循序渐进开发 Web 项目
操作说明书

版本：2.0

编制人：伍华聪

序号	修改人	修改日期	修改后版本	修改说明
1	伍华聪	2013-09-30	V1.0	正式版
2	伍华聪	2014-07-02	V2.0	根据最新功能进行完善
3	伍华聪	2014-12-03	V3.0	增加存储过程、Web 项目设计分析等

目　录

1. 引言

1.1. 背景

前几年，我一直在做 Web 项目，使用 ASP.NET 的 WebForm 方式开发了几个较为大型的行业管理系统，经过不断的完善和技术积累，也逐渐形成了一个标准的信息管理系统的 Web 开发框架，这个是早期技术的 Web 开发框架，包含了权限管理模块、菜单管理模块、字典管理以及行业管理业务模块等，同时开发了一些 Web 的相关控件，包括 Web 分页控件、Web 查询控件、Web 文件上传等控件，同时也已经结合代码生成工具 Database2Sharp 进行大部分的代码生成，包括 Web 前端的一些界面。

在原先的 Web 框架里面，主要是采用 FrameSet 的原始方式进行布局，很多内容依靠 Javascript 类库进行操作，小部分采用了 EasyUI 的一些特性，总体来说，是比较传统的一种框架模式，这个框架里面我已经集成了用户角色等权限方面的管理和控制、菜单管理、字典管理、业务流程审批管理等模块，因此对开发常规的行业应用有着比较快的开发效率，不过缺点也比较明显，就是在多浏览器支持方面，没有做的很好，框架里面采用的布局、样式及技术等方面不够统一，不够新颖，但即使这样，这套框架也顺利用来开发了几套很大规模的行业应用系统了。

随着 Web 前端技术的发展，MVC 的技术也已经渐趋成熟，JQuery 等应用技术已经是遍地开花，应用非常广泛，有很多组件模块可以复用，界面的美观以及易用方面都有很大的提高。为了引入更新、更强大的 Web 技术，我对 Web 开发框架整体进行了改造，使用基于 **MVC + EasyUI + Enterprise Library** 等相关的技术对原有的 Web 开发框架进行了升级，并引入了功能强大且流行很广的 EasyUI 界面组件，以及 Uploadify 文件上传组件、LODOP 打印组件、CKEditor 富文本编辑控件、Tags-Input 标签录入控件、HighCharts 图表展示控件、Word/Excel 文档导出组件等，对《Web 开发框架》整体进行加强和完善，该《Web 开发框架》是我们经过多年的项目积累，吸收众多框架产品的前沿思路，以及客户的宝贵意见，反复提炼优化而成的。

1.2. 编写目的

本文档主要介绍如何在《Web 开发框架》的基础上循序渐进进行项目的增量开发，并使得开发工程师了解整个框架的各部分内容及关联关系，以便能够更全面的、更真实了解《Web 开发框架》的相关特点。

1.3. 参考资料

序号	名称	版本/日期	来源
1	《Web 开发框架-架构设计说明书.doc》		内部
2	《Winform 开发框架-架构设计说明书.doc》		内部
3			内部

1.4. 术语缩写及约定

1 在本文件中出现的“Web开发框架”一词，除非特别说明，均指基于**MVC + EasyUI + Enterprise Library**等相关的技术研制的最新Web开发框架。

2 在本文安装.NET框架中，除非特别说明，均指 **.NET 4.5** 框架。

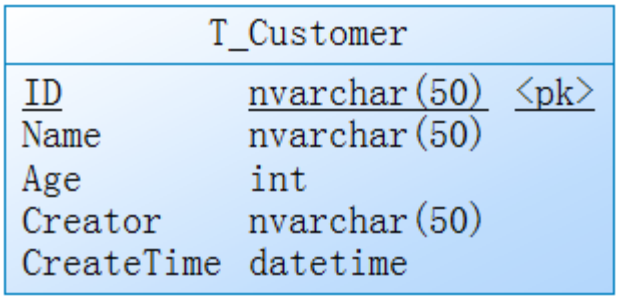
3 本文的开发环境为Visual Studio 2013，基于 MVC5 的Web开发技术。

2. 基础模块的设计分析

2.1. 数据库表设计

俗话说万层高楼从底起，开发应用项目，数据库的设计很重要，它可能是业务对象，业务流程的综合设计，好的数据库设计可以减少后期的重复返工，提高开发效率。

我们以一个简单的数据库表进行设计讨论，一步步分析其中的关系。



1）表和字段名称

一般表名称，根据不同的业务关系，我们可以使用不同的前缀进行区分，使用前缀，可以非常方便区分不同的业务表，如我自己一般基础表使用 “TB_” 定义前缀，权限系统表使用"T_ACL_"定义前缀，工作流表使用"TBAPP_"，业务表使用"T_"等，这样对于区分不同的业务，方便管理很有好处。

字段名称方面，我们可以约定一些规则，如约定主键使用 ID；一般来说，ID 作为主键，可以使用自增长的整形字段，也可以使用 GUID 的字符型字段，如果为了方便兼容不同的数据库且方便迁移或者开发基于网络方面的应用，我建议还是使用 GUID 的字符型字段，使用这种类型的字段，我们从创建数据的时候，就可以知道这个记录的主键，对于我们维护父子表等关系非常有利。

字段的命名，建议以简单为主，如客户名称，直接使用 Name 来命名即可，不需要使用 CustomerName 这样啰嗦的名称，如多个单词组合的话命名采用 Camel 的格式。

由于如果采用字符型的 ID 主键，那么我们如果需要正确排序的时候，可能需要增加一个 CreateTime 的日期类型，方便我们根据日期进行排序。

如果这个表还有一个外键的引用，建议统一命名标准，我一般使用 "表名称_ID" 这样的名称，如 User_ID、Contact_ID 等相似的名称作为外键，不需要表的前缀。

2）数据库的模型设计

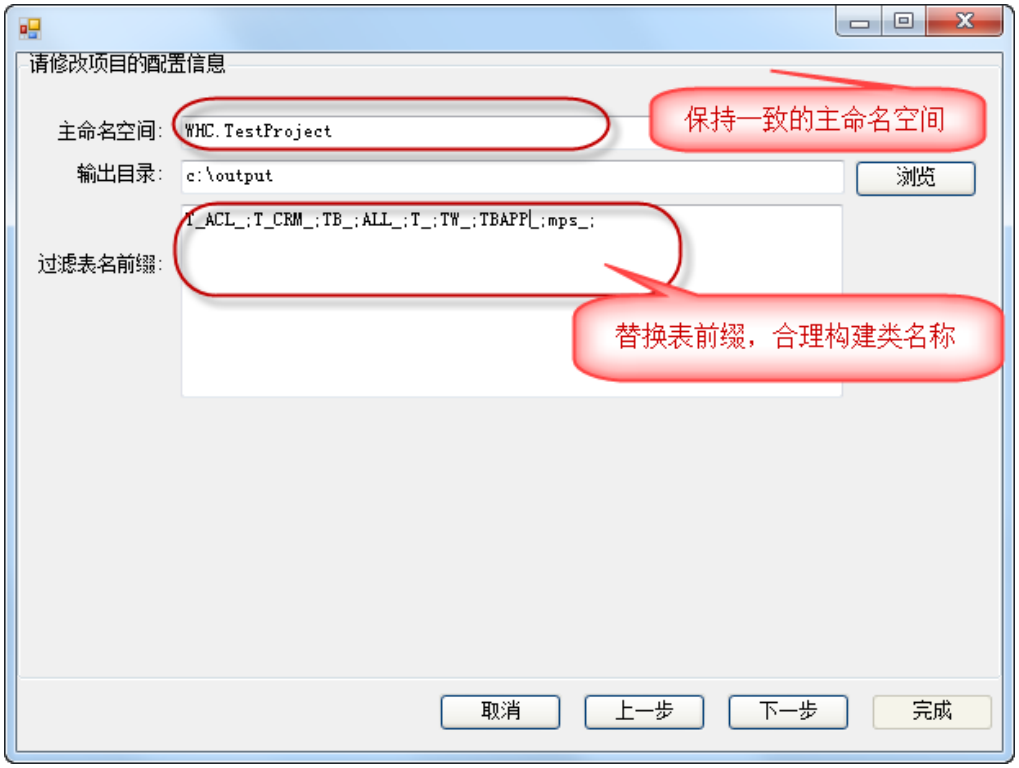
数据库的模型设计，我们建议在第三方的数据库设计工具上进行设计，如 PowerDesigner 这样的设计工具，使用工具设计数据库有很多好处，一个是可以高效率进行调整，二是根据需要生成不同的数据库类型 Sql 语句，三是可以全局了解各个表之间的关系等等。使用 PowerDesigner 这样的数据库设计工具，能够在很大程度上提高我们数据库的设计效率。

如果要开发基于 Sqlite 的数据库应用，建议先在 SqlServer 上进行开发，然后在 DAL 层

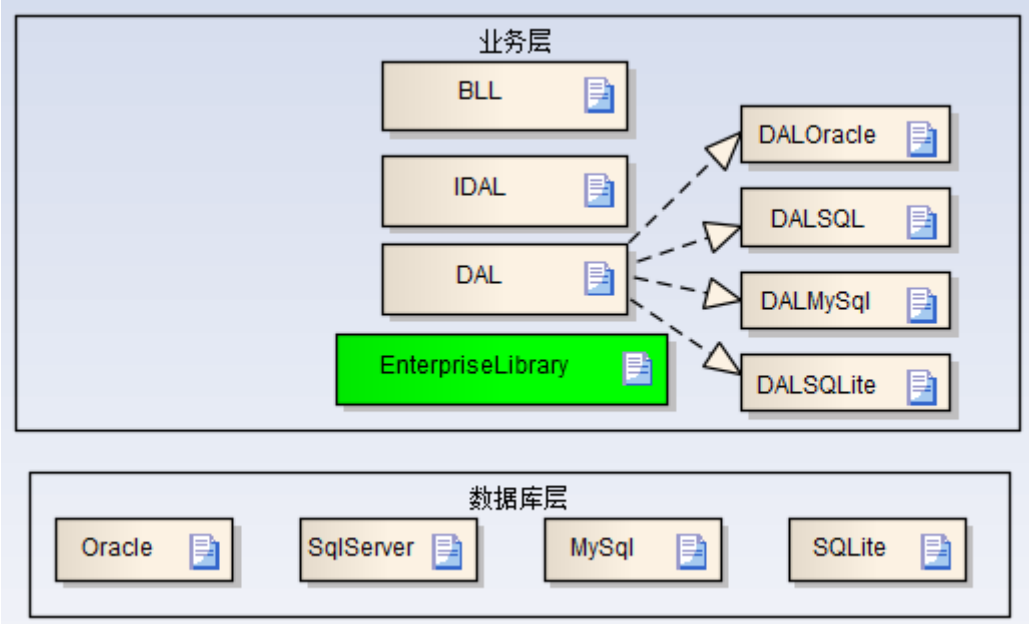
增加一个 Sqlite 的数据访问层，并修改配置文件指向就可以了（基本上和 SqlServer 差不多）。

2.2. 项目框架的生成

设计好数据库后，我们通过代码生成工具进行整个项目框架的生成，这样对于我们在开发新项目上有很好的好处，里面的项目层级、DLL 的 引用关系，已经处理好了，这样对我们非常方便。不过大多数情况下，我们都是增量开发较多，也就是我们可能前面已经完成了一些其他业务的开发，可能新增一个两个表，或者一批业务表的处理，这样也没关系，我们把新生成的代码复制到项目即可，由于项目生成的时候，指定了主命名空间和相关的表前缀，这样我们生成后的代码就方便阅读很多，减少累赘和出错的机会。

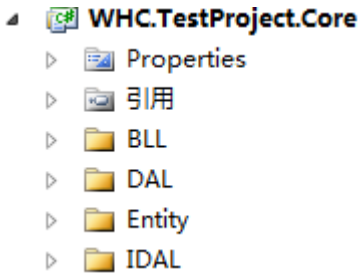


Web 开发框架(包括 Winform 项目)，常见的分层模式，可以分为 UI 层、BLL 层、DAL 层、IDAL 层、Entity 层、公用类库层等等。



这个分层，在 Web 项目或者 WInform 项目（包括 WPF 项目），这些分层都是可以重用的，这样我们就不用重复处理界面一下的逻辑，针对性的开发我们需要的界面层即可。

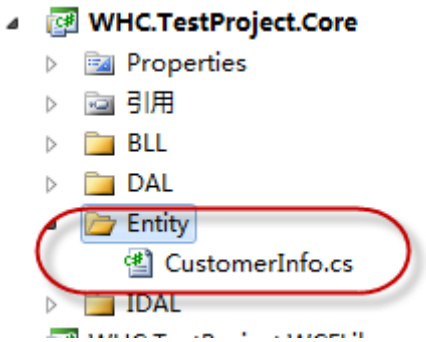
DAL 层根据不同的需要，扩展支持不同的数据库类型，每个数据库类型，对应一个数据库访问实现层即可，它们实现 IDAL 层的接口，称之为数据库访问接口实现层。



