**概述：**

1. **委托和事件在使用上基本一样，事件是委托的一种封装，第一个原因是为了避免让其它代码在类之外修改委托绑定的方法而对委托进行的封装（下文Part1），第二个原因是，委托可以在外部被调用，但事件只能在内部被调用（下文Part2）**
2. **委托有比较方便的泛型委托，可以直接使用或用来定义事件**
   * 泛型Action<T>委托表示引用一个void返回类型的方法。Action<T>委托类存在不同的变体，可以传递至多16种不同的参数类型，没有泛型参数的Action类可以调用没有参数的方法。例如：Action<in T1>调用带一个参数的方法，Action<in T1,in T2>调用带两个参数的方法等
   * Func<T>的用法和Action<T>用法类似，但是Func<T>表示引用一个带返回类型的方法，Func<T>也存在不同的变体，至多可以传递16个参数类型和1个返回类型，例如：Func<in T1,out Resout>表示带一个参数的方法，Func<in T1,in T2,out Resout>表示调用带两个参数的方法

**C#中的委托和事件 - Part.1**

2007-9-23 分类: C# 语言

**注意：**文中代码在VS2005下通过，由于VS2003(.Net Framework 1.1)不支持隐式的委托变量，所以如果在一个接受委托类型的位置直接赋予方法名，在VS2003下会报错，解决办法是显式的创建一个委托类型的实例(委托变量)。例如：委托类型 委托实例 = new 委托类型(方法名);

**1. 引言**

委托 和 事件在 .Net Framework中的应用非常广泛，然而，较好地理解委托和事件对很多接触C#时间不长的人来说并不容易。它们就像是一道槛儿，过了这个槛的人，觉得真是太容易了，而没有过去的人每次见到委托和事件就觉得心里别（biè）得慌，混身不自在。本文中，我将通过两个范例由浅入深地讲述什么是委托、为什么要使用委托、事件的由来、.Net Framework中的委托和事件、委托和事件对Observer设计模式的意义，对它们的中间代码也做了讨论。

**2. 将方法作为方法的参数**

我们先不管这个标题如何的绕口，也不管委托究竟是个什么东西，来看下面这两个最简单的方法，它们不过是在屏幕上输出一句问候的话语：

public void GreetPeople(string name) { // 做某些额外的事情，比如初始化之类，此处略 EnglishGreeting(name); } public void EnglishGreeting(string name) { Console.WriteLine("Morning, " + name); }

暂且不管这两个方法有没有什么实际意义。GreetPeople用于向某人问好，当我们传递代表某人姓名的name参数，比如说“Jimmy”，进去的时候，在这个方法中，将调用EnglishGreeting方法，再次传递name参数，EnglishGreeting则用于向屏幕输出 “Morning, Jimmy”。

现在假设这个程序需要进行全球化，哎呀，不好了，我是中国人，我不明白“Morning”是什么意思，怎么办呢？好吧，我们再加个中文版的问候方法：

public void ChineseGreeting(string name){ Console.WriteLine("早上好, " + name); }

这时候，GreetPeople也需要改一改了，不然如何判断到底用哪个版本的Greeting问候方法合适呢？在进行这个之前，我们最好再定义一个枚举作为判断的依据：

public enum Language{ English, Chinese } public void GreetPeople(string name, Language lang){ //做某些额外的事情，比如初始化之类，此处略 swith(lang){ case Language.English: EnglishGreeting(name); break; case Language.Chinese: ChineseGreeting(name); break; } }

OK，尽管这样解决了问题，但我不说大家也很容易想到，这个解决方案的可扩展性很差，如果日后我们需要再添加韩文版、日文版，就不得不反复修改枚举和GreetPeople()方法，以适应新的需求。

在考虑新的解决方案之前，我们先看看 GreetPeople的方法签名：

public void GreetPeople(string name, Language lang)

我们仅看 string name，在这里，string 是参数类型，name 是参数变量，当我们赋给name字符串“jimmy”时，它就代表“jimmy”这个值；当我们赋给它“张子阳”时，它又代表着“张子阳”这个值。然后，我们可以在方法体内对这个name进行其他操作。哎，这简直是废话么，刚学程序就知道了。

如果你再仔细想想，假如GreetPeople()方法可以接受一个参数变量，这个变量可以代表另一个方法，当我们给这个变量赋值 EnglishGreeting的时候，它代表着 EnglsihGreeting() 这个方法；当我们给它赋值ChineseGreeting 的时候，它又代表着ChineseGreeting()方法。我们将这个参数变量命名为 MakeGreeting，那么不是可以如同给name赋值时一样，在调用 GreetPeople()方法的时候，给这个MakeGreeting 参数也赋上值么(ChineseGreeting或者EnglsihGreeting等)？然后，我们在方法体内，也可以像使用别的参数一样使用MakeGreeting。但是，由于MakeGreeting代表着一个方法，它的使用方式应该和它被赋的方法(比如ChineseGreeting)是一样的，比如：

MakeGreeting(name);

好了，有了思路了，我们现在就来改改GreetPeople()方法，那么它应该是这个样子了：

public void GreetPeople(string name, \*\*\* MakeGreeting){ MakeGreeting(name); }

注意到 \*\*\* ，这个位置通常放置的应该是参数的类型，但到目前为止，我们仅仅是想到应该有个可以代表方法的参数，并按这个思路去改写GreetPeople方法，现在就出现了一个大问题：**这个代表着方法的MakeGreeting参数应该是什么类型的？**

**提示：**这里已不再需要枚举了，因为在给MakeGreeting赋值的时候动态地决定使用哪个方法，是ChineseGreeting还是 EnglishGreeting，而在这个两个方法内部，已经对使用“morning”还是“早上好”作了区分。

聪明的你应该已经想到了，现在是委托该出场的时候了，但讲述委托之前，我们再看看MakeGreeting参数所能代表的 ChineseGreeting()和EnglishGreeting()方法的签名：

public void EnglishGreeting(string name) public void ChineseGreeting(string name)

如同name可以接受String类型的“true”和“1”，但不能接受bool类型的true和int类型的1一样。**MakeGreeting的 参数类型定义 应该能够确定 MakeGreeting可以代表的方法种类，再进一步讲，就是MakeGreeting可以代表的方法 的 参数类型和返回类型。**

于是，委托出现了：**它定义了MakeGreeting参数所能代表的方法的种类，也就是MakeGreeting参数的类型。**

**提示：**如果上面这句话比较绕口，我把它翻译成这样：string 定义了name参数所能代表的**值的种类**，也就是name参数的类型。

本例中委托的定义：

public delegate void GreetingDelegate(string name);

可以与上面EnglishGreeting()方法的签名对比一下，除了加入了delegate关键字以外，其余的是不是完全一样？

现在，让我们再次改动GreetPeople()方法，如下所示：

public void GreetPeople(string name, GreetingDelegate MakeGreeting){ MakeGreeting(name); }

如你所见，委托GreetingDelegate出现的位置与 string相同，string是一个类型，那么GreetingDelegate应该也是一个类型，或者叫类(Class)。但是委托的声明方式和类却完全不同，这是怎么一回事？实际上，委托在编译的时候确实会编译成类。因为Delegate是一个类，所以在任何可以声明类的地方都可以声明委托。更多的内容将在下面讲述，现在，请看看这个范例的完整代码：

using System; using System.Collections.Generic; using System.Text; namespace Delegate { //定义委托，它定义了可以代表的方法的类型 public delegate void GreetingDelegate(string name); class Program { private static void EnglishGreeting(string name) { Console.WriteLine("Morning, " + name); } private static void ChineseGreeting(string name) { Console.WriteLine("早上好, " + name); } //注意此方法，它接受一个GreetingDelegate类型的方法作为参数 private static void GreetPeople(string name, GreetingDelegate MakeGreeting) { MakeGreeting(name); } static void Main(string[] args) { GreetPeople("Jimmy Zhang", EnglishGreeting); GreetPeople("张子阳", ChineseGreeting); Console.ReadKey(); } } } 输出如下： Morning, Jimmy Zhang 早上好, 张子阳

我们现在对委托做一个总结：

**委托是一个类，它定义了方法的类型，使得可以将方法当作另一个方法的参数来进行传递，这种将方法动态地赋给参数的做法，可以避免在程序中大量使用If-Else(Switch)语句，同时使得程序具有更好的可扩展性。**

**3. 将方法绑定到委托**

看到这里，是不是有那么点如梦初醒的感觉？于是，你是不是在想：在上面的例子中，我不一定要直接在GreetPeople()方法中给 name参数赋值，我可以像这样使用变量：

static void Main(string[] args) { string name1, name2; name1 = "Jimmy Zhang"; name2 = "张子阳"; GreetPeople(name1, EnglishGreeting); GreetPeople(name2, ChineseGreeting); Console.ReadKey(); }

而既然委托GreetingDelegate 和 类型 string 的地位一样，都是定义了一种参数类型，那么，我是不是也可以这么使用委托？

static void Main(string[] args) { GreetingDelegate delegate1, delegate2; delegate1 = EnglishGreeting; delegate2 = ChineseGreeting; GreetPeople("Jimmy Zhang", delegate1); GreetPeople("张子阳", delegate2); Console.ReadKey(); }

