

用经典案例学通.NET技术

ASP.NET企业级开发案例精解（C#编程篇）

刘庆国 聂元铭 编著

人民邮电出版社

图书在版编目（CIP）数据

ASP. NET 企业级开发案例精解. C#编程篇 / 刘庆国, 聂元铭编著.

—北京：人民邮电出版社，2006.2

ISBN 7-115-13707-2

I . A... II . ①刘... ②聂... III. ①主页制作—程序设计 ②C 语言—程序设计

IV. ①TP393.092 ②TP312

中国版本图书馆 CIP 数据核字（2006）第 008919 号

内 容 提 要

ASP.NET Starter Kit 是微软公司提供的免费下载 ASP.NET 入门指南，其中包括学习资料和示例解决方案。ASP.NET Starter Kit 中的企业级解决方案功能实用、代码编写规范，是学习和进行 ASP.NET 开发可借鉴的理想范例。但是它又非常复杂，在没有指导的情况下，初学者很难将其读懂并且应用到实际当中。本书从应用的角度出发，按照功能模块对 ASP.NET Starter Kit 中的简单电子商务系统和项目进度管理系统做了详尽的解析，并讲述了其中的设计思想和开发技巧。本书的范例采用的编程语言是 C#。

本书从实用的角度出发，结合 ASP.NET Starter Kit 中范例讲解 ASP.NET 技术，适合正在从事和希望学习 ASP.NET 开发的人员阅读。

用经典案例学通.NET 技术

ASP.NET 企业级开发案例精解（C#编程篇）

-
- ◆ 编 著 刘庆国 聂元铭
 - 责任编辑 屈艳莲
 - ◆ 人民邮电出版社出版发行 北京市崇文区夕照寺街 14 号
邮编 100061 电子函件 315@ptpress.com.cn
网址 <http://www.ptpress.com.cn>
 - 北京密云春雷印刷厂印刷
 - 新华书店总店北京发行所经销
 - ◆ 开本：787×1092 1/16
 - 印张：22.25
 - 字数：670 千字 2006 年 2 月第 1 版
 - 印数：1—5 000 册 2006 年 2 月北京第 1 次印刷

ISBN 7-115-13707-2/TP • 4827

定价：38.00 元

读者服务热线：(010) 67132692 印装质量热线：(010) 67129223



前言

ASP.NET Starter Kit 是一组开放源代码的 ASP.NET Web 应用程序, ASP.NET Starter Kit 的主要目的是为 ASP.NET 的初学者提供一个学习 ASP.NET 开发的示例性“教科书”。无论学习哪一门开发语言, 最有效的方式就是在学习过简单的语法之后, 研读示例程序。只有通过对实际的示例研究和分析, 才能学到实用的开发技巧。ASP.NET 的初学者很幸运有 ASP.NET Starter Kit 这样一组优秀的示例作为学习 ASP.NET 的突破点, ASP.NET Starter Kit 的每一个示例无论在结构设计还是在代码实现方面都堪称经典, 它有灵活的设计模式、严谨的编程风格、精练的代码语句, 值得初学者反复研究。

本书将以 ASP.NET Starter Kit 中的两个示例应用程序“Commerce Starter Kit——电子商务系统”和“Time Tracker Starter Kit——项目与工作管理系统”为基础, 讲解 ASP.NET 开发的技术要点、开发思路以及编程技巧。

Commerce Starter Kit 是一个电子商务应用程序, 实现了电子商务应用程序中关键的部分, 对准备使用 ASP.NET 开发电子商务应用程序的开发者有很大的帮助。

Time Tracker Starter Kit 是一个项目管理和工作进程管理的示例程序, 该示例以企业或组织结构内部的项目和工作管理为主要应用背景, 对准备使用 ASP.NET 开发企业信息化的开发者有很大帮助。

编写本书的目的是希望读者既学习到两个示例中具体的开发技巧, 也学习到 ASP.NET 的开发中的主要技术点以及针对不同项目的设计理念, 希望读者通过对本书中两个示例的分析和研究, 形成一套属于自己的 ASP.NET 开发方法。

本书的主要作者是刘庆国。同时, 要感谢为本书付出辛勤劳动的技术人员。首先感谢参与本书校对工作的刘哲, 他认真细致的修改保证了本书的质量。其次要感谢对本书部分章节提出过意见的张英男。还要感谢我的同事刘英鹏、谢丽玲、陈建新等人给予我的支持。最后要特别感谢我的合作伙伴聂元铭。

由于时间仓促加之水平有限, 书中错漏之处在所难免, 敬请读者批评指正。本书作者的联系方法是 wfconquer@163.com, 欢迎来信交流。读者也可以通过发送 E-mail 至 quyanlian2@ptpress.com.cn 与本书责任编辑交流。

刘庆国

2006 年 2 月



目 录

| | |
|-----------------------------------|----|
| 第一部分 Commerce Starter Kit——电子商务系统 | |
| 第 1 章 Commerce 简介 | 3 |
| 1.1 Commerce 的概况 | 3 |
| 1.1.1 Commerce 的开发背景 | 3 |
| 1.1.2 Commerce 的版本 | 4 |
| 1.2 Commerce 的安装 | 5 |
| 1.2.1 下载安装 Commerce | 5 |
| 1.2.2 配置运行 Commerce | 6 |
| 1.3 开发启示 | 6 |
| 1.3.1 Commerce 技术要点概述 | 6 |
| 1.3.2 Commerce 开发启示 | 6 |
| 第 2 章 需求分析与系统架构 | 8 |
| 2.1 系统功能 | 8 |
| 2.1.1 需求分析 | 8 |
| 2.1.2 系统功能 | 10 |
| 2.2 UML 图 | 11 |
| 2.2.1 用例图 | 11 |
| 2.2.2 类图 | 11 |
| 2.2.3 活动图 | 13 |
| 2.3 系统架构 | 13 |
| 2.4 数据库设计 | 14 |
| 2.5 设计启示 | 14 |
| 第 3 章 技术点解析 | 16 |
| 3.1 服务器控件 | 16 |
| 3.1.1 服务器控件的特点 | 16 |
| 3.1.2 服务器控件的简单应用 | 17 |
| 3.1.3 ASP.NET 可利用的服务器控件 | 18 |
| 3.2 多层结构 | 18 |
| 3.2.1 多层结构的概念 | 18 |
| 3.2.2 ASP.NET 中的多层结构 | 19 |
| 3.3 配置 Web.config | 20 |
| 3.3.1 Web.config 的功能和特点 | 20 |
| 3.3.2 配置 Web.config | 21 |
| 3.4 Web Service | 24 |
| 3.4.1 Web Service 简述 | 24 |
| 3.4.2 构建 ASP.NET 的 Web Service | 25 |
| 3.4.3 在 ASP.NET 中调用 Web Service | 27 |
| 3.5 技术点总结 | 28 |
| 第 4 章 系统设计 | 29 |
| 4.1 页面样式基础 | 29 |

| | |
|--------------------------------|-----------|
| 4.1.1 使用级联样式表文件..... | 29 |
| 4.1.2 ASPNETCommerce.css | 30 |
| 4.2 Global.asax | 31 |
| 4.2.1 Global.asax 的结构和作用..... | 31 |
| 4.2.2 使用 Global.asax..... | 32 |
| 4.3 用户控件 | 33 |
| 4.3.1 _Header.ascx | 33 |
| 4.3.2 _Menu.ascx | 35 |
| 4.3.3 在页面中调用用户控件..... | 38 |
| 4.4 小结 | 38 |
| 第5章 用户体系..... | 39 |
| 5.1 系统设计 | 39 |
| 5.1.1 需求分析..... | 39 |
| 5.1.2 功能设计..... | 40 |
| 5.1.3 数据库设计..... | 41 |
| 5.2 用户注册 | 41 |
| 5.2.1 实现效果..... | 41 |
| 5.2.2 关键技术点..... | 43 |
| 5.2.3 存储过程..... | 43 |
| 5.2.4 数据访问层..... | 44 |
| 5.2.5 业务逻辑层..... | 45 |
| 5.2.6 用户表示层..... | 48 |
| 5.3 用户登录 | 49 |
| 5.3.1 实现效果..... | 49 |
| 5.3.2 关键技术点..... | 50 |
| 5.3.3 存储过程..... | 50 |
| 5.3.4 数据访问层..... | 51 |
| 5.3.5 业务逻辑层..... | 53 |
| 5.3.6 用户表示层..... | 56 |
| 5.4 开发启示 | 56 |
| 第6章 Web商店..... | 58 |
| 6.1 系统设计 | 58 |
| 6.1.1 需求分析..... | 58 |
| 6.1.2 功能设计..... | 59 |
| 6.1.3 数据库设计..... | 60 |
| 6.2 关键技术点 | 62 |
| 6.2.1 ASP.NET 中的数据访问技术 | 62 |
| 6.2.2 ASP.NET 中的数据展现技术 | 67 |
| 6.3 商品浏览 | 69 |
| 6.3.1 实现效果..... | 69 |
| 6.3.2 存储过程..... | 70 |
| 6.3.3 数据访问层..... | 73 |
| 6.3.4 业务逻辑层..... | 77 |

| | |
|----------------------------------|------------|
| 6.3.5 用户表示层..... | 80 |
| 6.4 购物车..... | 83 |
| 6.4.1 实现效果..... | 83 |
| 6.4.2 存储过程..... | 84 |
| 6.4.3 数据访问层..... | 89 |
| 6.4.4 业务逻辑层..... | 96 |
| 6.4.5 用户表示层..... | 102 |
| 6.5 商品评论..... | 103 |
| 6.5.1 实现效果..... | 103 |
| 6.5.2 存储过程..... | 104 |
| 6.5.3 数据访问层..... | 105 |
| 6.5.4 业务逻辑层..... | 107 |
| 6.5.5 用户表示层..... | 109 |
| 6.6 开发启示..... | 110 |
| 第 7 章 购物订单..... | 112 |
| 7.1 系统设计..... | 112 |
| 7.1.1 需求分析..... | 112 |
| 7.1.2 功能设计..... | 113 |
| 7.1.3 数据库设计..... | 113 |
| 7.2 购物订单..... | 113 |
| 7.2.1 实现效果..... | 113 |
| 7.2.2 关键技术点..... | 114 |
| 7.2.3 存储过程..... | 116 |
| 7.2.4 数据访问层..... | 118 |
| 7.2.5 业务逻辑层..... | 122 |
| 7.2.6 用户表示层..... | 123 |
| 7.3 开发启示..... | 124 |
| 第 8 章 构建 Web Service..... | 125 |
| 8.1 系统设计..... | 125 |
| 8.1.1 需求分析..... | 125 |
| 8.1.2 功能设计..... | 126 |
| 8.2 实现 Web Service..... | 127 |
| 8.2.1 实现效果..... | 127 |
| 8.2.2 实现细节..... | 130 |
| 8.2.3 第三方调用 Web Service..... | 132 |
| 8.3 开发启示..... | 133 |
| 第 9 章 扩展 Commerce..... | 134 |
| 9.1 订单管理..... | 134 |
| 9.1.1 需求分析..... | 134 |
| 9.1.2 数据层..... | 136 |
| 9.1.3 逻辑层..... | 140 |
| 9.1.4 表示层..... | 142 |
| 9.2 用户数据..... | 147 |

| | |
|-------------------------|-----|
| 9.2.1 需求分析 | 147 |
| 9.2.2 数据层 | 149 |
| 9.2.3 逻辑层 | 150 |
| 9.2.4 表示层 | 151 |
| 9.3 扩展 Web Service | 156 |
| 9.3.1 需求分析 | 157 |
| 9.3.2 数据层 | 157 |
| 9.3.3 逻辑层 | 159 |
| 9.4 第三方调用 | 164 |
| 9.4.1 需求分析 | 164 |
| 9.4.2 开发 InvokeCommerce | 164 |
| 9.5 开发启示 | 178 |

第二部分 Time Tracker Starter Kit——项目与工作管理系统

| | |
|----------------------------|-----|
| 第 10 章 TimeTracker 简介 | 181 |
| 10.1 TimeTracker 概况 | 181 |
| 10.1.1 TimeTracker 的开发背景 | 181 |
| 10.1.2 系统功能 | 182 |
| 10.1.3 版本信息 | 182 |
| 10.2 TimeTracker 的安装 | 183 |
| 10.2.1 推荐环境 | 183 |
| 10.2.2 安装步骤 | 183 |
| 10.2.3 配置说明 | 188 |
| 10.3 技术点概述 | 189 |
| 第 11 章 需求分析与系统架构 | 190 |
| 11.1 系统功能 | 190 |
| 11.1.1 需求分析 | 190 |
| 11.1.2 功能设计 | 191 |
| 11.2 UML 图 | 192 |
| 11.2.1 类图 | 192 |
| 11.2.2 活动图 | 194 |
| 11.3 系统架构 | 194 |
| 11.4 数据库设计 | 194 |
| 11.5 设计启示 | 196 |
| 第 12 章 技术点解析 | 197 |
| 12.1 DataGridView 的高级应用 | 197 |
| 12.1.1 DataGridView 的事件 | 197 |
| 12.1.2 常用的 DataGridView 事件 | 198 |
| 12.2 对象集合类的使用 | 200 |
| 12.3 GDI+技术 | 202 |
| 12.3.1 GDI+简介 | 202 |
| 12.3.2 GDI+技术 | 202 |
| 12.3.3 GDI+技术简单应用 | 203 |

| | |
|------------------------------------|------------|
| 12.4 移动 Web 应用 | 205 |
| 12.4.1 移动 Web 应用基础知识..... | 205 |
| 12.4.2 ASP.NET 移动 Web 应用程序..... | 205 |
| 12.5 技术点总结 | 205 |
| 第 13 章 系统底层开发..... | 207 |
| 13.1 页面样式 | 207 |
| 13.2 Web.config | 207 |
| 13.2.1 系统级常量的定义..... | 207 |
| 13.2.2 用户身份验证..... | 208 |
| 13.3 Global.asax | 210 |
| 13.4 用户控件 Banner | 211 |
| 13.4.1 逻辑层 | 212 |
| 13.4.2 后台编码类..... | 212 |
| 13.4.3 前台页面..... | 214 |
| 13.5 用户控件 AdminTabs | 214 |
| 13.5.1 后台编码类..... | 215 |
| 13.5.2 前台页面..... | 216 |
| 13.6 开发启示 | 216 |
| 第 14 章 用户体系..... | 217 |
| 14.1 系统设计 | 217 |
| 14.1.1 功能设计..... | 217 |
| 14.1.2 数据结构..... | 219 |
| 14.2 关键技术点 | 220 |
| 14.2.1 ASP.NET 运行时用户信息的存储..... | 220 |
| 14.2.2 集合类的使用..... | 220 |
| 14.3 用户体系的实现 | 220 |
| 14.3.1 数据库设计 | 220 |
| 14.3.2 逻辑层 | 224 |
| 14.3.3 应用层 | 238 |
| 14.3.4 表示层 | 247 |
| 14.4 开发启示 | 249 |
| 第 15 章 项目管理与工作进程管理 | 250 |
| 15.1 系统设计 | 250 |
| 15.1.1 功能设计..... | 250 |
| 15.1.2 数据结构..... | 252 |
| 15.2 关键技术点 | 253 |
| 15.2.1 DataGrid 的高级应用 | 253 |
| 15.2.2 使用 ViewState 存储页面内部变量 | 254 |
| 15.3 项目管理 | 254 |
| 15.3.1 数据库设计 | 254 |
| 15.3.2 逻辑层 | 265 |
| 15.3.3 应用层 | 277 |
| 15.3.4 表示层 | 288 |

| | |
|-------------------------------|------------|
| 15.4 工作进程管理 | 290 |
| 15.4.1 数据库设计 | 290 |
| 15.4.2 逻辑层 | 292 |
| 15.4.3 应用层 | 295 |
| 15.4.4 表示层 | 301 |
| 15.5 开发启示 | 303 |
| 第 16 章 数据报表 | 304 |
| 16.1 系统设计 | 304 |
| 16.1.1 功能设计 | 304 |
| 16.1.2 数据库设计 | 306 |
| 16.2 页面形式的报表 | 309 |
| 16.2.1 逻辑层 | 309 |
| 16.2.2 应用层 | 314 |
| 16.2.3 表示层 | 318 |
| 16.3 图片形式的报表 | 324 |
| 16.3.1 应用层 | 324 |
| 16.3.2 逻辑层 | 325 |
| 16.4 开发启示 | 331 |
| 第 17 章 移动 Web 应用 | 332 |
| 17.1 系统设计 | 332 |
| 17.2 应用详解 | 333 |
| 17.2.1 用户登录 | 333 |
| 17.2.2 管理工作进程 | 336 |
| 17.3 开发启示 | 345 |

第一部分

Commerce Starter Kit——电子商务系统

案例简介

Commerce Starter Kit 是一个简单的电子商务应用程序，其中包括了电子商务应用程序中最主要的模块。作为一个教程式的应用程序范例，Commerce Starter Kit 展示了电子商务应用程序的基本设计思路和实现方法，开发者不仅可以将其作为一个学习的示例，也可以将其作为电子商务应用程序的开发基础，根据实际情况对其进行扩展和补充。Commerce Starter Kit 中的技术点相对来讲难度较低，但是其设计思路和代码风格对初学者很具有指导意义。

知识点提要

Commerce Starter Kit 的重点知识点如下。

- 服务器控件
- 多层结构
- Web.Config 的配置
- Web Service

资源下载

读者可到微软 ASP.NET 官方网站下载源代码，地址是：

<http://www.asp.net/StarterKits/DownloadCommerce.aspx?tabindex=0&tabid=1>



第 1 章

Commerce 简介

1.1 Commerce 的概况

ASP.NET Commerce Starter Kit 是 ASP.NET Starter Kit 中提供的一套虚拟网络商店模型 Web 应用程序，浏览者和系统用户可以浏览包括商品名称、图片、简介和价格在内的商品信息，因为这是一套虚拟的示例应用程序，所以这些被展示的商品和商品的信息都是虚构的。浏览者和系统用户可以根据需要把想要购买的商品放入购物车，从而完成一笔订单。用户还可以对自己感兴趣但又暂时不想购买的商品进行收藏和跟踪。

1.1.1 Commerce 的开发背景

ASP.NET Commerce Starter Kit 所提供的虚拟网络商店模型是一个展示 ASP.NET 在电子商务应用程序开发方面的模型与示例。在实际使用当中还要添加一些功能，比如说商品的添加、订单支付方面的管理等。运行界面如图 1-1 所示。

以下是对 ASP.NET Commerce Starter Kit 所涵盖功能的简单概述。

- 商品浏览。未注册的浏览者和系统用户都可以浏览商品的列表，用户可以通过商品的分类和快速搜索两种方式检索到商品。用户可以通过商品列表进入商品详细信息页面，商品的详细信息包括商品名称、商品价格、商品详细说明、商品型号、商品分类、商品的图片以及购买此商品的用户通常还购买过什么商品这些信息。由于 ASP.NET Commerce Starter Kit 只作为一个示例程序，因此并没有商品信息管理的功能模块。
- 购物车。当系统的用户决定购买某个商品的时候，就会把该商品放入自己的购物车里面，这个过程和现实购物中的程序是一样的。当用户完成一次购物的时候，会把目前购

物车内的所有商品作为这条订单的商品进行结账。当然，在用户结账之前可以对购物车内的商品进行任意的修改。

- 用户信息。普通的浏览者是不能通过本系统订购商品的，因为系统并不能得到普通浏览者的相关信息。只有系统的注册用户在登录之后才能订购商品，普通的浏览者可以通过系统提供的注册程序成为系统用户。由于 ASP.NET Commerce Starter Kit 只作为一个示例程序，因此并没有提供相应的用户管理的功能模块。

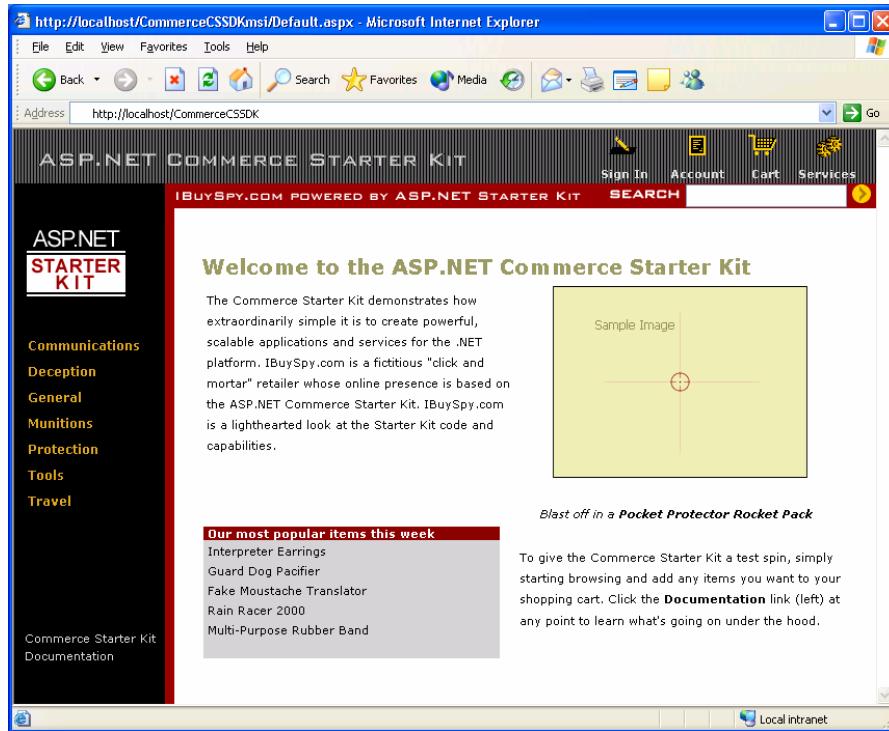


图 1-1 ASP.NET Commerce Starter Kit 运行界面

- 订单管理。ASP.NET Commerce Starter Kit 提供的订单管理的功能主要是用户可以对目前的订单和历史订单进行查看，包括每一条订单的总体情况和详细信息。但是 ASP.NET Commerce Starter Kit 并没有提供订单处理的功能模块。
- Web Service。为了实现系统和第三方应用程序的通信，ASP.NET Commerce Starter Kit 提供了一个 Web Service，该 Web Service 提供订购一笔订单和查看订单状态两个方法。通过这两个方法，第三方应用程序可以很方便地实现与系统的通信。

由于 ASP.NET Commerce Starter Kit 仅仅是提供了一个简单的电子商务模型，设计者在构建应用程序的时候充分考虑到基于现有的程序进行其他功能模块和现有模块的二次开发，在系统的设计上留下了可供扩展的接口，从而使得以 ASP.NET Commerce Starter Kit 为基础迅速开发出一套完整强大的电子商务应用程序变得相对简单。

1.1.2 Commerce 的版本

ASP.NET Commerce Starter Kit 一共有 6 个版本，如表 1-1 所示。

| | | |
|---|-------|-------------|
| 4 | 第 1 章 | Commerce 简介 |
|---|-------|-------------|

表 1-1

ASP.NET Commerce Starter Kit 的不同版本信息

| 版 本 号 | 使 用 语 言 | 推 荐 Web 服 务 器 | 备 注 |
|-------------------------------|---------|---------------|-------------------------|
| Commerce Starter Kit (VB SDK) | VB.NET | Web Matrix | |
| Commerce Starter Kit (CS SDK) | C# | Web Matrix | |
| Commerce Starter Kit (JS SDK) | J# | Web Matrix | 需要.NET Framework 1.1 支持 |
| Commerce Starter Kit (VB VS) | VB.NET | IIS 6.0 | |
| Commerce Starter Kit (CS VS) | C# | IIS 6.0 | |
| Commerce Starter Kit (JS VS) | J# | IIS 6.0 | 需要.NET Framework 1.1 支持 |

本书讲解的是 Commerce Starter Kit (CS VS) 版本。后面章节提到 ASP.NET Commerce Starter Kit 如不做特殊说明，均为该版本。

1.2 Commerce 的安装

1.2.1 下载安装 Commerce

登录 <http://www.asp.net/Default.aspx?tabindex=8&tabid=47>，这里是 ASP.NET Starter Kit 的主页。在这里可以免费下载到 ASP.NET Commerce Starter Kit 的最新版本。

安装 ASP.NET Commerce Starter Kit 推荐使用如下环境：

- .NET Framework 1.1；
- Internet 信息服务器 (Internet Information Server, IIS)；
- Microsoft SQL Server 2000 或 Microsoft SQL Server Desktop Engine 2000。

注意：以上列出的只是推荐使用的开发环境，并不是必需的开发环境。比如对 Web 服务器有特殊要求的话，可以使用 Cassini，因为目前最新版本的 Commerce Starter Kit 已经可以在 Cassini 下运行。

以下是 ASP.NET Commerce Starter Kit 具体的安装步骤。

- (1) 运行安装文件。打开“欢迎安装 ASP.NET Commerce Starter Kit”的页面。
- (2) 单击 Next 按钮，显示出安装说明和许可协议。然后选择是否将程序运行在 IIS 下，如果是，直接单击 Next 按钮。接下来选择应用程序安装在本机还是一个远程主机，如果选择安装在远程主机，那么必须有远程主机的管理员权限。
- (3) 选择程序的安装路径和程序的运行用户身份。在选择程序运行身份时，可选择本应用程序仅供安装用户运行，还可以选择当前服务器上所有用户都具有运行权限。
- (4) 程序开始自动进行安装，在应用程序安装完毕之后，会自动开始数据库的安装。在选择安装目标数据库之后必须测试数据库连接，如图 1-2 所示，测试成功之后才能进行数据库的安装。
- (5) 当数据库安装完毕之后，就会出现 ASP.NET Commerce Starter Kit 安装成功的画面。

以上讲述的是利用 ASP.NET Commerce Starter Kit 的安装程序进行安装，也可以通过手工部署应用程序和导入数据库的方式安装 ASP.NET Commerce Starter Kit。在使用安装程序安装成功之后，在 ASP.NET Commerce Starter Kit 的安装目录下的 Setup 目录中提供了导入数据库的 SQL 脚本。在 SQL Server 的查询分析器中依次运行 Commerce_CreateDB.sql、Commerce_GrantPermissions_ForLocal.sql（如果选择数据库服务器在远程主机，那么此处应运行 Commerce_GrantPermissions_ForRemote.sql）、Commerce_LoadData.sql。



图 1-2 测试数据库连接页面

1.2.2 配置运行 Commerce

安装结束之后，ASP.NET Commerce Starter Kit 就可以正常地运行了。如果没有选择使用安装程序安装或是更改目前应用程序的运行环境，那么就需要更改 ASP.NET Commerce Starter Kit 的配置文件。

更改数据库连接应配置 ASP.NET Commerce Starter Kit 的 Web.config 文件，查找名为 ConnectionString 的配置字符串。假如数据库服务器名称为 CommerceServer，数据库名称为 Commerce，数据库的登录名为 sa，密码为空，那么更改 Web.config 文件中的 ConnectionString 为如下格式：

```
<configuration>
<appSettings>
<add key="ConnectionString" value="server=CommerceServer;database=Commerce;uid=sa;pwd=;" />
.....
<appSettings>
.....
<configuration>
```

1.3 开发启示

1.3.1 Commerce 技术要点概述

ASP.NET Commerce Starter Kit 主要的技术要点如下：

- 服务器控件的使用，前台页面的高度 HTML 化；
- 使用页面高速缓存来提高性能；
- ADO.NET 三层架构的使用；
- 基于表单的用户身份认证；
- Web Service 的应用。

以后的章节将对这些技术要点在 ASP.NET Commerce Starter Kit 中的应用进行详细讲解。

1.3.2 Commerce 开发启示

ASP.NET Commerce Starter Kit 是一个简单的电子商务 Web 应用程序，其中包括商品浏览、购物车

和订单结算等电子商务应用程序中必不可少的模块。ASP.NET Commerce Starter Kit 的开发目的有以下两个。

(1) 为学习 ASP.NET 的初学者提供一个教程式的经典案例

学习一门语言最快和最有效的途径莫过于对经典的开发案例进行反复的研究和揣摸, ASP.NET Starter Kit 的目的正是为 ASP.NET 的初学者提供一组教程式的案例, 其中每一个案例都有具体的应用背景, 而每一个案例给学习者的启示都是不同的。ASP.NET Commerce Starter Kit 作为 ASP.NET Starter Kit 中的一个案例, 其技术难度并不是很高, 设计框架也不是很复杂, 尤其适用于 ASP.NET 的初学者的入门。在技术方面, ASP.NET Commerce Starter Kit 的设计者力求使用最简单实用的技术实现电子商务 Web 应用程序的设计, 这种技术的实现思路和电子商务应用的背景有很大关系, 因为在多数电子商务应用程序中, 性能和程序的稳定性是被人们首要考虑的, 繁琐的技术实现手段无论对于应用程序的性能还是稳定性都是有百害而无一利。在设计方面, ASP.NET Commerce Starter Kit 本着简洁通用的特点, 对多数电子商务应用中的对象和模块进行了抽取和提炼, 将其应用到 ASP.NET Commerce Starter Kit 的设计当中去。

(2) 为使用 ASP.NET 开发电子商务 Web 应用程序提供一个基础框架

ASP.NET Commerce Starter Kit 实际上只实现了电子商务应用程序中最关键的一部分模块, 而诸如商品管理、订单管理、配送管理以及财务统计等方面模块, ASP.NET Commerce Starter Kit 并没有涉及。而在现有的应用程序中, 商品对象以及用户对象也仅仅实现了该对象最基本的属性和方法。是 ASP.NET Commerce Starter Kit 的设计者粗心大意导致的么? 实际上并不是这样。ASP.NET Commerce Starter Kit 的设计者的另外一个目的就是将 ASP.NET Commerce Starter Kit 设计成为一个电子商务 Web 应用程序的基础框架, 其他的开发者可以基于这个框架, 根据实际情况开发出属于可以被应用的电子商务 Web 应用程序。正是基于这个目的, ASP.NET Commerce Starter Kit 的设计者才没有实现电子商务应用中全部的模块, 而 ASP.NET Commerce Starter Kit 中的主要对象也只具有基本的属性。这样可以留给开发者更广阔的空间, 以 ASP.NET Commerce Starter Kit 为基础开发出功能完备、特点鲜明的电子商务应用程序。

对于学习一门新的程序设计语言来说, 往往最需要学习的是如何实现常见的功能模块, 例如在 ASP.NET 的学习中页面元素以及列表的展现, 数据库的连接与操作, 用户机制的实现等都是初学者所关心的, 另外在新的程序设计语言中, 新的概念以及思想也是初学者所关心的。在 ASP.NET Commerce Starter Kit 中, 初学者可以找到这些问题的答案, 对于 ASP.NET Web 应用程序中页面元素以及列表的展现, 数据库的连接与操作, 用户机制的实现等关键问题, ASP.NET Commerce Starter Kit 都有涉及, 并且其实现方法和思路都十分经典。另外 ASP.NET Commerce Starter Kit 中使用了 Web Service 技术, 为初学者提供了一个学习 Web Service 技术基础的范例。

初学者通过对 ASP.NET Commerce Starter Kit 的学习, 不仅可以掌握 ASP.NET 开发的具体方法, 也可以学习到如何利用面向对象思想分析和抽取应用程序中的对象, 以及根据对象模型设计类库。在实际的应用程序开发中, 开发者对于系统中对象的把握和设计的类库, 犹如万丈高楼的根基直接决定系统的成功与否, 因此在面向对象设计中科学的方法论是非常重要的。希望读者能够通过对本书的阅读以及对 ASP.NET Commerce Starter Kit 的学习真正做到举一反三, 在其他 Web 应用程序的开发过程中应用科学的面向对象设计方法。

第 2 章

需求分析 与系统架构

ASP.NET Commerce Starter Kit 的设计者从根本上是想把项目设计成为一个模型，所以在系统设计方面，考虑的更多是通用性和可扩展性。通过对该项目的学习和研究，开发者能够深刻地理解 ASP.NET 的开发模式和开发技巧，同时 ASP.NET Commerce Starter Kit 也可以作为一个电子商务应用的基础模型。

型进行二次开发和扩展。本章中将从需求分析和系统架构方面分析 ASP.NET Commerce Starter Kit。

2.1 系统功能

2.1.1 需求分析

软件工程中包含需求、设计、编码和测试 4 个阶段，其中需求分析是软件工程第一个也是很重要的一个阶段，需求分析的主要任务是绘制关联图、创建开发原型、分析可行性、确定需求优先级、为需求建立模型、编写数据字典、应用质量功能调配。需求分析从总体上看是说明项目应该具有什么样的功能，而不考虑实现这些功能的具体技术。

ASP.NET Commerce Starter Kit 是一个电子商务应用站点，要想提出站点的需求，就需要从站点的目标和用户入手。在实际项目的开发中，需求分析这一阶段是面向商务应用的，往往是客户提出的，但是在 ASP.NET Commerce Starter Kit 的设计中则采用了虚拟需求的模式，因为并没有明确的项目应用背景，所以需求分析都是设计者根据大量的电子商务模型提取和抽象出来的。这些被提取出来的需求具有很强的通用性，电子商务必不可少的功能和多数电子商务都需要具备的一些基本功能成为 ASP.NET Commerce Starter Kit 的需求，例如用户管理和生成订单这样的功能模块。

除了上述实际的应用需求之外，还要考虑到项目开发和项目设计中的需求，比如开发人员的协作、应用程序的整体性能、系统的可扩展性和可维护性等。特别是 ASP.NET Commerce Starter Kit 提供的是一个可以供二次开发的应用模型，所以对系统的通用性和可扩展性要求就更高了。

系统的需求分为物理需求、结构需求、逻辑需求。

物理需求的任务很明确，就是确定 Web 站点的物理服务器的最终架构和软硬件环境。例如应用程序是否需要分布式部署，数据库服务器和 Web 服务器是否必须集成在同一台服务器上，是否允许第三方应用程序进行远程调用等。根据 ASP.NET Commerce Starter Kit 设计者的初衷，物理需求应包括如下几方面。

(1) 支持可分布式部署的服务器群组

支持分布式的服务器群组是优秀的网络应用程序必须提供的一个物理功能，因为大型的网络应用程序不可能将所有的应用和操作运行于同一台服务器。支持分布式的服务器群组有利于降低服务器负荷，使服务器的功能更加具有针对性。

例如可以将数据库服务器和 Web 服务器脱离开，这样不仅能够提高系统的性能，而且也便于管理。

(2) 支持 .NET 的服务器操作平台

这是一个必须要满足的需求。ASP.NET 应用程序不可能脱离 .NET Framework 的支持，因此 Web 服务器必须支持 .NET。

(3) 支持第三方应用程序的远程调用

作为系统扩展的接口，支持第三方应用程序的远程调用是必须的。第三方应用程序可以通过远程调用的形式完成和 ASP.NET Commerce Starter Kit 的交互，无形间也将 ASP.NET Commerce Starter Kit 的应用渗透到第三方应用程序中。

(4) 仅限于 Microsoft SQL Server 的数据库管理系统

支持多种数据库类型是一个不错的构想，但是 ASP.NET Commerce Starter Kit 更多地展示的是 ASP.NET 以及 ADO.NET 中数据操作的新特性，而在 ADO.NET 中针对于 Microsoft SQL Server 提供了很多具体的对象和方法。为了更多地介绍和展现 ADO.NET 中的对象和方法，ASP.NET Commerce Starter Kit 采用了 Microsoft SQL Server 作为系统的数据库管理系统 (DBMS)。

ASP.NET Commerce Starter Kit 的结构需求也是从设计者的初衷中抽象出来的，根据系统的目标和针对性，可以确定如下的结构需求。

(1) 站点的可维护性和可扩展性强

大多数的 Web 应用程序在实际应用中都需要不断地添加功能模块，ASP.NET Commerce Starter Kit 也是一样，在二次开发和实际应用中要根据项目的具体情况添加一些功能模块。因此项目在设计之初就要考虑到，当前的架构对系统的扩展工作会不会形成障碍。

使用 ASP.NET 中层的设计概念能够增强站点的维护性和扩展性，基于层的设计模式允许开发者以三层甚至多层的模式开发 ASP.NET 应用程序，将数据操作、业务逻辑、前台显示等单元分离开，每一层都有针对性，层是以一组序列分布在系统数据和用户之间的，不相邻的层在业务上没有耦合，每一层都是继承和调用上一层中的对象和方法。这种模式使得站点的功能分布更加合理化。例如扩展一部分业务逻辑，首先是要在业务逻辑层中建立相应的方法，然后才是在前台显示层中建立新的页面控件。关于层的详细应用，第 3 章将做详细的讲解。

(2) 站点的功能模块通用性强

由于 ASP.NET Commerce Starter Kit 是作为一个示例和应用程序框架被设计和开发的，因此其功能模块要具有较高的通用性，以保证功能模块中的业务逻辑、数据模型从多数的电子商务应用程序中提取。简单地说，ASP.NET Commerce Starter Kit 需要提供电子商务中最基本的对象和这些对象最基本的属性，只有这样才能使基于 ASP.NET Commerce Starter Kit 的二次开发具有更大的扩展性。例如商品信息只列出最基本的商品信息，至于一些具体应用中商品的特殊属性，并不应该出现在 ASP.NET Commerce Starter Kit 中。

模块化的构建同时也意味着模块之间尽量降低耦合度，这样做的好处是使得更改模块内部或新增其他模块对系统的稳定性影响不大。

物理需求和结构需求都是从设计框架上对系统提出了宏观的要求，而逻辑需求则是在实际项目的需求分析阶段对系统的业务逻辑提出的要求。以下是 ASP.NET Commerce Starter Kit 的逻辑需求。

(1) 完整但不需要完善的购物流程

在 ASP.NET Commerce Starter Kit 中需要提供一个完整的购物流程，但是这个购物流程不一定完善。因为 ASP.NET Commerce Starter Kit 并不是以应用为目的，而是为应用提供模型化的框架。因此在 ASP.NET Commerce Starter Kit 中，用户能够通过浏览商品、放入购物车、完成订单一系列连续的功能完成一个完整的购物流程，但是对于商品的后台管理、订单的后台管理等功能并不是必需的，这些功能是二次开发人员在将系统完善的时候需要考虑的。

(2) 以点盖面的电子商务推广思路

电子商务要求经营者和管理者在理念上创新，不能把商品和销售信息局限于简单的页面展示，经营者应当千方百计地考虑怎么样提高更多的商品浏览次数，这样才有机会将更多的商品销售出去。

系统中应该提供几个至少是一个这样的电子商务推广手段，通过这种创新的电子商务推广思路拓宽电子商务的策略。

(3) 抽象电子商务中的对象及方法

电子商务中有几个关键的对象，每个对象有其特有的方法。对象通过方法实现其自己的功能。例如可以将电子商务中的商品抽象成为一个对象，商品的一些属性如商品的名称、价格、说明等都可以构建成为对象的属性，而商品的信息展示、添加商品这些行为则可以抽象成为商品的方法。

面向对象的应用程序设计中最重要的一条原则就是：一切皆为对象。应用程序是由对象为单元组建完成的，应用程序的逻辑是通过对象提供的方法实现的。因此将电子商务中实际的对象抽象成为虚拟的对象是十分重要的。

(4) 在逻辑上与其他应用程序整合

建立电子商务的站点的目的是客户可以通过网络购买商品。但是电子商务应用并不是孤立存在的。电子商务提供商企业内部的其他应用、电子商务提供商的合作伙伴都是需要和电子商务应用进行交互的。例如电子商务提供商内部的数据分析应用需要对电子商务销售的数据进行分析形成报告，电子商务管理者通过报告的结果及时调整电子商务经营策略。这就涉及到电子商务应用和其他应用进行交互的问题，这种交互可能是数据上的共享，也可能是功能上的调用。

以上 3 个层面的需求基本上确定了 ASP.NET Commerce Starter Kit 需要解决的问题，以及在解决这些问题中必须符合的条件。问题可以看成系统最终的目标，解决问题符合的条件可以看成系统的功能规格。在实际项目的开发过程中，需求都是系统分析师经过对现实问题进行反复调研和总结之后形成的，需求需要经过多次的论证和讨论，形成最终的需求文档。需求文档直接指导项目的开发方向和技术框架设计，脱离需求文档的项目最终是不会取得成功的。

2.1.2 系统功能

根据需求分析中对系统的要求，ASP.NET Commerce Starter Kit 的设计者规划了一系列的系统功能。如果说需求分析是提出问题，那么系统功能就是解决问题，针对需求建立功能说明文档。

下面将 ASP.NET Commerce Starter Kit 系统功能总结为如下几个方面。

(1) 通用模块

通用模块实际上并不是一个提供具体功能应用的模块，而是将系统的通用部分提取出来，作为统一的控件进行调用。提取通用模块的优点在于统一站点的风格，最大程度地减少开发工作量等。

系统中可以被提取成为通用模块的部分都具有比较明显的共同点：被多次调用，具有单元性质但是并不能独立完成某个功能。凡是符合这个特点的模块都可以被提取成为通用模块。例如系统中所有页面的导航菜单和商品分类列表，在系统的绝大多数页面都被调用，而且这些部分具有单元性质，和页面中的其他元素没有耦合，因此可以将这些部分提取成为公用模块以供调用。

(2) 用户模块

用户是系统中最重要角色，所有的操作都是用户主动发起的，也可以说用户是系统的驱动力。在用户模块中，包含用户的注册、登录等方法，但是并不会包含用户参与购物的流程。

(3) 商品列表与信息

商品列表和信息是电子商务应用的基础，用户发起购物流程都是从对商品的浏览开始的。用户可以通过几种不同的方式获得商品信息，包括查看商品分类列表、热门商品列表或直接搜索商品等。

(4) 购物车

购物车模块是电子商务应用的核心。购物车的概念是从现实世界中的购物流程中抽象出来的，用户可以将自己想购买的商品放入购车，然后提交购物车信息完成一笔订单。购物车是一个临时存储购物信息的介质，其中的购物信息都是可以动态添加、修改、删除和维护的。

(5) 订单模块

订单是用户购物信息的最终体现形式。当用户将购物车中的信息提交成为订单，说明用户已经确认了购物信息，接下来电子商务的提供者或是管理员就可以处理订单信息，配送商品。订单信息实际上被分为多个状态来区分订单所处的阶段，具体可以划分为：订单被提交之后未被管理员确认、订单已经被确认但是尚未配送、商品已经配送但是未送达、商品顺利送达给客户。

(6) 第三方交互

第三方交互为第三方应用程序提供了和系统交互的接口。从严格意义上讲，系统在其他功能模块中提供的功能都可以提供交互接口，供第三方应用程序调用。但是在实际的应用中需要根据具体需求，设计第三方交互接口，只提供必要的功能和方法即可。

第三方交互需要严密的身份验证，因为交互的接口在物理上是暴露的，所以只有通过接口内部的身份验证来提高安全性。

2.2 UML 图

UML（统一建模语言）是一种用于对软件密集型系统的制品进行可视化、详述、构造和文档化的图形语言。UML 给出了一种描绘系统蓝图的标准方法，其中既包括概念性的事物，如业务过程和系统功能，也包括了具体的事物，如用特定的编程语言编写的类、数据库模式和可复用的软件结构。

UML 图根据描述的功能不同可分为很多种类型，如系统类构造图是描述系统中类的结构和属性的图，用例图是描述系统中用例基本信息的图，交互图是描述系统中交互行为的图，活动图是描述系统活动的流程图。

2.2.1 用例图

用例图主要是对系统、子系统或类的行为进行建模。图 2-1 是系统的总体用例图，描述系统中的用户与功能模块之间的关系。

2.2.2 类图

类图是面向对象建模中最常见的一种图，描述类、接口、协议以及它们之间的关系。图 2-2 是对系统中的类分别进行的类图建模。

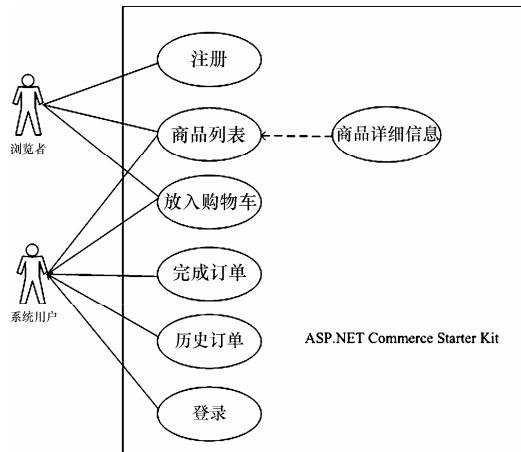


图 2-1 ASP.NET Commerce Starter Kit 系统用例图

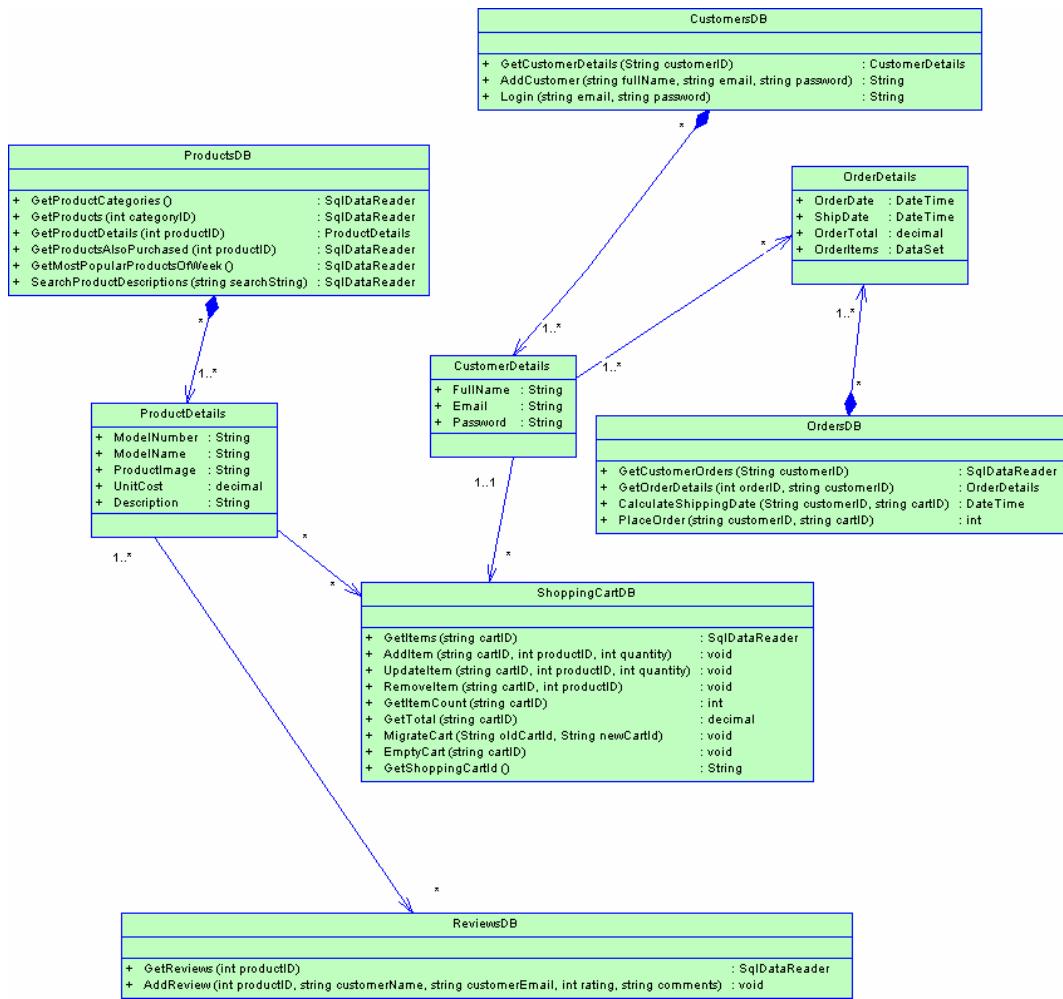


图 2-2 ASP.NET Commerce Starter Kit 类图

2.2.3 活动图

活动图是 UML 中用于对系统的动态方面建模的图形，一张活动图从本质上说是一张流程图，显示从活动到活动的控制流。本书中采用了对系统中用户通过身份验证和购物流程两个活动通过活动图建模，如图 2-3 和图 2-4 所示。

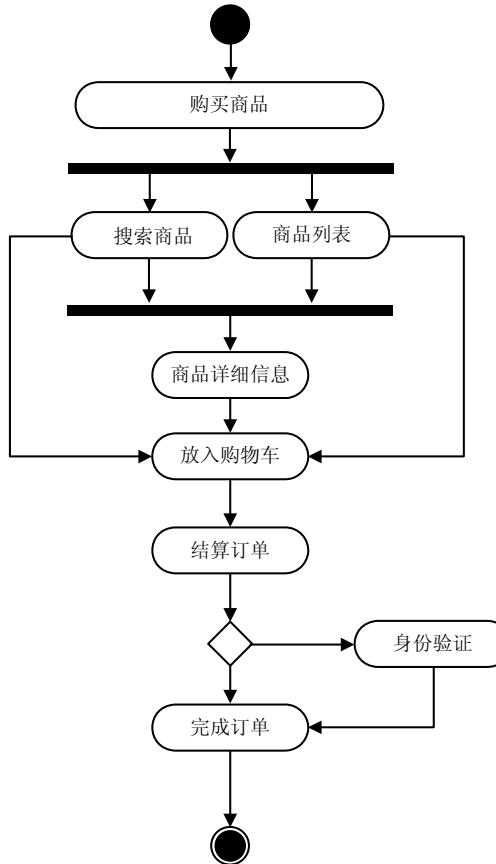


图 2-3 购物流程活动图

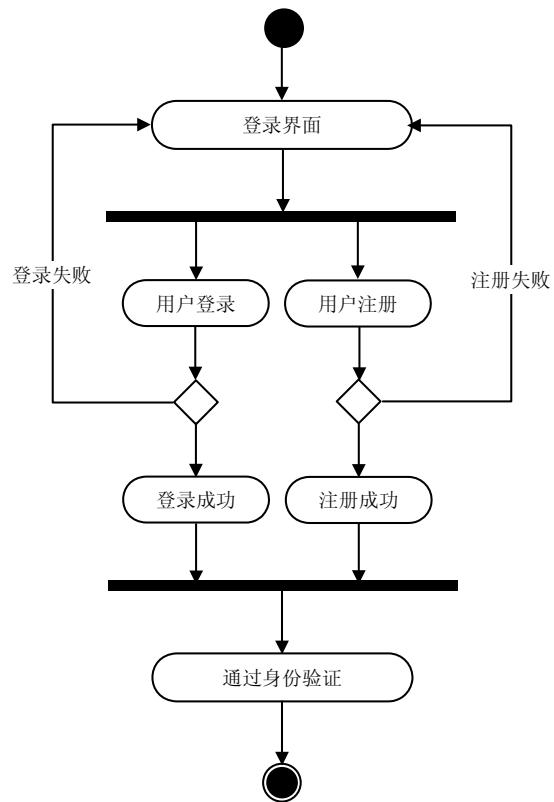


图 2-4 身份验证活动图

2.3 系统架构

经过了对系统需求分析和 UML 的分析，基本上完成了 ASP.NET Commerce Starter Kit 的系统框架的建立，但是已经做过的工作都是设计方面的，下面涉及到的就是实际的工作了。本节中介绍 ASP.NET Commerce Starter Kit 中的文件结构、命名和编码约定、部署等方面问题的总体规划。

(1) 命名和编码约定

命名和编码的约定是软件工程中很重要的一部分内容，特别在多人进行协同工作的团队中，命名和编码的约定显得尤为重要。每个软件工程师都有自己的一套命名规则、编码约定或一些其他的习惯，这就导致了同一事物的不同命名、表达和实现方式。

ASP.NET Commerce Starter Kit 作为一个教程式的示例，在命名和编码上的约定就要采用业界内部承认的标准。Microsoft 公司和.NET 开发团队推荐了一套统一的命名规则。命名和编码的原则实际上包含

两件事情：命名和大小写。Microsoft公司强烈推荐使用叫作Pascal的大小写规则，该规则约定在变量中使用的所有单词的第一个字符都大写，并且不使用空格和符号。Microsoft公司推荐的另外一种大小写规则叫作camel，该规则约定在变量中使用的第一个单词的首字母小写，其余单词的首字母都大写。Microsoft公司推荐的两种命名规则实际上是不会冲突的，因为两种命名规则的使用范围不一样。Microsoft公司推荐在方法的参数和私有成员变量命名中使用camel规则，在包括类、枚举值、枚举类型、名称、属性、事件、接口、方法、命名空间在内的大部分命名中使用Pascal规则。在ASP.NET Commerce Starter Kit的设计中严格地遵守了Microsoft公司推荐的命名规则。

(2) 文件夹结构

有组织的文件夹结构是一个成功项目必不可少的。项目中代码文件需要有组织地放置，在ASP.NET Commerce Starter Kit中，文件夹的结构显得十分有序。

注意：本书中仅介绍ASP.NET Commerce Starter Kit项目中开发涉及到的目录，对于ASP.NET Commerce Starter Kit的安装和帮助文件并不进行详细介绍，并把IIS中ASP.NET Commerce Starter Kit应用程序的主目录作为项目文件的主目录。

在ASP.NET Commerce Starter Kit的主目录下有几个需要介绍的目录：BIN目录为项目编译之后的DLL文件，Components目录为项目中数据访问层的类库文件夹，Images目录为项目中图片存放的文件夹，ProductImages目录为商品图片文件夹。在项目的主目录下存放页面文件、页面后台编码类文件以及页面资源文件。

(3) 命名空间

如果开发者没有大型的.NET项目经验，打开ASP.NET Commerce Starter Kit的工程之后会发现，命名空间的使用不可理解。.NET的命名空间是类、枚举类型等的逻辑容器，因此对命名空间的命名显得尤为重要。一个命名空间能够包含多个程序集和模块。

在ASP.NET Commerce Starter Kit中统一使用ASPNET.StarterKit.Commerce作为应用程序的顶级命名空间，例如商品模块的数据访问层命名空间为ASPNET.StarterKit.Commerce.ProductsDB，首页的后台编码类的命名空间为ASPNET.StarterKit.Commerce.CDefault。

2.4 数据库设计

在分析了系统需求以及系统架构之后，本节中将分析系统的数据库整体设计。由于在需求分析中，已经明确使用Microsoft SQL Server作为系统的数据库，因此基于关系型数据库的设计模式就是必需的。图2-5所示是系统中数据库的模型图。

注意：在后续章节具体功能模块介绍中会详细分析数据库中的数据表和存储过程。

2.5 设计启示

本章中分析了ASP.NET Commerce Starter Kit的需求分析、系统功能设计等设计方面的细节，从项目整体的角度分析了ASP.NET Commerce Starter Kit。经过分析会发现，ASP.NET Commerce Starter Kit的设计者实际上并不是在构建一个真正投入应用的应用程序，而是一个示例教程或一个可供二次开发的电子商务系统框架。基于系统特殊的应用背景，系统在设计上并没有将电子商务中所有的功能都包含，而且在系统的功能中也尽量采取从电子商务应用系统中抽象出来的数据模型。

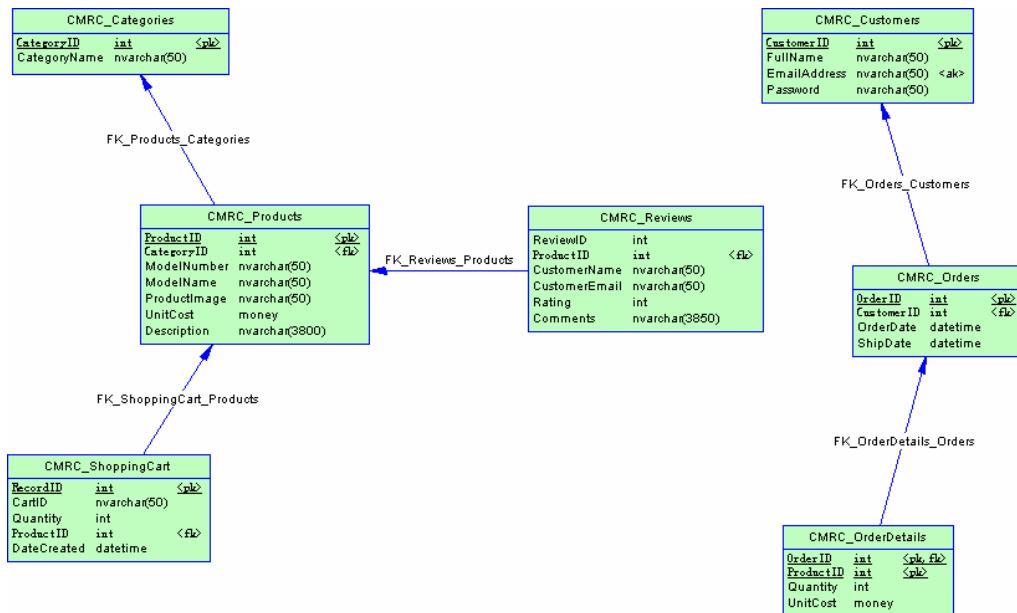


图 2-5 ASP.NET Commerce Starter Kit 数据模型图

设计者希望开发者通过对 ASP.NET Commerce Starter Kit 的学习和研究，总结和发现电子商务应用程序的模型以及电子商务应用程序的基本运作原理，并理解良好的软件设计思维和习惯对于一个软件项目的重要性。

第3章 技术点 解析

本章将详细分析 ASP.NET Commerce Starter Kit 中比较突出的几个技术点，其中包括服务器控件、多层结构、Web.config 的配置和 Web Service。这些技术点不仅是 ASP.NET Commerce Starter Kit 中比较突出的技术点，也是 ASP.NET 技术中比较重要的技术点。ASP.NET 项目的开发过程，和以往的 ASP 技术存

在着较大的差距，基于 .NET Framework 的 ASP.NET 技术提出了一套全新的 Web 开发理念和开发方法，用面向对象的思想进行程序设计、页面表示与逻辑代码的剥离、Web 站点的可配置性等方面都是 ASP.NET 中的新特性。在 ASP.NET Commerce Starter Kit 中，上面提到的 ASP.NET 新特性都有体现。本章是对知识点的简单介绍，尝试从知识点的本身出发，结合项目实践对知识点进行深入的探讨和研究，希望读者通过本章的内容能够掌握这些知识点。

3.1 服务器控件

3.1.1 服务器控件的特点

(1) 服务器控件的对象性

在 ASP.NET 中将面向对象的程序设计思想引入到 Web 应用程序的设计中，在 ASP.NET 页面中一切皆为对象。无论是页面本身，还是页面当中的元素以及后台的数据集，都可以用对象的思想来理解。Web 页面就像一个容器，承载了包括页面对象在内的页面内部用到的所有对象。页面内部的元素都是以服务器控件的形式存在于 ASP.NET 的页面中的。

服务器控件在项目编译时随同页面一起编译，当访问者访问 ASP.NET Web 应用程序时，查看发送到客户端的页面 HTML 代码便会发现，使用了服务器控件的页面在代码上和普通的 HTML 代码没有什么区别。但是在服务器端，服务器控件的代码和普通的 HTML 控件代码是不同的。

在 ASP.NET 页面的源代码中，服务器控件也是以普通的 HTML 标签的形式显示，只不过这些标签有统一的格式和书写方法，并且每个标签都有对属性 runat="server" 的声明，表明这些服务器控件是运行于服务器端的。既然服务器控件是运行于服务器端

的，那么设计者就可以在页面或是页面后台编码类中调用和使用这些控件，在后台编码类中可以像调用其

他对象一样调用这些服务器控件对象。

(2) 服务器控件的事件驱动性

服务器控件需要事件驱动，这和面向对象的思想很吻合。对象本身并不能完成某个功能或实现某个逻辑，需要调用其方法才能实现对对象的驱动。例如当 ASP.NET 页面被访问的时候，服务器首先调用页面的装载事件和页面中所有服务器控件的装载事件。虽然开发者也许不曾定义或设计过这些事件，但是这些事件还是默认地被触发。

所有的服务器控件都有一部分共同的事件，例如载入事件、销毁事件等。开发者可以定义这些事件的触发代码来完成需要的功能，例如数据的绑定、属性的设置等。

通过对服务器控件特点的了解和分析，可以看出服务器控件比普通的 HTML 控件有如下的优点。

- 服务器控件是一个对象，可以在后台编码类中使用面向对象的思想对其进行操作。
- 可以对服务器控件的值或是状态进行设置和操作，这种操作可以在前端页面完成也可以在后台编码类中完成。
- 不必调用 Request 对象就可以直接访问服务器端控件的值。
- 使用服务器端控件可以使前端页面完全 HTML 化，不必在前端页面嵌套服务器端代码。
- 能够响应控件的事件，并能够在服务器端对控件事件进行编程。

3.1.2 服务器控件的简单应用

在 ASP 技术中，页面的 HTML 代码部分就是发送到客户端的 HTML 代码，但是在 ASP.NET 中不是这样的。

例如对文本输入框 TextBox 的使用，ASP.NET 页面显示一个文本输入框的代码如下。

```
<asp:TextBox id="TextName" runat="server">
```

在 HTML 的语法中，TextBox 控件是没有 runat 属性的，显然这是服务器控件所特有的。在后台编码类的页面装载方法中编写下面的代码。

```
public void Page_Load(object sender, System.EventArgs e)
{
    TextName.Text = "Hello, ASP.NET Commerce Starter Kit";
    TextName.Size = 100;
}
```

页面被访问时，在客户端会看到文本输入框中显示的文本是 Hello, ASP.NET Commerce Starter Kit，并且文本输入框的大小是 100 像素。

除了 TextBox 之外，其他 HTML 中的控件也可以在 ASP.NET 页面中使用服务器控件代替。例如一个下拉列表控件，在前台的 ASP.NET 页面中书写如下代码。

```
<asp:DropDownList id="ddl" runat="server">
    <asp:ListItem value="ProductA">ProductA</asp:ListItem>
    <asp:ListItem value="ProductB">ProductB</asp:ListItem>
    <asp:ListItem value="ProductC">ProductC</asp:ListItem>
    <asp:ListItem value="ProductD">ProductD</asp:ListItem>
    <asp:ListItem value="ProductE">ProductE</asp:ListItem>
</asp:DropDownList>
```

在页面后台编码类的页面装载方法中书写如下代码。访问 ASP.NET 页面会发现，下拉列表控件的选定项是 ProductD。

```
private void Page_Load(object sender, System.EventArgs e)
{
    ddl.SelectedItem.Text = ddl.Items.FindByValue("ProductD").ToString();
}
```

上面两个例子演示了服务器控件的简单用法，服务器端控件功能强大，操作方便，对于设计人员来说，服务器端控件的优点不言而喻，但是并不是所有的情况下都适合使用服务器端控件的。实际在 ASP.NET 中普通的 HTML 元素，设置其属性 runat="server"，也可作为服务器端控件运行。那么什么时候选用普通的 HTML 元素，什么时候选用运行于服务器端的 HTML 元素，什么时候选用真正的服务器端控件呢？3 种形式的元素在表现形式上是一样的，但是其功能和运行原理不同。

- 当元素仅用于运行客户端脚本的时候，建议采用 HTML 元素。例如利用一个按钮打开另外一个页面或是完成当前页面向另外一个页面跳转，其中不涉及到页面内其他控件值的传递和操作，完全可以使用 JavaScript 代码来完成这部分工作。这样可以大大地减轻服务器的负担，因为 JavaScript 代码是运行于客户端的。
- 当元素不需要访问服务器资源与服务器产生交互的时候，建议采用 HTML 元素。例如页面内部固定的文本信息，就不需要采用服务器端控件。
- 当元素仅作为打开一个超链接的值而并不需要对超链接进行数据绑定的时候，建议采用 HTML 元素。

3.1.3 ASP.NET 可利用的服务器控件

ASP.NET 提供的服务器端控件分为 6 组。

- HTML 服务器控件。这一类服务器控件就是在 HTML 服务器控件的基础上设置其属性 runat="server"，这一类控件具有和 HTML 控件相同的 HTML 语法结构，但是是运行于服务器端的。
- ASP.NET 页面验证控件。这是一组特殊的服务器控件，这组控件的功能是对页面内部其他控件输入的值进行验证和判断。可以选择页面验证控件的验证行为是运行于服务器端或是运行于客户端。
- ASP.NET Web 表单控件。这一组和普通 HTML 元素相对应的控件，例如文本框、按钮、超链接等。这组控件可以设置的属性几乎相同，无论在前端页面还是在后台编码类，都可以轻松地对其属性进行设置或获取。
- ASP.NET 列表控件。这一组可以绑定自数据源的列表显示控件。数据源可以是来自数据库查询的数据集，也可以是数组变量，列表控件可以循环地显示数据源中的数据条目。这组列表控件主要包括 DataGrid、DataList、Repeater，开发者可根据实际需求的不同选用不同的控件。
- ASP.NET 功能控件。在 ASP.NET 中有几个功能性控件，例如日期控件 Calendar 和广告显示控件 Ad Rotator。这一类控件都有特殊的功能和应用背景，在接收和显示数据方面多功能控件也有各自的特点。
- ASP.NET 移动控件。这一类控件是专门应用于移动 Web 应用程序开发的控件。这类控件与其他一些 ASP.NET 控件具有相同的功能，同时还具有应用于移动设备的特殊扩展功能。

3.2 多层结构

3.2.1 多层结构的概念

将解决方案的组件分隔到不同的层中。每一层中的组件应保持内聚性，并且应大致在同一抽象级别。每一层都应与它下面的各层保持松散耦合。从最低级别的抽象开始称为第 1 层。这是系统的基础。通过将第 J 层放置在第 $J-1$ 层的上面逐步向上完成抽象阶梯，直到到达功能的最高级别称为第 N 层。客户端应用程序使用基于服务器的应用程序提供的一组服务。这些服务由服务器应用程序的最高层公开。因此客户

端必须只与最高层交互，而无法直接了解任何较低的层。但是多层结构也有松散的组织方式，较高的层可以直接操作较低的层而不是仅仅操作下一级的层。图 3-1 所示是多层结构的组织图。

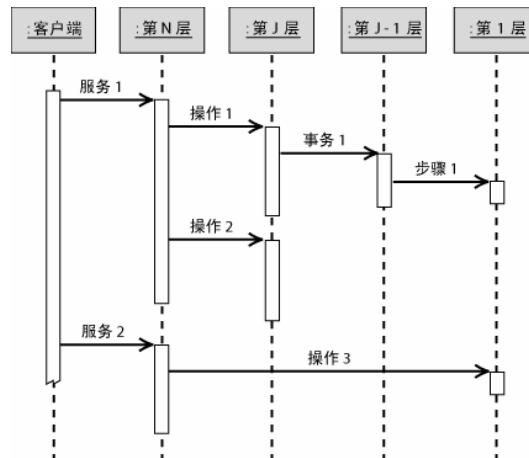


图 3-1 多层结构组织图

通过图 3-1 可以得出多层结构几个基本的特点。

- 一个传入调用会导致多个传出调用。第 N 层上的服务 1 的调用说明了这种情况。当较高级别服务聚合几个较低级别服务的结果，或协调多个必须按特定的顺序执行的较低级服务的执行时，通常会发生这种情况。例如，ASP.NET 页面可以将客户输入的财务信息提供给订单处理的组件，而订单处理组件的输出将提供给发票处理组件。
- 此方案说明了松散的分层方法。服务 2 的实现绕过了所有中间层而直接调用第 1 层。这种情况的常见示例是绕过任何中间业务逻辑层而直接访问数据访问层的表示层。数据维护应用程序通常使用这种方法。
- 顶层服务的调用并不一定会调用所有层。这一概念是通过服务 1 到操作 2 顺序来说明的。当较高级别处理自身中的调用，或者缓存了较早的某个请求的结果时，就会发生这种情况。例如，表示层组件通常缓存数据库查询的结果，这使得将来调用时不必调用数据访问层。

3.2.2 ASP.NET 中的多层结构

在 ASP.NET 中使用多层架构，也遵循多层架构的基本原理和概念，ASP.NET 中的层可以分为 3 种类别。

(1) 表示层

表示层提供应用程序的用户界面 (UI)，在 ASP.NET 中的页面就是 UI 的表现形式。表示层是系统与用户沟通的唯一渠道，是系统功能的展现。

(2) 业务层

业务层实现应用程序的业务功能。在 ASP.NET 中通常以类库的形式封装系统需要处理的业务逻辑，业务层的设计与开发是 ASP.NET 的核心，主要将围绕系统需求分析制定的业务逻辑封装成为 .NET 类库供表示层调用。

(3) 数据层

数据层提供对外部系统（如数据库）的访问。该层不仅是数据库系统，也包括 ASP.NET 应用程序中的数据访问类库，这些类库中并不包含系统的业务逻辑而仅仅是数据的存取操作方法。

对于简单的 ASP.NET 应用程序，3 层的概念也许很模糊，层与层之间在功能上的划分不是很明显。但

是对于复杂逻辑密集的 ASP.NET 应用程序，可能采取的就是多层的架构，但是每一层都应该属于 3 层分类中的某一个类别。

在 ASP.NET 应用程序中，多层架构的具体实现可能进行多种形式的扩展，例如前台表示层通过统一的控制器调用后台的逻辑应用程序，这种称之为前端控制 Front Controller 的设计模式实际上是在表示层和业务层之间建立一座沟通的桥梁，控制器既不明确属于表示层也不属于业务层。另外在某些大型的 ASP.NET 应用程序中，可能将业务层中重要的对象和方法封装到统一的类库中，这些类库以 DLL 形式提供给系统，这些类库内部也分了多层的结构，层与层之间也保持着多层结构的特点。

通过对在 ASP.NET 中应用多层架构的分析，可总结出如下优点。

- 降低前端页面逻辑密集程度。在 ASP.NET 页面中，设计人员可以在一个网页内实现方法。随着页面中所体现的业务逻辑复杂性不断提高，或者对网页之间共享代码的需要不断增加，分离代码的各部分就变得更加有用。
- 减少代码重复。在一个 ASP.NET 应用程序中，所有和数据处理相关的页面都用到了同样的操作数据库的方法，这时抽象出单独的数据层大大地降低了代码的复用率，同时也提高了开发效率。
- 分离职责和问题。修改 ASP.NET 页面所使用的技巧不同于编写数据库访问代码所使用的技巧。在一个项目的开发中，这两部分的工作可能有不同的开发人员来完成。通过分离视图和模型，可以保证两部分工作同时进行。
- 优化的可能性。将职责分成特定的类可以提高进行优化的可能性。每次发出请求时，就会从数据库加载数据。因此，在某些情况下可以对数据进行缓存，这样可以提高应用程序的总体性能。但是，如果不分离代码，缓存数据就会很难实现，或者不可能实现。
- 可测试性。通过将模型与视图分离，开发人员可以在 ASP.NET 环境以外测试业务逻辑层。

3.3 配置 Web.config

3.3.1 Web.config 的功能和特点

ASP.NET 的配置信息由 machine.config 和 Web.config 两部分组成，其中 machine.config 将服务器的公用配置信息存放在服务器系统目录下，Web.config 文件存储的是单个 ASP.NET Web 应用程序的配置信息，位于每一个 ASP.NET Web 应用程序的根目录下。Web.config 中的配置信息会覆盖 machine.config 中的配置信息。

machine.config 文件的功能是用来指定应用于服务器上的所有 ASP.NET 应用程序的设置，而且每个服务器上只能存在一个这样的文件。machine.config 存储的设置将被所有的 ASP.NET Web 应用程序所继承。例如，machine.config 文件中配置了会话状态将被保存的设置，那么该设置将应用于服务器的所有 ASP.NET Web 应用程序实例中去。但是如果有一个应用程序在运行时不保存任何会话状态，那么可以在该应用程序的 Web.config 中进行配置声明，这个过程相当于对 machine.config 文件的重写。所以每一个 ASP.NET Web 应用程序都可以拥有一个 Web.config 文件，而且文件中的设置总是重写先前在 machine.config 中相应的配置。

单个 ASP.NET Web 应用程序的配置信息是基于 Web.config 文件的，该文件以 XML 的形式描述某个 ASP.NET Web 应用程序的配置信息。格式如下。

```
<configuration>
<system.web>
<sessionState timeout="10"/>
```

```
</system.web>
</configuration>
```

上述代码的作用是将 ASP.NET Web 应用程序的会话超时时间设置为 10 分钟，重写 machine.Config 中会话超时时间为 20 分钟的设置。在这段 XML 文件的构建中，sessionState 标签包含在 system.web 标签中，整个文件包含在 configuration 标签中，是一种典型的 XML 文件格式。使用这种 XML 配置文件有如下优点。

● 配置设置的易读性

打开一个 Web.config 文件会发现，其文件格式和普通的 XML 几乎没有区别。无论是对于设计者还是浏览者，这种文件格式都清楚地解释了 ASP.NET 的配置信息。

● 更新的即时性

ASP.NET 中的程序代码一旦发生改变，需要经过重新编译才能生效。但是 Web.config 的配置信息更改后，是不需要 ASP.NET 应用程序重新编译执行就可以应用生效的。因此可以尽量将 ASP.NET 应用程序中需要更改的配置信息写入 Web.config 文件，提高应用程序的可配置性。

● 自定义配置信息节点

除了 Web.config 固有的信息节点，在 Web.config 中开发人员可以根据需要，自定义配置信息节点。这些自定义的配置信息可以随时被 ASP.NET Web 应用程序读取。

3.3.2 配置 Web.config

Web.config 可以对 ASP.NET 的多种信息和行为进行配置，每一种配置信息都是以节点的形式体现出来。下面就是 Web.config 文件中主要节点的说明。

1. <appSettings>

<appSettings> 节点中包含了特定于应用程序的配置。在多数的 ASP.NET Web 应用程序中将数据库连接字符串和一些配置路径信息放在这个节点里面。该区域在 add 标签中以键值对来添加配置设置，另外还支持 remove 和 clear 标签来删除和清除先前指定的应用程序设置。在一个 ASP.NET Web 引用程序的 Web.config 文件中对<appSettings> 节点做如下设置。

```
<appSettings>
    <add key="ConnectionString" value="server=localhost;database=Commerce;uid=sa;pwd=" />
</appSettings>
```

在应用程序的后台编码类中使用如下的代码读取该配置信息。

```
SqlConnection myConnection = new SqlConnection(ConfigurationSettings.AppSettings["ConnectionString"]);
```

这行代码通过调用 ConfigurationSettings.AppSettings 方法取得名为 ConnectionString 的配置字符串，将该字符串作为数据库连接字符串实例化一个数据库连接对象。

2. <authentication>

<authentication> 节点指定了应用程序的用户身份验证模式。ASP.NET 中支持 4 种身份验证模式，可以通过配置<authentication> 节点设置不同的身份验证模式，4 种身份验证模式如下。

(1) Windows

ASP.NET Web 应用程序默认为 Windows 身份验证，登录 Windows 的用户身份将作为验证信息，当然也包括 IIS 中定义的 Internet 来宾用户信息。在基于活动目录身份验证的 ASP.NET Web 应用程序中，采用这种验证模式可以获得活动目录域下当前用户的身份信息。

(2) Forms

如果身份验证模式被设置成为 Forms，则可以在 authentication 的 forms 子元素中定义基于 Web 表单的身份验证行为。

(3) Passport

设置为 Passport 模式，ASP.NET Web 应用程序可以使用 Microsoft Passport 身份验证和授权信息。

(4) None

使用该模式，应用程序将被设置为匿名用户访问，在应用程序中无法获得用户身份验证信息。

Forms 身份验证允许使用基于 Web 表单的身份验证，该种验证模式获得客户端的请求信息和 Cookie，一旦请求通过，将取用该 Cookie 信息作为用户的身份表示。这种验证模式相对灵活，可以在应用程序中调用 Forms 身份验证所提供的 API 对通过身份验证的用户信息进行调用。以下是 forms 元素的属性说明。

- name 属性。该属性指定了身份验证信息中用户 Id 使用的 Cookie 名称。
- loginUrl 属性。当用户请求没有通过身份验证的时候，应用程序将请求重新定向到一个页面，loginUrl 属性指定该页面的 URL。
- protection 属性。该属性指定了 Cookie 的值的发送形式，是否经过加密处理。
- timeout 属性。该属性指定了在身份验证中以分钟为单位的 Cookie 有效时间。
- path 属性。该属性指定了用户身份信息 Cookie 的存放位置，默认为 path="/"，即服务器的根目录。

下面是使用 Forms 模式身份验证的配置代码段。

```
<authentication mode="Forms">
  <forms name="CommerceAuth" loginUrl="login.aspx" protection="All" path="/" />
</authentication>
```

3. <authorization>

<authorization>节点指定了授权访问应用程序的用户。<authorization>节点包含 allow 和 deny 标签，开发者可以定义授权访问应用程序的用户和拒绝访问应用程序的用户。下面的配置信息是<authorization>的默认值，允许所有用户访问应用程序。

```
<authorization>
  <allow users="*"/>
</authorization>
```

下面的配置信息是拒绝所有未通过身份验证的用户访问应用程序。使用这种配置模式可以防止未通过身份验证的用户访问特定的 Web 页面或是资源。

```
<authorization>
  <deny users="?" />
</authorization>
```

4. <compilation>

<compilation>节点为 ASP.NET 应用程序指定编译选项。编译选项包括编译模式、编译器的类型、编译时引用的程序集和命名空间等信息。

5. <customErrors>

<customErrors>节点指定了应用程序运行出错时的客户端呈现模式。将<customErrors>的 mode 属性指定为 On、Off 或默认值 RemoteOnly。On 表示客户端接收到一个普通的错误消息，或被定向到 defaultRedirect 属性所指定的特定错误页面；Off 表示客户端会获得全部的 ASP.NET 错误信息；RemoteOnly 适用于开发模式，对于远程用户显示自定义的错误信息，对于服务器本地用户显示错误的详细信息。下面是<customErrors>节点示例。

```
<customErrors mode="RemoteOnly" defaultRedirect="ErrorPage.aspx" />
```

6. <globalization>

<globalization>节点指定了 ASP.NET Web 应用程序的全球化选项。属性 requestEncoding、responseEncoding、fileEncoding 指定了处于请求、响应和应用程序所处理文件的编码方式。Culture 属性指定了特定的语言选项，uiCulture 属性指定了由区域决定的语言选项。下面是<globalization>节

点的使用示例。

```
<globalization fileEncoding="utf-8" requestEncoding="utf-8" responseEncoding="utf-8"/>
```

7. <pages>

<pages>节点指定了 ASP.NET Web 应用程序特定于页面的设置。

8. <sessionState>

<sessionState>节点指定了 ASP.NET Web 应用程序如何管理会话范围的变量和状态。Mode 属性确定会话状态在哪里以及如何保持，Off 模式表示不保持任何会话状态，Inproc 模式表示会话状态由当前服务器保持，StateServer 模式表示使用一个不同的服务器保持会话状态，SQLServer 模式表示使用一个不同服务器的 SQL Server 数据库保存会话状态。其中 StateServer 和 SQLServer 模式只有在 ASP.NET 应用程序被部署到一个 Web Farm 中时才会生效。<sessionState>节点的 cookieless 属性指定会话是否使用 Cookie 的形式保存，True 模式为不使用，False 模式为使用。timeout 属性指定了以分钟为单位的会话有效期。stateConnectionString 属性和 sqlConnectionString 属性分别指定了 Mode 属性为 StateServer 和 SQLServer 模式时连接其他服务器的连接字符串。下面为<sessionState>节点的使用示例。

```
<sessionState mode="Off" />
```

9. 标签<location>

<location>在 Web.config 中不是一个独立的节点，而是一组标签。该标签组指定了特定目录或是特定文件的配置信息，该配置信息可以和整个 ASP.NET Web 应用程序的配置信息不同。一般对于某一部分文件或目录的某一项配置信息进行特殊定义的时候使用该标签。path 属性指定了需要特殊定义的文件名或目录。下面的配置示例是指定了特定的几个文件的用户授权信息，未通过身份验证的用户是不能访问这些文件的。

```
<location path="Checkout.aspx">
  <system.web>
    <authorization>
      <deny users="?" />
    </authorization>
  </system.web>
</location>
<location path="OrderList.aspx">
  <system.web>
    <authorization>
      <deny users="?" />
    </authorization>
  </system.web>
</location>
<location path="OrderDetails.aspx">
  <system.web>
    <authorization>
      <deny users="?" />
    </authorization>
  </system.web>
</location>
```

除了以上常用节点之外，Web.config 还包括很多其他节点，例如<httpRuntime>、<identity>、<machineKey>、<processModel>等。

3.4 Web Service

3.4.1 Web Service 简述

随着.NET平台的推出,.NET平台下的Web Service(Web服务)技术受到开发者们的广泛关注。在信息化高速发展的今天,信息的管理以及传递不仅局限于Web站点内部,Web站点之间的信息传递也越来越重要。由于受到Web站点以及服务器的安全考虑,完全开放服务器来达到信息传递以及共享的目的是不太现实的。需要一种新的技术框架,将站点间的信息联系起来,Web Service技术就是在这种背景下产生的。

从理论上讲,Web Service技术并不是新的概念,因为在此之前多种技术框架都允许分布式应用程序通过网络共享数据甚至业务逻辑。但是.NET Framework支持的Web Service使用了XML标准并支持HTTP,这使得Web Service遵循了标准的Web通信协议。XML和HTTP的合并组成了新的技术概念——简单对象访问协议(SOAP, Simple Object Access Protocol),这是一个专门用于分散和分布式环境下网络信息交互的通信协议,SOAP下的应用程序能够通过标准的HTTP协议进行通信,交换的数据信息是XML格式的。SOAP的目的就是提高异构程序和平台之间的相互交换能力,提高应用程序的扩展性和可交互性。.NET Framework提供了一整套创建和发布Web Service的类库及方法,基于.NET Framework下的Web Service能够与其他平台或应用程序发生交互。

Web Service技术中最重要的内容就是所有Web Service使用相同的标准以及数据格式,那么就涉及到了规范和标准的制定,下面介绍的就是Web Service技术中遵循的规范以及标准。

1. 规范

为了解决Web Service的发现、描述、集成以及协议的问题,将Web Service规范总结为3大类。

(1) 发现

有两个规范用于解决Web Service的发现问题——UDDI(Universal Description, Discovery and Integration,通用描述、发现和集成)和DISCO(Discovery)。UDDI提供一个统一发布Web Service的目录。DISCO可以将一台服务器上的所有Web服务归入一个组,并提供链接模式的文档,此文档用于描述服务。

(2) 描述

WSDL(Web Service Describing Language, Web服务描述语言)是一种专门用来描述Web服务的语言。该语言中定义了发现Web Service的位置、Web Service所支持的方法和属性、数据类型以及用于服务通信的协议等。使用WSDL形成的Web Service描述文档可以被Web Service提供者发布给Web Service的调用者。Web Service的调用者通过对该文档的分析构建调用端的对象。

(3) 协议

SOAP,描述使用HTTP传输XML格式数据的格式。

2. 使用发现、描述和协议

在调用Web Service时,设计者可以使用UDDI、DISCO和WSDL,其步骤如下。

- ① 通过UDDI,获取提供的Web Service目录。
- ② 通过UDDI提供的Web Service目录,检索到希望调用的服务和方法。
- ③ 通过UDDI提供服务的URI,可以映射和获得该服务的DISCO文档。
- ④ 分析DISCO文档发现Web Service的位置。
- ⑤ 跟踪DISCO文档获得该Web Service的WSDL文档类表的位置。
- ⑥ 分析WSDL文档,基于WSDL文档描述构建第三方应用程序调用Web Service时使用的对象以及方法。

注意：尽管 DISCO 文档和 WSDL 文档可以作为 Web Service 放置在相同的服务器上，但是这并不是必要的。DISCO 文档作为链接文档，可以存放在任何 Web Service 调用者可以获得的位置。而 WSDL 文档作为 Web Service 的描述文档，其作用是描述 Web Service，其放置位置同样不受限制。

通过以上发现 Web Service 的过程，开发者就可以在第三方应用程序中像使用本地应用程序的对象和方法一样调用 Web Service 提供的方法，并构建相应的对象。

3.4.2 构建 ASP.NET 的 Web Service

ASP.NET Web Service 可以使用 Visual Studio.NET 等开发工具进行开发。ASP.NET Web Service 的文件后缀名为.asmx，和普通 Web 页面一样具有代码后置等特性，这使得 ASP.NET Web Service 的开发过程和普通的 Web 页面一样，仅仅是缺少页面表示层的开发。下面将演示构建一个简单的 Web Service 的方法。首先在工程中创建一个 SumTest.asmx 的页面，在页面中添加如下代码。

```
<%@ WebService Language="c#" Codebehind="SumTest.asmx.cs" Class="SumTest" %>
```

以上代码定义了该 asmx 文件的后台编码类文件 SumTest.asmx.cs，在该后台编码类文件中添加如下代码。

```
using System;
using System.Web.Services;

public class SumTest : WebService {
    [WebMethod(Description="This is a Web Service Test!, EnableSession=false)]
    public int sumnum(int num1, int num2)
    {
        return num1+num2;
    }
}
```

以上 Web Service 中定义了一个简单的 Web Service 的方法 sumnum，该方法能返回两个 int 型数据的和，编译该 Web Service，在浏览器中访问 SumTest.asmx 会看到如图 3-2 所示的页面。



图 3-2 SumTest.asmx

该页面是 Web Service 的发布页面，页面提供了 SumTest.asmx 提供的可被调用方法的列表。单击 sumnum 方法的链接可以参看 sumnum 方法的详细信息，页面如图 3-3 所示。

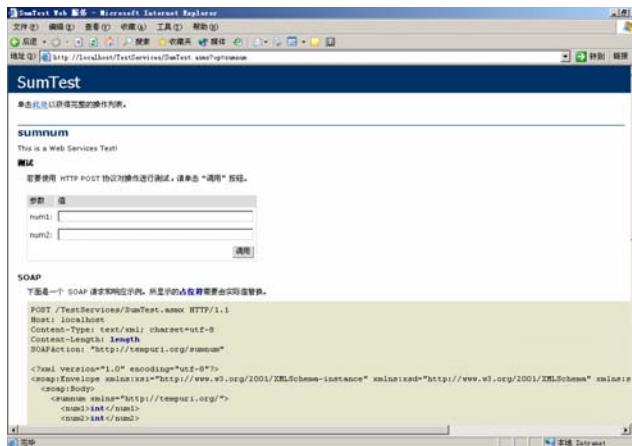


图 3-3 sumnum 方法页面

在 SumTest 方法的页面，首先给出了一个使用 HTTP POST 方式测试 SumTest 方法的表单，其次给出了 SumTest 方法的 SOAP 请求信息范例和 SOAP 响应信息范例，最后给出了 HTTP POST 调用该方法的范例。该方法的 SOAP 请求信息范例代码如下。

```
POST /TestServices/SumTest.asmx HTTP/1.1
Host: localhost
Content-Type: text/xml; charset=utf-8
Content-Length: length
SOAPAction: "http://tempuri.org/sumnum"

<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
    <soap:Body>
        <sumnum xmlns="http://tempuri.org/">
            <num1>int</num1>
            <num2>int</num2>
        </sumnum>
    </soap:Body>
</soap:Envelope>
```

该方法的 SOAP 响应信息范例代码如下。

```
HTTP/1.1 200 OK
Content-Type: text/xml; charset=utf-8
Content-Length: length

<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
    <soap:Body>
        <sumnumResponse xmlns="http://tempuri.org/">
            <sumnumResult>int</sumnumResult>
        </sumnumResponse>
    </soap:Body>
</soap:Envelope>
```

在测试表单中输入测试数据，单击调用按钮会返回一段 SOAP 响应 XML。以下是返回的 SOAP 响应信息。

```
<?xml version="1.0" encoding="utf-8" ?>
<int xmlns="http://tempuri.org/"></int>
```

实际上在这个简单的 Web Service 中，开发者并没有定义任何前端展示页面，Visual Studio.NET 在编译的过程中生成了该 Web Service 的表现页面，但是这些表现页面仅是面对开发人员的。Web Service 的原始目的就是供第三方应用程序调用，因此不具备正式的表现层页面。

3.4.3 在 ASP.NET 中调用 Web Service

在上一节中讲解了如何在 ASP.NET 中构建和发布 Web Services，本节将演示如何在 ASP.NET 应用程序中调用这个 Web Service。

首先在工程中添加 Web 引用并在 URL 中填入 Web Service 的地址，假设在上一节中构建的 Web Service 的 URL 为 <http://localhost/TestServices/SumTest.asmx>，则添加该 URL 之后会发现该 Web Service 的页面如图 3-4 所示。

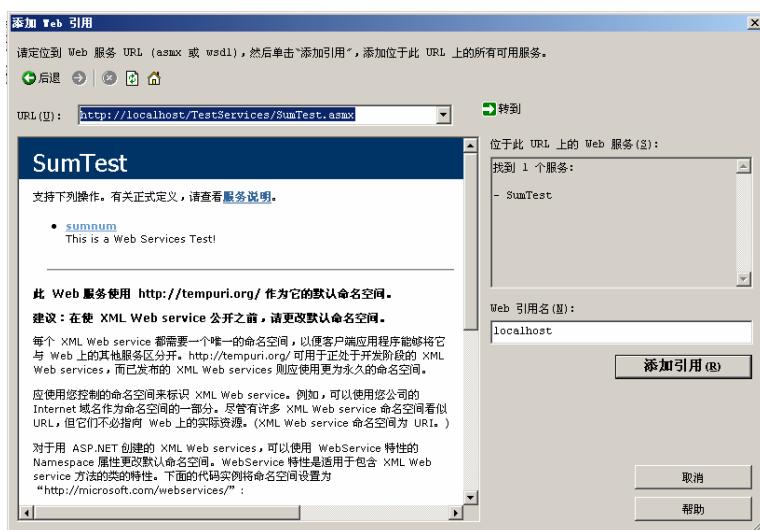


图 3-4 添加 Web 引用

填入 Web 引用的名称之后，单击添加引用的按钮，该 Web Service 就会被引用到当前的工程。在解决方案资源管理器中 Web Service 已经被引用进当前工程中来，如图 3-5 所示。



图 3-5 资源管理器中的 Web 引用

在当前的工程的代码中可以对该 Web Service 进行调用，创建一个 ASP.NET 页面，如图 3-6 所示。

在页面的按钮单击事件中添加如下代码。

```
localhost.SumTest objsumtest = new TestServices.localhost.SumTest();
lblResult.Text = "两个数相加的结果为:";
```

```
lblResult.Text  
objsumtest.sumnum(int.Parse(textBox1.Text.ToString()), int.Parse(textBox2.Text.ToString())).ToString();
```

这段代码的第一行首先实例化一个 Web Service localhost.SumTest 的对象，其中 localhost 为 Web 引用的名称，SumTest 为远程类名称。然后调用该对象的 sumnum 方法实现两个数相加的方法。单击按钮效果如图 3-7 所示。在这个 ASP.NET 页面中，完成了对 Web Service 的调用。



图 3-6 调用 Web Service 方法测试页面



图 3-7 调用 Web Service 测试

本示例实现了在 ASP.NET Web 应用程序中调用 Web Service，并根据 Web Service 提供的方法实现本地的逻辑应用。

3.5 技术点总结

本章中详细讲述了 ASP.NET Commerce Starter Kit 中比较突出的几个技术点，其中包括服务器组件、多层结构、Web.config 的配置和 Web Service 技术。这些技术点从多方面展现了 ASP.NET 的新特性，打破了以往 Web 开发的模式和理念，希望读者通过对这些技术点的学习和理解，对 ASP.NET 技术有更深一层的理解。

第 4 章

系统设计

在第 2 章和第 3 章重点介绍了项目的设计和技术方面的难题，并对 ASP.NET Commerce Starter Kit 的技术框架在宏观上进行了分析。实际在任何一个软件项目设计的初期，除了前面讨论过的问题外，接下来的工作就是系统的概要设计，包括界面风格的定义、编码的约定、通用模块的设计等。具体的工作根据实际的

项目安排而定。系统概要设计部分的工作对于整体系统的开发相当重要，而且对于项目的质量有决定性的作用。

在 ASP.NET Commerce Starter Kit 这个案例中，系统概要设计采取了一个偏向简易化的方向。在界面的整体风格定制以及通用模块的设计中并没有加入对整个项目的过多限制，这种设计模式不仅轻巧易用，也为系统的进一步升级扩展预留了良好的接口。

4.1 页面样式基础

页面的样式基础是系统的前端页面即用户界面（User Interface, UI）的基础，一个 Web 应用程序直接展现给用户的并不是后台的技术架构和原始的数据逻辑，而是前端页面。作为一个完整的 Web 应用程序，需要有统一的页面布局、样式，或是更加基本的 UI 内容，让用户感觉到整个 Web 应用程序的统一和连贯。

4.1.1 使用级联样式表文件

在 ASP.NET Commerce Starter Kit 中设计者采用了统一的级联样式表（Cascading Style Sheets, CSS）文件，级联样式表为 Web 应用程序提供了大量的站点统一的显示属性，这样做最大的好处就是使得整个 Web 应用程序的每一个页面都具有在级联样式表文件中定义好的显示属性。

例如，在显示商品列表的时候，使得每个商品的标题都是以一种特殊的样式显示。如果不采用 CSS 文件，则使用如下的写法。

```
<span style="color: black;font-family: Verdana, Arial;font-size: 12px; font-weight: bold;line-height: 14pt; text-decoration: underline;">This is a product title!</span>
```

在 style 属性中描述了这个 span 标签内部的样式特征，这种做法显然很繁琐。如果在其他页面同样需要列表显示商品，而每一

个

商品的标题在样式上都希望保持一致，那就需要对所有商品标题都一一定义，最大的麻烦在于，一旦站点的整体样式定义有改动，那么需要在每一个页面中都修改这些复杂的样式。

不过在使用级联样式表文件之后，一切将变得极其简单。只需要通过如下的代码就可以方便地实现对商品标题的定义。

```
<span class="ProductListHead">This is a product title!</span>
```

对 ProductListHead 样式做如下的定义。

```
.ProductListHead
{
    color: black;
    font-family: Verdana, Arial;
    font-size: 12px;
    font-weight: bold;
    line-height: 14pt;
    text-decoration: underline;
}
```

将以上这段代码保存为名字为 Style.css 的级联样式表文件，和需要使用样式表的页面保存在同一个文件目录下，并且在这些页面的头部加入如下的<link>标签。

```
<html>
<head>
    <link rel="stylesheet" type="text/css" href="Style.css">
</head>
<body>
.......
```

4.1.2 ASPNETCommerce.css

级联样式表文件作为应用程序 UI 设计最重要的一部分，会被直接以文件的形式部署到项目中去。在 ASP.NET Commerce Starter Kit 中，项目的所有页面使用了级联样式表文件 ASPNETCommerce.css，该文件被部署在应用程序的主目录下。在所有需要使用 CSS 修饰的页面文件的头部，用<link>标签来声明对 ASPNETCommerce.css 这个文件的引用。代码如下。

```
<head>
    <link rel="stylesheet" type="text/css" href="ASPNETCommerce.css">
</head>
```

ASPNETCommerce.css 中的主要样式在站点中的使用见表 4-1。

表 4-1 ASPNETCommerce.css 中的主要样式说明

| 样 式 名 | 使 用 |
|------------------|------------------------------|
| .HomeHead | 站点首页的欢迎信息和错误页面的说明信息样式 |
| .ContentHead | 包括登录页面、商品信息页面在内的部分页面顶部标题文字样式 |
| .SubContentHead | 包括商品列表、商品信息页面在内的部分页面子标题文字样式 |
| .UnitCost | 产品详细信息页面的产品价格文字样式 |
| .ModelNumber | 产品详细信息页面的产品型号文字样式 |
| .ErrorText | 出错信息文字样式 |
| .MostPopularHead | 首页热门商品部分标题样式 |

| | |
|----------------------|---------------------|
| .MostPopularItemText | 首页热门商品部分商品名称样式 |
| 续表 | |
| 样 式 名 | 使 用 |
| .ProductListHead | 商品列表头部标题信息样式 |
| .ProductListItem | 商品列表中商品信息样式 |
| .CartListHead | 购物车页面头部标题信息样式 |
| .CartListItem | 购物车页面中商品列表信息样式 |
| .CartListItemAlt | 购物车页面中商品列表信息交替出现的样式 |
| .CartListFooter | 购物车页面尾部信息样式 |
| .SiteLink | 所有页面左侧文档信息链接样式 |
| .SiteLinkBold | 所有页面上方导航按钮文字链接样式 |
| .MenuUnselected | 所有页面左侧导航栏菜单未选中项目样式 |
| .Menuselected | 所有页面左侧导航栏菜单选中项目样式 |

除了上表中的样式之外，`ASPNETCommerce.css` 还提供了部分供第三方开发人员在对系统进行二次开发时使用的样式。`ASPNETCommerce.css` 还对下列 HTML 标签的样式进行了定义。

- <body>
- <a>
- <small>
- <big>
- <blockquote>
- <pre>
- <hr>

在引用了 `ASPNETCommerce.css` 的页面中，以上 HTML 标签无须特殊说明都按照 `ASPNETCommerce.css` 定义的页面样式显示。

有了以上对于页面样式的定义，开发人员就可以很方便地更改整个站点的风格，在更改的过程中也能保持众多使用同一个样式页面的一致性。

4.2 Global.asax

ASP.NET 和以前的 ASP 一样，支持一个通用于所有 Web 应用程序的全局文件，该全局文件可以作用于某个 Web 应用程序的全局事件、全局对象以及全局变量。在 ASP 中这个文件的名字为 `Global.asa`，而在 ASP.NET 中这个文件改成了 `Global.asax`。

注意：`Global.asax` 的作用并不是配置 Web 应用程序，而是作为 Web 应用程序的一部分。`Global.asax` 和 `Web.Config` 的最大区别是 `Global.asax` 中包含的代码是可以被执行的，为 Web 应用程序的某个功能提供具体的实现。而 `Web.Config` 仅仅包含了 Web 应用程序的配置信息和参数等。和 Web 应用程序中的普通 `ASPX` 文件不同的是，`Global.asax` 文件是不能被浏览器直接访问的。

4.2.1 Global.asax 的结构和作用

在使用 Visual Studio .NET 创建 Web 应用程序的时候，在新创建的 Web 应用程序主目录下会自动生成一个 Global.asax 文件，Global.asax 和普通的 ASPX 页面一样同样支持代码后置。Global.asax 为开发人员提供了一个可以直接操作 Web 应用程序生存期或用户和 Web 应用程序交互过程的 4 种方法。

- 用户状态 (Session)。Session 可以在一定时间内为每一个用户保留数据，这些被保留的数据是针对于每一个提出请求的用户，不同用户之间的数据并不会被共享。
- 应用程序状态 (Application)。Application 可以为整个 Web 应用程序保留数据，Web 应用程序中的 ASPX 页面和 Web 服务都可以由 Application 进行全局设置。
- 暂时应用程序状态(Cache)。Cache 又称为高速缓存，其功能与 Application 极为相似。但是 Cache 包含了相关、回调和超时等 Application 不具有的功能。
- 静态变量。声明静态变量之后，无论创建多少个类的实例，都只是创建了静态变量的一个副本。因此，静态变量常常被用来存储经常会被多处调用的常量。

ASP.NET 支持基于以上方法的 18 种应用程序事件。当然，开发人员也开发可以自定义的事件。

4.2.2 使用 Global.asax

在 ASP.NET Commerce Starter Kit 的 Global.asax 中，有基于 Application_BeginRequest 事件的代码。

```
using System;
using System.Collections;
using System.ComponentModel;
using System.Web;
using System.Security;
using System.Security.Principal;
using System.Web.Security;
using System.Data.SqlClient;
using System.Threading;
using System.Globalization;

namespace ASPNET.StarterKit.Commerce
{
    public class Global : System.Web.HttpApplication
    {
        protected void Application_BeginRequest(Object sender, EventArgs e)
        {
            try
            {
                if (Request.UserLanguages != null)
                    Thread.CurrentThread.CurrentCulture = CultureInfo.CreateSpecificCulture(Request.UserLanguages[0]);
                else
                    Thread.CurrentThread.CurrentCulture = new CultureInfo("en-us");
            }
            Thread.CurrentThread.CurrentUICulture = Thread.CurrentThread.CurrentCulture;
        }
    }
}
```

```

        }
        catch (Exception ex)
        {
            Thread.CurrentThread.CurrentCulture = new CultureInfo("en-us");
        }

    }

}
}

```

Application_BeginRequest 事件是在 ASP.NET 处理的每一个请求上引发的，请求的可以是一个 Web 页面或 Web 服务。使用 Application_BeginRequest 事件可以在页面、Web 服务或其他 HTTP 目标源收到请求的时候执行预处理代码。在上面的程序中，针对每一个请求进程判断其客户端使用的语言，如果可以得到客户端的默认语言，那么该请求进程将使用该语言；如果没有得到默认语言或是出现异常，则使用 en-us（英语）作为该请求进程的默认语言。

Request 对象的 UserLanguages 属性是用来获得客户端语言首选项的排序字符串数组。CurrentThread 的 CurrentCulture 属性是用来设置当前进程的区域性，将取得的首选语言项赋给当前的进程。如果没有得到 UserLanguages 属性中的语言首选项就将当前进程的区域性首选语言设置为 en-us。

4.3 用 户 控 件

用户控件 (User Controls) 是可复用的代码或内容创建为单独的 ASP.NET 控件，在其他需要的页面中调用控件，甚至可以将一些业务逻辑封装在这些控件中。用户控件的开发模式和正常的 ASP.NET 页面的开发模式几乎是一样的。因为用户控件是支持复用的，所以在 Web 应用程序中一些在页面中被多次调用或具有同样功能的模块和区域常常被封装成用户控件，这样做的好处就是减少重复繁琐的代码，对系统进行更加模块化的封装。

在 ASP.NET Commerce Starter Kit 中，使用了 5 个用户控件，见表 4-2。

表 4-2 ASP.NET Commerce Starter Kit 中的用户控件简介

| 控件文件名 | 控 件 作 用 |
|--------------------|----------------------------|
| _Header.ascx | 页面顶端用户控件 |
| _Menu.ascx | 页面左侧菜单用户控件 |
| _AlsoBought.ascx | 商品页面中购买该商品的用户购买其他商品的信息用户控件 |
| _PopularItems.ascx | 热门商品信息用户控件 |
| _ReviewList.ascx | 用户关注商品信息用户控件 |

注意：由于 _AlsoBought.ascx、_PopularItems.ascx、_ReviewList.ascx 涉及到具体的业务逻辑，因此将在后面的章节中讲解。本节中只讲解 _Header.ascx、_Menu.ascx 这两个用户控件。

4.3.1 _Header.ascx

对于很多Web应用程序，都要一个统一的导航栏作为不同功能模块之间的连接桥梁。而这个导航栏在不同页面中的放置位置和外观样式都是一样的，所以把这个导航栏作为一个用户控件来封装就显得十分的必要。在ASP.NET Commerce Starter Kit中的系统设计者就这样做的。如图4-1所示。



图4-1 _Header.ascx

这个用户控件中包括了站点的名称，以及4个导航按钮Sign In、Account、Cart、Services和快速搜索，方便了用户无论在哪个页面都能快速在这4个功能之间切换和快速搜索自己需要的商品。这个用户控件被应用在ASP.NET Commerce Starter Kit所有页面的顶端。下面是_Header.ascx的代码清单。

```
<%@ Control CodeBehind="_Header.ascx.cs" Language="c#" AutoEventWireup="false" Inherits="ASPNSTarterKit.Commerce.C_Header" %>
<table cellspacing="0" cellpadding="0" width="100%" border="0">
<tr>
<td colspan="2" background="images/grid_background.gif" nowrap>
<table cellspacing="0" cellpadding="0" width="100%" border="0">
<tr>
<td colspan="2">

</td>
<td align="right" nowrap>
<table cellpadding="0" cellspacing="0" border="0">
<tr valign="top">
<td align="center" width="65">
<a href="Login.aspx" class="SiteLinkBold">
Sign In</a>
</td>
<td align="center" width="75">
<a href="OrderList.aspx" class="SiteLinkBold">
Account</a>
</td>
<td align="center" width="55">
<a href="ShoppingCart.aspx" class="SiteLinkBold">
Cart</a>
</td>
<td align="center" width="65">
<a href="InstantOrder.asmx" class="SiteLinkBold">
Services</a>
</td>
</tr>
</table>
</td>
<td width="10">
 
</td>
</tr>
```

```
</table>
</td>
</tr>
<tr>
    <td colspan="2" nowrap>
        <form method="post" action="SearchResults.aspx" id="frmSearch" name="frmSearch">
            <table cellspacing="0" cellpadding="0" width="100%" border="0">
                <tr bgcolor="#9D0000">
                    <td background="images/modernliving_bkgrd.gif">
                        
                    </td>
                    <td width="94" align="right" bgcolor="#9D0000">
                        
                    </td>
                    <td width="120" align="right" bgcolor="#9D0000">
                        <input type="text" name="txtSearch" ID="txtSearch" SIZE="20">
                    </td>
                    <td align="left" bgcolor="#9D0000">
                        &nbsp;<input type="image" src="images/arrowbutton.gif" border="0" id="image1" name="image1"> &nbsp;
                    </td>
                </tr>
            </table>
        </form>
    </td>
</tr>
</table>
```

