

## 多线程简介

- 1、什么是进程？通俗的讲，一个 exe 运行一次就会产生一个进程，一个 exe 的多个进程之间数据互相隔离。
- 2、一个进程里至少有一个线程：主线程。我们平时写的控制台程序默认就是单线程的，代码从上往下执行，一行执行完了再执行下一行；
- 3、什么是多线程：一个人一边烧水一边洗衣服比“先烧水再洗衣服”效率高。同一时刻一个人只能干一件事情，其实是在“快速频繁切换”，如果处理不当可能比不用多线程效率还低。讨论多线程先只考虑“单核 cpu”。
- 4、普通的代码是从上向下执行的，但是多线程的代码可以“并行”执行，我们可以把“线程”理解成独立的执行单元，线程中的代码可以“并行执行”。线程根据情况被分配给一定的“时间片”来运行，可能一个线程还没执行完，就又要要把时间片交给别的线程执行。把要在单独的线程放到一个方法中，然后创建 Thread 对象，运行它，这个 Thread 中的代码就会在单独的线程中执行。代码（用成熟的 .Net Framework 开发）：

```
static void Main(string[] args)
{
    Thread thread = new Thread(Run1);
    thread.Start();

    while(true)
    {
        Console.WriteLine("主线程中"+DateTime.Now);
    }
}
static void Run1()
{
    while (true)
    {
        Console.WriteLine("子线程中" + DateTime.Now);
    }
}
```

Main 中的 for 循环和 Run1 中的代码“并行执行”。

- 5、多线程的好处：有很大可能增加系统的运行效率；开发 winform 程序，避免界面卡；注册后向用户发送欢迎邮件，如果发送邮件很慢的话，避免注册过程很慢。

## 线程细节

- 1、参数化 Thread。当然可以使用 lambda 表达式简化线程代码的编写。需要注意如果需要给子线程传递参数的话，不要把局部变量直接通过 lambda 表达式传递，否则：

```
int i = 5;
Thread thread = new Thread(() => {
    Console.WriteLine("i=" + i);
});

thread.Start();
i = 6;
```

包括如下的写法:

```
for(int i=0;i<10;i++)
{
    Thread thread = new Thread(() => {
        Console.WriteLine("i=" + i);
    });
    thread.Start();
}
```

改成用 ParameterizedThreadStart 就不容易弄错。

```
Thread thread = new Thread(Run1);
thread.Start("a");
static void Run1(object obj)
{
    Console.WriteLine(obj);
}
```

改造成 lambda 形式:

```
Thread thread = new Thread((obj)=> {
    Console.WriteLine(obj);
});
```

把有问题的代码改造如下:

代码 1:

```
int i = 5;
Thread thread = new Thread((obj) => {
    Console.WriteLine("i=" + obj);
});
```

thread.Start(i);

i = 6;

代码 2:

```
for (int i = 0; i < 10; i++)
{
    Thread thread = new Thread((item) => {
        Console.WriteLine("i=" + item);
    });
```

```
thread.Start(i);  
}
```

上面代码的执行结果不一定是按顺序的 0 到 9;

2、Thread.Sleep(n)当前代码所在的线程“睡眠 N 毫秒”

3、线程默认是“非后台线程”，一个程序必须所有“非后台线程”执行结束后程序才会退出，否则看起来好像是主线程退出了，其实进程还没有结束（搞一个 winform 程序演示）

把线程设置为“后台线程”后，所有“非后台线程”执行结束后程序就会退出，不会等“后台线程”执行结束：

```
thread.IsBackground = true;
```

简单的验证：控制台程序，如果不加 `t1.IsBackground = true;` 是线程执行结束才退出。加上的话就是“闪退”。

```
Thread t1 = new Thread(()=> {  
    Console.WriteLine("t1 要睡了");  
    Thread.Sleep(5000);  
    Console.WriteLine("t1 醒了");  
});  
t1.IsBackground = true;  
t1.Start();
```

4、线程优先级，默认优先级：

```
int i = 0, j = 0;  
Thread t1 = new Thread(()=> {  
    while(true)  
    {  
        i++;  
    }  
});  
t1.Start();  
Thread t2 = new Thread(() => {  
    while (true)  
    {  
        j++;  
    }  
});  
t2.Start();
```

```
Thread.Sleep(3000);  
Console.WriteLine("i="+i+",j="+j);  
Console.ReadKey();
```

再改一下优先级：

```
int i = 0, j = 0;
```

```
Thread t1 = new Thread(() => {
    while(true)
    {
        i++;
    }
});
t1.Priority = ThreadPriority.Highest;
t1.Start();
Thread t2 = new Thread(() => {
    while (true)
    {
        j++;
    }
});
t2.Priority = ThreadPriority.Lowest;
t2.Start();
```

```
Thread.Sleep(3000);
Console.WriteLine("i="+i+",j="+j);
Console.ReadKey();
```

