第 3 章

系 统 架 构

在进行具有一定规模的数据库应用系统开发时，如果仍采用传统的代码编写方式，将一个功能模块的所有代码都写在 Form 中的做法是非常不合适的。因为以这种方式编写的代码，在面临需求变更时，所引发的修改往往是灾难性的。即使是一个数据库 IP 地址的变更都可能会导致数十甚至上百处的代码变化（每个涉及连接数据库的代码都要修改连接字符串）。所有的代码堆砌在一个窗体内，职责繁杂造成代码可读性、可维护性、可移植性差，违反了单一职责原则；在面临需求变更时，必须通读所有代码，再在合适的位置做修改，违反了对修改关闭、对扩展开放的开放-关闭原则；所有代码没有任何抽象，不对接口编程，代码耦合度太高，无法适应较大的需求变更，违反了依赖倒转原则。

例如，第 8 章的在线考试系统属于中小型应用系统，该系统分为两个子系统：教师端子系统和学生端子系统。首先，两个子系统都访问同一个数据库，其中有许多访问数据库的操作及业务逻辑都是相同的，如果再像传统的编写方式一样将代码都放在各个 Form 中，那就无法实现代码的复用，必然造成大量的冗余，为后续扩展和维护带来麻烦。此外，本在线考试系统面对的客户不同，其对于软件产品的经济投入也会不同，必然涉及不同的用户采用不同类型数据库的情况，如果代码不做到充分解耦，在面对数据库类型变更时，几乎要把所有访问数据库的代码修改一遍，显然是不现实的。

采用一些典型架构开发软件，其目的就是为了使软件代码更具可维护性、可扩展性、可复用性，从而令系统更为灵活，可以适应任何合理的需求变更。

3.1 三层架构简介

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，即三层架构（3-Tier Application），这三层从下至上分别为：数据访问层（Data Access Layer，DAL）、业务逻辑层（Business Logic Layer，BLL）以及表示层（User Interface，UI），如图 3-1 所示。

1. 数据访问层

数据访问层又称持久层，该层负责访问数据库，通俗点讲就是该层负责实现对数据库各表的增、删、改、查操作，当然数据存储方式不一定是数据库，也可能是文本文件、XML 文档等。该层不负责任何业务逻辑的处理，更不涉及任何界面元素。

2. 业务逻辑层

业务逻辑层又称领域层。该层是整个系统的核心，负责处理所有的业务流程，从简单的数据有效性验证到复杂的对一整条业务链的处理。例如，商城购物，从查询商品到添加购物

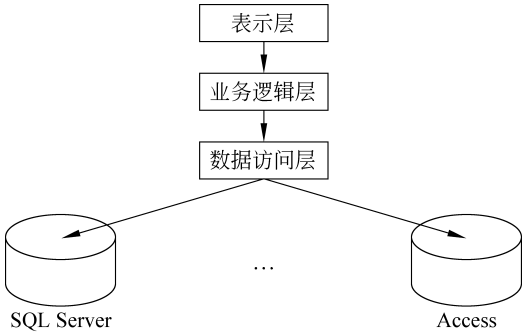

图 3-1　三层架构示意图

车，再到下订单，直至付款结束等过程。当然，不排除个别软件项目业务逻辑简单，导致业务逻辑层代码较少的情况。例如，本章由于篇幅所限，所提供的示例业务逻辑简单，就会出现业务逻辑层“瘦小”的现象。当业务逻辑层需要访问数据库时，它会通过调用数据访问层来实现，而不直接访问数据库。这样可以使业务逻辑层的实现与具体数据库无关，从而有效解耦。业务逻辑层同样不涉及任何界面元素。

3. 表示层

表示层即用户界面，表示层可以是 WinForm、WebForm 甚至是控制台，该层负责用户与系统的交互，接收用户的输入及事件触发。理想状态下，该层不应包含系统的业务逻辑，即使有逻辑代码，也应只与界面元素有关。例如，根据用户的身份控制按钮的可用性等。具体的业务逻辑可通过调用业务逻辑层来完成，该层不能直接调用数据访问层，更不能直接访问数据库。

如此分层具有以下优点。

（1）分散关注：开发人员可以只关注自己所负责一层的技术实现。例如，负责数据访问层的开发人员，可以不需要关心系统的任何业务逻辑，更不用关心界面的设计。只需要关心所访问的数据库类型及表结构，最大限度地实现对各数据表的增、删、改、查操作即可。如此，对于开发人员的技术要求可以降到最低，项目经理也可以根据团队成员的专长，合理为其分配擅长的领域工作。

（2）松散耦合：三层之间呈线性调用，业务逻辑层的实现不依赖于数据访问层的具体实现，表示层的实现同样不依赖于业务逻辑层的具体实现，可以很容易用新的实现来替换原有层次的实现，而不会对其他层造成影响。

（3）逻辑复用：个别层代码所生成的组件（动态链接库文件 DLL）可以直接被其他项目所使用。例如，某系统最初只有 WinForm 版本，随着业务扩展，逐渐有了 Web 版、手机版等需求，但各版本功能一致，业务逻辑相同。这样在新建 Web 版和手机版项目时，只需将原 WinForm 版项目中的业务逻辑层和数据访问层组件引用到新项目中，直接使用即可，无须再重写代码。

当然，在获得优点的同时，分层也不可避免地会付出一些“代价”，其缺点如下。

（1）降低了系统性能：原本在不采用分层的情况下，UI 可以直接访问数据库，但现在要通过层层调用才能达到同样的目的，必然会在运行效率上有所降低。

（2）容易导致级联修改：用户某些需求的变化，如用户觉得某个功能模块目前所维护的信息不够，需要再加入一些信息，势必导致 UI 的变化，为了存储这些数据也会导致相应数据表字段的增加，从而数据访问层和业务逻辑层都会受到影响，这就是级联修改。

