浅析：C# 中构建多线程应用程序

作者：刘志远

Email：592418843@qq.com

目 录

引言

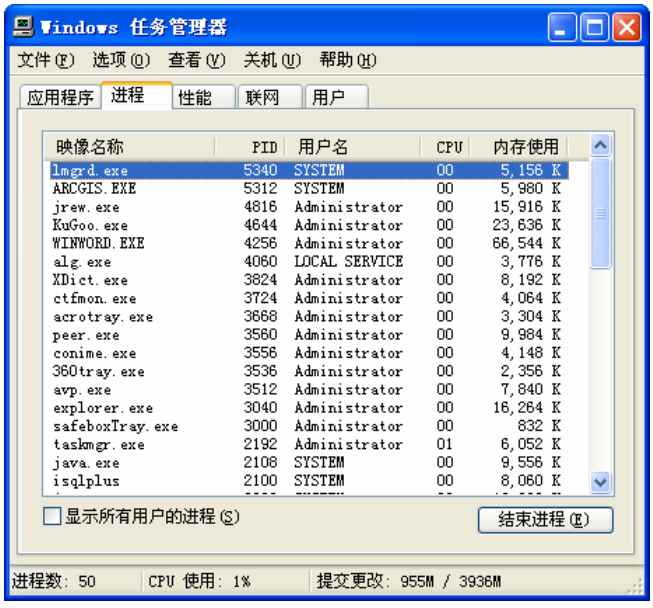
随着双核、四核等多核处理器的推广，多核处理器或超线程单核处理器的计算机已很常见，基于多核处理的编程技术也开始受到程序员们普遍关注。这其中一个重要的方面就是构建多线程应用程序（因为不使用多线程的话，开发人员就不能充分发挥多核计算机的强大性能）。

本文针对的是构建基于单核计算机的多线程应用程序，目的在于介绍多线程相关的基本概念、内涵，以及如何通过 System.Threading 命名空间的类、委托和 BackgroundWorker 组件等三种手段构建多线程应用程序。

本文如果能为刚接触多线程的朋友起到抛砖引玉的作用也就心满意足了。当然，本人才疏学浅，文中难免会有不足或错误的地方，恳请各位朋友多多指点。

1. 理解多线程

我们通常理解的应用程序就是一个*.exe 文件，当运行*.exe 应用程序以后，系统会在内存中为该程序分配一定的空间，同时加载一些该程序所需的资源。其实这就可以称为创建了一个进程，可以通过 Windows 任务管理器查看这个进程的相关信息，如映像名称、用户名、内存使用、PID（唯一的进程标示）等，如图下所示。



而线程则只是进程中的一个基本执行单元。一个应用程序往往只有一个程序入口，如：

```
[STAThread]
static void Main()   //应用程序主入口点
{
    Application.EnableVisualStyles();
    Application.SetCompatibleTextRenderingDefault(false);
    Application.Run(new MainForm());
}
```

进程会包含一个进入此入口的线程，我们称之为主线程。其中，特性 [STAThread] 指示应用程序的默认线程模型是单线程单元（相关信息可参考 http://msdn.microsoft.com/en-us/library/system.stathreadattribute(VS.71).aspx）。只包含一个主线程的进程是线程安全的，相当于程序仅有一条工作线，只有完成了前面的任务才能执行排在后面的任务。

然当在程序处理一个很耗时的任务，如输出一个大的文件或远程访问数据库等，此时的窗体界面程序对用户而言基本像是没反应一样，菜单、按钮等都用不了。因为窗体上控件的响应事件也是需要主线程来执行的，而主线程正忙着干其他的事，控件响应事件就只能排队等着主线程忙完了再执行。

为了克服单线程的这个缺陷，Win32 API 可以让主线程再创建其他的次线程，但不论是主线程还是次线程都是进程中独立的执行单元，可以同时访问共享的数据，这样就有了多线程这个概念。

