alarmTime += (hours * 60) * 60;
            }
        }
        catch (Exception e)
        {
            MessageBox.Show("inputToSeconds(): " + e.Message);
        }
    }

    public string secondsToTime(int seconds)
    {
        int minutes = 0;
        int hours = 0;

        while (seconds >= 60)
        {
            minutes += 1;
            seconds -= 60;
        }
```

```
            while (minutes >= 60)
            {
                hours += 1;
                minutes -= 60;
            }

            string strHours = hours.ToString();
            string strMinutes = minutes.ToString();
            string strSeconds = seconds.ToString();

            if (strHours.Length < 2)
                strHours = "0" + strHours;
            if (strMinutes.Length < 2)
                strMinutes = "0" + strMinutes;
            if (strSeconds.Length < 2)
                strSeconds = "0" + strSeconds;

            return strHours + ":" + strMinutes + ":" + strSeconds;
        }
    }
}
```

实际的执行代码比上面的要多，本范例演示了 Timer 在实际环境中的一个简单应用，仅仅使用了一些简单的基础知识来创建一个简单的应用。

3、Socket网络编程大全

3.1、版权声明

文章出处：http://wenku.baidu.com/view/360429ea81c758f5f61f67d4.html
文章作者：无尾兽零号机

3.2、内容详情

3.2.1 简单服务器端

```
using System.Data;
using System.Net.Sockets;
using System.Net;
using System.Threading;
private static int port = %%2;
private static Thread thThreadRead;
private static TcpListener TcpListen;
private static bool bListener = true;
private static Socket stRead;

private static void Listen()
{
    try
    {
        TcpListen = new TcpListener(port);
        TcpListen.Start();
        stRead = TcpListen.AcceptSocket();
        EndPoint tempRemoteEP = stRead.RemoteEndPoint;
        IPEndPoint tempRemoteIP = (IPEndPoint)tempRemoteEP;
        IPHostEntry host = Dns.GetHostByAddress(tempRemoteIP.Address);
        string sHostName = host.HostName;
        while (bListener)
        {
            stRead.Send(Encoding.ASCII.GetBytes(%%1));
            string sTime = DateTime.Now.ToShortTimeString();
            Byte[] byRead = new Byte[1024];
            int iRead = stRead.ReceiveFrom(byRead, ref tempRemoteEP);
```

```
                Byte[] byText = new Byte[iRead];
                Array.Copy(byRead, 0, byText, 0, iRead);
                string line = System.Text.Encoding.Default.GetString(byRead);
            }
        }
        catch (System.Security.SecurityException)
        {
            //监听失败
        }
}
thThreadRead = new Thread(new ThreadStart(Listen));
thThreadRead.Start();
```

3.2.2 简单客户端

```
using System.Data;
using System.Net.Sockets;
using System.Net;
private static IPEndPoint dateTimeHost;

string hostIPString=%%1;
string hostPortString=%%2;
IPAddress hostIP=IPAddress.Parse(hostIPString);
dateTimeHost=new IPEndPoint(hostIP,Int32.Parse(hostPortString));
Socket conn=new Socket(AddressFamily.InterNetwork,SocketType.Stream,ProtocolType.Tcp);
conn.Connect(dateTimeHost);
int bytes=0;
Byte[] RecvBytes=new Byte[256];
bytes=conn.Receive(RecvBytes,RecvBytes.Length,0);
string RecvString=Encoding.ASCII.GetString(RecvBytes,0,bytes);
Console.WriteLine(RecvString);
conn.Shutdown(SocketShutdown.Both);
conn.Close();
```

3.2.3 获得本机IP

```
using System.Net;
IPAddress[] addressList = Dns.GetHostByName(Dns.GetHostName()).AddressList;
```

```
string %%1=null;
for (int i = 0; i < addressList.Length; i++)
{
    %%1 += addressList[i].ToString();
}
```

3.2.4 端对端通信

```
using System.Net;
using System.Net.Sockets;

UdpClient client=new UdpClient(%%2);
IPAddress a=IPAddress.Parse("127001");
IPEndPoint receivePoint=new IPEndPoint(a,%%2);
IPAddress HostIP=null;
byte[] sendData=Encoding.UTF8.GetBytes(%%3);
byte[] recData;
try{
    HostIP=IPAddress.Parse(%%1);
}
catch {
    recData=client.Receive(ref receivePoint);
    %%3=Encoding.UTF8.GetString(recData);
    client.Send(sendData,sendData.Length,%%4,%%2);
    client.Close();
}
IPEndPoint host=new IPEndPoint(HostIP,%%2);
recData=client.Receive(ref receivePoint);
%%3=Encoding.UTF8.GetString(recData);
client.Close();
```

3.2.5 点对点通信

```
using System.Data;
using System.Net.Sockets;
using System.Net;
using System.Threading;
```

```
Thread th;
TcpListener tpListen1;
bool listenerRun=true;
NetworkStream tcpStream;
StreamWriter reqStreamW;
TcpClient tcpc;
Socket skSocket;
protected void Listen()
{
    try{
        tpListen1=new TcpListener(Int32.Parse(%%2));
        tpListen1.Start();
        skSocket=tpListen1.AcceptSocket();
        EndPoint tempRemoteEP=skSocket.RemoteEndPoint;
        IPEndPoint tempRemoteIP=(IPEndPoint)tempRemoteEP;
        IPHostEntry host=Dns.GetHostByAddress(tempRemoteIP.Address);
        string HostName=host.HostName;
        while(listenerRun)
        {
            Byte[] stream=new Byte[1024];
            string time=DateTime.Now.ToString();
            int i=skSocket.ReceiveFrom(stream,ref tempRemoteEP);
            string %%5=Encoding.UTF8.GetString(stream);
            //指定编码，从缓冲区中解析出内容
            //time+" "+HostName+":"
        }
    }
    catch(Security.SecurityException)
    {
        //防火墙安全错误！
    }
    try{
        string sMsg=%%4;
        string MyName=Dns.GetHostName();
        reqStreamW=new StreamWriter(tcpStream);
        reqStreamW.Write(sMsg);
        reqStreamW.Flush();
        string time=DateTime.Now.ToString();
        //显示传送的数据和时间
        //time+" "+MyName+":"
        //sMsg
    }
    catch(Exception)
```

```
    {
        //无法发送信息到目标计算机!
    }

    protected override void Dispose(bool disposing)
    {
        try{
            listenerRun=false;
            th.Abort();
            th=null;
            tpListen1.Stop();
            skSocket.Close();
            tcpc.Close();
        }
        catch{}
        if(disposing && component!=null)
        {
            components.Dispose();
        }
    }
    base.Dispose(disposing);
}
```

3.2.6 UDP对时服务器端

```
using System.Data;
using System.Net;
using System.Net.Sockets;
using System.Threading;
using System.Text;

public void start_server()
{
    while(true)
    {
        byte[] recData=server.Receive(ref receivePoint);
        ASCIIEncoding encode=new ASCIIEncoding();
        string Read_str=encode.GetString(recData));
        string[] tem=Read_str.Split('/');
        byte[] sendData=encode.GetBytes(DataTime.Now.ToString());
        server.Send(sendData,sendData.Length,temp[0],Int32.Parse(temp[1]));
```

```
    }
}
UdpClient server=new UdpClient(%%1);
IPEndPoint receivePoint=new IPEndPoint(new IPAddress("127001",%%1);
Thread startServer=new Thread(new ThreadStart(start_server));
startServer.Start();

protected override void Dispose(bool disposing)
{
    try{
        startServer.Abort();
        server.Close();
    }
    catch{}
    if(disposing && component!=null)
    {
        components.Dispose();
    }
}
base.Dispose(disposing);
}
```

3.2.7 UDP对时客户端

```
using System.Data;
using System.Net;
using System.Net.Sockets;
using System.Runtime.InteropServices;

[DllImport("Kernel32.dll")]
private static bool SetSystemTime(SystemTime time);
public class SystemTime
{
    public short year;
    public short Month;
    public short DayOfWeek;
    public short Day;
    public short Hour;
    public short Minute;
    public short Second;
    public short Milliseconds;
```

```
}
UdpClient client=new UdpClient(%%2);
IPEndPoint receivePoint=new IPEndPoint(a,%%2);
IPAddress a=new IPAddress.Parse("127001");
string timeString=DateTime.Now.ToString();
DataTime temp;
IPAddress HostIP;
bool continueLoop=true;
while(continueLoop)
{
    string hostName=Dns.GetHostName();
    Text.ASCIIEncoding encode=new Text.ASCIIEncoding();
    string sendString=hostName+"/"+%%2.ToString();
    byte[] sendData=encode.GetBytes(sendString);
    byte[] recData;
    try{
        HostIP=IPAddress.Parse(%%1);
    }
    catch {
        client.Send(sendData,sendData.Length,%%1,%%2);
        recData=client.Receive(ref receivePoint);
        timeString=encode.GetString(recData);
        client.Close();
        continueLoop=false;
        return;
    }
    IPEndPoint host=new IPEndPoint(HostIP,%%2);
    client.Send(sendData,sendData.Length,host);
    recData=client.Receive(ref receivePoint);
    timeString=encode.GetString(recData);
    client.Close();
    continueLoop=false;
}
try{
    temp=DataTime.Parse(timeString);
}
catch {
    return;
}
SystemTime st=new SystemTime();
st.year=(short)temp.Year;
st.Month=(short)temp.Month;
st.DayOfWeek=(short)temp.DayOfWeek;
st.Day=(short)temp.Day;
```

```
st.Hour=Conver.ToInt16(temp.Hour);
if(st.Hour>=12)
{
    st.Hour-=(short)8;
}
else if(st.Hour>=8)
{
    st.Hour-=(short)8;
}
else
{
    st.Hour+=(short)16;
}
st.Minute=Convert.ToInt16(temp.Minute);
st.Second=Convert.ToInt16(temp.Second);
st.Milliseconds=Convert.ToInt16(temp.Milliseconds);
if(SetSystemTime(st))
{
    //修改成功
}
else
{
    //修改不成功
}
```

3.2.8 点对点传输文件

```
using System.Sockes;
/*
System.Sockes 命名空间了实现 Berkeley 套接字接口。通过这个类，我们可以实现网络计算机之间的消息传输和发送.而在我下面要讨论的这个议题里,我们将讨论的是用套节子实现文件的传输.这种方法有别于 FTP 协议实现的的文件传输方法,利用 ftp 的方法需要一个专门的服务器和客户端,无疑于我们要实现的点对点的文件传输太为复杂了一些。在这里，我们实现一个轻量级的方法来实现点对点的文件传输，这样就达到了 intenet 上任何两个计算机的文件共享。
在两台计算机传输文件之前，必需得先有一台计算机建立套节子连接并绑定一个固定得端口，并在这个端口侦听另外一台计算机的连接请求。
*/
socket = new Socket(AddressFamily.InterNetwork,SocketType.Stream, ProtocolType.Tcp);
socket.Blocking = true ;
IPEndPoint computernode1 = new IPEndPoint(serverIpadress, 8080);
socket.Bind(computernode1);
socket.Listen(-1);
```

```
//当有其他的计算机发出连接请求的时候，被请求的计算机将对每一个连接请求分配一个线程，用于处理文件传输和其他服务。

while (true)
{
    clientsock = socket.Accept();
    if ( clientsock.Connected )
    {
        Thread tc = new Thread(new ThreadStart(listenclient));
        tc.Start();
    }
}
//下面的代码展示了 listenclient 方法是如何处理另外一台计算机发送过来的请求。首先并对发送过来的请求字符串作出判断，看看是何种请求，然后决定相应的处理方法。
void listenclient()
{
    Socket sock = clientsock ;
    try
    {
        while ( sock != null )
        {
            byte[] recs = new byte[32767];
            int rcount = sock.Receive(recs,recs.Length,0) ;
            string message = System.Text.Encoding.ASCII.GetString(recs) ;
            //对 message 作出处理，解析处请求字符和参数存储在 cmdList 中
            execmd=cmdList[0];
            sender = null ;
            sender = new Byte[32767];
            string parm1 = "";
            //目录列举
            if ( execmd == "LISTING" )
            {
                ListFiles(message);
                continue ;
            }
            //文件传输
            if ( execmd == "GETOK" )
            {
                cmd = "BEGINSEND " + filepath + " " + filesize ;
                sender = new Byte[1024];
                sender = Encoding.ASCII.GetBytes(cmd);
                sock.Send(sender, sender.Length , 0 );
                //转到文件下载处理
                DownloadingFile(sock);
```

```
                    continue ;
                }
            }
        }
        catch(Exception Se)
        {
            string s = Se.Message;
            Console.WriteLine(s);
        }
}
//至此，基本的工作已经完成了，下面我们看看如何处理文件传输的。
while(rdby < total && nfs.CanWrite)
{
    //从要传输的文件读取指定长度的数据
    len =fin.Read(buffed,0,buffed.Length) ;
    //将读取的数据发送到对应的计算机
    nfs.Write(buffed, 0,len);
    //增加已经发送的长度
    rdby=rdby+len ;
}
//从上面的代码可以看出是完成文件转换成 FileStream 流，然后通过 NetworkStream 绑定对应的套节子，
最后调用他的 write 方法发送到对应的计算机。
//我们再看看接受端是如何接受传输过来的流，并且转换成文件的：
NetworkStream nfs = new NetworkStream(sock) ;
try
{
    //一直循环直到指定的文件长度
    while(rby < size)
    {
        byte[] buffer = new byte[1024] ;
        //读取发送过来的文件流
        int i = nfs.Read(buffer,0,buffer.Length) ;
        fout.Write(buffer,0,(int)i) ;
        rby=rby+i ;
    }
    fout.Close();
}
catch(Exception){}
/*
从上面可以看出接受与发送恰好是互为相反的过程，非常简单。
至此，单方向的文件传输就完成了，只需要在每个对等的节点上同时实现上面的发送和接受的处理代码就
可以做到互相传输文件了。
*/
```

3.2.9 发送邮件

```
//using System.Net.Mail;
string strSmtpServer=%%1; //"100.100.100.100"
string strFrom=%%2; //"someone@xxx.com"
string strFromPass=%%3; //"someone@xxx.com"
string strto=%%4; //"xxxx"
string strSubject=%%5;//"webtest"
string strBody=%%6; TextBox1
SmtpClient client = new SmtpClient(strSmtpServer);
client.UseDefaultCredentials = false;
client.Credentials = new System.Net.NetworkCredential(strFrom, strFromPass);
client.DeliveryMethod = SmtpDeliveryMethod.Network;
MailMessage message = new MailMessage(strFrom, strto, strSubject, strBody);
Attachment attachment = new Attachment("c:\\log.log");
message.Attachments.Add(attachment);
message.BodyEncoding = System.Text.Encoding.UTF8;
message.IsBodyHtml = true;
client.Send(message);
```

3.2.10 接收邮件

```
/*
using System.Net.Sockets;
using System.Net;
using System.Security.Cryptography;
using System.IO;
*/
// 类名：Pop3
// 功能：接收电子邮件

namespace ZTSX.Email
{
    /// <summary>
    /// Pop3 的摘要说明。
    /// </summary>
    public class Pop3
    {
        private string mstrHost = null; //主机名称或 IP 地址
```