3.2　简单三层架构

简单三层架构即基本三层架构，其结构如图 3-1 所示。现以一个小示例介绍简单三层架构的代码编写方式。

示例所用数据库及业务需求如下。

要求设计一个通讯录软件，实现对联系人类型的定制以及联系人的管理，即实现对联系人类型和联系人的增、删、改、查功能。

数据库名为 MyDb，其中有两个数据表，名为 LinkmanType（联系人类型）表和 Linkmen（联系人）表，表结构分别如表 3-1 和表 3-2 所示。

表 3-1　LinkmanType（联系人类型）表结构

字　段　名	数 据 类 型	长度	主键	含　　义
typeId	nvarchar	20	是	联系人类型编号
typeName	nvarchar	20	否	联系人类型名称

表 3-2　Linkmen（联系人）表结构

字　段　名	数据类型	长度	主键	含　　义
lkmId	int	4	是	联系人编号（自动加 1）
lkmName	nvarchar	10	否	联系人姓名
lkmMPNum	nvarchar	12	否	联系人移动电话
lkmOPNum	nvarchar	20	否	联系人办公室电话
lkmEmail	nvarchar	30	否	联系人电子邮箱
lkmCompName	nvarchar	50	否	联系人单位名称
typeId	nvarchar	20	否	联系人类型编号

3.2.1　数据访问层

1. 创建一个空解决方案

打开 Visual Studio 2019，单击“创建新项目”，在弹出的“创建新项目”对话框中的“搜索模板”搜索框内输入“空白解决方案”，查询“空白解决方案”模板，如图 3-2 所示。选择查询到的“空白解决方案”模板，单击“下一步”按钮，打开“配置新项目”窗口，修改项目名称为“ThreeLayer”，选择好位置，单击“创建”按钮，完成空解决方案的创建，如图 3-3 所示。

2. 创建数据访问层项目 DAL

在图 3-3 中右侧的“解决方案资源管理器”子窗口中的“解决方案 ThreeLayer”节点上单

图 3-2 “创建新项目”对话框

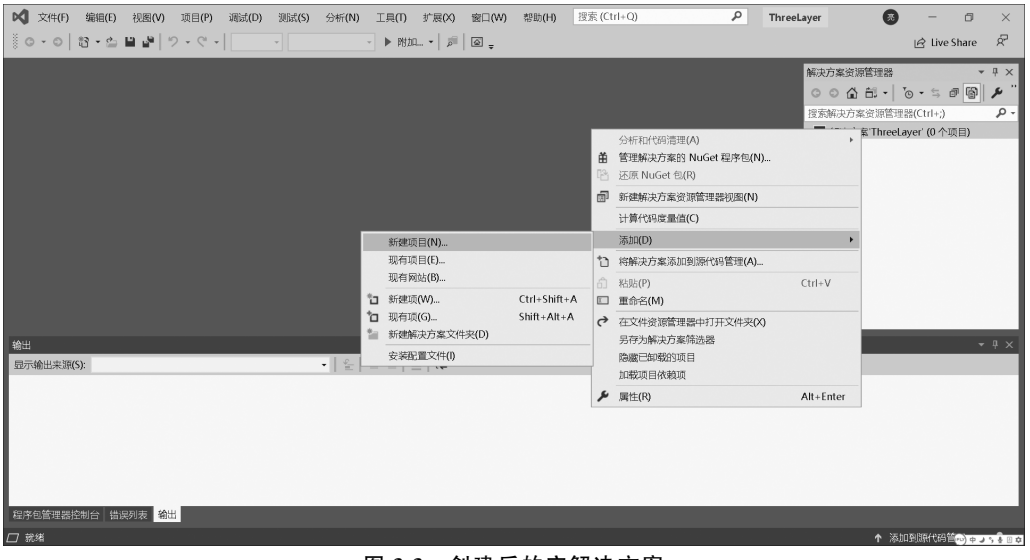

图 3-3 创建后的空解决方案

击鼠标右键，选择“添加”→“新建项目”，会再次弹出类似图 3-2 的“添加新项目”对话框，在其中搜索“类库”模板，选择“类库(.NET Framework)”模板，单击“下一步”按钮，将项目名称改为“DAL”，单击“创建”按钮，此时，图 3-3 的解决方案会变成如图 3-4 所示状态。

3. 编写数据访问层代码

右击 DAL 项目中的 Class1.cs 文件，在弹出菜单中选择“删除”命令，将 DAL 默认创建的 Class1.cs 文件删除。右击 DAL 项目，在弹出菜单中选择“添加”→“类”，弹出如图 3-5 所