相信到这，应该对多线程有个比较感性的认识了。但笔者在这要提醒一下，基于单核计算机的多线程其实只是操作系统施展的一个障眼法而已(但这不会干扰我们理解构建多线程应用程序的思路)，他并不能缩短完成所有任务的时间，有时反而还会因为使用过多的线程而降低性能、延长时间。之所以这样，是因为对于单 CPU 而言，在一个单位时间（也称时间片）内，只能执行一个线程，即只能干一件事。当一个线程的时间片用完时，系统会将该线程挂起，下一个时间内再执行另一个线程，如此，CPU 以时间片为间隔在多个线程之间交替执行运算（其实这里还与每个线程的优先级有关，级别高的会优先处理）。由于交替时间间隔很短，所以造成了各个线程都在“同时”工作的假象；而如果线程数目过多，由于系统挂起线程时要记录线程当前的状态数据等，这样又势必会降低程序的整体性能。但对于这些，多核计算机就能从本质上（真正的同时工作）提高程序的执行效率。

2. 线程异步与线程同步

从线程执行任务的方式上可以分为线程同步和线程异步。而为了方便理解，后面描述中用“同步线程”指代与线程同步相关的线程，同样，用“异步线程”表示与线程异步相关的线程。

线程异步就是解决类似前面提到的执行耗时任务时界面控件不能使用的问题。如创建一个次线程去专门执行耗时的任务，而其他如界面控件响应这样的任务交给另一个线程执行（往往由主线程执行）。这样，两个线程之间通过线程调度器短时间（时间片）内的切换，就模拟出多个任务“同时”被执行的效果。

线程异步往往是通过创建多个线程执行多个任务，多个工作线同时开工，类似多辆在宽广的公路上并行的汽车同时前进，互不干扰（读者要明白，本质上并没有“同时”，仅仅是操作系统玩的一个障眼法。但这个障眼法却对提高我们的程序与用户之间的交互、以及提高程序的友好性很有用，不是吗）。

在介绍线程同步之前，先介绍一个与此紧密相关的概念——**并发问题**。

前面提到，线程都是独立的执行单元，可以访问共享的数据。也就是说，在一个拥有多个次线程的程序中，每个线程都可以访问同一个共享的数据。再稍加思考你会发现这样可能会出问题：由于线程调度器会随机的挂起某一个线程（前面介绍的线程间的切换），所以当线程 a 对共享数据 D 的访问（修改、删除等操作）完成之前被挂起，而此时线程 b 又恰好去访问数据 D，那么线程 b 访问的则是一个不稳定的数据。这样就会产生非常难以发现 bug，由于是随机发生的，产生的结果是不可预测的，这样样的 bug 也都很难重现和调试。这就是并发问题。

为了解决多线程共同访问一个共享资源（也称互斥访问）时产生的并发问题，线程同步就应运而生了。线程同步的机理，简单的说，就是防止多个线程同时访问某个共享的资源。做法很简单，标记访问某共享资源的那部分代码，当程序运行到有标记的地方时，CLR（具体是什么可以先不管，只要知道它能控制就行）对各线程进行调整：如果已有线程在访问一资源，CLR 就会将其他访问这一资源的线程挂起，直到前一线程结束对该资源的访问。这样

就保证了同一时间只有一个线程访问该资源。打个比方，就如某资源放在只有一独木桥相连的孤岛上，如果要使用该资源，大家就得排队，一个一个来，前面的回来了，下一个再去，前面的没回来，后面的就原地待命。

这里只是把基本的概念及原理做了一个简单的阐述，不至于看后面的程序时糊里糊涂的。具体如何编写代码，下面的段落将做详细介绍。

3.创建多线程应用程序

这里做一个简单的说明：下面主要通过介绍通过 System.Threading 命名空间的类、委托和 BackgroundWorker 组件三种不同的手段构建多线程应用程序，具体会从线程异步和线程同步两个方面来阐述。

3.1 通过 System.Threading 命名空间的类构建

在.NET 平台下，System.Threading 命名空间提供了许多类型来构建多线程应用程序，可以说是专为多线程服务的。由于本文仅是想起到一个“抛砖引玉”的作用，所以对于这一块不会探讨过多、过深，主要使用 System.Threading.Thread 类。

先从 System.Threading.Thread 类本身相关的一个小例子说起,代码如下,解释见注释：

```
using System;
using System.Threading;//引入System.Threading命名空间
namespace MultiThread
{
    class Class
    {
        static void Main(string[] args)
        {
            Console.WriteLine("********** 显示当前线程的相关信息 **********");
            //声明线程变量并赋值为当前线程
            Thread primaryThread = Thread.CurrentThread;
            //赋值线程的名称
            primaryThread.Name = "主线程";
            //显示线程的相关信息
            Console.WriteLine("线程的名字：{0}", primaryThread.Name);
            Console.WriteLine("线程是否启动？ {0}", primaryThread.IsAlive);
            Console.WriteLine("线程的优先级： {0}", primaryThread.Priority);
            Console.WriteLine("线程的状态： {0}", primaryThread.ThreadState);
```

