导航

博客园
首 页
新随笔
联 系
订 阅 XML
管 理

< 2012年4月 >

日	一	二	三	四	五	六
25	26	27	28	29	30	31
1	2	3	4	5	6	7
8	9	10	11	12	13	14
15	16	17	18	19	20	21
22	23	24	25	26	27	28
29	30	1	2	3	4	5

公告

昵称：SkySoot
园龄：5年5个月
粉丝：209
关注：0
+加关注

搜索

找找看
谷歌搜索

常用链接

我的随笔
我的评论
我的参与
最新评论
我的标签

我的标签

ASP.NET MVC 注解 验证 特性 (1)
ASP.NET 生命周期(1)
C# Winform 多线程 UI线程 子线程 子线程修改 UI 线程控件的方式 (1)
C# Winform 多线程 生产 消费 (1)
C# 多线程 UI线程 子线程(1)
HttpHandler(1)
HttpModule(1)

随笔分类(264)

.NET 技术(11)
ASP.NET(110)
ASP.NET Dynamic Data(4)
ASP.NET MVC(20)
CSS(10)
HTML(4)
JavaScript(17)
JQuery
Lambda
LINQ
Reflection(4)
Silverlight(6)
Socket and WebSocket(1)
Sql Server(16)
Stream(7)
Thread(4)
WinForm(20)
操作系统概论(6)
技能点(5)
趣味应用(9)
数据库系统原理(9)
正则表达式(1)

C# 中的委托和事件(详解)

C# 中的委托和事件

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堵得慌，浑身不自在。本章中，我将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、委托中方法异常和超时的处理、委托与异步编程、委托和事件对Observer设计模式的意义，对它们的编译代码也做了讨论。

1.1 理解委托

1.1.1 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)
{
    EnglishGreeting(name);
}

public void EnglishGreeting(string name)
{
    Console.WriteLine("Good 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“Liker”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 “Good Morning, Liker”。

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“Good 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)
{
    switch (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“Liker”时，它就代表“Liker”这个值；当我们赋给它“李志中”时，它又代表着“李志中”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() 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？

说明：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“Good 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。

本例中委托的定义：

public delegate void GreetingDelegate(string name);

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 delegate void GreetingDelegate(string name);
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：

```
public delegate void GreetingDelegate(string name);

class Program
{
    private static void EnglishGreeting(string name)
    {
        Console.WriteLine("Good Morning, " + name);
    }

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

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

    static void Main(string[] args)
    {
        GreetPeople("Liker", EnglishGreeting);
        GreetPeople("李志中", ChineseGreeting);
```

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

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。

1.1.2 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)
{
    GreetPeople("Liker", EnglishGreeting);
    GreetPeople("李志中", ChineseGreeting);
    Console.ReadLine();
}
```

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("Liker", delegate1);
    GreetPeople("李志中", delegate2);
    Console.ReadLine();
}
```

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;
    delegate1 += ChineseGreeting;
    GreetPeople("Liker", delegate1);
    Console.ReadLine();
}
```

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;
    delegate1 += ChineseGreeting;
    delegate1("Liker");
    Console.ReadLine();
}
```

说明：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;
```

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;
    GreetPeople("Liker", delegate1);
    Console.WriteLine();

    delegate1 -= EnglishGreeting;
    GreetPeople("李志中", delegate1);
    Console.ReadLine();
}
```

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。

1.2 事件的由来

1.2.1 更好的封装性

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
{
    public delegate void GreetingDelegate(string name);

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

    class Program
    {
        private static void EnglishGreeting(string name)
        {
            Console.WriteLine("Good Morning, " + name);
        }

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

        static void Main(string[] args)
        {
            GreetingManager gm = new GreetingManager();
            gm.GreetPeople("Liker", EnglishGreeting);
            gm.GreetPeople("李志中", 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：

// ***

Good Morning, Liker

早上好, 李志中

// ***

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("Liker", delegate1);
}
```

输出：

Good Morning, Liker

早上好, Liker

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
{
    /// <summary>
    /// 在 GreetingManager 类的内部声明 delegate1 变量
    /// </summary>
    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("Liker", gm.delegate1);
}
```

输出为：

Good Morning, Liker

早上好, Liker

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, 既然如此，我们何不修改 GreetingManager 类成这样：

```
public class GreetingManager
 {
     /// <summary>
     /// 在 GreetingManager 类的内部声明 delegate1 变量
     /// </summary>
     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("Liker"); //注意，这次不需要再传递 delegate1 变量
}
```

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)
```

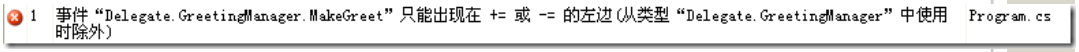
```
    {
        MakeGreet(name);
    }
}
```

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("Liker");
}
```