2.3. 项目实体层代码分析

通过代码工具，我们已经可以完整生成基础的项目框架了，下面我们来分析下项目的源码，从而知道整个框架的架构和代码的层次是如何的。

刚才我们看到，生成的项目里面，已经包含了实体类，我们以开篇介绍的一个表生成的代码来进行研究分析。



其中我们看到下面的代码，里面使用了基类 **BaseEntity**，这个是所有生成的实体类的基类，基类 BaseEntity 只是一个实体类的声明，没有什么属性，使用这个实体类基类，只是为了整个框架更好管理和控制。BaseEntity 来源于公用类库，已经封装在里面了。

```
/// <summary>
/// 客户信息
/// </summary>
[DataContract]
public class CustomerInfo : BaseEntity
{
```

另外，我们看到，实体类有注释，这些注释来自数据库的备注信息，包括字段的注释也是来自数据库的备注说明信息。

还有类的定义里面，还看到了**[DataContract]** 的标签，以及类的属性**[DataMember]**，这个是 WCF 技术里面传输数据的协议声明，我们目前开发的应用，一般都是基于.NET4.0 的了，因此包含这个属性方便我们在开发网络版项目的时候用到，一般情况下忽略即可。

我们继续看看实体类的其他部分代码：

```
        #region Field Members

        private string m_ID = System.Guid.NewGuid().ToString(); //编号
        private string m_Name; //姓名
        private int m_Age = 0; //年龄
        private string m_Creator; //创建人
        private DateTime m_CreateTime; //创建时间

        #endregion
```

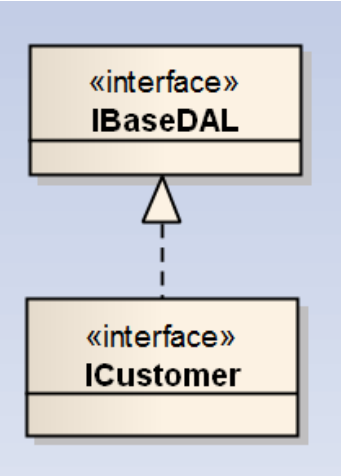
我们看到，对于字符型的 ID 主键字段，代码生成的时候，已经自动添加默认属性值（GUID：System.Guid.NewGuid().ToString())的了，这样我们创建实体类的时候，这个 ID 的值就已经生成了。

2.4. 数据访问接口的定义

上面我们分析了实体类的定义，本节继续分析其他部分的内容，如数据访问接口成的定义如下所示。

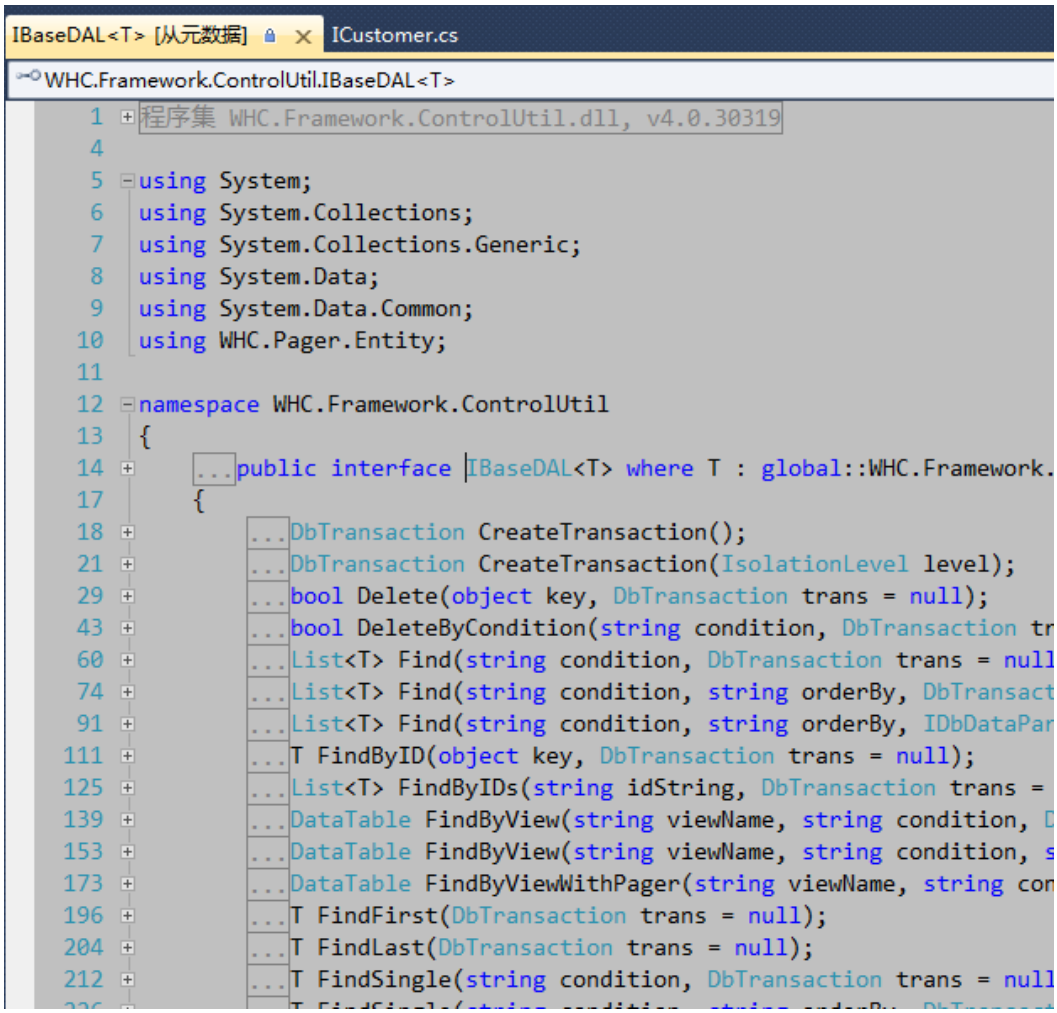
```
namespace WHC.TestProject.IDAL
{
    /// <summary>
    /// 客户信息
    /// </summary>
    public interface ICustomer : IBaseDAL<CustomerInfo>
    {
    }
}
```

这里面的代码很简单，没有多余的代码行，那么里面究竟发生了什么呢，其中的 IBaseDAL 又是什么定义呢？



其实，IBaseDAL 就是定义了很多我们开发用到的基础接口，如标准的增删改查，以及衍生出来的一些其他接口，如分页查询，条件查询等接口内容。这个 ICustomer 就是用来定义一些除了标准接口不能实现外的业务接口。

IBaseDAL 通过传入一个实体类，从而方便给基类接口提供强类型的数据类型指定，提高我们的开发效率，减少出错的机会。我们可以在 VS 里面查看 IBaseDAL 的定义，如下所示：

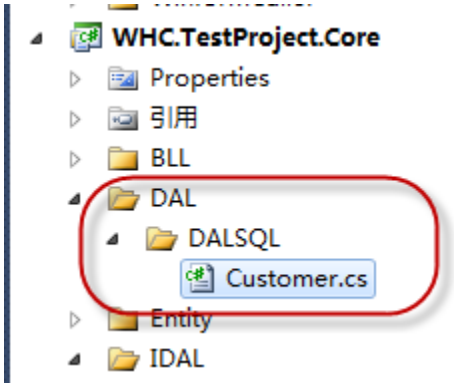


可以看到里面很多相关的接口定义，有返回实体 T 的，也有返回 List<T>的，还有 DataTable 类型等等，这些基础接口，经过我们多个项目的应用实践，已逐步稳定并能够提供很好的接口支持，方便我们快速调用处理。

即使我们在没有实现任何业务接口的情况下，仅仅利用标准的基类 API，也基本上能够完成绝大多数的数据操作功能了。

2.5. 数据访问接口实现类的定义

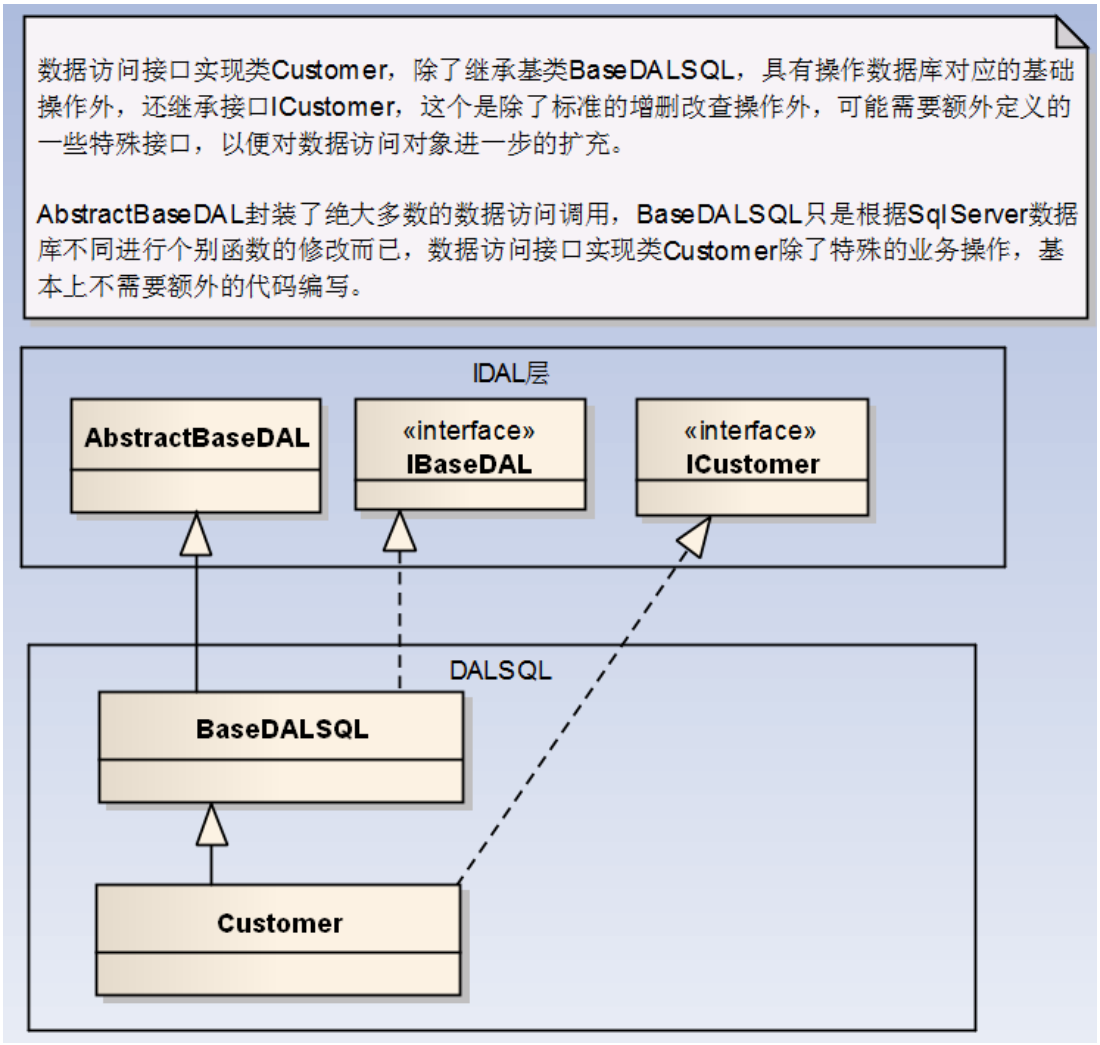
我们分析完 IDAL 的数据访问接口成的定义后，继续了解一下，如何基于这个接口进行访问层的实现设计的。数据访问的实现层在项目中的位置如下所示（以基于 SqlServer 的 DALSQL 层进行分析）。



它的类代码定义如下所示。

```
namespace WHC.TestProject.DALSQL
{
    /// <summary>
    /// 客户信息
    /// </summary>
    public class Customer : BaseDALSQL<CustomerInfo>, ICustomer
    {
```

数据访问接口实现层和接口定义层一样，都有一个基类，如基于 SqlServer 实现的基类为 BaseDALSQL，这个基于 SqlServer 的数据访问基类，它也是继承自一个超级基类（大多数的实现在这里）AbstractBaseDAL。他们之间的继承关系如下所示。