```
        private int mintPort = 110; //主机的端口号（默认为 110）
        private TcpClient mtcpClient = null; //客户端
        private NetworkStream mnetStream = null; //网络基础数据流
        private StreamReader m_stmReader = null; //读取字节流
        private string mstrStatMessage = null; //执行 STAT 命令后得到的消息（从中得到邮件数）

        /// <summary>
        /// 构造函数
        /// </summary>
        /// <remarks>一个邮件接收对象</remarks>
        public Pop3()
        {
        }

        /// <summary>
        /// 构造函数
        /// </summary>
        /// <param name="host">主机名称或 IP 地址</param>
        public Pop3(string host)
        {
            mstrHost = host;
        }

        /// <summary>
        /// 构造函数
        /// </summary>
        /// <param name="host">主机名称或 IP 地址</param>
        /// <param name="port">主机的端口号</param>
        /// <remarks>一个邮件接收对象</remarks>
        public Pop3(string host, int port)
        {
            mstrHost = host;
            mintPort = port;
        }

        #region 属性

        /// <summary>
        /// 主机名称或 IP 地址
        /// </summary>
        /// <remarks>主机名称或 IP 地址</remarks>
        public string HostName
        {
            get { return mstrHost; }
```

```
            set { mstrHost = value; }
        }

        /// <summary>
        /// 主机的端口号
        /// </summary>
        /// <remarks>主机的端口号</remarks>
        public int Port
        {
            get { return mintPort; }
            set { mintPort = value; }
        }

        #endregion

        #region 私有方法

        /// <summary>
        /// 向网络访问的基础数据流中写数据（发送命令码）
        /// </summary>
        /// <param name="netStream">可以用于网络访问的基础数据流</param>
        /// <param name="command">命令行</param>
        /// <remarks>向网络访问的基础数据流中写数据（发送命令码）</remarks>
        private void WriteToNetStream(ref NetworkStream netStream, String command)
        {
            string strToSend = command + "\r\n";
            byte[] arrayToSend =
System.Text.Encoding.ASCII.GetBytes(strToSend.ToCharArray());
            netStream.Write(arrayToSend, 0, arrayToSend.Length);
        }

        /// <summary>
        /// 检查命令行结果是否正确
        /// </summary>
        /// <param name="message">命令行的执行结果</param>
        /// <param name="check">正确标志</param>
        /// <returns>
        /// 类型：布尔
        /// 内容：true 表示没有错误，false 为有错误
        /// </returns>
        /// <remarks>检查命令行结果是否有错误</remarks>
        private bool CheckCorrect(string message, string check)
        {
            if (message.IndexOf(check) == -1)
```

```
            return false;
        else
            return true;
    }

    /// <summary>
    /// 邮箱中的未读邮件数
    /// </summary>
    /// <param name="message">执行完 LIST 命令后的结果</param>
    /// <returns>
    /// 类型：整型
    /// 内容：邮箱中的未读邮件数
    /// </returns>
    /// <remarks>邮箱中的未读邮件数</remarks>
    private int GetMailNumber(string message)
    {
        string[] strMessage = message.Split(' ');
        return Int32.Parse(strMessage[1]);
    }

    /// <summary>
    /// 得到经过解码后的邮件的内容
    /// </summary>
    /// <param name="encodingContent">解码前的邮件的内容</param>
    /// <returns>
    /// 类型：字符串
    /// 内容：解码后的邮件的内容
    /// </returns>
    /// <remarks>得到解码后的邮件的内容</remarks>
    private string GetDecodeMailContent(string encodingContent)
    {
        string strContent = encodingContent.Trim();
        string strEncode = null;

        int iStart = strContent.IndexOf("Base64");
        if (iStart == -1)
            throw new Pop3Exception("邮件内容不是 Base64 编码，请检查");
        else
        {
            strEncode = strContent.Substring(iStart + 6, strContent.Length - iStart - 6);
            try
            {
                return SX.Encode.TransformToString(strEncode);
            }
```

```
            catch (SX.EncodeException exc)
            {
                throw new Pop3Exception(exc.Message);
            }
        }
    }

    #endregion

    /// <summary>
    /// 与主机建立连接
    /// </summary>
    /// <returns>
    /// 类型：布尔
    /// 内容：连接结果（true 为连接成功，false 为连接失败）
    /// </returns>
    /// <remarks>与主机建立连接</remarks>
    public bool Connect()
    {
        if (mstrHost == null)
            throw new Exception("请提供 SMTP 主机名称或 IP 地址！");
        if (mintPort == 0)
            throw new Exception("请提供 SMTP 主机的端口号");
        try
        {
            mtcpClient = new TcpClient(mstrHost, mintPort);
            mnetStream = mtcpClient.GetStream();
            m_stmReader = new StreamReader(mtcpClient.GetStream());

            string strMessage = m_stmReader.ReadLine();
            if (CheckCorrect(strMessage, "+OK") == true)
                return true;
            else
                return false;
        }
        catch (SocketException exc)
        {
            throw new Pop3Exception(exc.ToString());
        }
        catch (NullReferenceException exc)
        {
            throw new Pop3Exception(exc.ToString());
        }
    }
```

```
        #region Pop3 命令

        /// <summary>
        /// 执行 Pop3 命令，并检查执行的结果
        /// </summary>
        /// <param name="command">Pop3 命令行</param>
        /// <returns>
        /// 类型：字符串
        /// 内容：Pop3 命令的执行结果
        /// </returns>
        private string ExecuteCommand(string command)
        {
            string strMessage = null; //执行 Pop3 命令后返回的消息

            try
            {
                //发送命令
                WriteToNetStream(ref mnetStream, command);

                //读取多行
                if (command.Substring(0, 4).Equals("LIST") || command.Substring(0,
4).Equals("RETR") || command.Substring(0, 4).Equals("UIDL")) //记录 STAT 后的消息（其中包含邮
件数）
                {
                    strMessage = ReadMultiLine();

                    if (command.Equals("LIST")) //记录 LIST 后的消息（其中包含邮件数）
                        mstrStatMessage = strMessage;
                }
                //读取单行
                else
                    strMessage = m_stmReader.ReadLine();

                //判断执行结果是否正确
                if (CheckCorrect(strMessage, "+OK"))
                    return strMessage;
                else
                    return "Error";
            }
            catch (IOException exc)
            {
                throw new Pop3Exception(exc.ToString());
            }
```

```
        }

        /// <summary>
        /// 在 Pop3 命令中，LIST、RETR 和 UIDL 命令的结果要返回多行，以点号（.）结尾，
        /// 所以如果想得到正确的结果，必须读取多行
        /// </summary>
        /// <returns>
        /// 类型：字符串
        /// 内容：执行 Pop3 命令后的结果
        /// </returns>
        private string ReadMultiLine()
        {
            string strMessage = m_stmReader.ReadLine();
            string strTemp = null;
            while (strMessage != ".")
            {
                strTemp = strTemp + strMessage;
                strMessage = m_stmReader.ReadLine();
            }
            return strTemp;
        }

        //USER 命令
        private string USER(string user)
        {
            return ExecuteCommand("USER " + user) + "\r\n";
        }

        //PASS 命令
        private string PASS(string password)
        {
            return ExecuteCommand("PASS " + password) + "\r\n";
        }

        //LIST 命令
        private string LIST()
        {
            return ExecuteCommand("LIST") + "\r\n";
        }

        //UIDL 命令
        private string UIDL()
        {
            return ExecuteCommand("UIDL") + "\r\n";
```

```
        }

        //NOOP 命令
        private string NOOP()
        {
            return ExecuteCommand("NOOP") + "\r\n";
        }

        //STAT 命令
        private string STAT()
        {
            return ExecuteCommand("STAT") + "\r\n";
        }

        //RETR 命令
        private string RETR(int number)
        {
            return ExecuteCommand("RETR " + number.ToString()) + "\r\n";
        }

        //DELE 命令
        private string DELE(int number)
        {
            return ExecuteCommand("DELE " + number.ToString()) + "\r\n";
        }

        //QUIT 命令
        private void Quit()
        {
            WriteToNetStream(ref mnetStream, "QUIT");
        }

        /// <summary>
        /// 收取邮件
        /// </summary>
        /// <param name="user">用户名</param>
        /// <param name="password">口令</param>
        /// <returns>
        /// 类型：字符串数组
        /// 内容：解码前的邮件内容
        /// </returns>
        private string[] ReceiveMail(string user, string password)
        {
            int iMailNumber = 0; //邮件数
```

```
        if (USER(user).Equals("Error"))
            throw new Pop3Exception("用户名不正确！");
        if (PASS(password).Equals("Error"))
            throw new Pop3Exception("用户口令不正确！");
        if (STAT().Equals("Error"))
            throw new Pop3Exception("准备接收邮件时发生错误！");
        if (LIST().Equals("Error"))
            throw new Pop3Exception("得到邮件列表时发生错误！");

        try
        {
            iMailNumber = GetMailNumber(mstrStatMessage);

            //没有新邮件
            if (iMailNumber == 0)
                return null;
            else
            {
                string[] strMailContent = new string[iMailNumber];

                for (int i = 1; i <= iMailNumber; i++)
                {
                    //读取邮件内容
                    strMailContent[i - 1] = GetDecodeMailContent(RETR(i));
                }
                return strMailContent;
            }
        }
        catch (Pop3Exception exc)
        {
            throw new Pop3Exception(exc.ToString());
        }
    }

    #endregion


    /// <summary>
    /// 收取邮件
    /// </summary>
    /// <param name="user">用户名</param>
    /// <param name="password">口令</param>
    /// <returns>
```

```
        /// 类型：字符串数组
        /// 内容：解码前的邮件内容
        /// </returns>
        ///<remarks>收取邮箱中的未读邮件</remarks>
        public string[] Receive(string user, string password)
        {
            try
            {
                return ReceiveMail(user, password);
            }
            catch (Pop3Exception exc)
            {
                throw new Pop3Exception(exc.ToString());
            }
        }

        /// <summary>
        /// 断开所有与服务器的会话
        /// </summary>
        /// <remarks>断开所有与服务器的会话</remarks>
        public void DisConnect()
        {
            try
            {
                Quit();
                if (m_stmReader != null)
                    m_stmReader.Close();
                if (mnetStream != null)
                    mnetStream.Close();
                if (mtcpClient != null)
                    mtcpClient.Close();
            }
            catch (SocketException exc)
            {
                throw new Pop3Exception(exc.ToString());
            }
        }

        /// <summary>
        /// 删除邮件
        /// </summary>
        /// <param name="number">邮件号</param>
        public void DeleteMail(int number)
        {
```

```
            //删除邮件
            int iMailNumber = number + 1;
            if (DELE(iMailNumber).Equals("Error"))
                throw new Pop3Exception("删除第" + iMailNumber.ToString() + "时出现错误！");
        }

    }
}
```

4、多线程使用thread、threadpool、timer

4.1、版权声明

文章出处：http://www.cnblogs.com/linzheng/archive/2011/02/21/1960257.html
文章作者：linzheng

4.2、内容详情

在 system.threading 命名空间提供一些使得能进行多线程编程的类和接口，其中线程的创建有以下三种方法：thread、threadpool、timer。下面我就他们的使用方法逐个作一简单介绍。

1. thread

这也许是最复杂的方法，但他提供了对线程的各种灵活控制。首先你必须使用他的构造函数创建一个线程实例，他的参数比较简单，只有一个 threadstart 委托：

```
public thread(threadstart start);
```

然后调用 start（）启动他，当然你能利用他的 priority 属性来设置或获得他的运行优先级（enum threadpriority： normal、 lowest、 highest、 belownormal、 abovenormal）。

见下例：他首先生成了两个线程实例 t1 和 t2，然后分别设置他们的优先级，接着启动两线程（两线程基本相同，只不过他们输出不相同，t1 为“1”，t2 为“2”，根据他们各自输出字符个数比可大致看出他们占用 cpu 时间之比,这也反映出了他们各自的优先级）。

```
static void main(string[] args)
{
    thread t1 = new thread(new threadstart(thread1));
    thread t2 = new thread(new threadstart(thread2));
    t1.priority = threadpriority.belownormal ;
    t2.priority = threadpriority.lowest ;
    t1.start();
    t2.start();
}
public static void thread1()
{
    for (int i = 1; i < 1000; i++)
    {
```

```
        //每运行一个循环就写一个“1” dosth();
        console.write("1");
    }
}
public static void thread2()
{
    for (int i = 0; i < 1000; i++)
    {
        //每运行一个循环就写一个“2” dosth();
        console.write("2");
    }
}
public static void dosth()
{
    //用来模拟复杂运算
    for (int j = 0; j < 10000000; j++)
    {
        int a=15;
        a = a*a*a*a;
    }
}
```

以上程式运行结果为：

```
1111111111111111111111111111111111111111112111111111111111111111111111111111111111111112
1111111111111111111111111111111111111111112111111111111111111111111111111111111111111112
1111111111111111111111111111111111111111112111111111111111111111111111111111111111111112
```

从以上结果我们能看出，t1 线程所占用 cpu 的时间远比 t2 的多，这是因为 t1 的优先级比 t2 的高，若我们把 t1 和 t2 的优先级都设为 normal，那结果是怎么？他们所占用的 cpu 时间会相同吗？是的，正如你所料，见下图：

```
121211221212121212121212121212121212121212121212121212121212121212121212121
212121212121212121212121212121212121212121212121212121212121212121212121212
121212121212121212
```

从上例我们可看出，他的构造类似于 win32 的工作线程，但更加简单，只需把线程要调用的函数作为委托，然后把委托作为参数构造线程实例即可。当调用 start()启动后，便会调用相应的函数，从那函数第一行开始执行。
接下来我们结合线程的 threadstate 属性来了解线程的控制。

threadstate 是个枚举类型，他反映的是线程所处的状态。

当一个 thread 实例刚创建时，他的 threadstate 是 unstarted；当此线程被调用 start()启动之后，他的 threadstate 是 running; 在此线程启动之后，如果想让他暂停（阻塞），能调用 thread.sleep() 方法，他有两个重载方法（sleep(int)、sleep(timespan)），只不过是表示时间量的格式不同而已，当在某线程内调用此函数时，他表示此线程将阻塞一段时间（时间是由传递给 sleep 的毫秒数或 timespan 决定的，但若参数为 0 则表示挂起此线程以使其他线程能够执行，指定 infinite 以无限期阻塞线程），此时他的 threadstate 将变为 waitsleepjoin，另外值得注意一点的是 sleep()函数被定义为了 static？！这也意味着他不能和某个线程实例结合起来用，也即不存在类似于 t1.sleep(10)的调用！正是如此，sleep()函数只能由需“sleep”的线程自己调用，不允许其他线程调用，正如 when to sleep 是个人私事不能由他人决定。不过当某线程处于 waitsleepjoin 状态而又不得不唤醒他时，可使用 thread.interrupt 方法 ，他将在线程上引发 threadinterruptedexception，下面我们先看一个例子(注意 sleep 的调用方法)：