会得到编译错误：



1.2.2 限制类型能力

使用事件不仅能获得比委托更好的封装性以外，还能限制含有事件的类型的能力。这是什么意思呢？它的意思是说：事件应该由事件发布者触发，而不应该由事件的客户端（客户程序）来触发。请看下面的范例：

```
using System;

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
    }
}

/// <summary>
/// 定义委托
/// </summary>
/// <param name="count"></param>
public delegate void NumberChangedEventHandler(int count);

/// <summary>
/// 定义事件发布者
/// </summary>
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);
        }
    }
}

/// <summary>
/// 定义事件订阅者
/// </summary>
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

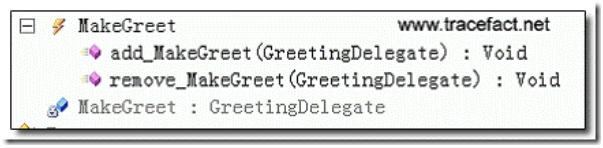
件，事件发布者的封装性会更好，事件仅仅是供其他类型订阅，而客户端不能直接触发事件（语句pub.NumberChanged(100)无法通过编译），事件只能在事件发布者Publisher 类的内部触发（比如在方法pub.DoSomething()中），换言之，就是NumberChanged(100)语句只能在Publisher 内部被调用。大家可以尝试一下，将委托变量的声明那行代码注释掉，然后取消下面事件声明的注释。此时程序是无法编译的，**当你使用了event 关键字之后，直接在客户端触发事件这种行为，也就是直接调用pub.NumberChanged(100)，是被禁止的。**事件只能通过调用DoSomething() 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。**

说 明：**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。**

1.3 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：



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。

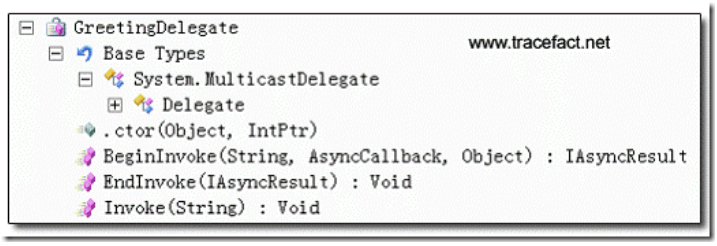
我们前面提到过两次，说委托实际上是一个类，在我们定义委托的时候：

```
// ***************************************************************************
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 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);

}

// ******************************************************************************
```



1.4 .NET 框架中的委托和事件

1.4.1 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
{
    /// <summary>
    /// 热水器
    /// </summary>
    public class Heater
    {
        /// <summary>
        /// 水温
        /// </summary>
        private int temperature;

        /// <summary>
        /// 烧水
        /// </summary>
        public void BoilWater()
        {
            for (int i = 0; i <= 100; i++)
            {
                temperature = i;
                if (temperature > 95)
                {
                    MakeAlert(temperature);
                    ShowMsg(temperature);
                }
            }
        }

        /// <summary>
        /// 发出语音警报
        /// </summary>
        /// <param name="param"></param>
        private void MakeAlert(int param)
        {
            Console.WriteLine("Alarm：嘀嘀嘀，水已经 {0} 度了：", param);
        }

        /// <summary>
        /// 显示水温
        /// </summary>
        /// <param name="param"></param>
        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();
        }
    }
}
```

1.4.2 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：

```
/// <summary>
/// 热水器
/// </summary>
public class Heater
{
    private int temperature;

    private void BoilWater()
    {
        for (int i = 0; i <= 100; i++)
        {
            temperature = i;
        }
    }
}

/// <summary>
/// 警报器
/// </summary>
public class Alarm
{
    private void MakeAlert(int param)
    {
        Console.WriteLine("Alarm: 嘀嘀嘀，水已经 {0} 度了：", param);
    }
}

/// <summary>
/// 显示器
/// </summary>
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：

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。

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。

1.4.3 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。

```
namespace Delegate
{
    public class Heater
    {
        private int temperature;

        public delegate void BoilHandler(int param);

        public event BoilHandler BoilEvent;

        public void BoilWater()
        {
            for (int i = 0; i <= 100; i++)
            {
                temperature = i;
                if (temperature > 95)
                {
                    if (BoilEvent != null)
                    {
                        BoilEvent(temperature); // 调用所有注册对象的方法
                    }
                }
            }
        }
    }

    public class Alarm
    {
        public void MakeAlert(int param)
        {
            Console.WriteLine("Alarm：嘀嘀嘀，水已经 {0} 度了：", param);
        }
    }

    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...