5、线程的终止，可以调用 Thread 的 Abort 方法终止线程的执行，会在当前执行的代码上“无风起浪”的抛出 ThreadAbortException，可以 catch 一下验证一下，一般不需要程序去 catch。

## 线程同步初步

1、线程同步问题的提出，每次执行的结果可能都不一样：

```
class Program
{
    private static int counter = 0;

    static void Main(string[] args)
    {
        Thread t1 = new Thread(() => {
            for(int i=0;i<1000;i++)
            {
                counter++;
                Thread.Sleep(1);
            }
        });
        t1.Start();

        Thread t2 = new Thread(() => {
```

```
for (int i = 0; i < 1000; i++)
{
    counter++;
    Thread.Sleep(1);
}
});
t2.Start();

while (t1.IsAlive){} ;
while (t2.IsAlive) {} ;
Console.WriteLine(counter);

Console.ReadKey();
}
}
```

线程同步问题就是解决多个线程同时操作一个资源的问题

`while (t1.IsAlive)` ;操作会大量消耗 cpu 空转，可以改成 `t1.Join()` 就是让当前线程等待 t1 线程的结束。

## 2、lock

改用 lock 解决多个线程同时操作一个资源。lock 是 C# 中的关键字，他要锁定一个资源，lock 的特点是：同时只能有一个线程进入 lock 的对象的范围，其他 lock 的线程就要等。

```
class Program
{
    private static int counter = 0;
    private static object locker = new object();
    static void Main(string[] args)
    {
        Thread t1 = new Thread(() => {
            for(int i=0;i<1000;i++)
            {
                lock(locker)
                {
                    counter++;
                }

                Thread.Sleep(1);
            }
        });
        t1.Start();

        Thread t2 = new Thread(() => {
            for (int i = 0; i < 1000; i++)
```

```
{
    lock (locker)
    {
        counter++;
    }
    Thread.Sleep(1);
}
});
t2.Start();
t1.Join();
t2.Join();
Console.WriteLine(counter);

Console.ReadKey();
}
```

注意 lock 要锁定同一个对象，而且必须是引用类型的对象

## 线程其他操作

- 1、Interrupt 用于提前唤醒一个在 Sleep 的线程，Sleep 方法会抛出 `ThreadInterruptedException` 异常：

```
Thread t1 = new Thread(()=> {
    Console.WriteLine("t1要睡了");
    try
    {
        Thread.Sleep(5000);
    }
    catch(ThreadInterruptedException)
    {
        Console.WriteLine("擦，叫醒我干啥");
    }
    Console.WriteLine("t1醒了");
});
t1.Start();

Thread.Sleep(1000);
t1.Interrupt();
```

- 2、Sleep 是静态方法，只能是自己主动要求睡，别人不能命令他睡。  
3、已经过时的方法：Suspend、Resume。不要用  
4、Abort()方法会强行终止线程，会引发线程内当前在执行的代码发出 `ThreadAbortException`

### 异常

5、t1.Join()当前线程等待 t1 线程执行结束（Join 这里翻译成“连接”：你完了我再接着），例子：

```
Thread t1 = new Thread(() => {
    for(int i=0;i<100;i++)
    {
        Console.WriteLine("t1 "+i);
    }
});
t1.Start();
Thread t2 = new Thread(() => {
    t1.Join();//等着 t1 执行结束
    for (int i = 0; i < 100; i++)
    {
        Console.WriteLine("t2 " + i);
    }
});
t2.Start();
```

6、

线程不是越多越好，不是线程就是好的，举例子：繁忙时候的餐馆小二。具体多少合适要根据系统的配置、业务操作的不同而不同。

## 线程同步深入

同步问题的正式引出：

```
class Program
{
    static int money = 10000;

    static void QuQian(string name)
    {
        Console.WriteLine(name+"查看一下余额"+money);
        int yue = money - 1;
        Console.WriteLine(name + "取钱");
        money = yue;//故意这样写，写成 money--其实就没问题
        Console.WriteLine(name+"取完了，剩"+money);
    }

    static void Main(string[] args)
    {
        Thread t1 = new Thread(() => {
```

```
        for(int i=0;i<1000;i++)
        {
            QuQian("t1");
        }
    });
    Thread t2 = new Thread(() => {
        for (int i = 0; i < 1000; i++)
        {
            QuQian("t2");
        }
    });
    t1.Start();
    t2.Start();
    t1.Join();
    t2.Join();
    Console.WriteLine("余额"+money);
    Console.ReadKey();
}
}
```