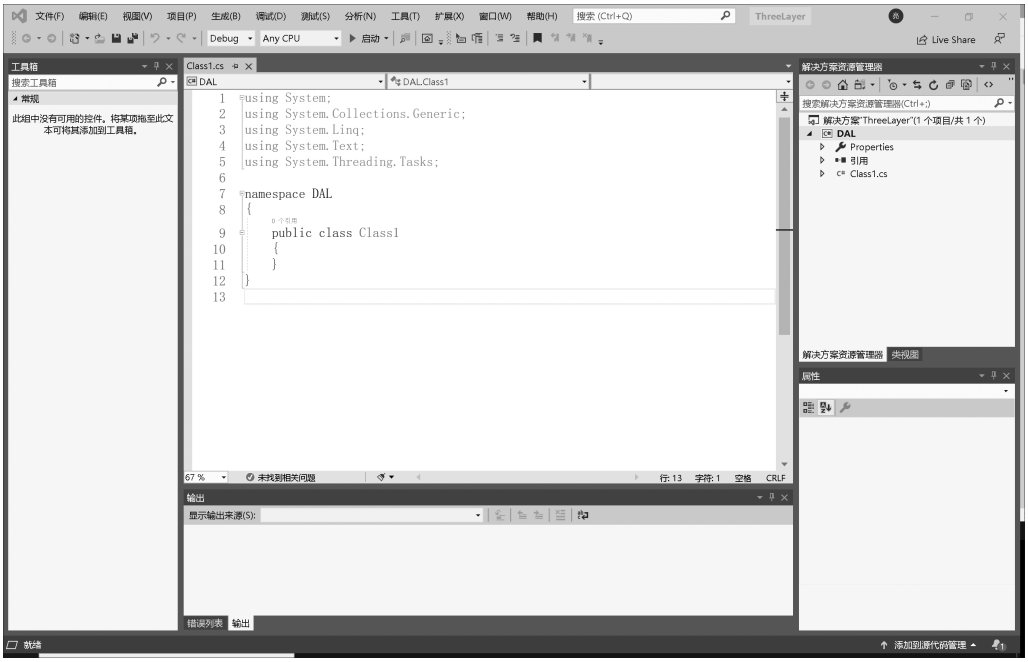

图 3-4　创建 DAL 项目之后的解决方案

示的“添加新项”对话框。

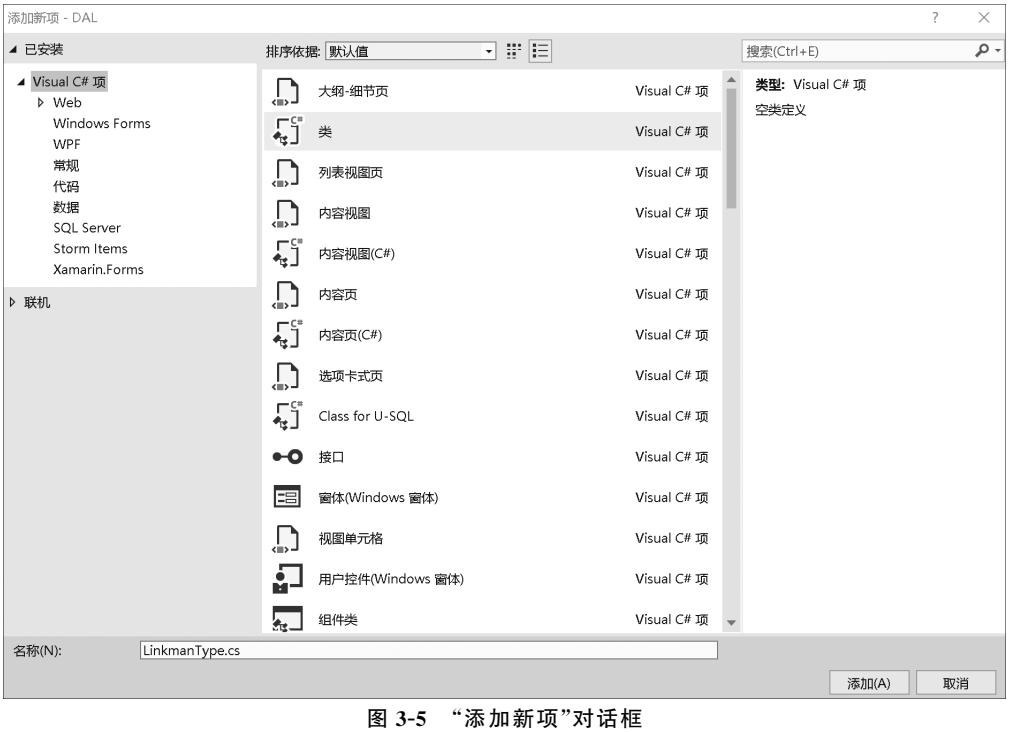

图 3-5　“添加新项”对话框

在图 3-5 中，将名称命名为“LinkmanType.cs”，单击“添加”按钮，在 DAL 项目中，会出现名为“LinkmanType.cs”的类文件。

在 3.1 节提到数据访问层的任务就是实现对数据库的操作，实现对各数据表数据的增加、删除、修改、查询。作为 LinkmanType 类，其功能就是要实现对 LinkmanType 数据表的增加、删除、修改、查询操作。

DAL 项目中的 LinkmanType.cs 文件代码如下。

```
namespace DAL          //需要手动引用 System.Data 和 System.Data.SqlClient 命名空间
{
  //LinkmanType 类默认无访问修饰符，此处必须声明为 public，因 BLL 在调用 DAL 中该类时，
  //属于跨项目访问，只有声明为 public 才能够被 BLL 所访问。
  public class LinkmanType
  {
      ///<summary>
      ///数据库连接字符串，server 为数据库服务器的名字或 IP 地址，database 为数据库的名字，
      ///uid 为数据库用户名，pwd 为该用户的密码
      ///</summary>
      public string connString = "server=(local);database=mydb;uid=test;pwd=
          test";
      ///<summary>
      ///插入数据的方法
      ///</summary>
      ///<param name="typeId">要插入的联系人类型编号</param>
      ///<param name="typeName">要插入的联系人类型名称</param>
      ///<returns>插入成功返回 true，插入失败返回 false</returns>
      public bool insert(string typeId,string typeName)
      {
          try
          {
              SqlConnection conn = new SqlConnection();
              conn.ConnectionString =connString ;
              conn.Open();
              SqlCommand cmd = new SqlCommand();
              cmd.Connection = conn;
              cmd.CommandText="insert into linkmantype(typeid,typename)
                               values(@typeid,@typename)";   //参数形式的 SQL 语句
              //以下两句为 SQL 参数赋值
              cmd.Parameters.Add(new SqlParameter("@typeid", typeId));
              cmd.Parameters.Add(new SqlParameter("@typename", typeName));
              cmd.ExecuteNonQuery();
              conn.Close();
              return true;
          }
          catch
          {
              return false;
          }
      }
      ///<summary>
      ///修改数据的方法
      ///</summary>
      ///<param name="typeId">要修改的联系人类型新编号</param>
      ///<param name="typeName">要修改的联系人类型新名称</param>
      ///<param name="oldTypeId">要修改的联系人类型原编号</param>
```

```
///<returns>修改成功返回 true,修改失败返回 false</returns>
public bool update(string typeId,string typeName,string oldTypeId)
{
    try
    {
        SqlConnection conn = new SqlConnection();
        conn.ConnectionString = connString;
        conn.Open();
        SqlCommand cmd = new SqlCommand();
        cmd.Connection = conn;
        cmd.CommandText="update linkmantype set typeid=@typeid,
                         typename=@typename where typeid=@oldTypeId";
        cmd.Parameters.Add(new SqlParameter("@typeid", typeId));
        cmd.Parameters.Add(new SqlParameter("@typename", typeName));
        cmd.Parameters.Add(new SqlParameter("@oldTypeId", oldTypeId));
        cmd.ExecuteNonQuery();
        conn.Close();
        return true;
    }
    catch
    {
        return false;
    }
}
///<summary>
///删除数据的方法
///</summary>
///<param name="typeId">要删除的联系人类型编号</param>
///<returns>删除成功返回 true,删除失败返回 false</returns>
public bool delete(string typeId)
{
    try
    {
        SqlConnection conn = new SqlConnection();
        conn.ConnectionString = connString;
        conn.Open();
        SqlCommand cmd = new SqlCommand();
        cmd.Connection = conn;
        cmd.CommandText = "delete from linkmantype where typeid=@typeid";
        cmd.Parameters.Add(new SqlParameter("typeid", typeId));
        cmd.ExecuteNonQuery();
        conn.Close();
        return true;
    }
    catch
    {
        return false;
    }
}
///<summary>
///查询数据的方法
///</summary>
///<param name="strWhere">strWhere 为空字符串时,代表查询表中所有数据
```

```
        ///为非空时应传入查询依据,即 where 子句的内容,如"typeName='朋友'"等</param>
        ///<returns>返回 null 代表查询失败,返回非 null 代表查询成功
        ///且结果存于返回的 DataTable 中</returns>
        public DataTable select(string strWhere)
        {
            try
            {
                string sql = "select * from linkmantype";
                if (strWhere != "")
                {
                    sql += " where " + strWhere;
                }
                SqlConnection conn = new SqlConnection();
                conn.ConnectionString = connString;
                SqlDataAdapter da = new SqlDataAdapter(sql, conn);
                DataSet ds = new DataSet();
                da.Fill(ds);
                return ds.Tables[0];
            }
            catch
            {
                return null;
            }
        }
    }
}
```