```
static void main(string[] args)
{
    thread t1 = new thread(new threadstart(thread1));
    t1.start();
    t1.interrupt ();
    e.waitone ();
    t1.interrupt ();
    t1.join();
    console.writeline(“t1 is end”);
}
static autoresetevent e = new autoresetevent(false);
public static void thread1()
{
    try {
        //从参数可看出将导致休眠
        thread.sleep(timeout.infinite);
    }
    catch(system.threading.threadinterruptedexception e)
    {
        //中断处理程式
        console.writeline (” 1st interrupt”);
    }
    e.set ();
    try {
        // 休眠
        thread.sleep(timeout.infinite );
    }
    catch(system.threading.threadinterruptedexception e)
    {
        console.writeline (” 2nd interrupt”);
    }
    //暂停 10 秒
```

```
    thread.sleep (10000);
}
```

运行结果为：

```
1st interrupt
2nd interrupt
(10s 后)t1 is end
```

从上例我们能看出 thread.interrupt 方法能把程式从某个阻塞（waitsleepjoin）状态唤醒进入对应的中断处理程式，然后继续往下执行（他的 threadstate 也变为 running），此函数的使用必须注意以下几点：

- 此方法不仅可唤醒由 sleep 导致的阻塞，而且对一切可导致线程进入 waitsleepjoin 状态的方法(如 wait 和 join)都有效。如上例所示，使用时要把导致线程阻塞的方法放入 try 块内， 并把相应的中断处理程式放入 catch 块内。
- 对某一线程调用 interrupt，如他正处于 waitsleepjoin 状态，则进入相应的中断处理程式执行， 若此时他不处于 waitsleepjoin 状态， 则他后来进入此状态时， 将被即时中断。若在中断前调用几次 interrupt， 只有第一次调用有效，这正是上例我用同步的原因，这样才能确保第二次调用 interrupt 在第一个中断后调用，否则的话可能导致第二次调用无效（若他在第一个中断前调用）。你能把同步去掉试试，其结果非常可能是： 1st interrupt

上例还用了另外两个使线程进入 waitsleepjoin 状态的方法：利用同步对象和 thread.join 方法。join 方法的使用比较简单，他表示在调用此方法的当前线程阻塞直至另一线程（此例中是 t1）终止或经过了指定的时间为止(若他还带了时间量参数)，当两个条件（若有）任一出现，他即时结束 waitsleepjoin 状态进入 running 状态(可根据.join 方法的返回值判断为何种条件，为 true，则是线程终止;false 则是时间到)。

线程的暂停还可用 thread.suspend 方法，当某线程处于 running 状态时对他调用 suspend 方法，他将进入 suspendrequested 状态，但他并不会被即时挂起，直到线程到达安全点之后他才能将该线程挂起，此时他将进入 suspended 状态。如对一个已处于 suspended 的线程调用则无效，要恢复运行只需调用 thread.resume 即可。

线程的销毁，对需销毁的线程调用 abort 方法，他会在此线程上引发 threadabortexception。我们可把线程内的一些代码放入 try 块内，并把相应处理代码放入相应的 catch 块内，当线程正执行 try 块内代码时如被调用 abort，他便会跳入相应的 catch 块内执行，执行完 catch 快内的代码后他将终止（若 catch 块内执行了 resetabort 则不同了：他将取消当前 abort 请求，继续向下执行。所以如要确保某线程终止的最佳用 join，如上例）。

2． threadpool

提供一个线程池，该线程池可用于发送工作项、处理异步 I/O、代表其他线程等待以及处理计时器。

线程池（threadpool）是一种相对较简单的方法，他适应于一些需要多个线程而又较短任务（如一些常处于阻塞状态的线程），他的缺点是对创建的线程不能加以控制，也不能设置其优先级。由于每个进程只有一个线程池，当然每个应用程式域也只有一个线程池（对线），所以你将发现 threadpool 类的成员函数都为 static！当你首次调用 threadpool.queueuserworkitem、threadpool.registerwaitforsingleobject 等，便会创建线程池实例。

下面我就线程池当中的两函数作一介绍：

```
public static bool queueuserworkitem( //调用成功则返回 true
    waitcallback callback,//要创建的线程调用的委托
    object state //传递给委托的参数
    )
    //他的另一个重载函数类似,只是委托不带参数而已 此函数的作用是把要创建的线程排队到线程池，
    //当线程池的可用线程数不为零时（线程池有创建线程数的限制，缺身值为 25），便创建此线程，
    //否则就排队到线程池等到他有可用的线程时才创建。
    public static registeredwaithandle registerwaitforsingleobject(
        waithandle waitobject,// 要注册的 waithandle
        waitortimercallback callback,// 线程调用的委托
        object state,//传递给委托的参数
        int timeout,//超时，单位为毫秒，
        bool executeonlyonce file://是/否只执行一次
);
public delegate void waitortimercallback(
    object state,//也即传递给委托的参数
    bool timedout//true 表示由于超时调用，反之则因为 waitobject
    );
```

此函数的作用是创建一个等待线程，一旦调用此函数便创建此线程，在参数 waitobject 变为终止状态或所设定的时间 timeout 到了之前，他都处于“阻塞”状态，值得注意的一点是此“阻塞”和 thread 的 waitsleepjoin 状态有非常大的不同：当某 thread 处于 waitsleepjoin 状态时 cpu 会定期的唤醒他以轮询更新状态信息，然后再次进入 waitsleepjoin 状态，线程的转换可是非常费资源的；而用此函数创建的线程则不同，在触发他运行之前，cpu 不会转换到此线程，他既不占用 cpu 的时间又不浪费线程转换时间，但 cpu 又怎么知道何时运行他？实际上线程池会生成一些辅助线程用来监视这些触发条件，一旦达到条件便启动相应的线程，当然这些辅助线程本身也占用时间，不过如果你需创建较多的等待线程时，使用线程池的优势就越加明显。见下例：

```
static autoresetevent ev=new autoresetevent(false);
public static int main(string[] args)
{
    threadpool.registerwaitforsingleobject(
        ev,
        new waitortimercallback(waitthreadfunc),
        4,
```

```
        2000,
        false//表示每次完成等待操作后都重置计时器，直到注销等待
        );
    threadpool.queueuserworkitem (new waitcallback (threadfunc),8);
    thread.sleep (10000);
    return 0;
}
public static void threadfunc(object b)
{
    console.writeline ("the object is {0}",b);
    for(int i=0;i<2;i++)
    {
        thread.sleep (1000);
        ev.set();
    }
}
public static void waitthreadfunc(object b,bool t)
{
    console.writeline ("the object is {0},t is {1}",b,t);
}
```

其运行结果为：

```
the object is 8
the object is 4,t is false
the object is 4,t is false
the object is 4,t is true
the object is 4,t is true
the object is 4,t is true
```

从以上结果我们能看出线程 threadfunc 运行了 1 次，而 waitthreadfunc 运行了 5 次。我们能从 waitortimercallback 中的 bool t 参数判断启动此线程的原因：t 为 false，则表示由于 waitobject，否则则是由于超时。另外我们也能通过 object b 向线程传递一些参数。

3. timer

提供以指定的时间间隔执行方法的机制。

使用 TimerCallback 委托指定希望 Timer 执行的方法。 计时器委托在构造计时器时指定，并且不能更改。 此方法不在创建计时器的线程上执行，而是在系统提供的 ThreadPool 线程上执行。

创建计时器时，可以指定在第一次执行方法之前等待的时间量（截止时间）以及此后的执行期间等待的时间量（时间周期）。 可以使用 Change 方法更改这些值或禁用计时器。

只要在使用 Timer，就必须保留对它的引用。对于任何托管对象，如果没有对 Timer 的引用，计时器会被垃圾回收。 即使 Timer 仍处在活动状态，也会被回收。
当不再需要计时器时，请使用 Dispose 方法释放计时器持有的资源。 如果希望在计时器被释放时接收到信号，请使用接受 WaitHandle 的 Dispose(WaitHandle) 方法重载。 计时器已被释放后，WaitHandle 便终止。

由计时器执行的回调方法应该是可重入的，因为它是在 ThreadPool 线程上调用的。 在以下两种情况中，此回调可以同时在两个线程池线程上执行：一是计时器间隔小于执行此回调所需的时间；二是所有线程池线程都在使用，此回调被多次排队。

System.Threading.Timer 是一个简单的轻量计时器，它使用回调方法并由线程池线程提供服务。 不建议将其用于 Windows 窗体，因为其回调不在用户界面线程上进行。 System.Windows.Forms.Timer 是用于 Windows 窗体的更佳选择。 要获取基于服务器的计时器功能，可以考虑使用 System.Timers.Timer，它可以引发事件并具有其他功能。

这和 win32 中的 settimer 方法类似。他的构造为：

```
public timer( timercallback callback,      //所需调用的方法
              object state,                //传递给 callback 的参数
              int duetime,                 //多久后开始调用
              callback int period          //调用此方法的时间间隔
                  );
```

// 如果 duetime 为 0，则 callback 即时执行他的首次调用。如果 duetime 为 infinite，则 callback 不调用他的方法。计时器被禁用，但使用 change 方法能重新启用他。如果 period 为 0 或 infinite，并且 duetime 不为 infinite，则 callback 调用他的方法一次。计时器的定期行为被禁用，但使用 change 方法能重新启用他。如果 period 为零 (0) 或 infinite，并且 duetime 不为 infinite，则 callback 调用他的方法一次。计时器的定期行为被禁用，但使用 change 方法能重新启用他。

在创建计时器之后若想改动他的 period 和 duetime，我们能通过调用 timer 的 change 方法来改动：

```
public bool change(
              int duetime,
              int period
         );//显然所改动的两个参数对应于 timer 中的两参数
```

见下例：

```
public static int main(string[] args)
{
```

```
    console.writeline("period is 1000");
    timer tm = new timer(new timercallback(timercall), 3, 1000, 1000);
    thread.sleep(2000);
    console.writeline("period is 500");
    tm.change(0, 800);
    thread.sleep(3000);
    return 0;
}
public static void timercall(object b)
{
    console.writeline("timercallback; b is {0}", b);
}
```

其运行结果为：

```
period is 1000
timercallback;b is 3
timercallback;b is 3
period is 500
timercallback;b is 3
timercallback;b is 3
timercallback;b is 3
timercallback;b is 3
```

总结

从以上的简单介绍，我们能看出他们各自使用的场合：thread 适用于那些需对线程进行复杂控制的场合；threadpool 适应于一些需要多个线程而又较短任务（如一些常处于阻塞状态的线程）；timer 则适用于那些需周期性调用的方法。只要我们了解了他们的使用特点，我们就能非常好的选择合适的方法。

5、谈谈多线程的思维方式

5.1、版权声明

文章出处：http://www.cnblogs.com/kevin-moon/archive/2009/04/24/1442469.html
文章作者：Kevin-moon

5.2、内容详情

前段时间仔细看过些关于多线程方面的资料，项目中用到线程的地方也不少，可是，当看了 Jeffrey 的一篇关于锁的文章后，发现自己虽然一直都在使用多线程，但是缺少了做多线程编程需要的思维！所以想从 Jeffrey 的 Optex（锁）入手，来谈谈我从其中体会的东西。

在 NET 中，我们用的最多的锁机制就是 lock，用起来很简单，短短几行程序就可以实现，例如：

```
public class TestThreading
{
    private System.Object lockThis = new System.Object();
    public void Function()
    {
        lock (lockThis)
        {
            // Access thread-sensitive resources.
        }
    }
}
```

其实我们也明白，lock 并不是锁，而是 MS 提供的一个简便式的写法，真正实现的是 Monitor 类中的 Enter 和 Exit 方法，既然提到了 Monitor 类也就说下有个需要注意的地方：

Pulse 和 PulseAll 方法，这两个方法就是把锁状态将要改变的消息通知给等待队列中的线程，不过这时如果等待队列中没有线程，那么该方法就会一直等待下去，直到有等待的线程进入队列，也就是说该方法可能造成类试死锁的情况出现。

上面的 lock + 线程(Thread 和 ThreadPool) = 多线程编程(N%)！？
对于该公式我曾经的 N 是 80，现在是 20。其中有很多东西影响我，让我从 80->20，下面的 Optex 就是一个入口点。

```
public sealed class Optex : IDisposable
```

```
{
    private Int32 m_Waiters = 0;
    private Semaphore m_WaiterLock = new Semaphore(0, Int32.MaxValue);

    public Optex() { }
    public void Dispose()
    {
        if (m_WaiterLock != null)
        {
            m_WaiterLock.Close();
            m_WaiterLock = null;
        }
    }

    public void Enter()
    {
        Thread.BeginCriticalRegion();
        // Add ourself to the set of threads interested in the Optex
        if (Interlocked.Increment(ref m_Waiters) == 1)
        {
            // If we were the first thread to show interest, we got it.
            return;
        }

        // Another thread has the Optex, we need to wait for it
        m_WaiterLock.WaitOne();
        // When WaitOne returns, this thread now has the Optex
    }

    public void Exit()
    {
        // Subtract ourself from the set of threads interested in the Optex
        if (Interlocked.Decrement(ref m_Waiters) > 0)
        {
            // Other threads are waiting, wake 1 of them
            m_WaiterLock.Release(1);
        }
        Thread.EndCriticalRegion();
    }
}
```

看完上面的代码，让我增加了两点认识：

1）Thread.BeginCriticalRegion()和 Thread.EndCriticalRegion();

因为这段时间正好看了一本多线程编程的书，既然将上面方法认为是进入临界区和退出临界区，对于临界区而言，进入该区的数据，在没有退出之前，如果临界区外的程序需要使用它，那么就必须出于等待。所以觉得已经使用临界区，为什么还要使用 Semaphore？！

可是，MS 只是取了个相同的名字，做的事情完全不同，上面两个方法完全没有临界区的概念，它只是设置一个区域(Begin 到 End 之间)，表示该区域内发生线程中断或未处理的异常会影响整个应用程序域。

2）m_Waiters 的作用

一开始以为在 Enter 的时候，直接写上：

m_WaiterLock.WaitOne();

it 的时候，写上：

m_WaiterLock.Release(1);

这样就可以了。m_Waiters 有什么意义？！

优化性能，Semaphore 是内核对象，我们都知道，要尽量少的进入内核模式，因为这是很消耗性能，所以尽量少的使用内核对象。m_Waiters 的意义就在这里，如果只有一个线程使用该锁对象的时候，是不需要去获取和释放的。 OK，上述的东西都是铺垫，铺完了也就进入主题了！

多线程的思维

```
namespace ThreadConcurrent.Lock
{
    public sealed class Optex : IDisposable
    {
        /// <summary>
        /// 琐的状态
        /// </summary>
        private Int32 m_LockState = c_lsFree;

        /// <summary>
        /// 自由状态
        /// </summary>
        private const Int32 c_lsFree = 0x00000000;

        /// <summary>
        /// 被拥有状态
        /// </summary>
        private const Int32 c_lsOwned = 0x00000001;

        /// <summary>
        /// 等待的线程数
        /// </summary>
```

```
private const Int32 c_lWaiter = 0x00000002;

private Semaphore m_WaiterLock = new Semaphore(0, Int32.MaxValue);

#region 构造函数

/// <summary>
///
/// </summary>
public Optex() { }

#endregion

/// <summary>
/// 请求锁
/// </summary>
public void Enter()
{
    Thread.BeginCriticalRegion();
    while (true)
    {
        Int32 ls = InterlockedOr(ref m_LockState, c_lsOwned);

        //自由状态
        if ((ls & c_lsOwned) == c_lsFree) return;

