

C#基础系列：委托和设计模式（2） - 文章 - 伯乐在线



前言：这篇打算从设计模式的角度去解析下委托的使用。我们知道使用委托可以实现对象行为（方法）的动态绑定，从而提高设计的灵活性。上次说过，方法可以理解为委托的实例，站在方法的层面，委托实例的一个非常有用的特性是它既不知道，也不关心其封装方法所属类的详细信息，对它来说最重要的是这些方法与该委托的参数和返回值的兼容性。即只要方法的返回类型和参数表是相同的，则方法与委托类型兼容，方法的名称及方法所属类等信息委托是不关心的。有一定编程经验的大侠们肯定都接触过设计模式，其实设计模式大多数都是面向对象多态特性的体现，通过重写子类方法去展现不同的设计需求，这样看，既然是方法重写，那么方法的参数类型和返回值类型肯定是一致的，这是不是和委托的实例十分相似，这样说来，我们通过多态去实现的设计模式是否可以用委托的形式去代替。博主觉得，为了更好的理解委托，可以从这方面着手试试。。。

此篇简单抽取了几个设计模式分别按照多态和委托的方式去实现，当然这里的重点并不是讲设计模式，而是为了使读者更好地理解委托。所以设计模式的很多细节，本篇可能会略过。

一、简单工厂模式：本篇就借助计算器的例子加以说明。

1、多态实现简单工厂模式。

C#

```
class Program2
{
    static void Main(string[] args)
    {
        //1. 使用多态实现简单工厂模式
        int x = 8, y = 2;
        var iRes1 = GetObject("+").Compute(x, y);
        var iRes2 = GetObject("-").Compute(x, y);
        var iRes3 = GetObject("*").Compute(x, y);
        var iRes4 = GetObject("/").Compute(x, y);
        Console.WriteLine(iRes1);
        Console.WriteLine(iRes2);
        Console.WriteLine(iRes3);
        Console.WriteLine(iRes4);
        Console.ReadKey();
    }

    static Calculator GetObject(string type)
    {
    }
```

```
Calculator oRes = null;
switch (type)
{
    case "+":
        oRes = new Add();
        break;
    case "-":
        oRes = new Subtract();
        break;
    case "*":
        oRes = new Multiply();
        break;
    case "/":
        oRes = new Divide();
        break;
}
return oRes;
}
}

public class Calculator
{
    public virtual int Compute(int x, int y)
    {
        return 0;
    }
}

public class Add : Calculator
{
    public override int Compute(int x, int y)
    {
        return x + y;
    }
}

public class Subtract : Calculator
{
    public override int Compute(int x, int y)
    {
        return x - y;
    }
}

public class Multiply : Calculator
{

```

```
        public override int Compute(int x, int y)
        {
            return x * y;
        }
    }

    public class Divide : Calculator
    {
        public override int Compute(int x, int y)
        {
            if (y == 0)
            {
                return 0;
            }
            return x / y;
        }
    }
```

代码应该很容易看懂，直接通过方法的重写去实现，在此就不过多讲解。

2、委托方式实现简单工厂模式。

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23

24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57

```
class Program2
{
    static void Main(string[] args)
    {
        #region 2. 委托实现简单工厂模式
        int x = 8, y = 2;
        var oCalculator = new Calculator();
        var iRes1 = oCalculator.Compute(x, y, oCalculator.Add); //将方法作为
```