如你所料，这样是没有问题的，程序一如预料的那样输出。这里，我想说的是委托不同于string的一个特性：可以将多个方法赋给同一个委托，或者叫将多个方法绑定到同一个委托，当调用这个委托的时候，将依次调用其所绑定的方法。在这个例子中，语法如下：

static void Main(string[] args) { GreetingDelegate delegate1; delegate1 = EnglishGreeting; // 先给委托类型的变量赋值 delegate1 += ChineseGreeting; // 给此委托变量再绑定一个方法 // 将先后调用 EnglishGreeting 与 ChineseGreeting 方法 GreetPeople("Jimmy Zhang", delegate1); Console.ReadKey(); } 输出为： Morning, Jimmy Zhang 早上好, Jimmy Zhang

实际上，我们可以也可以绕过GreetPeople方法，通过委托来直接调用EnglishGreeting和ChineseGreeting：

static void Main(string[] args) { GreetingDelegate delegate1; delegate1 = EnglishGreeting; // 先给委托类型的变量赋值 delegate1 += ChineseGreeting; // 给此委托变量再绑定一个方法 // 将先后调用 EnglishGreeting 与 ChineseGreeting 方法 delegate1 ("Jimmy Zhang"); Console.ReadKey(); }

**注意：**这在本例中是没有问题的，但回头看下上面GreetPeople()的定义，在它之中可以做一些对于EnglshihGreeting和ChineseGreeting来说都需要进行的工作，为了简便我做了省略。

注意这里，第一次用的“=”，是赋值的语法；第二次，用的是“+=”，是绑定的语法。如果第一次就使用“+=”，将出现“使用了未赋值的局部变量”的编译错误。

我们也可以使用下面的代码来这样简化这一过程：

GreetingDelegate delegate1 = new GreetingDelegate(EnglishGreeting); delegate1 += ChineseGreeting; // 给此委托变量再绑定一个方法

看到这里，应该注意到，这段代码第一条语句与实例化一个类是何其的相似，你不禁想到：上面第一次绑定委托时不可以使用“+=”的编译错误，或许可以用这样的方法来避免：

GreetingDelegate delegate1 = new GreetingDelegate(); delegate1 += EnglishGreeting; // 这次用的是 “+=”，绑定语法。 delegate1 += ChineseGreeting; // 给此委托变量再绑定一个方法

但实际上，这样会出现编译错误： “GreetingDelegate”方法没有采用“0”个参数的重载。尽管这样的结果让我们觉得有点沮丧，但是编译的提示：“没有0个参数的重载”再次让我们联想到了类的构造函数。我知道你一定按捺不住想探个究竟，但再此之前，我们需要先把基础知识和应用介绍完。

**既然给委托可以绑定一个方法，那么也应该有办法取消对方法的绑定，很容易想到，这个语法是“-=”：**

static void Main(string[] args) { GreetingDelegate delegate1 = new GreetingDelegate(EnglishGreeting); delegate1 += ChineseGreeting; // 给此委托变量再绑定一个方法 // 将先后调用 EnglishGreeting 与 ChineseGreeting 方法 GreetPeople("Jimmy Zhang", delegate1); Console.WriteLine(); delegate1 -= EnglishGreeting; //取消对EnglishGreeting方法的绑定 // 将仅调用 ChineseGreeting GreetPeople("张子阳", delegate1); Console.ReadKey(); } 输出为： Morning, Jimmy Zhang 早上好, Jimmy Zhang 早上好, 张子阳

让我们再次对委托作个总结：

**使用委托可以将多个方法绑定到同一个委托变量，当调用此变量时(这里用“调用”这个词，是因为此变量代表一个方法)，可以依次调用所有绑定的方法。**

**4. 事件的由来**

我们继续思考上面的程序：上面的三个方法都定义在Programe类中，这样做是为了理解的方便，实际应用中，通常都是 GreetPeople 在一个类中，ChineseGreeting和 EnglishGreeting 在另外的类中。现在你已经对委托有了初步了解，是时候对上面的例子做个改进了。假设我们将GreetingPeople()放在一个叫GreetingManager的类中，那么新程序应该是这个样子的：

namespace Delegate { //定义委托，它定义了可以代表的方法的类型 public delegate void GreetingDelegate(string name); //新建的GreetingManager类 public class GreetingManager{ public void GreetPeople(string name, GreetingDelegate MakeGreeting) { MakeGreeting(name); } } class Program { private static void EnglishGreeting(string name) { Console.WriteLine("Morning, " + name); } private static void ChineseGreeting(string name) { Console.WriteLine("早上好, " + name); } static void Main(string[] args) { // ... ... } } }

这个时候，如果要实现前面演示的输出效果，Main方法我想应该是这样的：

static void Main(string[] args) { GreetingManager gm = new GreetingManager(); gm.GreetPeople("Jimmy Zhang", EnglishGreeting); gm.GreetPeople("张子阳", ChineseGreeting); }

我们运行这段代码，嗯，没有任何问题。程序一如预料地那样输出了：

Morning, Jimmy Zhang 早上好, 张子阳

现在，假设我们需要使用上一节学到的知识，将多个方法绑定到同一个委托变量，该如何做呢？让我们再次改写代码：

static void Main(string[] args) { GreetingManager gm = new GreetingManager(); GreetingDelegate delegate1; delegate1 = EnglishGreeting; delegate1 += ChineseGreeting; gm.GreetPeople("Jimmy Zhang", delegate1); } 输出： Morning, Jimmy Zhang 早上好, Jimmy Zhang

到了这里，我们不禁想到：面向对象设计，讲究的是对象的封装，既然可以声明委托类型的变量(在上例中是delegate1)，我们何不将这个变量封装到 GreetManager类中？在这个类的客户端中使用不是更方便么？于是，我们改写GreetManager类，像这样：

public class GreetingManager{ //在GreetingManager类的内部声明delegate1变量 public GreetingDelegate delegate1; public void GreetPeople(string name, GreetingDelegate MakeGreeting) { MakeGreeting(name); } }

现在，我们可以这样使用这个委托变量：

static void Main(string[] args) { GreetingManager gm = new GreetingManager(); gm.delegate1 = EnglishGreeting; gm.delegate1 += ChineseGreeting; gm.GreetPeople("Jimmy Zhang", gm.delegate1); } 输出为： Morning, Jimmy Zhang 早上好, Jimmy Zhang

尽管这样做没有任何问题，但我们发现这条语句很奇怪。在调用gm.GreetPeople方法的时候，再次传递了gm的delegate1字段：

gm.GreetPeople("Jimmy Zhang", gm.delegate1);

既然如此，我们何不修改 GreetingManager 类成这样：

public class GreetingManager{ //在GreetingManager类的内部声明delegate1变量 public GreetingDelegate delegate1; public void GreetPeople(string name) { if(delegate1!=null){ //如果有方法注册委托变量 delegate1(name); //通过委托调用方法 } } }

在客户端，调用看上去更简洁一些：

static void Main(string[] args) { GreetingManager gm = new GreetingManager(); gm.delegate1 = EnglishGreeting; gm.delegate1 += ChineseGreeting; gm.GreetPeople("Jimmy Zhang"); //注意，这次不需要再传递 delegate1变量 } 输出为： Morning, Jimmy Zhang 早上好, Jimmy Zhang

尽管这样达到了我们要的效果，但是还是存在着问题：

在这里，delegate1和我们平时用的string类型的变量没有什么分别，而我们知道，并不是所有的字段都应该声明成public，合适的做法是应该public的时候public，应该private的时候private。

我们先看看如果把 delegate1 声明为 private会怎样？结果就是：**这简直就是在搞笑。因为声明委托的目的就是为了把它暴露在类的客户端进行方法的注册，你把它声明为private了，客户端对它根本就不可见，那它还有什么用？**

再看看把delegate1 声明为 public 会怎样？结果就是：**在客户端可以对它进行随意的赋值等操作，严重破坏对象的封装性。**

最后，第一个方法注册用“=”，是赋值语法，因为要进行实例化，第二个方法注册则用的是“+=”。但是，不**管是赋值还是注册，都是将方法绑定到委托上，除了调用时先后顺序不同，再没有任何的分别，这样不是让人觉得很别扭么？**

现在我们想想，如果delegate1不是一个委托类型，而是一个string类型，你会怎么做？**答案是使用属性对字段进行封装。**

于是，Event出场了，它封装了委托类型的变量，使得：**在类的内部，不管你声明它是public还是protected，它总是private的。在类的外部，注册“+=”和注销“-=”的访问限定符与你在声明事件时使用的访问符相同。**

我们改写GreetingManager类，它变成了这个样子：

public class GreetingManager{ //这一次我们在这里声明一个事件 public event GreetingDelegate MakeGreet; public void GreetPeople(string name) { MakeGreet(name); } }

很容易注意到：MakeGreet 事件的声明与之前委托变量delegate1的声明唯一的区别是多了一个event关键字。看到这里，在结合上面的讲解，你应该明白到：**事件其实没什么不好理解的，声明一个事件不过类似于声明一个进行了封装的委托类型的变量而已。**