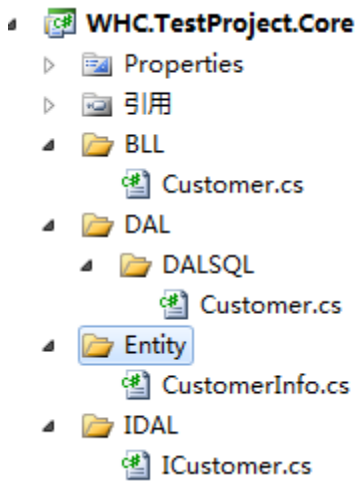
而我们刚才在项目工程的图里面看到，BaseDALSQL、IBaseDAL、AbstractBaseDAL 这些类库由于具有很大的通用性，为了减少在不同的项目中进行复制导致维护问题，因此我们全部把这些经常使用到的基类或者接口，抽取到一个独立的类库里面，为了和普通的 DotNET 公用类库命名进行区分（WHC.Framework.Commons），我们把它命名为 WHC.Framework.ControlUtil。

BaseDALSQL 基类的定义如下所示。

```
namespace WHC.Framework.ControlUtil
{
    public abstract class BaseDALSQL<T> : AbstractBaseDAL<T>, IBaseDAL<T> where T : global::WHC.Framework.ControlUtil.BaseEntity, new()
    {
        public BaseDALSQL();
        public BaseDALSQL(string tableName, string primaryKey);

        public override T FindFirst(DbTransaction trans = null);
        public override T FindLast(DbTransaction trans = null);
        public override DataTable FindToDataTable(string condition, PagerInfo info, string fieldToSort, bool desc, DbTransaction trans = null);
        public override List<T> FindWithPager(string condition, PagerInfo info, string fieldToSort, bool desc, DbTransaction trans = null);
        public override DataTable GetTopResult(string sql, int count, string orderBy, DbTransaction trans = null);
        public override int Insert2(Hashtable recordField, string targetTable, DbTransaction trans);
        public override DataTable SqlTable(string sql, DbTransaction trans = null);
        public override DataTable SqlTable(string sql, DbParameter[] parameters, DbTransaction trans = null);
        public override bool TestConnection(string connectionString);
    }
```

这样做的好处是，在所有的模块里面，避免复制导致的版本维护问题，同时也减少代码的重复生成，增量生成的全部代码，可以一次性复制到整个项目工程里面，而不会导致基础类库的替换，因为这些基类不在生成目录里面，所有生成的类文件，都是和业务表相关的，如下所示。



具体的数据访问实现类（如 Customer），它把数据库信息转换为实体类，有一个函数，在代码生成的时候已经生成；同时在把实体类的属性保存到数据库也有一个类似 CRM 的映射关系，从而实现可空的字段获取和更新操作。

```
/// <summary>
/// 将 DataReader 的属性值转化为实体类的属性值，返回实体类
/// </summary>
/// <param name="dr">有效的 DataReader 对象</param>
/// <returns>实体类对象</returns>
protected override CustomerInfo DataReaderToEntity(IDataReader dataReader)
{
    CustomerInfo info = new CustomerInfo();
    SmartDataReader reader = new SmartDataReader(dataReader);

    info.ID = reader.GetString("ID");
    info.Name = reader.GetString("Name");
    info.Age = reader.GetInt32("Age");
    info.Creator = reader.GetString("Creator");
    info.CreateTime = reader.GetDateTime("CreateTime");

    return info;
}

/// <summary>
```

```
/// 将实体对象的属性值转化为 Hashtable 对应的键值
/// </summary>
/// <param name="obj">有效的实体对象</param>
/// <returns>包含键值映射的 Hashtable</returns>
protected override Hashtable GetHashByEntity(CustomerInfo obj)
{
    CustomerInfo info = obj as CustomerInfo;
    Hashtable hash = new Hashtable();

    hash.Add("ID", info.ID);
     hash.Add("Name", info.Name);
     hash.Add("Age", info.Age);
     hash.Add("Creator", info.Creator);
     hash.Add("CreateTime", info.CreateTime);

    return hash;
}
```

2.6. 业务逻辑层的实现分析

分析完成了数据访问层的接口和实现类后，我们来进一步看看业务逻辑层的实现分析，由于数据访问层的本意是基于特定的数据库实现，因此业务逻辑层就是抽象不同的数据库，让它们根据配置，指向不同的数据库实现类，从而实现多数据库的支持。

```
namespace WHC.TestProject.BLL
{
    /// <summary>
    /// 客户信息
    /// </summary>
    public class Customer : BaseBLL<CustomerInfo>
    {
        public Customer() : base()
        {
            base.Init(this.GetType().FullName,
System.Reflection.Assembly.GetExecutingAssembly().GetName().Name);
        }
    }
}
```

业务逻辑层的代码也很简单，在构造函数里面 Init 一下即可，之所以使用这个 Init 操作，其实为了确定 BLL 层的业务对象名称和指定在哪个程序集里面进行构造的需要，让给基类进行必要的创建工作。

在 BaseBLL 的 Init 函数里面，我们根据子类传入的相关参数，由于我们约定了数据访问类的命名空间，因此只根据数据库配置的不同需要，替换部分名称，就可以具体的构造出一个数据访问类了。

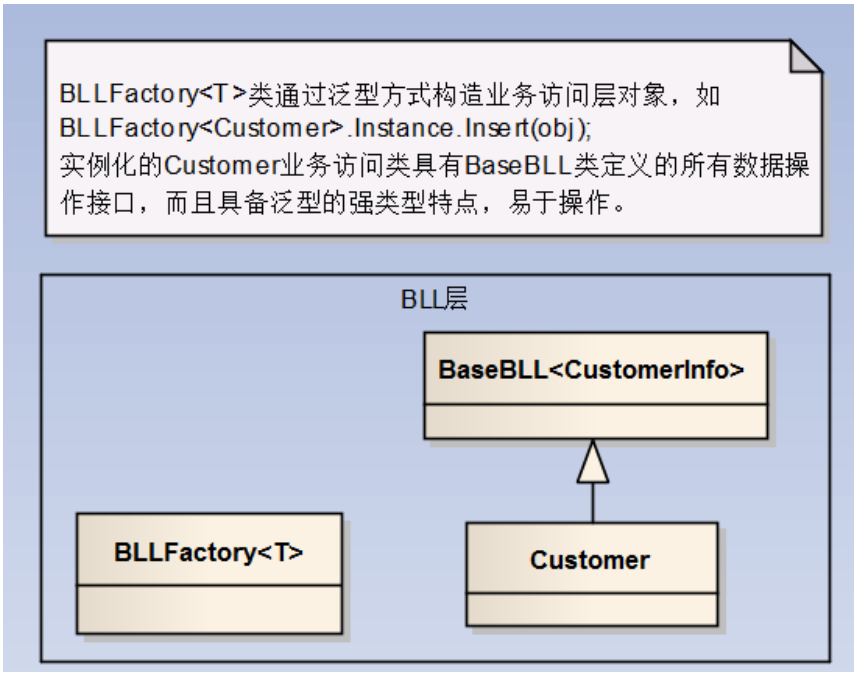
```
#region 根据不同的数据库类型，构造相应的 DAL 层
  AppConfig config = new AppConfig();
  string dbType = config.AppConfigGet("ComponentDbType");
  if (string.IsNullOrEmpty(dbType))
  {
      dbType = "sqlserver";
  }
  dbType = dbType.ToLower();

  string DALPrefix = "";
  if (dbType == "sqlserver")
  {
      DALPrefix = "DALSQL.";
  }
  else if (dbType == "access")
  {
      DALPrefix = "DALAccess.";
  }
  else if (dbType == "oracle")
  {
      DALPrefix = "DALOracle.";
  }
  else if (dbType == "sqlite")
  {
      DALPrefix = "DALSQLite.";
  }
  else if (dbType == "mysql")
  {
      DALPrefix = "DALMySql.";
  }
  #endregion

  this.dalName = bllFullName.Replace(bllPrefix, DALPrefix);//替换中级的 BLL.为 DAL.
就是 DAL 类的全名
  baseDal = Reflect<IBaseDAL<T>>.Create(this.dalName, dalAssemblyName);//构造对应的
DAL 数据访问层的对象类
```

这样精确构造出来的数据库访问访问对象，并把它转换为基类接口，那么就可以在

BaseBLL 类里的基类接口进行调用了。而构造业务对象，通过 BLLFactory<T>的泛型工厂，更能够精确构造出对应的业务对象类，这样构造出来的对象具有强类型，非常方便使用。



以上就是业务逻辑层，数据访问层和数据访问接口层的设计关系，为了高效进行开发工作，我们一定要使用强类型的接口调用，这样可以大大减少出错机会，而返回的基类接口，由于传入了特定的具体类型 T，也能够构造出强类型的列表或者对象。因此，合理利用泛型，能够是我们的开发体验更加美好，更加高效。

3. 存储过程的使用

在我前面很多篇关于框架设计和介绍的文章里面，大多数都是利用框架提供的基础性 API 进行各种的操作，包括增删改查、分页等各种实现和其衍生的实现，而这些实现绝大多数是基于 SQL 的标准操作实现的，由于框架的底层是利用了微软企业库 Enterprise Library，因此框架也是很好的支持存储过程的各种调用，不过由于整体性和数据库迁移方面的考虑，建议一般使用标准的 SQL 操作而已，这样能够很大程度上保证数据库可以很平滑过渡到其他数据库，如 Access、SQLite 等单机版数据库。

但是，有时候我们提供对存储过程的支持也是十分必要的，有些业务可能就只是固定在某种特定的数据库上跑，如 SQLServer、Oracle 等这些支持存储过程的关系型数据库，有

些业务可能还真的需要存储过程的整体性的封装；基于这个原因，我们在这小节对存储过程的使用进行介绍。

虽然存储过程一般用于处理一些复杂的逻辑关系或者报表内容，不过为了介绍方便，我们从几个较为基础的操作进行介绍。我们以一个客户表来进行对应的存储过程来介绍，先介绍客户表 T_Customer 的表定义。

	Name	Code	Comment	Default Value	Data Type	Length	Precision	P	F	M
1	ID	ID	编号		nvarchar(50)	50		☑	☐	☑
2	Name	Name	姓名		nvarchar(50)	50		☐	☐	☐
3	Age	Age	年龄		int			☐	☐	☐
4	Creator	Creator	创建人		nvarchar(50)	50		☐	☐	☐
➡	CreateTime	CreateTime	创建时间		datetime			☐	☐	☐
								☐	☐	☐
								☐	☐	☐

3.1. 存储过程的框架支持

前面介绍了，我们整个实例是以一个客户表 T_Customer 为例进行讲解的，整个表的框架支持代码，可以通过代码生成工具进行快速生成，生成后包括了 IDAL、Entity、DALSQL、BLL 层代码。框架在数据库的抽象基类 AbstractBaseDAL 里面，已经提供了对上面各种存储过程的支持，这样各个数据访问层的子类都可以进行调用，实现存储过程的处理。

对于存储过程的实现，我们分析一下各个接口，可以看到，输入参数是可选的，因为有些接口不需要输出参数；输出参数也是可选的，有些接口也不需要输出参数，返回的记录类型主要有 bool 类型，实体类型，实体集合类型，DataTable 类型这几种，当然虽然有年龄接口的整形，但是这个是通过输出参数来获得的。我们于是可以定义一些类似这样的通用接口参数集合，用来处理存储过程调用的封装接口。下面是在框架的数据库抽象基类 AbstractBaseDAL 里面定义好的通用存储过程接口。

```
#region 存储过程执行通用方法

/// <summary>
/// 执行存储过程，如果影响记录数，返回True，否则为False，修改并输出外部参数outParameters（如果有）。
/// </summary>
/// <param name="storeProcName">存储过程名称</param>
/// <param name="inParameters">输入参数，可为空</param>
/// <param name="outParameters">输出参数，可为空</param>
```

```
/// <param name="trans">事务对象，可为空</param>
/// <returns>如果影响记录数，返回True，否则为False</returns>
public bool StorePorcExecute(string storeProcName, Hashtable inParameters = null,
Hashtable outParameters = null, DbTransaction trans = null)

/// <summary>
/// 执行存储过程，返回实体列表集合，修改并输出外部参数outParameters（如果有）。
/// </summary>
/// <param name="storeProcName">存储过程名称</param>
/// <param name="inParameters">输入参数，可为空</param>
/// <param name="outParameters">输出参数，可为空</param>
/// <param name="trans">事务对象，可为空</param>
/// <returns>返回实体列表集合</returns>
public List<T> StorePorcToList(string storeProcName, Hashtable inParameters = null,
Hashtable outParameters = null, DbTransaction trans = null)

/// <summary>
/// 执行存储过程，返回DataTable集合，修改并输出外部参数outParameters（如果有）。
/// </summary>
/// <param name="storeProcName">存储过程名称</param>
/// <param name="inParameters">输入参数，可为空</param>
/// <param name="outParameters">输出参数，可为空</param>
/// <param name="trans">事务对象，可为空</param>
/// <returns>返回DataTable集合</returns>
public DataTable StorePorcToDataTable(string storeProcName, Hashtable inParameters =
null, Hashtable outParameters = null, DbTransaction trans = null)

/// <summary>
/// 执行存储过程，返回实体对象，修改并输出外部参数outParameters（如果有）。
/// </summary>
/// <param name="storeProcName">存储过程名称</param>
/// <param name="inParameters">输入参数，可为空</param>
/// <param name="outParameters">输出参数，可为空</param>
/// <param name="trans">事务对象，可为空</param>
/// <returns>返回实体对象</returns>
public T StorePorcToEntity(string storeProcName, Hashtable inParameters = null,
    Hashtable outParameters = null, DbTransaction trans = null)

#endregion
```

