# 系统架构师-基础到企业应用架构-服务层

**----------By CallHot**

## 一、上章回顾

上篇我们主要讲解了系统架构中的四种架构模式，并且分析了四种架构模式的实现及应用场景，那么先来回顾下架构中的业务逻辑层的使用及总结。


如果大家对图中讲述的内容不明白或者说是不深入那么可以参考上篇讲

解的内容：系统架构师-基础到企业应用架构-业务逻辑层。

## 二、摘要

本文将已架构的方式去分析分层结构中的服务层的设计，如何设计出来满足我们说的业务需求及设计规范的服务层将是我们的目标，可能我想大家在项目架构的

过程中可能有些同仁，没有用到该层，或者说是采用的是常用的分层结构的设计，而没有把服务层单独的抽出来，当然我们必须首先知道服务层是干什么用的？为什么

要单独写一个服务层呢？还有就是设计服务层我们从哪些方面入手呢？及怎么判定一个服务层设计的好坏呢？这些都是本章要讲解的具体内容，当然本文中的内容都是

本人平时在项目中的一些经验，可能在一些有丰富经验设计的大牛面前，我讲解的都是皮毛，但是我抱着能给初学者指引和为已知者温习的目的而写，错误之处再所难

免，请大家提出宝贵意见和建议。本文讲述以下内容：


下面我们将针对上面的问题分别进行讲述。

## 三、本章大纲

1、上章回顾。

2、摘要。

3、本章大纲。

4、服务层的介绍。

5、服务层实战。

6、本章总结。

7、系列进度。

8、下篇预告。

## 四、服务层的介绍

本节中将会对服务层的设计进行详细的分析。我们知道我们现在在软件开发的过程中，通常我们会将一些业务逻辑的代码写在表现层，当然这样的方式不是不允

许，当然一般情况下来说我们感觉没什么，但是采用这样的方式，那么表现层与业务逻辑层之间的关系是耦合性的，可能我们是属于那种希望设计简洁或者说对设计规

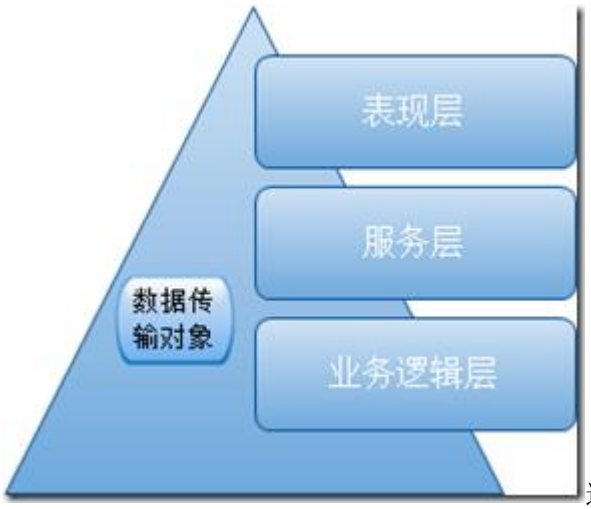
范严格要求的时候，那么我们就可以考虑将业务逻辑的组织通过服务层来实现，那么服务层的作用就是将表现层与业务逻辑层之间完成解耦。那么表现层中就不会出现

