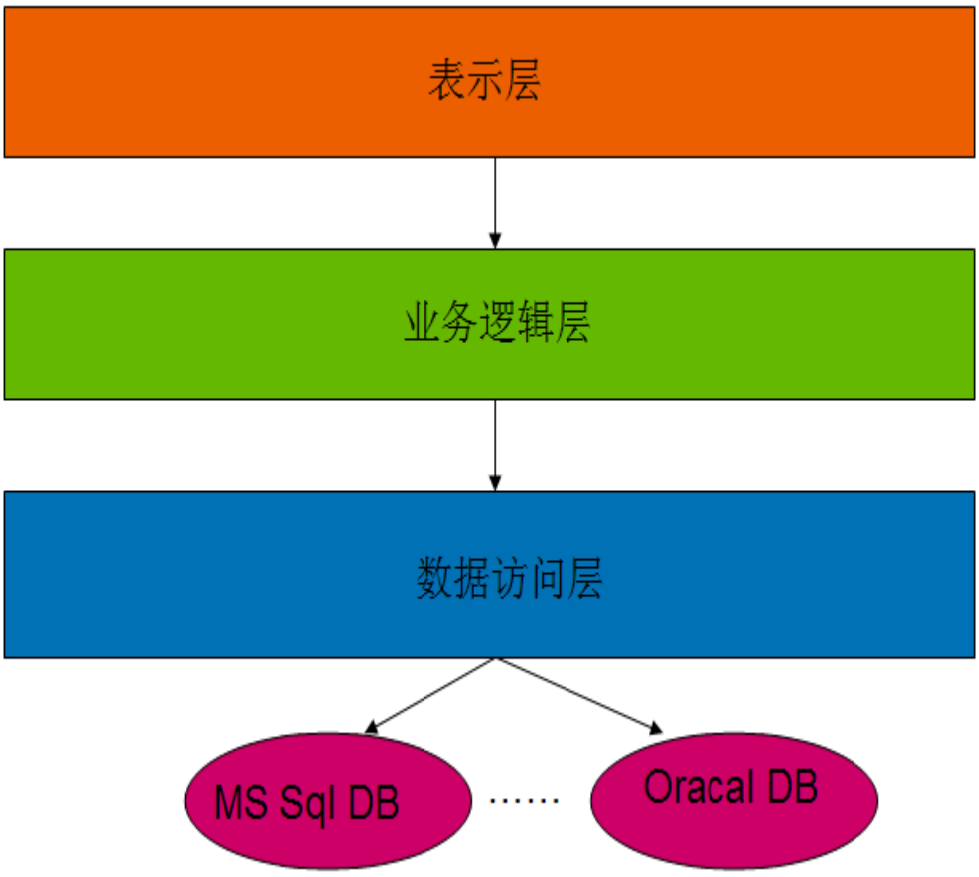
petshop4.0 详解之一（系统架构设计）

前言：PetShop 是一个范例，微软用它来展示.Net 企业系统开发的能力。业界有许多.Net 与 J2EE 之争，许多数据是从微软的 PetShop 和 Sun 的 PetStore 而来。这种争论不可避免带有浓厚的商业色彩，对于我们开发人员而言，没有必要过多关注。然而 PetShop 随着版本的不断更新，至现在基于.Net 2.0 的 PetShop4.0 为止，整个设计逐渐变得成熟而优雅，却又很多可以借鉴之处。PetShop 是一个小型的项目，系统架构与代码都比较简单，却也凸现了许多颇有价值的设计与开发理念。本系列试图对 PetShop 作一个全方位的解剖，依据的代码是 PetShop4.0，可以从链接 http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnbda/html/bdasamppet4.asp 中获得。

一、PetShop 的系统架构设计

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，从下至上分别为：数据访问层、业务逻辑层（又或成为领域层）、表示层，如图所示：



图一：三层的分层式结构

数据访问层：有时候也称为是持久层，其功能主要是负责数据库的访问。简单的说法就是实现对数据表的 Select，Insert，Update，Delete 的操作。如果要加入 ORM 的元素，那么就会包括对象和数据表之间的 mapping，以及对象实体的持久化。在 PetShop 的数据访问层中，并没有使用 ORM，从而导致了代码量的增加，可以看作是整个设计实现中的一大败笔。

业务逻辑层：是整个系统的核心，它与这个系统的业务（领域）有关。以 PetShop 为例，业务逻辑层的相关设计，均和网上宠物店特有的逻辑相关，例如查询宠物，下订单，添加宠物到购物车等等。如果涉及到数据库的访问，则调用数据访问层。

表示层：是系统的 UI 部分，负责使用者与整个系统的交互。在这一层中，理想的状态是不应包括系统的业务逻辑。表示层中的逻辑代码，仅与界面元素有关。在 PetShop 中，是利用 ASP.Net 来设计的，因此包含了许多 Web 控件和相关逻辑。

分层式结构究竟其优势何在？Martin Fowler 在《Patterns of Enterprise Application Architecture》一书中给出了答案：
1、开发人员可以只关注整个结构中的其中某一层；
2、可以很容易的用新的实现来替换原有层次的实现；
3、可以降低层与层之间的依赖；
4、有利于标准化；
5、利于各层逻辑的复用。

概括来说，分层式设计可以达至如下目的：分散关注、松散耦合、逻辑复用、标准定义。

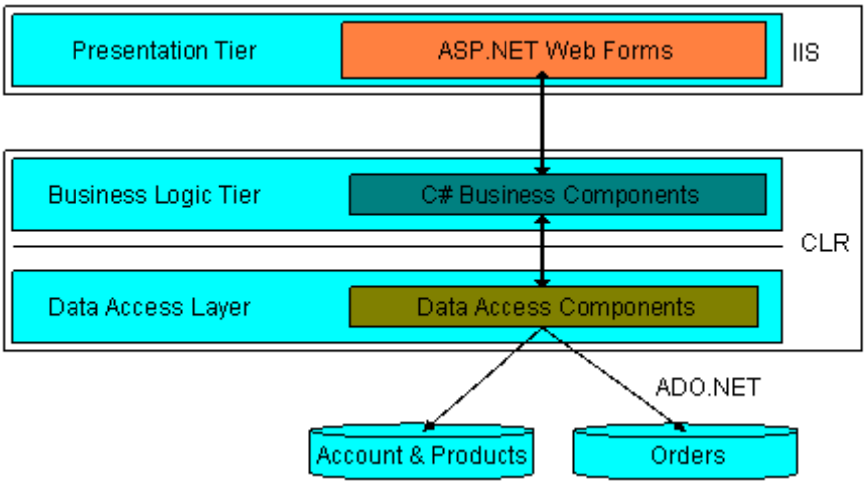
一个好的分层式结构，可以使得开发人员的分工更加明确。一旦定义好各层次之间的接口，负责不同逻辑设计的开发人员就可以分散关注，齐头并进。例如 UI 人员只需考虑用户界面的体验与操作，领域的设计人员可以仅关注业务逻辑的设计，而数据库设计人员也不必为繁琐的用户交互而头疼了。每个开发人员的任务得到了确认，开发进度就可以迅速的提高。

松散耦合的好处是显而易见的。如果一个系统没有分层，那么各自的逻辑都紧紧纠缠在一起，彼此间相互依赖，谁都是不可替换的。一旦发生改变，则牵一发而动全身，对项目的影响极为严重。降低层与层间的依赖性，既可以良好地保证未来的可扩展，在复用性上也是优势明显。每个功能模块一旦定义好统一的接口，就可以被各个模块所调用，而不用为相同的功能进行重复地开发。

进行好的分层式结构设计，标准也是必不可少的。只有在一定程度的标准化基础上，这个系统才是可扩展的，可替换的。而层与层之间的通信也必然保证了接口的标准化。

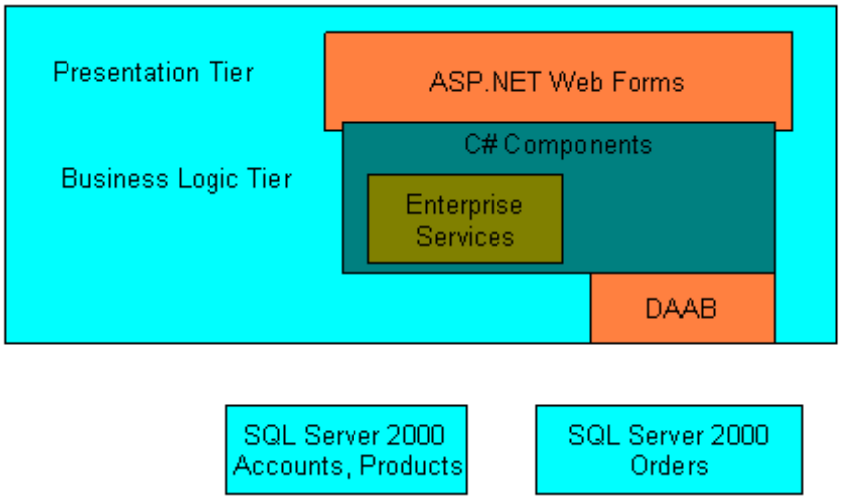
“金无足赤，人无完人”，分层式结构也不可避免具有一些缺陷：
1、降低了系统的性能。这是不言而喻的。如果不采用分层式结构，很多业务可以直接造访数据库，以此获取相应的数据，如今却必须通过中间层来完成。
2、有时会导致级联的修改。这种修改尤其体现在自上而下的方向。如果在表示层中需要增加一个功能，为保证其设计符合分层式结构，可能需要在相应的业务逻辑层和数据访问层中都增加相应的代码。

前面提到，PetShop 的表示层是用 ASP.Net 设计的，也就是说，它应是一个 BS 系统。在.Net 中，标准的 BS 分层式结构如下图所示：



图二：.Net 中标准的 BS 分层式结构

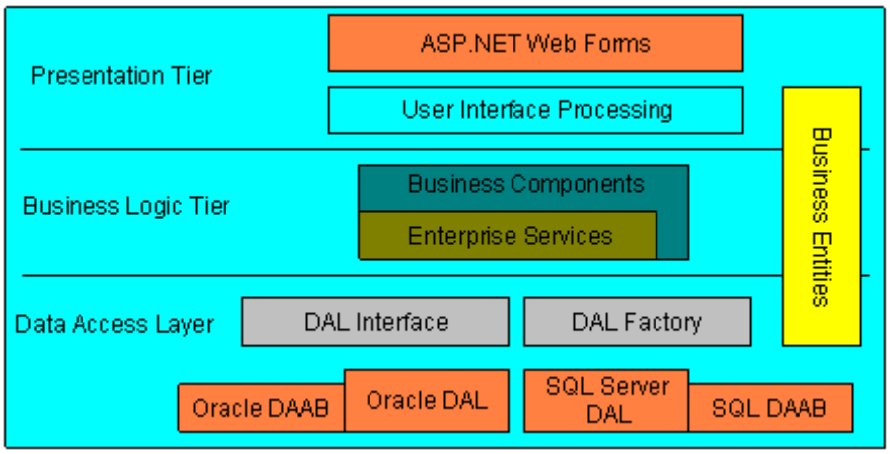
随着 PetShop 版本的更新，其分层式结构也在不断的完善，例如 PetShop2.0，就没有采用标准的三层式结构，如图三：



图三：PetShop 2.0 的体系架构

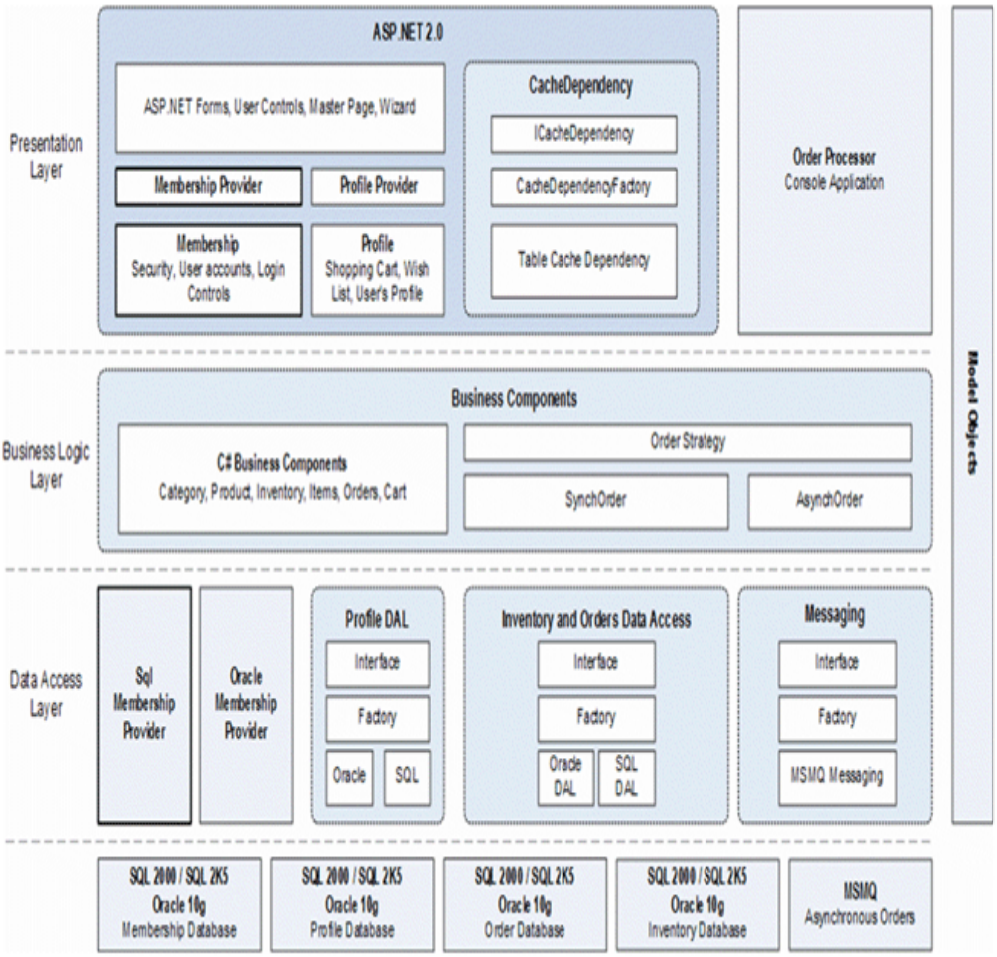
从图中我们可以看到，并没有明显的数据访问层设计。这样的设计虽然提高了数据访问的性能，但也同时导致了业务逻辑层与数据访问的职责混乱。一旦要求支持的数据库发生变化，或者需要修改数据访问的逻辑，由于没有清晰的分层，会导致项目作大的修改。而随着硬件系统性能的提高，以及充分利用缓存、异步处理等机制，分层式结构所带来的性能影响几乎可以忽略不计。

PetShop3.0 纠正了此前层次不明的问题，将数据访问逻辑作为单独的一层独立出来：



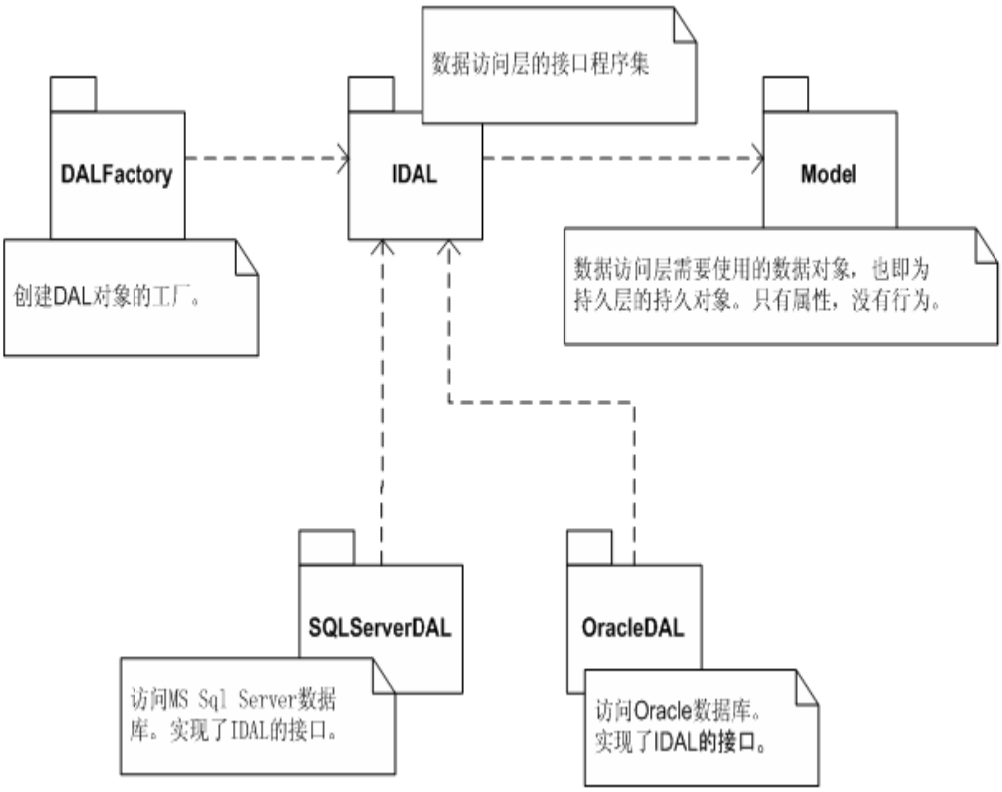
图四：PetShop 3.0 的体系架构

PetShop4.0 基本上延续了 3.0 的结构，但在性能上作了一定的改进，引入了缓存和异步处理机制，同时又充分利用了 ASP.Net 2.0 的新功能 MemberShip，因此 PetShop4.0 的系统架构图如下所示：



图五：PetShop 4.0 的体系架构

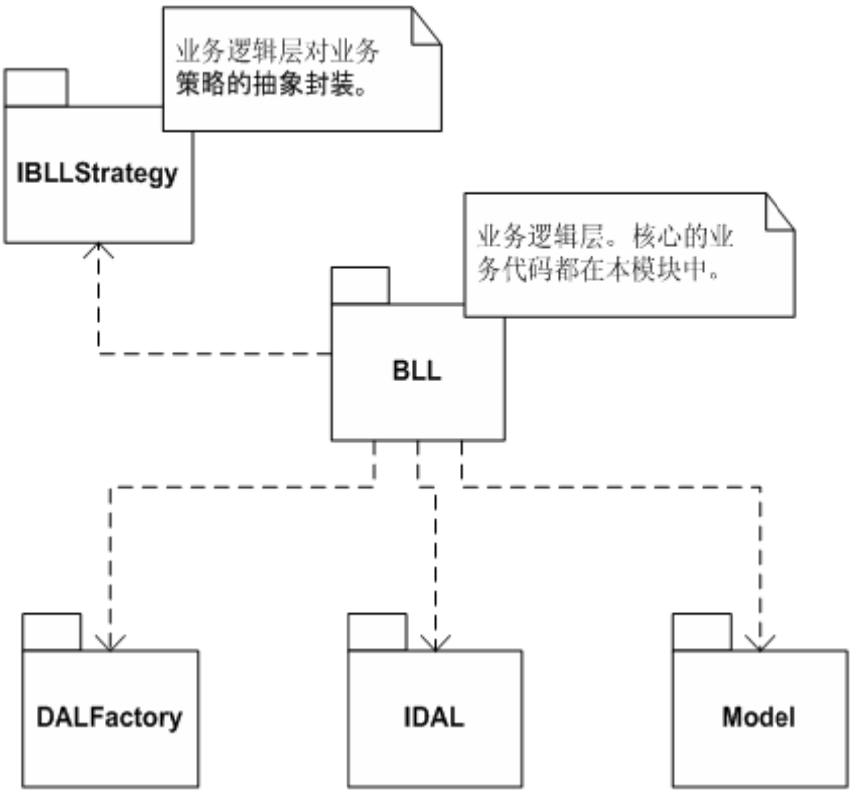
比较 3.0 和 4.0 的系统架构图，其核心的内容并没有发生变化。在数据访问层（DAL）中，仍然采用 DAL Interface 抽象出数据访问逻辑，并以 DAL Factory 作为数据访问层对象的工厂模块。对于 DAL Interface 而言，分别有支持 MS-SQL 的 SQL Server DAL 和支持 Oracle 的 Oracle DAL 具体实现。而 Model 模块则包含了数据实体对象。其详细的模块结构图如下所示：



图六：数据访问层的模块结构图

可以看到，在数据访问层中，完全采用了“面向接口编程”思想。抽象出来的 IDAL 模块，脱离了与具体数据库的依赖，从而使得整个数据访问层利于数据库迁移。DALFactory 模块专门管理 DAL 对象的创建，便于业务逻辑层访问。SQLServerDAL 和 OracleDAL 模块均实现 IDAL 模块的接口，其中包含的逻辑就是对数据库的 Select，Insert，Update 和 Delete 操作。因为数据库类型的不同，对数据库的操作也有所不同，代码也会因此有所区别。

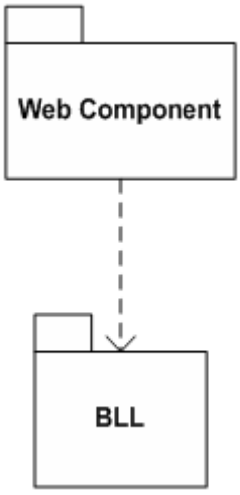
此外，抽象出来的 IDAL 模块，除了解除了向下的依赖之外，对于其上的业务逻辑层，同样仅存在弱依赖关系，如下图所示：