3.2. SQLServer 存储过程

3.2.1. SQLServer 存储过程的编写

1）提供执行处理，可对执行结果进行反馈

这种情况常常可以见到，如可以对插入、更新、删除等操作进行处理，并获得执行的结果，下面是这两种存储过程的代码。

```
-----------------------------------
--功能描述：插入数据到表中
-----------------------------------
 CREATE PROCEDURE dbo.T_Customer_Insert
 (
      @ID varchar(50),
      @Name varchar(50) ,
      @Age int
 )
 AS
 begin tran
 Insert into dbo.T_Customer( ID,Name,Age ) Values( @ID,@Name,@Age )
 if @@error!=0
     begin
         rollback
     end
 else
     begin
         commit
     end
 go
```

2）提供执行处理，获得一个或者多个返回性参数，并可对执行结果进行反馈。

基于上面的处理方式，我们可能还有一种情况，就是需要执行存储过程，并返回对应的返回参数，我们可以在程序里面利用代码获取这些返回参数的数值，从而用作其他用途。

因此，这种操作，如要是获取返回性参数的情况，如下所示是判断记录是否存在，以及获取客户最大年龄的两个存储过程。

```
-----------------------------------
--功能描述：以字段 ID 为关键字，检查表中是否存在符合条件的记录
-----------------------------------
 CREATE PROCEDURE dbo.T_Customer_ExistByID
 (
```

```
    @Exist int output ,
     @ID varchar(50)
)
AS
Select @Exist = Case When Exists (Select 1 From dbo.T_Customer Where ID=@ID) Then 1 Else
0 End
go
```

3）提供查询处理，并返回实体对象

这小节后面介绍的内容，都是存储过程的返回值，这些或者是一条记录，或者是多条记录的查询结果，这个在 SQLServer 里面很容易实现，而在 Oracle 里面需要通过游标进行处理。

下面存储过程脚本，是基于返回单条记录的存储过程。

```
-----------------------------------
--功能描述：以字段 ID 为关键字，检索表中的数据
-----------------------------------
CREATE PROCEDURE dbo.T_Customer_SelectByID
(
    @ID varchar(50)
)
AS
Select * from dbo.T_Customer Where ID= @ID
go
```

4）提供查询处理，并返回多条记录集合；包括实体列表集合或 DataTable 集合对象

对于返回多条集合的对象，在存储过程里面体现都一样的，我们可能在 C#处理的时候，把它转换为不同的对象即可，返回多个集合，在 SQLServer 里面，它们的存储过程代码如下所示。

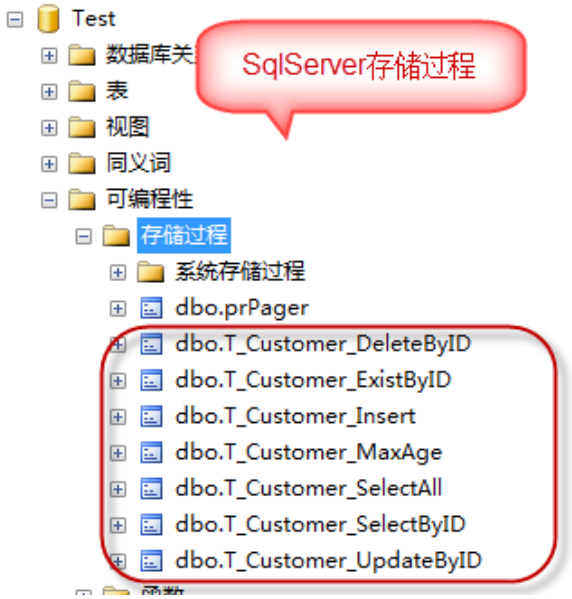
```
-----------------------------------
--功能描述：检索表中所有的数据
-----------------------------------
CREATE PROCEDURE dbo.T_Customer_SelectAll
AS
Select * from dbo.T_Customer
go
```

3.2.2. SQLServer 存储过程的使用

上面小节，介绍了在 SQLServer 里面，如何编写个各类存储过程，本小节主要介绍如何在 C#里面，如何对这些存储过程进行调用，并获取到对应的数据类型，如输出参数，单

个数据记录，多个数据记录等情况。

对存储过程的使用，也还需要我们进行定义相应的实现，如编写存储过程并在数据库上执行，然后在代码进行封装调用。我们可根据需要在 SQLServer 数据库上创建一些需要使用的存储过程，如下图所示。



例如对于 SqlServer 数据访问层，使用超级基类的接口，我们简化代码如下所示。

```
        public bool StorePorc_Insert(CustomerInfo info, DbTransaction trans =
null)
        {
            Hashtable inParameters = new Hashtable();
            inParameters.Add("ID", info.ID);
            inParameters.Add("Name", info.Name);
            inParameters.Add("Age", info.Age);

            return StorePorcExecute("T_Customer_Insert", inParameters, null,
trans);
        }

        public bool StorePorc_Update(CustomerInfo info, DbTransaction trans =
null)
        {
            Hashtable inParameters = new Hashtable();
            inParameters.Add("ID", info.ID);
            inParameters.Add("Name", info.Name);
            inParameters.Add("Age", info.Age);
```

```
            return StorePorcExecute("T_Customer_UpdateByID", inParameters,
null, trans);
        }

        public List<CustomerInfo> StorePorc_GetAll(DbTransaction trans = null)
        {
            return StorePorcToList("T_Customer_SelectAll", null, null, trans);
        }
        public DataTable StorePorc_GetAllToDataTable(DbTransaction trans =
null)
        {
            return StorePorcToDataTable("T_Customer_SelectAll", null, null,
trans);
        }
        public CustomerInfo StorePorc_FindByID(string ID, DbTransaction trans
= null)
        {
            Hashtable inParameters = new Hashtable();
            inParameters.Add("ID", ID);

            return StorePorcToEntity("T_Customer_SelectByID", inParameters,
null, trans);
        }
        public bool StorePorc_ExistByID(string ID, DbTransaction trans = null)
        {
            Hashtable inParameters = new Hashtable();
            inParameters.Add("ID", ID);

            Hashtable outParameters = new Hashtable();
            outParameters.Add("Exist", 0);

            StorePorcExecute("T_Customer_ExistByID", inParameters,
outParameters, trans);
            int exist = (int)outParameters["Exist"];
            return exist > 0;
        }

        public bool StorePorc_DeleteByID(string ID, DbTransaction trans = null)
        {
            Hashtable inParameters = new Hashtable();
            inParameters.Add("ID", ID);
```

```
            return StorePorcExecute("T_Customer_DeleteByID", inParameters,
null, trans);
        }

        public int StorePorc_GetMaxAge()
        {
            Hashtable outParameters = new Hashtable();
            outParameters.Add("MaxAge", 0);

            StorePorcExecute("T_Customer_MaxAge", null, outParameters, null);
            int MaxAge = (int)outParameters["MaxAge"];
            return MaxAge;
        }
```

3.3. Oracle 存储过程

3.3.1. Oracle 存储过程的编写

对应 SQLServer 的存储过程，Oracle 的存储过程也提供了对应的版本，下面是几种情况下的 Oracle 存储过程的编写。

1）提供执行处理，可对执行结果进行反馈

```
--------------------------------
--功能描述：插入数据到表中
--------------------------------
 Create Or Replace Procedure T_Customer_Insert
 (
     p_ID IN T_CUSTOMER.ID%TYPE,
     p_Name IN T_CUSTOMER.NAME%TYPE,
     p_Age IN T_CUSTOMER.AGE%TYPE
 )
 AS
 Begin
 Insert into T_CUSTOMER( ID,NAME,AGE ) Values( p_ID,p_Name,p_Age ) ;
 Commit;
 Exception
     When Others Then
 Rollback;

 End;
```

其中上面的代码涉及几个地方，T_CUSTOMER.ID%TYPE 是表示根据字段动态决定参数的类型，避免应硬编码或者反复修改参数类型。Oracle 的参数一般使用 p_的前缀开始，方便区分。

2）提供执行处理，获得一个或者多个返回性参数，并可对执行结果进行反馈。

```
--------------------------------
--功能描述：以字段 ID 为关键字，检查表中是否存在符合条件的记录
--------------------------------
Create Or Replace Procedure T_Customer_ExistByID
(
    p_Exist OUT Number  ,
    p_ID IN T_CUSTOMER.ID%TYPE
)
AS
Begin
--V9.i 以下使用的语句
Select Case When (Count(1)>0) Then 1 Else 0 End Into p_Exist From T_CUSTOMER Where ID=p_ID ;
--也可以使用的语句
-- Select Decode(Count(1),0,0,1) Into p_Exist From T_CUSTOMER Where ID=p_ID ;
End;
```

上面的代码，都有一个输出的参数，虽然他们执行没有影响记录函数，但是这个主要是通过输出参数的值进行处理了。

3）提供查询处理，并返回实体对象

提供查询处理，不管返回一条记录，还是多条记录，在 Oracle 里面，一般都是通过游标进行处理的，因此我们需要先定义一个游标类型，供我们返回记录使用的。下面定义一个游标的包代码如下。

```
--------------------------------
--功能描述：创建一个包，含有一个游标类型:(一个数据库中只需声明一次)
--------------------------------
CREATE OR REPLACE PACKAGE MyCURSOR
AS
    TYPE cur_OUT IS REF CURSOR;
End;
```

然后我们就可以在各个返回记录的存储过程里面使用这个游标类型了。

例如在下面的存储过程里面，返回一条指定的数据记录，那么输出参数里面需要有一个游标的定义参数，但是我们在 C#里面使用数据访问框架来处理数据的时候，可以忽略他它的存在，就只需要输入 p_ID 参数就可以了。

```
--------------------------------
--功能描述：以字段 ID 为关键字，检索表中的数据
--------------------------------
```

```
Create Or Replace Procedure T_Customer_SelectByID
(
    cur_OUT OUT MyCURSOR.cur_OUT  ,
    p_ID IN T_CUSTOMER.ID%TYPE
)
AS
Begin
OPEN cur_OUT FOR Select * from T_CUSTOMER Where ID = p_ID ;
End;
```

4）提供查询处理，并返回多条记录集合；包括实体列表集合或 DataTable 集合对象

和上面返回单条记录一样，需要返回多条记录的存储过程，也需要使用一个游标的输出参数来获取返回的记录，并可以对游标进行处理。

```
-----------------------------------
--功能描述：检索表中所有的数据
-----------------------------------
Create Or Replace Procedure T_Customer_SelectAll
( cur_OUT OUT MyCURSOR.cur_OUT )
AS
Begin
OPEN cur_OUT FOR Select * from T_CUSTOMER;
End;
```

3.3.2. Oracle 存储过程的使用

上面小节，介绍了在 Oracle 里面，如何编写个各类存储过程，本小节主要介绍如何在 C#里面，如何对这些存储过程进行调用，并获取到对应的数据类型，如输出参数，单个数据记录，多个数据记录等情况。

对于 Oracle 数据访问层的实现来说，它的接口实现一样简单，只是参数命名有所不同而已。

```
        public bool StorePorc_Insert(CustomerInfo info, DbTransaction trans =
null)
        {
            Hashtable inParameters = new Hashtable();
            inParameters.Add("p_ID", info.ID);
            inParameters.Add("p_Name", info.Name);
            inParameters.Add("p_Age", info.Age);
```

```
            return StorePorcExecute("T_Customer_Insert", inParameters, null,
trans);
        }
        public bool StorePorc_Update(CustomerInfo info, DbTransaction trans =
null)
        {
            Hashtable inParameters = new Hashtable();
            inParameters.Add("p_ID", info.ID);
            inParameters.Add("p_Name", info.Name);
            inParameters.Add("p_Age", info.Age);

            return StorePorcExecute("T_Customer_UpdateByID", inParameters,
null, trans);
        }
        public List<CustomerInfo> StorePorc_GetAll(DbTransaction trans = null)
        {
            return StorePorcToList("T_Customer_SelectAll", null, null, trans);
        }
        public DataTable StorePorc_GetAllToDataTable(DbTransaction trans =
null)
        {
            return StorePorcToDataTable("T_Customer_SelectAll", null, null,
trans);
        }
        public CustomerInfo StorePorc_FindByID(string ID, DbTransaction trans
= null)
        {
            Hashtable inParameters = new Hashtable();
            inParameters.Add("p_ID", ID);

            return StorePorcToEntity("T_Customer_SelectByID", inParameters,
null, trans);
        }
        public bool StorePorc_ExistByID(string ID, DbTransaction trans = null)
        {
            Hashtable inParameters = new Hashtable();
            inParameters.Add("p_ID", ID);

            Hashtable outParameters = new Hashtable();
            outParameters.Add("p_Exist", 0);

            StorePorcExecute("T_Customer_ExistByID", inParameters,
outParameters, trans);
```

```
            int exist = (int)outParameters["p_Exist"];
            return exist > 0;
        }

        public bool StorePorc_DeleteByID(string ID, DbTransaction trans = null)
        {
            Hashtable inParameters = new Hashtable();
            inParameters.Add("p_ID", ID);

            return StorePorcExecute("T_Customer_DeleteByID", inParameters,
null, trans);
        }

        public int StorePorc_GetMaxAge()
        {
            Hashtable outParameters = new Hashtable();
            outParameters.Add("p_MaxAge", 0);

            StorePorcExecute("T_Customer_MaxAge", null, outParameters, null);
            int MaxAge = (int)outParameters["p_MaxAge"];
            return MaxAge;
        }
```

