第3章

三层体系架构

3.1 软件体系结构简介

软件体系结构是具有一定形式的结构化元素，即构件的集合，包括处理构件、数据构件和连接构件。处理构件负责对数据进行加工，数据构件是被加工的信息，连接构件把体系结构的不同部分组合连接起来。这一定义注重区分处理构件、数据构件和连接构件，这一方法在其他的定义和方法中基本上得到保持。

20 世纪 80 年代中期出现了 Client/Server 分布式计算结构，应用程序的处理在客户器和服务器之间分担；请求通常被关系型数据库处理，PC 在接收到被处理的数据后实现显示和业务逻辑；系统支持模块化开发，通常有 GUI 界面。Client/Server 结构因为其灵活性得到了极其广泛的应用。但对于大型软件系统而言，这种结构在系统的部署和扩展性方面还是存在不足。

Internet 的发展给传统应用软件的开发带来了深刻的影响。Browser/Server 是 Web 兴起后的一种网络结构，Web 浏览器是客户端最主要的应用软件。这种模式统一了客户端，将系统功能实现的核心部分集中到服务器上，简化了系统的开发、维护和使用。客户机上只要安装一个浏览器如 Netscape Navigator 或 Internet Explorer，服务器安装 SQL Server、Oracle、MySQL 等数据库。浏览器通过 Web Server 同数据库进行数据交互。

基于 Internet 和 Web 的软件和应用系统无疑需要更为开放和灵活的体系结构。随着越来越多的商业系统被搬上 Internet，一种新的、更具生命力的体系结构被广泛采用，这就是所谓的“三层架构”。

3.2 三层体系架构原理

3.2.1 三层架构概述

三层架构即 3-Tier Architecture，通常意义上的三层架构就是将整个业务应用划分为：表示层（Presentation Layer）、业务逻辑层（Application Layer）、数据访问层（Data Access Layer）。区分层次是为了“高内聚低耦合”的思想。在软件体系架构设计中，分层式结构是最常见，也是最重要的一种结构。微软推荐的分层式结构一般分为三层，如图 3-1 所示。从下至上分别为：数据访问层、业务逻辑层（又称为领域层）、表示层。

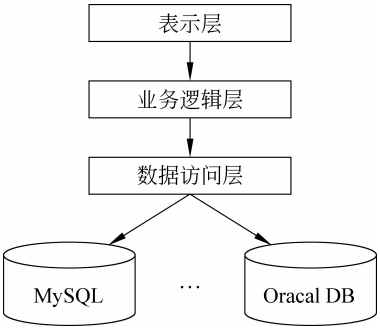

图 3-1　三层架构示意

三个层次中，系统主要功能和业务逻辑都在业务逻辑层进行处理。所谓三层体系结构，是在客户端与数据库之间加入了一个“中间层”，也叫组件层。这里所说的三层体系，不是指物理上的三层，不是简单地放置三台机器就是三层体系结构，也不仅仅有 B/S 应用才是三层体系结构。三层是指逻辑上的三层，即把这三个层放置到一台机器上。三层体系的应用程序将业务规则、数据访问、合法性校验等工作放到了中间层进行处理。通常情况下，客户端不直接与数据库进行交互，而是通过 COM/DCOM 通信与中间层建立连接，再经由中间层与数据库进行交互。

使用三层架构开发有以下优点。

(1) 从开发角度和应用角度看，三层架构比二层架构或单层架构都有更大的优势。三层架构适合团队开发，每人可以有不同的分工，协同工作使效率倍增。开发二层或单层应用程序时，每个开发人员都应对系统有较深的理解，能力要求很高，而开发三层应用程序时，则可以结合多方面的人才，只需少数人对系统有全面了解即可，在一定程度上降低了开发的难度。

(2) 三层架构可以更好地支持分布式计算环境。逻辑层的应用程序可以在多个计算机上运行，充分利用网络的计算功能。分布式计算的潜力巨大，远比升级 CPU 有效。美国人曾利用分布式计算解密，几个月就破解了据称永远都破解不了的密码。

(3) 三层架构的最大优点是它的安全性。用户只能通过逻辑层访问数据层，减少了入口点，把很多危险的系统功能都屏蔽了。

3.2.2　表示层

负责直接跟用户进行交互，一般也就是指系统的界面，用于数据录入、数据显示等。意味着只做与外观显示相关的工作，不属于它的工作不用做，主要表示 Web 方式，也可以表示成 Winform 方式，Web 方式也可以表现成 aspx。如果逻辑层相当强大和完善，无论表现层如何定义和更改，逻辑层都能完善地提供服务。

3.2.3　业务逻辑层

业务逻辑层主要负责对数据层的操作，也就是说把一些数据层的操作进行组合。其主要是针对具体问题的操作，也可以理解成对数据层的操作，对数据业务逻辑处理。如

果说数据层是积木,逻辑层就是对这些积木的搭建。

业务逻辑层(Business Logic Layer)无疑是系统架构中体现核心价值的部分。它的关注点主要集中在业务规则的制定、业务流程的实现等与业务需求有关的系统设计,即它是与系统应对的领域(Domain)逻辑有关,很多时候,也将业务逻辑层称为领域层。例如 Martin Fowler 在 *Patterns of Enterprise Application Architecture* 一书中,将整个架构分为三个主要的层:表示层、领域层和数据访问层。作为领域驱动设计的先驱 Eric Evans,对业务逻辑层做了更细致的划分,细分为应用层与领域层,通过分层进一步将领域逻辑与领域逻辑的解决方案分离。