接下来实现 Linkmen 表的数据访问层代码。采用与创建 LinkmanType.cs 类文件同样的方法,在 DAL 项目中创建 Linkmen.cs 类文件。在其中编写对 Linkmen 表的访问代码如下。

```
namespace DAL          //需要手动引用 System.Data 和 System.Data.SqlClient 命名空间
{
    public class Linkmen
    {
        public string connString = "server=(local);database=mydb;uid=test;
            pwd=test";
        ///<summary>
        ///插入数据的方法
        ///</summary>
        ///<param name="lkmName">姓名</param>
        ///<param name="lkmMPNum">移动电话</param>
        ///<param name="lkmOPNum">固定电话</param>
        ///<param name="lkmEmail">电子邮箱</param>
        ///<param name="lkmCompName">单位名称</param>
        ///<param name="typeId">联系人类型编号</param>
        ///<returns>插入成功返回 true,插入失败返回 false</returns>
        public bool insert(string lkmName,string lkmMPNum,string lkmOPNum,
                    string lkmEmail,string lkmCompName,string typeId)
        {
            try
            {
                SqlConnection conn = new SqlConnection();
```

```
            conn.ConnectionString = connString;
            conn.Open();
            SqlCommand cmd = new SqlCommand();
            cmd.Connection = conn;
            cmd.CommandText = "insert into linkmen(lkmname, lkmMPNum, lkmOPNum,
                        lkmEmail, lkmCompName, typeId) values(@lkmname,
                        @lkmMPNum, @lkmOPNum, @lkmEmail, @lkmCompName,
                        @typeId)";
            cmd.Parameters.Add(new SqlParameter("@lkmname", lkmName));
            cmd.Parameters.Add(new SqlParameter("@lkmMPNum", lkmMPNum));
            cmd.Parameters.Add(new SqlParameter("@lkmOPNum", lkmOPNum));
            cmd.Parameters.Add(new SqlParameter("@lkmEmail", lkmEmail));
            cmd.Parameters.Add(new SqlParameter("@lkmCompName", lkmCompName));
            cmd.Parameters.Add(new SqlParameter("@typeId", typeId));
            cmd.ExecuteNonQuery();
            conn.Close();
            return true;
        }
        catch
        {
            return false;
        }
    }
    ///<summary>
    ///修改数据的方法
    ///</summary>
    ///<param name="lkmId">联系人编号</param>
    ///<param name="lkmName">姓名</param>
    ///<param name="lkmMPNum">移动电话</param>
    ///<param name="lkmOPNum">固定电话</param>
    ///<param name="lkmEmail">电子邮箱</param>
    ///<param name="lkmCompName">单位名称</param>
    ///<param name="typeId">联系人类型编号</param>
    ///<returns>修改成功返回 true,修改失败返回 false</returns>
    public bool update(string lkmId,string lkmName, string lkmMPNum,
        string lkmOPNum,string lkmEmail, string lkmCompName, string typeId)
    {
        try
        {
            SqlConnection conn = new SqlConnection();
            conn.ConnectionString = connString;
            conn.Open();
            SqlCommand cmd = new SqlCommand();
            cmd.Connection = conn;
            cmd.CommandText = "update linkmen set lkmname=@lkmname,
                            lkmMPNum=@lkmMPNum, lkmOPNum=@lkmOPNum,
                            lkmEmail=@lkmEmail, lkmCompName=@lkmCompName,
                            typeId=@typeId where lkmid=@lkmid";
            cmd.Parameters.Add(new SqlParameter("@lkmname", lkmName));
            cmd.Parameters.Add(new SqlParameter("@lkmMPNum", lkmMPNum));
            cmd.Parameters.Add(new SqlParameter("@lkmOPNum", lkmOPNum));
            cmd.Parameters.Add(new SqlParameter("@lkmEmail", lkmEmail));
```

```
            cmd.Parameters.Add(new SqlParameter("@lkmCompName", lkmCompName));
            cmd.Parameters.Add(new SqlParameter("@typeId", typeId));
            cmd.Parameters.Add(new SqlParameter("@lkmid", lkmId));
            cmd.ExecuteNonQuery();
            conn.Close();
            return true;
        }
        catch
        {
            return false;
        }
    }
    ///<summary>
    ///删除数据的方法
    ///</summary>
    ///<param name="lkmId">要删除的联系人的编号</param>
    ///<returns>删除成功返回 true,删除失败返回 false</returns>
    public bool delete(string lkmId)
    {
        try
        {
            SqlConnection conn = new SqlConnection();
            conn.ConnectionString = connString;
            conn.Open();
            SqlCommand cmd = new SqlCommand();
            cmd.Connection = conn;
            cmd.CommandText = "delete from linkmen where lkmid=@lkmid";
            cmd.Parameters.Add(new SqlParameter("@lkmid", lkmId));
            cmd.ExecuteNonQuery();
            conn.Close();
            return true;
        }
        catch
        {
            return false;
        }
    }
    ///<summary>
    ///查询数据的方法
    ///</summary>
    ///<param name="strWhere">strWhere 为空字符串时,代表查询表中所有数据
    ///为非空时应传入查询依据,即 where 子句的内容,如"lkmName='张三'"等</param>
    ///<returns>返回 null 代表查询失败,返回非 null 代表查询成功
    ///且结果存于返回的 DataTable 中</returns>
    public DataTable select(string strWhere)
    {
        try
        {
            string sql = "select linkmen.typeid as typeid, typename, lkmname,
                lkmid,lkmMPNum, lkmOPNum, lkmEmail, lkmCompName from linkmen
                left join linkmantype on linkmen.typeid=linkmantype.typeid";
            if (strWhere != "")
```

```
                {
                    sql += " where " + strWhere;
                }
                SqlConnection conn = new SqlConnection();
                conn.ConnectionString = connString;
                SqlDataAdapter da = new SqlDataAdapter(sql, conn);
                DataSet ds = new DataSet();
                da.Fill(ds);
                return ds.Tables[0];
            }
            catch
            {
                return null;
            }
        }
    }
}
```