通过上面的代码可以看出，用户控件的编写模式和普通的 ASP.NET 页面在很多地方都是相同的。而且用户控件同样支持代码后置。页面的第一句声明了后台编码类、使用语言、以及类名等属性。

```
<%@ Control CodeBehind="_Header.ascx.cs" Language="c#" AutoEventWireup="false" Inherits="ASPNET.StarterKit.Commerce.C_Header" %>
```

由于 _Header.ascx 并没有实现前台页面和后台编码类的逻辑交互，在此就不详细说明其后台编码类。

4.3.2 _Menu.ascx

在 ASP.NET Commerce Starter Kit 的所有页面左侧都包含了一个商品分类的导航栏，列出了所有商品的分类。作为一个独立的模块，ASP.NET Commerce Starter Kit 的设计者将这个商品分类的导航栏也设计成了一个通用的用户控件。如图 4-2 所示。



图 4-2 _Menu.aspx

下面是_Menu.aspx 页面的代码清单。

```
<%@ OutputCache Duration="3600" VaryByParam="selection" %>
<%@ Control Language="c#" CodeBehind="_Menu.aspx.cs" AutoEventWireup="false" Inherits="ASPNET.
StarterKit.Commerce.C_Menu" %>


|                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                              |  |
|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--|
| <a href="default.aspx"></a>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |  |
| <asp:datalist cellpadding="3" cellspacing="0" id="MyList" runat="server" width="145"> SelectedItemStyle- BackColor="dimgray" EnableViewState="false"&gt; &lt;ItemTemplate&gt;     &lt;asp:HyperLink cssclass="MenuUnselected" id="HyperLink1" Text='&lt;%# DataBinder. Eval(Container.DataItem, "CategoryName") %&gt;' NavigateUrl='&lt;%# "productslist.aspx?CategoryID=" + DataBinder. Eval(Container.DataItem, "CategoryID") + "&amp;selection=" + Container.ItemIndex %&gt;' runat="server" /&gt; &lt;/ItemTemplate&gt; &lt;SelectedItemTemplate&gt;     &lt;asp:HyperLink cssclass="MenuSelected" id="HyperLink2" Text='&lt;%# DataBinder. Eval(Container.DataItem, "CategoryName") %&gt;' NavigateUrl='&lt;%# "productslist.aspx?CategoryID=" + DataBinder. Eval(Container.DataItem, "CategoryID") + "&amp;selection=" + Container.ItemIndex %&gt;' runat="server" /&gt; &lt;/SelectedItemTemplate&gt; </asp:datalist> |  |
| &ampnbsp                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                     |  |


```

```
<br>
<br>
<br>
<br>
<br>
<br>
<a href="docs/docs.htm" target="_blank" class="SiteLink">Commerce Starter Kit<br>
Documentation</a>
</td>
</tr>
</table>
```

这个页面主要是由一个 ID 为 MyList 的 DataList 构成，该 DataList 为商品分类的列表。以下则是该用户控件的后台编码类 `_Menu.aspx.cs` 的代码清单。

```
using System;
using System.Collections;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Web;
using System.Web.UI;
using System.Web.UI.WebControls;
using System.Web.UI.HtmlControls;

namespace ASPNET.StarterKit.Commerce {

    public abstract class C_Menu : System.Web.UI.UserControl {

        protected System.Web.UI.WebControls.DataList MyList;

        public C_Menu() {
            this.Init += new System.EventHandler(Page_Init);
        }

        private void Page_Load(object sender, System.EventArgs e) {

            String selectionId = Request.Params["selection"];

            if (selectionId != null) {
                MyList.SelectedIndex = Int32.Parse(selectionId);
            }
        }

        ASPNET.StarterKit.Commerce.ProductsDB products = new ASPNET.StarterKit.Commerce.ProductsDB();

        MyList.DataSource = products.GetProductCategories();
        MyList.DataBind();
    }

    private void Page_Init(object sender, EventArgs e) {
```

```
        InitializeComponent();  
    }  
  
    private void InitializeComponent() {  
        this.Load += new System.EventHandler(this.Page_Load);  
    }  
  
}
```

通过以上的代码可以看到，用户控件的后台编码类和普通页面的后台编码类极为类似。只不过普通页面的后台编码类继承了 System.Web.UI.Page 类，而用户控件继承了 System.Web.UI.UserControl。在页面的载入过程中，首先取得 URL 中的参数 selection，并将该参数设置为 myList 的选中项值。然后实例化一个对象 products 并调用其中的 GetProductCategories 方法返回一个数据集，为 myList 来设置数据源，最后完成数据绑定。

需要说明的是在该用户控件前台页面中有如下的代码。

```
<%@ OutputCache Duration="3600" VaryByParam="selection" %>
```

这段代码的目的是设定页面的高速缓存。在动态网页设计中，客户端每次对服务器发送请求，服务器都要根据客户端发送的请求经过逻辑程序的处理之后将处理的结果输出到客户端。但实际上对于单个用户控件在某个客户端的多次请求中并没有发生变化，这种情况下就不需要过多地浪费服务器的资源进行对用户控件的重新处理。对于这个问题，一个比较好的解决办法就是使用高速缓存。当然，只有对该用户控件的请求相同时，服务器端才不会重新组织页面，一旦客户端对该用户控件的某一次请求和上一次请求发生变化的时候，服务器端还是会该用户控件进行重新处理，并将处理结果发送到客户端。使用高速缓存大大地提升了页面的执行效率，避免了服务器处理一次又一次的重复请求。

输出高速缓存需要在页面的顶部添加@OutputCache 属性，并为该属性设置两个参数值，分别是 Duration 和 VaryByParam。其中 Duration 参数的值是页面结果需要进行高速缓存的时间，其单位为秒，VaryByParam 是高速缓存改变所依赖的查询字符串。在上面代码中，将页面的高速缓存时间设置为 3600 秒即一小时。将 VaryByParam 设置为 selection，即如果页面的查询字符串中参数 selection 发生变化的话，服务器将不使用高速缓存而是重新载入该页面。

注意：如果页面的高速缓存不依赖查询字符串，那么将 VaryByParam 的值设置为*。如果设置该页面使用相同的高速缓存，那么将 VaryByParam 的值设置为 none。

4.3.3 在页面中调用用户控件

用户控件需要被页面调用才能实现其作用。在页面中调用用户控件非常方便，首先在页面的顶部用 Register 方法进行调用。

```
<%@ Register TagPrefix="ASNETCommerce" TagName="Menu" Src="_Menu.ascx" %>  
<%@ Register TagPrefix="ASNETCommerce" TagName="Header" Src="_Header.ascx" %>
```

TagPrefix 为用户控件确定唯一的命名空间，TagName 为用户控件确定唯一的名字，Src 为所使用的用户控件文件名称和路径。在需要使用用户控件的页面部位调用代码如下。

```
<ASNETCommerce:Header ID="Header1" runat="server" />
```

在调用中，需要使用 TagPrefix:TagName 的格式作为标签的声明，并为用户控件指定一个唯一的 ID。

4.4 小结

本章介绍的主要是系统的基础 UI 开发，需要为 Web 应用程序建议一个清晰的、一致的 UI 基础，使得 Web 应用程序在后续的设计中变得更加有组织，也为后续的开发提供较好的扩展空间。希望读者通过本章学习能够理解以下内容。

- 级联样式表在 Web 应用程序中的应用与整体设计。
- Web 应用程序中 Global.asax 的作用。
- 简单用户控件的设计与开发。

第 5 章

用户体系

作为一个简单的电子商务模型，ASP.NET Commerce Starter Kit 必不可少的是要建立一个用户体系，用户是整个电子商务交易过程中的主角，所以用户管理这部分也是整个系统中最基础的一个应用模块。

ASP.NET Commerce Starter Kit 的用户体系主要包括用户

登录、用户注册两个模块。本章将对这两个模块做详细的剖析。因为只是一个基础模型，所以 ASP.NET Commerce Starter Kit 并没有提供关于用户管理方面的功能，但是这并不影响电子商务模型的正常运作。下面的表 5-1 是本章的知识点索引。

表 5-1 本章知识点索引

| 知 识 点 | 位 置 |
|---------------------|------------------|
| 存储过程设计 | 5.2.3 节, 5.3.3 节 |
| 数据访问层设计 | 5.2.4 节 |
| 用户类设计、返回用户详细信息方法的设计 | 5.3.4 节 |
| 用户注册业务逻辑设计 | 5.2.5 节 |
| 用户注册页面设计 | 5.2.6 节 |
| ASP.NET 页面验证控件详细说明 | 5.2.6 节 |
| 用户登录业务逻辑设计 | 5.3.5 节 |
| 用户登录页面设计 | 5.3.6 节 |
| 创建用户身份验证票 | 5.2.5 节 |
| 加密数据传输 | 5.3.5 节 |

5.1 系统设计

5.1.1 需求分析

用户体系对于电子商务模型来说是一套完整的体系，当一个电子商务系统的浏览者想真正地使用电子商务系统购买商品的时候，就必须注册成为系统的用户，提供一个系统中唯一的用户名，并且提供一个对应的登录系统的密码。图 5-1 描述了 ASP.NET Commerce Starter Kit 用户体系中的一些主要的环节和行为。

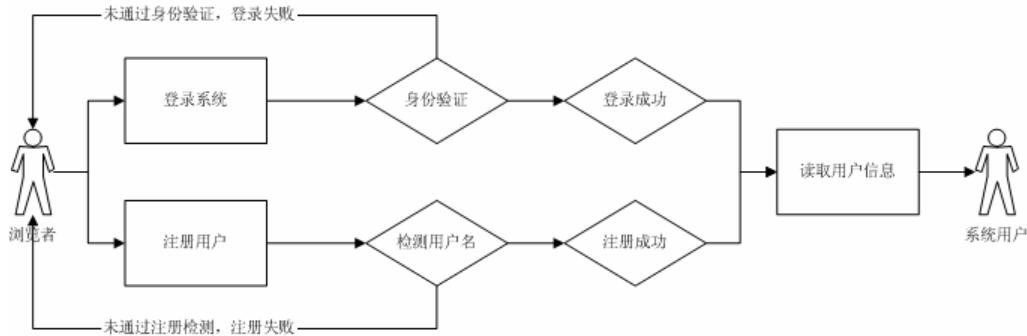


图 5-1 ASP.NET Commerce Starter Kit 用户体系

用户体系中的两个主要角色是浏览器和系统用户，浏览器可以通过注册获得登录的用户名，登录系统成为系统用户。系统用户对于系统来讲是拥有唯一标识的浏览器。系统之所以要求系统用户具有唯一标识是因为在电子商务交易中，需要记录和区别购买者的详细信息，为后续的配送、发货和收款等环节提供完整的用户信息。用户体系中的主要元素信息如表 5-2 所示。

表 5-2 用户体系中的元素

| 元素名称 | 元素类型 | 元素说明 |
|--------|------|-----------------------------|
| 浏览器 | 角色 | 匿名访问系统的用户，没有系统的唯一标识 |
| 系统用户 | 角色 | 通过系统身份验证的用户，具有系统的唯一标识 |
| 登录系统 | 过程 | 浏览器输入用户密码通过系统验证成为系统用户的过程 |
| 注册用户 | 过程 | 浏览器获取系统唯一标识成为系统用户的过程 |
| 读取用户信息 | 过程 | 获取通过系统验证用户的信息的过程 |
| 身份验证 | 行为 | 验证浏览器输入的 Email 和密码是否匹配 |
| 登录成功 | 行为 | 浏览器输入的 Email 和密码匹配，用户通过身份验证 |
| 检测用户名 | 行为 | 检测浏览器在注册过程中使用 Email 是否唯一 |
| 注册成功 | 行为 | 浏览器输入的用户资料成功通过注册 |

用户体系中两个进程分别为注册和登录，这两个进程也是浏览器成为系统用户所必需的。浏览器通过注册获得唯一标识的用户名，通过登录来让系统识别自己的唯一标识的用户名。

注意：作为一个电子商务应用，用户的身份验证和安全是十分关键和重要的，因此在密码的存储操作等方面显得尤为重要。在真正的电子商务应用中，用户的密码需要高度加密存储，用户的身份验证应该非常显著和强壮。但是在 ASP.NET Commerce Starter Kit 中，作为演示使用的简单电子商务系统并没有考虑这方面问题，在真正的商业应用中这样是远远不够的。

5.1.2 功能设计

根据模块的实际需求，用户体系主要由用户注册和用户登录两大功能构成。

(1) 用户注册。浏览器在用户注册页面按照系统的提示，输入全名、Email、密码和确认密码之后，系统会检测浏览器输入的 Email 在目前的用户数据库是否存在，如果已经存在提示注册失败，因此系统认为该用户已经存在。如果该 Email 并没有存在于目前的用户数据库中，则该 Email 可以被用户注册。

注意：和很多应用系统不一样，ASP.NET Commerce Starter Kit 把 Email 作为用户的惟一标识来区别，而并不是用户名。实际上用户的标识最主要的作用还是为了让系统区分用户，因此无论是使用用户名作为惟一标识还是使用 Email 作为惟一标识，对于系统的业务逻辑是没有影响的。

(2) 用户登录。浏览者在用户登录页面输入 Email 和密码，系统在当前用户数据库中寻找该用户名，如果在用户数据库中没有发现该用户名或是用户数据库中的密码和当前浏览器输入密码不匹配，则提示用户登录失败。反之，用户登录成功并重新定向到用户请求的页面。

注意：当浏览者在没有登录系统的情况下试图完成订单或是查看用户信息的时候，系统会自动定向到用户登录的页面，提示用户登录。而用户在完成登录之后，系统将重新将页面定向到用户登录之前请求的页面。

5.1.3 数据库设计

根据 ASP.NET Commerce Starter Kit 的用户体系实际需求，需要一个存储用户信息的数据表。此表名为 CMRC_Customers，具体数据信息如图 5-2 所示。其中 CustomerID 为该表的主键。

| | 列名 | 数据类型 | 长度 | 允许空 |
|--|--------------|----------|----|-----|
| | CustomerID | int | 4 | |
| | FullName | nvarchar | 50 | ✓ |
| | EmailAddress | nvarchar | 50 | ✓ |
| | Password | nvarchar | 50 | ✓ |

图 5-2 CMRC_Customers 表结构

注意：因为 ASP.NET Commerce Starter Kit 本身并不是一个完整的电子商务应用，ASP.NET Commerce Starter Kit 的设计者之所以采用这种简约的数据结构，目的是为了可以灵活地在 ASP.NET Commerce Starter Kit 之上进行扩展和二次开发。尽量避免增加没有必要的字段，使系统在设计结构上的扩展性和兼容性变得更好。

5.2 用 户 注 册

5.2.1 实现效果

用户注册页面如图 5-3 所示。

浏览者在用户注册页面 register.aspx 按照提示输入全名、Email、密码和确认密码。当以下情况出现时会提示错误，错误页面如图 5-4 所示。

- 全名的开头为空格。
- Email 的开头为空格。
- Email 的格式不正确（不含有@和. 符号）。
- 密码的开头为空格。
- 确认密码的开头为空格。
- 密码和确认密码不一样。

如果不出现以上情况，则页面被提交。但是并不是所有提交的页面都可以通过注册。在后台处理程序中，要确认目前提交注册的 Email 在系统中不存在，如果存在会返回一条错误信息，提示错误，不能通过

注册，提示如图 5-5 所示；如果 Email 不存在，则系统将通过该注册，浏览者将拥有惟一标识的用户名并被重新定向到购物车页面。

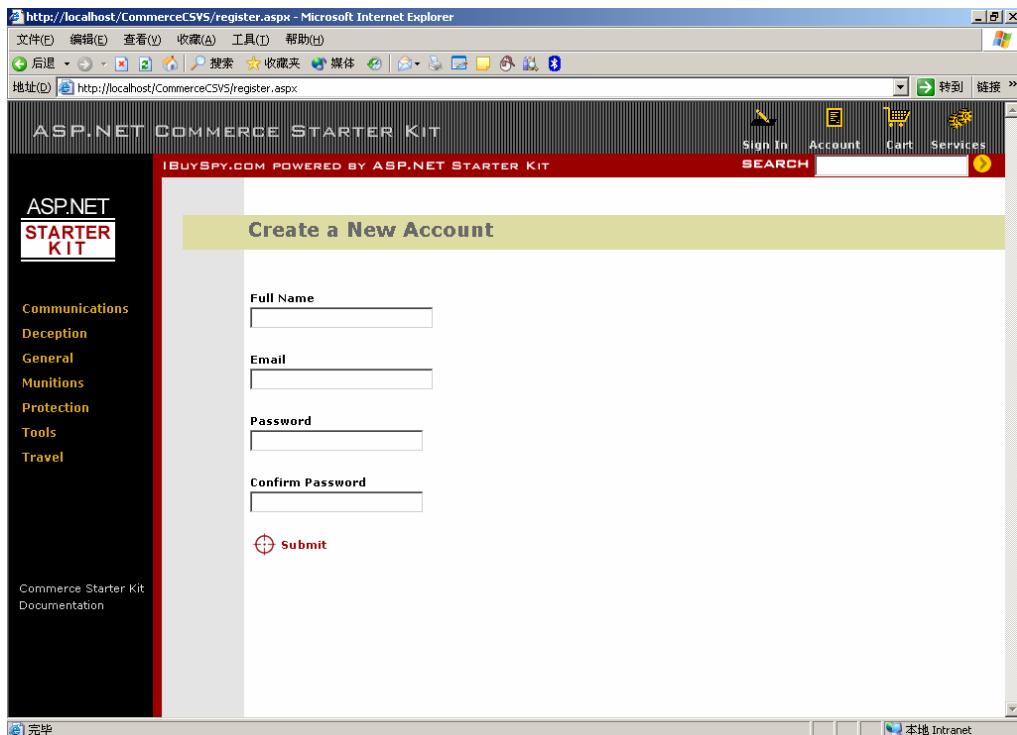


图 5-3 用户注册页面

A screenshot of the 'Create a New Account' page. The 'Email' field contains 'ss' and has a red error message: 'Must use a valid email address.'. The 'Password' field contains '****' and the 'Confirm Password' field contains '***', both of which have red error messages: 'Password fields do not match.' The 'Submit' button is visible at the bottom.

图 5-4 用户注册错误提示

A screenshot of the 'Create a New Account' page. A red error message at the bottom of the form states: 'Registration failed: That email address is already registered.'

图 5-5 用户注册不成功

5.2.2 关键技术点

用户注册模块有两个值得关注的技术点。

(1) 使用服务器的验证控件验证表单。

创建和用户交互的 Web 表单就需要获得用户输入控件中的内容，并对其进行必要的验证。为了避免因为错误的输入浪费服务器资源，验证逻辑往往被安排运行于客户端。多数情况下需要使用 JavaScript 来编写客户端的代码，但是 JavaScript 复杂的语法容易引发不同浏览器下的兼容性错误，而且很多情况下客户端代码需要实现的功能都是类似的，而 ASP.NET 中的页面验证控件提供了一个能够很好解决以上问题的方法。

所有的验证控件都是从基类 BaseValidator 中继承而来的，BaseValidator 是命名空间 System.Web.UI.WebControls.BaseValidator 中的成员，所有的验证控件都具有一些相同的属性和方法。如表 5-3 所示。除了所有验证控件共有的一些属性，每一个验证控件还具有其特有的属性和方法。

表 5-3 ASP.NET 中页面验证控件共有的属性

| 属性名 | 属性说明 |
|--------------------|--------------------------|
| ControlToValidate | 设置或返回验证控件的名称 |
| EnableClientScript | 设置或返回一个布尔值，以确定是否允许客户端验证 |
| Enable | 设置或返回一个布尔值，以确定该验证控件是否有效 |
| ErrorMessage | 设置或返回未通过当前验证控件验证时显示的错误信息 |
| IsValid | 返回一个布尔值，以确定是否通过该验证控件的验证 |

除了使用 ASP.NET 提供具有固定功能的页面验证控件之外，开发者可以使用 CustomValidator 控件根据实际需要开发出自己需要的页面验证控件，这极大地丰富了验证控件的功能。

(2) 在用户注册完毕后要创建用户身份验证票据并将其附加到 Cookie 的传出响应集合中。

在 ASP.NET 中可以使用基于表单的验证（也可以称之为基于 Cookie 的验证）。这种验证方法的原理是在 IIS 中接收用户的请求，当用户访问应用程序的时候，根据传入的 Cookie 判断用户是否通过验证，如果通过验证，Cookie 包含了他们的身份信息，ASP.NET 以此来判断当前用户是否具有权限访问所请求的资源。如果应用程序没有检查到相应的 Cookie，则拒绝用户的资源请求。

在 ASP.NET 中，使用基于表单的验证方法需要调用 System.Web.Security 命名空间下的类 FormsAuthentication。表 5-4 是该类下面的主要方法及说明。

表 5-4 System.Web.Security.FormsAuthentication 的主要方法及说明

| 方法名 | 说明 |
|-----------------------|---|
| Authenticate | 检查需要验证的用户名和密码与 Web.Config 中定义的是否匹配 |
| RedirectFromLoginPage | 验证用户之后所要执行的动作，添加 Cookie 到请求中，并重新定向到用户初始请求页面 |
| SignOut | 当前加密的 cookie |
| SetAuthCookie | 和 RedirectFromLoginPage 方法功能类似，但是不重新定向页面 |
| GetAuthCookie | 返回验证 Cookie |
| GetRedirectUrl | 在被重新定向到注册页面之前返回用户请求的初始页 |

5.2.3 存储过程

用户注册使用了存储过程 CMRC_CustomerAdd，以下是该存储过程。该存储过程向数据表

CMRC_Customers 添加了一条记录，并将添加记录的 CustomerID 返回。

```

CREATE Procedure CMRC_CustomerAdd
(
    @FullName      nvarchar(50),
    @Email         nvarchar(50),
    @Password      nvarchar(50),
    @CustomerID   int OUTPUT
)
AS
--向 CMRC_Customers 数据表中插入一条记录
INSERT INTO CMRC_Customers
(
    FullName,
    EmailAddress,
    Password
)

VALUES
(
    @FullName,
    @Email,
    @Password
)
--获取刚刚插入记录的 CustomerID 字段作为返回值
SELECT
    @CustomerID = @@Identity
GO

```

5.2.4 数据访问层

在 Components 文件夹下的 CustomersDB.cs 文件中提供了用户注册过程的数据访问层。类 CustomersDB 提供了一个添加用户的方法 CustomerAdd，CustomersDB 类属于命名空间 ASPNET.StarterKit.Commerce。该方法返回一个字符串，如果添加用户成功则返回添加用户的 customerId，如果注册不成功则返回空值。

首先是声明 AddCustomer 为返回 string 类型值的方法，并列出该方法的参数表。

```

public String AddCustomer(string fullName, string email, string password)
{
    //取得 Web.Config 中字符串 ConnectionString 的值，实例化数据库连接
    SqlConnection          myConnection           = new SqlConnection(ConfigurationSettings.AppSettings["ConnectionString"]);
    //利用建立的数据库连接建立 SqlCommand，执行名称为 CMRC_CustomerAdd 的存储过程
    SqlCommand  myCommand = new SqlCommand("CMRC_CustomerAdd", myConnection);
    //设置 SqlCommand 的模式为执行存储过程
    myCommand.CommandType = CommandType.StoredProcedure;

```

为将要执行的存储过程添加参数。包括定义每一个参数的名称、类型、大小、数据来源。

```

//定义参数的名称、数据类型、大小
SqlParameter parameterFullName = new SqlParameter("@FullName", SqlDbType.NVarChar, 50);
//定义参数的数据来源

```

```

parameterFullName.Value = fullName;
//将参数添加给存储过程
myCommand.Parameters.Add(parameterFullName);

SqlParameter parameterEmail = new SqlParameter("@Email", SqlDbType.NVarChar, 50);
parameterEmail.Value = email;
myCommand.Parameters.Add(parameterEmail);

SqlParameter parameterPassword = new SqlParameter("@Password", SqlDbType.NVarChar, 50);
parameterPassword.Value = password;
myCommand.Parameters.Add(parameterPassword);

SqlParameter parameterCustomerID = new SqlParameter("@CustomerID", SqlDbType.Int, 4);
//定义该参数为返回值
parameterCustomerID.Direction = ParameterDirection.Output;
myCommand.Parameters.Add(parameterCustomerID);

```

打开数据连接并执行命令，然后关闭连接。获得存储过程的返回值，将该值作为方法的返回值返回。如果发生异常，则返回空字符串。

```

try {
    //打开数据库连接
    myConnection.Open();
    //执行 SqlCommand
    myCommand.ExecuteNonQuery();
    //关闭数据库连接
    myConnection.Close();
    //获得存储过程返回的用户 customerId
    int customerId = (int)parameterCustomerID.Value;
    //将 customerId 返回
    return customerId.ToString();
}
//捕获异常
catch {
    //当发生异常返回一个空字符串
    return String.Empty;
}

```

5.2.5 业务逻辑层

在前台页面 Register.aspx 的后台编码类中，对于页面输入的值做了简单的逻辑处理之后调用了上一小节的方法 CustomerAdd。下面是 Register.aspx.cs 的代码清单。

```

using System;
using System.Collections;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Web;
using System.Web.Security;
using System.Web.UI;

```

```
using System.Web.UI.WebControls;
using System.Web.UI.HtmlControls;

namespace ASPNET.StarterKit.Commerce {

    public class Register : System.Web.UI.Page {
        //定义Register.aspx页面中使用的服务器端控件，其中包括文本输入框、提交按钮和验证控件
        protected System.Web.UI.WebControls.TextBox Name;
        protected System.Web.UI.WebControls.RequiredFieldValidator RequiredFieldValidator1;
        protected System.Web.UI.WebControls.TextBox Email;
        protected System.Web.UI.WebControls.RegularExpressionValidator RegularExpressionValidator1;
        protected System.Web.UI.WebControls.RequiredFieldValidator RequiredFieldValidator2;
        protected System.Web.UI.WebControls.TextBox Password;
        protected System.Web.UI.WebControls.RequiredFieldValidator RequiredFieldValidator3;
        protected System.Web.UI.WebControls.TextBox ConfirmPassword;
        protected System.Web.UI.WebControls.RequiredFieldValidator RequiredFieldValidator4;
        protected System.Web.UI.WebControls.CompareValidator CompareValidator1;
        protected System.Web.UI.WebControls.Label MyError;
        protected System.Web.UI.WebControls.ImageButton RegisterBtn;

        public Register() {
            Page.Init += new System.EventHandler(Page_Init);
        }
        //定义按钮RegisterBtn单击事件，当输入值通过页面验证控件的验证之后才触发添加用户的业务逻辑
        private void RegisterBtn_Click(object sender, System.Web.UI.ImageClickEventArgs e) {
            //判断页面是否通过验证
            if (Page.IsValid == true) {

```

实例化购物车类 ASPNET.StarterKit.Commerce.ShoppingCartDB 为对象 shoppingCart，并使用 shoppingCart 的方法 GetShoppingCartId 获得临时的购物车标识 tempCartId。

注意：此处是针对在没有注册的情况下使用购物车的浏览者。当没有注册的浏览者在浏览商品列表的时候将商品添加到购物车，系统将为此生成一个临时的购物车对象，保存购物车的信息，当浏览者决定确认购买目前购物车的商品的时候，系统将提示浏览者登录或是注册。当浏览者选择登录或是注册之后，被存储在临时购物车中的商品信息还将被保留，方便系统用户购买该商品。以上过程是通过类 ASPNET.StarterKit.Commerce.ShoppingCartDB 实例化之后，将临时购物车的信息存储在类实例化的对象中实现的。

```
//实例化类 ShoppingCartDB
ASPNET.StarterKit.Commerce.ShoppingCartDB shoppingCart = new ASPNET.StarterKit.Commerce.
ShoppingCartDB();
//获得临时购物车的tempCartId
String tempCartId = shoppingCart.GetShoppingCartId();
```

实例化类 ASPNET.StarterKit.Commerce.CustomersDB 为对象 accountSystem，将用户填写的注册信息以参数的形式传给方法 AddCustomer，并获得返回值 customerId。在上一小节数据访问层中详细分析过 AddCustomer，该方法返回的值为注册用户的 customerId，如果返回值为空值则说明程序在运行过程中出现异常，用户并没有通过注册。customerId 不为空则说明已经通过注册，调用 FormsAuthentication.SetAuthCookie 方法把 customerId 存放在 Cookie 中。System.Web.Security.FormsAuthentication 类是.NET Framework 提供身份验证票据的一个类，

SetAuthCookie 方法是创建用户身份验证票据并将其附加到 Cookie 的传出响应集合中的一个方法。

```
//实例化类 CustomersDB 成对象 accountSystem
ASPNET.StarterKit.Commerce.CustomersDB accountSystem = new ASPNET.StarterKit.Commerce.
CustomersDB();
//调用 accountSystem 的方法 AddCustomer 向数据库中添加一条用户注册数据，并获得返回的 customerId
String customerId = accountSystem.AddCustomer(Name.Text, Email.Text, ASPNET.StarterKit.
Commerce.Components.Security.Encrypt(Password.Text));
//如果返回的 customerId 不为空则说明用户注册成功
if (customerId != "") {
    //根据 customerId 创建用户身份验证票据并将其附加到 cookie 中
    FormsAuthentication.SetAuthCookie(customerId, false);
```

利用 shoppingCart 类中的方法 MigrateCart 将临时存储的购物车信息转化成注册用户的购物车信息。MigrateCart 是将一个用户的当前购物车信息转移到另外一个用户的当前购物车信息中，在没有注册之前浏览器使用的购物车是一个临时性质的购物车，在通过注册有了 customerId 之后，把原来临时的购物车信息转移成为当前用户的购物车信息。关于 MigrateCart 方法的具体内容将在后面的章节介绍。

```
//调用对象 shoppingCart 的 MigrateCart 方法将临时购物车整合为当前用户的购物车
shoppingCart.MigrateCart(tempCartId, customerId);
```

将注册的用户名写入名为 ASPNETCommerce_FullName 的 Cookies 中，方便在其他功能模块中取用，并将页面重定向到 ShoppingCart.aspx，也就是用户购物车信息页面。至此，一个正常的用户注册流程结束。

```
//设定名字为 ASPNETCommerce_FullName 的 Cookies 的值为用户名
Response.Cookies["ASPNETCommerce_FullName"].Value = Server.HtmlEncode(Name.Text);
//跳转到购物车页面
Response.Redirect("ShoppingCart.aspx");
}
//如果注册流程中出现错误，则显示错误信息
else {
    //将错误信息显示到前台页面中名字为 MyError 的 Label 中
    MyError.Text = "Registration failed: That email address is already
registered.<br> <img align=left height=1 width=92 src=images/lx1.gif>";
}
}
//页面装载事件
private void Page_Load(object sender, System.EventArgs e) {

}
private void Page_Init(object sender, EventArgs e) {
    InitializeComponent();
}
private void InitializeComponent() {
    //定义按钮的单击事件委托
    this.RegisterBtn.Click += new System.Web.UI.ImageClickEventHandler(this.RegisterBtn_Click);
    this.Load += new System.EventHandler(this.Page_Load);

}
```

5.2.6 用户表示层

注册页面 Register.aspx 的前台页面并没有包含过多的逻辑代码，主要原因就是大部分功能都是将业务逻辑通过服务器端控件交给后台的代码来完成，比如说在上一节分析的用户注册的整个过程，所有的逻辑都是在后台编码类中完成的。

前台页面的主要功能是对数据的捕获和验证。页面提供 TextBox 控件来获得访问者输入的信息，这点在实现起来和 HTML 没有太大的区别，只不过这些 TextBox 是运行于服务器端，可以通过后台编码类对其进行调用。在数据的验证方面，通常的办法是采用 JavaScript 脚本语言来完成，但是实现代码零乱，工作比较繁琐。ASP.NET 中提供的验证控件能够很好地完成验证方面的工作。ASP.NET 提供了 6 种页面验证控件，如表 5-5 所示。

表 5-5

ASP.NET 页面验证控件列表

| 验证控件 | 作用 |
|----------------------------------|---|
| <ASP:RequiredFieldValidator> | 检验某一控件的输入值是否为空，必须与其他输入控件共同使用 |
| <ASP:RangeValidator> | 检验某一控件的输入值是否在某一个数值范围内，必须与其他输入控件共同使用 |
| <ASP:CompareValidator> | 比较两个控件的输入值或一个控件的输入值与一特定值是否匹配。数据类型和比较操作可以指定。必须与其他输入控件共同使用 |
| <ASP:RegularExpressionValidator> | 比较某一个控件的输入值与一特定的正则表达式是否匹配。必须与其他输入控件共同使用。正则表达式的写法遵循 System.Text.RegularExpressions.Regex 的规则 |
| <ASP:CustomValidator> | 使用一个开发者自定义的函数对某一个控件的输入值进行验证。该函数可以运行于服务器端或是客户端，该控件必须与其他输入控件共同使用 |
| <ASP:ValidationSummary> | 显示所有当前验证控件提示错误的集合 |

验证控件的使用极大地减少了页面的代码量，使得运行于前台的 ASP.NET 页面更加 HTML 化，在 Register.aspx 中使用了如下的用户控件。

```

<!--检查 Name 文本输入框的值是否为空-->
<asp:RequiredFieldValidator ControlToValidate="Name" Display="dynamic" Font-Name="verdana" Font-Size="9pt"
ErrorMessage="Name' must not be left blank." runat="server" id=RequiredFieldValidator1></asp:RequiredFieldValidator>

<!--检查 Email 文本输入框的值格式是否正确-->
<asp:RegularExpressionValidator ControlToValidate="Email" ValidationExpression="[\w\.-]+(\.[\w-]*\.)?@[\\w-\\.\\.\\.]+[\\w-]" Display="Dynamic" Font-Name="verdana" Font-Size="9pt" ErrorMessage="Must use a valid email address." runat="server" id=RegularExpressionValidator1></asp:RegularExpressionValidator>

<!--检查 Email 文本输入框的值是否为空-->
<asp:RequiredFieldValidator ControlToValidate="Email" Display="dynamic" Font-Name="verdana" Font-Size="9pt"
ErrorMessage="Email' must not be left blank." runat="server" id=RequiredFieldValidator2></asp:RequiredFieldValidator>

<!--检查 Password 文本输入框的值是否为空-->
<asp:RequiredFieldValidator ControlToValidate="Password" Display="dynamic" Font-Name="verdana" Font-Size="9pt" ErrorMessage="Password' must not be left blank." runat="server" id=RequiredFieldValidator3></asp:RequiredFieldValidator>

<!--检查 ConfirmPassword 文本输入框的值是否为空-->
<asp:RequiredFieldValidator ControlToValidate="ConfirmPassword" Display="dynamic" Font-Name="verdana" Font-Size="9pt" ErrorMessage="Confirm' must not be left blank." runat="server" id=RequiredFieldValidator4></asp:
```

```

RequiredFieldValidator>
    <!--检查 Password 和 ConfirmPassword 中的值是否匹配-->
    <asp:CompareValidator ControlToValidate="ConfirmPassword" ControlToCompare="Password" Display="Dynamic"
Font-Name="verdana" Font-Size="9pt" ErrorMessage="Password fields do not match." runat="server" id=CompareValidator1>
</asp:CompareValidator>

```

在页面上使用如上的用户控件之后，在后台编码类中，只要检测 Page. IsValid 的值是否为真就可以判断前台输入的值是否符合要求，如不符合，将在页面上给出每一个不符合要求的错误信息。

5.3 用户登录

5.3.1 实现效果

用户登录页面 Login.aspx 如图 5-6 所示。

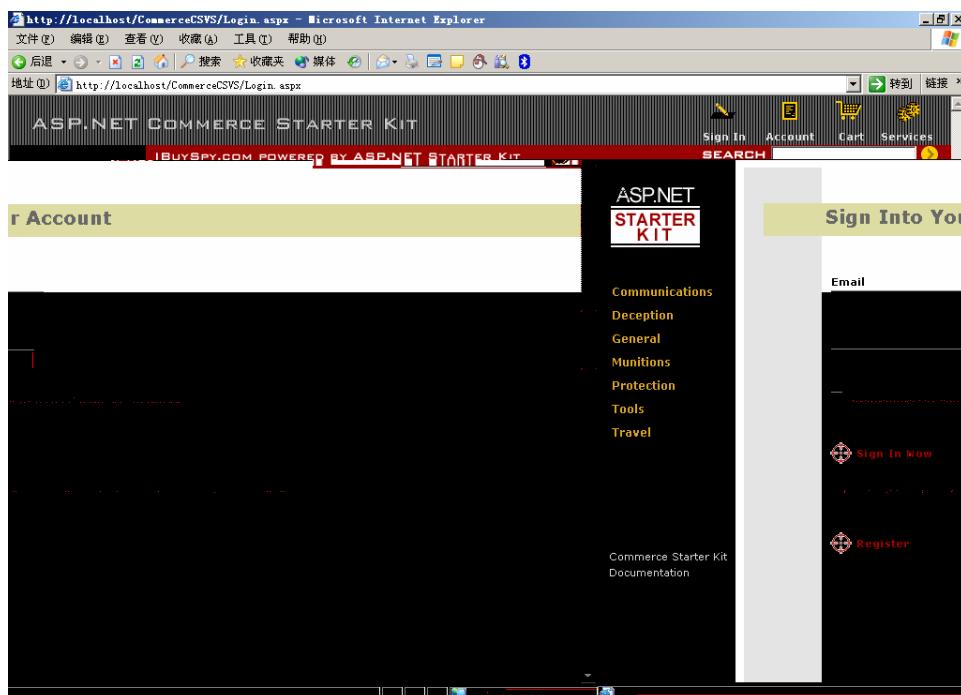


图 5-6 用户登录页面

当浏览器访问 Login.aspx 的时候，输入用户名和密码，并选择是否将用户信息保存在 Cookie 中。如果选择将用户信息保存在 Cookie 中，浏览器在下次访问 Web 应用程序的时候系统将会自动识别用户的身份信息。

当以下情况出现时，会出现页面错误提示信息。

- 用户 Email 为空或格式不正确。
- 用户密码为空或密码不正确。

如果浏览器输入的用户名和密码都是正确的，则登录成功，系统将记录当前的用户登录信息，页面会自动跳转到购物车页面。

5.3.2 关键技术点

用户登录模块所使用的关键技术点主要有以下两个。

(1) 用户类的设计

在用户体系的数据访问层中，设计者构建了一个用于存储用户信息数据的类，利用这个类被实例化之后的对象来设置、存储和传输用户信息。该类的代码如下。

```
public class CustomerDetails {  
    public String FullName;  
    public String Email;  
    public String Password;  
}
```

这是一种典型遵循面向对象方法的设计思想，将用户这个实际元素构建成为对象，将其 FullName、Email 和 Password 抽象为属性，方便用户信息的调用和传输。

(2) 加密数据的传输

在网络应用程序的设计中，部分数据的安全度要求很高，所以这些数据在传输和使用过程中需要加密。.NET Framework 在命名空间 System.Security.Cryptography 下提供了相应的类库和方法。通常的做法是将安全度要求比较高的数据变成二进制的字节型变量，经过加密算法的加密，用于网络中。存储于数据库中的数据都是经过加密之后的数据。因为使用的加密算法都是不可逆的，因此只能够比较数据的匹配性，而不能逆向得到数据。经过这样处理的数据的安全性大大提高了。

5.3.3 存储过程

用户登录模块在逻辑上用到的存储过程可以分为登录验证过程和返回用户详细信息过程两个部分。下面是用户登录验证过程的存储过程 CMRC_CustomerLogin。该存储过程接收传入的 Email 和 Password，验证用户密码的正确性，如果正确，则返回用户的 CustomerID；如果用户没有通过验证，则返回 0。该存储过程完成了用户登录这个过程在数据库层次上面的逻辑应用。

```
CREATE Procedure CMRC_CustomerLogin  
(  
    @Email      nvarchar(50),  
    @Password   nvarchar(50),  
    @CustomerID int OUTPUT  
)  
AS  
--选择 Email 和 Password 为输入参数的用户 CustomerID  
SELECT  
    @CustomerID = CustomerID  
  
FROM  
    CMRC_Customers  
  
WHERE  
    EmailAddress = @Email  
    AND  
    Password = @Password  
--如果没有符合的记录则返回值为 0
```

```
IF @@Rowcount < 1
SELECT
    @CustomerID = 0
```

另外一个存储过程就是返回用户详细信息的存储过程 CMRC_CustomerDetail。

```
CREATE Procedure CMRC_CustomerDetail
(
    @CustomerID int,
    @FullName nvarchar(50) OUTPUT,
    @Email nvarchar(50) OUTPUT,
    @Password nvarchar(50) OUTPUT
)
AS
--根据 CustomerID 选择对应用户的 FullName、EmailAddress 和 Password
SELECT
    @FullName = FullName,
    @Email = EmailAddress,
    @Password = Password

FROM
    CMRC_Customers

WHERE
    CustomerID = @CustomerID
```

该存储过程根据传入参数 CustomerID 返回该 CustomerID 所对应的用户的 Email、FullName、Password 的详细信息。该存储过程除了用于用户登录过程，还用于其他需要根据 CustomerID 返回用户详细信息的逻辑应用。

5.3.4 数据访问层

在上面介绍用户注册的数据访问层中，曾经介绍了用户注册和登录中的数据访问层。在 Components 文件夹下的 CustomersDB.cs 为 ASP.NET Commerce Starter Kit 的用户体系提供了数据访问层。以下是 CustomersDB.cs 的代码清单。

```
using System;
using System.Configuration;
using System.Data;
using System.Data.SqlClient;

namespace ASPNET.StarterKit.Commerce {
    //建立一个 CustomerDetails 类，用于存储用户对象内部信息
    public class CustomerDetails {
        //定义用户的 FullName
        public String FullName;
        //定义用户的 Email
        public String Email;
        //定义用户的 Password
        public String Password;
    }
}
```

```
public class CustomersDB {  
    //根据参数 customerID 实例化上面创建的 CustomerDetails 对象，并返回该对象  
    public CustomerDetails GetCustomerDetails(String customerID)  
    {
```

实例化一个 SqlConnection 连接数据库，并调用 SqlCommand 执行存储过程 CMRC_CustomerDetail，并为该存储过程添加参数，将返回的参数值赋给类 CustomerDetails 实例化之后的对象 myCustomerDetails，最终将这个对象返回。整个过程是一个根据用户 ID 返回用户对象的完整过程。

```
//实例化一个数据库连接对象 myConnection  
SqlConnection myConnection = new SqlConnection(ConfigurationSettings.AppSettings["ConnectionString"]);  
//实例化一个使用 myConnection 打开存储过程 CMRC_CustomerDetail 的数据库命令对象 myCommand  
SqlCommand myCommand = new SqlCommand("CMRC_CustomerDetail", myConnection);  
//定义 myCommand 为打开存储过程的数据库命令  
myCommand.CommandType = CommandType.StoredProcedure;  
//为 myCommand 添加参数 parameterCustomerID  
SqlParameter parameterCustomerID = new SqlParameter("@CustomerID", SqlDbType.Int, 4);  
parameterCustomerID.Value = Int32.Parse(customerID);  
myCommand.Parameters.Add(parameterCustomerID);  
//为 myCommand 添加参数 parameterFullName，并定义参数为返回值型  
SqlParameter parameterFullName = new SqlParameter("@FullName", SqlDbType.NVarChar, 50);  
parameterFullName.Direction = ParameterDirection.Output;  
myCommand.Parameters.Add(parameterFullName);  
//为 myCommand 添加参数 parameterEmail，并定义参数为返回值型  
SqlParameter parameterEmail = new SqlParameter("@Email", SqlDbType.NVarChar, 50);  
parameterEmail.Direction = ParameterDirection.Output;  
myCommand.Parameters.Add(parameterEmail);  
//为 myCommand 添加参数 parameterPassword，并定义参数为返回值型  
SqlParameter parameterPassword = new SqlParameter("@Password", SqlDbType.NVarChar, 50);  
parameterPassword.Direction = ParameterDirection.Output;  
myCommand.Parameters.Add(parameterPassword);  
//打开数据库连接 myConnection  
myConnection.Open();  
//执行数据库命令 myCommand  
myCommand.ExecuteNonQuery();  
//关闭数据库连接 myConnection  
myConnection.Close();  
//实例化 CustomerDetails  
CustomerDetails myCustomerDetails = new CustomerDetails();  
//获得存储过程返回的参数 parameterFullName 的值  
myCustomerDetails.FullName = (string)parameterFullName.Value;  
//获得存储过程返回的参数 parameterPassword 的值  
myCustomerDetails.Password = (string)parameterPassword.Value;  
//获得存储过程返回的参数 parameterEmail 的值  
myCustomerDetails.Email = (string)parameterEmail.Value;  
//返回 myCustomerDetails 对象  
return myCustomerDetails;  
}
```

添加用户的方法，在介绍用户注册的时候已经详细介绍过了，在此处省略其代码。

```
public String AddCustomer(string fullName, string email, string password)  
{
```

```
.....  
}
```

用户登录的方法，根据传入参数 email 和 password，建立数据库连接，执行 CMRC_CustomerLogin 存储过程，判断用户是否能够通过身份验证。如果存储过程返回值为 0，则说明用户没有通过验证；如果返回的是一个有效的 CustomerID，则说明该用户通过验证。

```
public String Login(string email, string password)  
{  
    //实例化一个数据库连接对象 myConnection  
    SqlConnection myConnection = new SqlConnection(ConfigurationSettings.AppSettings["ConnectionString"]);  
    //实例化一个使用 myConnection 打开存储过程 CMRC_CustomerLogin 的数据库命令对象 myCommand  
    SqlCommand myCommand = new SqlCommand("CMRC_CustomerLogin", myConnection);  
    //定义 myCommand 为打开存储过程的数据库命令  
    myCommand.CommandType = CommandType.StoredProcedure;  
    //为 myCommand 添加参数 parameterEmail  
    SqlParameter parameterEmail = new SqlParameter("@Email", SqlDbType.NVarChar, 50);  
    parameterEmail.Value = email;  
    myCommand.Parameters.Add(parameterEmail);  
    //为 myCommand 添加参数 parameterPassword  
    SqlParameter parameterPassword = new SqlParameter("@Password", SqlDbType.NVarChar, 50);  
    parameterPassword.Value = password;  
    myCommand.Parameters.Add(parameterPassword);  
    //为 myCommand 添加参数 parameterCustomerID，并定义参数为返回值型  
    SqlParameter parameterCustomerID = new SqlParameter("@CustomerID", SqlDbType.Int, 4);  
    parameterCustomerID.Direction = ParameterDirection.Output;  
    myCommand.Parameters.Add(parameterCustomerID);  
    //打开数据库连接 myConnection  
    myConnection.Open();  
    //执行数据库命令 myCommand  
    myCommand.ExecuteNonQuery();  
    //关闭数据库连接 myConnection  
    myConnection.Close();  
    //获得存储过程参数 parameterCustomerID 的返回值  
    int customerId = (int)(parameterCustomerID.Value);  
    //判断该返回值是否为 0  
    if (customerId == 0) {  
        //如果为 0 则返回一个空值  
        return null;  
    }  
    else {  
        //如果不为 0 则返回该值  
        return customerId.ToString();  
    }  
}
```

5.3.5 业务逻辑层

用户登录这个逻辑过程在多数 Web 应用程序中都有应用，但是根据系统要求的不同具体的实现也不同。但是其核心目的都是为了验证用户身份，并返回用户信息供用户在具体的系统业务逻辑中使用。在 ASP.NET Commerce Starter Kit 中系统登录实现的功能很简单，就是为了验证用户的身份并通过验证的用户分配详细的信息。下面是 Login.aspx.cs 的后台编码类的代码。

```
using System;
using System.Collections;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Web;
using System.Web.Security;
using System.Web.UI;
using System.Web.UI.WebControls;
using System.Web.UI.HtmlControls;

namespace ASPNET.StarterKit.Commerce {

    public class Login : System.Web.UI.Page {
        //定义页面控件
        protected System.Web.UI.WebControls.Label Message;
        protected System.Web.UI.WebControls.TextBox email;
        protected System.Web.UI.WebControls.RequiredFieldValidator emailRequired;
        protected System.Web.UI.WebControls.RegularExpressionValidator emailValid;
        protected System.Web.UI.WebControls.TextBox password;
        protected System.Web.UI.WebControls.RequiredFieldValidator passwordRequired;
        protected System.Web.UI.WebControls.CheckBox RememberLogin;
        protected System.Web.UI.WebControls.ImageButton LoginBtn;
        //构造函数
        public Login() {
            Page.Init += new System.EventHandler(Page_Init);
        }
    }
}
```

定义登录按钮单击事件，该事件捕获页面中的用户输入的 email 和 password 并对用户身份进行验证，如果通过验证，则添加 Cookie 信息合并临时购物车信息到该用户的购物车信息中，并转向用户购物车页面；如果没有通过验证，则提示相关信息。

```
//定义按钮单击事件 LoginBtn_Click
private void LoginBtn_Click(object sender, System.Web.UI.ImageClickEventArgs e) {
    //检测用户的输入值是否通过页面中验证控件的验证。
    if (Page.IsValid == true) {
        //实例化一个购物车对象 shoppingCart
        ASPNET.StarterKit.Commerce.ShoppingCartDB shoppingCart = new ASPNET.StarterKit.Commerce.ShoppingCartDB();
        //调用 GetShoppingCartId 方法获得临时的购物车的 tempCartID
        String tempCartID = shoppingCart.GetShoppingCartId();
```

实例化类 CustomersDB，传入用户输入的 email 和 password 进行验证并获得返回值。

```
//实例化一个用户对象 accountSystem
ASPNET.StarterKit.Commerce.CustomersDB accountSystem = new ASPNET.StarterKit.Commerce.CustomersDB();
```

```
//调用 accountSystem 的 Login 方法传入用户输入的 email 和 password 进行验证并获得返回值
String customerId = accountSystem.Login(email.Text, ASPNET.StarterKit.Commerce.Components.Security.Encrypt(password.Text));
```

```
//判断返回的 customerId 是否为空
if (customerId != null) {
    //如果返回的 customerId 不为空则说明用户通过验证，那么将临时购物车转移成当前用户的购物车
    shoppingCart.MigrateCart(tempCartID, customerId);
```

下面是根据用户的 CustomerId 调用 GetCustomerDetails 方法返回一个 customerDetails 对象，在上一节数据访问层中介绍过该对象中存储了用户的详细信息。和注册过程一样，将名字为 ASPNETCommerce_FullName 的 Cookies 的值设置为当前用户的 FullName，这样是为了显示欢迎用户信息。

```
//调用 accountSystem 对象的 GetCustomerDetails 方法返回一个 customerDetails 实例
ASPNET.StarterKit.Commerce.CustomerDetails customerDetails = accountSystem.GetCustomerDetails(customerId);
//设置名为 ASPNETCommerce_FullName 的 Cookies 值为用户的 FullName
Response.Cookies["ASPNETCommerce_FullName"].Value = customerDetails.FullName;
//根据传入的 Checkbox 的值设定 Cookies ASPNETCommerce_FullName 的有效时间
if (RememberLogin.Checked == true) {
    Response.Cookies["ASPNETCommerce_FullName"].Expires = DateTime.Now.AddMonths(1);
}
//根据 customerId 创建用户身份验证票据并将用户定向到其他页面
FormsAuthentication.RedirectFromLoginPage(customerId, RememberLogin.Checked);
}
else {
    //如果输入的 email 和 password 没有通过验证，则显示登录失败的信息
    Message.Text = "Login Failed!";
}
}
//页面装载事件
private void Page_Load(object sender, System.EventArgs e) {
}
private void Page_Init(object sender, EventArgs e) {
    InitializeComponent();
}
private void InitializeComponent() {
    //定义按钮的单击事件委托
    this.LoginBtn.Click += new System.Web.UI.ImageClickEventHandler(this.LoginBtn_Click);
    this.Load += new System.EventHandler(this.Page_Load);
}
```

在上面传入用户输入密码的过程，使用了一个名称为 ASPNET.StarterKit.Commerce.Components.Security.Encrypt 的方法，该方法处于类 ASPNET.StarterKit.Commerce.Components.Security 中。下面就是该类的代码。

```
using System;
//声明 .NET Framework 提供加密服务的命名空间
using System.Security.Cryptography;
using System.Text;
```

```
//声明提供.NET Framework 提供正则表达式引擎的访问的命名空间  
using System.Text.RegularExpressions;  
  
namespace ASPNET.StarterKit.Commerce.Components  
{  
    public class Security  
    {
```

下面方法的功能是将传入的参数 cleanString 转化成为经过 MD5 加密的格式。目的就是为了提高字符串的加密程度。MD5 加密算法是一种不可逆的加密算法，所以经过这种方式加密的数据在传输和使用中更加安全，适用于密码的存储和传递。在上面的用户登录过程中，验证用户的密码正是使用了这种经过加密的密码格式。

```
//声明静态方法 Encrypt  
public static string Encrypt(string cleanString)  
{  
    //将传入的参数转化成 UTF-16 编码格式的字节类型变量  
    Byte[] clearBytes = new UnicodeEncoding().GetBytes(cleanString);  
    //将传入的参数转化成经过 MD5 算法加密的字节类型变量  
    Byte[] hashedBytes = ((HashAlgorithm) CryptoConfig.CreateFromName("MD5")).ComputeHash(clearBytes);  
    //将经过加密计算的字节型变量转化成 string 型并返回  
    return BitConverter.ToString(hashedBytes);  
}
```

5.3.6 用户表示层

用户登录过程的前台用户表示层页面为 Login.aspx，该页面调用了在 ASP.NET Commerce Starter Kit 系统中作为页面通过控件的用户控件 Header 和 menu，另外和用户注册页面 register.aspx 页面一样，使用验证控件对用户输入信息进行验证。由于页面没有过多的业务逻辑处理，因此在此并不重复叙述。

5.4 开发启示

上面介绍了 ASP.NET Commerce Starter Kit 系统中的用户体系，虽然在功能和逻辑上相对简单，但是考虑到这只是一个基础的电子商务应用模板，开发者并没有希望该系统直接被应用，而是希望作为一个模版为其他的开发者提供一个电子商务应用的基础框架，或是一个学习开发.NET 框架下电子商务 Web 应用程序的范例。

可以看到，在 ASP.NET Commerce Starter Kit 系统的用户体系中设计者更注重电子商务应用中最核心的用户功能的实现，在如下方面可以得到体现。

- 用户在没有登录或注册时的临时购物车和用户登录之后的购物车的整合。这个过程对于一个成功的电子商务应用是必不可少的，这种业务逻辑有利于吸引浏览者和方便客户订购商品。
- 用户注册的核心过程。实际上用户注册的过程就是获得系统唯一标识的过程，而对于用户其他的注册信息，在具体的电子商务应用中将根据设计者的需要添加。
- 用户密码的加密传输。这点对于实际的电子商务应用也是必不可少的，防止密码的泄漏，保证系统用户的安全性。

- 在登录之后由数据访问层返回一个包含用户详细信息的对象。这是面向对象思想在本系统中一个很好的展现，将用户信息设计成为一个对象，根据实际需要实例化用以传递用户信息。

在 ASP.NET Commerce Starter Kit 系统的用户体系中可以进行部分功能的扩充，或是在实际的应用中根据具体需求进行二次开发，如下几个方面都是进行功能扩充和二次开发的着手点。

- 用户管理模块的开发，用于系统管理员对系统中的用户信息进行管理。在一个完整的电子商务应用系统中，这个功能模块是必不可少的。
- 用户注册信息的扩展，可以根据需要获得更多的用户信息。在不同的应用中，要根据实际情况获得用户的其他信息。
- 用户登录信息的记录，比如登录 IP、最后一次登录时间、登录次数等。
- 用户等级的设定，对于一个成功的电子商务应用来说，用户的激励是必不可少的，设立用户等级有利于提高用户的访问次数和消费金额。

第 6 章

Web 商店

Web 商店作为电子商务应用系统中最核心的模块，具有呈现商品列表、展示商品信息、查询商品信息等基本功能。和实际购物商店一样，用户可以在这里查阅商品的分类，比较同一个类别中的商品，发现需要的商品，评论商品，将商品放入购物车，对购物车中的商品进行订购。表 6-1 所示为本章的知识点索引。

表 6-1 本章知识点索引

| 知 识 点 | 位 置 |
|--------------------------------------|---------|
| ASP.NET 中的数据访问技术 | 6.2.1 节 |
| ASP.NET 中的数据展现技术 | 6.2.2 节 |
| 用面向对象的思想将商品详细信息抽象成为对象 | 6.3.3 节 |
| 页面与用户控件之间的数据传递 | 6.3.4 节 |
| DataGrid 控件的使用 | 6.3.5 节 |
| DataList 控件的使用 | 6.3.5 节 |
| Repeater 控件的使用 | 6.3.5 节 |
| 从 context.User.Identity.Name 中获得用户信息 | 6.4.4 节 |
| 随机产生 GUID 创建临时购物车 | 6.4.4 节 |
| 购物车详细信息页面的数据更新与维护 | 6.4.4 节 |

6.1 系统设计

6.1.1 需求分析

从在系统中发布商品信息，到用户发现商品信息，再到用户把商品放到购物车，最后购买商品，这一整套流程在逻辑上是一个连续完整的过程，也是电子商务应用中最核心的一个购物流程。在 ASP.NET Commerce Starter Kit 中提供的正是这样一套经典的流程。ASP.NET Commerce Starter Kit 中 Web 商店模块的主要的流程和动作参见图 6-1。

在图 6-1 中，其流程主要包括 3 类元素：角色、页面和动作。具体元素信息参见表 6-2。

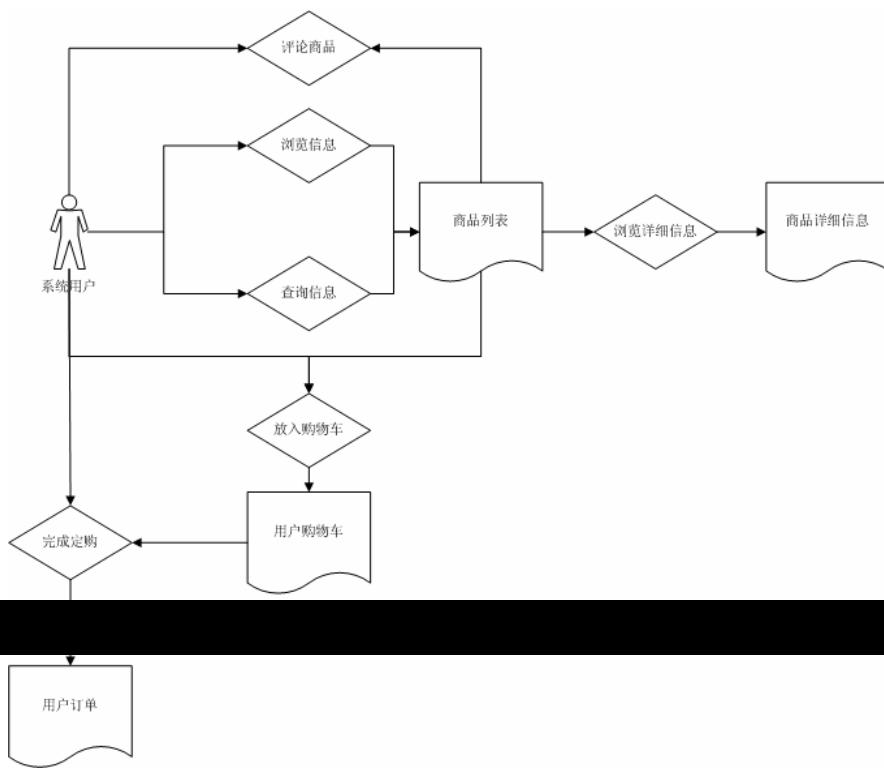


图 6-1 ASP.NET Commerce Starter Kit 的 Web 商店

表 6-2

Web 商店主要元素及说明

| 元素名称 | 元素类 | 元素 |
|------|-----|----|
| | | |

了解信息的信息源。用户可以通过分类浏览商品信息或查询商品信息获得需要的商品列表，然后通过商品列表打开商品详细信息页面。

在 ASP.NET Commerce Starter Kit 的商品列表页面中，每一条商品信息显示商品名称、商品图片、商品价格 3 个主要方面，并提供了两个链接，一个是单击商品图片或是商品名称可以进入商品详细信息页面，另一个是直接将商品放入购物车。用户可以先详细查看商品信息或快速将商品放入购物车。

(2) 商品详细信息

商品详细信息页面显示每一个商品的详细信息，包括商品名称、商品描述、商品图片、商品价格、购买此商品的顾客（同时还购买其他的商品）等信息。主要功能就是呈现商品的主要说明和参数，并且通过显示出购买此商品的顾客（同时还购买其他商品）的列表，在一定程度上给用户起到一个购物导向的作用，方便用户快速跳转到其他同类商品信息的页面。

(3) 商品评论

用户可以通过商品详细信息页面中评论商品的链接跳转到评论商品的页面，输入用户的用户名和 E-mail，可以对商品进行分数评价和文字评价。作为和用户沟通的一个很好的渠道，商品评论在很多电子商务应用中都被放到极为重要的位置。

注意：商品评论的最终目的是让电子商务的管理者和经营者了解到用户对商品的建议和意见，是架设在用户和电子商务提供者之间的桥梁。因此对于所有商品的查看和管理模块是必不可少的，但是在 ASP.NET Commerce Starter Kit 中并没有提供相应的功能模块，这并不是设计者的失误，而是为基于 ASP.NET Commerce Starter Kit 进行实际电子商务应用开发留下了更加广阔的扩展空间。

(4) 商品搜索

方便用户快速地找到需要的商品及其信息。当 Web 商店中的商品信息比较多的时候，用户通过列表的形式获得自己感兴趣的商品就变得困难。用户也许只知道该商品的部分信息，商品搜索正是为了满足这部分需求。搜索的结果页面实际上也是一个商品列表页面，罗列的商品信息和列表页面是一样的。

(5) 购物车

存储用户即将购买的商品，等待用户完成定购，即一个网络购物流程的结束。购物车是电子商务系统中最核心的部分，是每一笔订单形成的必经之路。

一个成功的电子商务应用的购物车模块不仅可以动态地存储用户即将购买的商品信息，也可以在用户不登录的情况下形成一个临时的购物车信息，当用户选择完成定购的时候再对用户的身份提出验证，如果用户通过身份验证，那么就将临时购物车中的数据信息转移成为当前用户的购物车信息，完成用户的定购。

另外用户可能在向购物车中添加了商品信息之后没有马上购买而退出系统，这时候需要系统能够记录用户购物车中的商品信息，以便用户在下次登录系统的时候对未完成的订单进行处理，也会防止用户在购物过程中因为网络连接问题或是客户端异常，而导致购物车信息丢失。

6.1.3 数据库设计

ASP.NET Commerce Starter Kit 中 Web 商店购物流程所对应的数据库模型参见图 6-2。数据结构比较简单，主要是围绕商品和订单两个模型构成整个数据库的模型。

由图 6-2 可以看出，整个数据库模型是以商品表 CMRC_Products 为主体，以订单的两个表 CMRC_Orders 和 CMRC_OrderDetails 为辅构建而成的。这是一个经典的电子商务的数据模型，各个数据表的主要描述参见表 6-3。

表 6-3 中不仅列出了各个表的主要作用，而且列出了表的主键和表与表之间的关系。如果说表是数据库的主体的话，那么关系就是数据库的灵魂。表的群集通过表与表之间的关系组成了关系数据库的一个整

体。每个表本身作为一个主体存储模型或是字典类别的信息，通过表之间的关系让数据结构显得更加健全和强壮。

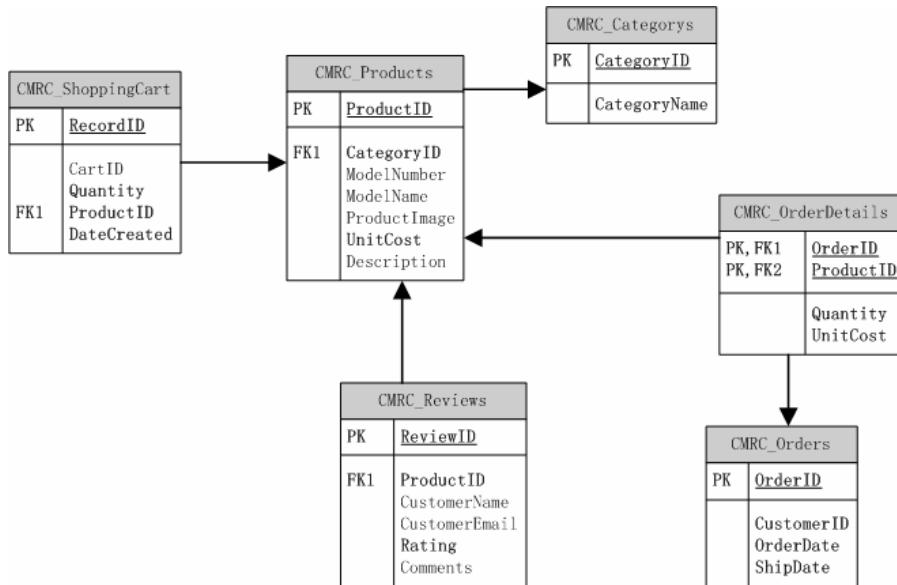


图 6-2 Web 商店购物流程所对应的数据库模型

表 6-3 数据表的主要描述

| 数据表名 | 主 要 描 述 |
|-------------------|--|
| CMRC_Products | 商品信息表，存储每一条商品信息数据。包括商品的名称、型号、图片地址、单价等。 这个表的主键是 ProductID。由 CMRC_OrderDetails 表的字段 ProductID 引用，由 CMRC_ShoppingCart 表的字段 ProductID 引用，由 CMRC_Reviews 表的字段 ProductID 引用 |
| CMRC_Categories | 商品类别表，存储商品的类别信息。包括类别的 ID 和类别名称。这个表的主键是 CategoryID。 由 CMRC_Products 表的字段 CategoryID 引用 |
| CMRC_Orders | 订单主体表，存储每一笔订单的主体信息。包括订单号码、购买用户 ID、订单日期等。 这个表的主键是 OrderID，由 CMRC_OrderDetails 表的字段 OrderID 引用 |
| CMRC_OrderDetails | 订单详细信息表，存储每一笔订单中所有的商品信息。包括订单号、商品 ID、购买数量和单价。 这是一个交叉映射表，有 OrderID 和 ProductID 两个主键，分别引用不同的表 |
| CMRC_Reviews | 商品评论表，存储商品的所有评论信息。包括评论的用户、评论的分数和评论的内容等。这个表的主键是 ReviewID |
| CMRC_ShoppingCart | 购物车表，存储即时的用户购物车内商品信息。包括购物车的用户、商品的 ID、商品的价格、购买的数量等。这个表的主键是 RecordID |

在设计关系数据库的时候，通常是先将需求抽象成模型，并将模型转化成具体的表。考虑到各个模型之间的联系，抽象成为表与表之间的关系。因此首先形成的是数据库整体模型的框架，在这个框架之上逐渐丰富表的内容，最终形成完整的数据库模型。

注意：在 ASP.NET Commerce Starter Kit 实例的数据库中并没有定义表与表之间的关系，本书为了方便读者理解和阅读，整理了数据库表之间的关系。希望通过理顺关系，能够帮助读者更好地理解 ASP.NET Commerce Starter Kit 的数据库模型设计。

本章用到的存储过程的简要说明参见表 6-4。

表 6-4

Web商店模块存储过程简要说明

| 存储过程名称 | 说 明 |
|-----------------------------|--------------------------|
| CMRC_ProductCategoryList | 获得商品列表的记录集 |
| CMRC_ProductDetail | 获得某件商品的详细信息 |
| CMRC_ProductsByCategory | 获得某一个商品分类所有商品的记录集 |
| CMRC_ProductSearch | 商品搜索 |
| CMRC_ProductsMostPopular | 获得热门商品列表的记录集 |
| CMRC_CustomerAlsoBought | 返回购买某个商品同时还购买其他商品的用户的记录集 |
| CMRC_ReviewsAdd | 添加一条商品评论 |
| CMRC_ReviewsList | 获得某件商品的所有评论记录集 |
| CMRC_ShoppingCartAddItem | 向用户购物车中添加一条商品信息 |
| CMRC_ShoppingCartEmpty | 清空某个购物车内的所有商品信息 |
| CMRC_ShoppingCartItemCount | 获得某个购物车内的商品种类数量 |
| CMRC_ShoppingCartList | 获得某个购物车内所有商品的详细信息记录集 |
| CMRC_ShoppingCartMigrate | 将两个购物车信息进行合并 |
| CMRC_ShoppingCartRemoveItem | 删除购物车内的某条商品信息 |
| CMRC_ShoppingCartTotal | 获得某个购物车内所有商品的总价格 |
| CMRC_ShoppingCartUpdate | 更新购物车内所有商品的信息 |

6.2 关键技术点

后台数据集的操作和前台数据页面的展示是本章中两个比较关键的技术点，设计者也充分使用了 ASP.NET 中常见的数据集操作和数据页面展示的几种方法，在不同的需求中使用有针对性的数据操作方法对于 ASP.NET 的设计来说显得尤为重要。

6.2.1 ASP.NET 中的数据访问技术

.NET Framework 提供了一系列新的数据访问技术的类库，.NET Framework 的数据访问技术被命名为 ADO.NET。新的数据访问类库并不是简单地对以往 ADO (ActiveX Data Objects) 数据访问技术的继承，而是提出了一项新的数据访问技术和框架。

在以往的 ADO 技术中，ASP 页面如果想对一个数据集中的数据进行展现，方法就是遍历当前数据集中的数据记录和字段，然后依次循环显示到 ASP 页面中的 HTML 元素中。数据集类型的单一和对数据库的实施依赖性大大降低了 ASP 页面中数据操作的效率。在 ADO.NET 技术中，没有一个统一的记录集对象，而是提供了 DataSet 和 DataReader 两个记录集对象。ADO.NET 中也没有了数据集中的游标概念，在 DataSet 对象中可以随时调用技术集中的某一条数据，而在 DataReader 中数据集只能向前逐条读取。在 ADO.NET 技术中，支持非连接数据集，DataSet 对象在数据集的使用过程中不再依赖数据库的实时连接。

ADO.NET 所使用的类库被统一封装到 System.Data 的几个命名空间下，其中部分命名空间的简单说明参见表 6-5。

表 6-5

System.Data 命名空间

| 命名空间 | 说明 |
|-----------------------|--|
| System.Data | 包含关系型数据访问和存储的基础对象 |
| System.Data.Common | 包含被其他命名空间所引用的通用类库，一般开发者不会直接用到这个命名空间下的类库 |
| System.Data.OleDb | 包含通过 OLE-DB 连接数据源所使用的类库 |
| System.Data.SqlClient | 包含用于访问托管空间中的 Microsoft SQL Server 数据库的类库和方法，由于删除了 OLE-DB 连接方式中的一些中间层，这种数据连接方式可以获得更好的数据访问性能 |
| System.Data.SqlTypes | 包含了 Microsoft SQL Server 数据库使用到的不同于.NET Framework 下标准数据类型所使用的类库 |

基于以上命名空间，ADO.NET 提提供了一系列新的数据对象。下面对 ADO.NET 所提供的数据对象进行详细的讲解。

(1) Connection 对象

和以往的数据库连接对象类似的对象是 OleDbConnection 和 SqlConnection 两个对象。这两个对象都是提供数据库连接的对象，不同的是 OleDbConnection 和 SqlConnection 两个对象属于不同命名空间。其中 OleDbConnection 属于 System.Data.OleDb 命名空间，而 SqlConnection 属于 System.Data.SqlClient 命名空间。在功能上，这两个数据库连接对象都是一样的，但是这两个数据库连接对象针对的分别是其特定的数据存储。

注意：因为 ADO.NET 提供了两套针对不同数据连接对象的数据库访问类库，在命名上两套类库是有规则的。System.Data.OleDb 命名空间下对象和方法都是以 OleDb 作为前缀，而 System.Data.SqlClient 命名空间下的对象和方法都是以 Sql 作为前缀。System.Data.OleDb 和 System.Data.SqlClient 两个命名空间提供的方法较为类似，开发者只需要根据所连接的数据对象的不同而选择不同的命名空间即可。在本书所讲解的示例中，后台数据库为 Microsoft SQL Server，所以主要使用的是 System.Data.SqlClient 命名空间下的对象和方法。

Connection 对象的主要方法和说明参见表 6-6。

表 6-6

Connection 对象的主要方法和说明

| 方法名 | 说明 |
|-------|--------------------------------|
| Open | 打开数据库方法，方法的参数为包含了数据库连接信息的连接字符串 |
| Close | 关闭数据库连接 |

在 ADO 技术中 Connection 对象是可以直接执行一条 SQL 语句的，但是在 ADO.NET 中 Connection 对象并不是直接对数据集进行操作的方法，而是仅仅作为数据库连接的对象被提供。

以 SqlConnection 为例，列举使用 Connection 对象的部分代码。

```
public void UserSqlConnection()
{
    //定义 SqlConnection 连接字符串
    string myConnectionString = "server=localhost;database=Commerce;uid=sa;pwd=";
    //实例化一个 SqlConnection
    SqlConnection myConnection = new SqlConnection(myConnectionString);
    //打开数据库连接
    myConnection.Open();
    //数据库详细操作此处省略
    //关闭数据库连接
}
```

```
    myConnection.Close();
}
```

(2) Command 对象

Command 对象可以直接执行 SQL 语句或是存储过程，能根据执行的 SQL 语句和存储过程返回一个记录集或在没有返回记录集的操作中返回受影响的记录数。针对不同的数据库连接对象，有对应的 OleDbCommand 和 SqlCommand。Command 对象的主要方法和说明参见表 6-7。

表 6-7

Command 对象的主要方法和说明

| 方 法 名 | 说 明 |
|-----------------|--|
| ExecuteNonQuery | 执行不返回记录集的 SQL 语句或存储过程的方法，例如 Update、Insert、Delete 操作，该方法返回一个 Integer 类型的数据，指明受操作影响的数据记录行数 |
| ExecuteReader | 执行返回记录集的 SQL 语句或存储过程的方法，返回一个 Reader 对象。在后面的章节中会对 Reader 对象进行详细讨论 |
| ExecuteScalar | 执行只返回单个值的 SQL 语句或存储过程的方法 |

以 SqlCommand 为例，列举使用 Command 对象的部分代码。

```
public void UserSqlCommand()
{
    //定义 SqlConnection 连接字符串
    string myConnectionString = " server=localhost;database=Commerce;uid=sa;pwd=";
    //实例化一个 SqlConnection
    SqlConnection myConnection = new SqlConnection(myConnectionString);
    //实例化一个 SqlCommand 并指定执行的 CommandText 属性和使用的 Connection 对象
    SqlCommand myCommand = new SqlCommand("CMRC_ProductCategoryList", myConnection);
    //指定 SqlCommand 的 CommandType
    myCommand.CommandType = CommandType.StoredProcedure;
    //打开数据库连接
    myConnection.Open();
    //执行 SqlCommand
    myCommand.ExecuteNonQuery();
    //关闭数据库连接
    myConnection.Close();
}
```

(3) DataSet 对象

DataSet 对象是 ADO.NET 中非连接存储和关系型数据处理的基础数据对象。可以将从数据库中取得的数据集全部存储于 DataSet 对象中，操作存储于 DataSet 的数据，然后再次连接到数据库，根据 DataSet 中的数据更新目前数据库中的数据。一个 DataSet 可以以 DataTable 的形式存储多个数据集，并且存储 DataTable 之间的关系 DataRelation，DataSet 相当于在内存中建立一个关系数据库的模型，所有对 DataSet 的数据库操作都是直接针对于当前 DataSet 存储的数据，而并不是填充 DataSet 的后台数据库或数据源。DataSet 对象所提供的主要方法及说明参见表 6-8。

表 6-8

DataSet 对象所提供的主要方法及说明

| 方 法 | 说 明 |
|---------------|--|
| Clear | 清空 DataSet 中的所有 DataTable |
| Merge | 把两个 DataSet 合并成为一个 DataSet |
| AcceptChanges | 更新自加载此 DataSet 或上次调用 AcceptChanges 以来对 DataSet 进行的所有更改 |

续表

| 方 法 | 说 明 |
|---------------|--|
| GetChanges | 获取自加载此 DataSet 或上次调用 AcceptChanges 以来对 DataSet 进行的所有更改 |
| HasChanges | 判断自加载此 DataSet 或上次调用 AcceptChanges 以来 DataSet 是否进行更改 |
| RejectChanges | 放弃 DataSet 或上次调用 AcceptChanges 以来对 DataSet 进行的所有更改 |

(4) DataTable 对象

每一个 DataSet 对象可以存储多个记录集，每个记录集就是以 DataTable 的形式存在的。如果把每一个 DataSet 看成一个关系数据库的话，那么 DataTable 对象就是关系数据库中的数据表。DataTable 对象和 DataSet 对象一样有 Clear、AcceptChanges、GetChanges、HasChanges、RejectChanges 等方法，而这些方法的功能也十分类似。在关系数据库中可以对数据表进行的操作在 DataTable 对象中也可以完成，比如说新建 DataTable，删除 DataTable 或对一个 DataTable 进行查询。

(5) DataColumn 对象

每一个 DataTable 对象中可能包含一个或多个数据字段，每一个数据字段就是以 DataColumn 的形式存在的，如果把每一个 DataSet 看成一个关系数据库的话，那么 DataColumn 就是关系数据库中数据表中的字段。

(6) DataRow 对象

每一个 DataTable 对象中可能包含一条或多条数据记录，每一条数据记录就是以 DataRow 的形式存在的，如果把每一个 DataSet 看成一个关系数据库的话，那么 DataRow 就是关系数据库中数据表中的每一条记录。

(7) DataView 对象

可以从 DataSet 对象中检索一个包含数据的 DataView 对象，DataView 对象类似于关系数据库中的数据视图。DataView 对象中包含的并不是真正的数据，而是基于 DataSet 对象中数据的查询条件。

(8) DataRelation 对象

在 DataSet 中可以定义 DataTable 之间的关系。和关系数据库中的数据关系一样，通过建立 DataRelation 对象可以将两个 DataTable 通过字段对应的关系对应起来。

手工构建一个 DataSet 的代码清单。

```
//创建一个 DataSet ds
DataSet ds = new DataSet();
//创建一个 DataTable dt1
DataTable dt1 = new DataTable();
//创建一个 DataTable dt2
DataTable dt2 = new DataTable();
//创建一个 DataColumn dc1
DataColumn dc1 = new DataColumn("CustomerID", typeof(int));
//创建一个 DataColumn dc2
DataColumn dc2 = new DataColumn("ProductID", typeof(int));
//创建一个 DataColumn dc3
DataColumn dc3 = new DataColumn("ProductName", typeof(string));
//添加 dc1 成为 dt1 的 DataColumn
dt1.Columns.Add(dc1);
//添加 dc2 成为 dt1 的 DataColumn
dt1.Columns.Add(dc2);
```

```

//添加 dc3 成为 dt2 的 DataColumn
dt2.Columns.Add(dc3);
//添加 dc4 成为 dt2 的 DataColumn
dt2.Columns.Add(dc4);
//添加 dt1 成为 ds 的 DataTable
ds.Tables.Add(dt1);
//添加 dt2 成为 ds 的 DataTable
ds.Tables.Add(dt2);
//在 dt1 的 ProductID 和 dt2 的 ProductID 之间建立名为 ProductIDs 的 DataRelation
ds.Relations.Add("ProductIDs", ds.Tables["dt1"].Columns["ProductID"],
ds.Tables["dt2"].Columns["ProductID"])

```

(9) DataReader 对象

DataSet 对象将数据对象存储于内存中，为非连接数据访问提供一套完整的解决方案。但是在很多情况下，开发者需要快速有效地访问数据，而并不需要对数据进行更新、删除等操作。例如在 ASP.NET 页面对某个数据集进行只读模式的显示，这种简单的显示只需要对一个数据集进行遍历显示即可，这时用 DataReader 对象就可以完成这些任务。

DataReader 对象提供了和 Recordset 中使用游标读取记录集类似的功能，但是 DataReader 对象对数据是只读和转发的，并没有更改数据的功能。而且 DataReader 只能对数据集进行向前的逐条读取，并不能跳过某条记录或是退回。DataReader 对象的部分方法及说明参见表 6-9。

表 6-9 DataReader 对象的部分方法及说明

| 方 法 | 说 明 |
|------------|--|
| Read | 将当前记录的指针移到下一条记录 |
| GetXXXX | 这是一组方法的集合，从当前结果集中以指定的数据类型获取某个值，例如 GetBoolean、GetInt16 和 GetChars |
| NextResult | 当执行的 SQL 语句或存储过程返回多个结果集的时候，该方法是将目前的结果集移到返回的下一个结果集 |
| Close | 关闭 DataReader 对象并释放对结果集的引用 |

本章中使用 SqlDataReader 返回商品分类记录集的代码片断。

```

public SqlDataReader GetProductCategories()
{
    //实例化一个数据库连接对象 myConnection
    SqlConnection myConnection = new
SqlConnection(ConfigurationSettings.AppSettings["ConnectionString"]);
    //实例化一个使用 myConnection 执行存储过程 CMRC_ProductCategoryList 的数据库命令 myCommand
    SqlCommand myCommand = new SqlCommand("CMRC_ProductCategoryList", myConnection);
    //设置 myCommand 的类型为执行存储过程
    myCommand.CommandType = CommandType.StoredProcedure;
    // 打开数据库连接 myConnection
    myConnection.Open();
    //执行 myCommand 并取得 SqlDataReader
    SqlDataReader result = myCommand.ExecuteReader(CommandBehavior.CloseConnection);
    //返回 SqlDataReader
    return result;
}

```

DataSet 和 DataReader 都可以完成对数据对象的存储及操作等行为，这对于开发人员在选择使用哪种对象作为数据库存储对象显得十分重要。相对于 DataReader 在数据读取方面的轻便和快捷来讲，

DataSet 对象过于复杂，但是 DataSet 对象强大的数据操作功能和非连接数据访问模式也是 DataReader 所不具备的。在总结和比较了 DataReader 和 DataSet 在功能和性能方面的特点之后，建议在如下情况中使用 DataSet。

- 当需要操作非连接数据，将数据发送到应用程序或是客户端，供使用者进行更改之后更新到数据库的时候，需要使用 DataSet。很明显，DataReader 满足不了这样的非连接数据操作。
- 当需要存储、传输和操作多个数据表，并且表之间存在着数据关系时，需要使用 DataSet。
- 当需要对记录集进行调度操作时，例如数据的分页浏览或是跳转到某条固定的记录，需要使用 DataSet。
- 在如下情况中建议使用 DataReader。
- 当需要一次性地读取并发送到客户端一个数据集的所有记录，而没有更新删除操作的时候，建议使用 DataReader。
- 当需要遍历一个数据集，对数据集的每一条记录进行只读性的数据分析的时候，建议使用 DataReader。

注意：在 ASP.NET Commerce Starter Kit 中并没有使用 DataSet 对象，基本上都是使用 SqlDataReader 对象作为数据对象，因为在 ASP.NET Commerce Starter Kit 中对数据传输以及管理的要求并不是很高，多数是数据的浏览及遍历，因此没有必要使用 DataSet 对象。DataReader 对象的高效和轻便很符合 ASP.NET Commerce Starter Kit 中对数据管理的需求。

6.2.2 ASP.NET 中的数据展现技术

在 ASP.NET 中的数据访问技术中，讨论了 ASP.NET 中提供的新的数据访问技术。数据访问的范畴是在应用程序中根据需求把数据库中的数据以数据集的形式提取，数据最终还是要以某种形式展现给使用者，数据的展现技术就是为了解决将数据集中的数据展现给客户这个问题的。

ASP.NET 引入了一些新的数据展现形式，这些新的展现形式都是通过一系列新的服务器控件实现的。数据展现的服务器控件和其他 ASP.NET 服务器控件一样，可以在 ASP.NET 的前端页面调用显示，也可以在后台编码类中对其进行操作。使用了数据展现的服务器控件之后，开发者只需完成控件和数据集的数据绑定，数据展现形式是脱离数据集而完全基于控件的，这样的数据展现模型更有利于创建和显示数据的业务规则与实现代码后置。

在 ASP.NET 页面的数据展现技术中，最重要的一个概念就是“数据绑定”。数据绑定概念并没有一个完整的定义，因为这个概念在不同的技术框架中都被提及。在 ASP.NET 页面中数据绑定则像一座桥梁连接数据集与数据展现控件，前端页面的每一个服务器控件都可以通过数据绑定和后台的数据集建立联系，而服务器端控件就会作为数据集的一种展现形式被使用者直接浏览。在 ASP.NET 中，不仅数据展现控件可以和数据集绑定，甚至普通的服务器控件都可以和后台的数据进行绑定。数据绑定的原理是将某个服务器端的数据集的一个或多个值插入到页面的某个控件中。在数据绑定中有两种基本的应用分类：单值绑定和重复值绑定。单值绑定是将后台编码类中某一个单独的值设置成为前台页面控件的某一个属性或属性标志；重复值绑定是将后台编码类中的某一个数据集绑定到一个可以显示到多个值的服务器控件上。在这里仅讨论 3 种专门的数据展现的服务器控件。

(1) DataGrid 控件

由 System.Web.UI.WebControls.DataGrid 类实现，DataGrid 是一个功能齐全的页面网格服务器控件，可以绑定来自 DataView、DataSet 和 DataReader 形式的数据源或集合形式的数据。DataGrid 控件在页面上生成一个 HTML 表格形式的可视界面，将栏或项的名称自动添加到标题行或表尾行，该控件具有很多强大的内置功能，方便开发者更加灵活方便地使用该控件显示和管理数据。

使用 DataGrid 控件可以将数据源的字段作为表中的列显示。DataGrid 控件中的每一行表示数据源

中的一个记录。DataGrid 控件支持选择、编辑、删除、分页和排序。不同的列类型决定控件中各列的行为。以下列出了可以使用的不同列类型。

- BoundColumn: 显示绑定到数据源中的字段的列, 以文本形式显示字段中的每个项。这是 DataGrid 控件的默认列类型。
- ButtonColumn: 为列中每个项显示一个命令按钮, 可以创建一列自定义按钮控件, 如 Add 按钮或 Remove 按钮。
- EditCommandColumn: 显示一列, 该列包含列中各个项的编辑命令。
- HyperLinkColumn: 将列中各项的内容显示为超级链接。列的内容可以绑定到数据源或静态文本中的字段。
- TemplateColumn: 按照指定的模板显示列中的各项。这使开发者可以在列中提供自定义控件。

默认情况下, DataGrid 控件的 AutoGenerateColumns 属性被设置为 true, 为数据源中的每一个字段创建一个 BoundColumn 对象, 然后作为 DataGrid 控件中的列呈现, 其顺序和每一字段在数据源中出现的顺序相同。当开发者想隐藏或是固定某一列的时候, 可以手动控制在 DataGrid 控件中显示哪些列, 方法是首先将 AutoGenerateColumns 属性设置为 false, 然后列出包括在开始和结束 <Columns> 标记之间的列。指定的列将以所列出的顺序添加到 Columns 集合中, 通过这种方法开发者可以方便地控制和管理 DataGrid 控件中的列。

(2) DataList 控件

由 System.Web.UI.WebControls.DataList 类实现, 可以绑定来自 DataView、DataSet 和 DataReader 形式的数据源或集合形式的数据。DataList 控件根据记录集中地记录显示<ItemTemplate>模板的内容。DataList 控件支持选择和编辑某一条被显示的记录, 但是 DataList 控件并没有 DataGrid 控件的功能强大, 因为 DataList 控件是基于模板式的控件, 除了循环显示模板<ItemTemplate>之外, DataList 控件还有以下模板。

- AlternatingItemTemplate: 为 DataList 中的交替项提供内容和布局。
- EditItemTemplate: 为 DataList 中当前编辑的项提供内容和布局。
- FooterTemplate: 为 DataList 的脚注部分提供内容和布局。
- HeaderTemplate: 为 DataList 的页眉部分提供内容和布局。
- SelectedItemTemplate: 为 DataList 中的当前选定项提供内容和布局。
- SeparatorTemplate: 为 DataList 中各项之间的分隔符提供内容和布局。

DataList 控件的显示方向可以是垂直或水平的。设置 RepeatDirection 属性以指定显示方向。DataList 控件的布局由 RepeatLayout 属性控制。将此属性设置为 RepeatLayout.Table 将以表的形式显示 DataList; 而设置为 RepeatLayout.Flow 则显示不具有表结构的 DataList。

(3) Repeater 控件

由 System.Web.UI.WebControls.Repeater 类实现, 也可以绑定来自 DataView、DataSet 和 DataReader 形式的数据源或集合形式的数据。Repeater 控件是一个基本模板数据绑定列表, 没有内置的布局或样式, 因此开发者必须在此控件的模板内显示声明所有的 HTML 布局、格式设置和样式标记。

Repeater 控件是惟一允许开发者在模板间拆分 HTML 标记的控件。若要利用模板创建表, 应在 HeaderTemplate 中包含表开始标记<table>, 在 ItemTemplate 中包含单个表行标记<tr>和<td>, 并在 FooterTemplate 中包含表结束标记</table>标签。

Repeater 控件没有内置的选择和编辑支持。开发者可以使用 ItemCommand 事件处理从模板引发到该控件的控件事件。控件将 ItemTemplate 和 AlternatingItemTemplate 绑定到由 DataSource 属性声明和引用的数据模型上。HeaderTemplate、FooterTemplate 和 SeparatorTemplate 都未进行数据绑定。如果设置了 Repeater 控件的数据源但该数据源未返回数据, 该控件将呈现不带项的 HeaderTemplate 和 FooterTemplate。每个 Repeater 必须至少定义一个 ItemTemplate, 用以显示数据源中的重复数据记录,

这也是 Repeater 控件的主体。

比较 DataGrid、DataList 和 Repeater 三个控件，在功能方面 DataGrid 控件功能最为强大，DataList 其次，Repeater 最弱。但是在轻巧性和易用性方面 Repeater 控件表现最好，其次是 DataList 控件，DataGrid 控件的使用最为复杂。3 个控件都有其各自的特点，在实际的项目中，要根据不同的需求选用不同的控件。ASP.NET Commerce Starter Kit 对于 3 种控件都有使用，例如在购物车页面 ShoppingCart.aspx 中，显示购物车内详细购物信息的时候，使用了 DataGrid 控件；在商品列表页面 ProductList.aspx 中，显示某个分类下面的所有商品的列表，使用了 DataList 控件；在热门商品列表用户控件 _PopularItems.ascx 中，显示热门商品列表，使用了 Repeater 控件。

6.3 商品浏览

6.3.1 实现效果

商品浏览主要包括两个主要的页面和一些辅助的列表。一个页面是商品列表页面，另外一个页面是商品详细信息页面 ProductDetails.aspx。两个页面的页面效果如图 6-3 和 6-4 所示。商品列表页面包括商品分类列表 productlist.aspx 和搜索列表 productslist.aspx 两个页面。

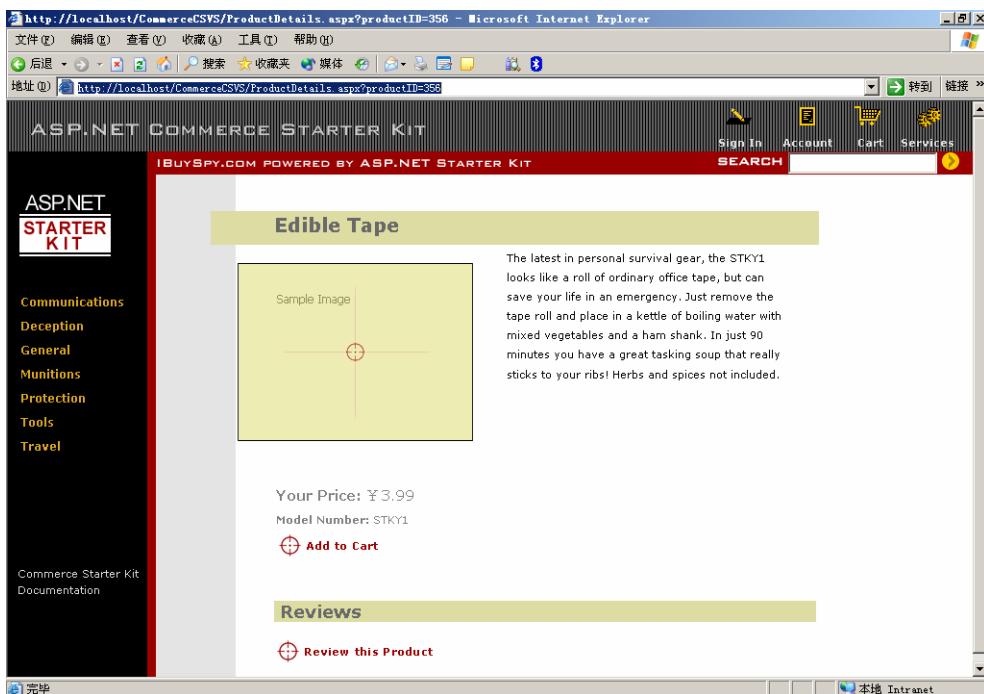


图 6-3 商品详细信息页面 ProductDetails.aspx 页面

单击两个商品列表页面的每个商品标题或是图片的链接，可以进入到商品详细信息页面。调用商品分类列表页面 productslist.aspx 是通过 URL 传入一个商品分类的 ID，调用搜索结果页面则是通过表单 Post 一个关键字符串。从商品列表页面传递一个商品 ID 的 URL 参数到商品详细信息页面，商品详细信息根据传入的商品 ID 显示具体的商品信息。商品列表页面中只显示商品的名称、价格、缩小图片和加入购物车

的链接。而商品详细信息页面则包含了所有的商品信息，并且包含加入购物车的链接、评论该商品的链接和该商品所有评论的列表。

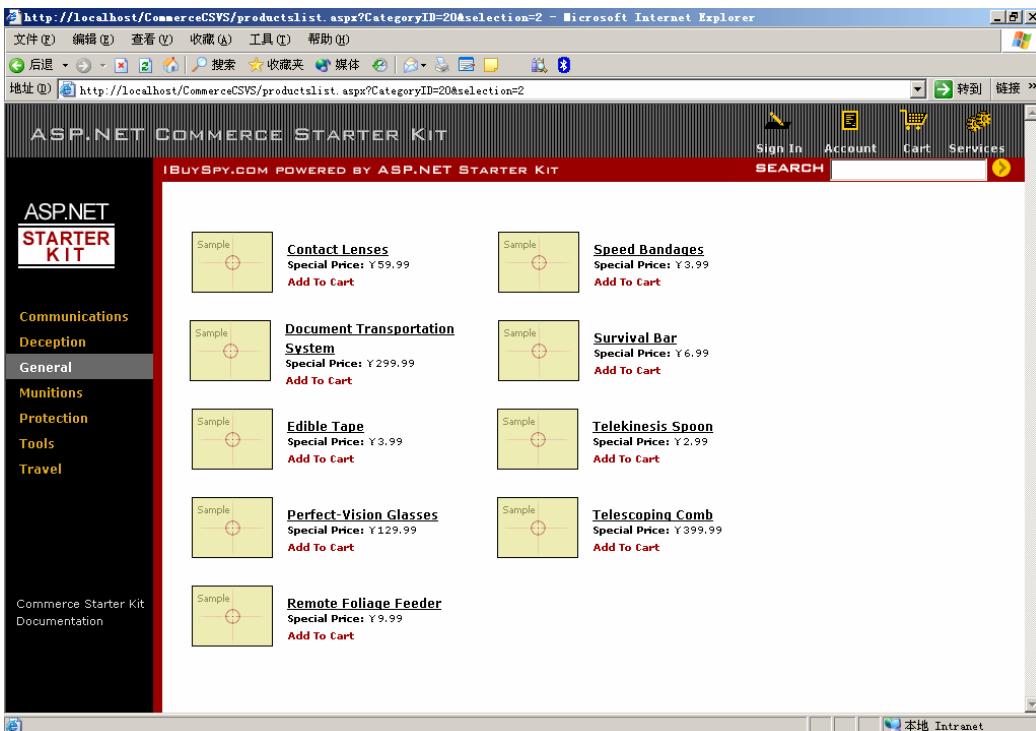


图 6-4 商品分类列表 productslist.aspx 页面效果

除了两个主要的页面，还有几个辅助的商品列表也可以显示商品信息并链接到商品详细信息页面。其中包括首页的热门商品列表，商品详细信息页面中购买此商品的用户通常还购买的其他商品列表，这两个辅助的列表都是以用户控件的形式存在的。分别是 PopularItems.ascx 和 AlsoBought.ascx。如图 6-5 和 6-6 所示。



图 6-5 热门商品列表



图 6-6 购买此商品的用户通常还购买的其他商品列表

6.3.2 存储过程

商品浏览这部分主要的存储过程有以下 6 个：CMRC_ProductCategoryList、CMRC_ProductDetail、CMRC_ProductsByCategory、CMRC_ProductSearch、CMRC_ProductsMostPopular、CMRC_CustomerAlsoBought。

(1) 存储过程 CMRC_ProductCategoryList

```
CREATE Procedure CMRC_ProductCategoryList
AS
```

```
/*按照 CategoryName 正序排列返回 CMRC_Categories 表中的所有记录*/
SELECT
    CategoryID,
    CategoryName
FROM
    CMRC_Categories
ORDER BY
    CategoryName ASC
GO
```

该存储过程比较简单，按照 CategoryName 字段正序排列返回一个所有商品类别的列表。在记录集中每一个 CategoryID 对应一个 CategoryName。这种只有两个字段一一对应的数据集可以被用到列表显示类别中，也可以被使用到诸如 DropDownList 这种选择性区域。

(2) 存储过程 CMRC_ProductDetail

```
CREATE Procedure CMRC_ProductDetail
(
    /*传入参数只有一个 ProductID，其余均为返回值型参数*/
    @ProductID      int,
    @ModelNumber    nvarchar(50) OUTPUT,
    @ModelName      nvarchar(50) OUTPUT,
    @ProductImage   nvarchar(50) OUTPUT,
    @UnitCost       money OUTPUT,
    @Description    nvarchar(4000) OUTPUT
)
AS
/*根据商品的 ProductID 返回商品的详细信息。*/
SELECT
    @ProductID      = ProductID,
    @ModelNumber    = ModelNumber,
    @ModelName      = ModelName,
    @ProductImage   = ProductImage,
    @UnitCost       = UnitCost,
    @Description    = Description
FROM
    CMRC_Products
WHERE
    ProductID = @ProductID
GO
```

该存储过程是根据某个商品的 ProductID 字段值返回该商品的 ModelNumber、ModelName、ProductImage、UnitCost 和 Description 字段信息。根据该存储过程的设计结构可以看出，该存储过程被用于根据 ProductID 查看商品详细信息的页面，在数据层根据商品的 ProductID 构建一个商品的对象也可以使用该存储过程。

(3) 存储过程 CMRC_ProductsByCategory

```
CREATE Procedure CMRC_ProductsByCategory
(
    @CategoryID int
)
AS
/*根据类别的 CategoryID 返回该类别所有商品信息的记录集*/
SELECT
```

```
ProductID,
    ModelName,
    UnitCost,
    ProductImage
FROM
    CMRC_Products
WHERE
    CategoryID = @CategoryID
ORDER BY
    ModelName,
    ModelNumber
GO
```

该存储过程是根据类别的 CategoryID 返回该类别下所有商品信息的记录集，并且把所有商品信息按照 ModelName 和 ModelNumber 排序。根据该存储过程的设计结构可以看出，该存储过程被用于商品分类列表页面，根据 URL 传入的商品分类参数列出该商品分类下所有商品的信息。同时该存储过程也可以被用到表单中表示某一个分类产品信息的 DropDownList 或是 CheckBox 控件中。

(4) 存储过程 CMRC_ProductSearch

```
CREATE Procedure CMRC_ProductSearch
(
    @Search nvarchar(255)
)
AS
/*返回以传入参数 Search 为关键字的商品信息查询结果集*/
SELECT
    ProductID,
    ModelName,
    ModelNumber,
    UnitCost,
    ProductImage
FROM
    CMRC_Products
WHERE
    /*对 ModelNumber、ModelName、Description 三个字段进行关键字模糊查询*/
    ModelNumber LIKE '%' + @Search + '%'
    OR
    ModelName LIKE '%' + @Search + '%'
    OR
    Description LIKE '%' + @Search + '%'
GO
```

该存储过程的功能是返回关键字模糊查询的商品信息记录集，以传入参数 Search 为关键字对商品信息表 CMRC_Products 中的 ModelNumber、ModelName、Description 三个字段进行模糊查询并返回结果集。该存储过程被用于商品搜索的功能模块。

(5) 存储过程 CMRC_ProductsMostPopular

```
CREATE Procedure CMRC_ProductsMostPopular
AS
/*返回 5 条被购买次数最多的商品的 ProductID、ModelName*/
SELECT TOP 5
    CMRC_OrderDetails.ProductID,
    /*计算某个商品的被购买总次数*/
```

```

        SUM(CMRC_OrderDetails.Quantity) as TotalNum,
        CMRC_Products.ModelName
    FROM
        CMRC_OrderDetails
        /*建立两个表 CMRC_OrderDetails、CMRC_Products 之间的查询连接*/
    INNER JOIN CMRC_Products ON CMRC_OrderDetails.ProductID = CMRC_Products.ProductID
        /*按照 ProductID 和 ModelName 进行分组*/
    GROUP BY
        CMRC_OrderDetails.ProductID,
        CMRC_Products.ModelName
    ORDER BY
        TotalNum DESC
    GO

```

该存储过程的功能是返回被购买次数最多的商品 ProductID 和 ModelName。该存储过程被用于热门商品列表用户控件 PopularItems.ascx 中。

(6) 存储过程 CMRC_CustomerAlsoBought

```

CREATE Procedure CMRC_CustomerAlsoBought
(
    @ProductID int
)
As

/* 返回购买某个商品的用户同时购买其他商品次数最多的 5 个商品记录集*/
SELECT TOP 5
    CMRC_OrderDetails.ProductID,
    CMRC_Products.ModelName,
    SUM(CMRC_OrderDetails.Quantity) as TotalNum
FROM
    CMRC_OrderDetails
    INNER JOIN CMRC_Products ON CMRC_OrderDetails.ProductID = CMRC_Products.ProductID
    WHERE OrderID IN
    (
        /* 利用子查询获得购买其他商品的 OrderID */
        SELECT DISTINCT OrderID
        FROM CMRC_OrderDetails
        WHERE ProductID = @ProductID
    )
    /* 需要排除该商品本身 */
    AND CMRC_OrderDetails.ProductID != @ProductID
GROUP BY CMRC_OrderDetails.ProductID, CMRC_Products.ModelName
ORDER BY TotalNum DESC
GO

```

该存储过程功能是返回购买某个商品的用户同时购买其他商品次数最多的 5 个商品记录集。该存储过程实现比较复杂，在主体查询中主要是获得该商品记录集的 ProductID、ModelName 和累计被购买的次数，并根据累计被购买次数进行排序，选出排在前 5 位的记录集，在子查询中是对订单 OrderID 的筛选。该存储过程被用于购买此商品的用户通常还购买其他商品的列表用户控件 AlsoBought.ascx 中。

