# 系统架构师-基础到企业应用架构-企业应用架构

## 一、上篇回顾

我们先来回顾下上篇讲解的内容，我们前面的几节分别讲述了，业务逻辑层、数据访问层、服务层、表现层，我们了解了这些分层的职责和分层之间的大概的关联

关系，本篇可能主要是简单的介绍下企业应用的几类模式，结合这几个分层直接的交互来完成系统功能的构建。我们还是先对我们学习的四个分层的职责和功能做个大

概的回顾，我们先来看看下图来回顾下我们讲述的内容。


表现层
Web
WinForm
WPF
SilverLight
WindowsMobile
MVC
Model2
MVP
PV
SVC
PM
MVVM
系统架构分层模式
数据访问对象
表现层
服务层
业务逻辑层
数据访问层
服务层针对四种业务逻辑模式的不同的处理方式。
业务逻辑层
业务逻辑层的四大模式：
事务脚本
表模块
活动记录
领域模型
业务逻辑层中的基类模式的应用环境及目前使用个类模式的优缺点。
数据访问层
数据访问层的职责和四个基本职能
CUD、
Read服务、
事务、
并发
业务逻辑层中的基类模式的应用环境及目前使用个类模式的优缺点。

我想通过上图，大家能回忆起我们讲述的相关内容，然后整理好自己的思路，我们本文将会针对这几个分层进行相应的模式的讲解，并且会结合实例来说明企业应

用架构的简单应用。我想这也是大家关心的内容，我也希望大家能多提出宝贵意见，大家共同提高。

之前说是提供 PDF 文件的下载的，以提供给需要学习的朋友更方便的形式，但是由于最近时间有限，没能整理好，等过阵子提供一个完整的整合好的 PDF 版本，提

供给大家下载，还望大家见谅。

## 二、开篇

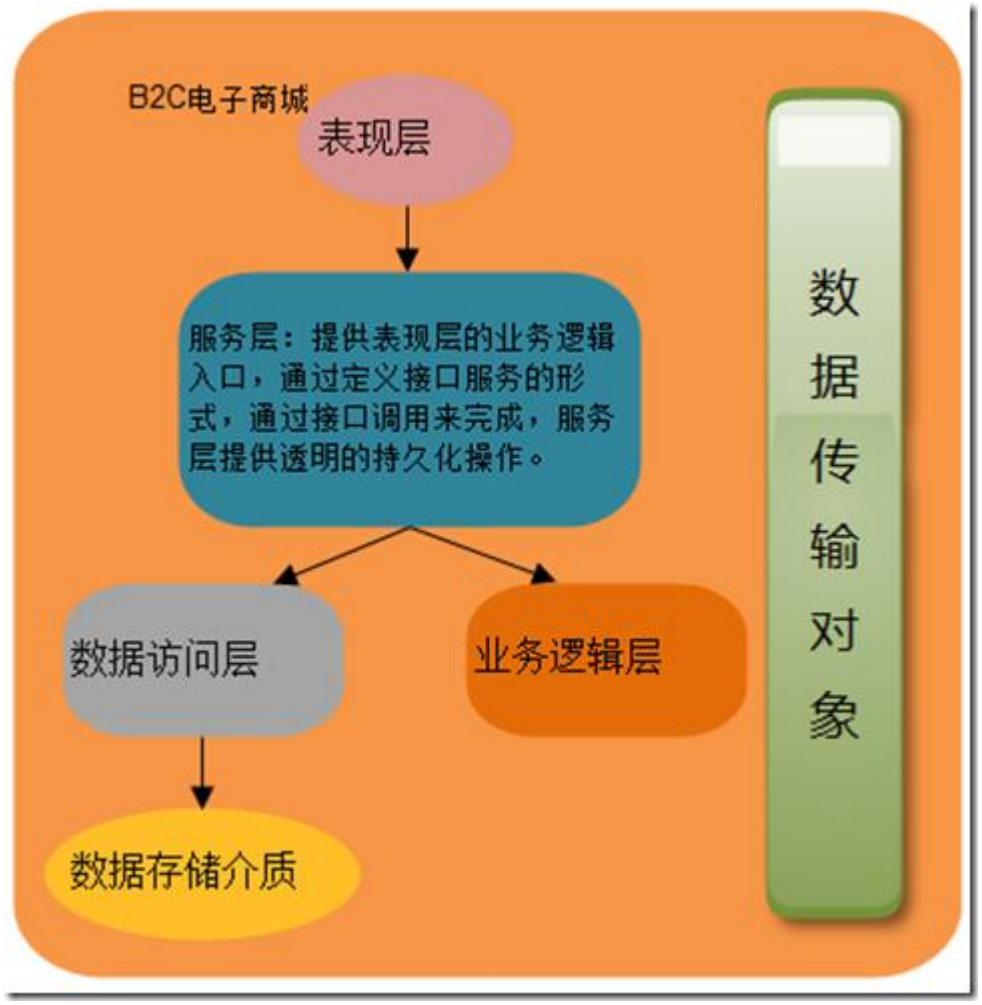
本篇我们将针对我们前面讲述的几个分层做个简单的整合，就是通过一个简单的实例代码来说明下，我们给出的一个可能的企业应用架构模式去完成企业应用，并