任何的业务代码，当然这样带来的好处也是显而易见的，就是当我们修改业务层代码时，我们不需要修改表现层的代码，当然如果服务层设计的不好，那么可能会造成

反效果。

服务层是干什么的？ 通过上面的简单介绍，我想大家都对服务层有了个简单的认识，那么下面我们还是通过图形的方式来看看服务层的位置及作用吧，可能那样

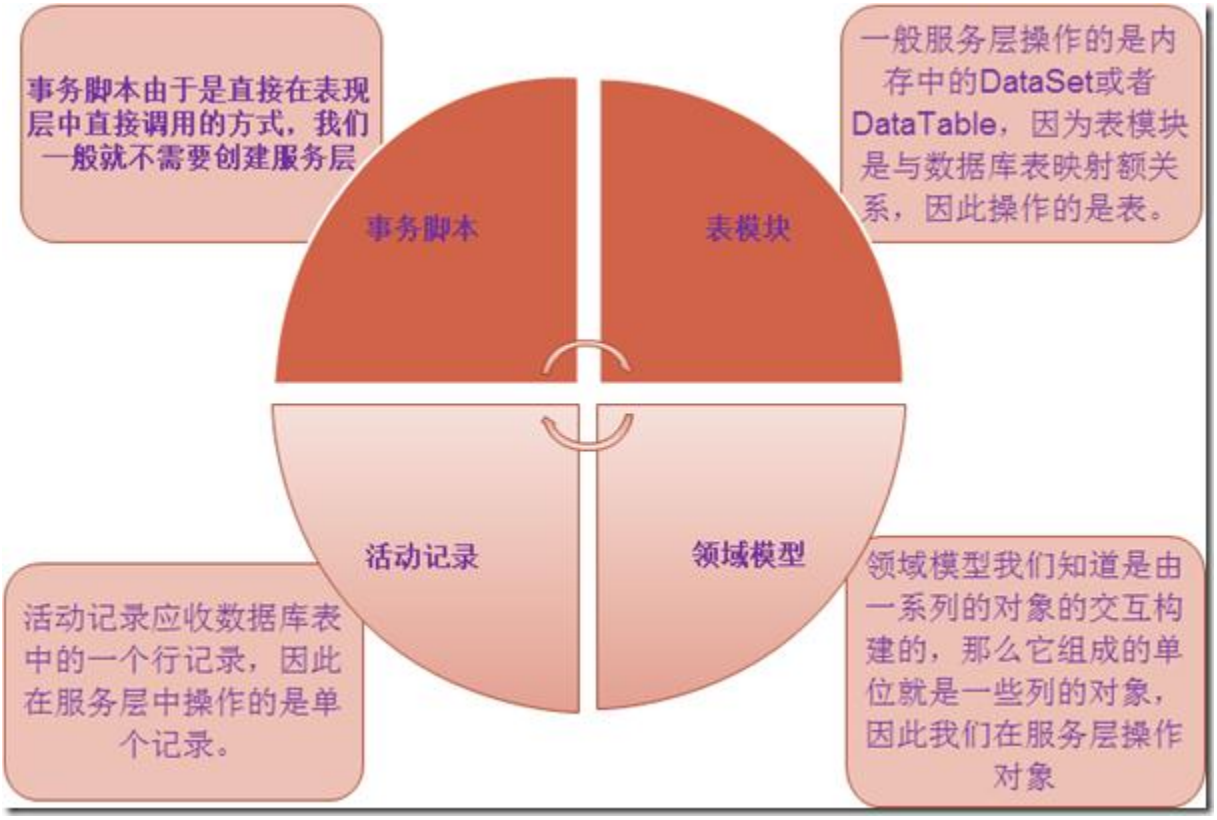
更容易理解。




这几层之间都是通过数据传输对象来完成各层之间的数据通信。通过上图我们知道，服务层是通过数据传输对象

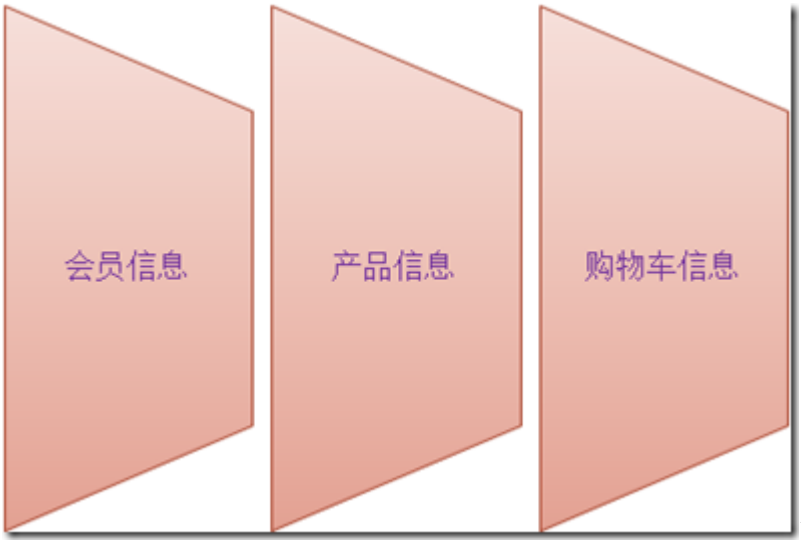
与业务逻辑层直接进行交互，那么业务逻辑层与服务层具体交互的方式及内容是什么呢？

下面我们来看看，业务逻辑层中的四种模式在服务层中的表现。




下面我们来举例说明服务层的作用。通过表现层与业务逻辑的解耦来说明服务层的作用。我们还是以 B2C 中的购物流程来说。

我们先以购物流程中的添加产品到购物车来说




可以简单的看作下面几个对象之间的交互，首先我们先要选择产品，然后将产品添加到购物车

中，然后当然这个购物车是某个会员登陆以后的购物清单，一个购物车中可能有多个产品。我们来看看代码可能如下：

我们定义的产品信息如下：

```
/// <summary>
/// 产品信息
/// </summary>
public class Product
{
    /// <summary>
    /// 返回所有的产品信息
    /// </summary>
    /// <returns></returns>
    public List<Product> GetAll()
    {
        return new List<Product>();
    }
    /// <summary>
    /// 返回产品信息根据产品 ID
    /// </summary>
    /// <returns></returns>
    public Product GetByID(int ID)
    {
        return new Product();
    }
}
```

产品信息中包含 2 个方法，一个是获取所有产品的列表，还有一个是获取实体的信息根据主键。我们来看看购物车的代码：

```
/// <summary>
/// 购物车
/// </summary>
```

```
public class ShopCar
{
    /// <summary>
    /// 购物车中的产品
    /// </summary>
    Dictionary<int, Product> products = new Dictionary<int,
Product>();
    /// <summary>
    /// 将指定产品 ID 的产品添加到购物车
    /// </summary>
    public bool Add(int ID)
    {
        Product product = new Product();
        product= product.GetByID(ID);
        if (products.ContainsKey(ID))
            return false;
        products.Add(ID,product);
        return true;
    }
}
```

下面我们来看看前台调用的代码：

```
public class ShopCar
{
    /// <summary>
    /// 将指定产品 ID 的产品添加到购物车
    /// </summary>
    public bool Add(int ID)
    {
```

```
            ShopCar cart = new ShopCar();
            return cart.Add(ID);
        }
    }
```

