

# C#中的委托和事件(续)

Delegates & Events in C#(continued)

张子阳

[www.tracefact.net](http://www.tracefact.net)

[jimmy\\_dev@163.com](mailto:jimmy_dev@163.com)

C#中的委托和事件(续) .....	1
引言 .....	3
为什么要使用事件而不是委托变量? .....	3
为什么委托定义的返回值通常都为void? .....	4
如何让事件只允许一个客户订阅? .....	5
获得多个返回值与异常处理 .....	8
委托中订阅者方法超时的处理 .....	12
委托和方法的异步调用 .....	17
总结 .....	22

# 引言

如果你看过了 [C#中的委托和事件](#) 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](#) 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 EventHandler 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 Subscriber2 {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 Subscriber1 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常用的方法了，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。

现在我们来了解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("Subscriber2 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 subscriber1，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 IAsyncResult 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 `IAsyncResult` 了, 因为 `AsyncCallback` 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](#)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。