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

输出结果如下：

********** 显示当前线程的相关信息 **********
线程的名字：主线程
线程是否启动？ True
线程的优先级： Normal
线程的状态： Running

对于上面的代码不想做过多解释，只说一下 Thread.CurrentThread 得到的是执行当前代码的线程。

3.1.1 异步调用线程

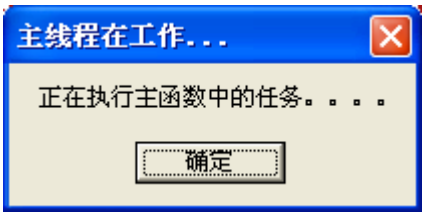
这里先说一下**前台线程**与**后台线程**。前台线程能阻止应用程序的终止，既直到所有前台线程终止后才会彻底关闭应用程序。而对后台线程而言，当所有前台线程终止时，后台线程会被自动终止，不论后台线程是否正在执行任务。默认情况下通过 Thread.Start()方法创建的线程都自动为前台线程，把线程的属性 IsBackground 设为 true 时就将线程转为后台线程。

下面先看一个例子，该例子创建一个次线程执行打印数字的任务，而主线程则干其他的事，两者同时进行，互不干扰。

```
using System;
using System.Threading;
using System.Windows.Forms;
namespace MultiThread
{
    class Class
    {
        static void Main(string[] args)
        {
            Console.WriteLine("********** 两个线程同时工作 **********");
            //主线程，因为获得的是当前在执行Main()的线程
            Thread primaryThread = Thread.CurrentThread;
            primaryThread.Name = "主线程";
```

```
            Console.WriteLine("-> {0} 在执行主函数 Main()。", Thread.CurrentThread.Name);
            //次线程，该线程指向PrintNumbers()方法
            Thread SecondThread = new Thread(new ThreadStart(PrintNumbers));
            SecondThread.Name = "次线程";
            //次线程开始执行指向的方法
            SecondThread.Start();

            //同时主线程在执行主函数中的其他任务
            MessageBox.Show("正在执行主函数中的任务。。。。", "主线程在工作...");
            Console.ReadLine();
        }
        //打印数字的方法
        static void PrintNumbers()
        {
            Console.WriteLine("-> {0} 在执行打印数字函数 PrintNumber()",
Thread.CurrentThread.Name);
            Console.WriteLine("打印数字： ");
            for (int i = 0; i < 10; i++)
            {
                Console.Write("{0}, ", i);
                //Sleep()方法使当前线程挂等待指定的时长在执行，这里主要是模仿打印任务
                Thread.Sleep(2000);
            }
            Console.WriteLine();
        }
    }
}
```

程序运行后会看到一个窗口弹出，如图所示，同时控制台窗口也在不断的显示数字。



输出结果为：

```
********** 两个线程同时工作 **********
-> 主线程 在执行主函数 Main()。
-> 次线程 在执行打印数字函数 PrintNumber()
打印数字：
0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
```

这里稍微对 Thread SecondThread = new Thread(new ThreadStart(PrintNumbers)); 这一句做个解

释。其实 ThreadStart 是 System.Threading 命名空间下的一个委托，其声明是 public delegate void ThreadStart()，指向不带参数、返回值为空的方法。所以当使用 ThreadStart 时，对应的线程就只能调用不带参数、返回值为空的方法。那非要指向含参数的方法呢？在System.Threading命名空间下还有一个 ParameterizedThreadStart 委托，其声明是 public delegate void ParameterizedThreadStart(object obj)，可以指向含 object 类型参数的方法，这里不要忘了 object 可是所有类型的父类哦，有了它就可以通过创建各种自定义类型，如结构、类等传递很多参数了，这里就不再举例说明了。