图七：业务逻辑层的模块结构图

图七中 BLL 是业务逻辑层的核心模块，它包含了整个系统的核心业务。在业务逻辑层中，不能直接访问数据库，而必须通过数据访问层。注意图中对数据访问业务的调用，是通过接口模块 IDAL 来完成的。既然与具体的数据访问逻辑无关，则层与层之间的关系就是松散耦合的。如果此时需要修改数据访问层的具体实现，只要不涉及到 IDAL 的接口定义，那么业务逻辑层就不会受到任何影响。毕竟，具体实现的 SQLServerDAL 和 OracalDAL 根本就与业务逻辑层没有半点关系。

因为在 PetShop 4.0 中引入了异步处理机制。插入订单的策略可以分为同步和异步，两者的插入策略明显不同，但对于调用者而言，插入订单的接口是完全一样的，所以 PetShop 4.0 中设计了 IBLLStrategy 模块。虽然在 IBLLStrategy 模块中，仅仅是简单的 IOrderStategy，但同时也给出了一个范例和信息，那就是在业务逻辑的处理中，如果存在业务操作的多样化，或者是今后可能的变化，均应利用抽象的原理。或者使用接口，或者使用抽象类，从而脱离对具体业务的依赖。不过在 PetShop 中，由于业务逻辑相对简单，这种思想体现得不够明显。也正因为此，PetShop 将核心的业务逻辑都放到了一个模块 BLL 中，并没有将具体的实现和抽象严格的按照模块分开。所以表示层和业务逻辑层之间的调用关系，其耦合度相对较高：



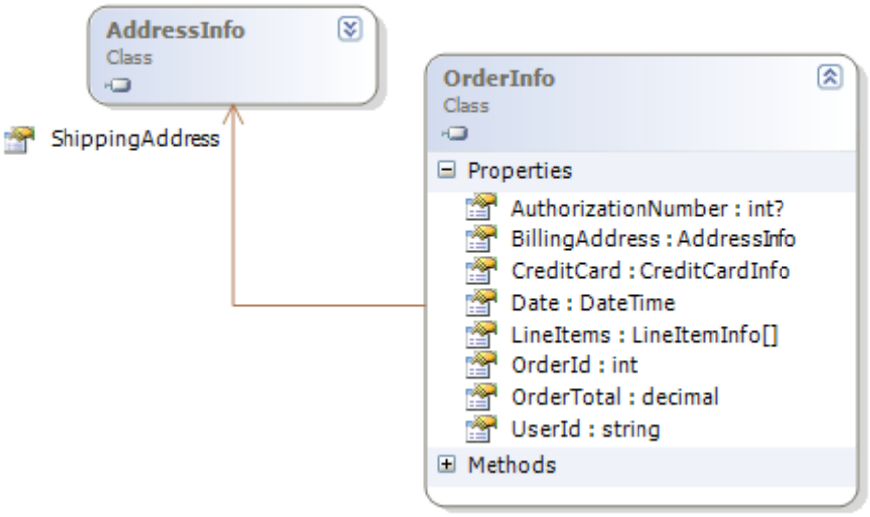
图八：表示层的模块结构图

在图五中，各个层次中还引入了辅助的模块，如数据访问层的 Messaging 模块，是为异步插入订单的功能提供，采用了 MSMQ（Microsoft Messaging Queue）技术。而表示层的 CacheDependency 则提供缓存功能。这些特殊的模块，我会在此后的文章中详细介绍。

petshop4.0 详解之二（数据访问层之数据库访问设计）

在系列一中，我从整体上分析了 PetShop 的架构设计，并提及了分层的概念。从本部分开始，我将依次对各层进行代码级的分析，以求获得更加细致而深入的理解。在 PetShop 4.0 中，由于引入了 ASP.Net 2.0 的一些新特色，所以数据层的内容也更加的广泛和复杂，包括：数据库访问、Messaging、MemberShip、Profile 四部分。在系列二中，我将介绍有关数据库访问的设计。

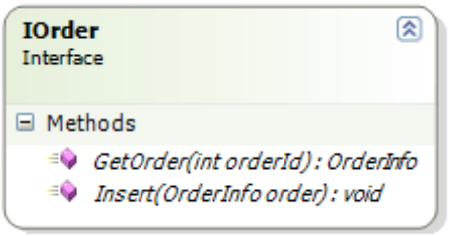
在 PetShop 中，系统需要处理的数据库对象分为两类：一是数据实体，对应数据库中相应的数据表。它们没有行为，仅用于表现对象的数据。这些实体类都被放到 Model 程序集中，例如数据表 Order 对应的实体类 OrderInfo，其类图如下：



这些对象并不具有持久化的功能，简单地说，它们是作为数据的载体，便于业务逻辑针对相应数据表进行读/写操作。虽然这些类的属性分别映射了数据表的列，而每一个对象实例也恰恰对应于数据表的每一行，但这些实体类却并不具备对应的数据库访问能力。

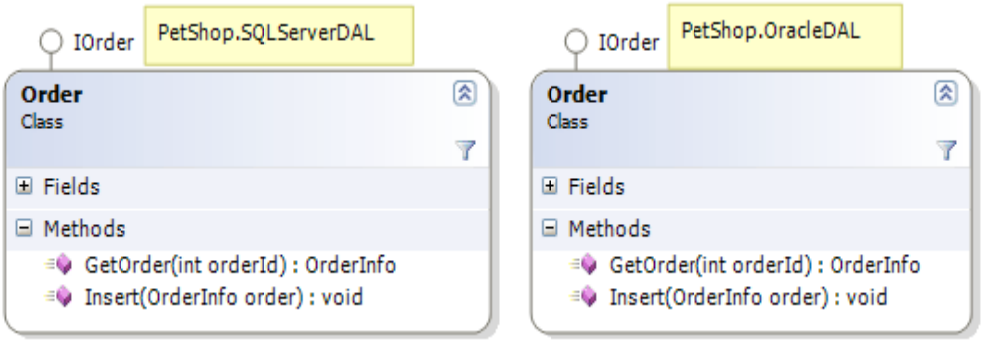
由于数据访问层和业务逻辑层都将对这些数据实体进行操作，因此程序集 Model 会被这两层的模块所引用。

第二类数据库对象则是数据的业务逻辑对象。这里所指的业务逻辑，并非业务逻辑层意义上的领域（domain）业务逻辑（从这个意义上，我更倾向于将业务逻辑层称为“领域逻辑层”），一般意义上说，这些业务逻辑即为基本的数据库操作，包括 Select，Insert，Update 和 Delete。由于这些业务逻辑对象，仅具有行为而与数据无关，因此它们均被抽象为一个单独的接口模块 IDAL，例如数据表 Order 对应的接口 IOrder：



将数据实体与相关的数据库操作分离出来，符合面向对象的精神。首先，它体现了“职责分离”的原则。将数据实体与其行为分开，使得两者之间依赖减弱，当数据行为发生改变时，并不影响 Model 模块中的数据实体对象，避免了因一个类职责过多、过大，从而导致该类的引用者发生“灾难性”的影响。其次，它体现了“抽象”的精神，或者说是“面向接口编程”的最佳体现。抽象的接口模块 IDAL，与具体的数据库访问实现完全隔离。这种与实现无关的设计，保证了系统的可扩展性，同时也保证了数据库的可移植性。在 PetShop 中，可以支持 SQL Server 和 Oracle，那么它们具体的实现就分别放在两个不同的模块 SQLServerDAL、OracleDAL 中。

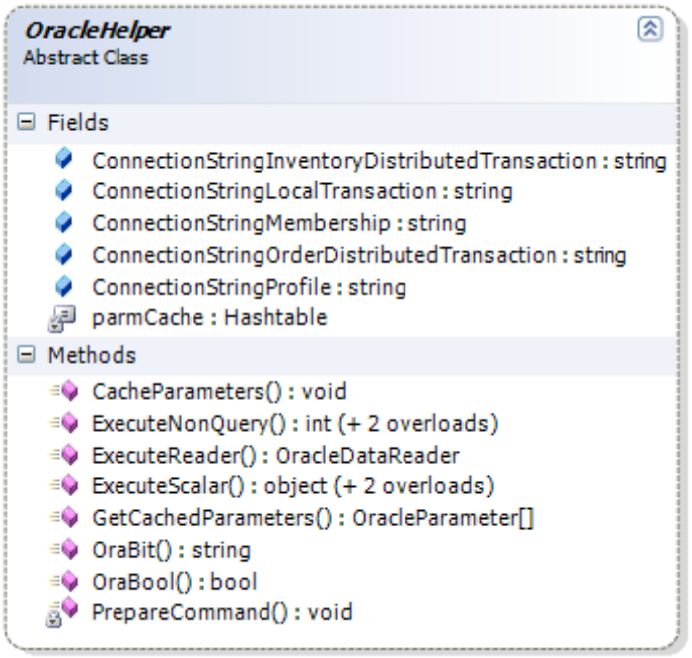
以 Order 为例，在 SQLServerDAL、OracleDAL 两个模块中，有不同的实现，但它们同时又都实现了 IOrder 接口，如图：



从数据库的实现来看，PetShop 体现出了没有 ORM 框架的臃肿与丑陋。由于要对数据表进行 Insert 和 Select 操作，以 SQL Server 为例，就使用了 SqlCommand，SqlParameter，SqlDataReader 等对象，以完成这些操作。尤其复杂的是 Parameter 的传递，在 PetShop

中，使用了大量的字符串常量来保存参数的名称。此外，PetShop 还专门为 SQL Server 和 Oracle 提供了抽象的 Helper 类，包装了一些常用的操作，如 ExecuteNonQuery、ExecuteReader 等方法。

在没有 ORM 的情况下，使用 Helper 类是一个比较好的策略，利用它来完成数据库基本操作的封装，可以减少很多和数据库操作有关的代码，这体现了对象复用的原则。PetShop 将这些 Helper 类统一放到 DBUtility 模块中，不同数据库的 Helper 类暴露的方法基本相同，只除了一些特殊的要求，例如 Oracle 中处理 bool 类型的方式就和 SQL Server 不同，从而专门提供了 OraBit 和 OraBool 方法。此外，Helper 类中的方法均为 static 方法，以利于调用。OracleHelper 的类图如下：



对于数据访问层来说，最头疼的是 SQL 语句的处理。在早期的 CS 结构中，由于未采用三层式架构设计，数据访问层和业务逻辑层是紧密糅合在一起的，因此，SQL 语句遍布与系统的每一个角落。这给程序的维护带来极大的困难。此外，由于 Oracle 使用的是 PL-SQL，而 SQL Server 和 Sybase 等使用的是 T-SQL，两者虽然都遵循了标准 SQL 的语法，但在很多细节上仍有区别，如果将 SQL 语句大量的使用到程序中，无疑为可能的数据库移植也带来了困难。

最好的方法是采用存储过程。这种方法使得程序更加整洁，此外，由于存储过程可以以数据库脚本的形式存在，也便于移植和修改。但这种方式仍然有缺陷。一是存储过程的测试相对困难。虽然有相应的调试工具，但比起对代码的调试而言，仍然比较复杂且不方便。二是对系统的更新带来障碍。如果数据库访问是由程序完成，在.Net 平台下，我们仅需要在修改程序后，将重新编译的程序集 xcopy 到部署的服务器上即可。如果使用了存储过程，出于安全的考虑，必须有专门的 DBA 重新运行存储过程的脚本，部署的方式受到了限制。

我曾经在一个项目中，利用一个专门的表来存放 SQL 语句。如要使用相关的 SQL 语句，就利用关键字搜索获得对应语句。这种做法近似于存储过程的调用，但却避免了部署上的问题。然而这种方式却在性能上无法得到保证。它仅适合于 SQL 语句较少的场景。不过，利用良好的设计，

我们可以为各种业务提供不同的表来存放 SQL 语句。同样的道理，这些 SQL 语句也可以存放到 XML 文件中，更有利于系统的扩展或修改。不过前提是，我们需要为它提供专门的 SQL 语句管理工具。

SQL 语句的使用无法避免，如何更好的应用 SQL 语句也无定论，但有一个原则值得我们遵守，就是“应该尽量让 SQL 语句尽存在于数据访问层的具体实现中”。

当然，如果应用 ORM，那么一切就变得不同了。因为 ORM 框架已经为数据访问提供了基本的 Select，Insert，Update 和 Delete 操作了。例如在 NHibernate 中，我们可以直接调用 ISession 对象的 Save 方法，来 Insert（或者说是 Create）一个数据实体对象：

```
public void Insert(OrderInfo order)
{
    ISession s = Sessions.GetSession();
    ITransaction trans = null;
    try
    {
    trans = s.BeginTransaction();
      s.Save( order);
      trans.Commit();
    }
    finally
    {
      s.Close();
    }
}
```

没有 SQL 语句，也没有那些烦人的 Parameters，甚至不需要专门去考虑事务。此外，这样的设计，也是与数据库无关的，NHibernate 可以通过 Dialect（方言）的机制支持不同的数据库。唯一要做的是，我们需要为 OrderInfo 定义 hbm 文件。

当然，ORM 框架并非是万能的，面对纷繁复杂的业务逻辑，它并不能完全消灭 SQL 语句，以及替代复杂的数据库访问逻辑，但它却很好的体现了“80/20（或 90/10）法则”（也被称为“帕累托法则”），也就是说：花比较少（10%-20%）的力气就可以解决大部分（80%-90%）的问题，而要解决剩下的少部分问题则需要多得多的努力。至少，那些在数据访问层中占据了绝大部分的 CRUD 操作，通过利用 ORM 框架，我们就仅需要付出极少数时间和精力来解决它们了。这无疑缩短了整个项目开发的周期。

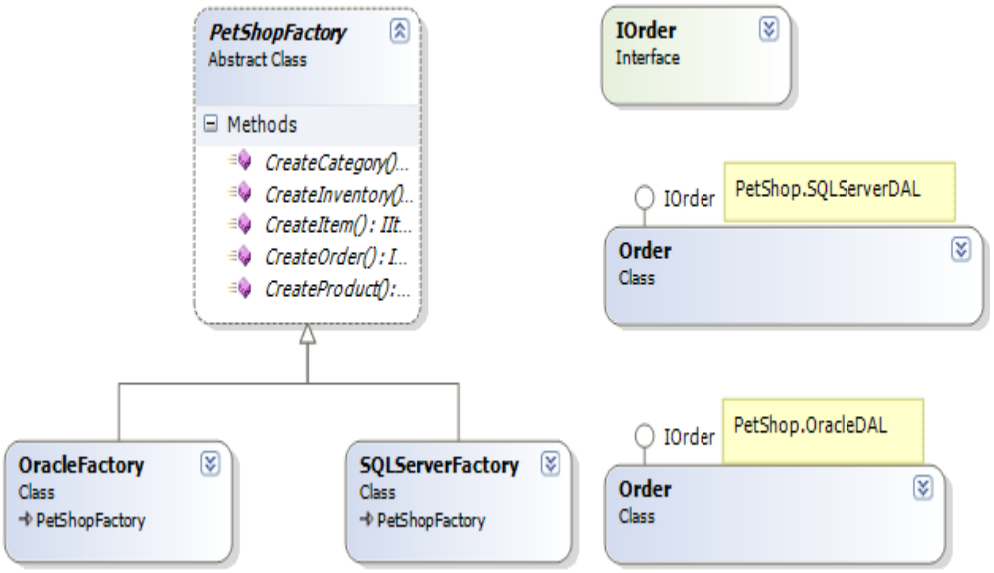
还是回到对 PetShop 的讨论上来。现在我们已经有了数据实体，数据对象的抽象接口和实现，可以说有关数据库访问的主体就已经完成了。留待我们的还有两个问题需要解决：

1、数据对象创建的管理

2、利于数据库的移植

在 PetShop 中，要创建的数据对象包括 Order，Product，Category，Inventory，Item。在前面的设计中，这些对象已经被抽象为对应的接口，而其实现则根据数据库的不同而有所不同。也就是说，创建的对象有多种类别，而每种类别又有不同的实现，这是典型的抽象工厂模式的应

用场景。而上面所述的两个问题，也都可以通过抽象工厂模式来解决。标准的抽象工厂模式类图如下：



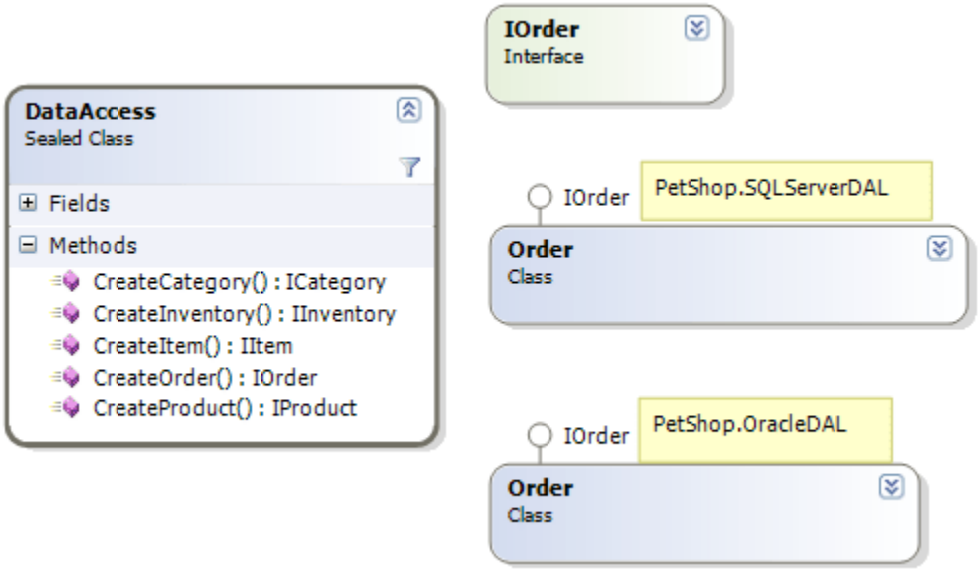
例如，创建 SQL Server 的 Order 对象如下：

```
PetShopFactory factory = new SQLServerFactory();
IOrder = factory.CreateOrder();
```

要考虑到数据库的可移植性，则 factory 必须作为一个全局变量，并在主程序运行时被实例化。但这样的设计虽然已经达到了“封装变化”的目的，但在创建 PetShopFactory 对象时，仍不可避免的出现了具体的类 SQLServerFactory，也即是说，程序在这个层面上产生了与 SQLServerFactory 的强依赖。一旦整个系统要求支持 Oracle，那么还需要修改这行代码为：

```
PetShopFactory factory = new oracleFactory();
```

修改代码的这种行为显然是不可接受的。解决的办法是“依赖注入”。“依赖注入”的功能通常是用专门的 IoC 容器提供的，在 Java 平台下，这样的容器包括 Spring，PicoContainer 等。而在.Net 平台下，最常见的则是 Spring.Net。不过，在 PetShop 系统中，并不需要专门的容器来实现“依赖注入”，简单的做法还是利用配置文件和反射功能来实现。也就是说，我们可以在 web.config 文件中，配置好具体的 Factory 对象的完整的类名。然而，当我们利用配置文件和反射功能时，具体工厂的创建就显得有些“画蛇添足”了，我们完全可以在配置文件中，直接指向具体的数据库对象实现类，例如 PetShop.SQLServerDAL.IOrder。那么，抽象工厂模式中的相关工厂就可以简化为一个工厂类了，所以我将这种模式称之为“具有简单工厂特质的抽象工厂模式”，其类图如下：



DataAccess 类完全取代了前面创建的工厂类体系，它是一个 sealed 类，其中创建各种数据对象的方法，均为静态方法。之所以能用这个类达到抽象工厂的目的，是因为配置文件和反射的运用，如下的代码片断所示：

```
public sealed class DataAccess
{
 // Look up the DAL implementation we should be using
    private static readonly string path =
ConfigurationManager.AppSettings[”WebDAL”];
    private static readonly string orderPath =
ConfigurationManager.AppSettings[”OrdersDAL”];

 public static PetShop.IDAL.IOrder CreateOrder()
 {
        string className = orderPath + “.Order”;
        return
(PetShop.IDAL.IOrder)Assembly.Load(orderPath).CreateInstance(className);
    }
}
```

在 PetShop 中，这种依赖配置文件和反射创建对象的方式极其常见，包括 IBLLStategy、CacheDependencyFactory 等等。这些实现逻辑散布于整个 PetShop 系统中，在我看来，是可以在此基础上进行重构的。也就是说，我们可以为整个系统提供类似于“Service Locator”的实现：

```
public static class ServiceLocator
{
 private static readonly string dalPath =
ConfigurationManager.AppSettings[”WebDAL”];
    private static readonly string orderPath =
ConfigurationManager.AppSettings[”OrdersDAL”];
```

```
 //……
 private static readonly string orderStategyPath =
ConfigurationManager.AppSettings[”OrderStrategyAssembly”];

 public static object LocateDALObject(string className)
 {
  string fullPath = dalPath + “.” + className;
  return Assembly.Load(dalPath).CreateInstance(fullPath);
 }
public static object LocateDALOrderObject(string className)
 {
  string fullPath = orderPath + “.” + className;
  return Assembly.Load(orderPath).CreateInstance(fullPath);
 }
public static object LocateOrderStrategyObject(string className)
 {
  string fullPath = orderStategyPath + “.” + className;
  return Assembly.Load(orderStategyPath).CreateInstance(fullPath);
 }
 //……
}
```

