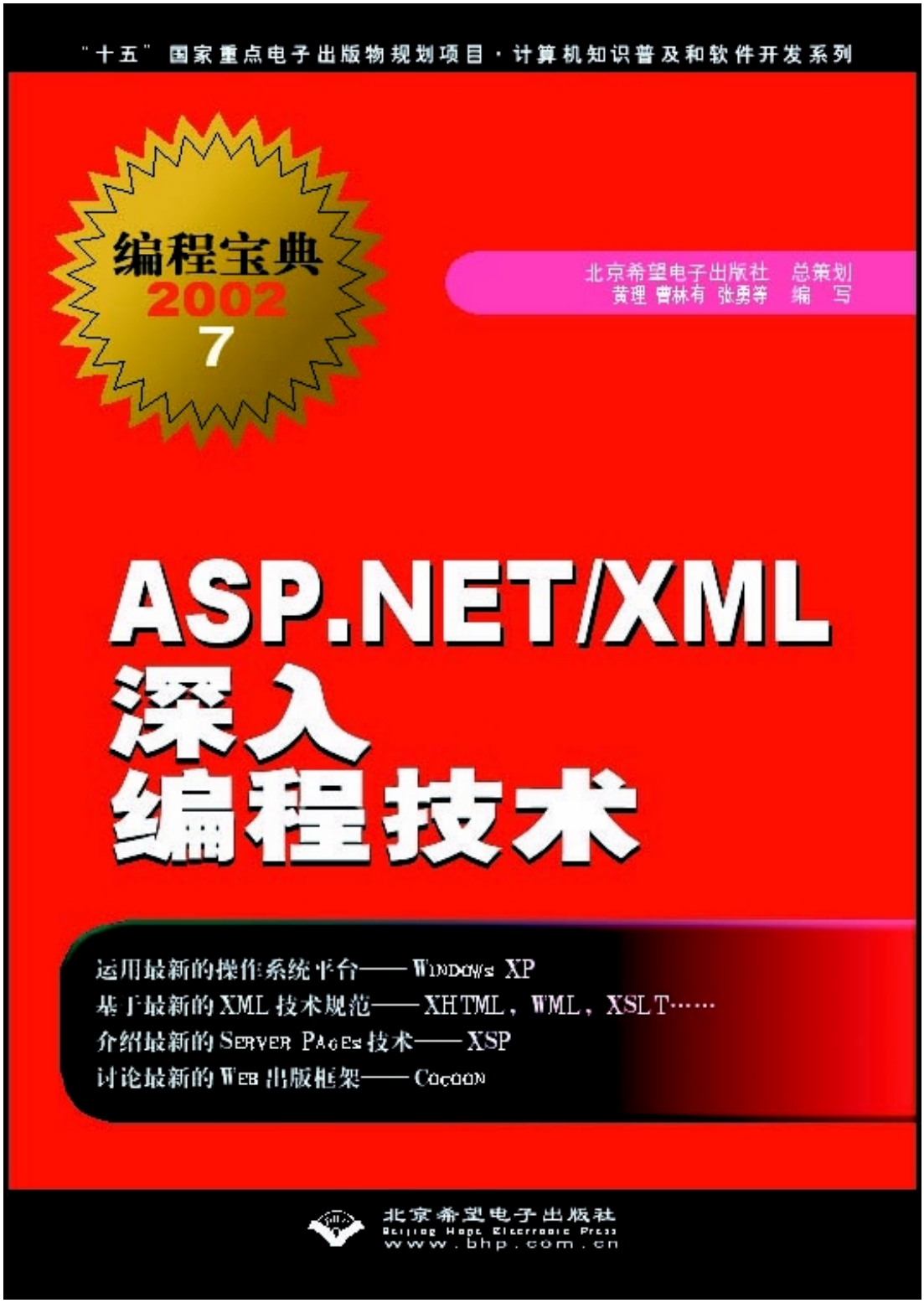

"十五" 国家重点电子出版物规划项目·计算机知识普及和软件开发系列
编程宝典
2002
7
北京希望电子出版社 总策划
黄理 曹林有 张勇等 编 写
ASP.NET/XML
深入
编程技术
运用最新的操作系统平台——Windows XP
基于最新的XML技术规范——XHTML，WML，XSLT……
介绍最新的Server Pages技术——XSP
讨论最新的Web出版框架——Cocoon
北京希望电子出版社
Beijing Hope Electronic Press
www.bhp.com.cn

“十五”国家重点电子出版物规划项目·计算机基础知识普及和软件开发系列

编程宝典 2002 （7）

ASP.NET/XML 深入编程技术

北京希望电子出版社　总策划

王超　张鹏　编写

本书特点

作者精心设计 66 个典型实例深入介绍 ASP.NET/XML 编程方法和技巧

读者对象

面向初、中级网络编程人员

对高级编程员也有重要参考价值

封面（或封底）宣传语：（本系列均适用）

带领你进入神秘的技术前沿

体会酣畅淋漓的新技术快感

掌握最新技术的第一手情报

全面引爆前方的新技术阵地

北京希望电子出版社
Beijing Hope Electronic Press
www.bhp.com.cn

内 容 简 介

这是关于 ASP.NET/XML 深入编程的技术书。作者精心设计了**66 个实例**详细介绍了.NET Framework 以及 XML 技术在.NET Framework 中的编程方法和技巧。

全书由 12 章构成，内容包括.NET 平台的建立、ASP.NET 的 Web Forms、控件、数据访问、Web Service、ASP.NET 的设置和跟踪、ASP.NET 的安全、ASP.NET 的缓冲机制等技术，对于 XML，书中则详细地介绍了 XML 的语法与应用，包括 DTD、Schema、XSLT、XPath 等，还有在.NET 中的 DOM 接口和模拟 SAX 的接口。

本书的**突出特色**：用丰富的范例将两种新技术融合的编程方法进行了详细描述，实用性和指导性强。通过学习，读者可以灵活自如地运用这些新技术于开发实践。

本书不但是从事用 ASP.NET/XML 进行编程的广大初、中级人员的自学指导书，同时对高级程序员也有重要参考价值。

本版 CD 为配套书。

盘书系列名：“十五”国家重点电子出版物规划项目·计算机基础知识普及和软件开发系列
编程宝典 2002 （7）
盘　书　名：ASP.NET/XML深入编程技术
总　策　划：北京希望电子出版社　总策划
文本著者：王超　张鹏
责任编辑：蒋湘群
CD 制作者：希望多媒体开发中心
CD 测试者：希望多媒体测试部
出版、发行者：北京希望电子出版社
地　　址：北京中关村大街 26 号，100080
网址: www.bhp.com.cn
E-mail: lwm@hope.com.cn
电话: 010-62562329,62541992,62637101,62637102,62633308,62633309（发行部）
010-62613322-215（门市）　010-62547735（编辑部）
经　　销：各地新华书店、软件连锁店
排　　版：希望图书输出中心　马君
CD 生产者：北京中新联光盘有限责任公司
文本印刷者：北京双青印刷厂
开本 / 规格：787 毫米×1092 毫米　16 开本　20.75 印张　492 千字
版次 / 印次：2002 年 1 月第 1 版　2002 年 1 月第 1 次印刷
印　　数：0001-5000 册
本 版 号：ISBN7-900088-05-9
定　　价：35.00 元（本版 CD）

前　言

Web 已经进入一个新时代！

上个世纪末，也就是短短的几年前，当 HTML 等 Web 技术呈现在我眼前时，这种语言看起来就像是一套玩具。我们尽可以利用它来搭建最酷、最炫的网页，将商业信息、个人风采展现在全世界的面前。然而很可惜，我是一个程序员——没有美工头脑，只有逻辑思维。我关心的是语言本身能否表达逻辑，表达和实现客户需求。企业大可以将商业的信息发布在 Web 上，我不否认这是种“超低成本+超大覆盖面”的宣传方式，特别是 Internet 用户正在以指数速度增长的时候。但是，在最初的时候，这种技术仅仅可以让企业一厢情愿的将静态的信息“推”给潜在的客户。我希望在某个音乐网站里寻找一首歌曲，于是我必须顺着超链接进入歌曲分类目录、歌手目录……Oh!页面不存在！

接下来，CGI、ASP、PHP 等技术为 HTML 页面加入了响应能力，从而带来了第一代交互式的 Web。我喜欢这种思路，因为写桌面程序的时候，我和我的程序用户都喜欢程序的界面可以获取用户的输入，而不是哗啦哗啦地光打印出一串“据说是有价值的”信息。Web 也具有这样的能力了！可是这些语言都太过简单，它们只是“脚本”语言。在这里我没有任何贬低的意思，这么说只是因为脚本语言往往（当然不是“总是”）是简单而能力有限的。使用它们当然可以完成很多任务，然而我总有些不满。当我第一次不得已用 ASP 写个论坛的时候，我发觉事情还是一团糟：我必须和用户界面打交道，而我尽管可以写 HTML 代码，却写不出来漂亮的页面；除此之外，我任务中的许多代码必须是“嵌”在 HTML 代码中的。于是我必须等设计页面的人把 HTML 模板写好后再添加 ASP 脚本代码。而且我用的是 VBScript 和 JScript，这是我第一次接触它们，在浏览了一遍参考手册后，我很沮丧。我没法使用很多我一向用来自以为是的技巧……

在 2000 年 6 月，微软公司宣布了.NET 战略。这个宏伟的战略所表达的思想，连微软内部的不少人也为之迷惑。时隔一年，随着.NET Framework SDK Beta 1 和 SDK Beta2、Visual Studio.NET Beta 2 的相继发布，这个战略已经慢慢显露出其雄心勃勃的内涵。抛开众多微软反对者的惊讶和微软推出这个战略的野心不讲，.NET 战略的确凭借其技术上设计的精良，体现了下一代软件开发的新趋势。“将软件作为服务提供”、“使信息尽在指尖”一直到.NET 新的表述“Empower people through great software any time, any place and on any device”，这种思路长期以来只是业界和计算机科学家们的梦想。现在，.NET 平台框架正开始提供实现梦想的强大工具。

在 HTML 正深入人心的时候，技术先锋们已经意识到这种语言的不足。应运而生的是 XML 语言和相关技术的产生。这种语言可以分离数据和数据表现形式，使得数据具有了描述自身的能力。紧接着 Microsoft 和 IBM 等便迅速提出 Web Services 的概念，使用 Web 的标准协议和 XML 来传输数据，进而实现松散耦合的远程调用。由于使用 XML，Web Service 可以描述自身功能和调用规范；由于使用 Web 传输技术，Web Services 可以轻易的穿透防火墙并被各种各样的智能设备调用。因此，“软件服务”、“Any Time And Place on Any Device”将不再是遥不可及的事情。作为软件开发人员，我喜欢这种想法，因为我将可以在 Web 上作更多的事情，利用 Web，不仅可以展示内容，还可以提供和利用服务！Web Service 和 XML 将 Web 上现有的“信息孤岛”联结起来，提供了软件开发的新思路。这种思路并不是.NET 战略专有的，即便是那些微软的反对者，我们也建议好好研究它引领的软件工业的新趋势。

.NET 框架以迅雷不及掩耳的速度，将正在兴起的 Web Service 包容在一整套的开发工具和软件产品当中。使用 ASP.NET 和.NET 的基础类库，软件开发人员可以轻松地开发出强大的 Web Service，并利用其他 Web Service 开发商业软件。

本书针对当前的形势，选择.NET 中最关键和最具有革命性的两项技术：XML 和 ASP.NET 作为主题，结合作者的开发经验，进行较为深入的探讨。本书撰写于.NET Framework SDK Beta 2 发布时。由于 Beta 1 和 Beta 2 的差别较大，撰写时的参考资料极为有限，因此我们加入了不少开发过程中积累的第一手经验，探讨了不少 Beta 2 参考文档中没有包含或言之不详的技术，希望能起到抛砖引玉的作用，以便读者不仅了解这些技巧，还可以得到探索.NET 经验的一些启示。值得庆幸的是，微软承诺 Beta 2 和即将发布的“厂商内部发行版本”以及最终的发布版本在基础结构上将不会有大的不同，因为.NET 产品的后期开发工作主要是平台的运行性能的调整上。因此，本书中介绍的技术和技巧，相信在最终的.NET 版本上可以较为顺利地被采用。

本书假定读者具有一定的编程经验。如果读者具有 ASP 或其他服务器端动态页面脚本语言编程背景，将会发觉 ASP.NET 与 ASP 的某些相通之处。但是请注意，作为 ASP 的改良版，ASP.NET 技术带来的不仅仅是动态的内容（Content）展示技术，更重要的是，ASP.NET 将是开发 Web 应用程序、实现一个 Programmable Web、最终将被称之为“信息孤岛”的现有的 Web 网站和 Web 应用程序连接起来的一项重要技术。如果读者具有面向对象编程经验，会发现 ASP.NET 具有他们所需要的强大的面向对象特征和动力，从而他们可以使用 ASP.NET 和面向对象技术来实现 Web 应用的解决方案。

ASP.NET 的编程可以使用任何一种.NET 平台所支持的语言。在本书中，我们将全部使用新的 C#（C Sharp)语言。本书不会对 C#语言进行探讨，读者可能需要一本 C#手册或者利用.NET Framework SDK 在线文档。值得放心的是，我们不会使用到太多的 C#的语言技巧，并且 C#语言的确是一种十分简单、学习曲线十分平滑的语言。当然如果读者熟悉 C++或者 Java，那么阅读书中的 C#源代码将几乎不会有什么障碍。

当然本书也适合那些几乎没有编程经验，但又希望接触最前沿的软件开发技术的读者。本书立足于新技术的实际应用，所以尽管涉及的技术领域较广，但不会深入讨论除 ASP.NET 和 XML 以外其他相关技术的细节，只要读者有相应的入门知识，理解这些讨论将不会是困难的事。

除了 XML 编辑器，本书也不会介绍编程环境的使用。尽管我们从微软拿到了当前最新的 Visual Studio.NET Enterprise Beta 2，我们没有使用它来编写书中的代码。因为介绍这个强大的开发工具或许是另一本书要介绍的内容，并且我们认为对于入门者，最好是使用一个简单的字编辑器一行一行地编写书中的程序代码，然后使用命令行编译器来编译连接程序，在命令行和浏览器中测试代码（书中的代码都是这样写出来的）。这样读者将可以更好地了解到构建应用程序的全过程，而不会被类似于 Visual Studio.NET 这样的开发工具自动产生的代码和隐藏的编译连接过程所迷惑。至于在实际的开发工作中，我们是十分欣赏并喜欢使用 Visual Studio 的，并且建议职业的开发者使用这些工具来提高开发效率。

全书共分十二章，前两章介绍了.NET 平台的简介和建立，接着使用了三章的篇幅介绍了 XML 的语法，包括 DTD、Schema、XSLT、XPath 等和 ASP.NET 的 Web Forms、ASP.NET 的控件等技术基础。然后，本书立足于.NET 平台，介绍了 XML 的 DOM 接口和模拟的 SAX 的接口。本书的八、九两章详细介绍了 ASP.NET 的数据访问机制和 Web Service，这两章的内容比较综合地运用了前面的技术，是开发的经验之谈。本书还介绍了 ASP.NET 的设置和跟踪、ASP.NET 的安全、ASP.NET 的缓冲机制等新技术在实际开发中的应用。最后，本书通过一个较完整的、体现实际应用的电子商务子系统，希望读者全面地认识新技术和新思想带来的巨大冲击，并学习如何综合运用书中所介绍的各部分知识来设计和构建一个增值商业价值的.NET 应用程序。

本书由王超和张鹏执笔编写。其中，王超负责编写了本书的第一、四、五、八、九、十二章，张鹏负责编写了本书的第二、三、六、七、十、十一章。此外，张东、李晓、范

智育、王宏生、李光龙、王瑾、吴浩、李炎、刘伟、刘华刚、朱峰、赵晓燕、李晓、马苍、郝春容、韦勇、成美华、萧峰、李菊、张浩然、李欣、张浩、李想、朱大成、周卫、赵中伟、杨竞锐、王贵新、张诚华、朱丽云、程松、李毅、赵郡超、孙名等同志在整理材料方面给予了作者很大的帮助。

由于时间仓促，加之编者的水平有限，缺点和错误在所难免，恳请专家和广大读者不吝赐教，批评指正。

编者

声　明

本电子版不包括第 2 章内容，请参看配套图书相关章节。

目　录

第一篇

XML和ASP.NET技术框架

作为本书的第一篇，该篇内容是比较简单的。

首先第 1 章从技术的角度对.NET 框架进行必要的介绍，并将着重点放在对 ASP.NET 中 XML 应用的开发有深层影响的特征技术上。

第 2 章介绍如何建立一个 ASP.NET 的开发和运行平台，并讲述开发和运行 ASP.NET 之前所要注意的地方。

第 3 章介绍 XML 的应用精要和几个功能强大的 XML 编辑器使用介绍。在本章中，读者可以学到定义 XML 文档的数据类型、使用 XSL 定义 XML 文档的显示形式。接着本章对 XML 的模式做了一个深入的探讨，包括 DTD 和 XML Schema 机制，以及 XML DR 等相关知识。

第 4 章介绍了 ASP.NET 的 Web Forms 技术。从与老版本的 ASP 的区别的讨论中读者可以轻松地获取关于 ASP.NET 的知识，并同时领略到 ASP.NET 的优势所在，从而更好地把握技术发展的方向，并为后续的网络应用开发打下坚实的理论基础。

第 1 章　.NET 框架、ASP.NET 与 XML 简介

本章将从技术角度对.NET 框架进行必要的介绍，并将着重点放在对 ASP.NET 中 XML 应用的开发有深层影响的特征技术上，请读者仔细体会。在以后的章节中遇到一些具体技术的时候，我们也会偶尔回过头来看看这些技术得以实现的原因。这些偶尔的提示将放在“提示”中。跳过它们不会成为理解本书的障碍，但他们可以帮助读者更好的理解 ASP.NET 的本质。

按照 Microsoft 的.NET 白皮书，.NET 战略将分为三个阶段实施。本章里讨论的.NET 框架，是指微软公司的.NET 战略里所谓的.NET 的第一代。

1.1　.NET 平台简介

ASP.NET 是 Microsoft 公司.NET 框架(.NET Framework)的组成部分。要了解 ASP.NET，先要对.NET 框架有个大体的了解。

1.1.1　.NET 战略的目标

.NET 能做到什么？

- 将现有的相互隔绝的网络应用程序编织成一张真正的下一代互联网
- 使随时随地、利用任何设备获取信息成为可能和乐事
- 简化应用程序的开发、部署

.NET 是如何做到的？

- Web Services
- 新的 ADO.NET 数据库接口技术，对 XML 的深层支持
- 统一高效的.NET 运行时服务、丰富的开发发布工具、XCOPY 分发方式

1.1.2　.NET Framework 的层次结构

.NET Framework 即以前所谓的 NGWS（Next Generation Windows Services）。它的目标是成为新一代基于 Internet 的分布式计算应用开发平台。其大体结构如图 1.1 所示。

.NET Framework 包括了两个最基本的组成部分，即一般语言运行时（Runtime）环境和.NET 类库。本书将会在下面引入一个简要的介绍。另外本书还会介绍其他部分的层次关系，以便读者对这个年轻而有活力的框架有更深刻的理解。

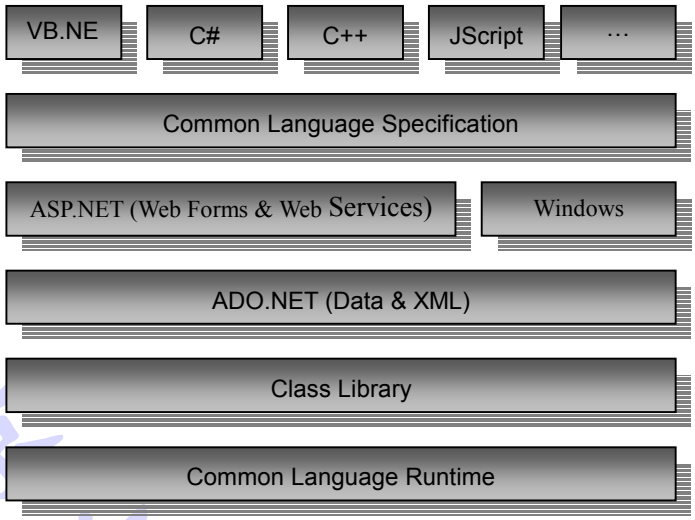


图 1.1　.NET 框架的基本层次

1.1.3　.NET Framework 的组成部分

1. 一般语言运行时（the Common Language Runtime, CLR）

正如其名称所暗示的，CLR 是.NET 框架的运行环境。该运行环境为基于.NET 平台的一切（是的，几乎一切）提供一个统一的、受控的运行环境。CLR 运行环境在.NET 平台中充当一个类似于代理人的角色，为图 1.1 中基于其上的层次提供统一的底层进程和线程管理（例如线程安全）、内存管理（例如内存垃圾回收机制）、安全管理、代码验证和编译以及其他的系统服务。CLR 通过中间语言等机制实现基于.NET 的编程语言的无关性（参见以下对 Common Language Specification 的讨论）。另外，CLR 也为.NET 框架带来潜在的平台无关性。

2. .NET 类库（.NET Framework Class Library）

.NET Framework 提供了一个包含许多高度可重用的接口、类型的类库。该类库是一个完全面向对象的类库，所以它不仅支持面向过程语言，还为面向对象语言提供了完美的支持。它既是.NET 应用软件开发的基础类库，也是.NET 平台本身的实现基础。

.NET 类库的组织是以名字空间（Name Space）为基础的。最顶层的名字空间是 System。查阅.NET 文档中的 Class Library Reference，您可以找到层次分明的各层名字空间下包含的各个不同功能的类型定义和详细使用说明。这些名字空间是以其功能模块命名的，所以可以很快地找到所需要的类。

.NET 类库包含了许多用以简化编程工作的类型，当然，该类库不是.NET 程序员可以使用的惟一一个类库，他们同样可以使用第三方厂商提供的类库。因为类库是以 Name Space 组织的，可以很容易地避免命名冲突。

3. ADO.NET

ADO.NET 为.NET 框架提供一套统一的数据访问技术。与以前的 ADO 各版本相比，

ADO.NET 主要引入了以下几个新特性：

- 对 XML 的充分支持（这将会是本书的一个主要议题）
- 新数据对象的引入（我们也会使用到这些新的对象）
- 语言无关的数据访问
- 使用和 CLR 一致的类型

4. ASP.NET 和 Windows Forms

ASP.NET 和 Windows Forms 是.NET Framework 的主要界面技术。

ASP.NET 是一个建立服务器端 Web 应用程序的框架，它是 ASP3.0 的后继版本，以前叫做 ASP+。ASP.NET 支持的界面包括 ASP Forms 和 ASP Web Service 两种形式。我们将会在本书详细地探讨这两种新型的界面技术，并体会它们带来的新的程序设计思路和模式（如果程序设计有什么模式的话）。

Windows Forms 是一项基于 Windows 平台的应用程序设计的新技术。该技术的实质是一套基于.NET Framework 的，所谓的 Rich Windows Client Library。使用这项新技术可以充分利用.NET Class Library 的面向对象特性、CLR 提供的各种服务等.NET 平台的底层支持，来开发基于 Windows 的应用程序。

Windows Forms 也可以用来开发多层结构的分布式系统的本地界面。

5. Common Language Specification（CLS）

前面提到了，Common Language Runtime 是.NET 平台的运行时环境，是.NET 的最基础部分。由于 Common Language Runtime 和 Common Language Specification 这样的设计，使得不同的语言可以进行互操作。

简单地说，在.NET 之前，一个 VB 写的类库是很难与一个 C++写的类库实现互相调用的。当然我们已经有了一些 COM 之类的机制来实现对象的互访。但现有的这些技术仍然是难以使用、并且难以适应可预见的下一代基于 Internet 的分布式计算的松散性要求的。

Common Language Specification 定义了一组运行于.NET Framework 的语言特性。CLS 和相关技术所体现的思想和当前已经实现的技术使得符合该规范的语言编写程序可以在.NET Framework 上无缝的集成。

从技术上讲，符合该规范的语言和编译器将可以把源代码编译成 CLR 所能识别的“中间语言”（Microsoft Intermediate Language，MSIL）和“元数据”（metadata）。由于使用统一的运行时和符合规范的类库（不要忘了，除了包括.NET Class Library，还可能包括其他厂商提供的类库，条件只是符合规范。在.NET SDK beta 2 中“符合 CLS 规范”称为 CLS-compliant），不同语言代码经过符合规范的编译器编译成的 IL 代码是一致的，从而是可互操作的。IL 代码和相应的 metadata 在运行时经由 CLR 的 JIT（Just In Time）编译器编译成在特定平台上运行的机器码并最终运行。

通过这样的机制，.NET 框架具有了支持几乎所有语言互操作的特性，只要为该语言实现了可以将源代码编译成 MSIL 代码的编译器，都可以用于.NET 平台的开发。在本书撰稿时，.NET SDK 已经提供了 C#、Managed Extensions for C++（一个 C++的针对.NET 的扩展，

使之可以和.NET Framework 集成起来）、VB.NET 的编译器。在不久的将来，会有其他语言的.NET 编译器出现（例如 Cobol、Perl），使得各种语言的程序员可以更方便地为.NET 平台开发组件和应用程序。在本书编写时，已经有 APL、COBOL、Component Pascal、Eiffel、Perl、Python、SmallTalk、Standard ML 等语言的.NET 开发工具发布。

另外，这种机制也带来了类似于 Java 的平台无关性。当然这种平台无关性是“潜在（potential）”的。.NET 会以 Windows 为目标运行平台。但仅从技术上讲，只要在除 Windows 之外的平台上实现了 CLR 和 Class Library，就可以使.NET 和基于.NET 的应用程序在该平台上运行。这种思路从市场角度看也是有可能的。在编写本书的时候，著名的开放源码 GNU 已经启动了一个名为 Mono 的计划，该计划将会为其他非 Windows 平台构建一个与.NET 在很大程度上兼容的平台无关的框架和运行环境，以免.NET 这种伟大的设计思想被垄断在少数公司手里。

6. 其他重要技术（如 CTS、CAS、线程技术）

.NET 体系中采用的带有革命性、创新性的技术还远不止前面提到的那些。前面仅仅介绍了对本书以后的讨论有较多影响的技术。

.NET 体系中还包含了许多其他关键技术，例如 CTS（Common Type System，CLS 的超集）、CAS（Code Access Security）。这些技术和上面提到的技术互相配合，构成了现在可以看到的.NET 框架。请注意，这些技术并不是相互分离的。

另外，贯穿.NET 体系的 XML 和相关技术（例如 SOAP），将在本章的 1.3.3 节和以后的篇章里详细介绍。在.NET 体系中，大量的开放标准和技术被采用，并渗透到整个体系结构的底部，这是.NET 战略的一个明显特征。XML 语言及相关技术被.NET 体系的设计者运用到几乎所有组成部分，包括：

- 程序语言层面上的 C#中可以使用 XML 来注释代码，并可指定特定的编译器开关自动来生成描述代码的 XML 文档
- 数据访问层面中，ADO.NET 技术中的关系型数据与 XML 层次型数据模式深度结合
- 应用配置层面中 XCOPY 分发方式和应用配置均使用 XML 文档来简化配置和避免臭名昭著的“DLL 陷阱”（DLL Hell）
- 提供完全符合 W3C DOM Core Level 1 和 DOM Level 2 Core 规范的 DOM 对象模型，并对此进行扩充来进一步简化开发者的工作
- 提供类似于 W3C SAX 规范的 XmlReader/XmlWriter 等对象模型
- 提供一系列 XSLT 相关的类来支持 XML XSLT/XPath 技术

等等。

.NET 还使用标准的 SOAP（Simple Object Access Protocol）协议、HTTP 协议来实现开发和使用 Web Services 的能力。基于这些开放标准的 Web Services 技术可以和任意平台的 Web Services 相集成。

.NET 体系不仅采用大量开放的工业标准，还将走上一条标准化的道路。当前而言，Microsoft 公司已将 C#语言规范草案提交到 ECMA 组织，同时被提交到该组织的还有.NET 的 Common Language Infrastucture。

1.1.4　.NET Platform 的运行

正如你已经预感到的一样，.NET 平台将带来全新的开发者和用户体验。但.NET 平台作为一个平台而非操作系统，仍需要一个特定的体系结构来支持其运行。

一开始.NET 将被定位于 Windows 操作系统上。这里所说的.NET Platform 是指运行 ASP.NET、XML Web Service、Windows Forms、ADO.NET 的平台，通常来讲是服务器端和运行新一代 Windows 操作系统的台式电脑。

但是请不要忘了，.NET 平台的目标是将互联网上相互隔离的网络应用程序（称为“信息孤岛”）连结起来，并使随时随地的信息获取成为可能。因此调用构建在.NET 平台上的 XML Web Service 的调用者程序或系统并不一定需要运行于 Windows 操作系统的服务器上。只需要遵循 XML Web Service 的规范（使用 SOAP 协议和 XML 语言来传输数据和其他信息），就可以获取其他平台提供的 Web Service。另外，新的移动电话、掌上电脑、个人数字助理（PDA）、车载电脑、信息家电等智能设备（当然还有很多我们想象不到的未来新设备）都可能是.NET 的客户端系统。这些客户端系统也不一定需要运行.NET Platform。

在本书撰稿时，.NET Platform 仍处于 Beta 2 阶段。微软公司发布了多个软件包来供开发人员和商务用户开发、测试和运行.NET 应用程序。这包括：

- Microsoft .NET Framework Redistributable Package Beta 2（简称.NET Framework Redist）
- Microsoft ASP.NET Premium Edition Beta 2
- Microsoft .NET Framework Software Development Kit (SDK) Beta 2
- Microsoft Visual Studio.NET Beta 2

Microsoft .NET Framework Redistributable Package 提供了运行基于.NET Framework 的应用程序所需要的一切，包括：.NET Common Language Runtime、.NET Framework Class Library、ASP.NET。

.NET Framework Redistributable Package 运行在下列操作系统上：Microsoft Windows XP、Windows 2000、Windows NT 4.0、Windows 98、Windows Millennium Edition（Windows Me）。另外要求安装 Internet Explorer 5.01 或更新版本。如果作为服务器端安装，要求安装 Microsoft Data Access Components 2.6，推荐安装 2.7 版本。

Microsoft ASP.NET Premium Edition 提供了开发、部署和运行 ASP.NET 应用程序所需要的一切，包括：Common Language Runtime、.NET Framework Class Library、ADO.NET、命令行的 C#、VB.NET、Jscript.NET 编译器和 ASP.NET 的核心支持模块。

ASP.NET Premium Edition 运行在操作系统 Windows 2000 或者 Windows XP（Professtional 版或 Server 版）上，其他要求和.NET Framework Redist 相同。

Microsoft .NET Framework SDK 提供了编写、建立、测试、部署基于.NET Framework 的应用程序所需要的一切，除了包含.NET Framework Redist 外，还有文档、实用工具、编译器和示范代码。

.NET Framework SDK 运行在 Windows XP、Windows 2000、Windows NT 4.0 上，要求安装 Internet Explorer 5.01 或更新版本。要求安装 Microsoft Data Access Components 2.6，推

荐安装 2.7 版本。

Visual Studio.NET 是快速开发企业级 Web 应用程序和高性能桌面应用程序的工具。Visual Stuido.NET 包括了基于组件的开发工具，例如 Visual C#、Visual Basic、Visual C++以及许多其它用来简化团队设计、开发和解决方案部署的技术。Visual Studio.NET 支持.NET Framework、ASP.NET，并包括了 MSDN Library 知识库。

Visual Studio.NET Beta 2 中包含了.NET Framework SDK Beta 2，不必另外安装。

作为商业应用的.NET Platform 将和新版本的 Windows 操作系统以及配套的 Microsoft .NET Enterprise Servers 服务器软件产品紧密结合在一起。截止到本书撰写的阶段，微软公司已经将 Windows 2000 Server 的下一个版本由 2001 年 4 月命名的 Whistler Server 改名为 Windows.NET Server。这预示着该服务器操作系统将包含.NET 运行平台。

其他的 Microsoft .NET Enterprise Servers 服务器软件产品包括：

- BizTalk Server 2000
- Exchange 2000 Server
- SQL Server 2000
- Intergration, Security and Acceleration Servers，其中包括：
  - Host Integration Server 2000
  - Internet Security and Acceleration Server 2000
- Application, Services and Commerce Servers，其中包括：
  - Application Center 2000
  - Commerce Server 2000
  - Mobile Information Server 2001
  - SharePoint Portal Server 2001

1.2　ASP.NET

1.2.1　什么是 ASP.NET

ASP.NET 是 ASP（Active Server Pages）的后继版本（在先前的文档中被称为 ASP+）。ASP.NET 和它的前期版本都是构建新一代动态网站和基于网络（特别是 Internet）的分布式应用的技术。ASP.NET 为网站设计人员和网络程序员提供了更简单快捷的开发方法。

ASP.NET 向前兼容 ASP，运行在.NET Platform 上。以前的 ASP 脚本几乎不经修改就可以在.NET Platform 上运行，从而保护了先前的投资。

注意：关于如何修改老版本的 ASP 代码，使它们和新的 ASP.NET 代码在您的基于.NET Platform 的 Web 服务器上共存（从而使向.NET 平台的过渡更加平滑），请参考.NET SDK 中关于 ASP.NET 的文档。本书将不会涉及该话题。

1.2.2　ASP.NET 的新特性

1. 新的语言特性

当前，ASP.NET 仅支持完全面向对象的 Visual Basic、C#和 JScript。VBScript 已经不被支持。ASP.NET 是完全基于组件的，所有的页面、COM 对象乃至 HTML 元素都可以视为对象。

提示：请回忆 1.1 节中关于语言无关性和互操作性的介绍。体会这种设计为程序员带来的好处。

2. 运行于.NET Platform 上

ASP.NET 中将可以使用.NET Platform 提供的各种运行环境服务，如丰富的类库、数据访问、自动内存管理等等，从而可以大大提高开发效率。

3. 更好的性能

ASP.NET 代码不再是解释型代码，可以经由 JIT 编译器编译后运行，并且引入了页面缓冲机制。

此外，ASP.NET 还具有以下特性：更简单易行的部署；更好的伸缩性和可用性；更先进的安全性能；更多更深层的特性……

1.3　可扩展标记语言 XML

1.3.1　XML 的出现

eXtensible Markup Language(可扩展标记语言)，又简称为 XML，是针对网络应用的一项新技术。World Wide Web Consortium(W3C)目前通过了 XML 标准 1.0（第二版）。XML 也是标记语言的一种。标记语言还包括 SGML、HTML 等很多语言。所谓“置标”，是指在数据中加入标记来说明信息的方法。标记语言是运用置标法描述结构化数据的形式语言。这种语言较早的成功应用范例有 SGML（Standard Generalized Markup Language，标准通用标记语言）。XML 就是 SGML 的一个子集。

Jon Bosak 和万维网联盟 W3C 的专家将 SGML 浓缩成为新的置标语言——XML。XML 不仅继承了 SGML 的优点，并在此基础上去掉了 SGML 的模糊、不重要的部分，使之变得简单、易学、好懂。还有，XML 继承了 HTML 的部分优点，使得 XML 易于理解和推广。

在 1997 年，XLL（可扩展链接语言）的草案出现。同年，Microsoft 公司第一个运用 XML 技术定义 CDF（Channel Definition Format）。这样，在经历了从理论到应用的发展和自我完善的过程，XML 终于在 1998 年 2 月被 W3C 承认为国际标准，成为世界通用的置标语言，并确定了版本号为 1.0。

SGML 本是 IBM 公司为解决公司内部大量文档的交换和存储而采取的解决办法。在 SGML 诞生之初被称为 GML(Generalized Markup Language)语言，经过改进和发展，在 1986

年被国际标准化组织公布为国际标准——ISO8879。SGML 十分复杂、昂贵，而且几个主要的浏览器厂商都明确表态拒绝 SGML，这两大缺陷使得 SGML 在网络上的普及十分困难。所以 SGML 多用在科技文献和政府文件中，这种场合对数据的分类和数据索引要求很高，正好体现了 SGML 的一大长处，同时 SGML 还具有良好的扩展性。

下面来看看 XML 和 HTML 的区别：

HTML 是置标语言的又一成功例子。HTML，HyperText Markup Language 即超文本标记语言，是由 CERN 设计的。HTML 针对 SGML 的缺点具有简单、廉价的特点，并且广为接受成为网络上的流行语言。但是 HTML 的缺点日益突出：重表现而轻数据、随着发展变得越来越复杂、效率低下，而且 HTML 的描述能力也不能满足当前的需求。HTML 对于数据的搜索几乎不支持，因为 HTML 的标记几乎不含任何数据信息。当前网络的搜索引擎无法智能地处理 HTML。例如，如果您要寻找一首叫“Hotel California”的歌曲，您输入了“Hotel California”。可能找到的是“Hotel”的地址列表、“California”的地方介绍，还可能有一个叫“Hotel California”的旅馆介绍，最后才是您要找的 Eagles 乐队的歌“Hotel California”的信息。但是如果有一个为地名、旅馆和歌曲制订的 DTD，用来说明标记的含义。这样，查找名叫“Hotel California”的歌就能得到更为精确和有用的结果了。一个好的代理能智能地解释信息，并作出响应。如果发送给代理的数据是用 XML 组织的数据，那代理能更容易地理解数据的含义以及与其已知数据间的联系。

HTML 的语法相当松散，随着 HTML 越来越复杂，使 HTML 代码的出错率很高，降低了代码的执行效率。XML 的语法有着严格的规定，同时又是可扩展的，于是 XML 的结构不会变得错综复杂。

网络的发展和网络技术的成熟，使得网络在信息交换方面的作用越来越重要。为了实现交流，我们需要通用的、标准的技术。比如 IP 协议、HTTP 协议等，还有上面提到的 HTML，这些都是可以被各方承认支持的技术，这些技术是可以不受操作平台限制的，是可以跨平台使用的。网络在商业上的应用也是极具潜力的，比如在电子商务中的应用。网络越来越成为 business to business 和 organization to organization 的信息交流支持平台。XML 简化了基于网络的 B2B 事务。XML 的最强之处在于它能进行数据交换。由于不同的组织间(甚至同一组织的不同部分)很少会统一于同一套工具，在不同组之间的通讯就会涉及大量的工作。这些数据的格式可能会是多种多样的，为了实现数据的共享和交换，需要公认的技术来支持这种数据交流。XML 也使 B2B 通讯更为简单。最后，XML 使得在网络上发送信息十分简单并保证了在格式转换过程中不会丢失任何信息。

上面讲到，HTML 不能胜任这一要求，尽管 HTML 已经是通用技术了。HTML 的数据描述能力实在有限，对于数据的意义 HTML 没办法加以表现。但是，在数据内容的显示和浏览器支持方面，HTML 还是不可替代的。

XML 出现以后，又出现了使用 XML 技术的 XHTML、SVG、SMIL 等置标语言。正是 XML 具有 SGML 的强大功能和 HTML 的简单易懂的优点，使得它在出现不久就被各方广为看好，这也使得 XML 的推广普及也非常迅速。比如，最早使用 XML 语言开发产品的 Microsoft 已将 XML 标准作为未来的标准，其产品也将全面支持 XML。本书重点介绍 XML 在 ASP.NET 开发平台的应用。

1.3.2　XML 的特点

在 XML1.0 标准中说明了 XML 的指定目标：

- XML 应该可以在互联网上直接使用
- XML 应该支持各种不同的应用方式
- XML 应该与 SGML 兼容
- 处理 XML 文档的应用程序应该容易编写
- XML 文档的可选择性数量应该减到最小，最好减至没有
- XML 文档应该具有良好的可读性，并且比较清晰
- 用 XML 设计的新的置标语言应该方便快捷
- XML 设计的置标语言应该真实、简洁
- XML 文档应容易编制
- XML 标记的简洁性并不重要

以上目标体现了人们对 XML 的期望和 XML 本身的优点。这些目标决定了 XML 的前途是光明的，XML 的发展方向符合计算机技术的前进方向。尽管因为 XML 出现时间不长，相应的编辑浏览工具不成熟，而且应用有限，但随着时间的推移，XML 将成为网络开发和数据库的重要技术。上述目标还指明了 XML 的特点。

XML 在商业上的应用前景也是因为 XML 满足了当前商务数据交换的需求：

- 数据可以跨平台使用并可以被人阅读理解
- 数据的内容和结构有明确的定义
- 数据之间的关系得以强化
- 数据的内容和数据的表现形式分离
- 使用的结构是开放的，可扩展的

要满足这些要求，XML 的设计者确定了 XML 的特点。

一般来说，XML 具有内容和形式分离，良好的扩展性、良好的跨平台移植性和良好的自描述性等特点。

1. 内容与形式分离

在 HTML 中，数据内容和表现形式是混在一起的，这样当改变数据的表现形式时，更新文档的工作量很可观。同时，对于 XML 文档而言，标记是包含信息的，比如关键字、继承关系等，这些信息对于数据的检索、描述起着巨大的简化作用。当只想改变数据的表现形式时，我们只需修改从 XML 文档中分离出的用于数据表现的样式单就可以了。

2. 良好的可扩展性

XML 允许程序员制定自己的标记集，满足自己的需要。同样，一个行业或某一特定人群也可以制定在自己范围内的通用标记集。这样，XML 可以轻松地适应每一个领域而无需对语言本身作大修改。另外，要说明的是 XML 的数据定义也是与数据本身分离，独立存在的。这样，使 XML 的标记集不致日益扩大，对于有特殊要求的人，他们可以选用需要

的标记集。现在比较典型的例子是 CML（化学置标语言）和 MathML（数学置标语言）。这两种语言都是利用了 XML 的扩展性这一特点。

3. 良好的移植性

XML 语言可以定义各种数据，像文本、图像、声音等。这些数据往往有很多种不同的格式，使得数据不能在各系统之间交流，或使用额外的转换软件来实现跨平台的交流。XML 的这个特性使得只要交换数据的系统都能处理一种格式的文件，即 XML 文档，就能处理由 XML 标注的各种数据，从而实现了不同格式数据的跨平台交换。

4. 良好的自描述性

良好的自描述性使得 XML 数据可以被不同的应用程序分析处理。并且 XML 的自描述性可以使一篇 XML 文档被人理解。通过标记、元素之间的关系，数据要表达的内容就会一清二楚了。标记<姓名>Tom</姓名>当然是说有个“名字”是“Tom”，数据是“Tom”，数据代表的是一个由标记表述的信息“名字”。

对于一份完整的 XML 文档而言包括三个部分：

数据部分：XML 的核心内容。数据部分通常是与数据库紧密联系的。现在，数据在网页中的作用越来越大，这就要求有更好的处理数据的方法。可仅有数据是不够的，XML 的处理程序还不能识别这些数据，所以我们还需要为这些数据提供标记说明，将信息通过标记和数据传给处理程序，使数据得以正确使用。

标记说明：这部分将数据的信息通过一定的格式和说明传递给 XML 的处理程序。对于数据本身，标记没有任何作用，标记只是体现数据的特性和数据间的关系。说明标记的常用方法是使用 DTD（文档类型定义），近来出现了另一种方式 Schema。Schema 具有超过 DTD 的优势，可以预计 Schema 将取代 DTD，成为新的 XML 数据类型定义方式。

数据表现：对于网页而言，最终结果是将数据表现给用户，样式单规定了数据的表现形式。样式单目前有 CSS（层叠样式单）和 XLT（可扩展样式单）两种。同一数据可以有不同的表现形式，用于表现同一数据的样式单可以是多种多样的。

XML 数据可以基于 HTTP 协议传输，在服务器和浏览终端之间传递。服务器可以将 XML 数据文档根据样式单进行处理，得到可浏览的 HTML 文档后，将 HTML 文档传给终端。这样对于服务器的要求比较高，加重了服务器的负担。服务器还可以将 XML 数据文档和样式单一起传到终端，在终端生成可浏览的 HTML 文档。这样做的好处是：服务器只需将相同的 XML 数据文档分发到各个浏览终端，然后由样式单去生成不同的浏览文档。这样数据流量比传送 HTML 文档要小，而且对服务器的要求也降低了。

还可以使用一种称为数据岛（Data Island）的方式在 HTML 文档中绑定 XML 数据。利用 HTML 文档的 script 处理这些 XML 数据。这些客户端的脚本将 XML 数据部分转换为 HTML 文档。其形式有两种：内联、外部引用。内联形式为：

```
<XML ID="XMLData">
<?xml version="1.0"?>
……
</XML>
```

省略号代表的是 XML 的内容。外部引用形式为：

```
<XML ID="XMLData" SRC="Xmldata.xml">
</XML>
```

外部 XML 文档 Xmldata.xml 由属性 SRC 指定。可以是文件名，也可以是 URL。关于数据岛再看一个例子，在 HTML 中是如何显示 XML 数据的。

```
<XML ID="XMLData">
<?xml version="1.0" ?>
<record>
    <name>Bill Smith</name>
    <sex>male</sex>
    <phone>12345678</phone>
</record>
</XML>

<title>example</title>
<font color=blue>
<H2>This is an example.</H2>
</font>

<TABLE border=2 bordercolor=oliver DATASRC="#XMLData">

    <TR>
       <TD><SPAN DATAFLD="name"></TD>
       <TD><SPAN DATAFLD="sex"></TD>
       <TD><SPAN DATAFLD="phone"></TD>
    </TR>
</TABLE>
```

在 XML 中定义了 ID，在表格中为属性“DATASRC”赋值“#ID”实现对 XML 部分的调用。在每个单元格中，使用“DATAFLD”属性指定单元格内容。更为复杂的对 HTML 文档中 XML 数据岛的处理可以通过 XML Data Source Object（DSO）对象来实现。在 IE 中，也是使用 DSO 来解析和处理 XML 数据岛的。上面例子的效果见图 1.2。

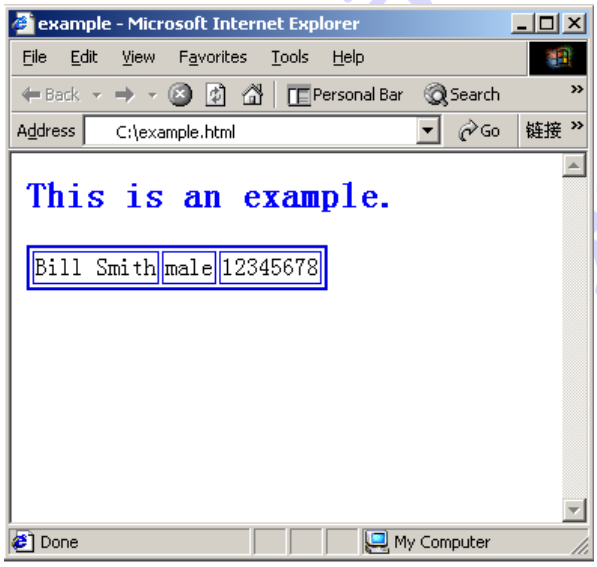


图 1.2　程序运行效果图

图中表格的数据来自于 XML 数据部分。

在以后的章节中，我们会为您逐一介绍这些内容的。这其中包括 XML 的语法、DTD、Schema、CSS、XLT 以及 XPath 等知识。

1.3.3　XML 与 ASP.NET 的联系

XML 本身只是一种标记语言，就像 HTML 一样。但是，XML 在数据描述方面远胜于 HTML。可以说，XML 拥有描述所有已知和未知数据的能力。这是因为 XML 的扩展性非常好，可以为新的数据类型制定新的数据描述规则，作为对标记集的扩展。

在 ASP.NET 中的应用主要是体现在数据访问和 Web 应用程序方面。

ASP.NET 的数据来源主要是 SQL Server 和 XML 资源。对于数据的操作，ASP.NET 是通过 ADO.NET 对象完成的。您可以参考本书的第八章，在那里我们会详细介绍 ADO.NET。XML 除了可以作为数据资源的格式以外，在.NET Framework 中，还有更为重要的作用。这就是作为服务器和客户端之间的信息交换语言。

使用 XML 作为信息交换的语言。XML 文档通过 API 接口动态地被应用程序生成，然后由指定的解析器解析文档，将得到的信息转给其他的应用程序来达到信息交换的目的。

SOAP 的出现正是适应这种使用 XML 格式传输数据的要求。SOAP 本身是一种 XML 格式的文档。正如在图 1.3 中 XML 的位置是网络中的信息传递员，它为 Web Forms、B2B 等和数据库之间建立沟通。

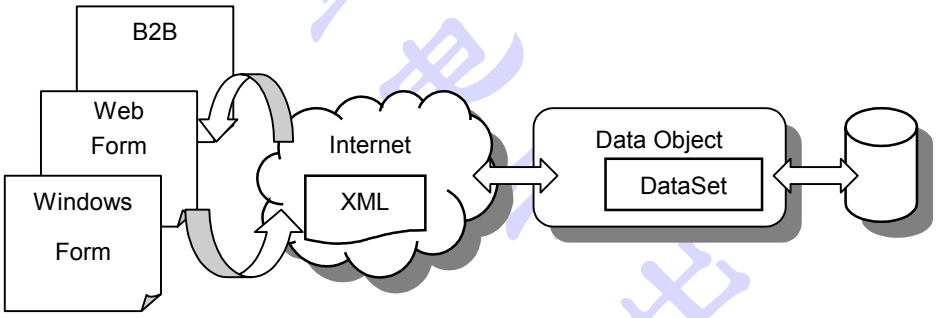


图 1.3　XML 在网络中的应用

在.NET Framework 中，可以使用 XmlDocument 类实现 DOM 接口。但是，对于 SAX 接口，.NET 使用 XmlReader 和 XmlWriter 两个类获得类似于 SAX 接口的数据访问方式。这两个类提供的数据访问方式和 SAX 一样，是基于流的。和 SAX 不同的是这两个类使用的是“拉”数据模式，而 SAX 使用的是“推”数据模式。

.NET Framework 支持 XPath，这在 DOM 和 XSLT 中十分有用。我们可以使用 XPath 精确定位需要的信息。

1.4　本章小结

本章介绍的是一些基础知识，主要包括书中最基本的两项技术：ASP.NET 和 XML。其中重点是.NET 战略、ASP.NET 和 XML 的基本概念。

在后面的章节中，我们将为您详细介绍这些激动人心的新技术。

第 2 章　ASP.NET 开发和运行平台的实现

声　明

本电子版不包括本章内容，请看配套图书相关章节。

北京希望电子出版社
2001

第 3 章　XML 的应用概要

本章将介绍 XML 的基本知识。在本章中，你将会了解 XML 语法的概要，如果你学习过 HTML 语法，这对你是有帮助的。为帮助你编写格式复杂的 XML 文档，我们会向你推荐几个功能强大的 XML 编辑器，工欲善其事，必先利其器。一个好的编辑器可以提高你的工作效率，缩短开发周期。在本章中，还可以学到如何定义 XML 文档的数据类型。其中我们会为你介绍两种方法：使用 DTD 或是 Schema，这里我们采用的是 XSD。我们还将为你介绍 XML 文档的表现形式——样式单（CSS 和 XSL）。在介绍 XSL 时，我们会着重介绍如何使用 XSLT 转换 XML 文档成为其他类型的文档，比如 HTML 文档，或者转换为另一个 XML 文档。本章中还会详细介绍在 XSLT 和 DOM 中十分重要的 XPath，XML 路径匹配。通过本章的学习，你掌握如何编写 XML 文档的知识。

3.1　XML 的基本语法

3.1.1　语法的基本要求和概念

与 HTML 不同，XML 的语法有着严格的要求。对于 XML 文档有两层要求，格式良好的（well-formed）和有效的（valid）。其中第一层，也是最基本要求是：格式良好的（well-formed）。只有当一篇 XML 文档是格式良好的时候，才能正确地分析和处理它。“格式良好的”这一标准是相对 HTML 语法的混乱而提出的，它大大提高了 XML 处理程序、处理 XML 数据的准确性和效率。一篇格式良好的文档同样易于人类阅读和修改。

所谓格式良好的要求包括：

（1）确定且惟一的根元素

在一篇 XML 文档中，有且只有一个根元素。并且这个根元素是必须的，因为文档中除了根元素以外所有的元素都必须是由根元素派生而来，是根元素的子元素或派生元素。

（2）开始标记和结束标记匹配

元素的标记要成对出现，是说每一个起始标记都要有相应的结束标记。要注意的是，标记是区分大小写的。比如说：<Student>和</student>是不匹配的，</student>不是<Student>的结束标记。要改为使用匹配的<student></student>标记或<Student></Student>标记。

（3）正确的元素标记嵌套

（4）属性值要用引号括起

为属性赋值时必须将属性值用引号括起。可以使用单引号或是双引号，但在文档中引号类型要一致。

（5）同一个元素的属性不能重复

对于重复数据，你要使用重复元素，而不是为同一个元素赋相同的属性。举个例子来说，一个人的亲属可以有很多，每一个亲属都要用元素来表示，作为元素“人”的子元素。

而不是为元素“人”赋相同的属性“亲属”。

格式良好的文档易于浏览器浏览，省去了很多不必要的显示错误，还简化了浏览器的设计。XML 的处理程序也要求 XML 文档格式良好。如果文档不是格式良好的，处理程序将报告说有“致命错误”并且不会对文档做出任何处理。

文档满足格式良好的要求后，会对文档进行有效性确认。有效性是通过对 DTD 或是 Schema 的分析判断的。在本章后面的部分会讲到这些关于 DTD 和 Schema 的规定。

3.1.2　建立符合要求的 XML 文档

XML 文档的基本组成有

- XML 的声明
- 处理指令 PI
- XML 元素

下面通过对一个比较完整的 XML 文档的解说，具体说明 XML 语法。

程序清单 3-1：example-3-1.xml

```
<!--以下是 XML 文档声明-->
<? xml version= "1.0" encoding ="GB2312" standalone= "no"?>
<? xml-stylesheet type="text/xsl" href="mystyle.xsl"?>

<!--书单的示范-->
<书单>
    <样书>
    <作者>Smith</作者>
    <作者介绍>He has written several books. He is a popular writer.
</作者介绍>
    <!--使用 CDATA 说明书的内容-->
    <章节概要>
        <! [CDATA [
            <第一章>
                <简介>第一章描绘了故事的背景。</简介>
            </第一章>
            <第二章>
                <简介>第二章主人公出场，情节开始展开。</简介>
            </第二章>
        ]]>
        </章节概要>
    </样书>
< /书单>
```

1. XML 的声明

```
<? xml version= "1.0" encoding ="GB2312" standalone= "no"?>
```

XML 标准要求声明须放在文档的第一行。声明也是处理指令的一种。声明一般由“<?”开始，至“?>”结束，声明的常用属性和其赋值由表 3-1 给出。

表 3-1　XML 声明

属性名	常用值	作用解释
version	1.0	声明中必须包括此属性，而且必须放在第一位。它指定了本文档所采用的 XML 版本号。现在 XML 的最新版本为 1.0 版，所以 version 的值均为 1.0
encoding	GB2312	文档使用的字符集为简体中文
	BIG5	文档使用的字符集为繁体中文
	UTF-8	文档使用的字符集为压缩的 Unicode 编码
	UTF-16	文档使用的字符集为压缩的 UCS 编码
standalone	yes	本文档是独立文档，没有 DTD（Document Type Definition）文档与之配套
	no	表示可能有 DTD 文档为本文档进行置标声明

还有其他的处理指令为处理 XML 的应用程序提供处理 XML 文档的信息，如示例程序的第二行。

2. 处理指令

```
<? xml-stylesheet type="text/xsl" href="mystyle.xsl"?>
```

处理指令 PI 为处理 XML 的应用程序提供信息。应用程序则根据这些信息处理文档。处理指令的格式为：

```
<? 处理指令名 处理指令信息 ?>
```

例子中的处理指令指明与本文档配套的样式单的类型（XSL）和文件名（mystyle.xsl）。

3. 元素

元素是 XML 文档的核心，格式如下：

```
<标记>数据内容</标记>
```

在本例中，“书单”、“样书”、“作者”等都是元素。元素中可以嵌套其他元素。如“样书”中包括了“作者”、“作者介绍”等元素。

XML 语法规定每个 XML 文档都要包含至少一个根元素。如例子中根元素就是“书单”。根标记必须是非空标记，包括整个文档的数据内容。一般根元素名可与 DTD 声明相配。

注意：如果 XML 文档中有多个根元素，这样的文档是不能被解析的。但对于数据的描述，多根元素的 XML 文档有时是有效的。

（1）数据内容：位于标记之间，可由任何合法 Unicode 字符组成。但不能包含标记开始符“<”。标记中的任何字符都是数据内容，包括换行符。比如：

```
<作者>Smith</作者>
```

和下面标记之间的数据

```
<作者>
Smith
</作者>
```

是不同的。后者的数据内容比前者多了两个换行符。

数据内容可以为空。其格式可以改写为：

```
<元素名/>
```

另外，如在数据内容中用到特殊字符，比如“<”，就需要采用实体引用的方法。具体见 1.1.3 节实体和实体引用。

（2）标记：标记基本格式为

```
起始标记：  <标记名  (属性名 = “属性值”) >
结束标记：  </标记名>
```

括号中的属性部分可以省略。标记名可由字母、数字、下划线“_”、冒号“:”、句号“。”和连字符“-”组成，但不能由数字、句号、连字符开始，中间不能出现空格和“xml”组合（Xml、XML、xmL 等）。属性名的命名规则和标记名相似，属性名中还可以出现空格、标点、实体引用等。

除了上面的元素，还可以通过辅助的内容来完善 XML 文档，这些辅助内容包括注释和 CDATA 两部分。

4. 注释 <!--this is one of his books-->

就像其他程序语言一样，XML 也有自己的注释方式：

```
<! -- 注释的内容 -- >
```

对于注释的内容 XML 的处理器不作任何处理。但为保证文档的格式良好，注释中不能含“-”、“--”，以防和注释结尾“-->”混淆。注释可以位于文档的任何位置，除了标记、实体声明中，当然也不能在声明前面，因为文档第一行必须是 XML 声明。

5. CDATA

格式为：

```
<! [CDATA [
内容
]]>
```

CDATA 标记中的内容 XML 处理器会看成字符数据，像注释一样，忽略 CDATA 标记内容中的标记、实体引用。但内容中不得含有“]]>”，以防和 CDATA 的结束标志混淆。忽略内容中的标记，这一点有时很有用。比如例子中要将一段 XML 代码作为元素“章节概要”的内容，根据 XML 语法，我们只能写成：

```
<章节概要>
    &lt;第一章&gt;
    &lt;简介&gt;第一章描绘了故事的背景。&lt;/简介&gt;
    &lt/;第一章&gt;
    &lt;第二章&gt;
    &lt 简介&gt;第二章主人公出场，情节开始展开。&lt/简介&gt;
    &lt;/第二章&gt;
</章节概要>
```

如果采用 CDATA 方式则要采用实体引用的方式写成：

```
<章节概要>
<! [CDATA [
    <第一章>
        <简介>第一章描绘了故事的背景。 </简介>
    </第一章>
    <第二章>
        <简介>第二章主人公出场，情节开始展开。</简介>
```

```
        </第二章>
    ]]>
    </章节概要>
```

显而易见，第一种写法明显难于阅读。使用 CDATA 标记则简化了文档的内容，使文档易于理解。在这里，我们用“<”、“>”表示“<”和“>”。这是实体引用的用法。实体引的介绍详见下文。

3.1.3　实体引用和实体

实体是 XML 文档中的一种别名。比如：我们可以给一段文本内容一个别名，然后在需要这段文本内容时，使用别名来代替。为内容增加别名的过程，就是声明实体的过程。然后在指定的位置使用别名来代替内容，就是使用实体引用。

你可以将实体看成一种替换，可以是文本替换，也可以是如图像、声音这样的外部数据替换。使用实体和实体引用可以简化我们的数据输入工作，还可以在 XML 文档中使用 XML 解析器不能解析的、类似于声音的、数据内容。

实体的使用包括声明和使用两个方面：

1. 实体声明

实体声明一般在 DTD 中，可以是外部 DTD，也可以是内部 DTD。比如下面是内部 DTD 定义根元素的形式：

```
<? xml version= "1.0" encoding ="GB2312" standalone= "no"?>
<!DOCTYPE 根元素名[
    实体声明
]>

<根元素名>
    //内容
</根元素名>
```

声明来自外部数据的实体，比如一张图片的格式为：

```
<!ENTITY Billphoto SYSTEM "Bill.jpg" NDATA JPG>
```

声明一个包含文本内容的实体的方法可以参考 DTD 中关于 ENTITY 类型的元素的说明：

```
<!ENTITY Bill
"生于 1972-2-11，喜爱体育和音乐，现在是市场部经理助理">
```

这里我们使用了 DTD 中声明实体的关键字 ENTITY 和 SYSTEM，下面我们如何在 XML 文档中引用这个别名为“Bill”的实体呢？

2. 实体引用

实体引用允许在元素内容和属性值中插入任何字符串。我们引用一个名为“name”的实体，格式为：

```
&name;
```

现在让我们在一篇 XML 文档中引用前面定义的实体“Bill”

```
<人员简介>
    <成员>
        <姓名>Bill</姓名>
        <个人说明>&Bill;</个人说明>
```

```
        </成员>
    </人员简介>
```

这样引用的效果和下面的写法是相同的：

```
    <人员简介>
        <成员>
            <姓名>Bill</姓名>
            <个人说明>生于 1972-2-11，喜爱体育和音乐，
            现在是市场部经理助理</个人说明>
        </成员>
    </人员简介>
```

提示：像个人简介或是地址这样的需要在多个 XML 文档中出现的内容，可以采用示例中的实体定义和引用的方法来处理，这样在数据更新时可以只需修改实体定义一处，即可完成多个位置的数据更新。

在 XML 标准中，有五个实体是标准提供的，用来代替特殊的符号，比如“<”和“>”，以防止使用这些符号时与标记、属性值中的符号混淆。就像前面的“<章节概要>”例子的情形。在标记的内容中，出现了“<”和“>”符号，这是 XML 语法所不允许的，因为这会导致歧义的发生。要在标记的数据内容中使用这些符号，就一定要采用实体引用的方法。这些特殊符号的实体引用由表 3-2 给出。

表 3-2　特殊的实体引用

特殊字符	实体引用
‘	'
“	"
&	&
<	<
>	>

上面已经介绍了实体引用的形式，但要注意除 XML 本身提供的实体外，文档中引用的每个实体都要声明过。实体可引用其他实体。但不能循环引用，比如 A 引用 B，然后 B 引用 C，C 再引用 A。

XML 具有良好的可读性和自描述性，可以描述信息的内部结构。对于文本信息，XML 分析器可以分析、处理它们。它们被称为已分析实体，前面提到的实体都是已分析实体。而非文本数据，则在文档中不加分析地引用，从而实现在 XML 文档中使用多种格式的信息、资源。这些实体称为未分析实体。

未分析实体可以是图像、视频、音频等。XML 使用未分析实体有两种方法：直接引用和为一个元素定义实体类型的属性，采用未分析实体作为属性值。

在介绍 DTD 时我们再讨论如何定义未分析实体。

3.1.4　名字空间（Namespace）

考虑下面的情形：

在 A 公司中，有雇员信息

```
<employee>
    <name>Bill</name>
    <positon>manager</postion>
    <payment>$300</payment>
</employee>
```

在 B 公司也有雇员信息

```
<employee>
    <name>Jim</name>
    <positon>clerk</postion>
    <phone>12345678</phone>
</employee>
```

雇员信息的标记是相同的，但是标记的内容（子元素）却是不同的。这样的情形是可能出现的，为了区别这些信息，使得在两个公司之间的信息交换不会产生歧义或误解，我们是为标记增加前缀来区分这些标记，这种前缀是名子空间前缀。其形式是：

```
<名字空间前缀:元素名 xmlns:名字空间前缀="名字空间的全称">
```

名字空间的全称是用来区分信息的标记，可以是任何字符串，要保证名字空间的全称所确定的前缀可以区分这些信息，换句话说，这些名字空间的字符串不能重复。这样才能保证这些加前缀的标记属于不同的名字空间。

下面使用名字空间区分上面的例子中重复的标记。

```
<record>
    <acom:employee xmlns:acom="http://www.Acompany.com">
        <acom:name>Jim</ acom:name >
        <acom:positon>manager</acom:postion>
        <acom:payment>$300</acom:payment>
</acom:employee>
    <bcom:employee xmlns:bcom="http://www.Bcompany.com">
        <bcom:name>Bill</bcom:name>
        <bcom:positon>clerk</bcom:postion>
        <bcom:phone>12345678</bcom:phone>
</bcom:employee>
</record>
```

“xmlns”属性是 XML 文档的名字空间属性。名字空间的全称为了防止重复，一般使用相应的 URI（Universal Record Indicator），比如 URL。通过指定前缀“acom-”，“bcom-”可以实现对标记名相同的“employee”的区分。使得在信息交换或文档中，不会因为标记名的相同造成麻烦。

事实上，名字空间前缀在省略时，即采用默认的名字空间前缀时，要保证指定名字空间的元素和其包含的所有后继元素都在指定的名字空间内。

名字空间还被用在其他类型的文档中。对于标记名相同的问题，为它们指定不同的名字空间可以实现标记的识别。但是在下面要讲到的 DTD 文档中，是不支持名字空间的。这也是 DTD 相对于 Schema 的一个劣势。

3.2　文档类型定义

3.2.1　什么是文档类型定义——DTD

当编写了有效、格式良好的 XML 文档后，你会发现用到了很多新标记，他们是你自

己定义的。XML 是允许程序员使用自己的标记的，但问题是如何使其他程序员和用户理解你的标记词汇表呢？

在 XML1.0 标准中，我们用一种规范——文档类型定义（Document Type Definition）解决这个问题。在 DTD 中你可以向其他人或 XML 的语法分析器精确解释你标记集中每一个字的含义。这也要求你要保证所有你使用的词汇表规则都在 DTD 中，否则 XML 解析器无法根据 DTD 验证文档的有效性。DTD 同样规定了关于你使用的新词汇的语法，这一点对于 XML 文档的分析是十分重要的。

DTD 根据其出现的位置可分为内部 DTD 和外部 DTD 两种。内部 DTD 是指 DTD 和相应的 XML 文档都在同一个文档中，具体格式下面将给出。如果你的 DTD 需要在不同的 XML 文档中共享，或者为了将来更好、更方便地更新文档数据，我们推荐你使用外部 DTD 文档。外部 DTD 文档是在 XML 文档之外，另创建文件名为“*.dtd”的文档。这样，你的 DTD 就可以实现多个 XML 文档共享了。同样，对于一个行业或领域而言，使用一个本行业、本领域公认的 DTD 标准无疑会使专业数据内容的表达变得异常简单，还容易被同行理解。你也不必再为专业术语的表达大伤脑筋了。

