

使用 Asp.Net 构建三层式 Web 应用程序

Part.1 三层式开发介绍、范例程序介绍、数据访问层、需求分析

张子阳

www.tracefact.net

jimmy_dev@163.com

引言

本文是“使用 Asp.Net 构建三层式 Web 应用程序”系列文章的第一部分。在这一系列文章中，我将系统的讲述如何使用 Asp.Net 设计、构建、实现三层式 Web 应用程序。本文的读者应该是有一定 Asp.Net 基础的开发者，同时要求对数据库、C#、Ajax、Web Service 也有一定的了解。这系列文章使用我目前正在使用的一个“个人理财程序”作为范例讲解，这个程序非常小，只有三个表，但麻雀虽小五脏俱全，我主要想利用它为大家阐明一些概念，可能功能并不完备，但对于教程所讨论的主题没有影响，感兴趣的话可以自行扩展它。

古人云：条条大道通罗马。所以，我这里讲述的，只是我个人的三层式 Web 应用程序实现，并不是说只有这一种实现方法，也不能说明这种实现方法是最好的。

这系列文章计划分为五个部分，其中每个部分的内容如下：

- Part 1. 讲解三层式 Web 应用程序的概念，数据访问层的实现方式，“个人理财程序”的程序介绍 以及 需求分析。
- Part 2. 讲解 系统的概要、详细设计，数据库的实现，业务层对象类的实现步骤和方法。
- Part 3. 讲解 数据访问层 和 业务逻辑层 的代码实现。
- Part 4. 讲解 用户界面层 的实现，以及如何使用 ObjectDataSource 调用 业务逻辑层中的对象和方法。
- Part 5. 讲解如何为 Part 4. 中实现的部分加上 Web Service 和 Asp.Net Ajax。

以下是几点说明：

- 本文中，我有时候会说到“用户界面层”，有时候会说到“表现层”，这两个在本系列文章中是一回事。
- 阅读本文前推荐阅读我的另一篇文章 [数据库对象命名参考](#)。
- 本系列文章的 Source Code 和 T-SQL 脚本将会在所有 Part 全部发布后提供下载。

本系列文章使用的开发环境是 VS 2005 + SQL Server 2000，操作系统是 Windows Server 2003 Enterprise Edition。T-SQL 代码我只在 SQL Server 2000 下测试了，如果在 SQL Server 2005 下不能通过，请反馈给我。

三层式开发介绍

分层式开发是一种开发模式，在这种模式中，用户界面层(用户所看到和与其交互的那部分)、业务逻辑层(业务规则(比如本例中每天的开销不能为负数)和业务对象)、以及数据访问层(对数据库进行查询和操作)，从代码的角度来看，是分开的。

这种模式具有很多的优点：

1. 你的代码非常的“干净”。可以试想一下，如果你把提交表单的 ADO.NET 数据库操作全

- 都写到页面的 CodeBehind 文件中，会是多么的凌乱？
2. 更好的可维护性，程序员之间的分工明确，各层之间只需要知道调用接口就可以了，而不需要知道是如何实现的。
 3. 更好的可移植性，可以联想一下 微软的数据访问技术 从 ODBC 到 OLEDB 到 ADO 再到 ADO.Net 1.1 一直到如今的 ADO.NET 2.0，几乎每三年就会有一次变革，采用分层式开发，可以在系统升级的时候更少的受到影响。
 4. 更好的对分布式应用程序的支持。

NOTE: 提到分布式应用程序时常会遇到两个英文单词: Tier 和 Layer，这两个单词的意思翻译过来都是“层”，但是有什么区别呢？老外通常提到 Tier 的时候，指的是物理上分层；提到 Layer 的时候，常指的是逻辑上分层。物理上分层说的简单点，就是用户界面层在一台服务器，业务逻辑层在一台服务器，数据访问层又在另一台服务器(也可以表现层和业务层在同一台服务器，数据访问层在一台服务器，总之三个层不在一台服务器)。而逻辑上分层我现在正在讲述，很容易就想得通：如果物理上分层了，逻辑上也一定分层了；但如果逻辑上分层了，物理上不一定是分层的，可以部署在同一台服务器上，比如我的这个“个人理财程序”。