且顺带分析下，企业应用中可能存在的瓶颈等，当然可能本章只是点出一些概念，然后在后面的章节中去完成，比如我们的性能优化，性能瓶颈等，这些我们后面通过

这篇：**系统架构师-基础到企业应用架构-性能优化(架构瓶颈)(后续篇)**详细的分析。

当然由于本章主要是基于前面讲述的内容的一个整合和分析，可能部分观点

还有更好的改进方案，还希望大家多提出宝贵意见和您的建议，谢谢！那么我们先来看看我们本章讲述的内容吧：




本章主要讲述下各分层之间的交互方式及可行的设计方案，并且分析我们在每个分层采用什么样的模式，及给出相应的示例代码，当然这里可能讲述的不会是最好的实现方案，还期待大家提出更好的改进方案。

## 三、本文提纲

1、内容回顾

2、开篇

3、本文提纲

4、企业应用架构实例

4.1、企业应用架构中的物理分层

4.2、数据访问层分析

4.3、业务逻辑层分析

4.4、服务层分析

4.5、表现层分析

4.6、分析总结

5、结束语

6、系列进度

7、下篇预告

## 四、企业应用架构实例

### 4.1、企业应用架构的物理分层

本节将会分层的相关介绍，并且分析分层的原因，为下篇：**系统架构师-基础到企业应用架构-分层[上篇] 做铺垫。**

分层我想对大家来说都不会太陌生。因为我们平时在开发的过程中一般都是采用分层架构的方式，通过分层将系统的功能进行划分，我们将系统中的若干有共同特征的部分放在一个分层中，然后通过分层之间的交互去完成系统的功能。

我们看看企业应用的几种应用程序模式：

1、单击应用程序就是一些本地服务的应用程序，这个应该没啥特别的难度，例如 Office 应用程序，当然现在有在线版本的 Office，这个应用的太多了。

2、客户端/服务器的形式，特别是在 WinForm、WPF、SilverLight 方面的应用等，例如我们的常用的 OutLook，虽然不是采用.NET 平台开发的，但是原理一样就

是通过客户端通过通信服务来访问服务器，然后取回相应的邮件信息返回给客户端，相当于我们通过客户端的形式来访问 Web 服务。

3、Web 服务：通过将开发的 Web 服务器部署在服务器上，然后我们就可以通过输入 HTTP 网址的形式来访问 web 服务，我们这里是通过浏览器来完成的，其实这个

模式也是客户端/服务器的形式。只不过这时的客户端变成浏览器+服务页面，然后通过浏览器来完成 Web 服务的访问。

4、其他服务：我们这里主要是针对一些其他的设备，例如通信设备等方面的访问，比如说我们现在提供一个卫星定位的服务，那么通过设备调用服务来完成，告知

用户的 GPS 定位信息等。很多这个方面的应用。这个方向应该是未来的一个主流吧。

我们上面简单的讲述了系统的物理分层的形式，那么我们来看看如何对系统的功能进行分层，也就是我们下面要详细讲解的实例分析。

## 4.2、数据访问层分析

数据访问层我们知道是唯一一个可以与数据库之间直接进行交互的分层，数据访问层必须满足的几个功能和职责，我们这里就不复述了，我们本篇可能就是直接给

出数据访问层的相关实现，并且分析出数据访问层与其他层之间的交互。


大家可能一看就知道是什么意思，可能在您的眼中应该就和你们平时项目中采用的分层结构有点类似吧。

我这里的数据访问层提供所有的数据库访问服务。我们来看看基本的服务功能吧

我们这里定义了一个 DDL 语句操作的枚举，CUD 的枚举，因为我们提供了一个统一的数据访问服务，通过 Excute 来完成。

```
/// <summary>
/// 数据访问操作类型
/// </summary>
public enum DDLType
{
    Create,
    Update,
    Delete
}
```

这里用枚举来定义数据库字段与查询条件值之间的关系

```
/// <summary>
/// 数据字段与字段值之间的关系表达式
/// </summary>
public enum FieldExpressionType
{
    EquleTo,
    BeginThen,
    LessThen,
    BetweenAnd,
    Like
}
```