上述代码为数据访问层的最基本实现，但还存在如下诸多不足。

(1) 增加、删除、修改功能的代码基本类似，存在大量的冗余。事实上，除了所执行的 SQL 语句以及所传递的 SQL 参数不同外，其他代码是一样的。此外，各类的查询功能实现也极其相似，只是 Select 语句不同罢了。再者，每个类中都维护着同一个数据库连接字符串，不但冗余，更会给数据库迁移带来莫大的灾难，每次数据库迁移，都要修改每个类的连接字符串，何其麻烦！因此，完全可以对这些冗余的代码进行精简。

(2) 以各类的 insert 函数为例，操作联系人类型表的 LinkmanType 类中的 insert 函数有两个形参：typeId 和 typeName，分别代表要插入类型的两个字段，而操作联系人表的 Linkmen 类中的 insert 函数则有 6 个参数：lkmName、lkmMPNum、lkmOPNum、lkmEmail、lkmCompName 和 typeId，可见，函数的形参个数与数据表的字段数是成正比的，如果一个数据表的字段数过多，势必导致相关函数的形参数过于冗长，非常不利于调用和规范化管理。

3.2.2　数据访问通用类库

解决以上第一个不足，精简冗余的数据库访问代码，可考虑再创建一个数据库访问辅助类库，将重复的代码进行封装。

1. 创建数据访问通用类库 DBUtility

右击解决方案，单击“添加”→“新建项目”，创建一个名为“DBUtility”的类库项目。添加项目之后的解决方案列表如图 3-6 所示。

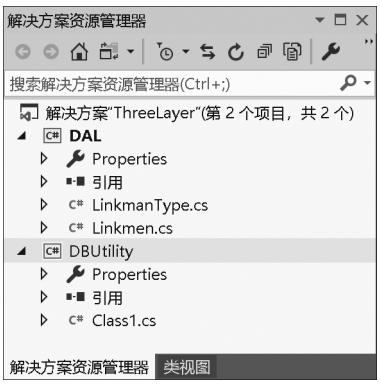

图 3-6　添加 DBUtility 类库后的解决方案

2. 编写数据访问通用类库中的类代码

在图 3-6 中，删除 DBUtitlity 项目中的 Class1.cs 类文件，新建一个名为“DbHelperSQL.cs”的类文件，DbHelperSQL.cs 类代码如下。

```
namespace DBUtility   //需要手动引用命名空间 System.Data 和 System.Data.SqlClient
{
    public class DbHelperSQL
    {
        public static string connString = "server=(local);database=mydb;
            uid=test;pwd=test";
        ///<summary>
        ///数据增、删、改要调用的通用方法
        ///</summary>
        ///<param name="sql">要执行的 SQL 语句</param>
        ///<param name="sqlParams">所有要传给 SQL 语句的参数集合</param>
        ///<returns>返回 true 表示执行成功,false 为执行失败</returns>
        public static bool ExecuteSql(string sql,List<SqlParameter> sqlParams)
        {
            try
            {
                SqlConnection conn = new SqlConnection();
                conn.ConnectionString = connString;
                conn.Open();
                SqlCommand cmd = new SqlCommand();
                cmd.Connection = conn;
                cmd.CommandText = sql;
                //遍历传过来的 SQL 参数集合,将其逐一加到 SqlCommand 对象的参数集合中
                for(int i=0;i<sqlParams.Count ;i++)
                    cmd.Parameters.Add(sqlParams[i]);
                cmd.ExecuteNonQuery();
                conn.Close();
                return true;
            }
            catch
            {
                return false;
            }
        }
        ///<summary>
        ///查询数据的通用方法
        ///</summary>
        ///<param name="sql">select 语句</param>
        ///<returns>返回 null 代表查询失败,返回非 null 代表查询成功
        ///且结果存于返回的 DataTable 中</returns>
        public static DataTable Query(string sql)
        {
            try
            {
                SqlConnection conn = new SqlConnection();
                conn.ConnectionString = connString;
                SqlDataAdapter da = new SqlDataAdapter(sql, conn);
                DataSet ds = new DataSet();
                da.Fill(ds);    //将 sql 语句的查询结果填充到 ds 中
                return ds.Tables[0];
              }
              catch
              {
```

```
                return null;
            }
        }
    }
}
```