解决思路：使用同步的技术避免两个线程同时修改一个余额。

解决方法 1：最大粒度——同步方法。

QuQian 方法上标注[[MethodImpl\(MethodImplOptions.Synchronized\)](#)]，这样一个方法只能同时被一个线程访问。

解决方法 2：对象互斥锁

`static object locker = new object();`

`static void QuQian(string name)`

```
{
    lock(locker)
    {
        Console.WriteLine(name + "查看一下余额" + money);
        int yue = money - 1;
        Console.WriteLine(name + "取钱");
        money = yue;//故意这样写，写成 money--其实就没问题
        Console.WriteLine(name + "取完了，剩" + money);
    }
}
```

同一时刻只能有一个线程进入同一个对象的 lock 代码块。必须是同一个对象才能起到互斥的作用。lock 后必须是引用类型，不一定是 object，只要是对象就行。

锁对象选择很重要，选不对起不到同步的作用；选不对可能会造成其他地方被锁，比如用字符串做锁（因为字符串拘留池导致可能用的是其他地方也在用的锁）

两个方法如果都用一个对象做锁，那么访问 A 的时候就不能访问 B，因此锁选择很重要。

解决方法 3（\*）：Monitor



其实 lock 关键字就是对 Monitor 的简化调用，lock 最终就编译成 Monitor，因此一般不直接用 Monitor 类：

```
static void QuQian(string name)
```

```
{
    Monitor.Enter(locker); //等待没有人锁定 locker 对象，我就锁定它，然后继续执行
    try
    {
        Console.WriteLine(name + "查看一下余额" + money);
        int yue = money - 1;
        Console.WriteLine(name + "取钱");
        money = yue; //故意这样写，写成 money--其实就没问题
        Console.WriteLine(name + "取完了，剩" + money);
    }
    finally
    {
        Monitor.Exit(locker); //释放 locker 对象的锁
    }
}
```

Monitor 有 TryEnter 方法，如果 Enter 的时候有人在占用锁，它不会等待，而是会返回 false。

```
static void F1(int i)
```

```
{
    if (!Monitor.TryEnter(locker))
    {
        Console.WriteLine("有人在锁着呢");
        return;
    }
    Console.WriteLine(i);
    Monitor.Exit(locker);
}
```

```
static void Main(string[] args)
```

```
{
    Thread t1 = new Thread(() => {
        for (int i = 0; i < 10000; i++)
        {
            F1(i);
        }
    });
    t1.Start();

    Thread t2 = new Thread(() => {
        for (int i = 0; i < 10000; i++)
        {
```

```
        F1(i);
    }
});
t2.Start();

Console.ReadKey();
}
```

## 单例模式与多线程

最简单的单例模式（饿汉），最简单，基本可以满足绝大部分要求，不装逼够用了。

```
class God
{
    private static God instance = new God();
    private God()
    {

    }

    public static God GetInstance()
    {
        return instance;
    }
}
```

有时候需要真正第一次用到的时候再创建那个唯一实例（懒汉）。

```
class God
{
    private static God instance = null;
    private God()
    {

    }

    public static God GetInstance()
    {
        if(instance == null)
        {
            instance = new God();
        }
        return instance;
    }
}
```

```
}  
}
```

这样写在多线程的环境下可能无法保证单例。

用 lock 可以保证

```
class God  
{  
    private static God instance = null;  
    private static object locker = new object();  
    private God()  
    {  
    }  
  
    public static God GetInstance()  
    {  
        lock(locker)  
        {  
            if (instance == null)  
            {  
                instance = new God();  
            }  
            return instance;  
        }  
    }  
}
```

但是每次其实只有 instance 为 null 的时候的那次加锁时候有意义的, 以后的千万次调用, 每个线程都要锁定 locker, 就会造成性能下降。如下改造, 进行双重检查 (double-check)

```
class God  
{  
    private static God instance = null;  
    private static object locker = new object();  
    private God()  
    {  
    }  
  
    public static God GetInstance()  
    {  
        if (instance == null)  
        {  
            lock (locker)  
            {  
                if (instance == null)  
                {  
                    instance = new God();  
                }  
            }  
        }  
    }  
}
```

```
    }  
    }  
    return instance;  
}  
}
```

## WaitHandle

除了锁之外，.Net 中还提供了一些线程间更自由通讯的工具，他们提供了通过“信号”进行通讯的机制，通俗的比喻为“开门”、“关门”：Set()开门，Reset()关门，WaitOne()等着开门。



比如：

```
ManualResetEvent mre = new ManualResetEvent(false);  
//构造函数 false 表示“初始状态为关门”，设置为 true 则初始化为开门状态  
Thread t1 = new Thread(()=> {  
    Console.WriteLine("开始等着开门");  
    mre.WaitOne();  
    Console.WriteLine("终于等到你");  
});  
t1.Start();  
  
Console.WriteLine("按任意键开门");  
Console.ReadKey();  
mre.Set();//开门
```

```
Console.ReadKey();
```