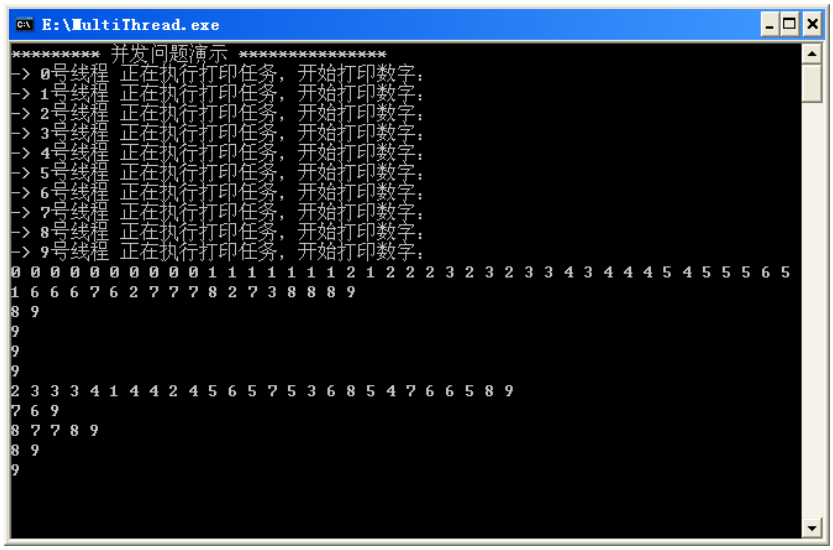
3.1.2 并发问题

这里再通过一个例子让大家切实体会一下前面说到的并发问题，然后再介绍线程同步。

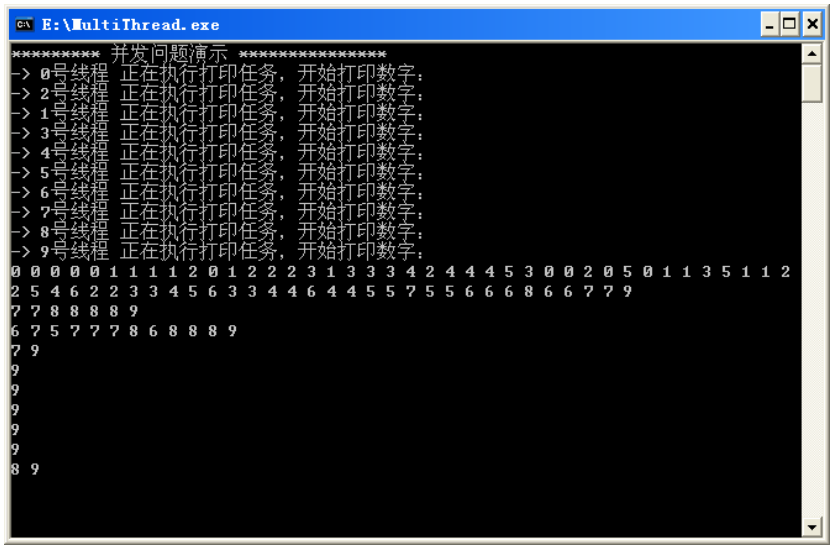
```
using System;
using System.Threading;
namespace MultiThread1
{
    class Class
    {
        static void Main(string[] args)
        {
            Console.WriteLine("********* 并发问题演示 **************");
            //创建一个打印对象实例
            Printer printer = new Printer();
            //声明一含5个线程对象的数组
            Thread[] threads = new Thread[10];

            for (int i = 0; i < 10; i++)
            {
                //将每一个线程都指向printer的PrintNumbers()方法
                threads[i] = new Thread(new ThreadStart(printer.PrintNumbers));
                //给每一个线程编号
                threads[i].Name = i.ToString() +"号线程";
            }

            //开始执行所有线程
            foreach (Thread t in threads)
                t.Start();
            Console.ReadLine();
        }
    }
    //打印类
    public class Printer
```

```
    {
        //打印数字的方法
        public void PrintNumbers()
        {
            Console.WriteLine("-> {0} 正在执行打印任务，开始打印数字：",
Thread.CurrentThread.Name);

            for (int i = 0; i < 10; i++)
            {
                Random r = new Random();
                //为了增加冲突的几率及，使各线程各自等待随机的时长
                Thread.Sleep(2000 * r.Next(5));
                //打印数字
                Console.Write("{0} ", i);
            }
            Console.WriteLine();
        }
    }
}
```

