分享到...

微信微博复制网址邮件QQ好友QQ空间人人网开心网印象笔记领英飞信豆瓣TwitterFacebook一键分享查看更多(96)

JiaThis

风尘浪子

只要肯努力，梦想总有一天会实现

C#综合揭秘——深入分析委托与事件

引言

本篇文章将为你介绍一下 Delegate 的使用方式，逐渐揭开 C# 当中事件（Event）的由来，它能使处理委托类型的过程变得更加简单。

还将为您解释委托的协变与逆变，以及如何使用 Delegate 使 Observer（观察者）模式的使用变得更加简单。

在事件的介绍上，会讲述事件的使用方式，并以ASP.NET的用户控件为例子，介绍一下自定义事件的使用。

最后一节，将介绍Predicate<T>、Action<T>、Func<T，TResult>多种泛型委托的使用和Lambda的发展过程与其使用方式。

因为时间仓促，文中有错误的地方敬请点评。

目录

一、委托类型的来由

二、建立委托类

三、委托使用方式

四、深入解析事件

五、Lambda 表达式

一、委托类型的来由

记得在使用C语言的年代，整个项目中都充满着针指的身影，那时候流行使用函数指针来创建回调函数，使用回调可以把函数回调给程序中的另一个函数。但函数指针只是简单地把地址指向另一个函数，并不能传递其他额外信息。

在.NET中，在大部分时间里都没有指针的身影，因为指针被封闭在内部函数当中。可是回调函数却依然存在，它是以委托的方式来完成的。委托可以被视为一个更高级的指针，它不仅仅能把地址指向另一个函数，而且还能传递参数，返回值等多个信息。系统还为委托对象自动生成了同步、异步的调用方式，开发人员使用 BeginInvoke、EndInvoke 方法就可以抛开 Thread 而直接使用多线程调用 。

回到目录

二、建立委托类

使用delegate就可以直接建立任何名称的委托类型，当进行系统编译时，系统就会自动生成此类型。您可以使用delegate void MyDelegate() 方式建立一个委托类，并使用ILDASM.exe观察其成员。由ILDASM.exe 中可以看到，它继承了System.MulticastDelegate类，并自动生成BeginInvoke、EndInvoke、Invoke 等三个常用方法。

Invoke 方法是用于同步调用委托对象的对应方法，而BeginInvoke、EndInvoke是用于以异步方式调用对应方法的。

对于异步调用的使用方式，可以参考：C#综合揭秘——细说多线程

复制代码

1 public class MyDelegate:MulticastDelegate

2 {

3 //同步调用委托方法

4 public virtual void Invoke();

5 //异步调用委托方法

6 public virtual IAsyncResult BeginInvoke(AsyncCallback callback,object state);

7 public virtual void EndInvoke(IAsyncResult result);

8 }

复制代码

MulticastDelegate是System.Delegate的子类，它是一个特殊类，编译器和其他工具可以从此类派生，但是自定义类不能显式地从此类进行派生。它支持多路广播委托，并拥有一个带有链接的委托列表，在调用多路广播委托时，系统将按照调用列表中的委托出现顺序来同步调用这些委托。

MulticastDelegate具有两个常用属性：Method、Target。其中Method 用于获取委托所表示的方法Target 用于获取当前调用的类实例。

MulticastDelegate有以下几个常用方法：

方法名称

说明

Clone 创建委托的浅表副本。

GetInvocationList 按照调用顺序返回此多路广播委托的调用列表。

GetMethodImpl 返回由当前的 MulticastDelegate 表示的静态方法。

GetObjectData 用序列化该实例所需的所有数据填充 SerializationInfo 对象。

MemberwiseClone 创建当前 Object 的浅表副本。

RemoveImpl 调用列表中移除与指定委托相等的元素

MulticastDelegate与Delegate给委托对象建立了强大的支持，下面向各位详细介绍一下委托的使用方式。

回到目录

三、委托使用方式

3.1 简单的委托

当建立委托对象时，委托的参数类型必须与委托方法相对应。只要向建立委托对象的构造函数中输入方法名称example.Method，委托就会直接绑定此方法。使用myDelegate.Invoke（string message），就能显式调用委托方法。但在实际的操作中，我们无须用到 Invoke 方法，而只要直接使用myDelegate(string message)，就能调用委托方法。

复制代码

1 class Program

2 {

3 delegate void MyDelegate(string message);

4

5 public class Example

6 {

7 public void Method(string message)

8 {

9 MessageBox.Show(message);

10 }

11 }

12

13 static void Main(string[] args)

14 {

15 Example example=new Example();

16 MyDelegate myDelegate=new MyDelegate(example.Method);

17 myDelegate("Hello World");

18 Console.ReadKey();

19 }

20 }

复制代码

3.2 带返回值的委托

当建立委托对象时，委托的返回值必须与委托方法相对应。使用下面的例子，方法将返回 “Hello Leslie” 。

复制代码

1 class Program

2 {

3 delegate string MyDelegate(string message);

4

5 public class Example

6 {

7 public string Method(string name)

8 {

9 return "Hello " + name;

10 }

11 }

12

13 static void Main(string[] args)

14 {

15 Example example=new Example();

16 //绑定委托方法

17 MyDelegate myDelegate=new MyDelegate(example.Method);

18 //调用委托，获取返回值

19 string message = myDelegate("Leslie");

20 Console.WriteLine(message);

21 Console.ReadKey();

22 }

23 }

复制代码

3.3 多路广播委托

在第二节前曾经提过，委托类继承于MulticastDelegate，这使委托对象支持多路广播，即委托对象可以绑定多个方法。当输入参数后，每个方法会按顺序进行迭代处理，并返回最后一个方法的计算结果。

下面的例子中，Price 类中有两个计算方法，Ordinary 按普通的9.5折计算，Favourable 按优惠价 8.5 折计算。委托同时绑定了这两个方法，在输入参数100以后，Ordinary、Favourable这两个方法将按顺序迭代执行下去，最后返回 Favourable 方法的计算结果 85。

复制代码

1 delegate double MyDelegate(double message);

2

3 public class Price

4 {

5 public double Ordinary(double price)

6 {

7 double price1 = 0.95 \* price;

8 Console.WriteLine("Ordinary Price : "+price1);

9 return price1;

10 }

11

12 public double Favourable(double price)

13 {

14 double price1 = 0.85 \* price;

15 Console.WriteLine("Favourable Price : " + price1);

16 return price1;

17 }

18

19 static void Main(string[] args)

20 {

21 Price price = new Price();

22 //绑定Ordinary方法

23 MyDelegate myDelegate = new MyDelegate(price.Ordinary);

24 //绑定Favourable方法

25 myDelegate += new MyDelegate(price.Favourable);

26 //调用委托

27 Console.WriteLine("Current Price : " + myDelegate(100));

28 Console.ReadKey();

29 }

30 }

复制代码

运行结果

3.4 浅谈Observer模式

回顾一下简单的 Observer 模式，它使用一对多的方式，可以让多个观察者同时关注同一个事物，并作出不同的响应。

例如下面的例子，Manager的底薪为基本工资的1.5倍，Assistant的底薪为基本工资的1.2倍。WageManager类的RegisterWorker方法与RemoveWorker方法可以用于注册和注销观察者，最后执行Execute方法可以对多个已注册的观察者同时输入参数。

复制代码

1 public class WageManager

2 {

3 IList<Worker> workerList = new List<Worker>();

4

5 public void RegisterWorker(Worker worker)

6 {

7 workerList.Add(worker);

8 }

9

10 public void RemoveWorker(Worker worker)

11 {

12 workerList.Remove(worker);