上面的代码引用了 ShopCar 对象，说明 UI 层的 ShopCar 与业务层的 ShopCar 有依赖关系，那么我们如何解耦呢？通过引入第三方类，来实现依赖的解除。具体的代码如下：

```
    public class ShopCar
    {
        /// <summary>
        /// 将指定产品 ID 的产品添加到购物车
        /// </summary>
        public bool Add(int ID)
        {
            IShopCar car;
            CarFactory factory = new CarFactory();
            car = factory.ShopCarFactory();
            car.Add(ID);
            return true;
        }
    }
```

修改后通过服务层中的购物车工厂构造出购物车实例，这样就可以把业务逻辑层与界面层之间的耦合关系通过新增加一个服务层来实现解耦，那么表现层的代码更简介，也符合设计规范。

其实我们这里的服务只是讲述了服务层的简单应用场景，其实具体的服务层在使用的过程中远比我们前面讲解的复杂，首先可能服务层还会与数据访问层直接交互，比如说操作数据库，持久化等，服务层主要是组织业务逻辑中的业务逻辑组件，服务层还通过 ORM 框架中提供的数据库访问服务完成相应的操作。我们前面讲过，

一般情况下，表现层与服务层交互是，都是通过数据传输对象来进行通信的，那么显然有时候我们需要在服务层做处理，将数据传输对象转换成领域模型，当然有时候

