C#程序设计及应用

唐大仕

dstang2000@263.net

北京大学

第10章 线程、异步、并行编程

唐大仕
dstang2000@263.net
http://www.dstang.com

线程及其控制

内容

- 1. 线程及其创建
- 2. 线程同步控制
- 3. 线程池及其他线程类
- 4. 线程在集合中使用
- 5. 线程在Window界面中使用

多线程的概念

- 进程Process
- 线程Thread
  - 线程中的指令：一个方法(委托)
  - 线程中的数据：相关的对象

System.Threading.Thread属性

Property	描　述
CurrentPrincipal	获取或者设定线程的当前安全性
CurrentThread	获得对当前正在运行的线程的一个引用(static属性)
IsAlive	如果线程已经被启动并且尚在生命周期内，则返回True
IsBackground	如果目标线程是在后台执行的，则为此属性赋值为True
Name	获取或者设定这个线程的名字
Priority	获取或者设定这个线程的优先级
ThreadState	获得线程的当前状态

System.Threading.Thread方法

Method	描　述
Abort	撤消这个线程
Interrupt	如果线程处于WaitSleepJoin状态，则中断它
Join	等待一个线程的结束
Resume	将被挂起的线程重新开始
Sleep	让线程休眠一定时间
Start	启动一个线程
Suspend	挂起一个线程

线程的创建

- Thread类有一个构造方法，格式如下：
  - public Thread(ThreadStart fun);
- 其中ThreadStart是一个委托：
  - public delegate void ThreadStart();
- 下面是创建一个Thread对象并启动这个线程的一般方法：
  - Thread thread =
    - new Thread(new ThreadStart(obj.fun));
  - thread.Start();
  - 有时，使用匿名函数及Lambda表达式更方便

ThreadTest.cs

线程的启动和停止

- 启动：调用线程对象的Start()
- 停止
  - 线程函数会一直执行下去，直至它结束
- 另外
  - Abort() 终止
  - Suspend() 挂起　Resume() 恢复
  - Sleep(毫秒数)

