.Net编程中最经常用的元素，事件必然是其中之一。无论在ASP.NET还是WINFrom开发中，窗体加载(Load),绘制(Paint),初始化(Init)等等。  
“protected void Page\_Load(object sender, EventArgs e)”这段代码相信没有人不熟悉的。细心一点一定会发现，非常多的事件方法都是带了“object sender, EventArgs e”这两个参数。这是不是和委托非常相似呢？  
  
**一、委托（有些书中也称为委派）**  
  
委托是什么呢？这个名字的意思已经赋予了我们想象的空间，你是编程的，你现在正在写一个ASP.NET网页，而JS是你不熟悉的，于是你委托你的一位同事来帮助你完成JS部分。这就是委托，把你所不能做的事情交给其他人去做。而怎么知道是哪个人去做呢？当然是要知道名字！而为了区别名字一样的不同人，因此，需要描述一个特征。  
  
在C#中，委托的作用是这样描述的：委托就像一个函数的指针，在程序运行时可以使用它们来调用不同的函数。这个其实和你委托同事完成 JS代码一样。如果有两位同事可以做这件事情，他们只要做的结果能够满足你的需求（就像一个接口），尽管他们做的过程不一样，并且作出的效果也不一样，但是，能够达到你的要求就可以了。  
  
1、简单的委托  
  
那委托需要承载哪些信息呢？首先，它存储了方法名，还有参数列表(方法签名),以及返回的类型。比如：  
delegate string/\*返回类型\*/ ProcessDelegate(int i);  
这就是一个委托的定义。蓝色部分是声明委托的关键字，红色部分是返回的类型，而黑色部分是委托的类型名，和一个类名差不多，而()里的就是参数部分。它的意思是，你要使用这个委托来做事情的话，那么，做事情的方法必须满足以下条件：  
1、返回类型和委托的返回类型一致，这里是string类型；  
2、能且只能有一个参数，并且是int类型。  
OK,满足以上两个条件，一切就可以工作了:)  
  
例如：

 1 using System;  
 2 using System.Collections.Generic;  
 3 using System.Text;  
 4   
 5 namespace TestApp  
 6 {  
 7     /// <summary>  
 8     /// 委托  
 9     /// </summary>  
10     /// <param name="s1"></param>  
11     /// <param name="s2"></param>  
12     /// <returns></returns>  
13     public delegate string ProcessDelegate(string s1, string s2);  
14   
15     class Program  
16     {  
17         static void Main(string[] args)  
18         {  
19             /\*  调用方法  \*/  
20             ProcessDelegate pd = new ProcessDelegate(new Test().Process);  
21             Console.WriteLine(pd("Text1", "Text2"));  
22         }  
23     }  
24   
25     public class Test  
26     {  
27         /// <summary>  
28         /// 方法  
29         /// </summary>  
30         /// <param name="s1"></param>  
31         /// <param name="s2"></param>  
32         /// <returns></returns>  
33         public string Process(string s1,string s2)  
34         {  
35             return s1 + s2;  
36         }  
37     }  
38 }

输出的结果是:  
Text1Tex2  
  
  
2、泛型委托  
  
泛型的委托，就是然参数的类型不确定,例如代码改写为：

using System;  
using System.Collections.Generic;  
using System.Text;  
  
namespace TestApp  
{  
    /// <summary>  
    /// 委托  
    /// </summary>  
    /// <param name="s1"></param>  
    /// <param name="s2"></param>  
    /// <returns></returns>  
    public delegate string ProcessDelegate<T,S>(T s1, S s2);  
  
    class Program  
    {  
        static void Main(string[] args)  
        {  
            /\*  调用方法  \*/  
            ProcessDelegate<string,int> pd = new ProcessDelegate<string,int>(new Test().Process);  
            Console.WriteLine(pd("Text1", 100));  
        }  
    }  
  
    public class Test  
    {  
        /// <summary>  
        /// 方法  
        /// </summary>  
        /// <param name="s1"></param>  
        /// <param name="s2"></param>  
        /// <returns></returns>  
        public string Process(string s1,int s2)  
        {  
            return s1 + s2;  
        }  
    }  
}

输出的结果就是：  
Text1100  
  
泛型的详细内容不属于本文的介绍范围，这里不加多说了。  
  
**二、事件**  
在某件事情发生时，一个对象可以通过事件通知另一个对象。比如，前台完成了前台界面，他通知你，可以把前台和你开发的程序整合了。这就是一个事件。可以看出事件是在一个时间节点去触发另外一件事情，而另外一件事情怎么去做，他不会关心。就事件来说，关键点就是什么时候，让谁去做。  
  
在C#中，时间定义关键字是event。例如：  
event ProcessDelegate ProcessEvent;  
  