        // 增加等待的线程数
        if (IfThen(ref m_LockState, ls, ls + c_lWaiter))
        {
            m_WaiterLock.WaitOne();
        }
    }
}

public void Exit()
{
    // 释放琐
    Int32 ls = InterlockedAnd(ref m_LockState, ~c_lsOwned);

    //无等待的线程
    if (ls == c_lsOwned)
    {
    }
    else
```

```
        {
            ls &= ~c_lsOwned;
            if (IfThen(ref m_LockState, ls & ~c_lsOwned, ls - c_lWaiter))
            {
                m_WaiterLock.Release(1);
            }
            else
            {
            }
        }
        Thread.EndCriticalRegion();
    }

    #region 原子化操作
    /// <summary>
    /// 与操作
    /// </summary>
    /// <param name="target"></param>
    /// <param name="with"></param>
    /// <returns></returns>
    private static Int32 InterlockedAnd(ref Int32 target, Int32 with)
    {
        Int32 i, j = target;
        do
        {
            i = j;
            j = Interlocked.CompareExchange(ref target, i & with, i);
        } while (i != j);
        return j;
    }

    /// <summary>
    /// 或操作
    /// </summary>
    /// <param name="target"></param>
    /// <param name="with"></param>
    /// <returns></returns>
    private static Int32 InterlockedOr(ref Int32 target, Int32 with)
    {
        Int32 i, j = target;
        do
        {
            i = j;
            j = Interlocked.CompareExchange(ref target, i | with, i);
```

```
            } while (i != j);
            return j;
        }

        #endregion
        private static Boolean IfThen(ref Int32 val, Int32 @if, Int32 then)
        {
            return (Interlocked.CompareExchange(ref val, @then, @if) == @if);
        }

        private static Boolean IfThen(ref Int32 val, Int32 @if, Int32 then, out Int32 prevVal)
        {
            prevVal = Interlocked.CompareExchange(ref val, @then, @if);
            return (prevVal == @if);
        }

        /// <summary>
        /// 释放资源
        /// </summary>
        public void Dispose()
        {
            if (m_WaiterLock != null)
            {
                m_WaiterLock.Close();
                m_WaiterLock = null;
            }
        }
    }
}
```

对于上面的这个代码，我晕眩了好一段时间，不过当我真正理解的时候，从晕眩中学到了做多线程编程应该具备的思维方式。

首先从简单的理解开始谈，

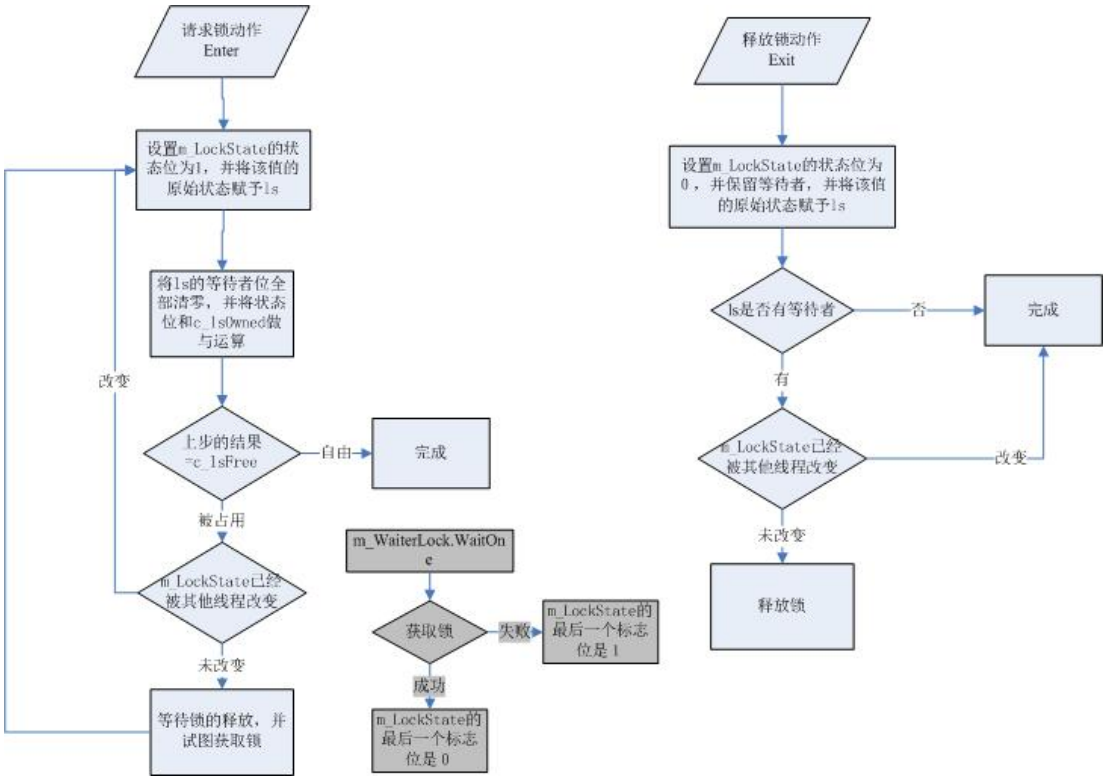
1）原子化操作

对于 InterLocked 类，曾经也知道，但是却用的很少，不过从该代码中知道，在多线程的编程中对共享数据的写入操作，一定要达到原子性。至于如何做到这点，InterlockedAnd 和 InterlockedOr 做了很好的诠释：

While 循环的目的就是保证 target 值以最新的值做与操作，如果传入的值在执行的过程被其他线程改变的话，那么是不会退出该循环的，并会利用改变后的值重新做次与操作。

2）理解 Enter 和 Exit

这两个方法很难写出来解释，用图是最清晰的。



曾经的晕眩：

- Enter 方法中为什么存在循环，为什么不是执行完 waitone 就结束，必须 m_lockState 等于 c_IsFree 的时候才结束？

线程的执行并不完全按照先前排好的顺序去执行，有时会发生一些特殊的情况来使改变线程的调度顺序，所以就可能会出现上图灰色部分的情况，则为了解决该可能发生的问题（概率很小）循环机制就出现了。

- 为什么在 WaitOne 和 Release 之前，除了增加和减少等待者外，还需要判断 m_lockstate 是否改变（进入 Enter 到执行 Waitone 前的这段时间）？

一般性的思维：

该程序的思维：

这样做的好处就是尽量少的操作内核对象，提高性能！

多线程编程虽然复杂，但是我觉得很有意思和挑战性，而且随着硬件的发展，多线程编程会更加重要，既然已经上路就让我们走到尽头！

6、断点续传、多线程上载

6.1、版权声明

文章出处：http://www.cnblogs.com/dlwang2002/archive/2008/09/12/1290017.html
文章作者：随心所欲

6.2、内容详情

现在已经有很多断点续传、多线程下载的软件了，比如网际快车等等。下面设计的程序是“断点续传、多线程上载”。

缘起：客户每天都有大量文件上传服务器。这些文件很多，并且体积挺大，FTP 有时候会出一些问题，导致传递失败，要重新上传。

基本解决方案：

1：把文件分割成块，每次只是传递一个文件块。
2：一个文件可以起多个发送任务（线程），同时发送。
3：记录文件发送状态，在网络出现问题时（或者客户端意外终止），知道上次发送文件大小和位置指针。再重新链接以后，继续发送。

对象和线程

这里面涉及到一个显示窗体 form1，有 timer 可以随时更新发送状态；一个上传类 Uploader（对应于一个文件）；Task 对象（也就是一个文件）；FileThunk 对象（每一个任务，对应于一个线程）；WebService 接受文件类。

发送状态需要记录在数据库。测试状态下，数据记录在 xml 文件。基本格式如下：

```
<tasks>
  <task name="contact.txt" percentage="" totalSeconds="" localFile="" remoteFile="" fileSize="">
   <thread name="thread1" begin="" end="" lastTime=""></thread>
  </task>
</tasks>
```

数据表结构也基本类似这样。

核心代码：

1：form1. 这个主要是显示。主要函数是 添加任务（Task）；更新任务状态

浏览文件，创建任务

```
private void btn_browse_Click(object sender, EventArgs e)
{
    //browse to select files
    OpenFileDialog ofd = new OpenFileDialog();
    ofd.Multiselect = true;
    DialogResult dr= ofd.ShowDialog();

   _taskList = new ArrayList();
   for (int i = 0; i < ofd.FileNames.Length; i++)
    {
        Task t = new Task();
        t.FileChunkCount = Convert.ToInt32(this.txt_threadsPerFile.Text);
        t.LocalFile = ofd.FileNames[i];
        t.RemoteFile = Path.GetFileName(t.LocalFile);
         t.Name = Path.GetFileName(t.LocalFile);
         //t.Percentage=0;
       t.TotalSeconds=0;

         t.Init(); //init, split the file to upload

        _taskList.Add(t);

    }
   //show in UI
    this.dataGridView1.DataSource = GetUpdatedStatusAsDT(_taskList,0)
}
```

显示和更新状态，返回 DataTable

```
private DataTable GetUpdatedStatusAsDT(ArrayList list, int seconds)
{
    DataTable dt = new DataTable();
    dt.Columns.Add("Name");
    dt.Columns.Add("FileSize");
    dt.Columns.Add("Percentage");
    dt.Columns.Add("TotalSeconds");
    dt.Columns.Add("LocalFile");
    dt.Columns.Add("RemoteFile");

    for (int i = 0; i < list.Count; i++)
```

```
    {
        //update task status to db
        Task t = (list[i] as Task);
        t.TotalSeconds += seconds;//running time
        t.Save();//save the status to db, so next time can load the status to continue.

        DataRow dr = dt.NewRow();
        dr["Name"] = t.Name;
        dr["FileSize"] = t.FileSize;
        dr["Percentage"] = t.Percentage;
        dr["TotalSeconds"] = t.TotalSeconds;
        dr["LocalFile"] = t.LocalFile;
        dr["RemoteFile"] = t.RemoteFile;

        dt.Rows.Add(dr);
    }
    return dt;
}
```

执行任务

```
private void btn_run_Click(object sender, EventArgs e)
{
    //timmer to show the status
    Timer timer = new Timer();
    timer.Tick += new EventHandler(timer_Tick);
    timer.Interval = 2 * 1000; //every 2 seconds
    timer.Enabled = true;
    timer.Start();
    //
    //continue
    //find task from datasource (DB or xml) and continue
    //ArrayList tasks = Task.LoadTasks();
    //
    ArrayList tasks = _taskList;  // get the task list.
    //
    for (int i = 0; i < tasks.Count; i++)
    {
        Uploader uld = new Uploader(tasks[i] as Task);
        uld.Strat();
    }
}
```

2：Uploader 的核心代码：主要是启动线程。

```
public void Strat()
{
    //_dataReceiver.Open(this._task.RemoteFile);//init webservice

    #region test
    //if (!File.Exists(_task.RemoteFile))
    //{
    //    FileInfo fi = new FileInfo(_task.LocalFile);

    //    FileStream fst = new FileStream(_task.RemoteFile, FileMode.CreateNew);

    //    BinaryWriter w = new BinaryWriter(fst);
    //    byte[] ab = new byte[fi.Length];
    //    w.Write(ab);
    //    w.Close();
    //    fst.Close();
    //}
    //
    #endregion

    //main code. find each fileChunk, running in seperate thread
    for (int i = 0; i < _task.FileChunks.Count; i++)
    {
        FileChunk fc = (_task.FileChunks[i] as FileChunk);
        Thread thread = new Thread(new ThreadStart(fc.Upload));
        fc.RunningThread = thread;//asign thread to the fileChucks, to stop the task
        thread.Name = _task.Name + "_" + i.ToString();
        thread.Start();
    }
}
```

3：Task 类的主要代码：主要是初始化任务，其他诸如和对数据的保存/提取

```
public void Init()
{
    //create new
    this.FileChunks = new ArrayList();

    FileInfo fi = new FileInfo(LocalFile);
    FileSize = fi.Length;

    long block = FileSize / this.FileChunkCount;
    long index = 0;
```

```
    for (int i = 0; i < this.FileChunkCount; i++)
    {
        FileChunk fc = new FileChunk(this);
        fc.Begin = index;
        if (i == this.FileChunkCount - 1)
            block = FileSize - index;
        index += block;
        fc.End = index;
        fc.LastTime = DateTime.Now;

        this.FileChunks.Add(fc);
    }
    //
}
```

4：FileChunk 的核心代码：主要是如何上传文件

```
/// <summary>
/// main function
/// </summary>
public void Upload()
{
    Stream fs = File.OpenRead(_task.LocalFile);

    long len = 2 * 64 * 1024;// 2 * 64 * 1024;

    while (Begin < End)
    {
        if (End - Begin < len)
            len = End - Begin;
        byte[] data = new byte[len];
        fs.Position = Begin;
        fs.Read(data, 0, (int)len);

        #region local test
        //Stream fs2 = File.OpenWrite(_task.RemoteFile);
        //fs2.Position = Begin;
        //fs2.Write(data, 0, (int)len);
        //fs2.Flush();
        //fs2.Close();
        //
        #endregion

        //call webservice to receive the data. this can be a socket also.
```

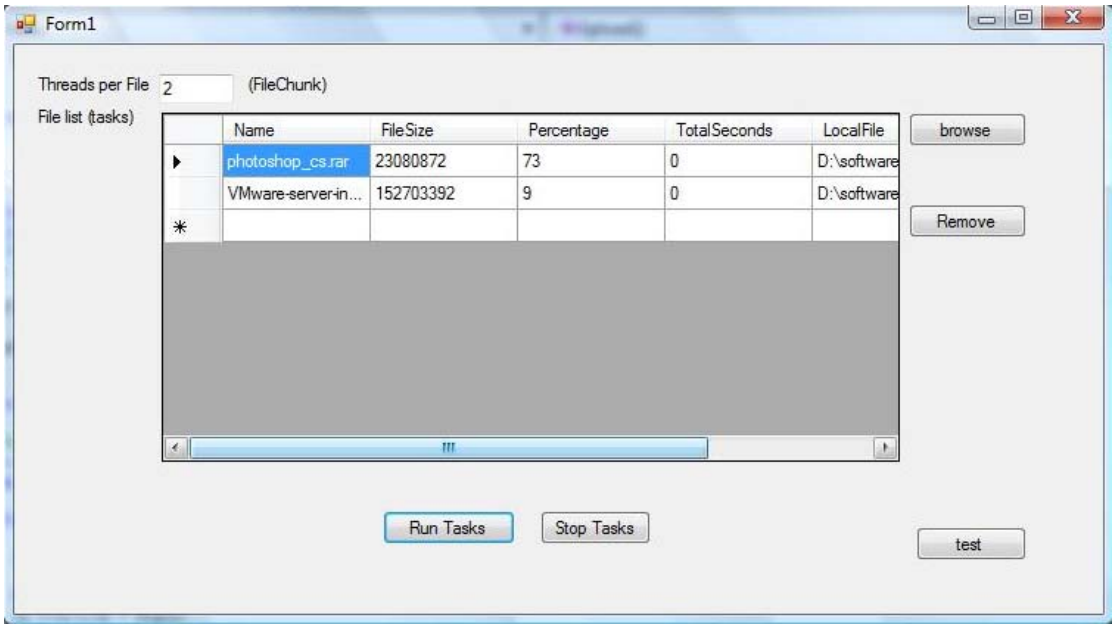
```
        string res = this._task.DataReceiver.Receive(_task.RemoteFile, Begin, len, data);
        if (res == "ok")
        {
            //
            Begin += len;
            _task.Save();
        }
        else
        {
            //wait for next while,
        }