上面的例子中，主线程产生的 10 个线程同时访问同一个对象实例 printer 的方法 PrintNumbers()，由于没有锁定共享资源（注意，这里是指控制台），所以在 PrintNumbers()输出到控制台之前，调用 PrintNumbers()的线程很可能被挂起，但不知道什么时候（或是否有）挂起，导致得到不可预测的结果。如下是两个不同的结果（当然，读者的运行结果可能会是其他情形）。



情形一



情形二

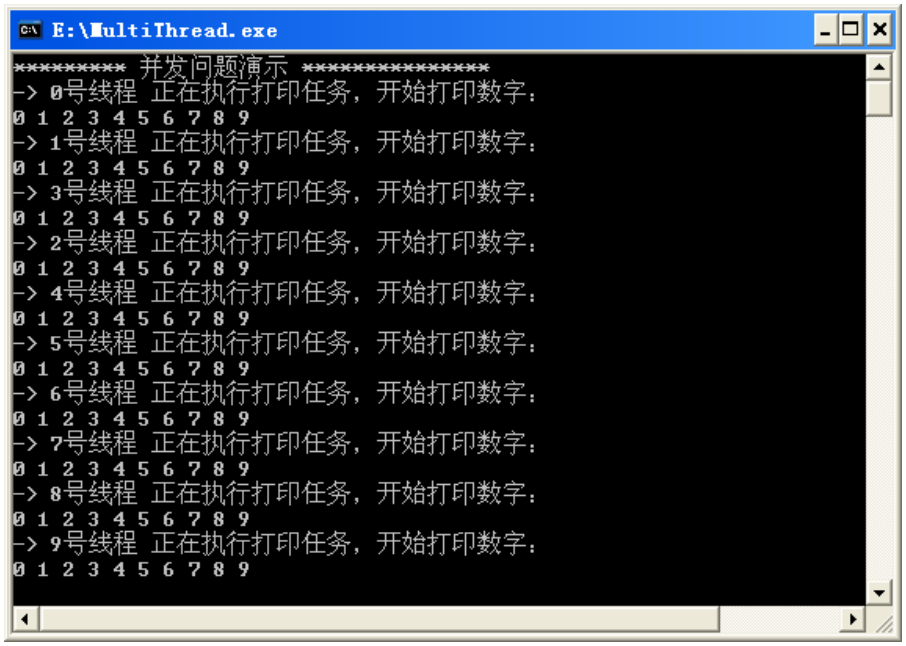
3.1.3 线程同步

线程同步的访问方式也称为阻塞调用，即没有执行完任务不返回，线程被挂起。可以使用 C#中的 lock 关键字，在此关键字范围类的代码都将是线程安全的。lock 关键字需定义一个标记，线程进入锁定范围是必须获得这个标记。当锁定的是一个实例级对象的私有方法时使用方法本身所在对象的引用就可以了，将上面例子中的打印类 Printer 稍做改动，添加 lock 关键字，代码如下：

```
    //打印类
    public class Printer
    {
        public void PrintNumbers()
        {
            //使用lock关键字，锁定d的代码是线程安全的
            lock (this)
            {
                Console.WriteLine("-> {0} 正在执行打印任务，开始打印数字：",
Thread.CurrentThread.Name);

                for (int i = 0; i < 10; i++)
                {
                    Random r = new Random();
                    //为了增加冲突的几率及，使各线程各自等待随机的时长
                    Thread.Sleep(2000 * r.Next(5));
                    //打印数字
                    Console.Write("{0} ", i);
                }
                Console.WriteLine();
```

```
            }
        }
    }
}
```

同步后执行结果如下：



也可以使用 System.Threading 命名空间下的 Monitor 类进行同步，两者内涵是一样的，但 Monitor 类更灵活，这里就不在做过多的探讨，代码如下：

```
    //打印类
    public class Printer
    {
        public void PrintNumbers()
        {
            Monitor.Enter(this);
            try
            {
                Console.WriteLine("-> {0} 正在执行打印任务，开始打印数字：",
Thread.CurrentThread.Name);

                for (int i = 0; i < 10; i++)
                {
                    Random r = new Random();
                    //为了增加冲突的几率及，使各线程各自等待随机的时长
                    Thread.Sleep(2000 * r.Next(5));
                    //打印数字
```

```
                Console.Write("{0} ", i);
            }
            Console.WriteLine();
        }
        finally
        {
            Monitor.Exit(this);
        }
    }
}
```

