委托、事件和 Lambda

委托是一个神奇的东西。委托的出现，使得方法可以作为参数进行传递。其中我们接触最多的应该就是通用库类。通用库类，正是由于这种机制才实现了其的通用性。

一、普通委托

委托类由关键字 delegate 来声明。我们先看看，定义一个委托类的语法：

[csharp] view plain copy

```
1. [访问限制符] delegate [返回值类型] [委托类的名称]( [参数列表] );
```

实际上这里隐藏了一个派生关系----委托类派生自基类 System.MulticastDelegate。然而我们只需知道有这么回事就好了，其余的编译器会帮我们完成。

1、单路委托

我们已经知道了委托类的定义，那么我们来具体看看委托使用的一个实例吧：

[csharp] view plain copy

```
1. public delegate int Calc(int a, int b); //定义了一个委托类 Calc
```

假设我们存在方法 int add(int x, int y)和方法 int Mul(int a, int b)。假定这两个方法分别是返回两数之和与两数之积。

[csharp] view plain copy

```
1. Calc TestDele = new Calc(add);  //委托对象的初始化和赋值和普通类一致
```

现在我们创建了一个委托对象 TestDele 并为将方法 add 作为初值赋给 TestDele。从这里我们又可以说明一个委托的性质-----作为值赋给委托的方法必

须与委托类的返回值类型，参数类型一致。只有满足这两个条件可以将值赋给委托类的对象，否则编译器将会报错。

[csharp] view plain copy

```
1. TestDele(1, 2);        //现在这个语句就等价于 add(1, 2)
```

我们除了上面这种方式调用方法，我们还可以使用 Invoke()方法来实现上述的功能。换而言之，TestDele.Invoke()和 TestDele()是完全等价的。(其实接触过 C\C++的同学会发现，委托的本质上其实就是 C\C++中的函数指针，唯一不同点就是委托比函数指针安全)

注意：我们通常将委托类和委托类的对象都成为委托，但是两者是有区别的。一旦定义了委托类，基本上就可以实例化它的实例，在这一点上和普通类似一致的。即我们也可以有委托数组。

2、多播委托

然而委托除了可以与方法建立一对一的关系，它还可以和方法有一对多的关系。我们称这种委托为多播委托。我们使用+=和-=来是实现增加方法和删除方法的功能。我们继续以上面的例子为例，在上面的基础上我们给委托对象加入第二种方法：

[csharp] view plain copy

```
1. TestDele += Mul;       //新加入的方法也必须满足返回值类型和参数列表与委托一致
2. TestDele(1, 2);
```

现在我们调用方法的话，那么编译器就会分别调用 add 和 Mul 方法，但是我们要注意两点(1、我们无法保证调用的顺序 2、如果是有返回值的多播委托，那么委托的返回值将是最后一个加入的方法)。

我们在使用多播委托的过程中可能会出现委托链中的某个方法抛出异常。那么这时委托的迭代就会停止。我们为了避免这个问题，Delegate 类就定义 GetInvocationList()方法 ,它返回一个 Delegate 对象数组。我们还是用上述的 Calc 委托类作为实例：

[csharp] view plain copy

```
1. Delegate[] delegates = TestDele.GetInvocationList();
2. //将 TestDele 的方法列表传给 Delegate 数组 delegates
3. foreach(Calc d in delegates){
4.     try{
5.         d(1 , 3);
6.     }
7.     catch(Excetion){
8.     //异常处理代码
9.     }
10.}
```

这样的话，程序运行过程中在捕获异常之后还会继续迭代下一个方法。

3、Action<T>和 Func<T>委托

我们之前都是根据返回值类型和参数列表来定义委托类 ,然后在根据委托类来生成委托的实例。现在我们还可以使用泛型委托类 Action<T>和 Func<T>。

泛型 Action<T>委托类表示引用一个 void 返回类型的方法，可以传递至多 16 个不同的参数类型。Action<in T1,in T2, …,in Tn> (n 最大为 16,例如 Action<in T1,in T2>就表示调用 2 个参数的方法)。Func<T>委托的使用方式和 Action<T>委托类似.Func<T>允许调用带有返回值的方法。Func<in T1, in T2, ...,in Tn, out TResult> (n 的最大值还是 16,Func<in T1, in T2, out TResult> 表示调用两个参数的方法且返回值类型为 TResult)。