    }
}
```

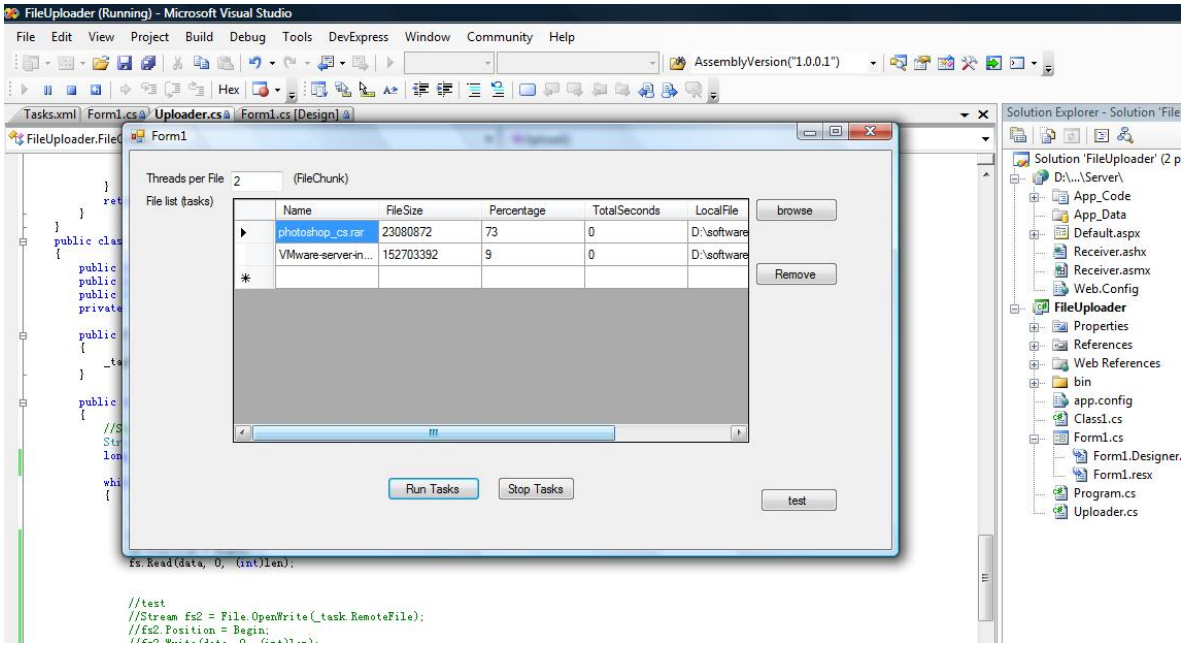
5：WebService 的核心代码

```
[WebMethod]
public string Receive(string file, long begin, long len, byte[] data)
{
    _fileName = AppDomain.CurrentDomain.BaseDirectory + "\\tmp\\" + file;
    try
    {
        _fs = File.OpenWrite(_fileName);
        _fs.Position = begin;
        _fs.Write(data, 0, (int)len);
        _fs.Flush();
        _fs.Close();
        _fs = null;
        return "ok";
    }
    catch (Exception ex)
    {
        string s = ex.Message;
        return "retry";
    }
}
```

小结：

技术难度不大；现实实用；上传速度尚可。


Form1
Threads per File
2
(FileChunk)
File list (tasks)
Name
FileSize
Percentage
TotalSeconds
LocalFile
photoshop_cs.rar
23080872
73
0
D:\software
VMware-server-in...
152703392
9
0
D:\software
browse
Remove
Run Tasks
Stop Tasks
test


FileUploader (Running) - Microsoft Visual Studio
File Edit View Project Build Debug Tools DevExpress Window Community Help
AssemblyVersion("1.0.0.1")
Tasks.xml
Form1.cs
Uploader.cs
Form1.cs [Design]
Solution Explorer - Solution 'File
Solution 'FileUploader' (2 p
D:\...\Server\
App_Code
App_Data
Default.aspx
Receiver.ashx
Receiver.asmx
Web.Config
FileUploader
Properties
References
Web References
bin
app.config
Class1.cs
Form1.cs
Form1.Designer.
Form1.resx
Program.cs
Uploader.cs
Form1
Threads per File
2
(FileChunk)
File list (tasks)
Name
FileSize
Percentage
TotalSeconds
LocalFile
photoshop_cs.rar
23080872
73
0
D:\software
VMware-server-in...
152703392
9
0
D:\software
browse
Remove
Run Tasks
Stop Tasks
test
fs.Read(data, 0, (int)len);
//test
//Stream fs2 = File.OpenWrite(_task.RemoteFile);
//fs2.Position = Begin;

7、C#多线程学习

7.1、版权声明

文章出处：http://www.cnblogs.com/rentiansheng/archive/2011/02.html
文章作者：任忌

7.2、内容详情

7.2.1 多线程的相关概念

什么是进程？

当一个程序开始运行时，它就是一个进程，进程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。

什么是线程？

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。

什么是多线程？

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。

多线程的好处：

可以提高 CPU 的利用率。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。

多线程的不利方面：

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；

接下来将对 C#编程中的多线程机制进行探讨。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，接下来的所有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。

线程使用收集

命名线程 Thread.CurrentThread.Name = "System Thread";
获得当前线程的名称 Thread.CurrentThread.Name
获得当前线程的状态 Thread.CurrentThread.ThreadState
线程操作的关键对象和语句
Lock Monitor ThreadPool ManualResetEvent Mutex AutoResetEvent InterLocked

任何程序在执行时，至少有一个主线程。
一个直观印象的线程示例：

```
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();
        }
    }
}
```

输出如下：

```
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。

在程序的头部，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。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。

我们通过其中提供的 Thread 类来创建和控制线程，ThreadPool 类用于管理线程池等。(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。)

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。

7.2.2 如何操纵一个线程

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），编写完全控制一个线程的代码示例：

```
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));//不用带括号
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。

主线程 Main()函数

所有线程都是依附于 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。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。

Thread.ThreadState 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，我们有时候可以通过对该值的判断来设计程序流程。

ThreadState 属性的取值如下：

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。

线程的优先级

当线程之间争夺 CPU 时间时，CPU 是按照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。

7.2.3 生产者和消费者

While 争夺发生，lock 语句

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。这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。

- 如果你想保护一个类的实例，一般地，你可以使用 this；
- 如果你想保护一个静态变量（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，在注释里说明了 lock 关键字的用法和用途。

示例如下：

```
using System;
using System.Threading;

namespace ThreadSimple
{
    internal class Account
    {
        int balance;
        Random r = new Random();
```

```
    internal Account(int initial)
    {
        balance = initial;
    }

    internal int Withdraw(int amount)
    {
        if (balance < 0)
        {
            //如果 balance 小于则抛出异常
            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 的值是不可能小于的
        lock (this)
        {
            Console.WriteLine("Current Thread:" + Thread.CurrentThread.Name);
            //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
            //另外一个线程却执行了 balance=balance-amount 修改了 balance 的值
            //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
            //但是，这个线程却继续执行 balance=balance-amount，所以导致 balance 可能小于
            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();
        }
    }
}
```

lock 与 Monitor 区别

lock 锁定一段代码，Monitor 锁定一个对象。

Monitor 类锁定一个对象

当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。(也就是说,在

一个线程运行的代码里面调用 Monitor.Enter(对象),那么其它线程就不能使用该对象了)

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。

获取的过程

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（注释中介绍了该程序的精要所在）。

首先，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。

以下例子并没有使用到 Monitor.Enter()和 Monitor.Exit()

而是使用了 Monitor.Wait()和 Monitor.Pulse()

示例如下：

```
using System;
using System.Threading;

//这里并没有使用 Monitor.Enter()和 Monitor.Exit()方法
//而是使用了 Monitor.Wait()和 Monitor.Pulse()方法
namespace threadingtest
{
    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
            {
                //如果现在不可读取
                //普通的做法是使用 else 或 return false
                //而这里是使用了一个 wait 方法,让程序暂停等待
                if (!readerFlag)
                {
                    try
                    {
                        //等待 WriteToCell 方法中调用 Monitor.Pulse()方法
                        Console.WriteLine("waiting...");
                        Monitor.Wait(this);
                    }
                    catch (SynchronizationLockException e)
                    {
                        Console.WriteLine(e);
                    }
                    catch (ThreadInterruptedException e)
                    {
                        Console.WriteLine(e);
                    }
                }
                Console.WriteLine("Now consume, the  consumable is:  {0}", cellContents);
                Console.WriteLine("Now repeating...");
                readerFlag = false;
                //重置 readerFlag 标志，表示消费行为已经完成
                Console.WriteLine("Now pulseing...");
                //通知 WriteToCell()方法（该方法在另外一个线程中执行，等待中）
                Monitor.Pulse(this);
            }
            return cellContents;
        }

        public void WriteToCell(int n)
        {
            lock (this)
            {
                if (readerFlag)
                {
                    try
                    {
                        //等待 ReadFromCell 方法中调用 Monitor.Pulse()方法
```

```
                    Console.WriteLine("Now waiting...");
                    Monitor.Wait(this);
                }
                catch (SynchronizationLockException e)
                {
                    //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)
                {
                    //当线程在等待状态的时候中止
                    Console.WriteLine(e);
                }
            }
            Console.WriteLine("Now producing...");
            cellContents = n;
            Console.WriteLine("the Productiong is:  {0}", cellContents);
            Console.WriteLine("sure the production");
            readerFlag = true;
            Monitor.Pulse(this);
            //通知另外一个线程中正在等待的 ReadFromCell()方法
        }
    }
}
//下面定义生产者类 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

    public CellProd(Cell box, int request)
    {
        //构造函数
        cell = box;
        quantity = request;
    }
    public void ThreadRun()
    {
        for (int looper = 1; looper <= quantity; looper++)
        {
            //生产者向操作对象写入信息
            //执行生产操作
```

```
                //这里是关键
                //如果 cell.ReadFromCell()正在执行,那么下面的调用将暂停等待
                cell.WriteToCell(looper);
            }
        }
    }

    public class CellCons
    {
        Cell cell;
        int quantity = 1;
        public CellCons(Cell box, int request)
        {
            //构造函数
            cell = box;
            quantity = request;
        }
        public void ThreadRun()
        {
            int valReturned;
            for (int looper = 1; looper <= quantity; looper++)
            {
                //消费者从操作对象中读取信息
                //执行消费操作
                //这里是关键
                //如果 cell.WriteToCell()正在执行,那么下面的调用将暂停等待
                valReturned = cell.ReadFromCell();
            }
        }
    }
    //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 zMain(String[] args)
        {
            int result = 0; //一个标志位，如果是表示程序没有出错，如果是表明有错误发生
            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次
            CellProd prod = new CellProd(cell, 20);
            CellCons cons = new CellCons(cell, 20);

            //创建两个线程,两个线程分别执行不同的代码和操作.
```

```
            //创建并不等于马上运行
            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();

                //这里是关键
                producer.Join();
                consumer.Join();
                Console.ReadLine();
            }
            catch (ThreadStateException e)
            {
                //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)
            {
                //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;
            Console.WriteLine(result.ToString());
        }
    }
}


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
            {
                //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;
        //重置 readerFlag 标志，表示消费行为已经完成
        Monitor.Pulse(this);
        //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)
            {
                //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)
            {
                //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);
            //通知另外一个线程中正在等待的 ReadFromCell()方法
        }
    }
}
```

下面定义生产者类 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); //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)
    {
        //构造函数
        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
    }
}
```

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
```

这个例子的重点在于 lock 和 Monitor.Wait()和 Monitor.Pulse()的搭配使用,同时使用了一个 readflags 标识.

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。

7.2.4 多线程的自动管理(线程池)

在多线程的程序中，经常会出现两种情况：

一种情况： 应用程序中，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

这一般使用 ThreadPool（线程池）来解决；

另一种情况：线程平时都处于休眠状态，只是周期性地被唤醒

这一般使用 Timer（定时器）来解决；

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。

将线程安放在线程池里，需使用 ThreadPool.QueueUserWorkItem()方法，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);
```

ThreadPool 类是一个静态类，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，那么该项目将是无法取消的。

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。

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 对象将被设置为有信号，从而通知主线程继续运行。

ManualResetEvent 对象有几个重要的方法：

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。

ThreadPool 的用法示例：

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

```
namespace ThreadExample
{
    //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;
        }

        //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)
            {
                //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);
            //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
            {
                //将工作项装入线程池
                //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");
            //等待事件的完成，即线程调用ManualResetEvent.Set()方法
            eventX.WaitOne(Timeout.Infinite, true);
            //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;
    }
  }
}
```

程序中应该引起注意的地方：

SomeState 类是一个保存信息的数据结构，它在程序中作为参数被传递给每一个线程，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。

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。输

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，这样才能得心应手地使用它。

7.2.5 多线程的自动管理(定时器Timer)

Timer 类：设置一个定时器，定时执行用户指定的函数。

定时器启动后，系统将自动建立一个新的线程，执行用户指定的函数。

初始化一个 Timer 对象：

```
Timer timer = new Timer(timerDelegate, s,1000, 1000);
```

// 第一个参数：指定了 TimerCallback 委托，表示要执行的方法；
// 第二个参数：一个包含回调方法要使用的信息的对象，或者为空引用；
// 第三个参数：延迟时间——计时开始的时刻距现在的时间，单位是毫秒,指定为“0”表示立即启动计时器；
// 第四个参数：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。指定 Timeout.Infinite 可以禁用定期终止。

Timer.Change()方法：修改定时器的设置。（这是一个参数类型重载的方法）

使用示例： timer.Change(1000,2000);

Timer 类的程序示例(来源：MSDN)：

```
using System;
using System.Threading;

namespace ThreadExample
{
    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 s 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();
        }
```

```
        //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)
            {
                //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。

7.2.6 互斥对象Mutex

如何控制好多个线程相互之间的联系，不产生冲突和重复，这需要用到互斥对象，即：System.Threading 命名空间中的 Mutex 类。

我们可以把 Mutex 看作一个出租车，乘客看作线程。乘客首先等车，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，可以把它理解为一个信号灯。这里用它的有信号状态来表示一个线程的结束。

```
// AutoResetEvent.Set()方法设置它为有信号状态
// AutoResetEvent.Reset()方法设置它为无信号状态
```

Mutex 类的程序示例：

```
using System;
using System.Threading;

namespace ThreadExample
{
    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; //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(); //线程 t2,t3 结束条件满足

        Thread.Sleep(1000);
        Console.WriteLine(" - Main releases gM2");
        gM2.ReleaseMutex(); //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
        gM1.ReleaseMutex(); //修正上一次出现的错误
        gM2.ReleaseMutex(); //修正上一次出现的错误
        Thread.Sleep(2000);
        Console.WriteLine("t1Start finished, Mutex.WaitAll(Mutex[]) satisfied");
        Event1.Set(); //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
        gM1.ReleaseMutex(); //修正上一次出现的错误
        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
            gM1.ReleaseMutex(); //修正上一次出现的错误
            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
            gM2.ReleaseMutex(); //修正上一次出现的错误
            Console.WriteLine("t4Start finished, gM2.WaitOne( )");
            Event4.Set();//线程结束，将 Event4 设置为有信号状态
        }
    }
}
```

程序的输出结果：

结果