提示：想想还记得怎样在 XML 文档中区分内部 DTD 和外部 DTD 吗？在 XML 文档声明的属性 standalone 表达了这一信息。standalone 的值为 yes，说明没有相应的外部 DTD 文档存在。如果值为 no，则可能存在这样一个文档。因为即使“standalone=no”，也可能没有外部 DTD 存在。

下面，我们将分别采用内部 DTD 和外部 DTD 两种方式完成 DTD 定义，并在 3.2.2 节具体介绍如何书写 DTD 定义。

1. 内部 DTD

内部 DTD 出现在 XML 的开始部分。内部 DTD 与 XML 在同一篇文档中。

程序清单 3-2：example-3-2.xml

```
<?xml version="1.0" encoding="GB2312" standalone="yes"?>
<!DOCTYPE 学生档案 [
    <!ELEMENT 学生档案 ANY>
    <!ELEMENT 学生 (姓名, 性别, 年龄, 来源, 班级)>
    <!ELEMENT 姓名 (#PCDATA)>
    <!ELEMENT 性别 (#PCDATA)>
    <!ELEMENT 年龄 (#PCDATA)>
    <!ELEMENT 来源 (省份, 城市)>
    <!ELEMENT 省份 (#PCDATA)>
    <!ELEMENT 城市 (#PCDATA)>
    <!ELEMENT 班级 (#PCDATA)>
]>
<?xml-stylesheet type="text/xsl" href="mystyle.xsl" ?>

<学生档案>
    <学生>
        <姓名> 张岳 </姓名>
        <性别> 男 </性别>
```

```
        <年龄> 20 </年龄>
        <来源>
            <省份> 江苏 </省份>
            <城市> 南京 </城市>
        </来源>
        <班级> 一班 </班级>
    </学生>

    </学生档案>
```

由示例可以看出内部 DTD 的定义形式为：

```
<?xml version="1.0" encoding="GB2312" standalone="yes"?>
<!DOCTYPE 根元素名[
    元素定义
]>

//XML 文档部分
```

在实际的应用中，通常是编写许多 XML 文档，但是这些 XML 文档可以共享同一个 DTD 定义。无疑，如果我们会将 DTD 部分取出放在单独的文件中（*.dtd 文件），这样我们的工作将得到简化，修改 DTD 也变得容易。

提示：我们推荐使用外部 DTD，这样 XML 的优点将得到体现，数据不仅与表现形式分离，而且与定义分离。这对将来的数据维护将起到极大的简化作用。事实上，很多行业组织都是将自己制定的外部 DTD 作为数据交换格式的行业规范。

2. 外部 DTD

我们改写上面的例子，你可以比较它们的异同点。

程序清单 3-3：example-3-3.dtd

```
<?xml version="1.0" encoding="GB2312"?>
    <!ELEMENT 学生档案 (学生)*>
    <!ELEMENT 学生 (姓名, 性别, 年龄, 来源, 班级)>
    <!ELEMENT 姓名 (#PCDATA)>
    <!ELEMENT 性别 (#PCDATA)>
    <!ELEMENT 年龄 (#PCDATA)>
    <!ELEMENT 来源 (省份, 城市)>
    <!ELEMENT 省份 (#PCDATA)>
    <!ELEMENT 城市 (#PCDATA)>
    <!ELEMENT 班级 (#PCDATA)>
```

程序清单 3-4：example-3-3.xml

```
<?xml version="1.0" encoding="GB2312" standalone="no"?>
<!DOCTYPE 学生档案 SYSTEM "example-3-3.dtd">
<?xml-stylesheet type="text/xsl" href="mystyle.xsl" ?>

<学生档案>
    <学生>
        <姓名> 张岳 </姓名>
        <性别> 男 </性别>
        <年龄> 20 </年龄>
        <来源>
```

```
            <省份> 江苏 </省份>
            <城市> 南京 </城市>
        </来源>
        <班级> 一班 </班级>
    </学生>

    </学生档案>
```

你注意到外部 DTD 与内部 DTD 的区别了吗？

是的，在 xml 部分除了将 standalone 的值变为“no”以外，还多加了一行：

```
<!DOCTYPE 学生档案 SYSTEM "example-3-3.dtd">
```

与内部 DTD 类似，DOCTYPE 说明根元素的属性。其中 SYSTEM 属性指明了外部 DTD 的位置，可以是绝对路径也可以是相对路径。下面两例都是合法的。

使用相对路径为：

```
<!DOCTYPE 学生档案 SYSTEM "example-3-3.dtd">
```

使用绝对路径为：

```
<!DOCTYPE 学生档案 SYSTEM "http://••••••/example-3-3.dtd">
```

如果采用外部 DTD 的话，就需要有两个文档，第一个文档就是关于 DTD 的文档，第二个文档就是遵守 DTD 格式的 XML 文档。实际上我们可以建立无穷多个遵守该 DTD 格式的 XML 文档。举一个例子来说，我们在构造关系数据库中表的时候，需要定义好表的结构（也就是表包含的字段集合），然后就可以往这个表中放入记录，记录的个数从理论上讲可以是无穷多个。这里关于表的结构就类似于 DTD 文档。记录类似于遵守 DTD 格式的内容文档。

外部 DTD 的好处是：它可以方便高效地被多个 XML 文件所共享。你只要写一个 DTD 文件，就可以被多个 XML 文件所引用。这样做不仅简化了输入工作，还保证当你需要对 DTD 做出改动时，不用一一去改每个引用了它的 XML 文件，只要改一个公用的 DTD 文件就足够了。不过需要注意，如果 DTD 的改动不是“向后兼容”的，这时原先根据该 DTD 编写的那些 XML 文件可能就会出问题了。

如果你使用的是上面所讲的已成为行业规范的通用的 DTD，那你要使用 PUBLIC 代替 DOCTYPE 格式如下：

```
<!DOCTYPE 根元素名 PUBLIC "DTD 文档名" "DTD 文档的 URL">
```

在 DTD 中，规定了 XML 元素的声明定义的规则，让我们来逐一介绍如何声明不同的 XML 元素。

3.2.2　元素的定义

在这一节中，我们将采用表格的方式来简化说明，突出重点。可以阅读这些表格，对照上面的范例理解其中的内容。

表 3-3 给出了 DTD 中的关键字说明。

表 3-3　DTD 的部分关键字

关键字	说明	使用举例
ANY	说明数据既可是纯文本也可谓子元素，多用来修饰根元素	<!ELEMENT 学生 ANY>
ATTLIST	定义元素的属性	<!ATTLIST 元素名 属性名 类型 缺省值>
DOCTYPE	描述根元素	<!DOCTYPE 学生档案 [……]>
ELEMENT	描述所有子元素	<!ELEMENT 学生档案>
EMPTY	空元素	<!ELEMENT 姓名 EMPTY>
SYSTEM	表示使用外部 DTD 文档	<!DOCTYPE 学生档案 SYSTEM example-3-3.dtd">
#FIXED	ATTLIST 定义的属性的值是固定的	<!ATTLIST 学生 学号 ID #FIXED>
#IMPLIED	ATTLIST 定义的属性不是必须赋值的	<!ATTLIST 学生 学号 ID #IMPLIED>
#PCDATA	数据为纯文本	<!ELEMENT 姓名 (#PCDATA)>
#REQUIRED	ATTLIST 定义的属性必须赋值	<!ATTLIST 学生 学号 ID #REQUIRED>
INCLUDE	表示包括的内容有效，类似与条件编译	
IGNORE	与 INCLUDE 相应，表示包括的内容无效	

当元素具有不确定性时，我们采用表达式来约束数据内容。

表 3-4 给出了表达式的说明。A、B、C 在这里代表元素名。

表 3-4　DTD 定义表达式

表达式	说明
A+	A 至少出现一次
A*	A 可以不出现，也可以出现多次
A?	A 出现一次或不出现均可
(A B C)	ABC 中间是空格，ABC 无顺序排列
(A,B,C)	ABC 中间是逗号，ABC 有顺序排列
A|B	A OR B，A 逻辑或 B

使用这些表达式，可以表达很多意思。比如：

```
<!ELEMENT 学生 (姓名，联系电话)+>
<!ELEMENT 姓名(#PCDATA)>
<!ELEMENT 联系电话(#PCDATA)>
可以定义如下的 XML 文档:
<学生>
        <姓名> 张岳 </姓名>
        <联系电话>12345678</联系电话>
</学生>
或是:
<学生>
        <姓名> 张岳 </姓名>
        <联系电话>12345678</联系电话>
```

```
        <姓名> 李文 </姓名>
        <联系电话>87654321</联系电话>
    </学生>
```

如果定义成：

```
    <!ELEMENT 学生 (姓名, 联系电话+) >
    <!ELEMENT 姓名(#PCDATA)>
    <!ELEMENT 联系电话(#PCDATA)>
```

则相应的 XML 文档要改为：

```
    <学生>
        <姓名> 张岳 </姓名>
        <联系电话>12345678</联系电话>
        <联系电话>23456789</联系电话>
        <联系电话>34567890</联系电话>
    </学生>
```

我们再举一个例子：

DTD 部分为“<!ELEMENT 学生 (姓名, 性别, 年龄)>”，那么它所定义的 XML 部分——姓名、性别、年龄的顺序是固定的，不能改变。但要是 DTD 写成“<!ELEMENT 学生 (姓名 性别 年龄)>”，用空格代替“,”，那么改变顺序是合法的。

对于元素的类型取值有 ID、IDREF、IDREFS、CDATA、Enumerated、ENTITIES、ENTITY、NMTOKEN、NMTOKENS、NOTATION 等几种。让我们看看它们的作用。

1. ID

顾名思义，ID 就是每个元素的编号或用来区别于其它元素的标记。元素的 ID 要具有惟一性，一个 ID 只能对应一个元素。

```
    <!DOCTYPE 学生档案 [
    <!ELEMENT 学生档案 ANY>
    <!ELEMENT 学生 (姓名, 性别)>
    <!ELEMENT 姓名 (#PCDATA)>
    <!ELEMENT 性别 (#PCDATA)>
    !ATTLIST 学生 学号 ID #REQUIRED>
        ])
```

XML 部分为：

```
    <学生档案>
        <学生 学号＝”001”>
            <姓名> 张岳 </姓名>
            <性别> 男 </性别>
            <年龄> 20 </年龄>
            </学生>

        <学生 学号＝”002”>
            <姓名> 李文 </姓名>
            <性别> 女 </性别>
            <年龄> 20 </年龄>
        </学生>
    </学生档案>
```

2. IDREF/IDREFS

你如果需要使用已有的 ID 值表示一个元素的双重或多重身份，你不必定义新元素，可以使用 IDREF/IDREFS 类型元素引用已有的 ID 值。

3. CDATA

CDATA 类型的属性，属性值是文本内容。比如：

```
<!ATTLIST 学生 学号 CDATA #REQUIRED>
<学生 学号="001" ></学生>
```

4. Enumerate

枚举类型指该类型的属性值是一系列可选的值。

```
<!ATTLIST 学生 性别 (男|女) #REQUIRED>
```

这样，学生的性别属性就只能有“男”和“女”两种取值了。尽管，事实也只能如此。

```
<学生 性别= "男"></学生>
```

也可以用引号标出缺省值。

```
<!ATTLIST 学生 性别 (男|女) "男" #REQUIRED>
```

这样，当没有为“学生”元素的“性别”属性指定取值时，“性别”属性自动会取指定的默认值——“男”。

5. ENTITY/ENTITIES

该类型为存在的实体取别名。

```
<!ENTITY 实体名 "实体内容">
```

或外部实体

```
<!ENTITY 实体名 SYSTEM "实体文件名">
```

6. NMTOKEN/ NMTOKENS

该类型指向一个 XML 名称，换句话说，属性值是一个 XML 名称。其中，NMTOKENS 类型的属性值可以是多个 XML 名称。因为 XML 名称的命名规则和 Java 的语言标识符的命名规则相同，所以可以使用 NMTOKENS 类型的属性来访问 Java 类。

7. NOTATION

对于非文本的数据，像音频、视频、图像等，作为未分析实体我们可以给它们定义自己的语法，然后向 XML 的处理程序指定外部处理程序处理这些未分析实体。

8. DTD 文档中的注释

你可以像编写 XML 文档一样在定义中加入注释,注释的格式与 XML 文档相同：

```
<!-- 注释内容 -->
```

实体还可以包含参数，含参数的实体只能用在 DTD 文档中，在 XML 文档中不能使用带参数的实体。声明带参数实体的格式是：

```
%实体名
```

比如说：

```
<!ENTITY %信息"姓名|性别|年龄">
<!ENTITY 学生 (%信息;|班级)>
```

```
<!ENTITY 教师 (%信息;|职称)>
```

最后，在 DTD 中还有类似 C 语言等程序设计语言中的条件编译的定义方式。这需要使用关键字 INCLUDE/IGNORE。其格式为：

```
<!INCLUDE/IGNORE[
定义
]>
```

include 表示“[]”中的内容有效,而 ignore 表示“[]”的内容可以省略。

3.3　新的文档描述方式——Schema

3.3.1　Schema 的产生

DTD 对于文档结构的描述是很出色的，但是 DTD 使用不同于 XML 的独立的语法规则，而且 DTD 不支持数据类型。在 DTD 中，只有“PCDATA”一种类型的数据。但是，在应用中，往往需要表达复杂的数据类型，像布尔型、时间、日期等。

DTD 的标记集是固定的，用户不能扩充标记。DTD 的新标记集只有通过新的 DTD 标准来定义。换句话说，DTD 不是开放的和可扩充的。Schema 则具有开放的和可扩充的特性。下面会讲到。

DTD 使用的是自己的语法结构，使用与 XML 不同的符号和标记，用户必须学习新的语法规则，使用新的标记和字符。这些语法和符号有时是复杂的。XML 强大的自描述性使得 XML 自己可以表示自己，于是有了 Schema 的出现。

新的 XML 描述方法——Schema 完善了 DTD 的不足。Schema 本身就是一种 XML 的应用形式。所以，使用 XML 的语法和标记，适用 XML 的编辑器和解析器。Schema 对于文档的结构、数据的属性、类型的描述是全面的。Schema 还是 DTD 的一种扩展和补充。对于数据类型的定义弥补了 DTD 存在的缺陷。还有 Schema 的出现解决了 DOM 和 SAX 无法应用于 DTD 的问题，因为 Schema 就是 XML 文档，DOM 和 SAX 当然可以实现对 XML 文档的访问了。作为新的文档描述方式，Schema 不仅弥补了 DTD 的不足，同时还具有 DTD 的优势，这使得可以预见 Schema 将会替代 DTD 成为 XML 新的描述语言。

尽管 Schema 现在还没有统一的国际标准，只有 W3C 在 2001 年 5 月 2 日公布的最终推荐标准；并且现在 Schema 的编辑、浏览工具还不成熟，但随着 Schema 的发展和 Microsoft 公司的支持，Schema 的普遍应用只是时间问题。

目前，Schema 的最终推荐标准包括两部分：第一部分 World Wide Web Consortium（W3C）XML Schema Part1：Structures 通过对 XML Schema Language（XSD）的说明，使用 XSD 定义了 XML 的数据类型和数据结构。XSD 定义的元素、属性和数据类型都是符合 W3C XML Schema Part1 标准的。第二部分 W3C XML Schema Part2：DataTypes 详细说明了 Schema 的数据类型的定义。

本书中使用的标准是 W3C 在 2001 年 3 月 30 日通过的第一部分的推荐标准和 2001 年 4 月 8 日通过的第二部分的推荐标准。所以本书中的 Schema 的名字空间都是：

```
http://www.w3.org/2001/XMLSchema
```

我们已经介绍过了 DTD 的语法，所以下面我们将对照 DTD 的定义方式，通过一个完整示例来解说 Schema。一方面使你获得 Schema 对于 DTD 的优势，另一方面，目前 DTD 仍是主要的文档描述方式，在学习 Schema 的同时复习一下学过的 DTD 知识。

先给出一篇 XML 数据文档，然后分别用 DTD 和 Schema 来定义文档的数据类型。

注意：文档 example-3-7.xml 是使用外部 DTD 作为类型定义文档的。要使用 Schema 需将“<!DOCTYPE orders SYSTEM "example-3-7.dtd">”一行去掉，并为元素“orders”增加属性说明 Schema 的名字空间和所使用的 XSD 文档的位置，可改写为："<orders xmlns:xsi="http://www.w3.org/2001/XMLSchema" " xsi:noNamespaceSchemaLocation="example-3-7.xsd">"

程序清单 3-5：example-3-5.xml

```
<?xml version="1.0"?>
<!DOCTYPE orders SYSTEM "example-3-7.dtd">
<orders>
    <order data="07-2001-4">
        <product>Book</product>
        <customer>Tom</customer>
        <address>
            <city>city of A</city>
            <street>No. 2351 of Centre Street </street>
        </address>
        <telephone>12345678</telephone>
        <remark>be quick, please.</remark>
    </order>
    <order data="07-2001-11">
        <product>Picture</product>
        <customer>Sala</customer>
        <address>
            <city>city of B</city>
            <street>No.0274 of East Street</street>
        </address>
        <telephone>23456789</telephone>
    </order>
</orders>
```

程序清单 3-6：example-3-5.dtd

```
<?xml version="1.0"?>
<!ELEMENT orders (order)*>
<!ELEMENT order (product, customer, address, telephone, remark*)>
<!ELEMENT address (city, street)>
<!ELEMENT product (#PCDATA)>
<!ELEMENT customer (#PCDATA)>
<!ELEMENT telephone (#PCDATA)>
<!ELEMENT city (#PCDATA)>
<!ELEMENT street (#PCDATA)>
<!ELEMENT remark (#PCDATA)*>
<!ATTLIST order date CDATA  #REQUIRED>
```

我们用 Schema 来改写上面的 DTD 文档，作为相同的 XML 文档的类型定义文档。

程序清单 3-7：example-3-5.xsd

```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2000/10/XMLSchema"
```

```
elementFormDefault="qualified">
        <xsd:element name="orders" type="OrdersType"/>
        <xsd:complexType name="OrdersType">
            <xsd:sequence maxOccurs="unbounded">
                <xsd:element name="order" type="OrderType"/>
            </xsd:sequence>
        </xsd:complexType>
            <xsd:complexType name="OrderType">
            <xsd:sequence>
                <xsd:element name="product" type="xsd:string"/>
                <xsd:element name="customer" type="xsd:string"/>
                <xsd:element name="address" type="addressType"/>
                <xsd:element name="telephone" type="xsd:decimal"/>
                <xsd:element name="remark" type ="xsd:string"
minOccurs="0"/>
            </xsd:sequence>
            <xsd:attribute name="date" type="xsd:string"
use="required"/>
            </xsd:complexType>
        <xsd:complexType name ="addressType">
            <xsd:sequence>
                <xsd:element name ="city" type="xsd:string"/>
                <xsd:element name="street" type="xsd:string"/>
            </xsd:sequence>
        </xsd:complexType>
    </xsd:schema>
```

通过比较，你应该看出在像实例这样简单的 XML 文档的类型定义文档，使用 DTD 是比较简洁的，Schema 表述相对复杂。但是，Schema 也有其强于 DTD 之处，我们看到元素“order”的属性“date”是代表日期的，Schema 为日期提供了 date 类的数据，可以更精确的表示日期时间。这也显示了 Schema 对于 XML 文档的各种数据类型支持得比 DTD 更好，在 DTD 中日期只好不加区分地声明为纯文本，CDATA 类型。

3.3.2　Schema 的语法

注意：Schema 的标准并不统一，本书介绍的 Schema 的标准是微软公司的 XSD。我们介绍的 XSD 是以 Visual Studio 7.0 Beta 2 标准为主的。

Schema 的定义方式是由 XML Schema Definition Language（XSD）作为描述语言对 XML 文档的标记进行说明的。XSD 的元素说明如下：

表 3-5　Schema 元素列表

元素名	说明
all	允许各种元素类型出现
any	允许出现同一名字空间内的各种类型元素，“namespace”指明使用的名字空间，“processContents”说明 XML 处理程序对文档元素不在指定名字空间内时的处理方式
anyAttribute	为“complexType”元素指定属性
annotation	注释的定义
appinfo	定义在“annotation”元素中的处理程序需要信息
attribute	声明属性

（续表）

元素名	说明
attributeGroup	为“complexType”声明一组属性。只能作为”schema”元素的子元素。“name”属性是必须的。组成员可以是“anyAttribute”、“attribute”和其他“attributeGroup”元素
choice	选取属性组中的一个属性元素
complexContent	复杂类型的定义。包括元素和注释两种类型
complexType	复杂元素的定义。包括元素名和相应的属性
documentation	定义“annotation”元素为使用者提供的信息
element	元素声明。支持的属性很多，后面我们会提到
extension	扩展“complexContent”、“simpleContent”两种元素的内容
field	声明运用 XPath 路径表达式的取值
group	一组元素的声明
import	声明 Schema 使用的名字空间
include	在当前的 Schema 使用的名字空间下包含指定的 Schema 文档
key	在一定的范围内指定元素或属性的值（一组值）为关键字
keyref	指定与“key”、“unique”元素相应的属性或元素的取值
list	定义一系列指定类型的“simpleType”元素
notation	符号的定义
redefine	允许外部 Schema 文档在当前 Schema 文档中对类型的定义
restriction(XSD)	定义对“simpleType”、“simpleContent”、“complexContent”这几种元素的限制
schema	Schema 的定义
selector	定义供 XPath 表达式选取的一组“unique”、“key”、“keyref”类型的元素
sequence	说明组中的元素具有顺序要求
simpleContent	内容可以是另一个 simpleType 类型数据或是对 complextype 类型数据加以限制或扩展后的内容
simpleType	简单类型，要求元素和属性的值必须为纯文本数据
union	从指定的简单数据类型（simple data type）中定义“simpletype”元素的集合
unique	说明在特定的代码段中，属性或元素的值是惟一的

1. 元素

对于元素的声明“element”，有很多附加属性。其基本格式为：

<element	
abstract = "布尔值"	为真时，元素不出现，取而代之的是相应的替代
block=#all 或 “ substitution, extension, restriction”的一种	说明禁止元素继承自替换方式(substitution)、扩展方式(extension)、约束方式(restriction)或全部方式
default = "string"	元素值为文本，相当于 DTD 中的“#PCDATA“标记
final	取值作用和“block”相似，但没有替换方式。指定元素 element 的默认值

（续表）

fixed = "string"	元素的值是确定的文本或简单类型值。“fixed”和“default”两种属性不可同时出现
form = "qualified | unqualified"	决定元素的属性是否使用名字空间的前缀。“qualified”是使用前缀
id	元素代号
maxOccurs	元素的最大出现次数值取非负整数或“unbounded”(次数不限，很多)
minOccurs	元素的最小出现次数，值取大于等于 0 的整数
name	元素名称
nullable = "布尔值"	布尔值为真，则元素可以有空属性(null attribute)
ref	元素类型在指定的 Schema 中的名称
substitutionGroup	可被替换的元素组
type	Schema 中定义的数据类型名称。“type”和“ref”
>	

下面我们定义一个简单的元素：

```
<xsd:element name="书名" type="xsd:string" id="001"/>
```

定义了一个名为“书名”的元素，其类型为字符串，即文本型，代号为 001。

2. 属性

在 XSD 中对于属性的定义形式为：

<attribute	
form = "qualified | unqualified"	取值类似元素的定义，判断是否使用名字空间的前缀
id	属性的代号
name	属性名
ref	属性使用的指定的 schema
type	使用当前 schema 的类型
use = "{ default | fixed | optional | prohibited | required }"	属性值的规定，取值依次为默认值、固定值、可选值、禁止使用和必须出现
value = "string"	属性值
>	
</attribute>	

定义元素和元素的属性是对 XML 文档描述的基本内容。

3. 替换

关于替换“ref”我们看一个例子

```
<xsd:element name="书" type="string"/>
<xsd:element name="报刊" type="string"/>
<xsd:element name="周刊" type="string"
```

```
    substitutionGroup="报刊" />
<xsd:element name="日刊" type="string"
    substitutionGroup ="报刊" />

<xsd:element name="出版物">
  <xsd:complexType>
    <xsd:choice minOccurs="1" maxOccurs="1">
      <xsd:element ref="周刊"/>
      <xsd:element ref="日刊"/>
    </xsd:choice>
  </xsd:complexType>
</xsd:element>
```

“周刊”和“日刊”是“报刊”的仅有的可替换元素，且必须替换其中一种。

4. simpleType 和 complexType

除了内置的数据类型，像 string，decimal 等，XSD 还允许用户定义新的数据类型。这是通过 simpleType 类型和 complexType 类型来实现的。

simpleType 数据类型只能是文本值，不能含有元素或是属性。定义元素或属性时，可以将类型声明为是 simpleType 类型。

simpleType 类型可以继承自另一个 simpleType 或是其他的内置数据类型。定义 simpleType 有三种方法：

（1）restriction：限制 simpleType 的取值范围一定是继承到的值的子集。比如：继承了 integer 类型，那么取值只能是 integer 类型，也就是整数。这个例子可以写成：

```
<xsd:simpleType name="salary">
    <xsd:restriction base="xsd:integer">
        <xsd:maxInclusive value="500"/>
        <xsd:maxInclusive value="990"/>
     </xsd:restriction>
</xsd:simpleType>
```

“minInclusive，maxInclusive”的值由于继承自整数（integer）类型，所以必须是类似于 500 这样的整数。

（2）list：定义的 simpleType 类型包含了一个用空白字符间隔的一组值。这组值是继承到的 simpleType 的值。比如：

```
<xsd:simpleType name="Dates">
  <xsd:list itemType="date">
</xsd:simpleType>
```

定义“Dates”为 simpleType 类型，值是“date”的列表。

（3）union：定义的 simpleType 类型的取值是继承到的多个 simpleType 的值的并集。

```
<xsd: attribute name="pizzasize">
  <xsd:simpleType>
    <xsd:union>
      <xsd:simpleType>
        <xsd:restriction base="weight"/>
      </xsd:simpleType>
      <xsd:simpleType>
        <xsd:restriction base="size"/>
      </xsd:simpleType>
    </xsd:union>
```

```
    </xsd:simpleType>
  </xsd:attribute>

  //属性 weight 的类型取值说明
  <xsd:simpleType name="weight">
    <xsd:restriction base="xsd:positiveInteger">
      <xsd:enumeration="300"/>
      <xsd:enumeration="500"/>
      <xsd:enumeration="800"/>
    </xsd:restriction>
  </xsd:simpleType>

  //属性 size 的类型取值说明
  <xsd:simpleType name="size">
    <xsd:restriction base="xsd:string">
      <xsd:enumeration value="small"/>
      <xsd:enumeration value="medium"/>
      <xsd:enumeration value="large"/>
    </xsd:restriction>
  </xsd:simpleType>
```

我们定义了属性“pizzasize”是 simpleType，其属性值是两个 simpleType 的值的并集。每个 simpleType 类型的取值都是枚举值。比如 size 可以是 small、medium 或是 large 中的一个，也只能是其中一个，因为使用了 restriction 类型元素。

再来看看 complexType。既然称为复杂类型，那么 complexType 肯定比 simpleType 更加复杂。事实也正是如此，在 complex 类型中，可以将元素和属性的集合作为内容，当然元素的文本内容也是允许的。

在定义元素时，最常用的就是 complexType 类型了。complexType 类型可以定义元素的元素名、元素的内容和子元素的结构。要使用 complexType 类型，需将 type 属性的值设为 complexType。当然使用 simpleType 类型时，就设为 simpleType。

在 complexType 类型中，可以含有以下几种元素中的一种元素来决定 complexType 类型中内容的类型，并且最多只能包含一种元素。

simpleContent	complexType 中只有文本字符数据或是 simpleType 类型数据，没有任何的子元素定义。
complexContent	在 complexType 类型中，只能含有元素作为内容。
group	complexType 类型的内容是引用的组中的元素。
sequence	complexType 类型中的内容是指定的队列中的元素。
choice	在 complexType 类型中，只允许出现由 choice 元素指定的元素中的元素。
all	在 complexType 类型允许所有的由 all 元素指定的元素出现一次。

如果指定 group、sequence、choice 或 all 元素为子元素，那么 complexType 类型中的属性可以选择声明为下面的元素：

attribute	complexType 含有指定的属性。
attributeGroup	complexType 中的属性是来自引用的 attributeGroup 中的。
anyAttribute	complexType 类型中可以含有指定的名字空间中的所有属性。

在 complexType 中，含有 attribute 元素或是 attributeGroup 元素时可以不限制它们的数量，但是，只能包含一个 anyAttribute 元素的实例。

来看个使用 complexType 类型的例子。比如有 XML 语句如下：

```
<book id="0001">BookA</book>
```

这种常见的 XML 元素的 Schema 定义可以写成：

```
<xsd:element name="book" type="bookType"/>
<xsd:complexType name="bookType">
 <xsd:extension base="xsd: string">
  <xsd:attribute name='id" type="xsd: decimal"/>
 </xsd:extension>
</xsd:complexType>
```

为 complexType 类型的元素“book”声明了内容类型（string）及属性类型（decimal）。上面的写法是使用 simpleType 声明元素“book”的类型，然后再使用 complexType 类型来定义前面用到的 simpleType 类型“bookType”。我们还可以使用匿名声明的 complexType 来实现对上面的 XML 元素的定义。

```
<xsd:element name="book">
 <xsd:complexType>
  <xsd:extension base="xsd: string ">
   <xsd:attribute name="id" type=" decimal" />
  </xsd:extension>
 </xsd:complexType>
</xsd:element>
```

simpleType 和 complexType 两种类型在 Schema 中非常重要，尤其是 complexType 类型。希望你能熟练地使用这两种类型定义所需的新的数据类型，这会非常有助于 Schema 文档精确地表达 XML 数据文档包含的数据类型和数据结构信息。

5. simpleContent 和 complexContent

simpleContent 的内容可以是扩展的或限制的 complexType 类型元素的内容，也可以是另一个 simpleType 类的元素。simpleContent 类型元素的内容必须是定义成 restriction 元素或是 extension 元素。restriction 元素是将元素的取值范围限制在 simpleType 继承到的子集中。比如说：

```
<xsd:element name="book">
  <xsd:complexType>
    <xsd:simpleContent>
      <xsd:extension base="xsd:string">
        <xsd:attribute name="genera">
          <xsd:restriction base="xsd:string">
            <xsd:enumeration value="science"/>
            <xsd:enumeration value="novel"/>
            <xsd:enumeration value="history"/>
          </xsd:restriction>
        </xsd:attribute>
      </xsd:extension>
    </xsd:simpleContent>
  </xsd:complexType>
</xsd:element>
```

定义了元素“book”，其属性“genera”的取值限于“science、novel、history”三种。如果写成

```
<xsd:element name="book" type="xsd:string"/>
```

就不能为 book 元素增加属性了。

所以 extension 元素通过增加属性可以扩展元素的内容。再比如说：

```
<xsd:element name="root">
  <xsd:complexType>
    <xsd:simpleContent>
      <xsd:extension base="xsd:string">
        <xsd:attribute name="relation" type="xsd:string" />
      </xsd:extension>
    </xsd:simpleContent>
  </xsd:complexType>
</xsd:element>
```

定义了一个元素“root”，元素的内容是一个 string 类型的数据，并且“root”元素有一个名为“relation”的属性。

要说明的是，前面两个例子中的属性“base”指定了要继承的 simpleType 的全名。属性“base”的值可以是内置数据类型或是指定的 simpleType 元素。

complexContent 的内容可以是 complexType 类型元素的扩展或是限制。complexType 类型元素可以在包含元素的同时夹杂文本内容。和 simpleContent 类似，complexContent 类型元素的内容也必须是定义成 restriction 元素或是 extension 元素。

```
<xsd:complexType name="undergraduate">
  <xsd:sequence>
   <xsd:element name="name"   type="string"/>
   <xsd:element name="specialty" type="string"/>
   <xsd:element name="guider"   type="string"/>
  </xsd:sequence>
 </xsd:complexType>

 <xsd:complexType name="graduate student">
  <xsd:complexContent>
   <xsd:extension base="undergraduate">
    <xsd:sequence>
     <xsd:element name="laboratory" type="string"/>
    </xsd:sequence>
   </xsd:extension>
  </xsd:complexContent>
 </xsd:complexType>
```

元素“graduate student”是在继承了“undergraduate”元素基础上，加入了新的子元素“laboratory”。这里“graduate student”和“undergraduate”都是复杂类型（complexType）元素。

前面介绍了 Schema 标准的第一部分：如何使用 XSD 定义数据类型和数据结构。在 W3C XML Schema Part2 中，将数据类型定义为内置原始类型、继承类型和 facet。下面我们来看看这三种类型的说明。

先来看看 Schema 中的数据类型之间的关系。图 3.1 给出了 Schema 各类型之间的继承关系。在第二部分标准中，非常详细的定义了各种 Schema 可能用到的数据类型，将数据类型分为内置原始类型、继承类型和范围三种。

限于篇幅，我们将为你介绍第二部分定义的众多数据类型中的一部分类型，常见的简单的类型只是简要地说明。

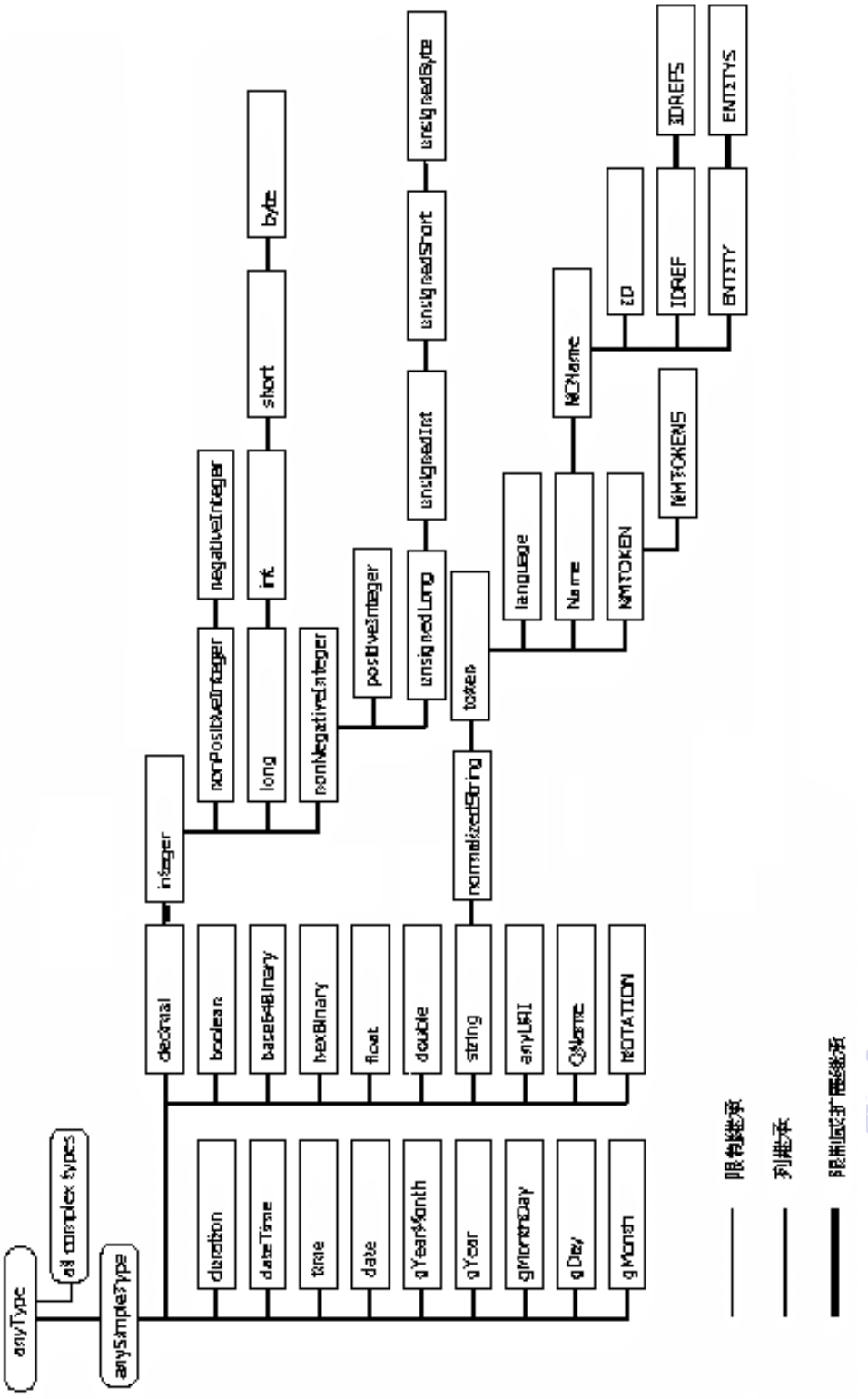

anyType
all complex types
anySimpleType
decimal
boolean
base64Binary
hexBinary
float
double
string
anyURI
QName
NOTATION
duration
dateTime
time
date
gYearMonth
gYear
gMonthDay
gDay
gMonth
integer
nonPositiveInteger
negativeInteger
long
int
short
byte
nonNegativeInteger
positiveInteger
unsignedLong
unsignedInt
unsignedShort
unsignedByte
token
normalizedString
language
Name
NMTOKEN
NMTOKENS
NCName
ID
IDREF
ENTITY
IDREFS
ENTITYS
限制继承
列继承
限制或扩展继承

图 3.1

Schema 中的内置原始类型包括：

string	字符串类型。
boolean	布尔值，值只能取“真”或“假”。
decimal	表示任意精度的数字。
float	单浮点数，即 32 位单精度浮点数。
double	表示 64 位的双精度浮点数。
duration	代表一段持续的时间。
datetime	表示一个确定的时间。
time	表示每天重复的确定时间。
date	表示日期。
hexBinary	十六进制编码的数据。
base64Binary	采用 base64 编码方式的数据。
anyURI	代表根据 RFC 2396 定义的 URI。
Qname	全名，包括名字空间前缀和本地名。
NOTATION	表示 NOTATION 类型属性。

继承类型的 XML 数据类型有：

normallizeString	继承自字符串类型，表示正规化空白的字符串。
token	继承自 normallizeString 类型，表示记号（tokenized）字符串。
language	继承自 token 类型，说明使用的何种自然语言。
IDREFS	表示 IDREFS 类型属性。
ENTITIES	表示 ENTITIES 类型的属性。
NMTOKEN	表示 NMTOKEN 类型的属性。NMTOKEN 是名字的集合，这些名字是由字符，包括字母、数字和其他字符构成。NMTOKEN 类型继承自 token 类型和 Name、NCName 两类型不同，NMTOKEN 类型对于名字的开始字符没有限制。下面会提到 Name 和 NCName 类型的限制。
NMTOKENS	表示 NMTOKENS 类型的属性，参见 NMTOKEN 类型。
Name	表示 XML 数据中的名字。名字必须是以字母、下划线或是冒号开始，后面接上字母、数字这样的字符。Name 类型继承自 token 类型。
NCName	和 Name 几乎相同，只是 NCName 类型的名字不能以冒号开始。NCName 类型继承自 Name 类型。
ID	表示 ID 属性类型。ID 属性的取值必须是 NCName。ID 类型继承自 NCName 类型。
IDREF	表示元素的 ID 属性对指定的 ID 属性的引用。IDREF 必须使用 NCName 类型的引用对象，并且这个引用对象在 XML 文档一定是

	元素或属性的值。IDREF 类型继承自 NCName 类型。
ENTITY	表示 ENTITY 类型的属性，用来标识一个引用的实体。继承自 NCName 类型。
integer	整数类型，可以为负数，比如“-6”。继承自 decimal 类型。
nonPositiveInteger	表示非正数类型，即取值小于等于零。继承自 integer 类型。
negativeInteger	负数类型。继承自 nonPositiveInteger 类型。
long	长整型数类型。取值的范围是 -9223372036854775808 到 9223372036854775807。long 类型继承自 integer。
int	整型数类型。取值介于-2147483648 和 2147483647 之间。int 类型继承自 long 类型。
short	短整型数。取值范围是-32768 到 32767。继承自 int 类型。
byte	用 8 位二进制数表示的整数，可能的取值限于-128 到 127。继承自 short 类型。
nonNegativeInteger	非负数类型。取值要求大于等于零。继承自 integer。
unsignedLong	无符号长整型数，取值限于 0 到 18446744073709551615。继承自 nonNegativeInteger 类型。
unsignedInt	无符号整型数，取 0 到 4294967295 之间的整数。继承自 unsignedLong 类型。
unsignedShort	无符号短整型数，值限于 0 到 65535。继承自 unsignInt 类型。
unsignedByte	表示取值限于 0 到 255 的整数。继承自 unsignedShort 类型。
positiveInteger	正整数，值大于等于 0。继承自 nonNegativeInteger 类型。

除了前面两大类的数据类型，Schema 还有一类称为“范围”类型的数据类型。因为范围类型的作用是为像 length、minInclusive 这样的内置数据类型确定取值范围。在范围类型确定的范围内的取值，是合法的值。

范围类型和元素一样定义，每个范围意思有一个 fixed 属性，属性值是布尔值。当已定义了简单类型时，可以阻止简单类型的派生元素修改指定的范围值，只要将 fixed 属性的值设为“真”（true）。

约束简单类型取值的范围类型有：

enumeration	枚举类型，将取值限制在指定的几个值的集合中。
fractionDigits	定义了小数部分位数的最大值。
length	联合体的长度。长度值取决于数据类型，并且要求长度值是 nonNegativeInteger 类型。
maxExclusive	确定了值的正极限，所有的取值都必须小于 maxExcusive 的值。
maxInclusive	元素能达到的最大值。
maxLength	length 的最大值。
minExclusive	可能的取值的负极限。

minInclusive	元素的最小值。
minLength	length 的最小值。
patten	指定了元素的取值需匹配的样式。
totalDights	小数的最大位数。
whiteSpace	取值限于“preserve”、“replace”和“collapse”三种之一。

至此，Schema 的标准已经介绍完了。对照，前面的实例，你可熟悉这众多的属性和数据类型的使用。Schema 的强大也在于对于数据类型的良好支持。尽管 Schema 书写起来比较复杂和繁琐，使用 Schema 文档在简洁方面和使用 DTD 的文档逊色了很多。但是，文档的简洁不是我们所关心的，因为我们可以使用工具来根据 XML 文档自动生成 Schema 文档的大致框架，然后，你可以手工加以修改，使之完全满足你的需要。

我们提倡这种修改开发工具自动生成的 Schema 代码的方法，因为这可以大大简化你的工作，你可以将更多的精力放在如何使用外部应用程序分析读取 XML 的问题上，或是如何使用更好的数据结构模型来表达数据的深层信息等更加重要的问题上。

我们试验了在.NETbeta2 平台上自动生成 Schema 代码，结果是基本可以满足你的要求，尽管还要加以修改。相信在.NET 开发平台正式推出时，自动生成的 Schema 会更加完善，更加智能化。

3.4　XSL 可扩展样式单语言

3.4.1　XSL 的机制

W3C 制定了另一种表现 XML 文档的样式单语言 XSL（eXtensible Stylesheet Language）。XSL 本身也是 XML 的应用，符合 XML 的语法规范，可以被 XML 的分析器处理。

CSS 是通过对 XML 文档的标记等元素进行逐一描述，是静态的描述文档的信息。CSS 本身广泛应用于 HTML 文档的表现，形成了较完整的语法规范，对于新的语言 XML 的支持不够全面，不能体现 XML 的一些特点和优势。而 XSL 是为 XML 量身定做的，在体现 XML 的特性时有着先天的优势。同时，XSL 的一部分 XSLT 广泛应用于 XML 文档的转换，也说明了 XSL 是符合 XML 特性要求的。

XSL 是一种语言，通过对 XML 文档进行转换，然后将转换的结果表现出来。转换的过程是根据 XML 文档特性运用 XSLT（XSL Transformations）将 XML 文档转换为树形结构（结果树），这棵树是带有样式信息的。再按照 FO（Formatted object）分析结果树，将 XML 文档表现出来。通过前面的介绍，你应该知道 XML 文档本身具有树形结构的特点。所谓树形结构是说文档中各元素的关系就像东方家庭的家谱一样，有确定的辈份和祖先。元素的位置决定了其在文档中的作用，就像我们在家谱中的位置决定了我们在家中的影响力一样。XML 文档的这一特性还被用于另一种场合，也就是后面要讲的 DOM 接口。

XSLT 是根据 XML 文档的结构建树，同时还运用 XPath（XML Path Language）标准说明识别、选择 XML 文档的数据，比如 XPath 规定了如何选取相同父节点的所有子节点。

XSLT 文档实际上是一个模板，规定了 XML 文档中的元素、属性等内容的关系和这些内容如何向树形结构转换的方式。

建树的结果是产生可供浏览的 HTML 文档，或是带 CSS 的 XML 文档等能被浏览器浏览的文档。运用 XSLT，我们不止可以转换 XML 文档到 FO 格式，还能转换 XML 文档到其他格式的文档，比如 HTML 文档、XHTML 文档等。

XSLT 和 XPath 的标准已经有正式的标准，由 W3C 于 1999 年 11 月 16 日推出了。但 FO 的标准还在商定中，到目前为止，还没有正式标准推出。

XSL 表现 XML 文档的两个步骤，建树和表现树。其中，建树可以在服务器端执行，也可以在客户端执行。在服务器端执行时，在服务器端转换生成 HTML 文档，然后传给客户端浏览。如果在客户端执行则需要客户端支持 XML 和 XSL，因为从服务器端传来的是 XML 和 XSL 文档，这些文档在客户解析执行。

XSL 得到了 Microsoft 公司的支持。在 Internet Explorer 中支持 XSL 的草案，并且 Microsoft 公司将在新版本的产品中支持最新的 XSL 标准。

3.4.2　XSLT

由于 XSLT 标准日趋成熟，XSLT 的作用已不仅仅是将 XML 转换为带样式信息的结果树，实际上，更多的情形是使用 XSLT 转换 XML 文档成为可以被各种系统或是应用程序解读的数据。比如，在要两个不同的商业系统之间交换数据，可以将双方根据不同的 Schema 建立的文档使用 XSLT 转换为可被对方理解的 XML 文档，实现了信息的交换。

XSLT 还可以将 XML 文档转换为可被各种浏览器浏览的 HTML 页，转换为供打印的 PDF 类型文档。这些转换都是动态完成的。

在本书中，我们只关心使用 XSLT 将 XML 文档转换成动态地可以被浏览的 HTML 的过程和方法。但是，我们会介绍 XSLT 的各种元素，使你充分了解 XSLT。

下面给出一个完整的使用 XSL 样式单的例子，通过这个例子，我们为你介绍 XSLT 的转换方法和 XSLT 的书写格式。

注意：我们将为代码增加行号。这只是为了说明的方便，你在代码中应该省略行号。任何 XML 代码都是没有行号的。

程序清单 3-8：example-3-7.xml

```
<?xml version="1.0" encoding="gb2312"?>
<?xml-stylesheet type="text/xsl" href="example-3-8.xsl"?>
<档案>
    <学生 性别="0">
        <姓名>张三</姓名>
        <ID>001</ID>
        <年龄>16</年龄>
        <电话>12345678</电话>
    </学生>
    <学生 性别="1">
        <姓名>李四</姓名>
        <ID>002</ID>
```

```
        <年龄>20</年龄>
        <电话>23456789</电话>
    </学生>
    <学生 性别＝"0">
        <姓名>王五</姓名>
        <ID>003</ID>
        <年龄>19</年龄>
        <电话>34567890</电话>
    </学生>
</档案>
```

程序清单 3-9：example-3-7.xsl

```
01  <?xml version="1.0" encoding="gb2312"?>
02  <xsl:stylesheet xmlns:xsl="http://www.w3.org/TR/WD-xsl">
03
04      <xsl:template><xsl:apply-templates/></xsl:template>
05      <xsl:template match ="档案">
06          <html>
07              <head>
08                  <title>学生档案示范</title>
09              </head>
10              <body>
11                  <table  border="2"  title="这是一个学生档案示范。"
12                       align ="center" >
13                  <caption >学生档案示范</caption>
14                  <tr bgcolor="yellow">
15                      <th>姓名</th>
16                      <th>学号</th>
17                      <th>年龄</th>
18                      <th>联系电话</th>
19                      </tr>
20                  <xsl:for-each select="学生">
21                      <tr>
22                          <td><xsl:value-of select="姓名"/></td>
23                          <td><xsl:value-of select="ID"/></td>
24                          <td><xsl:value-of select="年龄"/></td>
25                          <td><xsl:value-of select="电话"/></td>
26                      </tr>
27                  </xsl:for-each>
28                  </table>
29              </body>
30          </html>
31      </xsl:template>
32  </xsl:stylesheet>
```

example-3-7.xml 是一个 XML 文档，我们用 XSL 样式单 example-3-8.xsl 来表现这个 XML 文档。图 3.1 是浏览这个 XML 文档的效果图。下面，我们来逐行分析 example-3-8.xsl。

XSL 文档本身也是 XML 文档，所以第一行和 XML 文档一样，是文档声明。属性和属性值可参阅 3.1 节。第 2 行声明了本 XSL 样式单的名称空间。

第 4 行指明应用模板函数。第 5-31 行是模板函数的定义。第 5 行说明了模板要匹配的节点，在示例中，匹配根结点“档案”。第 7-9 行指明了该网页的名称。第 11、12 行指明了数据的表现形式，示例中采用表格形式。三个参数“border”说明表格的边框宽度，“title”

指明在鼠标指向表格时的提示“这是一个学生档案示范”，“align”的值为“center”说明表格居中显示。

第 13 行标记表格的名称。第 14-19 行，标记了表格的首行，参数“bgcolor”说明首行的背景为黄色。第 20-27 行标记了表格其他行的内容。第 20 行，是一个循环指令，要求匹配显示每一个相同标记的节点。第 24-25 行的内容类似，用指令“xsl:value-of select= ‘A’ ”来显示每一个标记为 A 的节点。

当然，XML 的基本规范“格式良好”要求标记要成对出现，且能匹配。所以，没提到的如 28、29 行等是对上面标记的匹配。

XSL 还提供一种类似于函数调用的方法去匹配模板。其形式为：

```
……
<xsl:apply-templates select="A"/>    匹配参数为 A 的模板
……
<xsl:template match="A">             参数为 A 的模板定义
……
</xsl:template>                      标记的匹配
```

通过这种方法，我们可以表现更复杂的内容。

XML 文档要求格式良好（well-formed），XSLT 可以将 XML 文档转换为格式良好的 HTML 文档供浏览器浏览。而且由于 XSLT 还能转换 XML 文档到其他格式的文档，使用 XSLT 可以保证转换出来的结果文档是格式良好的。

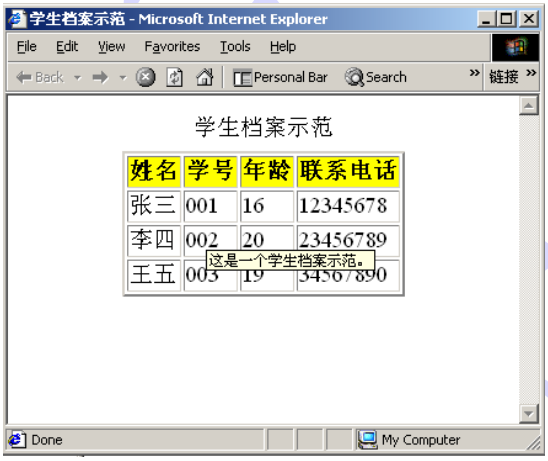


图 3.2

通过上面的例子，我们可以知道 XSLT 实际上是通过模板（template）将源文档按照模板的格式转换为结果文档的。模板定义了一系列的元素来描述源文档中的数据和属性等内容，在经过转换之后，建立树形结构（带信息的结果树）。

在示例中，我们使用了循环来匹配模板：

```
<xsl:for-each select="学生">
    <tr>
        <td><xsl:value-of select="姓名"/> </td>
        <td><xsl:value-of select="ID"/></td>
        <td><xsl:value-of select="年龄"/> </td>
```

```
                <td><xsl:value-of select="电话"/> </td>
            </tr>
        </xsl:for-each>
```

这是一种类似于函数的调用的匹配方式。“xsl:for-each”元素循环调用“xsl:value-of”元素来匹配不同的标记，直至将所有的“学生”标记都匹配一遍后，才结束循环。

关于模板其他元素的介绍见表 3-6。

表 3-6

元素名	说明
xsl:apply-imports	调用导入的外部模板，可以应用为部分文档的模板
xsl:apply-templates	应用模板，通过“select, mode”两个属性关键字确定要应用的模板。当多个模板符合要求时，根据优先级选择模板，同优先级时应用最后出现的样式
xsl:attribute	为元素输出定义属性节点，属性关键字“name”定义属性名，“namespace”声明名字空间
xsl:attribute-set	定义一组属性节点。关键字“name”定义组结点名，“use-attribute-sets”声明匹配属性到每个节点
xsl:call-template	调用由 call-template 指定的模板
xsl:choose	根据条件调用模板
xsl:comment	在输出加入注释
xsl:copy	复制当前节点到输出
xsl:copy-of	复制当前节点的所有子树，比如属性节点、子节点等到输出
xsl:decimal_format	定义格式化数字函数输出的小数格式，如“decimal-separator、infinity”等
xsl:element	在输出中创建新元素
xsl:fallback	当 XML 分析器不支持当前的 XML 版本时，会调用这个元素来判断能否正常处理，如不能则改用默认方式处理，并输出报错信息
xsl:for-each	循环调用模板匹配每个节点
xsl:if	模板在简单情况下的条件调用
xsl:import	导入样式单，只能作为"xsl:stylesheet"和"xsl:transform"的字节点使用。导入的样式单优先级最高(高于当前样式单和"xsl:include"指定的样式单)
xsl:include	导入其他样式单，其应用的方式与”xsl:import”类似
xsl:key	定义关键字，再使用 key（）函数作出关键字判断，主要应用在 XPath 中
xsl:message	发送文本消息给消息缓冲区或消息对话框
xsl:namespace-alias	改变当前名称空间
xsl:number	在结果树中插入带格式的数字
xsl:otherwise	条件调用模板
xsl:output	为序列化结果树指定应用选项
xsl:param	为 xsl:stylesheet 或 xsl:template 指定参数
xsl:preserve-space	在文档中保留空格

(续表)

元素名	说明
xsl:processing-instruction	在输出中加入处理指令
msxsl:script	在扩展脚本中定义全局变量、函数
xsl:sort	排序节点
xsl:strip-space	合并文档的空格
xsl:stylesheet	指定样式单
xsl:template	指定模板
xsl:text	输出文本
xsl:transform	等同 xsl:template
xsl:value-of	为选定节点加入文本值
xsl:variable	声明表达式中的变量
xsl:when	选择模板
xsl:with-param	撤销模板参数

除了上面提到的类似函数调用的循环匹配模板方式，XSLT 还有其他类似于我们常见的程序设计语言的元素。下面我们介绍简单条件下（只有匹配和不匹配两种情况）和多选择条件下的模板匹配调用方式。

简单条件下的模板调用格式为：

```
<xsl:if  test = "布尔表达式">
……
</xsl:if>
```

当布尔表达式的值为真时，执行省略号代表的内容，否则跳过这些内容。这只能判断两种情况。当需要判断多种情形，选择匹配模板时，配合使用元素<xsl:choose>、<xsl:otherwise>和<xsl:when>，格式如下：

```
<xsl:template match="节点名">
   <xsl: choose>
      <xsl:when test="布尔表达式 A">
         //代码段 1
      </xsl: when>
      <xsl:when test="布尔表达式 B">
         //代码段 2
      </xsl: when>
      <xsl: otherwise>
         /代码段 3
      </xsl: otherwise>
   </xsl: choose>
   <xsl: apply-templates />
</xsl: template>
```

就像程序设计语言一样：布尔表达式 A 为真时，编译执行代码段 1；表达式 B 为真时，执行代码段 2；其他情况下，执行代码段 3。我们可以通过这两种方法，有选择的对部分 XML 文档加以表现。

下面给出一些实例，演示如何使用表中的元素来完成 XML 的转换。

创建元素的写法是：

```
<xsl:element name="TITLE">
This is a test.
</xsl:element>
```

得到的元素为：

```
<TITLE> This is a test. </TITLE>
```

创建属性需要使用元素“xsl-attribute”。比如为元素“TITLE”创建的属性“ID”的写法是：

```
<TITLE>
    < xsl:attribute name ="ID">
        title1
    </xsl:attribute>
This is a test.
</TITLE>
```

为元素“TITLE”创建的属性为：

```
<TITLE ID="title1">This is a test. </TITLE>
```

我们还可以在文档中加入注释来增加代码的可读性：

```
<xsl:comment>Create a title.</xsl:comment>
```

转换后，得到的注释如下：

```
<!--Create a title.-->
```

在前面的表中，我们讲到使用“copy”和“copy-of”复制节点，比如：

```
<? xml version="1.0"?>
<A id="001">
    <B>comment</B>
    <C>text</C>
</A>
```

我们使用样式单来复制节点：

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/TR/WD-xsl" >
<xsl:template match="A">
        <DIV>
            <xsl:copy-of select="."/>
        </DIV>
        <DIV>
            <xsl:copy/>
</DIV>
</xsl:template>
</xsl:stylesheet>
```

转换的结果是：

```
<DIV>
    <A id="001"><B>comment</B><C>text</C>
</DIV>
<DIV>
    <A/>
</'DIV>
```

得到这样的结果是因为“xsl:copy-of”拷贝当前节点的所有内容，包括节点本身、属性和子元素。而“xsl:copy”只拷贝当前节点本身。

XSLT 可以将 XML 文档转换为多种类型的文档，我们使用元素“xsl:output”来指定输出的文档类型。

```
<xsl:output method="html">
```

说明输出的文档类型为 html 文档。输出的文档类型默认是 XML 文档，除非指定了输

出的文档类型。如果在模板中使用“<html>”标记，这样输出的文档类型是 html 文档。

“xsl:sort”元素可以将节点进行排序，比如，按照大小排序：

```
<xsl:sort case-order="lower-first" select="@ID">
<xsl:sort case-order="upper-first" select="@ID">
```

此时，按照使用“case-order”指定的方式，根据关键字 ID 排序。当“case-order”指定“lower-first”时，关键字 ID 值小写的在前，当“case-order”指定为“upper-first”时，关键字 ID 值大写的在前。

如果写成：

```
<xsl:sort order="ascending" select="@ID">
```

则排序的依据变为根据关键字的字母进行升序排序。如果“order”的值变为“descending”，那么将根据关键字的字母进行降序排序。这里关键字依然是“ID”属性的值。

“xsl:sort”元素还可以根据数字进行排序。有两种方法：

```
<xsl:sort data-type="text" select="@ID" >
<xsl:sort data-type="number" select="@ID" >
```

有一组 ID 属性值：156，26，7，34。使用前一种方法得到的结果是：156，26，34，7。使用第二种方法得到的结果是：7，26，34，156。两种方法区别不言而明。

“xsl:sort”还可以根据自定义的关键字来排序。比如：

```
<xsl:for-each select ="A" order-by="B">
```

指明匹配节点“A”，根据关键字 B 排序 A 的内容。

后面我们将介绍 XPath 的具体内容。

3.5　XPath 节点路径匹配

XPath 是 XSLT 的重要部分。XPath 的作用在于为 XML 文档的内容定位，并通过 XPath 来访问指定的 XML 元素。在 XPointer 和 XSLT 之间，我们使用一种通用的语法——XPath 来实现功能的共享。XPath 是一系列规则，这些规则指定了对 XML 文档树型结构的访问方式。通过 XPath 规则，可以实现对 XML 文档的树形结构中每一个节点按照制定的规则去访问。为 XML 文档的元素寻址是 XPath 的基本功能，也是 XPath 的本意。

作为功能的扩展，XPath 还可以判断一个节点是否与它的类型匹配，在 XPath 中是要匹配类型才能正确访问节点的，这一点在 XSLT 中得到了应用。

XPath 将 XML 的文档描绘为结构树，节点的类型有元素节点、属性节点、注释节点等共有七种。对于每一个节点，XPath 使用一种方法计算节点的值（string-value），XPath 完全支持名字空间，带有名字空间的节点名称为扩展节点名（expanded-name.）。

XPath 的主要语法结构是表达式。表达式对 XPath 的数据对象取值来匹配相应的访问规则。表达式经常用于 XML 文档的属性匹配，表达式的取值通常用于文本的处理。XPath 中的数据对象类型有四种：节点集、布尔值、字符串和数字，下面逐一介绍。

（1）节点集

按照给定的规则，根据 XPath 文法匹配得到的一组节点。其他的数据类型不能转化为节点集类型。另外，“|”运算的运算数只能是节点集类型。

（2）布尔值

布尔值有两种取值：真或假。

布尔表达式有：

or　对左右两个操作数进行“或”运算，或者为左右操作数取布尔值后进行“或”运算。当两操作数至少有一个为真时表达式就为真，否则表达式值均为假。需要说明的是，当第一个操作数为真时，不对第二个操作数进行取值。

and　对左右两个操作数进行“与”运算，或者为左右操作数取布尔值后进行“与”运算。当两操作数全为真时表达式就为真，其他情况表达式值均为假。与“或”运算类似，当第一个操作数为假时，不对第二个操作数进行取值。

EqualityExpr　包括“=、!=、<=、<、>=、>”六种运算。这些运算都是将操作数取布尔值，然后按照在数学中的运算进行比较，比较是从左向右进行的。比如表达式“7>9<1”的值为真，“7>9>4”的值为假。

表达式的优先级为：“or”>“and”>“EqualityExpr”，并且“=、!=”高于“<=、<、>、>=”。其中，取操作数的布尔值由布尔函数完成。

（3）字符串

数据由字符组成，要保证字符的编码不会重复，发生歧义，在使用 16 位的 Unicode 编码是可能出现这一问题。

（4）数字

数字类型在 XPath 中是浮点数。根据 IEEE754 标准，可以是双精度数据，非数据（），正无穷大，负无穷大，正负 0 等。

和布尔值类型相似，数字运算符对操作数调用数字函数进行取值运算。数字运算符有：

+、-　　数值的加减运算。

div　　浮点除法，使用 IEEE754 标准。

mod　　取模运算。比如：“8 mod -3”的值是 2，“-5 mod -2”值是 1。

XPath 对 XML 文档的树型结构的节点分为元素节点、属性节点和文本节点等。每种节点都有字符串值，有些节点字符串值是节点的一部分，有些节点的字符串值是通过计算得到的。这里的字符串值与 DOM 中的“nodeValue”的返回值是不同的。节点可以有节点扩展名，节点扩展名用来区分节点。类似于其他格式文档，比如 XML 文档。节点间是派生、兄弟和祖先关系。具体讲节点的类有：

（1）根节点

顾名思义，根节点是文档结构的最上层节点，且是惟一的。根节点可以派生出结构树的所有节点，有些节点是子节点，有些是子节点的子节点。匹配总是由根结点开始的，因为根节点是文档结构树的基础节点。根节点不能有节点扩展名（expanded-name）。根节点的字符串值是所有后继节点字符串值的和。

（2）元素节点

文档中的每个元素在结构树中都有元素节点与之对应。元素节点可以使用扩展元素名。

元素节点可以是元素节点、注释节点、指令节点和文本节点的子节点。如果在 DTD 文档中，元素属性有“ID”这一项，元素节点也将有确定且惟一的标识（ID）。元素节点的字符串值是其所有文本子节点的字符串值的和。

（3）属性节点

每个元素节点可以作为相关的一组属性节点的父节点。但是，属性节点不是元素节点的子节点。这种结构关系是单向的。元素节点不可以共享属性节点。在 XML 文档中不同元素可以有相同的属性，但在树形结构中这些元素的这一相同属性所对应的属性节点是不同的。对于有缺省属性的节点，在结构树中有属性节点对应。在 DTD 中声明过的，但声明为“#IMPLIED”，那么没有属性节点与之对应。

注意：运算符“=”比较的是节点的“值”是否相同，不是比较节点是否相同。也就是说，这些不同的属性节点的值是相同的，尽管是不同的节点。

属性节点有节点扩展名和字符串值。用于声明名字空间的属性没有属性节点对应。

（4）名字空间节点

每个元素都有相关的一组名字空间节点。这些节点包括：每个元素在特定范围内的指定名字空间前缀和默认名字空间前缀。类似于属性节点，名字空间节点与元素节点的关系也是单向的，名字空间节点也不可在元素之间共享。

（5）指令节点

每条处理指令（processing instruction）都有对应的指令节点，除了用于文档类型声明的指令。XML 文档声明没有指令节点。指令节点的节点扩展名为空，说明指令处理本地对象。指令节点的字符串值是指令对象部分和所有空格，不包括结束标记“?>”。

（6）注释节点

除了出现在文档类型声明时的注释，所有注释都有相应的节点称为注释节点。注释节点的字符串值是注释部分，不含开始标记“<!--”和结束标记“-->”。注释节点不能有扩展名。

（7）文本节点

文本节点是字符的集合。文本节点应包括尽可能多的字符，使得文本节点不存在前后相邻的兄弟文本节点。所谓字符是指 CDATA 段的内容。注释节点、指令节点和属性节点中的字符不产生文本节点。

文本节点没有节点扩展名。

XPath 的路径说明是比较简单的，最基本的是路径匹配。这很像文件路径的表达方式。路径匹配的符号有：

- “/”　选择根节点或是表示路径。比如“/A/B/C”表示匹配 A 节点的孙节点 C
- “//”　匹配子元素。比如：“//A”匹配所有 A 元素，“//A/B”匹配 A 中的所有 B 元素
- “*”　通配符。表示所有元素。比如“/A/*”匹配 A 节点的所有子节点
- “|”　或运算，表示路径的“或”运算。比如“//A|//B”表示所有 A 和 B 元素

仅仅有路径说明是不够的，下面加入关于位置的规定：

- [1]　　选择第一个元素
- [last()]　　选择最后一个元素
- [number]　　按照指定数字的确定位置。/booklist/book[3]，匹配 booklist 中第三个 book 元素
- [position()=number]　由数字指定第几个位置的元素
- [position()>number]　匹配所有位置号大于指定数字的元素

现在，我们来看看，加入对匹配属性的要求。这样，对数据内容的要求就比较完整了。对属性的匹配要求主要是通过“@”运算符来完成的。

- [@id]　　匹配具有属性“id”的元素
- [@name]　　匹配具有属性名“name”的元素
- [@*]　　匹配有属性的元素
- [not(@*)]　　匹配没有属性的元素
- [@id=“name”]　匹配有属性“id”为“name”的元素

XPath 还提供类似于亲属关系的匹配方式。每个节点都有祖先节点、父节点、兄弟节点、子节点、后继节点。

child 关键字匹配子元素。像“child::A”匹配子元素节点“A”，“child::*”匹配所有子元素节点，“child::text()”匹配所有文本子节点，“child::node()”匹配所有子节点，包括所有类型。

descendant 关键字匹配后继节点。“descendant::para”匹配后继节点“para”，“descendant::*”匹配所有后继节点，不包括当前节点。“descendant-or-self::*”属性匹配所有后继节点和当前节点。

还有“ancestor，foollow，self，parent”等关键字，使用方法举例说明如下：

ancestor-or-self::para	匹配 para 的祖先，如果当前节点是 para。匹配结果包括当前节点。
ancestor::para	匹配所有 para 的祖先。
self::para	所有 para 自身元素。
follow-sibling::*	匹配所有后面紧邻的兄弟元素节点。
follow::*	匹配所有后续节点。
preceding-sibling::book[position()=1]	匹配 book 的前一个兄弟节点。

对于属性，使用关键字“attribute”。比如“attribute::*”匹配所有属性节点。“attribute::name”匹配属性“name”指定的属性节点。

下面来看几个综合的例子：