13 }

14

15 public void Excute(double basicWages)

16 {

17 if (workerList.Count != 0)

18 foreach (var worker in workerList)

19 worker.GetWages(basicWages);

20 }

21

22 static void Main(string[] args)

23 {

24 WageManager wageManager = new WageManager();

25 //注册观察者

26 wageManager.RegisterWorker(new Manager());

27 wageManager.RegisterWorker(new Assistant());

28 //同时输入底薪3000元，分别进行计算

29 wageManager.Excute(3000);

30

31 Console.ReadKey();

32 }

33 }

34

35 public abstract class Worker

36 {

37 public abstract double GetWages(double basicWages);

38 }

39

40 public class Manager:Worker

41 {

42 //Manager实际工资为底薪1.5倍

43 public override double GetWages(double basicWages)

44 {

45 double totalWages = 1.5 \* basicWages;

46 Console.WriteLine("Manager's wages is " + totalWages);

47 return totalWages;

48 }

49 }

50

51 public class Assistant : Worker

52 {

53 //Assistant实际工资为底薪的1.2倍

54 public override double GetWages(double basicWages)

55 {

56 double totalWages = 1.2 \* basicWages;

57 Console.WriteLine("Assistant's wages is " + totalWages);

58 return totalWages;

59 }

60 }

复制代码

运行结果

开发 Observer 模式时借助委托，可以进一步简化开发的过程。由于委托对象支持多路广播，所以可以把Worker类省略。在WageManager类中建立了一个委托对象wageHandler，通过Attach与Detach方法可以分别加入或取消委托。如果观察者想对事物进行监测，只需要加入一个委托对象即可。记得在第二节曾经提过，委托的GetInvodationList方法能获取多路广播委托列表，在Execute方法中，就是通过去多路广播委托列表去判断所绑定的委托数量是否为0。

复制代码

1 public delegate double Handler(double basicWages);

2

3 public class Manager

4 {

5 public double GetWages(double basicWages)

6 {

7 double totalWages=1.5 \* basicWages;

8 Console.WriteLine("Manager's wages is : " + totalWages);

9 return totalWages;

10 }

11 }

12

13 public class Assistant

14 {

15 public double GetWages(double basicWages)

16 {

17 double totalWages = 1.2 \* basicWages;

18 Console.WriteLine("Assistant's wages is : " + totalWages);

19 return totalWages;

20 }

21 }

22

23 public class WageManager

24 {

25 private Handler wageHandler;

26

27 //加入观察者

28 public void Attach(Handler wageHandler1)

29 {

30 wageHandler += wageHandler1;

31 }

32

33 //删除观察者

34 public void Detach(Handler wageHandler1)

35 {

36 wageHandler -= wageHandler1;

37 }

38

39 //通过GetInvodationList方法获取多路广播委托列表，如果观察者数量大于0即执行方法

40 public void Execute(double basicWages)

41 {

42 if (wageHandler!=null)

43 if(wageHandler.GetInvocationList().Count() != 0)

44 wageHandler(basicWages);

45 }

46

47 static void Main(string[] args)

48 {

49 WageManager wageManager = new WageManager();

50 //加入Manager观察者

51 Manager manager = new Manager();

52 Handler managerHandler = new Handler(manager.GetWages);

53 wageManager.Attach(managerHandler);

54

55 //加入Assistant观察者

56 Assistant assistant = new Assistant();

57 Handler assistantHandler = new Handler(assistant.GetWages);

58 wageManager.Attach(assistantHandler);

59

60 //同时加入底薪3000元，分别进行计算

61 wageManager.Execute(3000);

62 Console.ReadKey();

63 }

64 }

复制代码

最后运行结果与上面的例子相同。

3.5 委托的协变与逆变

在 Framework 2.0 出现之前，委托协变这个概念还没有出现。此时因为委托是安全类型，它们不遵守继承的基础规则。即会这下面的情况：Manager 虽然是 Worker 的子类，但 GetWorkerHander 委托不能直接绑定 GetManager 方法，因为在委托当中它们的返回值 Manager 与 Worker 被视为完全无关的两个类型。

复制代码

1 public class Worker

2 {.......}

3 public class Manager:Worker

4 {.......}

5

6 class Program

7 {

8 public delegate Worker GetWorkerHandler(int id);

9 public delegate Manager GetManagerHandler(int id);

10

11 public static Worker GetWorker(int id)

12 {

13 Worker worker = new Worker();

14 ..............

15 return worker;

16 }

17

18 public static Manager GetManager(int id)

19 {

20 Manager manager = new Manager();

21 ..............

22 return manager;

23 }

24

25 static void Main(string[] args)

26 {

27 GetWorkerHandler workerHandler = new GetWorkerHandler(GetWorker);

28 var worker=workerHandler(1);

29

30 GetManagerHandler managerHandler = new GetManagerHandler(GetManager);

31 var manager = managerHandler(2);

32 Console.ReadKey();

33 }

34 }

复制代码

自从Framework 2.0 面试以后，委托协变的概念就应运而生，此时委托可以按照传统的继承规则进行转换。即 GetWorkerHandler 委托可以直接绑定 GetManager 方法。

复制代码

1 public class Worker

2 {.......}

3 public class Manager:Worker

4 {.......}

5

6 class Program

7 {

8 public delegate Worker GetWorkerHandler(int id);

9 //在 Framework2.0 以上，委托 GetWorkerHandler 可绑定 GetWorker 与 GetManager 两个方法

10

11 public static Worker GetWorker(int id)

12 {

13 Worker worker = new Worker();

14 return worker;

15 }

16

17 public static Manager GetManager(int id)

18 {

19 Manager manager = new Manager();

20 return manager;

21 }

22

23 static void Main(string[] args)

24 {

25 GetWorkerHandler workerHandler = new GetWorkerHandler(GetWorker);

26 Worker worker=workerHandler(1);

27 GetWorkerHandler managerHandler = new GetWorkerHandler(GetManager);

28 Manager manager = managerHandler(2) as Manager;

29 Console.ReadKey();

30 }

31 }

复制代码

委托逆变，是指委托方法的参数同样可以接收 “继承” 这个传统规则。像下面的例子，以 object 为参数的委托，可以接受任何 object 子类的对象作为参数。最后可以在处理方法中使用 is 对输入数据的类型进行判断，分别处理对不同的类型的对象。

复制代码

1 class Program

2 {

3 public delegate void Handler(object obj);

4

5 public static void GetMessage(object message)

6 {

7 if (message is string)

8 Console.WriteLine("His name is : " + message.ToString());

9 if (message is int)

10 Console.WriteLine("His age is : " + message.ToString());

11 }

12

13 static void Main(string[] args)

14 {

15 Handler handler = new Handler(GetMessage);

16 handler(29);

17 Console.ReadKey();

18 }

19 }

复制代码

运行结果

注意：委托与其绑定方法的参数必须一至，即当 Handler 所输入的参数为 A 类型，其绑定方法 GetMessage 的参数也必须为 A 类或者 A 的父类 。相反，当绑定方法的参数为 A 的子类，系统也无法辨认。

3.6 泛型委托

委托逆变虽然实用，但如果都以 object 作为参数，则需要每次都对参数进行类型的判断，这不禁令人感到厌烦。

为此，泛型委托应运而生，泛型委托有着委托逆变的优点，同时利用泛型的特性，可以使一个委托绑定多个不同类型参数的方法，而且在方法中不需要使用 is 进行类型判断，从而简化了代码。

复制代码

1 class Program

2 {

3 public delegate void Handler<T>(T obj);

4

5 public static void GetWorkerWages(Worker worker)