6.3.3 数据访问层

商品浏览部分的数据访问层被建立在 Components 文件夹下的 ProductDB.cs 文件中，在 ProductDB 类中设计者构建了一个商品的对象以及实现商品浏览相关操作的方法。以下是 ProductDB.cs 的详细代码清单。

```
using System;
using System.Configuration;
using System.Data;
using System.Data.SqlClient;

namespace ASPNET.StarterKit.Commerce {
```

首先构建一个商品详细信息的类，这是一个简单数据类。该类中包含了对商品对象所有属性的定义。这个类并不能完成任何的业务逻辑，但是却能为其他逻辑类传递和存储商品详细信息提供一个有效的载体。构建商品类主要理论是依据面向对象的思想，将商品抽象成为一个对象，将商品的信息抽象成为该对象的属性。

```
public class ProductDetails {
    //定义商品型号 ModelNumber
    public String ModelNumber;
    //定义商品名称 ModelName
    public String ModelName;
    //定义商品的图片位置 ProductImage
    public String ProductImage;
    //定义商品价格 UnitCost
    public decimal UnitCost;
    //定义商品说明 Description
    public String Description;
}
```

ProductsDB 类是一个真正的业务逻辑类，其中实现了商品浏览过程中使用到的所有数据访问逻辑。包括返回商品列表和返回商品详细信息。

```
public class ProductsDB {
```

GetProductCategories 方法是返回商品分类记录集的 SqlDataReader 的方法，该方法执行数据库的存储过程 CMRC_ProductCategoryList。

```
public SqlDataReader GetProductCategories()
{
    //实例化一个数据库连接对象 myConnection
    SqlConnection myConnection = new
SqlConnection(ConfigurationSettings.AppSettings["ConnectionString"]);
    //实例化一个使用 myConnection 执行存储过程 CMRC_ProductCategoryList 的数据库命令 myCommand
    SqlCommand myCommand = new SqlCommand("CMRC_ProductCategoryList", myConnection);
    //设置 myCommand 的类型为执行存储过程
    myCommand.CommandType = CommandType.StoredProcedure;
    // 打开数据库连接 myConnection
    myConnection.Open();
    //执行 myCommand 并取得 SqlDataReader
    SqlDataReader result = myCommand.ExecuteReader(CommandBehavior.CloseConnection);
    //返回 SqlDataReader
    return result;
}
```

GetProducts 方法是返回某个商品类别下所有商品信息的 SqlDataReader 的方法，传入的 int 类型参数 categoryID 是商品类别 ID，该方法调用了数据库存储过程 CMRC_ProductsByCategory。

```
public SqlDataReader GetProducts(int categoryID) {
    //实例化一个数据库连接对象 myConnection
    SqlConnection myConnection = new SqlConnection(ConfigurationSettings.AppSettings["ConnectionString"]);
    //实例化一个使用 myConnection 执行存储过程 CMRC_ProductsByCategory 的数据库命令 myCommand
    SqlCommand myCommand = new SqlCommand("CMRC_ProductsByCategory", myConnection);
    //设置 myCommand 的类型为执行存储过程
    myCommand.CommandType = CommandType.StoredProcedure;

    //定义一个 SqlParameter 类型的存储过程参数
    SqlParameter parameterCategoryID = new SqlParameter("@CategoryID", SqlDbType.Int, 4);
    //设置该存储过程参数的值
    parameterCategoryID.Value = categoryID;
    //将存储过程参数添加到数据库命令 myCommand 中去
    myCommand.Parameters.Add(parameterCategoryID);

    //打开数据库连接 myConnection
    myConnection.Open();
    //执行 myCommand 并取得 SqlDataReader
    SqlDataReader result = myCommand.ExecuteReader(CommandBehavior.CloseConnection);
    //返回 SqlDataReader
    return result;
}
```

GetProductDetails 方法是返回某个商品详细信息的方法。该方法返回商品信息是以上面定义的 ProductDetails 对象类别形式表现出来的。根据商品的 productID 返回的商品信息包含了商品的名称、价格、型号、图片和说明。

```
public ProductDetails GetProductDetails(int productID) {
    //实例化一个数据库连接对象 myConnection
    SqlConnection myConnection = new SqlConnection(ConfigurationSettings.AppSettings["ConnectionString"]);
    //实例化一个使用 myConnection 执行存储过程 CMRC_ProductDetail 的数据库命令 myCommand
    SqlCommand myCommand = new SqlCommand("CMRC_ProductDetail", myConnection);
    //设置 myCommand 的类型为执行存储过程
    myCommand.CommandType = CommandType.StoredProcedure;
    //定义一个 SqlParameter 类型的存储过程参数 parameterProductID
    SqlParameter parameterProductID = new SqlParameter("@ProductID", SqlDbType.Int, 4);
    //设置该存储过程参数的值
    parameterProductID.Value = productID;
    //将存储过程参数添加到数据库命令 myCommand 中去
    myCommand.Parameters.Add(parameterProductID);

    SqlParameter parameterUnitCost = new SqlParameter("@UnitCost", SqlDbType.Money, 8);
    parameterUnitCost.Direction = ParameterDirection.Output;
    myCommand.Parameters.Add(parameterUnitCost);

    SqlParameter parameterModelNumber = new SqlParameter("@ModelNumber", SqlDbType.NVarChar, 50);
    parameterModelNumber.Direction = ParameterDirection.Output;
```

```
myCommand.Parameters.Add(parameterModelNumber);

SqlParameter parametermodelName = new SqlParameter("@ModelName", SqlDbType.NVarChar, 50);
parametermodelName.Direction = ParameterDirection.Output;
myCommand.Parameters.Add(parametermodelName);

SqlParameter parameterProductImage = new SqlParameter("@ProductImage", SqlDbType.NVarChar, 50);
parameterProductImage.Direction = ParameterDirection.Output;
myCommand.Parameters.Add(parameterProductImage);

SqlParameter parameterDescription = new SqlParameter("@Description", SqlDbType.NVarChar, 3800);
parameterDescription.Direction = ParameterDirection.Output;
myCommand.Parameters.Add(parameterDescription);

//打开数据库连接 myConnection
myConnection.Open();
//执行数据库命令 myCommand
myCommand.ExecuteNonQuery();
//关闭数据库连接
myConnection.Close();
//构建一个商品详细信息的对象
ProductDetails myProductDetails = new ProductDetails();
//设置商品详细信息对象的成员变量的值
myProductDetails.ModelNumber = (string)parameterModelNumber.Value;
myProductDetails.ModelName = (string)parametermodelName.Value;
myProductDetails.ProductImage = ((string)parameterProductImage.Value).Trim();
myProductDetails.UnitCost = (decimal)parameterUnitCost.Value;
myProductDetails.Description = ((string)parameterDescription.Value).Trim();
//返回商品详细信息对象
return myProductDetails;
}
```

GetProductsAlsoPurchased 方法是返回购买某一种商品的用户同时购买的其他商品信息的方法，返回的商品信息是以 SqlDataReader 形式表现的。

```
public SqlDataReader GetProductsAlsoPurchased(int productID) {
    //实例化一个数据库连接对象 myConnection
    SqlConnection myConnection = new SqlConnection(ConfigurationSettings.AppSettings["ConnectionString"]);
    //实例化一个使用 myConnection 执行存储过程 CMRC_CustomerAlsoBought 的数据库命令 myCommand
    SqlCommand myCommand = new SqlCommand("CMRC_CustomerAlsoBought", myConnection);
    //设置 myCommand 的类型为执行存储过程
    myCommand.CommandType = CommandType.StoredProcedure;

    //定义一个 SqlParameter 类型的存储过程参数 parameterProductID
    SqlParameter parameterProductID = new SqlParameter("@ProductID", SqlDbType.Int, 4);
    //设置该存储过程参数的值
    parameterProductID.Value = productID;
    //将存储过程参数添加到数据库命令 myCommand 中去
    myCommand.Parameters.Add(parameterProductID);

    //打开数据库连接 myConnection
```

```
    myConnection.Open();
    //执行 myCommand 并取得 SqlDataReader
    SqlDataReader result = myCommand.ExecuteReader(CommandBehavior.CloseConnection);
    //返回 SqlDataReader
    return result;
}
```

GetMostPopularProductsOfWeek 方法是返回最新热门商品列表的方法。返回的商品列表是以 SqlDataReader 的形式表现出来的。

```
public SqlDataReader GetMostPopularProductsOfWeek() {
    //实例化一个数据库连接对象 myConnection
    SqlConnection myConnection = new SqlConnection(ConfigurationSettings.AppSettings["ConnectionString"]);
    //实例化一个使用 myConnection 执行存储过程 CMRC_ProductsMostPopular 的数据库命令 myCommand
    SqlCommand myCommand = new SqlCommand("CMRC_ProductsMostPopular", myConnection);
    //设置 myCommand 的类型为执行存储过程
    myCommand.CommandType = CommandType.StoredProcedure;
    //打开数据库连接 myConnection
    myConnection.Open();
    //执行 myCommand 并取得 SqlDataReader
    SqlDataReader result = myCommand.ExecuteReader(CommandBehavior.CloseConnection);
    //返回 SqlDataReader
    return result;
}
```

SearchProductDescriptions 方法是返回关键字搜索商品列表的方法。返回的商品列表是以 SqlDataReader 的形式表现出来的。

```
public SqlDataReader SearchProductDescriptions(string searchString) {
    //实例化一个数据库连接对象 myConnection
    SqlConnection myConnection = new SqlConnection(ConfigurationSettings.AppSettings["ConnectionString"]);
    //实例化一个使用 myConnection 执行存储过程 CMRC_ProductSearch 的数据库命令 myCommand
    SqlCommand myCommand = new SqlCommand("CMRC_ProductSearch", myConnection);
    //设置 myCommand 的类型为执行存储过程
    myCommand.CommandType = CommandType.StoredProcedure;

    //定义一个 SqlParameter 类型的存储过程参数 parameterSearch
    SqlParameter parameterSearch = new SqlParameter("@Search", SqlDbType.NVarChar, 255);
    //设置该存储过程参数的值
    parameterSearch.Value = searchString;
    //将存储过程参数添加到数据库命令 myCommand 中去
    myCommand.Parameters.Add(parameterSearch);
    //打开数据库连接对象 myConnection
    myConnection.Open();
    //执行 myCommand 并取得 SqlDataReader
    SqlDataReader result = myCommand.ExecuteReader(CommandBehavior.CloseConnection);
    //返回 SqlDataReader
    return result;
}
}
```

6.3.4 业务逻辑层

(1) 商品分类列表页面后台编码类 ProductsList.aspx.cs

商品分类列表页面 ProductsList.aspx 的后台编码类为 ProductsList.aspx.cs，在逻辑上这个类接收前台页面的 URL 参数 categoryID，调用数据访问层的 GetProducts 方法返回该类别下所有商品的列表，并绑定到前台页面的 DataList。

该类首先声明了一个 DataList，该 DataList 为前台显示商品列表的 DataList。

```
protected System.Web.UI.WebControls.DataList MyList;
```

以下是该类提供的页面载入方法 Page_Load 的代码清单。

```
private void Page_Load(object sender, System.EventArgs e) {
    //从前台页面获得参数 CategoryID，并转化成 int 型
    int categoryId = Int32.Parse(Request.Params["CategoryID"]);
    //实例化 ProductsDB 成为对象 productCatalogue
    ASPNET.StarterKit.Commerce.ProductsDB productCatalogue = new ASPNET.StarterKit.Commerce.
ProductsDB();
    //调用 productCatalogue 对象的 GetProducts 方法获得数据集并设定为 MyList 的数据源
    MyList.DataSource = productCatalogue.GetProducts(categoryId);
    //绑定 MyList
    MyList.DataBind();
}
```

以上为该页面编码类的主要逻辑代码，可以看到在后台的页面编码类中，逻辑方面只完成了两个功能：一是获得数据源，二是将数据源绑定到前台的 DataList。对于前台的 DataList 的其他属性，并没有涉及。

(2) 商品详细信息页面后台编码类 ProductDetails.aspx.cs

商品详细信息页面 ProductDetails.aspx 的后台编码类为 ProductDetails.aspx.cs，在逻辑上这个类接收前台页面的 URL 参数 ProductID，调用数据访问层的 GetProductDetails 方法返回一个数据库访问层中定义的 ProductDetails 对象，提取对象中的商品信息传给前台的服务器端控件。该后台编码类还声明了两个用户控件 ReviewList 和 AlsoBoughtList。

首先声明了前台页面中使用到的控件。

```
//定义前台页面中使用到的服务器控件
protected System.Web.UI.WebControls.Label ModelName;
protected System.Web.UI.WebControls.Image ProductImage;
protected System.Web.UI.WebControls.Label UnitCost;
protected System.Web.UI.WebControls.Label ModelNumber;
protected System.Web.UI.WebControls.HyperLink addToCart;
protected System.Web.UI.WebControls.Label desc;
//定义前台页面中使用到的用户控件
protected C_ReviewList ReviewList;
protected C_AlsoBought AlsoBoughtList;
```

以下是该类提供的页面载入方法 Page_Load 的代码清单。

```
private void Page_Load(object sender, System.EventArgs e) {
    //从前台页面获得参数 ProductID，并转化成 int 型
    int ProductID = Int32.Parse(Request.Params["ProductID"]);

    //实例化类 ProductsDB 成为一个对象 products
    ASPNET.StarterKit.Commerce.ProductsDB products = new ASPNET.StarterKit.Commerce.ProductsDB();
    //调用 products 对象的 GetProductDetails 方法获得商品信息对象 myProductDetails
    ASPNET.StarterKit.Commerce.ProductDetails myProductDetails = products.GetProductDetails
```

```
(ProductID);

    //提取 myProductDetails 中的产品信息，设置前台页面服务器控件的值
    desc.Text = myProductDetails.Description;
    //使用 String.Format 方法格式化产品价格的格式
    UnitCost.Text = String.Format("{0:c}", myProductDetails.UnitCost);
    ModelName.Text = myProductDetails.ModelName;
    ModelNumber.Text = myProductDetails.ModelNumber.ToString();
    ProductImage.ImageUrl = "ProductImages/" + myProductDetails.ProductImage;
    addToCart.NavigateUrl = "AddToCart.aspx?ProductID=" + ProductID;
    //设置页面引用的用户控件中变量的值，实现页面与用户控件之间的数据传递
    ReviewList.ProductID = ProductID;
    AlsoBoughtList.ProductID = ProductID;
}
```

(3) 商品搜索结果后台编码类 SearchResults.aspx.cs

商品信息搜索结果页 SearchResults.aspx 的后台编码类为 SearchResults.aspx.cs，该编码类在逻辑上接受前台页面中的参数 txtSearch，并将其作为关键字参数调用数据库访问层中的 SearchProductDescriptions 方法，返回一个商品关键字搜索的记录集，最后将返回的记录集绑定到前台的 DataList 中。

首先声明前台页面使用的 DataList 和 Label。

```
protected System.Web.UI.WebControls.DataList myList;
protected System.Web.UI.WebControls.Label errorMsg;
```

以下是该类提供的页面载入方法 Page_Load 的代码清单。

```
private void Page_Load(object sender, System.EventArgs e) {
    //实例化类 ProductsDB 成为一个对象 productCatalogue
    ASPNET.StarterKit.Commerce.ProductsDB productCatalogue = new ASPNET.StarterKit.Commerce.
ProductsDB();
    //获得前台页面的参数 txtSearch
    //调用 productCatalogue 的 SearchProductDescriptions 方法返回记录集作为 myList 的数据源
    myList.DataSource = productCatalogue.SearchProductDescriptions(Request.Params["txtSearch"]);
    //绑定 myList
    myList.DataBind();
    //判断 myList 中的商品记录是否为空
    if (myList.Items.Count == 0) {
        //如果为空则显示错误信息，提示没有找到记录
        errorMsg.Text = "No items matched your query.";
    }
}
```

(4) 热门商品列表用户控件后台编码类_PopularItems.ascx.cs

热门商品列表用户控件_PopularItems.ascx 的后台编码类为_PopularItems.ascx.cs，该后台编码类通过调用数据访问层的方法 GetMostPopularProductsOfWeek 获得热门商品的数据集，并将该数据集绑定在用户控件 Repeater。

首先声明前台页面使用的 Repeater 以及商品的 ProductID。

```
protected System.Web.UI.WebControls.Repeater productList;
public int ProductID;
```

以下是该类提供的页面载入方法 Page_Load 的代码清单。

```
private void Page_Load(object sender, System.EventArgs e) {
    //实例化类 ProductsDB 成为一个对象 products
```

```

        ASPNET.StarterKit.Commerce.ProductsDB products = new ASPNET.StarterKit.Commerce.ProductsDB();

        //将 productList 的数据源设置成调用对象 products 的 GetMostPopularProductsOfWeek 方法获得热门商品
记录集
        productList.DataSource = products.GetMostPopularProductsOfWeek();
        //绑定 productList
        productList.DataBind();
        //判断 productList 中的商品记录是否为空
        if (productList.Items.Count == 0) {
            //如果商品记录为空则隐藏 productList
            productList.Visible = false;
        }
    }
}

```

(5) 购买某种商品的用户购买其他商品信息列表用户控件的后台编码类_AlsoBought.ascx.cs

购买某种商品的用户购买其他商品信息列表用户控件_AlsoBought.ascx 的后台编码类为 _AlsoBought.ascx.cs，该后台编码类以商品的 ProductID 作为参数，通过调用数据访问层的方法 GetProductsAlsoPurchased 获得购买该商品的用户同时还购买其他商品列表的数据集，并将该数据集绑定到用户控件 AlsoBought 中的 Repeater。

首先声明了用户控件中使用的 DataList 以及商品的 ProductID。

注意：购买某种商品的用户购买其他商品信息列表用户控件 AlsoBought 仅在商品详细信息页面 ProductDetails.aspx 中被调用，用于显示该商品对应的用户购买其他商品信息列表。在商品详细信息页面 ProductDetails.aspx 的后台编码类中对用户控件 AlsoBought 的 ProductID 赋值，这样在用户控件 AlsoBought 的后台编码类就可以直接使用 ProductID 了。

```

protected System.Web.UI.WebControls.Repeater alsoBoughtList;
public int ProductID;

```

以下是该后台编码类提供的页面载入方法 Page_Load 的代码清单。

```

private void Page_Load(object sender, System.EventArgs e) {
    //实例化类 ProductsDB 成为一个对象 productCatalogue
    ASPNET.StarterKit.Commerce.ProductsDB productCatalogue = new ASPNET.StarterKit.Commerce.
ProductsDB();
    //以 ProductID 作为参数调用对象 productCatalogue 的 GetProductsAlsoPurchased 方法获得记录集
    //将记录集设置成 alsoBoughtList 的数据源
    alsoBoughtList.DataSource = productCatalogue.GetProductsAlsoPurchased(ProductID);
    //绑定 alsoBoughtList
    alsoBoughtList.DataBind();

    //判断 alsoBoughtList 中的商品记录是否为空
    if (alsoBoughtList.Items.Count == 0) {
        //如果商品记录为空，则隐藏 alsoBoughtList
        alsoBoughtList.Visible = false;
    }
}

```

6.3.5 用户表示层

(1) 商品分类列表页面 ProductsList.aspx

在商品分类列表页面首先使用了页面缓存。以页面 URL 参数 CategoryID 作为高速缓存更新查询字符串。

```
<%@ OutputCache Duration="6000" VaryByParam="CategoryID" %>
```

商品的信息是通过名称为 MyList 的 DataList 形式显示出来的，该 DataList 在后台编码类中获得数据源，在前台页面主要是对 DataList 的呈现样式定义。以下是 DataList 部分的代码。

```
<!-- 定义 DataList 的 id 和列数，此处定义的 DataList 的 id 要和后台编码类中 DataList 的 id 相同，列数定义为 2 即以两列显示 -->
<asp:DataList id="MyList" RepeatColumns="2" runat="server">
    <!-- 定义模板 -->
    <ItemTemplate>
        <table border="0" width="300">
            <tr>
                <td width="25">
                    &nbsp
                </td>
                <td width="100" valign="middle" align="right">
                    <!-- 使用 DataBinder.Eval 取得后台编码类中的变量值 -->
                    <a href='ProductDetails.aspx?productID=<%# DataBinder.Eval(Container.DataItem,
"ProductID") %>'>
                        <img src='ProductImages/thumbs/<%# DataBinder.Eval(Container.DataItem,
"ProductImage") %>' width="100" height="75" border="0">
                    </a>
                </td>
                <td width="200" valign="middle">
                    <a href='ProductDetails.aspx?productID=<%# DataBinder.Eval(Container.DataItem,
"ProductID") %>'>
                        <span class="ProductListHead">
                            <%# DataBinder.Eval(Container.DataItem, "ModelName") %>
                        </span>
                        <br>
                        </a><span class="ProductListItem"><b>Special Price: </b>
                            <%# DataBinder.Eval(Container.DataItem, "UnitCost", "{0:c}") %>
                        </span>
                        <br>
                        <a href='AddToCart.aspx?productID=<%# DataBinder.Eval(Container.DataItem,
"ProductID") %>'>
                            <span class="ProductListItem"><font color="#000099"><b>Add To Cart</b></font>
                        </span>
                    </a>
                </td>
            </tr>
        </table>
    </ItemTemplate>
    <!-- 结束模板定义 -->
</asp:DataList>
```

(2) 商品详细信息页面 ProductDetails.aspx

和商品分类列表页面一样，商品详细信息页面同样使用了页面高速缓存，但是高速缓存更新查询字符串变成了 ProductID。

```
<%@ OutputCache Duration="60" VaryByParam="ProductID" %>
```

在商品详细信息页面 ProductDetails.aspx 中使用了几个服务器控件来显示商品信息，而服务器控件的值在后台编码类中获得。

```
<!-- 商品名称 -->
<asp:label id="ModelName" runat="server" />
<!-- 商品图片 -->
<asp:image id="ProductImage" runat="server" height="185" width="309" border="0" />
<!-- 商品价格 -->
<asp:label id="UnitCost" runat="server" />
<!-- 商品型号-->
<asp:label id="ModelNumber" runat="server" />
<!-- 加入购物车的链接 -->
<asp:hyperlink id="addToCart" runat="server" ImageUrl="images/add_to_cart.gif" />
<!-- 商品描述 -->
<asp:label class="NormalDouble" id="desc" runat="server"></asp:label>
```

(3) 商品搜索结果页面 SearchResults.aspx

商品搜索结果页面和商品分类列表页面一样都是在页面中以 DataList 的形式呈现后台编码类中的一个商品信息列表记录集，所以在前台页面的代码中几乎是相同的。包括 DataList 的定义、Repeater 模板的定义也都是完全一样的，因此在此不再重复。

(4) 热门商品列表用户控件_PopularItems.ascx

热门商品列表用户控件 PopularItems 也使用了页面高速缓存。

```
<%@ OutputCache Duration="3600" VaryByParam="None" %>
```

热门商品列表是通过名为 productList 的 Repeater 呈现出来的。以下为 Repeater 的代码清单。

```
<!-- 定义 Repeater 的 ID，这里定义的 ID 要和后台编码类中 Repeater 的 ID 相同 -->
<asp:Repeater ID="productList" runat="server">
    <!-- 定义 Repeater 头部模板 -->
    <HeaderTemplate>
        <tr>
            <td class="MostPopularHead">
                &nbsp;Our most popular items this week
            </td>
        </tr>
    </HeaderTemplate>
    <!-- 定义 Repeater 模板 -->
    <ItemTemplate>
        <tr>
            <td bgcolor="#d3d3d3">
                &nbsp;
                <asp:HyperLink class="MostPopularItemText" NavigateUrl='<%# "ProductDetails.aspx?ProductID=" + DataBinder.Eval(Container.DataItem, "ProductID")%>' Text='<%#DataBinder.Eval(Container.DataItem, "ModelName")%>' runat="server" />
                <br>
            </td>
        </tr>
    </ItemTemplate>
    <!-- 定义 Repeater 脚部模板 -->
    <FooterTemplate>
```

```

<tr>
    <td bgcolor="#d3d3d3">
        &nbsp;
    </td>
</tr>
</FooterTemplate>
</asp:Repeater>

```

(5) 购买某种商品的用户购买其他商品信息列表用户控件_AlsoBought. ascx

购买某种商品的用户购买其他商品信息列表用户控件 AlsoBought 和热门商品列表用户控件 PopularItems 一样，都是在页面上采用 Repeater 形式呈现后台编码类中商品名称列表数据集，因此两个用户控件前台页面十分类似，因此不再重复。

6.4 购物车

6.4.1 实现效果

在以下 3 种情况中用户会浏览到购物车页面。

- (1) 单击商品信息中的 Add To Cart 链接之后。
- (2) 单击页面导航栏的购物车按钮。
- (3) 用户登录或是注册之后。

购物车页面 ShoppingCart.aspx 的实现效果参见图 6-7。

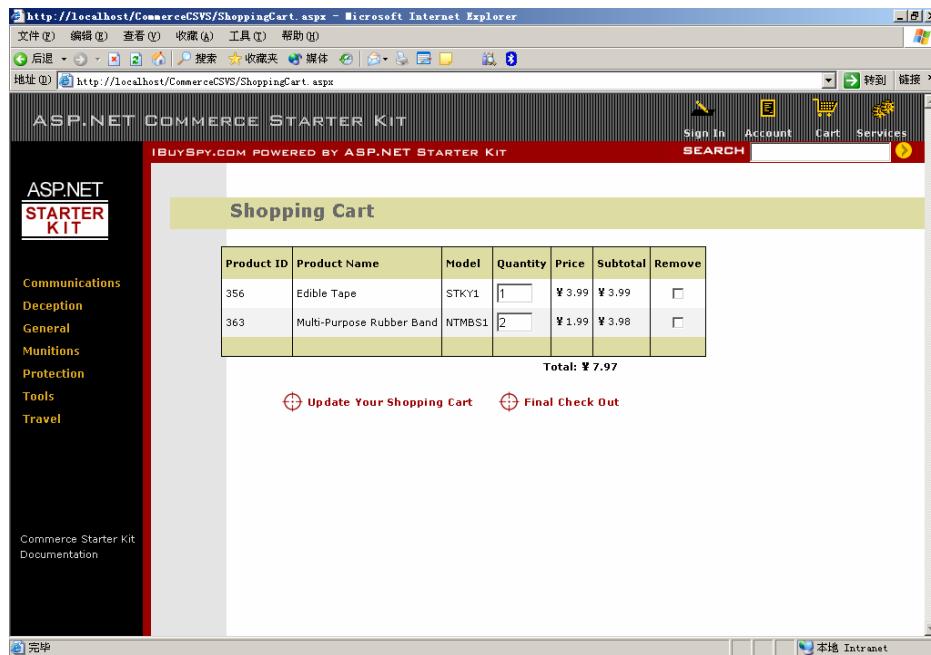


图 6-7 购物车页面 ShoppingCart.aspx 实现效果

购物车页面以列表的形式显示了当前用户购物车内的商品信息，包括商品名称、商品 ID、商品型

号、购买数量、商品单价、商品总价和订单价格总合。用户可以更改购买数量或是删除某个商品的购买信息。单击 Update Your Shopping Cart 链接之后更新对购物车信息的修改。单击 Final Check Out 链接之后跳转到确认购物信息页面 Checkout.aspx。确认购物信息页面 Checkout.aspx 的实现效果参见图 6-8。

确认商品信息页面会显示购物车中的信息，但是购物车中商品的信息并不可以修改，如果用户决定提交当前的订单，则单击 Submit 提交信息，完成一笔订单。如果用户还想对购物车中的信息进行修改，那么单击页面导航栏的 Cart 链接返回购物车页面。

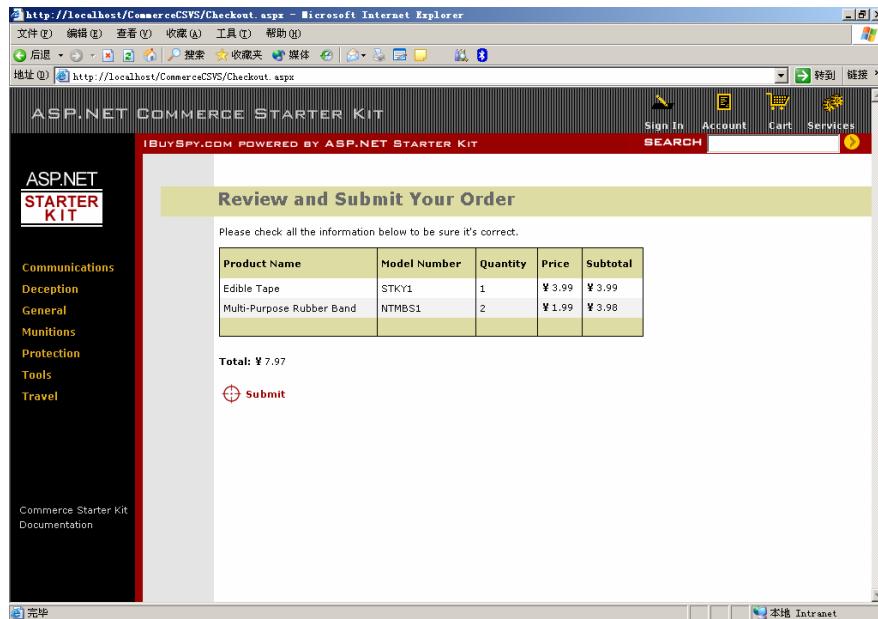


图 6-8 确认购物信息页面 Checkout.aspx 的实现效果

6.4.2 存储过程

购物车部分主要的存储过程有 9 个： CMRC_OrdersAdd、CMRC_ShoppingCartAddItem、CMRC_ShoppingCartEmpty、CMRC_ShoppingCartItemCount、CMRC_ShoppingCartList、CMRC_ShoppingCartMigrate、CMRC_ShoppingCartRemoveItem、CMRC_ShoppingCartTotal、CMRC_ShoppingCartUpdate。

(1) 存储过程 CMRC_OrdersAdd

该存储过程为向 CMRC_Orders 和 CMRC_OrdersDetail 表中添加订单信息的存储过程。该存储过程首先向 CMRC_Orders 添加一条包括用户 ID、购物车 ID、购买日期、送货日期信息的订单记录，并返回刚刚添加的订单 ID。然后从购物车表 CMRC_ShoppingCart 取得该订单的详细商品信息添加到 CMRC_OrdersDetail 中，最后调用清空用户购物车的存储过程清空用户的购物车。该存储过程被添加订单的数据访问层调用。以下是该存储过程的代码清单。

```

CREATE Procedure CMRC_OrdersAdd
(
    @CustomerID int,
    @CartID      nvarchar(50),
    @OrderDate   datetime,
    @ShipDate    datetime,
    @OrderID     int OUTPUT
)
AS

/* 声明存储过程代码块开始*/
BEGIN TRAN AddOrder

/* 向 CMRC_Orders 添加总的订单信息 */

```

```
INSERT INTO CMRC_Orders
(
    CustomerID,
    OrderDate,
    ShipDate
)
VALUES
(
    @CustomerID,
    @OrderDate,
    @ShipDate
)
/* 返回刚刚添加的订单的 OrderID */
SELECT
    @OrderID = @@Identity

/* 将从 CMRC_ShoppingCart 查询到的购物车详细商品信息添加到 CMRC_OrderDetails */
INSERT INTO CMRC_OrderDetails
(
    OrderID,
    ProductID,
    Quantity,
    UnitCost
)
/* 从 CMRC_ShoppingCart 查询用户购物车详细商品信息*/
SELECT
    @OrderID,
    CMRC_ShoppingCart.ProductID,
    Quantity,
    CMRC_Products.UnitCost
FROM
    CMRC_ShoppingCart
INNER JOIN CMRC_Products ON CMRC_ShoppingCart.ProductID = CMRC_Products.ProductID
WHERE
    CartID = @CartID

/* 调用清空购物车信息的存储过程，清空当前用户的购物车*/
EXEC CMRC_ShoppingCartEmpty @CartId
/* 声明存储过程代码块结束*/
COMMIT TRAN AddOrder
```

(2) 存储过程 CMRC_ShoppingCartAddItem

该存储过程为向用户购物车信息表 CMRC_ShoppingCart 中添加商品详细订购信息的存储过程。该存储过程首先判断所要添加的详细商品信息是否和目前购物车内的某一个详细商品信息重复。如果重复，则更新目前该条详细商品信息的订购数量加 1。如果目前购物车没有所添加的详细商品信息，则添加一条详细商品信息到购物车表 CMRC_ShoppingCart 中。该存储过程被向用户购物车中添加详细产品订购信息的数据访问层调用。以下是该存储过程的代码清单。

```
CREATE Procedure CMRC_ShoppingCartAddItem
(
    @CartID nvarchar(50),
```

```
    @ProductID int,
    @Quantity int
)
AS
/* 声明变量 CountItems */
DECLARE @CountItems int
/* 查询要添加的商品是否在目前用户购物车内已经存在 */
SELECT
    @CountItems = Count (ProductID)
FROM
    CMRC_ShoppingCart
WHERE
    ProductID = @ProductID
AND
    CartID = @CartID
/* 如果要添加的商品在目前购物车内已经存在，则更新该条商品订购信息*/
IF @CountItems > 0
    UPDATE
        CMRC_ShoppingCart
    SET
        Quantity = (@Quantity + CMRC_ShoppingCart.Quantity)
    WHERE
        ProductID = @ProductID
        AND
        CartID = @CartID
/* 如果要添加的商品在目前购物车内不存在，则添加该条商品订购信息*/
ELSE
    INSERT INTO CMRC_ShoppingCart
    (
        CartID,
        Quantity,
        ProductID
    )
    VALUES
    (
        @CartID,
        @Quantity,
        @ProductID
    )
```

(3) 存储过程 CMRC_ShoppingCartEmpty

该存储过程是清空目前购物车内的订购信息的存储过程。该存储过程被存储过程 CMRC_OrdersAdd 调用，在完成一笔订单之后，必须清空用户的购物车信息，这样用户在完成订单下次使用购物车的时候购物车内没有其他的商品订购信息。以下是该存储过程的代码清单。

```
CREATE Procedure CMRC_ShoppingCartEmpty
(
    @CartID nvarchar(50)
)
AS
/* 删除 CartID 为传入参数@CartID 的所有购物车信息*/
DELETE FROM CMRC_ShoppingCart
```

```
WHERE
```

```
    CartID = @CartID
```

(4) 存储过程 CMRC_ShoppingCartItemCount

该存储过程是获得当前购物车内商品种类总和的存储过程。以下是该存储过程的代码清单。

```
CREATE Procedure CMRC_ShoppingCartItemCount
(
    @CartID      nvarchar(50),
    @ItemCount  int OUTPUT
)
AS
/* 获得 CartID 为传入参数@CartID 的购物车内商品种类总和*/
SELECT
    @ItemCount = COUNT(ProductID)
FROM
    CMRC_ShoppingCart
WHERE
    CartID = @CartID
```

(5) 存储过程 CMRC_ShoppingCartList

该存储过程是获得当前购物车内商品详细订购信息的存储过程。根据购物车表 CMRC_ShoppingCart 中的 ProductID 获得被订购商品的详细信息，并根据订购数量和价格计算出订购该项商品的总价。该存储过程被获得某个用户购物车信息数据集的数据访问层调用。以下是该存储过程的代码清单。

```
CREATE Procedure CMRC_ShoppingCartList
(
    @CartID  nvarchar(50)
)
AS
/* 获得 CartID 为传入参数@CartID 的商品详细订购信息*/
SELECT
    /* 从商品信息表 CMRC_Products 获得商品的 ProductID */
    CMRC_Products.ProductID,
    /* 从商品信息表 CMRC_Products 获得商品的型号 */
    CMRC_Products.ModelName,
    /* 从商品信息表 CMRC_Products 获得商品的 ModelNumber */
    CMRC_Products.ModelNumber,
    /* 从购物车表 CMRC_ShoppingCart 获得商品的订购数量 */
    CMRC_ShoppingCart.Quantity,
    /* 从商品信息表 CMRC_Products 获得商品的单价 */
    CMRC_Products.UnitCost,
    /* 根据商品的订购数量和总价得出该项商品的总价格 */
    Cast((CMRC_Products.UnitCost * CMRC_ShoppingCart.Quantity) as money) as ExtendedAmount
FROM
    CMRC_Products,
    CMRC_ShoppingCart
WHERE
    CMRC_Products.ProductID = CMRC_ShoppingCart.ProductID
    AND
        CMRC_ShoppingCart.CartID = @CartID
    /* 根据商品的名称和型号进行排序 */
ORDER BY
```

```
CMRC_Products.ModelName,
CMRC_Products.ModelNumber
```

(6) 存储过程 CMRC_ShoppingCartMigrate

该存储过程是将一个购物车的信息转移给另外一个购物车的存储过程。存储过程接收两个购物车的 CartId，并将其中一个购物车的 CartId 设置为另外一个购物车的 CartId。以下是该存储过程的代码清单。

```
CREATE Procedure CMRC_ShoppingCartMigrate
(
    @OriginalCartId nvarchar(50),
    @NewCartId      nvarchar(50)
)
AS
/* 更新 CMRC_ShoppingCart 表中的 CartId*/
UPDATE
    CMRC_ShoppingCart
SET
    CartId = @NewCartId
WHERE
    CartId = @OriginalCartId
```

(7) 存储过程 CMRC_ShoppingCartRemoveItem

该存储过程是删除目前购物车内某条商品详细购物信息的存储过程。存储过程删除根据参数 CartID 和 ProductID 确定的 CMRC_ShoppingCart 表中某条详细购物信息。以下是该存储过程的代码清单。

```
CREATE Procedure CMRC_ShoppingCartRemoveItem
(
    @CartID nvarchar(50),
    @ProductID int
)
AS
/* 删除 CMRC_ShoppingCart 表中指定 CartID 和 ProductID 对应的记录 */
DELETE FROM CMRC_ShoppingCart
WHERE
    CartID = @CartID
    AND
    ProductID = @ProductID
```

(8) 存储过程 CMRC_ShoppingCartTotal

该存储过程是返回某个购物车内所有购物信息价格总和的存储过程。该存储过程查询参数 CartID 所对应的购物车详细信息表中所有购物信息的总和并返回。以下是该存储过程的代码清单。

```
CREATE Procedure CMRC_ShoppingCartTotal
(
    @CartID      nvarchar(50),
    @TotalCost   money OUTPUT
)
AS
/* 将购物车 CartID 对应的每条记录商品的价格 UnitCost 和采购数量 Quantity 相乘之后再求总和 */
SELECT
    @TotalCost = SUM(CMRC_Products.UnitCost * CMRC_ShoppingCart.Quantity)
FROM
    CMRC_ShoppingCart,
    CMRC_Products
WHERE
```

```
CMRC_ShoppingCart.CartID = @CartID
```

AND

```
CMRC_Products.ProductID = CMRC_ShoppingCart.ProductID
```

(9) 存储过程 CMRC_ShoppingCartUpdate

该存储过程是更新某个购物车中某条商品详细购物信息数量的存储过程。该存储过程更新参数 CartID 和 ProductID 所对应购物详细信息记录的数量 Quantity。以下是该存储过程的代码清单。

```
CREATE Procedure CMRC_ShoppingCartUpdate
(
    @CartID      nvarchar(50),
    @ProductID   int,
    @Quantity    int
)
AS
/* 更新参数 CartID 和 ProductID 所对应 CMRC_ShoppingCart 表中记录的 Quantity */
UPDATE CMRC_ShoppingCart
SET
    Quantity = @Quantity
WHERE
    CartID = @CartID
    AND
    ProductID = @ProductID
```

6.4.3 数据访问层

购物车模块的数据访问层为 Components 文件夹下的 ShoppingCartDB.cs 文件所提供的类 ASPNET.StarterKit.Commerce.ShoppingCartDB。该数据访问层提供了购物车模块业务逻辑层所使用的所有方法。以下是该数据访问层的代码清单。

```
using System;
using System.Configuration;
using System.Data;
using System.Data.SqlClient;

namespace ASPNET.StarterKit.Commerce {
```

```
    public class ShoppingCartDB {
```

GetItems 方法是返回购物车所有详细购物信息的数据集的方法。GetItem 方法首先创建了数据库连接和数据库命令，并指定数据库命令执行存储过程 CMRC_ShoppingCartList，然后创建了一个 SqlParameter 类型的变量 parameterCartID，然后将参数方法的参数 cartID 的值赋给 parameterCartID，最后将 parameterCartID 添加到数据库命令的参数表中。最后执行数据库命令获得 SqlDataReader 类型的变量 result，并将 result 返回。

```
    public SqlDataReader GetItems(string cartID) {
        //实例化一个数据库连接对象 myConnection
        SqlConnection          myConnection           = new SqlConnection(ConfigurationSettings.AppSettings["ConnectionString"]);
        //实例化一个使用 myConnection 执行存储过程 CMRC_ShoppingCartList 的数据库命令 myCommand
        SqlCommand  myCommand = new SqlCommand("CMRC_ShoppingCartList", myConnection);
        //设置 myCommand 的类型为执行存储过程
        myCommand.CommandType = CommandType.StoredProcedure;
```

```
//定义一个 SqlParameter 类型的存储过程参数 parameterCartID  
SqlParameter parameterCartID = new SqlParameter("@CartID", SqlDbType.NVarChar, 50);  
//将方法的参数 cartID 的值赋给 parameterCartID  
parameterCartID.Value = cartID;  
//将 parameterCartID 添加到数据库命令 myCommand 的参数表中去  
myCommand.Parameters.Add(parameterCartID);  
//打开数据库连接 myConnection  
myConnection.Open();  
//执行 myCommand 并获得 SqlDataReader 类型的记录集 result  
SqlDataReader result = myCommand.ExecuteReader(CommandBehavior.CloseConnection);  
//返回 result  
return result;  
}
```

AddItem 方法是向购物车内添加一条购物信息使用的方法。AddItem 方法首先创建了数据库连接和数据库命令，并指定数据库命令执行存储过程 CMRC_ShoppingCartAddItem，然后创建了三个 SqlParameter 类型的变量 parameterProductID、parameterCartID、parameterQuantity，然后将方法的参数 cartID 值赋给 parameterCartID，将方法的参数 productID 值赋给 parameterProductID，将方法的参数 quantity 值赋给 parameterQuantity。最后将三个 SqlParameter 类型的变量添加到数据库命令的参数表中。最后打开数据连接，执行数据库命令，最终关闭数据库连接。

```
public void AddItem(string cartID, int productID, int quantity) {  
    //实例化一个数据库连接对象 myConnection  
    SqlConnection myConnection = new SqlConnection(ConfigurationSettings.AppSettings["ConnectionString"]);  
    //实例化一个使用 myConnection 执行存储过程 CMRC_ShoppingCartAddItem 的数据库命令 myCommand  
    SqlCommand myCommand = new SqlCommand("CMRC_ShoppingCartAddItem", myConnection);  
    //设置 myCommand 的类型为执行存储过程  
    myCommand.CommandType = CommandType.StoredProcedure;  
  
    //定义一个 SqlParameter 类型的存储过程参数 parameterProductID  
    SqlParameter parameterProductID = new SqlParameter("@ProductID", SqlDbType.Int, 4);  
    //将方法的参数 productID 的值赋给 parameterProductID  
    parameterProductID.Value = productID;  
    //将 parameterCartID 添加到数据库命令 myCommand 的参数表中去  
    myCommand.Parameters.Add(parameterProductID);  
    //定义一个 SqlParameter 类型的存储过程参数 parameterCartID  
    SqlParameter parameterCartID = new SqlParameter("@CartID", SqlDbType.NVarChar, 50);  
    //将方法的参数 cartID 的值赋给 parameterCartID  
    parameterCartID.Value = cartID;  
    //将 parameterCartID 添加到数据库命令 myCommand 的参数表中去  
    myCommand.Parameters.Add(parameterCartID);  
    //定义一个 SqlParameter 类型的存储过程参数 parameterQuantity  
    SqlParameter parameterQuantity = new SqlParameter("@Quantity", SqlDbType.Int, 4);  
    //将方法的参数 quantity 的值赋给 parameterQuantity  
    parameterQuantity.Value = quantity;  
    //将 parameterQuantity 添加到数据库命令 myCommand 的参数表中去  
    myCommand.Parameters.Add(parameterQuantity);  
  
    //打开数据库连接 myConnection
```

```
myConnection.Open();
//调用数据库命令 myCommand 的 ExecuteNonQuery 方法执行数据库命令
myCommand.ExecuteNonQuery();
//关闭数据库连接 myConnection
myConnection.Close();
}
```

UpdateItem 方法是修改购物车内一条购物信息使用的方法。UpdateItem 首先检查参数表中的参数 quantity 是否是一个合法的数值，从业务逻辑上讲，购买商品的数量应该是一个大于 0 的整数。如果参数 quantity 不是一个复合业务逻辑的合法数值，则抛出一个异常。UpdateItem 方法而后创建了数据库连接和数据库命令，并指定数据库命令执行存储过程 CMRC_ShoppingCartUpdate，然后创建了 3 个 SqlParameter 类型的变量 parameterProductID、parameterCartID、parameterQuantity，然后将方法的参数 cartID 值赋给 parameterCartID，将方法的参数 productID 值赋给 parameterProductID，将方法的参数 quantity 值赋给 parameterQuantity。将 3 个 SqlParameter 类型的变量添加到数据库命令的参数表中。接下来打开数据连接，执行数据库命令，最后关闭数据库连接。

```
public void UpdateItem(string cartID, int productID, int quantity) {
    //检查方法的参数 quantity 是否是大于 0 的合法数值
    if (quantity < 0) {
        //如果 quantity 不是合法数值则抛出一个异常
        throw new Exception("Quantity cannot be a negative number");
    }

    //实例化一个数据库连接对象 myConnection
    SqlConnection myConnection = new SqlConnection(ConfigurationSettings.AppSettings["ConnectionString"]);
    //实例化一个使用 myConnection 执行存储过程 CMRC_ShoppingCartUpdate 的数据库命令 myCommand
    SqlCommand myCommand = new SqlCommand("CMRC_ShoppingCartUpdate", myConnection);
    //设置 myCommand 的类型为执行存储过程
    myCommand.CommandType = CommandType.StoredProcedure;

    //定义一个 SqlParameter 类型的存储过程参数 parameterProductID
    SqlParameter parameterProductID = new SqlParameter("@ProductID", SqlDbType.Int, 4);
    //将方法的参数 productID 的值赋给 parameterProductID
    parameterProductID.Value = productID;
    //将 parameterCartID 添加到数据库命令 myCommand 的参数表中去
    myCommand.Parameters.Add(parameterProductID);
    //定义一个 SqlParameter 类型的存储过程参数 parameterCartID
    SqlParameter parameterCartID = new SqlParameter("@CartID", SqlDbType.NVarChar, 50);
    //将方法的参数 cartID 的值赋给 parameterCartID
    parameterCartID.Value = cartID;
    //将 parameterCartID 添加到数据库命令 myCommand 的参数表中去
    myCommand.Parameters.Add(parameterCartID);
    //定义一个 SqlParameter 类型的存储过程参数 parameterQuantity
    SqlParameter parameterQuantity = new SqlParameter("@Quantity", SqlDbType.Int, 4);
    //将方法的参数 quantity 的值赋给 parameterQuantity
    parameterQuantity.Value = quantity;
    //将 parameterQuantity 添加到数据库命令 myCommand 的参数表中去
    myCommand.Parameters.Add(parameterQuantity);

    //打开数据库连接 myConnection
}
```

```

    myConnection.Open();
    //调用数据库命令 myCommand 的 ExecuteNonQuery 方法执行数据库命令
    myCommand.ExecuteNonQuery();
    //关闭数据库连接 myConnection
    myConnection.Close();
}

```

RemoveItem 方法是删除购物车内一条购物信息使用的方法。RemoveItem 方法首先创建了数据库连接和数据库命令，并指定数据库命令执行存储过程 CMRC_ShoppingCartRemoveItem，创建了两个 SqlParameter 类型的变量：parameterProductID、parameterCartID，然后将方法的参数 cartID 值赋给 parameterCartID，将方法的参数 productID 值赋给 parameterProductID。将两个 SqlParameter 类型的变量添加到数据库命令的参数表中。接下来打开数据连接，执行数据库命令，最后关闭数据库连接。

```

public void RemoveItem(string cartID, int productID) {
    //实例化一个数据库连接对象 myConnection
    SqlConnection myConnection = new SqlConnection(
        SqlConnection.ConfigurationSettings.AppSettings["ConnectionString"]);
    //实例化一个使用 myConnection 执行存储过程 CMRC_ShoppingCartRemoveItem 的数据库命令 myCommand
    SqlCommand myCommand = new SqlCommand("CMRC_ShoppingCartRemoveItem", myConnection);
    //设置 myCommand 的类型为执行存储过程
    myCommand.CommandType = CommandType.StoredProcedure;

    //定义一个 SqlParameter 类型的存储过程参数 parameterProductID
    SqlParameter parameterProductID = new SqlParameter("@ProductID", SqlDbType.Int, 4);
    //将方法的参数 productID 的值赋给 parameterProductID
    parameterProductID.Value = productID;
    //将 parameterCartID 添加到数据库命令 myCommand 的参数表中去
    myCommand.Parameters.Add(parameterProductID);
    //定义一个 SqlParameter 类型的存储过程参数 parameterCartID
    SqlParameter parameterCartID = new SqlParameter("@CartID", SqlDbType.NVarChar, 50);
    //将方法的参数 cartID 的值赋给 parameterCartID
    parameterCartID.Value = cartID;
    //将 parameterCartID 添加到数据库命令 myCommand 的参数表中去
    myCommand.Parameters.Add(parameterCartID);

    //打开数据库连接 myConnection
    myConnection.Open();
    //调用数据库命令 myCommand 的 ExecuteNonQuery 方法执行数据库命令
    myCommand.ExecuteNonQuery();
    //关闭数据库连接 myConnection
    myConnection.Close();
}

```

GetItemCount 方法是获得购物车内购物信息总数时使用的方法。GetItemCount 方法首先创建了数据库连接和数据库命令，并指定数据库命令执行存储过程 CMRC_ShoppingCartItemCount，创建了两个 SqlParameter 类型的变量 parameterCartID、parameterItemCount，然后将方法的参数 cartID 值赋给 parameterCartID，parameterItemCount 设置成返回值型存储过程参数。将两个 SqlParameter 类型的变量添加到数据库命令的参数表中，然后打开数据连接，执行数据库命令之后关闭数据库连接。最终返回 parameterItemCount 的值。

```

public int GetItemCount(string cartID) {
    //实例化一个数据库连接对象 myConnection

```

```
SqlConnection myConnection = new SqlConnection(ConfigurationSettings.AppSettings["ConnectionString"]);
//实例化一个使用 myConnection 执行存储过程 CMRC_ShoppingCartItemCount 的数据库命令 myCommand
SqlCommand myCommand = new SqlCommand("CMRC_ShoppingCartItemCount", myConnection);
//设置 myCommand 的类型为执行存储过程
myCommand.CommandType = CommandType.StoredProcedure;

//定义一个 SqlParameter 类型的存储过程参数 parameterProductID
SqlParameter parameterProductID = new SqlParameter("@ProductID", SqlDbType.Int, 4);
//将方法的参数 productID 值赋给 parameterProductID
parameterProductID.Value = productID;
//将 parameterCartID 添加到数据库命令 myCommand 的参数表中去
myCommand.Parameters.Add(parameterProductID);
//定义一个 SqlParameter 类型的存储过程参数 parameterItemCount
SqlParameter parameterItemCount = new SqlParameter("@ItemCount", SqlDbType.Int, 4);
//将 parameterItemCount 设置为返回值型存储过程参数
parameterItemCount.Direction = ParameterDirection.Output;
//将 parameterCartID 添加到数据库命令 myCommand 的参数表中去
myCommand.Parameters.Add(parameterItemCount);

//打开数据库连接 myConnection
myConnection.Open();
//调用数据库命令 myCommand 的 ExecuteNonQuery 方法执行数据库命令
myCommand.ExecuteNonQuery();
//关闭数据库连接 myConnection
myConnection.Close();
//返回存储过程参数 parameterItemCount
return ((int)parameterItemCount.Value);
}
```

GetTotal 方法是获得购物车内所有购物信息金额总和的方法。GetTotal 方法首先创建了数据库连接和数据库命令，并指定数据库命令执行存储过程 CMRC_ShoppingCartTotal，创建了两个 SqlParameter 类型的变量：parameterCartID、parameterTotalCost，然后将方法的参数 cartID 值赋给 parameterCartID，parameterTotalCost 设置成返回值型存储过程参数，将两个 SqlParameter 类型的变量添加到数据库命令的参数表中。然后打开数据连接，执行数据库命令之后关闭数据库连接。最后判定 parameterTotalCost 的值是否为空，如果不为空，则将 parameterTotalCost 转化成 decimal 类型并返回；如果 parameterTotalCost 为空，则返回 0。

```
public decimal GetTotal(string cartID) {
    //实例化一个数据库连接对象 myConnection
    SqlConnection myConnection = new SqlConnection(ConfigurationSettings.AppSettings["ConnectionString"]);
    //实例化一个使用 myConnection 执行存储过程 CMRC_ShoppingCartTotal 的数据库命令 myCommand
    SqlCommand myCommand = new SqlCommand("CMRC_ShoppingCartTotal", myConnection);
    //设置 myCommand 的类型为执行存储过程
    myCommand.CommandType = CommandType.StoredProcedure;

    //定义一个 SqlParameter 类型的存储过程参数 parameterProductID
    SqlParameter parameterProductID = new SqlParameter("@ProductID", SqlDbType.Int, 4);
    //将方法的参数 productID 的值赋给 parameterProductID
    parameterProductID.Value = productID;
```

```

//将 parameterCartID 添加到数据库命令 myCommand 的参数表中去
myCommand.Parameters.Add(parameterProductID);
//定义一个 SqlParameter 类型的存储过程参数 parameterTotalCost
SqlParameter parameterTotalCost = new SqlParameter("@TotalCost", SqlDbType.Money, 8);
//将 parameterTotalCost 设置为返回值型存储过程参数
parameterTotalCost.Direction = ParameterDirection.Output;
//将 parameterTotalCost 添加到数据库命令 myCommand 的参数表中去
myCommand.Parameters.Add(parameterTotalCost);

//打开数据库连接 myConnection
myConnection.Open();
//调用数据库命令 myCommand 的 ExecuteNonQuery 方法执行数据库命令
myCommand.ExecuteNonQuery();
//关闭数据库连接 myConnection
myConnection.Close();

//判定 parameterTotalCost 是否为空
if (parameterTotalCost.Value.ToString() != "") {
    //如果 parameterTotalCost 不为空，则将 parameterTotalCost 隐式转化成 decimal 类型变量并返回
    return (decimal)parameterTotalCost.Value;
}
else {
    //如果 parameterTotalCost 为空，则返回 0
    return 0;
}
}

```

MigrateCart 方法是将一个购物车的信息转移到另外一个购物车中使用的方法。MigrateCart 方法首先创建了数据库连接和数据库命令，并指定数据库命令执行存储过程 CMRC_ShoppingCartMigrate，创建了两个 SqlParameter 类型的变量：cart1、cart2，然后将方法的参数 oldCartId 值赋给 cart1，将方法参数 newCartId 的值赋给 cart2。之后将两个 SqlParameter 类型的变量添加到数据库命令的参数表中。最后打开数据连接，执行数据库命令之后关闭数据库连接。

```

public void MigrateCart(String oldCartId, String newCartId) {
    //实例化一个数据库连接对象 myConnection
    SqlConnection myConnection = new SqlConnection(ConfigurationSettings.AppSettings["ConnectionString"]);
    //实例化一个使用 myConnection 执行存储过程 CMRC_ShoppingCartMigrate 的数据库命令 myCommand
    SqlCommand myCommand = new SqlCommand("CMRC_ShoppingCartMigrate", myConnection);
    //设置 myCommand 的类型为执行存储过程
    myCommand.CommandType = CommandType.StoredProcedure;

    //定义一个 SqlParameter 类型的存储过程参数 cart1
    SqlParameter cart1 = new SqlParameter("@OriginalCartId ", SqlDbType.NVarChar, 50);
    //将方法的参数 oldCartId 的值赋给 cart1
    cart1.Value = oldCartId;
    //将 cart1 添加到数据库命令 myCommand 的参数表中去
    myCommand.Parameters.Add(cart1);
    //定义一个 SqlParameter 类型的存储过程参数 cart2
    SqlParameter cart2 = new SqlParameter("@NewCartId ", SqlDbType.NVarChar, 50);
    //将方法的参数 newCartId 的值赋给 cart2
}

```

```
cart2.Value = newCartId;
//将 cart2 添加到数据库命令 myCommand 的参数表中去
myCommand.Parameters.Add(cart2);

//打开数据库连接 myConnection
myConnection.Open();
//调用数据库命令 myCommand 的 ExecuteNonQuery 方法执行数据库命令
myCommand.ExecuteNonQuery();
//关闭数据库连接 myConnection
myConnection.Close();
}
```

EmptyCart 方法是删除购物车内所有详细购物信息所使用的方法。EmptyCart 方法首先创建了数据库连接和数据库命令，并指定数据库命令执行存储过程 CMRC_ShoppingCartEmpty，创建了一个 SqlParameter 类型的变量 cartid，将方法的参数 cartID 值赋给 cartid。然后将这个 SqlParameter 类型的变量添加到数据库命令的参数表中。最后打开数据连接，执行数据库命令之后关闭数据库连接。

```
public void EmptyCart(string cartID) {
    //实例化一个数据库连接对象 myConnection
    SqlConnection myConnection = new SqlConnection(ConfigurationSettings.AppSettings["ConnectionString"]);
    //实例化一个使用 myConnection 执行存储过程 CMRC_ShoppingCartEmpty 的数据库命令 myCommand
    SqlCommand myCommand = new SqlCommand("CMRC_ShoppingCartEmpty", myConnection);
    //设置 myCommand 的类型为执行存储过程
    myCommand.CommandType = CommandType.StoredProcedure;

    //定义一个 SqlParameter 类型的存储过程参数 cartid
    SqlParameter cartid = new SqlParameter("@CartID", SqlDbType.NVarChar, 50);
    //将方法的参数 cartID 值赋给 cartid
    cartid.Value = cartID;
    //将 cartid 添加到数据库命令 myCommand 的参数表中去
    myCommand.Parameters.Add(cartid);

    //打开数据库连接 myConnection
    myConnection.Open();
    //调用数据库命令 myCommand 的 ExecuteNonQuery 方法执行数据库命令
    myCommand.ExecuteNonQuery();
    //关闭数据库连接 myConnection
    myConnection.Close();
}
```

GetShoppingCartId 方法是获得当前用户的购物车 ID 的方法。GetShoppingCartId 方法首先实例化当前 HttpContext 对象，并试图获得 context.User.Identity.Name，即当前用户的名称。只有通过系统身份验证的用户在 HttpContext 对象中才存有名称，如果该名称存在，则返回该名称；如果该 context.User.Identity.Name 不存在即前用户没有通过系统的身份验证则试图获取名为 ASPNETCommerce_CartID 的 Cookies。如果该 Cookies 不为空，则返回该 Cookies 的值；如果名为 ASPNETCommerce_CartID 的值为空，则随机产生一个 Guid 作为返回值，并将名为 ASPNETCommerce_CartID 的 Cookies 值设置为随机产生的 Guid。

```
public String GetShoppingCartId() {
    //实例化当前用户的 System.Web.HttpContext
    System.Web.HttpContext context = System.Web.HttpContext.Current;
```

```
//获取 context.User.Identity.Name 判断当前用户是否已经通过系统身份验证
if (context.User.Identity.Name != "") {
    //如果通过系统身份验证，则返回当前用户的 context.User.Identity.Name
    return context.User.Identity.Name;
}

//获取名为 ASPNETCommerce_CartID 的 Cookies 并判断是否为空
if (context.Request.Cookies["ASPNETCommerce_CartID"] != null) {
    //如果名为 ASPNETCommerce_CartID 的 Cookies 不为空，则返回其值
    return context.Request.Cookies["ASPNETCommerce_CartID"].Value;
}
else {
    //随机产生一个 Guid
    Guid tempCartId = Guid.NewGuid();
    //设置名为 ASPNETCommerce_CartID 的 Cookies 值为随机产生的 Guid
    context.Response.Cookies["ASPNETCommerce_CartID"].Value = tempCartId.ToString();
    //返回随机产生的 Guid
    return tempCartId.ToString();
}
}
```

6.4.4 业务逻辑层

(1) 加入购物车页面后台编码类 AddToCart. aspx. cs

加入购物车页面 AddToCart.aspx 的后台编码类为 AddToCart.aspx.cs。和其他页面不同的是，加入购物车页面的用户表示层并没有任何页面表示元素代码，加入购物车页面的工作原理是接收 URL 的参数将所有逻辑处理都放在后台编码类，在处理业务逻辑结束之后跳转到购物车页面。加入购物车页面像一座桥梁，连接商品信息页面和购物车页面，完成将商品信息加入购物车这个动作，而加入购物车页面本身并不会直接展现给用户。

由于加入购物车页面的特殊工作原理，所以在后台编码类的页面装载方法 Page_Load 集中该类的全部业务逻辑实现。该方法首先取得 URL 参数 ProductID，实例化数据访问层并调用 GetShoppingCartId 方法取得当前用户的购物车 cartId，调用方法 AddItem 向购物车中添加详细购物信息，最后跳转到购物车页面 ShoppingCart.aspx。以下是 Page_Load 方法的代码清单。

```
private void Page_Load(object sender, System.EventArgs e) {
    //判断页面 URL 中参数 ProductID 是否为空
    if (Request.Params["ProductID"] != null) {
        //实例化购物车模块的数据访问层类 ShoppingCartDB 成对象 cart
        ASPNET.StarterKit.Commerce.ShoppingCartDB    cart    =    new    ASPNET.StarterKit.Commerce.
ShoppingCartDB();
        //调用 GetShoppingCartId 方法获取当前用户的购物车 cartId
        String cartId = cart.GetShoppingCartId();

        //向当前用户的购物车中添加一条详细购物信息
        cart.AddItem(cartId, Int32.Parse(Request.Params["ProductID"]), 1);
    }
    //跳转页面到购物车页面 ShoppingCart.aspx
}
```

```
        Response.Redirect("ShoppingCart.aspx");
    }
```

(2) 购物车页面后台编码类 ShoppingCart.aspx.cs

购物车页面 ShoppingCart.aspx 的后台编码类为 ShoppingCart.aspx.cs。该页面编码类包含了对购物车详细购物信息的显示、更新、删除和提交的方法，而购物车详细购物信息在页面上的体现则为 DataGrid 控件 MyList，因此在 ShoppingCart.aspx.cs 中提供了对 MyList 的操作方法。以下是该后台编码类的代码清单。

```
using System;
using System.Collections;
using System.ComponentModel;
using System.Data;
using System.Data.SqlClient;
using System.Drawing;
using System.Web;
using System.Web.UI;
using System.Web.UI.WebControls;
using System.Web.UI.HtmlControls;

namespace ASPNET.StarterKit.Commerce {

    public class ShoppingCart : System.Web.UI.Page {
        //定义页面中显示错误信息的控件 MyError
        protected System.Web.UI.WebControls.Label MyError;
        //定义页面中 DataGrid 控件 MyList
        protected System.Web.UI.WebControls.DataGrid MyList;
        //定义页面中显示价格汇总的 Label 控件 lblTotal
        protected System.Web.UI.WebControls.Label lblTotal;
        //定义页面中更新购物车的 ImageButton 控件 UpdateBtn
        protected System.Web.UI.WebControls.ImageButton UpdateBtn;
        //定义页面中确认购物信息的 ImageButton 控件 CheckoutBtn
        protected System.Web.UI.WebControls.ImageButton CheckoutBtn;
        protected System.Web.UI.WebControls.Panel DetailsPanel;
        //类构造函数
        public ShoppingCart() {
            Page.Init += new System.EventHandler(Page_Init);
        }
    }
}
```

页面的装载方法 Page_Load 是用来装载页面内部的 DataGrid 控件的。该方法首先取得 Page.IsPostBack 的值判断页面是否是第一次被请求，如果页面是第一次被请求，则调用 PopulateShoppingCartList 方法。PopulateShoppingCartList 方法是绑定 DataGrid 控件的具体方法，将会在后面进行介绍。

```
private void Page_Load(object sender, System.EventArgs e) {
    //获得 Page.IsPostBack 值判断页面是否是第一次被请求
    if (Page.IsPostBack == false) {
        //调用 PopulateShoppingCartList
        PopulateShoppingCartList();
    }
}
```

UpdateBtn_Click 方法为页面中按钮 UpdateBtn 的单击事件方法。当用户单击 UpdateBtn 的 ImageButton 之后触发该方法。该方法调用了其他两个方法：UpdateShoppingCartDatabase 和

PopulateShoppingCartList。

```
private void UpdateBtn_Click(object sender, System.Web.UI.ImageClickEventArgs e) {
    //调用 UpdateShoppingCartDatabase 方法
    UpdateShoppingCartDatabase();
    //调用 PopulateShoppingCartList 方法
    PopulateShoppingCartList();
}
```

UpdateBtn_Click 方法为页面中按钮 UpdateBtn 的单击事件方法。当用户单击 UpdateBtn 的 ImageButton 之后触发该方法。该方法首先调用了 UpdateShoppingCartDatabase 方法，然后实例化购物车模块的数据访问层类 ShoppingCartDB 成为对象 cart，然后调用对象 cart 的 GetShoppingCartId 方法获得当前购物车的 CartID，调用 cart 对象的 GetItemCount 方法判断当前购物车内是否有购物信息，如果有购物信息，则跳转页面到 Checkout.aspx 页面；如果没有，则显示错误信息。

```
private void CheckoutBtn_Click(object sender, System.Web.UI.ImageClickEventArgs e) {
    //调用 UpdateShoppingCartDatabase 方法更新购物车信息
    UpdateShoppingCartDatabase();
    //实例化 ShoppingCartDB 类成为对象 cart
    ASPNET.StarterKit.Commerce.ShoppingCartDB    cart    =    new    ASPNET.StarterKit.Commerce.
ShoppingCartDB();
    //获得当前用户的购物车 CartID
    String cartId = cart.GetShoppingCartId();
    //判断当前购物车内购物商品的数量
    if (cart.GetItemCount(cartId) !=0) {
        //若购物车内购物商品的数量不为 0，则跳转到 Checkout.aspx
        Response.Redirect("Checkout.aspx");
    }
    else {
        //若购物车内购物商品的数量为 0，则显示错误信息
        MyError.Text = "You can't proceed to the Check Out page with an empty cart.";
    }
}
```

PopulateShoppingCartList 方法是绑定页面 DataGrid 控件的方法。首先实例化购物车模块的数据访问层类 ShoppingCartDB 成为对象 cart，然后获得当前用户购物车 CartID，判断当前用户购物车是否有购物信息。如果没有购物信息，则不显示购物车详细信息；如果有购物信息，则显示购物车详细信息。

```
void PopulateShoppingCartList() {
    //实例化 ShoppingCartDB 类成为对象 cart
    ASPNET.StarterKit.Commerce.ShoppingCartDB    cart    =    new    ASPNET.StarterKit.Commerce.
ShoppingCartDB();
    //获得当前用户的购物车 CartID
    String cartId = cart.GetShoppingCartId();
    //判断当前购物车内购物商品的数量
    if (cart.GetItemCount(cartId) == 0) {
        //如果当前购物车内购物商品数量为 0，则提示错误信息，并且不显示购物车详细信息区域
        DetailsPanel
            DetailsPanel.Visible = false;
            MyError.Text = "There are currently no items in your shopping cart.";
    }
    else {
        //如果当前购物车内购物商品数量为 0，则获得当前购物车详细购物信息的记录集
        //设置购物车信息的 DataGrid 的数据源
    }
}
```

```
MyList.DataSource = cart.GetItems(cartId);
//绑定购物车信息的DataGrid
MyList.DataBind();
//设置页面的购物车商品总价格的lblTotal
lblTotal.Text = String.Format("{0:c}", cart.GetTotal(cartId));
}
}
```

UpdateShoppingCartDatabase 方法是更新当前购物车内详细购物信息的方法。首先实例化购物车模块的数据访问层类 ShoppingCartDB 成为对象 cart，获得当前用户的购物车 CartId，然后遍历页面中 DataGrid 的每一条购物信息，获取每一条购物信息的购物数量的 TextBox 和是否删除的 CheckBox 值，根据获取的信息更新当前购物车中的数据。

```
void UpdateShoppingCartDatabase() {
    //实例化 ShoppingCartDB 类成为对象 cart
    ASPNET.StarterKit.Commerce.ShoppingCartDB cart = new ASPNET.StarterKit.Commerce.ShoppingCartDB();
    //获得当前用户的购物车 CartID
    String cartId = cart.GetShoppingCartId();
    //遍历页面中 DataGrid 控件 MyList 的 Items
    for (int i=0; i < MyList.Items.Count; i++) {
        //获得 MyList 当前行的 TextBox 控件 quantityTxt
        TextBox quantityTxt = (TextBox) MyList.Items[i].FindControl("Quantity");
        //获得 MyList 当前行的 CheckBox 控件 remove
        CheckBox remove = (CheckBox) MyList.Items[i].FindControl("Remove");

        //定义 int 型变量 quantity
        int quantity;
        try {
            //将 TextBox 控件的 quantityTxt 值赋给 quantity
            quantity = Int32.Parse(quantityTxt.Text);
            //判断 quantityTxt 和 remove 是否被更改
            if (quantity != (int)MyList.DataKeys[i] || remove.Checked == true) {
                //获得存储当前商品的 ProductID 的 Label 控件
                Label lblProductID = (Label) MyList.Items[i].FindControl("ProductID");
                //判断商品的数量是否为 0 和用户是否将删除商品购买信息
                if (quantity == 0 || remove.Checked == true) {
                    //调用 cart 对象的 RemoveItem 方法删除商品购买信息
                    cart.RemoveItem(cartId, Int32.Parse(lblProductID.Text));
                }
                else {
                    //调用 cart 对象的 UpdateItem 方法更新商品购买信息
                    cart.UpdateItem(cartId, Int32.Parse(lblProductID.Text), quantity);
                }
            }
        }
        //捕获异常
        catch {
            //显示异常
            MyError.Text = "There has been a problem with one or more of your inputs.";
        }
    }
}
```

```
    }

    private void Page_Init(object sender, EventArgs e) {
        InitializeComponent();
    }

    private void InitializeComponent() {
        // UpdateBtn 控件的单击事件委托
        this.UpdateBtn.Click += new System.Web.UI.ImageClickEventHandler(this.UpdateBtn_Click);
        // CheckoutBtn 控件的单击事件委托
        this.CheckoutBtn.Click += new System.Web.UI.ImageClickEventHandler(this.CheckoutBtn_Click);
        this.Load += new System.EventHandler(this.Page_Load);
    }
}
```

(3) 订单信息确认页面后台编码类 CheckOut.aspx.cs

购物信息确认页面 CheckOut.aspx 的后台编码类为 CheckOut.aspx.cs，该后台编码类在页面加载方法 Page_Load 实例化购物车模块的数据访问层类为前台的 DataGrid 页面获取并绑定数据集，在前台按钮 SubmitBtn 的单击事件中获取 DataGrid 控件的购物详细信息，实例化订单处理模块的数据访问层类成为对象 ordersDatabase，调用其 PlaceOrder 方法将订单信息写入数据库。

注意：订单处理模块的数据访问层 OrdersDB.cs 将在下一章中详细介绍。

```
using System;
using System.Collections;
using System.ComponentModel;
using System.Data;
using System.Data.SqlClient;
using System.Drawing;
using System.Web;
using System.Web.SessionState;
using System.Web.UI;
using System.Web.UI.WebControls;
using System.Web.UI.HtmlControls;

namespace ASPNET.StarterKit.Commerce {

    public class CheckOut : System.Web.UI.Page {
        //定义页面中显示标题的控件 Header
        protected System.Web.UI.WebControls.Label Header;
        //定义页面中显示操作信息的控件 Message
        protected System.Web.UI.WebControls.Label Message;
        //定义页面中显示购物详细信息的 DataGrid 控件 MyDataGrid
        protected System.Web.UI.WebControls.DataGrid MyDataGrid;
        //定义页面中显示购物总额的控件 TotalLbl
        protected System.Web.UI.WebControls.Label TotalLbl;
        //定义页面中 ImageButton 控件 SubmitBtn
        protected System.Web.UI.WebControls.ImageButton SubmitBtn;

        public CheckOut() {
            Page.Init += new System.EventHandler(Page_Init);
        }

        void Page_Init(object sender, EventArgs e) {
            //处理 Page_Init 事件
        }

        protected void Page_Load(object sender, EventArgs e) {
            //处理 Page_Load 事件
        }

        protected void Header_Click(object sender, EventArgs e) {
            //处理 Header 点击事件
        }

        protected void Message_Click(object sender, EventArgs e) {
            //处理 Message 点击事件
        }

        protected void MyDataGrid_SelectedIndexChanged(object sender, EventArgs e) {
            //处理 MyDataGrid 选中行事件
        }

        protected void TotalLbl_Click(object sender, EventArgs e) {
            //处理 TotalLbl 点击事件
        }

        protected void SubmitBtn_Click(object sender, EventArgs e) {
            //处理 SubmitBtn 点击事件
        }
    }
}
```

}

页面的装载事件 Page_Load 首先判断页面是否被提交，如果没有被提交，则说明用户并没有提交购物信息，实例化购物车模块数据访问层获得当前用户的购物车信息数据集并绑定页面的 Datagrid 控件，获得当前用户购物车内商品的总价并显示到控件 TotalLbl 中。

```
private void Page_Load(object sender, System.EventArgs e) {
    //判断页面是否被提交
    if (Page.IsPostBack == false) {
        //实例化 ShoppingCartDB 类成为对象 cart
        ASPNET.StarterKit.Commerce.ShoppingCartDB cart = new ASPNET.StarterKit.Commerce.ShoppingCartDB();
        //获得当前用户的购物车 CartID
        String cartId = cart.GetShoppingCartId();
        //获得当前购物车详细购物信息的数据集并设置为 MyDataGrid 的数据源
        MyDataGrid.DataSource = cart.GetItems(cartId);
        //绑定 MyDataGrid
        MyDataGrid.DataBind();
        //设定页面 TotalLbl 显示购物车内所有商品的总价
        TotalLbl.Text = String.Format("{0:c}", cart.GetTotal(cartId));
    }
}
```

SubmitBtn_Click 方法是 ImageButton 控件 SubmitBtn 的单击事件方法，当用户单击 SubmitBtn 控件的时候触发该方法。方法首先实例化购物车模块的数据访问层为 cart 对象，获得当前用户购物车的 cartId 和用户的 customerId，实例化订单处理模块的数据访问层为对象 ordersDatabase，调用其 PlaceOrder 方法添加当前的购物车信息成为当前用户的购物订单，并显示购物成功信息。

```
private void SubmitBtn_Click(object sender, System.Web.UI.ImageClickEventArgs e) {
    //实例化 ShoppingCartDB 类成为对象 cart
    ASPNET.StarterKit.Commerce.ShoppingCartDB cart = new ASPNET.StarterKit.Commerce.ShoppingCartDB();
    //获得当前用户的购物车 CartID
    String cartId = cart.GetShoppingCartId();
    //获得当前用户的 customerId
    String customerId = User.Identity.Name;
    //判断用户的 customerId 和 cartId 是否为空
    if ((cartId != null) && (customerId != null)) {
        //实例化 OrdersDB 类成为对象 ordersDatabase
        ASPNET.StarterKit.Commerce.OrdersDB ordersDatabase = new ASPNET.StarterKit.Commerce.OrdersDB();
        //调用 ordersDatabase 对象的 PlaceOrder 方法将当前购物车信息添加成为当前用户的订单
        int orderId = ordersDatabase.PlaceOrder(customerId, cartId);
        //设置显示订单添加成功信息
        Header.Text = "Check Out Complete!";
        Message.Text = "<b>Your Order Number Is: </b>" + orderId;
        //隐藏提交按钮
        SubmitBtn.Visible = false;
    }
}
private void Page_Init(object sender, EventArgs e) {
    InitializeComponent();
}
```

```
private void InitializeComponent() {
    //SubmitBtn 控件单击事件委托
    this.SubmitBtn.Click += new System.Web.UI.ImageClickEventHandler(this.SubmitBtn_Click);
    this.Load += new System.EventHandler(this.Page_Load);
}
```

6.4.5 用户表示层

(1) 加入购物车页面 AddToCart.aspx

加入购物车页面 AddToCart.aspx 是向购物车页面添加一条购物信息的页面，该页面所有业务逻辑都集中在后台编码类中，前台并没有实际业务逻辑代码，因此不详细叙述。

(2) 购物车页面 ShoppingCart.aspx

购物车页面显示当前用户的购物车内的详细购物信息，该页面使用了 Datagrid 显示详细购物信息列表。用户可以更改购物信息中的购物数量或删除某一条详细购物信息，可单击更新购物车信息按钮更新购物车信息，单击确认订单信息按钮跳转到确认订单信息页面。以下是显示详细购物信息列表的 DataGrid 代码清单。

```
<!-- 定义 DataGrid 的基本属性和样式 -->
<asp:DataGrid id=MyList runat="server" BorderColor="black" GridLines="Vertical" cellpadding="4" cellspacing="0" Font-Name="Verdana" Font-Size="8pt" ShowFooter="true" HeaderStyle-CssClass="CartListHead" FooterStyle-CssClass="CartListFooter" ItemStyle-CssClass="CartListItem" AlternatingItemStyle-CssClass="CartListItemAlt" DataKeyField="Quantity" AutoGenerateColumns="false">
    <!-- 定义列集合 -->
    <Columns>
        <!-- 定义模板列 -->
        <asp:TemplateColumn HeaderText="Product&nbsp;ID">
            <!-- 定义模板列的常规行，加载 Label 控件 ProductID 显示数据字段 ProductID -->
            <ItemTemplate>
                <asp:Label id="ProductID" runat="server" Text='<%# DataBinder.Eval(Container.DataItem, "ProductID") %>' />
            </ItemTemplate>
        </asp:TemplateColumn>
        <!-- 定义数据列显示数据字段 ModelName -->
        <asp:BoundColumn HeaderText="Product Name" DataField="ModelName" />
        <!-- 定义数据列显示数据字段 ModelNumber -->
        <asp:BoundColumn HeaderText="Model" DataField="ModelNumber" />
        <!-- 定义模板列 -->
        <asp:TemplateColumn HeaderText="Quantity">
            <!-- 定义模板列的常规行，加载 TextBox 控件 Quantity 显示数据字段 Quantity -->
            <ItemTemplate>
                <asp:TextBox id="Quantity" runat="server" Columns="4" MaxLength="3" Text='<%# DataBinder.Eval(Container.DataItem, "Quantity") %>' width="40px" />
            </ItemTemplate>
        </asp:TemplateColumn>
        <!-- 定义数据列显示数据字段 UnitCost -->
        <asp:BoundColumn HeaderText="Price" DataField="UnitCost" DataFormatString="{0:c}" />
        <!-- 定义数据列显示数据字段 ExtendedAmount -->
        <asp:BoundColumn HeaderText="Subtotal" DataField="ExtendedAmount" DataFormatString="{0:c}" />
        <!-- 定义模板列 -->
        <asp:TemplateColumn HeaderText="Remove">
            <!-- 定义模板列的常规行，加载 CheckBox 控件 Remove -->
            <ItemTemplate>
                <center>
                    <asp:CheckBox id="Remove" runat="server" />
                </center>
            </ItemTemplate>
        </asp:TemplateColumn>
    </Columns>
</asp:DataGrid>
```

```

        </center>
    </ItemTemplate>
</asp:TemplateColumn>
</Columns>
</asp:DataGrid>

```

(3) 订单信息确认页面 CheckOut.aspx

订单信息确认页面是让用户完成订单之前确定当前购物车信息的页面，和购物车页面类似，该页面使用了 DataGrid 显示详细购物信息列表，但是在购物车页面用户是可以更改购物信息的，在订单信息确认页面购物信息是不可以被更改的。如果用户单击确认购物信息按钮，则将生成订单，完成订购。以下是显示详细购物信息列表的 DataGrid 代码清单。

```

<!-- 定义 DataGrid 的基本属性和样式 -->
<asp:DataGrid id="MyDataGrid" width="90%" BorderColor="black" GridLines="Vertical" cellpadding="4"
cellspacing="0" Font-Name="Verdana" Font-Size="8pt" ShowFooter="true" HeaderStyle-CssClass="CartListHead"
FooterStyle-CssClass="cartlistfooter" ItemStyle-CssClass="CartListItem"
AlternatingItemStyle-CssClass="CartListItemAlt" AutoGenerateColumns="false" runat="server">
<!-- 定义列集合 -->
<Columns>
    <!-- 定义数据列显示数据字段 ModelName -->
    <asp:BoundColumn HeaderText="Product Name" DataField="ModelName" />
    <!-- 定义数据列显示数据字段 ModelNumber -->
    <asp:BoundColumn HeaderText="Model Number" DataField="ModelNumber" />
    <!-- 定义数据列显示数据字段 Quantity -->
    <asp:BoundColumn HeaderText="Quantity" DataField="Quantity" />
    <!-- 定义数据列显示数据字段 UnitCost -->
    <asp:BoundColumn HeaderText="Price" DataField="UnitCost" DataFormatString="{0:c}" />
    <!-- 定义数据列显示数据字段 ExtendedAmount -->
    <asp:BoundColumn HeaderText="Subtotal" DataField="ExtendedAmount" DataFormatString="{0:c}" />
</Columns>
</asp:DataGrid>

```

6.5 商品评论

6.5.1 实现效果

在商品的详细信息页面用户可以查看到其他用户关于商品的评论，商品评论信息包括评论的用户、用户对该商品的打分，以及具体的评论文字。用户可以单击商品评论链接进入商品评论页面，对该商品发表自己的评论。商品详细信息页面商品评论的部分参见图 6-9。

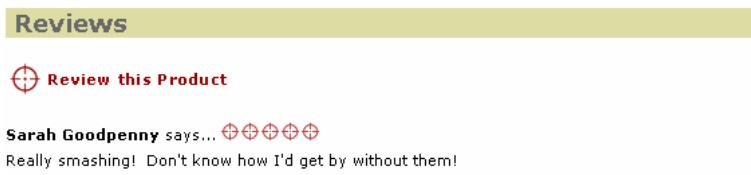


图 6-9 商品详细信息页面商品评论的部分

发表商品评论页面参见图 6-10。

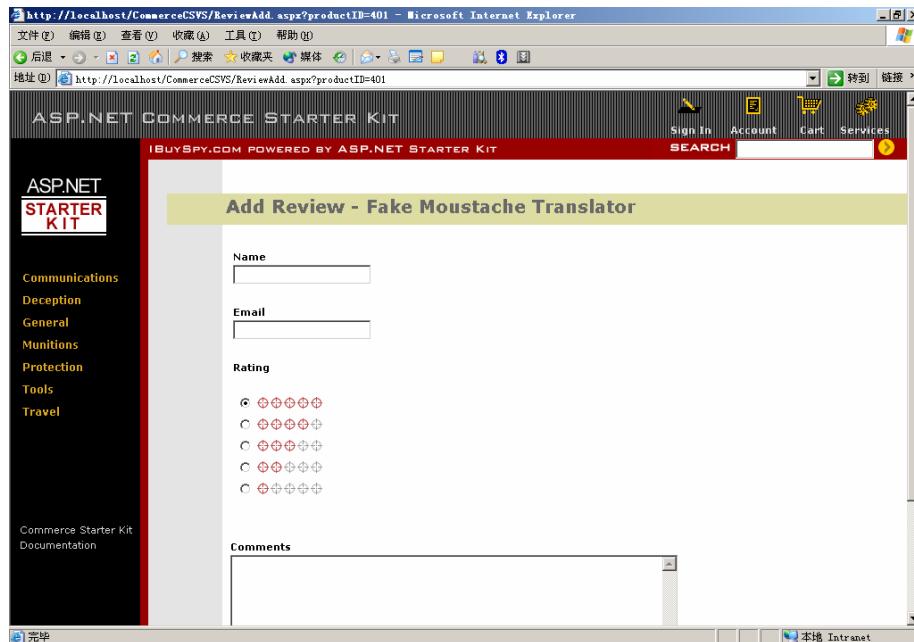


图 6-10 发表商品评论页面

6.5.2 存储过程

商品评论模块使用了两个存储过程：CMRC_ReviewsAdd 和 CMRC_ReviewsList。

(1) 存储过程 CMRC_ReviewsAdd

存储过程 CMRC_ReviewsAdd 为添加一条商品评论的存储过程。添加的评论信息包括商品的 ProductID、用户的 CustomerName 和 CustomerEmail、用户对该商品的打分 Rating 和评论 Comments，并返回被添加的评论的 ReviewID。

```
CREATE Procedure CMRC_ReviewsAdd
(
    @ProductID      int,
    @CustomerName   nvarchar(50),
    @CustomerEmail  nvarchar(50),
    @Rating         int,
    @Comments        nvarchar(3850),
    @ReviewID       int OUTPUT
)
AS
/* 向 CMRC_Reviews 表中添加一条记录 */
INSERT INTO CMRC_Reviews
(
    ProductID,
    CustomerName,
    CustomerEmail,
    Rating,
    Comments
)
```

```
VALUES
(
    @ProductID,
    @CustomerName,
    @CustomerEmail,
    @Rating,
    @Comments
)

SELECT
    @ReviewID = @@Identity
```

(2) 存储过程 CMRC_ReviewsList

CMRC_ReviewsList 存储过程是返回某个商品所有评论信息记录集的存储过程。

```
CREATE Procedure ReviewsList
(
    @ProductID int
)
AS

SELECT
    ReviewID,
    CustomerName,
    Rating,
    Comments
FROM
    Reviews
WHERE
    ProductID = @ProductID
```

6.5.3 数据访问层

商品评论模块的数据访问层为 Components 文件夹下的 ReviewsDB.cs 文件所提供的类 ASPNET.StarterKit.Commerce. ReviewsDB。该数据访问层提供了商品评论模块业务逻辑层所使用的添加评论和返回某个商品的评论列表两个方法。以下是 ReviewsDB.cs 的代码清单。

```
using System;
using System.Configuration;
using System.Data;
using System.Data.SqlClient;

namespace ASPNET.StarterKit.Commerce {
```

```
    public class ReviewsDB {
```

GetReviews 方法是购物车所有详细购物信息的数据集的方法。GetReviews 方法首先创建了数据库连接和数据库命令，并指定数据库命令执行存储过程 CMRC_ReviewsList，创建了一个 SqlParameter 类型的变量 parameterProductID，然后将参数方法的参数 productID 的值赋给 parameterCartID，再将 parameterCartID 添加到数据库命令的参数表中。最后执行数据库命令获得 SqlDataReader 类型的变量 result，并将 result 返回。

```
        public SqlDataReader GetReviews(int productID) {
```

```
//实例化一个数据库连接对象 myConnection
SqlConnection myConnection = new SqlConnection(ConfigurationSettings.AppSettings["ConnectionString"]);
//实例化一个使用 myConnection 执行存储过程 CMRC_ReviewsList 的数据库命令 myCommand
SqlCommand myCommand = new SqlCommand("CMRC_ReviewsList", myConnection);
//设置 myCommand 的类型为执行存储过程
myCommand.CommandType = CommandType.StoredProcedure;

//定义一个 SqlParameter 类型的存储过程参数 parameterProductID
SqlParameter parameterProductID = new SqlParameter("@ProductID", SqlDbType.Int, 4);
//将方法的参数 productID 的值赋给 parameterProductID
parameterProductID.Value = productID;
//将 parameterProductID 添加到数据库命令 myCommand 的参数表中去
myCommand.Parameters.Add(parameterProductID);

//打开数据库连接 myConnection
myConnection.Open();
//执行 myCommand 并获得 SqlDataReader 类型的记录集 result
SqlDataReader result = myCommand.ExecuteReader(CommandBehavior.CloseConnection);
//返回 result
return result;
}
```

AddReview 方法是添加一条商品评论的方法。AddItem 方法首先创建了数据库连接和数据库命令，并指定数据库命令执行存储过程 CMRC_ReviewsAdd，创建了 6 个 SqlParameter 类型的变量 parameterProductID、parameterCustomerName、parameterEmail、parameterRating、parameterComments、parameterReviewID，并为其赋值。后将 6 个 SqlParameter 类型的变量添加到数据库命令的参数表中。最后打开数据连接，执行数据库命令，最终关闭数据库连接。其中参数 parameterReviewID 为返回值型参数，返回添加的商品评论的 ReviewID。

```
public void AddReview(int productID, string customerName, string customerEmail, int rating, string comments) {
    //实例化一个数据库连接对象 myConnection
    SqlConnection myConnection = new SqlConnection(ConfigurationSettings.AppSettings["ConnectionString"]);
    //实例化一个使用 myConnection 执行存储过程 CMRC_ReviewsAdd 的数据库命令 myCommand
    SqlCommand myCommand = new SqlCommand("CMRC_ReviewsAdd", myConnection);
    //设置 myCommand 的类型为执行存储过程
    myCommand.CommandType = CommandType.StoredProcedure;

    //定义一个 SqlParameter 类型的存储过程参数 parameterProductID
    SqlParameter parameterProductID = new SqlParameter("@ProductID", SqlDbType.Int, 4);
    //将方法的参数 productID 的值赋给 parameterProductID
    parameterProductID.Value = productID;
    //将 parameterProductID 添加到数据库命令 myCommand 的参数表中去
    myCommand.Parameters.Add(parameterProductID);
    //定义一个 SqlParameter 类型的存储过程参数 parameterCustomerName
    SqlParameter parameterCustomerName = new SqlParameter("@CustomerName", SqlDbType.NVarChar, 50);
    //将方法的参数 customerName 的值赋给 parameterCustomerName
    parameterCustomerName.Value = customerName;
    //将 parameterCustomerName 添加到数据库命令 myCommand 的参数表中去
}
```

```
myCommand.Parameters.Add(parameterCustomerName);
//定义一个SqlParameter 类型的存储过程参数 parameterEmail
SqlParameter parameterEmail = new SqlParameter("@CustomerEmail", SqlDbType.NVarChar, 50);
//将方法的参数 customerEmail 的值赋给 parameterEmail
parameterEmail.Value = customerEmail;
//将 parameterEmail 添加到数据库命令 myCommand 的参数表中去
myCommand.Parameters.Add(parameterEmail);
//定义一个SqlParameter 类型的存储过程参数 parameterRating
SqlParameter parameterRating = new SqlParameter("@Rating", SqlDbType.Int, 4);
//将方法的参数 rating 的值赋给 parameterRating
parameterRating.Value = rating;
//将 parameterRating 添加到数据库命令 myCommand 的参数表中去
myCommand.Parameters.Add(parameterRating);
//定义一个SqlParameter 类型的存储过程参数 parameterComments
SqlParameter parameterComments = new SqlParameter("@Comments", SqlDbType.NVarChar, 3850);
//将方法的参数 comments 的值赋给 parameterComments
parameterComments.Value = comments;
//将 parameterComments 添加到数据库命令 myCommand 的参数表中去
myCommand.Parameters.Add(parameterComments);
//定义一个SqlParameter 类型的存储过程参数 parameterReviewID
SqlParameter parameterReviewID = new SqlParameter("@ReviewID", SqlDbType.Int, 4);
//设置参数 parameterReviewID 为返回值型参数
parameterReviewID.Direction = ParameterDirection.Output;
//将 parameterReviewID 添加到数据库命令 myCommand 的参数表中去
myCommand.Parameters.Add(parameterReviewID);

//打开数据库连接 myConnection
myConnection.Open();
//调用数据库命令 myCommand 的 ExecuteNonQuery 方法执行数据库命令
myCommand.ExecuteNonQuery();
//关闭数据库连接 myConnection
myConnection.Close();
}
}
```

6.5.4 业务逻辑层

(1) 商品评论列表用户控件后台编码类 ReviewList.ascx.cs

商品评论列表用户控件_ReviewList.ascx 的后台编码类为_ReviewList.ascx.cs，该用户控件在商品详细页面被调用，在控件的页面装载方法 Page_Load 中首先实例化商品评论模块的数据访问层对象 productReviews，然后调用 productReviews 对象的 GetReviews 方法获得当前商品的评论信息数据集，绑定到前台 DataList 控件。

```
private void Page_Load(object sender, System.EventArgs e) {
    //实例化 ReviewsDB 类成为对象 productReviews
    ASPNET.StarterKit.Commerce.ReviewsDB productReviews = new ASPNET.StarterKit.Commerce.ReviewsDB();
    //调用 productReviews 对象的 GetReviews 获得商品评论信息的数据集并设置成 myList 的数据集
    myList.DataSource = productReviews.GetReviews(ProductID);
```

```
//绑定 MyList
MyList.DataBind();
//设置添加评论信息链接 AddReview 的 NavigateUrl
AddReview.NavigateUrl = "ReviewAdd.aspx?productID=" + ProductID.ToString();
}
```

(2) 商品评论添加页面后台编码类 ReviewAdd.aspx.cs

商品评论添加页面 ReviewAdd.aspx 的后台编码类为 ReviewAdd.aspx.cs，该后台编码类在页面装载方法 Page_Load 中判断页面是否被提交，如果未被提交，则实例化商品浏览模块的数据访问层获得商品的 ModelName，并设置名为 productID 的 ViewState。在提交 ReviewAddBtn 的单击事件方法中，实例化商品评论模块的数据访问层成为对象 review，捕获页面中用户输入的商品评论信息，调用 review 对象的 AddReview 添加一条商品评论。

```
using System;
using System.Collections;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Web;
using System.Web.UI;
using System.Web.UI.WebControls;
using System.Web.UI.HtmlControls;

namespace ASPNET.StarterKit.Commerce {

    public class ReviewAdd : System.Web.UI.Page {
        //定义前台页面使用的控件
        protected System.Web.UI.WebControls.Label ModelName;
        protected System.Web.UI.WebControls.TextBox Name;
        protected System.Web.UI.WebControls.RequiredFieldValidator RequiredFieldValidator1;
        protected System.Web.UI.WebControls.TextBox Email;
        protected System.Web.UI.WebControls.RequiredFieldValidator RequiredFieldValidator2;
        protected System.Web.UI.WebControls.RadioButtonList Rating;
        protected System.Web.UI.WebControls.TextBox Comment;
        protected System.Web.UI.WebControls.RequiredFieldValidator RequiredFieldValidator3;
        protected System.Web.UI.WebControls.ImageButton ReviewAddBtn;

        public ReviewAdd() {
            Page.Init += new System.EventHandler(Page_Init);
        }
        //页面装载方法
        private void Page_Load(object sender, System.EventArgs e) {
            //判断页面是否被提交
            if (Page.IsPostBack != true) {
                //获取当前商品 productID
                int productID = Int32.Parse(Request["productID"]);
                //实例化商品浏览模块数据访问层 ProductsDB 类成为对象 products
                ASPNET.StarterKit.Commerce.ProductsDB products = new ASPNET.StarterKit.Commerce.
ProductsDB();
                //获取商品的 ModelName
                ModelName.Text = products.GetProductDetails(productID).ModelName;
            }
        }
        protected void ReviewAddBtn_Click(object sender, System.EventArgs e) {
            //调用 review 对象的 AddReview 方法添加商品评论
            review.AddReview();
        }
    }
}
```

```

        //设置 ViewState 供页面被提交的时候使用
        ViewState["productID"] = productID;
    }
}

//按钮 ReviewAddBtn 的单击事件方法
private void ReviewAddBtn_Click(object sender, System.Web.UI.ImageClickEventArgs e) {
    //判断页面数据是否通过验证控件验证
    if (Page.IsValid == true) {

        //从页面的 ViewState 对象中取得 productID
        int productID = (int) ViewState["productID"];
        //获取 Rating 控件的值
        int rating = Int32.Parse(Rating.SelectedItem.Value);
        //实例化商品评论模块数据访问层 ReviewsDB 类成为对象 review
        ASPNET.StarterKit.Commerce.ReviewsDB review = new ASPNET.StarterKit.Commerce.ReviewsDB();
        //调用 review 对象的 AddReview 方法添加一条商品信息
        review.AddReview(productID, Server.HtmlEncode(Name.Text), Server.HtmlEncode(Email.Text),
rating, Server.HtmlEncode(Comment.Text));
        //将页面重定向到商品详细信息页面
        Response.Redirect("ProductDetails.aspx?ProductID=" + productID);
    }
}

private void Page_Init(object sender, EventArgs e) {
    InitializeComponent();
}

private void InitializeComponent() {
    //按钮 ReviewAddBtn 的单击事件方法委托
    this.ReviewAddBtn.Click += new System.Web.UI.ImageClickEventHandler(this.ReviewAddBtn_Click);
    this.Load += new System.EventHandler(this.Page_Load);
}
}
}

```

6.5.5 用户表示层

(1) 商品评论列表控件_ReviewList.ascx

商品评论列表控件_ReviewList.ascx 被商品详细信息页面所调用，该用户控件以 DataList 形式显示当前商品评论信息的详细列表。

```

<!-- 定义 DataList 的主要属性及样式 -->
<asp:DataList ID="MyList" runat="server" width="500" cellpadding="0" cellspacing="0">
    <!-- 定义模板 -->
    <ItemTemplate>
        <!-- 定义 Label 控件显示发表评论的用户名 -->
        <asp:Label class="NormalBold" Text='<%# DataBinder.Eval(Container.DataItem, "CustomerName") %>' runat="server" />
        <!-- 定义 Label 控件显示用户对该商品的打分 -->
        <span class="Normal">says... </span><img src='images/ReviewRating<%# DataBinder.Eval

```

```
(Container.DataItem, "Rating") %>.gif'>
    <br>
    <!-- 定义 Label 控件显示用户对该商品的具体评论 -->
    <asp:Label class="Normal" Text='<%# DataBinder.Eval(Container.DataItem, "Comments") %>' runat="server" />
</ItemTemplate>
<!-- 定义分隔模板 -->
<SeparatorTemplate>
    <br>
</SeparatorTemplate>
</asp:DataList>
```

(2) 商品评论添加页面 ReviewAdd.aspx

商品评论添加页面 ReviewAdd.aspx 是为用户发表商品评论提供的一个页面，该页面显示被评论的商品名称，页面提供了商品评论信息的文本输入框。用户在填写完商品评论信息之后，单击提交按钮 ReviewAddBtn 之后完成对商品的评论。由于该页面并没有提供更多的用户逻辑，在此就不详细讲解代码了。

6.6 开发启示

本章所讲解的内容是 ASP.NET Commerce Starter Kit 中最核心的一部分内容，商品列表的构建、商品评论以及购物车模块构成了一个简单的电子商务应用的基础。设计者不仅仅展示了电子商务应用逻辑的实现，同时结合项目实际情况对 ASP.NET 中数据访问与数据展示技术进行了比较深入地讲解。

应用 ASP.NET 中数据访问与数据展示技术，对于同一需求可能有多种实现的方式和方案，所以要求开发者能够根据实际情况，选择一种合适的解决方案。在选择具体解决方案的时候，要从功能、性能、复杂度等多面考虑，例如最受欢迎商品的列表，使用 DataGrid、DataList 和 Repeater 控件都可以实现，但是考虑到性能和复杂度等因素，选用功能相对简单但是足以满足需求的 Repeater 控件实现，不仅提高了开发人员的效率，也提高了应用程序的运行效率。对于购物车详细信息列表则采用 DataGrid 控件，因为 DataList 和 Repeater 控件已经无法满足实际的需求，所以 DataGrid 控件对数据的显示和编辑等方面的优势是另外两个控件所无法比拟的。

在逻辑应用的设计上，设计者也充分考虑到电子商务应用程序和其他 Web 应用程序的区别。比如在将某个商品加入购物车这一步骤，设计者使用了 AddToCart.aspx 页面作为桥梁，在 AddToCart.aspx 页面处理添加购物详细信息的逻辑，然后跳转到购物车页面，而不是直接地将数据提交到购物车页面，这种做法防止用户在刷新购物车页面的时候重复地添加购物信息。另外，为了保证用户在没有登录的情况下也可以使用购物车存储购物信息，设计者引入临时购物车机制，为匿名用户提供一个临时的购物车存储购物信息，当匿名用户注册或登录成为正式用户之后仍然可以获得保存在临时购物车中的信息。

开发者也可以在目前 ASP.NET Commerce Starter Kit 提供的 Web 商店模块上进行扩展和二次开发，如下几个方面都是扩展和二次开发的着手点。

- 商品分类的管理和多级化。ASP.NET Commerce Starter Kit 提供的仅是一级商品分类，但是在实际的电子商务应用中，往往需要多级分类甚至多维分类，因此商品分类的管理和多级化应该是必不可少的一个模块。
- 商品列表的多种呈现方式。ASP.NET Commerce Starter Kit 的商品列表仅仅是基于商品分类或商品查询的，开发者可以根据实际需要设计多种呈现方式。
- 商品信息的添加和管理。ASP.NET Commerce Starter Kit 中并没有提供商品信息的添加和管理

模块，这显然和实际的需求有一定的差距。

- 商品评论信息的查看和管理。商品评论信息是架设在电子商务应用的管理员和使用者之间的一个沟通的桥梁，因此管理员对于商品评论的查看和管理是必不可少的。

第 7 章

购物订单

购物订单数据作为电子商务用户最终订购商品的载体，是电子商务流程中最重要的数据。而订单的查看和处理模块也是电子商务中最重要的模块。本章将讲解 ASP.NET Commerce Starter Kit 中订单查看模块的详细内容。在 ASP.NET Commerce Starter Kit 中提供了历史订单查看模块，其中包括查看历史订单列表和每一笔订单的详细内容。表 7-1 为本章知识点索引。

表 7-1 本章知识点索引

| 知 识 点 | 位 置 |
|--------------------------------|---------|
| 数据库适配器 SqlDataAdapter 对象 | 7.2.2 节 |
| DataGrid 的 DataFormatString 属性 | 7.2.2 节 |
| 存储过程返回多个记录集 | 7.2.3 节 |
| 将订单抽象成为对象模型 | 7.2.4 节 |
| 返回 DataSet 对象 | 7.2.5 节 |

7.1 系统设计

7.1.1 需求分析

在 ASP.NET Commerce Starter Kit 示例中，用户在确认订单之后系统会将用户购物车内的详细购物信息添加成为一笔订单。用户单击页面导航栏上的 Account 链接，会跳转到用户历史订单列表页面，单击历史订单列表中显示订单详细信息的链接可以链接到订单详细信息页面以查看每一笔订单的详细信息。

总结系统中关于用户订单的需求有如下几点。

- 添加订单。用户在确认订单之后系统会将用户购物车内的详细购物信息添加成为一笔订单。
- 查看历史订单列表。用户单击页面导航栏上的 Account 链接，会跳转到用户历史订单列表页面，该页面以列表的形式展现当前用户的历史订单记录，每一条记录包含订单号、订单日期、订单总额，以及订单送达日期和查看订单详细内容的链接。
- 查看订单详细信息。用户单击历史订单列表中每一笔订单显示详细信息的链接可以链接到订单详细信息页面查看每一笔订单的详细信息。订单的详细信息即订单中每一件商品的购物信息，包括商品名称、商品

型号、商品价格、购买数量和总价格。

7.1.2 功能设计

根据购物订单模块的需求分析，将购物订单模块分为3个功能子模块。

(1) 订单添加

该功能模块在第6章Web商店中有所介绍，将购物车中的数据添加到Orders和OrderDetail表中。

(2) 历史订单列表页面

该功能模块用以显示历史订单列表。

(3) 详细订单列表页面

该功能模块用以显示订单详细信息。

注意：在第6章已经讲解了订单添加模块的业务逻辑层和用户表示层，因此本章中将不重复该部分内容。

7.1.3 数据库设计

电子商务中的订单作为一个独特的数据模型，需要使用两张数据表存储一笔订单的信息。因为每笔订单的信息可以分成两部分，一部分是订单的总体信息，包括订单号、订单日期、订单总额等，另外一部分是订单的详细购物信息，其中包括订单号、具体购买商品的名称、价格，以及购买该商品的数量和总价。订单的主体信息和订单的详细购物信息是一对一或一对多的数据关系，因此要设计两个数据表：订单主体信息表和订单详细信息表。两个表通过订单的订单号实现关联。通过主表的订单号可以确定字表中属于该订单的记录，通过字表的订单号可以确定主表中哪条订购主体信息与之对应。