```
child::chapter/descendant::para
```

匹配当前节点的“chapter”子节点的“para”后继节点。

```
child::para[position()=last()-1]
```

匹配位于倒数第二个“para”子元素。

```
/child::doc/child::chapter[position()=5]/child::section[position()=2]
```

匹配“doc”节点的子节点——第 5 章（chapter）第 2 节（section）。

```
child::*[self::chapter or self::appendix]
```

匹配当前节点的“chapter”子元素和“appendix”子元素。

```
child::para[position()=5][attribute::type="warning"]
```

当当前节点的第 5 个“para”子节点具有值为“warning”的属性“type”，那么匹配这个子节点。

在 XPath 中，可以使用缩写来简化表达式的书写。下面给出一些缩写的例子：

para	表示选择当前节点的 para 子节点。
text()	选择当前节点的所有文本节点。
@name	选择当前节点的名为 name 的属性节点。
@*	选择当前节点的所有属性节点。
para[1]	选择当前节点的第一个 para 子节点。
para[last()]	选择当前节点的最后一个 para 子节点。
*/para	选择当前节点的所有 para 孙节点。
/doc/ch[5]/s[2]	选择 doc 节点的第 5 个 ch 节点的第二个 s 节点。
.	选择当前节点。
.//para	当前节点的 para 子节点。
..	选择当前节点的父节点。
../@attr	选择当前节点的父节点的属性节点 attr。
para[@type="warning"]	选择当前节点所有属性“type”为“warning”的子节点。
chapter[title]	选择当前节点至少有一个 title 子元素的 chapter 子元素。
A[@attr1 and @attr2]	选择当前节点的有属性 attr1 和 attr2 的 A 子元素。
div/para	等价于 child::div/child::para。“child::”可以省略。
para[attribute::type="warning"]	等价于 para[@type=”warning”]，“attribute::”等于“@”。
/descendant-or-self::node()/para	等价于//para，“/descendant-or-self::node()”等于“//”。

注意：//para[1]选择当前节点的所有 para 节点第一个子节点，而/descendant::para[1]选择的是当前节点的第一个 para 子元素。两者不等价，是不能互换的。

.//para	等价于 self::node()//para，“.”等于“self::node()”
../title	等于 parent::node()/title，“..”等于“parent::node()”。

通过上面介绍的简写方式，你是不是觉得书写 XPath 表达式变得容易了？

在 XPath 中还有函数的概念。使用函数可以用运算完成匹配要求。上面提到的数字函数和布尔函数等都是在 XPath 的核心函数库（Core Function Library）中的函数。每个在核心函数库的函数都有指定的函数原型，用来说明函数的返回值、函数名和参数类型。下面是对函数原型的介绍：

（1）节点集函数

表 3-7　节点集函数表

函数形式	说明
number last()	返回节点根据表达式选择的子节点数
number position()	返回由表达式指定的节点位置值
number count(node-set)	返回参数节点集（node-set）的节点个数
node-set id(object)	返回有指定 ID 的节点，返回值是节点集
string local-name(node-set?)	返回节点的节点扩展名中的本地名。如果参数节点集为空，或节点没有节点扩展名返回空串。如参数省略，将文本节点集作为缺省参数
string namespace-uri(node-set?)	按照节点集中的文档结构顺序返回节点扩展名中的名字空间 URI。当参数节点集为空、第一个节点没有节点扩展名或 URI 为空时，返回空串。省略参数时文本节点集是默认参数
string name（node-set?）	返回节点的节点扩展名

（2）字符串函数

表 3-8　字符串函数表

函数形式	说明
string string(object?)	转换参数对象为字符串。有下列情形： 1. 节点集按结构顺序将节点的字符串值转换为字符串。空节点集得到空串 2. 数字按照以下规则转换： NaN　—> NaN 串 正负 0　—> 0 正无穷　—> Infinity 负无穷　—> -Infinity 数字　—> 数字(用“-”标记负数，浮点数使用 IEEE754 标准表示) 3. 布尔值转换为 true 和 false
string concat(string, string, string*)	返回将参数字符串连接起来的字符串
Boolean start-with(string, string)	当第一个参数是以第二个参数开始时，返回真，否则为假
Boolean contains(string, string)	当第一个参数字符串包含了第二个参数时，函数返回真，否则为假
string substring-before (string, string)	返回在第二个参数字符串之前的子串。如果在第一个参数字符串中不包括第二个参数字符串，则返回空串
string substring-after(string, string)	和 substring-before()函数类似，但返回第二个参数之后的子串，或空串
string substring(string, number, number?)	返回参数字符串中以第一个数字为起点，第二个数字为长度的子串。第二个数字省略时，返回字符串直到参数字符串的结束。二个数字会根据 IEEE754 标准四舍五入 比如：substring("abcdef",3, 2)值为“de” substring("abcdef", 1.6, 4)值为“ab” substring("abcdef", 5 div 2)值为“abcdef” substring("abcdef", -1div0, 1 div 0)值为“”

(续表)

函数形式	说明
number string-length(string?)	返回参数字符串的字符数。省略参数时，将文本节点转换为字符串，返回文本节点的字符串值
string normalize-space(string?)	处理参数字符串的空格。去掉字符串首尾的空格，合并中间的多个空格成一个空格。省略参数时，将文本节点转换为字符串处理
string translate(string, string, string)	第一个参数是被替换对象，第二个参数为替换字母表，第三参数为字母表中的字母的格式，是按顺序对应字母表的 比如：translate("window","id","I")得到“wInow”。字母表中字母重复出现是，先出现的决定要替换的字母。第三个参数中不必要多余的内容会被省略

（3）布尔函数

表 3-9　布尔函数表

函数形式	说明
boolean boolean(object)	将参数转换为布尔值。规则如下： 数字（不包括 NaN 和正负 0），非空节点集，长度不为 0 的字符串转换为真
boolean not(boolean)	返回参数布尔值的否运算值
boolean true()	返回“真”
boolean false()	返回“假”
boolean lang(string)	判断文本节点的语言是否和声明中的“xml:lang”属性一致。一致返回真，否则返回假。如果文本节点没有“xml:lang”属性，用最近的祖先节点的“xml:lang”属性代替。如果没有这样的属性，返回“假”。对于属性值有后缀，忽略后缀

（4）数字函数

表 3-10　数字函数表

函数形式	说明
number number(object?)	将参数对象转换为数值。规则如下： 由被空白字符（可省略）包含的正（负）数构成的字符串，被转换为符合或最接近 IEEE754 标准的数学值。其他字符串转换为 NaN 布尔值“真”转换为“1”，“假”转换为“0” 节点集先转换为字符串，再按照字符串的方法转换为数值
number sum(node-set)	返回参数节点集中，每一个节点的字符串值累加的和
number floor(number)	返回不大于参数的最大的整数
number celling(number)	返回不小于参数的最小的整数
number round(number)	对参数四舍五入，返回结果。如果参数是 NaN、正负无穷大、正负 0 则返回本身原值，不进行转换。介于“-0.5”和“0”之间的数转换为“负 0”

3.6　XML 编辑器

3.6.1　Microsoft Visual Studio.NET 7.0

Microsoft Visual Studio.NET 7.0（VS7）是一个极其强大的开发平台，XML 部分的开发是由 XML Designer 这个开发工具完成的。一般的 XML 编辑器所具有的功能，XML Designer 都可以实现，比如像书写代码时提示正确的可用的元素名、属性名，XSL 预览功能等。

而且 VS7 的 XML Designer 还提供了自动生成 XML Schema 代码的功能，使用 XML Designer 生成的 Schema，开发人员只需进行简单地修改，就可以得到合适的 Schema 了。当然，对于某些程序员来说，他们更喜欢自己手工书写每一行代码，这样做的好处在于可以了解每一行代码的作用，了解整个程序的结构，并得到相对简洁的代码。但是，手工编写像 Schema 这样的格式要求严格的标记语言无疑会有许多机械的而又必须的重复劳动，比如标记的匹配这样的输入。而且与 DTD 相比较，Schema 格式更加复杂了。所以对于 XML Schema 文档来说，当开发工具生成的 Schema 可以满足数据类型的定义要求，我们只需根据具体情况要加以简单修改就可以使用这些 Schema 代码时，为什么不利用这种工具来简化我们的工作呢？虽然自动生成的 Schema 会比较拖沓，但这是无关紧要的。

下面先简单介绍一下 VS7 的特点，这些特点也体现了微软的新的技术趋势和 XML 应用的方向。然后着重介绍在 VS7 中编写 XML 文档的利器 XML Designer。

Microsoft Visual Studio.NET 7.0 的主要特点如下：

- 使用.NET 框架
- 带来全新的技术——Web Service
- 支持 XML，这一点后面我们会详述
- 拥有完整的开发环境。其中包括开发工具、应用程序模板、编辑工具、调试工具、产品安装配置工具、大量的记录和加强的自动对象模型
- 包括了示范如何使用微软技术和工具的简单应用程序
- 拥有一套微软的技术开发人员编写的非常完整和详细的开发文档。这些文档不仅可以作为开发时的参考手册，还可以作为学习新技术的教材

VS7 中的 XML Designer 可以简化 XML 的编辑和自动生成 XML Schema。XML Designer 提供了一系列处理 XML Schema、ADO.NET 数据集和 XML 文档的工具。对于 Schema，XML Designer 只支持 W3C 通过的 XML Schema Definition（XSD），不支持 DTD 和其他的 XML Schema 语言。比如不支持 XML-Data Reduced（XDR）。

1. Schema View（Schema 模式）

Schema View 提供一种可视化的方法表现元素、属性、数据类型等，从而表现整个 XML Schema 和 ADO.NET 的 DataSets，如图 3.3 所示。

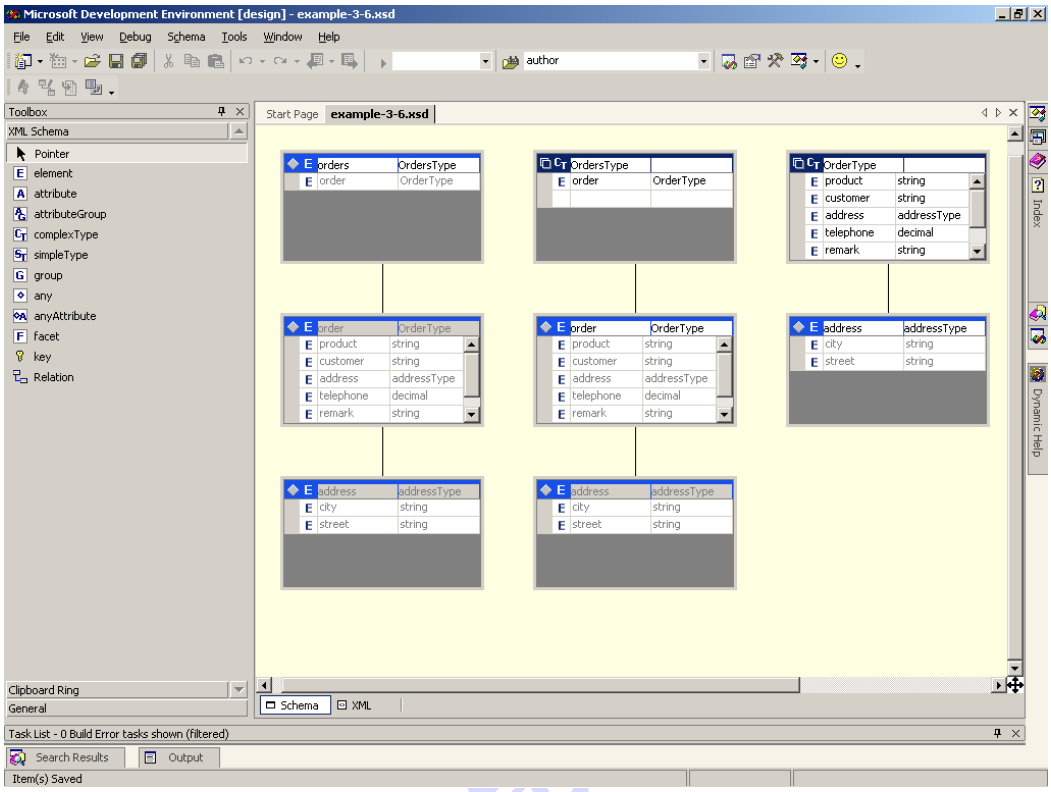


图 3.3

创建 Schema 和 datasets 时，可以直接使用工具箱（Toolbox）的 XML Schema 标签添加元素，或是在服务器浏览器（Server Explorer）中增加元素。最简单的方法就是使用鼠标右键的快捷菜单，在快捷菜单中选择“Add”命令，如图 3.4 所示。

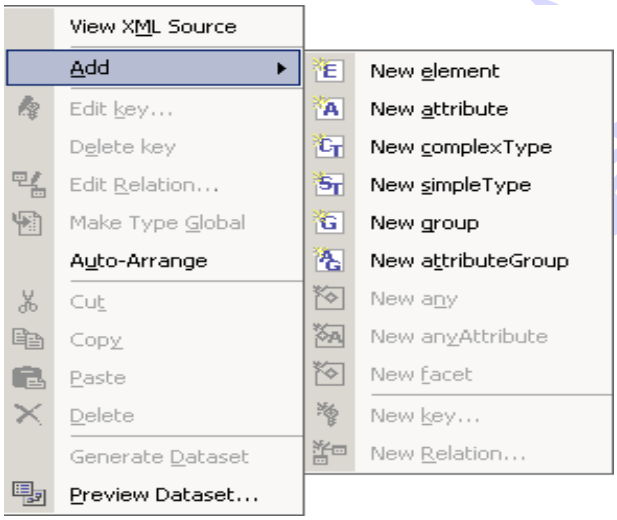


图 3.4

在 Schema 模式下，可以看到右边是工具箱，左边的窗口都自动收起来了。在 VS7 中，引入了将窗口自动收起的机制，这有些类似于 OICQ 的自动隐藏，将不常用的窗口收起来，可以有效地增大工作区的面积。这个窗口的设计在实际开发应用中会带来极大的方便。

Schema View 支持的文件类型为：*.xsd。

Schema View 提供了以下的功能：

- 构建和修改已有的 XML Schema 和 ADO.NET 的 datasets
- 创建和编辑表间的关系
- 创建和编辑关键字，比如增加元素或是属性
- 根据 XML Schema 得到 ADO.NET 的 datasets

注意：在使用 Schema view 时，是没有“撤销”操作的。所以，我们要小心使用“Schema view”模式并经常地保存文档。

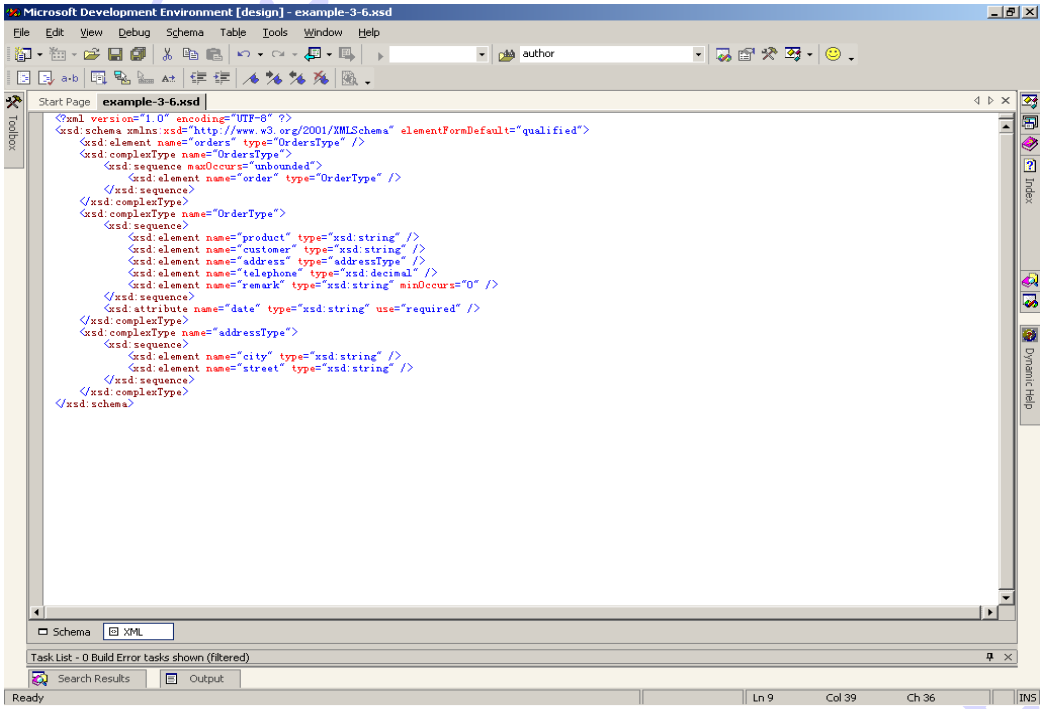


图 3.5

2. Data View（Data 模式）

Data 模式提供了一种可视化的方法利用表格编辑 XML 数据文档。在 Data 模式中，只能编辑 XML 文档的数据内容，不能处理标记和数据结构，如图 3.6 所示。

在 Data 模式中有两个单独的区域：数据表区域（Data Tables）和数据区域。在数据表区域中，列出了 XML 文档中数据间的联系（嵌套关系）。在数据区域中用表格显示在数据表区域中选中的数据。图中，因为 XML 文档元素之间的嵌套关系比较简单，所有在数据表区域只有“Forms”一个标记。

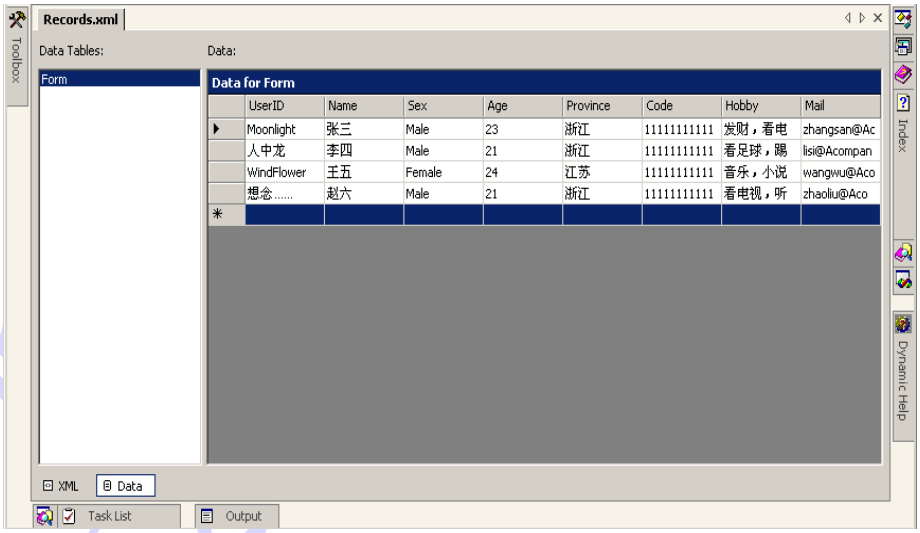


图 3.6

新创建的 XML 文档中没有任何数据，那么也就不能在 Data 模式中显示了。还有，对于某些 XML 文档，尽管这些文档可能是格式良好的，但也不能应用 Data 模式显示。此时，会得到这样的出错信息“Although this document is well formed，it contains structure that Data View cannot display.”，即“尽管文档是格式良好的，但是含有 Data 模式不能显示的结构。”

Data 模式提供了以下的功能：

- 手动建立数据表
- 编辑已有的数据表
- 根据 XML 文档生成 XML Schema
- Data 模式支持的文档类型：*.xml

3. XML View（XML 模式）

XML 模式主要用来处理 XML 文档和编辑各种语言的源代码。前面介绍的 Schema 模式和 Data 模式都提供了 XML 模式的文本编辑窗口，如图 3.7。

XML 模式提供的功能有：

- 根据内容的不同使用不同的颜色表示，就像使用 VS 编写其他语言的代码一样
- 在处理*.xsd 和*.xml 文档时，会根据可用的 Schema 自动生成相应的声明等代码
- 在书写标记的“<”字符后，会列出在当前位置可用的所有元素；输入元素名后按空格键会得到这个元素可用的属性名列表。不要小看这些细微处的快捷功能，熟悉这种快捷的输入方法后，编写 XML 文档的输入速度会大大提高

XML 模式支持绝大多数的与 XML 相关的文档类型：*.xml、*.xsd、*.xslt、*.wsdl、*.web、*.resx、*.tdl、*.wsf、*.hta、*.disco、*.vsdisco 和*.config 类型的文档。

Microsoft Visual Studio.NET 7.0 要求的配置环境可以参考本书第 2 章 ASP.NET 的开发和运行平台的建立。

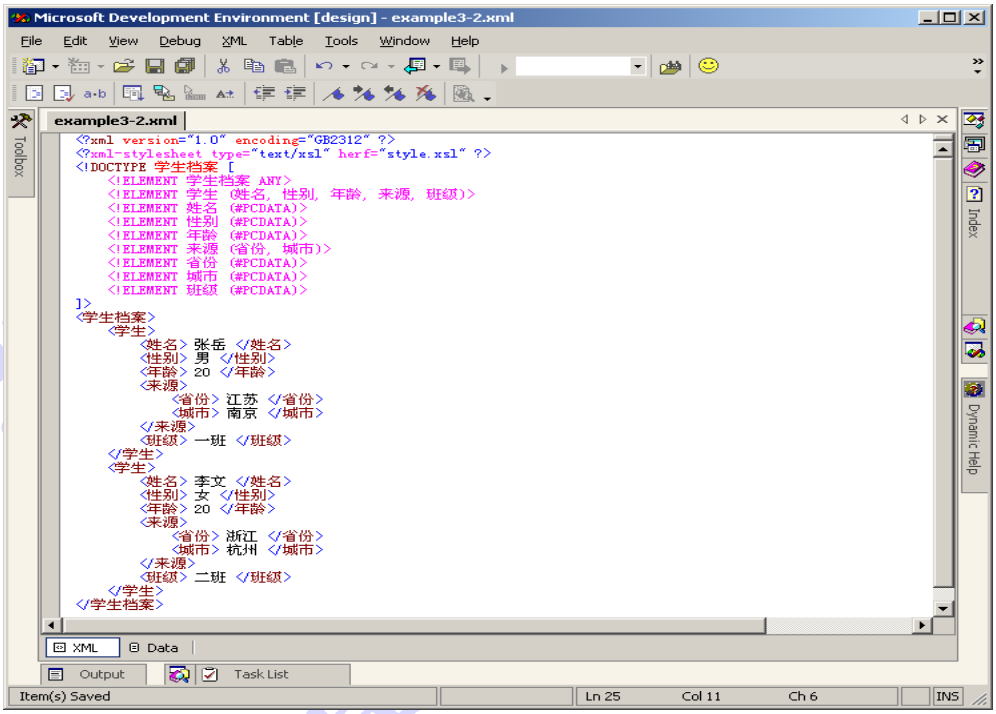


图 3.7

3.6.2　XML Spy

除了 VS7 之外，我们还向你推荐一个很好用的 XML 编辑工具——XML Spy。在国内主要的 XML 技术站点，XML Spy 的评价很高。XML Spy 是 I 由 con Information-Systems 公司开发的。XML Spy 提供了集成的开发环境 IDE，支持 Unicode、多字符集；支持格式良好的（well-formed）和有效的（validated）两种类型的 XML 文档。

XML Spy 还可以创建和编辑 DTD、Schema 和 XSLT 等类型的文档。

目前，XML Spy 的最新版本是 4.0b2。用户界面如图 3.8。这是 XML Spy 网站上提供的插图。

XML Spy 已经发展成为一个产品系列，包括 XML Spy 4.0 Integrated Development Environment（IDE）和 XML Spy 4.0 Document Framework 两部分。

其中 IDE 部分是在保留了 XML Spy 3.5 的成功之处的基础上，增加了对扩展 ODBC 数据库的访问支持；增强了用户界面；为第三方开发者提供了新的插件结构；支持 XML Schema 最终推荐标准。全新的用户界面，风格类似于 Office XP，如图 3.8。IDE 的新特性包括：

- 支持 W3C 于 2001 年 3 月 2 日发布的 XML Schema 的最终推荐标准
- 自动转换 Schema 版本，将以前的 Schema 版本转换到最新的标准
- 可以将已有的 DTD、XML-Data 或是 BizTalk Schema 转换为 W3C XML Schema Definition Language
- 从任何的 ADO 或是 ODBC 可访问的 SQL 数据库（包括数据类型和数据关系）自动产生 XML Schema

- 通过 import/export 函数扩展数据库连接性
- 基于 ADO SHAPE 的层次型数据库
- 可以自定义用户界面，比如菜单、键盘的快捷方式等等
- 新的插件结构允许第三方扩展 XML Spy IDE

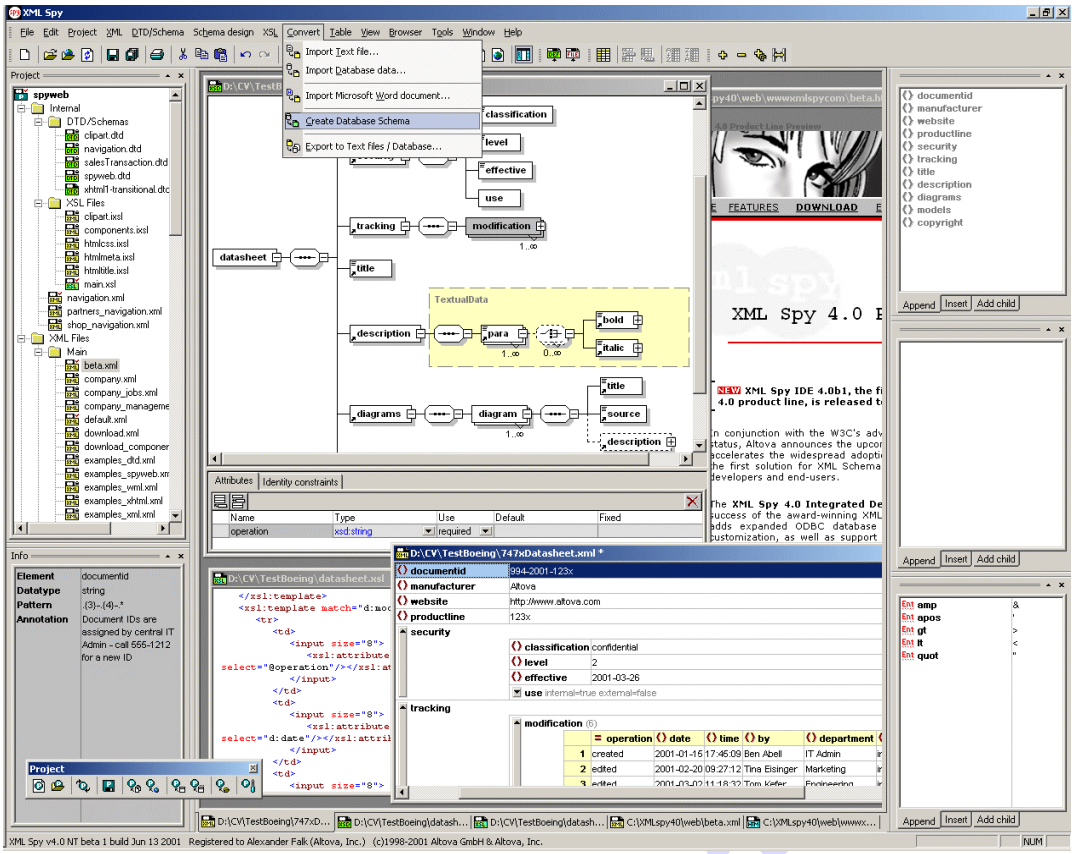


图 3.8

Document Framework 部分包括了两个独立的应用程序：

XML Spy 4.0 Document Editor：支持 WYSIWYG 的文本编辑功能，同时还具有其他，如使用不同的颜色显示不同的元素、基于表单的数据输入等功能。Document Editor 可以单独存在、集成在 IDE 中或是作为 Internet Explorer 的插件。

XML Spy 4.0 XSLT Designer：是一个图形化界面的创建样式单的工具。使用 XSLT Designer 可以指定文本编辑器编辑 XSLT，还可以选择根据 DTD 还是 XML Schema 来创建和编辑 XSLT 文档。

XML Spy 提供了五种视窗模式，分别为：

1. 增强数据视窗

在这个视窗中，通过嵌套关系来显示文档结构，同时，我们可以直接在这个视窗中添加新元素、属性或是扩展已有的元素、属性。这个视窗还有一个高级功能——支持拖拽。

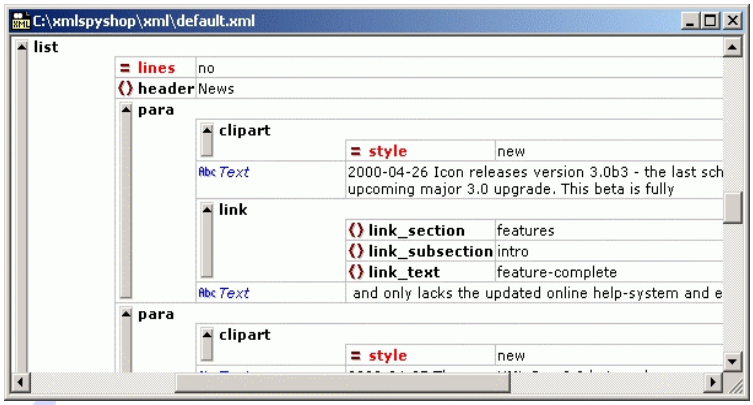


图 3.9

2. 数据视窗

在很多 XML 文档中，你会遇到一系列同类型的有顺序的元素，XML Spy 会自动将这种序列重新排列，以使这些元素以表格形式显示。

为了更容易地和其他应用程序交换数据，还可以将表中的元素直接拷贝粘贴到 Access 或是 Excel 中。

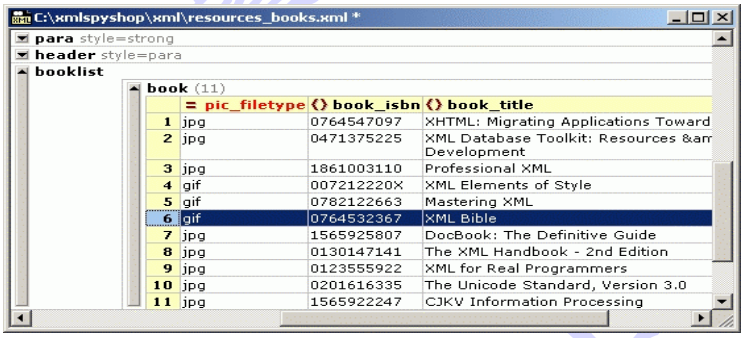


图 3.10

3. schema 设计视窗

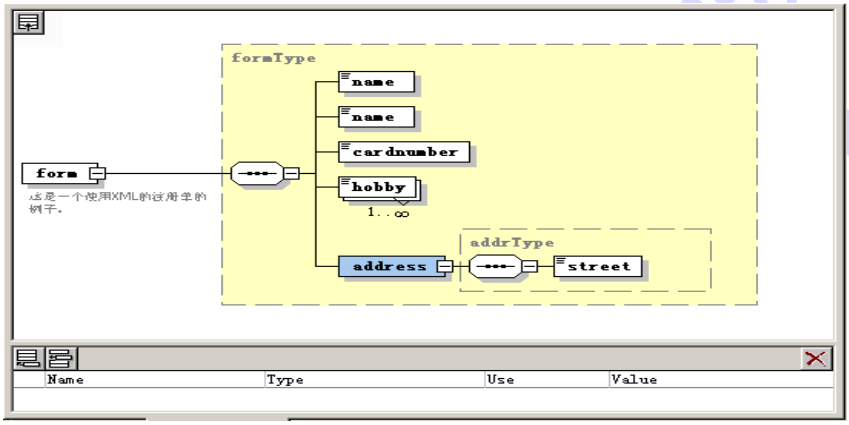


图 3.11

这是一个可以将 Schema 以独特的树形结构显示出来的视窗。在这个视窗中，可以拖拽和编辑 Schema 文档。

4. 文本编辑视窗

图 3.12

适合编辑各种文档源文件的窗口。在文本编辑视窗中，会为不同的内容标记不同的颜色。还可以提供可用的提示信息。

5. 浏览器视窗

内置的 XSL 和 CSS 的浏览器预览视窗，还可以显示使用 XSLT 转换得到的 HTML 文档。这个集成的浏览器需要 Microsoft Internet Explorer 5 支持。

XML Spy 还提供了工程（project）的开发模式。这里限于篇幅，我们就不详细介绍了。在实际的使用中，你会发现其功能的强大。

3.6.3 XML writer

我们再向你推荐一个功能强大的 XML 编辑器，由 Wattle Software 公司开发的 XML 编辑软件——XML writer。目前，XML writer 的最新版本是 1.21。

XML writer 与 XML 相关的特性有：

- 根据 DTD 或是 Schema 验证 XML 文档的有效性
- 检查 XML、XSL 或是 DTD 文档的格式是否良好
- 可以使用第三方的工具。比如 IBM 的解析器 XML 4C parser 解析文档
- 采用 TagBar 显示 XML Schema 文档中的元素、结构等信息。可以拖拽
- 提供命令行工具验证 XML 文档和转换 XSL 样式单
- 可以根据支持的文档模板创建新文档

项目管理特性包括：

- 项目视窗，以树形结构管理项目文档
- 可以实现批处理验证 XMLXML 和转换 XSL 样式单

XML writer 的用户界面如图 3.13，风格和 Visual Studio 的界面类似。XML writer 不支持 WYSIWYG，需要使用第三方浏览器才可以预览编辑页面。

XML writer 是用 C++编写的，所以比一些基于 Java 的编辑器在运行方面速度快，效率高。XML writer 是 32 位的应用程序，可以运行于 Windows9x 和 Windows 2000，以及 Windows NT4 平台。

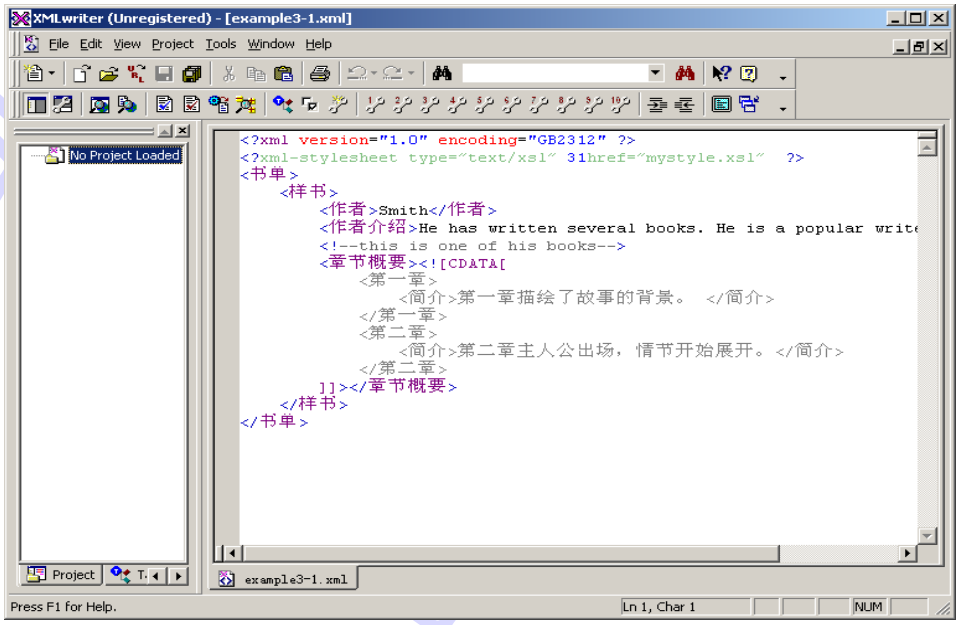


图 3.13

3.7　本章小结

本章的内容比较重要，可以说是本书一个非常重要的基础，因为它基本上介绍了 XML 的所有内容，从 XML 的基本语法到文档类型定义（DTD），从 Schema 描述机制到 XSL 扩展样式单，涵盖面较大。另外，本章最后还介绍了 XML 编辑器的问题，希望对读者有所帮助。

-第 4 章　ASP.NET Web Forms（网络表单）

本章介绍了 ASP.NET 的组成部分之一——Web Forms 技术。Web Forms 的网页代码与老版本的 ASP 很相似，本章将会简要地介绍必需的 Server 端动态页面的基本知识，以便没有接触过以前的 ASP 或 PHP 技术的读者为以后的学习做好准备。然后我们会将重点集中在 ASP.NET 技术与 ASP 技术的不同之处。有过 ASP 编程背景的读者会比较快地理解这种新的脚本组织和运行方式。当然如果是服务器端动态网页编程的入门者，也可以从这些讨论中获取关于 ASP.NET 的知识，并同时思考为什么 ASP.NET 要做这样的改变，从而更好地把握技术发展的方向。

4.1　Web Forms 简介

ASP Web Forms 是 ASP.NET 两项主要技术之一，使用 Web Forms 可以创建可编程的网页。

4.1.1　概述

首先，基于服务器端的动态页面技术，可以在服务器端根据客户端请求和提交的信息，动态地生成与客户端浏览器类型无关的通用 HTML 页面，也可以根据客户端浏览器类型，生成特定的数据表现形式。

为了更好地理解这一特性，让我们设想有一个手机和一个家用 PC 访问某个 ASP.NET 建成的网站的某个数据页面，例如：查询 Pizza 饼的售价和送货信息。对于我们用惯的 PC 来讲，它的屏幕可以很大，一屏可以容下四个 Pizza 饼的图片和其他信息；但是如果是手机用户也接收到同一个页面，抛开无线传输的开销不讲，其屏幕的大小也远不能满足该信息容量。如果潜在的客户为了看到页面中下一个 Pizza 饼的售价而必须花上几分钟的滚屏时间，客户大概会放弃该服务。

但是不必担心，我们的网站可以使用 Web Forms 来生成网页。我们设计的程序会根据浏览器类型（例如手机型号、屏幕信息）来生成合适的页面。例如对于上面的手机，我们会生成没有带图片的页面，对于字体的大小和风格也可以进行调整。当然运用以前的动态页面技术也可以做到这一点，但 Web Forms 赋予页面技术更强的编程能力，使创建这样的页面应用变得更加简单、易于代码的维护、开发过程的管理和系统的扩展。

第二，可以使用任何 CLR 支持的编程语言（例如现阶段支持的 C#、VB、JScript）进行开发，这些语言是完全面向对象的，程序员可以充分利用面向对象的特性，控制能力更强。

基于面向对象技术开发的 ASP.NET 代码可以利用该技术的继承、封装和可重用性。并且由于 CLR 的支持，不同语言的 ASP.NET 代码将是可以互操作的。设想一个 VB 程序员编写的 ASP.NET 控件（我们会在第五章提到如何创建自己的 ASP.NET Web 服务器端控件）

可以由另一个 C#程序员编写的 Web Form 调用，这将是多么激动人心的事！

第三，在 ASP.NET 可以使用许多.NET Platform 提供的类型和服务（例如类型安全、受控执行、即时编译），从而简化开发的工作，提高开发和运行时的效率。

第四，支持代码和页面内容的分离。

回想编写 ASP 的日子，当一个 ASP 页面变得很大时，往往很难从页面中看出程序的逻辑，因为 HTML 标记和 ASP 脚本代码是混杂在一起的。

第五，支持 WYSIWYG 编辑器。ASP.NET 代码的结构化特性使得识别这些代码变得简单。ASP.NET 程序员和 HTML 美工人员将可以对 ASP.NET Web Forms 进行可视化的编辑工作而不必担心编辑器自动添加或修改的代码会影响到脚本程序的运行。

第六，提供了更好的状态机制来使判断、获取请求间的状态更加方便，简单甚至无需干预。

4.1.2　两种代码写的实例简单比较

好了，现在让我们来看看 ASP 和 ASP.NET 的一些代码。

例如以下 ASP 代码（左边是为了方便分析而加入的行号，不是代码的一部分）：

程序清单 4-1：example-4-1.asp

```
00        <select size="1" name="Month">
01        <%
02             For i = 1 To 12
03                 strbirth = "<option"
04                 If month(rs.Fields("Birthday")) = i Then
05                     strbirth = strbirth & " selected"
06                 End If
07                 strbirth = strbirth & ">" & i & "</option>"
08                 Response.Write strbirth
09             Next
10        %>
11        </select>月
12        <select size="1" name="Day">
13        <%
14          For i = 1 To 31
15             strbirth = "<option"
16             If day(rs.Fields("Birthday")) = i Then
17                 strbirth = strbirth & " selected"
18             End If
19             strbirth = strbirth & ">" & i & "</option>"
20             Response.Write strbirth
21          Next
22        %>
23        </select>日</p>
```

这 23 行代码假定已经使用 ADO（注意不是 ADO.NET）从某个数据源获取了一个记录集 rs，然后生成两个下拉控件显示某“月”某“日”（第 11 行和第 23 行）。作为真正的程序员，我们习惯于懒惰而不喜欢重复同样的事情，所以我们使用程序代码生成 HTML 代码（第 02 行到第 09 行、第 14 行到第 21 行）。

提示： Lary Wall（Perl 语言的发明者）说过，懒惰、缺乏耐心、骄傲自大（Laziness，impatience，hubris）是程序员的三个优良品质。

注意： 细心的读者会发现这些代码里有不合理的地方，或者说有错误。在产生“日”的列表项目时，并没有考虑每个月的天数不同！当然我们可以在用户修改选择项后提交时在客户端进行验证，不过这只是个很无奈的折衷办法。绝对不推荐这么做！我们应当在生成 HTML 代码时就产生合适的解决方法，例如在客户端相应“月”列表框的更改事件中，根据不同的月份调整“日”列表框。为了节省篇幅并将注意力放在我们要讨论的话题上，我们没有写下这些代码。

另外，我们要把下拉控件的值预先置为 rs 记录里的出生日期（04 到 06 行、16 到 18 行），这样子使用这两个控件就可以用来显示信息和修改信息了。

这样的代码问题在哪呢？

首先，ASP 脚本代码是分散地嵌入到 HTML 代码中的，这样子很难看出程序的流程，尤其当代码很长的时候，你将很难找到与<% Loop %>对应的 While 在哪。

第二，输出的 HTML 代码既可能来自文件中的 HTML 代码，也可能是 ASP 脚本中 Response 对象的输出（第 08 行和第 20 行）。这样将会很难预料产生的 HTML 代码会是怎么样的。比如在 Response.Write 的时候多写了一个“>”号……

第三，代码是不能重用的。下次为了完成同样的功能，我们还得找到以前的代码，复制，粘贴……甚至还得替换变量名，因为这里面没有程序员熟悉的函数库、类库、名字空间之类的东西来避免命名的冲突和方便代码的封装和重用。

第四，ASP 代码经常依赖于 HTML 制作人员的工作。ASP 代码无论是在被解释运行时还是在编写时，都似乎是“嵌入”到 HTML 里面去的。ASP 程序员往往要等 HTML 制作人员把页面的模板编写好后才可以开始往里面填充代码。

第五，带有 ASP 页面是不能被 WYSIWYG（What You See Is What You Get）编辑器编辑的。因为编辑器难以解析这种面条似的页面代码。

如果我们使用 ASP.NET 来实现相同的功能呢？看看下面的 C#写的代码：

程序清单 4-2：example-4-2.aspx

```
00              <script Language="C#" runat=server>
01              public void Page_Load(Object sender, EventArgs e)
02              {
03                  ArrayList items = new ArrayList();
04                  for (int i = 1; i <= 12; ++i)
05                      items.Add(i);
06                  listMonth.DataSource = items;
07                  listMonth.DataBind();
08                  items.Clear();
09                  for (int i = 1; i <= 31; ++i)
10                      items.Add(i);
11                  listDay.DataSource = items;
12                  listDay.DataBind();
13                  // get the Recordset object rs first
14                  listMonth.Selected = Month
                    (rs.Fields("Birthday"));
```

```
15                  listDay.Selected=Day(rs.Fields("Birthday"));
16              }
17              </script>
18
19          <html>
20              <p>This is an ASP.NET version</p>
21              <asp:DropDownList id="listMonth" runat=server>
22              </asp:DropDownList>月
23              <asp:DropDownList id="listDay" runat=server>
24              </asp:DropDownList>日
25          </html>
```

可以看到这个 ASP.NET Web Form 的代码大致上可以分为两个段落：<script>标记和<html>标记。事实上 ASP.NET Web Forms 的确是将一个 Web 应用程序的用户界面分为两个部分：可视化的组件和用户界面逻辑。

可视化组件部分包含了各种标记和 Web Forms 特有的元素，这些组件构成了一个容器，用来容纳要显示的文本和控件。这个容器被称为页面，用后缀为 aspx 的文件名存储（也就是浏览者请求的 ASP.NET 页面）。

用户界面逻辑包含了用于和可视化组件部分交互的代码。用户界面逻辑可以放在 aspx 文件里，也可以单独地放在其他文件里。第二种方式会是更普遍并且是推荐的做法，因为这样有利于代码的重用和维护。我们会在下一节介绍这两种代码组织方式。

在这个例子中，很明显<script>和</script>标记间是我们的用户界面逻辑代码，<html>和</html>间是可视化组件部分，这里面 、 是传统的 HTML 标记，而<asp:DropDownList>则是 Web Forms 的控件。我们将这两部分放在一个 aspx 文件中（example-4-2.aspx）。

可以看到，可视化组件部分中描述了各个页面元素（在这里是 标记和两个 asp:DropDownList 控件）。然后在用户界面逻辑中与各个页面元素进行交互。在这个例子中，我们在一个特殊的过程（在 VB 中没有返回值的子程序称为过程，在 C/C++/C#中，所有子程序都称为函数或者方法）Page_Load()中对两个 DropDownList 控件进行初始化，在其中填充各个选项。在这里我们使用一个 ArrayList 类对象 items 来容纳将要添加的值，然后设置 DropDownList 控件的 DataSource 属性为 items，使 items 成为 DropDownList 的数据来源，最后调用其 DataBind()方法完成填充。

这里，Page_Load()是个特殊的函数，会在每次载入页面时被系统调用。Page 对象有个 IsPostBack 属性，利用它可以判断该页面是否第一次被加载，从而选择初始化页面或是对用户的输入进行处理。

ArrayList 类则是.NET Class Library 里的类。

具体的 ArrayList 类和 Page 类的成员列表、使用方法等，请参阅.NET Framework SDK 文档。本书不会详细介绍每个类的用法。关于数据绑定的内容，我们在以后会有进一步的讨论。

注意：示例代码 4-1 和 4-2 都不能不加修改的运行，因为没有正确处理数据库和 rs 对象。ASP.NET 和 ASP 处理数据的方式也有很多不同，但给出这两个例子只是为了讨论两者的代码组织方式。如果有兴趣运行第二个示例，注释掉或删除第 14、

15 行，便可运行了。

为了以后阐明 aspx 文件的基本结构方便，这里给出上面 4-2 示例代码的代码分离形式。界面逻辑代码和界面元素分离在不同文件中的做法称为 Code-behind。这两个示例代码会在本章后面的介绍中用到。

程序清单 4-3-1：example-4-3.aspx

```
00  <%@ Page Inherits=CodeBehindExample
Src="example-4-3-code.cs"%>
01
02  <HTML>
03      <H3>Example for Code-Behind Web Forms</H3>
04      <FORM RUNAT=SERVER>
05          <asp:Button id=btnClick runat="server"
06             Text="Click Me"
07             onclick="btnClick_Click" />
08            
09          <asp:Label id=lblState runat=server />
10      </FORM>
11  </HTML>
```

程序清单 4-3-2：example-4-3-code.cs

```
00  using System;
01  using System.Web.UI;
02  using System.Web.UI.WebControls;
03
04  public class CodeBehindExample : Page
05  {
06      public Label lblState;
07      public Button btnClick;
08      public void btnClick_Click(Object Source, EventArgs e)
09      {
10          lblState.Text="Button is clicked";
11      }
12  }
```

这两个文件结合在一起实现了前面与 4-2 示范代码同样的功能（不包括从数据集里读生日并改变两个下拉框的选择项）。

example-4-3-1.aspx 文件仍然是用户请求的入口文件，在浏览者看来，aspx 文件并不会因本身包含代码与否而有所不同。

4.2　ASP.NET Web Forms 的代码模型

ASP.NET Web Forms 包括两种类型的元素：标记和代码。这两种类型的组件组成了特定的 Web Form 单元。在页面被请求时，相应的 Web Form 会被编译，ASP.NET 解析这些标记和代码并动态地生成一个由 Page 类继承而来的新子类，并将其编译（成为 MSIL 代码，请回忆第一章的讨论）。

提示：Page 类在 System.Web.UI 名字空间中定义，请参阅 SDK 文档来找到更多关于 Page 类的信息。另外这样的编译工作并不是每次页面被请求时都会进行的。请

留意以后的 ASP.NET Cache 机制的讨论。

编译成的这个新的 Page 子类变成了一个.NET 可执行文件。当页面被请求时，该执行文件会由服务器执行（请回忆 JIT 机制）并根据请求内容和输入，产生相应页面的 HTML 代码，经由 Web 服务器发送回浏览器。

附带说明一下，.NET 可执行文件使用标准的 PE 文件头，CTS 编译器在编译时，会产生 MSIL 代码，并调用 JIT 编译器的一条跳转指令作为 PE 可执行文件的入口，PE 可执行文件的后部不是本地机器码，而是产生的 MSIL 代码。在运行这些可执行文件时，操作系统进入文件的执行入口点后，CLR 环境的 JIT 编译器便会得到控制权，对文件后部的 MSIL 代码进行实时编译，生成本地机器码后再由操作系统执行它们。因此，尽管我们说.NET 编译器产生的是 MSIL 代码而不是本地机器码，但对于.NET 编译器生成的可执行文件，我们仍可以像运行普通的 PE 可执行文件一样来运行它，JIT 编译器在背后起的作用，是由标准 PE 文件头中嵌入特殊的跳转指令来实现的。

因此，aspx 文件并不是一个严格意义上的可编译模块。虽然用户请求页面时通常是请求 aspx 文件（例如在浏览器中敲入 www.somesite.com/default.aspx），但在编译该页面的 Page 类（严格的说是一个由该页面的 ASP Web Forms 代码决定的 Page 类的子类）时，编译器的输入不仅是 aspx 文件，还可能有其他的代码文件。如果代码是分开放在其他文件中的话，会要求其源代码文件；如果使用到了类似于 AdRotator 之类的控件，会要求相应的 XML 文件。

因为 aspx 文件并不是严格意义上的一个独立模块，它需要其他的输入，例如程序代码。因此 aspx 文件使用“指令”（Directives）来标记外部的代码和 XML 资源。对于最常用的程序代码来说（因为将页面内容框架和代码分离是一种非常值得鼓励的做法），我们在@Page 指令的 Inherits 属性中指定使用的代码文件名。我们会在下一节看到一个具体的例子，那时我们会更清楚这种代码组织方式是怎么实现的。

4.3　ASP.NET Web Forms 的页面处理过程

在页面被请求时，服务器会编译并执行页面对应的 Page 类的可执行文件。Page 类对象的可执行文件在被执行时会有初始化、处理、生成 HTML 代码、释放对象四种阶段。但要注意，Page 类对象的生命周期其实只包括初始化、生成 HTML 代码、释放对象三个阶段。处理阶段其实是另一次页面请求。

每次页面被请求，都开始一个 Page 类实例对象的初始化过程。初始化时服务器会产生 Page_Init 事件和 Page_Load 事件，在 Page 类中相应的事件处理函数为 Page_Init()和 Page_Load()，我们可以在这两个事件处理函数中做许多事情，例如像示例 4-2 中一样对页面控件进行初始化。初始化过程结束后，服务器便执行 Page 类实例对象的某个方法来产生 HTML 代码。在生成 HTML 代码后，Page 类对象会被撤销。相应地，释放对象时会产生 Page_Unload 事件并调用 Page_Unload()事件处理函数，程序代码可以在这个函数中作一些事情，例如释放自己请求的资源。ASP.NET 服务器会将生成的 HTML 代码以 HTTP 等网络

流的形式发送给客户端。

客户端在接受到 HTML 代码后，可能对这些代码做出响应，除了单击某些外部链接外，还可能触发页面中控件的事件，例如按下页面中的某个按钮。Page 类可执行文件中可以包含代码来响应这些事件。

动态页面技术的事件响应的实现与桌面应用程序的事件响应有所不同。在本地机环境中，往往使用回调函数（Callback）之类的技术将事件触发点和事件响应者联系起来；在 Web 网络环境中，这种方法不再适用，我们将事件作为一种消息，以此来通知处于网络另一端的机器需要对此做出响应。在动态页面技术中，这种消息往往是一次 HTTP 请求，因此，事件响应和处理一次页面请求并没有本质上的不同。ASP.NET 的动态页面技术同样使用 HTTP 请求来实现事件的响应，某个页面或页面中控件的事件响应代码和页面的生成代码同样处于 Page 类代码中。当页面上某事件被触发时，页面产生的 HTML 代码会指引浏览器向页面所在的服务器发送包含事件信息的 HTTP 请求，服务器负责从 HTTP 请求报文中解析出特定事件的信息，并将这些信息作为页面 Page 类可执行文件输入，开始又一个 Page 类对象的初始化——生成 HTML 代码，撤销对象的生命周期。这个生命周期和页面第一次被请求时的生命周期存在微小的不同，因此在页面的 Page 对象初始化后，便具有一个 IsPostBack 属性，该属性指明了页面是第一次被请求还是作为一个“已存在过”的页面的回传响应被请求。程序员可以利用该属性判断页面的行为。注意这个属性是在页面对象被初始化后具有的，因此在页面 Page 类的构造函数中是不可用的。

某些响应客户端时间的代码适合在客户端机器上运行从而减少网络传输的开销，例如用户提交的数据的验证；但更多的是在服务器端的事件处理代码在起作用，ASP.NET 使得响应用户事件的工作更加简化和结构化。

在 ASP.NET 页面回传时，HTTP 请求报文中不仅会包含事件的信息，例如事件的名称、触发源、参数等，还可能包含页面的状态。这是 ASP.NET Web Forms 技术的另一优势。在 ASP.NET 中这种技术称为 View State（视图状态），具体来说，ASP.NET 会在生成的 HTML 页面中使用标准的 HTTP-GET/HTTP-POST，将页面的状态（例如文本框的内容）放在 HTTP 请求报文中回传给服务器，服务器除了解析事件消息外，还会在构造页面和页面上的组件（例如刚才的文本框）时，根据这些信息来初始化组件对象。因此，程序员不必像从前那样，手工地将需要保留的页面和组件的状态作为 POST 方法的一个域（field）或者 GET 方法的一个参数回传给服务器，然后又必须手工地从 Request 对象中提取这些参数，自己初始化那些控件的状态。在 ASP.NET 中，这一切是自动完成的。并且，程序员可以在除了页面构造函数外的其他程序代码中以“object.property”的形式访问和修改这些状态。

生成 HTML 代码是 ASP.NET 相对 ASP 而言特有的。ASP 的程序代码是嵌入在 HTML 代码中的，在页面被请求时，解释器（而不是编译器）解释运行这些程序代码并将产生的结果（数值、字符串等等）或 HTML 代码替换插入到“HTML 海洋”中。因此 ASP 并没有产生整个 HTML 代码。而对 ASP.NET 而言，该过程刚好相反，HTML 代码是由可执行文件产生的，可以视为该可执行文件的输出。

4.4　ASP.NET Web Forms 的结构和基本语法

4.4.1　aspx 文件的基本结构

aspx 文件除了 HTML 标记外，还可能由以下部分的一个或多个构成：

- 指令（Directives）
- 代码声明段（Code Declaration Blocks）
- 内联代码段（Code Render Blocks）
- 服务器端包含指令（Server-Side Include Directive Syntax）
- 服务器端的注释（Server-Side Comments）
- 定制的服务器端控件（User Server Controls）
- 服务器端控件的内联模板
- 数据绑定表达式
- 服务器端对象标记

在这里我们先介绍最常用到的几个部分的最常用语法和属性（指令、代码声明段、内联代码段、服务器端包含指令、服务器端注释），其他语法（数据绑定表达式、服务器端对象标记等）我们留待以后需要时再讨论，或者留给读者进一步学习时自己查阅有关文档。其中，定制的服务器端控件在 5.2.4 节中详细介绍。

1. 指令（Directives）

指令部分可以放在页面文件的任何地方，作为习惯，通常我们会将指令放在文件的开头。当然指令部分不是必需的，例如示例程序 4-2 就没有指令代码。示例程序 4-3-1 中，指令代码放在了第 00 行，其中只包含了一条@Page 指令：

```
00  <%@ Page Language="C#" Inherits=WSCodeBehindExam Src="example-4-3-code.cs"%>
```

每一种指令中可以包含对应的属性及合适的值，视指令类型而定。指令的大致格式为：

```
<%@指令 属性=值 属性=值 …… %>
```

现在我们会用到的 aspx 文件指令类型有以下几种：

（1）@Page

定义了该页面被解析器和编译器处理的方式。每个 aspx 文件只可包括一条@Page 指令。定义多个属性时，以空格分开各个“属性=值”对，注意每个属性和相应的值间的等号任何一边均不可有空格。

@Page 指令是 ASP.NET Web Forms 的缺省指令，因此该指令可以写成：

```
<@ 属性=值 … %>
```

即省略 Page 不写。

@Page 指令支持的属性包括：

- AspCompat：取值 true/false。设为 true 可以使该 Page 类运行于一个 STA（single-threaded apartment）线程，可以调用其他 STA 组件等等，但带来的代价是

较低的性能。缺省值为 false。

- AutoEventWireup：取值 true/false。指定页面的事件是否自动传送。缺省为 true（自动传送）。
- Buffer：取值 true/false。指定 HTTP 响应缓冲是否有效。缺省为 true（缓冲有效）。
- ClassName：指定页面被请求时被自动动态编译的类。
- ClientTarget：指定目标用户代理或其别名，以便 ASP.NET 服务器控件为该代理程序生成相应的内容。
- CodePage：如果使用了运行页面的 Web 服务器上的缺省代码页之外的代码页，则必须指定代码页。该代码页的值必须存在于创建该页的机器上。
- ComplierOptions：编译器选项。可以是任何编译器能识别的选项字符串，从而指定编译该页时的行为。
- ContentType：指定响应时的 HTTP content type。可以是任何合法的 HTTP content type 字符串。
- Culture：指定页面的文化设置。请参考.NET Class Library 中关于 CultureInfo 类的信息。
- Debug：指定编译该页时是否加入调试信息。程序设计人员会很熟悉什么是“调试信息”（debug symbols）的。
- Description：可以是任何用来描述该页的字符串。这一属性会被解析器忽略。
- EnableSessionState：取值为 true/ReadOnly/false。指定该页是否及如何使用 Session 状态。设为 true 时该页可以完全使用 Sessiong 状态对象；设为 ReadOnly 使得 Session 对象可以被该页读取，但页面不能改变 Session 对象；设为 false 时该页无法使用 Session 对象。缺省值为 true。关于 Session 对象是什么及如何使用该对象，我们会在讲状态管理时详细介绍。
- EnableViewState：取值 true/false，指定视图状态是否在页面请求过程中保留，缺省为 true。
- EnableViewStateMac：取值 true/false，指定是否在页面从客户端回传时运行机器身份正确性检查。选择 true 时，页面的视图状态中被加入了一个隐藏的加密过的变量值，在页面被回传时服务器会对该变量进行检查，确定页面状态是否被改动过。
- ErrorPage：定义了一个重定向的目标 URL，当一个没有被处理的页面意外出现时，浏览器会重定向到该 URL。可以用来显示友好的出错信息。
- Explicit：指定页面是否使用 Visual Basic 的 Option Explicit 模式来编译。可能的取值为 true/false，缺省为 false。
- Inherits：定义一个 code-behind 类，页面会继承该类。这个类可以是任何由 Page 类继承而来的类。我们会详细介绍这个属性的使用。在我们前面的示例 4-3-1 中的指令部分，包含了这个指令。

```
    00  <%@ Page Language="C#" Inherits=WSCodeBehindExam
Src="example-4-3-code.cs"%>
```

- Language：指定内联代码和代码声明段中代码使用的语言。可以是表示任何.NET 支持的语言，包括 Visual Basic，C#，Jscript.NET 等。在示例 4-3-1 中，我们指明使用了 C#语言。

注意：请回忆什么是“.NET 支持的语言”，参见第 1 章中的讨论。在以后的示例中，我们将一直使用 C#语言。关于 C#语言的语法，请参阅附录。

- LCID：定义该页面的本地化代码。ASP.NET 会使用 Web 服务器上缺省的本地化代码，除非在页面中指定了使用其他本地化代码。关于本地化代码的详细信息，请查阅 MSDN。
- ResponseEncoding：指定页面内容的编码方法。
- Src：指定页面被请求时一同被动态编译的 code-behind 类的源文件名。这个属性有时会和 Inherits 属性连用，但两者有所区别。我们稍后就会说明。
- Strict：指定页面被编译时是否使用 Visual Basic 的 Option Strict 模式。取值为 true/false，缺省为 false，即不使用该模式。
- Trace：指定页面跟踪是否有效，用在调试页面时。取值 true/false，缺省为 false。
- TraceMode：指定当 Trace 属性为 true 时跟踪信息的显示方式。可能的取值有 SortByTime/SortByCategory。当跟踪模式有效时，缺省的显示方式为 SortByTime。
- Transaction：指定在页面上 transaction 是否被支持。可能的取值有 NotSupported/Supported/Required/RequiresNew。缺省为 NotSupported。
- WarningLevel：指定编译器编译该页时的警告敏感程度（程序员会很熟悉这个概念），可能的取值是 0-4。关于各水平的详细情况，请参阅 ComplierParameters.WarningLevel Property 属性。

（2）@Import 指令

显式地导入一个名字空间，使页面可以使用该名字空间中的类和接口。这些名字空间可以是.NET Framework Class Library 的一部分，也可以是用户自定义的其他名字空间。每条@Import 指令只可引入一个名字空间。为了引入多个名字空间，可以使用多条@Import 指令。指令的格式为：

```
<%@ Import Namespace="value" %>
```

其中 value 为要引入的名字空间。

例如：

```
<%@ Import Namespace="System.Xml" %>
<%@ Import Namespace="System.Data.OleDb" %>
```

注意：System.Data.OleDb 名字空间在.NET SDK Beta 1 版中被改名为 System.Data.ADO。

以下的名字空间为所有页面均已自动引入的：

System

System.Collections

System.Collections.Specialized

System.Configuration

System.IO

System.Text

System.Text.RegularExpressions

System.Web

System.Web.Caching

System.Web.Security

System.Web.SessionState

System.Web.UI

System.Web.UI.HtmlControls

System.Web.UI.WebControls

2. 代码声明段（Code Declaration Blocks）

代码声明段定义了成员变量和方法，这些成员变量和方法是表示 ASP.NET 页面和用户控件的 Page 类或 UserControl 类的成员。

提示：使用用户控件是另一种更灵活、可重用性更高的 Code-behind 代码组织方法。

代码声明段的格式为：

```
<script runat="server" language="codelanguage" Src="pathname">
    代码
</script>
```

属性 language 指定该代码声明段中代码所用的语言，其值可以是任何.NET 支持的编程语言，例如 Visual Basic(VB)、C#、Jscript.NET 等。如果没有指明语言，该属性的缺省值是页面中@Page 指令（如果该代码段存在于用户控件中，则是用户控件的@Control 指令）的 Language 属性所指明的语言。如果仍没有指明，缺省为 VB。

属性 src 指定了外部脚本文件的路径和文件名，使用该属性时，外部脚本文件的代码会被引入并在页面被请求时被编译，而该代码声明段中的所有代码（<srcipt>和</script>标记间的代码）则都会被忽略。

使用 src 属性时，因为<script>和</script>标记间的内容将是无意义的，该指令可以写成：

```
<script runat="server" src="externalscript.vb" />
```

externalscript.vb 是个设想中的要引入的 VB 编写的源代码文件。

3. 内联代码段（Code Render Blocks）

内联代码段定义了内联代码或内联表达式，当页面在生成 HTML 代码的阶段这些代码和表达式被执行。

内联代码的格式如下：

```
<% inline code %>
```

内联表达式格式如下：

```
<%=inline expressiong %>
```

内联代码段的用法和以前版本的 ASP 几乎一样。内联代码使用 Response 对象输出 HTML 代码，在任务简单时，可以使用内联表达式来简化输出。在此我们不再详述。

值得注意的是，在内联代码段中，企图输出字符串“%>”将会在编译时出现错误。例如这样的代码：

```
<%
    Response.Write("%>");
%>
```

编译器将会报错。为了达到目的，我们可以使用这样的代码：

```
<%
    String s = "%" + ">";
    Response.Write(s);
%>
```

4. 服务器端的注释（Server-Side Comments）

使用服务器端注释在 aspx 文件中加入注释。其格式为：