整个事件定义方法以及执行过程：

using System;  
using System.Collections.Generic;  
using System.Text;  
  
namespace TestApp  
{  
    /// <summary>  
    /// 委托  
    /// </summary>  
    /// <param name="s1"></param>  
    /// <param name="s2"></param>  
    /// <returns></returns>  
    public delegate void ProcessDelegate(object sender, EventArgs e);  
  
    class Program  
    {  
          
  
        static void Main(string[] args)  
        {  
            /\*  第一步执行  \*/  
            Test t = new Test();  
            /\* 关联事件方法，相当于寻找到了委托人 \*/  
            t.ProcessEvent += new ProcessDelegate(t\_ProcessEvent);  
            /\* 进入Process方法 \*/  
            Console.WriteLine(t.Process());   
  
            Console.Read();  
        }  
  
        static void t\_ProcessEvent(object sender, EventArgs e)  
        {  
            Test t = (Test)sender;  
            t.Text1 = "Hello";  
            t.Text2 = "World";  
        }  
    }  
  
    public class Test  
    {  
        private string s1;  
  
        public string Text1  
        {  
            get { return s1; }  
            set { s1 = value; }  
        }  
  
        private string s2;  
  
        public string Text2  
        {  
            get { return s2; }  
            set { s2 = value; }  
        }  
  
  
        public event ProcessDelegate ProcessEvent;  
  
        void ProcessAction(object sender, EventArgs e)  
        {  
            if (ProcessEvent == null)  
                ProcessEvent += new ProcessDelegate(t\_ProcessEvent);  
            ProcessEvent(sender, e);  
        }  
  
        //如果没有自己指定关联方法，将会调用该方法抛出错误  
        void t\_ProcessEvent(object sender, EventArgs e)  
        {  
            throw new Exception("The method or operation is not implemented.");  
        }  
  
        void OnProcess()  
        {  
            ProcessAction(this, EventArgs.Empty);  
        }  
  
        public string Process()  
        {  
            OnProcess();  
            return s1 + s2;  
        }  
    }  
}

感觉到了什么？是不是和代码注入了差不多，相当于是可以用任意符合委托接口(委托确实很像接口)的代码，注入到Process过程。在他返回之前给他赋值。  
  
**三、回调函数**  
  
打了这么多字，好累啊!  
  
回调函数就是把一个方法的传给另外一个方法去执行。在C#有很多回调函数，比如异步操作的时候。这里先举个例子：

using System;  
using System.Collections.Generic;  
using System.Text;  
  
namespace TestApp  
{  
    /// <summary>  
    /// 委托  
    /// </summary>  
    /// <param name="s1"></param>  
    /// <param name="s2"></param>  
    /// <returns></returns>  
    public delegate string ProcessDelegate(string s1, string s2);  
  
    class Program  
    {  
        static void Main(string[] args)  
        {  
            /\*  调用方法  \*/  
            Test t = new Test();  
            string r1 = t.Process("Text1", "Text2", new ProcessDelegate(t.Process1));  
            string r2 = t.Process("Text1", "Text2", new ProcessDelegate(t.Process2));  
            string r3 = t.Process("Text1", "Text2", new ProcessDelegate(t.Process3));  
  
            Console.WriteLine(r1);  
            Console.WriteLine(r2);  
            Console.WriteLine(r3);  
        }  
    }  
  
    public class Test  
    {  
        public string Process(string s1,string s2,ProcessDelegate process)  
        {  
            return process(s1, s2);  
        }  
  
        public string Process1(string s1, string s2)  
        {  
            return s1 + s2;  
        }  
  
        public string Process2(string s1, string s2)  
        {  
            return s1 + Environment.NewLine + s2;  
        }  
  
        public string Process3(string s1, string s2)  
        {  
            return s2 + s1;  
        }  
    }  
}

输出结果：  
Text1Text2  
Text1  
Text2  
Text2Text1  
  
Process方法调用了一个回调函数，当然这里只执行了回调函数。可以看出，可以把任意一个符合这个委托的方法传递进去，意思就是说这部分代码是可变的。而设计上有一个抽离出可变部分代码的原则，这种用法无疑可以用到那种场合了。

C# 里面这个东西一般指某个委托.其实也可以是接口.  
  
delegate void WorkDone();  
void Working(WorkDone callBack){  
//Working code.  
//当工作完成的时候执行这个委托.  
callBack();  
}  
void WorkDoneHandler(){  
 //Do something other.  
}  
void Do(){  
 WorkDone callBack = new WorkDone(WorkDoneHandler);  
 Working(callBack);  
}  
  
也就是相当把某个方法(WorkDoneHandler())当作参数传递给另外一个方法(Working(WorkDone callBack)).  
这样做的好处在于,可以动态的指定执行哪个方法.  
比如在Do()方法中,我们指定的callback 是WorkDoneHandler 当然也可以是其它匹配的方法. 而Working()方法 更本不需要知道自己最后执行的是哪个Handler.