```
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
t2Start finished, gM1.WaitOne( ) satisfied
t3Start finished, Mutex.WaitAny(Mutex[])
 - Main releases gM2
t1Start finished, Mutex.WaitAll(Mutex[]) satisfied
t4Start finished, gM2.WaitOne( )
 Mutex Sample
```

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。

8、C#多线程编程

8.1、版权声明

文章出处：http://www.cnblogs.com/free722/category/291820.html
文章作者：辛勤的代码工

8.2、内容详情

8.2.1 lock使用注意事项

1.避免锁定 public 类型对象。

如果实例可以被公共访问，将出现 lock(this)问题。
如有一个类 MyClass，该类有一个 Method 方法通过 lock(this)来实现互斥：

```
public class MyClass
 {
        public void Method()
        {
            lock(this)
            {
                ......
                }
        }
 }
```

如果一个 MyClass 的实例在不同线程中执行 Method 方法，可以实现互斥。但如果多个 MyClass 的实例分别在不同的线程中执行 Method 方法，互斥将失效，因为此处的 lock(this)仅对当前的实例对象进行了加锁。

2.禁止锁定类型

lock(typeof(ClassName))锁定范围更为广泛，由于一个类的所有实例都只有一个类型对象（该对象是 typeof 的返回结果），锁定它，就锁定了该对象的所有实例。微软现在建议不要使用 lock(typeof(ClassName))。以下的微软的原文描述：

首先锁定类型对象是个很缓慢的过程，并且类中的其他线程、甚至在同一个应用程序域中运行的其他程序都可以访问该类型对象，因此，它们就有可能代替您锁定类型对象，完全

阻止您的执行，从而导致你自己的代码的挂起。

这里的基本问题是，您并未拥有该类型对象，并且您不知道还有谁可以访问它。总的来说，依靠锁定不是由您创建、并且您不知道还有谁可以访问的对象是一种很不好的做法。这样做很容易导致死锁。

3.禁止锁定字符串

锁住一个字符串更为神奇，只要字符串内容相同，就能引起程序挂起。

在.NET 中，字符串会被暂时存放，如果两个变量的字符串内容相同的话，.NET 会把暂存的字符串对象分配给该变量。所以如果有两个地方都在使用 lock("my lock")的话，它们实际锁住的是同一个对象。

如何正确使用 lock 呢？

微软给出的建议是：只锁定私有对象。

示例代码：

```
public class MyClass
{
    private static Object somePrivateStaticObject = new Object();
    // methods of class go here--can lock somePrivateStaticObject
    public void Method()
    {
        lock(somePrivateStaticObject)
        {
            ......
            }
    }
}
```

锁定私有对象的好处：

首先，类以外的任何代码都无法锁定 MyClass.somePrivateStaticObject，因此避免了许多死锁的可能。由于死锁属于那种最难找到根源的问题，因此，避免发生死锁的可能是一件很好的事情。

其次，应用程序中只有一份 MyClass.somePrivateStaticObject 的副本，并且系统上运行的其他每个应用程序也只有一个副本。因此，在同一个应用程序域中的应用程序之间没有相互影响。

8.2.2 集合类中Synchronized与SyncRoot属性原理分析

Synchronized vs SyncRoot 我们知道，在.net 的一些集合类型中，譬如 Hashtable 和 ArrayList，都有 Synchronized 静态方法和 SyncRoot 实例方法，他们之间有联系吗？我怎么才能用好他们呢？

我们以 Hashtable 为例，看看他们的基本用法：

```
Hashtable ht = Hashtable.Synchronized(new Hashtable());
lock (ht.SyncRoot)
{
    //一些操作
}
```

Synchronized 表示返回一个线程安全的 Hashtable，什么样的 hashtable 才是一个线程安全的呢？下边我们就从.NET 的源码开始理解。

```
public static Hashtable Synchronized(Hashtable table)
{
    if (table == null)
    {
        throw new ArgumentNullException("table");
    }
    return new SyncHashtable(table);
}
```

从源码不难看出，Synchronized 方法返回的其实是一个 SynchHashtable 类型的实例。在前边我们说过，Synchronized 表示返回一个线程安全的 Hashtable，从这个解释不难看出，SynchHashtable 应该是继承自 Hashtable。下边我们验证一下。看看 SynchHashtable 类型的源码：

```
[Serializable]
private class SyncHashtable : Hashtable
{
    // Fields
    protected Hashtable _table;

    // Methods
    internal SyncHashtable(Hashtable table)
        : base(false)
    {
        this._table = table;
    }
```

```
    internal SyncHashtable(SerializationInfo info, StreamingContext context)
        : base(info, context)
    {
        this._table = (Hashtable)info.GetValue("ParentTable", typeof(Hashtable));
        if (this._table == null)
        {
            throw                                                                    new
SerializationException(Environment.GetResourceString("Serialization_InsufficientState"));
        }
    }

    public override void Add(object key, object value)
    {
        lock (this._table.SyncRoot)
        {
            this._table.Add(key, value);
        }
    }

    public override void Clear()
    {
        lock (this._table.SyncRoot)
        {
            this._table.Clear();
        }
    }

    public override object Clone()
    {
        lock (this._table.SyncRoot)
        {
            return Hashtable.Synchronized((Hashtable)this._table.Clone());
        }
    }

    public override bool Contains(object key)
    {
        return this._table.Contains(key);
    }

    public override bool ContainsKey(object key)
    {
        return this._table.ContainsKey(key);
```

```
    }

    public override bool ContainsValue(object key)
    {
        lock (this._table.SyncRoot)
        {
            return this._table.ContainsValue(key);
        }
    }

    public override void CopyTo(Array array, int arrayIndex)
    {
        lock (this._table.SyncRoot)
        {
            this._table.CopyTo(array, arrayIndex);
        }
    }

    public override IDictionaryEnumerator GetEnumerator()
    {
        return this._table.GetEnumerator();
    }

    public override void GetObjectData(SerializationInfo info, StreamingContext context)
    {
        if (info == null)
        {
            throw new ArgumentNullException("info");
        }
        info.AddValue("ParentTable", this._table, typeof(Hashtable));
    }

    public override void OnDeserialization(object sender)
    {
    }

    public override void Remove(object key)
    {
        lock (this._table.SyncRoot)
        {
            this._table.Remove(key);
        }
    }
```

```
internal override KeyValuePairs[] ToKeyValuePairsArray()
{
    return this._table.ToKeyValuePairsArray();
}

// Properties
public override int Count
{
    get
    {
        return this._table.Count;
    }
}

public override bool IsFixedSize
{
    get
    {
        return this._table.IsFixedSize;
    }
}

public override bool IsReadOnly
{
    get
    {
        return this._table.IsReadOnly;
    }
}

public override bool IsSynchronized
{
    get
    {
        return true;
    }
}

public override object this[object key]
{
    get
    {
        return this._table[key];
    }
```

```
        set
        {
            lock (this._table.SyncRoot)
            {
                this._table[key] = value;
            }
        }
    }

    public override ICollection Keys
    {
        get
        {
            lock (this._table.SyncRoot)
            {
                return this._table.Keys;
            }
        }
    }

    public override object SyncRoot
    {
        get
        {
            return this._table.SyncRoot;
        }
    }

    public override ICollection Values
    {
        get
        {
            lock (this._table.SyncRoot)
            {
                return this._table.Values;
            }
        }
    }
}
```

呵呵，果然不出我们所料，SyncHashtable 果然继承自 Hashtable，SyncHashtable 之所有能实现线程的安全操作，就是因为在他们的一些方法中，就加了 lock，我们知道，哪一个线程执行了 lock 操作，在他还没有释放 lock 之前，其他线程都要处于堵塞状态。SyncHashtable 就是通过这种方法，来实现所谓的线程安全。

现在我们理解了 Synchronized 的含义和用法，那接下来我们看看他和 SyncRoot 之间的关系。SyncRoot 表示获取可用于同步 Hashtable 访问的对象，老实说，这个解释不好理解，要想真正理解他的用法，我们还得从源码开始：

```
public virtual object SyncRoot
{
    get
    {
        if (this._syncRoot == null)
        {
            Interlocked.CompareExchange(ref this._syncRoot, new object(), null);
        }
        return this._syncRoot;
    }
}
```

如果您清楚 Interlocked 的用法，这段代码没什么难理解的了（不清楚的朋友找 GOOGLE 吧），Interlocked 为多个线程共享的变量提供原子操作。原子操作就是单线程操作。在一个 Hashtable 实例中，不论我们在代码的任何位置调用，返回的都是同一个 object 类型的对象。我们在开始写的 lock(ht.SyncRoot)和下边的操作作用是一样的。

```
static object obj = new object();
lock(obj)
{
// 一些操作
}
```

他们之间不同的是，我们声明的 static object 类型对象是类型级别的，而 SyncRoot 是对象级别的。

通过上面的分析，我们都应该能理解 Synchronized 和 SyncRoot 用法，他们之间的关系就是：

Hashtable 通过 Synchronized 方法，生成一个 SynchHashtable 类型的对象，在这个对象的一个方法中，通过 lock (this._table.SyncRoot)这样的代码来实现线程安全的操作，其中 this._table.SyncRoot 返回的就是一个 object 类型的对象，在一个 SynchHashtable 对象实例中，不管我们调用多少次，他是唯一的。

另外，针对泛型集合的线程安全访问，由于泛型集合中没有直接公布 SyncRoot 属性，所以猛一看好似无从下手。

但是查看集合泛型集合的源代码后就可发现他们实际上都提供了 SyncRoot 属性。

以下以 Queue<T>集合为例。

```
bool ICollection.IsSynchronized
{
    get
    {
        return false;
    }
}

object ICollection.SyncRoot
{
    get
    {
        if (this._syncRoot == null)
        {
            Interlocked.CompareExchange(ref this._syncRoot, new object(), null);
        }
        return this._syncRoot;
    }
}
```

从以上源代码可以看出，这两个方法都被实现为了 private，不过使用 ICollection 接口还是可以访问的。

```
lock (((ICollection)_queue).SyncRoot)
{
    int item = _queue.Dequeue();
}
```

8.2.3 Monitor使用示例及Mutex简介

Monitor 类功效和 lock 类似：

```
System.Object obj = (System.Object)x;
System.Threading.Monitor.Enter(obj);
try
{
    DoSomething();
}
finally
{
    System.Threading.Monitor.Exit(obj);
}
```

lock 关键字比 Monitor 简洁，其实 lock 就是对 Monitor 的 Enter 和 Exit 的一个封装。

另外，Monitor 还有几个常用的方法：

(1) TryEnter

TryEnter 能够有效的决绝长期死等的问题，如果在一个并发经常发生，而且持续时间长的环境中使用 TryEnter，可以有效防止死锁或者长时间的等待。

比如我们可以设置一个等待时间：

bool gotLock = Monitor.TryEnter（myobject,1000）

让当前线程在等待 1000 秒后根据返回的 bool 值来决定是否继续下面的操作。

(2) Pulse、PulseAll、Wait

Pulse 以及 PulseAll 还有 Wait 方法是成对使用的，它们能让你更精确的控制线程之间的并发。

示例代码：

```
using System.Threading;

public class Program  {
    //同步对象
    static object ball = new object();

    public static void Main()
    {
        //创建并启动线程
        Thread threadPing = new Thread(ThreadPingProc);
        Thread threadPong = new Thread(ThreadPongProc);

        threadPing.Start();
        threadPong.Start();

        System.Console.ReadLine();
    }

    static void ThreadPingProc()
    {
        System.Console.WriteLine("Thread Ping Start!");
```

```
        //锁定 ball
        lock (ball)
        {
            for (int i = 0; i < 5; i++)
            {
                System.Console.WriteLine("ThreadPing: Ping ");
                //通知队列中锁定对象 ball 的状态更改
                Monitor.Pulse(ball);
                //释放 ball 对象上的锁，并阻止该线程，直到它重新获得 ball 对象锁
                Monitor.Wait(ball);
            }

            //通知队列中锁定对象 ball 的状态更改
            Monitor.Pulse(ball);
        }

        System.Console.WriteLine("ThreadPing: Bye!");
    }



    static void ThreadPongProc()
    {
        System.Console.WriteLine("Thread Pong Start!");
        //锁定 ball
        lock (ball)
        {
            for (int i = 0; i < 5; i++)
            {
                System.Console.WriteLine("ThreadPong: Pong ");
                //通知队列中锁定对象 ball 的状态更改
                Monitor.Pulse(ball);
                //释放 ball 对象上的锁，并阻止该线程，直到它重新获得 ball 对象锁
                Monitor.Wait(ball);
            }
            //通知队列中锁定对象 ball 的状态更改
            Monitor.Pulse(ball);
        }
        System.Console.WriteLine("ThreadPong: Bye!");
    }
}
```

执行结果如下：

```
ThreadPing: Hello!
```

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

当 threadPing 进程进入 ThreadPingProc 锁定 ball 并调用 Monitor.Pulse(ball)后，它通知 threadPong 从阻塞队列进入准备队列。当 threadPing 调用 Monitor.Wait(ball)阻塞自己后，它放弃了了对 ball 的锁定，所以 threadPong 得以执行。PulseAll 与 Pulse 方法类似，不过它是向所有在阻塞队列中的进程发送通知信号，如果只有一个线程被阻塞，那么请使用 Pulse 方法。

Mutex 对象

Mutex 与 Monitor 类似，这里不再累赘。需要注意的是 Mutex 分两种：一种是本地 Mutex，一种是系统级 Mutex。系统级 Mutex 可以用来进行跨进程间的线程的同步。尽管 Mutex 可以用于进程内的线程同步，但是使用 Monitor 通常更为可取，因为 Monitor 是专门为 .NET Framework 而设计的，因而它可以更好地利用资源。相比之下，Mutex 类是 Win32 构造的包装。尽管 Mutex 比 Monitor 更为强大，但是相对于 Monitor 类，它所需要的互操作转换更消耗计算资源。

8.2.4 同步事件和等待句柄

同步事件和等待句柄用于解决复杂的同步情况。比如一个大的计算步骤包含 3 个步骤 result = first term + second term + third term，如果现在想写个多线程程序，同时计算 first term,second term 和 third term，等所有 3 个步骤计算好后再把它们汇总起来，我们就需要使用到同步事件和等待句柄。

同步事件分有两个，分别为 AutoResetEvent 和 ManualResetEvent，这两个类可以用来代表某个线程的运行状态：终止和非终止。

等待句柄用来判断 ResetEvent 的状态，如果是非终止状态就一直等待，否则放行，让等待句柄下面的代码继续运行。

下面的代码示例阐释了如何使用等待句柄来发送复杂数字计算的不同阶段的完成信号。此计算的格式为：result = first term + second term + third term

```
using System;
using System.Threading;

class CalculateTest
{
    static void Main()
    {
        Calculate calc = new Calculate();

        Console.WriteLine("Result = {0}.", calc.Result(234).ToString());
        Console.WriteLine("Result = {0}.", calc.Result(55).ToString());