```
<%-- comments --%>
```

在代码声明段<script runat="server"> … </script>和内联代码段<% … %>中，可以使用代码的语言所允许的注释语法，而不使用服务器端注释语法。例如：

```
<script runat="server" language="C#">
// C# line comment
/*
        C# block comment
*/
</script>
```

或者

```
<script runat="server" language="VB">
    'VB comment
</script>
```

注意在<% ... %>之间不能使用服务器端注释语法。服务器端注释不能嵌套。

提示：注释可以用来对代码做出解释说明，还可以在测试代码时使暂时不用的或怀疑有问题的代码无效。

5. 服务器端包含指令（Server-Side Include Directive Syntax）

服务器端包含指令用来将指定文件的内容插入到 ASP.NET 页面中。语法为：

```
<!-- #include file|virtual = filename -->
```

属性 file 指定了要包含的文件在服务器上的物理路径，可以是绝对路径，也可以是相对路径，但必须是位于页面的同一路径下，或在页面所在的路径的子目录下。

属性 virtual 指定了一个网站上的虚拟路径，和 File 属性一样可以是绝对路径，也可以是相对路径。

属性值均为文件名 filename，并以引号包含。包含的文件在任何动态代码被执行前处理，类似于 C 语言中的#include 机制。

4.4.2　ASP.NET 服务器端控件（Server Controls）

创建 ASP.NET Web Forms 时可以使用服务器端控件（Server Controls），Server Controls

的概念总是和 Web Forms Server Controls 相一致的，因为它们是被专门设计来与 Web Forms 协同工作的。ASP.NET 中的服务器端控件概念与 Windows 编程中的控件概念并不完全一致，它们工作在 ASP.NET 框架中。

服务器端控件家族包括以下几种类型的控件：

- HTML Server Controls
- Web Server Controls
- Validation Controls
- User Controls
- Mobile Controls

服务器端控件拥有数据绑定特性，将服务器端控件绑定到某个数据存储上赋予了程序员几乎完全的控制来决定数据如何在页面上显示、修改并回传到数据存储上。

我们会在下一章详细地讨论服务器端控件。

4.4.3　ASP.NET Web Forms 的 Code-Behind 代码组织方式

Code-behind 的代码组织方式可以更好地将界面逻辑与页面内容分离，从而方便网络程序员和网页设计师的工作协调，可以开发出更好的 WYSIWYG（所见即所得）的 ASP.NET Web Forms 编辑器（例如 Visual Studio.NET 便带有一个极好的页面编辑器）等等。Code-behind 是.NET 战略中所要实现的 Programable Web（可编程 Web）的直接体现。

提示： Web 页面将不再仅仅是呈现信息的工具，而将更多地成为新一代程序设计的界面和信息传输的媒介。

Code behind 在 ASP.NET 中有多种实现方式。在这里我们将只介绍一种最常见的方式，如程序清单 4-3-1 和 4-3-2 所示。

我们在前面提到过，ASP.NET 的页面是一个 Page 子类，每一个页面都是在 Page 类基础上派生出来的，包含我们自己添加的 HTML 元素、服务器端控件、事件处理过程等等。页面被请求时该类的一个实例被服务器生成并执行，类代码的执行结果就是输出到客户端的 HTML 代码。

在这个示例中，我们的 Page 子类在 example-4-3-code.cs 文件中被定义：

```
00  using System;
01  using System.Web.UI;
02  using System.Web.UI.WebControls;
03
04  public class CodeBehindExample : Page
05  {
06      public Label lblState;
07      public Button btnClick;
08      public void btnClick_Click(Object Source, EventArgs e)
09      {
10          lblState.Text="Button is clicked";
11      }
12  }
```

注意这是一个 C#源文件，C#源文件以 cs 后缀结尾，编译器根据文件名后缀判断语言。

我们首先定义一个 Page 子类，名为 WSCodeBehindExample（第 04 行）。

由于我们要使用服务器端控件 Label 和 Button 类（在第 06 行和第 07 行声明了两个成员），这两个类包含在 System.Web.UI.WebControls 中；使用 Page 类（第 04 行），Page 类包含在 System.Web.UI 名字空间中；还要使用 Object 和 EventArgs 类（第 08 行，用来向事件处理函数传递参数），这两者包含在 System 名字空间中。因此我们在前三行导入了三个名字空间。

现在来看看 CodeBehindExample 的类成员。它包括了两个对象成员：lblState 和 btnClick，分别是 Lable 和 Button 类。还包括了一个成员函数 btnClick_Click()，我们用它来响应事件。关于 Web Forms 的事件模型，我们在下一节介绍，现在我们先把这个 CodeBehindExample 放下，记住它的成员就行。

转过来看 example-4-3.aspx 文件。这是一个 Web Form 的入口文件。

```
00  <%@ Page Inherits=CodeBehindExample
     Src="example-4-3-code.cs"%>
01
02  <HTML>
03      <H3>Example for Code-Behind Web Forms</H3>
04      <FORM RUNAT=SERVER>
05          <asp:Button id=btnClick runat="server"
06            Text="Click Me"
07            onclick="btnClick_Click"
08           />
09          <asp:Label id=lblState runat=server />
10      </FORM>
11  </HTML>
```

最重要的一行是第一行。

```
<%@ Page Inherits=CodeBehindExample Src="example-4-3-code.cs"%>
```

这里使用了前面提到过的@Page 指令，并指定了两个属性。Inherits 属性值为 CodeBehindExample，表示这个页面从 CodeBehindExample 类继承而来；Src 属性指定了 CodeBehindExample 类的文件。我们在这里省去了其他属性，以免混淆，当然前提是这个页面是可以运行的。

这样我们的最终 Page 子类就已经出现了。CodeBehindExample 从 Page 派生而来，增加了两个对象成员和一个函数成员。example-4-3.aspx 从 CodeBehindExample 派生而来，进一步定义了其父类成员 btnClick 和 lblState 的属性（第 05 到 07 行定义了 btnClick 的位置、Text 属性和事件处理函数，第 09 行定义了 lblState 的位置），并通过继承关系使从父类 CodeBehindExample 继承而来的成员函数 btnClick_Click 成为 btnClick 的 Click 事件的响应函数。

注意：这样的继承关系可能对于有面向对象背景的程序员来说比较容易接受，对于不熟悉面向对象理论的读者可能较有挑战性。但无论如何，这样的面向对象特性在 ASP.NET 乃至整个.NET 体系中都是贯穿始终、至关重要的，它是 Programmable Web 思想的根基。所以为了更好地理解这些新技术，请读者参阅其他面向对象理论的资料。

一个可能的问题是，为什么不指明 Language 属性呢？请参考前面对 Language 属性的

说明。这个属性声明的是所在 aspx 文件中的代码声明段和内联代码段的语言类型，而我们的 example-4-3.aspx 文件中并没有这些代码。那么编译器如何知道 Code-behind 代码的语言种类呢？答案仍然在前面：通过文件名后缀。

由此我们可以总结出一个编写 Code-behind 页面的方法。这种方法虽然在程序员看来过于死板，甚至有点愚蠢，但对于初次接触面向对象的新一代可编程 Web 程序设计的读者来说，可能会是一个入门的好方法。这个过程如下：

首先，编写只包括 HTML 元素的 aspx 文件，对需要在代码中操作的 HTML 元素（通常会是 Button、Label、TextBox、ListBox 等）赋予独一无二的 id 属性。通常，这个工作会由页面的美工设计人员来完成。

然后，编写一个从 Page 类继承而来的类，例如用 C# 编写的名为 ClassNameOfCodeBehindClass 的类声明为：

```
public class ClassNameOfCodeBehindClass : Page {
// …
}
```

注意：C#的类定义不像 C++。C++在“}”后还必须有;号，C#则不必。

接着，在该类中添加各个对象成员，成员名为在 aspx 文件中定义的各 HTML 元素的 id 属性，成员类型与 HTML 元素的类型相一致，例如<asp:Button id=“btnButton” runat=server/>声明为：

```
Button  btnButton;
```

并且在类中定义要在服务器端处理的事件响应函数。如何编写事件响应函数我们将留到下一节详细讲解。

最后在服务器端响应事件的服务器端控件的属性中指定事件响应函数，并加入适当的指令。例如在类中定义了 btnButton_OnClick 函数的话，在其 HTML 标记中添加属性 OnClick=“btnButton_OnClick”：

```
<asp:Button id="btnButton" OnClick="btnButton_OnClick" runat=server />
```

并在@Page 指令中增加 Inherits=ClassNameOfCodeBehindClass 和 Src=<filename>属性。

通过这样的步骤，就可以将程序代码和页面内容分离。我们可以看到，如果程序员和网络设计师事先约定好所需的程序代码的交互元素的类型、id 和事件响应函数，他们的工作就可以分开并行的进行，并且任何一方的修改并不会影响另一方的工作。这和 ASP 时代的面条型代码有很大的区别。ASP 的代码是混合的，任何页面内容的修改（譬如调换一个 Button 和 TextBox 的位置）都会引起代码的大幅变动。这样子使用 WYSIWYG 编辑工具来编辑 ASP 网页几乎是不可能不把代码弄得一塌糊涂的。但使用 ASP.NET，这种编辑工具的开发几乎与普通的 HTML 编辑器没什么不同。例如 Visual Studio.NET 中就集成了 ASP.NET 的可视化编辑工具。

尽管我们给出了这样一个编写包含 Code-Behind 代码的页面的步骤，从 ASP.NET 的面向对象特性角度看来，页面及其中的组件的派生顺序却是和这个步骤中这些组件的出现顺序恰好相反的。在这个对象模型中，我们实际上是首先定义（declare）了一个由 Page 基类派生出来的 Codebehind 的 Page 子类，并为它添加特有的页面组件成员和其他成员（例如

方法成员）的声明。我们在这个子类中没有定义这些组件的初始属性。然后，我们创建 aspx 文件，通过 Inherits 属性，该文件实质上定义了 Codebehind 子类的子类（也就是说，最终的页面对象和标准的 Page 基类是“祖父-祖孙”的关系）。aspx 文件定义的子类除了具有从 Codebehind 子类共有继承（public derivation）而来的所有组件成员和方法成员外，还通过页面上的标记属性的方式，定义了这些组件的初始属性（例如按钮 Button 控件的 Text 属性），以及组件的行为（例如将某个类方法成员作为组件事件的响应函数）。这种标记和类组件成员的联系是使用标记的 id 属性创建的。

为了验证这样的继承关系，我们可以看看下面的代码：

CodeBehind.cs 文件：

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

public class MyForm : Page
{
    public Button btn;
    public void E(object Sender, EventArgs e)
    {
        Response.Write("Hello");
    }
}
```

MyForm.aspx 文件：

```
<%@ Page language="c#" Src="CodeBehind.cs" Inherits="MyForm" %>
<html>
<form runat="server">
<asp:Button id="btn"
    OnClick="E"
    Text="Click"
    runat="server"
/>
</form>
</html>
```

如果 MyForm 类的 E 方法的访问权限是如上的 public 或者 protected，这上面的代码可以工作。如果改为 private，编译器会提示：

```
Compiler Error Message: CS0122: 'MyForm.E(object, System.EventArgs)'
is inaccessible due to its protection level
```

这是因为，在面向对象语言中，类中的 public 方法可以被派生类和外界访问，protected 方法可以被派生类访问而不可被外界访问，而 private 方法既不能被派生类访问，也不能被外界访问。因此，作为 MyForm 类的派生类 MyForm.aspx 对应的类（这个类代码由 ASP.NET 服务器自动生成并由编译器编译）可以访问 MyForm 类中的 public 或 protected 的 E 方法，将其作为按钮的事件处理函数。如果 E 方法为 private，则自然会由于无法访问父类的私有成员出错。

我们还有一种方法将控件的事件和响应它的函数分离开来和连接起来，这种方法将更彻底，也更为强大。我们在下一节讨论事件模型时介绍。

4.5　ASP.NET Web Forms 的事件模型

4.5.1　嵌入在 aspx 文件中的事件响应代码

在上一节我们已经讨论过 Code-behind 方式的事件响应代码的编写模式。在这里我们先来看一个嵌入在 aspx 文件中的事件响应代码的写法。

程序清单 4-4：example-4-4.aspx

```
<script runat=server language="C#">
    public void btnSummit_Click(Object Sender, EventArgs e)
    {
        // do something here
    }
</script>

<title>
Example for event handling
</title>

<html>
<body>
<form>
<asp:Button runat=server id="btnSummit"
    OnClick="btnSummit_Click"
    Text="Summit" >
</asp:Buttong>
</form>
</body>
</html>
```

这里使用代码声明段来将事件响应代码嵌入到 aspx 文件中。注意，System、System.UI.Web、System.UI.Web.WebControl 这三个名字空间对 aspx 文件已经是被缺省地导入了，我们不必像示例代码 example-4-3-code.cs 中一样显式地导入这三个名字空间（请参阅本章 4.2.3 中对@Import 指令的讨论）。

由此可见，响应事件的代码可以采用任何一种代码组织方式：传统的放在<script></script>标记间，或者使用新的 Code-Behind 方式，我们已经在 4.4.3 节中演示过其中一种 Code-Behind 方式，并介绍了使用这种方式的代码编写步骤，请读者回顾这种方式，我们会在以后介绍另一种 Code-Behind 的事件响应代码组织方式。

4.5.2　Web Forms 的事件模型

ASP.NET Web Forms 的事件概念和传统的基于客户端程序的事件概念有很大的区别。造成这些区别的主要原因是它们的事件触发位置和处理位置不同。

基于客户端程序的事件在本地触发，并在本地处理。而 ASP.NET 的事件触发和事件处理地点是分离的，大多数的事件在客户端触发，然而却在服务器端处理。

在客户端触发事件要求客户端提供一个捕获事件信息并将该事件信息传递到服务器的机制。使用 HTTP 协议的 Post 方法来实现这一步是一种必然，这是因为 ASP.NET Web Forms 运行在 Web 上且必须遵循最基本的 Web 运行机制。

服务器接收到该信息后，ASP.NET 会解析该信息，找出事件的类型，查找并调用相应的事件处理函数。

从捕获信息、传输、解析到控制权转入事件处理函数的整个过程，都是 ASP.NET 服务器负责的。这个过程不需也不可能由 ASP.NET 程序员干预。ASP.NET 程序员几乎可以像从前的 ASP 编程处理客户端事件响应那样来编写服务器端事件响应代码。

但是，由于服务器端事件响应的过程是一个服务器和客户端的交互过程，并且由于网络的带宽和稳定性等特殊环境的限制，ASP.NET 程序员在处理客户端事件时应该仔细考虑性能的问题。如果过多不必要的事件被设计为在服务器端响应，就会造成事件信息频繁地在客户端和服务器端传递，从而降低网络程序的性能。由于这个原因，服务器端控件提供给 ASP.NET 程序员在服务器端响应的事件十分有限。例如会在客户端被频繁触发的 OnMouseOver 事件，便不被服务器端控件支持。程序员应该仔细考虑哪些事件适合在服务器端处理、哪些适合在客户端处理。

ASP.NET Web Forms 的事件处理函数的参数总是有两个：Object 类型和 EventArgs 类型（或者其子类）。Object 类型的变量指示了事件的触发源，EventArgs 类型指定了对应该事件的参数。对于某些服务器端控件，EventArgs 也可能被更具体的类型替代，从而可以表示更具体的控件参数信息，例如 AdRotator 服务器端控件的 AdCreatedEventArgs 类型可以表示 AdRotator 控件的 AdCreated 事件所需要的特殊参数。

需要提醒的一点是，客户端触发的事件并不一定在被触发后立即通过 Post 方法传递回服务器。缺省情况下，只有 Button（按钮）服务器端控件在被点击后立即向服务器传递事件消息。其他支持 Change 事件的服务器端控件的 Change 事件在被触发后均会在客户端缓冲区保留，直至下一次 Post 事件发生时一次性传递给服务器。服务器在接收多个 Change 事件消息后调用相应的事件处理函数，其调用顺序将是不确定的。但可以确定的是 Change 事件的响应总是在其它事件被响应之前。这种情况下，程序员不应该设计运行流程依赖于事件触发顺序先后的程序。

然而，程序员可以在设计时将支持 Change 事件的服务器端控件的 AutoPostBack 属性设为 true（缺省为 false），这样该控件的 Change 事件在客户端被触发后会立即向服务器端传递并被及时处理。这种方法虽然赋予程序员更大的控制权，但正如前面所说的，滥用它可能会导致网络应用程序的性能下降。

在客户端浏览器支持的前提下，验证控件可以使用客户端脚本，在客户端处理输入验证而不必回传到服务器端。

对于 Button 类和 TextBox 类控件，可以同时拥有客户端和服务器端事件处理代码，从而在客户端处理某些事件而在服务器端处理另外一些事件。例如在客户端处理 OnMouseOver 事件，在服务器端处理 Click 事件。当然，它们不能响应到同一个事件，当设置为响应同一事件时，服务器端的事件处理会覆盖客户端的事件处理，也就是说事件被触发时 ASP.NET 只会执行服务器端事件处理代码。

另外，ASP.NET 应用程序还拥有更高层次的事件，包括 Application 对象的 ApplicationStart 和 ApplicationEnd 事件、Session 对象的 SessionStart 和 SessionEnd 事件。ApplicationStart 在 Web 应用程序第一次启动时触发，程序员可以响应这个事件来初始化应

用程序生命周期内会使用到的各种资源，然后在 ApplicationEnd 事件中释放这些资源。虽然.NET 体系中包括了先进的垃圾回收机制，但没有手动回收的资源仍然会在内存中占用一定时间和空间，直到被垃圾收集器发现。在使用资源后立即释放掉仍是专业程序员的美德，垃圾回收机制只是一个安全的辅助性的机制而已，程序员不应该过多地依赖于收集器，否则仍会降低系统性能。Session 对象和 Application 对象很相像，SessionStart 事件在页面被第一次请求时被触发，SesstionEnd 事件则在应用程序显式地关闭该 Session 对象或 Session 对象超时的时候被触发。关于 Application 对象和 Session 对象的使用，我们在下一节的状态管理中说明。

4.5.3　ASP.NET 事件模型的背后

上一节的最后我们提到，在响应控件事件时，除了在服务器端控件的标记中加上属性来指定处理事件的函数外可以达到目的外，还有另一种更为彻底和强大的方法。.NET Framework 是一个完全面向对象的框架，ASP.NET 作为框架的一部分，使用其一般运行时（CLR）和类库。ASP.NET 支持的语言面向对象和支持 CLS 的特性，因此程序员可以使用 C#和 VB.NET 支持的事件机制来处理 ASP.NET 应用程序的事件响应。这种处理方法在编写自己的服务器端控件时尤为有用，在处理 Web Forms 的事件响应时也可以用到，并且从本质上来讲，ASP.NET 在处理所有事件时都是使用这种方法的。

在.NET 中，这个机制称为“代表（Delegate）机制”。首先，声明要处理的事件的类型，这称为一个“代表（delegate）”。Delegate 可以看作是事件响应函数的原型。它声明了处理某一类事件的函数的入口和返回值，这类似于一个 C++中的函数指针原型，它是类型安全的。我们在前面看到的事件处理函数实际上都属于.NET Class Library 中的 System 名字空间中的 EventHandler 代理类型。它的 C#形式的声明如下：

```
public delegate void EventHandler(
   object sender,
   EventArgs e
);
```

这解释了为什么我们前面的事件处理函数的入口总是有两个 Object 类型和 EventArgs 类型的参数(当然有面向对象的特性我们知道第二个参数也可以是 EventArgs 类的派生类)，并且没有返回值。当然在用户控件（User Controls）或者其他类中，程序员可以使用 EventHandler，也可以定义自己的 delegate，比如不带参数的 delegate，来满足自己的需要。

在 C#的类中，必须声明该类的对象可能触发的事件，并同时声明处理该事件的函数的代表类型。这是通过 event 关键字和相应的代表类型来完成的。例如：

```
using System;
using System.Web.UI.Controls;
class MyButton : Controls
{
    // …
    public event EventHandler Click;
    // …
}
```

ASP.NET 支持的服务器端控件都已具有自己的一些可以在服务器端处理的事件。例如 Button 类控件，它可以根据是否指定了 CommandName 属性分为 Summit（提交）形按钮或

者 Command（命令）形按钮。Summit 形按钮具有 Click 事件，Command 形按钮具有 Command 事件。

Click 事件的声明是：

```
public event EventsHandler Click;
```

这意味着处理 Click 事件的函数必须是 EventsHandler 类型的，接受一个 Object 类型参数和一个 EventArgs 类型参数并没有返回值。

Command 事件的声明是：

```
public event CommandEventsHandler Command;
```

而代表 CommandEventsHandler 的声明是：

```
public delegate void CommandEventHandler(
   object sender,
   CommandEventArgs e
);
```

这意味着处理 Command 事件的函数和处理 Click 事件的函数有所不同，第二个参数变成了 CommandEventArgs 类型。关于 CommandEventArgs 类，请读者自行查阅 SDK 文档。

事件处理函数是否符合某个 delegate，编译器会根据其声明自行判断，不必在声明时附带任何额外的说明。

在处理事件时，.NET 框架使用了一种“订阅”机制来将特定的符合某种 delegate 类型的处理函数“挂接”到特定对象的某个事件上，一个事件允许挂接多个处理函数，并且.NET 在调用这些处理函数来响应事件时其调用顺序是不确定的。一个将 Button1_Click()函数（假定它是 delegate 类形的）订阅到 Button1 对象的 Click 事件（这是一个 Summit 形按钮）的例子是：

```
Button1.Click += new EventHandler(Button1_Click);
```

取消订阅的代码可以这样写：

```
Button1.Click -= new EventHandler(Button1_Click);
```

所以，将示例代码 4-3 修改成这种“订阅”方式，可以将 example-4-3.aspx 中第 07 行的 onclick 属性去掉，并在 example-4-3-code.cs 中添加 Page_Load 事件响应函数如下：

```
public void Page_Load(object Source, EventArgs e)
{
    btnClick.Click += new EventHandler(btnClick_Click);
}
```

需要注意的是 Page_Init、Page_Load、Page_Unload 这类特殊事件的处理函数是由 ASP.NET 自行订阅到相应事件的，程序员所要做的只是在需要时重定义（override）这些函数。

当然还有 Src 属性等必要但在这里不重要的修改。修改后的完整代码如下：

程序清单 4-5-1：example-4-5.aspx

```
00  <%@ Page Inherits=EventEssentialsExample
     Src="example-4-5-code.cs"%>
01
02  <HTML>
03     <H3>Example for Event Essentials</H3>
04     <FORM RUNAT=SERVER>
05        <asp:Button id=btnClick runat="server"
06         Text="Click Me"
```

```
07          />
08              
09          <asp:Label id=lblState runat=server />
10      </FORM>
11  </HTML>
```

程序清单 4-5-2：example-4-5-code.cs

```
00  using System;
01  using System.Web.UI;
02  using System.Web.UI.WebControls;
03
04  public class EventEssentialsExample : Page
05  {
06      public Label lblState;
07      public Button btnClick;
08      public void Page_Load(Object Source, EventArgs e)
09      {
10          btnClick.Click += new EventArgs(btnClick_Click);
11      }
12      public void btnClick_Click(Object Source, EventArgs e)
13      {
14          lblState.Text="Button is clicked";
15      }
16  }
```

这个代码实现的功能与示例 4-3-1 和 4-3-2 是一样的，但在这里事件响应代码和事件的联系是完全用程序代码实现的。使用这种方法程序员可以在程序逻辑的运行时刻，动态地确定响应事件的代码。这将为 ASP.NET 应用程序带来十分灵活的类似于函数指针和消息映射机制的功能，但却是类型和内存安全的。并且我们可以将多个事件处理函数挂接到同一个事件上。

当然，从继承的角度讲，这里的最终页面的事件响应特性是从其父类 EventEssentialsExample 类继承而来的。

注意，“订阅”的过程是必须在事件发生前就完成的。譬如在我们的这个例子中，我们在 Page_Load 时确定控件的事件响应函数。当然我们也可以在响应某个控件的事件时为其他控件订阅事件处理函数。

例如我们可以创建这样的代码，在用户选择与数据库建立更安全的连接时将某个数据绑定控件或者提交按钮的事件响应函数替换成可以完成这个功能的函数，否则替换成建立普通连接的函数。或者可以根据用户类型来选择“显示天气预报”按钮，单击处理函数，从而为不同用户显示不同样式、不同内容的天气预报。只要你足够聪明，你就会想到足够多的应用这种动态订阅技术的场合。

另外我们也可以将多个事件处理函数订阅为同一个事件的处理函数，这种过程同样既可以是在编程时确定（从而使代码更为结构化、更易于理解和维护、扩充），也可以是动态的（根据用户的行为和选择来确定哪些函数功能会在事件触发时完成）。再提醒一次，事件触发时各个事件处理函数被 ASP.NET 调用的顺序将是不确定的，这是由代表（delegate）机制决定的。

我们来看一个应用该技术的例子。在这个例子中，我们要做的是收集用户信息，用户必须提供所在的国家和省份，用户还可以选择是否提供个人收入信息和选择是否接受广告。

可以选择的省份会根据国家的改变而相应地改变。另外如果用户愿意提供收入信息，页面会允许用户从多个收入区间中选择一个，并且收入区间和货币单位会根据国家的改变而改变；如果用户愿意接受广告，页面会允许用户在多个广告类型中选择一个或多个，并且广告类型也会根据国家的改变而改变。

仔细分析这个用户体验。我们发现选择是否提供收入信息和选择是否接受广告邮件可以分别用两个选择框（CheckBox）来接受用户输入。另外国家和省份可以分别使用两个下拉框（DropDownList）来获取输入。单个收入区间可以使用单选按钮（RadioButton）来表示，所有收入区间构成一个单选按钮组（RadioButtonList）。单个广告类型可以用选择框表示，所有广告类型构成一个选择框组（CheckBoxList）。国家、省份、收入区间和广告类型可以从数据库的数据源获得，但这里我们为了简化问题，没有使用到数据库，而是在代码中硬编码地添加内容。我们也只演示了两个国家而已。至于用到的各种服务器端控件，我们会在第五章介绍到。

进一步考虑逻辑实现层，我们会发现我们要处理的主要是国家改变的事件，在这个事件触发时，必须改变省份下拉框控件的内容，并且根据用户是否愿意提供收入信息和接受广告邮件而改变收入区间和广告类型。每一种改变我们都可以使用一个函数来实现。我们可以实现一个事件响应函数来响应国家改变的事件，并在这个事件响应函数中按需要调用三个改变控件内容的函数。但我们不这么做，因为我们现在有了更加面向对象的方法——将这些函数订阅为国家控件改变事件的处理函数！因为省份下拉框的内容必然会随着国家的改变而改变，我们在 Page_Load 时将改变省份内容的函数订阅为国家控件改变事件的处理函数。另外，在用户选择愿意提供收入信息时将改变收入区间的函数订阅为国家改变事件的处理函数，在选择不愿意时取消订阅。同样在是否接受广告邮件被更改时为国家改变的事件订阅或取消订阅改变广告类型的函数。这样当国家信息改变时，最多会有三个函数、最少会有一个函数被 ASP.NET 调用，而且其调用顺序是不确定的。

为什么使用这样的方法呢？

先考虑第一种做法的没有第二种合理的地方。这种方法里我们的事件和其处理函数是一一对应的。或者我们可以视为他们实际上是同一回事。在一个控件的事件发生时我们必然会去做一件事情：改变另一个控件。改变另一个控件的控制权是在发生事件的控件那里的。这样一来事件和处理事件的代码实质上仍是在编译时就确定好了的，虽然在代码的组织形式上是分离的，但在逻辑上他们仍然没有实现分离。

在我们采用的做法里，我们在为事件处理函数订阅事件时类似于事件处理函数告诉触发事件的控件：“如果你的这个事件发生了，我想你通知我一声。”在取消订阅时相当于通知控件：“算了我不要这个事件的通知了，以后不要再烦我了。”事件发生时，控件（或者说是 ASP.NET 系统）就会通知向它订阅该事件的函数：“你要的事件已经发生了，该做什么事你自己看着办吧。”事件发生后应该如何响应该事件，是由这些订阅了这个事件的函数一起决定的（当然不是并行的执行），因此其控制权分散到了各个函数手里。当这些函数是其他控件的成员函数时，便分散到了这些控件的手里。引发事件的控件（在这里是国家下拉框）并不需要事先（也就是说在编译时）知道哪些控件希望自己的内容会随着国家的改变而改变，也不需要知道它们会如何改变自己的内容（例如通过数据库还是其他方式）。引

发事件的控件只需要提供一个方法，允许其他控件订阅自己的事件，然后在事件发生时一个一个的通知那些告诉过它想要得到这个通知的控件就行了。这样我们不仅在代码形式上分离了事件及其处理代码，也在逻辑上分离了他们。这样构造的系统将具有更松散的耦合性，可以更容易的扩充。

现在我们来看完整的代码，它是可以运行的：

程序清单 4-6-1：example-4-6.aspx

```
<%@ Page Inherits="DynamicEventHandler" Src="example-4-6-code.cs" %>
<HTML>
   <H3>EventHandler Sample</H3>
   <FORM RUNAT=SERVER>
    <asp:Label runat="server" Text="Country" />
    <asp:DropDownList id="dlCountry" runat="server"
      AutoPostBack=true />
    </p>
    <asp:Label runat="server" Text="State" />
    <asp:DropDownList id="dlState" runat="server" />
    </p>
    <asp:CheckBox id="cbIncome" runat="server"
      AutoPostBack=true
      Text="I'm willing to expose my income level"
    />
    </p>
    <asp:RadioButtonList id="rblIncome" runat="server" />
    </p>
    <asp:CheckBox id="cbAd" runat="server"
      AutoPostBack=true
      Text="I'm willing to receive advertisements"
    />
    </p>
    <asp:CheckBoxList id="cblAd" runat="server" />
   </FORM>
</HTML>
```

程序清单 4-6-2：example-4-6-code.cs

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

public class DynamicEventHandler : Page
{
    // Members of Web Controls
    public  DropDownList        dlCountry;
    public  DropDownList        dlState;
    public  CheckBox            cbIncome;
    public  CheckBox            cbAd;
    public  RadioButtonList rblIncome;
    public  CheckBoxList        cblAd;

    // Private member to construct the datasources
    private ArrayList   alCNProvs;
    private ArrayList   alUSStates;
    private ArrayList   alCNIncomes;
    private ArrayList   alUSIncomes;
```

```
private ArrayList   alCNAds;
private ArrayList   alUSAds;

public DynamicEventHandler()
{
// Construct the datasources for the controls in the constructor
   of the class
    alCNProvs = new ArrayList();
    alUSStates = new ArrayList();
    alCNIncomes = new ArrayList();
    alUSIncomes = new ArrayList();
    alCNAds = new ArrayList();
    alUSAds = new ArrayList();

    alCNProvs.Add("Guangdong");
    alCNProvs.Add("Zhejiang");
    alCNProvs.Add("Beijing");

    alUSStates.Add("Texas");
    alUSStates.Add("Florida");
    alUSStates.Add("Woshingtong");
    alUSStates.Add("Virginia");

    alCNIncomes.Add("<1000RMB");
    alCNIncomes.Add("1000-2000RMB");
    alCNIncomes.Add("2000-4000RMB");
    alCNIncomes.Add("4000-6000RMB");
    alCNIncomes.Add(">6000RMB");

    alUSIncomes.Add("<500$");
    alUSIncomes.Add("500-1000$");
    alUSIncomes.Add("1000-3000$");
    alUSIncomes.Add(">3000$");

    alCNAds.Add("Commerce");
    alCNAds.Add("Entertainments");

    alUSAds.Add("Commerce");
    alUSAds.Add("Books");
    alUSAds.Add("Sports");
}   // end of constructor DynamicEventHandler

public void ResetStates(object Sender, EventArgs e)
{
// Reset the content of the DropDownList of States/Provinces,
// according to the content of the Country Box
    switch (dlCountry.SelectedItem.Text)
    {
        case "US":
            dlState.DataSource = alUSStates;
            dlState.DataBind();
            break;
        case "China":
            dlState.DataSource = alCNProvs;
            dlState.DataBind();
            break;
    }   // end of switch
}   // end of ResetStates
```

```
public void ResetIncomes(object Sender, EventArgs e)
{
// Reset the content of the RadioButtonList of Incomes,
// according to the content of the Country Box
    if (rblIncome.Visible = cbIncome.Checked)
    {
        switch (dlCountry.SelectedItem.Text)
        {
            case "China":
                rblIncome.DataSource = alCNIncomes;
                rblIncome.DataBind();
                break;
            case "US":
                rblIncome.DataSource = alUSIncomes;
                rblIncome.DataBind();
                break;
        }   // end of switch
    }   // end of if
}   // end of ResetIncomes

public void ResetAds(object Sender, EventArgs e)
{
// Reset the content of the CheckBoxList of Ads,
// according to the content of the Country Box
    if (cblAd.Visible = cbAd.Checked)
    {
        switch (dlCountry.SelectedItem.Text)
        {
            case "China":
                cblAd.DataSource = alCNAds;
                cblAd.DataBind();
                break;
            case "US":
                cblAd.DataSource = alUSAds;
                cblAd.DataBind();
                break;
        }   // end of switch
    }   // end of if
}   // end of ResetAds

public void Page_Load(object Sender, EventArgs e)
{
    if (!Page.IsPostBack)
    {
        // initialize the content of the Country Box
        dlCountry.Items.Add("US");
        dlCountry.Items.Add("China");
        // subscribe the necessary event handler
        dlCountry.SelectedIndexChanged += new
        EventHandler(ResetStates);
        cbIncome.CheckedChanged += new
        EventHandler(cbIncome_Change);
        cbAd.CheckedChanged += new EventHandler(cbAd_Change);
    }

    // reset the States/Province according to the country
    ResetStates(this, e);

    // reset the optional event handler according to the country
```

```
            ResetIncomes (this, e);
            ResetAds(this, e);
        }   // end of Page_Load

        protected void cbIncome_Change(object Sender, EventArgs e)
        {
            // subscribe or unsubscribe the event handler of changing
            the content of Income group
            // for the country-changed event
            if (cbIncome.Checked)
            {
                dlCountry.SelectedIndexChanged += new
                EventHandler(ResetIncomes);
            }
            else
            {
                dlCountry.SelectedIndexChanged -= new
                EventHandler(ResetIncomes);
            }
            // Refresh the Income group if it is checked
            ResetIncomes(Sender, e);
        }   // end of cbIncome_Change

        protected void cbAd_Change(object Sender, EventArgs e)
        {
            // subscribe or unsubscribe the event handler of changing
               the content of Ads group
            // for the country-changed event
            if (cbAd.Checked)
            {
                dlCountry.SelectedIndexChanged += new
                EventHandler(ResetAds);
            }
            else
            {
                dlCountry.SelectedIndexChanged -= new
                EventHandler(ResetAds);
            }
            // Refresh the Ad group if it is checked
            ResetAds(Sender, e);
        }   // end of cbAd_Change
    }   // end of DynamicEventHandler class
```

这段代码完成了我们希望得到的用户体验的核心部分。这里没有包括任何改变显示样式的代码，因为这和我们要阐述的程序逻辑无关（另一个原因是写这段代码的作者没有任何美工基础）。请留意加粗的代码。

在改变几个控件包含的项目时，我们使用数据绑定来向这些控件填充值，因此我们在 DynamicEventHandler 类中声明了几个私有的 ArrayList 成员来作为数据绑定的数据源。

DynamicEventHandler 类中的第一个函数是该类的构造函数，我们在类对象被构造时（也就是页面每次被请求时，不论是否是第一次该页面被某个用户请求）初始化几个类的私有成员。这是因为 Page 类对象的生命周期只是每次页面被请求时到相应的 HTML 页面被生成为止，我们的 DynamicEventHandler 类也是如此，因此每次的页面对象的状态缺省都是没有保存的（在下面一节会提到），这包括作为私有对象的几个 ArrayList 成员。因此我们将这个初始化工作放到类的构造器中完成。

在页面每次被请求时，我们构造了几个私有对象后进入 Page_Load()函数。我们首先使用从 Page 类继承而来的 IsPostBack 属性判断该页面是否第一次被该对话请求，如果是第一次被请求，则初始化“国家”下拉框（我们不必在类的构造函数中初始化它，因为服务器端控件的状态会由 ASP.NET 自动保留，下一节我们会讲到视图状态），并挂接几个必要的事件处理函数。最后我们做每一次页面被请求时都要做的事情：重置省份下拉框内容和根据需要与否重置另外两个控件内容。虽然这三个控件的内容也可以被 ASP.NET 自动保存，但这是不必要的，因为我们要根据国家内容来改变它们的内容。

在 Page_Load()函数中，我们将 ResetStates 订阅为国家改变事件的一个响应函数，并订阅两个 CheckBox 控件（cbIncome 和 cbAd）的 CheckedChanged 事件。然后在这两个事件发生时，在响应代码中根据是否被改变成 Checked 决定是否将 ResetIncomes()和 ResetAds()函数订阅或取消订阅为国家改变事件的响应函数。

这里在订阅这两个函数时，并不管是否有其他函数已经是国家改变事件的响应函数，因为 ResetIncomes()和 ResetAds()函数并不依赖或者改变其他控件的值。这一点非常重要，因为各个事件响应函数的被调用顺序是不确定的，程序员必须避免写出依赖于其他控件的值的事件响应代码，除非可以确定没有其他事件响应代码会改变这些值。即便可以确定所依赖的值不会被改变，我们也不推荐这么写，因为这样的系统将极不利于扩展。

设想我们的 ResetIncomes()要使用到另一个 TextBox 控件的文本值，而恰好以后的某天另一个程序员添加了一个会改变这个 TextBox 控件的内容的函数，并将它订阅为国家改变事件的处理函数，这将会发生什么事情呢？TextBox 的值可能在改变后被 ResetIncomes()读取，也可能在被读取后才被改变，作为程序员的你能容忍这样的不确定性吗？并且这样的错误在系统变得很复杂后，将是很难发现、修正成本也是很高的。

另外要提醒使用这种技术的读者，不要忘了在适当的时候取消订阅。我们往往着迷于这种动态挂接的技术，但也要足够严谨地解除挂接。只关注使用新技术但却不能正确工作的代码是一文不值的。

4.6　ASP.NET Web Forms 的状态管理

客户机请求页面时，在服务器端都是一次全新的生成页面的过程，也就是说，这个过程是没有任何状态被保留的。为了满足需要，ASP.NET Web Forms 提供了几种状态管理的方法。

4.6.1　Application 对象

Application 对象是从以前的 ASP 技术继承过来的状态对象，但 ASP.NET 中的这个对象有了一些改变。

Application 对象中维护的各个变量，其实是一个特定的 Web 应用程序的全局变量。因此 ASP.NET 程序员必须考虑将各种各样的值作为全局变量储存到 Application 对象中可能产生的种种影响。

首先，作为应用程序的状态存储起来的变量，占用的内存资源在被移除或替换掉之前，

是不会被释放的。程序员在往 Application 对象中“堆放”变量时应该慎重，避免将极少使用的大对象（例如一个很大的数据集）维持在 Application 中。

第二，在多线程环境下，同一个 Application 对象的不同线程可能会同时访问一个 Application 对象中的变量。这时程序员必须注意了。当要访问的全局变量不是线程安全的时候，必须编写显式的同步方法，来避免死锁、诸塞或访问拒绝；当要访问的全局变量是线程安全的时候，程序员必须确保该对象已拥有内置的同步支持。另外，.NET Framework 中内置的集合对象并没有拥有内置的同步支持。因此程序员在将它们放入 Application 对象状态中时，必须显式地使用 HttpApplicationState 类中的 Lock 和 Unlock 方法，来避免上面提到的同步问题。

第三，ASP.NET 程序员必须注意到在一个多线程的服务器环境下，保护全局资源的“锁”本身也是全局的，运行在多线程的基础上的代码在访问全局资源时最终会在竞争这些“锁”资源。这时操作系统就会阻塞工作线程，让它等待一直到锁对它是可用的为止。在一个负载严重的服务器环境中，这种阻塞现象会引起严重的问题。在一个多处理器系统里，一个处理器的线程从理论上讲可能由于在等待同一个“锁”而被阻塞，从而可能导致处理器的利用率降低。

第四，注意存储在 Application 对象中的状态是和应用程序的生命周期有关的。.NET 的应用程序域或者运行.NET 应用程序的进程可能由于各种原因而在运行时的任意时刻被撤销或者销毁。当所在的应用程序域或者进程被撤销时，存储在 Application 对象中的状态也会同时被撤销。因此当要求在失败恢复后重新获取应用程序的状态时，程序员必须将该状态存储在数据库之类的不易损坏的存储机制中。

最后，应用程序状态在 Web Farm 或者 Web Garden 中并不是共享的。在一个 Web Farm 环境中，一个应用程序在多个服务器上处理；在一个 Web Garden 中一个应用程序由同一服务器上的多个进程运行。在这种情况下存储在 Application 状态中的变量仅对于应用程序在其中运行的特定进程来讲是全局的。每一个应用程序进程可能拥有不同的状态，因此程序员不能依赖应用程序状态储存特有的值或者更新全局计数器。

尽管有这么多注意事项，设计良好的应用程序级别的变量仍然可以为 Web 应用程序带来强大的威力。

可以通过两个状态集合来在程序里添加和访问状态：Contents 和 StaticObjects。

Contents 集合暴露所有那些已通过程序代码添加到应用程序状态集合中的那些变量。例如：

```
Application["Counter"] = 0;
Application["Counter"] = Application["Counter"] + 1;
```

为了同以前版本的 ASP 保持兼容，也可以通过访问 Application 对象的 Contents 属性来访问中间的变量。因此前面的代码也可以这样写：

```
Application.Contents["Counter"] = 0;
Application.Contents["Counter"] = Application.Contents["Counter"] + 1;
```

StaticObjects 集合暴露所有那些通过 Global.asax 文件中的<object runat=server>标记添加到应用程序状态集合中、并且作用范围是整个应用程序的那些变量。

4.6.2　Session 对象

ASP.NET 提供了 Session 对象，从而允许程序员识别、储存和处理同一个浏览器对服务器上某个特定的网络应用程序的若干次请求的上下文信息。Session 对象对应于浏览器与服务器的同一次对话，在浏览器第一次请求网络应用程序的某个页面时，服务器会触发 Session_OnStart 事件；在对话超时或被关闭时，会触发 Session_OnEnd 事件。程序员可以在代码中响应这两个事件来处理与同一次对话相关的任务，如开辟和释放该次对话要使用的资源等。程序员也可以在 Session 对象中存放本次会话有关的数据。当会话超时或被关闭时，ASP.NET 会自动释放存放于其中的数据和对象。

ASP.NET 中的 Session 对象和 ASP 保持了很大程度的兼容性，ASP 程序员可以很容易的使用这个对象来保存会话状态。

在 ASP.NET 的程序代码中使用 Session 对象时，必须确保页面的@Page 指令中 EnableSessionState 属性是 true 或者 ReadOnly（缺省为 true，请回顾 4.2.3 节中对@Page 指令的介绍），并且在 web.config 文件中正确地设置了 sessionState 的属性。

使用 Session 对象的语法与在 ASP 中一样，例如使用 C#向 Session 对象添加一个 nCounter 变量后使之递增 1 的代码是：

```
Session["nCounter"] = 0;
Session["nCounter"] = (int) Session["nCounter"] + 1;
```

使用 Session 可以存储任意的数据和 COM 对象。

与 Application 对象不同的是，ASP.NET 的 Session 对象可以在 IIS 服务器或工作进程重新启动时恢复启动前的状态而不会丢失其中的数据。这是因为存储在 Session 中的所有信息缺省的都是存储在一个作为 Windows 服务（Service）运行的状态服务器进程中的。状态可以被序列化并以二进制形式保存在内存中。程序员也可以选择使用 Microsoft SQL Server 数据库来储存数据。状态服务器服务和状态信息可以和 Web 应用程序一起存在于同一台服务器上，也可以保存到外部的状态服务器。为了指定如何储存信息，程序员可以在 web.config 文件中编写适当的配置。

注意：web.config 文件在.NET Framework 的 Beta 1 阶段时叫 config.web，但在 Beta 2 中改为 web.config。关于 web.config 的配置，我们在以后会详细介绍。在这里我们先说明我们要做的事情。读者可以参考后面的信息，也可以将这个问题留到以后在第十章我们讲到应用程序配置时再回顾这一节。

ASP.NET 应用程序的状态保存方式是由 web.config 文件中的<system.web>标记下的<sessionstate>标记的 mode 属性来决定的。该属性有四种可能的值：Off、Inproc、StateServer 和 SQLServer。

设为 Off 会禁用 session。

Inproc 是缺省的设定，这种模式和以前的 ASP 会话状态的方法是类似的，会话的状态被保存在 ASP.NET 进程中，它的优点是显而易见的：性能。进程内的数据访问自然会比跨进程的数据访问快上许多倍。然而，使用这种方法应用程序的状态将依赖于 ASP.NET 进程，当 IIS 进程崩溃或正常重启时，保存在进程中的状态将丢失。

为了克服 Inproc 模式的缺点，ASP.NET 提供了两种进程外保存会话状态的方法。但要注意这两种方法只在 ASP.NET 的 Professional 版本和 Enterprise 版本被支持。至少在 Beta 2 版本时是这样。

ASP.NET 首先提供了一个 Windows 服务（service）:ASPState。管理员启动了这个服务后，ASP.NET 应用程序就可以将 mode 属性设置为“StateServer”，来使用这个 Windows 服务提供的状态管理方法。

启动 ASPState 服务可以在管理控制台中完成，也可以通过命令行：

```
net start aspstate
```

除了在 web.config 文件中设置 mode 属性为“StateServer”外，还必须设置运行 State Server 的服务器 IP 地址或域名及 State Server 使用的端口。如果在 IIS 所在的服务器上运行 State Server，服务器地址就是“localhost”或者“127.0.0.1”，端口号就是 42424。这两个属性配置如下：

```
mode="StateServer"
connectionString="127.0.0.1:42424"
```

使用这种模式，会话状态的储存将不依赖于 IIS 进程的失败或重启。会话的状态将储存在 State Server 进程的内存空间中。

另一种会话状态存储模式是 SqlServer 模式。这种模式与 State Server 模式接近，但状态是保存在 SQL Server 数据库中的。使用这种模式前，必须有至少一台 SQL Server 服务器，并在该服务器中建立需要的表和存储过程。.NET SDK 提供了两个 SQL 脚本来简化这个工作：InstallSqlState.sql 和 UninstallSqlState。这两个文件存放在以下路径中：

```
<%SYSTEMDRIVE%>\Winnt\Microsoft.NET\Framework\<%version%>\
```

要配置 SQL 服务器，可以在命令行中运行 SQL Server 提供的命令行工具 osql.exe

```
osql -S [server name] -U [user] -P[password] <InstallSqlState.sql
```

例如：

```
osql -S (local)\NETSDK -U sa -P "" -i InstallSqlState.sql
```

在这里用户名必须是 SQL 服务器上的 sa 帐号，或者具有同等权限的其他帐号。有兴趣的读者可以打开这个脚本文件来了解 ASP.NET 是如何和 SQL Server 配合实现状态管理的。卸载这些表和存储过程，可以使用 UninstallSqlState.sql 脚本，使用方法与上面类似。

做好必要的数据库准备工作后，将 config.web 文件中的 sessiontstate 元素的 mode 属性改成“sqlserver”，并指定 SQL 连接字符串。具体如下：

```
mode="SQLServer"
sqlConnectionString="data source=127.0.0.1; userid=sa; password="
```

使用 SQLServer 模式除了可以使 ASP.NET 程序的状态存储不依赖于 IIS 服务器外，还可以利用 SQL Server 的集群，使状态存储不依赖于单个的 SQL Server，这样就可以为应用程序提供极大限度的可靠性。

另外 Session 对象也可以用在 Web Farms 和 Web Garden 环境中，管理员可以给重负荷的 Web 应用程序分配更多的处理器资源来获得更高的灵活性。

需要注意的是，进程外存储状态的两种方式仅会在 ASP.NET 发行时的企业版中可用。当前来讲，在.NET Framework SDK Beta 2 和 Visual Studio.NET Enterprise Edition Beta 2 中这两个特性仍是不可用的。

使用 Session 对象不必要求客户端浏览器支持 Cookies。程序员可以通过指定 sessionState 元素的 cookieless 属性为 true 或 false 来决定是否使用 cookies。缺省设置为 false，即使用 cookie，这意味着需要客户端浏览器的支持和允许使用 cookie。设置为 true 则无须客户端的支持，ASP.NET 会自动替换 Post 方法的 URL，通过在 URL 嵌入加密过的状态信息来避免使用 cookie，这种情况下如果用户更改了自动生成的 URL，则应用程序无法正常工作。两种模式各有优缺点。

4.6.3　视图状态（View State）

视图状态是 ASP.NET Web Forms 的一个状态管理机制，它指的是页面的状态和当前属性，以及页面上的控件的状态和它们的基本属性，可以在页面的服务器端和客户端多次来回传递过程中被自动保存。我们在讲述页面的处理过程时已经说过 View State 的特性。我们会在以后的对 Web Forms Server Controls（Web Forms 服务器端控件）的讨论中看到这个机制给 ASP.NET 程序员带来的极大的方便。

4.6.4　状态保存器（State Bag）

状态保存器是另一个在 ASP.NET Web Forms 中保存值的机制。页面和控件的状态已经可以通过 View State 机制自动的保存。程序员很自然的需要另一个机制来在页面的来回传递过程中保存他们的程序自己的变量值。State Bag 起到这个作用。

使用状态保存器仍然很接近于使用 Application 对象和 Sesstion 对象（因为它们都直接或间接地实现了 ICollection 接口和 IEnumerable 接口）。在 Web Forms 的程序代码中，使用 ViewState 对象来引用状态保存器。例如我们想在某个事件响应函数（当然也可以是其他任何一个 Page 类的成员函数）中实现一个类似于 C/C++语言中的函数静态变量，我们可以这么写：

程序清单 4-7-1：example-4-7.aspx

```
<%@ Page Language="C#" Inherits="PStaticVar"
Src="example-4-7-code.cs" %>
<html>
   <H3> Using State Bag to simulate static variables </H3>
   <form runat=server >
      <asp:Button id=Button1 runat="server"
         Text="Count"
         onclick="Button1_Click" />
      </p>
      <asp:Label id=Label1 runat=server />
   </form>
</html >
```

程序清单 4-7-2：example-4-7-code.cs

```
using System;
using System.Web.UI;
using System.Web.UI.WebControl;

public class PStaticVar : Page
{
    public Button Button1;
    public Label  Label1;
```

```
        protected void Button1_Click(object Source, EventArgs e)
        {
            int nCounter;
            if (ViewState["nCounter"]==null)
                nCounter = 0;
            else
                nCounter = ((int) ViewState["nCounter"]) + 1;
            Label1.Text="You clicked the button for " +
            nCounter.ToString() + " times";
            ViewState["nCounter"] = nCounter;
        }
    }
```

如果想查看枚举状态保存器中的所有状态，请参考下面的例子。它使用到了 C#的 foreach 语句。读者可以顺便预览一下我们即将涉及的 Web Forms 服务器端控件（Server Controls）和数据绑定技术。

提示：下面用到的数据绑定技术在本章开始的示范代码 4-2 中已经使用过了。

程序清单 4-8-1：example-4-8.aspx

```
<%@ Page Language="C#" Inherits="PEnumStates"
Src="example-4-8-code.cs" %>
<html>
   <H3> Using State Bag to simulate static variables </H3>
   <form runat=server >
      <asp:Button id=Button1 runat="server"
        Text="List"
        onclick="Button1_Click" />
      </p>
      <asp:ListBox id=ListBox1 runat=server visible=false/>
   </form>
</html >
```

程序清单 4-8-2：example-4-8-code.cs

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

public class PEnumStates : Page
{
   public Button    Button1;
   public ListBox   ListBox1;
   protected void Page_Load(object Sender, EventArgs e)
   {
    // Generating the states
    for (int i = 0; i < 4; i++)
        ViewState[i.ToString()] = "Item " + i.ToString();
   }
   protected void Button1_Click(object Sender, EventArgs e)
   {
    ArrayList arr = new ArrayList();

    foreach (string s in ViewState.Keys)
    {
        arr.Add(s + " " +ViewState[s]);
    }
```

```
            ListBox1.DataSource = arr;
            ListBox1.DataBind();
            ListBox1.Visible = true;
            }
        }
```

4.7　本章小结

本章是 ASP.NET 技术的介绍，内容比较多，而且比较复杂，包括了 ASP.NET 的核心技术之一：Web Foms(Web 表单)，以及 aspx 文件格式，而且本章最后还介绍了 ASP.NET 的几个常用对象，如 Application，Session 等，希望读者能仔细阅读并掌握。

第二篇

XML和ASP.NET数据交换

作为本书的第二篇，内容上较第一篇有了一定深入，主要结合一些程序实例介绍有关 XML 和 ASP.NET 的数据交换技术。

首先第 5 章介绍 ASP.NET 为 Web Forms 技术引入的 Server Controls（服务器端控件）的概念，系统地介绍 HTML Server Controls、ASP.NET Server Controls 的使用以及创建可重用的 User Controls 的技术。

第 6 章介绍 DOM 的基本知识和如何在.NET 应用程序开发过程中使用 DOM 支持。ASP.NET Framework 实现了对 DOM 规范的良好支持，并提供了一些扩展技术，使得开发人员对 XML 文档的处理更加方便。

第 7 章介绍 XML 的 SAX（Simple Application for XML）数据接口。读者会了解到 SAX 接口的产生和特点，与 DOM 接口相比的优势与不足。接着将详细地说明在.NET Framework 中使用 XmlReader 和 XmlWriter 两个类去实现一种可以体现 SAX 接口优势的数据读写方式。最后通过实例说明将 SAX 和 ASP.NET 结合起来实现收集网页中的数据信息并将得到的数据存储为 XML 文档的技术。

第 8 章介绍用 ADO.NET 技术来实现数据访问的通用解决方案。本章首先介绍有关 ADO.NET 的基础知识，然后介绍如何在 ASP.NET 应用程序中使用 ADO.NET 来进行数据访问。接着本章会对前几章已经使用过的数据绑定技术进行一个总结，并介绍 DataSet 对象和 XML 的协同工作问题。

第 5 章　使用 ASP.NET 控件

ASP.NET 为 Web Forms 技术引入了 Server Controls（服务器端控件）的概念。Server Controls 可以分为以下几类：

- HTML Server Controls
- Web Server Controls
- Validation Server Controls
- User Server Controls（Pagelets）
- Mobile Controls

其实我们已经在前面一章中接触到服务器端控件了。我们会在 5.1 节系统地学习 HTML Server Controls 和 ASP.NET Server Controls 的使用。在 5.2 节，我们将学习如何创建可重用的 User Controls。在 5.3 节我们将简要介绍其它两种控件的使用方法。

5.1　HTML Server Controls 和 ASP.NET Server Controls

5.1.1　HTML Server Controls

缺省情况下，ASP.NET 文件中的 HTML 标记被视为纯文本，它们对于程序员来说是不可访问的。但是 ASP.NET 允许在这些 HTML 标记中适当地添加属性，使得它们可以作为服务器端控件被程序代码访问并操纵。

为了将 HTML 元素变成服务器端控件，需要在标记中添加 runat=“server”属性。

为了使变成了服务器端控件的 HTML 元素可以被运行在服务器上的程序代码访问，还必须给该 HTML 元素赋上一个标示它的 id 属性。

做好这些工作后，还必须保证作为 HTML 服务器端控件的 HTML 标记被包括在一个<form></form>标记中，作为 HTML 服务器端控件的容器，这个<form>标记还必须具有 runat=“server”属性。当然，如果我们在程序代码中不会访问到这个<form>标记，可以不给它赋上 id 属性。

HTML Server Controls 分为两类：HTML 容器控件和 HTML 输入控件。

HTML 容器控件对应那些要求有配对的开闭标记的 HTML 元素，例如<form>、<button>、<table>标记。HTML 容器控件包括：HtmlTableCell、HtmlTable、HtmlTableRow、HtmlButton、HtmlForm、HtmlAnchor、HtmlGenericControl、HtmlSelect 和 HtmlTextArea 控件。

HTML 输入控件对应那些不要求关闭标记的 HTML 元素，这些 HTML 元素均包括 Type 属性来定义它们的输入类型。ASP.NET 中的 HTML 输入控件包括：HtmlInputText、HtmlInputButton、HtmlInputCheckBox、HtmlInputImage、HtmlInputHidden、HtmlInputFile

和 HtmlInputRadioButton 控件。

首先我们来看所有 HTML Server Controls 的共有的属性。

无论是容器控件还是输入控件，都有一个 Attributes 属性，该属性是一个集合，通过这个集合可以访问到并设置该控件被定义的所有属性名和属性值。另一个集合属性是 Style，可以通过它取得所有应用到该控件的层叠样式表属性。

每个 HTML Server Controls 控件还都具有以下属性：

Disabled：返回一个 bool 值，确定或者指定控件在生成 HTML 代码时是否包含 disabled 属性。

TagName：返回一个 string 类型的标记名称。

对于容器控件，共有的特性有两个：InnerHtml 和 InnerText。这两个属性都表示容器控件开闭标记之间的内容。不同的是 InnerHtml 属性不会自动地将其中的特殊字符（例如“<”“>”符号）转化为 HTML 引用（“<”的引用是&lt），或者将后者转化为前者。而 InnerText 属性则会进行这样的转换。

因此，下面的代码：

```
<script runat="server" language="C#">
    public void Page_Load(Object Sender, EventArgs e)
    {
        msg.InnerHtml = "<b>Hello</b>";
    }
</script>
<form>
<span id="msg" runat="server”></span>
</form>
```

会输出粗体的 Hello 字样。

而这样的代码：

```
<script runat="server" language="C#">
    public void Page_Load(Object Sender, EventArgs e)
    {
        msg.InnerText = "<b>Hello</b>";
    }
</script>
<form>
<span id="msg" runat="server"></span>
</form>
```

则输出Hello字样。

所有 HTML 输入控件均具有以下属性：

Name：获得或者设置输入控件特有的标示它的名称。

Value：获得或者设置输入控件的内容。

Type：获得该输入控件的类型。

这三个属性均为 string 类型。

各个 HTML 控件都有一些自己特有的属性，这些属性大多和它们对应的 HTML 标记的属性是一致的。请读者自行参阅 SDK 文档来获得这些属性的详细信息。

5.1.2　Web Server Controls

Web Server Controls（Web 服务器端控件）是一组从 WebControl 基类派生出来的控件，WebControl 类位于 System.Web.UI.WebControls 名字空间中。这组控件既包含传统的表单控件如 Label、TextBox、Button 等，也包含了更高抽象级别的控件如 Calendar、DataList 等。

Web Server Controls 具有更好的面向对象特性，所有控件的通用属性都在 WebControl 基类中实现，具有高度的一致性，从而简化编程人员的工作，减少错误。

Web Server Controls 可以自动地检测客户端浏览器的类型和功能，生成相应的 HTML 代码，从而最大程度的发挥浏览器的功能。

另外它们还具有数据绑定的特性，所有属性都可以进行数据绑定，某些控件甚至还可以向数据源提交数据。

使用 Web Server Controls 时，必须在类名前添加“asp:”前缀，“asp:”前缀用来映射到这些 Web Server Controls 所处的 System.Web.UI.WebControls 名字空间。并且必须带有 runat=“server”属性。例如：

```
<asp:Button id="btnSummit" runat="server" Text="Summit"></asp:Button>
```

表示一个 Button 控件，该控件的 id 为 btnSummit，程序代码可以使用这个 id 来引用该控件，控件的文本为 Summit。

当开闭标记之间没有内容时（即空元素），可以按照 XML 文法的规则，省略关闭标记（注意关闭标记仍是存在的）而简写成下面这种形式（空标记）：

```
<asp:Button id="btnSummit" runat="serve" Text="Summit" />
```

.NET Framework Beta 2 支持的 Web Server Controls 如表 5-1 所示。

表 5-1　Web Server Controls

Web 服务器端控件名称	控件简介
AdRotator	显示一个广告条
Button	创建一个下压按钮
Calendar	显示一个月历，允许用户选择月和日期
CheckBox	创建一个选择框，允许用户在选择与不选择之间切换
CheckBoxList	创建一个选择框组，可以根据数据源来动态的创建组内的选择框
DataGrid	以表的形式有选择地显示数据源中的数据，并允许用户选择、编辑、排序控件中的项目
DataList	使用模板来显示数据源中的数据。内容的显示形式可以通过模板来控制
DropDownList	显示一个下拉列表，用户可以在列表里的项目中选择其中的一个
HyperLink	显示一个超链接，允许用户跳转到该链接
Image	显示一个格式合适的图片
ImageButton	显示一个图片并允许用户单击该图片时触发 Click 事件，程序员可以捕获该事件并响应它
Label	用来显示静态文本，该文本是可通过程序控制的，并且不像 Literal 控件，Label 控件可以将样式属性应用到文本上

(续表)

Web 服务器端控件名称	控件简介
LinkButton	显示一个超链接形式的按钮
ListBox	显示一个列表框，用户可以选择列表框中的一个或多个项目
Literal	显示静态文本并可由程序控制该文本，但不像 Label 控件，不能将样式应用到文本内容上
Panel	作为其他控件的容器
PlaceHolder	在页面的控件层次结构中保留一个控件位置，从而可以通过程序在该位置上添加控件
RadioButton	显示一组单选按钮
RadioButtonList	显示一组单选按钮，并可以根据数据源来动态生成组中的按钮
Repeater	使用模板来显示数据源中的数据，但其内容的显示样式不能继承而来，必须定义控制内容显示的模板
Table	显示一个程序可操纵的表格
TableCell	该控件允许程序员声明一个表格控件中的单元格并通过程序操纵该单元格
TableRow	该控件允许程序员声明一个表格控件中的行并通过程序操纵该行
TextBox	显示一个文本框，该文本框可以是单行的，也可以是多行的
Xml	用来显示一个 XML 文档或者 XML 文档通过 XSL 转换后的结果

除了 Literal、PlaceHolder 和 Xml 控件外的其它所有 Web Server Controls 都具有一些共同的属性，如表 5-2 所示。

表 5-2　Web Server Controls 的一些共同属性

属性	属性描述
AccessKey	指定一个字母或数字键，用户可以通过 ALT 键和设定的这个键的组合来快速访问控件。这个特性只在 IE4.0 或更高版本被支持
Attributes	表示控件的没有被定义为公共属性而会在 HTML 代码中生成的那些属性的集合，这个属性只可在程序代码中使用
BackColor	控件的背景色
BorderColor	控件的边框颜色
BorderWidth	控件的边框宽度
BorderStyle	控件的边框样式。可能的取值有：NotSet、None、Dotted、Dashed、Solid、Double、Groove、Ridge、Inset、Outset
CssClass	指定应用到控件的层叠样式表（CSS）
Style	应用到控件的外层标记上的 CSS 样式属性集合
Enabled	指定控件是否工作，取值有 true 和 false 两种
Font	控件的字体
ForeColor	控件的前景色
Height	控件的高度

(续表)

属性	属性描述
TabIndex	指定使用 Tab 访问控件时该控件的序号
ToolTip	指定用户将鼠标移到控件上时显示的文本
Width	控件的固定宽度

除了这些共性之外，许多 Web Server Controls 还有自己的特性和使用方法。我们在这里举例介绍 AdRotator 控件的使用方法。读者可以利用 SDK 文档自行研究其他控件的详细信息。

AdRotator 控件是 ASP.NET 新加入的控件，使用它可以在页面上显示一个广告条。广告条的内容有一个外部的 XML 文件定义。AdRotator 控件的标记格式为：

```
<asp:AdRotator
    id="value"
    AdvertisementFile="AdvertisementFile"
    KeyWordFilter="KeyWord"
    Target="Target"
    OnAdCreated="OnAdCreatedMethod"
    runat="server"
/>
```

外部的 XML 文件路径由 AdvertisementFile 属性指定。这个 XML 文件中必须至少包含一个<Advertisements>和</Adevertisements>标记，这对标记之间的内容将会被 AdRotator 控件解析。但当文件中存在多对的<Advertisements>和</Advertisements>标记时，只有第一对标记会被解析，其他标记将会被忽略。在<Advertisements>和</Advertisements>标记中，使用一对或多对并列的<Ad></Ad>标记来表示每一个广告。<Ad>标记间可以包含表示该广告的各种信息的其它标记，这些标记既可以是自定义的，也可以是预定义的。预定义的标记有下列几种元素，见表 5-3。

表 5-3　预定义标记的元素

元素	元素的描述
<ImageUrl>	指定该广告的相关图片的 URL
<NavigateUrl>	指定用户单击该广告时跳转的 URL
<AlternateText>	指定当广告图片不可用时会显示的文本
<Keyword>	指定该广告的关键字，可以用来被 AdRotator 控件筛选
<Impressions>	一个指定该广告重要程度的值，这个值越大，这个广告被显示的频率越高。同个 XML 文件里的值的个数不能超过 2,047,999,999，否则控件会抛出一个运行时异常

AdRotator 控件在被创建时根据 Keyword 属性来从 AdvertisementFile 属性指定的 XML 文件中过滤出拥有该关键字的广告，如果没有符合该关键字的广告存在，AdRotator 将显示一个空白的图片，并引发一个跟踪时警告。如果有多个符合条件的广告存在，AdRotator 将根据各个广告的 Impression 值来选择一个广告，Impression 值越高的广告被选中的机会将越大。选择后 AdRotator 显示该广告的图片，并允许用户单击该广告。图片的大小将被调整为控件的大小。在用户单击广告时，AdRotator 将页面重定向到 NavigateUrl 指定的目标

URL 上，程序员可以将该 URL 置为自定义的 ASP.NET 程序，经过处理后再定向到提供广告信息的页面，从而实现对广告单击次数等的统计。

如果没有指定 AdvertisementFile，事件参数会是空的，这样，程序员可以判断并通过修改事件参数来指定广告的内容。

每个广告的用户自定义标记会在 AdCreated 事件触发时传送到 AdProperties 属性中（读者可能已经猜到这个属性是个集合）。程序员也可以在 AdCreated 事件响应代码中直接选择要显示的广告，这可以通过修改 AdProperties 属性而改变要生成的广告的 ImageUrl、NavigateUrl 和 AlternateText 元素值来实现。

AdRotator 类的继承路径是：

Object->Control->WebControl->AdRotator

因此除了特有的 AdvertisementFile 等属性和 AdCreated 事件外，它还具有许多从这些父类继承而来的其它属性和事件。可以查阅.NET Class Library 中对 AdRotator 类的描述来获得这些信息。