我们再来看看我们定义的查询条件的接口

```
    /// <summary>
    /// 定义子查询接口
    /// </summary>
    public interface ICondition
    {
        int Add(string fieldName,object value,FieldExpressionType
fetType);
        int Add(ICondition condition);

        string WhereCondition;
        string OrderCondition;
    }
```

数据访问层接口代码

```
    /// <summary>
    /// 定义统一的数据访问服务
    /// </summary>
    public interface IDataAccess
    {
        #region CUD

        int Create<T>(T item);

         int Update<T>(T item);

         int Delete<T>(T item);

         int Execute<T>(T item,DDLType ddlType);
```

```
        #endregion

        #region Read服务

         List<T> Query<T>(ICondition condition);

         T GetModelByPrimaryKey<T>(object key);

         List<T> GetAll<T>();

         #endregion

        #region 事务操作

         void BeginTransaction();

         bool Commit();

         bool RollBack();

         bool IsTransaction;

         #endregion
    }
```

我们知道具体的代码我们是通过反射+特性的形式来完成数据库操作语句的生成的，我们来看看可行的代码，属性项特性的定义。

```
    /// <summary>
```

```
/// Model中的字段属性特性
/// </summary>
[AttributeUsage(AttributeTargets.All, AllowMultiple = false)]
public class PropertyAttribute : Attribute
{
    private string dbColumnName;
    private bool isPrimary;
    private DbType dbType;
    private object defaultValue;
    private bool isIdentify;
    private int length;

    public string DbColumnName
    {
        get
        {
            return this.dbColumnName;
        }
        set
        {
            this.dbColumnName = value;
        }
    }

    public bool IsPrimary
    {
        get
        {
            return this.isPrimary;
        }
```

```
    set
    {
        this.isPrimary = value;
    }
}

public bool IsIdentify
{
    get
    {
        return this.isIdentify;
    }
    set
    {
        this.isIdentify = value;
    }
}

public DbType DbType
{
    get
    {
        return this.dbType;
    }
    set
    {
        this.dbType = value;
    }
}
```

```
        public object DefaultValue
        {
            get
            {
                return this.defaultValue;
            }
            set
            {
                this.defaultValue = value;
            }
        }

        public int DbLength
        {
            get
            {
                return this.length;
            }
            set
            {
                this.length = value;
            }
        }
        public PropertyAttribute(string dbName, bool isPrimery, DbType
type,object dValue)
        {
            this.dbColumnName = dbName;
            this.isPrimary = isPrimery;
            this.dbType = type;
            this.defaultValue = this.GetDefaultValue();
```

```
        }

        private object GetDefaultValue()
        {
            return new object();
        }

        public PropertyAttribute(string dbName)
        {
            this.dbColumnName = dbName;
            this.isPrimary = false;
            this.dbType = DbType.String;
            this.defaultValue = this.GetDefaultValue();
        }

        public PropertyAttribute(string dbName,bool isPrimery)
        {
            this.dbColumnName = dbName;
            this.isPrimary = isPrimery;
            this.dbType = DbType.String;
            this.defaultValue = this.GetDefaultValue();
        }

        public PropertyAttribute(string dbName, bool isPrimery, DbType
type)
        {
            this.dbColumnName = dbName;
            this.isPrimary = isPrimery;
            this.dbType = type;
            this.defaultValue = null;
```

```
    }
}
```

我们再来看看基于表上的特性

```
/// <summary>
/// 基于表的自定义特性类
/// </summary>
[AttributeUsage(AttributeTargets.All, AllowMultiple = false)]
public class TableAttribute : Attribute
{
    private string dbTableName;
    public TableAttribute(string dbName)
    {
        this.dbTableName = dbName;
    }

    public string TableName
    {
        get
        {
            return this.dbTableName;
        }
        set
        {
            this.dbTableName = value;
        }
    }
}
```

根据反射取出表名

```
/// <summary>
/// 返回 Model 对应的数据库表名
/// </summary>
/// <typeparam name="T"></typeparam>
/// <param name="model"></param>
/// <returns></returns>
public string DbTableName<T>(T model)
{
    string dbName = string.Empty;
    DPM.Common.TableAttribute attr = null;

    object[] attributes =
model.GetType().GetCustomAttributes(typeof(DPM.Common.TableAttribute),
true);

    if (attributes.Length > 0)
    {
        attr = (DPM.Common.TableAttribute)attributes[0];
    }

    if (attr != null)
        dbName = attr.TableName;

    return dbName;
}
```