如此一来，数据访问层 DAL 项目中的各函数，只需要调用 DBUtility 中的函数即可实现对数据库的访问。

3. 为数据访问层项目 DAL 添加引用

要在 DAL 中调用 DBUtility 中的类和函数，需将 DBUtility 项目引入到 DAL 项目中：右击 DAL 项目节点下的“引用”节点，在弹出式菜单中选择“添加引用”命令，会弹出如图 3-7 所示的“引用管理器”对话框。

图 3-7　“引用管理器”对话框

在图 3-7 中，选择“项目”→“解决方案”选项，勾选 DBUtility 项目，单击“确定”按钮，会发现 DAL 项目的引用列表中，多出一个名为“DBUtility”的引用，如图 3-8 所示。

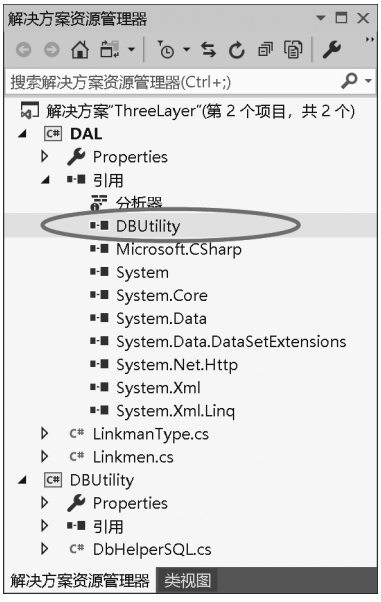

图 3-8　添加引用后的 DAL 引用列表

4. 修改 3.2.1 节中的数据访问层代码

现在，可以在 DAL 项目中调用 DBUtility 中所有公开（public）的类和函数。以 Linkmen 类的 insert 函数为例，此时，insert 函数可通过调用 DBUtility 的 DbHelperSQL 类中的 ExecuteSql 函数来实现数据的插入，而无须书写重复冗余的代码，修改后的 insert 函数如下。

```
public bool insert(string lkmName,string lkmMPNum,string lkmOPNum,
            string lkmEmail,string lkmCompName,string typeId)
{
    string sql = "insert into linkmen(lkmname,lkmMPNum,lkmOPNum,lkmEmail,
            lkmCompName, typeId)values(@lkmname, @lkmMPNum, @lkmOPNum,
            @lkmEmail, @lkmCompName, @typeId)";
    List<SqlParameter> sqlParams = new List<SqlParameter>();
    //以下六句代码把要传给 SQL 语句的参数存入集合，以便传递给函数
    sqlParams.Add(new SqlParameter("@lkmname", lkmName));
    sqlParams.Add(new SqlParameter("@lkmMPNum", lkmMPNum));
    sqlParams.Add(new SqlParameter("@lkmOPNum", lkmOPNum));
    sqlParams.Add(new SqlParameter("@lkmEmail", lkmEmail));
    sqlParams.Add(new SqlParameter("@lkmCompName", lkmCompName));
    sqlParams.Add(new SqlParameter("@typeId", typeId));
    //调用通用类库中的 ExecuteSql 方法执行 SQL 语句
    return DBUtility.DbHelperSQL.ExecuteSql(sql, sqlParams);
}
```

由上述代码可见，在 insert 函数中，已无须再编写 SqlConnection、SqlCommand 对象的创建和属性赋值、方法调用等代码。同样，update、delete 等函数也不必再写这些重复代码，代码复用度更高。此外，由于连接字符串已写在了 DbHelperSQL 类中，不会在其他类中出现，对于数据库迁移所造成的修改量降到了最低——只需修改 DbHelperSQL 中的连接字符串即可。完整的数据访问层代码将在解决第二个不足之后给出。

3.2.3 实体类库

针对第二个不足——函数参数个数因数据表字段过多而变得复杂的问题，可以利用面向对象的思想，将数据表封装成类。如对于 LinkmanType 表，可以定义 Model.LinkmanType 类，其中包括两个成员变量 typeId 和 typeName，分别对应 LinkmanType 表的两个字段。而对于 Linkmen 表，可以定义 Model.Linkmen 类，其中包含 7 个成员变量 lkmId、lkmName、lkmMPNum、lkmOPNum、lkmEmail、lkmCompName 和 typeId，分别对应 Linkmen 表的 7 个字段。这样在设计对应的 insert 和 update 函数时，可以用对应的类类型来作为形参，例如 Linkmen 类的 insert 函数，只需用以下方式定义即可。

```
public bool insert(Model.Linkmen mLKM);
```

如此大大缩减了形参数量，简化了函数结构。

可以创建一个专门的类库，用于设计这些针对数据表结构而产生的类，我们称为实体（Model）类库。

1. 创建实体类库 Model

右击解决方案，单击“添加”→“新建项目”，创建一个名为“Model”的类库项目。添加项

目之后的解决方案列表如图 3-9 所示。

图 3-9　添加 Model 之后的解决方案资源管理器

2. 编写实体类库中的类代码

在图 3-9 中，删除 Model 项目中的 Class1.cs 类文件，新建两个分别名为“LinkmanType.cs”和“Linkmen.cs”的类文件，各自代码如下。

LinkmanType.cs 类代码如下。

```
namespace Model
{
    public class LinkmanType
    {
        public string typeId;
        public string typeName;
    }
}
```

Linkmen.cs 类代码如下。

```
namespace Model
{
    public class Linkmen
    {
        public string lkmId;
        public string lkmName;
        public string lkmMPNum;
        public string lkmOPNum;
        public string lkmEmail;
        public string lkmCompName;
        public string typeId;
    }
}
```