5.1.3　选择 HTML Server Controls 还是 Web Server Controls

读者可能发现，许多 HTML 服务器端控件和 Web 服务器端控件实际上是一一对应的，在经由它们生成的传送给客户端浏览器的 HTML 代码中，这两种控件实际上产生相同的 HTML 标记。例如 Web 服务器端控件<asp:Button runat=“server”/>和 HTML 服务器端控件<input type=“submit” runat=“server”/>都会产生一个<input type=“summit”/>标记。读者可能会纳闷：什么情况下使用哪一种控件呢？

实际上，在同一个 ASP.NET 文件中这两种控件是可以共存的。HTML 服务器端控件能完成的工作，使用 Web 服务器端控件也可以完成，并且十分简单。Web 服务器端控件的能力从理论上来讲也是完全可以用 HTML 控件来实现的，当然这可能比较复杂（例如 AdRotator 控件就是一个<a href=“NavigateUrl” img=“ImageUrl” runat=“server”/>控件加上一些时间响应代码）。

选择哪一种控件可以从下面的角度来考虑。

在下面情况下，使用 Web Server Control：

- 偏向于使用类似 Visual Basic 的编程模型时
- 编写的页面可能被多种类型的浏览器显示时
- 需要 Web Server Controls 所特有的功能，如 AdRotator、Calendar 等控件时
- 创建的应用包含了嵌套的控件，并希望在容器层捕获这些控件的事件时

在下面情况下，使用 HTML Server Controls：

- 偏向于使用类似 HTML 的对象模型
- 工作在已有的 HTML 页面上，并希望能够快速地加入 Web Forms 的功能时
- 控件会同时和服务器端及客户端的脚本交互时

这里只是提供一个选择控件类型的参考。读者应该根据具体的应用环境来选用合适的控件。

5.2　User Server Controls

5.2.1　User Server Controls 简介

User Server Controls 技术在 ASP.NET 开发初期被叫做 Pagelet（在那时候 ASP.NET 仍被称为 ASP+）。这是除 Code-Behind 技术之外的另一种将代码和内容分离、实现代码重用的新技术。Code-Behind 技术可以让多个不同的页面的代码从同一基类派生而来，从而具有相似的行为。而 User Server Controls 不仅可以重用代码，还可以重用部分用户界面。

我们的页面往往会包含多个（当然也可以是零个）Web Server Control，这些 Web Server Controls 是协同工作来完成某一项界面工作或者一部分商业逻辑任务的。例如显示一个用户登录框并获取用户输入，然后利用后端的数据库来检验用户身份。程序员往往希望将这些相关的控件和代码封装为单个的控件，从而在站点中甚至以后的工程中重用这些代码。因此 ASP.NET 引入了 User Server Control 的概念。使用 User Server Controls，程序员可以使用标准的 Web Server Controls 甚至第三方的 Server Controls，包装出可重用的自定义控件。这些控件的开发者可以将内部组件的交互隐藏起来，只暴露出必要的属性、方法和事件。而控件的使用者只需和控件的有限的几个属性、方法和事件打交道。

例如对于一个最简单的用户登录框控件，这样的控件包含了一个普通的 TextBox 控件，一个 TextMode 属性为“Password”的密码 TextBox 控件，一个提交按钮，以及一个登录失败时显示错误信息的 Label 控件。我们来看看接触这个控件都会是哪些人，以及他们要做的事情。

1. 控件的设计者

控件设计者可以只暴露出用来获取和调整控件左上角坐标的 Left 和 Top 属性、一个登录成功事件、一个登录失败事件。在事件参数中可以传递自定义的数据结构来通知控件使用者企图登录用户的 ID 和密码。在用户单击“登录”按钮时捕获该按钮的 Click 事件并查询数据库。在登录成功时，触发登录成功事件，并通过事件参数告诉控件使用者登录者是谁。在登录失败时，设置合适的错误信息（例如“用户不存在”，或者“错误的密码”），并触发登录失败事件，通过事件参数告诉外界谁企图登录。

2. 页面设计者

页面设计者只需决定该控件在 Web Form 页面上的位置，无法更改其中的文本框和按钮的相对位置和大小，这样重用了的登录框控件便有统一的外观。当然，如果控件允许设置控件的背景色，页面设计者会更高兴。

3. 页面逻辑程序员

编写页面逻辑代码的程序员只需订阅登录成功事件，在合法用户登录时重定向到合适的页面上去。如果对那些登录不成功的事件也感兴趣（例如可能希望将失败的用户 ID 写入日志，或者希望重定向到一个专门的页面上去而不仅仅满足于在登录框控件中显示一条简单的错误信息），程序员也可以订阅登录失败事件。订阅登录框的事件和订阅标准的 Web

Server Controls 的事件一样有两种方法，一种是在页面的控件标记中设置事件响应的属性，另一种是在程序逻辑中动态地挂接事件响应函数。

那么，这样做有什么好处呢？

从前的 ASP 也可以使用#include 指令，像 C 语言一样从外部文件引入代码和内容。User Server Controls 与之相比更为先进。这种先进之处类似于 C++中类库的引入和直接的通过#include 指令包含源代码的区别：

首先，User Server Controls 和其使用者的名字空间是分离的，使用者不必面对命名冲突的尴尬。并且在同一页面中同一种 User Server Control 可以被重复使用，因为每个控件都使用自己的名字空间。例如上面提到的登录框中，控件设计者可能将 Labe 控件命名为 lblInfo，而在页面设计者的工作中，同样有可能在 Web Form 页面中添加一个 lblInfo 控件来显示和登录模块风马牛不相及的信息。如果使用#include 指令，便会有命名的冲突。使用了 User Server Control 时，两位设计者是不必担心这个问题的，因为两个 lblInfo 在不同的作用域中，就好像一个全局变量和一个局部变量的关系一样。同样，如果页面设计者引入了多个登录框（这个例子有点不可思议，不过技术上完全有可能，更何况还有其他功能的 User Server Controls），这些登录框中各自的 lblInfo 控件是完全不冲突的，因为它们是不同对象的实例成员（Instance Member）。

第二，User Server Controls 类似于一个黑箱（程序员背景的读者会很熟悉这个概念），调用者可以通过控件提供的接口（即它的公共方法）和属性来和控件交互，而不必考虑其内部的实现细节。在这个例子中，页面设计者不必考虑登录框内部几个控件的空间排列和样式问题，页面程序员也不必考虑如何捕获和处理“登录”按钮事件以及到何处去验证登录者的身份。事实上有时在控件设计者看来，控件使用者多少有点不应该知道这些问题。

第三，User Server Controls 的编写细节与其调用者无关，可以使用不同的语言（.NET 支持的语言）来实现。例如控件设计者可能偏好使用 C#，而页面程序员则擅长 Visual Basic。User Server Controls 还可以编译后发布，从而隐藏实现的源代码。

User Server Controls 不能被单独请求，而只能包含在 ASP.NET Web Forms 页面中。当所在的页面被请求时，User Server Controls 会自动地被动态编译。下一节我们学习创建 User Server Controls，然后看看如何将它包含到一个页面中并被使用。

5.2.2　User Server Controls 的创建

User Server Controls 的编写和 Web Forms 页面（aspx 文件）的编写是非常类似的。它们之间最大的不同是 User Server Controls 文件不能包含<html>标记和<body>标记。另外 User Server Controls 文件是以 ascx 为扩展名的。

创建一个 User Server Control，第一步要做的是使用一个文本编辑器创建一个 ascx 后缀的文件。然后在该文件中创建代码声明段。代码声明段中，可以包含要暴露给调用者的属性、事件响应函数和其他任意希望包含进去的代码。在 User Server Controls 的内部实现代码中，程序员可以封装各种商业逻辑代码，例如和数据库连接来获得数据，这些数据或许在控件中包含的 Web Server Controls 中显示出来，或者作为 User Server Control 的属性暴露给外界。

我们首先创建一个 ascx 文件，在其中编写登录框的界面，代码如下：

程序清单 5-1：example-5-1.ascx

```
<%@Control
    ClassName="usrLoginBox"
%>

    <table style="background-color: lightblue;font: 10pt verdana;
border-width:1; border-style:solid; border-color:black;"
cellspacing=15>
    <tr>
    <td><b>User: </b></td>
    <td><ASP:TextBox id="User" runat="server"/></td>
    </tr>
    <tr>
    <td><b>Password: </b></td>
    <td><ASP:TextBox id="Pass" TextMode="Password"
runat="server"/></td>
    </tr>
    <tr>
    <td></td>
    <td><ASP:Button id="btnSummit" Text="Login" runat="server"/></td>
    </tr>
    <tr>
    <td ColSpan=2><ASP:Label id="lblInfo" runat="server"/></td>
    </tr>
    </table>
```

注意到我们在这个 ascx 文件的第一行使用了@Control 指令。该指令只能在 User Server Controls 的 ascx 文件中使用，它定义了 User Server Control 被 ASP.NET 页面解析器和编译器使用的特有属性。这里先指定了 ClassName 属性，指定这个 User Server Control 被动态编译时的类名。

另外我们没有在这个文件中使用<form>标记，因为 ASP.NET 不允许<form>标记的嵌套，为了避免和控件使用者页面中可能存在的<form>标记冲突，我们在 User Server Controls 不使用<form>标记或者控件。

然后我们创建一个 C#源文件，在其中编写这个 User Server Control 的 Code-Behind 类代码。

程序清单 5-2：example-5-1-code.cs

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

public class CLoginBox : UserControl
{
    protected   Button  btnSummit;
    protected   TextBox User;
    protected   TextBox Pass;
    protected   Label   lblInfo;
}
```

我们不希望控件使用者直接访问控件内部的 Web Server Controls，因此对象成员 btnSummit 等的访问权限都设成 protected，这样只有继承这个 CLoginBox 的控件才能访问

到这几个成员。

然后在前端的控件界面文件 example-5-1.ascx 中引入这个 Code-Behind 类。这是通过@Control 指令的 Inherits 属性和 Src 属性完成的。修改 example-5-1.ascx 文件中的@Control 指令如下：

```
<%@Control
    ClassName="usrLoginBox"
    Inherits="CloginBox"
    Src="example-5-1-code.cs"
%>
```

可以看到，这和 Web Forms 页面中引入 Code-Behind 代码的做法几乎完全一样，只不过@Page 变成了@Control 而已。

@Control 指令的格式如下：

```
<%@ Control attribute="value"[attribute="value"]…%>
```

可以使用的属性包括：AutoEventWireup、ClassName、ComplierOption、Debug、Description、EnableViewState、Explicit、Inherits、Language、Strict、Src、WarningLevel。可以发现，这些属性和@Page 指令的属性是很接近的，它们的取值和作用也基本上是一致的。所不同的是，@Control 指令的 Inherits 属性指定的控件的基类必须是一个从 UserControl 类继承而来的类，而@Page 指令的这个属性要求的是一个 Page 类的子类，这一点是显而易见的。

和@Page 指令一样，@Control 指令只能在文件中出现一次。不同的是，它只能出现在 ascx 文件中，而@Page 指令只能出现在 aspx 文件中。

接下来我们要创建 btnSummit 按钮的 Click 事件响应代码。这也和 Web Forms 页面中处理事件响应代码类似：首先为 CLoginBox 类添加一个 protected 的成员函数 btnSummit_Click，然后为 example-5-1.ascx 文件中定义（面向对象程序员请注意：这一个标记的实质是实例化基类 CLoginBox 中定义的 btnSummit 类成员）btnSummit 按钮的标记添加 OnClick=“btnSummit_Click”属性。

最后的一步是到目前为止最吸引人的：定义和触发 User Server Control 自己的事件。读者如果对.NET 的事件机制和 C#的代表机制不熟悉，请首先参考第 4 章 4.5.3 节的讨论。

首先我们定义一个数据结构，用来作为事件参数，在事件触发时向外部传递试图登录的用户的 ID 和密码。因为会在控件外创建这个数据结构的实例，它必须声明为 public。为了简单起见，我们可以定义它如下：

```
public class UserInfo
{
    public string User;
    public string Pass;
}
```

然后我们声明事件的 delegate。这里我们的两个事件的参数表完全可以一致，因此只在 example-5-1-code.cs 中声明一个 delegate：

```
public delegate void LoginEventHandler(object Sender, UserInfo e);
```

提示：从使用上讲，我们已经在第四章看到，delegate 类似于一个函数指针原型的声明。但本质上讲，delegate 是一种类型。在这里编译器在背后为我们创建了一个从

System.MulticastDelegate 类派生而来的 LoginEventHandler 类，读者可以自己尝试使用.NET SDK 中的 ILDasm.exe 工具来验证这一点。这样 delegate 对象就具有了类型安全的特性。

接着我们可以为 CLoginBox 类添加两个事件。事件是 C#中类的成员的一种，使用 event 关键字声明，并使用某个 delegate 作为事件的类型。同样，作为成员的一种，事件也有自己的访问权限和其他修饰符，例如 public、protected、private、virtual 等等。在这里我们要在 Page 的派生类中使用一个 UserControl 的派生类的事件，因此这两个事件都应该是 public。具体声明如下：

```
public class CLoginBox : UserControl
{
    public event LoginEventHandler LoginSuccess;
    public event LoginEventHandler LoginFail;
    // …
}
```

触发事件时，我们可以像调用 C/C++中函数指针指向的函数一样，调用事件并传递适当的参数。例如我们在某个成员函数中触发一个 LoginSuccess 事件时可以这么写：

```
LoginSuccess(this, userinfo);
```

在这个例子中，我们会在 btnSummit 按钮的 Click 事件处理函数中决定触发哪个事件，Click 事件处理函数已经被我们声明为 CLoginBox 类的成员函数了。我们现在往这个函数填充代码。为了方便描述、节省篇幅，我们假定只有 Michael 一个用户，Michael 的密码为 password。

```
protected void btnSummit_Click(Object Sender, EventArgs e)
{
    // initialize the event argument
    UserInfo arg = new UserInfo(User.Text, Pass.Text);

    // verify the user's identification
    if ("Michael" == User.Text && "password" == Pass.Text)
    {
        // if the LoginSuccess event is subscribed
        if (null!=LoginSuccess)
            // fire the event
            LoginSuccess(this, arg);
    }
    else
    {
        if ("Michael" == User.Text && "password" != Pass.Text)
            lblInfo.Text = "Invalid Password!";
        else
            lblInfo.Text = "Invalid User ID";
        // if the LoginFailed event is subscribed
        if (null!=LoginSuccess)
            // fire the event
            LoginFail(this, arg);
    }
}
```

注意在触发事件前，我们先判断相应的事件是否为 null。回忆我们订阅事件时是怎么做的：我们用 new 创建一个 delegate 对象，同时把事件处理函数作为该 delegate 对象的构造函数参数，最后用重载后的+=运算符将该对象添加给对象的事件成员。我们可以猜想到，

每个对象的事件其实是一个 delegate 对象，它在内部维护一个引用或者引用列表，指向其他该事件的订阅者（也是一个 delegate 对象）或者 null，并通过重载+=和-=运算符来操纵该引用。当然这些操作都是编译器自动完成的。如果事件没有被订阅，则控件对象的这个事件是一个 null。在声明事件的类或者结构体（struct）中，某些事件可以被视为 field 看待，条件是该事件声明时不带有 abstract 修饰符，并且不带有事件附属声明块（event accessor declaration）。关于事件附属声明块，请参考 C#语言参考。

到这一步为止，我们已经完成了一个最简单的登录框 User Server 控件的创建。这里没有包含任何调整样式和位置的属性，完整的代码如下：

程序清单 5-3：example-5-1.ascx

```
<%@Control
    ClassName="usrLoginBox"
    Inherits="CLoginBox"
    src="example-5-1-code.cs"
%>

<table style="background-color:lightblue;font: 10pt verdana;
border-width:1; border-style:solid; border-color:black;"
cellspacing=15>
<tr>
<td><b>User: </b></td>
<td><ASP:TextBox id="User" runat="server"/></td>
</tr>
<tr>
<td><b>Password: </b></td>
<td><ASP:TextBox id="Pass" TextMode="Password"
runat="server"/></td>
</tr>
<tr>
<td></td>
<td><ASP:Button id="btnSummit" Text="Login"
OnClick="btnSummit_Click" runat="server"/></td>
</tr>
<tr>
<td ColSpan=2><ASP:Label id="lblInfo" runat="server"/></td>
</tr>
</table>
```

程序清单 5-4：example-5-1-code.cs

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

// Data Structure of argument for login events
public class UserInfo
{
    public UserInfo(string User, string Pass)
    {
        this.User = User;
        this.Pass = Pass;
    }
    public string User;
    public string Pass;
```

```
}

// Delegate for login Events
public delegate void LoginEventHandler(Object Sender, UserInfo e);

public class CLoginBox : UserControl
{
    protected   Button  btnSummit;
    protected   TextBox User;
    protected   TextBox Pass;
    protected   Label   lblInfo;

    public event LoginEventHandler LoginSuccess;
    public event LoginEventHandler LoginFail;

    protected void btnSummit_Click(Object Sender, EventArgs e)
    {
        // initialize the event argument
        UserInfo arg = new UserInfo(User.Text, Pass.Text);

        // verify the user's identification
        if ("Michael" == User.Text && "password" == Pass.Text)
        {
            // if the LoginSuccess event is subscribed
            if (null!=LoginSuccess)
                // fire the event
                LoginSuccess(this, arg);
        }
        else
        {
            if ("Michael" == User.Text && "password" != Pass.Text)
                lblInfo.Text = "Invalid Password!";
            else
                lblInfo.Text = "Invalid User ID";
            // if the LoginFailed event is subscribed
            if (null!=LoginSuccess)
                // fire the event
                LoginFail(this, arg);
        }
    }
}
```

5.2.3　在 Web Forms 页面中引入 User Server Controls

User Server Controls 不能被浏览器单独请求，User Server Controls 必须被包含在 Web Forms 页面中。当用户请求一个包含了 User Server Controls 的 Web Forms 页面时，User Server Controls 和 Page 一起被 JIT 编译器动态编译、执行并生成 HTML 代码（实际上不完全是这样，编译这个过程有时被跳过去了，我们在提到页面缓冲技术时在详细讨论，现在我们暂且认为它们首先被编译）。这些控件的生存周期和页面的生存周期是一致的。

为了在页面中引入 User Server Controls，第一步要做的是使用@Register 指令，告诉页面要使用的 User Server Controls 在哪里和如何“称呼”它。

@Register 指令用来为类和名字空间引入别名，从而使外部定制的 User Server Controls 在指令所在文件中可用。其格式如下：

```
<%@ Register Tagprefix="tagprefix" Namespace="namespace" Assembly="assembly" %>
```

```
<%@ Register Tagprefix="tagprefix" Tagname="tagname" Src="pathname" %>
```

Src 属性指定了 User Server Controls 所在的源文件。Assembly 属性指定了 User Server Controls 所在的库文件。根据我们要使用的 User Server Controls 以何种形式（二进制文件或者源文件）提供，我们选取这两种格式的一种。tagprefix 属性定义了名字空间的别名，tagname 属性定义了类的别名，Namespace 属性定义了 tagprefix 与之关联的名字空间。

在第一种格式中，我们用 Assembly 属性指向某个二进制文件。在该文件中存在了一个或者多个名字空间。我们要使用的 User Server Controls 存在于这些名字空间中的一个或者多个。因此我们使用 tagprefix 属性指定的值来指代该名字空间，然后我们就可以用它来引用这些名字空间中的控件。假设二进制文件 assemblyA.dll 中有两个名字空间：ACompany.Input 和 ACompany.Output，在 ACompany.Input 中有个 AdvancedLoginBox 控件类，我们的@Register 指令可以这么写：

```
<%@ Register
    Tagprefix="ainputcontrols"
    Namespace="ACompany:Input"
    Assembly="assemblyA"
%>
```

注意 Assembly 属性中不包括文件的后缀，名字空间的层次用“:”号分隔。这样我们就把 assemblyA.dll 中的 ACompany.Input 名字空间映射到 ainputcontrols 别名上。我们可以使用 ainputcontrols:AdvancedLoginBox 来引用 AdvancedLoginBox 控件。

在第二种格式中，我们用 Src 指明想引入的 ascx 文件。一个 ascx 文件其实是一个 UserControl 类，该类是由 JIT 编译器自动动态编译的，并且类名是不确定的（回忆上节中@Control 指令的 ClassName 属性，只有指定了该属性，其类名才是确定的）。因此，我们通过连用 tagprefix:tagname 来指明该类。

例如我们前面创建的 LoginBox 控件存在于 example-5-1.ascx 文件中，因此引入该控件的@Register 指令为：

```
<%@ Register Tagprefix="usrctrl" Tagname="LoginBox" Src="example-5-1.ascx" %>
```

这个例子中，我们把它映射为 usrctrl:LoginBox 控件。

值得指出的是，@Register 指令并不一定只用于 Web Forms 页面文件中，它同样可以用在 Web Forms 用户控件文件中。例如我们希望在 User Server Control 中包含和封装其他 User Server Controls 的时候，我们也同样可以在 ascx 文件中使用合适的@Register 指令。

下一节，我们讨论如何在 Web Forms 页面中使用 User Server Controls。

5.2.4　在 Web Forms 页面中使用 User Server Controls

在 Web Forms 页面中使用 User Server Controls 需要使用定制的服务器端控件语法：

```
<tagprefix:tagname id="id" attributename="value"
attributename-propertyname="value" eventname="eventhandlermethod"
runat="server" />
```

或者

```
<tagprefix:tagname id="id" runat="server"> </tagprefix:tagname>
```

tagprefix 属性和 tagname 属性的值必须是通过上节提到的@Register 指令引入的。它们连用指定了要使用的 User Server Control 的类型。

id 属性是一个在该命名域内独一无二的标识符，程序员使用该标志符来在程序代码中引用和操纵该控件。

控件的属性可以在程序逻辑中（比如在页面的 Page_Load()方法中）赋值，也可以在标记中直接赋值，只需在 attributename 处写上属性名，在 value 处写上属性值就可以了。在某些情况下，某些属性是一个对象，并且有自己的属性。为了在声明中定义这些属性，可以使用 attributename-propertyname=“value”语法，在 attributname 处写上对象属性名，在 propertyname 处写上对象属性的属性，最后在 value 处写上属性值。

处理 User Server Controls 的事件，也和标准的 Web Server Controls 同样可以有两种方法：在声明标记处直接挂接，或者在程序代码中使用+=运算符。在标记处挂接时，在 eventname 处写上事件名，在 eventhandlermethod 写上事件处理函数的函数名。

我们创建一个简单的 Web Forms 页面，页面中只包含已创建的 LoginBox，以及一个用来测试捕获控件的用户登录事件的 Label 控件。完整的代码如下：

程序清单 5-5：example-5-1.aspx

```
<%@ Page language="C#"%>
<%@ Register tagprefix="usrctrl" tagname="LoginBox"
 src="example-5-1.ascx" %>

<script language="C#" runat="server">
    void Page_Load(Object Sender, EventArgs e)
    {
        if (!IsPostBack)
        {
            info.Text = "Log in first please";
        }
    }
    void LoginSuccess(Object Sender, UserInfo e)
    {
        info.Text = "You have logged in successfully as " + e.User;
    }
    void LoginFail(Object Sender, UserInfo e)
    {
        info.Text = "Invalid user (id: " + e.User + ") tried password:
        " + e.Pass;
    }
</script>

<html>
<body>
<form runat="server">
<usrctrl:LoginBox runat="server"
    id="Login"
    OnLoginSuccess="LoginSuccess"
    OnLoginFail="LoginSuccess"
/>
</form>
<asp:Label id="info" runat="server" />
</body>
</html>
```

请注意这个 aspx 文件中使用到的各种我们已经学习过的 Web Forms 技术。我们使用了代码声明段而不是 Code-Behind 代码来放置三个页面的成员函数，因为页面和函数都比较

简单，并且我们不考虑在以后重用这些代码。我们使用了从 Page 类继承而来的 IsPostBack 属性来初始化 Label 控件的文本。我们在声明 usrctrl:LoginBox 控件的标记中直接挂接两个事件处理函数，注意到我们使用的 eventname 不是 LoginBox 控件定义的两个事件名，而是这两个事件名前加“On”。当然我们也可以不这么做，而是在 Page_Load 成员函数中添加这样的两行代码：

```
Login.LoginSuccess += new LoginEventHandler(LoginSuccess);
Login.LoginFail += new LoginEventHandler(LoginFail);
```

这里我们则不可写成 Login.OnLoginSuccess 或 Login.OnLoginFail。另外细心的读者会发现，两个事件处理函数的函数名和控件暴露出来的事件名是一样的。这有什么必然的联系吗？答案是：没有，一点也没有。请注意本质的东西：控件的事件是控件的成员，其名字域是控件类，而这里的两个事件处理函数是页面类的成员，名字域是这个页面类，它们分属于不相交的名字域，因此不必担心会有命名上的冲突。事实上，事件处理函数可以是任何名字，前提只是不和页面中的其他名字冲突。

运行这个页面，只有用户 ID 为 Michael 并且输入密码为 password 时，才会成功登录。这时页面会捕捉到成功登录的事件并在 Label 控件上显示成功的信息，当然，更一般的做法是在 LoginSuccess 事件处理函数中编写合适的代码，将页面重定向到合适的页面上。如果登录失败，控件本身会提示“Invalid User ID”或者“Invalid Password!”，如果我们对这个事件感兴趣，可以捕获这个事件。在这里我们的确捕获了这个事件并利用事件参数，在 Label 上显示非法用户的信息。更有建设性的做法可能是将用户引领到一个注册新用户的页面上去，或者先定向到一个 aspx 文件上，在 Page_Load 中将该事件记录到日志中，再定向到合适的内容上。但无论你想做什么，都可以从捕获这些事件开始。这里只是演示这种可能性。

5.3　Validation Server Controls

在从前的工作中，对 HTML 输入元素的值的检验是一项十分困难的工作。程序员可以将检验代码放在客户端运行，以此来减少网络传输量；也可以将它放在服务器端运行，以此来获得最大的控制权；也可以混合使用这两种方法，来获得性能和控制的平衡。无论哪种方法，程序员都必须编写代码来检验输入值，编写错误信息来指引用户更正他们的输入。这个过程无论对用户、开发人员还是服务器来说，都是一个负担。

ASP.NET 中引入了输入检验控件（Validation Server Controls），来使得输入检验的工作变得比从前的任何时候都轻松。开发人员可以将他们的精力更好地集中在商业逻辑的设计和开发上，无须为这种繁琐而乏味的输入检验过程花费过多的精力。

ASP.NET 共有 6 种输入检验控件：

表 5-4　ASP.NET 的校验控件

检验控件类型	作用
RequiredFieldValidator	用来保证用户在必要的输入控件中填写了内容
CompareValidator	用来将输入控件的值和一个常数、或者其他控件的值和属性、或者来自数据库的值进行比较

（续表）

检验控件类型	作用
RangeValidator	用来检验用户的输入是否处于某个特定的区间中，区间的边界必须是常数
RegularExpressionValidator	用来检验控件的值是否和给定的规则表达式匹配
CustomValidator	用来创建定制的客户端和服务器端检验代码
ValidationSummary	用来汇总显示页面中所有的其他种类的检验控件的错误信息

这些控件使用时方法是基本相通的，而且十分简单，在设计页面时，设置这些控件的属性来与其他用于获取输入的控件联系起来，然后设置验证的规则。具体控件的使用方法请读者在需要时自行查阅 SDK 文档。

5.4　本章小结

本章承接着上一章的内容，所以接着 Web Forms 介绍 HTML Server Control 和 User Server Controls 等控件，特别指出本章中还谈到了 ASP.NET 中新推出的 Validation Controls（验证控件）。

第 6 章　.NET 实现的 XML DOM

本章我们将介绍.NET Framework 中的 DOM 接口的使用。DOM 是处理 XML 文档的重要编程接口。.NET Framework 实现了对 DOM 规范的良好支持，并提供了一些扩展技术，使得开发人员对 XML 文档的处理更加方便。ASP.NET 基于.NET Framework，可以充分使用.NET Class Library 对 DOM 的支持。

本书介绍的主要是 DOM 的基本知识和如何在.NET 应用程序开发过程中使用 DOM 支持。这些知识并不局限于 ASP.NET，由于.NET 对 DOM 的支持源自底层的类库，在其他开发任务中（例如 Windows Forms 等）同样可以以类似的方法来使用 DOM 接口。

6.1　DOM 简介

DOM（Document Object Model）是一个程序接口，应用程序和脚本可以通过这个接口访问和修改 XML 文档数据。DOM 是根据 OMG（Object Management Group）和 IDL（Interface Definition Language）定义的。

XML 语言只是一种信息交换的载体，是一种信息交换的方法。在今后的网络技术中，XML 会是实现新技术的基础，比如 Web Services。这使得对充满信息的 XML 文档的处理变得十分重要。怎样从 XML 文档中获得想要的信息呢？

外部的处理程序是使用一种称为接口的技术来访问 XML 文档的。正像接口在数据库方面的应用一样，通过接口，不必关心数据库使用的是 ORACLE 或是 SYBASE，还是 SQL Server 等等，只要使用 ODBC/JDBC 接口就可以对数据库进行访问和数据操作。DOM 接口使得应用程序对 XML 文档的访问变得简单。

W3C 制定了 DOM 接口标准用来规范访问和修改 XML 数据的程序接口。DOM 只是一种规范，是对象化的 XML 数据接口。这也使得 DOM 具有跨平台的特性，因为 DOM 是通用标准，可以不依赖于操作平台和实现 DOM 的编程语言。DOM 可以用 C++、Java、或 Smalltalk 等语言实现，像 NT、Windows 98/95 和 UNIX 平台都可以支持 DOM。本书将介绍在 Windows NT 环境下，如何使用 ASP.NET 实现的 DOM 接口访问 XML 数据。

DOM 接口定义了一系列对象来实现对 XML 文档数据的访问和修改。DOM 接口将 XML 文档转换为树型的文档结构。在这棵对象树是 XML 文档内元素之间关系的反映，通过这棵树，可以访问和修改 XML 文档的数据。应用程序可以通过树型模型对 XML 文档数据进行层次化的访问。文档中的信息，包括数据、数据的意义和数据的关系都由 DOM 接口转换为树型结构的节点和节点的关系。应用程序通过 DOM 可以通过对树的各种操作实现对 XML 文档的操作。这包括：

- 遍历树的所有节点
- 通过对 DTD 或 Schema 检查 XML 文档的有效性
- 访问树的节点，得到所需的节点信息。比如节点的值，属性节点的属性值等

- 创建新节点。可以是元素节点、属性节点、注释节点、指令节点或文本节点
- 甚至可以创建全新的 XML 文档

DOM 将 XML 当成树型结构文档来访问。这个树形结构有一个根节点——Document 节点。树的所有节点都是由根节点通过继承得到的。在这个树形结构中，XML 文档的所有元素包括标记、属性、注释等都有相应的节点表示。下面是一个简单的 XML 文档，其树型结构由图 6.1 给出。

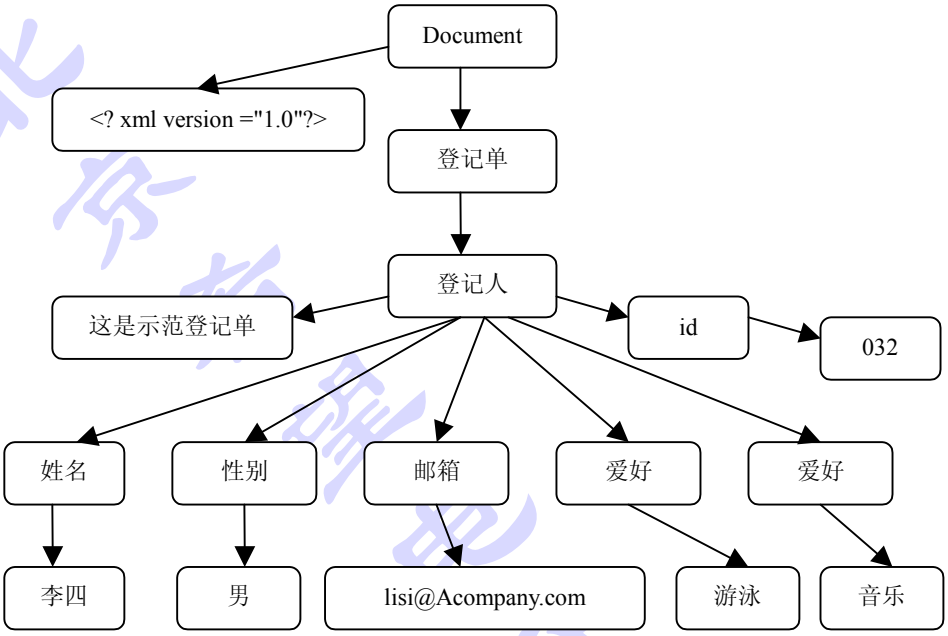


图 6.1　树型结构图

程序清单 6-1：example-6-1.xml

```
<?xml version="1.0" ?>
<登记单>
    <登记人 id="032">
        <!--这是示范登记单-->
        <姓名>李四</姓名>
        <性别>男</性别>
        <邮箱>lisi@Acompany.com</邮箱>
        <爱好>游泳</爱好>
        <爱好>音乐</爱好>
    </登记人>
</登记单>
```

在树型结构中，各个节点的内容和节点的关系表示了 example-6-1.xml 的所有内容。每个节点有节点名和节点类型。节点类型可以是注释、文本、属性等。XML 文档的树型结构比较复杂，实例是很简单的 XML 文档，如果是复杂的 XML 文档，用图形表示就十分繁琐。事实上，由于 DOM 分析器将整个 XML 文档转换为树型结构放进内存中，使得对于复杂的大型 XML 文档来说，需要较大的内存空间来放这棵树。

但是，由于所有的 XML 文档信息都被树型结构所包含，使得可以对 XML 文档数据进

行随机访问。XML 数据都有相应的表示方式，这也使得遍历 XML 文档显得繁琐、费时。

DOM 接口是通过 DOM 的分析器来处理 XML 文档的。

目前的 DOM 标准是分级别制定的。在 DOM 级别 1（DOM LEVEL 1）中，制定了一系列对象，用来表示 XML 文档结构。这些对象包括 Document，Element 等。对于表达不同信息的对象，有各种相应的操作来完成对文档数据的访问和修改。比如：对于 Document 对象，有“createElement、getElementsByTagName”等，实现创建新节点、获得元素值等操作。这些对象和对象操作将在下一节中详述。

在 DOM LEVEL1 标准中制定的 XML DOM 树的节点类型有以下几种：Document，NodeList，Node，NamedNodeMap，Element，Attr，CharacterData，Text，Comment，DocumetType，ProcessingInstruction。正是这些节点类型和对象操作方法，可以完成对外部应用程序对 XML 文档的操作。

目前，DOM LEVEL 2 正处于草案阶段，在 DOM LEVEL 2 中，将加入对名字空间、样式单、事件等的支持。下面是 DOM2 新增的功能接口：

- 层叠样式表（CSS2）：提供 CSS2 兼容的方法
- 文档遍历（Document traversal）：提供遍历文档层次的接口
- 文档范围（Document range）：提供分割文档范围的接口
- 事件（Events）：提供各种事件的接口
- 视图（View）：提供视图与文档的联系
- 样式表（Style Sheet）：提供访问和修改样式表的方法

6.2　.NET 中的 DOM 对象模型

6.2.1　.NET 的 DOM 实现

正如我们已经了解到的，Document Object Model（DOM）是一个 XML 文档在内存中的树型缓冲表示。这里的缓冲意味着对内存树的修改并不是立即反映到 XML 文档的。DOM 为程序员提供了一个遍历、操纵和修改 XML 文档的编程接口。

Microsoft .NET Framework 提供了对 XML DOM（Document Object Model）对象模型的支持。.NET Framework SDK Beta 2 实现了 W3C Document Object Model （Core）Level 1（www.w3.org/TR/REC-DOM-Level-1/level-one-core.html）和 Document Object Model Core（www.w3.org/TR/DOM-Level-2-Core/core.html）。这种支持是通过一系列相关的类来实现的。.NET 的 DOM 实现完全支持 W3C 的 DOM 规范，除此之外，Microsoft 还扩展了 W3C DOM Level 1 和 Level 2 中的类，使得对 XML 数据的处理更加容易。这种扩展包括了与关系型数据模式的协作和同步。我们将在第 8 章学习 XML 如何与关系型数据协作和同步。

W3C Document Object Model（DOM）Level 1 规范定义了两组编程类：基础类和扩展类。基础类组包括了用来编写操纵 XML 文档的应用程序所需要的类；扩展类被定义用来简化开发人员的编程工作的类。

W3C DOM Level 2 规范包含了对 XML 名字空间和其他特性的支持。

.NET Class Library 中支持 DOM 的类主要存在于 System.Xml 和 System.Xml.XmlDocument 名字空间中。这些类同样分为两个层次：基础类和扩展类。这两个层次并不完全对应于 W3C DOM 标准的两个层次。前者提供了 Core DOM Level 1 标准中用来描述底层的基础接口的集合的特性，这些基础接口可以用来表示任何结构化的文档，更特殊化地，它们可以被用来定义表示一个 XML 文档所需要的扩展接口。后者实现了 Core DOM Level 1 中的所有基础接口和 Core DOM Level 2 中定义的其它接口。

在基础类层次中，.NET 类库包含了三个类：XmlNode 用来表示文档树中的单个节点，这是一个抽象基类，在扩展类层次中我们会有这个基类的其他具体派生类的实现，这个基类描述了 XML 文档中各种具体节点类型的共性；XmlNodeList 类用来表示一个节点的有序集合，它提供了对迭代操作和索引器的支持。XmlNamedNodeMap 类用来表示一个节点的集合，该集合中的元素（即节点）可以使用节点名或索引来访问，支持了使用节点名称和迭代器来对属性集合的访问，并且包含了对名字空间的支持。

扩展类层次包括了众多的类，主要的类有以下几个，它们都是由 XmlNode 类派生出来的：

- XmlDocument 类用来表示 XML 文档的顶层节点，它实现了 W3C DOM（Core）Level 1 和 DOM Core Level 2
- XmlElement 类表示文档中的一个元素对象
- XmlAttribute 类表示 XmlElement 对象的一个属性，该属性的合法值和缺省值信息由 DTD 或 Schema 定义
- XmlAttributeCollection 类表示了 XmlElement 对象的属性集合。这些属性的合法值和缺省值信息由 DTD 或 Schema 定义
- XmlComment 类表示 XML 文档中的注释内容
- XmlDeclaration 类表示 XML 的声明节点
- XmlDocumentType 类表示 XML 文档的 DOCTYPE 声明节点
- XmlEntity 类表示 XML 文档中一个解析过或未解析过的实体
- XmlEntityReference 类表示一个实体引用
- XmlText 类表示了一个元素或属性的文本内容

6.2.2　.NET DOM 对象模型的主要类

类库中实现 DOM 接口的两个最主要的类是 XmlDocument 和 XmlNode。我们来详细看看这两个类。

1. XmlNode 基类

对于 XmlNode 类，其主要成员属性有：

表 6-1　XmlNode 类的主要成员属性

属性	描述
Value	返回或设置节点的值
Attributes	返回一个 XmlAttributeCollection 对象，这个对象包含了节点的所有属性节点
ParentNode	返回节点的父节点
ChildNodes	返回一个 XmlNodeList 对象，该对象包含了这个节点的所有子节点
FirstChild	返回节点的第一个子节点
LastChild	返回节点的最后一个子节点
HasChildNodes	返回一个布尔值，该值指明了这个节点是否包含了子节点
PreviousSibling	返回紧跟着该节点的上一个兄弟节点
NextSibling	返回紧跟着该节点的下一个兄弟节点
InnerText	返回或设置表示该节点及其子节点的字符串，包括了节点本身
InnerXml	返回或设置该节点包含的子节点的标记文本
OuterXml	返回该节点及其下子节点的标记文本
IsReadOnly	返回一个布尔值，该值指明了该节点是否是只读节点
Name	返回节点的全名（qualified name）
LocalName	返回节点的本地名（local name）
Prefix	返回或设置节点的名字空间前缀
NodeType	返回一个 XmlNodeType 对象，指明该节点的类型

XmlNode 类的主要成员方法有：

表 6-2　XmlNode 类的主要成员方法

成员方法	描述
PrependChild	将指定节点插入到子节点列表的最前端。返回加入的节点。如果要加入的节点已存在于树中，它首先会被从原位置删除。如果当前节点类型不允许新节点的加入，或者新节点是当前节点的祖先节点，则抛出 InvalidOperationException 异常
AppendChild	用来将一个指定的节点添加到子节点列表的最后
InsertAfter	接受两个同为 XmlNode 类型的节点对象，将第一个对象插入到第二个对象之后，其中第二个对象是该节点的一个子节点。当第二个对象为 null 时，将第一个对象插入到子节点列表的开头。当新节点是一个 XmlDocumentFragment 对象时，该新节点的所有子节点也同时以同样的顺序被插入。如果新节点已存在于树中，它首先会从原位置被删除。如果当前节点不允许插入新节点类型，或者新节点是当前节点的祖先节点时，该方法会

(续表)

成员方法	描述
	抛出一个 InvalidOperationException 异常。当新节点是从另一个不同的文档中创建而来时，或者第二个节点参数不是当前节点的子节点，或者当前节点是只读节点，都会抛出一个 ArgumentException 异常
InsertBefore	类似于 InsertAfter 方法，接受两个 XmlNode 对象参数，将第一个对象插入到第二个对象之后，其中第二个对象是当前节点的一个子节点
CloneNode	接受一个布尔值参数。该参数为 true 时，递归地复制节点本身及其下的所有子节点；为 false 时，仅复制节点本身。返回复制出的节点。当该节点不能被复制时，会抛出一个 InvalidOperationException 异常
Clone	相当于调用 CloneNode 方法并传递 true
RemoveAll	删除所有子节点和属性节点
RemoveChild	删除传递给该方法的子节点，如果该节点不是当前节点的子节点，抛出一个 ArgumentException 异常
ReplaceChild	用一个节点来替换一个子节点
SelectNodes	通过指定一个 string 类型的 XPath 表达式来选择节点，返回一个包含选中节点的 XmlNodeList 对象
SelectSingleNode	指定一个 XPath 表达式，返回与该表达式匹配的第一个节点
WriteContentTo	将节点的所有子节点写到指定的 XmlWrite 对象中
WriteTo	将当前节点写到指定的 XmlWriter 对象中

由这两个表可以看见，XmlNode 类定义了一整套灵活的遍历、访问和操纵 DOM 树中节点的方法。这些方法被具体的节点类继承并在特殊场合下被重载或重定义，例如对于文档节点，因为是根节点，其 ParentNode 属性被重载为返回 null 引用。然而 XmlNode 类是一个抽象基类，因此我们无法直接创建 XmlNode 对象。我们会在其他 XmlNode 类的派生类中看到这些方法和属性是如何被使用的。请读者仔细阅读上面两个表，我们继续介绍 XmlNode 的一个重要派生类：XmlDocument。

2. XmlDocument 对象

XmlDocument 节点类型是 XmlNode 的一个派生类，它用来表示一个 XML 文档对象。除了从 XmlNode 继承而来的众多属性外，它还有几个重要属性：

DocumentElement 属性返回一个 XmlElement 对象，该对象是该文档的根元素。对 XML 文档中数据的访问，往往是从该元素入手的。

DocumentType 属性返回一个 XmlDocumentType 对象，该对象表示了文档中的 DOCTYPE 声明节点。

XmlResolver 属性用来设置一个 XmlResolver 对象，从而可以向文档中装载 DTD 或外部实体引用。

另外，某些属性被 XmlDocument 类重载，以适应 XmlDocument 的特殊地位和行为。其中的一些返回值是与 XmlDocument 类型相关的，这些属性包括：

- BaseURI 属性返回 XML 文档的位置
- NodeType 属性被重载为返回 XmlNodeType.Document 值

除了从 XmlNode 类继承而来的成员方法外，XmlDocument 类还具有许多特有的方法。

Load 方法和 LoadXml 方法是最常用的方法，它们用来向 XmlDocument 对象装载 XML 数据。Load 方法有多个重载版本，可以接受来自 Stream 对象、TextReader 对象或 XmlReader 对象的 XML 数据，也可以接受一个表示 XML 文件路径和文件名的字符串。LoadXml 方法则从一个指定的字符串来装载 XML 文档。例如

```
XmlDocument xmldoc = new XmlDocument();
xmldoc.Load("www.somesite.com/data.xml");
```

或者

```
System.IO.FileStream fs = new System.IO.FileStream("data.xml");
XmlDocument xmldoc = new XmlDocument();
xmldoc.Load(fs);
fs.Close();
```

如果使用 LoadXml，代码类似于：

```
XmlDocument xmldoc = new XmlDocument();
xmldoc.LoadXml("<authors nationality='China PRC'" +
                    "<author>" +
                        "<name>Michael</name>" +
                    "</author>" +
                "</authors>");
```

将 XML 保存到特定位置，可以使用 Save 方法。同样，Save 方法接受 Stream 对象、TextWriter 对象或 XmlWriter 对象作为参数来指定保存的目的地，也可以指定文件路径和名称来将 XML 文档保存到磁盘上。例如将文档保存到磁盘，可以写成：

```
XmlDocument xmldoc = new XmlDocument();
xmldoc.Load("source.xml");
xmldoc.Save("dest.sml");
```

将 XML 文档输出到标准输出流对象，可以使用 System.Console 对象的 Out 属性，这在调试过程中尤为有用：

```
xmldoc.Save(Console.Out);
```

为了添加子节点，除了使用 XmlNode 类的 AppendChild 方法外，还可以使用一系列的 Create 方法，用来在文档中创建各种类型的节点：

CreateAttribute 方法创建一个节点（XmlAttribute 对象），创建对象后使用文档中的某个 XmlElement 对象的 SetAttributeNode 方法来将新创建的属性赋值给该 XmlElement 对象表示的元素。例如：

```
XmlDocument xmldoc = new XmlDocument();
xmldoc.Load("data.xml");
XmlAttribute attr = xmldoc.CreateAttribute("NewAttr").Value;
attr.Value = "AttributeValue"
xmldoc.DocumentElement.SetAttributeNode (attr);
```

CreateComment 方法创建一个 XmlComment 对象。创建对象后使用某个节点的 insert 系列方法或 AppendChild 等方法来将新节点插入到文档中的适当位置。根据 W3C XML 规范的 1.0 版本，注释节点只能被包含在文档节点和元素节点中。例如创建并添加一个包含在文档节点中的注释节点的代码如下：

```
XmlComment newcomment = xmldoc.CreateComment("This is a comment for
```

```
the document");
    xmldoc.InsertBefore(newcomment, xmldoc.DocumentElement);
```

CreateElement 方法创建一个元素对象，使用其 InnerText 属性来设置其内容，并使用 AppendChild 等方法将元素添加到文档中去。

其他 Create 方法的用法大同小异，使用它们可以创建各种类型的节点，设置其值和其它属性，并将它们插入到 DOM 树的适当位置上。在此我们不再赘述。

另一个重要方法是 ImportNode。到目前为止我们使用到的添加节点的方法，添加的节点均是原本已存在于文档中的或者使用同一文档的 Create 方法创建的。如果想将来自其他文档对象的节点添加到现有文档中，我们可使用 ImportNode 方法。在将来自外部的 XmlNode 对象传递给 ImportNode 的同时，我们还必须传递一个布尔值来指定是否同时将外部节点的子节点一同导入进来。导入外部节点时，得到的是外部节点的一个拷贝，外部节点并不会被删除。ImportNode 方法得到拷贝后的节点，该节点是没有父节点的，可以使用 Append、Insert 方法将它插入到文档的某个位置上。理论上讲，根元素也是可以被导入的，但 DOM 中规定了文档中最多只能有一个根元素，因此如果必须将从外部导入的元素作为根元素，必须先删除原来的根元素。例如：

```
// load xml documents
XmlDocument xmldoc1 = new XmlDocument()
xmldoc1.Load("data1.xml");
XmlDocument xmldoc2 = new XmlDocument()
xmldoc2.Load("data2.xml");

// import the new root element from data2.xml
XmlNode newroot = xmldoc1.ImportNode(xmldoc2.DocumentElement);
// remove the existing root element of data1.xml
xmldoc1.RemoveChild(doc.DocumentElement);
// append the new root element into data1.xml
xmldoc1.AppendChild(newroot);
```

3. 其他类对象的使用

其他的节点类包括表示元素节点的 XmlElement、表示属性节点的 XmlAttributes、表示注释节点的 XmlComment 等等。它们的继承关系如表 6-3 所示。

表 6-3　其他节点类的继承关系

类	继承路径
XmlElement	XmlLinkedNode
XmlAttribute	XmlNode
XmlCharacterData	XmlLinkedNode
XmlText	XmlCharacterData
XmlComment	XmlCharacterData
XmlCDataSection	XmlCharacterData
XmlWhitespace	XmlCharacterData
XmlSignificantWhitespace	XmlCharacterData
XmlProcessingInstruction	XmlLinkedNode

（续表）

类	继承路径
XmlDocumentFragment	XmlNode
XmlDocumentType	XmlLinkedNode
XmlEntityReference	XmlLinkedNode
XmlEntity	XmlNode
XmlNotation	XmlNode

每种节点都有 Name 属性和 Value 属性。Name 属性返回节点的全名，Value 属性返回节点的值。对于不同类型的节点，其返回值有所不同：

表 6-4　节点类的返回值

节点类型	Name 属性返回值	Value 属性返回值
Attribute	属性的全名	属性的值
CDATA	#cdata-section	CDATA 节的内容
Comment	#comment	注释的内容
Document	#document	null 引用
DocumentFragment	#document-fragment	null 引用
DocumentType	文档类型的名称	null 引用
Element	元素的全名	null 引用
Entity	实体的名称	null 引用
EntityReference	实体引用的名称	null 引用
Notation	Notation 的名称	null 引用
ProcessingInstruction	处理指令的目标名	除了目标外的所有内容
Text	#text	文本节点的内容
Whitespace	#whitespace	空白字符串
SignificantWhitespace	# significant-whitespace	空白字符串
XmlDeclaration	#xml-declaration	声明的内容，介于“<?xml”和“?>”之间的所有内容

每种节点还具有 NodeType 属性，可以使用该属性来判断节点的类型。

使用这些节点从 XmlNode 基类继承而来的公有属性 HasChildNodes、FirstChild、LastChild、NextSibling、PreviousSibling 属性和 ChildNodes 集合属性，我们可以写出遍历文档树各节点的递归程序，例如深度优先的递归程序的结构类似于：

```
public void DOMDepthFirst(XmlNode CurrentNode)
{
    // do something with CurrentNode here
    if (CurrentNode.HasChildNodes)
    {
        XmlNode node = CurrentNode.FirstChild;
        while (XmlNode!=null)
        {
            DOMDepthFirst(node);
```

```
                node = node.NextSibling;
            }
        }
    // do something else with CurrentNode here
    }
```

广度优先遍历文档树的程序结构如下，注意我们使用了.NET Class Library 提供的 Queue 类来进行队列操作。

```
public void DOMBreadthFirst(XmlNode root)
{
    Queue queue = new Queue();
    queue.Enduque(root);
    XmlNode CurrentNode;
    try
    {
        while (true)
        {
            CurrentNode = (XmlNode) queue.Dequeue();
            // if the queue is empty, control flow will jump to catch
            section
            // do something you like with CurrentNode here
            if (CurrentNode.HasChildNodes)
            {
                // insert each child into the queue
                foreach (XmlNode eachchild in
                CurrentNode.ChildNodes)
                    queue.Enqueue(eachchild);
            }   // end of if
        }   // end of while
    }   // end of try
    // when the queue is empty, Dequeue method throw an
       InvalidOperationException
    catch (InvalidOperationException e)
    {
        Console.WriteLine("Done!");
    }
}
```

可以传递任何 XmlNode 对象作为遍历的入口节点来遍历该节点下的所有子节点。如果是遍历整个文档树，入口就是 XmlDocument 对象：

```
// …
XmlDocument doc = new XmlDocument();
doc.Load("data.xml");
DOMDepthFirst(doc);
DOMBreadthFirst(doc);
```

6.2.3　使用 XPath 表达式来选取节点

XmlNode 类提供了 SelectNodes 和 SelectSingleNode 方法，使用这两个方法，可以在以当前节点为根的子树中选取符合某些条件的节点。SelectSingleNode 方法选取第一个符合条件的节点并返回该节点，而 SelectNodes 选取所有符合条件的节点，并返回包含这些节点的一个 XmlNodeList 对象。指定选取条件是通过传递一个 XPath 表达式来实现的。

XPath 是 W3C 的一个通用的查询语言规范，用来定位 XML 文档中的各个部分。.NET 框架对 XPath 的实现遵循了 W3C XML Path Language（XPath）Version 1.0 规范。在.NET 中，XML 文档对象 XmlDocument 及其派生类都支持使用 XPath 表达式来查询文档中的节

点，.NET 中还实现了一个 XPathDocument 类，使用该类来在进行 XPath 查询和 XSLT 操作中获得更高的性能。可以把 XPathDocument 看作是 DOM 对象为 XSLT/XPath 处理进行的一个优化，但与 W3C XML DOM 类不同的是，XPathDocument 类没有包含进 DOM 要求的所有规则和合法性检验。注意 XPathDocument 并不是 XmlDocument 的派生类。

例如，查询当前节点下的所有子节点的 XPath 表达式是：

```
"*"
```

查询当前节点以下的任何深度下的<authors>标记下的所有<author>元素节点的 XPath 表达式是：

```
"//authors/author"
```

因此查询一个文档的根元素节点下的所有子节点的代码是：

```
XmlDocument doc = new XmlDocument();
doc.Load("data.xml");
XmlNodeList children = doc.DocumentElement.SelectNodes("*");
```

查询文档中任何深度下的<authors>标记下的所有<author>元素节点的代码则是：

```
XmlNodeList allauthors = doc.SelectNodes("//authors/author");
```

详细的 XPath 表达式语法，请参阅第三章对 XPath 的介绍。

6.2.4　处理 XmlDocument 的事件

XmlDocument 对象会触发若干种事件：

表 6-6　XmlDocument 对象的事件

事件	触发条件
NodeChanging	当文档中的某个节点的 Value 属性将要被改变时。可以通过处理该事件来进行额外的检查工作，如果必要，可以抛 出一个异常来中止改变操作并将文档对象恢复到原来的状态。该事件只对那些拥有 Value 属性的节点有用
NodeChanged	当文档中的某个节点的 Value 属性被改变后
NodeInserting	当文档中的某个节点将要被插入到另一个节点中时。可以通过处理该事件来进行额外的检查工作，如果必要，可以抛出一个异常来中止插入操作并将文档对象恢复到原来的状态
NodeInserted	当文档中的某个节点被插入到另一个节点中后
NodeRemoving	当文档中的某个节点将要被从文档中删除时。可以通过处理该事件来进行额外的检查工作，如果必要，可以抛出一个异常来中止删除操作并将文档对象恢复到原来的状态
NodeRemoved	当文档中的某个节点被删除后

这三类事件的类型均是 XmlNodeChangedEventHandler 代表类型。这个代表类型的原型为：

```
public delegate void XmlNodeChangedEventHandler(
   object sender,
   XmlNodeChangedEventArgs e
);
```

可以看到，该原型使用了一个新的事件参数：XmlNodeChangedEventArgs 类。这个类

包括三个属性：NewParent、OldParent 和 Node，它们都是 XmlNode 类型，分别表示被改变、新插入、或被删除的节点原来的父节点和新的父节点，以及该节点本身。如果任何一个父节点是不存在的（例如新插入的节点便不拥有父节点），则该属性为 null 引用。程序员可以使用这个参数对象的这些参数来做许多事情。

很不幸，.NET Framework SDK Beta 2 文档中并没有提供这个 delegate 的其他信息。然而我们仍可以从其他途径获得 XmlNodeChangedEventHandler 代表的信息。在.NET 中探索文档中没有提及的“内幕”的一个很有用的工具是 IL DASM（ildasm.exe，这个工具通常可以在%windir%\Microsoft.NET\Framework\v1.0.xxxx 目录下找到，这里的 xxxx 代表.NET Framework 的版本号）。使用该工具我们可以反汇编 MSIL 文件，并发现 system.xml.dll 文件中包含了和这些事件相关的几个类：XmlNodeChanged、XmlNodeChangedEventHandler、XmlNodeChangedAction。我们可以进一步找出这些类的成员及这些成员的许多信息。

一个完整的处理文档对象的节点插入事件的程序如下：

```
using System;
using System.Xml;
public class EventSample
{
    public void oninsert(object Sender, XmlNodeChangedEventArgs e)
    {
        if (e.OldParent != null)
            Console.WriteLine(e.OldParent.Name);
        Console.WriteLine(e.NewParent.Name);
        Console.WriteLine(e.Node.Name);
    }

    public static void Main()
    {
        EventSample app = new Sample();
        app.Run();
    }

    public void Run()
    {
    XmlDocument doc = new XmlDocument();
        doc.LoadXml("<books>" +
                        "<book>" +
                            "<title>book1</title>" +
                        "</book>" +
                        "<book>" +
                            "<title>book2</title>" +
                        "</book>" +
                    "</books>");
        doc.NodeInserted += new
        XmlNodeChangedEventHandler(this.oninsert);
        XmlElement newnode = doc.CreateElement("book");
        newnode.InnerXml = "<title>newbook</title>";
        doc.DocumentElement.AppendChild(newnode);
    }
}
```

我们会在最后一章看到大量例子，这些例子演示了 DOM 接口在消息交换环境中是如何被应用来处理 XML 消息文档的。

6.3　本章小结

DOM 是 XML 技术中一个非常重要的接口，编者在假设读者已经对 DOM 有所了解的基础上，仅仅介绍了 DOM 最基本的概念，而将本书重点放在了.NET 技术中实现 DOM 接口。特别谈到了几种重要的类：XmlNode 类和 Xml Document 对象。

第 7 章　.NET 对 XML SAX 的模拟

XML 有两种数据访问接口：DOM（Document Object Model）和 SAX（Simple Application for XML）。本章介绍的是其中一种——SAX 接口。在本章的开始你会了解到 SAX 接口的产生和特点，与 DOM 接口相比较 SAX 接口的优势与不足。然后，我们将详细地说明在.NET Framework 中，如何使用 XmlReader 和 XmlWriter 两个类去实现一种可以体现 SAX 接口优势的数据读写方式。这种方式并不是严格意义上的 SAX 接口。在本章的最后，我们将 SAX 和 ASP.NET 结合起来，实现收集网页中的数据信息并将得到的数据存储为 XML 文档。

7.1　SAX 接口

在第 6 章 ASP.NET 实现的 DOM 中，我们介绍了 XML DOM 接口的实现。XML 还有另一种标准接口，称为 SAX 接口。SAX 的全称是 Simple APIs for XML。SAX 接口是比 DOM 接口更底层的接口。SAX 接口在读写 XML 文档的实际应用中体现了另一种思路。这不同于在 DOM 中先根据 XML 文档建 DOM 树，然后按照树形结构访问文档数据的想法，SAX 采用的是一种单向遍历文档的方法。什么是单向遍历，后面我们会讲到。

在某些实际应用中，DOM 接口暴露了一些不足。比如，DOM 是对 XML 进行全面的解析，将完整的 XML DOM 树放入内存中，这样做的好处是可以随机的快速访问数据，但是在 XML 文档复杂、庞大时，实现 DOM 树的速度比较慢，DOM 树也会占用可观的资源。同时，作为一群 XML 技术热衷者讨论的结果，SAX 接口渐渐成型。新的 SAX 接口可以避免 DOM 的这种缺陷，简单是 SAX 的一大特性，甚至 SAX 全称的直译就是 XML 简单应用程序接口。

SAX 是由事件驱动的，当 XML 解析器遇到特定的事件时，会调用不同的函数来处理特定事件。SAX 接口只是调用相应的函数，对于数据的处理是由函数完成的，SAX 没有太多的任务。这也是 SAX 被称为“简单”应用程序接口的原因。SAX 对于数据文档的访问是顺序的，从文档开始到文档结束，不能随机地访问文档中任意数据。这是因为 SAX 没有将所有的文档数据放入内存。

和 DOM 接口相比，SAX 的优势主要体现在：

- 当 XML 文档比较庞大时，使用 DOM 接口会占用大量的内存空间，比如，100K 的 XML 文档，要占用将近 1M 的空间。而 SAX 接口所需的空间要小得多，几乎不需要额外的空间。在处理较大的 XML 文档时，这一点体现了 SAX 相对于 DOM 最大的优势。
- SAX 允许在任何情况下中断解析进程。这样做的好处是，使用 SAX 接口完成像寻找数据并返回找到数据这样的操作时，在找到并返回所需数据之后就可以直接中断进程了。
- SAX 不必解析整个 XML 文档，然后将所需的信息返回。在只需 XML 文档中少量

的数据内容时，SAX 可以直接找到并返回这些数据而不必将多余的数据内容放入内存中。在这种情况下，SAX 节省了大量的系统资源。

- 在创建新的数据结构时，使用 SAX 可以运用高级的对象模型，而使用 DOM 则要用像元素、属性这样的低级对象来完成整个创建工作。比如，一个表示内容是“新闻”的数据结构，需要汇集从 XML 文档或其他资源文档中得到的数据“新闻内容”，使用 DOM 接口可用的只有元素、属性和指令等低级对象，而使用 SAX 可以更高效地创建这样的文档数据结构。
- 在你不能承受 DOM 占用过多的资源的情况下，使用 SAX 会让你满意。比如，在处理大量的复杂 XML 文档，而且文档是在客户端时，服务器端使用 SAX 接口能快速地，更有效地解析这些文档。与 DOM 相比，SAX 处理 XML 数据所需的缓冲更小，内存更小。

SAX 不是 W3C 的正式标准，事实上它与 W3C 没有任何联系，SAX 是 Internet 上一群热衷于 XML 技术的人共同研究出的成果，是由 XML_DEV 邮件列表中的成员提出的。David Megginson 是 SAX 研究开发工作的核心人物。作为一种基于事件的 XML 编程接口，SAX 已被各种 XML 团体广泛认可。

SAX 接口目前多用 Java 实现，C++也可以实现 SAX，只是 C++的分析器比较少。

在.NET beta2 中，还没有 ASP.NET 实现 SAX 接口标准。对于最新的 SAX2 标准，Microsoft 公司使用的是 Microsoft® Visual Basic®和 Microsoft Visual C++®来实现 SAX2 接口。我们在这里不会为你详细地介绍 SAX2 的特性，因为本书的重点是 XML 与 ASP.NET 的结合应用，所以我们将着重介绍在.NET Framework beta2 中，对于 SAX 接口的模拟实现。这种模拟力求体现 SAX 具有的优势，减少 SAX 本身的不足。

在.NET Frameword 中，主要使用 XmlReader 和 XmlWriter 两个类来模拟 SAX 接口的数据访问方式。使用指针来实现顺序的、只能向前的数据访问。比如：

```
XmlTextReader reader = new XmlTextReader ("example.xml");
```

声明了一个访问指针 reader，这个指针是 XmlReader 的派生类 XmlTextReader 的实例，是一个读指针，用来访问名为 example.xml 的数据文档。下面一行代码声明了一个写指针：

```
XmlTextWriter writer = new XmlTextWriter(Console.Out);
```

这个指针是 XmlTextWriter 的实例指针，作用是将数据输出到控制台。

XmlReader 和 XmlWriter 对数据的读写的基本思想是：将 XmlReader 和 XmlWriter 的派生类实例化，得到一个指针实例，然后通过这个指针调用 XmlReader 和 XmlWriter 两个类及这两个类的派生类所具有的公共方法来实现对数据的读写操作，并且根据各种公共成员属性来约束指针的操作。

和 SAX 一样，XmlReader 和 XmlWriter 的指针实例对数据的操作是快速的、只向前的、无缓冲的访问操作。指针是从第一个节点读起，然后依次根据调用的方法和使用的属性移动到后面的节点，当移动到文档中的最后一个节点时，将结束对文档数据的访问。但是严格地说，这两个类对数据的操作和 SAX 接口还是有所区别的，这主要是“推”数据和“拉”数据两种模式的区别，我们在下一节中会详细介绍。

SAX 自身也有不足，主要表现在：

- 不能对 XML 文档进行随机访问，只能对数据进行顺序访问。这是因为 SAX 并不把整个 XML 文档放入内存中。当文档中使用了很多元素之间的引用，比如使用 ID 和 IDREF 属性，使用 SAX 解析文档是十分困难的。
- 对于复杂的搜索，SAX 也是不能胜任的。你必须提前保留任何重要的、以后要用到的数据。像当前节点的祖先属性这样的数据，如果后面可能用到，那么就要在解析时保存起来。
- 目前的浏览器对于 SAX 的支持尚不够完善。

在了解了 SAX 的优势和不足之后，就可以根据你的需要选择适合自己的 XML 数据接口了。下面我们将介绍 SAX 在.NET 中的模拟实现，也可以进一步综合在.NET Framework 中“SAX 接口”的实现方法的优劣来决定是 DOM 还是 SAX 更适合你。

7.2　.NET Framework 中的 SAX

7.2.1　在.NET Framework 中的 SAX 对象

在开发.NET 应用程序时，我们需要对 XML 数据文档进行访问和修改。与 DOM 不同，对数据的访问主要是通过 XmlReader 类的继承类 XmlTextReader、XmlNodeReader 和 XmlValidatingReader 等来实现的。

XmlTextReader 类主要是提供了对于 XML 文档数据的快速访问。XmlNodeReader 提供的是对于 XmlNode 类型，也就是 DOM 节点类型对象的访问。这两个类的实例对象只要求被访问的 XML 文档是格式良好的，对于数据类型的有效性不作要求。

但在实际应用中，是需要检查数据是否和指定的数据类型匹配的。所以，.NET 提供了一个专门用来检查 XML 数据类型有效性的派生类——XmlValidatingReader。实例化这个派生类可以得到一个用来检查数据类型的指针对象。然后通过将指针从文档的开始，按照指定的方法一直移动到文档的结束标记，在移动的过程中完成对文档数据类型的检查。

在.NET 中，通过 XmlWriter 定义了一个写 XML 的接口。XmlWriter 提供了一种生成 XML 流的方法，是根据 W3C 的 XML1.0（第二版）的标准生成 XML 文档的。所以，文档的推荐名字空间是“www.w3.org/TR/REC-xml-names/”。

