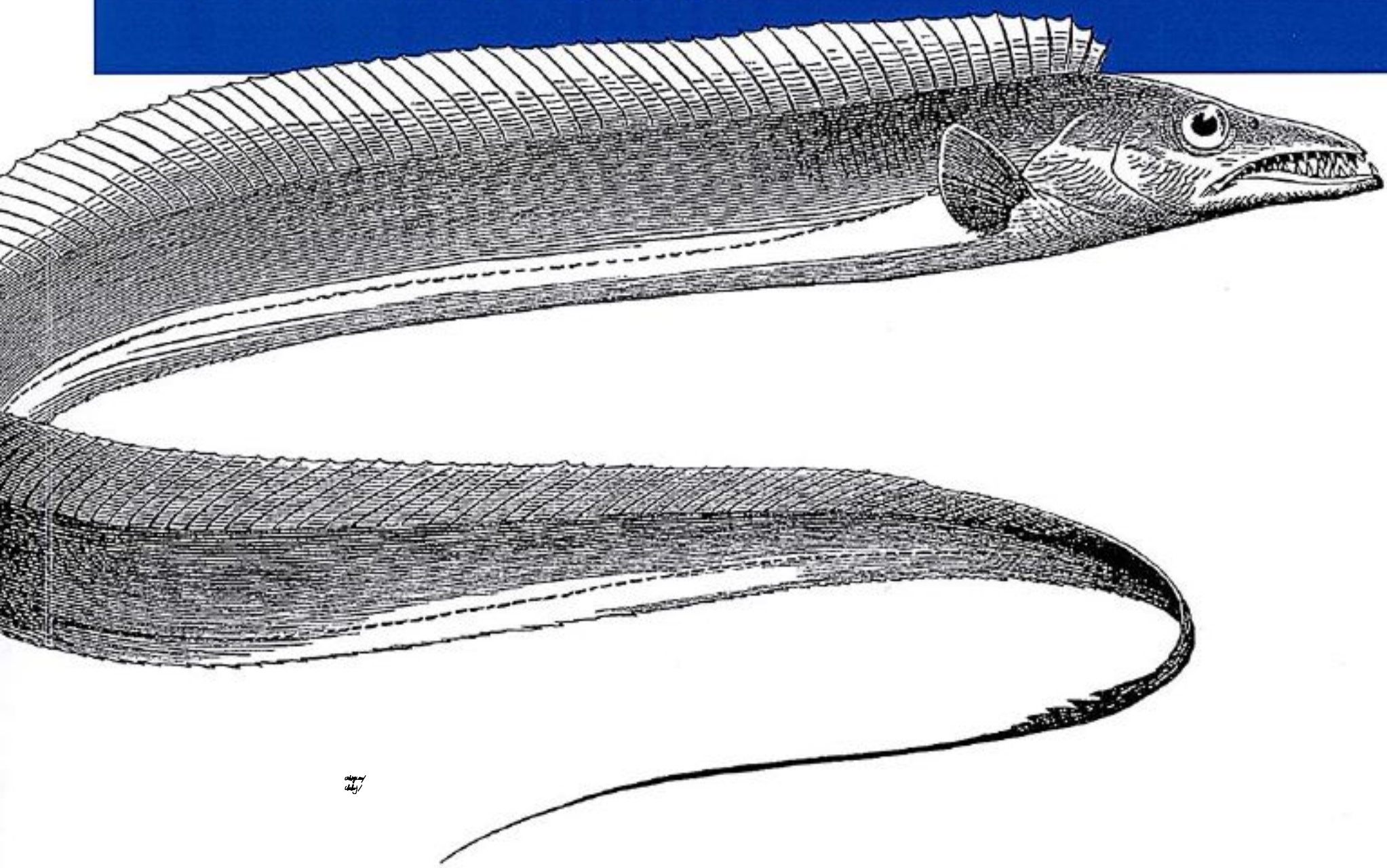
Programming ASP.NET MVC 4

ASP.NET MVC 4 Web编程



www.cnblogs.com/chehaoj

Jess Chadwick, Todd Snyder,
Hrusikesh Panda 著
徐雷 徐扬 译

Jackehe

O'REILLY®

華中科技大學出版社
http://www.hustp.com

ASP.NET MVC 4 Web编程

本书介绍了微软最新的ASP.NET MVC 4框架，帮助我们了解整个ASP.NET MVC 4框架的运行原理，包括如何使用ASP.NET MVC 4框架构建Web应用程序以及如何在真实开发场景中使用ASP.NET MVC 4框架新特性来解决不同的问题。通过本书的学习，将学会如何使用HTML、JavaScript、Entity Framework以及其他Web相关的技术，比如Web网站性能优化、缓存等技术。

通过本书的学习，我们不仅可以了解最基本的MVC架构模式，而且还可以了解与此相关的高级知识点。为了便于大家更好地学习ASP.NET MVC 4框架开发的最佳实践经验以及相关知识，作者特意编写了一个电子交易网站“EBuy”应用程序，作为参考实例。

- 了解ASP.NET MVC和ASP.NET Web Form的共同点
- 使用Entity Framework创建、维护Web应用数据库
- 创建富Web应用，使用jQuery框架进行客户端开发
- 使用AJAX实现网站无刷新交互
- 学习如何创建、调用ASP.NET Web API服务
- 为移动设备开发功能强大、一致体验的网站
- 学习错误处理、自动化测试以及自动化生成技术
- 使用不同方式部署ASP.NET MVC 4应用程序

“对于想要学习ASP.NET MVC Web编程的开发人员来说，本书无疑是最佳选择。作者分享了许多实际项目开发的最佳实践经验，包括如何使用jQuery、Entity Framework、ASP.NET Web API进行应用程序开发。”

——Marcin Dobosz
微软高级开发工程师

Jess Chadwick：专注于Web技术领域，独立软件技术顾问；有超过10年的软件开发经验，就职的公司包括嵌入式设备创业公司、世界500强企业；ASPInsider、微软ASP.NET 领域的MVP，经常在新泽西的NJDOTNET进行技术讲座。

Todd Snyder：有超过18年软件开发经验；微软平台的企业级系统架构提供技术咨询和开发指导；就职于美国Infragistics公司，经常作为讲师受邀参加各种技术会议。

Hrusikesh Panda：架构师、RIA专家、Web架构师以及开源爱好者。

图书分类：程序设计/.NET

策划编辑：谢燕群
责任编辑：谢燕群 陈元玉

O'REILLY®
oreilly.com.cn

O'Reilly Media, Inc.授权华中科技大学出版社出版

此简体中文版仅限于在中华人民共和国境内（但不允许在中国香港、澳门特别行政区和中国台湾地区）销售发行
This Authorized Edition for sale only in the territory of People's Republic of China (excluding Hong Kong, Macao and Taiwan)

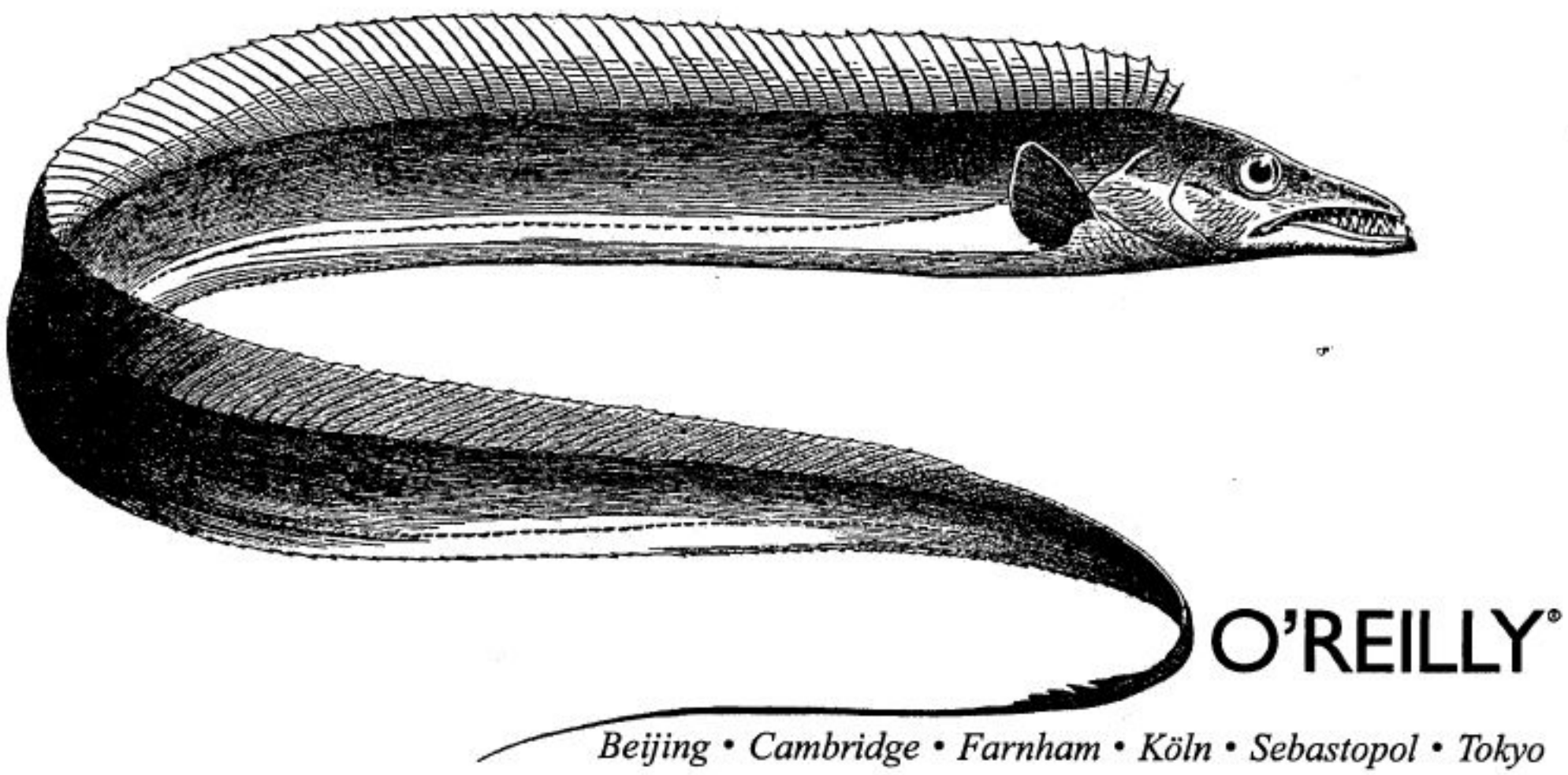
ISBN 978-7-5609-9114-6
9 787560 991146 >
定价：68.00元

ASP.NET MVC 4 Web 编程

Jess Chadwick,Todd Snyder

Hrusikesh Panda 著

徐雷 徐扬 译



O'Reilly Media, Inc. 授权华中科技大学出版社出版

华中科技大学出版社

中国·武汉

内 容 简 介

本书介绍了微软最新的 ASP. NET MVC 4 框架，包括如何使用 ASP. NET MVC 4 框架构建 Web 应用程序，ASP. NET MVC 4 框架的运行原理，如何在真实开发场景中使用 ASP. NET MVC 4 框架新特性来解决不同的需求，如何使用 HTML、JavaScript、Entity Framework 以及其他 Web 相关的技术。同时，分享了许多实际项目开发的最佳实践经验，包括如何使用 jQuery、Entity Framework、ASP. NET Web API 进行应用程序开发，而且还深入阐述了许多高级知识点，如网站 SEO、性能优化、日志、缓存等。

978-1-449-32031-7 Programming ASP. NET MVC 4 © 2012 by O'Reilly Media, Inc. Simplified Chinese Edition, jointly published by O'Reilly Media, Inc. and Huazhong University of Science and Technology Press, 2013. Authorized translation of the English edition, 2012 O'Reilly Media, Inc., the owner of all rights to publish and sell the same.

All rights reserved including the rights of reproduction in whole or in part in any form.

本书中文简体版专有出版权由 O'Reilly Media, Inc. 授予华中科技大学出版社，未经许可，不得以任何方式复制或抄袭本书的任何部分。

湖北省版权局著作权合同登记图字：17-2013-092

图书在版编目(CIP)数据

ASP. NET MVC 4 Web 编程/(美)查德威克(Chadwick, J.)，(美)斯奈德(Snyder, T.)，(美)潘达(Panda, H.) 著；徐雷，徐扬译. —武汉：华中科技大学出版社，2013.7

ISBN 978-7-5609-9114-6

Ⅰ. ①A… Ⅱ. ①查… ②斯… ③潘… ④徐… ⑤徐… Ⅲ. ①网页制作工具—程序设计 Ⅳ. ①TP393.092

中国版本图书馆 CIP 数据核字(2013)第 124664 号

Jess Chadwick, Todd Snyder, Hrusikesh Panda 著

ASP. NET MVC 4 Web 编程　　徐 雷 徐 扬 译

策划编辑：谢燕群　　责任校对：张 琳

责任编辑：谢燕群 陈元玉　　责任监印：周治超

出版发行：华中科技大学出版社(中国·武汉)

武昌喻家山　邮编：430074　电话：(027)81321915

录　排：武汉金睿泰广告有限公司

印　刷：湖北新华印务有限公司

开　本：787mm×980mm 1/16

印　张：26.25

字　数：622 千字

版　次：2013 年 7 月第 1 版第 1 次印刷

定　价：68.00 元

华中出版

本书若有印装质量问题，请向出版社营销中心调换

全国免费服务热线：400-6679-118　竭诚为您服务

版权所有　侵权必究

O'Reilly Media, Inc.介绍

O'Reilly Media通过图书、在线服务、杂志、调查研究和会议等方式传播创新者的知识。自1978年开始O'Reilly一直都是发展前沿的见证者和推动者。超级极客正在开创未来，我们关注着真正重要的技术趋势，通过放大那些“微弱的信号”来刺激社会对新科技的采用。作为技术社区中活跃的参与者，O'Reilly的发展充满着对创新的倡导、创造和发扬光大。

作为出版商O'Reilly为软件开发人员带来革命性的“动物书”，创造了第一个商业网站（GNN），组织开放源代码峰会以至于开源软件运动以此命名，通过创立Make杂志成为DIY革命的主要先锋，公司一如既往地用各种方式和渠道连接人们和他们所需要的信息。O'Reilly的会议和峰会聚集了超级极客和高瞻远瞩的商业领袖，共同描绘将开创新产业的革命性思想。作为技术人士获取信息的选择O'Reilly现在还将先锋专家的知识传递给普通计算机用户。无论是通过印刷书籍、在线服务或者面授课程，每一项O'Reilly的产品都反映了公司不可动摇的信念——信息是激发创新的力量。

业界评论

“O'Reilly Radar博客有口皆碑。”

——Wired

“O'Reilly凭借一系列（真希望当初我也想到了）非凡想法建立了数百万美元的业务。”

——Business 2.0

“O'Reilly Conference是聚集关键思想领袖的绝对典范。”

——CRN

“一本O'Reilly的书就代表一个有用、有前途、需要学习的主题。"

——Irish Times

“Tim是位少有的商人，他不光放眼于最长远、最广阔的视野并且切实地按照Yogi Berra的建议去做了：“如果你在路上遇到岔路口，走小路（岔路）。”回顾过去Tim似乎每一次都选择了小路，而且有几次都是一闪即瞬的机会，尽管大路也不错。”

——Linux Journal

译者序

——献给有节操的程序员！

非常荣幸，我受华中科技大学出版社委托担任经典技术书籍《Programming ASP.NET MVC 4》的翻译工作。这是我在翻译《WCF技术内幕》与《WCF服务编程》（第三版）之后翻译的第三本书。为了延续著名计算机图书出版公司O'REILLY的经典蓝皮书系列图书的特色，中文版本命名为《ASP.NET MVC 4 Web 编程》，与经典书籍《WCF服务编程》（第三版）一致。

为什么要学习ASP.NET MVC 框架开发？

相信很多人会有类似的问题：为什么要学习ASP.NET MVC？Web Form不是已经很好用了吗？干吗还要费劲学习这门技术？下面是我根据相关技术的开发工作出发整理的一些理由，供各位参考。

MVC模式最早在1978年由特里夫·里斯高（Trygve Reenskaug）提出，是施乐帕罗奥多研究中心（Xerox PARC）在20世纪80年代为程序语言Smalltalk发明的一种软件设计模式。MVC架构模式诞生30年后，因其提供的良好的松耦合、易于扩展、高可维护性等优点，重新在开发社区火起来。

ASP.NET MVC框架诞生于2007年12月10日，但是第一版于2009年3月17日发布，最新的ASP.NET MVC 4.0则是于2012年8月15日正式发布，并且集成到.NET 4.5中，Visual Studio 2012提供完美的开发支持。

以下是来自国外社区的建议：

1) 学习一种新的编程思维方式；
2) 学习一种完全不同的架构；
3) 强迫你熟悉HTML和HTTP协议规范；
4) ASP.NET MVC更好地支持单元测试（Unit Test）；
5) ASP.NET MVC将使你意识到你对ASP.NETWeb Form有多少是想当然的。

内容简介

本书全面介绍了微软最新的ASP.NET MVC 4框架，包括如何设计、构建、测试、部署ASP.NET MVC 4 Web应用程序。它可以帮助我们了解整个ASP.NET MVC 4框架的运行原理，以及如何在真实开发场景中使用ASP.NET MVC 4框架新特性来解决不同的需求。具体内容如下：

1) ASP.NET MVC与ASP.NET Web Form框架的底层差别；

2) Web应用程序架构设计、网站性能优化；

3) jQuery框架进行客户端开发；

4) Ajax实现网站无刷新交互；

5) Entity Framework与Web数据交互；

6) 各种Cache缓存机制；

7) 开发ASP.NET Web API服务

8) 移动设备开发网站

9) 错误处理、自动化测试、以及自动化生成技术

10) 部署ASP.NET MVC 4应用程序

11) OOP面向对象编程和Web架构设计

12) ASP.NET MVC 4 开发实践经验和指导原则

面向读者群

本书是学习ASP.NET MVC网站开发的.NET工程师、开发经理、微软.NET架构师等的最佳选择。此外，本书也可以作为高等学校的教材，适合计算机、通信、软件工程、电子等专业的学生学习。

致谢

在本书的翻译过程中，得到许多朋友热心的帮助。我的弟弟徐扬帮助我分担了部分工作；在翻译完毕后，时军帅、朱伟、郭俊超、余东升、景洋、王进祥、薛庆、蒋悠悠、王汉忠、杨威（排名不分先后）进行了审校工作，在此衷心地表示感谢！

译者寄语

虽然已经十分细心，但是难免会出现翻译错误，也希望各位多批评、指正！本书专门提供了读者交流QQ群：44206115。欢迎各位加入交流！

老徐为人算是比较靠谱，不喜欢忽悠，一直反对国内IT行业内的浮躁风气。在.NET平台诞生11年之际，老徐谨将本书献给：

那些在浮躁技术氛围中，仍然保持务实态度，不失节操、不断追求的技术人员！

老徐FrankXuLei

2013年5月28日

新浪微博：@老徐FrankXuLei

1) 吉林大学计算机科学与技术学士学位，上海交通大学软件工程硕士学位；

2) 微软最有价值专家（系统集成方向）微软WCF和Azure云计算中文技术论坛版主；

3) 《WCF技术内幕》与《WCF服务编程》第三版、《ASP.NET MVC4 Web 编程》译者；

4) 微软特邀讲师、技术顾问、架构师，多次参与微软MSDN官方WCF和 MVC资料的翻译和校订工作；

5) 专注于.NET平台下分布式应用系统开发和企业应用系统集成领域技术研究；

6) 喜欢格言：谦卑若愚，好学若饥（Stay Hungry, Stay Foolish）。

目录
Table of Contents

前言 I

第一部分　千里之行，始于足下

第 1 章　ASP.NET MVC 基础 3
Microsoft Web 开发平台 3
活动服务页面 3
ASP.NET Web 表单 4
ASP.NET MVC 4
MVC 架构 4
模型 5
视图 5
控制器 5
ASP.NET MVC 4 的新特性 6
EBuy 介绍 7
安装 ASP.NET MVC 8
创建 ASP.NET MVC 4 应用程序 8
项目模板 10
惯例优先原则 12
运行程序 13
路由 13
配置路由 14
控制器 16
控制器操作 16
操作结果 17
操作参数 18
操作过滤器 20
视图 21
定位视图 21
Razor，你好！ 22
区分代码和标记语言 23
布局 24
部分视图 25
显示数据 26
HTML 和 URL 帮助方法 29

模型 29
群英荟萃 30
路由 30
控制器 30
视图 32
验证 35
账号控制器 36
总结 38

第 2 章　ASP.NET Web Form 开发人员必读 39
ASP.NET 同门兄弟 39
工具、语言和 API 39
HTTP 处理程序和模块 40
管理状态 40
部署和运行时 41
更多的差别 41
应用程序业务和视图分离 42
URL 和路由 42
状态管理 43
渲染 HTML 代码 43
使用 Web Form 语法编写 ASP.NET MVC 视图 46
要点提示 47
总结 48

第 3 章　使用数据 49
构建表单 49
处理表单 Post 51
保存数据到数据库 51
代码优先：惯例优先原则 51
使用 Entity Framework 代码优先模式创建数据访问层 52
验证数据 53
使用数据声明指定业务规则 54
显示验证错误 56
总结 58

第 4 章　客户端开发 59
使用 JavaScript 59
选择器 61
处理事件 63
DOM 操作 65
AJAX 66
客户端验证 68
总结 72

第二部分　欲穷千里目，更上一层楼

第 5 章　Web 应用程序架构 75
模型-视图-控制器模式 75
分离关注点 75
MVC 与 Web 框架 76
Web 应用架构设计 77
逻辑设计 78
ASP.NET MVC Web 应用程序的逻辑设计 78
逻辑设计的最佳实践 80
物理设计 80
项目命名空间和程序集名称 81
部署选项 81
物理设计的最佳实践 82
设计原则 83
SOLID 原则 84
控制反转 89
DRY 原则 96
总结 96

第 6 章　使用 AJAX 提升网站体验 97
部分渲染 97
渲染部分视图 98
JavaScript 渲染 102
渲染 JSON 数据 102
请求 JSON 数据 104
客户端模板 104
重用跨 AJAX 和非 AJAX 请求逻辑代码 107
响应 AJAX 请求 108
处理 JSON 请求 108
跨控制器操作指定统一逻辑 109
发送数据到服务器 111
提交复杂的 JSON 对象 111
选择模型绑定器 113
高效地收发 JSON 数据 114
跨域 AJAX 请求 115
JSONP 115
启用跨站资源共享 118
总结 119

第 7 章　ASP.NET Web API 121
构建 Data Service 121
注册 Web API 路由 123
依赖惯例优先原则 123
重写惯例 124

钩住 API 124
数据分页与查询 127
异常处理 128
Media 格式化器 130
总结 132

第 8 章　高级数据 133
数据访问模式 133
Plain Old CLR Objects 133
使用 repository 模式 134
对象关系映射器 136
实体框架概述 137
选择数据访问方法 138
数据库并发 138
构建数据访问层 140
使用 Entity Framework 代码优先方法 140
EBuy 业务域模型 142
使用数据上下文 145
排序、过滤以及数据分页 146
总结 151

第 9 章　安全 153
构建安全的 Web 应用 153
深度防御 153
不信任任何输入数据 154
执行最小权限原则 154
假设外部系统是危险的 154
减少裸露面 154
关闭不必要的功能 154
保护程序 154
保护局域网应用 155
表单验证 159
防御攻击 166
SQL 注入 167
跨站脚本 170
跨站请求伪造 171
使用 ASP.NET MVC 来防御 CSRF 172
总结 173

第 10 章　移动 Web 网站开发 175
ASP.NET MVC 4 移动特性 175
让移动应用变得更友善 176
创建 Auction 移动视图 177
使用 jQuery Mobile 框架 178
增强视图 179
禁止移动网站显示桌面视图 183

改进移动用户体验 184
自适应渲染 184
Viewport 标签 184
移动特性探测 185
CSS 媒体查询 187
浏览器专用视图 188
从零开始创建新的移动应用 188
jQuery Mobile 范式转换 189
ASP.NET MVC 4 移动应用模板 189
使用 ASP.NET MVC 4 移动应用模板 191
总结 193

第三部分　会当临绝顶，一览纵山小

第 11 章　并行计算、异步和实时数据操作 197
异步控制器 197
创建异步控制器 197
何时使用异步控制器 199
实时异步通信 200
对比应用模型 200
HTTP 轮询 200
HTTP 长轮询 201
服务端推送事件 202
WebSockets 203
增强实时通信 203
配置和调整 207
总结 208

第 12 章　缓存 209
缓存的类型 209
服务端缓存 209
客户端缓存 209
服务端缓存技术 210
请求域内的缓存 210
用户域内的缓存 210
应用程序域内的缓存 211
ASP.NET 缓存 212
输出缓存 213
甜甜圈缓存 216
甜甜圈洞缓存 218
分布式缓存 219
客户端缓存技术 223
理解浏览器缓存 224
AppCache 缓存 225
本地存储 227
总结 228

第 13 章　客户端优化技术 229
页面剖析 229
HttpRequest 剖析 229
最佳实践 231
减少 HTTP 请求 231
使用 CDN 内容分发网络 233
添加 Expires 或 Cache-Control 消息头 234
GZip 组件 235
置顶样式文件 236
置底脚本文件 236
迁出脚本和样式代码 237
减少 DNS 查询 238
压缩 JavaScript 和 CSS 238
避免重定向 239
删除重复脚本 240
配置实体标签 241
测试客户端性能 241
运行 ASP.NET MVC 244
捆绑和压缩 244
总结 247

第 14 章　高级路由 249
路标指示系统 249
URL 和搜索引擎优化 250
构建路由 251
路由参数 252
路由顺序和优先级 254
路由到现有文件 254
忽略路由 254
捕获所有路由 255
路由约束 256
使用 Glimpse 观察路由 258
基于属性标记的路由 258
扩展路由 262
路由管道 262
总结 266

第 15 章　可复用 UI 组件 267
ASP.NET MVC 框架提供了什么 267
部分视图 267
HtmlHelper 扩展或自定义 HtmlHelper 267
显示和编辑模板 268
Html.RenderAction() 268
更进一步 268
Razor 单文件生成器 269
创建可复用的 ASP.NET MVC 视图 269

创建可复用 ASP.NET MVC Helpers 273
单元测试 Razor 视图 275
总结 277

第四部分　质量控制

第 16 章　日志 281
ASP.NET MVC 中的错误处理 281
启用自定义错误 281
控制器操作中的错误处理 282
定义全局错误处理器 283
日志和跟踪 285
记录错误日志 285
ASP.NET 健康监控 287
总结 289

第 17 章　自动化测试 291
测试的语义 291
人工测试 291
自动化测试概述 292
自动化测试的级别 293
单元测试 293
集成测试 295
验收测试 295
什么是自动化测试项目？ 296
创建 Visual Studio 测试项目 297
创建并执行单元测试 298
测试 ASP.NET MVC 应用程序 300
测试模型 300
测试驱动开发 303
编写干净的自动化测试代码 304
测试控制器 306
重构单元测试 309
模拟依赖 309
测试视图 313
代码覆盖率 315
100%代码覆盖率的秘密 316
开发可测试的代码 317
总结 318

第 18 章　自动化生成 319
创建生成脚本 319
Visual Studio 项目就是生成脚本 320
添加简单的生成任务 320
执行生成 320
一切皆有可能 321

自动化生成概述 321
自动化生成的类型 322
创建自动化生成 323
持续集成 326
发现问题 326
持续集成原则 327
总结 330

第五部分　实施

第 19 章　部署 333
需要部署什么 333
网站核心文件 333
静态内容 335
为什么不部署？ 335
数据和其他依赖 336
发布 EBuy 交易网站需要做的工作 337
部署到 IIS 服务器 337
必备条件 337
创建并配置 IIS 网站 338
使用 Visual Studio 发布 339
部署到 Windows Azure 342
创建 Windows Azure 账号 342
创建新的 Windows Azure 网站 343
通过源代码控制软件发布 Windows Azure 网站 343
总结 345

第六部分　附录

附录 A：ASP.NET MVC 与 Web Form 集成 349

附录 B：作为平台使用 NuGet 355

附录 C：ASP.NET MVC 开发最佳实践原则 371

附录 D：交叉引用：目标主题、特性和场景 383

索引 387

前言
Preface

Web应用的范围很广，种类繁多。Microsoft的ASP.NET框架构建于最可靠的平台——强大、成熟的.NET框架之上。ASP.NET MVC是微软提供的最新Web开发框架，是ASP.NET提供给Web开发人员的一种全新的开发技术，可帮助我们轻松地开发Web应用程序。

本书的目的非常简单：帮助我们学习并完全理解ASP.NET MVC 4开发框架。另外，本书不仅仅局限于此，它还把ASP.NET MVC的基本概念与实际项目结合起来。最新的Web开发技术（比如HTML 5和jQuery JavaScript框架）以及强大的设计模式使我们不仅会使用ASP.NET MVC框架开发网站，而且知道如何构建稳定、可伸缩、易于扩展、易于维护的Web应用程序。

本书的读者

本书面向那些希望学习如何使用Microsoft ASP.NET MVC框架开发强壮、可维护的Web应用的技术人员。本书使用了大量示例代码来详细介绍知识概念，不仅仅只面向开发人员。本书介绍的很多概念和技巧不仅有助于开发人员编写代码，更有利于Team Leader驾驭项目开发过程。

阅读前的准备

既然本书的目标旨在教我们如何使用Microsoft ASP.NET MVC框架开发强壮、可维护的Web应用，那么，这就要求我们已经掌握了Microsoft .NET 框架开发的基本知识。换句话说，我们应该对使用HTML、CSS、JavaScript构建简单的网站十分了解，并且十分熟悉.NET和C#编程语言。“Hello World”程序只能算是“小儿科”。

本书代码可以在此下载：https://github.com/ProgrammingAspNetMvcBook/CodeExamples

本书使用的体例

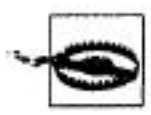
下面的排版格式是本书所使用的：

斜体（以及黑体）：用于技术术语、URL。

等宽字体：用于示例代码、描述、命名空间、class、程序集、接口规范、操作符、属性以及保留字。

等宽加粗字体：用于强调示例代码。

表示提示、建议或一般注释。



表示警告或注意。

使用示例代码

本书能帮助你更好地完成工作。一般来说，可以在程序和项目里直接使用本书的代码，不需要联系我们，除非复制了大量的代码。例如，使用本书的几块代码是不需要授权的，但销售或传播CD-ROM包含本书的代码就需要授权；引用本书的内容和代码页不需要授权，但在文档或者产品里大量使用本书的代码就需要授权。十分感谢你在部分引用内容时添加注释，注释应该包含标题、作者、出版社和ISBN，例如"Programming ASP.NET MVC 4, by Jess Chadwick, Todd Synder, and Hrusikesh Panda (O'Reilly). Copyright 2012 Jess Chadwick, Todd Synder, and Hrusikesh Panda, 978-1-449-32031-7."。如果在使用代码中遇到疑问或者要获得授权，请直接通过Email与我们联系，地址为：*permissions@oreilly.com*。

如何联系我们

请将本书的有关意见和问题告知出版商。

美国：

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472

中国：

北京市西城区西直门南大街2号成铭大厦C座807室（100035）
奥莱利技术咨询（北京）有限公司

O'Reilly有一个关于本书的网页，这里列出了勘误、例子和任何额外的信息。你可以访问这个网页：

http://www.oreilly.com/catalog/9781449320317/（原书）

本书还有自己的网站：

http://www.oreilly.com.cn/book.php?bn=978-7-5609-9114-6（中文版）

评论或者提出关于本书的技术问题，可以发送到email：

info@mail.oreilly.com.cn

关于本书的更多信息、会议、资料中心和网站，请访问以下网站：

http://www.oreilly.com
http://www.oreilly.com.cn

第一部分

千里之行，始于足下

Up and Running

第1章

ASP.NET MVC基础

Fundamentals of ASP.NET MVC

Microsoft ASP.NET MVC是一个全新的Web开发框架，它构建于大名鼎鼎的.NET平台之上。ASP.NET MVC框架完全支持那些提倡松耦合、高可维护性的开发模式和实践原则。

在本章里，首先会学习ASP.NET MVC的运行原理——从它支持的架构概念到如何使用Microsoft Visual Studio 2011开发出功能完整的ASP.NET MVC Web网站。然后，会深入学习ASP.NET MVC Web程序项目，从第一步创建项目开始，看看ASP.NET MVC带给我们什么新的东西，包括创建网页、使用内置的Form验证机制允许用户注册并登录你的网站。最后，不仅要学习如何开发ASP.NET MVC 网站，还要完全理解ASP.NET MVC的基本原理。本书的其余部分就是在讲解这些基本的知识点，介绍如何使用ASP.NET MVC框架去开发网站应用程序。

Microsoft Web开发平台

Microsoft's Web Development Platforms

了解过去是为了更好地面向未来。在开始学习ASP.NET MVC之前，先花点时间来了解一下其历史背景。

很久以前，微软看到基于Windows系统的Web开发平台的需求后就提供了自己的解决方案。在过去的20年中，微软已经向开发社区提供了多个Web开发平台。

活动服务页面

微软的第一个Web开发平台是ASP，它将脚本语言和代码放置于同一个文件里，网站里的每个页面对应一个物理文件。至今，仍有很多ASP网站还在运行。随着时间的推移，开发人员希望得到更多的支持，希望改进代码的复用性、更好地分离关注点以及更方便地进行面向对象的编程开发。在2002年，微软提供了一个新的Web开发平台——ASP.NET来满足这些需求。

ASP.NET Web表单

与ASP一样，ASP.NET网站也提供了基于页面的方式，每个页面对应一个物理文件，称为Web Form（Web表单），并且可以通过文件名访问。与ASP不同的是，Web Form页面提供了代码分离机制，把代码文件和HTML标签分离到两个不同的文件中。ASP.NET Web Form已经发展了很多年，但仍是很多开发人员的选择之一。也有一些.NET开发者认为ASP.NET Web Form过于抽象了底层的HTML、JavaScript和CSS！

ASP.NET MVC

微软很快发现了ASP.NET开发人员的新需求，这些需求不同于之前基于页面的Web Form方法。于是，微软在2008年发布了第一版ASP.NET MVC。这与之前的Web Form方法完全不同，ASP.NET MVC抛弃了基于页面的架构风格，使用了全新的MVC（模型-视图-控制器）架构。

与ASP.NET Web Form取代ASP不同，ASP.NET MVC并没有取代ASP.NET Web Form的意思。恰恰相反，ASP.NET MVC和ASP.NET Web Form可以共存。它们都构建于ASP.NET框架之上，并且都使用了很多相同的Web API。

ASP.NET MVC和ASP.NET Web Form只是开发ASP.NET网站的不同方法，这是贯穿本书的基本观点。第2章和附录A对这些概念做了更深入的介绍。

MVC架构

The Model-View-Controller Architecture

MVC模式是一种严格实现应用程序各部分隔离的架构模式。这种“隔离”有一个更响亮的名字“分离关注点”，更通俗的名称是“松耦合”。实际上，MVC架构包括后面的ASP.NET MVC框架，都是以实现应用程序模块之前的隔离为目标的。

005

松耦合的应用程序架构设计方式，无论是短期还是长期，都能带来巨大的好处。

开发

单个组件不直接依赖于其他组件，这就意味着每个组件可以独立部署，也可以被替换。这种不使用单一文件编译组件的方式减少了与之相关的组件之间的影响。

测试

组件之间的松耦合带来的好处就是允许测试代码可以替换真实的产品组件。这样可以尽量减少直接调用数据库，可以直接让数据库调用组件返回静态数据。这种模拟测试大大地简化、提升了系统的真实性测试流程。

维护

隔离组件逻辑意味着把影响隔离到最少的组件中——通常只有一个。改变的风险通常和

组件影响的范围有关系，改变的数量越少，影响就越小，这是显而易见的。

MVC模式把应用程序分割成三层：模型、视图和控制器（见图1-1）。每层拥有特定的职责，而且它不需要关注其他层如何工作。

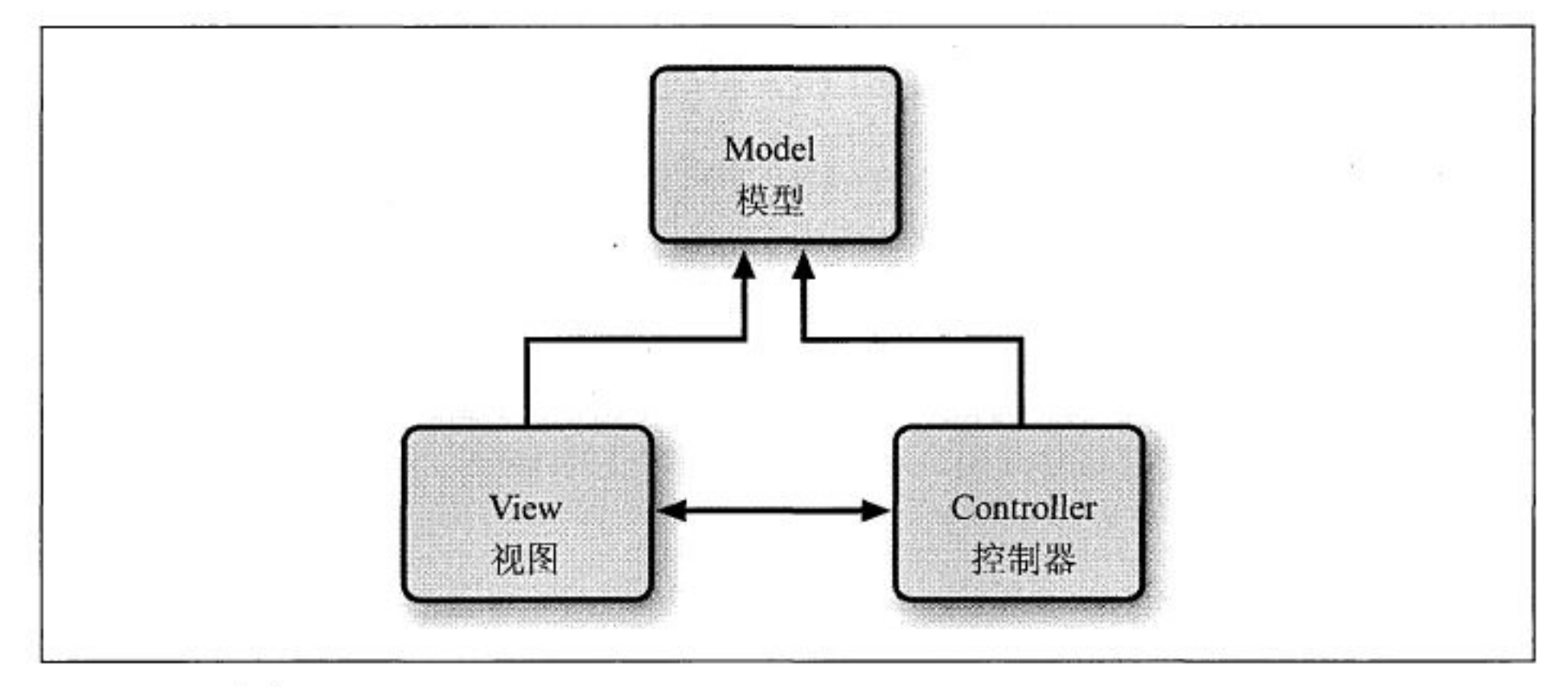


图1-1　MVC架构

模型

模型代表着核心的业务逻辑和数据。模型封装了域实体的属性和行为，并暴露出了实体的属性。例如，`Auction`类代表“拍卖”的逻辑概念，暴露了一些属性，如`Title`和`CurrentBid`，同时也包括表示行为的一些方法，如`Bid()`。

视图

视图负责转换模型并把它传递给表示层。在Web应用中，虽然视图有多种形式，但是通常指的是生成那些可以在用户浏览器中渲染的HTML代码，相同的模型可以在HTML、PDF、XML，甚至Excel电子表格里展示。

遵循“分离关注点”的原则，视图应该关注于如何**展示**数据，而不应该包含任何业务逻辑——业务逻辑封装在模型中，这些模型可以提供视图需要的任何东西。

控制器

控制器控制程序的逻辑，并且充当着视图和模型层之间协调的角色。控制器从视图层接收用户输入的信息，然后使用模型来执行特定的操作，并把最终的结果回传给视图。

ASP.NET MVC 4的新特性

What's New in ASP.NET MVC 4?

本书将会深入解析ASP.NET MVC框架，介绍此框架提供的绝大部分特性和功能。因为现在ASP.NET MVC已经到了4.0版本，除了介绍最新的特性外，本书同样会介绍整个框架的基本概念，以方便大家学习。

下面首先会对最新ASP.NET MVC 4.0版本做一个简要的介绍，然后会详细介绍这些新特性。

异步控制器

IIS会使用新线程来处理每个请求，所以每个新请求与IIS的有限可用线程息息相关，甚至包括那些空闲的线程（例如，等待数据库查询或者Web服务返回结果的线程）。在.NET 框架4.0和IIS 7中已经大幅增加了默认线程池的线程数量。尽量避免长时间独占资源仍然是最好的实践开发原则。为了更好地处理这种耗时很长的请求，ASP.NET MVC 4.0引入了异步控制器机制。通过使用异步控制器，就可以告诉框架释放处理长请求的线程，在等待期间优先去完成其他的任务。一旦这些临时任务结束，ASP.NET MVC框架就会让此线程返回到之前的长请求任务上。只要异步控制器正常执行完成，一样会返回结果，只是现在可以同时处理更多的请求！如果对异步控制器内容感兴趣，可以查看第11章的内容，那里有更深入的介绍。

显示模式

随着移动设备数量的增长，普通用户要求使用移动终端在互联网上冲浪、访问不同的网站，所以我们必须作好准备。大部分情况下，移动设备上的数据显示模式与传统的PC桌面应用的一样，除了一些特定的专为移动设备设计的图形元素以外。ASP.NET MVC显示模式提供了更便捷的针对不同移动设备的显示方式。第10章介绍了如何在自己的网站里使用这种显示模式以支持移动设备。

绑定与压缩

尽管现在可以通过更快的网速来访问互联网，但是这并不意味着可以随意开发网站而忽略客户端体验。事实上，只有完全了解各种资源的下载过程以及耗费的琐碎时间，才能理解这些问题对网站速度的负面影响。诸如JavaScript、CSS合并以及压缩或许已经不算是新鲜的概念了，但是仍然非常有用，况且，.NET Framework 4.5已经正式发布，现已成为整个ASP.NET框架的基础了。更确切地说，ASP.NET MVC封装和扩展了.NET框架的核心功能，以便于我们在ASP.NET MVC开发中更加方便地使用这些功能。第13章详细介绍了这些概念，并且演示了如何使用ASP.NET MVC 框架提供的工具。

Web API

简单的HTTP数据服务迅速成为应用程序、设备和平台提供数据的首选方式。ASP.NET MVC可以返回各种格式的数据，包括JSON和XML，而ASP.NET Web API还能简化这些步骤，相比控制器的操作方法，它能返回更加多样化的数据。第6章将会介绍如何使用AJAX客户端以及如何使用ASP.NET Web API服务实现AJAX交互。

> **你知道吗？**
>
> ASP.NET MVC是开源的！2012年4月，ASP.NET MVC、Web API以及Web Pages框架的代码都可以在CodePlex上下载了。不仅如此，开发者还可以创建自己的版本分支，甚至提交自己的代码。

EBuy介绍
Introduction to EBuy

本书不仅介绍简单的ASP.NET MVC框架概念，还将介绍如何使用这个框架进行真实的项目开发。“真实”意味着它不仅是个简单的Demo程序，还必须具有一定的复杂性。

为了能够尽量演示读者可能遇到的每个问题，本书已经收集了一个清单，包含相关开发人员最常遇到的问题和场景，以及能够从别处听到的最常见的问题。这个清单也许并不能包含开发中遇到的所有问题，但是确实可以代表大部分开发人员在使用ASP.NET MVC开发过程中遇到的大部分问题。

这不是开玩笑，真的有这么一个清单就在本书附录D里，包含了一个所有特性和场景交叉引用的列表，以及对应的章节。

为了包含列表里的全部场景，我们设计、开发了一个人人都能理解的Web应用程序——在线交易的电子商务网站，尽量接近真实的项目。

引入EBuy，使用ASP.NET MVC开发在线交易网站！从更高层次上说，这个网站相当简单：允许用户发布自己想要出售的商品，允许顾客购买自己想要的商品。假如要深入学习，将会发现程序比想象的复杂，不仅需要ASP.NET MVC，还需要集成其他的技术。

EBuy并不是一堆随书下载的示例代码。本书的每个章节不仅要介绍其特性和功能，还要使用它们来开发EBuy应用程序——从新建项目到部署项目，最好能在阅读本书的时候一起跟着编写代码。

我们必须承认的是，EBuy也是一套“代码”。事实上，可以在本书的官方网站下载这些代码。

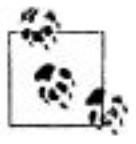
现在，让我们停止讨论这个还不存在的程序，现在就开始动手开发。

安装 ASP.NET MVC
Installing ASP.NET MVC

为了开发ASP.NET MVC应用，需要先下载并安装ASP.NET MVC 4框架。可以通过访问ASP.NET MVC官方网站，并点击“安装”按钮下载并安装ASP.NET MVC。

这样会自动启动Web Platform Installer（Web平台安装器），这款免费的工具可以简化安装过程，只需根据向导来下载并安装ASP.NET MVC 4以及它的依赖包。

注意：为了安装ASP.NET MVC 4，必须安装PowerShell 2.0和Visual Studio 2010 Service Pack 1或者Visual Web Developer Express 2010。假如没有安装，Web Platform Installer会提示，并且自动下载、安装最新版本的PowerShell和Visual Studio。



如果你使用的是老版本的ASP.NET MVC，并且想创建ASP.NET MVC 4程序，也想继续开发ASP.NET MVC 3程序，不要担心，因为ASP.NET MVC 3和ASP.NET MVC 4可以同时并存。

一旦配置好开发环境，接下来就可以进行下一步工作了：创建我们的第一个ASP.NET MVC 4应用程序。

创建ASP.NET MVC 4应用程序
Creating an ASP.NET MVC Application

ASP.NET MVC 4安装器新增了一种Visual Studio项目类型——ASP.NET MVC 4 Web Application。这是ASP.NET MVC世界的入口，我们可以使用它来创建ASP.NET MVC EBuy网上交易网站。

为了创建新项目，需要先选择Visual C#语言，然后选择ASP.NET MVC 4 Web Application项目模板，再输入网站名称“Ebuy”，如图1-2所示。

点击“OK”按钮后，就会看到另外一个带有更多选项的界面，如图1-3所示。这个新的对话框会要求定制ASP.NET MVC 4应用程序，然后Visual Studio就会创建指定的ASP.NET MVC 4网站。

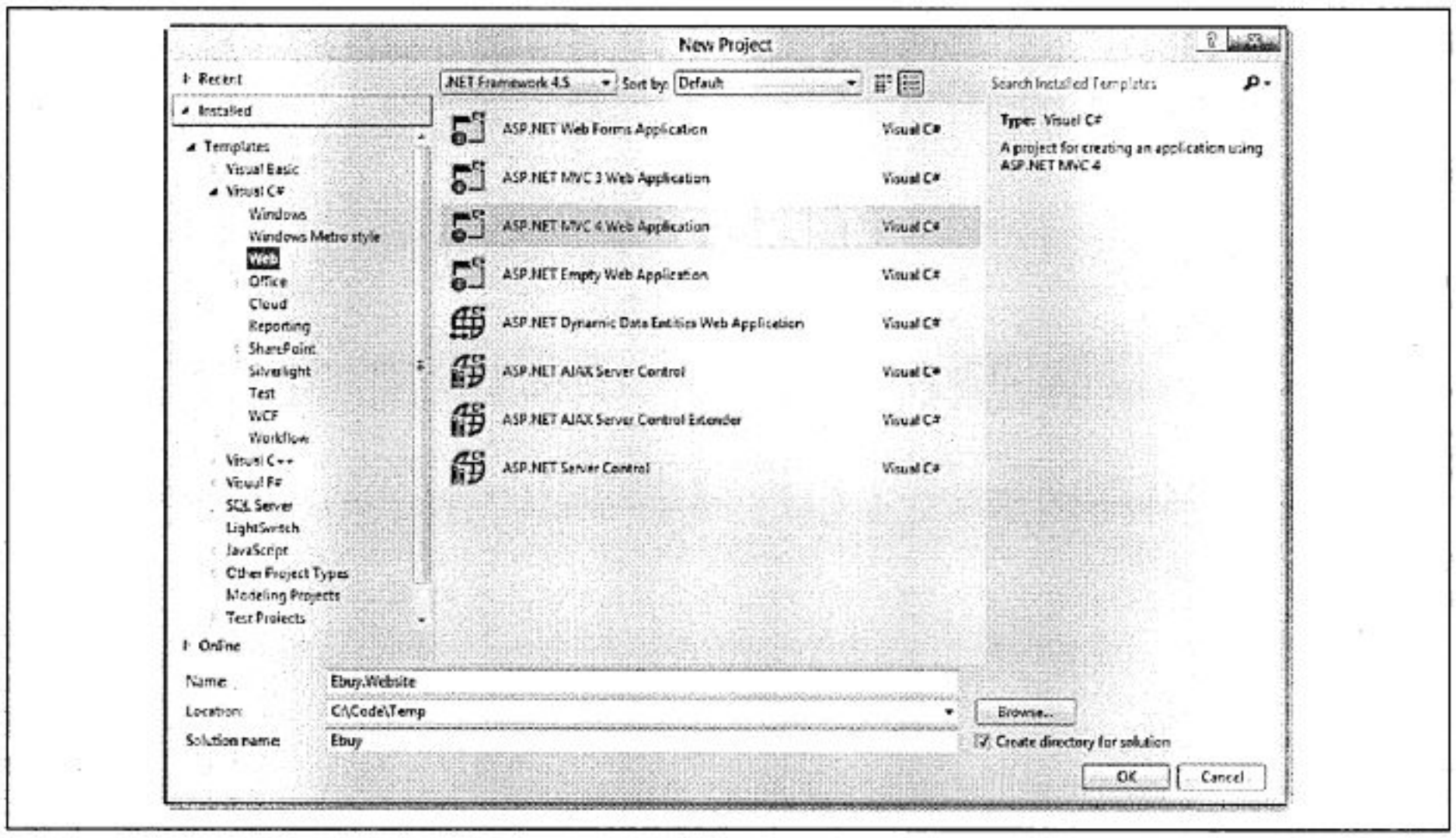


图1-2　创建EBuy项目

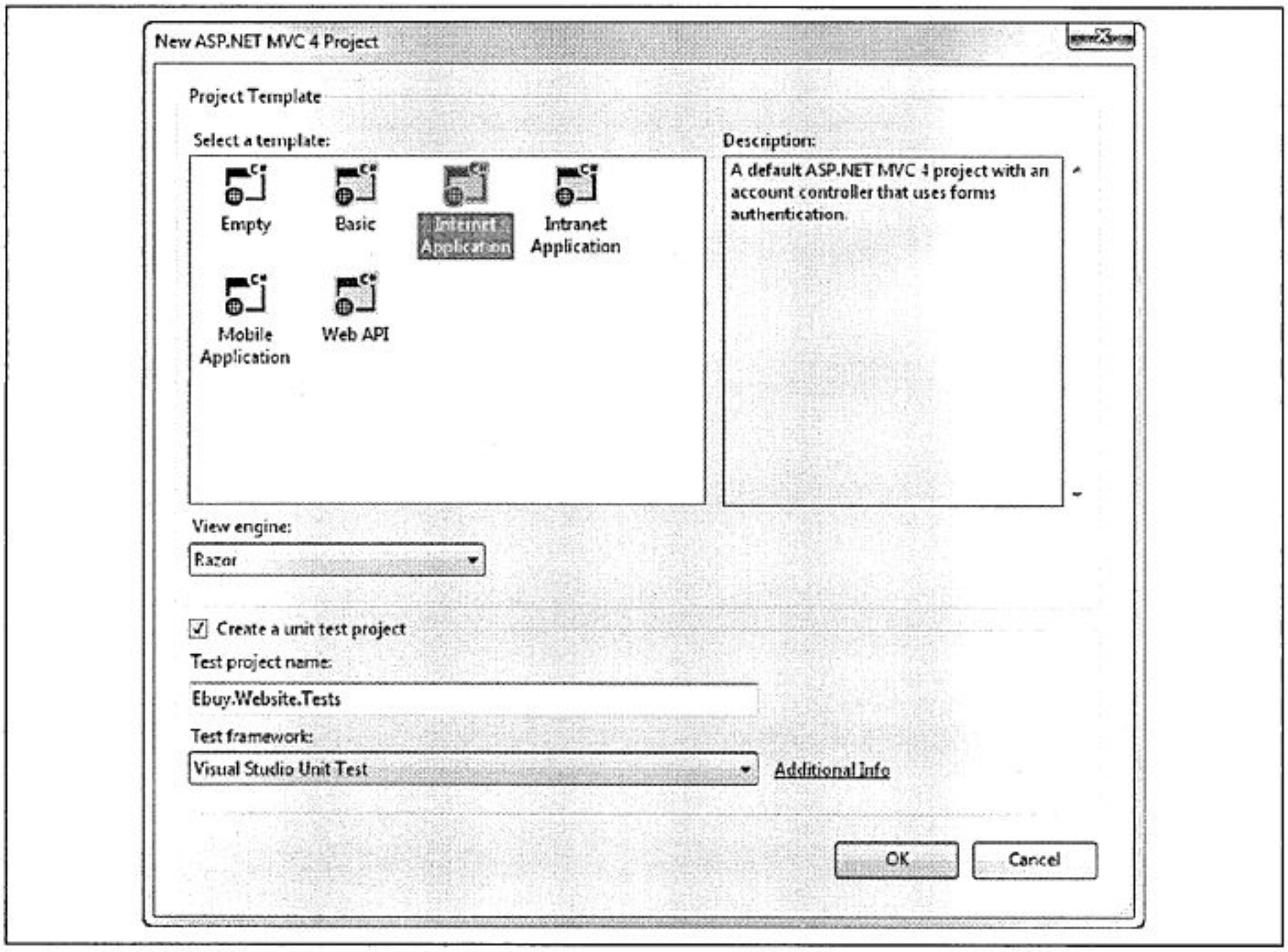


图1-3　设置EBuy项目

项目模板

ASP.NET MVC 4提供了几种不同的项目模板，用于满足不同的场景。

空模板

空模板用于创建ASP.NET MVC 4网站的架构，包含基本的文件夹结构，以及需要引用的ASP.NET MVC 程序集，也包含可能要使用的JavaScript库。模板同样包含默认的视图布局，以及标准配置代码的Global.asax文件。绝大部分ASP.NET MVC应用程序都会用到这些代码。

基本模板

011

基本模板按照ASP.NET MVC 4的规则创建了文件夹结构，包含ASP.NET MVC程序集的引用。这些模板表明了创建ASP.NET MVC 4项目所需要的最低标准的资源。现在我们可以从这个项目开始开发工作了。

互联网应用程序模板

互联网应用程序（Internet Application）模板源于空模板，它进行了扩展，包含简单的默认控制器(Home Controller)、账户控制器（Account Controller）。账户控制器包含用户注册和登录网站所需要的基本逻辑代码，以及这两个控制器需要的默认视图文件。

以太网应用程序模板

以太网应用程序(Intranet Application)模板与互联网应用程序模板很像，使用了基于Windows的验证机制，这也是企业局域网安全验证的首选机制。

移动应用程序模板

012

移动应用程序模板（Mobile Application template）是互联网应用程序（Internet Application）模板的一个变种。这个模板针对移动设备进行了优化，而且包含了jQuery Mobile JavaScript框架以及与jQuery Mobile完美兼容的视图模板。

Web API

Web API模板是互联网应用程序模板的变种，它预定义了Web API控制器。Web API是一种新的轻量级的RESTful HTTP Web服务框架，可以与ASP.NET MVC无缝集成。Web API是创建支持AJAX交互数据服务的首选，可以非常方便地用于创建这种轻量级服务。第6章详细介绍了Web API（译注1）。

新的ASP.NET MVC项目对话框可用于选择视图引擎，以及要使用的语法。我们将会使用Razor来开发EBuy网站，所以就选择默认的“Razor”选项。毫无疑问，也可以随时修改使用别的视图引擎。这个选项，只是告诉向导可以创建的视图引擎类型，而不是说把视图引擎固定死了。

译注1：我在微软的WebCast课程“WCF与Ajax开发实践系列课程”专门讲解了原始的Ajax请求、支持Ajax交互的ASP.NET Web服务以及WCF服务的原理和开发步骤，包括REST WCF服务。Web API属于REST服务，相比REST WCF服务，其开发、配置过程更加简单。

最后，选择是否要给这个解决方案创建单元测试项目。其实，不需要担心这个设置，因为与其他的Visual Studio解决方案一样，只要我们高兴，就可以给ASP.NET MVC网站程序随时添加单元测试项目。

如果对这些设置比较满意，那就点击“OK”按钮，让向导创建新的项目吧！

NuGet包管理器

如果查看Visual Studio里新项目的状态栏，就会注意到有一些消息（诸如“安装AspNetMvc...”），这是因为当前的项目正在使用NuGet包管理器去管理和安装当前项目使用的程序集。使用包管理器去管理程序依赖的概念（特别是新项目）非常有用，当然也包括ASP.NET MVC 4项目。

NuGet提供了全新的程序依赖的管理方式。开始引入ASP.NET MVC 3框架，虽然它不是ASP.NET MVC框架的一部分，但是对项目开发确实非常方便。

NuGet也许要包含程序集、内容甚至工具。在安装包的过程中，NuGet会自加载程序集到项目引用列表，添加内容到程序文件夹，并注册NuGet可能使用的工具到相应的路径。

但是，NuGet包最重要的方面实际上是依赖管理。.NET程序不是隔离的单程序集应用（大部分程序集需要引用别的程序集来进行自己的工作），更重要的是程序集往往要依赖于特定的版本（或者至少是最低版本）。

总之，NuGet会自动计算需要使用的程序集之间的关系，然后确定需要哪些程序集以及这些程序集的准确版本。

要使用NuGet，可以通过NuGet包管理器进行。有以下两种方式。

图形用户界面

NuGet包管理器带有图形用户界面，可以很方便进行查找、安装、更新以及卸载操作。通过右键点击解决方案里的项目，选择“NuGet包管理器”就可以打开NuGet用户界面。

控制台模式

类库包管理控制台是个Visual Studio窗口，它集成了PowerShell。如果在Visual Studio里找不到它，则可以通过工具→类库包管理器→包管理控制台菜单来访问。为了安装包管理控制台窗口，只需要输入`Install-Package_Package Name_`命令即可。例如，为了安装Entity Framework包，只要执行`Install-PackageEntityFramework`命令即可。包管理器控制台会自动下载EntityFramework包并安装到项目里。在“Install-Package”命令完成后，就可以在项目引用的列表里看到Entity Framework程序集。

惯例优先原则

为了简化Web开发、提升开发人员的工作效率，ASP.NET MVC依赖于“惯例优先原则”（译注2）。这意味着，ASP.NET MVC会假定开发人员遵守特定的惯例来构建自己的程序而不是使用配置文件。

图1-4所示ASP.NET MVC项目文件夹结构是“惯例优先原则”最好的例子。这里有三个元素对应于MVC架构模式：控制器、模型和视图文件夹。结构一目了然，相当清晰！

014

当仔细看这些文件夹时，就会发现更多的“惯例”。例如，不仅是控制器文件夹包含所有的控制器类，而且每个控制器类都以“Controller”结尾。整个MVC框架都使用这个惯例来注册控制器，并将它们与相应的路由器关联。

接下来，看看视图文件夹。这个惯例没这么明显，但是可以在视图文件夹内部看到“Shared”文件夹以及每个控制器对应的文件夹。这个惯例可以帮助开发人员清晰地掌握自己向用户展示的视图位置。开发人员也可以只提供一个名为“Index”的视图，这样MVC框架会尽一切能力在Views文件夹里查找它，首先在控制器指定的文件夹里找，如果找不到，再到Shared视图文件夹里查找。

015

咋一看，“惯例优先原则”的原则有点琐碎，但是，这些看似微小或者毫无意义的优化措施确实能够节约时间，改善代码的可读性以提高开发人员的效率。

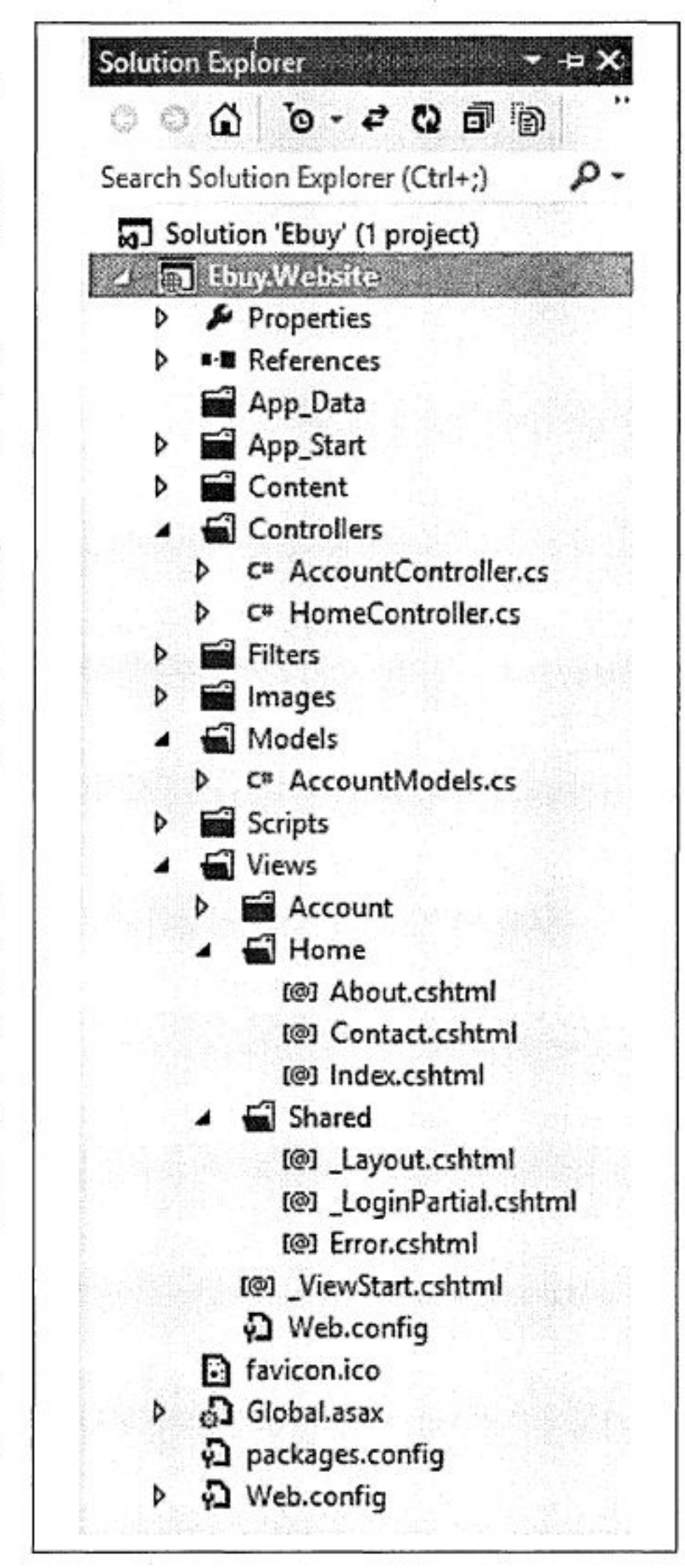


图1-4　ASP.NET MVC文件夹结构

译注2：惯例优先原则，是一种开发设计模式，指的是在项目开发过程中，尽量使用事先约定的习惯来命名，减少开发人员的疑惑，简化开发流程，而又不失灵活性。比如，模型有个类 Sales，数据库自然就有个表 sales。这就是设计开发的惯例。

运行程序

项目创建一旦完成，点击F5键就可以运行ASP.NET MVC网站了。在浏览器中可以看到它的样子。

恭喜！你已经成功创建了一个ASP.NET MVC 4的程序。

看到网页中展示的文字时，你也许非常激动。

图1-5展示了ASP.NET MVC处理请求的过程。

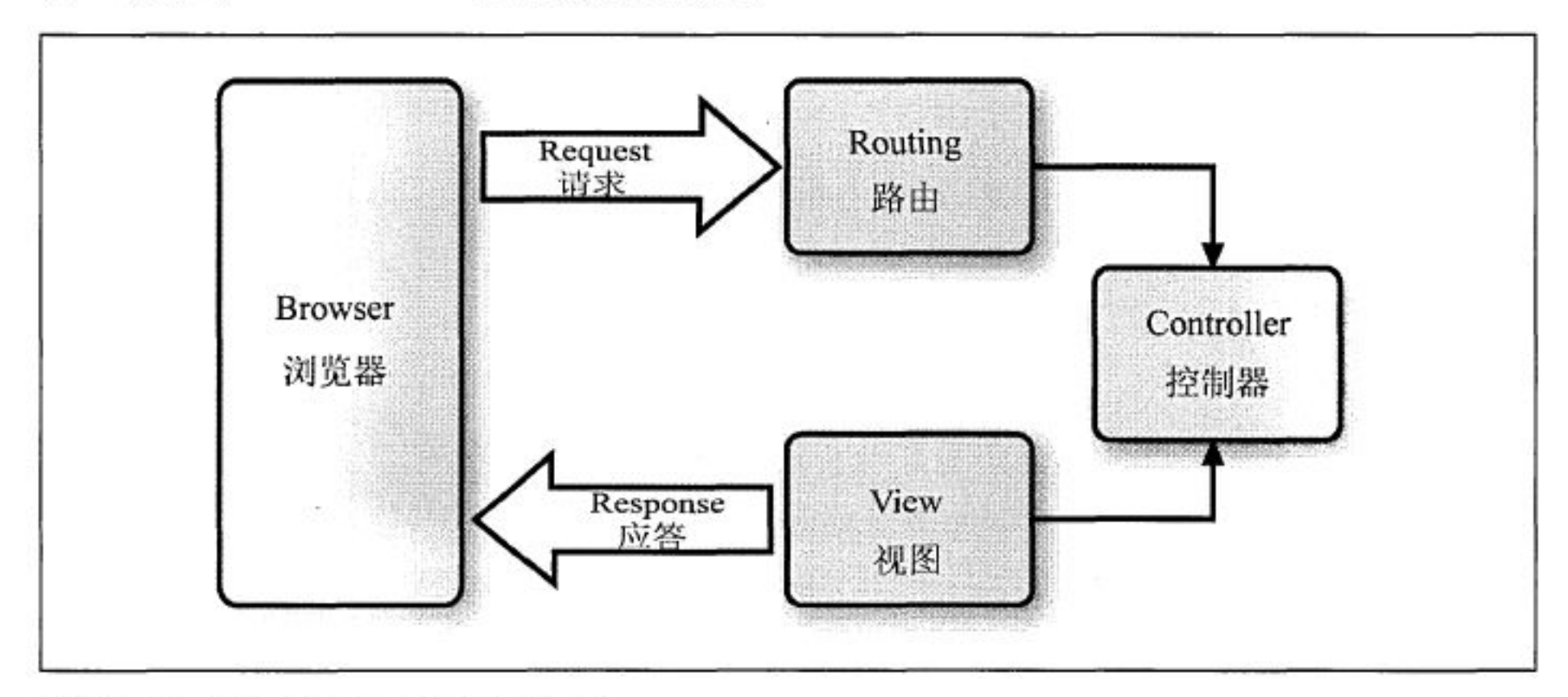


图1-5　ASP.NET MVC请求的生命周期

尽管本书的其余内容会深入讲解这个图中的组件，但是下面还是先介绍ASP.NET MVC的基本模块。

路由
Routing

所有的ASP.NET MVC请求开始与其他的网站一样，使用一个带有URL的请求。这意味着尽管没有到处提及路由，但是ASP.NET Routing仍然是ASP.NET MVC请求的核心。

简单来说，ASP.NET路由只是个模式匹配系统。开始时，应用程序使用路由表注册一种或者多种模式，告诉路由系统如何处理这些与模式匹配的请求。路由引擎在运行时接收到请求以后，它就会根据事先注册的URL模式匹配当前请求的URL，如图1-6所示。

当路由引擎在路由表里发现匹配的模式时，它就会把请求转发给特定的处理器来处理请求；如果找不到匹配的任何路由，路由引擎就不知道如何处理这个请求，就会返回404状态错误码。

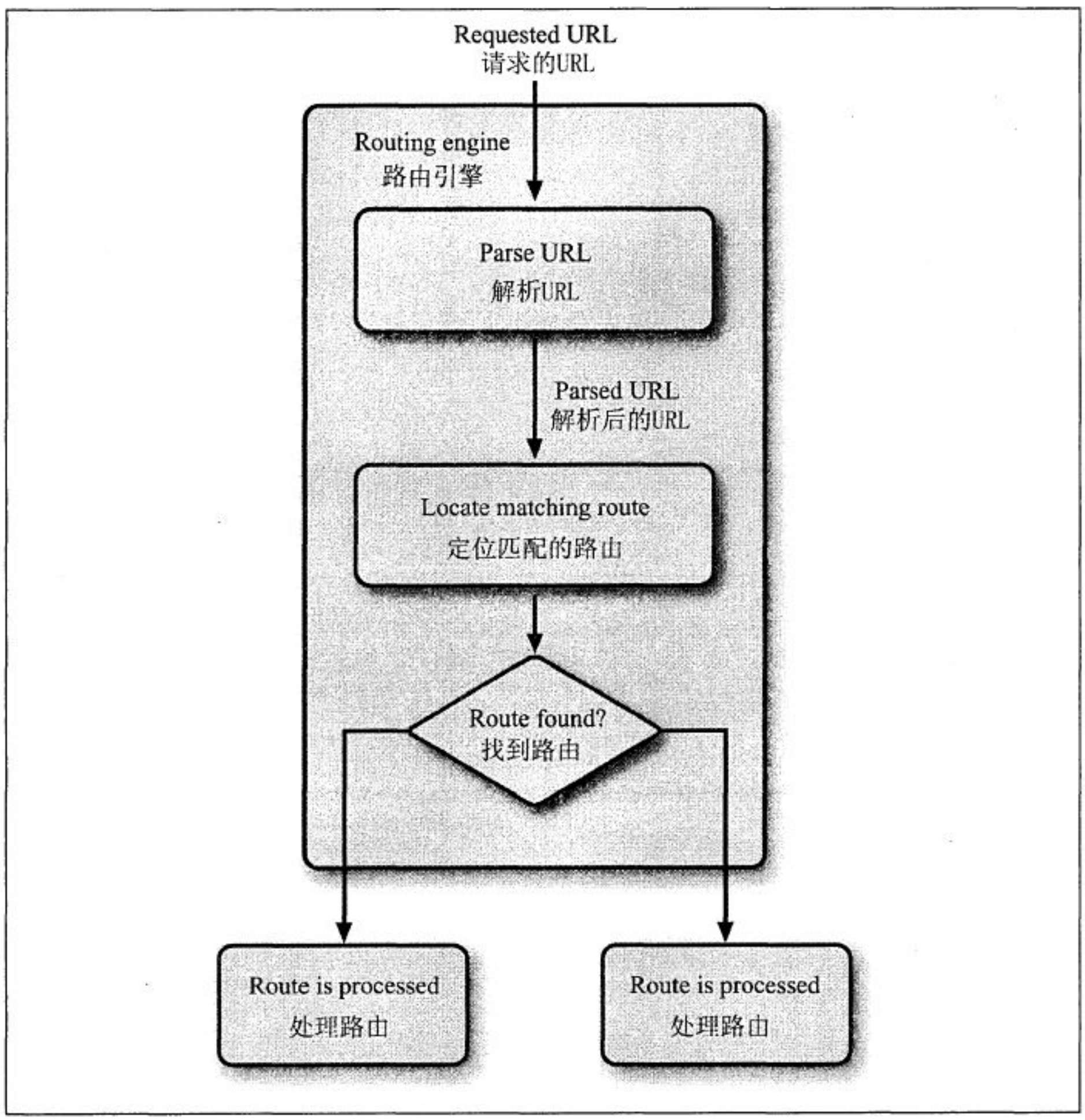


图1-6 ASP.NET路由

配置路由

ASP.NET MVC路由负责确定由哪个控制器操作来处理特定的URL请求。它由以下属性组成。

Unique name
: 路由唯一的名字。

URL pattern
: 将URL解析成有意义词语的简单模式语法。

Defaults
: URL模式里定义的参数变量的默认值。

Constraints

为URL匹配模式定义更严格的约束规则。

默认的ASP.NET MVC项目添加了一个通用的路由，它使用以下URL习惯来解析特定的URL请求，即分三个部分，包含在大括号内：controller、action、id。

```
{controller}/{action}/{id}
```

注册路由使用的扩展方法是`MapRoute()`，在程序启动的时候注册（在*App_Start/RouteConfig.cs*文件里）：

```
routes.MapRoute(
    "Default", // 路由名字
    "{controller}/{action}/{id}", // URL 参数
    new{ controller = "Home", action = "Index",
       id= UrlParameter.Optional } // 默认参数
);
```

除了name和URL pattern外，路由器同样定义了模式匹配事件中使用的一系列默认参数，但实际上并没有给每个参数提供默认值。

例如，表1-1包含了匹配这个路由的所有URL及与之对应的值。

表1-1　匹配路由模式的URL及其值

URL	Controller	Action	ID
/auctions/auction/1234	AuctionsController	Auction	1234
/auctions/recent	AuctionsController	Recent	
/auctions	AuctionsController	Index	
/	HomeController	Index	

表中的第一个URL(*/auctions/auction/1234*)完美匹配了路由模式，它符合路由模式各个部分的定义。但是，如果继续往下看这个表，逐渐移除URL的各个部分，就会发现那些URL未提供的默认值。

这是一个ASP.NET MVC如何使用“惯例优先原则”的非常重要的例子之一：当应用程序启动时，ASP.NET MVC会在程序集里查找所有可用的控制器，这些控制器类都继承自`System.Web.Mvc.IController`接口或它的子类，并且名字带有“Controller”后缀。当路由器框架确定需要访问的控制器以后，它就会去掉后缀，来获取控制器类的名称。所以，当需要使用控制器时，直接使用它的简称即可，比如`AuctionsController`指的控制器类是“`Auctions`”，而`HomeController`指的就是“`Home`”。另外值得注意的是，路由中的控制器和操作设置不区分大小写，这意味着——*/Auctions/Recent*、*/auctions/Recent*、*/auctions/recent*，甚至*/aucTionS/rEceNt*都可以解析到`AuctionsController`的`Recent`操作上。

URL路由模式对应程序根目录，所以不需要以“/”作为开始符，或者使用虚拟路径指示符“~/”。包含此字符的路由模式的三个字符串都是无效的，强制使用将会导致路由异常。

也许你已经看到了，URL路由包含了路由引擎可以提取的丰富信息。为了处理ASP.NET MVC请求，路由引擎必须能够确定两个关键信息：控制器和操作。运行时，路由引擎会把这些值传递给ASP.NET MVC去创建和执行特定的控制器和操作。

控制器
Controllers

在MVC架构模式的上下文里，控制器响应用户的输入（比如，用户点击“保存”按钮），并协调模型、视图以及（经常）数据访问层。在ASP.NET MVC程序里，控制器就是包含被路由框架处理请求时调用的方法的类。

下面看看ASP.NET MVC控制器的例子`HomeController`类，位于Controllers/ HomeController.cs里：

```
using System.Web.Mvc;

namespace Ebuy.Website.Controllers
{
    public class HomeController : Controller
    {
        public ActionResult Index()
        {
            ViewBag.Message = "Your app description page.";

            return View();
        }

        public ActionResult About()
        {
            ViewBag.Message = "Your quintessential app description page.";

            return View();
        }

        public ActionResult Contact()
        {
            ViewBag.Message = "Your quintessential contact page.";

            return View();
        }
    }
}
```

控制器操作

控制器类并无特别之处，跟别的.NET类几乎没什么区别。事实上，控制器类里的方法（称为控制器的操作）做了处理请求过程中的主要工作。

经常听到控制器和控制器操作这种词语，在本书里也这样称呼，其实MVC模型并不会区分二者。但是ASP.NET MVC框架十分关注控制器操作，因为它包含了处理请求的实际逻辑代码。

例如，HomeController类包含三个操作：Index、About和Contact。然而，假设默认的路由模式是{controller}/{action}/{id}，当一个请求的URL是/Home/About时，路由框架会决定由HomeController类中的About方法来处理这个请求。随后，ASP.NET MVC框架会创建HomeController的实例，并执行About()方法。

这个例子中，About()方法非常简单：它通过ViewBag属性把数据传递给视图（后面会详细介绍），然后ASP.NET MVC框架通过调用View()方法来显示名为“About”的视图，这个操作返回一个ViewResult类型的操作结果。

操作结果

值得注意的是，控制器的工作就是告诉ASP.NET MVC框架下一步应该做什么，而不是怎么做。这个沟通过程通过使用+ActionResult+s来实现，返回值就是控制器提供的操作。

例如，当控制器决定如何显示视图时，它就会告诉ASP.NET MVC框架通过返回ViewResult来展示视图，而不会自己渲染视图。这种松耦合的设计，也是操作中“分离关注点”原则的直接体现（应该干什么和怎么干）。

尽管每个控制器的操作都要返回ActionResult，但是大部分时间不需要你手动完成。相反，只需要使用System.Web.Mvc.Controller基类提供的帮助方法即可，例如：

020

Content()
: 返回文本类型的ContentResult，比如“Hello, world!”。

File()
: 返回文件类型的内容FileResult，比如PDF。

HttpNotFound()
: 返回包含404HTTP状态码的HttpNotFoundResult。

JavaScript()：：返回 JavaScriptResult
: 返回包含JavaScript内容的JavaScriptResult，比如“function hello() { alert(Hello, World!); }”。

Json()
: 返回JSON格式数据的JsonResult，比如“{ “Message” : Hello, World! }”。

PartialView()
: 返回包含部分视图内容的PartialViewResult（例如，视图可能不包含结局）。

Redirect()
: 返回一个包含302跳转状态值RedirectResult，跳转到给定的URL上。例如，“302 *http://www.ebuy.com/auctions/recent*”。这个方法包含一个同级别的方法RedirectPermanent()，它同样返回RedirectResult，但是使用的是301状态码去指示一个永久的跳转地址，而不是临时地址。

RedirectToAction()和RedirectToRoute()

与Redirect()类似，只有框架可以动态查询路由引擎来确定外部的URL。与Redirect()一样，它们同样包含永久跳转方法：RedirectToActionPermanent()和RedirectToRoutePermanent()。

View()

返回渲染视图的ViewResult。

综上所述，MVC框架提供了各种情况下需要的操作结果类型，我们可以自由决定使用哪种类型。

虽然所有的操作都要求提供操作结果以决定下一步请求的处理操作，但并不是所有的控制器操作都需要制定它们的返回类型。控制器操作可以指定任何ActionResult子类作为返回类型，甚至别的任意类型。

当ASP.NET MVC框架遇到一个非ActionResult类型作为返回控制器操作结果时，会自动在ContentResult里进行包装并渲染原始的内容。

操作参数

控制器操作，当单独来看时，和别的方法没什么两样。事实上，当执行操作时，控制器操作甚至可以使用ASP.NET MVC请求消息发送过来的参数。这种功能称为“模型绑定”，而且也是ASP.NET MVC最强大、最有用的特性。

在深入学习模型绑定之前，先来回顾传统的请求传值方式（译注3）：

```
public ActionResult Create()
{
   Var auction = new Auction() {
      Title = Request["title"],
      CurrentPrice = Decimal.Parse(Request["currentPrice"]),
      StartTime = DateTime.Parse(Request["startTime"]),
      EndTime = DateTime.Parse(Request["endTime"]),
   };
   // ...
}
```

在这个特定的例子里，控制器的操作使用请求传递的值来实例化Auction对象。因为Auction的属性类型并非是基元类型String，所以操作代码还需要尽量进行类型转换。

这个例子看起来简单，但是却十分脆弱：只要任何一种类型解析失败，整个操作就会失败。如果使用TryParse()方法也许可以避免大部分异常，但是这意味着要使用更多额外的代码。

译注3：传统的ASP.NET传值方式包括URL传值、Session传值、视图状态传值以及数据库、文件传值等。

这种方法的副作用非常明显：这种代码把责任丢给了你，一个“苦逼”的程序员去做所有工作，并且在每次工作的时候都要执行必要的检查。大量的代码也许会掩盖操作的真正目的，在这个例子里，只要增加一个新的Auction交易对象到系统里。

模型绑定的基本概念

模型绑定不仅可以避免使用大量显式代码，而且使用方便。
例如，和之前的控制器操作一样，以下方法使用了模型绑定方法参数：

```
public ActionResult Create(
     string title, decimal currentPrice,
     DateTime startTime, DateTime endTime
  )
{
  var auction = new Auction() {
     Title = title,
     CurrentPrice = currentPrice,
     StartTime = startTime,
     EndTime = endTime,
  };
  // ...
}
```

现在，就不需要从Request对象里解析数据了，因为操作方法已经声明了这些值作为参数。当ASP.NET MVC框架执行这个方法时，它会自动给操作的方法赋值。注意：尽管我们没有直接访问Request的字典，但这些参数仍然十分重要，因为它们仍然与Request对象里的值一一对应。

Request对象并不是ASP.NET MVC模型绑定器获取值的唯一方式。事实上，MVC框架可从几个地方查找需要的值，比如，路由数据、查询字符串参数、Post提交的数据值，甚至一些序列化的JSON对象中。例如，下面一段代码实现了如何通过声明相同的名字从URL取值：

例1-1　从URL里取得id的值

```
public ActionResult Auction(long id)
{
   var context = new EBuyContext();
   var auction = context.Auctions.FirstOrDefault(x =>x.Id == id);
   return View("Auction", auction);
}
```

ASP.NET MVC模型绑定器在哪里查找以及如何查找值都是可以配置和扩展的。第8章就ASP.NET MVC的模型绑定进行了深入的讨论。

正如例子演示的，模型绑定让ASP.NET MVC处理了大量烦琐的基础工作，让操作代码可以专注于提供业务逻辑上。剩下的代码不仅更加有意义，而且更加具备可读性。

模型绑定复杂对象

使用模型绑定方法，不管是不是简单的基元类型，都可以让代码更加简洁、直观。但在现实

世界中，事情往往非常复杂，只有简单的场景才只需要简简单单的几个参数。

幸运的是，ASP.NET MVC同样支持绑定复杂的数据类型。

下面这个例子的Create操作需要传递一个参数，这次直接使用了Auction对象：

```
public ActionResult Create(Auction auction)
{
    // ...
}
```

这个代码与之前的例子一样。没错，ASP.NET MVC的复杂数据绑定机制去除了创建Auction对象需要的烦琐代码。这个例子展示了模型绑定的真正优势。

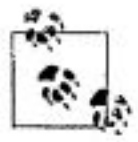
操作过滤器

操作过滤器通过提供更加简单强大的机制去修改或者增强ASP.NET MVC管道，在特定的点注入逻辑，帮助处理贯穿程序中各个模块的横切关注点(cross-cutting concerns)（译注4）问题。应用程序日志就是典型的横切关注点的例子——无论组件的主要职责是什么，日志贯穿于应用程序的任何模块。

操作过滤器逻辑首先是通过将ActionFilterAttribute属性标记到操作上来影响操作的执行。正如下面的例子里所设置的，它可以保护任何未授权的访问操作：

```
[Authorize]
public ActionResult Profile()
{
    // 为当前用户查询个人信息
    returnView();

}
```


ASP.NET MVC框架包含一些特定场景的操作过滤器。本书将介绍这些操作过滤器，以帮助我们使用更加简洁、松耦合的方式来完成不同的工作。



操作过滤器确实是在网站中应用定制化逻辑的很好的方式。记住，可以通过扩展ActionFilterAttribute或任意一个ASP.NET MVC过滤器来创建自己的操作过滤器。

译注4：横切关注点(cross-cutting concerns)和AOP方面的编程有关。在传统的软件开发过程中，除了竖向分布的关注点模块以外，还有些关注点的实现会弥散在整个软件内部，这时这些关注点同主关注点是横切的，比如日志、异常处理以及验证授权等。

视图
Views

在ASP.NET MVC框架中，想要返回给用户HTML的控制器操作，就要返回ActionResult类型的ViewResult实例，ActionResult知道如何渲染应答结果。当渲染视图时，ASP.NET MVC将会使用控制器提供的名字。

以HomeController的Index操作为例，如下：

```
public ActionResult Index()
{
    ViewBag.Message = "Your app description page.";
    returnView();
}
```

这个操作将会使用帮助View()来创建ViewResult。无参调用View()，与例子里一样，ASP.NET MVC会找到一个和当前操作一样的视图名字。在这个例子里，ASP.NET MVC将会查找名为“Index”的视图，但是去哪里找它呢？

定位视图

ASP.NET MVC依赖惯例是在网站根目录下面的Views文件夹查找这个视图文件。更确切地说，ASP.NET MVC希望视图文件放在以它们对应的控制器名字命名的文件夹中。

因此，如果MVC框架想为HomeController的Index操作显示视图，那么它就要在/Views/Home文件夹下查找名为Index的文件。图1-7展示了项目模板截图，已经包含了一个Index.cshtml的视图文件（译注5）。

如果在Views文件夹下没有找到对应的视图文件，ASP.NET MVC框架就会继续在/Views/Shared文件夹里查找。

/Views/Shared文件夹是保存共享多个控制器视图的绝佳位置。

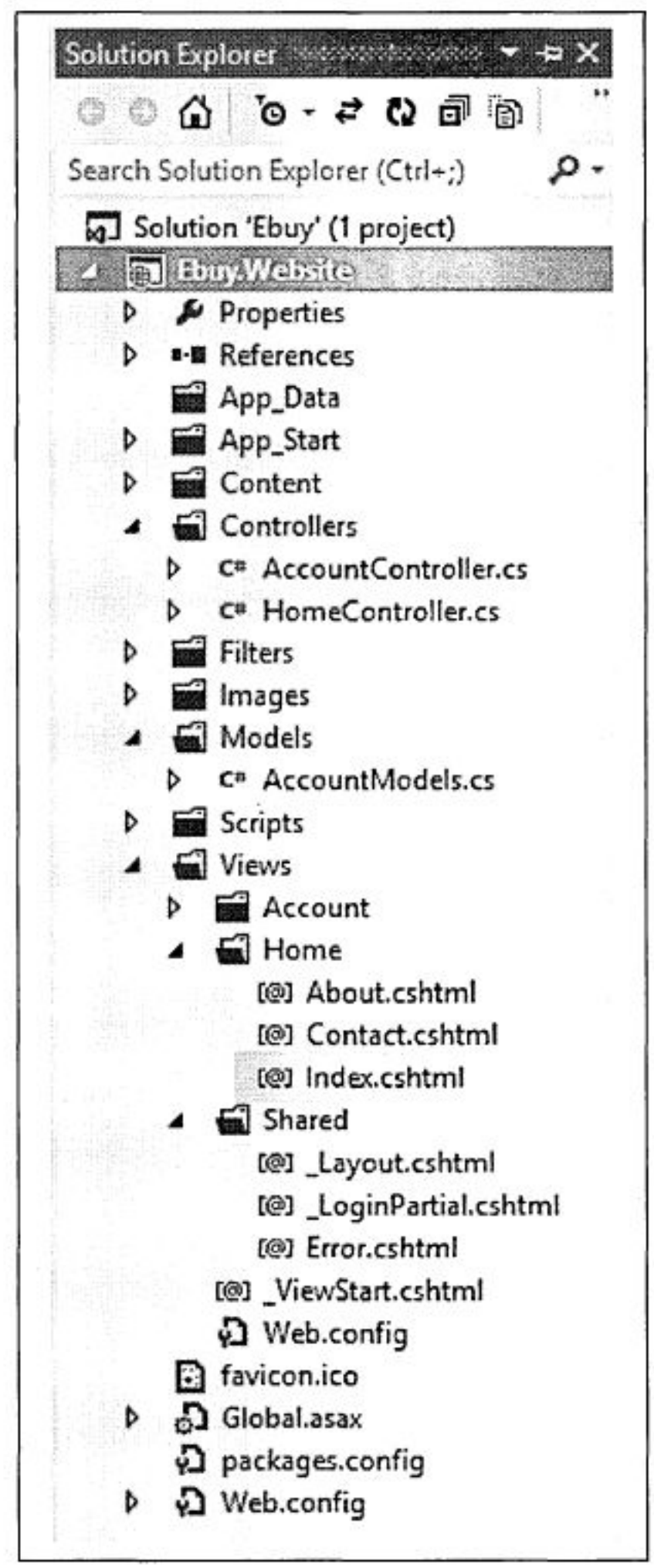


图1-7 查找Index视图文件

译注5：这里的跨越多个控制器的视图文件是指诸如网站页面的顶部导航栏、底部声明信息等共享部分的页面内容。

既然已经知道了操作请求的视图，现在就来看看视图文件包含什么内容：HTML标签和代码。但它并非仅仅是HTML标签和代码——它是Razor!

026

Razor，你好！

Razor是一种允许把代码和内容进行平滑集成的语法。尽管它引入了一些新的符号和关键字，但是Razor并不是一种新的语法。相反，Razor允许用户使用已知的语言来编写代码，比如C#或者VB.NET。

Razor的学习门槛很低，因为它允许使用已经具备的技能，而不要求学习一种新的语言。因此，如果知道如何编写HTML或者使用C#、VB.NET编写代码，就可以轻易编写下面类似的代码：

```
<div>This page rendered at @DateTime.Now</div>
```

这些代码会输出：

```
<div>This page rendered at 12/7/1941 7:38:00 AM</div>
```

这个例子以HTML标签(`<div>`)开头，然后是一些“硬编码”文本，然后就是动态内容，引用了.NET的时间类型(`System.DateTime.Now`)，最后是HTML结束符(`</div>`)。

Razor智能感知解析器允许用户编写更复杂的逻辑代码，而且可以在代码和标签之间轻易转换。虽然Razor语法与其他的标记语法不同（比如ASP.NET Web Form），但是它们的目标都是相同的，就是渲染HTML。

为了说明这一点，现在来看下面的例子：在Razor和ASP.NET Web Form标签里实现同样的功能。

这里是一个使用`if/else`语句的Web Form语法例子：

```
<% if(User.IsAuthenticated) { %>
    <span>Hello, <%: User.Username %>!</span>
<% } %>
<% else { %>
    <span>Please <%: Html.ActionLink("log in") %></span>
<% } %>
```

而使用Razor语法的代码为：

```
@if(User.IsAuthenticated) {
    <span>Hello, @User.Username!</span>
} else {
    <span>Please @Html.ActionLink("log in")</span>
}
```

以下这个`foreach`循环使用了WebFrom的语法：

```
<ul>
<% foreach( var auction in auctions) { %>
```

```
    <li><a href="<%: auction.Href %>"><%: auction.Title %></a></li>
<% } %>
</ul>
```

使用Razor语法的代码如下：

```
<ul>
@foreach( var auction in auctions) {
    <li><a href="@auction.Href">@auction.Title</a></li>
}
</ul>
```

虽然使用了不同的语法，但是两个例子的代码都是渲染相同的HTML。

区分代码和标记语言

Razor提供了两种不同的方式区分代码和标签：代码段和代码块。

代码段

代码段是一些简单的表达式，它们可以在一行中进行渲染，也可以与文本混合，例如：

```
Not Logged In: @Html.ActionLink("Login", "Login")
```

表达式跟在@之后，Razor能够智能确定左括号的结束部分。上面例子将会渲染以下的输出结果：

```
Not Logged In: <a href="/Login">Login</a>
```

注意：这行代码必须一直返回标记代码给视图渲染。如果编写的代码段返回了void，这会在执行视图的时候出错。

代码块

代码块是一段包含代码的视图，它只包含代码，而不是代码和标记语言的混合。Razor定义的代码块要求使用“@{}”包装。“@{”标记开始，中间不确定行数，以“}”结尾。

记住，代码块里的代码与代码段里的代码不同，前者是常规代码，必须符合当前语言的语法。例如，每行C#代码必须以“;”结尾，这与在.CS文件里编写C#类的代码一样。

下面是一个典型的代码块例子：

```
@{
 LayoutPage = "~/Views/Shared/_Layout.cshtml";
 View.Title = "Auction "+ Model.Title;
}
```

代码块不能渲染任何东西。相反，它允许用户编写任意没有返回值的代码。

同样，代码块里定义的变量可能被同一个域中的代码段使用。像foreach循环体里定义的变

量只能被容器内的代码访问，而定义在视图顶部的变量可以被相同视图中的代码块和代码段访问。

为了更好地说明这个问题，先看以下包含多个变量的视图文件：

```
@{
    // 整个视图都可以访问 title 和 bids 变量
    var title = Model.Title;
    var bids = Model.Bids;
}

<h1>@title<h1>
<div class="items">
<!--遍历 bids 变量中的对象集合-->
@foreach(var bid in bids) {
    <!--bid 变量只有在 foreach 循环中可用-->
    <div class="bid">
        <spanclass="bidder">@bid.Username</span>
        <spanclass="amount">@bid.Amount</span>
    </div>
}

<!--将会抛出错误：当前范围内不存在 bid 变量-->
<div>Last Bid Amount: @bid.Amount</div>
</div>
```

代码块是一种可以在模板中执行代码但是又不会返回任何值给视图的方式。与代码段不同的是，它不需要返回值，视图会完全忽略代码块的返回值。

布局

Razor通过layouts维护网站外观布局的一致性。使用布局，单个视图定义了整个网站的布局和样式，就像模板一样被其他视图使用。

布局模板包含基本的标签（Scripts、CSS以及诸如导航和内容容器等HTML元素），可以指定渲染视图内容的位置。网站中的每个视图可以引用这个布局，只包括内容、位置符合布局里约定的视图（译注6）。

Razor的基本布局文件（*_Layout.cshtml*）如下：


```
<!DOCTYPE html>

<html lang="en">
    <head>
        <meta charset="utf-8" />
        <title>@View.Title</title>
```

译注6：布局机制，类似于 ASP.NET Web Form 里的 Master Page 母版页机制，可以定义网站全局的布局样式，每个页面引用嵌套在这个全局的布局中。

```
    </head>
    <body>
        <div class="header">
            @RenderSection("Header")
        </div>

        @RenderBody()

        <div class="footer">
            @RenderSection("Footer")
        </div>
    </body>
</html>
```

布局文件包含HTML内容，它定义了整个网站的HTML结构。依赖于一些变量（如`@View.Title`）和帮助函数（如`@RenderSection([Section Name])`和`@RenderBody()`）来进行单个的视图交互。一旦Razor布局定义完成，视图引用当前布局以后就可以定义自己的段落内容。

下面的视图页面引用了前面的布局文件_Layout.cshtml：

```
@{ Layout = "~/_Layout.cshtml"; }

@section Header {
    <h1>EBuy Online Auction Site<h1>
}

@section Footer {
    Copyright @DateTime.Now.Year
}

<div class="main">
   This is the main content.
</div>
```

Razor布局和内容视图像批萨一样组合在一起，每一块定义了页面的特定部分。当所有的小块组合在一起时，就可以展示出一个完整的页面。

部分视图

虽然布局通过重用部分HTML代码提供了维护网站外观布局一致性的方法，但是某些场景还是无法处理的，这就需要一些特定的方法。

最常见的场景就是在网站中的多个位置显示高级别的信息。

例如，EBuy交易网站可能要显示一个交易列表——只显示交易的名称、当前价格和一段摘要，而且可能需要在网站的多个页面显示，如搜索页面、网站的主页。

ASP.NET MVC通过部分视图可以支持这些场景。

部分视图就是用来显示特定标签代码的小视图，它们属于大视图的一部分。下面的代码演示了上面提到的交易列表的部分视图代码结构：

```
@model Auction

<div class="auction">
    <a href="@Model.Url">
         <img src="@Model.ImageUrl" />
     </a>
    <h4><a href="@Model.Url">@Model.Title</a></h4>
    <p>Current Price: @Model.CurrentPrice</p>
</div>
```

为了以后使用这个部分视图，可以直接保存为一个单独的视图文件(例如，/Views/Shared/Auction.cshtml)，然后用ASP.NET MVC自带的HTML帮助方法（这个方法可以把部分视图作为另外一个视图的部分内容进行渲染）来调用它。

下面的代码会在Auction集合里进行迭代，然后使用部分视图渲染每个交易信息：

```
@model IEnumerable<Auction>

<h2>Search Results</h2>

@foreach(var auction in Model) {
     @Html.Partial("Auction", auction)
}
```

注意，`Html.Partial()`帮助方法的第一个参数是一个视图的名字，并不包含扩展文件名后缀。这是因为`Html.Partial()`帮助方法的基础是强大的ASP.NET MVC视图引擎。渲染部分视图的方式与控制器中操作方法调用`View()`方法返回视图结果的过程十分类似。引擎使用视图名称去查找并渲染特定的视图。

同样，部分视图的查找和执行过程与`View()`方法的十分相似。唯一的区别就是部分视图表示的是大视图的一部分内容。

与`View(_View Name_、_[Model]_)`方法一样，`Html.Partial()`帮助方法的第二个参数接受的是部分视图的模型。这个参数不是必须的，如果不指定，就使用调用`Html.Partial()`的视图的模型。例如，如果第二个参数没传递，则ASP.NET MVC会传递原来View视图的`Model`属性(`IEnumerable<Auction>`)。

上面的例子展示了如何重用部分视图以减少网站的重复代码以及复杂性。部分渲染十分有用，它也是唯一一种使用部分视图的方法。后面将介绍如何使用部分视图与Ajax进行交互。

显示数据

MVC架构依赖于模型、视图和控制器，虽然彼此分离，但是确实同心协力工作在一起。这种关系中，控制器充当了“交警”的角色。协调系统的不同部分执行不同的应用处理逻辑。这些处理过程可能会返回一些特定的数据给用户。但是，显示数据的工作却不属于控制器，这

是视图的职责。那么，控制器如何与视图交互呢？

ASP.NET MVC提供了两种方式在MVC边界之间传递数据：ViewData和TempData。所以，从控制器向视图传递数据就变得非常简单，只需要在控制器里赋值即可。下面的代码来自HomeController.cs：

```
public ActionResult About()
{
    ViewData["Username"] = User.Identity.Username;

    ViewData["CompanyName"] = "EBuy: The ASP.NET MVC Demo Site";
    ViewData["CompanyDescription"] =
        "EBuy is the world leader in ASP.NET MVC demoing!";

    return View("About");
}
```

在视图文件中使用这个值，如About.cshtml文件里的代码：

```
<h1>@ViewData["CompanyName"]</h1>
<div>@ViewData["CompanyDescription"]</div>
```


通过ViewBag访问ViewData的值

暴露ViewData属性的ASP.NET MVC控制器和视图同样暴露了一个叫做ViewBag相似的属性。ViewBag只是把暴露ViewData简单包装成一个dynamic（译注7）动态对象。

例如，任何引用ViewData字典取值的代码都可以修改成使用ViewBag对象的属性的方式，如下：

```
public ActionResult About()
{
    ViewBag.Username = User.Identity.Username;

    ViewBag.CompanyName = "EBuy: The ASP.NET MVC Demo Site";
    ViewBag.CompanyDescription = "EBuy is the world leader in ASP.NET MVC demoing!";

    return View("About");
}
```

和：

```
<h1>@ViewBag.CompanyName</h1>
<div>@ViewBag.CompanyDescription</div>
```

译注7：dynamic 类型是.NET 4.0 引入的新类型。与 var 不同的是，它不会在编译时检查类型，而是绕过编译时的类型检查，改为在运行时解析这些操作。

视图模型

除了基本的字典行为，ViewData对象也提供了Model属性，这是请求的原始模型对象。虽然ViewData.Model与ViewData["Model"]一样，但是它提升了模型对象的级别为一级，认为比其他数据更重要。

例如，前面两段代码里的CompanyName和CompanyDescription关系密切，完全可以封装在一个模型对象里。

看以下CompanyInfo.cs文件:

```
public class CompanyInfo
{
    public string Name { get; set; }
    public string Description { get; set; }
}
```

HomeController.cs中的About操作代码如下：

```
public ActionResult About()
{
    ViewBag.Username = User.Identity.Username;

    var company = new CompanyInfo {
        Name = "EBuy: The ASP.NET MVC Demo Site",
        Description = "EBuy is the world leader in ASP.NET MVC demoing!",
    };

    returnView("About", company);
}
```


下面的代码来自About.cshtml:

```
@{ var company = (CompanyInfo)ViewData.Model; }

<h1>@company.Name</h1>
<div>@company.Description</div>
```

这段代码里，使用的CompanyName和CompanyDescription字典值都已经封装到一个名为CompanyInfo (公司信息)的新类中。HomeController.cs的代码也在操作中使用了View()的重载方法。重载方法保留了第一个参数视图名，第二个参数表示要赋值给ViewData.Model属性的对象。

现在，company对象直接作为模型参数传递给View()方法，视图(About.cshtml)可以从company对象里获取它的值。

强类型视图

在默认情况下，Model属性可以在Razor视图里访问，并且是动态类型，这意味着可以直接访问这个类型，而不需要知道它的准确类型。但是，考虑到C#语言的静态属性以及Visual Studio

对Razor视图强大的智能感知支持，最好还是明确指定页面模型的具体类型。

幸运的是，Razor让一切变得简单——直接使用@model就可以指定Model的类型：

```
@model Auction

<h1>@Model.Name</h1>
<div>@Model.Description</div>
```

这个例子修改了之前的“Auction.cshtml”例子，省略了添加ViewData.Model转换的中间变量。相反，第一行使用了@model关键字告诉模型类型是CompanyInfo，这让所有ViewData.Model模型引用变成强类型，而且可以直接访问。

HTML和URL帮助方法

绝大部分Web请求的目标都是向用户发送HTML代码。正因为如此，ASP.NET MVC会尽力帮助你去创建HTML。除了Razor标记语言，ASP.NET MVC也提供了很多生成HTML代码的简单、有效的方式。最重要的两个帮助类就是HtmlHelper类和UrlHelper类，作为控制器和视图的Html和Url属性暴露出来，供大家使用。

这里是一些帮助类的例子：

```
<img src='@Url.Content("~/Content/images/header.jpg")' />
@Html.ActionLink("Homepage", "Index", "Home")
```

渲染的HTML代码如下：

```
<img src='/vdir/Content/images/header.jpg' />
<a href="/vdir/Home/Index">Homepage</a>
```

一般而言，HtmlHelper类和UrlHelper类并没有几个自己的方法，它们只是扩展了框架的一些附加行为。这使得它们成为重要的扩展点，将会在本书的其他内容中看到它们。

尽管有太多的方法无法一一列举，但是这里还是要强调一件事情：HtmlHelper类帮助我们生成HTML标记代码，而UrlHelper帮助我们生成URL地址。记住，当需要生成HTML或者URL的时候就使用它们。

模型

Models

前面已经学习了控制器和视图，下面介绍MVC架构中模型的概念，它被认为是MVC架构中最重要的部分。如果它真的如此重要，可为什么偏偏最后才介绍呢？

因为模型层是最复杂的，它往往包含了整个程序的所有业务逻辑，而且每个程序的业务逻辑千差万别。

从技术角度来看，模型由属性暴露数据、方法封装业务逻辑的类组成。这些类大小各异，最常见的例子就是“数据模型”或者“域模型”，它们的首要职责就是负责管理数据。

例如，下面的代码展示了Auction类——这个模型将会驱动整个EBuy网站：

```
public class Auction
{
    public long Id { get; set; }
    public string Title { get; set; }
    public string Description { get; set; }
    public decimal StartPrice { get; set; }
    public decimal CurrentPrice { get; set; }
    public DateTime StartTime { get; set; }
    public DateTime EndTime { get; set; }
}
```


在本书中，我们会给Auction类添加各种各样的功能和行为，比如验证等，这段代码完全可以表示“Auction”模型类。

正如会在本书中逐步完善Auction类一样，还会使用其他的类（比如服务类和帮助类）来编写MVC架构中的模型类。

群英荟萃

Putting It All Together

目前为止，已经介绍了ASP.NET MVC的所有组成部分，但仅仅是停留在Visual Studio使用项目模板创建的代码上。换句话说，实际上还没有做任何事情。所以，现在就行动起来。

下面一段将关注与实现这样一个例子：显示一条交易信息。回顾一下，每个ASP.NET MVC请求都至少需要包含三样东西：路由、控制器和视图（模型并不是必须的）。

路由

要弄清楚你想要为某项功能定义的路由模式，首先就要知道自己希望的那个功能URL的样子。这个例子里，将会选择一个相对标准的URLAuctions/Details/[Auction ID]，例如，*http://www.ebuy.biz/Auctions/Details/1234*。

太神奇了——默认的路由已经支持这个URL了。

控制器

接下来就要创建包含处理请求的操作方法的控制器。

因为控制器只是实现了ASP.NET MVC控制器接口的类，所以也可以向“Controllers”文件夹添加继承了System.Web.Mvc. Controller的新类，然后在新类中添加控制器操作代码。然而，Visual Studio提供的向导工具完成了大部分创建控制器的工作：仅仅需要用右键点击“Controllers”文件夹，然后选择“添加”→“控制器...”菜单选项，然后就会弹出添加控制器（Add Controller）的对话框，如图1-8所示。

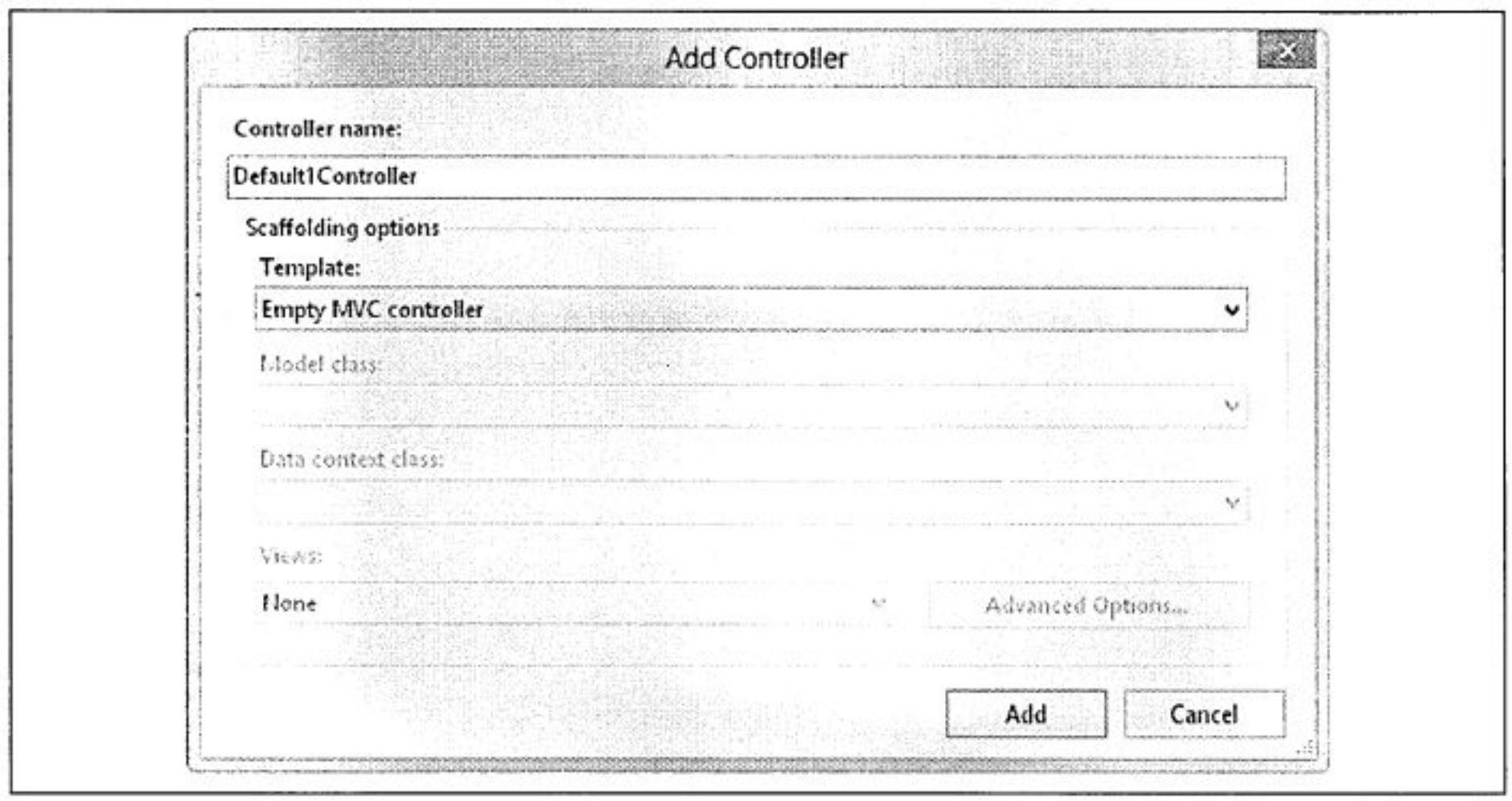


图1-8　向ASP.NET MVC网站添加控制器

添加控制器的对话框首先会要求输入控制器的名字(这个例子里,就叫它DefaultController)，然后是选择使用哪种模板，依次选择不同的选项就可以控制ASP.NET MVC生成新的控制器类。

控制器模板

添加控制器的对话框提供了几种不同的控制器模板，可以帮助你提升开发速度。

空MVC控制器

默认的模板（空MVC控制器）最简单，没有提供任何定制化的选项。因为它太简单了，所以不包含任何选项。仅仅是创建一个带有名字和一个Index操作的控制器类。

包含读/写操作的控制器和视图，使用EF框架

“包含读/写操作的控制器和视图，使用EF框架”模板名副其实，这个模板和“包含读/写操作的MVC控制器”模板有相同的输出，而且可以帮助我们生成访问EF对象的代码，更贴心的是还为这些对象生成了Create、Edit、Details和Delete视图。

如果你的项目使用了Entity Framework去访问数据，那么使用这个模板无疑等于成功了一半，而且某些情况下生成的代码应该就是想要的支持增、删、改、查的代码。

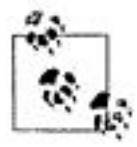
037

包含空读/写操作的MVC控制器

包含空读/写操作的MVC控制器与“空MVC控制器”模板生成的代码一样，只是增加了一些暴露标准数据操作的方法：`Details`、`Create`、`Edit`和`Delete`。

Web API控制器模板

最后的三个模板——“空API控制器”、“包含空读/写操作的API控制器”以及“包含读/写操作的API控制器和视图，使用EF框架”，属于Web API的部分内容，将在第6章详细介绍ASP.NET MVC's Web API的这些模板。



值得注意的一个有趣现象就是控制器里的代码。Visual Studio生成的代码中，`Index`和`Details`操作都只有一个操作方法，而`Create`、`Edit`和`Delete`却有两个重载方法——一个标记了`HttpPost`属性，另外一个没有。

这是因为`Create`、`Edit`和`Delete`需要两个请求来完成操作：第一个操作生成用户视图，第二个真正执行想到的操作（创建、编辑和删除）。这种情况在Web程序中非常普遍，本书里会看到几个这种例子。

不幸的是，目前我们还没讲到使用Entity Framework的内容，所以，可以先使用“包含空读/写操作的API控制器”模板选项，然后点击“添加”按钮生成新的控制器类。

Visual Studio创建`AuctionsController`之后，找到`Details`操作，添加创建`Auction`模型的代码，然后通过`View(object model)`方法传递给视图。

是的，这是一个比较次的例子。正常情况下，需要从别处读取数据，比如数据库（在第4章里将学习这个内容），但是为了这个例子，不得不使用硬编码（hardcoded）如下：

```
public ActionResult Details(long id = 0)
{
    var auction = new Ebuy.Website.Models.Auction {
            Id = id,
            Title = "Brand new Widget 2.0",
            Description = "This is a brand new version 2.0 Widget!",
            StartPrice = 1.00m,
            CurrentPrice = 13.40m,
            StartTime = DateTime.Parse("6-15-2012 12:34 PM"),
            EndTime = DateTime.Parse("6-23-2012 12:34 PM"),
        };
    return View(auction);
}
```

038

视图

控制器操作代码编写好并且可以向视图提供数据了，现在可以开始创建视图文件。

和之前的控制器类一样，也可以随便在“Views”文件夹里添加视图。但是，如果喜欢智能一些，Visual Studio也提供了专门创建视图的向导以及存放视图的文件夹。

使用Visual Studio添加视图向导的过程非常简单：右键点击控制器的任何地方，选择“添加视图”选项，就会弹出添加视图的对话框，如图1-9所示。它与添加`AuctionsController`的生成控制器向导十分类似。

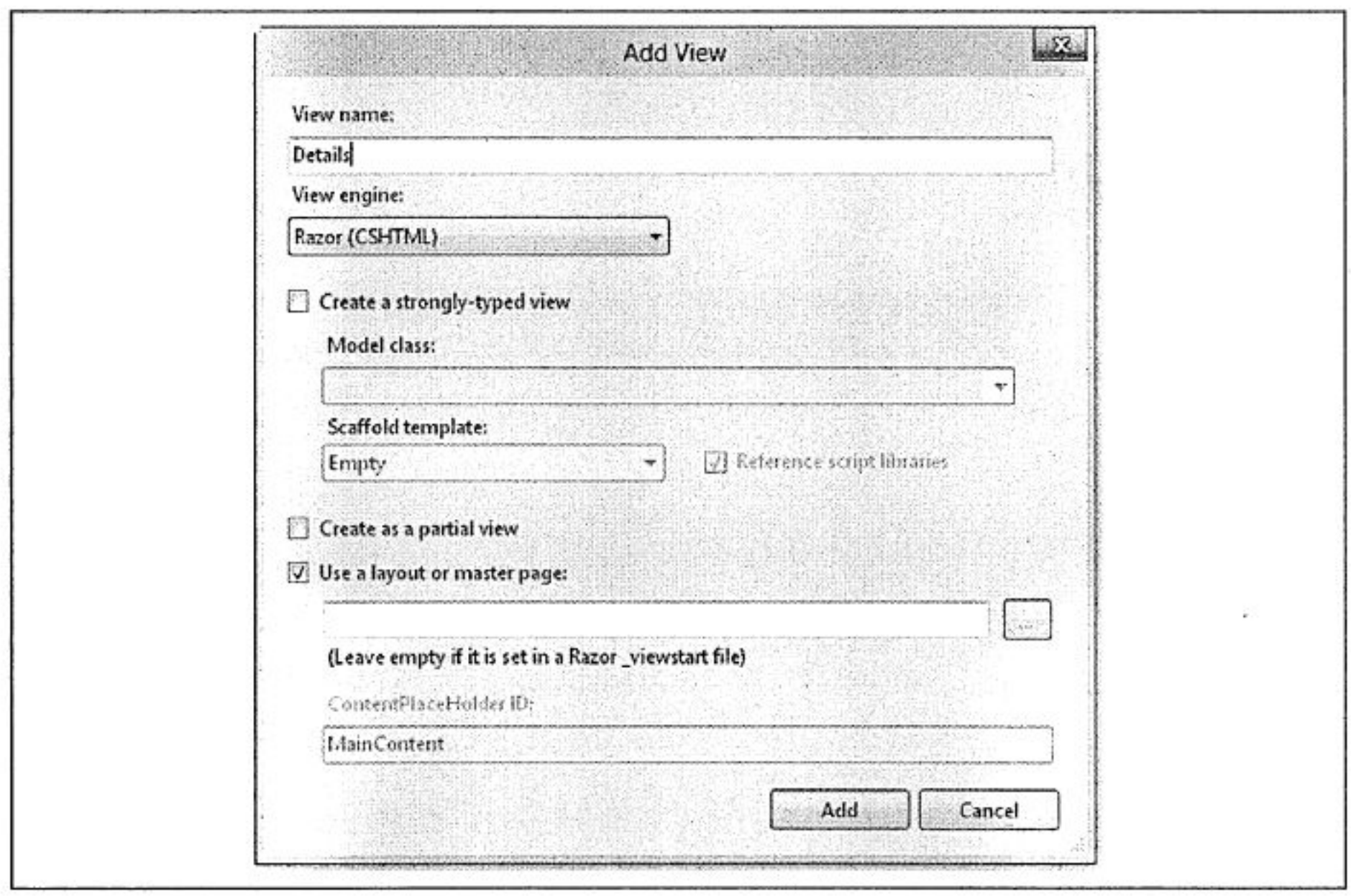


图1-9　向ASP.NET MVC网站添加视图

添加视图的对话框首先会要求输入新视图的名字。默认的名字是之前启动添加新视图Action的名字（比如Details操作触发的添加视图事件）。然后，对话框允许选择编写代码时使用的语法（视图引擎），默认为创建Web项目使用的语法，但是也可以选择别的语法，因为很可能有些视图使用Razor语法，有些ASPX使用Web Form语法。

与添加控制器的对话框一样，剩下的选项就与Visual Studio生成代码的过程有关系了。例如，可以通过从项目类列表中选择类名或者自己输入类名来生成强类型的视图模型。如果你选择了强类型视图，向导会让你选择一个模板（如Edit、Create、Delete），该模板会自动分析模型类型，并生成对应的表单域元素。

这是一种非常棒的加速开发的方式，可以节约大量输入类型名的时间。现在让我们来体验“强类型视图”带来的巨大便利！在“模型类”下拉列表框中选择“Auction”，然后选择“Details”模板。

Visual Studio“模型类”下拉列表中只包含编译成功的类，如果看不到之前创建的Auction类，就先编译整个解决方案，然后重新打开添加视图对话框。

最后，还需要告诉Visual Studio这个视图是不是部分视图，或者要不要引用布局视图。当使用ASPX Web Form语法编写代码时，选中“创建部分视图”，Visual Studio就会创建一个用户控件(.ascx)而不是完整的页面(.aspx)。当使用Razor语法时，Visual Studio会为部分视图和完整页面创建相同类型的文件(.cshtml或.vbhtml)。在Razor语法中，此复选框的唯一效果是获取在新视图生成的HTML标记。

这个demo的目的就是告诉大家：完全可以不使用默认设置。不要选中“创建部分视图”复选框，而应该选中“使用布局视图或模板页”，让布局文本框为空就行，如图1-10所示。

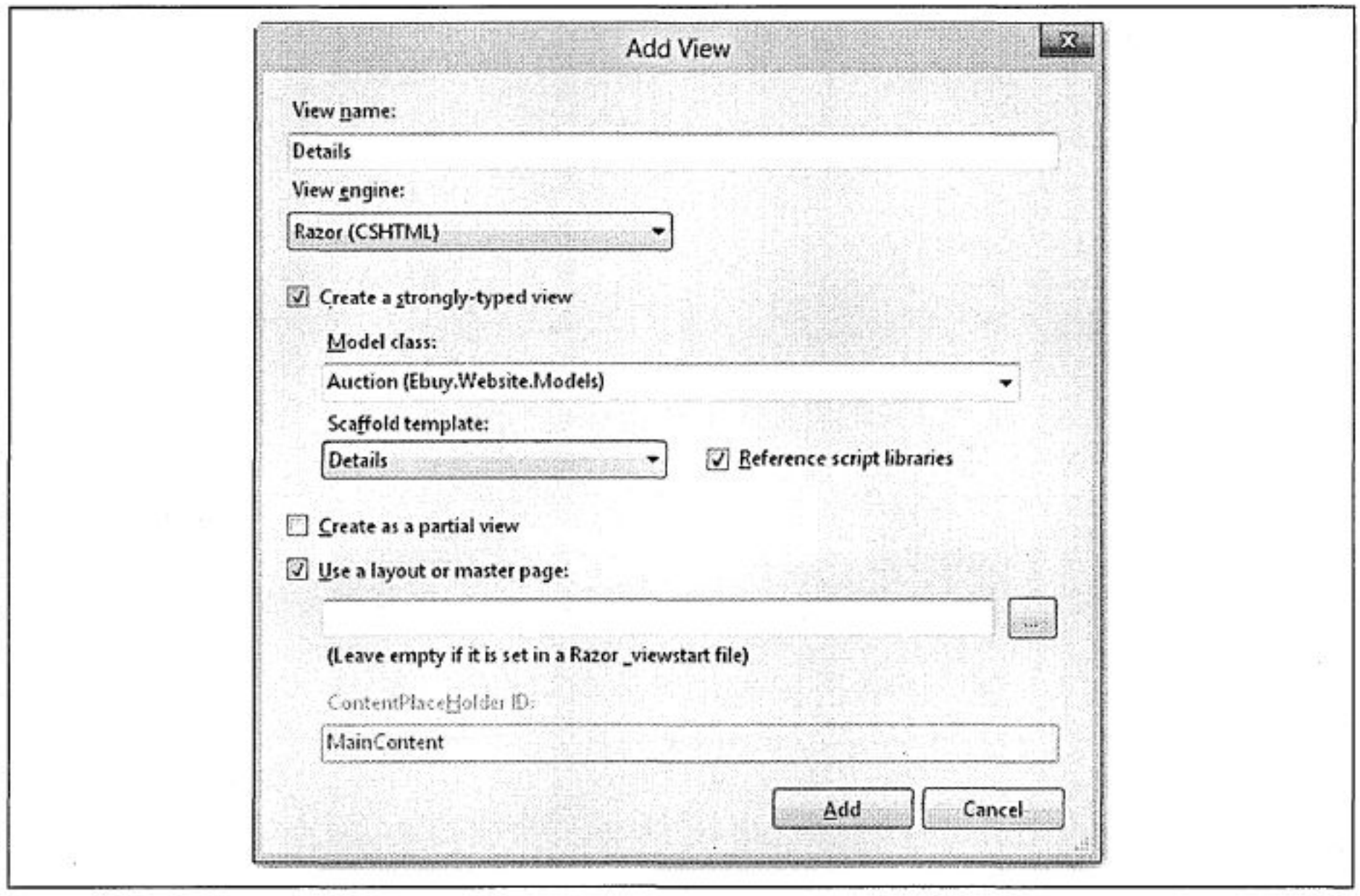


图1-10 自定义视图

配置完成以后，就可以让Visual Studio生成新的视图了。完成后，就会看到Visual Studio分析完Auction之后生成了需要的HTML代码。这里引用了HTML帮助方法如`Html.DisplayFor`，来显示所有的`Auction`域成员。

至此，已经可以运行网站了。找到控制器操作地址（比如/auctions/details/1234），看一下浏览器中渲染的Auction对象的详细内容，如图1-11所示。

生成的界面虽然不漂亮，但是Visual Studio生成HTML代码能节约必要的开发时间。一旦生成好之后，就可以随意修改HTML代码了。

恭喜！现在已经可以完全从头开始创建一个新的控制器和操作方法了。

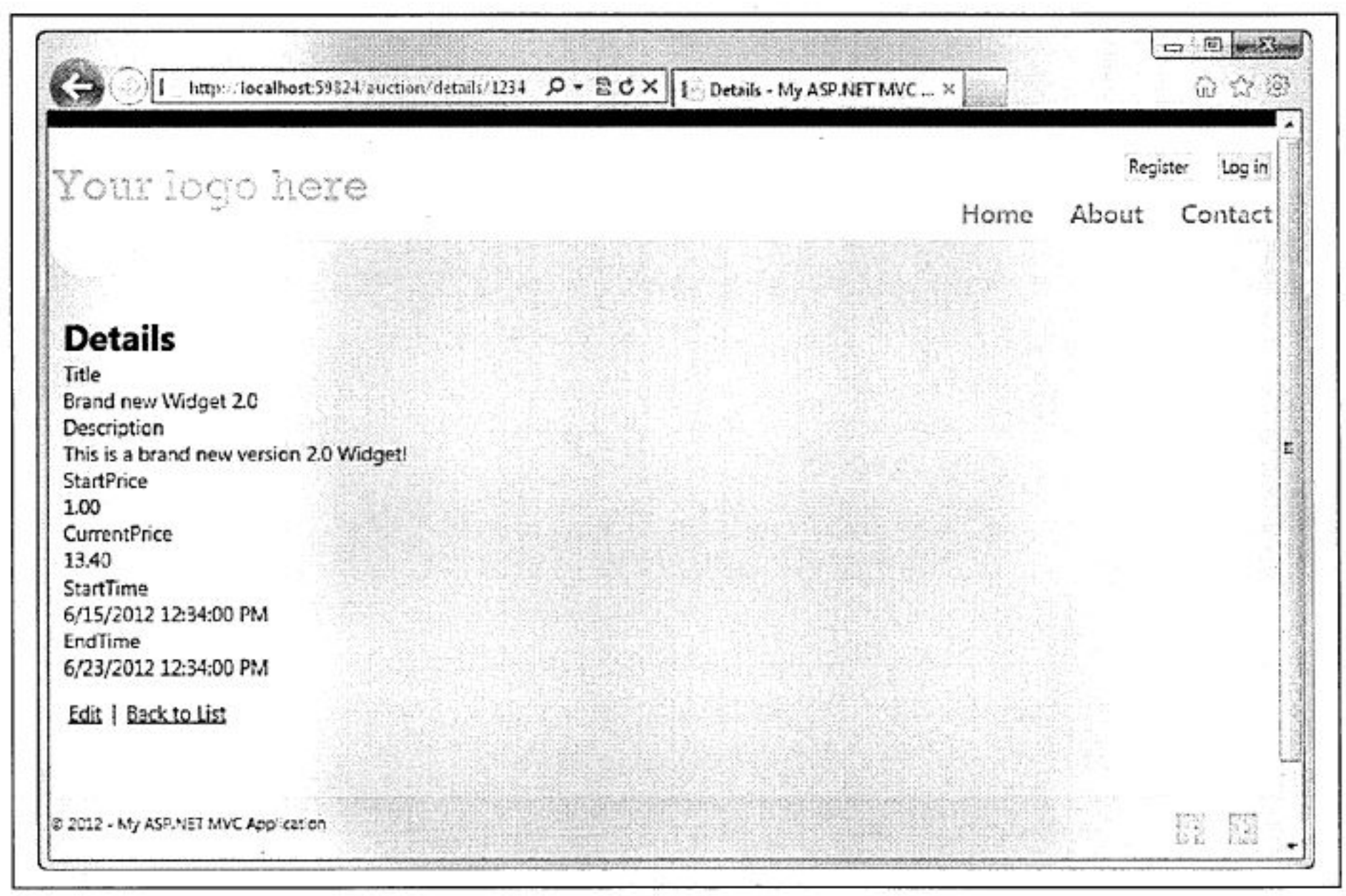


图1-11　浏览器里新渲染的视图

验证

Authentication

到目前为止，已经学习了所有创建ASP.NET MVC程序所需要的知识，但是在学习新内容之前，还有一个更重要的概念必须知道：如何在用户访问控制器操作之前验证他们的身份以保护网站的安全。

也许已经注意到了，互联网应用模板生成了`AccountController`（有一些视图支持这种验证方式），完全实现了Form验证方式。安全和身份验证是每个Web应用程序必须面对的问题，有时候需要对某些页面进行身份验证，有时候需要对整个网站进行身份验证，有时候要对特定的用户（或者用户组）进行限制以阻止未授权的用户访问网站。既然绝大部分时候都需要安全机制，为什么不从Visual Studio生成控制器和操作代码开始呢？

传统的ASP.NET网站添加身份验证的方式就是在web.config中添加设置。遗憾的是，ASP.NET MVC不支持这种方式。ASP.NET MVC程序依赖于控制器操作，而不是物理页面。

相反，ASP.NET MVC框架提供了`AuthorizeAttribute`标记属性，它可以直接应用到每个控制器操作（或者所有的控制器）上，限制只有验证通过的用户、特定的用户或者用户角色才

可以访问。

下面创建的UsersController的Profile操作，它会显示当前用户的信息：

```
public class UsersController
{
    public ActionResult Profile()
    {
        varuser = _repository.GetUserByUsername(User.Identity.Name);
        return View("Profile", user);
    }
}
```

显然，如果用户没有登录，这个操作就会失败。如果使用了AuthorizeAttribute标记属性的操作，任何未验证通过的请求都会被拒绝：

```
public class UsersController
{
    [Authorize]
    public ActionResult Profile()
    {
        var user = _repository.GetUserByUsername(User.Identity.Name);
        return View("Profile", user);
    }
}
```

如果想指定具体哪些用户可以访问这些操作，AuthorizeAttribute也暴露了User属性，可以用来设置访问的白名单、用户名。当然也可以通过Role属性设置用户角色。

现在，当未验证的用户访问这些地址时，页面会重定向到登录URL：AccountController的Login操作。

账号控制器

为了帮助我们提升开发应用程序的效率，ASP.NET MVC互联网模板包含了AccountController，也包含了使用ASP.NET成员提供者的操作方法。

AccountController提供了现成的功能，与视图一起支持每个操作。这意味着新建的ASP.NET MVC网站已经包含账号的所有相关功能，就不需要再编写代码了：