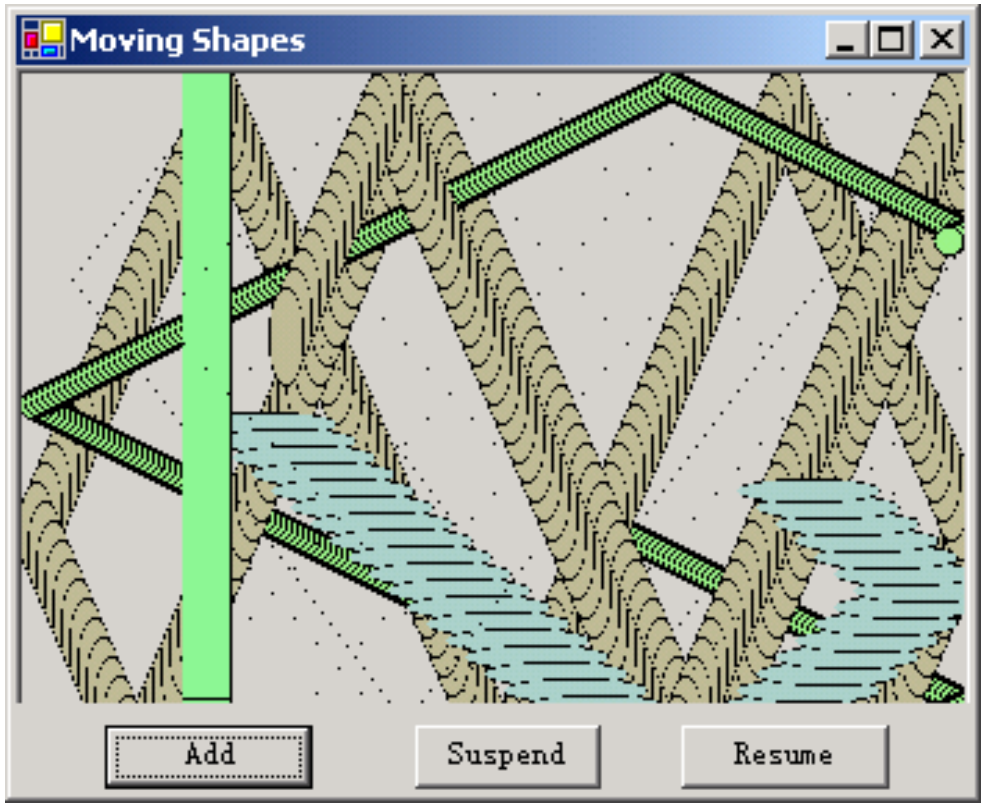
线程的状态 ThreadState

成 员	描 述
Aborted	线程已经被中断并且被撤销
AbortRequested	线程正在被请求中断
Background	线程充当后台线程的角色，并且正在执行
Running	线程正在运行
Stopped	线程停止运行(这个状态只限于内部使用)
StopRequested	线程正在被要求停止(这个状态只限于内部使用)
Suspended	线程已经被挂起
SuspendRequested	线程已经被要求挂起
Unstarted	线程还没有被启动
WaitSleepJoin	线程在一次Wait()、Sleep()以及Join()调用中被锁定

线程的优先级 ThreadPriority

成　员	描　述
Highest	线程具有最高优先级
AboveNormal	线程的优先级高于普通优先级
Normal	线程具有平均优先级
BelowNormal	线程的优先级低于普通优先级
Lowest	线程具有最低优先级

线程应用举例



ThreadDraw.cs

2. 线程的同步

- 使用Join()方法
  - 将单独的执行线程合并成一个线程

ThreadJoin.cs

Lock语句与Monitor类

- lock(对象或表达式){
- … 语句
- }

```
System.Threading.Monitor.Enter(对象或表达式);
try {
 ...
} finally {
        System.Threading.Monitor.Exit(对象或表达式);
}
```

用于同步控制的类

类	用　途
AutoResetEvent	等待句柄，用于通知一个或多个等待线程发生了一个事件。AutoResetEvent 在等待线程被释放后自动将状态更改为已发出信号。
Interlocked	为多个线程共享的变量提供原子操作。
ManualResetEvent	等待句柄，用于通知一个或多个等待线程发生了一个事件。手动重置事件的状态将保持为已发出信号，直至 Reset 方法将其设置为未发出信号状态。同样，该状态将保持为未发出信号，直至 Set 方法将其设置为已发出信号状态。当对象的状态为已发出信号时，任意数量的等待线程（即通过调用一个等待函数开始对指定事件对象执行等待操作的线程）都可以被释放。
Monitor	提供同步访问对象的机制。
Mutex	等待句柄，可用于进程间同步。
ReaderWriterLock	定义用于实现单个写入者和多个读取者的锁定。
Timer	提供按指定间隔运行任务的机制。
WaitHandle	封装操作系统特有的、等待对共享资源进行独占访问的对象。

3. 线程池及其他相关类

线程池（ThreadPool）

- Threadpool.QueueUserWorkItem()等方法来提交相应的任务
  - QueueUserWorkItem(WaitCallback, object)
  - QueueUserWorkItem(WaitCallback)
  - 其中public delegate void WaitCallback(object state);

ThreadPoolTest.cs

System.Threading.Timer类

- Timer的构造方法如下：
  - public Timer(
  - TimerCallback callback, //执行的任务
  - object state, // 数据
  - int dueTime, // 启动前的延时
  - int period // 任务之间的间隔
  -);
- 其中TimerCallback是：
  - public delegate void TimerCallback(object state);

TimerTest.cs

System.Windows.Forms.Timer类

- 直接从工具箱拖过来
  - 属性 Interval, Enabled
  - 事件 Tick

4. 集合的线程安全性

- IsSynchoronized属性用于判断是否为同步版本；SyncRoot属性提供了集合自己的同步版本
- Array，ArrayList，SortedList，Hashtable等，都可以使用Synchronized()方法获取一个线程安全的包装对象

SynchernizedArrayList.cs

5. Windows界面与线程

- 界面的主线程
- 对界面的更新只能使用主线程
- 其他线程则可以这样：
  - if(this.InvokeRequired){
    - this.BeginInvoke(new AddMsg(this.AddMsgFun), new object[]{ msg }); //显示到界面上
  - }else{
    - this.AddMsgFun(msg);
  - }
- 见示例 WinFormThreadUpdateUI