我们用 Func<T>委托来实现上述 Calc 委托：

[csharp] view plain copy

```
1. Func<int, int, int> TestDele = add;
```

这一条语句就等价于委托类的声明和委托对象的创建。同理，Action<T>也是一样的用法。而且功能上没有任何的不同。唯一不足之处就是参数的个数是有限制的，不过大多数的情况下 16 个的参数已经足够使用了。

二、匿名委托和 lambda 表达式

1、匿名委托

在这之前我们使用委托那么都必须先有一个方法。那么现在我们可以通过另一种方式使委托工作：匿名方法。用匿名方法来实现委托和之前的定义并没有太大的区别，唯一不同之处就在于实例化。我们就以之前 Calc 委托类为例：

[csharp] view plain copy

```
1. Calc TestDele = delegate(int a,int b)
2. {
3.         //代码块(因为 Calc 委托类是有返回值的，所以函数体内必须有 return 语句)
4. };    //这里有一个分号，千万不能漏
```

通过使用匿名方法，由于不必创建单独的方法，因此减少了实例化委托所需的编码系统开销。而且使用匿名方法可以有效减少要编写的代码，有助于降低代码的复杂度。

然而我们在使用匿名委托的时候我们要遵守两个原则：1、匿名方法中不能有跳转语句(break, goto 或 continue)跳转到匿名方法的外部，反之，外部代码也不能跳转到该匿名方法内部。2、在匿名方法中不能访问不安全代码。

注意：不能访问在匿名方法外部使用的 ref 和 out 参数。

2、Lambda **表达式**

由于 Lambda 表达式的出现使得我们的代码可以变得更加的简洁明了。我们现在来看看 Lambda 表达式的使用语法：

[csharp] view plain copy

```
1. [委托类] [委托对象名] = ( [参数列表] ) => { /*代码块*/ };          //结尾还是有一个分号
```

我们值得注意的是 lambda 表达式的参数列表，我们只需给出变量名即可，其余的编译器会自动和委托类进行匹配。如果委托类使用返回值的，那么代码块就需要 return 一个返回值。我们用一个例子来说明上述问题:

[csharp] view plain copy

```
1. Func<int, int> TestLam = (x) => { return x*x; };
```

在这里我们是使用一个参数的为例，上面的写法是 Lambda 表达式的正常写法，但是当参数只有一个时，x 两边的括号就可以去除，那么现在代码就变成这样了：

[csharp] view plain copy

```
1. Func<int, int> TestLam = x => { return x*x; };
```

当 Lambda 表达式代码块中只有一条语句，那么我们就可以把花括号丢了。如果这一条语句还是包含 return 的语句，那么我们在去除花括号的同时，必须将 return 同时删去。现在上述代码就变成了这样：

[csharp] view plain copy

```
1. Func<int, int> TestLam = x => x*x;
```

注意：Lambda 表达式可以用于类型为委托的任意地方。

3、**闭包**

通过 lambda 表达式可以访问 lambda 表达式外部的变量，于是我们就引出了一个新的概念-----闭包。我们来看一个例子：

[csharp] view plain copy

```
1. int someVal = 5;
2. Func<int, int> f = x => x+someVal;
```

现在我们很容易知道 f(3)的返回值是 8，我们继续：

[csharp] view plain copy

```
1. someVal = 7;
```

我们现在将 someVal 的值改为 7，那么这时我们在调用 f(3)，现在就会很神奇的发现 f(3)的返回值变成了 10。这就是闭包的特点，这个特点在编程上很大程度上能给我们带来一定的好处。但是有利终有弊，如果我们使用不当，那么这就变成了一个非常危险的功能。

我现在再来看看在 foreach 语句中的闭包，我们现在看看下面这段代码：

[csharp] view plain copy

```
1. List<int> values = new List<int>() { 10, 20, 30 };
2. var funcs = new List<Func<int>>();
3. foreach (var val in values)
4. {
5.     funcs.Add(() => val);
6. }
7. foreach (var f in funcs)
8. {
9.     Console.WriteLine(f());
10.}
```

用我们刚才的知识来判断的话，输出结果应该是 3 个 30。然而在 C#4.0 确实是这样，然而 C#5.0 会在 foreach 创建的 while 循环的代码块中创建一个不同的局部循环变量，所以这时在 C#5.0 中我们输出的结果应该是分别输出 10,20 和 30。