为了给大家一个更生动的认识，我用 Visio 画了个图给大家看看：

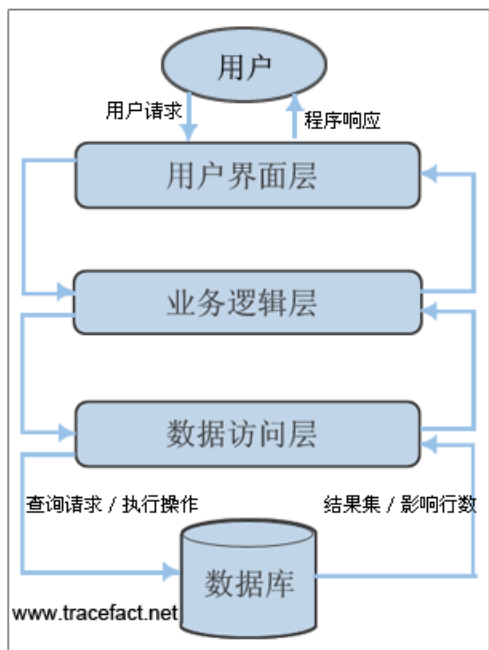


图 1. 三层式程序结构图

这张图描述了这个应用程序中数据流动的大致方向。

请大家从用户和左边的箭头看起：

1. 用户浏览网页时，首先面对的是用户界面层或者说表现层，如果用户进行一个对数据库查询的操作，请求首先会发送到业务逻辑层。
2. 业务逻辑层对用户提交的数据进行校验，如果有问题，将拒绝用户请求并给出错误提示；如果没有问题，业务逻辑层将用户请求递交给数据访问层。
3. 数据访问层充当业务逻辑层与数据库之间的一个桥梁，把请求递交给数据库，不应该在

这一层再去做一些数据校验的工作，来自业务逻辑层的数据应该被认为是无误的，这层的代码相对于业务逻辑层来说是很清爽的。

4. 数据库进行查询后将结果集返回给数据访问层，接着再返回给业务逻辑层，最后呈现给用户。这是一个典型的冒泡过程。在这个过程中，任何没有 Catch 的异常都会抛出，冒泡到用户界面层，这也就是为什么你访问 aspx 的页面，看到的却是来自数据库的“Can't connect to database”的原因。

个人理财 Web 程序介绍

我是一个不会理财的人，在过去两年零一个月的时间，一毛钱都没有攒下，一个人的时候，我时常思索为什么我的钱总是来也匆匆去也匆匆，在现在老婆房子女朋友都没有的三无情况下，这样继续下去将会给我的人生带来深远的影响。

沉默了许久之后，我终于觉悟了... ..我想做的第一件事，就是解决长期困扰我的头号问题 -- 我的钱都跑哪儿去了？于是，这个个人理财程序便应运而生了。

其实如果只是我一个人使用的话，只需要一个表，就足以构建这个应用程序了。我给这个表起名叫：DailyCost，用来记录每天的开销/收入项目，表的字段分别是：

CostId: 自动编号，主键。
Type: 0, 表示开销; 1, 表示收入
Purpost: 开销用途/收入来源
Amount: 数额
CostDate: 开销日期

但是，这么好的程序(NOTE: 和我一样不会理财的人应该也不少吧?)，只有我一个人用简直太浪费了，让我们来对它进行了一下扩展，让更多的人可以使用吧。这样，就很有必要再加一个 User 表，记录使用此系统的用户，这个表应该包含如下字段：

UserId: 自动编号，主键。
Name: 用户的名字
BirthDay: 生日
Phone: 用户的电话

此时，应该修改 DailyCost 表，以实现参照完整性(NOTE: 也叫外键约束)。给 DailyCost 表再加一个字段：FKUserId，此条消费记录是属于哪位用户的。

在现在这个苦力都拿手机的时代，人们的 Phone 是会有很多的，比如手机一个，办公室一个，家里一个。所以，我们应该把 Phone 字段抽象出来，形成与 User 表的一对多关系。