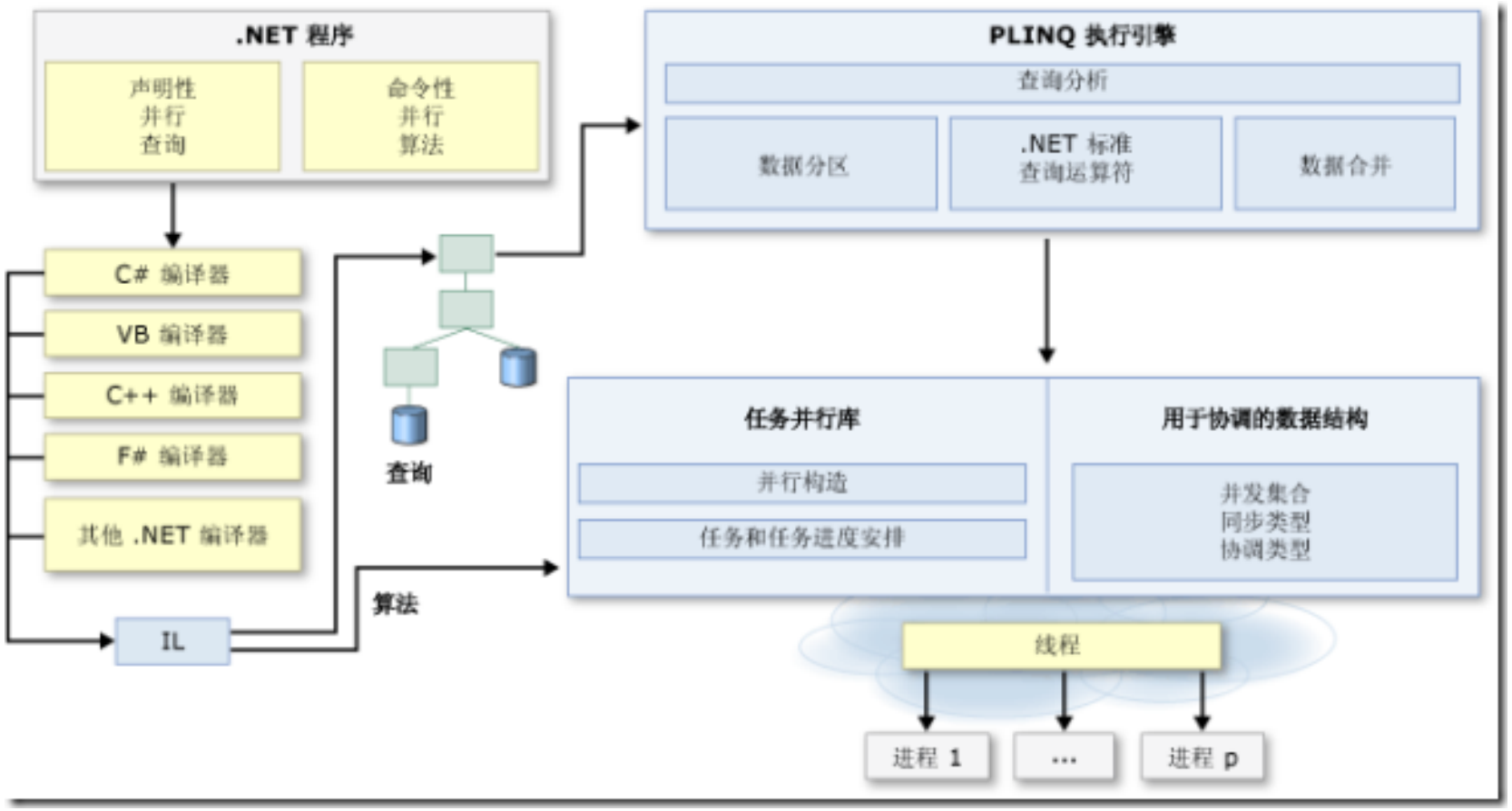
使用BackgroundWorker组件

- DoWork事件
- RunWorkerAsync方法

- 见示例 BackgroudWorkerTest

并行编程

并行任务库 TPL


.NET 程序
声明性
并行
查询
命令性
并行
算法
C# 编译器
VB 编译器
C++ 编译器
F# 编译器
其他 .NET 编译器
IL
查询
算法
PLINQ 执行引擎
查询分析
数据分区
.NET 标准
查询运算符
数据合并
任务并行库
并行构造
任务和任务进度安排
用于协调的数据结构
并发集合
同步类型
协调类型
线程
进程 1
...
进程 p

并行任务库

- 并行任务库（TPL，Task Parallel Library）
- 最重要的是Task类，还有Parallel类
- Task类，是利用线程池来进行任务的执行
  - 比直接用ThreadPool更优化，而且编程更方便
- Parallel类，是并行执行任务类的实用类
  - 好处是可以隐式地使用Task，更方便