- 登录；
- 退出；
- 注册新用户；
- 修改密码。

因此，当在每个操作上标记AuthorizeAttribute时，用户就会重定向到现有的登录页面，如图1-12所示。

图1-12　默认的登录页面

而且，当用户创建新的账户时，可以点击“Log in”超链接，直接跳转到注册页面，如图1-13所示。

此外，如果不喜欢ASP.NET MVC提供的视图模板，那么也可以自定义视图来满足特定需求。

ASP.NET MVC框架不仅可以轻易实现每个操作的验证功能，而且默认的项目模板也能实现用户验证需要的全部功能。

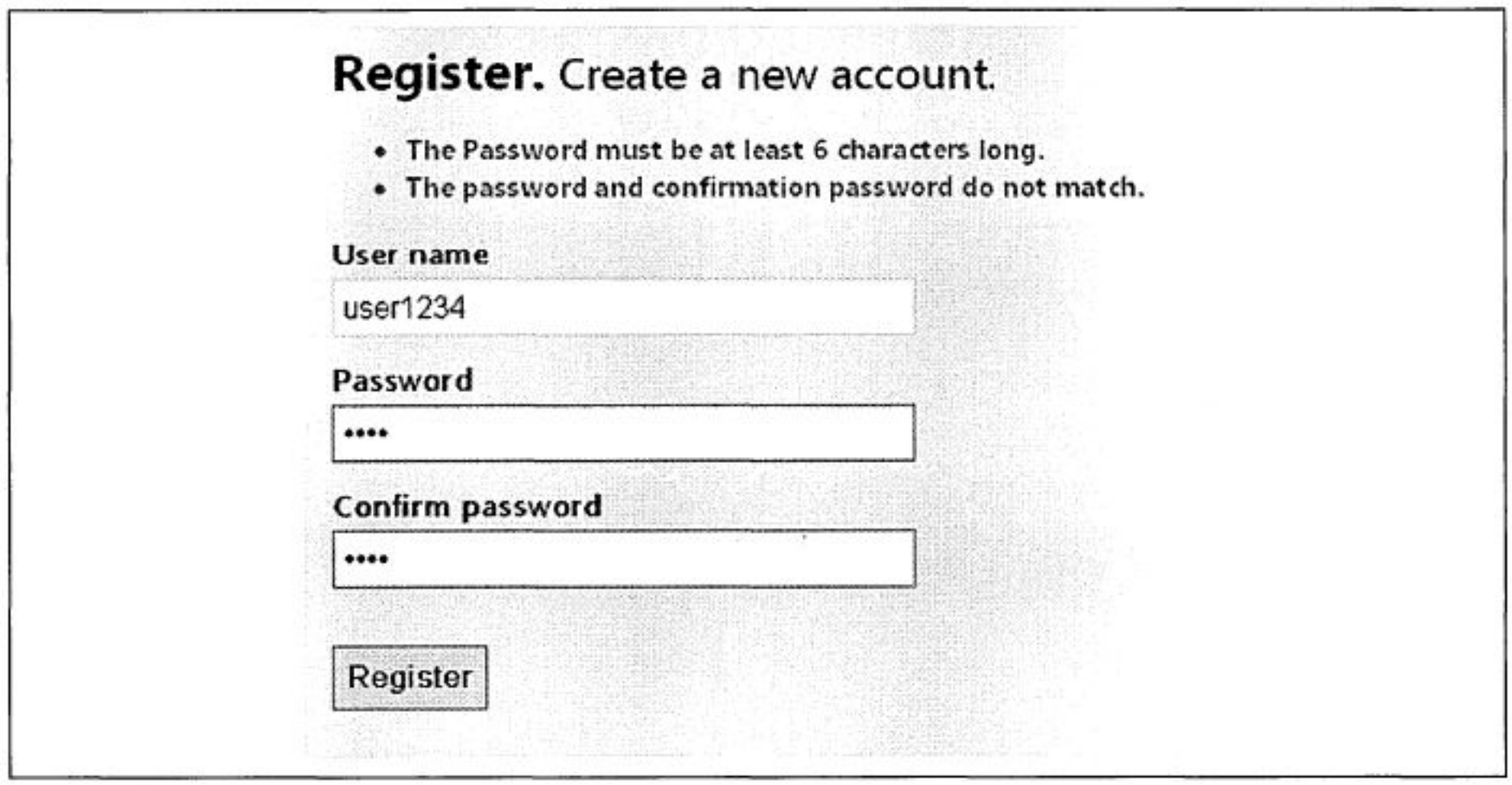


图1-13　默认的注册页面

总结

Summary

ASP.NET MVC不仅使用了久经考验的MVC架构模式，而且它还是一个全新的、支持松耦合架构以及各种面向对象编程模式和实践准则的网站开发框架。

ASP.NET MVC框架引入了强大的项目模板，并支持“惯例优先原则”，避免了大量烦琐的工作，节约了开发时间，提升了网站开发的效率。

本章介绍了构建ASP.NET MVC 4应用程序需要的基本概念和技术。接下来，本书会扩展这些基础知识，并介绍更多ASP.NET MVC框架的功能，以帮助我们构建强壮的、易维护的Web应用。

还等什么？让我们继续学习开发高性能Web应用所需的所有知识吧！

第2章

ASP.NET Web Form开发人员必读

ASP.NET MVC for Web Forms Developers

尽管ASP.NET MVC架构和Web Form架构区别很大，但是还是有很多共同之处。毕竟它们都是以ASP.NET API和.NET框架为基础构建的。所以，如果打算学习ASP.NET MVC，并且从事过Web Form开发，那么算是已经提前开始了。

本章将会比较ASP.NET MVC和Web Form框架，以便知道有多少Web Form的概念与ASP.NET MVC开发相关。注意，本章的目标是帮助那些想学习ASP.NET MVC开发的Web Form开发人员尽快转变思维，以适应新的开发框架。如果不熟悉Web Form框架，则可以考虑跳过本章（译注1）直接看下面的章节。

ASP.NET同门兄弟

It's All Just ASP.NET

也许我们不知道，基于.NET平台开发网站的框架（你可能称之为“ASP.NET”）实际上可以分成两个部分：可视化用户界面（Web Form）和后台Web组件（ASP.NET）。两者可以通过它们的命名空间区分开：所有`System.Web.UI.*`命名空间下的内容可以称为Web Form，而`System.Web.*`命名空间下的其他内容可以称为ASP.NET。

与Web Form一样，ASP.NET MVC（所有类都在`System.Web.Mvc`命名空间下）也是基于ASP.NET平台构建的。所以两个平台可以十分相似，也可以完全不同，这取决于怎么看待这个问题。本章介绍两者的相似之处，而其他章节则会介绍它们的区别。

工具、语言和API

对于初学者来说，两个框架都使用并扩展了.NET框架。本质上，两者都可以使用C#和VB.NET来访问.NET框架，当然也支持.NET框架上的任何语言。

译注1：个人强烈推荐！作为ASP.NET Web开发技术人员，有必要阅读本章。本章的知识可以帮助你了解Web开发框架的基本原理、两种架构的不同设计原则，能为以后的Web架构设计提供充足的参考依据。

这个特性让现有的ASP.NET代码具备了高复用性。例如，如果已经有一个Web Form的网站，它使用了System.Xml API访问XML文件，那么，就可以很轻易地在ASP.NET MVC网站中复用这部分代码，或者稍加改动即可。

大家应该对使用Visual Studio编辑ASP.NET MVC网站及其项目不会感到陌生，这和Web Form网站开发一样，都是基于.NET平台的应用程序。还可能要注意另外一些共享文件，比如web.config和Global.asax，它们在ASP.NET MVC 和 Web Form应用中起着重要作用。

HTTP处理程序和模块

ASP.NET MVC和Web Form共享的最著名的部分应该就是HTTP处理程序和模块。虽然大部分Web Form API（System.Web.UI命名空间下）不会在ASP.NET MVC程序下运行，但是HTTP处理程序和模块确实是ASP.NET(System.Web)API的一部分，所以它们仍然会在ASP.NET MVC上下文里发挥作用。事实上，ASP.NET MVC管道本身正是从使用HTTP处理程序（译注2）处理外部请求开始的。

确保已经阅读了本章关于ASP.NET MVC和Web Form的区别以及如何使用HTTP处理程序和模块的内容。HTTP处理程序和模块依然使用在ASP.NET MVC程序里，记住，有几种情况，比如View State（视图状态）不能在ASP.NET MVC程序里使用。

管理状态

管理状态即与用户相关的数据，在每个程序里都是个非常重要的内容，Web的无状态性（译注3）导致Web应用程序中的管理状态数据变得相当复杂。

为了处理管理状态这个问题，ASP.NET Web Form引入了“视图状态”的概念。每个请求的状态数据，在序列化之后会临时储存在页面的隐藏域控件里，随后的请求都会回传这些数据。视图是ASP.NET Web Form的重要部分，几乎每个页面和组件都会或多或少依赖于它。视图状态抽象也有它的缺陷，每个包含视图状态的请求必须以Form Post方式提交给服务器，而那些隐藏域的数据大小令人头疼：平时用不到，有时候却又非常大。

047

译注2：HTTP 处理程序和模块（Handler and Module）属于 ASP.NET 管道的两个部分。HTTP 处理程序负责处理特定类型的请求消息，常见的处理程序如 ASPX、ASMX、HTML 页面、JavaScript、图片等；模块可以拦截所有请求，比如可以自定义特定模块来实现安全验证、日志、统计信息等。两者的更详细信息可以参考 MSDN。

译注3：Web 的无状态性这个问题我在“WCF 分布式开发技术课程”之 REST WCF 服务一节中有专门的介绍。HTTP 规范本身是无状态性的。很多动态网站开发框架为了解决这个问题，都提供了自己的机制，如 ASP.NET 框架的会话功能。

状态管理解释了ASP.NET MVC完全不同的状态管理方法，它留给开发人员去实现（或者不实现）。最重要的是，ASP.NET MVC完全抛弃了视图状态机制。

幸运的是，除了视图状态，ASP.NET还提供了其他几种状态管理方法，同样可以在ASP.NET MVC里使用。例如，可以使用ASP.NET缓存和会话状态，或者`HttpContext.Items` API，这些可以帮助你在ASP.NET MVC应用程序里管理状态。

部署和运行时

ASP.NET MVC的产品环境部署和Web Form程序一样。这意味着之前学习的关于部署和维护ASP.NET程序的任何东西，比如IIS、.NET应用程序池、跟踪、错误调试和部署程序集bin文件夹等，同样可以应用到ASP.NET MVC程序的部署中来。虽然ASP.NET MVC和Web Form程序采用了不同的架构，但是，从根本上说，两者都是部署并执行.NET代码来处理HTTP请求的。

更多的差别

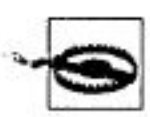
More Differences than Similarities

前文介绍的两者共同点可能让你感觉ASP.NET MVC和Web Form框架看起来是一样的。其实，只要我们深入了解它们，就会发现有更多的差别。

表2-1对比了两种框架的不同之处。

表2-1　ASP.NET MVC和Web Form的基本区别

Web Form	ASP.NET MVC
视图和逻辑紧密耦合	视图和逻辑分离
页面（基于文件的URL）	控制器（基于路由的URL）
状态管理（视图）	无自动化的视图管理
Web Form语法	自定义语法（默认为Razor）
服务器控件	HTML帮助方法
母版页	布局
用户控件	部分视图



两个框架都是基于ASP.NET平台构建的，所以有很多种方式可以让Web Form做到ASP.NET MVC的样子，当然，反过来也一样可以实现。也就是说，可以把MVC开发技巧应用到Web Form上，反之也成立。

不管怎样，都请记住：本章关于两种框架的比较参考了各自框架的开发实践，比如微软官方的文档和教程。

应用程序业务和视图分离

ASP.NET MVC和Web Form最大的区别就是它们各自采用基础架构概念。

例如，Web Form框架引入初期，就声称相比前任ASP框架，它能提供更好的分离关注点支持（现在听起来有点讽刺），ASP强制开发人员把业务逻辑和页面标记语言混合在单个页面里。Web Form框架不仅可以让业务逻辑和标记语言代码分离，还可以提供更强大的开发平台——.NET框架，可以基于此平台编写业务逻辑代码。但是，尽管Web Form框架相比传统的ASP框架更加先进，但是它依然关注于“网页”上；虽然其代码可移植到别的地方，但是仍然很难做到业务逻辑代码和用户视图分离（译注4）。

ASP.NET MVC通过集成松耦合组件来处理请求，构建于“分离关注点”概念之上。这个方法隔离了单个的组件开发，不仅有利于开发生命周期，更有利于组件的测试工作。同样，它也让动态处理请求更加方便，下面的章节里会做介绍。

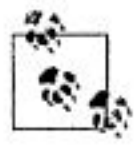
URL和路由

Web Form架构下，每个网站的URL由单个的物理文件.aspx表示，而且每个页面与单个页面类紧密耦合（代码后置（译注5））；页面无法动态选择自己绑定的类，代码后置类无法渲染别的视图。

Web Form框架基于页面的请求处理方式与ASP.NET MVC的请求处理方式形成鲜明的对比，后者依赖于复杂的路由规则动态映射外部的URL到正确的控制器操作上，并允许操作动态指定要显示给用户的视图。

例如，请求处理管道在处理来自用户浏览器跳转到URL */auctions/details/123*的结果和处理AJAX请求的结果可能不同，这是因为AJAX请求消息头里包含了一个标识AJAX请求的数据。用户直接访问URL时，服务器可能返回包含整个页面内容的HTML代码、JavaScript和CSS；而AJAX请求返回的只是Auction对象的序列化数据。此外，服务器可能选择不同的格式来序列化Auction对象，比如JSON或者XML，这些取决于请求要求的数据格式。

Web Form框架本身并不支持这种动态处理请求的功能，需要借助HTTP处理程序和HTTP模块才能实现。



ASP.NET的路由功能不是只有ASP.NET MVC程序可以使用，Web Form程序同样可以使用。

两者使用路由的不同之处是ASP.NET MVC框架离不开路由组件，而Web Form程序使用路由可以避免与页面路径相关的限制，从而获得对URL更好的控制。

译注4：这一点和Java的JSP开发框架很像。
译注5：有的文章将code-behind翻译成“代码隐藏”，本文将其翻译成“代码后置”。

状态管理

ASP.NET MVC和Web Form最具争议的区别也许就是它们如何跨请求处理用户的状态信息，特别是ASP.NET MVC抛弃了视图状态。为什么要删除这么重要的功能呢？简单地说就是ASP.NET MVC完全拥抱了Web标准的无状态的属性。为了更好地理解这种设计的初衷，我们先来回顾Web Form的历史。

此外，除了比ASP提供了更好的开发平台外，Web Form框架的另外一个特点就是引入了一种厚重的原生客户端应用开发技巧，比如“拖放控件”以及快速开发RAD的概念到Web开发中。

为了体验原生客户端应用开发，Web Form必须在Web开发的基础概念如HTML标签以及CSS之上抽象出一层。原生应用程序开发中最重要的概念就是有状态的（stateful），这意味着应用知道交互的用户状态，并且可以在跨应用之间重用状态信息。

Web是基于HTTP请求的，每个请求对应一个客户端请求和一个服务器应答。Web服务器必须分开处理每个请求，因此无法知道客户端请求的前后消息，这就让服务器和客户端无法进行有效的会话。

为了在无状态的中介上实现有状态的交互，必须抽象出来一层，这样，视图状态就诞生了。简单地说，视图状态序列化了客户端和服务端之间的交互状态信息，并把它们存储在每个页面的隐藏域里，随后发送给客户端。客户端需要在后续的请求中把这些会话状态信息传回给服务端。

ASP.NET MVC框架保留了Web无状态的本性，但并没有提供类似视图状态的机制，而是利用了缓存和会话状态。相反，ASP.NET MVC框架希望客户端请求包含所有需要的数据，以便于服务端处理它们。例如，ASP.NET MVC应答消息也许可以使用Auction的ID，而不是从数据库里查询出`Auction`数据后序列化整个对象发送给客户端，并在后续请求里再回传给整个对象。后续请求可以直接使用`Auction`的ID，ASP.NET MVC控制器可以使用它从数据库里查询Auction数据。

显然，两种方法各有千秋。视图状态让客户端和服务端的交互更加简单，但是它包含的数据可能会变得臃肿，占用大量的带宽。换句话说，视图状态意味着开发人员可以省去很多麻烦，但会以占用带宽为代价。当然，如果开发人员不考虑每个页面的存储数据，则问题可能会变得更糟。ASP.NET MVC的方法可以从某种程度上削减页面内容，但是可能增加后台处理请求和数据库请求的成本。

渲染HTML代码

每个网站各不相同，但是所有的网站都有一个共同点：生成各自需要的HTML代码。所以，Web开发框架的首要任务就是帮助开发人员高效地渲染HTML代码。ASP.NET MVC和Web Form都可以完美地渲染HTML代码，只是方法差别比较大。

Web Form视图比较容易区分，Web Form视图里的各个控件，从<label>到特定的Ajax控件都使用服务端控件来渲染HTML代码。

例如，下面就是一个典型的Web Form页面的 HTML视图：

```
<%@ Page Title="EBuy Auction Listings" Language="C#" AutoEventWireup="true"
    MasterPageFile="~/Layout.master"
    CodeBehind="Default.aspx.cs" Inherits="EBuy.Website._Default" %>
<%@ Register TagPrefix="uc" TagName="SecondaryActions"
    Src="~/Controls/SecondaryActions.ascx" %>

<asp:ContentID="HeaderContent" runat="server" ContentPlaceHolderID="HeadContent">
    <uc:SecondaryActionsrunat="server" />
</asp:Content>

<asp:ContentID="BodyContent" runat="server" ContentPlaceHolderID="MainContent">

<div class="container">
    <header>
        <h3>Auctions</h3>
    </header>

    <asp:Repeaterid="auctions" runat="server" DataSource="<%# Auctions %>">

    <HeaderTemplate>
        <ulid="auctions">
    </HeaderTemplate>

    <ItemTemplate>
        <li>
        <h4 class="title">
        <asp:HyperLinkrunat="server"
            NavigateUrl='<%# DataBinder.Eval(Container.DataItem, "Url") %>'>
            <%# Container.DataItem("title") %>
        </asp:HyperLink>
        </h4>
        </li>
    </ItemTemplate>

    <FooterTemplate>
        </ul>
    </FooterTemplate>

    </asp:Repeater>

</div>

<script type="text/javascript"
         src="<%: RelativeUrl("~/scripts/jquery.js") %>"></script>
</asp:Content>
```

现在，对比以上视图和以下Razor的代码，二者是等价的：

```
@model IEnumerable<Auction>
    @{
        ViewBag.Title = "EBuy Auction Listings";
    }

    @section HeaderContent {
        @Html.Partial("SecondaryActions")
```

```
}

<div class="container">
    <header>
        <h3>Auctions</h3>
    </header>

<ul id="auctions">

@foreach(var auction in Model.Auctions) {
    <li>
        <h4 class="title">
            <a href="@auction.Url">@auction.Title</a>
        </h4>
    </li>
}

</ul>
</div>

<script type="text/javascript" src="~/scripts/jquery.js"></script>
```

虽然两种方式不同，但是两者都可以高效地渲染HTML代码。

注意：Razor采用了更加专注于代码的方式来生成HTML，它依赖于像foreach循环这种特定的代码而不是特定的服务端控件，比如`<asp:Repeater>control`。

比较ASP.NET Web Form和Razor在渲染简单的URL连接时的代码区别。下面是ASP.NET Web Form的实现代码：

```
<asp:HyperLinkrunat="server" NavigateUrl='<%# auction.Url%>'>
    <%: auction.Title%>
</asp:HyperLink>
```

Razor语法的实现代码如下：

```
<a href="@auction.Url">@auction.Title</a>
```

这两个例子代表了两者框架的区别：Razor帮助开发人员编写HTML代码，而Web Form视图依赖于服务端控件来渲染HTML代码。

HTML帮助方法与服务器控件

再回顾一下上一节的内容，因为这相当重要。使用Web Form框架的抽象机制完全可以只使用服务器控件来开发整个页面而不需要使用HTML标签，这些服务器控件已经封装到Web Form框架里，可以直接使用它们来开发出HTML页面功能。

与此同时，用ASP.NET MVC方法渲染HTML代码的方式正好与Web Form相反，因为它要求开发者编写更多的HTML代码。但是，这并不是说ASP.NET MVC无法帮助生成HTML代码，它可通过HTML帮助方法来帮助渲染HTML代码，开发人员可以直接在视图里调用它的方法。

从逻辑上讲，HTML帮助方法和服务器控件的效果一样：它们都是基于组件的，并且在视图里执行以帮助生成HTML代码，这样就可以减少开发人员的工作量，提高开发效率。

HTML帮助方法和服务器控件主要的不同之处就是在它们的技术实现上：服务器控件是继承自某些成熟的基类，而HTML帮助方法扩展了ASP.NET MVC视图对象的`HtmlHelper`对象。

HTML帮助方法和服务器控件的另外一个重要的不同我们在本章多次提到：服务器控件可能使用了视图状态，而HTML帮助方法却完全没有使用。

除了这些不同外，大部分服务器控件使用了与HTML帮助方法等价的类。

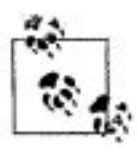
与Web Form标签`<asp:HyperLink>`不同，ASP.NET MVC提供了`Html.ActionLink()`方法。例如，

```
@Html.ActionLink(auction.Title, "Details", "Auction")
```

这个例子中，给`Html.ActionLink()`传递参数用于显示文本、操作（“Details”）和控制（“Auction”）的名字。它们的名字会用来构建URL，渲染如下：

```
<a href="/Auction/Details/">My Auction Title</a>
```

与Web Form不同，ASP.NET MVC并没有提供生成全部HTML代码的帮助方法，但是它提供了可以生成大部分核心HTML代码的帮助类。



与Web Form服务器控件一样，也可以自由创建自己的HTML帮助方法，封装高复用性的渲染逻辑代码。

部分视图与用户控件

HTML帮助方法与Web Form的服务器控件等价，ASP.NET MVC的部分视图与Web Form的用户控件本质上是同一个东西：视图被分别存放在不同的文件里，允许开发者把大视图文件分割成许多小视图文件，然后在运行时动态组合。与用户控件一样，部分视图提供了封装视图模块的绝佳方式，我们可以在不同的视图中重用这部分代码。

布局与母版页

最后，我们来看一下视图中最重要的、最基础的概念之一：布局（layout）。它可以定义页面结构以及网站主题，并且与所有的页面共享这个主题。这个概念类似于ASP.NET Web Form中的母版页（master page），在ASP.NET MVC框架中称之为布局。

ASP.NET MVC布局可以让开发人员定义每个页面共享的HTML代码。与母版页一样，ASP.NET MVC布局允许开发人员指定多个内容块。与Web Form的内容页一样，ASP.NET MVC视图也可以指定想要在哪个视图里渲染。但是，有一个主要区别：对ASP.NET MVC管道来说，设置仅仅作为“推荐”，我们也可以随意修改视图的布局，包括删除或者完全使用无布局来渲染视图，比如在Ajax请求的情况下。

使用Web Form语法编写ASP.NET MVC视图

Authoring ASP.NET MVC Views Using Web Forms Syntax

既然又到了本章的末尾，现在是时候揭开一个秘密了：Razor并不是开发ASP.NET MVC视图

的唯一方法。事实上，如果你对Web Form语法还痴心不悔、藕断丝连，那么就可以继续使用Web Form的语法来开发 ASP.NET MVC视图。

现在，不要兴奋过头了，记住，只是可以使用Web Form语法，而不是其框架。换句话说，本章介绍的内容都是正确的，无论使用何种语法，URL路由仍然会选择运行哪个控制器、显示哪个视图；ASP.NET服务器控件不能工作，而且更重要的是不可能再使用视图状态了。

所有的一切表明：使用Web Form语法来编写ASP.NET MVC视图代码只能在.aspx、.ascx和母版页里使用<% %>语法（与Razor的@语法对应）。母版页在这个规则下是个例外，但是它们可以像在Web Form程序里一样继续工作，所以ASP.NET MVC视图可以利用分离网站HTML和单独内容页代码的优势。

其实，ASP.NET MVC不仅支持Razor和Web Form语法，还可以混用它们。例如，Razor视图可能调用`@Html.Partial("MyPartial")`去引用MyPartial视图，而这个视图碰巧是个Web Form控件。但是不能混用布局样式，因为Razor布局和Web Form母版页方法两者并不兼容。这意味着Razor视图不能引用Web Form模板，Web Form内容页不能引用Razor布局。

要点提示

虽然我们在本章花费大量的篇幅来讨论如何把现有的Web Form开发技术应用到ASP.NET MVC中来，但是，如果拥有多年的Web Form开发经验，那么学习ASP.NET MVC时可能也有缺点：除了它们的许多共同点和相似性之外，ASP.NET MVC和Web Form框架架构和设计目标在根本上是不同的。如果你习惯于“Web Form方式”，而且喜欢把这些方法用到ASP.NET MVC中来，那么可能会带来问题。

正如之前介绍的，“最重要的”和“危险的”的区别就是ASP.NET Web Form尽全力引入并维护状态，而ASP.NET MVC没有。从技术角度来说，这个问题就是从ASP.NET Web Form转向ASP.NET MVC开发时不再有视图状态，ASP.NET Web Form中大部分“状态性的”东西都不会再出现了。

同样，还要记住需要放进视图状态的东西。通常，视图状态可以很方便地实现跨请求的用户数据共享。但是，此时需要使用ASP.NET MVC方式来解决这个问题，找到一个临时媒介来存储这些数据，比如会话状态和应用程序缓存。

另一个编写代码时的最大区别就是Web Form依赖于服务器控件和用户控件，而ASP.NET MVC使用HTML帮助方法和部分视图。虽然概念上十分相似，但是两种方法不能互换，而且最好不要混用（通常是不能混用的）。

使用Razor语法可以避免这个问题，因为我们不是在编写Web Form页面。ASP.NET提供了ASPX视图引擎，可以用Web Form语法开发ASP.NET MVC视图。虽然ASPX视图引擎渲染HTML页面的功能很强，但是如果不注意，也许就会不由自主地犯错而使用Web Form框架。为了避免混淆，有些视图虽然可用Web Form语法，但本书还是将所有视图使用Razor语法。

虽然这些问题非常值得关注，但这些问题不应该阻止我们学习并使用ASP.NET MVC，或者阻止我们继续使用ASPX视图引擎。当我们实现ASP.NET MVC功能时，只要记住本章提到的概念即可。如果开发中我们不断地问自己“这个符合MVC方法吗？”或者“我是否完全使用了ASP.NET MVC框架的功能？”，就可以完全利用Web Form的开发技巧并避免这些错误。

总结
Summary

因为ASP.NET MVC和ASP.NET Web Form共用了基础架构，所以Web Form开发人员学习ASP.NET MVC框架具备先天性优势。本章在介绍两个框架的相似之处的同时，也介绍了两者处理特定场景的不同之处。本书的附录A里会介绍如何基于共同的框架来进行兼容开发，附带的例子也会讲述如何从ASP.NET Web Form应用迁移到ASP.NET MVC网站。

第3章 使用数据

Working with Data

现在很难找到不处理数据的应用程序，所以对于ASP.NET MVC提供各种级别简化数据访问的框架这一点我们不会感到惊讶。本章将会介绍ASP.NET MVC提供支持的这些工具，以及如何通过电子交易网站EBuy中的驱动场景（data-driven scenario）使用这些功能来处理数据。

因为EBuy是个电子交易网站，所以网站最重要的场景就是允许用户创建要销售的商品信息及商品列表。下面来看一下ASP.NET MVC框架如何帮助我们处理这种场景。

构建表单

Building a Form

HTML表单（HTML form）的概念与Web一样古老。虽然现在的浏览器已经变得相当强大，可以随意设置我们喜欢的HTML表单样式，可通过JavaScript来控制它们的行为（这些都是5年前可能都无法想象的），但是本质上这些操作、显示以及回传到Web服务器的元素仍旧是一些朴素的旧的表单域。

虽然ASP.NET MVC框架鼓励我们手工编写更多的HTML代码，但是它也提供了一系列HTML帮助方法来帮助生成HTML标签，比如`Html.TextBox`、`Html.Password`和`Html.HiddenField`等。ASP.NET MVC同样也提供了一些更智能的帮助方法，比如`Html.LabelFor`和`Html.EditorFor`，它们可以根据名称和传入的模型属性来动态确定合适的HTML。

这些帮助方法就是我们在电子交易网站EBuy用来创建HTML表单的工具方法。通过这些HTML表单方法，用户可以向服务器`AuctionsController.Create`操作回传新创建的交易信息。要了解如何使用这些帮助方法：先添加一个名为Create.cshtml视图，然后使用下面的HTML标签来填充视图：

```
<h2>Create Auction</h2>

@using (Html.BeginForm()) {
    <p>
        @Html.LabelFor(model =>model.Title)
        @Html.EditorFor(model =>model.Title)
    </p>
    <p>
        @Html.LabelFor(model =>model.Description)
    @Html.EditorFor(model =>model.Description)
```

```
        </p>
        <p>
            @Html.LabelFor(model =>model.StartPrice)
            @Html.EditorFor(model =>model.StartPrice)
        </p>
        <p>
            @Html.LabelFor(model =>model.EndTime)
            @Html.EditorFor(model =>model.EndTime)
        </p>
        <p>
            <input type="submit" value="Create" />
        </p>
    }
```

接着在控制器里添加下面的操作来显示这个视图：

```
[HttpGet]
public ActionResult Create()
{
  return View();
}
```

这个视图将会渲染下面的HTML代码到浏览器中：

```
<h2>Create Auction</h2>

<form action="/auction/create" method="post">
    <p>
        <label for="Title">Title</label>
        <input id="Title" name="Title" type="text" value="">
    </p>
    <p>
        <label for="Description">Description</label>
        <input id="Description" name="Description" type="text" value="">
    </p>
    <p>
        <label for="StartPrice">StartPrice</label>
        <input id="StartPrice" name="StartPrice" type="text" value="">
    </p>
    <p>
        <label for="EndTime">EndTime</label>
        <input id="EndTime" name="EndTime" type="text" value="">
    </p>
    <p>
        <input type="submit" value="Create">
    </p>
</form>
```


用户可以在这个页面表单中填写商品数据，并提交给/auctions/create操作。从浏览器角度看，虽然提交数据的地址和开始的页面地址一样（最初的页面表单地址也是/auctions/create），但是现在提交给新数据的是另外一个操作，这个`Create`操作标记了`HttpPostAttribute`标记属性，用来告诉ASP.NET MVC框架，它是用来处理HTML表单`Post`方式提交的操作方法。

这些添加的代码一旦验证成功，那么可以编写逻辑代码来处理这些提交的数据了。但是该怎么做呢？

处理表单Post

Handling Form Posts

在处理这些传递给控制器操作的数据之前，首先要把这些数据从请求消息中取出来。这样做的最简单的方式就是把模型当做操作参数。幸运的是，我们已经创建了一个模型：Auction类。

为了绑定之前定义的Auction类，现在要在操作方法上把Auction模型作为参数类型，代码如下：

```
[HttpPost]
public ActionResult Create(Auction auction)
{
    // 在数据库里创建 Auction

    return View(auction);
}
```

此刻，Auction模型类最重要的地方就是它的属性名(Title、Description等)要与HTML表单中要提交的域元素名称一致。这些属性的名字至关重要，ASP.NET MVC模型绑定机制会从请求消息中提取这些字段的值，并把它们赋值给Auction模型对象。

如果运行网站、填写交易数据并提交HTML页面表单，就可以看到Auction模型参数已经赋值了。现在代码里只是回传了auction参数的值给视图，并没做别的处理。我们可以把这些值显示给用户，以进行确认。

到达这一步，已经非常接近实际项目的需求了。不过实际的代码更加复杂，可能要保存数据到数据库，并且返回给用户的视图可能是保存成功的或者失败的页面。现在就开始编写保存数据的代码。

保存数据到数据库

Saving Data to a Database

虽然ASP.NET MVC框架没有直接提供内置的数据库访问支持，但是有很多流行的.NET数据访问库可以用来轻易实现数据库访问操作。

微软的Entity Framework（也称EF）就是其中之一。Entity Framework是一个简单、灵活的对象关系映射(object relational mapping, ORM)框架,它可以使用面向对象的方式帮助开发人员查询、更新存储在数据库中的数据。

此外，Entity Framework实际上也是.NET框架的一部分，微软对它提供了全力支持，而且有很多技术文档供我们学习、使用。

Entity Framework提供了一些定义数据模型和使用模型访问数据库的方法，但是最好的方法应该是代码优先（Code First）方法。这个方法提倡以应用程序模型为中心，并以应用模型驱动整个开发过程。

代码优先：惯例优先原则

使用代码优先（Code First）方法开发项目时，数据库交互通过简单的模型类进行（朴素的旧

的对象Plain Old CLR Objects，简称POCO）。Entity Framework代码优先（译注1）方法非常强大，甚至可以在运行程序时根据我们定义的模型来生成数据库Schema，并使用这个Schema来创建数据库和实体（表、关系等）。

代码优先方法通过遵守特定的惯例来自动评估模型层中各种属性和模型类的信息、它们之间的关系以及如何使用数据库表示这些模型关系。

例如，电子交易网站EBuy里，Auction模型类映射到数据库中的Auctions表上，而且表中的每个列与模型类的属性一一对应。这个表和列名就是自动从模型类中获取的。

前面展示的Auction模型类十分简单，但是实际项目中随着应用程序业务逻辑的需要，可能会变得相当复杂：会添加更多的属性、业务逻辑，甚至是与其他模型之间的关系。这些都不是Entity Framework代码优先方法关心的，但是，复杂的模型是从简单的模型进化而来的。第8章里，根据现在的简单Auction模型构建了更复杂的Auction模型，证明了Entity Framework代码优先可以处理更复杂的映射关系。当然也有它无法处理的情况。

使用Entity Framework代码优先模式创建数据访问层

Entity Framework代码优先模式的核心依赖于`System.Data.Entity.DbContext`类。这个类（或者它的子类）是代码访问数据库的网关，它提供了数据库相关的操作。

在使用`DbContext`类之前，需要先继承它。事实上，编写子类代码非常容易：

```
using System.Data.Entity;

public class EbuyDataContext: DbContext
{
    public DbSet<Auction>Auctions { get; set; }
}
```

在例子Ebuy的DataContext.cs里，我们创建了一个自定义的数据上下文类，名字叫`EbuyDataContext`，它继承自`DbContext`。这个类定义了一个属性`System.Data.Entity. DbSet<T>`，`T`是编辑或保存到数据库的实体对象的类型。上面的例子定义了`System.Data.Entity.DbSet<Auction>`，表示需要保存和编辑`Auction`对象数据到数据库。可以在数据上下文里定义多个实体，随着学习的深入，会逐步增加更多的实体（或`DbSet`属性）到`EbuyDataContext`类里。

如果说创建自定义数据上下文很简单，那么使用起来就更加容易了。下面的代码调用了`Create`操作来保存`Auction`对象到数据库里：只需要把`Auction`对象放到`EbuyDataContext.Auctions`集合里，然后保存修改即可。

译注1：Entity Framework 4 提供了 Code-First 模式、Model-First 模式和 Database-First 模式。Code-First 模式可以理解为“代码优先”模式。代码优先模式是 Entity Framework 的一种新开发模式，可取代现有的数据库优先模式和模型优先模式。代码优先让用户使用 CLR 类定义模型，然后将这些类映射到现有数据库或使用这些类生成数据库架构。具体文章可以参考 *http://msdn.microsoft.com/zh-cn/magazine/hh126815.aspx*。

```
[HttpPost]
public ActionResult Create(Auction auction)
{
    var db= new EbuyDataContext();
    db.Auctions.Add(auction);
    db.SaveChanges();

    return View(auction);
}
```

当运行网站并提交填写过的表单页面时，数据库里就会自动添加一条新的信息。如果再修改几次例子，尝试一些不同的数据，就会发现ASP.NET MVC模型绑定十分宽松，用户可以随便输入，只要不把表单提交的数据转换为强类型（例如，当用户输入“ABC”时就无法转换为int类型）。如果需要对保存到数据库的数据进行严格的限制，就需要使用数据验证机制。

验证数据
Validating Data

当涉及数据问题时，通常有很多规则和限制可以用，比如，某些字段不能为空、必须在特定的范围内。本质上，ASP.NET MVC把数据验证作为重要部分集成到请求处理过程中。

作为控制器操作执行的一部分，ASP.NET MVC框架验证每个传递给操作的数据是否有效。控制器操作可以通过查询ModelState来检查请求是否有效。例如，保存有效数据到数据库、后缀返回包含错误提示信息的原始表单给用户。

这里是AuctionsController.Create操作，用于判断ModelState的有效性后进行“保存或者返回”操作：

```
[HttpPost]
public ActionResult Create(Auction auction)
{
    if(ModelState.IsValid)
    {
        var db= new EbuyDataContext();
        db.Auctions.Add(auction);
        db.SaveChanges();
        return RedirectToAction("Index");
    }

    return View(auction);
}
```

尽管验证代码很有作用，但是并不是只有ASP.NET MVC框架才可以对ModelState添加验证错误，开发者可以自由添加自己的逻辑去发现那些没有被捕获的错误并添加错误验证信息。可以使用ModelState.AddModelError(string key, string message)方法来添加错误验证信息。

假设我们的交易将持续至少一天时间，换句话说，交易的截止日期比当前的日期多一天。AuctionsController.Create操作可以在保存交易之前验证这个逻辑，当然在错误的时候给出提示信息：

```
[HttpPost]
public ActionResult Create(Auction auction)
{
    if(auction.EndTime<= DateTime.Now.AddDays(1))
    {
        ModelState.AddModelError(
            "EndTime",
            "Auction must be at least one day long"
        );
    }

    if(ModelState.IsValid)
    {
        var db= new EbuyDataContext();
        db.Auctions.Add(auction);
        db.SaveChanges();
        return RedirectToAction("Index");
    }
    return View(auction);
}
```

这个方法确实有效，但是它没有分离应用程序的关注点。控制器不应该包含这种业务逻辑，该业务逻辑属于模型，所以要把业务逻辑移到模型里。

使用数据声明指定业务规则

确保数据有效性（**数据验证**）是开发人员必须面对的问题。通常情况下，开发人员都是尽可能借助现有的框架来完成数据验证工作。

这个需求非常普遍，实际上，微软提供了非常高效、便捷的数据验证API，称为**数据标注**，属于.NET框架的核心部分。物如其名，数据标注API提供了一系列.NET标记属性，开发人员可以直接标记在数据对象的属性上。这些标记属性提供了设置验证规则的声明式方式。

ASP.NET MVC模型绑定提供了数据标记支持，不需要任何额外的配置。为了学习ASP.NET MVC数据标记，我们来看下`Auction`类的验证应用过程。为了使用验证逻辑，首先要考虑什么是期望的`Auction`属性值。哪些字段是必须的？哪些字段必须在特定的范围内才算有效？

必填字段

因为auction的`Title`和`Description`对出售的商品至关重要，所以要给这两个属性标记`RequiredAttribute`，以便验证数据的有效性：

```
[Required]
public string Title { get; set; }

[Required]
public string Description { get; set; }
```

除了标记字段为必须的（Required）外，还可以确保字符长度满足了最短长度的`StringLengthAttribute`属性标记的要求。例如，如果决定了auction标题的长度最多50个字符，就可以通过这个属性来强化控制：

```
[Required, StringLength(50)]
public string Title { get; set; }
```

如果用户提交的Title长度超过50个字符，ASP.NET MVC模型验证就会失败。

验证范围

接下来要考虑商品的起始价格：由StartPrice属性表示，使用decimal类型。因为decimal是一个值类型，StartPrice默认为0，所以就没必要标记为必填字段。但是交易的起始价格有个逻辑超过了必填字段的范围，因为这个逻辑值永远不能为负值，负值意味着卖家一开始就是欠账的！为了处理这个问题，需要使用RangeAttribute属性来标记StartPrice字段，最小值是1。因为RangeAttribute需要一个最大值，所以可以指定一个最大值作为上限：

```
[Range(1, 10000]
public decimal StartPrice{ get; set; }
```

这个例子的区间使用了double类型，RangeAttribute标记属性还有一个重载(Range(Type type, string min, string max))可以满足任何实现了IComparable的类型。最好的例子就是日期范围的验证，例如，确保日期晚于某个特定的时间点：

```
[Range(typeof(DateTime), "1/1/2012", "12/31/9999"]
public DateTime EndTime{ get; set; }
```

这个例子确保EndTime属性的值晚于2012年1月1日。

.NET标记属性参数必须是编译时的值，在运行时无法修改，而且不允许使用像DateTime.Now这样的值来设置范围，必须使用固定的日期，比如1/1/2012。虽然无法保证以后的日期有效，但是至少可以保证目前的设置在有效的日期范围内。

这种缺乏准确性是一种选择使用RangeAttribute的权衡结果。如果要求有更高的准确性，就可以使用自定义验证标记属性CustomValidation-Attribute，于是就可以执行任意验证代码。虽然这种方法很强大，但是它缺少声明的方法，限制了其他组件的可用信息，比如ASP.NET MVC客户端验证框架。

自定义错误信息

最后要重点提到的是，数据标记提供了ErrorMessage属性，可以指定返回给用户的错误信息，而不是由Data Annotations API生成的默认信息。现在把使用数据标记属性的模型都指定值。

下面的类应该包含了上面讨论的所有数据标记情况：

```
public class Auction
{
  [Required]
  [StringLength(50,
    ErrorMessage = "Title cannot be longer than 50 characters")]
  public string Title { get; set; }
```

```
    [Required]
    public string Description { get; set; }

    [Range(1, 10000,
      ErrorMessage = "The auction's starting price must be at least 1")]
    public decimal StartPrice{ get; set; }

    public decimal CurrentPrice{ get; set; }
    public DateTime EndTime{ get; set; }
  }
```

既然模型里已经定义了所有的验证逻辑，那么就再看看控制器和视图是如何返回错误信息给用户的。

显示验证错误

可以通过在Create操作里设置断点来查看验证规则，只要提交无效的值，然后观察ModelState属性来验证错误是什么就可以了。事实上，控制器返回Create视图，而不是保存新的交易数据，这恰恰证明了验证规则的正确性以及验证框架的有效性。虽然“Create”视图显示了无效红边框，但还是没返回任何错误提示信息来明确告诉用户哪里错了。所以，我们需要注意这个问题。

这是显示Title属性的代码：

```
<p>
    @Html.LabelFor(model =>model.Title)
    @Html.EditorFor(model =>model.Title)
    @ViewData.ModelState["Title"]
</p>
```

我们需要做的就是添加与Title相关的验证信息。最简单的方式就是查看ModelState——通过View Data.ModelState和ViewData.ModelState["Title"]返回的与Title属性相关的错误提示信息。

我们可以迭代这个集合把错误信息渲染到页面上，代码如下：


```
<p>
    @Html.LabelFor(model =>model.Title)
    @Html.EditorFor(model =>model.Title)
    @foreach(var error in ViewData.ModelState["Title"].Errors)
    {
        <span class="error">@error.ErrorMessage</span>
    }
</p>
```

虽然这个方法不错，但是ASP.NET MVC提供了更好的方法来渲染特定属性的错误提示信息：Html.ValidationMessage(string modelName)帮助方法。它可以让我们省去上面复杂的循环代码，只需要一行调用代码就可以达到同样的效果：

```
@Html.ValidationMessageFor(model =>model.Title)
```

为模型里的每个属性都添加调用Html.ValidationMessage()方法的代码。ASP.NET MVC会在相关的控件右边渲染出与验证有关的错误提示信息。

除了使用属性级别的Html.ValidationMessage()帮助方法外，ASP.NET MVC也提供了Html.ValidationSummary()。它可以帮助我们在一个地方渲染显示验证异常信息（例如，表单的顶部）。给用户一个摘要提示信息，以方便用户修正错误，正确提交表单。

这个帮助方法使用起来也相当简单，只需要在想调用它的地方加一行代码即可：

```
@using (Html.BeginForm())
{
    @Html.ValidationSummary()

    <p>
        @Html.LabelFor(model =>model.Title)
        @Html.EditorFor(model =>model.Title)
        @Html.ValidationMessageFor(model =>model.Title)
    </p>

    <!--表单的其余字段-->
}
```

当用户提交无效的值时，我们可以在两个地方看到错误提示信息，即顶部的汇总信息（调用Html.ValidationSummary()）和控件后边的错误提示信息（调用Html.ValidationMessage()），如图3-1所示。

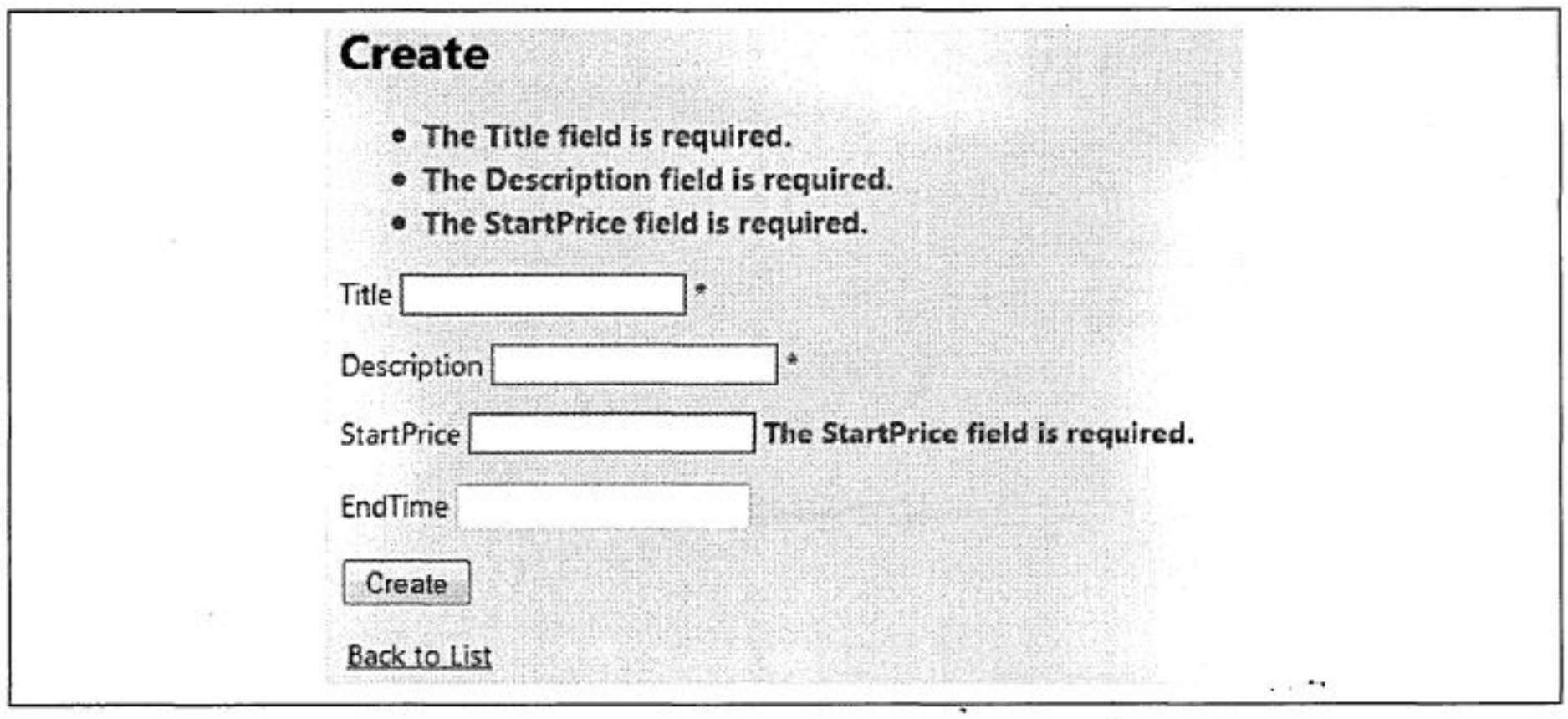


图3-1 通过Html.ValidationSummary()帮助方法显示错误

如果想避免重复显示错误信息，则可以修改调用Html.ValidationMessage()的代码，指定更短的自定义错误提示信息，比如简单的星号*，如下：

```
<p>
    @Html.LabelFor(model =>model.Title)
    @Html.EditorFor(model =>model.Title)
    @Html.ValidationMessageFor(model =>model.Title, "*")
</p>
```

下面是使用了验证标签后的Create视图文件的所有代码：

```
<h2>Create Auction</h2>
```

```
@using (Html.BeginForm())
{
    @Html.ValidationSummary()

    <p>
        @Html.LabelFor(model =>model.Title)
        @Html.EditorFor(model =>model.Title)
        @Html.ValidationMessageFor(model =>model.Title, "*")
    </p>
    <p>
        @Html.LabelFor(model =>model.Description)
        @Html.EditorFor(model =>model.Description)
        @Html.ValidationMessageFor(model =>model.Description, "*")
    </p>
    <p>
        @Html.LabelFor(model =>model.StartPrice)
        @Html.EditorFor(model =>model.StartPrice)
        @Html.ValidationMessageFor(model =>model.StartPrice)
    </p>
    <p>
        @Html.LabelFor(model =>model.EndTime)
        @Html.EditorFor(model =>model.EndTime)
        @Html.ValidationMessageFor(model =>model.EndTime)
    </p>
    <p>
        <input type="submit" value="Create" />
    </p>
}
```

目前展示的验证都是服务端代码验证，需要在服务器和客户端之间进行几回合的交互过程才能验证数据的有效性，并返回渲染过的视图结果。

这个方法虽然有效，但不是必须的。第4章展示了如何实现客户端验证与性能优化——不是所有的验证都在浏览器里进行。这可以避免不必要的请求，节约带宽和服务器资源。

总结
Summary

本章讨论了如何使用Entity Framework代码优先方法创建应用程序，并实现与数据库相关的操作。我们看到了使用Entity Framework的强大之处：只需几行代码，不需要编写复杂的架构图或者SQL语句就可以实现数据库操作。我们也讨论了Entity Framework如何遵守“惯例优于配置原则”来自动工作，以及一些简单的惯例原则。后面同样也讨论了如何使用ASP.NET MVC的模型绑定功能从请求消息中提取状态对象。最后介绍了如何验证数据并提供错误提示信息。

第4章

客户端开发

Client-Side Development

互联网使用HTML和JavaScript开发网页的历史由来已久。非常流行的Web应用如Gmail（译注1）和Facebook已经提升了用户的期望：用户不再只满足于简单的文本内容，他们渴望类似于桌面应用一样的强大的Web应用体验。

虽然很多书籍关注于如何使用ASP.NET MVC框架进行服务端开发，但本章我们还是稍做调整，先来学习Web应用开发，以及如何使用jQuery库简化客户端开发。

使用JavaScript

Working with JavaScript

浏览器的兼容性问题已经困扰Web开发人员很久了。各项浏览器功能以及标准规范的差异逼迫各种客户端库和框架通过隐藏各个浏览器的差别来提供一种真正的、标准的跨浏览器API。在众多的库中，最受欢迎的就是jQuery JavaScript库，它的口号为“Write less, Do more”，大大简化了HTML DOM的遍历、事件处理、动画以及Ajax交互过程。ASP.NET MVC 3开发把jQuery添加到了默认的项目模板中，这让我们的开发工作更加方便。

为了了解jQuery如何封装抽象浏览器的差异性，我们需要看下面的代码，这个代码要找出浏览器窗口的宽度和高度：

```
var innerWidth = window.innerWidth,
    innerHeight = window.innerHeight;
alert("InnerWidth of the window is: " + innerWidth);
alert("InnerHeight of the window is: " + innerHeight);
```


这个脚本在大部分浏览器里都可以显示窗口的高度和宽度，但是在IE6~8里就会出错。为什么呢？这是因为这些版本的IE浏览器提供了另外的方式`document.DocumentElement.clientWidth`和`document.DocumentElement.clientHeight`。

译注1：如果你是初学者，有必要知道，AJAX最早由微软Out Look Web Access团队发明。Ajax技术的流行很大程度上要感谢谷歌的Gmail。此外可能有些人对Facebook疑问很大，这是个什么网站？为什么我从来上不去？这不是你的错。这是一个社交类SNS网站，是哈佛大学的学生马克·扎克伯格和几位朋友创建的。这个网站被国内多家网站“山寨”。

所以，为了让代码在各个浏览器里正常工作，就必须处理IE的不一致性，如下所示：

```
varinnerWidth, innerHeight;

// IE9 及更新版本浏览器
if (typeofwindow.innerWidth!== "undefined") {
  innerWidth= window.innerWidth;
  innerHeight= window.innerHeight;
}
else{
  innerWidth= document.documentElement.clientWidth,
  innerHeight= document.documentElement.clientHeight
}

alert("InnerWidth of the window is: " + innerWidth);
alert("InnerHeight of the window is: " + innerHeight);
```

重新修改代码以后，就可以在不同浏览器里工作了。

W3C标准的不一致性导致了很多问题，旧版本的浏览器大部分都有这种问题，它们只部分支持一些标准。当像HTML5和CSS3这样的新规范还在草案阶段时，新的浏览器都通过自己的内核引擎提供了支持。想象一下，如果网站里每个元素都要对不同的浏览器进行兼容，该有多么恐怖！不仅代码会变得极其臃肿，而且需要及时对每个浏览器和标准的更新进行兼容。这是每个网站开发和维护人员的噩梦。

解决这个问题的最好办法就是在DOM操作和Web应用之间使用特定的客户端开发框架作为隔离层。jQuery就是一个轻量级的框架，它很好地解决了这些问题。而且jQuery API大大简化了操作DOM对象的复杂性，让开发人员更多地关注应用程序的功能，而不是浏览器的兼容性。

下面的代码中使用jQuery进行编写：

```
var innerWidth = $(window).width(),
    innerHeight = $(window).height();
alert("InnerWidth of the window is: " + innerWidth);
alert("InnerHeight of the window is: " + innerHeight);
```

这些代码与原生JavaScript代码很像，只做了极小的变换：

- `Window`对象封装在`$()`函数里，返回一个jQuery对象（后面将会详细介绍）。
- 代码直接调用`.width()`和`.height()`函数，而不是访问`.height`和`.width`属性。

可以看到使用jQuery的好处——代码与传统的JavaScript很像，这样也更加方便大家学习和使用，而且它很好地解决了跨浏览器的兼容性问题。更重要的是，有了jQuery的帮助，程序员可以专心来实现功能。

jQuery不仅简化了获取属性值的过程，而且也很方便赋值：

```
// 设置为 480 像素高
```

```
$(window).width("480px");

// 设置为 940 像素高
$(window).height("940px");
```

注意：相同的函数可用来设置和读取属性值，唯一的区别就是是否带有新值参数。这种API的使用方式也让jQuery的语法变得更易记、易读。

选择器
Selectors

操作DOM元素的第一步就是获取该元素的引用。可以通过ID、class名、属性，或者使用JavaScript逻辑代码遍历DOM树结构后定位元素等多种方式来实现。

例如，下面的代码展示了如何使用标准的JavaScript代码根据ID查询DOM元素：

```
<div id="myDiv">Hello World!</div>

<script type="text/javascript">
  document.getElementById("myDiv").innerText= "Hello jQuery";
</script>
```

这个简单的例子通过`document.getElementById()`方法根据ID获取到 元素，然后修改内部文本内容为“Hello jQuery”。这段代码可以在任何浏览器里运行，因为`document.getElementById()`是JavaScript语言的一部分，大部分主流浏览器都支持。

考虑另外的情况，即当需要通过class名称来访问元素时：

```
<div class="normal">Hello World!</div>

<script type="text/javascript">
  document.getElementsByClassName("normal")[0].innerText= "Hello jQuery";
</script>
```

看起来很简单——除了使用`document.getElementById()`，还可以使用`document.getElementsByClassName()`方法去访问集合里的第一个元素。但是，这个集合从哪里来？`document.getElementsByClassName()`方法会返回符合class条件的所有元素集合。幸运的是，上面的例子里只有一个 元素，所以第一个就会要找想要的元素。

在真实的Web程序里，通常页面都会包含几个带有或者没有ID的元素，而且可能多个元素使用同样样式class（有的元素没有样式class）。为了页面设计的需要，元素会驻留在元素容器里，比如在 、<p>和里。因为DOM就是一个树形文档结构，所以我们能够获得一层套一层的元素，而且都嵌套在根元素document里。

考虑下面的例子：

```
<div class="normal">
  <p>Hello World!</p>
```

```
    <div>
      <span>Welcome!</span>
    </div>
  </div>
```

为了获取并改变它的内容，需要继续获取外围的 (class="normal")，遍历它的所有子节点，判断它是不是，然后再进行操作。

使用JavaScript获取的典型代码如下：

```
var divNode= document.getElementsByClassName("normal")[0];
for(i=0; i<divNode.childNodes.length; i++) {
  var childDivs= divNode.childNodes[i].getElementsByTagName("div");
  for(j=0; j <childDivs.childNodes.length; j++) {
    var span = childDivs.childNodes[j].getFirstChild();
    return span;
  }
}
```

这么多代码就是为了获取一个元素！如果要访问<p>标签呢？太头疼了！这些代码可以重用吗？当然不行，因为<p>元素位于不同的树形节点。我们需要重新写一遍类似的代码来操作<p>元素，或者添加条件以便查找。如果在 内部，又该怎么做？要解决这些问题，会让代码越来越庞大、臃肿，因为还要添加各种逻辑条件。

如果想要在不同的地方重用这种代码，或者将来因修改HTML标签而影响了结构就必须重写、调整函数代码，那么很显然，这种方式只会让代码变得笨重、臃肿，错误不断。

jQuery选择器可以帮助我们解决这些头疼的问题。通过支持惯例优于配置原则就可以使用很少的代码来遍历DOM。让我们看一下它是如何实现这种功能，且帮助我们简化代码的。

下面是使用jQuery选择器重写的遍历逻辑代码：

```
$("#myDiv").text("Hello jQuery!");
```

这里调用了jQuery’的`$()`函数，根据预定义的模式传参。#表示ID选择器，所以`#myDiv`等价于`document.getElementById("myDiv")`。

一旦获得了元素的引用，就可以通过jQuery的`text()`方法来修改内容。这与设置`innerText`属性一样，但是更加简单。

这里要提到的一个有趣的问题就是：jQuery的所有方法都返回一个jQuery对象，里面封装了原生的DOM元素。这种封装允许链式调用，比如可以在一条语句里修改文本和颜色：

```
$(".normal > span")          // 返回一个 jQuery 对象
  .contains("Welcome!")      // 再次返回一个 jQuery 对象
  .text("...")               // 又返回一个 jQuery 对象
  .css({color: "red"});
```

因为每个调用方法（`.text()`，`.css()`）返回的都是jQuery对象，所以调用可以连续进行。这种风格的调用方式称为“链式调用”（chaining of calls），代码看起来十分流畅、自然，更具备可读性，还减少了代码量。

与ID模式一样，选择器还可以使用样式class名：

```
$(".normal").text("Hello jQuery!");
```

这个基于class的选择器是“.className”。

getElementsByClassName()返回的是集合。此时，jQuery会改变所有集合元素的文本内容！所以，如果有多个元素包含相同的样式名，就要进行更严格的筛选以便找到正确的元素。

现在，看一下如何使用父子关系来访问元素。查找的例子是这样的：

```
$(".normal > span").text("Welcome to jQuery!");
```

“>”表示父子关系。甚至可以据此来过滤别的元素：

```
$(".normal > span").contains("Welcome!").text("Welcome to jQuery!");
```

.contains()筛选出包含特定文本的元素。所以，如果有多个span，在没有ID和Class名的情况下，唯一区分它们的方式就是内容。jQuery选择器让一切简单起来！

jQuery提供了更多的选择器模式。要详细学习这些内容，可以查阅jQuery的官方文档。

处理事件

Responding to Events

HTML页面里的每个元素都可以激发事件，比如点击事件“click”、移动鼠标“mouse move”、修改“change”等。事件是方便页面交互的强大机制，可以侦听事件，然后执行更多的操作，增强用户体验。

例如，对于一个带有多个元素的Form表单，可通过侦听提交按钮的onClick事件来验证用户的输入信息是否有效，并提示错误信息，而不需要完全刷新页面。

添加一个按钮，当用户点击按钮时弹出“hello events!”的传统HTML/JavaScript，代码如下：

```
<input id="helloButton" value="Click Me" onclick="doSomething();">

<script type="text/javascript">
  function doSomething() {
    alert("hello events!");
  }
</script>
```

事件处理函数在HTML页面中定义onclick="doSomething();"，当点击按钮时，代码就弹出标准的包含“hello events!”的消息框。

也可以用代码方式注册事件：

075

```
<input id="helloButton" value="Click Me">

<script type="text/javascript">
  function doSomething() {
    alert("hello events!");
  }

  document.getElementById("helloButton").onclick= doSomething;
</script>
```

注意，HTML标签没有指定onclick行为。这里分离了展示层和行为，行为在展示代码之外附加。这样做不仅让代码更洁净，而且确保了展示层和行为代码的重用。

这是一个展示事件处理函数如何工作的例子。真实的项目中，JavaScript函数会更加复杂，而且可能不会给用户弹出警告信息。

现在来看一下jQuery如何绑定事件处理函数：

```
<input id="helloButton" value="Click Me">

<script type="text/javascript">
  function doSomething() {
    alert("hello events!");
  }

  $(function() {
    $("#helloButton").click(doSomething);
  });
</script>
```

使用jQuery，首先要使用`$("#helloButton")`获取按钮，然后调用.click()来附加事件处理函数，`.click()`可以简化为`.bind("click", handler)`。

`$(function)`告诉jQuery当浏览器在加载DOM的时候附加事件处理函数。记住，DOM树是从上往下加载的。

只要DOM树加载完脚本、样式和其他资源，浏览器就会触发`window.onload`事件。`$()`会侦听这个事件，并执行附加的元素处理函数（实际上是事件处理函数）。

换句话说，`$(function(){…})`是jQuery的编码方式：

```
window.onload= function() {
  $("#helloButton").click(function() {
    alert("hello events!");
  });
}
```

076

也可以这样指定事件处理函数（这被称为inline内联方式）：

```
$(function() {
  $("#helloButton").click(function() {
    alert("hello events!");
  });
});
```

有意思的是，如果没有指定参数给.click()，它就会真的触发一次点击事件。这在动态编程实现点击按钮时很有用：

```
$("#helloButton").click(); // 显示“hello events!”
```

DOM操作
DOM Manipulation

jQuery提供了更加简单、强大的操作DOM的机制。

例如，修改元素的CSS属性：

```
// 修改按钮的颜色为红色
$("#helloButton").css("color", "red");
```

我们已经看过这种代码了，还记得前面例子里获取高度“height”的例子吗？

```
// 返回元素的高度
var height = $("#elem").height();
```

除了这种操作之外，jQuery还允许我们从元素集合里创建、取代和删除元素。

下面的例子演示了如何向现有的 元素添加一个元素集合：

```
<div id="myDiv">
</div>

<script type="text/javascript">
  $("#myDiv").append("<p>I was inserted <i>dynamically</i></p>");
</script>
```

结果就是：

```
<div id="myDiv">
  <p>I was inserted <i>dynamically</i></p>
</div>
```

删除元素也很简单（删除集合也一样）：

```
<div id="myDiv">
  <p>I was inserted <i>dynamically</i></p>
</div>

<script type="text/javascript">
  $("#myDiv").remove("p"); // 删除<p>及其子元素
</script>
```

结果如下：

```
<div id="myDiv">
</div>
```

077

jQuery提供了几种方法去设置标签元素的位置，如表4-1所示。

表4-1　常见的DOM操作方法

方法	描述
.prepend()	插入到匹配元素的头部
.before()	插入到匹配元素之前
.after()	插入到匹配元素之后
.html()	替换匹配元素内部的HTML代码

AJAX

AJAX（异步JavaScript和XML）（译注2）是一种允许页面发送和提交数据而不需要完全刷新或回传页面的技术。

使用异步方式请求数据增强了用户体验，因为用户不需要再等待整个页面的加载，而只请求部分数据，也减少了服务器压力，提升了响应的速度。

AJAX技术的核心就是`XmlHttpRequest`对象，它最早由微软为Outlook邮件客户端程序开发的，是为了让Outlook通过Web方式访问Exchange Server 2000邮件服务器。后来很快就被行业巨头采纳，并提供支持，如Mozilla（火狐）、Google（谷歌）、Apple（苹果），现在已经成了W3C标准。

典型的使用`XmlHttpRequest`对象实现AJAX请求（译注3）的代码如下：

```
// 实例化 XmlHttpRequest 对象
var xhr= new XMLHttpRequest();

// 使用 Get 方式请求谷歌页面
xhr.open("GET", "http://www.google.com/", false);

// 发送请求
xhr.send(null);

if (xhr.status=== 200) { // 200 状态码表示正常

  // 只支持(Firefox、Chrome、IE 8+)
```

译注2：AJAX 不是新技术，是对现有技术 JavaScript 和 XML 的组合。2005 年 2 月，Jesse James Garrett 在其文章《A New Approach to Web Applications》里首次提出 AJAX 的概念。后因谷歌 Gmail 等 Web 应用的大量使用而流行起来。我在微软 WebCast 课程“WCF 与 AJAX 开发实践(1)AJAX 基本原理与纯 AJAX 示例”中有介绍。

译注3：XmlHttpRequest 实现 AJAX 的代码的问题在微软 WebCast 课程“WCF 与 Ajax 开发实践(1)Ajax 基本原理与纯 Ajax 示例”中有更加详细的介绍。

```
    console.log(xhr.responseText);
  }
  else { // 记录错误
        console.log("Error occurred: ", xhr.statusText);
  }
```

这个例子发送了一个**同步**请求（第三个参数在xhr.open()里），这意味着浏览器必须等待返回结果以后再进行处理工作。有时候想避免这种同步的AJAX请求，因为页面在完全返回数据以后才有反应的用户体验很差。

幸运的是，我们很容易把同步请求改成异步请求：只需要简单设置xhr.open()的第三个参数为true即可。异步机制的特性能让浏览器不需要等待，可以继续执行代码。因为请求可能不完整，所以这很可能出错。

为了处理这种情况，必须指定**回调**函数——当收到请求时立即执行。

看一下修改后的代码：

```
// 实例化 XmlHttpRequest 对象
var xhr= new XMLHttpRequest();

// 使用 get 方式获取 Google 首页
xhr.open("GET", "http://www.google.com/", true);

// 添加回调函数
xhr.onreadystatechange= function (evt) {

  // 因为请求状态不同，
  // readyState 的值会改变
  // 这个函数会在每次变化的时候调用
  // readyState=4 意味着处理完成
  if(xhr.readyState=== 4) {
    if(xhr.status=== 200) {
      console.log(xhr.responseText)
    }
    else{
      console.log("Error occurred: ", xhr.statusText);
    }
  }
};

// 发送空内容请求
xhr.send(null);
```

除了回调函数，这个例子的代码几乎和同步版本的代码一样。

必须在xhr.send()之前设置回调函数，否则无法调用。

让我们看一下jQuery实现的等价代码。jQuery提供了.ajax()方法来完成同样的AJAX工作。

这里是jQuery版本的AJAX代码：

```
$.ajax("google.com") // 获取谷歌首页
  .done(function(data) { // 状态码 200 表示成功
      console.log(data);
    })
  .fail(function(xhr) { // 错误处理，不是 200
    console.log("Error occurred: ", xhr.statusText);
    });
```

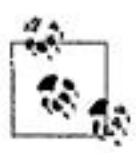
第一行代码指定了请求数据的地址，接着是请求成功和错误时的回调函数（jQuery通过检查readyState和状态码实现）。

注意，为什么我们不需要指定请求的类型（Get或者Post），也不需要指定是否是异步请求？这是因为jQuery的$.ajax()默认就是Get和异步请求方式。

你也可以重写这些代码以规范请求：

```
$.ajax({
  url: "google.com",
  async: true, // 表示同步请求
  type: "GET", // Get 或者 Post 方式，默认 Get 方式
  done: function(data) { // 状态码 200 表示成功
          console.log(data);
        },
  fail: function(xhr) { // 状态码非 200 表示失败
          console.log("Error occurred: ", xhr.statusText);
        }
});
```

jQuery AJAX提供了更多的参数。这里只做了简要介绍，如果想深入学习，可以阅读jQuery的官方文档。



jQuery 1.8引入了.done()方法和.fail()方法。如果使用旧版本的jQuery，可以使用.success()和.error()代替。

客户端验证
Client-Side Validation

第3章里介绍了服务端验证方法。本节将会介绍如何在jQuery及其验证插件帮助下实现等价的客户端验证，以增强用户体验。

ASP.NET MVC框架从第三版开始提供了客户端验证支持。默认是启用的，当然也可以在配置文件里设置启用和关闭这项功能：

```
<configuration>
  <appSettings>
    <add key="ClientValidationEnabled" value="true"/>
    <add key="UnobtrusiveJavaScriptEnabled" value="true"/>
  </appSettings>
</configuration>
```

使用jQuery验证插件的一大好处就是它也可以继续使用模型定义里的DataAnnotation数据标记属性，这样就可以很方便地使用它了。

回顾第3章的Auction模型，看看我们设置用来进行验证的数据标记属性：

```
public class Auction
{
    [Required]
    [StringLength(50,
     ErrorMessage = "Title cannot be longer than 50 characters")]
    public string Title { get; set; }

    [Required]
    public string Description { get; set; }

    [Range(1, 10000,
     ErrorMessage = "The auction's starting price must be at least 1")]
    public decimal StartPrice{ get; set; }

    public decimal CurrentPrice{ get; set; }
    public DateTime EndTime{ get; set; }
}
```

这里是渲染出来的验证消息：

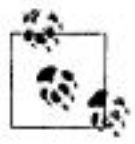
```
<h2>Create Auction</h2>

@using (Html.BeginForm())
{

  @Html.ValidationSummary()

  <p>
     @Html.LabelFor(model =>model.Title)
     @Html.EditorFor(model =>model.Title)
     @Html.ValidationMessageFor(model =>model.Title, "*")
  </p>
  <p>
     @Html.LabelFor(model =>model.Description)
     @Html.EditorFor(model =>model.Description)
     @Html.ValidationMessageFor(model =>model.Description, "*")
  </p>
  <p>
     @Html.LabelFor(model =>model.StartPrice)
     @Html.EditorFor(model =>model.StartPrice)
     @Html.ValidationMessageFor(model =>model.StartPrice)
  </p>
  <p>
     @Html.LabelFor(model =>model.EndTime)
     @Html.EditorFor(model =>model.EndTime)
     @Html.ValidationMessageFor(model =>model.EndTime)
  </p>
  <p>
     <input type="submit" value="Create" />
  </p>
}
```

执行的验证很简单，若使用服务端验证，则还需要回传服务器。输入数据在服务端验证，如果出现错误，消息还要被回传给客户端，并进行页面刷新才能显示给用户。

使用客户端验证，在提交的时候就可以验证输入，所以，无需提交给服务器，也不需要页面刷新，结果会立即显示给用户。

进行客户端验证，需要在视图文件里引用jQuery验证插件：

```
<script src="@Url.Content("~/Scripts/jquery.validate.min.js")"
      type="text/javascript"></script>
<script src="@Url.Content("~/Scripts/jquery.validate.unobtrusive.min.js")"
      type="text/javascript"></script>
```



如果使用Visual Studio的添加视图向导生成Create和Edit视图文件，则可以选择是否添加脚本库引用，Visual Studio会自动帮助我们完成。Visual Studio也可以使用~/bundles/jquery-val来达到同样的效果，它会把脚本文件绑定在视图底部。参见“绑定和压缩”内容。

如果现在运行网站查看创建Auction的页面代码（使用页面“查看源码”），就可以看到如下HTML标签（启用了独立的JavaScript 和客户端验证支持）：

```
<form action="/Auctions/Create" method="post" novalidate="novalidate">
  <div class="validation-summary-errors" data-valmsg-summary="true">
    <ul>
      <li>The Description field is required.</li>
      <li>The Title field is required.</li>
      <li>Auction may not start in the past</li>
    </ul>
  </div>
  <p>
    <label for="Title">Title</label>
    <input class="input-validation-error"
      data-val="true"
      data-val-length="Title cannot be longer than 50 characters"
      data-val-length-max="50"
      data-val-required="The Title field is required."
      id="Title" name="Title" type="text" value="">

    <span class="field-validation-error"
      data-valmsg-for="Title"
      data-valmsg-replace="false">*</span>
  </p>
  <p>
    <label for="Description">Description</label>

  <input class="input-validation-error"
      data-val="true"
      data-val-required="The Description field is required."
      id="Description" name="Description" type="text" value="">

    <span class="field-validation-error"
      data-valmsg-for="Description"
      data-valmsg-replace="false">*</span>
```

```
    </p>
    <p>
      <label for="StartPrice">StartPrice</label>

      <input data-val="true"
        data-val-number="The field StartPrice must be a number."
        data-val-range="The auction's starting price must be at least 1"
        data-val-range-max="10000"
        data-val-range-min="1"
        data-val-required="The StartPrice field is required."
        id="StartPrice" name="StartPrice" type="text" value="0">

      <span class="field-validation-valid"
        data-valmsg-for="StartPrice"
        data-valmsg-replace="true"></span>
    </p>
    <p>
      <label for="EndTime">EndTime</label>

      <input data-val="true"
        data-val-date="The field EndTime must be a date."
        id="EndTime" name="EndTime" type="text" value="">

      <span class="field-validation-valid"
        data-valmsg-for="EndTime"
        data-valmsg-replace="true"></span>
    </p>
    <p>
      <input type="submit" value="Create">
    </p>
  </form>
```

启用JavaScript客户端验证成功后，ASP.NET MVC就会在data-val-存储对应的验证条件和提示消息。jQuery验证插件会使用这些值来确定验证规则和对应的错误提示信息。

输入无效的数据，就会发现页面表单没有提交到服务端，而且会看到错误提示信息。

在此背后，jQuery验证插件会给表单的onsubmit添加事件处理函数。一旦表单提交，jQuery就会扫描所有的输入控件并根据规则来验证数据。发现错误时，就会显示错误提示消息。

使用Unobtrusive JavaScript（分离JavaScript）意味着既不需要在页面里嵌入jQuery验证代码，也不需要在页面里绑定客户端验证事件。相反，代码会自动添加onsubmit事件以及验证逻辑，这是jquery.validate.js和jquery.validate.unobtrusive.js文件的一部分。

使用Unobtrusive JavaScript（译注4）的好处就是，如果忘记了引用这2个脚本，则页面仍然可以正常渲染，不会出错。只有客户端验证无法正常工作时才会出错。

译注4：Unobtrusive JavaScript是一种新兴的技术，提倡将JavaScript从HTML标记语言中分离出来，类似于20世纪90年代CSS的诞生所带来的页面样式和HTML代码分离。

本节旨在告诉用户如何使用客户端验证，以及如何保持简单、紧凑的代码。jQuery验证控件提供了许多特性和自定义功能。我们可以从其官方网站里学习更多知识。

总结
Summary

jQuery使得跨浏览器开发异常简单。ASP.NET MVC本身就支持jQuery，这意味着我们使用很少的代码就可以快速构建高交互性的客户端UI。使用JavaScript进行客户端验证可以高效验证用户输入。所有这些客户端技术整合在一起就可帮助我们快速开发强大的Web应用。

第二部分

欲穷千里目，更上一层楼

Going to the Next Level

第5章 Web应用程序架构

Web Application Architecture

前几章介绍了使用ASP.NET MVC框架所必须了解的几个核心概念。本章会涵盖这些核心概念，详细介绍这些基本设计模式（design pattern）和用于构建ASP.MVC框架的原则，并探讨如何应用这些模式和原则（patterns and principles）来构建ASP.NET MVC Web应用程序。

模型-视图-控制器模式

The Model-View-Controller Pattern

模型-视图-控制器（MVC）模式（译注1）是一种用户界面架构模式，促进跨多个应用程序层的分离关注点（separation of concerns）。MVC模式不提倡将所有逻辑和数据访问代码的应用程序放在一个地方，提倡把应用程序逻辑封装到特定的类中，每个类都有自己特定的职责。

MVC模式并不是一个新概念，它既不依赖特定的.NET框架，也不依赖Web开发。事实上，它最初是在20世纪70年代后期由Xerox parc程序员创建的和应用于使用Smalltalk语言的应用程序。从那以后，许多人认为MVC是有史以来最好的应用架构模式之一。这是因为MVC模式是一种非常高级别的模式，它已经形成了许多相关的实现框架和子模式。

分离关注点

分离关注点属于计算机科学原则。这条原则提倡将跨多个组件的应用程序或用例责任分离开来，每个组件都有自己的特定的职责。“关注”可以用一组特定的功能或行为相关联。分离关注点传统上通过使用实现封装和抽象来更好地隔离其他组件的关注点。例如，分离关注点可以通过划分表示层（presentation）、业务逻辑层（business logic）和数据访问层(data access)来设计应用程序架构，每个层在逻辑上和物理上都是分离的。

分离关注点是一条经常用于设计平台和框架的强大原则。例如，在早期的Web网页中，经常把布局、样式和数据放在同一文档中。许多年后，出现了基于分离关注点的标准，现在把单个文档分为三个部分：一个HTML文档，主要侧重于内容结构上；一个或多个CSS样式表，负责定义该文档的样式；JavaScript文件，负责控制文档内容的行为。在MVC模式中，分离关注点用来定义关键组件的责任：模型、视图和控制器。图5-1展示了不同的MVC组件及其核心责

译注1：Model-View-Controller (MVC) pattern 简称为 MVC 模式。1974 年，Trygve Reenskaug（特里夫·里斯高格）为 Smalltalk 语言提出这种架构模式。

任之间的相互作用。

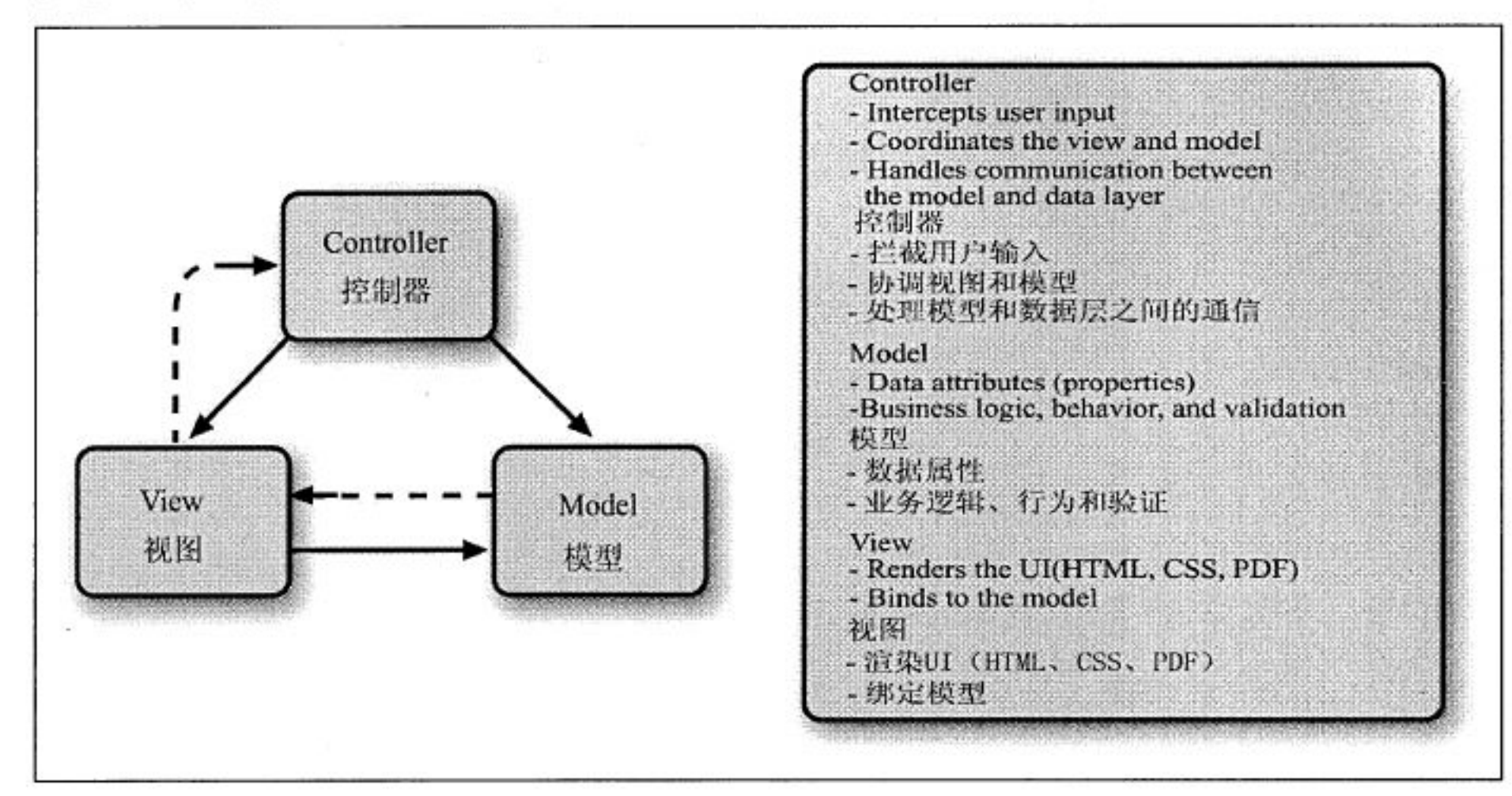


图5-1　MVC分离关注点

至关重要的是，虽然每个单独的组件有其自己的责任，但是组件可以互相依赖。例如，在MVC中，控制器负责从模型中检索数据，并且把视图中的更改同步回模型中。控制器可以与一个或多个视图相关联。每种视图以特定的方式负责显示数据，但它依赖控制器来处理和检索数据。虽然组件可能会严重依赖其他的组件，但是每个组件只专注于自己的责任，并且把其他的责任留给别的组件处理。

MVC与Web框架

最初设计MVC模式的时候，假设视图、控制器和模型在同一上下文环境中。该模式严重依赖每个能够彼此直接交互、共享用户状态的组件。例如，控制器将使用观察者模式监视要更改的视图，并对用户输入作出反应。当控制器、视图和模型都存在于相同的内存上下文环境中的，这种方法非常有效。

089

在Web应用程序中，所有的东西都是无状态的，而且视图（HTML）运行在客户端的浏览器内。控制器不能使用观察者模式监视视图的变化；相反，一个HTTP请求需要从视图发送到控制器。若要解决此问题，可以使用前端控制器模式，如图5-2所示。这种模式的主要理念是当发送一个HTTP请求时，控制器截获并处理它。控制器负责确定如何处理该请求并将结果发送回客户端。

当现在的Web应用程序要为多个请求执行相同的逻辑代码时，虽然可能会为每个单独的请求返回不同的内容，但前端控制器模式的优势还是非常明显。例如，相同的控制器操作可以处理正常的浏览器请求和AJAX请求，但浏览器希望呈现的是完整的网页（如布局、样式、脚本文件等），而AJAX请求期望的可能是部分HTML视图甚至是原始JSON数据。在所有这些例子中，控制器逻辑保持不变，而且视图仍然不知道数据是从哪里来的。

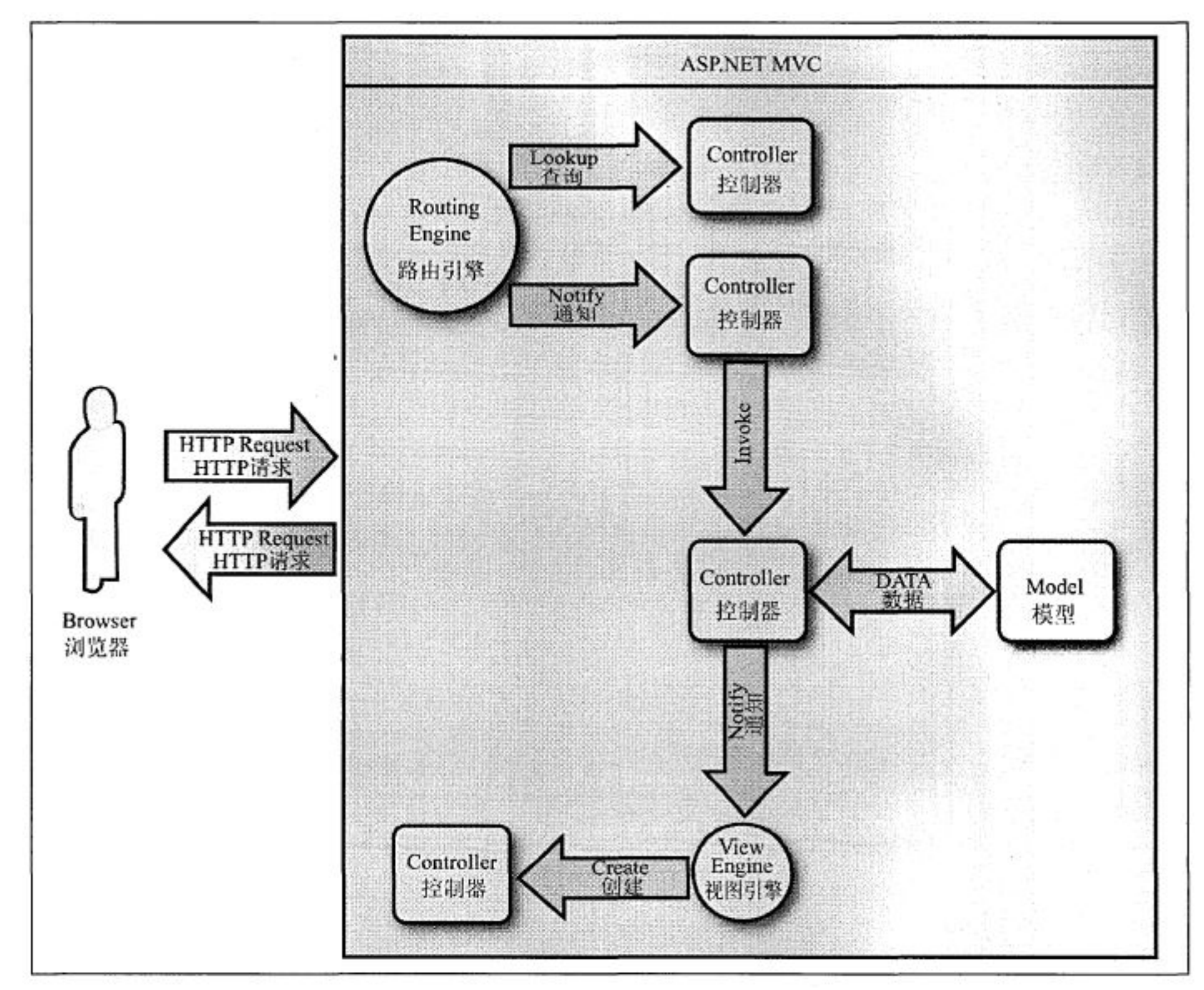


图5-2　前端控制器模式

在ASP.NET MVC应用程序中，路由和视图引擎涉及处理HTTP请求的工作。当收到请求URI时（如*/auctions/detail/25*），ASP.NET MVC运行库会在路由表中查找，并调用相应的控制器操作。控制器会处理该请求，并确定模型返回什么结果类型。当返回视图结果时，ASP.NET MVC框架会把加载和呈现相应请求视图的任务委派给视图引擎。

Web应用架构设计

Architecting a Web Application

ASP.NET MVC框架的核心设计依据就是分离关注点原则。除了路由和视图引擎以外，ASP.NET MVC框架提倡使用操作过滤器，用于处理横切关注点（cross-cutting concerns）（译注2），例如，安全、缓存和错误处理。当设计和构建一个ASP.NET MVC Web应用程序时，

译注2：横切关注点（cross-cutting concerns）是指相对于自上而下的请求流程，应用程序中有一些必须面对的问题，如安全、日志、缓存和错误处理等。这些问题就像在一个长条形上的物体，如在黄瓜上横向切了一刀之后我们看到的横切面。于是这些问题就被称为“横切关注点”。

重要的是要了解这个MVC框架如何支持这条原则，以及如何使用这一原则来设计应用程序。

逻辑设计

应用程序的逻辑（概念）架构重点关注组件之间的关系和交互，以及这些组件如何在逻辑层中实现特定的功能。

组件的设计目标就是强化分离关注点以及使用抽象跨组件通信。横切关注点是将安全、日志和缓存等隔离到不同的应用程序服务中。这些服务应该支持即插即用的模块方法（plug-and-play module approach）。切换不同的安全身份验证类型或使用不同的日志记录来源将不会影响应用程序的其他部分。

ASP.NET MVC Web应用程序的逻辑设计

ASP.NET MVC框架的设计目标就是提倡这种类型的逻辑设计。除了隔离视图、控制器和模型以外，ASP.NET MVC框架还包括处理不同类型横切关注点的几个操作过滤器，以及为视图、JSON、XML、部分页面定义多种操作结果类型。因为该框架支持无限的扩展，所以开发人员可以创建并插上自己的自定义操作过滤器和结果。

SingleSignOnAttribute标记属性就是一个自定义的操作过滤器例子，它支持跨多个ASP.NET Web应用程序的单点登录身份验证。代码如下：

```
public class SingleSignOnAttribute : ActionFilterAttribute, IActionFilter
{
    void OnActionExecuted(ActionExecutedContext filterContext)
    {
        // 检查安全令牌并验证用户
    }

    void OnActionExecuting(ActionExecutingContext filterContext)
    {
        // 用来检验安全令牌是否存在的预处理代码
    }
}
```

应用程序逻辑设计通信的最佳方法是为每个组件及其相应的层创建可视化表示形式。图5-3所示为典型的逻辑设计的ASP.NET MVC Web应用程序。注意横切关注点是如何分隔成不同的应用程序服务的。

表5-1介绍了图5-3所示中的不同元素。

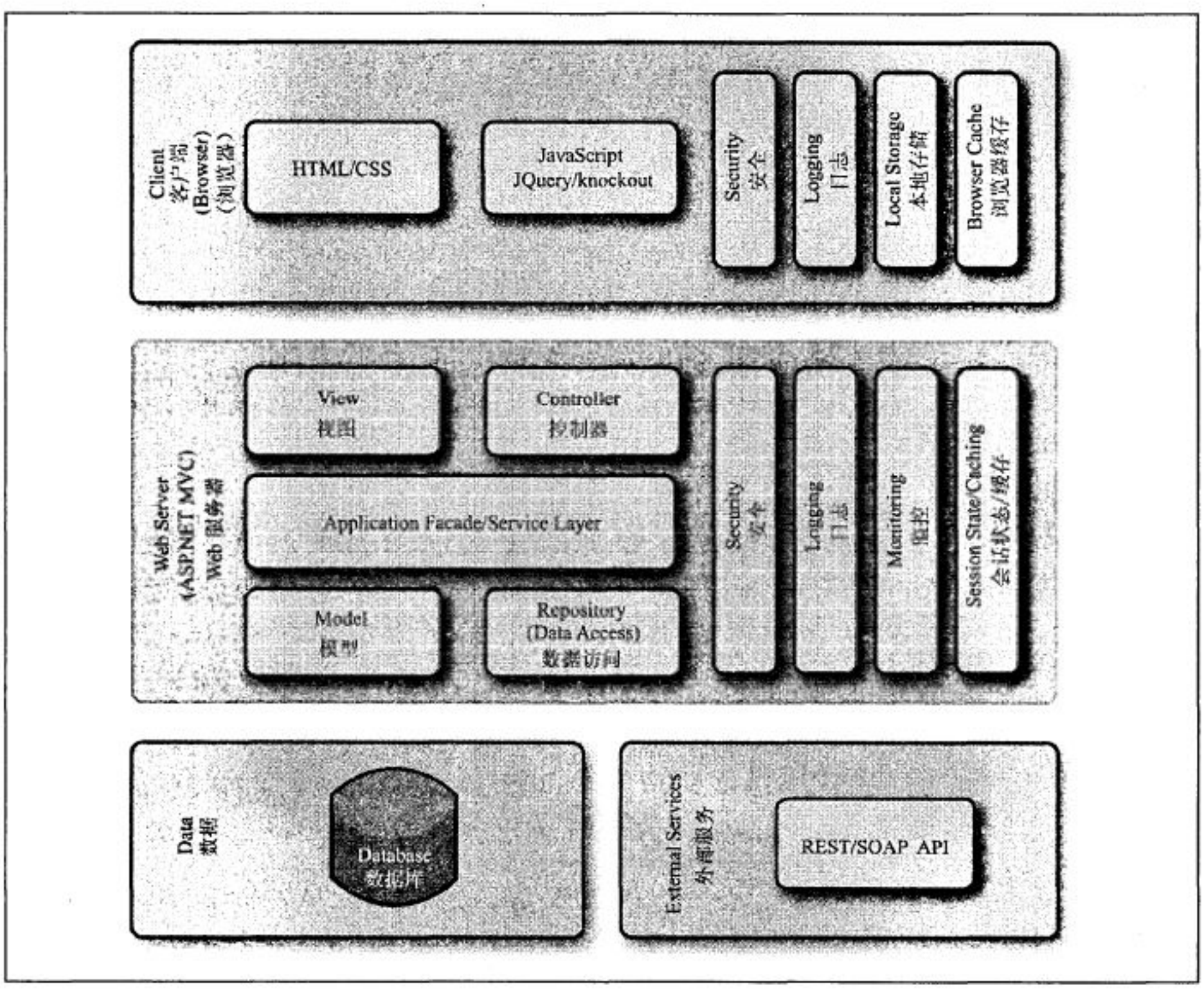


图5-3 Web应用程序逻辑架构

表5-1 组件介绍

名称	Layer（层）	描述
HTML/CSS	Client（客户端）	用来描述应用程序布局和样式的UI元素
JavaScript	Client（客户端）	用来验证和处理客户端的业务逻辑代码
Security（安全）	Client（客户端）	安全令牌（cookie）
Logging（日志）	Client（客户端）	用来日志和监控本地的服务
Local Storage（本地存储）	Client（客户端）	HTML 5本地存储（用来缓存/离线存储）
Browser Cache（浏览器缓存）	Client（客户端）	浏览器用来存储HTML、CSS、图片的缓存
View（视图）	Web Server	用来渲染HTML的服务端视图
Controller（控制器）	Web Server	应用程序控制器，处理转发用户请求逻辑

092

续表

名称	Layer（层）	描述
Model（模型）	Web Server	表示应用程序业务领域模型的类
Service Layer（服务层）	Web Server	用来封装复杂业务处理和持久化逻辑的服务层
Repository（存储库）	Web Server	数据访问组件（对象关系映射框架）
Security（安全）	Web Server	用来验证和授权用户的安全服务
Logging（日志）	Web Server	用来记录日志的应用程序服务
Monitoring（监控）	Web Server	用来监视健康的应用程序服务
Session/Caching(会话和缓存)	Data	用来转换状态的应用程序服务
External Service（外部服务）	Data	应用程序需要使用的外部系统

逻辑设计的最佳实践

图5-3所示的分层应用程序设计提供了最灵活的应用程序架构。每层拥有自己特定的职责，每层只是依赖堆栈上更低级别的层。例如，数据访问存储库与模型位于同一个层次，所以完全可以接受它只有一个依赖项。该模型与基础数据存储库是隔离的 ；它不在乎什么样的存储库如何处理持久化工作，也不关心是保存到本地文件还是数据库。

当设计Web应用程序时，通常出现的争论就是，在哪里强制业务和验证规则。MVC模式提倡模型负责业务逻辑。这样，虽然在分布式应用程序中，每层应该分别从某种程度上验证用户输入，但理想情况下是应该始终在发送到另一层之前检查输入数据。

每层应该承担它可以强化的验证级别责任。下游层永远不应假定调用层已验证了每个数据。在客户端，应使用JavaScript (JQuery)来验证必填字段并限制常见UI控件的输入数据（数字、日期、时间等）。应用程序业务模型应强制所有的业务和验证规则，而且数据库层应该使用强类型字段和强制约束以防止非法数据类型。

093

想要避免的就是在每个层之间复制复杂的业务逻辑代码。如果显示界面针对普通用户和管理员有其特定的功能，那么业务模型应该可以确定哪些功能是启用还是禁用，并提供一个标志让普通用户用来隐藏或禁用管理字段。

物理设计

物理架构设计的作用是为Web应用程序定义物理组件和部署模型。大多数Web应用程序都是基于N层模型的。客户端层由HTML、CSS和Web浏览器内部运行的JavaScript组成。客户端向服务器发送HTTP请求以检索HTML内容，或者执行AJAX请求以返回部分HTML页、XML或JSON数据。应用程序层包括ASP.NET MVC框架（在IIS Web服务器里运行）和任何自定义或者应用程序使用的第三方程序集。数据层可能包括一个或更多关系或NoSQL数据库，或一个或多个外部SOAP或者REST Web服务，或者第三方应用程序编程接口(API)。

项目命名空间和程序集名称

在部署ASP.NET MVC Web应用程序之前，开发人员需要决定如何将应用程序代码在物理上分隔成不同的命名空间和程序集。有很多不同的方法可用来设计ASP.NET MVC 应用程序。开发人员可以在网站里保留所有的应用程序组件，或者把组件分离到不同的程序集中。在大多数情况下，把网站的业务逻辑和数据访问层分离到不同的程序集是个不错的主意。这样做通常是为了更好地隔离UI和业务模型，而且更易于编写自动化测试，把测试代码关注在核心应用程序逻辑上。另外，使用这种方法可以在其他应用里重用业务逻辑和数据访问层代码（如控制台应用程序、网站、Web服务等）。

一个共同的根命名空间（例如，公司.{应用程序名称}）应该在所有程序集里都保持一致。每个程序集应该在此基础上创建自己的子命名空间。图5-4所示为网上购物网站EBuy引用的应用程序的项目结构。应用程序的功能已被划分为三个项目：Ebuy.WebSite包含的视图、控制器和其他与Web相关的文件；Ebuy.Core包含的业务模型的应用程序；CustomExtensions项目包含的应用程序的模型绑定、路由和控制器以及使用的其他自定义扩展。此外，该项目有两个测试项目（未显示）：UnitTests和IntegrationTests。

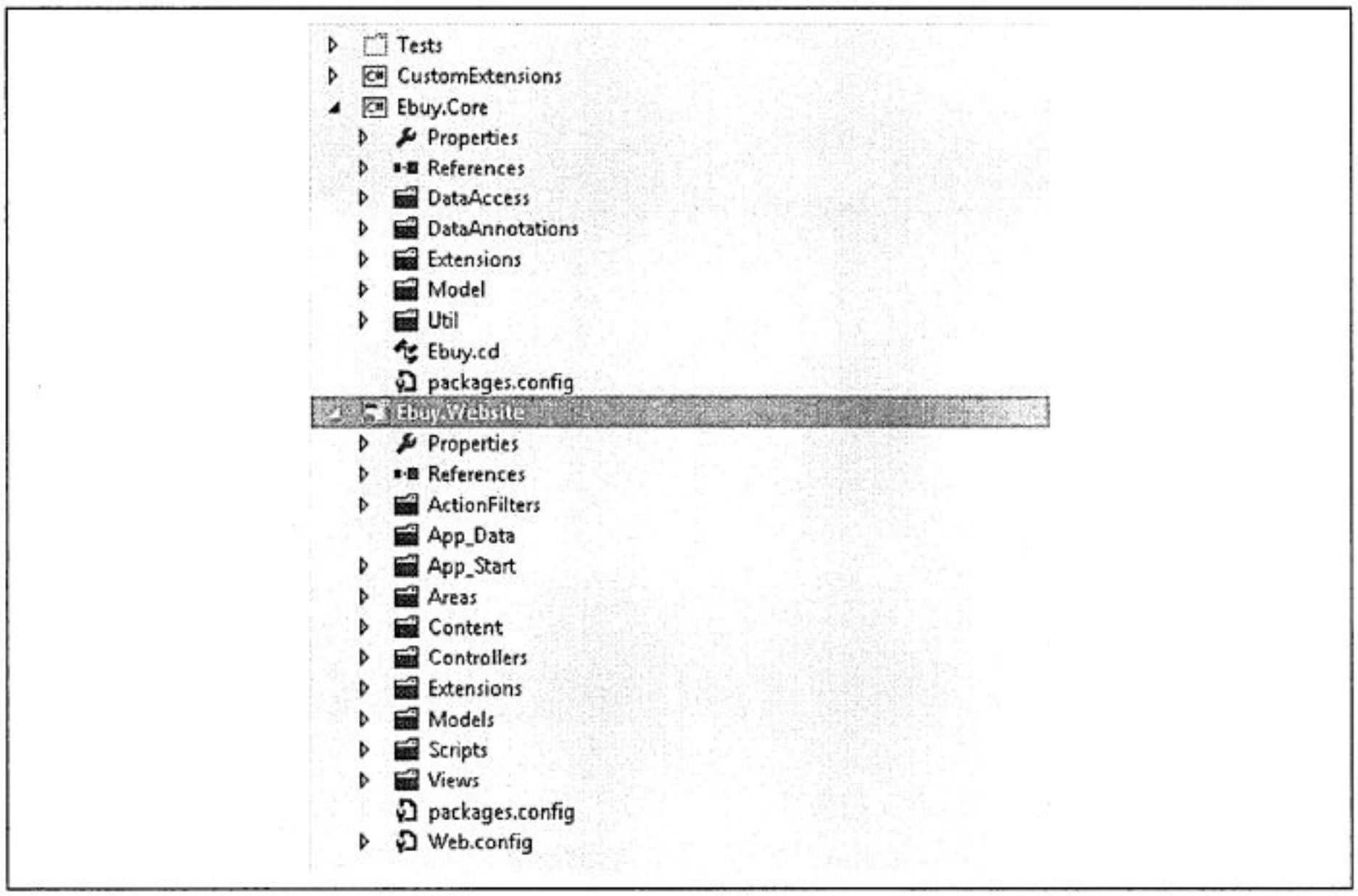


图5-4 Visual Studio项目结构

部署选项

一旦定义了Visual Studio解决方案和项目结构，开发人员便可以使用自动或手动方式部署编译过的应用程序到Web服务器。请参阅第19章有关如何部署ASP.NET MVC Web应用程序的详细内容。

图5-5展示了已配置为使用多服务器Web农场(Web farm)和SQL Server数据库集群的ASP.NET MVC Web应用程序。ASP.NET MVC Web应用程序支持所有种类的部署模型。应用程序可以使用一个本地的SQL Server Express数据库单独部署，也可以使用企业N-Tier模型部署。

物理设计的最佳实践

没有设计Web应用程序架构的“万能银弹”（译注3），而且每种选择都各有利弊。但重要的是应用程序的设计应该是灵活的，并且包含适当的监控，这样可以根据实时监控数据来调整应用程序。这是ASP.NET MVC框架的最大亮点之一。它在设计上就具备灵活性和可扩展性。ASP.NET MVC框架更容易使用IIS和ASP.NET中内置的服务，而且它提供了可扩展性，以及插入不同的组件和服务的能力。

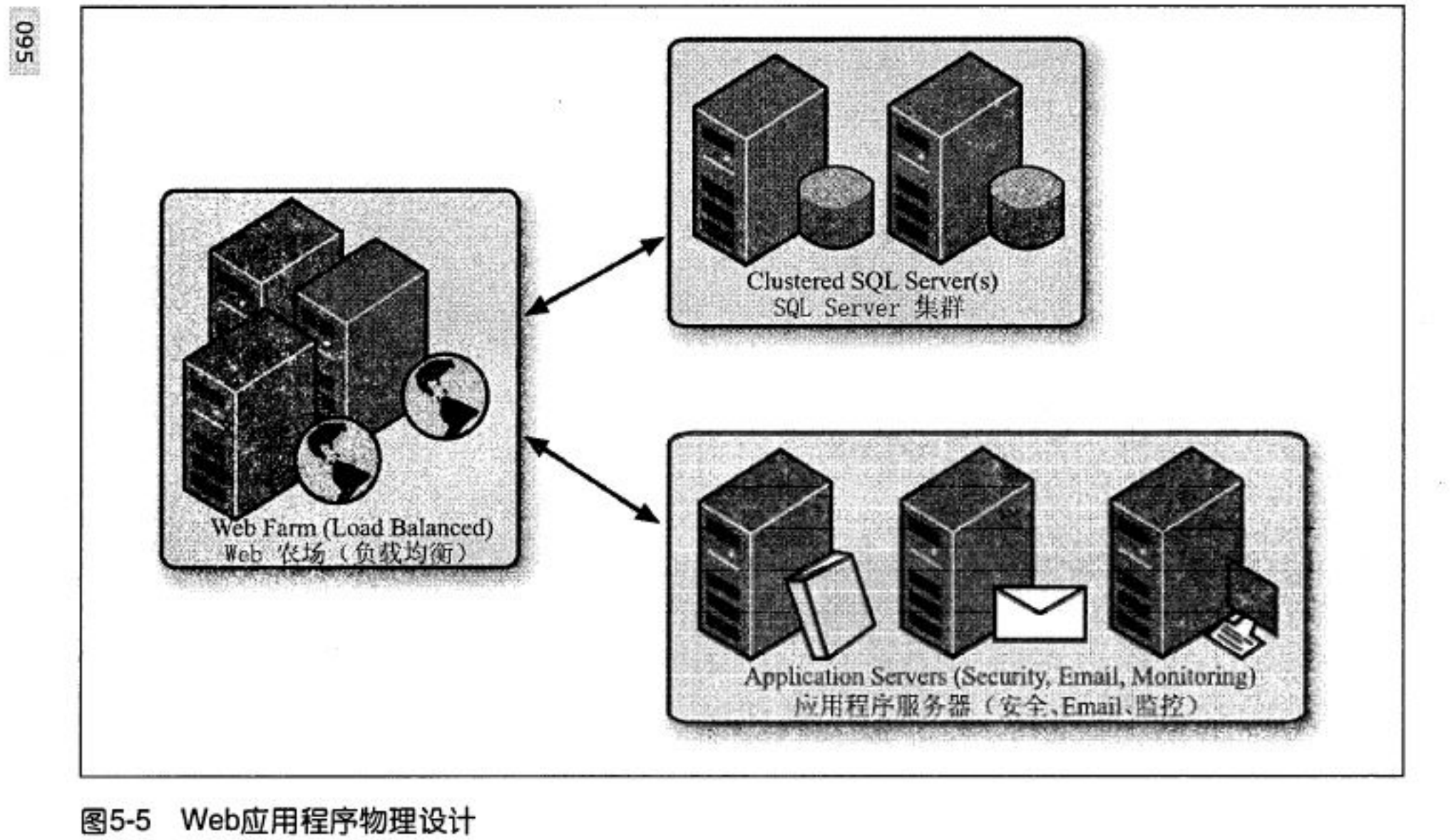


图5-5 Web应用程序物理设计

当设计一个Web应用程序时，有很多需要考虑的因素。最重要的4个因素是性能、可扩展性、带宽和延迟时间。处理一个或多个这些问题所做的选择可以对其他因素产生影响。设置监视策略可以正确评估应用程序的工作状况，尤其是底层负载，而且可以确定适当的平衡因素。

性能和可伸缩性

处理性能或可伸缩性问题都会轻易影响到其他方面。如果一个应用程序的设计需要保留大量

译注3：silver bullet 译为银弹。银色（镀银）子弹在欧洲传说中被认为是狼人和吸血鬼的克星，是杀死狼人的唯一方法（只是一种说法），专门对付妖怪，并具有驱魔的效力。现在经常被用做致命武器的代言词。本文指的是万能的架构设计方法。

的缓存数据，那么就会影响应用程序的内存使用。IIS辅助进程要考虑内存使用要求以正确配置：如果分配太多内存，则工作进程将被回收。了解.NET垃圾回收器（garbage collector, GC）如何释放资源也有非常重要的意义。不断地向会话状态里保存大型集合或数据集对象可能导致对象代龄（译注4）增加。.NET垃圾回收器标记该对象为第2代（generation 2）对象，或把对象放入大对象堆（large object heap）中。

Web农场（译注5）是一种提高Web应用程序可扩展性的好的方法。其关键理由是它是通过应用程序技术设计来实现的。使用Web农场会影响应用程序处理的瞬时状态。如果硬件和软件支持负载平衡，则可以使用保留会话的方法确保用户总是将路由定位到它最初建立会话的服务器。

另一种提高可扩展性的方法是使用会话状态服务器或持久化会话状态到数据库。通常要尽量减少使用会话状态，并确保已为缓存的数据定义超时规则。

由于会话状态和数据被缓存起来的位置是可配置的，因此要确保所有可能使用的类已正确设置为可序列化。这可以使用.NET `SerializableAttribute`标记属性，或者使用`ISerializeable`接口，或者使用WCF（译注6）数据契约来支持序列化。

带宽和延迟时间

要处理带宽和延迟时间，可能会有困难。延迟时间通常是恒定的约束，如果服务器位于纽约，而用户在日本浏览网站，则每个请求的延迟会非常严重（很可能为5秒钟）。有很多种方法可以用来解决这个问题，其中包括压缩JavaScript文件、使用图片映射以及限制请求的数目。带宽通常是变化的，但如果应用程序需要跨网络发送大量的数据，成本可能就会非常高。

处理这些问题的最佳选择就是尽量减少请求的数量，并保持较小的有效载荷。好的策略是在可能的情况下，对大型结果集启用分页、网络上只发送必要的字段，利用客户端进行验证和缓存。

设计原则
Design Principles

当设计应用程序、框架或者类时，最重要的是考虑代码的可扩展性，而不仅仅是开发出某些基本功能。这种思想也应用到ASP.NET MVC框架设计中。整个框架都使用并且提倡使用面向对象设计的最佳实践和基本原则。

译注4：对象代龄（object generation age），是.NET垃圾回收器的垃圾回收机制中采用的一种对象内存回收策略，会把内存中托管的对象分成不同的代龄。通常优先回收代龄比较低的对象，每次垃圾回收器执行回收工作都会导致未被回收的对象代龄增加。

译注5：相对于Web园（Web Garden）的单个物理服务器多个工作进程来说，Web农场指的是应用程序被多个服务器托管。

译注6：WCF是微软基于.NET平台构建的分布式开发框架，不仅支持并集成了许多微软的分布式技术，如.NET Remoting、MSMQ、WSE等，而且可以支持基于SOAP或者REST风格的Web服务。有兴趣的读者可以参阅我翻译的WCF经典书籍《WCF服务编程》（第三版）和《WCF技术内幕》。

SOLID原则

SOLID是一些特定的面向对象（object-oriented）应用程序开发原则的简称，它们用来指导面向对象的设计与开发工作。当整个应用程序采用这些原则时，这些技术将协同工作，可以创建出更易于测试和适应变化的模块化组件。SOLID包括以下这些原则。

单一职责原则

单一职责原则（SRP）是指对象应承担单一的责任，它们的行为应该关注在责任上。一个很好的例子就是不同的显示界面对应不同的控制器。`HomeController`应只包含主页有关的操作，而`ProductController`应该只处理产品页面的操作。同样，视图应该只关注渲染UI，而且要避免任何数据访问逻辑。

如下所示的`ErrorLoggerManager`类是打破了单一职责原则的一个常见类的例子。这种类有两个方法，一个将错误保存到事件日志中，一个用于将错误日志记录到文件。把这些方法组合到一起，在开始的时候并没有问题，但这个类包含了太多的职责。当添加其他日志记录方法时，这个问题会变得更加明显。通常代码要提防的是名为`xxxManager`的类。这种类可能包含了更多的职责。

```
public class ErrorLoggerManager
{
    public void LogErrorToEventLog(Exception e)
    {
        // 日志代码
    }

    public void LogErrorToFile(Exception e)
    {
        // 日志代码
    }
}
```

开放封闭原则

开放封闭原则(Open/Closed Principle，OCP)鼓励对扩展开放，对修改关闭。这一原则是对SRP原则很好的补充，也就是说，我们应该通过继承类来扩展其功能，而不是向类添加更多的行为和责任。一个很好的例子就是横切关注点服务，例如错误日志记录。它们都不是在相同的类中添加保存错误到数据库或文件的代码，而是创建一个抽象类，并由不同的子类实现自己的日志记录方法。

回顾一下`ErrorLoggerManager`类，就很容易看出这个类是如何违反SRP和OCP原则的。这个类目前有两个非常特别的实现方法：`LogErrorToEventLog`会把错误记录到事件日志，而`LogErrorToFile`会把错误记录到文件中。若需要记录其他的错误日志类型，那么这个类的可维护性就变得无法保证，变得无法控制。

```
public class ErrorLoggerManager
{
    public void LogErrorToEventLog(Exception e)
    {
        // 日志代码
    }

    public void LogErrorToFile(Exception e)
    {
        // 日志代码
    }
}
```

这里是指遵守了SRP和OCP的更新版ErrorLogger类。引入了一个名为IlogSource的接口，此外设立了两个子类继承自这个接口：EventLogSource和FileLogSource。这两个类分别负责处理特定类型的日志记录。现在有新的日志记录类型出现，而不需要修改任何现有的类了。

```
public class ErrorLogger
{
    public void Log(ILogSource source)
    {
    // 日志代码
    }
}

public interface ILogSource
{
    LogError(Exception e);
}

public class EventLogSource: ILogSource
{
    public void LogError(Exception e)
    {
    LogError(Exception e);
    }
}

public class FileLogSource: ILogSource
{
    public void LogError(Exception e)
    {
    LogError(Exception e);
    }
}
```

此时，我们可能会想，虽然这种方法看起来更清晰，但需要更多的实现代码。本书的例子将显示：遵守这些原则实现的松耦合的组件设计会带来许多好处。

里氏替换原则

里氏替换原则（Liskov Substitution Principle，LSP）的对象应易于被其子类型的实例替换，而不会影响对象的行为和规则。例如，尽管有一个共同的基类或接口是个不错的主意，但是这种做法可能会引起代码间接打破LSP。

看看ISecurityProvider和实现此接口的类。一切看起来都不错，直到UserController调用RemoveUsertheActiveDirectoryProvider类。在此示例中，只有DatabaseProvider类支持删除用户。解决这一问题的一种方法是添加特定类型的逻辑代码，这正是UserController目前在做的。但这种方法有很大的缺点：它打破了LSP。这种情况下会引发一个异常和强迫引入特定类型代码，是一个坏主意。要解决此问题的一种方法是利用SOLID的接口隔离原则，下面会进行详细讨论。

```
public interface ISecurityProvider
{
    User GetUser(string name);
    void RemoveUser(User user);
}

public class DatabaseProvider: ISecurityProvider
{

    public User GetUser(string name)
    {
        // 添加新用户
    }
    public void RemoveUser(User user)
    {
        // 保存用户
    }
}

public class ActiveDirectoryProvider: ISecurityProvider
{

    public User GetUser(string name)
    {
        // 添加新用户
    }

    public void RemoveUser(User user)
    {
        // 不允许用户删除 AD
        throw new NotImplementedException();
    }
}

public class UserController: Controller
{
    private ISecurityProvider securityProvider;

    public ActionResult RemoveUser(string name)
    {
        User user= securityProvider.GetUser(name);

        if(securityProvideris DatabaseProvider) // 打破 LSP
            securityProvider.Remove(user);
    }
}
```

接口隔离原则

接口隔离原则（Interface Segregation Principle, ISP）鼓励在整个应用程序中使用接口的同时，限制接口的大小。换句话说，应存在多个更小、更多的特定接口，而不是一个包含所有对象行为的超类接口。一个很好的例子是，.NET为序列化和销毁对象分别定义了单独的接口。一个类如果实现了ISerializable和IDisposable接口，那么就可以只关注序列化的调用者，只关心ISerializable接口的实现情况。

下面的代码是正确使用ISP的例子。创建两个单独的接口——一个（IsearchProvider）仅包含搜索方法，另一个（IRepository）用来持久化实体对象。注意：搜索控制器（SearchController）只关注搜索行为，而且仅仅引用ISearchProvider接口。这是强大的，因为可以使用这种技术来强制执行安全级别。例如，假设允许任何人可搜索一种产品，但只有管理员可以添加/删除产品，那么通过使用ISP可以确保允许匿名访问搜索控制器，且只搜索产品而并不会添加/移除它们。

```
public interface ISearchProvider
{
    IList<T>Search<T>(Criteria criteria);
}

public interface IRepository<T>
{
    T GetById(string id);
    void Delete(T);
    void Save(T)
}

public class ProductRepository< : ISearchProvider, IRepository<Product>
{
    public IList<Product>Search(Criteria criteria)
    {
            // 搜索代码
    }

    public Product GetById(string id)
    {
            // 数据访问代码
    }

    public void Delete(Product product)
    {
            // 数据访问代码
    }

    public void Save(Product product)
    {
            // 数据访问代码
    }
}

public class SearchController: Controller
{
```

101

```
        private ISearchProvider searchProvider;

        public SearchController(ISearchProvider provider)
        {
                this.searchProvider= provider;
        }

        Public ActionResult SearchForProducts(Criteria criteria)
        {
                IList<Products>products = searchProvider.Search<Product>(criteria);
                return view(products);
        }
    }
```

依赖倒置原则

依赖倒置原则（Dependency Inversion Principle，DIP）是指互相依赖的组件应该通过抽象来进行交互，而不是直接通过具体来实现。使用这个原则的一个很好的例子，就是将依赖于一个抽象类或接口的控制器与数据访问层交互，而不是直接创建特定类型的数据访问对象进行通信。

依赖倒置原则的优点是，使用抽象可允许不同的组件进行开发，彼此独立地进行更改，不仅可能引入新的抽象实现代码，而且易于测试，因为可以非常方便地模拟依赖项。

下一节将深入学习如何实现某些特定的依赖倒置原则——控制反转（Inversion of Control，IoC）（译注7）原则。这个原则可以让我们更容易通过单独组件来管理和创建这种抽象的生存期。

下面的代码是很好的依赖倒置原则的例子。`SearchController`类依赖于`ISearchProvider`接口，而不是直接创建`ProductRepository`实例，控制器使用`ISearchProvider`的实例传递到控制器中。下一节将介绍如何使用IoC容器来管理依赖。

```
public class SearchController: Controller
{
    private ISearchProvider searchProvider;

    public SearchController(ISearchProvider provider)
    {
            this.searchProvider= provider;
    }
}

public class ProductRepository: ISearchProvider
{}
```

102

译注7：IoC就是控制反转，意味着将设计好的类交给系统去控制，而不是在类内部控制。

控制反转

现在，我们已经掌握了SOLID设计原则的概念，是时候把所有这些原则放在一起使用了：控制反转是一种提倡实现松耦合层、组件和类的设计原则，它颠倒了应用程序的控制流程。

与传统的应用程序代码显示控制调用过程不同，IoC使用分离执行特定问题处理代码的概念。此方法允许独立开发应用程序的各个组件。例如，在MVC应用程序中，可以独立设计并构建模型、视图和控制器。

控制反转（IoC）设计原则的两个最流行的实现就是依赖注入（dependency injection）和服务定位（service location）。这两种方式使用相同的中心容器（central container）的概念来管理依赖项的生存期。两种实现方式的主要区别是如何访问依赖项：服务定位依赖调用者调用依赖，而依赖注入通过类的构造函数、属性或者执行方法来实现。

理解依赖

了解不同类型的依赖关系（dependency）以及如何管理依赖项之间的关系是减少应用程序复杂性的关键所在。依赖关系有多种形式：.NET程序集可以引用一个或多个其他.NET程序集，MVC控制器必须继承ASP.NET MVC基类控制器，ASP.NET应用程序需要一个IIS Web服务器托管它。

图5-6所示为控制器和存储库类（Repository）之间的关系。这种方案是控制器直接创建存储库类的一个实例。控制器与存储库类紧密耦合。对存储库中的公共接口的任何更改可能会影响到控制器类。例如，开发人员决定更改存储库类的默认构造函数，需要一个或多个参数，此更改会影响控制器类。

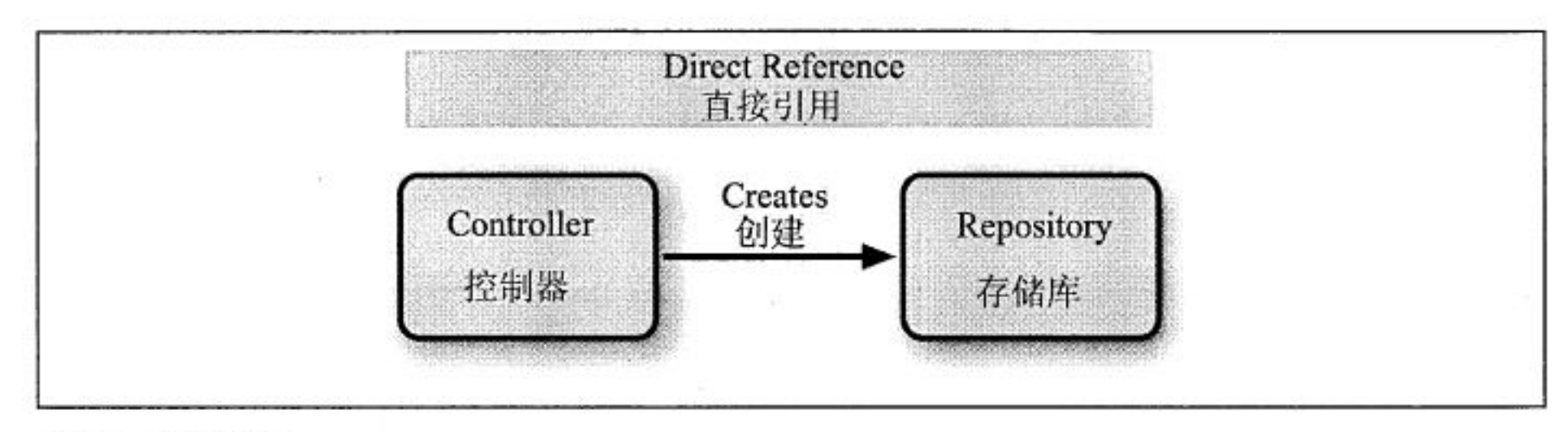


图5-6 管理依赖

为了使控制器与存储库之间的关系更加松耦合，开发人员可以使用抽象类IRepository并把创建的存储类的代码移动到工厂（模式）。图5-7说明了这种配置。控制器现在只是依赖`IRepository`接口，并对存储库类（Repository）的构造函数的任何更改都不影响控制器类。虽然这消除了控制器和存储库之间的紧耦合，但是它引入了一种控制器和工厂类之间的新关系。

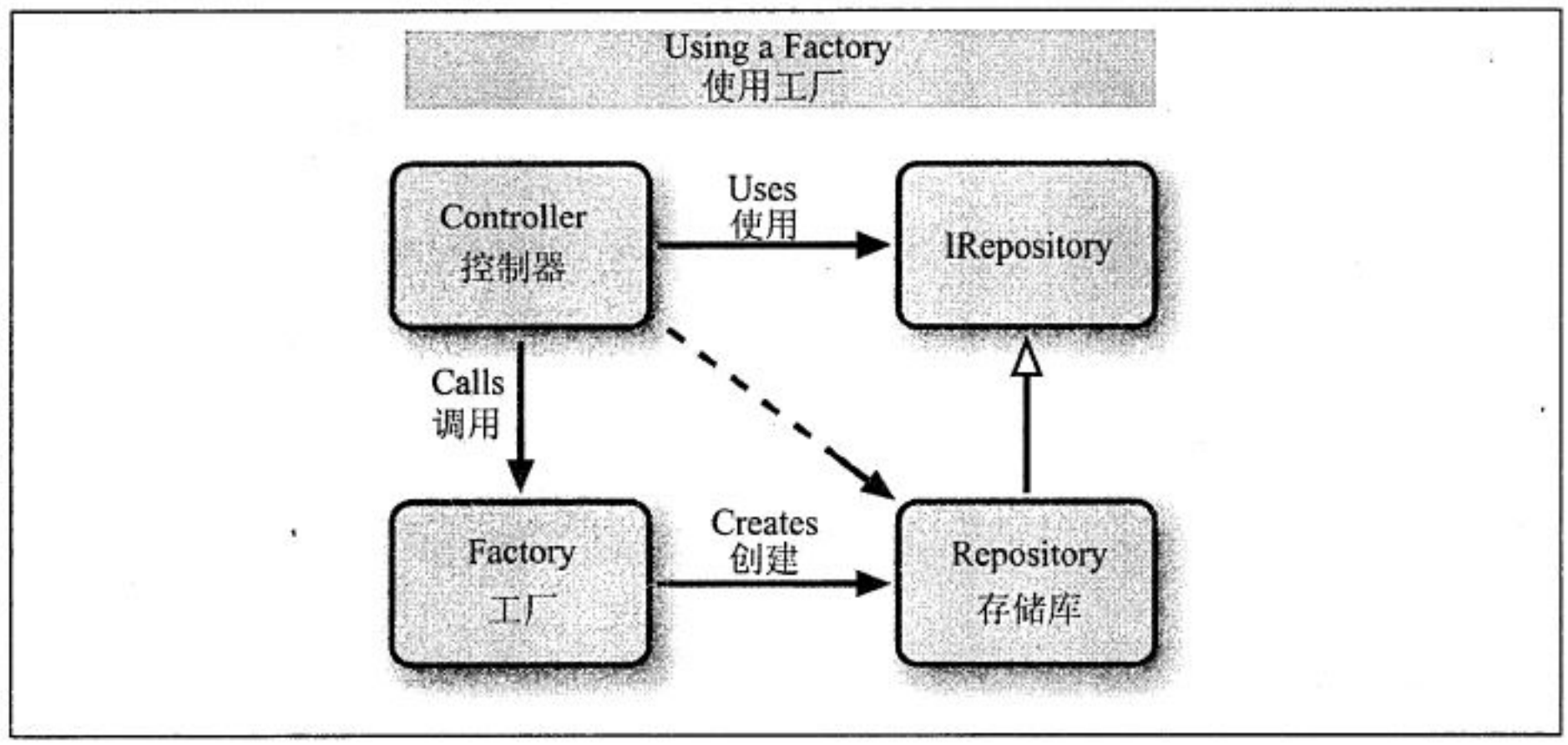


图5-7　使用抽象

图5-8显示了如何使用IoC容器替换工厂类，作为一种手段来管理控制器和存储库中类之间的依赖关系。此实现代码仍然使用`IRepository`接口作为控制器的抽象。现在只有控制器不知道存储库是怎么创建的——IoC容器负责创建并“注入”（即传递）到该控制器的存储库类实例。使用IoC容器为工厂提供其他级别的功能，可以通过配置IoC容器（IoC container）来实现类的生命周期管理。

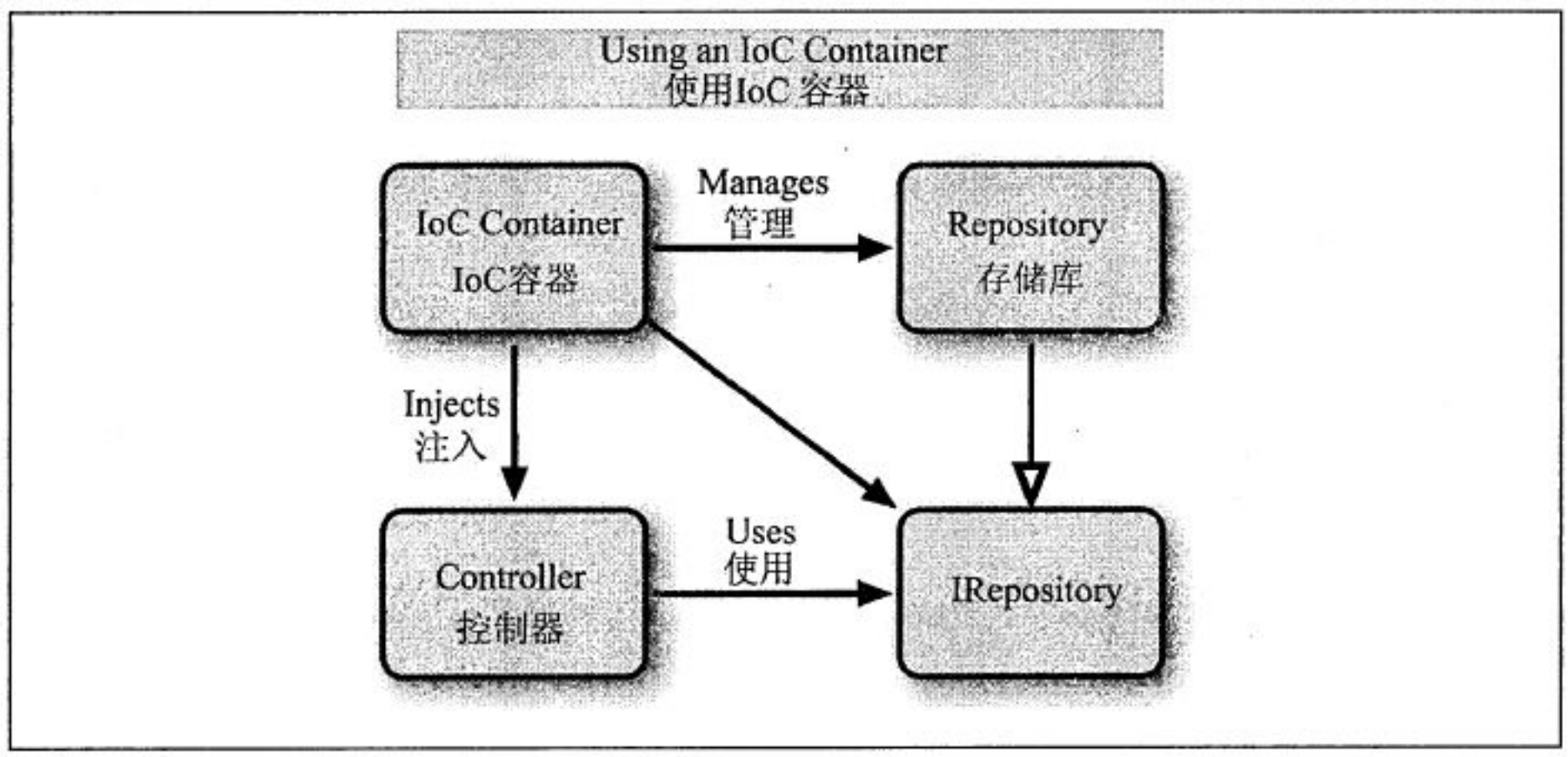


图5-8　使用IoC管理依赖

服务定位

使用服务定位器方式（service locator pattern）非常简单，开发人员只需要通过IoC 容器获取一个特定的服务类即可。用容器来检查被请求的类是否已配置，并且根据配置的生命周期管理规则创建一个新实例，或返回给请求者一个以前创建的实例。

服务定位器方式的功能十分强大，如要直接访问数据库中的某项特定服务、单个方法，仅需要请求服务的名称，并不需要接口。使用这种模式的主要缺点是代码需要直接访问IoC容器，这会导致代码和IoC容器API之间的紧密耦合。一种降低耦合的方法就是通过接口使用抽象IoC容器。

下面就是根据注册接口、调用IoC容器获取特定服务的示例代码：

```
public class AuctionsController: Controller
{
    private readonly IRepository _repository;

    public AuctionsController()
    {
        IRepository repository = Container.GetService<IRepository>();
        _repository = repository;
    }
}
```

依赖注入

与服务定位器方式相比，依赖注入方式提倡采用更松散耦合的方法。DI通过构造函数、属性或方法来传递依赖。通常情况下，大多数开发人员使用构造函数注入，因为大多数情况下需要立即使用依赖。然而，某些IoC容器允许通过延迟加载（lazy-loading）（译注8）方法注入依赖。这种情况下，调用属性之前是不会加载依赖项的。

`AuctionsController`类已经设置了构造函数注入方式。这个控制器类使用数据存储库来持久化和查询相关的交易数据。然而，该控制器并没有直接引用存储库的实现代码。相反，控制器使用的是`IRepository`接口，而且IoC容器会确定正确的`IRepository`实现类，并且在运行时“注入”它：

```
public class AuctionsController: Controller
{
    private readonly IRepository _repository;

    public AuctionsController(IRepository repository)
    {
        _repository = repository;
    }
}
```

依赖注入真正开始显示威力的时候是在涉及多个级别的依赖关系的时候，例如，自己的依赖项有其他依赖项的情况。当IoC容器注入一个依赖项时，它将检查是否已经有一个以前加载依赖项的实例。如果没有，它就会创建一个新实例，以查看它是否需要注入其他依赖项。在此方案中，当IoC容器遍历依赖树时，它就会创建必要的依赖。

下面的例子展示了依赖链的工作原理。当IoC容器创建`AuctionsController`实例时，它首先就会探测这个类是否依赖`IRepository`接口，然后会检查是否注册了`IRepository`的实例，并创建`AuctionsRepository`对象。因为这个类包含了自己的依赖，故IoC容器将创建一个