那么和所谓“依赖注入”相关的代码都可以利用 ServiceLocator 来完成。例如类 DataAccess 就可以简化为：

```
public sealed class DataAccess
{
 public static PetShop.IDAL.IOrder CreateOrder()
 {
        return (PetShop.IDAL.IOrder)ServiceLocator.
LocateDALOrderObject(”Order”);
    }
}
```

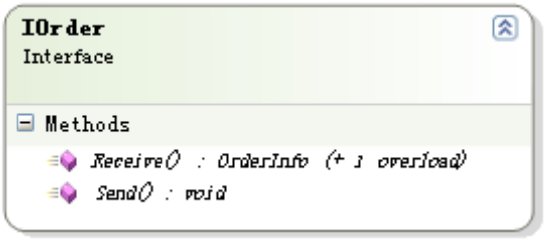
通过 ServiceLocator，将所有与配置文件相关的 namespace 值统一管理起来，这有利于各种动态创建对象的管理和未来的维护。

petshop4.0 详解之三(PetShop 数据访问层之消息处理)

在进行系统设计时，除了对安全、事务等问题给与足够的重视外，性能也是一个不可避免的问题所在，尤其是一个 B/S 结构的软件系统，必须充分地考虑访问量、数据流量、服务器负荷的问题。解决性能的瓶颈，除了对硬件系统进行升级外，软件设计的合理性尤为重要。

在前面我曾提到，分层式结构设计可能会在一定程度上影响数据访问的性能，然而与它给设计人员带来的好处相比，几乎可以忽略。要提供整个系统的性能，还可以从数据库的优化着手，例如连接池的使用、建立索引、优化查询策略等等，例如在 PetShop 中就利用了数据库的 Cache，对于数据量较大的订单数据，则利用分库的方式为其单独建立了 Order 和 Inventory 数据库。而在软件设计上，比较有用的方式是利用多线程与异步处理方式。

在 PetShop4.0 中，使用了 Microsoft Messaging Queue(MSMQ)技术来完成异步处理，利用消息队列临时存放要插入的数据，使得数据访问因为不需要访问数据库从而提供了访问性能，至于队列中的数据，则等待系统空闲的时候再进行处理，将其最终插入到数据库中。
PetShop4.0 中的消息处理，主要分为如下几部分：消息接口 IMessaging、消息工厂 MessagingFactory、MSMQ 实现 MSMQMessaging 以及数据后台处理应用程序 OrderProcessor。
从模块化分上，PetShop 自始自终地履行了“面向接口设计”的原则，将消息处理的接口与实现分开，并通过工厂模式封装消息实现对象的创建，以达到松散耦合的目的。
由于在 PetShop 中仅对订单的处理使用了异步处理方式，因此在消息接口 IMessaging 中，仅定义了一个 IOrder 接口，其类图如下：



在对消息接口的实现中，考虑到未来的扩展中会有其他的数据对象会使用 MSMQ，因此定义了一个 Queue 的基类，实现消息 Receive 和 Send 的基本操作：

```
public virtual object Receive()
{
    try
{
       using (Message message = queue.Receive(timeout, transactionType))
         return message;
    }
    catch (MessageQueueException mqex)
{
       if (mqex.MessageQueueErrorCode == MessageQueueErrorCode.IOTimeout)
         throw new TimeoutException();
           throw;
    }
}
public virtual void Send(object msg)
{
    queue.Send(msg, transactionType);
}
```

其中 queue 对象是 System.Messaging.MessageQueue 类型，作为存放数据的队列。MSMQ 队列是一个可持久的队列，因此不必担心用户不间断地下订单会导致订单数据的丢失。在 PetShopQueue 设置了 timeout 值，OrderProcessor 会根据 timeout 值定期扫描队列中的订单数据。
MSMQMessaging 模块中，Order 对象实现了 IMessaging 模块中定义的接口 IOrder，同时它还继承了基类 PetShopQueue，其定义如下：

```
public class order:PetShopQueue, PetShop.IMessaging.IOrder
```

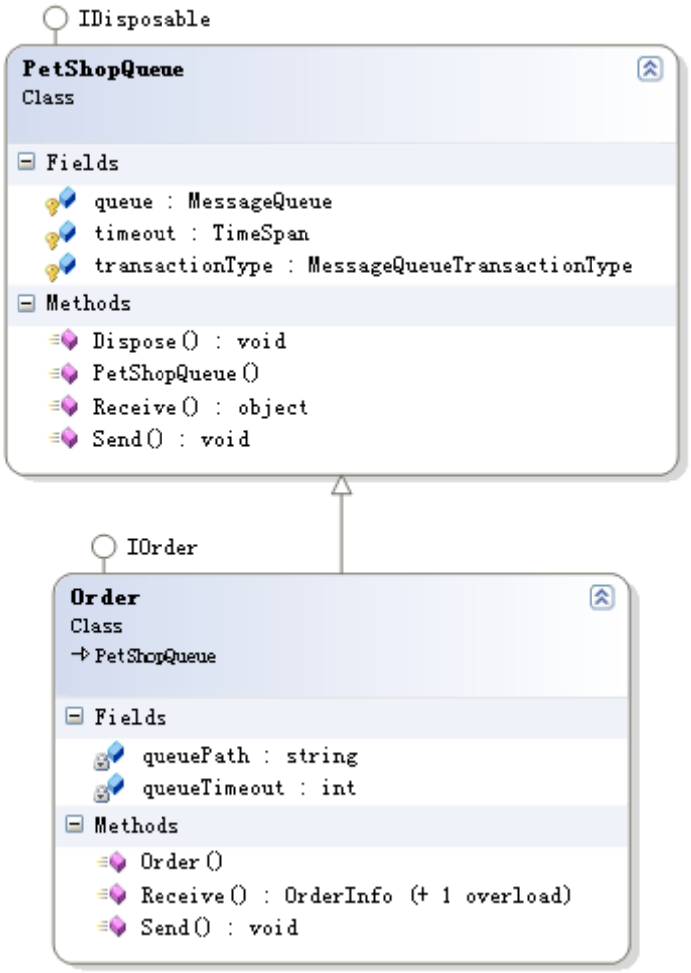
方法的实现代码如下：

```
    public new orderInfo Receive()
    {
        // This method involves in distributed transaction and need Automatic
Transaction type
        base.transactionType = MessageQueueTransactionType.Automatic;
        return (OrderInfo)((Message)base.Receive()).Body;
    }

    public orderInfo Receive(int timeout)
    {
        base.timeout = TimeSpan.FromSeconds(Convert.ToDouble(timeout));
        return Receive();
    }

    public void Send(OrderInfo orderMessage)
    {
        // This method does not involve in distributed transaction and optimizes
performance using Single type
        base.transactionType = MessageQueueTransactionType.Single;
        base.Send(orderMessage);
    }
```

所以，最后的类图应该如下：



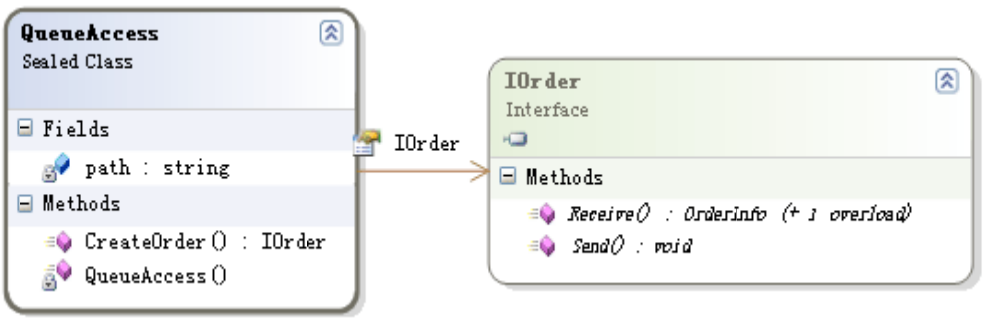
注意在 Order 类的 Receive()方法中，是用 new 关键字而不是 override 关键字来重写其父类 PetShopQueue 的 Receive()虚方法。因此，如果是实例化如下的对象，将会调用 PetShopQueue 的 Receive()方法，而不是子类 Order 的 Receive()方法：

```
PetShopQueue queue = new order();
queue.Receive();
```

从设计上来看，由于 PetShop 采用“面向接口设计”的原则，如果我们要创建 Order 对象，应该采用如下的方式：

```
IOrder order = new order();
order.Receive();
```

考虑到 IOrder 的实现有可能的变化，PetShop 仍然利用了工厂模式，将 IOrder 对象的创建用专门的工厂模块进行了封装：



在类 QueueAccess 中，通过 CreateOrder()方法利用反射技术创建正确的 IOrder 类型对象：

```
    public static PetShop.IMessaging.IOrder CreateOrder()
    {
        string className = path + ".Order";
        return
PetShop.IMessaging.IOrder)Assembly.Load(path).CreateInstance(className);
    }
```

path 的值通过配置文件获取：

```
private static readonly string path =
ConfigurationManager.AppSettings["OrderMessaging"];
```

而配置文件中，OrderMessaging 的值设置如下：

```
<add key="OrderMessaging" value="PetShop.MSMQMessaging"/>
```

之所以利用工厂模式来负责对象的创建，是便于在业务层中对其调用，例如在 BLL 模块中 OrderAsynchronous 类：

```
public class orderAsynchronous : IOrderStrategy
{
    private static readonly PetShop.IMessaging.IOrder asynchOrder =
PetShop.MessagingFactory.QueueAccess.CreateOrder();
    public void Insert(PetShop.Model.OrderInfo order)
{
        asynchOrder.Send(order);
    }
}
```

一旦 IOrder 接口的实现发生变化，这种实现方式就可以使得客户仅需要修改配置文件，而不需要修改代码，如此就可以避免程序集的重新编译和部署，使得系统能够灵活应对需求的改变。例如定义一个实现 IOrder 接口的 SpecialOrder，则可以新增一个模块，如 PetShop.SpecialMSMQMessaging，而类名则仍然为 Order，那么此时我们仅需要修改配置文件中 OrderMessaging 的值即可：

```
<add key="OrderMessaging" value="PetShop.SpecialMSMQMessaging"/>
```

OrderProcessor 是一个控制台应用程序，不过可以根据需求将其设计为 Windows Service。它的目的就是接收消息队列中的订单数据，然后将其插入到 Order 和 Inventory 数据库中。它利用了多线程技术，以达到提高系统性能的目的。

在 OrderProcessor 应用程序中，主函数 Main 用于控制线程，而核心的执行任务则由方法 ProcessOrders()实现：

```
    private static void ProcessOrders()
    {
        // the transaction timeout should be long enough to handle all of orders in the
batch
        TimeSpan tsTimeout =
TimeSpan.FromSeconds(Convert.ToDouble(transactionTimeout * batchSize));

        order order = new order();
        while (true)
        {
```

```
        // queue timeout variables
        TimeSpan datetimeStarting = new TimeSpan(DateTime.Now.Ticks);
        double elapsedTime = 0;

        int processedItems = 0;

        ArrayList queueOrders = new ArrayList();

        using (TransactionScope ts = new
TransactionScope(TransactionScopeOption.Required, tsTimeout))
        {
           // Receive the orders from the queue
           for (int j = 0; j < batchSize; j++)
           {
              try
              {
                 //only receive more queued orders if there is enough time
                 if ((elapsedTime + queueTimeout + transactionTimeout) <
tsTimeout.TotalSeconds)
                 {
                    queueOrders.Add(order.ReceiveFromQueue(queueTimeout));
                 }
                 else
                 {
                    j = batchSize;    // exit loop
                 }

                 //update elapsed time
                 elapsedTime = new
TimeSpan(DateTime.Now.Ticks).TotalSeconds - datetimeStarting.TotalSeconds;
              }
              catch (TimeoutException)
              {
                 //exit loop because no more messages are waiting
                 j = batchSize;
              }
           }
           //process the queued orders
           for (int k = 0; k < queueOrders.Count; k++)
           {
              order.Insert((OrderInfo)queueOrders[k]);
              processedItems++;
              totalOrdersProcessed++;
           }
```

```
            //batch complete or MSMQ receive timed out
            ts.Complete();
        }

        Console.WriteLine("(Thread Id " +
Thread.CurrentThread.ManagedThreadId + ") batch finished, " + processedItems +
" items, in " + elapsedTime.ToString() + " seconds.");
    }
}
```

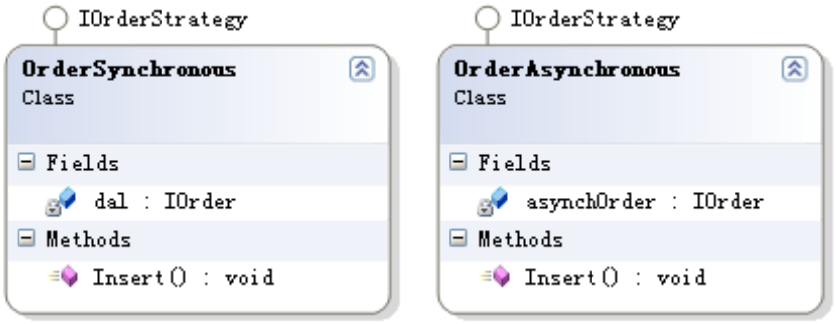
首先，它会通过 PetShop.BLL.Order 类的公共方法 ReceiveFromQueue()来获取消息队列中的订单数据，并将其放入到一个 ArrayList 对象中，然而再调用 PetShop.BLL.Order 类的 Insert 方法将其插入到 Order 和 Inventory 数据库中。

在 PetShop.BLL.Order 类中，并不是直接执行插入订单的操作，而是调用了 IOrderStrategy 接口的 Insert()方法：

```
public void Insert(OrderInfo order)
{
    // Call credit card procesor
    ProcessCreditCard(order);

    // Insert the order (a)synchrounously based on configuration
    orderInsertStrategy.Insert(order);
}
```

在这里，运用了一个策略模式，类图如下所示：



在 PetShop.BLL.Order 类中，仍然利用配置文件来动态创建 IOrderStategy 对象：

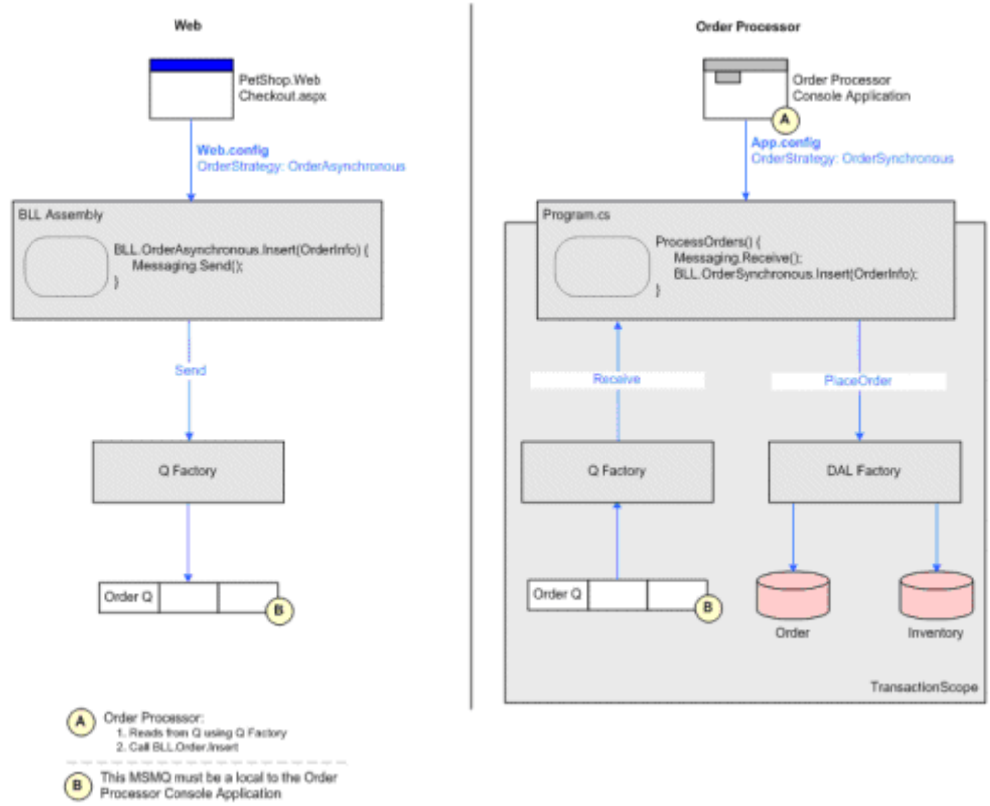
```
private static readonly PetShop.IBLLStrategy.IOrderStrategy orderInsertStrategy
= LoadInsertStrategy();
private static PetShop.IBLLStrategy.IOrderStrategy LoadInsertStrategy()
{
    // Look up which strategy to use from config file
    string path = ConfigurationManager.AppSettings["OrderStrategyAssembly"];
    string className = ConfigurationManager.AppSettings["OrderStrategyClass"];

    // Using the evidence given in the config file load the appropriate assembly and
class
    return
(PetShop.IBLLStrategy.IOrderStrategy)Assembly.Load(path).CreateInstance(class
```

Name);
}
由于 OrderProcessor 是一个单独的应用程序，因此它使用的配置文件与 PetShop 不同，是存放在应用程序的 App.config 文件中，在该文件中，对 IOrderStategy 的配置为：

```
<add key="OrderStrategyAssembly" value="PetShop.BLL" />
<add key="OrderStrategyClass" value="PetShop.BLL.OrderSynchronous" />
```

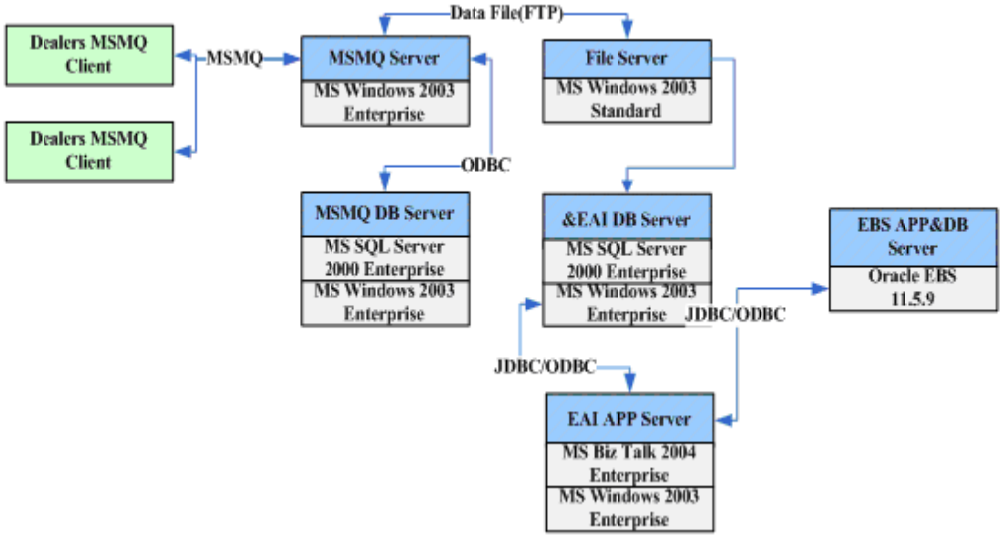
因此，以异步方式插入订单的流程如下图所示：



Microsoft Messaging Queue(MSMQ)技术除用于异步处理以外，它主要还是一种分布式处理技术。分布式处理中，一个重要的技术要素就是有关消息的处理，而在 System.Messaging 命名空间中，已经提供了 Message 类，可以用于承载消息的传递，前提上消息的发送方与接收方在数据定义上应有统一的接口规范。

MSMQ 在分布式处理的运用，在我参与的项目中已经有了实现。在为一个汽车制造商开发一个大型系统时，分销商 Dealer 作为.Net 客户端，需要将数据传递到管理中心，并且该数据将被 Oracle 的 EBS(E-Business System)使用。由于分销商管理系统（DMS）采用的是 C/S 结构，数据库为 SQL Server，而汽车制造商管理中心的 EBS 数据库为 Oracle。这里就涉及到两个系统之间数据的传递。

实现架构如下：



首先 Dealer 的数据通过 MSMQ 传递到 MSMQ Server，此时可以将数据插入到 SQL Server 数据库中，同时利用 FTP 将数据传送到专门的文件服务器上。然后利用 IBM 的 EAI 技术（企业应用集成，Enterprise Application Itegration）定期将文件服务器中的文件，利用接口规范写入到 EAI 数据库服务器中，并最终写道 EBS 的 Oracle 数据库中。
上述架构是一个典型的分布式处理结构，而技术实现的核心就是 MSMQ 和 EAI。由于我们已经定义了统一的接口规范，在通过消息队列形成文件后，此时的数据就已经与平台无关了，使得在.Net 平台下的分销商管理系统能够与 Oracle 的 EBS 集成起来，完成数据的处理。

petshop4.0 详解之四(PetShop 之 ASP.NET 缓存)