输出结果与上面的一样。

3.2 通过委托构建多线程应用程序

在看下面的内容时要求对委托有一定的了解,如果不清楚的话推荐参考一下博客园张子阳的《C# 中的委托和事件》，里面对委托与事件进行由浅入深的较系统的讲解：http://www.cnblogs.com/JimmyZhang/archive/2007/09/23/903360.html。

这里先举一个关于委托的简单例子，具体解说见注释：

```
using System;
namespace MultiThread
{
    //定义一个指向包含两个int型参数、返回值为int型的函数的委托
    public delegate int AddOp(int x, int y);
    class Program
    {
        static void Main(string[] args)
        {
            //创建一个指向Add()方法的AddOp对象p
            AddOp pAddOp = new AddOp(Add);
            //使用委托间接调用方法Add()
            Console.WriteLine("10 + 25 = {0}", pAddOp(10, 5));
            Console.ReadLine();
        }
        //求和的函数
        static int Add(int x, int y)
        {
            int sum = x + y;
            return sum;
        }
    }
}
```

运行结果为：
10 + 25 = 15

3.2.1 线程异步

先说明一下，这里不打算讲解委托线程异步或同步的参数传递、获取返回值等，只是做个一般性的开头而已，如果后面有时间了再另外写一篇关于多线程中参数传递、获取返回值的文章。

注意观察上面的例子会发现，直接使用委托实例 pAddOp(10, 5) 就调用了求和方法 Add()。很明显，这个方法是由主线程执行的。然而，委托类型中还有另外两个方法——BeginInvoke()和EndInvoke()，下面通过具体的例子来说明，将上面的例子做适当改动，如下：

```
using System;
using System.Threading;
using System.Runtime.Remoting.Messaging;
namespace MultiThread
{
    //声明指向含两个int型参数、返回值为int型的函数的委托
    public delegate int AddOp(int x, int y);
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("******* 委托异步线程 两个线程“同时”工作 *********");
            //显示主线程的唯一标示
            Console.WriteLine("调用Main()的主线程的线程ID是：{0}.",
Thread.CurrentThread.ManagedThreadId);
            //将委托实例指向Add()方法
            AddOp pAddOp = new AddOp(Add);
            //开始委托次线程调用。委托BeginInvoke()方法返回的类型是IAsyncResult，
            //包含这委托指向方法结束返回的值，同时也是EndInvoke()方法参数
            IAsyncResult iftAR = pAddOp.BeginInvoke(10, 10, null, null);
            Console.WriteLine("\nMain()方法中执行其他任务.....\n");
            int sum = pAddOp.EndInvoke(iftAR);
            Console.WriteLine("10 + 10 = {0}.", sum);
            Console.ReadLine();
        }
        //求和方法
        static int Add(int x, int y)
        {
            //指示调用该方法的线程ID，ManagedThreadId是线程的唯一标示
            Console.WriteLine("调用求和方法 Add()的线程ID是： {0}.",
Thread.CurrentThread.ManagedThreadId);
            //模拟一个过程，停留5秒
```

```
            Thread.Sleep(5000);
            int sum = x + y;
            return sum;
        }
    }
}
```

运行结果如下：

******* 委托异步线程 两个线程“同时”工作 *********
调用 Main()的主线程的线程 ID 是：10.

Main()方法中执行其他任务.....

调用求和方法 Add()的线程 ID 是： 7.
10 + 10 = 20.

3.2.2 线程同步

委托中的线程同步主要涉及到上面使用的 pAddOp.BeginInvoke(10, 10, null, null)方法中后面两个为 null 的参数，具体的可以参考相关资料。这里代码如下，解释见代码注释：

```
using System;
using System.Threading;
using System.Runtime.Remoting.Messaging;
namespace MultiThread
{
    //声明指向含两个int型参数、返回值为int型的函数的委托
    public delegate int AddOp(int x, int y);
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("******* 线程同步，“阻塞”调用，两个线程工作 *********");
            Console.WriteLine("Main() invokee on thread {0}.",
Thread.CurrentThread.ManagedThreadId);
            //将委托实例指向Add()方法
            AddOp pAddOp = new AddOp(Add);
            IAsyncResult iftAR = pAddOp.BeginInvoke(10, 10, null, null);
            //判断委托线程是否执行完任务，
            //没有完成的话，主线程就做其他的事
            while (!iftAR.IsCompleted)
            {
                Console.WriteLine("Main()方法工作中.....");
                Thread.Sleep(1000);
            }
            //获得返回值
```