// ***

1.4.4 .NET 框架中的委托与事件

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：

1. 委托类型的名称都应该以 EventHandler 结束。

2. 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)。

3. 事件的命名为委托去掉 EventHandler 之后剩余的部分。

4. 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
{
    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

        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)
            {
                Boiled(this, e); // 调用所有注册对象的方法
            }
        }

        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
                }
            }
        }

        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();
            }
        }

        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 ...

1.5 委托进阶

1.5.1 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。

1.5.2 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：

```
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
        }
    }
}
```

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。

1.7 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。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。

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。

我们先看这样一段代码，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.ReadLine();
    }
}

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：

1. 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。

2. 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。**

3. 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：

```
using System.Threading;
using System;

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.ReadLine();
    }
}

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;
    }
```

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：

```
private 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：

```
using System.Threading;
using System;
using System.Runtime.Remoting.Messaging;

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.ReadLine();
    }

    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
{
    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;
    }
}
```

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，这里就不再示范了。

1.8 总结

我们详细地讨论了C#中的委托和事件，包括什么是委托、为什么要使用委托、事件的由来、.NET Framework 中的委托和事件、委托中方法异常和超时的处理、委托与异步编程、委托和事件对Observer 设计模式的意义。拥有了本章的知识，相信你以后遇到委托和事件时，将不会再有所畏惧。

分类: .NET 技术

SkySoot
关注 - 0
粉丝 - 209
+加关注

6 推荐 0 反对

« 上一篇 : C# (输入输出流)
» 下一篇 : C# 消息处理机制及自定义过滤方式

posted on 2012-04-05 20:02 SkySoot 阅读(14597) 评论(11) 编辑 收藏

评论

#1楼

好文把委托和事件讲解的非常清楚

支持(1) 反对(0)

2015-03-24 14:02 | 高海东

#2楼

好文章 完全让我理解了什么是委托跟事件
也终于看懂了每次遇到都满脸黑线的挫命名！！
十分感谢作者大大的无私奉献，千万别删，1.2以后的用法我都看3遍了就是太笨记不住，已收藏 随时查阅。给作者100个赞

支持(3) 反对(0)

2016-11-01 10:29 | 程序员新兵

#3楼

写的太好了

支持(1) 反对(0)

2016-11-02 14:57 | 黑色之天使

#4楼

写的确实太好了，由浅入深，慢慢带入，很适合新人看，很容易懂
唯一有一个地方有点小疑问：
string rtn = NumberChanged();
Console.WriteLine("Return: {0}", rtn); // 打印返回的字符串，输出Subscriber3
这个地方，在上面的代码中NumberChanged()不是没有返回值么，为什么还用rtn接收了一下，输出Subscriber3又是什么意思
希望看到的兄弟能解答一下，感谢

支持(1) 反对(0)

2016-12-13 15:41 | xiaojunjor

#5楼[楼主]

@ xiaojunjor
NumberChanged 是事件，NumberChanged() 执行的是绑定在这事件上的方法。虽然委托的返回值并没有什么意义，所以一般都为 void，但委托也是可以有返回值的。这里就是说明下委托也可以绑定一个有返回值的方法，代码可能漏贴了委托的声明那一行。

支持(0) 反对(0)

2016-12-13 16:18 | SkySoot

#6楼

@ SkySoot
原来是有返回值的委托没有贴，那就明白了
哈哈，感谢回复！

支持(0) 反对(0)

2016-12-13 19:02 | xiaojunjor

#7楼

受益匪浅，为博主的贡献点赞。

支持(0) 反对(0)

2017-02-22 09:16 | automan_xmu

#8楼

不错的文章,受益匪浅啊，点赞

支持(0) 反对(0)

2017-03-03 17:23 | tongfei

#9楼

楼主，我想问下，如果委托注册了多个方法，怎么异步调用。
我的委托注册了多个方法异步调用的时候提示委托只能绑定一个方法。
如果注册了多个方法的委托不能异步调用，那为什么不用线程更方便呢？

支持(0) 反对(0)

2017-03-16 19:00 | 416962254

#10楼

好文章，先转了，后面的异步调用还没看懂，先转了。

支持(0) 反对(0)

2017-03-30 10:26 | Michael我想念你

#11楼

写的贼好！博主大大好人，写的这么详细明了。看了一半了，收藏先，先练练那个例子区，感觉对事件和委托明了了很多很多

支持(0) 反对(0)

2017-04-20 21:22 | 风人

刷新评论 刷新页面 返回顶部

注册用户登录后才能发表评论，请 登录 或 注册，访问网站首页。

最新IT新闻:
· C#将引入可空的引用类型
· 为什么要做Caffe2？贾扬清亲自给出答案
· 所谓"优酷数据泄露事件"的客观事实还原