如果对微型计算机硬件系统有足够的了解，那么我们对于 Cache 这个名词一定是耳熟能详的。在 CPU 以及主板的芯片中，都引入了这种名为高速缓冲存储器（Cache）的技术。因为 Cache 的存取速度比内存快，因而引入 Cache 能够有效的解决 CPU 与内存之间的速度不匹配问题。硬件系统可以利用 Cache 存储 CPU 访问概率高的那些数据，当 CPU 需要访问这些数据时，可以直接从 Cache 中读取，而不必访问存取速度相对较慢的内存，从而提高了 CPU 的工作效率。软件设计借鉴了硬件设计中引入缓存的机制以改善整个系统的性能，尤其是对于一个数据库驱动的 Web 应用程序而言，缓存的利用是不可或缺的，毕竟，数据库查询可能是整个 Web 站点中调用最频繁但同时又是执行最缓慢的操作之一，我们不能被它老迈的双腿拖缓我们前进的征程。缓存机制正是解决这一缺陷的加速器。

4.1 ASP.NET 缓存概述

作为.Net 框架下开发 Web 应用程序的主打产品，ASP.NET 充分考虑了缓存机制。通过某种方法，将系统需要的数据对象、Web 页面存储在内存中，使得 Web 站点在需要获取这些数据时，不需要经过繁琐的数据库连接、查询和复杂的逻辑运算，就可以“触手可及”，如“探囊取物”般容易而快速，从而提高整个 Web 系统的性能。

ASP.NET 提供了两种基本的缓存机制来提供缓存功能。一种是应用程序缓存，它允许开发者将程序生成的数据或报表业务对象放入缓存中。另外一种缓存机制是页输出缓存，利用它，可以直接获取存放在缓存中的页面，而不需要经过繁杂的对该页面的再次处理。

应用程序缓存其实现原理说来平淡无奇，仅仅是通过 ASP.NET 管理内存中的缓存空间。放入缓存中的应用程序数据对象，以键/值对的方式存储，这便于用户在访问缓存中的数据项时，可以根据 key 值判断该项是否存在缓存中。

放入在缓存中的数据对象其生命周期是受到限制的，即使在整个应用程序的生命周期里，也不能保证该数据对象一直有效。ASP.NET 可以对应用程序缓存进行管理，例如当数据项无效、过期或内存不足时移除它们。此外，调用者还可以通过 CacheItemRemovedCallback 委托，定义回调方法使得数据项被移除时能够通知用户。

在.Net Framework 中，应用程序缓存通过 System.Web.Caching.Cache 类实现。它是一个密封类，不能被继承。对于每一个应用程序域，都要创建一个 Cache 类的实例，其生命周期与应用程序域的生命周期保持一致。我们可以利用 Add 或 Insert 方法，将数据项添加到应用程序缓存中，如下所示：

```
Cache["First"] = "First Item";
Cache.Insert("Second", "Second Item");
```

我们还可以为应用程序缓存添加依赖项，使得依赖项发生更改时，该数据项能够从缓存中移除：

```
string[] dependencies = {"Second"};
Cache.Insert("Third", "Third Item",
new System.Web.Caching.CacheDependency(null, dependencies));
```

与之对应的是缓存中数据项的移除。前面提到 ASP.NET 可以自动管理缓存中项的移除，但我们也可以通过代码编写的方式显式的移除相关的数据项：

```
Cache.Remove("First");
```

相对于应用程序缓存而言，页输出缓存的应用更为广泛。它可以通过内存将处理后的 ASP.NET 页面存储起来，当客户端再一次访问该页面时，可以省去页面处理的过程，从而提高页面访问的性能，以及 Web 服务器的吞吐量。例如，在一个电子商务网站里，用户需要经常查询商品信息，这个过程会涉及到数据库访问以及搜索条件的匹配，在数据量较大的情况下，如此的搜索过程是较为耗时的。此时，利用页输出缓存就可以将第一次搜索得到的查询结果页存储在缓存中。当用户第二次查询时，就可以省去数据查询的过程，减少页面的响应时间。

页输出缓存分为整页缓存和部分页缓存。我们可以通过@OutputCache 指令完成对 Web 页面的输出缓存。它主要包含两个参数：Duration 和 VaryByParam。Duration 参数用于设置页面或控件进行缓存的时间，其单位为秒。如下的设置表示缓存在 60 秒内有效：

```
<%@ OutputCache Duration=“60“ VaryByParam=“none“ %>
```

只要没有超过 Duration 设置的期限值，当用户访问相同的页面或控件时，就可以直接在缓存中获取。

使用 VaryByParam 参数可以根据设置的参数值建立不同的缓存。例如在一个输出天气预报结果的页面中，如果需要为一个 ID 为 txtCity 的 TextBox 控件建立缓存，其值将显示某城市的气温，那么我们可以进行如下的设置：

```
<%@ OutputCache Duration=”60” VaryByParam=”txtCity” %>
```

如此一来，ASP.NET 会对 txtCity 控件的值进行判断，只有输入的值与缓存值相同，才从缓存中取出相应的值。这就有效地避免了因为值的不同而导致输出错误的数据。

利用缓存的机制对性能的提升非常明显。通过 ACT（Application Center Test）的测试，可以发现设置缓存后执行的性能比未设置缓存时的性能足足提高三倍多。

引入缓存看来是提高性能的“完美”解决方案，然而“金无足赤，人无完人”，缓存机制也有缺点，那就是数据过期的问题。一旦应用程序数据或者页面结果值发生的改变，那么在缓存有效期范围内，你所获得的结果将是过期的、不准确的数据。我们可以想一想股票系统利用缓存所带来的灾难，当你利用错误过期的数据去分析股市的风云变幻时，你会发现获得的结果真可以说是“失之毫厘，谬以千里”，看似大好的局面就会像美丽的泡沫一样，用针一戳，转眼就消失得无影无踪。

那么我们是否应该为了追求高性能，而不顾所谓“数据过期”所带来的隐患呢？显然，在类似于股票系统这种数据更新频繁的特定场景下，数据过期的糟糕表现甚至比低效的性能更让人难以接受。故而，我们需要在性能与数据正确性间作出权衡。所幸的是，.Net Framework 2.0 引入了一种新的缓存机制，它为我们的“鱼与熊掌兼得”带来了技术上的可行性。

.Net 2.0 引入的自定义缓存依赖项，特别是基于 MS-SQL Server 的 SqlCacheDependency 特性，使得我们可以避免“数据过期”的问题，它能够根据数据库中相应数据的变化，通知缓存，并移除那些过期的数据。事实上，在 PetShop 4.0 中，就充分地利用了 SqlCacheDependency 特性。

4.2 SqlCacheDependency 特性

SqlCacheDependency 特性实际上是通过 System.Web.Caching.SqlCacheDependency 类来体现的。通过该类，可以在所有支持的 SQL Server 版本（7.0，2000，2005）上监视特定的 SQL Server 数据库表，并创建依赖于该表以及表中数据行的缓存项。当数据表或表中特定行的数据发生更改时，具有依赖项的数据项就会失效，并自动从 Cache 中删除该项，从而保证了缓存中不再保留过期的数据。
由于版本的原因，SQL Server 2005 完全支持 SqlCacheDependency 特性，但对于 SQL Server 7.0 和 SQL Server 2000 而言，就没有如此幸运了。毕竟这些产品出现在.Net Framework 2.0 之前，因此它并没有实现自动监视数据表数据变化，通知 ASP.NET 的功能。解决的办法就是利用轮询机制，通过 ASP.NET 进程内的一个线程以指定的时间间隔轮询 SQL Server 数据库，以跟踪数据的变化情况。

要使得 7.0 或者 2000 版本的 SQL Server 支持 SqlCacheDependency 特性，需要对数据库服务器执行相关的配置步骤。有两种方法配置 SQL Server：使用 aspnet_regsql 命令行工具，或者使用 SqlCacheDependencyAdmin 类。

4.2.1 利用 aspnet_regsql 工具

aspnet_regsql 工具位于 Windows\Microsoft.NET\Framework\[版本]文件夹中。如果直接双击该工具的执行文件，会弹出一个向导对话框，提示我们完成相应的操作：

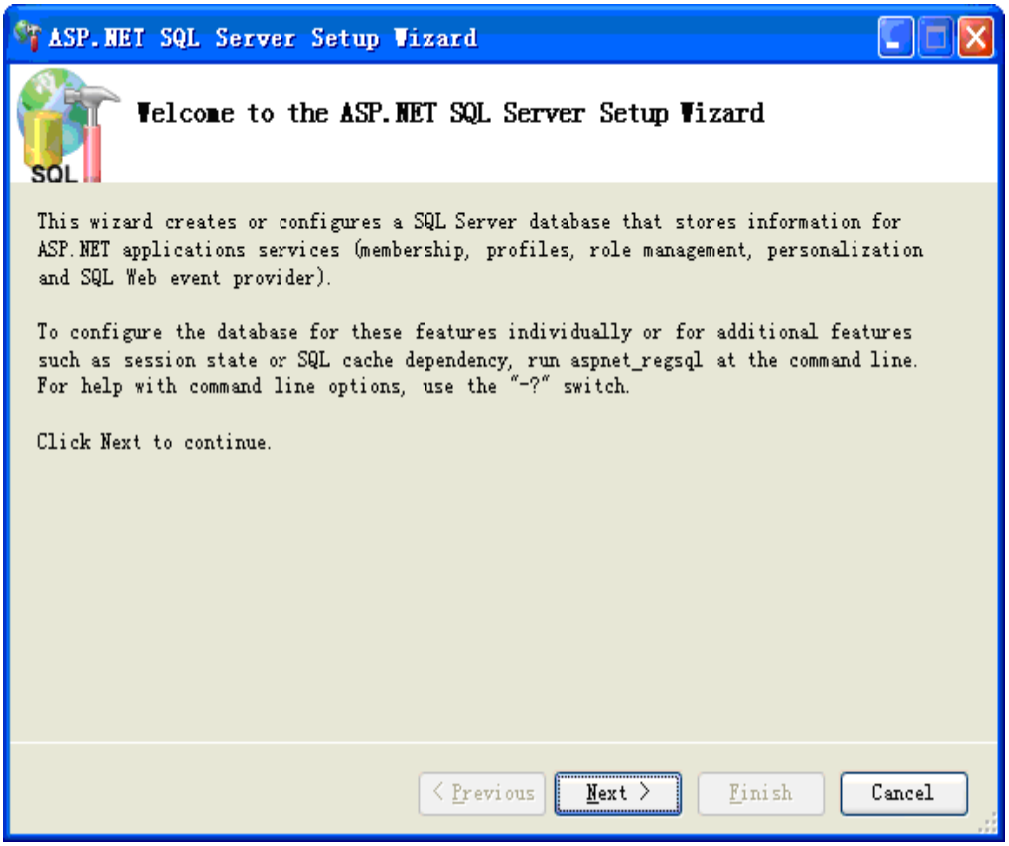


图 4-1 aspnet_regsql 工具

如图 4-1 所示中的提示信息，说明该向导主要用于配置 SQL Server 数据库，如 membership，profiles 等信息，如果要配置 SqlCacheDependency，则需要以命令行的方式执行。以 PetShop 4.0 为例，数据库名为 MSPetShop4，则命令为：

```
aspnet_regsql -S localhost -E -d MSPetShop4 -ed
```

以下是该工具的命令参数说明：

-? 显示该工具的帮助功能；
-S 后接的参数为数据库服务器的名称或者 IP 地址；
-U 后接的参数为数据库的登陆用户名；
-P 后接的参数为数据库的登陆密码；
-E 当使用 windows 集成验证时，使用该功能；
-d 后接参数为对哪一个数据库采用 SqlCacheDependency 功能；
-t 后接参数为对哪一个表采用 SqlCacheDependency 功能；
-ed 允许对数据库使用 SqlCacheDependency 功能；
-dd 禁止对数据库采用 SqlCacheDependency 功能；
-et 允许对数据表采用 SqlCacheDependency 功能；
-dt 禁止对数据表采用 SqlCacheDependency 功能；
-lt 列出当前数据库中有哪些表已经采用 sqlcachedependency 功能。

以上面的命令为例，说明将对名为 MSPetShop4 的数据库采用 SqlCacheDependency 功能，且 SQL Server 采用了 windows 集成验证方式。我们还可以对相关的数据表执行 aspnet_regsql 命令，如：

```
aspnet_regsql -S localhost -E -d MSPetShop4 -t Item -et
aspnet_regsql -S localhost -E -d MSPetShop4 -t Product -et
aspnet_regsql -S localhost -E -d MSPetShop4 -t Category -et
```

当执行上述的四条命令后，aspnet_regsql 工具会在 MSPetShop4 数据库中建立一个名为 AspNet_SqlCacheTablesForChangeNotification 的新数据库表。该数据表包含三个字段。字段 tableName 记录要追踪的数据表的名称，例如在 PetShop 4.0 中，要记录的数据表就包括 Category、Item 和 Product。notificationCreated 字段记录开始追踪的时间。changeId 作为一个类型为 int 的字段，用于记录数据表数据发生变化的次数。如图 4-2 所示：

	tableName	notificationCreated	changeId
▶	Category	2006-8-29 9:41:52	5
	Item	2006-8-29 9:41:52	82
	Product	2006-8-29 9:41:52	31
*			

图 4-2 AspNet_SqlCacheTablesForChangeNotification 数据表

除此之外，执行该命令还会为 MSPetShop4 数据库添加一组存储过程，为 ASP.NET 提供查询追踪的数据表的情况，同时还将为使用了 SqlCacheDependency 的表添加触发器，分别对应 Insert、Update、Delete 等与数据更改相关的操作。例如 Product 数据表的触发器：

```
Create TRIGGER dbo.[Product_AspNet_SqlCacheNotification_Trigger] ON [Product]
    FOR Insert, Update, Delete AS BEGIN
    SET NOCOUNT ON
    EXEC dbo.AspNet_SqlCacheUpdateChangeIdStoredProcedure N'Product'
END
```

其中，AspNet_SqlCacheUpdateChangeIdStoredProcedure 即是工具添加的一组存储过程中的一个。当对 Product 数据表执行 Insert、Update 或 Delete 等操作时，就会激活触发器，然后执行 AspNet_SqlCacheUpdateChangeIdStoredProcedure 存储过程。其执行的过程就是修改 AspNet_SqlCacheTablesForChangeNotification 数据表的 changeId 字段值：

```
Create PROCEDURE dbo.AspNet_SqlCacheUpdateChangeIdStoredProcedure
        @tableName NVARCHAR(450)
    AS
    BEGIN
        Update dbo.AspNet_SqlCacheTablesForChangeNotification WITH
(ROWLOCK) SET changeId = changeId + 1
        Where tableName = @tableName
    END
GO
```

4.2.2 利用 SqlCacheDependencyAdmin 类

我们也可以利用编程的方式来来管理数据库对 SqlCacheDependency 特性的使用。该类包含了五个重要的方法：

DisableNotifications	为特定数据库禁用 SqlCacheDependency

	对象更改通知
DisableTableForNotifications	为数据库中的特定表禁用 SqlCacheDependency 对象更改通知
EnableNotifications	为特定数据库启用 SqlCacheDependency 对象更改通知
EnableTableForNotifications	为数据库中的特定表启用 SqlCacheDependency 对象更改通知
GetTablesEnabledForNotifications	返回启用了 SqlCacheDependency 对象更改通知的所有表的列表

表 4-1 SqlCacheDependencyAdmin 类的主要方法

假设我们定义了如下的数据库连接字符串：

```
const string connectionStr = "Server=localhost;Database=MSPetShop4";
```

那么为数据库 MSPetShop4 启用 SqlCacheDependency 对象更改通知的实现为：

```
protected void Page_Load(object sender, EventArgs e)
{
    if (!IsPostBack)
   {
      SqlCacheDependencyAdmin.EnableNotifications(connectionStr);
   }
}
```

为数据表 Product 启用 SqlCacheDependency 对象更改通知的实现则为：

```
SqlCacheDependencyAdmin.EnableTableForNotifications(connectionStr,
"Product");
```

如果要调用表 4-1 中所示的相关方法，需要注意的是访问 SQL Server 数据库的帐户必须具有创建表和存储过程的权限。如果要调用 EnableTableForNotifications 方法，还需要具有在该表上创建 SQL Server 触发器的权限。

虽然说编程方式赋予了程序员更大的灵活性，但 aspnet_regsql 工具却提供了更简单的方法实现对 SqlCacheDependency 的配置与管理。PetShop 4.0 采用的正是 aspnet_regsql 工具的办法，它编写了一个文件名为 InstallDatabases.cmd 的批处理文件，其中包含了对 aspnet_regsql 工具的执行，并通过安装程序去调用该文件，实现对 SQL Server 的配置。

4.3 在 PetShop 4.0 中 ASP.NET 缓存的实现

PetShop 作为一个 B2C 的宠物网上商店，需要充分考虑访客的用户体验，如果因为数据量大而导致 Web 服务器的响应不及时，页面和查询数据迟迟得不到结果，会因此而破坏客户访问网站的心情，在耗尽耐心的等待后，可能会失去这一部分客户。无疑，这是非常糟糕的结果。因而在对其进行体系架构设计时，整个系统的性能就显得殊为重要。然而，我们不能因噎废食，因为专注于性能而忽略数据的正确性。在 PetShop 3.0 版本以及之前的版本，因为 ASP.NET 缓存的局限性，这一问题并没有得到很好的解决。PetShop 4.0 则引入了 SqlCacheDependency 特性，使得系统对缓存的处理较之以前大为改观。

4.3.1 CacheDependency 接口

PetShop 4.0 引入了 SqlCacheDependency 特性，对 Category、Product 和 Item 数据表对应的缓存实施了 SQL Cache Invalidation 技术。当对应的数据表数据发生更改后，该技术能够将相关项从缓存中移除。实现这一技术的核心是 SqlCacheDependency 类，它继承了 CacheDependency 类。然而为了保证整个架构的可扩展性，我们也允许设计者建立自定义的 CacheDependency 类，用以扩展缓存依赖。这就有必要为 CacheDependency 建立抽象接口，并在 web.config 文件中进行配置。

在 PetShop 4.0 的命名空间 PetShop.ICacheDependency 中，定义了名为 IPetShopCacheDependency 接口，它仅包含了一个接口方法：

```
public interface IPetShopCacheDependency
{
    AggregateCacheDependency GetDependency();
}
```

AggregateCacheDependency 是.Net Framework 2.0 新增的一个类，它负责监视依赖项对象的集合。当这个集合中的任意一个依赖项对象发生改变时，该依赖项对象对应的缓存对象都将被自动移除。
AggregateCacheDependency 类起到了组合 CacheDependency 对象的作用，它可以将多个 CacheDependency 对象甚至于不同类型的 CacheDependency 对象与缓存项建立关联。由于 PetShop 需要为 Category、Product 和 Item 数据表建立依赖项，因而 IPetShopCacheDependency 的接口方法 GetDependency()其目的就是返回建立了这些依赖项的 AggregateCacheDependency 对象。

4.3.2 CacheDependency 实现

CacheDependency 的实现正是为 Category、Product 和 Item 数据表建立了对应的 SqlCacheDependency 类型的依赖项，如代码所示：

```
public abstract class TableDependency : IPetShopCacheDependency
{
    // This is the separator that's used in web.config
    protected char[] configurationSeparator = new char[] { ',' };

    protected AggregateCacheDependency dependency = new
AggregateCacheDependency();
    protected TableDependency(string configKey)
```

```
    {
        string dbName =
ConfigurationManager.AppSettings["CacheDatabaseName"];
        string tableConfig = ConfigurationManager.AppSettings[configKey];
        string[] tables = tableConfig.Split(configurationSeparator);

        foreach (string tableName in tables)
            dependency.Add(new SqlCacheDependency(dbName, tableName));
    }
    public AggregateCacheDependency GetDependency()
   {
        return dependency;
    }
}
```

需要建立依赖项的数据库与数据表都配置在 web.config 文件中，其设置如下：

```
<add key="CacheDatabaseName" value="MSPetShop4"/>
<add key="CategoryTableDependency" value="Category"/>
<add key="ProductTableDependency" value="Product,Category"/>
<add key="ItemTableDependency" value="Product,Category,Item"/>
```

根据各个数据表间的依赖关系，因而不同的数据表需要建立的依赖项也是不相同的，从配置文件中的 value 值可以看出。然而不管建立依赖项的多寡，其创建的行为逻辑都是相似的，因而在设计时，抽象了一个共同的类 TableDependency，并通过建立带参数的构造函数，完成对依赖项的建立。由于接口方法 GetDependency()的实现中，返回的对象 dependency 是在受保护的构造函数创建的，因此这里的实现方式也可以看作是 Template Method 模式的灵活运用。例如 TableDependency 的子类 Product，就是利用父类的构造函数建立了 Product、Category 数据表的 SqlCacheDependency 依赖：

```
public class Product：TableDependency
{
    public Product()：base("ProductTableDependency") { }
}
```

如果需要自定义 CacheDependency，那么创建依赖项的方式又有不同。然而不管是创建 SqlCacheDependency 对象，还是自定义的 CacheDependency 对象，都是将这些依赖项添加到 AggregateCacheDependency 类中，因而我们也可以为自定义 CacheDependency 建立专门的类，只要实现 IPetShopCacheDependency 接口即可。

4.3.3 CacheDependency 工厂

继承了抽象类 TableDependency 的 Product、Category 和 Item 类均需要在调用时创建各自的对象。由于它们的父类 TableDependency 实现了接口 IPetShopCacheDependency，因而它们也间接实现了 IPetShopCacheDependency 接口，这为实现工厂模式提供了前提。