为了证明上面的推论，如果我们像下面这样改写Main方法：

static void Main(string[] args) { GreetingManager gm = new GreetingManager(); gm.MakeGreet = EnglishGreeting; // 编译错误1 gm.MakeGreet += ChineseGreeting; gm.GreetPeople("Jimmy Zhang"); }

会得到编译错误：事件“Delegate.GreetingManager.MakeGreet”只能出现在 += 或 -= 的左边(从类型“Delegate.GreetingManager”中使用时除外)。

**5. 事件和委托的编译代码**

这时候，我们注释掉编译错误的行，然后重新进行编译，再借助Reflactor来对 event的声明语句做一探究，看看为什么会发生这样的错误：

public event GreetingDelegate MakeGreet;



可以看到，实际上尽管我们在GreetingManager里将 MakeGreet 声明为public，但是，实际上MakeGreet会被编译成 私有字段，难怪会发生上面的编译错误了，因为它根本就不允许在GreetingManager类的外面以赋值的方式访问，从而验证了我们上面所做的推论。

我们再进一步看下MakeGreet所产生的代码：

private GreetingDelegate MakeGreet; //对事件的声明 实际是 声明一个私有的委托变量 [MethodImpl(MethodImplOptions.Synchronized)] public void add\_MakeGreet(GreetingDelegate value){ this.MakeGreet = (GreetingDelegate) Delegate.Combine(this.MakeGreet, value); } [MethodImpl(MethodImplOptions.Synchronized)] public void remove\_MakeGreet(GreetingDelegate value){ this.MakeGreet = (GreetingDelegate) Delegate.Remove(this.MakeGreet, value); }

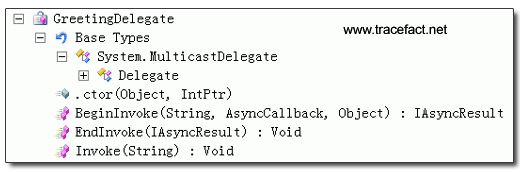
现在已经很明确了：**MakeGreet事件确实是一个GreetingDelegate类型的委托，只不过不管是不是声明为public，它总是被声明为private。另外，它还有两个方法，分别是add\_MakeGreet和remove\_MakeGreet，这两个方法分别用于注册委托类型的方法和取消注册。**实际上也就是： “+= ”对应 add\_MakeGreet，“-=”对应remove\_MakeGreet。而这两个方法的访问限制取决于声明事件时的访问限制符。

在add\_MakeGreet()方法内部，实际上调用了System.Delegate的Combine()静态方法，这个方法用于将当前的变量添加到委托链表中。我们前面提到过两次，说委托实际上是一个类，在我们定义委托的时候：

public delegate void GreetingDelegate(string name);

当编译器遇到这段代码的时候，会生成下面这样一个完整的类：

public sealed class GreetingDelegate:System.MulticastDelegate{ public GreetingDelegate(object @object, IntPtr method); public virtual IAsyncResult BeginInvoke(string name, AsyncCallback callback, object @object); public virtual void EndInvoke(IAsyncResult result); public virtual void Invoke(string name); }



关于这个类的更深入内容，可以参阅《CLR Via C#》等相关书籍，这里就不再讨论了。

**6. 委托、事件与Observer设计模式**

**6.1 范例说明**

上面的例子已不足以再进行下面的讲解了，我们来看一个新的范例，因为之前已经介绍了很多的内容，所以本节的进度会稍微快一些：

假设我们有个高档的热水器，我们给它通上电，当水温超过95度的时候：1、扬声器会开始发出语音，告诉你水的温度；2、液晶屏也会改变水温的显示，来提示水已经快烧开了。

现在我们需要写个程序来模拟这个烧水的过程，我们将定义一个类来代表热水器，我们管它叫：Heater，它有代表水温的字段，叫做temperature；当然，还有必不可少的给水加热方法BoilWater()，一个发出语音警报的方法MakeAlert()，一个显示水温的方法，ShowMsg()。

namespace Delegate { class Heater { private int temperature; // 水温 // 烧水 public void BoilWater() { for (int i = 0; i <= 100; i++) { temperature = i; if (temperature > 95) { MakeAlert(temperature); ShowMsg(temperature); } } } // 发出语音警报 private void MakeAlert(int param) { Console.WriteLine("Alarm：嘀嘀嘀，水已经 {0} 度了：" , param); } // 显示水温 private void ShowMsg(int param) { Console.WriteLine("Display：水快开了，当前温度：{0}度。" , param); } } class Program { static void Main() { Heater ht = new Heater(); ht.BoilWater(); } } }

**6.2 Observer设计模式简介**

上面的例子显然能完成我们之前描述的工作，但是却并不够好。现在假设热水器由三部分组成：热水器、警报器、显示器，它们来自于不同厂商并进行了组装。那么，应该是**热水器**仅仅负责烧水，它不能发出警报也不能显示水温；在水烧开时由**警报器**发出警报、**显示器**显示提示和水温。

这时候，上面的例子就应该变成这个样子：

// 热水器 public class Heater { private int temperature; // 烧水 private void BoilWater() { for (int i = 0; i <= 100; i++) { temperature = i; } } } // 警报器 public class Alarm{ private void MakeAlert(int param) { Console.WriteLine("Alarm：嘀嘀嘀，水已经 {0} 度了：" , param); } } // 显示器 public class Display{ private void ShowMsg(int param) { Console.WriteLine("Display：水已烧开，当前温度：{0}度。" , param); } }

这里就出现了一个问题：如何在水烧开的时候通知报警器和显示器？在继续进行之前，我们先了解一下Observer设计模式，Observer设计模式中主要包括如下两类对象：

1. Subject：监视对象，它往往包含着其他对象所感兴趣的内容。在本范例中，热水器就是一个监视对象，它包含的其他对象所感兴趣的内容，就是temprature字段，当这个字段的值快到100时，会不断把数据发给监视它的对象。
2. Observer：监视者，它监视Subject，当Subject中的某件事发生的时候，会告知Observer，而Observer则会采取相应的行动。在本范例中，Observer有警报器和显示器，它们采取的行动分别是发出警报和显示水温。

在本例中，事情发生的顺序应该是这样的：

1. 警报器和显示器告诉热水器，它对它的温度比较感兴趣(注册)。
2. 热水器知道后保留对警报器和显示器的引用。
3. 热水器进行烧水这一动作，当水温超过95度时，通过对警报器和显示器的引用，自动调用警报器的MakeAlert()方法、显示器的ShowMsg()方法。

类似这样的例子是很多的，GOF对它进行了抽象，称为Observer设计模式：**Observer设计模式是为了定义对象间的一种一对多的依赖关系，以便于当一个对象的状态改变时，其他依赖于它的对象会被自动告知并更新。Observer模式是一种松耦合的设计模式。**

**6.3 实现范例的Observer设计模式**

我们之前已经对委托和事件介绍很多了，现在写代码应该很容易了，现在在这里直接给出代码，并在注释中加以说明。

using System; using System.Collections.Generic; using System.Text; namespace Delegate { // 热水器 public class Heater { private int temperature; public delegate void BoilHandler(int param); //声明委托 public event BoilHandler BoilEvent; //声明事件 // 烧水 public void BoilWater() { for (int i = 0; i <= 100; i++) { temperature = i; if (temperature > 95) { if (BoilEvent != null) { //如果有对象注册 BoilEvent(temperature); //调用所有注册对象的方法 } } } } } // 警报器 public class Alarm { public void MakeAlert(int param) { Console.WriteLine("Alarm：嘀嘀嘀，水已经 {0} 度了：", param); } } // 显示器 public class Display { public static void ShowMsg(int param) { //静态方法 Console.WriteLine("Display：水快烧开了，当前温度：{0}度。", param); } } class Program { static void Main() { Heater heater = new Heater(); Alarm alarm = new Alarm(); heater.BoilEvent += alarm.MakeAlert; //注册方法 heater.BoilEvent += (new Alarm()).MakeAlert; //给匿名对象注册方法 heater.BoilEvent += Display.ShowMsg; //注册静态方法 heater.BoilWater(); //烧水，会自动调用注册过对象的方法 } } } 输出为： Alarm：嘀嘀嘀，水已经 96 度了： Alarm：嘀嘀嘀，水已经 96 度了： Display：水快烧开了，当前温度：96度。 // 省略...