3. 修改数据访问层代码

首先，与引用 DBUtility 项目相同的方法，将 Model 项目引入到 DAL 项目中。

修改 LinkmanType.cs 类文件如下。

```
namespace DAL              //需要引入 System.Data 和 System.Data.SqlClient 命名空间
{
    public class LinkmanType
    {
        public bool insert(Model.LinkmanType mLKMT)
        {
            string sql= "insert into linkmantype(typeid,typename)values
                (@typeid,@typename)";
            List<SqlParameter > sqlParams=new List<SqlParameter>();
            sqlParams.Add(new SqlParameter("@typeid",  mLKMT.typeId));
            sqlParams.Add(new SqlParameter("@typename", mLKMT.typeName));
            return DBUtility.DbHelperSQL.ExecuteSql(sql, sqlParams);
        }
        public bool update(Model.LinkmanType mLKMT,string oldTypeId)
        {
            string sql= "update linkmantype set typeid=@typeid,
                          typename=@typename where typeid=@oldTypeId";
            List<SqlParameter > sqlParams=new List<SqlParameter>();
            sqlParams.Add(new SqlParameter("@typeid", mLKMT.typeId));
            sqlParams.Add(new SqlParameter("@typename", mLKMT.typeName));
            sqlParams.Add(new SqlParameter("@oldTypeId", oldTypeId));
            return DBUtility.DbHelperSQL.ExecuteSql(sql, sqlParams);
        }
        public bool delete(string typeId)
        {
            string sql= "delete from linkmantype where typeid=@typeid";
            List<SqlParameter> sqlParams=new List<SqlParameter>();
            sqlParams.Add(new SqlParameter("typeid",typeId));
            return DBUtility.DbHelperSQL.ExecuteSql(sql, sqlParams);
        }
        public DataTable select(string strWhere)
        {
            string sql = "select * from linkmantype";
            if (strWhere != "")
            {
                sql += " where " + strWhere;
            }
            return DBUtility.DbHelperSQL.Query(sql);
        }
    }
}
```

修改 Linkmen 类文件如下。

```
namespace DAL
{
    public class Linkmen
    {
        public bool insert(Model.Linkmen mLKM)
        {
            string sql = "insert into linkmen(lkmname,lkmMPNum,lkmOPNum,
                lkmEmail, lkmCompName,typeId)values(@lkmname,@lkmMPNum,@lkmOPNum,
```

```
                @lkmEmail,@lkmCompName,@typeId)";
            List<SqlParameter> sqlParams = new List<SqlParameter>();
            sqlParams.Add(new SqlParameter("@lkmname", mLKM.lkmName));
            sqlParams.Add(new SqlParameter("@lkmMPNum", mLKM.lkmMPNum));
            sqlParams.Add(new SqlParameter("@lkmOPNum", mLKM.lkmOPNum));
            sqlParams.Add(new SqlParameter("@lkmEmail", mLKM.lkmEmail));
            sqlParams.Add(new SqlParameter("@lkmCompName", mLKM.lkmCompName));
            sqlParams.Add(new SqlParameter("@typeId", mLKM.typeId));
            return DBUtility.DbHelperSQL.ExecuteSql(sql, sqlParams);
        }
        public bool update(Model.Linkmen mLKM)
        {
            string sql = "update linkmen set lkmname=@lkmname,
                lkmMPNum=@lkmMPNum,lkmOPNum=@lkmOPNum,lkmEmail=@lkmEmail,
                lkmCompName=@lkmCompName,typeId=@typeId where lkmid=@lkmid";
            List<SqlParameter> sqlParams = new List<SqlParameter>();
            sqlParams.Add(new SqlParameter("@lkmname", mLKM.lkmName));
            sqlParams.Add(new SqlParameter("@lkmMPNum", mLKM.lkmMPNum));
            sqlParams.Add(new SqlParameter("@lkmOPNum", mLKM.lkmOPNum));
            sqlParams.Add(new SqlParameter("@lkmEmail", mLKM.lkmEmail));
            sqlParams.Add(new SqlParameter("@lkmCompName", mLKM.lkmCompName));
            sqlParams.Add(new SqlParameter("@typeId", mLKM.typeId));
            sqlParams.Add(new SqlParameter("@lkmid", mLKM.lkmId));
            return DBUtility.DbHelperSQL.ExecuteSql(sql, sqlParams);
        }
        public bool delete(string lkmId)
        {
            string sql= "delete from linkmen where lkmid=@lkmid";
            List<SqlParameter> sqlParams = new List<SqlParameter>();
            sqlParams.Add(new SqlParameter("@lkmid", lkmId));
            return DBUtility.DbHelperSQL.ExecuteSql(sql, sqlParams);
        }
        public DataTable select(string strWhere)
        {
            string sql = "select linkmen.typeid as typeid,typename,lkmname,
                lkmid,lkmMPNum, lkmOPNum,lkmEmail,lkmCompName from linkmen
                left join linkmantype on linkmen.typeid=linkmantype.typeid";
            if (strWhere != "")
            {
                sql += " where " + strWhere;
            }
            return DBUtility.DbHelperSQL.Query(sql);
        }
    }
}
```

上述 LinkmanType 类和 Linkmen 类，明显较上一版本要精简得多，函数实现得到精简，函数参数也得到了精简。此代码为本书简单三层架构终态的数据访问层代码结构。其中，DAL 调用了 DBUtility 项目的代码，但 DBUtility 类库不能称为一层，它的存在只是提供一套通用的访问数据库的方法。而 Model 类库更不能称为一层，在后面章节中可以看

到，Model 实际上是贯穿了数据访问层、业务逻辑层和表示层这三层的，因为三层均要调用 Model 中的类。

3.2.4 业务逻辑层

1. 创建业务逻辑层项目 BLL

右击解决方案，单击“添加”→“新建项目”，创建一个名为“BLL”的类库项目。添加项目之后的解决方案列表如图 3-10 所示。

图 3-10 添加 BLL 之后的解决方案资源管理器

2. 编写业务逻辑层代码

先引用 DAL 和 Model 项目，删除 BLL 项目中的 Class1.cs 文件，然后新建名为“LinkmanType.cs”的类文件，以实现联系人类型管理的业务逻辑。联系人类型管理的业务逻辑非常简单，只有基本的对联系人类型的增加、删除、修改、查询功能。因此，BLL 的 LinkmanType 类中的代码也只是简单地调用 DAL 中的 LinkmanType 类来实现这些功能。