6 {

7 Console.WriteLine("Worker's total wages is " + worker.Wages);

8 }

9

10 public static void GetManagerWages(Manager manager)

11 {

12 Console.WriteLine("Manager's total wages is "+manager.Wages);

13 }

14

15 static void Main(string[] args)

16 {

17 Handler<Worker> workerHander = new Handler<Worker>(GetWorkerWages);

18 Worker worker = new Worker();

19 worker.Wages = 3000;

20 workerHander(worker);

21

22 Handler<Manager> managerHandler = new Handler<Manager>(GetManagerWages);

23 Manager manager = new Manager();

24 manager.Wages = 4500;

25 managerHandler(manager);

26

27 Console.ReadKey();

28 }

29 }

复制代码

运行结果

回到目录

四、深入解析事件

4.1 事件的由来

在介绍事件之前大家可以先看看下面的例子， PriceManager 负责对商品价格进行处理，当委托对象 GetPriceHandler 的返回值大于100元，按8.8折计算，低于100元按原价计算。

复制代码

1 public delegate double PriceHandler();

2

3 public class PriceManager

4 {

5 public PriceHandler GetPriceHandler;

6

7 //委托处理，当价格高于100元按8.8折计算，其他按原价计算

8 public double GetPrice()

9 {

10 if (GetPriceHandler.GetInvocationList().Count() > 0)

11 {

12 if (GetPriceHandler() > 100)

13 return GetPriceHandler()\*0.88;

14 else

15 return GetPriceHandler();

16 }

17 return -1;

18 }

19 }

20

21 class Program

22 {

23 static void Main(string[] args)

24 {

25 PriceManager priceManager = new PriceManager();

26

27 //调用priceManager的GetPrice方法获取价格

28 //直接调用委托的Invoke获取价格，两者进行比较

29 priceManager.GetPriceHandler = new PriceHandler(ComputerPrice);

30 Console.WriteLine(string.Format("GetPrice\n Computer's price is {0}!",

31 priceManager.GetPrice()));

32 Console.WriteLine(string.Format("Invoke\n Computer's price is {0}!",

33 priceManager.GetPriceHandler.Invoke()));

34

35 Console.WriteLine();

36

37 priceManager.GetPriceHandler = new PriceHandler(BookPrice);

38 Console.WriteLine(string.Format("GetPrice\n Book's price is {0}!",

39 priceManager.GetPrice()));

40 Console.WriteLine(string.Format("Invoke\n Book's price is {0}!" ,

41 priceManager.GetPriceHandler.Invoke()));

42

43 Console.ReadKey();

44 }

45 //书本价格为98元

46 public static double BookPrice()

47 {

48 return 98.0;

49 }

50 //计算机价格为8800元

51 public static double ComputerPrice()

52 {

53 return 8800.0;

54 }

55 }

复制代码

运行结果

观察运行的结果，如果把委托对象 GetPriceHandler 设置为 public ，外界可以直接调用 GetPriceHandler.Invoke 获取运行结果而移除了 GetPrice 方法的处理，这正是开发人员最不想看到的。

为了保证系统的封装性，开发往往需要把委托对象 GetPriceHandler 设置为 private, 再分别加入 AddHandler，RemoveHandler 方法对 GetPriceHandler 委托对象进行封装。

复制代码

1 public delegate double PriceHandler();

2

3 public class PriceManager

4 {

5 private PriceHandler GetPriceHandler;

6

7 //委托处理，当价格高于100元按8.8折计算，其他按原价计算

8 public double GetPrice()

9 {

10 if (GetPriceHandler!=null)

11 {

12 if (GetPriceHandler() > 100)

13 return GetPriceHandler()\*0.88;

14 else

15 return GetPriceHandler();

16 }

17 return -1;

18 }

19

20 public void AddHandler(PriceHandler handler)

21 {

22 GetPriceHandler += handler;

23 }

24

25 public void RemoveHandler(PriceHandler handler)

26 {

27 GetPriceHandler -= handler;

28 }

29 }

30 ................

31 ................

复制代码

为了保存封装性，很多操作都需要加入AddHandler、RemoveHandler 这些相似的方法代码，这未免令人感到厌烦。

为了进一步简化操作，事件这个概念应运而生。

4.2 事件的定义

事件（event）可被视作为一种特别的委托，它为委托对象隐式地建立起add\_XXX、remove\_XXX 两个方法，用作注册与注销事件的处理方法。而且事件对应的变量成员将会被视为 private 变量，外界无法超越事件所在对象直接访问它们，这使事件具备良好的封装性，而且免除了add\_XXX、remove\_XXX等繁琐的代码。

1 public class EventTest

2 {

3 public delegate void MyDelegate();

4 public event MyDelegate MyEvent;

5 }

观察事件的编译过程可知，在编译的时候，系统为 MyEvent 事件自动建立add\_MyEvent、remove\_MyEvent 方法。

4.3 事件的使用方式

事件能通过+=和-=两个方式注册或者注销对其处理的方法，使用+=与-=操作符的时候，系统会自动调用对应的 add\_XXX、remove\_XXX 进行处理。

值得留意，在PersonManager类的Execute方法中，如果 MyEvent 绑定的处理方法不为空，即可使用MyEvent（string）引发事件。但如果在外界的 main 方法中直接使用 personManager.MyEvent (string) 来引发事件，系统将引发错误报告。这正是因为事件具备了良好的封装性，使外界不能超越事件所在的对象访问其变量成员。

注意：在事件所处的对象之外，事件只能出现在+=，-=的左方。

此时，开发人员无须手动添加 add\_XXX、remove\_XXX 的方法，就可实现与4.1例子中的相同功能，实现了良好的封装。

复制代码

1 public delegate void MyDelegate(string name);

2

3 public class PersonManager

4 {

5 public event MyDelegate MyEvent;

6

7 //执行事件

8 public void Execute(string name)

9 {

10 if (MyEvent != null)

11 MyEvent(name);

12 }

13 }

14

15 class Program

16 {

17 static void Main(string[] args)

18 {

19 PersonManager personManager = new PersonManager();

20 //绑定事件处理方法

21 personManager.MyEvent += new MyDelegate(GetName);

22 personManager.Execute("Leslie");

23 Console.ReadKey();

24 }

25

26 public static void GetName(string name)

27 {

28 Console.WriteLine("My name is " + name);

29 }

30 }

复制代码

4.4 事件处理方法的绑定

在绑定事件处理方法的时候，事件出现在+=、-= 操作符的左边，对应的委托对象出现在+=、-= 操作符的右边。对应以上例子，事件提供了更简单的绑定方式，只需要在+=、-= 操作符的右方写上方法名称，系统就能自动辩认。

复制代码

1 public delegate void MyDelegate(string name);

2

3 public class PersonManager

4 {

5 public event MyDelegate MyEvent;

6 .........

7 }

8

9 class Program

10 {

11 static void Main(string[] args)

12 {

13 PersonManager personManager = new PersonManager();

14 //绑定事件处理方法

15 personManager.MyEvent += GetName;

16 .............