ASP.NET Commerce Starter Kit示例中订单主体信息表为Orders，订单详细信息表为OrderDetail，两个表通过Orders.OrderID和OrderDetail.OrderID实现关联。两个表的数据结构参见图7-1和7-2。

| 列名 | 数据类型 | 长度 | 允许空 |
|------------|----------|----|-----|
| OrderID | int | 4 | |
| CustomerID | int | 4 | |
| OrderDate | datetime | 8 | |
| ShipDate | datetime | 8 | |

图7-1 数据表Orders的数据结构

| 列名 | 数据类型 | 长度 | 允许空 |
|-----------|-------|----|-----|
| OrderID | int | 4 | |
| ProductID | int | 4 | |
| Quantity | int | 4 | |
| UnitCost | money | 8 | |

图7-2 数据表OrderDetail的数据结构

在购物订单模块主要用到3个存储过程，3个存储过程的主要信息参见表7-2。

表7-2

存储过程及说明

| 存储过程名 | 说 明 |
|-------------------|------------------|
| CMRC_OrdersAdd | 向数据库中添加一笔订单信息 |
| CMRC_OrdersList | 获得某个用户的历史订单列表 |
| CMRC_OrdersDetail | 获取某个用户的某一订单的详细信息 |

7.2 购物订单

7.2.1 实现效果

单击导航栏的Account链接，跳转到用户历史订单列表OrderList页面，该页面显示的是当前用户的历史订单列表。该页面的实现效果参见图7-3。

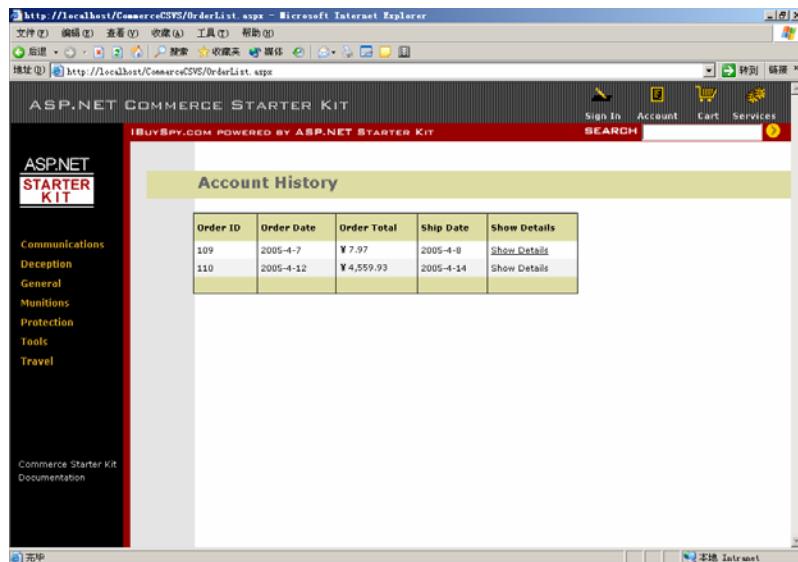


图 7-3 用户历史订单列表实现效果

用户单击历史订单列表页面中每条订单记录的 Show Details 链接就会链接到查看详细订单信息页面 OrderDetails.aspx。OrderDetails.aspx 页面显示了该笔订单的详细信息。该页面的实现效果参见图 7-4。

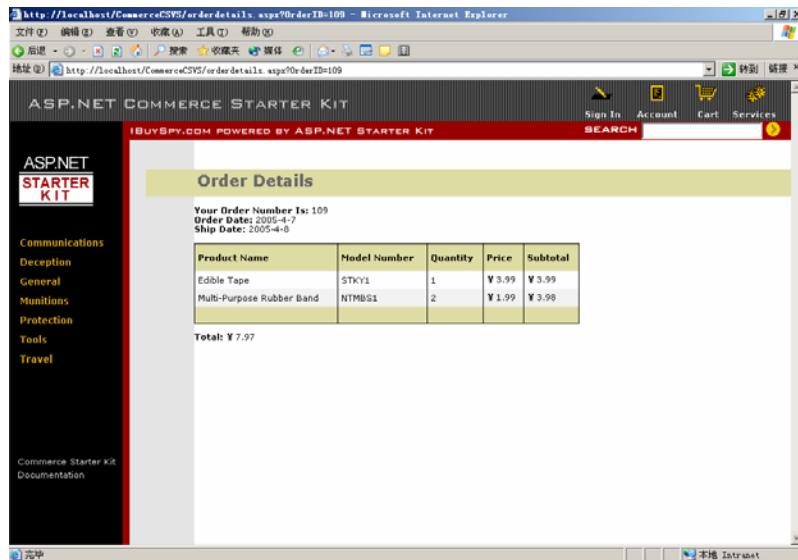


图 7-4 详细订单信息页面实现效果

7.2.2 关键技术点

(1) 数据库适配器 SqlDataAdapter 对象

在第 6 章 Web 商店的技术点详解中讲解了 ASP.NET 中的数据访问技术，其中提到了 DataSet 对象。但是并没有使用 DataSet 对象和 SQL Server 数据库进行交互，因为在使用 DataSet 获取 SQL Server 数据库中的数据集时应该使用 SqlDataAdapter 对象作为连接的桥梁。

SqlDataAdapter 是 DataSet 对象和 SQL Server 之间的桥接器，用于检索和保存数据。SqlDataAdapter

通过对数据源使用适当的 Transact-SQL 语句映射 Fill 方法和 Update 方法来提供这一桥接。其中 Fill 方法的主要功能是填充数据源中的数据到 DataSet 对象，Update 方法根据 DataSet 对象中的数据更新数据源的数据。

当 SqlDataAdapter 填充 DataSet 时，将在 DataSet 对象中创建和数据源中的数据对应的 DataTables、DataColumn、DataRow 甚至是 DataTables 的主键信息。SqlDataAdapter 与 SqlConnection 和 SqlCommand 一起使用，可以提高应用程序中连接 Microsoft SQL Server 数据库的性能。SqlDataAdapter 还包括可以设置 Transact-SQL 语句或存储过程命令类型的 SelectCommand、InsertCommand、DeleteCommand、UpdateCommand 和 TableMappings 属性，使数据的加载和更新更加方便。

以下程序范例是使用 SqlCommand、SqlDataAdapter 和 SqlConnection，从数据库选择记录并用选定的记录集填充 DataSet，然后返回已填充的 DataSet。为完成此任务，向该方法传递一个已初始化的 DataSet、一个连接字符串和一个查询字符串，后者是一个 Transact-SQL SELECT 语句。

```
public DataSet getDataSet(DataSet ds, string connstring, string sqlstring)
{
    //实例化一个数据库连接对象 conn
    SqlConnection conn = new SqlConnection(connstring);
    //实例化一个数据库适配器对象 da
    SqlDataAdapter da = new SqlDataAdapter();
    //设置 da 的 SelectCommand 属性
    da.SelectCommand = new SqlCommand(sqlstring, conn);
    //调用 da 的 Fill 方法填充 ds
    da.Fill(ds);
    //返回 ds
    return ds;
}
```

在使用 SqlDataReader 对象的过程中，直接采用将 SqlCommand 获取的记录集填充到 SqlDataReader 中的做法。因为 SqlDataReader 对象并不像 DataSet 对象一样，拥有关系数据库模型的特征，DataSet 本身可以看成一个建立在内存中的小型关系数据库，不能从 SqlCommand 返回的记录集中获得复杂的数据对象，因此必须使用 SqlDataAdapter 作为桥梁获得数据库中返回的数据集。

(2) 使用 DataGrid 中的 DataFormatString 属性来显示自定义格式的数据字段

在数据显示控件中，开发者经常会遇到需要根据自己定义的格式显示数据。例如显示一个 float 型变量到小数点后两位，或以货币的形式显示某个变量。在 DataGrid 中，可以使用行集合中数据字段的 DataFormatString 属性来设置显示自定义格式的数据字段。

数据格式字符串由以冒号分隔的两部分组成，形式为 { A: Bxx }。例如，格式化字符串 {0:F2} 将显示带两位小数的定点数。整个字符串必须放在大括号内，表示是格式字符串，而不是实际字符串。大括号外的任何文本均显示为实际文本。冒号前的值指定在从零开始的参数列表中的参数索引，也就是在索引为 A 的单元格内显示格式化之后的字符串。因为每个单元格中只有一个值，所以此值只能设置为 0。冒号后的字符指定值的显示格式。数据格式字符串的一些常用格式参见表 7-3。

表 7-3

数据格式字符串的一些常用格式

| 格 式 字 符 | 说 明 |
|---------|------------------|
| C | 以货币格式显示数值 |
| D | 以十进制格式显示数值 |
| E | 以科学计数法（指数）格式显示数值 |

续表

| 格 式 字 符 | 说 明 |
|---------|----------------|
| | 购物订单 第 7 章 115 |

| | |
|---|-------------|
| F | 以固定格式显示数值 |
| G | 以常规格式显示数值 |
| N | 以数字格式显示数值 |
| X | 以十六进制格式显示数值 |

通过表 7-3 可以知道, 将 DataFormatString 属性设置为 {0:C} 代表以货币的格式显示数据字段, 而将 DataFormatString 属性设置为 {0:D} 则代表以十进制格式显示数据字段。

7.2.3 存储过程

购物订单模块使用了 3 个存储过程: CMRC_OrdersAdd、CMRC_OrdersList、CMRC_OrdersDetail。由于在上一章 Web 商店中讲解过 CMRC_OrdersAdd, 在此不重复讲解, 下面将分别对 CMRC_OrdersList 和 CMRC_OrdersDetail 进行详细讲解。

(1) 存储过程 CMRC_OrdersList

存储过程 CMRC_OrdersList 是返回某个用户的历史订单列表的存储过程。历史订单列表中包含订单的 OrderId、订单的总价 OrderTotal、订单的提交日期 OrderDate 以及订单的送达日期 ShipDate。该存储过程的参数为用户的 CustomerID。以下是该存储过程的代码清单。

```
CREATE Procedure CMRC_OrdersList
(
    @CustomerID int
)
As

SELECT
    CMRC_Orders.OrderID,
    Cast(sum(CMRC_OrderDetails.quantity*CMRC_OrderDetails.unitcost) as money) as OrderTotal,
    CMRC_Orders.OrderDate,
    CMRC_Orders.ShipDate

FROM
    CMRC_Orders
INNER JOIN CMRC_OrderDetails ON CMRC_Orders.OrderID = CMRC_OrderDetails.OrderID

GROUP BY
    CustomerID,
    CMRC_Orders.OrderID,
    CMRC_Orders.OrderDate,
    CMRC_Orders.ShipDate

HAVING
    CMRC_Orders.CustomerID = @CustomerID
```

(2) 存储过程 CMRC_OrdersDetail

存储过程 CMRC_OrdersDetail 是返回某笔订单的详细订购信息的存储过程, 该存储过程返回的不仅是订单的主题信息, 还有订单中的详细订购信息。

```
CREATE Procedure CMRC_OrdersDetail
(
    @OrderID      int,
    @CustomerID  int,
```

```
@OrderDate datetime OUTPUT,
@ShipDate datetime OUTPUT,
@OrderTotal money OUTPUT
)
AS

/*返回根据 OrderID 和 CustomerID 返回订单主体信息的 OrderDate 和 ShipDate */
SELECT
    @OrderDate = OrderDate,
    @ShipDate = ShipDate

FROM
    CMRC_Orders

WHERE
    OrderID = @OrderID
    AND
    CustomerID = @CustomerID
    /*如果存在和参数中 OrderID 和 CustomerID 对应的结果集*/
IF @@Rowcount = 1
BEGIN

/*首先返回所有购物信息的累计金额 OrderTotal */
SELECT
    @OrderTotal = Cast(SUM(CMRC_OrderDetails.Quantity * CMRC_OrderDetails.UnitCost) as money)

FROM
    CMRC_OrderDetails

WHERE
    OrderID= @OrderID

/* 返回该订单所有详细购物信息的记录集 */
SELECT
    CMRC_Products.ProductID,
    CMRC_Products.ModelName,
    CMRC_Products.ModelNumber,
    CMRC_OrderDetails.UnitCost,
    CMRC_OrderDetails.Quantity,
    (CMRC_OrderDetails.Quantity * CMRC_OrderDetails.UnitCost) as ExtendedAmount

FROM
    CMRC_OrderDetails
INNER JOIN CMRC_Products ON CMRC_OrderDetails.ProductID = CMRC_Products.ProductID

WHERE
    OrderID = @OrderID

END
```

存储过程 CMRC_OrdersDetail 在设计上采用了多步返回记录集的形式，而且在多步返回记录集的基础

上，加入了数据集之间的一对多的数据关系。首先是根据参数表中的 OrderID 和 CustomerID 返回该比订单主体信息中的订购日期 OrderDate 和送达日期 ShipDate，如果返回的记录条数为 1，则说明通过 OrderID 和 CustomerID，在数据库中能够确定唯一的订单主体信息。返回该订单的累计购物总额，然后返回该订单的详细购物信息记录集。

7.2.4 数据访问层

购物订单模块的数据访问层位于 Components 文件夹下的 OrdersDB.cs，命名空间为 ASPNET.StarterKit.Commerce，类名 OrdersDB。以下是该类的代码清单。

```
using System;
using System.Configuration;
using System.Data;
using System.Data.SqlClient;

namespace ASPNET.StarterKit.Commerce {
```

和前面分析过的商品浏览模块的数据访问层一样，购物订单模块的数据访问层中同样将订单抽象成为一个对象。订单对象包含 4 个属性：订购日期、送达日期、订单总额，以及包含订单详细购物信息的 DataSet 型变量 OrderItems。类 OrderDetails 实现了这个对象，但是类 OrderDetails 并没把订单号 OrderID 作为一个属性定义，是因为 OrderDetails 类仅仅作为一个数据载体被实例化和调用。

```
public class OrderDetails {
    //定义 DateTime 型变量订单日期 OrderDate
    public DateTime OrderDate;
    //定义 DateTime 型变量订单送达日期 ShipDate
    public DateTime ShipDate;
    //定义 decimal 型变量订单总金额 OrderTotal
    public decimal OrderTotal;
    //定义 DataSet 型变量订单详细信息 OrderItems
    public DataSet OrderItems;
}

public class OrdersDB {
```

GetCustomerOrders 方法是返回一个 SqlDataReader 型用户历史订单列表数据集的方法。该方法首先实例化数据库连接和数据库命令，然后调用存储过程 CMRC.OrdersList，并为该存储过程添加 SqlParameter 类型的参数 parameterCustomerid。最终执行该存储过程，返回 SqlDataReader 类型的数据集。

```
public SqlDataReader GetCustomerOrders(String customerID)
{
    //实例化一个数据库连接对象 myConnection
    SqlConnection myConnection = new SqlConnection(
        SqlConnection(ConfigurationSettings.AppSettings["ConnectionString"]));

    //实例化一个使用 myConnection 执行存储过程 CMRC.OrdersList 的数据库命令 myCommand
    SqlCommand myCommand = new SqlCommand("CMRC.OrdersList", myConnection);

    //设置 myCommand 的类型为执行存储过程
    myCommand.CommandType = CommandType.StoredProcedure;

    //定义一个 SqlParameter 类型的存储过程参数 parameterCustomerid
```

```
SqlParameter parameterCustomerid = new SqlParameter("@CustomerID", SqlDbType.Int, 4);
//设置 parameterCustomerid 的值为方法的参数 customerID 的值
parameterCustomerid.Value = Int32.Parse(customerID);
//将 parameterCustomerid 添加到数据库命令 myCommand 中去
myCommand.Parameters.Add(parameterCustomerid);

//打开数据库连接 myConnection
myConnection.Open();
//执行数据库命令获得 SqlDataReader 型数据集 result
SqlDataReader result = myCommand.ExecuteReader(CommandBehavior.CloseConnection);

//返回 SqlDataReader 型数据集 result
return result;
}
```

GetOrderDetails 方法是返回一个 OrderDetails 型的订单详细信息的方法。首先实例化数据库连接和数据库适配器，调用存储过程 CMRC_OrdersDetail 并将 myCommand.SelectCommand.CommandType 设置为存储过程类型。然后为存储过程设置参数，最后为实例化一个 DataSet 对象 myDataSet，调用 myCommand.Fill 方法填充存储过程返回的订单详细购物信息的记录集。因为存储过程返回的是多结果集，因此要注意将调用 myCommand.Fill 方法填充 myDataSet 放到获取其他参数后面。判断数据库命令 myCommand 执行存储过程返回的结果集是否为空，如果不为空，则实例化一个 OrderDetails 类型的对象 myOrderDetails，将存储过程返回的订单的详细信息存储到 myOrderDetails 中，并将 myOrderDetails 返回。如果数据库命令 myCommand 执行存储过程返回的结果集为空，则返回一个空对象。

```
public OrderDetails GetOrderDetails(int orderID, string customerID)
{
    //实例化一个数据库连接对象 myConnection
    SqlConnection myConnection = new SqlConnection(ConfigurationSettings.AppSettings["ConnectionString"]);
    //实例化一个使用 myConnection 执行存储过程 CMRC_OrdersDetail 的数据库适配器 SqlDataAdapter
    SqlDataAdapter myCommand = new SqlDataAdapter("CMRC_OrdersDetail", myConnection);

    //设置 myCommand 的类型为执行存储过程
    myCommand.SelectCommand.CommandType = CommandType.StoredProcedure;

    //定义一个 SqlParameter 类型的存储过程参数 parameterOrderID
    SqlParameter parameterOrderID = new SqlParameter("@OrderID", SqlDbType.Int, 4);
    //设置 parameterOrderID 的值为方法的参数 orderID 的值
    parameterOrderID.Value = orderID;
    //将 parameterCustomerid 添加到数据库适配器 myCommand 的 SelectCommand 参数表中
    myCommand.SelectCommand.Parameters.Add(parameterOrderID);

    //定义一个 SqlParameter 类型的存储过程参数 parameterCustomerID
    SqlParameter parameterCustomerID = new SqlParameter("@CustomerID", SqlDbType.Int, 4);
    //设置 parameterCustomerID 的值为方法的参数 customerID 的值
    parameterCustomerID.Value = Int32.Parse(customerID);
    //将 parameterCustomerID 添加到数据库适配器 myCommand 的 SelectCommand 参数表中
    myCommand.SelectCommand.Parameters.Add(parameterCustomerID);

    //定义一个 SqlParameter 类型的存储过程参数 parameterOrderDate
```

```
SqlParameter parameterOrderDate = new SqlParameter("@OrderDate", SqlDbType.DateTime, 8);
//设置 parameterOrderDate 的类型为返回值型参数
parameterOrderDate.Direction = ParameterDirection.Output;
//将 parameterOrderDate 添加到数据库适配器 myCommand 的 SelectCommand 参数表中
myCommand.SelectCommand.Parameters.Add(parameterOrderDate);

//定义一个 SqlParameter 类型的存储过程参数 parameterShipDate
SqlParameter parameterShipDate = new SqlParameter("@ShipDate", SqlDbType.DateTime, 8);
//设置 parameterShipDate 的类型为返回值型参数
parameterShipDate.Direction = ParameterDirection.Output;
//将 parameterShipDate 添加到数据库适配器 myCommand 的 SelectCommand 参数表中
myCommand.SelectCommand.Parameters.Add(parameterShipDate);

//定义一个 SqlParameter 类型的存储过程参数 parameterOrderTotal
SqlParameter parameterOrderTotal = new SqlParameter("@OrderTotal", SqlDbType.Money, 8);
//设置 parameterOrderTotal 的类型为返回值型参数
parameterOrderTotal.Direction = ParameterDirection.Output;
//将 parameterOrderTotal 添加到数据库适配器 myCommand 的 SelectCommand 参数表中
myCommand.SelectCommand.Parameters.Add(parameterOrderTotal);

//实例化一个 DataSet myDataSet
DataSet myDataSet = new DataSet();
//调用 myCommand.Fill 方法填充 myDataSet
myCommand.Fill(myDataSet, "OrderItems");

//判断 parameterShipDate 是否为空，如果不为空，则说明存储过程返回的记录集不为空
if (parameterShipDate.Value != DBNull.Value) {

    //实例化一个 OrderDetails 类的对象 myOrderDetails
    OrderDetails myOrderDetails = new OrderDetails();
    //为 myOrderDetails 的属性赋值
    myOrderDetails.OrderDate = (DateTime)parameterOrderDate.Value;
    myOrderDetails.ShipDate = (DateTime)parameterShipDate.Value;
    myOrderDetails.OrderTotal = (decimal)parameterOrderTotal.Value;
    myOrderDetails.OrderItems = myDataSet;

    //返回 myOrderDetails
    return myOrderDetails;
}
else
    //返回空值
    return null;
}
```

在完整的电子商务流程中，订单产生之后得到管理员的确认就可以对货物进行配送，商品送达给客户的日期为商品的送达日期。但是在 ASP.NET Commerce Starter Kit 中并没有商品配送的应用逻辑，为了保证数据的完整性，商品的送达日期采用随机产生的形式。CalculateShippingDate 方法就是产生订单送达日期的方法。利用随机数发生器机制产生 0 到 3 之内的 double 型数字和当前时间累加得出一个新的日期作为订单送达日期。

```
public DateTime CalculateShippingDate(String customerID, string cartID) {
```

```
//定义一个随机数发生器
Random x = new Random();
//产生随机数
double myrandom = (double)x.Next(0, 3);
//产生的随机数与当前日期进行累加得出新的日期
return DateTime.Now.AddDays(myrandom);
}
```

PlaceOrder 方法是添加一笔订单全部信息的方法，其中包括添加订单主体信息和订单详细购物信息。该方法首先实例化数据库连接和数据库命令，执行存储过程 CMRC_OrdersAdd 添加订单的详细信息，最后返回所添加的订单 parameterOrderID。这个方法是在第 6 章 Web 商店模块中的用户确定订单被调用到的。

```
public int PlaceOrder(string customerID, string cartID)
{
    //实例化一个数据库连接对象 myConnection
    SqlConnection myConnection = new SqlConnection(ConfigurationSettings.AppSettings["ConnectionString"]);
    //实例化一个使用 myConnection 执行存储过程 CMRC_OrdersList 的数据库命令 myCommand
    SqlCommand myCommand = new SqlCommand("CMRC_OrdersAdd", myConnection);

    //设置 myCommand 的类型为执行存储过程
    myCommand.CommandType = CommandType.StoredProcedure;

    //为存储过程添加参数
    SqlParameter parameterCustomerID = new SqlParameter("@CustomerID", SqlDbType.Int, 4);
    parameterCustomerID.Value = Int32.Parse(customerID);
    myCommand.Parameters.Add(parameterCustomerID);

    SqlParameter parameterCartID = new SqlParameter("@CartID", SqlDbType.NVarChar, 50);
    parameterCartID.Value = cartID;
    myCommand.Parameters.Add(parameterCartID);

    SqlParameter parameterShipDate = new SqlParameter("@ShipDate", SqlDbType.DateTime, 8);
    parameterShipDate.Value = CalculateShippingDate(customerID, cartID);
    myCommand.Parameters.Add(parameterShipDate);

    SqlParameter parameterOrderDate = new SqlParameter("@OrderDate", SqlDbType.DateTime, 8);
    parameterOrderDate.Value = DateTime.Now;
    myCommand.Parameters.Add(parameterOrderDate);

    SqlParameter parameterOrderID = new SqlParameter("@OrderID", SqlDbType.Int, 4);
    parameterOrderID.Direction = ParameterDirection.Output;
    myCommand.Parameters.Add(parameterOrderID);

    //打开数据库连接 myConnection
    myConnection.Open();
    //执行数据库命令 myCommand
    myCommand.ExecuteNonQuery();
    //关闭数据库连接 myConnection
    myConnection.Close();
}
```

```
//返回存储过程的参数 parameterOrderID  
return (int)parameterOrderID.Value;  
}  
}  
}
```

7.2.5 业务逻辑层

(1) 历史订单列表页面后台编码类 OrderList.aspx.cs

历史订单列表页面 OrderList.aspx 的后台编码类为 OrderList.aspx.cs。该后台编码类在页面载入方法 Page_Load 中首先从 User.Identity.Name 获得当前用户的 CustomerID，根据用户的 CustomerID 获得用户历史订单列表的数据集，将该数据集绑定到页面控件 MyList。以下是页面载入方法 Page_Load 的代码清单。

```
private void Page_Load(object sender, System.EventArgs e) {  
    //从 User.Identity.Name 中取得 customerID  
    String customerID = User.Identity.Name;  
  
    //实例化用户订单模块的数据库访问层 OrdersDB 类成为对象 orderHistory  
    ASPNET.StarterKit.Commerce.OrdersDB orderHistory = new ASPNET.StarterKit.Commerce.OrdersDB();  
    //调用 orderHistory 的 GetCustomerOrders 方法获得用户历史订单列表数据集并设置为 MyList 的数据源  
    MyList.DataSource = orderHistory.GetCustomerOrders(customerID);  
    //绑定 MyList  
    MyList.DataBind();  
  
    //判断 MyList 控件的数据条目是否为 0  
    if (MyList.Items.Count == 0) {  
        //如果 MyList 控件的数据条目为 0，则显示提示信息并隐藏 MyList  
        MyError.Text = "You have no orders to display.";  
        MyList.Visible = false;  
    }  
}
```

(2) 订单详细购物信息页面后台编码类 OrderDetails.aspx.cs

订单详细购物信息页面 OrderDetails.aspx 的后台编码类为 OrderDetails.aspx.cs。在后台编码类中首先定义了页面中使用的控件，包括显示订单主体信息的控件和显示订单详细信息的 DataGrid 控件 GridControl1。

```
//定义页面使用的控件  
protected System.Web.UI.WebControls.Label lblOrderNumber;  
protected System.Web.UI.WebControls.Label lblOrderDate;  
protected System.Web.UI.WebControls.Label lblShipDate;  
protected System.Web.UI.WebControls.DataGrid GridControl1;  
protected System.Web.UI.WebControls.Label MyError;  
protected System.Web.UI.HtmlControls.HtmlTable detailsTable;  
protected System.Web.UI.WebControls.Label lblTotal;
```

在页面载入方法 Page_Load 中首先实例化数据库连接和数据库对象，从数据访问层中返回的对象 myOrderDetails 中取得订单的信息，并绑定到页面控件中。

```
private void Page_Load(object sender, System.EventArgs e) {
```

```

//获得页面 URL 中变量 OrderID
int OrderID = Int32.Parse(Request.Params["OrderID"]);

//从 User.Identity.Name 获得当前用户的 CustomerId
string CustomerId = User.Identity.Name;

//实例化用户购物订单模块的数据访问层 OrdersDB 类成为对象 orderHistory
ASPNET.StarterKit.Commerce.OrdersDB orderHistory = new ASPNET.StarterKit.Commerce.OrdersDB();
//调用 orderHistory 对象的 GetOrderDetails 方法实例化 OrderDetails 类成为对象 myOrderDetails
ASPNET.StarterKit.Commerce.OrderDetails myOrderDetails = orderHistory.GetOrderDetails(OrderID,
CustomerId);

//判断 myOrderDetails 是否为空
if (myOrderDetails != null) {

    //获得 myOrderDetails 对象的 OrderItems 属性并设置成为控件的 GridControl1 数据源
    GridControl1.DataSource = myOrderDetails.OrderItems;
    //绑定 GridControl1
    GridControl1.DataBind();

    //绑定显示订单主体信息的控件
    lblTotal.Text = String.Format("{0:c}", myOrderDetails.OrderTotal);
    lblOrderNumber.Text = OrderID.ToString();
    lblOrderDate.Text = myOrderDetails.OrderDate.ToShortDateString();
    lblShipDate.Text = myOrderDetails.ShipDate.ToShortDateString();
}

else {
    //myOrderDetails 为空则显示错误信息并隐藏 detailsTable
    MyError.Text = "Order not found!";
    detailsTable.Visible = false;
}
}
}

```

7.2.6 用户表示层

(1) 历史订单列表页面 OrderList.aspx

历史订单列表页面 OrderList.aspx 为显示用户历史订单列表的页面，用户历史订单列表为这个页面的主体以 DataGrid 控件的形式展现。以下是用户历史订单列表的前台实现代码清单。

```

<!-- 定义 DataGrid 的主要属性 -->
<asp:DataGrid id="MyList" width="90%" BorderColor="black" GridLines="Vertical" cellpadding="4" cellspacing="0"
Font-Name="Verdana" Font-Size="8pt" ShowFooter="true" HeaderStyle-CssClass="CartListHead" FooterStyle-CssClass=
"cartlistfooter" ItemStyle-CssClass="CartListItem" AlternatingItemStyle-CssClass="CartListItemAlt"
AutoGenerateColumns="false" runat="server">

<!-- 定义 DataGrid 的行集合 -->
<Columns>
    <asp:BoundColumn HeaderText="Order ID" DataField="OrderID" />
    <asp:BoundColumn HeaderText="Order Date" DataField="OrderDate" DataFormatString="{0:d}" />
    <asp:BoundColumn HeaderText="Order Total" DataField="OrderTotal" DataFormatString="{0:c}" />
    <asp:BoundColumn HeaderText="Ship Date" DataField="ShipDate" DataFormatString="{0:d}" />

```

```
<asp:HyperLinkColumn HeaderText="Show Details" Text="Show Details" DataNavigateUrlField="OrderID"
DataNavigateUrlFormatString="orderdetails.aspx?OrderID={0}" />
</Columns>
</asp:DataGrid>
```

(2) 订单详细购物信息页面 OrderDetails.aspx

订单详细购物信息页面 OrderDetails.aspx 为显示某笔订单的详细购物信息的页面，包括订单的主体信息和每一件商品的购买详细信息。其中每一件商品的购买详细信息列表构成了订单详细购物信息页面的主体并以 DataGrid 控件的形式展现。以下是商品的购买详细信息列表的前台实现代码清单。

```
<!-- 定义 DataGrid 的主要属性 -->
<asp:DataGrid id="GridControl1" width="90%" BorderColor="black" GridLines="Vertical" cellpadding="4"
cellspacing="0" Font-Name="Verdana" Font-Size="8pt" ShowFooter="true" HeaderStyle-CssClass="CartListHead"
FooterStyle-CssClass="cartlistfooter" ItemStyle-CssClass="CartListItem"
AlternatingItemStyle-CssClass="CartListItemAlt" AutoGenerateColumns="false" runat="server">
<!-- 定义 DataGrid 的行集合 -->
<Columns>
    <asp:BoundColumn HeaderText="Product Name" DataField="ModelName" />
    <asp:BoundColumn HeaderText="Model Number" DataField="ModelNumber" />
    <asp:BoundColumn HeaderText="Quantity" DataField="Quantity" />
    <asp:BoundColumn HeaderText="Price" DataField="UnitCost" DataFormatString="{0:c}" />
    <asp:BoundColumn HeaderText="Subtotal" DataField="ExtendedAmount" DataFormatString="{0:c}" />
</Columns>
</asp:DataGrid>
```

7.3 开发启示

购物订单模块为客户提供了查看历史订单和订单详细信息的功能。由于订单和商品详细购买信息在数据结构上存在着一对多的关系，因此在数据的查询和数据的显示上要同时抽取一笔订单的全部信息就要考虑到如何返回这种一对多的数据集。首先在存储过程的设计上采用了一个存储过程返回多个数据集的方式，在数据访问层使用 DataSet 对象作为订单对象的某个属性来存储商品详细购买信息的方案，最终在表示层通过 DataGrid 控件将商品详细购买信息显示到订单详细信息页面。利用 ASP.NET 中数据访问和数据展现技术的特点，结合面向对象的思想和设计模式，不仅可以从数据库查询简单记录集返回到客户端，而且可以传递包含有数据关系的数据对象。

作为一个完整的电子商务模型，管理员需要在管理平台上能够查看最新的购物订单，并且根据购物订单配送商品，在商品送达给客户之后要将订单的状态设置为全部完成。所以如果以 ASP.NET Commerce Starter Kit 为基础开发一个完整的电子商务应用，还应该添加和扩展购买订单模块的如下功能。

- 订单的后台查看。可以根据订单的不同属性分类查看订单，根据订单信息进行配货。
- 订单的后台统计。可以根据订单的数据生成有助于管理员分析 Web 商店运营状况的数据报表。
- 订单的状态管理。当管理员确定订单之后，应该将订单的状态设置为已处理；当管理员因为商品的短缺或是其他原因不能正常完成订单的配送，应该将订单的状态设置为等待；当订单正常送达之后，应该将订单的状态已送达。订单的状态设置可以让用户直观地了解到订单的目前情况。

第8章 构建 Web Service

ASP.NET Commerce Starter Kit 以 Web Service 的形式提供了和其他应用程序通信的方式，在第 3 章技术点解析中已经介绍和讲解了 Web Service 的概念、特点和实现。设计者将这项功能强大的新技术应用到了 ASP.NET Commerce Starter Kit 中，第三方应用程序可以通过 Web Service 与 ASP.NET Commerce

Starter Kit 实现数据方面的交互，其中包括订单的提交和订单状态的查询。表 8-1 为本章知识点索引。

表 8-1 本章知识点索引

| 知 识 点 | 位 置 |
|--|-------|
| ASP.NET Commerce Starter Kit 中 Web Service 的架构 | 8.1.1 |
| 电子商务应用程序可以通过 Web Service 实现的功能 | 8.1.2 |
| SOAP 响应返回订单详细信息 | 8.2.1 |
| 第三方调用 Web Service | 8.2.3 |

8.1 系统设计

8.1.1 需求分析

在电子商务应用程序的设计和实现中，与其他系统的数据交互显得尤为重要。例如可以将商品的信息发布到第三方应用程序中，也可以接受第三方应用程序提交的订单数据，同样可以接受来自第三方应用程序的数据查询。没有良好的数据交互接口，孤立的电子商务应用程序在交互性和扩展性方面受到了极大的限制，通过传统的 Web 架构显然无法满足这样的需求。

解决数据交互问题的最好方案就是提供一个通用的数据交互接口，第三方应用程序以一个公共的标准调用这个交互接口提供的方法，完成与系统的交互。Web Service 的应用就是为了提供通用数据交互接口。Web Service 作为一座桥梁，连接电子商务应用程序的核心业务逻辑和第三方应用程序，图 8-1 就是这种架构的一个简单的描述。

通过架构图可以看到，第三方应用程序最终的目的还是与 ASP.NET Commerce Starter Kit 的数据库进行交互，不过与第三方应用程序直接发生交互的是 Web Service 提供的接口。设计

者在 Web Service 中定义了可以被第三方应用程序调用的逻辑方法，这些逻辑方法也是通过调用 ASP.NET Commerce Starter Kit 数据访问层的对象和方法完成第三方应用程序的请求。因此，Web Service 中提供的逻辑操作方法在 ASP.NET Commerce Starter Kit 的 Web 应用程序中有其对应的功能模块，Web Service 相当于提供了 Web 页面中部门功能的另外一种实现方式，而将这些功能提供给第三方应用程序。

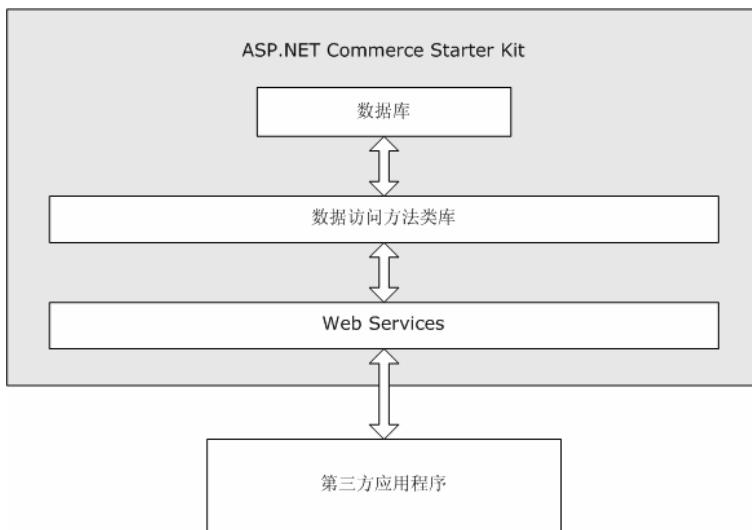


图 8-1 ASP.NET Commerce Starter Kit 中的 Web Service 架构

8.1.2 功能设计

一个电子商务应用程序可以在 Web Service 中提供的接口和方法很多，Web Service 这个技术框架的提出就是考虑到满足电子商务应用程序的实际需求。电子商务应用程序可以通过 Web Service 实现的功能分为以下几类。

(1) 提供数据

电子商务应用程序的提供商作为数据的提供者，通过 Web Service 将商品信息提供给用户，用户可以不登录电子商务的 Web 站点就获得最新的商品信息或是促销信息。

(2) 订单数据的集成处理

电子商务应用程序可以接收来自第三方应用程序的订单数据，并且可以接收第三方应用程序的订单状态查询请求。接收第三方应用程序提供的订单数据从另外一个方面讲扩展了电子商务的应用范围，打破了仅能在传统 Web 站点接收订单数据的局限。

(3) 与合作伙伴之间的应用集成

电子商务的提供商需要经常与商品提供商等其他合作伙伴之间进行大量的数据交互，通过 Web Service 可以进行灵活的信息沟通。

(4) 企业内部应用集成

电子商务的提供商在工作或组织内部可能存在着其他应用平台，例如内部的数据分析系统、公司的财务管理系统等等，通过 Web Service 可以实现内部应用的无缝集成。

ASP.NET Commerce Starter Kit 中的 Web Service 提供了两个可以被调用的方法：订单数据的接收和订单状态的查询。

订单数据接收的方法为远程的终端提供了一个向系统提交订单的方法。该方法接收包括用户名、密码、订购商品和订购数量在内的信息，生成一条订单信息之后返回订单日期、送货日期、订单总金额以及订单详细购物信息。

订单状态查询的方法为远程的终端提供了一个查询订单状态的方法。该方法接收包括用户名、密码、订单号在内的信息，查询订单数据之后返回订单日期、送货日期、订单总金额以及订单详细购物信息。

8.2 实现 Web Service

8.2.1 实现效果

ASP.NET Commerce Starter Kit 提供的 Web Service 文件为 InstantOrder.asmx，直接访问该文件会提供该 Web Service 的说明。页面效果如图 8-2 所示。

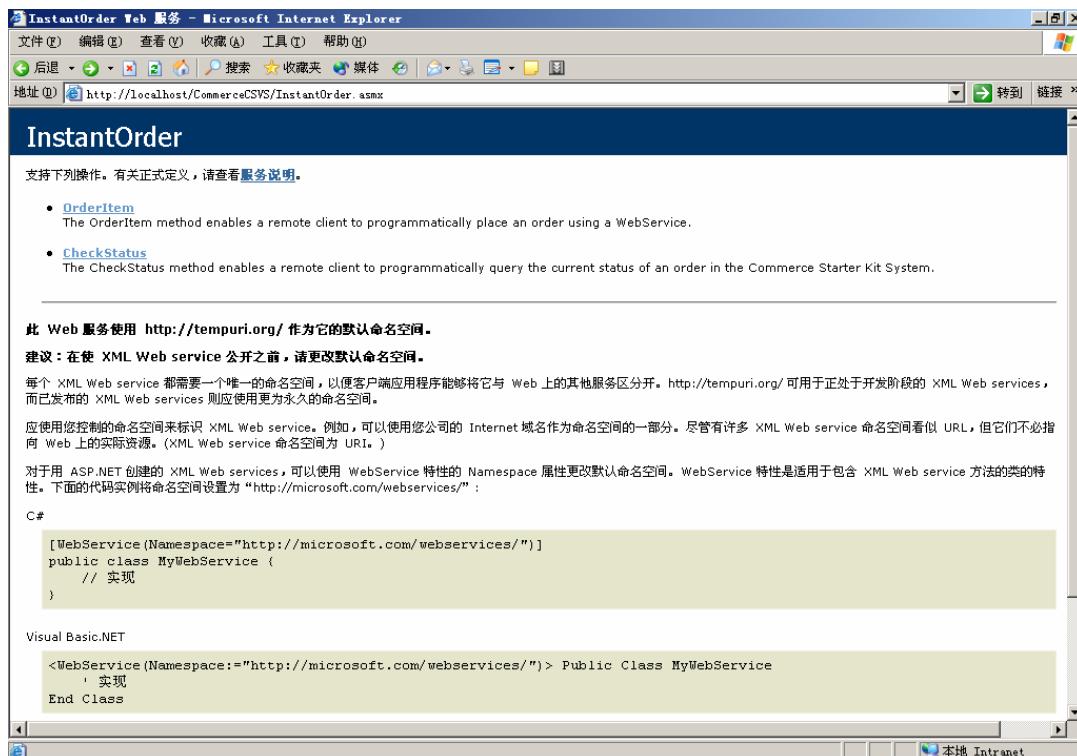


图 8-2 InstantOrder.asmx 页面效果图

该页面列出了 Web Service 提供的两个方法的链接，单击链接可以看到两个方法的详细信息。

注意：在实现效果中所展示的页面皆为 IIS 根据 Web Service 自动生成的 Web 页面，包括 HTTP POST 调用示例，SOAP 请求与响应。实际的 Web Service 是没有应用界面的。

(1) OrderItem 方法

OrderItem 方法是可以被远程调用添加一笔订单信息的方法。单击 OrderItem 方法的链接，可以查看 OrderItem 方法的详细信息。首先是提供了一个 HTTP POST 方式的调用测试，效果如图 8-3 所示。

页面中提示输入 Web Service 接收的参数，当单击调用按钮的时候将数据以 HTTP POST 的方式提交给 Web Service。

OrderItem

The OrderItem method enables a remote client to programmatically place an order using a WebService.

测试

若要使用 HTTP POST 协议对操作进行测试，请单击“调用”按钮。

| 参数 | 值 |
|------------|----------------------|
| userName: | <input type="text"/> |
| password: | <input type="text"/> |
| productID: | <input type="text"/> |
| quantity: | <input type="text"/> |

调用

图 8-3 OrderItem 方法的 HTTP POST 调用表单

以下列出该方法的 SOAP 请求代码。

```
<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
    <soap:Body>
        <OrderItem xmlns="http://tempuri.org/">
            <userName>string</userName>
            <password>string</password>
            <productID>int</productID>
            <quantity>int</quantity>
        </OrderItem>
    </soap:Body>
</soap:Envelope>
```

SOAP 请求中定义了 OrderItem 方法接收的数据内容以及数据格式。该方法接收的参数类如表 8-2 所示。

表 8-2 OrderItem 方法参数列表

| 参数名 | 参数类型 | 说明 |
|-----------|--------|----------|
| userName | string | 订购商品的用户名 |
| password | string | 用户密码 |
| productID | int | 订购商品 ID |
| quantity | int | 订购商品数量 |

OrderItem 方法的 SOAP 响应示例如下：

```
<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
    <soap:Body>
        <OrderItemResponse xmlns="http://tempuri.org/">
            <OrderItemResult>
                <OrderDate>dateTime</OrderDate>
                <ShipDate>dateTime</ShipDate>
                <OrderTotal>decimal</OrderTotal>
                <OrderItems>
                    <xsd:schema>schema</xsd:schema>xml</OrderItems>
                </OrderItemResult>
            </OrderItemResponse>
        </soap:Body>
    </soap:Envelope>
```

```
</soap:Body>
</soap:Envelope>
```

在 SOAP 响应的 OrderItemResponse 节点中定义了订单被接收之后返回的信息，表 8-3 是返回信息的说明。值得注意的是，商品的详细订购信息是以 XML 形式返回的，在 OrderItems 节点首先返回一个 XML 的格式说明 xsd，该段 xsd 是为了解释商品详细订购信息的 XML 片断，然后返回商品详细订购信息的 XML。

表 8-3

OrderItem 方法 SOAP 响应信息列表

| 返 回 字 段 | 类 型 | 说 明 |
|------------|----------|----------|
| OrderDate | datetime | 订购日期 |
| ShipDate | datetime | 订单送达日期 |
| OrderTotal | decimal | 订单总金额 |
| OrderItems | xml | 订单详细购买信息 |

(2) CheckStatus 方法

CheckStatus 方法是点击 CheckStatus 方法的链接，可以查看 CheckStatus 方法的详细信息。首先提供了一个 HTTP POST 方式的调用测试，效果如图 8-4 所示。

CheckStatus

The CheckStatus method enables a remote client to programmatically query the current status of an order in the Commerce Starter Kit System.

测试

若要使用 HTTP POST 协议对操作进行测试，请单击“调用”按钮。

| 参数 | 值 |
|-----------|----------------------|
| userName: | <input type="text"/> |
| password: | <input type="text"/> |
| orderID: | <input type="text"/> |
| 调用 | |

图 8-4 CheckStatus 方法的 HTTP POST 调用表单

下面给出该方法的 SOAP 请求代码。

```
<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
    <soap:Body>
        <CheckStatus xmlns="http://tempuri.org/">
            <userName>string</userName>
            <password>string</password>
            <orderID>int</orderID>
        </CheckStatus>
    </soap:Body>
</soap:Envelope>
```

SOAP 请求中定义了 CheckStatus 方法接收的数据内容以及数据格式。该方法接收的参数类如表 8-4 所示。

表 8-4

CheckStatus 方法参数列表

| 参 数 名 | 参 数 类 型 | 说 明 |
|----------|---------|----------|
| userName | string | 订购商品的用户名 |
| password | string | 用户密码 |
| orderID | int | 商品订单 ID |

CheckStatus 方法的 SOAP 响应如下：

```
<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
    <soap:Body>
        <CheckStatusResponse xmlns="http://tempuri.org/">
            <CheckStatusResult>
                <OrderDate>dateTime</OrderDate>
                <ShipDate>dateTime</ShipDate>
                <OrderTotal>decimal</OrderTotal>
                <OrderItems>
                    <xsd:schema>schema</xsd:schema>xm1</OrderItems>
                </CheckStatusResult>
            </CheckStatusResponse>
        </soap:Body>
    </soap:Envelope>
```

CheckStatus 方法的 SOAP 响应中的数据格式和 OrderItem 方法返回的响应信息是相同的，都是返回一条订单的主体信息和详细购物信息。因此省略了本方法的 SOAP 响应具体说明。

8.2.2 实现细节

InstantOrder.asmx 为 Web Service 的实现文件，前台的 InstantOrder.asmx 文件只有简单的调用声明，并没有具体的实现逻辑。后台的 InstantOrder.asmx.cs 中有该 Web Service 的具体实现代码。

```
using System;
using System.Web.Services;
using ASPNET.StarterKit.Commerce;

public class InstantOrder : WebService
```

OrderItem 方法的 SOAP 响应实际上是一个订单对象中的部分信息，因此在设计上返回了购物订单模块中数据库访问层的 OrderDetails 对象存储订单信息。该方法首先调用了用户模块数据访问层的 Login 方法，验证用户身份信息的合法性，然后通过调用购物订单模块数据访问层的方法添加一条订单记录。

```
{
    //Web Service 方法描述
    [WebMethod(Description="The OrderItem method enables a remote client to programmatically place an order
using a WebService.", EnableSession=false)]
    //定义方法 OrderItem
    public OrderDetails OrderItem(string userName, string password, int productID, int quantity) {

        //实例化一个 accountSystem 对象
        ASPNET.StarterKit.Commerce.CustomersDB accountSystem = new ASPNET.StarterKit.Commerce.CustomersDB();
        //调用 accountSystem 对象的 Login 方法判断用户身份的合法性
        String customerId = accountSystem.Login(userName, ASPNET.StarterKit.Commerce.Components.Security.Encrypt(password));
        //如果返回的 customerId 为空，则说明当前用户身份不合法
        if (customerId == null) {
            throw new Exception("Error: Invalid Login!");
        }
    }
}
```

```
//判断用户输入的订购数量的合法性，是否为合理的数字
int qty = System.Math.Abs(quantity);
if (qty == quantity && qty < 1000) {

    //实例化一个 myShoppingCart 购物车对象
    ASPNET.StarterKit.Commerce.ShoppingCartDB myShoppingCart = new ASPNET.StarterKit.Commerce.ShoppingCartDB();
    //添加一条购物信息
    myShoppingCart.AddItem(customerId, productID, qty);

    //实例化一个 orderSystem 订单对象
    ASPNET.StarterKit.Commerce.OrdersDB orderSystem = new ASPNET.StarterKit.Commerce.OrdersDB();
    //调用 orderSystem 的 PlaceOrder 方法将用户购物车中的信息生成一笔订单
    int orderID = orderSystem.PlaceOrder(customerId, customerId);

    //返回生成订单的详细信息
    return orderSystem.GetOrderDetails(orderID, customerId);
}
else {
    //如果输入的数据不合法，返回空对象
    return null;
}
}
```

CheckStatus 方法与 OrderItem 方法相同，在 SOAP 响应中也是订单对象的部分信息，因此也返回一个 OrderDetails 对象，该方法首先实例化一个用户模块数据访问层的对象，验证用户身份的合法性，然后返回请求的订单详细信息。

```
//Web Service 方法描述
[WebMethod(Description="The CheckStatus method enables a remote client to programmatically query the current status of an order in the Commerce Starter Kit System.", EnableSession=false)]
//定义方法 CheckStatus
public OrderDetails CheckStatus(string userName, string password, int orderID) {

    //实例化一个 accountSystem 对象
    ASPNET.StarterKit.Commerce.CustomersDB accountSystem = new ASPNET.StarterKit.Commerce.CustomersDB();
    //调用 accountSystem 对象的 Login 方法判断用户身份的合法性
    String customerId = accountSystem.Login(userName, ASPNET.StarterKit.Commerce.Components.Security.Encrypt(password));
    //如果返回的 customerId 为空，则说明当前用户身份不合法
    if (customerId == null) {
        throw new Exception("Error: Invalid Login!");
    }

    //实例化一个 orderSystem 订单对象
    ASPNET.StarterKit.Commerce.OrdersDB orderSystem = new ASPNET.StarterKit.Commerce.OrdersDB();
    //调用 orderSystem 的 GetOrderDetail 方法返回一个订单的详细信息
    return orderSystem.GetOrderDetails(orderID, customerId);
}
}
```

8.2.3 第三方调用 Web Service

第三方调用 Web Service 的方法在第 3 章中已经详细介绍过，下面演示调用 ASP.NET Commerce Starter Kit 的 Web Service 的详细步骤。

在 Visual Studio.NET 中新建工程，添加 Web 引用，并填入需要引用的 Web Service 的 URL，ASP.NET Commerce Starter Kit 中的 Web Service 的 URL 应该为：http://[Hostname]/[Path]/InstantOrder.asmx，其中[Hostname]为 Web 主机的名称，[Path]为 ASP.NET Commerce Starter Kit 的 Web 虚拟目录名称。如图 8-5 所示。

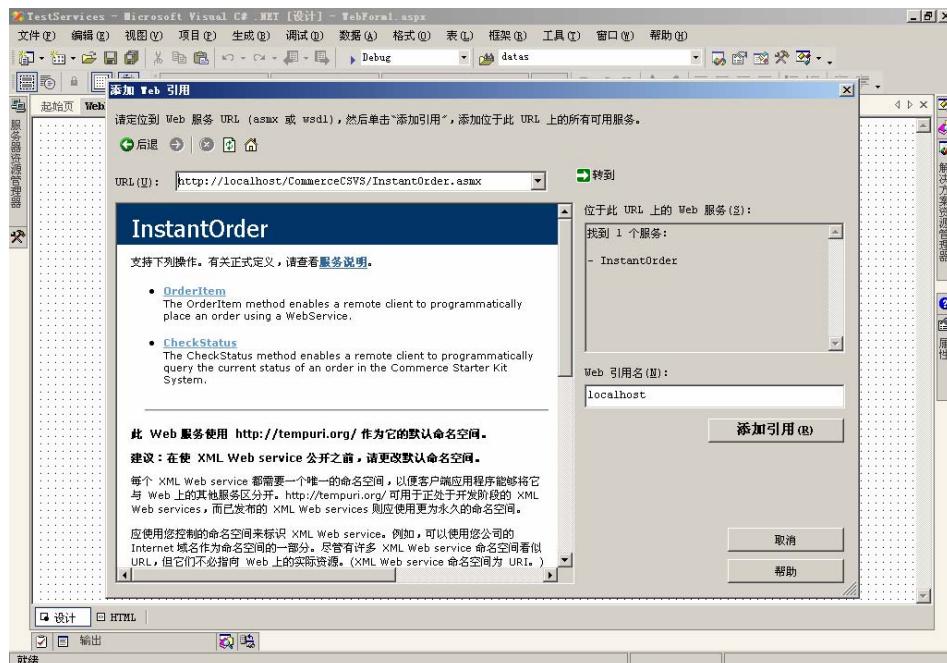


图 8-5 在第三方调用中添加 Web 引用

设置该 Web Service 的 Web 引用名之后，单击添加引用的按钮，就完成了对 Web Service 的引用。假设 Web 引用名为 localhost，则在应用程序中使用下列代码片断进行对该 Web Service 的引用。

```
//实例化一个myinstantorder对象对Web Service InstantOrder进行调用
localhost.InstantOrder myinstantorder = new TestServices.localhost.InstantOrder();
//实例化一个myOrderDetail对象
OrderDetail myOrderDetail = new OrderDetail;
//调用myinstantorder对象的OrderItem方法为myOrderDetail对象赋值
myOrderDetail = myinstantorder.OrderItem(username,password,productID,quantity);
```

通过以上的代码可以看到，添加了 Web 引用之后，第三方应用程序的开发者就像调用项目中类库的方法一样调用 Web Service 方法。

注意：实例化的 myOrderDetail 对象是在第三方应用程序中构建的一个对象，该对象和服务器端的订单对象并没有直接联系。开发者可以根据实际需求，参考 SOAP 响应中的信息构建类似对象。

8.3 开发启示

本章讲解了 ASP.NET Commerce Starter Kit 中的 Web Service 的功能和具体实现，通过对 InstantOrder 的分析，可以了解到 Web Service 架构在电子商务应用程序中的使用细节。实际上 Web Service 技术在电子商务应用之外的其他应用程序中也能发挥极其重要的作用。Web Service 可以为应用程序提供较高的可扩展性，Web Service 相当于一个可以无限扩展的接口，为应用程序提供了极大的扩展空间。

ASP.NET Commerce Starter Kit 提供的 Web Service 只有两个方法，略显简单，但是设计者是想通过一个简单的事例演示 Web Service 的架构和理念。因为 Web Service 也是通过调用数据访问层的类库方法来实现具体的功能，所以从理论上讲，ASP.NET Commerce Starter Kit 在页面上实现的功能都可以 Web Service 的形式实现。如果在 ASP.NET Commerce Starter Kit 现有的 Web Service 基础上进行功能性的扩展，那么如下几个功能都是可以考虑的。

- 用户历史订单信息的查询；
- 获取商品列表；
- 获取商品详细信息。

第9章 扩展 Commerce

在前面章节的讲解中已经提到，ASP.NET Commerce Starter Kit 作为一个示例程序，并没有实现电子商务应用中全部的功能模块，而开发者可以以 ASP.NET Commerce Starter Kit 作为基础框架进行二次开发实现一个完整的电子商务应用程序。在二次开发的过程中，需要对 ASP.NET Commerce Starter Kit 现有的数

据结构、对象模型以及类库中的方法进行扩展和补充。本章主要讲解的是在二次开发过程中对 ASP.NET Commerce Starter Kit 的扩展和补充。

本章将结合实际情况，实现在 ASP.NET Commerce Starter Kit 中没有实现的一部分普通电子商务应用中的通用模块。目的是向读者展示如何在当前 ASP.NET Commerce Starter Kit 框架基础上搭建一个功能完备的电子商务 Web 应用程序。

9.1 订单管理

9.1.1 需求分析

在 ASP.NET Commerce Starter Kit 提供的应用程序中，并没有提供一个完备的订单管理模块。当用户提交订购申请之后，系统随机产生了一个送达日期，订单对象的处理也就随之终结。在实际的电子商务应用中，当一笔订单被提交之后会触发一系列的处理，其中包括订单确认、订单配送以及订单结束等。表现在系统中为如下步骤。

(1) 订单确认

电子商务提供商的管理员可以通过应用程序查看到系统中最新被提交的订单，一旦发现有新提交的订单，则查看订单详细信息，对订单进行确认。当然，在一些大型的电子商务应用程序中，这些处理都可以通过应用程序自动完成，但是对于 ASP.NET Commerce Starter Kit 来说，添加一个电子商务提供商管理订单的模块显得十分实用。

电子商务提供商的管理员在查看订单详细信息之后，会对该笔订单进行处理即订单配送前的准备工作，例如向商品管理部门发送取货通知、向商品配送部门发送配货通知等等，一切准备工作结束之后，该笔订单就进入配送阶段。此时，根据商品的管理部门和配

送部门的反馈，电子商务提供商的管理员应该知道该笔订单的送达日期，管理员会更改该笔订单的状态为已处理并且在系统中确定具体的送达日期。

(2) 订单配送

电子商务提供商的商品管理部门和商品配送部门在接到配送通知之后，会将该订单的状态更改为配送中，然后指派具体的工作人员对该笔订单进行配送，获得订单提交者的具体送达地点以及联系方式，完成商品的配送。

(3) 订单完成

电子商务提供商的配送部门工作人员在完成商品配送之后，会将信息反馈给电子商务提供商的管理员，管理员则会将订单信息中的订单状态更新为已完成。一旦订单的状态更新为已完成则说明该订单的处理已经结束，订单的流转周期也随之结束。

几个对于订单的处理步骤是连续的，用活动图表示如图 9-1 所示。

通过对订单管理流程的分析，结合 ASP.NET Commerce Starter Kit 的情况，为了完成订单管理流程中需要的功能，应该添加订单管理列表和订单信息更新两个页面，对于两个页面的功能描述如下。

● 订单管理列表页面 ManageOrders.aspx

该页面中列出系统中最新提交的订单，该页面中以列表的形式显示订单的订单号、提交日期、送达日期、订单提交者、订单当前状态，管理员可以点击每条订单的链接查看订单的信息并可以点击管理订单的链接进入订单信息更新页面。ManageOrders.aspx 页面效果如图 9-2 所示。

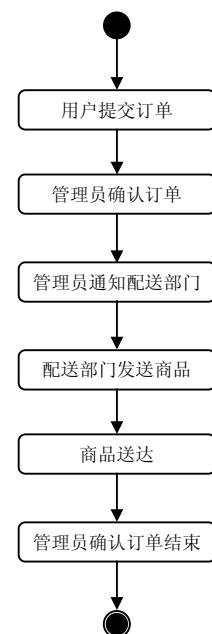


图 9-1 订单处理活动图

| Order ID | Order Date | Order Total | Ship Date | Customer Name | Order Status | Manage |
|----------|------------|-------------|------------|---------------|--------------|--------|
| 115 | 2005-7-11 | ¥ 299.99 | 2005-7-13 | xx | Dispose | Manage |
| 114 | 2005-7-11 | ¥ 79.97 | 2005-7-13 | xx | Dispose | Manage |
| 113 | 2005-7-11 | ¥ 729.98 | 2005-7-13 | xx | Dispose | Manage |
| 112 | 2005-4-19 | ¥ 668.96 | 2005-4-21 | dotnet | Dispose | Manage |
| 111 | 2005-4-19 | ¥ 49.99 | 2005-4-17 | cs | Dispose | Manage |
| 110 | 2005-4-12 | ¥ 4,559.93 | 2005-4-14 | LiuQingGuo | Dispose | Manage |
| 109 | 2005-4-7 | ¥ 7.97 | 2005-4-8 | LiuQingGuo | Dispose | Manage |
| 108 | 2005-2-24 | ¥ 14.99 | 2003-2-24 | Big Guy | Dispose | Manage |
| 107 | 2000-10-30 | ¥ 39,999.96 | 2000-10-31 | Test Account | Dispose | Manage |
| 106 | 2000-10-30 | ¥ 1,519.97 | 2000-10-30 | Test Account | Dispose | Manage |
| 105 | 2000-10-30 | ¥ 2,999.98 | 2000-10-31 | Test Account | Dispose | Manage |
| 104 | 2000-7-10 | ¥ 2,376.95 | 2000-7-11 | Guest Account | Dispose | Manage |
| 103 | 2000-7-10 | ¥ 1.99 | 2000-7-10 | Test Account | Dispose | Manage |
| 102 | 2000-7-10 | ¥ 729.98 | 2000-7-12 | Test Account | Dispose | Manage |
| 101 | 2000-7-10 | ¥ 629.97 | 2000-7-11 | Test Account | Dispose | Manage |
| 100 | 2000-7-6 | ¥ 919.98 | 2000-7-8 | Guest Account | Dispose | Manage |
| 99 | 2000-7-6 | ¥ 919.98 | 2000-7-7 | Guest Account | Dispose | Manage |

图 9-2 ManageOrders.aspx 页面效果

● 订单信息更新页面 ManagerOrderDetail.aspx

该页面中管理员可以查看到订单详细信息，其中包括订单的主体信息以及详细购买商品条目，管理员可以在该页面中更新订单的送达日期以及订单状态。ManagerOrderDetail.aspx 页面效果如图 9-3 所示。

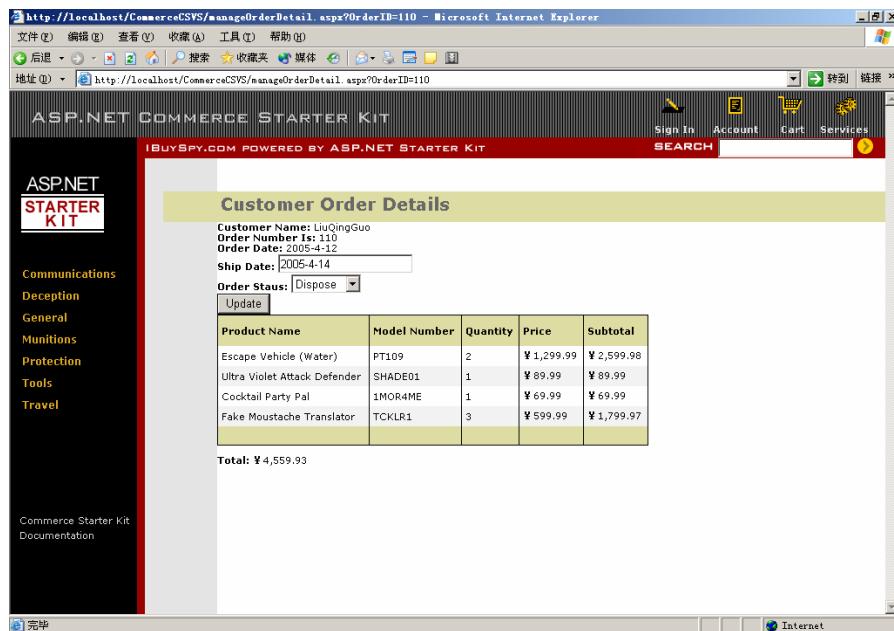


图 9-3 ManagerOrderDetail.aspx 页面效果

注意：为了不过多地更改 ASP.NET Commerce Starter Kit 原有的页面样式，本章中所增加的全部功能都未更改系统中通用的用户控件以及页面样式表，增加的所有页面都继承了 ASP.NET Commerce Starter Kit 原有页面的风格和样式，增加的所有代码都放置于 ASP.NET Commerce Starter Kit 逻辑层原有的几个类中。读者按照本章中所作出的修改对工程中的源文件进行更新，编译之后，可以以直接访问 Url 的形式访问本章中新增的页面。

在需求中出现了订单对象的一个新的属性：订单状态。该属性是为了表示订单当前的状态。根据订单管理流程的需求可以将其值列举为以下几项：处理中、已接受、配送中和已结束。需要对数据库中的订单表 CMRC_Orders 进行补充，添加订单状态的字段 OrderStatus，该字段类型为 tinyint。订单对象的订单状态属性在系统中的表述如表 9-1。

表 9-1 订单状态属性说明

| 状态名称 | 系统中表述 | OrderStatus 字段值 |
|------|------------|-----------------|
| 处理中 | Dispose | 0 |
| 已接受 | Accepted | 1 |
| 配送中 | In Transit | 2 |
| 已结束 | Finished | 3 |

9.1.2 数据层

根据需求分析中的新增功能，需要增加 CMRC_AllOrdersList、CMRC_ViewOrdersDetail 和 CMRC_ManageOrder 三个存储过程。

● CMRC_AllOrdersList

存储过程 CMRC_AllOrdersList 是一个返回系统中所有订单信息的存储过程，该存储过程返回的订单信息中包括订单提交者的 FullName 以及订单的总价。该存储过程的代码清单如下。

```
ALTER Procedure CMRC_AllOrdersList
As
SELECT
    CMRC_Orders.OrderID,
    /*计算该订单的总价*/
    Cast(sum(CMRC_OrderDetails.Quantity*CMRC_OrderDetails.UnitCost) as money) as OrderTotal,
    CMRC_Orders.OrderDate,
    CMRC_Orders.ShipDate,
    CMRC_Orders.OrderStatus,
    /*获取该订单提交者的 FullName*/
    CMRC_Customers.FullName
FROM
    CMRC_Orders
    INNER JOIN CMRC_OrderDetails ON CMRC_Orders.OrderID = CMRC_OrderDetails.OrderID
    INNER JOIN CMRC_Customers ON CMRC_Customers.CustomerID = CMRC_Orders.CustomerID
GROUP BY
    CMRC_Orders.OrderID,
    CMRC_Orders.OrderDate,
    CMRC_Orders.ShipDate,
    CMRC_Customers.FullName,
    CMRC_Orders.OrderStatus
/*逆序排列订单列表*/
ORDER BY CMRC_Orders.OrderID DESC
```

● CMRC_ViewOrdersDetail

存储过程 CMRC_ViewOrdersDetail 是从数据库中返回一笔订单详细信息的存储过程，该存储过程和 CMRC_OrdersDetail 类似，但是 CMRC_OrdersDetail 是返回某个用户所对应的指定订单的信息，而 CMRC_ViewOrdersDetail 则仅根据订单 OrderID 返回该订单的详细信息，并且 CMRC_ViewOrdersDetail 在返回的订单详细信息中增加了订单提交者和订单状态两个字段。存储过程 CMRC_ViewOrdersDetail 的代码清单如下。

```
ALTER Procedure CMRC_ViewOrdersDetail
(
    @OrderID      int,
    /*该订单提交者作为参数输出*/
    @CustomerName varchar(50) OUTPUT,
    @OrderDate    datetime OUTPUT,
    @ShipDate     datetime OUTPUT,
    /*该订单总价作为参数输出*/
    @OrderTotal   money OUTPUT,
    /*该订单状态作为参数输出*/
    @OrderStatus  tinyint OUTPUT
)
AS
/*获取订单基本信息*/
SELECT
    @OrderDate = CMRC_Orders.OrderDate,
    @ShipDate = CMRC_Orders.ShipDate,
    @OrderStatus = CMRC_Orders.OrderStatus,
    @CustomerName = CMRC_Customers.FullName
FROM
```

```
CMRC_Orders, CMRC_Customers
WHERE
    CMRC_Orders.OrderID = @OrderID AND CMRC_Orders.CustomerID = CMRC_Customers.CustomerID

IF @@Rowcount = 1
BEGIN

    /*如果该订单基本信息存在则获取其总价*/
    SELECT
        @OrderTotal = Cast(SUM(CMRC_OrderDetails.Quantity * CMRC_OrderDetails.UnitCost) as money)
    FROM
        CMRC_OrderDetails
    WHERE
        OrderID= @OrderID

    /*如果获取订单的详细信息*/
    SELECT
        CMRC_Products.ProductID,
        CMRC_Products.ModelName,
        CMRC_Products.ModelNumber,
        CMRC_OrderDetails.UnitCost,
        CMRC_OrderDetails.Quantity,
        (CMRC_OrderDetails.Quantity * CMRC_OrderDetails.UnitCost) as ExtendedAmount
    FROM
        CMRC_OrderDetails
    INNER JOIN CMRC_Products ON CMRC_OrderDetails.ProductID = CMRC_Products.ProductID
    WHERE
        OrderID = @OrderID

END
```

● CMRC_ManageOrder

存储过程 CMRC_ManageOrder 为更新订单信息中订单状态和送达日期的存储过程。该存储过程代码清单如下。

```
ALTER Procedure CMRC_ManageOrder
(
    @OrderID int,
    @OrderStatus tinyint,
    @ShipDate datetime
)
AS
UPDATE CMRC_Orders
SET
    OrderStatus = @OrderStatus,
    ShipDate = @ShipDate
WHERE
    OrderID = @OrderID
```

在数据层所对应的修改包括修改订单对象属性、增加返回系统中所有订单信息的方法、增加根据指定 OrderID 返回订单详细信息的方法以及更新订单信息的方法。

● 修改订单对象属性

修改后的订单对象如下，注意其中突出显示部分，添加了订单提交者和订单状态两个属性，其中订单

提交者属性 CustomerName 记录订单提交者的 FullName。

```
public class OrderDetails {
    public DateTime OrderDate;
    public DateTime ShipDate;
    public decimal OrderTotal;
    public DataSet OrderItems;
    public string CustomerName;
    public int OrderStatus;
}
```

● 返回系统中所有订单信息的方法

该方法位于类 ASPNET.StarterKit.Commerce.OrdersDB 下，调用存储过程 CMRC_AllOrdersList 返回一个存储系统中所有的订单数据的 SqlDataReader 对象。其代码清单如下。

```
public SqlDataReader GetAllOrders()
{
    //实例化 SqlConnection 和 SqlCommand 对象
    SqlConnection myConnection = new SqlConnection(ConfigurationSettings.AppSettings["ConnectionString"]);
    SqlCommand myCommand = new SqlCommand("CMRC_AllOrdersList", myConnection);
    //指定 SqlCommand 的执行类型
    myCommand.CommandType = CommandType.StoredProcedure;
    //打开 SqlConnection 对象并执行 SqlCommand 对象返回 SqlDataReader
    myConnection.Open();
    SqlDataReader result = myCommand.ExecuteReader(CommandBehavior.CloseConnection);
    return result;
}
```

● 根据指定 OrderID 返回订单详细信息的方法

该方法位于类 ASPNET.StarterKit.Commerce.OrdersDB 下，调用存储过程 CMRC_ViewOrdersDetail 返回参数表中指定的 OrderID 所对应订单的详细信息。其代码清单如下。

```
public OrderDetails ViewOrderDetails(int orderID)
{
    //实例化 SqlConnection 和 SqlCommand 对象
    SqlConnection myConnection = new SqlConnection(ConfigurationSettings.AppSettings["ConnectionString"]);
    SqlDataAdapter myCommand = new SqlDataAdapter("CMRC_ViewOrdersDetail", myConnection);

    //设置 SqlDataAdapter 的查询类型
    myCommand.SelectCommand.CommandType = CommandType.StoredProcedure;

    //向 SqlDataAdapter 对象添加查询参数 parameterOrderID
    SqlParameter parameterOrderID = new SqlParameter("@OrderID", SqlDbType.Int, 4);
    parameterOrderID.Value = orderID;
    myCommand.SelectCommand.Parameters.Add(parameterOrderID);
    //向 SqlDataAdapter 对象添加输出参数 parameterCustomerName
    SqlParameter parameterCustomerName = new SqlParameter("@CustomerName", SqlDbType.VarChar, 50);
    parameterCustomerName.Direction = ParameterDirection.Output;
    myCommand.SelectCommand.Parameters.Add(parameterCustomerName);
    //向 SqlDataAdapter 对象添加输出参数 parameterOrderDate
    SqlParameter parameterOrderDate = new SqlParameter("@OrderDate", SqlDbType.DateTime, 8);
```

```

        parameterOrderDate.Direction = ParameterDirection.Output;
        myCommand.SelectCommand.Parameters.Add(parameterOrderDate);
        //向 SqlDataAdapter 对象添加输出参数 parameterShipDate
        SqlParameter parameterShipDate = new SqlParameter("@ShipDate", SqlDbType.DateTime, 8);
        parameterShipDate.Direction = ParameterDirection.Output;
        myCommand.SelectCommand.Parameters.Add(parameterShipDate);
        //向 SqlDataAdapter 对象添加输出参数 parameterOrderTotal
        SqlParameter parameterOrderTotal = new SqlParameter("@OrderTotal", SqlDbType.Money, 8);
        parameterOrderTotal.Direction = ParameterDirection.Output;
        myCommand.SelectCommand.Parameters.Add(parameterOrderTotal);
        //向 SqlDataAdapter 对象添加输出参数 parameterOrderStatus
        SqlParameter parameterOrderStatus = new SqlParameter("@OrderStatus", SqlDbType.TinyInt);
        parameterOrderStatus.Direction = ParameterDirection.Output;
        myCommand.SelectCommand.Parameters.Add(parameterOrderStatus);

        //执行 SqlDataAdapter
        DataSet myDataSet = new DataSet();
        myCommand.Fill(myDataSet, "OrderItems");
        //如果订单记录存在则建立一个 OrderDetails 对象并返回
        if (parameterShipDate.Value != DBNull.Value)
        {
            //实例化一个 OrderDetails 对象
            OrderDetails CustomerOrderDetails = new OrderDetails();
            //为 OrderDetails 对象的属性赋值
            CustomerOrderDetails.CustomerName = (string)parameterCustomerName.Value;
            CustomerOrderDetails.OrderDate = (DateTime)parameterOrderDate.Value;
            CustomerOrderDetails.ShipDate = (DateTime)parameterShipDate.Value;
            CustomerOrderDetails.OrderTotal = (decimal)parameterOrderTotal.Value;
            CustomerOrderDetails.OrderStatus = Convert.ToInt32(parameterOrderStatus.Value.ToString());
            CustomerOrderDetails.OrderItems = myDataSet;

            //返回 OrderDetails 对象
            return CustomerOrderDetails;
        }
        else
            return null;
    }
}

```

9.1.3 逻辑层

订单管理模块新增的两个页面的后台编码类 ManageOrders.aspx.cs 和 ManageOrderDetail.aspx.cs。

● ManageOrders.aspx.cs

在该后台编码类中，对前台页面的 DataGrid 控件进行绑定数据源等操作，并在 DataGrid 的 DataItemBound 事件中对订单状态字段的显示进行定义。下面是该后台编码类的部分关键代码。

```

//页面载入事件
private void Page_Load(object sender, System.EventArgs e)
{
    //实例化一个 ASPNET.StarterKit.Commerce.OrdersDB 对象
    ASPNET.StarterKit.Commerce.OrdersDB ObjectOrderList = new ASPNET.StarterKit.Commerce.OrdersDB();
}

```

```
//调用 OrdersDB 对象的 GetAllOrders 方法为 DataGrid 控件 OrderList 指定数据源
OrderList.DataSource = ObjectOrderList.GetAllOrders();
//绑定 OrderList 控件
OrderList.DataBind();
}

//DataGrid 控件 OrderList 的 ItemDataBound 事件
private void OrderList _ItemDataBound(object sender, System.Web.UI.WebControls.DataGridItemEventArgs e)
{
    //判断当前行是否为数据行
    if(e.Item.ItemType == ListItemType.Item || e.Item.ItemType == ListItemType.AlternatingItem)
    {
        //获得 DataGrid 中显示订单状态的 Label 控件
        Label lblOrderStatus = (Label)e.Item.Cells[4].FindControl("lblOrderStatus");
        //从数据源中获得记录订单状态的 orderStatus 字段
        int orderStatus = Convert.ToInt32(DataBinder.Eval(e.Item.DataItem,"orderstatus"));
        //利用 Switch 结构判断并显示订单状态字段
        switch(orderStatus)
        {
            case 0:
                lblOrderStatus.Text = "Dispose";
                break;
            case 1:
                lblOrderStatus.Text = "Accepted";
                break;
            case 2:
                lblOrderStatus.Text = "In Transit";
                break;
            case 3:
                lblOrderStatus.Text = "Finished";
                break;
            default:
                lblOrderStatus.Text = "Ambiguity";
                break;
        }
    }
}
}
```

● ManageOrderDetail.aspx.cs

该后台编码类中绑定前台页面显示订单详细信息控件，其关键部分代码如下。

```
//定义页面中显示订单详细信息的控件
protected System.Web.UI.WebControls.Label MyError;
protected System.Web.UI.WebControls.Label lblOrderNumber;
protected System.Web.UI.WebControls.Label lblOrderDate;
protected System.Web.UI.WebControls.DataGrid GridControl1;
protected System.Web.UI.WebControls.Label lblTotal;
protected System.Web.UI.WebControls.Label lblCustomerName;
protected System.Web.UI.WebControls.TextBox txbShipDate;
protected System.Web.UI.WebControls.DropDownList ddlOrderStatus;
protected System.Web.UI.WebControls.Button btnUpdate;
protected void Page_Load(object sender, System.EventArgs e)
{
```

```
//从Url中获得所显示订单的OrderID
int OrderID = Int32.Parse(Request.Params["OrderID"]);
//实例化 ASP.NET.StarterKit.Commerce.OrdersDB 对象
ASPNET.StarterKit.Commerce.OrdersDB orderObject = new ASPNET.StarterKit.Commerce.OrdersDB();
//调用 OrdersDB 对象的 ViewOrderDetails 方法获得记录订单详细信息的 OrderDetails 对象
ASPNET.StarterKit.Commerce.OrderDetails customerOrderDetails =
orderObject.ViewOrderDetails(OrderID);
//判断记录订单详细信息的 OrderDetails 对象是否为空
if (customerOrderDetails != null)
{
    //绑定显示订单详细购物信息的 DataGrid 控件
    GridControl1.DataSource = customerOrderDetails.OrderItems;
    GridControl1.DataBind();
    //绑定显示订单主体信息的控件
    lblTotal.Text = String.Format("{0:c}", customerOrderDetails.OrderTotal);
    lblOrderNumber.Text = OrderID.ToString();
    lblOrderDate.Text = customerOrderDetails.OrderDate.ToShortDateString();
    txbShipDate.Text = customerOrderDetails.ShipDate.ToShortDateString();
    lblCustomerName.Text = customerOrderDetails.CustomerName.ToString();
    //设置显示订单状态的 DropDownList 控件的选中项
    ddlOrderStatus.Items.FindByValue(customerOrderDetails.OrderStatus.ToString()).Selected = true;

}
else
{
    MyError.Text = "Order not found!";
}
}

//更新订单信息按钮 btnUpdate 的单击事件
private void btnUpdate_Click(object sender, System.EventArgs e)
{
    //实例化 ASP.NET.StarterKit.Commerce.OrdersDB 对象
    ASPNET.StarterKit.Commerce.OrdersDB orderObject = new ASPNET.StarterKit.Commerce.OrdersDB();
    //调用 orderObject 对象的 ManageOrder 方法更新订单信息

    orderObject.ManageOrder(int.Parse(lblOrderNumber.Text), int.Parse(Request.Params["ddlOrderStatus"].ToString()),
    DateTime.Parse(txbShipDate.Text));
    //在更新订单信息结束之后跳转至订单管理列表页面
    Response.Redirect("manageorders.aspx");
}
```

9.1.4 表示层

订单管理部分的表示层为 ManageOrders.aspx 和 ManageOrderDetail.aspx 两个页面，两个页面都是以 DataGrid 控件为主。其中 ManageOrders.aspx 中以显示新近订单的 DataGrid 为主，而 ManageOrderDetail 则以显示某笔订单中详细订购信息的 DataGrid 为主。两个页面的代码清单如下。

ManageOrders.aspx

```
<%@ Page language="c#" Codebehind="ManageOrders.aspx.cs" AutoEventWireup="false"%>
```

```
Inherits="ASPNET.StarterKit.Commerce.ManageOrders" %>
<!-- 引入用户控件 Menu -->
<%@ Register TagPrefix="ASPNETCommerce" TagName="Menu" Src="_Menu.ascx" %>
<!-- 引入用户控件 Header -->
<%@ Register TagPrefix="ASPNETCommerce" TagName="Header" Src="_Header.ascx" %>
<HTML>
<HEAD>
<link rel="stylesheet" type="text/css" href="ASPNETCommerce.css">
</HEAD>
<body background="images/sitebkgrd.gif" leftmargin="0" topmargin="0" rightmargin="0" bottommargin="0"
marginheight="0" marginwidth="0">
<table cellspacing="0" cellpadding="0" width="100%" border="0">
<tr>
<td colspan="2">
<ASPNETCommerce:Header ID="Header1" runat="server" />
</td>
</tr>
<tr>
<td valign="top">
<ASPNETCommerce:Menu id="Menu1" runat="server" />

</td>
<td align="left" valign="top" width="100%" nowrap>
<table height="100%" align="left" cellspacing="0" cellpadding="0" width="100%" border="0">
<tr valign="top">
<td nowrap>
<br>
<form runat="server" ID="Form1">

<table cellspacing="0" cellpadding="0" width="100%" border="0">
<tr>
<td class="ContentHead">
Last Order List
<br>
</td>
</tr>
</table>
<br>

<table height="100%" cellspacing="0" cellpadding="0" width="700" border="0">
<tr valign="top">
<td width="100%">
<!-- 定义 DataGrid 控件 -->
<asp:DataGrid id="OrderList" width="90%" BorderColor="black" GridLines="Vertical" cellpadding="4"
cellspacing="0" Font-Name="Verdana" Font-Size="8pt" ShowFooter="true" HeaderStyle-CssClass=
"CartListHead"
FooterStyle-CssClass="cartlistfooter" ItemStyle-CssClass="CartListItem"
AlternatingItemStyle-CssClass="CartListItemAlt"
AutoGenerateColumns="false" runat="server">
<!-- 定义 DataGrid 控件的列集合 -->
```

```
<Columns>
    <!-- 定义链接列显示 OrderID 字段，并将链接的 Url 指向显示订单详细信息的页面 -->
    <asp:HyperLinkColumn HeaderText="Order ID" DataTextField="OrderID" DataNavigateUrlField="OrderID" DataNavigateUrlFormatString="vieworderdetails.aspx?OrderID={0}"></asp:HyperLinkColumn>
    <!-- 定义普通列显示 OrderDate 字段 -->
    <asp:BoundColumn HeaderText="Order Date" DataField="OrderDate" DataFormatString="{0:d}" />
    <!-- 定义普通列显示 OrderTotal 字段 -->
    <asp:BoundColumn HeaderText="Order Total" DataField="OrderTotal" DataFormatString="{0:c}" />
    <!-- 定义普通列显示 ShipDate 字段 -->
    <asp:BoundColumn HeaderText="Ship Date" DataField="ShipDate" DataFormatString="{0:d}" />
    <!-- 定义普通列显示 CustomerName 字段 -->
    <asp:BoundColumn HeaderText="Customer Name" DataField="FullName" DataFormatString="{0:d}" />
    <!-- 定义模板列显示 OrderStatus 字段 -->
    <asp:TemplateColumn HeaderText="Order Status">
        <!-- 定义模板列的 ItemTemplate -->
        <ItemTemplate>
            <!-- 定义显示订单状态的 Label 控件 -->
            <asp:Label ID="lblOrderStatus" Runat="server"></asp:Label>
        </ItemTemplate>
    </asp:TemplateColumn>
    <!-- 定义链接列显示编辑订单的链接，并将链接的 Url 指向编辑订单的页面 -->
    <asp:HyperLinkColumn HeaderText="Manage" Text="Manage" DataNavigateUrlField="OrderID" DataNavigateUrlFormatString="manageOrderDetail.aspx?OrderID={0}" />
    <!-- 结束 DataGrid 列集合定义 -->
</Columns>
<!-- 结束 DataGrid 定义 -->
</asp:DataGrid>
</td>
</tr>
</table>
</form>
</td>
</tr>
</table>
</td>
</tr>
</table>
</body>
</HTML>
```

在 DataGrid 控件的定义中，使用了一列模板列，并放置了一个 Label 控件在该模板列的 ItemTemplet 中。这样做的目的是为了显示订单状态字段，因此从数据库中取出的订单状态字段为 tinyint 型的数字，而在前面的需求分析中，已经定义了数据型的订单状态所对应的字符型订单状态。因此在 DataGrid 的列集合中要做一下转换。该转换操作实现的方法是在后台编码类中，捕捉 DataGrid 控件的 ItemDataBound 事件，在该事件中通过(Label)e.Item.Cells[4].FindControl("lblOrderStatus")获得 DataGrid 控件列集合中第五列 ItemTemplet 中定义 Label 控件，并对其进行操作。

注意：在 DataGrid 控件的列集合中的第一列定义了显示订单详细信息的链接，该链接指向页面 vieworderdetails.aspx 是在随后章节中专门为显示订单详细信息所设计的页面。

● ManageOrderDetail.aspx

```
<%@ Page language="c#" Codebehind="manageOrderDetail.aspx.cs" AutoEventWireup="false" Inherits="ASPNET.StarterKit.Commerce.manageOrderDetail" %>
<!-- 引入用户控件 Header -->
<%@ Register TagPrefix="ASPNETCommerce" TagName="Header" Src="_Header.ascx" %>
<!-- 引入用户控件 Menu -->
<%@ Register TagPrefix="ASPNETCommerce" TagName="Menu" Src="_Menu.ascx" %>
<HTML>
<HEAD>
<link rel="stylesheet" type="text/css" href="ASPNETCommerce.css">
</HEAD>
<body background="images/sitebkgrd.gif" leftmargin="0" topmargin="0" rightmargin="0" bottommargin="0" marginheight="0" marginwidth="0">
<form runat="server">
<table cellspacing="0" cellpadding="0" width="100%" border="0">
<tr>
<td colspan="2">
<ASPNETCommerce:Header ID="Header1" runat="server" />
</td>
</tr>
<tr>
<td valign="top">
<ASPNETCommerce:Menu id="Menu1" runat="server" />

</td>
<td align="left" valign="top" width="100%" nowrap>
<table height="100%" align="left" cellspacing="0" cellpadding="0" width="100%" border="0">
<tr valign="top">
<td nowrap>
<br>

<table cellspacing="0" cellpadding="0" width="100%" border="0">
<tr>
<td class="ContentHead">
Customer Order
Details
<br>
</td>
</tr>
</table>

<!-- 定义显示错误信息的 Label 控件 -->
<asp:Label id="MyError" CssClass="ErrorText" EnableViewState="false" runat="Server" />
<table height="100%" cellspacing="0" cellpadding="0" width="550" border="0">
<tr valign="top">
<td width="100%" class="Normal">
<b>Customer Name: </b>
<!-- 定义显示用户名字段的 Label 控件 -->
<asp:Label ID="lblCustomerName" EnableViewState="false" runat="server" />
<br>
<b>Order Number Is: </b>
```

```
<!-- 定义显示订单号字段的 Label 控件 -->
<asp:Label ID="lblOrderNumber" EnableViewState="false" runat="server" />
<br>
<b>Order Date: </b>
<!-- 定义显示订单日期字段的 Label 控件 -->
<asp:Label ID="lblOrderDate" EnableViewState="false" runat="server" />
<br>
<b>Ship Date: </b>
<!-- 定义显示送达日期字段的 Text 控件可供更改订单送达日期 -->
<asp:TextBox ID="txbShipDate" runat="server" />
<BR>
<b>Order Status:</b>
<!-- 定义显示订单状态的 DropDownList 控件可供更改订单状态 -->
<asp:DropDownList id="ddlOrderStatus" runat="server">
    <asp:ListItem Value="0">Dispose</asp:ListItem>
    <asp:ListItem Value="1">Accepted</asp:ListItem>
    <asp:ListItem Value="2">In Transit</asp:ListItem>
    <asp:ListItem Value="3">Finished</asp:ListItem>
</asp:DropDownList>
<br>
<!-- 定义更新按钮控件 -->
<asp:Button id="btnUpdate" runat="server" Text="Update"></asp:Button>
<br>
<!-- 定义显示订单购物信息的 DataGrid 控件 -->
<asp:DataGrid id="GridControl1" width="90%" BorderColor="black" GridLines="Vertical" cellpadding="4"
    cellspacing="0" Font-Name="Verdana" Font-Size="8pt" ShowFooter="true" HeaderStyle-CssClass=
    "CartListHead"
    FooterStyle-CssClass="cartlistfooter" ItemStyle-CssClass="CartListItem"
    AlternatingItemStyle-CssClass= "CartListItemsAlt"
    AutoGenerateColumns="false" runat="server">
    <!-- 定义 DataGrid 控件的列集合 -->
    <Columns>
        <!-- 定义显示 ModelName 字段的列 -->
        <asp:BoundColumn HeaderText="Product Name" DataField="ModelName" />
        <!-- 定义显示 ModelNumber 字段的列 -->
        <asp:BoundColumn HeaderText="Model Number" DataField="ModelNumber" />
        <!-- 定义显示 Quantity 字段的列 -->
        <asp:BoundColumn HeaderText="Quantity" DataField="Quantity" />
        <!-- 定义显示 UnitCost 字段的列 -->
        <asp:BoundColumn HeaderText="Price" DataField="UnitCost" DataFormatString="{0:c}" />
        <!-- 定义显示 ExtendedAmount 字段的列 -->
        <asp:BoundColumn HeaderText="Subtotal" DataField="ExtendedAmount" DataFormatString="{0:c}" />
    <!-- 结束 DataGrid 控件的列集合定义 -->
    </Columns>
    <!-- 结束 DataGrid 控件的定义 -->
</asp:DataGrid>
<br>
<b>Total: </b>
<!-- 定义显示订单总价字段的 Label 控件 -->
<asp:Label ID="lblTotal" EnableViewState="false" runat="server" /><BR>
```

```
</td>
</tr>
</table>
</td>
</tr>
</table>
</td>
</tr>
</table>
</form>
</body>
</HTML>
```

在 ManageOrderDetail.aspx 页面中，订单状态和订单送达日期是可以被更改的，用户更改订单状态或订单送达日期之后单击更新按钮，就可以完成对订单信息的更新。其中订单状态使用了一个 DropDownList 控件，在页面的后台编码类中使用 dd1OrderStatus.Items.FindByValue (customerOrderDetails.OrderStatus.ToString ()) 找到应该被选中的订单状态 ListItem 并将其选中。

9.2 用户数据

9.2.1 需求分析

在电子商务应用中最主要的一个主体就是使用电子商务应用订购商品的用户，没有用户使用的电子商务应用几乎是毫无意义的。针对用户这个对象，在电子商务的数据统计和数据管理中要有相应的模块。

首先是用户整体数据的统计，例如用户的订单数量、访问次数、用户的总体消费额等等。这些数据对于电子商务的提供商来说是必须了解的。

其次是每一个用户的详细数据统计，例如用户每笔订单的详细情况等等。虽然有些功能和订单管理中的功能重复或是类似，但是对于一个完整的电子商务应用来讲，根据用户索引订单的功能是必不可少的。

用户数据部分和订单管理部分是截然不同的两个模块，其中订单管理部分是以订单管理流程为核心，而用户数据部分则是以统计用户数据为核心。其设计结构如图 9-4 所示。

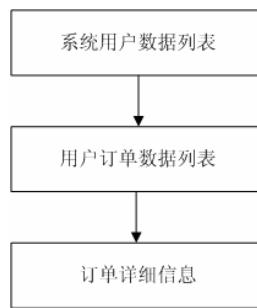


图 9-4 用户数据部分设计结构

根据上面的需求分析，将用户数据部分分为 3 个页面。

● 系统用户数据列表 CustomerList.aspx

系统用户数据列表页面中以列表的形式展现系统中所有用户，其中包括用户的 ID、FullName、Email、

订单总额、购买商品数据等字段，并提供链接到用户订单数据列表页面的链接。CustomerList.aspx 页面效果如图 9-5 所示。

| Full Name | Email Address | Total Order | Product Count | |
|---------------|-------------------|-------------|---------------|--------------------------------------|
| Big Guy | bg@aj.net | 14.9900 | 1 | View Customer Detail |
| cs | cs@cs.net | 49.9900 | 1 | View Customer Detail |
| Test Account | d | 45883.8400 | 10 | View Customer Detail |
| dotnet | dotnet@dotnet.com | 668.9600 | 4 | View Customer Detail |
| Guest Account | guest | 4231.9000 | 8 | View Customer Detail |
| LiuQingGuo | wf@wf.com | 4567.9000 | 6 | View Customer Detail |
| xx | xx@xx.com | 1109.9400 | 6 | View Customer Detail |

图 9-5 CustomerList.aspx 页面效果

● 用户订单数据列表 CustomerDetail.aspx

用户订单数据列表页面中以列表的形式展现了单个用户的所有订单信息，这和系统中的原有察看订单历史页面类似，并提供链接到订单详细信息页面的链接。CustomerDetail.aspx 页面效果如图 9-6 所示。

● 订单详细信息 ViewOrderDetail.aspx

订单详细信息页面列出了每一笔订单的详细信息其中包括订单的主体信息以及详细购物信息。ViewOrderDetail.aspx 页面效果如图 9-7 所示。

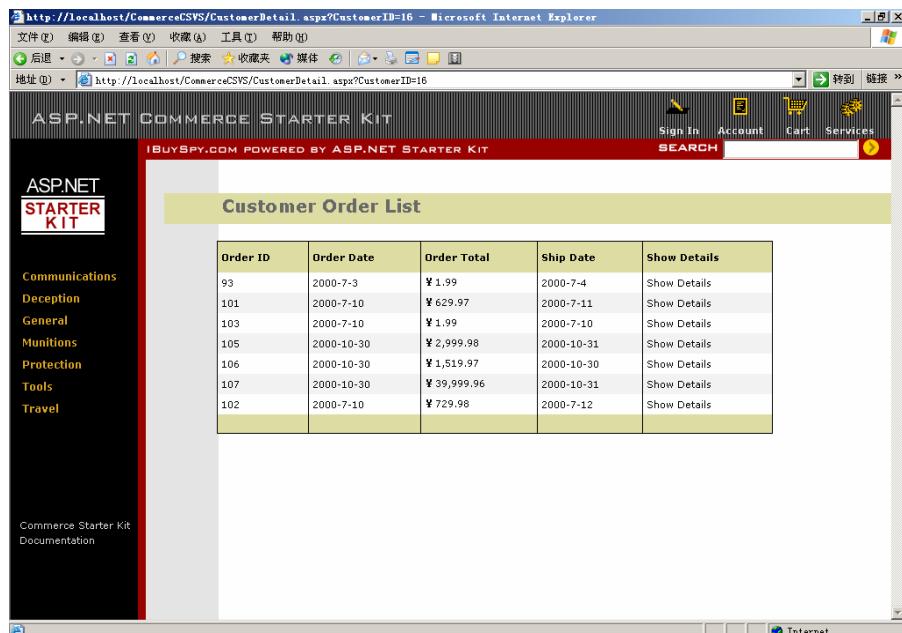


图 9-6 CustomerDetail.aspx 页面效果

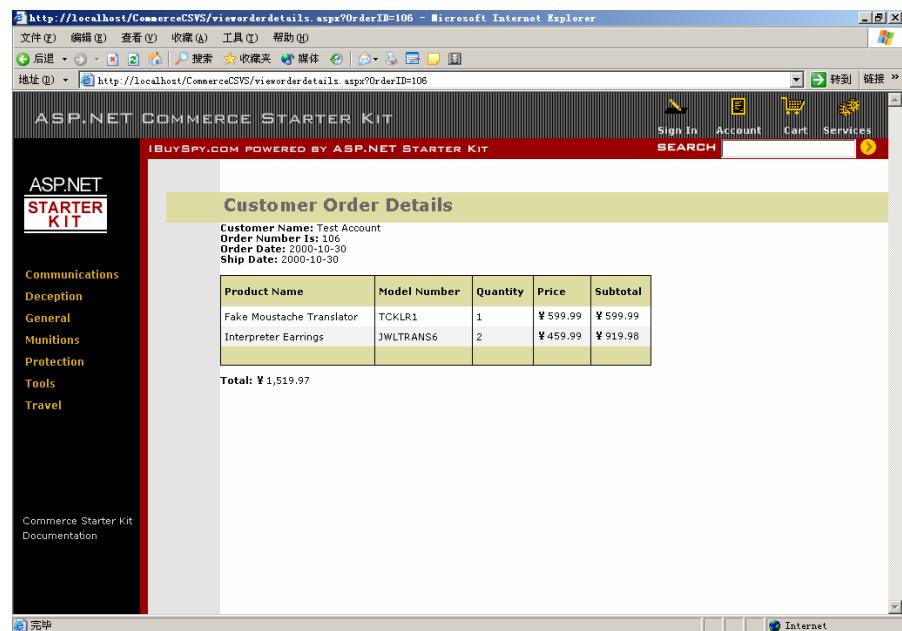


图 9-7 ViewOrderDetail.aspx 页面效果

9.2.2 数据层

在数据层方面，用户数据模块新增了一个存储过程 CMRC_CustomerList。

● CMRC_CustomerList

存储过程 CMRC_CustomerList 是返回系统中用户信息列表的存储过程，返回列表中的用户信息包括用户的 ID、FullName、EmailAddress、订单总额以及购买商品数量几个字段。其代码清单如下。

```
ALTER Procedure CMRC_CustomerList
```

```

AS
SELECT CMRC_Customers.CustomerID, CMRC_Customers.FullName, CMRC_Customers.EmailAddress,
       SUM(CMRC_OrderDetails.Quantity * CMRC_OrderDetails.UnitCost) AS TotalOrder,
       COUNT(CMRC_OrderDetails.ProductID) AS ProductCount
FROM CMRC_Customers INNER JOIN
     CMRC_Orders ON
     CMRC_Customers.CustomerID = CMRC_Orders.CustomerID INNER JOIN
     CMRC_OrderDetails ON CMRC_Orders.OrderID = CMRC_OrderDetails.OrderID
GROUP BY CMRC_Customers.CustomerID, CMRC_Customers.FullName, CMRC_Customers.EmailAddress

```

在ASP.NET Commerce Starter Kit数据层的CustomerDB类中增加一个列出系统中用户信息的方法，该方法返回一个SqlDataReader对象，该对象为包含系统用户列表的数据集。该方法的代码如下。

```

public SqlDataReader CustomerList()
{
    //实例化SqlConnection对象和SqlCommand对象
    SqlConnection myConnection = new SqlConnection(ConfigurationSettings.AppSettings["ConnectionString"]);
    SqlCommand myCommand = new SqlCommand("CMRC_CustomerList", myConnection);
    //制定SqlCommand对象的 CommandType 属性
    myCommand.CommandType = CommandType.StoredProcedure;
    //打开SqlConnection对象并获得记录集
    myConnection.Open();
    SqlDataReader result = myCommand.ExecuteReader(CommandBehavior.CloseConnection);
    //返回记录集 SqlDataReader 对象
    return result;
}

```

9.2.3 逻辑层

在需求分析中已经定义了用户数据模块的3个页面，3个页面的后台编码类分别为CustomerList.aspx.cs、CustomerDetail.aspx.cs、ViewOrderDetail.aspx.cs。

● CustomerList.aspx.cs

页面CustomerList.aspx的主体为显示用户列表的DataGrid控件，因此页面的后台编码类并不复杂，主要是在页面载入方法Page_Load中绑定DataGrid控件，Page_Load方法代码如下。

```

Private Void Page_load(Object Sender, System.EventArgs E)
{
    //实例化一个Aspnet.Starterkit.Commerce.Customersdb对象
    Aspnet.Starterkit.Commerce.Customersdb Objectcustorm = New Customersdb();
    //调用Customersdb对象的Customerlist获得DataGrid控件的数据源
    Customergird.Datasource = Objectcustorm.Customerlist();
    //绑定DataGrid控件
    Customergird.DataBind();
}

```

● CustomerDetail.aspx.cs

页面CustomerDetail.aspx的主体为显示用户订单列表的DataGrid控件，其实现逻辑和用户订单历史页面类似，只不过用户订单历史页面是从User对象中取得用户的CustomerID，而CustomerDetail.aspx是从Url中取得用户的CustomerID。其页面载入方法Page_Load代码如下。

```

private void Page_Load(object sender, System.EventArgs e)
{
}

```

```
//从Url中获得用户的customerID  
String customerID = Request.QueryString["CustomerID"].ToString() ;  
//实例化一个ASPNET.StarterKit.Commerce.OrdersDB对象  
ASPNET.StarterKit.Commerce.OrdersDB ObjectOrderList = new ASPNET.StarterKit.Commerce.OrdersDB();  
//调用OrdersDB对象的GetCustomerOrders方法获得DataGrid控件的数据源  
OrderList.DataSource = ObjectOrderList.GetCustomerOrders(customerID);  
//绑定DataGrid控件  
OrderList.DataBind();  
}  
● ViewOrderDetail.aspx.cs
```

订单详细信息页面 ViewOrderDetail.aspx 是显示包括订单主体信息以及详细购物条目在内的订单信息的页面，该页面和系统中原有的 OrderDetail.aspx 类似，但是 OrderDetail.aspx 是根据当前访问者 CustomerID 和订单 OrderID 确定所要查看的订单，但是 ViewOrderDetail.aspx 是只根据 Url 中的 OrderID 确定所要显示的订单信息。ViewOrderDetail.aspx.cs 的关键代码如下。

```
//定义详细订单信息的页面控件  
protected System.Web.UI.WebControls.Label MyError;  
protected System.Web.UI.WebControls.Label lblOrderNumber;  
protected System.Web.UI.WebControls.Label lblOrderDate;  
protected System.Web.UI.WebControls.Label lblShipDate;  
protected System.Web.UI.WebControls.DataGrid GridControl1;  
protected System.Web.UI.WebControls.Label lblTotal;  
protected System.Web.UI.WebControls.Label lblCustomerName;  
protected System.Web.UI.HtmlControls.HtmlTable detailsTable;  
//页面载入方法  
private void Page_Load(object sender, System.EventArgs e)  
{  
    //从Url中获得OrderID  
    int OrderID = Int32.Parse(Request.Params["OrderID"]);  
    //实例化一个ASPNET.StarterKit.Commerce.OrdersDB对象  
    ASPNET.StarterKit.Commerce.OrdersDB orderObject = new ASPNET.StarterKit.Commerce.OrdersDB();  
    //实例化一个ASPNET.StarterKit.Commerce.OrderDetails对象存储订单信息  
    ASPNET.StarterKit.Commerce.OrderDetails customerOrderDetails = orderObject.ViewOrderDetails(OrderID);  
  
    if (customerOrderDetails != null)  
    {  
        //指定显示订单详细购物条目的DataGrid控件的数据源并绑定  
        GridControl1.DataSource = customerOrderDetails.OrderItems;  
        GridControl1.DataBind();  
  
        //绑定显示订单信息的页面控件  
        lblTotal.Text = String.Format("{0:c}", customerOrderDetails.OrderTotal);  
        lblOrderNumber.Text = OrderID.ToString();  
        lblOrderDate.Text = customerOrderDetails.OrderDate.ToShortDateString();  
        lblShipDate.Text = customerOrderDetails.ShipDate.ToShortDateString();  
        lblCustomerName.Text = customerOrderDetails.CustomerName.ToString();  
    }  
    else  
    {  
        //如果数据库中没有要查看的订单信息则显示错误信息  
    }  
}
```

```
        MyError.Text = "Order not found!";
        detailsTable.Visible = false;
    }
}
```