根据反射取出表中的列

```
/// <summary>
```

```
        /// 动态创建表中字段列表
        /// </summary>
        /// <typeparam name="T"></typeparam>
        /// <param name="model"></param>
        /// <returns></returns>
        public string InitDbColumns<T>(T model)
        {
            StringBuilder commandBuilder = new StringBuilder();

            DPM.Common.PropertyAttribute attr = null;

            foreach (PropertyInfo property in
model.GetType().GetProperties())
            {
                object[] attributes =
property.GetCustomAttributes(typeof(DPM.Common.PropertyAttribute),
true);
                if (attributes.Length > 0)
                {
                    attr = (DPM.Common.PropertyAttribute)attributes[0];
                }

                if(commandBuilder.length>0)

                      commandBuilder.Append(“,”);

                commandBuilder.Append(attr.DbColumnName);
            }

            return commandBuilder.ToString();
        }
```

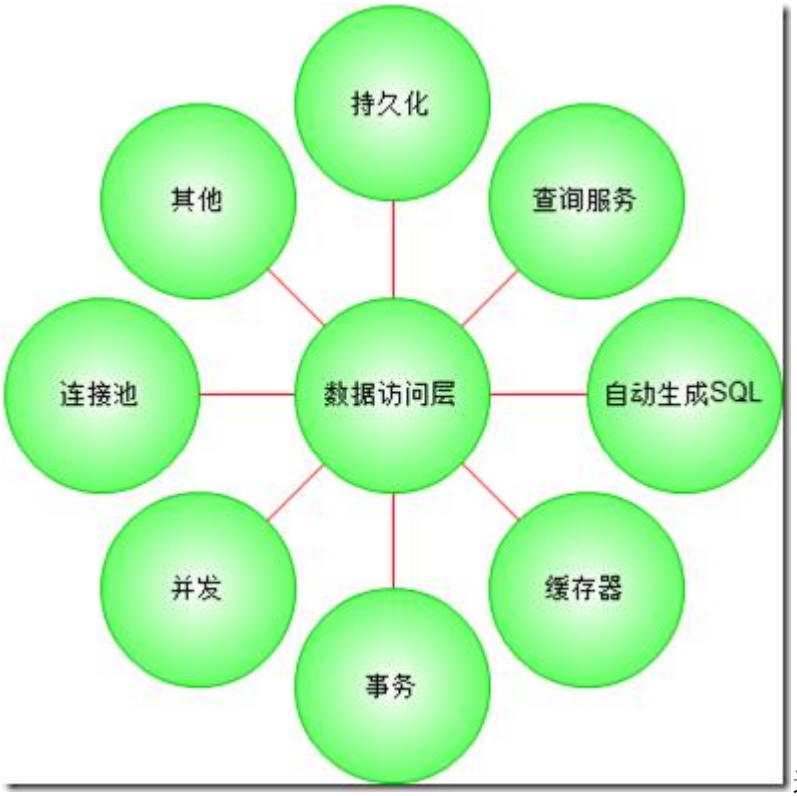
当然这里只是给出了简单的示例，我们来看看生成的 Insert 语句的格式吧

```
        /// <summary>
        /// 动态创建表中字段更新列表
       /// </summary>
        /// <typeparam name="T"></typeparam>
        /// <param name="model"></param>
        /// <returns></returns>
        public string InitInsertColumns<T>(T model)
        {
            StringBuilder filedBuilder = new StringBuilder();
            StringBuilder valueBuilder = new StringBuilder();
            StringBuilder commandBuilder = new StringBuilder();

            DPM.Common.PropertyAttribute attr = null;

            foreach (PropertyInfo property in
model.GetType().GetProperties())
            {
                object[] attributes =
property.GetCustomAttributes(typeof(DPM.Common.PropertyAttribute),
true);
                if (attributes.Length > 0)
                {
                    attr = (DPM.Common.PropertyAttribute)attributes[0];
                }
                if (attr.DbColumnName == "")
                    continue;
                if (attr.IsIdentify)
                    continue;
```

```
                if (filedBuilder.Length > 0)
                {
                    filedBuilder.Append("," + attr.DbColumnName);
                    valueBuilder.Append("," + property.GetValue(model,
null));
                }
                else
                {
                    filedBuilder.Append(attr.DbColumnName);
                    valueBuilder.Append(property.GetValue(model,
null));
                }

            }

            commandBuilder.Append(SqlDMLText.LKH);
            commandBuilder.Append(filedBuilder.ToString());
            commandBuilder.Append(SqlDMLText.RKH);
            commandBuilder.Append(SqlDMLText.VALUES);
            commandBuilder.Append(SqlDMLText.LKH);
            commandBuilder.Append(valueBuilder.ToString());
            commandBuilder.Append(SqlDMLText.RKH);

            return commandBuilder.ToString();
        }
```