4. Web 项目的设计分析

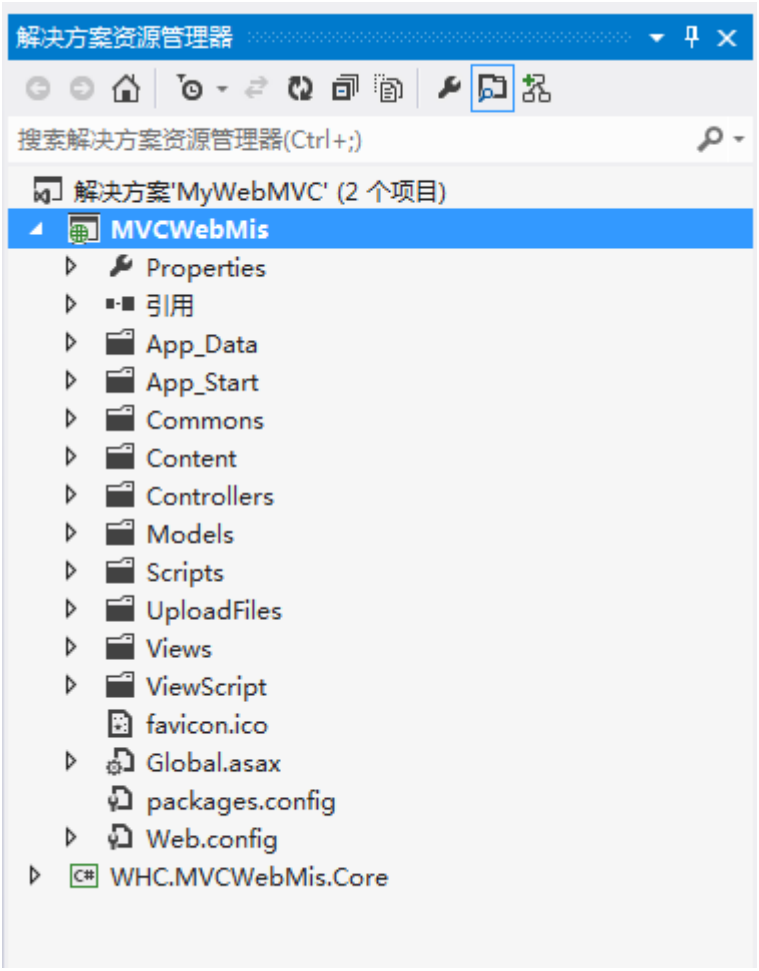
4.1. MVC 的 Web 架构说明

我们知道，在传统基于 ASP.NET 的 Web 开发中，采用的就是 WebForm 这种方式，这种方式可以在界面上拖动一个按钮等界面元素，然后双击可以进行后台代码进行控制相关的内容，这些界面元素是服务端控件。这种方式虽然方便操作理解，但是可能会引入过多的业务逻辑在后台代码里面，而随着代码量的增加，前台代码和后台代码的耦合性也很大，造成 WebForm 这种方式开发的项目，测试比较麻烦。

MVC 的 Web 开发，颠覆了过去那种服务端控件的方式，采用的是纯 HTML 控件进行界面展示，并引入了 MVC 的模式，使得控制器、视图和模型各司其职，使得 Web 开发更加规范、容易测试、更高效率等特点。

4.1.1. MVC 的项目目录说明

打开 Web 项目，我们可能看到下面的项目目录布局，如下所示。

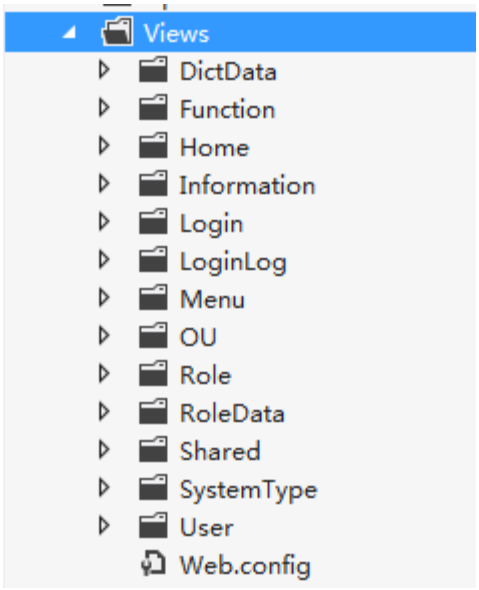


界面层的目录说明如下所示

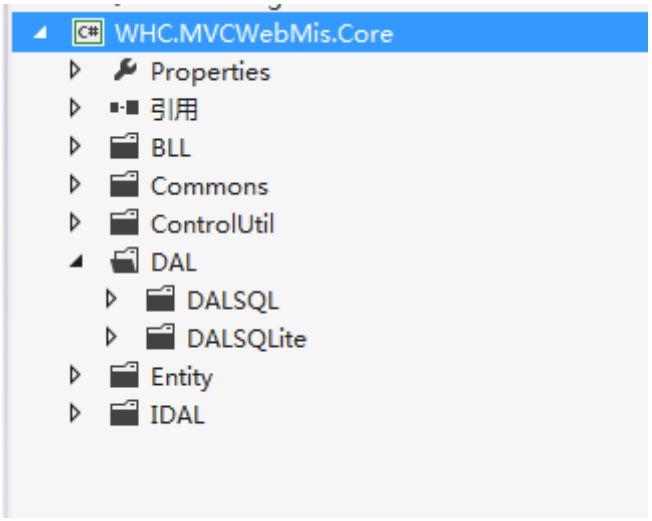
目录	说明
App_Data	放置数据的目录，如一般可以把 Sqlite 数据库文件放到这个目录下，可以通过在配置文件添加路径进行`|DataDirectory|`引用。
App_Start	放置配置文件代码，如 MVC 的文件路由设置等。
Commons	放置一些界面层用到的辅助类和扩展类代码。
Content	放置 css 和除了 JavaScript 脚本,图像以外的东西
Controllers	放置 MVC 模式中的控制器类
Models	放置数据描述、操纵类和业务对象类
Scripts	放置 JavaScript 脚本

UploadFiles	附件上传的基础目录，为了方便管理，我们把附件都上传到这个基目录下了。
Views	放置 MVC 模式中的视图类

视图目录必须和控制器的名称有对应关系，如有一个控制器为 TestController，必然有一个视图的目录为 Test，这个 Test 目录下可能有多个 cshtml 的视图文件，一般为 Index.cshtml，这个是主视图文件。



而 Web 框架的核心项目工程如下所示。



功能具体目录列表如下所示：

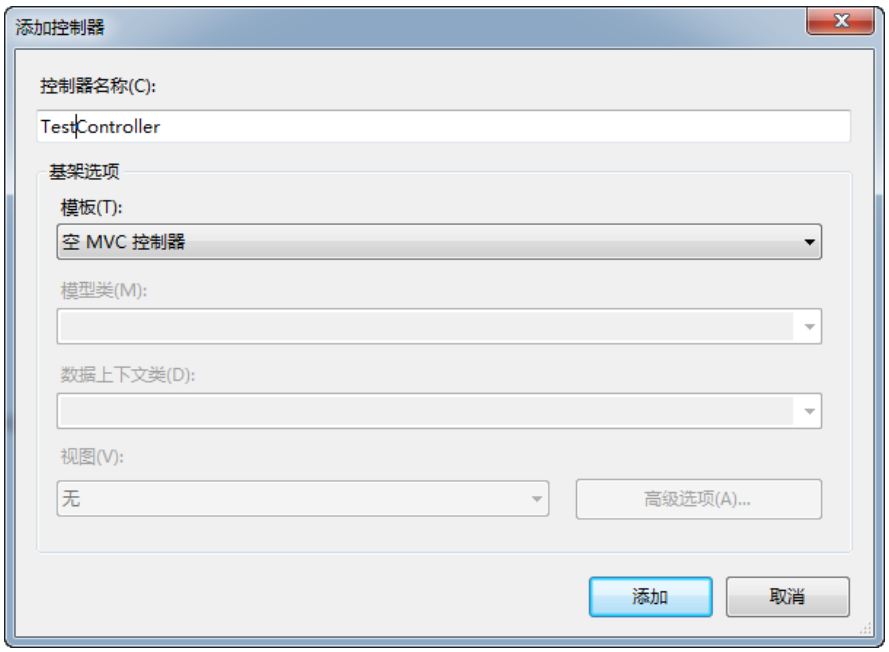
目录	说明
BLL	业务逻辑层的代码，使用代码生成的结构

Commons	公用类库里面的纯.NET 类库辅助类
ControlUtil	公用类库里面的第三方类库扩展辅助类
DAL	数据访问层的主目录
DAL/DALSQL	基于 Sqlserver 的数据访问层目录
DAL/DALSQLite	基于 SQLite 的数据访问层目录
Entity	实体层的目录
IDAL	数据访问层接口定义层目录

4.1.2. MVC 的控制器设计

MVC 的控制器很多都有相似的地方，在设计使用之初，我就希望尽可能的减少代码，提高编程模型的统一性。因此希望能够以基类继承的方式，和我 Winform 开发框架一样，尽可能通过基类，而不是子类的重复代码来实现各种通用的操作。

我们知道，一般我们创建一个 MVC 的控制器，都是基于 Controller 这样的基类来实现。如下代码所示。




```
public class TestController : Controller
{
    //
    // GET: /Test/
    public ActionResult Index()
```

```
        {
            return View();
        }
    }
```

在我的 Winform 开发框架里面，用到了泛型的类型，非常方便实现业务逻辑和数据访问基类的设计，控制器是否也可以这样做的呢？

我们知道，一般的 MVC 控制器需要验证用户是否已经登录了，这也是很多常见 Web 操作前的验证，还有对异常的处理，在 MVC 的基类，可以一并进行记录（这个非常不错），于是我们先来设计一个验证用户身份是否登录的基类 BaseController。

```
/// <summary>
/// 所有需要进行登录控制的控制器基类
/// </summary>
public class BaseController : Controller
{
    /// <summary>
    /// 当前登录的用户属性
    /// </summary>
    public UserInfo CurrentUserInfo { get; set; }

    /// <summary>
    /// 重新基类在 Action 执行之前的事情
    /// </summary>
    /// <param name="filterContext">重写方法的参数</param>
    protected override void OnActionExecuting(ActionExecutingContext
filterContext)
    {
        base.OnActionExecuting(filterContext);
        //得到用户登录的信息
        CurrentUserInfo = Session["UserInfo"] as UserInfo;

        //判断用户是否为空
        if (CurrentUserInfo == null)
        {
        Response.Redirect("/Login/Index");
        }
    }

    protected override void OnException(ExceptionContext filterContext)
    {
        base.OnException(filterContext);
```

```
        //错误记录
        WHC.Framework.Commons.LogTextHelper.Error(filterContext.Exception);

        // 当自定义显示错误 mode = On，显示友好错误页面
        if (filterContext.HttpContext.IsCustomErrorEnabled)
        {
            filterContext.ExceptionHandled = true;
            this.View("Error").ExecuteResult(this.ControllerContext);
        }
    }
.....................
    }
```

有了这个基类，我们在主页的 Home 控制类，就可以使用用户信息对象了进行操作了，而且必须要求客户登录了。

```
    public class HomeController : BaseController
    {
        public ActionResult Index()
        {
            if (CurrentUserInfo != null)
            {
                ViewBag.FullName = CurrentUserInfo.FullName;
                ViewBag.Name = CurrentUserInfo.Name;
            }
            return View();
        }
..............
    }
```