9.2.4 表示层

CustomerList.aspx

```
<!-- 引入用户控件 Menu -->
<%@ Register TagPrefix="ASNETCommerce" TagName="Menu" Src="_Menu. ascx" %>
<!-- 引入用户控件 Header -->
<%@ Register TagPrefix="ASNETCommerce" TagName="Header" Src="_Header. ascx" %>
<%@ Page language="c#" Codebehind="CustomerList.aspx.cs" AutoEventWireup="false" Inherits="ASPNET.
StarterKit.Commerce.CustomerList" %>
<HTML>
<HEAD>
<LINK href="ASNETCommerce.css" type="text/css" rel="stylesheet">
</HEAD>
<body bottomMargin="0" leftMargin="0" background="images/sitebkgrd.gif" topMargin="0"
rightMargin="0" marginwidth="0" marginheight="0">
<table cellSpacing="0" cellPadding="0" width="100%" border="0">
<tr><td colSpan="2">
<ASNETCOMMERCE:HEADER id="Header1" runat="server"></ASNETCOMMERCE:HEADER></td>
</tr><tr>
<td vAlign="top">
<ASNETCOMMERCE:MENU id="Menu1" runat="server"></ASNETCOMMERCE:MENU><IMG height="1" src="images/1x1.gif"
width="145"></td>
<td vAlign="top" noWrap align="left" width="100%">
<table height="100%" cellSpacing="0" cellPadding="0" width="100%" align="left" border="0">
<tr vAlign="top">
<td noWrap><br>
<form id="Form1" runat="server">

<table cellSpacing="0" cellPadding="0" width="100%" border="0">
<tr>
<td class="ContentHead">Customer List<br>
</td>
</tr>
</table>

<br>

<table height="294" cellSpacing="0" cellPadding="0" width="704" border="0">
<tr vAlign="top">
<td width="100%">
<!-- 定义显示用户列表的 DataGrid 控件 -->
<asp:datagrid id="CustomerGrid" runat="server" AlternatingItemStyle-CssClass="CartListItemAlt"
ItemStyle-CssClass="CartListItemIcon" FooterStyle-CssClass="cartlistfooter"
HeaderStyle-CssClass="CartListHead"
ShowFooter="true" Font-Size="8pt" Font-Name="Verdana" cellspacing="0" cellpadding="4" GridLines="Vertical">
```

```

    BorderColor="black" width="90%" AutoGenerateColumns="False">
    <!-- 定义 DataGrid 控件的列集合 -->
    <Columns>
        <!-- 定义列显示 FullName 字段 -->
        <asp:BoundColumn HeaderText="Full Name" DataField="FullName"></asp:BoundColumn>
        <!-- 定义列显示 EmailAddress 字段 -->
        <asp:BoundColumn HeaderText="Email Address" DataField="EmailAddress"></asp:BoundColumn>
        <!-- 定义列显示 TotalOrder 字段 -->
        <asp:BoundColumn HeaderText="Total Order" DataField="TotalOrder" ItemStyle-HorizontalAlign="Right"> /asp:BoundColumn>
        <!-- 定义列显示 ProductCount 字段 -->
        <asp:BoundColumn HeaderText="Product Count" DataField="ProductCount" ItemStyle-HorizontalAlign="Right"></asp:BoundColumn>
        <!-- 定义链接列指向查看用户订单信息页面 -->
        <asp:HyperLinkColumn Text="View Customer Detail" DataNavigateUrlField="CustomerID" DataNavigateUrlFormatString="CustomerDetail.aspx?CustomerID={0}" ItemStyle-HorizontalAlign="Center"></asp:HyperLinkColumn>
    <!-- 结束 DataGrid 控件的列定义 -->
    </Columns>
    <!-- 结束 DataGrid 控件定义 -->
</asp:datagrid></td>
</tr>
</table>
</form>
</td>
</tr>
</table>
</td>
</tr>
</table>
</body>
</HTML>

```

CustomerList.aspx 页面的主体为显示用户列表的 DataGrid 控件，该 DataGrid 控件中列集合的最后一列是链接列，该列的链接指向察看用户订单列表的页面 CustomerDetail.aspx。

CustomerDetail.aspx

```

<%@ Page language="c#" Codebehind="CustomerDetail.aspx.cs" AutoEventWireup="false" Inherits="ASPNET.
StarterKit.Commerce.CustomerDetail" %>
    <!-- 引入用户控件 Header
    <%@ Register TagPrefix="ASPNETCommerce" TagName="Header" Src="_Header.ascx" %>
    <!-- 引入用户控件 Menu
    <%@ Register TagPrefix="ASPNETCommerce" TagName="Menu" Src="_Menu.ascx" %>
<HTML>
<HEAD>
<LINK href="ASPNETCommerce.css" type="text/css" rel="stylesheet">
</HEAD>
<body bottomMargin="0" leftMargin="0" background="images/sitebkgrd.gif" topMargin="0"
rightMargin="0" marginwidth="0" marginheight="0">
<table cellSpacing="0" cellPadding="0" width="100%" border="0">
<tr><td colSpan="2">
<ASPNETCOMMERCE:HEADER id="Header1" runat="server"></ASPNETCOMMERCE:HEADER></td>

```

```
</tr>
<tr><td vAlign="top">
<ASPNETCOMMERCE:MENU id="Menu1" runat="server"></ASPNETCOMMERCE:MENU><IMG height="1" src="images/1x1.gif" width="145">
</td>
<td vAlign="top" nowrap align="left" width="100%">
<table height="100%" cellSpacing="0" cellPadding="0" width="100%" align="left" border="0">
<tr vAlign="top">
<td nowrap><br>
<form id="Form1" runat="server">

<table cellSpacing="0" cellPadding="0" width="100%" border="0">
<tr>
<td class="ContentHead">Customer
Order List
<br>
</td>
</tr>
</table>

<br>

<table height="294" cellSpacing="0" cellPadding="0" width="704" border="0">
<tr vAlign="top">
<td width="100%">
<!-- 定义显示用户订单列表的 DataGrid 控件 --&gt;
&lt;asp:DataGrid id="OrderList" width="90%" BorderColor="black" GridLines="Vertical" cellpadding="4"
cellspacing="0" Font-Name="Verdana" Font-Size="8pt" ShowFooter="true" HeaderStyle-CssClass="CartListHead"
FooterStyle-CssClass="cartlistfooter" ItemStyle-CssClass="CartListItem"
AlternatingItemStyle-CssClass= "CartListItemAlt"
AutoGenerateColumns="false" runat="server"&gt;
<!-- 定义 DataGrid 控件的列集合
&lt;Columns&gt;
    <!-- 定义列显示 OrderID 字段 --&gt;
    &lt;asp:BoundColumn HeaderText="Order ID" DataField="OrderID" /&gt;
    <!-- 定义列显示 OrderDate 字段 --&gt;
    &lt;asp:BoundColumn HeaderText="Order Date" DataField="OrderDate" DataFormatString="{0:d}" /&gt;
    <!-- 定义列显示 OrderTotal 字段 --&gt;
    &lt;asp:BoundColumn HeaderText="Order Total" DataField="OrderTotal" DataFormatString="{0:c}" /&gt;
    <!-- 定义列显示 ShipDate 字段 --&gt;
    &lt;asp:BoundColumn HeaderText="Ship Date" DataField="ShipDate" DataFormatString="{0:d}" /&gt;
    <!-- 定义链接列指向查看订单信息的页面 --&gt;
    &lt;asp:HyperLinkColumn HeaderText="Show Details" Text="Show Details" DataNavigateUrlField=
"OrderID" DataNavigateUrlFormatString="vieworderdetails.aspx?OrderID={0}" /&gt;
    <!-- 结束 DataGrid 控件的列集合定义 --&gt;
&lt;/Columns&gt;
<!-- 结束 DataGrid 控件定义 --&gt;
&lt;/asp:DataGrid&gt;
&lt;/td&gt;
&lt;/tr&gt;</pre>
```

```
</table>
</form>
</td>
</tr>
</table>
</td>
</tr>
</table>
</body>
</HTML>
```

CustomerDetail.aspx 页面的主体为显示用户订单列表的 DataGrid 控件，该 DataGrid 控件中列集合的最后一列是链接列，该列的链接指向察看订单详细的页面 ViewOrderDetail.aspx。

● ViewOrderDetail.aspx

```
<%@ Page language="c#" Codebehind="ViewOrderDetails.aspx.cs" AutoEventWireup="false" Inherits="ASPNET.StarterKit.Commerce.ViewOrderDetails" %>
<!-- 引入用户控件 Header -->
<%@ Register TagPrefix="ASPNETCommerce" TagName="Header" Src="_Header.ascx" %>
<!-- 引入用户控件 Menu -->
<%@ Register TagPrefix="ASPNETCommerce" TagName="Menu" Src="_Menu.ascx" %>
<HTML>
<HEAD>
<link rel="stylesheet" type="text/css" href="ASPNETCommerce.css">
</HEAD>
<body background="images/sitebkgrd.gif" leftmargin="0" topmargin="0" rightmargin="0" bottommargin="0" marginheight="0" marginwidth="0">
<table cellspacing="0" cellpadding="0" width="100%" border="0">
<tr>
<td colspan="2">
<ASPNETCommerce:Header ID="Header1" runat="server" />
</td>
</tr>
<tr>
<td valign="top">
<ASPNETCommerce:Menu id="Menu1" runat="server" />

</td>
<td align="left" valign="top" width="100%" nowrap>
<table height="100%" align="left" cellspacing="0" cellpadding="0" width="100%" border="0">
<tr valign="top">
<td nowrap>
<br>

<table cellspacing="0" cellpadding="0" width="100%" border="0">
<tr>
<td class="ContentHead">
Customer Order
Details
<br>
</td>
</tr>
```

```
</table>


<asp:Label id="MyError" CssClass="ErrorText" EnableViewState="false" runat="Server" />
<table id="detailsTable" height="100%" cellspacing="0" cellpadding="0" width="550" border="0"
EnableViewState="false" runat="server">
<tr valign="top">
<td width="100%" class="Normal">
<b>Customer Name: </b>

<asp:Label ID="lblCustomerName" EnableViewState="false" runat="server" />
<br>
<b>Order Number Is: </b>

<asp:Label ID="lblOrderNumber" EnableViewState="false" runat="server" />
<br>
<b>Order Date: </b>

<asp:Label ID="lblOrderDate" EnableViewState="false" runat="server" />
<br>
<b>Ship Date: </b>

<asp:Label ID="lblShipDate" EnableViewState="false" runat="server" />
<br>

<asp:DataGrid id="GridControl1" width="90%" BorderColor="black" GridLines="Vertical" cellpadding="4"
cellspacing="0" Font-Name="Verdana" Font-Size="8pt" ShowFooter="true" HeaderStyle-CssClass=
"CartListHead"
FooterStyle-CssClass="cartlistfooter" ItemStyle-CssClass="CartListItem"
AlternatingItemStyle-CssClass= "CartListItemAlt"
AutoGenerateColumns="false" runat="server">

<Columns>

<asp:BoundColumn HeaderText="Product Name" DataField="ModelName" />

<asp:BoundColumn HeaderText="Model Number" DataField="ModelNumber" />

<asp:BoundColumn HeaderText="Quantity" DataField="Quantity" />

<asp:BoundColumn HeaderText="Price" DataField="UnitCost" DataFormatString="{0:c}" />

<asp:BoundColumn HeaderText="Subtotal" DataField="ExtendedAmount" DataFormatString="{0:c}" />

</Columns>

</asp:DataGrid>
<br>
<b>Total: </b>

<asp:Label ID="lblTotal" EnableViewState="false" runat="server" />
```

```
<asp:Label ID="lblTotal" EnableViewState="false" runat="server" />
</td>
</tr>
</table>
</td>
</tr>
</table>
</td>
</tr>
</table>
</body>
</HTML>
```

ViewOrderDetail.aspx 页面中显示包括订单主体信息和详细购物条目在内的订单信息，其中订单主体信息字段都是以 Label 控件的形式展现的，订单详细购物条目是通过 DataGrid 控件形式展现的。

9.3 扩展 Web Service

本书在第 8 章中介绍了 ASP.NET Commerce Starter Kit 提供的一个简单的 Web Service，ASP.NET Commerce Starter Kit 提供的 Web Service 包括两个方法，但是对于实际的应用这两个方法还是远远不够的。第三方 Web 应用程序通过 Web Service 所提供的方法要完成一套完整的业务逻辑，其中包括从商品列表的浏览到商品详细信息的浏览，还有商品的订购和订单查询等功能。

本章将完善 ASP.NET Commerce Starter Kit 所提供的 Web Service 并建立一个通过调用 ASP.NET Commerce Starter Kit 提供的 Web Service 搭建的 Web 应用程序。本节将介绍完善和补充 ASP.NET Commerce Starter Kit 的 Web Service，在 9.4 节中将介绍搭建调用 ASP.NET Commerce Starter Kit 提供 Web Service 的 Web 应用程序。

9.3.1 需求分析

通过前面章节对 ASP.NET Commerce Starter Kit 功能方面的讨论，可以看到 ASP.NET Commerce Starter Kit 提供了一个电子商务 Web 应用程序最基本的功能。其中包括商品浏览、商品订购、订单查询和商品评论等。实际上为了方便第三方应用程序通过 Web Service 实现电子商务的应用逻辑，ASP.NET Commerce Starter Kit 应该将系统中基本的业务逻辑方法都暴露在 Web Service 上。如果 ASP.NET Commerce Starter Kit 的 Web Service 并没有提供某一个实现系统业务逻辑的方法，那么第三方应用程序是无法实现该业务逻辑的。

在补充完善 ASP.NET Commerce Starter Kit 的 Web Service 中，需要完善的业务逻辑功能有如下几个。

- 获得商品分类列表
- 按照商品分类获得该分类的商品列表
- 获得某个商品的详细信息
- 获得某个用户的订单列表
- 为某个商品添加评论
- 按照订单号获得该订单对象

以上罗列的方法加上 ASP.NET Commerce Starter Kit 原本提供的两个方法基本上能满足第三方应用

程序实现一个简单电子商务应用的需求。

9.3.2 数据层

为了实现 ASP.NET Commerce Starter Kit 的 Web Service 所提供的部分方法，需要对 ASP.NET Commerce Starter Kit 的数据层做一些小的调整，这些调整主要集中在数据对象的获得方面。因为 ASP.NET Commerce Starter Kit 的数据层中目前所提供的大部分方法都是以 SqlDataReader 对象作为数据载体，但是 SqlDataReader 作为一个需要保存数据库连接状态的数据对象，将其应用在 Web Service 中显得不灵活，因此决定采用 DataSet 对象作为 Web Service 中方法返回的数据对象载体。因此要扩充 ASP.NET Commerce Starter Kit 的数据层。

在数据层的商品类 ProductDB.cs 文件中加入如下两个方法。

● ws_GetProductCategories

ws_GetProductCategories 方法是调用存储过程 CMRC_ProductCategoryList 获得商品分类列表的 DataSet 的方法。该方法代码清单如下。

```
public DataSet ws_GetProductCategories()
{
    //实例化 SqlConnection 对象和 SqlDataAdapter 对象
    SqlConnection myConnection = new SqlConnection(ConfigurationSettings.AppSettings["ConnectionString"]);
    SqlDataAdapter myCommand = new SqlDataAdapter("CMRC_ProductCategoryList", myConnection);
    //指定 SqlDataAdapter 对象的执行类型
    myCommand.SelectCommand.CommandType = CommandType.StoredProcedure;
    //打开数据库连接对象
    myConnection.Open();
    //执行 SqlDataAdapter 填充 DataSet 对象
    DataSet result = new DataSet();
    myCommand.Fill(result);
    //关闭 SqlConnection 对象
    myConnection.Close();
    //返回 DataSet 对象
    return result;
}
```

● ws_GetProductsList

ws_GetProductsList 方法是根据商品分类 categoryId 获得该分类的商品列表。该方法调用存储过程 CMRC_ProductsByCategory 返回存储商品列表数据的 DataSet 对象。该方法的代码清单如下。

```
public DataSet ws_GetProductsList(int categoryId)
{
    //实例化 SqlConnection 对象和 SqlDataAdapter 对象
    SqlConnection myConnection = new SqlConnection(ConfigurationSettings.AppSettings["ConnectionString"]);
    SqlDataAdapter myCommand = new SqlDataAdapter("CMRC_ProductsByCategory", myConnection);
    //指定 SqlDataAdapter 对象的执行类型
    myCommand.SelectCommand.CommandType = CommandType.StoredProcedure;
    //为 SqlDataAdapter 对象添加参数 CategoryID
    SqlParameter parameterCategoryID = new SqlParameter("@CategoryID", SqlDbType.Int, 4);
    parameterCategoryID.Value = categoryId;
    myCommand.SelectCommand.Parameters.Add(parameterCategoryID);
    //打开数据库连接对象
    myConnection.Open();
```

```

//执行 SqlDataAdapter 填充 DataSet 对象
DataSet result = new DataSet();
myCommand.Fill(result);
//关闭 SqlConnection 对象
myConnection.Close();
//返回 DataSet 对象
return result;
}

```

在数据层的订单类 OrderDB.cs 文件中加入如下的方法。

● ws_GetCustomerOrders

ws_GetCustomerOrders 方法根据用户的 customerID 返回该用户的订单列表。该方法调用存储过程 CMRC_OrdersList 返回存储商品列表数据的 DataSet 对象。该方法的代码清单如下。

```

public DataSet ws_GetCustomerOrders(String customerID)
{
    //实例化 SqlConnection 对象和 SqlDataAdapter 对象
    SqlConnection myConnection = new SqlConnection(ConfigurationSettings.AppSettings["ConnectionString"]);
    SqlDataAdapter myCommand = new SqlDataAdapter("CMRC_OrdersList", myConnection);
    //指定 SqlDataAdapter 对象的执行类型
    myCommand.SelectCommand.CommandType = CommandType.StoredProcedure;
    //为 SqlDataAdapter 对象添加参数 CustomerID
    SqlParameter parameterCustomerid = new SqlParameter("@CustomerID", SqlDbType.Int, 4);
    parameterCustomerid.Value = Int32.Parse(customerID);
    myCommand.SelectCommand.Parameters.Add(parameterCustomerid);
    //打开数据库连接对象
    myConnection.Open();
    //执行 SqlDataAdapter 填充 DataSet 对象
    DataSet result = new DataSet();
    myCommand.Fill(result);
    //关闭 SqlConnection 对象
    myConnection.Close();
    //返回 DataSet 对象
    return result;
}

```

注意：为了统一命名规范，本书中针对增加 Web Service 方法所对 ASP.NET Commerce Starter Kit 的数据层增加的方法统一以“ws_”作为方法名的前缀。

9.3.3 逻辑层

ASP.NET Commerce Starter Kit 提供的 Web Service 的逻辑层统一建立在 InstantOrder.asmx.cs 中，为了方便引用，本章中新增的 Web Service 方法同样也建立在这个文件中。根据需求分析中的设计，新增如下方法。

● CategoriesList

```

[WebMethod(Description="获得商品分类列表", EnableSession=false)]
public DataSet CategoriesList()
{
    //实例化一个 ProductsDB 对象
    ASPNET.StarterKit.Commerce.ProductsDB objProduct = new ProductsDB();

```

```
//调用ProductsDB对象的ws_GetProductCategories方法返回存储商品分类的DataSet对象
return objProduct.ws_GetProductCategories();
}
```

CategoriesList方法的SOAP请求的示例如下。

```
<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
<soap:Body>
<CategoriesList xmlns="http://tempuri.org/" />
</soap:Body>
</soap:Envelope>
```

CategoriesList方法的SOAP响应的示例如下。

```
<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
<soap:Body>
<CategoriesListResponse xmlns="http://tempuri.org/">
<CategoriesListResult>
<xsd:schema>schema</xsd:schema>xml</CategoriesListResult>
</CategoriesListResponse>
</soap:Body>
</soap:Envelope>
```

● ProductsList

```
[WebMethod(Description="按照商品分类获得该分类的商品列表", EnableSession=false)]
public DataSet ProductsList(int categoryId)
{
    //实例化一个ProductsDB对象
    ASPNET.StarterKit.Commerce.ProductsDB objProduct = new ProductsDB();
    //调用ProductsDB对象的ws_GetProductsList方法返回存储商品信息的DataSet对象
    return objProduct.ws_GetProductsList(categoryId);
}
```

ProductsList方法的SOAP请求示例如下。

```
<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
<soap:Body>
<ProductsList xmlns="http://tempuri.org/">
<categoryId>int</categoryId>
</ProductsList>
</soap:Body>
</soap:Envelope>
```

ProductsList方法的SOAP响应示例如下。

```
<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
<soap:Body>
<ProductsListResponse xmlns="http://tempuri.org/">
<ProductsListResult>
<xsd:schema>schema</xsd:schema>xml</ProductsListResult>
```

```

    </ProductsListResponse>
    </soap:Body>
</soap:Envelope>

● getProduct

[WebMethod(Description="获得某个商品的详细信息", EnableSession=false)]
public ProductDetails getProduct(int productId)
{
    //实例化一个 ProductsDB 对象
    ASPNET.StarterKit.Commerce.ProductsDB objProduct = new ProductsDB();
    //调用 ProductsDB 对象的 GetProductDetails 方法返回一个商品对象 ProductDetails
    return objProduct.GetProductDetails(productId);
}

```

getProduct 方法的 SOAP 请求示例如下。

```

<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
    <soap:Body>
        <getProduct xmlns="http://tempuri.org/">
            <productId>int</productId>
        </getProduct>
    </soap:Body>
</soap:Envelope>

```

getProduct 方法的 SOAP 响应示例如下。

```

<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
    <soap:Body>
        <getProductResponse xmlns="http://tempuri.org/">
            <getProductResult>
                <ModelNumber>string</ModelNumber>
                <ModelName>string</ModelName>
                <ProductImage>string</ProductImage>
                <UnitCost>decimal</UnitCost>
                <Description>string</Description>
            </getProductResult>
        </getProductResponse>
    </soap:Body>
</soap:Envelope>

```

● getOrderList

```

[WebMethod(Description="获得某个用户的订单列表", EnableSession=false)]
public DataSet getOrderList(string userName, string password)
{
    //实例化一个 CustomersDB 对象
    ASPNET.StarterKit.Commerce.CustomersDB accountSystem = new ASPNET.StarterKit.Commerce.CustomersDB();
    //调用 CustomersDB 的 Login 方法验证用户登录
    String customerId = accountSystem.Login(userName, ASPNET.StarterKit.Commerce.Components.Security.Encrypt(password));
    //如果 Login 方法返回的 customerId 不为空则说明用户通过身份验证
    if (customerId == null)

```

```
{  
    throw new Exception("Error: Invalid Login!");  
}  
//实例化一个OrdersDB对象  
ASPNET.StarterKit.Commerce.OrdersDB orderSystem = new ASPNET.StarterKit.Commerce.OrdersDB();  
//调用OrdersDB对象的ws_GetCustomerOrders方法返回包含订单列表的DataSet对象  
return orderSystem.ws_GetCustomerOrders(customerId);  
}
```

getOrderList 方法的 SOAP 请求示例如下。

```
<?xml version="1.0" encoding="utf-8"?>  
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">  
  <soap:Body>  
    <getOrderList xmlns="http://tempuri.org/">  
      <userName>string</userName>  
      <password>string</password>  
    </getOrderList>  
  </soap:Body>  
</soap:Envelope>
```

getOrderList 方法的 SOAP 响应示例如下。

```
<?xml version="1.0" encoding="utf-8"?>  
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">  
  <soap:Body>  
    <getOrderListResponse xmlns="http://tempuri.org/">  
      <getOrderListResult>  
        <xsd:schema><xsd:schema>xml</getOrderListResult>  
      </getOrderListResponse>  
    </soap:Body>  
</soap:Envelope>
```

● getOrderDetails

```
[WebMethod(Description="按照OrderId获得一个订单对象", EnableSession=false)]  
public OrderDetails getOrderDetails(int orderId)  
{  
    //实例化一个OrdersDB对象  
    ASPNET.StarterKit.Commerce.OrdersDB objectOrder = new OrdersDB();  
    //调用OrdersDB对象的ViewOrderDetails方法返回一个订单对象OrderDetails  
    return objectOrder.ViewOrderDetails(orderId);  
}
```

getOrderDetails 方法的 SOAP 请求示例如下。

```
<?xml version="1.0" encoding="utf-8"?>  
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">  
  <soap:Body>  
    <getOrderDetails xmlns="http://tempuri.org/">  
      <orderId>int</orderId>  
    </getOrderDetails>  
  </soap:Body>  
</soap:Envelope>
```

getOrderDetails 方法的 SOAP 响应示例如下。

```
<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
<soap:Body>
<getOrderDetailsResponse xmlns="http://tempuri.org/">
<getOrderDetailsResult>
<CustomerName>string</CustomerName>
<OrderDate>dateTime</OrderDate>
<ShipDate>dateTime</ShipDate>
<OrderTotal>decimal</OrderTotal>
<OrderStatus>int</OrderStatus>
<OrderItems>
<xsd:schema>schema</xsd:schema>xml</OrderItems>
</getOrderDetailsResult>
</getOrderDetailsResponse>
</soap:Body>
</soap:Envelope>
```

● AddReview

```
[WebMethod(Description="为某个商品添加评论", EnableSession=false)]
public void AddReview(int productID, string customerName, string customerEmail, int rating, string comments)
{
    //实例化一个 ReviewsDB 对象
    ASPNET.StarterKit.Commerce.ReviewsDB objectReview = new ReviewsDB();
    //调用 ReviewsDB 对象的 AddReview 方法添加商品评论
    objectReview.AddReview(productID, customerName, customerEmail, rating, comments);
}
```

AddReview 方法的 SOAP 请求示例如下。

```
<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
<soap:Body>
<AddReview xmlns="http://tempuri.org/">
<productID>int</productID>
<customerName>string</customerName>
<customerEmail>string</customerEmail>
<rating>int</rating>
<comments>string</comments>
</AddReview>
</soap:Body>
</soap:Envelope>
```

AddReview 方法的 SOAP 响应示例如下。

```
<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
<soap:Body>
<AddReviewResponse xmlns="http://tempuri.org/" />
</soap:Body>
</soap:Envelope>
```

以上是对 ASP.NET Commerce Starter Kit 的 Web Service 的补充，经过补充之后第三方应用程序应该可以使用 ASP.NET Commerce Starter Kit 提供的 Web Service 完成简单的电子商务业务。访问 InstantOrder.asmx 效果如图 9-8 所示。

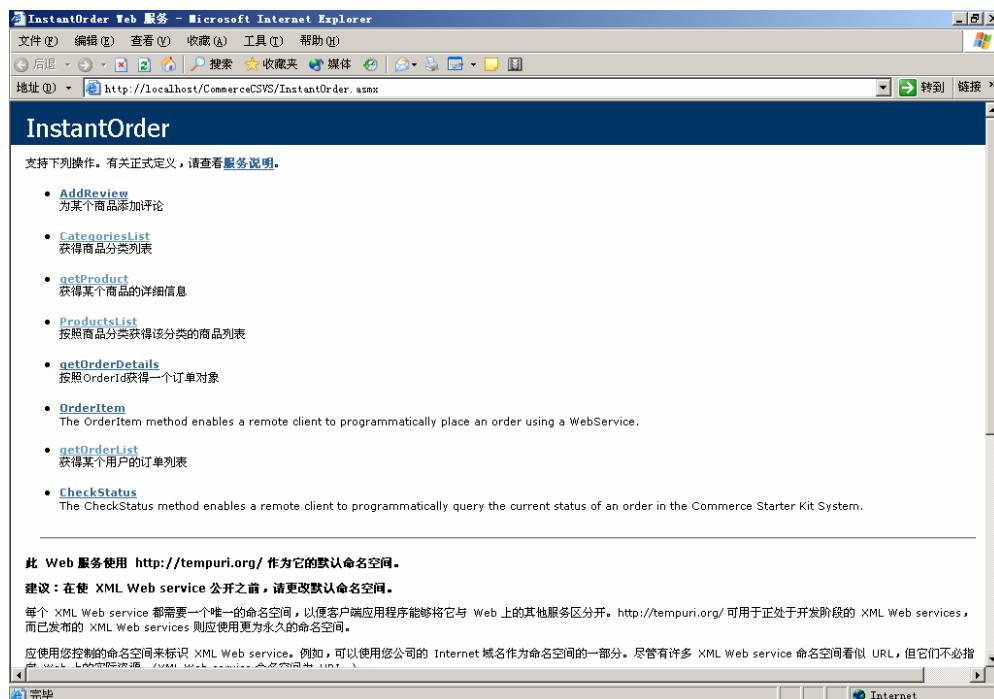


图 9-8 InstantOrder.asmx 效果

9.4 第三方调用

9.4.1 需求分析

在 9.3 节中介绍了扩展 ASP.NET Commerce Starter Kit 的 Web Service，实际上 Web 应用程序构建 Web Service 的目的还是以第三方应用程序和系统之间实现交互为目的的。如果没有第三方应用程序的调用，那么 Web Service 几乎形同虚设，毫无意义。因此，本节中将介绍构建调用 ASP.NET Commerce Starter Kit 的 Web Service 的第三方应用程序。

假设构建一个 ASP.NET Commerce Starter Kit 中文版站点的应用程序，其业务逻辑实现通过调用 ASP.NET Commerce Starter Kit 的 Web Service 来完成。为了达到演示的目的，简化部分系统中的功能，在新构建的站点中仅实现如下方面的功能。

- 商品分类列表
- 查看商品详细信息
- 商品订购
- 商品评论
- 订单查询
- 查看订单详细信息

对于用户对象，在新构建的站点中采用一个简化的处理办法。将用户名和密码写入 Web.config 配置

文件中，也就是说默认使用该站点的用户已经固定。当然，这也是一个B2B模式的简单体现。建立新的站点需要如下几个步骤。

- 首先，在Visual Studio .NET中建立名为InvokeCommerce的ASP.NET Web应用程序项目。
- 在新建立的Web应用程序项目中添加Web引用，Web引用的Url为ASP.NET Commerce Starter Kit的Web Service的Url，将Web引用名更改为CommerceService。
- 在新建立的Web应用程序的Web.config中添加默认用户名和密码的配置信息，代码如下。

```
<appSettings>
    <add key="CustomerName" value="wb@commerce.com" />
    <add key="Password" value="wb" />
</appSettings>
```

注意：本章中仅以演示建立调用Web Service的第三方应用程序为目的，因此省略和简化ASP.NET Commerce Starter Kit中的用户模块相关的功能，但是用户对象确实是电子商务应用程序中不可缺少的一部分，所以在新建立的Web应用程序中采用在Web.config中记录用户名和密码的方法，这种方法忽略了安全性等方面的因素。因此在实际的应用程序中，不建议采用这种方法。

9.4.2 开发 InvokeCommerce

在需求分析中已经提出了InvokeCommerce的功能要求，开发中将以这些功能为基础在InvokeCommerce中实现一个简单的电子商务应用。

- 商品分类列表 Default.aspx

商品分类列表页面Default.aspx为系统默认的首页，该页面中以列表的形式展现商品的部分信息。用户通过选择商品分类下拉列表控件选择所要查看的商品分类。其页面效果如图9-9所示。



图9-9 商品分类列表 Default.aspx

Default.aspx的后台编码类Default.aspx.cs的关键部分代码如下。

```
//定义页面控件
protected System.Web.UI.WebControls.DropDownList ddlCategory;
protected System.Web.UI.WebControls.DataGrid dgProduct;
```

```
//定义调用 Web Service 的对象
protected CommerceService.InstantOrder ObjectOrder;

private void Page_Load(object sender, System.EventArgs e)
{
    //实例化调用 Web Service 的对象
    ObjectOrder = new InvokeCommerce.CommerceService.InstantOrder();

    if(!Page.IsPostBack)
    {
        //当页面首次载入时绑定显示商品分类信息的 DropDownList 控件
        ddlCategory.DataSource = ObjectOrder.CategoriesList();
        ddlCategory.DataTextField = "CategoryName";
        ddlCategory.DataValueField = "CategoryID";
        ddlCategory.DataBind();
        //隐藏显示商品列表信息的 DataGrid 控件
        dgProduct.Visible = false;
    }
    else
    {
        //当页面是被提交时根据 DropDownList 控件的选定值绑定 DataGrid 控件
        dgProduct.Visible = true;
        dgProduct.DataSource = ObjectOrder.ProductsList(int.Parse(ddlCategory.SelectedValue));
        dgProduct.DataBind();
    }
}
```

Default.aspx 的前台页面代码如下。

```
<%@ Page language="c#" Codebehind="Default.aspx.cs" AutoEventWireup="false" Inherits="InvokeCommerce._Default" %>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN" >
<HTML>
<HEAD>
<title>Default</title>
</HEAD>
<body>
<form id="Form1" method="post" runat="server">
<table width="500" height="37" border="0" cellpadding="0" cellspacing="0">
    <tr>
        <td align="center" valign="middle">Commerce 电子商务站点(中文版)</td>
    </tr>
</table>
<table width="500" border="0" cellspacing="0" cellpadding="0">
    <tr>
        <td width="347">请您选择商品分类:</td>
        <td width="153" align="right"><a href="OrderList.aspx">我的订单列表</a></td>
    </tr>
</table>
<asp:DataGrid id="dgProduct" runat="server" Width="700px" AutoGenerateColumns="False" BorderColor="#E7E7FF"
```

```

    BorderStyle="None" BorderWidth="1px" BackColor="White" CellPadding="3" GridLines="Horizontal">
    <SelectedItemStyle Font-Bold="True" ForeColor="#F7F7F7" BackColor="#738A9C"></SelectedItemStyle>
    <AlternatingItemStyle BackColor="#F7F7F7"></AlternatingItemStyle>
    <ItemStyle ForeColor="#4A3C8C" BackColor="#E7E7FF"></ItemStyle>
    <HeaderStyle Font-Bold="True" ForeColor="#F7F7F7" BackColor="#4A3C8C"></HeaderStyle>
    <FooterStyle ForeColor="#4A3C8C" BackColor="#B5C7DE"></FooterStyle>
    <Columns>
        <asp:BoundColumn DataField="ProductID" ReadOnly="True" HeaderText="商品 ID"></asp:BoundColumn>
        <asp:HyperLinkColumn Target="_blank" DataNavigateUrlField="ProductId" DataNavigateUrlFormatString="ProductDetail.aspx?productid={0}"
            DataTextField="ModelName" HeaderText="商品名称"></asp:HyperLinkColumn>
        <asp:BoundColumn DataField="ModelNumber" HeaderText="商品型号" DataFormatString="[{0}]">
    </asp:BoundColumn>
        <asp:BoundColumn DataField="UnitCost" HeaderText="价格" DataFormatString="$ {0}">
    </asp:BoundColumn>
        <asp:HyperLinkColumn Text="购买" DataNavigateUrlField="ProductID" DataNavigateUrlFormatString="OrderItem.aspx?ProductID={0}"
            HeaderText="购买"></asp:HyperLinkColumn>
        <asp:HyperLinkColumn Text="评论" Target="_blank" DataNavigateUrlField="ProductId"
            DataNavigateUrlFormatString="ReviewProduct.aspx?ProductId={0}"
            HeaderText="评论"></asp:HyperLinkColumn>
    </Columns>
    <PagerStyle HorizontalAlign="Right" ForeColor="#4A3C8C" BackColor="#E7E7FF" Mode="NumericPages">
    </PagerStyle>
</asp:DataGrid></form>
</body>
</HTML>

```

● 商品详细信息用户控件

在电子商务 Web 应用程序中，除了商品详细信息查看页面之外很多页面都涉及到展示商品的详细信息，例如订购商品页面、商品评论页面等等。因此有必要将商品详细信息的功能实现成为一个用户控件，当其他页面需要调用商品详细信息的时候调用该用户控件就可以实现商品详细信息展示的功能，该用户控件具有较强的可重用性。将该用户空间命名为 Product.ascx，其后台编码类 Product.ascx.cs 的关键代码如下。

```

//定义页面中显示商品详细信息的控件
public System.Web.UI.WebControls.Label lbl modelName;
public System.Web.UI.WebControls.Label lbl ModelNumber;
public System.Web.UI.WebControls.Label lbl UnitCost;
public System.Web.UI.WebControls.Label lbl ProductDescription;
//定义调用 Web Service 的对象
public CommerceService.InstantOrder ObjectProduct;
//定义调用 Web Service 中的商品对象
public CommerceService.ProductDetails ObjectProductDetail;
//定义所显示商品的商品 ID
public int ProductId;

private void Page_Load(object sender, System.EventArgs e)
{
    //实例化调用 Web Service 的对象
    ObjectProduct = new InvokeCommerce.CommerceService.InstantOrder();
}

```

```
//调用 Web Service 的 getProduct 方法获取商品对象
ObjectProductDetail = ObjectProduct.getProduct(ProductId);
//如果商品对象不为空则绑定页面上显示商品信息的控件
if(ObjectProductDetail != null)
{
    lblModelName.Text = ObjectProductDetail.ModelName;
    lblModelNumber.Text = "[" + ObjectProductDetail.ModelNumber + "]";
    lblUnitCost.Text = "$ " + ObjectProductDetail.UnitCost.ToString();
    lblProductDescription.Text = ObjectProductDetail.Description;
}

}
```

Product.ascx 的前台页面代码如下。

```
<%@ Control Language="c#" AutoEventWireup="false" Codebehind="Product.ascx.cs" Inherits="InvokeCommerce.
Product" TargetSchema="http://schemas.microsoft.com/intellisense/ie5"%>


|       |                                                                                         |
|-------|-----------------------------------------------------------------------------------------|
| 商品名称: | &ampnbsp         <asp:Label id="lblmodelName" runat="server"></asp:Label></td>          |
| 商品型号: | &ampnbsp         <asp:Label id="lblModelNumber" runat="server"></asp:Label></td>        |
| 商品单价: | &ampnbsp         <asp:Label id="lblUnitCost" runat="server"></asp:Label></td>           |
| 商品描述: | &ampnbsp         <asp:Label id="lblProductDescription" runat="server"></asp:Label></td> |


```

在商品信息查看页面 ProductDetail.aspx 需要对 Product.ascx 进行调用，其调用方法极其简单，只需要对 Product 用户控件的 ProductId 属性进行赋值即可。其后台编码类中的关键代码如下。

```
protected Product ProductInfor;
private void Page_Load(object sender, System.EventArgs e)
{
    //从Url中获得ProductId并对其进行赋值
    ProductInfor.ProductId = int.Parse(Request.QueryString["ProductId"]);
    //实例化用户控件Product
    ProductInfor = new Product();
}
```

商品信息查看页面 ProductDetail.aspx 的页面效果如图 9-10 所示。

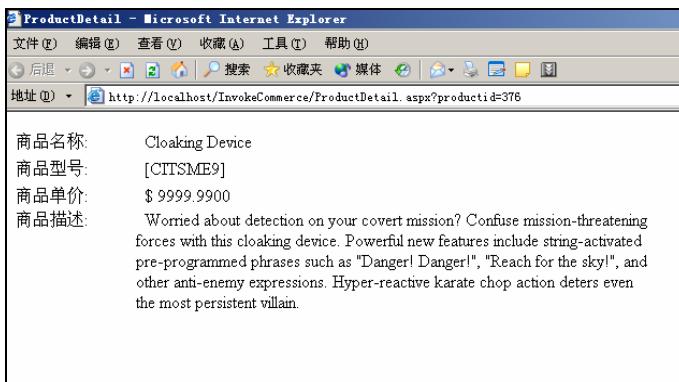


图 9-10 商品信息查看页面 ProductDetail.aspx

● 商品订购

商品订购页面 OrderItem.aspx 中首先要对商品信息进行显示，然后提供 TextBox 控件供用户输入购买商品的数量，在用户输入购买数量单击提交按钮之后完成订单的订购。其后台编码类 OrderItem.aspx.cs 的关键代码清单如下。

```
//定义页面显示控件及验证控件
protected System.Web.UI.WebControls.TextBox txbQuantity;
protected System.Web.UI.WebControls.CompareValidator CompareValidator1;
protected System.Web.UI.WebControls.Button btnSubmit;
protected System.Web.UI.WebControls.RequiredFieldValidator RequiredFieldValidator1;
protected System.Web.UI.HtmlControls.HtmlTable Ordertable;

//定义调用 Web Service 的对象
protected CommerceService.InstantOrder ObjectOrder;
//定义调用 Web Service 中的订单对象
protected CommerceService.OrderDetails ObjectOrderDetail;
//定义 Product 用户控件
protected Product ProductInfor;

private void Page_Load(object sender, System.EventArgs e)
{
    //从Url 获取 ProductId 并实例化用户控件 Product
    ProductInfor.ProductId = int.Parse( Request.QueryString["ProductId"] );
    ProductInfor = new Product();
    //当页面被提交时完成订单的生成
    if (Page.IsPostBack)
    {
        //获取用户名和密码
        string username = ConfigurationSettings.AppSettings["customername"];
        string password = ConfigurationSettings.AppSettings["password"];
        //获取购商品和订购数量
        int productid = int.Parse( Request.QueryString["ProductId"] );
        int quantity = int.Parse(txbQuantity.Text);
        //实例化调用 Web Service 的对象
        ObjectOrder = new InvokeCommerce.CommerceService.InstantOrder();
        //调用 Web Service 的 OrderItem 方法完成商品的订购
        ObjectOrderDetail = ObjectOrder.OrderItem(username, password, productid, quantity);
        //如果订单对象不为空则说明订购成功
    }
}
```

```
        if (ObjectOrderDetail != null)
    {
        Ordertable.Visible = false;
        Response.Write("订单已提交!请
```

商品订购页面 OrderItem.aspx 的代码清单如下。

```
<%@ Register TagPrefix="uc1" TagName="Product" Src="Product.ascx" %>
<%@ Page language="c#" Codebehind="OrderItem.aspx.cs" AutoEventWireup="false" Inherits="InvokeCommerce.OrderItem" %>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN" >
<HTML>
<HEAD>
<title>OrderItem</title>
</HEAD>
<body MS_POSITIONING="GridLayout">
<form id="Form1" method="post" runat="server">
<uc1:Product id="ProductInfor" runat="server"></uc1:Product>
<table width="500" border="0" cellspacing="0" cellpadding="0" id="Ordertable" runat="server">
<tr>
<td style="WIDTH: 104px">购买数量:</td>
<td style="WIDTH: 83px">&nbsp;
<asp:TextBox id="txbQuantity" runat="server" Width="30px"></asp:TextBox></td>
<td>&nbsp;
<asp:Button id="btnSubmit" runat="server" Text="提交"></asp:Button>
<asp:CompareValidator id="CompareValidator1" runat="server" ErrorMessage="购买数量必须为数字" ControlToValidate="txbQuantity" Type="Integer" Display="Dynamic" Operator="DataTypeCheck"></asp:CompareValidator>
<asp:RequiredFieldValidator id="RequiredFieldValidator1" runat="server" ErrorMessage="购买数量不能为空" ControlToValidate="txbQuantity" Display="Dynamic"></asp:RequiredFieldValidator></td>
<td>&nbsp;</td>
</tr>
</table>
</form>
</body>
</HTML>
```

商品订购页面 OrderItem.aspx 的页面效果如图 9-11 所示。

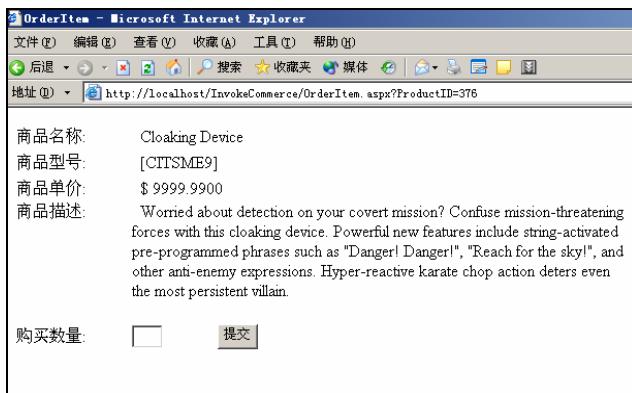


图 9-11 商品订购页面 OrderItem.aspx

● 商品评论

商品评论页面 ReviewProduct.aspx 和商品订购页面类似,首先调用用户控件 Product 显示商品信息。然后提供用户发表评论所需要的页面控件,用户在填写完评论信息之后单击提交按钮,完成对商品的评论,商品评论页面的后台编码类 Review.aspx.cs 的代码清单如下。

```
//定义页面输入控件及验证控件
protected System.Web.UI.WebControls.TextBox txbCustomerName;
protected System.Web.UI.WebControls.TextBox txbCustomerEmail;
protected System.Web.UI.WebControls.DropDownList ddlRating;
protected System.Web.UI.WebControls.Button btnSubmit;
protected System.Web.UI.WebControls.RequiredFieldValidator RequiredFieldValidator1;
protected System.Web.UI.WebControls.RequiredFieldValidator RequiredFieldValidator2;
protected System.Web.UI.WebControls.TextBox txbComments;
protected System.Web.UI.WebControls.RequiredFieldValidator RequiredFieldValidator3;
//定义调用 Web Service 的对象
protected CommerceService.InstantOrder ObjectReview;
protected System.Web.UI.HtmlControls.HtmlTable ReviewTable;
//定义显示商品信息的用户控件
protected Product ProductInfor;
//页面载入方法
private void Page_Load(object sender, System.EventArgs e)
{
    //从Url中获得商品的ProductId
    ProductInfor.ProductId = int.Parse(Request.QueryString["ProductId"]);
    //实例化显示商品信息的用户控件
    ProductInfor = new Product();
    //当页面被提交时完成对商品的评论
    if(Page.IsPostBack)
    {
        //获取用户输入的评论信息
        int productid = int.Parse(Request.QueryString["ProductId"]);
        string customername = txbCustomerName.Text;
        string customeremail = txbCustomerEmail.Text;
        int rating = int.Parse(Request.Params["ddlRating"]);
        string comments = txbComments.Text;
        //实例化调用 Web Service 的对象
        ObjectReview = new InvokeCommerce.CommerceService.InstantOrder();
    }
}
```

```
//调用 AddReview 方法完成商品评论的添加
ObjectReview.AddReview(productid, customername, customeremail, rating, comments);
ReviewTable.Visible = false;
Response.Write("评论意见已提交!请
```

商品评论页面 ReviewProduct.aspx 的前台页面代码如下。

```
<%@ Page language="c#" Codebehind="ReviewProduct.aspx.cs" AutoEventWireup="false" Inherits="InvokeCommerce.
ReviewProduct" %>
<%@ Register TagPrefix="uc1" TagName="Product" Src="Product.ascx" %>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN" >
<HTML>
<HEAD>
<title>ReviewProduct</title>
</HEAD>
<body>
<form id="Form1" method="post" runat="server">
<uc1:Product id="ProductInfor" runat="server"></uc1:Product><br>
<table width="500" border="0" cellspacing="0" cellpadding="0" id="ReviewTable" runat="server">
<tr>
<td width="92" height="25">评论人:</td>
<td width="408" height="25">&ampnbsp
<asp:TextBox id="txbCustomerName" runat="server"></asp:TextBox>
<asp:RequiredFieldValidator id="RequiredFieldValidator1" runat="server" ErrorMessage="评论人不能为空"
Display="Dynamic"
ControlToValidate="txbCustomerName"></asp:RequiredFieldValidator></td>
</tr>
<tr>
<td height="25">Email:</td>
<td height="25">&ampnbsp
<asp:TextBox id="txbCustomerEmail" runat="server"></asp:TextBox>
<asp:RequiredFieldValidator id="RequiredFieldValidator2" runat="server" ErrorMessage="Email 不能为空
Display="Dynamic"
ControlToValidate="txbCustomerEmail"></asp:RequiredFieldValidator></td>
</tr>
<tr>
<td height="25">打分:</td>
<td height="25">&ampnbsp
<asp:DropDownList id="ddlRating" runat="server">
<asp:ListItem Value="5">5 分</asp:ListItem>
<asp:ListItem Value="4">4 分</asp:ListItem>
<asp:ListItem Value="3">3 分</asp:ListItem>
<asp:ListItem Value="2">2 分</asp:ListItem>
<asp:ListItem Value="1">1 分</asp:ListItem>
</asp:DropDownList></td>
</tr>
<tr>
<td height="25">意见:</td>
<td height="25">&ampnbsp
<asp:TextBox id="txbComments" runat="server" Width="368px" TextMode="MultiLine" Rows="5">
```

```

</asp:TextBox>
    <asp:RequiredFieldValidator id="RequiredFieldValidator3" runat="server" ErrorMessage="意见不能为空"
Display="Dynamic" ControlToValidate="txbComments"></asp:RequiredFieldValidator></td>
</tr>
<tr>
<td colspan="2" align="center">
    <asp:Button id="btnSubmit" runat="server" Text="提 交"/></asp:Button>&nbsp;</td>
</tr>
</table>
</form>
</body>
</HTML>

```

商品评论页面 ReviewProduct.aspx 的页面效果如图 9-12 所示。

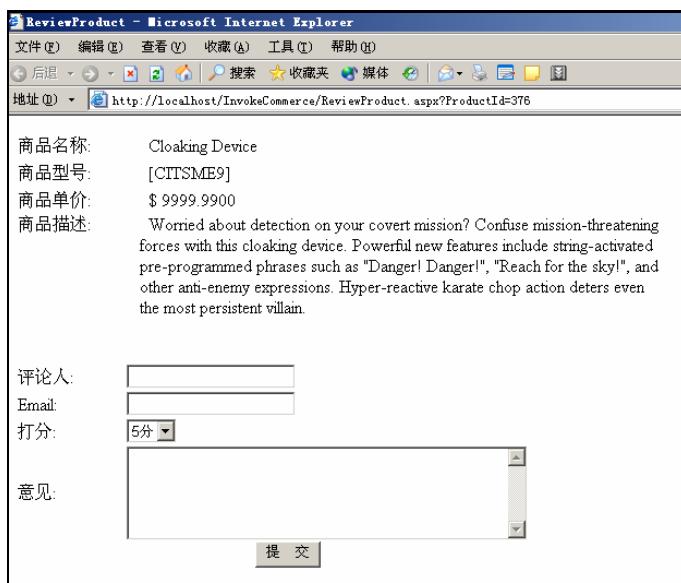


图 9-12 商品评论页面 ReviewProduct.aspx

● 订单查询

订单查询页面 OrderList.aspx 以列表的形式展示当前用户的订单列表，该列表中包含订单号、订单总价、订购日期、送达日期和当前状态等信息，用户可以点击每一条订单信息“详细”链接跳转到详细信息查看页面。订单查询页面的后台编码类 OrderList.aspx.cs 的关键代码清单如下。

```

//定义显示订单列表的 DataGrid 控件
protected System.Web.UI.WebControls.DataGrid dgOrderList;
//定义调用 Web Service 的对象
protected CommerceService.InstantOrder objServices;
//页面载入方法
private void Page_Load(object sender, System.EventArgs e)
{
    //获得用户名和密码
    string username = ConfigurationSettings.AppSettings["customername"];
    string password = ConfigurationSettings.AppSettings["password"];
    //实例化调用 Web Service 的对象
    objServices = new InvokeCommerce.CommerceService.InstantOrder();
}

```

```
//调用 getOrderList 方法为 DataGrid 控件获得数据源
dgOrderList.DataSource = objServices.getOrderList(username, password);
//绑定 DataGrid 控件
dgOrderList.DataBind();

}

//DataGrid 控件的 ItemDataBound 事件所触发的方法
private void dgOrderList_ItemDataBound(object sender, System.Web.UI.WebControls.DataGridItemEventArgs e)
{
    //当当前行为普通数据行时
    if(e.Item.ItemType == ListItemType.Item || e.Item.ItemType == ListItemType.AlternatingItem)
    {
        //获取显示订单状态的 Label 控件
        Label lblOrderStatus = (Label)e.Item.Cells[4].FindControl("lblOrderStatus");
        //获取订单状态信息
        int orderStatus = Convert.ToInt32(DataBinder.Eval(e.Item.DataItem, "orderstatus"));
        //根据订单状态信息设置订单状态 Label 控件的显示值
        switch(orderStatus)
        {
            case 0:
                lblOrderStatus.Text = "处理中";
                break;
            case 1:
                lblOrderStatus.Text = "已接受";
                break;
            case 2:
                lblOrderStatus.Text = "配送中";
                break;
            case 3:
                lblOrderStatus.Text = "已完成";
                break;
            default:
                lblOrderStatus.Text = "不明状态";
                break;
        }
    }
}
```

订单查询页面 OrderList.aspx 的前台页面使用 DataGrid 控件显示订单列表信息，其代码清单如下。

```
<%@ Page language="c#" Codebehind="Orderlist.aspx.cs" AutoEventWireup="false" Inherits="InvokeCommerce.Orderlist" %>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN" >
<HTML>
<HEAD>
<title>Orderlist</title>
</HEAD>
<body>
<form id="Form1" method="post" runat="server">
<TABLE id="Table1" height="37" cellSpacing="0" cellPadding="0" width="500" border="0">
    <TR>
        <TD vAlign="middle" align="center">Commerce 电子商务站点(中文版)</TD>
```

```
</TR>
</TABLE>
<table width="500" border="0" cellspacing="0" cellpadding="0">
<tr>
    <td height="25">&nbsp;</td>
    <td align="right"><a href="Default.aspx">商品列表</a></td>
</tr>
</table>
<asp:DataGrid id="dgOrderList" runat="server" AutoGenerateColumns="False" Width="600px" BorderColor="#E7E7FF"
    BorderStyle="None" BorderWidth="1px" BackColor="White" CellPadding="3" GridLines="Horizontal">
    <SelectedItemStyle Font-Bold="True" ForeColor="#F7F7F7" BackColor="#738A9C"></SelectedItemStyle>
    <AlternatingItemStyle BackColor="#F7F7F7"></AlternatingItemStyle>
    <ItemStyle ForeColor="#4A3C8C" BackColor="#E7E7FF"></ItemStyle>
    <HeaderStyle Font-Bold="True" ForeColor="#F7F7F7" BackColor="#4A3C8C"></HeaderStyle>
    <FooterStyle ForeColor="#4A3C8C" BackColor="#B5C7DE"></FooterStyle>
    <Columns>
        <asp:BoundColumn DataField="OrderID" ReadOnly="True" HeaderText="订单号"></asp:BoundColumn>
        <asp:BoundColumn DataField="OrderDate" HeaderText="订购日期" DataFormatString="{0:d}">
        </asp:BoundColumn>
        <asp:BoundColumn DataField="ShipDate" HeaderText="送达日期" DataFormatString="{0:d}">
        </asp:BoundColumn>
        <asp:BoundColumn DataField="OrderTotal" HeaderText="订单总额" DataFormatString="$ {0}">
        </asp:BoundColumn>
        <asp:TemplateColumn HeaderText="订单状态">
            <ItemTemplate>
                <asp:Label id="lblOrderStatus" runat="server"></asp:Label>
            </ItemTemplate>
        </asp:TemplateColumn>
        <asp:HyperLinkColumn Text="详细" DataNavigateUrlField="OrderId" DataNavigateUrlFormatString="OrderDetail.aspx?OrderId={0}"
            HeaderText="详细"></asp:HyperLinkColumn>
    </Columns>
    <PagerStyle HorizontalAlign="Right" ForeColor="#4A3C8C" BackColor="#E7E7FF" Mode="NumericPages">
    </PagerStyle>
</asp:DataGrid>
</form>
</body>
</HTML>
```

订单查询页面 OrderList.aspx 的页面效果如图 9-13。



图 9-13 订单查询页面 OrderList.aspx

● 查看订单详细信息

查看订单详细信息页面 OrderDetail.aspx 显示包括订单主体信息和详细订购信息在内的订单全部信息，该页面通过 Url 获得要显示订单的 OrderId，用多个 Label 控件显示订单主体信息，用 DataGrid 控件显示订单订购详细。查看订单详细信息页面后台编码类 OrderDetail.aspx.cs 代码清单如下。

```
//定义页面显示控件
protected System.Web.UI.WebControls.Label lblOrderId;
protected System.Web.UI.WebControls.Label lblOrderDate;
protected System.Web.UI.WebControls.Label lblOrderStatus;
protected System.Web.UI.WebControls.Label lblOrderTotal;
protected System.Web.UI.WebControls.DataGrid dgOrderDetail;
protected System.Web.UI.WebControls.Label lblShipDate;
//定义调用 Web Service 的对象
protected CommerceService.InstantOrder objServices;
//定义调用 Web Service 中的订单对象
protected CommerceService.OrderDetails objOrder;
//页面载入方法
private void Page_Load(object sender, System.EventArgs e)
{
    //获得订单 Id
    int orderId = int.Parse(Request.QueryString["OrderId"]);
    //实例化调用 Web Service 的对象
    objServices = new InvokeCommerce.CommerceService.InstantOrder();
    //调用 getOrderDetails 方法获得订单对象
    objOrder = objServices.getOrderDetails(orderId);
    //绑定显示订单信息的页面控件
    lblOrderId.Text = orderId.ToString();
    lblOrderDate.Text = objOrder.OrderDate.ToShortDateString();
    lblOrderTotal.Text = "$" + objOrder.OrderTotal.ToString();
    lblShipDate.Text = objOrder.ShipDate.ToShortDateString();
    int orderStatus = objOrder.OrderStatus;
    switch(orderStatus)
    {
        case 0:
            lblOrderStatus.Text = "处理中";
            break;
        case 1:
            lblOrderStatus.Text = "已接受";
            break;
    }
}
```

```
        case 2:  
            lblOrderStatus.Text = "配送中";  
            break;  
        case 3:  
            lblOrderStatus.Text = "已完成";  
            break;  
        default:  
            lblOrderStatus.Text = "不明状态";  
            break;  
    }  
    //绑定显示订单详细订购信息的 DataGrid 控件  
    dgOrderDetail.DataSource = objOrder.OrderItems;  
    dgOrderDetail.DataBind();  
}
```

OrderDetail.aspx 的前台页面代码清单如下。

```
<%@ Page language="c#" Codebehind="OrderDetail.aspx.cs" AutoEventWireup="false" Inherits="InvokeCommerce.  
OrderDetail" %>  
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN" >  
<HTML>  
<HEAD>  
<title>OrderDetail</title>  
</HEAD>  
<body>  
<form id="Form1" method="post" runat="server">  
<TABLE id="Table1" height="37" cellSpacing="0" cellPadding="0" width="500" border="0">  
<TR>  
<TD vAlign="middle" align="center">Commerce 电子商务站点(中文版)</TD>  
</TR>  
</TABLE>  
<table width="500" border="0" cellspacing="0" cellpadding="0">  
<tr>  
<td width="84" height="25">订单号:</td>  
<td width="416" height="25">&ampnbsp  
<asp:Label id="lblOrderId" runat="server"></asp:Label></td>  
</tr>  
<tr>  
<td width="84" height="25">订单日期:</td>  
<td width="416" height="25">&ampnbsp  
<asp:Label id="lblOrderDate" runat="server"></asp:Label></td>  
</tr>  
<tr>  
<td height="25">送达日期:</td>  
<td height="25">&ampnbsp  
<asp:Label id="lblShipDate" runat="server"></asp:Label></td>  
</tr>  
<tr>  
<td height="25">订单总额:</td>  
<td height="25">&ampnbsp  
<asp:Label id="lblOrderTotal" runat="server"></asp:Label></td>  
</tr>
```

```
<tr>
<td height="25">订单状态:</td>
<td height="25">&nbsp;
<asp:Label id="lblOrderStatus" runat="server"></asp:Label></td>
</tr>
<tr>
<td height="25" colspan="2">订单详细购买信息:</td>
</tr>
<tr align="left" valign="top">
<td height="173" colspan="2">&nbsp;
<asp:DataGrid id="dgOrderDetail" runat="server" Width="100%" AutoGenerateColumns="False">
<Columns>
    <asp:BoundColumn DataField="ModelName" HeaderText="产品名称"></asp:BoundColumn>
    <asp:BoundColumn DataField="ModelNumber" HeaderText="产品型号"></asp:BoundColumn>
    <asp:BoundColumn DataField="Quantity" HeaderText="购买数量"></asp:BoundColumn>
    <asp:BoundColumn DataField="UnitCost" HeaderText="单 价" DataFormatString="$ {0} ">
</asp:BoundColumn>
    <asp:BoundColumn DataField="ExtendedAmount" HeaderText=" 总 价 " DataFormatString="$ {0} ">
</asp:BoundColumn>
</Columns>
</asp:DataGrid></td>
</tr>
<tr>
<td height="25">&nbsp;</td>
<td height="25">&nbsp;</td>
</tr>
</table>
</form>
</body>
</HTML>
```

查看订单详细信息页面 OrderDetail.aspx 的页面效果如图 9-14 所示。

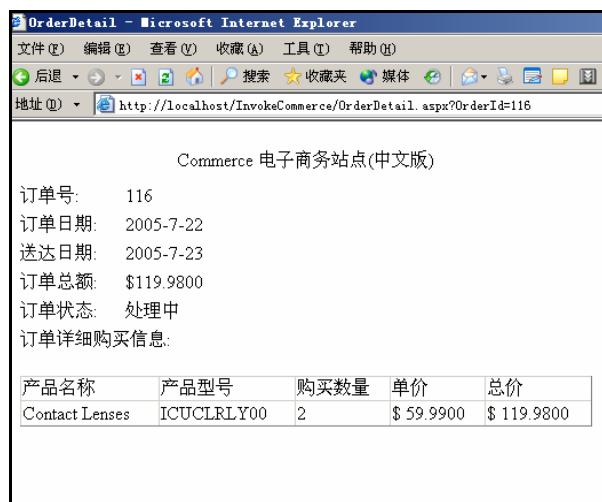


图 9-14 查看订单详细信息页面 OrderDetail.aspx

9.5 开发启示

本章中讲解了扩展 ASP.NET Commerce Starter Kit 方面的内容，目的是希望读者能够通过对本章的学习了解到如何以 ASP.NET Commerce Starter Kit 为基础进行二次开发，完成一个功能完善的电子商务 Web 应用程序。

本章中对 ASP.NET Commerce Starter Kit 的扩展不仅包括对 ASP.NET Commerce Starter Kit 本身功能方面的扩展，也包括开发了一个调用 ASP.NET Commerce Starter Kit 提供的 Web Service 来完成业务逻辑的第三方 Web 应用程序。通过对部分功能的增加和改进，启发读者更多的关于 ASP.NET Commerce Starter Kit 二次开发方面的思路。

第二部分

Time Tracker Starter Kit——项目与工作管理系统

案例简介

TimeTracker Starter Kit 是一个以公司或组织机构内部的项目与工作进程管理为背景的 Web 应用程序，其中包括项目管理、工作进程管理、数据报表以及移动 Web 应用几部分。TimeTracker Starter Kit 在技术层面相对来讲比较复杂，因此希望读者能够认真细致的对其代码进行分析。

知识点提要

TimeTracker Starter Kit 的重点知识点如下。

- DataGrid 控件的高级应用
- GDI+技术
- 移动 Web 应用
- 对象集合类

资源下载

读者可到微软 ASP.NET 官方网站下载源代码，地址是：

<http://www.asp.net/StarterKits/DownloadTimeTracker.aspx?tabindex=0&tabid=1>

第 10 章

TimeTracker 简介

10.1 TimeTracker 概况

ASP.NET Time Tracker Starter Kit 是 ASP.NET Starter Kit 中的一个示例程序，是一个项目时间管理方面的 Web 应用程序。和其他的 ASP.NET Starter Kit 示例程序一样，ASP.NET Time Tracker Starter Kit 并不是以商业应用为目的，而是作为一个示例程序演示 ASP.NET 技术的新特性和应用 ASP.NET 开发 Web 应用程序的步骤和细节。它的主界面如图 10-1 所示。

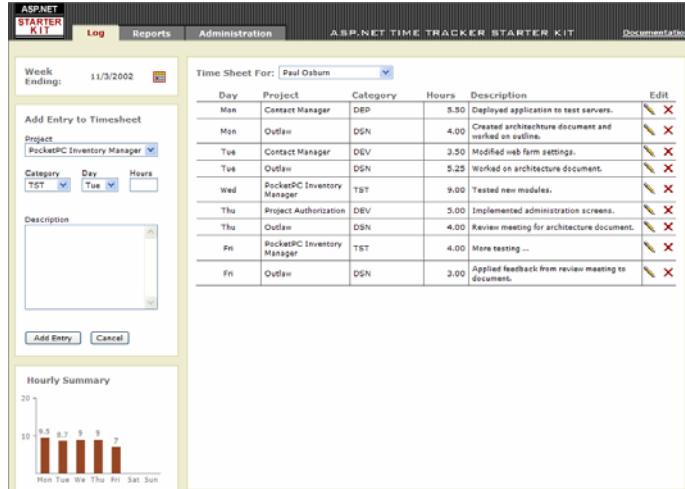


图 10-1 ASP.NET Time Tracker Starter Kit 的主界面

10.1.1 TimeTracker 的开发背景

ASP.NET Time Tracker Starter Kit 是以项目管理中的时间分配和时间统计为需求，为项目经理和项目成员提供一个动态管理

项目进程、图形化和报表化统计项目进程的 Web 应用程序。系统管理员应用 ASP.NET Time Tracker Starter Kit 可以建立公司内部的项目管理站点，公司的项目经理可以通过项目管理站点管理项目的进程和划分项目内部的工作，项目开发人员可以安排自己的工作时间，最终可以形成项目的工作进度报表和每一个项目开发人员的工作量图表，供项目经理和项目开发人员查看。

ASP.NET Time Tracker Starter Kit 并不是以商业应用为目的，但是 ASP.NET Time Tracker Starter Kit 是一个完整的项目管理应用程序，实现了项目管理中的基本功能，可以满足中小型企业内部对于项目管理方面的需求。ASP.NET Time Tracker Starter Kit 的另外一个目的是演示 ASP.NET 技术中新的特性，作为一个教程式的示例演示，使用 ASP.NET 开发 Web 应用程序的总体架构模型与具体实现形式，启迪学习 ASP.NET Time Tracker Starter Kit 的开发者的开发思路。

根据企业的实际需求，ASP.NET Time Tracker Starter Kit 提供基于移动设备的 Web 应用接口。在项目的实施过程中，项目经理和项目开发人员并不可能每天都在计算机上使用项目管理站点获取项目信息和安排自己的工作，所以基于移动设备的应用是十分重要的。方便项目组成员随时随地地访问项目管理站点，查看、管理和安排项目进度。

项目报表是项目进度最直观的体现，ASP.NET Time Tracker Starter Kit 提供了两种形式的报表——基于页面形式的报表和基于图片形式的报表。基于页面形式的报表是在 Web 页面上对数据进行分析和汇总，基于图片形式的报表则根据数据动态地生成图片报表，使报表更加直观，表现力更强。

10.1.2 系统功能

ASP.NET Time Tracker Starter Kit 的主要功能包括如下几部分。

(1) 项目管理

系统管理员可以在项目管理模块新建项目和为每一个项目分配项目经理和项目开发人员。项目参与人员可以根据实际工作情况安排工作时间并记录到系统中，项目经理可以参看项目成员的工作量统计和整个项目的工作量统计报告。

(2) 数据报表

数据报表模块的主要功能是统计项目进程的相关数据和项目参与人员的个人工作量，以表格或是图片的形式展现给系统用户。

(3) 人员管理

只有系统管理员才有权限使用人员管理模块，该模块的主要功能包括系统用户的建立，系统用户资料的维护。

(4) 移动设备应用

移动设备应用是专门为移动设备开发的应用站点，该站点并未包含 Web 站点的全部功能，但是用户可以通过移动设备应用、安排和参看项目进程。

10.1.3 版本信息

根据使用的开发工具和开发语言不同，ASP.NET Time Tracker Starter Kit 分为 6 个不同的版本。表 10-1 是各个版本的信息。

表 10-1 ASP.NET Time Tracker Starter Kit 的 6 个版本信息

| 版 本 号 | 使 用 语 言 | 推 荐 Web 服 务 器 | 备 注 |
|-----------------------------------|---------|---------------|-----|
| Time Tracker Starter Kit (VB SDK) | VB.NET | Web Matrix | |
| Time Tracker Starter Kit (CS SDK) | C# | Web Matrix | |

续表

| 版 本 号 | 使 用 语 言 | 推 荐 Web 服 务 器 | 备 注 |
|-----------------------------------|---------|---------------|-------------------------|
| Time Tracker Starter Kit (JS SDK) | J# | Web Matrix | 需要.NET Framework 1.1 支持 |
| Time Tracker Starter Kit (VB VS) | VB.NET | IIS 6.0 | |
| Time Tracker Starter Kit (CS VS) | C# | IIS 6.0 | |
| Time Tracker Starter Kit (JS VS) | J# | IIS 6.0 | 需要.NET Framework 1.1 支持 |

注意：因为本书讲解的是使用 C#语言开发 ASP.NET Web 应用程序，并且采用 VS.NET 作为模拟开发环境，所以采用 Time Tracker Starter Kit (CS VS) 版本作为本书的示例版本。后面章节提到 ASP.NET Time Tracker Starter Kit 如不做特殊说明，均为该版本。

10.2 TimeTracker 的安装

本节将对 ASP.NET Time Tracker Starter Kit 的安装步骤进行详细讲解，介绍在安装过程中可能遇到的问题。读者可以从 <http://www.asp.net> 获得 ASP.NET Time Tracker Starter Kit 的安装程序，下载得到的是一个名字为 ASP.NET TimeTracker (CSV) Installer v1.0.msi 的 MSI 安装包文件。

10.2.1 推荐环境

配合本书中讲解的示例程序，推荐采用如下的软件配置环境。

● Windows Server 2003 Enterprise

从理论上讲，任何服务器版本的 Windows 操作系统都可以安装并运行 ASP.NET Time Tracker Starter Kit，但是因为 Windows Server 2003 是和.NET Framework 整合最紧密的服务器版本的操作系统，所以推荐使用 Windows Server 2003 Enterprise 作为服务器的操作系统。

● .NET Framework 1.1

.NET Framework 目前有两个正式版本，1.0 版和 1.1 版。.NET Framework 1.1 版在修正了 1.0 版很多不足的基础上推出了很多新的 API。虽然这些新推出的 API 并不是必需的，但还是建议使用.NET Framework 1.1 版。

● Internet Information Server (IIS) 6.0

如果采用 Windows Server 2003 作为服务器操作系统，那么 IIS 6.0 就是系统中默认的 Web 服务器。

● Microsoft SQL Server 2000

作为 Microsoft SQL Server 2000 的简化版本，Microsoft SQL Server Desktop Engine 2000 虽然也能满足系统对数据库方面的要求，但是考虑到数据安全性方面的因素，建议采用 Microsoft SQL Server 2000 数据库服务器。

● Visual Studio.NET 2003

ASP.NET Time Tracker Starter Kit 是完全开放源代码的，在本书的讲解中需要查看项目中的源代码，并要做一些简单的改动，推荐使用 Visual Studio.NET 2003 作为系统的开发环境。

10.2.2 安装步骤

(1) 单击安装程序 MSI 文件，进入欢迎安装的界面，如图 10-2 所示。界面中显示了安装时应注意的

内容。

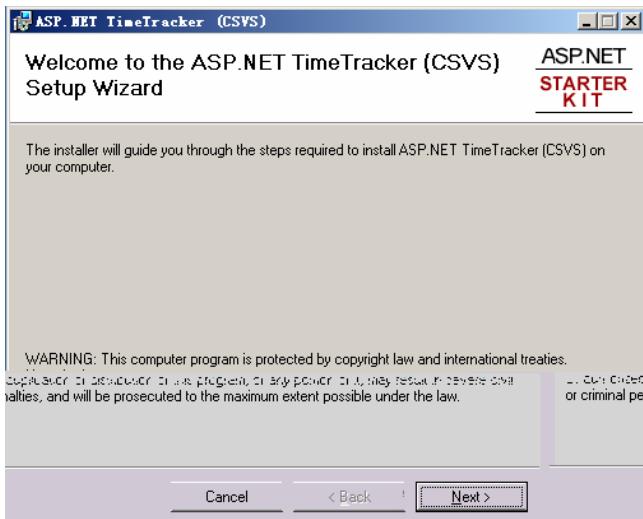


图 10-2 ASP.NET Time Tracker Starter Kit 安装步骤 1

(2) 单击 Next 按钮，打开安装信息页面，如图 10-3 所示，该页面中列出了系统所需的运行环境的信息。

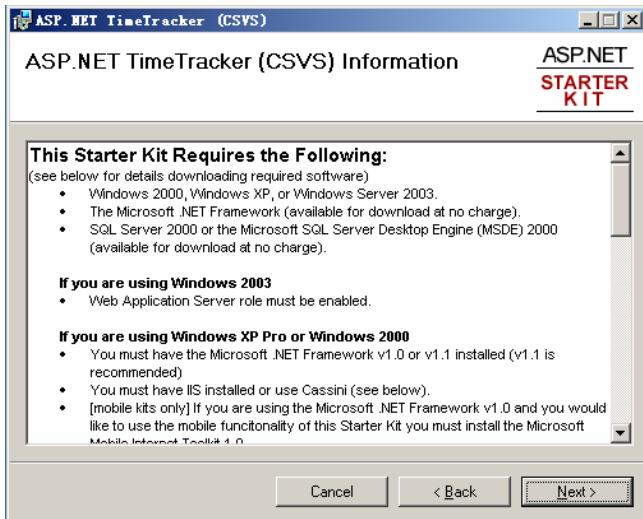


图 10-3 ASP.NET Time Tracker Starter Kit 安装步骤 2

(3) 单击 Next 按钮，打开安装许可页面，该页面显示的是安装许可协议，安装协议中声明了软件使用权力、用户使用中需要遵守的约定以及 ASP.NET Time Tracker Starter Kit 的版权信息等。用户在阅读完安装许可协议之后可以选择接受协议或是拒绝协议，接受协议选中 I Agree 才可以进行下一步安装。如图 10-4 所示。

(4) 单击 Next 按钮，进入 IIS 虚拟目录配置页面，如图 10-5 所示。在该页面中安装程序询问用户是否需要支持 IIS 的虚拟目录，如果使用其他 Web 服务器，可以选择不支持 IIS 虚拟目录。

(5) 单击 Next 按钮，打开 SQL Server 服务器配置页面，如图 10-6 所示。在该页面中安装程序会询问用户 SQL Server 是安装在本地服务器还是远程服务器上。

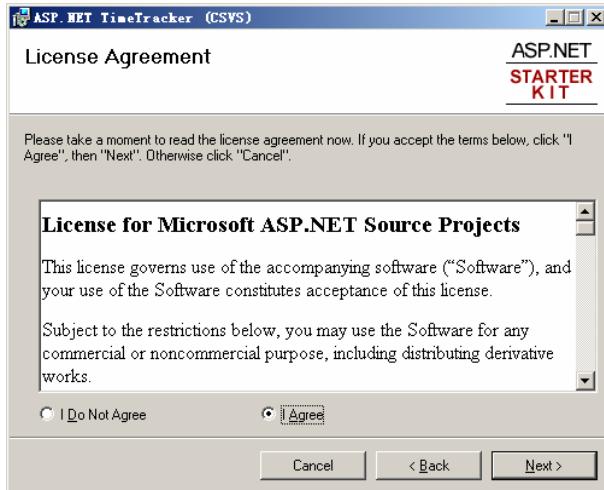


图 10-4 ASP.NET Time Tracker Starter Kit 安装步骤 3



图 10-5 ASP.NET Time Tracker Starter Kit 安装步骤 4

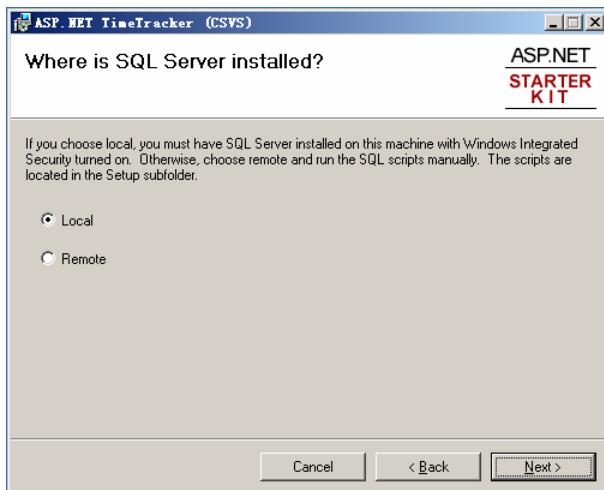


图 10-6 ASP.NET Time Tracker Starter Kit 安装步骤 5

(6) 单击 Next 按钮，打开安装路径选择页面，如图 10-7 所示，在该页面中安装程序会询问程序安装的主目录，用户可以任意选择安装目录。另外安装程序还将询问应用程序的使用权限，用户可以将应用程序的使用权限设置为当前用户或是本地所有用户。

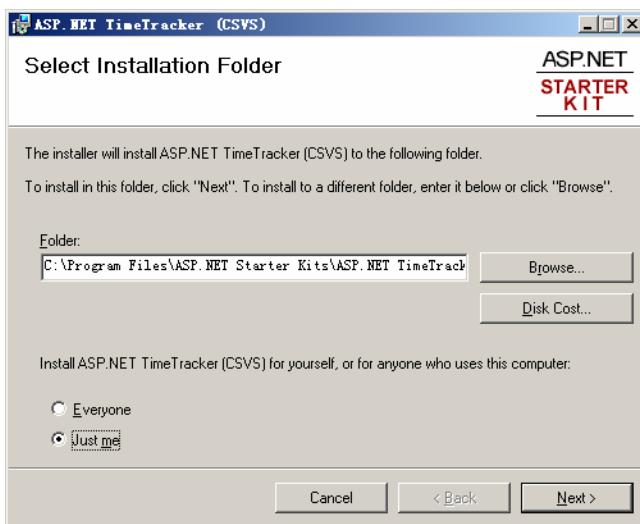


图 10-7 ASP.NET Time Tracker Starter Kit 安装步骤 6

(7)单击 Next 按钮，打开安装确认页面，如图 10-8 所示。该页面询问用户是否确定要安装 ASP.NET Time Tracker Starter Kit，如果想修改前面配置的安装信息，可以单击 Back 按钮进行修改。

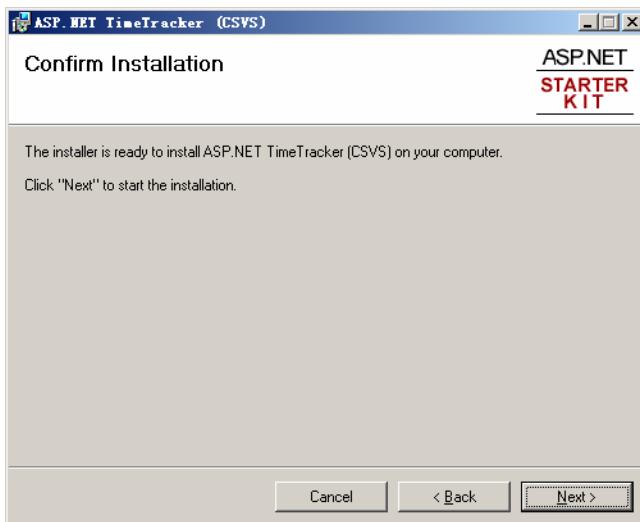


图 10-8 ASP.NET Time Tracker Starter Kit 安装步骤 7

(8) 单击 Next 按钮，打开安装进程画面，如图 10-9 所示。安装进程画面会显示应用程序当前的安装进度。

(9) 在安装的过程中会打开数据库选择页面，如图 10-10 所示。用户可以选择系统使用的 SQL Server 数据库，然后单击 Install 按钮即可安装数据库。Time Tracker 的默认数据库名称为 Timetracker。

(10) 安装结束会打开安装成功页面和系统配置信息页面，如图 10-11 所示。

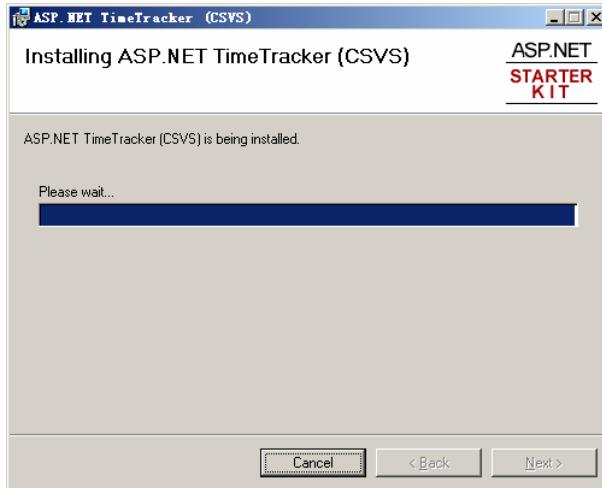


图 10-9 ASP.NET Time Tracker Starter Kit 安装步骤 8



图 10-10 ASP.NET Time Tracker Starter Kit 安装步骤 9

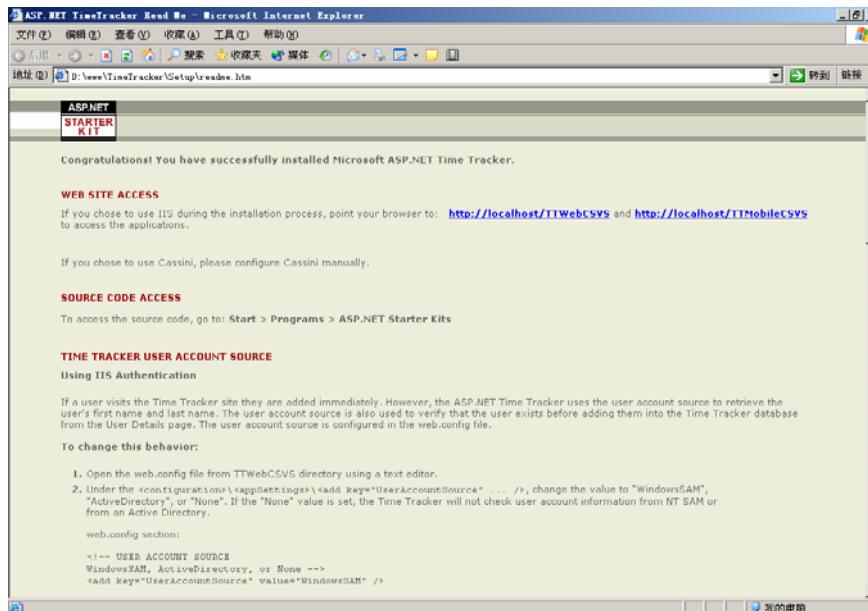


图 10-11 ASP.NET Time Tracker Starter Kit 安装成功信息页面

10.2.3 配置说明

除了使用应用程序进行安装之外，也可以手工完成 ASP.NET Time Tracker Starter Kit 的部署，手工复制应用程序到安装程序主目录，建立数据库，并配置数据库连接。本节所讲解的配置方法有可能在手工部署的时候使用。

(1) 配置数据库连接信息

在 Visual Studio.NET 中打开 ASP.NET Time Tracker Starter Kit 的项目，找到 Web 应用程序和移动设备应用的 Web.config 文件，配置 appSettings 节点下的 ConnectionString 字符串为如下格式。

```
<add key="ConnectionString" value="server=servername;uid=sqluser;pwd=passwd;database=TimeTracker" />
```

(2) 配置注册用户的默认身份

在 Visual Studio.NET 中打开 ASP.NET Time Tracker Starter Kit 的项目，找到 Web 应用程序和移动设备应用的 Web.config 文件，配置 appSettings 节点下的 DefaultRoleForNewUser 字符串，设置为 1 则注册用户的默认身份为管理员，设置为 2 则注册用户的默认身份为项目经理，设置为 3 则注册用户的默认身份为项目开发人员。

(3) 配置用户信息源

在 Visual Studio.NET 中打开 ASP.NET Time Tracker Starter Kit 的项目，找到 Web 应用程序和移动设备应用的 Web.config 文件，配置 appSettings 节点下的 UserAccountSource 字符串，设置为 WindowsSAM 是从 Windows SAM 中取得用户身份信息，设置为 ActiveDirectory 则从活动目录服务器取得用户身份信息，设置为 None 则表明没有用户身份信息源。

(4) 设定项目中星期的起始日期

在 Visual Studio.NET 中打开 ASP.NET Time Tracker Starter Kit 的项目，找到 Web 应用程序和移动设备应用的 Web.config 文件，配置 appSettings 节点下的 FirstDayOfWeek 字符串。可以设置为用 0~6 七个数字代表星期日到星期六的 6 个日期。

(5) 中文系统兼容性问题

ASP.NET Time Tracker Starter Kit 是一个全球化的应用程序，支持不同语言的操作系统，在页面浏览和信息输入等多个方面都支持不同的语言编码，但是在动态产生的图表中仅支持英文字符，因此在中文操作系统下动态产生图表会遇到因为使用中文字符而无法正常显示的问题。一个比较好的解决方案就是更改 Global.asax 文件中的默认语言选择部分的代码。在 Visual Studio.NET 中打开 ASP.NET Time Tracker Starter Kit 的项目，找到 Web 应用程序的 Global.asax.cs 文件中的 Application_BeginRequest 方法，其代码如下。

```
protected void Application_BeginRequest(Object sender, EventArgs e)
{
    try
    {
        if (Request.UserLanguages != null)
            Thread.CurrentThread.CurrentCulture =
                CultureInfo.CreateSpecificCulture(Request.UserLanguages[0]);
        else
            Thread.CurrentThread.CurrentCulture = new CultureInfo("en-us");
            Thread.CurrentThread.CurrentUICulture = Thread.CurrentThread.CurrentCulture;
    }
    catch (Exception ex)
    {
        Thread.CurrentThread.CurrentCulture = new CultureInfo("en-us");
    }
}
```

```
    }  
}
```

将上述代码更改如下。

```
protected void Application_BeginRequest(Object sender, EventArgs e)  
{  
    try  
    {  
        Thread.CurrentThread.CurrentCulture = new CultureInfo("en-us");  
        Thread.CurrentThread.CurrentUICulture = Thread.CurrentThread.CurrentCulture;  
    }  
    catch (Exception ex)  
    {  
        Thread.CurrentThread.CurrentCulture = new CultureInfo("en-us");  
    }  
}
```

使用如上的更改可以避免动态图表产生的模块因为系统的语言环境问题而产生错误。

10.3 技术点概述

在 ASP.NET Time Tracker Starter Kit 中使用了 ASP.NET 中的新特性和新技术，本书将对如下技术点进行详细的分析和讲解。

- 三层应用程序架构
- 使用 GDI+产生动态图表
- .NET 框架下移动设备应用开发技术
- 用户身份验证以及权限分配策略
- DataGrid 控件的高级应用
- .NET 应用程序中的数据访问 blocks 技术

ASP.NET Time Tracker Starter Kit 中采用了典型的三层应用程序架构。和 ASP.NET Commerce Starter Kit 不同的是，ASP.NET Time Tracker Starter Kit 独立抽象了商务逻辑层，前台的用户表示层和封装有业务逻辑的商务层之间采用了轻量级引用的模式，从而使得系统的技术框架变得轻巧。

GDI+是.NET Framework 提供的一套图形操作的 API 接口，调用 GDI+提供的 API 开发者可以在.NET Framework 下完成对图形的一系列操作。

在.NET Framework 下，移动设备应用的开发和普通应用程序开发有很大的不同，因为移动设备应用开发有专门的控件和 API，本书将对移动设备应用开发做详细的讲解。



第 11 章

需求分析与 系统架构

11.1 系统功能

ASP.NET Time Tracker Starter Kit 的主要功能是针对小型组织结构内部的项目管理和工作进程管理的。和 ASP.NET Commerce Starter Kit 一样，设计者在设计 ASP.NET Time Tracker Starter Kit 的时候并没有将实际的商业应用作为直接目的，单从目前的 ASP.NET Time Tracker Starter Kit 的功能上看，虽然可以满足简单的项目管理和工作进程管理的需求，但是和实际的商业应用还有一段距离。开发者希望将 ASP.NET Time Tracker Starter Kit 设计成为一个基于 B/S 结构的项目管理和工作进程管理应用程序框架，同时也能从技术层面体现 ASP.NET 中新的发展理念和开发思路，开拓读者的开发视野，给使用面向对象技术开发 Web 应用程序的开发者以更多的启示。

11.1.1 需求分析

从直观的功能划分上讲 ASP.NET Time Tracker Starter Kit 主要分成四大模块。

项目管理和工作进程管理是核心模块。

(1) 项目管理

在很多企业中，日常的工作基本上都是围绕着工作中一个又一个的工程项目展开的，只是每个公司因为其性质不同所开发的项目也不相同。例如软件公司开发的是软件项目，建筑公司开发的建筑项目。但是不同公司的不同项目在项目管理方面有着很多相似的地方：项目本身有着共同的属性；开发项目的流程从某种意义来讲有很大程度的通用性；公司的职员在项目中担任的角色基本上可以划分成相同的几个类别；项目开发中或结束之后对项目的数据总结所体现的数据很类似等等。项目这个概念好比面向对象思想中的基类，不同的公司开发的项目模型都是对这个基类的继承，而真正工程中的项目就是对项目模型的实例化。

正是基于以上考虑，ASP.NET Time Tracker Starter Kit 的设计者力求将系统中的项目模型设计得具有很强的通用性，这个项目模型可以被多数的公司或机构所使用。因此项目对象应该包含项目名称、项目负责人（项目经理）、项目开发人员、项目开始时间、项目整体工作量、项目中的工作分类等属性。系统同时也要提供对项目的新增、修改、删除等操作，但实际上这些操作并不是每一个系统用户都有权限的。

(2) 工作进程管理

工作进程就是日常工作中的每项工作，公司或组织机构中包括领导者、管理者、普通职员在内所有人员的日常工作，实际上就是围绕着一个又一个的工作进程展开的，而每一个工作进程基本上都是属于某一个项目下的。例如一个软件公司中的一个软件项目，将客户需求分析、设计系统框架和实际开发代码 3 部分工作交给 3 个不同的人去做，那么每个人都有自己的工作进程，而这 3 个工作进程属于同一个项目。在实际的项目实施过程中，每一个项目都会有若干个工作进程，而且可能会出现多个工作进程都是在完成同一部分或同一个类型的工作，即项目的工作分类。所以每一个工作进程都有一个工作分类与之对应。

工作进程的设计也偏重于通用性，工作进程对象一般具有包括工作进程描述、工作进程名称、工作进程的执行者、工作进程所属项目、工作进程所属分类、工作进程的工作量（一般以时间为单位）、工作进程执行日期等内容。系统中每一个用户都可以安排自己的工作进程，而管理员和项目经理也可以为其他的项目开发人员安排工作进程。

除了项目管理和工作进程管理之外，系统还应当具有数据统计和用户管理功能。

(3) 数据统计

在工作中，项目的管理人员和公司的管理人员都在时刻监督着项目的进行。管理者最关心的莫过于项目总体进程、每一个项目开发人员的工作进展情况以及工作量情况、项目中的每一部分的详细进展情况等等。不仅是在项目的实施过程中，即使在项目结束之后这些数据对管理者的意义也是十分重要的。因此系统有必要提供相关的数据统计功能，方便管理者查看和统计项目的进展情况。

除了管理者之外，每一个项目开发人员也很关心自己的工作量情况。包括个人所完成的所有工作进程的数据统计等。因此系统的数据统计部分也应该提供相应的功能。

对于数据的统计，最通用的方式应该是以数据报表形式展现给使用者。数据统计部分不仅能显示所统计的数据，而且要清晰、直观地显示用户所要查看的数据，因此系统中的数据统计部分采用数据报表的形式展示显得很有必要。

(4) 用户管理

系统的管理人员应该可以对系统中的用户信息进行简单的维护，毕竟 ASP.NET Time Tracker Starter Kit 面向的是公司或组织结构内部的成员。

11.1.2 功能设计

根据 11.1.1 节中提出的需求分析，系统在实际功能设计上应该分为系统管理、工作进程管理和数据报表 3 个部分。

● 系统管理

系统管理中包括项目管理和用户管理。项目管理包括新建项目、修改项目和删除项目。用户管理包括新增用户、修改用户。

● 工作进程管理

工作进程管理包括工作进程的查询、添加、修改和删除。

● 数据报表

数据报表部分包括项目的数据报表和人员的数据报表。在项目的数据报表部分可以查看每一个项目中每一个项目分类的数据统计情况以及所有的工作进程情况。在人员数据报表中可以查看人员参与的项目数据以及人员的工作进程数据。

考虑到需求中对角色的描述和分析，系统中的角色设置情况如下。

● 普通用户

最基本的系统用户，相当于实际公司中的普通职员。可以新增、修改和查看本人的工作进程。

● 项目经理

项目的负责人，除了具有普通用户的基本权限之外，一般还具有新建项目、查看和管理所负责项目中普通用户的工作进程等权限。

● 管理员

具有系统的所有权限。

11.2 UML图

11.2.1 类图

由于ASP.NET Time Tracker Starter Kit中类的设计比较复杂，因此本书在UML图部分将以类图为重点，希望能够通过对ASP.NET Time Tracker Starter Kit类图部分的讲解让读者学习到设计者所要传递的面向对象的设计思想。

(1) 用户类User和身份验证类Security

如图11-1所示。

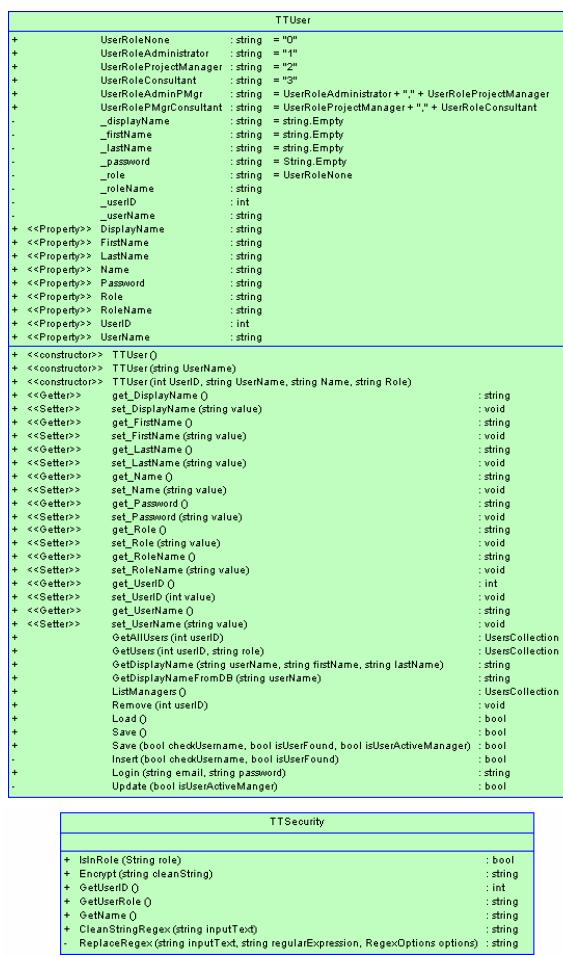


图11-1 用户类User和身份验证类Security类图

(2) 项目类 Project

如图 11-2 所示。

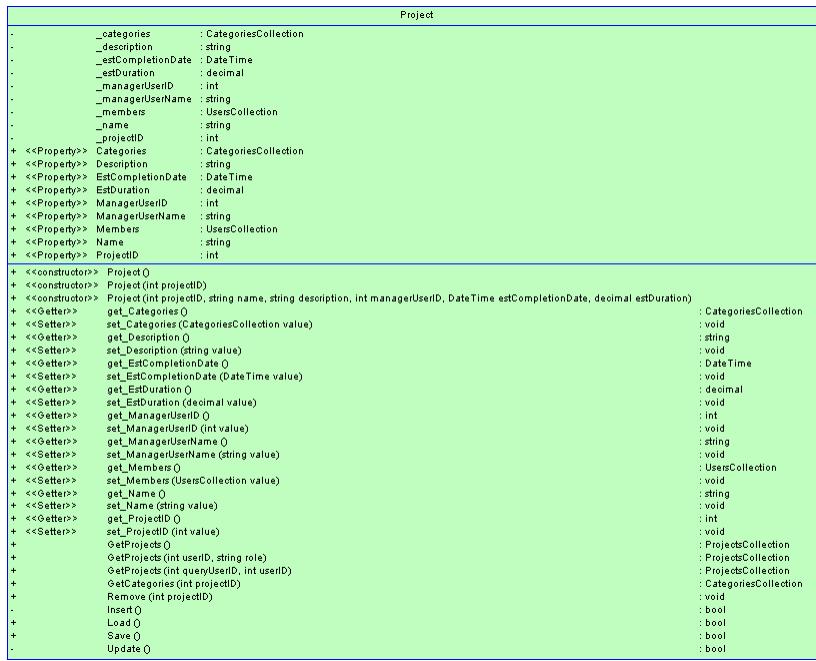


图 11-2 项目类 Project 类图

(3) 工作进程类 TimeEntry

如图 11-3 所示。

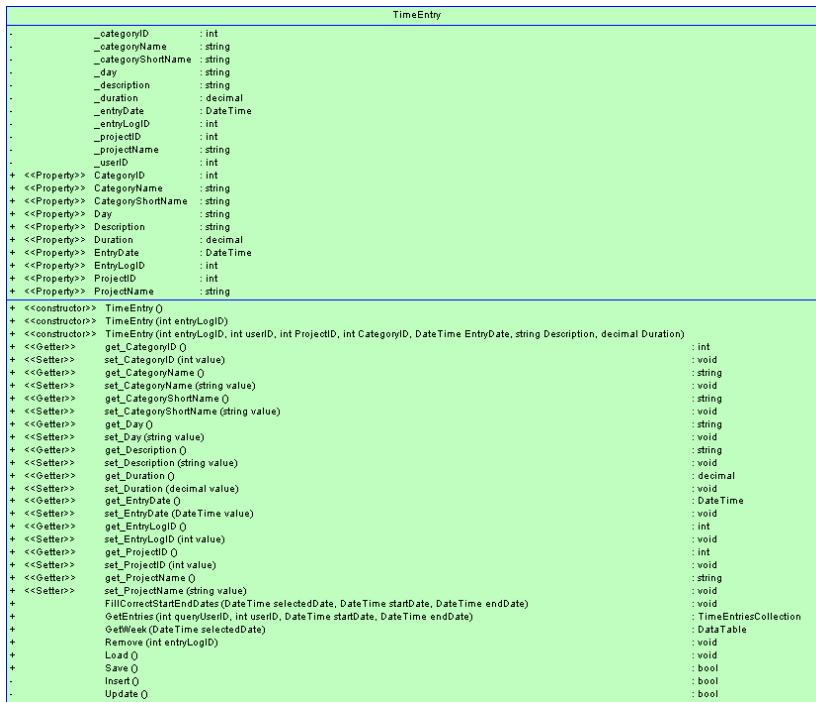


图 11-3 工作进程类 TimeEntry 类图

11.2.2 活动图

因为 ASP.NET Time Tracker Starter Kit 并不是为一个完整的流程服务的，系统的多数功能相互之间的贯穿性不是很强，所以用活动图并不能完全地展示系统中功能模块之间的联系。本节中选取新建项目这一活动作为演示，图 11-4 为新建项目活动图。

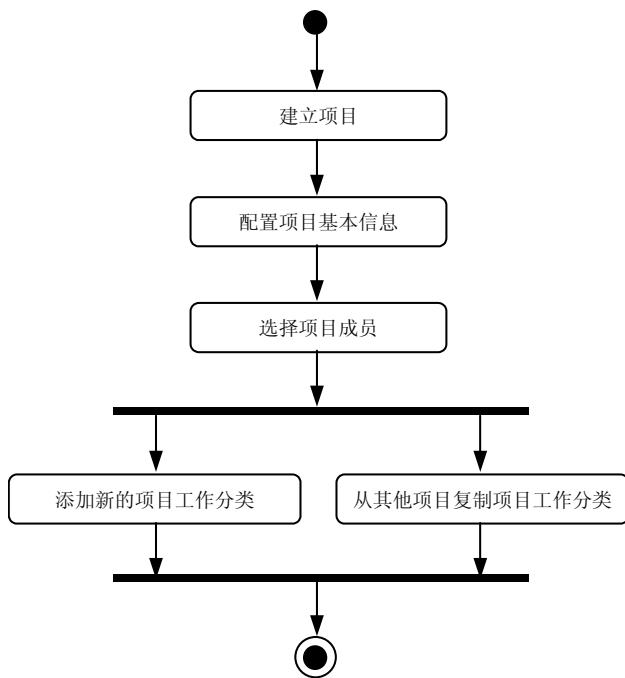


图 11-4 新建项目活动图

11.3 系统架构

ASP.NET Time Tracker Starter Kit 在架构方面分为两个部分：普通 Web 应用和专门为移动用户开发的移动 Web 应用。两部分应用程序实际上是系统的两部分不同的表征，用户可以使用 PC 访问普通 Web 应用站点进行管理工作进程等操作，也可以使用移动设备访问移动 Web 应用站点。两部分应用在功能上并不完全一样，移动 Web 应用提供的所有功能仅仅是普通 Web 应用提供的一小部分，因此两部分应用在逻辑上有着共同性，在设计上两部分应用可以共用逻辑层。其结构如图 11-5 所示。

11.4 数据库设计

ASP.NET Time Tracker Starter Kit 的数据库在结构设计上并不复杂，图 11-6 是 ASP.NET Time Tracker Starter Kit 的数据库模型图。

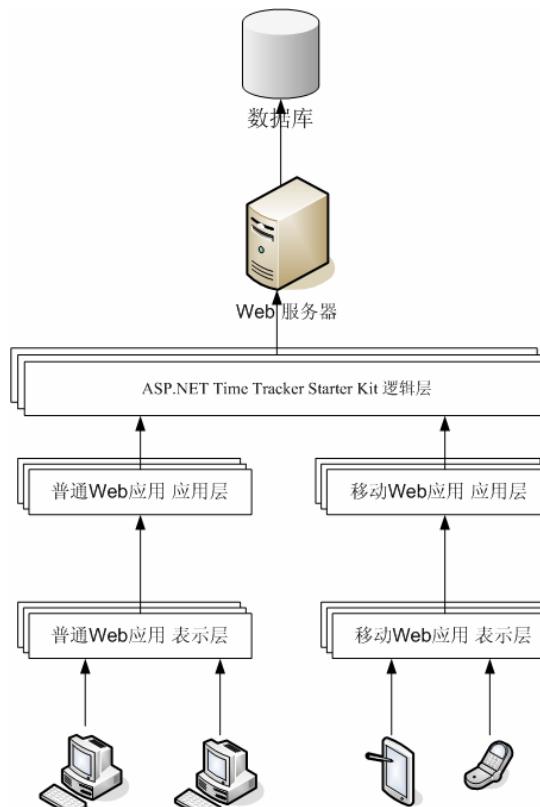


图 11-5 ASP.NET Time Tracker Starter Kit 系统架构

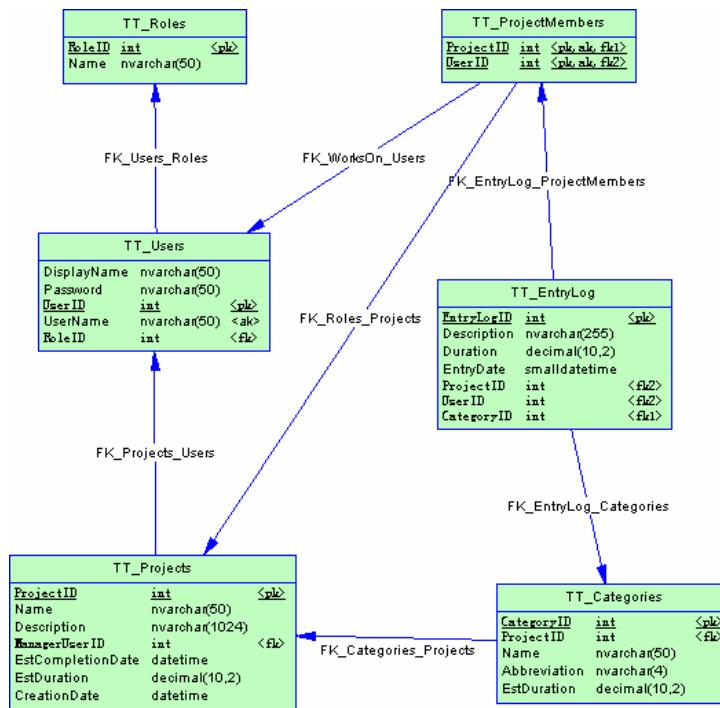


图 11-6 ASP.NET Time Tracker Starter Kit 数据库模型图

11.5 设计启示

在设计上，ASP.NET Time Tracker Starter Kit 将核心业务处理主要放在应用层和存储过程中，这样减轻了表示层和逻辑层的负担，也使得逻辑层变得轻巧并具有更好的可扩展性和可重用性，这种设计模式主要还是取决于 ASP.NET Time Tracker Starter Kit 的架构，因为要同时为不同的 Web 应用提供同一个逻辑层，要求逻辑层具有较高的可重用性，实际上这也是在设计中使用面向对象思想的一种体现。

第 12 章

技术点解析

本章将详细分析 ASP.NET Time Tracker Starter Kit 中比较突出的几个技术点，其中包括 DataGrid 的高级应用、对象集合类的使用、GDI+技术和移动 Web 应用。这些技术点要比第 3 章中讲解的 ASP.NET Commerce Starter Kit 的技术点在应用难度上有大幅提升。本章所讲解的技术点更能体现出 ASP.NET 技术的

特点和强大之处。

12.1 DataGrid 的高级应用

本书第 6 章中已经讲解了 DataGrid 控件的一些基础应用，但是 DataGrid 作为 ASP.NET 中最强大的数据展现控件，第 6 章中并没有介绍其全部的功能和特点，本节中将继续讲解 DataGrid 控件的高级应用。

