

无废话 C#设计模式之一：开篇

什么是设计模式？

什么是少林拳呢？少林拳是少林僧人经过长期的总结，得出的一套武功套路。有一本叫做少林拳法的武功秘籍，上面记载这套拳法的适用人群，打法套路和学成后的效果。设计模式虽然记录在了设计模式一书上，但是要真正掌握设计模式光靠看每一个模式的结构并且进行模仿是不够的。试想一下，在真枪实战的情况下，谁会和你按照少林拳法，一二三四的套路打呢？打套路也只能用来看看，只有当你能根据不同的场景灵活出招的时候才能说是学会了这套拳法。相似的例子还有三十六计，这也是一种模式，每种计谋都是针对不同场景的，如果不管遇到什么时候都来个“走为上”，那这仗还怎么打呢？

总之，设计模式要用活才能发挥作用。

设计模式有什么用？

设计模式可以让你在遇到需求变化的时候不至于手忙脚乱。设计模式可以让你程序的可维护性、可扩展性更好。设计模式可以让程序的性能更高。当然，这些的前提是正确使用了设计模式，如果滥用的话那么设计模式可以让程序没人看得懂，让程序速度慢到死，让程序不能维护，添加新的功能等于重做。

设计模式的原则？

- 单一职责：你不希望因为电脑内存损坏而更换 CPU 吧，同样也不应该让一个类有多种修改的理由。
- 对扩展开放，对修改封闭：你一定不希望电脑只有一个内存槽，加内存就要换主板吧，程序也应该能在不修改原先程序的情况下就能扩展功能。
- 里氏替换：如果你买的 DX9显卡不支持 DX9特性，那么这个显卡一定没法用。如果父类的方法在子类中没有实现那就晕了。在程序的世界中千万别认为鸟都会飞，先考虑清楚将会有哪些鸟吧。
- 依赖倒置：针对接口编程，这样即使实现有变也不需要修改外部代码。其实，现在电脑的硬件、网络通讯等都是符合这个原则的，比如 USB 接口、PCI-E 接口、TCP/IP 协议。
- 接口隔离：花3000买一个带拍照、听 MP3功能的手机还是花1000买一个手机、1000买一个 MP3、1000买一个数码相机呢？买了前者的话手机动不动就要修，而且还不一定是因为不能打电话而修，买了后面三样的话即使修也不影响其它使用，你说买哪个？

记得看过一个例子很恰当，说是修电脑比修收音机简单多了。电脑坏了，更换一个零件即可，原因是电脑中的各部分都是基于相对稳定的接口，而且部件各司其职，不会相互影响，电脑本身就是一个非常符合设计原则的产品。收音机的修理没有这么简单了，没有什么部件是插件式的，会修收音机的人肯定明白其中每一个部件的原理。

小程序就好像收音机，确实可以这么做，一共才一个人做的，即使重新做也用不了多少时间。几

十个人的大项目如果要改一个需求需要牵涉所有人来修改,那么这个项目用不了多少时间就会因为维护成本太大,维护后 BUG 太多而报废。

怎样学习设计模式？

学习新概念英文要什么基础？首先,要知道26个字母吧。如果你对面向对象完全没有概念的话,建议先可以看一下面向对象的一些知识。毕竟,设计模式是面向对象编程模式的一种总结。学了26个字母你就可以学习新概念了,但是,为了能更好地学习最好是先学一下国际音标。对于设计模式的学习来说,你可以学习一下 UML 的一些知识。当然,完全不知道 UML 也可以学习设计模式,在学习的过程中慢慢也就会 UML 了。

设计模式不是什么很高深的东西,有了这些知识大胆地学习吧。很多人说,看了很多设计模式的文章,为什么就是看不懂呢？我觉得原因可能有两个,第一就是你没有花时间认真看,第二就是看的文章不适合作为切入点。不管学习什么,切入点非常重要,如果切入点不是那么平易近人的话很可能会把你拒之门外,对于初学者来说从实例切入最合适。最好是能碰到自己做过的项目的实例作为切入点,这样你一比较就知道为什么设计模式好了。

如果要把设计模式的学习境界分一下级的话,我这么分：

- 第一重：能看懂设计模式的文章
- 第二重：能自己写一个设计模式的骨架
- 第三重：能自己编一个新的运用设计模式的例子
- 第四重：能在写代码的时候想到似乎有设计模式适合,在翻阅资料后找到了这种设计模

式

- 第五重：在理解项目的需求后就能意识到哪里可以使用哪种设计模式进行优化
- 第六重：完全掌握了设计模式的精髓，灵活使用各种设计模式以及其变种

不管怎么样，多看多做多替换才是学习的办法，别人举例十个都不及自己做一个例子，被动十个原则都不及自己体会出一个原则。每一种设计模式虽然都有一个骨架，但是也不必过于强调这个形式，很多时候根据自己的需求简化一点，改变一点，或者混杂一些其它的设计模式，只要能实现目的了，也是一个不错的选择。

很多人会觉得这么多种设计模式没有几种能用得上。我觉得这不是什么问题，用不上那就用不上，这些设计模式是大师经历无数大型项目后的精华，如果能在自己做的一个小项目中用上两三个就很不错了，用上二三十个的项目绝对是怪胎。用不上千万别强求，否则既不利于项目的可维护性又增加了工作量。

还有很多人会觉得这些设计模式很多都是相似的。而且每个人的感觉还不一样，有人觉得 A 和 B 很相似，有人却觉得 A 和 B 很好区分，但是 B 和 C 却很相似啊。感觉很好区分，说明你看准设计模式的着重点的，感觉一样说明你看到的还是它的形。双胞胎虽然形一样，但是神肯定不一样的，只要认准设计模式解决的问题，就不会看错。

关于本系列文章

本来这些内容都是用来进行公司内部每周知识分享活动的，既然有一些内容了，想想不妨就整理一下贴出来吧。也正由于这个原因，文章中的一些例子都基于团队内部成员所能理解的一些项目，

可能这些项目对大家来说比较陌生，不过好处是例子相对比较贴近实际一点。本系列一共有20篇左右，除了介绍23种 GOF 设计模式中常用的一部分之外（一些设计模式的思想在 C#语言中有了更简单的实现，一些设计模式不是很常用）还可能会介绍一些其它有用的设计模式。在这些文章中，我不会过多去说一些理论上的东西，也不会有结构图（这些内容网上到处都是），所有的内容都是围绕相对实际例子展开。我想，只有这样才能更快的吸收设计模式的神而不是其形。在看文章的时候建议你结合《设计模式》一书以及博客园的其它设计模式相关文章一起看，这样才能对设计模式理解的全面和充分一点。

每一篇文章都会有以下部分：

- 意图：抄设计模式一书的，因为意图实在是太重要，所以不得不首先列出。
- 场景：以一个实际的场景来说明为什么要引入设计模式。
- 示例代码：对引入设计模式后场景的说明。
- 代码说明：说明设计模式中的几个角色以及代码中需要注意的地方。
- 何时采用：从代码和应用两个角度说明何时采用这个模式。
- 实现要点：实现这种模式必要的几个地方，或者说模式主要的特点在哪里。
- 注意事项：模式的优点缺点以及什么时候不应该使用设计模式。

无废话 C#设计模式之二：Singleton

意图

保证一个类只有一个实例，并提供访问它的全局访问点。

场景

我们现在要做一个网络游戏的服务端程序，需要考虑怎么样才能承载大量的用户。在做 WEB 程序的时候有各种负载均衡的方案，不管是通过硬件实现还是软件实现，基本的思想就是有一个统一的入口，然后由它来分配用户到各个服务器上去。

需要考虑的问题是，即使在多线程的并发状态下，用户只能通过一个唯一的入口来分配，由此引入了 Singleton 模式来实现这个唯一的入口。

示例代码

```
using System;

using System.Collections.Generic;

using System.Threading;

namespace SingletonExample
{

    class Program
```

```

{

    static void Main(string[] args)

    {

        ParameterizedThreadStart ts = new ParameterizedThreadStart(EnterPlayer);

        for (int i = 0; i < 20; i++)

        {

            Thread t = new Thread(ts);

            t.Start("player" + i);

        }

        LoadBalanceServer.GetLoadBalanceServer().ShowServerInfo();

    }

    static void EnterPlayer(object playerName)

    {

        LoadBalanceServer lbs = LoadBalanceServer.GetLoadBalanceServer();

        lbs.GetLobbyServer().EnterPlayer(playerName.ToString());
    }
}

```

```
}
```

```
}
```

```
class LoadBalanceServer
```

```
{
```

```
    private const int SERVER_COUNT = 3;
```

```
    private List<LobbyServer> serverList = new List<LobbyServer>();
```

```
    private static volatile LoadBalanceServer lbs;
```

```
    private static object syncLock = new object();
```

```
    private LoadBalanceServer()
```

```
    {
```

```
        for (int i = 0; i < SERVER_COUNT; i++)
```

```
        {
```

```
            serverList.Add(new LobbyServer("LobbyServer" + i));
```

```
        }
```

```
    }
```



```
public static LoadBalanceServer Get LoadBalanceServer()
```

```
{
```

```
    if (l bs == null)
```

```
    {
```

```
        lock(syncLock)
```

```
        {
```

```
            if (l bs == null)
```

```
            {
```

```
                Thread.Sleep(100);
```

```
                l bs = new LoadBalanceServer();
```

```
            }
```

```
        }
```

```
    }
```

```
    return l bs;
```

```
}
```

```
public LobbyServer Get LobbyServer()
```

```

{

    LobbyServer ls = serverList[0];

    for(int i = 1; i < SERVER_COUNT; i++)

    {

        if(serverList[i].PlayerList.Count < ls.PlayerList.Count)

            ls = serverList[i];

    }

    return ls;

}

public void ShowServerInfo()

{

    foreach(LobbyServer ls in serverList)

    {

        Console.WriteLine("===== " + ls.ServerName
+ "=====");

        foreach(string player in ls.PlayerList)

        {

```

```

        Console.WriteLine(player);

    }

}

}

}

class LobbyServer
{

    private List<string> playerList = new List<string>();

    public List<string> PlayerList
    {

        get { return playerList; }

    }

    private string serverName;

    public string ServerName

```

```
{

    get{return serverName; }

}

public LobbyServer(string serverName)

{

    this.serverName = serverName;

}

public void EnterPlayer(string playerName)

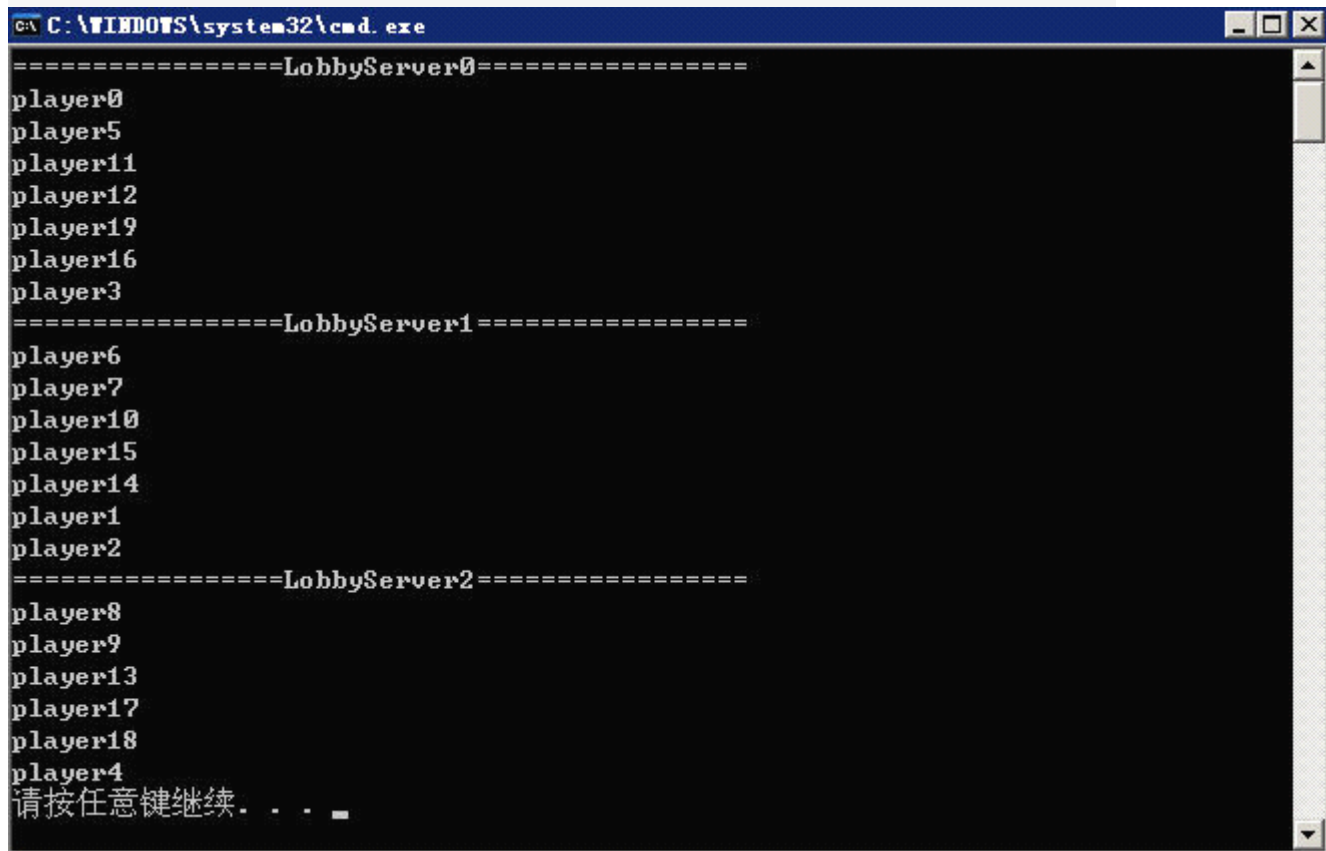
{

    playerLi st .Add(playerName);

}

}
```

代码执行结果如下图：



```
C:\WINDOWS\system32\cmd.exe

=====-LobbyServer0=====
player0
player5
player11
player12
player19
player16
player3
=====-LobbyServer1=====
player6
player7
player10
player15
player14
player1
player2
=====-LobbyServer2=====
player8
player9
player13
player17
player18
player4
请按任意键继续. . .
```

代码说明

- LoadBalanceServer 类实现了 Singleton 模式，也就是说无论在什么情况下，只会有一个 LoadBalanceServer 类的实例出现。
- LobbyServer 类表示大厅服务，用户进入大厅后和大厅服务进行服务，在这里我们仅仅在大厅服务里面保存了用户列表。
- Singleton 模式有很多实现方式，在这里使用的是双重锁定方式。对于 C#来说，可能使用静态初始化方式是最简洁的，这里就不演示了。

- LoadBalancerServer 类的 GetLobbyServer() 方法负责返回一个压力最小的 LobbyServer 对象。
- 实例化 LoadBalancerServer 的时候 Sleep 了线程，目的是模拟高并发的情况，在正式代码中没有必要这样做。

何时采用

- 从代码角度来说，当你希望类只有一个实例的时候。
- 从应用角度来说，你希望有一个总管来负责某一件事情。并且这件事情的分配只能有一个人进行，如果有多个进行肯定会弄乱。比如创建处理流水号如果有两个地方在创建的话是不是就会重复了呢？

实现要点

- 一个 Singleton 类，它能确保自身的实例是唯一的。

注意事项

- 不要滥用 Singleton 模式，只有非一个实例不可的情况下才考虑引入 Singleton。否则，程序的可扩展性可能会受到限制。

无废话 C#设计模式之三：Abstract Factory

意图

提供一个创建一系列相关或相互依赖对象的接口，而无需指定它们具体的类。

场景

还是上次说的那个网络游戏，定下来是一个休闲的 FPS 游戏。和 CS 差不多，8到16个玩家在游戏里面分成2组对战射击。现在要实现初始化场景的工作。要呈现一个三维物体一般两个元素是少不了的，一是这个物体的骨架，也就是模型，二就是这个骨架上填充的纹理。

我们知道，这样的游戏不可能只有一张地图，而且地图的数量肯定会一直增加的。如果游戏在初始化场景的时候需要根据不同的地图分别加载模型和纹理对象，那么势必就会使得场景的扩充变得很不方便。由此，我们引入 Abstract Factory，抽象工厂生产的都是实际类型的接口（或者抽象类型），如果加了新的场景可以确保不需要修改加载场景的那部分代码。

示例代码

```
using System;
```

```
using System.Reflection;
```

```
namespace AbstractFactoryExample
```

```
{
```

```
    class Program
```

```
    {
```

```
        static void Main(string[] args)
```

```
        {
```

```
            Patrix patri x = new Patrix();
```

```
            patri x.LoadScene("HalfPaper");
```

```
            patri x.LoadScene("Matrix");
```

```
        }
```

```
    }
```

```
    class Patrix
```

```
    {
```

```
        private PatrixSceneFactory GetGameScene(string gameSceneName)
```

```
        {
```



```
return(PatrxSceneFactory)Assembly.Load("AbstractFactoryExample").CreateInstance("AbstractF  
actoryExample." + gameSceneName);
```

```
}
```

```
publicvoidLoadScene(stringgameSceneName)
```

```
{
```

```
    PatrxSceneFactorypsf = GetGameScene(gameSceneName);
```

```
    Texturetexture = psf.CreateTexture();
```

```
    Modelmodel = psf.CreateModel();
```

```
    model.FillTexture(texture);
```

```
}
```

```
}
```

```
abstractclassPatrxSceneFactory
```

```
{
```

```
    publicabstractModelCreateModel();
```

```
    publicabstractTextureCreateTexture();
```

```
}
```

```
abstract class Model
```

```
{
```

```
    public abstract void FillTexture(Texture texture);
```

```
}
```

```
abstract class Texture
```

```
{
```

```
}
```

```
class HalfPaper : PatrixSceneFactory
```

```
{
```

```
    public override Model CreateModel()
```

```
    {
```

```
        return new HalfPaperModel();
```

```
    }
```

```
public override Texture CreateTexture()

{

    return new HalfPaperTexture();

}

}

class HalfPaperModel : Model

{

    public HalfPaperModel ()

    {

        Console.WriteLine("HalfPaper Model Created");

    }

    public override void FillTexture(Texture texture)

    {

        Console.WriteLine("HalfPaper Model is filled Texture");

    }

}

}
```

```
classHalfPaperTexture:Texture
```

```
{
```

```
    publicHalfPaperTexture()
```

```
    {
```

```
        Console.WriteLine("HalfPaper Texture Created");
```

```
    }
```

```
}
```

```
classMatrix:PatrxSceneFactory
```

```
{
```

```
    publicoverrideModelCreateModel()
```

```
    {
```

```
        returnnewMatrixModel();
```

```
    }
```

```
    publicoverrideTextureCreateTexture()
```

```
    {
```

```
return new MatrixTexture();
```

```
}
```

```
}
```

```
class MatrixModel:Model
```

```
{
```

```
public MatrixModel()
```

```
{
```

```
    Console.WriteLine("Matrix Model Created");
```

```
}
```

```
public override void FillTexture(Texture texture)
```

```
{
```

```
    Console.WriteLine("Matrix Model is filled Texture");
```

```
}
```

```
}
```

```
class MatrixTexture:Texture
```

```
{

    publicMatrixTexture()

    {

        Console.WriteLine("Matrix Texture Created");

    }

}

}
```

代码执行结果如下图：

代码说明

- PatrixSceneFactory 就是一个抽象工厂 ,它声明了创建抽象的场景以及抽象的纹理的接口。(广告时间 :Patrix 是我公司的一款休闲 FPS 游戏 ,详细请见 <http://www.qwd1.com>)
- Model 和 Texture 是抽象产品。在 Model 类中有一个抽象方法 ,用于为模型填充纹理。
- HalfPaper 和 Matrix 是具体工厂 ,它用于创建某个场景的模型和纹理。(你可能对两个类的名字不太理解 , 其实 HalfPaper 和 Matrix 是两个地图的名字)

- xxxModel 和 xxxTexture 就是具体的产品了。它们就是针对某个场景的模型和纹理，具体工厂负责创建它们。
- Patrix 这个类负责加载场景，为了避免加载不同场景使用 case 语句，在这里我们使用反射来加载具体工厂类。
- 可以看到，一旦有了新的场景（或者说地图），我们只需要设计新的 xxxModel 和 xxxTexture 以及具体工厂类就可以了，加载场景的那部分代码（也就是 Patrix 类）不需要做改动。
- 我们现在这个游戏可是不支持和电脑对战的，万一以后需要支持电脑了，那么场景中的元素除了纹理和模型之外就还需要加电脑了。也就是说抽象工厂还需要多生产一种类型的产品，这个时候抽象工厂就无能为力了。抽象工厂只能解决系列产品扩张的变化点（在我们的例子中就是地图的新增），因此千万把抽象工厂所能生产的产品考虑周全了。

何时采用

- 从代码角度来说，你希望在统一的地方创建一系列相互关联的对象，并且基于抽象对象的时候。
- 从应用角度来说，如果你的产品是成组成套的，并且肯定会不断扩展新系列的，那么就适用抽象工厂。比如说，外面买的塑料模型，里面总是有图纸、模型元件板和外壳包装三部分。那么生产模型的厂就是抽象工厂，打印图纸、打印包装盒以及生产元件板的三个流水线就是具体工厂了。需要生产新的模型，只需要制作新的图纸输入到三个流水线的电脑中就可以了。

实现要点

- 抽象工厂本身不负责创建产品 ,产品最终还是由具体工厂来创建的。比如 ,MatrixModel 是 Matrix 创建的 ,而不是 PatrixSceneFactory 创建的。在.NET 中可以使用反射来创建具体工厂 ,从而使得代码变动降到最低。
- 在抽象工厂中需要体现出生产一系列产品。这一系列产品是相互关联 ,相互依赖一起使用的。
- 抽象工厂对应抽象产品 ,具体工厂对应具体产品 ,外部依赖抽象类型 ,这样对于新系列产品的创建 ,外部唯一依赖的就是具体工厂的创建过程 (可以通过反射解决)。