WaitOne()还可以设置等待超时时间:

```
ManualResetEvent mre = new ManualResetEvent(false);
```

```
//false 表示“初始状态为关门”
```

```
Thread t1 = new Thread(()=> {  
    Console.WriteLine("开始等着开门");  
    if(mre.WaitOne(5000))  
    {  
        Console.WriteLine("终于等到你");  
    }  
    else  
    {  
        Console.WriteLine("等了 5 秒钟都没等到");  
    }  
});  
t1.Start();
```

```
Console.WriteLine("按任意键开门");
```

```
Console.ReadKey();
```

```
mre.Set();//开门
```

```
Console.ReadKey();
```

[WaitHandle.WaitAll\(WaitHandle\[\] waitHandles\)](#)用来等待所有信号都变为“开门状态”，[WaitHandle.WaitAny\(WaitHandle\[\] waitHandles\)](#) 用来等待任意一个信号都变为“开门状态”。

**ManualResetEvent** 是一旦设定 **Set()**后就一直开门，除非调用 **Reset** 关门。**Manual**: 手动;  
**Reset**: 关门。

```
ManualResetEvent mre = new ManualResetEvent(false);
```

```
//false 表示“初始状态为关门”
```

```
Thread t1 = new Thread(()=> {  
    while(true)  
    {  
        Console.WriteLine("开始等着开门");  
        mre.WaitOne(5000);  
        Console.WriteLine("终于等到你");  
    }  
});  
t1.Start();
```

```
Console.WriteLine("按任意键开门");
```

```
Console.ReadKey();
```

```
mre.Set();//开门
```

```
Console.ReadKey();  
mre.Reset();//关门
```

```
Console.ReadKey();
```

还有一个类 `AutoResetEvent`，他是在开门并且一个 `WaitOne` 通过后自动关门，因此命名为“`AutoResetEvent`”（Auto 自动-Reset 关门）

```
AutoResetEvent are = new AutoResetEvent(false);  
Thread t1 = new Thread(() => {  
    while (true)  
    {  
        Console.WriteLine("开始等着开门");  
        are.WaitOne();  
        Console.WriteLine("终于等到你");  
    }  
});  
t1.Start();
```

```
Console.WriteLine("按任意键开门");  
Console.ReadKey();  
are.Set();//开门
```

```
Console.WriteLine("按任意键开门");  
Console.ReadKey();  
are.Set();
```

```
Console.WriteLine("按任意键开门");  
Console.ReadKey();  
are.Set();
```

```
Console.ReadKey();
```

`ManualResetEvent` 就是学校的大门，开门大家都可以进，除非主动关门；`AutoResetEvent` 就是火车地铁的闸机口，过了一个后自动关门。

## 线程池

1、线程池：因为每次创建线程、销毁线程都比较消耗 `cpu` 资源，因此可以通过线程池进行优化。线程池是一组已经创建好的线程，随用随取，用完了不是销毁线程，然后放到线程池中，供其他人用。

2、用线程池之后就无法对线程进行精细化的控制了（线程启停、优先级控制等）

3、ThreadPool 类的一个重要方法：

```
static bool QueueUserWorkItem(WaitCallback callBack)
```

```
static bool QueueUserWorkItem(WaitCallback callBack, object state)
```

第二个重载是用来传递一个参数给线程代码的。

4、除非要对线程进行精细化的控制，否则建议使用线程池，因为又简单、性能调优又更好。

## WinForm 编程中的多线程

使用 WebClient 获取一个网页然后显示到 WinForm 中，界面会卡。因为网络操作阻塞了主线程。

因此我们应该把下载的过程放到单独的线程中：

```
System.Threading.ThreadPool.QueueUserWorkItem((state)=> {  
    System.Net.WebClient wc = new System.Net.WebClient();  
    string s = wc.DownloadString("http://www.rupeng.com");  
    textBox1.Text = s;  
});
```

这样写的话，会在 `textBox1.Text = s;` 报异常，因为不能在其他线程中直接访问 UI 控件。需要把对 UI 控件的访问代码放到 `BeginInvoke()` 中，`BeginInvoke` 的委托中的代码是运行在 UI 线程中的

```
System.Threading.ThreadPool.QueueUserWorkItem((state)=> {  
    System.Net.WebClient wc = new System.Net.WebClient();  
    string s = wc.DownloadString("http://www.rupeng.com");  
    this.BeginInvoke(new Action(() => {  
        textBox1.Text = s;  
    }));  
});
```