建立 Phone 表，字段如下：
PhoneId: 自动编号，主键。
Number: 号码
Type: 0, 手机; 1, 家里; 2, 公司; 3, 其他

FKUserId: 外键, 此 Phone 是哪位用户的

OK, 这部分就先介绍到这里, 在后面的 概要/详细 设计中会再次讲解。

NOTE: 概要设计和详细设计我合并了, 因为这个应用程序比较小, 合并到一起我想大家已经可以看明白, 分开讲解可能会显得文章过于繁琐。

实现数据访问层的不同方式

与用户界面揉在一起的数据访问层

这就是上面提到过的“初学者访问层”, 将 ADO.NET 代码揉到 CodeBehind 的事件处理中去, 这种方法几乎没有可维护性, 对数据库的任何改动, 比如增减一个字段, 或者给某个存储过程重新命名, 都要导致修改大量的相关文件。另外, 这种方法还有一个致命的缺点: 无法实现代码重用。其实这种方法不应该称作“层”, 但是为了文章的完备性, 我还是在这里把它提出来。

用这种方式写的代码通常都是这样的:

```
protected void Page_Load(object sender, EventArgs e)
{
    if (!Page.IsPostBack)
    {
        string sqlText = @"SELECT Purpose, Amount,
            Type FROM DailyCost WHERE CostId = 1";

        using (SqlConnection conn =
            new SqlConnection(ConfigurationManager.ConnectionStrings["JimmyCost"]
                .ConnectionString))
        {
            using (SqlCommand myCommand = new SqlCommand(sqlText, conn))
            {
                conn.Open();
                using (SqlDataReader myReader = myCommand.ExecuteReader())
                {
                    if (myReader.Read())
                    {
                        txtPurpose.Text = myReader.GetString(0);
                        txtAmount.Text = myReader.GetString(1);
                        txtType.Text = myReader.GetString(2);
                    }
                    myReader.Close();
                }
            }
            conn.Close();
        }
    }
}
```

```

    }
    }
}

protected void btnSave_Click(object sender, EventArgs e)
{
    string sqlText = @"UPDATE DailyCost SET Purpose='{0}',
    Amount='{1}', Type = '{2}' WHERE Id = 1";
    using (SqlConnection conn =
        new SqlConnection(ConfigurationManager.ConnectionStrings["JimmyCost"]
            .ConnectionString))
    {
        string sql = String.Format(sqlText,
            txtPurpose.Text, txtAmount.Text, txtType.Text);
        SqlCommand cmd = new SqlCommand(sql, conn);
        conn.Open();
        cmd.ExecuteNonQuery();
        conn.Close();
    }
}

```

使用 SqlDataSource 作为数据访问层

在这里我推荐给大家一本好书，由电子工业出版社出版的，奚江华 写作的《圣殿祭祀的 ASP.NET 2.0 开发详解》。在这本书中，作者在第 11 章 — “新一代数据访问方式 DataSource 控件”中详细讨论了 SqlDataSource 的种种特性，其中，比较重要的就是使用代码后置的方法动态的创建和配置 SqlDataSource 控件，并提供对页面数据控件的绑定。这就提供了一种新的实现数据访问层的思路 - 使用 SqlDataSource 作为数据访问层。

在这里不得不说明一点，ObjectDataSource 其实就是完全自定义的 SqlDataSource 控件，想想看你是如何使用 SqlDataSource 的？我想你一定在前端页面通过拖拽控件的方式使用过 SqlDataSource ，也一定注意到 SqlDataSource 的几个重要属性，SelectCommand、UpdateCommand 等，在使用 SqlDataSource 时，向导自动为你生成 Sql 语句(也可以自己写)，然后显示在.aspx 文件中，如果你实现了 UpdateCommand、InsertCommand (SelectCommand 是必须要实现的)，SqlDataSource 也就具有了相应的功能。而 ObjectDataSource 是如何运作的呢？它要求你自己写全部的 Code 以实现 Command，区别就是，你可以把这些 Command 全部封装到类或者说业务逻辑层中去。

好了，说了这么多，让我们看看使用 SqlDataSource 作为数据访问层的典型页面吧。

```

<asp:SqlDataSource
ID="SqlDataSource1" runat="server"
ConnectionString="<%$ ConnectionStrings:JimmyConn %>"
DeleteCommand="DELETE FROM [DailyCost] WHERE [CostId] = @original_Id"