译注8：依赖注入的三种实现方法：①构造函数注入；②Set 方法注入；③接口注入。

ErrorLogger的实例并注入到AuctionsRepository中：

```
public class AuctionsController: Controller
{
    private readonly IRepository _repository;

    public AuctionsController(IRepository repository)
    {
        _repository = repository;
    }
}

public interface IRepository<T>
{
    T GetById(string id);
    void Delete(T);
    void Save(T)
}

public class AuctionsRepository: IRepository<Auction>
{
    private readonly IErrorLogger_logger;

    public AuctionsRepository(IErrorLogger logger)
    {
        _logger = logger;
    }

    public Auctions GetById(string id)
    {
        // 数据访问代码
    }

    public void Delete(Auctions auction)
    {
        // 数据访问代码
    }

    public void Save(Auctions auction)
    {
        // 数据访问代码
    }
}

public class ErrorLogger: IErrorLogger
{
    public void Log(Exception e)
    {
            // 日志代码
    }
}

public interface IErrorLogger
{
    Log(Exception e);
}
```

选择IoC容器

当使用控制反转时，开发人员需要记住两件事情：性能和错误处理。使用IoC容器来管理并注入依赖的代价可能非常大。依赖需要合适的生命周期。如果某个依赖配置为单例模式，就可能会引起跨线程访问的问题，而且需要安全管理外部资源（例如 connection strings）。不要使用IoC容器创建大的集合对象，比如使用DI来创建的包含1000个项目的集合对象，要尽量避免这种做法。错误使用或者忘记注册依赖，对调试来说是个噩梦。开发人员需要跟踪相应的依赖项，而且要确保在应用程序启动的时候正确加载所有的依赖项。

因为绝大部分IoC容器都十分相似，所以选择哪个IoC容器并不一定要经过详细的比较，很多时候都是开发人员凭自己的喜好来选择。有几个可以使用的.NET IoC容器，每个都提供了不同的注入和管理依赖的方法。最流行的IoC容器如下：

1. Ninject: *http://www.ninject.org*
2. Castle Windsor: *http://www.castleproject.org/container/index.html*
3. Autofac: *http://code.google.com/p/autofac/*
4. StructureMap: *http://structuremap.net/structuremap/index.html*
5. Unity: *http://unity.codeplex.com*
6. MEF: *http://msdn.microsoft.com/en-us/library/dd460648.aspx*

对电子交易网站EBuy来说，使用的是Ninject容器，因为这个Ninject容器在ASP.NET MVC开发社区里非常流行。它包含许多可以在ASP.NET平台上使用的自定义扩展功能，非常易于配置和注册依赖。

要初始化和使用Ninject IoC容器，就需要设置启动程序项目。启动程序项目负责管理使用Ninject注册的模块，要检查的最重要的模块就是`BindingsModule`。注意bootstrapper类的`Start()`方法和`Stop()`方法。这些方法需要在Global.asax应用程序启动的时候调用。下面展示的是如何使用Ninject bootstrapper类：

```
private static readonly Bootstrapper bootstrapper= new Bootstrapper();

/// <summary>
/// 启动应用程序
/// </summary>
public static void Start()
{
    DynamicModuleUtility.RegisterModule(typeof(OnePerRequestModule));
    DynamicModuleUtility.RegisterModule(typeof(HttpApplicationInitializationModule));
    bootstrapper.Initialize(CreateKernel);
}

/// <summary>
/// 停止应用程序
/// </summary>
public static void Stop()
{
```

```
        bootstrapper.ShutDown();
    }

    /// <summary>
    /// 创建管理应用程序的核心部分
    /// </summary>
    /// <returns>The created kernel.</returns>
    private static IKernel CreateKernel()
    {
        var kernel = new StandardKernel();
        RegisterServices(kernel);
        return kernel;
    }

    /// <summary>
    /// 加载模块或者注册服务
    /// </summary>
    /// <param name="kernel">The kernel.</param>
    private static void RegisterServices(IKernel kernel)
    {
        kernel.Load(new BindingsModule());
    }
```

BindingsModule继承自基模块类Ninject，而且包含了ASP.NET MVC框架需要的依赖注册代码。观察在单例范围内定义的控制器和路由依赖，发现两个依赖会作为单例管理，而且只会创建一个实例对象。任何自定义依赖都应该在Load()方法里注册：

```
public class BindingsModule: Ninject.Modules.NinjectModule
{
    public override void Load()
    {
        Bind<ControllerActions>()
            .ToMethod(x =>ControllerActions.DiscoverControllerActions())
            .InSingletonScope();

        Bind<IRouteGenerator>().To<RouteGenerator>().InSingletonScope();

        Bind<DataContext>().ToSelf().InRequestScope()
            .OnDeactivation(x =>x.SaveChanges());

        Bind<IRepository>().To<Repository>().InRequestScope()
            .WithConstructorArgument("isSharedContext", true);
    }
}
```

当注册自己的依赖项时，可以定义容器应如何管理它们的周期、选择哪些参数传递到构造函数，定义依赖项将被停用时要执行什么行为。

使用控制反转扩展ASP.NET MVC

ASP.NET MVC框架严重依赖控制反转原则。这一框架包含了现成的控制器工厂类，可以通过拦截输入的请求、读取MVC路由信息来创建特定的控制器，最后根据路由定义来调用控制器的方法。IoC的另一个主要应用领域是管理应用程序的视图引擎，以及控制控制器和其相应的视图之间的执行过程。

只有在我们通过使用自定义IoC容器重写默认依赖解析器（dependency resolver）扩展ASP.NET框架来获取直接控制ASP.NET MVC管理依赖和创建对象的权利时，才能真正体现IoC的强大。重写ASP.NET MVC默认的依赖解析器和实现IDependencyResolver接口一样简单，而且可以使用ASP.NET MVC框架注册自定义依赖解析器。

若要深入了解这个问题，就可以看一下如何使用Ninject IoC来创建一个自定义依赖解析器。首先，实现IDependencyResolver接口，然后通过调用构造函数来传递IKernel（Ninject的IoC容器类）的子类实例。代码如下：

```
public class CustomDependencyResolver: IDependencyResolver
{
    private readonly Ninject.IKernel _kernel;

    public CustomDependencyResolver(Ninject.IKernel kernel)
    {
        _kernel = kernel;
    }

    public object GetService(Type serviceType)
    {
        return_kernel.TryGet(serviceType);
    }

    public IEnumerable<object>GetServices(Type serviceType)
    {
        return_kernel.GetAll(serviceType);
    }
}
```

其次，通过调用System.Web.Mvc.DependencyResolver类上的静态方法SetResolver()来实现注册。代码如下：

```
Ninject.IKernel kernel = new Ninject.StandardKernel();
DependencyResolver.SetResolver(new CustomDependencyResolver(kernel));
```

注意我们是如何必须开始创建一个Ninject IKernel实例，然后传递给CustomDependencyResolver的。虽然每个IoC容器的实现方式不同，但是大多数IoC框架在创建和解析依赖关系之前都要求配置容器。在这个例子中，Ninject的StandardKernel提供相同的默认配置。

上面的代码还不能正常执行——必须先告诉StandardKernel，需要它来管理类和接口。使用Ninject框架，可以使用Bind<T>()方法来实现。例如，当需要使用IErrorLogger时，就可以告诉Ninject调用具体实现类ErrorLogger：

```
kernel.Bind<IErrorLogger>().To<ErrorLogger>();
```

这里就是为ErrorLogger类添加新绑定的例子：

```
// 使用容器注册服务
kernel.Bind<IErrorLogger>().To<ErrorLogger>();
```

DRY原则
Don't Repeat Yourself

DRY（Don't Repeat Yourself，不要重复你自己）原则是一项与SOLID原则密切相关的设计原则，它鼓励开发人员避免重复相同或者非常相似的代码。

看一下SearchController控制器类。可以看到任何潜在违反DRY原则的代码吗？看起来一切非常正确，直到我们看出应用程序中大概有几十个控制器所包含的代码几乎是完全相同或者彼此类似。最初，可能认为把CheckUserRight()方法移到控制器基类里是个不错的主意。虽然这种方法可以正常工作，但ASP.NET MVC框架提供了一种更好的办法：开发人员可以创建自定义操作过滤器ActionFilter来处理这种行为。

```
public class SearchController: Controller
{
    public ActionResult Add(Product product)
    {
        if(CheckUserRights())
            // 添加产品代码

        return View();
    }

    Public ActionResult Remove(Product product)
    {
        if(CheckUserRights())
            // 删除产品代码

        return View();
    }

    private bool CheckUserRights()
    {
        // 用户执行操作权限验证代码
    }
}
```

总结
Summary

本章介绍了ASP.NET MVC框架设计过程中使用的关键设计模式和原则。开发人员可以使用这些原则（分离关注点、反转控制、SOLID）构建灵活、可维护的ASP.NET MVC Web应用程序。使用这些原则远远超出了写好代码的要求。它们是Web应用程序架构设计中必须掌握的知识技能。

第6章

使用AJAX提升网站体验

Enhancing Your Site with AJAX

过去20年，Web应用的概念发生了巨大变化。最初设计HTML的目的是暴露基于文本的内容，以及通过互联网链接到其他文本页面。一段时间后，用户和内容生产商希望网页可以展示更多的内容，所以，为了让静态内容具备可交互性，许多网站开始使用JavaScript和动态HTML技术。异步JavaScript和XML技术也就是大家熟知的AJAX（一个新的名词，为英文单词首字母的缩写）技术，表示向Web服务器发送异步请求时，只更新页面部分数据，而不需要全部刷新整个页面。

AJAX技术首先异步请求数据，然后使用返回的内容来更新部分页面，而不是整个页面。AJAX请求包含两种类型的内容：一种是服务端生成的HTML代码，可以直接嵌入页面里；另外一种就是原始的序列化数据，客户端JavaScript可以用来生成HTML代码或者更新浏览器中的页面。

本章将介绍如何利用ASP.NET MVC的强大功能来构建AJAX Web应用。

部分渲染

Partial Rendering

向服务器发送HTTP请求，并从服务器接收HTML标签是万维网（World Wide Web）的基础。因此，向服务器再次请求HTML标签来更新网页的内容也合乎情理。这种方法就叫“部分渲染”，属于AJAX的基本过程。

部分渲染技术包括发送异步请求给服务器，服务器返回包含HTML代码的数据插入到对应的页面区域。

例如，可以使用以下语句往页面ajax_content.html里插入数据：

```
<h2>This is the AJAX content!</h2>
```

以下就是要插入网页的内容：

```
<html>
<body>
<h1>Partial Rendering Demo</h1>
<div id="container" />
</body>
</html>
```

这个例子使用了`<div id="container" />`元素来标记要插入的元素。也可以使用`jQuery.load()`请求服务端内容来更新这个`<div id="container" />`元素：

```
$("#container").load('ajax_content.html')
```

.load()方法向服务器发送异步请求，然后把内容插入#container元素。调用返回之后，DOM将会动态更新元素的内容：

```
<html>
<body>
<h1>部分渲染示例</h1>
<div id="container">
  <h2>This is the AJAX content!</h2>
</div>
</body>
</html>
```

正如例子所示，部分渲染方法是一种十分简单且高效的更新网页内容的方法，在ASP.NET MVC程序里实现起来也非常简单！

渲染部分视图

绝大部分情况下，ASP.NET MVC会把部分渲染当成其他请求一样看待——请求被路由到特定的控制器，控制器执行特定的操作逻辑代码。

二者的区别在于请求结束渲染视图时：通常的操作方法是使用Controller.View()帮助方法返回ViewResult；而部分渲染需要调用Controller.Partial()帮助方法来返回PartialViewResult对象。这一点与ViewResult类似，除了PartialViewResult只渲染包含视图的内容，不会渲染外围布局。

为了说明差别，我们来比较一下二者返回的HTML标签：普通的Auction返回的视图和AJAX返回的部分视图。

渲染普通视图

113

下面是控制器操作(AuctionsController.cs)调用Controller.View()方法的代码：

```
public class AuctionsController : Controller
{
  public ActionResult Auction(long id)
  {
    var db = new DataContext();
    var auction = db.Auctions.Find(id);

    return View("Auction", auction);
  }
}
```

对应的Auction视图（Auction.cshtml）代码如下：

```
@model Auction

<div class="title">@Model.Title</div>

<div class="overview">
    <img src="@Model.ImageUrl" alt="@Model.Title" />
    <p>
```

```
            <strong>Current Price: </strong>
            <span class="current-price">@Model.CurrentPrice</span>
        </p>
    </div>

    <h3>Description</h3>
    <div class="description">
        @Model.Description
    </div>
```

最终的HTML代码如例6-1所示。

例6-1　渲染普通Auction视图

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <link href="/Content/Site.css" rel="stylesheet" type="text/css" />
    <link href="/Content/themes/base/jquery.ui.all.css" rel="stylesheet" type="text/css" />
    <script src="/Scripts/jquery-1.7.1.min.js" type="text/javascript"></script>
    <script src="/Scripts/jquery-ui-1.8.16.js" type="text/javascript"></script>
    <script src="/Scripts/modernizr-2.0.6.js" type="text/javascript"></script>
    <script src="/Scripts/AjaxLogin.js" type="text/javascript"></script>
</head>

<body>
    <header>
      <h1 class="site-title"><a href="/">EBuy: The ASP.NET MVC Demo Site</a></h1>
      <nav>
          <ul id="menu">
                  <li><a href="/categories/Electronics">Electronics</a></li>
                  <li><a href="/categories/Home_and_Outdoors">Home/Outdoors</a></li>
                  <li><a href="/categories/Collectibles">Collectibles</a></li>
          </ul>
      </nav>
    </header>

    <section id="main">

<div class="title">Xbox 360 Kinect Sensor with Game Bundle</div>

<div class="overview">
    <img src="/Content/images/products/kinect.jpg" alt="Xbox 360 Kinect Sensor with Game ↵
    Bundle" />
    <p>
        Closing in <span class="time-remaining">4 days, 19 hours</span>
    </p>
    <div>
        <a class="show-bid-history" href="/auctions/764cc5090c04/bids">Bid History</a>
    </div>
    <p>
        <strong>Current Price: </strong>
        <span class="current-price">$43.00</span>
    </p>
</div>
```

```
<h3>Description</h3>
<div class="description">
    You are the controller with Kinect for Xbox 360!
</div>

    </section>

    <footer>
      <p>&copy; 2012 - EBuy: The ASP.NET MVC Demo Site</p>
    </footer>
</body>
</html>
```

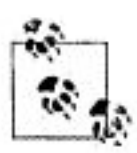
渲染部分视图

Auctions.cshtml是在网站整体布局文件中渲染的，这也是用户在第一次查看网页时需要注意的。但是，如果想重用这些布局、想让每个页面都使用统一的布局、想更新Auction交易信息而无须跳转到其他页面，该怎么办？

答案就是使用`Controller.PartialView()`方法来创建`PartialViewResult`，取代`Controller.View()`生成的`ViewResult`对象，代码如下：

```
public class AuctionsController : Controller
{
  public ActionResult Auction(long id)
  {
    var db = new DataContext();
    var auction = db.Auctions.Find(id);

    return View("Auction", auction);
  }

  public ActionResult PartialAuction(long id)
  {
    var db = new DataContext();
    var auction = db.Auctions.Find(id);

    return PartialView("Auction", auction);
  }
}
```

注意：除了将`View()`换成`PartialView()`方法，别的都一样。`PartialViewResult`甚至可以使用`ViewResult`依赖的视图。事实上，`PartialViewResult`和`ViewResult`几乎是等价的，只有一点不同，即`PartialViewResult`只渲染HTML标签内容，不包含外围布局和母版页内容。部分视图和正常的视图一样，任何语法都可以使用（比如Razor语法），而且可以完全使用ASP.NET MVC的功能，比如HTML帮助类。



因为部分视图不包含外围布局，所以也许要包含一些依赖，比如CSS或者JavaScript。要直接在部分视图中引用，而不是在布局文件中。

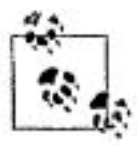
使用Auction.cshtml部分视图的渲染结果会如例6-2所示。

例6-2 渲染的auction部分视图代码

```
<div class="title">Xbox 360 Kinect Sensor with Game Bundle</div>

<div class="overview">
    <img src="/Content/images/products/kinect.jpg" alt="Xbox 360 Kinect Sensor with Game↵
    Bundle" />
    <p>
        Closing in <span class="time-remaining">4 days, 19 hours</span>
    </p>
    <div>
        <a class="show-bid-history" href="/auctions/764cc5090c04/bids">Bid History</a>
    </div>
    <p>
        <strong>Current Price: </strong>
        <span class="current-price">$43.00</span>
    </p>
</div>

<h3>Description</h3>
<div class="description">
    You are the controller with Kinect for Xbox 360!
</div>
```

修改以后，就可以使用下面的jQuery客户端代码去加载新交易的HTML代码了：

```
function showAuction(auctionId) {
  $('#main').load('/Auctions/PartialAuction/' + auctionId);
}
```



如果在Razor视图的外面编写前面的代码，则可以使用ASP.NET MVC `UrlHelper`来路由到`AuctionsController.Auction`操作。简单替换一下：

```
'/Auctions/PartialAuction/' + auctionId
```

with:

```
'@Url("PartialAuction", "Auctions")/' + auctionId
```

部分视图简化页面复杂度

前面的例子展示了如何使用部分视图显示没有布局的页面。这类部分视图通常不是全部页面。把一个完整的页面分割成不同的部分视图是非常好的简化复杂视图的有效方法。

简化页面复杂度的最好例子可能就是使用`foreach`循环，如下面代码里显示的交易信息：

```
@model IEnumerable<Auction>

<h1>Auctions</h1>
<section class="auctions">

@foreach(var auction in Model) {
  <section class="auction">
    <div class="title">@auction.Title</div>

    <div class="overview">
       <img src="@auction.ImageUrl" alt="@auction.Title" />
```

```
            <p>
                <strong>Current Price: </strong>
                <span class="current-price">@auction.CurrentPrice</span>
            </p>
        </div>

        <h3>Description</h3>
        <div class="description">
            @auction.Description
        </div>
    </section>
}

</section>
```

117

看看foreach循环里的HTML标签，是否感觉似曾相识？这就是前面我们用Auctions.cshtml部分视图的渲染结果！

这意味着可以使用Html.Partial()方法取代整段代码来渲染部分视图：

```
@model IEnumerable<Auction>

<h1>Auctions</h1>
<section class="auctions">

@foreach(var auction in Model) {
  <section class="auction">
    @Html.Partial("Auction", auction)
  </section>
}

</section>
```

注意：可以传递部分视图作为Html.Partial()方法的第二个参数来渲染网页。这样可以通过在Auction对象集合里进行迭代来渲染每个交易数据。

正如本例所示，使用部分视图不仅可以简化视图结构，避免冗余代码，而且有利于维护网站的一致性。

JavaScript渲染
JavaScript Rendering

虽然预渲染HTML的方法非常简单、高效，但是这种方式也非常浪费资源，因为有些HTML代码浏览器完全可以在客户端创建，而无需网络传输。因此，改进预渲染的方法之一就是只从服务端获取原始数据，然后在客户端动态创建HTML代码，直接操作DOM对象。

使用客户端渲染方法，必须具备两个条件：服务端可以产生序列化的数据，客户端知道如何把该数据转为标准的HTML代码。

118

渲染JSON数据

先来解决实现客户端渲染的第一个问题：序列化AJAX请求数据。在做之前，需要先确定使用什么技术。

对象标记（JSON）是互联网上一种简单、高效的数据传输格式。JSON对象使用两种数据结构：名值对集合以及有序值列表。正如名字的含义一样，JavaScript对象标记是基于JavaScript语言的子集，所以所有的新版浏览器都可以解析它。

ASP.NET MVC提供了对原生JSON的支持，使用的是`JsonResult`操作结果对象，它可以接受可序列化为JSON的模型对象。为了让控制器操作支持JSON格式的AJAX请求，可直接使用`Controller.Json()`方法来创建包含可被序列化对象的`JsonResult`。

为了在操作里使用`Json()`帮助方法和`JsonResult`，需要在`AuctionsController`里添加一个新的`JsonAuction`操作方法：

```
public ActionResult JsonAuction(long id)
{
    vardb = new DataContext();
    var auction = db.Auctions.Find(id);

    return Json(auction, JsonRequestBehavior.AllowGet);
}
```

新的控制器操作方法返回的应答消息包含序列化的JSON格式的交易数据。例如：

```
{
  "Title": "XBOX 360",
  "Description": "Brand new XBOX 360 console",
  "StartTime": "01/12/2012 12:00 AM",
  "EndTime": "01/31/2012 12:00 AM",
  "CurrentPrice": "$199.99",
  "Bids": [
    {
    "Amount" :"$200.00",
    "Timestamp": "01/12/2012 6:00 AM"
    },
    {
     "Amount" :"$205.00",
     "Timestamp": "01/14/2012 8:00 AM"
    },
    {
     "Amount" :"$210.00",
     "Timestamp": "01/15/2012 12:32 PM"
    }
  ]
}
```

使用JsonRequestBehavior防止JSON劫持

必须注意的是，`Json()`方法的第二个参数是`JsonRequestBehavior.AllowGet`，它可以通过ASP.NET MVC框架来接收`Get`方式的HTTP请求，然后返回JSON格式的数据。

这种情况下，`JsonRequestBehavior.AllowGet`的参数是必须的，因为默认情况下，ASP.NET MVC不允许对`Get`方式的HTTP请求返回JSON数据，这样就可以避免潜在的JSON劫持风险。此漏洞采用许多浏览器处理JavaScript `<script>`标记的方式，如果请求中的数据包括一个JSON数组，则可以导致公开敏感信息。

虽然有点复杂，但是必须知道避免这种漏洞的方法，即不要给不可知的HTTP GET请求返回JSON

数据。因此，在返回非敏感数据时ASP.NET MVC框架允许通过JsonRequestBehavior.AllowGet设置来允许这种不安全的方式返回JSON数据。

当需要通过JSON应答消息传输敏感信息时，可以通过在控制方法上添加HttpPostAttribute属性标记来限制只让HTTP POST方法提交请求，以避免安全漏洞。

```
[HttpPost]
public ActionResult JsonAuction(long id)
{
    var db = new DataContext();
    var auction = db.Auctions.Find(id);

    return Json(auction);
}
```

请求JSON数据

服务端配置完毕后，就可通过发送请求来查询JSON数据，在客户端构建HTML代码了。jQuery简化了这种工作。

为了向ASP.NET MVC操作请求JSON数据，可以使用$.ajax()来调用特定的URL，并处理返回数据。success函数的第一个参数(下面例子中名为“result”)包含返回的反序列化对象。

下面的代码展示了使用jQuery调用JsonAuction控制器操作，并使用返回的JSON数据通过.val()和.html()来操作DOM对象。

```
function updateAuctionInfo(auctionId) {
  $.ajax({
    url: "/Auctions/JsonAuction/" + auctionId,
    success: function (result) {
      $('#Title').val(result.Title);
      $('#Description').val(result.Description);
      $('#CurrentPrice').html(result.CurrentPrice);
    }
  });
}
```

虽然这种客户端渲染方法需要更多的代码，但是它能大大简化网络传输的数据大小，还可以大大提高传输速度和显示AJAX的效率。

客户端模板

虽然用这种字符串拼接方法生成客户端代码是非常高效的，但是它只能生成少量的HTML标签代码。随着数据的增加，这会增加拼接HTML代码的复杂性，导致难以维护。

客户端模板是非常高效的方法，可以让我们按可维护的方式快速、高效地把JSON数据转换为HTML代码。客户端模板可以使用表达式定义可重用的节点，使用数据填充控件元素。如果结合JavaScript逻辑代码，则可以在模板中执行更加强大的数据处理逻辑。

注意：客户端模板的概念并不是任何官方文档的正式定义，所以，为了使用这个方法，需要依赖JavaScript库。虽然每个库的语法千差万别，但是基本的概念仍然是相同的：JavaScript库会使用客户端模板标签，并在函数里解析JSON数据生成HTML标签。

下面的例子使用了Mustache模板语法来定义客户端HTML标签，并且使用了mustache.Js JavaScript库在浏览器里解析和执行客户端模板。其实，有很多模板可以供我们选择使用，但是需要仔细研究，对比它们的不同之处，选择最合适的一个来满足我们的需求。

首先使用客户端模板语法来重写Auction视图的代码：

```
<div class="title">{{Title}}</div>

<div class="overview">
  <img src="{{ImageUrl}}" alt="{{Title}}" />
  <p>
    <strong>Current Price: </strong>
    <span class="current-bid">{{CurrentPrice}}</span>
  </p>
</div>

<h3>Description</h3>
<div class="description">
  {{Description}}
</div>
```

注意，客户端模板的HTML标签代码几乎和最终的输出结果一样。事实上，唯一的区别就是客户端模板使用真实的数据来替代表达式。这个简单的例子也向我们展示了客户端模板机制——大部分客户端模板提供了比简单HTML容器控件更强大的功能。

其次就是编译客户端模板，或者把客户端模板HTML转换成可执行的JavaScript函数。

因为编译模板的代价最大，所以理想的办法是在保存文件以后立即编译。这样就可以立即使用编译后的模板，而不需要等待编译过程，大大提升了性能。

最后，传递需要转换的数据类型给编译后的目标，以返回格式化的HTML数据供以后直接在DOM里使用。

看看例6-3这个例子是如何使用客户端模板的。

例6-3　完整的客户端模板

```
@model IEnumerable<Auction>

<h2>Auctions</h2>

<ul class="auctions">
    @foreach(var auction in Model) {
        <li class="auction" data-key="@auction.Key">
            <a href="#">
            <img src="@auction.ThumbnailUrl" />
            <span>@auction.Title</span>
            </a>
        </li>
    }
</ul>
```

```
<section id="auction-details">
    @Html.Partial("Auction", Model.First())
</section>

<script id="auction-template" type="text/x-template">
    <div class="title">{{Title}}</div>

    <div class="overview">
        <img src="{{ImageUrl}}" alt="{{Title}}" />
        <p>
            <strong>Current Price: </strong>
            <span class="current-bid">{{CurrentPrice}}</span>
        </p>
    </div>

    <h3>Description</h3>
    <div class="description">
        {{Description}}
    </div>
</script>

<script type="text/javascript" src="~/scripts/mustache.js"></script>

<script type="text/javascript">
  $(function() {
    var templateSource = $('#auction-template').html();
    var template = Mustache.compile(templateSource);

    $('.auction').click(function() {
      var auctionId = $(this).data("key");
      $.ajax({
        url: '@Url.Action("JsonAuction", "Auctions")/' + auctionId,
        success: function(auction) {
          varhtml = template(auction);
          $('#auction-details').html(html);
        }
      });
    });
  });
</script>
```

此例中的代码虽然看起来非常多，但这个例子实际上还是很简单的：

1. 当加载页面时，脚本代码会从auction模板元素的内部查找客户端模板的HTML标签代码。
2. JavaScript脚本会把客户端HTML代码传递给`Mustache.compile()`方法，以编译到JavaScript函数里，替代之前使用的变量。
3. 脚本代码随后会侦听交易列表中每个`Auction`元素的点击事件，会触发AJAX请求，向服务端请求JSON序列化数据。
   - 如果成功返回`Auction`交易数据，则成功处理函数就会执行之前编译的客户端模板（存储在客户端模板的变量里），使用JSON数据来生成HTML代码。
   - 成功结果的处理函数通过调用`.html()`方法、使用模板函数生成的HTML代码来替

换auction-details元素的内容。

"text/x-template" MIME类型不限制返回数据格式，即可以使用任何无效的多媒体MIME类型。

浏览器会忽略自己无法识别的标签，所以，先把模板标签包装在script标签里，然后把MIME类型修改为“invalid”，比如"text/x-template"，最后会组织浏览器把数据渲染成HTML页面。

客户端模板方法看起来也许有点烦琐，但是，大部分情况下，易于维护和低带宽消耗两个优点足以抵消这些缺点。当程序使用AJAX大量交互复杂的客户端HTML代码时，客户端模板通常是一种绝佳的解决办法。

重用跨AJAX和非AJAX请求逻辑代码
Reusing Logic Across AJAX and Non-AJAX Requests

MVC模式驱动ASP.NET框架使用强大的“分离关注点”原则来确保每个组件之间的隔离性。虽然PartialAuction和JsonAuction控制器操作就是我们想要的，但如果仔细想想就会发现我们已经破坏了MVC模式的几条原则。

正常情况下，MVC应用逻辑不应该绑定到视图中。为什么还有三个控制器操作（见例6-4）执行相同的逻辑呢？而且唯一的区别只是返回给浏览器内容的方式不同。

例6-4　包含三种查询Auction数据的AuctionsController.cs

```
public class AuctionsController : Controller
{
  public ActionResult Auction(long id)
  {
    var db = new DataContext();
    var auction = db.Auctions.Find(id);
    return View("Auction", auction);
  }

   [HttpPost]
  public ActionResult JsonAuction(long id)
  {
      var db = new DataContext();
      var auction = db.Auctions.Find(id);

      return Json(auction);
  }

  public ActionResult PartialAuction(long id)
  {
    var db = new DataContext();
    var auction = db.Auctions.Find(id);

    return PartialView("Auction", auction);
  }
}
```

响应AJAX请求

为了减少重复的逻辑代码，ASP.NET MVC提供了Request.IsAjaxRequest()扩展方法，它可以帮助我们确定当前的请求是否是AJAX请求。我们可以使用这个方法来动态生成返回的数据格式。

Request.IsAjaxRequest()方法相当简单，它只是通过检查请求消息的头部XRequested-With来确定是否是XMLHttpRequest，这是大部分浏览器都会自动给AJAX请求添加的字段。

如果希望ASP.NET MVC发送AJAX请求，那么只需要在XMLHttpRequest的HTTP消息头里添加X-Requested-With的值即可。

为了演示Request.IsAjaxRequest()的使用方法，先来合并Auction和PartialAuction控制器。合并后的操作应该从数据上下文中查找auction实例对象，然后选择显示的方式。如果是AJAX请求，则使用PartialView()方法返回PartialViewResult，否则使用View()返回ViewResult对象：

```
public ActionResult Auction(long id)
{
  var db = new DataContext();
  var auction = db.Auctions.Find(id);
  if (Request.IsAjaxRequest())
    return PartialView("Auction", auction);

  return View("Auction", auction);
}
```


修改完毕后，Auction控制器操作就可以同时响应两种请求：HTTP Get和AJAX，代码逻辑保持不变。

处理JSON请求

不幸的是，ASP.NET代码没有提供类似Request.IsAjaxRequest()的方法来判断请求是否是JSON数据。但是，对它稍作修改就可以轻易实现自定义逻辑。

我们使用最简单的解决办法：在控制器操作方法里添加一个参数来指定是否返回JSON数据。例如，可以查找一个名为“format”的请求参数，当值为“JSON”时返回“JsonResult”。

```
public ActionResult Auction(long id)
{
  var db = new DataContext();
  var auction = db.Auctions.Find(id);

  if(string.Equals(request["format"], "json"))
    return Json(auction);

  return View("Auction", auction);
}
```

客户端可以通过在请求地址后面添加“?format=json”来请求JSON格式的交易数据，例如，/Auctions/Auction/1234?format=json。

可以把这些代码移植到单独的扩展方法里，以便复用这些代码，就像Request.IsAjaxRequest()一样：

```
using System;
using System.Web;

public static class JsonRequestExtensions
{
  public static bool IsJsonRequest(this HttpRequestBase request)
  {
    returnstring.Equals(request["format"], "json");
  }
}
```

使用IsJsonRequest()扩展方法后，之前的代码可以简化为：

```
public ActionResult Auction(long id)
{
  var db = new DataContext();
  var auction = db.Auctions.Find(id);

  if(Request.IsJsonRequest())
    return Json(auction);

  return View("Auction", auction);
}
```

跨控制器操作指定统一逻辑

如果把部分视图渲染和前面介绍的判断JSON请求返回数据的逻辑代码放在同一个控制器操作里，就可以在同一个应用里实现返回不同结果的灵活方法。下面是优化后的AuctionsController.cs：

```
public class AuctionsController : Controller
{
  public ActionResult Auction(long id)
  {
    var db = new DataContext();
    var auction = db.Auctions.Find(id);

    // 相应 AJAX 请求
    if(Request.IsAjaxRequest())
      return PartialView("Auction", auction);

    // 相应 JSON 请求
    if(Request.IsJsonRequest())
      return Json(auction);

    // 默认使用布局的正常视图
    return View("Auction", auction);
  }
}
```

这些代码比较灵活，但事实上没有其他的操作可以使用Auction控制器里定义的代码。所幸ASP.NET MVC提供了完美的机制在多个控制器操作上重用统一逻辑规则：操作过滤器。

把这些代码移植到操作过滤器中，就可以在其他控制器操作上重用这个过滤器。可以通过创建一个类继承System.Web.Mvc.ActionFilterAttribute类型，再重写OnActionExecuted()方法来实现。这样就可以在操作执行完毕之后来修改结果，但是需在操作结果修改之前：


```
public class MultipleResponseFormatsAttribute : ActionFilterAttribute
{
    public override void OnActionExecuted(ActionExecutedContext filterContext)
    {
      // 会在这里添加逻辑代码
    }
}
```

把Auction控制器操作里的代码移植到新的类里后，在AJAX或者JSON请求进来时修改操作结果：

```
using System;
using System.Web.Mvc;

public class MultipleResponseFormatsAttribute : ActionFilterAttribute
{
  public override void OnActionExecuted(ActionExecutedContext filterContext)
  {
    var request = filterContext.HttpContext.Request;
    var viewResult = filterContext.Result as ViewResult;

    if(viewResult == null)
       return;


    if(request.IsAjaxRequest())
    {
      // 使用 PartialViewResult 替代 result
      filterContext.Result = new PartialViewResult
          {
              TempData = viewResult.TempData,
              ViewData = viewResult.ViewData,
              ViewName = viewResult.ViewName,
          };
    }

    else if (Request.IsJsonRequest())
    {
      // 使用 JsonResult 替代 result
      filterContext.Result = new JsonResult
          {
              Data = viewResult.Model
          };
    }
  }
}
```

现在可以使用MultipleResponseFormatsAttribute操作过滤器属性标记任意操作方法，可以动态确定返回的结果类型是视图、部分视图或者JSON。可以根据请求消息来确定。

发送数据到服务器

Sending Data to the Server

本章的前半部分关注请求内容和服务器返回的AJAX数据结果。现在就要看另一半内容了：通过AJAX向服务器发送请求。

向Web服务器发送请求的两种方式就是通过URL查询参数和通过表单提交数据（HTTP Get和Post方法）。本书第1章介绍了ASP.NET MVC的模型绑定，以及ASP.NET MVC自动从请求消息里获取控制器参数。

除了从第1章知道查询字符串映射和普通HTTP表单Post数据之外，通过ASP.NET MVC模型绑定框架还知道如何映射JSON格式的数据到操作参数。这意味着操作控制器可以接受JSON格式的数据，不需要自己处理数据，因为ASP.NET MVC框架自动支持了。

我们先看一下客户端，jQuery提供的$.post()方法非常简单，可以向操作控制器发送JSON格式数据的请求消息。只要提供URL以及要发送的数据，jQuery就会把对象序列化成JSON格式的数据，作为表单的请求数据。

下面例子action里的代码实例化了一个新的Auction对象，并传给Create操作方法：

```
var auction = {
  "Title": "New Auction",
  "Description": "This is an awesome item!",
  "StartPrice": "$5.00",
  "StartTime": "01/12/2012 6:00 AM",
  "EndTime": "01/19/2012 6:00 AM"
};

$.post('@Url.Action("Create", "Auctions")', auction);
```

记住，控制器操作并不需要做什么特别处理，JSON格式的数据会自动映射到操作方法的参数上。整个控制器操作方法要做的就是执行逻辑代码（比如添加新数据到数据库）以及返回结果。Create操作方法代码如下：

```
[HttpPost]
public ActionResult Create(Auction auction)
{
  if(ModelState.IsValid)
  {
    var db = new DataContext();
    db.Auctions.Add(auction);
    return View("Auction", auction);
  }

  return View("Create", auction);
}
```


提交复杂的JSON对象

默认的JSON模型绑定逻辑包含一个重要的限制：0-1方法。这就是说，工厂期望整个应答消息只包含JSON格式的数据，不允许部分字段是JSON格式的数据，部分字段是其他格式的数据。

现在来看一下默认方法带来的问题。假设要传递一个账单Bid对象的集合，每个对象都包含两个属性，Amount和Timestamp：

```
Bid[0].Timestamp="01/12/2012 6:00 AM" &
Bid[0].Amount=100.00 &

Bid[1].Timestamp="01/12/2012 6:42 AM" &
Bid[1].Amount=73.64
```

这里是请求JSON格式的数据：

```
[
  { "Timestamp":"01/12/2012 6:00 AM", "Amount":100.00 },
  { "Timestamp":"01/12/2012 6:42 AM", "Amount":73.64 }
]
```

JSON数组不仅更干净、更简单、更小，而且在浏览器里更容易使用JavaScript构建和操作。这种简单性在动态构建表单的时候非常有用，例如，允许用户一次参与多种拍卖活动。

但是，为了使用默认的DefaultModelBinder在JSON数据里表示账单Bids的信息，还需要发送整个JSON对象。例如：

```
{
  Bids:
   [
    { "Timestamp":"01/12/2012 6:00 AM", "Amount":100.00 },
    { "Timestamp":"01/12/2012 6:42 AM", "Amount":73.64 }
   ],
}
```

表面上，给模型绑定提交JSON对象看起来不错，其实这个方法有很多弊端。

首先，客户端必须动态构建整个消息，而且必须知道对象的每个字段——HTML表单不再是个窗体，而是JavaScript逻辑代码收集数据的方式。

其次，服务端只会再以Content Type（内容类型）为"application/json"的JSON请求消息，所以这种方法对标准的HTTP GET请求没用，只对包含正确头部消息类型的AJAX请求有效。最后，如果只有一个字段有效，在默认的绑定逻辑情况下，模型绑定器会认为整个对象无效！

为了尽量避免这些问题，可以通过自定义模型绑定器来替换内置的JSON绑定模型逻辑，如例6-5所示。JsonModelBinder与JSON值提供者工厂（JSON value provider factory）不同，它允许在JSON数据里包含别的字段，而不强制整个请求消息为JSON格式。因为模型绑定器分别处理每个属性，所以可以混用不同的字段，部分字段支持JSON数据，部分字段可以不支持JSON数据。对大部分模型自定义绑定器来说，JSON模型绑定器继承自DefaultModelBinder，所以可以在不包含JSON数据时回退到模型的绑定逻辑。

例6-5　JsonModelBinder JSON模型绑定器

```
public class JsonModelBinder : DefaultModelBinder
{
  public override object BindModel
    (
      ControllerContext controllerContext,
      ModelBindingContext bindingContext
    )
```

```
{
  string json = string.Empty;

  var provider = bindingContext.ValueProvider;
  var providerValue = provider.GetValue(bindingContext.ModelName);

  if(providerValue != null)
    json= providerValue.AttemptedValue;

  // 基本表达式确保字符串以JSON对象({})或数组字符([])表示
  if(Regex.IsMatch(json, @"^(\[.*\]|{.*})$"))
  {
    return new JavaScriptSerializer()
          .Deserialize(json, bindingContext.ModelType);
  }

  return base.BindModel(controllerContext, bindingContext);
}
}
```

选择模型绑定器

假设ASP.NET MVC关注扩展性，那么它提供几种模型绑定器也不意外。事实上，ModelBinderDictionary.GetBinder()方法已经泄漏了MVC框架查找模型绑定器的秘密了：

```
private IModelBinder GetBinder(Type modelType, IModelBinder fallbackBinder)
{

      // 为次类型查询绑定器，我们使用此绑定器的优先顺序：
      // 1.提供者返回的绑定器
      // 2.全局表里注册的绑定器
      // 3.类型上定义的绑定器
      // 4.提供的回退绑定器
```

让我们从后面开始逐步解决问题。

替换默认（回退）绑定器

如果没有其他的配置，ASP.NET MVC会为所有的模型使用DefaultModelBinder。可以通过设置ModelBinders.Binders.DefaultBinder属性来将新绑定器替换为默认模型绑定器。例如：

```
protected void Application_Start()
{
ModelBinders.Binders.DefaultBinder = new JsonModelBinder();
// ...
}
```

代码设置完毕后，JsonModelBinder就会成为新的默认绑定器，并处理没有指定绑定器的所有模型。

为模型使用自定义标记属性

或许指定模型绑定器的最优雅方式就是使用抽象ystem.Web.Mvc.CustomModelBinderAttribute

声明方式来标记类和属性。虽然可以将这种方式用于任意想要的模型，但是最好还是和请求模型绑定，因为模型绑定本身就是为请求模型而生的。

为了使用CustomModelBinderAttribute方法，需要先创建子类来继承它。下面的代码展示了如何使用CustomModelBinderAttribute，以及如何应用到CreateProductRequest模型上：

```
[AttributeUsage(AttributeTargets.Class | AttributeTargets.Enum |
                AttributeTargets.Interface | AttributeTargets.Parameter |
                AttributeTargets.Struct | AttributeTargets.Property,
                AllowMultiple = false, Inherited = false)]
public class JsonModelBinderAttribute : CustomModelBinderAttribute
{
    public override IModelBinder GetBinder()
    {
        return new JsonModelBinder();
    }
}

public class CreateProductRequest
   {
    // ...

    [Required]
    [JsonModelBinder]
    public IEnumerable<CurrencyRequest>UnitPrice { get; set; }
}
```

使用JsonModelBinderAttribute标记CreateProductRequest.UnitPrice，也就是说，应该使用JsonModelBinder(JsonModelBinderAttribute.GetBinder()创建)来绑定CreateProductRequest.UnitPrice属性。

这还要在没有全局处理器或者模型绑定器提供者时才使用，注意前文提到的优先级。

注册全局绑定器

设置注册绑定器跟设置默认（回退）绑定器差不多，我们也可以为单独的类型注册模型绑定器。其语法与设置默认绑定器一样，非常简单。

这个例子表明MVC框架为每个Currency模型使用JsonModelBinder：

```
ModelBinders.Binders.Add(typeof(Currency), new JsonModelBinder());
```

这个方法可以让我们在整个程序范围内把绑定指定到某个特定类型上。这也是模型绑定器属性的一种等价代码实现方式。

高效地收发JSON数据

JSON是构建基于AJAX的RIA（译注1）Web应用的基础模块，所以了解如何正确使用JSON非常重要。正如下面的代码所示，jQuery让这一切变得非常简单，我们可以很方便地处理JSON和HTML元素。

译注1：富互联网应用（Rich Internet Applications，RIA）为具有高度互动性、丰富用户体验以及功能强大的客户端。值得一提的是RIA让普通Web应用具备传统桌面应用的体验。

最具挑战性的问题恐怕就是JSON数据的序列化。复杂对象通常包含很多关系，或者依赖数据访问技术，比如Entity Framework。当使用JSON时，可能出现无法序列化对象的情况，这时就会返回500状态码，表示服务器内部错误。

另外的缺点就是对复杂对象，JavaScript处理起来可能也有问题。解决这个问题的一个好经验，就是定义轻量级的实体类型，叫“数据转换对象（DTO）”，可方便进行类型转换。DTO可以使用简单的数据结构，并且可避免复杂的层级关系。

此外，DTO类应该只包含应用或者请求需要的字段。如果有多个DTO类也是可以的，甚至对同一个实体，每个不同的请求返回不同的结果。

下面是个简单的DTO类例子。简单的数据结构方便JavaScript进行操作。而且，由于DTO的体积更小，所以它比Auction模型更适合作为AJAX应答消息，这也是一种很好的优化措施。

```
public class AuctionDto
{
    public string Title { get; set; }
    public string Description { get; set; }
}
```

跨域AJAX请求
Cross-Domain AJAX

默认情况下，浏览器只允许来自本站的请求。这种限制可以避免很多安全问题，比如跨站脚本攻击（XSS）。有时候，应用确实需要与外部托管的REST API交互，比如Twitter或者Google。

这种情况下，外部的Web应用必须支持JSONP请求或者跨站资源共享（Cross-Origin Resource Sharing, CORS)。ASP.NET MVC不直接提供支持，要实现这种功能需一些代码和配置工作。

JSONP

JSONP（表示“JSON with Padding”）巧妙地利用了跨站请求伪造（Cross-Site Request Forgery）技术，允许我们实现AJAX跨域调用，不过浏览器会“非常辛苦”。

宏观上看，JSONP交互包含以下几步。

1. 客户端创建接收JSONP应答消息的JavaScript函数，比如`updateAuction`。
2. 客户端动态为DOM添加`<script>`标签，欺骗浏览器误以为它正在包含一个真正的脚本，然后利用浏览器允许`<script>`引用外站资源的“后门”。
3. `<script>`指定外部的JSONP服务器地址，然后指定第1步里回调的函数名称，例如：`<script href="http://other.com/auctions/1234?callback=updateAuction"/>`。
4. 服务器像处理别的JSON请求一样处理请求。一个重要的区别就是：它不是直接在应答消息里返回JSON对象，而是在客户端回调函数名里包装对象（如下面例子所示）。

注意，服务器既不知道也不关心回调函数是什么，只知道唯一的职责就是调用函数，并假定客户端一定存在这个函数。

```
updateAuction({
  "Title": "XBOX 360",
  "Description": "Brand new XBOX 360 console",
  "StartTime": "01/12/2012 12:00 AM",
  "EndTime": "01/31/2012 12:00 AM",
  "CurrentPrice": "$199.99",
  "Bids": [
    {
     "Amount" :"$200.00",
     "Timestamp": "01/12/2012 6:00 AM"
    },
    {
     "Amount" :"$205.00",
     "Timestamp": "01/14/2012 8:00 AM"
    },
    {
     "Amount" :"$210.00",
     "Timestamp": "01/15/2012 12:32 PM"
    }
  ]
});
```

值得注意的是，JSONP方法是一种完全不同的C/S（客户端/服务器）数据交换方法。它在回调的函数参数里包含原生的JSON数据，而不是直接返回JSON数据（正常的AJAX请求）。因此在客户端访问返回数据的方法只有在JSONP回调函数里实现。

上面的例子中，JSONP回调函数的参数里就包含了序列化的JSON数据。JSONP应答消息可能在执行回调之前执行其他逻辑代码，例如，在显示时间之前把格式转换为用户的当地时间：


```
var data = {
  "Title": "XBOX 360",
  "Description": "Brand new XBOX 360 console",
  "StartTime": "01/12/2012 12:00 AM",
  "EndTime": "01/31/2012 12:00 AM",
  "CurrentPrice": "$199.99",
  "Bids": [
    {
     "Amount" :"$200.00",
     "Timestamp": "01/12/2012 6:00 AM"
    },
    {
     "Amount" :"$205.00",
     "Timestamp": "01/14/2012 8:00 AM"
    },
    {
     "Amount" :"$210.00",
     "Timestamp": "01/15/2012 12:32 PM"
    }
  ]
```

```
};

/* 转换为本地时间 */

function toLocalTime(src) {
  return new Date(src+" UTC").toString();
}

bid.StartTime = toLocalTime(bid.StartTime);
bid.EndTime = toLocalTime(bid.EndTime);

for(var i = 0; i <data.Bids.length; i++) {
  var bid = data.Bids[i];
  bid.Timestamp = toLocalTime(bid.Timestamp);
}


/* 执行回调 */
updateAuction(data);
```

发送JSONP请求

jQuery `$.ajax`方法为JSONP请求提供了完美的支持。我们要做的就是在`dataType`和`jsonpCallback`里指定`jsonp`的数据类型以及指定客户端回调函数的名字。

下面的例子展示了jQuery `$.ajax` JSONP请求的代码：

```
function updateAuction(result) {
  var message = result.Title + ": $" + result.CurrentPrice;
  $('#Result').html(message);
}

$.ajax({
  type: "GET",
  url: "http://localhost:11279/Auctions/Auction/1234",
  dataType: "jsonp",
  jsonpCallback: "updateAuction"
});
```

注意，它是查询字符串参数，而不是使用`.success()`和`.error()`注册事件的JavaScript函数，回调函数必须是全局的、唯一的函数。否则JSONP脚本无法执行这个回调函数。

为ASP.NET MVC控制器操作添加JSONP支持

对JSONP，ASP.NET MVC并没有提供内置支持，所以要想在操作方法里使用JSONP，就要自己实现代码。幸运的是，JSONP返回的结果数据比ASP.NET MVC框架的`JsonResult`操作结果更新。

支持JSONP的最好方法也许就是创建自定义`ActionResult`。例6-6所示为自定义实现代码。

例6-6.JsonpResult：自定义JSONP操作结果

```
using System.Web.Mvc;

public class JsonpResult : JsonResult
```

```
{
  public string Callback { get; set; }

  public JsonpResult()
  {
    JsonRequestBehavior = JsonRequestBehavior.AllowGet;
  }

  public override void ExecuteResult(ControllerContext context)
  {
    var httpContext = context.HttpContext;
    var callback = Callback;

    if(string.IsNullOrWhiteSpace(callback))
      callback= httpContext.Request["callback"];

    httpContext.Response.Write(callback + "(");
    base.ExecuteResult(context);
    httpContext.Response.Write(");");
  }
}
```


也许你已注意到，`JsonpResult`硬编码把`JsonRequestBehavior`属性设置为了`JsonRequestBehavior.AllowGet`。这是因为，根据定义，所有的JSONP请求必须是GET请求。

每个JSONP请求都会存在安全漏洞，所以，我们必须避免通过JSONP发送敏感信息。

为了应答JSONP请求，只需要返回`JsonResult`对象：

```
public ActionResult Auction(long id)
{
  var db = new DataContext();
  var auction = db.Auctions.Find(id);

  return new JsonpResult { Data = auction };
}
```

启用跨站资源共享

跨域AJAX调用的首选方法是跨站资源共享（CORS）。与JSONP不同，CORS不会利用安全漏洞，而且，它是由特殊的HTTP消息头告诉浏览器服务允许跨域AJAX调用，避免了“黑客（hacks）”，让CORS方法更加简单，因为这样就不再需要JavaScript回调函数或者自定义操作结果类。

为了启用CORS支持，只需要给每个需要CORS支持的请求消息设置`Access-Control-Allow-Origin header`值即可。可以把允许访问的域名设置成“白名单”，或者使用“*”授权访问任何域名：

```
HttpContext.Response.AppendHeader("Access-Control-Allow-Origin", "*");
```

也有别的方法，如可以给网站的全部请求消息添加HTTP消息头，只要在配置文件`system.webServer > httpProtocol > customHeaders`节点下设置以下代码：

```
<system.webServer>
  <httpProtocol>
    <customHeaders>
      <add name="Access-Control-Allow-Origin" value="*" />
    </customHeaders>
  </httpProtocol>
</system.webServer>
```

接着发送普通的jQuery `$.ajax()`请求即可：

```
$.ajax({
  type: "GET",
  url: "http://localhost:11279/Auctions/Auction/1234",
  dataType: "json",
  success: function (result) {
    var message = result.Title + ": $" + result.CurrentPrice;
    '#Result').html(message);
  },
  error: function (XMLHttpRequest, textStatus, errorThrown) {
    alert("Error: " + errorThrown);
  }
});
```

添加完CORS的支持代码，其实就是实现了一个简单、高效的AJAX调用功能，但不同之处在于这个例子可以发送JSONP调用请求消息。

> **CORS浏览器支持**
>
> 编写本书时，跨站资源共享(CORS)仍处于工作草案阶段，所以还不支持主流的浏览器。因此，在使用这个方法前，要了解自己的网站面向的浏览器是否有这项功能。

总结

Summary

对于追求完美用户体验的网站开发人员来说，知道何时以及如何使用AJAX非常重要。本章介绍了几种不同的AJAX使用方法，并深入介绍了ASP.NET MVC框架对AJAX请求的支持。同时，也介绍了jQuery提供的强大API，让我们能够轻易实现网站的AJAX交互。最后还介绍了如何实现跨域AJAX请求。

第7章 ASP.NET Web API

The ASP.NET Web API

随着Web应用客户端UI AJAX请求数量的增长， ASP.NET MVC基于JsonResult的控制器操作将无法满足高级AJAX前端的需求。如果真的出现这种情况，就应该好好寻找一种更简单、优美的方法来处理AJAX请求。现在是开始使用ASP.NET Web API。

ASP.NET Web API框架同时使用了Web标准规范，比如HTTP、JSON和XML，以及一系列构建REST数据服务的参考原则。ASP.NET Web API与ASP.NET MVC很像，都使用了一些相同的核心概念，比如路由、控制器以及控制器操作结果。但它使用这些核心概念是为了支持不同的场景：那些需要使用数据服务，而非HTML标签的场景。

本章首先会详细介绍ASP.NET Web API框架，并演示如何创建ASP.NET Web API服务，然后介绍如何使用AJAX调用服务。

构建Data Service

Building a Data Service

添加ASP.NET Web API控制器与添加普通的控制器一样。下面几节会介绍如何向EBuy网站添加Web API控制器。

在开始之前，需要一个文件夹用来存放Web API控制器的源代码文件。当然，Web API控制器代码可以放在任何文件里，可根据个人的编程习惯而定。比如，有人喜欢在网站根目录创建一个API文件夹，当然也可以把它存放在控制器文件夹里，只要没有任何命名冲突，ASP.NET都会自动区分它们。

要添加新的Web API控制器，只需要用右键点击要保存的文件夹（这里是API），然后选择“控制器”即可。弹出的对话框和添加普通的控制器对话框一样，如图7-1所示。只是在这里要选择“带有空的read/write 操作的API 控制器”模板，而不是普通的ASP.NET MVC控制器模板。开始的时候，需要输入控制器的名字，例如，AuctionsController.cs。

设置完成以后，点击“添加（Add）”按钮就可以添加Web API控制器了。例7-1所示为新的Web API控制器代码。

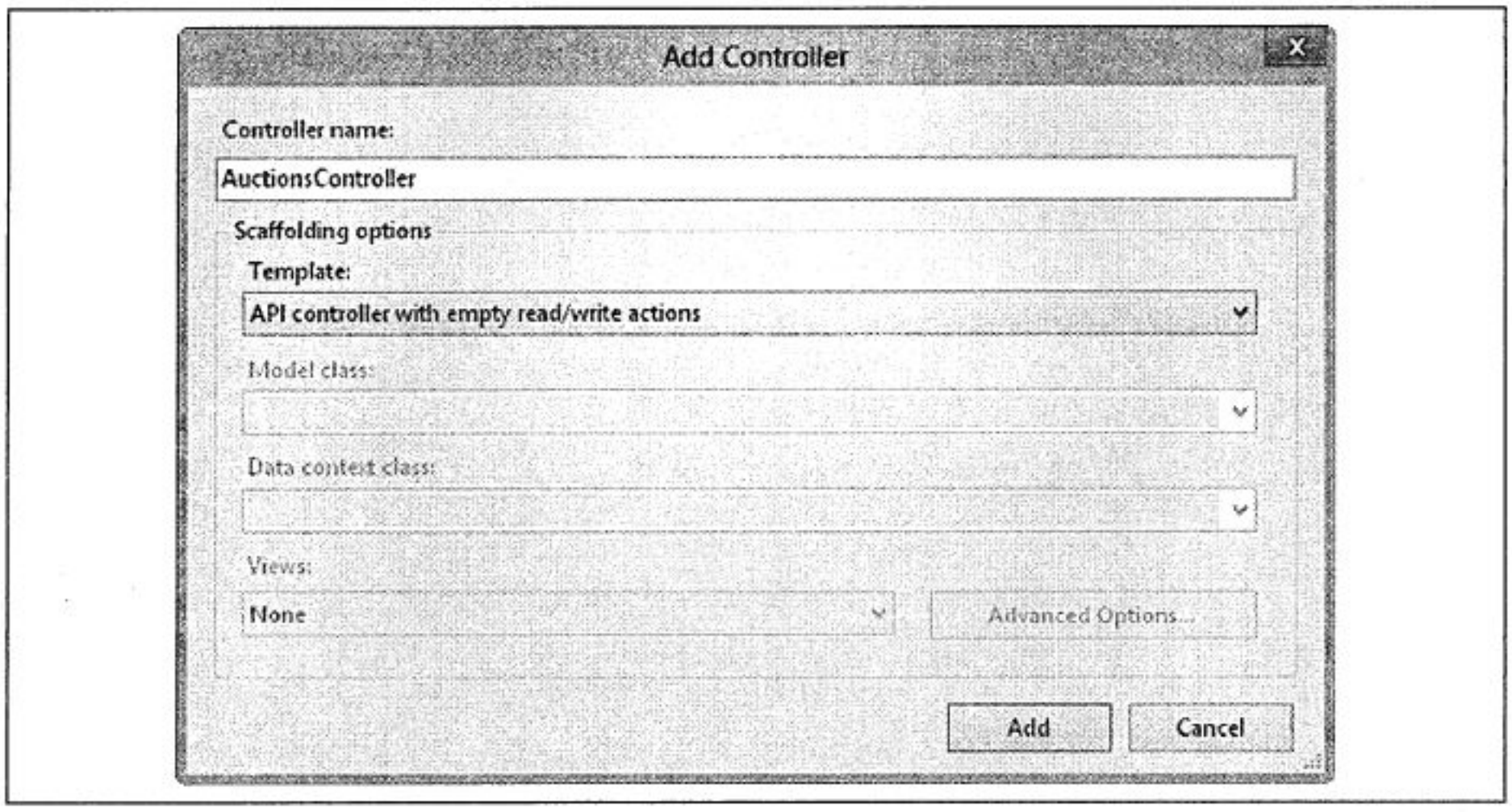


图7-1　添加Web API控制器

例7-1　Web API控制器

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Net;
using System.Net.Http;
using System.Web.Http;

namespace Ebuy.Website.Api
{
    public class AuctionsDataController: ApiController
    {

        // GET api/auctions
        public IEnumerable<string>Get()
        {
            return new string[] { "value1", "value2" };
        }

        // GET api/auctions/5
        public string Get(intid)
        {
            return"value";
        }

        // POST api/auctions
        public void Post(string value)
        {
        }

        // PUT api/auctions/5
        public void Put(intid, string value)
```

```
        {
        }

        // DELETE api/auctions/5
        public void Delete(intid)
        {
        }
    }
}
```

注册Web API路由

在使用这个控制器之前，必须在ASP.NET路由里注册它，否则无法接收请求消息。

既然是基于ASP.NET MVC框架的，那么ASP.NET Web API请求消息路由自然也是基于URL的。因此它的注册过程与普通的控制路由注册过程一样，唯一的区别就是Web API使用的是RouteTable. MapHttpRoute()扩展方法，而不是RouteTable.MapRoute()。

```
routes.MapHttpRoute(
    name: "DefaultApi",
    routeTemplate: "api/{controller}/{id}",
    defaults: new { id = RouteParameter.Optional}
);
```

这是因为Web API框架是使用“惯例优先原则”来查找正确的控制器操作方法。

我们不需要在URL路径里使用api字段，可以使用任意合法的字符串作为路由，只要不和其他的路由冲突即可。

ASP.MVC路由配置规则适用于Web API数据服务，但路由模式不要过于模糊或者过于细致。

依赖惯例优先原则

与ASP.NET MVC一样，ASP.NET Web API也大量使用了惯例优先原则以减轻繁重的开发工作。例如，ApiController依赖名称来关联不同的HTTP操作，而不是使用HttpPost Attribute来标记不同的操作方法（ASP.NET MVC操作里是通过标记属性来控制请求方式映射的）。

使用这些惯例很容易执行CRUD操作(Create、Read、Update、Delete)。标准的HTTP动词对应的CRUD操作是：

GET（读）
: 查询资源。

PUT（更新）（译注1）
: 更新现有资源（或者创建新示例。）

译注1：这里的 PUT 应该表示创建，POST 表示更新操作，我在“WCF 分布技术开发课程”11课“移动应用开发利器 REST WCF Service”介绍过，也可以参考 w3c 的官方文档：*http://www.w3.org/ Protocols/rfc2616/rfc2616-sec9.html*。

POST（创建）

创建新资源。

DELETE（删除）

删除资源。

PUT方法将会替换整个实体。为了支持部分更新，可以使用PATCH方法。

调用ASP.NET Web API数据服务非常简单。例如，下面的代码展示了如何使用jQuery的`$.getJSON()`方法来发送请求，地址是*/api/auction*。这个服务返回交易数据对象auctions的集合，格式是JSON。

```
<script type="text/javascript">

    $(function () {
        $.getJSON("api/auction/",
        function(data) {
            $.each(data, function (key, val) {
                var str= val.Description;
                $('<li/>', { html: str}).appendTo($('#auctions'));
            });
        });
    });

</script>
```

重写惯例

这里要着重指出的是，控制器命名惯例只适用于那些名字包含标准HTTP动词(Create、Read、Update、Delete)的控制器操作方法。如果想使用别的名称并且想调用Web API功能，那该怎么办呢？那就使用`AcceptVerbsAttribute`标记属性或者`HttpGetAttribute`和`HttpPostAttribute`标记属性直接标记到Web API的控制器方法上，这与标记其他的ASP.NET MVC控制器操作方法一样。下面的代码展示了这些标记属性的使用过程：

```
[HttpGet]
public Auction FindAuction(intid)
{
}
```

这个例子没有使用REST惯例Get，而是使用了自定义方法名称`FindAuction`。为了实现GET请求，我们在`FindAuction`操作上标记`HttpGetAttribute`属性。

钩住API

现在再来看一下之前创建的Web API控制器的设置，它可以执行CRUD操作。

为了访问EBuy数据库，`AuctionsDataController`构造函数需要接受数据持久化repository类的实例：

```
public class AuctionsDataController: ApiController
{
```

```
        private readonly IRepository _repository;

        public AuctionsDataController(IRepository repository)
        {
            _repository = repository;
        }
    }
```

默认情况下，Web API控制器需要默认构造参数（空参数）。因为要传递IRepository给控制器，所以启动应用程序时需要初始化自定义依赖解析器：

```
GlobalConfiguration.Configuration.DependencyResolver=
new NinjectWebApiResolver(kernel);
```

这个例子是一个使用了Ninject IoC容器的自定义依赖解析器。因为每个请求都要创建Web API控制器，所以自定义依赖解析器需要为每个请求创建一个新的依赖域（比如NinjectWebApiScope）：

```
using System.Web.Http.Dependencies;
using Ninject;

public class NinjectWebApiResolver: NinjectWebApiScope, IDependencyResolver
{
    private IKernel kernel;

    public NinjectWebApiResolver(IKernel kernel) : base(kernel)
    {
        this.kernel= kernel;
    }

    public IDependencyScope BeginScope()
    {
        return new NinjectWebApiScope(kernel.BeginBlock());
    }
}
```

以上是自定义Ninject域类的代码。当请求Web API控制器时，会调用GetService()方法；当创建控制器实例时，Resolve()会注入repository。

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web.Http.Dependencies;
using Ninject.Activation;
using Ninject.Parameters;
using Ninject.Syntax;

public class NinjectWebApiScope: IDependencyScope
{

    protected IResolutionRoot resolutionRoot;

    public NinjectWebApiScope(IResolutionRoot resolutionRoot)
    {
        this.resolutionRoot= resolutionRoot;
    }

    public object GetService(Type serviceType)
    {
```

```
        Return resolutionRoot.Resolve(this.CreateRequest(serviceType)).SingleOrDefault();
    }

    public IEnumerable<object>GetServices(Type serviceType)
    {
        return resolutionRoot.Resolve(this.CreateRequest(serviceType))
    }

    private IRequestCreateRequest(Type serviceType)
    {
        return resolutionRoot.CreateRequest(serviceType,
                                null,
                                new Parameter[0],
                                true,
                                true);
    }

    public void Dispose()
    {
        resolutionRoot= null;
    }
}
```

下面为Auctions类中执行CRUD操作的Web API控制器完整代码：

```
public class AuctionsDataController: ApiController
{
    private readonly IRepository _repository;

    public AuctionsDataController(IRepository repository)
    {
        _repository = repository;
    }

    public IEnumerable<Auction>Get()
    {
        return this._repository.All<Auction>();
    }

    public Auction Get(string id)
    {
        return_repository.Single<Auction>(id);
    }

    public void Post(Auction auction)
    {
        _repository.Add<Auction>(auction);
    }

    public void Put(string id, Auction auction)
    {
        var currentAuction= _repository.Single<Auction>(id);

        if(currentAuction!= null)
        {
            currentAuction= Mapper.DynamicMap<Auction>(auction);
        }
```

```
        }
        public void Delete(string id)
        {
            _repository.Delete<Auction>(id);
        }
    }
```


数据分页与查询
Paging and Querying Data

ASP.NET Web API框架最强大的功能之一就是可以通过开放数据协议（Open Data Protocol，OData）使用URL参数表达式来支持数据分页和过滤。例如，使用URI */api/Auction?$top=3&$orderby=CurrentBid*返回的是最上面的3个交易数据，排序的字段是`CurrentBid`属性。

表7-1所示为OData支持的一些常见查询参数。

表7-1　OData支持的查询字符串参数

查询参数	描述	例子
`$filter`	过滤符合布尔条件的值	/api/Auction?$filter=CurrentBidgt 2
`$orderby`	返回根据特定字段排序的实体集合	/api/Auction?$orderby=Description
`$skip`	指定跳过的开始n个实体	/api/Auction?$skip=2
`$top`	返回开始n个实体	/api/Auction?$top=3&

可以到OData网站学习更多关于开放数据协议(OData)的技术规范。

为了支持分页和过滤，ASP.NET Web API控制器操作必须返回`IQueryable<T>`类型的结果。当数据没存储在`IQueryable<T>`对象中时，也可以使用`AsQueryable()` LINQ扩展方法。ASP.NET Web API会处理`IQueryable<T>`结果，并且会把OData查询字符串转换为可以用来过滤`IQueryable<T>`数据的LINQ表达式。

接下来，ASP.NET Web API框架会处理LINQ表达式的查询结果，并把它们转换成JSON对象，这样数据就可以通过HTTP协议传输了。

```
public IQueryable<Auction> Get()
{
    return_repository.All<Auction>().AsQueryable();
}
```

147

异常处理
Exception Handling

开发人员构建基于AJAX的应用程序时，还需要额外关注异常处理问题。默认情况下，若服务端的AJAX请求出错，就会向客户端返回内部错误（500）。这会导致许多问题。

首先，告诉用户内部服务器错误没有意义。其次，返回错误信息对开发人员的帮助十分有限，不利于调试时跟踪问题。最后，更严重的是，如果不对错误信息进行封装过滤，就很可能带来系统安全风险：错误信息可能包含异常调用堆栈或者其他攻击者可以利用的信息！

图7-2所示是Web API控制器返回的内部服务器错误的例子，这些返回的信息只包含调用堆栈错误信息。

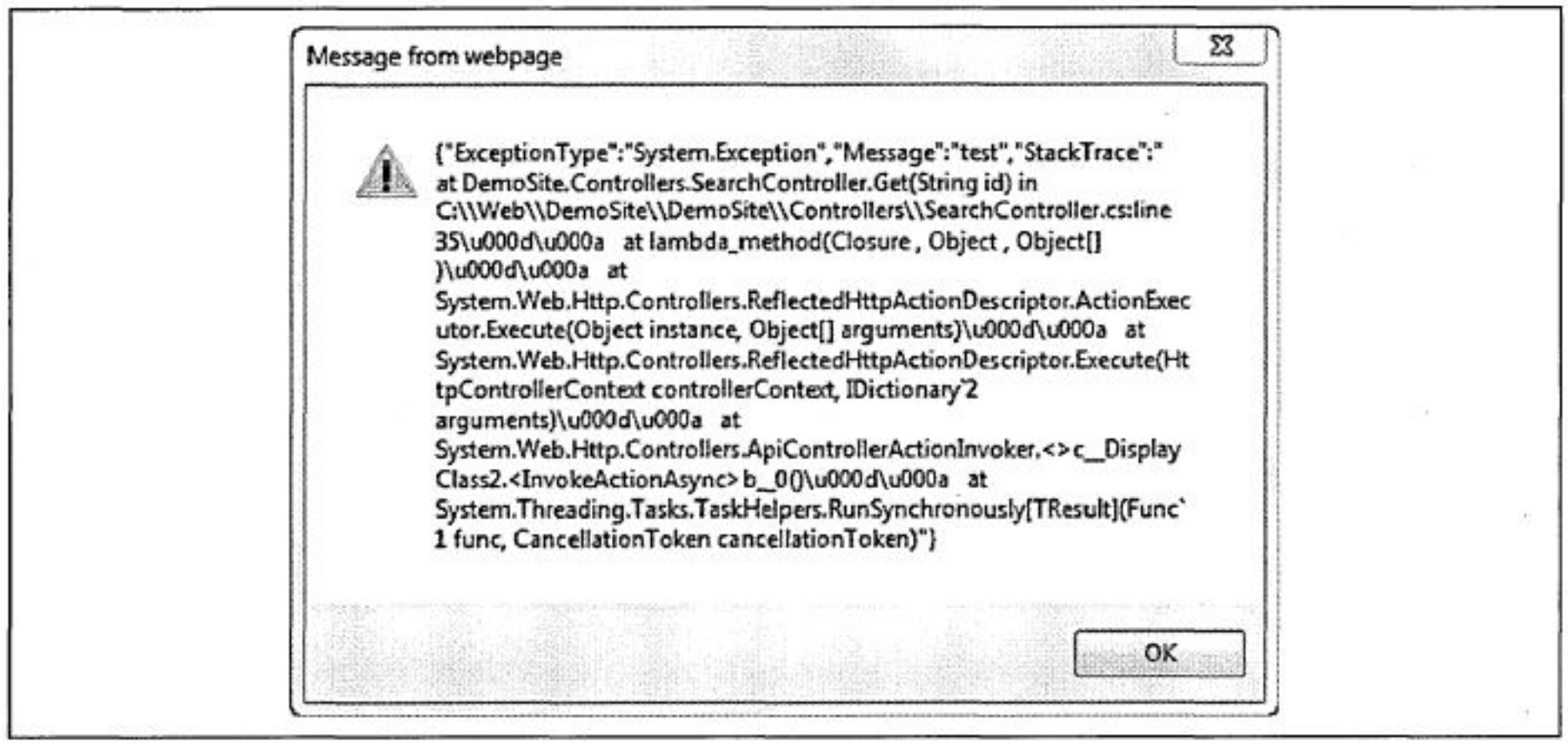


图7-2　内部服务器错误

幸运的是，ASP.NET Web API提供了异常处理机制，可以返回更加合理的信息给客户端应用。例如，`HttpResponseException`类可以更好地控制HTTP状态码和返回给客户端的错误信息。

下面的例子演示了如何使用`HttpResponseException`返回HTTP状态码（404）以及自定义错误信息：

148

```
public Auction Get(string id)
{
    var result = _repository.Single<Auction>(id);
    if(result == null)
    {
        var errorMessage= new HttpResponseMessage(HttpStatusCode.NotFound);
        errorMessage.Content= new StringContent
              (string.Format("Invalid id, no auction available for id: {0}.", id));
        errorMessage.ReasonPhrase= "Not Found";

        throw new HttpResponseException(errorMessage);
    }
```

```
        return result;
    }
```

除了使用HttpResponseException外，ASP.NET WebAPI还允许我们创建异常过滤器（exception filters）。异常过滤器可以处理控制器里非HttpResponseException类型的异常。

创建异常过滤器，可以直接继承System.Web.Http.Filters.IExceptionFilter接口或者ExceptionFilterAttribute。创建自定义标记属性也是一种创建异常过滤器的简单方法。这种方法要求重写OnException()：

```
using System.Diagnostics;
using System.Web.Http.Filters;

public class CustomExceptionFilter: ExceptionFilterAttribute
{
    public override void OnException(HttpActionExecutedContext context)
    {
        base.OnException(context);
    }
}
```

可以通过修改HttpActionExecutedContext参数来重写发送给客户端的HTTP应答消息：

```
using System.Web.Http.Filters;
using System.Net.Http;
using System.Net;

public class CustomExceptionFilter: ExceptionFilterAttribute
{
    public override void OnException(HttpActionExecutedContext context)
    {
        if(context.Response== null)
        {
            context.Response= new HttpResponseMessage();
        }
        context.Response.StatusCode= HttpStatusCode.NotImplemented;
        context.Response.Content= new StringContent("Custom Message");
        base.OnException(context);
    }
}
```

在创建完自定义异常过滤器之后，还要注册它才能使用。有两种注册方式：通过GlobalConfiguration.Configuration.Filters集合注册全局异常过滤器；在Web API控制器的操作方法上直接标记。全局异常过滤器会处理所有Web API控制器抛出的异常，除了HttpResponseException以外。

注册全局异常过滤器也非常简单，只需在程序启动阶段向GlobalConfiguration.Configuration. Filters集合里添加即可。

```
public class MvcApplication: System.Web.HttpApplication
{
    static void ConfigureApi(HttpConfigurationconfig)
    {
        config.Filters.Add(new CustomExceptionFilter());
    }
```

```
        protected void Application_Start()
        {
            ConfigureApi(GlobalConfiguration.Configuration);
        }
    }
```

当然也可以通过在Web API控制器方法上直接标记来自定义异常处理属性：

```
[CustomExceptionFilter]
public Auction Get(string id)
{
    var result = _repository.Single<Auction>(id);
    if(result == null)
    {
        throw new Exception("Item not Found!");
    }
    return result;
}
```

除了命名空间以及行为上的少许差别以外，ASP.NET Web API异常过滤器和ASP.NET MVC过滤器很像。例如，ASP.NET MVC `HandleErrorAttribute`类不能处理Web API控制器抛出的异常。

Media格式化器
Media Formatters

ASP.NET Web API强大的功能之一就是处理不同的多媒体类型(MIME)。MIME类型用来描述HTTP请求中不同的数据格式。MIME类型由两个字符串组成：类型和子类型，例如，`text.html`表示HTML格式。

客户端可以通过设置HTTP Accept消息头来告诉服务器客户端想要的MIME类型。例如，下面的Accept消息头告诉服务器客户端想要的是HTML或者XHTML：

```
Accept: text/html,application/xhtml+xml,application
```

ASP.NET Web API使用媒体类型来决定如何序列化和反序列化HTTP消息体。它支持XML、JSON以及编码的HTML表单数据。

创建自定义媒体格式化器需要继承`MediaTypeFormatter`或`BufferedMediaTypeFormatter`类。`MediaTypeFormatter`使用异步读/写方法；`BufferedMediaTypeFormatter`继承`MediaTypeFormatter`，然后包装异步读/写方法，暴露为异步操作。虽然继承`BufferedMediaTypeFormatter`很简单，但是可能引发线程阻塞问题。

下面的例子展示了如何通过创建一个自定义媒体类型来序列化`Auction`为CSV（逗号分隔值）格式。为了简单起见，自定义格式化器继承`BufferedMediaTypeFormatter`。支持的媒体类型需要在构造函数里确定。

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Net;
```

```
using System.Net.Http.Formatting;
using System.Net.Http.Headers;
using DemoSite.Models;

public class AuctionCsvFormatter: BufferedMediaTypeFormatter
{
    public AuctionCsvFormatter()
    {
        this.SupportedMediaTypes.Add(new MediaTypeHeaderValue("text/csv"));
    }
}
```

为了序列化或反序列化实体，必须重写CanWriteType()和CanReadType()方法。这些方法可以确定自定义格式化器的类型：

```
protected override boolCanWriteType(Type type)
{
    if(type == typeof(Auction))
    {
        return true;
    }
    else
    {
        Type enumerableType= typeof(IEnumerable<Auction>);
        return enumerableType.IsAssignableFrom(type);
    }
}

protected override bool CanReadType(Type type)
{
    return false;
}
```

当执行格式化器时，OnWriteToStream()方法会把类型序列化成stream流，OnReadFromStream()会从流里读取数据反序列化成特定类型对象。注意，Encode方法能做字符编码对自定义格式化器至关重要。

例7-2　序列化类型

```
protected override void OnWriteToStream(Type type,
    objectvalue, Stream stream,
    HttpContentHeaders contentHeaders,
    FormatterContext formatterContext,
    TransportContext transportContext)
{
    var source = value as IEnumerable<Auction>;
    if(source != null)
    {
        foreach(varitem in source)
        {
            WriteItem(item, stream);
        }
    }
    else
    {
        var item = value as Auction;
        if(item != null)
```

```
            {
                WriteItem(item, stream);
            }
        }
    }

    private void WriteItem(Auction item, Stream stream)
    {
        var writer = new StreamWriter(stream);
        writer.WriteLine("{0},{1},{2}",
            Encode(item.Title),
            Encode(item.Description),
            Encode(item.CurrentPrice.Value));
        writer.Flush();
    }
    static char[] _specialChars= new char[] { ',', '\n', '\r', '"' };
    private string Encode(object o)
    {
        string result = "";

        if(o != null)
        {
            string data = o.ToString();
            if(data.IndexOfAny(_specialChars) != -1)
            {
                result= String.Format("\"{0}\"", data.Replace("\"", "\"\""));
            }
        }

        return result;
    }
```

为了使用自定义媒体格式化器，还要先注册：在Global.asax.cs的Application_Start()方法里添加自定义媒体格式化器给GlobalConfiguration.Configuration.Filters集合：

```
static void ConfigureApi(HttpConfiguration config)
{
    config.Formatters.Add(new AuctionCsvFormatter());
}
```

一旦注册完毕，自定义媒体格式化器就会处理任何包含text/csv消息头的请求消息。

总结
Summary

本章介绍了微软全新的ASP.NET Web API框架，并详细介绍了如何使用这个框架来开发REST风格的Web应用数据服务接口。

第8章

高级数据

Advanced Data

目前为止，本书内容主要关注在ASP.NET MVC框架的核心组件上：模型、视图和控制器。本章稍作调整，来介绍数据访问层相关的技术，如何使用ORM（对象关系映射）数据访问模式来构建ASP.NET MVC Web应用程序。

数据访问模式

Data Access Patterns

ASP.NET MVC框架一个最大的特点就是它的扩展性。它设计的目标之一就是允许开发者灵活插入不同的组件和框架。因为ASP.NET MVC构建于.NET 4.5之上，所以任何流行的数据访问框架，包括ADO.NET、LINQ to SQL、ADO.NET EntityFramework或者Nhibernate，都可以用来构建ASP.NET MVC数据访问层。

无论选择什么数据访问框架，最重要的是要明白，这些数据层访问框架都是对ASP.NET MVC框架的完美补充。

Plain Old CLR Objects

Plain Old CLR Objects简写为POCO（朴素的旧的CLR对象），代表业务实体类（模型）的.NET类。这些类型专注于业务属性和业务行为，不需要任何数据库结构代码。

POCO类的主要目标就是让业务模型做到持久层隔离（persistence ignorance）。这种设计方法允许业务模型与数据访问层模型之间相互独立。因为业务模型不包含任何数据访问代码，所以很容易进行隔离测试，而且底层的数据存储可以很容易进行包装转换以适应变化的业务需求。

这里是一个简单的POCO类的例子，它仅仅包含属性和方法：

```
Public class product
{
    public long Id { get; set; }
    public string Name { get; set; }
    public double Price { get; set; }
    public int NumberInStock{ get; set; }
```

```
        public double CalculateValue()
        {
            return Price * NumberInStock;
        }
    }
```

注意，这个类并没有包含任何数据库访问代码。本章后面会介绍如何使用ORM框架以及repository模式（存储库模式）（译注1）来持久化POCO类。

使用该模式的最大好处就是将领域模型从客户代码和数据映射层之间解耦出来。业务模型可能包含几十个甚至上百个业务类。即使简单的模型也都要包含几个类，这对于创建包含共同属性的基类很有意义。

下面是一个基于实体类的例子。注意，类的方法为抽象方法，声明为abstract。这是抽象实体基类强调一致性的方式，让后续子类实现这些行为方法：

```
public abstract class BaseEntity
{
    public string Key { get; set; }

    public abstract void GenerateKey();
}
```

使用repository模式

repository模式是一种数据访问模式，它可以带来数据访问组件更多的松耦合。单独的一个或多个类（叫repository）会负责持久化应用业务模型，而不是直接在控制器或者业务模型类里包含数据访问逻辑。

repository模式很好地实现了MVC模式的设计原则——分离关注点。通过使用这种模式，可以把数据访问层和其余的应用代码隔离，以便利用POCO带来的好处。

有几种不同的方法来设计repository（存储库）。

每种业务模式和repository一对一

创建repository最简单的方法就是给每种业务模式类都创建一个repository。虽然这种方法很简单，但是可能带来问题，比如重复的代码和复杂性，特别是当每个repository需要与别的Repository存储库交互时。

使用聚合根

聚合根是指一个可以独立存在的类，它负责不同类之间的映射。例如，在电子商务网中，可以使用`OrderRepository`来处理订单，以及与订单相关的其他详细工作。

泛型repository一对多

与创建多个repository类的方法不同，开发人员可以使用.NET泛型机制来创建一个通用的repository，它可以在多个程序中使用。EBuy就包含了一个泛型repository的例子。

译注1：repository 模式的英文定义：A Repository mediates between the domain and data mapping layers, acting like an in-memory domain object collection。中文解释：领域模型和数据映射层之间的中介者，行为类似于内存里的域对象集合。
《企业架构模式》中 Repository 译为“资源库”。解释为：
通过用来访问领域对象的一个类似集合的接口，在领域与数据映射层之间进行协调。
《领域驱动设计：软件核心复杂性应对之道》中，Repository 译为“仓储”。解释为：一种用来封装存储、读取和查找行为的机制，它模拟了一个对象集合。

这里就是repository类的代码：

```
public class ModelRepository
{

    public ModelRepository()
    {
    }

    public void Add(Model instance)
    {
    }

    public void Update(Model instance)
    {
    }

    public void Delete(Model instance)
    {
    }

    public Model Get(string id)
    {
    }

    public ICollection<Model>GetAll()
    {
    }

}
```

除了执行CRUD操作以外，repository有时候还需要缓存实体对象。缓存处理的对象大部分都是静态对象，比如从下拉菜单里查询值。对经常更新的实体，这可能会有麻烦。

第12章将详细介绍关于缓存数据的内容。

ASP.NET MVC控制器与repository交互以加载和持久化业务模型数据。通过利用依赖注入（DI），repository可以注入到控制器的构造函数里。图8-1展示了repository和Entity Framework数据上下文之间的关系：ASP.NET MVC控制器与repository交互，而不是直接与Entity Framework交互。

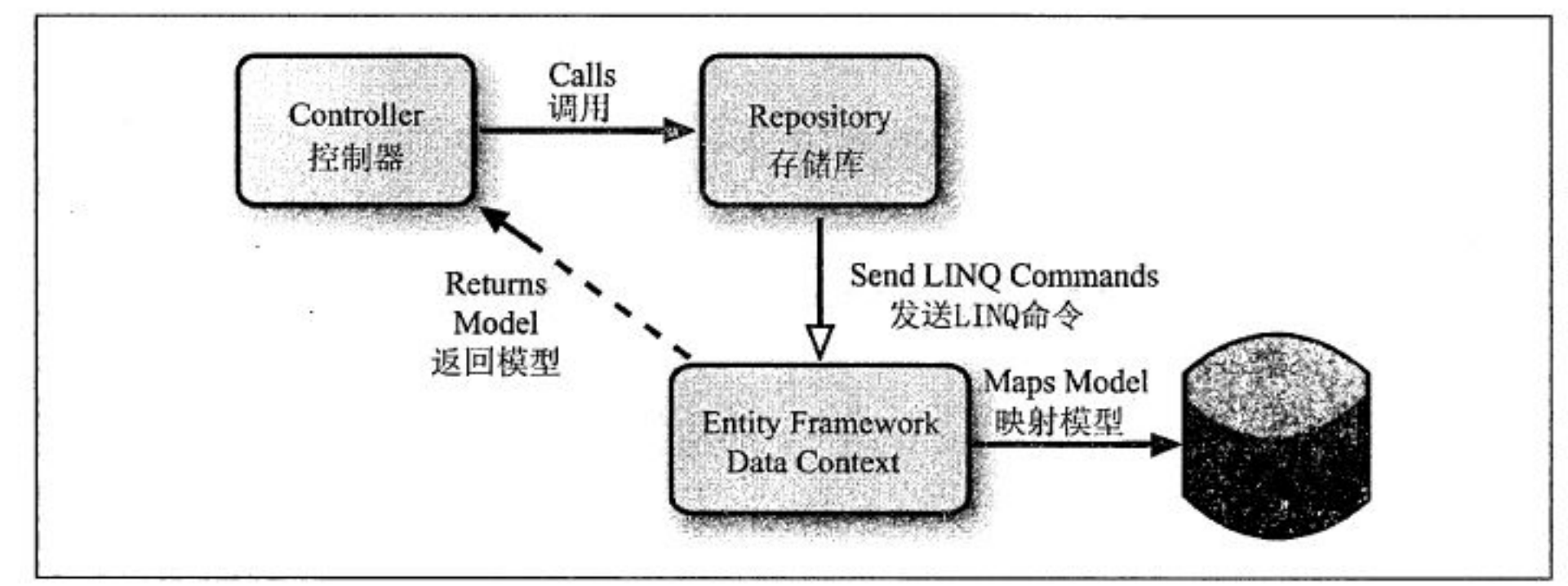


图8-1 使用repository时的交互情况

下面的例子展示了使用依赖注入方法来向控制器的构造函数注入repository对象，以及控制器如何使用repository来查询交易对象信息列表。使用依赖注入让模拟repository来测试控制器也变得非常简单：

```
public class AuctionsController: Controller
{
    private readonly IRepository _repository;

    public AuctionsController(IRepository repository)
    {
        _repository = repository;
    }

    public ActionResultIndex()
    {
        var auctions = _repository.GetAll<Auction>();
        return auctions;
    }
}
```

对象关系映射器

对象关系映射器是一种在支持类型（例如.NET类）和关系数据库模型之间进行实体映射的数据访问模式。使用这种模式的主要原因就是可以实现在业务模式和数据模型之间解耦。这种分离也称为**对象关系阻抗失配**（object relational impedance mismatch），这是对应用程序业务层和数据访问层无法兼容的一种戏称。

很容易就会落入只反映业务层中的关系数据库模型的陷阱。使用这种方法的问题是它限制了我们使用.NET平台全部功能的能力。

以下是对象关系阻抗失配的主要问题。

粒度

有时候模型类会包含比数据库的表更多的类。一个很好的例子就是Address类，这是因为现实世界中不同的地址可能关联不同的行为——例如，账单地址不仅仅处理快递地址。尽管用不同的类来表示不同的地址是个不错的主意，但是它们可能包含很多相同的数据，所以，希望用单个表里存储所有的Address类型。

继承

继承的概念，或者为了共享相同的逻辑类继承自别的类——这是OO编程的最重要方面，代码重用。不论OO中继承的概念多么重要，通常关系型数据库是无法理解这种继承的概念的。例如，可能有个数据库的Customers表存储客户数据，有个特定的字段会用于标示客户端是国内客户还是国外客户，业务模型类可能通过基类`Customer`以及几个子类来表示这种关系，比如`DomesticCustomer`和`InternationalCustomer`，来表示不同的客户。

标识

关系型数据库依赖单个的列来作为每条记录的唯一标识（主键）。这经常与.NET框架的做法冲突。对象相等通常通过对象标识相等（`a=b`）以及对象相等(`a.Equals(b)`)来判断，而这两种方法都没有单独的属性或者域来作为唯一的标识。

关联

关系型数据库使用主键和外键来建立实体之间的关联，而.NET框架使用单向引用表示对象关联。例如，在关系型数据库中可能跨表查询数据，而.NET中的关联使用者是一个类。所以，如果要支持双向关联，就需要拷贝这种关系。此外，也不可能知道一个类中复杂的多重关系。“一对多”和“多对多”的关系也是无法区分的。

数据导航

.NET框架访问数据的方式与数据库的完全不同。在.NET域模型中，需要通过不同的模型对象之间的关联来查询数据，而在数据库中只需要使用精简的SQL语句组合JOIN或者SELECT语句即可查询出不同的实体。

虽然开发人员可以通过使用模型来执行数据访问操作，比如加载数据和保存数据，但是数据库的重要性和职责仍然是最重要的，仍然应该遵守传统的数据访问层设计原则。每个表应该有单个的主键，一对多关系时应该使用外键等，例如，表示教师和学生之间的关系；而多对多关系应该使用第三个表（Class班级表），因为每个学生和教师都可能有多个班级。

实体框架概述

Entity Framework Overview

如果想自己重新开发一个全新的ORM框架，则显然不太现实，基本上也不太可能完成。幸运的是，可以直接使用很多ORM框架，而不需要自己亲自从零开始。微软也提供了两个框架：LINQ to SQL和ADO.NET Entity Framework。另外，也有很多第三方商业ORM框架或者开源免费的ORM框架，比如Nhibernate（译注2）。

ADO.NET Entity Framework（也简称为EF）是一个对象/关系映射器，现在已经包含在.NET框架里。当使用EF时，开发人员只需要与实体模型交互，而不需要直接与应用的关系型数据库模型交互。这些抽象是允许开发人员关注业务行为和实体关系，而不是如何存储实体对象到关系数据模型中。为了与实体模型交互，开发人员需要使用EF数据上下文来执行查询或持久化模型操作。当调用这些操作时，EF会生成执行这些操作的必要的SQL语句。

在传统数据访问方法向ORM方法转换的过程中，最具争议性的话题就是存储过程起什么作用。因为实体模型主要关注何时使用ORM，所以这就鼓励开发人员使用框架来处理实体映射，而不用考虑编写SQL语句。幸运的是，当我们工作的公司或者项目需要使用存储过程时，ADO.NET Entity Framework刚好也提供了对于调用存储过程的支持。详细内容可以参考*http://msdn.microsoft.com/en-us/library/bb399203(v=VS.90).aspx*。

译注2：NHibernate是一款面向.NET环境的对象/关系数据库映射工具，是一个基于.NET的针对关系型数据库的对象持久化类库。Java平台上的ORM框架Hibernate是.NET平台上的移植版本。对象/关系数据库映射技术用来把对象模型表示的对象映射到基于SQL的关系模型数据结构中去。

159

下面的例子展示了如何使用Entity Framework来添加新产品数据，调用的是SaveChanged()方法，Entity Framework会生成对应的SQL语句。

```
using(var db= new ProductModelContainer())
{
    db.Product.Add(new Product { Id = "173", Name = "XBox 360" });
    db.Product.Add(new Product { Id = "225", Name = "Halo 4" });
    db.SaveChanges()
}
```

选择数据访问方法

微软已意识到数据访问的方法一刀切是行不通的。一些开发人员喜欢以数据为中心的方法，优先设计数据库，然后生成业务模型，由数据库结构驱动模型。另外一些开发人员喜欢POCO类，把这些类与已经存在的数据库关联。

Entity Framework允许开发人员选择以下三种不同的方式。

数据库优先方式

对于更加喜欢以数据为中心的设计或者从现有数据库开始的开发人员来说，Entity Framework可以根据关系型数据库中的表来生成业务模型。Entity Framework使用特殊的配置文件（.edmx文件）来存储数据库schema、数据模型以及映射关系的信息。开发人员可以使用Visual Studio自带的Entity Framework设计器来生成、显示和编辑Entity Framework使用的概念模型。

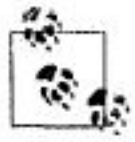
模型优先方式

对没有数据库的情况，Entity Framework提供的设计器可以创建概念数据模型。使用数据库优先模型，Entity Framework使用schema文件来存储模型到数据库schema的映射信息。在模型创建完毕之后，EF设计器可以生成创建数据库的schema。

代码优先方式

想使用持久层隔离（persistence ignorance）方法的开发人员可以直接在代码里创建业务模型。Entity Framework提供了一个特殊的映射API，以及一些惯例来支持这个方法工作。在使用代码优先方式时，Entity Framework没有使用任何外部文件来存储数据库schema，因为映射API在运行时使用这些惯例动态生成数据库schema。

160



目前，Entity Framework代码优先方式不支持存储过程映射。ExecuteSqlCommand()方法和SqlQuery()方法可以用来执行存储过程。

数据库并发

处理**并发冲突**是Web开发人员必须面对的问题。当多个用户同时修改相同的数据时，就会引发并发冲突问题。默认情况下，除非配置EF检测冲突，大多使用“最后进入”规则。例如，如果用户A和用户B同时加载相同的产品信息，那么后提交修改信息的用户将会起作用，将会覆盖先提交的修改数据。

根据应用程序的类型及其数据的变化，开发人员可以比较并发的成本是否超过了带来的好处。以下两个方法都可以用于处理并发。

保守式并发方法

保守式并发方法需要数据库组织其他的用户重写别人提交的修改数据。当查询单行数据时，会加上只读锁，直到同一个用户更新完数据，或者移除只读锁。这个方法在Web应用中会带来很多问题，因为Web网站依赖于无状态模型。其中主要的问题是何时移除只读锁，因为用户通过浏览器访问网站时，无法保证用户是否执行别的操作，以及是否触发移除数据库记录行上的只读锁。

开放式并发方法

与依赖数据库锁不同，开放式并发方法会检查最新查询的数据是否被修改。这个方法最简单的实现就是在表的末端加一个最后更新的时间戳。每次在更新之前都可以用来检查当前的数据记录是否被更新。

ADO.NET Entity Framework不直接支持开放式并发，但是它推荐使用开放式并发。Entity Framework框架提供了开放式并发的两种方法：给实体对象增加时间戳（timestamp）属性，以及处理Entity Framework数据上下文返回的`OptimisticConcurrencyException`异常。

下面是一个添加时间戳(`Timestamp`)属性给实体对象的例子。添加完毕后，任何数据的`UPDATE`和`DELETE`操作都会自动在SQL `Where`语句中添加新的时间戳。

```
[Timestamp]
public Byte[] Timestamp { get; set; }
```

为了处理`OptimisticConcurrencyException`异常，要使用.NET的`try/catch`方法来查询用户保存的实体对象的状态是否与数据库的状态一致：

```
try
{
    dbContext.Set<Product>().Add(instance);
    dbContext.SaveChanges();
}
catch(DbUpdateConcurrencyException ex)
{
    var entry = ex.Entries.Single();
    var databaseValues= (Product)entry.GetDatabaseValues().ToObject();
    var clientValues= (Product)entry.Entity;

    if(databaseValues.Name!= clientValues.Name)
       // 记录并发异常日志

}
catch(DataException ex)
{
    // 记录数据异常日志
}
catch(Exception ex)
{
    // 记录普通异常日志
}
```

构建数据访问层

Building a Data Access Layer

数据访问层的设计非常重要，因为它影响整个应用。以电子商务网站EBuy的Entity Framework代码优先方法为例，其中的EBuy业务模型设计使用了领域驱动设计（译注3）（domain-driven design，*http://www.domaindrivendesign.org*）方法，而且开发团队希望使用持久化隔离方法来确保应用程序可以很容易支持多种类型的持久化模型、关系型数据库、云存储以及NoSQL数据库。

使用Entity Framework代码优先方法

代码优先方法背后的驱动力是可以使用POCO（普通的旧的CLR对象）类。在数据库优先和模型优先方法中，EF会从`EntityObject`基类继承模型类，并提供一些基础方法来把模型类映射到数据库schema上。因为数据库优先和模型优先方法需要持久化`EntityObject`子类，所以这些子类不支持持久化隔离机制。

代码优先方法是使用惯例来映射POCO类，而不是使用实体基类：

- 表名使用多元化的窗体实体类名进行命名。
- 列名使用属性名称。
- 主键使用名为`ID`的属性或者类名`ID`。
- 默认的连接字符串匹配`DataContext`类名。

代码优先数据标记

Entity Framework包含开发人员可以用来定义实体映射的标记属性（见表8-1）。注意，ASP.NET MVC框架使用了一些相同的属性标记来做域成员级别的验证工作。

表8-1　代码优先数据标记

属性	描述
Column	数据库列的名称、序号位置和要映射到的属性的数据类型
ComplexType	在不包含键的类上使用，而且Entity Framework无法管理。通常用于管理相关实体中的标量属性
ConcurrencyCheck	用于指定属性是否应该参与开放式并发检查
DatabaseGenerated	用于标记应由数据库生成的属性

译注3：领域驱动设计（DDD）的概念由 Eric Evans 在其《领域驱动设计》一书中提出。DDD 强调以 Domain 域为关注点，而非实现技术，也就是关注 Model 模型设计。与之相关的另外两种软件设计方法为行为驱动开发(behavior-driven-development, BDD)与测试驱动开发(test-driven-development, TDD)

续表

属性	描述
ForeignKey	用于标识相关的实体，表示使用表之间的外键约束
InverseProperty	用来标识该属性所表示关系的另一端对应属性
Key	用于唯一标识实体的一个或多个属性
MaxLength	最大长度的属性（列）
MinLength	最小长度的属性（列）
NotMapped	标记将不会由实体框架映射的属性
Required	标记必须属性
StringLength	定义域成员的最小长度和最大长度
Table	定义实体使用的表名称
Timestamp	包含时间戳的属性列，用于保存之前的检查工作

重写惯例

虽然使用惯例的主要目的是提升开发效率，但是Entity Framework意识到某些情况下开发人员需要打破这些惯例，所以它提供了API允许开发人员修改这些默认设置的管理规则。

下面是一个使用Key标记属性的实体类的例子，它重写了默认的主键：

```
public class Product
{
    [Key]
    public string MasterKey{ get; set; }
    public string Name { get; set; }
}
```

除了可以使用标记来重写Entity Framework惯例外，开发人员也可以删除或者修改默认惯例。

下面的代码展示了如何删除PluralizingTableNameConvention表命名惯例，签名可以用在数据库表中：

```
public class ProductEntities: DbContext
{
    protected override void OnModelCreating(DbModelBuilder modelBuilder)
    {
        // 修改 Code First 惯例以便不使用 PluralizingTableName
        modelBuilder.Conventions.Remove<PluralizingTableNameConvention>();
    }
}
```

EBuy业务域模型

EBuy交易网站的业务模型由一些POCO类组成，使用的是DDD领域驱动设计方法。每个POCO实体类都继承自一个基实体类，这个基实体类包含所有业务模型类公共的行为和属性。

因为EBuy使用了SOLID设计原则，所以实体基类实现了两个接口：一个是自定义的IEntity接口，该接口为实体定义了以URL为键的命名原则；另外一个是.NET的IEquatable接口，这个接口定义了不同实体对象的比较丰富的基类：

```
public interface IEntity
{
    /// <summary>
    /// 实体的唯一公开标识符 (URL 唯一)
    /// </summary>
    /// <remarks>
    /// 这是通过 Web 等暴露的标识符
    /// </remarks>
    string Key { get; }
}

public abstract class Entity<TId> : IEntity, IEquatable<Entity<TId>>
where TId: struct
{

     [Key]
    public virtual TId Id
    {
        get
        {
            if(_id == null &&typeof(TId) == typeof(Guid))
                _id = Guid.NewGuid();

            return_id == null ? default(TId) : (TId)_id;
        }
        protected set { _id = value; }
    }
    private object _id;

    [Unique, StringLength(50)]
    public virtual string Key
    {
        get{ return _key = _key ?? GenerateKey(); }
        protected set { _key = value; }
    }
    private string _key;

    protected virtual string GenerateKey()
    {
        return KeyGenerator.Generate();
    }
}
```

继承自实体类的子类必须定义自己的ID。注意类中的其他行为，比如如何保证Key属性包含唯一的值，且值必须是50个字符等。另外也要注意，它是如何通过重写equal操作符来判断相同模型对象的等价关系的。

付款模型Payment继承自基类Entity，使用了基于GUID的标识符，包含基元类型和复杂类型的属性。复杂属性用来表示与其他模型之间的关系，例如，Payment类包含Auction和User对象的引用关系：

```
public class Payment : Entity<Guid>
{
    public Currency Amount { get; private set; }
    public Auction Auction{ get; private set; }
    public DateTime Timestamp { get; private set; }
    public User User{ get; set; }

    public Payment(User user, Auction auction, Currency amount)
    {
        User = user;
        Auction = auction;
        Amount = amount;
        Timestamp = Clock.Now;
    }

    private Payment()
    {
    }
}
```

使用域模型的关键就是要把模型分割成一个或者多个上下文，每个上下文定义成聚合群集（aggregate cluster），而每个聚合群集由多个作为单一逻辑单元协同工作的对象组成。

EBuy应用程序的聚合根就是Auction类。Auction类表示应用程序中所有其他类必须使用的类。

图8-2展示了组成EBuy域模型的核心类以及它们之间的关系。因为Auction类是聚合根，所以它与其他核心实体类的关系密切——包括Bid账单类，这是不同用户交易账单的集合；User类表示两种不同的角色（交易员和竞价者）；Payment类表示竞价者对交易员的支付价格。

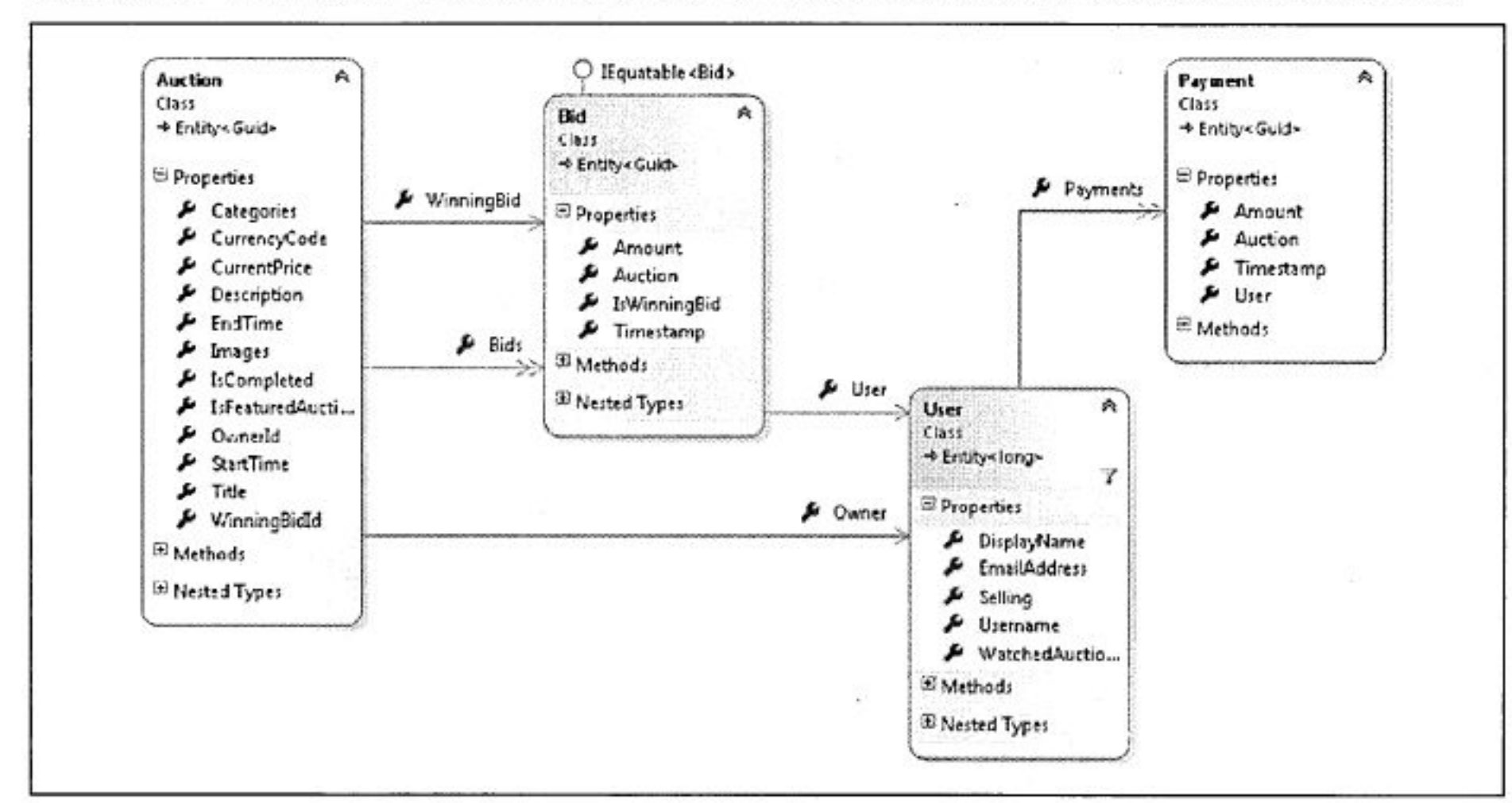


图8-2　领域模型

下面的代码展示了Auction类的内部工作机制以及相关的实体和行为。ICollection<T>用来定义不同的相关类，包括Bid、Category和Image。这个类的主要职责就是竞拍下单。

```
public class Auction : Entity<Guid>
{
    public virtual string Title { get; set; }
    public virtual string Description { get; set; }
    public virtual DateTime StartTime{ get; set; }
    public virtual DateTime EndTime{ get; set; }
    public virtual Currency CurrentPrice{ get; set; }

    public Guid? WinningBidId{ get; set; }
    public virtual Bid WinningBid{ get; private set; }

    public bool IsCompleted
    {
        get{ return EndTime<= Clock.Now; }
    }

    public virtual bool IsFeaturedAuction{ get; private set; }

    public virtual ICollection<Category>Categories { get; set; }

    public virtual ICollection<Bid>Bids { get; set; }

    public virtual ICollection<WebsiteImage>Images { get; set; }

    public long OwnerId{ get; set; }
    public virtual User Owner { get; set; }

    public virtual CurrencyCode CurrencyCode
    {
        get
        {
            return(CurrentPrice!= null) ? CurrentPrice.Code:null;
        }
    }

    public Auction()
    {
        Bids = new Collection<Bid>();
        Categories = new Collection<Category>();
        Images = new Collection<WebsiteImage>();
    }

    public void FeatureAuction()
    {
        IsFeaturedAuction= true;
    }

    public Bid PostBid(User user, double bidAmount)
    {
        return PostBid(user, new Currency(CurrencyCode, bidAmount));
    }

    public Bid PostBid(User user, Currency bidAmount)
    {
```

```
        Contract.Requires(user != null);

        if(bidAmount.Code!= CurrencyCode)
           throw new InvalidBidException(bidAmount, WinningBid);

        if(bidAmount.Value<= CurrentPrice.Value)
           throw new InvalidBidException(bidAmount, WinningBid);

        var bid = new Bid(user, this, bidAmount);

        CurrentPrice= bidAmount;
        WinningBidId= bid.Id;

        Bids.Add(bid);

        return bid;
    }

}
```

使用数据上下文

ADO.NET Entity Framework代码优先方法需要开发人员创建继承自DbContext的数据访问上下文类。这个类包含领域模型里每个实体的属性。自定义实体上下文类可以通过继承自实体上下文的基类来处理特定的查询和保存数据，当然也可以自定义一些实体映射的逻辑代码。

这里的Entity Framework代码优先数据上下文包含两个实体类：类别（Categories）和产品（Products）。下面是DataContext的定义，使用LINQ来查询特定类别的产品。

```
public partial class DataContext: DbContext
{
    public DbSet<Category>Categories { get; set; }
    public DbSet<Product>Products { get; set; }
}

public IList<Product>GetProductsByCategory(Category item)
{
    IList<Product>result = null;

    var db= new DataContext();
    result= db.Products.Where(q =>q.Category.Equals(item)).ToList();

    return resut;
}
```

为了处理实体之间多对多的关系，需要重写数据上下文DataContext的OnModelCreating()方法。下面就是在数据库表账单Bids和交易Auctions之间实现多对多关系的例子：

```
protected override void OnModelCreating(DbModelBuilder modelBuilder)
{
    modelBuilder.Entity<Bid>()
        .HasRequired(x =>x.Auction)
        .WithMany()
        .WillCascadeOnDelete(false);
}
```

默认情况下，Entity Framework会在ASP.NET MVC的配置web.config文件里查找与自定义数据访问类一样的名字的连接字符串：

```
<connectionStrings>
    <add name="Ebuy.DataAccess.DataContext" providerName="System.Data.SqlClient"
        connectionString="Data Source=.\SQLEXPRESS;AttachDbFilename=
                |DataDirectory|\Ebuy.mdf;InitialCatalog=Ebuy;
                Integrated Security=True;User Instance=True;
                MultipleActiveResultSets=True"
    />
</connectionStrings>
```

开发者可以使用repository模式实现（IRepository）接口的repository类，并允许开发人员使用IOC容器来注入repository到控制器中，而不是直接使用Entity Framework数据上下文。

下面是一个repository模式的例子。这里使用构造函数注入的方式向Repository注入Entity Framework数据上下文。

```
public class Repository : IRepository
{
    private readonly DbContext _context;

    public Repository(DbContext context)
    {
        _context = context;
        _isSharedContext= isSharedContext;
    }
}
```

排序、过滤以及数据分页
Sorting, Filtering, and Paging Data

为了支持排序、过滤以及数据分页，ADO.NET Entity Framework需要使用LINQ来与数据库交互。

169

开发者可以创建并根据Entity Framework数据上下文来调用LINQ。一旦LINQ定义完毕，开发人员就可以调用LINQ方法，比如调用ToList()来执行查询。Entity Framework可以把LINQ命令转换为对应的SQL语法并执行查询。执行完毕后，就会把返回结果转变成一个强类型的实体集合。

EBuy网站的查询页面（见图8-3）展示了几种排序、过滤以及数据分页结果。用户可以通过输入关键字来查询、修改排序属性，或者查询结果翻页。这些都是简单的LINQ查询。

在用户输入关键字并点击“提交”按钮后，就会调用SearchController控制器。Index方法会接受SearchCriteria参数。这个类包含所有用户可以修改的搜索关键字段：关键字、排序和页数。

当调用这个方法时，ASP.NET MVC模型绑定器会自动封装查询页面表单域到SearchCriteria类里。除了查询到页面上的域之外，这个类还包含一些快速获取排序字段和页码的方法：

```
public class SearchCriteria
{
    public enum SearchFieldType
    {
        Keyword,
```

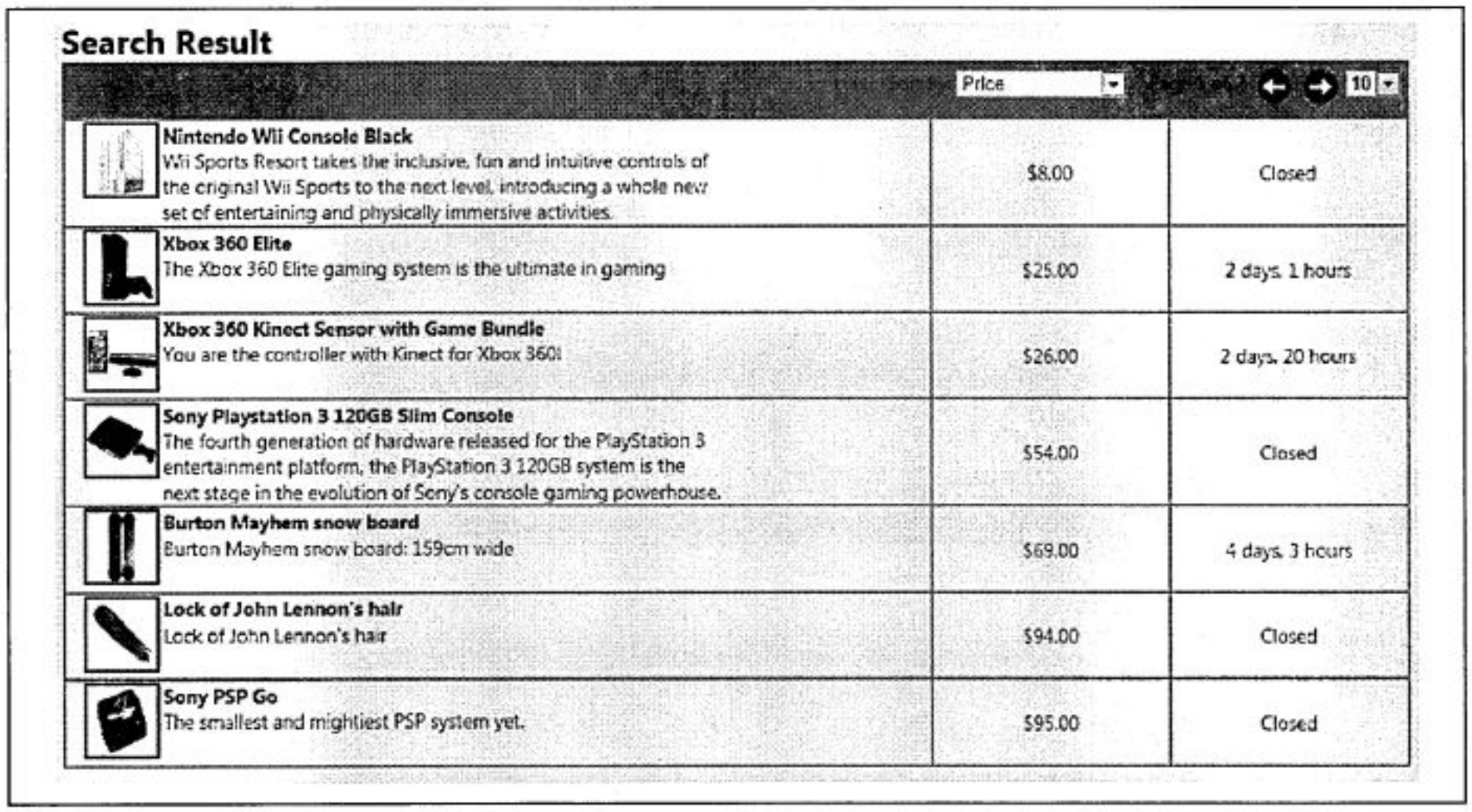


图8-3 EBuy搜索页面

```
        Price,
        RemainingTime
    }

    public string SearchKeyword{ get; set; }
    public string SortByField{ get; set; }
    public string PagingSize{ get; set; }
    public int CurrentPage{ get; set; }

    public int GetPageSize()
    {

        int result = 5;

        if(!string.IsNullOrEmpty(this.PagingSize))
        {
            int.TryParse(this.PagingSize, out result);
        }

        return result;
    }

    public SearchFieldType GetSortByField()
    {
        SearchFieldType result = SearchFieldType.Keyword;

        switch(this.SortByField)
        {
            case"Price":
                result= SearchFieldType.Price;
                break;
            case "Remaining Time":
                result= SearchFieldType.RemainingTime;
```

```
                break;
            default:
                result= SearchFieldType.Keyword;
                break;
        }

        return result;
    }
}
```

例8-1引入了一些新的概念，我们现在应该了解一下。视图使用了jQuery事件来关联查询条件字段，所以当这些条件改变时，跟踪当前视图的隐藏域控件就会提交当前表单到SearchController控制器。视图也引用了包含查询字段和结果的模型类SearchViewModel。

例8-1　查询视图

```
@model SearchViewModel

@{
    ViewBag.Title= "Index";
}

<script type="text/javascript">

    $(function () {
        $("#SortByField").change(function () {
            $("#CurrentPage").val(0);
            SubmitForm();
        });

        $("#PagingSize").change(function () {
            $("#CurrentPage").val(0);
            SubmitForm();
        });

        $("#Previous").click(function () {
            varcurrentPage= $("#CurrentPage").val();
            if(currentPage!= null &&currentPage>0) {
                currentPage--;
                $("#CurrentPage").val(currentPage);
            }
            SubmitForm();
        });

        $("#Next").click(function () {
            varcurrentPage= $("#CurrentPage").val();
            if(currentPage) {
                currentPage++;
                $("#CurrentPage").val(currentPage);
            }
            SubmitForm();
        });
    });

    function SubmitForm() {
        document.forms["SearchForm"].submit();
    }
```

```
</script>

@using (Html.BeginForm("Index", "Search", FormMethod.Post, new { id = "SearchForm" }))
{

    <div class="SearchKeyword">
        @Html.TextBoxFor(m =>m.SearchKeyword, new {@class="SearchBox"})
        <input id="Search" type="submit" value=" " class="SearchButton" />
    </div>

    <h2>Search Result</h2>

    <div>
        <div class="SearchHeader">
            @Html.Hidden("CurrentPage", @Model.CurrentPage)
        <div class="PagingContainer">
            <span class="CurrentPage">Page @Model.CurrentPage of @Model.MaxPages</span>
            <img id="Previous" src="@Url.Content("~/Content/Images/PagingPrevious.png")"↵
            class="PagingButton" />
            <img id="Next" src="@Url.Content("~/Content/Images/PagingNext.png")" ↵
            class="PagingButton" />
            <div class="PageSize">
                @Html.DropDownListFor(m =>m.PagingSize, new SelectList↵
                (Model.PagingSizeList))
            </div>
        </div>
        <div class="SortingContainer">
            <span>Sort By:</span>
            @Html.DropDownListFor(m =>m.SortByField, new SelectList↵
            (Model.SortByFieldList))
        </div>
    </div>
    <div class="SearchResultContainer">
        <table>
            @foreach(var item in @Model.SearchResult)
            {
                var auctionUrl= Url.Auction(item);
                <tr>
                    <td class="searchDescription">
                        <div class="fieldContainer">
                            <a href="@auctionUrl">@Html.SmallThumbnail(@item.Image, ↵
                            @item.Title)</a>
                        </div>
                        <div class="fieldContainer">
                            <div class="fieldTitle">@item.Title</div>
                            <div class="fieldDescription">
                                @item.Description
                            </div>
                        </div>
                    </td>
                    <td class="centered-field">@item.CurrentPrice</td>
                    <td class="centered-field">@item.RemainingTimeDisplay</td>
                </tr>
            }
        </table>
```

```
        </div>
    </div>

}
```

当用户输入查询关键字或者修改查询条件后，就会调用SearchController控制器。Index操作方法负责处理进入的请求消息，检查传入的模型是否包含查询条件。如果用户输入了关键字，它就会使用这个关键字来过滤查询结果。Repository类的Query()方法用来发送过滤数据（比如查询关键字）给Entity Framework数据上下文，以构建SQL查询语句并返回过滤后的数据给控制器。

173

```
    public ActionResult Index(SearchCriteria criteria)
    {
        IQueryable<Auction>auctionsData= null;

        // 根据关键字过滤 auctions 交易数据
        if(!string.IsNullOrEmpty(criteria.SearchKeyword))
            auctionsData= _repository.Query<Auction>(q =>q.Description.Contains↵
            (criteria.SearchKeyword));
        else
            auctionsData= _repository.All<Auction>();

        // 加载视图代码

        return View(viewModel);
    }
```

当用户修改了排序字段时，就会重新激活SearchController控制器。这个控制器会检查SearchCriteria类以确定排序使用的字段。为了实现数据排序，可以通过q⇒q.CurrentPrice.Value方式传递排序字段来调用LINQ OrderBy命令，示例代码如下。

```
    switch(criteria.GetSortByField())
    {
        caseSearchCriteria.SearchFieldType.Price:
            auctionsData= auctionsData.OrderBy(q =>q.CurrentPrice.Value);
            break;
        caseSearchCriteria.SearchFieldType.RemainingTime:
            auctionsData= auctionsData.OrderBy(q =>q.EndTime);
            break;
        caseSearchCriteria.SearchFieldType.Keyword:
        default:
            auctionsData= auctionsData.OrderBy(q =>q.Description);
            break;
    }
```

当用户点击“上一页”或者“下一页”按钮，或者点击不同的显示页面时，也会调用SearchController控制器。控制器会通过检查criteria参数来决定显示哪一页的数据。PageSearchResult()方法随后会调用自定义扩展方法Page()来确定有多少要显示的交易数据auctions。

如果交易auctions数量大于要显示的页数，那么会返回请求的页面对应的数据。如果请求页面大于等于实际的数据页面数量，就会返回所有符合条件的请求交易数据：

```
    private IEnumerable<Auction>PageSearchResult(SearchCriteria criteria, IQueryable↵
    <Auction>auctionsData)
```

```
{
    IEnumerable<Auction>result = null;

    var NumberOfItems= auctionsData.Count();

    if(NumberOfItems>criteria.GetPageSize())
    {
        var MaxNumberOfPages= NumberOfItems/ criteria.GetPageSize();

        if(criteria.CurrentPage>MaxNumberOfPages)
           criteria.CurrentPage= MaxNumberOfPages;

        result= auctionsData.Page(criteria.CurrentPage, criteria.GetPageSize());
    }
    else
    {
        result= auctionsData.ToArray();
    }

    returnresult;
}
```

为了便于实现翻页，这里实现了一个IEnumerable<T>的扩展方法。这个方法可以使用LINQ的Skip和Take命令参数根据当前页面和页面大小来返回特定页面的数据。

```
public static class CollectionExtensions
{
    public static IEnumerable<T>Page<T>(this IEnumerable<T>source,↵
    int pageIndex, int pageSize)
    {
        Contract.Requires(pageIndex>= 0, "Page index cannot be negative");
        Contract.Requires(pageSize>= 0, "Page size cannot be negative");

        int skip = pageIndex* pageSize;

        if(skip >0)
            source= source.Skip(skip);

        source= source.Take(pageSize);

        return source;
    }
}
```

总结

Summary

本章首先介绍了数据访问模式以及如何使用ADO.NET Entity Framework；接着详细介绍了Entity Framework提供的几种不同的数据访问方法，并讲解了如何使用Code First方法构建数据访问层；最后介绍了如何使用POCO类以及repository模式构建ASP.NET MVC Web应用程序。

第9章 安全

Security

本章首先将会详细讨论如何构建安全的ASP.NET MVC Web应用程序，包括如何确保Web应用的安全；其次将介绍互联网、局域网以及其他Web应用的不同之处；最后介绍如何使用.NET框架内置的安全机制来处理常见的Web应用安全问题。

构建安全的Web应用

Building Secure Web Applications

本杰明·富兰克林（Benjamin Franklin）曾经说过“一盎司的预防抵得上一磅的治疗”（译注1）。这句话所传达的理念正是技术人员在构建安全的Web应用程序时应该考虑的：世界危险，人心叵测，Web应用随时都可能成为攻击者的目标。最正确、最廉价的做法就是：提前做好防备。

非常不幸的是，目前还没有出现可以解决Web应用程序安全问题的“万能银弹”。这不像引用类库或者调用方法那么简单。安全需要自始至终都验证请求的资源，这样才能确保系统的安全。

下面介绍一些适用于ASP.NET MVC Web应用的安全原则。如果在设计、开发Web应用时遵守这些原则，就可以避免一些常见且严重的安全问题，从而提升系统的安全性。

深度防御

虽然网站是直接与外界交互的唯一应用层，但是并不意味着只有这一层才可以加强网站安全。恰恰相反，保护整个系统的安全需要所有层或者各个子系统各自负责好自己的安全，充当合格的门卫。通常来说，特定的层只能被信任的层调用，但是实际情况往往比这个复杂！相反，每层都应该像要直接与外界交互一样，在执行任何操作之前都进行验证并且授权用户。

译注1：“an ounce of prevention is worth a pound of cure”直接翻译是“一盎司的预防抵得上一磅的治疗”。比喻小成本的预防投入胜过事后巨额的补救。

不信任任何输入数据

任何用户数据或者来自其他系统的输入数据都应该被当做潜在的威胁，所以在使用之前要确保进行了校验操作。也不要轻易相信在其他地方已经校验过的数据。

例如，客户端在浏览器里使用的JavaScript校验机制，虽然带来了良好的用户体验，但是这只是客户端的防御措施，因为攻击者可以轻易向服务器提交伪造的请求。客户端校验可以作为一种便利措施，但不是安全的机制，所以控制器应该再次校验它们收到的数据。

执行最小权限原则

使用只包含任务所需权限的账户来执行代码（最小权利原则），除非特别需求，否则不允许提升权限。在需要提升权限的情况下，尽可能限制提升权限的有效期：一旦完成任务，立即删除权限。例如，为了保存上传到服务器的文件，不要试图在管理员权限下运行整个网站，而是创建一个只能访问特定目录的账号，并且这个账号只能用于创建文件，而不能用于删除、修改或者执行的权限。

假设外部系统是危险的

当外部计算机或者外部应用程序需要和系统交互时，可把它当做一个和普通用户一样的角色对待。当系统需要彼此交互时，可考虑使用不同的账号，并且限制每个账号所执行的操作。

减少裸露面

避免给外界暴露不必要的信息。例如，ASP.NET MVC应用在满足需求的情况下,尽量减少暴露的操作数量以及每步操作需要的数据参数。

ASP.NET MVC的模型绑定标记属性BindAttribute提供了Include和Exclude属性，允许我们指定逗号分隔的模型属性列表，从而标记这些属性是被映射或者忽略。

同样，记录和处理网站抛出的任何异常信息，都不要直接返回任何攻击者可以利用的系统信息，例如，文件路径、账户名称或者数据库schema信息等。

关闭不必要的功能

最常见的攻击就是对主流平台或者服务的常见安全漏洞进行攻击。为了避免成为攻击目标，最好的办法就是卸载或者关闭应用程序不需要的功能或者服务。

例如，如果应用程序不发送Email，就应该关闭服务器上的所有邮件服务。

保护程序

Securing an Application

Web应用经常要面对几种不同的用户，首先是终端用户，即网站的主要用户群；其次是执行

监控和部署网站等相关工作的管理员；最后是用于应用程序不同层之间通信或者与外部系统进行交互的用户账号。

总体来说，在设计ASP.NET Web应用时,首先要考虑的问题就是网站需要的验证模型；其次就是把网站的功能分割成不同需要的安全验证角色。在深入学习之前，先了解这几个重要的术语。

验证

验证就是鉴定应用程序访问者身份的过程。验证回答了以下问题：当前用户是谁？这个用户是否是有效的？ASP.NET和ASP.NET MVC同时支持两种验证方式：Windows验证和Form验证（表单验证）。

授权

授权是决定验证过的用户应该拥有何种级别的访问安全资源的权限。ASP.NET MVC允许在程序中使用AuthorizeAttribute标记属性来检查用户是否具备特定的权限。一旦定义好整个应用程序的安全模型，就应该考虑各个层之间如何通信了。

应用程序间通信最流行的方法之一就是给每层单独创建一个最小权限的应用程序服务账号。

例如，如果Web程序只需要查询数据和显示报表的功能，就给服务账户授权拥有只读本地的文件系统和数据库的权限。

当需要读/写权限时，也可以只针对服务账号授权——只允许访问特定系统或者功能的授权，例如，限制上传到单个文件夹、只允许服务账号更新或者写入特定的数据库表。

保护局域网应用

局域网（Intranet）和外联网（Extranet）（译注2）Web应用最常用的验证方式就是Windows验证。使用Windows验证时，用户的Windows安全令牌在用户访问整个Web网站期间使用HTTP请求,进行消息发送。

应用程序会使用这个令牌在本机（或者域）里验证用户账户的有效性，也会评估用户所在角色所具备的权限。当用户验证失败或者未授权时，浏览器就会定向到特定的页面让用户输入自己的安全凭证（用户名和密码）。

配置Windows验证

使用ASP.NET MVC创建支持Windows验证的Web应用程序非常简单。直接使用ASP.NET MVC 4的 Intranet Application（局域网应用）模板就可以了，如图9-1所示。这个模板的web.config文件包含了`<authentication mode="Windows" />`设置。

为了部署ASP.NET MVC局域网应用程序，首先要配置服务器为Windows验证方式。下面几节会详细介绍如何在IIS和IIS Express Web服务器上配置Windows验证。

译注2：Extranet，英语译为外联网。有关准确的 Extranet 定义还在讨论中，但大多数人都能接受的 Extranet 定义是：一种通过使用 Internet/Intranet 技术使企业与客户及其他企业相连来完成共同目标的合作网络。Extranet 可以作为公用的 Internet 和专用的 Intranet 之间的桥梁，也可以看作是一个能被企业成员访问或与其他企业合作的企业 Intranet 的一部分。

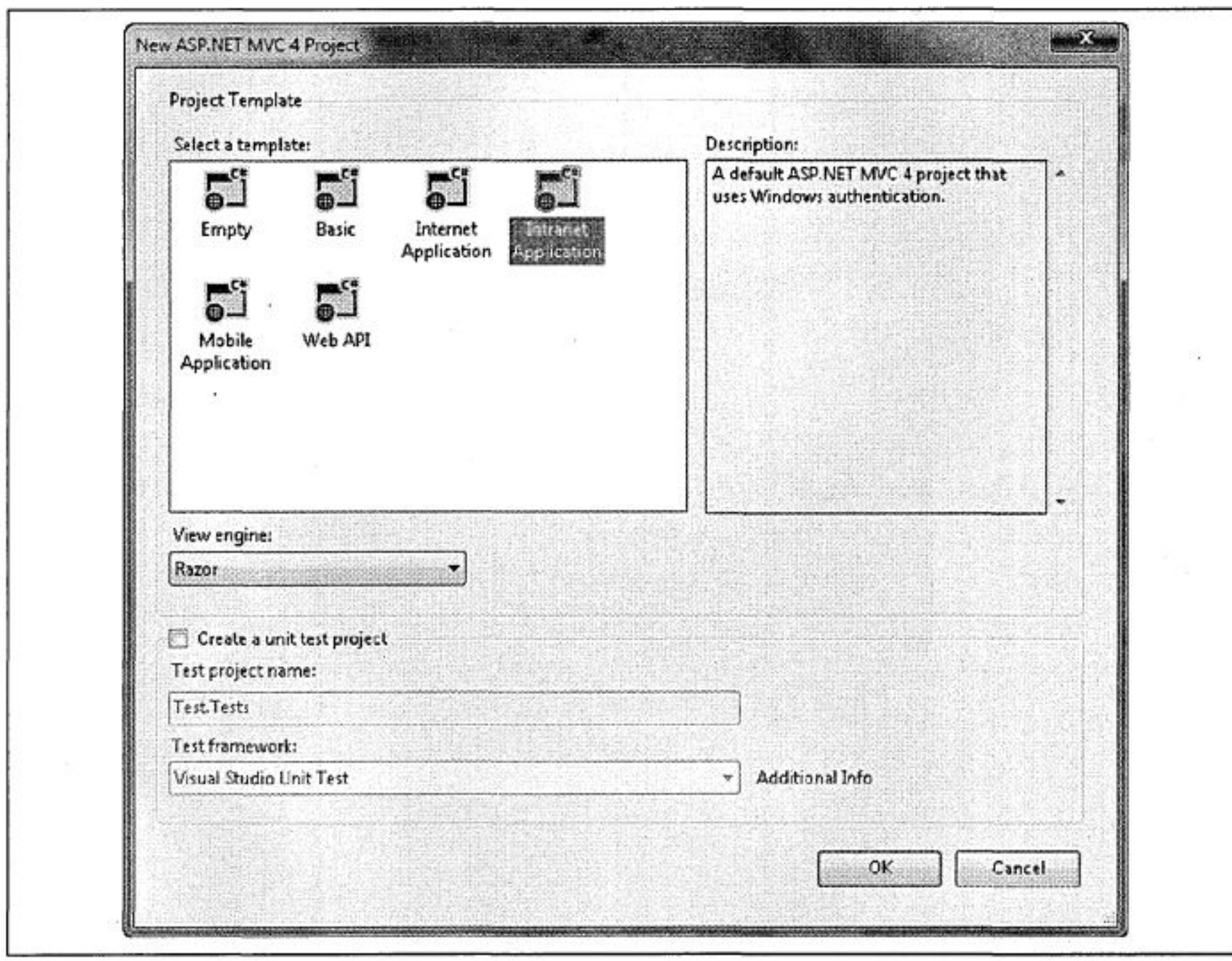


图9-1　创建ASP.NET MVC局域网应用程序

Visual Studio内置的Web服务器非常适合本地部署，但是它处理Windows验证失败的方式和IIS不同。所以，为了部署需要Windows验证的Web应用程序，必须使用IIS Express或者IIS 7.0+以上版本的Web服务器。