在 PetShop 4.0 中，依然利用了配置文件和反射技术来实现工厂模式。命名空间 PetShop.CacheDependencyFactory 中，类 DependencyAccess 即为创建 IPetShopCacheDependency 对象的工厂类：

```
public static class DependencyAccess
{
    public static IPetShopCacheDependency CreateCategoryDependency()
    {
        return LoadInstance("Category");
    }
    public static IPetShopCacheDependency CreateProductDependency()
    {
        return LoadInstance("Product");
    }
    public static IPetShopCacheDependency CreateItemDependency()
    {
        return LoadInstance("Item");
    }
    private static IPetShopCacheDependency LoadInstance(string className)
    {
        string path =
ConfigurationManager.AppSettings["CacheDependencyAssembly"];
        string fullyQualifiedClass = path + "." + className;
        return
(IPetShopCacheDependency)Assembly.Load(path).CreateInstance(fullyQualifiedCl
ass);
    }
}
```

整个工厂模式的实现如图 4-3 所示：

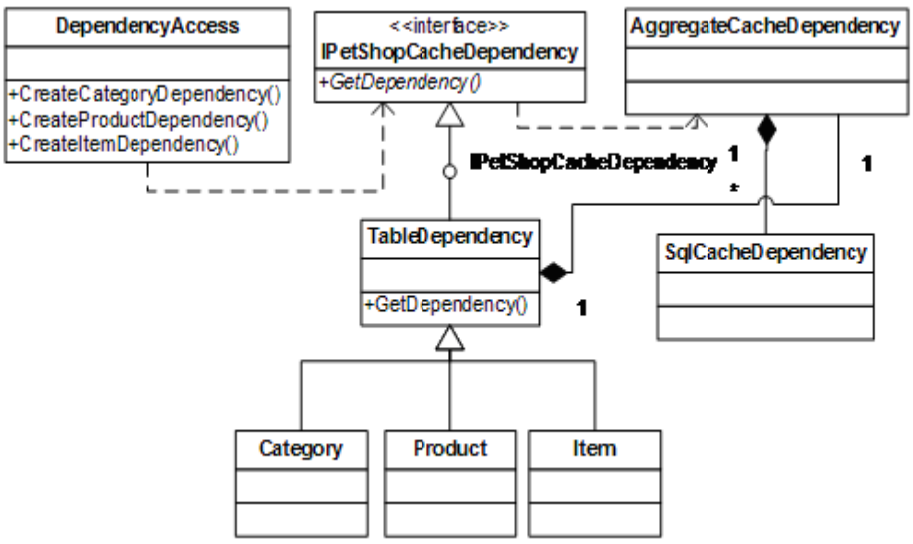


图 4-3 CacheDependency 工厂

虽然DependencyAccess类创建了实现了IPetShopCacheDependency接口的类Category、Product、Item，然而我们之所以引入 IPetShopCacheDependency 接口，其目的就在于获得创建了依赖项的 AggregateCacheDependency 类型的对象。我们可以调用对象的接口方法 GetDependency()，如下所示：

```
AggregateCacheDependency dependency =
DependencyAccess.CreateCategoryDependency().GetDependency();
```

为了方便调用者，似乎我们可以对 DependencyAccess 类进行改进，将原有的 CreateCategoryDependency()方法，修改为创建 AggregateCacheDependency 类型对象的方法。

然而这样的做法扰乱了作为工厂类的 DependencyAccess 的本身职责，且创建 IPetShopCacheDependency 接口对象的行为仍然有可能被调用者调用，所以保留原有的 DependencyAccess 类仍然是有必要的。

在 PetShop 4.0 的设计中，是通过引入 Facade 模式以方便调用者更加简单地获得 AggregateCacheDependency 类型对象。

4.3.4 引入 Facade 模式

利用 Facade 模式可以将一些复杂的逻辑进行包装，以方便调用者对这些复杂逻辑的调用。就好像提供一个统一的门面一般，将内部的子系统封装起来，统一为一个高层次的接口。一个典型的 Facade 模式示意图如下所示：

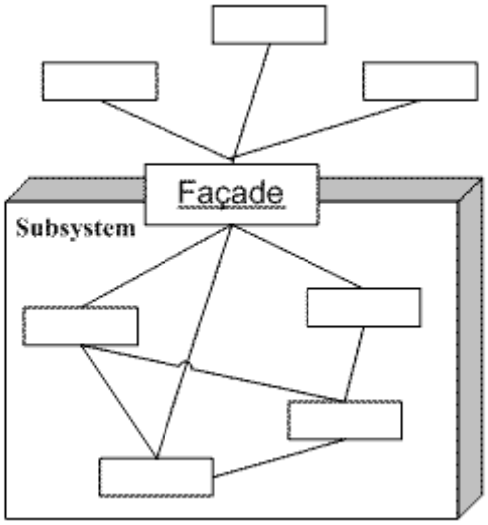


图 4-4 Facade 模式

Facade 模式的目的并非要引入一个新的功能，而是在现有功能的基础上提供一个更高层次的抽象，使得调用者可以直接调用，而不用关心内部的实现方式。以 CacheDependency 工厂为例，我们需要为调用者提供获得 AggregateCacheDependency 对象的简便方法，因而创建了 DependencyFacade 类：

```
public static class DependencyFacade
```

```
{
    private static readonly string path =
ConfigurationManager.AppSettings["CacheDependencyAssembly"];
    public static AggregateCacheDependency GetCategoryDependency()
    {
        if (!string.IsNullOrEmpty(path))
            return
DependencyAccess.CreateCategoryDependency().GetDependency();
        else
            return null;
    }
    public static AggregateCacheDependency GetProductDependency()
    {
        if (!string.IsNullOrEmpty(path))
            return DependencyAccess.CreateProductDependency().GetDependency();
        else
            return null;
        }
    public static AggregateCacheDependency GetItemDependency()
    {
        if (!string.IsNullOrEmpty(path))
            return DependencyAccess.CreateItemDependency().GetDependency();
        else
            return null;
    }
}
```

DependencyFacade 类封装了获取 AggregateCacheDependency 类型对象的逻辑，如此一来，调用者可以调用相关方法获得创建相关依赖项的 AggregateCacheDependency 类型对象：

```
AggregateCacheDependency dependency =
DependencyFacade.GetCategoryDependency();
```

比起直接调用 DependencyAccess 类的 GetDependency()方法而言，除了方法更简单之外，同时它还对 CacheDependencyAssembly 配置节进行了判断，如果其值为空，则返回 null 对象。

在 PetShop.Web 的 App_Code 文件夹下，静态类 WebUtility 的 GetCategoryName()和 GetProductName()方法调用了 DependencyFacade 类。例如 GetCategoryName()方法：

```
public static string GetCategoryName(string categoryId)
{
    Category category = new Category();
    if (!enableCaching)
        return category.GetCategory(categoryId).Name;

    string cacheKey = string.Format(CATEGORY_NAME_KEY, categoryId);
```

```
    // 检查缓存中是否存在该数据项;
    string data = (string)HttpRuntime.Cache[cacheKey];
    if (data == null)
    {
        // 通过 web.config 的配置获取 duration 值;
        int cacheDuration =
int.Parse(ConfigurationManager.AppSettings["CategoryCacheDuration"]);
        // 如果缓存中不存在该数据项，则通过业务逻辑层访问数据库获取;
        data = category.GetCategory(categoryId).Name;
        // 通过 Facade 类创建 AggregateCacheDependency 对象;
        AggregateCacheDependency cd =
DependencyFacade.GetCategoryDependency();
        // 将数据项以及 AggregateCacheDependency 对象存储到缓存中;
        HttpRuntime.Cache.Add(cacheKey, data, cd,
DateTime.Now.AddHours(cacheDuration), Cache.NoSlidingExpiration,
CacheItemPriority.High, null);
     }
     return data;
}
```

GetCategoryName()方法首先会检查缓存中是否已经存在 CategoryName 数据项，如果已经存在，就通过缓存直接获取数据；否则将通过业务逻辑层调用数据访问层访问数据库获得 CategoryName，在获得了 CategoryName 后，会将新获取的数据连同 DependencyFacade 类创建的 AggregateCacheDependency 对象添加到缓存中。

WebUtility 静态类被表示层的许多页面所调用，例如 Product 页面：

```
public partial class Products：System.Web.UI.Page
{
   protected void Page_Load(object sender, EventArgs e)
   {
      Page.Title =
WebUtility.GetCategoryName(Request.QueryString["categoryId"]);
   }
}
```

显示页面 title 的逻辑是放在 Page_Load 事件方法中，因而每次打开该页面都要执行获取 CategoryName 的方法。如果没有采用缓存机制，当 Category 数据较多时，页面的显示就会非常缓慢。

4.3.5 引入 Proxy 模式

业务逻辑层 BLL 中与 Product、Category、Item 有关的业务方法，其实现逻辑是调用数据访问层（DAL）对象访问数据库，以获取相关数据。为了改善系统性能，我们就需要为这些实现方法增加缓存机制的逻辑。当我们操作增加了缓存机制的业务对象时，对于调用者而言，应与 BLL

业务对象的调用保持一致。也即是说，我们需要引入一个新的对象去控制原来的 BLL 业务对象，这个新的对象就是 Proxy 模式中的代理对象。

以 PetShop.BLL.Product 业务对象为例，PetShop 为其建立了代理对象 ProductDataProxy，并在 GetProductByCategory()等方法中，引入了缓存机制，例如：

```
public static class ProductDataProxy
{

    private static readonly int productTimeout =
int.Parse(ConfigurationManager.AppSettings["ProductCacheDuration"]);
    private static readonly bool enableCaching =
bool.Parse(ConfigurationManager.AppSettings["EnableCaching"]);

    public static IList
GetProductsByCategory(string category)
    {
        Product product = new Product();

        if (!enableCaching)
            return product.GetProductsByCategory(category);

        string key = "product_by_category_" + category;
        IList data = (IList )HttpRuntime.Cache[key];

        // Check if the data exists in the data cache
        if (data == null)
        {
            data = product.GetProductsByCategory(category);

            // Create a AggregateCacheDependency object from the factory
            AggregateCacheDependency cd =
DependencyFacade.GetProductDependency();

            // Store the output in the data cache, and Add the necessary
AggregateCacheDependency object
            HttpRuntime.Cache.Add(key, data, cd,
DateTime.Now.AddHours(productTimeout), Cache.NoSlidingExpiration,
CacheItemPriority.High, null);
        }
        return data;
    }
}
```

与业务逻辑层 Product 对象的 GetProductsByCategory()方法相比，增加了缓存机制。当缓存内不存在相关数据项时，则直接调用业务逻辑层 Product 的 GetProductsByCategory()方法来获取数据，并将其与对应的 AggregateCacheDependency 对象一起存储在缓存中。

引入 Proxy 模式，实现了在缓存级别上对业务对象的封装，增强了对业务对象的控制。由于暴露在对象外的方法是一致的，因而对于调用方而言，调用代理对象与真实对象并没有实质的区别。

从职责分离与分层设计的角度分析，我更希望这些 Proxy 对象是被定义在业务逻辑层中，而不像在 PetShop 的设计那样，被划分到表示层 UI 中。此外，如果需要考虑程序的可扩展性与可替换性，我们还可以为真实对象与代理对象建立统一的接口或抽象类。然而，单以 PetShop 的表示层调用来看，采用静态类与静态方法的方式，或许更为合理。我们需要谨记，“过度设计”是软件设计的警戒线。

如果需要对 UI 层采用缓存机制，将应用程序数据存放到缓存中，就可以调用这些代理对象。以 ProductsControl 用户控件为例，调用方式如下：

```
productsList.DataSource =
ProductDataProxy.GetProductsByCategory(categoryKey);
```

productsList 对象属于自定义的 CustomList 类型，这是一个派生自 System.Web.UI.WebControls.DataList 控件的类，它的 DataSource 属性可以接受 IList 集合对象。

不过在 PetShop 4.0 的设计中，对于类似于 ProductsControl 类型的控件而言，采用的缓存机制是页输出缓存。我们可以从 ProductsControl.ascx 页面的 Source 代码中发现端倪：

```
<%@ OutputCache Duration="100000" VaryByParam="page;categoryId" %>
```

与 ASP.NET 1.x 的页输出缓存不同的是，在 ASP.NET 2.0 中，为 ASP.NET 用户控件新引入了 CachePolicy 属性，该属性的类型为 ControlCachePolicy 类，它以编程方式实现了对 ASP.NET 用户控件的输出缓存设置。我们可以通过设置 ControlCachePolicy 类的 Dependency 属性，来设置与该用户控件相关的依赖项，例如在 ProductsControl 用户控件中，进行如下的设置：

```
protected void Page_Load(object sender, EventArgs e)
{
    this.CachePolicy.Dependency = DependencyFacade.GetProductDependency();
}
```

采用页输出缓存，并且利用 ControlCachePolicy 设置输出缓存，能够将业务数据与整个页面放入到缓存中。这种方式比起应用程序缓存而言，在性能上有很大的提高。同时，它又通过引入的 SqlCacheDependency 特性有效地避免了“数据过期”的缺点，因而在 PetShop 4.0 中被广泛采用。相反，之前为 Product、Category、Item 业务对象建立的代理对象则被“投闲散置”，仅仅作为一种设计方法的展示而“幸存”与整个系统的源代码中。

petshop4.0 详解之五(PetShop 之业务逻辑层设计)

五 PetShop 之业务逻辑层设计

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，也即是说它是与系统所应对的领域（Domain）逻辑有关，很多时候，我们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、领域层和数据源层。作为领域驱动设计的先驱 Eric Evans，对业务逻辑层作了更细致地划分，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，它却是被调用者。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。

5.1 与领域专家合作

设计业务逻辑层最大的障碍不在于技术，而在于对领域业务的分析与理解。很难想象一个不熟悉该领域业务规则和流程的架构设计师能够设计出合乎客户需求的系统架构。几乎可以下定结论的是，业务逻辑层的设计过程必须有领域专家的参与。在我曾经参与开发的项目中，所涉及的领域就涵盖了电力、半导体、汽车等诸多行业，如果缺乏这些领域的专家，软件架构的设计尤其是业务逻辑层的设计就无从谈起。这个结论唯一的例外是，架构设计师同时又是该领域的专家。然而，正所谓“千军易得，一将难求”，我们很难寻觅到这样卓越出众的人才。

领域专家在团队中扮演的角色通常称为 Business Consultor（业务咨询师），负责提供与领域业务有关的咨询，与架构师一起参与架构与数据库的设计，撰写需求文档和设计用例（或者用户故事 User Story）。如果在测试阶段，还应该包括撰写测试用例。理想的状态是，领域专家应该参与到整个项目的开发过程中，而不仅仅是需求阶段。

领域专家可以是专门聘请的对该领域具有较深造诣的咨询师，也可以是作为需求提供方的客户。在极限编程（Extreme Programming）中，就将客户作为领域专家引入到整个开发团队中。它强调了现场客户原则。现场客户需要参与到计划游戏、开发迭代、编码测试等项目开发的各个阶段。由于领域专家与设计师以及开发人员组成了一个团队，贯穿开发过程的始终，就可以避免需求理解错误的情况出现。即使项目的开发与实际需求不符，也可以在项目早期及时修正，从而避免了项目不必要的延期，加强了对项目过程和成本的控制。正如 Steve McConnell 在构建活动的前期准备中提及的一个原则：发现错误的时间要尽可能接近引入该错误的时间。需求的缺陷在系统中潜伏的时间越长，代价就越昂贵。如果在项目开发中能够与领域专家充分的合作，就可以最大效果地规避这样一种恶性的链式反应。

传统的软件开发模型同样重视与领域专家的合作，但这种合作主要集中在需求分析阶段。例如瀑布模型，就非常强调早期计划与需求调研。然而这种未雨绸缪的早期计划方式，对架构师与需求调研人员的技能要求非常高，它强调需求文档的精确性，一旦分析出现偏差，或者需求发生变更，当项目开发进入设计阶段后，由于缺乏与领域专家沟通与合作的机制，开发人员估量不到这些错

误与误差，因而难以及时作出修正。一旦这些问题像毒瘤一般在系统中蔓延开来，逐渐暴露在开发人员面前时，已经成了一座难以逾越的高山。我们需要消耗更多的人力物力，才能够修正这些错误，从而导致开发成本成数量级的增加，甚至于导致项目延期。当然还有一个好的选择，就是放弃整个项目。这样的例子不胜枚举，事实上，项目开发的“滑铁卢”，究其原因，大部分都是因为业务逻辑分析上出现了问题。

迭代式模型较之瀑布模型有很大地改进，因为它允许变更、优化系统需求，整个迭代过程实际上就是与领域专家的合作过程，通过向客户演示迭代所产生的系统功能，从而及时获取反馈，并逐一解决迭代演示中出现的问题，保证系统向着合乎客户需求的方向演化。因而，迭代式模型往往能够解决早期计划不足的问题，它允许在发现缺陷的时候，在需求变更的时候重新设计、重新编码并重新测试。

无论采用何种开发模型，与领域专家的合作都将成为项目成败与否的关键。这基于一个软件开发的普遍真理，那就是世界上没有不变的需求。一句经典名言是：“没有不变的需求，世上的软件都改动过 3 次以上，唯一一个只改动过两次的软件的拥有者已经死了，死在去修改需求的路上。”一语道尽了软件开发的残酷与艰辛！

那么应该如何加强与领域专家的合作呢？James Carey 和 Brent Carlson 根据他们在参与的 IBM SanFrancisco 项目中获得的经验，提出了 Innocent Questions 模式，其意义即“改进领域专家和技术专家的沟通质量”。在一个项目团队中，如果我们没有一位既能担任首席架构师，同时又是领域专家的人选，那么加强领域专家与技术专家的合作就显得尤为重要了。毕竟，作为一个领域专家而言，可能并不熟悉软件设计方法学，也不具备面向对象开发和架构设计的能力，同样，大部分技术专家很有可能对该项目所涉及的业务领域仅停留在一知半解的地步。如果领域专家与技术专家不能有效沟通，则整个项目的前途就岌岌可危了。

Innocent Questions 模式提出的解决方案包括：
（1）选用可以与人和谐相处的人员组建开发团队；
（2）清楚地定义角色和职权；
（3）明确定义需要的交互点；
（4）保持团队紧密；
（5）雇佣优秀的人。

事实上，这已经从技术的角度上升到对团队的管理层次了。就好比篮球运动一样，即使你的球队集合了五名世界上最顶尖最有天赋的球员，如果各自为战，要想取得比赛的胜利依旧是非常困难的。团队精神与权责分明才是取得胜利的保障，软件开发同样如此。

与领域专家合作的基础是保证开发团队中永远保留至少一名领域专家。他可以是系统的客户，第三方公司的咨询师，最理想是自己公司雇佣的专家。如果项目中缺乏这样的一个人，那么我的建议是去雇佣他，如果你不想看到项目遭遇“西伯利亚寒流”的话。

确定领域专家的角色任务与职责。必须要让团队中的每一个人明确领域专家在整个团队中究竟扮演什么样的角色，他的职责是什么。一个合格的领域专家必须对业务领域有足够深入的理解，他应该是一个能够俯瞰整个系统需求、总揽全局的人物。在项目开发过程中，将由他负责业务规则和流程的制定，负责与客户的沟通，需求的调研与讨论，并于设计师一起参与系统架构的设计。

编档是领域专家必须参与的工作，无论是需求文档还是设计文档，以及用例的编写，领域专家或者提出意见，或者作为撰写的作者，至少他也应该是评审委员会的重要成员。

规范业务领域的术语和技术术语。领域专家和技术专家必须在保证不产生二义性的语义环境下进行沟通与交流。如果出现理解上的分歧，我们必须及时解决，通过讨论确立术语标准。很难想象两个语言不通的人能够相互合作愉快，解决的办法是加入一位翻译人员。在领域专家与技术专家之间搭建一座语义上的桥梁，使其能够相互理解、相互认同。还有一个办法是在团队内部开展培训活动。尤其对于开发人员而言，或多或少地了解一些业务领域知识，对于项目的开发有很大的帮助。在我参与过的半导体领域的项目开发，团队就专门邀请了半导体行业的专家就生产过程的业务逻辑进行了全方位的介绍与培训。正所谓“磨刀不误砍柴工”，虽然我们消费了培训的时间，但对于掌握了业务规则与流程的开发人员，却能够提升项目开发进度，总体上节约了开发成本。

加强与客户的沟通。客户同时也可以作为团队的领域专家，极限编程的现场客户原则是最好的示例。但现实并不都如此的完美，在无法要求客户成为开发团队中的固定一员时，聘请或者安排一个专门的领域专家，加强与客户的沟通，就显得尤为重要。项目可以通过领域专家获得客户的及时反馈。而通过领域专家去了解变更了的需求，会在最大程度上减少需求误差的可能。

5.2 业务逻辑层的模式应用

Martin Fowler 在《企业应用架构模式》一书中对领域层（即业务逻辑层）的架构模式作了整体概括，他将业务逻辑设计分为三种主要的模式：Transaction Script、Domain Model 和 Table Module。

Transaction Script 模式将业务逻辑看作是一个个过程，是比较典型的面向过程开发模式。应用 Transaction Script 模式可以不需要数据访问层，而是利用 SQL 语句直接访问数据库。为了有效地管理 SQL 语句，可以将与数据库访问有关的行为放到一个专门的 Gateway 类中。应用 Transaction Script 模式不需要太多面向对象知识，简单直接的特性是该模式全部价值之所在。因而，在许多业务逻辑相对简单的项目中，应用 Transaction Script 模式较多。