我们在设计系统架构时，如果系统中的领域模型较多，或者说是拆分后的数据传输对象太多时，我们只要将一个领域模型对应一个数据传输对象就好，只不过是把领域

模型中的行为省略而已。

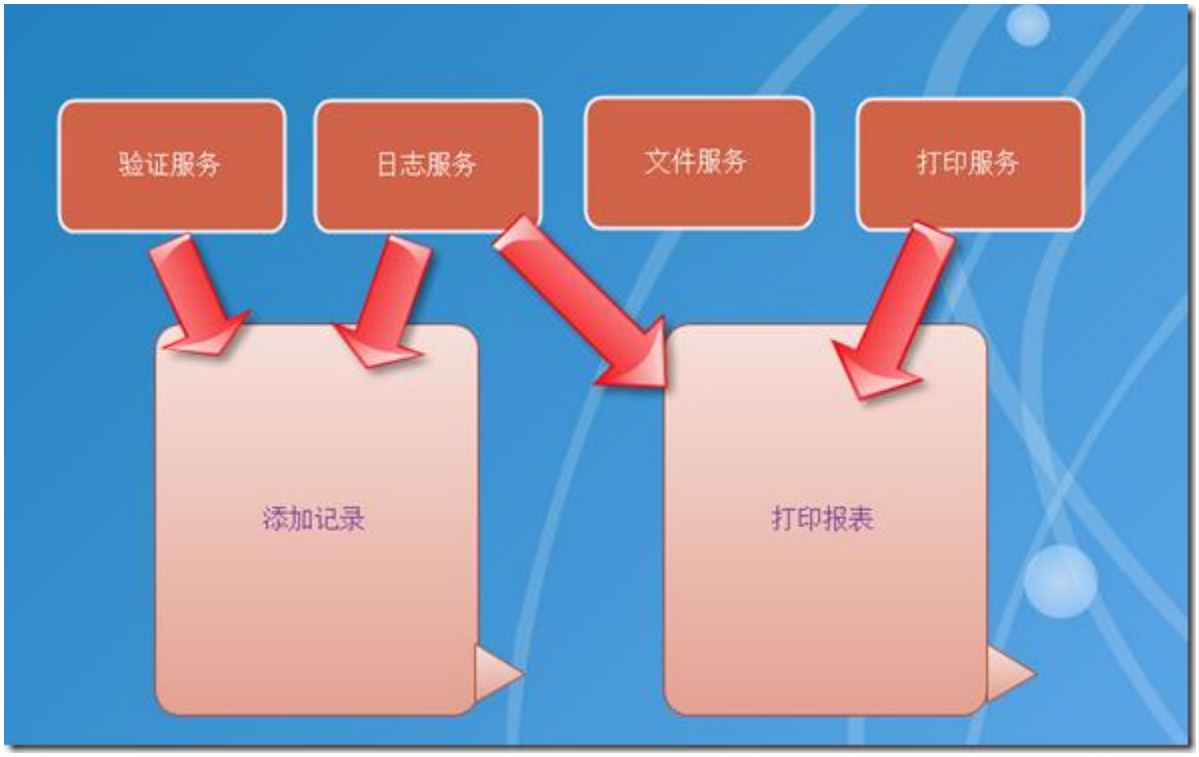
我们来看看目前比较流行的关于服务层的认识：

我想目前最流行的关于服务层的架构就是 SOA（面向服务架构），可以说是 SOA 的出现才让服务层流行起来，SOA 中对服务的理解是这样的，服务层是提供一系

列的服务，而具体的业务流程是通过一系列的服务组成的，把服务看作不是面向某种特定的技术，而是业务流程的组织方式。达到的目的是，提供了一系列的服务，只

需要配置组织服务的流程，就可以不管什么样的技术，都能满足要求的业务流程。当然这是理想化的形式，具体的实行起来还是有相当大的难度。

其实我们可以这样想象，服务就是一个提供了 API 的类，通过封装，外界访问服务时只能通过服务提供的接口来访问。




例如我们通过服务层提供上述的四种服务，然后在表现层中通过

服务层中的服务的调用来完成相应的功能,比如我们在表现层中有新纪录添加时，我们通过服务层的添加记录的方法来完成，服务层通过提供的服务直接完成相应的准备

工作，并且这些服务在定义时都是通过接口的方式来向外提供功能，因此只要满足接口契约就可以完成组件的替换工作，而不会影响系统的运行。

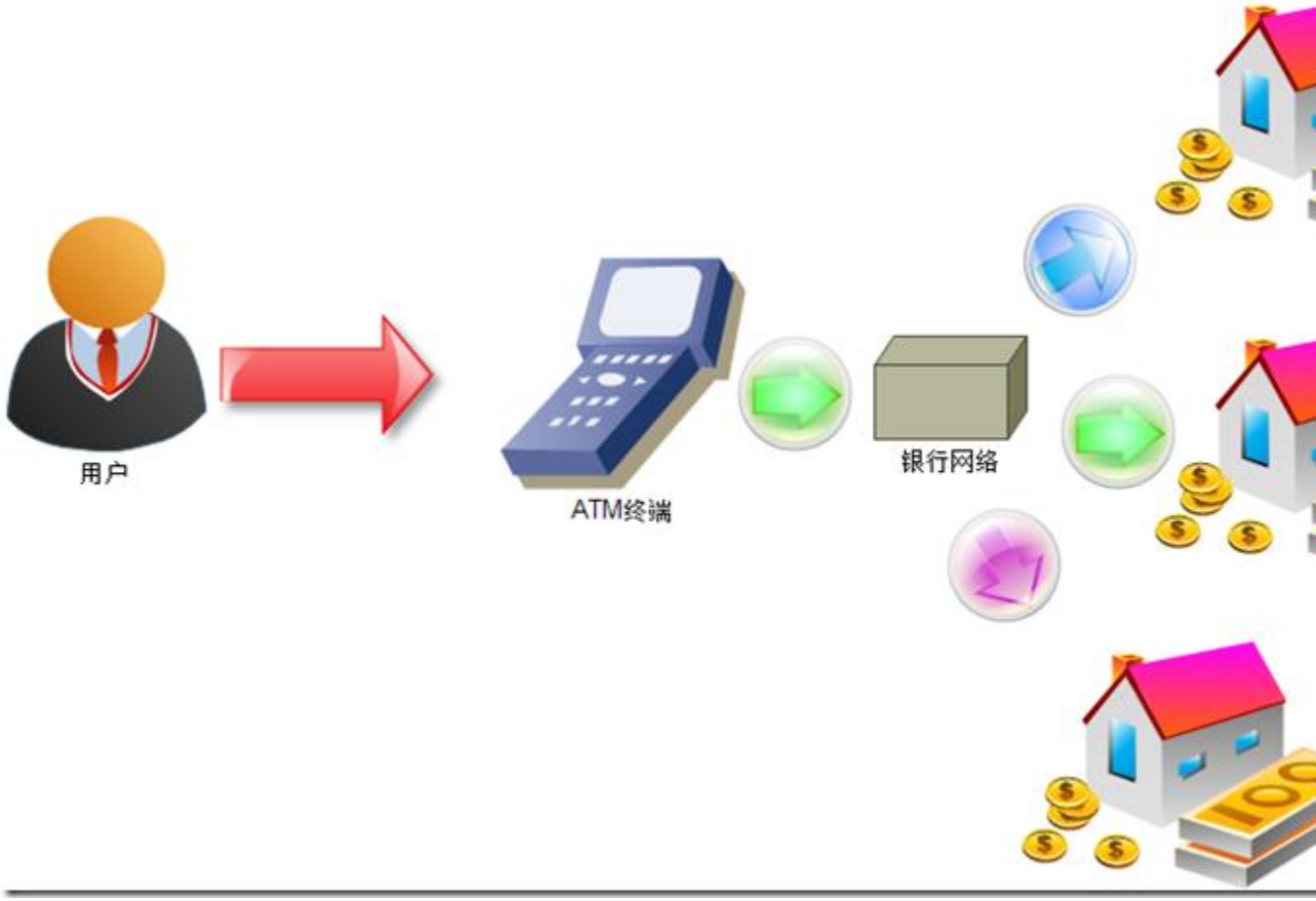
我们有的时候有这样的需求，我们的软件程序要求既有 B/S 的形式直接通过浏览器来完成应用，有时候还需要 C/S 客户端的形式访问系统的功能，这时候我们如果