配置IIS Express

下面介绍了如何配置IIS Express，从而支持Windows验证的ASP.NET Web应用程序。

1. 在Visual Studio解决方案浏览器(Solution Explorer）中，右键单击ASP.NET web项目，从菜单里选择“使用IIS Express...”（见图9-2），然后在对话框里点击“是”按钮。
2. 返回解决方案浏览器(Solution Explorer)，选择需要的项目，按F4键，显示项目的属性。然后设置“匿名验证（Anonymous Authentication）”为“关闭（Disabled）”。最后启用Windows验证，如图9-3所示。

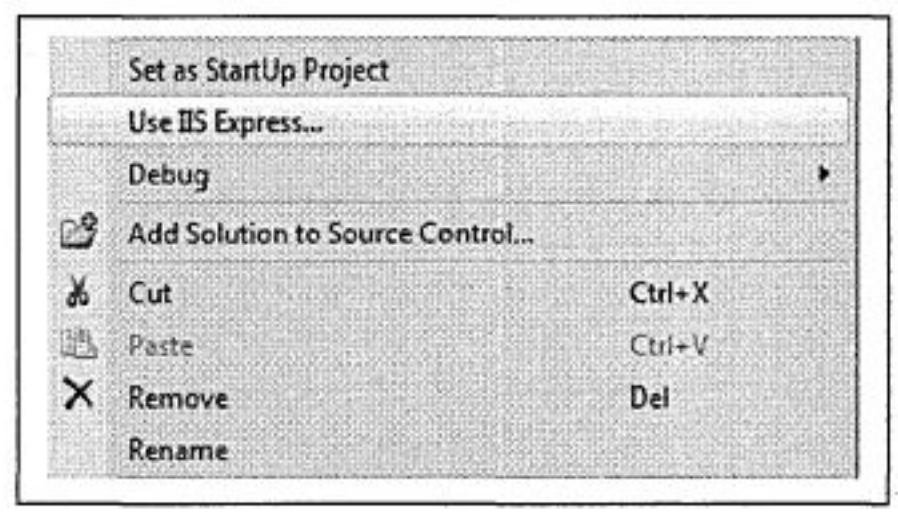


图9-2　选择IIS Express

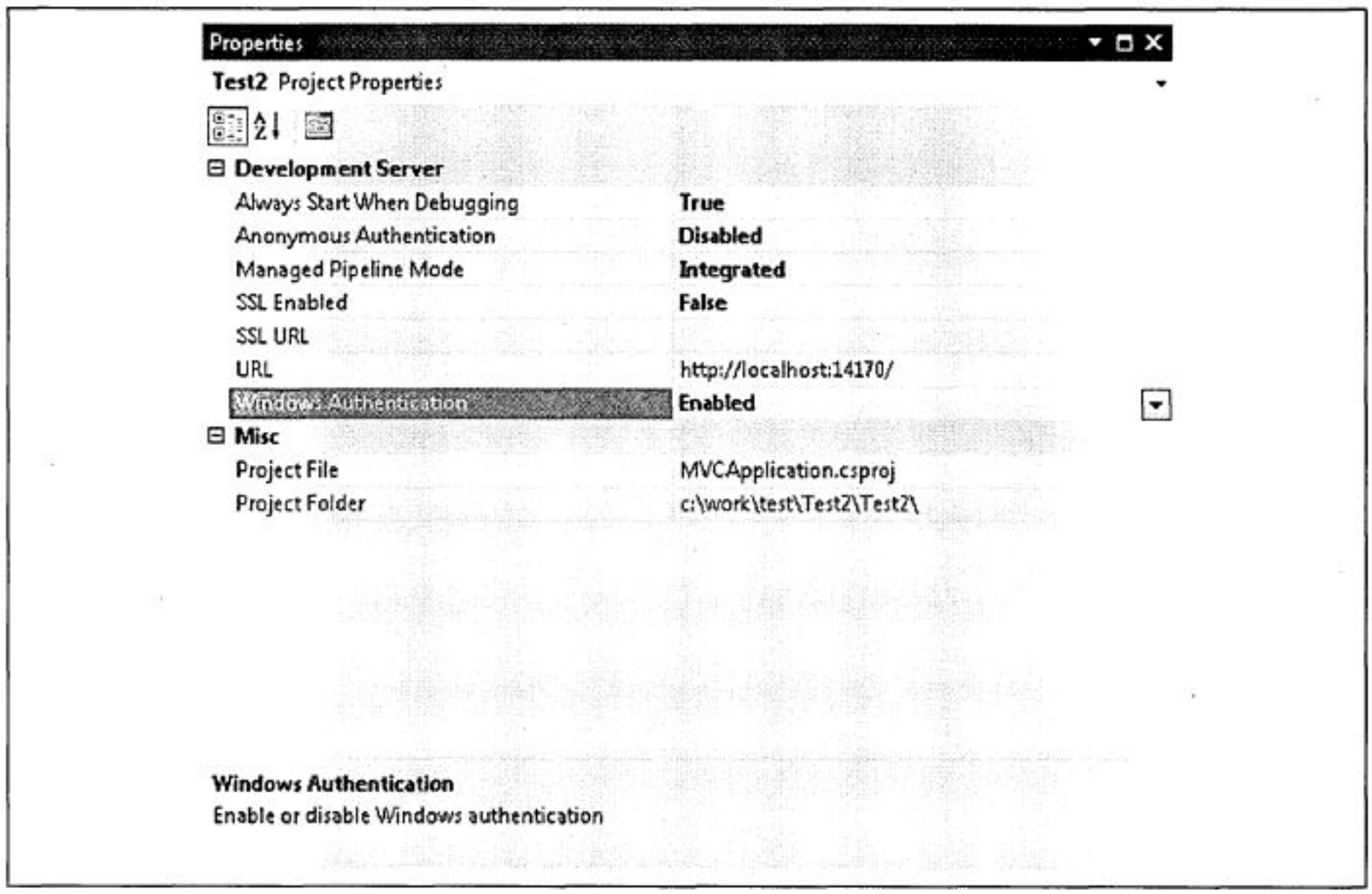


图9-3 配置项目属性

配置IIS 7.0

在IIS 7.0里配置Windows验证与IIS Express的方式有点不一样。以下为在IIS 7.0里配置Windows验证的步骤。

1. 打开IIS 7.0管理器，用右键单击任一网站（比如默认的Web Site站点），如图9-4所示，选择“添加Web应用”。

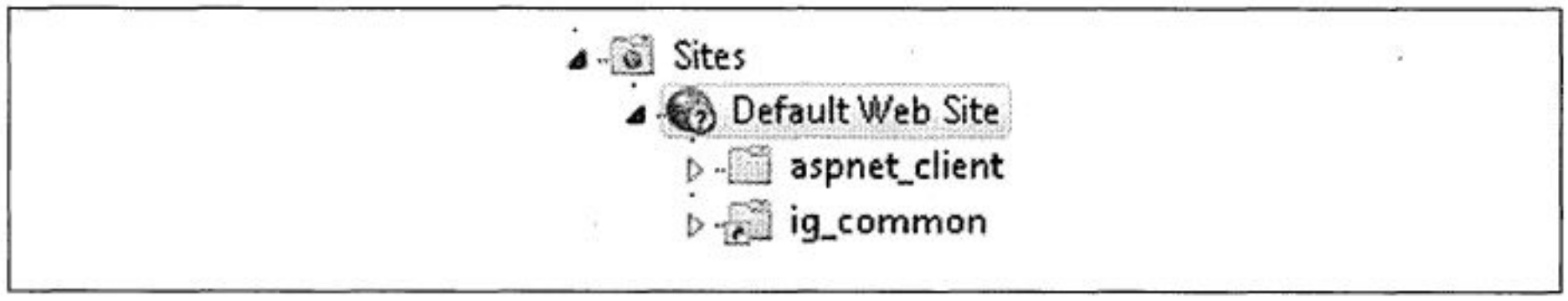


图9-4 选择要Windows验证的网站

2. 在添加应用的对话框里（见图9-5）填写网站简称，选择应用程序池（要选ASP.NET 4.0版本的）以及网站文件夹的物理路径。
3. 选中刚创建的网站，点击“验证”选项，然后关闭匿名验证，打开Windows验证，如图9-6所示。

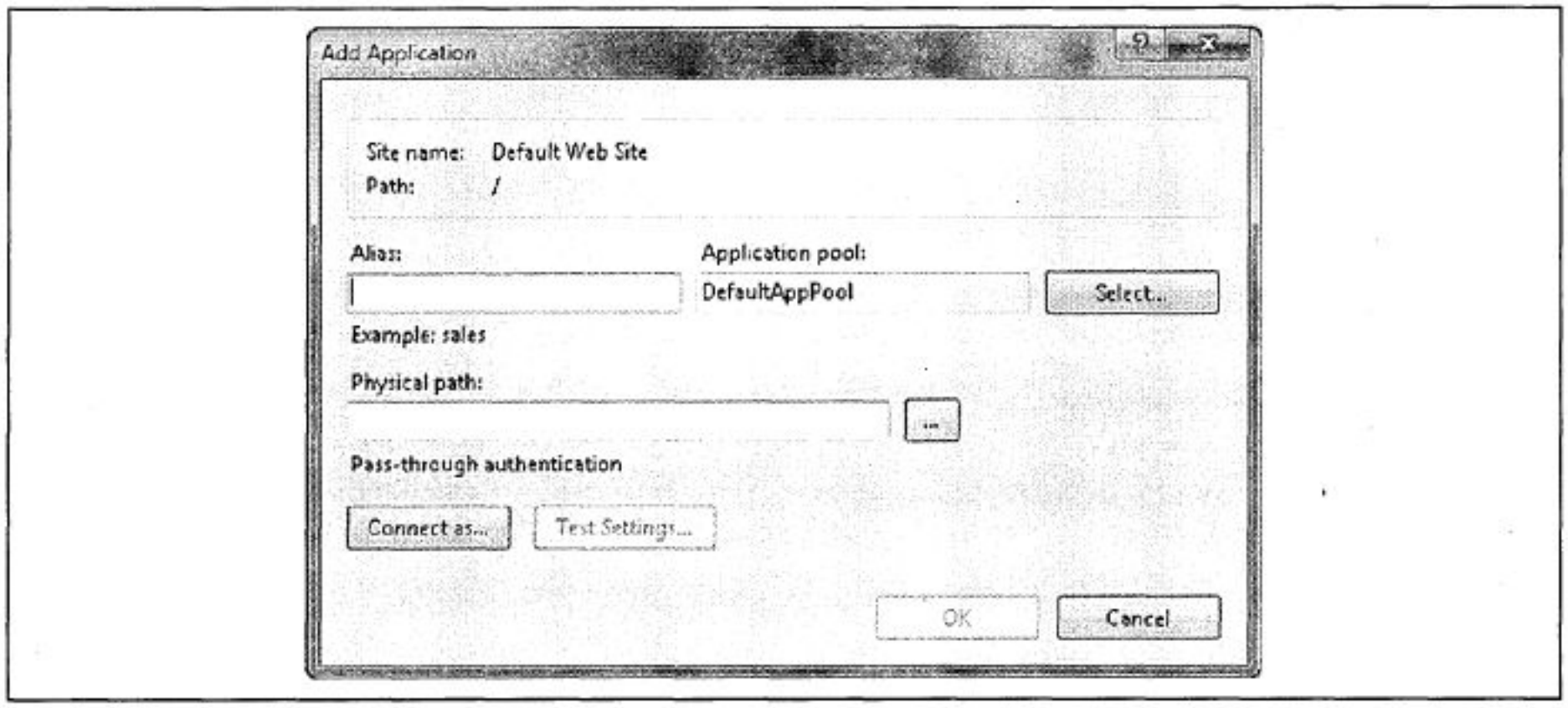


图9-5 IIS管理器添加网站

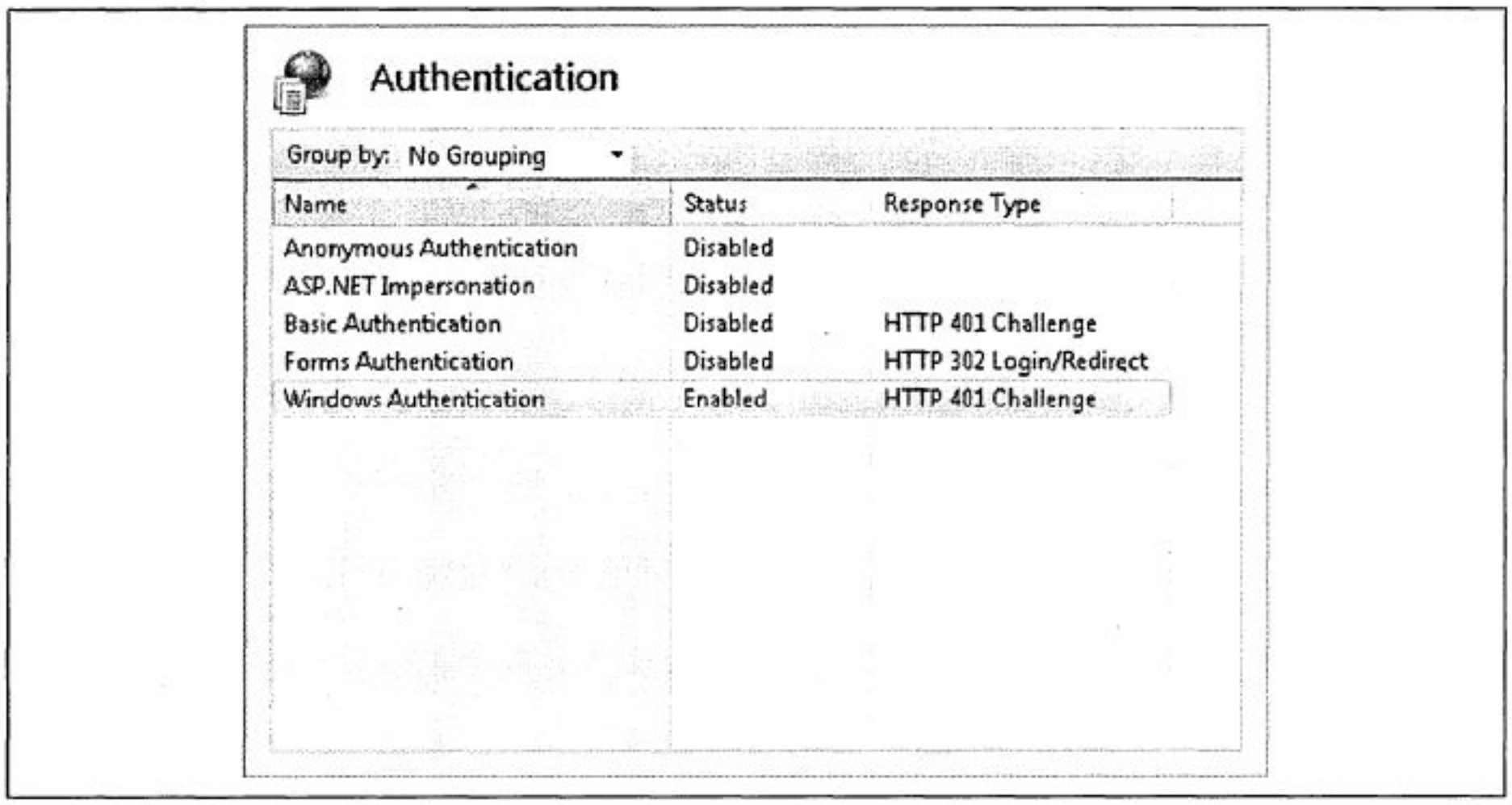
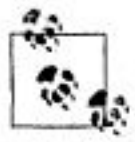


图9-6 配置验证方式

使用AuthorizeAttribute标记属性

`AuthorizeAttribute`允许我们显式地限制对控制器的访问，没有权限的用户无论何时试图访问控制器都会被拒绝。



如果Visual Studio内置的Web服务器托管了该网站，就会返回空白页，因为它不支持Windows验证的错误处理。

表9-1列出了`AuthorizeAttribute`的有效属性。

表9-1 AuthorizeAttribute的属性

属性	描述
Order	定义执行过滤器的顺序（继承自FilterAttribute）
Roles	获取或者设置允许访问控制器操作的角色
Users	获取或者设置允许访问控制器操作的用户名

下面的代码用于设置控制器操作只允许特定用户访问。这里只允许Company\Jess and Company\Todd两个账号访问AdminProfile控制器操作，并且禁止其他账号访问。

183

```
[Authorize(Users = @"Company\Jess, Company\Todd")]
publicActionResultAdminProfile()
{
    returnView();
}
```

尽管这个例子指定了一个用户名列表，但是通常推荐使用Windows用户组来替代。这样会更方便应用程序管理配置，因为相同的Windows用户组可以在多个地方使用，并且Windows用户组的成员可以方便修改。

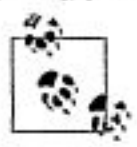
要使用Windows用户组代替用户名列表，只需要在操作上设置Role角色属性为Windows用户组即可：

```
[Authorize(Roles = "Admin, AllUsers")]
public ActionResult UserProfile()
{
    return View();
}

[Authorize(Roles = "Executive")]
public ActionResult ExecutiveProfile()
{
    return View();
}
```

表单验证

Windows验证的局限性非常明显，一旦有超出本地域控制器范围的外网用户访问网站，就会出现问题。此时互联网网站需要使用ASP.NET**表单验证**（Forms Authentication）。

为了使用ASP.NET表单验证方法，ASP.NET需要验证加密过的HTTP cookie（如果cookie被禁用，则使用查询字符串）来识别用户的所有请求。cookie与ASP.NET 会话机制（session）的关系密切，在会话超时或者用户关闭浏览器后，会话和cookie就会失效，用户需要重新登录网站建立新的会话。



强烈推荐SSL与ASP.NET表单验证结合使用。SSL可以自动加密用户的敏感信息，并且保证其他数据的可读性。更多关于SSL的介绍，可以阅读文章：*http://learn.iis.net/page.aspx/144/how-to-set-up-ssl-on-iis/*。

184

绝大部分ASP.NET MVC Web模板（除了局域网应用模板外）都推荐使用ASP.NET表单验证。

ASP.NET MVC的互联网应用模板更是实现了默认的表单验证功能，生成了AccountController控制器以及相关的登录视图文件。

图9-7所示为默认的ASP.NET互联网网站模板的登录页面，包含一张登录表单，登录表单又包含用户名、密码及“记住我（Remember me？）”复选框和注册新账户的连接。

默认的新建账户页面如图9-8所示，包含内置的密码验证以及密码确认等功能，并且密码长度不能少于6个字符。

可以考虑把这些视图当做整个网站的起点，并且可以根据需要随意修改这些代码。

图9-7 ASP.NET互联网网站模板默认的登录表单

AccountController

`AccountController`支持典型的ASP.NET表单验证场景。它基本上实现了包括注册用户、验证用户以及修改密码等主要功能。

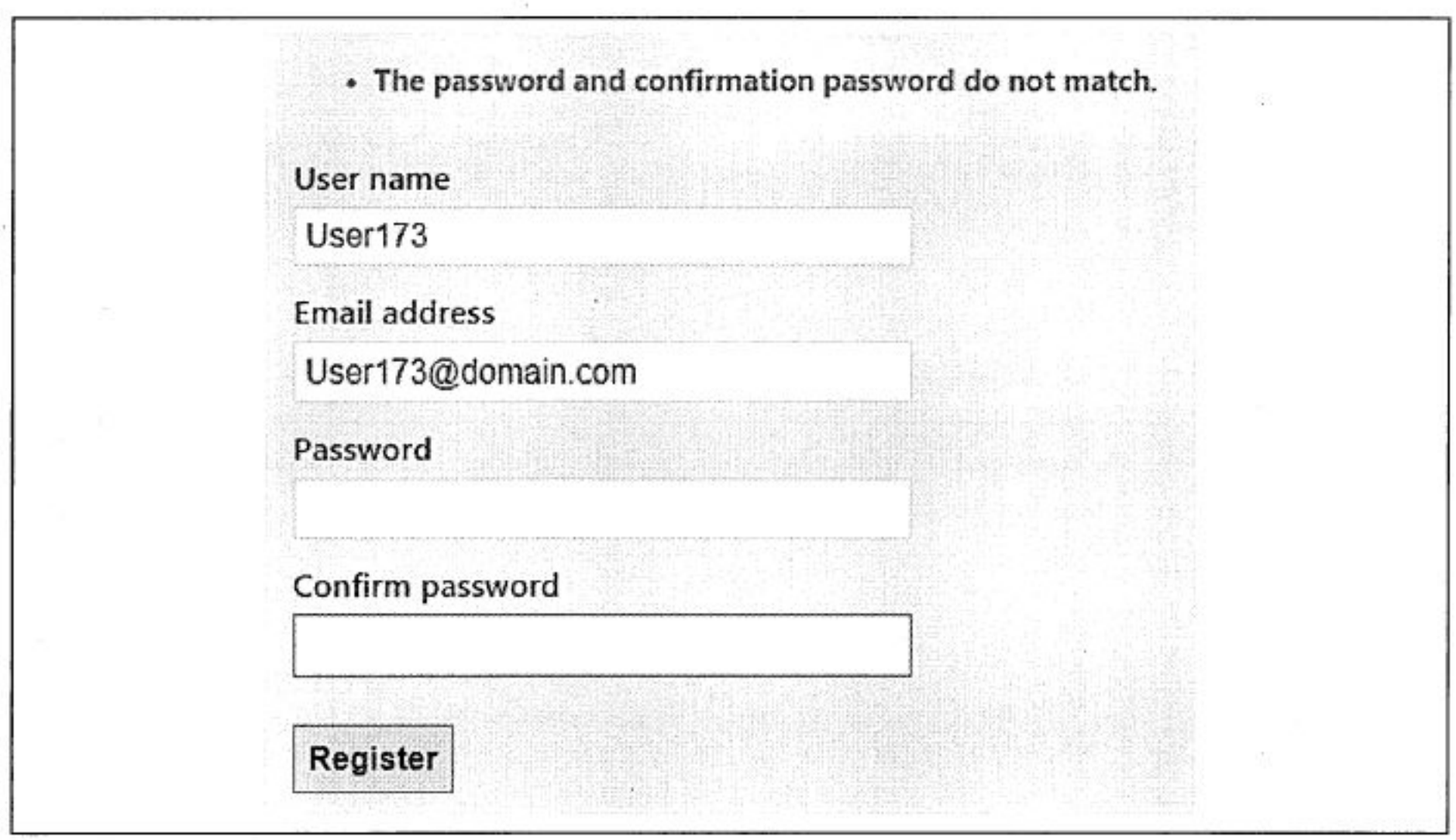


图9-8 默认的新建账户页面

这些功能背后的实现原理就是借助标准的ASP.NET成员资格和角色提供程序（ASP.NET Membership and Role Provider）（译注3）。这是标准的ASP.NET验证授权框架，可以直接在web.config文件里配置使用。默认情况下，网站的用户账号数据都会存储在网站/App_Data文件夹下的SQL Express类型的ASPNETDB.MDF数据库文件里。因为SQL Express只适用于开发，所以可能需要在生产环境中使用完全版的Microsoft SQL Server，甚至活动目录Active Directory。当然也可以自定义成员资格和角色提供程序来替代默认的ASP.NET成员资格和角色提供程序。

幸运的是，假如要修改默认的成员资格和角色提供程序，甚至使用自定义的成员资格和角色提供程序，只需要简单修改web.config文件即可，非常方便。

```
<membership defaultProvider="DefaultMembershipProvider">
  <providers>
    <add name="DefaultMembershipProvider" type="System.Web.Providers.DefaultMembership↵
        Provider, System.Web.Providers, Version=1.0.0.0, Culture=neutral, PublicKey↵
        Token=31bf3856ad364e35"
        connectionStringName="DefaultConnection" enablePasswordRetrieval="false" ↵
        enablePasswordReset="true"
        requiresQuestionAndAnswer="false" requiresUniqueEmail="false" maxInvalid↵
        PasswordAttempts="5"
        minRequiredPasswordLength="6" minRequiredNonalphanumericCharacters="0" password↵
       AttemptWindow="10" applicationName="/" />
  </providers>
</membership>
```

验证用户

当用户访问限制的页面时，就会自动重定向到登录页面。`AccountController`会使用`AllowAnonymousAttribute`来允许匿名用户访问。如果不允许匿名用户访问登录页面，那么任何用户都无法登录这个网站了！

因为标准的HTML登录表单和AJAX表单都可以访问`Login`控制器操作，所以控制器操作会根据收到的查询字符串content参数来判断请求来自哪种登录表单。如果`content`不为null，则为AJAX请求消息，所以操作方法返回AJAX版本的登录视图；否则为标准的HTTP请求，操作方法返回完整的登录表单视图。完整的用户登录操作代码如下：

```
[AllowAnonymous]
public ActionResult Login()
{
    return ContextDependentView();
}

private ActionResult ContextDependentView()
{
    string actionName= ControllerContext.RouteData.GetRequiredString("action");
```

译注3：ASP.NET 成员资格和角色提供程序提供了一种验证和存储用户凭据的内置方法。我们将ASP.NET 成员资格与 ASP.NET Forms 身份验证或 ASP.NET 登录控件一起使用以创建一个完整的用户身份验证系统。可以将成员资格信息（用户名、密码和支持数据）存储在 Microsoft SQL Server、Active Directory 或其他数据存储区，允许自定义扩展。WCF 也可以使用。更多介绍可以参考 *http://msdn.microsoft.com/zh-cn/library/yh26yfzy(v=vs.80).aspx*。

```
        if(Request.QueryString["content"] != null)
        {
            ViewBag.FormAction= "Json" + actionName;
            return PartialView();
        }
        else
        {
            ViewBag.FormAction= actionName;
            return View();
        }
    }
```

当用户输入安全账号并提交后，登录表单就会向服务器中的Login操作提交凭证信息以便验证账号。

使用ASP.NET表单验证功能验证用户的操作包含两步。

1. Login操作调用Membership.ValidateUser()方法来验证用户的有效性。
2. 如果验证用户有效，则Login操作会调用FormsAuthentication.SetAuthCookie()来创建用户的安全令牌（security token）（译注4）。

如果用户登录成功，就会重定向到最初访问的URL；如果验证出错，就会重新跳转到登录页面或者返回主页。整段代码如下：

```
    [AllowAnonymous]
    [HttpPost]
    public ActionResult Login(LoginModel model, string returnUrl)
    {
        if(ModelState.IsValid)
        {
            if(Membership.ValidateUser(model.UserName, model.Password))
            {
                FormsAuthentication.SetAuthCookie(model.UserName, model.RememberMe);
                if(Url.IsLocalUrl(returnUrl))
                {
                    return Redirect(returnUrl);
                }
                else
                {
                    return RedirectToAction("Index", "Home");
                }
            }
            else
            {
                ModelState.AddModelError("", "The user name or password provided is ↵
                incorrect.");
            }
        }

        // 如果出现失败，就会重新显示注册表单
        return View(model);
    }
```

译注4：很多技术人员可能听说过安全令牌（security token）这个词语。这个英文单词经常出现在与安全相关的技术资料中，如《WCF服务编程》（第三版）一书中有提到，“微软WCF分布式安全开发”课程也多次提到。

注册新用户

在网站验证用户之前，用户首先要拥有自己的账号。虽然网站管理员可以手动创建账号，但是最常见的情况还是用户注册自己的账号。ASP.NET MVC互联网网站模板提供了默认的Register控制器操作来负责这项工作。它首先会收集用户输入的账号数据，然后使用ASP.NET成员资格和角色提供程序来创建账号。

Register控制器操作和Login操作很像——都使用AllowAnonymous标记属性来允许匿名用户访问，并且可以根据请求来源是否是AJAX来决定是返回部分视图还是完整视图。

Register操作调用ASP.NET成员资格和角色提供程序的Membership.CreateUser()API方法来创建新账号。

当用户注册成功后，操作会使用FormsAuthentication.SetAuthCookie()方法来自动验证新用户，并重新跳转到网站的主页。

```
[AllowAnonymous]
public ActionResult Register()
{
    return ContextDependentView();
}

[AllowAnonymous]
[HttpPost]
public ActionResult Register(RegisterModel model)
{
    if(ModelState.IsValid)
    {
        // 尝试注册用户
        MembershipCreateStatus createStatus;
        Membership.CreateUser(model.UserName, model.Password, model.Email, password↵
        Question: null, passwordAnswer: null, isApproved: true, providerUserKey: null, ↵
        status: out createStatus);

        if(createStatus== MembershipCreateStatus.Success)
        {
            FormsAuthentication.SetAuthCookie(model.UserName, createPersistentCookie:↵
        false);
            return RedirectToAction("Index", "Home");
        }
        else
        {
            ModelState.AddModelError("", ErrorCodeToString(createStatus));
        }
    }

    // 如果出现失败，就会重新显示注册表单
    return View(model);
}
```

修改密码

AccountController提供了许多表格验证程序都能提供的功能：修改密码。

189

这个过程从发送消息到ChangePassword操作开始。控制器会使用Membership.GetUser API来获取用户的MembershipUser类型的账户信息实例，它包含这个用户的验证信息。

如果成功获取用户账号信息，ChangePassword操作就会调用MembershipUser的ChangePassword()方法来修改密码。

在修改密码成功以后，就会跳转到ChangePasswordSuccess视图，并通知用户：密码修改成功。

```
public ActionResult ChangePassword()
{
    return View();
}

[HttpPost]
public ActionResult ChangePassword(ChangePasswordModel model)
{
    if(ModelState.IsValid)
    {

        // 某些失败场景里，ChangePassword会抛出异常，而不是返回false
        bool changePasswordSucceeded;
        try
        {
            MembershipUser currentUser=Membership.GetUser(User.Identity.Name, ↵
            userIsOnline: true);
            changePasswordSucceeded= currentUser.ChangePassword(model.OldPassword, ↵
            model.NewPassword);
        }
        catch(Exception)
        {
            changePasswordSucceeded= false;
        }

        if(changePasswordSucceeded)
        {
            return RedirectToAction("ChangePasswordSuccess");
        }
        else
        {
            ModelState.AddModelError("", "The current password is incorrect or the new↵
            password is invalid.");
        }
    }

    // 如果出错
    return View(model);
}
```

190

与AJAX交互

除了标准的HTML GET/POST模型，AccountController也支持通过AJAX登录账号和注册用户。下面的代码块实现了与AJAX交互。

```
[AllowAnonymous]
[HttpPost]
public JsonResult JsonLogin(LoginModel model, string returnUrl)
{
    if(ModelState.IsValid)
    {
        if(Membership.ValidateUser(model.UserName, model.Password))
        {
            FormsAuthentication.SetAuthCookie(model.UserName, model.RememberMe);
            return Json(new { success = true, redirect = returnUrl});
        }
        else
        {
            ModelState.AddModelError("", "The user name or password provided is ↵
           incorrect.");
        }
    }

    // 如果出错
    return Json(new { errors = GetErrorsFromModelState() });
}

[AllowAnonymous]
[HttpPost]
public ActionResult JsonRegister(RegisterModel model)
{
    if(ModelState.IsValid)
    {
        // 尝试注册用户
        MembershipCreateStatus createStatus;
        Membership.CreateUser(model.UserName, model.Password, model.Email, ↵
        passwordQuestion: null, passwordAnswer: null, isApproved: true, ↵
        providerUserKey: null, status: out createStatus);

        if(createStatus== MembershipCreateStatus.Success)
        {
            FormsAuthentication.SetAuthCookie(model.UserName, createPersistentCookie:↵
            false);
            return Json(new { success = true });
        }
        else
        {
            ModelState.AddModelError("", ErrorCodeToString(createStatus));
        }
    }

    // 如果出错
    return Json(new { errors = GetErrorsFromModelState() });
}
```

如果感觉这些方法似曾相识，那么就对了——基于AJAX的操作与基于HTTP GET的操作的主要差别就是返回的消息是JSON消息(通过JSONResult)而不是HTML应答消息（通过ViewResult）。

用户授权

表单验证里的用户授权机制与Windows验证里的用户授权机制一样——通过在控制器操作上标记AuthorizeAttribute来限制特定的验证用户访问：

```
[Authorize]
public ActionResult About()
{
        return View();
}
```

当没有验证的用户访问该操作时，ASP.NET MVC就会拒绝请求并跳转到登录界面。此时，原始请求会作为ReturnUrl参数传递给登录页面，例如，*/Account/LogOn?ReturnUrl= %2fProduct %2fCreateNew*。登录成功后就会重新跳转到原始的请求页面。

当使用表单验证时，某些页面，比如主页、联系我们等页面，所有人都可以访问。AllowAnonymous标记属性可以授权所有的匿名用户访问这些页面：

```
[AllowAnonymous]
public ActionResult Register()
{
        return ContextDependentView();
}
```

除了可以通过AuthorizeAttribute的User或Groups属性来声明授权用户以外，还可以通过调用User.Identity.Name()或者User.IsInRole()方法来检查用户是否有权访问指定的操作：

```
[HttpPost]
[Authorize]
public Action ResultDetails(intid)
{
    Model model= new Model();

    if(!model.IsOwner(User.Identity.Name))
       return View("InvalidOwner");
    return View();
}
```

192

防御攻击
Guarding Against Attacks

管理用户安全是保护网站的第一步。接下来，最重要的挑战是防御潜在的入侵者的攻击。

虽然大部分网站用户不会破坏网站，但是有些人还是擅长发现导致安全漏洞的Bug，还有些从使用简单的攻击方法到使用蠕虫病毒自动攻击某些常见的安全漏洞而寻找乐趣的人，入侵者各式各样，千差万别。

无论是哪种类型的入侵者，最重要的防御黑客攻击的武器就是好好使用本章里介绍的一些安全规则。当然，适当的监测和审计工作也是需要的，这样一旦发生攻击，运营和开发团队就可以分析其原因并防御以后的攻击。

下面几节将介绍最常见的Web应用攻击方式，以及保护网站安全所采取的措施。

SQL注入

SQL注入攻击是指攻击者通过参数注入来非法获取或者修改网站数据。为了演示这种情况，下面有个简单的例子。记住，真实的攻击者通常会创建复杂的算法来找出漏洞并攻击这些漏洞。

现在来看SQL注入的例子。这个例子使用的是EBuy交易网站的数据库，包含了Auctions表和Categories表。Auctions表和Categories表（见图9-9和图9-10）为多对多的关系，通过表CategoryAuctions关联（图见9-11）。

Column Name	Data Type	Allow Nulls
Id	uniqueidentifier	☐
Title	nvarchar(500)	☐
Description	nvarchar(MAX)	☐
StartTime	datetime	☐
EndTime	datetime	☐
CurrentPrice_Code	nvarchar(MAX)	☑
CurrentPrice_Value	float	☐
WinningBidId	uniqueidentifier	☑
IsFeaturedAuction	bit	☐
OwnerId	bigint	☐
[Key]	nvarchar(50)	☐
User_Id	bigint	☑
		☐

图9-9　EBuy交易网站的Auctions表

Column Name	Data Type	Allow Nulls
Id	bigint	☐
Name	nvarchar(100)	☐
ParentId	bigint	☑
[Key]	nvarchar(50)	☐
		☐

图9-10　EBuy交易网站的Categories表

Column Name	Data Type	Allow Nulls
Category_Id	bigint	☐
Auction_Id	uniqueidentifier	☐
		☐

图9-11　EBuy交易网站的CategoryAuctions表

下面是接受ID，从而用于查询Categories表的控制器代码：


```
public ActionResult Details(string id)
{
    var viewModel= new CategoriesViewModel();

    var sqlString= "SELECT * FROM Categories WHERE id = " + id;
    var connString= WebConfigurationManager.ConnectionStrings["Ebuy.DataAccess.↵
```

```
        DataContext"].ConnectionString;
        using(var conn = new SqlConnection(connString))
        {
            var command = new SqlCommand(sqlString, conn);
            command.Connection.Open();

            IDataReader reader = command.ExecuteReader();

            while(reader.Read())
            {
                viewModel.Categories.Add(new Category { Name = reader[1].ToString()});
            }
        }

        return View(viewModel);
    }
```

正常情况下，用户浏览~/category/details/1%20or%201=1视图，一切显示正常——控制器根据ID加载单个的category类别信息，如图9-12所示。

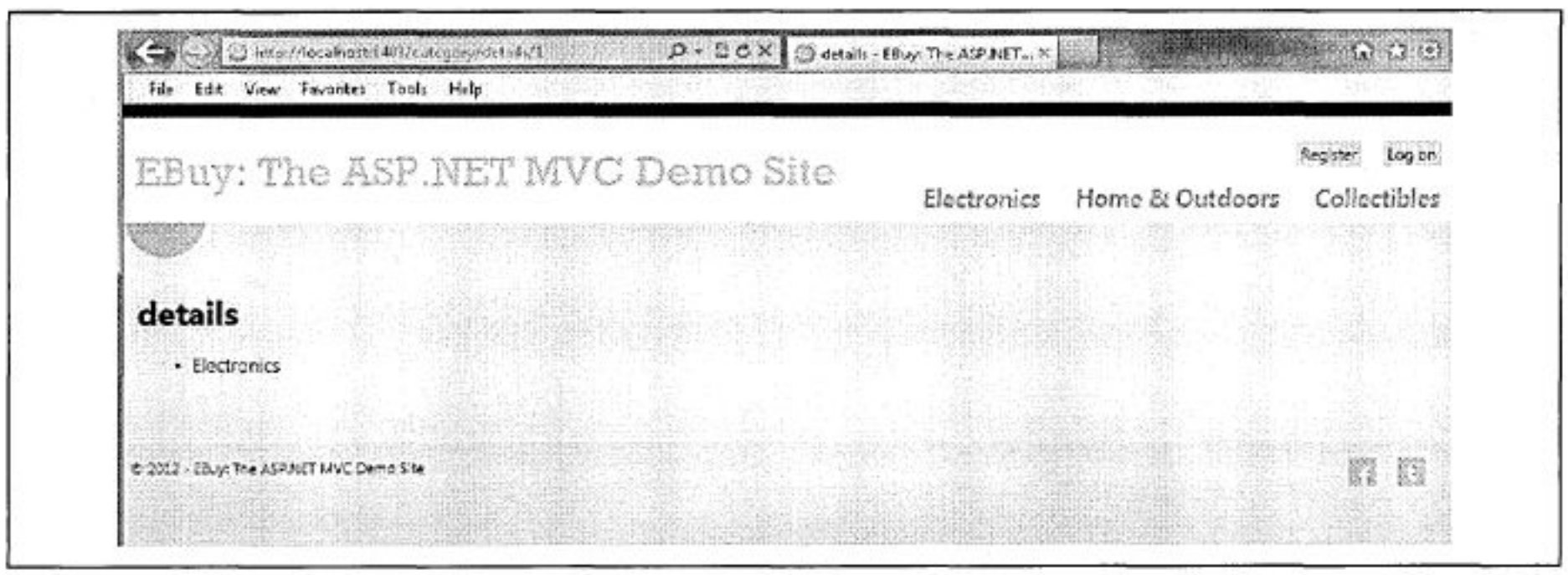


图9-12　SQL注入攻击视图

但是，若我们简单修改查询字符串为~/category/details/1 or 1=1，安全漏洞就出现了。现在不单单是返回一个数据，而是返回全部的类别信息，如图9-13所示。

195

一旦攻击者发现这个漏洞，他们就会尝试发现更多的漏洞，通过修改查询字符串来执行SQL语句。事实上，攻击者甚至不需要敲击键盘，整个过程可以自动执行。

下面的例子就不只是查询数据这么简单了，攻击者还可能修改甚至删除数据。

第一种，攻击者找出表中的某些字段。这非常有用，因为攻击者可以因此而发现外键，从而查询出其他表中的信息。这个例子中，攻击者提交了一个包含CategoryName的地址~/category/details/1或categoryname=\"。因为Categories不包含这个列，所以数据库会抛出异常，如图9-14所示。

默认情况下，显示全部的错误信息对任何ASP.NET程序都是最坏的情况，这样会暴露调用堆栈和其他敏感信息给攻击者，让他们可以定位和查找其他的安全漏洞。

196

第二种，攻击者发送查询命令。第一个查询用于确定表是否存在。如果攻击者知道表存在，他就可以继续插入、更新或者删除SQL数据库中的数据。

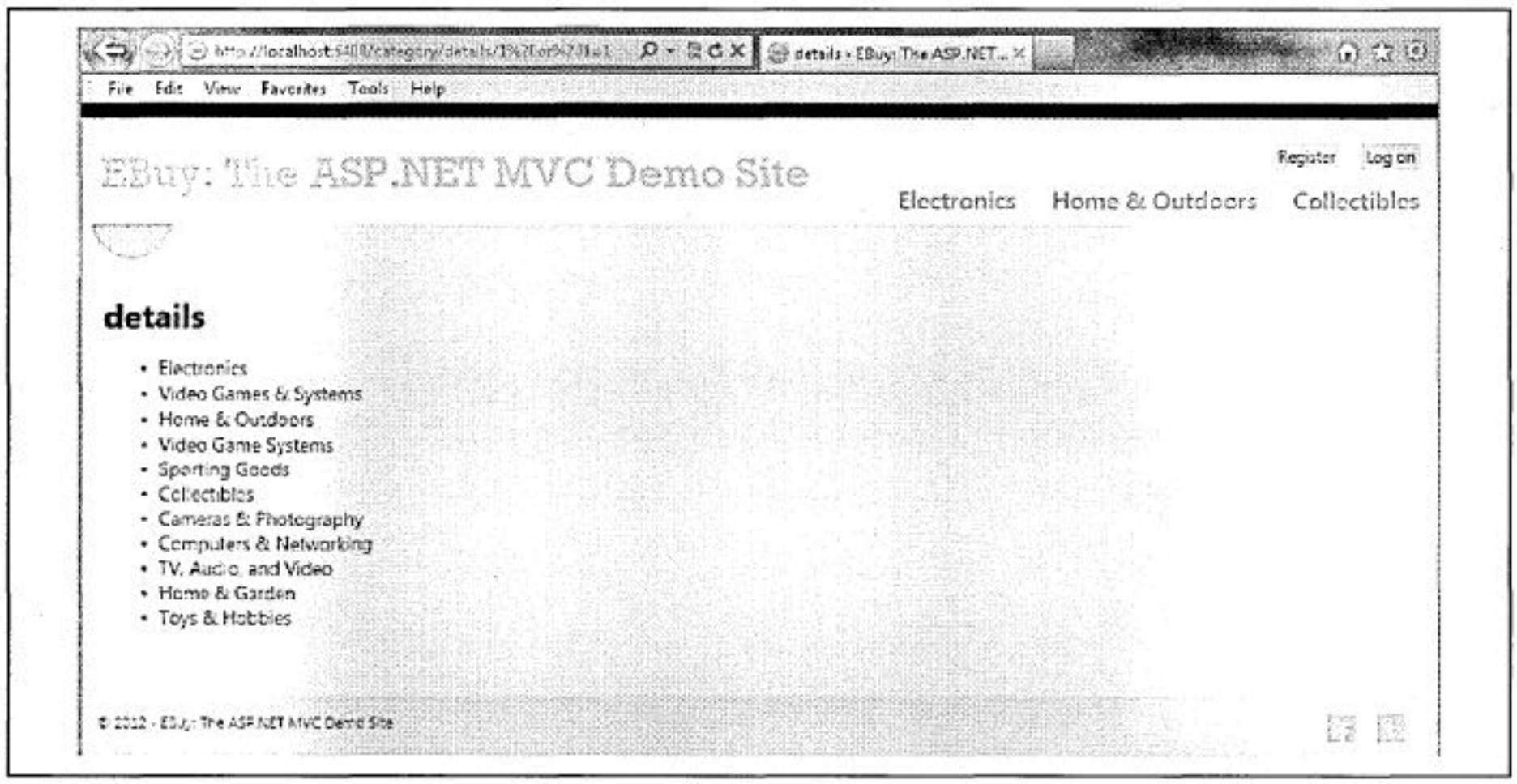


图9-13　SQL注入攻击漏洞

```
1 OR 1=(SELECT COUNT(1) FROM Auctions)

1; INSERT INTO Auctions VALUES (1, "Test", "Description")
```

幸运的是，有多种方法可以防御SQL注入攻击。最好的选择就是默认所有的输入数据都是非法的，包括查询字符串query strings、HTML表单、请求消息头以及其他的输入信息。

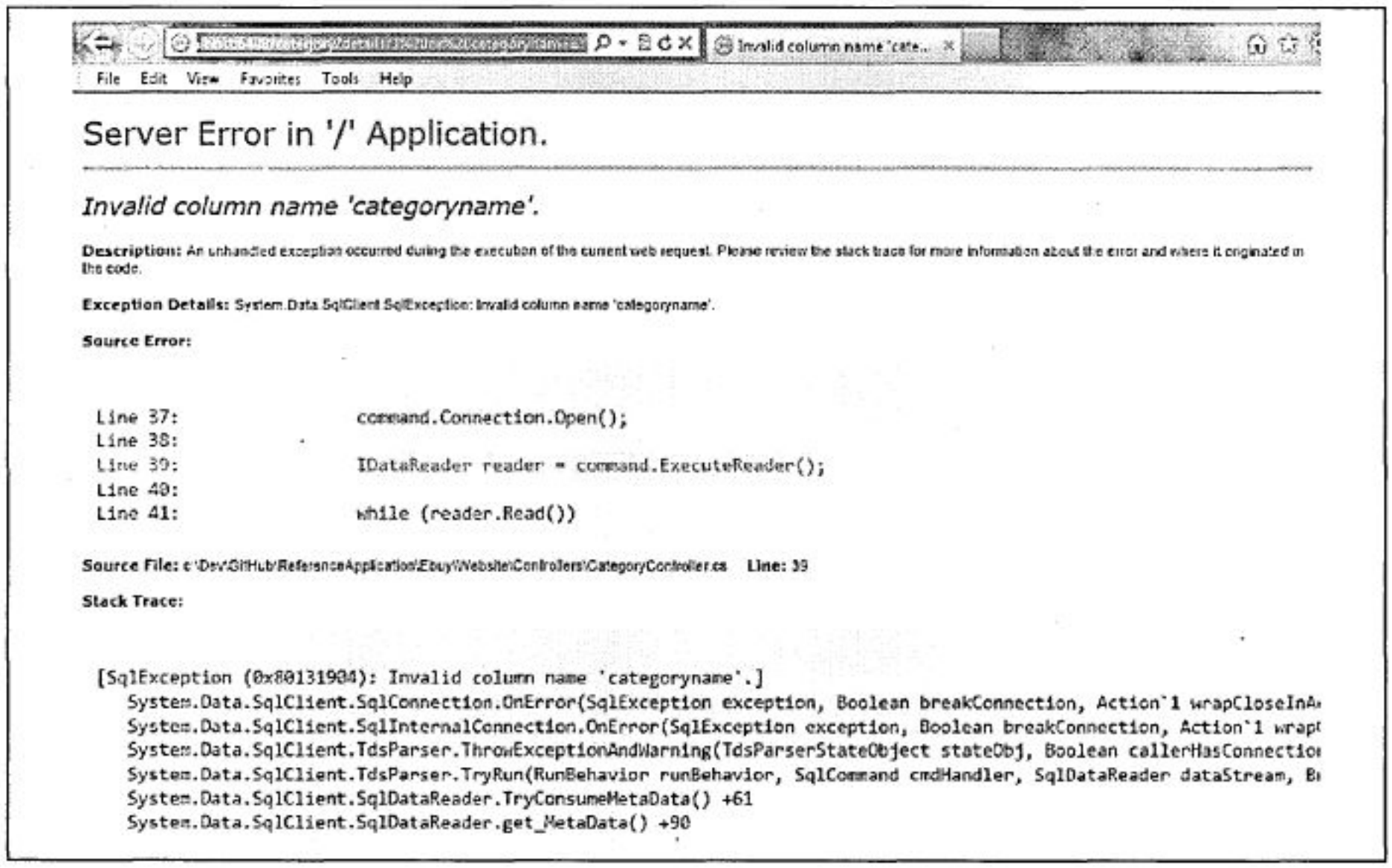


图9-14　SQL注入攻击不存在的字段

全局来看，大概有两种方法可以用来校验输入数据。

黑名单方法

黑名单方法需要依赖一个已知的列表来验证输入数据。黑名单看起来不错，因为可以动态防御已知的威胁，但是其有效性和我们提供的数据有关，通常只要修改攻击数据就可以轻易逃过审查。

197

白名单方法

白名单方法与黑名单方法恰恰相反，只有请求消息包含在预定义的白名单中才算是合法的输入。换句话说，白名单阻止所有其他不允许的输入数据。这种方法非常安全，它只允许特定的数据值输入系统。

白名单方法和黑名单方法都需要细心维护允许和不允许输入的值，黑名单方法无法阻止所有的潜在恶意输入，而白名单方法可能阻止合法但是没有允许的数据输入。

现在来了解白名单方法。下面的代码用于校验输入数字参数的有效性：

```
var message = "";
var positiveIntRegex= new Regex(@"^0*[1-9][0-9]*$");
if(!positiveIntRegex.IsMatch(id))
{
   message= "An invalid Category ID has been specified.";
}
```

虽然有些对象关系映射器框架如LINQ to SQL和EntityFramework会自动处理SQL注入问题，但是它们并不能防御所有的注入问题。

以下是微软关于使用Entity Framework处理SQL注入攻击的指导原则。

Entity SQL注入攻击

攻击者可以通过输入有害的查询参数在Entity SQL里实现SQL注入攻击。为了避免SQL注入攻击，不让用户输入和Entity SQL命令绑定在一起。

Entity SQL查询接受任意字符串输入参数。我们应该使用参数化的查询方式代替这种外部注入字符串的方式查询。当然也可以考虑使用查询构建器来生成安全的Entity SQL语句。

LINQ to Entities注入攻击

虽然也可以在LINQ to Entities中实现查询组合，但是它是通过对象模型的API执行的。与EntitySQL查询不同，LINQ to Entities查询不是使用字符串拼接的方式，它不适合用于传统的SQL注入攻击。

198

跨站脚本

与SQL注入攻击一样，跨站脚本（XSS）攻击也对接受用户输入的网站的安全构成严重威胁。

跨站脚本（XSS）攻击的根源是对用户输入验证不足。XSS攻击通常是攻击者模拟伪装成用户访问的网站或者在邮件里嵌入非法网址的连接。

攻击者经常盗取包含用户登录凭据和会话ID的cookie，因为他们可以通过这些cookie来访问用户信息或者进行其他的有害操作，比如提交额外的HTML内容或者有害的JavaScript代码。

幸运的是，微软已意识到XSS的威胁并且在ASP.NET 框架里提供了基本的保护措施来校验用户请求。当ASP.NET接收到一个新请求时，它就会查询HTML标签或者脚本（比如表单字段、消息头或者cookie等）。如果发现可疑内容，ASP.NET就会通过抛出异常来拒绝请求。此外，ASP.NET 4.5也集成了最新的Microsoft XSS库。

某些情况下，比如内容管理系统（CMS）、论坛和博客都需要支持HTML内容。考虑到这种情况，ASP.NET 4.5引入了“延迟请求验证机制”以及访问无效请求数据的方法。使用这些方法时要格外注意：需要进行验证工作。

为了配置ASP.NET使用延迟请求验证机制，只要直接在“web.config”里更新`httpRuntime>requestValidationMode`为4.5即可：

```
<httpRuntime requestValidationMode="4.5" />
```

当启用延迟请求验证机制时，验证过程会在应用程序第一次调用请求集合的时候触发（比如`Request.Form["post_content"]`），而不是请求进入网站请求管道时触发。为了跳过输入验证，可以使用`HttpRequest.Unvalidated()`辅助方法来访问未验证的集合：

```
using System.Web.Helpers;

var data = HttpContext.Request.Unvalidated().Form["post_content"];
```

微软已经在ASP.NET 4.5集成了最流行的Microsoft Anti-XSS Library（微软防御XSS攻击库）。其中`AntiXSSEncoded`类的一个方法就是在`System.Web.Security.AntiXss`命名空间下进行编码。我们可以直接调用`AntiXSSEncoded`类的静态编码方法。

使用防御XSS攻击功能最简单的方法就是在ASP.NET Web网站中使用这个类，在web.config中把`encoderType`设置成`AntiXssEncoded`。一旦设置完成，整个网站的输出编码都会使用XSS编码功能。

```
<httpRuntime...
  encoderType="System.Web.Security.AntiXss.AntiXssEncoder,System.Web, Version=4.0.0.0, ↵
Culture=neutral, PublicKeyToken=b03f5f7f11d50a3a" />
```

ASP.NET 4.5包含的防御XSS攻击的类库如下。

- `HtmlEncode`、`HtmlFormUrlEncode`和`HtmlAttributeEncode`；
- `XmlAttributeEncode`和`XmlEncode`；
- `UrlEncode`和`UrlPathEncode`(新增加到ASP.NET 4.5!)；
- `CssEncode`。

跨站请求伪造

无论你怎样保护网站，Web并不十分安全，总会有人试图绕过审查攻击网站。我们已经详细介绍了XSS和SQL注入攻击，现在来看看另外一个潜在的、更严重的安全问题——大部分网站开发人员都会忽视的问题：跨站请求伪造(CSRF)。

CSRF可以利用Web的工作原理进行攻击。下面是可以接受CSRF攻击的控制器的例子代码。代码看起来简单明了，但是这个控制器是跨站请求伪造(CSRF)攻击的首选目标：

```
public class ProductController: Controller
{
    public ViewResult Details()
    {
```

```
        return View();
    }

    public ViewResult Update()
    {
        Product product= DbContext.GetProduct();

        product.ProductId= Request.Form["ProductId"];
        product.Name= Request.Form["Name"];
        SaveUProduct(product);

        return View();
    }
}
```

为了攻击这个控制器，所有的攻击者都需要建立一个攻击页面。一旦攻击者说服了某个用户访问该页面，这个页面就会提交请求给控制器：

```
<body onload="document.getElementById('productForm').submit()">
    <form id=" productForm'" action="http://.../Product/Update" method="post">
        <input name="ProductId" value="123456" />
        <input name="Name" value="My Awesome Hack" />
    </form>
</body>
```

如果用户已经通过了Windows验证或者表单验证，控制器显然就会成为CSRF攻击的目标。什么才是对付这种潜在安全威胁的有效措施呢？

有两种防御CSRF攻击的措施。

域参考

检查进入的请求消息是否包含域头信息。这可以阻止来自第三方提交的非法请求，但也有些缺点：用户可能因为私人原因禁止了域头信息，并且假如用户安装了旧版本的Adobe Flash，攻击者还可以伪造域头信息。

用户生成令牌

使用HTML隐藏域来存储用户令牌（例如，服务器生成令牌信息），检验提交的令牌是否有效。生成的令牌可以存储在用户会话Session里或者HTTP cookie里。

使用ASP.NET MVC来防御CSRF

ASP.NET MVC包含了一些抵制CSRF攻击的方法，可以通过在视图和控制器之间创建用户令牌来验证每个请求。这需要使用`@Html.AntiForgeryToken()` HTTP帮助方法在页面上增加隐藏域，控制器会验证每个请求。为了增加安全性，HTML辅助方法同样可以接受一个Salt Key（辅助参数）参数，以增加令牌的随机性：

```
@Html.AntiForgeryToken()
@Html.AntiForgeryToken("somerandomsalt")
```

为了使这些代码可以正常工作，控制器操作还需要能够处理包含令牌的表单信息。同时，还需要使用`ValidateAntiForgeryTokenAttribute`标记属性。这个标记属性可以检查进入请求消息的cookie或者检查表单字段是否包含名为`RequestVerificationToken`的值，或者两者同

时存在：

```
[ValidateAntiForgeryToken]
public ViewResult Update()
{
}
```

ASP.NET MVC包含的防御CSRF的辅助方法非常有用，但是使用这些辅助方法也有一些限制，具体如下。

- 合法的用户必须接受cookie。如果用户浏览器禁用了cookie，`ValidateAntiForgeryTokenAttribute`过滤器就会拒绝用户请求。
- 这个方法只对`POST`请求有效，对`GET`请求无效。事实上，这也不是大问题，因为我们只需要使用`GET`请求来获取只读操作数据。
- 如果网站包含任何XSS跨站脚本攻击漏洞，攻击者就可以轻易获取到反CSRF令牌。
- 与jQuery一样，对AJAX请求，ASP.NET MVC框架也不会自动传递需要的cookie和HTML隐藏域。我们需要自己实现传输和读取防御CSRF令牌的方案。

总结

Summary

本章讲述了如何构建安全的ASP.NET MVC Web应用程序，并深入比较了Windows验证和表单验证的区别，随后介绍了如何使用`AuthorizeAttribute`标记属性来授权不同的用户和组，最后介绍了如何防御SQL注入、跨站脚本攻击XSS以及如何使用跨站请求伪造防御辅助方法。

第10章 移动Web网站开发

Mobile Web Development

随着移动设备的爆炸式增长，移动Web也显得越来越重要，它可以提供强大的媒介功能，可以把内容展示给更多的用户。因此，将移动设备市场纳入我们的项目初期规划变得十分重要。

移动Web网站开发最痛苦的事情就是移动设备的多样性。由于不同的设备拥有不同的硬件能力、分辨率、浏览器、特性支持、触摸功能，等，因此让网站兼容所有的设备并带来一致性的用户体验是一项非常艰巨的任务。

本章会详细介绍如何使用ASP.NET MVC框架的移动Web网站开发功能——ASP.NET MVC 4新增的功能，尽可能开发出一个兼容多种设备并且提供一致性的用户体验的强大移动Web网站。

ASP.NET MVC 4移动特性

ASP.NET MVC 4 Mobile Features

从第3版开始，ASP.NET MVC框架就提供了一些支持移动网站开发的新特性。这些功能在ASP.NET MVC 4中得到了进一步增强。

下面列举了ASP.NET MVC 4移动开发的新特性。本章的后续部分就会结合网站例子来详细介绍这些新特性。

ASP.NET MVC 4移动Web模板

如果想从头开始创建一个新的移动Web网站程序，那么ASP.NET MVC 4提供了移动应用模板，可以帮助我们加速开发。跟ASP.NET MVC 的Web应用模板一样，ASP.NET MVC 4会自动创建移动网站视图，设置jQuery Mobile MVC NuGet包，并且创建网站的整个骨架。本章的“ASP.NET MVC 4移动模板”一节详细、深入介绍了ASP.NET MVC 4移动应用模板内容。

显示模式

为了更好地兼容不同设备，ASP.NET MVC 4框架提供了显示模式来探测并适应不同的设备。

不同的设备有不同的分辨率、不同的浏览器行为，甚至有不同的功能，所以可以根据设备的不同行为和功能提供不同的设备视图。

假如有个名称为Index.cshtml的普通视图，现在需要为智能手机和平板电脑创建一个移动Web版本的视图，那么使用显示模式时，就可以创建Index.iPhone.cshtml和Index.iPad.cshtml两个不同设备的视图文件，然后使用ASP.NET MVC 4 Framework's DisplayModeProvider在程序启动时进行注册。根据过滤条件，ASP.NET MVC框架能够自动查找带有不同后缀的视图文件（“iPhone”或“iPad”），然后渲染这些视图。（注意ASP.NET MVC遵守“[View Name].[Device].[Extension]”这种命名习惯。）在本章后面“特定浏览器视图”一节里，可以看到如何使用不同的特性来兼容不同的设备。

适应移动视图而重写普通视图

ASP.NET MVC 4引入了简单的机制允许我们重写普通视图（包括布局和部分视图）以适应不同的浏览器。要提供移动设备浏览器专用视图，只需要创建带有.Mobile的视图文件即可。例如，为了创建Index视图，可以拷贝Views\Home\Index.cshtml，并且重新命名为Views\Home\Index.Mobile.cshtml，ASP.NET会自动在浏览器里渲染这个视图来代替普通视图Views\Home\Index.cshtmlto Views\Home\Index.Mobile.cshtml。非常有意思的是，虽然显示模式允许我们指定特定的移动浏览器，但是这个特性也提供更加通用的使用方式。如果视图可以通用于不同的浏览器或者使用了jQuery Mobile这种兼容不同移动平台的开发框架，这个特性就显得非常有用。

jQuery Mobile

jQuery Mobile框架给移动应用带来了所有的jQuery和jQuery UI丰富且强大的功能。只需要创建一个移动Web网站，就可以兼容所有的设备，而不需要担心不同的浏览器的兼容性问题。它带来的逐步增强技术的所有优点、灵活的设计，使较旧的设备仍然可以支持应用程序的基本功能（但不一定漂亮或功能丰富），同时还支持新设备使用HTML 5等新特性来访问网站。jQuery Mobile框架同样支持强大的主题，我们可以利用这个框架创建支持良好用户体验的网站而又不需要牺牲任何逐步改进的新功能。本章会介绍如何使用jQuery Mobile框架构建强大的移动Web网站。

让移动应用变得更友善

Making Your Application Mobile Friendly

“移动Web应用”的概念十分广泛，包含了许多网站开发人员需要考虑的问题。也许最重要的问题就是如何更好地提供信息以及如何与用户交互。

想象传统的Web网站体验：浏览器通常都有一个大屏幕，网站的访问速度都非常快而且稳定，用户可以由键盘和鼠标与网站进行交互。恰恰相反，移动Web体验受限于小屏幕，Web访问是断断续续的，并且只能通过触笔或手指输入数据。

这些限制决定了我们必须选择性地显示内容以及支持轻量级的功能集合，这与强大的用传统PC浏览器开发的网站形成了鲜明对比。然而，移动Web也提供了传统Web所不具备的机会，比如基于位置的数据（location-specific data）、便携式的通信以及视频和语音通话功能。

了解目标用户的需求是确定移动策略的第一步，如以下常见的移动设备用途：

- 逛街时偶尔看一下电子邮件Email（当然，走路时不要撞到电线杆子上！）。
- 乘坐地铁或者火车时看一下热点新闻。
- 一手拿着咖啡，一手拿着手机看银行卡里的余额。

上面所有的场景有一个共同之处就是用户的关注点是分开的——大家都想尽快完成一项任务，以便继续其他的工作。

这就意味着需要移动Web网站能够给用户提供一种快速、高效且易于理解的信息展示方式。

创建Auction移动视图

当开发移动Web网站时，可以通过给已有网站添加移动专用视图或者完全重新创建一个新的移动Web网站来实现。许多因素会影响我们的选择，但是对于开发者来说,两者各有优点和缺点。记住，ASP.NET MVC 4框架为两种方式都提供了开发工具，本章后续部分会做出相应的介绍。本节先给已有网站添加移动专用视图，然后再使用ASP.NET MVC 4框架提供的新特性来增强移动视图体验。

开始要先拷贝一份Auctions.cshtml视图重新命名为Auctions.Mobile.cshtml文件，表明这是一个移动浏览器专用视图。

为了区别是否是移动Web网站视图文件，现在把视图文件里的<H1>标签内容修改为“Mobile Auctions”（移动版交易页面）。

运行网站后，从手机浏览器里导航到Auctions页面，结果如图10-1所示。

从图示中可以看到，页面头部显示了“Mobile Auctions”，这就可以确认视图是移动Web视图(如果使用普通的浏览器导航到Auctions页面，则显示“Auctions”）。MVC 4框架显示模式的特性就是可以自动探测客户端浏览器并加载正确的视图。

移动设备请求网站时，ASP.NET MVC不仅会自动加载“mobile”视图，而且它还要加载布局和部分视图。另外一个实事就是，jQuery.Mobile.MVC包会为移动设备创建基于jQuery Mobile框架的优化的布局。

图10-1 ASP.NET MVC框架可以自动探测并显示浏览器专用视图

使用jQuery Mobile框架

jQuery Mobile可以用来创建更加原生的移动视图，使用这个框架，不仅可以定制网站主题（theme），而且可以确保旧版本浏览器获取一个基本正常（样式可能不是最好）、不缺少功能并且可以正常使用的页面。

要使用jQuery Mobile，就需要安装jQuery.Mobile.MVC包。可以使用NuGet包管理器来进行安装，如图10-2所示。

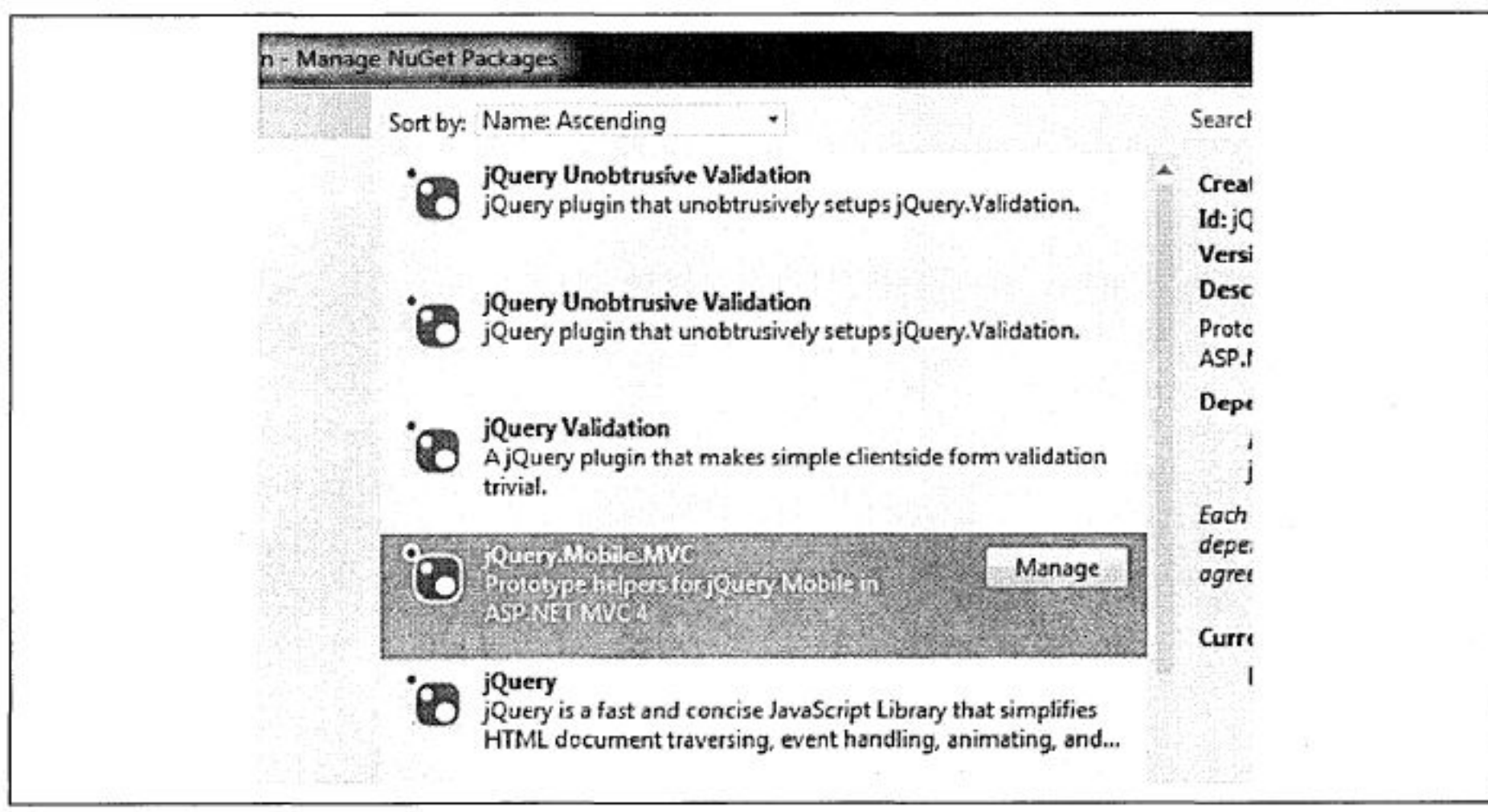


图10-2　通过NuGet安装jQuery Mobile框架

这个包会添加下面的文件：

Mobile框架

它是JavaScript（jQuery.mobile-1.1.0.js）文件和CSS（jQuery.mobile-1.1.0.css）文件的集合，是压缩版和图片文件。

/Content/Site.Mobile.css

一个新的移动设备专用的stylesheet。

Views/Shared/_Layout.Mobile.cshtml

为优化移动设备的布局，引用了jQuery Mobile框架文件（JS和CSS）。ASP.NET MVC会自动为移动设备加载这些文件。

The view-switcher component

view-switcher组件由Views/Shared/_ViewSwitcher.cshtml部分视图文件和View-Switcher-Controller.cs控制器组成。这个组件包含一个可以在移动视图和普通视图之间切换的连接。本章后面的“在移动视图和普通视图之间切换”一节将详细介绍。

jQuery Mobile框架一直保持不断地更新，所以可能会看到更新的版本，这很正常。

为了让jQuery Mobile框架使用页面样式，首先需要打开Views/Shared/_Layout. Mobile.cshtml视图，并修改部分代码：

```
<body>
  <div data-role="page" data-theme="b">
    <header data-role="header">
      <h1>@Html.ActionLink("EBuy: The ASP.NET MVC Demo Site", "Index", "Home")</h1>
    </header>
    <div id="body" data-role="content">
      @RenderBody()
    </div>
  </div>
</body>
```

然后通过修改Auctions.Mobile.cshtml来优化移动布局：

```
@model IEnumerable<AuctionViewModel>
<link href="@Url.Content("~/Content/product.css")" rel="stylesheet" type="text/css" />
@{
  ViewBag.Title = "Auctions";
}

<header>
  <h3>Mobile Auctions</h3>
</header>

<ul id="auctions">
  @foreach (var auction in Model)
  {
    <li>
      @Html.Partial("_AuctionTile",
auction);
    </li>
  }
</ul>
```

修改完成以后进行编译，然后运行网站，最后在移动浏览器里浏览到网站主页，就会看到图10-3所示的网页。

从中可看到，Auctions视图为了适应移动浏览器而改变了。虽然界面看起来不够完美，但jQuery.Mobile.MVC包还是提供了快速构建移动视图的基本功能。

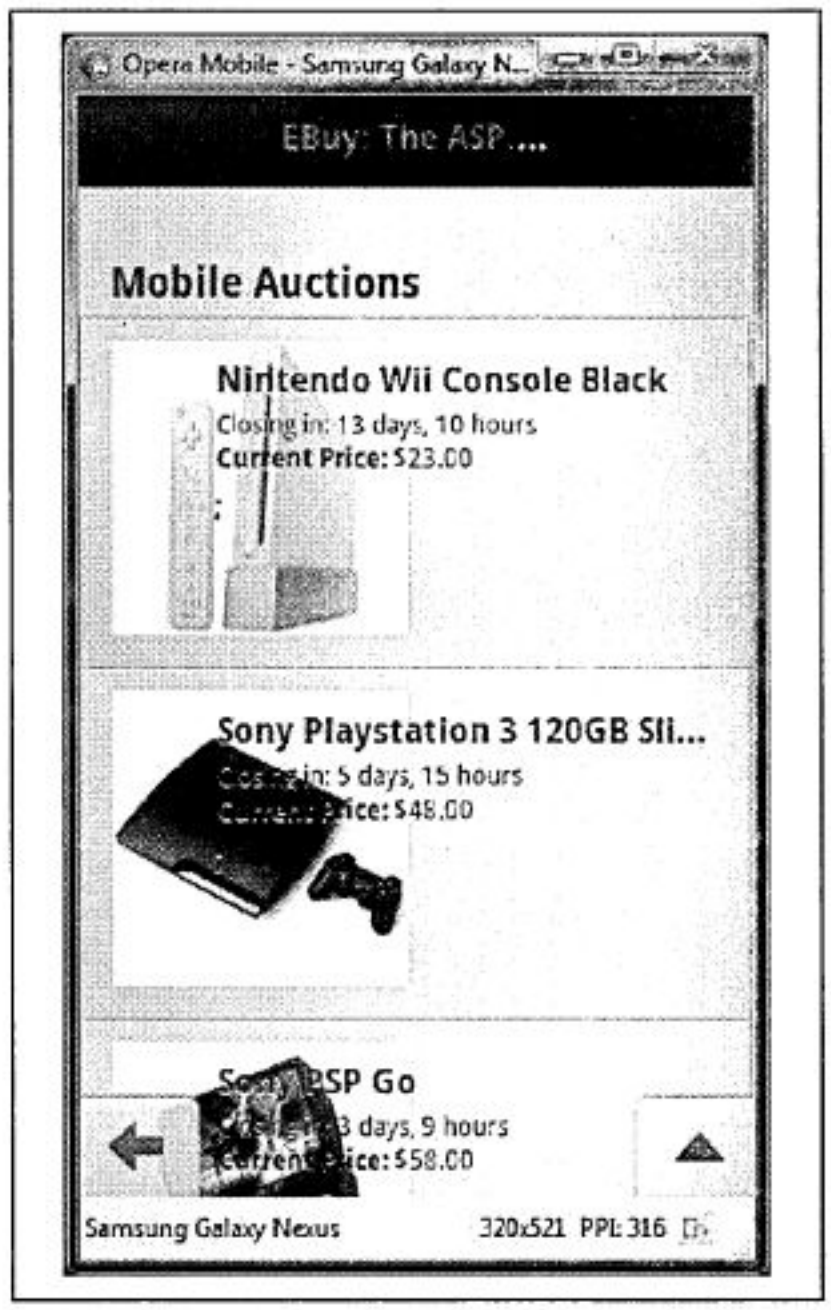


图10-3 优化移动布局后的EBuy电子商务网站

增强视图

虽然jQuery.Mobile.MVC包做了很多底层工作，但是UI看起来还是不够自然。其实jQuery Mobile提供了许多组件和样式，让Web网站看起来更加真实。

210

使用jQuery Mobile的“listview”列表控件来改进交易信息auctions列表

现在通过使用jQuery Mobile的“listview”列表控件来完善交易信息auctions列表。jQuery Mobile需要根据data-role属性来执行大部分转换工作。为了把auctions交易信息对象转换成listview列表，需要给Auction的<ul>标签添加data-role="listview"属性：

```
<ul id="auctions" data-role="listview">
  @foreach (var auction in Model.Auctions)
  {
    <li>
      @Html.Partial("_AuctionTileMobile", auction);
    </li>
  }
</ul>
```

修改_AuctionTileMobile部分视图，代码如下：

```
@model AuctionViewModel
@{
  var auctionUrl = Url.Auction(Model);
}

<a href="@auctionUrl">
  @Html.Thumbnail(Model.Image, Model.Title)
  <h3>@Model.Title</h3>
  <p>
    span>Closing in: </span>
    <span class="time-remaining" title="@Model.EndTimeDisplay"> ↲
    @Model.RemainingTimeDisplay</span>
  </p>
  <p>
    <strong>Current Price: </strong>
    <span class="current-bid-amount">
@Model.CurrentPrice</span>
    <span class="current-bidder">@Model.
WinningBidUsername</span>
  </p>
</a>
```

现在重新在移动浏览器里打开Auctions视图，就会看到一个更漂亮的页面，如图10-4所示。

图10-4　使用jQuery Mobile框架来渲染<ul>列表

假设<ul>已经包含了交易信息的列表元素，那么就会发现添加“listview”role可能是多余的。<ul>将会显示成列表，但是对小屏幕的移动设备来说无法显示超链接。data-role="listview"的真正目的其实是方便在列表项目后面追加一个超链接图标。

211

使用jQuery Mobile的“data-filter”实现auction交易搜索

接下来为页面添加一项搜索功能，允许用户筛选交易信息。用jQuery Mobile可以很方便地实现这项功能——只需要给auctions <ul>标签添加data-filter="true"属性即可：

```
<ul id="auctions" data-role="listview" data-filter="true">
  @foreach (var auction in Model.Auctions)
  {
    <li class="listitem">
      @Html.Partial("_AuctionTileMobile", auction);
    </li>
  }
</ul>
```

刷新移动浏览器就可以看到顶部的搜索框了，如图10-5所示。

在搜索框里输入关键字，jQuery Mobile就会自动过滤搜索结果，如图10-6所示。

图10-5　使用jQuery Mobile实现auctions搜索功能

图10-6　jQuery Mobile根据关键字自动过滤数据列表

我们已经看到，jQuery Mobile框架可以用来修改任何页面，让其自适应移动平台，使其看起来更加自然。除了这些功能，jQuery Mobile还包含一些可以让用户随意访问其他视图的组件。这些特性列表可以在jQuery Mobile的API官方文档里看到。

在普通视图和移动视图之间切换

如果要给移动浏览器提供专门的网站，最好的办法是直接让移动用户访问移动版网站。当然也可以提供让用户在普通网站和移动版网站之间自由切换的功能。

注意，ASP.NET MVC移动网站模板的顶部带有一个可以切换到普通视图的连接按钮。这项功能也称为视图切换器（ViewSwitcher）部件，已作为jQuery.Mobile. MVCNuGet包的一部分安装了。

为了了解这个部件如何工作，我们来深入了解这个组件。先看一个新的部分视图_ViewSwitcher.cshtml，代码如下：

```
@if (Request.Browser.IsMobileDevice&&Request.HttpMethod == "GET")
{
  <div class="view-switcher ui-bar-a">
    @if (ViewContext.HttpContext.GetOverriddenBrowser().IsMobileDevice)
    {
      @: Displaying mobile view
      @Html.ActionLink("Desktop view", "SwitchView", "ViewSwitcher",
      new { mobile = false, returnUrl = Request.Url.PathAndQuery },
      new { rel = "external" })
    }
    else
    {
      @: Displaying desktop view
      @Html.ActionLink("Mobile view", "SwitchView", "ViewSwitcher",
      new { mobile = true, returnUrl = Request.Url.PathAndQuery },
      new { rel = "external" })
    }
  </div>
}
```


GetOverriddenBrowser()方法返回重载浏览器的功能HttpBrowserCapabilities对象列表，或者不重载时，可以判断这个请求是否来自移动设备。随后小部件会做出判断，决定是渲染普通视图还是移动视图，并且会包含一个视图切换的连接。

另外，可以通过在RouteValue字典中设置mobile属性来指定当前显示的视图是普通视图还是移动视图。

接下来看ViewSwitcherController类，它包含切换视图功能的逻辑代码：

```
public class ViewSwitcherController: Controller
{
  public RedirectResult SwitchView(boolmobile, string returnUrl) {
    if(Request.Browser.IsMobileDevice== mobile)
      HttpContext.ClearOverriddenBrowser();
    else
      HttpContext.SetOverriddenBrowser(mobile ? BrowserOverride.Mobile
                                              : BrowserOverride.Desktop);
    return Redirect(returnUrl);
  }
}
```

根据请求是否来自移动设备（Request.Browser.IsMobileDevice指示），控制器使用ClearOverriddenBrowser()和SetOverriddenBrowser()方法来告诉ASP.NET MVC框架如何处理请求。如果是移动浏览器，就会显示移动版本网站；如果是普通的浏览器，就会显示正常的网站。

Layout.mobile.cshtml文件里的<body>标签内添加如下代码，在页面底部渲染ViewSwitcher部分视图，如图10-7所示。

```
<div data-role="footer">
  @Html.Partial("_ViewSwitcher")
</div>
```

图10-7　页面底部的视图切换连接

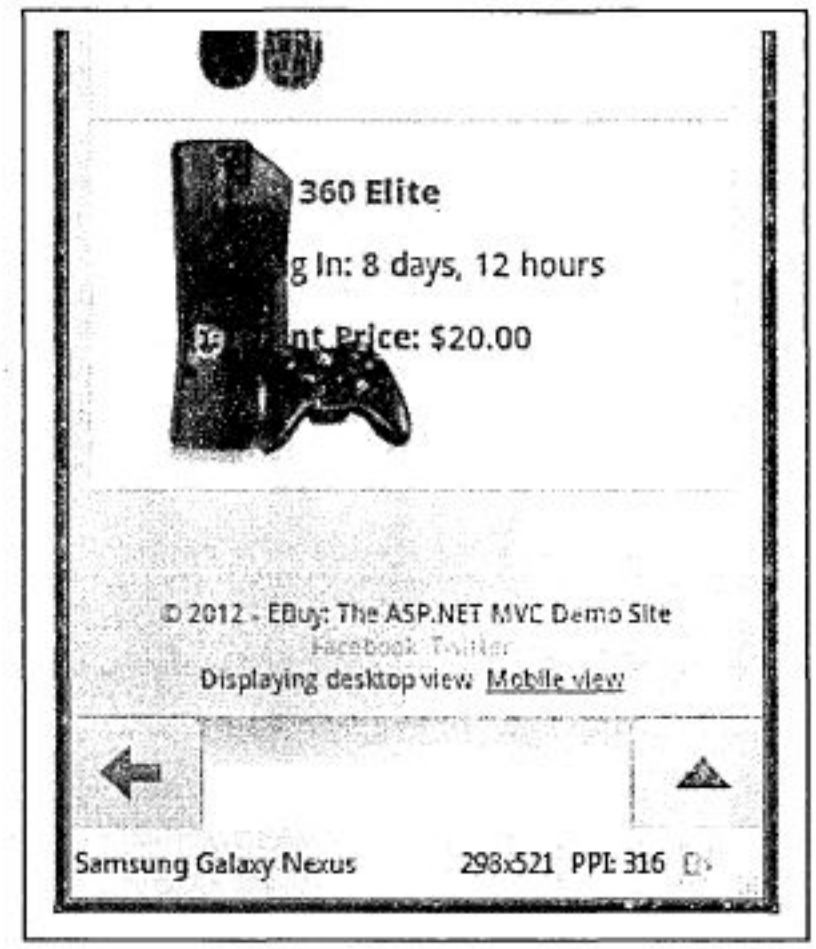


图10-8　普通桌面视图中的视图切换器

如果点击图10-7所示中的“桌面视图（Desktop view）”，就可以看到普通的Auctions交易信息视图。注意，桌面视图并不包含切换到移动视图的连接。要修改这个Bug，就要打开_Layout.cshtml全局共享视图，然后添加一行代码：

```
@Html.Partial("_ViewSwitcher")
```

运行网站，使用移动浏览器查看任意页面就可以在页面底部看到移动视图和普通桌面视图的连接，如图10-8所示。

禁止移动网站显示桌面视图

当缺少移动视图时，我们会看到ASP.NET MVC是如何在移动布局里渲染桌面视图的。

遵守某些标准、规范确实能帮助我们正常显示视图，但是有时候我们想关闭这项功能。

为了让ASP.NET MVC不在移动布局里显示普通视图，可以通过设置RequireConsistent-DisplayMode=true来关闭这种功能：

```
@{
  Layout = "~/Views/Shared/_Layout.cshtml";
  DisplayModeProvider.Instance.RequireConsistentDisplayMode= true;
}
```

这样就可以避免在移动网站布局里显示任何默认的非移动视图了。

我们还可以通过设置全局方式，在“/Views/_ViewStart.cshtml”文件里把RequireConsistent DisplayMode设置为true来关闭所有的视图。

改进移动用户体验

Improving Mobile Experience

移动浏览器可以在一定程度上显示HTML页面，但是，依靠浏览器来提供最好的用户体验显然不够。假设浏览器只能支持普通级别的放大缩小页面和图片的操作，那么作为内容编辑者，他们能决定的就是相关的页面元素，以及在移动平台小屏幕上哪些该高亮显示哪些该放弃掉。因此，我们的工作就是如何让网站更漂亮，以及保证跨浏览器网站功能正常。

幸运的是，可以使用类似自适应渲染和持续增强类型的技术来改进网站显示效果，ASP.NET MVC 4和jQuery Mobile可以帮助我们很方便地解决这些问题。本章后面会继续介绍这些内容。

自适应渲染

Adaptive Rendering

自适应渲染是指视图自动适应浏览器渲染的能力。例如，假如页面上有很多Tab选项卡，每点击一个选项卡，AJAX会调用服务获取显示内容。如果禁用JavaScript，选项卡就无法显示内容。但是，使用自适应渲染，选项卡会根据情况调整自动渲染出的内容，用户仍然可以访问网页。

另外一个例子是水平显示超链接的导航栏。它在传统桌面浏览器里显示正常，但是在小屏幕上会出现积压重叠的情况。如果使用自适应渲染，导航栏就可以采用下拉方式来显示自适应小屏幕了。

使用自适应渲染技术的好处就是可以让网站在其他移动浏览器里正常工作。虽然网站在不同的设备上看起来会千差万别，但是仍然可以使用。

记住，无论网站使用什么技术，如果用户是回头客，多次访问我们的网站，就需要确保良好的用户体验。当然，ASP.NET MVC 4包含了支持自适应技术的jQuery Mobile框架，我们可以直接使用。

Viewport标签

在计算机图形学中，视区（viewport）是指矩形的可视区域。对浏览器来说，就是用来显示HTML文档的浏览器窗口。换句话说，这是所有<html>标签的构造容器，是所有标签元素的根元素。

当放大或者缩小浏览器窗口时会发生什么？改变设备方向时会怎么样呢？会改变视区吗？

在移动设备中，答案稍微有点巧妙，因为在现实中，存在着不是一个而是两个视区——“布局”视区和“视觉”视区。

“布局”视区不会改变，它是<html>页面的外围约束容器。当缩放和转向时改变的是“视觉”视区，而且可视效果是通过设备的边框界定的。

我们需要把视区当做提供功能和用户体验的手段。当移动设备渲染网页时，重要的是要做到网页不太宽也不太窄，最好能够完美适应屏幕尺寸。当适应屏幕以后，网页不应该出现页面的微缩版本，而应该是适合比例的实际页面。

在新版浏览器中，可以通过meta viewport标签而不是CSS来控制“视觉”视区（visual viewport）的大小。

可以通过设置meta viewport标签来控制视区大小：

```
<meta name="viewport" content="width=device-width" />
```

width=device-width的值是用来设置视区宽度的，以适应不同设备的实际屏幕宽度。这个值是最灵活也是最常用的。

也可以通过设置content属性来设置内容的固定宽度，代码如下：

```
<meta name="viewport" content="width=320px" />
```

现在，无论移动设备的屏幕有多宽，我们页面内容显示的宽度都是320像素，也就是说，在更大的屏幕上用户可能想放大网页内容，而在更小的屏幕上用户可能需要缩小页面内容。

<meta name="viewport">标签是一个行业标准规范，但是不属于W3C标准规范。

iPhone的浏览器首先支持这个新特性，而且很快——由于iPhone的流行——其他的移动设备终端也开始支持这个特性。

移动特性探测

由于每个移动设备支持的功能不一样，因此不能保证某项功能在任何浏览器上都能够正常使用。

例如，假设网站要使用HTML 5的Web存储（Web Storage）新特性，那么这个HTML 5新特性就只有部分智能手机支持（如iPhone、Android、Blackberry及Windows Phone 设备），其他手机都不支持。

传统上，开发人员需要依赖浏览器技术来判断自己的网站程序是否可以在特定的浏览器中运行。一种经典的方法就是去检查目标浏览器是不是Opera Mini，而不是检查它是不是支持Web存储。

这种方法有几个陷阱，需要当心！例如：

- 可能排除一些没有被明确包含但是实际支持这些功能的浏览器。
- 如果用户从另外一个设备访问我们的网站，可能有些功能不正常。

以下是这种方法的例子代码：

```
// 警告：不要使用这些代码
if(document.all) {
  // IE4+
  document.write('<link rel="stylesheet" type="text/css" src="style-ie.css">');
}
else if (document.layers) {
  // Navigator 4
  document.write('<link rel="stylesheet" type="text/css" src="style-nn.css">');
}
```

上面的示例代码中只为Internet Explore和NetscapeNavigator4提供了样式表。即使这样，浏览器必须启用JavaScript支持，这意味着其他浏览器如Netscape 6、Netscape 7、CompuServe 7、Mozilla（译注1）和Opera也许无法正常浏览网站。

即使已经支持了大部分浏览器，但还是有可能遗漏某一个已经支持了所要功能的新版本浏览器。另外一个问题就是错误识别浏览器。因为浏览器探测功能很大程度上需要使用用户代理（user agent）字符串和某些属性，所以很有可能我们错误识别了浏览器：

```
// 警告：不要使用这些代码
if(document.all) {
  // IE4+
  elm= document.all['menu'];
}
else{
  // Assume Navigator 4
  elm= document.layers['menu'];
}
```

注意，以上示例假定不是IE浏览器就是网景导航4（Netscape Navigator 4），从而试图访问layers对象。

这些问题的根源就是使用了基于Gecko和Opera内核的浏览器。因为这些原因，所以最好还是显式检查浏览器是否支持某个特性，而不是假定某个浏览器版本支持或者不支持某个特性。

下面的例子和上面的一样，重构成特性探测而不是浏览器探测：

```
// 如果有本地存储 localStorage，就使用它
if(('localStorage' in window) &&window.localStorage!== null) {
  // 简化对象属性 API
  localStorage.wishlist= '["Unicorn","Narwhal","Deathbear"]';

} else {

  // 没有 sessionStorage，只能使用 cookie
  // 使用 document.cookie 的 API:(
```

220

译注1：Mozilla 浏览器内核为 Gecko，IE 内核为 Trident（IE 4.0 版本及以上），苹果浏览器和谷歌浏览器内核为 WebKit（用于 Safari 以及 Google Chrome），Opera 浏览器内核为 Presto（2013 年 2 月 13 日改用 WebKit）。

```
    var date = new Date();
    date.setTime(date.getTime()+(365*24*60*60*1000));
    var expires = date.toGMTString();
    var cookiestr= 'wishlist=["Unicorn","Narwhal","Deathbear"];'+
                   ' expires='+expires+'; path=/';
    document.cookie= cookiestr;
  }
```

这不仅增加了可靠性，而且未来任何浏览器只要支持Web存储新特性，这段代码都将自动获得新特性。

CSS媒体查询

CSS媒体查询是一种渐进增强技术，可让我们根据不同浏览器的条件调整或显示备用样式。

CSS2规范允许我们根据媒体类型来指定样式，比如显示到屏幕(screen)和用于打印(print)。CSS3规范提供了**媒体查询**（media queries）的技术概念，这是一种帮助探测浏览器特性的标准方法。

非常不幸的是，CSS3规范还处于“候选推荐”阶段，这意味着媒体查询以及CSS3里的其他新特性不是所有的浏览器都能很好地支持。

因此，要给某些不支持新特性的浏览器提供备用样式，一旦浏览器不兼容，就可以使用备用样式。

我们已经知道，视区标签可以根据设备大小定义一个默认宽度，同时视区也可以让页面在默认缩放比例下正常显示。用户自己控制缩放时，视区不会起作用，会自动使用用户缩放的参数。

当布局宽度改变时，需要通知浏览器限制内容的宽度以便正常显示页面。

现在来看一个简单的使用CSS媒体查询特性来实现这个自动修改宽度功能的例子：

```
body{background-color:blue;}
@media only screen and (max-width: 800px) {
    body{background-color:red;}
}
```

因为CSS作用规则是自上向下的，所以先从设置Body的背景色为蓝色开始，然后定义一个设备专用的媒体查询规则来重新设置背景色。当屏幕的宽度小于800像素时，把背景色设置为红色。

在支持CSS3媒体查询新特性的设备上，当宽度小于800像素时，背景色就会变成红色；否则就保持蓝色。（虽然在放大或者缩小网页时或许并不希望改变背景色，但是这个例子仅仅是想演示CSS3的媒体查询新特性：基于某些条件可以动态改变网页的样式。）

非常重要的是，我们可以在开始部分定义通用规则，在支持媒体查询新特性的设备上再根据需要修改这些规则。

这种做法可以给支持新特性的浏览器带来更强大的用户体验，并保持可以在旧的浏览器上访问网站的基本功能。

浏览器专用视图

ASP.NET MVC 4的新特性显示模式（display mode）可以让我们根据预定义条件来加载不同的视图。使用这个新特性的例子就是要分别为智能手机、平板电脑和PC分别创建不同的视图。针对不同的设备创建不同的视图可以更好地利用屏幕空间，带来更好的用户体验。这种做法就叫定义浏览器专用视图。

在应用程序开始时注册为显示模式：

```
using System.Web.WebPages;

// iPhone 专用视图
DisplayModeProvider.Instance.Modes.Insert(0, new DefaultDisplayMode("iPhone")
{
   ContextCondition= (ctx=>ctx.Request.UserAgent.IndexOf(
      "iPhone", StringComparison.OrdinalIgnoreCase) >= 0)
});

// Windows Phone 专用视图
DisplayModeProvider.Instance.Modes.Insert(0, new DefaultDisplayMode("WindowsPhone")
{
   ContextCondition= (ctx=>ctx.Request.UserAgent.IndexOf(
      "Windows Phone", StringComparison.OrdinalIgnoreCase) >= 0)
});
```

222 现在，首先复制Auctions.mobile.cshtml文件，重新命名为Auctions.iPhone.cshtml，以此来创建一个iPhone专用视图。然后把标题修改为“iPhone Auctions”，以区分其他的视图。可以使用浏览器模拟器来访问网站，如图10-9所示。（这个例子使用Firefox浏览器的User Agent Switcher插件来模拟iPhone浏览器。）

为了查看Windows Phone版本的页面，再次复制一份Auctions.mobile.cshtml视图文件，然后重命名为Auctions.WindowsPhone.cshtml，并把页面标题修改为“Windows Phone Auctions”以区分其他视图，同样启用网站在浏览器模拟器里访问页面，如图10-10所示。

223

ASP.NET框架内部根据移动浏览器规范来检查请求是否来自移动设备。

可以在HttpBrowserCapabilities里获取浏览器的很多信息，这个对象可以在Request. Browserproperty属性里访问到。

当然，如果不想使用浏览器内置的定义信息，还可以使用类似51Degrees.mobi这种服务，它维护了各种各样的移动浏览器信息，从远古到最新的版本都有。

224

从零开始创建新的移动应用

Creating a New Mobile Application from Scratch

ASP.NET MVC 4不仅可以很容易向现有网站添加移动视图，而且创建一个全新的移动网站也是小菜一碟！当然这对没有传统网站的情况非常有用，不论什么原因，我们都不喜欢把移动网站和传统网站混在一起。

图10-9　iPhone专用视图

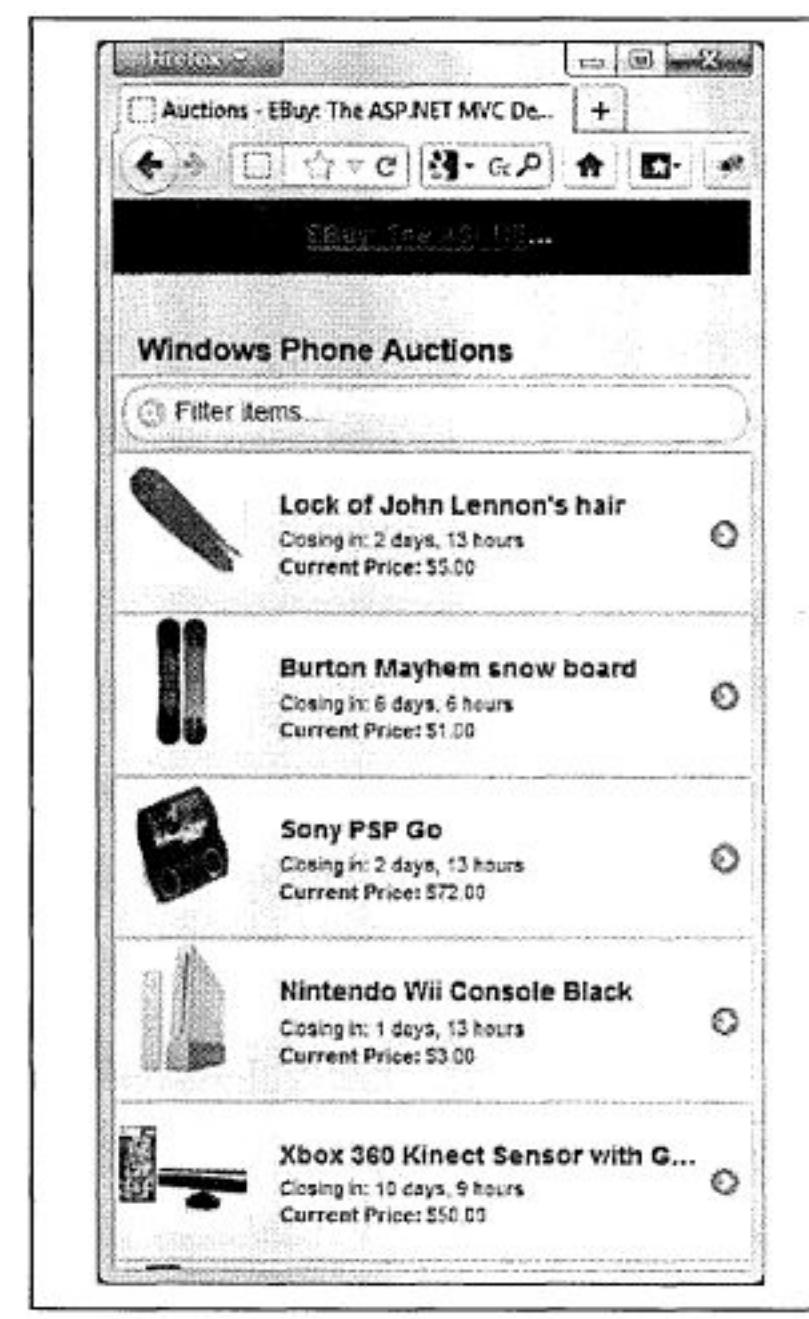


图10-10　Windows Phone专用视图

ASP.NET MVC 4包含了移动应用模板，使用这个模板可以很快进行移动应用开发。这个模板使用jQuery Mobile实现了大部分功能，虽然可以快速开发网站，但是先要理解jQuery Mobile框架。

jQuery Mobile范式转换

使用jQuery Mobile最重要的一个区别就是“页面”的概念。在传统的网站开发过程中，页面是指单个HTML文档或者是ASP.NET Web Form的ASPX、或者是ASP.NET MVC中的.cshtml视图文件。这些文件包含HTML标签以及渲染单个页面的逻辑代码。

jQuery Mobile中的页面有所不同，单个文件可能包含多个移动“页面”。从技术角度来说，jQuery Mobile页面实际是指包含`data-role="page"`属性的`<div>`标签。可以把所有的页面都放到一个文件里，jQuery Mobile会自动在浏览器里一次显示一个页面。

因为单个的普通网页很可能会分割成多个移动视图网页（这取决于移动设备版本页面的重新设计），所以这个方法可以帮助我们减少因分割页面而产生的视图文件数量。

ASP.NET MVC 4移动应用模板

创建ASP.NET移动应用网站的工作，非常简单，其做法跟创建其他的ASP.NET MVC网站一样：选择文件→新建→项目（`File→New→Project`），然后选择“ASP.NET MVC 4 Web Application

（ASP.NET MVC 4 Web应用）”，如图10-11所示。

选择图10-12所示的移动应用模板，就会创建一个带有例子控制器和视图的ASP.NET MVC移动网站。整个网站已经添加了ASP.NET MVC移动网站所需的基本条件，我们可以很快上手进行开发了。

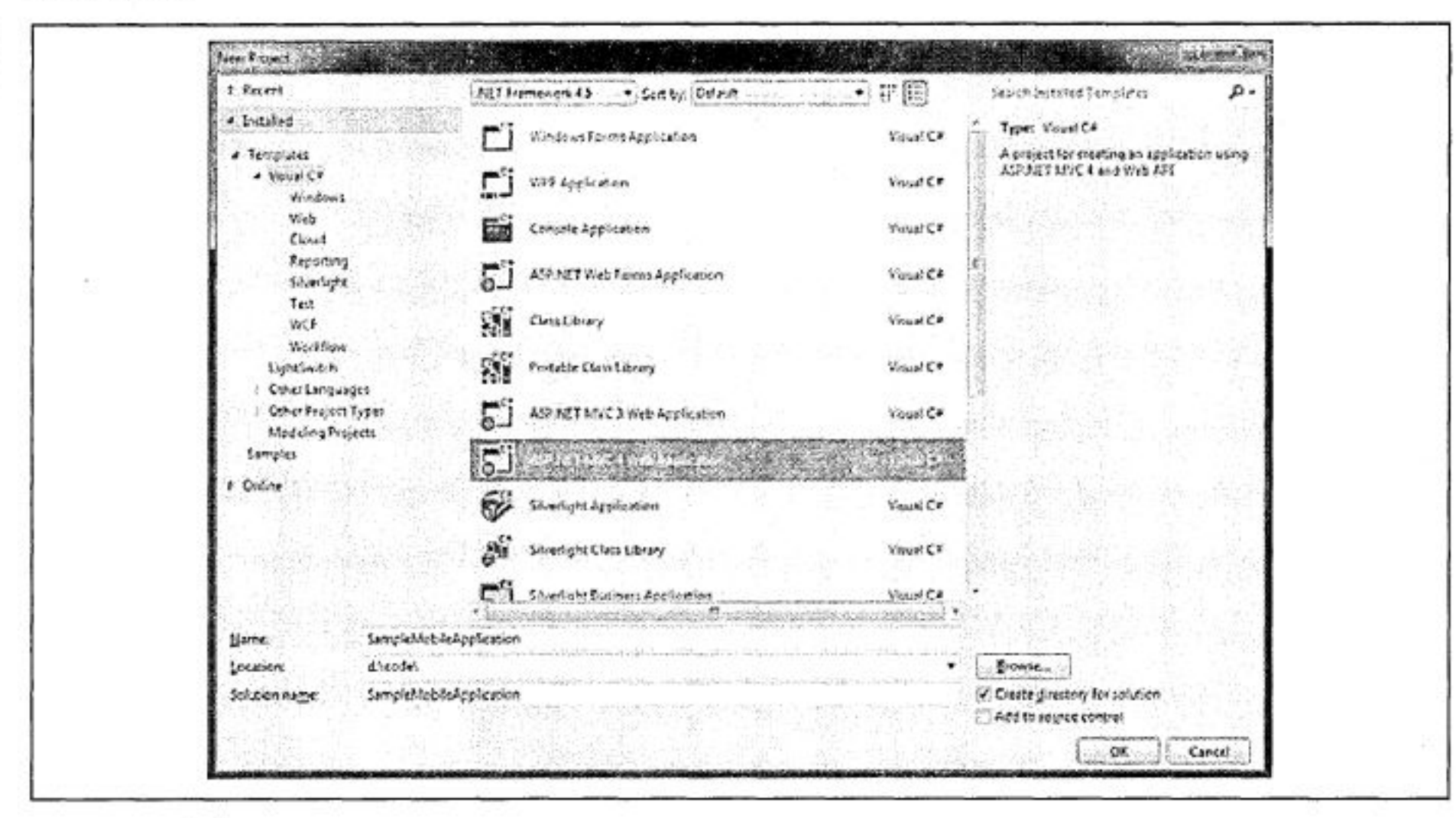


图10-11　新建项目

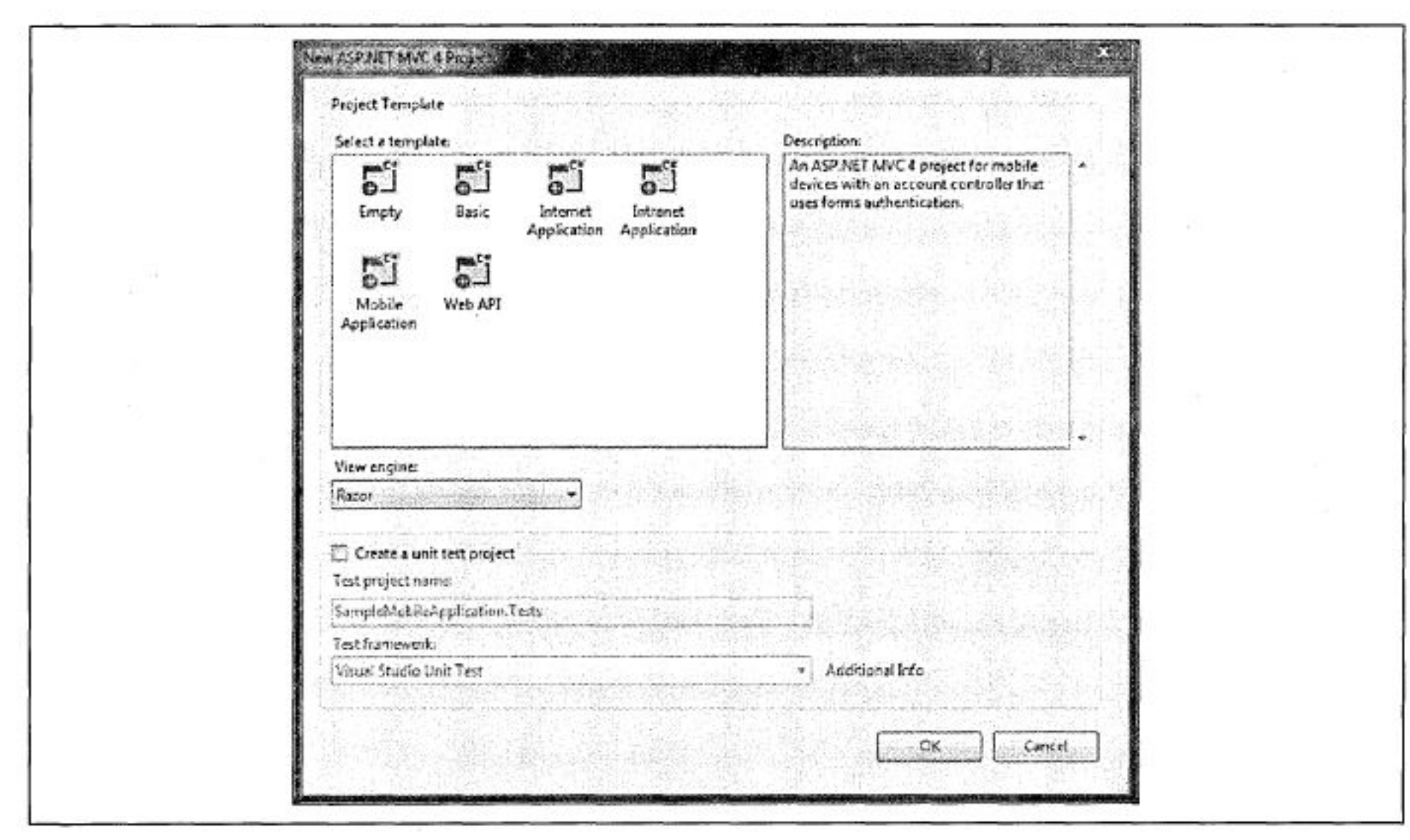


图10-12　选择移动应用模板

点击F5键运行这个网站，或者选择“Debug→Start（调试→开始）”菜单，Visual Studio就会编译整个解决方案，然后启动浏览器打开移动网站默认的首页（见图10-13）。

图10-13　移动网站默认首页

使用ASP.NET MVC 4移动应用模板

正如我们看到的，很多移动网站基础框架的代码已经写好了。第一眼看到这个移动网站，就发现和其他普通网站很相似，只是添加了一些新东西。

- Content文件夹包含jQuery Mobile的样式文件，如图10-14所示。
  —jquery.mobile-1.1.0.css（以及压缩版本）。
  —jquery.mobile.structure-1.1.0.css（以及压缩版本）。

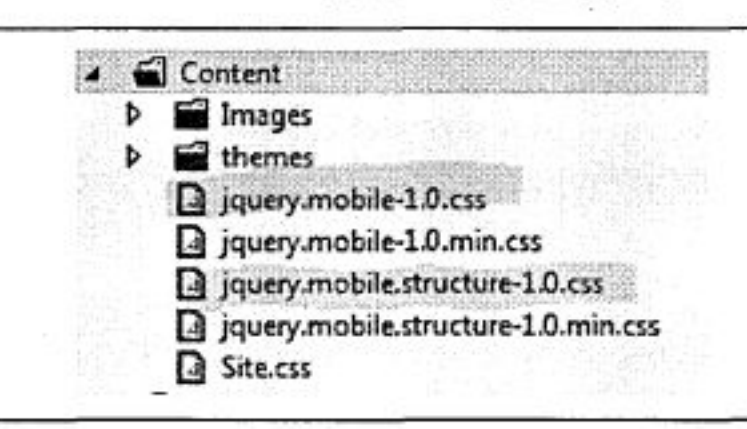


图10-14　新项目的Content文件夹

- Scripts文件夹包含两个文件，如图10-15所示。
  —jquery.mobile-1.1.0.js。
  —jquery.mobile-1.1.0.min.js。

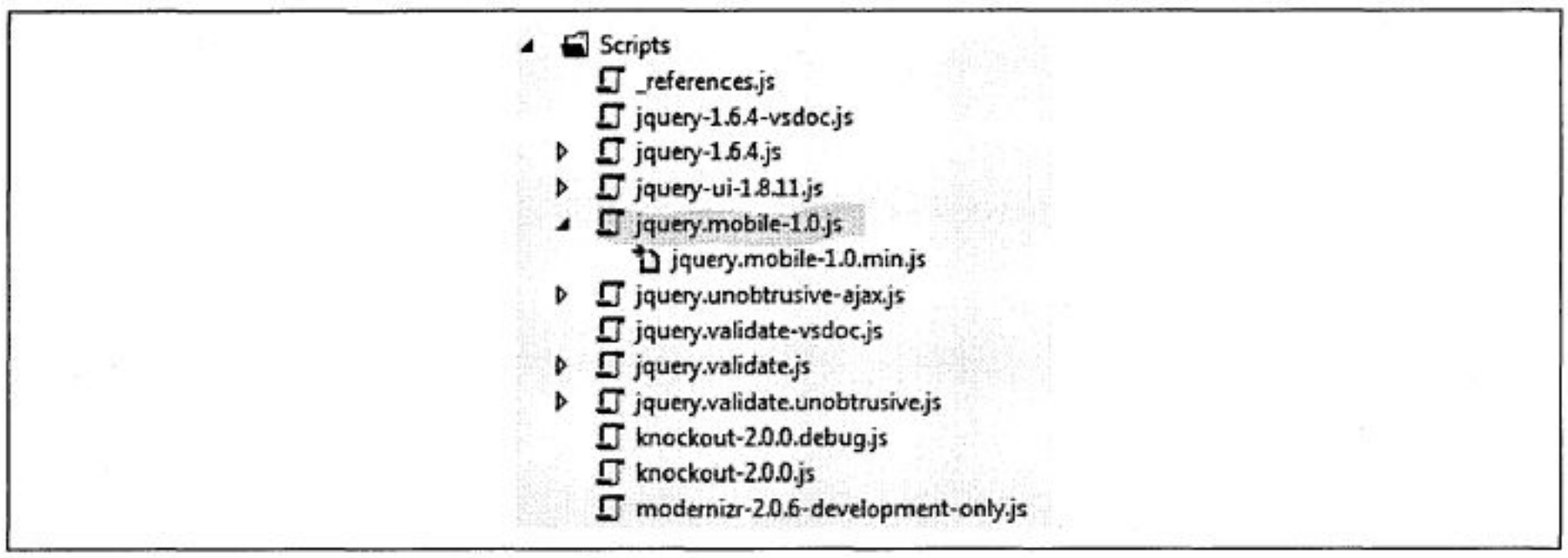


图10-15　新项目的Scripts脚本文件夹

这些新文件是jQuery Mobile框架的一部分，JavaScript框架为移动应用开发带来了jQuery和jQuery UI的许多优点。

现在来看修改版的_Layout.cshtml。其head标签包含一些新代码，viewport标签可以指定视区标签的尺寸。这一点非常重要，因为大部分浏览器都允许用户放大、缩小视图，设置页面内容的初始宽度可以提供更好的用户体验。正如前面提到的，"width=device-width"会自动把网页内容的宽度设置成设备的宽度：

```
<meta name="viewport" content="width=device-width" />
```

228

或者通过固定的像素来设置视区的初始宽度。例如，这些代码设置页面的初始宽度为320像素：

```
<meta name="viewport" content="width=320px" />
```

下面的标签包含引用了jQuery Mobile样式文件。这样就可以使用jQuery主题框架来配置网站主题了：

```
<link rel="stylesheet" a href="@Url.Content("~/Content/jquery.mobile-1.0b2.min.css")" />
```

这个script标签在页面上使用了jQuery Mobile框架的脚本文件，这样就可以使用诸如AJAX、动画、校验等功能了：

```
<script type="text/javascript" src="@Url.Content("~/Scripts/jquery.mobile-1.0b2.min.js")">↵
</script>
```

现在来看新版的HTML页面代码，它包含一些新的属性。jQuery Mobile通过data-role属性来区分不同的元素，比如页面、按钮、列表等。再看body标签，可以看到默认的模板已经包含 +s with these +data-role属性：

```
<body>
  <div data-role="page" data-theme="b">
    <div data-role="header">
      @if (IsSectionDefined("Header")) {
        @RenderSection("Header")
      } else {
        <h1>@ViewBag.Title</h1>
```

```
            @Html.Partial("_LogOnPartial")
          }
        </div>

        <div data-role="content">
            @RenderBody()
        </div>
      </div>
    </body>
```

第一个 标签包含了data-role="page"属性，ASP.NET MVC框架会把这个 当做页面对待；头部使用了datarole="header"属性；body内容带有data-role="content"属性。

jQuery Mobile为一些主要的HTML元素定义了各种属性，比如<H1>、<H2>、<P>和<table>元素，也包括列表和Form元素，比如按钮、文本、选择列表等。若要深入学习，则可以阅读jQuery Mobile网站的官方文档和Demo等技术资料。

229

总结
Summary

本章介绍了移动Web开发各方面的知识，比如，什么是“移动Web”以及移动Web与传统Web网站的区别。随后介绍了各种提升开发效率的开发框架和技巧，最后介绍了如何处理浏览器兼容性的问题，以便移动Web网站获得更好的用户体验。

ASP.NET MVC 4框架包含的移动开发的特性如下：

- 改进的移动应用模板；
- 通过重写布局、视图和部分视图来自定义布局；
- 支持特定浏览器（比如支持iPhone的视图）以及重写浏览器兼容性的功能；
- 通过jQuery Mobile框架来增强移动视图。

第三部分

会当临绝顶，一览纵山小

Going Above and Beyond

第11章

并行计算、异步和实时数据操作

Parallel, Asynchronous, and Real-Time Data Operations

传统的Web编程模型基于同步的C/S(客户端/服务器)通信模式，客户端发送请求给服务器后等待服务器返回应答消息。虽然这种模式可以很好地满足大部分情况，但是在处理长请求或者复杂的交易时效率就会非常低。

本章将介绍如何使用ASP.NET MVC框架强大的异步编程和并行处理功能来处理更复杂的场景，比如使用异步请求处理和实时双工通信从/向客户端发送和接收消息。

异步控制器

Asynchronous Controllers

当请求消息到达时，ASP.NET会从线程池里抓取一个线程来处理请求。如果请求是同步的，线程就会阻塞直到当前的任务结束才能处理别的请求。

绝大部分情况下，请求执行的过程非常快，ASP.NET可以处理一些阻塞的线程。但是，如果应用程序需要处理大量的请求消息或者有很多长请求，线程池就会没有多余的线程可用，也就会发生thread starvation（线程饥饿）现象。当出现这种问题时，Web服务器会将进入的新请求消息放入队列。当达到一定数量时，请求消息塞满了队列，就不会再接受新的请求，然后服务器就会返回503（服务器忙碌）的状态码。

为了避免发生线程池枯竭的问题，ASP.NET MVC控制器可以使用异步执行来代替同步执行（默认）。使用异步控制器不需要强求改变耗费的时间。它只是释放执行请求的线程，这样的线程会重新回到ASP.NET线程池。

这是异步处理请求的步骤。ASP.NET从线程池获得线程来处理请求，异步调用完ASP.NET MVC操作后，它就会将线程返回到线程池以便处理别的请求。异步操作在不同的线程上执行时，处理完请求后，它会通知ASP.NET。ASP.NET会重新抓取一个线程（可能不是第一次的线程），然后调用这个线程来处理请求。这个过程包括渲染结果（输出数据）。

创建异步控制器

创建异步控制器也非常简单，只需要继承基类`AsyncController`，这个基类提供了异步处理请求消息的帮助方法。

```
public class SearchController: AsyncController
{

}
```

因为SearchForBids()方法使用了复杂的LINQ查询逻辑，要耗费几秒才能完成处理，所以需要使用异步控制器：

```
public ActionResult SearchForBids(DateTime startingRange, DateTime endingRange)
{
  var bids = _repository
             .Query<Bid>(x =>x.Timestamp>= startingRange&&x.Timestamp ↵
             <= endingRange)
             .OrderByDescending(x =>x.Timestamp)
             .ToArray();

  return Json(bids, JsonRequestBehavior.AllowGet);
}
```

在使用ASP.NET MVC 4框架之前，一定要遵守下面的开发惯例来创建异步控制器。

操作名Async

此方法必须返回void，它会开始异步处理过程。

操作名Completed

此方法会在异步处理完成之后调用，它处理返回的结果ActionResult。

下面是配置使用了后台线程的SearchForBids()方法，异步查询账单信息：

```
public void SearchForBidsAsync(DateTime startingRange, DateTime endingRange)
{
    AsyncManager.OutstandingOperations.Increment();

    var worker = new BackgroundWorker();
    worker.DoWork+= (o, e) =>SearchForBids(Id, e);
    worker.RunWorkerCompleted+= (o, e) =>
        {
            AsyncManager.Parameters["bids"] = e.Result;
            AsyncManager.OutstandingOperations.Decrement();
        };

    worker.RunWorkerAsync();
}

private void SearchForBids(string Id, DoWorkEventArgs e)
{
        var bids = _repository
            .Query<Bid>(x =>x.Timestamp>= startingRange&&x.Timestamp<= endingRange)
            .OrderByDescending(x =>x.Timestamp).ToList();

        e.Result= bids;
}

public ActionResult SearchForBidsCompleted(IEnumerable<Bid>bids)
{
    return Json(bids, JsonRequestBehavior.AllowGet);
}
```

注意，AsyncManager.OutstandingOperations在操作开始前+1，在操作结束前-1。此代码是通知ASP.NET框架目前有多少个等待操作。当属性OutstandingOperations为0时，ASP.NET会完成所有的异步请求方法，并且调用SearchForBidsCompleted()方法。

代码虽然很多，但是幸运的是，.NET Framework 4.5引入了新的关键字async和await来简化异步编程模型。

可在微软MSDN官方文档里可看到更多关于.NET 4.5异步编程的介绍内容，网址为：*http://msdn.microsoft.com/en-us/library/hh191443(v=vs.110).aspx*.