Domain Model 模式是典型的面向对象设计思想的体现。它充分考虑了业务逻辑的复杂多变，引入了 Strategy 模式等设计模式思想，并通过建立领域对象以及抽象接口，实现模式的可扩展性，并利用面向对象思想与身俱来的特性，如继承、封装与多态，用于处理复杂多变的业务逻辑。唯一制约该模式应用的是对象与关系数据库的映射。我们可以引入 ORM 工具，或者利用 Data Mapper 模式来完成关系向对象的映射。

与 Domain Model 模式相似的是 Table Module 模式，它同样具有面向对象设计的思想，唯一不同的是它获得的对象并非是单纯的领域对象，而是 DataSet 对象。如果为关系数据表与对象建立一个简单的映射关系，那么 Domain Model 模式就是为数据表中的每一条记录建立一个领域对象，而 Table Module 模式则是将整个数据表看作是一个完整的对象。虽然利用 DataSet 对象会丢失面向对象的基本特性，但它在为表示层提供数据源支持方面却有着得天独厚的优势。尤其是在.Net 平台下，ADO.NET 与 Web 控件都为 Table Module 模式提供了生长的肥沃土壤。

5.3 PetShop 的业务逻辑层设计

PetShop 在业务逻辑层设计中引入了 Domain Model 模式，这与数据访问层对于数据对象的支持是分不开的。由于 PetShop 并没有对宠物网上商店的业务逻辑进行深入，也省略了许多复杂细节的商务逻辑，因而在 Domain Model 模式的应用上并不明显。最典型地应该是对 Order 领域对象的处理方式，通过引入 Strategy 模式完成对插入订单行为的封装。关于这一点，我已在第 27 章有了详尽的描述，这里就不再赘述。

本应是系统架构设计中最核心的业务逻辑层，由于简化了业务流程的缘故，使得 PetShop 在这一层的设计有些乏善可陈。虽然在业务逻辑层中，针对 B2C 业务定义了相关的领域对象，但这些领域对象仅仅是完成了对数据访问层中数据对象的简单封装而已，其目的仅在于分离层次，以支持对各种数据库的扩展，同时将 SQL 语句排除在业务逻辑层外，避免了 SQL 语句的四处蔓延。

最能体现 PetShop 业务逻辑的除了对订单的管理之外，还包括购物车（Shopping Cart）与 Wish List 的管理。在 PetShop 的 BLL 模块中，定义了 Cart 类来负责相关的业务逻辑，定义如下：

```
[Serializable]
public class Cart
{
    private Dictionary cartItems = new Dictionary();
    public decimal Total
    {
        get
        {
            decimal total = 0;
            foreach (CartItemInfo item in cartItems.Values)
                total += item.Price * item.Quantity;
            return total;
        }
    }
    public void SetQuantity(string itemId, int qty)
    {
        cartItems[itemId].Quantity = qty;
    }
    public int Count
    {
        get { return cartItems.Count; }
    }
    public void Add(string itemId)
    {
        CartItemInfo cartItem;
        if (!cartItems.TryGetValue(itemId, out cartItem))
        {
            Item item = new Item();
            ItemInfo data = item.GetItem(itemId);
            if (data != null)
```

```
            {
                CartItemInfo newItem = new CartItemInfo(itemId, data.ProductName,
1, (decimal)data.Price, data.Name, data.CategoryId, data.ProductId);
                cartItems.Add(itemId, newItem);
            }
        }
        else
            cartItem.Quantity++;
    }
    //其他方法略;
}
```

Cart 类通过一个 Dictionary 对象来负责对购物车内容的存储，同时定义了 Add、Remove、Clear 等方法，来实现对购物车内容的管理。

在前面我提到 PetShop 业务逻辑层中的领域对象仅仅是完成对数据对象的简单封装，但这种分离层次的方法在架构设计中依然扮演了举足轻重的作用。以 Cart 类的 Add()方法为例，在方法内部引入了 PetShop.BLL.Item 领域对象，并调用了 Item 对象的 GetItem()方法。如果没有在业务逻辑层封装 Item 对象，而是直接调用数据访问层的 Item 数据对象，为保证层次间的弱依赖关系，就需要调用工厂对象的工厂方法来创建 PetShop.IDAL.IItem 接口类型对象。一旦数据访问层的 Item 对象被多次调用，就会造成重复代码，既不离于程序的修改与扩展，也导致程序结构生长为臃肿的态势。

此外，领域对象对数据访问层数据对象的封装，也有利于表示层对业务逻辑层的调用。在三层式架构中，表示层应该是对于数据访问层是“无知”的，这样既减少了层与层间的依赖关系，也能有效避免“循环依赖”的后果。

值得商榷的是 Cart 类的 Total 属性。其值的获取是通过遍历购物车集合，然后累加价格与商品数量的乘积。这里显然简化了业务逻辑，而没有充分考虑需求的扩展。事实上，这种获取购物车总价格的算法，在大多数情况下仅仅是其中的一种策略而已，我们还应该考虑折扣的情况。例如，当总价格超过 100 元时，可以给与顾客一定的折扣，这是与网站的促销计划相关的。除了给与折扣的促销计划外，网站也可以考虑赠送礼品的促销策略，因此我们有必要引入 Strategy 模式，定义接口 IOnSaleStrategy：

```
public interface IOnSaleStrategy
{
    decimal CalculateTotalPrice(Dictionary cartItems);
}
```

如此一来，我们可以为 Cart 类定义一个有参数的构造函数：

```
private IOnSaleStrategy m_onSale;
public Cart(IOnSaleStrategy onSale)
{
    m_onSale = onSale;
}
```

那么 Total 属性就可以修改为：

```
public decimal Total
{
    get {return m_onSale.CalculateTotalPrice(cartItems);}
}
```

如此一来，就可以使得 Cart 类能够有效地支持网站推出的促销计划，也符合开-闭原则。同样的，这种设计方式也是 Domain Model 模式的体现。修改后的设计如图 5-1 所示：

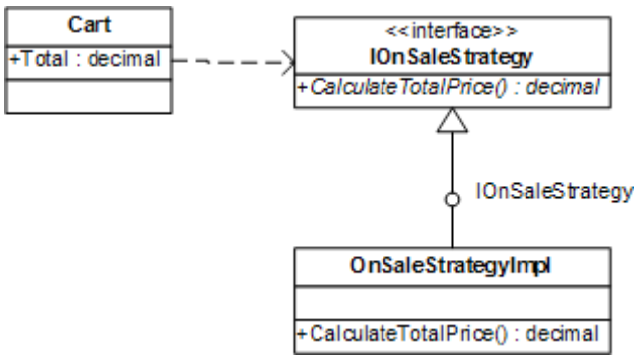


图 5-1 引入 Strategy 模式

作为一个 B2C 的电子商务架构，它所涉及的业务领域已为大部分设计师与开发人员所熟悉，因而在本例中，与领域专家的合作显得并不那么重要。然而，如果我们要开发一个成功的电子商务网站，与领域专家的合作仍然是必不可少的。以订单的管理而言，如果考虑复杂的商业应用，就需要管理订单的跟踪（Tracking），与网上银行的合作，账户安全性，库存管理，物流管理，以及客户关系管理（CRM）。整个业务过程却涵盖了诸如电子商务、银行、物流、客户关系学等诸多领域，如果没有领域专家的参与，业务逻辑层的设计也许会“败走麦城”。

5.4 与数据访问层的通信

业务逻辑层需要与数据访问层通信，利用数据访问层访问数据库，因此业务逻辑层与数据访问层之间就存在依赖关系。在数据访问层引入接口程序集以及数据工厂的设计前提下，能够做到两者间关系为弱依赖。我们从业务逻辑层的引用程序集中可以看到，BLL 模块并没有引用 SQLServerDAL 和 OracleDAL 程序集。在业务逻辑层中，有关数据访问层中数据对象的调用，均利用多态原理定义了抽象的接口类型对象，然后利用工厂对象的工厂方法创建具体的数据对象。如 PetShop.BLL.PetShop 领域对象所示：

```
namespace PetShop.BLL
{
    public class Product
    {
    //根据工厂对象创建 IProduct 接口类型实例;
        private static readonly IProduct dal
=   PetShop.DALFactory.DataAccess.CreateProduct();
        //调用 IProduct 对象的接口方法 GetProductByCategory();
  public IList
```

```
GetProductsByCategory(string category)
  {
  // 如果为空则新建 List 对象;
  if(string.IsNullOrEmpty(category))
   return new List ();

  // 通过数据访问层的数据对象访问数据库;
  return dal.GetProductsByCategory(category);
 }
      //其他方法略;
   }
}
```

在领域对象 Product 类中，利用数据访问层的工厂类 DALFactory.DataAccess 创建 PetShop.IDAL.IProduct 类型的实例，如此就可以解除对具体程序集 SQLServerDAL 或 OracleDAL 的依赖。只要 PetShop.IDAL 的接口方法不变，即使修改了 IDAL 接口模块的具体实现，都不会影响业务逻辑层的实现。这种松散的弱耦合关系，才能够最大程度地支持架构的可扩展。

领域对象 Product 实际上还完成了对数据对象 Product 的封装，它们暴露在外的接口方法是一致地，正是通过封装，使得表示层可以完全脱离数据库以及数据访问层，表示层的调用者仅需要关注业务逻辑层的实现逻辑，以及领域对象暴露的接口和调用方式。事实上，只要设计合理，规范了各个层次的接口方法，三层式架构的设计完全可以分离开由不同的开发人员同时开发，这就可以有效地利用开发资源，缩短项目开发周期。

5.5 面向接口设计

也许是业务逻辑比较简单地缘故，在业务逻辑层的设计中，并没有秉承在数据访问层中面向接口设计的思想。除了完成对插入订单策略的抽象外，整个业务逻辑层仅以 BLL 模块实现，没有为领域对象定义抽象的接口。因而 PetShop 的表示层与业务逻辑层就存在强依赖关系，如果业务逻辑层中的需求发生变更，就必然会影响表示层的实现。唯一可堪欣慰的是，由于我们采用分层式架构将用户界面与业务领域逻辑完全分离，一旦用户界面发生更改，例如将 B/S 架构修改为 C/S 架构，那么业务逻辑层的实现模块是可以完全重用的。

然而，最理想的方式仍然是面向接口设计。根据第 28 章对 ASP.NET 缓存的分析，我们可以将表示层 App_Code 下的 Proxy 类与 Utility 类划分到业务逻辑层中，并修改这些静态类为实例类，并将这些类中与业务领域有关的方法抽象为接口，然后建立如数据访问层一样的抽象工厂。通过“依赖注入”方式，解除与具体领域对象类的依赖，使得表示层仅依赖于业务逻辑层的接口程序集以及工厂模块。

那么，这样的设计是否有“过度设计”的嫌疑呢？我们需要依据业务逻辑的需求情况而定。此外，如果我们需要引入缓存机制，为领域对象创建代理类，那么为领域对象建立接口，就显得尤为必要。我们可以建立一个专门的接口模块 IBLL，用以定义领域对象的接口。以 Product 领域对象为例，我们可以建立 IProduct 接口：

```
public interface IProduct
```

```
{
   IList GetProductByCategory(string category);
   IList GetProductByCategory(string[] keywords);
   ProductInfo GetProduct(string productId);
}
```

在 BLL 模块中可以引入对 IBLL 程序集的依赖，则领域对象 Product 的定义如下：

```
public class Product:IProduct
{
  public IList GetProductByCategory(string category) { //实现略; }
  public IList GetProductByCategory(string[] keywords) { //实现略; }
  public ProductInfo GetProduct(string productId) { //实现略; }
}
```

然后我们可以为代理对象建立专门的程序集 BLLProxy，它不仅引入对 IBLL 程序集的依赖，同时还将依赖于 BLL 程序集。此时代理对象 ProductDataProxy 的定义如下：

```
using PetShop.IBLL;
using PetShop.BLL;
namespace PetShop.BLLProxy
{
  public class ProductDataProxy:IProduct
  {
    public IList GetProductByCategory(string category)
    {
      Product product = new Product();
      //其他实现略;
    }
    public IList GetProductByCategory(string[] keywords) { //实现略; }
    public ProductInfo GetProduct(string productId) { //实现略; }
  }
}
```

如此的设计正是典型的 Proxy 模式，其类结构如图 5-2 所示：

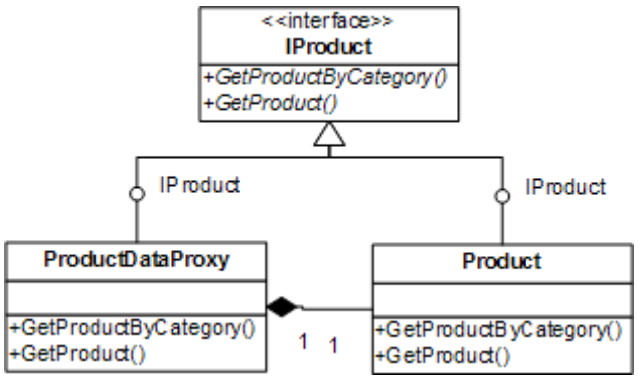


图 5-2 Proxy 模式

参照数据访问层的设计方法，我们可以为领域对象及代理对象建立抽象工厂，并在 web.config 中配置相关的配置节，然后利用反射技术创建具体的对象实例。如此一来，表示层就可以仅仅依赖 PetShop.IBLL 程序集以及工厂模块，如此就可以解除表示层与具体领域对象之间的依赖关系。表示层与修改后的业务逻辑层的关系如图 5-3 所示：

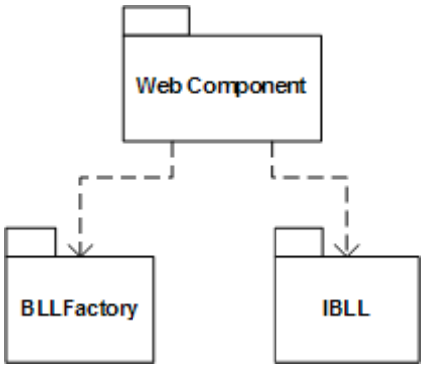


图 5-3 修改后的业务逻辑层与表示层的关系

图 5-4 则是 PetShop 4.0 原有设计的层次关系图：

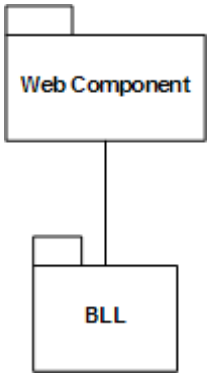


图 5-4 PetShop 4.0 中表示层与业务逻辑层的关系

通过比较图 5-3 与图 5-4，虽然后者不管是模块的个数，还是模块之间的关系，都相对更加简单，然而 Web Component 组件与业务逻辑层之间却是强耦合的，这样的设计不利于应对业务扩展与需求变更。通过引入接口模块 IBLL 与工厂模块 BLLFactory，解除了与具体模块 BLL 的依赖关系。这种设计对于业务逻辑相对比较复杂的系统而言，更符合面向对象的设计思想，有利于我们建立可抽取、可替换的“抽屉”式三层架构。

petshop4.0 详解之六(PetShop 表示层设计)

表示层（Presentation Layer）的设计可以给系统客户最直接的体验和最十足的信心。正如人与人的相交相识一样，初次见面的感觉总是永难忘怀的。一件交付给客户使用的产品，如果在用户界面（User Interface，UI）上缺乏吸引人的特色，界面不友好，操作不够体贴，即使这件产品性能非常优异，架构设计合理，业务逻辑都满足了客户的需求，却仍然难以讨得客户的欢心。

俗语云：“佛要金装，人要衣装”，特别是对于 Web 应用程序而言，Web 网页就好比人的衣装，代表着整个系统的身份与脸面，是招徕“顾客”的最大卖点。

“献丑不如藏拙”，作为艺术细胞缺乏的我，并不打算在用户界面的美术设计上大做文章，是以本书略过不提。本章所关注的表示层设计，还是以架构设计的角度，阐述在表示层设计中对模式的应用，ASP.NET 控件的设计与运用，同时还包括了对 ASP.NET 2.0 新特色的介绍。

6.1 MVC 模式

表示层设计中最重要的模式是 MVC（Model-View-Controller，即模型-视图-控制器）模式。MVC 模式最早是由 SmallTalk 语言研究团提出的，被广泛应用在用户交互应用程序中。Controller 根据用户请求（Response）修改 Model 的属性，此时 Event（事件）被触发，所有依赖于 Model 的 View 对象会自动更新，并基于 Model 对象产生一个响应（Response）信息，返回给 Controller。Martin Fowler 在《企业应用架构模式》一书中，展示了 MVC 模式应用的全过程，如图 6-1 所示：

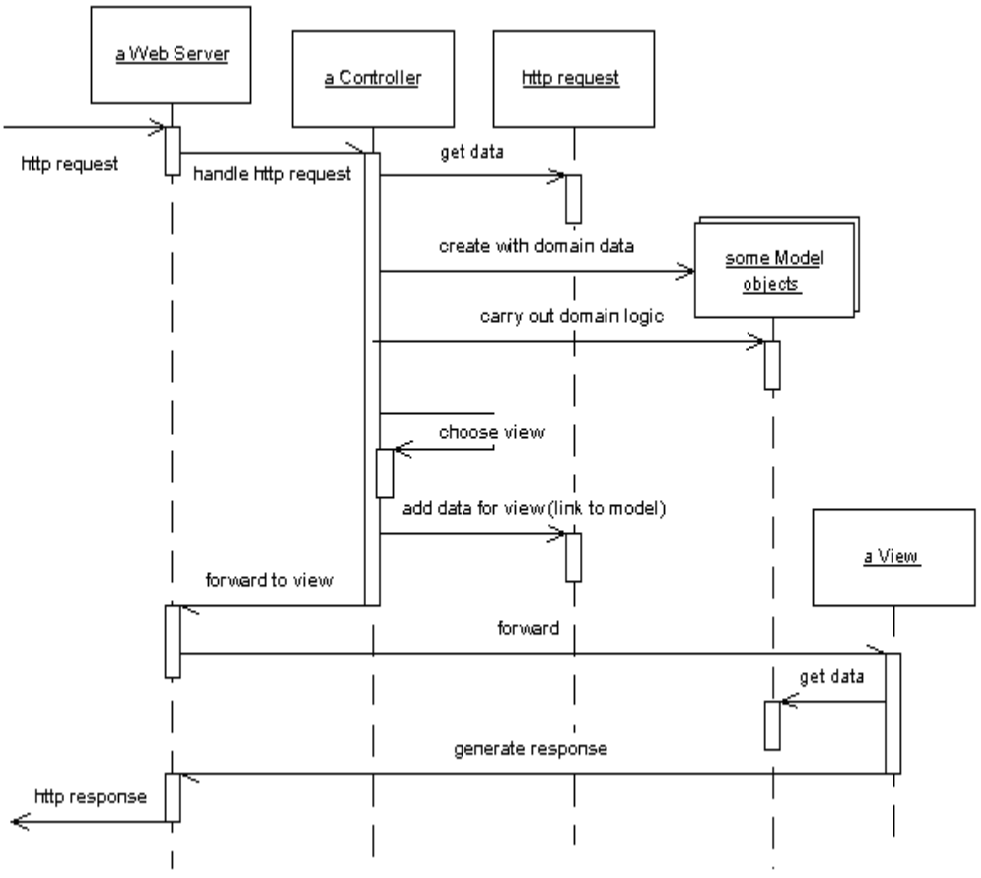


图 6-1 典型的 MVC 模式

如果将 MVC 模式拆解为三个独立的部分：Model、View、Controller，我们可以通过 GOF 设计模式来实现和管理它们之间的关系。在体系架构设计中，业务逻辑层的领域对象以及数据访问层的数据值对象都属于 MVC 模式的 Model 对象。如果要管理 Model 与 View 之间的关系，可以利用 Observer 模式，View 作为观察者，一旦 Model 的属性值发生变化，就会通知 View 基

于 Model 的值进行更新。而 Controller 作为控制用户请求/响应的对象，则可以利用 Mediator 模式，专门负责请求/响应任务之间的调节。而对于 View 本身，在面向组件设计思想的基础上，我们通常将它设计为组件或者控件，这些组件或者控件根据自身特性的不同，共同组成一种类似于递归组合的对象结构，因而我们可以利用 Composite 模式来设计 View 对象。

然而在.NET 平台下，我们并不需要自己去实现 MVC 模式。对于 View 对象而言，ASP.NET 已经提供了常用的 Web 控件，我们也可以通过继承 System.Web.UI.UserControl，自定义用户控件，并利用 ASPX 页面组合 Web 控件来实现视图。ASP.NET 定义了 System.Web.UI.Page 类，它相当于 MVC 模式的 Controller 对象，可以处理用户的请求。由于利用了 codebehind 技术，使得用户界面的显示与 UI 实现逻辑完全分离，也即是说，View 对象与 Controller 对象成为相对独立的两部分，从而有利于代码的重用性。比较 ASP 而言，这种编程方式更符合开发人员的编程习惯，同时有利于开发人员与 UI 设计人员的分工与协作。至于 Model 对象，则为业务逻辑层的领域对象。此外，.NET 平台通过 ADO.NET 提供了 DataSet 对象，便于与 Web 控件的数据源绑定。