XmlWriter 只有一个派生类 XmlTextWriter，通过这个派生类的实例指针，我们可以输出合乎规范的 XML 文档。

7.2.2　使用 XmlReader 访问数据

XmlReader 是.NET 中用来访问 XML 数据和文档的重要类对象。XmlReader 既可以访问数据流，也可以访问数据文档。

在 Beta2 的.NET Class Library（类库）中，XmlReader 本身是一个抽象基类，根据访问的数据类型和不同情况下的应用，从 XmlReader 派生出了三个子类：

- XmlTextReader，用来实现对 XML 数据快速的、只向前的访问。实例化 XmlTextReader 得到一个访问指针，通过指针的移动，将指针指向的内容返回给应

用程序。使用 XmlTextReader 的实例指针是访问 XML 数据文档时的常用方法。XmlTextReader 不支持 DTD 和 Schema。

- XmlNodeReader，XML Document Object Model（DOM）将 XML 文档解析为 DOM 树后，XmlNodeReader 类的实例指针提供对 DOM 树中 XmlNode 对象的解析。XmlNodeReader 的实例指针返回访问 XmlNode 对象到的任何类型的节点，其中包括实体引用节点。和 XmlTextReader 类似，XmlNodeReader 不支持根据 DTD 和 Schema 进行数据有效性的检查。但是，XmlNodeReader 可以解析 DTD 中声明的实体引用。
- XmlValidatingReader，XmlValidatingReader 的实例指针根据 DTD、XML Schema Definition（XSD）和 XML-Data Reduced Language（XDR）检查 XML 数据类型的有效性。通过 ValidationType 属性可以决定检查的规则类型，比如是 DTD 还是 Schema，或是根本不进行任何的检查。

除了上述的几个继承类以外，还可以根据不同的需要编写自己的类，或扩展现有的类，实现有特殊要求的 XML 文档数据访问。

XmlReader 提供的数据访问方式是定义一个无缓冲的、只向前的、只读访问的指针来访问数据。XmlReader 只是对数据进行访问和判断数据对象的格式是否良好，并不会检查数据对象的有效性。当数据不满足格式良好的要求时，XmlReader 会抛出异常 XmlExceptions。

在.NET beta2 中，XmlReader 使用的名字空间是“www.w3.org/TR/REC-xml-names”。

通过 XmlReader 类，可以实现忽略对某些数据的访问。这些数据一般是指与当前应用程序无关的数据或此时不需要的数据。在跳过这些无用数据时，XmlReader 使用一种称为“拉”模式的方法。这和 SAX 中的“推”模式有所不同，这也是 XmlReader 和 SAX 的主要区别之一。

“推”模式是解析器每次读到属性、处理指令等内容都向客户程序通报，然后由客户程序来处理这些数据内容。对于不必要的数据，客户程序不予理睬。这样，实现了在访问数据跳过了指定的数据。这其中的关键是，由客户程序来完成数据选择的。

在“拉”模式中，解析器只会通报由客户程序指定的数据内容，对于其他无关的内容，一律不用通报，直接跳过。通报的规则是客户程序指定的，由解析器来应用这些规则。也就是说，是由解析器来完成数据选择的过程。

“推”模式可以建立在“拉”模式之上，换句话说，“推”模式在访问数据时跳过特定数据的过程，可以使用“拉”模式来实现。而“拉”模式不能建立在“推”模式之上。

“推”模式在多输入流时显得相当复杂，而“拉”模式允许客户程序将多输入流集中在一起，从而使处理多数据输入的过程得到简化。

在如何避免复制多余的字符串问题上，“推”模式是将解析器缓存中的数据读到字符串类中，再推进客户程序的缓存。“拉”模式允许客户程序直接为解析器开辟数据缓存，减少了多余的数据复制。

XmlReader 类的方法在继承类中大多得到继承，对于不同的继承类，方法和属性可能会重新定义或重载。比如：

MoveToAttribute 方法可以实现对当前节点的属性节点的遍历。当调用 MoveToAttribute 方法后，得到的将是节点的属性的名称、名字空间 URI 和名字空间前缀。属性不一定要 DTD 或 Schema 来指定，属性可以有默认值。对于从 DTD 或 Schema 中得到的值，MoveToAttribute 方法视为 XML 流对属性的赋值。

属性 IsDefaul 表示属性值的赋值是否为默认值。如果当前节点是属性节点并且属性没有在 XML 流中指定，而是使用了在 DTD 或 Schema 中声明的默认值，那么属性 IsDefault 的返回值为“真”。对于 XmlTextReader，IsDefault 返回“假”，因为没有可用的 DTD 信息。对于 XmlValidatingReader，当属性值在 XML 流中明确指出时，返回“假”；当属性值是由 DTD 中声明的默认值得到的时候，返回“真”。对于 XmlNodeReader 而言，当 XmlDocument 有 DTD 定义时，IsDefault 返回“真”，否则返回“假”。

前面讲到了 XmlReader 对于非关键的信息一律跳过，在实现层次上，是可以调用 MoveToContent 方法。还可以使用另一种直接的 Skip 方法，略过当前节点的子节点。

MoveToContent 方法先判断当前节点是否为文本节点，如果不是，就略过当前节点到下一个节点，直至文档结束。略过的节点类型有：XML 声明、指令节点、根节点、注释节点、属性节点、空白。在不需要将访问指针置于下一个节点或不需要再访问前，先检验节点类型以保证节点为文本节点时，或应用程序只需要文本时，MoveToContent 方法更有效。

如果应用程序定位于属性节点，MoveToContent 方法会将当前节点变为属性的元素节点（文本节点）。如果已经位于文本节点，则只返回属性值。

对于 XmlNodeType.Element 类型的节点，Skip 方法将直接跳到结束标记，略过所有内容。

关于 XmlReader 的方法和属性我们再介绍一个类——XmlNameTable。

在解析数据或比较 XML 文档时，使用 XmlNameTable 类，XmlReader 可以实现使用指针比较元素名和属性名，不需进行字符串比较。使用 NameTable 的情况下，当需要多次返回相同的名字时，只用将表中的名字字符串重复送回应用程序。在需要使用和比较元素名和属性名的时候，可以使用 NameTable 访问 XmlReader 的继承类。

XmlNameTable 类可以被应用程序调用和实例化。XmlNameTable 类是抽象类，NameTable 是类的构造。NameTable 中存放了元素节点名和属性名，包括名字空间的 URI 和前缀，这些内容都是独立的，可以看成表格中的单元。如果应用程序需要大量的名字比较操作时，可以创建 NameTable 来简化比较过程。使用者可以通过访问指针的 NameTable 属性来操作 NameTable 这个表。

应用程序可以使用 Add 方法来向表中添加名字，使用 Get 方法来获得字符串对象。Add 方法的重载列表如表 7-1。

表 7.1　add 方法重载表

声明形式	说明
public abstract string Add(string)	将参数字符串添加为新名字原子
public abstract string Add(char[], int, int)	第一个整数参数是字符串内的偏移。第二个整数参数是长度。将字符串参数中偏移量开始的指定长度的部分添加为新名字。如果长度为空，返回 String.Empty。当偏移量大于数组长度且小于 0 时，或长度大于数组长度与偏移量的差且小于 0 时，抛出异常

NameTable 类还有 Get 方法与 Add 方法类似。可以得到包含指定字符串的字符串对象和包含指定数组中的特定偏移后的指定长度的字符串对象。如果在取指定数组中的片段时越界，会抛出类型为 IndexOutOfRangeException 的异常。

下面，我们来逐一介绍 XmlReader 的派生类。

1. XmlTextReader

如何在处理 XML 数据和文档时访问 XML 数据呢？XmlReader 的继承类 XmlTextReader 提供了一种方法。首先实例化一个 XmlTextReader 对象来作为访问数据的指针，然后通过移动这个指针到不同的位置来访问数据。看一个例子：

程序清单 7-1：example-7-1.cs

```
using System;
using System.Xml;

class example
{
    public static void Main()
    {
        XmlTextReader reader = new XmlTextReader ("data.xml");

        while ( reader.Read())
        {
            switch (reader.NodeType)
            {
                case XmlNodeType.DocumentType:
                    // The node is a DocumentType
                Console.WriteLine (reader.NodeType + "<" +
                reader.Name
                    + ">" + reader.Value);
                break;

                case XmlNodeType.Element:
                    // The node is an Element
                Console.Write ("<" + reader.Name);
                while (reader.MoveToNextAttribute())        //
                Read attributes
                    Console.Write (" " + reader.Name + "='" +
                    reader.Value
                            + "'");
                Console.Write (">");
                reader.MoveToContent ();
```

```
                        // Read the content of node.
                    Console.WriteLine (reader.ReadInnerXml ());
                    break;
                }
            }
        }
    }
```

程序清单 7-2：data.xml

```
<? xmlversion =1.0?>
<! DOCTYPE person [
    <! ELEMENT person ANY>
    <! ELEMENT name (#PCDATA)>
    <! ELEMENT position (#PCDATA)>
    <! ATTLIST person id CDATA #REQUIRED>
    <! ATTLIST position department CDATA #REQUIRED>
]>
<! -- This is a personal record -->
<person id="001">
    <name>John</name>
    <position department="market">clerk</position>
</person>
```

输出的结果为：

```
DocumentType<person>
        <!ELEMENT person ANY>
        <!ELEMENT name (#PCDATA)>
        <!ELEMENT position (#PCDATA)>
        <!ATTLIST person id CDATA #REQUIRED>
        <!ATTLIST position department CDATA #REQUIRED>

<person id='001'>
        <name>John</name>
        <position department="market">clerk</position>
```

example-9-1.cs 代码中的第一部分是名字空间的说明。XmlReader 使用的名字空间是 System.Xml。

然后，我们定义了一个访问指针“reader”，换句话说，就是实例化了一个 XmlTextReader 的对象。“XmlTextReader reader = new XmlTextReader ("data.xml");”中的“data.xml”是外部的 XML 文档名。

接下来，我们使用 Read()方法来判断是否能读到下一个节点，当 reader.Read()方法得到“真”，将指针移动到节点“person”。属性“NodeType”用来判断节点类型，当节点类型是文档节点（DocumentType）时，输出节点的名称和值。

如果是元素节点的话，将调用 while 循环将属性及属性值（用单引号括起）输出。其中，MoveToAttribute 方法是将指针移到属性节点。如果需要将指针再移动回元素节点，需调用 MoveToElement 方法。

reader.MoveToContent ()方法将指针移动到“person”节点的第一个子节点“name”节点，然后调用 reader.ReadInnerXml ()方法输出“person”节点的内容。其中调用 ReadInnerXml 方法得到从节点的始标记到结束标记间包括标记本身的内容。此时，指针已经移动到了“</person>”，因为指针是单向向前的，所有下一步将跳出循环，结束程序。

注意：属性“department”的值输出时是用双引号括起的－department=“market”，这是第一次循环输出的结果，是“person”节点的内容。指针并不会移动到“position”节点去处理“department”属性的。所以属性值是不会用单引号括起的。

如果注释掉下面两句：

```
reader.MoveToContent ();            // Read the content of node.
Console.WriteLine (reader.ReadInnerXml ());
```

“department”属性值的输出结果就会用单引号括起来了。

上面提到的公有方法和属性是从 XmlReader 类继承得到的，有些是经过重定义得到的。下面来看看 XmlTextReader 特有的方法：

（1）ReadChars（char[ ]，int，int）

ReadChars 是用来从大量的数据流中读取的连续字符放入指定位置开始，指定长度的缓冲区（char[ ]）中。在读取的过程中，是将所有的字符，包括标记都放入缓冲区中，不会对读到的数据进行解析。

比如读取 XML 数据文档：data.xml，当调用 MoveToComent 方法之后，将指针移至元素节点“name”时，使用下面的方法：

buffer = new char[20];

reader.ReadChars(buffer, 0, 20)

将得到从开始标记算起的 20 个字符，这些字符包括空格和标记。并且指针移动到节点的结束位置。如果读入的字符数小于缓冲区大小时，指针将停在节点的结束标记之后，结束对当前节点的访问。

ReadChars 方法只能用于元素节点，否则，返回的将是“0”。

ReadChars 方法是将数据读为字符，再放进缓冲区的。对于大量的非文本数据，像图形等二进制或 base64 编码的数据文档，XmlTextReader 提供了两种方法来读取这种非文本格式的数据。

（2）ReadHex（byte[], int, int）和 ReadBase64(byte[ ], int, int）

这两个方法和 ReadChars 方法很相似，都是从大量的数据流中读取连续的信息放入缓冲区中。后两个参数的作用也一样，只是记得计算的单位是字节（byte）不是字符（char）。和 ReadChars 方法不同的是 ReadHex 和 ReadBase64 两种方法开辟的缓冲区是字节数组（byte array）。

ReadHex 方法是将 base64 编码的数据解码读为二进制数据放入缓冲区，ReadBase64 方法则相反，放入缓冲区的是 base64 编码的数据。

XmlTextReader 还有一些从 Object 对象继承到的方法。我们来看看 Equals 和 ToString 方法。Equals 方法根据不同的情况重新定义，如何定义方法不是我们关心的，我们只是用这些方法来编写代码，实现需要的功能。我们可以合法的借用其他人的成果来简化工作，这也是一种快速开发的模式。

Equals 可以比较指定对象与当前对象，返回一个布尔值。比较的方法是使用虚函数来实现的。Equals 还可以比较两个对象是否相同。

程序清单 7-3：example-7-2.cs

```
using System;

namespace example9_1
{
    class example9_1
    {
        public static void Main()
        {
            string Str1="England";
            string Str2="American";
            Console.WriteLine("\"{0}\" equals \"{1}\" ?   {2}.",
            Str1, Str2, Str1.Equals(Str2));
            Console.WriteLine("\"{0}\" equals \"{1}\"?
            {2}.",Str1,Str2,Object.Equals(Str1,
                    Str2));

            string Str3=null;
            Console.WriteLine("\"{0}\" equals \"{1}\"?
            {2}.",Str1,Str3,Object.Equals(Str1,
                    Str3));
            Console.WriteLine(Str1.ToString());
        }
    }
}
```

程序的输出见图 7.1。

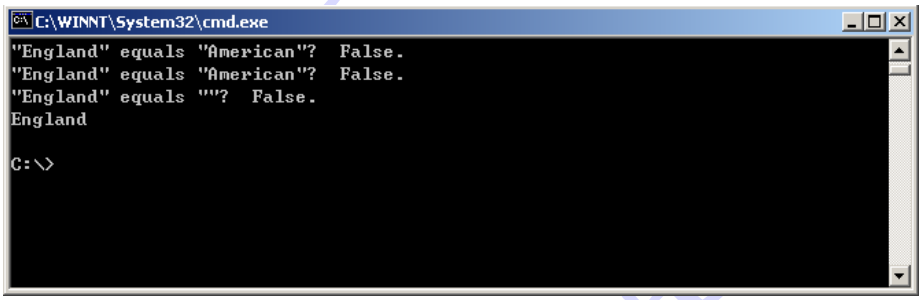


图 7.1　输出示例

在示例中，我们判断了字符串 1“England”和字符串 2，3 是否相同。Equals 的两种比较方式，见 Str1 和 Str2 的比较。最后示例演示了公有方法 ToString。输出了 Str1 的字符串“England”。

这些方法绝大多数的类都可以从 Object 类继承到。并且经过重定义，在不同的类中，这些方法的作用也是基本相同的，所以我们在以后介绍其他类时将不再提及这些方法。

从 XmlReader 类继承来的方法和属性，除了前面介绍过的 Read 方法、NodeType 属性等以外，XmlTextReader 还有其他的公有方法和属性。

比如 WhitespaceHandling 属性指定了怎样处理空白，可能的取值包括 All、None、Significant 等。

还有用来报告出错位置的属性 LineNumber 和 LinePostion，使用这两个属性可以得到当前位置的行数和位置数。要注意，在计算位置时，标记中的符号是有效的。比如：指定的行内容为“</node>”，LinePosition = 4 时，得到“o”，LinePosition = 2 时，得到“/”。

XmlTextReader 有一些用来判断情形，并返回布尔值结果的公共成员属性。包括：

- EOF 属性判断是否到达文档的结尾
- IsEmptyElement 属性判断是否为空元素
- IsDefault 属性判断属性节点值是否为 DTD 中定义的默认值
- Namespaces 属性判断是否支持名字空间等

XmlTextReader 类从 XmlReader 类继承到的方法，XmlReader 的其它派生类也可以继承到，以后，我们就不再说明这些属性和方法了。这些方法还可以参考本节最后的 XmlReader 公有属性表和方法表。XmlTextReader 继承的公共成员方法主要有：

- GetAttribute 得到属性值
- GetRemainder 得到 XML 缓冲区中的剩余部分
- MoveToAttribute 方法移动到指定的属性
- MoveToElement 方法移动到包括当前属性节点的元素节点
- Read 方法读取下一个流中的节点
- ReadAttributeValue 方法返回“真”如果有属性节点返回，如果方法返回“假”，说明访问指针没有位于属性节点或属性节点都已读过了。一般是和 MoveToAttribute 配合使用
- ReadString 方法将元素节点或文本节点的内容作为字符串读取
- ResolveEntity 方法分析实体引用

XmlTextReader 只要求文档的格式良好，对于数据是否有效不作判断。XmlTextReader 是强制执行 XML 格式良好标准的，并且使用 DTD 来检查文档的格式是否良好。

需要进行数据的有效性确认时，使用 XmlValidatingReader 类的方法。

2. XmlValidatingReader

在开发.NET 应用程序时，我们使用 XmlTextReader 类来解析 XML 文档。尽管.NET Framework 提供了完全的 XML DOM 支持和 SAX 类，使用 XML 的读写指针体现了在代码执行速度、内存空间、效率和 XML 的易用性之间的平衡。但是，XmlTextReader 并不进行数据的有效性检查。有效性检查的工作是由 XmlValidatingReader 类来完成的。

因为 XmlValidatingReader 是用指针对象来判断数据类型的。不能先用这个指针判断数据格式是否有效，如果格式有效再使用这个相同的指针来处理数据。需要使用不同的指针判断有效性和访问数据。

在.NET Framework 中为了提高解析 XML 数据的速度，访问数据的指针是不对数据格式有效性进行检查的。但在实际应用中，根据 DTD、XDR 或 Schema 来检验数据格式有效性是很有必要的。如果省略连续的、重复的数据格式检查，会优化应用程序的执行速度。事实上，这些检查也是可以简化的，因为很多情况下，已检查过格式有效的数据是被重复检查的。这种省去对格式有效的数据的检查是常用的方法，这很像发布 Windows 应用程序时，对调试信息的省略。但是，在开发 XML 应用程序时，为了避免异常的发生，对每段代码进行有效性检查是很有必要的。

在.NET Framework 中，XmlReader 的继承类 XmlValidatingReader 负责对数据格式进行检查。和 XmlReader 不同，XmlValidatingReader 允许指定要检查的类型和如何处理出错信息。

要使用 XmlValidatingReader 的方法检查数据格式是否有效，需先定义一个指针对象，这和前两个派生类很相似。在声明之后，再调用 XmlValidatingReader 的 Reader 方法，将指针跳到要检查的节点，判断其数据格式是否和类型声明一致。声明步骤如下：

```
XmlTextReader reader = new XmlTextReader(data.xml);
XmlValidatingReader vreader = new XmlValidatingReader(reader);
```

XmlValidatingReader 是不能直接根据文档创建指针对象的。要先定义 XmlTextReader 的对象指针“reader”，然后在声明 XmlValidatingReader 的对象“vreader”。

在创建了指针实例后，可以像操作其他指针一样操作 XmlValidatingReader 的指针实例。下面是一个操作示范：

```
vreader.ValidationType = ValidationType.Auto;
vreader.ValidationEventHandler += new ValidationEventHandler(this.ShowError);
```

第一行说明对象自动寻找用来判断的规则，并应用这种规则。这种判断的规则一共有四种：Auto、XDR、Schema 和 DTD。如果不需要对数据进行类型检验，可以有第五种选择——None。

如果 ValidationEventHangler 属性值为 Auto，则按照以下的顺序确定判断的规则。首先，检查 DTD 定义。指针将寻找 DOCTYPE 类型的声明，并根据找到的所有 DTD 判断格式是否有效。如果找不到 DTD 的声明，指针将会寻找 XSD 的 SchemaLocation 属性。如果没有 XSD 声明，将检查 XDR 的 x-Schema 属性，来定位源文件。最后，指针将检查是否有内联的 Schema 定义(由<Schema>标记指定)。实际上，如果指定使用的规则可以省去多余的规则检查步骤，直接应用指定的规则。

第二行声明了出错时的处理方法。当指针从一个节点移动到下一个节点，同时作出判断。这种对单个节点的判断，我们称为“读”当前节点。如果遇到错误异常的话，可以选择停止进程或继续进程。如果通过为 ValidationEventHandler 设定属性值来指定处理出错事件的方式不能处理当前的错误，进程将停止。如果没有指定处理方式，将先调用 XmlException 来处理出现的错误异常。异常停止的进程可以通过代码手动恢复执行。

如果指定了 ValidationEventHandler，XmlValidatingReader 类将使用它处理所有的异常。否则，进程不会停止，除非到达文档的结尾。

前面简要的说明了使用 XmlValidatingReader 类指针对象的基本过程。我们来看一个例子,通过对 DTD 和 Schema 的检查类判断能否检查文档的数据类型：

程序清单 7-4：example-7-3.cs

```
using System;
using System.IO;
using System.Xml;
using System.Xml.Schema;

namespace example_7_3
{
    class Class1
    {
```

```
private const string file = "example-7-3.xml";
private const string schema = "example-7-3.xsd";
private XmlTextReader txtreader = null;
private XmlValidatingReader reader = null;
private Boolean success = true;

public Class1 ()
{
    Validate(file, ValidationType.Schema);
    Validate(file, ValidationType.DTD);
}

static void Main()
{
    Class1 Validation = new Class1();
}

private void Validate(String filename, ValidationType vt)
{

    try
    {
        //Implement the readers.  Set the ValidationType.
        txtreader = new XmlTextReader(filename);
        reader = new XmlValidatingReader(txtreader);
        reader.ValidationType = vt;

        //If the reader is set to validate, set the event
          handler.

        if (vt==ValidationType.Schema)
        {
            Console.WriteLine("\nParsing XML file " +
                    filename.ToString());
        }
        else
        {
            Console.WriteLine("\nValidating XML file " +
                    filename.ToString());
            success = false;

            //Set the validation event handler.
            reader.ValidationEventHandler +=
             new ValidationEventHandler
            (ValidationCallBack);
        }
            Console.WriteLine ("Validation finished.
            Validation {0}",
                     (success==true ? "successful" :
                      "failed"));

        while (reader.Read())
        {
            if(success)
            {
                //output the nodes' names and values.
                Console.Write(reader.Name);
                Console.Write(reader.Value);
            }
```

```
                }
            }

            finally         //Close the reader.
            {
                if (reader != null)
                reader.Close();
            }

        }

        public void ValidationCallBack (object sender,
                ValidationEventArgs args)
        {
            Console.Write("\r\nValidation error: " + args.Message);
        }
    }
}
```

在示例中，先声明了访问指针“reader”和检查指针“vreader”，然后通过布尔变量“success”标记是否可应用文档解析规则。如果不能，则抛出异常，由 ValidationCallBack 函数处理异常，输出出错消息。最后调用 close 方法关闭指针。

如果能按照指定的规则解析，就调用“while”循环输出节点名和节点值。并显示“successful”。由于没有指定 DTD，所以在使用 DTD 检查数据类型时会出错。

下面是相关的 XML 文档和 XSD 文档。如果你自己编写了合适的 DTD 文档，就不会再出错了。

程序清单 7-5：example-7-3.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<records xmlns:xsi="http://www.w3.org/2000/10/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation=" example-9-2.xsd">
    <person id="024">
        <name>Smith</name>
        <phone>12345678</phone>
        <e-mail>smith@Acompany.com</e-mail>
    </person>
    <person id="541">
        <name>Rose</name>
        <phone>23456789</phone>
        <e-mail>rose@Acompany.com</e-mail>
    </person>
    <person id="217">
        <name>Mike</name>
        <phone>87654321</phone>
        <e-mail>mike@Bcompany.com</e-mail>
    </person>
</records>
```

程序清单 7-6：example-7-3.xsd

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2000/10/XMLSchema"
        elementFormDefault="qualified">
    <xsd:element name="records" type="recordType"/>
    <xsd:complexType name="recordType">
        <xsd:sequence maxOccurs="unbounded">
            <xsd:element name="person" type="personType"/>
```

```
            </xsd:sequence>
        </xsd:complexType>
        <xsd:complexType name="personType">
            <xsd:sequence>
                <xsd:element name="name" type="xsd:string"/>
                <xsd:element name="phone" type="xsd:decimal"/>
                <xsd:element name="e-mail" type="xsd:string"/>
            </xsd:sequence>
            <xsd:attribute name="id" type="xsd:decimal"/>
        </xsd:complexType>
    </xsd:schema>
```

如果有兴趣，你可以将这两个文件和编译 example-9-2.cs 得到的可执行文件拷贝到相同的文件夹中，运行可执行文件，观察输出的结果。

在指定好是应用 DTD 还是 Schema 或是 XDR 来检查 XML 文档之后，就要对文档进行类型检查。我们使用 vreader 的 read 方法读每个节点，如果全都通过的话，就不改变“success”的值，输出“successful”。

```
while (vreader.Read()){}
Console.WriteLine ("Validation finished. Validation {0}",
(success==true ? "successful" : "failed"));
```

一旦不能通过检查，就抛出异常。用“catch”来处理异常。捕捉 XmlException 和 Exception 两类异常，输出出错信息。

```
catch(XmlException e)
        {
            Console.WriteLine ("Validation failed.");
            Console.WriteLine ("XmlException:"+ e.ToString());
        }

catch(Exception e)
        {
            Console.WriteLine ("Validation failed.");
            Console.WriteLine ("Exception"+ e.ToString());
        }
```

并且在调用 ValidationCallBack 函数时，改变了 success 的值，使得输出“failed”。

```
public void ValidationCallBack (object sender, ValidationEventArgs args)
        {
            success = false;
            Console.Write("\r\nValidation error: " + args.Message);
        }
```

需要说明的是，如果使用 Schema 来检验文档的数据类型，就要求先声明一个 XmlSchemaCollection 集合类，然后使用 add 方法来添加要用来判断的 Schema 文档。

```
XmlSchemaCollection vreaderCollection = new XmlSchemaCollection();
vreaderCollection.Add ( "example-9-2.xsd", new XmlTextReader(reader));
vreader.Schemas .Add (vreaderCollection);
```

3. XmlNodeReader

XmlReader 类使用另一个派生类 XmlNodeReader 来读取 XML DOM 树的节点 Xml Node 类对象。

XmlNodeReader 可以访问、分析和 XML DOM 树，返回 DOM 树中的节点，包括实体引用节点。XmlNodeReader 可以分析 DTD 中定义过的实体。同 XmlTextReader 一样，需要调用 XmlValidatingReader 进行有效性检查。

和 XmlReader 的其他派生类一样，XmlNodeReader 也是通过定义一个访问指针来完成对节点的访问和分析。对于在文档中的外部实体引用同样可以通过指针访问加以解析。

一般而言，XmlNodeReader 多用于 XmlDocument 类的访问。定义一个指针对 XmlDocument 的对象实例加以访问和解析。定义的形式：

```
XmlDocument exampledoc = new XmlDocument();
exampledoc.Load("data.xml");
XmlNodeReader nreader = new XmlNodeReader (exampledoc);
while (nreader.Read()) {
    //以 XML 数据流的形式读取 XmlDocument 对象的内容
}
```

来看一个完整的例子。程序先按照上面说的形式创建了一个指针，然后通过指针访问 XML 数据文档中的名为“position”的节点，调用循环输出的节点名和节点值。在代码中，使用了对于 XmlDocument 对象的简单操作，使用 SelectSingleNode 方法和 Xpath 表达式选取了指定的节点“position”。

程序清单 7-7：example-7-4.cs

```
using System;
using System.Xml;

namespace NodeReader
{
    class Class1
    {
        public static void Main()
        {
            XmlNodeReader nreader = null;
            try
            {
                XmlDocument exampledoc = new XmlDocument();
                exampledoc.Load("data.xml");
                XmlNode name = exampledoc.SelectSingleNode
                ("/person/position");

                if(name!= null)
                {
                    nreader = new XmlNodeReader(name);
                    while(nreader.Read())
                    {
                        Console.Write (nreader.Value);
                        Console.Write (nreader.Name+"  ");
                    }
                }
            }
            finally
            {
                if(nreader!=null)
                    nreader.Close();
            }
        }
    }
}
```

注意：为了节约篇幅，我们使用了 example-9-1 中的 XML 数据文档 data.xml。你可以在前面的介绍中，找到 data.xml 的源代码。

对 DOM 树节点的操作，还可以参考本书的第 6 章 ASP.NET 实现的 XML。

最后我们给出 XmlReader 类的成员列表，可以在编写代码时灵活地运用这些属性和方法。XmlReader 的派生类也同样包括了这些公有成员，有些成员是通过重定义及重载获得的。每个派生类各自扩充的成员前面都介绍过了。

表 7.2　XmlReader 公有属性表

名称	类型	说明
AttributeCount	Int	计算节点的属性数
BaseURI	String	得到节点的 URI
CanResolveEntity	Bool	判断指针指向的实体能否被解析
Depth	Int	得到节点的深度
EOF	Bool	判断是否到达文档的结尾
HasAttributes	Bool	判断节点是否有属性节点
HasValue	Bool	判断节点是否有值
IsDefault	Bool	判断节点的属性值是否是由 DTD 或 schema 中定义的默认值得到的
IsEmptyElement	Bool	判断元素是否为空
Item	String	得到属性的值，在 C#中这个属性充当了 XmlReader 类的索引
LocalName	String	得到元素的本地名
Name	String	得到元素名的全称，包括名字空间前缀
NamespaceURI	String	得到名字空间的 URI
NameTable	nametable	详见关于 XmlNameTable 类的说明
NodeType	string	得到节点的类型
Prefix	string	得到名字空间的前缀
QuoteChar	char	得到用引号括起来的属性值
ReadState	enum	得到指针的状态
Value	string	得到节点的值
XmlLang	string	得到 xml:lang 属性的值
XmlSpace	preserve/none/default	得到 xml:space 属性的值

下面是 XmlReader 的公共方法，其中由 Object 类继承到的方法就省略了。

表 7-3　XmlReader 公有方法表

名称	说明
Close	将指针变为关闭状态
GetAttributes	得到属性值
IsStartElement	判断是否为开始元素
LookupNamespace	解析名字空间前缀

(续表)

名称	说明
MpveToAttribute	将指针移动到属性节点
MoveToContent	将指针移动到节点内容的开始
MoveToElement	将指针移动到元素节点
MoveToFirstAttribute	将指针移动到第一个属性节点
MoeToNextAttribute	将指针移动到下一个属性节点
Read	读取下一个节点
ReadAttribueValue	读取解析属性值
ReadElementString	以字符串形式读取简单元素
ReadEndElement	检查是否为最后的元素
ReadInnerXml	读取元素的标记（包括标记）之间的所有内容
ReadOuterXml	读取元素标记间（含标记）的所有内容、节点本身和子节点
ReadStartElement	检查是否为开始元素
ReadString	将节点内容作为字符串读取
ResolveEntity	解析实体引用中的实体
Skip	跳过当前节点

4. XmlWriter

前面介绍了如何实现对 XML 文档的数据访问。这是一个“读”的过程，与之相应的就是“写”的过程。在开发 ASP.NET 的应用程序时，如何写出需要的 XML 文档同样是至关重要的。应用程序自动书写 XML 文档的过程，实际上也就是动态生成 XML 数据文档的过程。与 XmlReader 类似，.NET 平台同样提供了一个 XmlWriter 类来实现动态输出 XML 文档。通过操作这个类的派生类，我们可以实现输出复杂的 XML 数据文档。

XmlWriter 和 XmlReader 一样，也是一个抽象基类，也是通过派生类来实现“写”的功能。XmlWriter 提供只向前的、只读的、无缓冲的方法来生成 XML 流。

XmlWriter 的派生类只有 XmlTextWriter，下面我们将结合实例来介绍 XmlTextWriter 的方法和属性。如果你有较好的 XML 语法知识，那么在下面的学习过程中，你会发现操作 XmlTextWriter 输出正确的 XML 文档是很容易的。

XmlTextWriter 可以将 XML 数据写入 XML 数据文档、控制台、数据流和其他的输出格式。在使用 XmlTextWriter 的方法输出 XML 时，为了保证格式良好就需要作额外的工作。比如：WriteAttributeString 方法需要避免依赖于所找到的属性的值。WriteSrting 有时需要避开特定的字符，用“& < &gt”和数字字符代替那些特定的字符。Close 方法要检查 XML 文档是否有效，如果不能通过检查的话，将抛出“InvalidOperationException”异常。

为了保证输出的是格式良好的 XML 文档，XmlTextWriter 要做一些额外的工作：

- 保证 XML 元素是出现在正确的位置上的。比如：属性不能写在元素外面；CDATA 节不能出现在属性中；不能有多个根元素等等。还要保证 XML 声明出现在最前，DOCTYPE 节点出现在根结点之前等。
- 保证属性 xml:space 和 xml:lang 的格式和属性值是正确的，这个值必须是 XML1.0 标准（第二版）中允许的。
- 当字符串作为参数时检查其是否符合 W3C 的标准。

究竟怎样输出真正的 XML 数据呢？

先来看看如何声明一个 XmlTextWriter 的实例，稍后我们将用这个实例指针输出各种格式和类型的 XML 数据。

```
XmlTextWriter writer = new XmlTextWriter(Console.Out);
```

这行代码先声明了 XmlTextWriter 的指针实例“writer”，然后指定这个对象将结果输出到控制台。如果要将结果输出到 XML 流中，需写成：

```
XmlTextWriter writer = new XmlTextWriter(streamname, encoding);
```

其中，“streamname”指定了要写入 XML 的流的流名。encoding 规定了编码方式，默认值是 utf-8 编码。类似的，要将 XML 写入名为“file”的文档，只需将等式右边改写为“new XmlTextWriter(file, encoding)”。

在声明了对象之后就可以使用 XmlTextWriter 的方法和属性来输出 XML 数据了。比如：

```
//writer.Formatting = Formatting.Indented;
 writer.WriteStartElement("media");
 writer.WriteStartElement("video");
writer.WriteElementString("rm", "welcome.rm");
 writer.WriteElementString("mov", "hello.mov");
writer.WriteEndElement();
 writer.WriteEndElement();
```

上面的代码将输出以下结果,我们注释掉了第一行关于格式的说明，使用 Formatting 的默认值 null 得到的输出的结果。

```
<media><video><rm>welcome.rm</rm><mov>hello.mov</mov></video></m
edia>
```

这种样式的输出显然难以阅读，我们可以通过改变属性 Formatting 的值来得到下面的较为清晰的 XML 输出。于是，我们在代码中加上被注释掉的内容，得到：

```
<media>
    <video>
        <rm>welcome.rm</rm>
        <mov>hello.mov</mov>
    </video>
</media>
```

你可以根据不同的表达需要选择合适的 Formatting 属性值得到易于理解的 XML 输出。

事实上，通过方法名的字面意思你可能就大概了解了元素及元素的嵌套的输出方法。输出元素时，要求两个方法配对出现，这两个方法就是“WriterStartElement”和“WriteEndElement”。前一个方法标记了元素的开始标记的位置，后一个方法则说明了元素的结束标记的位置。其中，WriteStartElement 方法中的参数要求是字符串，因为参数指定的是元素的本地名，如果 WriterStartElement 方法有两个字符串参数的话，前一个参数指定了本地名，后一个参数指定的是名字空间的 URI。如果是三个字符串参数，那么含义依

次为名字空间前缀，本地名，名字空间的 URI。而 WriteEndElement 方法就简单多了，是没有参数的方法，只是匹配前面的 WriteEndElement 方法。要注意的是，如果元素的内容为空，则结束标记将简写为“/>”，即“<标记名/>”。比如：

```
writer.WriteStartElement ("acom", "video", "http://www.Acompany.com");
……
writer.WriterEndElement ();
```

得到的输出是：

```
<acom:video xmlns="http://www.Acompany.com">
……
</acom:video>
```

XmlTextWriter 还有一个方法 WriteFullEndElement，是用来输出完整的结束标记的。因为在某些特殊情况下，要求有完整的结束标记，这时就不能简化结束标记了。比如，浏览器一般都期望 HTML 文档中的脚本语言部分以“</script>”结束。

在前面的代码中，我们还用到了 WriteElementString 方法输出元素。这个方法的参数可以是两个字符串，比如示例中的调用。前一个参数是标记本地名，后一个参数是标记间的内容。WriteElementString 还可以有三个参数，参数的含义依次为标记本地名，名字空间的 URI 和标记间的内容。来看个例子：

```
writer.WriteElementString ("rm", "http://www.Acompany.com", "welcome.rm");
```

这行代码得到的输出为：

```
<rm xmlns="www.Acompany.com">welcome.rm</rm>
```

看到这你也许会问，如果一个元素既有子元素，也有文本内容该如何输出这个元素呢？我们可以使用 WriteString 方法，在 WriterStartElement 和 WriteEndElement 方法之间输出字符串。比如：

```
writer.WriteStartElement("mov");
writer.WriteElemenString("name", "hello.mov");
writer.WriteString("This is a example.");
writer.WriteEndElement();
```

输出为：

```
<mov>
    <name>hello.mov</name>
    This is a example.
</mov>
```

现在，我们来看 XmlTextWriter 是如何输出元素的属性的。

XmlTextWriter 提供了 WriteStartAttribute、WriteEndAttribute、WriteAttributes 和 WriteAttributeString 等方法。其中，有些方法是继承自 XmlReader 的，我们将不加区分的介绍这些属性和方法。

先来看看使用 WriteStartAttribute、WriteEndAttribute 方法的例子。这两个方法和 WriterStartElement、WriteEndElement 方法类似。

```
writer.WriteStartElement("video");
writer.WriteStartAttribute("genera", "http://www.Acompany.com");
writer.WriteString("example");
writer.WriteEndAttribute();
writer.WriteStartElement("rm");
writer.WriteString("welcome.rm");
writer.WriteEndElement();
```

```
        writer.WriteEndElement();
```

输出的结果为：

```
        <video genera="example" xmlns:genera="http://www.Acompany.com">
            <rm>welcome.rm</rm>
        </video>
```

其由输出的结果可以看出 WriteStartAttribute 方法的参数的含义：属性名和属性名的名字空间 URI。类似的，WriteStartAttribute 方法也可以有三个参数，这些参数的说明依次为：名字空间前缀，本地名，名字空间 URI。

方法 WriteAttributeString 也可以输出属性的取值。我们先使用 WriteAttributeString 方法得到和上面的例子中属性“genera”相同的输出结果：

```
        writer.WriteAttributeString("genera", "http://www.Acompany.com", "example");
```

WriteAttributeString 方法的参数还可以只有两个参数字符串，含义依次为本地名，属性值。如果为属性指定了名字空间而且还使用了名字空间前缀，那么 WriteAttributeString 方法的参数将达到四个，依次为名字空间前缀，本地名，名字空间 URI 和属性值。

输出属性的方法还有 WriteAttributes，这个方法是输出 XmlTextReader 的实例所指向的节点的全部属性。其参数有两个：前一个是 XmlTextReader 的实例名，后一个是布尔值参数，为“真”时输出默认属性，否则不输出默认属性。我们来看一个比较完整的函数：

```
        public void ShowAttributes()
        {
        XmlTextReader reader = new XmlTextReader("data.xml");
        XmlTextWriter writer = new XmlTextWriter(Console.Out);
        writer.Formatting = Formatting.Indented;

        while (reader.Read())
            {
              if (reader.NodeType == XmlNodeType.Element)
              {
                writer.WriteStartElement(reader.Name.);
                writer.WriteAttributes(reader);
                if (reader.IsEmptyElement) writer.WriteEndElement();
              }
              else if (reader.NodeType == XmlNodeType.EndElement)
              {
                writer.WriteEndElement();
              }
            }
            writer.Close();
            reader.Close();
        }
```

函数首先定义了一个读指针 reader，然后调用循环读取每一个节点。当节点的类型是元素节点时，就使用 WriteAttributes 方法输出 reader 指向的节点的属性。如果 reader 指向的是空节点，就直接结束对元素的输出。

当判断到当前节点的类型是元素的结束节点时，结束对元素的输出。最后，将两个指针设为关闭状态。

XmlTextWriter 对于名字空间的输出是很完整的，可以输出复杂而精确名字空间嵌套。XmlTextWriter 使用多个方法用来输出名字空间，为了输出精确的结果，常常使代码变得比较复杂。

在前面的方法和属性的介绍中，也提到了关于名字空间的问题。这些输出名字空间的方法多是在重载方法时，通过添加参数来加入对名字空间的支持。运用这些重载过的方法和 XmlTextWriter 本身提供的两个方法，像 LookupPrefix 和 WriteQualifiedName，我们可以表达精确的名字空间范围和前缀等参数。

关于名字空间的嵌套，我们来看一个例子：

```
XmlTextWriter writer = new XmlTextWriter (Console. Out);
writer.Formatting = Formatting.Indented;
writer.WriteStartElement ("root","http://www.Acompany.com");
writer.WriteStartElement ("child","http://www.Acompany.com");
writer.WriteEndElement ();
writer.WriteEndElement ();
writer. Close ();
```

输出的结果为：

```
<root xmlns=" http://www.Acompany.com ">
  <child/>
</root>
```

因为 root 元素和 child 元素的名字空间一致，所以在 child 元素的输出中，省略了名字空间的说明，这是符合 XML 语法的。但是，如果 root 元素和 child 元素的名字空间不相同，就不能省略名字空间的说明了。比如：child 的名字空间改为“http://www.Bcompany.com”，即“writer.WriteStartElement("child"," http://www.Bcompany.com ");”那么输出就是：

```
<root xmlns=" http://www.Acompany.com ">
    <child xmlns="http://www.Bcompany.com"/>
</root>
```

在前面介绍如何输出属性的内容中，讲到了如何为单独的属性确定名字空间以及重载方法时如何输出名字空间前缀，再结合为单独的元素确定名字空间的方法，你应该可以彻底解决输出 XML 时的名字空间问题了。通过重载 XmlTextWriter 的方法，对每一个元素或是属性都可以赋予单独的、不同的名字空间。从而保证了在元素名或是属性名相同时可以凭借名字空间来确定其惟一性，避免了发生歧义的可能。

关于名字空间，XmlTextWriter 还可以通过两个方法来获得名字空间的前缀（LookupPrefix）和名字空间的全名（WriteQualifiedName）。

```
pre=LookupPrefix("http://www.Acompany.com");
```

我们将名字空间“http://www.Acompany.com”的名字空间前缀字符串赋给了“pre”。要注意的是：pre 应是字符串类型。

得到名字空间的全名的方法是 WriteQualifiedName，用法如下：

```
writer.Formatting = Formatting.Indented;
writer.WriteStartElement("root");
 writer.WriteAttributeString("xmlns","pre", null,
  "http://www.Acompany.com");
   writer.WriteStartElement("child");
     writer.WriteStartAttribute("href",null);
     writer.WriteQualifiedName("test"," http://www.Acompany.com ");
     writer.WriteEndAttribute();
         writer.WriteString("hello,world");
   writer.WriteEndElement();
 writer.WriteEndElement();
 writer.Close();
```

得到的输出如下：

```
<root xmlns="http://www.Acompany.com">
  <child href="pre:test">hello,world</child>
</root>
```

我们将由本地名“test”和名字空间前缀“pre”构成的全名作为属性值赋给了属性“href”。

在一篇完整的 XML 文档中，还有很多节点类型，比如说，PI 指令、注释、空白、实体引用和 CDATA 节等。使用这些类型可以丰富 XML 数据文档的表现能力，提高代码的可读性。我们来逐一的介绍这些类型的内容是如何输出的。

先来看看每篇 XML 文档的第一行代码——XML 声明，及其他 PI 指令是如何输出的。XmlTextWriter 提供了 WriteProcessingInstruction 方法来处理所有 PI 输出的问题。这个方法有两个字符串参数，前一个参数是指令名，后一个参数是指令的文本内容。比如下面给出两个常见的指令输出示例：

```
writer.WriteProcessingInstruction("xmlverson". =1.0);
    //xml 声明

String PI="type='text/xsl' href='style.xsl'";
    //应用 XSLT 样式单
writer.WriteProcessingInstruction("xml-stylesheet", PI);
```

再来看看注释的输出：

```
writer.WriteComment ("This is an example.");
```

得到的结果是：

```
<! -- This is an example. -->
```

XmlTextWriter 的 WriteWhitespace 方法是用来输出空白的，并且这个方法只有一个空白字符串参数。WriteWhitespace 方法将空白参数字符串直接作为结果输出。

实体引用的输出是调用 WriteEntityRef 方法的。这个方法只有一个用来指定实体名的字符串参数。

在 XML 输出中加入 CDATA 节是通过调用 WriteCData 方法来实现的。其调用形式也比较简单，只有一个字符串参数表示要输出的 CDATA 节的文本内容。比如：

```
writer.WriteCData("<book><English/></book>");
```

得到的输出结果是：

```
<![CDATA[<book><English/></book>]]>
```

如果 XML 文档使用 DTD 的话，在文档中还要有 DOCTYPE 类型的节点。XmlTextWriter 提供了 WriteDocType 方法来输出 DOCTYPE 节点。其参数的说明如下：

```
WriteDocType(节点名, PUBLIC 关键字说明, SYSTEM 关键字说明, 子树);
```

只有当 PUBLIC 或 SYSTEM 关键字说明的值为 null 时，才能在输出时不出现 PUBLIC 或 SYSTEM 关键字。换句话说，在两个关键字说明中，至少有一个一定要为 null，因为 PUBLIC 和 SYSTEM 两个关键字不能同时出现。

与 XmlTextReader 的实例对象配合使用的方法还有 WriteNode 方法。WriteNode 将 XmlTextReader 的访问指针指向的节点的所有内容输出，然后将访问指针移动到下一个兄弟节点。对于不同类型的节点，WriteNode 方法输出的结果也是不同的。见表 7-4。

表 7-4　WriteNode 的输出列表

节点类型	输出结果
元素节点	输出元素节点和该节点的所有属性
属性节点	调用 WriteAttributeString，WriteStartAttribute 方法得到结果
文本节点	输出文本内容
CDATA 节点	输出 CDATA 节的内容
实体引用节点	输出实体引用节点的内容
注释节点	输出注释的内容
DocumentType 节点	输出文档类型节点的内容
SignificantWhitespace	输出节点的内容一空格
Whitespace	输出空格
结束元素节点	不输出
结束引用节点	不输出
XML 声明节点	输出 XML 声明

下面是调用 WriteNode 方法的示例：

```
XmlTextReader reader = new XmlTextReader("data.xml");
XmlTextWriter writer = new XmlTextWriter(Console.Out);
writer.WriteNode(reader, false);
```

XmlTextWriter 还有其他的方法来输出 XML，比如输出 XmlTextReader 的对象实例指向的节点，输出一个指定的字符实体引用等等。下表是这些方法的说明，在表格之后，我们会给出实例来说明如何使用这些方法。

7-5　XmlTextWriter 公有方法表（部分）

方法名	说明
WriteChars	用于在一次输出大量文本时，建立输出缓冲区。其参数有：用来作缓冲区的字符数组（char[]），数组中开始写的位置（int），写的总数（int）
WriteCharEntity	输出特定的参数字符实体的 unicode 值
WriteRaw	输出对标记不加分析的内容
WriteSurrogateCharEntity	输出字符实体的相应的替代值，采取 16 进制的数字输出。低位字符的取值在 0xDC00 和 0xDFFF 之间，高位字符的取值在 0xD800 和 0xDBFF 之间
WriteNmToken	输出指定的 NMTOKEN 类型节点的节点名
WriteStartDocument	输出 XML 声明，即 xmlversion = 1.0
WriteEndDocument	与 WriteStartDocument 相应，关闭所有开放的元素和属性，将写指针移回开始状态

WriteString 方法对参数字符串是进行分析的，如果有“>”、“<”、“&”则采取实体引用的方法依次输出为：>，<和&。WriteRaw 方法则不加分析地将这些标记输出。所以：

```
writer.Formatting = Formatting.Indented;
writer.WriteStartElement("string");
writer.WriteStartElement("string1");
writer.WriteRaw("5 > 4");
writer.WriteEndElement();
writer.WriteStartElement("string2");
writer.WriteString("5 > 4");
writer.WriteEndElement();
writer.WriteEndElement();
writer.Close();
```

得到输出是：

```
<string>
    <string1>5 > 4</string1>
    <string2>5 &gt; 4</string2>
</string>
```

从结果中，我们可以清楚地看到字符串“5＞4”使用两种方法输出的差异：“>”被替换为“>”。WriteRaw 方法还可以不分析地输出指定的字符数组。形式如下：

```
WriteRaw(buf, index, count);
```

其中的参数 buf 指明要输出字符数组；参数 index 是整数，指明要输出的开始位置；参数 count 则表示要输出的字符数，同样要求是整数。

如果我们要输出的是完整的 XML 文档，那么我们还要使用 WriteStartDocument 和 WriteEndDocument 方法。比如：

```
writer.WriteStartDocument();
writer.WriteElementString("name", "bill");
writer.WriteEndDocument();
```

我们将得到输出：

```
<?xml version="1.0" encoding="gb2312"?>
    <name>bill</bill>
```

由于笔者使用的是中文版的 Windows2000，所以输出时采用的是 gb2312 编码。对于一篇完整的 XML 文档，第一行“<?xml version=”1.0“encoding=”gb2312“?>”是必不可少的。如果为 WriteStartDocument 方法增加 false 参数，那么输出为：

```
<?xml version="1.0" encoding="gb2312" standalone="no" ?>
    <name>bill</bill>
```

我们通过一个对比来介绍 WriteRaw 方法，看例子：

```
writer.WriteRaw ("<name>");
writer.WriteString ("<name>");
```

得到输出为：

```
<name>
&lt;name&gt;
```

由输出我们可以看到，当使用 WriteString 方法输出“<”和“>”时，会得到“<”“>”，所以，如果要得到“<、>”这样的符号，就需要使用 WriteRaw 方法。如示例的输出结果，将“<name>”不加改变地输出。同样，“&”符号也只能由 WriteRaw 方法输出。

同 XmlTextReader 类似，XmlTextReader 也支持二进制和 base64 编码之间的转换。方法 WriteBase64 是二进制的数据输出为 base64 编码的数据，WriteBinHex 方法则将 base64 编码的数据转换为二进制输出。WriteBase64 和 WriteBinHex 方法的参数是一样的，第一个参数是指定了创建一个字节数组作为缓冲区，后两个参数是整数，用来指定缓冲区的起始

位置和要输出的字节总数。由于这两种方法不是本书的重点，所以只是介绍一下，说明 XmlTextWriter 功能方法的完整和由此体现出的对数据格式的良好支持。

7.3　结合 ASP.NET 运用“SAX”

前面我们介绍了模拟 SAX 接口的两个重要的类 XmlReader 和 XmlWriter，在本节我们将应用这两个类及其派生类的方法、属性来编写示例代码。

我们可以将 XmlWriter 的实例放入 ASP.NET 的事件处理函数中，实现收集 ASP.NET 网页中的信息到 XML 文档中，当然，我们还可以使用其他的技术来实现这个程序。看一个例子：

程序清单 7-8：example-7-5.aspx

```
<%@ Page Inherits="Sample" Src="example-9-.cs"
 EnableSessionState=true %>
<%@ Import Namespace="System.Xml" %>

</script>

<h2>Please fill these blanks.</h2>
<form runat="server">
<asp:Label Text="name"  runat="server"/>
<asp:TextBox id="Name" runat="server">
</asp:TextBox>
</p>
<asp:Label Text="Hobby"  runat="server"/>
<asp:TextBox id="Hobby" runat="server">
</asp:TextBox>
</p>
<asp:Button id="Summit" runat="server" OnClick="Summit_Click"
  Text="Summit" />
</form>
```

程序清单 7-9：example-7-5.cs

```
using System;
using System.IO;
using System.Xml;
using System.Web.UI;
using System.Web.UI.WebControls;

public class Sample : Page
{
    public TextBox  Name;
    public TextBox  Hobby;
    public Button  Summit;
    public void Summit_Click(Object Sender, EventArgs e)
    {
        if (Session["nCounter"] == null)
        {
            Session["nCounter"] = (int) 0;
        }
        else
        {
            Session["nCounter"] = (int) Session["nCounter"] + 1;
```

```
                }

                int nCounter = (int) Session["nCounter"];

                XmlTextWriter writer = new XmlTextWriter("Record.xml",
                null);

                writer.WriteStartDocument();
                writer.WriteStartElement("forms");
                writer.WriteStartElement("form");
                writer.WriteAttributeString("ID",nCounter.ToString());
                writer.WriteStartElement("name");

                writer.WriteString(Name.Text);
                writer.WriteEndElement();
                writer.WriteStartElement("hobby");
                writer.WriteString(Hobby.Text);
                writer.WriteEndElement();
                writer.WriteEndElement();
                writer.WriteEndElement();
                writer.WriteEndDocument();
                writer.Close();

            }
        }
```

生成的 aspx 页面如图 7.2。

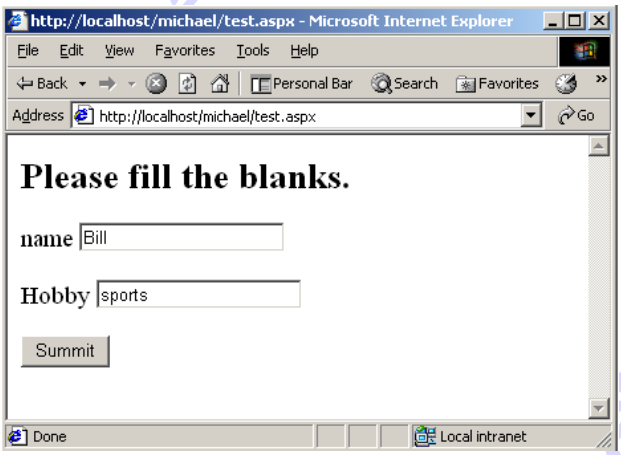


图 7.2　程序运行效果

当按图填好数据时，单击 Summit 按钮，即可生成 data.xml 文档来存放页面中的数据。我们使用了 Session 对象根据单击按钮的次数来确定 form 的 ID 属性值。

下面是第一次点击按钮生成的 XML 文档：

程序清单 7-10：Record.xml

```
<?xml version="1.0"?>
<forms>
    <form ID="0">
        <name>Bill</name>
        <hobby>sports</hobby>
    </form>
```

```
</forms>
```

其中，ID 属性等于 0，是因为 nCounter 是从 0 开始计数的。这个例子相当简单，你还可以自己完善这个收集页面数据的例子，比如增加对于提交内容的判断，只在填写的内容不相同时才写入 XML 文档。

7.4　本章小结

SAX 是除了 DOM 以外，XML 的另一个较为重要的接口，在 ASP.NET 中的应用广泛，所以本章做了详尽的介绍。同第 6 章一样，首先介绍 SAX 接口，然后介绍.NET Framework 中实现 SAX 接口的问题，重点是各种接口的介绍。

第 8 章　ASP.NET 的数据访问

数据访问已经成为现代软件开发中一个主要的任务，无论对于单独工作的应用程序还是基于 Web 的应用程序来说都是如此。ADO.NET 技术作为.NET 框架中的一部分，提供了一个解决许多和数据访问有关问题的通用解决方案。

创建基于 Web 的应用程序时，我们可以在 ASP.NET 中使用 ADO.NET 技术来通过网络和 Web 进行数据访问。

本章首先介绍有关 ADO.NET 的基础知识，然后介绍如何在 ASP.NET 应用程序中使用 ADO.NET 来进行数据访问。接着我们会对前几章已经使用过的数据绑定技术进行一个总结，来帮助读者理顺思路。最后我们会介绍 DataSet 对象和 XML 的协同工作问题。

8.1　ADO.NET 基础

ADO.NET 技术是一个基于标准的、面向创建分布式数据共享应用程序的编程模型，是 ADO（ActiveX Data Objects）技术的延伸和发展，带来了比 ADO 技术更为优越的互操作性、可维护性、可编程性和更出色的性能。

ADO.NET 包含了众多的类，程序员通过使用这些类来获得数据访问服务。这些类位于.NET Class Library 中，可以被任何基于.NET 设计的应用程序使用，包括前端的数据库客户程序、中间层的各种应用程序、工具、语言或者浏览器调用的商业逻辑对象。

ADO.NET 使用 XML 作为一般的数据传输格式，因此只要接收数据一方使用一个 XML 解析器便可以解析出数据，具有更优越的互操作性。

8.1.1　ADO.NET 的对象体系

ADO.NET 中的许多对象是从 ADO 技术中进化而来的，例如 Connection 和 Command，也有许多对象是全新的，例如 DataReader、DataSet、DataView、DataAdapter 等等。

为了将数据访问和数据操纵分离开来，ADO.NET 使用了两种组件：DataSet 对象和.NET Data Providers。

DataSet 对象在 ADO.NET 中处于核心地位，它提供了一个与数据来源无关的数据表示方式，可以表示、存储和管理来自远程或本地数据库、XML 文件或数据流甚至应用程序的局部数据。一个 DataSet 对象包含了一个 DataTable 的集合属性，用来放置一个或多个 DataTable 对象。DataTable 对象对应于关系数据库中的“表”的概念，用来容纳以行列形式组织起来的数据和主键、约束、关系等信息。

.NET Data Provider 是 ADO.NET 体系中的另一个核心元素，它包含了 Connection、Command、DataReader、DataAdapter 对象，.NET 程序员使用这些元素来实现对实际数据的操纵，这些操纵的结果，或者被直接处理，或者被放到 DataSet 对象中。.NET Data Provider 是轻量的，是一个数据源和代码间的一个尽可能小的层，从而在不牺牲功能的前提下提高

性能。

Connection 对象用来实现和数据源的链接，是数据访问者和数据源之间的对话通道。Command 对象包含了提交给实际数据库的信息，例如一个查询并返回数据的命令、一个修改数据的命令、一个调用数据库存储过程的命令及其参数等等。DataAdapter 充当了 DataSet 对象和数据源之间的桥梁，它使用 Command 对象、在 Connection 对象的连接辅助下访问数据源，将 Command 对象中的命令执行结果传递给 DataSet 对象,并将 DataSet 对象中的数据的改动回馈给数据源。DataAdapter 对 DataSet 对象隐藏了实际数据操纵的细节，从而使得 DataSet 的数据源无关成为现实。DataReader 提供了一个简单的方法，来允许程序在数据记录间进行只读的、单向（向前）的数据访问。DataReader 对象提供的数据访问接口没有 DataSet 对象那样功能强大，但性能更高，因此在某些场合下（例如一个简单的、不要求回传更新数据的查询）往往更能符合应用程序的需要。

对于任何形式的数据源，都可以有.NET Data Provider 的实现，从而允许.NET 应用程序使用这些数据源。.NET Framework 已经自带了两个 Data Provider：SQL Server .NET Data Provider 和 OLE DB .NET Data Provider。前者用于 SQL Server 7.0 或更高版本的数据源，后者用于某些版本的 OLE DB Provider。相对应地，上面提到的 Connection 对象、Command 对象、DataReader 对象、DataAdapter 对象都有两个派生类版本，它们分别位于 System.Data.SqlClient 名字空间和 System.Data.OleDb 名字空间中：

Connection：SqlConnection 和 OleDbConnection

Command：SqlCommand 和 OleDbCommand

DataReader：SqlDataReader 和 OleDbDataReader

DataAdapter：SqlDataAdapter 和 OleDbDataAdapter

SQL Server 数据源同样可以以 OLE DB 的形式被访问，但这样执行效率将不如直接使用 SQL Server .NET Data Provider 访问 SQL Server 数据源的效率高。

在.NET Framework PDC 版和 Beta 1 版中，OLEDB 对象被称为 ADO 对象，下表列出了 Beta 1 和 Beta 2 版间 ADO.NET 的一部分命名上的变化。

表 8-1　.NET Framework Beta 1 版和 Beta 2 版 ADO.NET 部分对象名称的改变

Beta 1 中	Beta 2 中
名字空间 System.Data.SQL	名字空间 System.Data.SqlClient
名字空间 System.Data.ADO	名字空间 System.Data.OleDb
ADODataSetCommand	OleDbDataAdapter
SQLDataSetCommand	SqlDataAdapter
ADOConnection	OleDbConnection
ADOCommand	OleDbConnection
ADODataReader	OleDbDataReader
DataSetView	DataViewManager

在本书撰稿时，微软公司已经发布了 ODBC .NET Data Provider Beta 1，它被设计为与

所有兼容 ODBC 的本地数据库驱动程序一起工作，但只在下列驱动程序上测试过：

Microsoft SQL ODBC Driver

Microsoft ODBC Driver for Oracle

Microsoft Jet ODBC Driver

作为.NET Framework SDK Beta 2 的附加组件，安装该组件后会在.NET Class Library 中引入新的名字空间 System.Data.Odbc 及相关的类型。

8.1.2　ADO.NET DataSet 的对象模型

DataSet 对象提供了一个驻于内存中的数据表示形式，是一个与数据源无关的关系型数据编程模型。它包含了一个由关系表、约束和表间关系构成的全集。DataSet 对象中包含的方法和对象，都是和关系型数据库模型相一致的，除此之外，DataSet 对象的内容还可以被保存为 XML 格式，并从 XML 格式恢复。

DataSet 类在 System.Data 名字空间中。它包含了以下几个重要属性：

1. DataTableCollection 类型的 Tables 属性

该属性是一个只读属性，表示了该 DataSet 对象包含的数据表集合（DataTableCollection 类），如果没有包含任何数据表，该属性是一个 null 值。

DataTableCollection 是零个或多个 DataTable 对象的集合。DataTable 对象在 System.Data 名字空间中定义，表示了一个驻于内存中的数据构成的表，它包含了一个列的集合（DataColumnCollection 对象），还包括了一个行的集合（DataRowCollection 对象）。DataColumnCollection 对象包含了零个或多个 DataColumn 对象，它们一起定义了每一行数据的内容规范，对应于关系数据库中的字段的集合；DataRowCollection 对象容纳了表中的数据，这些数据以行（DataRow 对象）的形式出现，对应于关系数据库中的记录。DataRow 对象不仅记录了它们当前的状态，还记录了初始状态并跟踪着数据发生改变。DataTable 对象还有一个 PrimaryKey 属性，该属性是一个 DataColumnCollection 对象，通过集合中的 DataColumn 的组合来构成表的主键。指定了主键的表中数据，不可以有任意两行的这些构成主键的列的值是完全相同的。

Tables 属性是只读的，这意味着程序员不应该也不可以任意的更改这个属性指向的 DataTableCollection 对象。但 DataTableCollection 对象中包含的表（DataTable 对象）是可以更改的。判断某个表是否在集合中，可以使用 DataTableCollection 对象的 Contains 方法。该方法返回一个布尔值（bool 类型）。向表集合中添加表，可以使用 DataTableCollection 对象的 Add 方法；从集合中删除表，可以使用 Remove 方法。清除集合中包含的所有表对象，可以使用 Clear 方法。注意在删除某个表前，可以使用 DataTableCollection 对象的 CanRemove 方法来确定该表是否是可以删除的。DataRowCollection 和 DataColumnCollection 对象也有类似的方法来操纵其中的 DataRow 和 DataColumn 元素。另外一个常用的方法是 DataRowCollection 对象的 Find 方法，程序员使用该方法来查找主键为给定的值组合的 DataRow，如果不存在符合条件的行，Find 方法返回 null 引用。

注意：Add、Remove 和 Clear 方法都是.NET 类库中的集合对象的标准方法。请读者参考.NET Class Library 来获取 DataTableCollection 类的继承路径。

2. DataRelationCollection 类型的 Relations 属性

该属性也是一个只读的集合属性，通过该属性来获得 Tables 属性的表集合中表间的关系的集合。每一个关系由一个 DataRelation 对象表示，关系的集合构成一个 DataRelationCollection 对象。如果没有任何关系（DataRelation 对象）存在，该属性为 null 值。操纵 DataRelationCollection 来访问、添加、删除集合中的元素，可以使用类似于 DataTableCollection 对象的方法，因为 DataRelationCollection 和 DataTableCollection 的继承路径是一样的，它们有许多共同的方法。但方法的使用可能是不一样的，譬如各个集合对象的 Add 方法都有重载过的版本。

DataRelation 对象用来将两个 DataTable 对象通过它们的 DataColumn 对象的匹配关系联系起来。建立联系的两个表，一个被称为 parent，另一个被称为 child，它们用来建立联系的列（DataColumn 对象）分别对应于关系数据库中的主键（primary key）和副键（foreign key）的关系。

每个 DataRow 对象都有一个 GetChildRows 方法，返回一个包含了通过给定的 DataRelation 对象实现关联的其他 DataRow 组成的 DataRowsCollection 对象。这些 DataRow 都是给定的 DataRow 对象在给定的 DataRelation 对象联系下的 child 对象。

每个 DataRow 对象还有一个 GetParentRow 方法和 GetParentRows 方法，它们分别返回一个通过给定的 DataRelation 实现关联的 DataRow 对象或者 DataRowCollection 对象。这些 DataRow 都是给定的 DataRow 对象在给定的 DataRelation 对象联系下的 parent 对象。

因此，使用 DataRelation 对象，程序员可以从任何一个表中的行对象（DataRow）顺着 DataRelation 关系遍历到和该对象关联的其他 DataRow 对象。

一个这样的例子是，一个记录商品信息的表 Products 和一个记录存货信息的表 Inventories。Products 表的主键是 ProductID，Inventories 表的主键是 InventoryID。另外 Inventories 表中的每一行记录都有一个 ProductID 列，该列和 Products 表中的 ProductID 列建立联系，在这里 Inventories 表里的 ProductID 列是副键，Products 表中的 ProductID 是主键，因此建立的联系中前者是 child，后者是 parent。将这两个表放入一个 DataSet 对象中。