三、事件

事件是一种特殊多播委托，换句话来说，事件是经过深度封装的委托。一个事件简单的可以看作一个多播委托加上两个方法(+=订阅消息和-=取消订阅)。

1、普通事件

我们使用 event 关键字来声明事件，语法如下：

[csharp] view plain copy

```
1. [访问权限修饰符] event [委托类类名] [名称];
```

事件一般是用于通知代码发生了什么。由此我们又可以引出两个概念：1、事件发布方 2、事件侦听方。我们现在用一个简单的例子来说明这两个概念，我们以烧开水为例，当水温为 95 至 100 度时发出警报。我们先来定义在事件发生时，需要传输的数据成员：

[csharp] view plain copy

```
1. public class Water   //事件发布程序中的基本数据成员类
2. {
3.         public int Temperature { get; private set; }
4.         public Water(int t)
5.         {
6.                 this.Temperature = t;
7.         }
8. }
```

有了传输的数据，那么我们现在就可以定义事件触发类：：

[csharp] view plain copy

```
1. public delegate void WaterHandler(object sender, Water w); //sender 为事件发送者，w 为发送的数据
2. public class Heater //事件发布程序中的，事件触发类
3. {
```

```
4.      public event WaterHandler WaterEvent;       //深度的封装委托
5.      public void HeatWater() //该方法用于触发事件
6.      {
7.          for (int i = 0; i < 101; i++)
8.          {
9.              if (i > 95 && i < 101)
10.             {
11.                 RegWaterEvent(i);           //触发事件
12.             }
13.         }
14.     }
15.     protected virtual void RegWaterEvent(int t)
16.     {
17.         WaterHandler temp = WaterEvent;
18.         if (temp != null)
19.             temp(this, new Water(t));     //如果委托不为空，我们就执行委托，我们无需知道具体执行了哪些方法
20.     }
21.}
```

现在我们已经完整了定义好了事件发布方了 ,通过这个例子我们也知道了事件发布方由两部分组成：1、基本数据类　2、事件触发类。接下来我们继续看看事件侦听方又是怎么样的：

[csharp] view plain copy

```
1.  public class Alarm   //事件侦听类
2.  {
3.      public void Waring(object sender, Water w)        //侦听接口，由于侦听事件的发布
```

```
4.      {
5.          Console.WriteLine("当前水温已经到达 {0} ℃！", w.Temperature);
6.      }
7. }
```

通过这个例子我们可以发现，事件侦听类，只需有一个和被监听事件一致的方法即可。

[csharp] view plain copy

```
1. Heater heater = new Heater();          //生成事件发布实例
2. Alarm alarm = new Alarm();
3. heater.WaterEvent += alarm.Waring;  //对事件发布方进行订阅(侦听),反之我们使用-=取消订阅
4. heater.HeatWater();                //触发事件，那么现在 Alarm 类对象 alarm 将会侦听到这次事件
```

通过上述例子我们就大致的了解了事件的工作情况，以及事件发布方和事件侦听方的概念。

.Net 平台为我们提供了泛型委托 EventHandler<T>,有了这个泛型委托之后我们就不在需要定义委托类了。我们来看看泛型委托 EventHandler<T>的原型：

[csharp] view plain copy

```
1. public delegate void EventHandler<TEventArgs>(object sender, TEventArgs e)
2. where TEventArgs: EventArgs;
```

参数列表中第一个参数是对象，包含事件的发送者，第二个参数提供了事件的相关信息。现在我们定义事件时，只需让基本数据类继承 EventArgs，然后我们就能泛型委托来定义事件了。

注意：事件只能在本类型内部“触发”，委托不管在本类型内部还是外部都可以“调用”。事件在类的外部只能使用+=或-=来增加/取消订阅。

2、弱事件

我们通过事件，将事件发布方(source)与事件侦听方(listener)连接在一起。但是现在问题来了。当事件发布方(source)比事件侦听方(listener)具有更长的生命期，且事件侦听方没有被其他对象引用也不需要改事件。由于事件发布方还有保存着侦听方的一个引用，这时就会导致垃圾回收器不能清空事件侦听器所占用的内存。于是，就发生内存泄露现象。

<1>弱事件管理器

每当侦听器需要注册事件，而该侦听器并不明确了解什么时候注销时，就可以使用弱事件模式。我们这时只要让事件发布方的变为弱引用，那么在我们不使用侦听器的时候，垃圾回收机制就可以发挥它的作用了。.Net 平台为我们听过了 WeakEventManager 类作为发布程序与侦听器之间的中介，也就是弱事件管理器。现在我们增加/取消订阅通过 WeakEventManager 的方法 AddListener 和 RemoveListener 来实现，这样发布程序的引用就变为了弱引用。要使用弱事件那么就需要一个派生自 WeakEventManager 类(System.Windows 名称空间中)的类，不仅如此还需要让侦听器实现接口 IWeakEventsListener。