但是也不能因此把 `DownloadString` 等也放到 `BeginInvoke` 中，否则又会界面卡死了。

在任意一个 UI 控件上都可以调用 `BeginInvoke` 方法

## 三种异步编程模型

.Net 中很多的类接口设计的时候都考虑了多线程问题，简化了多线程程序的开发。不用自己去写 `WaitHandler` 等这些底层的代码。由于历史的发展，这些类的接口设计有着三种不同的风格：`EAP(*)`、`APM(*)`和 `TPL`。目前重点用 `TPL`。

1、EAP

EAP 是 Event-based Asynchronous Pattern（基于事件的异步模型）的简写，类似于 Ajax 中的 `XmlHttpRequest`，`send` 之后并不是处理完成了，而是在 `onreadystatechange` 事件中再通知处理完成。

```
WebClient wc = new WebClient();
```

```
wc.DownloadStringCompleted += Wc_DownloadStringCompleted;
```

```
wc.DownloadStringAsync(new Uri("http://www.baidu.com"));
```

```
private void Wc_DownloadStringCompleted(object sender, DownloadStringCompletedEventArgs e)
{
    MessageBox.Show(e.Result);
}
```

优点是简单，缺点是当实现复杂的业务的时候很麻烦，比如下载 A 成功后再下载 b，如果下载 b 成功再下载 c，否则就下载 d。

EAP 的类的特点是：一个异步方法配一个\*\*\*Completed 事件。.Net 中基于 EAP 的类比较少。也有更好的替代品，因此了解即可。

## 2、APM

APM(Asynchronous Programming Model)是.Net 旧版本中广泛使用的异步编程模型。使用了 APM 的异步方法会返回一个 **IAsyncResult** 对象，这个对象有一个重要的属性 **AsyncWaitHandle**，他是一个用来等待异步任务执行结束的一个同步信号。

```
FileStream fs = File.OpenRead("d:/1.txt");
byte[] buffer = new byte[16];
IAsyncResult aResult = fs.BeginRead(buffer, 0, buffer.Length, null, null);
aResult.AsyncWaitHandle.WaitOne();//等待任务执行结束
MessageBox.Show(Encoding.UTF8.GetString(buffer));
fs.EndRead(aResult);
```

如果不加 **aResult.AsyncWaitHandle.WaitOne()** 那么很有可能打印出空白，因为 **BeginRead** 只是“开始读取”。调用完成一般要调用 **EndXXX** 来回收资源。

APM 的特点是：方法名字以 **BeginXXX** 开头，返回类型为 **IAsyncResult**，调用结束后需要 **EndXXX**。

.Net 中有如下的常用类支持 APM：Stream、SqlCommand、Socket 等。

APM 还是太复杂，了解即可。

## 3、TPL

TPL (Task Parallel Library) 是.Net 4.0 之后带来的新特性，更简洁，更方便。现在在.Net 平台下已经大面积使用。

```
FileStream fs = File.OpenRead("d:/1.txt");
byte[] buffer = new byte[16];
Task<int> task = fs.ReadAsync(buffer, 0, buffer.Length);
task.Wait();
MessageBox.Show("读取了"+task.Result+"个字节");
MessageBox.Show(Encoding.UTF8.GetString(buffer));
```

这样用和 APM 比起来的好处是：不需要 **EndXXX**。

精彩不仅于此：

```
private async void button1_Click(object sender, EventArgs e)
{
    FileStream fs = File.OpenRead("d:/1.txt");
    byte[] buffer = new byte[16];
    int len = await fs.ReadAsync(buffer, 0, buffer.Length);
    MessageBox.Show("读取了" + len + "个字节");
    MessageBox.Show(Encoding.UTF8.GetString(buffer));
}
```



注意方法中如果有 `await`，则方法必须标记为 `async`，不是所有方法都可以被轻松的标记为 `async`。WinForm 中的事件处理方法都可以标记为 `async`、MVC 中的 Action 方法也可以标记为 `async`、控制台的 Main 方法不能标记为 `async`。

TPL 的特点是：方法都以 XXXAsync 结尾，返回值类型是泛型的 `Task<T>`。

TPL 让我们可以用线性的方式去编写异步程序，不再需要像 EAP 中那样搞一堆回调、逻辑跳来跳去了。`await` 现在已经被 JavaScript 借鉴走了！

用 `await` 实现“先下载 A，如果下载的内容长度大于 100 则下载 B，否则下载 C”就很容易了  
再看看 WebClient 的 TPL 用法：

```
WebClient wc = new WebClient();  
string html = await wc.DownloadStringTaskAsync("http://www.rupeng.com");//不要丢了 await  
MessageBox.Show(html);
```