为了方便控制器对封装的业务类的调用，我们在 BaseController 的基础上再次引入一个 BusinessController<B, T>控制器基类，让其默认就具有某些数据访问操作的功能，包括控制器一般的增删改查等常规数据操作，这样可以把所有常规的操作都放到基类 BusinessController<B, T>里面进行实现，而需要数据访问操作的控制器继承这个基类就可以了，减少为每个控制器类都编写相似的代码，提高开发效率，增强代码的健壮性和可维护性。

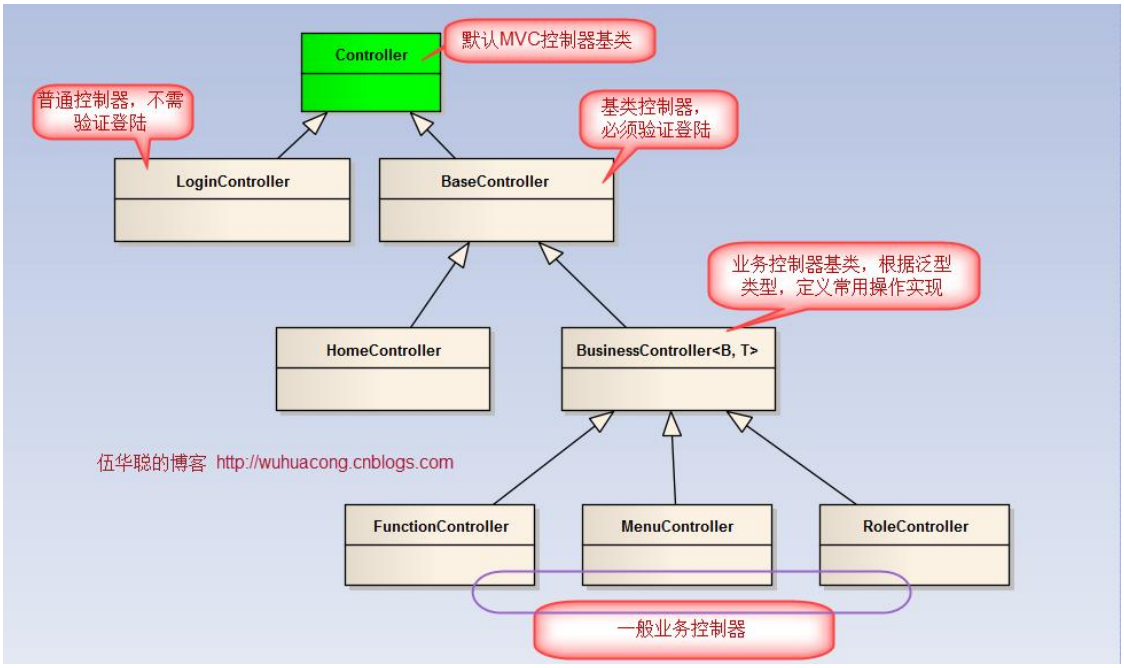
```
    /// <summary>
    /// 本控制器基类专门为访问数据业务对象而设的基类
    /// </summary>
    /// <typeparam name="B">业务对象类型</typeparam>
    /// <typeparam name="T">实体类类型</typeparam>
    public class BusinessController<B, T> : BaseController
        where B : class
        where T : WHC.Framework.ControlUtil.BaseEntity, new()
```

```
    {

        /// <summary>
        /// 插入指定对象到数据库中
        /// </summary>
        /// <param name="info">指定的对象</param>
        /// <returns>执行操作是否成功。</returns>
        public virtual ActionResult Insert(T info)
        {
            bool result = false;
            if (info != null)
            {
                result = baseBLL.Insert(info);
            }
            return Content(result);
        }

.................
```

最终项目的控制器设计层级图如下所示。



我们以角色控制器来说明，它的定义如下所示，如果不需要实现额外的接口（除了常见的操作），基本上不需要写任何代码了，因为所有很多常见的操作，都已经封装在了基类控制器 BusinessController<B, T>里面了。

```
    /// <summary>
    /// 角色业务操作控制器
```

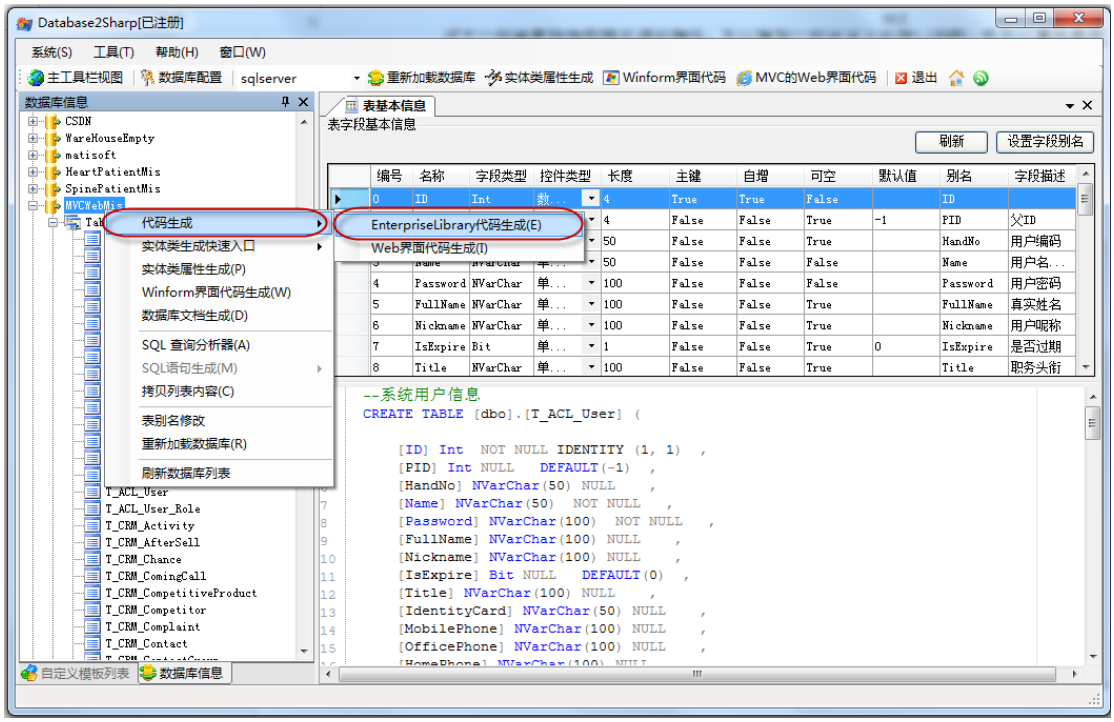
```
    /// </summary>
    public class RoleController : BusinessController<Role, RoleInfo>
    {
        public RoleController() : base()
        {
        }

.....
```

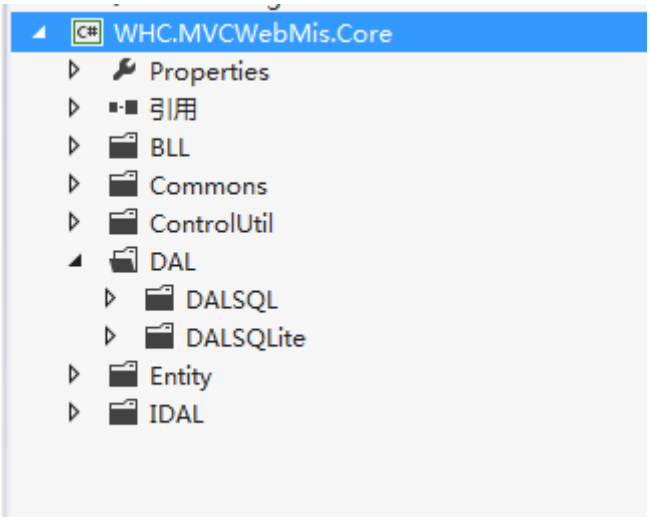
对于一些需要特殊数据处理的操作，可以增加一些自定义的接口函数，也可以重写基类的一些接口，实现数据的相应处理。

4.2. Web 项目代码的生成操作

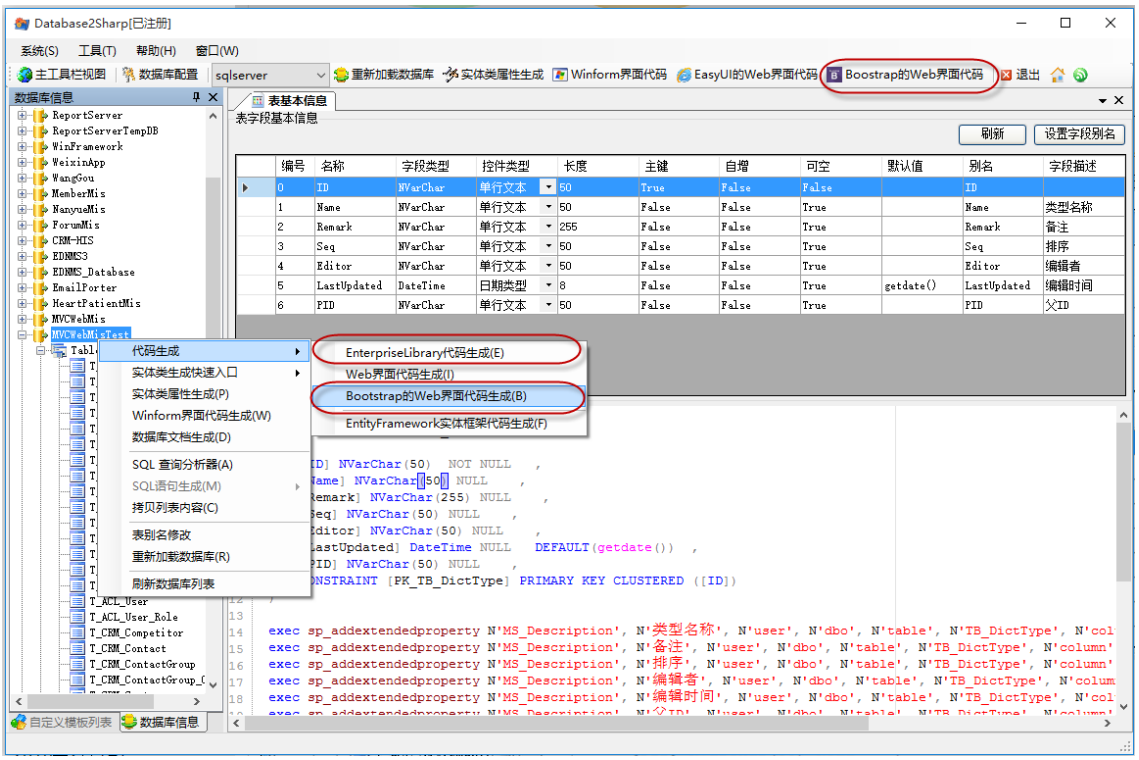
4.2.1. 核心逻辑代码生成



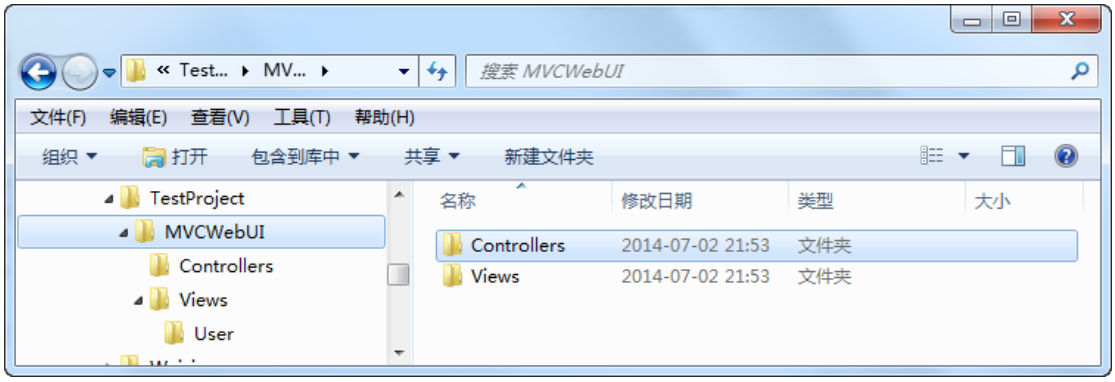
选择【代码生成】【Enterprise Library 代码生成】后，会提示选择数据库和表后，然后就会根据数据库和配置信息，生成核心逻辑代码，生成的代码的结构和下面的结构相似。



4.2.2. Web 界面层代码生成



选定表并一步步进行 Web 界面生成后，最后会生成相应控制器和视图代码，控制器已经继承了基类 BusinessController，而 Web 界面的视图已经包含常规的查询、列表、增加、编辑、查看、删除等操作的界面和功能操作，文件如下所示。



控制器代码如下所示。

```
namespace WHC.TestProject.Controllers
{
    public class UserController : BusinessController<User, UserInfo>
    {
        public UserController() : base()
        {
        }

    }
}
```