通过服务来组织业务逻辑那么我们只需要写一个服务层就可以完成远程服务访问，而不用 B/S 下写一次业务逻辑调用，然后 C/S 下再写一次，而且这样一旦修改了相应的

业务逻辑，那么我们需要变动的代价很大。我们来看看服务层给我们提供了什么。


通过服务层我们可以不关心业务逻辑的实现，我们在用户图形

化界面中只需要访问相关的服务即可，举个简单例子，就行银行的银联，跨行取款的行为。下面可以简单的描述了用户取款的服务。




上面简单的描述了，用户的取款服

务，当然只要是银联的银行卡，都可以享受到跨行的异地取款，当然不管什么卡，提供给用户的服务都是相同的。当然我们这里说的是 C/S 客户端服务这样的要求，当

然如果说是 B/S 架构的形式，那么可能我们不用单独抽出这样的远程服务的形式。而服务层可能就是表现层的一部分，这时候我们建议不要把服务层设计成 web 服务，这

时候我们设计服务层时更关心服务层的抽象，而不是实现方式。

一般情况下来说服务层的设计实现，可能有时候和部署时的要求有关，例如有时候我们需要将服务部署在应用服务器上，这时候我们就必须考虑服务层必须发布成

远程调用服务或者 Web 服务，当然具体的远程调用服务可以有几种方式，我们主要看基于什么通信方式，是 remoting 还是 soap，还是 socket 通信等。

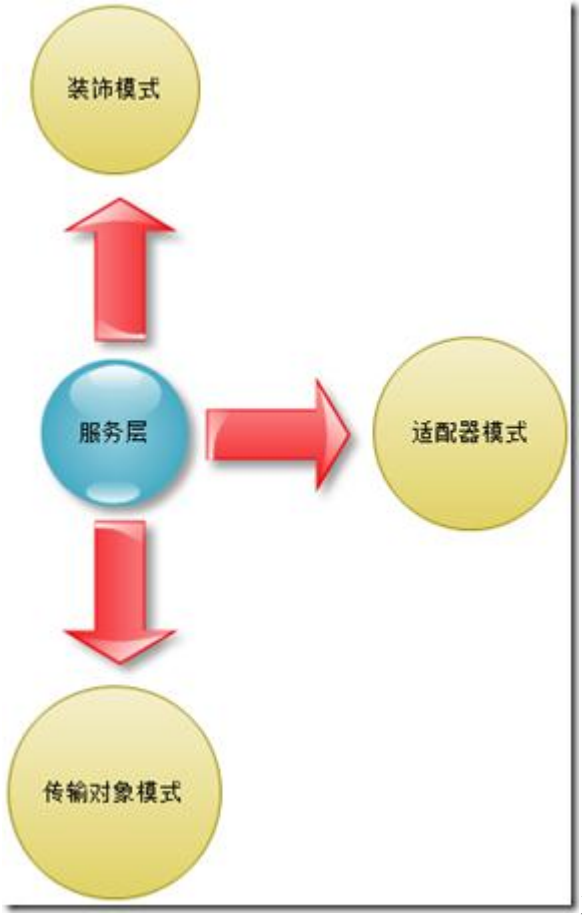
当然有时候我们对服务分为二种类型，粗粒度的服务与细粒度的服务，那么我们怎么理解它呢？其实说白了粗粒度的服务就是某个大的服务，而细粒度的就是大

的服务内部的子服务。可以这样理解，粗粒度是按照用例来组织操作的，例如我们上面的银行取款服务，那么粗粒度的服务就看作这样的流程，用户插卡-输入密码-取

款-退卡。而细粒度的服务关系的是：检查用户账户，余额的转换等，包括一些比较详细的，密码的验证等等，这些都是细粒度的服务，可以把粗粒度看作某个用例的大

的业务操作，对领域中的对象不关心，只关心领域模型中的交互。细粒度可以看作领域模型中的具体对象及对象的行为。

接下来我们将讲述服务层常用的几种架构模式：




第五节中将会详细的讲述每种模式及模式直接的区别及应用。

我们来看看服务层的应用场景及什么情况下使用服务层？

一般来说服务层适合企业应用系统中，也适合多层系统中，特别是业务逻辑比较复杂的系统程序中，如果说应用程序在多中形式的终端上使用时，推荐使用服务

层，当然如果你的系统中只有一种类型的前端表现形式时，例如 Web 应用程序，只有一个前端要求时，例如通过浏览器访问的形式，并且业务逻辑层能够很好的与表现层交互时，那么如果我们还把业务逻辑与表现层直接的通信抽出来通过服

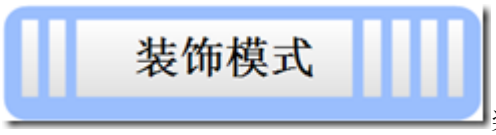
务层来完成的话，那么服务层只是完成任务的转发，并没有实际行的好处及减少开销，那么此时不推荐使用服务层。