```

```

InsertCommand="INSERT INTO [DailyCost] ([Purpose], [Type],
    [Amount], [CostDate]) VALUES (@Purpose,
    @Type, @Amount, @CostDate)"
SelectCommand="SELECT * FROM [DailyCost]"
UpdateCommand="UPDATE [DailyCost] SET [Purpose] = @Purpose,
    [Type] = @Type, [Amount] = @Amount,
    [CostDate] = @CostDate WHERE [CostId] = @original_Id"
OldValuesParameterFormatString="original_{0}"
>
<DeleteParameters>
    <asp:Parameter Name="original_Id" Type="Int32" />
</DeleteParameters>
<UpdateParameters>
    <asp:Parameter Name="Purpose" Type="String" />
    <asp:Parameter Name="Type" Type="Bool" />
    <asp:Parameter Name="Amount" Type="Decimal" />
    <asp:Parameter Name="CostDate" Type="DateTime" />
    <asp:Parameter Name="original_Id" Type="Int32" />
</UpdateParameters>
<InsertParameters>
    <asp:Parameter Name="Purpose" Type="String" />
    <asp:Parameter Name="Type" Type=" Bool " />
    <asp:Parameter Name="Amount" Type="Decimal" />
    <asp:Parameter Name="CostDate" Type="DateTime" />
</InsertParameters></asp:SqlDataSource>

```

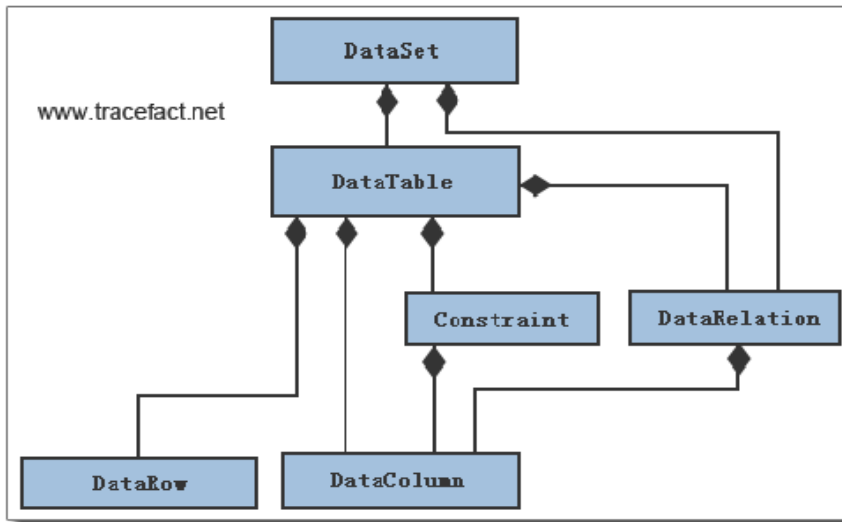
很明显就可以看到，不仅关于数据访问的 Sql 语句直接写到了 .aspx 页面中，而且代码臃肿不堪。尽管你可以使用代码后置的方式去实现这些，但代价就是降低了开发效率，因为手写比你拖动控件、使用向导要慢得太多。

使用 TableAdapter 和 强类型的 DataSet

VS 2005 提供强类型的 DataSet，可以通过 VS2005 向导 来创建，熟练的话，是创建数据访问层最快的一种方式，微软官方网站 www.asp.net，有一个关于数据访问的系列教程，目前出到快 70 章了(NOTE: 我大概算了一下，可以出本 900 页的书了)，仍在不断更新中，这个教程使用的就是这种模型。