而页面视图代码则包含内容比较多，它包括了列表界面、新增、编辑、查看等层的界面定义和数据绑定的脚本等内容。

1）相关文件和脚本的引用

```
index.cshtml
<html>
    <head>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
    <meta name="viewport" content="width=device-width" />
    <title>系统用户信息</title>
    @*添加Jquery EasyUI的样式*@
    <link href="~/Content/JqueryEasyUI/themes/default/easyui.css" rel="stylesheet" type="text/css" />
    <link href="~/Content/JqueryEasyUI/themes/icon.css" rel="stylesheet" type="text/css" />

    <link href="~/Content/themes/Default/style.css" rel="stylesheet" type="text/css" />
    <link href="~/Content/themes/Default/default.css" rel="stylesheet" type="text/css" />

    @*添加Jquery，EasyUI和easyUI的语言包的JS文件*@
    <script type="text/javascript" src="~/Content/JqueryEasyUI/jquery.min.js"></script>
    <script type="text/javascript" src="~/Content/JqueryEasyUI/jquery.easyui.min.js"></script>
    <script type="text/javascript" src="~/Content/JqueryEasyUI/locale/easyui-lang-zh_CN.js"></script>

    @*日期格式的引用*@
    <script src="~/Content/datapattern.js"></script>

    @*添加对ckeditor的支持*@
    <script src="~/Content/JQueryTools/ckeditor/ckeditor.js"></script>
    <script src="~/Content/JQueryTools/ckeditor/adapters/jquery.js"></script>
  <script>
```

上面这种方式应用 JS 文档或者 CSS 文件，是一种常见的方式，不过每个页面如果这样应用，会显得比较臃肿，因此我在框架里面，对这个引用方式进行了优化，使用 MVC 里面

的 Bundles 属性。

在 ASP.NET MVC 出来之后，引入了一个叫做 Bundle 的东西，它用来将 js 和 css 文件捆绑为一个块进行输出，能够极大简化界面代码，并默认对这些内容进行压缩处理，提高效率。最终简化的界面代码如下所示。

Index.cshtml

```
@{
    ViewBag.Title = "用户管理";
}

<!DOCTYPE html>
<html>
    <head>
    <title>用户管理</title>
    <meta name="viewport" content="width=device-width" />
    @using System.Web.Optimization;
    @Scripts.Render("~/bundles/jquery")
    @Styles.Render("~/Content/css")
    @Scripts.Render("~/bundles/jquerytools")
    @Styles.Render("~/Content/jquerytools")

    <!--常用的一些组件业务脚本函数,放置此处方便脚本提示-->
    <script src="~/Scripts/ComponentUtil.js"></script>
```

2）通过脚本的页面内容初始化操作

index.cshtml

```
<script type="text/javascript">
    var isAddOrEdit = 'add';//标识是新增还是编辑对话框弹出，用于删除附件的操作

    $(function () {
        initEditor();     //初始化CKEditor
        var queryData = {  };//可添加一些预设条件
        InitGrid(queryData);//初始化Datagrid表格数据
        InitDictItem();   //初始化字典信息

        BindSearchEvent();//绑定查询按钮事件
        BindAddEvent();   //绑定添加按钮的事件
        BindEditEvent();  //绑定修改按钮的事件

    });

    //初始化字典信息
    function InitDictItem() {
        //$('#PID').combobox({
        //    url: '/User/GetDictJson',
        //    valueField: 'Value',
        //    textField: 'Text'
        //});
    }

    //实现对DataGird控件的绑定操作
    function InitGrid(queryData) {
        $('#grid').datagrid({   //定位到Table标签，Table标签的ID是grid
            url: '/User/FindWithPager',   //指向后台的Action来获取当前用户的信息的Json格式的数据
            title: '系统用户信息',
            iconCls: 'icon-view',
            height: 650,
            width: function () { return document.body.clientWidth * 0.9 },//自动宽度
            nowrap: true,
```

以及各 DIV 层的数据绑定的等操作均在一个页面内实现。

5. Web 框架的架构特点

5.1. 技术特点

1）框架架构设计

整个基于 Metronic 的 Bootstrap 开发框架，界面部分采用较新的 Bootstrap 技术，采用当前最新的 Bootstrap3.x，集成了众多功能强大的 Bootstrap 控件。

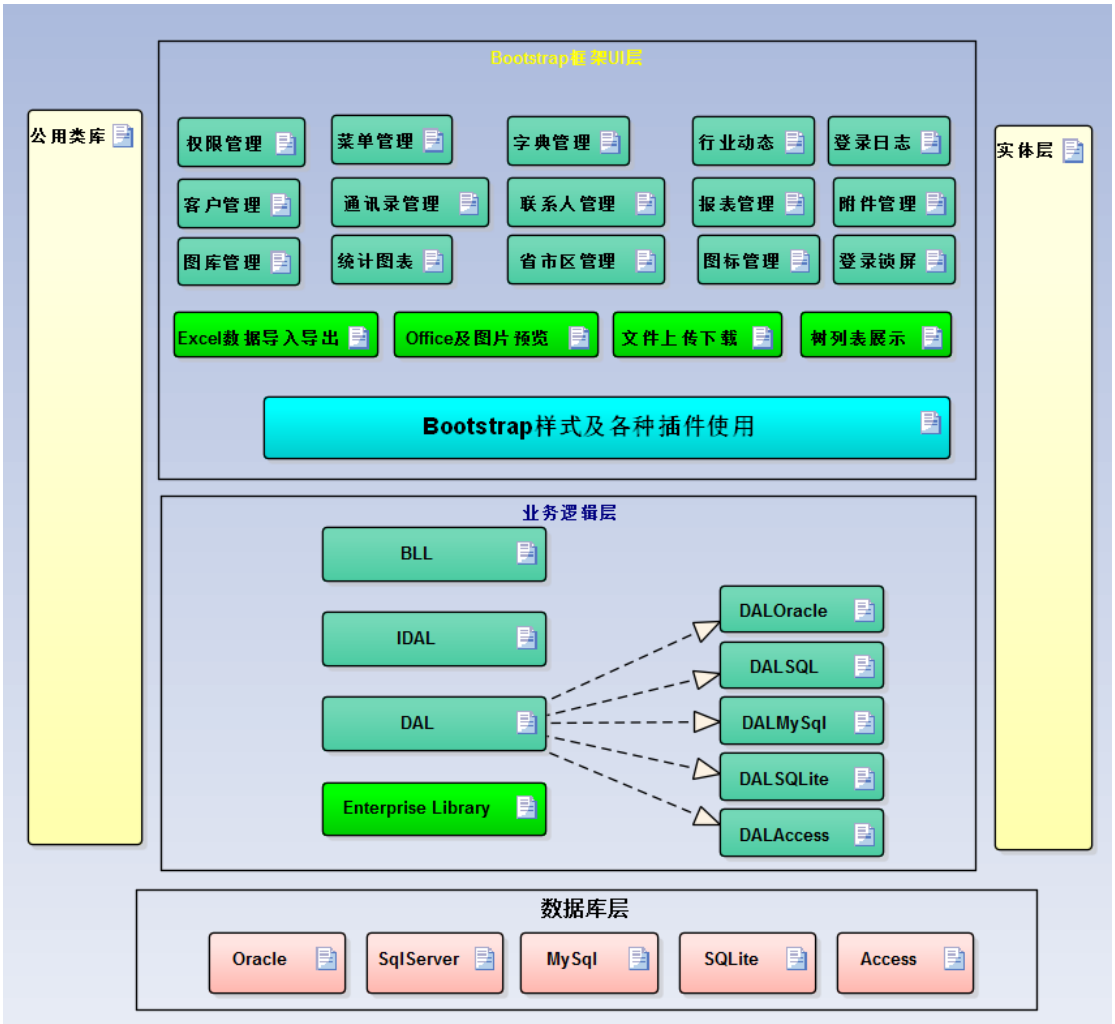
Bootstrap 是一个前端的技术框架，很多平台都可以采用，JAVA/PHP/.NET 都可以用来做前端界面，整合 JQuery 可以实现非常丰富的界面效果，目前也有很多 Bootstrap 的插件能够提供给大家使用，本框架集合了众多最为优秀的插件，能给我们 Web 的用户体验提升到一个前所未有的水平。

Metronic 是一个国外的基于 HTML、JS 等技术的 Bootstrap 开发框架整合，整合了很多 Bootstrap 的前端技术和插件的使用，是一个非常不错的技术框架。本框架以这个为基础，结合我对 MVC 的 Web 框架的研究，整合了基于 MVC 的 Bootstrap 开发框架，使之能够符合实际项目的结构需要。

框架后台采用基于 C#的 MVC 技术，是目前.NET 开发最为成熟流行的技术，框架后台数据库支持 Oracle、SqlServer、MySql、Sqlite、Access 等常规数据库，可通过配置进行自由切换，使用 Enterprise Library 模块进行数据访问的控制，使得数据访问更方便轻松。

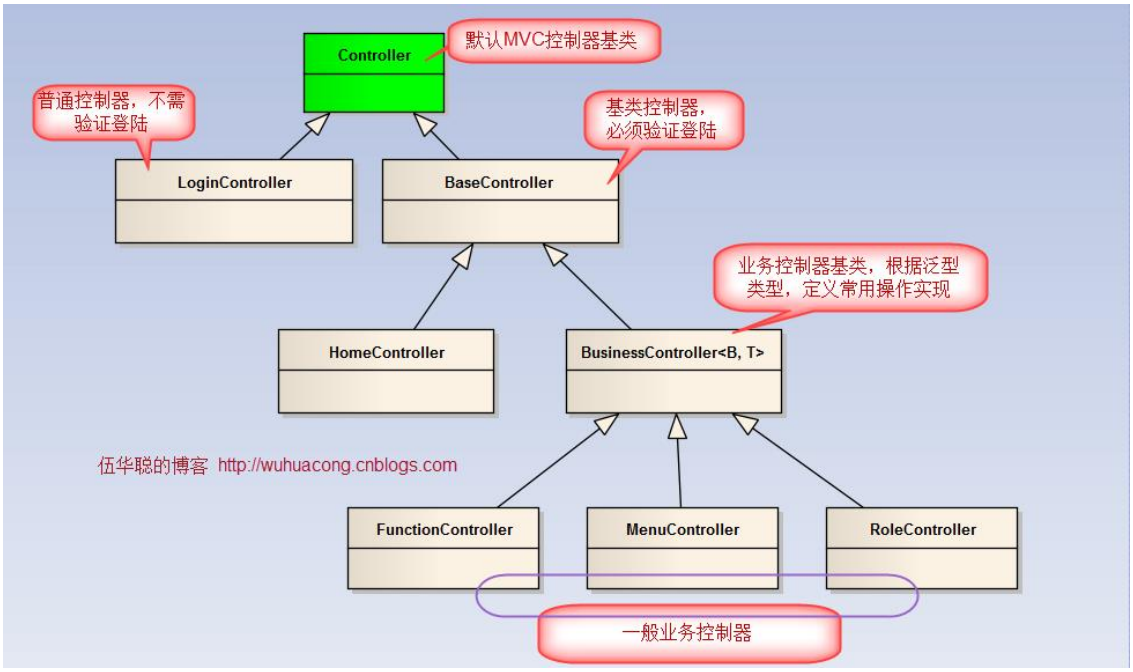
整体框架开发采用 Visual Studuio 2013 以及页面编辑工具 Sublime Text 结合开发，页面以及后台代码，通过代码生成工具 Database2Sharp 进行快速开发，实现整体性开发的最大效率提高。

框架的总体结构如下所示：



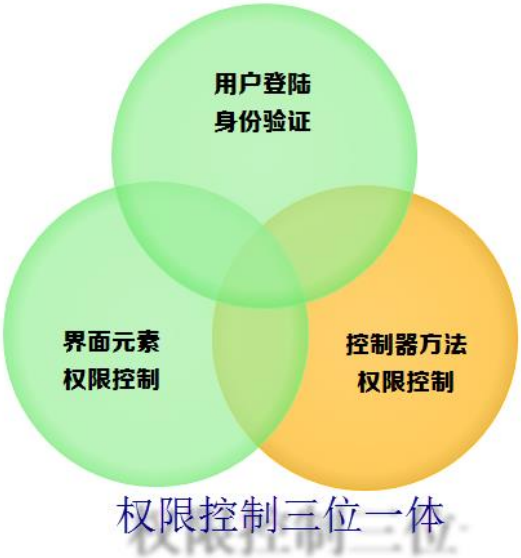
2）控制器设计

Bootstrap 开发框架沿用了我的《Winform 开发框架》和《基于 EasyUI 的 Web 框架》的很多架构设计思路和特点，对 Controller 进行了封装。使得控制器能够获得很好的继承关系，并能以更少的代码，更高效的开发效率，实现 Web 项目的开发工作，整个控制器的设计思路如下所示。