其他的语句类似了，这部分详细的代码我们会在 ORM 篇详细的讲述，并且对反射的性能通过优化来提供性能。

我们来总结下数据访问层应该有的功能吧。我认为应该提供以下的功能

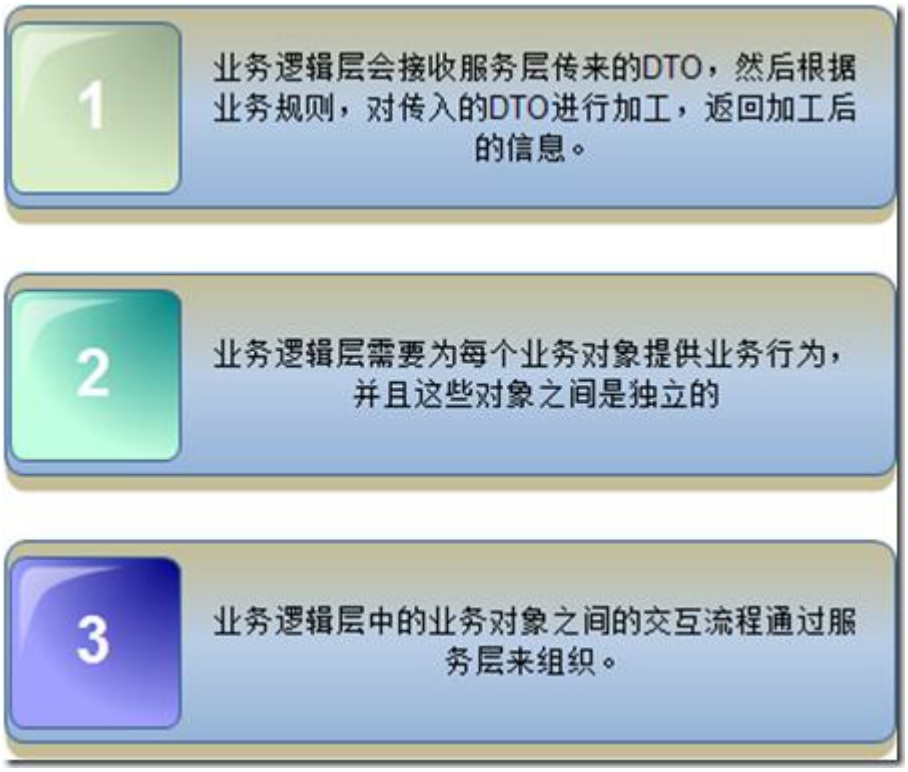


这样的一个数据访问层基本上能够满足功能的需要。当然我这里就不把代码全部贴出来了，太多了，我们接下来讲讲业务逻辑层吧。

### 4.3、业务逻辑层分析

上篇我们贴出来了一些比较典型的数据访问层的代码，当然没有上全部的代码，本文将会提供 demo 实例下载，大家可以参考其中的部分代码。我们来看看业务层中的每个业务对象应该具有什么样的智能呢？我们知道我们这里不建议业务对象直接来访问数据访问层，那么我们如何实现持久化透明的方式呢？我记得之前我讲述业务逻辑层的时候也提到这个方面的要求了，还有不少热心的园友问我如何实现，我这里给出几个可能的办法，当然如果有更好的办法，还请大家多多提出。我这里给出个简单的业务对象的职责，当然我这里给出的职责是我理解的在我的这个实例中业务对象应该具有的职责。

接下来我们看看业务逻辑层的职责吧，然后我们给出示例代码




我们来举例说明一个业务对象可能具有的功能吧？我们这里以还是以B2C系统为例吧，我们来说说订单的可能行为吧：

```
/// <summary>
/// 订单
/// </summary>
 public class Order
{
    /// <summary>
    /// 产品列表
    /// </summary>
    private List<Product> products = new List<Product>();

    /// <summary>
    /// 获取订单的金额
    /// </summary>
    /// <param name="order"></param>
```

```
        /// <returns></returns>
        public decimal GetOrderCount(Model.Order order)
        {
            decimal totalCount = 0;
            foreach (Product product in products)
            {
                totalCount += product.Cash;
            }

            return totalCount;
        }

        /// <summary>
        /// 获取该订单下的所有产品
        /// </summary>
        /// <param name="order"></param>
        /// <returns></returns>
        public List<Product> GetProduct(Model.Order order)
        {
            return new List<Product>();
        }
    }
```