我们就以上述烧开水的为例，在定义一个弱事件管理器类 WeakBoilWaterEventManager 之前，我们得先把上述例子的事件用泛型委托 EventHandler<T> 重新定义（在此就不写代码了），然后在定义 WeakBoilWaterEventManager：

[csharp] view plain copy

```
1. class WeakBoilWaterEventManager : WeakEventManager  //继承自WeakEventManager 的弱事件管理类
2. {
3.     public static void AddListener(object source, IWeakEventListener listener)  //增加订阅
4.     {
5.         CurrentManager.ProtectedAddListener(source, listener);  //将提供的侦听器(listener)添加到为托管事件所提供的事件发布方(source)中。
6.     }
7.     public static void RemoveListener(object source, IWeakEventListener listener) //取消订阅
8.     {
9.         CurrentManager.ProtectedRemoveListener(source, listener); //从提供事件发布方的中移除以前添加的侦听器。
10.     }
11.     public static WeakBoilWaterEventManager CurrentManager //WeakBoilWaterEventManager 的实例
12.     {
13.         get
14.         {
15.             var manager = GetCurrentManager(typeof(WeakBoilWaterEventManager)) as WeakBoilWaterEventManager;
16.             if (manager == null)
17.             {
18.                 manager = new WeakBoilWaterEventManager();
19. SetCurrentManager(typeof(WeakBoilWaterEventManager), manager);
20.             }
```

```
21.                 return manager;
22.             }
23.     }
24.     void Heater_WaterEvent(object sender, Water w)
25.     {
26.         DeliverEvent(sender, w);      //将正在托管的事件传送到每个侦听器。
27.     }
28.     protected override void StartListening(object source)      //开始侦听被托管的事件
29.     {
30.         (source as Heater).WaterEvent += Heater_WaterEvent;
31.     }
32.     protected override void StopListening(object source)      //停止侦听被托管的事件
33.     {
34.         (source as Heater).WaterEvent -= Heater_WaterEvent;
35.     }
36.}
```

现在我们就创建好了一个弱事件管理类，因为它是管理事件 WaterEvent 和侦听器之间的连接，所以我们把这个类实现为单态模式，即我们继续创建一个实例---静态属性 CurrentManager。它用于访问弱事件管理器类中的单态对象。

现在我们光有弱事件管理类还不够，我们还需要让侦听器实现 IWeakEventListener 的接口。该接口只有一个 ReceiveWeakEvent 方法。触发事件时从弱事件管理器中调用这个方法。

[csharp] view plain copy

```
1.  public class Alarm : IWeakEventListener
```

```
2. {
3.     public void Waring(object sender, Water w)
4.     {
5.         Console.WriteLine(" 当 前 水 温 已 经 到 达  {0}  ℃ ！ ", w.Temperature);
6.     }
7.
8.     public bool ReceiveWeakEvent(Type managerType, object sender, EventArgs e)
9.     {
10.        Waring(sender, e as Water);
11.        return true;
12.    }
13.}
```

现在万事俱备了，接下我们只需要使用 AddListener 和 RemoveListener 方法来进行增加/取消订阅即可：

[csharp] view plain copy

```
1. Heater heater = new Heater();
2. Alarm alarm = new Alarm();
3. WeakBoilWaterEventManager.AddListener(heater, alarm);
4. heater.HeatWater();
```

现在事件发布方和事件侦听方之间不再是强连接，当不再引用侦听器时，他就会被垃圾回收。

<2>泛型弱事件管理器

通过刚才的例子我们可以发现，像这样处理弱事件十分的麻烦。于是于是 .Net 平台提供了泛型版本的弱事件管理器。泛型类 WeakEventManager<TEventSource, TEventArgs>派生自 WeakEventManager，它简化我们弱事件的处理。使用这个类时不需要在为每个事件定义弱事件管理器，

也不需要让侦听器实现接口 IWeakEventsListener。我们只需使用 AddHandler 和 RemoveHandler 来实现增加/取消订阅。

我们还是使用烧开水的那个例子来说明(事件 WaterEvent 还是要用泛型委托 EventHandler<T>来实现)：

[csharp] view plain copy

```
1. Heater heater = new Heater();
2. Alarm alarm = new Alarm();
3. WeakEventManager<Heater, Water>.AddHandler(heater, "WaterEvent", alarm.Waring);
4. heater.HeatWater();
```