3）权限控制

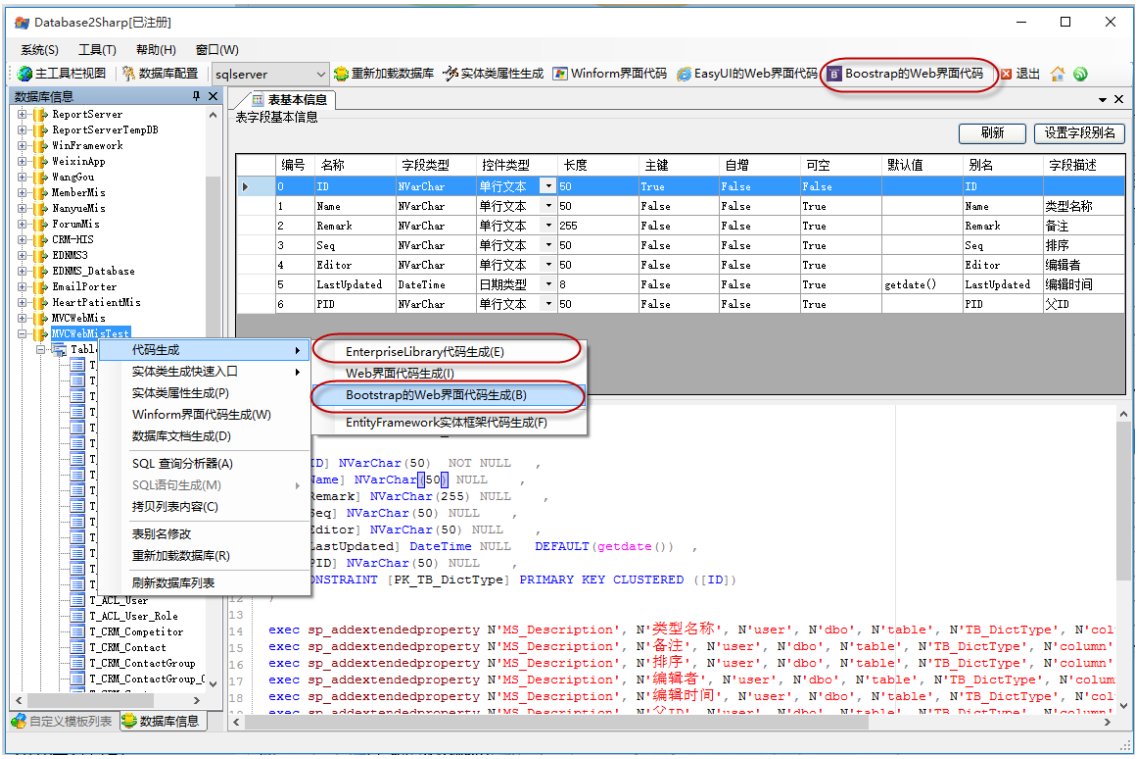
良好的控制器设计规则，可以为 Web 开发框架本身提供了很好用户访问控制和权限控制，使得用户界面呈现菜单、Web 界面的按钮和内容、Action 的提交控制，均能在总体权限功能分配和控制之下。



4）代码快速生成

良好的架构使得无论在业务逻辑层、控制器层、Web 界面的 UI 层，均能提供统一的代码逻辑，这些代码均能通过代码生成工具 Database2Sharp 进行生成。Web 界面代码可以充

分利用代码生成工具 Database2Sharp 的元数据信息，实现 Web 界面的快速生成。有效减少出错的几率，提高 Web 界面编码的开发效率和乐趣，更可以使得企业内部的编码模式进行高效的统一。



Enterprise Library 代码生成，可以快速生成除界面外的整体性的框架代码，Bootstrap 的 Web 界面代码生成，可以快速生成基于 Metronic 的 Bootstrap 的前端界面代码和后台控制器代码，界面部分包括查询、分页、数据展示、数据导入导出、新增、编辑、查看、删除等基础功能界面，生成后我们可以基于这个基础上进行简单、快速的修改即可符合实际需要，极大提高我们 Web 界面的开发效率。

5.2. 代码生成工具 Database2Sharp 的整合

整个框架通过与代码生成工具 Database2Sharp 进行配合，能够一键生成整体性框架代码，Web 开发框架的业务逻辑代码和界面代码，开发更高效。

在整个 Web 开发框架中， Database2Sharp 生成出来的代码体现了非常完美的整合性，能够无缝接入开发的框架系统中，无论是常规的业务逻辑和数据访问层代码，以及列表、编辑界面、添加界面、查看详细界面的 Web 界面代码，都能快速生成，稍作调整即可满足业务模块的需要。

Database2Sharp 是一个简单点击几次鼠标就能完成一周代码量的代码生成工具，效率惊人、友好体贴，真正的开发好伴侣。提供了对 SqlServer 2000/2005/2008、Oracle、Mysql、Access、SQLite 的支持；可以生成各种架构代码以及 Web 界面代码，并且和 Web 开发框架完美整合，体现出更高的开发效率。

在开发框架中，代码生成工具 Database2Sharp 可以根据需要生成各种不同框架的代码，极大提高了开发效率和企业内部的编码统一。

基于 MVC 的 Web 界面代码，也可以通过我们提供的代码模板文档，在代码工具中运行生成，复制到 Web 界面项目的 View 视图文件夹中即可运行使用。

5.3. 基于多数据库的数据查询模块和通用查询模块

基于多数据库的数据查询模块和通用高级查询模块，查询数据更方便。

在我的 Web 开发框架中，使用了一个查询辅助类 SearchCondition 来实现查询条件的获取和转化，这个辅助类内置了对多种数据库条件的分析处理，因此能够很好生成所需要的数

据查询条件，正确高效获取所需的数据进行显示。

```
        /// <summary>
        /// 根据查询条件构造查询语句
        /// </summary>
        private string GetConditionSql()
        {
            //如果存在高级查询对象信息，则使用高级查询条件，否则使用主表条件查询
            SearchCondition condition = advanceCondition;
            if (condition == null)
            {
                condition = new SearchCondition();
                condition.AddCondition("ItemName", this.txtName.Text, SqlOperator.Like)
                    .AddCondition("ItemBigType", this.txtBigType.Text, SqlOperator.Like)
                    .AddCondition("ItemType", this.txtItemType.Text, SqlOperator.Like)
                    .AddCondition("Specification", this.cmbSpecNumber.Text, SqlOperator.Like)
                    .AddCondition("MapNo", this.txtMapNo.Text, SqlOperator.Like)
                    .AddCondition("Material", this.txtMaterial.Text, SqlOperator.Like)
                    .AddCondition("Source", this.txtSource.Text, SqlOperator.Like)
                    .AddCondition("Note", this.txtNote.Text, SqlOperator.Like)
                    .AddCondition("Manufacture", this.txtManufacture.Text, SqlOperator.Like)
                    .AddCondition("ItemNo", this.txtItemNo.Text, SqlOperator.LikeStartAt)
                    .AddCondition("WareHouse", this.txtWareHouse.Text, SqlOperator.Like)
                    .AddCondition("Dept", this.txtDept.Text, SqlOperator.Like)
                    .AddCondition("UsagePos", this.txtUsagePos.Text, SqlOperator.Like)
                    .AddCondition("StoragePos", this.txtStoragePos.Text, SqlOperator.Like);
            }
            string where = condition.BuildConditionSql().Replace("Where", "");
            return where;
        }
    /// <summary>
    /// Sql 的查询符号
    /// </summary>
    public enum SqlOperator
    {
        [Description("Like 模糊查询")]
        Like,
        [Description("Not LiKE 模糊查询")]
        NotLike,
        [Description("Like 开始匹配模糊查询，如 Like 'ABC%'")]
        LikeStartAt,
        [Description("= 等于号")]
        Equal,
        [Description("<> (≠) 不等于号")]
```

```
        NotEqual,
        [Description("> 大于号")]
        MoreThan,
        [Description("<小于号")]
        LessThan,
        [Description("≥大于或等于号 ")]
        MoreThanOrEqual,
        [Description("≤ 小于或等于号")]
        LessThanOrEqual,
        [Description("在某个字符串值中")]
        In
    }
```

另外，结合上面的多数据库的数据查询模块，基于 MVC 的 Web 界面，我们可以设计一个通用的分页条件查询操作。

先在客户端的绑定相关条件的值，如果是数值、日期类型的，那么是一个区间的值，用波浪符号~进行分开，并且每个参数以一个特殊的前缀开始，如 WHC_，如下代码所示。

```
//绑定搜索按钮的的点击事件
function BindSearchEvent() {
    $("#btnSearch").click(function () {
        //字段增加WHC_前缀字符，避免传递如URL这样的Request关键字冲突
        var queryData = {
            WHC_SystemType_ID: $("#txtSystemType_ID").combobox('getValue'),
            WHC_LoginName: $("#txtLoginName").val(),
            WHC_FullName: $("#txtFullName").val(),
            WHC_Note: $("#txtNote").val(),
            WHC_IPAddress: $("#txtIPAddress").val(),
            WHC_MacAddress: $("#txtMacAddress").val(),
            WHC_LastUpdated: $("#txtLastUpdated").datebox('getValue') + "~" + $("#txtLastUpdated2").datebox('getValue')
        }
        //将值传递给
        InitGrid(queryData);
        return false;
    });
}
```

在控制器处理端，我们把分页查询的操作，封装在了业务基类 Business<B，T>这个类上面，可以为继承于它的业务控制器类提供了通用的分页查询。下面是控制器基类获取传递过来的参数并构造查询条件的代码。

```
/// <summary>
/// 获取分页操作的查询条件
/// </summary>
protected virtual string GetPagerCondition()
{
    #region 根据数据库字段列，对所有可能的参数进行获值，然后构建查询条件
    SearchCondition condition = new SearchCondition();
    DataTable dt = baseBLL.GetFieldTypeList();
    foreach (DataRow dr in dt.Rows)
    {
        string columnName = dr["ColumnName"].ToString();
        string dataType = dr["DataType"].ToString();
        //字段增加WHC_前缀字符，避免传递如URL这样的Request关键字冲突
        string columnValue = Request["WHC_" + columnName] ?? "";

        if (IsDateTime(dataType))
        {
            condition.AddDateCondition(columnName, columnValue);
        }
        else if (IsNumericType(dataType))
        {
            condition.AddNumberCondition(columnName, columnValue);
        }
        else
        {
            condition.AddCondition(columnName, columnValue, SqlOperator.Like);
        }
    }
    #endregion

    return condition.BuildConditionSql().Replace("Where", "");
}
```

由于我一直希望我的 Web 开发框架能够精益求精，所以设计了这个通用查询模块，希望使用该 Web 开发框架的客户，能够快速、高效地实现数据的查询。

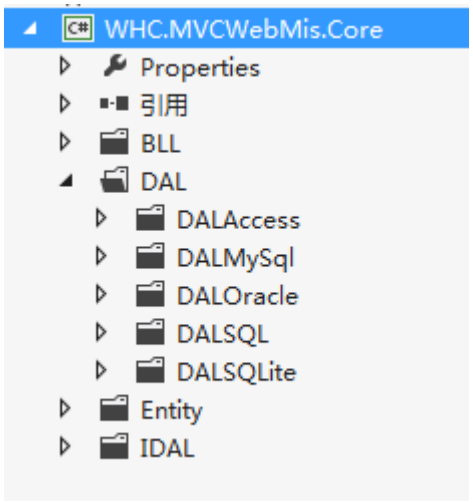
5.4. 框架提供基于多种数据库的整合

框架提供基于多种数据库（Sqlserver/Oracle/Mysql/Sqlite/Access)的整合。

虽然我们在实际项目中，一般采用一种数据库进行处理，但是不同的项目，采用的数据库类型可能不同，本 Web 开发框架为了方便演示和扩展的需要，内置支持了 Sqlserver/Oracle/Mysql/Sqlite/Access，更多的数据库，也可以通过扩展数据库访问基类的方式进行更多数据库的支持。

Web 开发框架里面的所有模块，如用到了数据存储的，如权限管理管理模块、通用数据字典管理模块，均内置支持这几种数据库的整合支持。整个 Web 开发框架的数据库访问，能够手动配置数据库类型，对于同一种数据库，也可以把数据存储分开存储，如业务数据存

储在一个数据库，权限管理控制存储在另外一个数据库这种方式。



Web 开发框架提供多种数据库支持，但数据访问基类依然很精简，因为我们利用的数据库访问模块是 Enterprise Library，把数据库抽象化，并且我把所有数据库通用操作放在了一个超级基类上，具体的数据库基类只需要实现变化的部分即可，业务访问类则使用泛型进行封装处理。

因此，Web 开发框架提供了高度封装的数据访问基类，开发代码更少更高效。