我们来总结下服务层的优势与劣势，已衡量我们在系统中使用服务层的必要性：


## 五、服务层实战

前面已经讲述了服务层的优缺点及应用场景的介绍，那么我们本节中将要详细的讲解服务层设计的几种模式，主要是体现出服务层设计实现的几个思路，方便我

们在项目的实践过程中少走弯路。为我们的系统带来更好的适应性及扩展性要求。我们闲来将第一类模式




装饰模式在服务层的含义，跟设计模式中的装饰模式可以说是有着异曲同工之妙，就是将服务层的一系列方法包装成干净

的接口，以供外部调用，通过该模式，我们能够将细粒度的服务包装成粗粒度的服务，这样可以更方便的为表现层服务，并且可以通过装饰模式将服务包装成远程调用

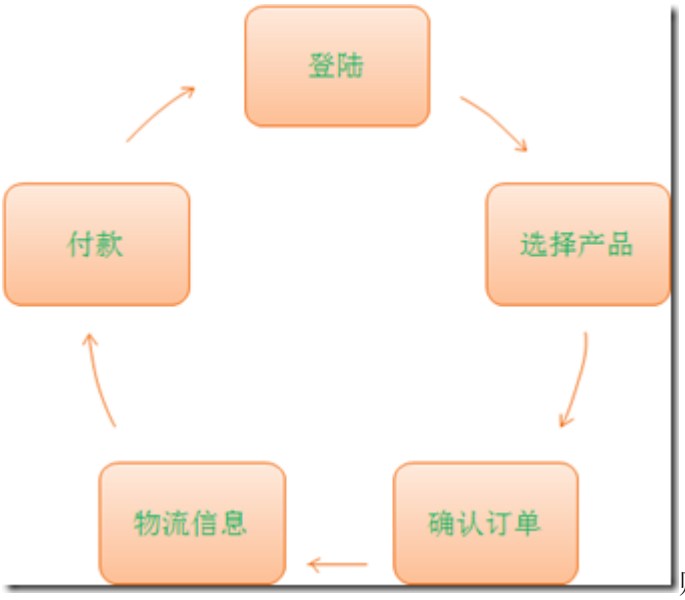
服务的方式，具体内部的实现都不是主要的关注点，对客户来说，他们可以以统一的方式不管客户端的形式是怎么样的。

我们都知道面向对象的设计原则是要求某个对象功能单一，尽可能的简单，但是通常我们在表现层中的一些业务流程中要求有多个实体之间进行交互时，那么我

们通过服务来组织就会显得比较好，我们通过装饰模式来实现这样的业务要求就会比较好，我们将一系列细粒度比较复杂的业务逻辑放在一个服务的 API 方法中，表现层

通过调用这个方法来完成复杂的业务逻辑交互。

服务层中的装饰模式更关心的是如何为表现层提供更好的服务，隐藏内部的细节，下面我们来看看相关的例子吧，我们这里还是以购物流程来说吧。




购物流程的简单流程可能如此，那么当然我们在购物流程中还有其他的服务，比如我们在购物的过程中的

短信提醒给卖家，或者说是发送邮件给卖家。等这些是我们系统提供的服务，包括一些日志性的服务。那么这些我们如何去做呢？当然在上面的流程中，用户只需要关

系最后的一步，提交订单，付款的环节，当用户付款后会给用户发送短信，那么显然我们在服务中就可以把下单的过程中默认提供系统日志，系统安全，权限等等问题

一并处理，而给客户提供的方法则只包含支付的接口。

```
public interface IPay
{
    /// <summary>
    /// 支付接口
    /// </summary>
    /// <param name="product"></param>
    void Pay(Rule.Product product);
}
```

上面我们定义了支付的接口，来看业务层中的订单操作：

```
public class Order
{
    /// <summary>
    /// 添加产品
    /// </summary>
    /// <returns></returns>
    public int Add(Product product)
    {
        return 0;
    }
    /// <summary>
    /// 保存
    /// </summary>
    /// <returns></returns>
    public int Save()
    {
        return 0;
    }
    /// <summary>
    /// 删除
    /// </summary>
    /// <returns></returns>
    public int Delete()
    {
        return 0;
    }
    /// <summary>
    /// 更新
    /// </summary>
    /// <returns></returns>
```

```
    public int Update()
    {
        return 0;
    }
}
```

我们来看看服务类中接口的实现：

```
public class Pay : IPay
{
    public bool PayMent(Rule.Product product)
    {
        //具体的下单操作
        Rule.Order order = new Rule.Order();
        //持久化操作
        order.Add(product);
        //发送手机短信，发送给卖家，通知有人买什么产品等
        SendMessage.Instance.SendMsg(string.Empty, string.Empty);
        return true;
    }
    #region IPay 成员
    public void IPay.Pay(Rule.Product product)
    {
        this.PayMent(product);
    }
    #endregion
}
```

那么上面我们在服务层组合多个简单的服务提供给一个方法，那么 UI 层只要简单的调用服务层接口中提供的方法即可，就能完成服务的调用。我们来看看 UI 层的代码

```
public class Order
{
    public void Pay()
    {
        Service.PayFactory factory=new Service.PayFactory();
        //调用服务层中的支付
        Service.IPay pay = factory.CreatePay();
        //这里只是测试，所以没有屏蔽New的代码
        pay.Pay(new Rule.Product());
    }
}
```