**7. .Net Framework中的委托与事件**

尽管上面的范例很好地完成了我们想要完成的工作，但是我们不仅疑惑：为什么.Net Framework 中的事件模型和上面的不同？为什么有很多的EventArgs参数？

在回答上面的问题之前，我们先搞懂 .Net Framework的编码规范：

* 委托类型的名称都应该以EventHandler结束。
* 委托的原型定义：有一个void返回值，并接受两个输入参数：一个Object 类型，一个 EventArgs类型(或继承自EventArgs)。
* 事件的命名为 委托去掉 EventHandler之后剩余的部分。
* 继承自EventArgs的类型应该以EventArgs结尾。

再做一下说明：

1. 委托声明原型中的Object类型的参数代表了Subject，也就是监视对象，在本例中是 Heater(热水器)。回调函数(比如Alarm的MakeAlert)可以通过它访问触发事件的对象(Heater)。
2. EventArgs 对象包含了Observer所感兴趣的数据，在本例中是temperature。

**上面这些其实不仅仅是为了编码规范而已，这样也使得程序有更大的灵活性。**比如说，如果我们不光想获得热水器的温度，还想在Observer端(警报器或者显示器)方法中获得它的生产日期、型号、价格，那么委托和方法的声明都会变得很麻烦，而如果我们将热水器的引用传给警报器的方法，就可以在方法中直接访问热水器了。

现在我们改写之前的范例，让它符合 .Net Framework 的规范：

using System; using System.Collections.Generic; using System.Text; namespace Delegate { // 热水器 public class Heater { private int temperature; public string type = "RealFire 001"; // 添加型号作为演示 public string area = "China Xian"; // 添加产地作为演示 //声明委托 public delegate void BoiledEventHandler(Object sender, BoiledEventArgs e); public event BoiledEventHandler Boiled; //声明事件 // 定义BoiledEventArgs类，传递给Observer所感兴趣的信息 public class BoiledEventArgs : EventArgs { public readonly int temperature; public BoiledEventArgs(int temperature) { this.temperature = temperature; } } // 可以供继承自 Heater 的类重写，以便继承类拒绝其他对象对它的监视 protected virtual void OnBoiled(BoiledEventArgs e) { if (Boiled != null) { // 如果有对象注册 Boiled(this, e); // 调用所有注册对象的方法 } } // 烧水。 public void BoilWater() { for (int i = 0; i <= 100; i++) { temperature = i; if (temperature > 95) { //建立BoiledEventArgs 对象。 BoiledEventArgs e = new BoiledEventArgs(temperature); OnBoiled(e); // 调用 OnBolied方法 } } } } // 警报器 public class Alarm { public void MakeAlert(Object sender, Heater.BoiledEventArgs e) { Heater heater = (Heater)sender; //这里是不是很熟悉呢？ //访问 sender 中的公共字段 Console.WriteLine("Alarm：{0} - {1}: ", heater.area, heater.type); Console.WriteLine("Alarm: 嘀嘀嘀，水已经 {0} 度了：", e.temperature); Console.WriteLine(); } } // 显示器 public class Display { public static void ShowMsg(Object sender, Heater.BoiledEventArgs e) { //静态方法 Heater heater = (Heater)sender; Console.WriteLine("Display：{0} - {1}: ", heater.area, heater.type); Console.WriteLine("Display：水快烧开了，当前温度：{0}度。", e.temperature); Console.WriteLine(); } } class Program { static void Main() { Heater heater = new Heater(); Alarm alarm = new Alarm(); heater.Boiled += alarm.MakeAlert; //注册方法 heater.Boiled += (new Alarm()).MakeAlert; //给匿名对象注册方法 heater.Boiled += new Heater.BoiledEventHandler(alarm.MakeAlert); //也可以这么注册 heater.Boiled += Display.ShowMsg; //注册静态方法 heater.BoilWater(); //烧水，会自动调用注册过对象的方法 } } } 输出为： Alarm：China Xian - RealFire 001: Alarm: 嘀嘀嘀，水已经 96 度了： Alarm：China Xian - RealFire 001: Alarm: 嘀嘀嘀，水已经 96 度了： Alarm：China Xian - RealFire 001: Alarm: 嘀嘀嘀，水已经 96 度了： Display：China Xian - RealFire 001: Display：水快烧开了，当前温度：96度。 // 省略 ...

**8. 总结**

在本文中我首先通过一个GreetingPeople的小程序向大家介绍了委托的概念、委托用来做什么，随后又引出了事件，接着对委托与事件所产生的中间代码做了粗略的讲述。

在第二个稍微复杂点的热水器的范例中，我向大家简要介绍了 Observer设计模式，并通过实现这个范例完成了该模式，随后讲述了.Net Framework中委托、事件的实现方式。

感谢阅读，希望这篇文章能给你带来帮助！

**C#中的委托和事件 - Part.2**

## 引言

如果你看过了 [C#中的委托和事件](http://www.tracefact.com/CSharp-Programming/Delegates-and-Events-in-CSharp.aspx) 一文，我想你对委托和事件已经有了一个基本的认识。但那些远不是委托和事件的全部内容，还有很多的地方没有涉及。本文将讨论委托和事件一些更为细节的问题，包括一些大家常问到的问题，以及事件访问器、异常处理、超时处理和异步方法调用等内容。

## 为什么要使用事件而不是委托变量？

在 [C#中的委托和事件](http://www.tracefact.com/CSharp-Programming/Delegates-and-Events-in-CSharp.aspx) 中，我提出了两个为什么在类型中使用事件向外部提供方法注册，而不是直接使用委托变量的原因。主要是从封装性和易用性上去考虑，但是还漏掉了一点，**事件应该由事件发布者触发，而不应该由客户端（客户程序）来触发。**这句话是什么意思呢？请看下面的范例：

**NOTE：**注意这里术语的变化，当我们单独谈论事件，我们说发布者(publisher)、订阅者(subscriber)、客户端(client)。当我们讨论Observer模式，我们说主题(subject)和观察者(observer)。客户端通常是包含Main()方法的Program类。

class Program {  
    static void Main(string[] args) {  
        Publishser pub = new Publishser();  
        Subscriber sub = new Subscriber();  
          
        pub.NumberChanged += new NumberChangedEventHandler(sub.OnNumberChanged);  
        pub.DoSomething();          // 应该通过DoSomething()来触发事件  
        pub.NumberChanged(100);     // 但可以被这样直接调用，对委托变量的不恰当使用  
    }  
}  
  
// 定义委托  
public delegate void NumberChangedEventHandler(int count);  
  
// 定义事件发布者  
public class Publishser {  
    private int count;  
    public NumberChangedEventHandler NumberChanged;         // 声明委托变量  
    //public event NumberChangedEventHandler NumberChanged; // 声明一个事件  
  
    public void DoSomething() {  
        // 在这里完成一些工作 ...  
  
        if (NumberChanged != null) {    // 触发事件  
            count++;  
            NumberChanged(count);  
        }  
    }  
}  
  
// 定义事件订阅者  
public class Subscriber {  
    public void OnNumberChanged(int count) {  
        Console.WriteLine("Subscriber notified: count = {0}", count);  
    }  
}

上面代码定义了一个NumberChangedEventHandler委托，然后我们创建了事件的发布者Publisher和订阅者Subscriber。当使用委托变量时，客户端可以直接通过委托变量触发事件，也就是直接调用pub.NumberChanged(100)，这将会影响到所有注册了该委托的订阅者。而**事件的本意应该为在事件发布者在其本身的某个行为中触发，比如说在方法DoSomething()中满足某个条件后触发。**通过添加event关键字来发布事件，事件发布者的封装性会更好，**事件仅仅是供其他类型订阅，而客户端不能直接触发事件（语句pub.NumberChanged(100)无法通过编译），事件只能在事件发布者Publisher类的内部触发（比如在方法pub.DoSomething()中），换言之，就是NumberChanged(100)语句只能在Publisher内部被调用。**

大家可以尝试一下，将委托变量的声明那行代码注释掉，然后取消下面事件声明的注释。此时程序是无法编译的，当你使用了event关键字之后，直接在客户端触发事件这种行为，也就是直接调用pub.NumberChanged(100)，是被禁止的。事件只能通过调用DoSomething()来触发。这样才是事件的本意，事件发布者的封装才会更好。

就好像如果我们要定义一个数字类型，我们会使用int而不是使用object一样，给予对象过多的能力并不见得是一件好事，应该是越合适越好。尽管直接使用委托变量通常不会有什么问题，但它给了客户端不应具有的能力，而使用事件，可以限制这一能力，更精确地对类型进行封装。

**NOTE：**这里还有一个约定俗称的规定，就是订阅事件的方法的命名，通常为“On事件名”，比如这里的OnNumberChanged。

## 为什么委托定义的返回值通常都为void？

尽管并非必需，但是我们发现很多的委托定义返回值都为void，为什么呢？这是因为委托变量可以供多个订阅者注册，如果定义了返回值，那么多个订阅者的方法都会向发布者返回数值，结果就是后面一个返回的方法值将前面的返回值覆盖掉了，因此，实际上只能获得最后一个方法调用的返回值。可以运行下面的代码测试一下。除此以外，发布者和订阅者是松耦合的，发布者根本不关心谁订阅了它的事件、为什么要订阅，更别说订阅者的返回值了，所以返回订阅者的方法返回值大多数情况下根本没有必要。

class Program {  
    static void Main(string[] args) {  
        Publishser pub = new Publishser();  
        Subscriber1 sub1 = new Subscriber1();  
        Subscriber2 sub2 = new Subscriber2();  
        Subscriber3 sub3 = new Subscriber3();  
  
        pub.NumberChanged += new GeneralEventHandler(sub1.OnNumberChanged);  
        pub.NumberChanged += new GeneralEventHandler(sub2.OnNumberChanged);  
        pub.NumberChanged += new GeneralEventHandler(sub3.OnNumberChanged);  
        pub.DoSomething();          // 触发事件  
    }  
}  
  