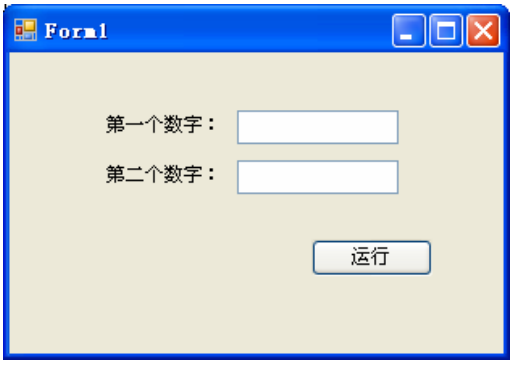
```
            int answer = pAddOp.EndInvoke(iftAR);
            Console.WriteLine("10 + 10 = {0}.", answer);
            Console.ReadLine();
        }
        //求和方法
        static int Add(int x, int y)
        {
            //指示调用该方法的线程ID，ManagedThreadId是线程的唯一标示
            Console.WriteLine("调用求和方法 Add()的线程ID是： {0}.",
Thread.CurrentThread.ManagedThreadId);
            //模拟一个过程，停留5秒
            Thread.Sleep(5000);
            int sum = x + y;
            return sum;
        }
    }
}
```

运行结果如下：

```
******* 线程同步，“阻塞”调用，两个线程工作 *********
Main() invokee on thread 10.
Main()方法工作中.......
调用求和方法 Add()的线程 ID 是： 7.
Main()方法工作中.......
Main()方法工作中.......
Main()方法工作中.......
Main()方法工作中.......
10 + 10 = 20.
```

3.3BackgroundWorker 组件

BackgroundWorker 组件位于工具箱中，用于方便的创建线程异步的程序。新建一个 WindowsForms 应用程序，界面如下：



代码如下，解释参见注释：

```
        private void button1_Click(object sender, EventArgs e)
        {
            try
            {
                //获得输入的数字
                int numOne = int.Parse(this.textBox1.Text);
                int numTwo = int.Parse(this.textBox2.Text);
                //实例化参数类
                AddParams args = new AddParams(numOne, numTwo);
                //调用RunWorkerAsync()生成后台线程，同时传入参数
                this.backgroundWorker1.RunWorkerAsync(args);
            }
            catch (Exception ex)
            {
                MessageBox.Show(ex.Message);
            }
        }

        //backgroundWorker新生成的线程开始工作
        private void backgroundWorker1_DoWork(object sender, DoWorkEventArgs e)
        {
            //获取传入的AddParams对象
            AddParams args = (AddParams)e.Argument;
            //停留5秒，模拟耗时任务
            Thread.Sleep(5000);
            //返回值
            e.Result = args.a + args.b;
        }

        //当backgroundWorker1的DoWork中的代码执行完后会触发该事件
        //同时，其执行的结果会包含在RunWorkerCompletedEventArgs参数中
        private void backgroundWorker1_RunWorkerCompleted(object sender,
RunWorkerCompletedEventArgs e)
        {
            //显示运算结果
            MessageBox.Show("运行结果为：" + e.Result.ToString(), "结果");
        }
    }

    //参数类，这个类仅仅起到一个记录并传递参数的作用
    class AddParams
    {
        public int a, b;
```

```
        public AddParams(int numb1, int numb2)
        {
            a = numb1;
            b = numb2;
        }
    }
```

注意，在计算结果的同时，窗体可以随意移动，也可以重新在文本框中输入信息，这就说明主线程与 backgroundWorker 组件生成的线程是异步的。

4. 总结

本文从线程、进程、应用程序的关系开始，介绍了一些关于多线程的基本概念，同时阐述了线程异步、线程同步及并发问题等。最后从应用角度出发，介绍了如何通过 System.Threading 命名空间的类、委托和 BackgroundWorker 组件等三种手段构建多线程应用程序。

再次说一下，由于本人才疏学浅，文中难免会有不足或错误的地方，真诚期盼各位同道朋友批评指正。

参考资料：

Andrew Troelsen：《C#与.NET 3.5 高级程序设计》(第 4 版)