LinkmanType.cs 类文件代码如下。

```
namespace BLL
{
    public class LinkmanType
    {
        DAL.LinkmanType dLKMT = new DAL.LinkmanType();
        public bool insert(Model.LinkmanType mLKMT)
        {
            return dLKMT.insert(mLKMT);
        }
        public bool update(Model.LinkmanType mLKMT, string oldTypeId)
        {
            return dLKMT.update(mLKMT, oldTypeId);
        }
        public bool delete(string typeId)
        {
            return dLKMT.delete(typeId); ;
        }
        public DataTable select(string strWhere)
        {
            return dLKMT.select(strWhere);
        }
    }
}
```

现在来实现联系人管理的业务逻辑层代码，与联系人类型管理的业务逻辑相比，在实现联系人增加、删除、修改、查询功能之外，还需实现随机抽取一名联系人的功能，这就是一种业务逻辑，需在业务逻辑层实现。

在 BLL 项目中新建名为“Linkmen.cs”的类文件，添加代码如下。

```
namespace BLL
{
    public class Linkmen
    {
        DAL.Linkmen dLKM = new DAL.Linkmen();
        public bool insert(Model.Linkmen mLKM)
        {
            return dLKM.insert(mLKM);
        }
        public bool update(Model.Linkmen mLKM)
        {
            return dLKM.update(mLKM);
        }
        public bool delete(string stuId)
        {
            return dLKM.delete(stuId);
        }
        public DataTable select(string strWhere)
        {
            return dLKM.select(strWhere);
        }
        ///<summary>
        ///随机抽取一名联系人
        ///</summary>
        ///<returns>返回抽取到的联系人对象</returns>
        public Model.Linkmen randomFriend()
        {
            DataTable dt=select("");
            if (dt.Rows.Count > 0)
            {
                Random random = new Random();
                int currNum = random.Next(0, dt.Rows.Count);
                Model.Linkmen mLKM = new Model.Linkmen();
                mLKM.lkmId = dt.Rows[currNum]["lkmid"].ToString();
                mLKM.lkmName = dt.Rows[currNum]["lkmname"].ToString();
                mLKM.lkmMPNum = dt.Rows[currNum]["lkmmpnum"].ToString();
                mLKM.lkmOPNum = dt.Rows[currNum]["lkmopnum"].ToString();
                mLKM.lkmEmail= dt.Rows[currNum]["lkmemail"].ToString();
                mLKM.lkmCompName = dt.Rows[currNum]["lkmcompname"].ToString();
                mLKM.typeId = dt.Rows[currNum]["typeid"].ToString();
                return mLKM;
            }
            else
            {
                return null;
            }
        }
    }
}
```

可见随着业务逻辑的复杂度增加，业务逻辑层代码也会相应增加。

3.2.5 表示层

1. 创建表示层项目 UI

3.1 节提到表示层其实就是用户界面层，所以表示层的项目类型不是 DAL、BLL 这样的类库，而是 Windows 窗体应用程序。

右击解决方案，选择“添加”→“新建项目”命令，打开“添加新项目”对话框，搜索“Windows 窗体应用”模板，选择“Windows 窗体应用(.NET Framework)”模板，单击“下一步”按钮，将项目名称改为“UI”，单击“创建”按钮。此时，解决方案资源管理器中的项目结构如图 3-11 所示。

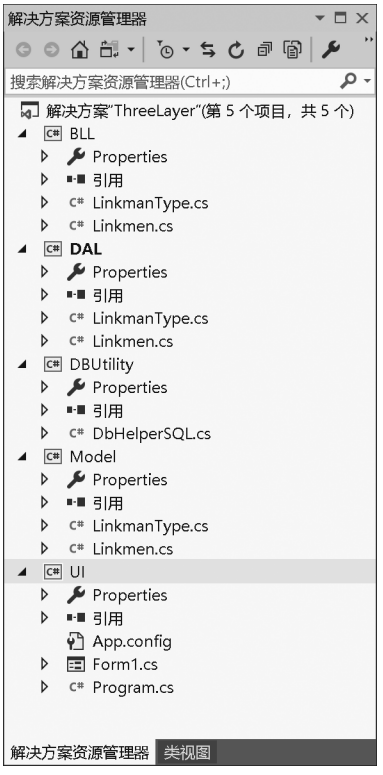

图 3-11 添加 UI 之后的解决方案资源管理器

2. 编写表示层代码

表示层需要引用 BLL 和 Model 项目，无须引用 DAL。删除 UI 项目中的 Form1.cs 窗体文件后，右击 UI 项目，在弹出式菜单中选择“添加”→“窗体(Windows 窗体)”命令，在弹出的“添加新项”窗体中，将名称改为 MainForm.cs，单击“添加”按钮完成窗体的添加。用同样的方法再添加四个 Windows 窗体，分别命名为 LinkmanTypeManage.cs、LinkmanTypeEdit.cs、LinkmenManage.cs 以及 LinkmenEdit.cs。各窗体的作用如表 3-3 所示。

表 3-3 各窗体作用

窗 体 名	作 用
MainForm	入口主窗体，程序的 MDI 父窗体
LinkmanTypeManage	联系人类型管理窗体，显示联系人类型列表并且是编辑入口
LinkmanTypeEdit	联系人类型编辑窗体
LinkmenManage	联系人管理窗体，显示联系人列表并且是编辑入口
LinkmenEdit	联系人编辑窗体

将 UI 项目中的 Program.cs 文件的 Main 方法中的 Application.Run(new Form1())语句改为 Application.Run(new MainForm())，指明应用程序运行时首先运行 MainForm 窗体。

接下来设置 MainForm 窗体的若干属性及其值，如表 3-4 所示。