下面是简化后的SearchForBids()方法的异步编程代码，使用了新的异步关键字：

```
public async Task<ActionResult>SearchForBids(string Id)
{
    var bids = await Search(Id);
    return Json(bids, JsonRequestBehavior.AllowGet);
}

private async Task<IEnumerable<Bid>>Search(string Id)
{
    var bids = _repository
          .Query<Bid>(x =>x.Timestamp>= startingRange&&x.Timestamp<= endingRange)
          .OrderByDescending(x =>x.Timestamp).ToList();
    return bids;
}
```


控制器操作返回一个可以配置超时时间的Task实例对象。要设置超时时间，可以直接使用AsyncTimeout标记属性。下面的例子展示一个设置了2500毫秒超时时间的控制器操作方法。如果超时，就会返回AjaxTimed-Out视图。

```
[AsyncTimeout(2500)]
[HandleError(ExceptionType = typeof(TaskCanceledException), View = "AjaxTimedOut")]
public async Task<ActionResult>SearchForBids(string Id)
{
}
```

何时使用异步控制器

关于何时使用异步操作并没有硬性规定。下面为一些使用同步操作的典型场景，可以帮助我们决定何时使用异步操作。

- 简单而且短的操作；
- 简单、比效率更重要的情况；
- CPU密集型操作（异步操作不会带来任何好处，还会增加操作负载）。

下面为一些使用异步操作的典型场景：

- 可能引发性能瓶颈的长操作；
- 网络或者I/O密集型操作；
- 允许用户取消长操作的能力。

实时异步通信

Real-Time Asynchronous Communication

万维网（World Wide Web）是一个不断变化的环境，几个月前能正常工作的应用模型或许现在已无法满足用户的需求了。越来越多的开发人员选择单个页面架构方法用来构建应用程序或者用来实时更新少数页面，而不是构建几十个页面的Web应用。

实时数据技术可以追溯到社交网站和移动设备爆炸式增长的时代。当今社会，生活节奏非常快，人们总是忙于奔波，大家都想立即获取最新的资讯，或者是自己喜欢的体育资讯，或者是热门的股票信息，也可能是朋友带来的最新的微博资讯。随着越来越多的人使用移动设备访问网站，Web网站探测网络可用性并很好地跨设备平台兼容浏览器显得尤为重要。

对比应用模型

传统的Web应用模型依赖同步通信。当用户访问网站时，就会通过浏览器向Web服务器发送请求，服务器会返回当前网站请求的数据。因为无法保证用户是否还会发送新的请求，所以用户当前请求的数据很可能不是最新的，会导致数据冲突。

可以使用类似于AJAX的技术来解决这种问题。绝大部分情况下，用户会继续提交请求。AJAX依赖传统的请求/应答模型，交互是典型的事务性的和原子性的，任何当前事务之外的交互都无法进行通信——只有重新发送请求才能够回到同步（back in sync）。只有在浏览器和服务端之间使用更加高级、支持长会话的技术才能支持这种场景。

现在来看其他的实时通信模型。记住：HTTP协议是围绕着请求/应答通信模式（request/response communication pattern）设计的，只有当客户端率先发起请求时才会服务和支持实时通信。

HTTP轮询

图11-1所示为一个通过模仿“持续连接”来实现会话的例子。这个例子使用的是标准的AJAX请求。这项功能通常是使用JavaScript计时器来定时发送AJAX请求的。

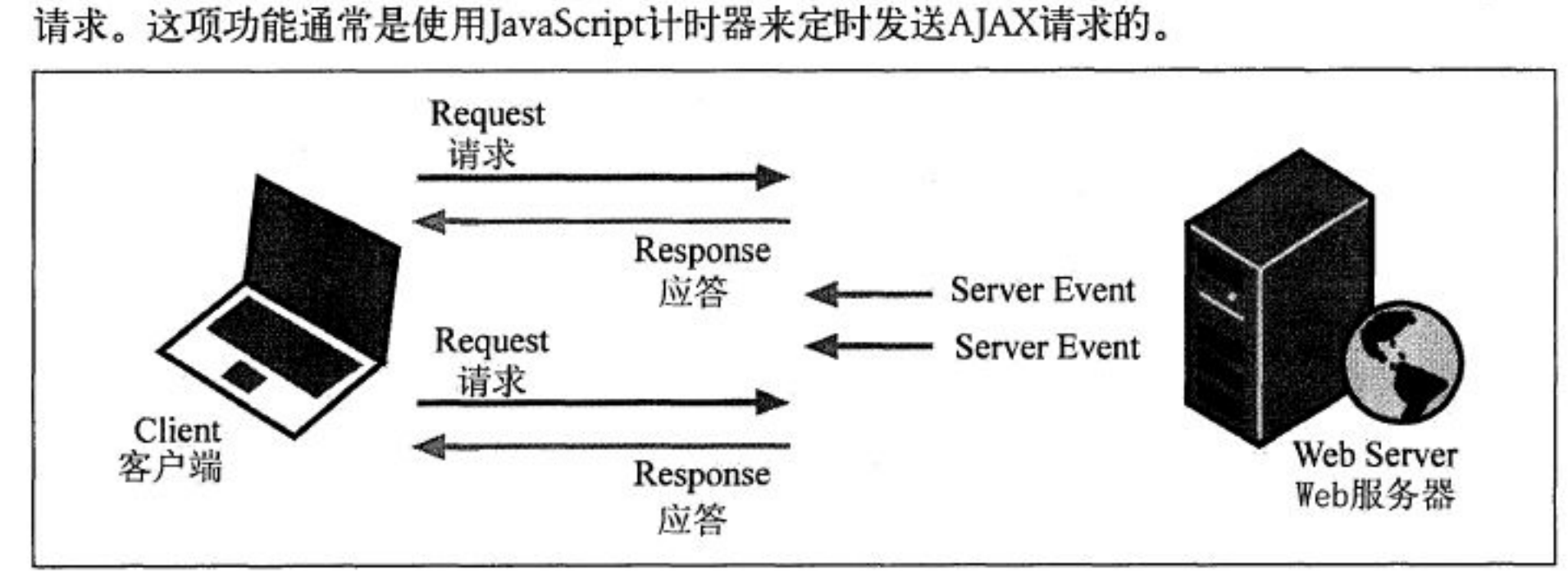


图11-1　HTTP轮询

图11-1所示为HTTP轮询的过程。这种技术最重要的一点就是浏览器在每个请求结束以后需要重新发起一个新的请求（无论完成的请求是否包含数据），需要浏览器支持高效的容错能力。

因此，轮询是最可靠，但是最不安全的实时通信方法——这种可靠性依赖高代价。如图11-2所示，轮询必须产生相对大量的网络带宽和服务器资源负载压力，特别是考虑到无论服务器是否有任何更新都必须处理请求的情况（很多请求都不会返回数据，当然不是大多数请求）。

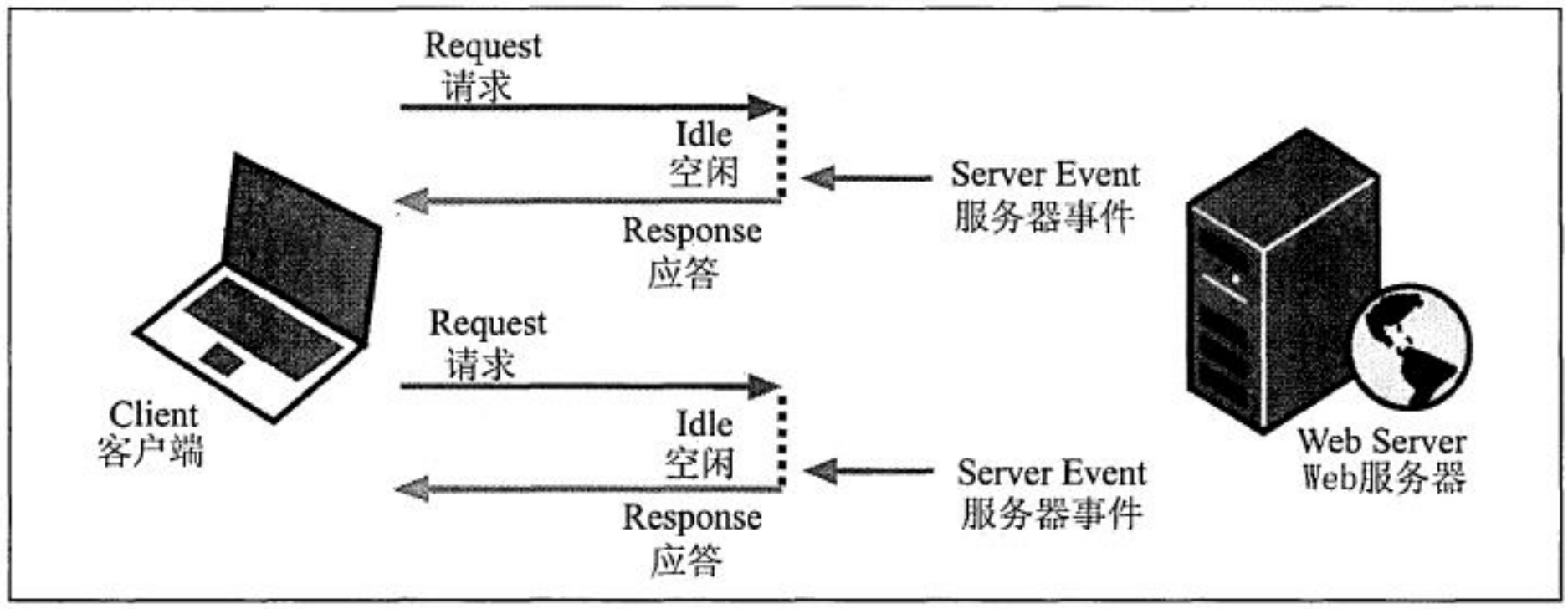


图11-2 HTTP长轮询

浏览器支持

轮询使用了各种起源于远古图形时代的浏览器技术，所以无论是什么浏览器，只要支持JavaScript就行。

轮询的缺点

轮询也有几个缺点。相对于实际传输的数据来说，大量的请求消息就是天大的浪费。客户端请求和服务器事件并不能一直保持同步，多个客户端请求之间很可能有多个服务器事件。如果无法检查，那么此方法可能带来DOS（denial-of-service，拒绝服务）攻击。

HTTP长轮询

HTTP长轮询技术实际上也是利用了AJAX技术，由客户端发送请求向服务端获取数据，服务器保持连接直到数据返回。这与前面短轮询里的请求/应答方法形成了鲜明的对比，短轮询里如果服务端无法提供AJAX请求所需的数据，它就会立即返回空数据应答消息。

长轮询需要在可能的未来服务器事件的预期使用请求。换句话说，需要服务器和客户端长期保持连接，以便服务器向客户端推送请求。客户端的请求到来后，建立连接，直到服务器触发事件或者连接超时（连接断开）。随后客户端必须重新发起请求，进行下一轮交互以便获取新数据（译注1）。

译注1：这种编程模型有点类似于WCF服务里的双工通信（回调）机制，都需要客户端提前向服务端发送请求（订阅事件），保持长连接，服务端后续可以回调客户端（通知客户端）推送事件。对此，《WCF服务编程》一书中有详细介绍。这也是大家熟知的设计模式中的“发布订阅模式”。

浏览器支持

因为长轮询有很多种实现技术，这种技术工作——有不同的可靠性——在不同的浏览器上。

长轮询的缺点

因为互联网基础结构设计是基于简单的HTTP请求/应答模式的，并没有提供对长连接的支持，所以长连接请求是不可靠的。因为经常断线，所以在长轮询中经常出现连接中断的现象。处理错误和处理成功请求一样重要。处理方法就是：重新发送一个请求。然而，这种复杂性导致此方法在实际项目中难以实现。

作为广泛支持的技术，长轮询实现往往只需要所有浏览器支持的最低功能，比如HTTP `GET`请求（可以应用到`IFRAME`或者`<script>`标签上的URL）。

服务端推送事件

服务端推送事件（EventSource）方法与长轮询类似：客户端发送HTTP请求给服务器，服务端保持连接直到返回客户端需要的数据。这两种方法的不同之处在于服务端推送事件方法不会在返回数据后立即关闭连接。相反，服务端会继续保持连接，以便向客户端推送更多的更新信息，如图11-3所示。

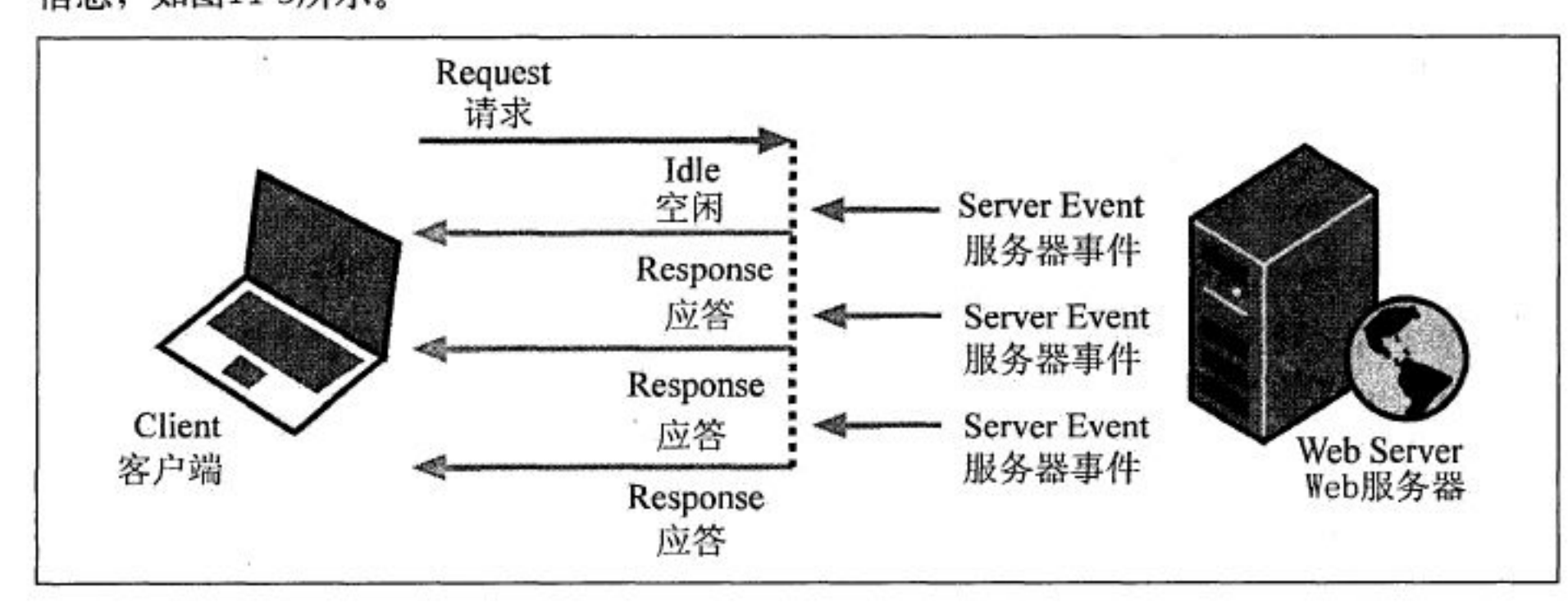


图11-3 服务端推送事件

服务端推送事件方法和它名字的含义一样，使用了服务端到客户端的单向通信。也就是说，客户端在建立连接第一次发送请求给服务器之后，就不能再发送任何信息给服务端，只允许服务端向客户端推送消息。客户端为了和服务端通信，必须提交额外的AJAX请求。但是，客户端不必关闭服务端推送事件通道来发送请求——客户端可以由标准的AJAX技术来发送请求信息给服务器，服务器可以通过服务端推送事件或者其他的AJAX请求来应答消息（或者两者同时使用）。

“EventSource”属于JavaScript EventSource API的一部分，是标准的客户端API（HTML 5规范的一部分），这种方法可以在浏览器里使用服务端推送事件。

浏览器支持

除了IE，绝大部分主流浏览器都提供了服务端推送事件新特性的支持。尤其是Chrome 9+、

Firefox 6+、Opera 11+和Safari 5+，还提供了原生支持。

服务端推送事件缺点

虽然这种方法允许服务端向客户端推送实时更新消息，但是它只允许服务端向客户端单向推送消息。也就是说，开放的通道不是双向的。客户端要通过别的AJAX请求和服务端继续通信。

WebSockets

WebSocket API是一个新的规范协议（HTML 5规范的一部分），它可以把HTTP请求连接转化为双向通信，即全双工TCP通信通道。最新版本的协议还提供了安全加密支持（还未广泛支持）。WebSocket的工作原理如图11-4所示。

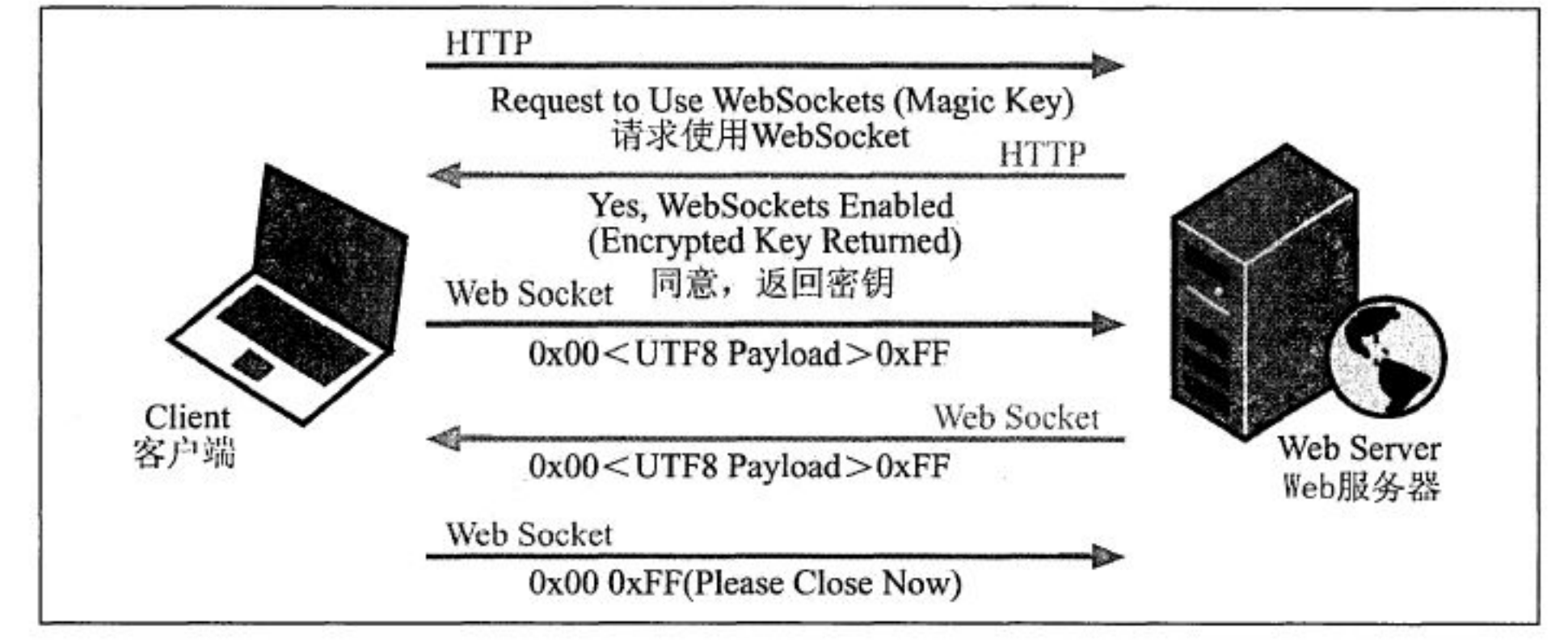


图11-4 WebSocket

浏览器支持

除了IE和Opera，绝大部分浏览器都提供了对WebSocket的某种程度的支持，IE 10、Firefox 6.0+、Chrome 4.0+、Safari 5.0+和iOS 4.2+还提供了对WebSocket的原生支持。

当遇到浏览器不支持WebSocket协议时，可使用别的机制代替（比如web-socket-js，使用Flash来实现）。这种方法也可以当做原生支持，因为可以临时解决整个问题。

WebSockets的缺点

虽然越来越多的浏览器支持WebSocket，但是毕竟不是所有的主流浏览器都提供了支持，所以无法保证所有用户都能够使用WebSocket。杀毒软件、防火墙以及HTTP代理都可能阻止WebSocket连接，导致WebSocket无法使用。

增强实时通信

在Web应用中添加实时通信已经不是什么复杂的任务了，现在开发人员借助.NET平台可以很

容易实现这项功能。幸运的是，微软认识到这个需求后，在.NET 4.0里引入了并行任务处理机制，而且ASP.NET专门创建了开源的异步信号库SignalR（译注2）。

SignalR可以轻易实现Web应用的实时双工通信功能。它提供了对HTTP连接的抽象支持，并提供给开发人员两种不同的编程模型：集线器（hubs）和永久连接（persistent connections）。整个库包含.NET服务API和客户端JavaScript库。

它支持几种不同的传输模型。每种模型定义了如何发送和接收数据，以及客户端如何与服务器连接。默认情况下，SignalR会根据浏览器来提供最好的传输模型（开发人员也可以自己指定传输模型）。

SignalR支持的传输模型如下：

- WebSocket；
- 服务器推送事件；
- 永久框架；
- 长轮询。

非常简单的安装SignalR包的方法是使用NuGet，输入命令即可：

```
Install-Package SignalR
```

242

持续连接

安装完SignalR包以后，就可以开始编写实时通信代码了。我们先来看如何配置持续连接在客户端和服务端之间发送消息。

首先，需要通过继承`PersistentConnection`基类来创建自定义连接对象。下面的代码展示了如何实现自定义连接类。重写`OnReceivedAsync()`方法后会给连接到服务器的所有客户端广播消息：

```
using System.Threading.Tasks;
using SignalR;

public class EbuyCustomConnection: PersistentConnection
{
    protected override Task OnReceivedAsync(IRequest request, string connectionId, ↵
    string data)
    {
        // 向所有客户端广播消息
        return Connection.Broadcast(data);
    }
}
```

其次，通过ASP.NET MVC路由表来注册自定义连接（为了避免冲突，要确保SignalR在其他路由之前）。

```
RouteTable.Routes.MapConnection<EbuyCustomConnection>("echo", "echo/{*operation}");
```

译注2：ASP.NET SignalR 由 Damian Edwards 和 David Fowler 发明，Damian Edwards 就职于微软，职位为 ASP.NET 程序经理。

现在，还要在客户端添加需要使用的SignalR JavaScript文件：

```
<script src="http://code.jquery.com/jquery-1.7.js" type="text/javascript"></script>
<script src="Scripts/jquery.signalR-0.5.0.min.js" type="text/javascript"></script>
<script type="text/javascript">
```

为了接收消息，需要初始化对象并订阅接收事件。最后一步就是通过调用connection.start()来发起连接请求：

```
$(function () {
  var connection = $.connection('/echo');

  connection.received(function (data) {
      $('#messages').append('<li>' + data + '</li>');
  });

  connection.start();
});
```

为了发送消息，需要调用连接对象上的send()方法：

```
connection.send("Hello SignalR!");
```

集线器模式

使用集线器模式（hubs）比创建自定义连接对象简单得多。集线器模式提供了基于PersistentConnection构建的远程过程调用(RPC)框架。应该在自定义连接对象上使用集线器模式来避免直接处理分发消息。

与自定义+PersistentConnection+s不同，集线器模式不需要任何路由配置，因为它可以通过简单的URL访问(/signalr)。

创建自定义集线器模式非常简单：创建一个类继承Hub基类，然后添加发送消息的方法。使用强类型客户端提供的用于客户端和服务端的自定义方法来进行通信（比如显示消息）。

```
public class EbuyCustomHub: Hub
{
    public void SendMessage(string message)
    {
        Clients.displayMessage(message);
    }
}
```

为了和集线器模式通信，首先要添加必需的SignalR JavaScript文件：

```
<script src="Scripts/jquery-1.6.2.min.js" type="text/javascript"></script>
<script src="Scripts/jquery.signalR-0.5.0.min.js" type="text/javascript"></script>
<script src="/signalr/hubs" type="text/javascript"></script>
```

为了接收消息，需要创建集线器模式的JavaScript代理实例，并且订阅客户端动态对象的方法。调用$.connection.hub.start()方法来初始化客户端和服务端之间通信的通道：

```
$(function () {
    // 动态创建代理
    var proxy = $.connection.ebuyCustomHub;

    // 声明回调函数
```

```
        proxy.displayMessage= function(message) {
            $('#messages').append('<li>' + message + '</li>');
        };

        // 开始连接请求
        $.connection.hub.start();
    });
```

为了发送消息，需要调用集线器模式上定义的公开方法。

```
connection.sendMessage("Hello SignalR!");
```


集线器模式允许扩展，具备很强的扩展性。我们可以很容易创建几个不同的方法来处理服务器事件。当然，除了发送基于字符编码的消息外，还可以在客户端和服务端之间发送JSON对象。

下面是支持多种消息类型的集线器模式的例子代码：

```
public class EbuyCustomHub: Hub
{
    public void PlaceNewBid(string jsonObject)
    {
        var serializer= new JavaScriptSerializer();
        var bid = serializer.Deserialize<Bid>(jsonObject);

        Clients.newBidPosted(bid);
    }

    public void AuctionClosed(Auction auction)
    {
        Clients.auctionClosed(auction);
    }
}
```

实际上，发送给Clients对象的.NET类型会自动序列化为JSON对象。而返回的消息需要手动使用JSON序列化器进行反序列化。

以下代码用来接收和发送新交易的通知消息。为了提交新的交易，需要创建并序列化JSON对象为字符串；进入集线器模式的事件参数Bid将会作为JSON对象传入：

```
$(function () {

    // 动态创建代理
    varproxy = $.connection.ebuyCustomHub;

    // 声明服务端回调方法
    proxy.newBidPosted= function (bid) {
        $('#bids').append('<li>Auction: ' + bid.Auction.Title+ ' Latest Bid: ' ↵
        + bid.Amount.Value+ ' </li>');
    };

    // 开始连接
    $.connection.hub.start();

    $("#postBid").click(function () {
        var bidToPost= GetBid();
        proxy.placeNewBid(bidToPost);
```

```
    });
});

function GetBid() {
    var bidPrice= $('#bidPrice').val();
    var newBid= "{ 'Code': 'USD', 'Value': '" + bidPrice+ "' }";
    return "{ 'Auction': {'Id': '61fdb6eb-b565-4a63-b048-0418dcb8b28d', 'Title': ↵
    'XBOX 360'}, 'Amount': " + newBid + "}";
}
```


配置和调整

与传统的短连接的Web应用不同，实时通信需要更长的连接周期。适当地监控和优化对根据应用需求实现SignalR的高性能通信以及资源的平衡至关重要。

管理SignalR连接

SignalR运行时通过`IConfigurationManager`接口暴露了可以用来配置SignalR连接的属性。表11-1列举了可配置的参数。

表11-1　SignalR配置参数

配置	描述
ConnectionTimeout	连接空闲的最大时间（默认为110秒）
DisconnectTimeout	等待连接关闭的最大时间（默认为20秒）
HeartBeatInterval	心跳包间隔时间（默认10秒）
KeepAlive	等待发送keepalive ping数据的连接空闲时间（默认为20秒）。当处于活动状态时，ConnectionTimeout失效；设置null表示关闭。

可以在应用程序启动时指定SignalR的设置参数：

```
// 修改连接超时为 60 秒
GlobalHost.Configuration.ConnectionTimeout= TimeSpan.FromSeconds(60);
```

配置环境

默认情况下，ASP.NET和IIS为了提供最好的伸缩性已经进行了最优的设置，以便管理大量的请求消息。为了支持实时通信，还需要修改一些设置来处理大规模的并发连接。

为了增加IIS处理并发请求的能力，要使用管理员权限来打开命令窗口，然后修改目录到%windir%\System32\inetsrv\下。最后运行下面的命令来修改默认`appConcurrentRequestLimit`的IIS连接从5000到100000：


```
appcmd.exe set config /section:serverRuntime /appConcurrentRequestLimit:100000
```

默认情况下，ASP.NET 4.0 每个CPU支持5000个连接。为了支持额外的CPU连接，在配置文件aspnet.config中需要修改`maxConcurrentRequestsPerCPU`的值：

```
<system.web>
    <applicationPool maxConcurrentRequestsPerCPU="20000" />
</system.web>
```

这个文件位于.NET Framework系统目录下（%windir%\Microsoft.NET\Framework\v4.0.30319 32位系统，%windir%\Microsoft.NET\Framework64\v4.0.30319 64位系统）。

当请求数量大于每个CPU的最大设置时，ASP.NET会使用队列来限制请求数量（例如，maxConcurrentRequestsPerCPU——机器上逻辑处理器的数量）。为了控制限流队列，可以修改machine.config文件里的requestQueueLimit设置（或者aspnet.config设置）。

为了修改限流队列的大小，可以把processModel元素的autoConfig属性设置为false，然后更新requestQueueLimit大小即可：

```
<processModel autoConfig="false" requestQueueLimit="250000" />
```

总结
Summary

本章首先介绍了如何在网站设计和开发中使用并行运算机制，然后介绍了如何使用异步控制器来处理长请求，最后介绍了如何在Web网站开发中使用SignalR实现实时双工通信。

第12章 缓存

Caching

每个网站都有一些变化很少的内容，尤其是静态页面，只有在网站更新或者内容页面变化时才更新，每隔一些天或者每隔数小时进行更新。

问题是，网站每次费了九牛二虎之力请求生成页面，可能还不知道：同样的内容已经重复生成了几千次。难道不能避免这种没有意义的事情吗？能不能只生成一次内容后使用相同的内容来处理后续的请求，避免一次又一次重新生成相同的内容？

这种存储和重用生成数据的概念就是缓存，这也是改善网站性能最有效的方法。那么应该缓存什么内容呢？要缓存多长时间？网站本身并不知道这些具体的需求。我们必须给网站提供需要缓存数据的清晰配置参数，帮助网站应用缓存策略。

幸运的是，ASP.NET Framework和ASP.NET MVC Framework都提供了许多缓存API来满足不同的需求。本章就会介绍各种缓存技术和API，以及如何使用这些技术来改善ASP.NET MVC网站的性能。

缓存的类型

Types of Caching

Web应用缓存技术大体上可以分为两类：服务端缓存和客户端缓存。虽然两种目标都是减少重复性内容的生成和网络传输工作，但是二者主要的区别就是缓存数据存储的位置不同，分为服务端浏览器和客户端浏览器。

服务端缓存

服务端缓存技术关注优化服务端数据查询、生成或者操作技术。其主要目标就是减少处理请求的工作量，减少数据库查询次数和生成HTML数据的CPU周期——减少每个bit的数据。

减少处理请求的工作量不仅可以减少每个处理请求耗费的时间，而且可以释放更多的服务器资源以便处理更多的请求。

客户端缓存

除了服务器缓存内容以外，现在很多新的浏览器都会提供几种客户端缓存技术。客户端缓存技术对改善应用程序性能提供了新的解决方法，它避免了向服务器重复提交获取重复数据的

请求次数，把一些重复数据缓存到本地。

服务端缓存技术的目标是为了更快地处理完客户端请求，而客户端缓存的目标则是为了避免不必要的请求。这种做法不仅可以改进客户端发送请求的用户体验，而且可以降低服务器的负载压力，同时也有助于提升所有用户的网站体验。

要知道，客户端缓存技术和服务端缓存技术各有千秋，两种技术同样至关重要。通常最有效的缓存策略就是将两种技术结合起来使用。

服务端缓存技术

Server-Side Caching Techniques

当使用服务端缓存时，从最简单的内存缓存技术到复杂的分布式缓存服务器，会有很多种技术供我们选择。事实上，ASP.NET MVC提供的很多缓存技术并不是来自ASP.NET MVC框架，而是来自ASP.NET框架。

下面会介绍在ASP.NET MVC Web应用程序里经常使用的服务端缓存技术。

请求域内的缓存

每个ASP.NET请求都会在ASP.NET框架里创建一个新的`System.Web.HttpContext`对象实例，这个上下文对象实例会贯穿整个请求的处理过程。

249

`HttpContext`对象的一个属性就是`HttpContext.Items`，这个属性也会存在于请求的整个生命周期，在此期间很多组件都会操作这个属性。

此属性的访问性——只能存在于当前请求范围内——`Items`字典对象是存储与当前请求有关数据的绝佳方式。使用`HttpContext.Items`对象作为中间媒介而不让组件之间直接彼此交互的方式为组件之间传送数据提供了相对松耦合的方式。

因为这只是个简单的`IDictionary`类型，所以使用`Items`属性非常简单。

例如，下面就是如何在Items集合里存储数据的例子代码：

```
HttpContext.Items["IsFirstTimeUser"] = true;
```

同样，查询数据也非常简单，代码如下：

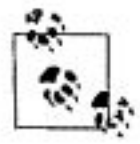
```
bool IsFirstTimeUser= (bool)HttpContext.Items["IsFirstTimeUser"];
```

注意，`Items`字典属性不是强类型的，所以必须做类型映射，转换为需要的类型后再使用它。

用户域内的缓存

ASP.NET会话状态（session state）允许我们在多个请求之间存储共享数据。但是ASP.NET会话状态只能存储针对每个用户的应用数据。也就是说，这些数据只能为当前的用户所用。

当启用会话时，组件就可以访问HttpContext.Session或者Session属性，以便为同一个用户缓存数据，供该用户后续的请求消息使用。



因为ASP.NET会话状态只能针对当前用户有效，所以它无法提供跨用户级别的数据共享功能。

与HttpContext.Item字典类似，Session对象也是非强类型的字典类型，所以也可以用相同的方式交互。

例如，下面的代码展示了如何在会话对象里存储username字段：

```
HttpContext.Session["username"] = "Hrusi";
```

从会话对象里读取信息并进行类型转换的代码如下：

```
string name = (string)HttpContext.Session["username"];
```

会话生命周期

存储在Session里的对象信息直到服务器销毁对象才被释放掉。通常在用户空闲一段时间后，会话才会过期。

默认的会话超时时间是20分钟，可以直接在网站的web.config配置文件的system.web>sessionState下进行修改。例如，下面的代码修改了默认的会话超时时间为30分钟：

```
<system.web>
    <sessionState timeout="30" />
</system.web>
```

存储会话数据

会话状态存储的位置是可以灵活设置的。默认的会话存储位置是在当前服务器的内存中，但是也可以通过设置ASP.NET Session State Service（会话状态服务）和SQL Server等其他数据源来存储会话信息，把会话信息存储在其他机器上。

应用程序域内的缓存

ASP.NET提供了HttpApplicationState类来存储应用程序级别的数据，可以通过HttpContext.Application属性来操作这个缓存对象。HttpContext.Application是个键/值对集合类型，这一点与HttpContext.Items和会话对象HttpContext.Session类似。除了生命周期和作用范围不同外，其他使用方式和数据结构级别类似。

可以在HttpApplicationState存储数据，代码如下：

```
Application["Message"] = "Welcome to EBuy!";
Application["StartTime"] = DateTime.Now;
```

从HttpApplicationState读取数据进行类型转换的代码如下：

```
DateTime appStartTime= (DateTime)Application["StartTime"];
```

存储在HttpApplicationState的数据生命周期与托管网站的IIS工作进程一样长，因为IIS不是ASP.NET，管理着工作线程的生命周期。一定要知道，HttpApplicationState也许不是可靠的数据存储方式。

只有存储跨工作进程的数据时才会使用HttpApplicationState（译注1），也就是说，HttpApplicationState是应用程序级别的存储器。例如，如果从磁盘文件上读取数据或者从数据库里读取数据，而且这些数据变化很少，就可以使用HttpApplicationState来存储这些数据，以减少昂贵的数据库查询开销。

251

ASP.NET缓存

除了HttpApplicationState外，还有一种存储应用程序级别数据的方法，就是使用System.Web.Cache对象。可以通过HttpContext.Cache属性来访问这个对象。

System.Web.Cache与HttpContext.Items和HttpSessionState一样，也是个键/值集合，但是，存储的数据不好限制在单个请求或者用户会话范围内。事实上，HttpContext.Cache与HttpApplicationState很像，除了后者可以跨工作进程访问以外，它还删除了HttpApplicationState一些固有缺点，因此它是一种不错的选择。

ASP.NET会自动管理缓存数据的清理工作，它会通知应用程序何时删除数据，所以可以再次填充数据。当下列情况发生时，ASP.NET就会删除缓存数据：

- 缓存数据失效；
- 缓存依赖项目失效；
- 服务器内存资源耗尽。

过期

向Cache对象添加数据时，就可以指定数据的有效时间。这个生命周期可以通过以下两种方式指定。

悄悄过期

用于指定缓存数据在最后一次访问之后多长时间过期。例如，当设置这个时间为20分钟时，如果该缓存数据每分钟不停地被访问，那么这个数据就会无限期待在缓存里（假设缓存数据没有任何依赖，而且服务器内存资源充足）。如果应用程序最后一次访问该缓存数据超过了20分钟，那么这个数据就会失效。

绝对过期

用于指定缓存数据失效的具体时间。与上一种方式不同的是，这种指定的是失效的时间点，不在乎访问次数。例如，如果指定某个缓存数据的失效时间是10:20:00 PM，那么在10:20:00 PM之后，该缓存数据一定失效，无法访问。

针对一个缓存数据只能使用一种过期策略，不能够同时指定两种过期策略。但是，可以针对不同的缓存数据使用不同的失效策略。

译注1：HttpApplicationState 可以存储配置文件的数据，很多网站实现的在线用户人数也可以借助 HttpApplicationState 来实现。

缓存依赖

我们也可以为缓存数据指定依赖项目，由该项目来决定其生命周期，比如文件或者数据库。当修改依赖项目时，ASP.NET会从缓存里删除该缓存数据。

例如，如果网站要显示一个由XML文件生成的报表，就可以把这个报表对象中的内容缓存起来，并把依赖项指定为这个XML文件。当XML文件更新时，ASP.NET就会从内存中删除报表对象。下一次请求报表时，ASP.NET代码首先会检查报表是否存在，如果不存在，就重新生成一次。这样可以保证每次都使用最新版本的数据生成报表。

文件依赖不是ASP.NET提供的唯一依赖方式——ASP.NET也提供其他几种类型的依赖，如表12-1所示，可以根据自己的需要来创建不同的依赖项。

表12-1　缓存依赖策略

依赖类型	定义
Aggregate（聚合）	提供多个依赖（通过`System.Web.Caching.AggregateCache`）。只有删除所有依赖项才会删除该缓存数据
Custom（自定义）	缓存数据依赖一个继承自`System.Web.Caching.Cache`的自定义类。例如，可以创建一个Web服务缓存依赖，当查询某个值时就删除缓存数据
File（文件）	缓存数据依赖某个外部的文件，删除文件就会删除该数据
Key（键）	缓存数据依赖应用程序缓存里的另外一个缓存数据（通过缓存的key关联）。删除依赖的缓存数据就会删除该数据
SQL	缓存数据依赖Microsoft SQL Server某表的数据变化。当表更新时，缓存数据就会删除

清理

当内存资源不足时，删除某些缓存数据的过程叫**清理**（scavenging）。删除的缓存数据通常都是在特定的时间范围内未被使用的数据，或者标识了低优先级的数据。ASP.NET提供了`CacheItemPriority`对象来指定缓存数据的优先级，以管理缓存数据清理时的删除策略。

在所有的情况中，ASP.NET提供的`CacheItemRemovedCallback`回调函数用于通知应用程序：某些缓存项目被删除。

输出缓存

虽然上面介绍的缓存技术都关注于缓存数据本身，但是ASP.NET也提供了更高级别的缓存机制，即为请求消息缓存生成的HTML数据。这种技术称为**输出缓存**（output cache），这是第一版ASP.NET框架就提供的特性。为了让输出数据尽量简单，ASP.NET MVC框架提供了`OutputCacheAttribute`标记属性：可以告诉ASP.NET MVC把渲染的控制器操作结果保存到缓存的过滤器。

使用OutputCacheAttribute配置控制器操作缓存非常简单。默认情况下，这个标记属性对HTML内容设置60秒的绝对缓存时间。在下一次请求没有到达，而缓存的HTML数据失效后，ASP.NET MVC会再次执行操作，生成相同的HTML代码缓存起来。

现在来看ASP.NET MVC输出缓存。先在EBuy电子购物网站的一个控制器操作上添加OutputCacheAttribute标记属性：

```
[OutputCache(Duration=60, VaryByParam="none")]
public ActionResult Contact()
{
  ViewBag.Message= DateTime.Now.ToString();
  returnView();
}
```

当设置完输出缓存之后再执行操作，就会看到ViewBag.Message内的数据每60秒修改一次。为了进一步验证，可以在方法上加个断点，就会看到断点只在第一次请求时才会执行（当缓存版本不存在时）。当然，每次更新之后也会重新执行一次代码。

配置缓存地点

OutputCacheAttribute包含结果参数，允许我们完全控制页面内容的缓存地点。

默认情况下，参数Location设置为Any，这意味着内容可以缓存到三个地方：Web服务器、任意代理服务器和用户浏览器。可以修改Location参数为下面的值：Any、Client、Downstream、Server、None或ServerAndClient。

默认的值Any可以满足大多数情况，但是不适用于需要进行细粒度控制缓存的情况。

例如，在加入要缓存显示当前用户名的页面中，如果使用了Any值，那么第一个请求页面的人的姓名就会显示给每个用户。

为了避免这种情况，可以使用Location设置为OutputCacheLocation.Client和NoStore，把数据存储在用户的浏览器中：

254

```
[OutputCache(Duration = 3600, VaryByParam= "none", Location = OutputCacheLocation.Client,↵
NoStore= true)]
public ActionResultAbout()
{
    ViewBag.Message= "The current user name is "+ User.Identity.Name;
    return View();
}
```

根据请求参数设置输入缓存

输出缓存最强大的地方是可以根据请求参数来缓存同一个操作、不同版本的输出结果。

例如，加入名为Details操作方法，用来显示auction交易的详细信息：

```
public ActionResult Details(string id)
{
  var auction = _repository.Find<Auction>(id);
  return View("Details", auction);
}
```

如果使用了默认的输出缓存设置，那么每次请求页面都会显示相同的产品信息。为了解决这个问题，可以根据参数或者查询字符串参数来创建不同的缓存内容，通过VaryByParam属性来设置：

```
[OutputCache(Duration = int.MaxValue, VaryByParam = "id")]
public ActionResult Details(string id)
{
  var auction = _repository.Find<Auction>(id);
  return View("Details", auction);
}
```

VaryByParam属性提供了一些选项用于制定创建的缓存版本。如果指定为None，就会一直显示第一次请求的页面内存。如果使用*，则每次请求都会显示不同的缓存。也可以通过分割查询字符串来定义列表或者查询字符串参数缓存规则。

表12-2列举了OutputCacheAttribute定义的属性。

表12-2 输出缓存参数

参数	描述
CacheProfile	使用的输出缓存策略的名字
Duration	缓存内容的生命周期
Enabled	是否启用缓存
Location	缓存地址
NoStore	是否启用HTTP Cache-Control
SqlDependency	缓存依赖的数据库和表名称
VaryByContentEncoding	逗号分隔的字符编码列表，用来区分输入缓存
VaryByCustom	自定义字符串用来区分输出缓存
VaryByHeader	逗号分隔的HTTP消息头，区分缓存
VaryByParam	通过分割的Post表单或者查询字符串来区分缓存

255

输出缓存配置信息

也可以通过在整个应用程序的web.config配置文件里定义output cache profiles来使用全局缓存规则，而不需要为每个控制器操作设置OutputCacheAttribute标记属性。

因为输出缓存配置存在于单个配置文件中，所以通过修改缓存逻辑就可以一次性控制整个网站。有个优势就是这个缓存配置修改并不需要重新编译或者重新部署整个网站就可以起作用。

为了使用输出缓存配置，需要在网站的web.config配置文件里添加输出缓存节点，并设置与缓存相关的参数。

例如，下面的ProductCache配置缓存一个小时的页面内容，而且根据请求的id参数来区分缓存数据：

```
<caching>
  <outputCacheSettings>
    <outputCacheProfiles>
      <add name="ProductCache" duration="3600" varyByParam="id"/>
    </outputCacheProfiles>
  </outputCacheSettings>
</caching>
```

也可以使用缓存配置文件，代码如下：

```
[OutputCache(Duration = 0, VaryByParam = "none")]
public JSONResult Index()
{
  User user= new User { FirstName= "Joe", LastName= "Smith"};
  return Json(user);
}
```

甜甜圈缓存

对复杂的动态网站来说，需要经常缓存整个页面，可能需要缓存动态生成页面的某些内容。

例如，在电子交易网站EBuy中，绝大部分情况下，缓存首页可以提升性能，但不会缓存与用户登录相关的页面内容，例如登录后显示的当前用户信息UserName——显然，不需要给每个用户缓存相同的用户名信息。

如果认为解决办法是使用VaryByParam根据用户ID来配置OutputCache缓存，那么就要再思考一下。因为OutputCache存储的是整个页面，所以使用这种方法，网站将会为每个不同的用户名缓存一个页面（无论动态字段是什么）。除了部分信息，大部分页面数据都是多余的。这也就是为什么会引入甜甜圈（译注2）缓存（donut caching）概念的原因。

虽然ASP.NET MVC的Razor视图引擎没有支持甜甜圈缓存，但是ASP.NET Web Form提供了替代控件（substitution control），用于处理页面部分动态更新数据，代码如下：

```
<header>
    <h1>Donut Caching Demo</h1>

    <div class="userName">
        <asp:Substitutionrunat="server" MethodName="GetUserName" />
    </div>
</header>

<!--页面其余缓存的内容-->
```

译注2：这是ASP.NET MVC提供的缓存机制，与之前ASP.NET Web Form提供的针对控件级别的缓存机制有点类似。甜甜圈，顾名思义是一种甜的圈状的食品，中间带有一个孔。这个名称是个类比，非常形象，是指甜甜圈缓存只缓存页面的外围部分内容，允许部分内容动态变化；甜甜圈洞缓存（donut hole caching）与之相反，缓存内部内容，而允许外部数据更新。

这个控件使用ASP.NET输出缓存注册了一个回调事件，当请求缓存页面时在页面上调用一个静态方法：

```
partial class DonutCachingPage: System.Web.UI.MasterPage
{
  public static string GetUserName(HttpContext Context)
  {
    return"Hello " + Context.User.Identity.Name;
  }
}
```

无论何时请求DonutCachingPage，除了username部分的内容每次会动态生成之外，其余会返回整个缓存页面。

因为ASP.NET MVC是基于ASP.NET平台之上的，所以可以使用替代控件的API，在ASP.NET MVC里实现同样的功能。HttpResponse包含一个WriteSubstitution()方法，替代控件的代码就是使用了这个方法。

使用这个方法，就可以通过自定义HtmlHelper来复制相同的逻辑：

```
public delegate string CacheCallback(HttpContextBase context);

public static object Substitution(this HtmlHelper html, CacheCallbackccb) {
    html.ViewContext.HttpContext.Response.WriteSubstitution(
        c=>HttpUtility.HtmlEncode(
            ccb(new HttpContextWrapper(c))
        ));
    return null;
}
```

使用了这个扩展方法，我们就可以在帮助方法里重写前面的例子了：

```
<header>
    <h1>MVC Donut Caching Demo</h1>

    div class="userName">
     Hello @Html.Substitution(context =>context.User.Identity.Name)
    </div>
</header>

    <!--页面其余缓存的内容-->
```

现在除了 标签代码，整个视图都会缓存起来，ASP.NET MVC就会实现与ASP.NET Web Forms替代控件（Substitution control）类似的功能。

MvcDonutCachingNuGet包

上面这个例子展示了一个简化版本的甜甜圈缓存，不过它还无法满足很多高级场景。虽然甜甜圈缓存没有集成到ASP.NET MVC 4框架里，但是MvcDonutCachingNuGet包可以帮助我们实现很多高级场景。

这个包为HTML帮助方法包添加了几种高级扩展方法，也提供了操作方法使用的DonutOutputCacheAttribute标记属性，以方便使用甜甜圈缓存。

甜甜圈洞缓存

甜甜圈洞缓存与甜甜圈缓存正好相反：甜甜圈缓存缓存整个页面，只有部分区域不被缓存；而甜甜圈洞缓存缓存的只是页面的一小部分（甜甜圈洞）。

例如，Ebuy交易网站包含了一个交易类别列表，这个列表基本不会变化，所以只缓存这一部分信息的HTML代码就很有价值，只生成一次。

甜甜圈洞缓存在这种情况下非常有用，因为页面的大部分内容都是动态的，只是很少一部分变化，或者根据参数变化。与甜甜圈缓存不同的是，ASP.NET MVC对甜甜圈洞缓存提供了很好的支持，我们可以在操作上直接使用。

我们来看在EBuy交易网站的类别上使用甜甜圈洞缓存的例子，下面是视图的代码：

```
@{
    Layout = null;
}

<ul>
    @foreach(var category in ViewBag.Categories as IEnumerable<Category>)
        <li>@Html.ActionLink(@category.Name, "category", "categories",
                                                    new { categoryId = category.Id })</li>
</ul>
```

部分视图会迭代所有的类别后渲染成列表形式。每个类别都包含一个超链接，可以连接到Categories类别对应的控制器，这里使用参数操作类别Id。

现在，我们来创建显示视图的Action操作方法：

```
[ChildActionOnly]
[OutputCache(Duration=60)]
public ActionResult CategoriesChildAction()
{
    // 从数据库获取类别信息，然后通过操作传递给视图
    ViewBag.Categories= Model.GetCategories();

    return View();
}
```

注意OutputCacheAttribute是如何缓存60秒数据的。

然后通过调用@Html.Action("CategoriesChildAction")在上级视图中调用新的操作，代码如下：

```
<header>
  <h1>MVC Donut Hole Caching Demo</h1>
</header>

<aside>
  <section id="categories">
    @Html.Action("CategoriesChildAction")
  </section>
</aside>

<!--未缓存的页面数据-->
```

现在当渲染页面时，Categories类别的子操作方法可以调用生产类别列表。

此次调用的结果会通过OutputCacheAttribute缓存起来，所以当下次渲染页面时，就会直接从缓存中获取Categories类别列表，而不像其他页面数据一样是动态生成的。

分布式缓存

若网站要运行在多个Web服务器上，那么请求消息可能被其中某个服务器处理。每次请求都会到达不同的服务器，如果请求的缓存数据没有在当前服务器上，就需要再次生成一次缓存数据。

当然这也与生成缓存数据的复杂程度有关系，如果一遍又一遍地重复生成缓存数据，效率就会变得非常低。如果使用更高效的解决方案，就只生成一次数据，可以存储在多个服务器或者Web Farm（译注3）中。

这种在一个应用实例上缓存数据并共享给其他网站的技术叫**分布式缓存**，这也是所有缓存技术中最完善的一种。

分布式缓存是对普通缓存技术的扩展，通过把存储在数据库或者会话中的数据存储在某个中心位置，让所有的应用都可以访问到。

使用分布式缓存的好处非常多，大致如下。

性能

因为服务器内存中可能存储了大量的数据，所以可以减少服务器I/O磁盘查询时间，显著提升数据读取的性能，提高页面加载速度。

伸缩性

伸缩性为集群的基本功能，可以增加或者删除节点服务器，允许应用程序扩展硬件以满足更高的要求以及更轻松地响应请求。与云存储结合以后，群集中的节点可以根据需求进行调整，高峰时增加节点，低潮时释放资源，以获得最好的经济效益。

冗余

当有一个服务器节点出错时，冗余保证整个应用程序不会受到影响。相反，其他的服务器会重新来处理刚才的请求而不需要人工干预。对许多分布式缓存解决方案来说，故障转移和冗余都是必不可少的组成部分。

分布式缓存解决方案

现在可以使用很多种分布式缓存的解决方案，虽然每种解决方案提供的API和处理数据的方式不同，但是分布式缓存的基本概念还是相通的。为了介绍分布式缓存的概念，下面看一下如

译注3：Web Farm（Web 农场）是指物理上独立的服务器群集。为了提高伸缩性和性能，每个 Web 程序运行于独立的 Web 服务器上。另外一个概念 Web Garden（Web 园），是指应用程序池包含多个工作进程，也是用于提升网站性能，但是它是单个服务器。这两个概念有时候会出现在 ASP.NET 高级面试题中。

何实现微软分布式缓存解决方案，称为Velocity（速度）（译注4）。EBuy交易网站里也使用了这个分布式解决方案。

微软分布式缓存Velocity是Windows AppFabric（微软应用服务器）的缓存层。所以，要安装Velocity，就必须下载Windows AppFabric或者通过Web Platform Installer下载安装。

开始安装之后，会看到图12-1所示的功能选择页面。选择“Caching Services（缓存服务）”和“Cache Administration（缓存管理）”功能。如果使用Windows 7，安装IIS 7管理器远程管理扩展插件，它可以允许我们通过Windows 7机器很方便地远程管理IIS 7。

如果只想使用缓存部分，那么只要选择自动安装或者使用SETUP /i CACHINGSERVICE命令即可。

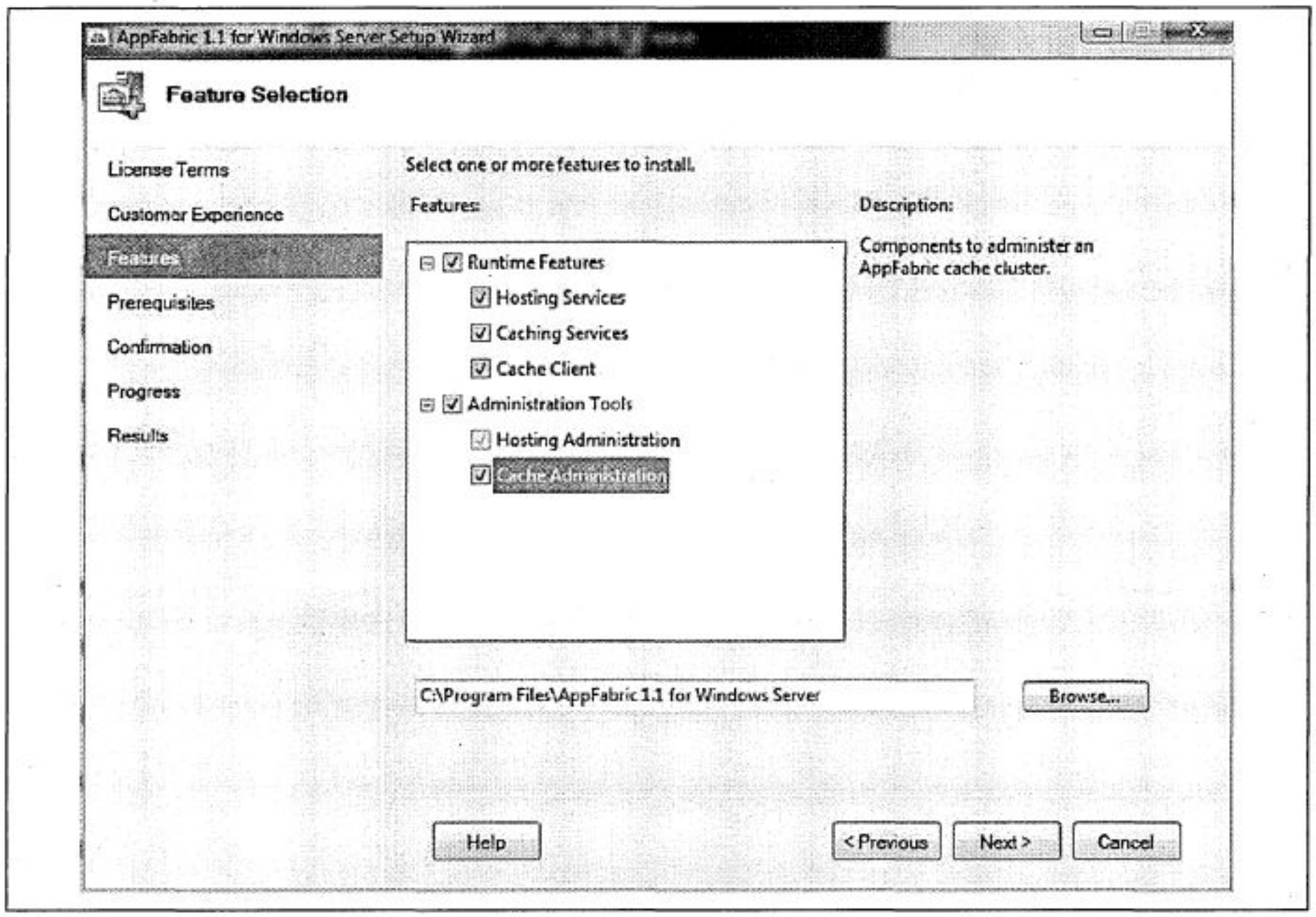


图12-1 Windows AppFabric安装过程中的功能选择界面

Windows AppFabric一旦安装完成，就会显示安装向导，向导会帮助我们完成剩下的步骤，从Velocity配置信息开始。

译注4：Microsoft Distributed Caching Service（代号为“Velocity”）提供了一种在内存缓存中存储数据的解决方案，以供将来检索，消除了从磁盘或数据存储器获取数据的需求。这可以显著提高应用程序的性能。此外，Velocity支持服务群集，这可以提高应用程序的可伸缩性。详细介绍见MSDN *http://msdn.microsoft.com/zh-cn/library/ee851751.aspx*

作为设置数据库参数Demo，我们先要选择数据库选项，然后点击“下一步”，如图12-2所示。

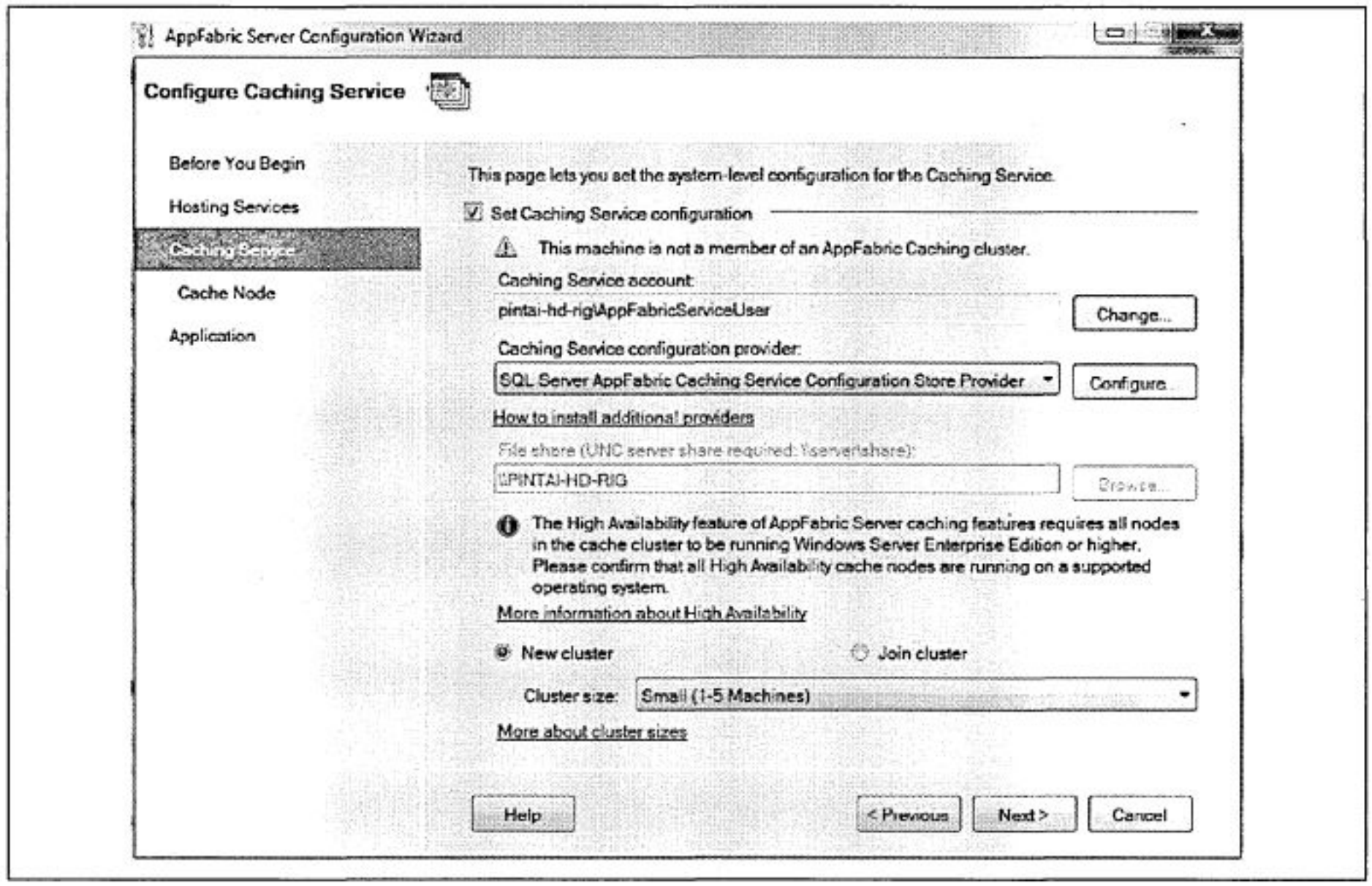


图12-2　配置数据库参数

使用PowerShell管理内存群集

接下来就是使用PowerShell来管理缓存了。此时，可以在应用程序菜单里看到“Caching Administration Windows PowerShell（缓存管理Windows PowerShell）”。使用控制台，可以管理缓存、检查活动状态以及创建新的缓存。在开始前，我们必须启动缓存集群，在PowerShell控制台里输入以下命令参数：

```
C:\> Start-CacheCluster
```

接下来，运行下面的命令授权用户像客户端一样可以访问缓存集群：

```
C:\> Grant-CacheAllowedClientAccount 'domain\username'
```

要验证我们的账号是否已经授权，只需使用Get-CacheAllowedClientAccounts命令即可。

*如果想看所有的缓存命令，则可以使用get-command *cache*命令。*

使用缓存

缓存可以在web.config配置文件或者代码里。下面是手动操作缓存的例子代码：

```
using Microsoft.ApplicationServer.Caching;
using System.Collections.Generic;

public class CacheUtil
{
```

```
    private static DataCacheFactory _factory = null;
    private static DataCache _cache = null;

    public static DataCache GetCache()
    {
      if(_cache != null)
        return_cache;

      // 为一个缓存宿主定义数组
      List<DataCacheServerEndpoint>servers = new List<DataCacheServerEndpoint>(1);

      // 指定缓存宿主详细信息
      // 参数 1 为宿主名称
      // 参数 2 为缓存端口号
      servers.Add(new DataCacheServerEndpoint("mymachine", 22233));

      // 缓存配置
      DataCacheFactoryConfiguration configuration = new DataCacheFactoryConfiguration();

      // 设置缓存宿主
      configuration.Servers= servers;

      // 本地缓存设置默认属性（关闭本地缓存）
      configuration.LocalCacheProperties= new DataCacheLocalCacheProperties();

      //关闭跟踪以避免在网页泄露信息
      DataCacheClientLogManager.ChangeLogLevel(System.Diagnostics.TraceLevel.Off);

      //给 cacheFactory 构造函数传递配置参数
      _factory = new DataCacheFactory(configuration);

      // 获取名为"default"的缓存
      _cache = _factory.GetCache("default");

      return_cache;
    }
  }
```

一旦缓存设置完毕，使用起来就非常简单了。下面是向缓存添加缓存对象数据的例子代码：

263

```
var cache = CacheUtil.GetCache();

cache.Add(orderid, order);
```

从缓存里查询数据也非常简单：

```
Order order= (Order)cache.Get(orderid);
```

更新现有的数据也非常简单：

```
cache.Put(orderid, order);
```

也可以使用AppFabric缓存来代替默认的缓存机制。

下面是web.config配置文件的例子代码：

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
```

```
  <!--configSections必须是第一个元素-->
  <configSections>
    <!—必须读取<dataCacheClient>元素-->
    <section name="dataCacheClient"
      type="Microsoft.ApplicationServer.Caching.DataCacheClientSection,
        Microsoft.ApplicationServer.Caching.Core, Version=1.0.0.0,
        Culture=neutral, PublicKeyToken=31bf3856ad364e35"
      allowLocation="true"
      allowDefinition="Everywhere"/>
</configSections>

<!--缓存客户端-->
<dataCacheClient>
  <!--缓存宿主-->
  <hosts>
    <host
      name="CacheServer1"
      cachePort="22233"/>
  </hosts>
</dataCacheClient>

<system.web>
  <sessionStatemode="Custom" customProvider="AppFabricCacheSessionStoreProvider">
    <providers>
      <!--指定会话数据的缓存-->
      <add
        name="AppFabricCacheSessionStoreProvider"
        type="Microsoft.ApplicationServer.Caching.DataCacheSessionStoreProvider"
        cacheName="NamedCache1"
        sharedId="SharedApp"/>
      </providers>
    </sessionState>
  </system.web>
</configuration>
```

正如我们看到的，从简单的缓存机制到取代复杂的ASP.NET缓存功能，AppFabric提供了完善的功能。我们可以在MSDN上学习更多关于AppFabric的概念、特性以及架构的知识。

264

客户端缓存技术

Client-Side Caching Techniques

浏览器通过从服务器获取HTML、数据以及其他资源，如CSS文件、图片、JavaScript文件、Cookie、Flash媒体等来为用户显示网页。但是，无论网速有多快，从用户硬盘直接取数据总比通过互联网从服务器获取数据快得多。

浏览器设计者考虑到了这个问题，他们知道如何使用本地缓存来尽量减少网络流量。过程很简单：每当访问网页，浏览器都会检查本地磁盘，看看是否有缓存页面需要的文件。如果没有，就会从服务器下载。若是第一次访问网页，就需要下载全部的文件。后续的访问，如果

浏览器已经有了网页的本地缓存文件（除非请求的网页不同或者资源不同），那么浏览器加载网站的速度会明显快很多。

浏览器缓存（译注5）的网页资源会放到事先定义的文件夹中。用户可以控制缓存文件的磁盘空间。浏览器需要负责清理陈旧的数据，而不需要用户输入信息。

接下来看如何使用客户端缓存或者浏览器缓存来加速网站访问。

理解浏览器缓存

浏览器缓存在本地的资源通过三个机制控制：新的、验证的和失效的（freshness, validation, invalidation）。这三者属于HTTP的一部分，并且定义在HTTP消息头里。

新的（Freshness）：允许应答消息直接使用，而不需要经过服务器检查，可以同时被服务器和客户端控制。例如，Expires过期的消息头设置了资源过期的时间，而Cache-Control: maxage则提供的是应答消息需要经过多久才再次刷新。

下面的代码展示了如何在服务端通过代码设置HTTP消息头：

```
public ActionResult CacheDemo()
{
        // 设置缓存控制头的值
        http://msdn.microsoft.com/en-us/library/system.web.httpcacheability(v=vs.110).aspx↵
        [HttpCacheability]

        //设置缓存控制：公开指定是否可以被客户端或者代理缓存
        Response.Cache.SetCacheability(HttpCacheability.Public);

        // 设置缓存控制最大时间为 20 分钟
        Response.Cache.SetMaxAge(DateTime.Now.AddMinutes(20));

        // 设置过期时间为 11:00 P.M
        Response.Cache.SetExpires(DateTime.Parse("11:00:00PM"));

        return View();
}
```

验证（Validation）：用来检测陈旧的缓存数据是否依然有效。例如，如果缓存的应答数据包含`Last-Modified`消息头，则缓存就可以使用`If-Modified-Since`消息头来检查内容是否更新过。这是相当弱的验证形式。但是，可以借助ETag（实体标签）机制实现更强的验证机制。

下面的代码演示了这两种方法：

译注5：浏览器缓存机制有利有弊，虽然一方面可以提升网站加载速度，但是另外一方面可能导致网站样式更新失效。网站开发人员都知道，尤其在调试网站样式时，可能出现本地浏览器无法立即显示新样式的情况。这个问题就和浏览器缓存有关系。当然这个问题也可以通过一些方法解决。

```
public ActionResult CacheDemo()
{
        // 设置 Last-Modified HTTP 消息头
        Response.Cache.SetLastModified(DateTime.Parse("1/1/2012 00:00:01AM"));

        //.设置 ETag HTTP 消息头
        Response.Cache.SetETag("\"someuniquestring:version\"");

        return View();
}
```

失效（Invalidation）通常是指另一个缓存请求的错误请求。例如，如果URL相关的请求应答结果返回的是POST、PUT或者DELETE请求，这个缓存应答结果就是失效的。

虽然提升页面加载速度是很棒的特性，但是基于浏览器的缓存存在固有的问题，如Bug、安全问题以及缺少对缓存数据的细粒度控制等，给Web开发人员带来了很多挑战。此外，当某些缓存数据改变时，我们需要让本地的缓存数据失效。这通常需要我们自己来特别实现某些捣乱的代码（这种工作也称为“黑客”）。

为了解决这些问题，新的HTML 5规范为开发者提供了新的技术以及更细粒度的客户端缓存控制技术。

AppCache缓存

HTML 5规范定义了应用程序缓存（或AppCache）API来允许开发人员直接访问本地浏览器缓存。

为了在网站里支持AppCache，必须完成以下三步：

1. 定义清单；
2. 引用清单；
3. 发送清单给用户。

让我们来详细看看每步的具体实现过程。

定义清单

定义清单的操作非常简单，只需要先创建一个txt文本文件并修改扩展名为.manifest即可：

```
CACHE MANIFEST

# version 0.1

home.html
site.css
application.js
logo.jpg
```

这是一个简单的清单，告诉浏览器应该缓存的4个文件。第一行必须包含“CACHE MANIFEST”字符。

下面是更加复杂的例子演示代码，有更好的细粒度控制缓存清单：

```
CACHE MANIFEST
# Generated on 04-23-2012:v2

# Cached entries.
CACHE:
/favicon.ico
home.html
site.css
images/logo.jpg
scripts/application.js

# Resources that are "always" fetched from the server
NETWORK:
login.asmx

# Serve index.html (static version of home page) if /Home/Index is inaccessible
# Serve offline.jpg in place of all images in images/ folder
# Serve appOffline.html in place of all other routes
FALLBACK:
/Home/Index /index.html
images/ images/offline.jpg
* /appOffline.html
```

可以看到清单文件里使用了一些基本的惯例：

- 以#作为开始符的是注释语句。
- CACHE部分包含的是第一次访问网站要缓存的文件列表。
- NETWORK部分列举的是浏览器每次都要重新获取的文件。换句话说，这些文件从来不会缓存在本地。
- FALLBACK部分定义找不到对应的文件时应该使用的文件。这是可选的，不是必须的。

下一步就告诉浏览器网站里的缓存清单文件。

引用清单

要引用清单文件，只需要在<html>标签里设置manifest属性即可。代码如下：

```
<!DOCTYPE html>
<html manifest="site.manifest">
...
</html>
```

当浏览器看到manifest标记属性以后，它就会知道网站定义了缓存清单，然后就会自动从网站下载这些文件。

正确发送清单文件

使用清单文件的关键就是如何使用正确的MIME类型（“text/cachemanifest”）：

```
Response.ContentType= "text/cache-manifest";
```

如果没有指定MIME类型，浏览器就无法识别清单文件，而且网站无法使用AppCache。

如果网站启用了AppCache缓存，浏览器只会在以下三种情况下从服务器获取文件：

1. 用户清除了缓存，也就是清除了所有缓存的内容。
2. 服务器修改了清单文件。简单修改注释或者保证文件都会被当做更新文件。
3. 通过JavaScript更新缓存时。

正如我们看到的，AppCache给了我们完全控制缓存内容的权利，允许我们控制何时更新缓存文件，而不需要借助黑客方法。

接下来将会介绍HTML 5规范的另外一个新特性。与AppCache略有不同，这个新特性也允许我们在浏览器上缓存数据。

本地存储

HTML 5规范引入的另外一个新特性就是支持离线、基于浏览器的存储机制，叫**本地存储**。可以把本地存储当做不会被限制大小的“超级cookie”：它允许在用户设备上保存大数据文件。

本地存储API由两个管理本地数据存储的终结点组成：`localStorage`和`sessionStorage`。虽然`localStorage`和`sessionStorage`都暴露相似的方法，但是两者最大的不同就是存储在本地存储里的数据是无限期有效的，而`sessionStorage`里的数据只对当前的浏览器有效，即只针对当前的用户会话有效。

与许多的服务端缓存对象机制一样，本地存储也使用了一种基于字符串的字典数据结构。所以，如果要想查询非字符串数据，就需要进行类型转换，比如调用`parseInt()`或者`parseFloat()`来转换基本的JavaScript数据类型。

为了在本地存储里存储数据，可以使用`setItem()`方法：

```
localStorage.setItem("userName", "john");
localStorage.setItem("age", 32);
```

或者使用方括号语法：

```
localStorage[userName"] = "john";
localStorage["age"] = 32;
```

当然也可以直接获取数据：

```
var userName= localStorage.getItem("userName");
var age = parseInt(localStorage.getItem("age"));

// 或者使用方括号

var userName= localStorage["userName"];
var age = parseInt(localStorage["age"]);
```

可以使用`removeItem()`来删除任何本地存储的数据：

```
localStorage.removeItem("userName");
```

或者使用`clear()`方法一次性清除所有数据：

```
localStorage.clear();
```

分配给本地存储的内存不是无限的，因此对存储数据的大小有限制，在HTML 5规范草案里，这个大小是5MB。浏览器可以申请更大的存储空间，其间会弹出一个命令窗口给用户。用户可以通过命令窗口来控制是否要增加存储空间。

本地存储提供了探测剩余空间大小的API，以及申请存储空间的API。`localStorage.remainingSpace()`可以显示剩余的磁盘空间大小，单位是字节。当存储的数据超出了限制以后，浏览器就会抛出`QuotaExceededError`异常，或者要求用户分配更多的存储空间。

最新的浏览器都支持本地存储新特性：无论怎样，优先判断浏览器是否支持这个特性后再使用它，这是一条很好的实践原则。下面的代码展示了如何检查浏览器是否支持本地存储功能：

```
function IsLocalStorageSupported() {
  try{
    return'localStorage' in window &&window['localStorage'] !== null;
  } catch (e) {
    return false;
  }
}
```

总结
Summary

缓存是构建高伸缩性和高性能应用的非常重要的技术。本章介绍了几种缓存技巧，并且讨论了各种缓存技术的实际使用场景。除了内置的缓存技术如`HttpContext.Application`、`HttpContext.Session`和`OutputCache`外，还可以使用分布式缓存技术来达到更高效的目标。甜甜圈缓存和甜甜圈洞缓存提供了不同于传统缓冲技术的解决方案，而且在某些场景中非常高效。我们可以根据实际项目需要选择合适的缓存方案。

缓存并非只限制于服务端，对HTML 5的新特性，也可以在客户端扩展使用一些缓存机制。HTML 5规范定义了两个新的、非常灵活的机制来支持客户端缓存或者浏览器缓存，用户可以直接控制这些缓存数据。

此外，HTML 5规范提供了客户端存储机制可以让我们更好地利用离线功能。我们可以缓存必须的应用程序数据而不需要实时连接互联网。

第13章 客户端优化技术

Client-Side Optimization Techniques

页面优化的最终目标都是尽可能快地加载页面。加载得越快，网站响应就越快，用户就越开心。本章介绍的就是最经典的页面加速与优化技术。

本章关注几种最重要、最基本的页面加速与优化技术。事实上，没有“万能的银弹”可以一劳永逸地提速网站，但是若遵循这些规则，就可以帮助改善网站设计，并显著提高页面加载速度。

这里介绍的大部分技术并不需要我们重写大量的代码，只需要在设计良好的Web应用中使用它们就可以了。

为什么要进行页面性能优化？在有限的带宽条件下，页面加载越迅速，意味着加载时间更短。如果网站用户的网络非常差、网速很慢，那么快速加载就可以让用户更快地看到网页内容。

页面剖析

Anatomy of a Page

为了更好地理解影响页面加载速度的因素，有必要来看浏览器是如何渲染网页的。

网页通常由HTML代码、JavaScript文件以及CSS样式文件组成，有些网页还包含图片以及多媒体文件，如Flash、Silverlight对象或者音频、视频文件。

浏览器渲染网页遵循从上到下的方式：从页面顶部的HTML标签开始，依次下载需要的资源文件，如图13-1所示。页面只有在下载完所有的资源以后才会渲染出来。换句话说，即使页面已经下载了所有的HTML代码，用户依然只能看到空白页面，只有浏览器下载完所有的资源（如图片、CSS样式文件以及JavaScript文件）才会显示整个页面。

总结一下：

- 资源越少，加载时间越短；
- 调整页面资源可以影响页面或者部分页面的显示时间。

HttpRequest剖析

更多的请求会导致页面速度变慢。为什么呢？先来看一下当请求资源时什么会影响下载时间。具体分为以下几步：

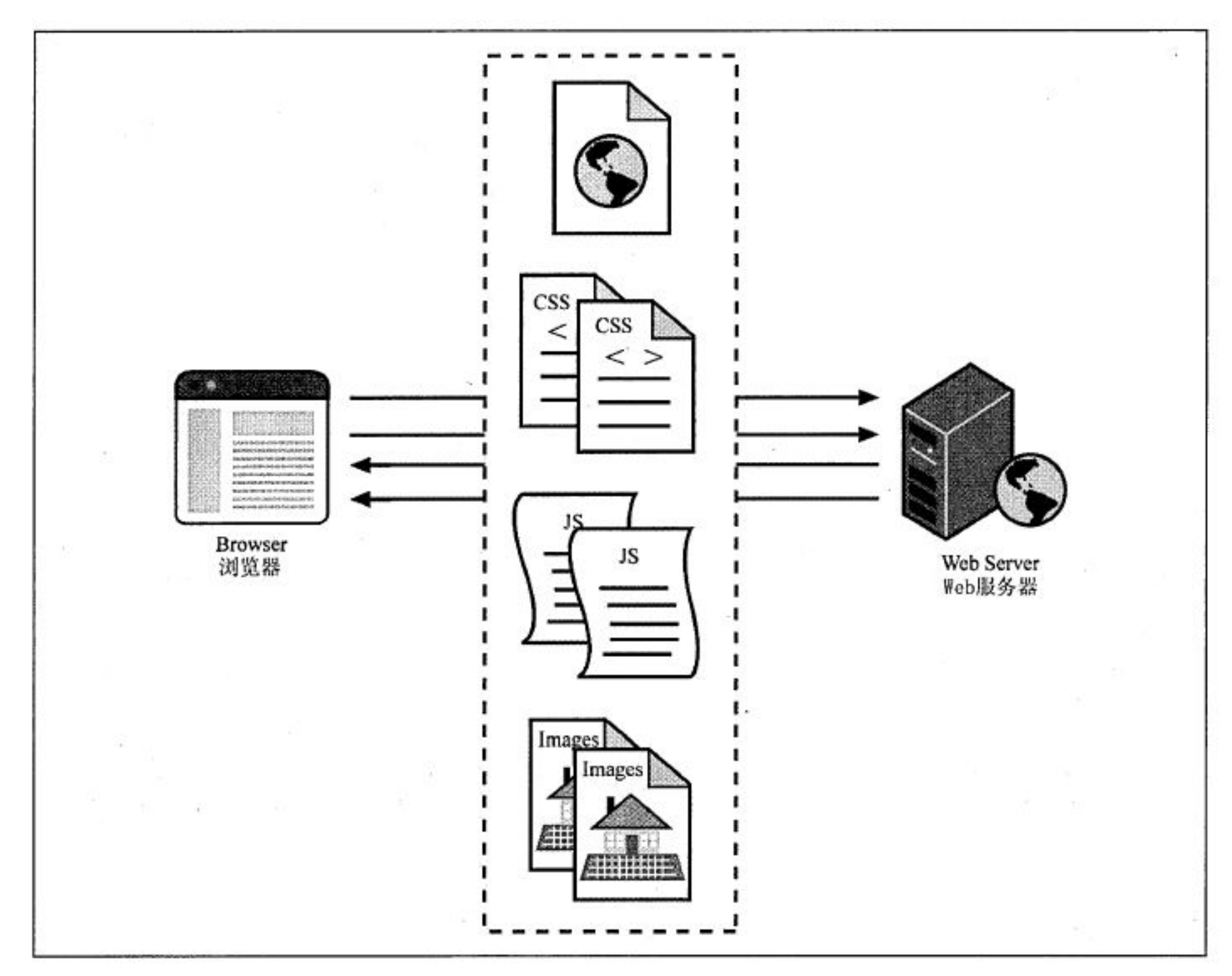


图13-1　页面剖析

1. DNS查询。解析请求的域名地址：首先，浏览器或者客户端发送一个DNS请求给本地ISP DNS服务器；然后，DNS服务器返回域名对应的IP地址。
2. 连接。客户端会和该IP地址的服务器建立TCP连接。
3. 初始化`HttpRequest`请求。浏览器会向网站服务器（如IIS）发送HTTP请求。
4. 等待。浏览器会等待Web服务器响应请求消息：
   - 服务端，Web服务器处理请求，包括查找资源以及回发应答消息给客户端。
   - 浏览器收到Web服务器的第一个包的第一个字节，它包含HTTP应答消息头和内容。
5. 加载。浏览器加载应答消息的内容。
6. 关闭。在接收完最后一个字节之后，浏览器随后会请求服务器关闭连接。

这些步骤对每个请求都是重复的，过程基本一样，如图13-2所示，但浏览器缓存的页面资源除外。如果请求的资源在浏览器缓存（译注1）里，浏览器就会直接从缓存里读取而不需要到服务器下载了。一旦浏览器从服务器端下载完资源，它就会试图把这些资源缓存在本地。

优化其中某些步骤就可以帮助提升页面加载速度。

译注1：浏览器缓存，其实是提升网站加载速度的一种方式，大部分浏览器都支持缓存网站资源，比如IE、Chrome、Firefox等。

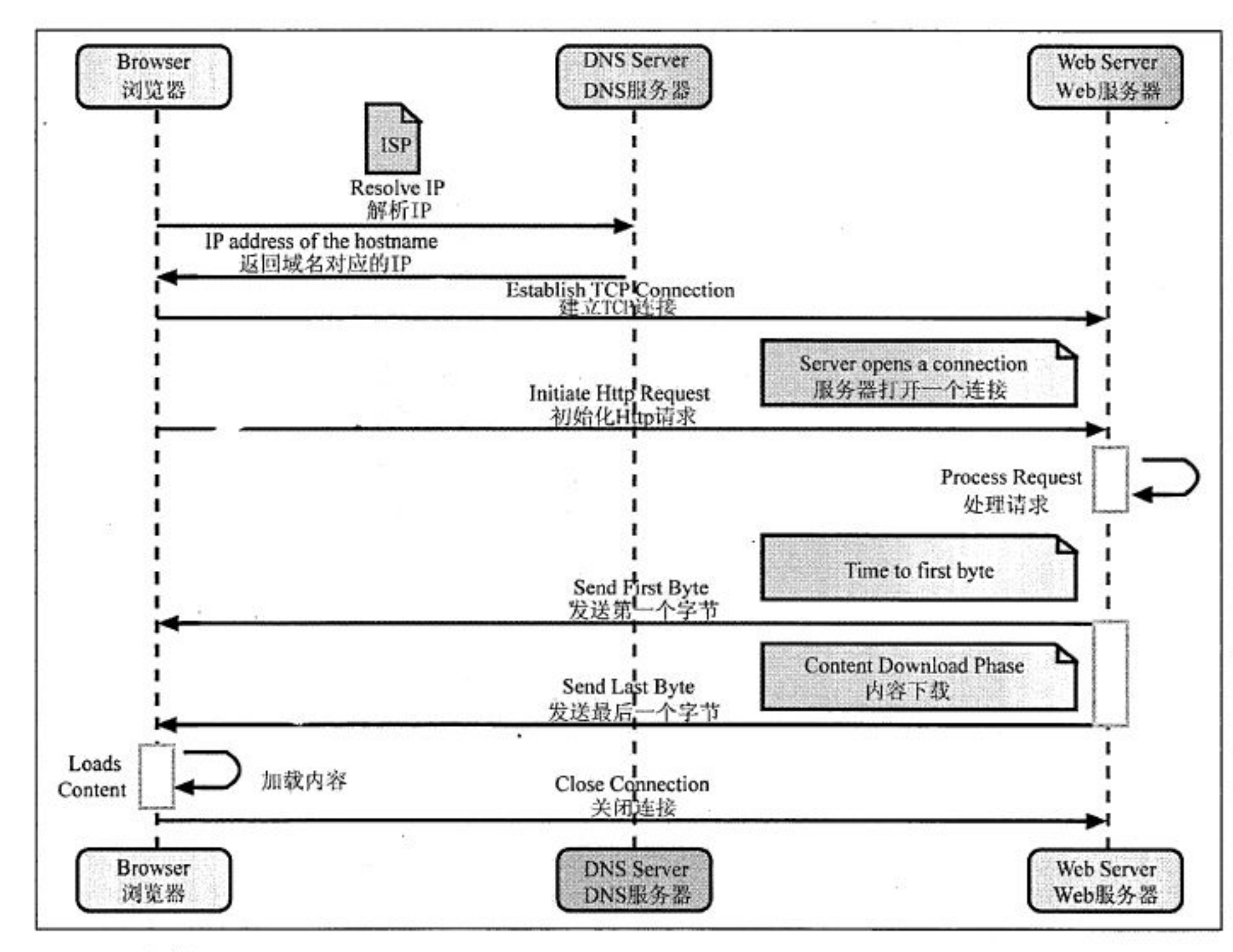


图13-2　剖析HttpRequest

最佳实践

Best Practices

雅虎卓越性能团队总结了35条帮助改进网页性能的最佳实践原则（译注2）。从最简单的13条原则开始，逐步扩展到7大类35条原则。完整的列表可以在该团队的开发者博客中阅读。谷歌同样提供了自己的网站性能优化原则（译注3）。在谷歌的推荐文档里，分为6大类30条原则。下面几节将会介绍其中的几条可以显著提升网站性能的设计原则。

274

减少HTTP请求

用户80%的响应时间都花在前端上了，用于下载页面组件，比如图片、CSS样式文件、脚本、Flash等。减少组件的数量可以减少HTTP请求的次数，如图13-3所示，这是页面加速的关键所在。

译注2：雅虎性能优化最佳实践原则（Best Practices for Speeding Up Your Web Site）可参阅 *http://developer.yahoo. com/performance/rules.html/*。

译注3：谷歌网站性能优化最佳实践（Web Performance Best Practices）可参阅 *https://developers.google.com/ speed/docs/best-practices/rules_intro*。

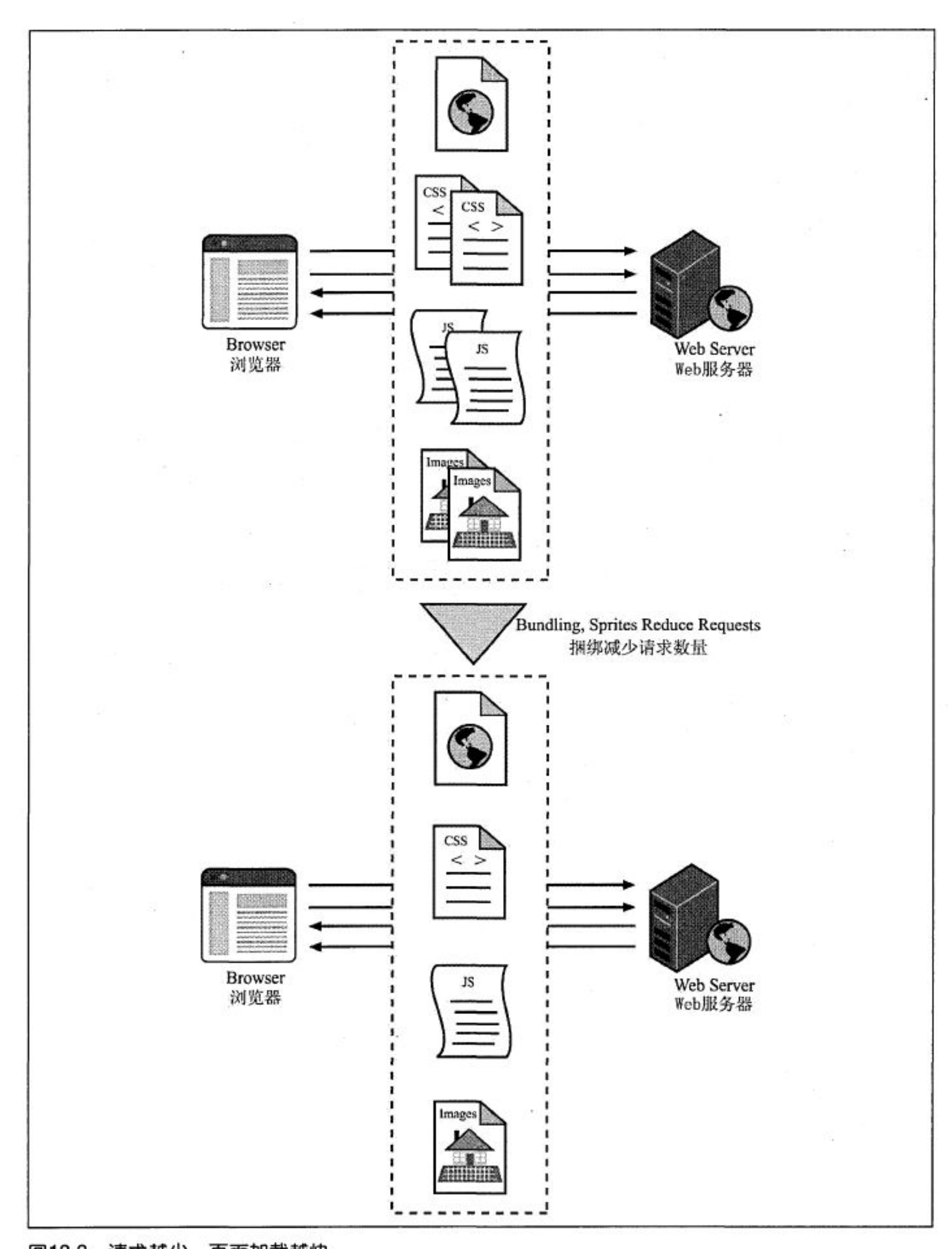


图13-3　请求越少，页面加载越快

可以重新设计页面来减少组件数量，或者使用外部资源链接（JavaScript文件、CSS样式文件、图片）来减少客户端下载的资源数量。ASP.NET MVC 4提供了捆绑（bundling）新特性，它

支持把多个JavaScript和CSS文件捆绑成单一文件，这就符合了减少下载资源数量的原则。本章后面“捆绑和压缩（Bundling and Minification）”一节中有详细的介绍。这种技术可以用于减少脚本和样式文件的请求数量，但是无法应用到图片上。如果页面包含很多图片，就考虑使用CSS Sprites来减少图片请求数量。另外一种技巧就是使用data: URLscheme在页面中嵌入图片或者内联（inline）图片，但是不是所有的主流浏览器都支持这种机制。

使用CDN内容分发网络

内容分发网络（CDN）是指网络中分布于不同位置的Web服务器向用户快速发送内容。Web服务器发送给特定用户的数据一般是优先选择临近节点：选择间隔网络跃点（hop）最少的服务器或者应答时间最短的服务器。切换CDN服务器非常简单，可以显著提升网站速度。

另外，可以通过服务器和CDN的多个子域名来最大限度地下载图片、CSS和脚本，因为浏览器限制了单个域名上的资源下载连接。通过多个域名分散映射资源，可以提高并发下载连接的数量，因为浏览器把每个子域名当做不同的域名对待，因此可以使用更多的连接，如图13-4所示。

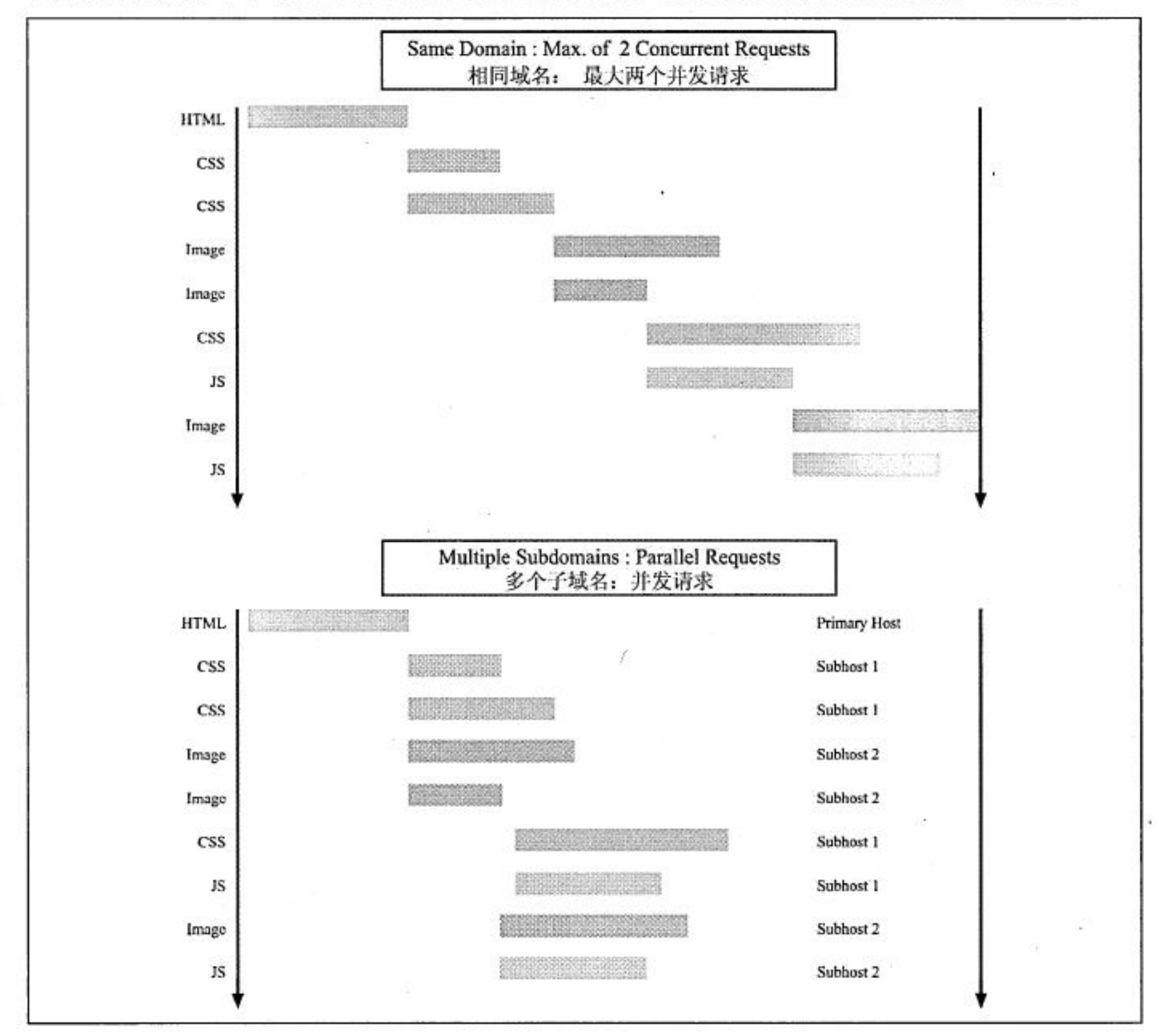


图13-4 使用CDN内容分发网络

添加Expires或Cache-Control消息头

研究结果（*http://yuiblog.com/blog/2007/01/04/performance-research-part-2/*）表明，40%~60%的日常访问者没有缓存。上面介绍的技术（减少HTTP请求、使用CDN）可帮助提升第一次访问速度，而通过在客户端启用缓存，使用Expires或Cache-Control可以加速后续页面的访问速度。

浏览器（代理）通过使用缓存来减少HTTP请求消息的数量和大小，提升页面加载速度。Web服务器在HTTP应答消息里使用Expires消息头来告诉客户端数据可以缓存多长时间。下面的代码就是通过Expires消息头告诉浏览器：这个应答消息应该在2013年5月20日过期：

```
Expires: Wed, 20 May 2013 20:00:00 GMT
```

这可以通过在IIS里添加Expires和Cache-Control的值或通过ASP.NET MVC编程方式来实现。

在IIS里设置客户端缓存

IIS 7允许我们使用<staticContent>元素的<clientCache>来设置客户端缓存消息头。

httpExpires标记属性可以用于添加HTTP Expires过期消息头，指定内容过期的时间。Cache-Control消息头可以使用cacheControlMaxAge标记属性添加（注意，这个行为取决于cacheControlMode标记属性）。

通过ASP.NET MVC设置客户端缓存

也可以通过编程方式在ASP.NET MVC中使用Cache.SetExpires()和Cache.SetMaxAge()方法分别设置Expires和Cache-Control消息头。

以下为一个简单实例：

```
// 设置 Cache-Control：最大 1 年
Response.Cache.SetMaxAge(DateTime.Now.AddYears(1));
// 设置 Expires：本地时间 11:00 P.M
Response.Cache.SetExpires(DateTime.Parse("11:00:00PM"));
```

我们已在本章前面“理解浏览器缓存（Understanding the Browser Cache）”一节中学习过缓存控制Cache-Control：max-age和Expires消息头。

注意：同时指定Expires和Cache-Control消息头是多余的——只需要为每个资源指定一个属性即可。

缓存清理

如果使用了过期消息头，修改内容时就必须通知浏览器。如果通知失败，浏览器会继续使用旧的缓存文件。

因为浏览器会根据URL来缓存内容，所以必须通过修改URL地址来更新缓存。通常我们会在查询参数后面添加版本参数（译注4）。实际上，ASP.NET MVC捆绑和压缩新特性提供了内置的“缓存清理（cachebusting）”功能，这项功能会在组件发生变化时自动清理缓存。

译注4：很多网站会在JS和CSS文件后面加上版本参数，例如，CSS和JavaScript文件引用：

```
<link href="css/54peixun.css?version=1" rel="stylesheet" type="text/css" />
<scriptsrc="scripts/54peixun.js?version=1" type=text/javascript></script>
```

可以在网站更新资源文件版本时重新下载新的资源文件。

Expires消息头只有在用户访问了网站之后才会影响页面视图。当用户第一次访问网站和浏览器的缓存为空时，它对HTTP请求的数量没有影响。因此，当浏览器已经包含所有的组件资源时，页面改进的性能及影响取决于用户多久使用“原始”缓存访问网站。

GZip组件

压缩文本内容，比如HTML、JavaScript、CSS和JSON数据，可以减少网络传输的时间，因此压缩机制显著改善了网站响应时间。

使用GZip压缩组件通常情况下可以减少70%的时间。旧的浏览器和代理服务器可能不支持压缩机制，因为它们无法在客户端和服务端很好地支持压缩协议，也可能服务端或者客户端无法理解压缩数据。这些是个例，虽然有时候会出现这种问题，但是这种情况比较少见，主流的浏览器和Web服务器基本都支持压缩机制。

一些代理服务器和杀毒软件会从HTTP请求消息中删除Accept-Encoding: gzip、deflate消息头，这种情况会导致服务器返回未压缩的数据。但是这无伤大雅，它只是导致没有效果但并不会降低性能。

Web服务器可以配置根据文件或者MIME类型来支持压缩内容，或者动态配置支持压缩内容。虽然对基于文本的内容进行压缩非常有意义，但是对二进制数据，比如图片、音频、视频和PDF等已经压缩了的数据进行压缩会导致性能下降。再压缩这些数据会导致文件增大。

尽可能多地压缩文件是减少文件大小的简单方式，这样可以加速页面加载，改善用户体验。

在IIS 7中，<httpCompression>元素可以指定HTTP压缩设置。默认IIS 7启用压缩机制，安装了性能模块（performance module）。

下面的代码展示了ApplicationHost.config文件里的默认配置：

```
<httpCompression
      directory="%SystemDrive%\inetpub\temp\IIS Temporary Compressed Files">
   <scheme name="gzip" dll="%Windir%\system32\inetsrv\gzip.dll" />
   <dynamicTypes>
      <add mimeType="text/*" enabled="true" />
      <add mimeType="message/*" enabled="true" />
      <add mimeType="application/javascript" enabled="true" />
      <add mimeType="*/*" enabled="false" />
   </dynamicTypes>
   <staticTypes>
      <add mimeType="text/*" enabled="true" />
      <add mimeType="message/*" enabled="true" />
      <add mimeType="application/javascript" enabled="true" />
      <add mimeType="*/*" enabled="false" />
   </staticTypes>
</httpCompression>
```


我们可以在web.config文件中<system.webserver>节点下的<httpCompression>元素重新设置网站的压缩功能。

注意：动态压缩会增加CPU消耗，因为它要为每个请求执行压缩操作。结果无法缓存，因为每次都要重新生成结果，数据本身是动态的。

如果动态内容相对是静态的（换句话说，不是每次请求都会变化），我们就可以在<urlCompression>节点下通过设置dynamicCompression- BeforeCache属性来缓存这些数据。

置顶样式文件

置顶CSS样式文件可以允许逐步渲染页面文件。因为我们关注的是性能，希望页面逐步加载，因此，希望页面尽快显示所有内容。这一点对于包含很多内容的页面和网速缓慢的用户来说非常重要。

在我们的例子中，HTML页面就是进度指示器！当浏览器逐步加载页面时，页面头、导航栏、Logo都位于页面顶部，其他页面元素都作为视觉反馈随着页面加载显示给用户。

如果把CSS样式文件放在页面底部就会产生问题，可能阻止浏览器包括IE逐步渲染页面内容。CSS样式文件发生变化，就会导致整个页面元素重新渲染，这就意味着用户浏览器会停滞在空白页面上。

置底脚本文件

脚本块并行下载。HTTP/1.1规范建议浏览器不要从单个主机名同时下载多个文件（新版本浏览器支持并行下载机制）。如果把网站图片放在多个主机服务器上，就可以并行下载多种资源。当下载脚本文件时，浏览器不会下载其他的资源，即使是在不同的主机服务器上，如图13-5所示。

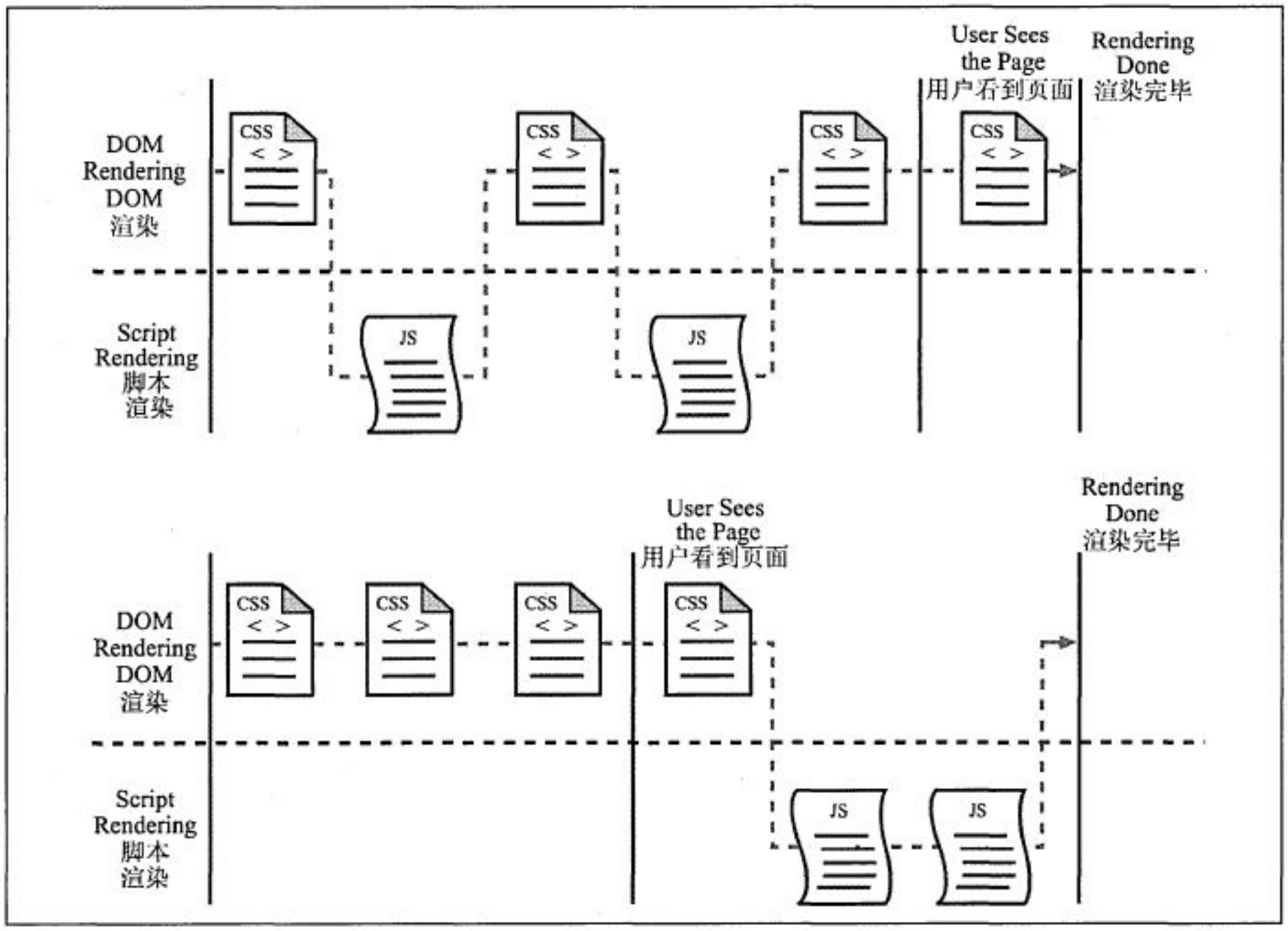


图13-5　脚本置底对页面渲染的影响

很多情况下无法把脚本置底。例如，页面如果是使用document.write向页面插入新内容，就无法放到页面下方。不过，在许多情况下，有种方法可以解决这种情况。我们会在本章后面的内容里介绍这些方法。

推迟脚本执行

可以使用<script>的DEFER属性来推迟执行页面里的脚本代码。DEFER属性可以告诉浏览器跳过当前代码继续渲染。但是，不同的浏览器处理此属性的方式不同，所以导致这种方法相当不靠谱。

如果可以推迟一个脚本执行，那么它也可以移到页面底部，以便更快地呈现网页。

延迟加载脚本

某些网站，例如，Gmail，使用了延迟加载技术来渲染评论里的JavaScript代码。浏览器会自动忽略这些脚本来渲染页面。当需要脚本（用户操作）时，就可以通过评论标签来访问脚本块，使用eval()来解析JavaScript代码。虽然用法不够优雅，但是这种方法确实比脚本置底或者延迟执行脚本更加有益。

281

迁出脚本和样式代码

把脚本文件和CSS样式文件放在外部文件里的方法与传统的代码内置到页面里的方式不同，前者允许浏览器缓存这些文件。可以让后续页面加载得更快，如图13-6所示。

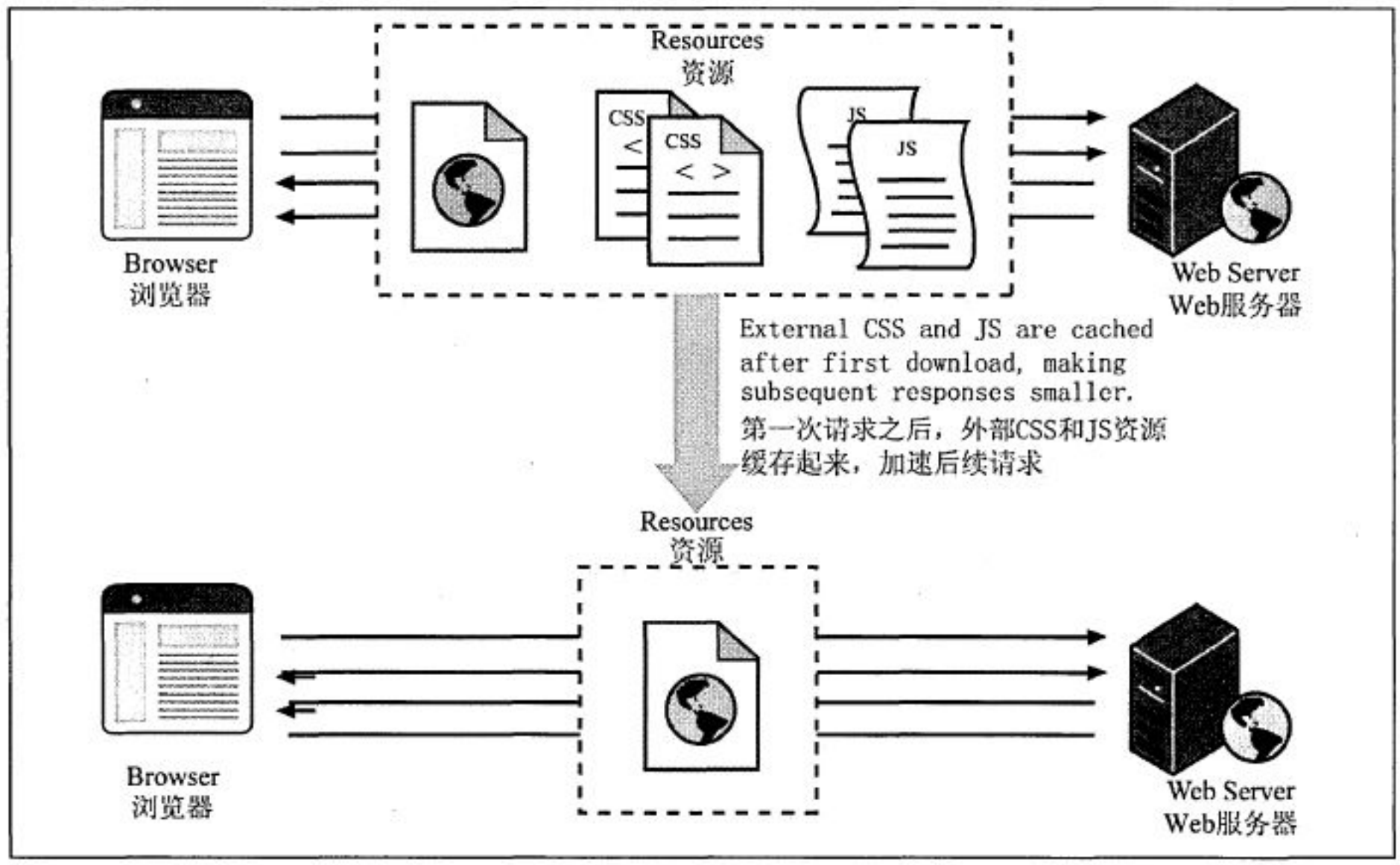


图13-6　外部CSS样式文件和JavaScript文件

页面内置的JavaScript代码和CSS样式代码无法缓存，还增加了页面数据大小。不过这种做法的好处是减少了HTTP请求，每次加载页面都不需要去获取外部的JS或者CSS文件。

什么可以带来更大的好处（减少请求数量或者使用外部文件存储并可以缓存）？对于不同的应用来说答案也各不相同。虽然好处难以用准确的数据量化，但是可以在一定程度上估算一下。例如，如果网站要在不同的页面上重用这些资源，显然使用外部文件更好；但是，如果网站页面很少，或者每个页面使用不同的资源，或许采用页面内嵌代码方式更佳。

282

一个折中的办法就是结合页面内联代码和动态加载外部资源（使用如AJAX的异步加载技术或者<script>标签的async属性）。这样可以让页面开始时快速加载，后续的其他页面也可以使用缓存资源。Yahoo（雅虎）的官方主页使用的就是这种方法。

减少DNS查询

DNS查询是指通过域名解析出服务器IP。通常，每个请求会花费20~120 ms的时间，在此期间，浏览器无法执行任何其他任务，造成浏览器阻塞。

减少HTTP请求就可以尽量减少这种问题，但是网页上还有一些资源必须使用外部请求（例如，图片、CSS样式文件和JavaScript文件）。减少页面内主机域名的数量，可以优化这个请求过程。因为浏览器根据DNS查询来缓存结果，也就是缓存资源和域名，如图13-7所示。

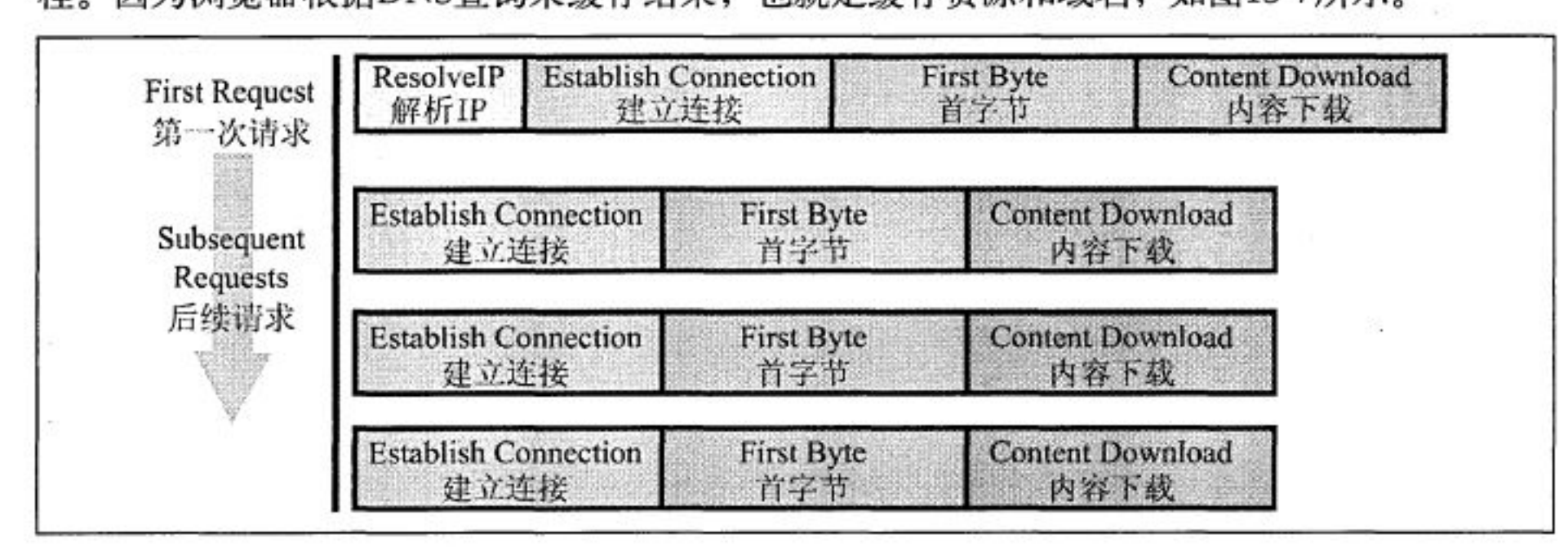


图13-7　减少DNS查询以提升性能

减少主机名可能导致无法并行下载资源。因此，要在主机名数量和并行下载的子域名之间进行权衡。Yahoo（雅虎）开发团队推荐的最佳平衡方案是把这些资源分成2、3或者4个主机。

压缩JavaScript和CSS

压缩（Minification）是指从代码中移除不必要的字符以减少数据大小，从而节约传输时间。当代码压缩后，所有的评论、不需要的空白字符（空格、回车和Tab）都会被删除，如图13-8所示。对JavaScript代码，这样做可以改善应答时间，因为下的文件变小了。两种最流行的JavaScript代码压缩工具是JSMin和YUI Compressor。YUI Compressor也可以压缩CSS文件。

283

模糊处理（Obfuscation）属于另外一种代码优化方法。在美国前10名的网站调查中，压缩和模糊处理的数据减少比例是21%:25%。虽然模糊处理可以更高效地压缩数据，但是压缩JavaScript文件方式的风险更低。压缩代码比模糊处理简单得多，模糊处理的方法更加复杂，

而且代码生成过程中可能产生一些Bug。

除了压缩JavaScript脚本文件和CSS样式文件外，还可以压缩页面里内嵌的<script>和<style>代码块。因为就算使用了GZip，压缩脚本仍然可以减少5%以上的数据规模。随着JavaScript和CSS样式文件的增加，压缩代码的优势更加明显。

```
/* http://meyerweb.com/eric/tools/css/reset/
   v2.0 | 20110126
   License: none (public domain)
*/

html, body, div, span, applet, object, iframe,
h1, h2, h3, h4, h5, h6, p, blockquote, pre,
a, abbr, acronym, address, big, cite, code,
del, dfn, em, img, ins, kbd, q, s, samp,
small, strike, strong, sub, sup, tt, var,
b, u, i, center,
dl, dt, dd, ol, ul, li,
fieldset, form, label, legend,
table, caption, tbody, tfoot, thead, tr, th, td,
article, aside, canvas, details, embed,
figure, figcaption, footer, header, hgroup,
menu, nav, output, ruby, section, summary,
time, mark, audio, video {
        margin: 0;
        padding: 0;
        border: 0;
        font-size: 100%;
        font: inherit;
        vertical-align: baseline;
}
/* HTML5 display-role reset for older browsers */
article, aside, details, figcaption, figure,
footer, header, hgroup, menu, nav, section {
        display: block;
}
body {
        line-height: 1;
}
ol, ul {
        list-style: none;
}
blockquote, q {
        quotes: none;
}
blockquote:before, blockquote:after,
q:before, q:after {
        content: '';
        content: none;
}
table {
        border-collapse: collapse;
        border-spacing: 0;
}
```

minified
压缩

```
html,body,div,span,applet,object
,iframe,h1,h2,h3,h4,h5,h6,p,bloc
kquote,pre,a,abbr,acronym,addres
s,big,cite,code,del,dfn,em,img,i
ns,kbd,q,s,samp,small,strike,str
ong,sub,sup,tt,var,b,u,i,center,
dl,dt,dd,ol,ul,li,fieldset,form,
label,legend,table,caption,tbody
,tfoot,thead,tr,th,td,article,as
ide,canvas,details,embed,figure,
figcaption,footer,header,hgroup,
menu,nav,output,ruby,section,sum
mary,time,mark,audio,video{margi
n:0;padding:0;border:0;font-
size:100%;font:inherit;vertical-
align:baseline}article,aside,det
ails,figcaption,figure,footer,he
ader,hgroup,menu,nav,section{dis
play:block}body{line-
height:1}ol,ul{list-
style:none}blockquote,q{quotes:n
one}blockquote:before,blockquote
:after,q:before,q:after{content:
'';content:none}table{border-
collapse:collapse;border-
spacing:0}
```

图13-8　压缩脚本和样式文件的结果

避免重定向

请求重定向发生在当用户浏览器打开的URL和请求的地址不同时。我们可以使用HTTP的状态码301和302实现。

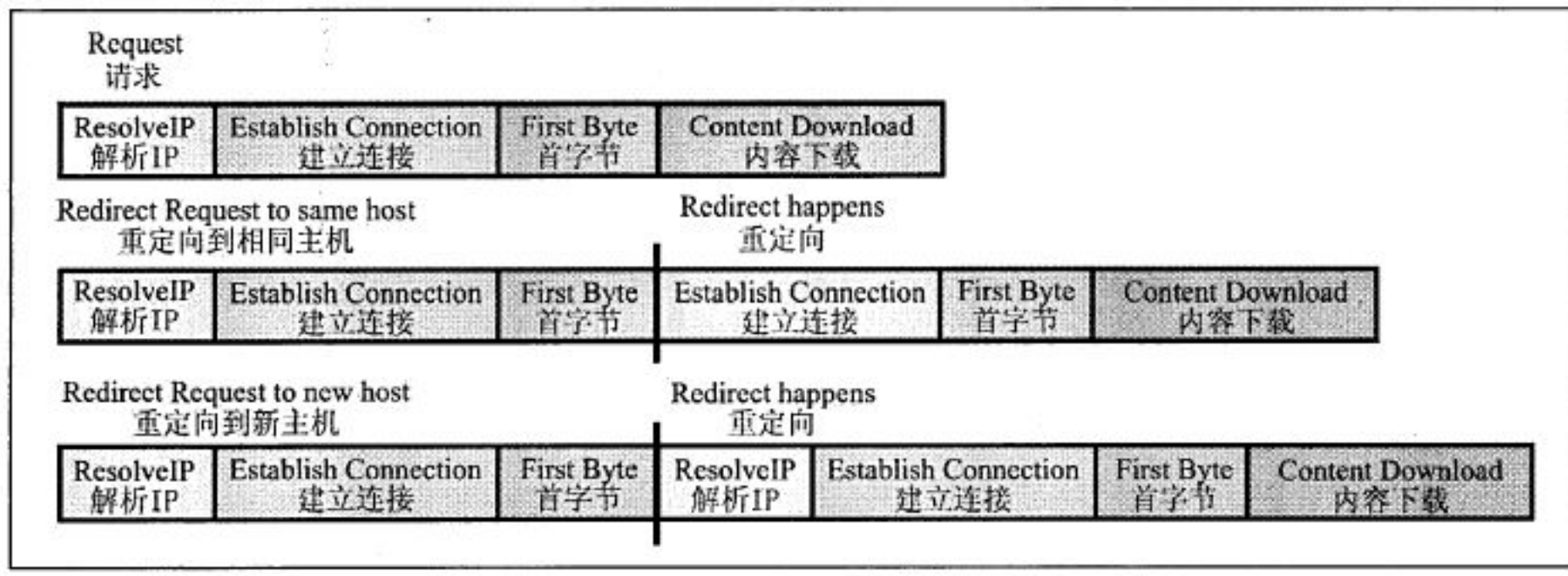
重定向的性能很低，因为结果通常无法缓存（除非明确使用Expires或Cache-Control消息头控制缓存），而且会引起第二次请求。图13-9展示了什么情况下会发生重定向。

重定向对处理无效链接（译注5）（例如，有些旧的页面被删除了）、URL简化或者从多个域名重定向到单一域名（例如，*wikipedia.net*到*wikipedia.com*）时非常有用。

284

虽然以上介绍的请求重定向都是必要的，但是有些重定向，开发人员可能是无法察觉的。例如，在ASP.NET MVC网站中（或者其他IIS里托管的应用），调用*http://www.ebuy.biz/Home/About*地址，会引起重定向到*http://ebuy.biz/Home/About/*（后面加个“/”也一样）。

译注5：几乎所有的网站都在处理无效链接。URL简化编码，新浪微博的连接使用了这项功能，我们可以看到发到微博里的链接都会经过新浪服务器重新定向。多个域名跳转到单一域名，如现在的京东商城就是个最好的例子。当然，新青年IT网54peixun.com也使用了重定向机制。



13-9 大部分情况要避免重定向

这个Bug可以使用永久重定向解决“permanent”（HTTP状态码301）。

下面是301应答消息里的HTTP消息头代码：

```
HTTP/1.1 301 Moved Permanently
Location: http://yourhostname.com/Home/About
Content-Type: text/html
```

有很多方法处理这个问题，例如：

- 编写自己的`HttpModule`；
- 在控制器里处理重定向；
- IISRewrite模块是个HTTP模块，已经被植到IIS里了。我们可以在web.config文件里使用`system.webServer> rewrite`元素来配置IIS Rewrite模块。

下面展示了使用IIS Rewrite模块实现重定向的代码：

```
<rewrite>
  <rules>
    <!--删除 URL 尾部的 URL-->
    <rule name="Strip trailing slash" stopProcessing="true">
      <match url="(.*)/$" />
      <conditions>
        <add input="{REQUEST_FILENAME}" matchType="IsFile" negate="true" />
        <add input="{REQUEST_FILENAME}" matchType="IsDirectory" negate="true" />
      </conditions>
      <action type="Redirect" redirectType="Permanent" url="{R:1}" />
    </rule>
  </rules>
</rewrite>
```

上面的代码使用了正则表达式来删除请求尾部的斜线。如果存在，就会直接删除，以后永久重定向到不带尾部斜线的地址上。

删除重复脚本

如果页面两次引用同一个JavaScript文件，那么对页面的性能损伤会很大——这可能会超出你的想象。通过对美国前10名网站的调查发现，居然有两个网站包含重复的脚本代码。两个因素导致了在单个页面中重复引用代码：开发团队人数和脚本数量。重复引用脚本文件对性能的损伤会非常大。

重复的脚本会导致IE里不必要的HTTP请求（但是Firefox不会）。在IE里，如果页面引用了两次外部脚本而且没有缓存，就会在加载页面时产生两次HTTP请求。即使已经缓存了脚本文件，在用户重新加载页面时也会导致额外的HTTP请求。

除了导致额外的、重复的HTTP请求以外，浏览器多次评估脚本代码也会浪费大量时间。无论是否缓存过该脚本文件，冗余的脚本都在Firefox和IE里执行。

避免脚本重复的方法就是在网站中实现脚本管理模块。最典型的方法就是使用`<script>`标签在页面里引用外部脚本文件。

配置实体标签

实体标签（Etag）是指唯一标识资源或者组件版本的标识符，比如图片、CSS样式文件或者脚本。ETag技术是Web服务器和浏览器用来检测缓存文件是否和服务器原始文件版本一致的方法。

Etag技术提供了验证资源版本的灵活方法，而且比使用最后修改日期标识文件（这种方法不可靠，因为浏览器只知道从服务器获取资源而并不关注文件的修改时间，并且每个浏览器获取的方法也不太一样）更加可靠。

`ETag`应答消息头如下：

```
HTTP/1.1 200 OK
Last-Modified: Tue, 29 May 2012 00:00:00 GMT
ETag: "8e12af-3bd-632a2d18"
Content-Length: 14625
```

286

为了验证资源是否为最新版本，浏览器使用“`If-None-Match`”消息头来传输ETag给网站服务器。如果ETag一样，服务器就会返回304状态码（这个例子中，应答消息减少了14,625B）：

```
GET /images/logo.png HTTP/1.1
Host: yourhostname.com
If-Modified-Since: Tue, 29 May 2012 00:00:00 GMT
If-None-Match: "8e12af-3bd-632a2d18"
HTTP/1.1 304 Not Modified
```

如果网站使用了Web Farm（Web园）或者Web服务器群集，就要给每个服务器一样的ETag——否则，浏览器会把不同服务器上的资源当做不同的版本，不断重复下载，这样就违背了使用ETag的初衷。

不幸的是，Apache和IIS支持Etag的机制让我们很难在Web Farm里跨服务器使用相同的ETag。这样就导致无法针对Web Farm部署的情况来测试ETag。这种情况下，更简单的办法就是使用“`Last-Modified`”消息头来验证请求资源的版本。如果选择了这种方法，就应该从应答消息里删除ETag，这样可以减少HTTP请求和应答消息的大小。微软官方技术文档提供了从IIS删除ETag的详细步骤：*http://support.microsoft.com/?id=922733*。

注意：只需要设置`Last-Modified`和`ETag`中的一种。同时使用两种方法是多余的，完全没有必要。

测试客户端性能

Measuring Client-Side Performance

为了优化某些指标，首先要知道如何衡量它。如果未经检测或分析，就不能找出瓶颈，那么

也无法验证改进的效果。

有很多测量工具可供我们使用，最简单、最方便的工具应该就是YSlow（译注6）。YSlow可以在很多浏览器上使用。虽然后面的内容是以Firefox作为实例浏览器，但是也可以选择其他浏览器，因为这些核心的概念和技巧都是一样的。YSlow使用Yahoo开发团队推荐的网站性能优化35条规则中的第23条来测试网站。

开始前，要安装Firefox浏览器插件。一旦安装完毕，就可以创建一个简单的ASP.NET MVC网站，然后看看YSlow如何测试这个网站的指标。

创建方法为：在Visual Studio里选择“文件→新建项目→ASP.NET MVC 4应用”，然后选择互联网应用模板，如图13-10所示。

287

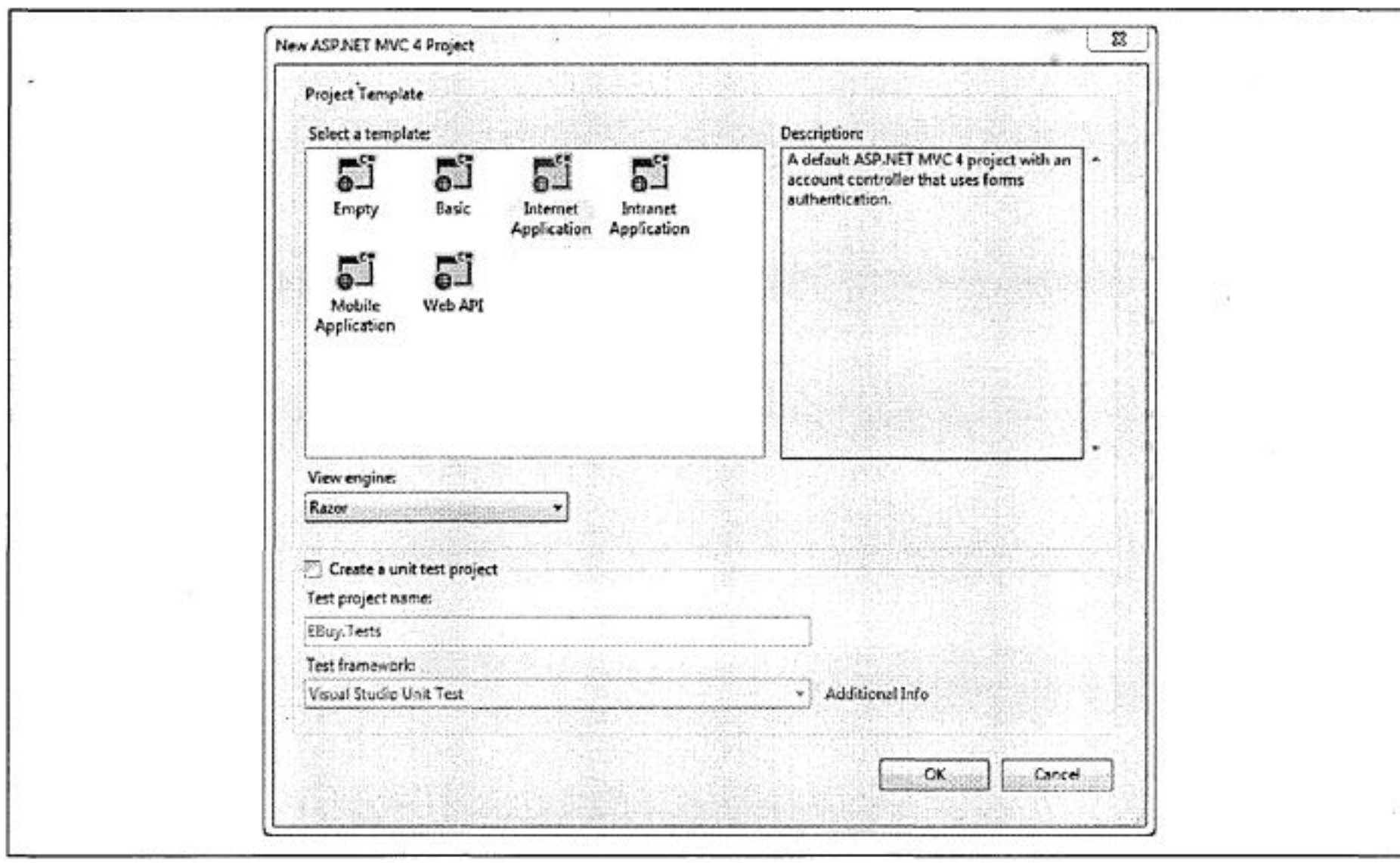


图13-10　创建一个简单的ASP.NET MVC 4网站

在创建过程中不需要编写任何代码，直接编译、运行网站即可。如果默认的浏览器不是Firefox，就要打开Firefox，然后输入默认的新网站URL。打开Firebug，然后选择图13-11所示YSlow选项卡，再点击“测试”按钮进行测试。

288

使用默认的测试规则(YSlow V2)，测试结果很不错。ASP.NET MVC 4的默认模板已经符合大部分页面性能优化的基本原则，在大部分测试指标里的得分都是“A”级，就像我们看到的图13-12所示的测试结果一样。页面总体得分，平均应该在“B”级。

在图13-12中，最差的得分是规则1#“Make fewer HTTP requests（减少HTTP请求。）”，测试报告给出的建议是立即改进——合并13个CSS样式文件。如果检查组件资源列表如图13-13

译注6：YSlow是Yahoo基于网站性能优化规则推出的浏览器扩展插件。其最新版本基本支持所有主流浏览器。大家可以搜索安装。此外，IE、Chrome、Firefox默认的运行页面调试功能快捷键都是F12，因此可以直接打开浏览器后点击F12,进入快速调试模式。

所示，就会发现页面请求了很多不同的资源。注意：该网站只使用了2个JavaScript文件，但是随着网站功能的增加，网站的JavaScript文件数量会变得庞大，那时就需要捆绑压缩所有的JavaScript文件。

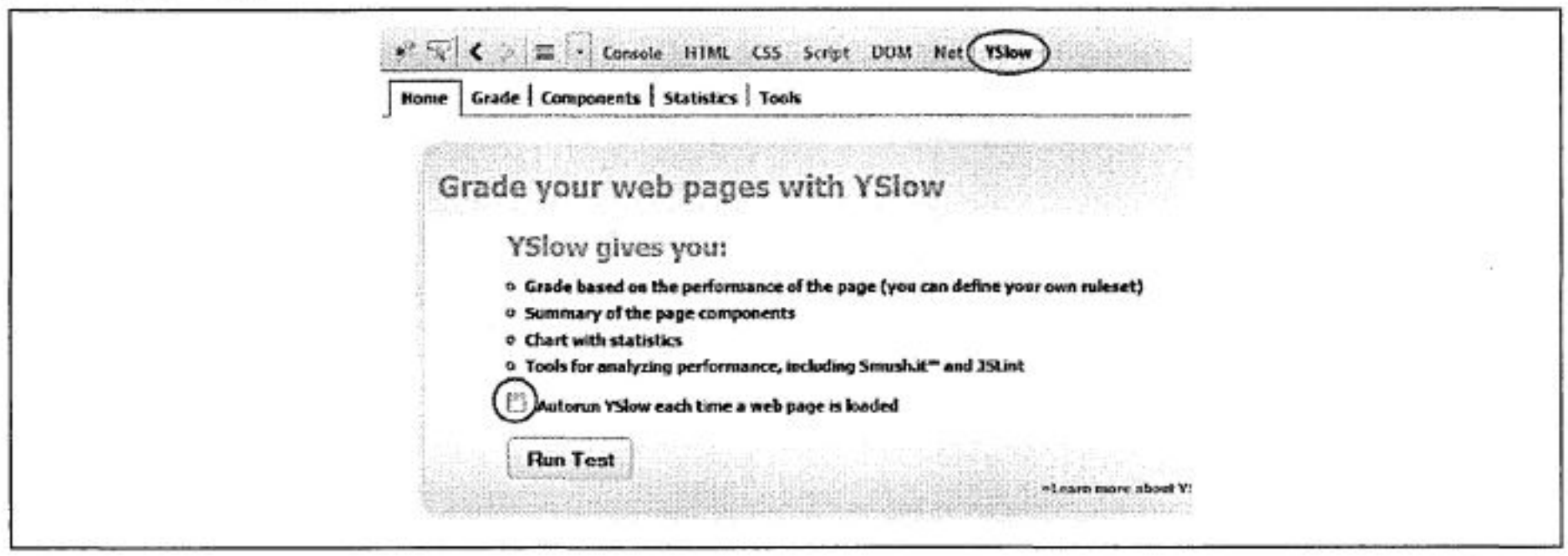


图13-11　Firebug的YSlow选项卡

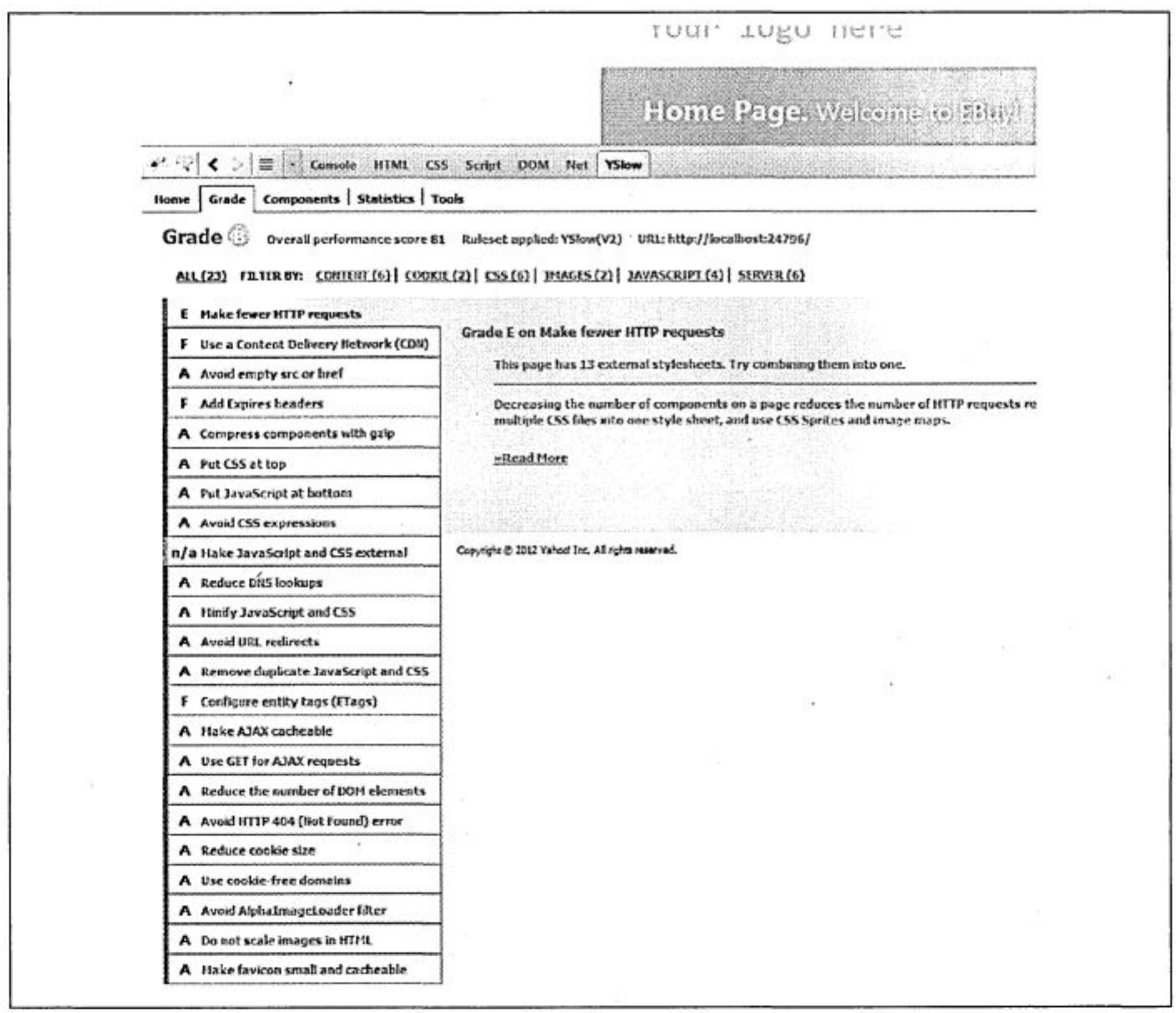


图13-12　ASP.NET MVC默认网站模板的测试评分

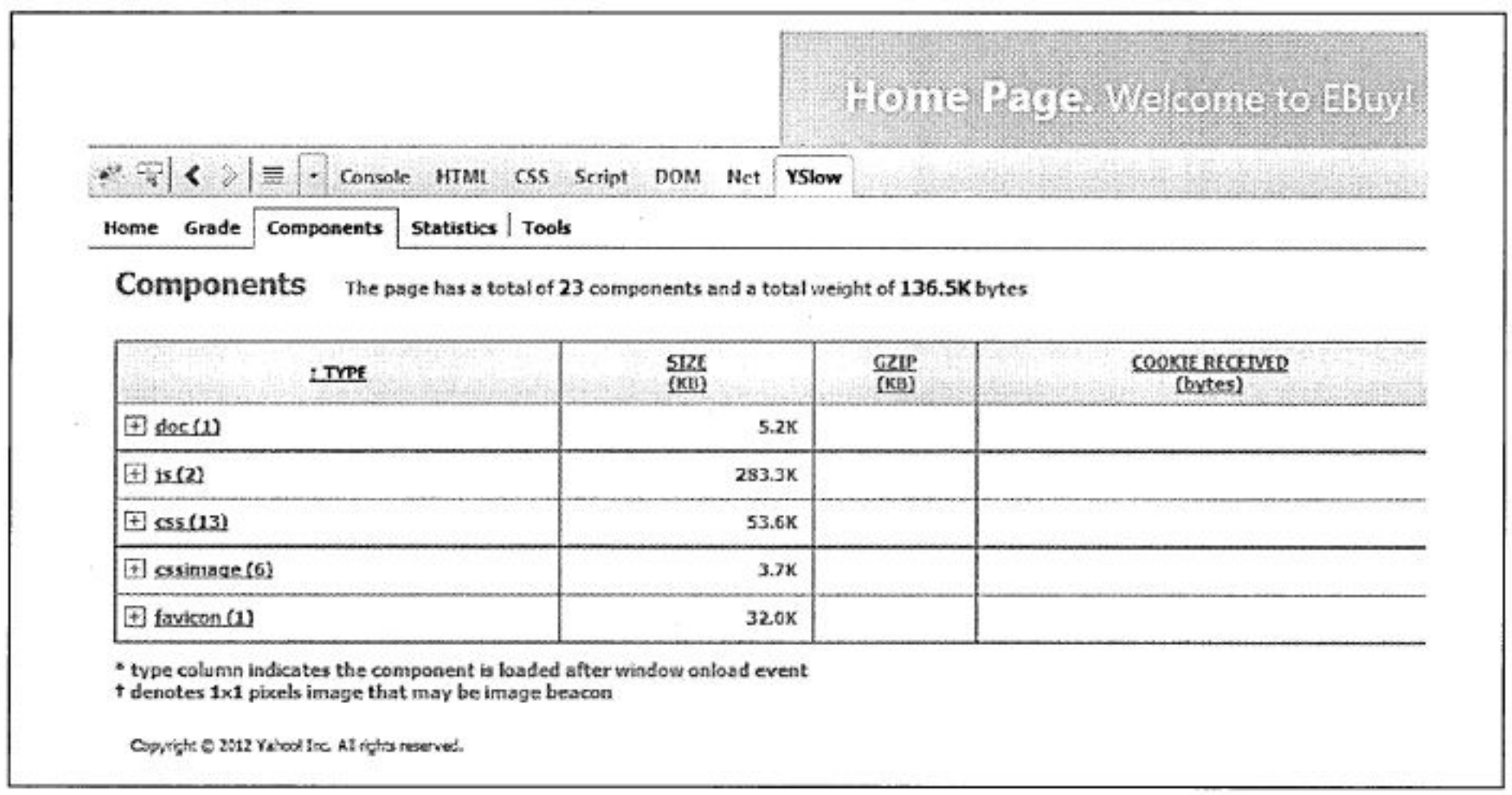


图13-13 YSlow的“组件”选项卡显示的页面请求资源列表

下面来看如何使用ASP.NET MVC 4内置的文件捆绑特性来减少页面的HTTP请求数量。

运行ASP.NET MVC

Putting ASP.NET MVC to Work

ASP.NET MVC 4和.NET Framework 4.5提供了支持捆绑和压缩的新类库`System.Web.Optimization`（译注7）。该类库提供了根据不同的自定义规则进行不同的资源捆绑的基本功能（后续几节会介绍），包括内置的JavaScript和CSS文件压缩器。这个类库也提供了自动清理缓存功能，例如，当内容变化时检测浏览器缓存。另外一个重要的新特性就是它提供了自动合并CSS样式文件和JavaScript文件的功能（当它们在同一路径时）。

绝大部分情况下，这种方法足够满足需求。但是，如果网站有特殊的要求，就可能要借助第三方的解决方案。

捆绑和压缩

如果打开“Layout.cshtml”文件看下顶部代码，就会看到`System.Web.Optimization`类库的两个方法`@Styles`和`@Scripts`：