17 }

18

19 public static void GetName(string name)

20 {.........}

21 }

复制代码

如果觉得编写 GetName 方法过于麻烦，你还可以使用匿名方法绑定事件的处理。

复制代码

1 public delegate void MyDelegate(string name);

2

3 public class PersonManager

4 {

5 public event MyDelegate MyEvent;

6

7 //执行事件

8 public void Execute(string name)

9 {

10 if (MyEvent != null)

11 MyEvent(name);

12 }

13

14 static void Main(string[] args)

15 {

16 PersonManager personManager = new PersonManager();

17 //使用匿名方法绑定事件的处理

18 personManager.MyEvent += delegate(string name){

19 Console.WriteLine("My name is "+name);

20 };

21 personManager.Execute("Leslie");

22 Console.ReadKey();

23 }

24 }

复制代码

4.5 C#控件中的事件

在C#控件中存在多个的事件，像Click、TextChanged、SelectIndexChanged 等等，很多都是通过 EventHandler 委托绑定事件的处理方法的，EventHandler 可说是C#控件中最常见的委托 。

public delegate void EventHandler (Object sender, EventArgs e)

EventHandler 委托并无返回值，sender 代表引发事件的控件对象，e 代表由该事件生成的数据 。在ASP.NET中可以直接通过btn.Click+=new EventHandler(btn\_onclick) 的方式为控件绑定处理方法。

复制代码

1 <html xmlns="http://www.w3.org/1999/xhtml">

2 <head runat="server">

3 <title></title>

4 <script type="text/C#" runat="server">

5 protected void Page\_Load(object sender, EventArgs e)

6 {

7 btn.Click += new EventHandler(btn\_onclick);

8 }

9

10 public void btn\_onclick(object obj, EventArgs e)

11 {

12 Button btn = (Button)obj;

13 Response.Write(btn.Text);

14 }

15 </script>

16 </head>

17 <body>

18 <form id="form1" runat="server">

19 <div>

20 <asp:Button ID="btn" runat="server" Text="Button"/>

21 </div>

22 </form>

23 </body>

24 </html>

复制代码

更多时候，只需要在页面使用 OnClick=“btn\_onclick" 方法，在编译的时候系统就会自动对事件处理方法进行绑定。

复制代码

1 <html xmlns="http://www.w3.org/1999/xhtml">

2 <head runat="server">

3 <title></title>

4 <script type="text/C#" runat="server">

5 public void btn\_onclick(object obj, EventArgs e)

6 {

7 Button btn = (Button)obj;

8 Response.Write(btn.Text);

9 }

10 </script>

11 </head>

12 <body>

13 <form id="form1" runat="server">

14 <div>

15 <asp:Button ID="btn" runat="server" Text="Button" OnClick="btn\_onclick"/>

16 </div>

17 </form>

18 </body>

19 </html>

复制代码

EventHandler 只是 EventHandler<TEventArgs> 泛型委托的一个简单例子。事实上，大家可以利用 EventHandler<TEventArgs> 构造出所需要的委托。

public delegate void EventHandler<TEventArgs> (Object sender, TEventArgs e)

在EventHandler<TEventArgs>中，sender代表事件源，e 代表派生自EventArgs类的事件参数。开发人员可以建立派生自EventArgs的类，从中加入需要使用到的事件参数，然后建立 EventHandler<TEventArgs> 委托。

下面的例子中，先建立一个派生自EventArgs的类MyEventArgs作为事件参数，然后在EventManager中建立事件myEvent , 通过 Execute 方法可以激发事件。最后在测试中绑定 myEvent 的处理方法 ShowMessage，在ShowMessage显示myEventArgs 的事件参数 Message。

复制代码

1 public class MyEventArgs : EventArgs

2 {

3 private string args;

4

5 public MyEventArgs(string message)

6 {

7 args = message;

8 }

9

10 public string Message

11 {

12 get { return args; }

13 set { args = value; }

14 }

15 }

16

17 public class EventManager

18 {

19 public event EventHandler<MyEventArgs> myEvent;

20

21 public void Execute(string message)

22 {

23 if (myEvent != null)

24 myEvent(this, new MyEventArgs(message));

25 }

26 }

27

28 class Program

29 {

30 static void Main(string[] args)

31 {

32 EventManager eventManager = new EventManager();

33 eventManager.myEvent += new EventHandler<MyEventArgs>(ShowMessage);

34 eventManager.Execute("How are you!");

35 Console.ReadKey();

36 }

37

38 public static void ShowMessage(object obj,MyEventArgs e)

39 {

40 Console.WriteLine(e.Message);

41 }

42 }

复制代码

运行结果

4.6 为用户控件建立事件

在ASP.NET开发中，页面往往会出现很多类似的控件与代码，开发人员可以通过用户控件来避免重复的代码。但往往同一个用户控件，在不同的页面中需要有不同的响应。此时为用户控件建立事件，便可轻松地解决此问题。

下面例子中，在用户控件 MyControl 中建立存在一个GridView控件，GridView 控件通过 GetPersonList 方法获取数据源。在用户控件中还定义了 RowCommand 事件，在 GridView 的 GridView\_RowCommand 方法中激发此事件。这样，在页面使用此控件时，开发人员就可以定义不同的方法处理 RowCommand 事件。

复制代码

1 public class Person

2 {

3 public int ID

4 { get; set; }

5 public string Name

6 { get; set; }

7 public int Age

8 { get; set; }

9 }

10

11 <!-- 用户控件 -->

12 <%@ Control Language="C#" AutoEventWireup="true" CodeFile="MyControl.ascx.cs" Inherits="MyControl" %>

13 <script type="text/C#" runat="server">

14 protected void Page\_Load(object sender, EventArgs e)

15 {

16 GridView1.DataSource = GetPersonList();

17 GridView1.DataBind();

18 }

19

20 //绑定数据源

21 protected IList<Person> GetPersonList()

22 {

23 IList<Person> list = new List<Person>();

24 Person person1 = new Person();

25 person1.ID = 1;

26 person1.Name = "Leslie";

27 person1.Age = 29;

28 list.Add(person1);

29 ...........

30 return list;

31 }

32

33 public event GridViewCommandEventHandler RowCommand;

34

35 protected void GridView1\_RowCommand(object sender, GridViewCommandEventArgs e)

36 {

37 if (RowCommand != null)

38 RowCommand(sender, e);

39 }

40 </script>

41 <div>

42 <asp:GridView ID="GridView1" runat="server" AutoGenerateColumns="False"

43 onrowcommand="GridView1\_RowCommand">

44 <Columns>

45 <asp:BoundField DataField="ID" HeaderText="ID"/>

46 <asp:BoundField DataField="Name" HeaderText="Name"/>

47 <asp:BoundField DataField="Age" HeaderText="Age"/>

48 <asp:ButtonField CommandName="Get" Text="Select"/>

49 </Columns>

50 </asp:GridView>

51 </div>

52

53 <!-- 页面代码 -->

54 <%@ Page Language="C#" AutoEventWireup="true" CodeFile="Default2.aspx.cs" Inherits="Default2" %>

55 <%@ Register Src="~/MyControl.ascx" TagPrefix="ascx" TagName="myControl" %>

56 <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

57

58 <html xmlns="http://www.w3.org/1999/xhtml">

59 <head runat="server">

60 <title></title>

61 <script type="text/C#" runat="server">

62 protected void myControl\_RowCommand(object sender, GridViewCommandEventArgs e)

63 {

64 if (e.CommandName == "Get")

65 {

66 GridView gridView=(GridView)sender;

67 int index = int.Parse(e.CommandArgument.ToString());

68 label.Text=gridView.Rows[index].Cells[1].Text;

69 }

70 }

71 </script>

72 </head>

73 <body>

74 <form id="form1" runat="server">

75 <div>

76 <ascx:myControl ID="myControl" runat="server" OnRowCommand="myControl\_RowCommand"></ascx:myControl>

77 <br />

78 Select Name ： <asp:Label ID="label" runat="server"></asp:Label><br />

79 </div>

80 </form>

81 </body>

82 </html>

复制代码

运行结果

使用控件已有的事件固然简单，但它限制了传送的参数类型，使开发人员无法传送额外的自定义参数。在结构比较复杂的用户控件中，使用已有的控件事件，显然不够方便，此时，您可以考虑为用户控件建立自定义事件。