// 定义委托  
public delegate string GeneralEventHandler();  
  
// 定义事件发布者  
public class Publishser {  
    public event GeneralEventHandler NumberChanged; // 声明一个事件  
    public void DoSomething() {  
        if (NumberChanged != null) {    // 触发事件  
            string rtn = NumberChanged();  
            Console.WriteLine(rtn);     // 打印返回的字符串，输出为Subscriber3  
        }  
    }  
}  
  
// 定义事件订阅者  
public class Subscriber1 {    
    public string OnNumberChanged() {  
        return "Subscriber1";  
    }  
}  
public class Subscriber2 { /\* 略，与上类似，返回Subscriber2\*/ }  
public class Subscriber3 { /\* 略，与上类似，返回Subscriber3\*/ }

如果运行这段代码，得到的输出是Subscriber3，可以看到，只得到了最后一个注册方法的返回值。

## 如何让事件只允许一个客户订阅？

少数情况下，比如像上面，为了避免发生“值覆盖”的情况（更多是在异步调用方法时，后面会讨论），我们可能想限制只允许一个客户端注册。此时怎么做呢？我们可以向下面这样，将事件声明为private的，然后提供两个方法来进行注册和取消注册：

// 定义事件发布者  
public class Publishser {  
    private event GeneralEventHandler NumberChanged;    // 声明一个私有事件  
    // 注册事件  
    public void Register(GeneralEventHandler method) {  
        NumberChanged = method;  
    }  
    // 取消注册  
    public void UnRegister(GeneralEventHandler method) {  
        NumberChanged -= method;  
    }  
  
    public void DoSomething() {  
        // 做某些其余的事情  
        if (NumberChanged != null) {    // 触发事件  
            string rtn = NumberChanged();  
            Console.WriteLine("Return: {0}", rtn);      // 打印返回的字符串，输出为Subscriber3  
        }  
    }  
}

**NOTE：**注意上面，在UnRegister()中，没有进行任何判断就使用了NumberChanged-=method语句。这是因为即使method方法没有进行过注册，此行语句也不会有任何问题，不会抛出异常，仅仅是不会产生任何效果而已。

注意在Register()方法中，我们使用了赋值操作符“=”，而非“+=”，通过这种方式就避免了多个方法注册。上面的代码尽管可以完成我们的需要，但是此时大家还应该注意下面两点：

1、将NumberChanged声明为委托变量还是事件都无所谓了，因为它是私有的，即便将它声明为一个委托变量，客户端也看不到它，也就无法通过它来触发事件、调用订阅者的方法。而只能通过Register()和UnRegister()方法来注册和取消注册，通过调用DoSomething()方法触发事件（而不是NumberChanged本身，这在前面已经讨论过了）。

2、我们还应该发现，这里采用的、对NumberChanged委托变量的访问模式和C#中的属性是多么类似啊？大家知道，在C#中通常一个属性对应一个类型成员，而在类型的外部对成员的操作全部通过属性来完成。尽管这里对委托变量的处理是类似的效果，但却使用了两个方法来进行模拟，有没有办法像使用属性一样来完成上面的例子呢？答案是有的，C#中提供了一种叫事件访问器（Event Accessor）的东西，它用来封装委托变量。如下面例子所示：

class Program {  
    static void Main(string[] args) {  
        Publishser pub = new Publishser();  
        Subscriber1 sub1 = new Subscriber1();  
        Subscriber2 sub2 = new Subscriber2();  
  
        pub.NumberChanged -= sub1.OnNumberChanged;  // 不会有任何反应  
        pub.NumberChanged += sub2.OnNumberChanged;  // 注册了sub2  
        pub.NumberChanged += sub1.OnNumberChanged;  // sub1将sub2的覆盖掉了  
          
        pub.DoSomething();          // 触发事件  
    }  
}  
  
// 定义委托  
public delegate string GeneralEventHandler();  
  
// 定义事件发布者  
public class Publishser {  
    // 声明一个委托变量  
    private GeneralEventHandler numberChanged;  
    // 事件访问器的定义  
    public event GeneralEventHandler NumberChanged {  
        add {  
            numberChanged = value;  
        }  
        remove {  
            numberChanged -= value;  
        }  
    }  
      
    public void DoSomething() {  
        // 做某些其他的事情  
        if (numberChanged != null) {    // 通过委托变量触发事件  
            string rtn = numberChanged();  
            Console.WriteLine("Return: {0}", rtn);      // 打印返回的字符串  
        }  
    }  
}  
  
// 定义事件订阅者  
public class Subscriber1 {  
    public string OnNumberChanged() {  
        Console.WriteLine("Subscriber1 Invoked!");  
        return "Subscriber1";  
    }  
}  
public class Subscriber2 {/\* 与上类同，略 \*/}  
public class Subscriber3 {/\* 与上类同，略 \*/}

上面代码中类似属性的public event GeneralEventHandler NumberChanged {add{...}remove{...}}语句便是事件访问器。使用了事件访问器以后，在DoSomething方法中便只能通过numberChanged委托变量来触发事件，而不能NumberChanged事件访问器（注意它们的大小写不同）触发，它只用于注册和取消注册。下面是代码输出：

Subscriber1 Invoked!  
Return: Subscriber1

## 获得多个返回值与异常处理

现在假设我们想要获得多个订阅者的返回值，以List<string>的形式返回，该如何做呢？我们应该记得委托定义在编译时会生成一个继承自MulticastDelegate的类，而这个MulticastDelegate又继承自Delegate，在Delegate内部，维护了一个委托链表，链表上的每一个元素，为一个只包含一个目标方法的委托对象。而通过Delegate基类的GetInvocationList()静态方法，可以获得这个委托链表。随后我们遍历这个链表，通过链表中的每个委托对象来调用方法，这样就可以分别获得每个方法的返回值：

class Program4 {  
    static void Main(string[] args) {  
        Publishser pub = new Publishser();  
        Subscriber1 sub1 = new Subscriber1();  
        Subscriber2 sub2 = new Subscriber2();  
        Subscriber3 sub3 = new Subscriber3();  
  
        pub.NumberChanged += new DemoEventHandler(sub1.OnNumberChanged);  
        pub.NumberChanged += new DemoEventHandler(sub2.OnNumberChanged);  
        pub.NumberChanged += new DemoEventHandler(sub3.OnNumberChanged);  
  
        List<string> list = pub.DoSomething();  //调用方法，在方法内触发事件  
  
        foreach (string str in list) {  
            Console.WriteLine(str);  
        }             
    }  
}  
  
public delegate string DemoEventHandler(int num);  
  
// 定义事件发布者  
public class Publishser {  
    public event DemoEventHandler NumberChanged;    // 声明一个事件  
  
    public List<string> DoSomething() {  
        // 做某些其他的事  
  
        List<string> strList = new List<string>();  
        if (NumberChanged == null) return strList;  
  
        // 获得委托数组  
        Delegate[] delArray = NumberChanged.GetInvocationList();  
  
        foreach (Delegate del in delArray) {  
            // 进行一个向下转换  
            DemoEventHandler method = (DemoEventHandler)del;  
            strList.Add(method(100));       // 调用方法并获取返回值  
        }  
          
        return strList;  
    }  
}  
  
// 定义事件订阅者  
public class Subscriber1 {  
    public string OnNumberChanged(int num) {  
        Console.WriteLine("Subscriber1 invoked, number:{0}", num);  
        return "[Subscriber1 returned]";  
    }  
}  
public class Subscriber3 {与上面类同，略}  
public class Subscriber3 {与上面类同，略}

如果运行上面的代码，可以得到这样的输出：

Subscriber1 invoked, number:100  
Subscriber2 invoked, number:100  
Subscriber3 invoked, number:100  
[Subscriber1 returned]  
[Subscriber2 returned]  
[Subscriber3 returned]

可见我们获得了三个方法的返回值。而我们前面说过，很多情况下委托的定义都不包含返回值，所以上面介绍的方法似乎没有什么实际意义。其实通过这种方式来触发事件最常见的情况应该是在异常处理中，因为很有可能在触发事件时，订阅者的方法会抛出异常，而这一异常会直接影响到发布者，使得发布者程序中止，而后面订阅者的方法将不会被执行。因此我们需要加上异常处理，考虑下面一段程序：

class Program5 {  
    static void Main(string[] args) {  
        Publisher pub = new Publisher();  
        Subscriber1 sub1 = new Subscriber1();  
        Subscriber2 sub2 = new Subscriber2();  
        Subscriber3 sub3 = new Subscriber3();  
  
        pub.NumberChanged += new DemoEventHandler(sub1.OnNumberChanged);  
        pub.NumberChanged += new DemoEventHandler(sub2.OnNumberChanged);  
        pub.NumberChanged += new DemoEventHandler(sub3.OnNumberChanged);  
    }  
}  
  
public class Publisher {  
    public event EventHandler MyEvent;  
    public void DoSomething() {  
        // 做某些其他的事情  
        if (MyEvent != null) {  
            try {  
                MyEvent(this, EventArgs.Empty);  
            } catch (Exception e) {  
                Console.WriteLine("Exception: {0}", e.Message);  
            }  
        }  
    }  
}  
  
public class Subscriber1 {  
    public void OnEvent(object sender, EventArgs e) {  
        Console.WriteLine("Subscriber1 Invoked!");  
    }  
}  
  
public class Subscriber2 {  
    public void OnEvent(object sender, EventArgs e) {  
        throw new Exception("Subscriber2 Failed");  
    }  
}  
public class Subscriber3 {/\* 与Subsciber1类同，略\*/}