注意事项

- 一般来说需要创建一系列对象的时候才考虑抽象工厂。比如 ,创建一个场景 ,需要创建模型和纹理 ,并且模型和纹理之间是有一定联系的 ,不太可能把 PatrixTexture 套用在 MatrixModel 上。
- 如果系统的变化点不在新系列的扩充上 ,那么就没有必要使用抽象工厂。比如 ,如果我们不会增加新地图的话 ,那么也就没有必要引入抽象工厂。

.NET 中的抽象工厂

- 我们说过，抽象工厂针对系列产品的应变。在使用 ADO.NET 进行数据访问的时候，如果目标数据库是 Access，我们会使用 OleDbConnection、OleDbCommand 以及 OleDbDataAdapter 等一系列 ADO.NET 对象。那么如果数据库是 SQL Server，我们又会改用 SqlConnection、SqlCommand 以及 SqlDataAdapter 等一系列 ADO.NET 对象。如果只使用一套对象，没有什么大问题，如果我们的数据访问有系列变化的需求，比如可以针对 Access 和 SQL Server，而且希望改换数据库尽量对客户端代码透明，那么就需要引入抽象工厂模式。
- 好在，ADO.NET 2.0中已经有了整套抽象工厂的类型。看下面的代码，你应该能辨别这些类型在抽象工厂中的角色：

```
publicabstractclassDbProviderFactory

{

    // Methods

    protectedDbProviderFactory()

    {

    }

    publicvirtualDbCommand CreateCommand()

    {
```

```
        returnnull;

    }

    publicvirtualDbCommandBuil der CreateCommandBuil der()

    {

        returnnull;

    }

    publicvirtualDbConnecti on CreateConnecti on()

    {

        returnnull;

    }

    publicvirtualDbConnecti onStri ngBuil der CreateConnecti onStri ngBuil der()

    {

        returnnull;

    }
```

```
publicvirtualDbDataAdapter CreateDataAdapter()
```

```
{
```

```
    returnnull;
```

```
}
```

```
publicvirtualDbDataSourceEnumerator CreateDataSourceEnumerator()
```

```
{
```

```
    returnnull;
```

```
}
```

```
publicvirtualDbParameter CreateParameter()
```

```
{
```

```
    returnnull;
```

```
}
```

```
publicvirtualCodeAccessPermission CreatePermission(PermissionState state)
```

```
{
```

```
    returnnull;
```

```
}

// Properties

publicvirtualboolCanCreateDataSourceEnumerator

{

    get

    {

        returnfalse;

    }

}

}
```

```
publicsealedclassOleDbFactory:DbProviderFactory
```

```
{

    // Fields

    publicstaticreadonlyOleDbFactoryInstance =newOleDbFactory();

    // Methods
```

```
private OleDbFactory()
```

```
{
```

```
}
```

```
public override DbCommand CreateCommand()
```

```
{
```

```
    return new OleDbCommand();
```

```
}
```

```
public override DbCommandBuilder CreateCommandBuilder()
```

```
{
```

```
    return new OleDbCommandBuilder();
```

```
}
```

```
public override DbConnection CreateConnection()
```

```
{
```

```
    return new OleDbConnection();
```

```
}
```

```
public override DbConnectionStringBuilder CreateConnectionStringBuilder()
```

```
{
```

```
    return new OleDbConnectionStringBuilder();
```

```
}
```

```
public override DbDataAdapter CreateDataAdapter()
```

```
{
```

```
    return new OleDbDataAdapter();
```

```
}
```

```
public override DbParameter CreateParameter()
```

```
{
```

```
    return new OleDbParameter();
```

```
}
```

```
public override CodeAccessPermission CreatePermission(PermissionState state)
```

```
{
```

```
        return new OleDbPermission(state);

    }

}

public sealed class SqlClientFactory : DbProviderFactory, IServiceProvider

{

    // Fields

    public static readonly SqlClientFactory Instance = new SqlClientFactory();

    // Methods

    private SqlClientFactory()

    {

    }

    public override DbCommand CreateCommand()

    {

        return new SqlCommand();

    }

}
```

```
public override DbCommandBuilder CreateCommandBuilder()
```

```
{
```

```
    return new SqlCommandBuilder();
```

```
}
```

```
public override DbConnection CreateConnection()
```

```
{
```

```
    return new SqlConnection();
```

```
}
```

```
public override DbConnectionStringBuilder CreateConnectionStringBuilder()
```

```
{
```

```
    return new SqlConnectionStringBuilder();
```

```
}
```

```
public override DbDataAdapter CreateDataAdapter()
```

```
{
```



```
        return new SqlDataAdapter();

    }

    public override DbDataSourceEnumerator CreateDataSourceEnumerator()

    {

        return SqlDataSourceEnumerator.Instance;

    }

    public override DbParameter CreateParameter()

    {

        return new SqlParameter();

    }

    public override CodeAccessPermission CreatePermission(PermissionState state)

    {

        return new SqlClientPermission(state);

    }

}
```

```
objectIServiceProvider.GetService(Type serviceType)

{

    objectobj2 =null;

    if(serviceType == GreenMthods.SystemDataCommonDbProviderServices_Type)

    {

        obj2 =

GreenMthods.SystemDataSqlClientSqlProviderServices_Instance();

    }

    returnobj2;

}

// Properties

publicoverrideboolCanCreateDataSourceEnumerator

{

    get

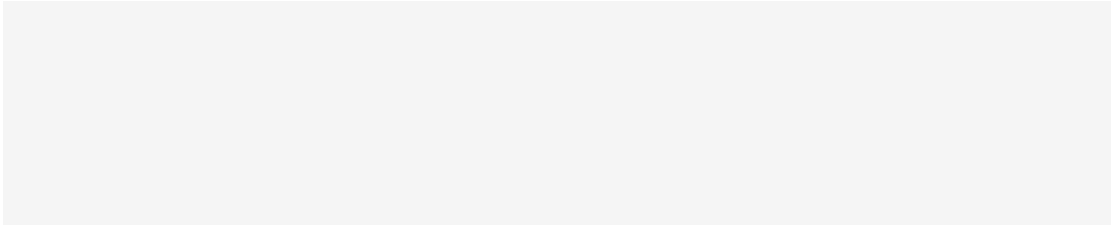
    {

        returntrue;

    }

}
```

```
}  
  
}
```



无废话 C#设计模式之四：Factory Method

意图

定义一个创建产品对象的工厂接口，将实际创建工作推迟到子类中。

场景

上次，我们使用抽象工厂解决了生产一组产品的问题，但是我们把各个场景作为了具体工厂来生产场景模式和场景纹理两个产品。在调用代码中也并没有出现具体工厂的影子。其实，场景类要做的不仅仅是创建具体的产品系列，可能它还需要做一个初始化工作。那么，我们就需要在调用代码中能得到这个场景类。

在前一节中，由于场景类（比如 HalfPaper）本身是具体级别的（具体工厂）。那么，我们也不应该在调用代码中直接依赖场景类。因此，我们可以使用工厂方法来生产这个具体产品。

示例代码

```
using System;

using System.Reflection;

namespace FactoryMethodExample
{

    class Program

    {

        static void Main(string[] args)

        {

            Patrix patri x = new Patrix();

            patri x.LoadScene("HalfPaper");

            patri x.LoadScene("Matrix");

        }

    }

}
```

```

class Patrix

{

    private PatrixSceneFactory GetGameScene(string gameSceneName)

    {

        return (PatrixSceneFactory) Assembly.Load("FactoryMethodExample").CreateInstance("FactoryMethodExample." + gameSceneName + "Factory");

    }

    public void LoadScene(string gameSceneName)

    {

        PatrixSceneFactory psf = GetGameScene(gameSceneName);

        PatrixScene ps = psf.CreateScene();

        ps.InitScene();

    }

}

abstract class PatrixSceneFactory

```

```
{  
  
    public abstract PatrixScene CreateScene();  
  
}
```

```
abstract class PatrixScene
```

```
{  
  
    public void InitScene()  
  
    {  
  
        Texture texture = CreateTexture();  
  
        Model model = CreateModel();  
  
        model.FillTexture(texture);  
  
    }  
  
}
```

```
    public abstract Model CreateModel();
```

```
    public abstract Texture CreateTexture();
```

```
}
```

```
abstract class Model
```

```
{
```

```
    public abstract void FillTexture(Texture texture);
```

```
}
```

```
abstract class Texture
```

```
{
```

```
}
```

```
class HalfPaperFactory : PatrixSceneFactory
```

```
{
```

```
    public override PatrixScene CreateScene()
```

```
    {
```

```
        return new HalfPaper();
```

```
    }
```

```
}
```

```
classHalfPaper:PatrxScene

{

    publicHalfPaper()

    {

        Console.WriteLine("HalfPaper Creating");

    }

    publicoverrideModelCreateModel ()

    {

        returnnewHalfPaperModel();

    }

    publicoverrideTextureCreateTexture()

    {

        returnnewHalfPaperTexture();

    }

}
```



```
classHalfPaperModel:Model

{

    publicHalfPaperModel ()

    {

        Console.WriteLine("HalfPaper Model Creating");

    }

    publicoverridevoidFillTexture(Texturetexture)

    {

        Console.WriteLine("HalfPaper Model is filled Texture");

    }

}

classHalfPaperTexture:Texture

{

    publicHalfPaperTexture()

    {

        Console.WriteLine("HalfPaper Texture Created");

    }

}
```

```
}
```

```
}
```

```
class MatrixFactory:PatrxSceneFactory
```

```
{
```

```
    public override PatrxScene CreateScene()
```

```
    {
```

```
        return new Matrix();
```

```
    }
```

```
}
```

```
class Matrix:PatrxScene
```

```
{
```

```
    public Matrix()
```

```
    {
```

```
        Console.WriteLine("Matrix Created");
```

```
    }
```

```
    public override Model CreateModel()
```

```
{  
  
    return new MatrixModel();  
  
}
```

```
public override Texture CreateTexture()  
  
{  
  
    return new MatrixTexture();  
  
}
```

```
}
```

```
class MatrixModel : Model
```

```
{  
  
    public MatrixModel()  
  
    {  
  
        Console.WriteLine("Matrix Model Created");  
  
    }
```

```

public override void FillTexture(Texture texture)
{
    Console.WriteLine("Matrix Model is filled Texture");
}

}

class MatrixTexture:Texture
{
    public MatrixTexture()
    {
        Console.WriteLine("Matrix Texture Created");
    }
}
}

```

代码执行结果如下图：

```
C:\WINDOWS\system32\cmd.exe
HalfPaper Creating
HalfPaper Texture Created
HalfPaper Model Creating
HalfPaper Model is filled Texture
Matrix Created
Matrix Texture Created
Matrix Model Created
Matrix Model is filled Texture
请按任意键继续. . .
```

代码说明

- 这个代码基于前一节抽象工厂的代码修改而来，因此代码比较厂。其中能体现的设计模式有抽象工厂、工厂方法以及模版方法。
- 这次的 PatrixSceneFactory 和前一节的不同，它是一个产品的抽象工厂，也就是工厂方法模式中的抽象工厂角色，它是具体产品工厂的抽象形式。
- HalfPaperFactory 和 MatrixFactory 是工厂方法模式中的具体工厂角色，它负责创建具体的产品。
- PatrixScene 是工厂方法模式中的抽象产品角色，同时也是抽象工厂模式中的抽象工厂角色。它既是场景的抽象形式，又负责创建每个场景中的产品系列，也就是模型和纹理。还有，它的 InistScene()方法还体现了模版方法的思想，封装了产品初始化过程中的共同步骤。
- HalfPaper 和 Matrix 当然就是工厂方法模式中的具体产品角色了，同时，它们也是抽象工厂模式中的具体工厂角色。
- 从这个例子可以看出，抽象工厂针对一组产品的创建进行抽象，抽象程度比较高。抽象

工厂生产重点在于规范一组产品的创建，能让产品线保持产品的一致。比如，N 卡不管是7系列还是8系列，总会分低端的7300，8300和中端的7600，8600以及高端的7900，8900。

- 而工厂方法针对某种产品的创建，每种产品在创建的过程中可能会有一些相似的步骤，那么就可以在抽象产品中进行一些提取，自然而然运用到了模版方法。工厂方法还能针对具体产品创建时的易变性，在这里我们可能很清楚 `HalfPaperFactory` 一定会创建 `HalfPaper` 这个产品，但是万一以后改为创建 `HalfPaperSpecial` 了呢？有了工厂方法，我们可以只需修改 `HalfPaperFactory` 就可以了。
- 不管怎么样，目的还是让调用方尽量和接口依赖（或者说和稳定的东西去依赖，让变化在接口下面变），既是要以来具体类型，也希望能只依赖一个。试想一下，如果没有抽象工厂和工厂方法，也少了这些抽象类型，那么调用方可能就要依赖具体场景类型和具体的纹理以及模型类型。并且在调用的时候，通过条件来判断并且创建各种具体类型，一旦有新的场景需要实现，调用方代码可能就需要做很大的调整。暂且不说调整的工作量有多少，调整所带来的风险谁能承担呢？

何时采用

- 从代码角度来说， 如果我们需要创建一个易变的对象，或是希望对象由子类决定创建哪个对象的时候可以考虑工厂方法。
- 从应用角度来说， 如果我们觉得具体产品的创建不稳定，或者客户端根本无需知道创建哪个具体产品的时候可以使用工厂方法。后者对于框架和工具包软件来说更常见，比如有一个打印类负责打印图纸，我们需要得到一个打印对象，对于调用方来说并不知道要使

用超宽打印对象还是普通打印对象，我们可以通过工厂方法使客户端和抽象打印工厂直接沟通，由它来决定具体创建哪个打印对象。

实现要点

- 通过继承创建具体产品。很多时候，每一种具体产品对应一个具体的工厂来创建。
- 使用具体工厂类来决定怎么样创建具体产品。调用方并不关心工厂创建的是哪个游戏场景，它只用知道工厂给我的是一个游戏场景即可。

注意事项

- 工厂方法通常需要为每个具体产品对应一个具体工厂，如果滥用的话会使得类的数目急剧增多。

无废话 C#设计模式之五：Prototype

意图

用原型实例指定创建对象的种类，并且通过拷贝这些原型创建新的对象。

场景

游戏场景中的有很多相似的敌人，它们的技能都一样，但是随着敌人出现的位置不同，这些人的能力不太一样。假设，我们现在需要把三个步兵组成一队，其中还有一个精英步兵，能力特别高。那么，你或许可以创建一个敌人抽象类，然后对于不同能力的步兵创建不同的子类。然后，使用工厂方法等设计模式让调用方依赖敌人抽象类。

问题来了，如果有无数种能力不同步兵，难道需要创建无数子类吗？还有，步兵模型的初始化工作是非常耗时的，创建这么多步兵对象可能还会浪费很多时间。我们是不是可以通过只创建一个步兵原型，然后复制出多个一模一样的步兵呢？复制后，只需要调整一下这些对象在地图上出现的位置，或者调整一下它们的能力即可。原型模式就是用来解决这个问题的。

示例代码

```
using System;  
  
using System.Threading;  
  
using System.Collections.Generic;  
  
using System.IO;  
  
using System.Runtime.Serialization.Formatters.Binary;  
  
using System.Diagnostics;
```



```
namespace PrototypeExample
{
    class Program
    {
        static void Main(string[] args)
        {
            Stopwatch sw = new Stopwatch();

            sw.Start();

            Enemy enemyPrototype = new FootMan(5, 4, new Location(100, 200));

            GameScene gs = new GameScene();

            List<Enemy> enemyGroup = gs.CreateEnemyGroup(enemyPrototype);

            foreach(FootMan ft in enemyGroup)
            {
                ft.ShowInfo();

                ft.FootmanAttack();
            }
        }
    }
}
```

```

        Console.WriteLine(sw.ElapsedMilliseconds);

    }

}

class GameScene

{

    public List<Enemy> CreateEnemyGroup(Enemy enemyPrototype)

    {

        List<Enemy> enemyGroup = new List<Enemy>();

        Enemy e1 = enemyPrototype.Clone(true);

        e1.Location.x = enemyPrototype.Location.x - 10;

        Enemy e2 = enemyPrototype.Clone(true);

        e2.Location.x = enemyPrototype.Location.x + 10;

        Enemy elite = enemyPrototype.Clone(true);

        elite.Power = enemyPrototype.Power * 2;

        elite.Speed = enemyPrototype.Speed * 2;

        elite.Location.x = enemyPrototype.Location.x;

        elite.Location.y = enemyPrototype.Location.y + 10;
    }
}

```

```
        enemyGroup.Add(e1);

        enemyGroup.Add(e2);

        enemyGroup.Add(elite);

        return enemyGroup;
```

```
    }
```

```
}
```

```
[Serializable]
```

```
class Location
```

```
{
```

```
    public int x;
```

```
    public int y;
```

```
    public Location(int x, int y)
```

```
    {
```

```
        this.x = x;
```

```
        this.y = y;
```

```
    }
```

```
}
```

```
[Serializable]
```

```
abstract class Enemy
```

```
{
```

```
    protected Location location;
```

```
    public Location getLocation()
```

```
{
```

```
        return location; }
```

```
    setLocation(Location value) {
```

```
    }
```

```
    protected int power;
```

```
    public int getPower()
```

```
{
```

```
        return power; }
```

```
        set{ power =value; }  
  
    }
```

```
protected int speed;
```

```
public int Speed
```

```
{  
  
    get{ return speed; }  
  
    set{ speed =value; }  
  
}
```

```
public abstract Enemy Clone(boolean DeepCopy);
```

```
public abstract void ShowInfo();
```

```
public Enemy(int power, int speed, Location location)
```

```
{  
  
    Thread.Sleep(1000); // Construct method is assumed to be a high calc work.  
  
    this.power = power;
```

```

        this.speed = speed;

        this.location = location;

    }

}

[Serializable]

class FootMan: Enemy

{

    private string model;

    public FootMan(int power, int speed, Location location)

        : base(power, speed, location)

    {

        model = "footman";

    }

    public override void ShowInfo()

    {

```

```

        Console.WriteLine("model:{0} power:{1} speed:{2} location:({3},{4})", model,
power, speed, location.x, location.y);

    }

    public override Enemy Clone(bool isDeepCopy)

    {

        FootMan footman;

        if (isDeepCopy)

        {

            MemoryStream memoryStream = new MemoryStream();

            BinaryFormatter formatter = new BinaryFormatter();

            formatter.Serialize(memoryStream, this);

            memoryStream.Position = 0;

            footman = (FootMan)formatter.Deserialize(memoryStream);

        }

        else

        {

            footman = (FootMan)this.MemberwiseClone();

        }

        return footman;
    }

```

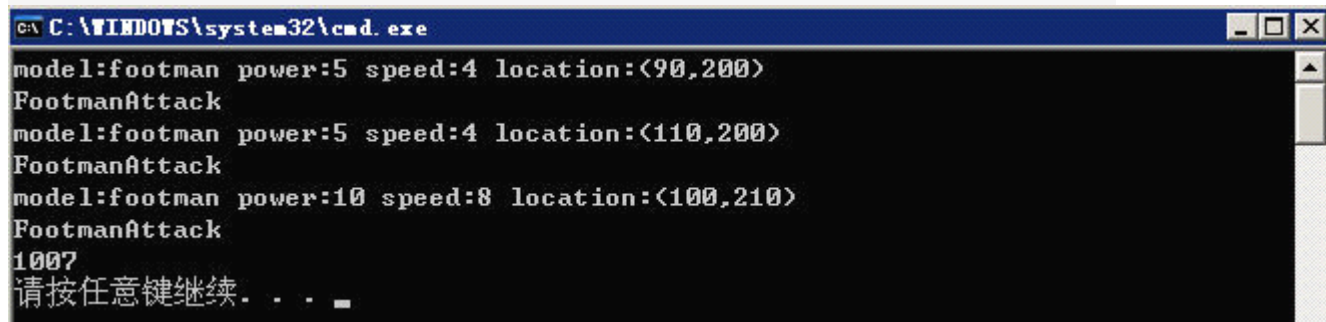
```

    }

    public void FootmanAttack()
    {
        Console.WriteLine("FootmanAttack");
    }
}
}
}

```

代码执行结果如下图：



```

C:\WINDOWS\system32\cmd.exe
model:footman power:5 speed:4 location:<90,200>
FootmanAttack
model:footman power:5 speed:4 location:<110,200>
FootmanAttack
model:footman power:10 speed:8 location:<100,210>
FootmanAttack
1007
请按任意键继续. . .

