委托详解

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。

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：

以下为引用的内容：

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：

以下为引用的内容：

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：

以下为引用的内容：

view plaincopy to clipboardprint?

1. public enum Language{

2. English, Chinese

3. }

4. public void GreetPeople(string name, Language lang){

5. //做某些额外的事情，比如初始化之类，此处略

6. swith(lang){

7. case Language.English:

8. EnglishGreeting(name);

9. break;

10. case Language.Chinese:

11. ChineseGreeting(name);

12. break;

13. }

14. }

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; } }

< type="text/javascript"> < type="text/javascript">

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。 Www~

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：

以下为引用的内容：

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？

NOTE：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 的参数类型和祷乩嘈汀?br /> 于是，委托出现了：它定义了MakeGre

eting参数所能代表的方法的种类，也就是MakeGreeting参数的类型。

NOTE：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：

以下为引用的内容：

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：

.

以下为引用的内容：

view plaincopy to clipboardprint?

1. using System;

2. using System.Collections.Generic;

3. using System.Text;

4. namespace Delegate {

5. //定义委托，它定义了可以代表的方法的类型

6. public delegate void GreetingDelegate(string name);

7. class Program {

8. private static void EnglishGreeting(string name) {

9. Console.WriteLine(”Morning, ” + name);

10. } Www~

11. private static void ChineseGreeting(string name) {

12. Console.WriteLine(”早上好, ” + name);

13. }

14. //注意此方法，它接受一个GreetingDelegate类型的方法作为参数

15. private static void GreetPeople(string name, GreetingDelegate MakeGreeting) {

16. MakeGreeting(name);

17. }

18. static void Main(string[] args) {

19. GreetPeople(”Jimmy Zhang”, EnglishGreeting);

20. GreetPeople(”张子阳”, ChineseGreeting);

21. Console.ReadKey();

22. }

23. }

24. } Www~

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); } Www~ 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 Gre

etPeople(string name, GreetingDelegate MakeGreeting) { MakeGreeting(name); } static void Main(string[] args) { GreetPeople(”Jimmy Zhang”, EnglishGreeting); GreetPeople(”张子阳”, ChineseGreeting); Console.ReadKey(); } } } Www~

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。

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：

以下为引用的内容：

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？

以下为引用的内容：

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：

以下为引用的内容：

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：

以下为引用的内容：

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”);

Con

sole.ReadKey();

}

NOTE：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。

Www~

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 @com

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“-=”：

以下为引用的内容：

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 Www\_

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。

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：

Www\_

以下为引用的内容：

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);

} @com

static void Main(string[] args) {

// … …

}

}

}

Www~

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：

以下为引用的内容：

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：

以下为引用的内容：

static void Main(string[] args) {

GreetingManager gm = new GreetingManager();

GreetingDelegate delegate1;

delegate1 = EnglishGreeting;

delegate1 += ChineseGreeting;

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

}

输出：

Www~

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：

以下为引用的内容：

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：

以下为引用的内容：

static void Main(string[] args) {

GreetingManager gm = new GreetingManager();

gm.delegate1 = EnglishGreeting;

gm.delegate1 += ChineseGreeting;

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

}

尽管这样达到了我们要的效果，但是似乎并不美气，

光是第一个方法注册用“=”，第二个用“+=”就让人觉得别扭。此时，轮到Event出场了，C# 中可以使用事件来专门完成这项工作，我们改写GreetingManager类，它变成了这个样子：

以下为引用的内容：

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, GreetingDelegate MakeGreeting) {

MakeGreeting(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，你差不多明白到：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委托类型的变量而已。

我们想当然地改写Main方法：

以下为引用的内容：

static void Main(string[] args) {

GreetingManager gm = new GreetingManager();

gm.MakeGreet = EnglishGreeting; // 编译错误1

gm.MakeGreet += ChineseGreeting;

gm.GreetPeople(”Jimmy Zhang”, gm.MakeGreet); //编译错误2

}

这次，你会得到编译错误：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)。

事件和委托的编译代码

这时候，我们不得不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。

我们进一步看下MakeGreet所产生的代码：

以下为引用的内容：

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：

以下为引用的内容：

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);

}

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。

委托、事件与Observer设计模式

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()。

以下为引用的内容：

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();

}

}

}

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：

以下为引用的内容：

// 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：

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：

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)。

热水器知道后保留对警报器和显示器的引用。

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。 Www~

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。

以下为引用的内容：

using System;

using System.Collections.Generic;

using System.Text;

namespace Delegate {

// 热水器

public class Heater {

private int temperature;

public delegate v

oid 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;

Www~

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： Www\_

Alarm：嘀嘀嘀，水已经 96 度了：

Display：水快烧开了，当前温度：96度。

// 省略… @com

.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：

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)。

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：

以下为引用的内容：

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, BoliedEventArgs e);

public event BoiledEventHandler Boiled; //声明事件

// 定义BoliedEventArgs类，传递给Observer所感兴趣的信息

public class BoliedEventArgs : EventArgs {

public readonly int temperature;

public BoliedEventArgs(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 OnBolied(BoliedEventArgs 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BoliedEventArgs 对象。

BoliedEventArgs e = new BoliedEventArgs(temperature); .

OnBolied(e); // 调用 OnBolied方法

}

}

}

}

// 警报器

public class Alarm {

public void MakeAlert(Object sender, Heater.BoliedEventArgs 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.BoliedEventArgs 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

}

}

}

Www\_

输出为：

以下为引用的内容：

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 …

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。