12.1.1 DataGrid 的事件

在概念上可以把 DataGrid 控件理解成是 HTML 里面的一个 Table，普通的 HTML Table 是由行 tr 和列 td 组成，而 DataGrid 的行实际上是 DataGrid 的 Item 对象，而列则是 DataGrid 的 Column 对象。在 DataGrid 控件的属性集中可以对 Items 和 Columns 属性进行读取和设置。在 ASP.NET 发送到客户端的 HTML 页的源代码中，DataGrid 控件的表现形式也就是 Table，而 DataGrid 控件的行集合 Items 和 Columns 也被解释成为 HTML 的 tr 和 td 标签。

DataGrid 控件之所以具有强大的数据展现功能是因为其具有功能比较完备的事件机制，DataGrid 控件支持的一些常用事件列表如表 12-1 所示。

表 12-1 DataGrid 控件支持的一些常用事件列表

| 事件名 | 触发条件 |
|---------------|-----------------------------------|
| CancelCommand | 对 DataGrid 控件中的某一行单击 Cancel 按钮时发生 |
| DataBinding | 当 DataGrid 控件绑定到数据源时发生 |
| DeleteCommand | 对 DataGrid 控件中的某一行单击 Delete 按钮时发生 |
| EditCommand | 对 DataGrid 控件中的某一行单击 Edit 按钮时发生 |

续表

| 事件名 | 触发条件 |
|----------------------|-----------------------------------|
| ItemCommand | 当单击 DataGrid 控件中的任一按钮时发生 |
| ItemCreated | 当在 DataGrid 控件中创建项时在服务器上发生 |
| ItemDataBound | 在项被数据绑定到 DataGrid 控件后发生 |
| PageIndexChanged | 当单击 DataGrid 控件分页选择按钮之一时发生 |
| SelectedIndexChanged | 在两次服务器发送之间，在数据列表控件中选择了不同的项时发生 |
| SortCommand | 对列进行排序时发生 |
| UpdateCommand | 对 DataGrid 控件中的某一行单击 Update 按钮时发生 |

12.1.2 常用的 DataGrid 事件

DataGrid 在创建行时首先触发 ItemCreated 事件，每创建完任何一行，无论新创建的行是普通的数据行，还是 EditItem 或 SelectedItem，都会触发 ItemCreated 事件，同样通过 ItemCreated 事件，也可以定制 DataGrid 的任何类型的行。在该事件中可以定制 Header(标题行)、Pager(分页导航行)以及 Footer(脚注行)。执行完 ItemCreated 事件后，DataGrid 的行的模型就已经被建立，该行中无数据绑定的列也已经被建立。接着要对需要进行数据绑定的列进行数据绑定，也就是把数据源的值赋到各个列中，也包括执行 aspx 页面的数据绑定表达式，表达式的格式类似于<%# 表达式 %>。当 DataGrid 的行完成数据绑定后触发 ItemDataBound 事件，在 ItemDataBound 事件，提供了访问 DataGrid 数据源的机会。在 ItemDataBound 事件中一般都是对 DataGrid 的模板列进行定制，包括指定模板列中的自定义控件的数据源或属性等。在 ItemCreated 事件和 ItemDataBound 事件，可以通过参数表中的参数 DataGridItemEventArgs 访问到当前行，并对其进行操作，例如参数 DataGridItemEventArgs 为 e，则可以通过 e.Item.ItemIndex 访问到当前行的 ItemIndex。根据数据源中绑定到当前行的数据，可以定制或改变 DataGrid 当前行和当前行中的任意列的样式，例如参数 DataGridItemEventArgs 为 e，则可以通过 e.Item.Cells[0] 访问到当前行的第一列。

在使用 DataGrid 控件的 ItemCreated 和 ItemDataBound 事件时，要注意以下两点。

- 只有在创建 Item 行、AlternatingItem 行、SelectedItem 行、Edititem 行时 e.Item.DataItem 才有值；创建其他类型的行时，e.Item.DataItem 的值都为 null。所以我们在处理 e.Item.DataItem 时必须先判断 e.Item.ItemType 属性为上述类型行时，才能取出数据源的值；否则会抛出“对象不能为 null”的异常。
- 创建 Pager 行时并不会触发 ItemDataBound 事件，因为 Pager 行不进行数据绑定。

执行完 ItemDataBound 事件后，一行的创建就完成了。DataGrid 的 Items.Count 属性在每一行创建结束之后累加 1，表示这一行创建完毕，将新创建的行加入到 DataGrid.Items 集合中。Datagrid 控件循环地调用 ItemCreated 和 ItemDataBound 事件创建完所有的行后，就完成了 DataGrid 的创建工作。也就是说，执行完 DataBinding 事件后（ItemCreated 和 ItemDataBound 是嵌套在 DataBinding 事件里执行的），整个 DataGrid 就创建完毕。如果在 DataGrid 创建结束之后再访问 DataGrid 里面的数据或内容，就只有通过 DataGird 的 Items 集合访问行数据，因此 DataGrid 的 ItemCreated 和 ItemDataBound 事件仅仅当每行被创建的时候才发生。在 DataGrid 创建结束之后，也可以通过 DataGird 中 Item 项的 Cells 集合访问 DataGrid 的某一行中的列数据或列中包含的控件。如果此时通过 DataGirdItem.DataItem 属性访问 DataGrid 绑定到该行的数据源，就会抛出“找不到对象或对象为 null”的异常。因为在执行完 ItemDataBound 事件后 DataGridItem.DataItem 属性已经被置为 null 了。

下面一段代码展示了如何应用 ItemDataBound 事件。

```
// CreateDataSource 接口是为 DataGrid 控件临时定义的数据源
```

```
ICollection CreateDataSource()
{
    DataTable dt = new DataTable();
    DataRow dr;

    // 该数据源中包含 3 列
    dt.Columns.Add(new DataColumn("IntegerValue", typeof(Int32)));
    dt.Columns.Add(new DataColumn("StringValue", typeof(String)));
    dt.Columns.Add(new DataColumn("CurrencyValue", typeof(Double)));

    //产生数据源的数据
    for (int i=0; i<=10; i++)
    {
        dr = dt.NewRow();
        dr[0] = i;
        dr[1] = "Item " + i.ToString();
        dr[2] = 1.23 * (i + 1);
        dt.Rows.Add(dr);
    }

    DataView dv = new DataView(dt);
    return dv;
}

//页面载入方法
void Page_Load(object sender, EventArgs e)
{
    //为名为 ItemsGrid 的 DataGrid 定义 ItemDataBound 事件
    ItemsGrid.ItemDataBound += new DataGridItemEventHandler(this.Item_Bound);
    //为 DataGrid 空间绑定数据源
    if (!IsPostBack)
    {
        ItemsGrid.DataSource = CreateDataSource();
        ItemsGrid.DataBind();
    }
}

//ItemsGrid 的 ItemDataBound 事件
void Item_Bound(object sender, DataGridItemEventArgs e)
{
    //判断当前行的类型
    if((e.Item.ItemType == ListItemType.Item) || (e.Item.ItemType == ListItemType.AlternatingItem))
    {
        // 从数据源中获取数据并对其进行操作
        Double Price = Convert.ToDouble(e.Item.Cells[2].Text);
        // 将数据绑定到当前行的指定列
        e.Item.Cells[2].Text = Price.ToString("c");
    }
}
```

另外比较常用的一个事件是 ItemCommand 事件, 如果在 DataGrid 的列模板中设置 ButtonColumn 列或者在模板列 ItemTemplate 中放置 Linkbutton 控件、CheckBox 控件、Button 控件等, 所有由这些控件引起页面提交时, 都会触发 DataGrid 的 ItemCommand 事件, 它们并不会触发它们本身的事件, 例如 Button

的 Click 事件等。这种事件机制在 ASP.NET 里面称之为“冒泡事件”。“冒泡事件”是指不同于每个按钮单独引发一个事件，来自嵌套控件（这些控件放在 DataGrid 容器中）的事件是“冒泡的”，也就是说，这些事件都将发送到容器，该容器又引发一个带有参数的一般事件，名为 ItemCommand，从参数表中的参数可以获得触发该事件的控件源。通过响应此单个事件（指仅响应 ItemCommand 事件），可以避免不必要的为子控件编写单独的事件处理程序。

实际上 CancelCommand 事件、DeleteCommand 事件、EditCommand 事件、UpdateCommand 事件都是 ItemCommand 事件，这些事件都是由固定的控件所触发的 ItemCommand 事件，CancelCommand 事件是由取消按钮所触发的，DeleteCommand 事件是由删除按钮所触发的，EditCommand 事件是由编辑按钮所触发的，而 UpdateCommand 事件是由更新按钮所触发的。

12.2 对象集合类的使用

对象集合类实际上是所有面向对象语言共有的一个概念，针对用于存储其他对象集合的对象所编写的类，称为对象集合类。对象集合类都继承自 System.ArrayList，也就是说对象集合类实际上是一个数组，而存储于对象集合类中的数据都是数组中的元素。

通常对象集合类有两个比较重要的作用。

(1) 对象的数据载体

对象集合类实际上最主要是一个数据载体的作用。就像从数据库中读取的数据集可以存储在 DataSet 或是专门的数据结合中一样，一组从同一类中实例化出来的对象同样可以存储于对象集合类中。

(2) 对象的排序

存储与对象集合类中的对象可以根据其属性进行排序，但是这需要在设计对象集合类时重载 ArrayList 对象的 Sort 方法来实现。

下面的代码是 ASP.NET Time Tracker Starter Kit 中项目对象 Project 的对象集合类。这是一个典型对象集合类，其中通过对 ArrayList 对象的 Sort 方法进行重载，完成根据项目对象的属性对对象集合类中的对象进行排序的目的。

```
using System;
using System.Collections;

namespace ASPNET.StarterKit.TimeTracker.BusinessLogicLayer
{
    //项目集合类继承自对象 ArrayList
    public class ProjectsCollection : ArrayList
    {
        //定义项目对象的属性字段的枚举
        public enum ProjectFields
        {
            InitValue,
            Name,
            ManagerUserName,
            CompletionDate,
            Duration
        }
        //重载 ArrayList 的 Sort 方法
        public void Sort(ProjectFields sortField, bool isAscending)
        {

```

```
//获得排序字段并进行排序
switch (sortField)
{
    case ProjectFields.Name:
        base.Sort(new NameComparer());
        break;
    case ProjectFields.ManagerUserName:
        base.Sort(new ManagerUserNameComparer());
        break;
    case ProjectFields.CompletionDate:
        base.Sort(new CompletionDateComparer());
        break;
    case ProjectFields.Duration:
        base.Sort(new DurationComparer());
        break;
}
if (!isAscending) base.Reverse();
}

private sealed class NameComparer : IComparer
{
    public int Compare(object x, object y)
    {
        Project first = (Project) x;
        Project second = (Project) y;
        return first.Name.CompareTo(second.Name);
    }
}

private sealed class ManagerUserNameComparer : IComparer
{
    public int Compare(object x, object y)
    {
        Project first = (Project) x;
        Project second = (Project) y;
        return first.ManagerUserName.CompareTo(second.ManagerUserName);
    }
}

private sealed class CompletionDateComparer : IComparer
{
    public int Compare(object x, object y)
    {
        Project first = (Project) x;
        Project second = (Project) y;
        return first.EstCompletionDate.CompareTo(second.EstCompletionDate);
    }
}

private sealed class DurationComparer : IComparer
```

```
{  
    public int Compare(object x, object y)  
    {  
        Project first = (Project) x;  
        Project second = (Project) y;  
        return first.EstDuration.CompareTo(second.EstDuration);  
    }  
}  
}  
}
```

12.3 GDI+技术

12.3.1 GDI+简介

GDI+是Microsoft Windows XP操作系统的子系统，负责在屏幕和打印机上显示信息。顾名思义，GDI+是GDI(Windows早期版本提供的图形设备接口)的后续版本。GDI+是一种应用程序编程接口(API)，通过一套部署为托管代码的类来展现。这套类被称为GDI+的“托管类接口”。

应用程序的程序员可利用GDI+这样的图形设备接口在屏幕或打印机上显示信息，而不需要考虑特定显示设备的具体情况。应用程序的程序员调用GDI+类提供的方法，而这些方法又反过来相应地调用特定的设备驱动程序。GDI+将应用程序与图形硬件隔离，而正是这种隔离允许开发人员创建与设备无关的应用程序。

12.3.2 GDI+技术

GDI+技术由二维矢量图形、图像处理和版式3个部分组成。

(1) 二维矢量图形

矢量图形包括坐标系统中的序列点指定的绘图基元(直线、曲线和图形等)。例如，直线可通过它的两个端点来指定，而矩形可通过确定其左上角位置的点并给出其宽度和高度的一对数字来指定。简单路径可由通过直线连接的点的数组来指定。

GDI+提供了存储基元自身相关信息的类(和结构)、存储基元绘制方式相关信息的类、以及实际进行绘制的类。例如，Rectangle结构存储矩形的位置和尺寸，Pen类存储有关线条颜色、线条粗细和线型的信息，而Graphics类具有用于绘制直线、矩形、路径和其他图形的方法。还有几种Brush类，它们存储有关如何使用颜色或图案来填充封闭图形和路径的信息。

使用GDI+可以在图元文件中记录矢量图像(图形命令的序列)。GDI+提供了Metafile类，可用于记录、显示和保存图元文件。MetafileHeader和MetaHeader类允许检查图元文件头中存储的数据。

(2) 图像处理

某些种类的图片很难或者根本无法借助矢量图形技术来显示。例如，工具栏按钮上的图片和显示为图标的图片就难以指定为直线和曲线的集合。高分辨率数字照片会更难利用矢量技术来制作。这种类型的图像可存储为位图，即代表屏幕上单个点颜色的数字数组。GDI+提供了Bitmap类，可用于显示、操作和保存位图。

(3) 版式

版式关系到使用各种字体、字号和样式来显示文本。GDI+为这种复杂任务提供了大量的支持。GDI+中

的新功能之一是子像素消除锯齿，它可以使文本在 LCD 屏幕上呈现时显得比较平滑。

GDI+ 的托管类接口包含大约 60 个类、50 个枚举和 8 个结构。Graphics 类是 GDI+ 的核心功能，它是实际绘制直线、曲线、图形和文本的类。

许多类与 Graphics 类一起使用。例如，Graphics.DrawLine 方法接收 Pen 对象，该对象中存有所要绘制的线条的属性（颜色、宽度、虚线线型和外观）。Graphics.FillRectangle 方法可以接收指向 LinearGradientBrush 对象（它使用 Graphics 对象以渐变色填充矩形）的指针。Font 和 StringFormat 对象影响 Graphics 对象绘制文本的方式。Matrix 对象存储并操作 Graphics 对象的全局变形，该对象用于旋转、缩放和翻转图像。

GDI+ 为组织图形数据提供了几种结构（例如，Rectangle、Point 和 Size），而且，某些类的主要作用是结构化数据类型。例如，BitmapData 类是 Bitmap 类的助手，而 PathData 类是 GraphicsPath 类的助手。

GDI+ 定义了几种枚举，它们是相关常量的集合。例如，LineJoin 枚举包含元素 Bevel、Miter 和 Round，它们指定用于连接两个线条的样式。

12.3.3 GDI+技术简单应用

(1) 绘制线条

若要在 GDI+ 中绘制线条，需要 Graphics 对象和 Pen 对象。Graphics 对象提供 DrawLine 方法，Pen 对象保存该线条的属性，如颜色和宽度。Pen 对象作为参数传递给 DrawLine 方法。

下面一段代码示范了绘制一条从坐标为 (0, 0) 的点到坐标为 (100, 100) 点的一条直线。

```
// 实例化一个 Graphics 对象
Graphics g = new Graphics();
// 实例化一个 Pen 对象
Pen blackPen = new Pen(Color.Black, 3);
// 实例化一个 Point 对象作为起始点
Point startPoint = new Point(0, 0);
// 实例化一个 Point 对象作为结束点
Point endPoint = new Point(100, 100);
// 调用 Graphics 对象 DrawLine 方法画线
g.DrawLine(blackPen, startPoint, endPoint);
```

(2) 绘制文字

下面一段代码示范了使用 GDI+ 技术绘制文字的过程。

```
// 实例化一个 Graphics 对象
Graphics g = new Graphics();
// 实例化一个 Brush 画笔颜色对象
Brush blackBrush = new SolidBrush(Color.Black);
// 实例化一个 FontFamily 字体对象
FontFamily familyName = new FontFamily("Times New Roman");
Font myFont = new Font(familyName, 24, FontStyle.Regular, GraphicsUnit.Pixel);
// 实例化一个 PointF 起始点对象
PointF startPoint = new PointF(10, 20);
// 调用 Graphics 对象的 DrawString 方法绘制文字
g.DrawString("Hello World!", myFont, blackBrush, startPoint);
```

传递给画笔颜色对象 SolidBrush 构造函数的参数是 Color 对象的一个系统定义的属性，表示不透明黑色。字体对象 FontFamily 构造函数接收标识字体序列的单个字符串参数。FontFamily 对象是传递给 Font 构造函数的第一个参数。传递给 Font 构造函数的第二个参数指定字号，第三个参数指定字形。

Regular 值是 FontStyle 枚举的一个成员。传递给 Font 构造函数的最后一个参数规定字体大小（以像素计）。Pixel 值是 GraphicsUnit 枚举的一个成员。传递给 DrawString 方法的第一个参数是要绘制的字符串，第二个参数是 Font 对象。第三个参数是指定字符串颜色的 Brush 对象，最后一个参数是保存字符串绘制位置的 PointF 对象。

(3) 创建图形对象

通过上面的讲解可以知道若要在显示设备上绘制图形，则需要 Graphics 对象。Graphics 对象与绘制表面相关联。

下面一段代码示例演示了如何使用 Graphics 对象绘制一个矩形。

```
//创建Graphics对象
Graphics myGraphics = this.CreateGraphics();
//创建画笔颜色对象
SolidBrush redBrush = new SolidBrush(Color.Red);
//绘制矩形
myGraphics.FillRectangle(redBrush, 0, 0, 100, 100);
```

除了使用同一颜色填充矩形外，还可以使用阴影或是其他图像的纹理填充图案，下面一段代码示例演示了如何使用阴影填充图案。其中 HatchBrush 对象就是填充图形阴影对象，HatchBrush 对象的构造函数带有 3 个参数：阴影样式、阴影线颜色和背景颜色。

```
HatchBrush hBrush = new HatchBrush(
    HatchStyle.Horizontal,
    Color.Red,
    Color.FromArgb(255, 128, 255));
e.Graphics.FillEllipse(hBrush, 0, 0, 100, 60);
```

下面一段代码示例演示了如何使用 Image 类和 TextureBrush 类，用纹理填充闭合的形状，也就是使用图形纹理填充椭圆区域。

```
//实例化一个Image对象
Image image = new Bitmap("ImageFile.jpg");
//实例化一个填充闭合区域的TextureBrush对象
TextureBrush tBrush = new TextureBrush(image);
//绘制图像之前应用到该图像的变换
tBrush.Transform = new Matrix(75.0f/640.0f, 0.0f, 0.0f, 75.0f/480.0f, 0.0f, 0.0f);
//绘制图形
e.Graphics.FillEllipse(tBrush, new Rectangle(0, 150, 150, 250));
```

(4) 坐标系统

GDI+ 使用 3 个坐标空间：全局、页面和设备。进行 myGraphics.DrawLine(myPen, 0, 0, 160, 80) 调用时，传递到 DrawLine 方法的点 (0, 0) 和 (160, 80) 位于全局坐标空间中。坐标先要通过变形序列，然后 GDI+ 才能在屏幕上绘制线条。一种变形将全局坐标转换为页面坐标，另一种变形将页面坐标转换为设备坐标。

例如，使用绘制坐标原点位于绘制区域的主体而非左上角的坐标系统，需要让原点位于距工作区左边缘 100 像素、距顶部 50 像素的位置。三种坐标空间中线条始终点的坐标为：全局 (0, 0) 到 (160, 80)、页 (100, 50) 到 (260, 130)、设备 (100, 50) 到 (260, 130)。

将全局坐标映射到页面坐标的变形称为“全局变形”，使用 Graphics 对象的 Transform 方法可以对当前绘制图形的坐标系统进行“全局变形”。例如全局变形是在 x 方向平移 100 个单位、在 y 方向平移 50 个单位，下面的示例演示了 Graphics 对象的全局变形，并随后使用该 Graphics 对象来绘制前图中显示的线条。

```
myGraphics.TranslateTransform(100, 50);
myGraphics.DrawLine(myPen, 0, 0, 160, 80);
```

12.4 移动 Web 应用

12.4.1 移动 Web 应用基础知识

现今移动设备已成为人类日常生活中的一部分。当这些移动设备连接到 Internet 时，移动设备的力量将无穷无尽，移动设备的使用者可以在任何事件、任何地点使用移动设备接收和发送数据。当然，移动设备的使用离不开移动设备应用程序的服务器，典型的移动应用程序是在服务器上使用 WML、WMLScript 和 WBMP 开发的。

对于动态 WML 应用程序，开发者可以使用 ASP、JSP、PHP 等等。移动设备包括蜂窝电话、寻呼机、掌中浏览器、袖珍 PC 和车载 PC。这些设备中少数支持 WML，少数支持 HTML，还有少数同时支持 WML 和 HTML。如果想确保应用程序能在大多数的移动设备中使用，必须以 WML 和有限的 HTML 创建应用程序。

.NET 框架包括用于移动 Web 开发的 ASP.NET。基于 ASP.NET 的移动 Web 应用程序既支持传统的 Web 客户端如 IE 和 Netscape，又支持移动客户端如 SmartPhone、PDA 等。ASP.NET 移动 Web 应用程序可以在任何.NET 支持的如 VB.NET、JScript、C++、C# 等语言环境下进行开发。

12.4.2 ASP.NET 移动 Web 应用程序

要开发 ASP.NET 移动 Web 应用程序，必须引用由.NET Mobile Web SDK 提供（通过 MobileUI.DLL 文件）的命名空间 System.Mobile.UI。.NET Mobile Web SDK 提供了 3 个容器对象：MobilePage、Form 和 Panel。MobilePage 控件是移动应用程序的重要容器，相当于普通 ASP.NET 中 Page 对象，一个单独的 MobilePage 可以有一个或多个 Form 控件。移动 Form 控件和普通 ASP.NET 中的 Form 控件的概念基本一致，一个移动 Form 控件可以有 0 个或多个 Panel 控件。Panel 控件用于给各种 Mobile 控件分组。

Mobile 控件可以被分为 3 个主要的组。

(1) 用户界面 (UI) 控件

用户界面控件是如 Label 控件一样允许用户控制用户界面的一组控件。该组控件基本上能和普通的 ASP.NET 中的控件一一对应。

(2) 验证 (Validation) 控件

验证控件允许验证用户输入值的正确性，如 RequiredFieldValidator 控件。这些控件在向服务器发送数据之前验证用户输入的数据。这组控件和普通的 ASP.NET 中的验证控件十分类似。

(3) 功能 (Utility) 控件

功能控件是诸如日历控件一类的控件。

12.5 技术点总结

本章讲解了 ASP.NET Time Tracker Starter Kit 中比较突出的几个技术点，本章对技术点的讲解注重概念和基础，希望读者在通过对本章的学习之后，能理解几个技术点的基本概念和基本应用。在后面的章节中，将根据几个技术点在 ASP.NET Time Tracker Starter Kit 中的详细应用，具体讲解这几个技术点。希望通过本章的学习读者可以了解以下内容。

- DataGrid 常用事件
- 对象集合类的基本概念和应用
- GDI+技术的基本概念

- 使用GDI+技术绘制简单图形
- ASP.NET移动Web应用程序的基本概念

第 13 章

系统底层 开发

在前面的两章中，从整体的角度介绍了系统的构建以及系统中主要的技术点。和实际的项目实施一样，本书也试图通过从设计到开发的角度分析 ASP.NET Starter Kit 的示例。这种角度本身符合逻辑思维中分析问题、解决问题的顺序，也符合软件开发过程中的逻辑顺序。本章将讲解 ASP.NET Time Tracker Starter

Kit 中包括页面样式、Web.config、Global.asax、用户控件等几个基础设计方面的问题，这几个问题是 ASP.NET Web 应用程序在系统开发初期需要解决的问题，是整个 Web 应用程序共享的通用的部分，对 Web 应用程序开发起着极其重要的作用。

13.1 页面样式

对于一个成功的 Web 应用程序来说，完整而又统一的页面样式是必不可少的。虽然页面样式的设计并不直接影响系统的技术框架等核心问题，但是却和用户界面有紧密的联系。完整而又统一的页面样式既提高了系统的可维护性，又简化了用户界面的表现方式。

ASP.NET Time Tracker Starter Kit 中所有页面使用了同一个页面样式文件 Style.css，该文件位于主目录下。

13.2 Web.config

在本书的第 3 章中，讲解了 Web.config 的基础知识以及 Web.config 的部分节点定义。本节中将结合 ASP.NET Time Tracker Starter Kit 的 Web.config 文件进一步讲解 Web.config 文件的一些深入应用。

13.2.1 系统级常量的定义

在 Web 应用程序的设计过程中经常需要定义一些系统级常量，这些常量有可能因为应用的不同而改变，但是在同一个应用中这些常量总是保持一致的，例如数据库连接字符串等等。为了提高系统的灵活性，这些系统级常量会被存储于 Web.config 中的 <appSettings> 节点下，在本书的第 3 章中已经讲解过了这种存储

方法的优点。在 ASP.NET Time Tracker Starter Kit 的 Web.config 中存储了如下的几个系统级常量。

```
<appSettings>
    <add key="ConnectionString" value="server=FAYE\FAYE;uid=sa;pwd=;database=TimeTracker" />
    <add key="DefaultRoleForNewUser" value="1" />
    <add key="UserAccountSource" value="None" />
    <add key="FirstDayOfWeek" value="1" />
</appSettings>
```

第一个字符串 ConnectionString 是数据库连接字符串。

第二个字符串 DefaultRoleForNewUser 是系统新注册用户的默认用户组，将该字符串的值设定为 1 代表新注册的用户默认为系统管理员，2 代表项目经理，3 代表普通项目开发人员。

第三个字符串 UserAccountSource 是系统用户的来源，可以将 ASP.NET Time Tracker Starter Kit 的用户系统设置从活动目录中取得。

第四个字符串 FirstDayOfWeek 是系统中星期的第一天，可以设定从 0 到 6 代表从周日到周六。因为 ASP.NET Time Tracker Starter Kit 中涉及到以周作为单位统计工作量，所以有必要指定星期的第一天。

以上 4 个字符串存储了 4 个系统级常量的值，这些常量可以看成是不同应用的配置信息，更改这些配置信息并不需要重新编译应用程序。

13.2.2 用户身份验证

在 Web.config 中的<system.web>节点下配置了系统的用户身份验证。

```
<authorization>
    <deny users="?" />
</authorization>
<authentication mode="Forms">
    <forms name=".ASPxAUTH" protection="All" timeout="60" loginUrl="desktopdefault.aspx" />
</authentication>
```

上述代码首先禁止了所有没有通过身份验证用户的请求，然后设置系统的用户身份验证模式为 Forms，并指定了未通过身份验证用户的所有请求必须转向登录页面的 URL。

Web.config 还单独制定了部分页面和目录的身份验证方式。

```
<location path="Register.aspx">
    <system.web>
        <authorization>
            <allow users="?" />
        </authorization>
    </system.web>
</location>
<location path="AccessDenied.aspx">
    <system.web>
        <authorization>
            <allow users="?" />
        </authorization>
    </system.web>
</location>
<location path="ErrorPage.aspx">
    <system.web>
        <authorization>
            <allow users="?" />
        </authorization>
    </system.web>
</location>
```

```
</system.web>
</location>
<location path="SourceViewer/srcview.aspx">
    <system.web>
        <authorization>
            <allow users="?" />
        </authorization>
    </system.web>
</location>
<location path="SourceViewer/tabview.aspx">
    <system.web>
        <authorization>
            <allow users="?" />
        </authorization>
    </system.web>
</location>
<location path="SourceViewer">
    <system.web>
        <authorization>
            <allow users="?" />
        </authorization>
    </system.web>
</location>
<location path="styles.css">
    <system.web>
        <authorization>
            <allow users="?" />
        </authorization>
    </system.web>
</location>
<location path="script.js">
    <system.web>
        <authorization>
            <allow users="?" />
        </authorization>
    </system.web>
</location>
<location path="images">
    <system.web>
        <authorization>
            <allow users="?" />
        </authorization>
    </system.web>
</location>
<location path="docs">
    <system.web>
        <authorization>
            <allow users="?" />
        </authorization>
    </system.web>
</location>
```

以上页面和目录都是访问者可以在未通过身份验证的前提下访问的，因为在 Web.config 文件中的 <system.web> 节点下限定了未通过身份验证的用户是不能访问整个应用程序的，但是部分文件和目录还是允许未通过身份验证用户访问的，因此单独对这部分文件和目录进行配置。

13.3 Global.asax

ASP.NET Time Tracker Starter Kit 的 Global.asax 文件中定义了 Application_BeginRequest、Application_AuthenticateRequest、GetApplicationPath 三个方法。

(1) Application_BeginRequest

Application_BeginRequest 方法的代码如下。

```
protected void Application_BeginRequest(Object sender, EventArgs e)
{
    try
    {
        //判断请求的客户端操作系统语言是否为空
        if (Request.UserLanguages != null)
            //取用默认的语言环境
            Thread.CurrentThread.CurrentCulture = CultureInfo.CreateSpecificCulture (Request.
UserLanguages[0]);
        else
            //如果请求的客户端未提供语言环境则采用 en-us 的语言环境
            Thread.CurrentThread.CurrentCulture = new CultureInfo("en-us");
        //设定应用程序的运行区域性和请求的语言环境相匹配
        Thread.CurrentThread.CurrentUICulture = Thread.CurrentThread.CurrentCulture;
    }
    //捕获异常
    catch (Exception ex)
    {
        //如果抛出异常，则采用 en-us 的语言环境
        Thread.CurrentThread.CurrentCulture = new CultureInfo("en-us");
    }
}
```

从上述代码中可以看到 Application_BeginRequest 方法的功能是设置应用程序的全球化，使得使用不同语言环境的客户端都能够正常使用应用程序。例如当客户端使用中文操作系统时，应用程序就会采用中文环境。但是在动态数据报表模块中，由于使用 GDI+ 技术绘制动态报表会出现编码格式不兼容的情况，因此为了保证动态图表模块的正常运行，在中文环境下运行该应用程序有必要对 Application_BeginRequest 方法做出如下的调整。

```
protected void Application_BeginRequest(Object sender, EventArgs e)
{
    try
    {
        Thread.CurrentThread.CurrentCulture = new CultureInfo("en-us");
        Thread.CurrentThread.CurrentUICulture = Thread.CurrentThread.CurrentCulture;
    }
    catch (Exception ex)
    {
```

```

        Thread.CurrentThread.CurrentCulture = new CultureInfo("en-us");
    }
}

```

另外一种解决方案就是更改 Web.config 文件中的 globalization 节点。

```
<globalization requestEncoding="gb2312" responseEncoding="gb2312" culture="en-US" uiCulture="en-US"/>
```

更改 Web.config 文件的 globalization 节点会使得使用 GDI+技术产生的动态图表中部分文字无法显示完整，因此不建议采用此种方案。

(2) Application_AuthenticateRequest

Application_AuthenticateRequest 方法的主要功能是检验用户的身份验证信息，调用了用户模块后台类库，因此将在用户管理章节进行介绍。

(3) GetApplicationPath

GetApplicationPath 方法的代码如下。

```

public static string GetApplicationPath(HttpContext request)
{
    string path = string.Empty;
    try
    {
        //判断请求的路径是否为应用程序的主目录
        if(request.ApplicationPath != "/")
            //如果不是主目录，则获得请求路径
            path = request.ApplicationPath;
    }
    catch (Exception e)
    {
        throw e;
    }
    //返回请求路径
    return path;
}

```

通过上述代码可以看出，GetApplicationPath 的功能是获得当前请求的路径。

13.4 用户控件 Banner

用户控件 Banner 是 ASP.NET Time Tracker Starter Kit 的页面顶部导航栏用户控件，效果如图 13-1 中用线圈起的部分。



图 13-1 用户控件 Banner

13.4.1 逻辑层

ASP.NET Time Tracker Starter Kit 中的逻辑层为所有选项卡式的导航按钮专门建立了一个类 TabItem，该类可以被实例化成为一个选项卡的对象，存储该选项卡的显示文字和链接地址。TabItem 的代码如下。

```
using System;

namespace ASPNET.StarterKit.TimeTracker.BusinessLogicLayer
{
    public class TabItem
    {
        private string _name;
        private string _path;

        public TabItem(string newName, string newPath)
        {
            _name = newName;
            _path = newPath;
        }

        public string Name
        {
            get { return _name; }
            set { _name = value; }
        }

        public string Path
        {
            get { return _path; }
            set { _path = value; }
        }
    }
}
```

TabItem 的代码比较清晰，将 TabItem 抽取成为对象，构建了两个构造器 Name 和 Path 用于存储和设定 TabItem 对象的两个属性。

13.4.2 后台编码类

用户控件 Banner 的后台编码类为 Banner.ascx.cs，其代码如下。

```
using System;
using System.Collections;
using System.Web.Security;
using ASPNET.StarterKit.TimeTracker.BusinessLogicLayer;

namespace ASPNET.StarterKit.TimeTracker.Web
{
```

```
public abstract class Banner : System.Web.UI.UserControl
{
    protected System.Web.UI.WebControls.LinkButton LogOff;
    protected System.Web.UI.WebControls.DataList tabs;
```

在用户控件的载入方法 Page_Load 中判断导航栏中的各个选项卡按钮是否需要显示。

```
private void Page_Load(object sender, System.EventArgs e)
{
    //定义数组存储选项卡按钮集合
    ArrayList tabItems = new ArrayList();
    //判断用户是否通过身份验证
    if (Request.IsAuthenticated)
        //如果用户通过身份验证则显示 Log 选项卡按钮
        tabItems.Add(new TabItem("Log", "TimeEntry.aspx?index=" + tabItems.Count));

    //判断系统的身份验证方式是否是基于表单的
    if (Context.User.Identity.AuthenticationType == "Forms")
        //如果系统的身份验证方式是基于表单的则显示 Log off 退出按钮
        LogOff.Visible = true;
    //判断用户的身份是否是管理员或项目经理
    if (TTSecurity.IsInRole(TTUser UserRoleAdminPMgr))
    {
        //如果用户的身份是管理员或项目经理则显示 Reports 和 Administration 选项卡按钮
        tabItems.Add(new TabItem("Reports", "Reports.aspx?index=" + tabItems.Count));
        tabItems.Add(new TabItem("Administration", "ProjectList.aspx?index=" + tabItems.Count));
    }
    //定义前台页面选项卡按钮 DataList 控件的数据源
    tabs.DataSource = tabItems;
    //获得 URL 查询字符串 index 的值设定选项卡的选中项
    tabs.SelectedIndex = (Request["index"]==null) ? 0 : Convert.ToInt32(Request["index"]);
    //绑定前台页面选项卡按钮 DataList 控件
    tabs.DataBind();
}

override protected void OnInit(EventArgs e)
{
    InitializeComponent();
    base.OnInit(e);
}

private void InitializeComponent()
{
    this.LogOff.Click += new System.EventHandler(this.LogOff_Click);
    this.Load += new System.EventHandler(this.Page_Load);
}
```

用户控件中 LogOff 按钮是用户退出系统的按钮，后台编码类中定义了该按钮的单击事件。

```
private void LogOff_Click(object sender, System.EventArgs e)
{
    //注销表单身份验证
    FormsAuthentication.SignOut();
```

```
//注销 Cookies  
Response.Cookies["userroles"].Value = "";  
Response.Cookies["userroles"].Path = "/";  
Response.Cookies["userroles"].Expires = new System.DateTime(1999, 10, 12);  
//注销当前用户  
Context.User = null;  
//重定向页面  
Response.Redirect("Default.aspx", false);  
}  
}
```

13.4.3 前台页面

用户控件 Banner 的前台页面通过一个 DataList 控件显示导航栏选项卡按钮。

```
<asp:datalist      id="tabs"      SelectedItemStyle-CssClass="tab-active"      ItemStyle-CssClass="tab-inactive"
CellSpacing="0" CellPadding="0" runat="server" EnableViewState="false" RepeatDirection="horizontal">
    <itemtemplate>
        <a href='<%# Global.GetApplicationPath(Request) %>'><%# ((ASPNET.StarterKit.TimeTracker.
BusinessLogicLayer.TabItem) Container.DataItem).Path %></a>
        <%# ((ASPNET.StarterKit.TimeTracker.BusinessLogicLayer.TabItem) Container.DataItem).Name %>
    </itemtemplate>
    <selecteditemtemplate>
        <%# ((ASPNET.StarterKit.TimeTracker.BusinessLogicLayer.TabItem) Container.DataItem).Name %>
    </selecteditemtemplate>
</asp:datalist>
```

在 DataList 中定义了模板列和选中列，该 DataList 的数据源在后台编码类中已经定义，值得注意的是，在 DataList 中将数据源中的项隐式转换成 TabItem 对象之后获得其属性并显示。在后台编码类中，是将一组 TabItem 对象存储于数组中形成 DataList 控件的数据源，但是由于在前台页面并不能直接获得数组中对象的属性，需要经过隐式转换之后方可调用对象的属性。

13.5 用户控件 AdminTabs

用户控件 AdminTabs 是系统管理模块中选项卡按钮的用户控件，当用户单击页面导航按钮中的 Administration 按钮就会进入系统管理模块，在系统管理模块中可以对项目和系统用户进行分别管理。AdminTabs 用户控件就是项目管理和用户管理选项卡按钮的用户控件。其效果如图 13-2 所示。



图 13-2 用户控件 AdminTabs

AdminTabs 和 Banner 控件一样将选项卡按钮抽象成为类，实例化逻辑层中的类 TabItem 成为选项卡按钮对象。

13.5.1 后台编码类

AdminTabs 的后台编码类为 AdminTabs.ascx.cs，其代码如下。

```
namespace ASPNET.StarterKit.TimeTracker.Web
{
    using System;
    using System.Collections;
    using ASPNET.StarterKit.TimeTracker.BusinessLogicLayer;

    public abstract class AdminTabs : System.Web.UI.UserControl
    {
        protected System.Web.UI.WebControls.DataList tabs;

        在用户控件的载入方法 Page_Load 中，根据用户权限判断选项卡中的按钮显示与否。
        private void Page_Load(object sender, System.EventArgs e)
        {
            ArrayList tabItems = new ArrayList();
            //获得 URL 查询字符串 index 的值
            int mainIndex = (Request["index"]==null) ? 0 : Convert.ToInt32(Request["index"]);

            //添加 project 选项卡按钮
            tabItems.Add(new TabItem("Projects", "ProjectList.aspx?index=" + mainIndex + "&adminIndex=" +
tabItems.Count));
            //判断用户是否具有管理员权限
            if (TTSecurity.IsInRole(TTUser UserRoleAdministrator))
            {
                //如果用户具有管理员权限则添加 User 选项卡按钮
                tabItems.Add(new TabItem("Users", "UserList.aspx?index=" + mainIndex + "&adminIndex=" +
tabItems.Count));
            }
            //指定前台页面中 DataList 的数据源
            tabs.DataSource = tabItems;
            //指定前台页面中 DataList 的选中项
            tabs.SelectedIndex = (Request["adminIndex"]==null) ? 0 : Convert.ToInt32(Request["adminIndex"]);
            //绑定 DataList
            tabs.DataBind();
        }

        override protected void OnInit(EventArgs e)
        {
            InitializeComponent();
            base.OnInit(e);
        }

        private void InitializeComponent()
        {
            this.Load += new System.EventHandler(this.Page_Load);
        }
    }
}
```

```
        }
    }
}
```

AdminTabs 的页面载入方法和 Banner 用户控件的页面载入方法极其类似，都是为前台页面中的 DataList 控件构造数据源，该数据源为由一组 TabItem 对象组成的数组。

13.5.2 前台页面

AdminTabs 用户控件的前台页面通过一个 DataList 控件显示选项卡按钮，其实现方式和 Banner 控件类似，DataList 控件代码如下。

```
<asp:datatable ID="tabs" RepeatDirection="horizontal" EnableViewState="false" runat="server" CellPadding="0"
CellSpacing="0" ItemStyle-CssClass="admin-tab-inactive" SelectedItemStyle-CssClass="admin-tab-active">
    <itemtemplate>
        <a href='<% Global.GetApplicationPath(Request) %>'/><%# ((ASNET.StarterKit.TimeTracker.
BusinessLogicLayer.TabItem) Container.DataItem).Path %>'>
            <%# ((ASNET.StarterKit.TimeTracker.BusinessLogicLayer.TabItem) Container.DataItem).Name %>
        </a>
    </itemtemplate>
    <selecteditemtemplate>
        <%# ((ASNET.StarterKit.TimeTracker.BusinessLogicLayer.TabItem) Container.DataItem).Name %>
    </selecteditemtemplate>
</asp:datatable>
```

13.6 开发启示

本章中讲解了 ASP.NET Time Tracker Starter Kit 的页面样式、Web.config、Global.asax 和用户控件等方面的内容，这几部分内容都是系统的基础构件，并没有过多的设计系统的应用逻辑，希望读者通过对这几部分内容的分析和研究学会如下内容。

- 配置和安排 ASP.NET 项目中的 Web.config、Global.asax 两个系统全局文件
- 抽取系统的导航栏按钮成为对象并设计该对象的类
- 隐式转换对象获得其属性

第 14 章

用户体系

ASP.NET Time Tracker Starter Kit 的用户体系中主要包含用户管理和用户角色两大模块，引入了角色与用户对应的体制为用户分配权限。并且可以设置多种数据信息来源，包括从数据库、从 WinSAM 或从活动目录获取用户信息。在技术层面，ASP.NET Time Tracker Starter Kit 的用户体系基于.NET 中的

身份验证体系，继承了部分.NET 身份验证体系中的类，实现用户的身份验证。表 14-1 为本章知识点索引。

表 14-1 本章知识点索引

| 知 识 点 | 位 置 |
|---|--------|
| 构建向 Context.User 属性存储用户信息的 CustomPrincipal 对象 | 13.3.2 |
| 从 WindowsSAM 和活动目录中获得用户信息 | 13.3.2 |
| 构建类集合对象 | 13.3.2 |
| 自定义过滤条件过滤 string 型变量 | 13.3.2 |
| 基于表单的用户身份验证 | 13.3.3 |
| 基于角色的用户权限管理 | 13.3.3 |
| DataGrid 控件的使用 | 13.3.4 |

14.1 系统设计

14.1.1 功能设计

在功能上 ASP.NET Time Tracker Starter Kit 的用户体系分为用户注册、用户登录、用户管理 3 个模块，匿名的访问者不具有系统的权限，只有通过注册和登录的用户才具有系统的权限，当匿名访问者试图访问系统时会被重新定向到登录页面。以下描述了 3 个功能模块的功能概要。

(1) 用户注册

匿名用户可以通过用户注册模块成为正式的系统用户，匿名用户在注册过程中需填写用户全名、Email、密码等信息，新注册用户的角色会根据系统配置信息自动分配。

(2) 用户登录

匿名用户可以通过用户登录模块登录成为系统用户，登录过程

中匿名用户需填写 Email 和密码，如果 Email 和密码信息匹配则会通过系统身份验证，并获取用户信息，成为系统用户。

(3) 用户管理

系统管理员可以使用用户管理模块对系统用户体系进行维护和管理。用户管理模块中管理员可以浏览系统用户列表，查看具体用户信息，更改用户信息，还可以新增系统用户。用户体系功能说明如图 14-1 所示。

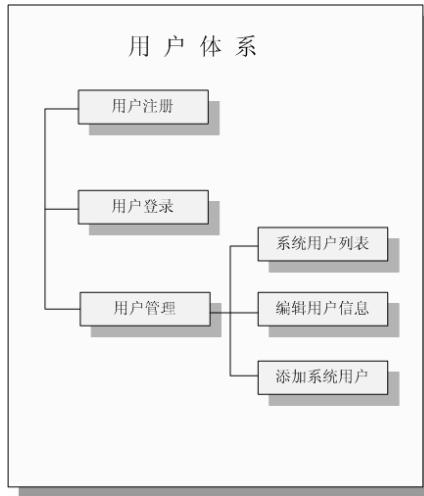


图 14-1 用户体系功能说明

ASP.NET Time Tracker Starter Kit 提供 3 种用户角色：管理员、项目经理、开发人员。角色的具体信息见表 14-2。

表 14-2

3 种用户角色信息

| 角色名称 | RoleId | RoleName |
|------|--------|-----------------|
| 管理员 | 1 | Administrator |
| 项目经理 | 2 | Project Manager |
| 开发人员 | 3 | Consultant |

3 种角色在权限结构上是从上向下兼容的，即管理员具有项目经理所具有的全部权限和管理员的特殊权限，项目经理具有开发人员所具有的全部权限和项目经理的特殊权限。表 14-3 具体描述了系统中角色与权限的对应关系。

表 14-3

角色与权限的对应关系

| 权限 | 开发人员 | 项目经理 | 管理员 |
|--------|------------|----------------------|------|
| 新建工作安排 | 对所在工程具有该权限 | 对所在工程和担任项目经理的工程具有该权限 | 具有权限 |
| 修改工作安排 | 对所在工程具有该权限 | 对所在工程和担任项目经理的工程具有该权限 | 具有权限 |
| 删除工作安排 | 对所在工程具有该权限 | 对所在工程和担任项目经理的工程具有该权限 | 具有权限 |
| 查看工作安排 | 对所在工程具有该权限 | 对所在工程和担任项目经理的工程具有该权限 | 具有权限 |
| 新建项目 | | 具有权限 | 具有权限 |
| 修改项目信息 | | 对担任项目经理的工程具有该权限 | 具有权限 |
| 删除项目 | | | 具有权限 |

续表

| 权限 | 开发人员 | 项目经理 | 管理员 |
|----------|------|-----------------|------|
| 查看项目信息 | | 对担任项目经理的工程具有该权限 | 具有权限 |
| 查看项目报表 | | 对担任项目经理的工程具有该权限 | 具有权限 |
| 查看项目人员报表 | | 对担任项目经理的工程具有该权限 | 具有权限 |
| 新建用户 | | | 具有权限 |
| 查看用户信息 | | | 具有权限 |
| 修改用户信息 | | | 具有权限 |

14.1.2 数据结构

数据库设计上用户体系中使用了 Users 和 Roles 表, 如图 14-2 所示的 User 表存储了用户的信息, Roles 表存储了用户的角色信息。Roles 表的 RoleID 字段作为主键关联 User 表的 RoleID 字段。实际上在 ASP.NET Time Tracker Starter Kit 中角色信息是相对固定的, 因此仅有 User 表作为用户体系中的主体信息表。从设计结构上可以看出, 设计者应尽可能地减轻数据机构的繁琐程度, 这样有利于提高数据库的性能。

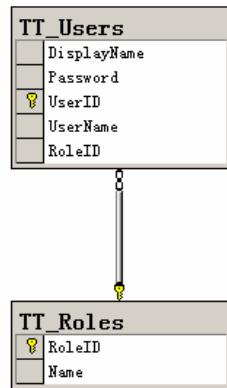


图 14-2 ASP.NET Time Tracker Starter Kit 用户体系部分数据结构

用户体系的数据库设计中采用了表 14-4 中的存储过程, 存储过程具体实现细节将在后面章节讲到。

表 14-4 用户体系所使用到的存储过程

| 存储过程名称 | 存储过程说明 |
|----------------------|------------------|
| TT_ListAllRoles | 查询系统中的所有角色 |
| TT_ListUsers | 查询系统中的用户 |
| TT_ListManagers | 查询系统中的所有项目经理和管理员 |
| TT_DeleteUser | 删除用户 |
| TT_GetUserByUserName | 根据用户的全名查询用户 |
| TT_AddUser | 添加用户 |
| TT_UpdateUser | 更新用户 |

14.2 关键技术点

14.2.1 ASP.NET运行时用户信息的存储

ASP.NET Time Tracker Starter Kit 在运行时的用户信息存储于 `HttpContext` 对象的 `User` 属性中，因此在逻辑层中构建了一个基于 `System.Security.Principal` 命名空间下的 `Iprincipal` 接口的类，用以构建用户对象。

`HttpContext` 对象封装了每一个 HTTP 请求的相关信息，当然也包括请求的用户信息。`HttpContext.User` 属性为当前 HTTP 请求获取或设置用户安全信息。由于 ASP.NET 页包含对 `System.Web` 命名空间（含有 `HttpContext` 类）的默认引用，因此在 `.aspx` 页上可以引用 `HttpContext` 的成员。例如，可以只使用 `User.Identity.Name` 获取用户名称，当前进程正代表该用户而运行。但是，如果想从 ASP.NET 代码隐藏模块中使用 `Iprincipal` 接口的成员，则必须在该模块中包括对 `System.Web` 命名空间的引用，同时还要完全限定对当前活动的请求和响应上下文以及要使用的 `System.Web` 中类的引用。

ASP.NET Time Tracker Starter Kit 在逻辑层中构建基于 `Iprincipal` 接口的类用以存储和传递用户信息，实际上是对 `Iprincipal` 接口的重载，其中重载其 `IsInRole` 完成了对用户角色验证的方法。

14.2.2 集合类的使用

对象集合类是用来实例化之后存储对象的集合对象，因此集合类继承了 `ArrayList` 类。设计者根据需求为某些系统中需要集中存储的对象设计对象集合类，在 ASP.NET Time Tracker Starter Kit 中，用户对象 `User`、角色对象 `Roles`、项目对象 `Project`、项目分类对象 `Category` 等都有对应的集合类。构建集合类时，需要考虑到对集合类内部存储对象某一个属性排序的情况，重载 `ArrayList` 类的 `Sort` 方法可以实现对集合类内部存储对象某一个属性排序，`ArrayList` 类提供 `Sort` 方法，该方法将 `Icomparer` 接口实现作为一个参数采用。

14.3 用户体系的实现

14.3.1 数据库设计

在 14.1.2 数据结构章节中已经列出了数据表的数据库结构以及用户体系中采用的存储过程列表。本节将对存储过程做详细讲解。

(1) TT_ListAllRoles

存储过程 `TT_ListAllRoles` 返回系统中所有角色的 ID 和角色名。代码清单如下。

```
CREATE PROCEDURE TT_ListAllRoles
AS
SELECT
    RoleID,
    Name
FROM
```

TT_Roles

(2) TT_ListUsers

存储过程 TT_ListUsers 根据用户的 ID 和角色返回包括用户 ID、用户名、用户角色 ID、用户角色名在内的该用户可以安排的用户列表。如果用户角色是开发人员，那么将返回该用户的列表；如果用户角色是项目经理，那么将返回该用户担任项目经理的项目中的所有人员列表；如果用户角色是管理员，那么将返回系统中所有用户的列表。代码清单如下。

```
CREATE PROCEDURE TT_ListUsers
(
    @UserID int,
    @RoleID int
)
AS
/*如果用户角色是管理员则返回系统中的所有用户信息*/
IF @RoleID = 1
BEGIN
SELECT
    UserID,
    UserName,
    TT_Users.RoleID,
    TT_Roles.Name 'RoleName'
FROM
    TT_Users INNER JOIN TT_Roles ON TT_Users.RoleID = TT_Roles.RoleID
END
/*如果用户角色是项目经理则返回该用户担任项目经理的项目中的所有项目人员信息*/
ELSE IF @RoleID = 2
BEGIN
SELECT DISTINCT
    TT_Users.UserID,
    TT_Users.UserName,
    TT_Users.RoleID,
    TT_Roles.Name 'RoleName'
FROM TT_Users
INNER JOIN TT_ProjectMembers ON TT_Users.UserID=TT_ProjectMembers.UserID
INNER JOIN TT_Projects ON TT_ProjectMembers.ProjectID=TT_Projects.ProjectID
INNER JOIN TT_Roles ON TT_Users.RoleID = TT_Roles.RoleID
WHERE @UserID = TT_Projects.ManagerUserID OR @UserID = TT_Users.UserID
END
/*如果用户角色是开发人员则返回该用户的信息*/
ELSE
SELECT
    UserID,
    UserName,
    TT_Users.RoleID,
    TT_Roles.Name 'RoleName'
FROM
    TT_Users INNER JOIN TT_Roles ON TT_Users.RoleID = TT_Roles.RoleID
WHERE UserID = @UserID
```

(3) TT_ListManagers

存储过程 TT_ListManagers 返回系统中角色为项目经理或管理员的所有用户列表。代码清单如下。

```
CREATE PROCEDURE TT_ListManagers  
  
AS  
  
SELECT  
    UserID, UserName, RoleID  
  
FROM  
    TT_Users  
Where  
    RoleID = 2  
OR  
    RoleID = 1
```

(4) TT_DeleteUser

存储过程 TT_DeleteUser 为删除系统用户的存储过程。代码清单如下。

```
CREATE PROCEDURE TT_DeleteUser  
(  
    @UserID int  
)  
AS  
  
DELETE FROM  
    TT_Users  
  
WHERE  
    UserID = @UserID
```

(5) TT_GetUserByUserName

存储过程 TT_GetUserByUserName 根据用户的 Username 返回用户信息，代码清单如下。

```
CREATE PROCEDURE TT_GetUserByUserName  
(  
    @UserName nvarchar(50)  
)  
  
AS  
  
SELECT  
    UserID,  
    UserName,  
    Password,  
    RoleID  
  
FROM  
    TT_Users WHERE UserName = @UserName
```

(6) TT_AddUser

存储过程 TT_AddUser 为添加系统用户的存储过程，如果添加成功，则返回新添加用户的 UserID；如果要被添加用户的 Username 已经存在，则返回的-2。代码清单如下。

```
CREATE PROCEDURE TT_AddUser  
(  
    @UserName nvarchar(50),
```

```

    @Password nvarchar(50),
    @DisplayName nvarchar(50),
    @RoleID int
)
AS

/*判断即将添加用户的 Username 是否已经存在*/
IF Not Exists (SELECT UserName FROM TT_Users WHERE UserName=@UserName)
BEGIN

    INSERT INTO TT_Users
    (
        Username,
        Password,
        DisplayName,
        RoleID
    )
    VALUES
    (
        @UserName,
        @Password,
        @DisplayName,
        @RoleID
    )
/*返回新添加用户的 UserId*/
SELECT
    @@Identity AS UserID
END
/*如果将要添加用户的 Username 已经存在则返回-2*/
ELSE
    SELECT -2 AS UserID

```

(7) TT_UpdateUser

存储过程 TT_UpdateUser 为更新用户信息的存储过程，如果更新成功，则返回 0；如果更新过程中发生错误，则返回字段 retval，标识出错种类。代码清单如下。

```

CREATE PROCEDURE TT_UpdateUser
(
    @UserID int,
    @UserName nvarchar(50),
    @Password nvarchar(50),
    @DisplayName nvarchar(50),
    @RoleID int
)
AS

/*判断将要更新用户的 UserId 和 Username 是否存在*/
IF Not Exists (SELECT UserName FROM TT_Users WHERE UserName=@UserName AND UserID<>@UserID)
BEGIN
    UPDATE
        TT_Users
    SET UserName=@UserName,

```

```
        Password = @Password,
        DisplayName = @DisplayName,
        RoleID = @RoleID
    WHERE
        UserID=@UserID
    /*更新过程中出错则返回 retval 的字段为 1*/
    IF (@@Error<>0) GOTO ErrorHandler
    SELECT
        1 AS retval
    END
    /*如果将要更新用户的 UserId 和 Username 不存在，则返回 retval 字段为-2*/
    ELSE
        SELECT
            -2 AS retval
    IF @@Error<>0
        GOTO ErrorHandler
    ELSE
        BEGIN
            RETURN (0)
        END
ErrorHandler:
BEGIN
    RETURN (1)
END
```

14.3.2 逻辑层

ASP.NET Time Tracker Starter Kit 的用户体系调用了系统的逻辑层命名空间 ASPNET.StarterKit.TimeTracker.BusinessLogicLayer 中的如下类。

(1) CustomPrincipal.cs

类 CustomPrincipal 继承了 System.Security.Principal 命名空间下的 Iprincipal 接口，所以该类可以被实例化成为存储用户信息的对象，System.Security.Principal 命名空间封装了定义表示应用程序运行时安全上下文的用户对象。继承 Iprincipal 接口就必须重载其 IIdentity 接口和 IsInRole 方法，其中 IIdentity 接口为获得当前用户标识的接口，IsInRole 方法为判断当前用户是否属于某个角色的方法。在 Global.asax 中调用了该类实例化之后的对象，存储当前请求用户信息，并判断当前用户的角色信息。类 CustomPrincipal 的代码清单如下。

```
using System;
//引用命名空间 System.Security.Principal
using System.Security.Principal;

namespace ASPNET.StarterKit.TimeTracker.BusinessLogicLayer
{
    //类 CustomPrincipal 继承 Iprincipal 接口
    public class CustomPrincipal : IPrincipal
    {
        private int _userID;
```

```
private string _userRole = String.Empty;
private string _name;

//定义 IIdentity 接口
protected IIdentity _Identity;

public CustomPrincipal()
{
}

//重载构造函数, 为类中私有成员变量赋值
public CustomPrincipal(
    IIdentity identity,
    int userID,
    string userRole,
    string name)
{
    _Identity = identity;
    _userID = userID;
    _userRole = userRole;
}

//重载 IIdentity 接口
public IIdentity Identity
{
    get { return _Identity; }
    set { _Identity = value; }
}

public int UserID
{
    get { return _userID; }
    set { _userID = value; }
}

public string UserRole
{
    get { return _userRole; }
    set { _userRole = value; }
}

public string Name
{
    get { return _name; }
    set { _name = value; }
}
```

类 CustomPrincipal 重载了接口 Iprincipal 的 IsInRole 方法，该方法判断用户是否属于某个角色，并返回一个布尔型变量。

```
public bool IsInRole(string role)
{
    string[] roleArray = role.Split(new char[] {','});
```

```
        foreach (string r in roleArray)
    {
        if (_userRole == r)
            return true;
    }
    return false;
}
}
```

(2) DirectoryHelper.cs

类 DirectoryHelper 调用命名空间 System.DirectoryServices 下的 DirectorySearcher 类判断用户是否是 WindowsSAM 或活动目录中的用户。System.DirectoryServices 命名空间用以从托管代码简便地访问活动目录。该命名空间包含两个组件类，即 DirectoryEntry 和 DirectorySearcher，它们使用活动目录服务接口（ADSI）技术。ADSI 是 Microsoft 提供的一组接口，作为使用各种网络提供程序的灵活的工具。DirectoryHelper 类的代码清单如下。

```
using System;
using System.Configuration;
//引用命名空间 System.DirectoryServices
using System.DirectoryServices;

namespace ASPNET.StarterKit.TimeTracker.BusinessLogicLayer
{
    public class DirectoryHelper
    {
        //定义全局私有变量_activeDirectoryPath 存储活动目录路径
        private const string _activeDirectoryPath = @"GC://";
        private const string _filter = "(&(ObjectClass=Person)(SAMAccountName={0}))";
        //定义全局私有变量_windowsSAMPPath 存储 WindowsSAM 路径
        private const string _windowsSAMPPath = @"WinNT://";

        private static string _path = String.Empty;

        private DirectoryHelper()
        {
        }
    }
}
```

方法 FindUser 根据传入的用户存储路径和用户的 FirstName 和 LastName 在 WindowsSAM 或活动目录中判断该用户是否存在并返回布尔型变量

```
public static bool FindUser(string identification, ref string FirstName, ref string LastName)
{
    bool result = false;

    //确定用户的数据源是 WindowsSAM 还是活动目录
    //如果用户数据源是 WindowsSAM
    if (ConfigurationSettings.AppSettings[Web.Global.CfgKeyUserAcctSource] == "WindowsSAM")
    {
        //提取域名和用户名
        string [] samPath = identification.Split(new char[] { '\\' });
        //构造路径
        string userPath = string.Format("{0}\\{1}", samPath[0], samPath[1]);
        //从 Active Directory 中查找用户
        result = FindUser(userPath, FirstName, LastName);
    }
}
```

```
_path = _windowsSAMPPath + samPath[0];

try
{
    //寻找用户是否存在
    DirectoryEntry entryRoot = new DirectoryEntry(_path);
    DirectoryEntry userEntry = entryRoot.Children.Find(samPath[1], "user");
    LastName = userEntry.Properties["FullName"].Value.ToString();
}
catch
{
    //如果用户不存在则返回 false
    result = false;
}
//如果用户存在则返回 true
result = true;
}

//如果用户数据源是活动目录
else if (ConfigurationSettings.AppSettings[Web.Global.CfgKeyUserAcctSource] == "ActiveDirectory")
{
    _path = _activeDirectoryPath;

    //提取用户名
    identification = identification.Substring(identification.LastIndexOf(@"\") + 1,
                                                identification.Length - identification.LastIndexOf(@"\") - 1);
    string userNameFilter = string.Format(_filter, identification);

    //实例化类 DirectorySearcher 成为 searcher 对象寻找用户路径是否存在
    DirectorySearcher searcher = new DirectorySearcher(_path);
    if (searcher == null)
    {
        //如果用户路径不存在，则返回 false
        return false;
    }

    //为 searcher 对象添加属性
    searcher.PropertiesToLoad.Add("givenName");
    searcher.PropertiesToLoad.Add("sn");

    //为 searcher 对象添加过滤器
    searcher.Filter = userNameFilter;

    try
    {
        //执行 searcher 对象的 FindOne 方法获得查询结果对象 search
        SearchResult search = searcher.FindOne();
        //如果查询结果对象 search 的属性与用户信息匹配则返回 true 否则返回 false
        if (search != null)
```

```
{  
    FirstName = SearchResultProperty(search, "givenName");  
    LastName = SearchResultProperty(search, "sn");  
    result = true;  
}  
else  
    result = false;  
}  
catch  
{  
    //如果查找过程中发生异常返回 false  
    result = false;  
}  
}  
else  
{  
    //如果数据源不为 WindowsSAM 或活动目录但调用该方法寻找用户，则返回 false  
    result = false;  
}  
  
return result;  
}  
// SearchResultProperty 方法是从 SearchResult 对象中获得某个属性值的方法  
private static String SearchResultProperty(SearchResult sr, string field)  
{  
    if (sr.Properties[field] != null)  
    {  
        return (String)sr.Properties[field][0];  
    }  
  
    return null;  
}  
}  
}  
}
```

(3) Roles.cs

类 Roles 为用户的角色操作的基类，该类实例化之后的对象为用户角色对象，类中也包含了角色操作方法 GetRoles，GetRoles 方法返回 RolesCollection 类型的系统角色对象集合。该类的代码清单如下。

```
using System;  
using System.Data;  
using System.Configuration;  
using ASPNET.StarterKit.TimeTracker.DataAccessLayer;  
  
namespace ASPNET.StarterKit.TimeTracker.BusinessLogicLayer  
{  
    public class Roles  
    {  
        // 定义私有成员变量  
        private string _name;  
        private int _roleID;
```

```
//构建角色对象的属性
public string Name
{
    get {return _name;}
    set {_name = value;}
}

public int RoleID
{
    get {return _roleID;}
    set {_roleID = value;}
}

//GetRoles 方法返回 RolesCollection 类型的角色对象集合
public static RolesCollection GetRoles()
{
    //获得包含系统中角色信息的 DataSet
    DataSet ds = SqlHelper.ExecuteDataset(ConfigurationSettings.AppSettings[Web.Global.CfgKeyConnString],
"TT_ListAllRoles");
    //实例化 RolesCollection
    RolesCollection roleArray = new RolesCollection();

    //将 DataSet 中的角色记录信息赋给角色对象并存储于角色对象集合中
    foreach(DataRow r in ds.Tables[0].Rows)
    {
        Roles role = new Roles();
        role.RoleID = Convert.ToInt32(r["RoleID"]);
        role.Name = r["Name"].ToString();

        roleArray.Add(role);
    }
    //返回角色对象集合
    return roleArray;
}

}

(4) RolesCollection.cs
```

(4) RolesCollection.cs

类 RolesCollection 为角色对象的集合类，该类继承 ArrayList 对象，重载其 Sort 方法。类 RolesCollection 为建立可存储的角色对象集合的基类。该类代码清单如下。

```
using System;
using System.Collections;

namespace ASPNET.StarterKit.TimeTracker.BusinessLogicLayer
{
    //类 RolesCollection 继承对象 ArrayList
    public class RolesCollection : ArrayList
    {
        //定义枚举对象 RolesFields
        public enum RolesFields
```

```
{  
}  
//重载 Sort 方法，该方法可以对枚举对象 RolesFields 排序  
public void Sort(RolesFields sortField, bool isAscending)  
{  
  
}  
}  
}
```

(5) TTUser.cs

类 TTUser 为用户对象的基类，在该类中构建了用户对象的属性获取用户的角色信息，并定义了用户对象的方法。该类的代码清单如下。

```
using System;  
using System.Data;  
using System.Configuration;  
using ASPNET.StarterKit.TimeTracker.DataAccessLayer;  
  
namespace ASPNET.StarterKit.TimeTracker.BusinessLogicLayer  
{  
    public class TTUser  
    {  
        //定义全局变量  
        public const string UserRoleNone = "0";  
        public const string UserRoleAdministrator = "1";  
        public const string UserRoleProjectManager = "2";  
        public const string UserRoleConsultant = "3";  
        public const string UserRoleAdminPMgr = UserRoleAdministrator + "," + UserRoleProjectManager;  
        public const string UserRolePMgrConsultant = UserRoleProjectManager + "," + UserRoleConsultant;  
        //定义私有成员变量  
        private string _displayName = string.Empty;  
        private string _firstName = string.Empty;  
        private string _lastName = string.Empty;  
        private string _password = String.Empty;  
        private string _role = UserRoleNone;  
        private string _roleName;  
        private int _userID;  
        private string _userName;  
  
        public TTUser()  
        {  
        }  
        //重载构造函数设置用户对象的 UserName 属性  
        public TTUser(string UserName)  
        {  
            _userName = UserName;  
        }  
        //重载构造函数设置用户对象的 UserID 属性、UserName 属性、Name 属性、Role 属性  
        public TTUser(int UserID, string UserName, string Name, string Role)  
        {  
            _userID = UserID;
```

```
_userName = UserName;
_displayName = Name;
_role = Role;
}
//构造用户对象属性
public string DisplayName
{
    get { return _displayName; }
    set { _displayName = value; }
}

public string FirstName
{
    get { return _firstName; }
    set { _firstName = value; }
}

public string LastName
{
    get { return _lastName; }
    set { _lastName = value; }
}

public string Name
{
    get { return _displayName; }
    set { _displayName = value; }
}

public string Password
{
    get { return _password; }
    set { _password = value; }
}

public string Role
{
    get { return _role; }
    set { _role = value; }
}

public string RoleName
{
    get { return _roleName; }
    set { _roleName = value; }
}

public int UserID
{
    get { return _userID; }
    set { _userID = value; }
}
```

```
public string UserName
{
    get { return _userName; }
    set { _userName = value; }
}
```

GetAllUsers 方法为返回系统所有用户对象集合的方法，该方法调用了 GetUsers 方法，并设定查询用户的权限为管理员。

```
public static UsersCollection GetAllUsers(int userID)
{
    return GetUsers(userID, TTUser UserRoleAdministrator);
}
```

GetUsers 方法为根据用户 UserId 和用户权限返回用户对象集合的方法。

```
public static UsersCollection GetUsers(int userID, string role)
{
    string firstName = string.Empty;
    string lastName = string.Empty;
    //执行存储过程 TT_ListUsers 返回存储用户信息的 DataSet
    DataSet ds = SqlHelper.ExecuteDataset(ConfigurationSettings.AppSettings[Web.Global.CfgKeyConnString],
                                         "TT_ListUsers", userID, Convert.ToInt32(role));
    //实例化 UsersCollection
    UsersCollection users = new UsersCollection();

    //将用户信息从 DataSet 中赋值到用户对象并将用户对象存储到用户对象集合中
    foreach (DataRow r in ds.Tables[0].Rows)
    {
        TTUser usr = new TTUser();
        usr.UserName = r["UserName"].ToString();
        usr.Role = r["RoleID"].ToString();
        usr.RoleName = r["RoleName"].ToString();
        usr.UserID = Convert.ToInt32(r["UserID"]);
        usr.Name = GetDisplayName(usr.UserName, ref firstName, ref lastName);
        usr.FirstName = firstName;
        usr.LastName = lastName;
        users.Add(usr);
    }
    return users;
}
```

GetDisplayName 方法为获得用户全名的方法，该方法从 Web.Config 定义的系统用户数据源中取得用户的全名并返回。

```
public static string GetDisplayName(string userName, ref string firstName, ref string lastName)
{
    string displayName = string.Empty;
    string dbName = string.Empty;

    //调用 DirectoryHelper 类的 FindUser 方法获得用户的全名
    DirectoryHelper.FindUser(userName, ref firstName, ref lastName);
```

```

//判断 FindUser 方法是否返回正确的用户全名
if (firstName.Length > 0 || lastName.Length > 0)
{
    displayName = firstName + " " + lastName;
}
else
{
    //如果 FindUser 方法为返回正确的用户全名则调用 GetDisplayNameFromDB 方法获得用户全名
    dbName = GetDisplayNameFromDB(userName);
    if (dbName != string.Empty)
        displayName = dbName;
    else
        displayName = userName;
}
//返回用户全名
return displayName;
}

```

GetDisplayNameFromDB 为从数据库中读取用户全名的方法，该方法在使用 GetDisplayName 方法获得用户全名时，在 Web.Config 中定义用户数据源不为 WindowsSAM 或活动目录的情况下被调用。

```

public static string GetDisplayNameFromDB(string userName)
{
    string displayName = string.Empty;
    //调用存储过程 TT_GetUserDisplayName 从数据库中获得用户全名
    displayName = Convert.ToString(SqlHelper.ExecuteScalar(ConfigurationSettings.AppSettings[Web.
Global.CfgKeyConnString], "TT_GetUserDisplayName", userName));
    return displayName;
}

```

ListManagers 方法为调用存储过程 TT_ListManagers 返回系统内部具有管理员权限的 DataSet，然后将 DataSet 中的数据转移成为一个 UsersCollection 对象返回。

```

public static UsersCollection ListManagers()
{
    string firstName = string.Empty;
    string lastName = string.Empty;

    //构建 DataSet
    DataSet ds = new DataSet();
    SqlHelper.ExecuteDataset(ConfigurationSettings.AppSettings[Web.Global.CfgKeyConnString],
                           CommandType.StoredProcedure, "TT_ListManagers");
    //实例化 UsersCollection 对象
    UsersCollection managersArray = new UsersCollection();

    //将 DataSet 中的数据转移成为 UsersCollection 对象
    foreach (DataRow r in ds.Tables[0].Rows)
    {
        //实例化一个 User 对象
        TTUser usr = new TTUser();
        usr.UserName = r["UserName"].ToString();
        usr.Role = r["RoleID"].ToString();
    }
}

```

```

        usr.UserID = Convert.ToInt32(r["UserID"]);
        usr.Name = GetDisplayName(usr.UserName, ref firstName, ref lastName);
        usr.FirstName = firstName;
        usr.LastName = lastName;
        //将 User 对象添加到 User 对象集合 UsersCollection 中
        managersArray.Add(usr);
    }
    return managersArray;
}

```

Remove 方法为调用存储过程 TT_DeleteUser 删除一个用户的方法。

```

public static void Remove (int userID)
{
    SqlHelper.ExecuteNonQuery(ConfigurationSettings.AppSettings[Web.Global.CfgKeyConnString],
                           "TT_DeleteUser", userID);
}

```

Load 方法调用存储过程 TT_GetUserByUserName，根据用户名获得用户信息，并将用户赋给 User 对象的成员变量，最终返回一个 bool 值常量表示对象成员变量赋值是否成功。

```

public bool Load ()
{
    //从数据中获得包含用户信息的 DataSet
    DataSet ds
    SqlHelper.ExecuteDataset(ConfigurationSettings.AppSettings[Web.Global.CfgKeyConnString],
                           "TT_GetUserByUserName", _userName);
    //如果 DataSet 为空则返回 false 值
    if (ds.Tables[0].Rows.Count < 1)
        return false;
    //取得 DataRow
    DataRow dr = ds.Tables[0].Rows[0];
    _userID = Convert.ToInt32(dr["UserID"]);
    _userName = dr["UserName"].ToString();
    _role = dr["RoleID"].ToString();
    _password = dr["Password"] == DBNull.Value ? "" : dr["Password"].ToString();
    //调用 GetDisplayName 方法传入 User 对象的私有成员变量(firstName 和 lastName 的引用值
    _displayName = GetDisplayName(_userName, ref _firstName, ref _lastName);
    //如果赋值成功则返回 true
    return true;
}

```

Save 方法调用其重载版本保存用户信息，注意该方法的参数中 User 对象的两个成员变量的引用值。

```

public bool Save ()
{
    bool isUserFound = false;
    bool isUserActiveManager = true;
    return Save(false, ref isUserFound, ref isUserActiveManager);
}

```

根据 User 对象的私有成员变量(userID 判断是向数据库中添加用户信息还是更新数据库中的用户信息。如果(userID 为 0，则调用 Insert 方法添加用户；如果(userID 不为 0，则调用 Update 方法更新用户信息。

```

public bool Save (bool checkUsername, ref bool isUserFound, ref bool isUserActiveManager)
{
}

```

```
//根据(userID 判断添加或是修改用户信息
if (_userID == 0)
    return Insert(checkUsername, ref isUserFound);
else if (_userID > 0)
    return Update(ref isUserActiveManager);
else
{
    _userID = 0;
    return false;
}
}
```

Insert 方法用于向数据库添加一条用户信息。

```
private bool Insert(bool checkUsername, ref bool isUserFound)
{
    string firstName = string.Empty;
    string lastName = string.Empty;
    isUserFound = false;
    //如果应用程序的用户数据源设置为 WindowsSAM 或 Active Directory
    if (ConfigurationSettings.AppSettings[Web.Global.CfgKeyUserAcctSource] != "None")
    {
        //根据 checkUsername 判断是否需要从 WindowsSAM 或 Active Directory 获得用户信息
        if (checkUsername)
        {
            TTUser.GetDisplayName(_userName, ref firstName, ref lastName);
            isUserFound = (firstName != string.Empty || lastName != string.Empty);
        }
    }
    else
    {
        checkUsername = false;
        isUserFound = true;
    }
    //如果应用程序的用户数据源设置为 WindowsSAM 或 Active Directory 并且获得了用户信息
    //则调用存储过程 TT_AddUser 向数据库中添加注册信息
    if ((checkUsername && isUserFound) || (!checkUsername))
    {
        _userID =
Convert.ToInt32(SqlHelper.ExecuteScalar(ConfigurationSettings.AppSettings[Web.Global.CfgKeyConnString],
"TT_AddUser",
    _userName, _password, _displayName, Convert.ToInt32(_role)));
        isUserFound = true;
    }
    return (_userID > 0);
}
```

Login 方法是用户登录之后判断用户身份的方法，如果通过验证方法，则返回的 string 型变量存储用户名；如果失败，则返回一个空的 string 型变量。

```
public string Login(string email, string password)
{
    string userName = string.Empty;
    //调用存储过程 TT_UserLogin 获得返回值
```

```

        userName = Convert.ToString(SqlHelper.ExecuteScalar(ConfigurationSettings.AppSettings[Web.Global.CfgKeyConnString], "TT_UserLogin", email, password));

        if (userName != "" || userName != string.Empty)
            return userName;
        else
            return string.Empty;

    }

```

Update 方法为更新用户信息的方法，如果用户角色是项目开发人员或某个开发中项目的项目经理，则不更新用户信息，否则更新用户信息。

```

private bool Update(ref bool isUserActiveManger)
{
    //判断用户角色
    if (_role == UserRoleConsultant &&
        (isUserActiveManger = Convert.ToInt32(SqlHelper.ExecuteScalar(ConfigurationSettings.AppSettings[Web.Global.CfgKeyConnString], "TT_GetManagerProjectCount", _userID)) == 0))
        return false;

    return Convert.ToInt32(SqlHelper.ExecuteScalar(ConfigurationSettings.AppSettings[Web.Global.CfgKeyConnString], "TT_UpdateUser",
        _userID, _userName, _password, _displayName, Convert.ToInt32(_role)));
}

```

(6) UsersCollection.cs

UsersCollection 类是继承自 System.Collections.ArrayList 类的 List User 对象集合类，用以存储一组 User 对象。其代码清单如下。

```

using System;
using System.Collections;

namespace ASPNET.StarterKit.TimeTracker.BusinessLogicLayer
{
    //继承自 System.Collections.ArrayList
    public class UsersCollection : ArrayList
    {
        public enum UserFields
        {
            InitValue,
            Name,
            RoleName
        }

        //重载 Sort 方法，对 User 对象的 Name 和 RoleName 属性排序
        public void Sort(UserFields sortField, bool isAscending)
        {
            switch (sortField)
            {

```

```

        case UserFields.Name:
            base.Sort(new NameComparer());
            break;
        case UserFields.RoleName:
            base.Sort(new RoleNameComparer());
            break;
    }
    if (!isAscending) base.Reverse();
}
//基于 IComparer 接口的类 NameComparer 比较两个对象的 Name 的属性
private sealed class NameComparer : IComparer
{
    public int Compare(object x, object y)
    {
        TTUser first = (TTUser) x;
        TTUser second = (TTUser) y;
        return first.Name.CompareTo(second.Name);
    }
}
//基于 IComparer 接口的类 RoleNameComparer 比较两个对象的 RoleName 的属性
private sealed class RoleNameComparer : IComparer
{
    public int Compare(object x, object y)
    {
        TTUser first = (TTUser) x;
        TTUser second = (TTUser) y;
        return first.RoleName.CompareTo(second.RoleName);
    }
}
}

```

(7) TTSecurity.cs

TTSecurity 类中包含了两个获取当前用户角色信息的方法和一些用户体系中用到的辅助方法。

```

using System;
using System.Web;
using System.Security.Cryptography;
using System.Text;
using System.Text.RegularExpressions;

namespace ASPNET.StarterKit.TimeTracker.BusinessLogicLayer
{
    public class TTSecurity
    {

```

IsInRole 方法调用 HttpContext.Current.User.IsInRole 方法判断用户是否属于某个角色。

```

        public static bool IsInRole(String role)
        {
            return HttpContext.Current.User.IsInRole(role);
        }

```

Encrypt 方法是使用 MD5 加密算法将字符串进行加密处理的方法，用于用户密码的加密处理。

```

        public static string Encrypt(string cleanString)

```

```

    {
        Byte[] clearBytes = new UnicodeEncoding().GetBytes(cleanString);
        Byte[] hashedBytes = ((HashAlgorithm) CryptoConfig.CreateFromName("MD5")).ComputeHash(clearBytes);

        return BitConverter.ToString(hashedBytes);
    }

```

GetUserID 方法返回 HttpContext.Current.User 对象的 UserID 属性。

```

public static int GetUserID()
{
    return ((CustomPrincipal)HttpContext.Current.User).UserID;
}

```

GetUserRole 方法返回 HttpContext.Current.User 对象的 UserRole 属性。

```

public static string GetUserRole()
{
    return ((CustomPrincipal)HttpContext.Current.User).UserRole;
}

```

GetName 方法返回 HttpContext.Current.User 对象的 Name 属性。

```

public static string GetName()
{
    return ((CustomPrincipal)HttpContext.Current.User).Name;
}

```

CleanStringRegex 方法过滤字符串中的特殊字符，保证字符串中的字符都在 a~z、A~Z、0~9 这些字符之中。

```

public static string CleanStringRegex(string inputText)
{
    RegexOptions options = RegexOptions.IgnoreCase;
    return ReplaceRegex(inputText, @"[^\\\.!?"", \-\w\s@]", options);
}

```

ReplaceRegex 方法为过滤掉字符串中指定的特殊字符，传入参数 RegexOptions 类型的 options 为指定的过滤条件。

```

private static string ReplaceRegex(string inputText, string regularExpression, RegexOptions options)
{
    Regex regex = new Regex(regularExpression, options);
    return regex.Replace(inputText, "");
}
}

```

14.3.3 应用层

(1) 用户注册

用户注册页面的后台编码类中捕捉前台按钮的单击事件，在该事件中调用逻辑层中的类库方法，完成用户注册。该事件的代码清单如下。

```

private void RegisterBtn_Click(object sender, System.EventArgs e)
{
    //判断是否通过页面数据验证
    if (Page.IsValid)
    {

```

```

//实例化 TTUser 类成 accountSystem 对象
TTUser accountSystem = new TTUser(0, Email.Text, TTSecurity.CleanStringRegex(DisplayName.Text),
ConfigurationSettings.AppSettings["DefaultRoleForNewUser"]);
accountSystem.Password = TTSecurity.Encrypt(Password.Text);
//调用 accountSystem 对象的 Save 方法添加用户
if (accountSystem.Save())
{
    //调用 FormsAuthentication 对象的 SetAuthCookie 方法设置用户通过身份验证
    FormsAuthentication.SetAuthCookie>Email.Text, false;

    //重定向页面
    Response.Redirect("TimeEntry.aspx");
}
else
{
    //如果注册失败显示注册失败信息
    Message.Text = "Registration Failed! <" + "u" + ">" + Email.Text + "<" + "/u" + "> is
already registered." + "<" + "br" + ">" + "Please register using a different email address.";
}
}

```

(2) 用户信息加载

ASP.NET Time Tracker Starter Kit 的用户信息加载过程是在应用程序的 Global.asax 文件 Application_AuthenticateRequest 事件中进行的，该事件的代码清单如下。

```

protected void Application_AuthenticateRequest(Object sender, EventArgs e)
{
    string userInformation = String.Empty;
    //判断用户是否通过验证
    if (Request.IsAuthenticated == true)
    {
        //判断 Cookies[UserRoles] 是否为空
        if ((Request.Cookies[UserRoles] == null) || (Request.Cookies[UserRoles].Value == ""))
        {
            //如果 Cookies[UserRoles] 为空则根据 User.Identity.Name 实例化 User 对象获得用户信息
            TTUser user = new TTUser(User.Identity.Name);
            if (!user.Load())
            {
                // 如果通过身份验证但是数据库中没有用户信息，则向数据库中添加用户信息
                TTUser newUser = new TTUser(0, Context.User.Identity.Name,
                    String.Empty, ConfigurationSettings.AppSettings[CfgKeyDefaultRole]);
                //保存用户信息
                newUser.Save();
                user = newUser;
            }
        }

        //构建保存用户信息的字符串
        userInformation = user.UserID + ";" + user.Role + ";" + user.Name;

        //建立 Forms 用户身份验证模式下保存用户信息的 Cookies
        FormsAuthenticationTicket ticket = new FormsAuthenticationTicket(
            1,                                     // version
            User.Identity.Name,                     // user name

```

```

        DateTime.Now,           // issue time
        DateTime.Now.AddHours(1), // expires every hour
        false,                  // don't persist cookie
        userInformation
    );

    //对用户身份信息 Cookies 进行加密
    String cookieStr = FormsAuthentication.Encrypt(ticket);

    //将用户身份信息 Cookies 发送到客户端浏览器
    Response.Cookies[UserRoles].Value = cookieStr;
    Response.Cookies[UserRoles].Path = "/";
    Response.Cookies[UserRoles].Expires = DateTime.Now.AddMinutes(1);

    //将自定义的用户身份对象 CustomPrincipal 赋值给存储当前用户信息的 Context.User 对象
    //Context.User 对象和 CustomPrincipal 一样都是继承自 Iprincipal 接口
    Context.User = new CustomPrincipal(User.Identity, user.UserID, user.Role, user.Name);
}

else
{
    //如果 Cookies[UserRoles]不为空，则可以从 Cookies 中获得用户身份信息
    FormsAuthenticationTicket ticket =
FormsAuthentication.Decrypt(Context.Request.Cookies[UserRoles].Value);
    userInformation = ticketUserData;

    //提取用户身份信息，建立自定义的用户身份对象 CustomPrincipal 并传递 Context.User 对象
    string[] info = userInformation.Split(new char[] { ';' });
    Context.User = new CustomPrincipal(
        User.Identity,
        Convert.ToInt32(info[0].ToString()),
        info[1].ToString(),
        info[2].ToString());
}
}
}
}

```

(3) 用户登录

ASP.NET Time Tracker Starter Kit 中用户控件 SingIn.ascx 为用户登录的用户控件，用户控件的后台编码类中捕捉用户登录按钮的单击事件，在该事件中验证用户登录。后台编码类中用户登录按钮的单击事件代码清单如下。

```

private void Btsignin_Click(object sender, System.EventArgs e)
{
    //实例化 TTUser 类为对象 accountSystem
    TTUser accountSystem = new TTUser();
    //调用 accountSystem 对象的 Login 方法
    string userId = accountSystem.Login(email.Text, TTSecurity.Encrypt(password.Text));

    if (userId != "" && userId != string.Empty)
    {
        //如果用户名密码匹配则调用 FormsAuthentication 对象的 SetAuthCookie 方法通过用户身份验证
    }
}

```

```

        FormsAuthentication.SetAuthCookie(email.Text, RememberCheckbox.Checked);
        //重定向页面
        FormsAuthentication.RedirectFromLoginPage(email.Text, false);
    }
    else
        //如果用户名密码不匹配则显示登录失败信息
        Message.Text = "<" + "br" + ">Login Failed!" + "<" + "br" + ">";
}

```

(4) 用户身份验证

当用户试图登录系统或需要验证用户身份信息时，系统会自动重定向到 Web.config 中 authentication 节点定义的验证页面。

```

<authentication mode="Forms">
    <forms name=".ASPAUTH" protection="All" timeout="60" loginUrl="desktopdefault.aspx" />
</authentication>

```

desktopdefault.aspx 页面的页面载入事件 Page_Load 中定义了如果用户通过身份验证则重定向到 TimeEntry.aspx，实际上 TimeEntry.aspx 才是 ASP.NET Time Tracker Starter Kit 的主页面。 desktopdefault.aspx 页面的页面载入事件 Page_Load 的代码如下。

```

private void Page_Load(object sender, System.EventArgs e)
{
    if (Request.IsAuthenticated)
        Response.Redirect("TimeEntry.aspx");
}

```

(5) 用户管理列表

在用户管理列表页面，管理员可以浏览系统用户列表，还可以单击浏览用户详细信息页面浏览用户详细信息。其后台编码类代码清单如下。

```

using System;
using System.Data;
using System.Web.UI.WebControls;
using ASPNET.StarterKit.TimeTracker.BusinessLogicLayer;

namespace ASPNET.StarterKit.TimeTracker.Web
{

    public class UserList : System.Web.UI.Page
    {
        //定义页面空间
        protected System.Web.UI.WebControls.Button btnNewProject;
        protected System.Web.UI.WebControls.DataGrid Users;
        protected System.Web.UI.WebControls.Button NewUserButton;

        //实例化一个 TTUser 对象
        protected TTUser _userInput = new TTUser(0, string.Empty, string.Empty, string.Empty);
    }
}

```

在页面载入事件中判断当前用户角色是否是管理员，如果当前用户不具备管理员权限，则定向到禁止访问的页面。如果当前页面未被提交则调用 LoadUsers 方法载入用户数据集信息到 DataGrid 控件。

```

private void Page_Load(object sender, System.EventArgs e)
{
    //判断访问者是否有权限访问当前页面
    if (TTSecurity.IsInRole(TTUser UserRoleAdministrator) == false)

```

```
{  
    //如果访问者没有权限，则定位到无浏览权限页面  
    Response.Redirect("AccessDenied.aspx?Index=-1", true);  
}  
  
if (!Page.IsPostBack)  
{  
    LoadUsers();  
}  
}
```

GetRoles 方法为调用类 Roles 返回系统中角色对象的对象集合。

```
protected RolesCollection GetRoles()  
{  
    return Roles.GetRoles();  
}
```

LoadUsers 方法实现的是获得用户对象集合类，并绑定到 DataGrid 控件。

```
private void LoadUsers()  
{  
    //调用 TTUser 对象 GetUsers 返回用户对象集合 users  
    UsersCollection users = TTUser.GetUsers(TTSecurity.GetUserID(), TTSecurity.GetUserRole());  
  
    //为用户对象集合 users 排序  
    SortGridData(users, SortField, SortAscending);  
    //设置 DataGrid 控件的数据源为 Users 并绑定 DataGrid 控件  
    Users.DataSource = users;  
    Users.DataKeyField = "UserID";  
    Users.DataBind();  
}
```

SortGridData 方法是根据字段为用户对象集合类排序的方法，在 LoadUsers 方法中被调用。

```
private void SortGridData(UsersCollection list, string sortField, bool asc)  
{  
    //获得 UsersCollection 对象 UserFields  
    UsersCollection.UserFields sortCol = UsersCollection.UserFields.InitValue;  
    //根据不同字段对 UsersCollection 对象中的 User 对象进行排序  
    switch(sortField)  
    {  
        case "Name":  
            sortCol = UsersCollection.UserFields.Name;  
            break;  
        case "RoleName":  
            sortCol = UsersCollection.UserFields.RoleName;  
            break;  
    }  
    //为用户对象集合类 list 排序  
    list.Sort(sortCol, asc);  
}  
  
override protected void OnInit(EventArgs e)  
{  
    InitializeComponent();  
}
```

```

        base.OnInit(e);
    }

    private void InitializeComponent()
    {
        this.NewUserButton.Click += new System.EventHandler(this.NewUserButton_Click);
        this.UsersPageIndexChanged += new System.Web.UI.WebControls.DataGridPageChangedEventHandler(this.
Users_PageIndexChanged);
        this.Users.SortCommand += new System.Web.UI.WebControls.DataGridSortCommandEventHandler
(this.Users_Sort);
        this.Load += new System.EventHandler(this.Page_Load);

    }

```

Users_Sort 方法为 DataGrid 控件的排序事件，在该事件中获得排序字段调用 LoadUsers 方法绑定经过排序的用户控件。

```

private void Users_Sort(object source, System.Web.UI.WebControls.DataGridSortCommandEventArgs e)
{
    //捕捉排序条件
    SortField = e.SortExpression;
    LoadUsers();
}

```

Users_PageIndexChanged 方法为 DataGrid 控件的分页跳转事件，在该事件中获得跳转页面的页码，调用 LoadUsers 方法重新绑定 DataGrid 控件。

```

private void Users_PageIndexChanged(object source, System.Web.UI.WebControls.
DataGridPageChangedEventArgs e)
{
    Users.EditItemIndex= -1;
    Users.CurrentPageIndex = e.NewPageIndex;
    LoadUsers();
}

```

NewUserButton_Click 方法为新增用户按钮单击事件，在该事件中定义了跳转新增用户页面的 URL。

```

private void NewUserButton_Click(object sender, System.EventArgs e)
{
    int mainIndex = (Request["index"]==null) ? 0 : Convert.ToInt32(Request["index"]);
    int adminIndex = (Request["adminindex"]==null) ? 0 : Convert.ToInt32(Request["adminindex"]);
    Response.Redirect(String.Format("UserDetail.aspx?index={0}&adminindex={1}", mainIndex, adminIndex),
false);
}

```

SortField 构造器从 ViewState 中读取和存储 SortField 的值。

```

string SortField
{
    get
    {
        //如果 ViewState 中 SortField 字段为空则返回空字符串
        object o = ViewState["SortField"];
        if (o == null)
            return string.Empty;

        return (string)o;
    }
}

```

```
    }

    set
    {
        //如果请求排序字段已经存储在 ViewState["SortField"] 中则改变排序方式 SortAscending
        if (value == SortField)
            SortAscending = !SortAscending;

        ViewState["SortField"] = value;
    }
}
```

SortAscending 构造器从 ViewState 中读取和存储 SortAscending 的值。

```
bool SortAscending
{
    get
    {
        //如果 ViewState 中的 SortAscending 字段为空则返回 true
        object o = ViewState["SortAscending"];
        if (o == null)
            return true;

        return (bool)o;
    }

    set
    {
        ViewState["SortAscending"] = value;
    }
}
}
```

(6) 用户详细信息页面

在用户详细信息页面 UserDetail.aspx 可以浏览用户信息、添加用户信息、修改用户信息。其后台编码类代码如下。

```
using System;
using System.Data;
using System.Web.UI.WebControls;
using ASPNET.StarterKit.TimeTracker.BusinessLogicLayer;

namespace ASPNET.StarterKit.TimeTracker.Web
{
    public class UserDetail : System.Web.UI.Page
    {
        //定义页面控件
        protected System.Web.UI.WebControls.Button Save;
        protected System.Web.UI.WebControls.Button Cancel;
        protected System.Web.UI.WebControls.TextBox UserName;
        protected System.Web.UI.WebControls.RequiredFieldValidator RequiredFieldValidator1;
        protected System.Web.UI.WebControls.DropDownList Roles;
```

```
protected System.Web.UI.WebControls.Label lblError;
protected System.Web.UI.WebControls.Label ErrorLabel;
protected System.Web.UI.WebControls.Label ErrorLabel2;
protected System.Web.UI.WebControls.Label userNamelbl;
protected System.Web.UI.WebControls.Label rolelbl;
protected System.Web.UI.WebControls.Label displayNamelbl;
protected System.Web.UI.WebControls.TextBox DisplayName;
protected System.Web.UI.WebControls.Label passwordlbl;
protected System.Web.UI.WebControls.TextBox Password;
protected System.Web.UI.WebControls.RequiredFieldValidator RequiredFieldValidator3;
protected System.Web.UI.WebControls.Label confirmPasswordlbl;
protected System.Web.UI.WebControls.TextBox ConfirmPassword;
protected System.Web.UI.WebControls.RequiredFieldValidator RequiredFieldValidator4;
protected System.Web.UI.WebControls.CompareValidator CompareValidator1;

//实例化类 TTUser
private TTUser user = new TTUser();
```

根据页面载入事件 Page_Load 判断当前用户角色是否为管理员，如果为管理员，则重定向页面到禁止访问页面。如果页面未被提交，则调用 BindInfo 方法加载所要编辑的用户信息。如果当前所做操作为添加用户，则隐藏在编辑用户时使用的控件。

```
private void Page_Load(object sender, System.EventArgs e)
{
    //判断当前用户角色
    if (TTSecurity.IsInRole(TTUser UserRoleAdministrator) == false)
    {
        Response.Redirect("AccessDenied.aspx?Index=-1", true);
    }
    //判断页面是否被提交
    if (!Page.IsPostBack)
    {
        //如未提交则调用 BindInfo 方法加载所要编辑的用户信息
        BindInfo();

        //如果新增用户则隐藏部分编辑用户时使用的控件
        if ((user.Password == "" && user.UserID != 0) || (Context.User.Identity.AuthenticationType != "Forms" && user.UserID==0))
        {
            displayName.Visible = false;
            DisplayName.Visible = false;
            passwordlbl.Visible = false;
            Password.Visible = false;
            confirmPasswordlbl.Visible = false;
            ConfirmPassword.Visible = false;
            RequiredFieldValidator3.Visible= false;
            RequiredFieldValidator4.Visible= false;
            userNamelbl.Text = @"Username (Use the format: DOMAIN or Computer Name\Username )";
        }
    }
}
```

BindInfo 方法为根据 URL 中的查询字符串 UserName 来加载用户信息的方法。

```
private void BindInfo()
{
    //获取 URL 中查询字符串 UserName 的值
    string User = Request.QueryString["UserName"];

    //绑定用户角色下拉列表框
    Roles.DataSource = BusinessLogicLayer.Roles.GetRoles();
    Roles.DataTextField = "Name";
    Roles.DataValueField = "RoleID";
    Roles.DataBind();

    //如果 URL 中查询字符串 UserName 的值不为空则加载用户信息
    if (User != null)
    {
        user.UserName = User;
        user.Load();
        UserName.Text = user.UserName;
        UserName.ReadOnly = true;
        Roles.Items.FindByValue(user.Role).Selected = true;
        DisplayName.Text = user.DisplayName;
    }
    else
    {
        UserName.ReadOnly = false;
    }
}

override protected void OnInit(EventArgs e)
{
    InitializeComponent();
    base.OnInit(e);
}

private void InitializeComponent()
{
    this.Save.Click += new System.EventHandler(this.Save_Click);
    this.Cancel.Click += new System.EventHandler(this.Cancel_Click);
    this.Load += new System.EventHandler(this.Page_Load);
}
```

Save_Click 方法为保存信息按钮的单击事件，该事件中调用 user 对象的 Save 方法完成对用户信息的保存。

```
private void Save_Click(object sender, System.EventArgs e)
{
    bool isUserFound = false;
    bool isUserActiveManager = false;

    //获取页面提交 Save_Click
    user.UserName = TTSecurity.CleanStringRegex(UserName.Text);
```

```
//调用 user 对象的 Load 方法加载用户信息
user.Load();
user.Role = Roles.SelectedItem.Value;
user.DisplayName = TTSecurity.CleanStringRegex(DisplayName.Text);
if (Password.Text != string.Empty)
    user.Password = TTSecurity.Encrypt(Password.Text);
//调用 user 对象的 Save 方法保存用户信息
user.Save(true, ref isUserFound, ref isUserActiveManager);

//如果用户数据源为 WindowsSAM 或活动目录而当前添加用户没有被找到，则显示错误提示信息
if (user.UserID == 0 && !isUserFound)
    ErrorLabel.Text = "The username was not found on your system or network.";
else if (isUserActiveManager)
    ErrorLabel.Text = "User Role can not be changed, user is a manager of one or more projects.";

else
    //添加或保存用户信息成功重定向页面到 UserList.aspx
    Response.Redirect("UserList.aspx?index=2&adminIndex=1", false);
}
```

Cancel_Click 方法为取消按钮单击事件，当用户单击取消按钮则重定向页面到 UserList.aspx。

```
private void Cancel_Click(object sender, System.EventArgs e)
{
    Response.Redirect("UserList.aspx?index=2&adminIndex=1", false);
}
```

14.3.4 表示层

ASP.NET Time Tracker Starter Kit 用户体系中前台表示层页面结构如图 14-3 所示。

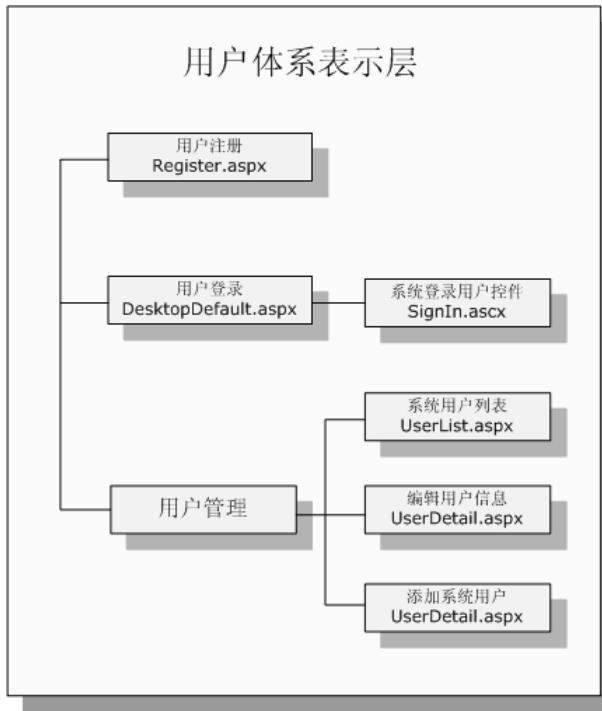


图 14-3 ASP.NET Time Tracker Starter Kit 用户体系的表示层

大部分页面的表示层没有和系统业务逻辑相耦合，仅仅是页面控件的罗列。其中用户列表页面中的用户列表 DataGrid 控件绑定了数据源，其实现代码如下。

```

<asp:datagrid id="Users" runat="server" BorderStyle="None" Width="100%" CellPadding="2" AutoGenerateColumns="False"
=AllowSorting="True" BorderColor="White" AllowPaging="True">
    <headerstyle font-bold="True" cssclass="grid-header"></headerstyle>
    <columns>
        <!-- 用户名模板列 -->
        <asp:templatecolumn HeaderText="Name (click name to edit)" SortExpression="Name">
            <headerstyle horizontalalign="Left" width="80%" cssclass="grid-header" verticalalign="Middle"></headerstyle>
            <itemstyle cssclass="grid-first-item"></itemstyle>
            <!-- 用户名模板列显示模板 -->
            <itemtemplate>
                <a title="Edit User" class="blacklineheight" href='UserDetail.aspx?UserName=<%# DataBinder.Eval(Container, "DataItem.UserName") %>&index=2'><%# DataBinder.Eval(Container, "DataItem.Name") %></a>
            </itemtemplate>
        </asp:templatecolumn>
        <!-- 用户角色模板列 -->
        <asp:templatecolumn HeaderText="Role" SortExpression="RoleName">
            <headerstyle horizontalalign="Left" width="20%" cssclass="grid-header" verticalalign="Middle"></headerstyle>
            <itemstyle cssclass="grid-item"></itemstyle>
            <!-- 用户角色模板列显示模板 -->
            <itemtemplate>
                <asp:label ID="lblGridRoleName" Text='<%# DataBinder.Eval(Container, "DataItem.RoleName") %>'></asp:label>
            </itemtemplate>
        </asp:templatecolumn>
    </columns>
</asp:datagrid>
  
```

```

%>' Runat="server" Visible="true" />
</itemtemplate>
<!-- 用户角色模板列编辑模板 -->
<edititemtemplate>
    <asp:DropDownList Width="100px" ID="ddlRoles" CssClass="Standard-text" DataSource ='<%# GetRoles() %>' DataTextField="Name" DataValueField="RoleID" Runat="server" />
</edititemtemplate>
</asp:templatecolumn>
<asp:templatecolumn Visible="False" HeaderText="UserID">
    <itemtemplate>
        <asp:label ID="lblGridUserID" Text='<%# DataBinder.Eval(Container, "DataItem.UserID") %>' Runat="server" Visible="true" />
    </itemtemplate>
</asp:templatecolumn>
<asp:templatecolumn Visible="False" HeaderText="UserName">
    <itemtemplate>
        <asp:label ID="lblGridUserName" Text='<%# DataBinder.Eval(Container, "DataItem.UserName") %>' Runat="server" Visible="true" />
    </itemtemplate>
</asp:templatecolumn>
</columns>
<pagerstyle horizontalalign="Center"></pagerstyle>
</asp:datagrid>

```

用户管理列表页面的效果如图 14-4 所示。

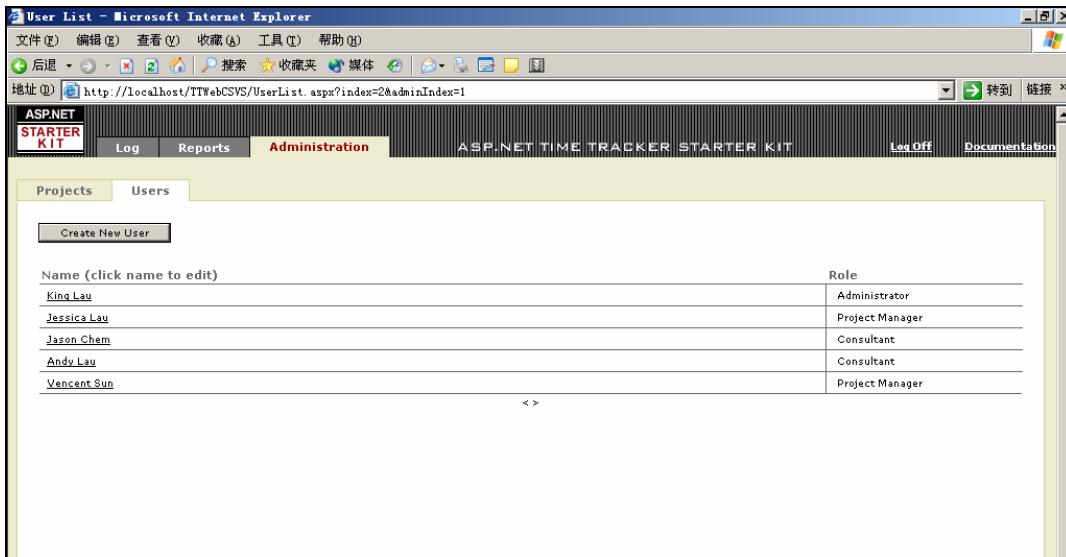


图 14-4 用户管理列表页面的效果

14.4 开发启示

在本章中讲解了 ASP.NET Time Tracker Starter Kit 的用户体系，在设计结构上这个用户体系略显简单，但是在技术实现途径上 ASP.NET Time Tracker Starter Kit 的用户体系提供了一套非常严密的技

术框架。以下是 ASP.NET Time Tracker Starter Kit 的用户体系值得学习和借鉴的几个技术点。

- 基于角色的权限分配
- 基于表单的身份验证
- 可选择的用户数据源
- 支持 WindowsSAM 和活动目录的用户数据源

第 15 章

项目管理与 工作进程管理

ASP.NET Time Tracker Starter Kit 的主要功能是管理项目和管理工作进程，本章将详细讲解项目管理与进程管理的实现。项目管理包括新建项目、配置项目信息、设置项目工作分类等功能，进程管理包括新增工作进程、修改工作进程、查看当周工作进程等功能。

15.1 系统设计

15.1.1 功能设计

在功能设计上，本章分为两大部分内容：项目管理与进程管理。项目是 ASP.NET Time Tracker Starter Kit 中最主体的对象，工作进程是属于项目的，是项目实施的具体组成。

项目管理只有具有项目经理和系统管理人角色的用户才能使用，其中包括以下功能。

(1) 项目列表

项目经理和管理员可以在项目管理模块中查看项目列表，项目经理可以查看所负责的项目列表，系统管理员可以查看系统内所有项目的列表，项目列表中显示的信息包括项目名称、项目经理、项目完成日期、项目总工作量。项目列表页面的效果如图 15-1 所示。

(2) 新增项目

项目经理和管理员可以选择新建项目，在新建项目中用户必须为项目指定工作分类，一个项目至少要有一个类别。项目的类别可以是用户新创建的，也可以是继承其他项目的。另外新建项目中要指定项目的名称、项目经理（如果用户角色为项目经理，则该项为用户本身不得修改）、项目结束日期、项目总工作量、项目人员项目描述等信息。新建的项目页面如图 15-2 所示。

(3) 修改项目信息

项目经理和管理员可以在项目列表页面选择修改项目，进入项目信息修改页面，可以修改包括项目名称、项目人员和项目分类在内的全部项目信息。

进程管理包括新建工作进程、修改工作进程和删除工作进程 3 个方面的功能，所有的用户都具有使用进程管理模块的权限。项目开发人员只具备安排个人工作进程的权限，项目经理具有为所

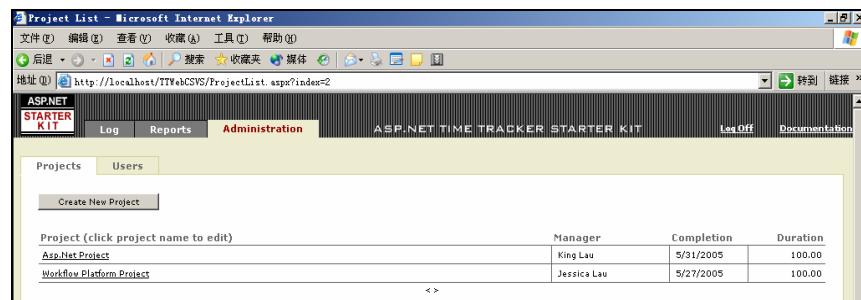


图 15-1 项目列表页面

Project Configuration

Define the project and specify which users will be part of the project. Then add categories to the project to help keep track of specific areas of product. Press the **SAVE** button for configuration to take effect.

