如何调用一个函数？

void fun(fp1,fp2...)

1,直接调用

fun(ap1,ap2...);

2,委托（函数指针）

delete void DelFun(fp1,fp2...)

DelFun delfun = new DelFun(fun);

delfun(ap1,ap2...);

将一个函数A作为另外一个函数B的参数

B的形参是：委托 + 函数A的参数

class Program

{

static int Add(int x,int y)

{

return x + y;

}

static int Sub(int x,int y)

{

return x - y;

}

delegate int DelegateCalculator(int x, int y);

static void Test(DelegateCalculator delCalculator,int x,int y)

{

Console.WriteLine(delCalculator(x, y));

}

static void Main(string[] args)

{

Test(Sub, 100, 72);

}

}

对象A的函数a执行到某一业务逻辑点，触发对象B的函数b

方法1，直接调用

缺点：

1，类A需要定义一个类B的成员变量,，且构造函数还要初始化该成员变量。

2，类A要是触发多个类B对象，那么需要定义多个类B的成员变量

3，类B的函数b被删除或形参变化，都需要修改A的函数a的内容；

例子：厨房有一个煤气泄漏监视器，当检测到空气中CO浓度超过0.01就报警。

class Air

{

public Monitor monitor;

public Air(Monitor monitor)

{

this.monitor = monitor;

}

private double rateOfCO = 0.2;

public void CheckRateOfCO()

{

if (rateOfCO > 0.01)

{

monitor.Alarm();

}

}

}

class Monitor

{

public void Alarm()

{

Console.WriteLine("煤气泄露了！");

}

}

class Program

{

static void Main(string[] args)

{

Monitor monitor = new Monitor();

Air air = new Air(monitor);

air.CheckRateOfCO();

}

}

方法2：

类A声明一个与将要调用的函数的类型一致的委托，实例化一个委托变量。

在触发调用点

if(委托变量 != null) {

委托变量；

}

类B的不需要定义一个类A的成员变量，但是要在其构造函数中传入类A的引用，并注册其委托；

优点：

1，不必定义一个成员变量

2，类B的函数被删除，类A不会产生bug；类B的函数接口发生变化，只需要去修改类A的委托类型。

缺点：同时触发多个对象的方法，但是如果各个对象的方法不同，那么类A需要定义多个类型的委托和委托变量。

class Air

{

public delegate void AlarmDelegateHandler();

public AlarmDelegateHandler DelAlarm;

private double rateOfCO = 0.2;

public void CheckRateOfCO()

{

if (rateOfCO > 0.01)

{

if (DelAlarm != null)

{

DelAlarm();

}

}

}

}

class Monitor

{

public Monitor(Air air)

{

air.DelAlarm += Alarm;

}

public void Alarm()

{

Console.WriteLine("煤气泄露了！");

}

}

class Program

{

static void Main(string[] args)

{

Air air = new Air();

Monitor monitor = new Monitor(air);

air.CheckRateOfCO();

}

}

事件处理机制

事件的本质

事件是软件系统里的两个子系统之间，或者两个模块之间，或者两个对象之间发送消息，并处理消息的过程。在面向对象的世界里，就可以统一认为是两个对象之间的行为。

两个对象之间发送的这种消息，对发送方来讲是产生一个事件，对接受方来讲是需要处理某个事件。这种消息可以是用户操作产生的或者软件系统里的某个对象产生的。

事件发送者（执行到某一业务逻辑点，触发函数）

事件捕获者（被调用）

逻辑：发送者只管发送事件，接收者只管实现事件处理程序，然后订阅事件。

事件相比委托的优点：发布者只定义一种委托类型即可，能满足所有的订阅者的方法。订阅者只需要注册即可，订阅者的任何改变，不会影响到发布者一丝一毫。

缺点：订阅者的订阅方法参数是固定的（object obj , EventArgs e）

例子：气象局播报天气是发布者；农民和商人是订阅者

class InfoEventArgs : EventArgs

{

public string weather;

public int days;

public InfoEventArgs(string weather,int days)

{

this.weather = weather;

this.days = days;

}

}

class WeatherBureau

{

public EventHandler<InfoEventArgs> eventHandler;

public void Declare(string weather,int days)

{

Console.WriteLine("19:30到了，气象局主持人上班，开始播报天气...");

Console.WriteLine("明天天气是{0},将持续{1}天！",weather,days);

if(eventHandler != null)

{

eventHandler(this, new InfoEventArgs(weather,days));

}

}

}

class Farmer

{

public Farmer(WeatherBureau wb)

{

wb.eventHandler += DoSomething;

}

public void DoSomething(object obj,InfoEventArgs e)

{

if(e.weather == "雨" && e.days > 3)

{

Console.WriteLine("把刚收割的麦子拢一起，装袋入仓，天晴后再晒！");

}

if (e.weather == "雨" && e.days < 3)

{

Console.WriteLine("把刚收割的麦子拢一起,用塑料布盖着就行了");

}

}

}

class Businessman

{

public Businessman(WeatherBureau wb)

{

wb.eventHandler += DoSomething;

}

public void DoSomething(object obj,InfoEventArgs e)

{

if(e.weather == "雨" && e.days < 2)

{

Console.WriteLine("关门休息,在店里刷剧,等天晴开门！");

}

if(e.weather == "雨" && e.days > 2)

{

Console.WriteLine("关门回家，天晴了再来店里开门做生意!");

}

}

}

class Program

{

static void Main(string[] args)

{

WeatherBureau wb = new WeatherBureau();