注意到我们在Subscriber2中抛出了异常，同时我们在Publisher中使用了try/catch语句来处理异常。运行上面的代码，我们得到的结果是：

Subscriber1 Invoked!  
Exception: Subscriber2 Failed

可以看到，**尽管我们捕获了异常，使得程序没有异常结束，但是却影响到了后面的订阅者，因为Subscriber3也订阅了事件，但是却没有收到事件通知（它的方法没有被调用）。**此时，我们可以采用上面的办法，先获得委托链表，然后在遍历链表的循环中处理异常，我们只需要修改一下DoSomething方法就可以了：

public void DoSomething() {  
    if (MyEvent != null) {  
        Delegate[] delArray = MyEvent.GetInvocationList();  
        foreach (Delegate del in delArray) {  
            EventHandler method = (EventHandler)del;    // 强制转换为具体的委托类型  
            try {  
                method(this, EventArgs.Empty);  
            } catch (Exception e) {  
                Console.WriteLine("Exception: {0}", e.Message);  
            }  
        }  
    }  
}

注意到Delegate是EventHandler的基类，所以为了触发事件，先要进行一个向下的强制转换，之后才能在其上触发事件，调用所有注册对象的方法。除了使用这种方式以外，还有一种更灵活方式可以调用方法，它是定义在Delegate基类中的DynamicInvoke()方法：

public object DynamicInvoke(params object[] args);

这可能是调用委托最通用的方法了，适用于所有类型的委托。它接受的参数为object[]，也就是说它可以将任意数量的任意类型作为参数，并返回单个object对象。上面的DoSomething()方法也可以改写成下面这种通用形式：

public void DoSomething() {  
    // 做某些其他的事情  
    if (MyEvent != null) {  
        Delegate[] delArray = MyEvent.GetInvocationList();  
        foreach (Delegate del in delArray) {                      
            try {  
                // 使用DynamicInvoke方法触发事件  
                del.DynamicInvoke(this, EventArgs.Empty);     
            } catch (Exception e) {  
                Console.WriteLine("Exception: {0}", e.Message);  
            }  
        }  
    }  
}

注意现在在DoSomething()方法中，我们取消了向具体委托类型的向下转换，现在没有了任何的基于特定委托类型的代码，而DynamicInvoke又可以接受任何类型的参数，且返回一个object对象。所以我们完全可以将DoSomething()方法抽象出来，使它成为一个公共方法，然后供其他类来调用，我们将这个方法声明为静态的，然后定义在Program类中：

// 触发某个事件，以列表形式返回所有方法的返回值  
public static object[] FireEvent(Delegate del, params object[] args){  
  
    List<object> objList = new List<object>();  
  
    if (del != null) {  
        Delegate[] delArray = del.GetInvocationList();  
        foreach (Delegate method in delArray) {  
            try {  
                // 使用DynamicInvoke方法触发事件  
                object obj = method.DynamicInvoke(args);  
                if (obj != null)  
                    objList.Add(obj);  
            } catch { }  
        }  
    }  
    return objList.ToArray();  
}

随后，我们在DoSomething()中只要简单的调用一下这个方法就可以了：

public void DoSomething() {  
    // 做某些其他的事情  
    Program5.FireEvent(MyEvent, this, EventArgs.Empty);  
}

注意FireEvent()方法还可以返回一个object[]数组，这个数组包括了所有订阅者方法的返回值。而在上面的例子中，我没有演示如何获取并使用这个数组，为了节省篇幅，这里也不再赘述了，在本文附带的代码中，有关于这部分的演示，有兴趣的朋友可以下载下来看看。

## 委托中订阅者方法超时的处理

订阅者除了可以通过异常的方式来影响发布者以外，还可以通过另一种方式：超时。一般说超时，指的是方法的执行超过某个指定的时间，而这里我将含义扩展了一下，凡是方法执行的时间比较长，我就认为它超时了，这个“比较长”是一个比较模糊的概念，2秒、3秒、5秒都可以视为超时。超时和异常的区别就是超时并不会影响事件的正确触发和程序的正常运行，却会导致事件触发后需要很长才能够结束。在依次执行订阅者的方法这段期间内，客户端程序会被中断，什么也不能做。因为当执行订阅者方法时（通过委托，相当于依次调用所有注册了的方法），当前线程会转去执行方法中的代码，调用方法的客户端会被中断，只有当方法执行完毕并返回时，控制权才会回到客户端，从而继续执行下面的代码。我们来看一下下面一个例子：

class Program6 {  
    static void Main(string[] args) {  
  
        Publisher pub = new Publisher();  
        Subscriber1 sub1 = new Subscriber1();  
        Subscriber2 sub2 = new Subscriber2();  
        Subscriber3 sub3 = new Subscriber3();  
  
        pub.MyEvent += new EventHandler(sub1.OnEvent);  
        pub.MyEvent += new EventHandler(sub2.OnEvent);  
        pub.MyEvent += new EventHandler(sub3.OnEvent);  
  
        pub.DoSomething();      // 触发事件  
  
        Console.WriteLine("\nControl back to client!"); // 返回控制权  
    }  
  
    // 触发某个事件，以列表形式返回所有方法的返回值  
    public static object[] FireEvent(Delegate del, params object[] args) {  
        // 代码与上同，略  
    }  
}  
  
public class Publisher {  
    public event EventHandler MyEvent;  
    public void DoSomething() {  
        // 做某些其他的事情  
        Console.WriteLine("DoSomething invoked!");  
        Program6.FireEvent(MyEvent, this, EventArgs.Empty); //触发事件  
    }  
}  
  
public class Subscriber1 {  
    public void OnEvent(object sender, EventArgs e) {  
        Thread.Sleep(TimeSpan.FromSeconds(3));  
        Console.WriteLine("Waited for 3 seconds, subscriber1 invoked!");  
    }  
}  
public class Subscriber2 {  
    public void OnEvent(object sender, EventArgs e) {  
        Console.WriteLine("Subscriber2 immediately Invoked!");  
    }  
}  
public class Subscriber3 {  
    public void OnEvent(object sender, EventArgs e) {  
        Thread.Sleep(TimeSpan.FromSeconds(2));  
        Console.WriteLine("Waited for 2 seconds, subscriber2 invoked!");  
    }  
}

在这段代码中，我们使用Thread.Sleep()静态方法模拟了方法超时的情况。其中Subscriber1.OnEvent()需要三秒钟完成，Subscriber2.OnEvent()立即执行，Subscriber3.OnEvent需要两秒完成。这段代码完全可以正常输出，也没有异常抛出（如果有，也仅仅是该订阅者被忽略掉），下面是输出的情况：

DoSomething invoked!  
Waited for 3 seconds, subscriber1 invoked!  
Subscriber2 immediately Invoked!  
Waited for 2 seconds, subscriber2 invoked!  
  
Control back to client!

但是这段程序在调用方法DoSomething()、打印了“DoSomething invoked”之后，触发了事件，随后必须等订阅者的三个方法全部执行完毕了之后，也就是大概5秒钟的时间，才能继续执行下面的语句，也就是打印“Control back to client”。而我们前面说过，很多情况下，尤其是远程调用的时候（比如说在Remoting中），发布者和订阅者应该是完全的松耦合，发布者不关心谁订阅了它、不关心订阅者的方法有什么返回值、不关心订阅者会不会抛出异常，当然也不关心订阅者需要多长时间才能完成订阅的方法，它只要在事件发生的那一瞬间告知订阅者事件已经发生并将相关参数传给订阅者就可以了。然后它就应该继续执行它后面的动作，在本例中就是打印“Control back to client！”。**而订阅者不管失败或是超时都不应该影响到发布者，但在上面的例子中，发布者却不得不等待订阅者的方法执行完毕才能继续运行。**

现在我们来看下如何解决这个问题，先回顾一下之前我在C#中的委托和事件一文中提到的内容，我说过，委托的定义会生成继承自MulticastDelegate的完整的类，其中包含Invoke()、BeginInvoke()和EndInvoke()方法。当我们直接调用委托时，实际上是调用了Invoke()方法，它会中断调用它的客户端，然后在客户端线程上执行所有订阅者的方法（客户端无法继续执行后面代码），最后将控制权返回客户端。注意到BeginInvoke()、EndInvoke()方法，在.Net中，异步执行的方法通常都会配对出现，并且以Begin和End作为方法的开头（最常见的可能就是Stream类的BeginRead()和EndRead()方法了）。它们用于方法的异步执行，即是在调用BeginInvoke()之后，客户端从线程池中抓取一个闲置线程，然后交由这个线程去执行订阅者的方法，而客户端线程则可以继续执行下面的代码。