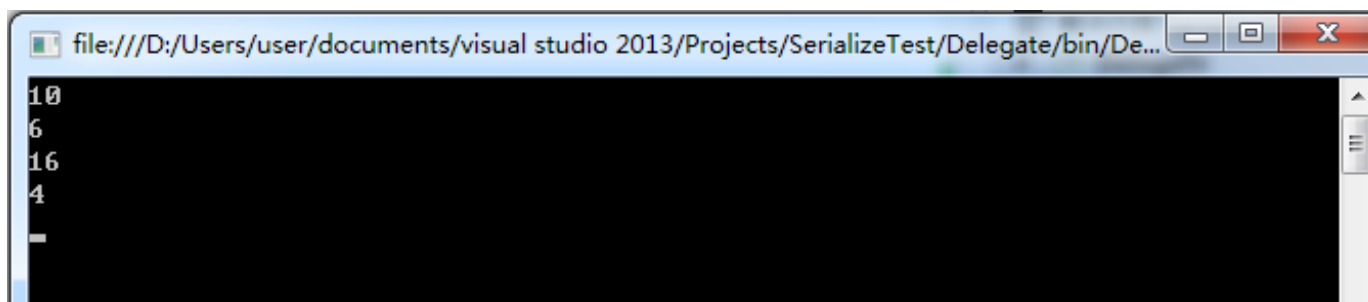
参数传下去

```
        var iRes2 = oCalculator.Compute(x, y, oCalculator.Subtract);
        var iRes3 = oCalculator.Compute(x, y, oCalculator.Multiply);
        var iRes4 = oCalculator.Compute(x, y, oCalculator.Divide);
        Console.WriteLine(iRes1);
        Console.WriteLine(iRes2);
        Console.WriteLine(iRes3);
        Console.WriteLine(iRes4);
        #endregion
        Console.ReadKey();
    }
}

public delegate int DelegateCalculator(int x, int y);
public class Calculator
{
    //将方法的实例传递进来，在Compute方法里面执行
    public int Compute(int x, int y, DelegateCalculator calculator)
    {
        return calculator(x, y);
    }
    public int Add(int x, int y)
    {
        return x + y;
    }
    public int Subtract(int x, int y)
    {
        return x - y;
    }
    public int Multiply(int x, int y)
    {
        return x * y;
    }
    public int Divide(int x, int y)
    {
        if (y == 0)
        {
            return 0;
        }
        return x / y;
    }
}
```

这里需要定义四个实现方法Add、Subtract、Multiply、Divide，而不用在意这四个方法在哪个类下面，

只要这四个方法的参数和返回值和委托的定义保持一致即可。这也验证了上面说的“站在方法的层面，委托实例的一个非常有用的特性是它既不知道，也不关心其封装方法所属类的详细信息，对它来说最重要的是这些方法与该委托的参数和返回值的兼容性”。两种方式得到的结果是相同的：



二、观察者模式：观察者模式最典型的场景就是订阅者和订阅号的场景

1、纯多态方式实现观察者模式：这种代码园子里面非常多。

C#

```
class Program3
{
    static void Main(string[] args)
    {
        // 具体主题角色通常用具体自来来实现
        ConcreteSubject subject = new ConcreteSubject();
        subject.Attach(new ConcreteObserver(subject, "Observer A"));
        subject.Attach(new ConcreteObserver(subject, "Observer B"));
        subject.Attach(new ConcreteObserver(subject, "Observer C"));
        subject.SubjectState = "Ready";
        subject.Notify();
        Console.Read();
    }
}

//抽象主题类
public abstract class Subject
{
    private IList<Observer> observers = new List<Observer>();

    /// <summary>
    /// 增加观察者
    /// </summary>
    /// <param name="observer"></param>
    public void Attach(Observer observer)
    {
        observers.Add(observer);
    }

    /// <summary>
    /// 移除观察者
```

```
    /// </summary>
    /// <param name="observer"></param>
    public void Detach(Observer observer)
    {
        observers.Remove(observer);
    }
    /// <summary>
    /// 向观察者（们）发出通知
    /// </summary>
    public void Notify()
    {
        foreach (Observer o in observers)
        {
            o.Update();
        }
    }
}
//具体主题类
public class ConcreteSubject : Subject
{
    private string subjectState;
    /// <summary>
    /// 具体观察者的状态
    /// </summary>
    public string SubjectState
    {
        get { return subjectState; }
        set { subjectState = value; }
    }
}
//抽象观察者类
public abstract class Observer
{
    public abstract void Update();
}
//具体观察者
public class ConcreteObserver : Observer
{
    private string observerState;
    private string name;
    private ConcreteSubject subject;
    /// <summary>
```

```

/// 具体观察者用一个具体主题来实现
/// </summary>
public ConcreteSubject Subject
{
    get { return subject; }
    set { subject = value; }
}

public ConcreteObserver(ConcreteSubject subject, string name)
{
    this.subject = subject;
    this.name = name;
}
/// <summary>
/// 实现抽象观察者中的更新操作
/// </summary>
public override void Update()
{
    observerState = subject.SubjectState;
    Console.WriteLine("The observer's state of {0} is {1}", name,
observerState);
}
}