Farmer f = new Farmer(wb);

Businessman b = new Businessman(wb);

wb.Declare("晴",10);

wb.Declare("雨", 6);

wb.Declare("雨", 1);

}

}

事件发布者发布的事件在实质上可以看成对外提供的回调函数指针列表。这个列表的容量可以动态增长。事件订阅者可以把自己的事件注册到这个列表或者撤销注册，但是它从原则上无法更改或者对其它订阅者的注册产生影响。事件发布者通过两种手段使得订阅者正确地使用事件机制：一是定义一种delegate委托类型，事件订阅者只能按照这种类型定义事件的处理方法；二是定义与这个委托相关的event对象，使得订阅者只负责注册和撤销自己的处理过程而不能随意对别人的处理过程产生影响。

从运行结果和reader2对象把同一个处理方法注册了两次的前提可以看到，对于一个事件，同一个订阅者可以把同一个处理过程注册多次，而这个方法最终也会被执行多次。

执行事件订阅列表中方法的顺序不能被保证；而且，在这里采用的是同步调用方法，只有一个响应函数执行完毕，其它函数才会被执行。如果要方法不被阻塞（包括这里的等待用户输入等），就需要采用异步调用方式。

声明一个委托：定义一种函数类型（定义函数指针能指向的函数类型）【所以委托是类型安全的】

委托是继承于System.Delegate的类，类中有一个调用列表，列表中包含着被委托函数的引用。

向一个委托中注册多个函数，叫做多播委托。函数必须是void类型。

匿名函数初始化委托： 外层变量被匿名函数捕获，生存周期延长至委托被销毁。

委托就像一个仓库，仓库里面存储了很多方法，这个仓库可以存储任意类的任意方法。通过调用仓库可以运行所有方法。

泛型委托

自定义委托

namespace DelegateT {

//定义泛型委托

public delegate T1 Fun<T1,T2>(T2 t2);

class Program {

static void Main(string[] args) {

//实例化委托

Fun<bool, int> fun = new Fun<bool, int>(Check);

//调用委托

Console.WriteLine(fun(100));

}

static bool Check(int tar) {

return tar % 2 == 0 ? true : false;

}

}

}

内置泛型委托

C#共有三种内置泛型委托

1、Action<T> 委托必须是无返回值的方法，

Action<string> \_action = new Action<string>(函数名);

\_action(函数形参);

2、Fun<TResult> 委托必须有返回值的方法

Func<int, bool> \_func = new Func<int, bool>(函数名);

\_func(5);

3、Predicate: 委托必须是固定返回一个bool值，有形参的方法，该委托通常引用一个"判断条件函数"。

Predicate<int> \_predicate = new Predicate<int>(Check);

//使用Lambda表达式

Predicate<int> predicate = p => p % 2 == 0;

\_predicate(26);

委托的应用场景

1，A类的方法a要调用B类的方法b和C类的方法c

* 1. B和C作为A的的成员变量
  2. B和C作为方法a的形参
  3. a的形参是委托（类型符合b和c的签名）
  4. 定义一个委托作为A的成员变量

**用（四）实现**

/\*实验室有 灯，空调，电脑 三种设备

\* 一个控制器，它的turnoff方法 能够一次性 关闭实验室的三种设备

\*/

namespace DelegateDemo1

{ //灯

class Bulb

{

public void TurnOffBulb()

{

Console.WriteLine("灯关了");

}

}

//空调

class AirCondition

{

public void TurnOffAirCondition()

{

Console.WriteLine("空调关了");

}

}

//电脑

class Computer

{

public void TurnOffComputer(int num)

{

Console.WriteLine("关闭了" + num + "台电脑");

}

}

//控制器

class Controller

{

//委托

public delegate void TurnOffSomething();

private TurnOffSomething tos;

//向委托加入方法

public void Add(TurnOffSomething tos)

{

this.tos += tos;

}

//从委托去除方法

public void Remove(TurnOffSomething tos)

{

this.tos -= tos;

}

//关闭功能

public void TurnOff() {

this.tos();

}

}

class Program

{

static void Main(string[] args)

{

Bulb bulb = new Bulb();

AirCondition airCondition = new AirCondition();

Computer computer = new Computer();

Controller controller = new Controller();

controller.Add(bulb.TurnOffBulb);

controller.Add(airCondition.TurnOffAirCondition);

//lambda表达式

controller.Add(() => { computer.TurnOffComputer(10); });

//调用委托

controller.TurnOff();

}

}

}

**”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。“**

委托一般都使用在 Observer模式（观察者模式）。

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模式主要包括如下两类对象：

被监视对象：往往包含着其他对象所感兴趣的内容。

监视者：当对象中的某件事发生的时候，会告知建设者，而建设者则会采取相应的行动。

**八、利用Func委托代码优化**

在项目开发过程中经常会看到类似的代码：

try  
            {  
                Do();  
            }  
            catch (Exception ex)  
            {  
                LogException(ex);  
            }  
            finally  
            {  
                DoFinally();  
            }

造成代码量的冗余，给日后代码维护带来很多的不便。

有很多种方法可以实现，例如：AOP、委托等。在这里我们主要讲如何利用Func委托来实现代码优化。

   
        private void CallMethod(Func<string> func)  
        {  
            try  
            {  
                func();  
            }  
            catch (Exception ex)  
            {  
                LogException(ex);  
            }  
            finally  
            {  
                DoFinally();  
            }  
        }

CallMethod(new Func<string>(Do));

我们将方法作为委托进行传入，这样节省了很多的冗余代码。