6.2 Page Controller 模式的应用

通观 PetShop 的表示层设计，充分利用了 ASP.NET 的技术特点，通过 Web 页面与用户控件控制和展现视图，并利用 codebehind 技术将业务逻辑层的领域对象加入到表示层实现逻辑中，一个典型的 Page Controller 模式呼之欲出。

Page Controller 模式是 Martin Fowler 在《企业应用架构模式》中最重要的表示层模式之一。在.NET 平台下，Page Controller 模式的实现非常简单，以 Products.aspx 页面为例。首先在 aspx 页面中，进行如下的设置：

```
<%@ Page AutoEventWireup="true" Language="C#" MasterPageFile="~/MasterPage.master" Title="Products" Inherits="PetShop.Web.Products" CodeFile="~/Products.aspx.cs" %>
```

Aspx 页面继承自 System.Web.UI.Page 类。Page 类对象通过继承 System.Web.UI.Control 类，从而拥有了 Web 控件的特性，同时它还实现了 IHttpHandler 接口。作为 ASP.NET 处理 HTTP Web 请求的接口，提供了如下的定义：

```
[AspNetHostingPermission(SecurityAction.InheritanceDemand,
Level=AspNetHostingPermissionLevel.Minimal),
AspNetHostingPermission(SecurityAction.LinkDemand,
Level=AspNetHostingPermissionLevel.Minimal)]
public interface IHttpHandler
{
    void ProcessRequest(HttpContext context);
    bool IsReusable { get; }
}
```

Page 类实现了 ProcessRequest()方法，通过它可以设置 Page 对象的 Request 和 Response 属性，从而完成对用户请求/相应的控制。然后 Page 类通过从 Control 类继承来的 Load 事件，将 View 与 Model 建立关联，如 Products.aspx.cs 所示：

```
 public partial class Products：System.Web.UI.Page
{
    protected void Page_Load(object sender, EventArgs e)
    {
        //get page header and title
        Page.Title = WebUtility.GetCategoryName(Request.QueryString["categoryI
d"]);
    }
}
```

事件机制恰好是 observer 模式的实现，当 ASPX 页面的 Load 事件被激发后，系统通过 WebUtility 类（在第 28 章中有对 WebUtility 类的详细介绍）的 GetCategoryName()方法，获得 Category 值，并将其显示在页面的 Title 上。Page 对象作为 Controller，就好似一个调停者，用于协调 View 与 Model 之间的关系。

由于 ASPX 页面中还可以包含 Web 控件，这些控件对象同样是作为 View 对象，通过 Page 类型对象完成对它们的控制。例如在 CheckOut.aspx 页面中，当用户发出 CheckOut 的请求后，作为 System.Web.UI.WebControls.Winzard 控件类型的 wzdCheckOut，会在整个向导过程结束时，触发 FinishButtonClick 事件，并在该事件中调用领域对象 Order 的 Insert()方法，如下所示：

```
 public partial class CheckOut：System.Web.UI.Page…


    protected void wzdCheckOut_FinishButtonClick(object sender, WizardNavigat
ionEventArgs e) {
        if (Profile.ShoppingCart.CartItems.Count > 0) {
            if (Profile.ShoppingCart.Count > 0) {

                // display ordered items
                CartListOrdered.Bind(Profile.ShoppingCart.CartItems);

                // display total and credit card information
                ltlTotalComplete.Text = ltlTotal.Text;
                ltlCreditCardComplete.Text = ltlCreditCard.Text;

                // create order
                OrderInfo order = new OrderInfo(int.MinValue, DateTime.Now, User.
Identity.Name, GetCreditCardInfo(), billingForm.Address, shippingForm.Address,
```

```
Profile.ShoppingCart.Total, Profile.ShoppingCart.GetOrderLineItems(), null);

            // insert
            Order newOrder = new Order();
            newOrder.Insert(order);

            // destroy cart
            Profile.ShoppingCart.Clear();
            Profile.Save();
        }
    }
    else {
        lblMsg.Text = "<p><br>Can not process the order. Your cart is empty.
</p><p class=SignUpLabel><a class=linkNewUser href=Default.aspx>Continue
shopping</a></p>";
        wzdCheckOut.Visible = false;
    }
}
```

在上面的一段代码中，非常典型地表达了 Model 与 View 之间的关系。它通过获取控件的属性值，作为参数值传递给数据值对象 OrderInfo，从而利用页面上产生的订单信息创建订单对象，然后再调用领域对象 Order 的 Inser()方法将 OrderInfo 对象插入到数据表中。此外，它还对领域对象 ShoppingCart 的数据项作出判断，如果其值等于 0，就在页面中显示 UI 提示信息。此时，View 的内容决定了 Model 的值，而 Model 值反过来又决定了 View 的显示内容。

6.3 ASP.NET 控件

ASP.NET 控件是 View 对象最重要的组成部分，它充分利用了面向对象的设计思想，通过封装与继承构建一个个控件对象，使得用户在开发 Web 页面时，能够重用这些控件，甚至自定义自己的控件。在第 8 章中，我已经介绍了.NET Framework 中控件的设计思想，通过引入一种“复合方式”的 Composite 模式实现了控件树。在 ASP.NET 控件中，System.Web.UI.Control 就是这棵控件树的根，它定义了所有 ASP.NET 控件共有的属性、方法和事件，并负责管理和控制控件的整个执行生命周期。

Control 基类并没有包含 UI 的特定功能，如果需要提供与 UI 相关的方法属性，就需要从 System.Web.UI.WebControls.WebControl 类派生。该类实际上也是 Control 类的子类，但它附加了诸如 ForeColor、BackColor、Font 等属性。

除此之外，还有一个重要的类是 System.Web.UI.UserControl，即用户控件类，它同样是 Control 类的子类。我们可以自定义一些用户控件派生自 UserControl，在 Visual Studio 的 Design 环境下，我们可以通过拖动控件的方式将多种类型的控件组合成一个自定义用户控件，也可以在 codebehind 方式下，为自定义用户控件类添加新的属性和方法。

整个 ASP.NET 控件类的层次结构如图 6-2 所示：

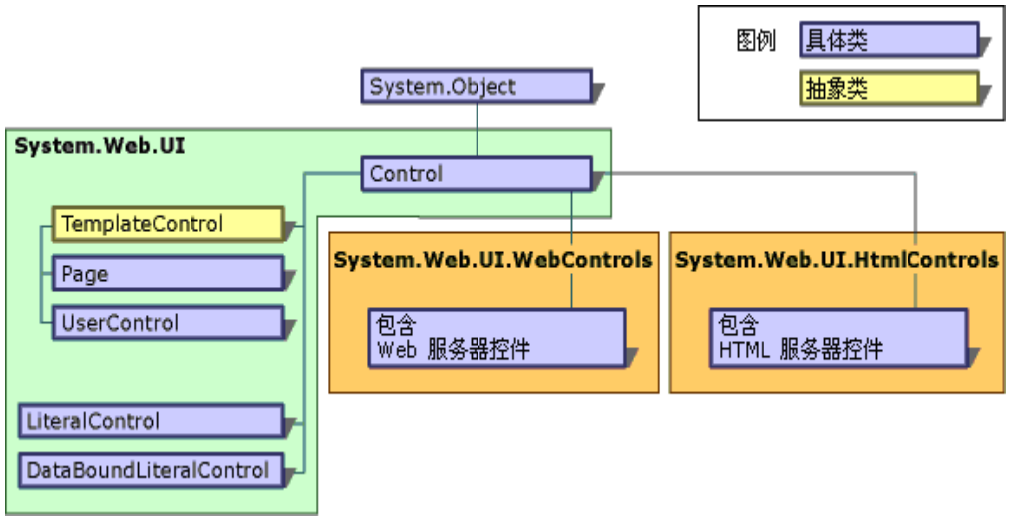


图 6-2 ASP.NET 控件类的层次结构

ASP.NET 控件的执行生命周期如表 6-1 所示：

阶段	控件需要执行的操作	要重写的方法或事件
初始化	初始化在传入 Web 请求生命周期内所需的设置。	Init 事件（OnInit 方法）
加载视图状态	在此阶段结束时，就会自动填充控件的 ViewState 属性，控件可以重写 LoadViewState 方法的默认实现，以自定义状态还原。	LoadViewState 方法
处理回发数据	处理传入窗体数据，并相应地更新属性。 注意：只有处理回发数据的控件参与此阶段。	LoadPostData 方法（如果已实现 IPostBackDataHandler）
加载	执行所有请求共有的操作，如设置数据库查询。此时，树中的服务器控件已创建并初始化、状态已还原并且窗体控件反映了客户端的数据。	Load 事件（OnLoad 方法）
发送回发更改通知	引发更改事件以响应当前和以	RaisePostDataChangedEvent

	前回发之间的状态更改。 注意：只有引发回发更改事件的控件参与此阶段。	方法（如果已实现 IPostBackDataHandler）
处理回发事件	处理引起回发的客户端事件，并在服务器上引发相应的事件。 注意：只有处理回发事件的控件参与此阶段。	RaisePostBackEvent 方法（如果已实现 IPostBackEventHandler）
预呈现	在呈现输出之前执行任何更新。可以保存在预呈现阶段对控件状态所做的更改，而在呈现阶段所对的更改则会丢失。	PreRender 事件（OnPreRender 方法）
保存状态	在此阶段后，自动将控件的 ViewState 属性保持到字符串对象中。此字符串对象被发送到客户端并作为隐藏变量发送回来。为了提高效率，控件可以重写 SaveViewState 方法以修改 ViewState 属性。	SaveViewState 方法
呈现	生成呈现给客户端的输出。	Render 方法
处置	执行销毁控件前的所有最终清理操作。在此阶段必须释放对昂贵资源的引用，如数据库链接。	Dispose 方法
卸载	执行销毁控件前的所有最终清理操作。控件作者通常在 Dispose 中执行清除，而不处理此事件。	UnLoad 事件（On UnLoad 方法）

表 6-1 ASP.NET 控件的执行生命周期

在这里，控件设计利用了 Template Method 模式，Control 基类提供了大部分 protected 虚方法，留待其子类改写其方法。以 PetShop 4.0 为例，就定义了两个 ASP.NET 控件，它们都属于 System.Web.UI.WebControls.WebControl 的子类。其中，CustomList 控件派生自 System.Web.UI.WebControls.DataList，CustomGrid 控件则派生自 System.Web.UI.WebControls.Repeater。

由于这两个控件都改变了其父类控件的呈现方式，故而，我们可以通过重写父类的 Render 虚方法，完成控件的自定义。例如 CustomGrid 控件：

```
public class CustomGrid : Repeater...
//Static constants
    protected const string HTML1 = "<table cellpadding=0
cellspacing=0><tr><td colspan=2>";
    protected const string HTML2 = "</td></tr><tr><td class=paging align=left
>";
    protected const string HTML3 = "</td><td align=right class=paging>";
    protected const string HTML4 = "</td></tr></table>";
    private static readonly Regex RX = new Regex(@"^&page=\d+",
RegexOptions.Compiled);
    private const string LINK_PREV = "<a href=?page={0}>< Previous</
a>";
    private const string LINK_MORE = "<a href=?page={0}>More ></a>";
private const string KEY_PAGE = "page";
    private const string COMMA = "?";
    private const string AMP = "&";

override protected void Render(HtmlTextWriter writer) {

      //Check there is some data attached
        if (ItemCount == 0) {
          writer.Write(emptyText);
          return;
      }
      //Mask the query
      string query = Context.Request.Url.Query.Replace(COMMA, AMP);
      query = RX.Replace(query, string.Empty);
      // Write out the first part of the control, the table header
      writer.Write(HTML1);
      // Call the inherited method
      base.Render(writer);
      // Write out a table row closure
      writer.Write(HTML2);
      //Determin whether next and previous buttons are required
      //Previous button?
      if (currentPageIndex > 0)
          writer.Write(string.Format(LINK_PREV, (currentPageIndex - 1) + query
));
      //Close the table data tag
      writer.Write(HTML3);

      //Next button?
      if (currentPageIndex < PageCount)
          writer.Write(string.Format(LINK_MORE, (currentPageIndex + 1) + quer
```

```
y));

        //Close the table
        writer.Write(HTML4);
    }
```

由于 CustomGrid 继承自 Repeater 控件，因而它同时还继承了 Repeater 的 DataSource 属性，这是一个虚属性，它默认的 set 访问器属性如下：

```
  public virtual object DataSource
{
    get  {... }
    set
    {
        if (((value != null) && !(value is IListSource)) && !(value is IEnumerabl
e))
        {

            throw new ArgumentException(SR.GetString("Invalid_DataSource_
Type", new object[] { this.ID }));
        }
        this.dataSource = value;
        this.OnDataPropertyChanged();
    }
}
```

对于 CustomGrid 而言，DataSource 属性有着不同的设置行为，因而在定义 CustomGrid 控件的时候，需要改写 DataSource 虚属性，如下所示：

```
  private IList dataSource;
  private int itemCount;

override public object DataSource {
    set {
    //This try catch block is to avoid issues with the VS.NET designer
        //The designer will try and bind a datasource which does not derive from I
LIST
        try {
         dataSource = (IList)value;
         ItemCount = dataSource.Count;
        }
        catch {
         dataSource = null;
         ItemCount = 0;
        }
```

```
    }
}
```

当设置的 value 对象值不为 IList 类型时，set 访问器就将捕获异常，然后将 dataSource 字段设置为 null。

由于我们改写了 DataSource 属性，因而改写 Repeater 类的 OnDataBinding()方法也就势在必行。此外，CustomGrid 还提供了分页的功能，我们也需要实现分页的相关操作。与 DataSource 属性不同，Repeater 类的 OnDataBinding()方法实际上是继承和改写了 Control 基类的 OnDataBinding()虚方法，而我们又在此基础上改写了 Repeater 类的 OnDataBinding()方法：

```
override protected void OnDataBinding(EventArgs e) {

    //Work out which items we want to render to the page
    int start = CurrentPageIndex * pageSize;
    int size = Math.Min(pageSize, ItemCount - start);

    IList page = new ArrayList();
    //Add the relevant items from the datasource
    for (int i = 0; i < size; i++)
        page.Add(dataSource[start + i]);

    //set the base objects datasource
    base.DataSource = page;
    base.OnDataBinding(e);
}
```

此外，CustomGrid 控件类还增加了许多属于自己的属性和方法，例如 PageSize、PageCount 属性以及 SetPage()方法等。正是因为 ASP.NET 控件引入了 Composite 模式与 Template Method 模式，当我们在自定义控件时，就可以通过继承与改写的方式来完成控件的设计。自定义 ASP.NET 控件一方面可以根据系统的需求实现特定的功能，也能够最大限度地实现对象的重用，既可以减少编码量，同时也有利于未来对程序的扩展与修改。
在 PetShop 4.0 中，除了自定义了上述 WebControl 控件的子控件外，最主要的还是利用了用户控件。在 Controls 文件夹下，一共定义了 11 个用户控件，内容涵盖客户地址信息、信用卡信息、购物车信息、期望列表（Wish List）信息以及导航信息、搜索结果信息等。它们相当于是一些组合控件，除了包含了子控件的方法和属性外，也定义了一些必要的 UI 实现逻辑。以 ShoppingCartControl 用户控件为例，它会在该控件被呈现（Render）之前，做一些数据准备工作，获取购物车数据，并作为数据源绑定到其下的 Repeater 控件：

```
public partial class ShoppingCartControl : System.Web.UI.UserControl ...

    protected void Page_PreRender(object sender, EventArgs e) {
        if (!IsPostBack) {
```

```
            BindCart();
        }
    }
    private void BindCart() {

        ICollection<CartItemInfo> cart = Profile.ShoppingCart.CartItems;
        if (cart.Count > 0) {
            repShoppingCart.DataSource = cart;
            repShoppingCart.DataBind();
            PrintTotal();
            plhTotal.Visible = true;
        }
        else {
            repShoppingCart.Visible = false;
            plhTotal.Visible = false;
            lblMsg.Text = "Your cart is empty.";
        }
    }
```

在 ShoppingCart 页面下，我们可以加入该用户控件，如下所示：

```
<PetShopControl:shoppingcartcontrol id="ShoppingCartControl1" runat="server"
></PetShopControl:shoppingcartcontrol>
```

由于 ShoppingCartControl 用户控件已经实现了用于呈现购物车数据的逻辑，那么在 ShoppingCart.aspx.cs 中，就可以不用负责这些逻辑，在充分完成对象重用的过程中，同时又达到了职责分离的目的。用户控件的设计者与页面设计者可以互不干扰，分头完成自己的设计。特别是对于页面设计者而言，他可以是单一的 UI 设计人员角色，仅需要关注用户界面是否美观与友好，对于表示层中对领域对象的调用与操作就可以不必理会，整个页面的代码也显得结构清晰、逻辑清楚，无疑也“干净”了不少。

petshop4.0 详解之七(PetShop 表示层设计)

6.4 ASP.NET 2.0 新特性

由于 PetShop 4.0 是基于.NET Framework 2.0 平台开发的电子商务系统，因而它在表示层也引入了许多 ASP.NET 2.0 的新特性，例如 MemberShip、Profile、Master Page、登录控件等特性。接下来，我将结合 PetShop 4.0 的设计分别介绍它们的实现。

6.4.1 Profile 特性

Profile 提供的功能是针对用户的个性化服务。在 ASP.NET 1.x 版本时，我们可以利用 Session、Cookie 等方法来存储用户的状态信息。然而 Session 对象是具有生存期的，一旦生存期结束，

该对象保留的值就会失效。Cookie 将用户信息保存在客户端，它具有一定的安全隐患，一些重要的信息不能存储在 Cookie 中。一旦客户端禁止使用 Cookie，则该功能就将失去应用的作用。

Profile 的出现解决了如上的烦恼，它可以将用户的个人化信息保存在指定的数据库中。ASP.NET 2.0 的 Profile 功能默认支持 Access 数据库和 SQL Server 数据库，如果需要支持其他数据库，可以编写相关的 ProfileProvider 类。Profile 对象是强类型的，我们可以为用户信息建立属性，以 PetShop 4.0 为例，它建立了 ShoppingCart、WishList 和 AccountInfo 属性。

由于 Profile 功能需要访问数据库，因而在数据访问层（DAL）定义了和 Product 等数据表相似的模块结构。首先定义了一个 IProfileDAL 接口模块，包含了接口 IPetShopProfileProvider：

```
public interface IPetShopProfileProvider
{
    AddressInfo GetAccountInfo(string userName, string appName);
    void SetAccountInfo(int uniqueID, AddressInfo addressInfo);
    IList<CartItemInfo> GetCartItems(string userName, string appName,
bool isShoppingCart);
    void SetCartItems(int uniqueID, ICollection<CartItemInfo> cartItems,
bool isShoppingCart);
    void UpdateActivityDates(string userName, bool activityOnly, string appName);
    int GetUniqueID(string userName, bool isAuthenticated, bool ignoreAuthenticati
onType,
    string appName);
    int CreateProfileForUser(string userName, bool isAuthenticated, string appNam
e);
    IList<string> GetInactiveProfiles(int authenticationOption,
DateTime userInactiveSinceDate, string appName);
    bool DeleteProfile(string userName, string appName);
    IList<CustomProfileInfo> GetProfileInfo(int authenticationOption,
string usernameToMatch, DateTime userInactiveSinceDate, string appName,
out int totalRecords);
}
```

因为 PetShop 4.0 版本分别支持 SQL Server 和 Oracle 数据库，因而它分别定义了两个不同的 PetShopProfileProvider 类，实现 IPetShopProfileProvider 接口，并放在两个不同的模块 SQLProfileDAL 和 OracleProfileDAL 中。具体的实现请参见 PetShop 4.0 的源代码。
同样的，PetShop 4.0 为 Profile 引入了工厂模式，定义了模块 ProfileDALFActory，工厂类 DataAccess 的定义如下：

```
public sealed class DataAccess {

    private static readonly string profilePath = ConfigurationManager.AppSettings
["ProfileDAL"];

```

```
    public static PetShop.IProfileDAL.IPetShopProfileProvider CreatePetShopProfil
eProvider() {
  string className = profilePath + ".PetShopProfileProvider";
  return (PetShop.IProfileDAL.IPetShopProfileProvider)Assembly.Load(profilePath
).CreateInstance(className);
    }
}
```

在业务逻辑层（BLL）中，单独定义了模块 Profile，它添加了对 BLL、IProfileDAL 和 ProfileDALFactory 模块的程序集。在该模块中，定义了密封类 PetShopProfileProvider，它继承自 System.Web.Profile.ProfileProvider 类，该类作为 Profile 的 Provider 基类，用于在自定义配置文件中实现相关的配置文件服务。在 PetShopProfileProvider 类中，重写了父类 ProfileProvider 中的一些方法，例如 Initialize()、GetPropertyValues()、SetPropertyValues()、DeleteProfiles()等方法。此外，还为 ShoppingCart、WishList、AccountInfo 属性提供了 Get 和 Set 方法。至于 Provider 的具体实现，则调用工厂类 DataAccess 创建的具体类型对象，如下所示：

```
private static readonly IPetShopProfileProvider dal =
DataAccess.CreatePetShopProfileProvider();
```