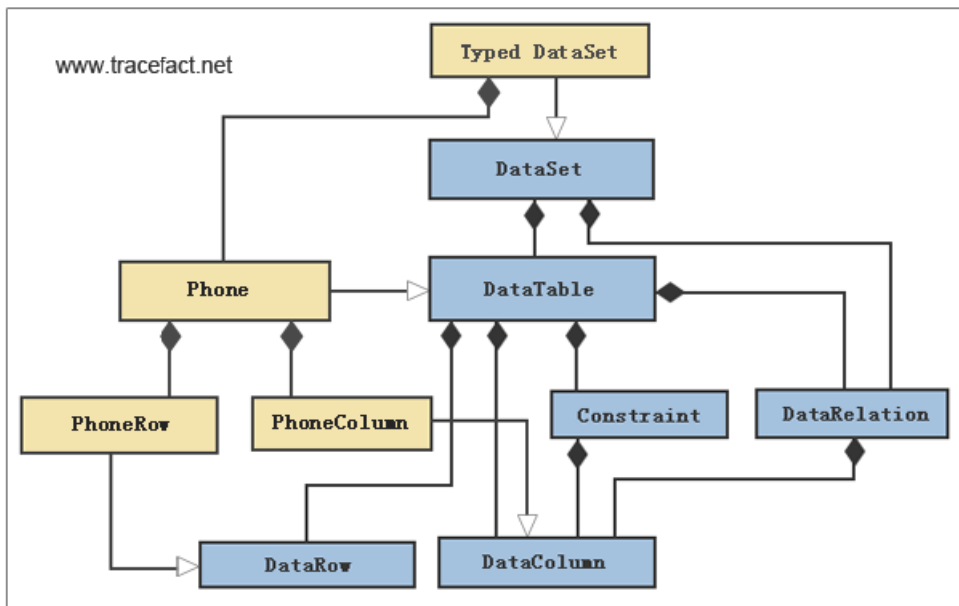
这里我仅简要说明一下什么是强类型的 DataSet，和普通的 DataSet 有什么区别，至于如何创建强类型的 DataSet，需要一个独立的章节来介绍，我以后再发文章补上。

强类型 DataSet (Typed DataSet)并不是 .NET 框架直接提供的一系列类，而是从 DataSet 类继承而来的，下图先给大家演示了传统的 DataSet 和它包含的各个元素之间的关系。



这里黑色的菱形箭头表示的是合成的方向，下面这幅图中，空心的三角，表示的是泛化的方向。合成，说通俗一点，或者用数据库的术语来说(NOTE: 用在这里并不恰当，纯粹为了解说方便)，就是多对一关系，拿这幅图来说，就是好多个 DataRow 合成了 DataTable，同时 DataTable 还可以有 DataColumn, Constraint 和 DataRelation; 同时，很多个 DataColumn 又合成了 Constraint 和 DataRelation。最后，好多个 DataTable 和 DataRelation 合成了 DataSet。

相比之下，强类型 DataSet 是从这些类派生出来的。



这幅图该如何解读我就不详细解释了，留给大家一点思考的空间。

那又为什么称它们为“强类型”DataSet 呢？因为在强类型 DataSet 中，是使用一种类型安全的方式来使用其中的每一个对象，光是说术语很难懂什么叫类型安全的方式，让我们看下面的代码范例吧。

我们先看一段使用传统 DataSet 的代码。

```
// 前面省略部分代码
SqlDataAdapter daDailyCost = new SqlDataAdapter("Select * From DailyCost", conn);

DataSet dataSet = new DataSet;

daDailyCost.Fill(dataSet, "ShowCost");

Console.WriteLine(dataSet.Tables["ShowCost"].Rows[0]["CostId"].ToString());
```

可以发现，我们在获取一个字段的值时，需要使用 DataSet 的索引器，逐步获得 DataSet 层次结构的每一个片段，直到最后抵达行级，在这个过程中，任何的拼写错误都会导致抛出异常。

现在再看看 强类型 DataSet 是如何运作的。

```
//前面省略部分代码
SqlDataAdapter daDailyCost = new SqlDataAdapter("Select * From DailyCost",conn);

//DailyCostTD 是实现了的强类型 dataSet，至于如何实现，我以后会另写文章
DailyCostTD dataSet = new DailyCostTD();

daDailyCost.Fill(dataSet,"ShowCost");

Console.WriteLine(dataSet.Customer[0].Purpose);
```

在这里，大家似乎觉得这两个没太大区别，无非就是在 Console.WriteLine 的时候少打几个字而已，其实不然，在你用强类型 DataSet 的时候，VS2005 会提供智能提示(NOTE：就是打个 i，int 就显示出来了，你只要拍下空格就好了)，这样，就大大降低了编写时发生手误的机会。

既然强类型 DataSet 这么好用，为什么不干脆使用它作为数据访问层呢？前面我也提到过，使用类型化 DataSet 是创建数据访问层最快的方式，你只需要建立一个.xsd 数据集文件，然后拖拽控件，再使用向导设置一下基本就可以 Run 了。但是，代价就是几乎没有扩展性，因为代码都是自动生成的，你很难去修改它(NOTE：其实还是有办法可以改的，你可以通过写部分类的方式去扩展它)，另外，重用性也比较差。