        Console.ReadKey();
    }
}

class Calculate
{
    //基数、数 1、数 2、数 3
    double baseNumber, firstTerm, secondTerm, thirdTerm;
    //自动重置同步事件数组
    AutoResetEvent[] autoEvents;
    //手动重置同步事件
    ManualResetEvent manualEvent;
    //随机数生成器
    Random randomGenerator;

    public Calculate()
    {
        //初始化同步事件
        autoEvents = new AutoResetEvent[]
                     {
                         new AutoResetEvent(false),
                         new AutoResetEvent(false),
                         new AutoResetEvent(false)
                     };

        manualEvent = new ManualResetEvent(false);
    }

    void CalculateBase(object stateInfo)
```

```
    {
        baseNumber = randomGenerator.NextDouble();

        //置事件状态为终止状态，使等待线程继续
        manualEvent.Set();
    }

    void CalculateFirstTerm(object stateInfo)
    {
        double preCalc = randomGenerator.NextDouble();
        //等待手动重置事件终止
        manualEvent.WaitOne();

        firstTerm = preCalc * baseNumber * randomGenerator.NextDouble();
        //置自动事件终止信号
        autoEvents[0].Set();
    }
    void CalculateSecondTerm(object stateInfo)
    {
        double preCalc = randomGenerator.NextDouble();
        //等待手动重置事件终止
        manualEvent.WaitOne();
        secondTerm = preCalc * baseNumber * randomGenerator.NextDouble();
        //置自动事件终止信号
        autoEvents[1].Set();
    }

    void CalculateThirdTerm(object stateInfo)
    {
        double preCalc = randomGenerator.NextDouble();
        //等待手动重置事件终止
        manualEvent.WaitOne();
        thirdTerm = preCalc * baseNumber * randomGenerator.NextDouble();
        //置自动事件终止信号
        autoEvents[2].Set();
    }

    public double Result(int seed)
    {
        randomGenerator = new Random(seed);

        //将待执行工作加入线程
        ThreadPool.QueueUserWorkItem(new WaitCallback(CalculateBase));
        ThreadPool.QueueUserWorkItem(new WaitCallback(CalculateFirstTerm));
```

```
        ThreadPool.QueueUserWorkItem(new WaitCallback(CalculateSecondTerm));
        ThreadPool.QueueUserWorkItem(new WaitCallback(CalculateThirdTerm));

        //等待所有自动事件终止
        WaitHandle.WaitAll(autoEvents);
        //重置手动事件为非终止状态
        manualEvent.Reset();

        return firstTerm + secondTerm + thirdTerm;
    }
}
```

该示例一共有 4 个 ResetEvent，一个 ManualEvent，三个 AutoResetEvent，分别反映 4 个线程的运行状态。

ManualEvent 和 AutoResetEvent 有一点不同：

AutoResetEvent 是在当前线程调用 set 方法激活某线程之后，AutoResetEvent 状态自动重置；ManualEvent 则需要手动调用 Reset 方法来重置状态。

接着来看看上面那段代码的执行顺序，Main 方法首先调用的是 Result 方法，Result 方法开启 4 个线程分别去执行，主线程阻塞在 WaitHandle.WaitAll(autoEvents)处，等待 3 个计算步骤的完成。4 个 ResetEvent 初始化状态都是非终止(构造实例时传入了 false)，CalculateBase 首先执行完毕，其他 3 个线程阻塞在 manualEvent.WaitOne()处，等待 CalculateBase 执行完成。CalculateBase 生成 baseNumber 后，把代表自己的 ManualEvent 状态设置为终止状态。其他几个线程从 manualEvent.WaitOne()处恢复执行，在执行完自己的代码后把自己对应的 AutoResetEvent 状态置为终止。当 3 个计算步骤执行完后，主线程从阻塞中恢复，把三个计算结果累加后返回。

还要多补充一点的是 WaitHandle 的 WaitAll、WaitAny 方法。如果等待多个事件终止就用 WaitAll，如本例中的：WaitHandle.WaitAll(autoEvents)。WaitAny 是等待的事件数组中有一个为终止状态则停止等待。

8.2.5 Timer（定时器）使用示例

Timer 类：设置一个定时器，定时执行用户指定的函数。定时器启动后，系统将自动建立一个新的线程，执行用户指定的函数。

构造函数：Timer(TimerCallback callback, object state, int dueTime, int period)
参数说明

callback：一个 System.Threading.TimerCallback 委托，表示要执行的方法。

state：一个包含回调方法要使用的信息的对象，或者为 null。

dueTime：调用 callback 之前延迟的时间量（以毫秒为单位）。指定 System.Threading.Timeout.Infinite 可防止启动计时器。指定零(0) 可立即启动计时器。

period：调用 callback 的时间间隔（以毫秒为单位）。指定 System.Threading.Timeout.Infinite 可以禁用定期终止。

Timer.Change()方法：修改定时器的设置。（这是一个参数类型重载的方法）

示例代码：

```
using System;
using System.Threading;

namespace ThreadExample{
    //Timer 信息对象 7
    class TimerObject
    {
        public int Counter = 0;
    }
    class App
    {
        public static void Main()
        {
            TimerObject s = new TimerObject();
            //创建委托对象 TimerCallback，该委托将被定时调用
            TimerCallback timerDelegate = new TimerCallback(CheckStatus);
            //创建一个时间延时 2s 启动，间隔为 1s 的定时器
            Timer timer = new Timer(timerDelegate, s, 2000, 1000);
            //检查计数器值
            while (s.Counter < 10)
            {
                Thread.Sleep(1);
            }
            //更改 timer 执行周期，延时 1s 开始
            timer.Change(1000, 500);
            Console.WriteLine("Timer 被加快了！");
            //检查计数器值
            while (s.Counter < 20)
            {
                Thread.Sleep(1);
            }
            //释放 Timer
            timer.Dispose();
            Console.WriteLine("Timer 示例运行结束！");
            Console.ReadLine();
```

```
        }
        //下面是被定时调用的方法
        static void CheckStatus(Object state)
        {
            TimerObject s =(TimerObject)state;
            s.Counter++;
            Console.WriteLine("当前时间：{0} 计数器值：{1}", DateTime.Now.ToString(), s.Counter);
        }
    }
}
```

程序首先创建一个定时器，延时 2 秒启动，每隔 1 秒调用一次 CheckStatus()方法。当调用 9 次以后，修改时间间隔为 500 毫秒，且指定在 1 秒后重新开始。当计数达到 19 次，调用 Dispose()方法删除了 timer 对象，终止程序。

8.2.6 volatile关键字的原理探讨

volatile 关键字

volatile 关键字仅应用于类或结构字段，用于通知编译器，将有多个线程访问该字段，因此它不应当对此成员的状态做任何优化，这样可以确保该字段在任何时间呈现的都是最新的值。

不是所有的类型都可以被定义为 volatile 字段，只有以下类型才可被定义为 volatile：

- 引用类型。
- 指针类型（在不安全的上下文中）。
- 整型，如 sbyte、byte、short、ushort、int、uint、char、float 和 bool。
- 具有整数基类型的枚举类型。
- 已知为引用类型的泛型类型参数。
- IntPtr 和 UIntPtr。

注意观察一下，就能发现只有值或引用的位数不超过本机整型值的位数（在 32 位系统中，为 4 个字节）的类型才能成为 volatile。为什么会这样呢？我的理解是：编译器之所以保障 volatile 字段在多线程情况下总是获取到最新值，最重要的一点是 volatile 字段操作的原子性，即编译后的本地代码只用一条机器指令就能对 volatile 字段赋值了。如何保证操作的原子性呢？32 位系统中，对任何数据操作都是以 4 字节为基础，自然一条机器指定就能搞定执行一个小于 4 字节的赋值操作。但如果字段占用内存大于 4 个字节，那生成赋值语句的机器指令肯定大于一条，这样在多线程的上下文切换中，有可能刚刚赋值到一半，就被切换到其他线程了。这样，便无法保障 volatile 字段在多线程环境下总是呈现一个完整的、合法的最新值了。

以上是我对 volatile 关键字的理解，大家如有看法，请发表自己的评论。

8.2.7 Interlocked类操作

System.Threading.Interlocked 类为多个线程共享的变量提供原子操作。

为整型类提供原子类操作

经验显示，那些需要在多线程情况下被保护的资源通常是整型值，且这些整型值在多线程下最常见的操作就是递增、递减或相加操作。Interlocked 类提供了一个专门的机制用于完成这些特定的操作。这个类提供了 Increment、Decrement、Add 静态方法用于对 int 或 long 型变量的递增、递减或相加操作。

示例代码：

```
using System;
using System.Threading;

namespace ProcessTest
{
    class Program
    {
        static int counter = 1;
        static void Main(string[] args)
        {
            Thread t1 = new Thread(new ThreadStart(F1));
            Thread t2 = new Thread(new ThreadStart(F2));
            t1.Start();
            t2.Start();
            t1.Join();
            t2.Join();
            System.Console.ReadKey();
        }

        static void F1()
        {
            for (int i = 0; i < 5; i++)
            {
                Interlocked.Increment(ref counter);
                System.Console.WriteLine("Counter++ {0}", counter);
                Thread.Sleep(10);
            }
```

```
        }
        static void F2()
        {
            for (int i = 0; i < 5; i++)
            {
                Interlocked.Decrement(ref counter);
                System.Console.WriteLine("Counter-- {0}", counter);
                Thread.Sleep(10);
            }
        }
    }
}
```

Interlocked 类还提供了 Exchange（为整型或引用对象赋值）、CompareExchange（比较后再对整型或引用对象赋值），用于为整型或引用类型的赋值提供原子操作。

8.2.8 使用Semaphore类限制资源并发访问数

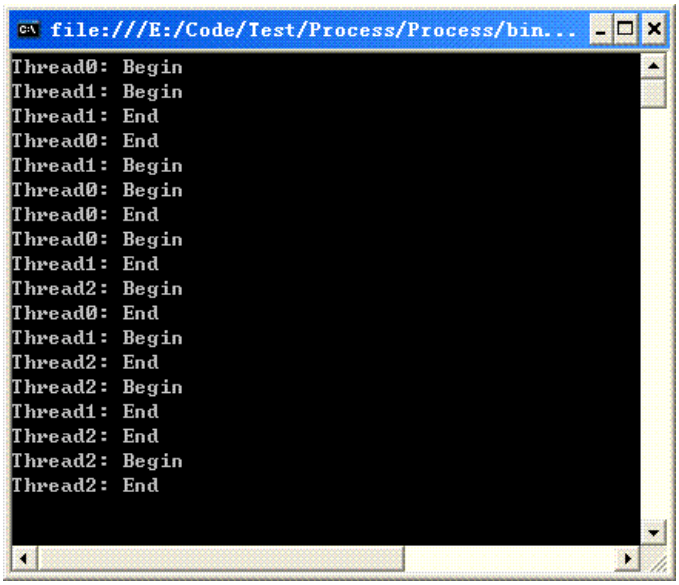
考虑这样的一个场景：一个停车场的只含一定数量的停车位，只有当停车场中还有空位时，停车场的入口才会打开。C#提供了 Semaphore 类来处理这种场景。Semaphore 类可以在构造时指定资源数量，使用 WaitOne 方法等待一个资源，使用 Release 方法释放一个资源。示例代码：

```
using System;
using System.Threading;

namespace ProcessTest{
    class Program
    {
        static Semaphore semaphore;
        static void Main(string[] args)
        {
            //创建一个限制资源类
            //资源数为 5，开放资源数为 2
            //主线程自动占有 3 个资源
            semaphore = new Semaphore(2, 5);
            //开启 3 个线程，让它们竞争剩余的 2 个资源
            for (int i = 0; i < 3; i++)
            {
                Thread t = new Thread(new ThreadStart(WorkerProc));
                t.Name = "Thread" + i;
                t.Start();
```

```
                Thread.Sleep(30);
            }
            System.Console.ReadKey();
        }
        static void WorkerProc()
        {
            for (int i = 0; i < 3; i++)
            {
                //等一个资源信号
                semaphore.WaitOne();
                Console.WriteLine(Thread.CurrentThread.Name + ": Begin");
                Thread.Sleep(200);
                Console.WriteLine(Thread.CurrentThread.Name + ": End");
                //释放资源
                semaphore.Release();
            }
        }
    }
}
```

程序首先创建一个 Semaphore 类，该类指定资源数为 5，但只释放出 2 个资源给 3 个线程竞争，其它 3 个资源被主线程所占用。子线程首先等待一个资源，获取到资源后，休眠 200ms，随后释放资源。

这段程序的输出如下：



8.2.9 用ReaderWriterLock类实现多用户读/单用户写同步

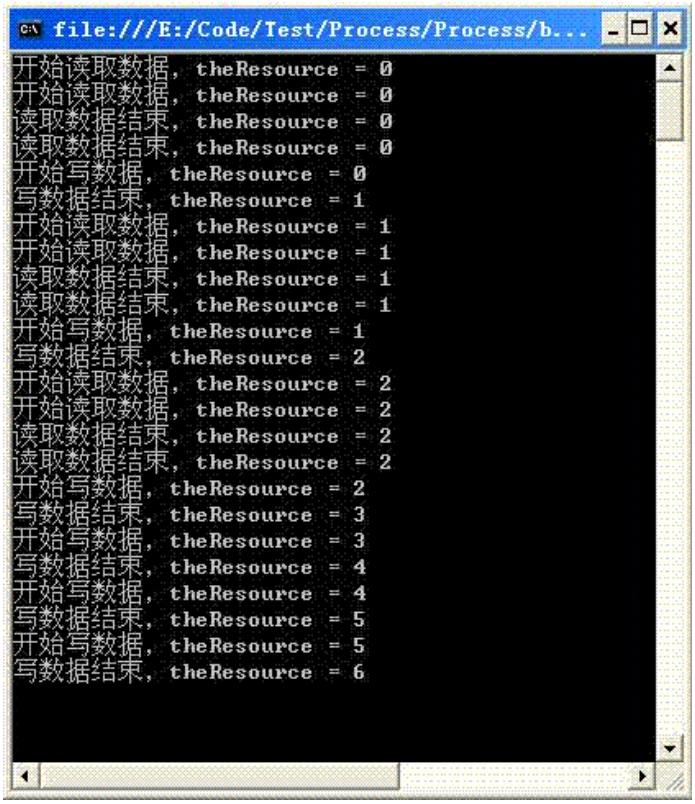
使用 Monitor 或 Mutex 进行同步控制的问题：由于独占访问模型不允许任何形式的并发访问，这样的效率总是不太高。许多时候，应用程序在访问资源时是进行读操作，写操作相对较少。为解决这一问题，C#提供了 System.Threading.ReaderWriterLock 类以适应多用户读/单用户写的场景。该类可实现以下功能：如果资源未被写操作锁定，那么任何线程都可对该资源进行读操作锁定，并且对读操作锁数量没有限制，即多个线程可同时对该资源进行读操作锁定，以读取数据。如果资源未被添加任何读或写操作锁，那么一个且仅有一个线程可对该资源添加写操作锁定，以写入数据。简单的讲就是：读操作锁是共享锁，允许多个线程同时读取数据；写操作锁是独占锁，同一时刻，仅允许一个线程进行写操作。

示例代码如下：