```

可以看到虽然已经很好的实现了观察者Observer 和主题Subject之间的分离。但是Subject的内部还是有对观察者的调用：

```

C#
class Program3
{
    static void Main(string[] args)
    {
        // 具体主题角色通常用具体自来来实现
        ConcreteSubject subject = new ConcreteSubject();
        //传入的只是观察者的通过方法。
        subject.Attach(new ConcreteObserver(subject, "Observer A").Update);
        subject.Attach(new ConcreteObserver(subject, "Observer B").Update);
        subject.Attach(new ConcreteObserver(subject, "Observer C").Update);
        subject.SubjectState = "Ready";
        subject.Notify();
        Console.Read();
    }
}

public delegate void ObserverDelegate();

```


//抽象主题类

```
public abstract class Subject
{
    public ObserverDelegate observedelegate;
    /// <summary>
    /// 增加观察者
    /// </summary>
    /// <param name="observer"></param>
    public void Attach(ObserverDelegate observer)
    {
        observedelegate += observer;
    }
    /// <summary>
    /// 移除观察者
    /// </summary>
    /// <param name="observer"></param>
    public void Detach(ObserverDelegate observer)
    {
        observedelegate -= observer;
    }
    /// <summary>
    /// 向观察者（们）发出通知
    /// </summary>
    public void Notify()
    {
        if (observedelegate != null)
        {
            observedelegate();
        }
    }
}
```

//具体主题类

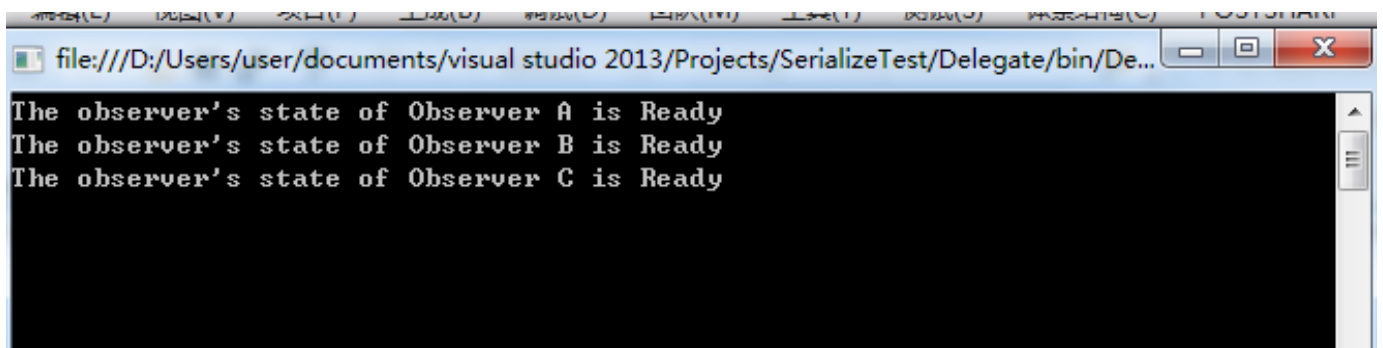
```
public class ConcreteSubject : Subject
{
    private string subjectState;
    /// <summary>
    /// 具体观察者的状态
    /// </summary>
    public string SubjectState
    {
        get { return subjectState; }
        set { subjectState = value; }
    }
}
```

```

    }
}
//具体观察者
public class ConcreteObserver
{
    private string observerState;
    private string name;
    private ConcreteSubject subject;
    /// <summary>
    /// 具体观察者用一个具体主题来实现
    /// </summary>
    public ConcreteSubject Subject
    {
        get { return subject; }
        set { subject = value; }
    }
    public ConcreteObserver(ConcreteSubject subject, string name)
    {
        this.subject = subject;
        this.name = name;
    }
    /// <summary>
    /// 实现抽象观察者中的更新操作
    /// </summary>
    public void Update()
    {
        observerState = subject.SubjectState;
        Console.WriteLine("The observer's state of {0} is {1}", name,
observerState);
    }
}

```

得到结果：



```

file:///D:/Users/user/documents/visual studio 2013/Projects/SerializeTest/Delegate/bin/De...
The observer's state of Observer A is Ready
The observer's state of Observer B is Ready
The observer's state of Observer C is Ready