当然这里面的取得与订单相关的产品列表，因为我们没有设计在 DTO 中默认包含这个属性，那么我们这里其实可以考虑增加延迟加载的形式，一旦加载后，也可以

第一次加载后，缓存起来，下次使用的过程中直接取出来。

我们来看看与这个订单业务逻辑相关的服务层代码

```
    /// <summary>
    /// 订单服务
```

```
    /// </summary>
    public class OrderService
    {
        private List<Model.Product> products = new
List<Model.Product>();

        /// <summary>
        /// 持久化透明的方式
        /// </summary>
        /// <param name="order"></param>
        /// <param name="type"></param>
        /// <returns></returns>
        public int Execute(Model.Order order,DAL.DDLType type)
        {
            return DAL.SQLServer.Instance.Execute(order, type);
        }

        /// <summary>
        /// 获取订单的总金额
        /// </summary>
        /// <param name="order"></param>
        /// <returns></returns>
        public decimal GetOrderCash(Model.Order order)
        {
            BLL.Order = new BLL.Order(order);

            return order.GetOrderCount();
        }

        /// <summary>
```

```
/// 保存订单信息
/// </summary>
/// <param name="order"></param>
/// <param name="type"></param>
/// <returns></returns>
public int SaveOrder(Model.Order order, DAL.DDLType type)
{
    int iAff = 0;
    //将与订单相关的产品信息也同步进行保存。
    //我们这里通过事务的方式进行处理
    try
    {
        DAL.SQLServer.Instance.BeginTransaction();

        //具体的逻辑
        //插入产品信息
        foreach (Model.Product product in products)
        {
            DAL.SQLServer.Instance.Create(product);
            iAff++;
        }

        //插入订单信息
       iAff+= DAL.SQLServer.Instance.Create(order);

        DAL.SQLServer.Instance.Commit();
        return iAff;

    }
    catch
```

```
            {
                DAL.SQLServer.Instance.RollBack();
                return 0;
            }
            finally
            {
                DAL.SQLServer.Instance.Dispose();
            }
        }
    }
```

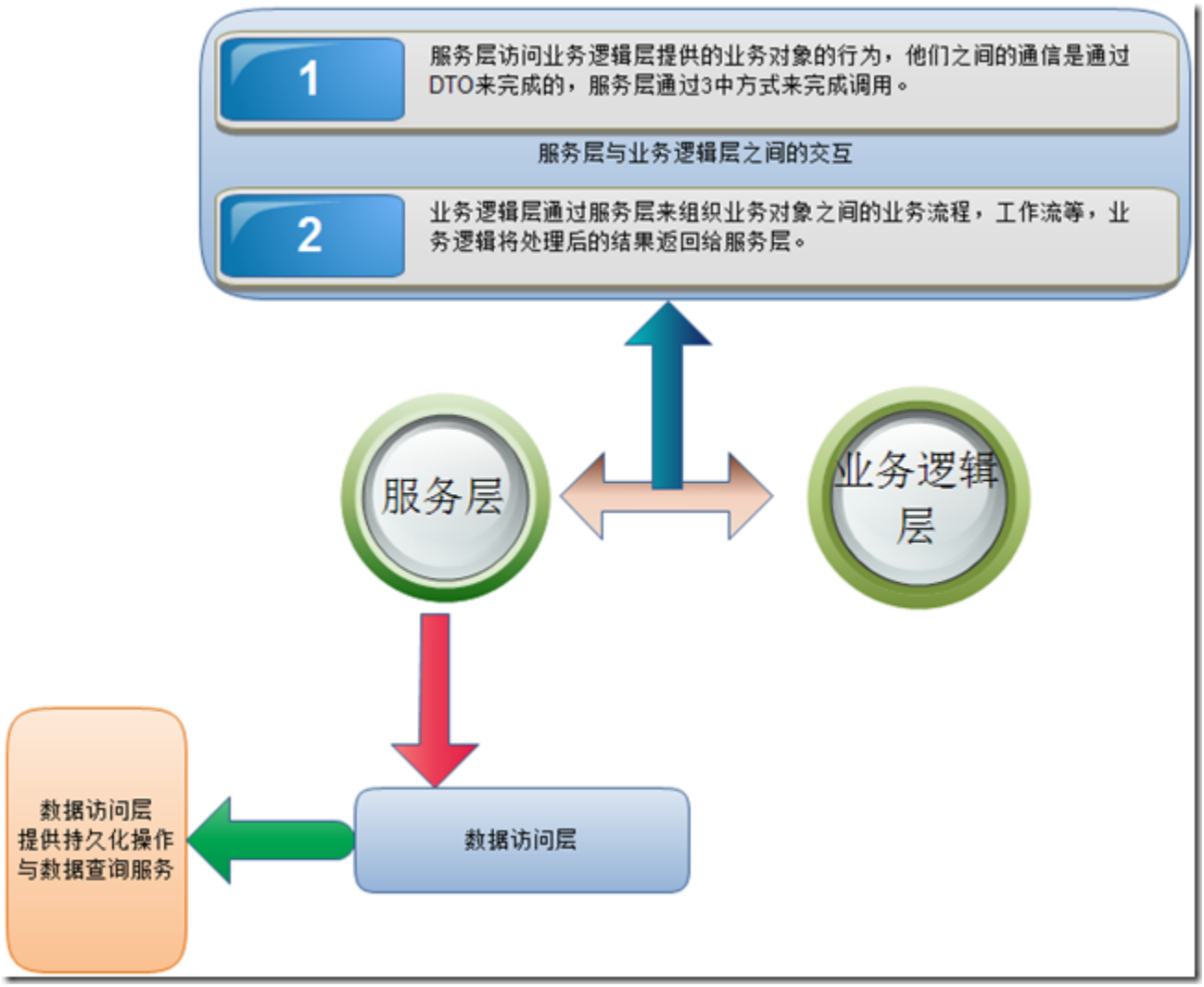
通过上面的代码我们可以看出，目前的业务逻辑层中的业务对象只是处理自身的相关业务逻辑，通过服务层与数据层进行交互，然后业务对象并不关系自身的