BeginInvoke()接受“动态”的参数个数和类型，为什么说“动态”的呢？因为它的参数是在编译时根据委托的定义动态生成的，其中前面参数的个数和类型与委托定义中接受的参数个数和类型相同，最后两个参数分别是AsyncCallback和Object类型，对于它们更具体的内容，可以参见下一节委托和方法的异步调用部分。现在，我们仅需要对这两个参数传入null就可以了。另外还需要注意几点：

* 在委托类型上调用BeginInvoke()时，此委托对象只能包含一个目标方法，所以对于多个订阅者注册的情况，必须使用GetInvocationList()获得所有委托对象，然后遍历它们，分别在其上调用BeginInvoke()方法。如果直接在委托上调用BeginInvoke()，会抛出异常，提示“委托只能包含一个目标方法”。
* 如果订阅者的方法抛出异常，.NET会捕捉到它，但是只有在调用EndInvoke()的时候，才会将异常重新抛出。而在本例中，我们不使用EndInvoke()（因为我们不关心订阅者的执行情况），所以我们无需处理异常，因为即使抛出异常，也是在另一个线程上，不会影响到客户端线程（客户端甚至不知道订阅者发生了异常，这有时是好事有时是坏事）。
* BeginInvoke()方法属于委托定义所生成的类，它既不属于MulticastDelegate也不属于Delegate基类，所以无法继续使用可重用的FireEvent()方法，我们需要进行一个向下转换，来获取到实际的委托类型。

现在我们修改一下上面的程序，使用异步调用来解决订阅者方法执行超时的情况：

class Program6 {  
    static void Main(string[] args) {  
  
        Publisher pub = new Publisher();  
        Subscriber1 sub1 = new Subscriber1();  
        Subscriber2 sub2 = new Subscriber2();  
        Subscriber3 sub3 = new Subscriber3();  
  
        pub.MyEvent += new EventHandler(sub1.OnEvent);  
        pub.MyEvent += new EventHandler(sub2.OnEvent);  
        pub.MyEvent += new EventHandler(sub3.OnEvent);  
  
        pub.DoSomething();      // 触发事件  
  
        Console.WriteLine("Control back to client!\n"); // 返回控制权  
        Console.WriteLine("Press any thing to exit...");  
        Console.ReadKey();      // 暂停客户程序，提供时间供订阅者完成方法  
    }  
}  
  
public class Publisher {  
    public event EventHandler MyEvent;  
    public void DoSomething() {           
        // 做某些其他的事情  
        Console.WriteLine("DoSomething invoked!");  
  
        if (MyEvent != null) {  
            Delegate[] delArray = MyEvent.GetInvocationList();  
  
            foreach (Delegate del in delArray) {  
                EventHandler method = (EventHandler)del;  
                method.BeginInvoke(null, EventArgs.Empty, null, null);  
            }  
        }  
    }  
}  
  
public class Subscriber1 {  
    public void OnEvent(object sender, EventArgs e) {  
        Thread.Sleep(TimeSpan.FromSeconds(3));      // 模拟耗时三秒才能完成方法  
        Console.WriteLine("Waited for 3 seconds, subscriber1 invoked!");  
    }  
}  
  
public class Subscriber2 {  
    public void OnEvent(object sender, EventArgs e) {  
        throw new Exception("Subsciber2 Failed");   // 即使抛出异常也不会影响到客户端  
        //Console.WriteLine("Subscriber2 immediately Invoked!");  
    }  
}  
  
public class Subscriber3 {  
    public void OnEvent(object sender, EventArgs e) {  
        Thread.Sleep(TimeSpan.FromSeconds(2));  // 模拟耗时两秒才能完成方法  
        Console.WriteLine("Waited for 2 seconds, subscriber3 invoked!");  
    }  
}

运行上面的代码，会得到下面的输出：

DoSomething invoked!  
Control back to client!  
  
Press any thing to exit...  
  
Waited for 2 seconds, subscriber3 invoked!  
Waited for 3 seconds, subscriber1 invoked!

需要注意代码输出中的几个变化：

1. 我们需要在客户端程序中调用Console.ReadKey()方法来暂停客户端，以提供足够的时间来让异步方法去执行完代码，不然的话客户端的程序到此处便会运行结束，程序会退出，不会看到任何订阅者方法的输出，因为它们根本没来得及执行完毕。原因是这样的：客户端所在的线程我们通常称为主线程，而执行订阅者方法的线程来自线程池，属于后台线程(Background Thread），当主线程结束时，不论后台线程有没有结束，都会退出程序。（当然还有一种前台线程(Foreground Thread)，主线程结束后必须等前台线程也结束后程序才会退出，关于线程的讨论可以开辟另一个庞大的主题，这里就不讨论了）。
2. 在打印完“Press any thing to exit...”之后，两个订阅者的方法会以2秒、1秒的间隔显示出来，且尽管我们先注册了subscirber1，但是却先执行了subscriber3，这是因为执行它需要的时间更短。除此以外，注意到这两个方法是并行执行的，所以执行它们的总时间是最长的方法所需要的时间，也就是3秒，而不是他们的累加5秒。
3. 如同前面所提到的，尽管subscriber2抛出了异常，我们也没有针对异常进行处理，但是客户程序并没有察觉到，程序也没有因此而中断。

## 委托和方法的异步调用

通常情况下，如果需要异步执行一个耗时的操作，我们会新起一个线程，然后让这个线程去执行代码。但是对于每一个异步调用都通过创建线程来进行操作显然会对性能产生一定的影响，同时操作也相对繁琐一些。.Net中可以通过委托进行方法的异步调用，就是说客户端在异步调用方法时，本身并不会因为方法的调用而中断，而是从线程池中抓取一个线程去执行该方法，自身线程（主线程）在完成抓取线程这一过程之后，继续执行下面的代码，这样就实现了代码的并行执行。使用线程池的好处就是避免了频繁进行异步调用时创建、销毁线程的开销。

如同上面所示，当我们在委托对象上调用BeginInvoke()时，便进行了一个异步的方法调用。**上面的例子中是在事件的发布和订阅这一过程中使用了异步调用，而在事件发布者和订阅者之间往往是松耦合的，发布者通常不需要获得订阅者方法执行的情况；而当使用异步调用时，更多情况下是为了提升系统的性能，而并非专用于事件的发布和订阅这一编程模型。**而在这种情况下使用异步编程时，就需要进行更多的控制，比如当异步执行方法的方法结束时通知客户端、返回异步执行方法的返回值等。本节就对BeginInvoke()方法、EndInvoke()方法和其相关的IAysncResult做一个简单的介绍。

**NOTE：**注意此处我已经不再使用发布者、订阅者这些术语，因为我们不再是讨论上面的事件模型，而是讨论在客户端程序中异步地调用方法，这里有一个思维的转变。

我们看这样一段代码，它演示了不使用异步调用的通常情况：

class Program7 {  
    static void Main(string[] args) {  
  
        Console.WriteLine("Client application started!\n");  
        Thread.CurrentThread.Name = "Main Thread";  
  
        Calculator cal = new Calculator();  
        int result = cal.Add(2, 5);  
        Console.WriteLine("Result: {0}\n", result);   
          
        // 做某些其它的事情，模拟需要执行3秒钟  
        for (int i = 1; i <= 3; i++) {  
            Thread.Sleep(TimeSpan.FromSeconds(i));  
            Console.WriteLine("{0}: Client executed {1} second(s).",  
                Thread.CurrentThread.Name, i);    
        }  
  
        Console.WriteLine("\nPress any key to exit...");  
        Console.ReadKey();  
    }  
}  
  
public class Calculator {  
    public int Add(int x, int y) {  
        if (Thread.CurrentThread.IsThreadPoolThread) {  
            Thread.CurrentThread.Name = "Pool Thread";  
        }  
        Console.WriteLine("Method invoked!");             
  
        // 执行某些事情，模拟需要执行2秒钟  
        for (int i = 1; i <= 2; i++) {  
            Thread.Sleep(TimeSpan.FromSeconds(i));  
            Console.WriteLine("{0}: Add executed {1} second(s).",   
                Thread.CurrentThread.Name, i);    
        }  
        Console.WriteLine("Method complete!");  
        return x + y;  
    }  
}

上面代码有几个关于对于线程的操作，如果不了解可以看一下下面的说明，如果你已经了解可以直接跳过：

* Thread.Sleep()，它会让执行当前代码的线程暂停一段时间（如果你对线程的概念比较陌生，可以理解为使程序的执行暂停一段时间），以毫秒为单位，比如Thread.Sleep(1000)，将会使线程暂停1秒钟。在上面我使用了它的重载方法，个人觉得使用TimeSpan.FromSeconds(1)，可读性更好一些。
* Thread.CurrentThread.Name，通过这个属性可以设置、获取执行当前代码的线程的名称，值得注意的是这个属性只可以设置一次，如果设置两次，会抛出异常。
* Thread.IsThreadPoolThread，可以判断执行当前代码的线程是否为线程池中的线程。

通过这几个方法和属性，有助于我们更好地调试异步调用方法。上面代码中除了加入了一些对线程的操作以外再没有什么特别之处。我们建了一个Calculator类，它只有一个Add方法，我们模拟了这个方法需要执行2秒钟时间，并且每隔一秒进行一次输出。而在客户端程序中，我们使用result变量保存了方法的返回值并进行了打印。随后，我们再次模拟了客户端程序接下来的操作需要执行2秒钟时间。运行这段程序，会产生下面的输出：

Client application started!  
  