我们可以写出建立这些数据结构的 C#代码，如下所示。为了反映普遍情况，我们又在两个表中分别添加了商品的描述信息和存货量两个列。请注意一个表的主键是列的集合，而在这里我们使用的两个表的主键均是由单个列构成的集合。

```
using System;
using System.Data;
// 建立 DataSet 对象来容纳两个表
DataSet ds = new DataSet();
// 建立两个表并放入 ds 对象
DataTable dtProducts = ds.Tables.Add("Products");
DataTable dtInventories = ds.Tables.Add("Inventories");

// 建立两个表中的列
dtProducts.Columns.Add("ProductID",
```

```
        System.Type.GetType("System.Decimal"));
    dtProducts.Columns.Add("Description",
        System.Type.GetType("System.String"));
    dtInventories.Columns.Add("InventoryID",
        System.Type.GetType("System.Decimal"));
    dtInventories.Columns.Add("ProductID",
        System.Type.GetType("System.Decimal"));
    dtInventories.Columns.Add("Inventory",
        System.Type.GetType("System.Decimal"));

    // 设置两个表的主键
    DataColumn[] pkey = new DataColumn[1];
    pkey[0] = dtProducts.Columns["ProductID"];
    dtProducts.PrimaryKey = pkey;
    DataColumn[] ikey = new DataColumn[1];
    ikey[0] = dtInventories.Columns["InventoryID"];
    dtInventories.PrimaryKey = ikey;

    // 建立两个表的 ProcudtID 列的联系
    ds.Relations.Add("ProductID",
        dtProducts.Columns["ProductID"],
        dtInventories.Columns["ProductID"]);
```

建立了数据结构后，我们可以往两个表里添加行记录。Rows 属性表示的是一个 DataRowCollection 对象，它的 Add 方法原型如下：

```
public virtual DataRow Add(DataRow);
public virtual DataRow Add(object[]);
```

在下面的例子中，我们使用第二个重载函数，首先建立包含了各字段值的 object 数组，再添加到 DataRowCollection 集合中。

```
object[] pnew = new object[2];
object[] inew = new object[3];
pnew[0] = 1;
pnew[1] = "Telephone";
inew[0] = 1;
inew[1] = 1;
inew[2] = 4;
dtProducts.Rows.Add(pnew);
dtInventories.Rows.Add(inew);
```

注意在添加数据时，已加入的 DataRelation 对象会检查数据是否符合约束条件并自动匹配关系。当数据不匹配时，会抛出异常。例如这个例子中，将 inew[1]=1 一行改为 inew[1]=2，程序就会因为遇到没有处理的异常而中止。读者可以自己查阅 SDK 文档，并设计可以捕获异常的程序。

提示：请记住，职业的程序员永远应该假设用户是愚蠢的，而程序想做的事情总会是一波三折的。在任何可能抛出异常的场合都应该编写捕获和处理异常的代码。要写出具有工业强度的代码便不可想当然，否则代码只能是玩具代码而已。

我们再来看使用 DataRelation 对象来遍历的例子。下面的例子中我们从 Products 表的第一行开始，打印出各列的信息，之后通过迭代枚举，找到这一行在已建立的 ProductID 关系下的每一个 child，打印出它们的 Inventory 列的信息。在前面建立的数据中，只有 Inventories 表中的惟一一行与 Products 表第一行关联，因此迭代只会有一次。

```
DataRow record = dtProducts.Rows[0];
```

```
        Console.Write(
            "ProductID={0}, ProductDescription={1}, ",
            record["ProductID"],
            record["Description"]);
        foreach (DataRow child in
        record.GetChildRows(ds.Relations["ProductID"]))
            Console.WriteLine("Inventory={0}", child["Inventory"]);
```

最后，我们可以对这两个表中的数据进行查询和搜索。

```
        // 搜索主键
        object[] findkey = new object[1];
        findkey[0] = 1;
        DataRow result = dtProducts.Rows.Find(findkey);
        if (result!=null)       // 必须避免访问 null 引用的不存在的成员，否则会抛
                                出一个异常
            Console.WriteLine("Result = {0}", result["Description"]);

        // 搜索 Description 为"Car"的行，将结果放入 results 集合中，并打印出
           所有结果的 ProductID
        System.Collections.ArrayList results = new
    System.Collections.ArrayList();
        foreach (DataRow eachrow in dtProducts.Rows)
            if (eachrow["Description"] == "Car")
                results.Add(eachrow);
        foreach (DataRow eachrow in results)
            Console.WriteLine("ProductID = {0}", eachrow["ProductID"]);
```

由此可见，使用 DataSet 对象来表示和存储数据是十分灵活的。虽然和 ASP 中的 Recordset 对象名称相近，但这两者是非常不同的。Recordset 对象是与数据源连接的或者不连接的，且更多情况是用来提供连接数据源的数据访问的，而 DataSet 提供的是不连接的数据访问；一个 Recordset 对象只能存放一个二维表，为了把来自多个表的数据联系起来，Recordset 对象使用 SQL 语言中的 JOIN 查询，而 DataSet 对象可以存放多个表，这些表可以用关系联系起来，我们通过 DataRelation 对象来建模这些关系；在 Recordset 中，我们对其中的行进行线性的扫描来遍历数据，在 DataSet 中，我们利用关系来从一个表中的行跳转到相关的其他表中的行，从而遍历相关的数据，利用 foreach 迭代等方法，来遍历表中的行；最后，传输不连接的 Recordset 对象使用的是 COM 对象，而传输 DataSet（它必然是不连接的）对象使用的是 XML 文件。

8.2　使用 DataSet 对象访问数据库

8.2.1　ADO.NET 访问数据库的流程

DataSet 是一个与数据源无关的内存数据表示方法，DataSet 的一个主要应用就是用来在内存中容纳访问数据库得到的数据。

ADO.NET 对数据访问过程分成了几个层次的抽象，每个层次对应了一系列类对象，由这些类对象提供的接口实现这些层次的功能。

直接和数据源交互的对象是 Connection 对象，所有对数据源的操作命令、存储过程参数和数据，都是在一定的 Connection 上面运行的。可以把 Connection 对象看成一条与数据

源连接的高速公路，所有向数据源发送的送货请求和在数据源和应用程序之间交互的数据都可以看成在这条高速公路上行驶的汽车。因此为了访问数据源中的数据，首要做的事就是建立起一条合适的 Connection 通道。不仅如此，还应该向数据源提供合适的信息，只有提交数据源认可的信息（例如访问身份和权限），这条通道才能建立起来。

向数据源提交的信息由另一个对象来容纳，这是 Command 对象。Command 对象可以是一个更新数据的语句，一个查询数据的语句，或者一个调用数据源中的存储过程的命令。这好比在 Connection 通道上行驶的汽车，装载着公路一端的码头向另一端的仓库请求的提货单，还可能装载着仓库告诉码头如何提货的信息。

最后，真正在内存中容纳了数据的对象是 DataSet 对象。它好比向仓库请求货物的码头。仓库在接受到用 Command 对象通过 Connection 对象建立的通道提交的命令后，告诉 DataSet 对象在何处（内存中的某个地方）获得需要的货物。DataSet 对象将这些数据装载入自己的空间中（类似于码头上临时堆放货物的空地），等待程序的处理（例如进行加工并运到太平洋上去，当然也可以送回到仓库里）。DataSet 对象使用 DataAdapter 对象来代理同 Connection 对象和 Command 对象之间的交互，这样 DataApapter 对象可以为 DataSet 对象隐藏掉和 Connection 及 Command 对象之间的交互细节，从而保证了 DataSet 对象的数据源无关性。DataAdapter 还负责 DataSet 对象中的数据和数据库中的数据的同步问题，这是因为 DataSet 对象提供的是无连接的数据访问服务。另外，DataSet 对象是与数据源无关的，而且可以容纳多个表。这就好像一个码头可以堆放来自不同仓库的货物一样，只要使用不同的 Connection 和 Command 对象。在数据库向 DataSet 对象传送完数据后，如果再不需要 Connection 通道和 Command 对象，它们便可以关闭，因为传送后 DataSet 对象便与数据源断开了。

另一个允许程序员访问数据库的数据对象是 DataReader 对象。DataReader 对象好比一个在给定的 Connection 通道上来回奔忙的送货员。DataReader 根据给定的 Command 对象从仓库中提取货物。为了送货效率，DataReader 每次只提取一件货物（也就是一条记录）。因此，在 DataReader 的工作周期内（例如在提取到需要的记录之前），建立连接的 Connection 对象是不能被关闭的，并且是被独占的。使用 DataReader 对象来访问数据时，不必像使用 DataSet 对象一样将数据全部放入内存中，因此效率会更高。当然它也不如使用 DataSet 对象时那样灵活，例如一个 DataReader 对象只对应到一个 Connection 对象和一个 Command 对象，并且只提供只读和单向向前的数据访问；而使用 DataSet 却可以在不同的时间使用不同的 Connection 对象和 Command 对象对它填充数据，并且 DataSet 维护一个数据整体在内存中的拷贝，可以对数据进行随机访问和修改，并利用与之匹配的 DataAdapter 对象将修改反映到数据源中。

8.2.2　使用 Connection 对象与数据库建立连接

对于不同的.NET Data Provider，ADO.NET 使用的 Connection 对象也不一样。与两种 Data Provider 相对应，Connection 有两种类型：SqlConnection 和 OleDbConnection。它们的使用方法是十分接近的：在创建连接时在类的构造器里指定连接字符串，然后调用 Open 方法打开连接。

我们来看看两种连接的创建和打开方法示例。

对于 SQL Server .NET Data Provider:

```
SqlConnection sqlconn = new SqlConnection("data source=mysqlserver;
database=mydatabase; uid=sa; pwd=");
sqlconn.Open();
```

对于 OLE DB .NET Data Provider

```
OleDbConnection oledbconn = new OleDbConnection("Provider=SQLOLEDB;
Database=mydatabase");
```

关于具体的连接字符串写法，请参考 SDK 文档。在对象创建后，还可以使用 ConnectionString 属性来获取和改变连接字符串。

在 Connection 对象不再被使用时，必须释放连接。这可以通过调用 Connection 对象的 Close 方法或 Dispose 方法来实现。注意 Connection 对象退出作用域而被 CLR 的垃圾回收器释放，并不意味着与数据库的连接也同时被释放。程序员必须显式地关闭连接。

两种 Connection 对象还有许多其他共同的属性：

表 8-2　两种 Conpection 对象共同的属性

属性	描述
ConnectionTimeOut	用来获取和数据库尝试建立连接的超时时间。该属性类型为 int，单位为秒，缺省值为 15 秒
Database	获取当前连接的或者当连接被打开时使用的数据库的名称，类型为 string
DataSource	用来获取数据源。对于 SqlConnection 对象，返回的是连接到的 SQL Server 实例的名称；对于 OleDbConnection 对象，返回的是数据源的路径和文件名
ServerVersion	返回一个字符串，该字符串包含了数据库的版本信息
State	返回一个 ConnectionState 对象，该对象用于指明连接的状态

OleDbConnection 对象特有的属性有 Provider 属性，返回 OLE DB Provider 的名字。

SqlConnection 对象特有的属性有 PacketSize 属性，返回一个 int 型数字，表明用于与 SQL Server 实例连接时的网络传输包的字节数。WorkStationID 属性返回标识数据库客户端身份的字符串。

两种对象共有的主要成员方法有：

表 8-3　两种 Connection 对象共有的主要成员方法

成员方法	描述
BeginTransaction	开始一个数据库事务
ChangeDatabase	改变已打开的连接所连接到的数据库
Close	关闭与数据库的连接
CreateCommand	创建并返回一个与该连接相关联的合适的 Command 对象
Dispose	该方法是重载过的，在显式释放对象时关闭数据库连接

8.2.3　使用 Command 对象向数据库递交信息

在和数据源建立了连接后，便可以使用 Command 对象来对数据源执行命令并从数据

源返回数据。创建 Command 对象可以使用 Command 对象的构造器，也可以使用 Connection 对象的 CreateCommand 方法。同样 Command 也有两种类型：SqlCommand 和 OleDbCommand，分别对应于两种数据源类型。

使用构造器来创建 Command 时，必须通过构造函数参数表指定一个 SQL 语句，该语句用来在数据源上执行。另外还必须传递一个 Connection 对象，该对象是 Command 对象和数据源交互的通道。例如：

```
SqlCommand sqlcmd = new SqlCommand("SELECT * FROM Books", sqlconn);
OleDbCommand oledbcmd = new OleDbCommand("SELECT * FROM Books",
oledbconn);
```

如果是使用 Connection 对象来创建 Command 对象，代码会是像这样子的：

```
SqlCommand sqlcmd = sqlconn.CreateCommand();
sqlcmd.CommandText ="SELECT * FROM Books";
OleDbCommand oledbcmd = oledbconn.CreateCommand();
oledbcmd.CommandText = "SELECT * FROM Books";
```

创建了 Command 对象后，可以使用它的一系列 Execute 方法来完成命令的执行。如果希望返回数据流，可以使用 ExecuteReader 方法来返回一个 DataReader 对象；对于 SQL 数据源，如果将 CommandText 属性置为合法的带有 FOR XML 子句的 T-SQL 语句，可以使用 SqlCommand 对象的 ExecuteXmlReader 方法来返回一个 XmlReader 对象。如果希望返回单个值，使用 ExecuteScalar 方法；执行没有返回值的 SQL 语句时，调用 ExecuteNonQuery 方法。

例如：

```
SqlDataReader dr = sqlcmd.ExecuteReader();
sqlcmd.CommandText = "SELECT * FROM Books FOR XML AUTO, XMLDATA";
XmlReader xr = sqlcmd.ExecuteXmlReader();
sqlcmd.CommandText = "DELETE * FROM Books WHERE Price = 10";
sqlcmd.ExecuteNonQuery();
sqlcmd.CommandText = "SELECT count(*) FROM Books";
sqlcmd.ExecuteScalar();
```

Command 对象不仅可以在数据源上执行 SQL 语句，也可以调用数据源提供的存储过程。一个存储过程是存放在数据库中的特定 SQL 语句的序列。数据库设计者往往从安全性的角度考虑，将某些操作以存储过程的方式实现，并允许其他开发人员调用这些存储过程，同时将数据库中的实际表和其他资源隐藏起来，不允许开发人员访问它们。这样，数据库的使用者只与这些存储过程打交道，数据库的设计者可以改变数据库的内部结构，只要保持存储过程接口不变，就不必担心破坏使用该数据库的应用程序。

调用数据源的存储过程时，首先创建一个和某个 Connection 对象关联的 Command 对象，将 CommandType 属性设为 StoredProcedure，将 CommandText 属性设为存储过程的名称，并设置其 Parameters 属性为合适的调用参数的集合。存储过程的调用参数用一个专门的 Parameter 对象表示。对于 SqlCommand，使用 SqlParameter 对象；对于 OleDbCommand，使用 OleDbCommand 对象。

Parameter 对象的主要属性有：

表 8-4　Parameter 对象的主要属性

属性	描述
ParameterName	获取或设置参数名称，string 类型，缺省为一个空串。参数名称总是以“@”号开始，紧接着实际的参数名称，参数名称必须和提供的存储过程的参数名相一致
Value	获取或设置参数的值，是一个 object 类型，缺省为 null
Direct	该属性是一个 ParameterDirection 对象，通过该对象设置或获取该参数是 Input、InputOutput、Output 还是 ReturnValue 类型的。Input 指明参数是一个输入参数，InputOutput 指明参数既可以作为输入也可以作为输出，Output 指明是一个输出参数，而 ReturnValue 指明参数是作为存储过程的返回值。缺省为 Input
DbType	

Command 对象的 Parameters 属性是一个 Parameter 对象的集合，通过向该集合中添加新的 Parameter 对象，就可以在执行命令时向存储过程传递这些参数

例如，一个 SQL Server 数据库提供了名为 QueryAuthor 的存储过程，接受一个字符串型的参数，参数名为 Author。假定我们已经创建好一个 Connection 对象为 sqlconn，则我们可以这样创建一个 SqlCommand 对象来调用该存储过程，并向 Author 参数传递值 Michael：

```
SqlCommand sqlQueryAuthor = sqlconn.CreateCommand();
sqlQueryAuthor.CommandText = "QueryAuthor";
sqlQueryAuthor.CommandType = CommandType.StoredProcedure;
SqlParameter sqlParamAuthor =
sqlQueryAuthor.Parameters.Add("@Author", SqlDbType.NVarChar);
sqlParamAuthor.Value = "Michael";
```

或者更省略一点：

```
SqlCommand sqlQueryAuthor = sqlconn.CreateCommand();
sqlQueryAuthor.CommandText = "QueryAuthor";
sqlQueryAuthor.CommandType = CommandType.StoredProcedure;
sqlQueryAuthor.Parameters.Add("@Author",
SqlDbType.NVarChar).Value = "Michael";
```

同样，SqlCommand 对象和 OleDbCommand 对象也有不少其他共同的属性和方法：

表 8-5　SalCommand 对象和 OleDbCommand 对象共同的属性和方法

属性	描述
CommandText	设置或获取在数据源上执行的 SQL 语句或者存储过程名
CommandTimeout	设置或获取在中止执行命令并产生一个错误前等待的时间，其值是一个 int 数字，以秒为单位。缺省值为 30 秒。设为 0 意味着没有限制超时时间，这是应该避免的
CommandType	设置或获取一个 CommandType 对象，该对象指明了 CommandText 属性是如何被理解的。可能的取值有：StoredProcedure、TableDirect（只适用于 OLE DB .NET Data Provider）和 Text。分别对应到存储过程名、返回所有烈的表名和 SQL 文本命令
Connection	设置或返回与该命令相关联的 Connection 对象

（续表）

属性	描述
Transaction	返回该命令所处的 Transaction 对象。对于 SqlCommand，返回的是 SqlTransaction 对象；对于 OleDbCommand，返回的是 OleDbTransaction 对象。缺省值是 null 引用

成员方法	描述
Cancel	取消命令的执行
CreateParameter	创建一个 Parameter 对象的实例。对于 SqlCommand，创建一个 SqlParameter 对象实例；对于 OleDbCommand，创建一个 OleDbParameter 对象实例
ResetCommandTimeout	将 CommandTimeout 属性值重置为缺省值

8.2.4　使用 DataReader 对象来读取数据库的数据

使用 Connection 对象和 Command 对象来同数据库连接并交互后，我们有两种办法来访问获得的结果，一是使用 DataReader 来逐行地从数据源获取数据并处理，一是使用 DataSet 对象来将数据放置到内存中进行处理。

使用 DataReader 时，首先建立与数据库的连接，然后建立要在数据库上执行的命令对象，例如一个查询 SQL 语句，然后调用命令对象的 ExecuteReader 方法来创建一个 DataReader 对象。这个创建过程是必须而且是惟一的，程序员不可能使用 DataReader 对象的构造函数来创建一个 DataReader，这是因为 DataReader 没有公共的构造函数。

因为 DataReader 和数据源类型紧密相连，它被设计为对应的两种类型：SqlDataReader 和 OleDbDataReader，分别对应于 SQL Server 数据源和 OLE DB 数据源。

连接对象打开后，程序员就可以使用 DataReader 的 Read 方法来通过关联的 Connection 对象从数据源获取到一个或多个结果集。第一次使用该方法时，ADO.NET 会将隐含的记录指针指向第一个结果集的第一条记录。

然后程序员通过调用一次 Read 方法来获取一行数据记录，并将隐含的记录指针向后移一步。Read 方法返回一个布尔值，程序员根据这个值来判断是否移到结果集的最后一条记录上，返回 true 表示仍有记录未读取，返回 false 表示已经读完最后一条记录。这样程序员可以像使用 ADO 中的 Recordset 对象一样，在结果集中迭代，从而遍历每条结果记录。

读取记录后，程序员可以使用 DataReader 的多个 Get 方法和本身的索引器（Indexer）来获得各个列的数据。既可以以列数、也可以以列名的形式使用索引器来指明要获取的列数据。

DataReader 对象的 Get 方法有多个，使用这些方法时（除了 GetBytes 和 GetChars 之外）必须确信指定列的数据类型是和该方法匹配的，否则会抛出一个异常。

表 8-6　Data Reader 对象的 Get 方法及其适应的数据类型

Get 方法	适用的数据类型
GetBoolean	Boolean
GetByte	Byte
GetBytes	将指定偏移量的字节流读入给定的 byte 数组

（续表）

Get 方法	适用的数据类型
GetChar	char
GetChars	将指定偏移量的字符读入给定的 char 数组
GetDateTime	DataTime 对象
GetDecimal	decimal
GetDouble	double
GetFloat	float
GetGuid	Guid 对象
GetInt16	short
GetInt32	int
GetInt64	long
GetString	string
GetTimeSpan	TimeSpan 对象

根据各列数据的数据类型不同，程序员选用不同的 Get 方法。例如当第 3 列（由 0 开始计数）为 String 时，使用这样的引用（假设使用 SQL .NET Data Provider，OLE DB .NET Data Provider 的使用方法类似）：

```
// …
SqlDataReader sqldr = sqlcmd.ExecuteReader();
while (sqldr.Read())
    Console.WriteLine(sqldr.GetString(3));
```

GetValue 方法获得一个对象，对象的类型为该列数据的原始类型和格式。

DataReader 对象具有索引器，因此可以使用 DataReaderObject[int] 或 DataReaderObject[“ColumnName”]的形式来访问各个列的数据。

另外，可以将一个 object 类型的一维数组传递给 GetValues 方法，来一次获取多个列的数据，该方法返回一个 int 数值表明实际放入数组的列数据个数。获取的数据列数由数组的大小决定。当数组比所有数据列的数目还大时，数组的前端（由 0 开始计数）放置了这些列的数据，程序员通过 GetValues 方法的返回值来确定数组中哪些是有效的列数据；当数组可容纳元素个数比所有列的列数要小时，数组中容纳了前 n 列数据（n 为数组大小），自然，此时的返回值总是数组的大小 n。例如：

```
// …
SqlDataReader sqldr = sqlcmd.ExecuteReader();
while (sqldr.Read())
{
    object[] cols = new object[5];
    int nGets = sqldr.GetValues(cols)
    if (5 > nGets)
        for (int i = 0; i < nGets; ++i)
            Console.Write(sqldr[i].ToString() + "   ");
    else
        foreach (object col in cols)
            Console.Write(col.ToString() + "   ");
    Console.WriteLine();
}
```

另有几个 Get 方法和获取数据无关：GetDataTypeName 返回指定列的数据类型的字符串表示；GetFieldType 返回指定列的数据类型的 Type 对象表示；GetName 返回指定列数的列名；GetType 返回当前对象（DataReader）的 Type 对象表示。

当返回的结果集有多个时，我们已经知道第一次调用 Read 方法会定位到第一个结果集的第一条记录。程序员可以通过调用 NextResult 方法来判断是否存在下一个结果集。如果方法的返回值为 true，则同时记录指针会定位到下一个结果集的第一条记录之前；如果返回值为 false，则表明已经到达最后一个结果集。定位到下一个结果集后，使用 Read 方法来使记录指针指向结果集中的第一条记录。

例如，我们可以给出遍历所有结果集中的每一条记录的 C#程序框架的一个例子：

```
SqlConnection sqlconn = new SqlConnection(
                        "data source=mysqlserver; database=Press;
                        uid=sa; pwd=password");
SqlCommand sqlcmd = new SqlCommand("SELECT * FROM Authors;" +
                                   "SELECT * FROM Books", sqlconn
                                   );
sqlconn.Open();
SqlDataReader sqldr = sqlcmd.ExecuteReader();
do
{
    // do something with each result set
    while (sqldr.Read())
    {
        // do something with each record
    }
} while (sqldr.NextResult());
```

在 DataReader 对象工作时，在与之关联的 Connection 对象上不能进行其他操作。这种状况将一直持续下去，直至 DataReader 对象的 Close 方法被调用。在调用 DataReader 对象的 Close 方法后，DataReader 对象只有两个属性可以被使用：IsClosed 和 RecordAffected。DataReader 对象还有一个 FieldCount 属性，返回当前指向的行的列数目。在使用完 DataReader 对象后，程序员必须调用 DataReader 对象的 Close 方法来关闭与数据源的连接。如果在 Command 对象中包含了输出参数，这些输出参数的值只有在 DataReader 对象被使用 Close 方法关闭后才是可用的。

另外，DataReader 对象还提供了一个 GetSchemaTable 方法，该方法返回一个没有包含数据的 DataTable，这个 DataTable 对象包含若干行和列，表示了当前结果集的数据组织模式的信息。表中的每一行表示结果集中的每一列，表中的每一列表示结果集中的列的每一个属性。

8.2.5　使用 DataSet 对象来访问数据库的数据

DataSet 是新的与数据源无关的数据表示方式。为了用来引入和容纳数据源的数据，DataSet 被设计为和另一种对象配合工作：DataAdapter。DataSet 是与数据源无关的，而 DataAdapter 则充当 DataSet 和实际数据源之间的桥梁，它可以用来向 DataSet 对象填充数据，并将对 DataSet 中的数据的改变反映到实际数据库中进行更新。为了达到这些目的，DataAdapter 对象提供了 SelectCommand、InsertCommand、DeleteCommand、UpdateCommand 和 TableMappings 属性。

将数据源中的数据填充入 DataSet 对象中，可以使用 DataAdapter 的 Fill 方法。在调用该方法前，必须指定 SelectCommand 来向数据源传递要进行的查询。SelectCommand 是一个 Command 对象，其创建方法和普通 Command 对象一样。在为 SelectCommand 属性赋值时，原来的 Command 对象并没有被复制，SelectCommand 属性只是一个指向原对象的引用。当该命令执行结果没有返回行时，DataSet 对象中既没有新表被创建，也不会抛出异常。

Fill 方法有多个重载版本，可以将数据填充到 DataSet 对象中指定表名的表（DataTable）里，如果表不存在，会自动创建表；也可以不指定表名，填充到缺省创建的表“Table”里

使用 SQL Server 数据源来向 DataSet 对象填充数据的例子如下：

```
        string strconn = "data source=mysqlserver; Initial
    Catalog=mydatabase";
        string strquery = "SELECT * FROM Authors";
        DataSet ds = new DataSet();
        SqlConnection sqlconn = new SqlConnection(strconn);
        SqlDataAdapter sqladp = new SqlDataAdapter();
        sqladp.SelectCommand = new SqlCommand(strquery, sqlconn);
        sqladp.Fill(ds);
        // do something you like with ds.Tables["Table"]
```

为了在 DataSet 对象中的数据有所改动后更新数据源，可以使用 DataAdapter 对象的 Update 方法。Update 方法使用 InsertCommand、DeleteCommand、UpdateCommand 属性的 Command 对象中的命令，来将 DataSet 中的指定 DataRow 或者 DataTable 或者全部数据更新到数据源中。DateSet 对象本身可以记录包含的数据的改变。Update 方法有多种重载形式，可以接受 DataRow 数组、DataTable 对象、DataSet 对象、DataRow 数组和 DataTableMapping 对象、DataSet 对象和表名等参数，Update 方法会根据 InsertCommand、DeleteCommand、UpdateCommand 属性将这些参数中的更改数据送回数据源。

为了减少编写 InsertCommand 等属性的编码量，ADO.NET 提供了一个 CommandBuilder 类，对应于不同的数据源，分为两种：SqlCommandBuilder 类和 OleDbCommandBuilder 类。这两个类对象可以自动为相应的 DataAdapter 生成 InsertCommand、DeleteCommand、UpdateCommand 属性需要的 Command 对象，从而简化更新数据时所需要编写的代码量。

为了使用 CommandBuilder 来简化更新数据的代码，我们首先必须生成一个 DataAdapter 对象，并设置其 SelectCommand 属性，然后创建一个 CommadnBuilder 对象与该 DataAdapter 对象关联。接着用 DataAdapter 对象的 Fill 方法填充 DataSet 对象，修改 DataSet 中的数据，然后使用 DataAdapter 对象的 Update 方法更新数据源。例如：

```
        string strconn ="data source=mysqlserver; Initial
    Catalog=mydatabase";
        string strquery = "SELECT * FROM Authors";

        DataSet ds = new DataSet();
        SqlConnection sqlconn = new SqlConnection(strconn);
        SqlDataAdapter sqladp = new SqlDataAdapter();
        sqladp.SelectCommand = new SqlCommand(strquery, sqlconn);
        SqlCommandBuilder sqlcmdbuilder = new SqlCommandBuilder(sqladp);

        sqlconn.Open();
        sqladp.Fill(ds, "Authors");

        // modify data in ds.Tables["Authors"]
```

```
sqladp.Update(ds, "Authors");
sqlconn.Close();
```

8.2.6　使用 Command 对象来从数据库返回单个值

我们已经提到过 Command 的 ExecuteSchalar 方法，该方法用来执行一条 SQL 语句，并返回执行结果的结果集的第一行的第一列的数量值。我们往往可以使用这个方法来执行例如 SELECT COUNT(*) FROM…这样的 SQL 查询：

```
SqlCommand sqlcmd = new SqlCommand("SELECT COUNT(*) FROM Authors", sqlconn);
int nCountAuthor = (int) sqlcmd.ExecuteScalar();
```

8.3　使用 DataSet 对象访问 XML 数据

DataSet 对象被设计为具有非常多的 XML 特性。DataSet 对象在传输时是以 XML 流的形式，而不是用 ADO 中的 Recordset 对象的 COM 对象的形式，这使得在异构系统间传递数据更为方便。这一点在我们学习 Web Services 的时候会看得更清楚。

DataSet 对象还可以读取 XML 数据文件或数据流，从而将树型层次结构的 XML 数据转换成关系型结构的数据形式。前者我们使用 DOM 对象模型来描述和操纵，后者我们使用表（DataTable）、列（DataColumn）、行（DataRow）、和关系（DataRelation）来描述和操纵。我们可以使用.NET 提供的 DataSet 对象和 XMLDataDocument 对象来从两个截然不同的角度操纵内存中的同一数据，并且这种操纵是同步的！

8.3.1　使用 DataSet 读取和导出 XML 数据和数据模式

DataSet 对象的设计和 XML 结合得十分紧密，DataSet 对象中的数据和数据组织模式（Schema，指的是数据是如何以表、列、关系等关系数据模型或者 XML 的层次模型来组织存放的）本质上都是以 XML 和 XML Schema 来表示的。

程序员可以使用多种方法来建立 DataSet 对象中的数据组织模式，例如使用 DataSet 的各种属性和方法来添加表、列、行和关系，或者在从数据库获取数据的同时移用数据库对这些数据的组织模式（例如获取整个表时，表的列结构便反映到目标 DataSet 对象的表中而不必手工建立列对象集合），程序员还可以使用已存在的 XML Schema 来在 DataSet 对象中建立数据模式。使用 XML XSD Schema 来在 DataSet 对象中创建相应的数据组织模式，可以调用 DataSet 对象的 ReadXmlSchema 方法。ReadXmlSchema 方法有多个重载版本，可以接受 Stream、string、TextReader 和 XmlReader 对象，程序员可以使用这些对象中的一个来告诉 ReadXmlSchema 方法从哪获取 XSD Schema 文件和流。例如：

```
// 使用文件流对象:
DataSet ds = new DataSet();
System.IO.FileStream fs = new System.IO.FileStream("xmlschema.xml",
System.IO.FileMode.Open);
ds.ReadXmlSchema(fs);
fs.Close();

// 使用文件名
DataSet ds = new DataSet();
```

```
        ds.ReadXmlSchema("xmlschema.xml");

        // 使用 StreamReader (TextReader 的派生类)
        DataSet ds = new DataSet();
        System.IO.StreamReader  sr = new
    System.IO.StreamReader("xmlschema.xml");
        ds.ReadXmlSchema(sr);
        sr.Close()

        // 使用 XmlTextReader (XmlReader 的派生类)
        DataSet ds = new DataSet();
        System.IO.FileStream fs = new System.IO.FileStream("xmlschema.xml",
    System.IO.FileMode.Open);
        System.Xml.XmlTextReader xmlreader = new
    System.Xml.XmlTextReader(fs);
        ds.ReadXmlSchema(xmlreader);
        xmlreader.Close();
```

在导入 XML XSD Schema 在 DataSet 对象中建立相应的数据组织模式后，程序员如何来得知创建的数据模式是什么样的呢？回忆 DataSet 的对象模型。DataSet 对象具有一个 Tables 属性，可以根据该属性来获取 DataSet 对象中包含的所有表的信息；对于每一个表，还具有一个 Columns 属性，根据该属性可以获取表中的所有列的信息；对于每一个列，都具有 DataType 属性，根据该属性可以获取该列的数据类型。因此，我们可以在这些表和列的集合中循环迭代，找到我们需要的所有信息。下面的代码演示了如何使用 foreach 语句来进行这种迭代：

```
        // 读入 XML 形式的数据模式
        DataSet ds = new DataSet()
        ds.ReadXmlSchema("xmlschema.xml");

        // 确定表的数目
        Console.WriteLine("There is/are {0} table(s) in the DataSet.",
    ds.Tables.Count);

        // 打印每个表的信息
        foreach (DataTable eachtable in ds.Tables)
        {
            // 打印表名和列数
            Console.Write("Table " + eachtable.TableName + " contains ");
            Console.WriteLine("{0} column(s)", eachtable.Columns.Count);
            // 打印每列的列名和数据类型
            foreach (DataColumn eachcolumn in eachtable.Columns)
            {
                Console.Write("The Type of Column " + eachcolumn.ColumnName
                + " is ");
                Console.WriteLine(eachcolumn.DataType.ToString());
            }
        }
```

程序员也可以使用 DataSet 对象的 ReadXml 方法从 XML 文件或流中导入数据。当给定的 XML 文件或流中包含了内联的 XSD Schema 时，ReadXml 方法会使用内联的 Schema，如果 DataSet 对象中已存在 Schema，则会扩展 DataSet 中现有的 Schema，如果有任何冲突，该方法会抛出一个异常。如果没有包含内联的 Schema，ReadXml 方法会尝试用推断的方法来生成或扩展 DataSet 中现有的 Schema，如果从给定的 XML 数据中无法推断出其 Schema，

则会抛出异常。ReadXml 方法的返回类型是一个 XmlReadMode 枚举类型，程序员可以通过改枚举类型的值来判断导入 XML 数据时采用的方式，其取值有 Auto、DiffGram、Fragment、IgnoreSchema、InferSchema、ReadSchema。

ReadXml 方法对于每一种 XML 数据来源（Stream、string、TextReader 和 XmlReader），都提供了两种形式的重载函数，一种是仅指定 XML 数据来源，另一种不仅指定数据来源，还允许程序员控制 ReadXml 方法在读取数据时生成数据模式 Schema 时的行为，这是通过传递一个 XmlReadMode 枚举值来控制的。

DataSet 也可以轻易的将其数据和数据组织模式导出为 XML 和 XML Schema。将数据导出为 XML 可以使用 DataSet 对象的 GetXml 方法，该方法返回一个 string 对象。将数据的组织模式导出为 XML Schema，可以使用 DataSet 对象的 GetXmlSchema 方法，同样，该方法返回一个 String。例如：

```
        SqlDataAdapter sqladp = new SqlDataAdapter("SELECT * FORM Authors",
sqlconn);
        DataSet ds = new DataSet;
        sqladp.Fill(ds,"Authors");
        string strXMLData = ds.GetXml();
        string strXMLSchema = ds.GetXmlSchema();
```

另一种将 DataSet 对象中的数据和 Schema 以 XML 的形式写出的方法是调用 DataSet 对象的 WriteXml 方法和 WriteXmlSchema 方法。这两个成员方法分别和 ReadXml 和 ReadXmlSchema 方法对应，有多个重载版本，可以将数据和数据模式写到流对象、磁盘文件、TextWrite 对象等。例如：

```
        // 从 XML 文件导入数据到 DataSet 对象中
        DataSet ds = new DataSet();
        ds.ReadXml("sourcedata.xml");

        // 使用文件流对象将数据写入文件
        System.IO.FileStream fsdata = new
System.IO.FileStream("destdata.xml", System.IO.FileMode.Create);
        ds.WriteXml(fsdata);
        fsdata.Close();

        // 使用文件名将数据模式写入文件
        System.IO.Filestream fsschema = new
System.IO.FileStream("destschema.xml", System.IO.FileMode.Create);
        ds.WriteXmlSchema(fsschema);
        fsschema.Close();
```

8.3.2　从 SQL Server 数据源读取 XML 数据

SQL Server 数据库具有输出 XML 数据的能力，只要在 T-SQL 语句中包含正确的 FOR XML 子句，SQL Server 就可以将查询结果以合适的 XML 格式返回给客户。对于 SQL Server .NET Data Provider，SqlCommand 对象拥有一个 ExecueteXmlReader 方法来执行带 FOR XML 子句的 T-SQL 语句并返回一个 XmlReader 对象来允许调用者使用包含返回结果的 XML 文档流。例如我们建立和 SQL 服务器 mysqlserver 上的数据库实例 mydatabase 间的量解，建立命令对象来查询 Authors 表的所有行并要求以 XML 格式返回查询结果。然后执行该命令，将结果放置到一个 XmlTextReader 对象中（这是一个 XmlReader 的子类）。最

后用 DataSet 对象的 ReadXml 方法导入数据，此时我们使用 Fragment 的读取模式，来指定读取 XmlTextReader 中 XML 文档的内联 XDR Schema 段。

```
        SqlConnection sqlconn = new SqlConnection("server=mysqlserver;
    uid=sa; pwd=password; database=mydatabase";
        SqlCommand sqlcmd = new SqlCommand("SELECT * FROM Authors FOR XML
    AUTO, XMLDATA", sqlconn);
        sqlcmd.CommandTimeout = 15;
        sqlconn.Open();
        XmlTextReader xmlreader = sqlcmd.ExecuteXmlReader();
        DataSet ds = new DataSet();
        ds.ReadXml(xmlreader, XmlReadMode.Fragment);
        ds.WriteXml("resultxml.xml");
```

当前来讲，只有 SQL Server .NET Data Provider 支持将查询结果以 XML 的形式表示。这并不意味着只有 SQL Server 数据库支持 XML。事实上，市场上的主要数据库软件都已经或者将包含进对 XML 的深层支持。但由于其他各种数据库软件对 XML 的支持方式不同和 OLE DB .NET Data Provider 的局限性，无法为 OleDbCommand 等对象提供类似 ExecuteXmlReader 的方法。如果为其他某个特定的数据库实现.NET Data Provider，则可能具有类似的功能。

8.4　XML 和 DataSet 对象的同步化

在上一章中我们已经知道如何使用.NET 中对 DOM 的支持来对 XML 文档树中的数据进行处理。从数据处理的角度来讲，XML 提供的是一种树型的层次视图，而我们沿用多年的关系数据模型则提供了一种基于关系的二维表视图。数据和数据的表现形式是相分离的，这意味着对于同一数据，我们可以使用不同的数据模型来描述和表示它。XML 和二维表便是两种有效的数据视图。.NET Framework 提供了 XmlDocument 来实现 DOM 编程接口，从而允许程序员从 XML 层次视图的角度处理数据；另外，.NET Framework 还提供了 DataSet，来表示关系数据模型，从而允许程序员从二维表、行、列和关系的角度来处理数据。在不同的应用场合下，使用不同的数据视图来处理数据会带来不同的便利之处。例如在处理从关系数据库获得的数据时，使用 DataSet 来表示获得的数据将使开发人员感觉十分自然和熟悉；在处理系统间信息交换问题时，XML 则表现出特别的优势。

我们已经知道，使用 DataSet 可以将 XML 文档中的数据转化为关系模式并导入 DataSet 对象中，也可以将 DataSet 对象中的关系型表示的数据以 XML 的格式写出。并且，在网络环境下，DataSet 对象也是以 XML 的形式传输的。

但.NET 还提供了一种更为便利的方法，允许程序员使用两种数据视图对同一数据进行操作，并且这种操作是同步的，这意味着程序中更改 DataSet 中数据时，这种更改会立即反映到 XML 文档视图中，程序员不必考虑如何同步这两种视图所“看到”的数据，这一切是自动的。

为了达到这个目的，.NET 提供了一个 XmlDocument 类的派生类——XmlDataDocument 类。XmlDataDocument 类扩展了 XmlDocument 类，它不仅可以将 XML 文档装载入内存中的文档树中，还允许这些装载入的结构化数据被 DataSet 类存储、获取和操纵。另外，从

DataSet 类看到的视图可以是装载入的所有 XML 数据，也可以仅是一部分被映射到关系视图上的数据。

因为 XmlDataDocument 类是 XmlDocument 类的派生类，它可以在任何 XmlDocument 对象被使用的地方替代 XmlDocument 对象。装载和导出 XML 数据的方法和 XmlDocument 是一致的。

因此，XmlDataDocument 和 DataSet 为开发人员带来极大的灵活性，可以创建单一的应用程序，对同一数据，同时使用围绕 DataSet 建立的全部服务（例如 Web Forms Controls）和与 XML 有关的一系列服务（如 Extensible Stylesheet Language、XML Transformations、XPath）。开发人员将不必在设计时为选用哪一套技术而伤脑筋，也不必忍痛舍弃任何一套技术的某些先进特性了。

将 DataSet 对象和 XmlDataDocument 对象关联起来有若干种方式。

8.4.1　为已有的关系型数据提供层次型视图

这种方法适用于当我们已经拥有一个包括数据的 DataSet 对象，而希望以 XML 层次模型的角度来访问 DataSet 中的数据的情况。自然，由于 DataSet 对象包含了数据，因而也必然已经包括了数据的组织模式。我们可以将该 DataSet 对象作为参数传递给 XmlDataDocument 对象的构造函数，这样创建出来的 XmlDataDocument 对象将和 DataSet 对象相同步，XmlDataDocument 中的数据是对 DataSet 对象的数据的引用，而不是拷贝。任何通过 DataSet 对象对数据的改变都立刻反映到 XmlDataDocument 对象中，相反，通过 XmlDataDocument 对象对数据的修改，如果这些修改与 DataSet 对象定义的数据注视模式相匹配，也会实时地反映到 DataSet 对象中。创建过程如下：

```
        DataSet ds = new DataSet();
        string ConnectionString = "data source=localhost; uid=sa; pwd=;
    database=mydb";
        string SqlString = "SELECT * FROM Authors";
        SqlConnection conn = new SqlConnection(ConnectionString);
        SqlDataAdapter adapter = new SqlDataAdapter();
        adapter.SelectCommand = new SqlCommand(SqlString, conn);
        adapter.Fill(ds);
        // the DataSet object has been created and filled with data, creating
           the schema simutaneously

        XmlDataDocument xmldoc = new XmlDataDocument(ds);
```

无论在映射 XmlDataDocument 对象到 DataSet 对象前是否对 DataSet 对象添加数据(例如使用 DataAdapter 的 Fill 方法)，都无法在映射后更改 DataSet 对象中的数据组织模式，包括增加或删除表、列、关系等对象，而只能更改行对象和各行的列数据。如果试图改动 DataSet 对象的组织模式，将会触发一个 InvalidOperationException 异常。

8.4.2　为已有的层次型数据提供关系型的视图

这种方法适用于在已经拥有一个层次型的 XML 数据文档的情况下，从关系型模式的角度去操纵这些数据。由于 DataSet 对象本身可以用 XML 文档的形式表示，在上一种方法中 XmlDataDocument 在和已包含数据的 DataSet 对象同步后便映射出 DataSet 对象中的全

部数据。在下面这种方法中，我们先生成一个不带数据的 DataSet 对象，并为该对象定义数据组织模式（也就是建立表、列、关系、约束等），然后将它与一个 XmlDataDocument 对象同步，接着再使用 XmlDataDocument 对象来向其中填充数据。这时，DataSet 对象的表名和列名作为数据组织模式的一部分，会和 XML 文档中的标记相匹配，这种匹配是大小写敏感的。DataSet 对象的数据组织模式不必和整个 XML 文档匹配，ADO.NET 会控制该匹配过程，选取可以匹配的 XML 元素来映射到 DataSet 对象中去。因此，程序员可以为 DataSet 对象定义合适的数据组织模式，使该模式仅反映整个 XML 文档层次数据模式的一部分。也就是说，通过定义 DataSet 中的数据组织模式，DataSet 对象可以成为一个非常大的 XML 文档的视图“窗口”，仅将 XML 文档中需要暴露为关系型数据的部分映射到 DataSet 对象中，这是非常有用而且经济的。在匹配过程中，虽然只有一部分数据映射到 DataSet 对象中，但匹配过程也是 XmlDataDocument 对象的装载过程，XmlDataDocument 会解析整个 XML 文档。

同样，在同步化 DataSet 对象和 XmlDataDocument 对象后，不能再让 XmlDataDocument 对象装载 XML 文档，否则，一个 InvalidOperationException 异常会被引发。

这种方法的示例代码如下：

```
DataSet ds = new DataSet();

ds.Tables.Add("books");
ds.Tables["books"].Columns.Add("title");
ds.Tables["books"].Columns.Add("price");
// create the schema in DataSet

XmlDataDocument xmldoc = new XmlDataDocument(ds);
xmldoc.Load("books.xml");
```

8.4.3　使用 XmlDataDocument 对象的 DataSet 属性

XmlDataDocument 类具有一个 DataSet 对象属性，使用这个属性可以获得一个 DataSet 对象，该对象提供了 XmlDataDocument 对象中的数据的 XML 视图。

在上一种方法中，程序员必须创建反映 XML 文档的部分数据的数据模式，这意味着必须在往 XmlDataDocument 对象中装载 XML 数据之前，程序员是了解这些数据的格式的。在某些情况下，程序员是不清楚或无法确定装载到 XmlDataDocument 对象中的 XML 数据格式的。比较容易设想到的情况是需要装载的 XML 文档可能是某几个已知格式（这些格式可以用 DTD 或 XML Schema 定义）的其中一个，并将数据格式化成二维表后输出为 HTML（我们会有数据绑定技术来简化这项工作，但总会有些情况是我们希望自己来完成它）。为了程序的可扩展性和可重用性，开发人员还希望可以处理某些未知的格式。这些情况下，程序员是无法（或者说很难）建立具有特定格式的 DataSet 来从装载的 XML 文档中提取信息的。

为了处理这种情况，XmlDataDocument 对象包含了 DataSet 属性。在向 XmlDataDocument 对象装载 XML 文档之后，DataSet 属性对象中并不包括任何数据组织模式（也就是表、列、关系等对象）。程序员可以使用遍历或 XPath 语句来获得 XML 文档的结构，然后根据文档中的节点为 DataSet 属性对象添加这些组织模式。表名和列名必须和

文档中的元素名匹配，同样，这种匹配也是大小写敏感的。这样，程序员就可以根据 XML 文档的结构来创建关系型数据视图，从而避免了在装载 XML 文档前猜测文档的层次结构。

8.4.4　几种 XML 和关系型数据同步化技术的对比

将 XmlDataDocument 对象与 DataSet 对象同步的方式具有一个优点。因为 DataSet 对象并不保存例如空白字符、层次结构、元素顺序等格式信息，而且也会忽略掉那些和 DataSet 对象中已存在的数据组织模式不相匹配的 XML 元素，因此，当使用 DataSet 对象的 ReadXml 方法来从一个 XML 文档生成关系型视图后，使用 WriteXml 方法将同一数据集合输出为 XML 文档时，文档的格式可能和原来的输入文档的格式有很大的不同。如果将一个 XmlDataDocument 对象与 DataSet 对象同步，原来的 XML 文档中的格式信息就可以保留在 XmlDataDocument 对象中，而 DataSet 只包含和它的关系型模式匹配的那部分数据和模式信息。

反过来，将 DataSet 对象与 XmlDataDocument 对象同步的方式是否会改变层次中的格式信息呢？这和 DataSet 对象中的关系信息是否嵌套有关。

在关系型数据组织模式中，不同的表之间的关系是通过这些表拥有共同的列或列集合来表达的。具体到 DataSet 对象中，ADO.NET 使用 DataRelation 对象来抽象地将这些关系表示为对象。前面已经说明了，DataRelation 对象使用父/子关系来表示将不同表联系起来的那些共同的列的关系。这里面表和列都是分离的实体。因此父/子关系都是成对出现的。

在 XML 提供的数据表示方式中，父/子关系是通过父元素中嵌套子元素来实现的。这种方式中父子关系可以是嵌套多层的，例如可以有祖父节点、父亲节点、儿子节点……

为了使从 XmlDataDocument 对象同步而来的 DataSet 对象可以将其中的父/子关系对“串”起来，从而维持嵌套结构的 XML 层次，DataRelation 对象提供了一个 Nested 属性。将这个属性置为 true 后，在输出为 XML 文档或与 XmlDataDocument 对象同步时，处于关系中儿子一方的每一行数据会被包括在处于关系中的父亲一方的列中。缺省情况下，DataRelation 对象的 Nested 属性值会是 false。

8.5　ASP.NET 的数据绑定

我们在前面几章的示例代码中已经见过不少使用数据绑定的程序。在这一节中我们对这些方法做一个总结。

数据绑定技术是服务器端控件的一个重要特性。使用数据绑定，可以大大简化控件的数据填充过程。反过来看，将数据集合作为服务器端控件的数据源绑定到控件中进行显示，是将使用前面几节介绍的服务器端数据访问技术来从数据层得到的数据反映到表现层的主要方式。这种技术主要出现在表现层逻辑的实现代码中。

同时，ASP.NET 还为 Web Forms 技术提供了数据绑定表达式语法，它是一种声明形式的表达式。程序员不仅可以将服务器端控件的任意属性绑定到数据源，还可以绑定到简单的属性、集合、表达式，甚至是函数调用的返回值。

数据绑定表达式在页面中可以出现在以下两种地方：

```
<tagprefix:tagname property="<%# databinding expression %> runat="server" />"
```

或者

```
literal text <%# databinding expression %>
```

在前者中，数据绑定表达式作为服务器端控件属性的值，后者则将值替换到页面中任何文本可以出现的地方。

数据表达式出现在表现层的样式代码中，虽然它和逻辑代码有十分密切的关联，我们仍将它与逻辑代码分开讨论。

8.5.1　ASP.NET 服务器端控件的数据绑定

Web Forms 是.NET Framework 中实现表现层的主要方法之一（另一种是 Win Forms，但显然在未来的.NET 世界中，作为 Web 应用程序表现层的 Web Forms 将会是更受重视的一种技术）。Web Forms 技术提供了服务器端控件来简化 HTML 页面的生成，程序员可以在服务器端代码中操纵这些服务器端控件，如同在 VB 等桌面应用程序中一样，ASP.NET 会负责生成这些服务器端控件的 HTML 代码并输出到请求页面的客户端。不仅如此，服务器端控件还具有与客户端浏览器类型无关的特性，无论请求页面的浏览器是 IE 或者其他 PC 浏览器，还是 PDA 或其他智能终端，ASP.NET 都会为相应的浏览器类型生成目标代码（例如为 WAP 手机生成 WML 代码），程序员在大多数情况下都不必考虑这些细节。

服务器端控件的另一个优点是，它可以很方便的将页面逻辑代码和页面表现代码分离开来。这里所说的“页面表现代码”是指那些控制内容如何显示的代码，在桌面浏览器中，这些代码往往是控制页面样式的 HTML 代码；而页面逻辑代码是指那些控制显示什么内容的代码，在 ASP.NET 中，这些代码主要是页面中的<script>标记中的内容和那些内联代码、Code-Behind 代码。

例如，我们希望动态地显示我们可以提供的商品的信息，这些信息从数据库获得，并按照统一的样式（例如左边是商品的图片，右边是一个列举商品文本信息的表格，底部是一个购买的按钮）。在以前的 ASP 页面中，我们会在 HTML 代码之间加入内联的脚本代码，来在页面被请求时由服务器解释这些代码并重复或根据条件生成 HTML 代码，并在代码中间使用内联代码来从数据库获取信息并填充到代码中间（譬如获取商品价格后填充到两个<td>标记之间）。这种方式很难将表现商品信息的样式和商品信息本身分离开来。当需要表现的商品信息发生改变（譬如数据库中增加了某一字段来描述商品另一个方面的信息）时，不仅需要修改获取商品信息的脚本程序代码，还必须修改表示样式的 HTML 代码，例如在表格中的每一行中增加一对<td>标记。这些工作往往既有美工人员参与，也有程序开发人员参与，他们的工作很难分离，也很难并行进行。常见的情况是，Web 程序开发人员会等待美工人员编写好页面的显示样式架构，然后填充那些控制显示内容的代码，或使用脚本代码来重复某些样式代码，使之应用到动态生成的显示内容上。

在 ASP.NET 中，我们可以利用 Web Server Controls 来简化和分离这些工作。请看下面的代码：

```
<%@ Page Language="C#" %>

<script runat="server">
    void Page_Load(Object Sender, EventArgs e)
```

```
    {
        if (!IsPostBack)
        {
            ArrayList values = new ArrayList();

            values.Add("Begin");
            values.Add(0);
            values.Add(1);
            values.Add(2);
            values.Add(3);
            values.Add(4);
            values.Add(5);
            values.Add(6);
            values.Add("Done!");

            List.DataSource = values;
            List.DataBind();
        }
    }
</script>

<html>
<head>
    <title>Simple DataBinding</title>
</head>
<body>
    <asp:ListBox id="List" runat="server"
        Font-Name="Verdana"
        Font-Size="11pt"
        GridLines="Both"
        CellPadding="3"
        CellSpacing="0"
        Width="100"
    />
<body>
</html>
```

在这段代码中，界面逻辑代码和界面表现代码很清晰的分置在两个部分，具体而言，界面逻辑代码在<script runat=“server”></script>标记之间，而界面表现代码则处于传统的<html></html>标记间。美工设计人员在<html></html>标记间设计页面元素的排列、显示样式等内容，程序设计人员在<script runat=“server”></script>标记间或 Code-Behind 代码中操纵页面元素的行为或动态修改它们的内容。数据绑定技术为动态改变服务器端控件的内容提供了一个极为方便的途径。许多服务器端控件都具有 DataSource 属性和 DataBind 方法，程序员可以为 DataSource 属性赋上一个集合对象变量或数据库数据源，然后使用 DataBind 方法来让服务器端控件自己从 DataSource 属性指向的对象来获取数据，并将这些数据的全部或某部分作为控件包含的数据。

具有数据绑定能力的服务器端控件主要有：

- Control 类的派生控件：Repeater 控件
- BaseDataList 类的派生类控件：DataGrid 控件、DataList 控件
- ListControl 类的派生类控件：CheckBoxList 控件、DropDownList 控件、ListBox 控件、RadioButtonList 控件

前两类控件的数据绑定行为稍有不同，主要表现在如何选择数据源的数据上，这种不同的行为是分别从两个基类继承而来的。相同点在于，这三个基类都是抽象基类（意味着不能被实例化），并且前两个基类 BaseDataList 和 ListControl 都是同一个 WebControl 类派生出来的，而 WebControl 类则是 Control 类的派生。上面所有的具体类都存在于 System.Web.UI.WebControls 名字空间中。

BaseDataList 类和 Repeater 控件主要用来显示从数据库等数据源获取的二维关系表形式的数据。使用这些控件的步骤往往是：建立与数据源的连接 Connection 对象，使用 DataAdapter 对象来选择数据库中的某些记录，然后使用 DataAdapter 对象的 Fill 方法向 DataSet 对象填充数据。如果需要，例如需要对数据进行筛选或排序时，可以根据 DataSet 对象建立新的 DataView 对象，然后设置 DataSource 属性来将 DataSet 对象或 DataView 对象绑定到这些控件上，最后调用控件的 DataBind 方法来填充数据。

Repeater 控件从 Control 类派生而来，它的外观完全由它的模板来控制。它的形式十分松散：在页面逻辑代码中确定控件的数据源（从而确定显示的数据内容），在页面的界面表现代码中编写模板，由模板来决定数据的位置、显示样式，使用数据绑定表达式来在模板中显示数据。因此页面编辑者使用 Repeater 控件可以获得类似于 ASP 时代时显示数据的控制，但不必再将程序代码放置到数据表现层代码中。Repeater 控件的模板实质是控件标记的子标记，包括 HeaderTemplate、FooterTemplate、ItemTemplate 和 AlternatingItemTemplate 等几种，标记间是标准的 HTML 代码、数据绑定表达式以及内联程序代码等。HeaderTemplate 模板标记间的代码被生成一次，作为控件头部；然后根据数据源中数据项的数量（例如如果数据源是一个表，就是记录的数量），重复生成 ItemTemplate 模板标记间的内容；如果定义了 AlternatingItemTemplate 标记，则首先按 ItemTemplate 标记生成第一个数据项的内容，然后按 AlternatingItemTemplate 标记生成第二个数据项的内容，接着交替按照 ItemTemplate 标记和 AlternatingItemTemplate 标记的定义生成各数据项的样式表示。因为 Repeater 控件在显示数据时主要由数据绑定表达式来控制显示的数据，我们会在讲述数据绑定表达式的时候再看看 Repeater 控件的例子。

BaseDataList 除了标准的数据绑定属性 DataSource 和 DataBind 方法外，还拥有另外三个和数据绑定有关的特有属性。DataKeyField 属性用来设定或获取控件中数据集的主键名称，该主键来自 DataSource 属性所引用的数据源。DataKeys 属性用来获得数据源的主键集合。当 DataSource 属性引用的数据源具有多个成员（例如具有多个表）时，使用 DataMember 属性来获取或指定绑定到控件的数据成员，该属性是 string 类型，缺省值为 String.Empty。例如按下面的方法在页面上定义一个 DataGrid 对象 dg：

```
<html>
<head>
    <title>Filling the content of DataGrid using Data Binding</title>
</head>
<body>
    <asp:DataGrid id="dg" runat="server"
        Font-Name="Verdana"
        Font-Size="11pt"
        GridLines="Both"
        CellPadding="3"
        CellSpacing="0"
```

```
                Width="400"
                Heigth="100"
            />
        <body>
        </html>
```

在页面前端的<script runat=“server”>代码中添加 Page_Load 函数并将以下代码放在函数中：

```
        SqlConnection conn = new SqlConnection("data source=(local)\\NetSDK;
    uid=sa; pwd=; database=Supply");
        SqlCommand getalbums = new SqlCommand("SELECT * FROM Albums", conn);
        SqlCommand getartists = new SqlCommand("SELECT * FROM Artists", conn);

        DataSet ds = new DataSet();
        SqlDataAdapter adapter = new SqlDataAdapter();
        adapter.SelectCommand = getalbums;
        adapter.Fill(ds, "Albums");
        adapter.SelectCommand = getartists;
        adapter.Fill(ds, "Artists");
```

运行上面的代码后，ds 对象中包含了两个表。接着分别运行下面两种代码进行绑定来显示 ds 对象中的 Aritsts 表：

方式一：

```
        dg.DataSource = ds.Tables["Artists"].DefaultView;
        dg.DataBind();
```

方式二：

```
        dg.DataSource = ds;
        dg.DataMember = "Artists";
        dg.DataBind();
```

这两种方式的运行结果是一样的。第一种方式使用 Artists 表的缺省视图作为数据源；第二种方式使用整个 DataSet 作为数据源；如果没有设定 dg 对象的 DataMember 属性，则按缺省绑定数据源中的第一个数据成员（在这里指 ds 对象中的第一个表 Albums）。

BaseDataList 类还定义了其他控制显示样式的属性，除了从 WebControl 类继承而来的 BackColor、BorderColor、BorderStyle、BorderWidth、ControlStyle、CssClass、Enabled、Font、ForeColor、Height、Style、ToolTip、Width 等属性外，还有下面特有的几个属性来定义表格的样式：

表 8-7　Base Date List 类的显示属性

样式属性	说明
CellPadding	获取或设置单元格中内容与单元格边框的空间，类型为 int，单位为像素，缺省为-1，表示属性没有被设置
CellSpacing	获取或设置单元格间的空间，同样为 int 型，但缺省值为 0
GridLines	获取或设置单元格线的样式，可能的枚举取值有 Horizontal、Vertical 、None 和 Both，缺省是 Both

BaseDataList 类的两个派生类主要不同在于：DataGrid 控件用来将数据源中的数据以表的形式显示出来，并且允许对表中的项目进行选择、排序和编辑，表格是自动生成的；DataList 控件则是用模板的方法来定义绑定的数据源数据，并且允许选择和编辑数据项。

1. DataGrid 控件

DataGrid 控件允许为表格每一列定义它们的类型，这些类型包括：

- BoundColumn 表示当前列绑定到数据源中的一个字段，将字段的值作为文本显示出来。这是控件中列的缺省类型
- ButtonColumn 将列中的每个项目显示为一个命令按钮。可以用这种类型创建一列按钮。
- EditCommandColumn 为列中的每一个项目显示一个按钮，使用该按钮可以编辑按钮所在的行的每一列数据。
- HyperLinkColumn 将列中的每一项目显示为一个超链接。
- TemplateColumn 使用模板来显示列中的每一项。

缺省情况下，控件的 AutoGenerateColumns 属性为 true，DataGrid 控件会为数据源中的每一字段产生一个BoundColumn 列，并按这些字段在数据库中存储的顺序依次显示这些列。如果设置这个属性为 false，就可以手工控制控件的显示。为了做到这一点，除了设置 AutoGenerateColumns 属性为 false 外，还必须在<Columns>标记间依次列出希望显示的特殊的列。例如：

```
<asp:DataGrid id="ItemsGrid"
    AutoGenerateColumns="false"
    runat="server">

    <Columns>

        <asp:ButtonColumn
            HeaderText="Add to cart"
            ButtonType="PushButton"
            Text="Add"
            CommandName="AddToCart"
        />

        <asp:ButtonColumn
            HeaderText="Remove from cart"
            ButtonType="PushButton"
            Text="Remove"
            CommandName="RemoveFromCart"
        />

        <asp:BoundColumn
            HeaderText="Price"
            DataField="CurrencyValue"
            DataFormatString="{0:c}"
        />

        <asp:BoundColumn
            HeaderText="Item"
            DataField="StringValue"
        />

        </Columns>

</asp:DataGrid>
```

上面的例子演示了几种定制的绑定列类型。可以看到，对于用来显示值的列（例如 BoundColumn），既可以绑定到数据源中的某个字段（用 BoundColumn 对象的 DataField 属性指定），还可以进行格式化（使用 DataFormatString 属性）；对于用来与用户交互的列（BottonColumn），可以使用 CommandName 等来控制控件的行为。也可以将 AutoGenerateColumns 功能和定制列结合起来，这种情况下，定制列会产生在自动生成的列前。

至于 DataGrid 控件的显示样式，可以通过 FooterStyle、ItemStyle、ShowFooter、ShowHeader 等属性来控制。

2. DataList 控件

DataList 控件的定义格式如下：

```
<asp:DataList 属性=值 [属性=值 …… ]>
    <模板名 属性=值 [属性=值 …… ]>
        …… (模板内容)
    </模板名>
    ……
</asp:DataList>
```

ASP.NET 根据模板内容来生成 HTML 代码

DataList 控件具有以下几种模板标记来控制数据的显示：

- HeaderTemplate 模板和 FooterTemplate 模板分别定义控件的头部和尾部，模板内容只显示一次。
- ItemTemplate 和 AlternatingItemTemplate 用来循环显示数据源中的各个记录，后者如果有定义，则交替使用这两个模板定义的样式来显示记录，否则全部使用 ItemTemplate 定义的样式。
- EditItemTemplate 模板定义了当前被编辑的项目的样式和内容，如果没有定义，则使用 ItemTemplate。
- SelectedItemTemplate 模板定义了当前被选择的项目的样式和内容，如果没有定义，则使用 ItemTemplate。
- SeparatorTemplate 模板定义了控件中各项目间的分割器样式，如果没有定义，则不会显示分割器。

在这些模板中，ItemTemplate 是最基本、必不可少的。

另外，还可以通过 DataList 控件的样式属性来定义控件各部分的样式。可以操纵的样式属性包括：HeaderStyle、FooterStyle、ItemStyle、AlternatingItemStyle、EditItemStyle、SelectedItemStyle、SeparatorStyle。和 DataGrid 类似，DataList 也有 ShowFooter 和 ShowHeader 属性来控制头部和底部的显示。

8.5.2　ASP.NET 的数据绑定表达式

数据绑定表达式用来将页面或服务器端控件的任何属性绑定到数据源上，数据的绑定发生在页面的 DataBind 方法被调用时，页面对象是服务器端控件的容器，页面的 DataBind 方法会引起容器内组件的数据绑定。因此，我们往往在 Page_Load 函数中调用 Page.DataBind

来引发页面上的数据绑定表达式的计算。例如：

```
protected void Page_Load(Object Sender, EventArgs e)
{
    // …
    DataBind();
}
```

数据绑定表达式无论是出现在页面中的什么地方，都必须包含在<%#和%>标记之间。ASP.NET 的数据绑定模型是层次型的，也就是说，支持服务器端控件属性与其父容器的数据源的绑定，可以将容器页面或者控件的上一层容器控件的公共属性（property）或公共域（field）成员作为数据源。

具体而言，在数据绑定表达式中，提供了一个静态方法 DataBinder.Eval 来计算后期绑定的数据绑定表达式的值，并可以使用它来对表达式值格式化而产生一个字符串。另外，可以使用 Container.DataItem 来获取服务器端控件的容器页面或容器控件对象。DataBinder.Eval 有两个重载版本，第一个可以接受两个参数，分别是容器对象和表达式字符串，返回一个 object；第二个版本除了接受这两个参数，还使用第三个参数来进行格式化，产生一个字符串作为数据绑定表达式的值。它可以将 Object 类型转化成一个字符串，从而在浏览器上显示出来。对于列表 Web 控件，例如我们介绍的 DataGrid、DataList、Repeater 对象，在这些控件的模板中出现的数据绑定表达式，必须使用 Container.DataItem 来获取容器对象并传递给 DataBinder.Eval 方法；对于页面上的 DataBinder.Eval 方法，则必须使用 Page 对象。

例如，可以使用这两个常用方法来构造下面的数据绑定表达式，该表达式用 Container.DataItem 获得容器对象，然后使用第二个参数获取 Price 字段的值，该值属于某一种数据类型，DataBinder.Eval 第三个参数是可选的。

```
<%# DataBinder.Eval(Container.DataItem, "Price", "{0:c}") %>
```

数据绑定表达式带来极大的灵活性，不仅可以用在基于模板的列表类控件（如 Repeater 控件和 DataList 控件）的模板中来实现数据绑定，甚至还可以做例如调用函数、动态选择数据源之类的事情：

```
<script runat="server">
//…
    string GetDataSource()
{
    //…
}
</script>

<asp:DataList DataSource=<%# GetDataSource() %>
    runat="server">
    <ItemTemplate>
        ….
    </ItemTemplate>
</asp:DataList>
```

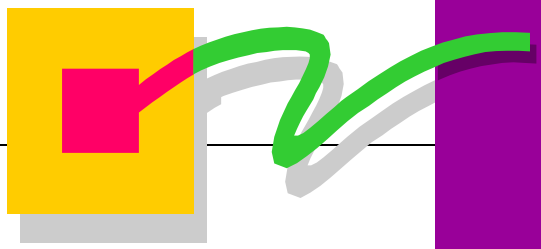
更多的例子可以在.NET SDK 文档中找到。只要是你能想得到的，使用数据绑定表达式几乎都可以达到。读者可能会觉得这些方法很像 ASP 中的内联代码，然而从本质上来说，数据绑定表达式是完全基于对象的，同时也更加结构化。

8.6　本章小结

相信读者已对 ASP 不陌生了，那么对 ADO 对象肯定再熟悉不过了，在 ASP.NET 中，也定义了这样一个访问数据库的接口，称为 ADO.NET，当然它与 ADO 已经有显著的区别了。

第三篇

XML和ASP.NET技术实现



作为本书的第三篇，内容的深度进一步加深，深入到ASP.NET 和 XML 技术的应用核心。

第 9 章首先介绍了在.NET Framework 上的 Web Service 构建技术。这种平台可以使用.NET Framework 的种种先进特性，如语言无关性、丰富的类库、集成的安全服务等等。同时，ASP.NET 提供包括 XML 在内的各种标准化的技术来描述和传输 Web Service，从而可以轻易的实现与其他平台上的 Web Service 互相沟通，这为各种网络电子商务应用提供了强大的技术支持。

第 10 章接着介绍了 ASP.NET 的一个非常有用的特性——跟踪。使用跟踪功能，用户可以快速地发现应用程序的问题所在。在本章的最后，介绍了 ASP.NET 中的安全验证和授权机制。

第 11 章为读者深入介绍 ASP.NET 技术的缓冲机制，在介绍 ASP.NET 缓冲机制及其优势的基础上引出 ASP.NET 的三种缓冲方法——输出缓冲、部分缓冲和数据缓冲。在本章的最后，作为前面说明的补充，将介绍一些 ASP.NET 直观认识缓冲技术的例子，以突出新技术的特性，并帮助读者进一步理解缓冲机制。

第 12 章则将本书涉及的技术，包括 Web Forms、Web Services、XML 及其相关技术、以及.NET Framework 的基础架构贯穿起来，并结合现代电子商务技术的发展趋势，使用.NET Framework 来创建一个简单的分布式信息流支持系统。这个系统不仅会使用到本书前面提到的几乎所有先进技术，还会演示如何使用.NET 架构中对 MSMQ（Microsoft Message Queue）的支持，来为一个现代企业设计适应下一代互联网环境的业务流程的一部分，并提供一个简单而又强大的实现基础。

第 9 章　新一代的组件 Web Services

本章我们会学习 Web Services 的概念，并介绍如何在 ASP.NET 中编写和调用 WebServices。Web Services 作为一个新兴的概念，正在被迅速的接受。ASP.NET 不仅具有创建 Web Services 的能力，而且使这种创建工作更为简单，开发人员可以迅速地掌握这种新技术，并将其运用到可编程 Web 时代的网络应用程序中去。

9.1　Web Service 和 ASP.NET Web Service

9.1.1　什么是 Web Service?