```

这样设计的优势：

（1）将通知的方法Update通过委托的形式传入主题对象。这样主题对象Subject就完全和观察者隔离。

更好地实现了低耦合。

（2）减少了观察者抽象类的定义。使整个设计更加精简。

（3）如果将设计更进一步，观察者这边自定义delegate void ObserverDelegate() 这种类型的方法。比如需要执行Update() 方法之后还要记录一个日志的操作。如：

```
C#
class Program4
{
    static void Main(string[] args)
    {
        var oTem1 = new DeviceMML();
        oTem1.Spider();
        Console.WriteLine("");
        var oTem2 = new DeviceTL2();
        oTem2.Spider();
        Console.ReadKey();
    }
}

public abstract class TemplateDevice
{
    // 模板方法，不要把模版方法定义为Virtual或abstract方法，避免被子类重写，防止更改流程的执行顺序
    public void Spider()
    {
        Console.WriteLine("设备采集开始");
        this.Login();
        this.Validation();
        this.SpiderByType1();
        this.SpiderByType2();
        this.LoginOut();
        Console.WriteLine("设备采集结束");
    }
    // 登陆
    public void Login()
    {
        Console.WriteLine("登陆");
    }
    // 验证
    public void Validation()
    {
        Console.WriteLine("验证");
    }
}
```

```
}
// 采集
public abstract void SpiderByType1();
public abstract void SpiderByType2();
// 注销
public void LoginOut()
{
    Console.WriteLine("注销");
}
}
//MML类型的设备的采集
public class DeviceMML : TemplateDevice
{
    public override void SpiderByType1()
    {
        Console.WriteLine("MML类型设备开始采集1");
        //.....
    }
    public override void SpiderByType2()
    {
        Console.WriteLine("MML类型设备开始采集2");
    }
}
//TL2类型设备的采集
public class DeviceTL2 : TemplateDevice
{
    public override void SpiderByType1()
    {
        Console.WriteLine("TL2类型设备开始采集1");
        //.....
    }
    public override void SpiderByType2()
    {
        Console.WriteLine("TL2类型设备开始采集2");
    }
}
```

父类里面的非abstract方法都是模板方法，也就是子类公用并且不可以重写的方法。SpiderType1和SpiderType2是需要子类重写的方法。模板方法模式在抽象类中定义了算法的实现步骤，将这些步骤的实现延迟到具体子类中去实现，从而使所有子类复用了父类的代码，所以模板方法模式是基于继承的一种实现代码复用的技术。