```

代码说明

- Enemy 类是抽象原型，它有两个用途，一是定义了原型的一些抽象内容，二是定义了原型模式必须的拷贝方法。在这里，我们看到，每个敌人的属性有位置、攻击力、速度等，

并且能通过 ShowInfo()方法来获取这个人的信息。

- FootMan 类就是具体原型了，它显示了敌人的具体参数以及实现了克隆自身。
- GameScene 类就是调用方，在这里我们并没有看到有和具体原因进行依赖，通过复制传入的克隆原型，得到一些新的敌人，在原型的基础上稍微调整一下就变成了一支敌人部队。
- 原型模式通过对原型进行克隆来替代无数子类，因此也就减少了调用方和具体类型产生依赖的程序。
- Clone()方法接受一个参数，表示是否是深拷贝。在这里，我们通过序列化反序列化实现深拷贝，深拷贝实现对象的完整复制，包括对象内部的引用类型都会复制一份全新的。在这里，如果3个敌人对象的 Location 都指向内存同一个地址的话，那么它们就分不开了，因此，在复制的时候需要进行深拷贝，使得它们的 Location 是独立的。
- 在初始化 Enemy 的时候，我们 Sleep()了一下，目的是模拟对象的创建是一个非常耗时的工作，这也体现了原型模式的另一个优势，在生成敌人的时候，我们其实无需再做这些工作了，我们只需要得到它的完整数据，并且进行一些修改就是一个新的敌人。
- 运行程序后可以看到，虽然创建了三个敌人，但是只耗费了一个敌人的创建时间，三个敌人都是从原型克隆出来的。由于进行了深拷贝，修改了一个敌人的位置并不会影响其它敌人。

何时采用

- 从代码角度来说，如果你希望运行时指定具体类(比如是使用 Footman 作为敌人还是使用其它)，或者你希望避免创建对象时的初始化过程(如果这个过程占用的时间和资源都非常多)，或者是希望避免使用工厂方法来实现多态的时候，可以考虑原型模式。
- 从应用角度来说，如果你创建的对象是多变化、多等级的产品，或者产品的创建过程非常耗时的时候(比如，有一定的计算量，或者对象创建时需要从网络或数据库中获取一定的数据)，或者想把产品的创建独立出去，不想了解产品创建细节的时候可以考虑使用。不得不说不，原型模式给了我们多一种创建对象，并且不依赖具体对象的选择。

实现要点

- .NET 中使用 Object 的 MemberwiseClone()方法来实现浅拷贝，通过序列化和反序列化实现深拷贝，后者代价比较大，选择何时的拷贝方式。
- 原型模式同样需要抽象类型和具体类型，通过相对稳定的抽象类型来减少或避免客户端的修改可能性。
- 在代码中，我们把敌人作为抽象类型，抽象层次很高。完全可以把步兵作为抽象类型，下面有普通步兵，手榴弹步兵等等，再有一个坦克作为抽象类型，下面还有普通坦克和导弹坦克。这样 GameScene 可能就需要从两种抽象类型克隆出许多步兵和坦克。不管怎么样抽象，只要是对象类型由原型实例所指定，新对象通过原型实例做拷贝，那么这就是原型模式。

注意事项

- 注意选择深拷贝和浅拷贝。
- 拷贝原型并进行修改意味着原型需要公开更多的数据，对已有系统实现原型模式可能修改的代价比较大。

无废话 C#设计模式之六：Builder

意图

将一个复杂的构建与其表示相分离，使得同样的构建过程可以创建不同的表示。

场景

在电脑城装机总有这样的经历。我们到了店里，先会有一个销售人员来询问你希望装的机器是怎么样的配置，他会给你一些建议，最终会形成一张装机单。和客户确定了装机配置以后，他会把这张单字交给提货的人，由他来准备这些配件，准备完成后交给装机技术人员。技术人员会把这些配件装成一个整机交给客户。

不管是什么电脑，它总是由 CPU、内存、主板、硬盘以及显卡等部件构成的，并且装机的过程总是固定的：

- 把主板固定在机箱中

- 把 CPU 安装到主板上
- 把内存安装到主板上
- 把硬盘连接到主板上
- 把显卡安装到主板上

但是，每台兼容机的部件都各不相同的，有些配置高一点，有些配置低一点，这是变化点。对于装机技术人员来说，他不需要考虑这些配件从哪里来的，他只需要把他们组装在一起了，这是稳定的装机流程。要把这种变化的配件和稳定的流程进行分离就需要引入 Builder 模式。

示例代码

```
using System;

using System.Collections.Generic;

using System.Text;

using System.Reflection;

namespace BuilderExample
{

    class Program

    {
```


cb.SetupMainboard();**cb.Set upCpu();****cb.Set upMemory();****cb.Set upHarddisk();****cb.Set upVi deocard();**

```
Console.WriteLine(">>>>>>>>>>>Build " + cb.Name + "
```

Completed");

```
Console.WriteLine();
```

}

}

```
abstract class ComputerBuilder
```

{

```
protectedstringname;
```

publicstringName

{

```
get{return name; }
```

```
        set{ name =value; }

    }

    protected Computer computer;

    public Computer Computer

    {

        get{ return computer; }

        set{ computer =value; }

    }

    public Computer Builder()

    {

        computer =new Computer();

    }

    public abstract void Set upMi nboard();

    public abstract void Set upCpu();
```

```

publicabstractvoidSet upMemory();

publicabstractvoidSet upHarddi sk();

publicabstractvoidSet upVi deocard();

}

classOfficeComputerBuilder:ComputerBuilder

{

    publicOffi ceComputerBui lder()

    {

        name ="OfficeComputer";

    }

    publicoverridevoidSet upMi nboard()

    {

        computer.Mi nboard ="Abit 升技 LG-95C 主板(Intel 945GC 芯片组/LGA
775/1066MHz) ";

    }

```



```
public override void SetupCpu()

{

    computer.Cpu = "Intel 英特尔赛扬 D 336 (2.8GHz/LGA 775/256K/533MHz)";

}


public override void SetupMemory()

{

    computer.Memory = "Patriot 博帝 DDR2 667 512MB 台式机内存";

}


public override void SetupHarddisk()

{

    computer.Harddisk = "Hitachi 日立 SATAII 接口台式机硬盘(80G/7200转/8M)盒装";

}


public override void SetupVideocard()

{


```

```
        computer.Videocard ="主板集成";

    }

}

classGameComputerBuilder:ComputerBuilder

{

    publicGameComputerBuilder()

    {

        name ="GameComputer";

    }

    publicoverridevoidSetupMainboard()

    {

        computer.Mainboard ="GIGABYTE 技嘉 GA-965P-DS3 3.3主板(INTEL P965
东莞产)";

    }

    publicoverridevoidSetupCpu()
```

```
{

    computer.Cpu ="Intel 英特尔酷睿 E4400 (2.0GHz/LGA 775/2M/800MHz)盒
装";

}

public override void SetupMemory()

{

    computer.Memory ="G.SKILL 芝奇 F2-6400CL5D-2GBNQ DDR2 800 1G*2台
式机内存";

}

public override void SetupHarddisk()

{

    computer.Harddisk ="Hitachi 日立 SATAII 接口台式机硬盘(250G/7200转/8M)
盒装";

}

public override void SetupVideocard()

{
```

```
        computer.Videocard ="七彩虹逸彩 GT-GD3 UP 烈焰战神 H10显卡(GeForce  
8600GT/256M/DDR3)支持 HDMI!";
```

```
    }
```

```
}
```

```
classComputer
```

```
{
```

```
    privatestringvideocard;
```

```
    publicstringVideocard
```

```
{
```

```
        get{returnvideocard; }
```

```
        set{ videocard =value; }
```

```
}
```

```
    privatestringcpu;
```

```
    publicstringCpu
```

```
{
```

```
        get{ return cpu; }

        set{ cpu =value; }

    }

private string mainboard;

public string Mainboard

{

    get{ return mainboard; }

    set{ mainboard =value; }

}

private string memory;

public string Memory

{

    get{ return memory; }

    set{ memory =value; }
```

```
}
```

```
private string harddisk;
```

```
public string Harddisk
```

```
{
```

```
    get { return harddisk; }
```

```
    set { harddisk = value; }
```

```
}
```

```
public void ShowSystemInfo()
```

```
{
```

```
    Console.WriteLine("=====SystemInfo=====");
```

```
    Console.WriteLine("CPU:" + cpu);
```

```
    Console.WriteLine("MainBoard:" + mainboard);
```

```
    Console.WriteLine("Memory:" + memory);
```

```
    Console.WriteLine("VideoCard:" + videocard);
```

```
    Console.WriteLine("HardDisk:" + harddisk);
```

```

    }
}
}

```

代码执行结果如下图：

```
C:\WINDOWS\system32\cmd.exe

>>>>>>>>>>>>>>>>Start Building OfficeComputer
>>>>>>>>>>>>>>>>Build OfficeComputer Completed

=====SystemInfo=====
CPU:Intel 英特尔赛扬D 336 <2.8GHz/LGA 775/256K/533MHz>
MainBoard:Abit升技 LG-95C 主板<Intel 945GC芯片组/LGA 775/1066MHz>
Memory:Patriot博帝 DDR2 667 512MB 台式机内存
VideoCard:主板集成
HardDisk:Hitachi日立 SATAII接口台式机硬盘 <80G/7200转/8M>盒装

>>>>>>>>>>>>>>>>Start Building GameComputer
>>>>>>>>>>>>>>>>Build GameComputer Completed

=====SystemInfo=====
CPU:Intel 英特尔酷睿2 E4400 <2.0GHz/LGA 775/2M/800MHz>盒装
MainBoard:GIGABYTE技嘉 GA-965P-DS3 3.3 主板<INTEL P965 东莞产>
Memory:G.SKILL 芝奇 F2-6400CL5D-2GBNQ DDR2 800 1G*2台式机内存
VideoCard:七彩虹 逸彩 8600GT-GD3 UP烈焰战神 H10 显卡 <GeForce 8600GT/256M/DDR3>支持HDMI!
HardDisk:Hitachi日立 SATAII接口台式机硬盘 <250G/7200转/8M>盒装
请按任意键继续...
```

代码说明

- ComputerFactory 是建造者模式的指导者。指导者做的是稳定的建造工作，假设它就是一个技术人员，他只是在按照固定的流程，把配件组装成计算机的重复劳动工作。他不知道他现在组装的是一台游戏电脑还是一台办公用电脑，他也不知道他往主板上安装的

内存是1G 还是2G 的。呵呵，看来是不称职的技术人员。

- ComputerBuilder 是抽象建造者角色。它主要是用来定义两种接口，一种接口用于规范产品的各个部分的组成。比如，这里就规定了组装一台电脑所需要的5个工序。第二种接口用于返回建造后的产品，在这里我们没有定义抽象方法，反正建造出来的总是电脑。
- OfficeComputerBuilder 和 GameComputerBuilder 是具体的建造者。他的工作就是实现各建造步骤的接口，以及实现返回产品的接口，在这里后者省略了。
- Computer 就是建造出来的复杂产品。在代码中，我们的各种建造步骤都是为创建产品中的各种配件服务的，Computer 定义了一个相对具体的产品，在应用中可以把这个产品进行比较高度的抽象，使得不同的具体建造者甚至可以建造出完全不同的产品。
- 看看客户端的代码，用户先是选择了一个具体的 Builder，用户应该很明确它需要游戏电脑还是办公电脑，但是它可以对电脑一无所知，由销售人员给出一个合理的配置单。然后用户让 ComputerFactory 去为它组装这个电脑。组装完成后 ComputerFactory 开机，给用户验收电脑的配置是否正确。
- 你或许觉得 ComputerBuilder 和是抽象工厂模式中的抽象工厂角色差不多，GameComputerBuilder 又像是具体工厂。其实，建造者模式和抽象工厂模式的侧重点不同，前者强调一个组装的概念，一个复杂对象由多个零件组装而成并且组装是按照一定的标准顺序进行的，而后者强调的是创建一系列产品。建造者模式适用于组装一台电脑，而抽象工厂模式适用于提供用户笔记本电脑、台式电脑和掌上电脑的产品系列。

何时采用

- 从代码角度来说，如果你希望分离复杂类型构建规则和类型内部组成，或者希望把相同的构建过程用于构建不同类型的时候可以考虑使用建造者模式。
- 从应用角度来说，如果你希望解耦产品的创建过程和产品的具体配件，或者你希望为所有产品的创建复用一套稳定并且复杂的逻辑的时候可以考虑使用建造者模式。

实现要点

- 对象的构建过程由指导者完成，具体的组成由具体建造者完成，表示与构建分离。
- 建造者和指导者是建造者模式的关键点，如果进行合并或省略就可能会转变到模版方法模式。
- 如果对象的建造步骤是简单的，并且产品拥有一致的接口可以转而使用工厂模式。

注意事项

- 返回产品的方法是否必须，是否一定要在抽象建造者中有接口根据实际情况而定。如果他们有统一的接口可以在抽象建造者中体现这个抽象方法，如果没有统一的接口（比如，生产毫无关联的产品）则可以在具体建造者中各自实现这个方法，如果创建的产品是一种产品，那么甚至可以省略返回产品的接口（本文的例子就是这样）。

无废话 C#设计模式之七：Adapter

意图

把一个类的接口变换成客户端所期待的另一种接口，从而使原本接口不匹配而无法在一起工作的两个类能够在一起工作。

场景

假设网络游戏的客户端程序分两部分。一部分是和服务端通讯的大厅部分，大厅部分提供的功能有道具购买、读取房间列表、创建房间以及启动游戏程序。另一部分就是游戏程序了，游戏程序和大厅程序虽然属于一个客户端，但是由不同的公司在进行开发。游戏大厅通过实现约定的接口和游戏程序进行通讯。

一开始的设计就是，大厅程序是基于接口方式调用游戏程序启动游戏场景方法的。在大厅程序开发接近完成的时候，公司决定和另外一家游戏公司合作，因此希望把大厅程序能适用另一个游戏。而这个新游戏的遵循的是另一套接口。是不是可以避免修改原先调用方法来启动场景呢？或许你会说，既然只有一个方法修改，那么修改一下也无妨，我们假设大厅程序和游戏程序之间有100个接口，其中的大部分都有修改呢？因为游戏程序接口的修改，大厅程序可能要修改不止100个地方。这样接口的意义何在呢？

此时可以考虑使用 Adapter 模式来适配这种接口的不匹配情况。

示例代码

```
using System;

using System.Collections.Generic;

using System.Text;

namespace AdapterExample
{
    class Program
    {
        static void Main(string[] args)
        {
            Lobby lobby = new Lobby();

            lobby.CreateRoom("HalfPaper");

            lobby.StartGame();
        }
    }
}
```

```
interface IGame
```

```
{
```

```
    void StartScene(string sceneName);
```

```
    void EnterPlayer(string playerName);
```

```
}
```

```
class Lobby
```

```
{
```

```
    private string sceneName;
```

```
    public void CreateRoom(string sceneName)
```

```
    {
```

```
        this.sceneName = sceneName;
```

```
    }
```

```
    public void StartGame()
```

```
    {
```

```
        IGame game = new GameAdapter();
```

```
        game.StartScene(sceneName);

        game.EnterPlayer("yzhu");

    }

}

class Game

{

    public void LoadScene(string sceneName, string token)

    {

        if (token == "Abcd1234")

        {

            Console.WriteLine("Loading " + sceneName + "...");

        }

        else

        {

            Console.WriteLine("Invalid token!");

        }

    }

    public void EnterPlayer(int playerID)

    {

        Console.WriteLine("player: " + playerID + " entered");

    }

}
```

```
}
```

```
}
```

```
class GameAdapter: IGame
```

```
{
```

```
    private Game game = new Game();
```

```
    public void StartScene(string sceneName)
```

```
    {
```

```
        game.LoadScene(sceneName, "Abcd1234");
```

```
    }
```

```
    public void EnterPlayer(string playerName)
```

```
    {
```

```
        game.EnterPlayer(GetPlayerIDByPlayerName(playerName));
```

```
    }
```

```
    private int GetPlayerIDByPlayerName(string playerName)
```

```
{  
  
    return 12345;  
  
}  
  
}  
  
}
```

代码执行结果如下图：



```
C:\WINDOWS\system32\cmd.exe  
Loading HalfPaper...  
player:12345 entered  
请按任意键继续...
```

代码说明

- 可以看到，原先的接口中，启动游戏场景只需要一个参数，就是游戏场景名，而进入新的玩家需要提供玩家 ID（新游戏都使用玩家 ID 而不使用玩家账户名）。
- IGame 接口就是适配器模式中的目标角色，这是客户所期待的接口。也是针对老的游戏程序所遵循的接口。
- Lobby 类相当于调用方或者客户，它原先的代码可能是如下的：

```
IGame game = new Game();
```

但是由于接口的改变，现在不能直接实例化游戏类，只能实例化适配器类型。虽然还是需要改动，但是这个改动是很小的，而且完全可以通过用动态加载程序集来消除这种改动。

- GameAdapter 类是适配器角色，它是适配器模式的核心，用于把源接口转变为目标接口。在这里，我们看到，它实现目标接口。
- Game 类型是源角色，或者说是需要适配的对象。或许它也遵循了另外一套接口，不过我们不是很关心这个，因此代码中也没有体现。
- 使用了适配器模式后，客户端代码没有做什么修改。客户端代码老老实实的依赖接口，它并没有错，如果因此依赖对象的修改而需要大幅度修改就很无辜了，我们在适配器中把本来没有关联的两个接口适配在了一起。我们可以看到，适配器做的不仅仅是换一换方法名，如果源角色和目标角色的差异非常大，那么适配器需要做很多工作。

何时采用

- 从代码角度来说， 如果需要调用的类所遵循的接口并不符合系统的要求或者说并不是客户所期望的，那么可以考虑使用适配器。
- 从应用角度来说， 如果因为产品迁移、合作模块的变动，导致双方一致的接口产生了一致，或者是希望在两个关联不大的类型之间建立一种关系的情况下可以考虑适配器模式。

实现要点

- 适配器模式是否能成功运用的关键在于代码本身是否是基于接口编程的，如果不是的话，那么适配器无能为力。
- 适配器模式的实现很简单，基本的思想就是适配器一定是遵循目标接口的。
- 适配器模式的变化比较多，可以通过继承和组合方式进行适配，适配器可以是一组适配器产品，适配器也可以是抽象类型。
- 适配器模式和 Facade 的区别是，前者是遵循接口的，后者可以是不遵循接口的，比较灵活。

- 适配器模式和 Proxy 的区别是，前者是为对象提供不同的接口，或者为对象提供相同接口，并且前者有一点后补的味道，后者是在设计时就会运用的。

注意事项

- 在对两个无关类进行适配的时候考虑一下适配的代价，一个非常庞大的适配器可能会对系统性能有影响。

无废话 C#设计模式之八：Facade

意图

为子系统的一组接口提供一个一致的界面，Facade 模式定义了一个高层接口，这个接口使得这一子系统更加容易使用。

场景

在一个为游戏充值的网站中，创建订单需要与三个外部接口打交道：

- 用户系统：根据用户名获取用户 ID、查看用户是否已经激活了游戏
- 卡系统：查看某种类型的充值卡是否还有库存
- 充值系统：创建一个订单，并且返回订单号

如果直接让网站和三个外部接口发生耦合，那么网站因为外部系统接口修改而修改的概率就很大了，并且就这些小接口来说并不是十分友善，它们提供的大多数是工具方法，具体怎么去使用还是要看充值网站创建订单的逻辑。

Facade 的思想就是在小接口上封装一个高层接口，屏蔽子接口的调用，提供外部更简洁，更易用的接口。

示例代码

```
using System;  
  
using System.Collections.Generic;
```

```
using System.Text;

namespace FacadeExample
{
    class Program
    {
        static void Main(string[] args)
        {
            PayFacade pf = new PayFacade();

            Console.WriteLine("order:" + pf.CreateOrder("yzhu", 0, 1, 12) + " created");
        }
    }

    class PayFacade
    {
        private AccountSystem account = new AccountSystem();

        private CardSystem card = new CardSystem();

        private PaySystem pay = new PaySystem();
    }
}
```

```

public string CreateOrder(string userName, int cardID, int cardCount, int areaID)
{
    int userID = account.GetUserIDByUserName(userName);

    if(userID == 0)

        return string.Empty;

    if(!account.UserIsActive(userID, areaID))

        return string.Empty;

    if(!card.CardHasStock(cardID, cardCount))

        return string.Empty;

    return pay.CreateOrder(userID, cardID, cardCount);
}
}

```

```

class AccountSystem
{
    public bool UserIsActive(int userID, int areaID)
    {

```

```
return true;
```

```
}
```

```
public int GetUserIDByUserName(string userName)
```

```
{
```

```
    return 123;
```

```
}
```

```
}
```

```
class CardSystem
```

```
{
```

```
    public bool CardHasStock(int cardID, int cardCount)
```

```
{
```

```
        return true;
```

```
}
```

```
}
```

```
class PaySystem
```

```
{

    public String CreateOrder(int userID, int cardID, int cardCount)

    {

        return "0000000001";

    }

}

}
```

代码执行结果如下图：

代码说明

- PayFacade 类就是门面类型，提供给客户端调用，它本身调用子接口。可以看到，创建一个订单首先要根据用户名获取用户 ID、然后要看用户是否已经激活了游戏、然后看充值卡是否有库存，最后才是创建订单。
- AccountSystem、CardSystem 以及 PaySystem 就是子接口，它们提供了账户、卡以及充值相关的一些接口方法。
- Facade 模式太常用了，把和多方关联的逻辑代码再进行一次封装，提供一个高层接口

就是 Facade 的思想。比如在做论坛程序的时候，一些操作需要调用权限访问模块（发帖、管理帖子），另外一些操作可以直接调用（首页论坛板块、登陆）数据访问模块，由网站来做这个判断并调用不同的子模块并不合适，可以加一个业务逻辑层来统一接受网站各种操作请求，这其实就是 Facade。

何时采用

- 从代码角度来说，如果你的程序有多个类是和一组其它接口发生关联的话可以考虑在其中加一个门面类型。
- 从应用角度来说，如果子系统的接口是非常细的，调用方也有大量的逻辑来和这些接口发生关系，那么就可以考虑使用 Facade 把客户端与子系统的直接耦合关系进行化解。你可能会说，子系统改了门面不是照样改？的确是需要改，但是如果客户端本身的工作已经比较复杂，或者说可能有多个需要调用门面的地方，这个时候门面的好处就体现了。

实现要点

- 通过一个高层接口让子系统和客户端不发生直接关联，使客户端不受子系统变化的影响。
- Facade 不仅仅针对代码级别，在构架上，特别是 WEB 应用程序的构架上，Facade 的应用非常普遍。

注意事项

- Facade 不一定只能是一个，可以考虑把门面进行细分。

无废话 C#设计模式之九：Proxy

意图

为其他对象提供一种代理以控制对这个对象的访问。

场景

代理模式非常常用，大致的思想就是通过为对象加一个代理来降低对象的使用复杂度、或是提升对象使用的友好度、或是提高对象使用的效率。在现实生活中也有很多代理的角色，比如明星的经纪人，他就是一种代理，经纪人为明星处理很多对外的事情，目的是为了节省被代理对象也就是明星的时间。保险代理人帮助投保人办理保险，目的降低投保的复杂度。

在开发中代理模式也因为目的不同效果各不相同。比如，如果我们的网站程序是通过 .NET Remoting 来访问帐号服务的。在编写代码的时候可能希望直接引用帐号服务的 DLL，直接实例化帐号服务的类型以方便调试。那么，我们可以引入 Proxy 模式，做一个帐号服务的代理，网站只需要直接调用代理即可。在代理内部实现正式和测试环境的切换，以及封装调用 .NET Remoting 的工作。