首先用户控件中包含订单信息与订单明细列表，首先定义一个事件参数 MyEventArgs，里面包含了订单信息与一个 OrderItem 数组。然后建立用户控件的委托MyDelegate 与对应的事件 MyEvent，在 Button 的 Click 事件中激发 MyEvent 自定义事件。这样在页面处理方法 myControl\_Click 中就可以通过事件参数 MyEventArgs 获取用户控件中的属性，计算订单的总体价格。

复制代码

1 <!-- 基础类 -->

2 public class OrderItem

3 {

4 public OrderItem(string id,string goods,double price,int count)

5 {

6 this.OrderItemID = id; //明细单ID

7 this.Goods = goods; //商品名称

8 this.Price = price; //商品单价

9 this.Count = count; //商品数量

10 }

11

12 public string OrderItemID

13 { get; set; }

14 public string Goods

15 { get; set; }

16 public double Price

17 { get; set; }

18 public int Count

19 { get; set; }

20 }

21

22 /// 事件参数

23 public class MyEventArgs：EventArgs

24 {

25 public MyEventArgs(string name,string address,string tel,

26 string orderCode,IList<OrderItem> orderItemList)

27 {

28 Name = name; //买家姓名

29 Address = address; //买家地址

30 Tel = tel; //买家电话

31 OrderCode = orderCode; //订单号码

32 OrderItemList = orderItemList; //订单明细

33 }

34

35 public string Name

36 { get;set; }

37 public string Address

38 { get; set; }

39 public string Tel

40 { get; set; }

41 public string OrderCode

42 { get; set; }

43 public IList<OrderItem> OrderItemList

44 { get; set; }

45 }

46

47 <!-- 用户控件 -->

48 <%@ Control Language="C#" AutoEventWireup="true" CodeFile="MyControl.ascx.cs" Inherits="MyControl" %>

49 <script type="text/C#" runat="server">

50 protected void Page\_Load(object sender, EventArgs e)

51 {

52 GridView1.DataSource = GetList();

53 GridView1.DataBind();

54 }

55

56 //模拟数据源

57 protected IList<OrderItem> GetList()

58 {

59 IList<OrderItem> list = new List<OrderItem>();

60 OrderItem orderItem = new OrderItem("1", "Asus N75S", 8800, 2);

61 list.Add(orderItem);

62 ..........

63 return list;

64 }

65

66 //自定义委托

67 public delegate void MyDelegate(object sender,MyEventArgs myEventArgs);

68 //自定义事件

69 public event MyDelegate MyEvent;

70

71 //按下Button时激发自定义事件

72 protected void btn\_click(object sender, EventArgs e)

73 {

74 if (MyEvent != null)

75 {

76 MyEventArgs myEventArgs = new MyEventArgs(labelName.Text, labelAddress.Text, labelTel.Text

77 , labelOrderCode.Text, GetList());

78 MyEvent(this,myEventArgs);

79 }

80 }

81 </script>

82 <div>

83 Name : <asp:Label ID="labelName" runat="server">Leslie</asp:Label><br />

84 Address : <asp:Label ID="labelAddress" runat="server">ZhongShan University 2A 501</asp:Label><br />

85 Tel : <asp:Label ID="labelTel" runat="server">13660123456</asp:Label><br />

86 Order Code : <asp:Label ID="labelOrderCode" runat="server">A12012031223B0030</asp:Label><br /><br />

87 <asp:GridView ID="GridView1" runat="server" AutoGenerateColumns="False" CellPadding="5">

88 <Columns>

89 <asp:BoundField DataField="OrderItemID" HeaderText="ID"/>

90 <asp:BoundField DataField="Goods" HeaderText="Goods"/>

91 <asp:BoundField DataField="Price" HeaderText="Price"/>

92 <asp:BoundField DataField="Count" HeaderText="Count"/>

93 </Columns>

94 </asp:GridView>

95 <br />

96 <asp:Button ID="btn" runat="server" Text="Account" OnClick="btn\_click"/>

97 </div>

98

99 <!-- 页面处理 -->

100 <%@ Page Language="C#" AutoEventWireup="true" CodeFile="Default2.aspx.cs" Inherits="Default2" %>

101 <%@ Register Src="~/MyControl.ascx" TagPrefix="ascx" TagName="myControl" %>

102 <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

103

104 <html xmlns="http://www.w3.org/1999/xhtml">

105 <head runat="server">

106 <title></title>

107 <script type="text/C#" runat="server">

108 //在页面定义用户控件MyEvent事件的处理方法

109 protected void myControl\_Click(object sender,MyEventArgs e)

110 {

111 //计算订单总体价格

112 double totalPrice=0;

113 IList<OrderItem> list=e.OrderItemList;

114 foreach(OrderItem item in list)

115 totalPrice+=item.Price\*item.Count;

116 //展示订单号及总体费用

117 labelOrderCode.Text = e.OrderCode;

118 labelTotalPrice.Text = totalPrice.ToString();

119 }

120 </script>

121 </head>

122 <body>

123 <form id="form1" runat="server">

124 <div>

125 <ascx:myControl ID="myControl" runat="server" OnMyEvent="myControl\_Click"></ascx:myControl>

126 <br />

127 OrderCode ： <asp:Label ID="labelOrderCode" runat="server"></asp:Label><br />

128 TotalPrice : <asp:Label ID="labelTotalPrice" runat="server"></asp:Label>

129 </div>

130 </form>

131 </body>

132 </html>

复制代码

运行结果

若对自定义事件不太熟悉的朋友很多时候会使用 UserControl.FindControl 的方式获取用户控件中的属性，但当你深入了解自定义事件的开发过程以后，就能有效简化开发的过程。

回到目录

五、Lambda 表达式

5.1 Lambda 的意义

在Framework 2.0 以前，声明委托的唯一方法是通过方法命名，从Framework 2.0 起，系统开始支持匿名方法。

通过匿名方法，可以直接把一段代码绑定给事件，因此减少了实例化委托所需的编码系统开销。

而在 Framework 3.0 开始，Lambda 表达式开始逐渐取代了匿名方法，作为编写内联代码的首选方式。总体来说，Lambda 表达式的作用是为了使用更简单的方式来编写匿名方法，彻底简化委托的使用方式。

5.2 回顾匿名方法的使用

匿名方法的使用已经在4.4节简单介绍过，在此回顾一下。

使用下面的方式，可以通过匿名方法为Button的Click事件绑定处理方法。

复制代码

1 static void Main(string[] args)

2 {

3 Button btn = new Button();

4 btn.Click+=delegate(object obj,EventArgs e){

5 MessageBox.Show("Hello World !");

6 };

7 }

复制代码

总是使用 delegate（）{......} 的方式建立匿名方法，令人不禁感觉郁闷。于是从Framework 3.0 起， Lambda 表达式开始出现。

5.3 简单介绍泛型委托

在介绍 Lambda 表达式前，先介绍一下常用的几个泛型委托。

5.3.1 泛型委托 Predicate<T>

早在Framework 2.0 的时候，微软就为 List<T> 类添加了 Find、FindAll 、ForEach 等方法用作数据的查找。

public T Find ( Predicate<T> match）

public List<T> FindAll（Predicate<T> match）

在这些方法中存在一个Predicate <T> 表达式，它是一个返回bool的泛型委托，能接受一个任意类型的对象作为参数。

public delegate bool Predicate<T>（T obj）

在下面例子中，Predicate 委托绑定了参数为Person类的方法Match作为查询条件，然后使用 FindAll 方法查找到合适条件的 List<Person> 集合。

复制代码

1 class Program

2 {

3 static void Main(string[] args)

4 {

5 List<Person> list = GetList();

6 //绑定查询条件

7 Predicate<Person> predicate = new Predicate<Person>(Match);

8 List<Person> result = list.FindAll(predicate);

9 Console.WriteLine（“Person count is : ” + result.Count);

10 Console.ReadKey();

11 }

12 //模拟源数据

13 static List<Person> GetList()

14 {

15 var personList = new List<Person>();

16 var person1 = new Person(1,"Leslie",29);

17 personList.Add(person1);

18 ........