```
using System;
using System.Threading;

namespace ProcessTest{
    class Program
    {
        //资源
        static int theResource = 0;
        //读、写操作锁
        static ReaderWriterLock rwl = new ReaderWriterLock();
        static void Main(string[] args)
        {
            //分别创建 2 个读操作线程，2 个写操作线程，并启动
            Thread tr0 = new Thread(new ThreadStart(Read));
            Thread tr1 = new Thread(new ThreadStart(Read));
            Thread tr2 = new Thread(new ThreadStart(Write));
            Thread tr3 = new Thread(new ThreadStart(Write));
            tr0.Start();
            tr1.Start();
            tr2.Start();
            tr3.Start();
            //等待线程执行完毕
            tr0.Join();
            tr1.Join();
            tr2.Join();
            tr3.Join();
            System.Console.ReadKey();
        }
        //读数据
        static void Read()
```

```
        {
            for (int i = 0; i < 3; i++)
            {
                try
                {
                    //申请读操作锁，如果在 1000ms 内未获取读操作锁，则放弃
                    rwl.AcquireReaderLock(1000);
                    Console.WriteLine("开始读取数据，theResource = {0}", theResource);
                    Thread.Sleep(10);
                    Console.WriteLine("读取数据结束，theResource = {0}", theResource);
                    //释放读操作锁
                    rwl.ReleaseReaderLock();
                }
                catch (ApplicationException)
                {
                    //获取读操作锁失败的处理
                }
            }
        }
        //写数据
        static void Write()
        {
            for (int i = 0; i < 3; i++)
            {
                try
                {
                    //申请写操作锁，如果在 1000ms 内未获取写操作锁，则放弃
                    rwl.AcquireWriterLock(1000);
                    Console.WriteLine("开始写数据，theResource = {0}", theResource);
                    //将 theResource 加 1
                    theResource++;
                    Thread.Sleep(100);
                    Console.WriteLine("写数据结束，theResource = {0}", theResource);
                    //释放写操作锁
                    rwl.ReleaseWriterLock();
                }
                catch (ApplicationException)
                {
                    //获取写操作锁失败
                }
            }
        }
    }
}
```

上例中分别创建 2 个读取线程和 2 个写入线程，交替进行读、写操作。运行结果如下图：



观察运行结果，我们很容易看出：读操作锁是共享锁，允许多个线程同时读取数据；写操作锁是独占锁，仅允许一个线程进行写操作。

如果一个线程在获取读操作锁后，进行读操作的途中，希望提升锁级别，将其变为写操作锁，可以调用 ReaderWriterLock 类的 UpgradeToWriterLock(int timeOut)方法，该方法返回一个 LockCookie 值，该值保存了 UpgradeToWriterLock 方法调用前线程锁的状态。待写操作完成后，可调用 DowngradeFromWriterLock(LockCookie lockcookie)方法，该方法根据传入的 LockCookie 参数值，将线程锁恢复到 UpgradeToWriterLock 方法调用前的状态。具体使用方法，大家可以查看 MSDN 以获取相关示例。

8.2.10 异步方法调用

异步方法

当一个线程调用方法后，直到方法执行完毕，线程才继续执行，这种方法被称为同步方法。然而，有些方法执行时间可能非常长，比如串口操作或访问网络，这样线程被阻塞，而无法响应用户的其他请求。这种情况通常是无法忍受的，所以这时我们应该使用异步方法。

异步方法的原理是，在方法调用前为异步方法指定一个回调函数，方法调用后被线程池

中的一个线程接管，执行该方法。主线程立即返回，继续执行其他工作或响应用户请求。如果异步方法执行完毕，回调函数被自动执行，以处理异步方法的调用结果。

如何实现异步方法呢？C#通过异步委托调用 BeginInvoke 和 EndInvoke 方法来实现异步方法。

BeginInvoke 方法原型：

IAsyncResult BeginInvoke(....., AsyncCallback callback, object o);
.....表示异步委托中定义的参数列表。
AsyncCallback 参数是一个用于回调函数的委托，它的原型为：
public delegate void AsyncCallback(IAsyncResult ar)。其中 IAsyncResult 参数用于包装异步方法的执行结果。
Object 参数用于在主线程与回调函数间传递一些附加信息，如同步信息。

EndInvoke 方法原型：

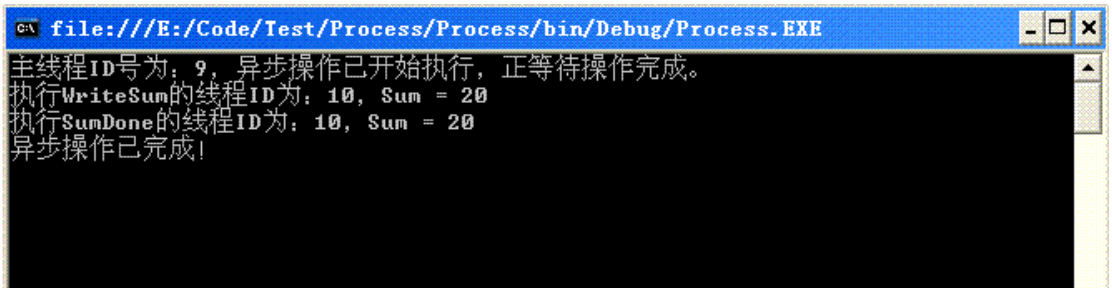
xxx EndInvoke(IAsyncResult result);
xxx 表示异步委托原型中定义的返回数据类型，IAsyncResult 用于包装异步方法的执行结果。

这么看着是不是有点迷糊？看个例子就明白了：

```
using System;
using System.Threading;
using System.Runtime.Remoting.Messaging;

namespace ProcessTest{
    class Program
    {
        //异步调用执行完成同步信号
        static AutoResetEvent ev = new AutoResetEvent(false);
        //定义委托
        public delegate int Deleg(int a, int b);
        static int WriteSum(int a, int b)
        {
            //显示当前线程 ID 号及 Sum 值
            Console.WriteLine("执行 WriteSum 的线程 ID 为：{0}，Sum = {1}",
Thread.CurrentThread.ManagedThreadId, a + b);
            return a + b;
        }
        //回调函数
        static void SumDone(IAsyncResult async)
```

```
        {
            //等待 1 秒，模拟线程正在执行其他工作
            Thread.Sleep(1000);
            //async 中包装了异步方法执行的结果
            //从操作结果 async 中还原委托
            Deleg proc = ((AsyncResult)async).AsyncDelegate as Deleg;
            //获取异步方法的执行结果
            int sum = proc.EndInvoke(async);
            //显示结果
            Console.WriteLine("执行 SumDone 的线程 ID 为：{0}，Sum = {1}",
Thread.CurrentThread.ManagedThreadId, sum);
            //使用 AsnycState 属性获取主线程中传入的同步信号
            //释放同步信号表示异步调用已完成
            ((AutoResetEvent)async.AsyncState).Set();
        }
        static void Main(string[] args)
        {
            //创建一个委托
            Deleg proc = new Deleg(WriteSum);
            //采用异步方式调用委托
            //指定 SumDone 为异步操作完成后的回调函数
            //指定 ev 为 object 参数，用于同步回调函数与主线程间操作
            IAsyncResult async = proc.BeginInvoke(10, 10, SumDone, ev);
            Console.WriteLine("主线程 ID 号为：{0}，异步操作已开始执行，正等待操作完成。",
Thread.CurrentThread.ManagedThreadId);
            //等待异步操作完成
            ev.WaitOne();
            Console.WriteLine("异步操作已完成！");
            System.Console.ReadKey();
        }
    }
}
```

下图是程序的运行结果：



注意观察运行结果，异步方法和回调函数是在同一步线程执行。

为方法指定 OneWay 特性

我们可将 System.Runtime.Remoting.Messaging.OneWay 特性应用于任何一个方法，该特性告诉 CLR 该方法不返回任何信息。即使该方法实际返回了数据（通过 return 语句或 out、ref 定义的参数），但只要被标记了 OneWay 特性，那它就不会再返回任何信息。

一个被标记为 OneWay 特性的方法即可以同步方式调用，也可以异步方式调用。如果在它的执行过程中引起了一个异常却没有捕获，在同步方式下，该异常会向上传播；但在异步方式下，该异常将不会被传播。大多数情况下，被标记为 OneWay 的方法是以异步方式工作。

8.2.11 异步事件调用

当一个事件被触发时，订阅该事件的方法将在触发该事件的线程中执行。也就是说，订阅该事件的方法在触发事件的线程中同步执行。由此，存在一个问题：如果订阅事件的方法执行时间很长，触发事件的线程被阻塞，长时间等待方法执行完毕。这样，不仅影响后续订阅事件方法的执行，也影响主线程及时响应用户的其他请求。如何处理这个问题呢？讲到此，我想您已经想到了，那就是异步事件调用。

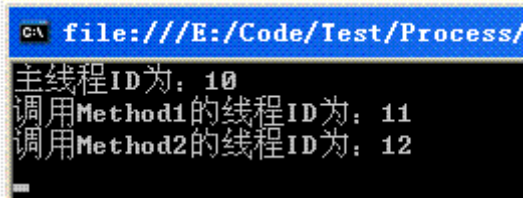
怎样实现异步事件调用呢？如果您对事件比较了解的话，您应该知道事件的本质其实是一种 MulticastDelegate（多播委托）。MulticastDelegate 类提供了一个 GetInvocationList 方法，该方法返回此多播委托的委托调用数组。利用该方法就能实现我们的异步事件调用功能。

示例代码：

```
using System;
using System.Threading;
using System.Runtime.Remoting.Messaging;

namespace ProcessTest{
    class Program
    {
        //定义一个事件
        public static event EventHandler<EventArgs> OnEvent;
        //方法 1
        static void Method1(object sender, EventArgs e)
        {
            //显示执行该方法的线程 ID
            Console.WriteLine("调用Method1的线程ID为：{0}",
Thread.CurrentThread.ManagedThreadId);
            Thread.Sleep(1000);
        }
        //方法 2
```

```
        static void Method2(object sender, EventArgs e)
        {
            Console.WriteLine("调用Method2的线程ID为：{0}",
Thread.CurrentThread.ManagedThreadId);
            Thread.Sleep(1000);
        }
        static void Main(string[] args)
        {
            //显示主线程 ID
            System.Console.WriteLine("主线程ID为：{0}",
Thread.CurrentThread.ManagedThreadId);
            //将 Method1 和 Method2 注册到事件中
            OnEvent += new EventHandler<EventArgs>(Method1);
            OnEvent += new EventHandler<EventArgs>(Method2);
            //下面的代码实现事件的异步调用
            //获取事件中的多路委托列表
            Delegate[] delegAry = OnEvent.GetInvocationList();
            //遍历委托列表
            foreach (EventHandler<EventArgs> deleg in delegAry)
            {
                //异步调用委托
                deleg.BeginInvoke(null, EventArgs.Empty, null, null);
            }
            System.Console.ReadKey();
        }
    }
}
```

代码执行结果如下图：



注意观察运行结果就可发现，主线程 ID、执行 Method1 函数的线程 ID、执行 Method2 函数的线程 ID 都不相同，由此可知我们实现了异步事件调用。

8.2.12 BackgroundWorker

可能您会觉得使用委托的 BeginInvoke 方法实现异步操作太过繁琐了，我们在多数情况

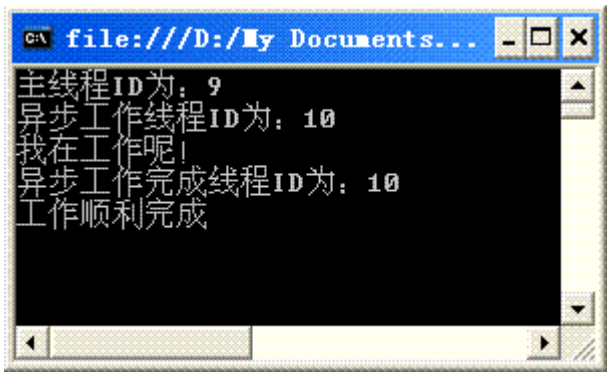
下仅仅是希望启动一个线程进行一些工作，完成后执行一个回调函数就 OK 了，并没有太多其他的需求。有没有更方便的方法实现异步操作呢？就像事件那样简单，注册一个异步操作函数，待函数执行完毕后再自动执行一个回调函数，而异步线程启动、运行、结束之类的操作都由程序自动完成。答案是有的，这就是 System.ComponentModel.BackgroundWorker 类的功能。

我们先看代码：

```
using System;
using System.Collections.Generic;
using System.Text;
using System.ComponentModel;
using System.Threading;

namespace ThreadTest
{
    class Program
    {
        //创建 BackgroundWorker 对象
        static BackgroundWorker bw = new BackgroundWorker();
        //同步信号
        static AutoResetEvent av = new AutoResetEvent(false);
        static void Main(string[] args)
        {
            //显示主线程 ID
            Console.WriteLine("主线程 ID 为：" + Thread.CurrentThread.ManagedThreadId);
            //添加异步操作的事件函数
            bw.DoWork += new DoWorkEventHandler(bw_DoWork);
            //添加异步操作完成后的回调函数
            bw.RunWorkerCompleted += new
RunWorkerCompletedEventHandler(bw_RunWorkerCompleted);
            //启动异步操作，并为其传递事件参数
            bw.RunWorkerAsync("我在工作呢！");
            //等待异步操作完成
            av.WaitOne();
            Console.ReadKey();
        }

        //异步操作工作函数
        static void bw_DoWork(object sender, DoWorkEventArgs e)
        {
            //显示异步工作线程 ID
            Console.WriteLine("异步工作线程 ID 为：" + Thread.CurrentThread.ManagedThreadId);
            //休眠 1 秒，模拟长时间工作
```

```
            Thread.Sleep(1000);
            //显示事件传入的参数
            Console.WriteLine(e.Argument);
            //写入异步操作的结果
            e.Result = "工作顺利完成";
        }
        //异步操作完成回调函数
        static void bw_RunWorkerCompleted(object sender, RunWorkerCompletedEventArgs e)
        {
            //显示异步工作完成线程 ID
            Console.WriteLine("异步工作完成线程ID为：" +
Thread.CurrentThread.ManagedThreadId);
            //显示异步操作结果
            Console.WriteLine(e.Result);
            //发送同步信号
            av.Set();
        }
    }
}
```

程序运行的结果如下图：



从运行结果我们可看出，异步操作的线程与主线程各自处于不同线程，实现了异步操作。

简要解释一下代码：

1. 创建一个 BackgroundWorker 类。
2. 为该类的 DoWork 事件注册异步操作工作函数。
3. 为该类的 RunWorkerCompleted 事件注册异步操作完成后的回调函数。
4. 执行该类的 RunWorkerAsync 方法，启动异步操作。可为该方法指定一个 object 类型的参数，该参数被传递给异步操作工作函数，用于存储一些工作函数所需的必要信息。
5. 在异步操作工作函数中可用 DoWorkEventArgs 参数的 Argument 属性读取由主线程存入的数据；可用该参数的 Result 属性存入异步操作的结果，该结果会被传递给异步操作完成后的回调函数。

6. 回调函数可使用传入的 RunWorkerCompletedEventArgs 参数的 Result 属性接收工作函数中存入的操作结果，并进行相关处理。