示例代码

```
using System;

using System.Collections.Generic;

using System.Text;

namespace ProxyExample
{

    class Program

    {

        static void Main(string[] args)

        {

            AccountProxy ap = new AccountProxy();

            ap.Register();

        }

    }

    interface IAccount
```

```
{

    void Register();

}

class Account:IAccount

{

    public void Register()

    {

        System.Threading.Thread.Sleep(1000);

        Console.WriteLine("Done");

    }

}

class AccountProxy:IAccount

{

    readonly bool isDebug = true;

    IAccount account;

    public Account Proxy()
```

```
{

    if (isDebug)

        account = new Account();

    else

        account = (IAccount)Activator.GetObject(typeof(IAccount), "uri");

}

public void Register()

{

    Console.WriteLine("Please wait...");

    account.Register();

}

}

}
```

代码执行结果如下图：

代码说明

- IAccount 就是抽象主题角色。代理对象和被代理对象都遵循这个接口，这样代理对象就能替换被代理对象。
- AccountProxy 就是代理主题角色。代理主题通常会存在一些逻辑或预处理或后处理操作，不会仅仅是对操作的转发。
- Account 就是真实主题角色。

何时采用

- 代理模式应用非常广泛，如果你希望降低对象的使用复杂度、或是提升对象使用的友好度、或是提高对象使用的效率都可以考虑代理模式。

实现要点

- 代理对象和被代理对象都遵循一致的接口。
- 在某些情况下，可以不必保持接口一致性，如果封装确实需要损失一些透明度，那么也可以认为是 Proxy。

注意事项

- Proxy、Facade 以及 Adapter 可能都是对对象的一层封装，侧重点不同。Proxy 基于一致的接口进行封装，Facade 针对封装子系统，转化为高层接口，而 Adapter 的封装是处于适配接口的目的。

无废话 C#设计模式之十：Flyweight

意图

运用共享技术有效地支持大量细粒度的对象。

场景

在比较底层的系统或者框架级的软件系统中，通常存在大量细粒度的对象。即使细粒度的对象，如果使用的数量级很高的话会占用很多资源。比如，游戏中可能会在无数个地方使用到模型数据，虽然从数量上来说模型对象会非常多，但是从本质上来说，不同的模型可能也就这么几个。

此时，我们可以引入享元模式来共享相同的模型对象，这样就可能大大减少游戏对资源（特别是内存）的消耗。

示例代码

```
using System;

using System.Collections;

using System.Text;

using System.IO;

namespace FlyweightExample
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine(GC.GetTotalMemory(false));

            Random rnd = new Random();

            ArrayList al = new ArrayList();

            for(int i = 0; i < 10000; i++)
            {
                string modelName = rnd.Next(2).ToString();

                Model model = ModelFactory.GetInstance().GetModel(modelName);
            }
        }
    }
}
```

```

        //Model model = new Model(modelName);

        al.Add(model);

    }

    Console.WriteLine(GC.GetTotalMemory(false));

    Console.ReadLine();

}

}

class Model

{

    private byte[] data;

    public Model(string modelName)

    {

        data = File.ReadAllBytes("c:\\\" + modelName + ".txt");

    }

}

```

```
class ModelFactory

{

    private Hashtable<model Name> model List = new Hashtable();

    private static ModelFactory instance;

    public static ModelFactory Get Instance()

    {

        if(instance == null)

            instance = new ModelFactory();

        return instance;

    }

    public Model Get Model (string model Name)

    {

        Model model = model List[model Name] as Model;

        if(model == null)

            model List.Add(model Name, new Model(model Name));

        return model;

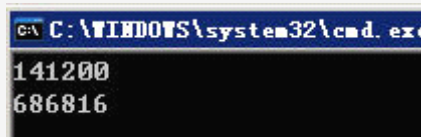
    }

}
```

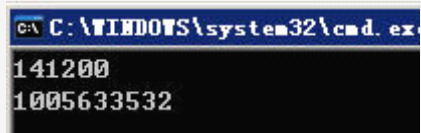


```
}  
  
}  
  
}
```

代码执行结果如下图（前面是使用享元模式的结果，后面是没有使用享元模式的结果）：



C:\WINDOWS\system32\cmd.exe
141200
686816



C:\WINDOWS\system32\cmd.exe
141200
1005633532

代码说明

- 这里的 ModelFactory 就是享元工厂角色。它的作用是创建和管理享元对象。可以看到，每加载一个模型都会在 Hashtable 中记录一下，之后如果客户端还是需要这个模型的话就直接把已有的模型对象返回给客户端，而不是重新在内存中加载一份模型数据。
- ModelFactory 本身应用了 Singleton，因为如果实例化多个享元工厂是的话就起不到统一管理和分配享元对象的目的了。
- Model 就是享元角色。在构造方法中传入 modelName，然后它从指定路径加载模型数据，并且把数据放入字段中。
- 从代码的运行结果中可以看到，如果没有应用享元模式，那么在内存中就会有10000套

模型对象，由于一共就2个模型，所以9998个对象是可以通过享元来消除的。

何时采用

- 系统中有大量耗费了大量内存的细粒度对象，并且对外界来说这些对没有任何差别的（或者说经过改造后可以是没有差别的）。

实现要点

- 享元工厂维护一张享元实例表。
- 享元不可共享的状态需要在外部维护。
- 按照需求可以对享元角色进行抽象。

注意事项

- 享元模式通常针对细粒度的对象，如果这些对象比较拥有非常多的独立状态（不可共享的状态），或者对象并不是细粒度的，那么就不适合运用享元模式。维持大量的外蕴状态不但会使逻辑复杂而且并不能节约资源。
- 享元工厂中维护了享元实例的列表，同样也需要占用资源，如果享元占用的资源比较小或者享元的实例不是非常多的话（和列表元素数量差不多），那么就不适合使用享元，关键

还是在于权衡得失。

无废话 C#设计模式之十一：Composite

意图

将对象组合成树形结构以表示“部分-整体”的层次结构。Composite 模式使得用户对单个对象和组合对象的使用具有一致性。

场景

我们知道，一个网络游戏通常会有多个游戏大区。每一个游戏大区会有很多游戏服务器（一个游戏大区就是一组游戏服务器）。每一个游戏服务器上会有不同的服务（可以是多个服务）。这是一个明显的部分-整体关系，假设我们现在需要制作一个服务器管理工具，用于显示所有大区、服务器以及服务的信息，并且能开启这些服务（可以是单独开启一个服务，也可以是开启整个服务器上的所有服务，也可以是开启整个大区的所有服务）。

可以看到，游戏服务器和游戏大区都是一个组合对象，而游戏服务是最底层的节点。客户端在开启一个游戏大区服务的时候，必须和游戏服务器以及游戏服务进行依赖，而在开启游戏服务器上所有服务的时候，必须和游戏服务进行依赖。试想一下，如果一个公司的总裁在管理上不但需要和各总监以及经理进行沟通，还有和底层的员工沟通，那么总裁是不是会太忙碌了一点？由此，我们引入组合模式，使组合对象和单个对象具有一样的表现形式。

示例代码

```
using System;

using System.Collections.Generic;

using System.Text;

namespace CompositeExample
{
    class Program
    {
        static void Main(string[] args)
        {
            Element server1 = new GameServer("GS1", "192.168.0.1");

            server1.Add(new GameService("Lobby1", 1, "S5Lobby1"));

            server1.Add(new GameService("Gate1", 2, "S5Gate1"));

            server1.Add(new GameService("DataExchange1", 3, "S5DataExchange1"));

            server1.Add(new GameService("Rank1", 4, "S5Rank1"));
        }
    }
}
```

```
server1.Add(newGameService("Log1", 5,"S5Log1"));

Elementserver2 =newGameServer("GS2","192.168.0.2");

server2.Add(newGameService("Lobby2", 1,"S5Lobby2"));

server2.Add(newGameService("Gate2", 2,"S5Gate2"));

server2.Add(newGameService("DataExchange2", 3,"S5DataExchange1"));

server2.Add(newGameService("Rank2", 4,"S5Rank2"));

server2.Add(newGameService("Log2", 5,"S5Log2"));

Elementarea =newGameArea("电信区");

area.Add(server1);

area.Add(server2);

area.Display();

area.Start();

area.Stop();

}

}
```

```
abstractclassElement
```

```
{
```

```
protected string name;
```

```
public Element(string name)
```

```
{
```

```
    this.name = name;
```

```
}
```

```
public abstract void Add(Element element);
```

```
public abstract void Remove(Element element);
```

```
public abstract void Display();
```

```
public abstract void Start();
```

```
public abstract void Stop();
```

```
}
```

```
class GameService: Element, IComparable<GameService>
```

```
{
```

```
    private int serviceType;
```

```
    private string serviceName;
```

```
public CanService(string name, int serviceType, string serviceName)
```

```
    : base(name)
```

```
{
```

```
    this.serviceName = serviceName;
```

```
    this.serviceType = serviceType;
```

```
}
```

```
public override void Add(Element element)
```

```
{
```

```
    throw new ApplicationException("xxx");
```

```
}
```

```
public override void Remove(Element element)
```

```
{
```

```
    throw new ApplicationException("xxx");
```

```
}
```

```
public override void Display()

{

    Console.WriteLine(string.Format("name:{0},serviceType:{1},serviceName:{2}",
name, serviceType, serviceName));

}


public override void Start()

{

    Console.WriteLine(string.Format("{0} started", name));

}


public override void Stop()

{

    Console.WriteLine(string.Format("{0} stopped", name));

}


public int CompareTo(GameService other)

{
```



```
return other.serviceType.CompareTo(serviceType);
```

```
}
```

```
}
```

```
class GameServer : Element
```

```
{
```

```
private string serverIP;
```

```
private List<GameService> serviceList = new List<GameService>();
```

```
public GameServer(string name, string serverIP)
```

```
    : base(name)
```

```
{
```

```
    this.serverIP = serverIP;
```

```
}
```

```
public override void Add(Element element)
```

```
{
```

```
        serviceList.Add((GameService)element);

    }

    public override void Remove(Element element)

    {

        serviceList.Remove((GameService)element);

    }

    public override void Display()

    {

        Console.WriteLine(string.Format("{0}{1}({2}){3}", new string('+', 10), name,
serverIP, new string('+', 10)));

        foreach(Element element in serviceList)

        {

            element.Display();

        }

    }

}
```

```
public override void Start()

{

    serviceList.Sort();

    Console.WriteLine("=====Starting the whole " + name
+"=====");

    for(int i = 0; i < serviceList.Count; i++ )

    {

        serviceList[i].Start();

    }

    Console.WriteLine("=====The whole " + name + "
started=====");

}


public override void Stop()

{

    Console.WriteLine("=====Stopping the whole " + name
+"=====");

    for(int i = serviceList.Count -1; i >= 0; i--)
```

```

    {

        serviceList[i].Stop();

    }

    Console.WriteLine("=====The whole " + name + "
stopped=====");

}

}

class GameArea:Element

{

    private List<GameServer> serverList = new List<GameServer>();

    public GameArea(string name)

        :base(name) { }

    public override void Add(Element element)

    {

        serverList.Add((GameServer)element);
    }
}

```

```
}
```

```
public override void Remove(Element element)
```

```
{
```

```
    serverList.Remove((GameServer)element);
```

```
}
```

```
public override void Display()
```

```
{
```

```
    Console.WriteLine(new string('-', 20));
```

```
    Console.WriteLine("        " + name);
```

```
    Console.WriteLine(new string('-', 20));
```

```
    foreach(Element element in serverList)
```

```
    {
```

```
        element.Display();
```

```
    }
```

```
}
```

```
public override void Start()

{

    Console.WriteLine("=====Starting the whole " + name
+"=====");

    foreach (Element element in serverList)

    {

        element.Start();

    }

    Console.WriteLine("=====The whole " + name + "
started=====");

}

public override void Stop()

{

    Console.WriteLine("=====Stopping the whole " + name
+"=====");

    foreach (Element element in serverList)

    {
```

```
        element.Stop();  
  
    }  
  
    Console.WriteLine("=====The whole " + name + "  
stopped=====");  
  
    }  
  
    }  
  
}
```

代码执行结果如下图：

```

=====
        电信1区
=====
+++++++GS1<192.168.0.1>+++++++
name:Lobby1,serviceType:1,serviceName:$5Lobby1
name:Gate1,serviceType:2,serviceName:$5Gate1
name:DataExchange1,serviceType:3,serviceName:$5DataExchange1
name:Rank1,serviceType:4,serviceName:$5Rank1
name:Log1,serviceType:5,serviceName:$5Log1
+++++++GS2<192.168.0.2>+++++++
name:Lobby2,serviceType:1,serviceName:$5Lobby2
name:Gate2,serviceType:2,serviceName:$5Gate2
name:DataExchange2,serviceType:3,serviceName:$5DataExchange1
name:Rank2,serviceType:4,serviceName:$5Rank2
name:Log2,serviceType:5,serviceName:$5Log2
=====Starting the whole 电信1区=====
=====Starting the whole GS1=====
Log1 started
Rank1 started
DataExchange1 started
Gate1 started
Lobby1 started
=====The whole GS1 started=====
=====Starting the whole GS2=====
Log2 started
Rank2 started
DataExchange2 started
Gate2 started
Lobby2 started
=====The whole GS2 started=====
=====The whole 电信1区 started=====
=====Stopping the whole 电信1区=====
=====Stopping the whole GS1=====
Lobby1 stopped
Gate1 stopped
DataExchange1 stopped
Rank1 stopped
Log1 stopped
=====The whole GS1 stopped=====
=====Stopping the whole GS2=====
Lobby2 stopped
Gate2 stopped
DataExchange2 stopped
Rank2 stopped
Log2 stopped
=====The whole GS2 stopped=====
=====The whole 电信1区 stopped=====
请按任意键继续. . .

```

代码说明

- Element 类型就是抽象构件的角色，它给组合对象以及单个对象提供了一个一致的接口，使得它们都能有一致的行为。
- 这里就出现一个问题，组合对象需要通过 Add 和 Remove 方法来为其添加子节点，而最底层的树叶构件下并没有任何子节点，在接口中定义这些操作子节点方法的方式叫做透明方式的合成模式，缺点就是不够安全，容易在运行时出现异常。如果把这些操作子节点的方法定义从抽象构件中删除，由各树枝构件来实现的话，就是安全方式的合成模式，缺点也就是不够透明。具体怎么做还要根据自己的需求。
- GameService 当然就是树叶构件 如果调用它的 Add 以及 Remove 方法会抛出一个异常。当然，你也可以以其它方式来记录这种逻辑错误。
- GameServer 是一个树枝构件。一个游戏服务器上会有多个游戏服务。这里注意到一点，在开启服务器上所有服务的时候，我们对服务进行了排序，排序是按照服务的类型进行的。然后，我们按照服务类型从大到小的次序开启了服务。一般一个网络游戏的服务会有很多种，而这些服务的开启是有先后次序的，先开记录日志的服务、排名服务、再开数据交换的服务、最后才是大厅服务等。我们不可能让用户进入大厅的时候没有地方写数据和日志吧。开启服务是按照次序的，那么关闭服务也就是按照相反的次序了。
- 从上面这点，我们就可以看到组合模式的好处了，树枝构件在管理树叶构件的时候通常还会有一些逻辑，不会是简单的增加和删除操作。如果这些工作交给客户端去做的话，就太不合理了。
- 最上层的树枝构件就是 GameArea 类型。它并没有什么特殊的地方。

何时采用

- 从代码角度来说，如果类型之间组成了层次结构，你希望使用统一的接口来管理每一个层次的类型的时候。
- 从应用角度来说，如果你希望把一对多的关系转化为一对一的关系的时候。

实现要点

- 使用透明模式还是安全模式根据自己的需要定。
- 在某些情况下，树叶构件可以访问树枝构件获取一些信息。
- 如果树叶构件数量比较多，树枝构件频繁遍历子节点的话可以考虑进行缓存。
- 既然所有对象有了统一的接口，客户端应该针对抽象构件进行编程。

注意事项

- 无

无废话 C#设计模式之十二：Bridge

意图

将抽象部分与实现部分分离，使它们都可以独立的变化。

场景

还是说我们要做的网络游戏，多个场景需要扩充的问题我们已经采用了创建型模式来解决。现在的问题就是，不仅仅是游戏场景会不断扩充，而且游戏的模式也在不断扩充。比如，除了最基本的战斗模式之外，还会有道具模式，金币模式等。

对于这种在多个维度上都会有变化或扩充需求的项目来说，可以考虑引入桥接模式。或许你会说，不管是什么场景，不管什么模式，都可以是抽象场景的一个子类，但是，如果这样的话，4个场景和3种模式就会产生12个子类，而10个场景5种模式就会有50个子类。一味进行继承并不是什么好方法，桥接模式的思想是把继承转化为组合，把乘法（ $10 \times 5 = 50$ ）转化为加法（ $10 + 5 = 15$ ）。

示例代码

```
using System;

using System.Collections.Generic;

using System.Text;

namespace BridgeExample
{
```

```
class Program
```

```
{
```

```
    static void Main(string[] args)
```

```
    {
```

```
        PatrixScene halfPaper = new HalfPaper();
```

```
        halfPaper.Mode = new GoldMode();
```

```
        halfPaper.LoadScene();
```

```
        PatrixScene matrix = new Matrix();
```

```
        matrix.Mode = new PrpoertyMode();
```

```
        matrix.LoadScene();
```

```
    }
```

```
}
```

```
abstract class PatrixScene
```

```
{
```

```
    protected GameMode mode;
```

```
    public GameMode Mode
```

```
{  
  
    get{return mode; }  
  
    set{ mode =value; }  
  
}
```

```
publicabstractvoidLoadScene();  
  
}
```

```
classHalfPaper:PatrxScene
```

```
{  
  
    publicoverridevoidLoadScene()  
  
    {  
  
        Console.WriteLine("Load HalfPaper Completed");  
  
        mode.InitScene();  
  
    }  
  
}
```

```
classMatrix:PatrxScene
```

```
{  
  
    public override void LoadScene()  
  
    {  
  
        Console.WriteLine("Load Matrix Completed");  
  
        mode.InitScene();  
  
    }  
  
}
```

```
abstract class GameMode
```

```
{  
  
    public abstract void InitScene();  
  
}
```

```
class PrpoertyMode : GameMode
```

```
{  
  
    public override void InitScene()  
  
    {  
  
        Console.WriteLine("Init Property Mode Completed");  
  
    }  
  
}
```

```
    }

}

class GoldMode:GameMode

{

    public override void InitScene()

    {

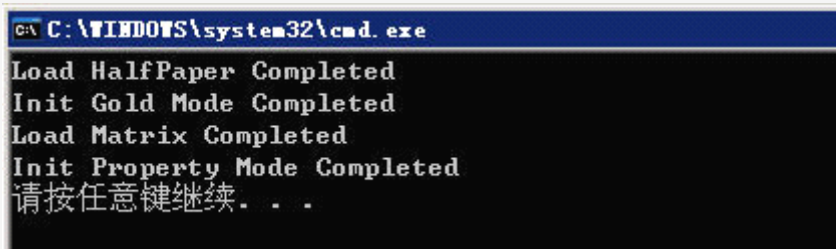
        Console.WriteLine("Init Gold Mode Completed");

    }

}

}
```

代码执行结果如下图：



```
C:\WINDOWS\system32\cmd.exe
Load HalfPaper Completed
Init Gold Mode Completed
Load Matrix Completed
Init Property Mode Completed
请按任意键继续...
```

代码说明

- PatrxScene 类是抽象化角色。虽然说针对第一维度也就是游戏场景，PatrxScene 也是一个抽象，但是我觉得这里说的抽象化和实现化还是针对第二维度的，也就是游戏模式。
- GameMode 类就是实现化角色。你或许会说对于多个维度 把哪个作为抽象化角色呢？虽然维度是一个平行的概念，但是对于 Bridge 模式来说，我觉得它是把相对高层的角色作为抽象化角色，而把比较底层的操作作为实现化角色的。比如，对于场景和模式来说，模式是为场景服务的，我们就把场景作为抽象化角色。
- HalfPaper 和 Matrix 都是修正抽象化角色。按照 GOF 的定义是说修正父类的抽象化定义。其实，我觉得抽象化角色不一定必须是对方法有默认实现，并且由子类进行修正。
- PropertyMode 和 GoldMode 是具体实现化角色。它们用来实现实现化角色定义的接口。
- 从一个角度来说，抽象化和修正抽象化角色相对应实现化和具体实现化角色，从另外一个角度来说，抽象化和实现化角色对应修正抽象化和具体实现化角色。
- 客户端代码中直接选择合适的具体实现化角色。看到这里，你可能觉得和策略模式很像。其实，策略模式针对面更小一点，一是针对算法替换，二是只针对一个维度的变化点，因此它也就只有一个抽象角色。

何时采用

- 从代码角度来说，如果类型的继承是处于2个目的（违背单一职责原则）的话可以使用 Bridge 模式避免过多的子类。
- 从应用角度来说，如果应用会在多个维度上进行变化，客户端希望两个维度（场景、游戏模式）的对象相对独立，动态耦合（客户端决定哪个场景和哪个游戏模式耦合）的时候可以考虑 Bridge 模式。

实现要点

- 选择合适的类型作为抽象化角色（第一维度）。
- 抽象化角色和实现化角色通过组合进行关联。
- 抽象和实现不绑定，允许客户端作切换。

注意事项

- 无

无废话 C#设计模式之十三：Decorator

意图

动态地给一个对象添加一些额外的职责。就增加功能来说，Decorator 模式相比生成子类更为灵活。

场景

在设计网络游戏的武器系统时，开始并没有考虑到武器的强化和磨损。之后，策划人员说希望给游戏增加强化系统和修理系统，那么我们的武器类型就需要对外提供强化、磨损、修理等方法了。发生这种改动是我们最不愿意看到的，按照设计原则，我们希望功能的扩展尽可能不要修改原来的程序。你可能会想到使用继承来实现，但是策划人员的需求是有的武器能磨损能修理，不能强化，有的武器能强化，但是不会磨损，有的武器既能强化还能磨损和修理。遇到这样的情况，继承的方案可能不适合了，一来继承的层次可能会很多，二来子类的数量可能会很多。