PROJECT INFORMATION

Project Name:
Description:
Project Manager:
Est.Completion Date: Est.Duration: hrs

SPECIFY PROJECT MEMBERS

Project Members (CTRL+Click for multiples):
King Lau
Jessica Lau
Jason Chem
Andy Lau
Vincent Sun

DEFINE PROJECT CATEGORIES FOR PROJECT MANAGEMENT

Categories can be added in two ways. You can **ADD** a category by specifying name, abbreviation (4 characters max), and duration - the amount of hours that may be charged under the category. Or, You can **COPY** categories that already have been defined in another project to this project.

Name: Abbrev.: Duration(hrs): Add OR Add Categories From Another Project: Copy

List of Project Categories:

图 15-2 新建项目页面

负责项目中所有人员安排项目进程的权限，系统管理员具有管理和安排系统中所有人员工作进程的权限。进程管理的全部功能都可以在 TimeEntity.aspx 页面内完成。该页面效果如图 15-3 所示。

(4) 新增工作进程

系统用户可以在工作进程管理页面中新增工作进程区域（图 15-3 B 区域）添加项目的工作进程，指定项目、分类、日期、工作量、工作描述等工作进程信息。

(5) 查看工作进程

系统用户可以以星期为单位查看工作进程，工作进程列表会显示在工作进程管理页面的工作进程列表区域（图 15-3 D 区域），通过选择星期结束日期（图 15-3 A 区域）查看不同星期的工作进程列表。项目开发人员仅可以查看自己的工作进程列表，项目经理可以查看所负责项目中所有人员的工作进程，管理员可以查看系统内所有用户的工作进程。

(6) 修改工作进程

系统用户可以修改有浏览权限的所有工作进程，包括进程所在项目、进程分类、日期、工作量和描述等信息。系统用户也可以删除工作进程。

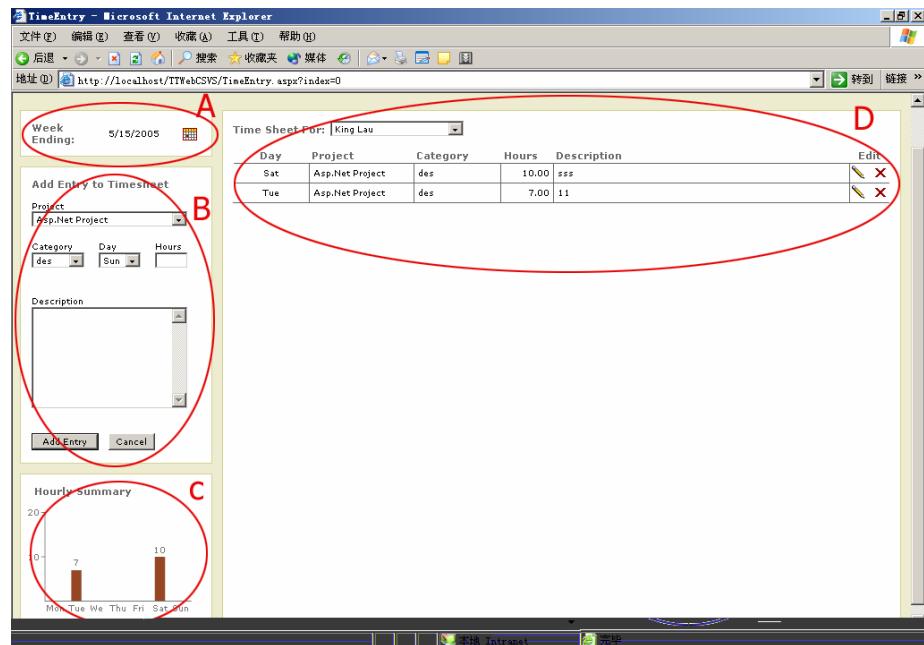


图 15-3 进程管理页面

15.1.2 数据结构

项目管理和进程管理主要使用到了项目表 TT_Project、工作进程表 TT_EntryLog、工作分类表 TT_Categories 和项目人员表 TT_ProjectMembers。图 15-4 描述了 4 个表的字段和表与表之间的关系。

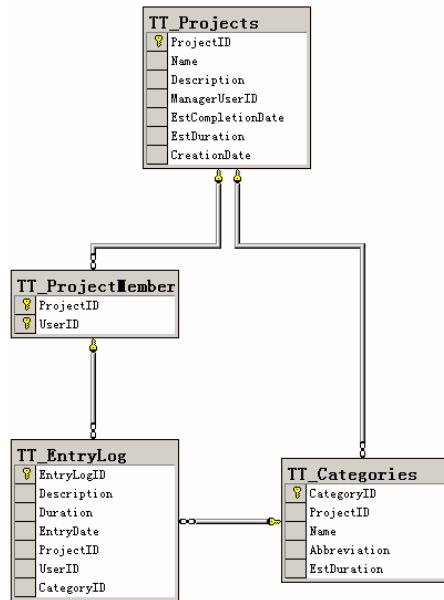


图 15-4 数据结构

项目管理和进程管理使用到的存储过程如表 15-1 所示。

表 15-1

项目管理和进程管理使用到的存储过程

| 存储过程名 | 说 明 |
|----------------------------|-----------------|
| AddProject | 添加项目 |
| GetProject | 获得指定项目的信息 |
| UpdateProject | 更新指定项目的信息 |
| ListAllProjects | 获得所有项目列表 |
| ListProjects | 根据角色信息获得所有项目列表 |
| ListProjectsWithMembership | 获得某个用户的所有参与项目列表 |
| ListCategories | 获得指定项目的所有工作进程分类 |
| DeleteProject | 删除项目 |
| AddTimeEntry | 添加工作进程 |
| DeleteTimeEntry | 删除工作进程 |
| GetTimeEntry | 获得指定工作进程信息 |
| ListTimeEntries | 获得某个项目所有工作进程列表 |
| UpdateTimeEntry | 更新工作进程 |

15.2 关键技术点

15.2.1 DataGrid 的高级应用

DataGrid 作为 Asp.NET 中功能最为强大的数据展现控件，能够完成包括数据的绑定、数据的分页分行分列展现、数据的修改、数据的删除等一系列相关功能。DataGrid 的数据源可以是 DataSet、 DataView、DataTable 等多种数据载体。

DataGrid 本身支持多种事件，例如 DataGrid_ItemBound、DataGrid_OnEdit、DataGrid_OnDelete 等等。这些事件都是对 DataGrid 进行特定操作之后触发的。表 15-2 是 DataGrid 的所有常用事件以及说明。

表 15-2

DataGrid 的所有常用事件

| 事 件 | 说 明 |
|------------------|-----------------------------------|
| CancelCommand | 对 DataGrid 控件中的某个项单击 Cancel 按钮时发生 |
| DataBinding | 当 DataGrid 控件绑定到数据源时发生 |
| DeleteCommand | 对 DataGrid 控件中的某个项单击 Delete 按钮时发生 |
| EditCommand | 对 DataGrid 控件中的某个项单击 Edit 按钮时发生 |
| ItemCommand | 当单击 DataGrid 控件中的任一按钮时发生 |
| ItemCreated | 当在 DataGrid 控件中创建项时在服务器上发生 |
| ItemDataBound | 在项被数据绑定到 DataGrid 控件后发生 |
| PageIndexChanged | 当单击 DataGrid 控件页选择元素之一时发生 |
| SortCommand | 对 DataGrid 控件中的列进行排序时发生 |
| UpdateCommand | 对 DataGrid 控件中的某个项单击 Update 按钮时发生 |

15.2.2 使用 ViewState 存储页面内部变量

Asp.NET 页面内部的任何服务器端控件，包括页面本身都有 ViewState 属性，该属性可以在同一页的多个请求之间保存和还原服务器端控件的视图状态。简单地理解为 ViewState 属性可以存储一些自定义的信息在同一个页面中。

下面的示例说明从页面的 ViewState 属性存储和检索值名为 Text 的值。

```
public String Text {  
    get {  
        return (String) ViewState["Text"];  
    }  
    set {  
        ViewState["Text"] = value;  
    }  
}
```

15.3 项目管理

15.3.1 数据库设计

项目管理部分使用到的数据存储过程如下。

(1) AddProject

存储过程 AddProject 为添加项目的存储过程，添加的项目信息包括项目基本信息、项目人员信息和项目分类信息。因此涉及到向不同数据表添加彼此存在关系的数据，该存储过程使用了 SQL Server 中的事务功能，当添加项目信息的过程出现错误时将实现自动回滚，保证不会影响到数据的完整性或产生错误数据。AddProject 的代码清单如下。

```
CREATE PROCEDURE TT_AddProject  
(  
    @Name nvarchar(50),  
    @Description nvarchar(1024),  
    @ManagerUserID int,  
    @EstCompletionDate datetime,  
    @EstDuration int,  
    @Members nvarchar(2000),  
    @Categories nvarchar(4000)  
)  
AS  
/* 定义存储过程中使用的变量 */  
DECLARE @Error int  
DECLARE @ProjectID int  
DECLARE @TempString varchar(4000)  
DECLARE @Temp nvarchar(4000)  
DECLARE @Count int  
DECLARE @TempTable TABLE(UserID int PRIMARY KEY)  
DECLARE @InnerTemp nvarchar(50)
```

```
DECLARE @CatName varchar(20)
DECLARE @Abbrev varchar(5)
DECLARE @Duration decimal(10, 2)
DECLARE @InnerCount int
/* 开始事务 */
BEGIN TRANSACTION
/* 向 TT_Projects 添加数据 */
INSERT INTO TT_Projects
(
    [Name],
    [Description],
    ManagerUserID,
    EstCompletionDate,
    EstDuration,
    CreationDate
)
VALUES
(
    @Name,
    @Description,
    @ManagerUserID,
    @EstCompletionDate,
    @EstDuration,
    getdate()
)
/* 判断是否出错, 如果出错则跳转到错误处理模块 */
SET @Error = @@ERROR
IF @Error != 0 GOTO ERROR_HANDLER
/* 获取项目 ProjectID */
SET @ProjectID = @@Identity
/* 获得项目成员字符串, 并添加项目成员 */
SET @TempString = @Members

SET @Count = CHARINDEX(',', @TempString)

WHILE @Count > 0
BEGIN
    SET @Temp = SUBSTRING(@TempString, 1, (@Count - 1))
    INSERT INTO @TempTable VALUES(CAST(@Temp AS int))
    SET @TempString = SUBSTRING(@TempString, (@Count + 1), (LEN(@TempString) - @Count))
    SET @Count = CHARINDEX(',', @TempString)
END

INSERT INTO @TempTable VALUES(CAST(@TempString AS int))

INSERT INTO TT_ProjectMembers
    SELECT @ProjectID, UserID FROM @TempTable
/* 判断是否出错, 如果出错则跳转到错误处理模块 */
SET @Error = @@ERROR
IF @Error != 0 GOTO ERROR_HANDLER
```

```
/* 获得并分隔项目工作分类字符串，并添加项目工作分类 */
SET @TempString = @Categories
/* 含有“;”符号说明项目分类为多个，提取每一个分类之后逐个添加 */
SET @Count = CHARINDEX(';', @TempString)
WHILE @Count > 0
BEGIN
    SET @Temp = SUBSTRING(@TempString, 1, (@Count - 1))

    SET @InnerCount = CHARINDEX(' ', @temp)

    SET @InnerTemp = SUBSTRING(@temp, 1, (@InnerCount - 1))
    SET @CatName = @InnerTemp

    SET @temp = SUBSTRING(@temp, (@InnerCount + 1), (LEN(@temp) - @InnerCount))
    SET @InnerCount = CHARINDEX(' ', @temp)

    SET @InnerTemp = SUBSTRING(@temp, 1, (@InnerCount - 1))
    SET @Abbrev = @InnerTemp

    SET @temp = SUBSTRING(@temp, (@InnerCount + 1), (LEN(@temp) - @InnerCount))
    SET @InnerCount = CHARINDEX(' ', @temp)

    SET @duration = CAST(@temp AS int)

    INSERT INTO TT_CATEGORIES
    (
        ProjectID,
        [Name],
        Abbreviation,
        EstDuration
    )
    VALUES
    (
        @ProjectID,
        @CatName,
        @Abbrev,
        @Duration
    )

    SET @Error = @@ERROR
    IF @Error != 0 GOTO ERROR_HANDLER

    SET @TempString = SUBSTRING(@TempString, (@Count + 1), (LEN(@TempString) - @Count))
    SET @Count = CHARINDEX(';', @TempString)

END
/* 当项目分类为一个的时候，直接添加该项目分类 */
set @temp = @tempstring
SET @InnerCount = CHARINDEX(' ', @temp)
```

```

SET @InnerTemp = SUBSTRING(@temp, 1, (@InnerCount - 1))
SET @CatName = @InnerTemp

SET @temp = SUBSTRING(@temp, (@InnerCount + 1), (LEN(@temp) - @InnerCount))
SET @InnerCount = CHARINDEX(',', @temp)

SET @InnerTemp = SUBSTRING(@temp, 1, (@InnerCount - 1))
SET @Abbrev = @InnerTemp

SET @temp = SUBSTRING(@temp, (@InnerCount + 1), (LEN(@temp) - @InnerCount))
SET @InnerCount = CHARINDEX(',', @temp)

SET @duration = CAST(@temp AS int)

INSERT INTO TT_CATEGORIES
(
    ProjectID,
    [Name],
    Abbreviation,
    EstDuration
)
VALUES
(
    @ProjectID,
    @CatName,
    @Abbrev,
    @Duration
)

SET @Error = @@ERROR
IF @Error != 0 GOTO ERROR_HANDLER
/* 事务结束 */
COMMIT TRANSACTION

SELECT @ProjectID AS ProjectID
/* 定义错误处理 */
ERROR_HANDLER:
IF @@TRANCOUNT != 0 ROLLBACK TRANSACTION
RETURN @Error

```

(2) GetProject

存储过程 GetProject 是获得指定项目信息的存储过程，首先获得项目主体信息，然后通过调用存储过程 TT_ListMembers 和存储过程 TT_ListCategoriesByProject 获得项目的成员信息和项目工作分类信息。

```

CREATE PROCEDURE TT_GetProject
(
    @ProjectID int
)
AS

SELECT

```

```
name, Description, ManagerUserID, EstCompletionDate, EstDuration

FROM
    TT_Projects

WHERE
    ProjectID = @ProjectID
/* 调用 TT_ListMembers 获得项目成员信息 */
exec TT_ListMembers @ProjectID
/* 调用 TT_ListCategoriesByProject 获得项目工作分类信息 */
exec TT_ListCategoriesByProject @ProjectID
```

(3) UpdateProject

存储过程 UpdateProject 为更新项目信息的存储过程，该存储过程和 AddProject 类似，使用数据库事务更新包括项目主体信息、项目成员信息、项目工作分类信息在内的项目信息，其中项目成员信息和项目工作分类信息通过分隔字符串的方式获得。

```
CREATE PROCEDURE TT_UpdateProject
(
    @ProjectID int,
    @Name nvarchar(50),
    @Description nvarchar(1024),
    @ManagerUserID int,
    @EstCompletionDate datetime,
    @EstDuration int,
    @SelectedMembers nvarchar(2000),
    @Categories nvarchar(4000)
)
AS
/* 定义存储过程使用变量 */
DECLARE @Error int
DECLARE @TempTable TABLE(UserID int PRIMARY KEY)
DECLARE @TempString varchar(2000)
DECLARE @Temp varchar(4000)
DECLARE @Count int
DECLARE @TempCatTable TABLE(CategoryID int primary key)
DECLARE @InnerTemp nvarchar(50)
DECLARE @CatID int
DECLARE @CatName varchar(20)
DECLARE @Abbrev varchar(5)
DECLARE @Duration int
DECLARE @InnerCount int
/* 开始数据库事务 */
BEGIN TRANSACTION

/* 更新项目主体信息 */
UPDATE
    TT_Projects
SET
    Name=@Name,
    Description = @Description,
    ManagerUserID = @ManagerUserID,
```

```
EstCompletionDate = @EstCompletionDate,
EstDuration = @EstDuration

WHERE
    ProjectID = @ProjectID
    /* 判断是否出错, 如果出错则跳转到错误处理模块 */
    SET @Error = @@ERROR
    IF @Error != 0 GOTO ERROR_HANDLER

    SET @TempString = @SelectedMembers
    /* 获得并分隔项目成员的字符串, 更新项目成员表 */
    SET @Count = CHARINDEX(',', @TempString)

    WHILE @Count > 0
    BEGIN
        SET @Temp = SUBSTRING(@TempString, 1, (@Count - 1))
        INSERT INTO @TempTable VALUES(CAST(@Temp AS int))
        SET @TempString = SUBSTRING(@TempString, (@Count + 1), (LEN(@TempString) - @Count))
        SET @Count = CHARINDEX(',', @TempString)
    END

    INSERT INTO @TempTable VALUES(CAST(@TempString AS int))

    DELETE
        TT_ProjectMembers
    WHERE
        ProjectID = @ProjectID
        AND UserID NOT IN(SELECT UserID FROM @TempTable)

    INSERT INTO TT_ProjectMembers
        SELECT @ProjectID, UserID FROM @TempTable WHERE UserID NOT IN
            (SELECT UserID FROM TT_ProjectMembers WHERE ProjectID = @ProjectID)
        /* 判断是否出错, 如果出错则跳转到错误处理模块 */
        SET @Error = @@ERROR
        IF @Error != 0 GOTO ERROR_HANDLER
        /* 获得并分隔项目工作分类字符串, 并更新项目工作分类 */

        SET @TempString = @Categories
        /* 含有:符号说明项目分类为多个, 分隔每一个工作分类然后更新 */
        SET @Count = CHARINDEX(';', @TempString)
        WHILE @Count > 0
        BEGIN
            SET @Temp = SUBSTRING(@TempString, 1, (@Count - 1))

            SET @InnerCount = CHARINDEX(',', @temp)

            SET @InnerTemp = SUBSTRING(@temp, 1, (@InnerCount - 1))
            SET @CatID = @InnerTemp

            SET @temp = SUBSTRING(@temp, (@InnerCount + 1), (LEN(@temp) - @InnerCount))
```

```
SET @InnerCount = CHARINDEX(' ', @temp)

SET @InnerTemp = SUBSTRING(@temp, 1, (@InnerCount - 1))
SET @CatName = @InnerTemp

SET @temp = SUBSTRING(@temp, (@InnerCount + 1), (LEN(@temp) - @InnerCount))
SET @InnerCount = CHARINDEX(' ', @temp)

SET @InnerTemp = SUBSTRING(@temp, 1, (@InnerCount - 1))
SET @ABrev = @InnerTemp

SET @temp = SUBSTRING(@temp, (@InnerCount + 1), (LEN(@temp) - @InnerCount))
SET @InnerCount = CHARINDEX(' ', @temp)

SET @duration = CAST(@temp AS int)
/* 如果需要更新的工作分类并不存在，则添加该项目工作分类 */
IF not exists(SELECT CategoryID from TT_Categories WHERE ProjectID = @projectID AND CategoryID
= @catID)
BEGIN

    INSERT INTO TT_Categories
    (
        ProjectID,
        [Name],
        Abbreviation,
        EstDuration
    )
    VALUES
    (
        @ProjectID,
        @CatName,
        @ABrev,
        @Duration
    )

    SET @Error = @@ERROR
    IF @Error != 0 GOTO ERROR_HANDLER

    SELECT @CatID = @@IDENTITY

END
/* 如果需要更新的工作分类存在，则更新该项目工作分类 */
ELSE
BEGIN
    UPDATE
        TT_Categories
    SET
        [Name] = @CatName,
        Abbreviation = @ABrev,
        EstDuration = @Duration

```

```
WHERE
    ProjectID = @projectId
    AND
        CategoryID = @catID
/* 判断是否出错, 如果出错则跳转到错误处理模块 */
    SET @Error = @@ERROR
    IF @Error != 0 GOTO ERROR_HANDLER
END

INSERT INTO @TempCatTable VALUES(@CatID)

SET @TempString = SUBSTRING(@TempString, (@Count + 1), (LEN(@TempString) - @Count))
SET @Count = CHARINDEX(';', @TempString)
END
/* 当所要更新的项目分类为一个的时候, 获得该项目工作分类的各个字段然后更新 */
set @temp = @tempstring
SET @InnerCount = CHARINDEX(',', @temp)

SET @InnerTemp = SUBSTRING(@temp, 1, (@InnerCount - 1))
SET @CatID = @InnerTemp

SET @temp = SUBSTRING(@temp, (@InnerCount + 1), (LEN(@temp) - @InnerCount))
SET @InnerCount = CHARINDEX(',', @temp)

SET @InnerTemp = SUBSTRING(@temp, 1, (@InnerCount - 1))
SET @CatName = @InnerTemp

SET @temp = SUBSTRING(@temp, (@InnerCount + 1), (LEN(@temp) - @InnerCount))
SET @InnerCount = CHARINDEX(',', @temp)

SET @InnerTemp = SUBSTRING(@temp, 1, (@InnerCount - 1))
SET @Abbrev = @InnerTemp

SET @temp = SUBSTRING(@temp, (@InnerCount + 1), (LEN(@temp) - @InnerCount))
SET @InnerCount = CHARINDEX(',', @temp)

SET @duration = CAST(@temp AS int)
/* 如果所要更新的项目工作分类并不存在, 则添加该项目工作分类 */
IF not exists(SELECT CategoryID from TT_Categories WHERE ProjectID = @projectId AND CategoryID
= @catID)
BEGIN
    INSERT INTO TT_Categories
    (
        ProjectID,
        [Name],
        Abbreviation,
        EstDuration
    )
    VALUES
    (

```

```
@ProjectID,
@CatName,
@Abbrev,
@Duration
)

SET @Error = @@ERROR
IF @Error != 0 GOTO ERROR_HANDLER

SELECT @CatID = @@IDENTITY

END
/* 如果所要更新的项目工作分类存在，则更新该项目工作分类 */
ELSE
BEGIN
    UPDATE
        TT_Categories
    SET
        [Name] = @CatName,
        Abbreviation = @Abbrev,
        EstDuration = @Duration
    WHERE
        ProjectID = @ProjectID
        AND
        CategoryID = @catID

    SET @Error = @@ERROR
    IF @Error != 0 GOTO ERROR_HANDLER
END

INSERT INTO @TempCatTable VALUES(@CatID)

DELETE
    TT_Categories
WHERE
    ProjectID = @ProjectID
    AND
    CategoryID NOT IN(SELECT CategoryID FROM @TempCatTable)

SET @Error = @@ERROR
IF @Error != 0 GOTO ERROR_HANDLER
/* 结束数据库事务 */
COMMIT TRANSACTION
RETURN 0
/* 定义错误处理 */
ERROR_HANDLER:
    IF @@TRANCOUNT != 0 ROLLBACK TRANSACTION
    RETURN @Error
```

(4) ListAllProjects

存储过程 ListAllProjects 返回数据中所有项目的列表。

```
CREATE PROCEDURE TT_ListAllProjects
AS
SELECT ProjectID,
       Name as ProjectName,
       Description,
       ManagerUserID,
       EstCompletionDate,
       EstDuration
  FROM TT_Projects
```

(5) ListProjects

存储过程 ListProjects 根据参数中的用户角色返回项目列表，如果用户角色为管理员，则返回数据库中的所有项目列表；如果用户角色为项目经理，则返回该项目经理所负责的项目列表。

```
CREATE PROCEDURE TT_ListProjects
(
    @UserID int,
    @RoleID int
)
AS
/* 如果用户角色为管理员，则返回所有项目 */
IF @RoleID = 1
BEGIN
    SELECT ProjectID,
           Name as ProjectName,
           Description,
           ManagerUserID,
           UserName,
           EstCompletionDate,
           EstDuration
      FROM
        TT_Projects
     INNER JOIN
        TT_Users
    ON
        ManagerUserID = UserID
END
/* 如果用户角色为项目经理，则返回该项目经理所负责的项目 */
ELSE IF @RoleID = 2
BEGIN
    SELECT ProjectID,
           Name as ProjectName,
           Description,
           ManagerUserID,
           UserName,
           EstCompletionDate,
           EstDuration
      FROM
        TT_Projects
```

```
INNER JOIN
    TT_Users
ON
    ManagerUserID = UserID
WHERE ManagerUserID = @UserID
END
```

(6) ListProjectsWithMembership

存储过程 ListProjectsWithMembership 根据参数 QueryUserID 和 UserID 获得某个用户有权限浏览的项目列表。如果查询的用户的角色为项目开发人员，则只返回他们本身的项目；如果查询用户的角色为项目经理，则返回该项目经理所负责的所有项目中项目成员的项目；如果查询用户的角色为管理员，则返回该用户 UserID 的所有项目列表。

```
CREATE PROCEDURE TT_ListProjectsWithMembership
(
    @QueryUserID int,
    @UserID int
)
AS
/* 获得查询用户 QueryUserID 的用户角色 */
DECLARE @@QueryUserRoleID int
SELECT @@QueryUserRoleID = TT_Users.RoleID FROM TT_Users WHERE TT_Users.UserID = @QueryUserID

IF @@QueryUserRoleID = 2
BEGIN
    SELECT      TT_Projects.ProjectID,
                Name,
                Description,
                ManagerUserID,
                EstCompletionDate,
                EstDuration
        FROM TT_Projects
    INNER JOIN TT_ProjectMembers ON TT_ProjectMembers.ProjectID = TT_Projects.ProjectID
    WHERE UserID = @UserID AND ManagerUserID = @QueryUserID
END
IF @@QueryUserRoleID = 1 OR @QueryUserID = @UserID
BEGIN
    SELECT      TT_Projects.ProjectID,
                Name,
                Description,
                ManagerUserID,
                EstCompletionDate,
                EstDuration
        FROM TT_Projects
    INNER JOIN TT_ProjectMembers ON TT_ProjectMembers.ProjectID = TT_Projects.ProjectID
    WHERE UserID = @UserID
END
```

(7) ListCategories

存储过程 ListCategories 为返回某个项目的所有工作类别的存储过程。代码清单如下。

```
CREATE PROCEDURE TT_ListCategories
()
```

```

    @ProjectID int
)
AS

SELECT
    CategoryID, Name, Abbreviation, EstDuration

FROM
    TT_Categories

WHERE
    ProjectID = @ProjectID

```

(8) DeleteProject

存储过程 DeleteProject 为删除某个项目的存储过程。代码清单如下。

```

CREATE PROCEDURE TT_DeleteProject
(
    @ProjectID int
)
AS

DELETE FROM
    TT_Projects

WHERE
    ProjectID = @ProjectID

```

15.3.2 逻辑层

项目管理部分的逻辑层实际上是对系统中的两个对象的管理，项目对象 Project 和项目工作分类对象 Category，类 Project 和 Category 实现了两个对象及其基本的方法。类 ProjectsCollection 和 CategoryCollection 为两个对象的集合类，这两个集合类的主要目的是存储和传递类 Project 和 Category 实例化之后的对象，并根据对象的部分属性对对象集合中的对象进行排序。

下面将分别对这 4 个类进行讲解。

(1) Project.cs

Project 类用于描述系统中的项目对象，Project 类中定义了项目对象的属性和方法。

```

using System;
using System.Data;
using System.Configuration;
using System.Text;
using ASPNET.StarterKit.TimeTracker.DataAccessLayer;

namespace ASPNET.StarterKit.TimeTracker.BusinessLogicLayer
{
    public class Project
    {

```

Project 类首先定义了项目对象的所有属性所对应的私有成员变量，这些私有成员变量的目的是存储和传递项目对象的属性，包括项目工作分类集合、项目描述、项目完成时间、工作总量、项目经理 ID、项目经理名称、项目成员集合、项目名称、项目 ID 等。

```
private CategoriesCollection _categories;
private string _description;
private DateTime _estCompletionDate;
private decimal _estDuration;
private int _managerUserID;
private string _managerUserName;
private UsersCollection _members;
private string _name;
private int _projectId;
```

Project类具有3个构造方法的重载。

```
//构造方法
public Project()
{
}
//构造方法重载，指定项目对象的项目ID
public Project(int projectId)
{
    _projectId = projectId;
}
//构造方法重载，指定项目对象的部分属性
public Project(
    int projectId,
    string name,
    string description,
    int managerUserID,
    DateTime estCompletionDate,
    decimal estDuration)
{
    _projectId = projectId;
    _name = name;
    _description = description;
    _managerUserID = managerUserID;
    _estCompletionDate = estCompletionDate;
    _estDuration = estDuration;
}
```

Project类中项目对象的属性都是通过构造器的结构来定义的，以下是定义对象属性的构造器。

```
//定义项目工作分类集合
public CategoriesCollection Categories
{
    get{ return _categories; }
    set{ _categories = value; }
}
//定义项目描述属性
public string Description
{
    get{ return _description; }
    set{ _description = value; }
}
//定义项目结束时间属性
public DateTime EstCompletionDate
```

```
{  
    get{ return _estCompletionDate; }  
    set{ _estCompletionDate = value; }  
}  
//定义项目结束时间属性  
public decimal EstDuration  
{  
    get{ return _estDuration; }  
    set{ _estDuration = value; }  
}  
//定义项目经理 ID 属性  
public int ManagerUserID  
{  
    get{ return _managerUserID; }  
    set{ _managerUserID = value; }  
}  
//定义项目经理名属性  
public string ManagerUserName  
{  
    get{ return _managerUserName; }  
    set{ _managerUserName = value; }  
}  
//定义项目成员集合属性  
public UsersCollection Members  
{  
    get{ return _members; }  
    set{ _members = value; }  
}  
//定义项目名称属性  
public string Name  
{  
    get{ return _name; }  
    set{ _name = value; }  
}  
//定义项目 ID 属性  
public int ProjectID  
{  
    get{ return _projectID; }  
    set{ _projectID = value; }  
}
```

GetProjects 方法提供了几种不同版本的重载，根据不同的条件获得 ProjectsCollection 类型的项目对象集合。

```
//获得所有项目对象集合  
public static ProjectsCollection GetProjects()  
{  
    //执行存储过程 TT_ListAllProjects 获得数据集  
    DataSet ds = SqlHelper.ExecuteDataset(  
        ConfigurationSettings.AppSettings[Web.Global.CfgKeyConnString],  
        CommandType.StoredProcedure, "TT_ListAllProjects");
```

```
//建立对象集合
ProjectsCollection projects = new ProjectsCollection();
//从数据集中获得项目对象的数据，建立对象，并存储到对象集合中
foreach(DataRow r in ds.Tables[0].Rows)
{
    Project prj = new Project();
    prj.ProjectID = Convert.ToInt32(r["ProjectID"]);
    prj.Name = r["ProjectName"].ToString();
    prj.Description = r["Description"].ToString();
    prj.ManagerUserID = Convert.ToInt32(r["ManagerUserID"]);
    prj.EstCompletionDate = Convert.ToDateTime(r["EstCompletionDate"]);
    prj.EstDuration = Convert.ToDecimal(r["EstDuration"]);
    projects.Add(prj);
}
//返回对象集合
return projects;
}

//根据用户的 ID 和权限返回项目对象集合
public static ProjectsCollection GetProjects(int userID, string role)
{
    string firstName = string.Empty;
    string lastName = string.Empty;
    //执行存储过程 TT_ListProjects 获得数据集
    DataSet ds = SqlHelper.ExecuteDataset(
        ConfigurationSettings.AppSettings[Web.Global.CfgKeyConnString],
        "TT_ListProjects", userID, Convert.ToInt32(role));
    //建立对象集合
    ProjectsCollection projects = new ProjectsCollection();
    //从数据集中获得项目对象的数据，建立对象，并存储到对象集合中
    foreach(DataRow r in ds.Tables[0].Rows)
    {
        Project prj = new Project();
        prj.ProjectID = Convert.ToInt32(r["ProjectID"]);
        prj.Name = r["ProjectName"].ToString();
        prj.Description = r["Description"].ToString();
        prj.ManagerUserID = Convert.ToInt32(r["ManagerUserID"]);
        prj.ManagerUserName =
            TTUser.GetDisplayName(Convert.ToString(r["UserName"]), ref firstName, ref lastName);
        prj.EstCompletionDate = Convert.ToDateTime(r["EstCompletionDate"]);
        prj.EstDuration = Convert.ToDecimal(r["EstDuration"]);
        projects.Add(prj);
    }
    //返回对象集合
    return projects;
}

//根据查询用户的权限返回某个用户参与的项目集合
public static ProjectsCollection GetProjects(int queryUserID, int userID)
{
    //建立项目对象集合
    ProjectsCollection projects = new ProjectsCollection();
```

```

//执行存储过程 TT_ListProjectsWithMembership 获得数据集
DataSet ds = SqlHelper.ExecuteDataset(
    ConfigurationSettings.AppSettings[Web.Global.CfgKeyConnString],
    "TT_ListProjectsWithMembership", queryUserID, userID);
//从数据集中获得项目对象的数据，建立对象，并存储到对象集合中
foreach(DataRow r in ds.Tables[0].Rows)
{
    Project prj = new Project();
    prj.ProjectID = Convert.ToInt32(r["ProjectID"]);
    prj.Name = r["Name"].ToString();
    prj.Description = r["Description"].ToString();
    prj.ManagerUserID = Convert.ToInt32(r["ManagerUserID"]);
    prj.EstCompletionDate = Convert.ToDateTime(r["EstCompletionDate"]);
    prj.EstDuration = Convert.ToDecimal(r["EstDuration"]);
    projects.Add(prj);
}
return projects;
}

```

GetCategories 方法为获得某个项目的所有工作分类的方法，该方法返回 CategoriesCollection 类型的项目工作分类集合。

```

public static CategoriesCollection GetCategories(int projectID)
{
    //执行存储过程 TT_ListCategories 获得数据集
    DataSet ds = SqlHelper.ExecuteDataset(
        ConfigurationSettings.AppSettings[Web.Global.CfgKeyConnString],
        "TT_ListCategories", projectID);
    //建立项目分类对象集合
    CategoriesCollection categories = new CategoriesCollection();
    //从数据集中获得项目对象的数据，建立对象，并存储到对象集合中
    foreach(DataRow r in ds.Tables[0].Rows)
    {
        Category cat = new Category();
        cat.CategoryID = Convert.ToInt32(r["CategoryID"]);
        cat.ProjectID = projectID;
        cat.Name = r["Name"].ToString();
        cat.Abbreviation = r["Abbreviation"].ToString();
        cat.EstDuration = Convert.ToDecimal(r["EstDuration"]);
        categories.Add(cat);
    }
    return categories;
}

```

Remove 方法为删除某个项目的方法。

```

public static void Remove (int projectID)
{
    //执行存储过程 TT_DeleteProject 删除指定项目
    SqlHelper.ExecuteNonQuery(ConfigurationSettings.AppSettings[Web.Global.CfgKeyConnString],
    "TT_DeleteProject", projectID);
}

```

Insert 方法为向数据库中添加项目信息的方法，该方法返回值为布尔值，标识添加项目信息是否成

功。向数据库中添加的项目信息包括项目成员信息、项目工作分类信息和项目的基本信息。

```
private bool Insert()
{
    //构建项目成员信息字符串，该字符串的格式是所有项目成员ID中间以，隔开
    StringBuilder selectedMembers = new StringBuilder(_members.Count);
    int index = 1;
    foreach(TTUser user in _members)
    {
        selectedMembers.Append(user.UserID);

        //如果是最后的项目则不添加分隔符
        if (index != _members.Count)
        {
            selectedMembers.Append(",");
        }
        index++;
    }
    //构建项目工作分类字符串
    StringBuilder categories = new StringBuilder(_categories.Count);
    index = 1;
    foreach(Category cat in _categories)
    {
        string categoryString =
            string.Format("{0},{1},{2}", cat.Name, cat.Abbreviation, cat.EstDuration);
        categories.Append(categoryString);

        //如果是最后的项目则不添加分隔符
        if (index != _categories.Count)
        {
            categories.Append(";");
        }
        index++;
    }
    //调用存储过程 TT_AddProject 添加项目信息
    _projectId = Convert.ToInt32(SqlHelper.ExecuteScalar(
        ConfigurationSettings.AppSettings[Web.Global.CfgKeyConnString], "TT_AddProject",
        _name,
        _description,
        _managerUserID,
        _estCompletionDate,
        _estDuration,
        selectedMembers.ToString(),
        categories.ToString()
    ));
    //如果添加成功则返回 true
    return (_projectId > 0);
}
```

Load方法为载入项目属性数据的方法，Load方法调用数据库存储过程 TT_GetProject，返回的DataSet中包含了3个DataTable：第一个DataTable为项目主体信息，第二个DataTable包含项目成员信息数据，第三个DataTable为项目工作分类数据。

```
public bool Load()
{
    //调用存储过程 TT_GetProject 获得项目信息
    DataSet ds = SqlHelper.ExecuteDataset(
        ConfigurationSettings.AppSettings[Web.Global.CfgKeyConnString],
        "TT_GetProject", _projectID);
    //如果没有要查询的数据，则返回 false
    if (ds.Tables[0].Rows.Count < 1)
        return false;
    //获得项目主体信息
    _name = Convert.ToString(ds.Tables[0].Rows[0]["Name"]);
    _description = Convert.ToString(ds.Tables[0].Rows[0]["Description"]);
    _managerUserID = Convert.ToInt32(ds.Tables[0].Rows[0]["ManagerUserID"]);
    _estCompletionDate = Convert.ToDateTime(ds.Tables[0].Rows[0]["EstCompletionDate"]);
    _estDuration = Convert.ToDecimal(ds.Tables[0].Rows[0]["estDuration"]);
    //获得项目成员信息
    _members = new UsersCollection();
    foreach (DataRow row in ds.Tables[1].Rows)
    {
        TTUser user = new TTUser();
        user.UserID = Convert.ToInt32(row["UserID"]);
        user.UserName = Convert.ToString(row["UserName"]);
        _members.Add(user);
    }
    //获得项目工作分类信息
    _categories = new CategoriesCollection();
    foreach (DataRow row in ds.Tables[2].Rows)
    {
        Category cat = new Category();
        cat.ProjectID = _projectID;
        cat.CategoryID = Convert.ToInt32(row["CategoryID"]);
        cat.Name = Convert.ToString(row["Name"]);
        cat.Abbreviation = Convert.ToString(row["CategoryShortName"]);
        cat.EstDuration = Convert.ToDecimal(row["EstDuration"]);
        _categories.Add(cat);
    }
    //成功载入项目信息返回 true
    return true;
}
```

Save 方法为保存项目信息的方法，该方法根据对象的_projectID 判断对数据库进行添加操作还是更新操作。

```
public bool Save()
{
    if (_projectID == 0)
        return Insert();
    else if (_projectID > 0)
        return Update();
    else
    {
        _projectID = 0;
    }
}
```

```
        return false;
    }
}
```

Update 方法为更新项目信息的方法，更新的项目信息包括项目主体信息、项目成员信息和项目工作分类信息。

```
private bool Update()
{
    //构建项目成员信息字符串
    StringBuilder selectedMembers = new StringBuilder(_members.Count);
    int index = 1;
    foreach(TTUser user in _members)
    {
        selectedMembers.Append(user.UserID);
        if (index != _members.Count)
        {
            selectedMembers.Append(",");
        }
        index++;
    }
    //构建项目工作分类信息字符串
    StringBuilder categories = new StringBuilder(_categories.Count);
    index = 1;
    foreach(Category cat in _categories)
    {
        string categoryString =
            string.Format("{0}, {1}, {2}, {3}", cat.CategoryID, cat.Name,
            cat.Abbreviation, cat.EstDuration);
        categories.Append(categoryString);
        if (index != _categories.Count)
        {
            categories.Append(";");
        }
        index++;
    }
    //调用存储过程 TT_UpdateProject 更新项目信息
    try
    {
        SqlHelper.ExecuteNonQuery(
            ConfigurationSettings.AppSettings[Web.Global.CfgKeyConnString], "TT_UpdateProject",
            _projectId,
            _name,
            _description,
            _managerUserID,
            _estCompletionDate,
            _estDuration,
            selectedMembers.ToString(),
            categories.ToString()
        );
    }
    catch
```

```
        {
            return false;
        }
        return true;
    }
}
```

(2) ProjectsCollection.cs

ProjectsCollection 类是 Project 对象的集合类，该类的代码清单如下。

```
using System;
using System.Collections;

namespace ASPNET.StarterKit.TimeTracker.BusinessLogicLayer
{
```

ProjectsCollection 类继承自 ArrayList 类，并重载了 ArrayList 类的 Sort 方法。

```
public class ProjectsCollection : ArrayList
{
    //定义项目对象的排序字段集合
    public enum ProjectFields
    {
        InitValue,
        Name,
        ManagerUserName,
        CompletionDate,
        Duration
    }

    //Sort 方法为根据某个字段对项目对象集合排序
    public void Sort(ProjectFields sortField, bool isAscending)
    {
        switch (sortField)
        {
            case ProjectFields.Name:
                base.Sort(new NameComparer());
                break;
            case ProjectFields.ManagerUserName:
                base.Sort(new ManagerUserNameComparer());
                break;
            case ProjectFields.CompletionDate:
                base.Sort(new CompletionDateComparer());
                break;
            case ProjectFields.Duration:
                base.Sort(new DurationComparer());
                break;
        }

        if (!isAscending) base.Reverse();
    }
}

//类 NameComparer 继承自 IComparer 接口实现对项目对象的 Name 属性的比较
private sealed class NameComparer : IComparer
{
```

```
public int Compare(object x, object y)
{
    Project first = (Project) x;
    Project second = (Project) y;
    return first.Name.CompareTo(second.Name);
}

//类ManagerUserNameComparer继承自 Icomparer 接口实现对项目对象的 ManagerUserName 属性的比较
private sealed class ManagerUserNameComparer : IComparer
{
    public int Compare(object x, object y)
    {
        Project first = (Project) x;
        Project second = (Project) y;
        return first.ManagerUserName.CompareTo(second.ManagerUserName);
    }
}

//类CompletionDateComparer继承自 Icomparer 接口实现对项目对象的 CompletionDate 属性的比较
private sealed class CompletionDateComparer : IComparer
{
    public int Compare(object x, object y)
    {
        Project first = (Project) x;
        Project second = (Project) y;
        return first.EstCompletionDate.CompareTo(second.EstCompletionDate);
    }
}

//类DurationComparer继承自 Icomparer 接口实现对项目对象的 Duration 属性的比较
private sealed class DurationComparer : IComparer
{
    public int Compare(object x, object y)
    {
        Project first = (Project) x;
        Project second = (Project) y;
        return first.EstDuration.CompareTo(second.EstDuration);
    }
}
```

(3) Category.cs

Category 类构建了项目工作分类对象，实际上该类并没有具体的功能和方法，仅仅依靠其内部的属性构造器发挥一个数据载体的作用。Category 类的代码清单如下。

```
using System;

namespace ASPNET.StarterKit.TimeTracker.BusinessLogicLayer
{
    public class Category
    {
        //定义私有成员变量
        private string _abbreviation;
```

```
private int      _categoryID;
private decimal _estDuration;
private string   _name;
private int      _projectID;
//定义构造器实现对象的属性
public string Abbreviation
{
    get{ return _abbreviation; }
    set{ _abbreviation = value; }
}

public int CategoryID
{
    get{ return _categoryID; }
    set{ _categoryID = value; }
}

public decimal EstDuration
{
    get{ return _estDuration; }
    set{ _estDuration = value; }
}

public string Name
{
    get{ return _name; }
    set{ _name = value; }
}

public int ProjectID
{
    get{ return _projectId; }
    set{ _projectId = value; }
}
```

(4) CategoriesCollection.cs

CategoriesCollection 为 Categories 对象的集合类。该集合类提供了根据存储在集合类中的 Categories 对象的名称、简称和时间排序的方法。CategoriesCollection 类的代码清单如下。

```
using System;
using System.Collections;

namespace ASPNET.StarterKit.TimeTracker.BusinessLogicLayer
{
    public class CategoriesCollection : ArrayList
    {
        public enum CategoryFields
        {
            Abbreviation,
            Duration,
```

```
        InitValue,
        Name
    }

    public void Sort(CategoryFields sortField, bool isAscending)
    {
        switch (sortField)
        {
            case CategoryFields.Name:
                base.Sort(new NameComparer());
                break;
            case CategoryFields.Abbreviation:
                base.Sort(new AbbreviationComparer());
                break;
            case CategoryFields.Duration:
                base.Sort(new DurationComparer());
                break;
        }

        if (!isAscending) base.Reverse();
    }

    private sealed class NameComparer : IComparer
    {
        public int Compare(object x, object y)
        {
            Category first = (Category) x;
            Category second = (Category) y;
            return first.Name.CompareTo(second.Name);
        }
    }

    private sealed class AbbreviationComparer : IComparer
    {
        public int Compare(object x, object y)
        {
            Category first = (Category) x;
            Category second = (Category) y;
            return first.Abbreviation.CompareTo(second.Abbreviation);
        }
    }

    private sealed class DurationComparer : IComparer
    {
        public int Compare(object x, object y)
        {
            Category first = (Category) x;
            Category second = (Category) y;
            return first.EstDuration.CompareTo(second.EstDuration);
        }
    }
}
```

```
        }  
    }  
}
```

15.3.3 应用层

在项目管理页面中实现了对系统中的项目进行管理的功能，其中包括项目的基本信息管理、项目参与人员管理以及项目工作分类管理。实现页面为 ProjectDetails.aspx，页面内部包含了所有项目管理中涵盖的功能，其后台编码类 ProjectDetails.aspx.cs 的代码如下。

```
using System;  
using System.Web;  
using System.Web.UI.WebControls;  
//引入逻辑层的命名空间  
using ASPNET.StarterKit.TimeTracker.BusinessLogicLayer;  
  
namespace ASPNET.StarterKit.TimeTracker.Web  
{  
  
    public class ProjectDetails : System.Web.UI.Page  
    {  
        //定义页面控件  
        protected System.Web.UI.WebControls.TextBox ProjectName;  
        protected System.Web.UI.WebControls.TextBox Description;  
        protected System.Web.UI.WebControls.TextBox CompletionDate;  
        protected System.Web.UI.WebControls.TextBox Duration;  
        protected System.Web.UI.WebControls.ListBox Members;  
        protected System.Web.UI.WebControls.Button CopyButton;  
        protected System.Web.UI.WebControls.Button AddButton;  
        protected System.Web.UI.WebControls.Button SaveButton;  
        protected System.Web.UI.WebControls.DropDownList Managers;  
        protected System.Web.UI.WebControls.DropDownList Projects;  
        protected System.Web.UI.WebControls.TextBox CategoryName;  
        protected System.Web.UI.WebControls.TextBox Abbrev;  
        protected System.Web.UI.WebControls.TextBox CatDuration;  
        protected System.Web.UI.WebControls.DataGrid CategoriesGrid;  
        protected System.Web.UI.WebControls.RequiredFieldValidator ProjectNameRequiredfieldvalidator;  
        protected System.Web.UI.WebControls.RequiredFieldValidator ManagerRequiredFieldValidator;  
        protected System.Web.UI.WebControls.CompareValidator CompletionDateCompareValidator;  
        protected System.Web.UI.WebControls.RequiredFieldValidator CompletionDateRequiredFieldValidator;  
        protected System.Web.UI.WebControls.CompareValidator DurationCompareValidator;  
        protected System.Web.UI.WebControls.RequiredFieldValidator DurationRequiredFieldValidator;  
        protected System.Web.UI.WebControls.CustomValidator ProjectsGridCustomValidator;  
        protected System.Web.UI.WebControls.RegularExpressionValidator RegularExpressionValidatorAbbrev;  
        protected System.Web.UI.WebControls.CompareValidator CatDurationValidator;  
        protected System.Web.UI.WebControls.CustomValidator AbbrevCustomValidator;  
        protected System.Web.UI.WebControls.Label ErrorMessage;  
        protected System.Web.UI.WebControls.Button CancelButton;  
        protected System.Web.UI.WebControls.Button CancelButton2;  
        protected System.Web.UI.WebControls.Button CancelButton3;
```

```
protected System.Web.UI.WebControls.Button DeleteButton;
protected System.Web.UI.WebControls.Button DeleteButton2;
protected System.Web.UI.WebControls.Button SaveButton2;
protected System.Web.UI.WebControls.RangeValidator RangeValidator1;
protected System.Web.UI.WebControls.RangeValidator RangeValidator2;
protected System.Web.UI.WebControls.Label CategoryErrorMessage;
protected System.Web.UI.WebControls.RegularExpressionValidator CategoryNameValidator;
protected System.Web.UI.WebControls.RegularExpressionValidator RegularExpressionValidator1;

//定义项目ID为私有成员变量
private int _projID;

private void Page_Load(object sender, System.EventArgs e)
{
```

在用户管理章节讨论过 IsInRole 方法，该方法判断用户是否属于某个角色。在项目管理页面首先用到了这个方法，判断当前用户是否具有管理员或是项目经理的角色。如果不具有管理员或是项目经理的角色，则定向到权限信息页面。

```
//判断用户是否有权限浏览该页面，如果没有权限则重定向到错误信息页面
//只有当用户的角色是管理员或项目经理的情况下才能浏览该页面
if (TTSecurity.IsInRole(TTUser.UserRoleAdminPMgr) == false)
{
    Response.Redirect("AccessDenied.aspx?Index=-1", true);
}

//获得Url中的项目ID，如果Url中并没有包含项目ID则设置为0
_projID = (Request.QueryString["id"] == null) ? 0 : Convert.ToInt32(Request.QueryString["id"]);
```

接下来判断页面是否被提交，如果页面未被提交，则调用相关方法绑定页面数据。从名为 catArray 的 Session 中取得当前页面中项目工作分类的信息，因此在用户没有保存项目之前，所有的项目工作分类是暂时保存到 Session 中的。

```
if (!IsPostBack)
{
    //获取Session中的工作分类数据并赋给项目工作分类的对象集合
    Session["catArray"] = new CategoriesCollection();

    BindManagers();
    BindMembers();
    BindOtherProjects();

    //判断_projID是否存在，如果存在，则绑定当前项目
    if (_projID != 0)
        BindProject();
}

//为删除按钮添加 onclick 事件
DeleteButton.Attributes.Add("onclick", "return confirm('Deleting a project will also delete all the time entries and categories associated with the project. This deletion cannot be undone. Are you sure you want to delete this project?')");
DeleteButton2.Attributes.Add("onclick", "return confirm('Deleting a project will also delete all the time entries and categories associated with the project. This deletion cannot be undone. Are you sure you want to delete this project?')");
```

```

        to delete this project?'");
    }

    override protected void OnInit(EventArgs e)
    {
        InitializeComponent();
        base.OnInit(e);
    }

    private void InitializeComponent()
    {
        this.SaveButton.Click += new System.EventHandler(this.SaveButton_Click);
        this.CancelButton.Click += new System.EventHandler(this.CancelButton_Click);
        this.DeleteButton.Click += new System.EventHandler(this.DeleteButton_Click);
        this.AddButton.Click += new System.EventHandler(this.AddButton_Click);
        this.AbbrevCustomValidator.ServerValidate += new System.Web.UI.WebControls.ServerValidateEventHandler(this.AbbrevCustomValidator_ServerValidate);
        this.CopyButton.Click += new System.EventHandler(this.CopyButton_Click);
        this.CategoriesGrid.ItemCreated += new System.Web.UI.WebControls.DataGridItemEventHandler(this.CategoriesGrid_ItemCreated);
        this.CategoriesGrid.SortCommand += new System.Web.UI.WebControls.DataGridSortCommandEventHandler(this.CategoriesGrid_Sort);
        this.ProjectsGridCustomValidator.ServerValidate += new System.Web.UI.WebControls.ServerValidateEventHandler(this.ValidateCategories);
        this.SaveButton2.Click += new System.EventHandler(this.SaveButton_Click);
        this.CancelButton2.Click += new System.EventHandler(this.CancelButton_Click);
        this.DeleteButton2.Click += new System.EventHandler(this.DeleteButton_Click);
        this.Load += new System.EventHandler(this.Page_Load);

    }

```

BindManagers 方法用于绑定页面 DropDownList 控件 Managers 的数据，供用户从中选取当前项目的项目经理。

```

private void BindManagers()
{
    Managers.DataSource = TTUser.ListManagers();
    Managers.DataValueField = "UserID";
    Managers.DataTextField = "Name";
    Managers.DataBind();

    //添加一个空白字段供选
    Managers.Items.Insert(0, new ListItem("Select Manager...", String.Empty));
}

```

BindMembers 方法用于绑定页面 ListBox 控件 Members 的数据，供用户从中选择当前项目的项目成员。

```

private void BindMembers()
{
    Members.DataSource = TTUser.GetAllUsers(TTSecurity.GetUserID());
    Members.DataValueField = "UserID";
    Members.DataTextField = "Name";
}

```

```
        Members.DataBind();
    }
```

BindOtherProjects 方法用于绑定页面 DropDownList 控件 Projects 的数据，用户利用该数据可以从其他项目中复制项目分类到当前项目。

```
private void BindOtherProjects()
{
    Projects.DataSource = Project.GetProjects();
    Projects.DataTextField = "Name";
    Projects.DataValueField = "ProjectID";
    Projects.DataBind();

    //将当前项目从列表中移除
    Projects.Items.Remove(Projects.Items.FindByValue(_projID.ToString()));

    //如果选中项目的项目分类数为 0，则将负责项目分类的按钮设置为不可操作
    if (Projects.Items.Count == 0)
        CopyButton.Enabled = false;
}
```

BindProject 方法为绑定当前项目信息的方法。

```
private void BindProject()
{
    //根据_projID 实例化 project 对象
    Project project = new Project(_projID);
    project.Load();

    //绑定当前页面的内部控件
    ProjectName.Text = project.Name;
    Description.Text = project.Description;
    CompletionDate.Text = project.EstCompletionDate.ToShortDateString();
    Duration.Text = project.EstDuration.ToString();
    Managers.Items.FindByValue(project.ManagerUserID.ToString()).Selected = true;

    //读取当前项目的成员信息来设定 Members 控件的选中项
    foreach (TTUser user in project.Members)
    {
        Members.Items.FindByValue(user.UserID.ToString()).Selected = true;
    }
    //绑定项目分类的 DataGrid
    BindCategoriesGrid(project.Categories);
}
```

BindCategoriesGrid 方法是绑定当前项目的项目分类 DataGrid 的方法。值得注意的是，在绑定显示项目工作分类的 DataGrid 控件之后，BindCategoriesGrid 方法又将当前的项目工作分类的对象集合存储于 Session 中。之所以选择将项目分类对象集合存储于 Session 中而不是 Cookie，是因为 Session 可以存储对象类型的数据，而 Cookie 不可以。

```
private void BindCategoriesGrid(CategoriesCollection cats)
{
    //调用 SortGridData 方法对对象集合类中的对象进行排序
    SortGridData(cats, SortField, SortAscending);
    //指定显示项目工作分类的 DataGrid 的数据源并绑定
```

```

CategoriesGrid.DataSource = cats;
CategoriesGrid.DataBind();

//将当前项目的工作分类信息存储于 Session 中
Session["catArray"] = cats;
}

```

ReturnToProjectList 方法为返回项目列表的方法，该方法在销毁存储项目分类对象集合所使用的 Session 之后拼接项目列表页面的 URL。

```

private void ReturnToProjectList()
{
    //销毁存储项目分类对象集合所使用的 Session
    Session["catArray"] = null;

    //拼接项目列表页面的 URL，并重定向到该 URL
    int mainIndex = (Request["index"]==null) ? 0 : Convert.ToInt32(Request["index"]);
    int adminIndex = (Request["adminindex"]==null) ? 0 : Convert.ToInt32(Request["adminindex"]);
    Response.Redirect(String.Format("ProjectList.aspx?index={0}&adminindex={1}", mainIndex, adminIndex),
false);
}

```

SortGridData 为根据排序字段对项目工作分类对象集合进行排序的方法。

```

private void SortGridData(CategoriesCollection list, string sortField, bool asc)
{
    CategoriesCollection.CategoryFields sortCol = CategoriesCollection.CategoryFields.InitValue;
    //根据不同字段对对象集合进行排序
    switch(sortField)
    {
        case "Name":
            sortCol = CategoriesCollection.CategoryFields.Name;
            break;
        case "Abbrev":
            sortCol = CategoriesCollection.CategoryFields.Abbreviation;
            break;
        case "Duration":
            sortCol = CategoriesCollection.CategoryFields.Duration;
            break;
    }

    list.Sort(sortCol, asc);
}

```

CancelButton_Click 方法为取消按钮 CancelButton 的单击事件所触发的方法，调用 ReturnToProjectList 方法返回项目列表页。

```

private void CancelButton_Click(object sender, System.EventArgs e)
{
    ReturnToProjectList();
}

```

DeleteButton_Click 方法为删除按钮 DeleteButton 的单击事件所触发的方法，首先调用 Project 对象的 Remove 方法删除当前对象，然后调用 ReturnToProjectList 方法返回项目列表页。

```

private void DeleteButton_Click(object sender, System.EventArgs e)
{
}

```

```
        Project.Remove(_projID);
        ReturnToProjectList();
    }
```

SaveButton_Click 方法为保存按钮 SaveButton 的单击事件所触发的方法，该方法实现了对当前项目数据进行逐项保存，包含的逻辑比较复杂。

```
private void SaveButton_Click(object sender, System.EventArgs e)
{
    //首先将项目经理也作为一个普通的项目成员加入到项目成员列表中
    Members.Items.FindByValue(Managers.SelectedItem.Value).Selected = true;

    //判断当前项目至少具有一个项目分类
    ProjectsGridCustomValidator.Validate();
    if (!ProjectsGridCustomValidator.IsValid)
        return;

    //判断当前项目的项目分类中是否重复
    AbbrevCustomValidator.Validate();
    if (!AbbrevCustomValidator.IsValid)
        return;

    //从 Session 中获得当前项目分类的对象集合
    CategoriesCollection catArray = (CategoriesCollection)Session["catArray"];

    //获得当前项目的所有被选定的项目成员并存储在一个用户对象集合中
    UsersCollection selectedMembers = new UsersCollection();
    foreach (ListItem li in Members.Items)
    {
        if (li.Selected)
        {
            TTUser user = new TTUser();
            user.UserID = Convert.ToInt32(li.Value);
            selectedMembers.Add(user);
        }
    }
}
```

根据当前页面中的数据实例化一个 Project 对象 prj，调用该对象的 Save 方法，如果该方法返回的布尔值为 True，则表示当前项目的数据被成功保存；如果返回值为 false，则说明当前项目中某个已经分配具体工作的项目成员被移除导致保存项目信息失败。

```
Project prj = new Project(
    _projID,
    TTSecurity.CleanStringRegex(ProjectName.Text),
    TTSecurity.CleanStringRegex(Description.Text),
    Convert.ToInt32(Managers.SelectedItem.Value),
    Convert.ToDateTime(CompletionDate.Text),
    Convert.ToDecimal(Duration.Text)
);
prj.Categories = catArray;
prj.Members = selectedMembers;

if (prj.Save())
```

```
{  
    ReturnToProjectList();  
}  
else  
{  
    ErrorMessage.Text = "There was an error. You cannot remove a member with time entries from  
this project.";  
}  
}  
}
```

AddButton_Click 方法是为当前项目添加一个项目工作分类的方法，注意该方法仅仅是将新的项目工作分类添加到 DataGrid 和 Session 中，而并没有真正地写入数据库。

```
private void AddButton_Click(object sender, System.EventArgs e)  
{  
    //首先判断所要添加的项目分类是否有重复  
    if (!AbbrevCustomValidator.IsValid)  
        return;  
  
    //判断工作分类名是否为空  
    if (CategoryName.Text == "")  
    {  
        CategoryErrorMessage.Text = "Category name is required.";  
        return;  
    }  
    //判断 TextBox 控件 Abbrev 是否为空  
    if (Abbrev.Text == "")  
    {  
        CategoryErrorMessage.Text = "Category abbreviation is required.";  
        return;  
    }  
    //从 Session 中获取当前项目的工作分类对象集合类  
    CategoriesCollection catArray = (CategoriesCollection)Session["catArray"];  
    if (catArray == null)  
        catArray = new CategoriesCollection();  
  
    //为每一个即将添加的工作分类条目赋一个惟一标识的 ID  
    //如果该 ID 不存在则为-1  
    int catID = (Session["catID"] != null) ? (Convert.ToInt32(Session["catID"]) - 1) : -1;  
    Session["catID"] = catID;  
    //将该工作分类条目添加到工作分类对象的集合类中  
    Category cat = new Category();  
    cat.CategoryID = catID;  
    cat.Name = TTSecurity.CleanStringRegex(CategoryName.Text);  
    cat.Abbreviation = TTSecurity.CleanStringRegex(Abbrev.Text);  
    cat.EstDuration = (CatDuration.Text.Length==0) ? 0 : Convert.ToDecimal(CatDuration.Text);  
    catArray.Add(cat);  
  
    //调用自定义的验证控件 ProjectsGridCustomValidator 进行验证  
    ProjectsGridCustomValidator.Validate();  
    //调用 BindCategoriesGrid 完成对工作分类的绑定  
    BindCategoriesGrid(catArray);  
}
```

```
    CategoryName.Text = string.Empty;
    Abbrev.Text = string.Empty;
    CatDuration.Text = string.Empty;
}
```

在本页面中用户在维护项目的工作分类时可以选择从另外个项目中复制项目分类，CopyButton_Click 方法为复制项目工作分类按钮单击触发事件。该方法的设计思路和 AddButton_Click 方法类似，只是 CopyButton_Click 方法是一次性地从数据库读取并添加多个工作项目分类。

```
private void CopyButton_Click(object sender, System.EventArgs e)
{
    int catID;
    //首先从 Session 中获得
    CategoriesCollection catArray = (CategoriesCollection)Session["catArray"];
    if (catArray == null)
        catArray = new CategoriesCollection();
    //获得被复制项目的工作分类
    CategoriesCollection projcatArray =
Project.GetCategories(Convert.ToInt32(Projects.SelectedItem.Value));
    foreach (Category cat in projcatArray)
    {
        //为每一条新添加的工作分类赋惟一的标识并存储在对象集合中
        catID = (Session["catID"] != null) ? (Convert.ToInt32(Session["catID"]) + 1) : 1;
        Session["catID"] = catID;
        cat.CategoryID = catID;
        catArray.Add(cat);
    }

    //验证数据的合法性
    ProjectsGridCustomValidator.Validate();
    //保存存储当前项目分类的 Session
    Session["catArray"] = catArray;
    //绑定当前项目分类的数据
    BindCategoriesGrid(catArray);
}
```

删除项目分类按钮的单击事件触发 CategoriesGrid_OnDelete 方法，该方法的作用是删除当前项目的某一个工作分类条目，实际上仅仅是移除当前用户所操作的 Session 中保存的项目工作分类。

```
protected void CategoriesGrid_OnDelete(Object sender, DataGridCommandEventArgs e)
{
    //首先获得要删除的项目工作分类 ID
    int catID = Convert.ToInt32(CategoriesGrid.DataKeys[(int)e.Item.ItemIndex]);
    //然后从 Session 中获得工作分类对象集合
    CategoriesCollection catArray = (CategoriesCollection)Session["catArray"];
    //删除工作分类
    for (int i = 0; i < catArray.Count; i++)
    {
        if (((Category)catArray[i]).CategoryID == catID)
        {
            catArray.RemoveAt(i);
        }
    }
}
```

```
        }
        //重新绑定 DataGrid
        BindCategoriesGrid(catArray);
    }
```

CategoriesGrid_OnCancel 方法为编辑某一个工作分类时取消按钮的单击事件所触发的方法，该方法的作用是取消 DataGrid 的编辑项，并重新绑定 DataGrid。

```
protected void CategoriesGrid_OnCancel(Object sender, DataGridViewEventArgs e)
{
    CategoriesGrid.EditRowIndex = -1;

    BindCategoriesGrid((CategoriesCollection)Session["catArray"]);
}
```

CategoriesGrid_OnUpdate 方法为编辑某一个工作分类时更新按钮的单击事件所触发的方法，该方法的作用是更新当前编辑工作分类的信息。该方法首先获得当前编辑行中的数据，然后新建一个存储工作分类的对象集合类，将编辑行中的数据加入对象集合类中之后又将原来存储于 Session 中的其他工作分类数据加入到新的对象集合类中。

```
protected void CategoriesGrid_OnUpdate(Object sender, DataGridViewEventArgs e)
{
    //根据当前编辑行中的数据生成新的工作分类对象
    Category editCat = new Category();
    editCat.CategoryID = Convert.ToInt32(CategoriesGrid.DataKeys[(int)e.Item.Index]);
    editCat.Name = TTSecurity.CleanStringRegex(((TextBox)e.Item.FindControl("EditName")).Text);
    editCat.Abbreviation = TTSecurity.CleanStringRegex(((TextBox)e.Item.FindControl("EditAbbreviation")).Text);
    editCat.EstDuration = Convert.ToDecimal(((TextBox)e.Item.FindControl("EditDuration")).Text);

    //遍历存储于对象集合类中的工作分类排除工作分类名重复的情况
    CategoriesCollection catArray = (CategoriesCollection)Session["catArray"];
    if (catArray != null)
    {
        string editAbbrev = ((TextBox)e.Item.FindControl("EditAbbreviation")).Text;
        foreach (Category cat in catArray)
        {
            //如果工作分类名重复则给出提示并取消当前更新数据
            if (cat.Abbreviation == editCat.Abbreviation && cat.CategoryID != editCat.CategoryID)
            {
                CategoryErrorMessage.Text = "Duplicate categories abbreviations are not allowed";
                return;
            }
        }
    }

    //新建一个对象集合类
    CategoriesCollection catNew = new CategoriesCollection();

    //将当前编辑工作分类加入对象集合类中
    catNew.Add(editCat);
    //从原来的对象集合类中加入未做修改的工作分类对象
    if (catArray == null)
```

```

catArray = new CategoriesCollection();
foreach (Category cat in catArray)
{
    if (cat.CategoryID != editCat.CategoryID)
    {
        catNew.Add(cat);
    }
}
//取消 DataGridView 的编辑行
CategoriesGrid.EditRowIndex = -1;
//重新绑定 DataGridView
BindCategoriesGrid(catNew);
}

```

CategoriesGrid_OnEdit 方法为显示工作分类数据的 DataGridView 控件在正常模式下单击编辑按钮编辑某项工作分类事件所触发的方法，该方法设置了 DataGridView 的编辑行。

```

protected void CategoriesGrid_OnEdit(object sender, DataGridViewCommandEventArgs e)
{
    CategoriesGrid.EditRowIndex = e.Item.RowIndex;

    BindCategoriesGrid((CategoriesCollection)Session["catArray"]);
}

```

CategoriesGrid_Sort 方法为按照某列对显示项目工作条目的 DataGridView 进行排序时所触发的方法。

```

private void CategoriesGrid_Sort(object source, System.Web.UI.WebControls.DataGridSortCommandEventArgs e)
{
    SortField = e.SortExpression;

    BindCategoriesGrid((CategoriesCollection)Session["catArray"]);
}

```

ValidateCategories 方法为验证存储工作分类对象集合的 Session 中是否有工作分类对象的方法，如果有则通过验证，如果没有则验证失败。

```

private void ValidateCategories(object source, System.Web.UI.WebControls.ServerValidateEventArgs args)
{
    CategoriesCollection catArray = (CategoriesCollection)Session["catArray"];
    if (catArray != null)
    {
        if (catArray.Count > 0)
        {
            args.IsValid = true;
            return;
        }
    }
    args.IsValid = false;
}

```

AbbrevCustomValidator_ServerValidate 方法为自定义的验证控件 AbbrevCustomValidator 的验证时间触发方法，该方法验证了存储工作分类对象集合 Session 中每一个工作分类集合对象的名称不重复。

```

private void AbbrevCustomValidator_ServerValidate(object source, System.Web.UI.WebControls.ServerValidateEventArgs args)
{
}

```

```

CategoriesCollection catArray = (CategoriesCollection)Session["catArray"];
if (catArray.Count != 0)
{
    catArray.Sort(CategoriesCollection.CategoryFields.Abbreviation, true);
    for(int i=0; i < catArray.Count; i++)
    {
        if (((Category)catArray[i]).Abbreviation == Abbrev.Text)
        {
            args.IsValid = false;
            return;
        }
        if (i!=0 && ((Category)catArray[i]).Abbreviation ==
((Category)catArray[i-1]).Abbreviation )
        {
            args.IsValid = false;
            return;
        }
    }
}
args.IsValid = true;
}

```

CategoriesGrid_ItemCreated 方法为显示工作分类对象的 DataGrid 每一行记录生成事件所触发的方法，该方法为每条数据的删除按钮添加一个客户端 onclick 事件。

```

private void CategoriesGrid_ItemCreated(object sender, System.Web.UI.WebControls.
DataGridItemEventArgs e)
{
    //判断是否为数据行
    if (e.Item.ItemType == ListItemType.Item || e.Item.ItemType == ListItemType.AlternatingItem)
    {
        ((ImageButton)e.Item.FindControl("CatDeleteButton")).Attributes.Add("onclick", "return
confirm ('Deleting an existing category will also delete all the time entries and categories associated with the
category. Are you sure you want to delete this category?')");
    }
}

```

SortField 构造器和 SortAscending 构造器的功能是读取和设置 DataGrid 排序字段和排序方式。SortField 和 SortAscending 的值存储于当前页面的 ViewState 中，值得注意的是，当设置排序字段的时候，如果原有排序字段和所要存储的排序字段相同，则更改当前的排序方式。

```

string SortField
{
    get
    {
        object o = ViewState["SortField"];
        if (o == null)
        {
            return String.Empty;
        }
    }
}

```

```

        return (string)o;
    }

    set
    {
        //如果原有排序字段和所要存储的排序字段相同，则更改当前的排序方式
        if (value == SortField)
        {
            SortAscending = !SortAscending;
        }
        ViewState["SortField"] = value;
    }
}

bool SortAscending
{
    get
    {
        object o = ViewState["SortAscending"];
        if (o == null)
        {
            return true;
        }
        return (bool)o;
    }

    set
    {
        ViewState["SortAscending"] = value;
    }
}
}

```

15.3.4 表示层

项目管理部分包括项目列表页面 ProjectList.aspx 和项目编辑页面 ProjectDetail.aspx，实际上项目编辑页面和项目添加页面是同一个页面。由于项目管理页面中所采用的 DataGrid 在逻辑上比较简单，因此本书不做过多介绍，本书将重点讲解项目编辑页面的前台页面所包含的业务逻辑。

项目编辑页面中大部分控件都是直接和项目的数据进行绑定的，因此这些控件在前台页面中并没有过多的业务逻辑，显示项目分类数据的 DataGrid 的代码如下。

```

<!-- 首先定义了 DataGrid 的几个事件所触发的方法 -->
<asp:datagrid id="CategoriesGrid" runat="server" Width="100%" font-names="Verdana" BorderColor="White"
AllowSorting="True" Font-Name="Verdana" AutoGenerateColumns="False" CellPadding="2" OnEditCommand=
"CategoriesGrid_OnEdit" OnDeleteCommand="CategoriesGrid_OnDelete" OnCancelCommand="CategoriesGrid_OnCancel"
OnUpdateCommand="CategoriesGrid_OnUpdate" DataKeyField="CategoryID" BorderStyle="None">
    <!-- 定义 DataGrid 的 HeaderStyle -->
    <HeaderStyle Font-Bold="True" CssClass="grid-header"></HeaderStyle>
    <!-- 定义 DataGrid 的行 -->

```

```
<Columns>
    <!-- 定义工作分类名称的模板列 -->
    <asp:TemplateColumn SortExpression="Name" HeaderText="Name">
        <HeaderStyle HorizontalAlign="Left" CssClass="grid-header" VerticalAlign="Middle">
</HeaderStyle>
        <ItemStyle CssClass="grid-first-item"></ItemStyle>
        <!-- 普通状态的模板 -->
        <ItemTemplate>
            <%# DataBinder.Eval(Container, "DataItem.Name") %>
        </ItemTemplate>
        <!-- 编辑状态的模板 -->
        <EditItemTemplate>
            <asp:TextBox id=EditName Text='<%# DataBinder.Eval(Container, "DataItem.Name") %>' CssClass="Standard-text" MaxLength="50" Runat="server" AutoPostBack="false"></asp:TextBox>
            <asp:RequiredFieldValidator id="RequiredFieldValidator1" runat="server" ErrorMessage="Category name is required" ControlToValidate="EditName" Display="Dynamic"></asp:RequiredFieldValidator>
        </EditItemTemplate>
        </asp:TemplateColumn>
        <!-- 定义工作分类简称的模板列 -->
        <asp:TemplateColumn SortExpression="Abbrev" HeaderText="Abbrev.">
            <HeaderStyle HorizontalAlign="Left" Width="70px" CssClass="grid-header" VerticalAlign="Middle">
</HeaderStyle>
            <ItemStyle CssClass="grid-item"></ItemStyle>
            <!-- 普通状态的模板-->
            <ItemTemplate>
                <%# DataBinder.Eval(Container, "DataItem.Abbreviation") %>
            </ItemTemplate>
            <!-- 编辑状态的模板-->
            <EditItemTemplate>
                <asp:TextBox id=EditAbbreviation Text='<%# DataBinder.Eval(Container, "DataItem.Abbreviation") %>' CssClass="Standard-text" width="70px" Runat="server" AutoPostBack="false"></asp:TextBox>
                <asp:RequiredFieldValidator id="RequiredFieldValidator5" runat="server" ErrorMessage="This is a required value." ControlToValidate="EditAbbreviation" Display="Dynamic"></asp:RequiredFieldValidator>
                <!-- 字符串长度验证控件 -->
                <asp:RegularExpressionValidator id="RegularExpressionValidator3" runat="server" ErrorMessage="Abbreviation must be between 1-4 characters long." ControlToValidate="EditAbbreviation" Display="Dynamic" ValidationExpression="\S{1,4}"></asp:RegularExpressionValidator>
            </EditItemTemplate>
            </asp:TemplateColumn>
        <!-- 定义工作分类时间的模板列 -->
        <asp:TemplateColumn SortExpression="Duration" HeaderText="Duration(hrs)">
            <HeaderStyle Wrap="False" HorizontalAlign="Right" Width="100px" CssClass="grid-header" VerticalAlign="Middle"></HeaderStyle>
            <ItemStyle HorizontalAlign="Right" CssClass="grid-item"></ItemStyle>
            <!-- 普通状态的模板-->
            <ItemTemplate>
                <%# DataBinder.Eval(Container, "DataItem.EstDuration") %>
            </ItemTemplate>
            <!-- 编辑状态的模板-->
        </asp:TemplateColumn>
    </Columns>
```

```
<EditItemTemplate>
    <asp:textbox id=EditDuration Text='<%# DataBinder.Eval(Container, "DataItem.EstDuration") %>' CssClass="Standard-text" width="70px" Runat="server" AutoPostBack="false"> </asp:textbox>
    <asp:requiredfieldvalidator id="Requiredfieldvalidator2" runat="server" ErrorMessage="This is a required value." ControlToValidate="EditDuration" display="Dynamic"></asp:requiredfieldvalidator>
    <!-- 定义验证控件验证输入字符是否为数字型 -->
    <asp:CompareValidator id="CompareValidator1" runat="server" ErrorMessage="Hours must be an integer value" ControlToValidate="EditDuration" Display="Dynamic" Type="Integer" Operator="DataTypeCheck">
</asp:CompareValidator>
</EditItemTemplate>
</asp:TemplateColumn>
<!-- 定义编辑按钮的模板列 -->
<asp:TemplateColumn HeaderText="Edit">
    <HeaderStyle HorizontalAlign="Left" Width="50px" CssClass="grid-header" VerticalAlign="Middle"></HeaderStyle>
    <ItemStyle HorizontalAlign="Center" CssClass="grid-edit-column"></ItemStyle>
    <ItemTemplate>
        <!-- 定义编辑按钮 -->
        <asp:imagebutton runat="server" ImageUrl="images/icon-pencil.gif" CommandName="Edit" CausesValidation="false" ID="EditButton"></asp:imagebutton>
        
        <!-- 定义删除按钮 -->
        <asp:imagebutton Runat="server" ImageUrl="images/icon-delete.gif" CommandName="Delete" CausesValidation="False" ID="CatDeleteButton"></asp:imagebutton>
    </ItemTemplate>
    <EditItemTemplate>
        <!-- 定义更新按钮 -->
        <asp:imagebutton runat="server" ImageUrl="images/icon-floppy.gif" CommandName="Update" CausesValidation="True" ID="UpdateButton"></asp:imagebutton>
        
        <!-- 定义取消更新按钮 -->
        <asp:imagebutton runat="server" ImageUrl="images/icon-pencil-x.gif" CommandName="Cancel" CausesValidation="false" ID="CatCancelButton"></asp:imagebutton>
    </EditItemTemplate>
</asp:TemplateColumn>
</Columns>
</asp:datagrid>
```

15.4 工作进程管理

工作进程管理页面 TimeEntry.aspx 为用户登录系统之后默认的主页面，用户可以在该页面上分配管理自己的工作进程，管理员和项目经理也可以根据权限查看和管理其他项目成员的工作进程。

15.4.1 数据库设计

工作进程管理部分所调用的存储过程有如下几个。

(1) AddTimeEntry

存储过程 AddTimeEntry 向数据库添加一条工作进程数据，并返回新添加的工作进程的 EntryLogID。

(2) DeleteTimeEntry

存储过程 DeleteTimeEntry 为删除一条工作进程的存储过程。

(3) GetTimeEntry

存储过程 GetTimeEntry 为根据参数 EntryLogID 获得一条工作进程数据的存储过程。

(4) ListTimeEntries

存储过程 ListTimeEntries 为返回一个用户在一段时间内的所有工作进程列表的存储过程，该存储过程根据调用者的权限返回工作进程列表。如果调用该存储过程的用户角色为项目经理则该用户只可以查询自己担任项目经理的项目中用户的工作进程列表，如果用户角色为管理员或是调用者查询本人的工作进程，则返回所有的工作进程。该存储过程的代码清单如下。

```
CREATE PROCEDURE TT_ListTimeEntries
(
    @QueryUserID int,
    @UserID int,
    @StartDate datetime,
    @EndDate datetime
)
AS
//定义变量 QueryUserRoleID 并获得调用者的角色信息
DECLARE @@QueryUserRoleID int

SELECT @@QueryUserRoleID = TT_Users.RoleID FROM TT_Users WHERE TT_Users.UserID = @QueryUserID
//当用户角色为项目经理则该用户只可以查询自己担任项目经理的项目中的用户工作进程列表
IF @@QueryUserRoleID = 2
BEGIN
    SELECT
        EntryLogID, TT_EntryLog.Description, Duration, EntryDate, TT_EntryLog.ProjectID AS ProjectID,
        TT_EntryLog.CategoryID AS CategoryID, TT_Categories.Abbreviation AS CategoryName,
        TT_Projects.Name AS ProjectName,
        ManagerUserID, TT_Categories.Abbreviation AS CatShortName
    FROM
        TT_EntryLog
        INNER JOIN
        TT_Categories
        ON
        TT_EntryLog.CategoryID = TT_Categories.CategoryID
        INNER JOIN
        TT_Projects
        ON
        TT_EntryLog.ProjectID = TT_Projects.ProjectID
    WHERE
        UserID = @UserID
        AND
        Convert(nvarchar, EntryDate, 1) >= Convert(nvarchar, @StartDate, 1)
        AND
        Convert(nvarchar, EntryDate, 1) <= Convert(nvarchar, @EndDate, 1)
```

```
AND
ManagerUserID = @QueryUserID
END
//用户角色为管理员或是调用者查询本人的工作进程，则返回所有的工作进程
ELSE IF @@QueryUserRoleID = 1 or @QueryUserID = @UserID
BEGIN
SELECT
    EntryLogID, TT_EntryLog.Description, Duration, EntryDate, TT_EntryLog.ProjectID AS ProjectID,
    TT_EntryLog.CategoryID AS CategoryID, TT_Categories.Abbreviation AS CategoryName,
    TT_Projects.Name AS ProjectName,
    ManagerUserID, TT_Categories.Abbreviation AS CatShortName
FROM
    TT_EntryLog
    INNER JOIN
    TT_Categories
    ON
        TT_EntryLog.CategoryID = TT_Categories.CategoryID
    INNER JOIN
    TT_Projects
    ON
        TT_EntryLog.ProjectID = TT_Projects.ProjectID
WHERE
    UserID = @UserID
    AND
    Convert(nvarchar, EntryDate, 1) >= Convert(nvarchar, @StartDate, 1)
    AND
    Convert(nvarchar, EntryDate, 1) <= Convert(nvarchar, @EndDate, 1)
END
```

(5) UpdateTimeEntry

存储过程 UpdateTimeEntry 为更新一条工作进程的存储过程。

15.4.2 逻辑层

工作进程管理的逻辑层为类 TimeEntry，位于文件 TimeEntry.cs 中。该类中定义了工作进程对象所拥有的属性及方法。

首先该类采用构造器的结构定义了工作进程对象的属性，其中包括：CategoryID、CategoryName、CategoryShortName、Day、Description、Duration、EntryDate、EntryDate、ProjectID 和 ProjectName。构造器定义部分的代码省略。

TimeEntry 有 3 个构造方法，代码如下。

```
public TimeEntry()
{
}

public TimeEntry(int entryLogID)
{
    _entryLogID = entryLogID;
}
```

```

    public TimeEntry(int entryLogID, int userID, int ProjectID, int CategoryID, DateTime EntryDate, string
Description, decimal Duration)
{
    _entryLogID = entryLogID;
    _userID = userID;
    _projectID = ProjectID;
    _categoryID = CategoryID;
    _entryDate = EntryDate;
    _description = Description;
    _duration = Duration;
}

```

FillCorrectStartEndDates 方法为根据方法参数表中的 selectedDate 日期确定该日期所在周的开始日期和结束日期的方法，因为在 ASP.NET Time Tracker Starter Kit 系统的时间管理上，可以将一周 7 天中的任意一天设置成为该周的第一天，因此就需要使用 FillCorrectStartEndDates 方法来判断一周的开始日期和结束日期。

```

    public static void FillCorrectStartEndDates(DateTime selectedDate, ref DateTime startDate, ref
DateTime endDate)
{
    //从 Web.config 中获得系统中设置的一周的开始
    int firstDayOfWeek = Convert.ToInt32(ConfigurationSettings.AppSettings[Web.Global.
CfgKeyFirstDayOfWeek]);

    //对一周 7 天做一次循环
    for(int i = 0; i<7; i++)
    {
        //获取一周的起始点
        if (Convert.ToInt32(selectedDate.AddDays(i).DayOfWeek) == firstDayOfWeek)
        {
            //填写开始和结束日期
            startDate = selectedDate.AddDays(i);
            if(i!=0)
                startDate = startDate.AddDays(-7);
            endDate = startDate.AddDays(6);
            break;
        }
    }
}

```

GetEntries 方法为获得某个用户一段时间内的工作条目信息的方法，该方法调用存储过程 TT_ListTimeEntries，将返回的数据存储于工作进程的对象集合类 TimeEntriesCollection 中，对象集合类 TimeEntriesCollection 是用于存储工作进程对象 TimeEntry 的对象集合类。

```

    public static TimeEntriesCollection GetEntries(int queryUserID, int userID, DateTime startDate,
DateTime endDate)
{
    //执行存储过程 TT_ListTimeEntries 获得数据
    DataSet dsData = SqlHelper.ExecuteDataset(
        ConfigurationSettings.AppSettings[Web.Global.CfgKeyConnString],
        "TT_ListTimeEntries", queryUserID, userID, startDate, endDate);
    TimeEntriesCollection entryList = new TimeEntriesCollection();
}

```

```
//从DataSet中获得数据形成工作进程对象并加入对象集合类
foreach(DataRow row in dsData.Tables[0].Rows)
{
    TimeEntry time = new TimeEntry();
    time.EntryLogID = Convert.ToInt32(row["EntryLogID"]);
    time.Description = row["Description"].ToString();
    time.Duration = Convert.ToDecimal(row["Duration"]);
    time.EntryDate = Convert.ToDateTime(row["EntryDate"]);
    time.ProjectID = Convert.ToInt32(row["ProjectID"]);
    time.CategoryID = Convert.ToInt32(row["CategoryID"]);
    time.CategoryName = row["CategoryName"].ToString();
    time.ProjectName = row["ProjectName"].ToString();
    time.Day = time.EntryDate.ToString("dddd");
    time.CategoryShortName = row["CatShortName"].ToString();

    entryList.Add(time);
}
//返回对象集合类
return entryList;
}
```

GetWeek方法返回参数表中selectedDate日期所在周7天的一个DataTable，该DataTable包含两列：一列是日期，一列是星期。该方法绑定了添加工作进程表单中的 DropDownList 控件。

```
public static DataTable GetWeek(DateTime selectedDate)
{
    DateTime start = DateTime.MinValue;
    DateTime end = DateTime.MinValue;
    DataTable dt = new DataTable();
    dt.Columns.Add("Day");
    dt.Columns.Add("Date");
    //调用FillCorrectStartEndDates方法获得传入日期所在周的开始日期和结束日期
    FillCorrectStartEndDates(selectedDate, ref start, ref end);
    DataRow workRow;

    //构建DataTable
    for (int i = 0; i < 7; i++)
    {
        workRow = dt.NewRow();
        workRow["Day"] = Convert.ToDateTime(start.AddDays(i)).ToString("ddd");
        workRow["Date"] = start.AddDays(i);
        dt.Rows.Add(workRow);
    }
    return dt;
}
```

Remove方法为删除指定工作进程的方法，该方法调用了存储过程TT_DeleteTimeEntry。

```
public static void Remove(int entryLogID)
{
    SqlHelper.ExecuteNonQuery(ConfigurationSettings.AppSettings[Web.Global.CfgKeyConnString],
    "TT_DeleteTimeEntry", entryLogID);
```

```
}
```

Load 方法获得指定工作进程信息并绑定到当前工作进程对象的属性，该方法调用了存储过程 TT_GetTimeEntry。

```
public void Load()
{
    DataSet ds = SqlHelper.ExecuteDataset(ConfigurationSettings.AppSettings[Web.Global.CfgKeyConnString],
"TT_GetTimeEntry", _entryLogID);
    DataRow row = ds.Tables[0].Rows[0];
    _description = row["Description"].ToString();
    _duration = Convert.ToDecimal(row["Duration"]);
    _entryDate = Convert.ToDateTime(row["EntryDate"]);
    _projectId = Convert.ToInt32(row["ProjectID"]);
    _userID = Convert.ToInt32(row["UserID"]);
    _categoryID = Convert.ToInt32(row["CategoryID"]);
    _ projectName = row["ProjectName"].ToString();
}
```

Save 方法首先判断私有成员变量 _entryLogID 是否为 0，如果为 0，则调用 Insert 方法向数据库添加一条工作进程，否则调用 Update 方法更新工作进程。

```
public bool Save()
{
    if (_entryLogID == 0)
        return Insert();
    else
        return Update();
}
```

Insert 方法为向数据库中新增一条工作进程的方法，返回值为布尔型。

```
private bool Insert()
{
    //调用存储过程 TT_AddTimeEntry 添加一条工作进程的记录
    _entryLogID = Convert.ToInt32(SqlHelper.ExecuteScalar(ConfigurationSettings.AppSettings[Web.Global.CfgKeyConnString],
"TT_AddTimeEntry", _userID, _projectId, _categoryID,
_entryDate, _description, _duration));
    return (0 < _entryLogID);
}
```

Update 方法为更新一条工作进程的方法，返回值同样为布尔型。

```
private bool Update()
{
    //调用存储过程 TTUpdateTimeEntry 更新一条工作进程的记录
    try
    {
        SqlHelper.ExecuteNonQuery(ConfigurationSettings.AppSettings[Web.Global.CfgKeyConnString],
"TTUpdateTimeEntry", _entryLogID, _userID, _projectId, _categoryID,
_entryDate, _description, _duration);
        return true;
    }
    catch {
        return false;
    }
}
```

15.4.3 应用层

工作进程管理部分的页面 TimeEntry.aspx 的后台编码类 TimeEntry.aspx.cs 中包含了工作进程管理部分的应用层代码。TimeEntry.aspx.cs 的主要方法代码如下。

首先是页面载入 Page_Load 方法，该方法中定义了页面载入事件的代码。

```
private void Page_Load(object sender, System.EventArgs e)
{
    //实例化一个 User 对象，获得当前用户信息
    _user = new TTUser(
        TTSecurity.GetUserID(),
        User.Identity.Name,
        TTSecurity.GetName(),
        TTSecurity.GetUserRole());
    //如果页面是首次载入则把当前日期作为页面中的默认日期
    if (!Page.IsPostBack)
    {
        //将当前日期填充到页面中
        BusinessLogicLayer.TimeEntry.FillCorrectStartEndDates(DateTime.Today,
            ref _weekStartingDate, ref _weekEndingDate);
        WeekEnding.Text = _weekEndingDate.ToShortDateString();

        BindUserList();
        BindEntryFields();
        BindTimeSheet(_user.UserID, _weekStartingDate, _weekEndingDate);
    }
    else
    {
        //如果页面被提交之后，则按照控件 WeekEnding 中选定的日期绑定页面信息
        BusinessLogicLayer.TimeEntry.FillCorrectStartEndDates(Convert.ToDateTime(WeekEnding.Text),
            ref _weekStartingDate, ref _weekEndingDate);
        _dayListTable = BusinessLogicLayer.TimeEntry.GetWeek(Convert.ToDateTime(WeekEnding.Text));
    }
}
```

BindCategoryList 方法为绑定页面中的下拉列表控件 CategoryList 的方法，该控件绑定了每一个项目的工作分类。

```
private void BindCategoryList()
{
    if (ProjectList.SelectedItem != null)
    {
        //根据下拉列表控件 ProjectList 的选定获得项目工作分类列表
        CategoryList.DataSource =
Project.GetCategories(Convert.ToInt32(ProjectList.SelectedItem.Value));
        //指定下拉列表控件 CategoryList 的 DataValueField 和 DataTextField
        CategoryList.DataValueField = "CategoryID";
        CategoryList.DataTextField = "Abbreviation";
        //绑定下拉列表控件 CategoryList
        CategoryList.DataBind();
    }
}
```

}

BindDates 方法为绑定页面中下拉列表控件 Days 的方法，下拉列表控件 Days 显示当前周的日期。

```
private void BindDates()
{
    //调用 GetWeek 方法获得当前周日期和星期所对应的 DataTable
    _dayListTable = BusinessLogicLayer.TimeEntry.GetWeek(Convert.ToDateTime(WeekEnding.Text));
    //指定 Days 控件的数据源以及 DataValueField 和 DataTextField 并绑定控件
    Days.DataSource = _dayListTable;
    Days.DataValueField = "Date";
    Days.DataTextField = "Day";
    Days.DataBind();
    Days.Items.FindByText(Convert.ToDateTime(DateTime.Today).ToString("ddd")).Selected = true;
}
```

BindProjectList 方法绑定了页面中下拉列表控件 ProjectList，下拉列表控件 ProjectList 中显示所有项目列表。

```
private void BindProjectList()
{
    //如果控件 ProjectList 的选定值不为空，则记录控件的选定值
    if (ProjectList.SelectedItem != null)
        _userInput.ProjectID = Convert.ToInt32(ProjectList.SelectedItem.Value);
    //调用 ListUserProjects 方法为控件获得数据源并绑定控件
    ProjectList.DataSource = ListUserProjects();
    ProjectList.DataTextField = "Name";
    ProjectList.DataValueField = "ProjectID";
    ProjectList.DataBind();

    //如果控件的选定值不为空，则重新绑定控件
    if (ProjectList.Items.FindByValue(_userInput.ProjectID.ToString()) != null)
        ProjectList.Items.FindByValue(_userInput.ProjectID.ToString()).Selected = true;
}
```

DrawTimeGraph 方法为调用动态绘制图片的方法，该方法所绘制的图片显示的是当前查看用户在当前选定周内的工作量柱状图，关于绘制图片的具体方法将在下一章中介绍。DrawTimeGraph 方法本身并没有调用绘制图片的方法，而是通过调用 TimeEntryBarChart.aspx 并以 QueryString 形式传入参数的方法。参数包括 xValues 和 yValues，其中 xValues 中以 | 为分隔符记录一周 7 天的日期，yValues 以 | 为分隔符记录一周 7 天的工作量。TimeEntryBarChart.aspx 通过处理之后返回的图片形式的字节流来完成动态显示图片的过程。

```
private void DrawTimeGraph(TimeEntriesCollection chartData, DataTable dtWeek)
{
    //建立一个 HasTable 用以存储日期和工作量的对应键值对
    Hashtable htWeekSummary = new Hashtable(7);

    StringBuilder xValues = new StringBuilder();
    StringBuilder yValues = new StringBuilder();

    //为 htWeekSummary 中的日期列赋值
    foreach (DataRow row in dtWeek.Rows)
    {
        htWeekSummary.Add(Convert.ToDateTime(row["Date"]).ToShortDateString(), 0m);
    }
}
```

```
}

//为 htWeekSummary 中的工作量列赋值
foreach (BusinessLogicLayer.TimeEntry time in chartData)
{
    //累加得到每一天的工作量总和
    string key = time.EntryDate.ToShortDateString();
    Decimal dayDuration = Convert.ToDecimal(htWeekSummary[key]);
    dayDuration += time.Duration;
    htWeekSummary[key] = dayDuration;
}

//遍历 htWeekSummary 生成 xValues 和 yValues 两个字符串
int index = 1;
foreach (DataRow row in dtWeek.Rows)
{
    string key = Convert.ToDateTime(row["Date"]).ToShortDateString();
    string currentDay = Convert.ToDateTime(key).ToString("ddd");
    string currentHour = Convert.ToString(htWeekSummary[key]);

    xValues.Append(currentDay);
    yValues.Append(currentHour);

    if (index != dtWeek.Rows.Count)
    {
        xValues.Append("|");
        yValues.Append("|");
    }
    index++;
}

//形成 URL
TimeGraph.ImageUrl = "TimeEntryBarChart.aspx?" +
    "xValues=" + xValues.ToString() +
    "&yValues=" + yValues.ToString();
}
```

在 TimeEntry.aspx 的前台页面当前周工作进程列表采用 DataGrid 方式展现，该 DataGrid 名为 TimeEntryGrid，其所对应事件触发的方法在 TimeEntry.aspx.cs 中都有定义，例如 TimeEntryGrid_Itembound、TimeEntryGrid_OnEdit、TimeEntryGrid_OnUpdate、TimeEntryGrid_OnDelete 和 TimeEntryGrid_OnCancel 等。下面分别介绍几个由 DataGrid 对应的事件所触发的方法。

TimeEntryGrid_OnEdit 方法是在 TimeEntryGrid 中单击编辑某行时所触发的方法，该方法是将被编辑行的数据保存到一个工作进程对象 _userInput 中，供 TimeEntryGrid_Itembound 方法在对编辑行控件赋值时使用。

```
protected void TimeEntryGrid_OnEdit(Object sender, DataGridCommandEventArgs e)
{
    TimeEntryGrid.EditItemIndex = e.Item.ItemIndex;

    //将被编辑行的数据保存到 _userInput 对象中
    _userInput.Day = ((Label)e.Item.FindControl("EntryDay")).Text;
```

```

    _userInput.ProjectName = ((Label) e.Item.FindControl("EntryProject")).Text.Trim();
    _userInput.ProjectID = Convert.ToInt32(((Label) e.Item.FindControl("EntryProjectID")).Text);
    _userInput.CategoryName = ((Label) e.Item.FindControl("EntryCategory")).Text.Trim();
    _userInput.Duration = Convert.ToDecimal(((Label) e.Item.FindControl("EntryDuration")).Text);
    _userInput.Description = ((Label) e.Item.FindControl("GridDescription")).Text.Trim();
    BindTimeSheet(_user.UserID, Convert.ToInt32(UserList.SelectedItem.Value), _weekStartingDate,
    _weekEndingDate);
}

```

TimeEntryGrid_Itembound 方法是 TimeEntryGrid 在生成每行数据时所触发的方法，该方法的主要功能是绑定被编辑行中控件的值。该方法代码如下。

```

private void TimeEntryGrid_Itembound(object sender, System.Web.UI.WebControls.DataGridItemEventArgs e)
{
    //当生成行为被编辑行时，则从_userInput 对象中获得并绑定编辑状态下的控件值
    if (e.Item ItemType == ListItemType.EditItem)
    {
        DropDownList currentCbo = (DropDownList) e.Item.FindControl("EntryDays");
        currentCbo.SelectedIndex = currentCbo.Items.IndexOf(currentCbo.Items.FindByText(_userInput.Day));

        currentCbo = (DropDownList) e.Item.FindControl("EntryProjects");
        currentCbo.SelectedIndex = currentCbo.Items.IndexOf(currentCbo.Items.FindByText(_userInput.
ProjectName));

        currentCbo = (DropDownList) e.Item.FindControl("EntryCategories");
        currentCbo.SelectedIndex = currentCbo.Items.IndexOf(currentCbo.Items.FindByText(_userInput.
CategoryName));
    }
}

```

TimeEntryGrid_OnUpdate 方法是 TimeEntryGrid 在编辑状态下单击保存当前编辑行按钮所触发的事件。该方法在更新数据之前调用验证控件对将要更新的数据进行验证，然后对数据进行更新。

```

protected void TimeEntryGrid_OnUpdate(Object sender, DataGridCommandEventArgs e)
{
    DataGridItem item = TimeEntryGrid.Items[TimeEntryGrid.EditItemIndex];
    //调用验证空间对数据进行验证
    RequiredFieldValidator required = (RequiredFieldValidator)
item.FindControl("RequiredFieldValidator_GridHours");
    CompareValidator comparer = (CompareValidator) item.FindControl("CompareValidatorGridHours");

    required.Validate();
    comparer.Validate();

    //如果数据通过验证则实例化一个工作进程对象并对其属性赋值，然后更新数据
    if (required.IsValid && comparer.IsValid)
    {
        int entryLogID;
        int userID;
        int projectID;
        int categoryID;
        DateTime taskDate;

```

```
        string description;
        decimal duration;
        BusinessLogicLayer.TimeEntry te = null;

        entryLogID = Convert.ToInt32(TimeEntryGrid.DataKeys[(int)e.Item.ItemIndex]);
        userID = Convert.ToInt32(UserList.SelectedItem.Value);
        projectID = Convert.ToInt32((DropDownList)
e.Item.FindControl("EntryProjects").SelectedItem.Value);
        categoryID = Convert.ToInt32((DropDownList)
e.Item.FindControl("EntryCategories").SelectedItem.Value);
        taskDate = Convert.ToDateTime((DropDownList)
e.Item.FindControl("EntryDays").SelectedItem.Value);
        description = TTSecurity.CleanStringRegex((TextBox)
e.Item.FindControl("EntryDescription")).Text;
        duration = Convert.ToDecimal((TextBox) e.Item.FindControl("EntryHours")).Text;

        te = new BusinessLogicLayer.TimeEntry(entryLogID, userID, projectID, categoryID, taskDate,
description, duration);
        te.Save();

        TimeEntryGrid.EditItemIndex = -1;
        BindTimeSheet(_user.UserID, Convert.ToInt32(UserList.SelectedItem.Value),
_weekStartingDate, _weekEndingDate);
    }
}
```

UserProjects_OnChange 方法是 TimeEntryGrid 在编辑状态下更改项目下拉列表选定值时所触发的方法，该方法对编辑状态下的项目工作分类列表进行了重新绑定。

```
protected void UserProjects_OnChange(object sender, System.EventArgs e)
{
    DataGridItem item = TimeEntryGrid.Items[TimeEntryGrid.EditItemIndex];

    //保存当前编辑状态下的控件值
    if (item != null)
    {
        TextBox txt = (TextBox) item.FindControl("EntryHours");
        _userInput.Duration = Convert.ToDecimal(txt.Text);

        TextBox txtDesc = (TextBox) item.FindControl("EntryDescription");
        _userInput.Description = txtDesc.Text;

        DropDownList EntryDays = (DropDownList) item.FindControl("EntryDays");
        _userInput.EntryDate = Convert.ToDateTime(EntryDays.SelectedItem.Value);
        _userInput.Day = EntryDays.SelectedItem.Text;

    }
    //获得下拉列表选定值更改之后的值并重新进行绑定
    DropDownList EntryProjects = (DropDownList) sender;
    _userInput.ProjectID = Convert.ToInt32(EntryProjects.SelectedItem.Value);
}
```

```
    _userInput.ProjectName = EntryProjects.SelectedItem.Text;
    BindTimeSheet(_user.UserID, Convert.ToInt32(UserList.SelectedItem.Value), _weekStartingDate,
    _weekEndingDate);
}
```