上面的代码并不是完全等价于

```
WebClient wc = new WebClient();  
var task = wc.DownloadStringTaskAsync("http://www.rupeng.com");  
task.Wait();  
MessageBox.Show(task.Result);
```

因为如果按照上面的写法，会卡死 UI 线程

而 `await` 则不会。。。好像不是？？？那只是因为把 `html` 这么长的字符串 `MessageBox.Show` 很慢，`MessageBox.Show(html.Substring(10))`就证明了这一点

`Task<T>`中的 `T` 是什么类型每个方法都不一样，要看文档。

WebClient、Stream、Socket 等这些“历史悠久”的类都同时提供了 APM、TPL 风格的 API，甚至有的还提供了 EAP 风格的 API。尽可能使用 TPL 风格的。

## 如何编写异步方法？

1、返回值为 `Task<T>`，潜规则（不要求）是方法名字以 `Async` 结尾：

```
static Task<string> F2Async()  
{  
    return Task.Run(() => {  
        System.Threading.Thread.Sleep(2000);  
        return "F2";  
    });  
}
```

## HttpClient

1、在程序调用 `Http` 接口、请求 `http` 资源、编写 `http` 爬虫等的时候都需要在程序集中进行 `Http` 请求。

很多人习惯的 `WebClient`、`HttpWebRequest` 在 TPL 下很多用起来不方便的地方，TPL 下推荐使用 `HttpClient`（`using System.Net.Http;`），.net core 下已经不支持 `WebClient` 等。

2、HttpClient 发出 Get 请求获取文本响应：`string html = await hc.GetStringAsync("http://www.rupeng.com");`

3、HttpClient 发出 Post 请求使用 `Task<HttpResponseMessage> PostAsync(string requestUri, HttpContent content)` 方法，第一个参数是请求的地址，第二个参数就是用来设置请求内容的。`HttpContent` 是抽象类，主要的子类有 `FormUrlEncodedContent`（表格式请求）、`StringContent`（字符串请求）、`MultipartFormDataContent`（Multipart 表单请求，一般带上传文件信息）、`StreamContent`（流内容）。使用提前写好的“HttpClient 用测试服务器端”部署到 IIS，然后方便测试。

a) 表格式请求，报文是“`userName=admin&password=123`”这样的格式

```
HttpClient client = new HttpClient();
Dictionary<string, string> keyValues = new Dictionary<string, string>();
keyValues["userName"] = "admin";
keyValues["password"] = "123";
FormUrlEncodedContent content = new FormUrlEncodedContent(keyValues);
var respMsg = await client.PostAsync("http://127.0.0.1:6666/Home/Login/", content);// 不要错误的调用了 PutAsync，应该是 PostAsync
```

```
string msgBody = await respMsg.Content.ReadAsStringAsync();
MessageBox.Show(respMsg.StatusCode.ToString());
MessageBox.Show(msgBody);
```

b) 普通字符串做报文

```
string json = "{userName:'admin',password:'123'}";
HttpClient client = new HttpClient();
StringContent content = new StringContent(json);
//contentype 必不可少
content.Headers.ContentType = MediaTypeHeaderValue.Parse("application/json");
var respMsg = await client.PostAsync("http://127.0.0.1:6666/Home/Login2/", content);
string msgBody = await respMsg.Content.ReadAsStringAsync();
MessageBox.Show(respMsg.StatusCode.ToString());
MessageBox.Show(msgBody);
```

c) 上传

```
HttpClient client = new HttpClient();
MultipartFormDataContent content = new MultipartFormDataContent();
content.Headers.Add("UserName", "admin");
content.Headers.Add("Password", "123");
using (Stream stream = File.OpenRead(@"D:\temp\logo 透明.png"))
{
    content.Add(new StreamContent(stream), "file", "logo.png");
    var respMsg = await client.PostAsync("http://127.0.0.1:6666/Home/Upload/", content);
    string msgBody = await respMsg.Content.ReadAsStringAsync();
    MessageBox.Show(respMsg.StatusCode.ToString());
    MessageBox.Show(msgBody);
}
```

## TPL 探秘

有如下异步方法

```
private async void button1_Click(object sender, EventArgs e)
{
    string i1 = await F1Async();
    MessageBox.Show("i1=" + i1);
    string i2 = await F2Async();

    MessageBox.Show("i2=" + i2);
}
```

```
static Task<string> F1Async()
{
    MessageBox.Show("F1 Start");
    return Task.Run(() => {
        System.Threading.Thread.Sleep(1000);
        MessageBox.Show("F1 Run");
        return "F1";
    });
}
```

```
static Task<string> F2Async()
{
    MessageBox.Show("F2 Start");
    return Task.Run(() => {
        System.Threading.Thread.Sleep(2000);
        MessageBox.Show("F2 Run");
        return "F2";
    });
}
```