由此，引入装饰模式来解决这个问题。装饰模式使得我们能灵活赋予类额外的职责，并且使得设计和继承相比更合理。

示例代码

```
using System;  
  
using System.Collections.Generic;  
  
using System.Text;
```

```
namespaceDecoratorExample
```

```
{
```

```
classProgram
```

```
{
```

```
staticvoidMain(string[] args)
```

```
{
```

```
Weaponw =newRifle();
```

```
w.ShowInfo();
```

```
EnhanceenhancedWeapon =newEnhance(w);
```

```
enhancedWeapon.EnhanceAmmo();
```

```
enhancedWeapon.ShowInfo();
```

```
WearwornWeapon =newWear(w);
```

```
wornWeapon.WearByRate(0.8);
```

```
wornWeapon.ShowInfo();
```

```
}
```

```
}
```

```
abstractclassWeapon
```

```
{  
  
    private double ammo;  
  
    public double Ammo  
    {  
  
        get { return ammo; }  
  
        set { ammo = value; }  
  
    }  
  
    private double attack;  
  
    public double Attack  
    {  
  
        get { return attack; }  
  
        set { attack = value; }  
  
    }  
  
    private double speed;
```

```
public double Speed
```

```
{
```

```
    get{ return speed; }
```

```
    set{ speed = value; }
```

```
}
```

```
private String name;
```

```
public String Name
```

```
{
```

```
    get{ return name; }
```

```
    set{ name = value; }
```

```
}
```

```
public abstract void ShowInfo();
```

```
}
```

```
class Rifle: Weapon
```

```
{

    public Rifle()

    {

        this.Ammo = 100;

        this.Attack = 10;

        this.Speed = 5;

        this.Name = "Rifle";

    }

    public override void ShowInfo()

    {

        Console.WriteLine(string.Format("ammo\t{0}", Ammo));

        Console.WriteLine(string.Format("attack\t{0}", Attack));

        Console.WriteLine(string.Format("speed\t{0}", Speed));

        Console.WriteLine(string.Format("name\t{0}", Name));

    }

}
```

```
abstract class Decorator: Weapon
{
    protected Weapon w;

    public Decorator(Weapon w)
    {
        this.w = w;
    }

    public override void ShowInfo()
    {
        w.ShowInfo();
    }
}

class Enhance: Decorator
{
    public Enhance(Weapon w) : base(w) { }
}
```

```
public void EnhanceAmmo()

{

    w.Ammo += 20;

    Console.WriteLine(">>>>>>>>>Enhanced");

}

}
```

```
class Wear:Decorator

{

    public Wear(Weapon w) :base(w) { }

    public void WearByRate(double rate)

    {

        w.Speed = w.Speed * rate;

        w.Attack = w.Attack * rate;

        Console.WriteLine(">>>>>>>>>Worn");

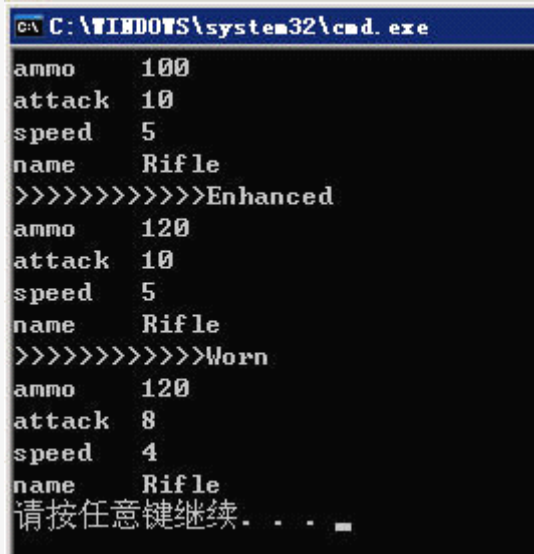
    }

}
```



```
}  
  
}
```

代码执行结果如下图：



```
C:\WINDOWS\system32\cmd.exe  
ammo    100  
attack  10  
speed   5  
name    Rifle  
>>>>>>>>>>Enhanced  
ammo    120  
attack  10  
speed   5  
name    Rifle  
>>>>>>>>>>Worn  
ammo    120  
attack   8  
speed    4  
name    Rifle  
请按任意键继续. . .
```

代码说明

- Weapon 是抽象构件角色。
- Rifle 是具体构件角色，实现抽象构件的接口。
- Decorator 是装饰角色。装饰角色有两个特点，一是继承了抽象构件的接口，二是有一个构件角色的实例。
- Enhance 和 Wear 是具体装饰角色，它们负责给构件附加责任。
- 客户端在使用装饰角色的时候并没有针对抽象构件进行编程，因为我们确实需要使用具

体装饰角色提供的额外方法，这种类型的装饰叫做半透明装饰。

何时采用

- 从代码角度来说，如果你觉得由于功能的交叉扩展不会导致非常多的子类或者非常多的继承层次的话可以考虑装饰模式。
- 从应用角度来说，如果你希望动态给类赋予或撤销一些职责，并且可以任意排列组合这些职责的话可以使用装饰模式。

实现要点

- 让装饰角色还继承抽象构件角色也是装饰模式最大的特点，目的就是给抽象构件增加职责，对外表现为装饰后的构件。
- 让装饰角色拥有构件角色实例的目的就是让构件能被多个装饰对象来装饰。
- 在具体应用中可以灵活一点，不一定要有抽象构件和装饰角色。但是，装饰对象继承装饰对象并且拥有它实例的两大特点需要体现。
- 透明装饰一般通过在基类方法前后进行扩充实现，半透明装饰一般通过新的接口实现。

注意事项

- 装饰模式和桥接模式的区别是，前者是针对功能的扩展，本质上还是一样东西，而后者针对多维护变化。装饰模式的思想在于扩展接口而桥接模式的思想是分离接口。
- 装饰类可能会比较琐碎，并且不利于复用，装饰模式在增加了灵活性的同时也降低了封装度，在实际应用中可以和其它模式配合。

无废话 C#设计模式之十四：Template Method

意图

定义一个操作中的算法的骨架，而将一些步骤延迟到子类中。Template Method 使得子类可以不改变一个算法的结构即可重定义该算法的某些特定步骤。

场景

模版方法是非常容易理解的设计模式，一来是因为它没有过多结构上的交错，二来是因为这种代码复用技术对于掌握 OO 知识的人来说非常容易可以想到，很可能你已经在很多地方运用了模版方法。在运用一些设计模式的时候常常也会一起运用模版方法，甚至有的设计模式本身就带有模版方法的思想。

今天，我们给出这样一个实际的例子。做过银行支付、支付宝支付的人都知道，一个支付的过程是基于两个接口的。提交接口和网关返回接口，虽然各大网关的支付接口格式不同，比如有的网关对于支付金额的参数是 money，有的网关又是 amount，但是从支付的提交过程来说，我们一般都会

经历以下步骤：

- 获取订单信息，验证订单的合法性
- 生成用于提交到各大网关的表单
- 记录日志
- 把表单提交到相应的网关

对于各个网关，生成的提交表单以及记录日志的方式是不一样的，但是整个支付流程以及流程中的获取订单信息、提交表单的过程是一样的。由此引入模版方法模式来复用不变的部分，把可变的分留给子类去实现。

示例代码

```
using System;

using System.Collections.Generic;

using System.Text;

namespace TemplateMethodExample
{

    class Program

    {
```

```
static void Min(string[] args)
```

```
{
```

```
    PayGateway pg = new IPSGateway();
```

```
    pg.SubmitOrder(new Order());
```

```
}
```

```
}
```

```
class Order
```

```
{
```

```
}
```

```
class SubmitForm
```

```
{
```

```
}
```

```
abstract class PayGateway
```

```
{
```

```
    protected abstract void WriteLog(SubmitForm f);
```

```
protected abstract SubmitForm GenerateOrderForm(Order order);
```

```
public void SubmitOrder(Order order)
```

```
{
```

```
    if (order == null)
```

```
    {
```

```
        Console.WriteLine("Invalid Order");
```

```
        return;
```

```
    }
```

```
    SubmitForm sf = GenerateOrderForm(order);
```

```
    if (sf == null)
```

```
    {
```

```
        Console.WriteLine("Generate Submit Form Failed");
```

```
        return;
```

```
    }
```

```
    WriteLog(sf);
```

```
}
```

```
}
```

```
class IPSGateway: PayGateway
{
    protected override void WriteLog(SubmitForm sf)
    {
        Console.WriteLine("Log Wrote");
    }

    protected override SubmitForm GenerateOrderForm(Order order)
    {
        Console.WriteLine("Submit Form Generated");

        return new SubmitForm();
    }
}
```

代码执行结果如下图：

```
C:\WINDOWS\system32\cmd.exe
Submit Form Generated
Log Wrote
请按任意键继续. . .
```

代码说明

- PayGateway 类型是抽象模版角色。它定义了支付过程不变的部分，并且把变化部分定义为抽象操作，让子类去实现。其中的 SubmitOrder 方法是模版方法。
- IPSGateway 类型是具体模版角色。它代表了某一种支付网关，并且按照这种支付网关的接口标准来实现生成提交表单和记录日志的操作。

何时采用

- 如果某些类型的操作拥有共同的实现骨架和不同的实现细节的话，可以考虑使用模版方法来封装统一的部分。

实现要点

- 复用算法的骨架，将可变的实现细节留给子类实现。
- 留给子类实现的方法需要在父类中定义，可以是抽象方法也可以是带有默认实现的方法。

注意事项

- 模版方法可以说是最不像设计模式的设计模式，通常很多设计模式会和模版方法一起使用。

无废话 C#设计模式之十五：Strategy

意图

定义一系列的算法，把它们一个一个封装起来，并且使它们可相互替换。本模式使得算法可以独立于它的客户而变化。

场景

在开发程序的时候，我们经常会根据环境不同采取不同的算法对对象进行处理。比如，在一个新闻列表页面需要显示所有新闻，而在一个新闻搜索页面需要根据搜索关键词显示匹配的新闻。如果在新闻类内部有一个 ShowData 方法的话，那么我们可能会传入一个 searchWord 的参数，并且在方法内判断如果参数为空则显示所有新闻，如果参数不为空则进行搜索。如果还有分页的需求，那么还可能在方法内判断是否分页等等。

这样做有几个不好的地方，一是 ShowData 方法太复杂了，代码不容易维护并且可能会降低

性能，二是如果还有其它需求的话势必就需要修改 ShowData 方法，三是不利于重用一些算法的共同部分。由此引入策略模式，把算法进行封装，使之可以灵活扩展和替换。

示例代码

```
using System;

using System.Collections.Generic;

using System.Text;

using System.Collections;

namespace StrategyExample
{
    class Program
    {
        static void Main(string[] args)
        {
            Data data = new Data();

            data.Add("aaa");
        }
    }
}
```

```
data.Add("bbb");
```

```
data.Add("ccc");
```

```
data.Add("abc");
```

```
data.Show();
```

```
data.SetShowDataStrategy(newSearchData("a"));
```

```
data.Show();
```

```
data.SetShowDataStrategy(newShowPagedData(2,1));
```

```
data.Show();
```

```
}
```

```
}
```

```
abstractclassShowDataStrategy
```

```
{
```

```
abstractpublicvoidShowData(IListdata);
```

```
}
```

```
classShowAllData:ShowDataStrategy
```

```
{
```

```
public override void ShowData(ICollection data)

{

    for(int i = 0; i < data.Count; i++)

    {

        Console.WriteLine(data[i]);

    }

}

}

class ShowPagedData : ShowDataStrategy

{

    private int pageSize;

    private int pageIndex;

    public ShowPagedData(int pageSize, int pageIndex)

    {

        this.pageSize = pageSize;

        this.pageIndex = pageIndex;

    }

}
```

```
}
```

```
public override void ShowData(IList data)
```

```
{
```

```
    for(int i = pageSize * pageIndex; i < pageSize * (pageIndex + 1); i++)
```

```
    {
```

```
        Console.WriteLine(data[i]);
```

```
    }
```

```
}
```

```
}
```

```
class SearchData: ShowDataStrategy
```

```
{
```

```
    private string searchWord;
```

```
    public string SearchWord
```

```
    {
```

```
        get { return searchWord; }
```

```
        set{ searchWord =value; }

    }

    publicSearchData(stringsearchWord)

    {

        this.searchWord = searchWord;

    }

    publicoverridevoidShowData(IListdata)

    {

        for(inti = 0; i < data.Count; i++)

        {

            if(data[i].ToString().Contains(searchWord))

            Console.WriteLine(data[i]);

        }

    }

}
```

```
class Data

{

    private ShowDataStrategy strategy;

    private IList<data> data = new ArrayList();

    public void SetShowDataStrategy(ShowDataStrategy strategy)

    {

        this.strategy = strategy;

    }

    public void Show()

    {

        if (strategy == null)

            strategy = new ShowAllData();

        Console.WriteLine(strategy.GetType().ToString());

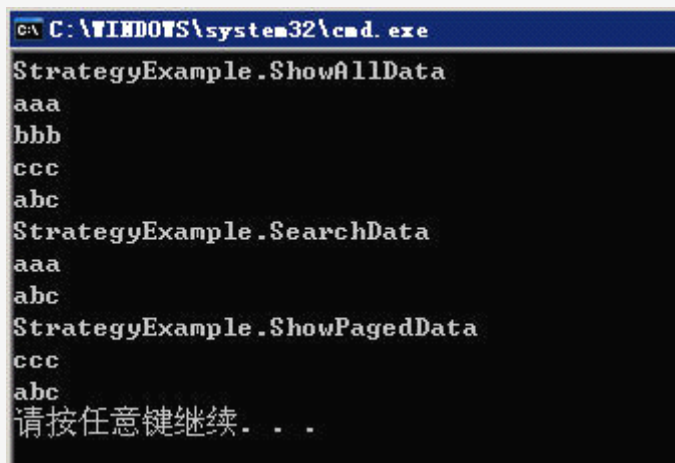
        strategy.ShowData(data);

    }

}
```

```
public void Add(string name)
{
    data.Add(name);
}
}
```

代码执行结果如下图：



C:\WINDOWS\system32\cmd.exe

```
StrategyExample.ShowAllData
aaa
bbb
ccc
abc
StrategyExample.SearchData
aaa
abc
StrategyExample.ShowPagedData
ccc
abc
请按任意键继续. . .
```

代码说明

- Data 类就是环境或者说上下文角色，持有对策略角色的引用。在这里，我们通过一个方法来设置环境使用的策略，你也可以根据需求在构造方法中传入具体策略对象。
- ShowDataStrategy 抽象类就是抽象策略角色，它定义了策略共有的接口。

- ShowAllData、ShowPagedData 以及 SearchData 类都是具体策略角色，它们实现真正的算法或行为。
- 客户端在调用的时候才决定去使用哪种策略模式。
- 可以看到，由于显示数据由各个具体策略类来实现，使得环境角色的复杂度降低了很多。并且如果以后还需要增加新的显示数据方式的话只需要增加新的具体策略类（实现抽象策略接口）就可以了，环境类的代码不需要做改动。对于各具体策略实现过程中可复用的部分也可以放在抽象策略类中实现。

何时采用

- 从代码角度来说，如果一个类有多种行为，并且在类内部通过条件语句来实现不同的行为的时候可以把这些行为单独封装为策略类。
- 从应用角度来说，如果系统需要选择多种算法中的一种并且希望通过统一的接口来获取算法的输出的话可以考虑策略模式。

实现要点

- 在环境角色中拥有策略角色的实例。
- 如果策略角色需要使用环境中的数据，一般可以让环境把数据传给所有策略角色，或者可以让环境把自身传给策略角色，前者会带来不必要的通讯开销，后者会使环境和策略角

色发生紧密耦合。根据需要选择合适的方式。

- 环境角色可以在客户端没有提供策略角色的时候可以实现模式的策略。

注意事项

- 策略模式的缺点是客户端需要了解具体的策略，因此仅当客户端能做出这样选择的时候才去使用策略模式。
- 过多的策略对象可能会增加系统负担，可以考虑把各种策略角色实现为无状态对象的享元，需要保存的额外状态由环境角色进行统一管理和处理。

无废话 C#设计模式之十六：State

意图

允许一个对象在其内部状态改变时改变它的行为。对象看起来似乎修改了它的类。

场景

我们在制作一个网上书店的网站，用户在书店买了一定金额的书后可以升级为银会员、黄金会员，不同等级的会员购买书籍有不同的优惠。你可能会想到可以在 User 类的 BuyBook 方法中判断

用户历史消费的金额来给用户不同的折扣，在 GetUserLevel 方法中根据用户历史消费的金额来输出用户的等级。带来的问题有三点：

- 不用等级的用户给予的优惠比率是经常发生变化的，一旦变化是不是就要修改 User 类呢？
- 网站在初期可能最高级别的用户是黄金会员，而随着用户消费金额的累计，我们可能要增加钻石、白金等会员类型，这些会员的折扣又是不同的，发生这样的变化是不是又要修改 User 类了呢？
- 抛开变化不说，User 类承担了用户等级判断、购买折扣计算等复杂逻辑，复杂的 User 类代码的可维护性会不会很好呢？

由此引入 State 模式，通过将对象和对象的状态进行分离，把对象状态的转化以及由不同状态产生的行为交给具体的状态类去做，解决上述问题。

示例代码

```
using System;

using System.Collections.Generic;

using System.Text;

namespace StateExample
```

```
{

class Program

{

    static void Main(string[] args)

    {

        User user = new User("zhuye");

        user.BuyBook(2000);

        user.BuyBook(2000);

        user.BuyBook(2000);

        user.BuyBook(2000);

    }

}

class User

{

    private UserLevel userLevel;

    public UserLevel UserLevel
```

```
{  
  
    get{returnuserLevel; }  
  
    set{ userLevel =value; }  
  
}
```

```
privatestringuserName;
```

```
privatedoublepai dMney;
```

```
publicdoublePai dMney
```

```
{  
  
    get{returnpai dMney; }  
  
}
```

```
publicUser(stringuserName)
```

```
{  
  
    this.userName = userName;  
  
    this.paidMoney = 0;
```

```

        this.UserLevel = new NormalUser(this);

    }

    public void BuyBook(double amount)

    {

        Console.WriteLine(string.Format("Hello {0}, You have paid ${1}, You Level is
{2}.", userName, paidMoney, userLevel.GetType().Name));

        double realAmount = userLevel.CalcRealAmount(amount);

        Console.WriteLine("You only paid $" + realAmount + " for this book.");

        paidMoney += realAmount;

        userLevel.StateCheck();

    }

}

abstract class UserLevel

{

    protected User user;

```

```
public UserLevel (User user)
```

```
{
```

```
    this.user = user;
```

```
}
```

```
public abstract void StateCheck();
```

```
public abstract double CalcReal Amount (double amount );
```

```
}
```

```
class DiamondUser:UserLevel
```

```
{
```

```
    public DiamondUser (User user)
```

```
    {
        :base(user) { }
```

```
    public override double CalcReal Amount (double amount )
```

```
    {
```

```
        return amount * 0.7;
```

```
    }
```

```
public override void StateCheck()

{

}

}
```

```
class GoldUser : UserLevel

{

    public GoldUser(User user)

        : base(user) { }

    public override double CalcRealAmount(double amount)

    {

        return amount * 0.8;

    }

    public override void StateCheck()
```



```
{  
  
    if(user.PaidMoney > 5000)  
  
        user.UserLevel =newDiamondUser(user);  
  
}  
  
}
```

```
classSilverUser:UserLevel
```

```
{  
  
    publicSilverUser(Useruser)  
  
        :base(user) { }  
  
    publicoverridedoubleCalcRealAmount(doubleamount )  
  
    {  
  
        returnamount * 0.9;  
  
    }  
  
    publicoverridevoidStateCheck()  
  
    {
```

```

        if(user.PaidMoney > 2000)

            user.UserLevel =newGoldUser(user);

        }

    }

classNormalUser:UserLevel

{

    publicNormal User(Useruser)

        :base(user) { }

    publicoverridedoubleCalcReal Amount(doubleamount )

    {

        returnamount * 0.95;

    }

    publicoverridevoidStateCheck()

    {

        if(user.PaidMoney > 1000)

```

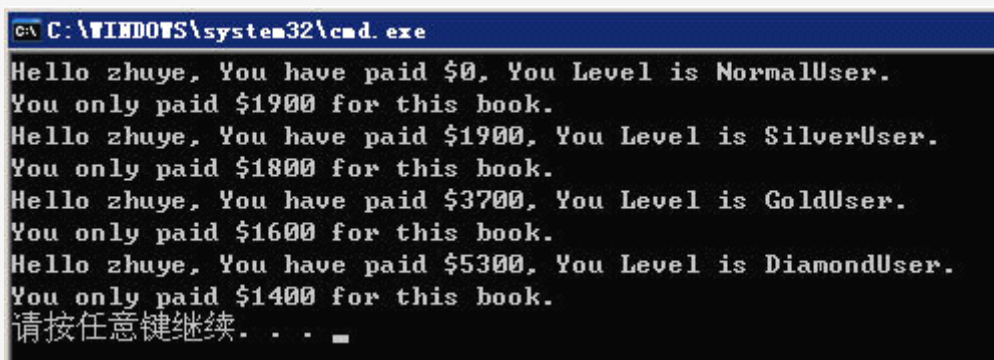
```
        user.UserLevel = new SilverUser(user);

    }

}

}
```

代码执行结果如下图：



```
C:\WINDOWS\system32\cmd.exe
Hello zhuye, You have paid $0, You Level is NormalUser.
You only paid $1900 for this book.
Hello zhuye, You have paid $1900, You Level is SilverUser.
You only paid $1800 for this book.
Hello zhuye, You have paid $3700, You Level is GoldUser.
You only paid $1600 for this book.
Hello zhuye, You have paid $5300, You Level is DiamondUser.
You only paid $1400 for this book.
请按任意键继续. . .
```

代码说明

- User 类型是环境角色。它的作用一是定义了客户端感兴趣的方法（购买书籍），二是拥有一个状态实例，它定义了用户的当前状态（普通会员、银会员、黄金会员还是钻石会员）。
- UserLevel 类型是抽象状态角色。它的作用一是定义了和状态相关的行为的接口，二是拥有一个环境实例，用于在一定条件下修改环境角色的抽象状态。
- NormalUser、SilverUser、GoldUser 以及 DiamondUser 就是具体状态了。它们都实现了抽象状态角色定义的接口。
- 为 User 转化 UserLevel 的操作是在各个具体状态中进行的。在这里可以看到这种状态

的转化是有序列的，这样也只会前后两个状态产生依赖。假设现在的会员系统中是没有钻石用户的，那么 GoldUser 的 StateCheck()方法中应该是没有什么代码的，即使以后要加 DiamondUser 类，我们也只需要修改 GoldUser 的 SateCheck()方法，以根据一定的规则来为环境转化状态。

- 在这里，我们在环境中调用了 StateCheck 方法，在实际应用中，你可以根据需要在抽象状态中引入模版方法，对外公开这个模版方法，并且在模版方法中调用行为方法和转化状态的方法，当然，具体的行为还是由具体状态来实现的。

何时采用

- 从代码角度来说，如果一个类有多种状态，并且在类内部通过的条件语句判断的类状态来实现不同行为时候可以把这些行为单独封装为状态类。
- 从应用角度来说，如果一个对象有多种状态，如果希望把对象状态的转化以及由不同状态产生的行为交给具体的状态类去做，那么可以考虑状态模式。

实现要点

- 在环境角色中拥有状态角色的实例。
- 在状态角色中拥有环境角色的实例用于在具体状态中修改环境角色的状态。
- 状态对象之间的依赖可以通过加载外部配置的转化规则表等方法来消除。

- 状态模式和策略模式的主要区别是，前者的行为实现方式是由条件决定的，并且应当能不在客户端干预的情况下自己迁移到合适的状态，而后的行为实现方式是由客户端选择的，并且能随时替换。

注意事项

- 过多的状态对象可能会增加系统负担，可以考虑把各种状态角色实现为无状态对象的享元，需要保存的额外状态由环境角色进行统一管理和处理。

无废话 C#设计模式之十七：Chain Of Resp.