19 return personList;

20 }

21 //查询条件

22 static bool Match(Person person)

23 {

24 return person.Age <= 30;

25 }

26 }

27

28 public class Person

29 {

30 public Person(int id, string name, int age)

31 {

32 ID = id;

33 Name = name;

34 Age = age;

35 }

36

37 public int ID

38 { get; set; }

39 public string Name

40 { get; set; }

41 public int Age

42 { get; set; }

43 }

复制代码

5.3.2 泛型委托 Action

Action<T> 的使用方式与 Predicate<T> 相似，不同之处在于 Predicate<T> 返回值为 bool , Action<T> 的返回值为 void。

Action 支持0~16个参数，可以按需求任意使用。

public delegate void Action（）

public delegate void Action<T1>（T1 obj1）

public delegate void Action<T1，T2> （T1 obj1, T2 obj2）

public delegate void Action<T1，T2，T3> （T1 obj1, T2 obj2，T3 obj3）

............

public delegate void Action<T1，T2，T3，......，T16> （T1 obj1, T2 obj2，T3 obj3，......，T16 obj16）

复制代码

1 static void Main(string[] args)

2 {

3 Action<string> action=ShowMessage;

4 action("Hello World");

5 Console.ReadKey();

6 }

7

8 static void ShowMessage(string message)

9 {

10 MessageBox.Show(message);

11 }

复制代码

5.3.3 泛型委托 Func

委托 Func 与 Action 相似，同样支持 0~16 个参数，不同之处在于Func 必须具有返回值

public delegate TResult Func<TResult>（）

public delegate TResult Func<T1,TResult>（T1 obj1）

public delegate TResult Func<T1,T2,TResult>（T1 obj1,T2 obj2）

public delegate TResult Func<T1,T2,T3,TResult>（T1 obj1,T2 obj2,T3 obj3）

............

public delegate TResult Func<T1,T2,T3,......,T16,TResult>（T1 obj1,T2 obj2,T3 obj3,......,T16 obj16）

复制代码

1 static void Main(string[] args)

2 {

3 Func<double, bool, double> func = Account;

4 double result=func(1000, true);

5 Console.WriteLine("Result is : "+result);

6 Console.ReadKey();

7 }

8

9 static double Account(double a,bool condition)

10 {

11 if (condition)

12 return a \* 1.5;

13 else

14 return a \* 2;

15 }

复制代码

5.4 揭开 Lambda 神秘的面纱

Lambda 的表达式的编写格式如下：

x=> x \* 1.5

当中 “ => ” 是 Lambda 表达式的操作符，在左边用作定义一个参数列表，右边可以操作这些参数。

例子一, 先把 int x 设置 1000，通过 Action 把表达式定义为 x=x+500 ，最后通过 Invoke 激发委托。

复制代码

1 static void Main(string[] args)

2 {

3 int x = 1000;

4 Action action = () => x = x + 500;

5 action.Invoke();

6

7 Console.WriteLine("Result is : " + x);

8 Console.ReadKey();

9 }

复制代码

例子二，通过 Action<int> 把表达式定义 x=x+500， 到最后输入参数1000，得到的结果与例子一相同。

注意，此处Lambda表达式定义的操作使用 { } 括弧包括在一起，里面可以包含一系列的操作。

复制代码

1 static void Main(string[] args)

2 {

3 Action<int> action = (x) =>

4 {

5 x = x + 500;

6 Console.WriteLine("Result is : " + x);

7 };

8 action.Invoke(1000);

9 Console.ReadKey();

10 }

复制代码

例子三，定义一个Predicate<int>，当输入值大约等于1000则返回 true , 否则返回 false。与5.3.1的例子相比，Predicate<T>的绑定不需要显式建立一个方法，而是直接在Lambda表达式里完成，简洁方便了不少。

复制代码

1 static void Main(string[] args)

2 {

3 Predicate<int> predicate = (x) =>

4 {

5 if (x >= 1000)

6 return true;

7 else

8 return false;

9 };

10 bool result=predicate.Invoke(500);

11 Console.ReadKey();

12 }

复制代码

例子四，在计算商品的价格时，当商品重量超过30kg则打9折，其他按原价处理。此时可以使用Func<double,int,double>，参数1为商品原价，参数2为商品重量，最后返回值为 double 类型。

复制代码

1 static void Main(string[] args)

2 {

3 Func<double, int, double> func = (price, weight) =>

4 {

5 if (weight >= 30)

6 return price \* 0.9;

7 else

8 return price;

9 };

10 double totalPrice = func(200.0, 40);

11 Console.ReadKey();

12 }

复制代码

例子五，使用Lambda为Button定义Click事件的处理方法。与5.2的例子相比，使用Lambda比使用匿名方法更加简单。

复制代码

1 static void Main(string[] args)

2 {

3 Button btn = new Button();

4 btn.Click += (obj, e) =>

5 {

6 MessageBox.Show("Hello World!");

7 };

8 Console.ReadKey();

9 }

复制代码

例子六，此处使用5.3.1的例子，在List<Person>的FindAll方法中直接使用Lambda表达式。

相比之下，使用Lambda表达式，不需要定义Predicate<T>对象，也不需要显式设定绑定方法，简化了不工序。

复制代码

1 class Program

2 {

3 static void Main(string[] args)

4 {

5 List<Person> personList = GetList();

6

7 //查找年龄少于30年的人

8 List<Person> result=personList.FindAll((person) => person.Age =< 30);

9 Console.WriteLine("Person count is : " + result.Count);

10 Console.ReadKey();

11 }

12

13 //模拟源数据

14 static List<Person> GetList()

15 {

16 var personList = new List<Person>();

17 var person1 = new Person(1,"Leslie",29);

18 personList.Add(person1);

19 .......

20 return personList;

21 }

22 }

23

24 public class Person

25 {

26 public Person(int id, string name, int age)

27 {

28 ID = id;

29 Name = name;

30 Age = age;

31 }

32

33 public int ID

34 { get; set; }

35 public string Name

36 { get; set; }

37 public int Age

38 { get; set; }

39 }

复制代码

当在使用LINQ技术的时候，到处都会弥漫着 Lambda 的身影，此时更能体现 Lambda 的长处。

但 LINQ 涉及到分部类，分部方法，IEnumerable<T>，迭代器等多方面的知识，这些已经超出本章的介绍范围。

通过这一节的介绍，希望能够帮助大家更深入地了解 Lambda 的使用。

回到目录

本章小结

本章主要介绍了委托（Delegate）的使用，委托对象是一个派生自 System.MultcastDelegate 的类，它能通过 Invoke 方式进行同步调用，也可以通过 BeginInvoke，EndInvoke 方式实现异步调用。而事件（Event）属于一种特殊的委托，它与委托类型同步使用，可以简化的开发过程。