定义了 PetShop.Profile.PetShopProfileProvider 类后，才可以在 web.config 配置文件中配置如下的配置节：

```
  <profile automaticSaveEnabled="false" defaultProvider="ShoppingCartProvider">
   <providers>
    <add name="ShoppingCartProvider" connectionStringName="SQLProfileConnS
tring" type="PetShop.Profile.PetShopProfileProvider" applicationName=".NET Pet
Shop 4.0"/>
    <add name="WishListProvider" connectionStringName="SQLProfileConnString
" type="PetShop.Profile.PetShopProfileProvider" applicationName=".NET Pet Shop
 4.0"/>
    <add name="AccountInfoProvider" connectionStringName="SQLProfileConnStr
ing" type="PetShop.Profile.PetShopProfileProvider" applicationName=".NET Pet S
hop 4.0"/>
   </providers>
   <properties>
    <add name="ShoppingCart" type="PetShop.BLL.Cart" allowAnonymous="true
" provider="ShoppingCartProvider"/>
    <add name="WishList" type="PetShop.BLL.Cart" allowAnonymous="true" prov
ider="WishListProvider"/>
    <add name="AccountInfo" type="PetShop.Model.AddressInfo" allowAnonymou
s="false" provider="AccountInfoProvider"/>
   </properties>
  </profile>
```

在配置文件中，针对 ShoppingCart、WishList 和 AccountInfo（它们的类型分别为 PetShop.BLL.Cart、PetShop.BLL.Cart、PetShop.Model.AddressInfo）属性分别定义了 ShoppingCartProvider、WishListProvider、AccountInfoProvider，它们的类型均为 PetShop.Profile.PetShopProfileProvider 类型。至于 Profile 的信息究竟是存储在何种类型的数据库中，则由以下的配置节决定：

```
<add key="ProfileDAL" value="PetShop.SQLProfileDAL"/>
```

而键值为 ProfileDAL 的值，正是 Profile 的工厂类 PetShop.ProfileDALFactory.DataAccess 在利用反射技术创建 IPetShopProfileProvider 类型对象时获取的。

在表示层中，可以利用页面的 Profile 属性访问用户的个性化属性，例如在 ShoppingCart 页面的 codebehind 代码 ShoppingCart.aspx.cs 中，调用 Profile 的 ShoppingCart 属性：

```
public partial class ShoppingCart : System.Web.UI.Page {

    protected void Page_PreInit(object sender, EventArgs e) {
        if (!IsPostBack) {
            string itemId = Request.QueryString["addItem"];
            if (!string.IsNullOrEmpty(itemId)) {
                Profile.ShoppingCart.Add(itemId);
                Profile.Save();
                // Redirect to prevent duplictations in the cart if user hits "Refresh"
                Response.Redirect("~/ShoppingCart.aspx", true);
            }
        }
    }
}
```

在上述的代码中，Profile 属性的值从何而来？实际上，在我们为 web.config 配置文件中对 Profile 进行配置后，启动 Web 应用程序，ASP.NET 会根据该配置文件中的相关配置创建一个 ProfileCommon 类的实例。该类继承自 System.Web.Profile.ProfileBase 类。然后调用从父类继承来的 GetPropertyValue 和 SetPropertyValue 方法，检索和设置配置文件的属性值。然后，ASP.NET 将创建好的 ProfileCommon 实例设置为页面的 Profile 属性值。因而，我们可以通过智能感知获取 Profile 的 ShoppingCart 属性，同时也可以利用 ProfileCommon 继承自 ProfileBase 类的 Save()方法，根据属性值更新 Profile 的数据源。

6.4.2 Membership 特性

PetShop 4.0 并没有利用 Membership 的高级功能，而是直接让 Membership 特性和 ASP.NET 2.0 新增的登录控件进行绑定。由于.NET Framework 2.0 已经定义了针对 SQL Server 的 SqlMembershipProvider，因此对于 PetShop 4.0 而言，实现 Membership 比之实现 Profile 要简单，仅仅需要为 Oracle 数据库定义 MembershipProvider 即可。在 PetShop.Membership 模块中，定义了 OracleMembershipProvider 类，它继承自 System.Web.Security.MembershipProvider 抽象类。

OracleMembershipProvider 类的实现具有极高的参考价值，如果我们需要定义自己的 MembershipProvider 类，可以参考该类的实现。
事实上 OracleMemberShip 类的实现并不复杂，在该类中，主要是针对用户及用户安全而实现相关的行为。由于在父类 MembershipProvider 中，已经定义了相关操作的虚方法，因此我们需要作的是重写这些虚方法。由于与 Membership 有关的信息都是存储在数据库中，因而 OracleMembershipProvider 与 SqlMembershipProvider 类的主要区别还是在于对数据库的访问。对于 SQL Server 而言，我们利用 aspnet_regsql 工具为 Membership 建立了相关的数据表以及存储过程。也许是因为知识产权的原因，Microsoft 并没有为 Oracle 数据库提供类似的工具，因而需要我们自己去创建 membership 的数据表。此外，由于没有创建 Oracle 数据库的存储过程，因而 OracleMembershipProvider 类中的实现是直接调用 SQL 语句。以 CreateUser()方法为例，剔除那些繁杂的参数判断与安全性判断，SqlMembershipProvider 类的实现如下：

```
public override MembershipUser CreateUser(string username, string password, str
ing email, string passwordQuestion, string passwordAnswer, bool isApproved, obj
ect providerUserKey, out MembershipCreateStatus status)
{
    MembershipUser user1;
    //前面的代码略;
    try
    {
        SqlConnectionHolder holder1 = null;
        try
        {
            holder1 = SqlConnectionHelper.GetConnection(this._sqlConnection
String, true);
            this.CheckSchemaVersion(holder1.Connection);
            DateTime time1 = this.RoundToSeconds(DateTime.UtcNow);
            SqlCommand command1 = new SqlCommand("dbo.aspnet_Membe
rship_CreateUser", holder1.Connection);
            command1.CommandTimeout = this.CommandTimeout;
            command1.CommandType = CommandType.StoredProcedure;
            command1.Parameters.Add(this.CreateInputParam("@ApplicationN
ame", SqlDbType.NVarChar, this.ApplicationName));
            command1.Parameters.Add(this.CreateInputParam("@UserName",
SqlDbType.NVarChar, username));
            command1.Parameters.Add(this.CreateInputParam("@Password", S
qlDbType.NVarChar, text2));
            command1.Parameters.Add(this.CreateInputParam("@PasswordSal
t", SqlDbType.NVarChar, text1));
            command1.Parameters.Add(this.CreateInputParam("@Email", SqlD
bType.NVarChar, email));
            command1.Parameters.Add(this.CreateInputParam("@PasswordQu
```

```
estion", SqlDbType.NVarChar, passwordQuestion));
            command1.Parameters.Add(this.CreateInputParam("@PasswordAns
wer", SqlDbType.NVarChar, text3));
            command1.Parameters.Add(this.CreateInputParam("@IsApproved",
 SqlDbType.Bit, isApproved));
            command1.Parameters.Add(this.CreateInputParam("@UniqueEmail
", SqlDbType.Int, this.RequiresUniqueEmail ? 1 : 0));
            command1.Parameters.Add(this.CreateInputParam("@PasswordFor
mat", SqlDbType.Int, (int) this.PasswordFormat));
            command1.Parameters.Add(this.CreateInputParam("@CurrentTime
Utc", SqlDbType.DateTime, time1));
            SqlParameter parameter1 = this.CreateInputParam("@UserId", Sql
DbType.UniqueIdentifier, providerUserKey);
            parameter1.Direction = ParameterDirection.InputOutput;
            command1.Parameters.Add(parameter1);
            parameter1 = new SqlParameter("@ReturnValue", SqlDbType.Int);
            parameter1.Direction = ParameterDirection.ReturnValue;
            command1.Parameters.Add(parameter1);
            command1.ExecuteNonQuery();
            int num3 = (parameter1.Value != null) ? ((int) parameter1.Value) :
 -1;
            if ((num3 < 0) || (num3 > 11))
            {
                num3 = 11;
            }
            status = (MembershipCreateStatus) num3;
            if (num3 != 0)
            {
                return null;
            }
            providerUserKey = new Guid(command1.Parameters["@UserId"].V
alue.ToString());
            time1 = time1.ToLocalTime();
            user1 = new MembershipUser(this.Name, username, providerUserK
ey, email, passwordQuestion, null, isApproved, false, time1, time1, time1, time1,
new DateTime(0x6da, 1, 1));
        }
        finally
        {
            if (holder1 != null)
            {
                holder1.Close();
                holder1 = null;
            }
```

```
        }
    }
    catch
    {
        throw;
    }
    return user1;
}
```

代码中，aspnet_Membership_CreateUser 为 aspnet_regsql 工具为 membership 创建的存储过程，它的功能就是创建一个用户。

OracleMembershipProvider 类中对 CreateUser()方法的定义如下：

```
public override MembershipUser CreateUser(string username, string password, str
ing email, string passwordQuestion, string passwordAnswer, bool isApproved, obj
ect userId, out MembershipCreateStatus status) {
    //前面的代码略;
  //Create connection
  OracleConnection connection = new OracleConnection(OracleHelper.Connection
StringMembership);
  connection.Open();
  OracleTransaction transaction = connection.BeginTransaction(IsolationLevel.Re
adCommitted);
  try {
   DateTime dt = DateTime.Now;
   bool isUserNew = true;

   // Step 1: Check if the user exists in the Users table: create if not
   int uid = GetUserID(transaction, applicationId, username, true, false, dt, out is
UserNew);
   if(uid == 0) { // User not created successfully!
    status = MembershipCreateStatus.ProviderError;
    return null;
  }
   // Step 2: Check if the user exists in the Membership table: Error if yes.
   if(IsUserInMembership(transaction, uid)) {
    status = MembershipCreateStatus.DuplicateUserName;
    return null;
  }
   // Step 3: Check if Email is duplicate
   if(IsEmailInMembership(transaction, email, applicationId)) {
    status = MembershipCreateStatus.DuplicateEmail;
    return null;
```

```
    }
    // Step 4: Create user in Membership table
    int pFormat = (int)passwordFormat;

    if(!InsertUser(transaction, uid, email, pass, pFormat, salt, "", "", isApproved, d
t)) {
      status = MembershipCreateStatus.ProviderError;
      return null;
    }
    // Step 5: Update activity date if user is not new
    if(!isUserNew) {
      if(!UpdateLastActivityDate(transaction, uid, dt)) {
        status = MembershipCreateStatus.ProviderError;
        return null;
      }
    }
    status = MembershipCreateStatus.Success;
    return new MembershipUser(this.Name, username, uid, email, passwordQuesti
on, null, isApproved, false, dt, dt, dt, dt, DateTime.MinValue);
  }
  catch(Exception) {
    if(status == MembershipCreateStatus.Success)
      status = MembershipCreateStatus.ProviderError;
    throw;
  }
  finally {
    if(status == MembershipCreateStatus.Success)
      transaction.Commit();
    else
      transaction.Rollback();
    connection.Close();
    connection.Dispose();
  }
}
```

petshop4.0 详解之八(PetShop 表示层设计)

代码中，InsertUser()方法就是负责用户的创建，而在之前则需要判断创建的用户是否已经存在。InsertUser()方法的定义如下：

```
private static bool InsertUser(OracleTransaction transaction, int userId, string em
ail, string password, int passFormat, string passSalt, string passQuestion, string p
assAnswer, bool isApproved, DateTime dt) {

```

```
 string insert = "Insert INTO MEMBERSHIP (USERID, EMAIL, PASSWORD, PASSWORDFORMAT, PASSWORDSALT, PASSWORDQUESTION, PASSWORDANSWER, ISAPPROVED, CreateDDATE, LASTLOGINDATE, LASTPASSWORDCHANGEDDATE) VALUES (:UserID, :Email, :Pass, :PasswordFormat, :PasswordSalt, :PasswordQuestion, :PasswordAnswer, :IsApproved, :CDate, :LLDate, :LPCDate)";

 OracleParameter[] insertParms = { new OracleParameter(":UserID", OracleType.Number, 10), new OracleParameter(":Email", OracleType.VarChar, 128), new OracleParameter(":Pass", OracleType.VarChar, 128), new OracleParameter(":PasswordFormat", OracleType.Number, 10), new OracleParameter(":PasswordSalt", OracleType.VarChar, 128), new OracleParameter(":PasswordQuestion", OracleType.VarChar, 256), new OracleParameter(":PasswordAnswer", OracleType.VarChar, 128), new OracleParameter(":IsApproved", OracleType.VarChar, 1), new OracleParameter(":CDate", OracleType.DateTime), new OracleParameter(":LLDate", OracleType.DateTime), new OracleParameter(":LPCDate", OracleType.DateTime) };
 insertParms[0].Value = userId;
 insertParms[1].Value = email;
 insertParms[2].Value = password;
 insertParms[3].Value = passFormat;
 insertParms[4].Value = passSalt;
 insertParms[5].Value = passQuestion;
 insertParms[6].Value = passAnswer;
 insertParms[7].Value = OracleHelper.OraBit(isApproved);
 insertParms[8].Value = dt;
 insertParms[9].Value = dt;
 insertParms[10].Value = dt;

 if(OracleHelper.ExecuteNonQuery(transaction, CommandType.Text, insert, insertParms) != 1)
  return false;
 else
  return true;
}
```

在为 Membership 建立了 Provider 类后，还需要在配置文件中配置相关的配置节，例如 SqlMembershipProvider 的配置：

```
 <membership defaultProvider="SQLMembershipProvider">
  <providers>
   <add name="SQLMembershipProvider" type="System.Web.Security.SqlMembershipProvider" connectionStringName="SQLMembershipConnString" applicationName=".NET Pet Shop 4.0" enablePasswordRetrieval="false" enablePasswordReset="true" requiresQuestionAndAnswer="false" requiresUniqueEmail="false" passwordFormat="Hashed"/>
```

```
  </providers>
 </membership>
```

对于 OracleMembershipProvider 而言，配置大致相似：

```
<membership defaultProvider="OracleMembershipProvider">
 <providers>
  <clear/>
  <add name="OracleMembershipProvider"
   type="PetShop.Membership.OracleMembershipProvider"
   connectionStringName="OraMembershipConnString"
   enablePasswordRetrieval="false"
   enablePasswordReset="false"
   requiresUniqueEmail="false"
   requiresQuestionAndAnswer="false"
   minRequiredPasswordLength="7"
   minRequiredNonalphanumericCharacters="1"
   applicationName=".NET Pet Shop 4.0"
   hashAlgorithmType="SHA1"
   passwordFormat="Hashed"/>
 </providers>
</membership>
```

有关配置节属性的意义，可以参考 MSDN 等相关文档。

6.4.3 ASP.NET 登录控件

这里所谓的登录控件并不是指一个控件，而是 ASP.NET 2.0 新提供的一组用于解决用户登录的控件。登录控件与 Membership 进行集成，快速简便地实现用户登录的处理。ASP.NET 登录控件包括 Login 控件、LoginView 控件、LoginStatus 控件、LoginName 控件、PasswordRescovery 控件、CreateUserWizard 控件以及 ChangePassword 控件。PetShop 4.0 犹如一本展示登录控件用法的完美教程。我们可以从诸如 SignIn、NewUser 等页面中，看到 ASP.NET 登录控件的使用方法。例如在 SignIn.aspx 中，用到了 Login 控件。在该控件中，可以包含 TextBox、Button 等类型的控件，用法如下所示：

```
<asp:Login ID="Login" runat="server" CreateUserUrl="~/NewUser.aspx" SkinID
="Login" FailureText="Login failed. Please try again.">
 </asp:Login>
```

又例如 NewUser.aspx 中对 CreateUserWizard 控件的使用：

```
<asp:CreateUserWizard ID="CreateUserWizard" runat="server" CreateUserButto
nText="Sign Up" InvalidPasswordErrorMessage="Please enter a more secure pass
```

```
word." PasswordRegularExpressionErrorMessage="Please enter a more secure pa
ssword."
  RequireEmail="False" SkinID="NewUser">
  <WizardSteps>
          <asp:CreateUserWizardStep ID="CreateUserWizardStep1" runat="serv
er">
    </asp:CreateUserWizardStp>
   </WizardSteps>
  </asp:CreateUserWizard>
```

使用了登录控件后，我们毋需编写与用户登录相关的代码，登录控件已经为我们完成了相关的功能，这就大大地简化了这个系统的设计与实现。

6.4.4 Master Page 特性

Master Page 相当于是整个 Web 站点的统一模板，建立的 Master Page 文件扩展名为.master。它可以包含静态文本、html 元素和服务器控件。Master Page 由特殊的@Master 指令识别，如：

```
<%@ Master Language="C#" CodeFile="MasterPage.master.cs" Inherits="Master
Page" %>
```

使用 Master Page 可以为网站建立一个统一的样式，且能够利用它方便地创建一组控件和代码，然后将其应用于一组页。对于那些样式与功能相似的页而言，利用 Master Page 就可以集中处理为 Master Page，一旦进行修改，就可以在一个位置上进行更新。

在 PetShop 4.0 中，建立了名为 MasterPage.master 的 Master Page，它包含了 header、LoginView 控件、导航菜单以及用于呈现内容的 html 元素，如图 6-3 所示：

图 6-3 PetShop 4.0 的 Master Page

@Master 指令的定义如下：

```
<%@ Master Language="C#" AutoEventWireup="true" CodeFile="MasterPage.master.cs" Inherits="PetShop.Web.MasterPage" %>
```

Master Page 同样利用 codebehind 技术，以 PetShop 4.0 的 Master Page 为例，codebehind 的代码放在文件 MasterPage.master.cs 中：

```
public partial class MasterPage : System.Web.UI.MasterPage {

    private const string HEADER_PREFIX = ".NET Pet Shop :: {0}";

    protected void Page_PreRender(object sender, EventArgs e) {
        ltlHeader.Text = Page.Header.Title;
        Page.Header.Title = string.Format(HEADER_PREFIX, Page.Header.Title);

    }
    protected void btnSearch_Click(object sender, EventArgs e) {
        WebUtility.SearchRedirect(txtSearch.Text);
    }
}
```

注意 Master Page 页面不再继承自 System.Web.UI.Page，而是继承 System.Web.UI.MasterPage 类。与 Page 类继承 TemplateControl 类不同，它是 UserControl 类的子类。因此，可以应用在 Master Page 上的有效指令与 UserControl 的可用指令相同，例如 AutoEventWireup、ClassName、CodeFile、EnableViewState、WarningLevel 等。

每一个与 Master Page 相关的内容页必须在@Page 指令的 MasterPageFile 属性中引用相关的 Master Page。例如 PetShop 4.0 中的 CheckOut 内容页，其@Page 指令的定义如下：

```
<%@ Page Language="C#" MasterPageFile="~/MasterPage.master" AutoEventWireup="true" CodeFile="CheckOut.aspx.cs" Inherits="PetShop.Web.CheckOut" Title="Check Out" %>
```

Master Page 可以进行嵌套，例如我们建立了父 Master Page 页面 Parent.master，那么在子 Master Page 中，可以利用 master 属性指定其父 MasterPage：
<%@ Master Language="C#" master="Parent.master"%>

而内容页则可以根据情况指向 Parent.master 或者 Child.master 页面。

虽然说 Master Page 大部分情况下是以声明方式创建，但我们也可以建立一个类继承 System.Web.UI.MasterPage，从而完成对 Master Page 的编程式创建。但在采用这种方式的同时，应该同时创建.master 文件。此外对 Master Page 的调用也可以利用编程的方式完成，例如动态地添加 Master Page，我们重写内容页的 Page_PreInit()方法，如下所示：

```
void Page_PreInit(Object sender, EventArgs e)
{
    this.MasterPageFile = "~/NewMaster.master";
}
```

之所以重写 Page_PreInit()方法，是因为 Master Page 会在内容页初始化阶段进行合并，也即是说是在 PreInit 阶段完成 Master Page 的分配。
ASP.NET 2.0 引入的新特性，并不仅仅限于上述介绍的内容。例如 Theme、Wizard 控件等新特性在 PetShop 4.0 中也得到了大量的应用。虽然 ASP.NET 2.0 及时地推陈出新，对表示层的设计有所改善，然而作为 ASP.NET 2.0 的其中一部分，它们仅仅是对现有框架缺失的弥补与改进，属于“锦上添花”的范畴，对于整个表示层设计技术而言，起到的推动作用却非常有限。

直到 AJAX（Asynchronous JavaScript and XML）的出现，整个局面才大为改观。虽然 AJAX 技术带有几分“旧瓶装新酒”的味道，然而它从诞生之初，就具备了王者气象，大有席卷天下之势。各种支持 AJAX 技术的框架如雨后春笋般纷纷吐出新芽，支撑起百花齐放的繁荣，气势汹汹地营造出唯 AJAX 独尊的态势。如今，AJAX 已经成为了 Web 应用的主流开发技术，许多业界大鳄都呲牙咧嘴开始了对这一块新领地的抢滩登陆。例如 IBM、Oracle、Yahoo 等公司都纷纷启动了开源的 AJAX 项目。微软也不甘落后，及时地推出了 ASP.NET AJAX，这是一个基于 ASP.NET 的 AJAX 框架，它包括了 ASP.NET AJAX 服务端组件和 ASP.NET AJAX 客户端组件，并集成在 Visual Studio 中，为 ASP.NET 开发者提供了一个强大的 AJAX 应用环境。

我现在还无法预知 AJAX 技术在未来的走向，然而单单从表示层设计的角度而言，AJAX 技术亦然带了一场全新的革命。我们或者可以期待未来的 PetShop 5.0，可以在表示层设计上带来更多的惊喜。