意图

使多个对象都有机会处理请求，从而避免请求的发送者和接受者之间的耦合关系。将这些对象连成一条链，并沿着这条链传递该请求，直到有一个对象能处理请求为止。

场景

假设我们在制作一个游戏的客服系统，客服有三种角色，分别是普通客服、客服经理和客服总监。玩家在网站中提问后，根据问题的分类和重要性处理的流程不一样。规则如下：

- 分类为1（游戏问题）和2（角色问题）的问题会由普通客服处理。
- 分类为3（充值问题）和4（账户问题）的问题会由客服经理处理。
- 普通客服和客服经理都没有能处理的问题由客服总监进行处理。
- 所有的问题都分为普通问题和重要问题，如果问题是重要问题则需要上一级的客服进行审核（普通客服回复的问题需要客服经理审核、客服经理回答的问题需要客服总监审核）。

这个处理的业务规则比较复杂，您可能会想到创建三个具体客服类型继承抽象客服，然后根据问题的性质传给不同的具体客服来处理。可是这样做有几个缺点：

- 客户端需要知道三个角色能处理的问题属性，三个角色能处理的问题应该只有三个角色自己清楚就可以了。
- 客户端需要和三个角色发生耦合，并且在一个处理过程中会先后调用不同的角色。普通客服在回复了重要问题后应该通知客服经理来审核，现在这个过程交给了客户端去做

客户端知道了太多客服处理问题的细节，对于玩家来说他只需要知道把这些问题告诉客服人员并且得到客服人员的处理结果，具体背后的处理流程是怎么样的它不用知道。由此，引入责任链模式来解决这些问题。

示例代码

```
using System;  
  
using System.Collections.Generic;
```

```

using System.Text;

namespace ChainOfRespExample
{
    class Program
    {
        static List<CustomerService> gmTeam = new List<CustomerService>();

        static List<CustomerService> managerTeam = new List<CustomerService>();

        static List<CustomerService> directorTeam = new List<CustomerService>();

        static Random random = new Random();

        static void InitCOR()
        {
            if (managerTeam.Count > 0)
            {
                foreach (CustomerService cs in gmTeam)
                {
                    cs.SetLeader(managerTeam[random.Next(managerTeam.Count)]);
                }
            }
        }
    }
}

```

```

else

{

    foreach(CustomerServicecs in gmTeam)

        cs.SetLeader(directorTeam[random.Next(directorTeam.Count)]);

}

foreach(CustomerServicecs in managerTeam)

    cs.SetLeader(directorTeam[random.Next(directorTeam.Count)]);

// These configs above depends on business logic.

}

staticvoidInitGM()

{

    for(inti = 1; i <= 9; i++)

    {

        CustomerServicegn =newNormalGM("gm"+ i);

        gn.SetResponsibleCaseCategory(newint[] { 1, 2 });

        gmTeam.Add(gn);

    }

}

```



```

        for(int i = 1; i <= 2; i++)

        {

            CustomerServiceManager manager = new GMManager("manager" + i);

            manager.SetResponsibleCaseCategory(new int[] { 3, 4 });

            managerTeam.Add(manager);

        }

        for(int i = 1; i <= 1; i++)

            directorTeam.Add(new GMDirector("director" + i));

        // These configs above should be from database.

    }

    static void Main(string[] args)

    {

        InitGM();

        InitCOR();

        CustomerService gm = gmTeam[random.Next(gmTeam.Count)];

        gm.HandleCase(new Case(1, false));

        Console.WriteLine(Environment.NewLine);
    }

```

```
gm = gmTeam[random.Next(gmTeam.Count)];
```

```
gm.HandleCase(newCase(2,true));
```

```
Console.WriteLine(Environment.NewLine);
```

```
gm = gmTeam[random.Next(gmTeam.Count)];
```

```
gm.HandleCase(newCase(3,false));
```

```
Console.WriteLine(Environment.NewLine);
```

```
gm = gmTeam[random.Next(gmTeam.Count)];
```

```
gm.HandleCase(newCase(4,true));
```

```
Console.WriteLine(Environment.NewLine);
```

```
gm = gmTeam[random.Next(gmTeam.Count)];
```

```
gm.HandleCase(newCase(5,true));
```

```
}
```

```
}
```

```
classCase
```

```
{
```

```
privateintcaseCategory;
```

```
public int CaseCategory
```

```
{
```

```
    get { return caseCategory; }
```

```
}
```

```
private bool ImportantCase;
```

```
public bool ImportantCase
```

```
{
```

```
    get { return ImportantCase; }
```

```
}
```

```
private string reply;
```

```
public string Reply
```

```
{
```

```
    get { return reply ; }
```

```
    set { reply = value; }
```

```
}
```

```
public Case(int caseCategory, boolean importantCase)
```

```
{
```

```
    this.caseCategory = caseCategory;
```

```
    this.importantCase = importantCase;
```

```
}
```

```
}
```

```
abstract class CustomerService
```

```
{
```

```
    protected CustomerService leader;
```

```
    protected List<int> responsibleCaseCategory = new List<int>();
```

```
    protected String name;
```

```
    public String Name
```

```
{
```

```
        get { return name; }  
    }
```

```
}
```

```
public CustomerService(string name)
```

```
{
```

```
    this.name = name;
```

```
}
```

```
public void SetLeader(CustomerService leader)
```

```
{
```

```
    this.leader = leader;
```

```
}
```

```
public abstract void HandleCase(Case gameCase);
```

```
public void SetResponsibleCaseCategory(int[] responsibleCaseCategory)
```

```
{
```

```
    foreach (int i in responsibleCaseCategory)
```

```
        this.responsibleCaseCategory.Add(i);
```

```
}
```

```
}
```

```
class NormalGM: CustomerService
```

```
{
```

```
    public NormalGM(string name) : base(name) { }
```

```
    public override void HandleCase(Case gameCase)
```

```
{
```

```
    if (responsibleCaseCategory.Contains(gameCase.Category))
```

```
{
```

```
        gameCase.Reply = "OK";
```

```
        Console.WriteLine("The case has been replied by " + name);
```

```
        if (gameCase.IsImportantCase)
```

```
{
```

```
            Console.WriteLine("The reply should be checked.");
```

```
            leader.HandleCase(gameCase);
```

```
}
```

```

        else

            Console.WriteLine("The reply has been sent to player.");

        }

        elseif(leader !=null)

        {

            Console.WriteLine(string.Format("{0} reports this case to {1}.", name,
leader.Name));

            leader.HandleCase(gameCase);

        }

    }

}

class GManager:CustomerService

{

    public GManager(string name) :base(name) { }

    public override void HandleCase(Case gameCase)

    {

```

```
if(responsibleCaseCategory.Contains(gameCase.Category))

{

    gameCase.Reply = "OK";

    Console.WriteLine("The case has been replied by " + name);

    if(gameCase.ImportantCase)

    {

        Console.WriteLine("The reply should be checked.");

        leader.HandleCase(gameCase);

    }

    else

        Console.WriteLine("The reply has been sent to player.");

}

elseif(gameCase.Reply != null)

{

    Console.WriteLine("The case has been checked by " + name);

    Console.WriteLine("The reply has been sent to player.");

}

elseif(leader != null)
```



```

        {

            Console.WriteLine(string.Format("{0} reports this case to {1}.", name,
leader.Name));

            leader.HandleCase(gameCase);

        }

    }

}

```

```

class GMDirector: CustomerService

```

```

{

    public GMDirector(string name) : base(name) { }

    public override void HandleCase(Case gameCase)

    {

        if(gameCase.Reply != null)

        {

            Console.WriteLine("The case has been checked by " + name);

            Console.WriteLine("The reply has been sent to player.");

```

```
    }

    else

    {

        gameCase.Reply ="OK";

        Console.WriteLine("The case has been replied by " + name);

        Console.WriteLine("The reply has been sent to player.");

    }

}

}

}
```

代码执行结果如下图（后面一图为注释掉 InitGM()方法中初始化 managerTeam 代码的执行结果）：

C:\WINDOWS\system32\cmd.exe

The case has been replied by gm7
The reply has been sent to player.

The case has been replied by gm4
The reply should be checked.
The case has been checked by manager2
The reply has been sent to player.

gm7 reports this case to manager1.
The case has been replied by manager1
The reply has been sent to player.

gm8 reports this case to manager1.
The case has been replied by manager1
The reply should be checked.
The case has been checked by director1
The reply has been sent to player.

gm2 reports this case to manager1.
manager1 reports this case to director1.
The case has been replied by director1
The reply has been sent to player.
请按任意键继续. . .

C:\WINDOWS\system32\cmd.exe

The case has been replied by gm5
The reply has been sent to player.

The case has been replied by gm9
The reply should be checked.
The case has been checked by director1
The reply has been sent to player.

gm5 reports this case to director1.
The case has been replied by director1
The reply has been sent to player.

gm8 reports this case to director1.
The case has been replied by director1
The reply has been sent to player.

gm6 reports this case to director1.
The case has been replied by director1
The reply has been sent to player.
请按任意键继续. . .

代码说明

- Case 类代表了问题。CaseCategory 属性代表了问题的分类，普通客服和客服经理处理不同分类的问题。ImportantCase 属性代表了问题是否是重要问题，如果是重要问题，则需要上级领导审核。Reply 属性代表了客服对问题的回复。
- CustomerService 类是责任链模式中的抽象处理者。我们看到，它定义了下个责任人的引用，并且提供了设置这个责任人的方法。当然，它也定义了统一的处理接口。
- NormalGM 是具体处理者，它实现了处理接口。在这里，我们看到普通客服的处理逻辑是，如果这个问题的分类在它负责的分类之外则直接提交给上级领导进行处理（把对象通过责任链传递），否则就回复问题，回复问题之后看这个问题是否是重要问题，如果是重要问题则给上级领导进行审核，否则问题处理结束。
- GMManager 也是一个具体处理者。客服经理处理问题的逻辑是，首先判断问题的分类是否在其负责的分类之内，如果是的话则进行处理（重要问题同样提交给上级处理），如果不是的话就看问题是否有了回复，如果有回复说明是要求审核的问题，审核后问题回复，如果问题还没有回复则提交给上级处理。
- GMDirector 的处理流程就相对简单了，它并没有上级了，因此所有问题必须在其这里结束。对于没有回复的问题则进行回复，对于要求审核的问题则进行审核。
- 再看看客户端的调用。首先，执行了 InitGM()方法来初始化客服团队的数据，在这里我们的团队中有9个普通客服、2个客服经理和1个客服总监。普通客服只能回复分类1和分

类2的问题，而客服经理只能回复分类3和分类4的问题。

- 然后，调用 `InitCOR()`方法来初始化责任链，在这里我们并没有简单得设置普通客服的上级是客服经理、客服经理的上级是客服总监，而是自动根据是否有客服经理这个角色来动态调整责任链，也就是说如果没有客服经理的话，普通客服直接向客服总监汇报。
- 最后，我们模拟了一些问题数据进行处理。对于客户端（玩家）来说任何普通客服角色都是一样的，因此我们为所有问题随机分配了普通客服作为责任链的入口点。
- 首先来分析有客服经理时的处理情况。分类为1的问题直接由普通客服处理完毕。分类为2的重要问题由普通客服回复后再由客服经理进行审核。分类为3的问题直接由普通客服提交到客服经理进行处理。分类为4的重要问题也直接由普通客服提交到客服经理进行处理，客服经理回复后再提交到客服总监进行审核。分类为5的问题由普通客服提交到客服经理进行处理，客服经理再提交给客服总监进行处理。
- 再来分析没有客服经理时的处理情况。分类为1的问题直接由普通客服处理完毕。分类为2的重要问题由普通客服回复后再由客服总监进行审核。分类为3的问题直接由普通客服提交到客服总监进行处理。分类为4的重要问题也直接由普通客服提交到客服总监进行处理。分类为5的问题也直接由普通客服提交到客服总监进行处理。
- 如果没有责任链模式，这个处理流程将会多么混乱。

何时采用

- 从代码角度来说，如果一个逻辑的处理由不同责任对象完成，客户端希望能自定义这个

处理流程并且不希望直接和多个责任对象发生耦合的时候可以考虑责任链模式。

- 从应用角度来说，如果对一个事情的处理存在一个流程，需要经历不同的责任点进行处理，并且这个流程比较复杂或只希望对外公开一个流程的入口点的话可以考虑责任链模式。其实，责任链模式还可以在构架的层次进行应用，比如.NET 中的层次异常处理关系就可以看作是一种责任链模式。

实现要点

- 有一个抽象责任角色，避免各责任类型之间发生耦合。
- 抽象责任角色中定义了后继责任角色，并对外提供一个方法供客户端配置。
- 各具体责任类型根据待处理对象的状态结合自己的责任范围来判断是否能处理对象，如果不能处理提交给上级责任人处理(也就是纯的责任模式,要么自己处理要么提交给别人)。当然，也可以在部分处理后提交给上级处理(也就是不纯的责任链模式)。
- 需要在客户端链接各个责任对象，如果链接的不恰当，可能会导致部分对象不能被任何一个责任对象进行处理。

注意事项

- 责任链模式和状态模式的区别在于，前者注重责任的传递，并且责任链由客户端进行配置，后者注重对象状态的转换，这个转换过程对客户端是透明的。

无废话 C#设计模式之十八：Command

意图

将一个请求封装为一个对象，从而使你可用不同的请求对客户进行参数化；对请求排队或记录请求日志，以及支持可撤消的操作。

场景

我们知道，网络游戏客户端需要不断把当前人物的信息发送到游戏服务端进行处理（计算合法性、保存状态到数据库等）。假设有这样一种需求，在服务端收到客户端的请求之后需要判断两次请求间隔是不是过短，如果过短的话就考虑可能是游戏外挂，不但不执行当前请求还要把前一次请求进行回滚。暂且把问题简单化一点不考虑客户端和服务端之间的通讯来进行程序设计的话，你可能会创建一个 Man 类型，其中提供了一些人物移动的方法，执行这些方法后，服务端内存中的人物会进行一些坐标的修改。客户端定时调用 Man 类型中的这些方法即可。那么如何实现防外挂的需求呢？你可能会想到在 Man 方法中保存一个列表，每次客户端调用方法的时候把方法名和方法调用的时间保存进去，然后在每个方法执行之前就进行判断。这样做有几个问题：

- Man 类型应该只是负责执行这些操作的，是否应该执行操作的判断放在 Man 类型中是否合适？
- 如果方法的调用还有参数的话，是不是需要把方法名、方法的参数以及方法调用时间都

保存到列表中呢？

- 如果需要根据不同的情况回滚一组行为，比如把 Man 类型的方法分为人物移动和装备损耗，如果客户端发送命令的频率过快希望回滚所有人物移动的行为，如果客户端发送命令的频率过慢希望回滚所有装备损耗的行为。遇到这样的需求怎么实现呢？

由此引入命令模式，命令模式的主要思想就是把方法提升到类型的层次，这样对方法的执行有更多的控制力，这个控制力表现在对时间的控制力、对撤销的控制力以及对组合行为的控制力。

示例代码

```
using System;

using System.Collections.Generic;

using System.Text;

namespace CommandExample
{
    class Program
    {
        static void Main(string[] args)
        {

```



```
Man mn = new Man();

Server server = new Server();

server.Execute(new MoveForward(mn, 10));

System.Threading.Thread.Sleep(50);

server.Execute(new MoveRight(mn, 10));

server.Execute(new MoveBackward(mn, 10));

server.Execute(new MoveLeft(mn, 10));

}

}
```

```
class Man

{

    private int x = 0;

    private int y = 0;

    public void MoveLeft(int i) { x -= i; }

    public void MoveRight(int i) { x += i; }
```

```
public void MoveForward(int i) { y += i; }
```

```
public void MoveBackward(int i) { y -= i; }
```

```
public void GetLocation()
```

```
{
```

```
    Console.WriteLine(string.Format("{0},{1}", x, y));
```

```
}
```

```
}
```

```
abstract class GameCommand
```

```
{
```

```
    private DateTime time;
```

```
    public DateTime Time
```

```
{
```

```
        get { return time; }
```

```
        set { time = value; }
```

```
}
```

```
protected Man man;
```

```
public Man Man
```

```
{
```

```
    get{ return man; }
```

```
    set{ man = value; }
```

```
}
```

```
public GameCommand(Man man)
```

```
{
```

```
    this.time = DateTime.Now;
```

```
    this.man = man;
```

```
}
```

```
public abstract void Execute();
```

```
public abstract void UnExecute();
```

```
}
```

```
class MoveLeft:GameCommand
```

```
{
```

```
    int step;
```

```
    public MoveLeft(Man man,int i) :base(man) {this.step = i; }
```

```
    public override void Execute()
```

```
    {
```

```
        man.MoveLeft(step);
```

```
    }
```

```
    public override void UnExecute()
```

```
    {
```

```
        man.MoveRight(step);
```

```
    }
```

```
}
```

```
class MoveRight:GameCommand

{

    int step;

    public MoveRight(Man mn,int i) :base(mn) {this.step = i; }

    public override void Execute()

    {

        mn.MoveRight(step);

    }

    public override void UnExecute()

    {

        mn.MoveLeft(step);

    }

}
```

```
class MoveForward:GameCommand
```

```
{

    int step;

    public MoveForward(Man mn, int i) : base(mn) { this.step = i; }

    public override void Execute()

    {

        mn.MoveForward(step);

    }

    public override void UnExecute()

    {

        mn.MoveBackward(step);

    }

}

class MoveBackward : GameCommand

{
```

```
int step;
```

```
public MoveBackward(Man mn, int i) : base(mn) { this.step = i; }
```

```
public override void Execute()
```

```
{
```

```
    mn.MoveBackward(step);
```

```
}
```

```
public override void UnExecute()
```

```
{
```

```
    mn.MoveForward(step);
```

```
}
```

```
}
```

```
class Server
```

```
{
```

```
    GameCommand lastCommand;
```

```

public void Execute(GameCommand cmd)

{

    Console.WriteLine(cmd.GetType().Name);

    if (lastCommand != null && (TimeSpan)(cmd.Time - lastCommand.Time)
< new TimeSpan(0, 0, 0, 0, 20))

    {

        Console.WriteLine("Invalid command");

        lastCommand.UnExecute();

        lastCommand = null;

    }

    else

    {

        cmd.Execute();

        lastCommand = cmd;

    }

    cmd.Mn.GetLocation();

}

}

```



```
}
```

代码执行结果如下图：



```
C:\WINDOWS\system32\cmd.exe
MoveForward
<0,10>
MoveRight
<10,10>
MoveBackward
Invalid command
<0,10>
MoveLeft
<-10,10>
请按任意键继续. . .
```

代码说明

- 在代码实例中，我们只考虑了防止请求过频的控制，并且也没有考虑客户端和服务端通讯的行为，在实际操作中并不会这么做。
- Man 类是接受者角色，它负责请求的具体实施。
- GameCommand 类是抽象命令角色，它定义了统一的命令执行接口。
- MoveXXX 类型是具体命令角色，它们负责执行接受者对象中的具体方法。从这里可以看出，有了命令角色，发送者无需知道接受者的任何接口。
- Server 类是调用者角色，相当于一个命令的大管家，在合适的时候去调用命令接口。

何时采用

有如下的需求可以考虑命令模式：

- 命令的发起人和命令的接收人有不同的生命周期。比如，下遗嘱的这种行为就是命令模式，一般来说遗嘱执行的时候命令的发起人已经死亡，命令是否得到有效的执行需要靠律师去做的。
- 希望能让命令具有对象的性质。比如，希望命令能保存以实现撤销；希望命令能保存以实现队列化操作。撤销的行为在 GUI 中非常常见，队列化命令在网络操作中也非常常见。
- 把命令提升到类的层次后我们对类行为的扩展就会灵活很多，别的不说，我们可以把一些创建型模式和结构型模式与命令模式结合使用。

实现要点

- 从活动序列上来说通常是这样的一个过程：客户端指定一个命令的接受者；客户端创建一个具体的命令对象，并且告知接受者；客户端通过调用者对象来执行具体命令；调用者对象在合适的时候发出命令的执行指令；具体命令对象调用命令接受者的方法来落实命令的执行。
- 命令模式从结构上说变化非常多，要点就是一个抽象命令接口。抽象命令接口包含两个含义，一是把方法提升到类的层次，二是使用统一的接口来执行命令。
- 有了前面说的这个前提，我们才可以在调用者角色中做很多事情。比如，延迟命令的执行、为执行的命令记录日志、撤销执行的命令等等。
- 在应用的过程中可以省略一些不重要的角色。比如，如果只有一个执行者或者执行的逻

辑非常简单的话，可以把执行的逻辑合并到具体命令角色中；如果我们并不需要使用调用者来做额外的功能，仅仅是希望通过命令模式来解除客户端和接受者之间耦合的话可以省略调用者角色。