最后，本文还介绍了匿名方法的使用方式，以及 Lambda 表达式的由来。

对 .NET 开发有兴趣的朋友欢迎加入QQ群：230564952 共同探讨 ！

C#综合揭秘

通过修改注册表建立Windows自定义协议

Entity Framework 并发处理详解

细说进程、应用程序域与上下文

细说多线程（上）

细说多线程（下）

细说事务

深入分析委托与事件

作者：风尘浪子

http://www.cnblogs.com/leslies2/archive/2012/03/22/2389318.html

原创作品，转载时请注明作者及出处

分类: C#综合揭秘

标签: .NET, C#, 委托Delegate, 事件Event, Lambda表达式, 匿名方法, 自定义事件, 委托的协变与逆变, 观察者模式Observer, 泛型委托Action、Predicate、Func

好文要顶 关注我 收藏该文

风尘浪子

关注 - 11

粉丝 - 1719

荣誉：推荐博客

+加关注

75

0

« 上一篇：C#综合揭秘——细说进程、应用程序域与上下文之间的关系

» 下一篇：.NET基础篇——Entity Framework 数据转换层通用类

posted on 2012-03-22 11:04 风尘浪子 阅读(51119) 评论(42) 编辑 收藏

评论

#1楼 2012-03-22 11:40 楼上那个蜀黍

不错不错，学习了！

支持(0)反对(0)

#2楼 2012-03-22 11:42 火禾

这个话题虽然很多人说过了，但是常说常新，楼主讲的十分好，即使已经理解这些的人再读读也是好的！

支持(0)反对(0)

#3楼 2012-03-22 12:58 lonely\_rain

推荐，学习！

支持(0)反对(0)

#4楼 2012-03-22 14:11 LEX123

好长哦

支持(0)反对(0)

#5楼 2012-03-22 15:01 TimYang

你的这几个系列度不错的！慢慢来学习！

支持(0)反对(0)

#6楼 2012-03-22 15:58 Garnett

顶一下,lz辛苦了.

写的这么详细

支持(0)反对(0)

#7楼 2012-03-22 16:35 jhlong

写的很详细，慢慢学习

支持(0)反对(0)

#8楼 2012-03-22 17:48 5207

楼主写的很详细，讲的很到位。我一直是委托、事件傻傻分清的主，这下有些明白了

支持(0)反对(0)

#9楼 2012-03-22 17:51 xu\_happy\_you

写得很好，现在看时都看懂了，用还是不会，以后多用用就更熟悉了！

支持(1)反对(0)

#10楼 2012-03-23 09:52 kevin0227

写了很不错，楼主继续加油，收藏了

支持(0)反对(0)

#11楼 2012-03-23 10:08 深蓝医生

Mark,收藏.

支持(0)反对(0)

#12楼 2012-03-23 10:22 咖啡色

写的很好，强烈推荐。

支持(0)反对(0)

#13楼 2012-03-23 16:11 xinsiyus

呵呵，看完了，多谢分享。

支持(0)反对(0)

#14楼 2012-03-25 21:56 随意随意

楼主 强大 收藏

支持(0)反对(0)

#15楼 2012-03-29 11:30 AceYue

看到这篇文章对委托和事件的理解更深了，而且还明白了“=>”的意义。

支持(0)反对(0)

#16楼 2012-03-30 21:22 夜の魔王

好长啊，楼主真是有心人。赞

支持(0)反对(0)

#17楼 2012-03-31 23:09 零点漂移

顶起！！！！！！！！！！！

支持(0)反对(0)

#18楼 2012-06-07 23:16 MSSQL123

非常好

支持(0)反对(0)

#19楼 2012-07-08 22:25 6572789

收藏学习

支持(0)反对(0)

#20楼 2012-08-02 16:48 browser\_yy

继续学习，谢谢楼主分享！

支持(0)反对(0)

#21楼 2012-08-26 11:01 mchenx

引用注意：委托与其绑定方法的参数必须一至，即当 Handler 所输入的参数为 object 类型，其绑定方法 GetMessage 的参数也必须为 object 。否则，即使绑定方法的参数为 object 的子类，系统也无法辨认。

写的不错，温故而知新啊。我同意你举的例子，但“委托与其绑定方法的参数必须一至”这个说法是有问题的，相反，如果有多个绑定方法，并且绑定方法的参数既有object也有object的子类,那么Handler参数必须为子类，必须是最具体的那个子类。

支持(0)反对(0)

#22楼[楼主] 2012-08-27 11:48 风尘浪子

@ mchenx

我想可能是我的解释不够清楚，令阁下误会了。比如在下面的方法当中，Manager是Person的子类，但即使在委托当中输入的方法参数为Person类，在绑定方法中也不能以Manager类声明方法参数，否则系统是无法辨认参数而释放出“方法与重载不匹配”的错误。

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

public class Program

{

public class Person

{ }

public class Manager : Person

{ }

delegate void MyDelegate(Person person);

public static void Main(string[] args)

{

Program program = new Program();

MyDelegate myDelegate = new MyDelegate(program.GetManager);

Console.ReadKey();

}

public void GetManager(Manager manager)

{ ...... }

}

所以正确的做法是以Person声明参数，而在方法内部使用类型转换。如果希望在方法使用子类声明参数，可以用泛型来声明参数类型，有关资料可参考第三节：委托使用方式

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

public class Program

{

public class Person

{ }

public class Manager : Person

{ }

delegate void MyDelegate(Person person);

public static void Main(string[] args)

{

Program program = new Program();

MyDelegate myDelegate = new MyDelegate(program.GetManager);

Console.ReadKey();

}

public void GetManager(Person person)

{

Manager manger=(Manager) person;

.........

}

}

支持(2)反对(0)

#23楼 2012-08-27 18:45 mchenx

@ 风尘浪子

引用@mchenx

我想可能是我的解释不够清楚，令阁下误会了。比如在下面的方法当中，Manager是Person的子类，但即使在委托当中输入的方法参数为Person类，在绑定方法中也不能以Manager类声明方法参数，否则系统是无法辨认参数而释放出“方法与重载不匹配”的错误。

所以正确的做法是以Person声明参数，而在方法内部使用类型转换。如果希望在方法使用子类声明参数，可以用泛型来声明参数类型，有关资料可参考第三节：委托使用方式

谢谢楼主的细心解答，大概是我钻牛角尖了，或者我们说的不是同一个问题。我想你的意思是在delegate已经被确定的情况下，再去定义委托方法，因为使用了object类型作为委托参数，所以委托方法也必须为object类型（因为找不到Object的超类了）。而我说的是先有方法，再定义委托，比如以你说的Person和Manager为例，有如下几个方法，

void func1(Person value)

void func2(Manager value) ,

或者还有void func3(object value)，

那么这三个都可以绑定到

delegate void MyDelegate(Manager value)上，也就是如下代码都合法：

MyDelegate d1 = func1;

MyDelegate d2 = func2;

MyDelegate d3 = func3;

原因是Object和Person都是Manager的超类，这就是“逆变”，体现在签名的向上兼容性上（在使用时，体现为传入的是具体的子类的对象）。你给出来的例子，签名上都一样，不同的只是传入的对象类型不一样。

最典型的例子就是WinForm的事件,比如KeyDown, DragOver，这些事件的委托方法其实需要KeyEventArgs和DragEventArgs类型的参数，但通过委托逆变，可以定义一个EventArgs参数的方法，然后绑定到这些事件上来做统一处理，KeyDown事件传入KeyEventArgs对象，DragOver事件传入DragEventArgs对象。

void MyHandler(object sender, EventArgs e) {...}

control.KeyDown += MyHandler;

control.DragOver += MyHandler;

如果没有委托逆变的特性，以上两行代码会出错。

以上是我的理解，如有不当，望楼主指教

支持(2)反对(0)