Task类的使用

- 使用Task.Run方法来得到Task的实例
- Task<double> task = Task.Run(()=>SomeFun());
- double result = task.Result; //等待直到获得结果
- 可以使用 Task.WaitAll(task数组)
- 可以使用 task. ContinueWith(另一个task)

Task中的异常

- 可以使用AggregateException（合并的异常）

```
try
{
    Task.WaitAll(task1, task2, task3);
}
catch (AggregateException ex)
{
    foreach (Exception inner in ex.InnerExceptions)
    {
        Console.WriteLine("Exception type {0} from {1}",
                inner.GetType(), inner.Source);
    }
}
```

Parallel类的使用

- Parallel.Invoke(Action[] actions);　并行执行多个任务，直到完成
- Parallel.For(0, 100, i =>{...})
- Parallel.ForEach(list, item =>{...})

- 示例：并行计算矩阵乘法

并行Linq

- 并行Linq (即PLinq)
- 只要在集合上加个 .AsParallel()
- var a = (from n in persons.AsParallel()
- 　　where n.Age > 20 && n.Age < 25
- 　　select n)
- 　.ToList();

异步编程

异步编程

- 异步 asynchronize
- 主要解决的事情是
  - 等待一些耗时的任务（特别是文件、网络操作）而不阻塞当前任务
  - 异步编程提高响应能力（特别是UI)
- 开始一个任务后，让任务在另一个线程中执行，本线程可以继续执行别的事情，然后等待那个任务执行完毕

传统的方法1

- 使用委托的BeginInvoke及EndInvoke
- 如下
  - PrintDelegate printDelegate = Print;
  - IAsyncResult result= printDelegate.BeginInvoke("Hello World.", null, null);
  - Console.WriteLine("主线程继续执行...");
  - //当使用BeginInvoke异步调用方法时，如果方法未执行完，EndInvoke方法就会一直阻塞，直到被调用的方法执行完毕
  - int n = printDelegate.EndInvoke(result);

传统的方法2

- 使用回调
- 如：

```
Console.WriteLine("主线程.");
PrintDelegate printDelegate = Print;
    printDelegate.BeginInvoke("Hello world.", PrintComeplete, printDelegate);
Console.WriteLine("主线程继续执行...");

//回调方法要求        //1.返回类型为void          //2.只有一个参数IAsyncResult
    public static void PrintComeplete(IAsyncResult result)
    {
                (result.AsyncState as PrintDelegate).EndInvoke(result);
                Console.WriteLine("当前线程结束." + result.AsyncState.ToString());
    }
```

C#5.0的新方法

- C#5.0 (.net framework4.5，Visual Studio 2013)以上
- 新增await及async两个关键词
- await表示等待任务的执行
- async修饰一个方法，表示其中有await语句，

一般的写法

```
Task<double> FacAsync(int n) {   //用Task表示要执行任务
    return Task<double>.Run( ()=>{
        double s = 1; for(int i=1; i<n; i++) s = s*i;  return s;
    });
}
async void Test() {
    double result = await FacAsync(10); // 调用异步方法
    Console.WriteLine( result);  //异步方法执行完后才执行此句
}
```

```
double result = await FacAsync(10);        //此处会开新线程处理然后方法马上返回
//这之后的所有代码都会被封装成委托，在任务完成时调用
Console.WriteLine( result);
```

- 它解决了传统方法中“异步任务与回调方法分开写”的问题
- 相当于

```
System.Runtime.CompilerServices.TaskAwaiter<double> awaiter
    = FacAsync(10).GetAwaiter();
awaiter.OnCompleted(() =>
{
    double result = awaiter.GetResult();
    Console.WriteLine( result);
});
```