CRUD 的业务，而是通过服务层来完成的，那么对于 CUD 我想大家应该没有什么争议，因为持久化透明只是说把业务逻辑层原本通过继承或者是实现接口，或者依赖注

入的形式来完成持久化的操作。

我们这里则是将这样的持久化操作提出来放在服务层来完成，这样可以降低业务层与数据访问层的依赖，那么大家肯定关心如何实现业务逻辑对象的查询服务呢？

我的思路是这样的，大家请看看是否合理，或者您有好的思路或者想法一定要告知我，不甚感激！




不知道我们上面的上图是否表述的很清楚，还请大家多多的批评指出。当然上述的模式，可能在实际的企业应用中还是有点困难的，还有一种业务逻辑层的持久化

透明实现。就是通过 AOP 来完成，或者是通过代理类来实现。这里说心里话，我对 AOP 的具体实现并没有研究过，但是通过动态注入的方式的确可以实现这个功能。具

体的实现方式就交给大家来完成了。

### 4.4 服务层分析

服务层作为协调业务对象进行相应的业务流的控制，当然服务层只是协调业务对象之间的交互，并不是负责具体的业务逻辑，这里的服务层应该是只负责业务对象

之间的交互。当然我这里还把部分对数据完整性，数据类型等方面的内容放在服务层来做。下面来说明服务层可能包含的功能：


我们在业务逻辑层中也贴出了部分代码。当然服务层中其实还可以有很多的内容，比如通过我们在前面的服务层讲解时介绍的几个模式，我们也都是可以在服务层

中应用的。当然我们这里的实例代码可能就不会贴出这几个模式了，我们这里举例来说明服务层中的部分服务代码。例如我们前面经常举例说明的订单管理中的提醒功

能。

比如我们有时候我们的提醒功能，需要邮件提醒或者是短信提醒的几种服务方式，这时候我们可以通过提供统一的服务接口，然后不需要单独的在界面层去单独的

定义，我们通过接口的形式为后期的变化方便扩展，我们来给出部分代码：

```
/// <summary>
/// 提醒服务
/// </summary>
public interface IAlarmService
{
    /// <summary>
    /// 发送消息的服务
    /// </summary>
    /// <param name="reciveObject">接收方</param>
    /// <param name="title">标题</param>
```

```
        /// <param name="content">消息内容</param>
        /// <returns></returns>
        bool SendMessage(string reciveObject,string title,string
content);
    }
```

我们来看看邮件提醒服务的简单实现代码

```
public class EmailAlarm : IAlarmService
    {
        #region IAlarmService 成员

        /// <param name="strSmtpServer">邮件服务器(如果是 163 邮箱就写
smtp.163.com)</param>
        /// strSmtpServer
        /// <param name="strFrom">发件人的帐号</param>
        /// strFrom
        /// <param name="strFromPass">发件人密码</param>
        /// strFromPass
        public bool SendMessage(string reciveObject, string title,
string content)
        {
            bool isSuccess = true;

            MailSetting model =
DAL.SQLServer.Instance.GetModelByPrimaryKey<MailSetting>();

            try
            {
                System.Net.Mail.SmtpClient client = new
SmtpClient(model.strSmtpServer);
```

```
                client.UseDefaultCredentials = false;

                client.Credentials = new
System.Net.NetworkCredential(model.strFrom, model.strFromPass);

                client.DeliveryMethod = SmtpDeliveryMethod.Network;

                System.Net.Mail.MailMessage message = new
System.Net.Mail.MailMessage(model.strFrom, reciveObject, title,
content);

                message.BodyEncoding = System.Text.Encoding.UTF8;

                message.IsBodyHtml = true;

                client.Send(message);
            }
            catch
            {
                isSuccess = false;
            }
            return isSuccess;
        }

        #endregion
    }
```