- 如果需要实现类似于宏命令的命令组可以使用组合模式来封装具体命令。
- 如果需要实现 undo 操作，那么命令接受者通常也需要公开 undo 的接口。在应用中，undo 操作往往不是调用一下 undo 方法这么简单，因为一个操作执行后所改变的环境往往是复杂的。

注意事项

- 不要被命令模式复杂的结构所迷惑，如果你不能理解的话请思考这句话“把方法提升到类的层次的好处也就是命令模式的好处”。
- 和把状态或算法提到类的层次的状态模式或策略模式相比，命令模式可能会产生更多的类或对象。

无废话 C#设计模式之十九：Observer

意图

定义对象间的一种一对多的依赖关系,当一个对象的状态发生改变时,所有依赖于它的对象都得

到通知并被自动更新。

场景

这次不说游戏了，假设我们需要在一个 Web 页面上分页显示数据。首先需要有一个分页控制器和一个显示数据的表格。开始，客户的需求很简单，需要两个向前翻页向后翻页的按钮作为控制器，还需要一个 GridView 来显示数据。你可能会这么做：

- 在页面上放两个按钮和一个 GridView 控件
- 点击了下一页按钮判断是否超出了页面索引，如果没有的话更新 GridView 中的数据，然后更新控件的当前页，如果翻页后是最后一页的话把下一页按钮设置为不可用。
- 点击了上一页按钮判断是否超出了页面索引，如果没有的话更新 GridView 中的数据，然后更新控件的当前页，如果翻页后是第一页的话把上一页按钮设置为不可用。

在这里，我们的翻页控件仅仅和 GridView 进行依赖，看似问题不大。没有想到，客户看了 Demo 后觉得这样的体验不好，希望在页面上呈现当前页和总共页。于是，我们又在页面上加了一个 Label 控件，在按钮的点击事件里面再去更新 Label 控件的值。客户挺满意的，随着软件中数据越来越多，总页数达到了几十页，客户觉得前后翻页太不合理的了，希望有一个显示页数列表的分页控制器，客户的这个请求彻底使我们晕了，代码被我们修改的非常混乱：

- 点击了列表分页控件的页数后更新自身状态、通知 GridView 加载数据、通知按钮分页控件更新自身状态、通知 Label 更新页数信息。
- 点击了按钮分页控件后更新自身状态、通知 GridView 加载数据、通知列表分页控件更

新自身状态、通知 Label 更新页数信息。

如果今后页面上还需要针对分页功能有任何修改的话，真不知道怎么去改。由此引入观察者模式来解决这些问题。

示例代码

```
using System;

using System.Collections.Generic;

using System.Text;

namespace ObserverExample
{
    class Program
    {
        static void Main(string[] args)
        {
            ButtonPager buttonPager = new ButtonPager();

            ListPager listPager = new ListPager();
```

```
Control gri dvi ew =new GridView();
```

```
Control l abel =new Label();
```

```
but t onPager .changePageHandl er
```

```
+=new Pager.ChangePageHandler(but t onPager .ChangePage);
```

```
but t onPager .changePageHandl er
```

```
+=new Pager.ChangePageHandler(gri dvi ew.ChangePage);
```

```
but t onPager .changePageHandl er
```

```
+=new Pager.ChangePageHandler(l abel .ChangePage);
```

```
but t onPager .changePageHandl er
```

```
+=new Pager.ChangePageHandler(l i st Pager .ChangePage);
```

```
l i st Pager .changePageHandl er
```

```
+=new Pager.ChangePageHandler(but t onPager .ChangePage);
```

```
l i st Pager .changePageHandl er
```

```
+=new Pager.ChangePageHandler(gri dvi ew.ChangePage);
```

```
l i st Pager .changePageHandl er
```

```
+=new Pager.ChangePageHandler(l abel .ChangePage);
```

```
l i st Pager .changePageHandl er
```

```
+=new Pager.ChangePageHandler(l i st Pager .ChangePage);
```

```
but t onPager .Next Page();
```

```
Console.WriteLine();
```

```
buttonPager.NextPage();
```

```
Console.WriteLine();
```

```
buttonPager.NextPage();
```

```
Console.WriteLine();
```

```
buttonPager.PreviousPage();
```

```
Console.WriteLine();
```

```
buttonPager.PreviousPage();
```

```
Console.WriteLine();
```

```
listPager.SelectPage(2);
```

```
Console.WriteLine();
```

```
listPager.SelectPage(1);
```

```
Console.WriteLine();
```

```
listPager.SelectPage(0);
```

```
}
```

```
}
```

```
abstract class Pager
{
    protected int pageIndex = 0;

    public int PageIndex
    {
        get { return pageIndex; }

        set { pageIndex = value; }
    }

    protected int pageCount = 3;

    public int PageCount
    {
        get { return pageCount; }
    }

    public event ChangePageHandler changePageHandler;
```



```
public delegate void ChangePageHandler(Pager sender);
```

```
protected void ChangePage()
```

```
{
```

```
    if (changePageHandler != null)
```

```
        changePageHandler(this);
```

```
}
```

```
}
```

```
class ButtonPager : Pager, Control
```

```
{
```

```
    public void NextPage()
```

```
    {
```

```
        if (pageIndex < pageCount - 1)
```

```
        {
```

```
            Console.WriteLine("Click NextPage Button...");
```

```
            pageIndex++;
```

```
            ChangePage();
```

```
    }

}

public void PreviousPage()

{

    if (pageIndex > 0)

    {

        Console.WriteLine("Click PreviousPage Button...");

        pageIndex--;

        ChangePage();

    }

}

public void ChangePage(Pager sender)

{

    base.pageIndex = sender.PageIndex;

    if (pageIndex > 0 && pageIndex < pageCount - 1)

        Console.WriteLine("<<Previous Next>>");

}
```

```
elseif(pageIndex == 0)

    Console.WriteLine("Next>>");

else

    Console.WriteLine("<<Previous");

}

}

class ListPager: Pager, Control

{

    public void SelectPage(int pageIndex)

    {

        if (pageIndex >= 0 && pageIndex < pageCount)

        {

            Console.WriteLine(string.Format("Click <{0}> Link...", pageIndex + 1));

            base.pageIndex = pageIndex;

            ChangePage();

        }

    }

}
```

```
public void ChangePage(Pager sender)

{

    base.pageIndex = sender.PageIndex;

    for (int i = 1; i <= pageCount; i++)

    {

        if (pageIndex + 1 == i)

            Console.WriteLine(string.Format("<{0}> ", i));

        else

            Console.WriteLine(string.Format("{0} ", i));

    }

    Console.WriteLine();

}

}
```

```
interface Control
```

```
{

    void ChangePage(Pager sender);

}
```

```
}

class GridView:Control

{

    public void ChangePage(Pager sender)

    {

        Console.WriteLine(string.Format("GridView->Show data of page {0}",
sender.PageIndex + 1));

    }

}

class Label:Control

{

    public void ChangePage(Pager sender)

    {

        Console.WriteLine(string.Format("Label.Text=[{0}/{1}]", sender.PageIndex + 1,
sender.PageCount));

    }

}
```

```
}  
  
}
```

代码执行结果如下图：

```

C:\WINDOWS\system32\cmd.exe
Click NextPage Button...
<<Previous Next>>
GridView->Show data of page 2
Label.Text=[2/3]
1 <2> 3

Click NextPage Button...
<<Previous
GridView->Show data of page 3
Label.Text=[3/3]
1 2 <3>

Click PreviousPage Button...
<<Previous Next>>
GridView->Show data of page 2
Label.Text=[2/3]
1 <2> 3

Click PreviousPage Button...
Next>>
GridView->Show data of page 1
Label.Text=[1/3]
<1> 2 3

Click <3> Link...
<<Previous
GridView->Show data of page 3
Label.Text=[3/3]
1 2 <3>

Click <2> Link...
<<Previous Next>>
GridView->Show data of page 2
Label.Text=[2/3]
1 <2> 3

Click <1> Link...
Next>>
GridView->Show data of page 1
Label.Text=[1/3]
<1> 2 3
请按任意键继续. . .

```

代码说明

- 在这里，我们使用 C# 语言事件机制来实现观察者模式，虽然和 GOF 的“标准”模式不

同，但是还是可以看出观察者模式最基本的几个角色。要知道，GOF 设计模式虽然是经典，但是毕竟是很久以前提出的，可以考虑使用 C#的一些特性来改进。

- Pager 类型是抽象主体角色（或者叫作被观察者、发布方、主动方、目标、主题），传统的抽象主体用于保存观察者。在这里的 ChangePage 方法用于在有变化后触发事件。另外，从 ChangePageHandler 代理中看到，我们把抽象主体作为了参数，这样，观察者就能根据主体的状态作一些调整。
- ButtonPage 是一个具体主体角色。NextPage()方法中首先判断请求的页面是否超过了页面索引，如果没有超过的话，则更新页面索引并且调用了基类的 ChangePage()方法来通知所有的观察者。PreviousPage()方法也是一样的道理。
- Control 接口是一个抽象观察者角色（或者说观察者、订阅方、被动方），它定义了一个统一的接口，如果接受到了事件通知，则调用这个方法进行处理。
- GridView 和 Label 则是具体观察者，可以看到它们不用考虑怎么被通知的事情，只需要考虑被通知后做什么。在这里，GridView 重新绑定了数据，Label 显示了页数信息。
- 这样其实已经组成了一个最基本的观察者模式的结构。获取你也注意到了，ButtonPager 还实现了 Control 接口，说明它还是一个具体的观察者。这并没有什么不可以，它一方面可以在翻页后通知 GridView、Label 等对象，一方面又可以被别人通知。还记得客户需要实现一个 ListPager 的需求吗？在 ListPager 翻页后还需要通知 ButtonPager 来改变状态呢。
- 一样的道理，ListPager 也是一个观察者。它需要观察 ButtonPager 的变动。
- 注意到在 ListPager 和 ButtonPager 的 ChangePage()方法中都更新了页面的索引值，

你或许不理解为什么 Label 和 GridView 不更新呢？其实，这并没有什么奇怪，ButtonPager 翻页后通知 ListPager 更新状态，最需要更新的状态就是页面索引值，用户不是直接点击 ListPager 翻页的，当然需要更新。Label 和 GridView 中并没有实现是因为我们并没有实现具体的一些细节，在实际应用中这些控件保存一些状态也不奇怪。

- 最后来看一看怎么牵线搭桥。我们在 ButtonPager 的改变页面状态事件中注册了四个代理，也就是说它改变状态后需要通知四个观察者。怎么是四个呢？还包括它自己，从逻辑上可能难以理解，其实这是可行的重用代码的方案。对 ButtonPager 来说，是点击哪个控件翻页的并不重要，作为主体它的责任就是通知观察者，作为观察者它的责任就是更新状态或说对事件作出响应。
- 此例完整了一个四个观察者、两个主体的观察者模式。你可能角色一个类型既是观察者又是主体不可理解，其实这在现实生活中非常多的，生物链中的大部分生物既是观察者又是主体，“螳螂捕蝉，黄雀在后”中的螳螂就是。
- 再谈谈耦合和扩展。要再增加一个下拉框分页的分页控件怎么办？无须修改原来的代码，再写一个 DropDownPager（继承 Pager，实现 Control），并且为它的修改分页事件和所有观察者挂钩就可以了。要再增加一个 ListBox 控件针对不同页数显示不同数据怎么办？也无须修改原来的代码，再写一个 ListBox 控件（实现 Control），实现翻页响应的方法，并且订阅所有 Pager 的翻页事件即可。
- 注意，本例仅仅用来演示观察者模式的结构，并没有遵循 .NET 事件模型的最佳实践。

何时采用

通过这个例子，我们就很容易理解观察者模式的适用点了：

- 一个对象的行为引发其它多个对象的行为。前者成为主体，后者称为观察者。
- 为了降低耦合，不希望主体直接调用观察者的方法，而是采用动态订阅主体事件的方式来进行自动的连锁响应行为。
- 为了增加灵活性，希望动态调整订阅主体事件的观察者，或者希望动态调整观察者订阅主体的事件。

实现要点

- 抽象主体角色公开了自身的事件，可以给任意观察者订阅。
- 抽象观察者角色定义了统一的处理行为，在 C#中使用事件-代理模式的话，统一的处理行为并不这么重要，有的时候甚至还会限制灵活性。由于本例的特殊原因，并没有从这个接口中得益。
- 响应方法订阅代理事件的操作可以在观察者中定义也可以在外部分定义，根据自己的需求决定，放在外部定义灵活性更高。
- 具体观察者往往只需要实现响应方法即可。
- 可以有多个主体角色、多个观察者角色交错，也可以一个类型是两个角色，主体也可以提供多个事件。从应用上来说观察者模式变化是非常多的。

注意事项

- 由于这种灵活性，在观察者订阅事件的时候需要考虑是否会出现破坏行为？是否会出现无限循环或死锁等问题？观察者响应的时候是否会影响其它观察者？
- 对于观察者数量很多的时候使用观察者模式并不适合，可能会造成性能问题。
- 在不能采用事件-代理方式完成观察者模式的情况下（比如跨网络应用等）可以考虑采用传统的观察者模式。

无废话 C#设计模式之二十：Mediator

意图

用一个中介对象来封装一系列对象的交互。中介者使得各对象不需要显式相互引用，从而使其松散耦合，而且可以独立地改变它们之间的交互。

场景

我们知道，一个网络游戏往往有很多大区。每一个大区可以是一组服务器，也可以是多组服务器，在这里假设一个大区是一组服务器。为了效率，一般每个大区都会有一个数据库，玩家的创建角色、充值、消费行为只是在这一个大区中有效。现在公司有了新的需求，那就是玩家的一些信息能在多个大区中共享。比如，在注册的时候就把玩家的账户信息写入多个信息共享的大区，玩家在某个大区中充值需要“通知”其它大区修改账户余额，玩家在某个大区中消费也需要“通知”其它大区修改账户

余额。

如果我们现在有 ABC 三个大区，下面的方法可以实现需求：

- 网站的注册方法调用 A、B 和 C 大区的注册方法
- A 大区的充值方法调用 B 和 C 的充值方法
- B 大区的充值方法调用 A 和 C 的充值方法
- C 大区的充值方法调用 A 和 B 的充值方法
- A 大区的消费方法调用 B 和 C 的充值方法
- B 大区的消费方法调用 A 和 C 的充值方法
- C 大区的消费方法调用 A 和 B 的充值方法

我想，没有人会这么做吧。你肯定会想到在一个统一的地方去维护所有大区的信息，任何一个大区的行为不直接和其它大区的行为关联，它们所有的行为都提交到一个统一的地方（假设它是 AccountSystem）去处理。这么做有几个好处：

- 各大区不需要关心还有哪些其它的大区，它只直接和 AccountSystem 对话。
- 只需要调整 AccountSystem 就能调整各大区之间的交互行为，比如我们仅仅希望 A 和 B 大区共享信息、C 和 D 大区共享信息，那么对于这种交互策略的改变也需要修改 AccountSystem。
- 有利于大区的扩充，有了新的大区后，我们不用在大区中考虑它的交互行为，统一交给 AccountSystem 去安排。

现在，再看看引入 AccountSystem 后的通讯：

- 网站调用 AccountSystem 的注册方法 (1)
- AccountSystem 调用 A、B 和 C 大区的注册方法 (2)
- A、B 和 C 大区的充值方法调用 AccountSystem 的充值方法 (3)
- A、B 和 C 大区的消费方法调用 AccountSystem 的充值方法 (4)
- AccountSystem 的充值方法调用 A、B 和 C 大区的专有充值方法 (只针对本大区的充值) (5)
- AccountSystem 的充值方法调用 A、B 和 C 大区的专有消费方法 (只针对本大区的消费) (6)

至此，你已经实现了中介者模式。你可能会觉得，(1) 和 (2) 非常类似门面模式，没错，它确实就是门面模式，而有了 (3) ~ (6) 的行为，AccountSystem 也就是一个中介者的角色了。

示例代码

```
using System;

using System.Collections.Generic;

using System.Text;

namespace MediatorExample
{
```

```
class Program
```

```
{
```

```
    static void Main(string[] args)
```

```
    {
```

```
        AccountSystem accountSystem = new AccountSystem();
```

```
        GameSystem gameArea1 = new GameArea1(accountSystem);
```

```
        GameSystem gameArea2 = new GameArea2(accountSystem);
```

```
        accountSystem.RegisterGameArea(gameArea1);
```

```
        accountSystem.RegisterGameArea(gameArea2);
```

```
        string userName = "aaa";
```

```
        accountSystem.CreateAccount(userName);
```

```
        gameArea1.Recharge(userName, 200);
```

```
        gameArea2.Consume(userName, 50);
```

```
        accountSystem.QueryBalance(userName);
```

```
    }
```

```
}
```

```
class AccountSystem
```

```
{

    private Dictionary<string,int> userBalance = new Dictionary<string,int>();

    private List<GameSystem> gameAreaList = new List<GameSystem>();


    public void RegisterGameArea(GameSystem gs)

    {

        gameAreaList.Add(gs);

    }


    public void CreateAccount(string userName)

    {

        userBalance.Add(userName, 0);

        foreach(GameSystem gs in gameAreaList)

        {

            gs.CreateAccountSelf(userName);

        }

    }


    public void Recharge(string userName, int amount)

    {
```

```
        if(userBalance.ContainsKey(userName))

        {

            bool ok = true;

            foreach(GameSystem gs in gameAreaList)

            {

                ok = gs.RechargeSelf(userName, amount);

                if(ok)

                {

                    userBalance[userName] += amount;

                }

            }

        }

    }

}
```

```
public void Consume(string userName, int amount)

{

    if(userBalance.ContainsKey(userName))

    {

        bool ok = true;

        foreach(GameSystem gs in gameAreaList)

        {

            ok = gs.ConsumeSelf(userName, amount);

            if(ok)
```



```
        userBalance[userName] -= amount;
```

```
    }
```

```
}
```

```
public void QueryBalance(string userName)
```

```
{
```

```
    Console.WriteLine("Your balance is " + userBalance[userName]);
```

```
}
```

```
}
```

```
abstract class GameSystem
```

```
{
```

```
    private AccountSystem accountSystem;
```

```
    protected Dictionary<string, int> userBalance = new Dictionary<string, int>();
```

```
    public GameSystem(AccountSystem accountSystem)
```

```
{
```

```
        this.accountSystem = accountSystem;
```

```
}
```

```
internal virtual bool CreateAccountSelf(string userName)
```

```
{
```

```
    userBalance.Add(userName, 0);
```

```
    return true;
```

```
}
```

```
internal virtual bool RechargeSelf(string userName, int amount)
```

```
{
```

```
    if (userBalance.ContainsKey(userName))
```

```
        userBalance[userName] += amount;
```

```
    return true;
```

```
}
```

```
internal virtual bool ConsumeSelf(string userName, int amount)
```

```
{
```

```
    if (userBalance.ContainsKey(userName))
```

```
        userBalance[userName] -= amount;
```

```
return true;
```

```
}
```

```
public void Recharge(string userName, int amount)
```

```
{
```

```
    accountSystem.Recharge(userName, amount);
```

```
}
```

```
public void Consume(string userName, int amount)
```

```
{
```

```
    accountSystem.Consume(userName, amount);
```

```
}
```

```
}
```

```
class GameArea1 : GameSystem
```

```
{
```

```
    public GameArea1(AccountSystem accountSystem) : base(accountSystem) { }
```

```

internal override bool CreateAccountSelf(string userName)

{

    Console.WriteLine(userName + " Registered in GameArea1");

    return base.CreateAccountSelf(userName);

}


internal override bool RechargeSelf(string userName, int amount)

{

    base.RechargeSelf(userName, amount);

    Console.WriteLine(userName + "'s amount in GameArea1 is " +
userBalance[userName]);

    return true;

}


internal override bool ConsumeSelf(string userName, int amount)

{

    base.ConsumeSelf(userName, amount);

    Console.WriteLine(userName + "'s amount in GameArea1 is " +

```

```
userBalance[userNam];
```

```
        return true;
```

```
    }
```

```
}
```

```
class GameArea2:GameSystem
```

```
{
```

```
    public GameArea2(AccountSystem accountSystem) :base(accountSystem) { }
```

```
    internal override bool CreateAccountSelf(string userName)
```

```
    {
```

```
        Console.WriteLine(userName + " Registered in GameAre2");
```

```
        return base.CreateAccountSelf(userName);
```

```
    }
```

```
    internal override bool RechargeSelf(string userName,int amount)
```

```
    {
```

```

        base.RechargeSelf(userName, amount);

        Console.WriteLine(userName + "'s amount in GameArea2 is " +
userBalance[userName]);

        return true;
    }

    internal override bool ConsumeSelf(string userName, int amount)
    {
        base.ConsumeSelf(userName, amount);

        Console.WriteLine(userName + "'s amount in GameArea2 is " +
userBalance[userName]);

        return true;
    }
}

```

代码执行结果如下图：

```
C:\WINDOWS\system32\cmd.exe
aaa Registered in GameArea1
aaa Registered in GameArea2
aaa's amount in GameArea1 is 200
aaa's amount in GameArea2 is 200
aaa's amount in GameArea1 is 150
aaa's amount in GameArea2 is 250
Your balance is 150
请按任意键继续. . .
```

代码说明

- AccountSystem 是一个中介者角色，它负责各个同事类之间的交互。要使同事对象参与它的管理，就需要在内部维护一个同事对象的列表。
- 我们看到，AccountSystem 的注册、充值和消费方法会遍历相关的同事对象并且调用它们的专有方法进行操作。在全部操作完成之后，它才会更新自己的账户。
- GameSystem 是一个抽象同事。充值和消费方法都有两种。一种是给外部调用的充值和消费方法，一种是给外部调用的，另外一种是为 AccountSystem 调用的。在对外的方法中，GameSystem 仅仅是把这个请求转发给中介者，它自己不做实质性的操作，而在 xxxSelf() 方法中才做真正的充值、消费操作。
- GameArea1 和 GameArea2 是具体同事，调用父类构造方法来和中介者关联。
- 中介者模式的特点就是同事自己意识到它需要和一个中介者关联，而在实际的操作过程中，它们只是负责和中介者通讯并且接受中介者的请求，而不再和其它同事发生直接的关联。