`Task.Run()` 一个用来把一个代码段包装为 `Task<T>` 的方法 `Run` 中委托的代码体就是异步任务执行的逻辑，最后 `return` 返回值。

把 `button1_click` 改成：

```
private async void button1_Click(object sender, EventArgs e)
{
    Task<string> task1 = F1Async();
    Task<string> task2 = F2Async();
    string i1 = await task1;
    MessageBox.Show("i1=" + i1);
    string i2 = await task2;
```

```
MessageBox.Show("i2=" + i2);
```

```
}
```

因此不是只有 await 执行后 Task 才会真正执行。

使用 Reflector 反编译看看内部原理。主要在 MoveNext()方法内部。

```
static Task<string> GetRuPengAsync()
```

```
{
```

```
    这里可以return Task<string>类型的值，不能return “a”;
```

```
}
```

```
static async Task<string> GetRuPengAsync()
```

```
{
```

```
    这里不可以return Task<string>类型的值，而可以能return “a”;
```

```
}
```

- 1、只要方法是 Task<T>类型的返回值，都可以用 await 来等待调用获取返回值
- 2、如果一个返回 Task<T>类型的方法被标记了 async，那么只要方法内部直接 return T 这个类型的实例就可以。
- 3、一个返回 Task<T>类型的方法没有被标记了 async，那么需要方法内部直接 Task 实例。

```
private async void button2_Click(object sender, EventArgs e)
```

```
{
```

```
    //int i = await F1Async();
```

```
    int i = await F2Async();
```

```
    MessageBox.Show(i.ToString());
```

```
}
```

```
static Task<int> F1Async()
```

```
{
```

```
    return Task.Run(()=> {
```

```
        return 2;
```

```
    });
```

```
}
```

```
static async Task<int> F2Async()
```

```
{
```

```
    return 2;
```

```
}
```

## TPL 高级

- 1、如果方法内部有 await，则方法必须标记为 async。asp.net mvc 的 Action、WinForm 的事件处理函数都可以标记 async，控制台 Main 不能 async。对于不能标记为怎么办？

F1Async().Result 注意有的上下文下会有死锁。

2、如果返回值就是一个立即可以随手可得值，那么就用 `Task.FromResult()`

如果是一个需要休息一会的任务（比如下载失败则过 5 秒钟后重试。主线程不休息，和 `Thread.Sleep` 不一样），那么就用 `Task.Delay()`。

3、`Task.Factory.FromAsync()`把 `IAsyncResult` 转换为 `Task`，这样 APM 风格的 api 也可以用 `await` 来调用。

4、编写异步方法的简化写法。如果方法声明为 `async`，那么可以直接 `return` 具体的值，不再用创建 `Task`，由编译器创建 `Task`：

```
static async Task<int> F1Async()
```

```
{  
    return 1;  
}
```

```
static Task<int> F2Async()
```

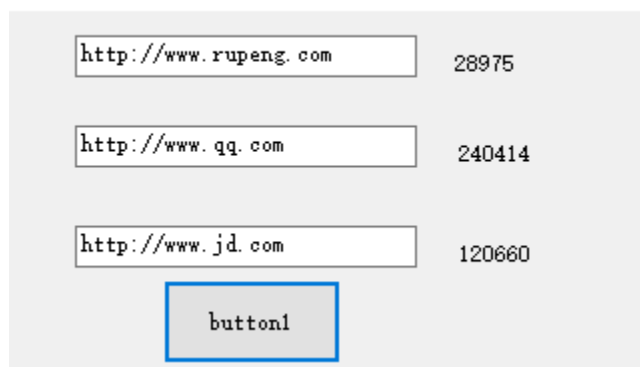
```
{  
    return Task.FromResult(3);  
}
```

```
static Task<int> F3Async()
```

```
{  
    return Task.Run(() => {  
        return 1 + 3;  
    });  
}
```

5、案例：WinForm 程序依次下载三个网址：

 Form1



<code>http://www.rupeng.com</code>	28975
<code>http://www.qq.com</code>	240414
<code>http://www.jd.com</code>	120660

button1

```
private async void button1_Click(object sender, EventArgs e)
```

```
{  
    HttpClient wc = new HttpClient();  
    string s1 = await wc.GetStringAsync(textBox1.Text);  
    label1.Text = s1.Length.ToString();  
    string s2 = await wc.GetStringAsync(textBox2.Text);  
    label2.Text = s2.Length.ToString();  
    string s3 = await wc.GetStringAsync(textBox3.Text);  
}
```

```
label3.Text = s3.Length.ToString();  
}
```