那么通过上面的简单形式我们就完成了一个简单的装饰模式的服务层的设计，是不是很简单呢？可能看起来代码有点多，不过这样的设计很利于我们在后期的扩展

性和适应性，特别是等到系统的功能更复杂更多时。好的设计就能体现出它的价值了。

当然上面我们通过了直接使用领域模型中的对象作为数据传输，当然我们可以通过数据传输对象的自定义对象来完成，情况更好，我这里就不举例说明了，下面我


们来讲述下一个模式：传输对象模式。那么我们前面也讲过了数据传输对象，其实这个模式只是讲解了数据传输对象的用法。

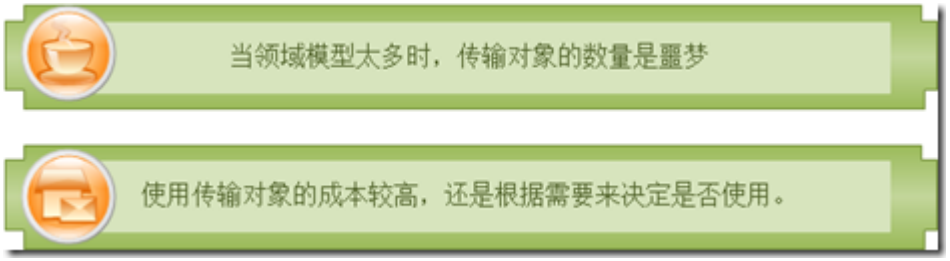
传输对象模式：

该模式主要是针对系统中各分层之间的数据传输模式的设计，通过传输对象可以降低各层之间分发数据的次数，提高系统性能，通常来说该模式非常有用。但是也

有它的弊端。比如说当领域模型太多的时候，如果把领域模型中的每个对象的数据载体，都设计成传输对象，那么系统将是一个非常庞大的工程，因为过度设计，让系

统难于维护与控制。我们来总结下使用该模式的优缺点：


那么有优点肯定就有缺点，我们来看看传输对象可能带来的劣势：




现在目前我们在使用数据传输对象的时候，都必须手动的去维护及创建，目前没有比较好的工具去完成自动创建的功能。比如说能将同一个对象，根据不同 UI 的需

求自动的将一些属性屏蔽或者启用等。可能通过 XML 配置文件来完成会是可行的方案，不过目前还没有一个比较好的工具去自动的根据领域模型中的对象自动的创建传

输对象，然后还能提供这个传输对象根据不同 UI 界面要求完成不同的自定义配置功能，希望各位如果了解的可以给小弟指点下，跪求！

传输对象模式我想具体的实例代码我就简单的书写下吧，就是把对象中的行为去掉，只包含数据信息，就和我们平时说的 3 层结构中的 Model 层一样，只有 get;set;

访问器和私有成员变量，我们来看看实例代码吧？

```
/// <summary>
/// 产品信息
/// </summary>
public class Product
{
    private int _pro_id;
    private string _pro_property = string.Empty;
    private string _pro_cid;
    private int? _pro_brandid;
    private string _pro_name;
```

```
private string _pro_model;
/// <summary>
/// 产品 ID
/// </summary>
public int pro_ID
{
    set
    {
        _pro_id = value;
    }
    get
    {
        return _pro_id;
    }
}
/// <summary>
/// 扩展属性值
/// </summary>
public string pro_Property
{
    set
    {
        _pro_property = value;
    }
    get
    {
        return _pro_property;
    }
}
/// <summary>
```

```
/// 商品分类
/// </summary>
public string pro_CID
{
    set
    {
        _pro_cid = value;
    }
    get
    {
        return _pro_cid;
    }
}
/// <summary>
/// 商品品牌
/// </summary>
public int? pro_BrandID
{
    set
    {
        _pro_brandid = value;
    }
    get
    {
        return _pro_brandid;
    }
}
/// <summary>
/// 商品名称
/// </summary>
```

```
        public string pro_Name
        {
            set
            {
                _pro_name = value;
            }
            get
            {
                return _pro_name;
            }
        }
        /// <summary>
        /// 商品型号
        /// </summary>
        public string pro_Model
        {
            set
            {
                _pro_model = value;
            }
            get
            {
                return _pro_model;
            }
        }
}
```

我这里提供一个我认为的生成领域模型的思路，主要还是通过 XML 文件来完成，将具体的数据传输对象不是通过类文件的形式来完成，通过序列化成 XML 文件来完成,这样就相当于每个 XML 对应一个序列化文件,然后这个文件中会保存相应的配置信息，比如说哪个页面显示哪些字段，那个页面调用这个类时不掉用这个页面。具