何时采用

如果一组接口相对稳定（如果 GameArea1和 GameArea2的充值方法定义不一样，那么 AccountSystem 就有点晕眩了）的对象之间的依赖关系错综复杂，依赖关系难以维护，或者会发生变动可以考虑引入中介者模式。

实现要点

- 在 C#中可以适用 delegate 关联中介者和各同事之间的交互行为,这样各同事就不需要直接和中介者进行耦合。
- 中介者模式和观察者模式的区别是，前者应用于多对多杂乱交互行为的统筹处理，后者应用于一（多）对多关系的灵活定制。对于本例来说，集中处理后还需要分散处理，那么后半阶段的处理过程可以应用观察者模式。对于前一节的例子来说，如果有多个主体角色和多个观察者进行多对多通讯的话，也可以应用中介者模式来统筹这个多对多的过程（大家可以自己尝试修改前一节的实例来应用中介者模式）。
- 中介者模式和门面模式的区别是，前者的各同事类需要依靠中介者进行双向通讯，应用于子系统之间，而后的子系统往往不会通过门面去和调用方进行通讯，趋向于单向通讯，应用于子系列和更高层次的系统。本例中就有门面模式和中介者模式的影子。
- 中介者模式往往可以在构架的层次进行应用，有的时候和观察者模式以及门面模式一起使用，有的时候又会向观察者模式和门面模式退化。其实在应用模式的过程中不必过多考虑模式的准确定位，如果我们确实从中得以，那么这个名字就不重要了。

注意事项

- 不是所有的系统都需要应用中介者模式把多对多的关系转化为多对一对多的。如果各个同事之间本来的关联就很清晰（没有交错关联），或这种关联并不复杂，没有必要应用中介者。
- 在实际的应用过程中，中介者做的控制并不会向本例那样简单，它可能包含很多的处理逻辑。如果还伴随着需求的变化，中介者角色可能会越来越难维护，此时可以考虑对中介者角色或处理行为应用其它的一些设计模式。

无废话 C#设计模式之二十一：Visitor

意图

实现通过统一的接口访问不同类型元素的操作，并且通过这个接口可以增加新的操作而不改变元素的类。

场景

想不出什么好例子，我们在组合模式的那个例子上进行修改吧。我们知道，无论是游戏大区、游戏服务器还是游戏的服务都是一个元素，只不过它们的层次不一样。对于这样的层次结构，我们使用

了组合模式来统一各层的接口，这样对游戏大区的操作和对游戏服务器的操作对调用方来说没有什么两样。在现实中，组合模式的运用往往没有这么顺利：

- 如果元素需要增加新的操作，那么势必需要在抽象元素中增加接口，这个时候的改动就非常大了，几乎每一个具体元素都需要修改。
- 如果元素并没有统一的接口，并且树枝角色中有多种树叶角色，那么树枝角色势必需要根据树叶类型来调用不同的方法。
- 如果树叶角色的接口经常发生变动，那么一旦发生变动操作树叶的树枝角色也需要发生修改。

访问者模式可以解决这些问题。

示例代码

```
using System;

using System.Collections;

using System.Collections.Generic;

using System.Text;

namespace VisitorExample
{
```

```
class Program
```

```
{
```

```
    static void Main(string[] args)
```

```
    {
```

```
        Element server1 = new GameServer("GS1", "192.168.0.1");
```

```
        server1.Add(new GameService("Lobby1", 1, "S5Lobby1", 100));
```

```
        server1.Add(new GameService("Lobby2", 1, "S5Lobby2", 200));
```

```
        server1.Add(new GameService("Gate1", 2, "S5Gate1"));
```

```
        server1.Add(new GameService("DataExchange1", 3, "S5DataExchange1"));
```

```
        server1.Add(new GameService("Rank1", 4, "S5Rank1"));
```

```
        server1.Add(new GameService("Log1", 5, "S5Log1"));
```

```
        Element server2 = new GameServer("GS2", "192.168.0.2");
```

```
        server2.Add(new GameService("Lobby3", 1, "S5Lobby3", 150));
```

```
        server2.Add(new GameService("Lobby4", 1, "S5Lobby4", 250));
```

```
        server2.Add(new GameService("Gate2", 2, "S5Gate2"));
```

```
        server2.Add(new GameService("DataExchange2", 3, "S5DataExchange1"));
```

```
        server2.Add(new GameService("Rank2", 4, "S5Rank2"));
```

```
        server2.Add(new GameService("Log2", 5, "S5Log2"));
```

```
Elementarea =newGameArea("电信区");

area.Add(server1);

area.Add(server2);

area.Accept(newStartGameVisitor());//A1

area.Accept(newStopGameVisitor());//B1

server1.Accept(newQueryPlayerCountVisitor());

server2.Accept(newQueryPlayerCountVisitor());

area.Accept(newQueryPlayerCountVisitor());

}

}
```

```
interfaceIVisitor{ }
```

```
interfaceIGameServiceVisitor
```

```
{
```

```
voidVisit(GameServicegameService);
```

```
}
```

```
interface IGameServerVisitor
```

```
{
```

```
    void Visit(GameServer gameServer);
```

```
}
```

```
interface IGameAreaVisitor
```

```
{
```

```
    void Visit(GameArea gameArea);
```

```
}
```

```
class StartGameVisitor: IVisitor, IGameServiceVisitor, IGameServerVisitor, IGameAreaVisitor
```

```
{
```

```
    public void Visit(GameService gameService)
```

```
    {
```

```
        //A9
```

```
        gameService.StartGameService(this);
```

```
    }
```

```
public void Visit(GameServer gameServer)
```

```
{
```

```
    //A6
```

```
    gameServer.StartGameServer(this);
```

```
}
```

```
public void Visit(GameArea gameArea)
```

```
{
```

```
    //A3
```

```
    gameArea.StartGameArea(this);
```

```
}
```

```
}
```

```
class StopGameVisitor:IVisitor,IGameServiceVisitor,IGameServerVisitor,IGameAreaVisitor
```

```
{
```

```
    public void Visit(GameService gameService)
```

```
    {
```

```
        //B7
```

```

        Console.WriteLine(string.Format("{0} stopped", gameService.Name));

    }

    public void Visit(GameServer gameServer)

    {

        //B5

        Console.WriteLine("=====Stopping the whole " + gameServer.Name
+ "=====");

        for(int i = gameServer.ServiceList.Count - 1; i >= 0; i--)

        {

            gameServer.ServiceList[i].Accept(this);

        }

        Console.WriteLine("=====The whole " + gameServer.Name + "
stopped=====");

    }

    public void Visit(GameArea gameArea)

    {

```

```

//B3

Console.WriteLine("=====Stopping the whole " + gameArea.Name
+"=====");

foreach(GameServerelement ingameArea.ServerList)

{

    element.Accept(this);

}

Console.WriteLine("=====The whole " + gameArea.Name + "
stopped=====");

}

}

classQueryPlayerCountVisitor:IVisitor,IGameServerVisitor,IGameAreaVisitor

{

    publicvoidVisit(GameServergameServer)

    {

        intplayerCount = 0;

        foreach(GameServicegameService ingameServer)

```



```

    {

        if(gameService.ServiceType == 1)

            playerCount += gameService.PlayerCount;

    }

    Console.WriteLine("=====Player Count on " + gameServer.Name + "
is:" + playerCount);

}

public void Visit(GameArea gameArea)

{

    int playerCount = 0;

    foreach(GameServer gameServer in gameArea)

    {

        foreach(GameService gameService in gameServer)

        {

            if(gameService.ServiceType == 1)

                playerCount += gameService.PlayerCount;

        }

    }

}

```

```

    }

    Console.WriteLine("=====Player Count on " + gameArea.Name + " is:"
+ playerCount);

    }

}

abstract class Element
{

    protected string name;

    public string Name

    {

        get { return name; }

    }

    public Element(string name)

    {

        this.name = name;

```

```
}
```

```
public abstract void Add(Element element);
```

```
public abstract void Remove(Element element);
```

```
public abstract void Accept(IVisitor visitor);
```

```
}
```

```
class GameService: Element, IComparable<GameService>
```

```
{
```

```
    private int serviceType;
```

```
    public int ServiceType
```

```
    {
```

```
        get { return serviceType; }
```

```
        set { serviceType = value; }
```

```
    }
```

```
    private string serviceName;
```

```
private int playerCount;
```

```
public int PlayerCount
```

```
{
```

```
    get{ return playerCount; }
```

```
    set{ playerCount = value; }
```

```
}
```

```
public GameService(string name, int serviceType, string serviceName)
```

```
    : base(name)
```

```
{
```

```
    this.serviceName = serviceName;
```

```
    this.serviceType = serviceType;
```

```
}
```

```
public GameService(string name, int serviceType, string serviceName, int playerCount)
```

```
    : base(name)
```

```
{
```

```
this.serviceName = serviceName;

this.serviceType = serviceType;

this.playerCount = playerCount;

}

public override void Add(Element element)

{

    throw new ApplicationException("xxx");

}

public override void Remove(Element element)

{

    throw new ApplicationException("xxx");

}

public override void Accept(IVisitor visitor)

{

    //A8,B6
```

```

        IGameServiceVisitor gameServiceVisitor = visitor as IGameServiceVisitor;

        if (gameServiceVisitor != null) gameServiceVisitor.Visit(this);

    }

    public int CompareTo(GameService other)

    {

        return other.serviceType.CompareTo(serviceType);

    }

    public void StartGameService(IVisitor visitor)

    {

        //A10

        Console.WriteLine(string.Format("{0} started", name));

    }

}

class GameServer:Element

{

```

```
private string serverIP;
```

```
private List<GameService> serviceList = new List<GameService>();
```

```
public List<GameService> ServiceList
```

```
{
```

```
    get { return serviceList; }
```

```
}
```

```
public GameServer(string name, string serverIP)
```

```
    : base(name)
```

```
{
```

```
    this.serverIP = serverIP;
```

```
}
```

```
public override void Add(Element element)
```

```
{
```

```
    serviceList.Add((GameService)element);
```

```
}
```

```
public override void Remove(Element element)
```

```
{
```

```
    serviceList.Remove((GameService)element);
```

```
}
```

```
public override void Accept(IVisitor visitor)
```

```
{
```

```
    //A5,B5
```

```
    IGameServerVisitor gameServerVisitor = visitor as IGameServerVisitor;
```

```
    if (gameServerVisitor != null) gameServerVisitor.Visit(this);
```

```
}
```

```
public IEnumerator<GameService> GetEnumerator()
```

```
{
```

```
    foreach (GameService gameService in serviceList)
```

```
        yield return gameService;
```



```

    }

    public void StartGameServer(IVisitor visitor)

    {

        //A7

        IGameServerVisitor gameServerVisitor = visitor as IGameServerVisitor;

        if (gameServerVisitor != null)

        {

            serviceList.Sort();

            Console.WriteLine("=====Starting the whole " + name
+ "=====");

            foreach (Element gameService in serviceList)

            {

                gameService.Accept(visitor);

            }

            Console.WriteLine("=====The whole " + name +
started=====");

        }

```

```
}
```

```
}
```

```
class GameArea:Element
```

```
{
```

```
    private List<GameServer> serverList = new List<GameServer>();
```

```
    public List<GameServer> ServerList
```

```
    {
```

```
        get { return serverList; }
```

```
    }
```

```
    public GameArea(string name)
```

```
        :base(name) { }
```

```
    public override void Add(Element element)
```

```
    {
```

```
        serverList.Add((GameServer)element);
```

```
}
```

```
public override void Remove(Element element)
```

```
{
```

```
    serverList.Remove((GameServer)element);
```

```
}
```

```
public override void Accept(IVisitor visitor)
```

```
{
```

```
    //A2,B2
```

```
    IGameAreaVisitor gameAreaVisitor = visitor as IGameAreaVisitor;
```

```
    if(gameAreaVisitor != null) gameAreaVisitor.Visit(this);
```

```
}
```

```
public IEnumerator<GameServer> GetEnumerator()
```

```
{
```

```
    foreach(GameServer gameServer in serverList)
```

```
        yield return gameServer;
```

```

    }

    public void StartGameArea(IVisitor visitor)
    {

        //A4

        IGameAreaVisitor gameAreaVisitor = visitor as IGameAreaVisitor;

        if (gameAreaVisitor != null)
        {

            Console.WriteLine("=====Starting the whole " + name
+ "=====");

            foreach (Element gameServer in serverList)
            {

                gameServer.Accept(visitor);

            }

            Console.WriteLine("=====The whole " + name +
started=====");

        }

    }
}

```

```
}  
  
}
```

代码执行结果如下图：



```
C:\WINDOWS\system32\cmd.exe  
=====Starting the whole 电信1区=====  
=====Starting the whole GS1=====  
Log1 started  
Rank1 started  
DataExchange1 started  
Gate1 started  
Lobby1 started  
Lobby2 started  
=====The whole GS1 started=====  
=====Starting the whole GS2=====  
Log2 started  
Rank2 started  
DataExchange2 started  
Gate2 started  
Lobby3 started  
Lobby4 started  
=====The whole GS2 started=====  
=====The whole 电信1区 started=====  
=====Stopping the whole 电信1区=====  
=====Stopping the whole GS1=====  
Lobby2 stopped  
Lobby1 stopped  
Gate1 stopped  
DataExchange1 stopped  
Rank1 stopped  
Log1 stopped  
=====The whole GS1 stopped=====  
=====Stopping the whole GS2=====  
Lobby4 stopped  
Lobby3 stopped  
Gate2 stopped  
DataExchange2 stopped  
Rank2 stopped  
Log2 stopped  
=====The whole GS2 stopped=====  
=====The whole 电信1区 stopped=====  
=====Player Count on GS1 is:300  
=====Player Count on GS2 is:400  
=====Player Count on 电信1区 is:700  
请按任意键继续. . .
```

代码说明

代码从组合模式的例子修改过来，有一点乱，一点一点来分析：

- IVisitor 是一个空接口，目的是为了抽象所有的访问者，抽象层次相当于 Element。
- IGameServerVisitor、IGameServiceVisitor 以及 IGameAreaVisitor 定义了访问者的访问操作（操作接口）。在这里，我们并没有把它们合并为一个接口，因为访问者可能仅针对一部分元素，或一个层次的元素，不一定都针对所有元素。
- StartGameVisitor、StopGameVisitor 以及 QueryPlayCountVisitor 是具体的访问者。它们根据自己的需求实现不同的操作接口，虽然都是访问者但是它们的目的不太一样，见下。
- Element 类型就是抽象构件（抽象元素），它给组合对象以及单个对象提供了一个一致的接口，使得它们都能有一致的行为。唯一和组合模式不同的是，在这里定义了一个接受访问者的接口。
- GameService、GameServer 以及 GameArea 都是具体的元素。从组合角度来看，GameServer 和 GameArea 是树枝，GameService 是树叶。看一下 Accept()方法，对于具体元素来说，它应该明确接受特定层次的访问者，如果类型转换正确的话，那么调用访问者的访问方法。
- 这个例子中的 StartGameVisitor 的目的是统一元素的接口。注意到 GameService、GameServer 以及 GameArea 中开启服务的方法都不同（并没有实现统一的接口），这样的话，高层的元素需要直接耦合低层元素的某个方法。通过访问者，我们使得它们直接和访问者的统一方法耦合，由访问者再适配不同的方法。可以从代码中以 A 打头的数字看出开启服务的整个流程。

- 这个例子中的 StopGameVisitor 体现了访问者模式最主要的作用，那就是为元素增加新的操作。这得益于访问者统一 Accept 接口以及双重分派的机制。可以从代码中以 B 打头的数字看出关闭服务的整个流程。
- 由于本例中把 Visitor 的接口按照功能分成了小接口，并且还有一个抽象顶层的空接口。这样，我们就可以为具体访问者实现需要的接口，并且为层次中增加元素也变得不是那么困难。从 QueryPlayerCountVisitor 中可以看到，具体访问者可以某些元素的新操作，而无需实现所有元素的新操作。
- 访问者模式还有一个应用是使不同类型元素组成的集合的遍历访问变得简单、符合开闭原则。由于 GameServer 的子元素一般只有 GameService 而不会有 GameService 和 GameAgent，所以本例没有体现这样的应用。
- 访问者模式的结构比较复杂，变化也比较多。一般不管怎么样变化，主要还是依靠访问者模式双重分派和统一的 Accept 接口来实现。访问者模式的效果有的时候也类似适配器、装饰模式等，由于往往用于操作集合所以一般也通常会和迭代器、组合模式一起使用。

何时采用

访问者模式适用下面的情况：

- 对象结构中包含很多类型，这些类型没有统一的接口，而我们又希望使得对象的操作进行统一。
- 希望为对象结构中的类型新增操作，并且不希望改变原有的代码。

- 对象结构中的一些类型之间发生耦合，而它们的实现又经常会发生变动。
- 针对结构中同一层次的不同类型甚至是不同层次的类型进行迭代。

从这里可以看出，访问者模式的主要适用还是针对对象结构（往往有层次关系），并且需要在设计的时候实现为各类型预留接受访问者的接口。

实现要点

- 每个元素都需要设置 `Accept()` 方法来接受访问者。
- 两次多态分发，确定访问者以及访问者中的方法。
- 如果一个结构层次中有多个类型的元素，那么可以通过一个 `ObjectStructure` 的角色进行封装。

注意事项

- 访问者模式一个主要的缺点就是难以扩展对象结构，其实，这点是可以通过一些变化进行化解的。
- 访问者模式第二个缺点是需要过多暴露对象的内部元素，否则访问者难以对对象进行实质的操作。
- 第三个缺点是需要实现考虑到这样的需求并且提前设置接受访问者的方法。

无废话 C#设计模式之二十二：总结(针对 GOF23)

比较

设计模式	常用程度	适用层次	引入时机	结构复杂度
Abstract Factory	比较常用	应用级	设计时	比较复杂
Builder	一般	代码级	编码时	一般
Factory Method	很常用	代码级	编码时	简单
Prototype	不太常用	应用级	编码时、重构时	比较简单
Singleton	很常用	代码级、应用级	设计时、编码时	简单
Adapter	一般	代码级	重构时	一般
Bridge	一般	代码级	设计时、编码时	一般
Composite	比较常用	代码级	编码时、重构时	比较复杂
Decorator	一般	代码级	重构时	比较复杂
Facade	很常用	应用级、构架级	设计时、编码时	简单
Flyweight	不太常用	代码级、应用级	设计时	一般
Proxy	比较常用	应用级、构架级	设计时、编码时	简单
Chain of Resp.	不太常用	应用级、构架级	设计时、编码时	比较复杂

Command	比较常用	应用级	设计时、编码时	比较简单
Interpreter	不太常用	应用级	设计时	比较复杂
Iterator	一般	代码级、应用级	编码时、重构时	比较简单
Mediator	一般	应用级、构架级	编码时、重构时	一般
Memento	一般	代码级	编码时	比较简单
Observer	比较常用	应用级、构架级	设计时、编码时	比较简单
State	一般	应用级	设计时、编码时	一般
Strategy	比较常用	应用级	设计时	一般
Template Method	很常用	代码级	编码时、重构时	简单
Visitor	一般	应用级	设计时	比较复杂

注：常用程度、适用层次、使用时机等基于自己的理解，结构复杂度基于 C#语言，表格中所有内容仅供参考。

原则、变化与实现

设计模式	变化	实现	体现的原则
------	----	----	-------

Abstract Factory	产品家族的扩展	封装产品族系列内容的创建	开闭原则
Builder	对象组建的变化	封装对象的组建过程	开闭原则
Factory Method	子类的实例化	对象的创建工作延迟到子类	开闭原则
Prototype	实例化的类	封装对原型的拷贝	依赖倒置原则
Singleton	唯一实例	封装对象产生的个数	
Adapter	对象接口的变化	接口的转换	
Bridge	对象的多维度变化	分离接口以及实现	开闭原则
Composite	复杂对象接口的统一	统一复杂对象的接口	里氏代换原则
Decorator	对象的组合职责	在稳定接口上扩展	开闭原则
Facade	子系统的高层接口	封装子系统	开闭原则
Flyweight	系统开销的优化	封装对象的获取	
Proxy	对象访问的变化	封装对象的访问过程	里氏代换原则
Chain of Resp.	对象的请求过程	封装对象的责任范围	
Command	请求的变化	封装行为对对象	开闭原则
Interpreter	领域问题的变化	封装特定领域的变化	

Iterator	对象内部集合的变化	封装对象内部集合的使用	单一职责原则
Mediator	对象交互的变化	封装对象间的交互	开闭原则
Memento	状态的辅助保存	封装对象状态的变化	接口隔离原则
Observer	通讯对象的变化	封装对象通知	开闭原则
State	对象状态的变化	封装与状态相关的行为	单一职责原则
Strategy	算法的变化	封装算法	里氏代换原则
Template Method	算法子步骤的变化	封装算法结构	依赖倒置原则
Visitor	对象操作变化	封装对象操作变化	开闭原则

学习

- 掌握设计模式的意图以及解决的问题
- 掌握设计模式所封装的变化点以及优缺点
- 了解设计模式的结构图以及各角色的职责
- 项目中是否应用了设计模式不重要，重要的是设计模式是否正确应用

- 项目中应用的设计模式和 GOF 设计模式的结构是否一致不重要，重要的是是否从这个结构中得意
- 不管用了还是没有用设计模式，如果违背了原则，就是不恰当的设计
- 没有设计模式是万能的，沉迷于获得一个解决方案的话可能会导致项目结构复杂、代码可读性差、并且造成项目延期

结束语

- 常用的 GOF 23种设计模式介绍完了，这才是起点。
- 本系列文章并没有结束，关注之后非 GOF 23种设计模式的相关文章。
- 如果适当运用 C# 2.0一些有用的特性(特别是代理、泛型以及分部类和设计模式关联比较大)的话，传统的设计模式有非常大的改进的余地。在实际运用的过程中，优先考虑适用语言特性，如果不行再去考虑适用设计模式。
- 迭代器模式(在 C# 2.0中实现非常简单) 解释器模式(应用面非常小，自己也没有整明白)以及备忘录模式(比较简单，没有什么可说的)没有单独立文介绍，但在代码包中包含了相应的例子，所有代码点击[这里](#)下载。