使用异步方法不一定是 `await`，只有需要按序执行采用 `await`。上面的程序改成并发下载，`Task.WaitAll` 是等待所有任务完成：

```
private async void button1_Click(object sender, EventArgs e)  
{  
    HttpClient hc = new HttpClient();  
    var task1 = hc.GetStringAsync(textBox1.Text);  
    var task2 = hc.GetStringAsync(textBox2.Text);  
    var task3 = hc.GetStringAsync(textBox3.Text);  
    Task.WaitAll(task1, task2, task3);  
    label1.Text = task1.Result.Length.ToString();  
    label2.Text = task2.Result.Length.ToString();  
    label3.Text = task3.Result.Length.ToString();  
}  
private async void button1_Click(object sender, EventArgs e)  
{  
    HttpClient wc = new HttpClient();  
    string s1 = await wc.GetStringAsync(textBox1.Text);  
    label1.Text = s1.Length.ToString();  
    string s2 = await wc.GetStringAsync(textBox2.Text);  
    label2.Text = s2.Length.ToString();  
    string s3 = await wc.GetStringAsync(textBox3.Text);  
    label3.Text = s3.Length.ToString();  
}
```

尽量用 `HttpClient`，不要用 `WebClient`

## 6、异步接口的声明

```
interface ITest
```

```
{  
    Task<int> GetAsync();//不要标注 async  
}
```

```
class Test : ITest
```

```
{  
    public async Task<int> GetAsync()  
    {  
        return 3;  
    }  
}
```

## TPL 异常处理

1、TPL 中，如果程序中出现异常，除非进行 try...catch，否则有可能是感觉不到出了异常的。测试，把上面下载程序的域名改成一个不存在的域名。

2、TPL 程序有时候还会抛出 [AggregateException](#)，这通常发生在并行有多个任务执行的情况下。比如：

```
private async void button1_Click(object sender, EventArgs e)
{
    try
    {
        HttpClient hc = new HttpClient();
        var task1 = hc.GetStringAsync(textBox1.Text);
        var task2 = hc.GetStringAsync(textBox2.Text);
        var task3 = hc.GetStringAsync(textBox3.Text);
        Task.WaitAll(task1, task2, task3);
        label1.Text = task1.Result.Length.ToString();
        label2.Text = task2.Result.Length.ToString();
        label3.Text = task3.Result.Length.ToString();
    }
    catch (AggregateException ae)
    {
        MessageBox.Show(ae.GetBaseException().ToString());
    }
}
```

因为多个并行的任务可能有多个有异常，因此会包装为 [AggregateException](#) 异常，[AggregateException](#) 的 [InnerExceptions](#) 属性可以获得多个异常对象信息。

## asp.net mvc 与 TPL

1、返回值改为 [Task<ActionResult>](#)即可，如果方法标记为 [async](#)，连自己创建 [Task](#) 都省了：  
[public async Task<ActionResult> Index\(\)](#)

```
{
    return View();
}
```

2、为什么 asp.net 中用 [async](#) 可以提升性能。

准确来讲，不是提升性能，不会提高访问速度，而是提升服务器的“吞吐量”，也就是可以处理的并发请求数。

用宿舍大妈安排接待男生访问女生宿舍的例子：不过不用异步，那么就是宿舍大妈要去

排队的男生那里找一个男生，带到女生房间，等他们会面万笔才能再去招呼下一个等位的。如果用了异步，大妈就是来管理男生排队了，把男女生撮合后，交给他们自己去处理，他再去招呼另外一个男生，这样就提升了吞吐量。但是单个男女的会面时间（性能）是不变的。

能用异步方法，就不要用同步方法，可以轻松的提升系统的吞吐量！

## TPL 常见误区

### 1、一定要 async 到底

一定要让 async 的传染性（调用异步方法要用 await，用了 await 方法就要声明为 async，调用我这个 async 方法的地方必须要 await……）不要轻易直接调用 Task 的 Wait、WaitAll 等方法。等待一个用 await，而不是 task.Wait(); 等待多个用 await Task.WhenAll(), 而不是 Task.WaitAll()

### 2、用了异步性能一定高？

这样的异步就没有意义：

```
public async Task<string> GetArticleContentByNoRightWayAsync()
{
    return await Task.Run(() =>
    {
        using (var client = new WebClient())
        {
            return client.DownloadString("http://www.rupeng.com");
        }
    });
}
```

用了异步之后就能用异步 api 就用异步 api，只要 await、WhenAll，不要 task.Result、Wait()、WaitAll()。ADO.Net、IO、EF 等都有异步的 API。

开发一个异步的增删改查，不分层了，使用 EF 的异步。做一个使用异步 IO 导出到 txt 的功能。

using System.Data.Entity 之后 First 之类的也有异步方法