#24楼[楼主] 2012-08-28 13:59 风尘浪子

@mchenx

不好意思，可能本人误会了阁下的意思，其实我与你的观点没有冲突，只是从不同的方式的表达。阁下是将子类绑定到委托当中，然后在处理方法中使用父类作为输入参数，这是可行。而本人的例子指当委托绑定其父类时，方法中使用子类定义参数就会引起错误。

谢谢您的意见，我想您的例子更能体现“逆变”的意义。

支持(1)反对(0)

#25楼 2012-10-31 07:59 xwchen1

感谢LZ分享；好文已推荐！

我看LZ和mchenx的讨论，感觉应该是OO中父类，子类AS或者转型的方法，子类永远可以向上转型为父类丢失子类扩展的东东，声明为父类，创建时强制转换为子类构造则此时引用的肯定是子类实例

支持(0)反对(0)

#26楼 2012-12-11 03:17 chelseajcole

有一个小bug, 在最后一个例子中：

List<Person> result=personList.FindAll((person) => person.Age =< 30);

错了一个小于等于的运算符，应该是<=

支持(0)反对(0)

#27楼 2012-12-11 03:18 chelseajcole

一样的@xu\_happy\_you

引用写得很好，现在看时都看懂了，用还是不会，以后多用用就更熟悉了！

一样的感觉

支持(1)反对(0)

#28楼 2014-05-01 21:29 Louy

确实写得好

支持(0)反对(0)

#29楼 2014-05-13 10:17 微软一点都不软

good nice.

支持(0)反对(0)

#30楼 2014-08-15 11:12 天朝程序员

写的真的很好

支持(0)反对(0)

#31楼 2015-07-08 16:57 我是大方

可以把委托写在public class Program吗

支持(0)反对(0)

#32楼[楼主] 2015-07-09 13:06 风尘浪子

@ 我是大方

可以的

支持(0)反对(0)

#33楼 2015-08-21 11:37 Mr.Walker

表示感谢和敬佩啊！

支持(0)反对(0)

#34楼 2016-02-24 16:52 崎岖行者

博主从简单委托到具有返回值的简单委托，再到多播委托、事件、委托的协变和逆变，再引出泛型委托。从引入问题到解决问题，一路娓娓道来，由浅入深，引人入胜。非有深厚的技术功底者不能胜任之，牛人一枚。

支持(0)反对(0)

#35楼 2016-04-11 14:31 行成于思

把因果关系和来龙去脉分析得很透彻。

支持(0)反对(0)

#36楼 2016-06-14 14:58 马战强777

很好。。。

支持(0)反对(0)

#37楼 2016-10-28 14:27 benarvin

报告博主：委托的GetInvodationList方法获取委托数量是.length不是.count

报告完毕

支持(0)反对(0)

#38楼 2016-11-09 17:18 邪心魔佛一页书

先Mark一下，后续再拜读

支持(0)反对(0)

#39楼 2016-12-05 22:22 哈德森

以前只知道事件是一种功能被限制的特殊委托，不知道什么情况下用事件，认认真真敲了博主的代码后，明白了什么时候要用事件了，谢谢博主~~~

支持(0)反对(0)

#40楼 2017-04-11 11:34 su19890510

//加入观察者

public void Attach(Handler wageHandler1)

{

wageHandler += wageHandler1;

}

对于delegate 第一个怎能直接用+= 呢？ 应该从第二个开始吧

如果是event的话是可以的

支持(0)反对(0)

#41楼 2017-07-06 17:20 神器在手天下我有

mark

支持(0)反对(0)

#42楼 2017-08-12 15:33 FZW19940904

都是大神

支持(0)反对(0)

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。

【推荐】50万行VC++源码: 大型组态工控、电力仿真CAD与GIS源码库

【推荐】极光开发者服务平台，五大功能一站集齐

【推荐】腾讯云域名+云解析 限时折扣抓紧抢购

【推荐】阿里云“全民云计算”优惠升级

【推荐】一小时搭建人工智能应用，让技术更容易入门

阿里云0830

最新IT新闻:

· 深圳再诞生一个腾讯有多难？

· 阿里影业上半年收入10.62亿元，同比增幅逾三倍

· 谷歌助手将与家电设备整合 语音控制洗衣机不是梦

· 罗永浩“得到”停更事件，我看见“知识付费”的三个坑

· 支付宝接入南非10000家商户

» 更多新闻...

极光0801

最新知识库文章:

· 做到这一点，你也可以成为优秀的程序员

· 写给立志做码农的大学生

· 架构腐化之谜

· 学会思考，而不只是编程

· 编写Shell脚本的最佳实践

» 更多知识库文章...

Powered by:

博客园

Copyright © 风尘浪子

导航

博客园

首页

联系

订阅订阅

管理

统计

随笔 - 45

文章 - 0

评论 - 958

引用 - 0

公告

个人简介

02年毕业于中山大学物理系，专门从事.NET、JAVA的项目开发，研究大型企业系统架构。对领域驱动设计DDD、面向服务架构SOA、分布式开发、.NET与JAVA的相互调用等方面有着深厚的兴趣。本博客中的文章皆属原创，转载时请注明出处。

.NET高级编程 Java高级编程

技术交流

欢迎加入以下小组共同探讨

博客园小组：

.NET高级编程

QQ群：

.NET 高级编程 230564952

JAVA 高级编程 174850571

昵称：风尘浪子

园龄：8年3个月

荣誉：推荐博客

粉丝：1719

关注：11

+加关注

搜索

随笔分类(50)

.NET基础篇(6)

Android开发笔记(3)

ASP.NET基础篇(3)

C#综合揭秘(7)

Javascript摘要(1)

Java与.NET的相互调用(3)

Java远程通讯技术(2)

SOA面向服务架构(3)

WCF揭秘(5)

WF与Web服务的相互调用(4)

程序人生(3)

服务类工具的应用与管理(3)

利用远程对象实现分布式开发(3)

软件工具(3)

项目管理(1)

Android开发笔记

Cordova 5.3 搭建 Android 开发环境

深入剖释 Android Service 服务

通过CordovaPlugin插件调用 Activity 实例

C#综合揭秘

Entity Framework 并发处理详解

深入分析委托与事件

通过修改注册表建立Windows自定义协议

细说多线程（上）

细说多线程（下）

细说进程、应用程序域与上下文

细说事务

JAVA与.NET的相互调用

TCP/IP相互调用基本架构

利用JNBridge实现远程通讯

通过Web服务实现相互调用

JAVA远程通信技术

JAVA RMI远程方法调用简单实例

JAVA远程通信技术——Axis实战

SOA面向服务架构

SOA的概念

SOA面向服务架构设计

结合领域驱动设计的SOA分布式架构

程序人生

反流程升职记

分享8年开发经验，明确自己发展方向

积分与排名

积分 - 117652

排名 - 2219

最新评论

1. Re:数字证书应用综合揭秘（包括证书生成、加密、解密、签名、验签）

不错，支持下

--小猿人

2. Re:数字证书应用综合揭秘（包括证书生成、加密、解密、签名、验签）

mark

--q269384828

3. Re:C#综合揭秘——细说多线程（下）

楼主我想问个问题，在异步WebRequest例子里， 我运行ashx文件，这条语句会因为数组超过大小而报错，导致我运行不了服务器：int id = int.Parse(condition.Split(......

--BlueDr

4. Re:C#综合揭秘——细说进程、应用程序域与上下文之间的关系

感谢分享，受益匪浅

--BlueDr

5. Re:软件项目管理流程总结

tt

--wxtstar