业务逻辑层在体系架构中的位置很关键,它处于数据访问层与表示层中间,起到了数据交换中承上启下的作用。由于层是一种弱耦合结构,层与层之间的依赖是向下的,底层对于上层而言是"无知"的,改变上层的设计对于其调用的底层而言没有任何影响。如果在分层设计时遵循了面向接口设计的思想,那么这种向下的依赖也应该是一种弱依赖关系。因而在不改变接口定义的前提下,理想的分层式架构,应该是一个支持可抽取、可替换的"抽屉"式架构。正因为如此,业务逻辑层的设计对于一个支持可扩展的架构尤为关键,因为它扮演了两个不同的角色。对于数据访问层而言,它是调用者;对于表示层而言,它是被调用者。依赖与被依赖的关系都纠结在业务逻辑层上,如何实现依赖关系的解耦,则是除了实现业务逻辑之外留给设计师的任务。

3.2.4 数据访问层

数据访问层,顾名思义,就是专门跟数据库进行交互,执行数据的添加、删除、修改和显示等。需要强调的是,所有的数据对象只在这一层被引用,如 System、Data、SqlClient 等,除数据层之外的任何地方都不应该出现这样的引用。主要看数据层里面有没有包含逻辑处理,实际上它的各个函数主要完成各个对数据文件的操作,而不必管其他操作。主要是对原始数据(数据库或者文本文件等存放数据的形式)的操作层,而不是指原始数据。也就是说,是对数据的操作,而不是数据库,具体为业务逻辑层或表示层提供数据服务。

3.2.5 三层架构的辅助类

根据系统设计,常用信息管理系统架构示意如图 3-2 所示。

本架构中的辅助类包括 Model、DBUtility、Common 三个模块中的类。

Model 中包括数据库表对应的模型类。这些模型类中包含了和数据库表中字段相对应的属性,作为实现数据库、DAL、BLL、Web 之间的数据传递载体,实际上实现了整个系统的数据持久化功能。

DBUtility 中包含了针对数据库访问所需的一些最小粒度的、针对不同数据库的通用数据库访问工具类,这些类中的相关方法实现了针对相应数据库的基本的原子数据库访问功能。其中的 DbHelperSQL 类是专门针对 SQL Server 数据库的操作而创建的。

Common 中相关的一些类则实现了针对字符串的操作、加密操作、配置文件操作、Excel 表操作、缓存操作等常用和通用的基本操作,如图 3-3 所示。

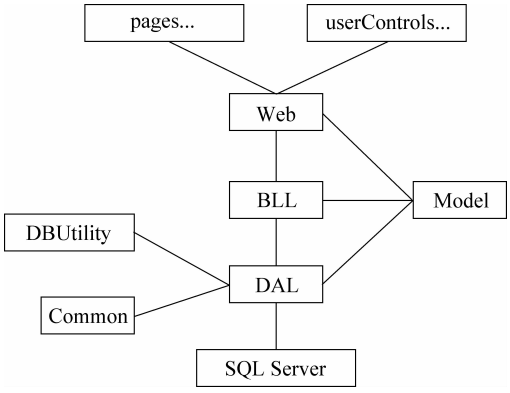

图 3-2　常用信息管理系统架构示意

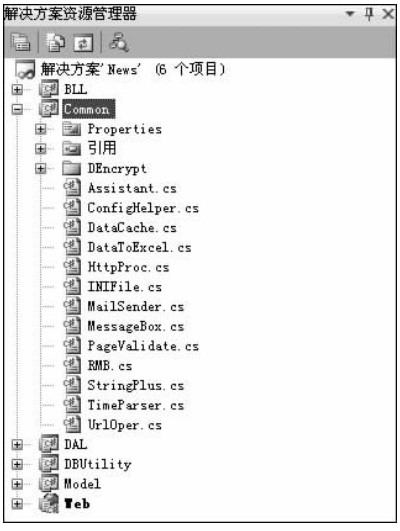

图 3-3　Common 中相关的一些类的操作

例如，其中 StringPlus.cs 文件中的 StringPlus 类就实现了字符串操作的相关方法，具体代码如下：

```
public class StringPlus
{
    ///<summary>
    ///转全角的函数(SBC case)
    ///</summary>
    ///<param name="input"></param>
    ///<returns></returns>
    public static string ToSBC(string input)
    {
        //半角转全角
        char[] c=input.ToCharArray();
        for (int i=0; i <c.Length; i++)
        {
            if (c[i]==32)
            {
                c[i]=(char)12288;
                continue;
            }
            if (c[i] <127)
                c[i]=(char)(c[i]+65248);
        }
        return new string(c);
    }
    ///<summary>
    ///转半角的函数(SBC case)
    ///</summary>
```

```
        ///<param name="input">输入</param>
        ///<returns></returns>
        public static string ToDBC(string input)
        {
            char[] c=input.ToCharArray();
            for (int i=0; i <c.Length; i++)
            {
                if (c[i]==12288)
                {
                    c[i]=(char)32;
                    continue;
                }
                if (c[i] >65280 && c[i] <65375)
                    c[i]=(char)(c[i]-65248);
            }
          return new string(c);
        }
        public static string GetCleanStyle(string StrList, string SplitString)
        {
            string RetrunValue="";
            //如果为空,返回空值
            if (StrList==null)
            {
                RetrunValue="";
            }
            else
            {
                //返回去掉分隔符
                string NewString="";
                NewString=StrList.Replace(SplitString, "");
                RetrunValue=NewString;
            }
            return RetrunValue;
        }
        #endregion
         public static string GetNewStyle(string StrList, string NewStyle, string
         SplitString, out string Error)
        {
            string ReturnValue="";
            //如果输入空值,返回空,并给出错误提示
            if (StrList==null)
            {
                ReturnValue="";
                Error="请输入需要划分格式的字符串";
            }
            else
            {
                //检查传入的字符串长度和样式是否匹配,如果不匹配,则说明使用错误
```