```
@Styles.Render("~/Content/themes/base/css", "~/Content/css")
@Scripts.Render("~/bundles/modernizr")
```

正如它们的名字一样，它们负责捆绑和压缩CSS文件和JavaScript文件或者其他资源。

译注7：System.Web.Optimization 包含了微软提供的针对 Web 优化的工具类，详细介绍见 *http://msdn.microsoft.com/zh-cn/library/system.web.optimization.aspx*。

Render()方法接受要渲染的虚拟路径字符串作为参数，且可以同时传入多个虚拟路径字符串。在运行时会替换成HTML标签，代码如下：

```
<link href="/Content/site.css" rel="stylesheet" type="text/css" />
<script src="/Scripts/modernizr-2.0.6.js" type="text/javascript"></script>
```

我们完全可以控制这些绑定功能，后面一节将会做详细介绍。如前面的例子所示，我们可以创建多个捆绑，并在单个调用中渲染这些数据。这里，我们使用了两个捆绑方法，但是这两个方法是在同一个调用里渲染的——一个是基本的主题，另外一个是网站的默认主题。

我们发现，在body标签结束符的上面有JavaScript渲染的调用代码：

```
@Scripts.Render("~/bundles/jquery")
```

ASP.NET MVC默认的项目模板已经遵循了“CSS样式文件置顶”和“JavaScript文件置底”的网站优化原则。

唯一的特例就是modernizr.js，它在页面文档顶部渲染。ModernizrJavascript库的作用就是探测浏览器支持的功能（例如，视频、音频、SVG、HTML 5新特性等），然后给<head>标签添加CSS样式的Class名。

使用这个类库，可以让网站很好地兼容旧的浏览器，尤其在有些浏览器不支持网站需要的功能时。如果把Modernizr放到页面底部，就会延后整个浏览器功能的探测，影响整个页面的显示（有些元素可能会出现混乱，等页面加载完毕后又恢复正常）。

定义捆绑

可以通过调研BundleCollection.Add()（System.Web.Optimization命名空间下）方法来自定义捆绑规则，传递的参数是ScriptBundle或者StyleBundle。为了创建ScriptBundle，需要传递包含一个或者多个脚本的虚拟路径给构造函数（视图使用的是虚拟路径），代码如下：

```
// 通配符——包含所有以"jquery-1"开头的 js 文件
var j QueryBundle= new ScriptBundle("~/bundles/jquery").Include(
                        "~/Scripts/jquery-1.*");

// 明确包含脚本文件
var jQueryValBundle= new ScriptBundle("~/bundles/jqueryval").Include(
                        "~/Scripts/jquery.unobtrusive-ajax.js",
                        "~/Scripts/jquery.validate.js",
                        "~/Scripts/jquery.validate.unobtrusive.js")
```

我们也可以使用相似的方法创建StyleBundle：

```
var siteBundle= new StyleBundle("~/Content/css").Include("~/Content/site.css")
```


ASP.NET MVC模板会自动包含这些CSS样式资源——在App_Start\BundleConfig.cs文件中定义。这个文件放在App_Start文件夹下，保证网站启用时只运行一次。但是，只要在应用开始阶段注册了，就可以在项目里任何位置使用捆绑功能。注册代码如下：

```
public class MvcApplication: System.Web.HttpApplication
{
    protected void Application_Start()
```

```
        {
            AreaRegistration.RegisterAllAreas();

            FilterConfig.RegisterGlobalFilters(GlobalFilters.Filters);
            RouteConfig.RegisterRoutes(RouteTable.Routes);

            // 注册捆绑模块
            BundleConfig.RegisterBundles(BundleTable.Bundles);
        }
    }
```

启用捆绑

如果还记得我们的第一次测试。YSlow给出的差评是由于HTTP请求太多——现在就来处理这个问题。与其单独请求不同的资源文件，为什么不尝试把这些文件捆绑在一起以减少外部请求呢？

问题的答案是在调试模式下，Visual Studio禁用了捆绑和其他的优化功能，以便调试网站的代码。可以通过设置`BundleTable.EnableOptimizations=true`来强制启用这项功能。

为了看下效果，让我们在Release发布模式下编译网站，然后重新使用YSlow进行测试。虽然整体得分还是B级（由于其他原因），但是可以看到YSlow对减少HTTP请求这一项的评分变高了，如图13-14所示。图13-15所示的YSlow组件资源选项卡反映了捆绑机制的作用，现在只有两个样式文件：/Content/themes/base/css和/Content/css。这和之前定义的一样。

在浏览器里请求这个URL地址，也同样验证了这种捆绑的结果！

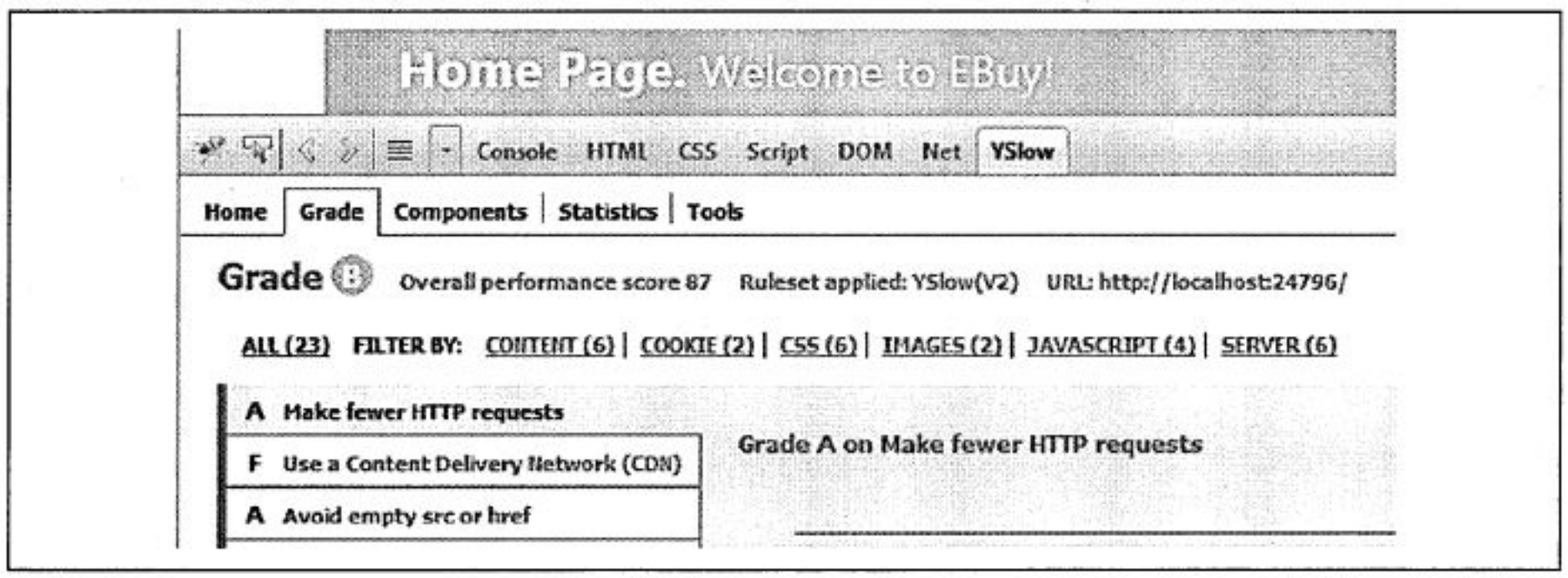


图13-14　使用捆绑来减少HTTP请求数量

清理缓存

记住：浏览器是根据URL来缓存数据的。浏览器无论何时请求资源，都会先根据URL来检查缓存里是否包含了该资源文件。如果包含了，浏览器就不会再去请求，而是使用缓存的文件来渲染网页。所以，为了使用浏览器缓存，要尽量保持URL不变。

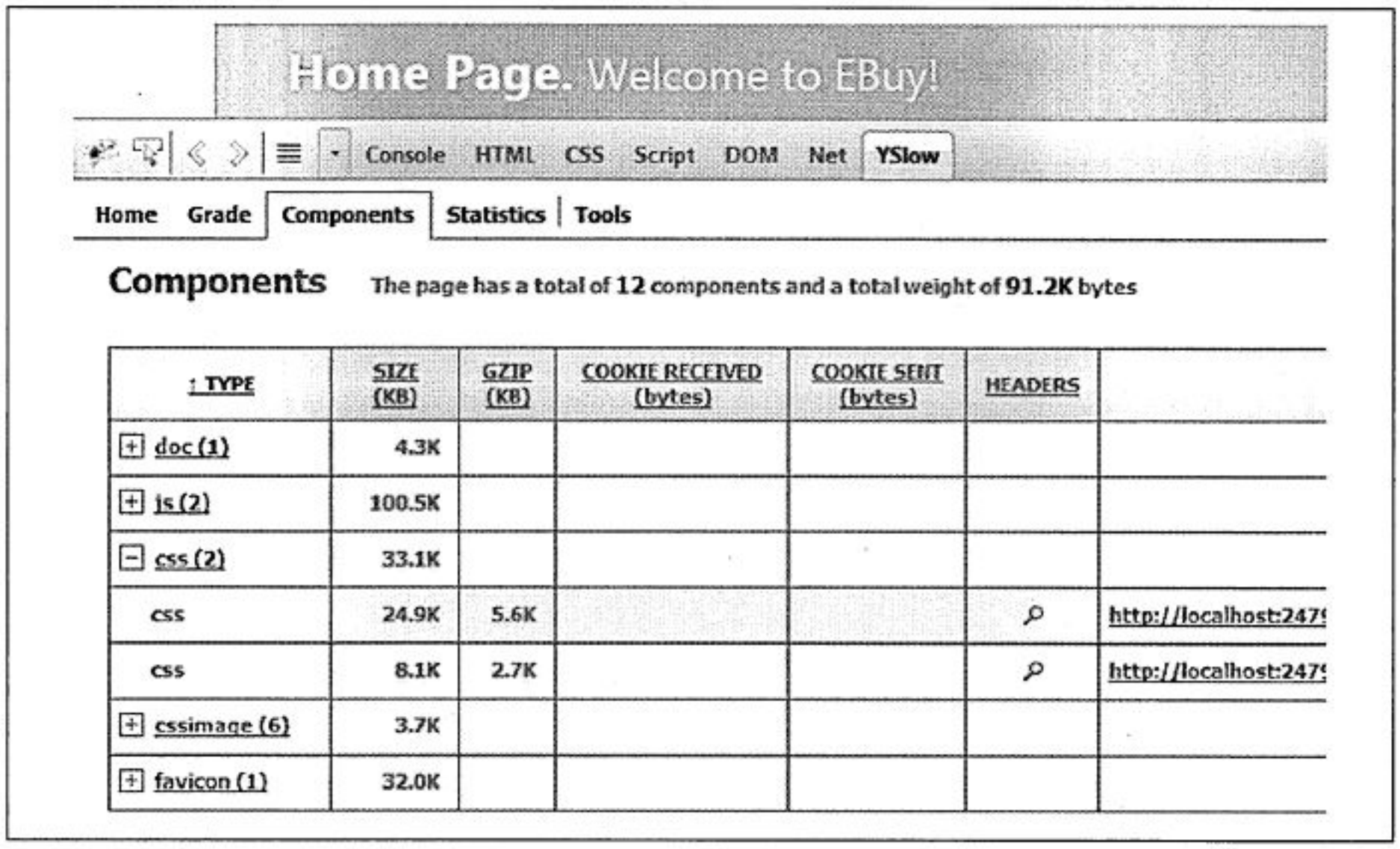


图13-15 CSS样式文件现在捆绑的两个文件

这样也给Web开发人员带来了一些问题：当在开发网站中修改JavaScript和CSS样式文件后，打开浏览器发现网页内容使用的还是旧的样式。因此使用相同的URL没有任何帮助，因为浏览器会继续使用旧的缓存文件。通用的解决办法就是在URL后面加上版本号，代码如下：

```
<link type="text/css" rel="stylesheet" href="/Content/site.css?v=1.0">
```

现在修改样式文件时，只要修改了版本号，浏览器就会下载新的文件。这个方法也有坏处，就是每次需要手动修改版本号，且可能出错。不过，ASP.NET MVC提供的捆绑机制会自动处理这个问题，它会给请求的URL自动添加哈希值，而不需要手动操作，代码如下：

```
<link type="text/css" rel="stylesheet"
        href="/Content/themes/base/css?v=UM624qf1uFt8dYtiIV9PCmYhsyeewBIwY4Ob0i8OdW81">
```

现在，无论何时，只要修改了文件内容，就会重新生成新的哈希值，浏览器就会看到不同的URL，然后获取新版本的文件来渲染网页内容。

总结

Summary

本章展示了网页加速的基本原则。同时也介绍了如何使用IIS和ASP.NET MVC内置的功能来实现一些列举的优化规则，比如捆绑、压缩以及缓存。这里介绍的方法易于实现，而且指明了优化网站性能的基本方向。

第14章

高级路由

Advanced Routing

第1章简单介绍了ASP.NET MVC框架的路由基础知识，介绍的例子是Visual Studio默认为ASP.NET MVC项目设置的路由，而且仅仅是介绍ASP.NET MVC框架如何使用路由来映射每个请求和控制器、操作方法之间的关系。

绝大部分情况下，无需担心任何比Visual Studio默认创建的路由更高级的技术问题。默认的路由遵循了标准的ASP.NET MVC惯例，且允许我们创建新的控制器和操作方法而无需担心路由引擎如何定位它们。只有当默认的路由和URL模式无法满足网站的特殊需求时，才会需要ASP.NET MVC框架强大的路由机制。

本章将超越常规的ASP.NET MVC路由开发来深入介绍强大的ASP.NET MVC路由引擎。首先，从URL对于网站用户体验的重要性开始，介绍网站页面的URL如何影响网站在搜索引擎中的排名。然后，通过了解不同的URL模式来深入更加高级的路由知识点。同样，会介绍如何定义路由约束，并且会介绍一款非常有用的方便调试路由错误的Glimps工具。最后，会介绍如何扩展ASP.NET MVC路由框架来帮助改善网站开发体验，随后介绍如何使用ASP.NET MVC路由的扩展性来创建自定义路由引擎。

路标指示系统

Wayfinding

路标指示系统是指可以指引人们从一个地方到达另外一个地方的路线指示牌。在万维网中，最自然的路标指示系统就是URL。事实上，很多人上网的第一步通常就是，打开浏览器，输入URL网址。例如，每个人都知道的Yahoo地址：

```
www.yahoo.com
```

URL是可以帮助用户发现更多有价值的网站资源的最简单最直观的方式，更重要的是，它可以帮助我们找到需要的信息。URL易于记忆，而且易于传播给他人。

在互联网早期阶段，URL通常是从域名直接扩展而来，下面的URL例子就是1996年版本的Yahoo网站例子。

```
http://www.yahoo.com/Computers_and_Internet/Software/Data_Formats/HTML/HTML_2_0/
```

在这个URL例子里，目录结构相当有条理且易于读/写。当然，这个URL也很容易被“入侵”，用户可以通过URL跳转到网站的不同目录层级中，基本上，只要略懂网络知识的人就可以猜测出其他同类资源的URL。

简化URL非常重要，如果经常忽视这个问题，可能直接影响网站用户体验。事实上，10多年前雅各布·尼尔森（Jacob Nielsen）已认识到良好的URL结构对网站整体的实用性非常重要，而且在2007年，另外一个报告证实了这个观点——雅各布·尼尔森（Jacob Nielsen）团队和微软研究院（Microsoft Research）的Edward Cutrell和Zhiwei Guan跟踪研究发现，用户大约花费24%的时间用于关注URL。

随着动态网页技术的发展，越来越多的网站支持动态网页，开发人员可以使用不代表网站资源目录路径的URL，也可以存储用户行为信息和应用状态信息。随着开发技术的发展，网站的URL变得更加面向机器而不是用户，包含各种神秘代码和数据ID信息，例如，

```
http://demo.com/store.aspx?v=c&p=56&id=1232123&s=12321-12321321312-12312&s=0&f=red
```

这种URL显然非常难读和难记。若不信，就可以试试在电话里告诉朋友这个网址，看看结果如何。

值得庆幸的是，这些Web开发人员逐渐开始认同雅各布尼尔森（Jacob Nielsen）等人的观点：URL的结构非常重要，而且在网站的实用性方面起着非常重要的作用。为了解决这些问题，引入了新的概念“路由”——作为URL和系统功能之前的中枢层。

路由允许我们在应用逻辑之上构建URL层。例如，在上面例子里使用路由就可以隐藏store.aspx，并且可以把功能参数映射到URL字段中。

换句话说，可以把丑陋的URL转换得优雅一些，而且更具意义，例如：

297

```
http://example.com/store/toys/RadioFlyer/ClassicRedWagon/12321-12321321312-12312/red
```

虽然新URL也提供了相同的信息，但是这个URL更加具备用户友好性。用户很容易就可以看出正在访问的商品，要买的就是玩具类别中的Radio Flyer经典红色小拖车。用户甚至可以自己修改URL参数以浏览网站的其他页面。

在ASP.NET MVC的世界里，路由引擎扮演的角色是应用控制器和操作的中枢，而不是为.aspx终节点服务。所以当开发ASP.NET MVC网站时，很有必要思考网站暴露的URL结构。

URL和搜索引擎优化
URLs and SEO

除了网站通用的实用性外，友好的URL还可以为网站带来另外一个好处：改善搜索引擎排名。URL的用户友好性对搜索引擎同样适用。URL优化属于改善搜索引擎排名的另外一种更强大的技术，称为搜索引擎优化（Search Engine Optimization，SEO）。

SEO的目标就是增加网站在搜索引擎结果里的排名PR（pages rank）值。如果认为搜索结果排名对网站非常重要，那么要考虑如何使用SEO技巧来优化网站URL。记住，SEO也是一门很深的学问，因为每种搜索引擎可能使用了不同的排名规则——并且通常都是秘密的——网页排名算法，所以没有明确的规则可用（译注1）。

译注1：谷歌搜索引擎排名规则相对客观，搜索问题时返回的结果人为干预较少。国内的某些搜索引擎采用竞价排名，即使网站优化得非常好，往往也会被付更多钱的广告商挤在后面。

通常，也有些比较好的搜索引擎优化技巧。这里收集整理了一些经典的规则，具体如下：

URL越短越好

谷歌的算法是给URL前5个之后的单词更少的权重，所以URL越长越不具优势。此外，URL越短越能增加可用性和可读性。

破折号代替下划线

谷歌搜索引擎会把破折号看成单词分隔符，而且会为每个单词创建索引。

小写字母

意识到URL所起的作用，坚持尽可能使用小写字母。绝大部分搜索引擎遵守HTTP规范，也就是说，URL是区分大小写的，搜索引擎会把Page1.htm 和 page1.htm当成两个不同的页面。这样会导致内容被索引两次，不利于搜索排名。

更多关于URL优化的内容，可以查阅SEOmoz.org开发人员创建的“SEO作弊表”，地址是：http://www.seomoz.org/blog/seo-cheat-sheet-anatomy-of-a-url。

298

构建路由
Building Routes

通过前面的学习，已经知道了URL对网站整体的用户体验的重要性。了解了这些后，下面再来看看如何使用ASP.NET MVC路由框架来控制用户访问网站的URL路径。

先从深入分析路由中用户定义URL模式的几种不同方式开始，再使用ASP.NET MVC新网站创建时生成的默认路由：

```
routes.MapRoute(
        "Default", // 路由名称
        "{controller}/{action}/{id}", // URL 参数
        new{ controller = "Home", action = "Index", id = UrlParameter.Optional }
        );
```

路由中的URL由一些字段组成。这些字段可以是常量字符串，也可以是{}包含的参数，还可以被斜线分割开来。

因为路由往往相当于网站根路径，所以无法以斜线(/)或波浪线(~)表示。如果违反这些原则，那么路由引擎将会抛出异常。

路由可以区分每个占位符，因为它们都包含在大括号里，例如，

```
{controller}
```

这个例子中，每个大括号都是一个独立的占位符，可由斜线分开。当然，也可以创建包含多个参数的URL，中间用常量字符串分割，例如，

```
{param1}-{param2}-{param3}
```

当路由引擎解析URL时，它会从每个大括号里提取数值，然后实例化`RouteData`类。每个类包含一个用来存储路由中需要的所有值的字典对象，这个类实例被ASP.NET MVC使用。另外，要注意的是，默认添加到字典集合里的值默认类型都是String类型。

在前面默认路由的例子中，意味着实际请求的URL是：

http://demo.com/Home/Index/1234

路由引擎会解析如表14-1中的键值对并存入RouteData字典中。

表14-1　RouteData字典中的键值对

参数	值
{controller}	Home
{action}	Index
{id}	1234

路由参数

正如第1章中介绍的，路由引擎使用{controller}和{action}的值来决定实例化哪个控制器，以及执行哪种操作方法。但是，浏览器中加载的那个URL不包含任何占位符里的参数，这时MapRoute()方法里的第三个参数开始起作用，允许用户为任何占位符设置默认值。

如果回头看一下默认的路由，就会发现调用MapRoute()方法中的一个匿名类型为括号中定义的三个参数设置默认值，编译器会动态确定匿名类型。

```
routes.MapRoute(
    "Default", // 路由名称
    "{controller}/{action}/{id}", // 包含参数的 URL
    new{ controller = "Home", action = "Index", id = UrlParameter.Optional }
);
```

现在就了解网站默认打开Home Controller的奥秘了。尽管URL没有包含全部控制器或者操作需要的值，但是路由包含了默认值，路由引擎系统可以定位到默认的控制器和操作上。当路由引擎分析URL时，默认的值就会加入RouteData字典，而且，如果解析URL时找到了括号中的值，就会直接重写默认值。

下面看看URL路由的其他例子。下面的例子中，URL包含了一些常量字符串作为开始部分。这意味着请求的URL只有包含这些常量字符才能匹配这个路由：

```
routes.MapRoute(
    "Default",
    "Admin/{controller}/{action}/{id}",
    new{ controller = "Home", action = "Index", id = UrlParameter.Optional }
);
```

这里是一些满足这个路由的URL例子：

- *http://demo.com/Admin*
- *http://demo.com/Admin/Home*
- *http://demo.com/Admin/Home/Index*
- *http://demo.com/Admin/Home/Index/1234*

```
routes.MapRoute(
    "Default",
    "{site}/{controller}/{action}/{id}",
    new{ controller = "Home", action = "Index", id = UrlParameter.Optional }
);
```

下面是一些满足这个路由的URL例子：

- *http://demo.com/Admin*
- *http://demo.com/Store/Home*
- *http://demo.com/Store/Home/Index*
- *http://demo.com/Store/Home/Index/1234*

下面的路由定义只包含一个{id}，这个id值依赖于控制器和操作方法设置的默认值，而且参数是可选的，并不是必须的：

```
routes.MapRoute(
    "Default",
    "{id}",
    new{ controller = "Home", action = "Index", id = UrlParameter.Optional }
);
```

匹配路由的URL如下：

- *http://demo.com/*
- *http://demo.com/1234*

下面的路由包含常量部分和单个大括号的参数部分{id}，同样，这个值也取决于控制器和操作方法设置的默认值：

```
routes.MapRoute(
    "Default",
    "users/{id}",
    new{ controller = "Home", action = "Index", id = UrlParameter.Optional }
);
```

匹配这个路由的URL如下：

- *http://demo.com/users*
- *http://demo.com/users/1234*

```
routes.MapRoute(
    "Default",
    "category/{id}/export{type}.{format}/",
    new{ controller = "Category", action = "Export" }
    );
```

匹配这个路由的URL例子如下：

- *http://demo.com/category/123abc/exportEvents.json*

路由顺序和优先级

随着应用变得日益复杂，就需要注册更多的路由来处理这些需求。当添加新路由时，要特别注意路由的顺序。当路由引擎尝试定位路由时，它会遍历路由集合中的所有路由。只要发现一个匹配的路由，就会立即停止搜索。这个逻辑也会导致很多意外问题。下面的代码注册了两个不同的路由：

```
routes.MapRoute(
    "generic",
    "{site}",
    new{ controller = "SiteBuilder", action = "Index" }
);

routes.MapRoute(
    "admin",
    "Admin",
    new{ controller = "Admin", action = "Index" }
);
```

第一个路由包含一个大括号，而且默认的控制器是SiteBuilder。第二个路由包含了一个常量字段，默认的控制参数值是Admin。

两个路由都是有效的，配置完全正确、合理，但是，定义的先后顺序可能会引起一些问题。因为第一个路由会优先匹配任何输入的值。这意味着第一个路由会优先匹配*http://demo.com/Admin*。因为第一个找到的路由完全匹配了这个URL，所以第一个路由永远无法访问到此URL。

一定要记住，路由定义的顺序对请求处理的影响至关重要。

路由到现有文件

ASP.NET MVC路由引擎会优先处理请求的物理文件而不是路由表里定义的虚拟路由。因此，对物理文件的请求就不需要路由的处理过程，路由引擎会直接返回物理文件数据，而不是使用URL来匹配路由。某些情况下，也可以重写这种行为，强制ASP.NET MVC框架路由所有的请求。可以通过设置`RouteCollections.RouteExistingFiles=false`来禁用文件请求。

忽略路由

302

除了定义控制器和操作使用的路由外，ASP.NET MVC同样允许给某些URL定义忽略路由，也就是对符合当前请求的URL不予理睬。`RoutesTable`对象暴露了`MapRoute()`方法，同样也暴露了`IgnoreRoute()`方法，通过这个方法就可以定义让路由引擎忽略的URL模式。

下面的代码是Visual Studio为ASP.NET MVC网站创建的默认路由，包含一行忽略路由代码，忽略路由包括.axd这个文件扩展名，可以使用通用的ASP.NET处理程序处理，如Trace.axd和WebResource.axd文件：

```
routes.IgnoreRoute("{resource}.axd/{*pathInfo}");
```

设置了IgnoreRoute()忽略路由代码以后，对符合这种请求的URL，路由引擎就不会处理，而是交给原始的ASP.NET处理程序处理。

通常也可以使用IgnoreRoute()方法来忽略其他请求。例如，网站有时候包含了一些当前的ASP.NET MVC运行时无法处理的字符或代码。此时，就需要使用如下的代码来告诉ASP.NET MVC忽略以php-app开头的URL：

```
routes.IgnoreRoute("php-app/{*pathInfo}");
```

注意：如果打算在网站中使用IgnoreRoute()方法，请记住这个方法调用的位置要在正常的路由代码之前。通常放在开始部分。

捕获所有路由

ASP.NET MVC中URL解析引擎的另外一个特性就是，可以指定“匹配所有占位符（catch-all placeholder）”。匹配所有占位符要求必须把*号放到前面，而且只能整体位于路由定义的末端。

事实上，在前一个例子里已经看到过一个捕获所有括号的代码：

```
routes.IgnoreRoute("php-app/{*pathInfo}");
```

匹配所有占位符也可以在正常的路由中使用。例如，在搜索功能里可以使用匹配所有占位符来获取原始的查询字段数据，代码如下：

```
routes.MapRoute(
    "Default",
    "{controller}/{action}/{*queryValues}",
    new{ controller = "Store", action = "Search" }
);
```

这个路由里，虽然使用了正常的控制器和操作占位符，但是URL末尾添加了一个匹配所有的占位符。所以，如果请求的URL是*http://demo.com/store/search/wagon/RadioFlyer*，路由引擎就会像表14-2那样来解析它。

因此，如果用户输入了*http://demo.com/store/search/wagon/Radio-Flyer*这种URL，那么路由引擎会解析成\{controller\}和\{action\}，然后把\{action\}后面的数据保存到RouteData字典里，键的名称是queryValues。

表14-2　解析匹配所有路由

参数	值
{controller}	store
{action}	search
{queryValues}	wagon/RadioFlyer

也可以使用匹配所有参数让路由引擎忽略包含特定文件扩展名的请求。例如，如果希望路由引擎忽略任意带有ASPX文件的请求，那么可以这样定义匹配所有路由：

```
routes.IgnoreRoute("{*allaspx}", new {allaspx=@".*\.aspx(/.*)?"});
```

这个例子中，使用了IgnoreRoute的重载方法，这个方法包含2个参数，定义要忽略的URL参数以及URL参数的表达式集合。第一个URL参数会评估所有的URL请求，而第二个参数定义的正则表达式会检查这些请求是否包含了.aspx文件扩展名。

路由约束
Route Constraints

到目前为止，我们已经学习了如何在网站中创建路由，以及使用占位符构建URL的不同方式。但是，到目前为止，介绍的内容还没有包含路由如何限制用户URL输入的内容。这意味着即使URL中只包含整数，用户也可以任意输入其他类型的数据，这样可能会导致系统出错、操作方法类型转换失败、无法执行操作。

幸运的是，路由引擎包含验证括号中占位符参数的机制，称为路由约束（route constraints）。

MapRoute()方法包含了一个重载方法，允许对占位符参数设置类型约束。设置约束像创建匿名类型（译注2）一样简单。

这个例子使用的是简单的正则表达式来限制id的值类型：

304

```
routes.MapRoute(
    "Default",
    "{controller}/{action}/{id}",
    new{ controller = "Home", action = "Index", id = UrlParameter.Optional },
    new{ id = "(|Ford|Toyota|Honda)" }
);
```

而这个例子限制id值是numeric数值类型：

```
routes.MapRoute(
    "Default",
    "{controller}/{action}/{id}",
    new{ controller = "Home", action = "Index", id = UrlParameter.Optional },
    new{ id = "\d+" }
);
```

这个例子限制的是三位numeric数值：

```
routes.MapRoute(
    "Default",
    "{controller}/{action}/{id}",
    new{ controller = "Home", action = "Index", id = UrlParameter.Optional },
    new{ id = "\d{3}" }
);
```

通过使用正则表达式来定义约束规则，可以更好地控制占位符参数的数值类型。

如果路由没有满足当前的约束规则，则路由引擎会认为当前请求不匹配，而继续遍历路由表寻找匹配的路由。

译注2：匿名类型（anonymous type）提供了一种将一组只读属性封装到单个对象中而无需首先显式定义一种类型的方便方法。类型名由编译器生成，并且不能在源代码级使用。每种属性的类型由编译器推断。
可通过使用 new 运算符和对象初始值创建匿名类型。有关对象初始值设定项的更多信息请参见对象和集合初始值设定项（《MSDN C#编程指南》）。

了解了这些内容后，就可以使用约束来处理URL相同但控制器或操作不同的路由情况了。

下面的代码展示了使用URL集合相同但约束规则不同的例子：

```
routes.MapRoute(
   "noram",
   "{controller}/{action}/{id}",
   new{ controller = "noram", action = "Index", id = UrlParameter.Optional },
   new{ id = "(us|ca)" }
);

routes.MapRoute(
    "europe",
    "{controller}/{action}/{id}",
    new{ controller = "europe", action = "Index", id = UrlParameter.Optional },
    new{ id = "(uk|de|es|it|fr|be|nl)" }
);
```

虽然通过正则表达式可以覆盖主要的验证用例，但是有时候还需要更复杂的验证规则。这时，可以使用IRouteConstraint接口来创建自定义约束规则。

以下代码中的IRouteConstraint接口包含一个Match()方法声明，这个方法必须在子类中实现：

```
public class CustomerConstraint : IRouteConstraint
{
    public bool Match(HttpContextBase httpContext, Route route, string parameterName,
                      RouteValueDictionary values, RouteDirection routeDirection)
    {
        var cdx = new UsersDataContext();

        // 执行数据库查询
        var result = (from u in cdx.Users
                      where u.Username = values["user"]
                      select u).FirstOrDefault();

        return result != null;
    }
}
```

以上这个示例代码展示了如何创建更复杂的路由约束规则来验证数据是否存在于数据库中。下面的代码展示了如何在创建路由时使用自定义约束规则：

```
routes.MapRoute(
    "Default",
    "{controller}/{action}/{id}",
    new{ controller = "Home", action = "Index" },
    new{ id = new CustomerConstraint() }
);
```

记住，以上代码对路由约束来说是一个有效的用例，而且应用非常普遍。另外，在真实的场景中，还要考虑数据库查询验证数据带来的性能影响。

值得一提的是，有很多开源项目提供了预定义的路由约束规则，而且功能远远好于简单的正则表达式——这意味着我们不需要再编写自己的约束代码。其中一个开源项目就是ASP.NET MVC扩展项目（*http://mvcextensions.codeplex.com/*），它包含了很多路由约束实现代码，如Range、Positive Int/Long、Guid和Enum。

使用Glimpse观察路由

因为路由是请求和应用之前的转向层，因此调试路由有点麻烦。有款非常好用的工具，可以帮助我们在运行时查看路由信息，它就是Glimpse。

Glimpse包含Routes（路由）选项卡，不仅可以显示已经注册的所有路由，还包含很多其他信息，例如，当前页面匹配的路由信息、默认的路由值、默认的约束规则和URL占位符真实的值。图14-1所示为GlimpseRoutes（路由）选项卡界面。

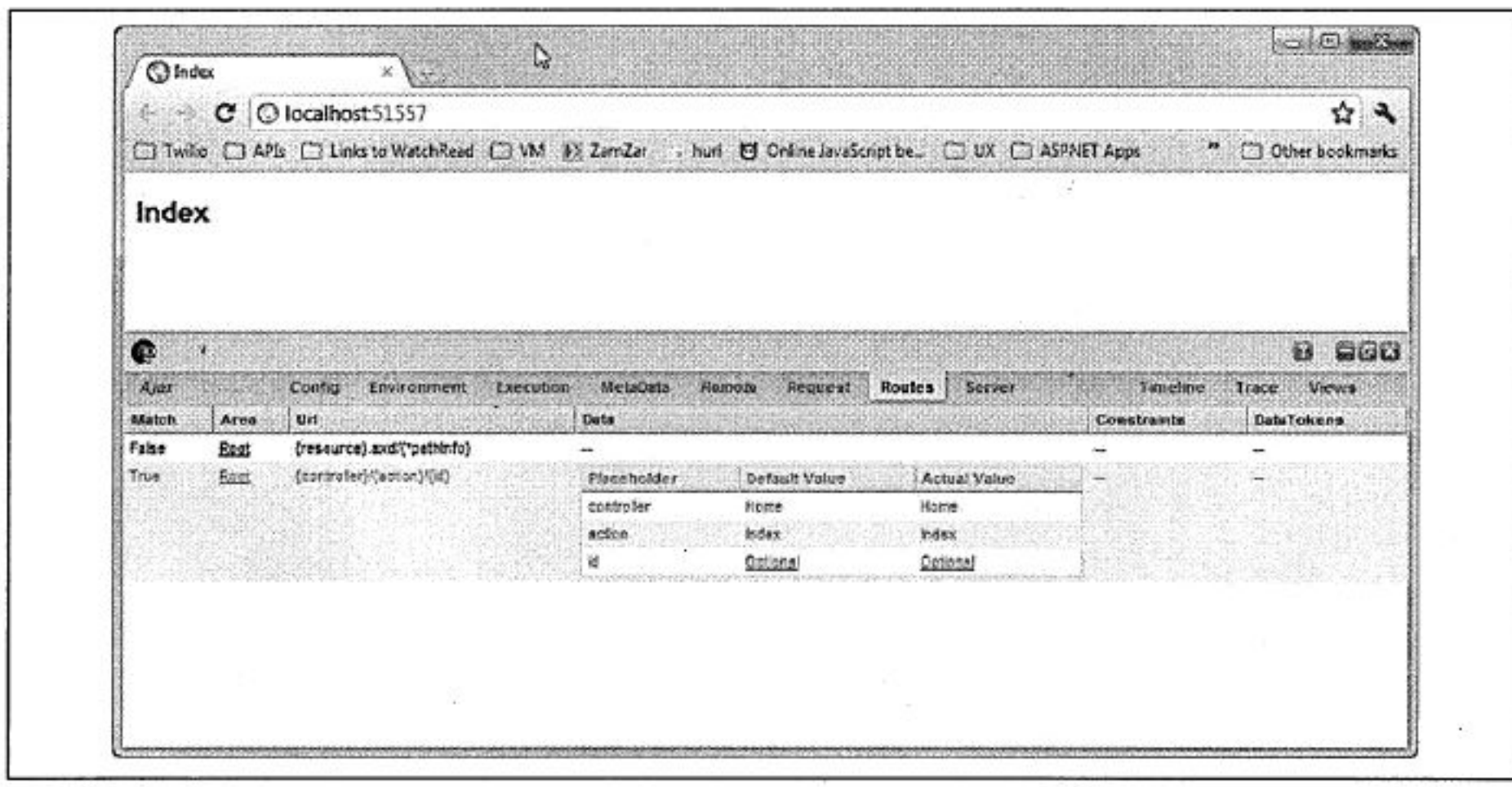


图14-1　Glimpse中的路由表

基于属性标记的路由

Attribute-Based Routing

使用MapRoute()方法注册网站路由非常简单，但是也有缺点。路由注册代码与最终映射的控制器和操作代码完全隔离，对大型应用来说是一件非常头疼的事情。

解决这个问题的方法就是使用**基于属性标记的路由**配置方法。这个方法基于基础路由引擎构建，使用了标准的.NET标记属性，允许通过设置标记属性来直接执行路由。

为了说明如何使用这个方法，我们来创建一个简单的路由标记属性。需要以下两步：

- 一个attribute类。
- 生成路由的类。

第一步，需要使用RouteAttribute来继承System.Attribute，实现自定义标记属性：

```
[AttributeUsage(AttributeTargets.Method, Inherited = true, AllowMultiple = true)]
public class RouteAttribute : Attribute
{
    /// <summary>
```

```
    /// JSON对象，包含的路由数据部分约束
    /// </summary>
    public string Constraints { get; set; }

    /// <summary>
    /// JSON对象，包含的路由数据部分默认
    /// </summary>
    public string Defaults { get; set; }

    /// <summary>
    /// URL路由模式，路由数据占位符
    /// </summary>
    public string Pattern { get; set; }

    public RouteAttribute(string pattern)
    {
        Pattern = pattern;
    }
}
```

RouteAttribute暴露了定义路由的一些简单属性，可以通过这些属性直接定义路由的URL，也就是括号中默认占位符的值及占位符的约束关系。这就是RouteAttribute类全部的代码：自定义的路由标记属性的代码已经创建完成。

要使用这个标记属性，只需要在控制器的操作上直接标记即可，例如，

```
[Route("auctions/{key}-{title}/bids")]
public ActionResult Auctions(string key, string title)
{
    // 查询并返回Auction交易数据
}
```

这个例子中，路由标记属性值是一种定义了从映射到控制器和操作的路由模式。可以使用其他的属性来设置默认的占位符的值和约束规则，也可以设置多个路由标记属性，以便多个路由映射到同一个操作上。

第二步，创建一种方式让RouteAttribute真正注册成功。现在需要做的是创建一个新类RouteGenerator。

RouteGenerator的构造函数需要传递一些参数，包括RouteCollection路由集合实例对象、RequestContext请求上下文及当前网站中定义的所有控制器的集合。在构造函数里，还要创建一个JavaScriptSerializer序列化对象，来序列化和反序列化JSON对象，代码如下：

```
public RouteGenerator(
        RouteCollection routes, RequestContext requestContext,
        ControllerActions controllerActions
    )
{
    _routes = routes;
    _controllerActions = controllerActions;
    _requestContext = requestContext;

    _javaScriptSerializer = new JavaScriptSerializer();
}
```

接下来，需要一个生成新路由的方法：

```
public virtual IEnumerable<RouteBase>Generate()
{
    IEnumerable<Route>customRoutes =
        fromcontrollerAction in _controllerActions
        fromattribute in controllerAction.Attributes.OfType<RouteAttribute>()
        let defaults = GetDefaults(controllerAction, attribute)
        let constraints = GetConstraints(attribute)
        let routeUrl = ResolveRoute(attribute, defaults)
        select new Route(routeUrl, defaults, constraints, new MvcRouteHandler());

    return customRoutes;
}
```

Generate()方法首先使用控制器操作的列表，然后使用LINQ来查询所有标记了RouteAttribute属性的操作，最后为每个路由注册。为了实现这项功能，LINQ使用了很多查询属性值、URL、默认值和约束的帮助方法，然后转换成新的路由可以理解的数据类型。

例14-1展示了RouteGenerator类的完整代码。

例14-1 RouteGenerator类

```
public class RouteGenerator
{
    private readonly RouteCollection _routes;
    private readonly RequestContext _requestContext;
    private readonly JavaScriptSerializer _javaScriptSerializer;
    private readonly ControllerActions _controllerActions;

    public RouteGenerator(
            RouteCollection routes, RequestContext requestContext,
            ControllerActions controllerActions
        )
    {
        Contract.Requires(routes != null);
        Contract.Requires(requestContext != null);
        Contract.Requires(controllerActions != null);

        _routes = routes;
        _controllerActions = controllerActions;
        _requestContext = requestContext;

        _javaScriptSerializer = new JavaScriptSerializer();
    }

    public virtual IEnumerable<RouteBase>Generate()
{
    IEnumerable<Route>customRoutes =
        from controllerAction in _controllerActions
        from attribute in controllerAction.Attributes.OfType<RouteAttribute>()
        let defaults = GetDefaults(controllerAction, attribute)
        let constraints = GetConstraints(attribute)
        let routeUrl = ResolveRoute(attribute, defaults)
        select new Route(routeUrl, defaults, constraints, new MvcRouteHandler());

    return customRoutes;
}
```

309

```
private RouteValueDictionary GetDefaults(
        ControllerAction controllerAction,
        RouteAttribute attribute
    )
{
    var routeDefaults = new RouteValueDictionary(new {
        controller= controllerAction.ControllerShortName,
        action= controllerAction.Action.Name,
    });

    if(string.IsNullOrWhiteSpace(attribute.Defaults) == false)
    {
        var attributeDefaults =
            _javaScriptSerializer.Deserialize<IDictionary<string, object>>(
                attribute.Defaults);

        foreach(var key in attributeDefaults.Keys)
        {
            routeDefaults[key] = attributeDefaults[key];
        }
    }

    return routeDefaults;
}

private RouteValueDictionary GetConstraints(RouteAttribute attribute)
{
    var constraints =
        _javaScriptSerializer.Deserialize<IDictionary<string, object>>(
            attribute.Constraints ??string.Empty);

    return new RouteValueDictionary(constraints ?? new object());
}

private string ResolveRoute(
        RouteAttribute attribute,
        RouteValueDictionary defaults
    )
{
    // 显式URL胜过一切
    string routeUrl = attribute.Pattern;

    // 如果不存在，则尝试找出它
        if(string.IsNullOrEmpty(routeUrl))
            routeUrl= _routes.GetVirtualPath(_requestContext, defaults).VirtualPath;

        if((routeUrl ?? string.Empty).StartsWith("/"))
            routeUrl= routeUrl.Substring(1);

        return routeUrl;
    }
}
```

最后，调用RouteGenerator类就可以在应用程序开始的时候注册路由了。为此，需要在RegisterRoutes()方法里实例化RouteGenerator对象，让它来生成所有的路由。只需要简单循环即可把每个路由插入路由表中，代码如下：

```
var routeGenerator = new RouteGenerator(routes,
    HttpContext.Current.Request.RequestContext,
    ControllerActions.Current);

var actionroutes = routeGenerator.Generate();

for each(var route in actionroutes)
{
    RouteTable.Routes.Insert(0, route);
}
```

编写完RouteGenerator调用代码以后，现在就可以通过属性标记来给操作注册路由，而不需要在外部注册路由了。显然，RouteGenerator只是一种简单路由注册方法。还有很多功能更强大的开源项目可以实现类似的属性标记路由注册方法。如果不喜欢自己编写自定义代码，也可以直接在ASP.NET MVC网站里使用开源项目代码。

扩展路由

Extending Routing

相信大家对如何在ASP.NET MVC网站里创建路由已经非常熟悉了。但是，仅仅使用这些内置的功能可能还无法满足ASP.NET MVC应用开发的需求。

幸运的是，ASP.NET MVC框架预留了很多扩展点，允许我们扩展路由引擎等功能。本节将介绍路由管道的一些底层细节，包括如何使用这些扩展点。

路由管道

现在开始学习ASP.NET MVC管道的更多详细内容（见图14-2）。

首先，UrlRouteModule会处理新请求，并且这个模块负责把请求URL映射到网站中定义的路由器上。如果找到了匹配的模块，它就会创建对应的IRouteHandler路由处理器的实例。默认情况下，ASP.NET MVC实例化的是MvcRouteHandler对象。IRouteHandler会返回一个HTTP处理器实例来负责处理请求消息。

在ASP.NET中，默认的IRouteHandler路由处理器是MvcRouteHandler，而且默认创建的HTTP处理器就是MvcHandler。HTTP处理器使用控制器工厂来创建对应的控制器对象。ASP.NET MVC管道真正神奇的地方在于整个处理过程到处分布着扩展点。所以，完全可以通过继承UrlRouteModule类向路由行为添加新的功能，或者给IRouteHandler和IHttpHandler插入自定义代码。

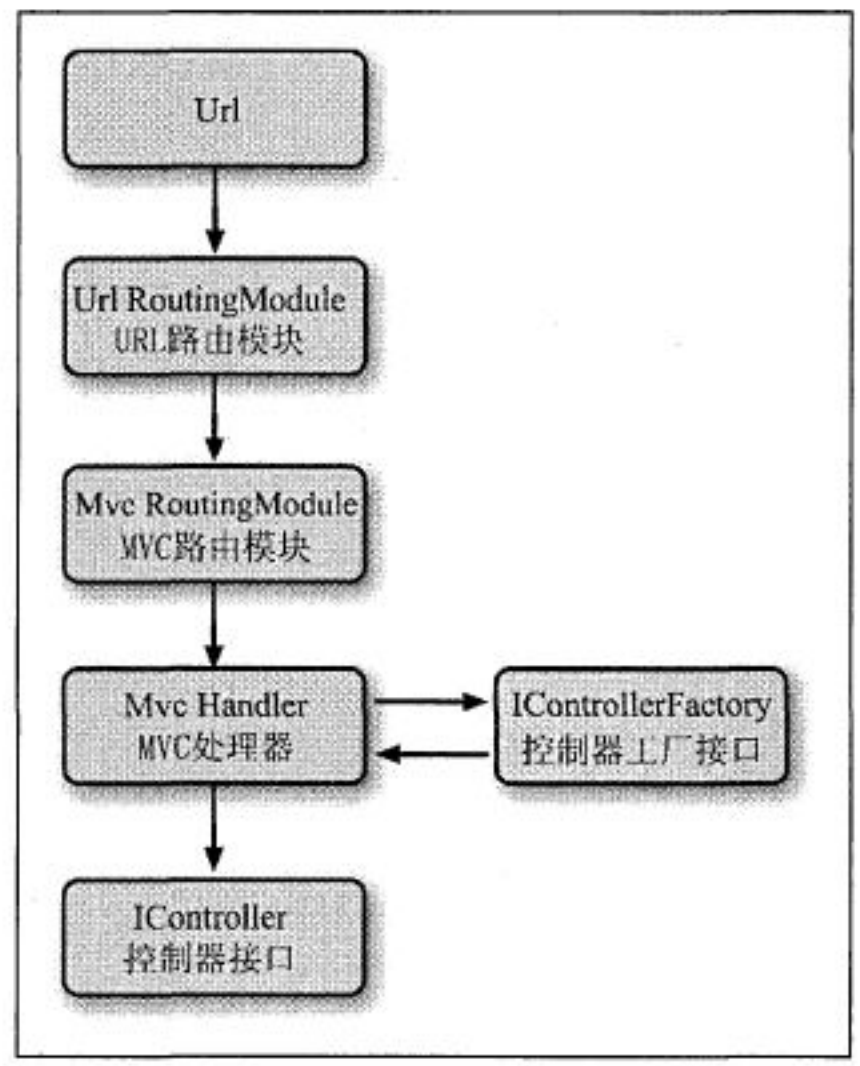


图14-2　ASP.NET MVC管道

现在看看简单的自定义路由处理器类，代码如下：

```
public class SimpleRouteHandler : IRouteHandler
{
    public IHttpHandler GetHttpHandler(RequestContext requestContext)
    {
        return new SimpleHandler(requestContext);
    }
}
```

如以上代码所示，IRouteHandler接口包含一个必须实现的方法GetHttpHandler()。在这个方法里，我们可以控制要实例化哪个IHttpHandler处理器来处理路由请求。正常情况下，MvcRouteHandler会创建一个MvcHandler类的实例，但是，使用自定义IRouteHandler就可以实例化自定义处理器类。在这个例子中，可以创建新的SimpleHandler类，传递requestContent参数给它的构造函数，再返回处理器的实例。

下面的代码展示了如何在新建的网站中注册新创建的SimpleRouteHandler：

```
routes.Add(new Route("{controller}/{action}/{id}", new SimpleRouteHandler()));
```

正如我们看到的，Route类的重载构造函数接受一个路由处理器。如果UrlRoutingModule选择了某个路由器，它就会知道要创建SimpleRouteHandler的实例，而不再使用默认的MvcRouteHandler路由处理器。

现在来看一个如何构建自己的路由处理器的真实的例子。我们在某个公司工作多年，有时要处理COM对象，提供一些网站需要的功能。虽然.NET框架可以很方便地调用COM对象，但COM代码交互的过程与原生.NET框架代码交互过程相比，有些复杂。

例如，在ASP.NET运行时中，每个请求都有独立的线程处理，而且当使用COM处理时，每个线程是独立的“多线程套间”（或MTA）（译注3）。

由于VB6 COM组件与ASP.NET的COM多线程套间模式并不兼容，因此，ASP.NET每次请求与COM对象交互时，COM必须通过“单线程套间（STA）”执行这些请求。这种情形会导致非常严重的性能瓶颈，正如名字的含义一样，每个应用程序只包括一个单线程套间，而且请求必须顺序等待每个请求执行，因此会造成很大的性能瓶颈。

通过使用GetApartmentState()方法可以很容易发现请求的套间状态，代码如下：

```
public string ThreadState()
{
    var thread = System.Threading.Thread.CurrentThread;
    ApartmentState state = thread.GetApartmentState();
    return state.ToString();
}
```

译注3：多线程套间（Multithreaded Apartment，MTA）是COM编程中的概念，是指COM对象的生存空间。单线程套间，只能包含一个线程，通过调用CoInitialize(NULL)进入。多线程套间，可以包含任意多的线程(具体数目由操作系统决定)。一个进程只能包含一个这种套间，所有调用CoInitializeEx(NULL, COINIT_MULTITHEADED)的线程都会进入这个套间。

如果在正常的ASP.NET网站中运行这个方法，就会发现多线程套间显示在浏览器中。

设计上，COM不支持为单个请求使用多线程套间模式。当第一个请求进来时，COM会创建单线程套间来处理这个请求。然后，对后续的每个请求，不会再创建新的请求，而是把所有的请求放入这个线程的请求处理队列中。

ASP.NET Web Form提供了简单的解决方案：使用AspCompat页面指令。如果在Web Form页面上设置的AspCompat页面指令等于True，COM就会适应单线程套间线程池来处理页面的请求和应答对象。这样也可以在它们的创建器套间内创建使用ThreadingModel="Apartment"方式注册的COM对象，提供的创建器运行在单线程套间线程中。其优势是，因为COM对象与创建器共享了套间，所以多个请求可以并行执行请求，从而不需要排队等待，因此不会导致性能瓶颈。

然而，ASP.NET MVC没有提供任何类似AspCompat的机制，所以需要自己模仿AspCompat指令。这是我们创建自定义路由处理器的绝佳机会，允许请求线程可以运行在单线程套间中，而不是多线程套间。

现在看看如何使用ASP.NET MVC扩展点来模仿AspCompat指令创建自定义路由处理器。首先从创建重定向URL请求到自定义HTTP处理器类的自定义路由处理器，代码如下：

```
public class AspCompatHandler : IRouteHandler
{
    protected IHttpHandler GetHttpHandler(RequestContext requestContext)
    {
        return new AspCompatHandler(requestContext);
    }
}
```

如前所示，在AspCompatHandler类中，使用GetHttpHandler来提供给ASP.NET MVC一个AspCompatHandler实例，这就是我们用来路由新请求的处理器实例。

然后创建AspCompatHandler。从创建System.Web.UI.Page的子类开始，通过继承标准的Web Form Page类，就可以访问ASP.NET的AspCompat指令了，代码如下：

```
public class AspCompatHandler : System.WebForms.UI.Page
{
    public AspCompatHandler(RequestContext requestContext)
    {
        this.RequestContext = requestContext;
    }

public RequestContext RequestContext { get; set; }

protected override void OnInit(EventArgs e)
{
    string requiredString = this.RequestContext.RouteData.GetRequiredString ↵
     ("controller");
    var controllerFactory = ControllerBuilder.Current.GetControllerFactory();
    var controller = controllerFactory.CreateController(this.RequestContext, ↵
        requiredString);
        if(controller == null)
            throw new InvalidOperationException("Could not find controller: " ↵
            + requiredString);
        try
```

```
            {
                controller.Execute(this.RequestContext);
            }
            finally
            {
                controllerFactory.ReleaseController(controller);
            }
            this.Context.ApplicationInstance.CompleteRequest();
        }
    }
```

在这个类中，我们通过重写OnInit()方法来查找和执行要处理请求的控制器。这模拟了标准的MvcHandler类的基本行为，即路由请求。我们也可以重写Page类的ProcessRequest()方法，以确保这个方法不会被意外调用。

一旦创建完了基本的路由类，就需要给这个类添加IHttpAsyncHandler的实现代码。实现这个接口是ASP.NET MVC允许使用AspCompatHandler路由处理器的一种方式。IHttpAsyncHandler包含两个要实现的方法，即ProcessRequest()和EndProcessRequest()，它们指示ASP.NET使用AspCompat指令来处理请求消息，代码如下：

```
public IAsyncResult BeginProcessRequest(HttpContext context, AsyncCallback cb, ↵
object extraData)
{
        return this.AspCompatBeginProcessRequest(context, cb, extraData);
}

public void EndProcessRequest(IAsyncResult result)
{
        this.AspCompatEndProcessRequest(result);
}
```

由以上代码可以看出，使用IHttpAsyncHandler方法只需要简单地传递请求消息给AspCompat-BeginProcessRequest()方法，从AspCompatEndProcessRequest()方法接收返回结果即可（都是通过Page类暴露的）。

既然已经编写了路由处理器代码，最后的工作就是把RouteHandler附加到路由定义代码中，代码如下：

```
context.MapRoute("AspCompatRoute", "{controller/{action}",
                    new{ controller = "Home", action = "Index"}
                    ).RouteHandler = new AspCompatHandler();
```

如果现在允许相同的代码来查询当前的ApartmentState（套间状态），就会看到控制器是在单线程套间模式下执行的。

而且，因为请求是运行在单线程套间线程中的，所以COM没有必要创建自己的单线程套间线程来执行COM组件调用。此外，因为COM组件寄宿在各自创建的单线程套间中，所以彼此之间是独立执行的，允许真正意义上的并行执行。

总结

Summary

首先，本章介绍了路由的概念以及如何在ASP.NET MVC框架中使用路由。通过介绍Web网站的URL作用，从使用角度和SEO角度逐步阐述了网站URL的重要性。

然后，介绍了如何使用`RouteTable.MapRoute()`方法来创建新路由，也就是向静态`RouteTable`字典对象里添加路由，以及如何构造不同的路由URL。

接下来，介绍了如何通过路由约束来控制用户提交给网站的数据，以及如何为网站创建自定义路由约束。

最后，介绍了如何使用基于属性标记的路由创建方法，同样也介绍了ASP.NET MVC路由管道中的扩展点。现在应该可以理解并掌握ASP.NET MVC框架中的路由机制了。

第15章

可复用UI组件

Reusable UI Components

到目前为止，我们已经介绍了各种创建ASP.NET MVC应用程序可复用组件的方法。然而，这些方法仅仅允许创建单个项目中可复用的视图或操作。换句话说，相比真正的“复用”，需要更多的“共享”组件。不能在项目以外使用它们，只能通过“代码复用”（又称“复制/粘贴”）。本章将介绍如何创建真正的、可用做库的、在不同项目之间共享的可复用组件。

ASP.NET MVC框架提供了什么

What ASP.NET MVC Offers out of the Box

在深入探究创建跨项目的可复用组件之前，先快速查看ASP.NET MVC 框架能提供哪些功能。

部分视图

部分视图（partial view）允许创建可复用的内容。为了实现真正的可复用，部分视图应包含很少或没有逻辑功能的代码，因为部分视图代表了更大视图中的布局模块单元。部分视图保存在the/Views/Shared/文件夹下，但是带有.cshtml或.vbhtml扩展名。要呈现部分视图（Partial view），可以使用`@Html.Partial("_ partialViewName")`这些语法，如下：

```
@Html.Partial("_Auction")
```

HtmlHelper扩展或自定义HtmlHelper

自定义`HtmlHelper`扩展方法是指在`HtmlHelper`类上可以使视图输出纯净HTML的扩展方法。它与部分视图遵守的规则相同（例如，没有逻辑代码、表示小布局单元），但是更集中。一个常见的例子是`HTMLHelper`扩展，它呈现一个文本框和一个匹配的标签，通常是为了起辅助作用：

```
@Html.TextBoxAccessible("FirstName", @Model.FirstName)
```


这里是相应的扩展代码：

```
public static class HtmlHelperExtensions
{
    public static HtmlString TextBoxAccessible(this HtmlHelper html, string id, ↵
    stringtext)
    {
        return new HtmlString(html.Label(id)
                + html.TextBox(id, text).ToString());
```

```
        }
    }
```

显示和编辑模板

在ASP.NET MVC 2中开始引入了显示和编辑模板（display and editor template），它允许创建强类型视图（strongly typed view），例如，

```
@Html.DisplayFor(model =>model.Product)
```

显示和编辑模板位于控制器下(或在Views\Shared文件夹下)DisplayTemplates或EditorTemplates子文件夹的部分视图中。例如，在Views\Shared\DisplayTemplates或Views\Product\DisplayTemplates下创建名为Product.cshtml的部分视图，并且使用某种方式来显示“Product”标签，`@Html.DisplayFor(model⇒model.Product)`就会使用`DisplayTemplate`显示模板来呈现`Product`产品信息：

```
@model Product

@if (Model != null) {
   <!--渲染产品信息的标签-->
}
```

如果类型不能匹配模板，就会返回该对象的`.ToString()`结果。

因为绑定了某种特定的模型，所以这些模板可以包含一定的业务逻辑。因为绑定了强类型模型，所以这些模板在编译时还可以捕获错误，而不是在运行时捕获这些错误。

Html.RenderAction()

`RenderAction()`帮助方法先执行控制器操作，然后将HTML输出结果插入父视图中。因此，`RenderAction()`可以像布局一样允许复用逻辑功能。当布局要经常复用业务逻辑时，通常会使用HTML帮助方法。

319

更进一步

Taking It a Step Further

当在同一应用程序中复用组件时，前面讨论的方法可以运行得很好。当共享的视图在同一个文件夹下，或者在其他视图控制器的相同文件夹中，或者在该网站的共享文件夹Shared中时，多个控制器操作可以引用同一视图。不同的视图可以在自己的范围内调用自定义帮助方法复用表现层逻辑。但是，如何跨项目共享视图或组件呢？

在ASP.NET Web Form中，可以通过创建用户控件或自定义控件并编译成独立的程序集来实现这些目标。而且这些程序集来可以分布在项目中，从而使其可以在多个项目中复用。

ASP.NET Web Form视图引擎提供的`ViewUserControl`类可以用来为ASP.NET MVC框架创建此类组件。然而，在ASP.NET MVC中，Razor视图引擎没有提供此种方法。本节将介绍如何使用Razor API实现类似功能。

Razor单文件生成器

Razor视图最终会生成.NET代码，这些代码可以编译成程序集，而且这些编译的程序集肯定可以跨项目复用！然而，我们需要的工具就是可以直接在单独的项目中设计Razor视图，并根据视图调用Razor API生成.NET代码。尽管这一章给出了创建这种工具所需要的所有信息，但是其实并不需要我们亲自开发这种工具。开放源代码社区的某些开发者已经为我们创建了这种工具！

安装Razor单文件生成器

虽然CodePlex网站（译注1）上保存了完整的源代码，但是Razor单文件生成器安装程序在Visual Studio扩展库中找到，所以最简单的安装方法是从扩展库安装它。通过打开Visual Studio扩展管理器（...Tools→Extension管理器）并搜索在线库“Razor生成器”，如图15-1所示。

安装Razor生成器后（确保已重启Visual Studio），可创建一个新项目来共享视图。当然这个新项目没有任何特别之处。只需在原有的解决方案中创建一个新的类库（Class Library）项目（见图15-2），并将它命名为ReusableComponents即可。

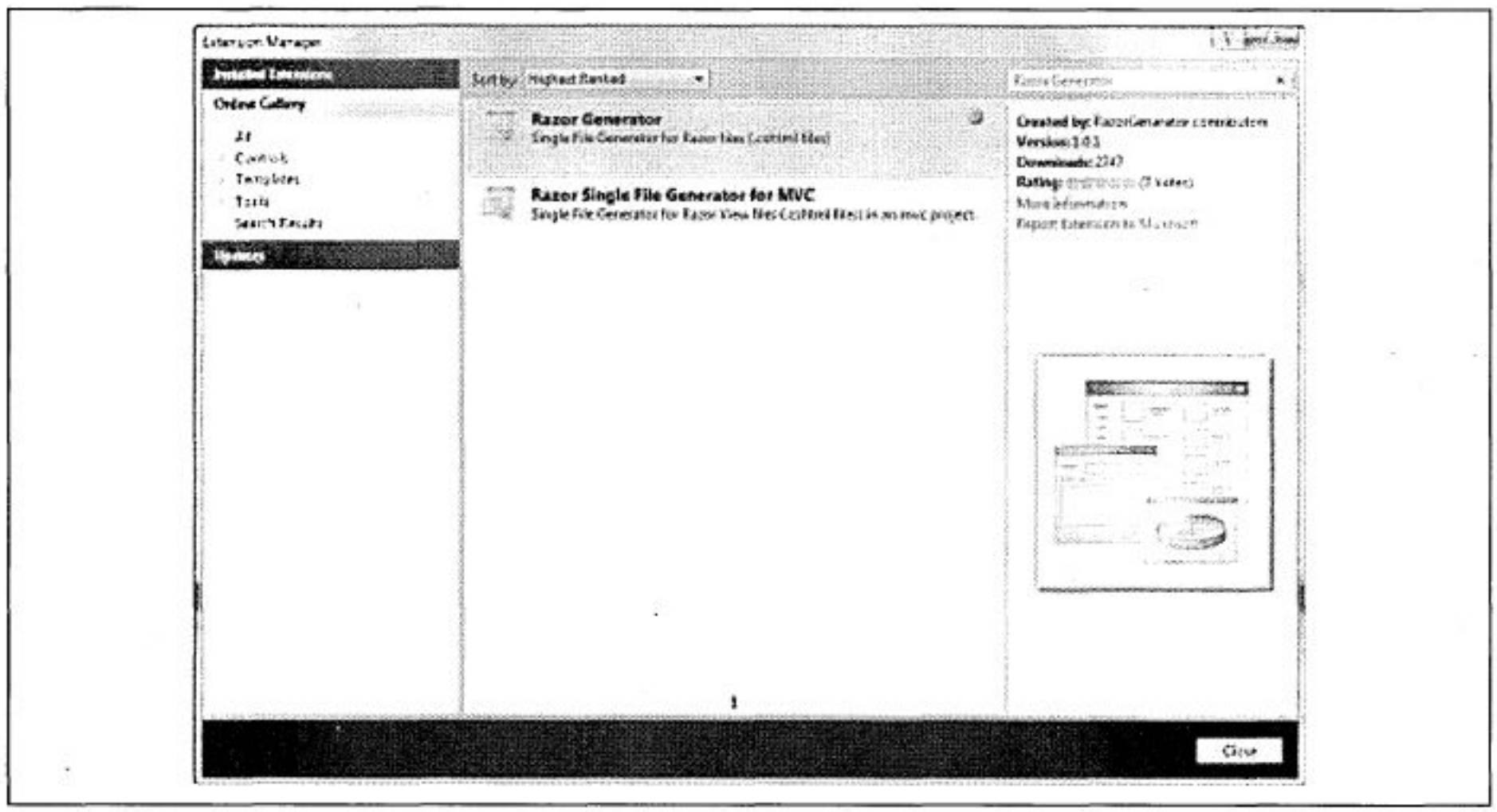


图15-1　在扩展管理器里安装Razor生成器

创建可复用的ASP.NET MVC视图

一种在项目之间共享且使用最广泛的视图就是通用的错误页面。所以，我们通过创建一个视图来了解Razor单文件生成器是如何处理ASP.NET MVC视图的。

译注1：CodePlex是微软最大的开源项目网站，MVC源代码就发布在这个网站上，CodePlex网址为：*http://www.codeplex.com/*。其中涉及诸多微软最新技术的开源项目，同时也可以建立自己的开源项目。

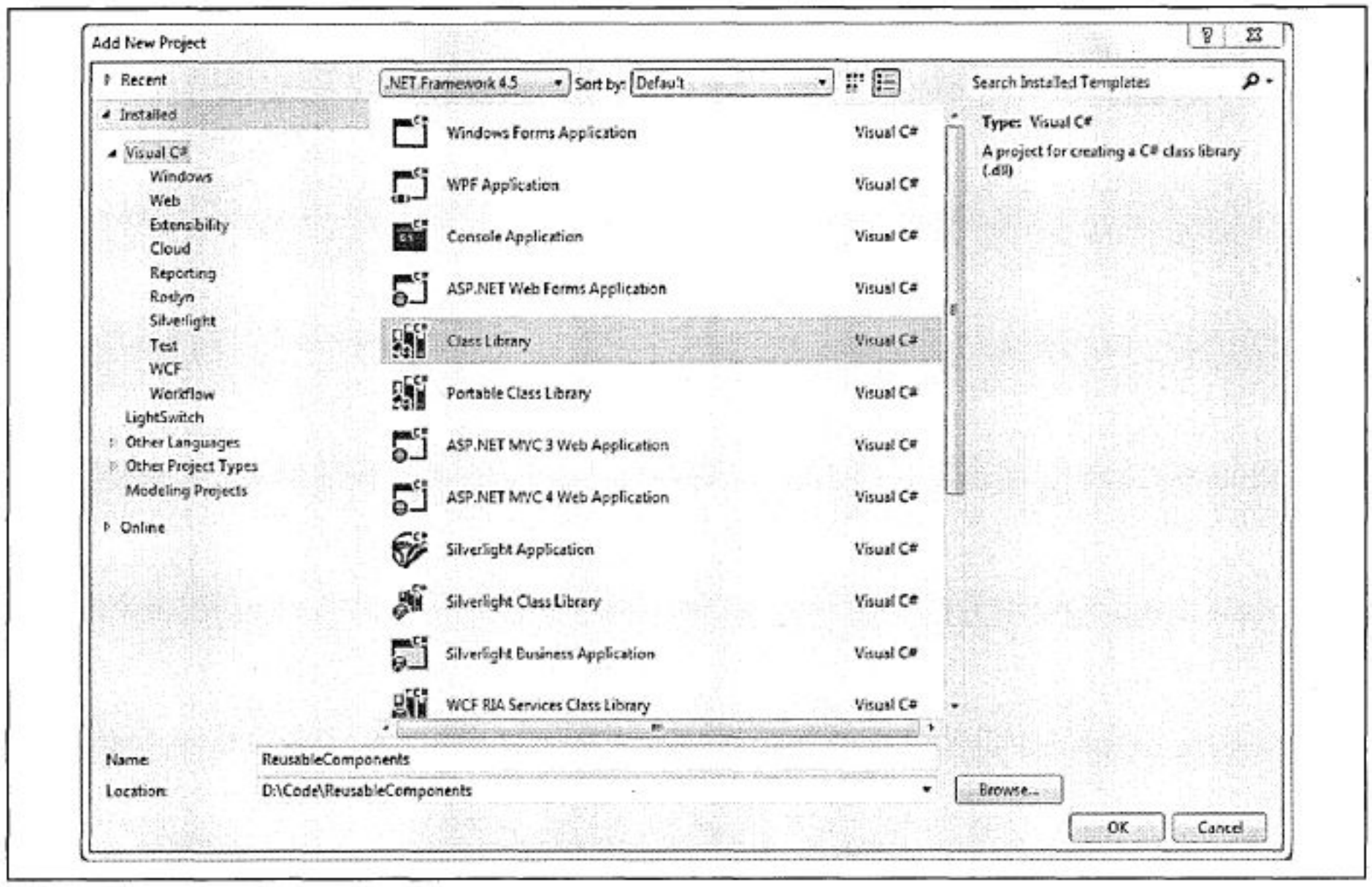


图15-2　为可复用视图创建新的类库项目

用Razor单文件生成器创建可复用的ASP.NET MVC视图与在ASP.NET MVC项目内创建视图的操作一样。当创建一个类似于ASP.NET MVC的~/Views文件夹结构时，要做的唯一事情就是将视图与Razor单文件生成器关联的视图自定义工具CustomTool属性设置为RazorGenerator。

因为新的ReusableComponents类库不属于ASP.NET MVC 项目，它没有~/Views文件夹，所以需要创建一个。要增加的新视图将跨多个控制器使用，因此类库的文件夹结构应该这样：直接在~/Views文件夹下创建一个命名为 Shared的文件夹，镜像ASP.NET MVC应用程序文件夹惯例。完成这些工作后，ReusableComponents类库应该看起来如图15-3所示。

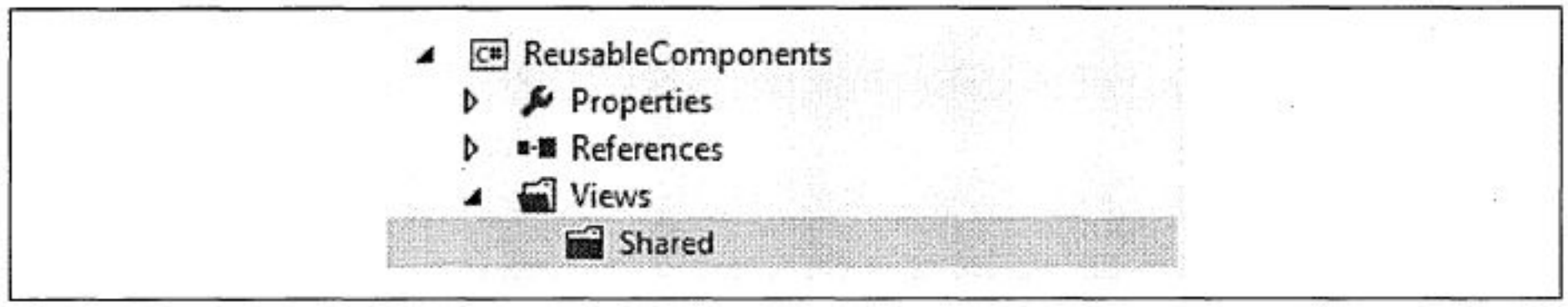


图15-3　带有~/Views文件夹结构的ReusuableComponents项目

文件夹结构创建完毕后，现在添加一个新的名为GenericError.cshtml的文件到共享文件夹，这可通过右键单击它并从上下文菜单中选择添加→新建项目（Add→New Item）选项来完成。由于该项目是一个类库项目而不是一个ASP.NET MVC项目，所以Visual Studio会拒绝显示的MVC 4 View Page (Razor)项目类型。没关系，可选择另一个纯内容文件类型，如文本文件或HTML文件。因为新创建的项目（GenericError.cshtml）带有cshtml扩展名，所以Visual Studio知道这是一个Razor模板。虽然Visual Studio可以识别新的Razor模板文件，但是还是需要告诉

Razor单文件生成器以便根据这个模板生成代码。为了启动生成器，还需要打开GenericError.cshtml文件的属性，并将其自定义工具CustomTool属性设置为RazorGenerator。图15-4展示了如何正确配置Razor生成器。

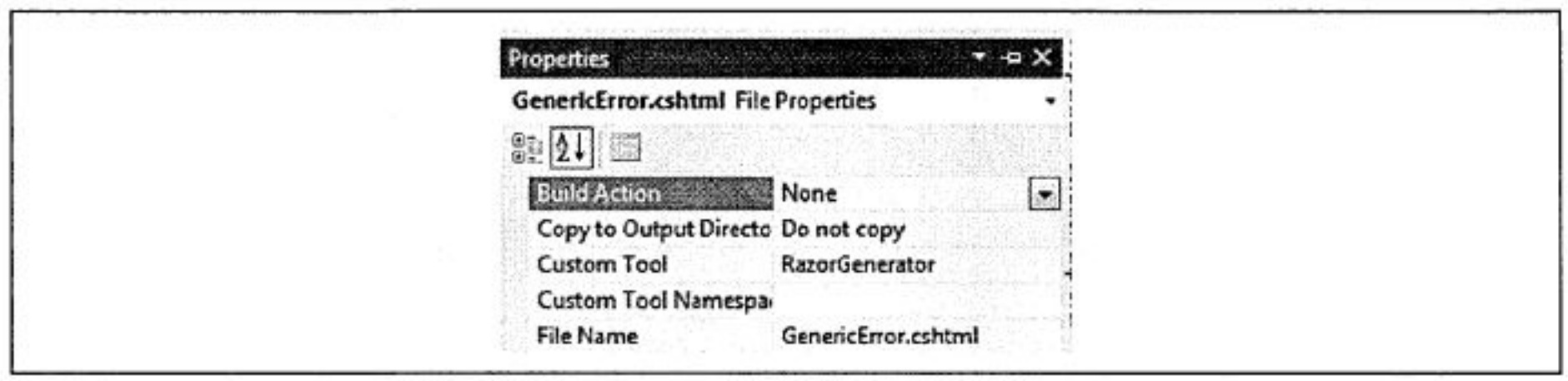


图15-4　设置RazorGenerator的自定义工具属性

使用下面的Razor标签来替换GenericError.cshtml文件中的所有内容：

```
@{ Layout = null; }
<html>
<head>
    <title>Website Error!</title>
    <style>
        body{ text-align: center; background-color: #6CC5C3; }
        .error-details .stack-trace { display: none; }
        .error-details:hover.stack-trace { display: block; }
    </style>
</head>
<body>
    <h2>we're sorry, but our site have encountered an error!</h2>
    <imgsrc="http://bit.ly/pjnXyE" />

@if (ViewData["ErrorMessage"] != null) {
    <div class="error-details">
        <h2>@ViewData["ErrorMessage"]</h2>
        <divclass="stack-trace">@ViewData["StackTrace"]</div>
    </div>
}
</body>
</html>
```

在指定自定义工具属性后，在GenericError.cshtml（见图15-5）视图文件下面可以看到Razor单文件生成器已生成的类文件GenericError.cs。

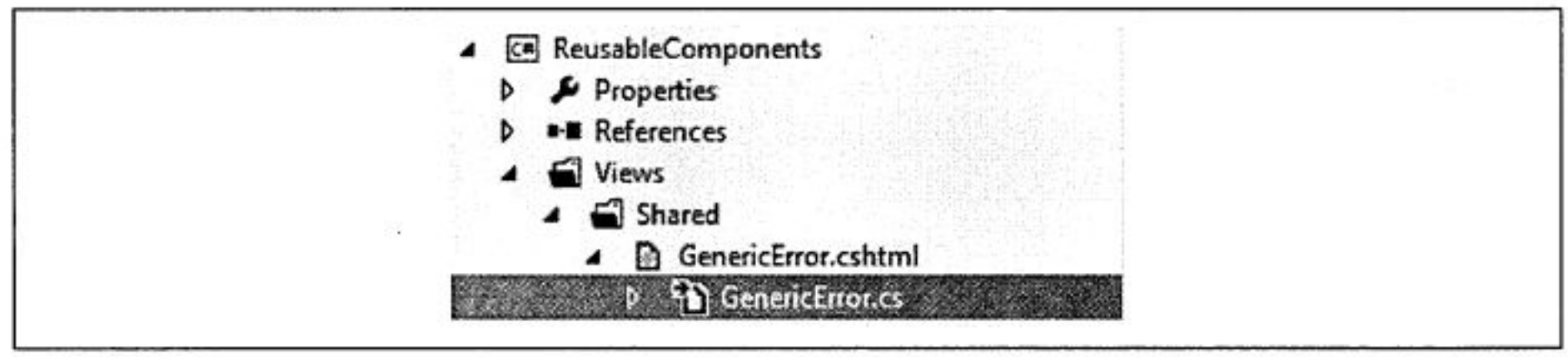
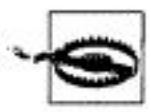


图15-5　通过Razor生成器生成的新文件



如果看不到所生成的文件，那么肯定是出问题了。请确保已正确拼写自定义工具的名称（RazorGenerator，中间无空格）。如果仍不能工作，就尝试从这一节的开始步骤重新操作。请确保安装Razor生成器工具后重新启动Visual Studio，检查所有安装日志并确保在安装过程中没有出现任何错误。

随意打开此新文件并检查其生成的内容。生成的代码就像任何其他代码一样，可以编译成一个程序集，共享给其他ASP.NET网站。

ASP.NET MVC Web应用程序中包含预编译视图

执行完这些步骤后，现在得到的是一个预编译的ASP.NET MVC Razor视图项目库。由于标准ASP.NET MVC Razor视图引擎使用的标准惯例，因此视图引擎在其标准搜索路径之外无法定位视图（ASP.NET MVC Web应用程序中的Views文件夹），不知道预编译视图是否存在，更不知道如何执行它们。现在该怎么办呢？

这个问题的答案是使用预编译MVC引擎PrecompiledMvcEngine来查找预定义视图。Razor单文件生成器的开发者创建的自定义视图引擎扩展了核心Razor视图引擎。

使用PrecompiledMvcEngine最简单的方法就是使用NuGet包管理器对该类库项目安装PrecompiledMvcEngine软件包，其中包含预定义视图。

PrecompiledMvcEngine包会向项目添加以下几个文件：

几个web.config文件

Razor API和Visual Studio Razor智能感知假设Razor视图存在于Web应用程序项目中，并从项目的web.config文件中读取其配置信息。即使项目是一个类库项目，PrecompiledMvcEngine包添加的web.config文件也为Visual Studio提供了足够的信息来启用Razor智能感知，甚至使用Razor单个文件生成器生成的视图。

Razor视图例子

PrecompiledMvcEngine包在项目的~/Views/Home 文件夹中添加了一个名为Test.cshtml的Razor例子视图，展示如何进行预编译视图配置。如果一切正常，则会看到这个视图立即生成一个代码后置文件（Test.cs）。Test.cshtml视图只是一个引用，以便我们可以按照自己的意愿修改、重命名或者删除它。

~/App_Start/PrecompiledMvcViewEngineStart.cs

PrecompiledMvcViewEngineStart.cs文件中包含了逻辑代码，这些代码可以告诉ASP.NET MVC应用程序为此类库项目中所有预编译的Razor视图使用PrecompiledMvcEngine引擎。PrecompiledMvcViewEngineStart.cs 文件还包括WebActivatorPreApplicationStartMethod标记属性，它会告诉WebActivator类库在启动Web应用程序时，执行PrecompiledMvcViewEngineStart.Start()方法，在ViewEngines集合中注册PrecompiledMvcEngine引擎。PrecompiledMvcViewEngineStart.cs文件的内容如下：


```
[assembly: WebActivator.PreApplicationStartMethod(
    typeof(ReusableComponents.App_Start.PrecompiledMvcViewEngineStart),
    "Start"
)]
```

```
public static class PrecompiledMvcViewEngineStart{
    public static void Start() {
        var currentAssembly= typeof(PrecompiledMvcViewEngineStart).Assembly;
        var engine = new PrecompiledMvcEngine(currentAssembly);
        ViewEngines.Engines.Insert(0, engine);
        VirtualPathFactoryManager.RegisterVirtualPathFactory(engine);
    }
}
```

一旦PrecompiledMvcViewEngineNuGet包安装完毕，而且已经把~/Views/Home/Index. cshtml从例子博客网站移到ReusableComponents类库项目中，就应该可以运行这个网站来检查是否所有的功能都像之前一样运行正常了。ASP.NET MVC会从类库中执行预编译的Index. Cshtml文件，而不会在意本地文件夹~/Views里是否存在这个文件。预编译MVC视图引擎PrecompiledMvcViewEngine如何知道渲染哪个视图呢？

我们已经知道如何在ASP.NET MVC应用程序中通过PrecompiledMvcViewEngine渲染预编译的Razor视图。PrecompiledMvcViewEngineStart会负责为网站注册PrecompiledMvcViewEngine预编译MVC视图引擎。现在万事俱备只欠东风，即定位预编译视图。虽然PrecompiledMvcViewEngine预编译MVC视图引擎同样遵守ASP.NET MVCViews文件夹惯例，使用相对路径来定位视图文件，但是这存在误区。其实PrecompiledMvcViewEngine预编译MVC视图引擎不会查找物理文件——它只会查找System.Web.WebPages.PageVirtualPathAttribute标记属性，Razor单文件生成器会为每个它生成的视图添加标记属性，这个标记属性会包含视图的相对文件路径。

下面展示了视图例子Test.cshtml中的前几行代码，其中包括PageVirtualPathAttribute标记属性：

```
[System.Web.WebPages.PageVirtualPathAttribute("~/Views/Home/Test.cshtml")]
public class Test : System.Web.Mvc.WebViewPage<dynamic>
```

因为虚拟路径名称是相对的，所以不论~/Views/Home/ Test.cshtml视图是否存在于ASP.NET MVC应用程序或类库项目中，它的虚拟路径是相同的。因此，在ASP.NET MVC应用程序请求Home控制器的Test视图时，PrecompiledMvcViewEngine知道使用注册的虚拟路径~ /Views/Home/Test.cshtml来查找预编译的Test.cshtml视图文件。

确保已将PrecompiledMvcEngine包添加到该类库项目，其中包含预编译的视图，而不是ASP.NET MVC Web应用程序项目。Web应用程序在运行时需要PrecompiledMvcViewEngine预编译MVC视图引擎程序集，但是NuGet包安装到类库项目中的内容只是表示类库项目包含了预编译的Razor视图。

325

创建可复用ASP.NET MVC Helpers

如果模板保存在ASP.NET MVC应用程序的App_Code文件夹中，也可以对Razor模板使用包括Razor Helper帮助方法的Razor单文件生成器来生成类似的结果。

Razor单文件生成器希望Razor帮助器模板存在于~/Views/文件夹中，所以在创建任何Helpers帮助类以前，必须要创建这个Helpers文件夹。在创建Helpers文件夹后，再按照早些时候创建Razor模板文件的步骤去创建新的Helpers文件夹，命名为TwitterHelpers.cshtml。最后就像为

ASP.NET MVC视图模板的设置赋值一样，将自定义工具Custom Tool属性设置为RazorGenerator。

设置属性之后，就可以立刻看到自动生成的文件TwitterHelpers.cs。打开文件可看到Razor生成器已成功解析空Razor模板和生成一个C#类，这个类准备用来封装一些帮助方法。

一个空类对我们来说没有任何好处，因此要创建一个帮助方法来使用标准的Razor语法，如下：

```
@helper TweetButton(string url, string text) {
    <script src="http://platform.twitter.com/widgets.js" type="text/javascript">
    </script>
    <div>
        <a href="http://twitter.com/share" class="twitter-share-button"
            data-url="@url" data-text="'@text'">Tweet</a>
    </div>
}
```

保存该文件并切换回生成TwitterHelpers.cs文件，表明这个文件已再次更新。现在的静态帮助类包含了自定义的TweetButton帮助类的代码。例如，例15-1包含完整的自动生成的代码（为了更好的可读性，已经删除了评论和空白字符）。

例15-1　自动生成的MvcHelper代码

```
namespace ReusableComponents.Views.Helpers
{
    using System;
    using System.Collections.Generic;
    using System.IO;
    using System.Linq;
    using System.Net;
    using System.Text;
    using System.Web;
    using System.Web.Helpers;
    using System.Web.Mvc;
    using System.Web.Mvc.Ajax;
    using System.Web.Mvc.Html;
    using System.Web.Routing;
    using System.Web.Security;
    using System.Web.UI;
    using System.Web.WebPages;

     [System.CodeDom.Compiler.GeneratedCodeAttribute("RazorGenerator", "1.1.0.0")]
    public static class TwitterHelpers
    {
    public static System.Web.WebPages.HelperResult
                        TweetButton(string url, string text) {
        return new System.Web.WebPages.HelperResult(__razor_helper_writer=> {
            WebViewPage.WriteLiteralTo(@__razor_helper_writer,
                "<script src=\"http://platform.twitter.com/widgets.js\" "+
                "type=\"text/javascript\">" +
                "</script>\r\n");

            WebViewPage.WriteLiteralTo(@__razor_helper_writer,
                "<div>\r\n" +
```

326

```
                "<a href=\"http://twitter.com/share\" "+
                "class=\"twitter-share-button data-url=\""
            );

            WebViewPage.WriteTo(@__razor_helper_writer, url);

            WebViewPage.WriteLiteralTo(@__razor_helper_writer, "\" data-text=\"\'");

            WebViewPage.WriteTo(@__razor_helper_writer, text);

            WebViewPage.WriteLiteralTo(@__razor_helper_writer,
                "\'\">Tweet</a>\r\n" +
                "</div>\r\n"
            );
        });
    }
  }
}
```

自动生成类设置完成以后，引用ReusableComponents程序集的ASP.NET MVC网站就可以使用TweetButton帮助方法了，就像使用定义在网站的App_Code文件夹中任何其他帮助方法一样。例如：

```
@using ReusableComponents.Views.Helpers
<div>
    @TwitterHelpers.TweetButton(url, message)
</div>
```

单元测试Razor视图

Unit Testing Razor Views

许多最佳实践原则主张视图中尽可能地简短、简单的逻辑代码。无论如何，根据Razor MVC视图进行单元测试，某些情况下确实可以带来很多好处。

下面的代码段是一个ASP.NET MVC Razor视图的示例：

```
<p>
    Order ID:
    <spanid='order-id'>@Model.OrderID</span>
</p>
<p>
        Customer:
        @(Html.ActionLink(
                @Model.CustomerName,
                "Details", "Customer",
                new { id = @Model.CustomerID },
                null))
</p>
```

默认的 ASP.NET MVC Razor视图类暴露了一些视图依赖的属性，如Model、Html等。为了在ASP.NET MVC运行时外编译和执行视图，就必须创建并实现含有这些属性的自定义模板基类。下面的示例包含OrderInfoTemplateBase类的代码段，修改后已包含Model和Html属性，以便它可用于编译以前的视图：

```
public abstract class OrderInfoTemplateBase
{
    public CustomerOrder Model { get; set; }
    public HtmlHelper Html { get; set; }
}
```

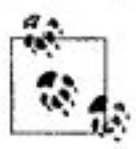
OrderInfoTemplateBase类现在拥有了ASP.NET MVC基类的模板依赖，这样它就成了ASP.NET MVC基类合法子类。引入自定义基类，如OrderInfoTemplateBase，可以提供给我们完全控制模板的属性和功能。自定义基类也需要在ASP.NET MVC运行时执行ASP.NET MVC视图。

例15-2展示了如何使用模拟对象来替换产品组件。

例15-2 使用模拟对象执行Razor模板实例的单元测试

```
public void ShouldRenderLinkToCustomerDetails()
{
   var mockHtmlHelper= new Mock<HtmlHelper>();
   var order = new CustomerOrder()
      {
         OrderID= 1234,
         CustomerName= "Homer Simpson",
      };

   // 创建实例并设置属性
   var template = (OrderInfoTemplateBase)Activator.CreateInstance(/*...*/);

   template.Html = mockHtmlHelper.Object;
   template.Model= customerOrder;

   template.Execute();

   // 检验生成的连接
   mockHtmlHelper.Verify(htmlHelper=>
      htmlHelper.ActionLink(
         order.CustomerName,
         "Details", "Customer",
         It.IsAny<object>()
      );
}
```


通过模拟实现代码来替换产品HtmlHelper类，单元测试可以根据视图中的代码轻易做出测试判断，验证视图中的代码正确性，而不需要依赖ASP.NET MVC运行时。



如果使用Razor单文件生成器来创建可复用视图，则不需要使用基于反射的方法，如Activator.CreateInstance()。由于Razor单文件生成器生成实际的类，所以需要做的是先创建类的新实例，例如vartemplate = new-CustomerOrderTemplate();，然后根据新实例运行测试。

注入模拟和存根对象（mock and stub object）（译注2）对单元测试来说非常有用。没有这些

译注2：stub objects 译为存根对象，它就是一个共享对象，提供了与真实对象一样的功能接口，同时不包含任何代码或数据。不能在运行时使用存根对象，然而，可以根据一个存根对象创建应用程序代码，因为存根对象提供了可以在运行时使用的对象名称。可参考 *http://docs.oracle.com/cd/E23824_01/html/819-0690/chapter2-22.html*。

支持，绝大部分网站必须使用效率低下、不可靠的浏览器测试方式进行UI测试。注入模拟和存根对象允许开发人员创建在几毫秒内执行完毕的UI单元测试代码，这大大提高了测试效率。

总结

Summary

ASP.NET MVC中提供了许多创建可复用组件的方法。部分视图、显示和编辑模板、HTMLHelpers帮助方法/功能及`RenderAction()`提供了在单个项目中复用组件的简便方法。使用Razor API，也可能创建在项目之间共享的可复用组件。本章介绍了如何通过安装和使用Razor单文件生成器来创建可复用视图，还介绍了如何使用模拟对象来对视图进行单元测试。

第四部分

质量控制

Quality Control

第16章 日志

Logging

无论架构设计得多么完美，或者代码编写得多么强壮，软件Bug还是无法避免的。

为了尽量减少错误对网站的影响，需要开发人员像对待其他的重要特性一样对待错误处理和日志，即尽早在网站项目设计阶段就把错误处理和日志设计方案考虑进去。

本章将会介绍错误处理、日志以及用来提示性能的监控工具，并且会介绍这些工具的使用方法。

ASP.NET MVC中的错误处理

Error Handling in ASP.NET MVC

当网站忙于处理HTTP请求时，很多内容都可能出错。幸运的是，ASP.NET MVC让错误处理工作变得相对简单了很多。

因为ASP.NET MVC应用是运行在ASP.NET框架之上的，所以可以像Web Form网站一样访问底层框架的核心功能，包括自定义错误处理页面及显示错误状态码。

现在以EBuy电子商务交易网站作为异常错误的例子，来看看ASP.NET MVC Web网站是如何处理错误的。为了测试，要在`HomeController`控制器的`About`操作中添加一行抛出异常的代码，用来模拟抛出异常：

```
Public ActionResult About()
{
     ViewBag.Message= "Your quintessential app description page.";

     throw new Exception("Something went wrong!");
}
```

为了触发这个操作，需要在浏览器里输入*/home/about* URL地址，这样就可以看到标准的ASP.NET错误页面了（见图16-1）。

网站已经产生了异常，现在就来为网站添加错误处理代码。

启用自定义错误

ASP.NET MVC应用中错误处理的第一步与其他的ASP.NET应用一样，即启用自定义错误特性。

Server Error in '/' Application.

Something went wrong!

Description: An unhandled exception occurred during the execution of the current web request. Please review the stack tr code.

Exception Details: System.Exception: Something went wrong!

Source Error:

```
Line 20:            ViewBag.Message = "Your quintessential app description page.";
Line 21:
Line 22:            throw new Exception("Something went wrong!");
Line 23:
Line 24:            return View();
```

Source File: C:\Work\Training\MVCBook\MyMvc\MyMvc\Controllers\HomeController.cs **Line:** 22

图16-1 标准的ASP.NET错误页面

这个特性提供了三种模式。

`On`
: 启用自定义错误处理功能。当发生错误时，可显示不同的自定义错误页面。

`Off`
: 关闭自定义错误处理功能。无论发生什么错误，都显示默认的错误诊断页面。

`RemoteOnly`
: 启用自定义错误处理功能，但是只对来自远程机器的请求有效。如果从托管网站的本地机器上访问，则会看到详细的错误诊断信息来帮助我们调试网站。用户仍将看到自定义错误页面。

为了启用自定义错误，只需要在web.config配置文件里把`system.web>customErrors`的`mode`属性值设置为`On`或`RemoteOnly`即可：

```
<customErrors mode="On" defaultRedirect="GenericErrorPage.htm">
   <error statusCode="404" redirect="~/error/notfound"></error>
</customErrors>
```

设置完毕后，下一步就是加强默认的ASP.NET MVC错误处理体验。

控制器操作中的错误处理

虽然启用自定义错误处理功能可以在网站发生错误时显示自定义错误页面，但是有时候只简单显示自定义错误信息是不够的。

对这种情况，ASP.NET MVC提供了`HandleErrorAttribute`标记属性，提供了对操作级别发生错误更细粒度的控制。

`HandleErrorAttribute`暴露了两个属性。

`ExceptionType`
: 要处理的异常类型。

View

发生该异常时要显示的视图名称。使用方法非常简单，只要在相应的控制器操作方法上标记并设置要捕获的异常类型即可。

例如，当Auction操作抛出数据库异常(System.Data.DataException)时，ASP.NET MVC就会显示DatabaseError视图，代码如下：

```
[HandleError(ExceptionType = typeof(System.Data.DataException),
View = "DatabaseError")]]
public ActionResult Auction(long id)
{
     var db= new EbuyDataContext();
     return View("Auction", db.Auctions.Single(x =>x.Id== id));
}
```

与其他绝大部分操作标记属性一样，HandleErrorAttribute也可以应用到控制器级别，这样该控制器所有的操作方法都会使用相同的错误处理机制。

```
[HandleError(ExceptionType = typeof(System.Data.DataException),
View = "DatabaseError")]]
public class AuctionsController: Controller
{
     /*设置了 HandleError 标记属性的控制器*/
}
```

定义全局错误处理器

如果想要比控制器级别更高层次的控制，则可以使用HandleErrorAttribute为整个网站注册全局错误处理器。

为了注册全局错误处理器，打开/App_Start/FilterConfig.cs文件，然后找到RegisterGlobalFilters()方法。这样就可以看到ASP.NET MVC已经在GlobalFilterCollection全局过滤器集合中注册了HandleErrorAttribute。

同理，为了注册自定义逻辑，只要把自定义过滤器注册到全局过滤器集合中即可，代码如下：

```
public static void RegisterGlobalFilters(GlobalFilterCollection filters)
{
     filters.Add(new HandleErrorAttribute
     {
          ExceptionType= typeof(System.Data.DataException),
          View = "DatabaseError"
     });
        filters.Add(new HandleErrorAttribute());
}
```

记住，默认情况下，全局过滤器会按照它们注册的顺序执行，所以一定要确保在其他错误过滤器（error filter）之前注册特定异常类型的错误过滤器，比如更加宽泛的错误过滤器（如上所示）。

还有一个方法，就是通过filters.Add()方法的order参数来设置过滤器执行的顺序，例如，

```
public static void RegisterGlobalFilters(GlobalFilterCollection filters)
{
    filters.Add(new HandleErrorAttribute
    {
        ExceptionType= typeof(System.Data.DataException),
        View = "DatabaseError"
    }, 1);

    filters.Add(new HandleErrorAttribute(), 2);
}
```

由以上代码可以看到，注册过滤器时添加了数值1和2，这样就可以确保自定义错误DatabaseError过滤器可以在其他错误过滤器之前执行。

自定义错误页面

启用了自定义错误过滤器后，全局的HandleError错误过滤器就会拦截错误，而且重定向到错误页面（见图16-2）。

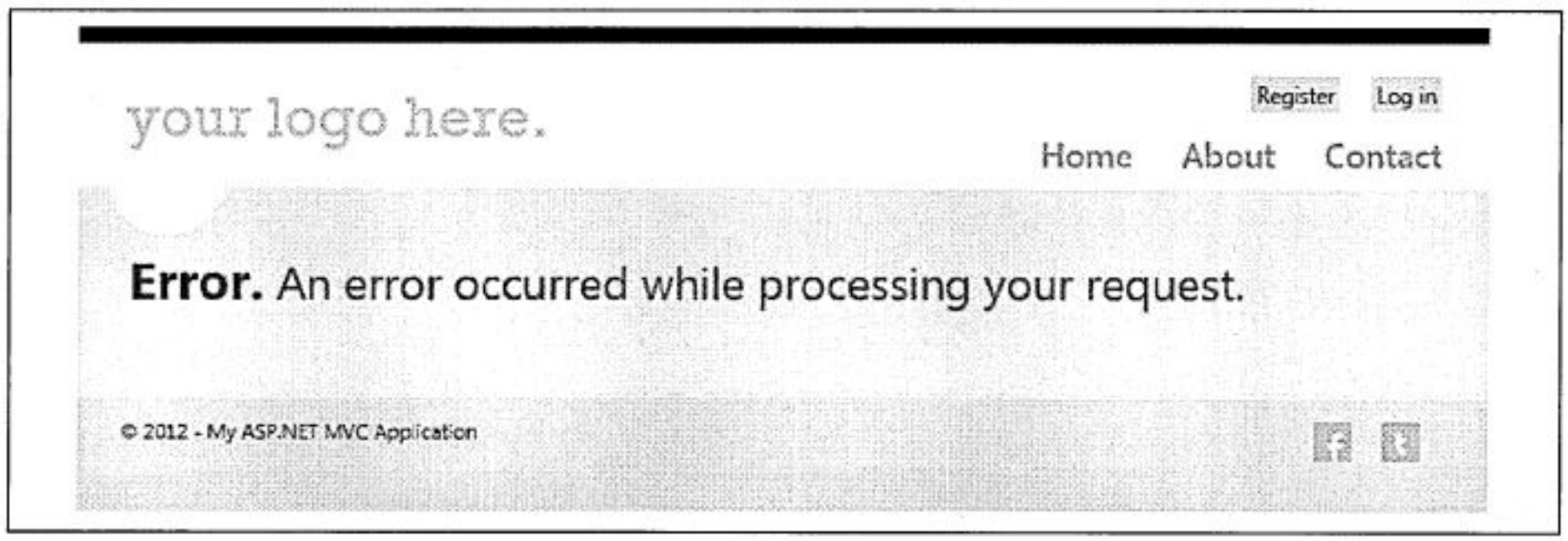


图16-2 自定义错误页面

只能处理ASP.NET MVC管道抛出的500种错误（异常）；还需要为其他类型的HTTP错误自定义错误规则，比如404错误。

如果启用了自定义错误功能，而且使用的是HandleErrorAttribute，ASP.NET MVC运行时就会在当前请求的文件夹或共享视图文件夹里查找Error.chtml文件。这种情况下，会忽略默认的defaultRedirect重定向（GenericErrorPage.htm）和状态码重定向URI。

如果启用了自定义错误功能，但没有使用HandleErrorAttribute，ASP.NET MVC运行时就会重定向用户到web.config里设置的defaultRedirect页面上。可以调用Global.asax.cs文件里的FilterConfig.RegisterGlobalFilters(GlobalFilters.Filters)方法来演示这种机制。

ASP.NET MVC项目模板包含默认的错误页面(*~/Views/Shared/Error.cshtml*)，而且可以自定义网站的错误页面。默认的HTML代码如下：

```
@model System.Web.ASP.NET MVC.HandleErrorInfo

@{
    ViewBag.Title = "Error";
}
```

```
<hgroup class="title">
    <h1 class="error">Error.</h1>
    <h2 class="error">处理请求时发生了错误 </h2>
</hgroup>
```

默认的错误页面非常简单，所以需要为自己的网站定义自己的内容。例如，有时候我们需要为用户提供联系网站的技术支持来帮助解决问题。

默认的错误页面是强类型视图，使用了HandleErrorInfo模型类。这个类暴露了一些包含异常信息的属性，包括发生错误的控制器操作信息。

日志和跟踪

Logging and Tracing

当网站发生错误时，就需要尽可能多的信息来帮助跟踪、调试错误。虽然显示错误信息是一种很友好的通知用户的方式，但是它并不能帮助开发人员处理问题。

为了能找出网站出错的根本原因，必须让网站记录一些必要的信息以及引起错误时进行的操作。这个记录通常称为**日志**（logging），日志可能是帮助我们调试网站最重要的工具。

记录错误日志

网站中可以有多种方式启用日志记录异常信息。

简单的日志帮助方法

下面的例子使用了自定义日志帮助方法调用Logger类。这个类会把异常日志记录到本机系统的事件日志（event log）中，如果想在本机上进行代码测试，那么首先要为应用创建一个事件源（event source），代码如下：

```
public class Logger
{
    public static void LogException(Exception ex)
    {
        EventLog log = new EventLog();
        log.Source= "Ebuy";
        log.WriteEntry(ex.Message);
    }
}
```

最简单的注册事件源方法就是使用regedit.exe来为EBuy网站在HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\Eventlog\Application里添加一个注册表入口地址。

简化try/catch处理器

第一选择就是把try/catch放到网站控制器操作方法代码里，具体示例代码如下：

```
public ActionResult About()
{
    try
    {
        ViewBag.Message= "Your quintessential app description page.";
        throw new Exception("Something went wrong!");
    }
    catch(Exception ex)
    {
        LogException(ex);
    }

    return View();
}
```

这种方法需要为每个控制器方法添加重复的代码。这种情况是应该尽量避免的，除非想单独处理某种特定的异常类型或特别记录日志。

重写Controller.OnException()

使用这种方式可以重写每个控制器的OnException()方法，而不是重复添加try/catch代码块。例如，

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

      LogException(filterContext.Exception);

      if(filterContext.HttpContext.IsCustomErrorEnabled)
      {

         // 如果启用了全局错误过滤器，就不需要下面的代码
         filterContext.ExceptionHandled= true;
         this.View("Error").ExecuteResult(this.ControllerContext);
      }
}
```

一个更好的方法是创建controller基类，提供单一的记录日志方法。当重写这个方法时，应该确保传递进去的上下文是非Null的，并且异常标记成“已处理”。如果没标记异常为“已处理”，那么ASP.NET MVC管道会继续抛出这些异常。

如果使用了HandleError全局过滤器，就应该从代码中删除标记异常为“已处理”的代码，这样就可以显示错误视图，因为全局过滤器会处理这些异常。

自定义错误过滤器

ASP.NET MVC应用中的另外一个处理错误的方法就是创建自定义错误过滤器（custom error filter）。自定义错误过滤器允许在某个位置定义处理错误的逻辑代码，然后使用到整个网站中，这样就可以减少重复代码的数量。自定义错误过滤器同样也允许控制器关注于请求处理逻辑，而无须担心发生的异常信息。

为了创建自定义错误处理器，需要继承HandleErrorAttribute类，并且重写OnException()

方法。在记录了异常日志以后，就要检查是否启用了自定义错误处理，代码如下：

```
public class CustomHandleError: HandleErrorAttribute
{
    public override void OnException(ExceptionContext filterContext)
    {
        if(filterContext== null)
            base.OnException(filterContext);

        LogException(filterContext.Exception);

        if(filterContext.HttpContext.IsCustomErrorEnabled)
        {
            filterContext.ExceptionHandled= true;
            base.OnException(filterContext);
        }
    }

    private void LogException(Exception ex)
    {
        EventLog log = new EventLog();
        log.Source= "Ebuy";
        log.WriteEntry(ex.Message);
    }
}
```

ASP.NET健康监控

虽然记录事件日志是监控网站很好的开始，但是一种更好的选择是启用ASP.NET健康监控（ASP.NET health monitoring）功能。ASP.NET健康监控功能远远超出记录异常日志的范畴，而且还可以记录应用程序或请求生命周期内发生的事件。

ASP.NET健康监控系统监控以下事件：

- 应用程序生命周期事件，包括应用程序开始和停止的事件。
- 安全事件，例如，登录失败、URL授权请求。
- 应用程序错误，包括未处理的异常、请求验证异常、编译错误等。

ASP.NET健康监控可以在网站的配置文件web.config里通过`healthMonitoring`节点设置，这个节点包含三个子节点。

`eventMappings`
: 定义要监控的事件类型。

`providers`
: 定义可用的提供者。

`rules`
: 定义在事件和提供者之间的用来记录事件的映射关系。

ASP.NET健康监控配置代码如下：

```
<healthMonitoring enabled="true">
  <eventMappings>
    <clear />
    <!--记录所有错误事件-->
    <add name="All Errors"
         type="System.Web.Management.WebBaseErrorEvent"
         startEventCode="0"
         endEventCode="2147483647" />
    <!--记录应用程序开始和停止事件-->
    <add name="Application Events"
         type="System.Web.Management.WebApplicationLifetimeEvent"
         startEventCode="0"
         endEventCode="2147483647" />
  </eventMappings>
  <providers>
    <clear />
    <add connectionStringName="DefaultConnection"
         maxEventDetailsLength="1073741823"
         buffer="false"
         name="SqlWebEventProvider"
         type="System.Web.Management.SqlWebEventProvider" />
  </providers>
  <rules>
    <clear />
    <add name="All Errors Default"
         eventName="All Errors"
         provider="SqlWebEventProvider"
         profile="Default"
         minInstances="1"
         maxLimit="Infinite"
         minInterval="00:00:00" />
    <add name="Application Events Default"
         eventName="Application Events"
         provider="SqlWebEventProvider"
         profile="Default"
         minInstances="1"
         maxLimit="Infinite"
         minInterval="00:00:00" />
  </rules>
</healthMonitoring>
```


ASP.NET健康监控包含把记录信息保存到Microsoft SQL Server数据库、本地事件日志，以及通过Email通知管理员等几种不同的提供者。当然它也允许我们创建自定义的提供者来记录到其他的数据源上。

为了使用微软的SQL Server数据库健康监控提供者，需要在网站数据库的表中添加一些表。可以直接使用.NET命令里的aspnet_regsql.exe工具来实现。

既然已经启用了ASP.NET健康监控功能，现在就来修改之前的自定义错误过滤器，把异常信息保存到健康监控提供者设置的数据源里。

因为健康监控系统的System.Web.Management.WebRequestErrorEvent类没有任何公开的

构造函数，所以，必须创建一个自定义Web请求错误事件类，代码如下：

```
public class CustomWebRequestErrorEvent: WebRequestErrorEvent
{
        public CustomWebRequestErrorEvent(
            string message, object eventSource,
            int eventCode, Exception exception)
            : base(message, eventSource, eventCode, exception)
        {
        }

        public CustomWebRequestErrorEvent(
            string message, object eventSource, int eventCode,
            int eventDetailCode, Exception exception)
         : base(message, eventSource, eventCode,
                eventDetailCode, exception)
        {
        }
}
```

在创建完这个类以后，修改CustomHandleError以调用这个自定义Web请求错误类：

```
public class CustomHandleError: HandleErrorAttribute
{
    public override void OnException(ExceptionContext filterContext)
    {
        if(filterContext.HttpContext.IsCustomErrorEnabled)
        {
           base.OnException(filterContext);
           new CustomWebRequestErrorEvent(
                 "An unhandled exception has occurred.",
                 this, 103005, filterContext.Exception)
                 .Raise();
        }
    }
}
```

代码编写完毕，设置好并作为全局错误过滤器进行注册以后， ASP.NET MVC网站所有的异常信息都会被路由到ASP.NET健康监控系统中，并且日志被保存起来。

总结

Summary

当设计和构建Web应用时，要考虑如何处理错误，程序运行时要记录日志和监控事件，要优化应用程序以提升性能，这些都非常重要。

本章介绍了ASP.NET内置的错误处理、日志和健康监控的强大功能。我们可以使用ASP.NET MVC框架的这些功能来构建强壮的Web应用程序。

第17章

自动化测试

Automated Testing

目前已经介绍了架构模式和应用程序开发实践，例如，MVC架构模式、分离关注点、SOLID和其他内容，也介绍了如何让应用程序中的组件变得更加可复用和可维护，以及如何构建高质量的应用程序。

以上这些技术只关注了长远利益，短期内无法体现出价值。例如，为什么组件要可扩展的？应用程序第一次迭代过程中没有扩展这个组件该怎么办？就无法立即体现可扩展性的优势。

这些技术的真正价值在后续应用程序的生命周期里才开始体现出来。当应用程序发布以后，开发人员必须修改重大问题，添加新特性，还要降低影响现已发布并且正在运行的应用程序的风险。

但是，短期来看这些技术带来了价值。而且我们有办法利用它们带来的这些价值来改善应用程序的长期质量：通过使用自动化测试技术来测试组件。

本章将介绍如何测试应用程序，使用不同的工具和技术来编写并运行测试代码，检验应用程序代码是否符合系统设计的要求。还将探讨如何把这些概念应用到整个代码中，特别是如何测试ASP.NET MVC应用程序，从服务端的控制器和服务到运行在浏览器中的客户端代码。

测试的语义

The Semantics of Testing

软件开发关注于创建软件应用程序任务，这些程序通过执行任意数量的特定行为来解决问题。开发人员称这些特定的信息为“需求”。但是，在发布应用（也叫产品）之前，必须验证系统的这些行为是否符合预期的要求，也就是必须检查所编写的代码是否是高质量的，并且是可靠的。

人工测试

检验已经实现的功能最简单的方法就是：运行应用程序，模拟正常用户的操作行为。因为本章内容介绍的是测试相关的技术，所以，作为补充，我们会介绍这种测试方法——任何需要人工参与的验证测试工作——称为人工测试（manual testing）。当然这个方法也有很多弊端。

人容易犯错

人工测试是基于人的判断力的，而人特别容易犯错。

虽然人会对最终的结果做出判断，以决定是否正确实现了某项功能，但是绝大部分人的判断都是主观的。对于某些特殊情况，人是无法判断某些功能是否是工作正常的。

例如，尽管我们十分清醒，而且保持高度警惕、准备揪出哪怕最小的Bug，但是凭借肉眼的分辨能力能区分string类型的1和int类型的1吗？正如每个修改过Bug的开发人员所知道的，这些问题听起来无关痛痒，也就是鸡毛蒜皮的小事，但是可能会导致整个应用程序出错。正所谓，失之毫厘，谬以千里。

计算机更高效

人工测试的效率不高，因为人工测试应用程序时，不会实例化类并且调用方法，只会通过某种UI来与应用程序交互。

测试人员必须按照正常操作使用应用程序，若为了方便测试而随意修改程序，则可能带来很多问题，比如损害测试结果或者带来很多新的、潜在的Bug，要重现特定场景的简单测试，要包括很多必须遵守的步骤，且必须按照顺序执行，这让人工测试变得枯燥乏味，并增加了人工出错的概率。

人工测试费时

人工测试很耗费时间——测试人员可以把这些时间用来做其他更重要的工作。例如，开发人员实现某项功能后，他要在每次小的修改之后停下来执行一些潜在的复杂的步骤来验证这种变化。如果这些小修改无法工作，那么就要接着修改……修改……

又例如，同一个开发人员如何测试异常条件？这个异常条件本质上是很难重现的。

显然，计算机更适合执行这种类型的任务。总之，人工测试中的很多问题都可以采用自动化测试，可让计算机自动执行测试，完成测试任务。

自动化测试概述

自动化测试的想法就是编写可以测试其他软件的软件来执行测试工作，通过配置让计算机执行特定的测试工作以解决人工测试的弊端，即自动执行测试任务。使用自动化测试方法，人类仍然可以定义自己想要的测试工作内容——设置测试的期望结果——跟之前做的工作一样。人工测试与自动化测试最大的区别是，自动化测试一旦创建，就可以多次运行，无论何时，只要想要验证应用程序的功能，就可以立即进行测试。这不仅比人工测试更简单，而且可以更快地执行测试任务，大大节约了测试工作耗费的时间。

显然，自动化测试就是康庄大道。本章其余部分将会从不同层次介绍自动化测试，学习如何创建自动化测试项目，以及一些ASP.NET MVC应用程序测试的最佳实践。

自动化测试的级别

Levels of Automated Testing

除了可以显著降低测试时间外，自动化测试还可以测试特定的组件，例如，**被测试单元**（unit under test，有时候也称为被测试系统或SUT）。被测试单元是指被测试的质量、性能和可靠性的组件。

在软件开发领域，被测试单元可以指任意级别的软件架构，例如，方法、类、整个应用程序或者多个一起工作的应用程序。同样，创建自动化测试的多个集合就是个绝佳的主意，可以根据不同级别的软件架构来定义不同的测试集合。

根据不同的测试架构层次级别，每个测试又可以划分为不同的类别：单元测试（unit test）、集成测试（integration test）或验收测试（acceptance test），这些内容会在后续章节中介绍，每个类别都有相当明确的定义。

单元测试

单元测试就是用细致的验证条件来检验最低级别的应用程序功能。单元测试中，被测试单元为非常特别、低级别组件，比如单个类或类的单个方法。事实上，因为测试的是非常特别、低级别的功能，所以通常需要许多单元测试来验证某个被测试单元。

单元测试的目标就是要检验被测试单元的实际逻辑。简言之，失败的单元测试应说明代码中存在的Bug。

为了实现这个级别的要求，单元测试必须是自治的（atomic）、可重复的、独立的、快速的。没有满足这四个级别条件的测试不能当成真正的单元测试。

下面几节解释自动化测试中的概念。

自治的

单元测试应该关注于验证某项单一的功能。通常来说，这是类展示的单个行为，或者业务用例。大部分时间里，单元测试只关注于类中的某个方法（有时候也可能是某个方法的某种条件下的执行情况）。实际编程中，这相当于一些短测试和有意义的断言（`Assert.That([…])`）。

常见误区包括：

- 一个测试中包括几十行代码。
- 存在两个以上断言，特别是测试多个目标对象时。

可重复的

虽然测试环境包含许多复杂的依赖条件，如.NET框架，但无论何时何地，运行同一个单元测试代码都应该生成相同的结果。

测试不能依赖于任何外部的无法直接控制的环境。例如，不应该依赖于任何网络/互联网连接性、数据库访问、文件系统权限，或者当前时间（DateTime.Now）。

常见误区包括：

- 第一次执行成功，但是一些或全部的后续执行失败（反之亦然）。例如，也许会看到这样的提示："XYZTest必须在这个测试之前运行"。

独立的

作为前两个标准的扩展，单元测试应该不依赖任何其他的系统和单元测试，能够独立执行。也就是说，单元测试不应该假设其他测试已经运行或依赖外部系统（如数据库）处于特定的状态或者产生特定的结果。此外，单元测试不应该创建或留下一些垃圾数据影响其他测试。这不是说单元测试不可以与其他测试共享方法或类——事实上，是鼓励这种做法。其真正的含义是说，单元测试不应该假设其他测试之前已经运行完毕或随后会运行。这些依赖应该被明确的方法调用所取代，或者在测试代码中使用特定的代码完成，在单个测试之前和之后立即运行这些代码。

常见误区包括：

- 数据访问。
- 当禁用网络或者VPN测试失败。
- 没有运行外部脚本（包括build script）测试失败。
- 当配置修改或不正确时测试失败。
- 测试必须在特定的权限下执行。

快速的

如果满足了上面的所有条件，则所有的测试都会很快完成（例如，瞬间完成）。无论如何，所有的单元测试都应该瞬间执行完毕，显示说明仍然是有好处的。毕竟，自动化测试套件的一个主要好处就是可以立即获取代码质量的反馈。随着运行测试套件所需时间的增加，执行测试套件的频率也会下降，且大部分时间直接耗费在生成Bug和发现Bug的工作上。

常见误区包括：

- 独立测试耗时更长。

聪明的读者可能会注意到，上面的列表中可能被安排入一个可爱的小缩略词，像FAIR（公平）单词。虽然这可以帮助我们方便记住单元测试的四个特性，但是这里的排序是故意这么做的——基本上代表了各个特性的重要性。

下面是遵循以上原则的单元测试示例代码：

```
[TestMethod]
public void CalculatorShouldAddTwoNumbers()
{
    var sum = new Calculator().Add(1, 2);
    Assert.AreEqual(1+2, sum);
}
```

这个单元测试非常简单。只实例化了一个Calculator类，然后调用了Add()方法，传递了两个参数作为加数。接下来，测试代码使用了Assert.AreEqual()测试方法来设置断言的结果是否与期望的值一致，即对两个数字求和。

这个测试代码不仅简单易懂，而且遵守了前面介绍的单元测试的指导原则。

首先，它是自治的。它专注于验证类的行为Calculator.Add()方法（被测试的单元），而且测试代码非常简单。

其次，它是可重复的。这个测试代码无论什么时间、什么机器，无论运行多少次，都会生成一样的结果。

再次，它是独立的。这个测试没有任何先决条件，也不会修改测试环境。它不依赖于任何其他的测试代码。

最后，它非常快。如果Add()的测试代码需要执行几个小时（对两个数字求和，这是无法想象的），而这个测试只需几毫秒就可以生成正确的结果。

集成测试

与主要是验证特定类或方法的逻辑和功能的单元测试相反，集成测试验证的是两个或多个组件之间的交互。换句话说，集成测试应确保所有独立的系统模块一起协同工作以生成期望的结果，即一个可以正常工作的应用程序。

集成测试也有自己的缺点。因为它们关注于验证多个组件是否可以协同正常工作，所以，当测试组件的时候，就会增加隔离测试组件和外部世界的难度。这将带来一大堆的问题，通常会违反大多数的原则——即使不是上文所述的应用于单元测试的全部指导原则。

缓慢与脆弱也许是集成测试的主要缺点。这意味着集成测试不仅不会像单元测试这样频繁地执行，而且测试失败的概率更高（与应用程序逻辑不符的测试结果可以当成失败）。

单元测试失败时，肯定会指出代码中的Bug。集成测试恰恰相反，当测试失败时，意味着也许代码中存在Bug，也许是测试环境中的其他问题，例如，数据库连接或者异常测试数据。这些失败测试——通过一种有用的方式来告诉开发人员环境中存在错误——让开发人员分心当前的工作来解决这个问题，这通常会降低开发人员的效率。尽管集成测试存在缺点，但是它和单元测试一样重要，无可替代——各有千秋，侧重点不同。

假设我们千方百计要进行完美的集成测试工作，则应该通过单元测试套件来扩展测试代码覆盖率，然后使用集成测试套件（integration test suite）来进一步提升测试代码覆盖率（反之不行）。

验收测试

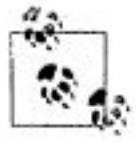
最后一种测试类型就是**验收测试**（acceptance test），它只有一个目标，即确保已经构建的系统符合最初的系统需求。简单来说，验收测试确保系统能做用户期望的所有工作。因为验收测试是根据定义——通常比较主观，而且很难自动完成测试工作。不过，有几种不同的技术，允许开发人员自动执行应用程序，以验证系统行为是否符合设计的预期。通过应用这些技术，开发人员能够避免单调乏味地手动测试应用程序，同时仍可以对应用程序进行可靠的自动化

测试，高效完成验收测试工作。

用户验收测试

本章主要关注于如何让开发人员测试自己的应用程序以验证是否工作正常，通过把软件转给用户，让测试软件来把用户验收测试（UAT）的概念引入软件开发过程中。用户验收测试是验收测试的子集。尽管不会对所有用户公布非最终发布版本的产品，但是，用户验收测试确实可以告诉我们这款软件是不是用户真正想要的东西。

另外，在开发阶段允许用户加入进来，甚至让用户参与测试部分实现的功能，可以帮助我们尽早暴露、发现相关系统技术和通信的问题。而且用户越早加入开发过程，就能越早发现并解决这些问题。所以，虽然用户验收测试也许无法轻易集成到默认的解决方案中，但是越早越频繁地执行测试，带来的好处就越大越多。



虽然有人会使用这三种测试类型中的一种，第一级别——单元测试和集成测试，通常来说非常具有技术性，并且需要特定的实现代码，这种代码一般由开发团队中的人来编写。在交付给其他测试团队之前，由开发团队验证系统是否满足了最初的设计需求（例如，质量保证团队（Quality Assurance）或QA团队），由QA团队验证系统是否符合业务需求。

350

什么是自动化测试项目？
What Is an Automated Test Project?