以上介绍了 TimeEntry.aspx.cs 中的部分主要方法，其他的方法由于思路比较清晰，本书就不对其代码进行罗列。

15.4.4 表示层

在 TimeEntry.aspx 页面中显示当前周工作进程的 DataGrid 为页面的主体部分，因为该 DataGrid 在后台定义了相应事件所触发的方法，因此在前台页面的展现上逻辑较为复杂。该 DataGrid 的代码清单如下。

```
<!-- 首先定义了 DataGrid 的一些基本属性 -->
<asp:datagrid id="TimeEntryGrid" runat="server" Width="100%" BorderStyle="None" CellPadding="2"
AutoGenerateColumns="False" Font-Name="Verdana" FontSize="11px" AllowSorting="True" DataKeyField="EntryLogID"
BorderColor="White">

<headerstyle font-bold="True" cssclass="grid-header"></headerstyle> --
<columns>
    <!-- 工作进程日期模板列 -->
    <asp:templatecolumn SortExpression="EntryDate" HeaderText="Day">
        <headerstyle horizontalalign="Center" width="80px" cssclass="grid-header" verticalalign="Middle">
    </headerstyle>
        <itemstyle horizontalalign="Center" cssclass="grid-first-item"></itemstyle>
        <!-- 工作进程日期模板列在普通模式下的模板 -->
        <itemtemplate>
            <asp:label ID="EntryDay" Text='<%# DataBinder.Eval(Container, "DataItem.EntryDate", "{0:ddd}") %>' Runat="server" Width='40px' />
        </itemtemplate>
        <!-- 工作进程日期模板列在编辑模式下的模板 -->
        <edititemtemplate>
            <asp:dropdownlist Width="48px" ID="EntryDays" CssClass="Standard-text" DataSource='<%# _dayListTable %>' DataTextField = "Day" DataValueField = "Date" Runat="server"></asp:dropdownlist>
        </edititemtemplate>
    </asp:templatecolumn>
    <!-- 项目名称模板列 -->
    <asp:templatecolumn SortExpression="ProjectName" HeaderText="Project">
        <headerstyle horizontalalign="Center" width="40px" cssclass="grid-header" verticalalign="Middle">
    </headerstyle>
        <itemstyle width="108px" cssclass="grid-item"></itemstyle>
        <!-- 项目名称模板列在普通模式下的模板 -->
        <itemtemplate>
            <asp:label ID="EntryProject" Text='<%# DataBinder.Eval(Container, "DataItem.ProjectName") %>' Runat="server" />
            <asp:label ID="EntryProjectID" Text='<%# DataBinder.Eval(Container, "DataItem.ProjectID") %>' Runat="server" Visible="False" />
        </itemtemplate>
        <!-- 项目名称模板列在编辑模式下的模板 -->
        <edititemtemplate>
            <asp:dropdownlist Width="100px" ID="EntryProjects" AutoPostBack="True" CssClass="Standard-text" DataSource='<%# ListUserProjects() %>' DataTextField="Name" DataValueField="ProjectID" Runat="server" OnSelectedIndexChanged="UserProjects_OnChange" />
        </edititemtemplate>
    </asp:templatecolumn>
    <!-- 工作分类模板列 -->
    <asp:templatecolumn SortExpression="CategoryName" HeaderText="Category">
```

```
<headerstyle horizontalalign="Center" width="40px" cssclass="grid-header" verticalalign="Middle">
</headerstyle>
<itemstyle width="88px" cssclass="grid-item"></itemstyle>
<!-- 工作分类模板列在普通状态下的模板 --&gt;
&lt;itemtemplate&gt;
    &lt;asp:label ID="EntryCategory" Text='&lt;%# DataBinder.Eval(Container, "DataItem.CategoryName") %&gt;' Runat="server" /&gt;
&lt;/itemtemplate&gt;
<!-- 工作分类模板列在编辑状态下的模板 --&gt;
&lt;edititemtemplate&gt;
    &lt;asp:dropdownlist Width="80px" ID="EntryCategories" CssClass="Standard-text" DataSource='&lt;%# ListGridCategories(_userInput.ProjectID) %&gt;' DataTextField="Abbreviation" DataValueField="CategoryID" Runat="server"&gt;
        &lt;/asp:dropdownlist&gt;
    &lt;/edititemtemplate&gt;
&lt;/asp:templatecolumn&gt;
<!-- 工作量模板列 --&gt;
&lt;asp:templatecolumn SortExpression="Duration" HeaderText="Hours"&gt;
    &lt;headerstyle horizontalalign="Center" width="40px" cssclass="grid-header" verticalalign="Middle"&gt;
&lt;/headerstyle&gt;
    &lt;itemstyle horizontalalign="Right" width="50px" cssclass="grid-item"&gt;&lt;/itemstyle&gt;
    <!-- 工作量模板列在普通状态下的模板 --&gt;
    &lt;itemtemplate&gt;
        &lt;asp:label ID="EntryDuration" Text='&lt;%# DataBinder.Eval(Container.DataItem, "Duration", "{0:f}") %&gt;' Runat="server" /&gt;
    &lt;/itemtemplate&gt;
    <!-- 工作量模板列在编辑状态下的模板 --&gt;
    &lt;edititemtemplate&gt;
        &lt;asp:textbox Width="40px" AutoPostBack=false CssClass="Standard-text" Runat="server" ID="EntryHours" Text='&lt;%# _userInput.Duration %&gt;' /&gt;
        &lt;asp:requiredfieldvalidator id="RequiredFieldValidatorGridHours" runat="server" ErrorMessage="Hours is required." ControlToValidate="EntryHours" Display="Dynamic"&gt;&lt;/asp:requiredfieldvalidator&gt;
        &lt;asp:comparevalidator id="CompareValidatorGridHours" runat="server" ErrorMessage="Hours must be numeric value." ControlToValidate="EntryHours" Display="Dynamic" Type="Currency" Operator="DataTypeCheck"&gt;
    &lt;/asp:comparevalidator&gt;
    &lt;/edititemtemplate&gt;
&lt;/asp:templatecolumn&gt;
<!-- 工作描述模板列--&gt;
&lt;asp:templatecolumn SortExpression="Description" HeaderText="Description"&gt;
    &lt;headerstyle horizontalalign="Left" cssclass="grid-header" verticalalign="Middle"&gt;&lt;/headerstyle&gt;
    &lt;itemstyle cssclass="grid-item"&gt;&lt;/itemstyle&gt;
    <!-- 工作描述模板列在普通状态下的模板--&gt;
    &lt;itemtemplate&gt;
        &lt;asp:label ID="GridDescription" Text='&lt;%# DataBinder.Eval(Container, "DataItem.Description") %&gt;' Runat="server" /&gt;
    &lt;/itemtemplate&gt;
    <!-- 工作描述模板列在编辑状态下的模板--&gt;
    &lt;edititemtemplate&gt;
        &lt;asp:textbox runat="server" CssClass="Standard-text" ID="EntryDescription" Text='&lt;%# _userInput.Description %&gt;' Width="250" MaxLength="255"&gt;&lt;/asp:textbox&gt;
    &lt;/edititemtemplate&gt;
&lt;/asp:templatecolumn&gt;</pre>
```

```
</edititemtemplate>
</asp:templatecolumn>
<!-- 按钮列 -->
<asp:templatecolumn HeaderText="Edit">
    <headerstyle horizontalalign="Center" width="40px" cssclass="grid-header" verticalalign="Middle">
</headerstyle>
    <itemstyle cssclass="grid-edit-column"></itemstyle>
    <!-- 普通状态的按钮列 -->
    <itemtemplate>
        <!-- 编辑按钮 -->
        <asp:imagebutton runat="server" ImageUrl="images/icon-pencil.gif" AlternateText="Edit" CommandName="Edit" CausesValidation="false" ID="Imagebutton1" NAME="Imagebutton1"></asp:imagebutton>
        
        <!-- 删除按钮 -->
        <asp:imagebutton Runat="server" ImageUrl="images/icon-delete.gif" AlternateText="Delete" CommandName="Delete" CausesValidation="False" ID="Imagebutton2" NAME="Imagebutton2"></asp:imagebutton>
    </itemtemplate>
    <!-- 编辑状态的按钮列 -->
    <edititemtemplate>
        <!-- 更新按钮 -->
        <asp:imagebutton runat="server" ImageUrl="images/icon-floppy.gif" AlternateText="Update" CommandName="Update" CausesValidation="False" ID="Imagebutton3" NAME="Imagebutton3"></asp:imagebutton>
        
        <!-- 取消按钮 -->
        <asp:imagebutton runat="server" ImageUrl="images/icon-pencil-x.gif" AlternateText="Cancel" CommandName="Cancel" CausesValidation="False" ID="Imagebutton4" NAME="Imagebutton4"></asp:imagebutton>
    </edititemtemplate>
    </asp:templatecolumn>
</columns>
</asp:datagrid>
```

15.5 开发启示

本章讲解了 ASP.NET Time Tracker Starter Kit 中最为重要的两个模块：项目管理模块与工作进程管理模块。项目管理模块和工作进程管理模块在业务逻辑上比较复杂，包含了 ASP.NET Time Tracker Starter Kit 中大部分的业务逻辑。希望通过本章的学习，读者能够掌握怎样在实际的业务逻辑和具体的技术实现之间架设一道桥梁，并且能够理解如下技术方面的内容。

- DataGrid 的高级应用
- 存储过程中执行错误的判断
- 使用 ViewState 存储页面内部变量

第 16 章

数据报表

数据报表一直是大部分和数据相关的应用程序中必不可少的模块。数据报表模块是按照客户实际需求对系统内部数据进行有规划的呈现，用户通过对数据报表的查看就可以对系统内部数据进行比较和分析，对系统的数据进行宏观的把握。

ASP.NET Time Tracker Starter Kit 中的数据报表模块主

要分为页面形式的报表和图片形式的报表。页面形式的报表通过 ASP.NET 页面呈现数据，而图片形式的报表则使用 GDI+技术动态生成报表图片。

16.1 系统设计

16.1.1 功能设计

ASP.NET Time Tracker Starter Kit 的主要功能是管理项目进度并安排项目成员工作。因此对项目和项目人员的数据统计显得尤为重要。对项目数据的统计可以查看某个项目的时间安排情况，对项目人员的数据统计可以查看该成员在所有项目中的工作分配情况。每个成员也可以查看自己在某一段时间内的工作量统计情况。

(1) 项目数据统计

项目数据统计的主要功能是汇总并列出某个项目中具体的工作分配条目、列出项目的总体工作量以及目前完成的工作量、分别列出项目中具体工作条目以及工作的完成人。浏览者可以对项目的进展以及工作分配情况一目了然。图 16-1 所示为项目数据的统计报表。

图 16-1 中的区域 A 为项目整体数据统计，区域 B 为项目工作分类的数据统计，区域 C 为项目成员在该项目中的工作量统计。

(2) 个人数据统计

个人数据统计的主要功能是显示一段时间内个人的总体工作量以及个人完成的每一项工作的详细数据。每一项工作的详细数据包括完成日期、项目名称、工作分类、工作时间以及工作描述等。图 16-2 所示为个人数据统计报表。

图中区域 A 为该数据报表的统计的开始时间和结束时间，区域 B 为被统计人的总体工作量，区域 C 为被统计人的详细工作条目列表。

| Project Name | Est. Hours | Actual Hours | Est. Completion |
|---------------------------|------------|--------------|-----------------|
| Workflow Platform Project | 100.00 | 1.00 | 5/27/2005 |
| Category | Est. Hours | Actual Hours | |
| code | 100 | 1.00 | |
| Consultant | Hours | | |
| Vencent Sun | 1.00 | | |

图 16-1 项目数据统计报表

| Beginning Date | Ending Date | | | |
|---------------------|-----------------|------|-------|-------------|
| 3/1/2005 | 6/12/2005 | | | |
| Consultant | Total Hours | | | |
| King Lau | 30.00 | | | |
| Resource Log | | | | |
| Date | Project | Cat. | Hrs. | Description |
| 5/3/2005 | Asp.Net Project | des | 3.00 | |
| 5/18/2005 | Asp.Net Project | des | 10.00 | |
| 5/14/2005 | Asp.Net Project | des | 10.00 | sss |
| 5/10/2005 | Asp.Net Project | des | 7.00 | 11 |

图 16-2 个人数据统计报表

(3) 个人每周工作量图表

个人每周工作量图表以图表的形式显示个人在一周 7 天内的每一天的工作量。图 16-3 所示为个人每周工作量图表。

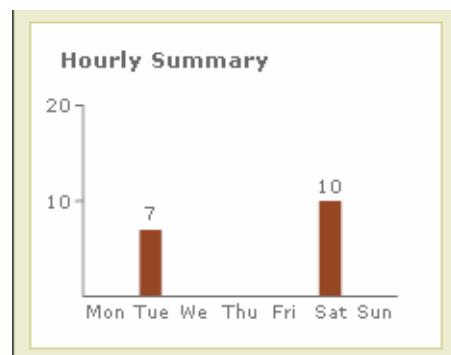


图 16-3 个人每周工作量图表

16.1.2 数据库设计

由于数据报表部分是对当前数据库中的数据进行查询，因此本部分的数据库设计工作主要体现在查询的存储过程的设计上。以下是数据报表部分所使用到的存储过程详解。

(1) ListCategoriesByProject

存储过程 ListCategoriesByProject 是返回某个项目的所有工作分类数据的存储过程。该存储过程根据传入参数 ProjectID 返回该项目中所有工作分类的名称、简称、工作总量以及完成工作量。该存储过程的代码清单如下。

```
CREATE PROCEDURE TT_ListCategoriesByProject
(
    @ProjectID int
)
AS

SELECT C.CategoryID,
       C.Name,
       C.Abbreviation AS CategoryShortName,
       C.EstDuration,
       ISNULL(Sum(EL.Duration), 0) AS ActualHours
FROM TT_Categories C
LEFT OUTER JOIN TT_EntryLog EL
ON C.CategoryID = EL.CategoryID
WHERE C.ProjectID = @ProjectID
GROUP BY C.CategoryID, C.Name, C.Abbreviation, C.EstDuration
```

(2) ListProjectsByIDs

存储过程 ListProjectsByIDs 是根据用户权限返回某个项目的数据统计。如果用户的角色是系统管理员，则可以查看该项目数据报表中的所有数据；如果用户角色是项目经理并且为该项目的项目经理，则可以查看项目数据报表中的所有数据；如果用户角色是普通项目成员，则不可以参看该项目的数据。值得一提的是，该存储过程实现的方式是根据查询条件在存储过程中构建 SQL 语句然后执行，首先查询用户的角色信息，根据用户的角色信息构建不同的 SQL 语句。另外当用户的角色是普通项目成员的时候，在 SQL 语句的 WHERE 条件中使用 1=0 这样的表达式来控制返回结果为空。该 SQL 语句的代码清单如下。

```
CREATE PROCEDURE TT_ListProjectsByIDs
(
    @ProjectIDs nvarchar(512),
    @UserID int
)
AS
/* 首先获得用户的角色信息 */
DECLARE @sql nvarchar(1024),
        @RoleID int

SELECT @RoleID = RoleID
FROM TT_Users
WHERE UserID = @UserID;
/* 根据用户角色的不同构建不同的 SQL 语句 */
IF (@RoleID = 1)
BEGIN
```

```

SET @sql = 'SELECT P.ProjectID,
SET @sql = @sql + ' P.Name AS ProjectName,
SET @sql = @sql + ' P.EstCompletionDate,
SET @sql = @sql + ' P.EstDuration AS EstHours,
SET @sql = @sql + ' Sum(EL.Duration) AS ActualHours ,
SET @sql = @sql + ' FROM TT_Projects P INNER JOIN TT_EntryLog EL'
SET @sql = @sql + ' ON P.ProjectID = EL.ProjectID '
SET @sql = @sql + ' WHERE P.ProjectID IN (' + @ProjectIDs + ') '
SET @sql = @sql + ' GROUP BY P.ProjectID, P.Name, P.EstCompletionDate, P.EstDuration'
END
ELSE IF (@RoleID = 2)
BEGIN
SET @sql = 'SELECT P.ProjectID,
SET @sql = @sql + ' P.Name AS ProjectName,
SET @sql = @sql + ' P.EstCompletionDate,
SET @sql = @sql + ' P.EstDuration AS EstHours,
SET @sql = @sql + ' Sum(EL.Duration) AS ActualHours ,
SET @sql = @sql + ' FROM TT_Projects P INNER JOIN TT_EntryLog EL'
SET @sql = @sql + ' ON P.ProjectID = EL.ProjectID '
SET @sql = @sql + ' WHERE P.ProjectID IN (' + @ProjectIDs + ') '
SET @sql = @sql + ' AND P.ManagerUserID = ' + CAST(@UserID AS nvarchar(20))
SET @sql = @sql + ' GROUP BY P.ProjectID, P.Name, P.EstCompletionDate, P.EstDuration'
END
ELSE
BEGIN
SET @sql = 'SELECT P.ProjectID,
SET @sql = @sql + ' P.Name AS ProjectName,
SET @sql = @sql + ' P.EstCompletionDate,
SET @sql = @sql + ' P.EstDuration AS EstHours,
SET @sql = @sql + ' P.EstDuration AS ActualHours ,
SET @sql = @sql + ' FROM TT_Projects P WHERE 1=0 '
END
/* 执行 SQL 语句 */
EXEC sp_executesql @sql

```

(3) ListTimeEntriesByCategory

存储过程 ListTimeEntriesByCategory 是返回某个工作分类下所有详细工作条目的存储过程。该存储过程根据传入参数 CategoryID 查询该工作分类下所有工作条目的用户名、用户 ID、该项工作的开始日期、结束日期以及工作总量。该存储过程的代码清单如下。

```

CREATE PROCEDURE TT_ListTimeEntriesByCategory
(
    @CategoryID int
)
AS

SELECT
    U.UserName,
    U.UserID,
    temp.MinEntryDate,
    temp.MaxEntryDate,

```

```
        Sum(EL.Duration) AS Duration
    FROM TT_EntryLog EL
        INNER JOIN TT_Users U
            ON EL.UserID = U.UserID
        INNER JOIN (SELECT UserID, MIN(EntryDate) AS MinEntryDate, MAX(EntryDate) AS MaxEntryDate FROM
TT_EntryLog GROUP BY UserID) AS temp
            ON temp.UserID = EL.UserID
    WHERE EL.CategoryID = @CategoryID
        GROUP BY U.UserName, U.UserID, temp.MinEntryDate, temp.MaxEntryDate
```

(4) ListUserTimeSummary

存储过程 ListUserTimeSummary 是根据用户角色返回一段时间内某个用户列表中所有用户的工作量统计。该存储过程的参数中 UserIDList 为用户列表，其格式为所有用户 ID 以“,”为分隔符隔开的形式。根据参数中 ManagerUserID 所对应用户的角色返回相应的数据，如果用户角色为管理员，则可以查看到用户列表中所有用户的工作量统计数据；如果用户角色为项目经理，则返回该用户所担任项目经理的项目中用户列表中用户的工作量统计；如果用户角色是普通项目成员，则返回空记录集。和前面介绍的存储过程 ListProjectsByIDs 类似，存储过程 ListUserTimeSummary 也是根据查询条件构建 SQL 语句执行返回记录集的，不同的是存储过程 ListUserTimeSummary 所构建和执行的 SQL 语句是带有参数的。该存储过程的代码清单如下。

```
CREATE PROCEDURE TT_ListUserTimeSummary
(
    @ManagerUserID int,
    @UserIDList nvarchar(512),
    @StartDate datetime,
    @EndDate datetime
)
AS

DECLARE
    @sSqlString nvarchar(1024),
    @sSubSql nvarchar(1024),
    @RoleID int

/* 获得用户的角色信息 */
SELECT @RoleID = RoleID
FROM TT_Users
WHERE UserID = @ManagerUserID;
/* 根据用户的角色信息构建 SQL 语句 */
IF (@RoleID = 1)
    BEGIN
        SET @sSqlString = 'SELECT Sum(EL.Duration) TotalHours, U.UserID, U.UserName'
        SET @sSqlString = @sSqlString + ' FROM TT_EntryLog EL Inner Join TT_Users U On EL.UserID =
U.UserID WHERE U.UserID IN (' + @UserIDList + ')'
        SET @sSqlString = @sSqlString + ' and EL.EntryDate >= @1 and EL.EntryDate <= @2 GROUP BY
U.UserID, U.UserName'
    END
ELSE IF (@RoleID = 2)
    BEGIN
        SET @sSubSql = 'SELECT PM.UserID FROM TT_Projects P INNER JOIN TT_ProjectMembers PM'
        SET @sSubSql = @sSubSql + ' ON P.ProjectID = PM.ProjectID WHERE P.ManagerUserID = @3 AND
PM.UserID IN (' + @UserIDList + ')'
```

```

        SET @sSqlString = 'SELECT Sum(EL.Duration) TotalHours, U.UserID, U.UserName'
        SET @sSqlString = @sSqlString + ' FROM TT_EntryLog EL Inner Join TT_Users U On EL.UserID =
U.UserID WHERE U.UserID IN (' + @sSubSql + ')'
        SET @sSqlString = @sSqlString + ' AND EL.ProjectID IN (SELECT ProjectID From TT_Projects Where
ManagerUserID = @3)'
        SET @sSqlString = @sSqlString + ' and EL.EntryDate >= @1 and EL.EntryDate <= @2 GROUP BY
U.UserID, U.UserName'
        END
    ELSE
        SET @sSqlString = 'SELECT U.UserID AS TotalHours, U.UserID, U.UserName From TT_Users U Where 1=0'
    /* 执行带有参数列表的 SQL 语句 */
    EXEC sp_executesql @sSqlString, N'@1 datetime, @2 datetime, @3 int', @StartDate, @EndDate, @ManagerUserID

```

16.2 页面形式的报表

页面形式的报表分为项目数据统计和个人数据统计两个页面，在系统的页面中单击导航栏中的 Reports 标签进入页面数据报表选择页面，在该页面中可以选择查看项目数据报表或个人数据报表。

16.2.1 逻辑层

页面数据报表模块的逻辑层包括如下内容。

(1) ProjectReportCategory.cs

类 ProjectReportCategory 的主要功能是获得某个项目中所有项目工作分类统计数据信息。该类的主要方法 GetCategorySummary 根据参数表中的 projectID 调用数据库的存储过程 TT_ListCategoriesByProject 返回该项目中所有项目分类的统计数据。该类的代码清单如下。

```

using System;
using System.Configuration;
using System.Data;
using ASPNET.StarterKit.TimeTracker.DataAccessLayer;

```

该类被封装在命名空间 ASPNET.StarterKit.TimeTracker.BusinessLogicLayer 下。

```

namespace ASPNET.StarterKit.TimeTracker.BusinessLogicLayer
{
    public class ProjectReportCategory
    {
        //定义该类的私有成员变量
        private decimal _actualHours;
        private int      _categoryID;
        private string   _categoryName;
        private string   _categoryShortName;
        private decimal _estDuration;
        //类构造方法初始化私有成员变量
        public ProjectReportCategory()
        {
            _categoryID = 0;
            _categoryName = string.Empty;
        }
    }
}

```

```
_categoryShortName = string.Empty;
_estDuration = 0M;
_actualHours = 0M;
}
//定义构造器构造该类所实例化的对象的属性
public decimal ActualHours
{
    get { return _actualHours; }
    set { _actualHours = value; }
}

public int CategoryID
{
    get { return _categoryID; }
    set { _categoryID = value; }
}

public string CategoryName
{
    get { return _categoryName; }
    set { _categoryName = value; }
}

public string CategoryShortName
{
    get { return _categoryShortName; }
    set { _categoryShortName = value; }
}

public decimal EstDuration
{
    get { return _estDuration; }
    set { _estDuration = value; }
}
```

GetCategorySummary 为 ProjectReportCategory 类惟一的静态方法，该方法返回一个 ProjectReportCategoryCollection 类型的对象，ProjectReportCategoryCollection 为项目中工作条目对象的集合类，由于该集合类比较简单，所以本书不再专门进行讲解。

```
public static ProjectReportCategoryCollection GetCategorySummary(int projectID)
{
    //调用存储过程 TT_ListCategoriesByProject 返回 DataSet
    DataSet dsCats = SqlHelper.ExecuteDataset(ConfigurationSettings.AppSettings[Web.Global.CfgKeyConnString],
                                                "TT_ListCategoriesByProject", projectID);
    //实例化对象集合 ProjectReportCategoryCollection 用于存储方法返回值
    ProjectReportCategoryCollection categoryList = new ProjectReportCategoryCollection();

    //从 DataSet 中获得数据填充 ProjectReportCategoryCollection 对象
    foreach(DataRow row in dsCats.Tables[0].Rows)
    {
        ProjectReportCategory cat = new ProjectReportCategory();
```

```
        cat.CategoryID = Convert.ToInt32(row["CategoryID"]);
        cat.CategoryName = row["Name"].ToString();
        cat.CategoryShortName = row["CategoryShortName"].ToString();
        cat.EstDuration = Convert.ToDecimal(row["EstDuration"]);
        cat.ActualHours = Convert.ToDecimal(row["ActualHours"]);

        categoryList.Add(cat);
    }
    //返回 ProjectReportCategoryCollection 类型的对象
    return categoryList;
}
}
}
```

(2) ProjectReportEntryLog.cs

类 ProjectReportEntryLog 的主要功能是获得某个项目的工作分类下所有项目工作进程的统计数据信息。该类的主要方法 GetEntrySummary 根据参数表中的 categoryID 调用数据库的存储过程 TT_ListTimeEntriesByCategory 返回该项目中所有项目分类的统计数据。ProjectReportEntryLog 类的代码清单如下。

```
using System;
using System.Configuration;
using System.Data;
using ASPNET.StarterKit.TimeTracker.DataAccessLayer;
//该类被封装在命名空间 ASPNET.StarterKit.TimeTracker.BusinessLogicLayer 下
namespace ASPNET.StarterKit.TimeTracker.BusinessLogicLayer
{
    public class ProjectReportEntryLog
    {
        //定义类私有成员变量
        private decimal _duration;
        private string _fullName;
        private DateTime _maxEntryDate, _minEntryDate;
        private int _userID;
        private string _userName;
        //类构造方法中对私有成员变量进行初始化赋值
        public ProjectReportEntryLog()
        {
            _userName = string.Empty;
            _duration = 0M;
            _userID = 0;
            _minEntryDate = DateTime.MinValue;
            _maxEntryDate = DateTime.MaxValue;
            _fullName = string.Empty;
        }

        public decimal Duration
        {
            get { return _duration; }
            set { _duration = value; }
        }
    }
}
```

```
public string FullName
{
    get { return _fullName; }
    set { _fullName = value; }
}

public int UserID
{
    get { return _userID; }
    set { _userID = value; }
}

public string UserName
{
    get { return _userName; }
    set { _userName = value; }
}

public DateTime MaxEntryDate
{
    get { return _maxEntryDate; }
    set { _maxEntryDate = value; }
}

public DateTime MinEntryDate
{
    get { return _minEntryDate; }
    set { _minEntryDate = value; }
}
```

GetEntrySummary 方法是类 ProjectReportEntryLog 的主要方法，该方法根据参数表中的项目工作分类 ID 获得该工作分类下的所有工作进程数据，并将每一条工作进程数据存储在一个 ProjectReportEntryLog 对象中，最终返回 ProjectReportEntryLog 对象集合类 ProjectReportEntryLogCollection。

```
public static ProjectReportEntryLogCollection GetEntrySummary(int categoryID)
{
    //调用存储过程 TT_ListTimeEntriesByCategory 将返回数据存储在 DataSet 中
    DataSet dsData = SqlHelper.ExecuteDataset(ConfigurationSettings.AppSettings[Web.Global.CfgKey
ConnectionString],
    "TT_ListTimeEntriesByCategory", categoryID);
    ProjectReportEntryLogCollection entryList = new ProjectReportEntryLogCollection();

    //从 DataSet 中获得数据并将 DataSet 中的数据存储在 ProjectReportEntryLog 对象中
    foreach(DataRow row in dsData.Tables[0].Rows)
    {
        ProjectReportEntryLog entry = new ProjectReportEntryLog();
        string firstName = string.Empty;
        string lastName = string.Empty;

        entry.UserName = row["UserName"].ToString();
        entry.Duration = Convert.ToDecimal(row["Duration"]);
```

```

        entry.UserID = Convert.ToInt32(row["UserID"]);
        entry.MinEntryDate = Convert.ToDateTime(row["MinEntryDate"]);
        entry.MaxEntryDate = Convert.ToDateTime(row["MaxEntryDate"]);
        entry.FullName = TTUser.GetDisplayName(entry.UserName, ref firstName, ref lastName);
        //将 ProjectReportEntryLog 对象添加到对象集合类中
        entryList.Add(entry);
    }
    //返回该对象集合类
    return entryList;
}
}
}

```

(3) ProjectReportProject.cs

类 ProjectReportProject 的主要功能是获得某个用户的在指定项目中的工作量统计信息。该类在设计结构上与 ProjectReportCategory、ProjectReportEntryLog 几乎一样。最主要的方法为 GetProjectSummary 方法，该方法返回参数表中的 userID 所对应的用户在项目列表 projectIdList 中的工作量统计对象集合类，projectIdList 为一个或多个 projectId 以“,” 隔开的字符串。用户的项目工作量统计对象包含项目 Id、项目名称、项目结束日期、项目总体工作量以及该用户在项目中的工作量几个属性。GetProjectSummary 的代码如下。

```

public static ProjectReportProjectCollection GetProjectSummary(string projectIdList, int userID)
{
    //调用存储过程获得数据并将数据保存到 DataSet 中
    DataSet dsProjects =
        SqlHelper.ExecuteDataset(ConfigurationSettings.AppSettings[Web.Global.CfgKey ConnString],
                               "ListProjectsByIDs", projectIdList, userID);
    ProjectReportProjectCollection projectList = new ProjectReportProjectCollection();

    foreach (DataRow row in dsProjects.Tables[0].Rows)
    {
        //从 DataSet 中获得数据并将数据保存到 ProjectReportProject 对象中
        ProjectReportProject proj = new ProjectReportProject();
        proj.ProjectID = Convert.ToInt32(row["ProjectID"]);
        proj.ProjectName = row["ProjectName"].ToString();
        proj.EstCompletionDate = Convert.ToDateTime(row["EstCompletionDate"]);
        proj.EstHours = Convert.ToDecimal(row["EstHours"]);
        proj.ActualHours = Convert.ToDecimal(row["ActualHours"]);
        //将 ProjectReportProject 对象添加到对象集合类中
        projectList.Add(proj);
    }
    //返回对象集合类
    return projectList;
}

```

(4) ResourceReportUser.cs

类 ResourceReportUser 的主要功能是获得某个项目经理下属项目中项目人员的工作量统计对象集合类。该类在设计结构上与以上介绍过的 3 个类几乎一样。最主要的方法为 GetUserSummary 方法，该方法返回参数表中的 mgrUserID 对应的用户作为项目经理所负责的项目中人员列表 userIdList 中所有人员的工作量统计对象集合类，userIdList 为一个或多个 userId 以“,” 隔开的字符串。项目开发人员的工

作量统计对象包含用户ID、用户名、用户总工作量统计以及用户全称几个属性。 GetUserSummary 的代码如下。

```
public static ResourceReportUserCollection GetUserSummary(int mgrUserID, string userIdList, DateTime startDate, DateTime endDate)
{
    //调用 TT_ListUserTimeSummary 获得数据并将数据存储在 DataSet 中
    DataSet dsUsers = SqlHelper.ExecuteDataset(ConfigurationSettings.AppSettings[Web.Global.CfgKey
ConnString],
    "TT_ListUserTimeSummary", mgrUserID, userIdList, startDate, endDate);
    ResourceReportUserCollection userList = new ResourceReportUserCollection();

    string firstName = string.Empty;
    string lastName = string.Empty;

    foreach (DataRow row in dsUsers.Tables[0].Rows)
    {
        //从 DataSet 中获得数据并保存到 ResourceReportUser 对象中
        ResourceReportUser usr = new ResourceReportUser();
        usr.UserID = Convert.ToInt32(row["UserID"]);
        usr.UserName = row["UserName"].ToString();
        usr.TotalHours = Convert.ToDecimal(row["TotalHours"]);
        usr.FullName = TTUser.GetDisplayName(usr.UserName, ref firstName, ref lastName);
        //将 ResourceReportUser 对象存储在对象集合类中
        userList.Add(usr);
    }
    //返回该对象集合类
    return userList;
}
```

16.2.2 应用层

(1) Reports.aspx.cs

Reports.aspx.cs 是页面 Reports.aspx 的后台编码类，Reports.aspx 的主要功能是供用户选择所要查看的报表种类和所要查看的报表资源。报表资源指的是报表中数据的主体来源，例如项目工作量报表的报表资源是具体的某个项目，人员工作量报表的报表资源是具体的某个项目开发人员。

Reports.aspx.cs 中定义了页面载入方法 Page_Load，该方法定义了页面中项目选择和人员选择列表框控件的数据源。

```
private void Page_Load(object sender, System.EventArgs e)
{
    //判断当前用户是否有权限访问报表模块
    if (TTSecurity.IsInRole(TTUser.UserRoleAdminPMgr) == false)
    {
        Response.Redirect("AccessDenied.aspx?Index=-1", true);
    }
    //获取统计报表的开始日期和结束日期的默认值
    DateTime startingDate = DateTime.Today;
    DateTime endingDate = DateTime.Today;
```

```

    _pageIndex = Request.QueryString["Index"];
    //当页面首次载入未被提交时绑定选择人员和选择项目列表框的控件值
    if (!this.IsPostBack)
    {
        //指定项目列表控件 ProjectList 的数据源并绑定该控件
        ProjectList.DataSource = Project.GetProjects(TTSecurity.GetUserID(),
TTSecurity.GetUserRole());
        ProjectList.DataTextField = "Name";
        ProjectList.DataValueField = "ProjectID";
        ProjectList.DataBind();
        //指定人员列表控件 UserList 的数据源并绑定该控件
        UserList.DataSource = TTUser.GetUsers(TTSecurity.GetUserID(), TTSecurity.GetUserRole());
        UserList.DataTextField = "Name";
        UserList.DataValueField = "UserID";
        UserList.DataBind();
        //调用 FillCorrectStartEndDates 获取当前日期所在周的开始和结束日期
        BusinessLogicLayer.TimeEntry.FillCorrectStartEndDates(DateTime.Today, ref startDate,
ref endDate);
        StartDate.Text = startDate.ToShortDateString();
        EndDate.Text = endDate.ToShortDateString();
    }
}

```

BuildValueList 方法是获取列表控件中所有被选中的值，返回 string 变量中被选定的值以“，”隔开。

```

private string BuildValueList(ListItemCollection items)
{
    //建立一个 StringBuilder 存储被选项值
    StringBuilder idList = new StringBuilder();
    //遍历控件的所有项获取被选定项的值并以, 隔开拼接成一个字符串
    foreach (ListItem item in items)
    {
        if (item.Selected)
        {
            if (idList.ToString() != string.Empty)
                idList.Append(",");
            idList.Append(item.Value.ToString());
        }
    }
    //返回拼接后的字符串
    return idList.ToString();
}

```

GenProjectRpt_Click 方法是查看项目报表 GenProjectRpt 按钮单击事件所触发的方法，该方法在判断 ProjectList 控件的选定值不为空之后，调用 Server.Transfer 方法将页面定向到项目报表查看页面。

```

private void GenProjectRpt_Click(object sender, System.EventArgs e)
{
    //调用验证控件对 ProjectList 进行验证
    ProjectListRequiredFieldValidator.Validate();

    //如果通过验证，则调用 Server.Transfer 方法将页面定向到项目报表查看页面 ProjectReport.aspx
    if (ProjectListRequiredFieldValidator.IsValid)

```

```

        Server.Transfer("ProjectReport.aspx?IDs=" + BuildValueList(ProjectList.Items) +
            "&index=" + _pageIndex);
    }

```

GenResourceRpt_Click 方法是查看人员报表 GenResourceRpt 按钮单击事件所触发的方法，该方法在判断 UserList 控件的选定值不为空之后，调用 Server.Transfer 方法将页面定向到人员报表查看页面。

```

private void GenResourceRpt_Click(object sender, System.EventArgs e)
{
    //调用验证控件对 UserList 进行验证
    UserListRequiredFieldValidator.Validate();

    //如果通过验证，则调用 Server.Transfer 方法将页面定向到人员报表查看页面 ResourceReport.aspx
    if (UserListRequiredFieldValidator.IsValid)
        Response.Redirect("ResourceReport.aspx?IDs=" + BuildValueList(UserList.Items) +
            "&Start=" + StartDate.Text + "&End=" + EndDate.Text + "&index=" + _pageIndex, false);
}

```

(2) ProjectReport.aspx.cs

ProjectReport.aspx.cs 是项目报表页面 ProjectReport.aspx 的后台编码类，项目报表页面 ProjectReport.aspx 的主要功能是呈现项目统计数据报表，其中包括项目的总体信息、项目的工作分类信息、每一个项目工作分类下所有工作的信息。该页面中最主要的方法 BindProject 就是将当前查看项目数据绑定到页面的 DataList 上。

```

private void BindProject(string ids)
{
    //调用 ProjectReportProject 对象的 GetProjectSummary 获得并绑定数据源
    ProjectReportProjectCollection prjData = ProjectReportProject.GetProjectSummary(ids, TTSecurity.
GetUserID());
    ProjectList.DataSource = prjData;
    ProjectList.DataBind();

    //如果数据源中没有数据，则显示没有数据的提示信息，隐藏 DataList 控件
    if (prjData.Count == 0)
    {
        NoData.Visible = true;
        ProjectList.Visible = false;
    }
}

```

ListCategory 方法根据参数表中的 projectID 获得某个项目的所有工作分类的统计数据。

```

protected ProjectReportCategoryCollection ListCategory(int projectID)
{
    //调用 ProjectReportCategory 对象的 GetCategorySummary 方法获得数据
    ProjectReportCategoryCollection listCategory = ProjectReportCategory.GetCategorySummary
(projectID);
    return listCategory.Count == 0 ? null : listCategory;
}

```

ListTimeEntries 方法根据参数表中的 categoryID 获得某个项目工作分类下的所有工作进程的统计数据。

```

protected ProjectReportEntryLogCollection ListTimeEntries(int categoryID)
{
    //调用 ProjectReportEntryLog 对象的 GetEntrySummary 方法获得数据
}

```

```

    ProjectReportEntryLogCollection entryLog = ProjectReportEntryLog.GetEntrySummary(categoryID);
    return entryLog.Count == 0? null : entryLog;
}

```

(3) ResourceReport.aspx.cs

ResourceReport.aspx.cs 是人员报表页面 ResourceReport.aspx 的后台编码类，人员报表页面 ResourceReport.aspx 的主要功能是呈现人员统计数据报表，其中包括人员的总体统计数据以及该项目开发人员的所有工作进程统计数据。该页面中最主要的方法 BindList 就是将当前查看人员统计数据绑定到页面的 DataList 上。

```

private void BindList(string userIDs, DateTime start, DateTime end)
{
    //调用 ResourceReportUser 对象的 GetUserSummary 方法获得数据并绑定数据
    ResourceReportUserCollection userData = ResourceReportUser.GetUserSummary(TTSecurity.GetUserID(),
userIDs, start, end);
    UserList.DataSource = userData;
    UserList.DataBind();

    //如果数据源中没有数据，则显示没有数据的提示信息，隐藏 DataList 控件
    if (userData.Count == 0)
    {
        UserList.Visible = false;
        NoData.Visible = true;
    }
}

```

SortGridData 方法是对该方法参数表中的工作进程对象集合类进行排序的方法。

```

private void SortGridData(TimeEntriesCollection list, string sortField, bool asc)
{
    //首先实例化一个对象集合类中的字段 sortCol
    TimeEntriesCollection.TimeEntryFields sortCol = 
TimeEntriesCollection.TimeEntryFields.InitValue;
    //判断参数表中的排序字段
    switch(sortField)
    {
        case "EntryDate":
            sortCol = TimeEntriesCollection.TimeEntryFields.Day;
            break;
        case "ProjectName":
            sortCol = TimeEntriesCollection.TimeEntryFields.Project;
            break;
        case "CategoryName":
            sortCol = TimeEntriesCollection.TimeEntryFields.Category;
            break;
        case "Duration":
            sortCol = TimeEntriesCollection.TimeEntryFields.Hours;
            break;
        case "Description":
            sortCol = TimeEntriesCollection.TimeEntryFields.Description;
            break;
    }
    //对对象集合类进行排序
}

```

```

        list.Sort(sortCol, asc);
    }
}

```

ListTimeEntry 方法返回某个用户在一段时间内所有工作进程的统计数据。

```

protected TimeEntriesCollection ListTimeEntry(int mgrID, int userID, DateTime start, DateTime end)
{
    //调用 TimeEntry 对象的 GetEntries 方法
    TimeEntriesCollection entryList = BusinessLogicLayer.TimeEntry.GetEntries(mgrID, userID, start,
end);

    //调用 SortGridData 方法对对象集合类进行排序
    if (entryList.Count > 0)
        SortGridData(entryList, SortField, SortAscending);
    else
        entryList = null;
    //返回经过排序的对象集合类
    return entryList;
}

```

16.2.3 表示层

在统计报表模块的前台表层中，项目统计报表和人员统计报表页面为了分层次显示数据用到了 DataList 控件的迭代，即在 DataList 控件内部定义一个或多个不同数据源的 DataList 或 DataGrid，内部的 DataList 或 DataGrid 的数据源和外部的 DataList 的数据相关联。项目报表页面和人员统计报表页面的多层次数据结构以及控件的结构如图 16-4 所示。



图 16-4 项目报表页面和人员统计报表页面控件结构

项目报表页面中显示项目总体统计数据的 DataList 代码清单如下。

```

<!-- 定义项目总体统计数据 ProjectList -->
<asp:datalist id="ProjectList" RepeatColumns="1" RepeatDirection="Vertical" runat="server">
    <headerstyle cssclass="header-gray" />
    <!-- 定义 ProjectList 的 headerstyle -->
    <headertemplate>
        Project Report
    </headertemplate>
    <!-- 定义 ProjectList 的 itemtemplate -->
    <itemtemplate>
        <table border="0" cellpadding="0" cellspacing="0" class="Content" width="100%">

```

```
<tr>
    <td valign="top"> </td>
</tr>
<tr>
    <td>
        <table border="0" cellpadding="0" cellspacing="0" class="Content" width="100%">
            <tr>
                <td width="180" class="report-main-header">Project Name</td>
                <td width="70" align="right" class="report-main-header">Est. Hours</td>
                <td width="100" align="right" class="report-main-header">Actual Hours</td>
                <td width="150" class="report-main-header">Est. Completion</td>
            </tr>
            <tr>
                <!-- 显示项目名称 -->
                <td class="report-text"><%# DataBinder.Eval(Container.DataItem, "ProjectName")%></td>
                <!-- 显示项目总体工作量-->
                <td class="report-text" align="right"><%# DataBinder.Eval(Container.DataItem, "EstHours") %></td>
                <!-- 显示项目已完成工作量 -->
                <td class="report-text" align="right"><%# DataBinder.Eval(Container.DataItem, "ActualHours") %></td>
                <!-- 显示项目结束日期 -->
                <td class="report-text"><%# DataBinder.Eval(Container.DataItem, "EstCompletionDate", " {0:d} ) %></td>
            </tr>
        </table>
    </td>
</tr>
<tr>
    <td valign="top"> </td>
</tr>
<tr>
    <td>
        <!-- 定义显示工作分类数据的 CategoryList，其数据源调用 ListCategory 方法，参数为 ProjectList 中的 ProjectID-->
        <asp:datalist id="CategoryList" RepeatColumns="1" RepeatDirection="Vertical" DataSource='<%# ListCategory((int) DataBinder.Eval(Container.DataItem, "ProjectID")) %>' runat="server">
            <!-- 定义 CategoryList 的 itemtemplate -->
            <itemtemplate>
                <table border="0" cellpadding="0" cellspacing="0" class="Content" width="100%">
                    <tr>
                        <td height="12" width="105"></td>
                        <td valign="top" width="72" class="report-header">Category</td>
                        <td valign="top" width="70" align="right" class="report-header">Est. Hours</td>
                        <td valign="top" width="100" align="right" class="report-header">Actual Hours</td>
                    </tr>
                    <tr>
                        <td height="12" width="105"></td>
```

```
width="1"></td>
        <!-- 显示该工作分类名称-->
        <td valign="top" class="report-text"><%# DataBinder.Eval(Container.DataItem,
"CategoryShortName") %></td>
        <!-- 显示该工作分类总体工作量 -->
        <td valign="top" class="report-text" align="right"><%# DataBinder.Eval
(Container.DataItem, "EstDuration") %></td>
        <!-- 显示该工作分类完成工作量 -->
        <td valign="top" class="report-text" align="right"><%# DataBinder.Eval
(Container.DataItem, "ActualHours") %></td>
    </tr>
    <tr>
        <td valign="top" colspan="4"></td>
    </tr>
    <tr>
        <td valign="top" colspan="2"> </td>
        <td colspan="2">
            <!-- 定义显示该工作分类下所有工作进程的DataList 调用 ListTimeEntries
获得数据源 -->
<asp:datalist id="EntryList" Width="100%" RepeatColumns="1"
RepeatDirection="Vertical" DataSource='<%# ListTimeEntries((int) DataBinder.Eval(Container.DataItem, "CategoryID"))
%' runat="server">
            <!-- 定义 EntryList 的 headertemplate -->
            <headertemplate>
                <table border="0" cellpadding="0" cellspacing="0" class=
"Content" width="100%">
                    <tr>
                        <td valign="top" align="left" class="report-header">
                            Consultant</td>
                        <td valign="top" align="right" class="report-header">
                            Hours</td>
                    </tr>
                </headertemplate>
            <!-- 定义 EntryList 的 itemtemplate 模板 -->
            <itemtemplate>
                <tr>
                    <td valign="top" class="report-text">
                        <!-- 显示工作进程的执行人并定义链接到
该工作人员的人员统计报表-->
<a href='ResourceReport.aspx?index=<%#
_pageIndex %>&IDs=<%#
DataBinder.Eval(Container.DataItem, "UserID") %>&Start=<%#((DateTime)DataBinder.Eval(Container.DataItem,
"MinEntryDate")).ToShortDateString()%>&End=<%#((DateTime)DataBinder.Eval(Container.DataItem,
"MaxEntryDate")).ToShortDateString()%>'>
                            <%# DataBinder.Eval(Container.DataItem,
"FullName") %>
                        </a>
                    </td>
                </tr>
            </itemtemplate>
        </td>
    </tr>
</asp:datalist>

```

```
<!-- 显示工作进程的工作量 -->
<td valign="top" class="report-text" align="right"><%# DataBinder.Eval(Container.DataItem, "Duration") %></td>
</tr>
</itemtemplate>
<!-- 定义 EntryList 的 footer template -->
<footertemplate>
</table>
</FooterTemplate>
<!-- EntryList 结束 -->
</asp:datalist>
</td>
</tr>
<tr>
<td height="25" valign="top" colspan="4"></td>
</tr>
</table>
<!-- CategoryList 结束 -->
</itemtemplate>
</asp:datalist>
</td>
</tr>
</table>
<!-- ProjectList 结束-->
</itemtemplate>
</asp:datalist>
```

人员统计报表页面中显示人员统计数据的 DataList 代码清单如下。

```
<!-- 定义显示用户统计数据的 DataList -->
<asp:datalist id="UserList" runat="server" Width="100%">
<!-- 定义 UserList 的 header template -->
<headertemplate>
<table cellSpacing="0" cellPadding="0" width="100%" border="0">
<tr>
<td colspan="3" class="header-gray">Resource Report</td>
</tr>
<tr>
<td colspan="3" > </td>
</tr>
<tr>
<td width="150" class="report-header">Beginning Date</td>
<td width="150" class="report-header">Ending Date</td>
<td width="*" class="report-header"> </td>
</tr>
<tr>
<td class="report-text">
<!-- 显示统计的开始日期 -->
<asp:Label ID="StartDate" Runat="server" Text='<%# _startDate.ToString("d") %>' />
</td>
<td class="report-text">
```

```
<!-- 显示统计的结束日期 -->
<asp:Label ID="EndDate" Runat="server" Text='<%# _endDate.ToString("d") %>' />
</td><td></td>
</tr>
<tr><td colspan="3"> </td></tr>
</table>
</headertemplate>
<!--定义UserList的itemtemplate-->
<itemtemplate>
    <table cellspacing="0" cellpadding="0" width="100%" border="0">
        <tr>
            <td width="150" class="report-main-header">Consultant</td>
            <td width="78" align="right" class="report-main-header">Total Hours</td>
            <td width="*" class="report-main-header"> </td>
        </tr>
        <tr>
            <!-- 显示项目开发人员的全名 -->
            <td class="report-text"><%# DataBinder.Eval(Container.DataItem, "FullName") %>
        </td>
            <!-- 显示项目开发人员的总体工作量 -->
            <td class="report-text" align="right"><%# DataBinder.Eval(Container.DataItem, "TotalHours") %></td>
        </tr>
        <tr><td height="24" colspan="3"> </td>
        </tr>
    </table>
    <table cellspacing="0" cellpadding="3" width="100%" border="0">
        <tr>
            <td>
                <!-- 定义显示项目开发人员所有工作进程的DataGrid -->
                <!-- 调用ListTimeEntry方法获得DataGrid的数据源 -->
                <asp:datagrid id="TimeEntryGrid" DataSource='<%# ListTimeEntry(TTSecurity.GetUserID(), (int)DataBinder.Eval(Container.DataItem, "UserID"), _startDate, _endDate) %>' runat="server"
                    BorderStyle="None" Width="100%" CellPadding="4" AutoGenerateColumns="False" Font-Name="Verdana"
                    AllowSorting="True" BorderColor="White" OnSortCommand="TimeEntryGrid_Sort">
                    <headerstyle Font-Bold="True" CssClass="report-header"></headerstyle>
                    <columns>
                        <!-- 定义列显示工作进程执行日期 -->
                        <asp:templatecolumn SortExpression="EntryDate" HeaderText="Date">
                            <headerstyle HorizontalAlign="Left" Width="25px" CssClass="report-header" VerticalAlign="Middle"></headerstyle>
                            <itemstyle HorizontalAlign="Left" Width="25px" CssClass="report-text"></itemstyle>
                        <itemtemplate>
                            <%# DataBinder.Eval(Container, "DataItem.EntryDate", "{0:d}") %>
                        </itemtemplate>
                    </asp:templatecolumn>
                </asp:datagrid>
            </td>
        </tr>
    </table>
</td>
```

```
<!-- 定义列显示项目名称 -->
<asp:templatecolumn SortExpression="ProjectName" HeaderText=
"Project">
    <headerstyle HorizontalAlign="Left" Width="120px" CssClass=
"report-header" VerticalAlign="Middle"></headerstyle>
    <itemstyle CssClass="report-text" Width="120px" Wrap=
"False"></itemstyle>
    <itemtemplate>
        <%# DataBinder.Eval(Container, "DataItem.ProjectName") %>
    </itemtemplate>
</asp:templatecolumn>
<!-- 定义列显示工作分类名称 -->
<asp:templatecolumn SortExpression="CategoryName" HeaderText=
"Cat.">
    <headerstyle HorizontalAlign="Left" Width="20px" CssClass=
"report-header" VerticalAlign="Middle"></headerstyle>
    <itemstyle CssClass="report-text" Width="20px"></itemstyle>
    <itemtemplate>
        <%# DataBinder.Eval(Container.DataItem,
"CategoryShortName") %>
    </itemtemplate>
</asp:templatecolumn>
<!-- 定义列显示工作进程工作量 -->
<asp:templatecolumn SortExpression="Duration" HeaderText="Hrs.">
    <headerstyle HorizontalAlign="Right" Width="40px" CssClass=
"report-header" VerticalAlign="Middle"></headerstyle>
    <itemstyle HorizontalAlign="Right" CssClass="report-text"
Width="40px"></itemstyle>
    <itemtemplate>
        <%# DataBinder.Eval(Container.DataItem, "Duration",
" {0:f} ") %>
    </itemtemplate>
</asp:templatecolumn>
<!-- 定义列显示工作进程的描述信息 -->
<asp:templatecolumn SortExpression="Description" HeaderText=
>Description">
    <headerstyle HorizontalAlign="Left" CssClass="report-header"
VerticalAlign="Middle"></headerstyle>
    <itemstyle CssClass="report-text" Wrap="False"></itemstyle>
    <itemtemplate>
        <%# DataBinder.Eval(Container, "DataItem.Description") %>
    </itemtemplate>
</asp:templatecolumn>
</columns>
<!-- DataGrid 结束 -->
</asp:datagrid>
</td>
</tr>
<tr>
```

```
<td height="48"> </td>
</tr>
</table>
<!-- DataList 结束 --&gt;
&lt;/itemtemplate&gt;
&lt;/asp:dataslist&gt;</pre>
```

16.3 图片形式的报表

ASP.NET Time Tracker Starter Kit 中最为复杂的一部分就是图片形式报表，实际上这部分表现内容在逻辑上并不复杂，仅仅是在工作进程管理页面上的图片形式的报表，但是由于涉及使用 GDI+技术动态生成图片才显得这部分在技术实现方面尤为复杂。也正是由于这种原因，本节中将先讨论应用层，通过应用层对逻辑层的调用逐步深入讲解逻辑层。

GDI+技术在本书的第 12 章中已经进行了初步的介绍，在本节中将对 ASP.NET Time Tracker Starter Kit 使用 GDI+技术绘制动态图片进行进一步的分析和研究。

16.3.1 应用层

在本书的第 15 章关于工作进程管理页面的讲解中可以看到，实际上图片形式的报表是通过对 TimeEntryBarChart.aspx 页面的访问来实现的，TimeEntryBarChart.aspx 根据请求 URL 中的参数返回图片格式的字节流，发送到客户端形成图片形式的报表。在 TimeEntryBarChart.aspx 页面的后台编码类 TimeEntryBarChart.aspx.cs 中，通过对 URL 参数的解析调用逻辑层中的方法，形成图片形式的字节流。TimeEntryBarChart.aspx.cs 的页面载入方法如下。

```
private void Page_Load(object sender, System.EventArgs e)
{
    //设置返回到客户端的响应格式为 image/png 即 png 格式的图片
    Response.ContentType = "image/png";

    string xValues, yValues;

    //从 URL 中获得参数
    xValues = Request.QueryString["xValues"];
    yValues = Request.QueryString["yValues"];

    if (xValues != null && yValues != null)
    {
        //实例化 Bitmap 对象
        Bitmap bmp;
        //实例化一个 MemoryStream 内存流对象
        MemoryStream memStream = new MemoryStream();
        //实例化一个 BarGraph 对象
        BarGraph bar = new BarGraph(Color.White);

        //调用 BarGraph 对象的 SetColor 设置输出颜色
        for (int i = 0; i < 7; i++) bar.SetColor(i, Color.Sienna);
    }
}
```

```

//设置 BarGraph 对象的属性值
bar.VerticalTickCount = 2;
bar.ShowLegend = false;
bar.ShowData = true;
bar.Height = 119;
bar.Width = 195;
bar.TopBuffer = 5;
bar.BottomBuffer = 15;
bar.FontColor = Color.Gray;
//调用 BarGraph 对象的 CollectDataPoints 方法
bar.CollectDataPoints(xValues.Split("|".ToCharArray()),
yValues.Split("|".ToCharArray()));
//调用 BarGraph 对象的 Draw 方法
bmp = bar.Draw();

//将 Bitmap 对象以内存流的形式发送到客户端
bmp.Save(memStream, ImageFormat.Png);
memStream.WriteTo(Response.OutputStream);
}
}

```

16.3.2 逻辑层

在应用层中产生图片格式的内存流主要使用了 BarGraph 对象的 CollectDataPoints 方法和 Draw 方法，在逻辑层的分析中首先从这两个方法入手。

CollectDataPoints 方法主要是形成所要绘制的图片上的点和线的集合，而 Draw 方法则是将 CollectDataPoints 方法中形成的点和线的集合绘制出来。下面是 CollectDataPoints 方法的代码清单，CollectDataPoints 的参数中为所要显示图片中日期与工作量的数组。

```

public void CollectDataPoints(string[] labels, string[] values)
{
    if (labels.Length == values.Length)
    {
        for (int i=0; i<labels.Length; i++)
        {
            //获取当前所要显示的日期与工作量
            float temp = Convert.ToSingle(values[i]);
            string shortLbl = MakeShortLabel(labels[i]);

            //将坐标原点加入点集合
            DataPoints.Add(new ChartItem(shortLbl, labels[i], temp, 0.0f, 0.0f, GetColor(i)));

            //在循环中求出最大工作量
            if (_maxValue < temp) _maxValue = temp;

            //求出横坐标中显示日期名称的字符串长度
            if (_displayLegend)
            {
                string currentLbl = labels[i] + " (" + shortLbl + ")";
                float currentWidth = CalculateImgFontWidth(currentLbl, _legendFontSize, FontFamily);
            }
        }
    }
}

```

```
        if(_maxLengthWidth < currentWidth)
        {
            _longestLabel = currentLb1;
            _maxLengthWidth = currentWidth;
        }
    }
}

//以下调用的4个方法为确定输出图片部分参数的方法
CalculateTickAndMax();
CalculateGraphDimension();
CalculateBarWidth(DataPoints.Count, _graphWidth);
CalculateSweepValues();
}
else
throw new Exception("X data count is different from Y data count");
}
```

Draw方法重载了BarGraph类的基类Chart中的Draw方法，实现了图片的输出。

```
public override Bitmap Draw()
{
    //获得输出图片的高度和宽度
    int height = Convert.ToInt32(_totalHeight);
    int width = Convert.ToInt32(_totalWidth);
    //实例化一个Bitmap图片对象
    Bitmap bmp = new Bitmap(width, height);
    //实例化一个作图对象Graphics
    using(Graphics graph = Graphics.FromImage(bmp))
    {
        //设置Graphics对象的部分属性
        graph.CompositingQuality = CompositingQuality.HighQuality;
        graph.SmoothingMode = SmoothingMode.AntiAlias;
        //绘制图片的背景
        graph.FillRectangle(new SolidBrush(_backColor), -1, -1, bmp.Width+1, bmp.Height+1);
        //调用以下4个方法分别绘制图片上的不同区域
        DrawVerticalLabelArea(graph);
        DrawBars(graph);
        DrawXLabelArea(graph);
        if (_displayLegend) DrawLegend(graph);
    }

    return bmp;
}
```

通过对CollectDataPoints方法和Draw方法的研究发现，实际上产生并绘制点和线集合的还有几个重要的方法。

```
// DrawBars方法为绘制工作量高度条的方法
private void DrawBars(Graphics graph)
{
    SolidBrush brsFont = null;
    Font valFont = null;
    StringFormat sfFormat = null;
```

```
try
{
    //实例化绘制工作量高度条需要的对象
    brsFont = new SolidBrush(_fontColor);
    valFont = new Font(_fontFamily, _labelFontSize);
    sfFormat = new StringFormat();
    sfFormat.Alignment = StringAlignment.Center;
    int i = 0;

    //从点集合中获取工作量高度条的数据并绘制
    foreach(ChartItem item in DataPoints)
    {
        //实例化一个填充图形的对象 SolidBrush 并使用该对象
        using(SolidBrush barBrush = new SolidBrush(item.ItemColor))
        {
            float itemY = _yOrigin + _graphHeight - item.SweepSize;

            //绘制工作量高度条
            graph.FillRectangle(barBrush, _xOrigin + item.StartPos, itemY, _barWidth,
item.SweepSize);

            //判断是否绘制工作量的数值
            if (_displayBarData)
            {
                //确定绘制工作量数值的横坐标
                float startX = _xOrigin + (i * (_barWidth + _spaceBtwBars));
                //确定绘制工作量数值的纵坐标
                float startY = itemY - 2f - valFont.Height;
                //实例化一个 RectangleF 对象存储绘制工作量数值位置的 4 个浮点型数字
                RectangleF recVal = new RectangleF(startX, startY, _barWidth + _spaceBtwBars, valFont.
Height);
                //绘制工作量数值
                graph.DrawString(item.Value.ToString("#,###.##"), valFont, brsFont, recVal, sfFormat);
            }
            i++;
        }
    }
    finally
    {
        //销毁绘制过程中所使用的部分对象
        if (brsFont != null) brsFont.Dispose();
        if (valFont != null) valFont.Dispose();
        if (sfFormat != null) sfFormat.Dispose();
    }
}
//用 DrawVerticalLabelArea 方法来绘制纵坐标的标签以及标记
private void DrawVerticalLabelArea(Graphics graph)
{
    //实例化绘制中所使用的部分对象
```

```
Font lblFont = null;
SolidBrush brs = null;
StringFormat lblFormat = null;
Pen pen = null;
StringFormat sfVLabel = null;

try
{
    //设置绘制中所使用的部分对象
    lblFont = new Font(_fontFamily, _labelFontSize);
    brs = new SolidBrush(_fontColor);
    lblFormat = new StringFormat();
    pen = new Pen(_fontColor);
    sfVLabel = new StringFormat();
    lblFormat.Alignment = StringAlignment.Near;

    //绘制纵坐标的最长单位
    RectangleF recVLabel = new RectangleF(0f, _yOrigin-2*_spacer-lblFont.Height, _xOrigin*2, lblFont.
Height);
    sfVLabel.Alignment = StringAlignment.Center;
    graph.DrawString(_yLabel, lblFont, brs, recVLabel, sfVLabel);

    //绘制所有工作量的单位标记
    for (int i=0; i<_yTickCount; i++)
    {
        //确定纵坐标刻度
        float currentY = _topBuffer + (i * _yTickValue/_scaleFactor);
        //确定纵坐标标注文字位置
        float labelY = currentY-lblFont.Height/2;
        //实例化一个 RectangleF 对象存储绘制纵坐标标注文字位置的四个浮点型数字
        RectangleF lblRec = new RectangleF(_spacer, labelY, _maxTickValueWidth, lblFont.Height);
        //计算纵坐标的长度
        float currentTick = _maxValue - i*_yTickValue;
        //绘制工作量的纵向刻度
        graph.DrawString(currentTick.ToString("#,###.##"), lblFont, brs, lblRec, lblFormat);
        graph.DrawLine(pen, _xOrigin, currentY, _xOrigin - 4.0f, currentY);
    }

    //绘制纵坐标
    graph.DrawLine(pen, _xOrigin, _yOrigin, _xOrigin, _yOrigin + _graphHeight);
}

finally
{
    //销毁绘制中所使用的部分对象
    if (lblFont != null) lblFont.Dispose();
    if (brs != null) brs.Dispose();
    if (lblFormat != null) lblFormat.Dispose();
    if (pen != null) pen.Dispose();
    if (sfVLabel != null) sfVLabel.Dispose();
}
```

```
}

//用 DrawXLabelArea 方法来绘制横坐标及横向文字标注
private void DrawXLabelArea(Graphics graph)
{
    //实例化绘制中所使用的部分对象
    Font lblFont = null;
    SolidBrush brs = null;
    StringFormat lblFormat = null;
    Pen pen = null;

    try
    {
        //设置绘制中所使用的部分对象
        lblFont = new Font(_fontFamily, _labelFontSize);
        brs = new SolidBrush(_fontColor);
        lblFormat = new StringFormat();
        pen = new Pen(_fontColor);

        lblFormat.Alignment = StringAlignment.Center;

        //绘制横坐标轴
        graph.DrawLine(pen, _xOrigin, _yOrigin + _graphHeight, _xOrigin + _graphWidth, _yOrigin + _graphHeight);

        float currentX;
        //设置横坐标轴说明文字的纵向位置
        float currentY = _yOrigin + _graphHeight + 2.0f;
        //设置横坐标轴说明文字的宽度
        float labelWidth = _barWidth + _spaceBtwBars;
        int i = 0;

        //绘制横坐标轴说明文字
        foreach(ChartItem item in DataPoints)
        {
            //设置横坐标轴说明文字的横向位置
            currentX = _xOrigin + (i * labelWidth);
            //实例化一个 RectangleF 对象存储绘制横坐标轴说明文字位置的四个浮点型数字
            RectangleF recLbl = new RectangleF(currentX, currentY, labelWidth, lblFont.Height);
            //判断横坐标轴说明文字的显示宽度模式
            string lblString = _displayLegend ? item.Label : item.Description;
            //绘制横坐标轴说明文字
            graph.DrawString(lblString, lblFont, brs, recLbl, lblFormat);
            i++;
        }
    }
    finally
    {
        //销毁绘制中所使用的部分对象
        if (lblFont != null) lblFont.Dispose();
        if (brs != null) brs.Dispose();
    }
}
```

```
        if (lblFormat != null) lblFormat.Dispose();
        if (pen != null) pen.Dispose();
    }
}

// CalculateGraphDimension 方法是获得所有工作量刻度的方法
private void CalculateGraphDimension()
{
    //调用 FindLongestTickValue 方法获得最长的工作量
    FindLongestTickValue();

    //计算出纵坐标轴的长度
    _longestTickValue += "0";
    _maxTickValueWidth = CalculateImgFontWidth(_longestTickValue, _labelFontSize, FontFamily);
    float leftOffset = _spacer + _maxTickValueWidth;
    float rtOffset = 0.0f;
    //根据是否显示图例计算图例的宽度和纵坐标位置
    if (_displayLegend)
    {
        _legendWidth = _spacer + _rectangleSize + _spacer + _maxLengthWidth + _spacer;
        rtOffset = _graphLegendSpacer + _legendWidth + _spacer;
    }
    else
        rtOffset = _spacer;
    //确定显示纵坐标刻度标注文字的高度、宽度及位置
    _graphHeight = _totalHeight - _topBuffer - _bottomBuffer;
    _graphWidth = _totalWidth - leftOffset - rtOffset;
    _x0Origin = leftOffset;
    _y0Origin = _topBuffer;

    _scaleFactor = _maxLength / _graphHeight;
}
//用 CalculateImgFontWidth 方法来确定参数表中文字的绘制宽度
private float CalculateImgFontWidth(string text, int size, string family)
{
    //实例化计算中所用到的对象
    Bitmap bmp = null;
    Graphics graph = null;
    Font font = null;

    try
    {
        //设置字体对象
        font = new Font(family, size);

        //确定文字绘制之后的宽度
        bmp = new Bitmap(1, 1, PixelFormat.Format32bppArgb);
        graph = Graphics.FromImage(bmp);
        SizeF oSize = graph.MeasureString(text, font);
        //返回该宽度
        return oSize.Width;
    }
}
```

```
        }
        finally
        {
            //销毁该方法在计算过程中所用到的部分对象
            if (graph != null) graph.Dispose();
            if (bmp != null) bmp.Dispose();
            if (font != null) font.Dispose();
        }
    }

    //用 CalculateTickAndMax 方法计算出最长的工作量表示条以及每一条工作量表示条的刻度
    private void CalculateTickAndMax()
    {
        float tempMax = 0.0f;

        //确定一个默认的最长工作量表示条长度
        _maxValue *= 1.1f;
        //计算最长工作量表示条长度
        if (_maxValue != 0.0f)
        {
            double exp = Convert.ToDouble(Math.Floor(Math.Log10(_maxValue)));
            tempMax = Convert.ToSingle(Math.Ceiling(_maxValue / Math.Pow(10, exp)) * Math.Pow(10, exp));
        }
        else
            tempMax = 1.0f;

        //确定最长工作量表示条长度之后确定工作量刻度的单位
        _yTickCount = tempMax / _yTickCount;
        double expTick = Convert.ToDouble(Math.Floor(Math.Log10(_yTickCount)));
        _yTickCount = Convert.ToSingle(Math.Ceiling(_yTickCount / Math.Pow(10, expTick)) * Math.Pow(10, expTick));

        //根据工作量刻度的单位重新计算最长的工作量表示条长度
        _maxValue = _yTickCount * _yTickCount;
    }
```

以上介绍了 BarGraph 类中比较重要的一部分方法，这部分方法基本上完成了绘制图片中最核心的工作，其他的方法本书中不详细列出，读者如果感兴趣可以自行查阅。

16.4 开发启示

本章中介绍了 ASP.NET Time Tracker Starter Kit 中的数据报表模块，通过本章的讲解希望读者不仅仅学习到了数据报表模块的一些技术实现方法，还要学习到在 Web 应用程序中开发数据报表模块的设计思想和设计方法。掌握数据报表模块展现数据的形式和方法，学会如何运用 .NET 技术框架中的方法在已有的应用程序中实现一个直观透彻、功能强大的数据报表模块。

第 17 章

移动 Web 应用

作为一套教科书似的 Web 应用程序，ASP.NET Time Tracker Starter Kit 提供了一个移动 Web 应用模块。在本书的第 12 章中已经介绍了移动 Web 应用的基础知识，本章中将详细讲述移动 Web 应用技术在 ASP.NET Time Tracker Starter Kit 中的应用。

17.1 系统设计

移动设备已经成为我们生活的一部分，例如掌上计算机、智能手机等等。这些新的移动设备都可以连接网络或者执行应用程序。专为移动设备开发的应用程序越来越多地被人们应用，其优点就是可以让使用移动设备的用户在任何时刻都能访问到应用程序。这样不仅改善了应用程序的兼容性，同时也大大扩大了应用程序的应用范围。

从理论上讲，ASP.NET Time Tracker Starter Kit 在普通 Web 应用程序中所提供的功能都可以在移动 Web 应用下提供，但在实际的情况下并没有这个必要，移动 Web 应用只需要提供部分系统中的核心功能即可。在 ASP.NET Time Tracker Starter Kit 中，移动 Web 应用主要提供以下功能。

(1) 用户登录

和普通 Web 应用一样，移动 Web 应用也需要对用户的身份进行识别和验证。因为系统中所有的操作都是基于用户身份和用户角色的，如果用户不能提供有效的身份信息，则无论是在普通 Web 应用还是移动 Web 应用下都不能登录系统。图 17-1 就是用户登录系统的页面。

(2) 按日期查看工作进程

移动 Web 应用可以按照日期查看工作进程，日期以星期为单位，用户可以选择查看某个星期内所有的工作进程。图 17-2 为选择所要查看星期的页面。

系统将以列表的形式展示在该周内所有存在工作进程的日期，单击日期查看该日期下的所有工作进程情况。图 17-3 为查看一个星期内所有存在工作进程发生的日期列表页面。

(3) 管理工作进程

用户可以新增、修改、删除个人的工作进程，其中工作进程的信息包括工作进程的工作分类、发生日期、工作量统计以及工作进程描述。图 17-4 是修改工作进程的页面。



图 17-1 用户登录系统页面



图 17-2 选择所要查看星期的页面

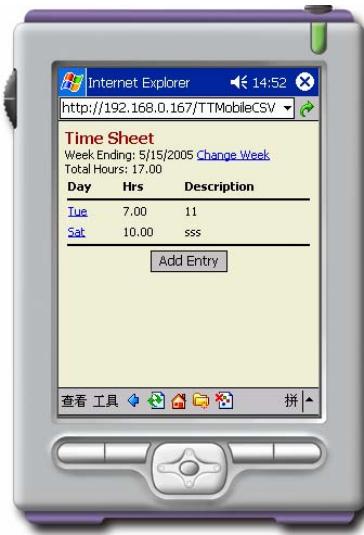


图 17-3 日期列表页面

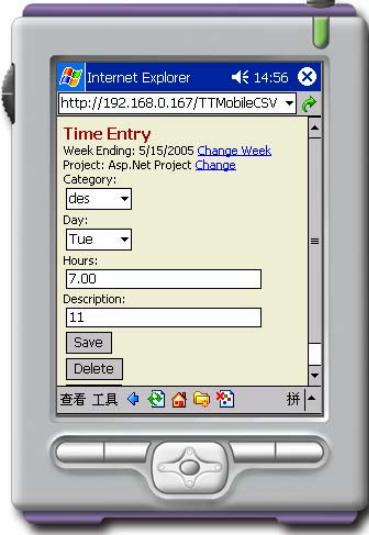


图 17-4 修改工作进程的页面

17.2 应用详解

在本书的第 11 章中已经介绍了 ASP.NET Time Tracker Starter Kit 的系统架构，其中重点提到移动 Web 应用和普通 Web 应用共用同一个逻辑层。因此，在本章中将重点介绍移动 Web 应用的应用层和表示层。

17.2.1 用户登录

SignIn.aspx 是用户登录页面，用户输入用户名、密码进行身份验证，如果用户名和密码匹配，则登

录成功转向选择日期页面；如果用户名和密码不匹配，则显示登录失败的信息。下面是 SignIn.aspx 的后台编码类 SignIn.aspx.cs 中最为重要的一个方法 Submit_Click，该方法是页面提交按钮的单击事件所触发的方法，该方法用于完成对用户的身份验证。

```
private void Submit_Click(object sender, System.EventArgs e)
{
    //实例化一个用户对象
    TTUser accountSystem = new TTUser();
    //调用用户对象的 Login 方法对用户进行身份验证并返回 userId
    string userId = accountSystem.Login(eMailText.Text, TTSecurity.Encrypt(passwordText.Text));

    //判断用户的 userId 是否为空
    if (userId != "" && userId != string.Empty)
    {
        //如果 userId 不为空，则表明用户通过身份验证
        FormsAuthentication.SetAuthCookie(eMailText.Text, false);
        //转向工作进程页面
        RedirectToMobilePage("TimeSheet.aspx");
    }
    else
        //如果 userId 为空，则说明身份验证失败
        signInMsg.Text = "Login Failed!";
}
```

在 SignIn.aspx 的前台页面中呈现的是 ASP.NET 的移动 Web 应用控件，其代码清单如下。

```
<%@ Register TagPrefix="mobile" Namespace="System.Web.UI.MobileControls" Assembly="System.Web.Mobile,
Version=1.0.3300.0, Culture=neutral, PublicKeyToken=b03f5f7f11d50a3a" %>
<%@ Page language="c#" Codebehind="SignIn.aspx.cs" Inherits="TTMobileCSV.SignIn" AutoEventWireup ="false"
%>
<meta name="CODE_LANGUAGE" content="C#">
<meta content="Microsoft Visual Studio .NET 7.0" name="GENERATOR">
<meta content="http://schemas.microsoft.com/Mobile/Page" name="vs_targetSchema">
<body Xmlns:mobile="http://schemas.microsoft.com/Mobile/WebForm">
    <!-- 使用 mobile:stylesheet 标签引入页面的样式表文件 -->
    <mobile:stylesheet id="Stylesheet1" ReferencePath="Styles.ascx" runat="server"></mobile:stylesheet>
    <!-- 使用 mobile:Form 表单标签定义表单 -->
    <mobile:Form id="FormSignIn" runat="server" StyleReference="Form">
        <!-- 使用 mobile:Label 标签定义 Label 控件 -->
        <mobile:Label id="LabelSignInTitle" StyleReference="title"
Runat="server">Sign-In</mobile:Label>
        <mobile:Label id="Label5" runat="server">e-mail address:</mobile:Label>
        <!-- 使用 mobile:TextBox 标签定义 TextBox 控件 -->
        <mobile:TextBox id="eMailText" runat="server"></mobile:TextBox>
        <!-- 使用 mobile:RequiredFieldValidator 标签定义验证控件 -->
        <mobile:RequiredFieldValidator id="RequiredFieldValidatorEmail" runat="server"
ControlToValidate ="eMailText" ErrorMessage="e-mail address is required"></mobile:RequiredFieldValidator>
        <mobile:RegularExpressionValidator id="RegularExpressionValidator1" runat="server"
ControlToValidate ="eMailText" ErrorMessage="e-mail is invalid"
ValidationExpression="\w+([-.]\w+)*@\w+([-.]\w+)*\.\w+ ([-.]\w+)*"></mobile:RegularExpressionValidator>
        <mobile:Label id="password" runat="server">password:</mobile:Label>
        <mobile:TextBox id="passwordText" runat="server" Password="True"></mobile:TextBox>
```

```
<mobile:RequiredFieldValidator id="RequiredfieldvalidatorPassword" runat="server">
    ControlToValidate = "passwordText" ErrorMessage="password is required"></mobile:RequiredFieldValidator>
    <!-- 使用 mobile:Command 标签定义按钮控件 -->
    <mobile:Command id="Submit" runat="server">Submit</mobile:Command>
    <mobile:Label id="signInMsg" runat="server"></mobile:Label>
    <!-- 结束表单定义 -->
</mobile:Form>
</body>
```

在用户身份读取和保存方面，移动 Web 应用使用的方法和普通 Web 应用使用的方法类似，都是对应用程序全局文件 Global.asax 中的 Application_AuthenticateRequest 事件进行操作，Application_AuthenticateRequest 事件的代码清单如下。

```
protected void Application_AuthenticateRequest(Object sender, EventArgs e)
{
    string userInformation = String.Empty;
    //判断用户是否通过身份验证
    if (Request.IsAuthenticated == true)
    {
        //如果用户通过身份验证则获得客户端 Cookie
        if ((Request.Cookies[MobileUserRoles] == null) || (Request.Cookies[MobileUserRoles].Value == ""))
        {
            //实例化一个用户对象获取用户身份信息
            TTUser user = new TTUser(User.Identity.Name);
            if (!user.Load())
            {
                //如果没有在数据库中发现用户身份验证信息则添加新用户
                TTUser newUser = new TTUser(0, Context.User.Identity.Name,
                    String.Empty, ConfigurationSettings.AppSettings[CfgKeyDefaultRole]);
                newUser.Save();
                user = newUser;
            }
        }

        //建立一个字符串存储用户的 ID、角色和名字
        userInformation = user.UserID + ";" + user.Role + ";" + user.Name;

        //实例化一个 FormsAuthenticationTicket 对象存储基于 Cookie 的身份验证信息
        FormsAuthenticationTicket ticket = new FormsAuthenticationTicket(
            1,
            User.Identity.Name,
            DateTime.Now,
            DateTime.Now.AddHours(1),
            false,
            userInformation
        );

        //加密身份验证信息
        String cookieStr = FormsAuthentication.Encrypt(ticket);

        //将存储有身份验证信息的 Cookie 发送到客户端
        Response.Cookies[MobileUserRoles].Value = cookieStr;
        Response.Cookies[MobileUserRoles].Path = "/";
    }
}
```

```
Response.Cookies[MobileUserRoles].Expires = DateTime.Now.AddMinutes(1);

        Context.User = new CustomPrincipal(User.Identity, user.UserID, user.Role, user.Name);
    }
    else
    {
        //如果在数据库中发现用户身份信息则从Cookie中读取用户信息并通过验证
        FormsAuthenticationTicket ticket = FormsAuthentication.Decrypt(Context.Request.Cookies[MobileUserRoles].Value);
        userInformation = ticketUserData;

        //将用户身份信息保存到Context.User中
        string[] info = userInformation.Split(new char[] { ';' });
        Context.User = new CustomPrincipal(
            User.Identity,
            Convert.ToInt32(info[0].ToString()),
            info[1].ToString(),
            info[2].ToString());
    }
}
```

17.2.2 管理工作进程

TimeSheet.aspx是一个含有4个表单的移动Web页面，在该页面中可以完成日期浏览、查看某一星期内的工作进程安排、修改和删除现有工作进程的功能。多个表单在同一页面内的这种设计模式是在普通ASP.NET页面中是没有的，在普通ASP.NET页面中数据的传递是通过表单的提交完成的，但是在移动ASP.NET页面中多个表单在同一页面内，表单之间的数据交换则可以使用状态保持的对象来完成。在每一次表单的跳转过程中，系统将需要传递和保存的数据暂时存放在状态保持对象中。以下是TimeSheet.aspx.cs的代码清单。

```
//声明状态保持对象_entryHolder
private TimeEntry _entryHolder = new TimeEntry();
private TTUser _user;
private DateTime _weekEndingDate = DateTime.MinValue;
private DateTime _weekStartingDate = DateTime.MinValue;
protected System.Web.UI.MobileControls.Command DeleteCommand;
private decimal _totalHours = decimal.Zero;
//定义表单的枚举
private enum FormName {
    FormMain, FormDetail
}
//页面载入方法
private void Page_Load(object sender, System.EventArgs e)
{
    //首先获得用户对象
    _user = new TTUser(
        TTSecurity.GetUserID(),
        User.Identity.Name,
        TTSecurity.GetName(),
```

```

    TTSecurity.GetUserRole();

    if (!IsPostBack)
    {
        //如果页面首次载入则将状态保持对象中的日期设置为当前日期
        _entryHolder.EntryDate = DateTime.Today;
        //将隐藏域 HiddenDate 的值也设置成当前日期
        HiddenDate.Text = DateTime.Today.ToString("yyyy-MM-dd");
    }
    else
    {
        //如果页面被提交则从隐藏域中获得值
        _entryHolder.EntryDate = Convert.ToDateTime(HiddenDate.Text);
        if (CategoryList.Selection != null) _entryHolder.CategoryID = Convert.ToInt32(CategoryList.Selection.Value);
        _entryHolder.Description = TTSecurity.CleanStringRegex(EntryDescriptionText.Text);

        //从隐藏域中获得值赋给状态保持对象的属性
        try
        {
            _entryHolder.ProjectID = Convert.ToInt32(HiddenProjectID.Text);
            _entryHolder.EntryLogID = Convert.ToInt32(HiddenLogID.Text);
            _entryHolder.ProjectName = TTSecurity.CleanStringRegex(HiddenProjectName.Text);
            _entryHolder.Duration = Convert.ToDecimal(HoursText.Text);
        }
        catch {}
    }
}

```

FormMain_Activate 方法是表单 FormMain 的激活方法，在 FormMain 表单中用户可以按周查看工作进程。

```

private void FormMain_Activate(object sender, System.EventArgs e)
{
    //调用 TimeEntry 对象的 FillCorrectStartEndDates 方法获得当前所查看周的起始日期和结束日期
    TimeEntry.FillCorrectStartEndDates(_entryHolder.EntryDate, ref _weekStartingDate, ref _weekEndingDate);
    TotalHoursLabel.Text = "Total Hours: 0";
    WeekEndingMain.Text = "Week Ending: " + _weekEndingDate.ToString("yyyy-MM-dd") + " ";
    //调用 BindTimeSheet 方法绑定列表
    BindTimeSheet(TimeEntryGrid, _user.UserID, _weekStartingDate, _weekEndingDate);
}

```

TimeEntryGrid_Select 方法是用户选择查看 objectlist 控件中某一条工作进程的详细信息事件所触发的方法，该方法获得用户所要查看的工作进程的 ID，并将状态保持对象实例化成用户所选择工作进程的对象，最终跳转表单到 FormDetail。

```

private void TimeEntryGrid_Select(object sender, System.Web.UI.MobileControls.ObjectListSelectEventArgs e)
{
    //获得用户所要查看的工作进程的 ID
    int currentEntryLogID = Convert.ToInt32(e.SelectedItem["EntryLogID"]);
    //将状态保持对象实例化成用户所选择工作进程的对象
    _entryHolder = new TimeEntry(currentEntryLogID);
}

```

```
_entryHolder.Load();
//跳转表单到 FormDetail
ActiveForm = FormDetail;
}
```

TimeEntryGrid_ItemBind 方法是显示工作进程列表的 objectlist 控件列表项绑定事件所触发的方法。在该方法中累加每项工作进程的工作量最终绑定显示总体工作量的 Label 控件。

```
private void TimeEntryGrid_ItemBind(object sender, System.Web.UI.MobileControls.ObjectListDataBindEventArgs e)
{
    //累加每项工作进程的工作量
    _totalHours += ((TimeEntry)e.DataItem).Duration;
    //绑定显示总体工作量的 Label 控件
    TotalHoursLabel.Text = "Total Hours: " + _totalHours.ToString();
}
```

ChangeWeekMain_Click 方法是更改当前查看周的按钮单击事件所触发的方法。

```
private void ChangeWeekMain_Click(object sender, System.EventArgs e)
{
    HiddenOrigin.Text = FormName.FormMain.ToString();
    ActiveForm = FormCalendar;
}
```

AddEntryCommand2_Click 方法是添加工作进程按钮的单击事件方法。

```
protected void AddEntryCommand2_Click(object sender, System.EventArgs e)
{
    ActiveForm = FormProject;
}
```

FormProject_Activate 方法是表单 FormProject 的激活方法，表单 FormProject 是用户选择添加或修改工作进程时选择项目的表单，该表单的 ListBox 控件 ProjectListBox 中列出当前用户有权限添加工作进程的所有项目。

```
private void FormProject_Activate(object sender, System.EventArgs e)
{
    //实例化一个项目对象集合类
    ProjectsCollection dsProject = BindProjectList(ProjectListBox, _user.UserID);
    //选择默认项目
    //如果是添加工作进程，则以列表中第一个项目为默认项目
    //如果是修改工作进程项目，则从 Session 中获得该工作进程原项目作为默认项目
    //如果用户没有权限向任何项目添加工作进程，则列表为空
    if (_entryHolder.ProjectID==0 && dsProject.Count>0)
        _entryHolder.ProjectID = ((Project)dsProject[0]).ProjectID;

    if (null != dsProject) ProjectListBox.SelectedIndex = FindProjectIndex(ProjectListBox,
    _entryHolder.ProjectID);
}
```

SelectProjectCommand_Click 是表单 FormProject 中选择项目按钮的单击事件所触发的方法，如果用户选择了项目，则将项目信息保存在状态保持对象和隐藏文本域中，供下一个表单 FormDetail 调用。

```
private void SelectProjectCommand_Click(object sender, System.EventArgs e)
{
    //如果用户没有选择项目，则跳转到 FormMain 表单
    if (null == ProjectListBox.Selection)
    {
        ActiveForm = FormMain;
    }
}
```

```

    }
    else
    {
        //将用户选择的项目信息保存到状态保持对象和隐藏文本域中
        HiddenProjectID.Text = ProjectListBox.Selection.Value;
        _entryHolder.ProjectID = Convert.ToInt32(ProjectListBox.Selection.Value);
        HiddenProjectName.Text = TTSecurity.CleanStringRegex(ProjectListBox.Selection.Text);
        _entryHolder.ProjectName = TTSecurity.CleanStringRegex(ProjectListBox.Selection.Text);
        //跳转当前表单到 FormDetail
        ActiveForm = FormDetail;
    }
}

```

FormDetail_Activate 方法是表单 FormDetail 的激活方法，表单 FormDetail 可以用于修改和添加工作进程。

```

private void FormDetail_Activate(object sender, System.EventArgs e)
{
    //调用 TimeEntry 对象的 FillCorrectStartEndDates 方法获得当前所在周的开始和结束日期
    TimeEntry.FillCorrectStartEndDates(_entryHolder.EntryDate, ref _weekStartingDate, ref
_weekending Date);
    //绑定项目分类列表
    BindCategoryList(CategoryList, _entryHolder.ProjectID, _entryHolder.CategoryID);

    if (_entryHolder.EntryLogID == 0)
    {
        //如果状态保持对象的 EntryLogID 属性为空，则说明当前表单是添加工作进程表单
        //以当前日期作为工作进程的默认日期并将删除工作进程按钮隐藏
        BindDayList(DayList, _entryHolder.EntryDate, DateTime.Today);
        DeleteCommand.Visible = false;
    }
    else
    {
        //如果状态保持对象的 EntryLogID 属性不为空，则说明当前表单是修改工作进程表单
        //获得当前工作进程的日期并显示删除工作进程按钮
        BindDayList(DayList, _entryHolder.EntryDate, _entryHolder.EntryDate);
        DeleteCommand.Visible = true;
    }
    //绑定显示当前所在周结束日期的控件
    WeekEndingDetail.Text = "Week Ending: " + _weekendingDate.ToShortDateString() + " ";
    //从状态保持对象中获得数据调用 PopulateFormDetail 方法绑定当前表单数据
    PopulateFormDetail(_entryHolder);
}

```

ChangeWeekDetail_Click 方法是更改日期按钮单击事件所触发的方法。

```

private void ChangeWeekDetail_Click(object sender, System.EventArgs e)
{
    //在打开选择日期控件之前记录当前表单位置
    HiddenOrigin.Text = FormName.FormDetail.ToString();
    ActiveForm = FormCalendar;
}

```

ChangeProjectCommand_Click 方法是更改工作进程所属项目按钮单击事件所触发的方法。

```

private void ChangeProjectCommand_Click(object sender, System.EventArgs e)

```

```
{  
    ActiveForm = FormProject;  
}
```

SaveCommand_Click 方法是保存当前工作进程按钮单击事件所触发的方法。该方法首先使用验证控件验证当前表单数据，实例化一个 TimeEntry 对象，调用其 Save 方法将数据更新到数据库，并将状态保持对象更改为保存过的工作进程对象，最终跳转表单到 FormMain。

```
private void SaveCommand_Click(object sender, System.EventArgs e)  
{  
    //调用验证控件验证表单的数据  
    RequiredFieldValidatorHrs.Validate();  
    RequiredFieldValidatorDesc.Validate();  
    CompareValidatorHrs.Validate();  
  
    if (RequiredFieldValidatorHrs.IsValid && RequiredFieldValidatorDesc.IsValid && CompareValidatorHrs.  
IsValid)  
    {  
        //更新状态保持对象的 EntryDate 属性  
        _entryHolder.EntryDate = Convert.ToDateTime(DayList.Selection.Value);  
        //实例化一个 TimeEntry 对象调用其 Save 方法将数据更新到数据库  
        TimeEntry time = new TimeEntry(_entryHolder.EntryLogID, _user.UserID, _entryHolder.ProjectID,  
_entryHolder.CategoryID, _entryHolder.EntryDate, _entryHolder.Description, _entryHolder.Duration);  
        time.Save();  
  
        //将状态保持对象更改为保存过的工作进程对象  
        _entryHolder = new TimeEntry();  
        _entryHolder.EntryDate = Convert.ToDateTime(DayList.Selection.Value);  
  
        //调用 PopulateFormDetail 方法绑定表单  
        PopulateFormDetail(_entryHolder);  
        //跳转表单到 FormMain  
        ActiveForm = FormMain;  
    }  
}
```

DeleteCommand_Click 方法为删除工作进程按钮的单击事件所触发的方法，该方法从数据库中删除状态保持对象中所保存的工作进程对象，并删除了状态保持对象中的数据，跳转表单到 FormMain。

```
private void DeleteCommand_Click(object sender, System.EventArgs e)  
{  
    //从数据库中删除状态保持对象中所保存的工作进程对象  
    TimeEntry.Remove(_entryHolder.EntryLogID);  
  
    //删除了状态保持对象中的数据  
    _entryHolder = new TimeEntry();  
    _entryHolder.EntryDate = Convert.ToDateTime(DayList.Selection.Value);  
  
    //绑定当前表单  
    PopulateFormDetail(_entryHolder);  
    //跳转表单到 FormMain  
    ActiveForm = FormMain;  
}
```

CancelCommand_Click 方法为取消工作进程浏览按钮的单击事件所触发的方法，该方法和 DeleteCommand_Click 方法类似，只是没有从数据库中删除工作进程的数据。

```
private void CancelCommand_Click(object sender, System.EventArgs e)
{
    //删除了状态保持对象中的数据
    _entryHolder = new TimeEntry();
    _entryHolder.EntryDate = Convert.ToDateTime(DayList.Selection.Value);

    //绑定当前表单
    PopulateFormDetail(_entryHolder);
    //跳转表单到 FormMain
    ActiveForm = FormMain;
}
```

FormCalendar_Activate 方法为表单 FormCalendar 的激活事件所触发的方法，表单 FormCalendar 为选择日期的表单。

```
private void FormCalendar_Activate(object sender, System.EventArgs e)
{
    //从配置文件中获取系统中定义的每周第一天
    int configFirstDay = Convert.ToInt32(ConfigurationSettings.AppSettings[Global.CfgKeyFirstDayOfWeek]);
    //设置日期选择控件 Calendar1 的属性
    Calendar1.FirstDayOfWeek = (System.Web.UI.WebControls.FirstDayOfWeek) configFirstDay;
    Calendar1.SelectedDate = _entryHolder.EntryDate;
    Calendar1.VisibleDate = _entryHolder.EntryDate;
}
```

Calendar1_Change 方法为日期选择控件 Calendar1 的日期更改事件所触发的方法。

```
private void Calendar1_Change(object sender, System.EventArgs e)
{
    HiddenDate.Text = Calendar1.SelectedDate.ToShortDateString();
    _entryHolder.EntryDate = Calendar1.SelectedDate;

    //跳转表单
    if (HiddenOrigin.Text.ToString() == FormName.FormMain.ToString())
        ActiveForm = FormMain;
    else
        ActiveForm = FormDetail;
}
```

PopulateFormDetail 方法是根据参数列表中的工作进程对象绑定表单中控件的方法。

```
private void PopulateFormDetail(TimeEntry entry)
{
    ProjectLabel.Text = "Project: " + entry.ProjectName + " ";
    EntryDescriptionText.Text = entry.Description;
    HoursText.Text = entry.Duration.ToString();
    HiddenLogID.Text = entry.EntryLogID.ToString();
    HiddenProjectID.Text = entry.ProjectID.ToString();
    HiddenProjectName.Text = entry.ProjectName;
    HiddenDate.Text = entry.EntryDate.ToShortDateString();
}
```

BindTimeSheet 方法根据参数列表中的用户 ID、开始日期和结束日期，获得该用户在一段时间内的所有工作进程并绑定到参数列表中的 ObjectList 控件 olGrid。

```
private void BindTimeSheet( ObjectList olGrid, int userID, DateTime start, DateTime end )
{
    //获得用户在该段时间内的工作进程数据
    TimeEntriesCollection entryList = TimeEntry.GetEntries(userID, userID, start, end);

    //设置 olGrid 的默认属性
    olGrid.SelectedIndex = 0;
    olGrid.ViewMode = ObjectListViewMode.List;
    //绑定 ObjectList 控件 olGrid
    if (entryList.Count > 0)
    {
        entryList.Sort( TimeEntriesCollection.TimeEntryFields.Day, true );
        olGrid.DataSource = entryList;
        olGrid.DataBind();
    }
    else
        olGrid.Items.Clear();
}
```

BindProjectList 方法为绑定列表控件 cboProject 的方法，该方法获得参数表中 userID 所对应用户的项目列表并绑定到 cboProject 控件。

```
private ProjectsCollection BindProjectList( SelectionList cboProject, int userID )
{
    ProjectsCollection dsProject = Project.GetProjects(userID, userID);
    cboProject.DataSource = dsProject;
    cboProject.DataBind();

    return dsProject;
}
```

BindCategoryList 方法为绑定项目工作分类的方法。

```
private void BindCategoryList( SelectionList cboCategory, int projectID, int categoryID )
{
    CategoriesCollection dsCategory = Project.GetCategories(projectID);
    cboCategory.DataSource = dsCategory;
    cboCategory.DataBind();
    cboCategory.SelectedIndex = FindCategoryIndex(cboCategory, categoryID);
}
```

BindDayList 方法为获得某个日期所在周的开始和结束日期的方法。

```
private void BindDayList( SelectionList cboDay, DateTime selectedDate, DateTime selectedDay )
{
    DataTable dtEntryWeek = TimeEntry.GetWeek(selectedDate);
    cboDay.DataSource = dtEntryWeek;
    cboDay.DataBind();
    cboDay.SelectedIndex = FindDayIndex(cboDay, selectedDay);
}
```

FindCategoryIndex 方法为获得参数列表中 cboCategory 控件的选中项的方法，此方法遍历 cboCategory 控件并获取其选中值。

```
private int FindCategoryIndex( SelectionList cboCategory, int categoryID )
{
    int index = 0;
```

```

foreach (MobileListItem item in cboCategory.Items)
{
    if (((Category) item.DataItem).CategoryID == categoryID) break;
    index++;
}

if (index == cboCategory.Items.Count)
    return -1;
else
    return index;
}

```

FindProjectIndex 方法为获得参数列表中列表控件 cboProject 的选中项的方法，该方法的实现思路和 FindCategoryIndex 方法类似。

```

private int FindProjectIndex(SelectionList cboProject, int projectID)
{
    int index = 0;
    foreach (MobileListItem item in cboProject.Items)
    {
        if (((Project) item.DataItem).ProjectID == projectID) break;
        index++;
    }

    if (index == cboProject.Items.Count)
        return -1;
    else
        return index;
}

```

在 TimeSheet.aspx 的前台页面中定义了 4 个表单以及表单中的控件，TimeSheet.aspx 的代码清单如下。

```

<%@ Page language="c#" Codebehind="TimeSheet.aspx.cs" Inherits="ASPNET.StarterKit.TimeTracker.
MobileWeb.TimeSheet" AutoEventWireup="false" %>
<%@ Register TagPrefix="mobile" Namespace="System.Web.UI.MobileControls" Assembly="System.Web.Mobile,
Version=1.0.3300.0, Culture=neutral, PublicKeyToken=b03f5f7f11d50a3a" %>
<meta content="Microsoft Visual Studio 7.0" name="GENERATOR">
<meta content="C#" name="CODE_LANGUAGE">
<meta content="http://schemas.microsoft.com/Mobile/Page" name="vs_targetSchema">
<body Xmlns:mobile="http://schemas.microsoft.com/Mobile/WebForm">
<!-- 引入样式表定义文件 -->
<mobile:stylesheet id="Stylesheet1" ReferencePath="Styles.ascx" runat="server"></mobile:stylesheet>
<!-- FormMain 表单 -->
<mobile:form id="FormMain" runat="server" StyleReference="Form">
    <mobile:label id="Title" runat="server" StyleReference="title">Time Sheet</mobile:label>
    <mobile:label id="WeekEndingMain" runat="server" BreakAfter="False">Week Ending</mobile:label>
    <!-- 更改当前周的按钮 -->
    <mobile:command id="ChangeWeekMain" runat="server" Wrapping="NoWrap" Format="Link">Change
    Week</mobile:command>
    <mobile:label id="TotalHoursLabel" runat="server">Total Hours:</mobile:label>
    <!-- 用户显示工作进程列表的 objectlist 控件 -->
    <mobile:objectlist id="TimeEntryGrid" runat="server" CommandStyle-ForeColor="White" CommandStyle-
    StyleReference="subcommand" CommandStyle-BackColor="White" LabelField="Description" TableFields="Day;Hours;

```

```
Description" AutoGenerateFields="False" LabelStyle-StyleReference="list-header" BackColor="White">
    <command name="SelectCommand" text="Need to create a command here to make the first column a link"></command>
        <!-- 在列表中显示日期字段 -->
        <field title="Day" name="Day" dataformatstring="{0:ddd}" datafield="EntryDate"></field>
        <!-- 在列表中显示工作量字段 -->
        <field title="Hrs" name="Hours" dataformatstring="{0:f}" datafield="Duration"></field>
        <!-- 在列表中显示工作进程描述字段 -->
        <field title="Description" name="Description" datafield="Description"></field>
        <!-- 工作进程 Id 字段 -->
        <field title="EntryLogID" name="EntryLogID" datafield="EntryLogID" visible="False"></field>
</mobile:objectlist>
<mobile:panel id="Panel2" runat="server">
    <mobile:devicespecific id="DeviceSpecific2" runat="server">
        <choice filter="IsHTML32">
            <contenttemplate>
                <div width="100%" align="center">
                    <!-- 定义添加工作进程的按钮 -->
                    <input id="Button1" type="button" value="Add Entry" name="Button1" runat="server" onserverclick="AddEntryCommand2_Click"></input>
                </div>
            </contenttemplate>
        </choice>
    </mobile:devicespecific>
    <mobile:command id="AddEntryCommand2" runat="server" BreakAfter="False" Wrapping="NoWrap" Alignment="Center">Add Entry</mobile:command>
</mobile:panel>
<!-- 结束 MianForm 的定义 -->
</mobile:form>
<!-- 定义 FormProject 表单 -->
<mobile:form id="FormProject" runat="server" StyleReference="Form">
    <mobile:label id="Label2" runat="server" StyleReference="title" Font-Name="Arial">Project</mobile:label>
    <mobile:label id="Label1" runat="server">Select a Project:</mobile:label>
    <!-- 选择项目的 selectionlist 控件 -->
    <mobile:selectionlist id="ProjectListBox" runat="server" Wrapping="NoWrap" Rows="6" SelectType="ListBox" DataTextField="Name" DataValueField="ProjectID"></mobile:selectionlist>
    <mobile:command id="SelectProjectCommand" runat="server">Select</mobile:command>
    <!-- 结束 FormProject 表单 -->
</mobile:form>
<!-- 定义 FormDetail 表单 -->
<mobile:form id="FormDetail" runat="server" StyleReference="Form">
    <mobile:label id="Label3" runat="server" StyleReference="title">Time Entry</mobile:label>
    <!-- 定义显示当前所在周结束的 label 控件 -->
    <mobile:label id="WeekEndingDetail" runat="server" BreakAfter="False">Week Ending:</mobile:label>
    <mobile:command id="ChangeWeekDetail" runat="server" Wrapping="NoWrap" Format="Link"> Change Week</mobile:command>
    <mobile:label id="ProjectLabel" runat="server" BreakAfter="False">Project:</mobile:label>
    <mobile:command id="ChangeProjectCommand" runat="server" Wrapping="NoWrap" Format="Link"> Change</mobile:command>
    <mobile:label id="CategoryLabel" runat="server">Category:</mobile:label>
```

```

<mobile:selectionlist id="CategoryList" runat="server" DataTextField="Abbreviation"
DataValueField="CategoryID"></mobile:selectionlist>
    <mobile:label id="DayLabel" runat="server">Day:</mobile:label>
    <mobile:selectionlist id="DayList" runat="server" DataTextField="Day" DataValueField="Date">
</mobile:selectionlist>
    <mobile:label id="HoursLabel" runat="server">Hours:</mobile:label>
    <mobile:textbox id="HoursText" runat="server"></mobile:textbox>
    <mobile:label id="EntryDescriptionLabel" runat="server">Description:</mobile:label>
    <mobile:textbox id="EntryDescriptionText" runat="server"></mobile:textbox>
    <mobile:command id="SaveCommand" runat="server">Save</mobile:command>
    <mobile:command id="DeleteCommand" runat="server">Delete</mobile:command>
    <mobile:command id="CancelCommand" runat="server">Cancel</mobile:command>
    <mobile:requiredfieldvalidator id="RequiredFieldValidatorHrs" runat="server" Font-Size="Small"
ControlToValidate="HoursText" ErrorMessage="Hours is required"></mobile:requiredfieldvalidator>
    <mobile:requiredfieldvalidator id="RequiredFieldValidatorDesc" runat="server"
Font-Size="Small" ControlToValidate="EntryDescriptionText" ErrorMessage="Description is
required"></mobile:requiredfieldvalidator>
    <mobile:comparevalidator id="CompareValidatorHrs" runat="server" Font-Size="Small" ControlToValidate
="HoursText" ErrorMessage="Hours field must be numeric" Operator="DataTypeCheck" Type="Currency">
</mobile:comparevalidator>
    <mobile:label id="HiddenProjectName" runat="server" Visible="False">0</mobile:label>
    <mobile:label id="HiddenProjectID" runat="server" Visible="False">0</mobile:label>
    <mobile:label id="HiddenLogID" runat="server" Visible="False">0</mobile:label>
    <mobile:label id="HiddenDate" runat="server" Visible="False">EntryDate</mobile:label>
    <mobile:label id="HiddenOrigin" runat="server" Visible="False">Originated Form</mobile:label>
<!--结束 FormDetail 表单 -->
</mobile:form>
<!-- 定义 FormCalendar 表单 -->
<mobile:form id="FormCalendar" runat="server" StyleReference="Form">
    <mobile:label id="Label4" runat="server" StyleReference="title">Select Week</mobile:label>
    <!-- 定义日期选择控件 Calendar1 -->
    <mobile:calendar id="Calendar1" runat="server" StyleReference="standard-text" BackColor="White"
Font-Name="verdana" Font-Size="Small"></mobile:calendar>
    <!-- 结束 FormCalendar 表单 -->
</mobile:form>
</body>

```

17.3 开发启示

本章详细讲述了 ASP.NET Time Tracker Starter Kit 的移动 Web 应用模块的技术实现方法。通过对 ASP.NET Time Tracker Starter Kit 中移动 Web 应用模块的分析，读者应该从中学到移动 Web 应用模块的开发思路和开发方法。移动 Web 应用的开发和普通 Web 应用的开发既有相似的地方也有不同的地方，希望通过本章的学习读者能够了解以下知识点。

- 移动 Web 应用的身份验证机制
- 移动 Web 应用中状态保持对象的使用
- 移动 Web 应用的页面多表单机制
- 包括 objectlist、command 等控件在内的移动 Web 应用中常用控件的应用