Method invoked!  
Main Thread: Add executed 1 second(s).  
Main Thread: Add executed 2 second(s).  
Method complete!  
Result: 7  
  
Main Thread: Client executed 1 second(s).  
Main Thread: Client executed 2 second(s).  
Main Thread: Client executed 3 second(s).  
  
Press any key to exit...

如果你确实执行了这段代码，会看到这些输出并不是一瞬间输出的，而是执行了大概5秒钟的时间，因为线程是串行执行的，所以在执行完Add()方法之后才会继续客户端剩下的代码。

接下来我们定义一个AddDelegate委托，并使用BeginInvoke()方法来异步地调用它。在上面已经介绍过，BeginInvoke()除了最后两个参数为AsyncCallback类型和Object类型以外，前面的参数类型和个数与委托定义相同。另外BeginInvoke()方法返回了一个实现了IAsyncResult接口的对象（实际上就是一个AsyncResult类型实例，注意这里IAsyncResult和AysncResult是不同的，它们均包含在.Net Framework中）。

AsyncResult的用途有这么几个：传递参数，它包含了对调用了BeginInvoke()的委托的引用；它还包含了BeginInvoke()的最后一个Object类型的参数；它可以鉴别出是哪个方法的哪一次调用，因为通过同一个委托变量可以对同一个方法调用多次。

EndInvoke()方法接受IAsyncResult类型的对象（以及ref和out类型参数，这里不讨论了，对它们的处理和返回值类似），所以在调用BeginInvoke()之后，我们需要保留IAsyncResult，以便在调用EndInvoke()时进行传递。这里最重要的就是EndInvoke()方法的返回值，它就是方法的返回值。除此以外，当客户端调用EndInvoke()时，如果异步调用的方法没有执行完毕，则会中断当前线程而去等待该方法，只有当异步方法执行完毕后才会继续执行后面的代码。所以在调用完BeginInvoke()后立即执行EndInvoke()是没有任何意义的。我们通常在尽可能早的时候调用BeginInvoke()，然后在需要方法的返回值的时候再去调用EndInvoke()，或者是根据情况在晚些时候调用。说了这么多，我们现在看一下使用异步调用改写后上面的代码吧：

public delegate int AddDelegate(int x, int y);  
  
class Program8 {      
  
    static void Main(string[] args) {  
  
        Console.WriteLine("Client application started!\n");  
        Thread.CurrentThread.Name = "Main Thread";  
                      
        Calculator cal = new Calculator();  
        AddDelegate del = new AddDelegate(cal.Add);  
        IAsyncResult asyncResult = del.BeginInvoke(2,5,null,null);  // 异步调用方法  
  
        // 做某些其它的事情，模拟需要执行3秒钟  
        for (int i = 1; i <= 3; i++) {  
            Thread.Sleep(TimeSpan.FromSeconds(i));  
            Console.WriteLine("{0}: Client executed {1} second(s).",  
                Thread.CurrentThread.Name, i);  
        }  
  
        int rtn = del.EndInvoke(asyncResult);  
        Console.WriteLine("Result: {0}\n", rtn);  
  
        Console.WriteLine("\nPress any key to exit...");  
        Console.ReadKey();  
    }  
}  
  
public class Calculator { /\* 与上面同，略 \*/}

此时的输出为：

Client application started!  
  
Method invoked!  
Main Thread: Client executed 1 second(s).  
Pool Thread: Add executed 1 second(s).  
Main Thread: Client executed 2 second(s).  
Pool Thread: Add executed 2 second(s).  
Method complete!  
Main Thread: Client executed 3 second(s).  
Result: 7  
  
  
Press any key to exit...

现在执行完这段代码只需要3秒钟时间，两个for循环所产生的输出交替进行，这也说明了这两段代码并行执行的情况。可以看到Add()方法是由线程池中的线程在执行，因为Thread.CurrentThread.IsThreadPoolThread返回了True，同时我们对该线程命名为了Pool Thread。另外我们可以看到通过EndInvoke()方法得到了返回值。

有时候，我们可能会将获得返回值的操作放到另一段代码或者客户端去执行，而不是向上面那样直接写在BeginInvoke()的后面。比如说我们在Program中新建一个方法GetReturn()，此时可以通过AsyncResult的AsyncDelegate获得del委托对象，然后再在其上调用EndInvoke()方法，这也说明了AsyncResult可以唯一的获取到与它相关的调用了的方法（或者也可以理解成委托对象）。所以上面获取返回值的代码也可以改写成这样：

static int GetReturn(IAsyncResult asyncResult) {  
    AsyncResult result = (AsyncResult)asyncResult;  
    AddDelegate del = (AddDelegate)result.AsyncDelegate;  
    int rtn = del.EndInvoke(asyncResult);  
    return rtn;  
}

然后再将int rtn = del.EndInvoke(asyncResult);语句改为int rtn = GetReturn(asyncResult);。注意上面IAsyncResult要转换为实际的类型AsyncResult才能访问AsyncDelegate属性，因为它没有包含在IAsyncResult接口的定义中。

BeginInvoke的另外两个参数分别是AsyncCallback和Object类型，其中AsyncCallback是一个委托类型，它用于方法的回调，即是说当异步方法执行完毕时自动进行调用的方法。它的定义为：

public delegate void AsyncCallback(IAsyncResult ar);

Object类型用于传递任何你想要的数值，它可以通过IAsyncResult的AsyncState属性获得。下面我们将获取方法返回值、打印返回值的操作放到了OnAddComplete()回调方法中：

public delegate int AddDelegate(int x, int y);  
  
class Program9 {  
  
    static void Main(string[] args) {  
  
        Console.WriteLine("Client application started!\n");  
        Thread.CurrentThread.Name = "Main Thread";  
  
        Calculator cal = new Calculator();  
        AddDelegate del = new AddDelegate(cal.Add);  
        string data = "Any data you want to pass.";  
        AsyncCallback callBack = new AsyncCallback(OnAddComplete);  
        del.BeginInvoke(2, 5, callBack, data);      // 异步调用方法  
  
        // 做某些其它的事情，模拟需要执行3秒钟  
        for (int i = 1; i <= 3; i++) {  
            Thread.Sleep(TimeSpan.FromSeconds(i));  
            Console.WriteLine("{0}: Client executed {1} second(s).",  
                Thread.CurrentThread.Name, i);  
        }  
  
        Console.WriteLine("\nPress any key to exit...");  
        Console.ReadKey();  
    }  
  
    static void OnAddComplete(IAsyncResult asyncResult) {  
        AsyncResult result = (AsyncResult)asyncResult;  
        AddDelegate del = (AddDelegate)result.AsyncDelegate;  
        string data = (string)asyncResult.AsyncState;  
  
        int rtn = del.EndInvoke(asyncResult);  
        Console.WriteLine("{0}: Result, {1}; Data: {2}\n",   
            Thread.CurrentThread.Name, rtn, data);  
    }  
}  
public class Calculator { /\* 与上面同，略 \*/}

它产生的输出为：

Client application started!  
  
Method invoked!  
Main Thread: Client executed 1 second(s).  
Pool Thread: Add executed 1 second(s).  
Main Thread: Client executed 2 second(s).  
Pool Thread: Add executed 2 second(s).  
Method complete!  
Pool Thread: Result, 7; Data: Any data you want to pass.  
  
Main Thread: Client executed 3 second(s).  
  
Press any key to exit...

这里有几个值得注意的地方：1、我们在调用BeginInvoke()后不再需要保存IAysncResult了，因为AysncCallback委托将该对象定义在了回调方法的参数列表中；2、我们在OnAddComplete()方法中获得了调用BeginInvoke()时最后一个参数传递的值，字符串“Any data you want to pass”；3、执行回调方法的线程并非客户端线程Main Thread，而是来自线程池中的线程Pool Thread。另外如前面所说，在调用EndInvoke()时有可能会抛出异常，所以在应该将它放到try/catch块中，这里我就不再示范了。

## 总结

这篇文章是对我之前写的[C#中的委托和事件](http://www.tracefact.com/CSharp-Programming/Delegates-and-Events-in-CSharp.aspx)的一个补充，大致分为了三个部分，第一部分讲述了几个容易让人产生困惑的问题：为什么使用事件而不是委托变量，为什么通常委托的定义都返回void；第二部分讲述了如何处理异常和超时；第三部分则讲述了通过委托实现异步方法的调用。

感谢阅读，希望这篇文章能给你带来帮助。