为了创建并执行本章讨论的自动化测试代码，需要创建一个测试项目（test project）来辅助完成这项工作。

在Visual Studio世界里，测试项目是一个相对普通的类库项目，由一组测试类组成（通常称为test fixture（译注1）），每个类都是普通的.NET类，包含一些测试方法。每个测试方法创建一个被测试的单元，然后通过使用测试API来验证断言（例如，属性的值是否和期望的值一致），通过执行测试组件来验证它们的行为。

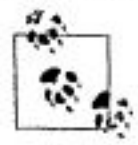
这种测试项目使用的方法——创建、执行并按照顺序执行验证，也被称为“Arrange-Act-Assert”模式（译注2）。

为了在测试项目中执行自动化测试，测试项目要经过编辑，然后传递给**测试运行器**（test runner），再自动在应用程序集里定位并执行所有的测试，跟踪记录所有的测试过程。当测试运行器执行每个测试时，它会记录测试过程中发生的所有事情，包括控制台或调试输出的测试结果。

译注1：test fixture 通常译为测试套具，是指用某个固定状态参数作基准来测试物体特性。其目的是确保固定的测试环境可以反复测试，而且测试结果可重复。有些人把这称为测试上下文环境（test context）（来自 Wiki 百科）。比如测试飞机的风洞、汽车碰撞实训室的设备等都是测试套具的例子。

译注2：“Arrange-Act-Assert”模式也称为 AAA 模式，是单元测试中组织测试代码的一种方式，提倡将方法按照功能分组，使用空白线分隔：（1）组织先决条件和输入数据；（2）测试对象行为；（3）断言测试结果。

当每个测试成功或者失败时，测试运行器会显示一个所有单元测试运行结果的概要报告，以方便我们快速查看测试结果。



接下来的例子是使用非Express版本的Visual Studio默认包含的工具和API。但是，如果使用Express版本的Visual Studio，那么，有很多种自动化测试工具和框架可以选择，而且很多都是开源的。虽然不一定提供了和Visual Studio一样强大的多级别测试工具，但是它们都遵守基本的测试流程，所以，本书介绍的测试思想和原则同样适用于其他的框架。

最好的办法就是对所有的测试工具进行评估，然后选择一种最适合自己团队的测试工具。我们直接在互联网上搜索“.NET unit testing”，就可以找到很多资料（译注3）。

创建Visual Studio测试项目

有几种方法可以创建Visual Studio单元测试项目（Visual Studio unit test project）。

第一种方法是，在图17-1所示的新建ASP.NET MVC 4项目（New ASP.NET MVC 4 Project）对话框中点击“创建单元测试项目（Create a unit test project）”，Visual Studio就会自动创建单元测试项目，并且把它添加到ASP.NET MVC网站解决方案中。

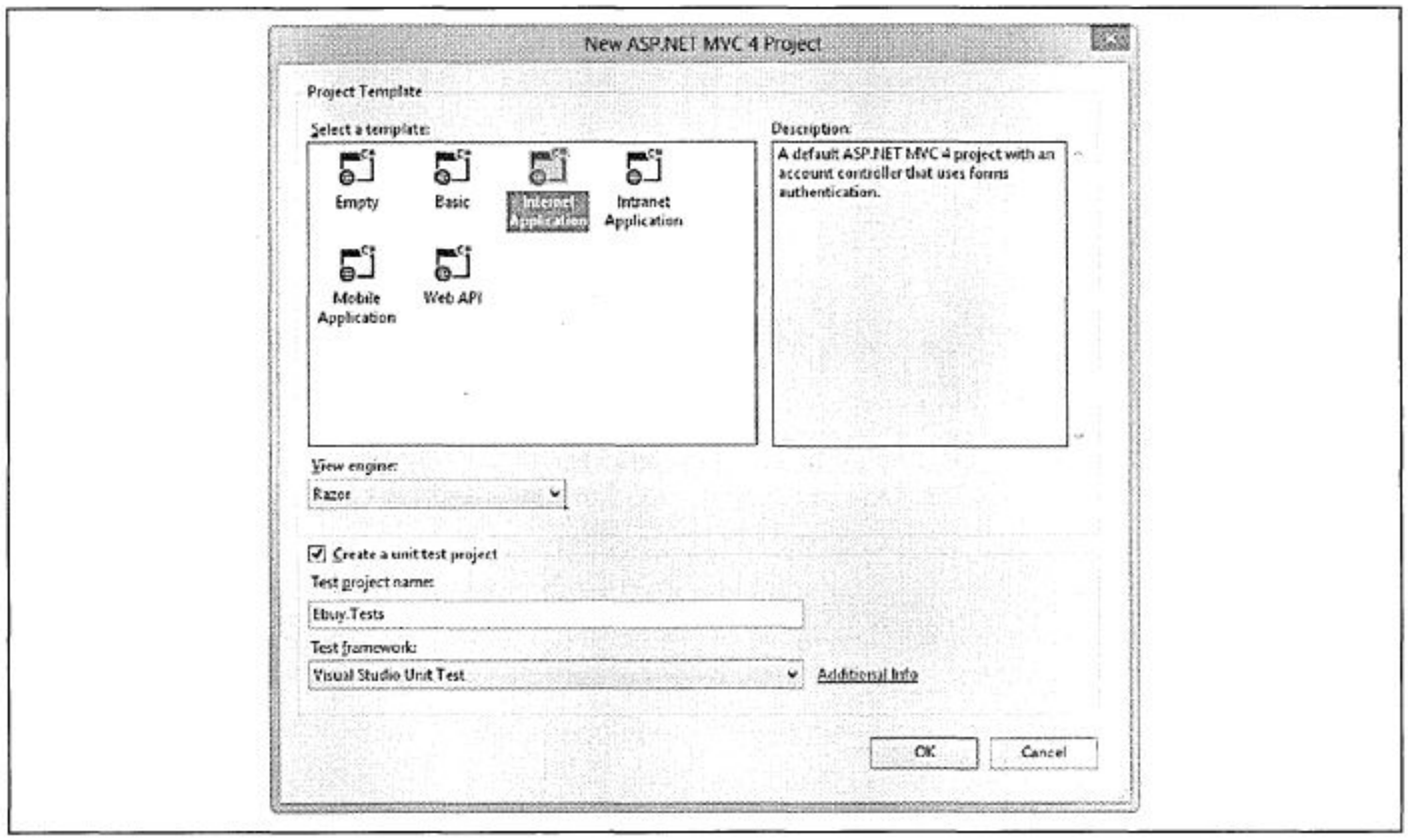


图17-1　在Visual Studio中创建新的单元测试项目

另外一种方法是，随时可以通过文件→添加→新项目...(File→Add→New Project...)方式向已有的解决方案添加新的单元测试项目（Unit Test Project），然后在测试类别里选择单元测试项目类型以及编程语言（见图17-2）。

译注3：我一直推荐使用 Google 搜索或 Bing 搜索查资料，这样查到的资料比较新、准确、权威。

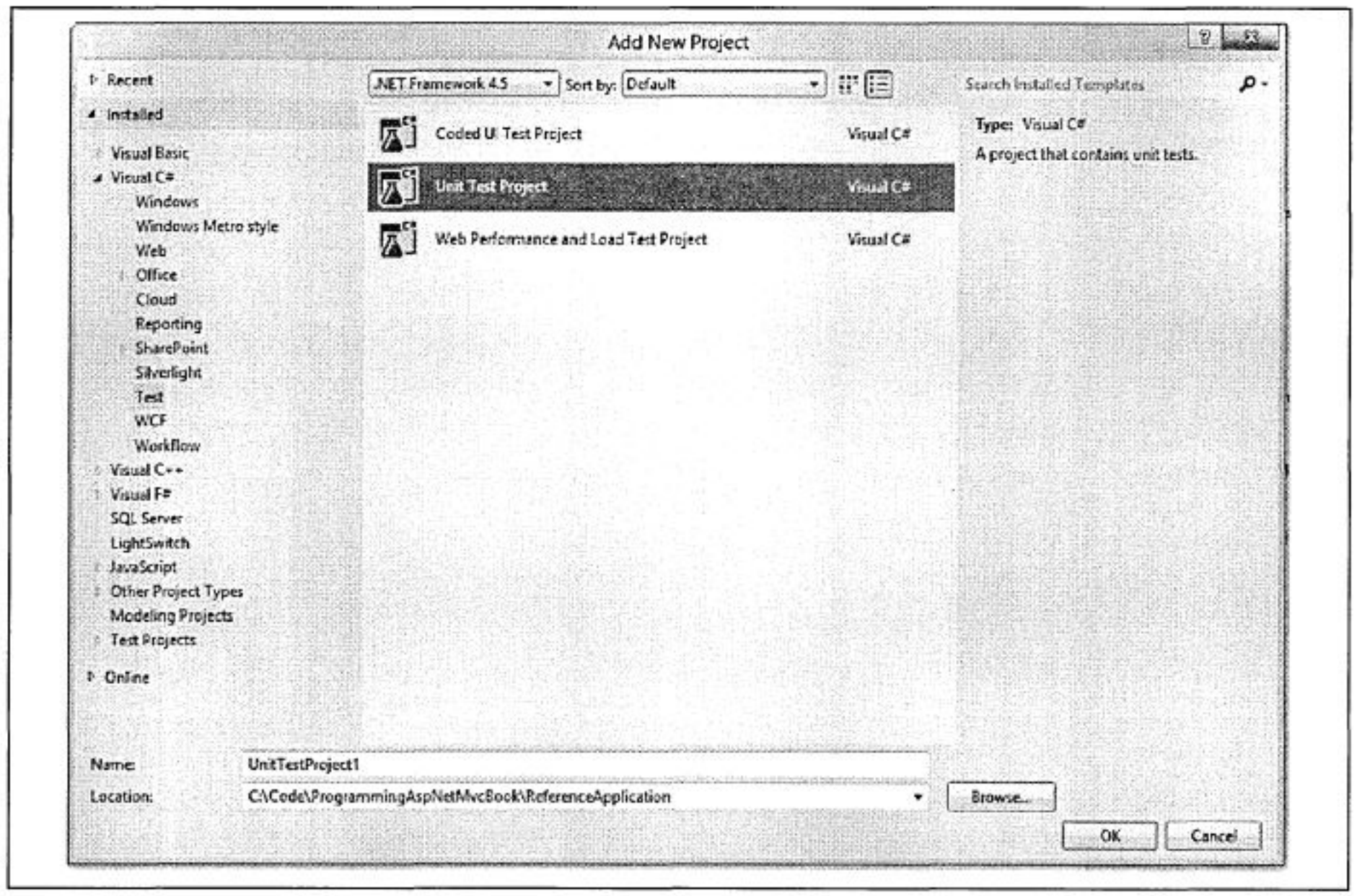


图17-2　向已有项目添加单元测试项目

以上两种方法都可以给解决（solution）方案添加新的单元测试项目，而且创建完毕之后，可以直接在单元测试项目里添加单元测试代码。

创建并执行单元测试

为了验证工作是否一切正常，可以右键单击单元测试项目，然后通过在菜单里选择添加→单元测试...(Add→Unit Test...)来添加新的单元测试代码。最后可以在创建的方法里添加测试逻辑代码，以确保使用了测试API的帮助方法来帮助我们检验测试结果是否正确。

例如，可以在下面的代码里输入测试逻辑，这样，测试类就变成了这个样子：

```
using Microsoft.VisualStudio.TestTools.UnitTesting;

namespace Ebuy.Tests
{
    [TestClass]
    public class UnitTestExample
    {
        [TestMethod]
        public void CanAddTwoNumbersTogether()
        {
            varsum = 1 + 2;
            Assert.AreEqual(3, sum);
        }
    }
}
```

当修改完代码以后，就可以用右键单击方法内的任何地方，然后选择“运行单元测试（Run Unit Tests…）”，或者使用Visual Studio快捷键Ctrl-R、T运行单元测试。如果需要编译Visual Studio，首先会编译单元测试项目，然后启动单元测试浏览器窗口（见图17-3）显示运行测试的状态。测试完毕，窗口中会显示测试的状态是否通过。

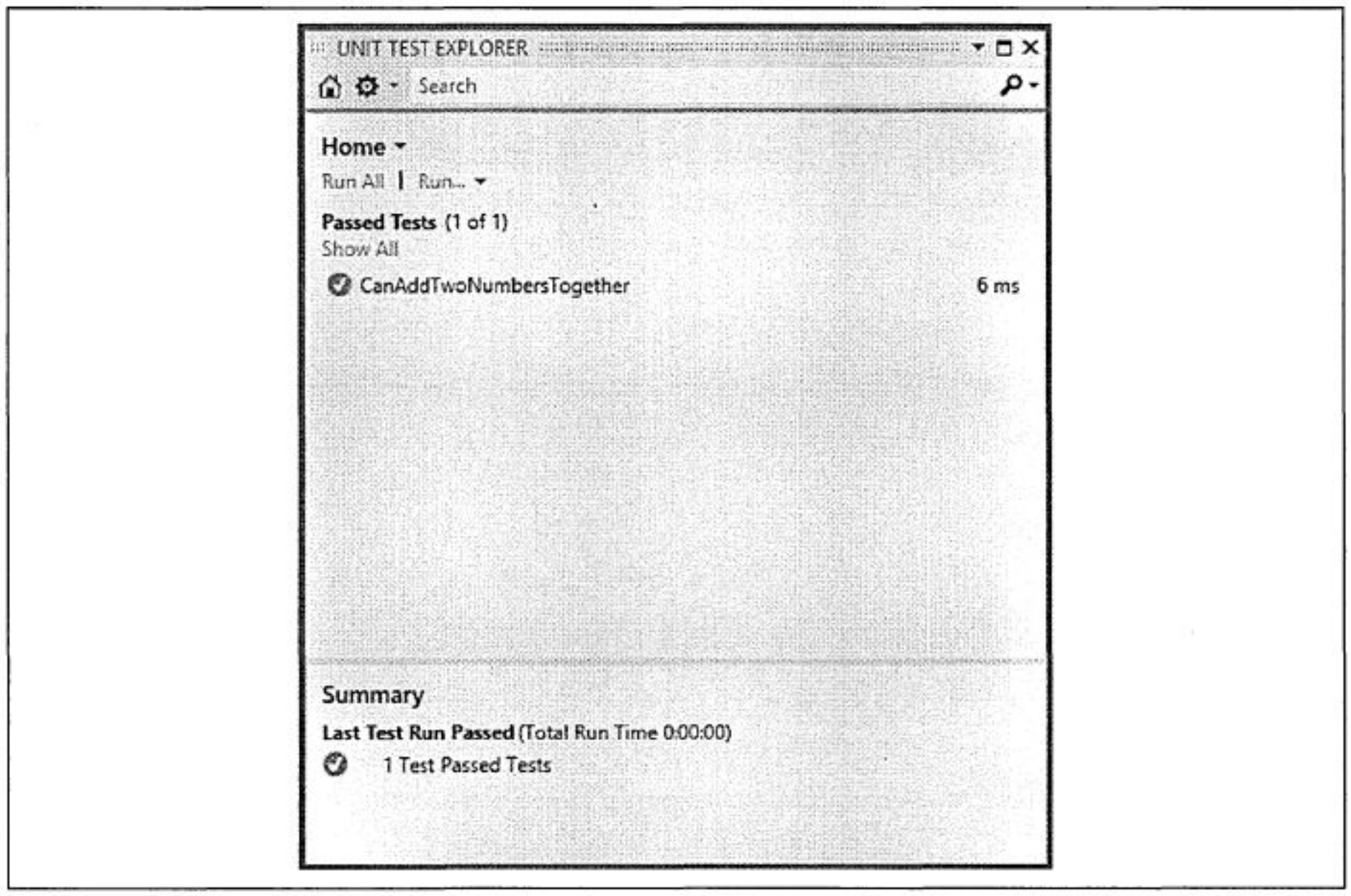


图17-3　检查运行的单元测试的状态

如果单元测试运行失败，或者测试逻辑验证失败（比如，修改代码`var sum = 1 + 2;`为`var sum = 1 + 3;`），则单元测试浏览器（Unit Test Explorer）就会显示失败，红叉（见图17-4）。我们可以通过单击相关内容来查看详细的失败信息。

注意图17-4所示的消息为断言失败，期望值为3，实际是4（`Message: Assert.AreEqual failed. Expected: <3>.Actual: <4>.`）。提示用户：`Assert.AreEqual()`帮助方法期望的值是3，但是返回的实际数值是4，和期望值不匹配。

354

既然已经学习完了Visual Studio单元测试项目，那么现在就用这款工具来测试ASP.NET MVC应用程序了。

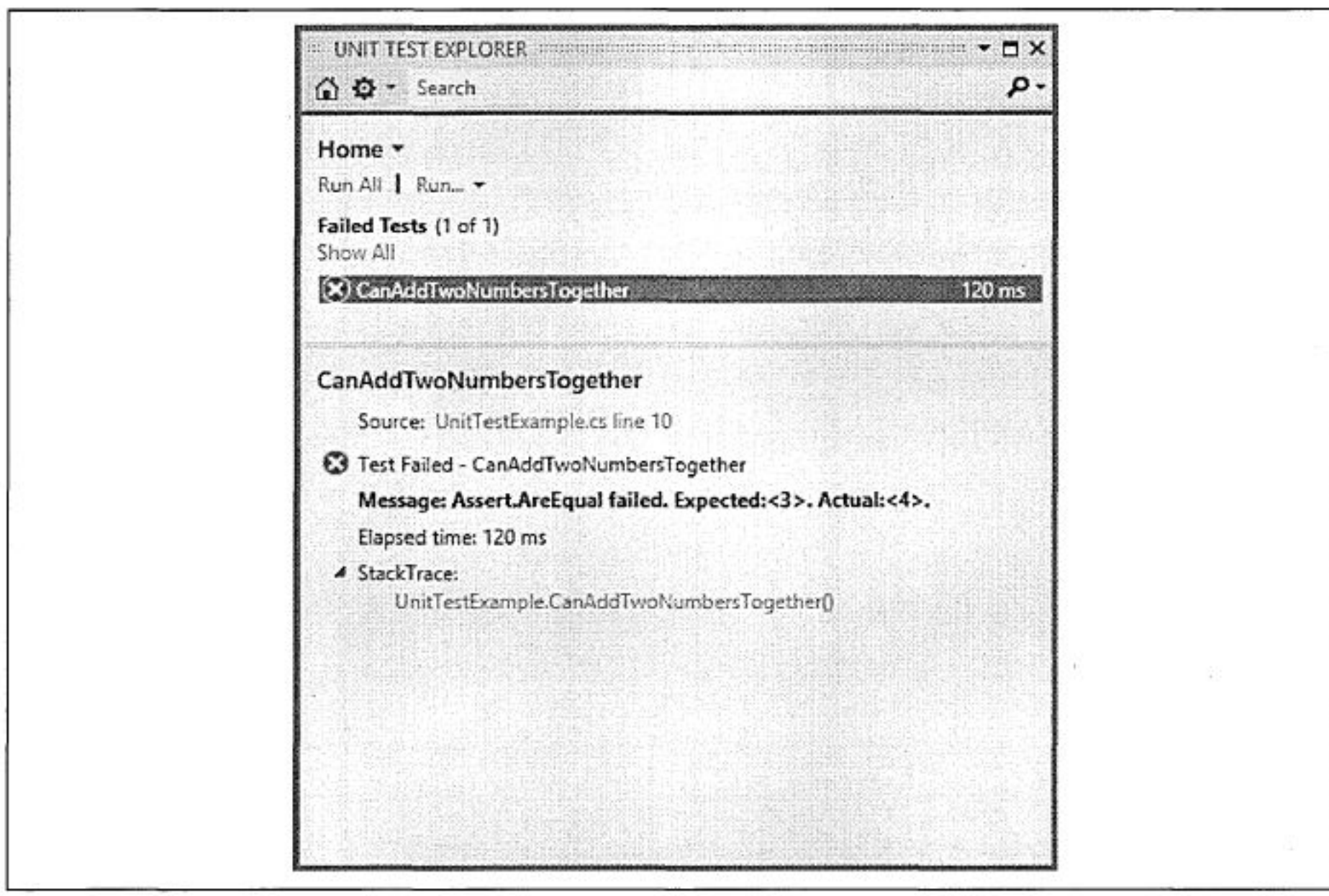


图17-4　单元测试浏览器显示的测试失败结果及原因

测试ASP.NET MVC应用程序

Testing an ASP.NET MVC Application

为了高效测试ASP.NET MVC应用程序，确保它在每个级别的测试中都可以正常工作，就需要一个强大的自动化测试套件，它最好集成支持单元测试、集成测试和验收测试。

虽然有很多种不同的方法可以达到这个目的，但是有一种很好的方法——利用ASP.NET MVC的分离关注点，独立测试每个架构层。正如下面几节所述，独立测试每个架构层确实是个好主意，因为每一层都有自己最适合的测试技术。而且，如果分开独立测试每一层，则可以进行更细致的测试，发现更多的Bug，修改以后就会更有信心，这样在集成测试时，就不会出现太多的问题。

355

测试模型

因为模型是ASP.NET MVC应用程序中最重要的部分，它包含绝大部分的逻辑代码，所以比较耗费测试精力。但是，当看到其他测试层以后，也许就会认为模型层是最简单的，因为它最直接，而且依赖最少。换句话说，模型通常由一些普通的、旧的.NET类（plain old .NET Class）组成，这些类易于实例化和进行独立测试。

为了说明问题，下面编写一段验证`Auction.PostBid()`方法的单元测试逻辑代码：

```
public class Auction
{
    public long Id { get; internal set; }
    public decimal CurrentPrice { get; private set; }
    public ICollection<Bid>Bids { get; private set; }

    // ...

    public Bid PostBid(User user, decimal bidAmount)
    {
        if(bidAmount <= CurrentPrice)
            throw new InvalidBidAmountException(bidAmount, CurrentPrice);

        var bid = new Bid(user, bidAmount);

        Bids.Add(bid);

        CurrentPrice = bidAmount;

        return bid;
    }
}
```

关注积极成果

在测试组件前，首先要搞清楚组件的定义，这样才能够编写单元测试代码并验证行为。在这个前提下，从最理想的输出结果开始，考虑测试行为所需要的东西。

在Auction.PostBid()的测试用例中，这个方法负责为交易的账单记录添加一个账单，然后更新交易响应的当前账单价格。换句话说，当赢得拍卖交易的金额超过当前的交易金额时，就会出现下面的情况。

- 向拍卖交易的账单记录里新增一条记录，包括竞拍人和竞拍价的信息。
- 拍卖交易的当前价格（CurrentPrice）被更新成当前最新的竞拍价(bidAmount)。

这些针对系统的逻辑期望翻译成单元测试代码是这样的：

356

```
using System.Linq;
using Microsoft.VisualStudio.TestTools.UnitTesting;

namespace Ebuy.Tests
{
    [TestClass]
    public class AuctionTests
    {
        [TestMethod]
        public void ShouldAddWinningBidToBidHistory()
        {
            var user = new User();
            var auction = new Auction { CurrentPrice = 1.00m };

            var bid = auction.PostBid(user, 2.00m);

            CollectionAssert.Contains(auction.Bids.ToArray(), bid);
        }
```

```
        [TestMethod]
        public void ShouldUpdateCurrentPriceWithWinningBidAmount()
        {
            var user = new User();
            var auction = new Auction { CurrentPrice = 1.00m };

            var bid = auction.PostBid(user, 2.00m);

            Assert.AreEqual(auction.CurrentPrice, 2.00m);
        }
    }
}
```

当使用Visual Studio单元测试运行器编译并运行这个例子时，应该可以看到结果是通过的，这表明`Auction.PostBid()`方法做了我们期望做的工作，逻辑功能符合要求（见图17-5）。

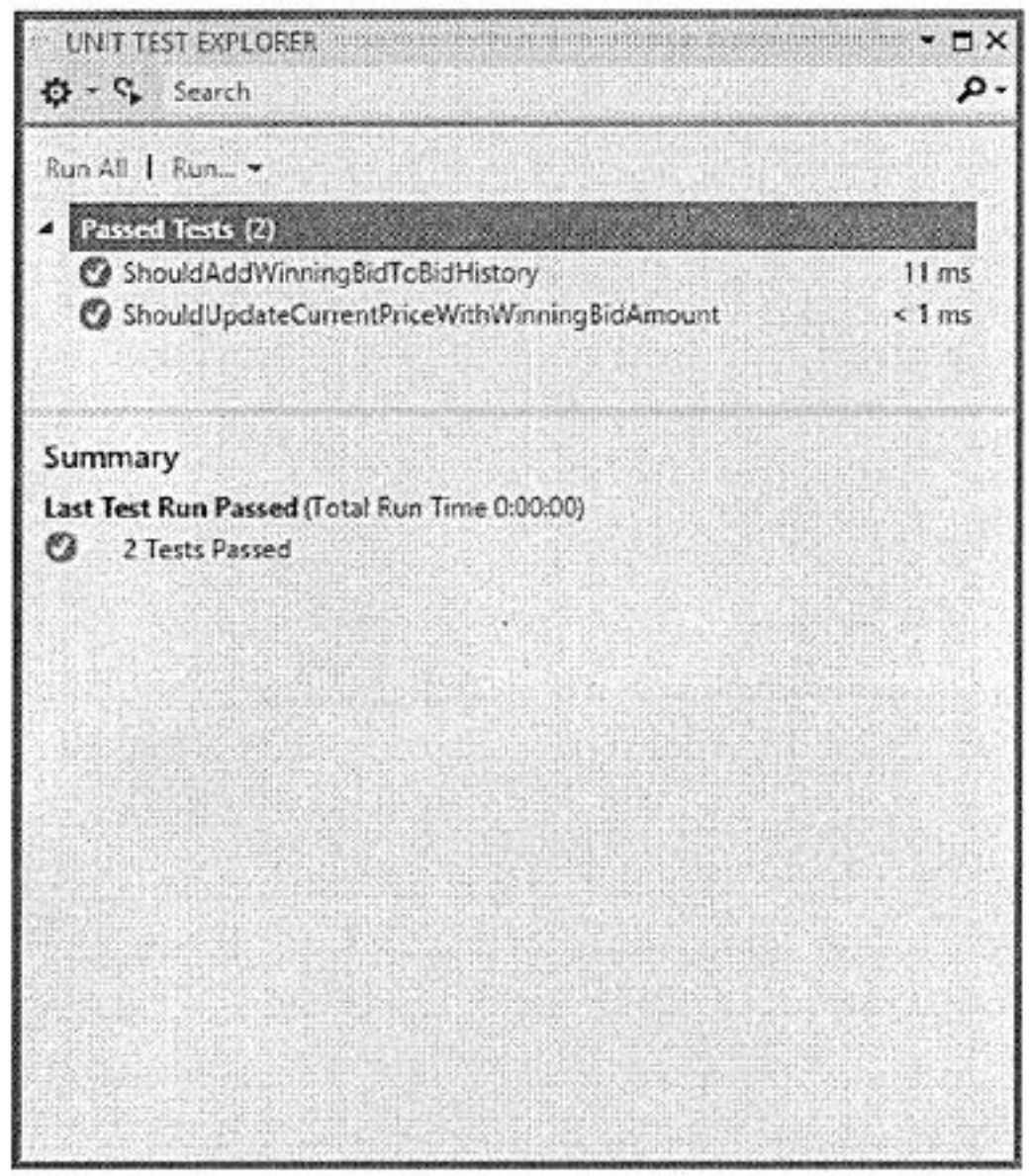


图17-5　检验Auction.PostBid()方法是否做了应该做的工作

远离错误

前面已经验证了`Auction.PostBid()`方法确实如我们所愿，正常工作。现在人为让它无法工作。换句话说，尝试去破坏它，以便看到测试失败的结果。

现在先删除之前的测试条件。前面的单元测试代码验证的是成功竞标交易价格超出原来价格时系统的行为是否正常，现在来验证当成功竞标交易价格低于原价格时系统会出现什么结果。

```
[TestMethod]
[ExpectedException(typeof(InvalidBidAmountException))]
```

```
public void ShouldThrowExceptionWhenBidAmountIsLessThanCurrentBidAmount()
{
    var user = new User();
    var auction = new Auction { CurrentPrice = 1.00m };

    auction.PostBid(user, 0.50m);

    //没有断言代码，因为前面一行代码会抛出异常
}
```

这个测试打算提交一个比当前最高交易竞标价格（1.00m，100W）低的价格（0.50m），这样系统应该会抛出一个InvalidBidAmountException异常。如果测试没有抛出异常而正常提交数据，那么说明系统逻辑功能存在Bug。

如果传递的User对象为null，则会发生什么呢？Auction.PostBid()方法目前的代码没有检查User对象是否为null，实际应该检查。这里通过测试会发现系统的Bug，但是照样应该在代码里给出测试，而不应该回避这些问题。

测试驱动开发

在修改Auction.PostBid()代码之前，先来看看另外一个非常有名的自动化测试方法：测试驱动开发（Test-Driven Development，简称TDD或测试优先的开发（test-first development））。

测试驱动开发遵循的原则是“红灯-绿灯-重构（Red-Green-Refactor）”，这条原则是指从失败情况（红灯）开始测试，然后编写尽可能少的代码（也可能是丑陋的代码）让测试通过（绿灯），最后，当测试代码通过时，意味着系统可以正常工作了，然后回头修改、重构代码，清除垃圾数据，让测试代码和标准项目代码一样整洁。

要在Auction.PostBid()方法上使用测试驱动开发，就必须事先定义好目标：使用null User值，从红灯状态开始测试，如下：

```
[TestMethod]
[ExpectedException(typeof(ArgumentNullException))]
public void ShouldThrowAnExceptionWhenUserIsNull()
{
    var auction = new Auction { CurrentPrice = 1.00m };

    auction.PostBid(null, auction.CurrentPrice + 1);

    //没有断言代码，因为前面一行代码会抛出异常
}
```

这个测试代码试图创建一个新的竞拍交易并传递一个null值作为User参数值给PostBid()方法。在测试代码中使用了ExpectedExceptionAttribute标记属性来断言抛出的异常是否符合我们的预期，如前一个例子所示。

前一个例子和现在这个例子的不同之处在于，当运行当前的Auction.PostBid()测试代码时，测试会失败，以为系统代码没有检查User的值是否为null，因而会抛出异常。

现在处于红灯状态，表明测试失败，现在应该想办法让测试通过。为此，需要给系统代码添加参数检查代码，看看User的值是否为null：

```
public Bid PostBid(User user, decimal bidAmount)
```

```
    {
        if(user == null)
            throw new ArgumentNullException("user");

        if(bidAmount <= CurrentPrice)
            throw new InvalidBidAmountException(bidAmount, CurrentPrice);

        var bid = new Bid(user, bidAmount);

        Bids.Add(bid);

        CurrentPrice = bidAmount;

        return bid;
    }
```

359

检查User参数值的代码编写完毕后，新的单元测试代码就可以正常通过了：处于绿灯的状态。通常，在测试代码达到绿灯状态后，也就是测试通过后，我们应该花点时间回头检查编写的代码，确保这些代码整洁、高效，或者说尽善尽美、符合项目开发的标准规范。

编写干净的自动化测试代码

虽然自动化测试与产品代码的目的完全不同，但是自动化测试代码是代码，所以绝大部分标准编码实践仍然适用。

重复代码

对于“菜鸟”来说，不鼓励编写重复代码这种做法，在自动化测试代码编写中和产品代码编写中都一样，禁止使用。

以刚才编写的代码为例，所有的代码都以这两行代码开始：

```
var user = new User();
var auction = new Auction { CurrentPrice = 1.00m };
```

虽然这两行代码创建的对象对执行测试来说至关重要，但是与要测试的逻辑没有特定关系。例如，编写的测试代码并不关心auction对象的CurrentPrice属性值是什么，只要有值就可以了。

幸运的是，所有的单元测试框架都提供了在测试执行之前安装代码的方法。Visual Studio test API支持在测试方法上标记TestInitializeAttribute属性来包含测试安装代码（译注4）。

例17-1展示了标记TestInitializeAttribute属性后的单元测试类的样子，这样可以删除很多重复代码。

例17-1　自动化测试代码文件AuctionTests.cs

```
using System;
using System.Linq;
using Microsoft.VisualStudio.TestTools.UnitTesting;
```

译注4：测试安装代码（test setup code），指执行的工作就是测试环境的初始化工作，比如准备测试参数等。

```
namespace Ebuy.Tests
{
    [TestClass]
    public class AuctionTests
    {
        private User _user;
        private Auction _auction;

        [TestInitialize]
        public void TestInitialize()
        {
            _user = new User();
            _auction = new Auction { CurrentPrice= 1.00m };
        }

        [TestMethod]
        public void ShouldAddWinningBidToBidHistory()
        {
            var bid = _auction.PostBid(_user, _auction.CurrentPrice + 1);

            CollectionAssert.Contains(_auction.Bids.ToArray(), bid);
        }

         [TestMethod]
        public void ShouldUpdateCurrentPriceWithWinningBidAmount()
        {
            var winningBidAmount = _auction.CurrentPrice + 1;

            _auction.PostBid(_user, winningBidAmount);

            Assert.AreEqual(_auction.CurrentPrice, winningBidAmount);
        }

        [TestMethod]
        [ExpectedException(typeof(InvalidBidAmountException))]
        public void ShouldThrowExceptionWhenBidAmountIsLessThanCurrentBidAmount()
        {
            _auction.PostBid(_user, 0.50m);
        }

         [TestMethod]
         [ExpectedException(typeof(ArgumentNullException))]
        public void ShouldThrowAnExceptionWhenUserIsNull()
        {
            _auction.PostBid(null, _auction.CurrentPrice + 1);
        }
    }
}
```

正如我们看到的，例17-1中把初始化逻辑代码放到`TestInitialize()`方法中了，这样可以让每个测试方法更加简洁，而且专注于自己的测试逻辑上。

命名

因为自动化测试代码永远也不会被应用程序调用（除了测试运行器），所以，有些注明的例

外编码规则。命名（naming）就是其中一个例外之处。类和方法名称在生产代码中非常重要，因为它们要告诉开发人员在应用程序中的类或方法是如何参与工作的——方法是做什么的，类是负责什么业务的。

在自动化测试代码中，命名同样非常重要，但是有一个基本的区别：测试类通常只有一个目的，就是作为测试方法的容器；而测试方法的目的就是测试特定的单元。

因此，测试类应该以被测试的单元命名，测试方法应该描述它们要验证的方法行为。

现在再看看本章已编写的测试代码的类名和方法名。到目前为止，只测试了一个目标类，所以只有一个测试类名字，`AuctionTests`。这个名字告诉我们，当前测试类的所有方法是为了测试模型中的交易类`Auction`。

测试方法的命名非常有意思。注意，测试方法的名字不仅仅是做个描述，而且应该读起来像实际的完整句子一样。例如，`ShouldAddWinningBidToBidHistory`测试验证的是`Auction`类，应该添加成功竞标的交易记录到交易历史里。测试方法名不限制长度，只要符合语言平台的最大限制要求即可（例如，可以使用`ShouldThrowExceptionWhenBidAmountIsLessThan-CurrentBidAmount`）。

可以根据测试类和方法的目的随便编写自己的命名规范：测试代码就是为了验证系统功能并找出潜在的问题。无论何时，只要测试失败，具备详细描述性的名称都可以给调试工作提供更大的帮助。这样，就可以很快找出系统中逻辑失败的部分，以方便调试中解决问题。

测试控制器

ASP.NET MVC框架的强大之处在于它是使用可测试的指导思想开发的。这意味着，可以非常方便地为ASP.NET MVC框架中的任意类创建实例并执行单元测试，正如对模型类进行的测试一样。

ASP.NET MVC控制器也一样，没有特别之处——事实上，控制器也是类，而且控制器操作也是类方法。

下面以`HomeController`控制器为例：

```
using System.Web.Mvc;

namespace Ebuy.Website.Controllers
{
    public class HomeController : Controller
    {
        public ActionResult Index()
        {
            ViewBag.Message = "Your app description page.";

            return View();
        }

        public ActionResult About()
        {
            ViewBag.Message = "Your quintessential app description page.";

        return View();
```

```
        }

        public ActionResult Contact()
        {
            ViewBag.Message = "Your quintessential contact page.";

            return View();
        }
    }
}
```

为了测试HomeController控制器的操作方法，我们要做的就是使用默认的构造函数实例化这个HomeController，然后调用操作方法。为了演示，现在来做一个简单的测试，以验证Index操作返回的视图：

```
[TestClass]
public class HomeControllerTests
{
    [TestMethod]
    public void ShouldReturnView()
    {

        var controller = new HomeController();

        var result = controller.Index();

        Assert.IsInstanceOfType(result, typeof(ViewResult));
    }
}
```

注意看看单元测试是如何实例化HomeController控制器类的，并且如何在ASP.NET MVC管道之外调用控制器操作方法。这就是ASP.NET MVC框架松耦合特性的强大之处，同样也是控制器操作直接返回ActionResult对象而不是继续执行余下的请求处理步骤的原因。

它不仅提供执行请求期间的强大功能和灵活性，而且允许单元测试来单独验证请求的各个部分。

测试数据访问逻辑代码

在学习了HomeController Index操作的单元测试代码之后，继续介绍包含数据访问的单元测试例子代码。现在看一下本书中最早实现的AuctionsController控制器的Auction操作方法的代码，在第8章重构之前，我们使用了更具测试性的存储库模式（repository pattern）：

```
public ActionResult Auction(long id)
{
    var db = new EbuyDataContext();

    var auction = db.Auctions.Find(id);

    return View(auction);
}
```

363

和前面编写的测试代码一样，现在通过从定义控制器期望的操作结果开始，使用自己熟悉的

语言。它通过传入的Id参数从数据库里查询Auction交易数据后显示到视图上。非常简单吧。

但是，当编写测试代码时，问题就变得复杂了，问题如下：

- 要查询的交易记录，数据库里存在吗？
- 如果存在，应该使用哪个Id来查询交易信息？
- 假设交易信息存在，而且知道这个Id，那么测试代码如何检测查询的数据是否正确？（例如，如果测试代码查询的交易是5，如何保证测试代码查询的结果不是8呢？）

这些问题的最简单的答案就是在数据库里创建一条新的交易记录，这样就可以保证测试代码知道每次请求的数据。

下面看看如何编写这种单元测试代码：

```
using Ebuy.Website.Controllers;
using Microsoft.VisualStudio.TestTools.UnitTesting;

namespace Ebuy.Tests.Website.Controllers
{
    [TestClass]
    public class AuctionsControllerTests
    {
        private Auction _auction;

        [TestInitialize]
        public void TestInitialize()
        {
            using(var db = new EbuyDataContext())
            {
                _auction = new Auction { Title= "Test Auction" };
                db.Auctions.Add(_auction);
                db.SaveChanges();
            }
        }

        [TestMethod]
        public void ShouldRetrieveAuctionById()
        {
            var controller = new AuctionsController();

            dynamic result = controller.Auction(_auction.Id);

            Assert.AreEqual(_auction.Id, result.Model.Id);
            Assert.AreEqual(_auction.Title, result.Model.Title);
        }
    }
}
```

364

注意测试类是如何使用TestInitialize()初始化方法在运行测试逻辑代码之前向数据库添加新的测试交易记录数据的。测试代码可以使用最初生成的交易信息Auction的Id从数据库查询数据。为了确定控制器操作方法查询的交易数据的准确性，需要比较Id和Title属性的值，以确保查询逻辑的正确性。

在成功运行单元测试代码之前，必须先在测试项目的配置文件里设置Entity Framework的连接字符串（connection string），如果还没在App.config文件里设置，那么必须设置，代码如下：

```
<connectionStrings>
<add name="DefaultConnection"
     connectionString="Data Source=(LocalDb)\v11.0;Initial Catalog=EBuy.Tests; ↵
     Integrated Security=true"
     providerName="System.Data.SqlClient" />
</connectionStrings>
```

当所有的配置工作完成之后，再检查单元测试是否工作，控制器逻辑工作是否正常。这样，就完成了使用自动化测试对控制器逻辑代码的测试工作。

重构单元测试

前面的单元测试例子验证了AuctionsController控制器的逻辑代码，而且这些代码无意中也测试了数据库访问逻辑代码，因为添加数据和查询数据都需要连接数据。虽然这个方法对模仿生产环境中系统集成情况下的模块测试非常有价值，但是它同样无法避免集成测试带来的缺点：让整个测试过程变慢，更容易出错，不可靠。同时使用两个组件进行单元测试的最大问题就是：当出错时，无法确定是哪个组件出了问题。

解决这种问题的最好办法就是，当问题出现时，让每个单元测试只关注验证单个的组件逻辑，然后把所有的组件放到一起进行集成测试。当测试应用程序中的组件时，要认真考虑每个组件的工作，而且确保单元测试只关注当前的测试组件行为，不要随意扩展测试范围。

当编写自动化测试代码，特别是单元测试代码时，如果假定所有被测试的单元交互的其他组件都各司其职，就可以很容易编写出单元测试代码。

事实上，当使用这种方法为所有的组件编写测试代码时，最后的结果就是一套高度集中的测试代码套件。当这种自动测试代码套件编写完成后，若测试中再出现错误，就可以很容易找出问题所在的组件，因为出问题的组件就是问题所在的组件，而无需查找一堆无关的组件代码，这样大大节约了查找问题的时间，提高了测试效率。

例如，当为从存储库查找数据的控制器操作编写单元测试代码时，真正想要测试的就是控制器操作方法中的代码，而不是数据访问层中查询数据的代码。问题就是，刚刚编写的测试代码要测试两层组件。下一节将会介绍解决这种问题的方法。

模拟依赖

当遇到前面所描述的问题时，第8章里介绍的存储库模式（repository pattern）这种技术开始体现出其价值了。这种模式引入了一个新的抽象层，允许使用假的组件来替换产品组件，这样就可以控制提供给被测试单元的数据了（这里是控制器操作方法）。

为了达到测试目的，使用假的组件代替真实产品组件的做法，称为模拟（mocking）。这种产生测试数据的组件也称为测试双打（test double）（译注5）。它们也有很多别名，比如假人（dummies）、假货（fakes）、存根（stub）、模拟（最常用），等等。

一旦可以确认依赖组件的行为不会变化，就可以通过删除外部的依赖组件和把关注点放在控

译注5：测试双打指的就是为了便于测试，模拟真实组件的替代品。

制器的逻辑代码上来把集成测试分解成单元测试。为了演示如何进行这种转换，下面看看如何在AuctionsController控制器上使用模拟技术，如第8章里的重构存储库模式一样：

```
using System.Web.Mvc;

namespace Ebuy.Website.Controllers
{
    public class AuctionsController : Controller
    {
        private readonly IRepository _repository;

        public AuctionsController()
            : this(new DataContextRepository(new EbuyDataContext()))
        {

        }

        public AuctionsController(IRepository repository)
        {
            _repository = repository;
        }

        public ActionResult Index()
        {
            var auctions = _repository.Query<Auction>();

            return View(auctions);
        }

        public ActionResult Auction(long id)
        {
            var auction = _repository.Single<Auction>(id);

            return View(auction);
        }
    }
}
```

注意，这里的代码中有两个构造函数：一个为接受IRepository类型的参数，控制器使用这个参数可进行数据库访问；另外一个为默认创建的参数，用于实现IRepository接口，会用在产品环境中。

这样，当控制器使用默认的构造函数创建（例如，ASP.NET MVC框架使用的构造函数）时，它将会使用IRepository接口的实现类(DataContextRepository)。然而，接受IRepository参数的构造函数为单元测试提供了注入模拟对象的机会，我们可以借此来控制提供给控制器的数据。

手动创建模拟对象

AuctionsController可以接受一个IRepository类型的模拟对象，这是一种很大的优势。在使用这种优势之前，首先必须创建这个模拟对象。或许创建IRepository模拟对象最简单、最直接的方法就是直接编写创建代码，即创建一个类，实现IRepository接口。其目的只是进行AuctionsController的自动化测试。

下面为IRepository接口的最基本的实现代码块，IRepository接口可以用于控制Single<TModel>()方法的返回结果：

```
public class MockAuctionRepository : IRepository
{
    private readonly Auction _auction;

    public MockAuctionRepository(Auction auction)
    {
        _auction = auction;
    }

    public TModel Single<TModel>(object id) where TModel : class
    {
        return_auction as TModel;
    }

    public IQueryable<TModel>Query<TModel>() where TModel : class
    {
        throw new System.NotImplementedException();
    }
}
```

MockAuctionRepository包含几个非常重要的特性。

第一点也是最重要的一点，这个类暴露的方法、每次调用的结果都是一样的，这可以保证每次与其他组件的交互行为都是可预测的。

第二点模拟类允许测试代码控制Single<TModel>()方法返回的结果数据。每次构造函数接受Auction对象，然后通过调用Single<TModel>()方法返回模拟对象。

第三点MockAuctionRepository实现了自动化测试所需的IRepository接口最低限度的代码。换句话说，MockAuctionRepository模板就是一个测试场景（test scenario），并不是为了达到通用测试目的。还有一个事实，Query<TModel>()类抛出了一个NotImplementedException异常，而且是故意这么做的——因为我们知道，被测试单元(AuctionsController.Auction()方法)就是想要调用IRepository.Single<TModel>()方法。如果在测试环境中执行了Query<TModel>()方法，就会抛出这个异常，而且测试将会失败（当然这里我们期望失败，实际是正确的）。

现在，通过重写早期的集成测试代码来使用这个新的模拟存储类：

```
[TestClass]
public class AuctionsControllerTests
{
    [TestMethod]
    public void ShouldRetrieveAuctionById()
    {
        var auction = new Auction { Id = 123 };
        var mockRepository = new MockAuctionRepository(expectedAuction);

        var controller = new AuctionsController(mockRepository);
        dynamicresult = controller.Auction(expectedAuction.Id);

        Assert.AreSame(expectedAuction, result.Model);
    }
}
```

可以看到，测试代码是如何使用基于存储库模式的AuctionsController控制器，通过创建MockAuctionRepository的实例并传递给控制器来代替基于数据库的IRepository实现代码的。测试代码会先调用controller.Auction()方法，再调用MockAuctionRepository.Single<Auction>()之后就获取模拟对象，并模拟数据库操作。测试代码随后根据result.Model模型对象做一些断言来验证查询到的模拟交易Auction信息与Auction交易对象（期望的Auction）是否一致。

注意测试代码是如何设置Auction.Id属性的值的。尽管这个属性设置器已经设置了internal访问修改属性，但测试类在另外一个与Auction类不同的程序集中（不在同一个程序集内部域内）。设置应用程序集中的标记属性InternalsVisibleTo到Ebuy.Core项目（Auction类所在的项目）上后，InternalsVisibleTo标记属性允许把Ebuy.Core程序集暴露给Ebuy.Tests测试项目，代码如下：

```
[assembly: InternalsVisibleTo("Ebuy.Tests")]
```

这个方法有另外一个好处：它不需要确保数据库中必须存在特定Id的Auction交易对象数据了（测试代码自己创建了Auction交易对象），因为测试代码使用模拟存储对象代替基于数据库的IRepository的实现代码。

与之类似，也可以使用模拟对象取代任何外部的依赖，并且可以把集成测试转化成单元测试。

使用模拟框架

通过手工创建模拟类，如MockAuctionRepository，在测试过程中使用模拟组件代替产品组件，这是一种非常好的方法，但是也有一些问题。

虽然模拟类只在测试过程中实现，但是它们也是真正的接口实现类，这意味着测试人员为了单元测试不得不编写更多的代码，而且随着代码库的增长，他们还必须维护模拟代码。更糟糕的是，有时候可能为了测试多个场景，开发人员不得不编写多个接口的不同实现版本。正如MockAuctionRepository的例子一样，目前它只是实现了Single<Auction>()方法。若要修改底层接口带来的影响，则可能要影响几个甚至几十个模拟接口实现代码。

幸运的是，有可以代替手工编写模拟类的方法，即使用模拟框架。

模拟框架是指可以提供给开发人员使用的、可以根据实际需求动态创建模拟类的框架。简单来说，无论是在开发测试阶段，还是在后期维护单元测试套件阶段，模拟框架可以给我们带来手工创建模拟类的所有好处，且只需编写部分代码。

虽然.NET平台上有很多模拟框架，但是它们本质上都执行同样的任务：动态创建模拟对象。因为这些模拟框架主要是在各自创建模拟对象的语法上有差别，所以，选择哪个模拟框架就看大家的喜好了。

本书的例子代码中使用的是著名的开源模拟框架Moq framework（译注6），但是可以随意更换成任何自己喜欢的模拟框架。例如，也许想试试别的开源模拟框架，像Rhino Mocks、Easy-Mock.NET、NMock或FakeItEasy，甚至也可以选择某些收费的商业框架。

译注6：Moq framework 由 Daniel Cazzulino、Kaleb Pederson、Tim Kellogg 创立，可以在这里下载源代码：*https://github.com/Moq*。

例如，下面的代码块使用一个IRepository对象生成的模拟类替换手工编写的MockAuction-Repository类：

```
[TestClass]
public class AuctionsControllerTests
{
    [TestMethod]
    public void ShouldRetrieveAuctionById()
    {
        var expectedAuction = new Auction { Id = 123 };

        var mockRepository = new Moq.Mock<IRepository>();
        mockRepository
            .Setup(repo =>repo.Single<Auction>(expectedAuction.Id))
            .Returns(expectedAuction);

        var controller = new AuctionsController(mockRepository.Object);
        dynamic result = controller.Auction(expectedAuction.Id);

        Assert.AreSame(expectedAuction, result.Model);
    }
}
```

由上可以看到，模拟框架只需要几行代码就可以支持IRepository接口。首先，测试代码通过Moq.Mock<T>对象动态创建一个IRepository类型模拟实例：new Moq.Mock<IRepository>()。然后告诉模拟对象使用Moq framework的Setup()来指定调用期望的方法(repo.Single<Auction>(expectedAuction.Id))，并且指明在应答消息中应该返回的应答值。

这个例子的结果证明可以使用动态模拟IRepository对象来替代手工编写的MockAuction-Repository类，并且模拟类具备手工编写模拟类的好处，且只需要几行代码。对维护来说，非常简单。因为模拟框架可以大大减少生成模拟依赖的工作量，与集成测试相比，这个方法可以帮助开发人员更加高效地编写更多的测试代码。

370

测试视图

在模型和控制器成功测试以后，现在来测试视图。

在开始测试之前，问问自己到底要测试什么？考虑用户和ASP.NET MVC应用的交互过程，ASP.NET MVC框架最担心的还是显示到用户浏览器上的HTML页面。

如第4章所介绍的，现在的Web页面工作已远远超过在浏览器里渲染的HTML和CSS。高级JavaScript和AJAX技术为简单的内容页面提供了绝佳的用户体验，并且让浏览器变成了强大的开发平台，这个平台和ASP.NET MVC框架联系不大，没有深度耦合。

现在所有问题都可以归结为一点，就是测试应用程序视图层远远不像实例化对象和编写几行代码测试行为那么简单。实际上，视图层也可以分解成多个不同的层，即包含内容和页面结构的HTML视图文件、包含应用逻辑的JavaScript文件，以及包含CSS样式代码的文件。与页面相关的这三个方面需要顺序组合才能达到完美的用户体验，而且它们需要完全不同的测试方法和技术。

在浏览器中测试应用逻辑

虽然上面的例子对网站使用基于文本的自动化测试确实可行，但是基于文本的测试方法的好处非常有限，因为浏览器接受的内容和它选择渲染的内容是两码事。如果考虑不同的浏览器

渲染不同的HTML代码，那就更复杂。为了真实地测试实际用户和网站的交互情况，只有把网站加载到浏览器里，并且像普通用户一样查看。

这个方法也称为浏览器测试（browser testing），因为这个测试过程需要真正打开浏览器，而且要通过使用虚拟鼠标点击、敲击键盘来模拟实际的用户行为。显然，最简单的执行浏览器测试的方式就是通过手工打开浏览器，然后像正常网站用户一样操作网站。然而，本章是关于自动化测试的，所以将关注在计算机可帮助执行的自动化测试任务上；换句话说，就是自动化浏览器测试。

有很多工具可以帮助我们自动执行浏览器测试，只需要使用一个自动化测试套件即可。本章中使用的工具叫WatiN。

为了在网站中使用WatiN工具，需要在NuGet包管理器添加WatiN包到项目的引用列表里。添加引用完毕，就可以创建正常的测试类和测试方法——剩下的就和之前编写测试代码一样了——要引用`WatiN.Core`命名空间。

例如，下面的单元测试使用了IE（Internet Explorer）来浏览在线EBuy电子交易网站：

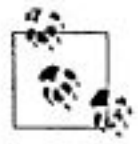
```
using Microsoft.VisualStudio.TestTools.UnitTesting;
using WatiN.Core;

namespace Ebuy.Tests.Website.Browser
{
    [TestClass]
    public class AuctionTests
    {
        [TestMethod]
        public void ShouldNavigateToAnAuctionListingFromTheAuctionsList()
        {
            const string baseUrl = "http://localhost:65193";
            using(var browser = new IE(baseUrl + "/auctions", true))
            {
                var auctionDiv = browser.Div(Find.ByClass("auction"));
                var auctionTitle = auctionDiv.Element(Find.ByClass("title")).Text;

                auctionDiv.Links.First().Click();

                Assert.IsFalse(string.IsNullOrWhiteSpace(auctionTitle));
                Assert.AreEqual(
                    auctionTitle,
                    browser.Element(Find.BySelector("h2.title")).Text);
            }
        }
    }
}
```

这个测试代码通知IE浏览器打开*http://localhost:65193/auctions*地址，这个URL显示的是运行在本机的65193端口的网站，显示所有拍卖交易的信息列表。一旦页面加载完毕，这个测试代码就会获取第一个`Auction`交易信息元素(`<divclass="auction">`)，然后查找`Auction`交易信息元素的title(`<span class="title">`)，以确保当前测试加载的页面是正确的页面。

编写完代码之后，就要执行测试了。测试代码会先找到列表中的第一个`Auction`交易信息元素的连接，然后触发link元素的`.Click()`方法，这样就可以跳转到选定的交易信息的详细页面。

最后，测试代码通过验证交易信息title标题属性来(从交易详细页面列表选取的)和详细页面的`<h2class="title">`进行比较，以断言网站是否运行正常。如果运行正常，页面的`<h2>`元素

就没有任何问题，而且应该包含正确的交易标题（auction title）。



如果想要执行单元测试，并且接收到测试运行器无法加载Interop.SHDocVw程序集的消息，就意味着COM互操作对象设置得不正确。

为了解决这个问题，需要在项目的程序集引用列表里找到Interop. SHDocVw程序集，然后修改Embed Interop Types属性的值为false，并且把Copy Local的值设置为true。

这样就可以让测试运行器与IE通信，并且可以成功执行测试代码了。

这个例子虽然简单，但是很好地演示了浏览器自动化测试的功能。注意通过WatiN扩展API与浏览器交互的不同方式。例如，IE类提供了测试浏览器的模拟对象，而且Find类提供了许多不同快速高效的通过DOM查找元素的方法。不仅可以定位和分析浏览器元素，而且可以访问不同的元素方法，如.Click()方法，可以让测试代码模拟用户使用浏览器和网站的交互行为。

代码覆盖率
Code Coverage

代码覆盖率（code coverage）是一种根据测试应用程序过程中执行代码行数来判断自动化测试套件测试质量的代码分析技术。代码覆盖率通常用百分比衡量，例如，如果一个组件包含100行代码，而测试套件执行了其中的75行，则可以说这个测试代码具备75%的代码覆盖率。

为了评估项目的测试代码覆盖率，必须采用代码覆盖率分析工具。这个工具可以根据配置过程执行自动测试套件，并且跟踪每行代码的实际执行情况。当执行完测试代码后，代码覆盖率分析工具就会生成被测试程序各个部分的代码覆盖率统计报告。

如果使用的是非Express版本的Visual Studio，那么很幸运，因为这个版本的开发工具已经内置了代码覆盖率分析工具。要使用Visual Studio代码覆盖率分析工具，通过单元测试（Unit Test）→分析代码覆盖率（Analyze Code Coverage）菜单，选择“选择测试（Selected Tests）”或“所有测试（All Tests）”中的一个选项即可。

这样，Visual Studio代码覆盖率分析工具会执行单元测试代码，并且会自动分析代码覆盖率，在Visual Studio代码覆盖率结果窗口里显示出来，如图17-6所示。

Hierarchy	Not Covered (Blocks)	Not Covered (% Blocks)	Covered (Blocks)	Covered (% Blocks)
ebuy.core.dll	26	36.62 %	45	63.38 %
Ebuy	26	36.62 %	45	63.38 %
Auction	0	0.00 %	24	100.00 %
Auction()	0	0.00 %	9	100.00 %
PostBid(Ebuy.User, decimal)	0	0.00 %	15	100.00 %
Bid	0	0.00 %	6	100.00 %
Bid(Ebuy.User, decimal)	0	0.00 %	6	100.00 %
DataContextRepository	3	33.33 %	6	66.67 %
DataContextRepository(System.Data.Entity.DbContext)	0	0.00 %	2	100.00 %
Query<T>()	3	100.00 %	0	0.00 %
Single<T>(object)	0	0.00 %	4	100.00 %
EbuyDataContext	0	0.00 %	4	100.00 %
EbuyDataContext()	0	0.00 %	4	100.00 %
EbuyDataContext.Initializer	23	100.00 %	0	0.00 %
Seed(Ebuy.EbuyDataContext)	23	100.00 %	0	0.00 %
InvalidBidAmountException	0	0.00 %	5	100.00 %
InvalidBidAmountException(decimal, decimal)	0	0.00 %	5	100.00 %

图17-6　Visual Studio代码覆盖率结果

图17-6中，Visual Studio可以显示方法级别的自动化测试中所有代码的覆盖率和非覆盖率。

这个例子中，ebuy.core.dll程序集只有63.38%的代码覆盖率。假设这个项目中使用了扩展自动化测试套件，那么这个数字将会迅速提升，而我们希望看到的结果最好接近100%。

为了了解100%代码覆盖率的秘密，首先来看下平均代码覆盖率中最低的部分。快速浏览列表中的结果就会发现有两个方法没有覆盖到，即`DataContextRepository.Query<T>()`和`EbuyDataContext.Initializer.Seed()`。

这两个方法有一个共同之处：它们都和数据库交互。而且`EbuyDataContext.Initializer.Seed()`方法只在数据库创建时执行一次，后续就不会再执行。

这个信息包含几个隐含含义，其中最重要的一个表明，测试代码是根据已有数据库执行的，不会创建新的数据库了。这也许是设计的原因，这个方法缺少代码覆盖率情有可原。否则，就证明一定出现了重大错误，应该为每次执行创建一个新的数据库。

事实上，`DataContextRepository.Query<T>()`方法从来都不会执行。这个结论表明，也许为这个方法编写更多的测试代码，或者这个方法根本不需要存在，应该删除它。

这些都是我们必须要做的分析工作，可以了解问题的根源。根据实际的情况，有时候也许我们要为自动化测试覆盖率没有达到100%找个理由——这就是真正原因。

100%代码覆盖率的秘密

评估测试套件的代码覆盖率是一种可以确保我们竭尽所能地测试系统代码的绝佳方式。

虽然完美的情况就是测试应用程序的每一行代码，但是要达到100%的代码覆盖率还存在两个问题。

1. 100%的代码覆盖率实际上是不可能达到的。尽管有很多种易于测试代码的方法（下一节会介绍），但是，仍然有很多种情况使用自动化测试方法难以覆盖到。例如，有些组件直接与`HttpRequest`对象交互，而HttpRequest对象需要做很多工作来进行对象实例化。复杂的逻辑导致自动化测试很难覆盖所有的情况。此外最简单、最有效的测试方法就是在ASP.NET管道里测试这个对象，但是这个测试方法违背了本章前面提到的测试指导原则。
2. 虚假的安全感（sense of security）。可回顾一下早期创建的计算器单元测试示例代码。即使测试代码覆盖了所有的代码，也仍然无法保证输入参数的异常。

即使这个测试代码实现了对`Calculate.Add()`方法100%的代码覆盖率，也就是说，测试代码成功执行了该方法中的每一行代码，难道100%的代码覆盖率就意味着`Calculate.Add()`不会出错吗？这里只使用了数字1和2进行了测试，表明代码可以良好地支持其他任意值而已！如果尝试一些极端的情况呢？

这并不是说较高的代码覆盖率有什么问题。只是想让大家知道，只要想做，就可以让代码覆盖率达到100%，如果没有达到，也不能说明测试代码有问题，或者系统不安全。

开发可测试的代码

Developing Testable Code

如果高效的软件测试方法关注检验组件或应用程序是否可以在生产环境中按照预期的行为运行，那么任何在模拟生成环境进行的测试工作，测试环境越接近生产环境，测试工作就会越复杂，越难检验测试代码。

例如，回忆一下最近测试的组件，然后问自己下面的问题：

- 安装和配置等组件测试准备工作花了多少成本？
- 这个组件需要与多少其他组件交互（要求额外的配置）？
- 这个组件是否依赖于其他类库、数据库、Web服务或者本地文件系统？
- 依赖项的可靠性和一致性如何？例如，Web服务是否返回同样的结果？

所有这些都会影响软件或组件的可测试性。

以下面这两个方法为例。虽然各自只包含一行代码，但第一个方法比第二个方法更易于测试，第二个方法基本代表了最复杂的测试场景。第一个方法是GetVersionNumber()，代码如下（非常简单）：

```
public static int GetVersionNumber()
{
    return1;
}
```

是什么让GetVersionNumber()方法这么容易测试？是因为这个方法每次都返回相同的结果（int值1），无论我们调用多少次，结果都一样。更重要的是方法的可访问性。因为它使用了public修饰符声明，所以任何组件都可以调用这个方法并获得结果。Static修饰符表示这是个静态方法，因此不需要实例化对象就可以直接访问——这只是个简单的方法调用而已。

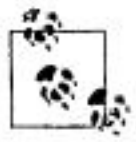
换句话说，CallRemoteWebService()方法完全不同，它代表了另外一种测试场景，代码如下：

```
private ServiceResult CallRemoteWebService()
{
    return new WebService("http://thirdparty.com/service")↵
    .GetServiceResult("some special value");
}
```

这个方法使用private修饰符定义了私有访问级别，所以此方法只能在类内部调研，也就是说，这个方法已经与外界绝缘了。这意味着，要想执行这个方法，测试代码必须先调用同一个类中的另外一个方法，这就必须引入一些无关的逻辑代码，这正是我们应尽力避免的。而且还要在调用这个方法之前创建类的实例，这就要使用到类的依赖，使问题更复杂了。

另外，CallRemoteWebService()方法要调用一个依赖项（远程的Web服务），这样在执行这个方法时可能会带来一些潜在的问题。这个Web服务可用吗？在调用这个方法时会不会出现网络问题？Web服务接口会不会返回正确的数据？如果是第三方服务，收费吗？如果收费，而我们又不知道这种情况，在测试之后，真要是收到一大堆收费单就痛苦了。

最后，最重要的问题是如何调用外部Web服务，这个问题让`CallRemoteWebService()`方法变得难以测试。注意，测试方法创建了一个`WebService`类的实例，然后使用硬编码URL来调用外部服务。因此，不调用产品环境下的外部服务就不可能测试这个方法，也就意味着我们没有办法保证外部Web服务行为的正确性。



注意new关键字——这也意味着，我们有机会使用依赖注入（dependency injection）来实现组件之间的松耦合关系。

当一个组件收到一个通过依赖注入传入的实例时，它就可以在测试时轻易替换掉测试的对象。但是，当组件对象是自己创建的时候，替换工作几乎不可能完成。

因此，任何使用了外部Web服务执行的针对这个方法的集成测试都是比较高效的。虽然这听起来不错，但是它意味着这个方法不可能进行单元测试，因为它几乎违反了上面所述的真正单元测试的所有指导原则。

当考虑这种情况的缺点时，就会发现问题严重得多。这个方法不仅无法进行单元测试，而且没有任何单元测试方法能够调用这个方法。如果应用程序中包含这种方法太多，那么很快，整个应用程序就会变得无法进行单元测试。

总结
Summary

自动化测试是确保整个应用程序质量和功能的最佳方式之一。传统上，对ASP.NET Web应用程序很难进行自动化测试，但是，ASP.NET MVC框架带来的松耦合架构让ASP.NET Web网站的自动化测试变得易如反掌。

把ASP.NET MVC框架与本书介绍的优秀模式、最佳实践及工具结合起来，很快就可以体会到自动化测试带来的快感和好处。

第18章

自动化生成

Build Automation

从原始代码到用户可以使用的功能软件（或者叫应用程序），中间还有很多工作要做。例如，如果应用程序是使用C#这种静态语言编写的，那么首先要编译代码，然后把编译后的程序集拷贝到特定的文件夹（ASP.NET Web网站里是Bin文件夹）中。另外，应用程序可能还需要其他的条件才能正常运行，比如图片、脚本文件，甚至整个数据库。这个准备过程——包括为了应用程序正常工作所做的所有工作，通常称为“生成（Build）”。

第17章详细介绍了这个问题，人的最大缺点就是无法保证重复执行任务的准确性。但是，计算机可以轻而易举地、高精确度地重复相同的工作。第17章演示了如何使用计算机来帮助我们执行自动化测试工作。

本章会继续拓展软件开发过程中与自动化技术相关的内容，把自动化技术应用到生成和部署应用程序的工作中。下面会介绍如何使用自动化方法来提升团队开发人员和其他项目组成员在创建、验证以及交付软件工作时的效率。

这一章主要是使用Microsoft MSBuild和TeamFoundation Server工具来演示例子。其实还有很多可以替代MSBuild和TeamFoundation Server的开源工具，每款工具都有自己创建和执行脚本的方式。

抛开这些不同之处，几乎所有的生成和部署框架都是由相同的基本概念和技巧驱动的。所以，只要学会了本章的例子，记住了自动化生成和部署（automated build and deployment）的概念，使用任何工具就都不成问题，正所谓万变不离其宗。

创建生成脚本

Creating Build Scripts

在创建自动化生成脚本之前，必须先确定希望计算机为我们做什么工作。可以使用包含一系列任务的生成脚本文件（build scripts—files）来定义计算机执行的任务。

生成脚本（build scripts）的实际格式和语法因使用的工具而千差万别，但是，无论使用什么语言编写，生成脚本都必须包含诸如这样的任务：执行应用程序单元测试套件、评估应用程序代码质量、生成并执行数据库脚本，这只是可以想象的和项目生成、部署相关的任务。

像Visual Studio这样的集成开发环境，可以通过“解决方案”和“项目”这样的概念为自动化生成提供完善的支持，而且可以轻易定义应用程序需要的东西，以及如何使用这些东西来安

装可执行的应用程序。例如，Visual Studio C# Project(.csproj)文件也许会包含C#代码文件名称集合，还有调用C#编译器(csc.exe)把这些源代码文件编译成.NET程序集或者可执行文件的逻辑步骤。无论Visual Studio集成开发环境多么强大，项目和解决方案只是自动化生成工作中“万里长征的第一步”。

Visual Studio项目就是生成脚本

虽然有很多其他的自动化脚本工具可以帮助我们创建自动化任务，但是，也许会惊喜地发现Visual Studio里已经集成了一款强大的脚本工具，即微软生成引擎（Microsoft Build Engine，也称为MSBuild)。MSBuild使用XML schema文件定义不同的任务脚本。实际上，所有的Visual Studio项目和解决方案文件都是MSBuild文件，只是添加了一些特殊的文件扩展，而且——每次点击F5，调试程序——Visual Studio都会把这些文件交给MSBuild，以便编译应用程序代码。

添加简单的生成任务

为了添加生成任务，首先要打开资源浏览器（Windows Explorer），进入EBuy网站解决方案下的EBuy.Website文件夹，再使用文本编辑器打开EBuy.Website.csproj项目文件。在项目文件里，可以发现`<Project>`节点下包含很多子节点，如`<Import>`、`<PropertyGroup>`和`<ItemGroup>`——这些元素就是要告诉MSBuild引擎如何生成项目。

在项目文件的底部，可看到下面的注释文字：

```
<!-- To modify your build process, add your task inside one of the targets below and
     uncomment it. Other similar extension points exist, see Microsoft.Common.targets.
<Target Name="BeforeBuild">
</Target>
<Target Name="AfterBuild">
</Target> -->
```

正如其名字的含义一样，这两个目标——`BeforeBuild`和`AfterBuild`——允许在生成任务执行之前或执行之后执行特定的代码。

为了了解修改生成文件有多简单，先来修改文件，让它在生成任务结束后显示信息。为此，先删除`AfterBuild`的注释，然后在其中添加调用MSBuild `<Message>`的代码，如下：

```
<!-- To modify your build process, add your task inside one of the targets below and
     uncomment it. Other similar extension points exist, see Microsoft.Common.targets.
<Target Name="BeforeBuild">
</Target>
  -->
<Target Name="AfterBuild">
  <Message Importance="High" Text="**** The build has completed! ****" />
</Target>
```

执行生成

修改完成后，就来执行生成并查看结果。这里有两个选项，即使用Visual Studio生成，或者直接在命令行使用MSBuild命令生成。

使用Visual Studio生成

在Visual Studio里执行MSBuild比较麻烦，但这些工作是使用Visual Studio开发和部署.NET应用程序时必须做的工作。现在在Visual Studio里打开包含项目的解决方案，然后点击其中一个Visual Studio快捷方式（使用快捷键Ctrl-B），启动生成。

一旦生成任务执行完毕，就会看到在Visual Studio的输出窗口中显示以下信息：

```
3>------ Rebuild All started: Project: Ebuy.Website, Configuration: Debug Any CPU ------
3> Ebuy.Website -> C:\Code\EBuy\trunk\Website\bin\Ebuy.Website.dll
3> **** The build has completed! ****
========== Rebuild All: 3 succeeded, 0 failed, 0 skipped ==========
```

使用命令行工具生成

Visual Studio项目和解决方案的优势之一就是我们不需要通过Visual Studio就能打开项目。相反，可以通过MSBuild命令直接执行项目的文件生成脚本。

首先，在计算机的开始菜单里找到并打开Visual Studio命令提示符（Command Prompt），这样会自动配置环境到.NET工具上——包括MSBuild——可以直接使用。

然后，输入并执行MSBuild命令，但要传递解决方案或项目文件的名称给`msbuild`命令：

```
msbuild Ebuy.sln
```

这个命令会根据解决方案执行MSBuild工具链，而且会在Visual Studio输出窗口中产生不同的文本消息。在这些输出信息中，应该可以找到自定义的`AfterBuild`消息，代码如下：

```
[...]
AfterBuild:
  The build has completed!
Done Building Project "C:\Code\EBuy\Website\Ebuy.Website.csproj" (default targets).
[...]
```

一切皆有可能

为Visual Studio的生成输出结果添加自定义消息看上去没有什么意义，其实是因为这个例子太简单。这个例子真正要演示的是可以在生成过程中插入自定义的逻辑代码，并且允许在命令行里执行任意的.NET代码或脚本。

自动化生成概述

Automating the Build

如果只需要在Visual Studio中点击一个按键或者使用命令行工具就可以自动执行生成脚本，那确实不错，但这些方法还不够完美，因为还需要手工操作、人员干预。自动化生成是指计算机可以执行生成脚本，而不需要人工干预，这才是自动化生成脚本的真正价值。

一旦定义完自动化生成任务的脚本，接下来只需直接把任务脚本交给自动化生成服务即可，这项服务通常称为**生成服务器**（build server）。生成服务器会一直运行，并等待接收外部传入的生成脚本，然后执行任务。

生成服务器产品被.NET开发团队集成到微软的Team Foundation Server (TFS)里。Team Foundation Server产品非常流行，因为它集成了支持软件开发生命周期中的许多重要概念，比如源代码控制（source control）、工作项目跟踪（work item tracking）、报表以及自动化生成与测

因为本章主要关注自动化生成的概念，所以本书假设读者已经安装并配置了Team Foundation Server，且使用Team Foundation Server作为源代码控制工具。如果没有安装，那么可以直接访问免费的在线微软Team Foundation Server服务器，这个服务器提供了本章中所需要的所有功能。如果选择了其他的自动化生成服务器，那么可以把本书中的很多概念映射到自己选定的具中（译注1）。

自动化生成的类型

因为生成脚本可以执行任何我们希望的逻辑，所以，对每个生成脚本，最好准确定义每个脚本要实现的职责范围。

例如，下面的列表包含了几种常见的自动化生成类型。

持续生成

持续生成(continuous builds)是指任何团队开发人员提交修改代码到代码库后，都会触发立即执行生成工作。这个方法的主要目的就是对立即提交的修改代码做出即时反馈。为了检验代码质量，持续生成的任务就是执行编译和单元测试套件，检测代码工作是否正常。这些单元测试代码可以很快执行完毕，因为持续生成触发的频率很高，所以在每次提交修改代码之后可以很快生成，而不会导致多个版本互相重叠。

滚动生成

滚动生成(rolling builds)和持续生成类似，区别在于它会限制特定时间内的生成次数。例如，可以配置每隔5分钟执行一次滚动生成，而不需要像持续生成那样在每次修改之后都要生成。在5分钟时间内，开发人员可以随意提交修改，直到5分钟时间间隔结束才会执行下一次生成工作。

Gated签入生成

Gated签入生成(gated check-in builds)和持续生成也很类似，区别在于它是在某个人提交了修改代码之后显示红色的小旗子符号，但是并不会允许立即将这些修改代码提交到代码库。每次提交代码之后就立即执行Gated签入（像持续生成），或者在特定的时间范围内执行某个数量的生成工作（像滚动生成）。

译注1：有很多自动化生成服务软件工具，如Pulse、TeamCity、QuickBuild等。

计划生成

计划生成（scheduled builds）是根据特定的计划执行，而不需要与特定的提交行为绑定。计划生成方法最流行的例子称为夜间生成（nightly build），因为其生成工作被设定在每夜的特定时间，在开发团队成员结束一天的工作之后，在晚上大家休息的时间里执行新的生成工作。因为这种类型的生成方式不需要和提交代码的行为关联，所以对代码的即时性要求不是很高的项目可以使用这种方法。这种一天左右的缓冲时间还是可以接受的。计划生成通常需要花费更多的执行时间，或许要执行更多的单元测试，如创建安装包。

各种类型的自动化生成方式的主要区别在于，每次生成执行的间隔时间以及每次生成需要耗费的时间。执行生成的每种类型的工作是这样扩展而来的。随着生成频率的降低，每次生成工作花费的时间也会越来越长，因此生成任务的复杂性会逐步增加，耗费的时间也会逐步增多。

例如，持续生成关注的是执行最小的工作量，检验代码库当前最新的质量；而其他方式，每夜生成或者每周生成可能要花费几个小时甚至几天的时间来执行繁重的生成任务，比如执行扩展深入自动化测试套件、编译大量的代码文件，或者为发布工作打包产品套件等。

自动化生成最好的方法通常同时包含几种不同的生成方法，只是优先级不同而已。

例如，可以为自己的项目采用3种不同的自动化生成方式。

1. 每次代码签入都执行持续生成验证代码质量。
2. 每小时执行一次滚动生成，且执行更详细的自动化测试。
3. 每夜生成一次发布当天最新的修改版本，用户或者QA团队尽可能早地跟踪项目进度并报告Bug。

创建自动化生成

383

在确定使用哪种自动化生成方式之后，接下来就可以定义自动化生成项目了。另外，生成服务器还需要一些准备工作才能进行工作：首先，生成脚本，定义包含生成服务器必须执行的任务；其次，源代码（其他的文件）、生成脚本需要根据这些文件来执行任务。

定义生成脚本可以使用Team Foundation Server；在团队浏览器选项卡（Team Explorer tab）里选择生成选项，然后点击“新建生成定义（New Build Definition）”，随后会弹出一个新建生成定义向导（Build Definition Wizard），用于配置新的生成任务。

对这个例子，我们会创建一项持续生成的任务，所以在新建生成定义向导的“生成定义名称（Build definition name）”文本框里输入Continuous，其他选项使用默认值（见图18-1）。

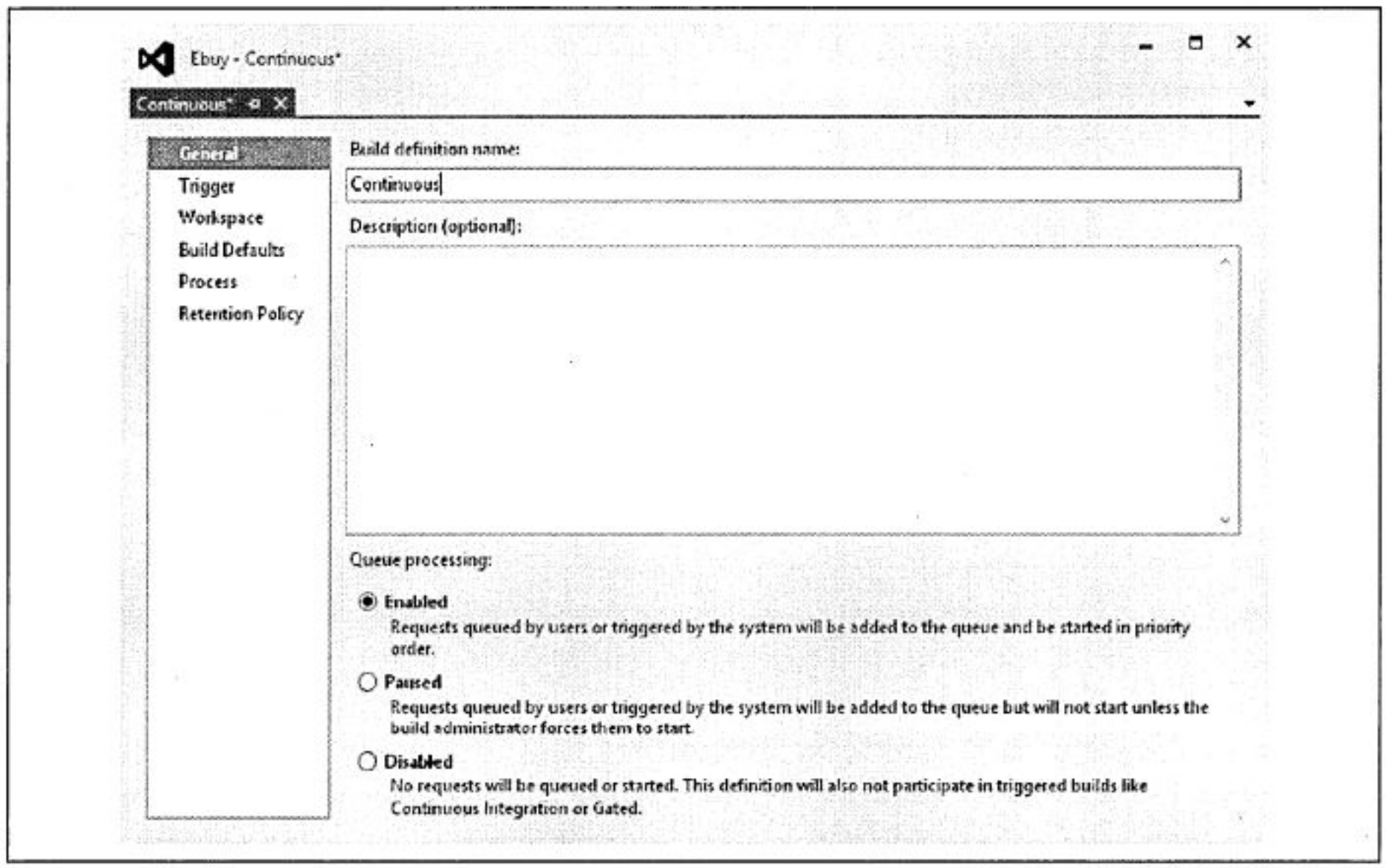


图18-1　新建生成定义向导——通用选项卡

选择“持续集成——创建每个签入（Continuous Integration-Build each check-in）”选项，如图18-2所示。

直接跳过Workspace和Build Defaults选项卡，因为可以直接使用默认值。点击过程（Process）选项卡，找到“Items to Build”的配置属性，如图18-3所示。

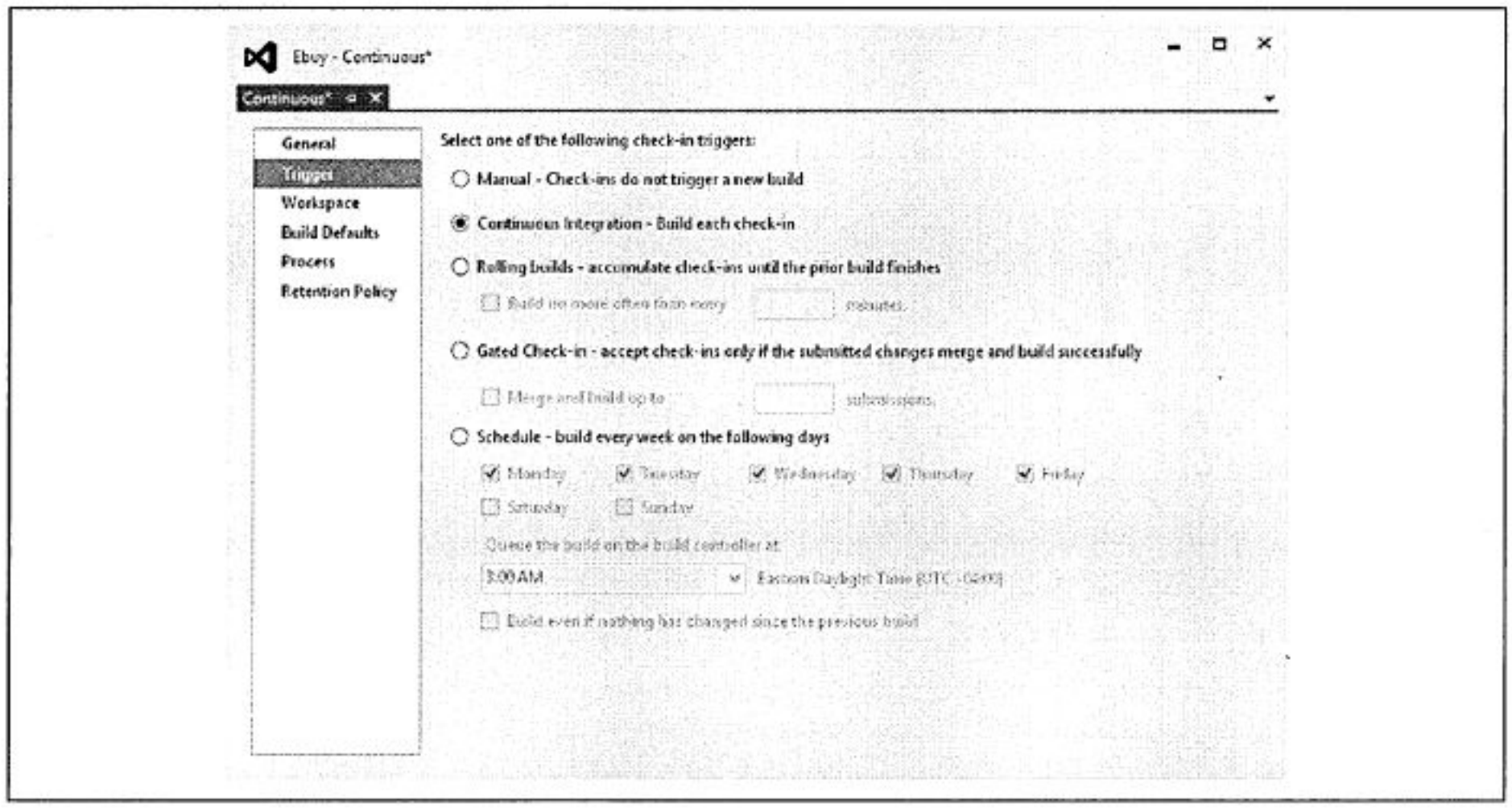


图18-2　新建生成定义向导——触发器选项卡

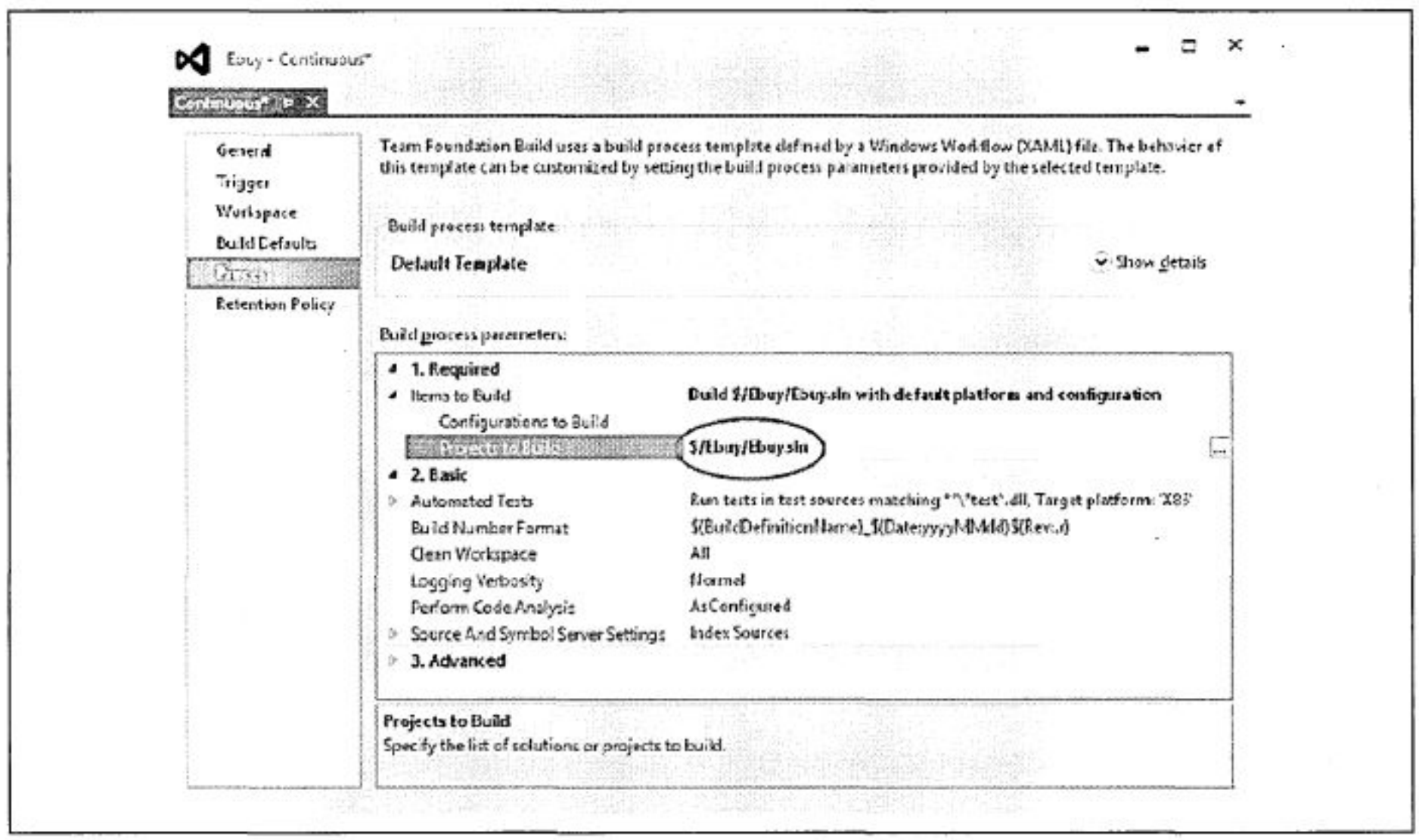


图18-3　新建生成定义向导——过程选项卡

待生成项目（Items to Build）配置属性包含了将要执行的MSBuild项目文件。假设要编译所有文件，则Visual Studio会自动选择当前的解决方案。使用配置属性，也可以指定其他的MSBuild项目文件，也就是可以使用自定义生成逻辑文件。当提供了多个MSBuild项目文件时，MSBuild会根据设置的顺序来依次执行这些生成文件。当然，若有错误，MSBuild就会停止工作。

同样，也要注意自动化测试属性，其默认的行为是执行所有名称匹配***test*.dll的程序集。也就是说，只要创建人和自动化测试项目包含test字符串，Team Foundation Server就会不断执行自动化测试，不需要其他配置。

新建生成定义配置完成以后，保存配置信息（可以直接使用Ctrl-S快捷键）。这时在团队浏览器（Team Explorer）选项卡节点下就会出现新的配置名称（见图18-4）。

为了查看生成配置是否正常工作，现在尝试提交一个新的修改到源代码控制器里，或者用右键单击新建生成，选择“新生成保存到队列（Queue New Build...）”选项，然后点击对话框中的队列（Queue）。Team Foundation Server随后就会开始执行生成工作，从解决方案源代码库里获取最新版本的代码，然后执行设置的生成脚本。

若这项生成任务可以成功执行，就说明刚才的生成配置工作做得不错。如果生成失败，则需要根据生成错误来修改生成配置文件。若再次签入代码，就会自动触发一次新的生成任务。

恭喜！已经创建了一个可以工作的自动化生成脚本。

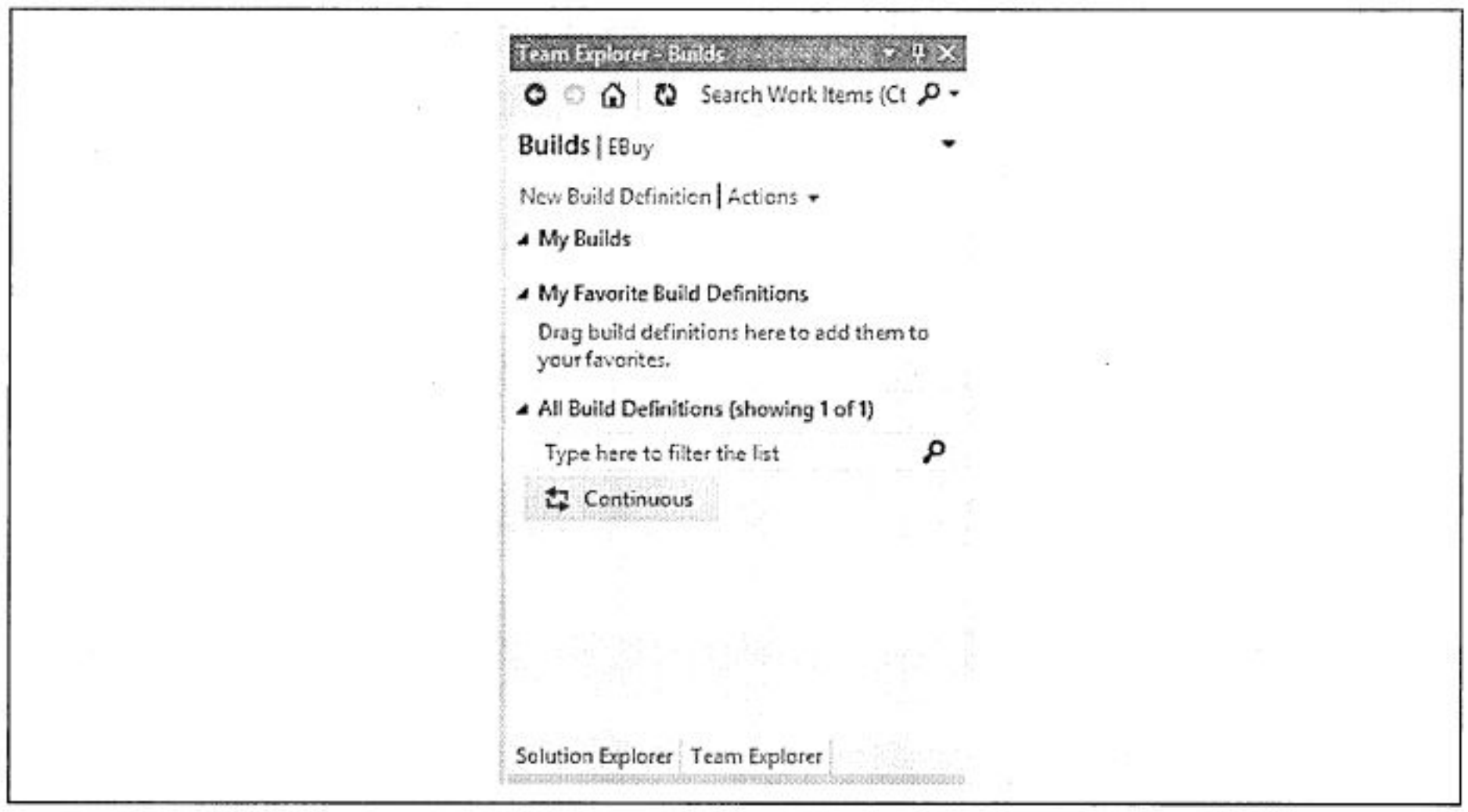


图18-4　新建生成定义向导——团队浏览器选项卡

386

持续集成

Continuous Integration

如果认为上面的自动化生成类型看上去有点枯燥，那么，持续集成相比自动化生成就复杂得多。

事实上，持续集成同样涉及如何使用最少的代码来确保应用程序的质量、如何定期把功能集合持续集成到更大的应用程序中等问题。它可以缩短反馈的周期，并尽快把新功能集成进来；当然，在修复系统Bug的过程中也扮演着重要的角色——只要问题出现在代码库中，就可以进行修复后再持续集成。

使用持续集成时，鼓励团队中的所有成员向中心代码版本控制器提交自己的修改代码，把自己的代码与最新的版本集成在一起。这可以避免某个开发人员修改了本地代码，但是没有提交到代码库。越晚提交代码，代码库中版本和某个开发人员本地代码的差别会越大，合并代码的时间更长。因此，越早、越频繁地让开发人员提交代码，集成生成花在代码合并等工作上的时间会越少。

发现问题

软件开发生命周期（software development lifecycle）包括收集需求、系统分析、开发、测试、部署等步骤。这些步骤是连贯组合的，直到产品发布为止。使用这种方法进行开发时，若开发过程中出现Bug，则可以及时修复；但如果是测试阶段发现Bug，则修复Bug花费的成本就更高了。

原因很简单：为了实现某项功能，而编写完代码后不久，我们还能很清楚地记得代码逻辑，因此发现和修复Bug就会非常快。换句话说，当很久之前编写的代码出现了Bug时，由于忘记了代码逻辑，且代码逻辑中涉及复杂的组件交互，因此发现和修复Bug就非常令人头疼。越早发现问题越好，可以尽快修复Bug。

持续集成原则

下面列举了10条实现高效持续集成代码最核心的原则。也许不需要全部遵守，但是在实现持续集成时，用这些最佳实践原则可以为测试工作带来很大好处。

这些实践原则大部分由Martin Fowler（译注2）总结整理。Martin Fowler是敏捷软件开发的发起人之一。更多关于持续集成的技术资料可以在Martin Fowler网站上查阅：*http://martinfowler.com/articles/continuousIntegration.html*。

维护单一代码库

源代码控制系统提供了一个统一的中心存储库，用于在团队成员之间高效共享代码和其他资源。源代码控制系统也是表示“真理”的地方——中心库是唯一一个存储最新版本的、可运行的应用程序代码的地方。

自动化生成

无论在什么机器上，都需要两步才能完成生成应用程序的任务。

1. 从代码库获取最新代码。
2. 执行单个命令。

为了正常执行生成工作，必须确保源代码库包含所有生成所需要的内容，外部核心系统级的依赖关系（比如操作系统的某些功能、数据库服务或者.NET框架等）除外。在生成和执行应用程序时，不需要再安装任何其他内容。当安装工作无法避免时，这些内容应该被包含在自动化生成任务中，在执行生成时自动安装完成。

当包含数据库时要格外注意一条原则：只要有可能，就尽量不使用远程数据库；相反，自动化生成应该包含数据库schema和初始化数据库的种子数据（seed data）。最好不在本地安装任何数据库就可以完成安装工作（例如，使用微软的本地数据库LocalDb的嵌入版本，而不是安装Microsoft SQL Server or SQL Server Express版本）。

译注2：Martin Fowler 的个人网站是：*http://martinfowler.com*。Martin Fowler 是享誉世界的技术专家，敏捷开发方法的创始人之一，ThoughtWorks 公司的首席科学家。他在面向对象分析设计、UML、模式、软件开发方法学、XP、重构等方面有深入研究，并发表了很多文章和技术书籍，其中《分析模式》、《UML 精粹》、《重构》、《企业应用架构模式》、《NoSQL 精髓》基本上是架构师必备的技术书籍。真正的大师，IT 领域目前还没有这种人物。

让生成任务自己测试

这条原则是对本书中的所有模式和实践的最好总结。SOLID原则和松耦合架构为快速、高效的单元测试铺平了道路，可以让我们随时随地测试代码。

持续集成过程可以根据代码签入活动，通过执行自动化测试套件测试系统，尽可能早地发现问题。在持续集成过程中出现错误时，可以迅速定位出错的代码，修改Bug也非常容易，影响也小。

不知道如何检验自己生成代码质量的持续集成（continuous integration）是没有太大价值的。

开发人员经常提交代码

为了进行开发工作，开发人员必须在本地保存一份源代码副本。随着时间的推移，当本地代码和源代码存储库断开时，开发人员在本地修改代码的机会就会越来越多。如果长期允许这种开发模式，那么代码差别日积月累，越到后期越要花更多的精力来合并代码。

因此，团队开发人员要保持本地代码和源代码存储库的同步，避免长时间断开连接。对于本地长期积累修改的代码，每天至少提交一次，最好一天提交多次，或者每次大的修改之后尽快提交代码。

只有这样，团队中的每个成员才可以和源代码存储库保持同步，而且源代码存储库里存储的是当前项目最新的代码。

每次提交都要在集成机器上重新生成

程序员常说的一句话：我机器上没问题。这句话的意思是指，程序员通常可以在自己的机器上完美无缺地调试、运行程序。这句话一方面可以说，暗示这程序员已经实现了某项功能或者修改好了某个Bug；另一方面也是个非常危险的信号，表示这个程序可能在其他机器上无法运行。

为了处理这个问题，持续集成要求配置一台机器（或者多台机器）——通常称为集成服务器（integration machine）——编译并执行每个代码库提交的修改。集成服务器尽量和产品环境一样，越接近越好，这样，如果应用程序能在集成服务器上编译并执行成功，那么就可以在产品环境下正常运行。

在使用集成服务器检验每次提交代码之后，在开发机器上的程序运行实际上就不在考虑范围内了，只需要集成服务器验证即可——应用程序必须在集成服务器上运行成功，如果失败，那么表明这次生成过程是失败的，必须立即修复。

快速生成

由于持续集成的核心是在问题引入的时候可以尽可能早地发现问题，越快越好。因此，当务之急就是尽快完成持续集成，尽快报告发现的问题。

引入问题和发现问题之间的时间越久，修改Bug花费的成本越多。

在克隆产品环境中测试

当应用程序在产品环境不同的服务器上运行时，即使出现错误，也和以后产品环境的关系不会太大。所以要尽可能模拟真实的产品环境来执行生成任务。

每个人都可以方便获取最新的可执行版本

持续集成鼓励经常反馈意见，允许客户在应用程序开发阶段扮演积极的角色。

能够访问最新生成的程序集的好处很大，但是对Web应用程序，有时候要等Web程序部署到测试服务器才行，这样每个人都可以测试网站并提供自己的反馈意见。

每个人都可以看到整个过程

也许听过这个古老的哲学问题，“森林中有一棵树，它周围没有任何树，如果这棵树倒下了，那它会发出声音吗？”这个问题也适用于软件开发领域：“如果一次生成失败，没有人知道（关注），那这次失败还有价值吗？”

持续集成的状态与代码块甚至整个项目的整体健康状况有直接关系。这可以进一步理解为：无论生成工作是成功还是失败，实际上是指能否满足某些代码生成的标准，例如，执行生成花费的时间是否超出预期，或者生成过程中的代码覆盖率是否达标。因此，对于涉及项目的每个人来说，了解自动化生成的细节非常重要，要详细了解生成工作执行的每步的详细情况。

人们想出了许多办法让团队成员及时了解生成项目的状态，从系统提醒软件到巨大的悬挂式显示器，各种办法应有尽有。

当然，最重要的是让每个人知道生成工作的状态，而不是传播信息的方式。所以，应该根据项目的实际情况选择一种自己的共享生成状态的方式。

对于持续集成来说，要理解的最重要概念就是：当生成失败时，意味着应用程序哪里出了问题，那么从现在开始最重要的工作就是修复这个Bug。当生成失败时，团队要放下所有的工作，立即修复这个问题——修复应用程序——这项任务的优先级最高。

失败的生成任务应该成为一面镜子。如果反复遇到不是应用程序导致的失败（例如，网络连接问题、第三方API导致的错误等），就需要考虑重新组织生成任务了，或者重构测试代码，以避免这种错误再次发生。

在这种阉割版本环境中，持续生成程序无法准确反映代码库的健康状况，因而价值也会大打折扣。也许我们会逐步开始适应各种失败的生成过程，而且不会再关注它们。

失败的生成结果通常意味着代码库存在问题。我们绝对不能接受或者忽视这个事实，必须立即找出问题，修复Bug。

自动化部署

持续集成原则关注于如何让用户方便使用应用程序。正如最好在克隆产品环境中测试应用程序一样，成功的持续集成生成应该包含自动化部署到产品环境的步骤。

自动化部署有两大好处。首先，它提供了可以看到并使用到最新版本的应用程序的好处，对测试Bug和验证修复的Bug都非常简单。

其次，把应用程序尽可能地部署到一个接近产品环境的服务器中，这样可以把持续集成工作的价值从验证应用程序扩展到验证应用程序部署中。而且，自动化测试可以帮助我们尽可能早地发现代码库中存在的Bug，自动化部署也可以帮助我们尽早发现部署的问题（比如丢失依赖或者安全、权限问题）。

虽然无法回避部署问题，但是，至少可以在产品发布后到最终的部署环境之前模拟几次部署经历。这也意味着，我们可以清楚将来应用程序部署到产品环境中可能遇到的所有问题，而且清楚如何在发布产品的时候解决这些问题。现在这些工作变成了（自动化）部署过程中的一步。

读者可以查阅第19章中关于自动化部署的更多详细内容。

总结
Summary

本章主要介绍了在软件项目开发过程中如何尽可能多地使用本书中介绍的开发模式和实践经验（pattern and practice）工具来自动化完成生成任务，以减少开发人员的工作量。只要在项目开发中使用了SOLID面向对象的开发实践原则以及自动化测试套件来验证应用程序，就可以持续确保软件开发项目的质量。

第五部分

实施

Going Live

第19章 部署

Deployment

即使我们在ASP.NET MVC框架下花大量的时间开发出了世界上最好的网站，也只有当网站文件部署到Web服务器上，让用户可以访问网站时才真正体现出它应有的价值。拷贝网站文件到服务器并发布网站的这种行为通常称为**部署**。当然，这个概念并非是ASP.NET MVC网站独有的，这个概念适用于所有的软件系统。

本章将介绍一些最流行的互联网网站部署方法——从简单的文件拷贝部署方式到使用最新的高扩展性的云主机部署。

学习本章时，请记住，许多网站都有其独特的部署需求，这里介绍的各种部署方式也许无法直接应用到实际项目部署中。但是，可以把本章中介绍的各种不同的部署工具和方法当成备选方案，选择适合实际项目部署需要的工具或者技术，而不需要再查询详细的部署文档。所以，不要把本章看成是一份带有详细步骤的“how-to”（“怎么做”）指南，可以试着把它当成对不同部署方式的概述——这些部署方式是任你挑选的——并留心那些适合的工具和技术。

需要部署什么

What Needs to Be Deployed

在开始拷贝网站文件之前，先讨论一下我们要做什么工作。概括地讲，绝大部分网站都要依赖以下三种类型的内容：.NET程序集和各种包含网站业务逻辑的文件、网站需要的所有自定义内容（如CSS或JavaScript文件）、网站运行时需要使用的外部依赖（如数据库或外部服务）。

网站核心文件

ASP.NET Web应用程序至少包含一个/bin文件夹，这个文件夹保存由编译应用程序代码所生成的程序集和该程序依赖的其他.NET程序集。因此，/bin文件夹是ASP.NET Web应用程序部署策略的最关键部分。

还有其他一些特殊文件，虽然它们可能不是网站正常运作所必需的，但是包含了网站重要的配置信息。除了/bin文件夹外，这些文件，例如web.config和Global.asax，通常都会包含在网站中。

如果做过ASP.NET Web Form网站开发，那么不会感到惊讶，因为上面介绍的内容在ASP.NET Web Form网站部署中都有涉及。

然而，当涉及视图时，ASP.NET MVC Web应用程序部署与传统的ASP.NET Web Form应用程序部署迥然不同。除了/bin文件夹和一些特殊的ASP.NET相关文件外，ASP.NET MVC Web应用程序还要包含自己的视图文件（/Views文件夹）以及其他需要部署的内容。这是因为ASP.NET MVC视图文件采用的是和ASP.NET Web Form .aspx视图一样的Just-In- Time (JIT)编译、部署及维护方式。实际上，两者相差不大。

“bin部署”ASP.NET MVC库

为了使ASP.NET MVC网站能够运作，被部署的应用程序要能访问ASP.NET MVC框架程序集。这可以通过两种方式实现：直接在Web服务器上安装ASP.NET MVC框架，或者把所有的ASP.NET MVC程序集和其他的程序集一起拷贝到网站的/bin文件夹中。

在Web服务器上安装ASP.NET MVC框架非常简单，就像第1章里介绍的一样——只需在服务器上运行Web平台安装器（Web Platform Installer）就可以选择安装ASP.NET MVC框架包了。

实践证明：避免在Web服务器上安装任何东西。我们可能更倾向把ASP.NET MVC程序集直接拷贝到网站的/bin文件夹下，和其他的程序集一样。这种方法通常被称为“bin部署”，这也是最简单、最稳定、最方便维护的部署方法。

Visual Studio使得“bin部署”变得非常简单，它包含一个菜单项，可以自动把ASP.NET MVC依赖项添加到应用程序中。即在Visual Studio中用右键单击ASP.NET MVC项目，然后选择“添加部署依赖（Add Deployable Dependencies...）”选项，如图19-1所示。

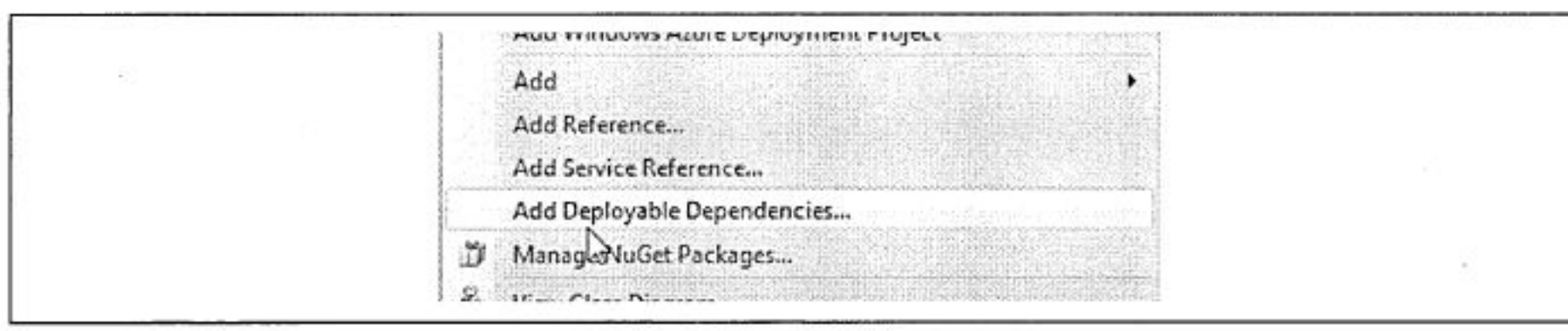


图19-1　添加ASP.NET MVC依赖的菜单选项

选择该选项后，Visual Studio就会弹出一个窗口，让我们选择要添加哪些依赖程序集到网站中。选择ASP.NET MVC选项，如图19-2所示，点击“OK”按钮即可。

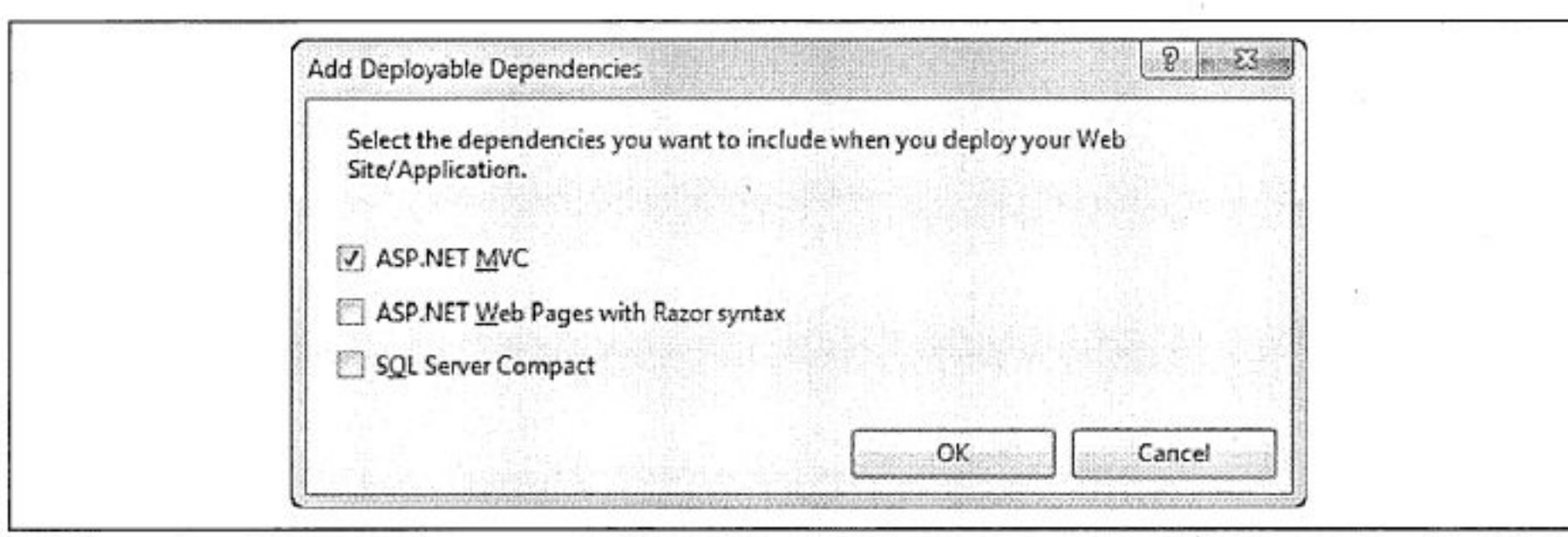


图19-2　选择ASP.NET MVC选项

Visual Studio随后会创建一个称为/_bin_deployableAssemblies的新文件夹，它包含部署要使用的ASP.NET框架程序集（见图19-3）。

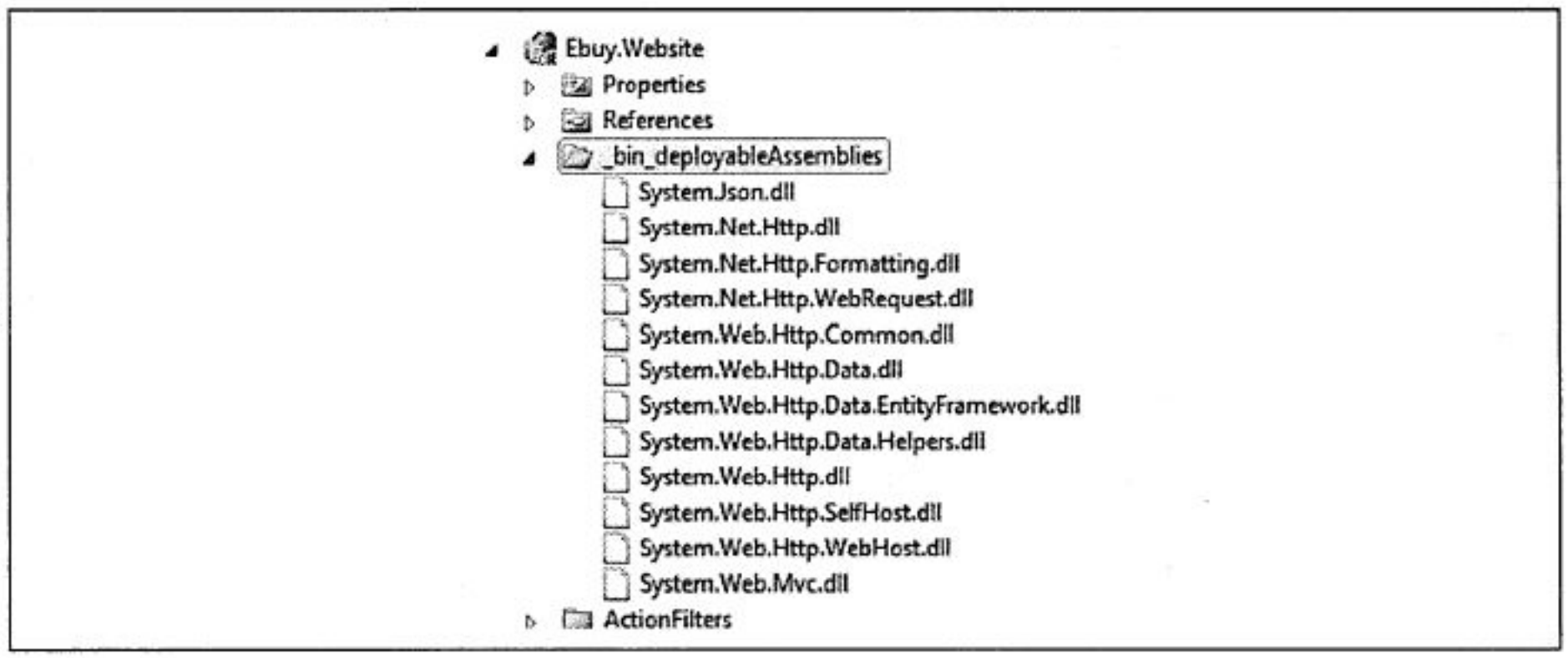


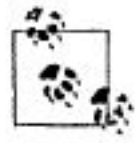
图19-3　新创建的/_bin_deployableAssemblies文件夹

现在，当发布网站时，Visual Studio会把/_bin_deployableAssemblies文件夹中的程序集连同其他程序集一起发布到服务器。

398

静态内容

静态内容（static content）通常可以是任意类型的文件，但是这些文件主要是定义客户端逻辑和样式的JavaScript、CSS样式文件及图片。虽然这些文件可以保存在网站的任意文件夹下，但是ASP.NET MVC项目模板为它们分别创建了/Scripts、/Images及/Content文件夹，可以直接保存JavaScript、图片文件、CSS文件到这些文件夹下。因此，如果使用了这些惯例，那么这三个文件夹应该可以保存网站中的所有静态内容文件。



请确保在Visual Studio项目里把每个静态文件的Build Action属性设置为Content，这样Visual Studio就会把这些静态文件同网站一起部署。

为什么不部署？

如果使用了ASP.NET MVC项目模板定义的惯例，那么每个要部署的ASP.NET MVC网站的目录结构看起来就应该是相似的，如图19-4所示。

注意，这个目录结构与我们在Visual Studio中开发网站时的结构差别很大（见图19-5）。

更特别的是，部署版本的网站是不会包含ASP.NET网站的任何后台逻辑源代码文件的（译注1）。因为项目在部署之前已经编译完成，所以这些后台源代码文件会被编译成不同的程序集文件保存在/bin目录中。了解了这些情况以后，就可以随意保存和组织项目的源代码文件了，因为这些文件夹都不会成为部署版本应用程序的部分。

399

译注1：前台文件JavaScript和CSS是可以的。

图19-4　通用ASP.NET MVC 部署层次结构

图19-5　默认的ASP.NET MVC网站项目结构

数据和其他依赖

虽然几乎每个ASP.NET MVC网站部署版本的文件夹结构与图19-5所示的结构图很像，但是ASP.NET MVC项目开发版本和部署版本的项目结构还是存在一些区别，尤其是前面提到的源代码文件夹结构，只是核心部分不会变化。到目前为止，ASP.NET MVC 应用程序部署的基本情况已经介绍完。下文开始介绍一些独特的部署需求。

除了网站物理文件外，绝大部分网站部署到服务器上都会依赖其他的项目，比如从数据库中查询数据，或者与外部的Web服务进行交互。这些外部依赖关系只有在应用程序松耦合、分布式部署或者采用面向服务的架构（Service-Oriented Architecture，SOA）时才会用到。

虽然研究网站部署如何与这些外部依赖项目进行交互超出了本书的范畴，但是可以帮助我们从全局角度探讨这种架构的网站部署工作。下面的问题可以帮助我们发现各种依赖关系，以及了解部署网站需要做的工作。

1. 要部署的应用程序需要什么系统级应用或API（如OS版本、IIS版本、.NET框架版本）？
   - 服务器上需要安装其他软件吗？
2. 应用程序需要什么系统级的文件夹或文件？
   - 应用程序需要特定的文件夹路径吗？（这种情况应该尽量避免）
3. 应用程序需要数据库吗？
   - 如果需要，那么数据库schema自上一版本后有没有更新？
   - 该应用程序是否需要特别的数据库用户？如果需要，那么针对该用户的数据库访问权限配置正确吗？
4. 应用程序需要和其他服务器或服务交互吗？
   - 访问它们需要修改网络设置（如防火墙规则、用户或角色安全）吗？
5. 软件有没有购买正版的序列号？

注意，这些问题并没有包含成功发布应用所必备的所有工作，但可以帮助我们处理绝大部分问题，而且可以根据实际情况来调整发布策略。

发布EBuy交易网站需要做的工作

现在以发布网站为例，先理清在部署EBuy交易网站需要做的工作，比如依赖项目、配置环境等。

- **系统级API和服务**。EBuy交易网站是一个十分基础的网站，它只依赖.NET 4.5框架，不使用任何系统级API和服务。.NET 4.5框架提供了所有API依赖项，包括ASP.NET MVC 4框架——在/bin文件夹中（当然是必须部署的（译注2））。
- **视图文件、脚本文件、CSS样式文件及图片**。除了像.NET框架和包含网站逻辑代码的程序集依赖项，视图文件、脚本文件、CSS样式文件及图片通常也是网站正常运行必备的资源。需要在部署网站时确保这些资源已被拷贝。
- **数据库**。EBuy交易网站是数据驱动的网站，使用了Entity Framework Code First（代码优先）数据模式与后台数据库交互，以持久化应用程序的数据。
- **上传图片的存储场所**。当用户创建新的拍卖商品列表时，可以选择是否提交商品的图片。这些图片应该保存到网站的某个地方。当然，实际保存图片文件的位置和方法有很多种，这取决于Web服务器的架构是单服务器还是服务器集群，或者使用了类似Azure的云主机托管服务。无论图片保存在哪里，应用程序都必须要有访问该位置物理路径的权限（例如，网络或文件系统），而且还要有适当的读/写图片的权限。

搞清楚了所有这些问题，并且确定了部署网站要做的一切准备工作后，现在就可以开始部署网站了。

401

部署到IIS服务器

Deploying to Internet Information Server

托管ASP.NET MVC网站最常见的情况是使用Internet Information Server (IIS)来创建并配置网站。正好，ASP.NET MVC网站绝大部分情况下与其他任何ASP.NET网站是一样的，如果已经对部署ASP.NET Web应用程序到IIS的过程十分熟悉，那么部署ASP.NET MVC网站是小菜一碟，几乎没什么区别。如果是第一次在IIS上部署ASP.NET网站，也不要害怕，下面几节内容会详细介绍部署网站的步骤。

必备条件

在创建并部署自己的网站之前，首先要确定目标Web服务器已经具备了托管ASP.NET MVC网站的必备条件。在.NET框架早期的版本中，部署ASP.NET应用程序和其他必备条件到Web服务器上需要相当大的工作量。

译注2：也可以在Web服务器安装.NET 4.5。

幸运的是，目前已经发展到这一步：我们唯一要做的事情就是安装.NET框架（4.0或更新的版本），因为已经安装好IIS了（译注3）。

部署ASP.NET MVC框架程序集

ASP.NET MVC 4也需要部署程序集，下面介绍两种部署方式。部署时任选其中一种即可。

1. 如第1章介绍的一样，使用ASP.NET MVC 4安装器安装。
2. 使用本章前面介绍的bin部署（bin-deploying）方法将ASP.NET MVC框架程序集包含到网站的Bin文件夹中。

如果计划在单个服务器上运行多个ASP.NET MVC 4网站，那么有必要考虑第一种选择：直接安装ASP.NET MVC 4框架，而无需担心别的事情。

选择这个方法的唯一一个有说服力的理由是，在应用程序中可节省ASP.NET MVC框架程序集所占用的磁盘空间。

在几乎所有的情况下，都建议采用第二种方法来部署ASP.NET MVC框架程序集到网站bin文件夹中，正如部署其他网站程序集一样。使用这种方式部署，网站应用程序更易于管理、维护，甚至可以独立更新每一个网站，而无需担心服务器级别的软件更新可能影响到其他网站。

402

创建并配置IIS网站

在IIS里创建网站十分简单。

首先，新建网站文件夹以存放要托管的网站文件，例如，C:\inetpub\wwwroot\Ebuy。然后，打开IIS管理器（Internet Information Services (IIS) Manager），用右键单击“默认网站（Default Web Site）”，再选择“添加应用程序（Add Application...）”（见图19-6），就会出现添加应用程序的对话框。

在图19-7所示的对话框中，输入网站的名字（如Ebuy），然后选择在上一步中创建的存放网站文件的目录（*C:\inetpub\wwwroot\Ebuy*）。

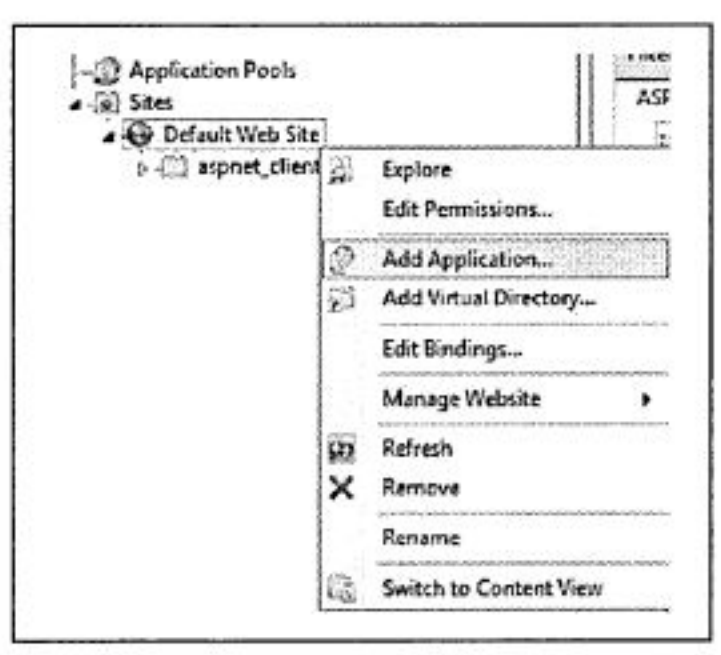


图19-6　创建新的IIS网站

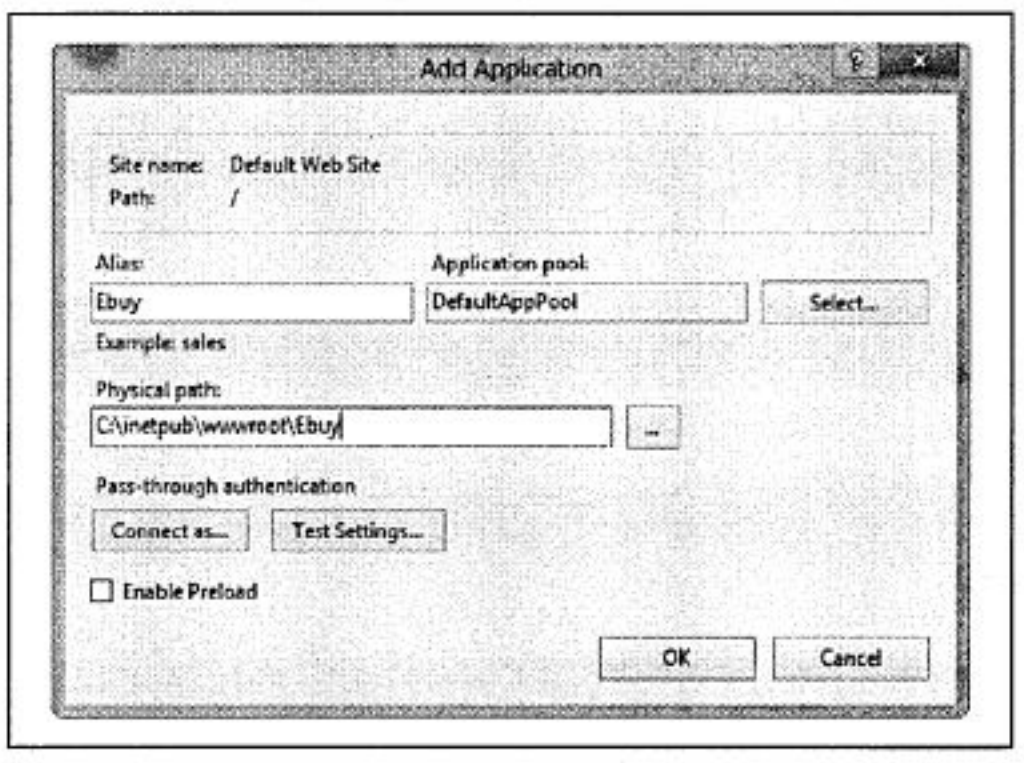


图19-7　添加应用程序对话框

译注3：如果系统没有安装IIS，那么还需要安装对应的IIS Web服务器。

对其他选项，可以使用默认的设置。但是，为了更好地运营管理网站，点击“应用程序池(Application pool)”旁边的选择(Select...)按钮，就会弹出选择应用程序池对话框（Select Application Pool），以确保默认的应用程序池使用的版本是.NET 4.0（见图19-8）。

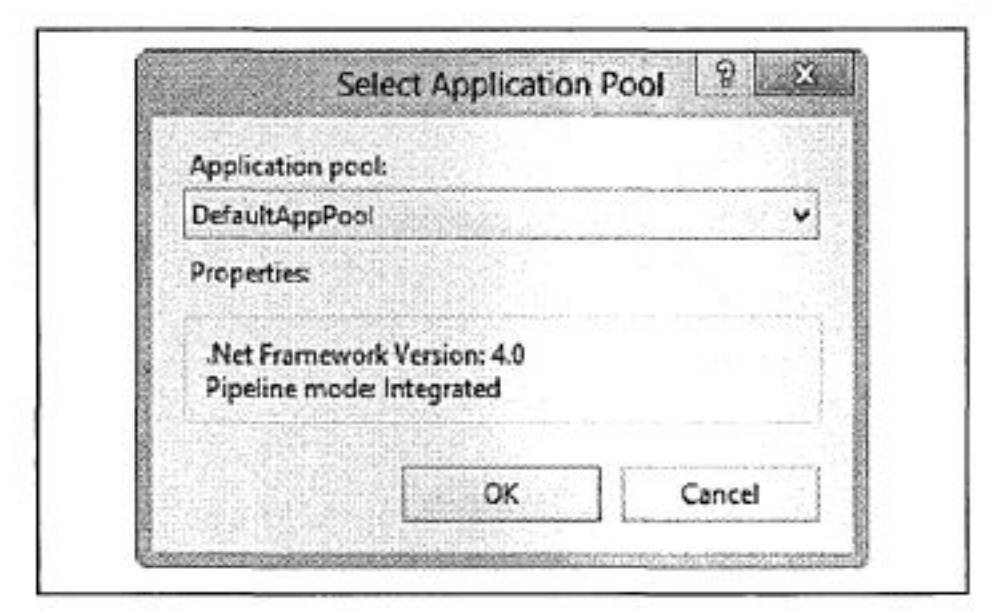
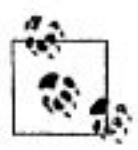


图19-8 默认的应用程序池使用的是.NET 4.0框架

如果默认的应用程序池配置的不是.NET 4.0版本，那么要重新创建一个新的使用了.NET 4.0版本的应用程序池。

如果在可用的.NET框架列表里找不到.NET 4.0框架的选项，那么有可能是由于.NET 4.0框架安装不正常导致的，可以尝试卸载并重新安装.NET 4.0框架。如果有必要，可以运行`%FrameworkDir%\%FrameworkVersion%\aspnet_regiis.exe`命令以便在IIS里配置.NET框架。

最后，点击“OK”按钮可让IIS帮助我们创建网站。现在就有自己可以部署的网站了。



如果在IIS 6里托管旧版本的ASP.NET MVC（1.0 和2.0）网站，那么需要特殊的设置来让ASP.NET MVC支持无文件后缀名的URL路由。现在这已经不是问题了，因为最新的ASP.NET 4已经支持无URL文件后缀路由机制了。

注意，如果在Windows Vista SP2、Windows Server 2008、Windows Server 2008 R2 SP2、Windows 7上运行IIS 7或IIS 7.5，就需要在系统中应用一个批处理文件，可以参考这个网址：http://support.microsoft.com/kb/980368。

使用Visual Studio发布

一旦在IIS中创建并配置了应用程序，就可以部署网站了。有几种部署方式可供选择。使用最多的部署方式是Visual Studio内置的发布功能。

要使用这种方式发布网站，可以用右键单击ASP.NET MVC项目，然后从上下文菜单中选择“发布”（Publish...）选项（见图19-9），以便打开发布向导（Publish Webwizard）。

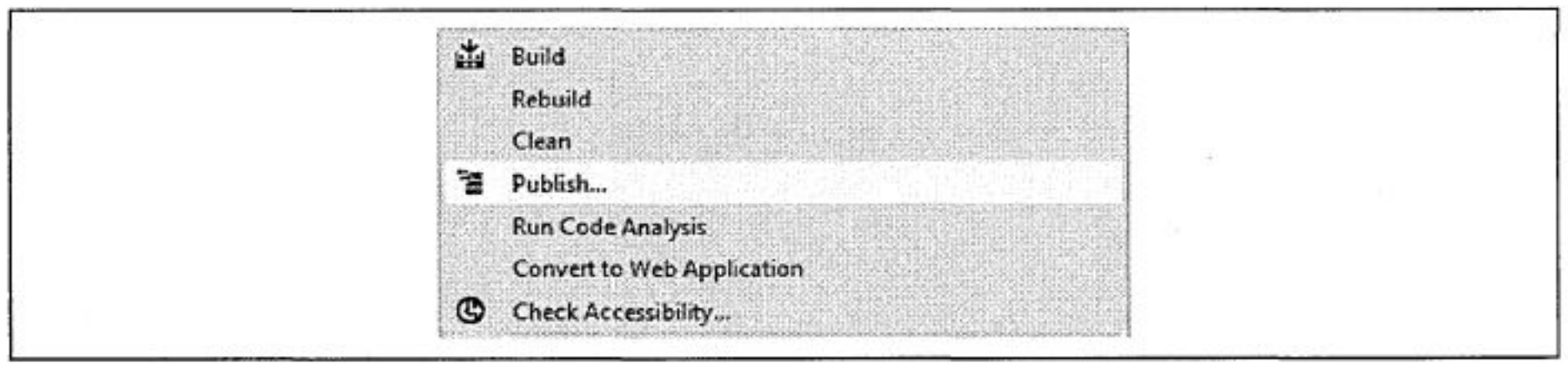


图19-9 打开Visual Studio发布向导

要创建发布网站的配置参数，可以在配置选项卡的下拉列表框中选择“新建（<New...>）”选项，并输入新的配置名称（如Local IIS Website（本地IIS网站））。然后，因为要部署在本地文件系统中，所以从可用的发布方式中选择“文件系统（File System）”选项，如图19-10所示。

“文件系统（File System）”发布选项只适用于对目标Web服务器的文件系统有直接网络访问权限的情况。如果使用的是Web托管服务，如Web空间，则可能无法直接使用文件系统拷贝方式，所以只能选择FTP发布方式来部署网站，这也是主流Web主机支持的方式。

一旦选择了某种发布方式，Visual Studio向导就会要求填写该发布方法对应的其余配置信息。可以使用不同的配置文件名来保存不同的发布配置文件，然后点击“保存（Save）”按钮。

配置了所有需要的发布方法选项后，点击“发布（Publish）”按钮，Visual Studio就会把网站部署到特定的位置上。成功部署完网站之后，Visual Studio就会自动打开浏览器，然后导航到新部署网站的主页上。

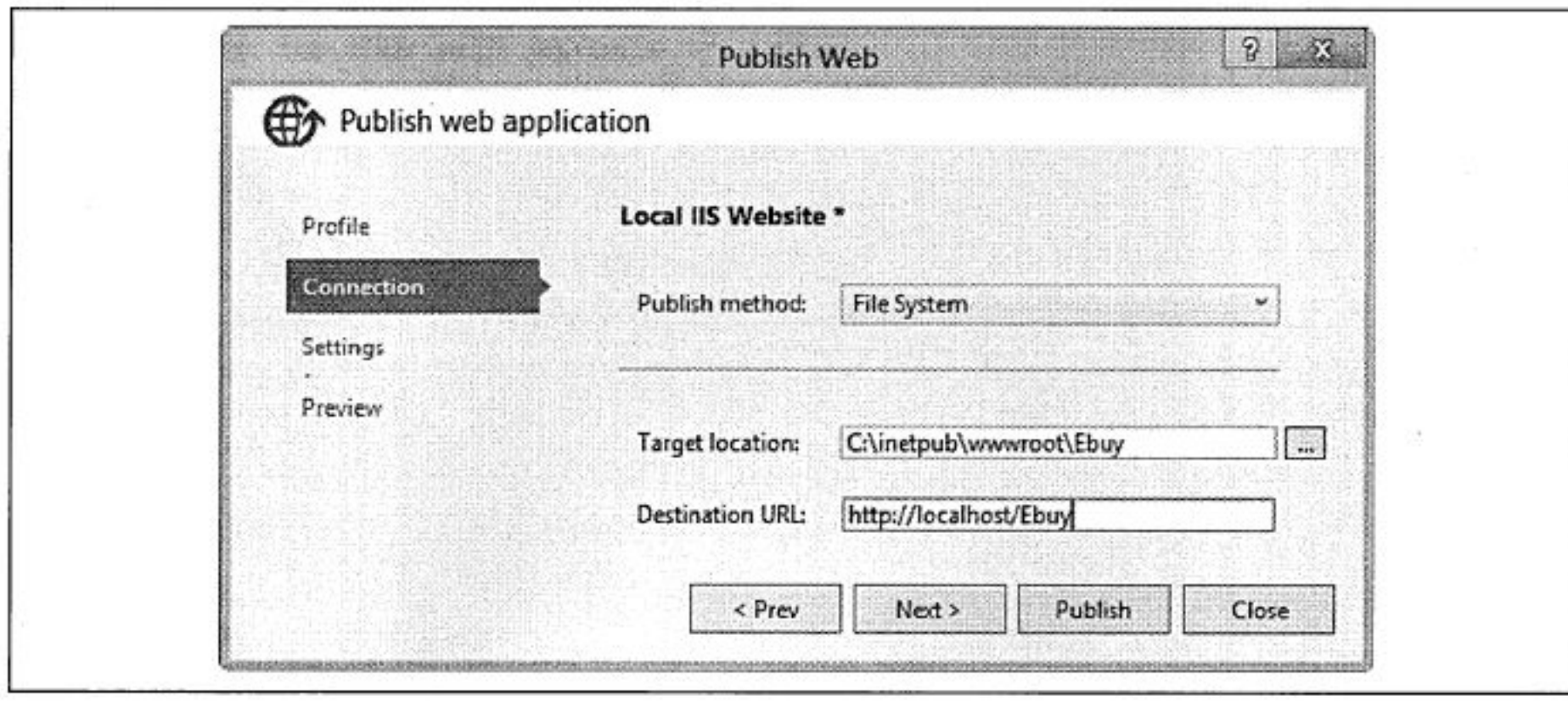


图19-10　发布Web网站向导

如果完全按照上述步骤在本机尝试发布ASP.NET MVC网站，但是最终结果还是失败了——这里是故意这样安排的，以便展示如何诊断部署问题。当发布网站时，Visual Studio会在输出窗口记录所有与发布相关的日志信息，部署过程中遇到的任何问题，它都会显示出来。

例如，刚刚试图执行的部署工作失败了，这也许是因为我们没有文件夹*C:\inetpub\wwwroot\Ebuy*的访问权限。如果是这种情况，就可以在Visual Studio的输出结果窗口里看到“拒绝访问（ACCESS DENIED）”的提示消息。为了修复这个错误，可以给目标文件夹更新读/写权限，然后再次尝试发布网站。这次网站发布应该可以成功了，可以看到一个浏览器窗口打开了刚才部署的网站。

使用MSBuild拷贝文件

如第18章所述，尽可能地自动化程序开发过程是个好主意——自动化用于部署是再合适不过了。尽管Visual Studio发布网站机制非常方便，但如果每次发布网站都要打开Visual Studio，那就太麻烦了。所以，现在看看如何使用MSBuild脚本控制Visual Studio发布向导程序（Publish Web Wizard）来自动发布网站。

下面的例子展示了MSBuild脚本的代码，首先它会指定MSBuild编译整个解决方案，然后会使用拷贝任务把生成的网站部署文件拷贝到目标文件夹中：

```
<?xml version="1.0" encoding="utf-8"?>
<Project ToolsVersion="4.0" DefaultTargets="Deploy"
         xmlns="http://schemas.microsoft.com/developer/msbuild/2003">

  <PropertyGroup>
    <BuildDir>$(MSBuildProjectDirectory)\build\</BuildDir>
  </PropertyGroup>

  <Target Name="Deploy">
    <MSBuild Projects="EBuy.sln" Properties="OutDir=$(BuildDir)" />

    <ItemGroup>
      <WebsiteFiles Include="$(BuildDir)\_PublishedWebsites\Ebuy.Website\**" />
    </ItemGroup>

    <Copy SourceFiles="@(WebsiteFiles)"
          DestinationFiles="@(WebsiteFiles->'$(DeploymentDir)\%(RecursiveDir)%↵
          (Filename)%(Extension)')"
          SkipUnchangedFiles="true"
    />
  </Target>
</Project>
```

为了执行这个脚本，要打开Visual Studio 命令提示符（Command Prompt），导航到Ebuy网站解决方案所在的文件夹，然后执行下面的命令：

```
msbuild.exe deploy.proj /p:DeploymentDir="_[Path to Destination]_"
```

这会生成网站，并且直接输出到临时的生成目录中（生成在当前目录），然后拷贝MSBuild为发布网站创建的PublishedWebsites文件夹中的内容到部署网站的目标文件夹中。

使用MSBuild执行数据库脚本

对部署文件来说，Visual Studio文件系统发布（File System Publish）和MSBuild发布机制都是很好的选择，但是，当网站使用数据库时，还可以这样简单通过拷贝文件来发布网站吗？

Entity Framework Code First（代码优先）中的一项功能就是可以自动管理数据库的版本——可以告诉Entity Framework，在网站的启动阶段，如果看到模型发生变化就必须改变数据库架构（database schema）时，则应自动升级数据库。如果没有使用Entity Framework Code First（代码优先）功能，就需要根据实际系统的数据库变化情况来部署对应的版本了。

当部署的数据库发生变化时，通常有以下两种选择。

1. 每次都重新创建一个新的数据库。
2. 让数据库根据脚本情况自动升级。为了让目标数据库保持最新的schema而执行脚本文件。

显然，第一种方法在开发过程中最为简单方便——因为这种方法无需担心任何现有的数据库升级带来的潜在风险。然而，除非只想发布一次产品数据库，否则，这种方法无法符合最终的产品部署要求，违反了持续集成的基本精神，即尽可能多地测试产品部署以便尽可能早地发现问题。

如果不考虑第一种方法，第二种方法即通过脚本增量修改数据库schema的方式就是实际的选择。

幸运的是，使用MSBuild和SQLServer的SQLCMD工具实现上述两种部署方式都非常简单。若想给生成脚本添加SQL脚本，则只需首先在MSBuild文件里添加以下代码：

```
<Target Name="DeployDatabase">
    <ItemGroup>
      <ScriptFiles Include="$(ScriptsDir)\*.sql" />
    </ItemGroup>

    <Exec Command="sqlcmd -E -S $(SqlServer) -i&quot;%(ScriptFiles.FullPath)&quot;" />
</Target>
```

这些代码的功能就是在指定的路径($(Scripts-Dir)*.sql)中查找对应的SQL脚本，然后针对$(SqlServer)属性指定的SQL数据库实例使用SQLCMD执行每个SQL脚本文件。注意，执行这些脚本文件的顺序和MSBuild发现时的顺序是一样的。这个顺序也就是脚本当初在文件系统中的顺序——根据文件名排序，所以，如果想控制脚本文件的顺序，则可以使用一些命名惯例来命名脚本文件，如添加数字前缀。