体的配置通过提供一个可视化的方式来维护就好了，然后在前台绑定的时候根据读取或者写入 XML 文件来完成，可能这也是比较灵活的方式，具体的实现我没有去做，

请大家提出更好的思路，小弟谢过！

我们接下来讲述第三种模式：适配器模式，这个也是设计模式中最常用的设计模式的一种，适配器模式的主要作用是将某个接口转换成我们需要的另外一个接口，

这个怎么理解呢？

我们把手机服务包装成 MP3 的接口，或者把 MP3 的接口包装成手机，都是可以的，可能我这里的例子举得不合适

但是意思就是将某种服务，通过适配器转换成另外一种服务。

我这里简单的讲解几个例子来完整适配器模式的介绍，我们先以将传输对象转换为我们的领域模型中的对象，通过适配器来完成数据的转换。

我们先来看看不通过适配器模式来完成领域对象中的类与传输对象之间的交互，通过构造函数注入的方式来完成。

```
/// <summary>
/// 产品信息
/// </summary>
public class ProductCase
{
    private Product _product;
    public ProductCase(Product product)
    {
        _product = product;
    }
    /// <summary>
    /// 产品 ID
    /// </summary>
    public int pro_ID
```

```
{
    set
    {
        _product.pro_ID = value;
    }
    get
    {
        return _product.pro_ID;
    }
}
/// <summary>
/// 扩展属性值
/// </summary>
public string pro_Property
{
    set
    {
        _product.pro_Property = value;
    }
    get
    {
        return _product.pro_Property;
    }
}
/// <summary>
/// 商品分类
/// </summary>
public string pro_CID
{
    set
```

```
    {
        _product.pro_CID = value;
    }
    get
    {
        return _product.pro_CID;
    }
}
/// <summary>
/// 商品品牌
/// </summary>
public int? pro_BrandID
{
    set
    {
        _product.pro_BrandID = value;
    }
    get
    {
        return _product.pro_BrandID;
    }
}
/// <summary>
/// 商品名称
/// </summary>
public string pro_Name
{
    set
    {
        _product.pro_Name = value;
```

```
        }
        get
        {
            return _product.pro_Name;
        }
    }
    /// <summary>
    /// 商品型号
    /// </summary>
    public string pro_Model
    {
        set
        {
            _product.pro_Model = value;
        }
        get
        {
            return _product.pro_Model;
        }
    }
}
```

上面的方式通过构造函数的注入完成相应的访问，通过 get;set;访问器来完成。

下面我们通过适配器的方式来实现转换，看看有什么不同：

```
public class ProductTest
{
    private Product _product;
    public ProductTest(Product product)
    {
```

```
            ProductAdapter adapter = new ProductAdapter(product);
            adapter.InitDTO(this);
        }
    }
```

下面看看具体的适配器中的代码：

```
    public class ProductAdapter
    {
        private Product _product;
        public ProductAdapter(Product product)
        {
            this._product = product;
        }
        public bool InitDTO(ProductTest test)
        {
            //赋值的过程，将 Product 中的信息转换为 ProductTest 对象
            test.pro_BrandID = _product.pro_BrandID;
            //...
            return true;
        }
    }
```

我们上面看到了，通过依赖注入的形式，将要包装的接口传入到适配器，然后在适配器的内部进行相应的包装，传出包装后的接口，这就是一个完整的适配器流程，具体的业务逻辑就是根据需要来做了。通过上面的方式我们的确完成了相应的转换，不过转换的代价是非常的大，不过有的时候我们的业务需求是这样的，可能我们也没有更好的办法，只能通过这样的方式来做，可能对解决方案的实现比效率更有价值。其实我们在使用传输对象的时候还是需要仔细的斟酌项目的需求，看看是不是必须要使用这个，如果不是必须的，其实我们可以不需要强迫性的使用。

## 六、本文总结

本章主要讲述了系统架构中的服务层的架构中的注意事项及几个简单的设计模式及使用，并且讲述了服务层应用的场景和带来的好处，当然我们也需要服务层的

优劣处，还有就是服务的实现方案，本文前面可能没有讲解发布服务的几种方式，这里简单的用图来说明下吧？


WCF 已经内置继承了 remoting，socket 和 SOAP 的方式来进行远程调用服务，当然 HTTP 方式的 SOAP 的服务方

式，还是推荐使用 Web 服务的方式来做。

## 七、系列进度

前篇

1、系统架构师-基础到企业应用架构系列之--开卷有益

2、系统架构师-基础到企业应用架构-系统建模[上篇]

3、系统架构师-基础到企业应用架构-系统建模[中篇](上)

4、系统架构师-基础到企业应用架构-系统建模[中篇](下)

5、系统架构师-基础到企业应用架构-系统建模[下篇]

6、系统架构师-基础到企业应用架构-系统设计规范与原则[上篇]

7、系统架构师-基础到企业应用架构-系统设计规范与原则[下篇]

8、系统架构师-基础到企业应用架构-设计模式[上篇]

9、系统架构师-基础到企业应用架构-设计模式[中篇]

10、系统架构师-基础到企业应用架构-设计模式[下篇]

中篇

11、系统架构师-基础到企业应用架构-企业应用架构

12、系统架构师-基础到企业应用架构-分层[上篇]

13、系统架构师-基础到企业应用架构-分层[中篇]

14、系统架构师-基础到企业应用架构-分层[下篇]

15、系统架构师-基础到企业应用架构-表现层

16、系统架构师-基础到企业应用架构-服务层

17、系统架构师-基础到企业应用架构-业务逻辑层

18、系统架构师-基础到企业应用架构-数据访问层

19、系统架构师-基础到企业应用架构-组件服务

20、系统架构师-基础到企业应用架构-安全机制

后篇

21、单机应用、客户端/服务器、多服务、企业数据总线全解析

22、系统架构师-基础到企业应用架构-单机应用(实例及 demo)

23、系统架构师-基础到企业应用架构-客户端/服务器(实例及 demo)

24、系统架构师-基础到企业应用架构-多服务(实例及 demo)

25、系统架构师-基础到企业应用架构-企业数据总线(实例及 demo)

26、系统架构师-基础到企业应用架构-性能优化(架构瓶颈)

27、系统架构师-基础到企业应用架构-完整的架构方案实例[上篇]

28、系统架构师-基础到企业应用架构-完整的架构方案实例[中篇]

29、系统架构师-基础到企业应用架构-完整的架构方案实例[下篇]

30、系统架构师-基础到企业应用架构-总结及后续

## 八、下篇预告

下一篇我们将会开始讲解软件设计中最重要也是最基本的技能-设计模式，希望大家多多提出已经，后面 3 篇我们将会讲解如何在项目中使用设计模式及使用设计

模式需要注意的事项,将会举例说明每个设计模式可能出现的场景。希望大家持续关注！