使用封装的 ADO.NET 数据访问层类

与“初学者访问层”相同的地方是：这种方式也是采用的全手写 ADO.NET 代码来实现的；不同的是，将所有对于数据库的操作封装到了一个类中，这个类是高度整合的，你甚至可以不加改动将它用在其它的项目中。

我能想到这种方式，当然很多人也能想到，所以，网上的开源站点及一些个人站点已经提供了这个封装好的类的下载，其中大多是静态类，因为类作者水平高低不均，所以类的质量也是良莠不齐。另外，几乎所有这个类都约定俗成般起了同一个名字，叫做 -- SqlHelper 。

这里有一个比较不错的老外写的 [SqlHelper类](#)，感兴趣的可以参考一下。

使用新一代的 LINQ

还有一种方式就是使用 LINQ，很多人可能还是第一次听说这个名词，那么我就稍微的介绍一下：

简单来说，Linq 就是帮助编译器理解和实现内存中保存的对象集合的一组特性。这样说可能比较绕口，Linq 中有一个组件，称做 Linq To Sql，它提供了一个在运行时将关系数据库数据处理成对象的底层机制，并且还不丧失关系数据库的查询功能。它通过将面向语的查询翻译成数据库可执行的查询语句，再将语句提交给数据库，然后把数据库返回的结果集映射成开发者定义的对象来完成。

上面的说法如果没有实际动手体会一下，或者看一些范例，相信不是太好理解。在我的个人博客 jimmyzhang.cnblogs.com 上已经陆续发布了来自微软开发团队的ScottGu的“Linq To Sql 介绍”系列文章，有兴趣的朋友可以去看一看。

不使用 Linq 的主要原因是它现在还处于 Beta 版本，正式发布时可能还会有改进。另外，目前的主机大多都只支持到 .Net Framework 2.0，所以使用 LINQ 尚为时过早。

使用自定义的业务逻辑层对象

终于，我们的主角登场了。很多情况下，业务对象只是一个由其他对象继承的简单的类，也可以实现一些接口让它变得更好用一些。在我们的这个系统中，Phone 对象像下面这样：

```
public class Phone
{
    private int phoneid = -1;
    private string number;
    private PhoneType type;           // PhoneType 是一个枚举

    public string Number{
        get{return number;}
        set{number = value;}
    }
}

//以下代码省略
```

在实践中，通常业务对象只包含数据，而关于对象操作的方法则封装到业务逻辑中去。

好了，数据访问层的实现方法至此告一段落，下面让我们开始进行 需求分析 吧。

需求分析

做任何开发之前，需求分析都是很重要的步骤，如果需求都没有搞好，就等于没有回答用户需要什么解决方案的问题，如果在需求分析不明确的情况下冒然进行开发，结果往往是苦战两个月，做出来的产品却不是用户想要的。另外，需求分析并不是一次成型的，在这个阶段，由于还没有深入到概要设计或者详细设计中去，对系统的理解可能并不完整，所以会时常回过头来修订需求分析。

由于这个“个人理财程序”的用户和开发人员都是我自己，所以需求分析可以认为只是打字而已。

在这个 Web 应用程序中，我们要实现的功能主要有这些：

1. 要求可以添加、编辑、删除用户
2. 要求可以添加、编辑、删除用户电话
3. 要求可以添加、编辑、删除每日的开销项目
4. 要求可以删除 某日、某月 的全部开销
5. 要求可以查看某天、某月、某年的花费报表
6. 要求可以统计每天、每月、每年各开销了多少钱
7. 要求可以统计历史花钱最高的日期和开销/收益的数额
8. 要求可以统计历史花钱最低的日期和开销/收益的数额
9. 要求可以统计自系统运行以来总共开销了多少钱
10. 要求可以统计自系统运行以来总共收益了多少钱
11. 要求可以查看某个用户所有的联系方式

总结

本文是使用 Asp.Net 构建三层 Web 应用程序的第一部分。

我们首先了解了什么是三层式开发模式，接着提出了一个很现实的需要解决的问题，并对这个问题做了一些简单的分析。

然后，我们花了很大的篇幅，讲解实现数据访问层的各种方式，并简要介绍了我们即将采用的方法。

最后，我们对这个“个人理财程序”做了一下简单需求分析。

希望这篇文章能给你带来帮助。