然后可以使用下面的命令（所有命令在同一行）让MSBuild执行SQL脚本，自动生成数据库：

```
msbuild.exe deploy.proj /t:DeployDatabase /p:ScriptsDir=Scripts /p:SqlServer=.\SQLEXPRESS
```

SQLCMD工具会在安装标准版Microsoft SQL Server数据库时被安装。但是不是只有安装了Microsoft SQL Server才能使用SQLCMD。

作为替换方案，可以直接安装免费的Microsoft SQL Server Feature Pack工具包，而不需要安装Microsoft SQL Server。可以通过搜索引擎搜索这个安装包来安装最新版本的Microsoft SQL Server Feature Pack工具包，或者直接下载Microsoft SQL Server 2008 R2 SP1 Feature Pack进行安装。微软官方网址为：*http://www.microsoft.com/en-us/download/details.aspx?id=26728.*

部署到Windows Azure

Deploying to Windows Azure

如果不喜欢自己托管网站，并且想利用云平台的伸缩性，那么可以选择微软的云托管平台——Windows Azure。使用Windows Azure，可以只关注网站开发，让微软管理维护基础设备，我们只需要通过互联网发布和维护网站即可。

本节剩下的内容将会介绍如何使用Windows Azure部署自己的网站。学习结束后，我们会拥有一个寄宿在云端的公共网站。

创建Windows Azure账号

在使用Windows Azure部署网站之前，必须注册Windows Azure账号。为此，要访问微软的Windows Azure官方网站：*www.windowsazure.com*。在官方网站上直接点击“免费试用（Free trial）”或“注册（Register）”，然后输入信息就可以创建自己的账号。

一旦注册完毕，就可以进入Windows Azure账号管理主页（Management Portal），在线管理

云主机服务个人账户（译注4）。

创建新的Windows Azure网站

为了在Windows Azure门户创建新的网站，需要点击个人账号管理页面左下方的“网站（Web Site）”选项，然后点击“创建带数据库的网站（Create with Database）”来打开New Web Site wizard窗口（见图19-11）。填写网站需要的基本信息之后，在数据库下拉列表框中选择“创建新的SQL数据库（Create a new SQL database）”选项。

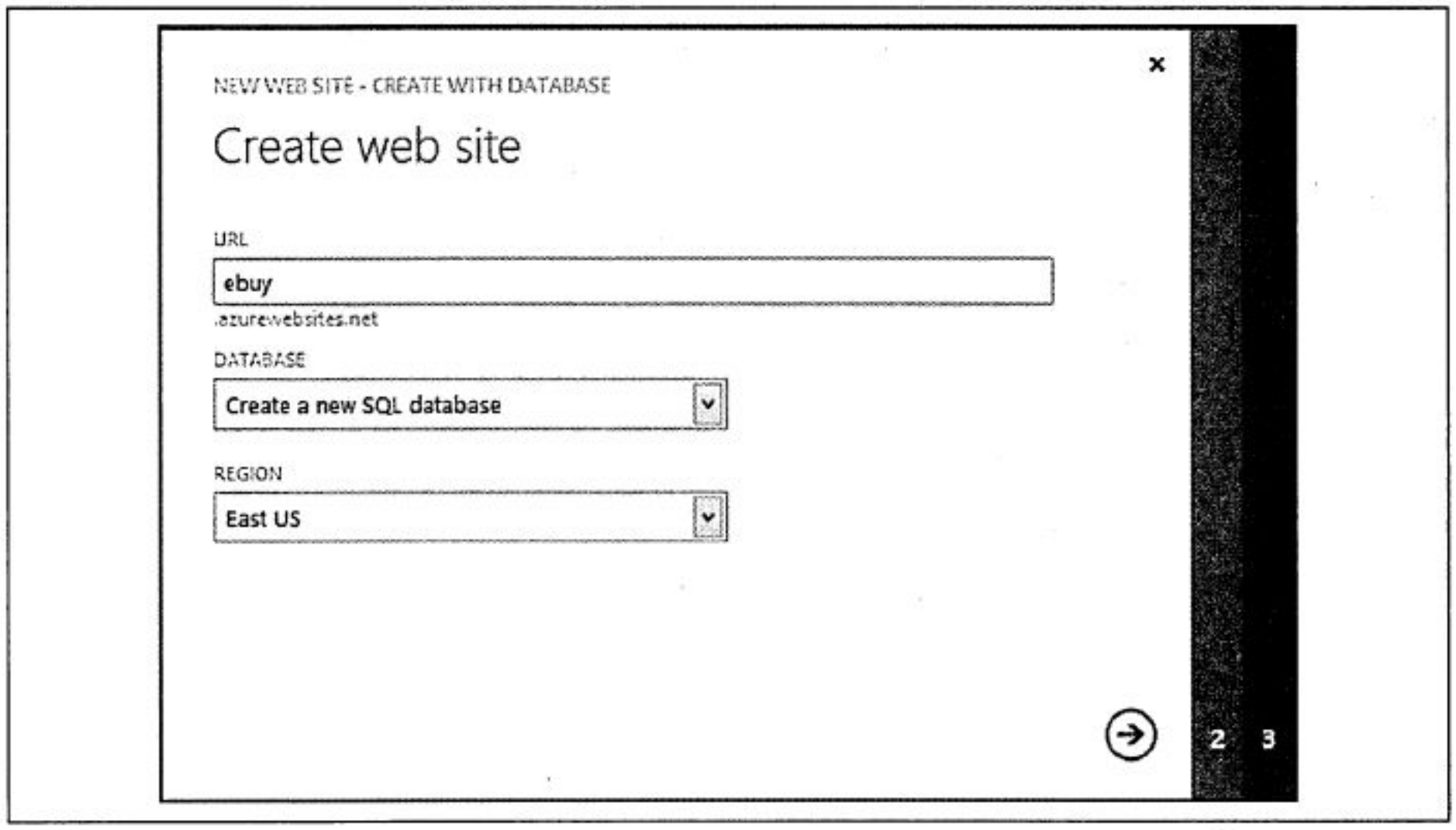


图19-11　Windows Azure的新建网站向导

通过下面的步骤，可以配置新的数据库。默认设置对于大多数小网站来说是没有问题的。在完成所有用于网站和数据库的配置工作以后，点击“创建网站（Create Web Site）”选项。Windows Azure将花费一定的时间来创建新的Web应用。完成创建工作之后，就可以从开始管理界面上看到新建的网站了。

通过源代码控制软件发布Windows Azure网站

到目前为止，Windows Azure上部署网站最简单、最方便的方法就是使用Team Foundation Server (TFS)或Git等代码控制系统内置支持的发布网站功能（译注5）。

译注4：Windows Azure 提供了 3 个月的免费试用期，目前还没有针对中国大陆地区的云主机服务。

译注5：Git 是一个免费的、开源的分布式版本控制工具，版本控制工具有 VSS、Subversion、CVS、TFS 等。它是 Linux 内核开发者林纳斯·托瓦兹（Linus Torvalds）为了更好地管理 Linux 内核开发而创立的。

因为第18章中已经介绍了TFS，所以现在直接使用相关例子。

请记住，这里是使用TFS来演示发布Windows Azure网站的操作过程，而使用Git版本控制软件发布网站的操作方法与此类似。

要使用TFS源代码控制器发布网站，首先要点击Windows Azure管理面板上的“设置TFS发布（Set up TFS publishing）”超链接（见图19-12）来打开TFS源代码控制器配置向导。然后输入第18章介绍的TFS Preview版本的账号（如果没有TFS Preview预览版账号，则可以点击超链接创建一个），点击“立即授权（Authorize now）”超链接授权Windows Azure主机访问TFS Preview账号。

图19-12　Windows Azure的网站监控仪表盘

410

在Windows Azure授权设置成功以后，就要从下一个对话框的源代码控制项目里选择一个要关联发布到Windows Azure上的网站，然后点击复选框，完成关联。稍等一会，Windows Azure会把网站与指定的TFS项目关联起来。

设置完成以后，每次在TFS项目中签入代码都会触发TFS重新生成整个项目。如果每个生成都成功，那么最新的网站就会被自动部署到Windows Azure上，也可以立即打开看到所做的最新修改，不需要做其他额外的工作了。

持续部署

自动化部署的一个主要好处就是，相当简单——而且非常快——无论部署到什么环境中。一旦设置了自动化部署，下面的任务就是尽可能频繁地执行部署任务，比如签入代码。这种频繁的应用程序部署方式称为持续部署（continuous deployment）。

持续部署是向用户展示当前项目最新状态的最佳方式。可以通过这种十分透明的方式非常及时了解应用程序的最新进展情况，这样，就可以尽可能早地使用其系统功能，然后在开发阶段给出反馈意见。可以把这种方式当做人工的、用户驱动的尽早发现问题的持续集成概念（continuous integration concept）。

如果使用了前一节介绍的使用TFS部署Windows Azure网站的方法，那么应该已经使用了持续部署。但是，如果没有在发布Windows Azure网站时使用TFS，也不意味着不可以使用持续部署方法——只是说可能需要多做一些额外的工作才能完成网站部署。

为了给项目添加类似持续部署的机制，需要花点时间来考虑本章和第18章中的哪些概念适合我们的项目，然后把这些步骤通过自动化工具来自动完成网站部署工作。

如果方案正确，那么持续部署将是把网站更新及时展现给用户的最佳方式，可以尽可能早地从用户处获得应用程序的反馈信息，以辅助我们的项目开发工作。

总结
Summary

任何软件开发的最终目标就是把软件交付给用户使用。对ASP.NET MVC网站来说，它可能包括几项任务，但是，其中最重要的任务就是把网站部署到IIS中，并运行起来，无论是在本地的服务器上，还是托管在云平台上。

第六部分

附录

Appendixes

ASP.NET MVC与Web Form集成

ASP.NET MVC and Web Forms Integration

ASP.NET MVC框架并不是微软第一次进军Web开发生态系统。ASP.NET MVC的前一版本为ASP.NET Web Form（在ASP.NET MVC之前称为ASP.NET），其第一个版本于2002年年初发布。ASP.NET Web Form属于.NET框架的一部分。在接下来的十余年，ASP.NET Web Form框架稳步发展，目前互联网上越来越多功能强大的网站均采用了ASP.NET框架。同样，大量开发人员在ASP.NET Web Form周围建立了一套强大的知识体系，用来创建、维护网站。几年后，新的Web开发框架ASP.NET MVC发布（译注1）。

现有的网站和技术不可能因为某一种新技术的出现而立即消失，或者完全抛弃。恰恰相反，事实上——这些网站都代表着重大的投资，能提供真实的、持续的商业价值。本附录会介绍各种相关的概念和策略，以将ASP.NET MVC框架引入现有的ASP.NET Web Form应用程序中。还会介绍一些误区，以帮助我们尽量避免这些误区，这样应用程序转换就会平滑、自然得多。

下面将介绍实现ASP.NET MVC和ASP.NET Web Form应用程序集成和共生的不同技术。可以自由选择适合我们团队工作的技术实现方式。

ASP.NET MVC和ASP.NET Web Form之间进行选择

在许多方面，虽然两个框架有很多相似之处，并且基于相同的.NET平台，但是ASP.NET MVC和ASP.NET Web Form代表两个不同的、相互竞争的框架。这两个框架都可以帮助ASP.NET开发人员快速、高效地提供Web解决方案，但每个框架都有自己独特的实现方式。

ASP.NET Web Form应用程序通常不能很好地支持在第2章中介绍的面向对象的SOLID设计原则。这意味着很多喜欢SOLID设计原则的开发人员不得不使用ASP.NET Web Form来交付基于.NET的Web应用程序，尽管他们非常需要一种支持SOLID设计原则的框架来满足其需要。当这些开发人员看到ASP.NET MVC时，就会立刻被这个全新的Web开发框架所吸引，ASP.NET MVC框架完美地支持了面向对象的SOLID设计原则，而且支持MVC架构，提供了完全不同的开发和测试体验。

如果有团队成员不喜欢，或者不适合ASP.NET MVC框架开发技术，或者本书所介绍的新框架的概念和技术没有让他们心动、感到兴奋，那么ASP.NET MVC框架真的不适合你们的项目。如果出现这种情况，则继续使用ASP.NET Web Form绝对没有错，不需要切换到ASP.NET MVC

译注1：2007年12月10号，发布ASP.NET MVC CTP版本；2009年4月13号，发布ASP.NET MVC 1.0。

框架。ASP.NET Web Form肯定不会消亡，事实上，它还在不停地完善、升级中，每个新版本的.NET框架发布之后，ASP.NET Web Form都会扩展许多更好的功能。

在决定要迁移到新的框架之前，请确保你的团队成员了解并认同ASP.NET MVC的基本概念——第2章中介绍的，如SOLID设计原则。

如果强行从ASP.NET Web Form向ASP.NET MVC转换应用程序，仅仅是为了使用“最新的、最好的框架”，则可能会给整个团队带来灾难性的后果。因为两个框架密切相关，并共享同一个基础平台，所以可以非常容易通过将“Web Form原则”应用于ASP.NET MVC框架来编写一个“ASP.NET Web Form应用程序”。这正是为什么没有ASP.NET Web Form开发经验的工程师更有优势：白纸一张更容易接受新事物。同样，没有ASP.NET Web Form开发经验，也不会影响他们掌握新的MVC概念。

记住，如果团队成员都是经验丰富的ASP.NET Web Form开发人员，就要持续评估代码，以确保大家编写的代码符合MVC模式，而不要回退到Web Form的编程方式上。

转换ASP.NET Web Form网站到ASP.NET MVC

对全新的“绿地”应用程序（译注2），一般来说，建议选择一个框架来开发就可。但是，如果需要将现有的ASP.NET Web Form应用程序移植到ASP.NET MVC框架中，就无法限制在一个Web框架上。上面介绍的两个框架彼此竞争，也是将两者集成到一个应用程序中的原因，即它们都建立在相同的ASP.NET平台上。

先思考ASP.NET Web Form应用程序中是如何处理请求的：IIS接收一个新的请求，在该站点的文件夹结构中查找它映射到的物理文件（.aspx页面），并执行ASP.NET Web Form的HTTP处理程序来执行该页面。然后思考ASP.NET MVC处理请求的过程：IIS 接收一个新请求，但找不到相应的物理文件，所以它会查询路由表（route table），找到请求对应的MVC HTTP 处理程序，再执行该处理程序。最后思考这个问题：路由表是ASP.NET的核心功能，它可以在ASP.NET MVC应用程序里运行，就像在ASP.NET Web Form应用程序里一样。

现在回头看看，发现创建的既不是基于控制器和视图的ASP.NET MVC应用程序，也不是基于ASPX页面的ASP.NET Web Form应用程序，而是基于`HttpModules`和`HttpHandlers`的ASP.NET应用程序。很容易理解：ASP.NET MVC的概念和ASP.NET Web Form概念其实共存于同一个应用程序中。从这些角度考虑应用程序时，就会发现让两个框架一起工作，只是简单的文件替换和配置问题。

下面介绍几种技术。使用这些技术从ASP.NET Web Form过渡到ASP.NET MVC框架可以减少很多麻烦，可以最大限度地利用现有的ASP.NET Web Form应用程序上的资源。

但是，所有这些技术依赖于一个基本概念：IIS要确定进入的请求是来自ASP.NET MVC还是来自ASP.NET Web Form应用程序。一旦确定请求的来源，IIS就可以将请求发送到相应的处理

译注2：绿地应用程序（“greenfield” applications），词语源于建筑行业，是指空地上崭新的建筑，不受各种约束、限制。同样，“绿地”应用程序是指崭新的应用程序。

程序，该应用程序的行为就会和预期的一样，正常执行了。

添加ASP.NET MVC到现有的ASP.NET Web Form应用程序

如果想规避风险、想尽可能少地更改应用程序，则可以考虑使用ASP.NET MVC功能增强现有的ASP.NET Web Form应用程序。换句话说，不让ASP.NET MVC“接管”应用程序，只是让它来处理某些非常特定的请求。使用此方法时，我们要通过注册特定路由规则来定义哪些请求由ASP.NET MVC处理。

为了让现有的应用程序把请求路由到ASP.NET MVC，需要完成以下几点配置。

1. 将System.Web.Mvc和System.Web.Razor程序集添加到应用程序的程序集引用里。
2. 将以下项添加到web.config配置文件的根目录`system.web`→`compilation`→`assemblies`集合中：

   ```
   System.Web.Mvc,Version=4.0.0.0,Culture=neutral,PublicKeyToken=31BF3856AD364E35
   System.Web.WebPages,Version=2.0.0.0,Culture=neutral,PublicKeyToken=31BF3856AD364E35
   ```

3. 在应用程序的根目录中创建一个/Views文件夹。此文件夹的位置在应用程序中非常重要，因为ViewFactory会像ASP.NET MVC应用程序一样来查找相同的物理文件路径。
   - 与标准的ASP.NET MVC应用程序一样，创建一个/Views/Shared文件夹，保存全部共享的视图文件。
   - 从一个现有的ASP.NET MVC网站复制/Views/web.config文件，或者使用例A-1所示的文件。此配置文件很重要，因为它注册了Razor文件类型处理程序，并告诉IIS，此文件夹下的文件内部应用程序仅供内部应用程序使用和不向外开放（如果有外部请求这些资源，IIS应返回“404未找到”提示信息）。
4. 可选。创建一个用于存放ASP.NET MVC控制器的Controllers文件夹。它不同于/Views文件夹，与ASP.NET MVC控制器的位置并不冲突，所以可以随意放置这个文件夹。从技术上讲，甚至可以把Controllers文件夹放在一个完全不同的项目中。
5. 在定义`HttpApplication`的类中添加路由配置（一般在Global.asax.cs文件中），就像一个标准的ASP.NET MVC应用程序。从技术角度来看，使用了与ASP.NET MVC的相同的API来注册路由。在这个场景中的根本区别是，相同的URL对应两个处理框架。创建路由时需要注意这个问题。

418

例A-1展示了上面提到的文件/Views/web.config。

例A-1　/Views/web.config配置文件

```
<?xml version="1.0"?>

<configuration>
  <configSections>
    <sectionGroup name="system.web.webPages.razor"
        type="System.Web.WebPages.Razor.Configuration.RazorWebSectionGroup,System.Web.↵
```

```
      WebPages.Razor,Version=2.0.0.0,Culture=neutral,PublicKeyToken=31BF3856AD364E35">
    <section name="host"
      type="System.Web.WebPages.Razor.Configuration.HostSection,System.Web.WebPages.Razor,↵
      Version=2.0.0.0,Culture=neutral,PublicKeyToken=31BF3856AD364E35"
      requirePermission="false" />
    <section name="pages"
      type="System.Web.WebPages.Razor.Configuration.RazorPagesSection,System.Web.WebPages.↵
      Razor,Version=2.0.0.0,Culture=neutral,PublicKeyToken=31BF3856AD364E35"
      requirePermission="false" />
  </sectionGroup>
</configSections>

<system.web.webPages.razor>
  <host factoryType="System.Web.Mvc.MvcWebRazorHostFactory,System.Web.Mvc,Version=4.0.0.0,↵
  Culture=neutral,PublicKeyToken=31BF3856AD364E35" />
  <pages pageBaseType="System.Web.Mvc.WebViewPage">
    <namespaces>
      <add namespace="System.Web.Mvc" />
      <add namespace="System.Web.Mvc.Ajax" />
      <add namespace="System.Web.Mvc.Html" />
      <add namespace="System.Web.Routing" />
    </namespaces>
  </pages>
</system.web.webPages.razor>

<appSettings>
  <add key="webpages:Enabled" value="false" />
</appSettings>

<system.web>
  <httpHandlers>
    <add path="*" verb="*" type="System.Web.HttpNotFoundHandler"/>
  </httpHandlers>

  <pages
      validateRequest="false"
      pageParserFilterType="System.Web.Mvc.ViewTypeParserFilter, System.Web.Mvc,↵
      Version=4.0.0.0, Culture=neutral, PublicKeyToken=31BF3856AD364E35"
      pageBaseType="System.Web.Mvc.ViewPage, System.Web.Mvc, Version=4.0.0.0,↵
      Culture=neutral, PublicKeyToken=31BF3856AD364E35"
      userControlBaseType="System.Web.Mvc.ViewUserControl, System.Web.Mvc,↵
      Version=4.0.0.0, Culture=neutral, PublicKeyToken=31BF3856AD364E35">
    <controls>
      <add assembly="System.Web.Mvc, Version=4.0.0.0, Culture=neutral,↵
      PublicKeyToken=31BF3856AD364E35" namespace="System.Web.Mvc" tagPrefix="mvc" />
    </controls>
  </pages>
</system.web>

<system.webServer>
  <validation validateIntegratedModeConfiguration="false" />

  <handlers>
    <remove name="BlockViewHandler"/>
    <add name="BlockViewHandler" path="*" verb="*"
        preCondition="integratedMode" type="System.Web.HttpNotFoundHandler" />
```

```
    </handlers>
  </system.webServer>
</configuration>
```

请记住，物理文件（例如，ASP.NET Web Form.aspx页面文件）总是优先于任何自定义路由，这意味着应用程序的ASP.NET Web Form部分在ASP.NET MVC控制器逻辑之前执行。若不考虑例外情况，则可以随时按照本书的其余技术创建ASP.NET Web Form/MVC混合应用程序，就像“纯正”的ASP.NET MVC应用程序一样。

复制Web Form功能到ASP.NET MVC应用程序

正如上面所述，让ASP.NET MVC和ASP.NET Web Form在同一应用程序中共生的主要工作就是配置ASP.NET MVC。因此，要根据当前Web Form应用程序创建一个新的ASP.NET MVC应用程序，并把现有的ASP.NET Web Form页面指向新的ASP.NET MVC应用程序。

虽然这个新的应用程序使用ASP.NET MVC配置，但是，ASP.NET Web Form的物理.aspx页面仍然优先于ASP.NET MVC路由逻辑执行。无论怎么做，这两种方法实现的结果相同：ASP.NET Web Form和ASP.NET MVC的功能可以共存于同一站点中。

420

集成Web Form和ASP.NET MVC功能

前面介绍了允许共存于同一应用程序中的ASP.NET MVC框架和Web Form框架的技术。由于这两个框架建立在ASP.NET平台之上，它们都共享了核心功能，这样，两个框架不仅仅是简单地“共存”，而是可以“共生”，彼此共享数据和功能，换言之，可以彼此集成。

用户管理

也许，最重要的，或至少最常使用的共享功能是基于ASP.NET的窗体身份验证、Windows身份验证、角色、成员资格和配置文件程序。这些功能不仅可以与ASP.NET MVC应用程序完美兼容，而且能够保留使用这些验证机制的Web Form代码（如登录页、用户配置文件或管理员页面等）。

例如，当ASP.NET Web Form页面使用窗体身份验证提供程序来验证用户时，窗体身份验证提供程序会为此用户生成令牌（token），而且ASP.NET API核心模块会使用这个令牌来验证用户的每次请求。这意味着用户可以通过Web Form验证自己的身份，然后被重定向到ASP.NET MVC控制器，但是，用户状态仍然显示已通过了身份验证。

缓存管理

另一个ASP.NET Web Form和ASP.NET框架经常共用的机制就是ASP.NET的缓存功能。自.NET Framework版本1.1开始，ASP.NET Web Form开发人员一直致力于在应用程序范围内使用`System.Web.Caching.Cache`类（通过`HttpContext.Cache`属性访问）和`userscopedSystem.Web.HttpSessionState+`类（可通过`+HttpContext.Session`属性访问）来管理缓存数据，而且没有任何理由不继续支持这种缓存机制。

就像前面提到的用户管理提供者，这两个类是ASP.NET API的核心部分，ASP.NET Web Form和ASP.NET MVC开发人员都可以使用。当ASP.NET Web Form和ASP.NET MVC共存于同一网站时，它们也可以共享应用程序进程，这意味着通过缓存和`HttpSessionStateApi`写入的数据，可以由一个框架在后续请求期间共享给其他的框架。

421

更多

还有更多的API可以同时在ASP.NET MVC和ASP.NET Web Form应用程序中使用。若要找到这些API，则可以查看API文档和现有的代码——任意一个不以`System.Web.UI`开头的命名空间都可能同时支持两个框架。

虽然ASP.NET Web Form和ASP.NET MVC可以访问ASP.NET框架的许多部分，但是视图状态在ASP.NET Web Form中支持而在ASP.NET MVC中无法支持。大多数情况下，视图状态用于ASP.NET Web Form当前页面进行通信，所以在ASP.NET Web Form页面和ASP.NET MVC控制器之间遇到跨请求视图状态（`ViewState`）问题的几率很低。

当转换ASP.NET Web Form应用程序到ASP.NET MVC时，应注意ASP.NET Web Form应用程序中任何使用了视图状态的代码，因为视图状态数据不会存在于ASP.NET MVC请求过程中，因此，使用它的代码很可能会出错。

总结

本附录主要介绍了如何将“新的”ASP.NET MVC框架融入现有的“遗留”ASP.NET Web Form应用程序中。事实证明，两者共用的ASP.NET API提供了从现有ASP.NET Web Form代码到ASP.NET MVC的强大升级路径。当无法升级或者替换现有代码时，底层的ASP.NET平台同样支持ASP.NET Web Form和ASP.NET MVC应用良好地共生在一起，而不需要强制把两者集成到一起。

作为平台使用NuGet

Leveraging NuGet as a Platform

第1章介绍了NuGet包管理工具（NuGet package management tool），这款工具可以帮助我们安装、配置并维护应用程序需要的各种依赖项。在本书的其他章节里，也展示了一些使用微软和社区开发的包的例子。我们可以自由使用包，而不只是使用别人发布的包。

附录B会详细介绍NuGet包管理器，包括什么是NuGet包，以及如何创建、使用NuGet包。一旦掌握了这些知识，就会发现NuGet包管理器的提示和开发技巧可以为我们带来更好的开发体验。

本附录的目标不是介绍NuGet文档中的所有内容。与之相反，本附录会简要介绍如何使用NuGet工具，然后展示如何把NuGet当成平台使用。

安装NuGet命令行工具

虽然ASP.NET MVC安装包安装了NuGet包管理器（NuGet package manager）用来调用项目中的NuGet包，但是，为了创建、发布自己的包，就需要从NuGet CodePlex网站下载、安装NuGet命令行工具（NuGet command-line tool），地址是*http://nuget.codeplex.com/*。

在NuGet CodePlex网站的下载页面找到“NuGet Command Line Bootstrapper（NuGet命令行引导程序）”（地址是*http://nuget.codeplex.com/releases/view/58939*），下载并安装这个程序。最初的下载文件只是个引导程序——第一次运行时，它会自动下载最新的NuGet命令行工具，并安装最新的版本。

下载并执行完引导程序之后，会更新最新版本的NuGet命令行工具。在开始程序里找到Visual Studio命令提示符（Visual Studio command prompt）或.NET Framework目录就可以运行起来。

现在可以创建NuGet包了。

创建NuGet包

创建NuGet包最简单的方式是根据现有的Visual Studio项目来执行`nuget`包命令：

```
nuget pack MyApplication.csproj
```

这个命令会使用从AssemblyInfo.cs项目文件中查询的程序集版本、项目名称及其他元数据来创建NuGet包。

NuSpec文件

NuSpec文件是一个用来指定包内容、元数据（如包ID、版本、名称、依赖项等）的XML配置文件。每个NuGet包都需要这个文件，因为这个文件包含了NuGet需要确定的关键信息，如需要下载哪些包及这些包是什么版本，以便满足特定的依赖项。

即使是从Visual Studio项目文件生成NuGet包也如此，正如前一个例子。尽管我们从来看不到，但是NuGet实际上临时生成了一个NuSpec文件，用来生成最终的NuGet包。

使用这种方式最大的问题就是我们放弃了很多对生成NuGet包的控制权力。而且，有时候可能对NuGet要使用哪个程序集有特别的需求。

因此，最好在生成NuGet包时自定义NuSpec文件，而不是让NuGet自动生成。下面将详细介绍如何创建并自定义NuSpec文件。

使用NuGet命令行工具

第一个创建NuSpec文件的方法是前面使用过的命令的变种：

nuget spec。

除了`nuget spec`命令会保存NuSpec文件在磁盘上，`nuget spec`命令与`nuget pack`命令类似。因此，可以在最终的包生成之前修改NuSpec文件。

例如，可以在Visual Studio命令窗口中执行以下命令为MyApplication.csproj项目文件生成NuSpec文件：

```
nuget spec MyApplication.csproj
```


或者，使用之前生成的程序集来执行同样的命令：

```
nuget spec -a MyApplication.dll
```

这些命令都会生成名为MyApplication.nuspec的文件，内容如下：

```
<?xml version="1.0"?>
<package>
  <metadata>
    <id>$id$</id>
    <version>$version$</version>
    <title>$title$</title>
    <authors>$author$</authors>
    <owners>$author$</owners>
    <licenseUrl>http://LICENSE_URL_HERE_OR_DELETE_THIS_LINE</licenseUrl>
    <projectUrl>http://PROJECT_URL_HERE_OR_DELETE_THIS_LINE</projectUrl>
    <iconUrl>http://ICON_URL_HERE_OR_DELETE_THIS_LINE</iconUrl>
    <requireLicenseAcceptance>false</requireLicenseAcceptance>
    <description>$description$</description>
    <releaseNotes>Summary of changes made in this release of the package.</releaseNotes>
    <copyright>Copyright 2012</copyright>
    <tags>Tag1 Tag2</tags>
  </metadata>
</package>
```

在初始状态上，NuSpec文件中的很多字段都是使用`$[name]$`字符填充的，在执行`nuget pack`命令时会使用NuGet包来替换实际的值。

显然，这个模板没有任何特别的信息。它支持定义了与项目相关的可以自定义的信息。因此，可以在XML编辑器（如Visual Studio内置的XML编辑器）里打开NuSpec文件，手动修改文件来配置NuGet包的内容。

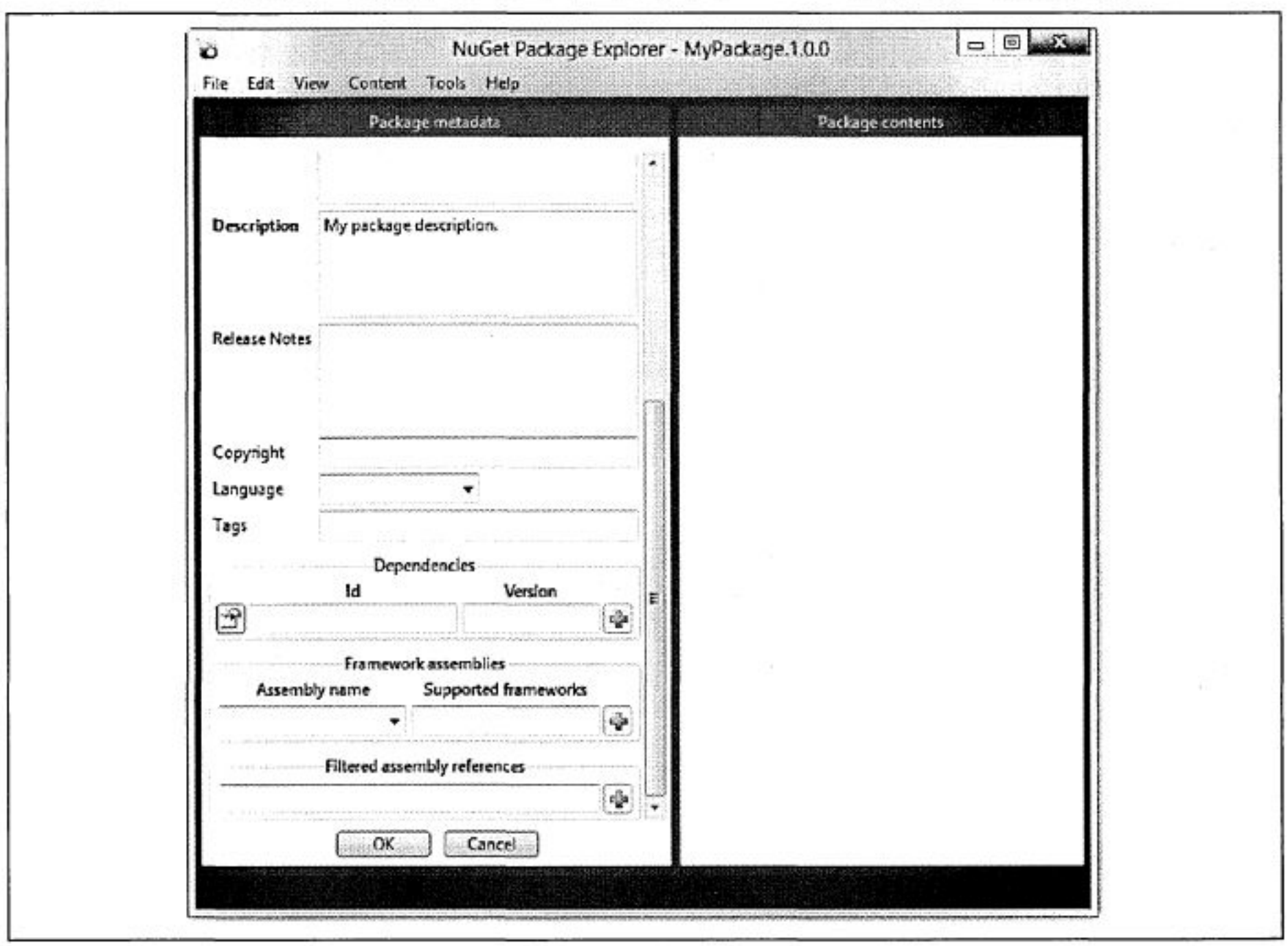
使用NuGet包浏览器

采用另外一种手工编辑NuSpec XML文件的方法可以在NuGet网站*http://nuget.codeplex.com/releases*下载NuGet包浏览器。

与其他包管理功能一样，NuGet包浏览器可提供完美的UI界面来帮助我们方便地生成NuSpec文件。

NuGet包浏览器简化了NuSpec文件创建工作：

1. 从程序主页选择创建新包选项（Create New Package (Ctrl-N)）。
2. 选择编辑→编辑包元数据（Edit→Edit Package Metadata...）菜单选项来创建可以编辑的新项目。
3. 在NuGet包浏览器编辑状态（见图B-1）下，使用GUI来指定包的各种信息。



图B-1 使用NuGet包浏览器编辑包

4. 一旦完成自定义包的设置工作，就可以使用文件→保存菜单选项（File→Save，或者Ctrl-S快捷键）来生成并保存NuGet包到磁盘上，或者选择文件→另存为元数据（File→Save Metadata As...）来保存NuSpec文件到磁盘上。

从NuSpec文件生成NuGet包

一旦定义好NuSpec文件的内容，就可以使用`nuget pack`命令来生成NuGet包了。

例如，利用前面例子中的MyApplication.nuspec文件生成NuGet包，只需要执行下面的命令：

```
nuget pack MyApplication.nuspec
```

427

如果没问题，这个命令就会生成一个名为MyApplication.1.0.0.nupkg的NuGet包，包含所有NuSpec里定义的内容和程序集。

生成NuGet包，就将它部署到NuGet储存库中以便将来在应用程序中使用。

指定占位符的值

当NuSpec文件中包含默认文件里生成的占位符时，`nuget pack`命令可能会不知道如何处理它们。此时，可以通过为`-Properties`设置分号分割的键值对来指定占位符的值。

例如，下面的命令会使用“My custom package description”字符串来替换所有的`$description$`占位符的内容：

```
nuget pack MyApplication.nuspec -Properties description="My custom package description"
```

设置版本

`nuget pack`命令暴露了`-Version`开关用于指定生成包的版本。`-Version`开关可以应用到任何NuSpec文件中，无论是否指定Version的值。

例如，无论NuSpec文件中是否指定了版本号，下面的命令将会生成版本为1.7.0的“MyApplication”包。

```
nuget pack MyApplication.nuspec -Version 1.7.0
```

NuGet包剖析

既然已经知道如何创建NuGet包，那么现在再来详细分析NuGet包到底是什么。

其实，NuGet包只是包含自定义元数据（.nuspec文件）及其程序集（也称为库）、内容及工具的zip压缩文件。

例如，如果使用工具打开NuGet包文件，就会看到例B-1中所示的文件结构。

例B-1　NuGet文件结构

```
\Content
        \App_Start
                ConfigureMyApplication.cs.pp
        web.config.transform
        [Other content files and folders]
\libs
        \net40
```

428

```
                MyApplication.dll
        \sl4
                MyApplication.dll
         [Folders for other supported frameworks]
\tools
        init.ps1
        install.ps1
        uninstall.ps1
MyApplication.nuspec
```

内容

Content文件夹表示目标程序的根文件夹。任何文件夹中的内容，包括图片、文本文件、类模板或子文件夹等都会直接拷贝到目标程序中。

除了正常的文件拷贝外，Content文件夹还可能包含配置文件和源代码转换模板。这样便于修改目标项目的某个特定部分。

例如，例B-1展示了web.config.transform转换配置文件。这个文件大体上包含下面的内容：

```
<configuration>
    <system.webServer>
        <handlers>
            <add name="MyHandler" path="MyHandler.axd" verb="GET,POST"
                 type="MyApplication.MyHandler, MyApplication" preCondition=↵
                "integratedMode" />
        </handlers>
    </system.webServer>
</configuration>
```

当NuGet添加包含web.config.transform转换文件的包到项目时，NuGet会更新项目的web.config配置文件，而且会添加“MyHandler” HTTP handler（HTTP处理程序）到配置文件中。

例B-1也包含了App_Start\ConfigureMyApplication.cs.pp代码转换模板，它的内容大体如下：

```
[assembly: WebActivator.PreApplicationStartMethod(
                        typeof($rootnamespace$.MyHandlerInitializer), "Initialize")]

namespace $rootnamespace$
{
        public class MyHandlerInitializer
        {
                public static void Initialize()
                {
                        // 运行时配置 MyHandler
        }
        }
}
```

当NuGet安装含有web.config.transform转换文件的包时，它就会拷贝ConfigureMyApplication.cs.pp文件到项目的App_Start文件夹，运行转换后删除“pp”扩展名，创建项目使用的完整功能的类。

程序集

Content文件夹之后是libs文件夹。这个文件夹十分简单：存放的程序集都会添加到项目的引用（references）集合中。

程序集可以放在根文件夹中，或者更好的方式——放在框架指定的文件夹，如net40中，以指出当前程序集使用的框架和版本信息。通过把程序集放到不同的文件夹中，可以让我们更高效地同时满足多个不同版本的框架包。

NuGet可以识别三种不同的框架，详细信息如表B-1所示。

表B-1　NuGet能够识别的框架

框架	缩写
.NET框架	net
Silverlight	sl
.NET Micro框架	netmf

例B-1通过包含两个版本的MyApplication.dll程序集展示了其功能：一个版本是.NET Framework 4.0(net40)，另一个版本是Silverlight 4.0(sl4)。

工具

tools文件夹包含脚本、可执行文件或者其他开发人员可能要使用的内容，但是不打算包含到项目里作为内容或引用程序集。

tools文件夹可以包含一个或多个特殊的PowerShell脚本，NuGet会在处理每个包时执行这些脚本。

init.ps1
当在解决方案中第一次安装包时运行。

install.ps1
每次安装包时都会运行一次。

uninstall.ps1
每次卸载包时都会运行一次。

NuGet会在Visual Studio环境中执行这些脚本，并且可以访问Visual Studio的 DTE API。这样就可以在包初始化、安装或卸载时使用这些脚本来查询、操作Visual Studio中的内容。

除了执行上面提到的特殊脚本之外，无论什么时候在解决方案中安装包，NuGet都会在包管理控制台（package management console）路径添加任意的tools文件夹。这样就可以很方便地分发脚本和支持动态开发的可执行文件。但是，这些文件不会和最终应用程序一起部署。

例如，MvcScaffolding包含一些可以帮助开发人员生成ASP.NET MVC应用程序中模型、视图及控制器的PowerShell脚本。这些PowerShell脚本可以大量节约时间，并提升开发效率。但是，它们只是在开发过程中提供帮助，并不会和最终产品一起发布。

NuGet包的类型

既然已经知道了NuGet包包含的内容，现在来看看如何使用NuGet包。

总体来看，NuGet包可以分成几类：程序集包（assembly package）、工具包（tool package），以及元数据包（meta package）。虽然它们使用相同的规范创建，并且通过NuGet管理，但是每个类别的包的用途不一样。

程序集包

程序集包的主要目的是为项目添加程序集，以及这些程序集需要的辅助内容或配置信息。程序集包是最常见的类型，因此这也是NuGet存在的主要原因。

工具包

工具包引入在开发过程中要使用的开发工具。这里说的工具，是指帮助提升开发效率或测试效率，但是不会和最终应用程序一起发布的程序。工具可以是PowerShell脚本，也可以是任何强大的应用程序。

元数据包

元数据包是指引用了其他包的包。它的主要作用是通过安装一个包就可以自动下载所有的其他依赖项来加速项目运行。

例如，我们创建一个虚拟的包：EF Code First + ELMAH + Glimpse +Ninject，这个包包含所有本书例子程序需要的包。而本书中所有的例子，我们需要做的就是打开文件→新建应用程序→ASP.NET MVC4 Web应用程序（File→New Application...→ASP.NET MVC 4 Web Application），然后使用NuGet来安装元数据包，这样就会自动添加所有项目需要的应用。

共享NuGet包

一旦创建完NuGet包，我们需要做的就是把包添加到包存储库中，这样可以方便分享给其他开发人员，其他开发人员就可以在自己的程序中使用相同的包了。

要分享创建的包，有两种选择：可以把包发布到公开的NuGet.org包存储库，或者自己托管自己的包存储库。

发布到公共NuGet.org包存储库

在安装过程中，NuGet安装器会预先配置一个存储库——NuGet.org上托管的NuGet包存储库（NuGet package repository）。因为是预先配置的，所以NuGet.org包存储库是与其他开发人员分享包的最方便方式。如果创建的包包含了你所需要的与世界上其他开发人员分享的功能，那么分享NuGet.org包存储库无疑是最佳的选择。

在上传包到公共的NuGet.org包存储库之前，需要在NuGet.org网站上创建自己的账户，网址是*http://nuget.org*，选择页面上的注册（Register）按钮进行注册即可。

使用NuGet.org包上传向导

注册账号之后，就可以在NuGet.org网站发布包了。最简单的方式是使用在线上传向导，它可以引导我们完成所有上传需要的步骤。

可以在NuGet.org网站上选择上传包（Upload Package）菜单，开始上传包。

使用NuGet命令行工具

另外一种方法就是使用NuGet命令行工具的发布功能来部署包。这种方法对使用NuGet.org网站比较方便，因为如果使用NuGet命令行制作包，那么可以顺便再执行一次命令来发布这个包到网站上。

这个命令十分简单：

```
nuget push [package name]。
```

例如，发布之前创建的包到NuGet.org网站上，需要的命令就是：

```
nuget push MyApplication
```

如果是第一次执行这个命令，那么会遇到一个错误，提示我们没有为要发布的（公共的NuGet.org存储库）NuGet包源指定API Key。API key是由存储库创建的唯一标志和安全令牌，可以控制对存储库的访问。虽然公共NuGet.org存储库允许任何注册账户的人发布包，但是至少要提供一个有效的账户才行。幸运的是，NuGet.org会在我们创建账户的时候生成API key。为了查询这个API key，只需登录账户，并查看个人信息页面即可。这里有个API key超链接，点击一下就可以显示和拷贝API key。

一旦拷贝了API key，就可以把这个参数传递给NuGet。为此，执行的命令就是`nuget setApiKey [API key]`，示例如下：

```
nuget setApiKey ae19257f-9f0c-4dcf-b46a-60792fd5ff2d
```

API key设置完毕后，就可以执行`nuget push`命令，向公共NuGet.org存储库发布我们的项目包。

托管自己的包存储库

也可以不使用公共NuGet.org存储库来发布我们的项目包。实际上，甚至都不需要离开本地机器。

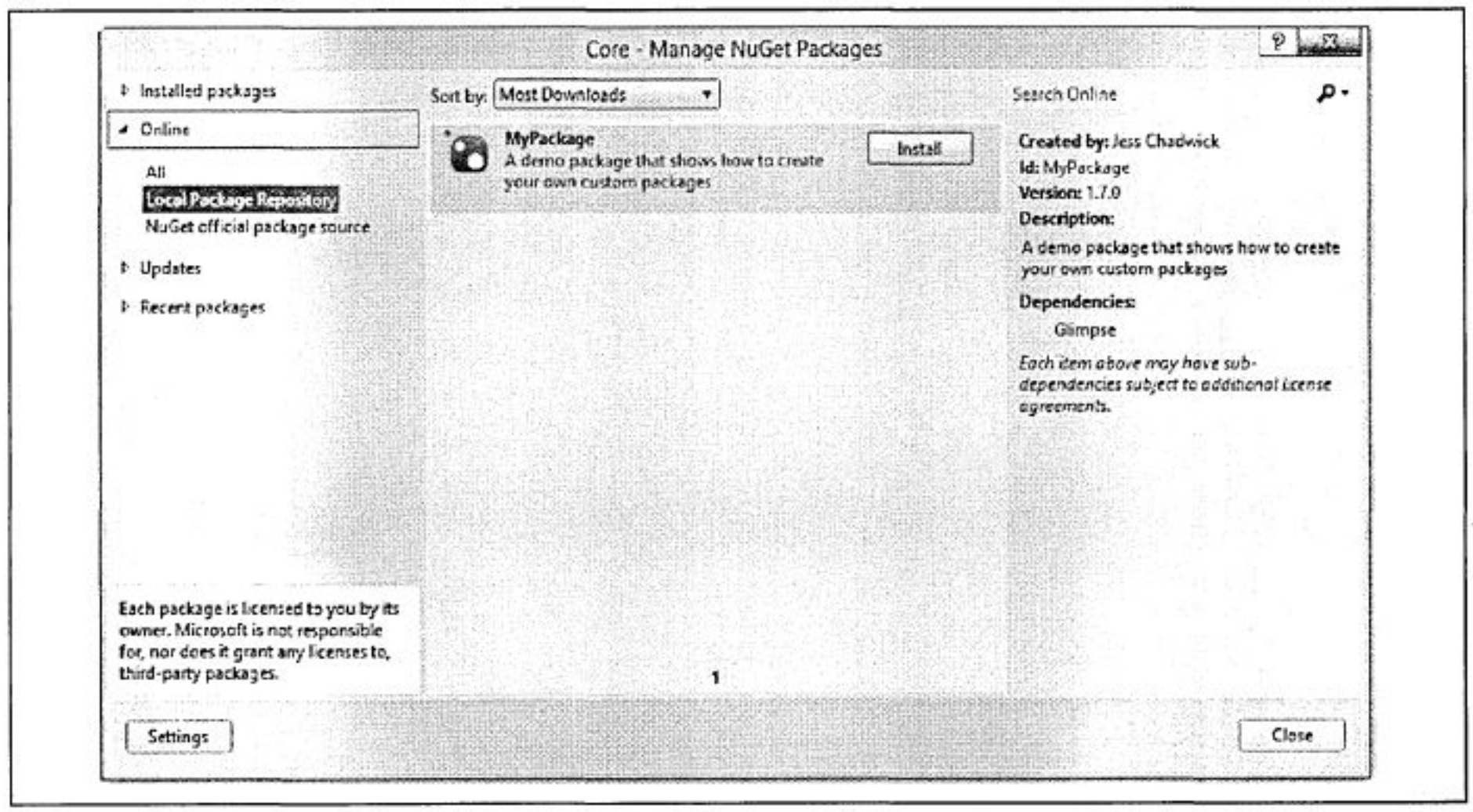
NuGet提供了两种主要的托管和使用包的方式，即设置文件系统存储库和托管我们自己的NuGet Web服务器。

使用文件系统存储库

文件系统存储库（filesystem repository）就是我们可以访问的存储在文件系统上的包集合。其设置和运行过程非常简单。

为了创建文件系统存储库，可以参考下面的步骤：

1. 在本地磁盘新建名为C:\NuGetPackages的文件夹。
2. 打开NuGet设置对话框：选择工具→库包管理器→Visual Studio中的包管理器设置（Tools→Library Package Manager→Package Manager Settings in Visual Studio）选项，然后切换到包源部分来添加新的包源（package source）地址。
3. 通过设置包源的名称和路径来设置与包源相关的信息。这个例子是C:\NuGetPackages，如图B-2所示。



图B-2 添加新包源

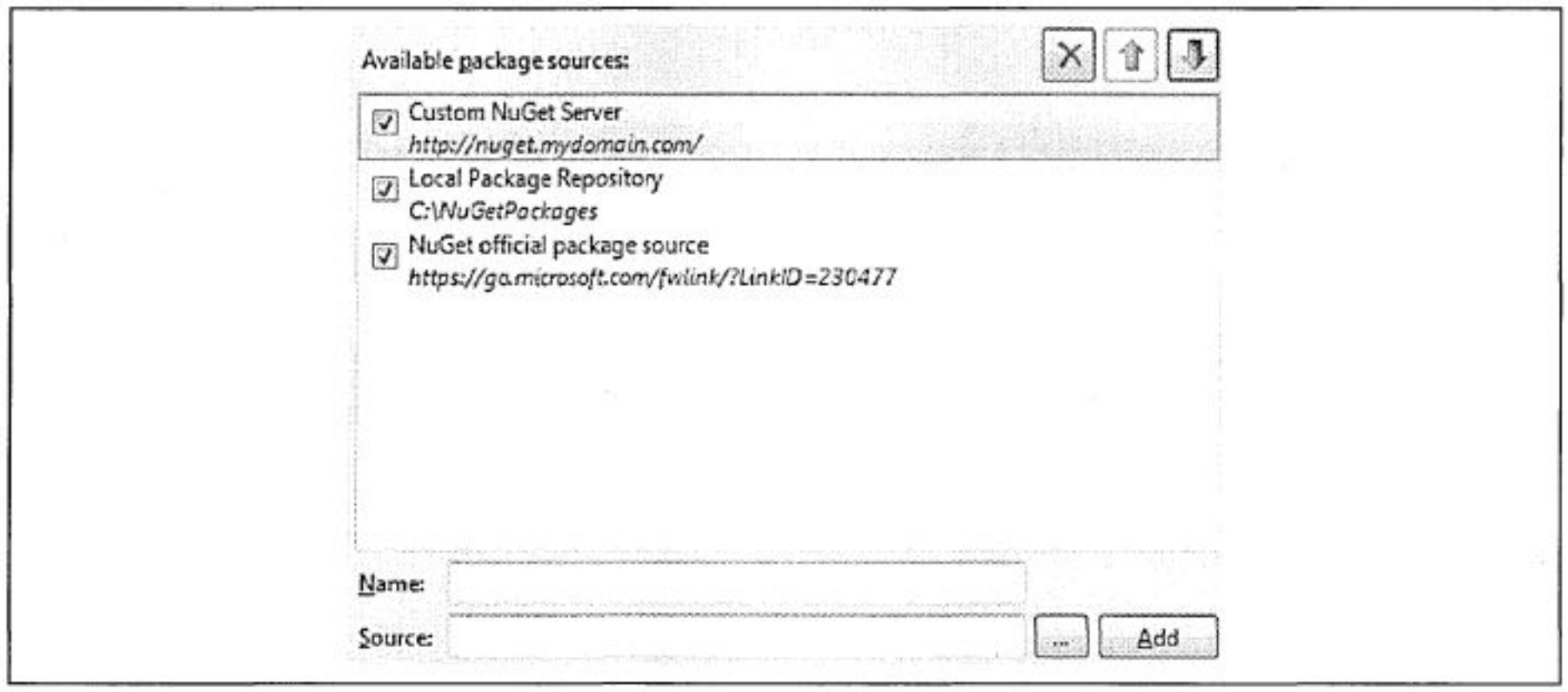
4. 点击添加按钮，将存储库添加到包源列表中。

下次在使用包管理器时，就会看到新的包源列表，而且新添加的包源也会显示在包管理器列表中，如图B-3所示。

这种在本地硬盘上设置包存储库的方式可能有点“傻”！其实文件路径不一定非要设置在本地磁盘上。存放包存储库的文件系统可以是任意能够通过Windows文件浏览器查看的文件路径，包括网络文件共享。实际上，在某个中心服务器上托管团队自定义NuGet包存储库，是最简单也是最有效的团队分享包的方式。

托管NuGet服务器存储库

NuGet服务器是一个托管了许多包含NuGet包信息的OData web services网站，这也是公共的NuGet.org存储库的访问网站。虽然自定义托管文件系统包存储库（filesystem package repository）的配置非常简单、方便，但是NuGet服务器同样也不复杂，而且提供了更强大的功能和更大的灵活性。



图B-3　新包源应该出现在包管理器的列表中

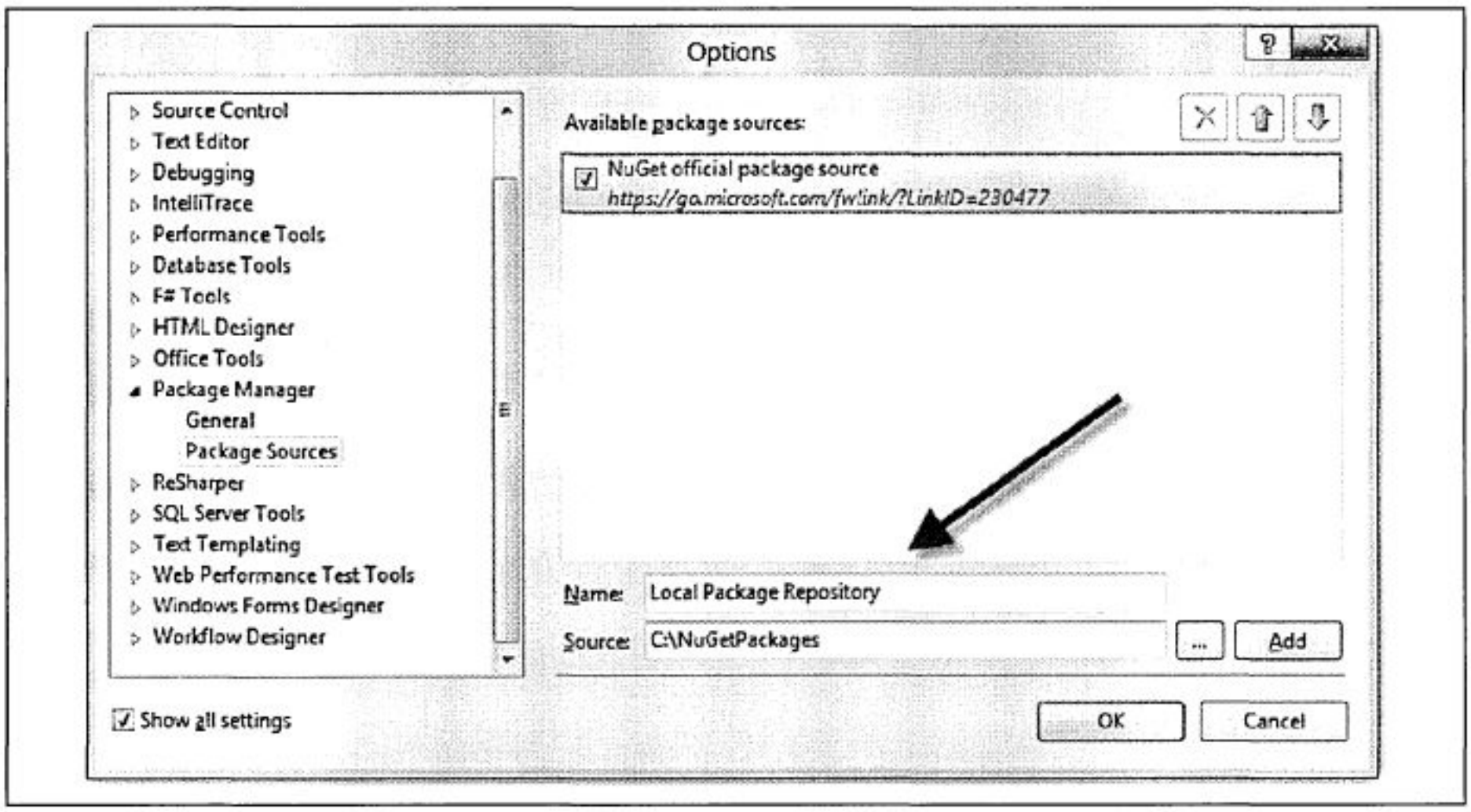
以下是托管NuGet服务器的步骤。

1. 打开Visual Studio，选择ASP.NET 空网站模板（ASP.NET Empty Web Application template）创建网站。
2. 使用NuGet包管理器查找并安装NuGet.Server NuGet包，它会自动下载配置、运行NuGet Server需要的一切资源。
3. 默认情况下，NuGet.Server包会创建Packages文件夹来托管NuGet包文件。如果想把NuGet包存放到别的地方，那么可以在项目的配置文件web.config里的`appSettings`→`packagesPath`节点下设置新建包文件夹的路径。
4. 一旦设置完成，就可以像部署其他网站一样部署这个包存储库网站了。记住，这个网站需要访问Packages目录内的文件。

部署完网站之后，如果能确保一切运行正常，就可以通过NuGet设置窗口：通过选择工具→库包管理器→Visual Studio中的包管理器设置（Tools→Library Package Manager→Package Manager Settings in Visual Studio）来添加新的包源路径了。

这个步骤和前面添加文件系统存储库包源路径的方式一样，除了地址是新的NuGet服务器的URL以外。可用的包源列表如图B-4所示。

再强调一下，当下次使用包管理器时，就会看到新的包源列表。任何添加到包服务器库的包都会显示到包管理器的列表中。



图B-4　可用的包源

提示、技巧和误区

从技术角度讲，使用NuGet可以很容易下载、安装、配置程序集依赖，甚至添加其他内容到网站中。同样，创建和发布自定义NuGet包也很简单。但是，有些情况下使用NuGet就比较复杂，特别是当处理包含多个项目、多个开发团队的大型应用程序时。

下面列举了使用NuGet要注意的一些误区，同样也包括一些帮助我们更好地在项目中使用NuGet的提示和技巧。

436

误区：NuGet不能解决“DLL hell”

“DLL地狱（DLL hell）”（译注1）也称为“依赖地狱（dependency hell）”，是指应用程序在运行时使用了程序集或其他的库而引发的问题。NuGet的主要目标之一就是确保应用程序更新到最新版本，而且不存在冲突——这非常有助于解决DLL hell问题——但是有时候可能不起作用。

译注1：DLL地狱（DLL hell）是指在Microsoft Windows系统中，因为动态链接库（DLL）的版本或兼容性问题而造成程序无法正常运行。Windows早期并没有很严谨的DLL版本管理机制，以致经常发生安装了某软件后，因为其覆盖了系统上原有的同一个DLL文档，而导致原有可运行的程序无法运行。但还原回原有的DLL文档后，新安装的软件也无法运行。若覆盖到系统所使用的重要DLL，也可能让系统死机甚至无法正常启动。此内容来自维基百科。

最常见的两种冲突之一是，当两个包期望使用不同版本的第三方组件时。例如，Ninject和Glimpse都要依赖Nlog，但是，Ninject使用的最高版本是Nlog v2.0，而Glimpse使用的是Nlog v2.5。显然，两个包不能包含到同一个项目。幸运的是，这种冲突的原因非常明显，而且NuGet会很容易发现问题（NuGet在添加包时会自动显示这个错误），所以影响会降到最低。

另外一种冲突是两个包依赖不同版本的第三方组件——像第一种冲突一样，但是它们的NuSpec元数据没有提供足够的信息让NuGet来了解两个包之间的冲突。换句话说，NuGet用来判断程序集版本的逻辑代码要根据它读取的版本信息，当包信息不够准确时，就会导致潜在的冲突问题。

在前面的例子中，Ninject需要使用NLog v2.0，而Glimpse需要使用NLog v2.5。假定Ninject只支持NLog v2.0及其早期版本，而不支持NLog v2.5，且又不知道这些信息，那么NuGet就会安装NLog v2.5，这样就带来了不兼容的问题，运行时Ninject就会出错。

即使是使用工具如NuGet，也需要继续面对这些经典的程序集版本控制冲突。只要设置了正确的版本和依赖信息，NuGet就可以十分高效地帮助我们避免潜在的依赖问题。但是，没有万无一失的东西！在混用和匹配各种程序集时，仍然需要注意。

提示：使用Install-Package–Version安装特定版本的包

当安装或升级NuGet包时也许需要分几次进行，但有时我们并不想要安装最新的版本。

例如，假设应用程序使用的是Ninject v2.1.0，希望升级到更新的版本（如v2.2.5），想使用这个版本的一些最新的特性，但是，发现最新的版本（如v2.4.0）有些Bug，可能导致程序无法运行。

这种情况下，如果使用NuGet包管理界面（Manage NuGet Packages GUI），或者直接使用`Install-Package`命令请求安装Ninject包，NuGet就会安装最新的版本v2.4.0，而且默认这是我们想要的版本。其实我们要的不是这个版本！

幸运的是，NuGet提供了`-Version`参数，允许开发人员控制NuGet下载包的版本（这个参数通过UI看不到）。为使用版本参数`-Version`，就要在包管理器控制台输入命令`Install-Package`加上`-Version`。

例如，可使用下面的命令来安装版本为2.2.5的包，避免升级到最新的2.4.0：

```
Install-Package Ninject -Version 2.2.5
```

使用这个命令会自动下载，并且安装Ninject v 2.2.5，NuGet不会获取最新版本的Ninject包。

提示：使用语义版本控制

实际上，无论是否关心程序集的版本，所有的程序集都必须有一个版本号。为了创建分享NuGet包，必须给每个包分配一个唯一的版本号，以区分其他包，并能暗示出包发布的顺序。默认情况下，Visual Studio会生成版本号，但是，当自定义NuGet包版本号时，就必须使用有意义的数字。当定义和管理这些版本号时，要认真思考，并起积极作用。

从本质上讲，产品和程序版本最好相对独立——最坏的情况就是混乱不堪，某个产品的版本“version 1.0”实际上和其他产品的“version 1.0”没有任何关系。因此，“version 1.0”是什

么意思呢？它和同一个产品的版本“0.1”、“1.5”或“2.0”有什么不同呢？

因为包版本控制是依赖管理最重要的方面，所以，NuGet使用了最流行的版本控制方式——语义版本控制（semantic versioning）。虽然语义版本控制规定了定义版本不同部分的详细规则，但是语义版本控制实际上可以归结到模式`[Major].[Minor].[Patch]`上。每部分都是非负数的整数，且从0开始递增，每次加1。只要代码库的修改足够引起版本变化，那么可以增加版本号。

什么是“代码库的修改足够引起版本变化”？通常来说，语义版本控制部分依照下列规则递增。

Major
: 任意时候引入向后不兼容的修改。

Minor
: 任意时候引入新的向后不兼容的修改。

Patch
: 任意时候引入向后兼容的修改。

可以给要分享的NuGet包定义任何我们喜欢的版本控制模式，但是NuGet会根据上述的版本规则来选取要更新的包。

提示：使用预发布的版本标记器来标记“Beta”包

当向潜在用户发布测试包的时候会发生什么事情呢？大部分情况下，当发布一个新的包到库服务器时，NuGet会为每个用户自动下载并安装这个包，这显然不是我们期望的。幸运的是，NuGet支持预发布包（prerelease package）的概念——这些预发布的包可以和正常发布的包共存于同一个库中，而且对用户是不可见的，除非用户想使用这个包。

为了标记预发布的包，需要把包的版本号转换成语义版本控制对应的号码（使用“-”分割）。例如，版本号为“1.0-beta”，表示这个包是“version 1.0”的预发布版本，可能很快就发布正式版本了。

把这个包发布到正常的包存储库服务器上后，NuGet客户端可以访问，但是团队开发人员无法执行正常的包管理操作。要想使用预发布的包，需要在NuGet查询命令里加上-Prerelease标签，或者安装命令。例如，如果MyApplication预发布包的版本号是“1.0-beta”，则直接使用`Get-Packages`命令获取这个包会出错，必须使用`-Prerelease`前缀才能找到可用的预发布包：

```
PM> Get-Packages MyApplication
```

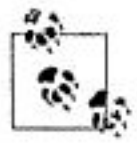
其他操作也一样，要加上`-Prerelease`标签才可以查询Beta包：

```
PM> Get-Package -ListAvailable -Filter -Prerelease MyApplication
Id                                Version              Description/Release Notes
--                                ------               -------------------------
MyApplication                        1.0-beta             My awesome package.
```

预发布的包可以使用`Install-Package`命令安装，但也要添加`-Prerelease`标签，例如，

```
PM> Install-Package -Prerelease MyApplication
```

测试完成后，准备好正式发布包了。可以去掉“-beta”前缀，包的版本为“1.0”，每个团队中的开发人员就可以看到这个包，而且每个NuGet包操作都会使用最新版本的包。



预发布版本控制作为语义版本控制模式的一部分，这些提示实际上也是对前面语义版本控制的提示。

这也是在包上使用语义版本控制的另外一个原因。

误区：避免在NuSpec文件里指定“Strict（严格的）”版本依赖

NuSpec配置文件允许包去指定其他包——版本信息——它们依赖的包的版本信息。NuSpec配置文件中的依赖部分必须是可扩展且足够详细的，这样NuGet就有足够的信息来判断哪个包是正确的安装版本，而不会导致冲突问题。

正如前文中“DLL地狱（DLL hell）”中提到的，很难避免在软件开发过程中不混用程序集的情况。DLL地狱（DLL hell）可能导致应用程序瘫痪。然而，当使用最新版本的库来分发包时，就可以在NuSpec配置文件里指定解决冲突的版本，以避免这些冲突。

例如，“DLL地狱（DLL hell）”例子中的Ninject包只与NLog v2.0兼容，而与NLog v2.5冲突。它的NuSpec文件也许包含下面的文字，指明期望的最大版本都是NLog v2.0，NuGet不应该安装NLog 2.0以上的版本，配置代码如下：

```
<dependency id="NLog" version="2.0" />
```

虽然这种明确的版本设置可以避免运行时程序集冲突，但是记住，这些配置并不会直接影响项目中的包，而且它们的限制可能导致很多冲突，让管理包的工作变得更加复杂。事实上，最糟糕的情况是当升级到相同包的最新版本时，某些特定版本的包已经解决了某些依赖问题。

必须依赖于特定版本的程序集，实际上有时候无法避免，但是有时候使用这种限制又非常犹豫，因为这种决定非常偶然，后面可能就不需要了。

如果系统从某个特定的包版本升级到更高的但是非特定的版本，则可以直接使用包管理控制，并输入`-IgnoreDependencies`和`Install-Packages`命令来重写任一NuGet的冲突。例如，

```
Install-Package -IgnoreDependencies Ninject
```

这个命令会告诉NuGet应该安装最新版本的包，并且忽略任何发现的依赖冲突。

记住，当使用-IgnoreDependencies命令时，就绕过所有的NuGet的保障措施，且必须确保所有的包依赖关系是正确的。

提示：使用自定义储存库来控制包版本

NuGet预配置了一个存储库：NuGet.org上托管的公共存储库。这对于绝大多数人员来说是个不错的选择，假设所有的开发人员都想使用这个存储库。使用公共存储库的缺点是无法完全控制NuGet.org上托管的公共存储库。

如果包可以持续稳定更新，并且版本稳定增加，确实是件好事，很多开发团队并不喜欢频繁的、缺乏控制的更新。例如，如果使用公开的NuGet存储库来作为首选的库源，而Ninject发布了一个新的无法兼容的版本，NuGet就会试图更新最新的版本，虽然我们知道这会导致兼容性问题。

虽然可以很容易忽略NuGet的推荐更新，但是这种情况也会在某些包使用最新发布的Ninject更新包时变得异常复杂。如果NuGet不更新，则可能影响某些正常的包；如果更新，则会导致兼容性错误。

一种避免强制更新的方式就是获取存储库的控制权，通过配置自定义存储库来替代默认的NuGet存储库来实现。也可以使用自己团队要使用的包自定义存储库来填充存储库。

这些包可以来自任何地方。有些可能是我们自己创建的包，但大部分可能是从公共NuGet存储库上获取的安装包。使用自定义存储库来替换默认的存储库可以集两者优势于一身。它允许我们继续使用NuGet强大的依赖管理功能，仍然保留对包安装过程的控制权。

事实上，很多开发人员对选择安装哪个NuGet包并没有兴趣，但是某些情况下——特别是当多个团队同时使用时——需建立一个稳定的、可持续的存储库来降低DLL地狱（DLL hell）的影响。

441

提示：配置持续集成生成任务来生成NuGet包

持续集成生成和NuGet是天生的一对。

虽然这个规定对不同的持续集成平台可能不同，但是持续集成生成NuGet包的过程是相同的。

1. 为系统生成的所有NuGet包创建NuSpec配置文件，而且应确保它们都签入了源代码控制器中。
2. 在每次成功生成之后使用NuGet命令行工具来执行持续集成。
3. 每次生成包都确保每个包有唯一的版本。事实上，大部分持续集成系统包含了每次生成工作使用的唯一的生成数字。
4. 把生成的包转移到中心存储库上。

   请参见“托管自己的包存储库”一节内容，了解如何配置自定义NuGet存储库。

生成NuGet包作为持续集成生成的一部分，可以给持续集成和部署工作带来很多好处，简化工作量，提升工作效率。

总结

Summary

虽然有时候管理程序集引用的工作十分困难，但是NuGet确实是个强大的依赖管理系统，它提供了强大的依赖管理功能帮助我们完成很多复杂、烦琐的工作。NuGet包不仅使用十分简单，而且便于创建和分享。如果使用得当，NuGet是开发工具箱中仅次于Visual Studio的最强大工具之一。

ASP.NET MVC开发最佳实践原则

Best Practices

本书涵盖了关于ASP.NET MVC Web开发方面的全部内容，涉及不同程度的细节问题，并且提出了很多建议。但是，有时很难判断信息的重要性，尤其是在冗长的上下文中。

本附录把本书中曾经提到的很多开发实践经验制成一个列表，以便大家在实际开发中学习、使用，检查自己在开发中是否使用了本书中介绍的这些经典的开发模式与实践经验，以便大家能够立刻判断出自己是否遵循了在书中详细描述的经典的开发模式和方法。

使用NuGet包管理器管理依赖

NuGet包管理器，又一个开发辅助利器，开发人员的一大福音，可以把开发人员解放出来。在项目开发的过程中，不需要再花费大量的时间来检查依赖项是否有新的版本问题了，NuGet包管理器可以自动解决这些问题。

如果公司中的几个团队共享了同一个类库，则必须考虑创建自定义NuGet包来共享类库，托管NuGet存储器来提供更加高效的分布式和版本控制方案。

依赖抽象

抽象（abstraction）提倡使用分离契约和实现代码的方式来使系统达到松耦合的目的。抽象具有易交互性，不仅易于维护，而且方便单元测试。

避免New关键字

当每次使用new关键字来实例化新的对象时，我们——根据定义——没有依赖抽象。虽然这种编程方式不是大问题（如实例化`new StringBuilder()`、`new List<string>()`等），但是花时间考虑一下，也许使用new关键字实例化对象的代码可以依赖注入方式实现（dependency injection）。无论何时，尽可能让另外一个组件来创建对象。

避免直接使用HttpContext（使用HttpContextBase）

ASP.NET MVC(.NET 4之后)引入了ASP.NET框架核心部分的抽象`System.Web.Abstractions`。这里使用的就是SOLID原则中的依赖抽象（depend on abstractions）。尤其是在ASP.NET开发中，使

用得最多的一个对象就是HttpContext——一般会使用HttpContextBase抽象基类替代。

避免“魔力字符串”

魔力字符串（magic string）——非常重要，但是值可以是任意类型——大多数情况下也是必需的。但是，也存在很多问题，具体如下：

- 没有任何内在的意义（例如，无法分辨一个ID如何或是否关联到另外一个ID）。
- 拼写错误或大小写错误都会导致问题。
- 解析数据类型非常困难。对重构没有反应。
- 大量重复。

以下两个例子中，第一个例子使用了“魔力字符串”来访问ViewData字典中的数据，第二个例子使用了强类型模型对象来解析字段值：

```
<p>
    <label for="FirstName">First Name:</label>
    <span id="FirstName">@ViewData["FirstName"]</span>
</p>

<p>
    <label for="FirstName">First Name:</label>
    <span id="FirstName">@Model.FirstName</span>
</p>
```

“魔力字符串”确实非常简单并易于操作，但是这种易用性也会给后期的维护工作带来潜在的问题。

优先使用模型而不是ViewData

如前一个例子所述，在ASP.NET MVC 应用程序中，是使用“魔力字符串”访问ViewData字典中的数据。但是，最好还是尽量避免使用它。强类型的模型对象可以避免这种从ViewData字典里赋值或从数据字典提取数据查询数据的操作（译注1）。

445

不要在后台编写HTML代码

在程序开发过程中要遵守SOLID中的分离关注点（separation of concerns）原则：这不是控制器的职责，不要使用后台代码来渲染HTML。当然，例外就是使用UI帮助类来生成HTML代码。这些帮助类应该当成视图的一部分，而不是后台代码类。

帮助类的唯一职责就是帮助视图渲染代码。

译注1：ViewData 字典赋值和查询数据的操作，实际耗费性能的地方在于数据的装箱和拆卸操作。

不要在视图中执行业务逻辑

与上一条对应，而且同样适用：视图文件里不应该包含业务逻辑代码。事实上，视图应该尽量不包含任何逻辑代码。视图应该关注如何显示数据，而不是关注如何操作数据。

使用帮助方法操作部分视图

用户控件（user control）、服务器控件（server control）及控件（control）的概念早已经深入人心，而且确实非常好用，方便了开发工作。这个概念通常是指把重用的逻辑代码封装起来，做成控件，以便更好地重用和维护。ASP.NET MVC框架与ASP.NET Web Form不同，并不是控件驱动式的，它引入了帮助方法的概念来替代控件机制。帮助方法可以生成整段的HTML代码（ASP.NET Web Form通常叫控件），或者定义可以通过URL访问的单个页面。例如，可以注册到很多成员资格页面(~/membership)，帮助方法代码如下：

```
@Html.ActionLink("Membership", "Index", "Membership", [...])
```

也可以通过直接调用帮助方法来实现这项功能（不是使用魔力字符串）：

```
@Html.MembershipLink()
```

ASP.NET MVC框架预定义了这个帮助方法。

使用展示模型而不是直接使用业务对象

通常来说，尽量避免因修改业务模型而直接影响视图文件。模型可以解决这个问题，因为展示层依赖于模型对象。

在视图中使用HTML帮助方法封装代码

集成代码与HTML标签都非常强大。但是，请思考以下if/else语句：

```
@if(Model.IsAnonymousUser) {
    <img src="@Url.Content("~/content/images/anonymous.jpg")" />
} else if(Model.IsAdministrator) {
    <img src="@Url.Content("~/content/images/administrator.jpg")" />
} else if(Model.Membership == Membership.Standard) {
    <img src="@Url.Content("~/content/images/member.jpg")" />
} else if(Model.Membership == Membership.Preferred) {
    <img src="@Url.Content("~/content/images/preferred_member.jpg")" />
}
```

可以使用这种方法，但是感觉代码非常臃肿、烦琐，而且混乱，难以维护。可以考虑使用下面的代码进行封装：

```
public static string UserAvatar(this HtmlHelper<User> helper)
{
    var user = helper.ViewData.Model;

    string avatarFilename = "anonymous.jpg";

    if (user.IsAnonymousUser)
```

```
        {
            avatarFilename = "anonymous.jpg";
        }
        else if (user.IsAdministrator)
        {
            avatarFilename = "administrator.jpg";
        }
        else if (user.Membership == Membership.Standard)
        {
            avatarFilename = "member.jpg";
        }
        else if (user.Membership == Membership.Preferred)
        {
            avatarFilename = "preferred_member.jpg";
        }

        var urlHelper = new UrlHelper(helper.ViewContext.RequestContext);
        var contentPath = string.Format("~/content/images/{0}", avatarFilename);
        string imageUrl = urlHelper.Content(contentPath);

        return string.Format("<img src='{0}' />", imageUrl);
    }
```

现在，可以直接在需要显示用户个人信息的地方直接调用帮助方法来显示：

```
@Html.UserAvatar()
```


这种方式不仅使代码更整洁，而且更具复用性，把逻辑代码封装在统一的方法里，随处可以调用，便于维护。例如，如果需要修改，以支持自定义头像，则只需要修改`Html.UserAvatar()`帮助方法中的代码就可以了。

显示指定视图名称

ASP.NET MVC控制器操作代码中调用的`View()`方法并没有指定视图的名称。这对简单的例子来说非常合适，但是当测试或操作之间要互相调用时，弊端就显现出来了。当不指定任何视图名称时，ASP.NET MVC框架默认使用最初调用的操作名称作为视图名称。因此，调用Index操作会导致ASP.NET MVC框架去查找名为`Index`的视图文件Index.cshtml——该文件可能不存在（但是List.cshtml文件存在）。代码如下：

```
public ActionResult Index()
{
    return List();
}

public ActionResult List()
{
    var employees = Employee.GetAll();
    return View(employees);
}
```

如果修改了`List`操作代码，则使用特定的视图名称去调用`View()`方法，也可以正常工作，代码如下：

```
public ActionResult List()
{
```

```
        var employees = Employee.GetAll();
        return View("List", employees);
    }
```

优先使用参数对象而不是参数列表

这个建议并不是只针对ASP.NET MVC框架。由于长长的参数列表通常被当成“code smell（丑陋的代码）”，因此应尽量避免使用这种方式。此外，ASP.NET MVC框架强大的模型捆绑器让下面的做法异常简单。

下面两个完全不同的代码块中，第一个使用了长参数列表：

```
public ActionResult Create(
        string firstName, string lastName, DateTime? birthday,
        string addressLine1, string addressLine2,
        string city, string region, string regionCode, string country
        [... and many, many more]
    )
{
    var employee = new Employee( [Long list of parameters...] )
    employee.Save();
    return View("Details", employee);
}
```

第二个代码块的参数为对象：

```
public ActionResult Create(Employee employee)
{
    employee.Save();
    return View("Details", employee);
}
```

显然，参数对象的方式更加简单、明了，而且它利用ASP.NET MVC框架提供强大的模型捆绑器和模型验证机制，让代码变得更安全、更易于维护。

使用操作过滤器或子操作(Html.RenderAction)封装通用的功能、逻辑、数据

无论网站多么复杂，都会包含一些页面公用的元素。网站的导航菜单——网站中每个页面要出现的模块就是一个典型的例子，每个页面都会包含公用的逻辑内容。这些元素需要的数据可能来自某些地方，如果每个控制器操作方法里都要执行一次数据库查询，那么这对维护来说是个噩梦。操作过滤器或子操作（通过`Html.RenderAction()`方法执行）提供了简单的实现方式。

下面的布局代码块（从布局页面中剪切出来的）负责渲染导航栏：

```
<ul id="global-menu">
    @foreach (var menuItem in ViewData.SingleOrDefault<NavigationMenu>()) {
        <li class="@(menuItem.IsSelected ? "selected" : null)">
            @Html.RouteLink(menuItem.DisplayName, menuItem.RouteData)
        </li>
    }
</ul>
```

NavigationMenu ViewData对象需要从别处获取数据，因为要在控制器请求之前执行这些数据。操作过滤器就是最适合处理这种问题的机制，即给ViewData对象提前赋值。下面是为导航栏查询需要数据的操作过滤器代码：

```
public class NavigationMenuPopulationFilter : ActionFilterAttribute
{
    private readonly INavigationDataSource _dataSource;

    public NavigationMenuPopulationFilter(INavigationDataSource dataSource)
    {
        _dataSource = dataSource;
    }

    public override void OnActionExecuting(ActionExecutingContext filterContext)
    {
        NavigationMenu mainMenu = _dataSource.GetNavigationMenu("main-menu");
        filterContext.Controller.ViewData["MainNavigationMenu"] = mainMenu;
    }
}
```

这个过滤器非常简单，它先从数据源中获取正确的导航数据，然后在请求操作方法之前将其添加到ViewData集合中。任何需要导航栏数据的组件都可以从ViewData中查询。

根据业务概念划分控制器封装操作

例如，创建CustomersController来封装与处理客户业务相关的逻辑操作代码。

避免根据技术相关性划分控制器封装操作

例如，避免创建类似于AjaxController这种包含网站所有AJAX操作的控制器。相反，需要把这些操作按照相关的逻辑分类。例如，提供客户数据或部分视图的AJAX操作方法与其他客户相关的操作都应该放置在CustomersController类中。

操作过滤器置于合适层次

几乎所有的操作过滤器(action filter)标记属性都可以应用到方法(action)或类(controller)级别。当属性标记到控制器中的操作上时，可以选择把属性直接标记到控制器上。另外，也可以进一步考虑属性标记是否能提升到控制器的依赖链上层（比如，控制器的基类）。

使用多个视图（或者部分视图）来替代复杂的If-Then-Else逻辑代码

ASP.NET Web Form控制器模式鼓励回传数据到相同的页面，根据请求消息显示或者隐藏页面的某些部分。由于ASP.NET MVC支持分离关注点原则，所以，可以为每种不同的情况创建单独的视图，减少或消除对复杂视图逻辑的需求。Wizard.cshtml示例代码如下：

```
@if(Model.WizardStep == WizardStep.First) {
    <!-- The first step of the wizard -->
} else if(Model.WizardStep == WizardStep.Second) {
    <!-- The second step of the wizard -->
} else if(Model.WizardStep == WizardStep.Third) {
    <!-- The third step of the wizard -->
}
```

以上是决定显示哪个视图的逻辑代码，它与业务逻辑紧密耦合了。我们把这部分逻辑代码移到控制器（WizardController.cs）中：

```
public ActionResult Step(WizardStep currentStep)
{
    // 这是简单的逻辑，实际上可能很复杂
    string view = currentStep.ToString();

    return View(view);
}
```

可以把原始视图分割成多个视图，比如First.cshtml

```
<!--向导第一步-->
```

Second.cshtml：

```
<!--向导最后一步-->
```

Third.cshtml：

```
<!-- The third step of the wizard -->
```

提交数据时选择Post-Redirect-Get模式

Post-Redirect-Get (PRG)模式是一种通用的设计模式，Web开发人员可以借助这种模式避免重复的表单数据提交操作，而且用户代理使用书签或者点击刷新按钮时更加自然。因为ASP.NET Web Form野蛮控制器模式需要开发人员把同一个上下文特定环境中的所有操作都回传到相同的页面上（例如，显示员工数据，可以编辑并修改数据），所以在ASP.NET MVC中，PRG模式不像ASP.NET Web Form那样用得多。ASP.NET MVC把操作分离成单个的URL，对更新数据的情况，处理起来有点麻烦。思考下面的EmployeeController控制器代码：

```
public class EmployeeController : Controller
{
    public ActionResult Edit(int id)
    {
        var employee = Employee.Get(id);

        return View("Edit ", employee);
    }

    [AcceptVerbs(HttpVerbs.Post)]
    public ActionResult Update(int id)
    {
        var employee = Employee.Get(id);
```

```
        UpdateModel(employee);

        return View("Edit", id);
    }
}
```

在这个例子中，当用户提交数据到Update操作时，用户查找的是Edit视图，而URL显示的还是*/employees/update/1*。如果用户刷新页面，隐藏这个URL地址的书签等，结果就是重新更新这个员工的信息，或者根本不工作。我们真正想要的Update操作是更新完员工的信息后重定向用户到Edit（编辑）页面，这样用户就可以回到原始的Edit（编辑）页面了。这种情况下，也许可以使用PRG模式，这样文件的第一部分就会被忽略掉，只展示修改的Update操作代码：

```
[AcceptVerbs(HttpVerbs.Post)]
public ActionResult Update(int id)
{
    var employee = Employee.Get(id);

    UpdateModel(employee);

    return RedirectToAction("Edit", new { id });
}
```

虽然这只是个从View()方法到RedirectToAction()方法的微小修改，但会导致客户端在更新完用户信息之后重定向（与最初的例子服务端重定向相反）到合适的URL（/employees/edit/1）上。

使用启动任务执行逻辑代码而不是Application_Start(Global.asax)

绝大部分ASP.NET MVC Demo例子建议在Application_Start()文件里修改Application_Start()方法，以便在应用程序开始执行的时候执行某些特定的逻辑。虽然这是最简单、最直接的方法，但WebActivator框架提供了另外一种方法，叫启动任务（startup task）方法。这种新方法非常容易实现，而且可以在应用程序开始的时候自动发现并执行代码。它可以提供更加整洁的代码，而且符合第5章里介绍的SOLID原则之一的单一职责原则（Single Responsibility Principle）。

452

选择Authorize标记属性而不是强制的安全检查

传统上，授权控制的实现类似下面代码：

```
public ActionResult Details(int id)
{
    if (!User.IsInRole("EmployeeViewer"))
        return new HttpUnauthorizedResult();

    // 操作逻辑代码
}
```

授权代码是必须实现的代码，但是这种做法难以进行应用程序级别的修改。ASP.NET MVC AuthorizeAttribute标记属性提供一种简单的、声明式的方式来定义操作授权。这种授权代码也可以重写如下：

```
[Authorize(Roles = "EmployeeViewer")]
public ActionResult Details(int id)
{
    // 操作逻辑代码
}
```

使用路由属性而不是通用全局路由

最特殊的路由就是直接映射到某个操作上，而且仅仅映射到一个操作上。

使用防伪造令牌来避免CSRF攻击

对表单提交来说，要关注的就是安全问题。ASP.NET MVC提供了探测某种攻击类型的机制，其中一种措施就是防伪造令牌（antiforgery token）。这种令牌包含服务端和客户端组件，代码会在表单中插入一个隐藏域以保存用户特定的令牌：

```
@using(Html.Form("Update", "Employee")) {
    @Html.AntiForgeryToken()
    <!--表单剩余代码-->
}
```

而且服务端会优先在数据处理之前执行这些令牌验证代码，如下：

```
[ValidateAntiForgeryToken]
[AcceptVerbs(HttpVerbs.Post | HttpVerbs.Put)]
public ActionResult Update(int id)
{
    // 处理表单数据
}
```

使用AcceptVerbs标记属性限制操作调用

453

许多操作假设了在应用程序中是如何调用和何时调用操作方法的。例如，Employee.Update操作会在某个员工信息编辑页面上，这个页面包含编辑员工信息需要的属性、字段信息，最终会提交给Employee.Update操作方法来更新员工信息。如果调用Employee.Update操作方法的是别的方法（例如GET方法，没有表单数据），那么这个操作方法就不会工作，而且可能会导致一些异常。

ASP.NET MVC框架提供了AcceptVerbs标记属性来限制特定操作的HTTP动词。因此，对前面提到的Employee.Update场景，解决办法如下：

```
[AcceptVerbs(HttpVerbs.Post | HttpVerbs.Put)]
public ActionResult Update(int id)
```

这样，使用AcceptVerbs标记属性将会限制这个操作的HTTP动词为POST或PUT。其他动词都会省略（如GET）。

输出缓存

输出缓存（output caching）是提升网站性能最简单的方法之一。当请求的页面内容不变时，缓存渲染过的HTML代码是提升性能最好的方法。ASP.NET MVC框架提供了`OutputCacheAttribute`属性来实现缓存机制。`OutputCacheAttribute`借鉴了ASP.NET Web Form的输出缓存机制，而且接受更多的属性。

删除没用的视图引擎

ASP.NET MVC默认注册了Web Form和Razor视图引擎，意味着视图定位器（view locator）会为Web Form和Razor视图查找视图。这样网站就可以同时使用两种类型的视图。

但是，为了一致性，大部分团队会选择一种视图类型，而且整个网站中只使用一种视图类型，这样ASP.NET MVC视图定位器就会浪费时间去查找其他的类型。例如，如果用户选择了Razor视图，则视图定位器仍然会去查找Web Form视图，尽管不可能找到这个文件。

幸运的是，可以避免这种无用的工作，可以删除那些不用的视图引擎，以优化应用程序的性能。

下面的代码展示了如何删除Web Form视图引擎的注册（只保留Razor视图）：

```
var viewEngines = System.Web.Mvc.ViewEngines.Engines;

var webFormsEngine = viewEngines.OfType<WebFormViewEngine>().FirstOrDefault();

if (webFormsEngine != null)
    viewEngines.Remove(webFormsEngine);
```

只需要在应用程序启动的时候执行这些代码就可以了，而且视图定位器再不会浪费时间来查找不存在的视图文件了。

对唯一场景自定义ActionResults

ASP.NET MVC请求管道（ASP.NET MVC request pipeline）通过让处理中的每一步完成不同的任务来分离关注点。每步只要为后续需要执行的任务提供足够信息即可。例如，决定要显示给客户端的控制器操作方法并不会加载视图引擎，并命令视图引擎来执行视图。它只是返回下一步（加载视图引擎并执行视图）需要的`ViewResult`对象。

对控制器操作方法结果来说，声明性是最重要的。例如，ASP.NET MVC框架提供了带有`StatusCode`属性的`HttpStatusCodeResult`类，但是它又定义了一个名为`HttpUnauthorizedResult`的自定义`HttpStatusCodeResult`。虽然两行代码在运行效率上是一样的，但后者提供了更具声明性的和满足控制器意图的强类型表达式。

```
return new HttpStatusCodeResult(HttpStatusCode.Unauthorized);

return new HttpUnauthorizedResult();
```

当操作生成的结果无法满足正常的结果时，可以考虑使用自定义操作结果。通用的例子包含像RSS源、Word文档、Excel电子表格等这样的内容。

对并行的控制器任务使用异步控制器

多任务并行支持可以给网站带来更多提升性能的机会。为此，ASP.NET MVC提供了`AsyncController`基类，用于帮助处理多线程请求的问题。当使用多处理器逻辑代码执行操作时，就要思考操作是否包含可以安全并行执行的元素。更多信息可以参考第11章的内容。

交叉引用：目标主题、特性和场景

Cross-Reference: Targeted Topics, Features, and Scenarios

下面列举了已经介绍过的概念、本书中交叉引用这些概念的地方。

主题	章
ASP.NET MVC 4新特性	ASP.NET MVC 4新特性
移动模板	第10章
JavaScript捆绑与压缩	第13章
ASP.NET Web API	第7章
异步控制器	第11章
lowAnonymousAttribute	第9章
ASP.NET MVC 特性	ASP.NET MVC 特性
控制器操作	第1章
操作过滤器	第1章
路由	第1章，第14章
Razor标签	第1章
HTML帮助方法	第1章，第3章
URL帮助方法	第1章
Form帮助方法	第3章
客户端验证	第3章，第8章
区域	第1章
JSON结果	第6章
部分视图	第1章，第6章，第15章
Razor @Helper	第15章
模型绑定	第6章

续表

Topic主题	Chapter(s)章
验证	第3章
错误处理	第16章
视图引擎	第1章
子操作	第12章
输出缓存	第12章
bin可部署程序集	第19章
自定义项目模板	第15章
ASP.NET MVC 项目类型	ASP.NET MVC 项目类型
Empty	第1章
互联网应用	第1章
局域网应用	第9章
移动应用	第10章
Web API	第6章
模式与实践	模式与实践
MVC模式	第5章
N层模型	第5章
SOLID原则	第5章
模型绑定与验证	第6章
对象关系映射（ORM）	第8章
日志与健康监控	第16章
单元测试	第17章
自动化浏览器测试	第17章
SEO	第14章
优雅声明	第13章
逐步增强	第10章
客户端模板	第4章
移动开发	第10章
跨站脚本攻击（XSS）	第9章
跨站请求伪造(CSRF)	第9章

续表

Topic主题	Chapter(s)章
SQL注入攻击	第9章
Web服务和REST	第7章
存储库模式	第8章
持续集成	第18章
持续部署	第18章
Choud/farm deployment	第19章
服务端缓存	第12章
客户端缓存	第12章
工具、框架与技术	工具、框架与技术
jQuery	第4章
客户端验证	第3章
验证/授权	第9章
捆绑和压缩	第13章
ASP.NET Web API	第7章
Entity Framework	第3章，第8章
jQuery Mobile	第10章
Web Sockets	第11章
SignalR	第11章
Windows Azure	第19章
浏览器：本地存储	第12章

索引

Index

Symbols

symbol, 73
$() function, 71, 73
: (semicolon), 27
<% %> code syntax, 54
> symbol, 74
@ symbol, 27, 54

A

absolute expiration, 251
abstractions
  best practices, 443
  repository pattern and, 365
Accept header (HTTP), 150
acceptance tests, 349
AcceptVerbsAttribute class, 143, 453
ActionFilterAttribute class
  about, 23, 90
  OnActionExecuted() method, 126
ActionResult class, 19, 24, 454
actions (see controller actions)
Active Server Pages (ASP), 3
adaptive rendering
  about, 217
  browser-specific views, 221–222
  CSS media queries, 220
  mobile feature detection, 218–220
  viewport tag, 217
Add Application dialog box, 403
Add Controller dialog box, 36
Add View wizard, 39
ADO.NET Entity Framework (see Entity Framework)
aggregate root, 155
AggregateCacheDependency class, 252
AJAX (Asynchronous JavaScript and XML)
  client-side development and, 77–79
  cross-domain, 133–138
  Forms Authentication and, 190
  JavaScript rendering, 117–123
  partial rendering and, 111–117
  responding to requests from, 124
  reusing logic across requests, 123–127
  sending data to the server, 128–133
AllowAnonymousAttribute class, 186, 188, 191
AntiXssEncoder class
  about, 198
  CssEncode() method, 199
  HtmlAttributeEncode() method, 199
  HtmlEncode() method, 199
  HtmlFormUrlEncode() method, 199
  UrlEncode() method, 199
  UrlPathEncode() method, 199
  XmlAttributeEncode() method, 199
  XmlEncode() method, 199
App Cache, 265–267
application development (see web applications)
application service account, 178
application-scoped caching, 250
ASP (Active Server Pages), 3
ASP.NET health monitoring, 338–341
ASP.NET MVC Framework
  about, 3, 4, 317–318
  adding to existing Web Forms, 417–419
  associated namespace, 17, 45
  authentication, 41–43
  authoring using Web Forms syntax, 54
  choosing between Web Forms and, 415

We'd like to hear your suggestions for improving our indexes. Send email to *index@oreilly.com*.

creating applications, 9–15, 35–40
deployment and runtime, 47
differences from Web Forms, 47–54
EBuy project, 8
HTTP handlers and modules, 46
installing, 9
integrating with Web Forms functionality, 420–421
IoC and, 108
logical design in, 90–92
MVC pattern and, 4–6
new features, 6–7
open source availability, 8
project folder structure, 13
rendering HTML, 50–54
routing traffic, 15–18
state management, 46, 49
tools, languages, APIs, 46
transitioning from Web Forms, 416–420
web development platforms, 3–4
ASP.NET platform
about, 45
routing and, 49
ASP.NET session state, 249
ASP.NET Web API
about, 7, 139
building data service, 139–145
exception handling, 147–149
media formatters, 149–152
paging and querying data, 146
asp:Hyperlink tag, 53
asp:Repeater tag, 52
AspCompat page directive, 313
aspnet_regsql.exe command, 340
.aspx pages, 48
ASPX view engine, 55
assemblies
dependencies and, 436
naming, 93
semantic versioning, 437
assembly packages, 430
AsyncController class, 234, 454
asynchronous controllers
about, 6, 233
creating, 234–236
usage considerations, 236
Asynchronous JavaScript and XML (see AJAX)
AsyncManager.OutstandingOperations property, 235
attribute-based routing, 306–310
authentication
about, 41–43, 177
Forms Authentication, 183–191
Single Sign On Authentication, 90
user, 186–187
Windows Authentication, 178–181
authorization
defined, 177
user, 191
AuthorizeAttribute class
action filters and, 23
best practices, 452
controller actions and, 42, 177
usage considerations, 182–183
user authorization and, 191
Autofac site, 107
automated testing
defined, 345
levels of, 345–349
test projects for, 350–354
writing clean tests, 359–361
.axd file extension, 302

B

"backend code", 445
Basic template, 11
BindAttribute class, 176
blacklist-based approach, 196
browsers, 271
(see also web pages)
cache management, 264, 292
HTTP polling and, 238, 239
server-sent events, 240
specific views for, 221–222
testing application logic in, 370
WebSocket API, 241
BufferedMediaTypeFormatter class, 150
build automation
about, 377, 380
continuous integration and, 386, 441
creating, 383–385
creating build scripts, 378–380
executing the build, 379
types of, 381
build scripts, 378–380
bundling concept, 7, 289–293
business rules, specifying with Data Annotations API, 63–65

C

Cache class
  about, 251–252
  adding items to, 262
cache management
  about, 420
  best practices, 276–277, 292, 453
  cache dependencies, 252
  client-side, 248, 264–269, 277
  scavenging process and, 252
  server-side, 248–264
Cache-Control header, 264, 276–277
CacheDependency class, 252
CacheItemPriority enumeration, 252
CacheItemRemovedCallback delegate, 252
Castle Windsor site, 107
CDN (content delivery network), 274
CI (continuous integration), 386–391, 441
client-side caching
  about, 248
  App Cache, 265–267
  browser cache, 264
  LocalStorage mechanism, 268
  setting up, 277
client-side development
  AJAX technique and, 77–79
  DOM manipulation, 76–77
  JavaScript and, 69–71
  responding to events, 74–76
  selectors in, 71–74
  validating data, 79–83
client-side optimization
  about, 271
  anatomy of web pages, 271–273
  ASP.NET MVC support, 289
  avoiding redirects, 283–285
  cache expiration, 276–277
  configuring ETags, 285
  content delivery networks and, 274
  externalizing scripts and styles, 281
  GZip compression, 278
  HTTP requests and, 274
  measuring client-side performance, 286
  minifying JavaScript and CSS, 282
  reducing DNS lookups, 282
  removing duplicate scripts, 285
  script placement on web pages, 279
  stylesheets and, 279
client-side templates, 120–123
code blocks, 27
code coverage in testing, 372–374
Code First approach
  about, 159
  annotation attributes, 162
  convention over configuration, 60
  usage considerations, 161
  working with data context, 167–168
code nuggets, 27
comma-separated values (CSV) format, 150
concurrency conflicts (databases), 160
configuring
  ETags, 285
  IIS, 178
  real-time communication, 245–246
  routes, 16–18
#container element, 112
containers, IoC, 107–109
content delivery network (CDN), 274
Content folder, 428
continuous builds, 381
continuous deployment, 410
continuous integration (CI), 386–391, 441
controller actions
  about, 19
  action filters, 23, 449
  action parameters, 21–23
  asynchronous, 233–236
  AuthorizeAttribute class and, 42
  best practices, 447, 449
  building HTML forms, 57–59
  error handling and, 333
  implementation example, 35–37
  JSONP support, 136
  logging errors, 336
  names corresponding to HTTP actions, 142
  properties for, 16
  repositories and, 155
  returning results, 19
  reusing logic across requests, 123–127
  testing, 361–364
Controller class
  about, 35
  Content() method, 20
  File() method, 20
  HttpNotFound() method, 20
  JavaScript() method, 20
  Json() method, 20, 118

OnException() method, 337
PartialView() method, 20, 112, 114, 124
Redirect() method, 20
RedirectToAction() method, 20
RedirectToRoute() method, 20
View() method, 20, 112
Controller component (MVC pattern)
about, 6, 18
component interaction and, 88–90
Controllers folder, 14, 35
convention over configuration concept
about, 13, 17
Code First approach and, 60
usage considerations, 141–143
CORS (Cross-Origin Resource Sharing), 133, 137
Cross-Site Request Forgery (CSRF), 133, 199–201, 452
cross-site scripting (XSS) attacks, 133, 198
CRUD operations, 142, 143, 155
.cshtml file extension, 317
CSRF (Cross-Site Request Forgery), 133, 199–201, 452
CSS
media queries, 220
minifying, 282
CSV (comma-separated values) format, 150
CustomModelBinderAttribute class, 131
CustomValidationAttribute class, 64
Cutrell, Edward, 296

D

data access layer
about, 161
Code First approach, 161–163
EBuy business model and, 163–166
working with data context, 167–168
data access patterns
about, 153
choosing approach, 159
object relational mappers, 156–158
POCO classes, 153
repository pattern, 154–156
Data Annotations API
client-side validation and, 80
Error Message property, 64
specifying business rules with, 63–65
data manipulation
building data access layer, 161–168
building forms, 57–59
data access patterns, 153–158
Entity Framework and, 158–161
filtering data, 168–174
handling form posts, 59
paging data, 146, 168–174
querying data, 146, 168–174
saving data to databases, 59–61
sorting data, 168–174
validating data, 61–67
data services
building, 139–145
exception handling, 147–149
media formatters, 149–152
paging data, 146
querying data, 146
data transfer objects (DTOs), 133
data:URL scheme, 274
Database class
ExecuteSqlCommand() method, 159
SqlQuery() method, 159
Database First model, 159
databases
concurrency conflicts, 160
deployment considerations, 399
many-to-many relationships, 168, 192
object relational impedance mismatch, 156–158
saving data to, 59–61
DbContext class
about, 61, 167
OnModelCreating() method, 168
DbSet class, 61
DefaultModelBinder class, 129, 131
DELETE method (HTTP), 142
dependencies
best practices, 443
cache, 252
deployment considerations, 399
IoC principle and, 102
mocking, 365–370
version, 439
dependency injection (DI) pattern, 102, 104, 156
Dependency Inversion Principle (DIP), 101
dependency management, 13
DependencyResolver class, 109
deployment
ASP.NET MVC, 47

automating, 390
considerations for, 395–401
continuous, 410
to Internet Information Server, 401–407
web application options, 94
Web Forms, 47
to Windows Azure, 407–410
desktop views
avoiding in mobile site, 216
switching between mobile and, 212
development, application (see web applications)
DI (dependency injection) pattern, 102, 104, 156
DIP (Dependency Inversion Principle), 101
display modes feature, 7, 204
display templates, 318
distributed caching, 259–264
"DLL hell", 436
DNS lookup, 272, 282
document object
DocumentElement property, 70
getElementById() method, 71
write() method, 280
DOM (Document Object Model)
manipulating, 76–77
referencing elements, 71–74
donut caching, 255–257
donut hole caching, 257–258
DRY (Don't Repeat Yourself) principle, 110
DTOs (data transfer objects), 133

E

EBuy project
about, 8
business domain model, 163–166
creating, 9
deployment considerations, 400
editor templates, 318
.edmx file extension, 159
Empty template, 10
Entity class, 164
Entity Framework
about, 60, 158
Code First approach, 60, 159, 161–163, 167–168
database concurrency, 160
Database First model, 159
Model First approach, 159, 161
Entity Tag (ETag), 285
EntityObject class, 161
error and exception handling
about, 331
ASP.NET Web API, 147–149
concurrency conflicts, 160
controller actions and, 333
Data Annotation API, 64
data validation and, 65–67
defining global error handlers, 334–336
enabling custom errors, 332
logging errors, 336–338
ETag (Entity Tag), 285
events
monitoring, 338–341
responding to, 74–76
server-sent, 239
exception handling (see error and exception handling)
ExceptionFilterAttribute.OnException() method, 148
ExpectedExceptionAttribute class, 357
expiration, cache, 251, 276–277
Expires header, 264, 276–277
extension methods, 53

F

filesystem repositories, 432
filtering
controller actions, 23, 449
data, 168–174
errors, 337
foreach loop, managing complexity with, 116
formatters, media, 149–152
forms (see HTML forms; Web Forms)
Forms Authentication, 183–191
FormsAuthentication.SetAuthCookie() method, 187, 188
Fowler, Martin, 387
Franklin, Benjamin, 175
front controller pattern, 89

G

gated check-in builds, 382
GET method (HTTP), 119, 128, 142
Get-Packages command, 438
Git source control systems, 409
Glimpse tool, 305

Global.asax file, 107, 451
GlobalFilterCollection class, 334
Google's best practices rules, 274
Grant-CacheAllowedClientAccount cmdlet, 261
Guan, Zhiwei, 296
GZip compression, 278

H

HandleErrorAttribute class, 149, 333–336, 338
health monitoring, 338–341
HTML
  building forms, 57–59
  handling form posts, 59
  rendering, 50–54, 58
HTML helpers, 52, 317, 446
HtmlHelper class
  about, 33, 317
  ActionLink() method, 53
  EditorFor method, 57
  extending, 53
  HiddenField method, 57
  LabelFor method, 57
  Partial() method, 117
  Password method, 57
  RenderAction() method, 318
  TextBox method, 57
  ValidationMessage() method, 66
  ValidationSummary() method, 66
HTTP handlers, 46
HTTP headers, 150
HTTP Long Polling technique, 238
HTTP methods
  best practices, 274, 285
  CRUD operations and, 142
  JSON hijacking and, 119
  sending data to servers, 128
HTTP modules, 46
HTTP polling, 237–239
HttpActionExecutedContext class, 148
HttpApplicationState class, 250, 251
HttpBrowserCapabilities class, 214, 224
HttpContext class
  Application property, 250
  best practices, 444
  Cache property, 251, 420
  Items property, 47, 249
  Session property, 249, 420
HttpGetAttribute class, 143
HttpPostAttribute class, 142, 143
HttpRequest class
  anatomy of request, 272
  Browser property, 214, 224
  Unvalidated() method, 198
HttpResponse.WriteSubstitution() method, 256
HttpResponseException class, 147
HttpSessionState class, 251, 420
HttpStatusCodeResult class, 454
HttpUnauthorizedResult class, 454
Hub class, 243
hubs, connections and, 243–244

I

ICollection<T> interface, 166
IComparable interface, 64
IConfigurationManager interface, 245
IController interface, 17
IDependencyResolver interface, 109
IDictionary interface, 249
IDisposable interface, 100
IEntity interface, 163
IEnumerable<T> interface, 174
IEquatable interface, 163
IExceptionFilter interface, 148
If-Modified-Since header, 265
If-None-Match header, 286
if/else statement
  best practices, 446
  Web Forms example, 26
IHttpAsyncHandler interface, 314
IHttpHandler interface, 311
IIS (Internet Information Server)
  asynchronous controllers and, 6
  client caching and, 277
  configuring, 178–181
  deploying to, 401–407
IIS Express dialog box, 179
IKernel interface, 109
inheritance concept, 157, 242
Install-Package command, 13, 437, 440
installing
  ASP.NET MVC Framework, 9
  NuGet Package Manager, 423
  packages from PackageManager Console window, 13
  Razor Single File Generator, 319

Velocity, 259
integration machines, 388
integration tests, 348
Interface Segregation Principle (ISP), 100
Internet Application template, 11, 42, 184
Internet Information Server (IIS)
asynchronous controllers and, 6
client caching and, 277
configuring, 178–181
deploying to, 401–407
Intranet Application template, 11, 178
intranet applications, securing, 178–183
Inversion of Control design principle (see IoC design principle)
IoC (Inversion of Control) design principle
about, 102
dependencies and, 102
dependency injection pattern, 104
picking containers, 106–109
service location and, 104
IQueryable<T> interface, 146
IRepository interface, 143, 168, 366
IRouteConstraint interface, 304
IRouteHandler interface, 311
ISerializable interface, 100
ISP (Interface Segregation Principle), 100

J

JavaScript language
client-side development and, 69–71
minifying, 282
referencing DOM elements, 71–74
rendering and, 117–123
responding to events, 74
JavaScript Object Notation (see JSON)
JavaScriptSerializer class, 307
jQuery library
$() function, 71, 73
about, 69–71
.after() method, 77
.ajax() method, 79, 135
.before() method, 77
.click() method, 75
client-side validation, 79–83
.contains() method, 74
.css() method, 73
.done() method, 79
.error() method, 79, 136
.fail() method, 79
.getJSON() method, 142
.height() method, 71
.html() method, 77, 120, 123
JSON data and, 132
.load() method, 112
manipulating elements, 76
.post() method, 128
.prepend() method, 77
referencing DOM elements, 71–74
responding to events, 75
.success() method, 79, 136
.text() method, 73
.val() method, 120
.width() method, 71
jQuery Mobile Framework
about, 204
adaptive rendering, 217–222
creating mobile applications from scratch, 224–228
data-filter attribute, 211
data-role attribute, 210, 228
enhancing views with, 209–215
getting started with, 207–209
improving mobile experience, 216
Mobile Application template and, 12, 203
paradigm shift, 224
"listview" component, 210
jQuery.Mobile.MVC package, 207, 213
JSON (JavaScript Object Notation)
posting complex objects, 129
rendering data, 118–119
requesting data, 119
responding to requests, 125
sending and receiving data effectively, 132
JSON hijacking, 119
JSONP (JSON with Padding)
about, 133–135
controller actions and, 136
making requests, 135
JsonRequestBehavior enumeration, 137

L

Language Integrated Query (LINQ), 168–174, 308
Last-Modified header, 265, 286
layout template, 28
layouts
loading for mobile views, 207
master pages versus, 54

web applications and, 28
lazy loading technique, 280
least privilege, principle of, 176
Library Package Manager Console, 13
libs folder, 429
LINQ (Language Integrated Query), 168–174, 308
LINQ to Entities injection attacks, 197
Liskov Substitution Principle (LSP), 98
"listview" component (jQuery Mobile), 210
LocalStorage mechanism, 268
Logger class, 336
logging errors, 336–338
logical design in web applications, 90–93
LSP (Liskov Substitution Principle), 98

M

magic strings, 444
.manifest file extension, 266–267
manual testing, 344
many-to-many relationships, 168, 192
MapRoute() extension method
about, 17
method override and, 303
parameters and, 299
registering routes for applications, 306
master pages, layouts versus, 54
media formatters, 149–152
media queries, 220
MediaTypeFormatter class
about, 150
CanReadType() method, 150
CanWriteType() method, 150
MEF site, 107
Membership class
CreateUser() method, 188
GetUser() method, 189
ValidateUser() method, 187
MembershipUser class, 189
meta packages, 431
MIME types, 149, 267
minification concept, 7, 282, 289–293
Mobile Application template
about, 12, 203
usage considerations, 226–228
ViewSwitcher widget and, 212
mobile feature detection, 218–220
Mobile template, 224
mobile views
browser-specific, 221–222
creating, 205
enhancing with jQuery Mobile, 209–215
loading layouts for, 207
overriding regular views with, 204
switching between desktop and, 212
mobile web development
adaptive rendering, 217–222
creating applications from scratch, 224–228
features supporting, 203–205
improving mobile experience, 216
usability considerations, 205–216
mocking dependencies, 365–370
model binding
about, 21–23
data annotations and, 63
JSON and, 128, 130
registering binders, 132
specifying, 131–132
Model component (MVC pattern)
about, 5, 34
component interaction and, 88–90
Model First approach, 159, 161
@model keyword, 33
Model-View-Controller pattern (see MVC pattern)
ModelBinderDictionary.GetBinder() method, 131
ModelBinders class, 131
Models folder, 13
ModelState class
about, 62
AddModelError() method, 62
monitoring system health, 338–341
MSBuild tool, 380, 405–407
Mustache template syntax, 120
mustache.js library, 120
MVC (Model-View-Controller) pattern
about, 4–6, 87
component interaction and, 88–90
Controller component, 6, 88–90
Model component, 5, 34, 88–90
reusing logic across requests, 123–127
separation of concerns principle, 87, 154
View component, 6, 88–90
MvcDonutCaching NuGet package, 257
MvcRouteHandler class, 311

N

namespaces
  ASP.NET-related, 45
  naming, 93
navigating data, 158
New ASP.NET MVC Project dialog box, 12
new keyword, 443
Ninject IoC containers, 107, 144
nuget pack command
  about, 424
  -Properties switch, 427
  -Version switch, 427
NuGet Package Explorer, 425
NuGet package management tool
  anatomy of NuGet packages, 427–430
  creating NuGet packages, 424–427
  hosting package repositories, 432–435
  installing, 423
  sharing NuGet packages, 431–435
  SignalR signaling library and, 241
  tips, tricks, and pitfalls, 435–441
  types of NuGet packages, 430
  usage considerations, 424
NuGet Package Manager
  about, 12
  accessing, 13
  best practices, 443
  installing, 423
NuGet packages
  anatomy of, 427–430
  controlling versions, 440
  creating, 424–427
  generating from NuSpec files, 426
  sharing, 431–435
  types of, 430
  version control, 436–441
nuget push command, 432
NuGet Server repository, 434
nuget setApiKey command, 432
nuget spec command, 424
NuGet.org repository
  NuGet package upload wizard, 431
  publishing to, 431
NuSpec files
  about, 424–426
  generating NuGet packages from, 426
  version dependencies, 439

O

obfuscation technique, 283
object relational impedance mismatch, 156–158
object relational mappers (ORMs), 154, 156–158
observer pattern, 88
OCP (Open/Closed Principle), 97
OData (Open Data Protocol), 146
onClick event, 74–76
onsubmit event, 83
Open Data Protocol (OData), 146
Open/Closed Principle (OCP), 97
optimistic concurrency approach, 160
OptimisticConcurrencyException class, 160
optimization techniques (see client-side optimization)
ORMs (object relational mappers), 154, 156–158
output caching, 252–255, 453
OutputCache class, 253, 256
OutputCacheAttribute class
  about, 253
  best practices, 453
  donut hole caching and, 258
  parameters supported, 253–255

P

Page class, 313
paging data, 146, 168–174
partial rendering, 111–117
partial views
  about, 29, 317
  rendering, 112–117
  user controls versus, 54
password management, 188
persistence ignorance (PI), 153
persistent connections, 242
PersistentConnection class, 242, 243
pessimistic concurrency approach, 160
physical design in web applications, 93, 94–96
PI (persistence ignorance), 153
pipeline, routing, 310–315
Plain Old CLR Objects (POCOs), 60, 153
PluralizingTableNameConvention class, 163
POCOs (Plain Old CLR Objects), 60, 153
POST method (HTTP), 128, 142
Post/Redirect/Get (PRG) pattern, 450

precompiled views, 323–324
PrecompiledMvcEngine package, 323
prerelease packages, 438
prerelease versioning, 439
principle of least privilege, 176
project templates, 10–12
projects, naming, 93
properties, controller actions, 16
publishing
  from within Visual Studio, 403–407
  to NuGet.org repository, 431
  Windows Azure website via source control, 409
PUT method (HTTP), 142

Q

quality control
  automated testing and, 343–376
  build automation and, 377–391
  logging and, 331–341
querying data, 146, 168–174

R

RangeAttribute class, 64
Razor Single File Generator
  creating reusable helpers, 325–326
  creating reusable views, 321–324
  installing, 319
  unit testing Razor views, 327–328
Razor syntax
  @ symbol, 27, 54
  about, 12, 26–27
  differentiating code and markup, 27
  layouts and, 28
  rendering web pages, 51
Razor view engine, 256, 323
real-time data operations
  about, 236
  comparing application models, 237
  configuring and tuning, 245–246
  empowering communication, 241–244
  HTTP Long Polling technique, 238
  HTTP polling, 237
  server-sent events, 239
  WebSocket API, 240
redirects, avoiding, 283–285
Remote Procedure Call (RPC) framework, 243
rendering
  adaptive, 217–222
  HTML, 50–54, 58
  JavaScript, 117–123
  JSON data, 118–119
  partial, 111–117
  partial views, 112–117
  web pages, 271–273, 280
repository pattern, 154–156, 365
request-scoped caching, 248
Request.IsAjaxRequest() method, 124, 125
RequiredAttribute class, 63
rolling builds, 381
RouteData class, 298
RouteGenerator class, 308–310
RouteValue dictionary, 214
routing
  about, 15
  ASP.NET approach, 49
  attribute-based, 306–310
  best practices, 452
  building routes, 298–303
  catch-all routes, 302
  configuring routes, 16–18
  determining pattern for, 35
  extending, 310–315
  ignoring routes, 302
  registering Web API routes, 141
  route constraints, 303–306
  URLs and SEO, 297
  wayfinding, 295–297
RPC (Remote Procedure Cal) framework, 243
runtime considerations
  ASP.NET MVC, 47
  Web Forms, 47

S

scavenging process, 252
scheduled builds, 382
scripts
  build, 378–380
  deferring execution of, 280
  executing with MSBuild, 406–407
  externalizing, 281
  lazy loading technique, 280
  placement on web pages, 279–281
  removing duplicate, 285
@Scripts annotation, 289
Search Engine Optimization (SEO), 297
Search view, 170

securing web applications
about, 177
defense in depth, 175
disabling unnecessary features, 177
distrusting input, 176
Forms Authentication, 183–191
guarding against attacks, 192–201
insecurity of external systems, 176
intranet applications, 178–183
principle of least privilege, 176
reducing surface area, 176
selectors in client-side development, 71–74
Semantic Versioning scheme, 437, 439
semicolon (;), 27
SEO (Search Engine Optimization), 297
separation of concerns principle, 87, 154
server controls, 52
server-sent events, 239
server-side caching
about, 248
application-scoped caching, 250
distributed caching, 259–264
donut caching, 255–257
donut hole caching, 257–258
output caching, 252–255
request-scoped caching, 248
user-scoped caching, 249
service locator pattern, 102, 104
session states, 249
SignalR signaling library, 241, 243, 245
Single Responsibility Principle (SRP), 96
Single Sign On Authentication, 90
sliding expiration, 251
SOLID design principles, 96–101, 163
sorting data, 168–174
SQL injection attack, 192–197
SQLCMD utility, 407
SRP (Single Responsibility Principle), 96
SSL encryption, 183
StandardKernel class, 109
Start-CacheCluster cmdlet, 261
state management, 46, 49
static content, 398
stored procedures, 158, 159
storing session data, 250
StringLengthAttribute class, 63
StructureMap site, 107
@Styles annotation, 289
stylesheets, best practices, 279
Substitution control, 256
SUT (system under test), 345
synchronous communication, 78
system under test (SUT), 345
System.Data.Entity namespace, 60
System.Web namespace, 45
System.Web.Mvc namespace, 17, 45
System.Web.Optimization namespace, 289
System.Web.Security.AntiXss namespace, 198
System.Web.UI namespace, 45
System.Xml namespace, 46

T

TDD (test-driven development), 358
Team Foundation Server tool, 381, 383–385, 409
TempData dictionary, 31
templates
authentication and, 178, 184
client-side, 120–123
controller, 36
display, 318
editor, 318
layout, 28
mobile application, 12, 203, 212, 226–228
project, 10–12
test classes, 350
test doubles, 365
test fixtures, 350
test projects
creating, 350
defined, 350
test-driven development (TDD), 358
testing
applications, 354–372
automated, 345–349, 359–361
builds, 387
code coverage in, 372–374
controllers, 361–364
developing testable code, 374–376
manual, 344
mocking dependencies, 365–370
models, 355–357
refactoring to unit tests, 364
TDD and, 358
test projects and, 350–354
views, 370–372
TestInitializeAttribute class, 359
timestamps, 135, 160

tool packages, 430
tools folder, 429
try/catch block, 161, 336
tuning real-time communication, 245–246

U

UAT (user acceptance testing), 349
unit testing
  about, 345–348
  creating and executing, 352
  Razor views, 327–328
  refactoring to, 364
Unity site, 107
UrlHelper class, 33, 116
UrlRoutingModule class, 311
URLs
  ASP.NET MVC approach, 48
  SEOs and, 297
  wayfinding and, 295–297
  Web Forms approach, 48
user acceptance testing (UAT), 349
User class, 164, 191
user controls, partial views versus, 54
user management
  about, 420
  authenticating users, 186–187
  authorization process, 191
  changing passwords, 188
  registering new users, 187
user-scoped caching, 249

V

ValidateAntiForgeryTokenAttribute class, 200
validating data
  about, 61
  best practices, 92
  client-side development and, 79–83
  displaying errors, 65–67
  specifying business rules, 63–65
.vbhtml file extension, 317
Velocity distributed caching solution, 259
View component (MVC pattern)
  about, 6
  component interaction and, 88–90
view engines
  about, 12
  ASPX, 55
  best practices, 453
  Razor, 256, 323
  Web Forms, 319
View State mechanism
  about, 46, 50
  usage considerations, 54, 55, 421
View Switcher component, 208
ViewBag object, 32
ViewData dictionary, 31, 65, 444
viewport tag, 217
ViewResult class, 19, 24, 454
views, 204
  (see also mobile views)
  about, 24
  best practices, 445
  creating reusable, 321–324
  differentiating code and markup, 27
  display modes feature and, 204
  displaying data, 31–33
  HTML and URL helpers, 33
  implementation example, 38–40
  layouts and, 28
  locating, 24
  overriding regular with mobile, 204
  partial, 29, 54, 112–117, 317
  precompiled, 323–324
  Razor, 26–27, 319, 327–328
  Search, 170
  separation of application and view logic, 48
  testing, 370–372
  Web Forms syntax and, 54
Views folder, 14
ViewSwitcher widget, 213–215
ViewUserControl class, 319
Visual Studio, publishing from within, 403–407

W

WatiN tool, 371
wayfinding, 295–297
Web API (see ASP.NET Web API)
Web API template, 12
web applications, 69
  (see also client-side development; mobile web development; securing web applications)
  architecting, 90–96
  authentication and, 41–43

convention over configuration, 13, 17, 60, 141–143
creating, 9, 35–40
deployment options, 94
development techniques, 49
differentiating code and markup, 27
DRY principle, 110
IoC design principle, 102–109
layouts and, 28
logical design in, 90–93
Microsoft development platforms, 3–4
MVC pattern, 4–6, 87–90
naming considerations, 93
physical design in, 93, 94–96
project templates, 10–12
Razor syntax and, 26–27
running, 15
separation of application and view logic, 48
SOLID design principles, 96–101, 163
testing, 354–372
web browsers, 271
(see also web pages)
cache management, 264, 292
HTTP polling and, 238, 239
mobile feature detection, 218–220, 218–220
server-sent events, 240
specific views for, 221–222
testing application logic in, 370
WebSocket API, 241
Web Forms
about, 4
adding ASP.NET MVC to existing applications, 417
AspCompat page directive, 313
associated namespace, 45
authoring ASP.NET MVC views using, 54
choosing between ASP.NET MVC and, 415
deployment and runtime, 47
differences from ASP.NET MVC, 47–54
HTTP handlers and modules, 46
if/else statement example, 26
integrating with ASP.NET MVC functionality, 420–421
rendering HTML, 50–54
state management, 46, 49
tools, languages, APIs, 46
transitioning to ASP.Net MVC, 416–420
ViewUserControl class and, 319
Web Forms Page Controller pattern, 449
web pages
anatomy of, 271–273
avoiding redirects, 283–285
cache expiration, 276–277
configuring ETags, 285
content delivery networks and, 274
externalizing scripts and styles, 281
GZIP compression, 278
HTTP requests and, 274
minifying JavaScript and CSS, 282
reducing DNS lookups, 282
removing duplicate scripts, 285
script placement on, 279–281
stylesheets and, 279
web.config file
authentication-mode element, 178
client-side validation settings, 80
data access class names in, 168
distributed caching settings, 262
membership and role providers, 185
output caching section, 255
packagesPath setting, 434
WebRequestErrorEvent class, 340
WebSocket API, 240
whitelist-based approach, 197
window object
about, 71
onload event, 75
Windows Authentication, 178–181
Windows Azure, 407–410

X

XmlHttpRequest object
about, 77
open() method, 78
send() method, 78
status attribute, 78
XSS (cross-site scripting) attacks, 133, 198

Y

Yahoo!'s Exceptional Performance team, 273
YSlow tool, 286
YUI Compressor, 282

作者介绍

Jess Chadwick：专注于Web技术领域，独立软件技术顾问；有超过10年的软件开发经验，就职的公司包括嵌入式设备创业公司以及世界500强企业；ASPInsider和微软ASP.NET领域的MVP，经常在新泽西的NJDOTNET进行技术演讲。Jess与贤惠的妻子、女儿还有一条黑色的拉布拉多犬居住在美国宾夕法尼亚州的费城。

Todd Snyder：从事软件开发工作超过18年，他提供基于微软平台的企业级系统架构技术咨询和开发指导。在Infragistics工作期间，他的主要工作就是设计和构建RIA、移动和N层应用系统。Todd是新泽西州.NET User Group（*http://www.njdotnet.net/*）的创始人之一，经常作为讲师受邀参加各种技术会议。

Hrusikesh Panda：架构师、RIA专家，他还是一位充满激情的.NET开发人员、Web架构师以及开源爱好者。

封面介绍

本书封面动物是一条银色的带鱼（安哥拉带鱼，Lepidopus caudatus）。这类鱼都很细长，呈钢铁蓝或银的颜色，这就是其名字的由来。它们没有盆腔和尾鳍，这使得它们的外形很像鳗鱼。成年鱼类可以长到2米多，体重达9 kg。银色无鳞高度扁平的身体看起来像彩带一样，当它们游过我们身边时，就像一道白光划过。带鱼上下颚上都有锋利的牙齿，而且下颚比较突出。通常雌性带鱼比雄性更长、更大。

带鱼种类很多，它们属于食肉动物，主要捕食甲壳类动物（特别是磷虾和十足类）、小型软体动物、雷翅片灯笼鱼、豚鼻鱼和鲱鱼等作为食物。它的主要天敌是鲨鱼、无须鳕和鱿鱼。在大西洋、地中海和太平洋等热冷水域，都可以找到这个物种，并且带鱼可以在1000米水下生活。它们分布十分广泛，在墨西拿海峡有专门一种捕捞带鱼的职业：spadularu或者带鱼渔夫。

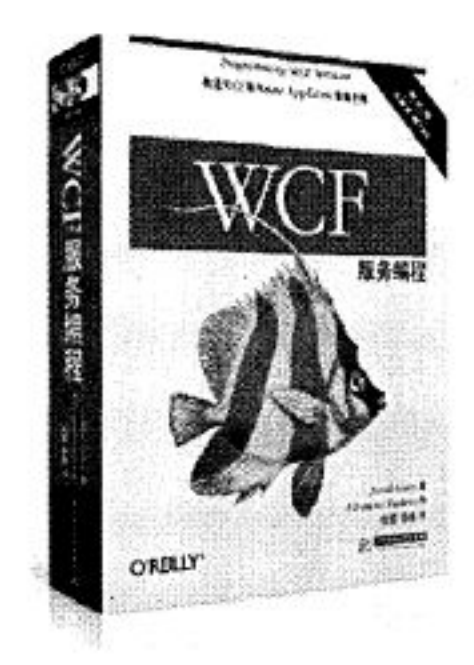
其独特的口味也为其赢得了不少美誉，意大利墨西拿（Messina）当地人称之为“海之少女”。它们柔软、口感独特而且无鳞，非常适合食用（尽管外形有点难看）。

图片摘自于约翰的《自然史》（Johnson’s Natural History）。

《WCF服务编程》（第三版）——.NET开发者决战SOA的制胜利剑！

世界范围内经典的WCF书籍！学习WCF的两把利剑！评价五颗星！

微软硅谷区域总监，WCF项目架构顾问Juval Löwy著作。



当当网读者评价摘选

trampor：正在进行相关开发，这本书很合适，深入浅出，强力推荐！

GhOstRider ：介绍得很详细，WCF各种配置，实现方式，配合作者的个人心得，讲得好。

无昵称用户：内容一如既往的精彩，之前看了第二版，这次买主要是冲着4.0特性和服务总线来的，这次的翻译真的比上次的还要棒，通俗了许多，非常好。

小烟99：WCF经典书籍，这本书的确是经典，号称和《CLR VIA C#》齐名。是学WCF的必选，比起国内那些书好多了，有国外的书籍是不会买国内的。

知佛不知经：很好的一本书。我毕业后做了一年的WinForm，没什么服务编程基础，此书看了一个星期就融入到项目中了，我的感觉是这本书的附录含金量更高。

京东读者评价摘选

解释所有不解释问题：相当不错！！！推荐。等了一个月，终于到手了，看后有种爱不释手的感觉，是学习WCF、SOA架构的必备书。

keyname：经典，个人觉得没得说，配合MSDN应该就够了。

yuer5：很不错的一本书。之前也看过一些WCF的书，感觉还是这本最好，由浅入深，讲的也比较有深度，是一本不可多得的好书，最主要的是翻译得不错。

爱生活_爱网购：粗略翻了一下，感觉本书非常值得一看。作者的来头不小，同时对WCF有深入了解，参加过MS的WCF战略性评审，同时有自己独到的见解，对SOA的介绍就显得非一般作者所写。但是读者朋友也要做好准备，这方面的书籍不像WPF等与界面相关的书籍那样直观，初看时会感觉到比较抽象，读不出书中的核心内容。

帅气的大白菜：权威著作,内容详实,讲解细腻,最新版本,融合了理论原理与编程,很不错,正在细

细品读!

Micro-Sanvey：深入、经典、详细、绝对值。

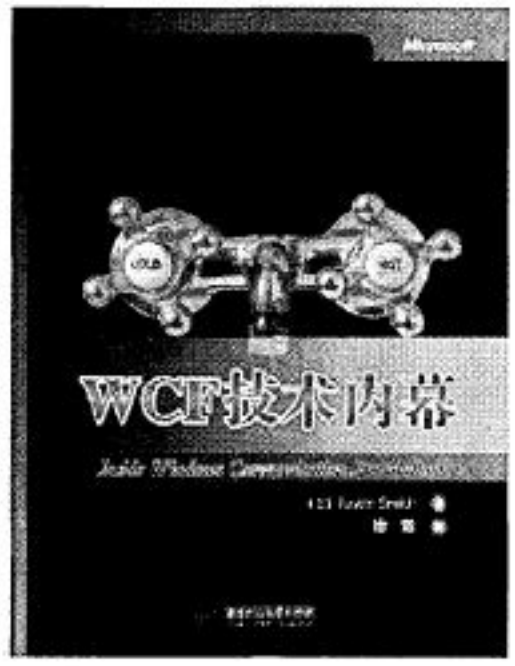
小煜娃娃：想学WCF的话，本书非常适合入门，作者给出的一些帮助类也相当有价值。

往事如风0112：真正高级编程书籍，不适合初学者。非常棒的WCF编程参考书，介绍的知识点和编程技巧非常有用。建议想提高WCF水平的一定要读。

孔曰成仁孟曰取义：非常满意。好书，值得购买，内容很棒!

《WCF技术内幕》——深入WCF框架底层机制的制胜利剑!

微软WCF Data Service团队开发经理Justin Smith著。



京东读者评价摘选

80x86_LM：内容比较深入，针对的是底层的原理，对初学者来说特别难。

聖光牧浴：3.14买的，WCF、SOA尽含其中。这本书偏重概念，特别适合初学者，再配合谷歌开源，学起来相当快。

believemee：经典书籍不容错过，学习WCF的好帮手。

irene9751：深入理解WCF的利器。内容详实，理论通俗易懂，不过不适用于初学者，最好是使用过一段时间WCF后再来看此书会更有效果

帅气魔鬼：介绍很全面，适合有经验的读者学习。

毅无涯：很值得推荐的书！这本书介绍的是WCF的原理,大家可以看一下。