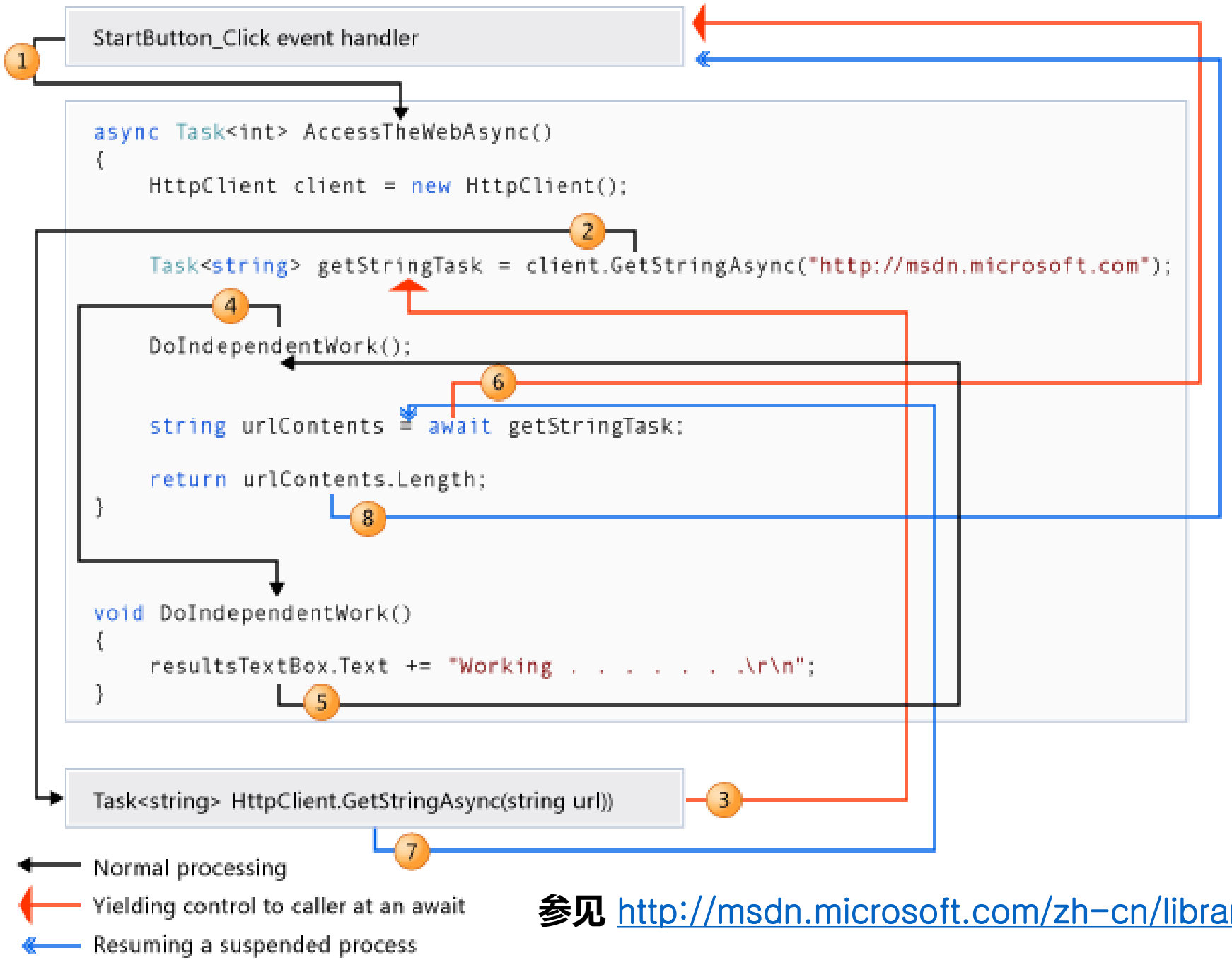
在WinForm中

- 当异步执行完成后，使用界面线程来执行回调，所以写起来更简洁

```
async Task<string> AccessTheWebAsync(string url)
        {
            HttpClient client = new HttpClient();
            Task<string> task = client.GetStringAsync(url); //异步
            DoIndependentWork(); //做其他事
            string urlContents = await task; //等待异步执行完毕
            return urlContents;
        }
```

```
async private void button1_Click(object sender, EventArgs e)
{
    string content = await AccessTheWebAsync(url);
    this.textBox2.Text = content;  //编译器让这句在界面线程上执行
}
```

- await后面的语句，就不用麻烦写成 Invoke(委托) 了



参见 http://msdn.microsoft.com/zh-cn/library/hh191443.aspx

异步的流

- 与上面的HttpClient相似，Stream等类也提供了异步方法
- 如

```
await myStream.WriteAsync(....)
```

- 这比传统的 BeginWrite() + 回调函数+EndWrite() 要方便很多
- 也可以这样：

```
Task task = myStream.WriteAsync(); //异步
DoIndependentWork(); //做其他事
 await task; //等待异步执行完毕
```

问题与讨论

dstang2000@263.net