短信服务也和邮件服务类似，通过实现接口来提供服务，那么我们在比如说订单，或者其他的可能会提醒的地方，就可以通过自定义实现不同的格式服务，来完成调用。

## 4.5、表现层分析

我们都知道表现层是最终显示与用户交互的功能的，那么我们的表现层是怎么来调用服务层的呢？我们这里的实例是采用服务层调用的方式来完成。我们这里通过

定义接口的形式，然后通过实现不同的用户 UI，然后我们通过展示器来调用服务层，然后将服务层中的处理结果返回给展示器，展示器与视图之间进行交互，我们来看

看表现层的层级关系。


大家看到这个图请不要诧异，这里的展示器层可能还会直接访问业务逻辑层，也可能直接访问数据访问层，这

些都是有可能的，我们需要根据项目具体的情况去决定。

我们来看看 UI 层的视图代码：

我们在视图中保持展示器的引用。

```
/// <summary>
/// 定义统一的视图接口代码
/// </summary>
public interface IView
{
    /// <summary>
    /// 展示器访问器
    /// </summary>
    B2C.Presenter.IPresenter Presenter
```

```
    {
        get;
        set;
    }
}
```

我们来看看展示器中的部分代码

```
/// <summary>
/// 展示器接口
/// </summary>
public interface IPresenter
{
    /// <summary>
    /// 重新生成视图
    /// </summary>
    /// <returns></returns>
    string Review()
    {
        return string.Empty;
    }

    /// <summary>
    /// 操作 Model
    /// </summary>
    /// <typeparam name="T"></typeparam>
    /// <param name="?"></param>
    /// <returns></returns>
    T Controller<T>(T item);
}
```

视图层通过依赖注入的方式，或者是属性注入，还是实现接口的方式来实现展示器的引用，展示器提供接处理用户操作的功能，然后根据处理后的结果决定是否刷

新视图。

当然如果你这里是借助的 MVC 框架来实现的话，可能控制器中的代码就要是访问服务层，或者业务逻辑层，数据访问层了，可能代码的形式会有所不同。其实我们

还有更好的方式去实现视图的展现。比如我们可以让视图层与 XML 文件进行绑定的形式，展示器通过生成 XML 文件，然后视图层界面去解析这个 XML 文件中的相应配

置，来完成展示，这样的话，可能视图层不需要处理单独的代码，而且视图可以通过序列化与反序列化的形式，视图的更改或者操作都可以有系统自动完成，而不需要

提供展示器来完成。当然这样的形式仅限简单的应用程序时，通过这样的形式来处理可能会比较灵活。

### 4.6、分析总结

我们上面主要讲述了企业架构中的一个可能的架构，当然这样的架构不是最好的方案，还需要大家多多讨论和交流，我只是根据自己的经验，结合一些项目中的一

些问题进行了总结之后分析这样的架构模式，还不知道在实际的姓名中是否能满足适应性和很好的扩展性，这个还有待检验，我这里并没有给出完整的代码实现，只是

给出一种可能的架构模式的结构，具体的实现代码还需要大家自己完善。

我们总结下我们讲述的内容：

1、分析了四个层次中我们对于设计的考虑和各分层中职责和他们之间的交互，我这里是通过实体层来处理 DTO 的。

2、举例给出部分代码说明每个分层的职责并且分工要明确。

3、表现层中与其他各层之间可能的交互关系。

我想通过上面的可能的模式分析，给大家一个思路，让大家通过这个思路有个更多的思考。

## 五、结束语

本篇主要讲述的是一个架构思路，给出的代码可能不是完全的，只是给了一个各层的大概的结构，因为要实现这样的一个完整的实例 demo，那是需要非常多的代

码和要考虑的因素很多，本篇很多内容并没有涵盖，不过我想这样的方式是不太好的，我后面会尽量将前面讲过的内容，整理出来，以后讲述实例的时候尽量就是一个

可以运行的 demo 实例。

# 六、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

# 七、下篇预告

下一篇我们将会开始讲解系统架构中的分层-上-中-下进行相关讨论及设计方案分析，这些都是本人在工作中的经验总结，由于本人能力有限，不足之处，

还请大家多多指出。希望大家持续关注！