2、使用委托改写后：

```
C#
class Program4
{
    static void Main(string[] args)
    {
        var oTem1 = new TemplateDevice(DeviceMML.SpiderByType1,
DeviceMML.SpiderByType2);
        oTem1.Spider();
        Console.WriteLine("");
        var oTem2 = new TemplateDevice(DeviceTL2.SpiderByType1,
DeviceTL2.SpiderByType2);
        oTem2.Spider();
        Console.ReadLine();
    }
}

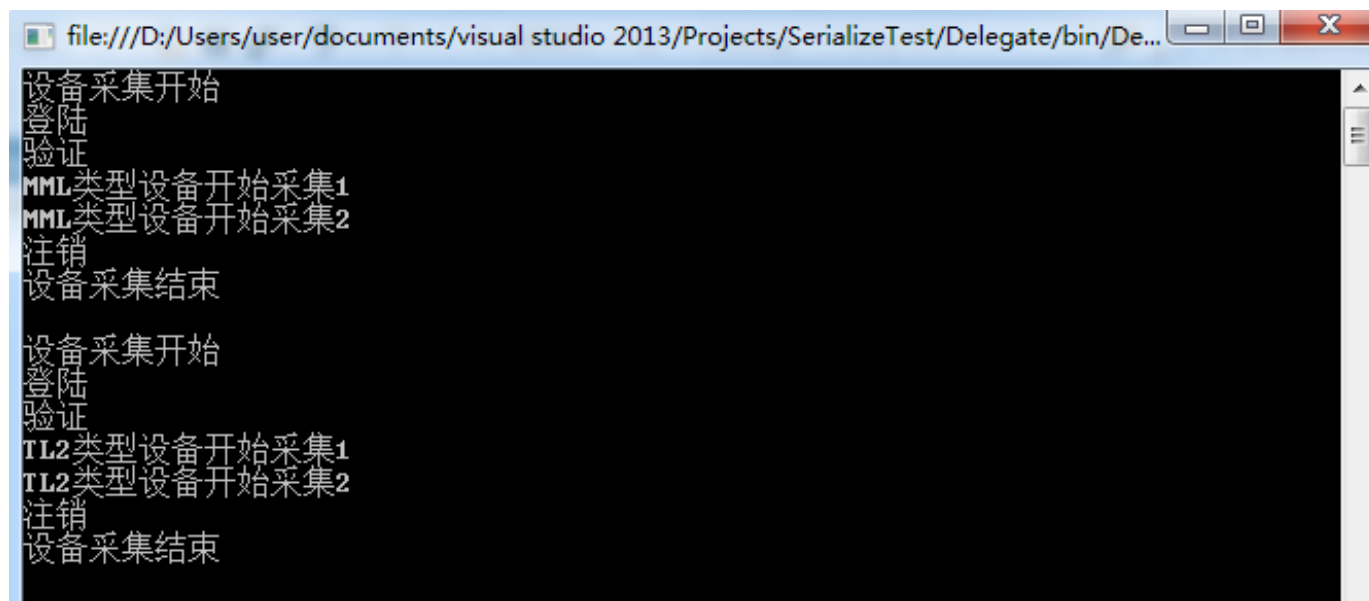
public delegate void DeviceDelegate();
public class TemplateDevice
{
    public DeviceDelegate oDelegate;
    public TemplateDevice(params DeviceDelegate[] lstFunc)
    {
        foreach (var oFunc in lstFunc)
        {
            oDelegate += oFunc;
        }
    }

    // 模板方法，不要把模版方法定义为Virtual或abstract方法，避免被子类重写，防止更改流程的执行顺序
    public void Spider()
    {
        Console.WriteLine("设备采集开始");
        this.Login();
        this.Validation();
        if (oDelegate != null)
        {
            oDelegate();
        }
        this.LoginOut();
        Console.WriteLine("设备采集结束");
    }

    // 登陆
    public void Login()
```

```
{
    Console.WriteLine("登陆");
}
// 验证
public void Validation()
{
    Console.WriteLine("验证");
}
// 注销
public void LoginOut()
{
    Console.WriteLine("注销");
}
}
//MML类型的设备的采集
public class DeviceMML
{
    public static void SpiderByType1()
    {
        Console.WriteLine("MML类型设备开始采集1");
        //.....
    }
    public static void SpiderByType2()
    {
        Console.WriteLine("MML类型设备开始采集2");
    }
}
//TL2类型设备的采集
public class DeviceTL2
{
    public static void SpiderByType1()
    {
        Console.WriteLine("TL2类型设备开始采集1");
        //.....
    }
    public static void SpiderByType2()
    {
        Console.WriteLine("TL2类型设备开始采集2");
    }
}
```

得到结果：



优化模板方法模式的意义：

（1）解除了子类和父类之间的继承关系，更好地实现了对象间的低耦合。

（2）采用委托可以动态实现方法的组合，这种方式更加灵活，子类可以更加灵活的设计不同部分的方法。然后方法的数量通过params来传递，方法的数量没有什么严格的限制。

当然其他设计模式也可以使用委托去优化设计，博主在这里就暂时只分享这三种模式的异同。总的来说，委托不可能代替多态去实现各种模式，但是它和多态联合起来使用可以实现更加灵活的设计。通过这两篇下来，不知道你是否对委托有点感觉了呢，委托这东西，重在实战，就像游泳一样，如果不用那么几次，你永远也不可能学会。以上只是博主个人的理解，可能很多方便没有考虑得那么全面，希望各位园友拍砖斧正~~

合作联系

Email: bd@jobbole.com

QQ: 2302462408 （加好友请注明来意）

更多频道

[小组](#) - 好的话题、有启发的回复、值得信赖的圈子

[头条](#) - 分享和发现有价值的内容与观点

[相亲](#) - 为IT单身男女服务的征婚传播平台

[资源](#) - 优秀的工具资源导航

[翻译](#) - 翻译传播优秀的外文文章

[文章](#) - 国内外的精选文章

[设计](#) - UI, 网页, 交互和用户体验

[iOS](#) - 专注iOS技术分享

[安卓](#) - 专注Android技术分享

[前端](#) - JavaScript, HTML5, CSS

[Java](#) - 专注Java技术分享

[Python](#) - 专注Python技术分享