在应用开发层面上，微软.NET 战略中最重要的组成部分就是对 Web Service 的支持和实现。那么首先究竟什么是 Web Service 呢？

从开发人员的角度看，我们一直都在使用着各种各样的 Services，这些服务来自操作系统，或者来自其他组件。在桌面应用开发阶段，我们通过集成本地系统服务的方式来开发应用程序。例如，在 DOS 时代我们编写底层系统软件的时候，我们使用诸如中断的服务；在 Windows 桌面时代，我们使用 Windows 操作系统提供的 API 函数，这些也是操作系统为我们提供的服务。开发人员对这些服务的控制是相当精确的。

现在，开发人员已经在很大程度上摆脱了这种开发模式的束缚。我们已经有了 Client/Server 结构来实现分布式的两层结构的应用程序，有了 DCOM（Distributed Component Object Model 组件对象模型）等技术来实现分布对象计算中跨系统间的组件对象调用和传输。在组件时代，我们使用自己开发的或者第三方厂商提供的 COM 组件或其他组件提供的服务，来构建复杂的 N 层体系。我们使用服务的目的有很多，可能出于系统建模的考虑，可能出于软件重用的目的，也可能是我们的优秀品质：懒惰。往往我们不喜欢做些重复的工作，因此当有人提供了某些服务，并且这些服务恰好能满足我们的需求时，我们倾向于购买和使用这些服务。这样，开发人员可以将注意力集中在商业逻辑的设计和开发上，而不必为软件的基础结构（如不同组件间的相互通讯）而操心，从而缩短软件开发周期，提高开发效率。

然而，这些技术在异构系统中仍然很难得以应用，这是因为在现在的世界里存在多个分布对象计算的标准，包括微软公司的 COM/DCOM/COM+、OMG 组织的 CORBA（Common Object Request Broker Architecture）和 SUN 公司的 Java RMI（Java Remote Method Invocation）、EJB（Enterprise Jave Beans）等标准。这些技术要求服务器和客户端必须是同质的基础体系。尽管在 Windows 平台上 COM/DCOM/COM+技术被广泛应用，在非 Windows 平台上所受的支持却极为有限；CORBA 技术是 OMG 组织制定的一个协议，但不同厂商的 CORBA 实现往往不同，并引入了各自特有的某些特性，因此当开发人员使用了这些特性时，构建出来的应用程序的可扩展性便受到抑制；使用 Java RMI 和 EJB 则将开发人员绑定在 Java 语言上。另外，使用分布对象组件技术时，当任何一端接口发生变化，另一端的程

序必须相应作出变动，否则两端的通讯将会失效。在互联网时代，这些技术的这种相对紧耦合性无法适应基于 Internet 计算要求的松散性。Internet 使得众多的企业信息系统的互相沟通成为可能，但这种沟通不可能构建在一个统一的系统类型之上，因为作为开发人员，我们很难预见到与我们开发的企业级信息系统通讯的其他企业的系统类型和接口的变化，我们无法确保另一个使用的是某个操作系统、某种对象模型或某种编程语言。

而 Web Service 技术是一种基于标准的 Web 协议的可编程组件（a programmable application component accessible via standard web protocols）。我们可以把 Web Service 看作 Web 上的组件，Web 服务提供者开放一系列 API，开发人员通过调用这些 API 来集成 Web 服务，构建自己的应用程序。这种调用和从前的本地服务调用是很接近的，不同在于调用的服务存在于某个远程系统上，我们不关心这个远程系统是在地球的另一端还是在隔壁的屋子里，因为本质上来讲我们使用域名和 IP 来访问这些系统；这种调用也和从前的组件对象调用很接近，不同在于这些调用是基于消息机制和标准的 Web 协议之上的，如 HTTP、SMTP 等基于消息的异步技术，因此它们可以轻易的穿过企业的防火墙并具有相当大的可扩展性。

Web Service 包含了服务发现（Discovery）、服务描述（Description）、连接格式（Wire Format）三个阶段。服务发现阶段指的是发现和定位某个服务并获取关于其能力的信息，服务发布者通过一个 XML 格式的 disco 文件来提供给服务使用者在程序中发现服务的能力；服务描述阶段使用某些语言（例如基于 XML 的 WSDL，Web Services Description Language 语言）来获取该服务的调用规范等信息；根据 WSDL 语言中描述的连接格式，采用标准的 Web 协议如 HTTP-POST、HTTP-GET、SOAP、MIME 等来实现服务提供者和使用者之间的通讯，包括描述服务、请求服务和相应服务。

软件成为一种服务提供，这是倡导已久的论调。在基于 Internet 计算的时代，XML 技术使我们可以构建大型的、可以被任何设备、任何地点使用的应用。这使得软件被作为一种服务来提供成为可能。我们已经在各种各样的信息服务，例如移动电话，便是一种典型的信息服务：我们预订需要的服务（基础服务是接听和发出电话呼叫，扩展服务包括天气预报信息、股票信息、E-mail 服务等等），然后使用这些服务。在软件作为服务提供的时代，我们预订需要的软件（例如办公软件），然后使用它们，使用者不必再为软件的安装和配置操心，开发人员的维护工作将更简便，开发周期也将更短。

Web Service 将是把这种梦想变为现实的第一步。Web Service 可以基于 XML 技术，从而这种服务可以是自描述的。这样一来，通讯双方将不必事先掌握对方的许多信息，如调用的接口的详细信息等等。通讯前，双方之间的了解越少，双方的系统构建就会更为灵活，更具可扩展性。在 B2B 电子商务的数据交换中，Web Service 技术可以解决不同商业系统间的互操作问题。

例如，网上零售商店的应用系统集成了多个供货商的供货服务，如果这些服务均以 Web Service 的方式提供，我们只需通过 Web 标准协议和各个供货商的系统通讯来收发消息，这些消息是基于 XML 的，因此双方系统可以轻易地从 XML 流中识别出订货信息，只要这些信息是符合某个格式（可以通过 XML Schema 等标准来定义这些格式），而不必考虑对方的防火墙、对方的操作系统等等细节问题。我们的信息系统也将随时迎接新的供货商加入而

不必作出很大的修改，无论新的供货商是使用什么具体的接口技术，我们只需和对方进行通讯层上的接触。

因此我们大致可以看到，Web Service 实际是一种构建在 Web 上的组件编程技术。当前来讲我们已经有一些构建 Web Service 应用的工具，例如 Microsoft Visual Studio.NET、Interprise Delphi 6、IBM Web Service Toolkit。其中，Microsoft Visual Studio.NET 提供了构建基于.NET Framework 的 Web Services 的强大工具套件。

在服务发现的层面上，由 Microsoft 和 IBM 提出的 UDDI 规范（Universal Description, Discovery and Integration 统一描述、发现和集成协议）正在制定中。UDDI 是一套基于 Web 的、分布式的、为 Web 服务提供信息注册中心的实现标准规范，同时也包含一组使企业能将自身提供的 Web 服务注册以使得别的企业能够发现的访问协议实现标准。在 Web Service 的最初阶段，现有商业伙伴间的 Web Service 位置是已知的，这种方法很难提供一种寻找可以提供所需要的服务的商业伙伴的机制。另一种发现服务的方法是在网站上放置服务描述文件，这样可以通过网络爬虫程序来发现这些服务，然而这种方式很难跟踪服务描述的变化。

UDDI 提供了一种基于分布式的注册中心的方法，各个 Web Service 可以在任何一个 UDDI 注册中心上提交注册服务的信息。每个 UDDI 注册中心都维护一个注册信息的全集，因此这些 UDDI 注册中心构成一个分布式的 P2P 对等网，以此来同步服务的全目录 。并且，这些信息描述的格式是基于通用的 XML 格式的。服务用户可以通过任何一个 UDDI 注册中心的页面或通过其开放的 API 接口（UDDI 中心同样是一种 Web Service）来搜索获得 Web Service 的位置和技术信息。

UDDI 的规范正在制定阶段，在本书撰稿时，UDDI 组织已经制定并发布了 UDDI Specification V2 的开放草案。详细的信息，可以从 www.uddi.org 获得。

9.1.2　什么是 ASP.NET Web Service

在.NET Framework 上，所有组件都被设计为可以作为 Web Service 提供，因此基于.NET 平台的 Web Service 可以使用.NET Framework 的种种先进特性，如语言无关性、丰富的类库、集成的安全服务等。.NET 提供的 Web Service 技术采用各种标准化的技术来描述和传输 Web Service，如 XML、WSDL 、SOAP、HTTP-GET、HTTP-POST 等，从而在.NET Framework 上构建的 Web Service 可以轻易的与其他平台上的 Web Service 互相沟通。

提示：在.NET Framework Beta 1 中，Web Service 的描述语言使用的是 SDL（Services Description Language）语言而非 WSDL 语言。

.NET Framework 中构筑 Web Service 的功能被安置在 ASP.NET 体系中，这是因为 Web Service 技术带来的将是一个“可编程的 Web”（a programmable Web）。ASP.NET Web Services 提供了一层抽象，使得开发人员不必专注于 Web Services 中使用到的各种底层的例如 HTTP、SOAP、WSDL 等细节问题，而可以将注意力集中在服务本身提供的功能上。

在 ASP.NET 中，一个 Web Service 被实现为一个类，通过指定该类的某些成员函数是服务的接口，来暴露给 Web Service 的请求者，从而为请求者提供服务。这些类被放置在扩

展名为 asmx 的文本文件中。

Web Service 的请求者有两种：浏览器和程序逻辑代码。我们可以直接访问 Web Service 的 URL，在页面上通过单击调用该服务提供的方法。也可以在程序逻辑中访问 Web Service，将远程的 Web Service 作为程序的一个组件看待。为了做到后者，ASP.NET 提供了一个 Web Service Proxy Class 层，程序员可以使用.NET Framework SDK 中附带的 wsdl.exe 工具，使用要访问的 Web Service 的 WSDL 描述来产生一个与该服务对应的 Proxy Class（代理类）。在 Web Service 客户端，程序逻辑只需像使用本地对象一样使用代理类，就可以访问远程的 Web Service 了。代理类可以由 Web Service 的提供者生成并以 Assembly 的形式提供给外界，也可以由服务的使用者根据服务的 WSDL 描述生成。ASP.NET 为开发人员隐藏了大部分的细节，例如传输过程的 XML Schema 和 XML 数据的生成、数据的发送和接收等等。当然，如果开发人员需要访问底层结构来获得更精确的控制，.NET Class Library 也提供了一整套的类来帮助开发人员完成这些工作。

9.2　使用 ASP.NET 构建简单的 Web Service 应用程序

9.2.1　构建简单的 ASP.NET Web Service

在 ASP.NET 中构建 Web Service 和写一个类源文件是很接近的。这个构建过程包括两个步骤：声明一个 Web Service 和定义 Web Service 的接口方法。

我们这里使用 C#作为编程语言。一个 Web Service 是一个以 asmx 为扩展名的文本文件，其中必须包含一条@WebService 指令，该指令用来声明 Web Service。

Web Service 的实现代码由至少一个类组成，既可以写在同一个 asmx 文件中，也可以放在另一个源文件中并通过@WebService 指令来引入。可以看到这是类似于 Web Forms 和 User Controls 的 Code-Behind 代码组织方法。

我们来看一个最简单的 ASP.NET Web Service：

程序清单 9-1：noservice.asmx

```
<%@ WebService Language="C#" class="NoService " %>

public class NoService
{
}
```

将这个文件放在某个 IIS 虚拟目录下，并使用浏览器访问它。例如我们将它放到 localhost 上的 MyServices 虚拟目录下，并在 Internet Explorer 浏览器中访问 http://localhost/MyServices/noservice.asmx，浏览器的显示页面如图 9.1 所示。

该页面是由 ASP.NET 引擎产生的，用户可以通过类似的页面来访问 Web Service 的接口和自描述信息。这个 Web Service 其实什么接口和描述信息也没有提供，但他仍然作为合法的 Web Service 被对待。注意到页面上有一个 Service Description 连接，单击后出现图 9.2 的页面。这是一个 XML 文档，其内容是该服务的 WSDL 描述。

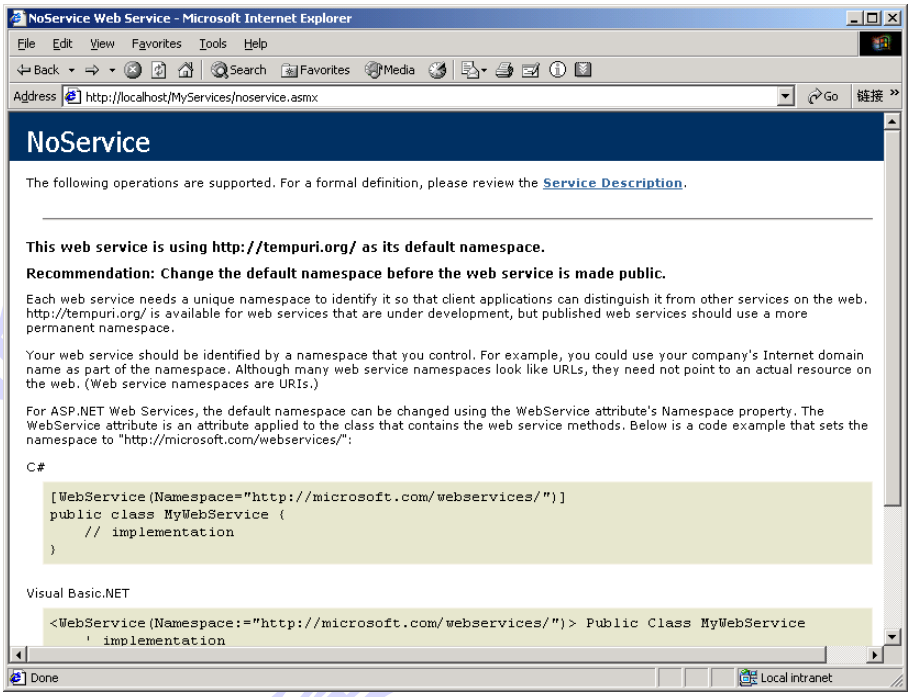


图 9.1　最简单的 ASP.NET Web Service 的显示效果

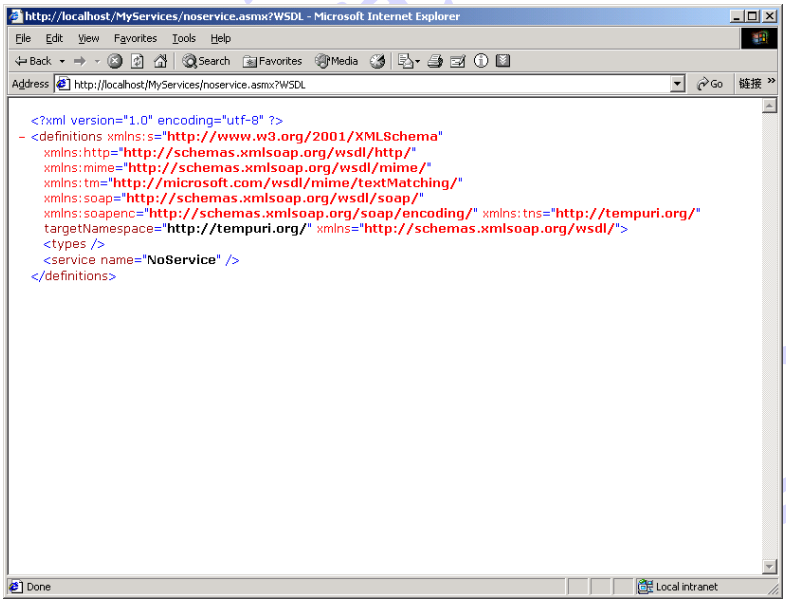


图 9.2　Web Service 服务的 WSDL 描述

从状态栏上可以看出它的 URL 是 http://localhost/MyServices/noservice.asmx?WSDL，因此我们也可以通过这个 URL 来获得 Web Service 的 WSDL 描述。

在这个 Web Service 里，我们将 Web Service 的实现代码放在同一个 asmx 文件中，通过文件顶端的@WebService 指令的 Class 属性来指定作为 Web Service 的类是文件中的哪个

类。在这个文件中，我们只有一个作为 Web Service 的 noservice 类。当然我们也可以在文件中包含其他辅助的、不作为 Web Service 暴露给外界的其他类。另外我们使用了 Language 属性来指明实现代码使用的程序语言。这个属性可以取值 C#、VB 和 JS，分别对应到 C#、Visual Basic.NET 和 Jscript.NET。

让我们再来看看一个有接口的 Web Service：

程序清单 9-2：HelloWorld.asmx

```
<%@ WebService Language="C#" class="HelloWorld" %>
using System;
using System.Web.Services;

public class HelloWorld
{
    [WebMethod]
    public string SayHello()
    {
        return "Hello";
    }
}
```

将它放置到 MyServices 目录下，并在浏览器中访问 http://localhost/MyServices/HelloWorld.asmx，得到的页面如图 9.3。

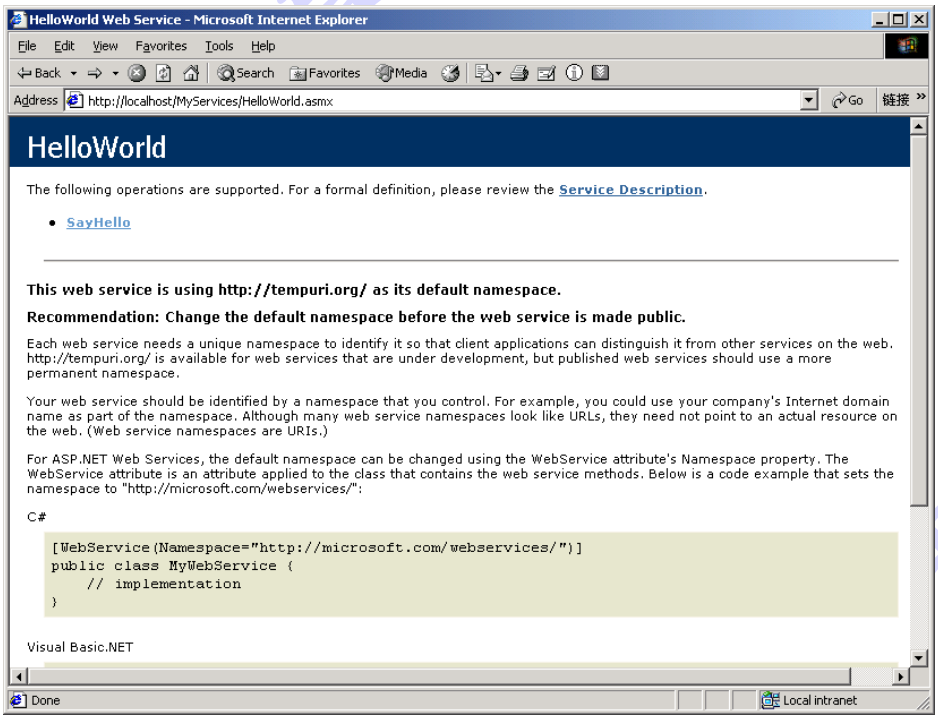


图 9.3　有接口的“HelloWorld”Web Service 的显示效果

可以看到这个 Web Service 有一个操作是对外界开放的：SayHello。这是由 HelloWorld 类代码中 SayHello 方法前面的[WebMethod]来定义的。可以看到，在 ASP.NET 中定义 WebService 接口方法是十分简单的，只要在 public 的类成员前加上[WebMethod]就可以了。

在页面中，提示了关于我们构建的 Web Service 的名字空间的信息。每一个 Web Service 都需要一个特有的名字空间，这样 Web Service 的使用者才可以将众多的 Web Service 相互区别开来。例如某个食品供应商提供了一个 Purchase 服务，另一个汽车供应商也提供了一个 Purchase 服务，为了在程序逻辑中区分它们，使用者的应用程序必须通过它们的名字空间来判断出它们不是同一个 Purchase 服务。我们没有指定 noservice 和 HelloWorld 所属的名字空间，因此 ASP.NET 为这两个 Web Service 自动生成的名字空间是 http://tempuri.org/，这是为开发中的 Web Service 保留的名字空间。作为一个公开的 Web Service，ASP.NET 建议我们修改缺省的名字空间。为了达到这个目的，可以在作为 Web Service 的类的声明前加上一句：

```
[WebService(Namespace="myuri")]
```

在 myuri 处填上指明名字空间的 URI 就可以了。

我们还可以为 Web Methods 添加上关于该方法的描述字符串，只需要在成员方法前加上。

```
[WebMethod(Description="mydescription")]
```

例如我们修改 HelloWorld.asmx 文件为：

程序清单 9-3：HelloWorldWithDescription.asmx

```
<%@ WebService Language="C#" class="HelloWorld" %>
using System;
using System.Web.Services;

[WebService(Namespace="http://microsoft.com/webservices/")]
public class HelloWorld
{
    [WebMethod(Description="Say Hello to you")]
    public string SayHello()
    {
        return "Hello";
    }
}
```

则访问 http://localhost/MyServices/HelloWorldWithDescription.asmx 的结果如图 9.4。

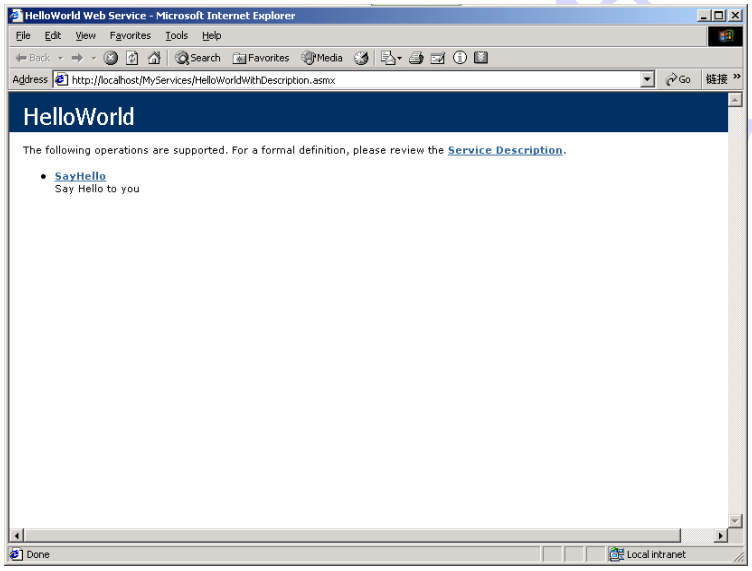


图 9.4　为 Web Methods 添加上关于该方法的描述字符串

关于名字空间的提示没有了，并且 Web Service 提供的操作下，也有了一行注释。

在页面上，Web Service 提供的操作是以超链接的形式出现的。在这个例子中浏览者可以单击 SayHello 超链接，来获得这些操作的详细信息和设置调用这些操作的参数。当在浏览器里单击 Invoke 按钮时，ASP.NET 服务器会在服务器端执行这些操作的代码，并将该成员方法的返回值以 XML 的形式回传给浏览器。单击 SayHello 的结果如图 9.5。因为 SayHello 方法参数表为空，页面上没有填写参数的控件。在图 9.5 所示页面中单击 Invoke 按钮，结果如图 9.6。很明显，这是一个 XML 文档。

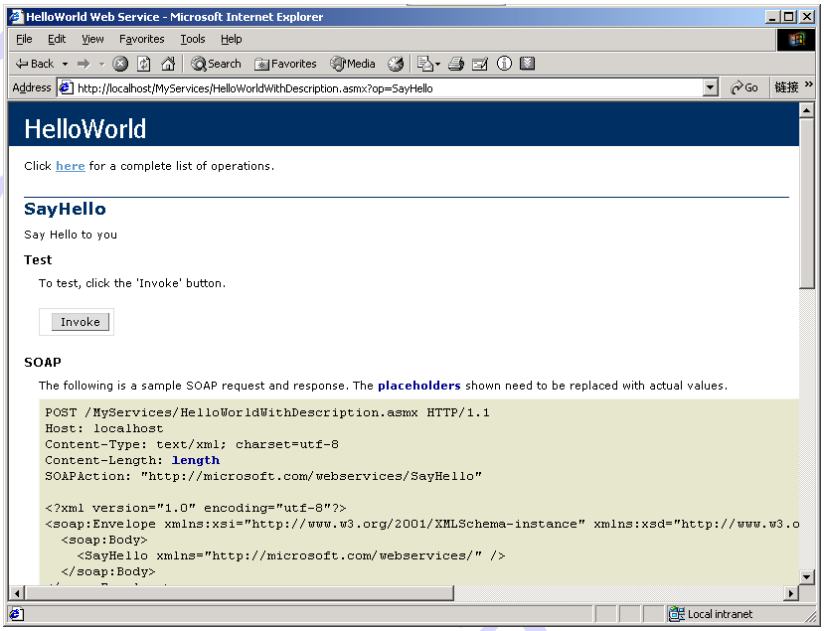


图 9.5　单击 SayHello 的运行结果

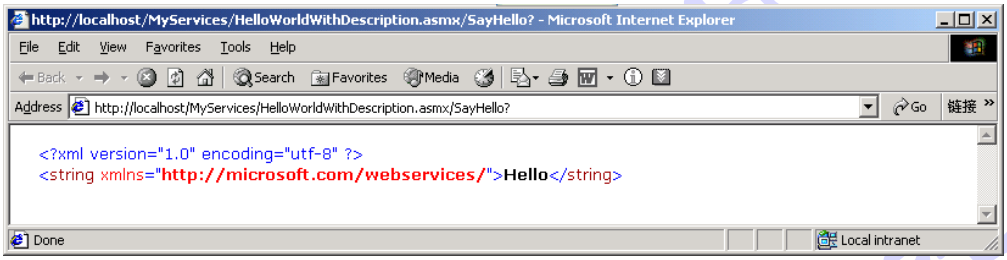


图 9.6　单击 Invoke（调用）按钮可显示 XML 文档

到现在为止，我们已经演示了如何创建一个简单的 ASP.NET Web Service，并通过浏览器来调用它。然而 Web Service 更多的调用是在程序逻辑中发生的。在其他应用程序可以调用这些 Web Services 之前，必须先发布 Web Service。

9.2.2　发布 Web Service

发布 Web Service 的主要工作是将 asmx 文件及其他相关的被该 Web Service 使用而不是.NET Framework 组成部分的文件拷贝到将要提供 Web Service 的 Web 服务器上。这包括

以下几个步骤：

（1）在目标 Web 服务器上的 IIS 中建立虚拟目录。

（2）将 asmx 文件和 disco 文件（如果有的话）放置到该虚拟目录中。asmx 文件将作为客户调用 Web Service 时的 URL 入口点。disco 文件为该 Web Service 提供了一个服务的发现机制，来向特定的客户暴露出可用的 Web Services 的信息。

（3）将 Web Service 中使用到的而不是.NET Framework 组成部分的 assembly 文件放到虚拟目录下的 bin 子目录中。

（4）在客户端程序代码中调用 ASP.NET 服务时，与客户端程序打交道的不是 Web Service 类，而是其代理类。客户端像使用本地对象一样使用代理类对象，由代理类对象负责隐藏与实际的远程 Web Service 的通讯。因此，发布 Web Service 的另一个主要工作就是发布代理类。

.NET Framework SDK 中提供了一个命令行工具 wsdl.exe，来帮助开发人员从 Web Service 的 WSDL 语言描述、XSD Schema 或 discomap 文件生成该 Web Service 的代理类源代码。例如我们使用 wsdl.exe 来生成 HelloWorld.asmx 的 C#语言的代理类：

```
wsdl http://localhost/MyServices/HelloWorld.asmx
```

该命令会在当前路径下生成一个 HelloWorld.cs 文件，文件的内容如下：

程序清单 9-4：HelloWorld.cs

```
//------------------------------------------------------------------
// <autogenerated>
//     This code was generated by a tool.
//     Runtime Version: 1.0.2914.16
//
//     Changes to this file may cause incorrect behavior and will be
lost if
//     the code is regenerated.
// </autogenerated>
//------------------------------------------------------------------

//
// This source code was auto-generated by wsdl, Version=1.0.2914.16.
//
using System.Diagnostics;
using System.Xml.Serialization;
using System;
using System.Web.Services.Protocols;
using System.Web.Services;


    [System.Web.Services.WebServiceBindingAttribute(Name="HelloWorld
Soap", Namespace="http://microsoft.com/webservices/")]
    public class HelloWorld :
System.Web.Services.Protocols.SoapHttpClientProtocol
    {

        [System.Diagnostics.DebuggerStepThroughAttribute()]
        public HelloWorld()
        {
            this.Url = "http://localhost/MyServices/HelloWorld.asmx";
        }
```

```
        [System.Diagnostics.DebuggerStepThroughAttribute()]
    [System.Web.Services.Protocols.SoapDocumentMethodAttribute(
        "http://microsoft.com/webservices/SayHello",
    RequestNamespace="http://microsoft.com/webservices/",
ResponseNamespace="http://microsoft.com/webservices/",
Use=System.Web.Services.Description.SoapBindingUse.Literal,
ParameterStyle=System.Web.Services.Protocols.SoapParameterStyle.Wrap
ped)]
        public string SayHello()
        {
            object[] results = this.Invoke("SayHello", new object[0]);
            return ((string)(results[0]));
        }

        [System.Diagnostics.DebuggerStepThroughAttribute()]
        public System.IAsyncResult BeginSayHello(System.AsyncCallback
         callback, object asyncState)
        {
            return this.BeginInvoke("SayHello", new object[0], callback,
             asyncState);
        }

        [System.Diagnostics.DebuggerStepThroughAttribute()]
        public string EndSayHello(System.IAsyncResult asyncResult)
        {
            object[] results = this.EndInvoke(asyncResult);
            return ((string)(results[0]));
        }
    }
```

可以看到，这是一个标准的 C#源文件，它使用的全部是.NET Class Library 中的类对象和.NET 支持的语法。因此，我们完全有可能自己创建这样的代理类。

wsdl.exe 命令行工具提供了多个选项开关，可以指定生成的源文件语言（C#、VB、JS，缺省是 C#），可以指定输出的源文件名，还可以指定代理类与 Web Service 通讯时使用的协议（SOAP、HttpGet、HttpPost）等等，在这里我们不详细介绍各个命令行参数，请读者自行阅读工具的说明。

创建了代理类后，必须先编译成 dll 文件后再由客户端使用。编译的命令行大致如下：

```
csc /target:library HelloWorld.cs
```

编译器会在当前目录下输出 HelloWorld.dll 文件。

客户端应用程序的开发人员将该文件放置到客户端的 ASP.NET 应用程序所在目录下的 bin 目录中，就可以在 ASP.NET 代码中使用这个代理类。

9.2.3　在客户端中调用 Web Service

Web Service 的客户端可以是任何引用和使用 Web Service 的组件或应用程序。因此，这个客户端并不一定是一个运行在 Web 应用程序客户端的组件或应用程序，它可以是一个 Web 应用程序的服务器端的 Web Forms 或 Code-Behind 代码，甚至也可以是一个 Web Service。无论是哪种类型的客户端，它们对 Web Service 的调用方式都是十分类似的：

- 获得 Web Service 的代理类
- 在客户端代码中引用代理类

- 在客户端代码中创建代理类的实例
- 通过调用代理类实例的方法来调用相应的 Web Service 接口方法

对于大多数的客户端来说，这些步骤主要在如何引用代理类和如何配置 Web Service 客户端上。我们将仅介绍如何在 ASP.NET 应用程序中调用 Web Service。

在 ASP.NET 应用程序中调用 Web Service 时，可以像调用本地组件一样对待。不同的在于，ASP.NET 会在调用点路径下的 bin 目录下寻找编译好的 Web Service 的代理类的 dll 文件，而不必在程序中显式指明该代理类在何处。

例如，我们在 Web Forms 中调用 HelloWorld 服务的 SayHello 方法的代码如下。注意我们将 HelloWorld.dll 文件放置在 client.aspx 文件所在的虚拟目录下的 bin 子目录中。至于如何生成代理类并编译为 dll 文件，请参阅上一节内容。

程序清单 9-5：client.aspx

```
<%@Page Language="C#"%>

<script language="C#" runat="server">
void Page_Load(Object Sender, EventArgs e)
{
    HelloWorld hellosvc = new HelloWorld();
    lbl.Text = hellosvc.SayHello();
}
</script>
<asp:Label id="lbl" runat="server"/>
```

我们已经提到，Web Services 技术是一种基于消息的远程方法调用。在调用 Web Service 接口方法时，我们实际上是向 Web Service 所在 Web 服务器（在.NET 中我们采用 IIS 服务器）发送一定格式的 XML 消息。Web Services 的描述文档说明了它们能接受的这些 XML 消息的格式。调用的方法名称、参数和方法执行后的返回值都是这样的 XML 流。XML 消息到达服务器后服务器会识别出这些 XML 流，解析出调用的方法和参数，并由服务器创建 Web Service 对象，调用相应方法的程序代码。执行后，服务器会向客户端回传方法的返回值和按引用调用的参数。传输过程中的 XML 流可以是以 SOAP 消息或者 HTTP-GET、HTTP-POST 等方式传输。

在 ASP.NET Web Service 中，如何发送调用请求、传输请求消息、解析请求消息、执行实际代码、返回请求结果的步骤都被很好的封装到 Web Service 类的代理类和 IIS 服务器及.NET Framework 运行中了。大部分时间里，应用的设计者和程序员只需考虑要传递什么请求和如何处理请求结果就足够了。当然如果需要，.NET 也允许程序员接触更底层的运行控制，这从前面生成的代理类代码是完全的 C#代码就可以得到体现了。

9.3　构造更复杂的 Web Service 和 Web Service 客户端

9.3.1　应用场景和用户体验概述

现在我们构建一个复杂一点的、带参数的 Web Service Method，再看看如何在一个 ASP.NET Web Service 中如何调用这个 Web Service，以及如何传递参数和返回值。你会发

现这几乎和本地对象的调用没有什么区别。

我们的应用环境会是一个电子商务环境中经常遇见的场景：我们的企业接受最终用户或者供应链下游企业的供货查询和订货单，并向我们的上游企业，也即我们的供货商，查询其供货能力，并根据我们的客户的订单向我们的供货商订货。当然，我们会向我们的客户和供货商提供增值服务（否则我们的客户何不直接向我们的供应商订货呢？），这可以通过多种方式来完成，例如我们集成了多个供货商的供货服务并向客户提供搜索、比较、商品推介等服务。为了便于阐述问题，我们的例子中只实现了一个供货商。这不妨碍我们技术上的讨论。

可以看到，这个应用场景中有多个服务：我们的供货商向我们提供供货能力查询和订货服务，我们向客户提供商品查询和订货服务。这些服务都可以以 Web Service 的形式来实现，这样，最终客户在我们的网站上订购商品，这种情况下网站的 Web Forms 会调用我们为客户提供的 Web Service 并将结果以 Web Forms 的形式在浏览器中反馈给最终客户；客户也使用其他客户端程序来调用我们暴露出的 Web Service，这些客户端程序理论上讲可以由任何人提供，因为我们开放了我们的 Web Service 的调用规则（可能通过网站上的 WSDL 描述文件，可能通过 UDDI 注册中心，也可能通过开发人员间的电话联系）。无论以哪种形式提交的服务请求，本质上都是 XML 描述的数据，因此我们的 Web Service 并不比关心请求来自 Web 网站的浏览者还是坐在纽约中心公园里用移动电话（当然，假设电话中嵌入了合适的客户端程序）订货的下了班的人。我们的 Web Service 统一收集需要的信息，并向供货商提交他们感兴趣的信息。例如，当顾客告诉我们的 Web Service 希望购买某张唱片时，Web Service 可以向 20 家供应商的 Web Services 查询他们是否供应这张唱片，并向顾客提交可以供应唱片的供应商的供货价格，由顾客选择向哪家供应商购买唱片。这是一个查询商品的过程。对于订货过程，也是类似的，当然这个过程可能要加入对安全性、个性化等的支持，这些技术和 Web Service 是处在不同层次的，因此我们将这些话题留给读者自己探讨。

9.3.2　供应商 Web Services 的实现

首先我们来实现一个供货商的 Web Service。

作为一个供货商，我们需要一个储存商品信息的数据库。我们使用 SQL Server 来构建这个数据库。我们采用 SQL Server 2000 版本，在某个数据库实例中（假设采用.NET Framework SDK 建立的.NETSDK 实例）创建一个新的名为 Supply 的数据库，并创建四个表：Albums、Artists、ArtistTypes、Orders。各个列的设置如表所示。

表 9-1　Albums 表的结构定义

列名	数据类型	长度	允许空	主键
AlbumID	int	4	否	是
Title	nvarchar	50	是	否
AritistID	int	4	否	否
Price	money	8	是	否

(续表)

列名	数据类型	长度	允许空	主键
Style	nvarchar	50	是	否
Description	ntext	16	是	否
Inventory	int	4	是	否

表 9-2　Artists 表的结构定义

列名	数据类型	长度	允许空	主键
ArtistTypeID	int	4	否	是
TypeDescription	nvarchar	50	是	否

表 9-3　ArtistTypes 表的结构定义

列名	数据类型	长度	允许空	主键
OrderID	int	4	否	是
Customer	nvarchar	50	否	否
AlbumID	int	4	否	否
Number	int	4	否	否
Shipped	bit	1	否	否

为了更好的演示程序的运行效果，我们依次在这些表中添加示范数据：

表 9-4　向 ArtistTypes 表添加数据

ArtistTypeID	TypeDescription
1	Male
2	Female
3	Band

表 9-5　向 Artists 表添加数据

ArtistID	Artist	ArtistTypeID	Description
1	Beatles	3	A famous Band from UK
2	Michael Jackson	1	A Super Star from US
3	Sinead O'Cornor	2	An Irish Singer
4	The Cranberries	3	A young band in Ireland
5	Suede	3	A Brit-Pop band

表 9-6　向 Albums 表添加数据

AlbumID	Title	ArtistID	Price	Style	Description	Inventory
1	No Need To Argue	4	10	Rock		100
2	Yellow Submarine	1	20	Rock		20
3	Black Or White	2	15	Pop		70

另外，创建关系图，将 Albums 表中的 ArtistID 字段作为外键和 Artists 表主键 ArtistID 字段关联，将 Artists 表中的 ArtistTypeID 字段作为外键和 ArtistTypes 表主键 ArtistTypeID 字段关联，将 Orders 表中的 AlbumID 字段作为外键和 Albums 表主键 AlbumID 字段关联。

然后我们编写实际的 Web Service，实现查询是否供货、查询商品详细信息、查询供货能力、订货四个简单的接口方法。完整的代码如下：

程序清单 9-6：DBServices.asmx

```
<%@ WebService Language="C#" class=Supplier.DBServices%>
using System;
using System.Data;
using System.Data.SqlClient;
using System.Web.Services;

namespace Supplier
{

    public class InvalidParameterException : Exception {}

    [WebService(Namespace="http://localhost/MyServices/")]
    public class DBServices
    {
        private const string ConnectionString =
        "server=(local)\\NetSDK; database=Supply; uid=sa; pwd=";
        private const string CompanyString = "Sonier";

        private const string DetailString = "SELECT * " +
            "FROM Albums INNER JOIN Artists " +
            "ON Albums.ArtistID=Artists.ArtistID " +
            "INNER JOIN ArtistTypes " +
            "ON Artists.ArtistTypeID=ArtistTypes.ArtistTypeID " +
            "WHERE Albums.Title=@AlbumTitle";
        private const string ListString = "SELECT AlbumID, Title,
        Price FROM Albums";
        private const string CountString = "SELECT COUNT(*) FROM Albums
        WHERE Title=@AlbumTitle";
        private const string VerifyString = "SELECT COUNT(*) AS
        RowNumber, SUM(Inventory) AS TotalInventory FROM Albums
        WHERE Title=@AlbumTitle";
        private const string QueryString= "SELECT AlbumID, Inventory
        FROM Albums " +
            "WHERE Title=@AlbumTitle";
        private const string OrderString = "INSERT INTO Orders " +
            "(Customer, AlbumID, Number, Shipped) " +
            "VALUES (@Customer, @AlbumID, @Number, @Shipped)";
        private const string UpdateString = "UPDATE Albums SET " +
            "Inventory=@Inventory WHERE AlbumID=@AlbumID";

        [WebMethod(Description="Get the Company Name")]
        public string GetCompany()
        {
            return CompanyString;
        }

        [WebMethod(Description="Get the album list in the
        database")]
        public DataSet GetAlbums(string TableName)
```

```
        {
            DataSet ds = new DataSet();
            SqlConnection conn = new
            SqlConnection(ConnectionString);
            try
            {
                SqlDataAdapter adapter = new SqlDataAdapter();
                adapter.SelectCommand = new SqlCommand(ListString,
                conn);
                adapter.Fill(ds, TableName);
            }
            catch (Exception e)
            {
                throw e;
            }
            finally
            {
                if (conn.State == ConnectionState.Open)
                    conn.Close();
            }
            return ds;
        }

        [WebMethod(Description="Get the detail information of the
        specific album")]
        public DataSet GetAlbumDetail(string AlbumTitle, string
        TableName)
        {
            DataSet ds = new DataSet();
            SqlConnection conn = new
            SqlConnection(ConnectionString);
            try
            {
                SqlDataAdapter adapter = new SqlDataAdapter();
                adapter.SelectCommand = new
                SqlCommand(DetailString, conn);

adapter.SelectCommand.Parameters.Add("@AlbumTitle",
SqlDbType.NVarChar).Value = AlbumTitle;
                adapter.Fill(ds, TableName);
            }
            catch(Exception e)
            {
                throw e;
            }
            finally
            {
                if (conn.State == ConnectionState.Open)
                {
                    conn.Close();
                }
            }
            return ds;
        }

        [WebMethod(Description="Determine whether the Supplier has the
        specific Album in the inventory")]
        public bool HasAlbum(string AlbumTitle)
        {
```

```
        SqlConnection conn = new
        SqlConnection(ConnectionString);
        bool bResult;
        try
        {
            SqlCommand cmd = new SqlCommand(CountString, conn);
            cmd.Parameters.Add("@AlbumTitle",
            SqlDbType.NVarChar).Value = AlbumTitle;
            conn.Open();
            bResult = ((int)cmd.ExecuteScalar() > 0);
            conn.Close();
        }
        catch(Exception e)
        {
            throw e;
        }
        finally
        {
            if (conn.State == ConnectionState.Open)
            {
                conn.Close();
            }
        }
        return bResult;
    }

    [WebMethod(Description="Determine whether the supplier has
    enough inventory to accept an order or not")]
    public bool CanSupply(string AlbumTitle, int Number)
    {
        if (Number < 0) throw new InvalidParameterException();
        SqlConnection conn = new
        SqlConnection(ConnectionString);
        bool bResult = false;
        try
        {
            SqlDataAdapter adapter = new SqlDataAdapter();
            SqlCommand cmd = new SqlCommand(VerifyString, conn);
            cmd.Parameters.Add("@AlbumTitle",
            SqlDbType.NVarChar).Value = AlbumTitle;
            conn.Open();
            SqlDataReader reader = cmd.ExecuteReader();
            reader.Read();
            if ((int)reader["RowNumber"] > 0 &&
            (int)reader["TotalInventory"] >= Number)
                bResult = true;
            conn.Close();
        }
        catch(Exception e)
        {
            throw e;
        }
        finally
        {
            if (conn.State == ConnectionState.Open)
            {
                conn.Close();
            }
        }
```

```
    return bResult;
}

[WebMethod(Description="Order a specific Album")]
public bool Order(string AlbumTitle, int Number, string
Customer)
{
    bool bResult = false;

    if (CanSupply(AlbumTitle, Number)) {
        SqlConnection conn = new
        SqlConnection(ConnectionString);

        try
        {
            SqlCommand ordercmd = new
            SqlCommand(OrderString, conn);
            SqlCommand updatecmd = new
            SqlCommand(UpdateString, conn);
            SqlCommand querycmd = new
            SqlCommand(QueryString, conn);

            conn.Open();
            querycmd.Parameters.Add(new
            SqlParameter("@AlbumTitle",
                SqlDbType.NVarChar)).Value = AlbumTitle;
            SqlDataReader reader =
            querycmd.ExecuteReader();
            reader.Read();
            int AlbumID = (int)reader["AlbumID"];
            int Inventory = (int)reader["Inventory"];
            reader.Close();

            ordercmd.Parameters.Add(new
            SqlParameter("@Customer",
                SqlDbType.NVarChar)).Value = Customer;
            ordercmd.Parameters.Add(new
            SqlParameter("@AlbumID",
                SqlDbType.Int)).Value = AlbumID;
            ordercmd.Parameters.Add(new
            SqlParameter("@Number",
                SqlDbType.Int)).Value = Number;
            ordercmd.Parameters.Add(new
            SqlParameter("@Shipped",
                SqlDbType.Bit)).Value = false;

            updatecmd.Parameters.Add(new
            SqlParameter("@Inventory",
                SqlDbType.Int)).Value = ((Inventory - Number
                > 0) ? Inventory - Number : 0);
            updatecmd.Parameters.Add(new
            SqlParameter("@AlbumID",
                SqlDbType.Int)).Value = AlbumID;

            ordercmd.ExecuteNonQuery();
            updatecmd.ExecuteNonQuery();

            conn.Close();
            bResult = true;
```

```
                    }
                    catch (Exception e)
                    {
                        throw e;
                    }
                    finally
                    {
                        if (conn.State == ConnectionState.Open)
                        {
                            conn.Close();
                        }
                    }
                }
                return bResult;
            }
        }   // end of class
    }   // end of namespace
```

请仔细阅读这段代码，它隐藏了对供应商数据库的操作，仅向外界暴露逻辑上的功能。当然，这段代码没有考虑客户身份认证的问题，也没有实现对事务（Transaction）的支持。我们会在下一节介绍这些高级话题。

我们在 DBServices 类中嵌套定义一个异常类，用以标识并向外界抛出非法参数的异常。

```
public class InvalidParameterException : Exception {}
```

然后我们将于后台建立数据库连接的连接字符串、标识供应商身份的字符串和其他 SQL 查询字符串定义成私有常量。这是为了避免在程序中出现硬编码，以利于调试、修改和增强可读性。

```
private const string ConnectionString = "server=(local)\\NetSDK;
database=Supply; uid=sa; pwd=";
private const string CompanyString = "Sonier Music Co., Ltd.";

private const string DetailString = "SELECT * " +
    "FROM Albums INNER JOIN Artists " +
    "ON Albums.ArtistID=Artists.ArtistID " +
    "INNER JOIN ArtistTypes " +
    "ON Artists.ArtistTypeID=ArtistTypes.ArtistTypeID " +
    "WHERE Albums.Title=@AlbumTitle";
```

我们定义了一个返回供应商名称的 Web Service 接口方法 GetCompany()，然后再定义若干个业务相关的 Web Service 方法，包括查询所有可供应的唱片的简单目录，查询某张唱片的详细资料，查询是否供应某张唱片，查询唱片供应量是否充足，以及最后的订货接口。

注意到这段代码中对数据库的访问技巧。我们使用字符“@”来标记 SQL 查询字符串中的参数，然后使用 SqlCommand 对象的 Parameters 集合属性来对这些参数赋值，这样就不必自己构造字符串了，例如在向订单表中追加订单时，我们的命令构造是这样的：

```
SqlCommand ordercmd = new SqlCommand(OrderString, conn);
// …
ordercmd.Parameters.Add(new SqlParameter("@Customer",
                SqlDbType.NVarChar)).Value = Customer;
ordercmd.Parameters.Add(new SqlParameter("@AlbumID",
                SqlDbType.Int)).Value = AlbumID;
ordercmd.Parameters.Add(new SqlParameter("@Number",
                SqlDbType.Int)).Value = Number;
ordercmd.Parameters.Add(new SqlParameter("@Shipped",
                SqlDbType.Bit)).Value = false;
```

因为 Parameters.Add 方法返回值就是新添加的参数对象，而且我们只修改它的 Value 属性，因此直接使用…Add(…).Value = …的语法。这在 C#中是合法的。

因为 Connection 对象在退出作用域后虽可以被 CLR 的垃圾收集器释放，但其建立的与数据库的连接是不会自动释放的。因此我们利用异常机制和 Connection 对象的 State 属性来保证连接始终会被关闭。实现这样的功能的代码的大致框架如下：

```
SqlConnection conn = new SqlConnection(ConnectionString);
try
{
    // do something
}
// catch other exception
catch (Exception e)
{
    // handle the exception
}
finally
{
    if (conn.State == ConnectionState.Open) // if the Connection is
    open
    {
        conn.Close();   // Close the connetion
        }
}
```

我们将这个 Web Services 放置在本地 IIS 服务器上的 MyServices 虚拟目录下。从浏览器访问该 Web Service 的 URL 是：

http://localhost/MyServices/DBServices.asmx

9.3.3　零售商 Web Services 和网站的实现

1. 浏览者的用户体验

然后我们构建几个简单的零售商页面，来允许客户在零售商网站上订购唱片。假定我们将这些页面放置在本地 IIS 服务器的 MusicCity 虚拟目录下。

我们最前端的页面会是像图 9.7。这个页面十分简陋，因为为了减少不必要的代码量以便读者将注意力放在重要的如何和 Web Services 交互的代码上，我们几乎没有在美工上下功夫。以下的页面都是如此，除了有一个对前面提到的 CSS 技术的演示。在首页中，用户可以查询零售商所能够供应的全部或某张唱片。注意，这里的查询和零售商的供应商的 Web Services 不是简单的一一对应。为了向顾客提供附加价值，零售商可以代替顾客查询多个供应商（因为每个供应商的 Web Services 借口都可能不一样，零售商可以提供更进一步的集成这些 Web Serivces 的自己的 Web Services 来方便顾客的查询），零售商还可以通过在报价中加入自己的运营成本和预期利润来在顾客的购买过程中赚取中介利润。

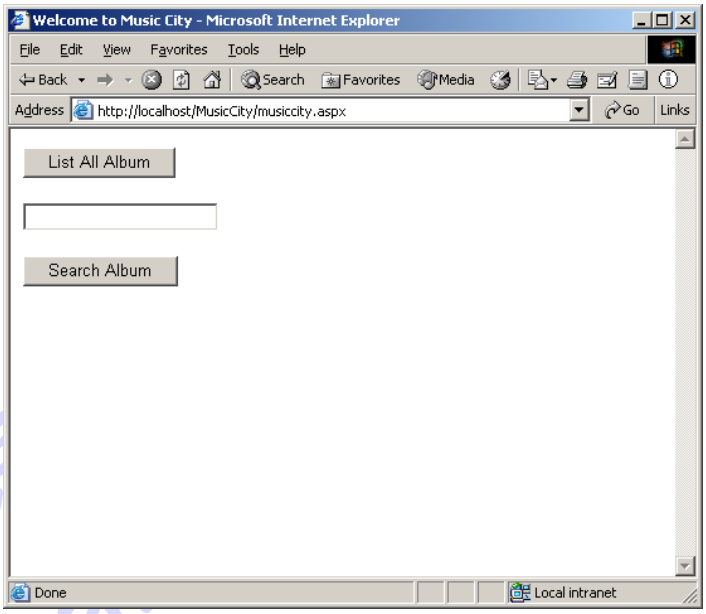


图 9.7　零售商查询页面

在进行查询后，作为顾客，我们会希望看到这样的结果页面（当然我们也希望这个页面会更漂亮），如图 9.8。这个稍微漂亮点的表格使用了 CSS 来完成。当然这和我们当前主要关心的 Web Service 是没什么相关的地方的。

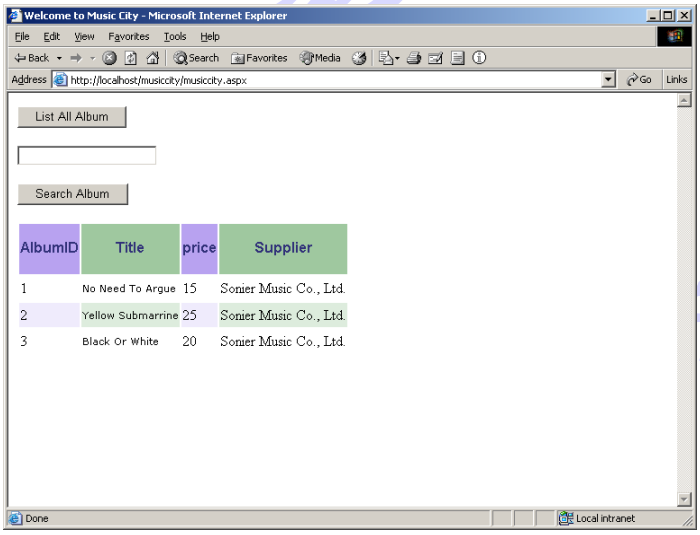


图 9.8　查询结果界面

这个表格中只简要列举了我们供货商可以提供的唱片目录。如果它太长，可以使用在 ASP 中常用的技术来分页显示，我们不举这样的例子，因为构建一个具有足够多数据的实例唱片数据库实在是很费时间而且乏味的工作。我们希望用户单击某张唱片后会出现该唱片的详细资料。例如单击“Yellow Submarrine”后会出现图 9.9 所示。

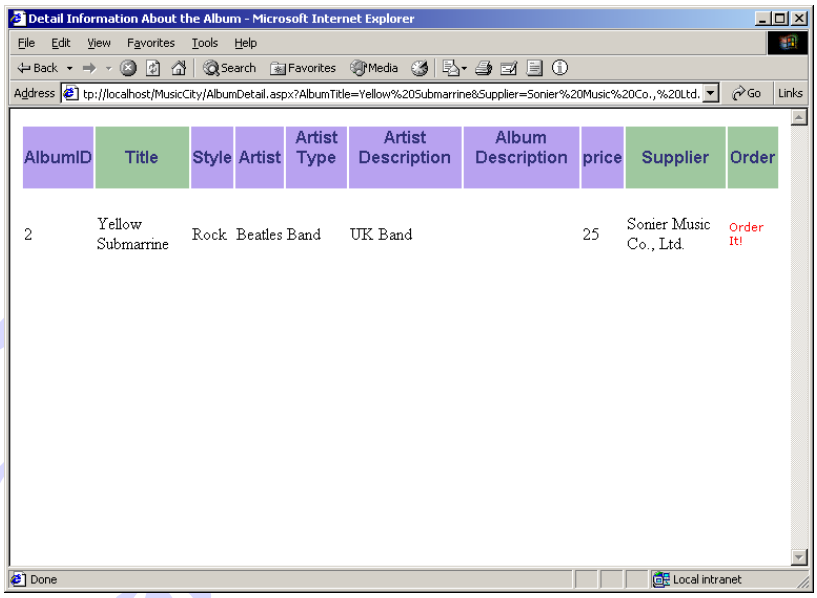


图 9.9　显示唱片详细信息

单击“Order It”之后，顾客可以填写订单并提交，如图 9.10 所示。

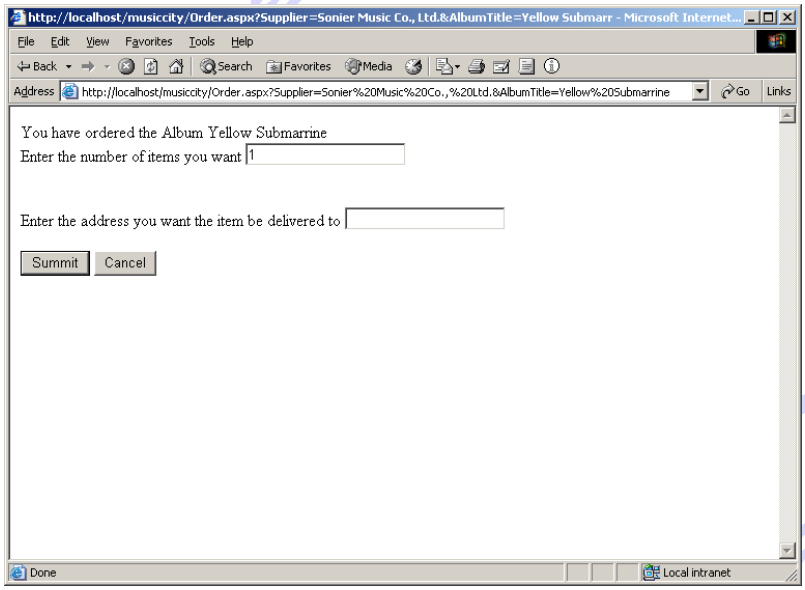


图 9.10　填写定单并且提交

为了完成这样的用户体验，我们可以先构建这些页面的 HTML 代码，并存放到 aspx 文件中，再向这些 Web Forms 添加界面逻辑代码。为了使得这些服务不仅在浏览器中可以提供，我们还必须将这些服务实现为 Web Services，以便可以开发其他的应用程序来完成同样的工作。例如以后的某天，零售商网站或者其他合作者可能需要开发一套桌面工具，客户可以在这套桌面工具中查寻我们网站提供的天气预报，并可以通过桌面工具来订购唱片。将这些服务实现为 Web Services 将可以使我们的网站提供的服务扩展到每一个角落。

这也正是 Web Services 思想的闪光之处。

因此我们的界面逻辑程序同样只须作为这些 Web Services 和它们交互，并将交互结果显示到 HTML 页面中就可以了。

2.实现零售商的 Web Services

于是第一步的工作是实现零售商网站的 Web Services，这些 Web Services 包装了对各个供应商 Web Services 的访问。我们的示范代码中只包含了和前面实现的供应商 Web Services 交互的部分。

完整的 Web Services 代码如下：

程序清单 9-7：MusicCityServices.asmx

```
<%@ WebService Language="c#" Class="Retailer.MusicCityServices" %>
using System;
using System.Data;
using System.Collections;
using System.Web.Services;
using System.Xml;

namespace Retailer
{

    [WebService(Namespace="http://localhost/MusicCity/")]
    public class MusicCityServices : System.Web.Services.WebService
    {

        private const int DefaultProfit = 2;

        // Get the profit setting from an xml document
        // the profit for each supplier is independent.
        private int GetProfit(string Company)
        {
            XmlDocument configdoc = new XmlDocument();
            try
            {
                configdoc.Load(Server.MapPath("services.xml"));
                XmlNode ProfitNode = configdoc.SelectSingleNode
                ("//Supplier[@Company='" + Company + "']");
                return Convert.ToInt32(ProfitNode.Attributes
                ["Profit"].Value);
            }
            catch (Exception exception)
            {
                return DefaultProfit;
            }
        }

        [WebMethod(Description="Get the album list that all the
        suppliers supplies")]
        public DataSet GetList()
        {

            try
            {
                // instantiate the WebService class for Supplier
                DBServices Supplier = new DBServices();
```

```
            string Company = Supplier.GetCompany();

            // get the supply list from DBServices
            DataSet ds = Supplier.GetAlbums(Company);

            // reset the price
            int ProfitPerAlbum = GetProfit(Company);
            foreach (DataTable eachtable in ds.Tables)
                foreach (DataRow eachrow in
                ds.Tables[Company].Rows)
                    eachrow["Price"] = (Decimal)eachrow
                    ["Price"] + ProfitPerAlbum;

            // Get other suppliers' supply list
            // fill them into ds with each table name as the
                supplier's identification string
            // reset the price accordingly

            // clear the profit information
            //ds.AcceptChanges();

            return ds;
        }
        catch (Exception exception)
        {
            return new DataSet();
        }
    }

    [WebMethod(Description="Search an specific album in all the
     suppliers")]
    public DataSet SearchAlbum(string AlbumTitle)
    {
        DataSet ds;
        string CompanyName;
        int ProfitPerAlbum;

        DBServices Supplier = new DBServices();
        if (Supplier.HasAlbum(AlbumTitle))
        {
            CompanyName = Supplier.GetCompany();
            ProfitPerAlbum = GetProfit(CompanyName);
            ds = Supplier.GetAlbumDetail(AlbumTitle,
            CompanyName);
            foreach (DataRow eachrow in ds.Tables
            [CompanyName].Rows)
                eachrow["Price"] = (Decimal)eachrow["Price"] +
                ProfitPerAlbum;
        }
        // else if (another supplier)
        // Search other Suppliers by invoking their Web Services
        // and merge the DataSets into a single DataSet
        // setting the name of result tables as the supplier name
        else
        {
            ds = new DataSet();
        }
```

```
            ds.AcceptChanges();
            return ds;
        }

        [WebMethod]
        public bool Order(string AlbumTitle, int Number, string
        Supplier, string Customer)
        {
            // instantiate the WebService class for Supplier
            DBServices server = new DBServices();
            // instantiate other supplier WebService class

            if (Supplier == server.GetCompany())
                return (server.Order(AlbumTitle, Number,
                Customer));
            // else if (Supplier == other suppliers' identification
                string)
            //  ...
            else
                return false;
        }
    } // end of class
} // end of namespace
```

这个 Web Service 调用了 DBServices 的服务，是 DBServices 的客户端。为了使 MusicCityServices 服务可以工作，我们首先使用 wsdl.exe 和 csc.exe 两个.NET SDK 提供的命令行工具来生成 DBServices 的代理类。请注意，我们使用缺省的命令行参数，并假设 wsdl.exe 和 csc.exe 所在路径在当前 Path 环境变量设置中：

```
wsdl.exe http://localhost/MyServices/DBServices.asmx
```

这个命令在当前目录生成代理类的 C#源程序，生成的文件名为 DBServices.cs。

然后我们使用 C#编译器编译这个类，因为我们在 Web 应用程序（在这里 Web Services 也属于 Web 应用程序的一种）中使用它，需要将它编译成 dll 文件，因此使用/target 编译开关。/target 开关可以简写成/t，冒号后紧接的可以是 exe、winexe、library 或者 module，这里我们使用 library 来生成 dll 文件：

```
csc.exe /t:library DBServices.cs
```

生成的文件名是 DBServices.dll。将它拷贝到 MusicCityServices.asmx 所在的虚拟目录下的 bin 子目录中。这里是 localhost/MusicCity/bin/下。

到此为止，这个 MusicCityServices Web Service 应该可以运行了。可以在浏览器中访问它来检验这一点。

为了在交易中获利，我们希望对成交的每一张唱片赚取一定的利润。我们用一个 XML 文档来设置和记录对每个供货商产品要追加的价格，然后实现一个私有函数来从该文件中读取这些设置。我们利用前面学习过的 XmlDocument 实现的 DOM 模型，将该文档装载入内存并使用 XPath 表达式来选取特定供应商的利润属性。这个 XML 文件内容如下：

程序清单 9-8：services.xml

```
<ServicesConfig>
    <ProfitPerAlbum>
        <Supplier Company="Sonier Music Co., Ltd." Profit="5">
        </Supplier>
    </ProfitPerAlbum>
```

```
</ServicesConfig>
```

读取出利润额的代码如下：

```
private const int DefaultProfit = 2;

private int GetProfit(string Company)
{
    XmlDocument configdoc = new XmlDocument();
    try
    {
        configdoc.Load(Server.MapPath("services.xml"));
        XmlNode ProfitNode = configdoc.SelectSingleNode
        ("//Supplier[@Company='" + Company + "']");
        return Convert.ToInt32(ProfitNode.Attributes
        ["Profit"].Value);
    }
    catch (Exception exception)
    {
        return DefaultProfit;
    }
}
```

这里又一次使用了异常机制，确保无论出现何种异常（可能的原因有 XML 文件不存在，或者文件格式不是我们 XPath 表达式中所预料的），都会返回缺省的利润额。在打开 XML 文档时，我们使用了 Server 对象的 MapPath 方法。该方法将虚拟目录下的某个文件路径映射为本地路径。Server 对象是从 WebService 类继承而来的公共属性。除了 WebService 类具有该属性外，Page 类也同样具有 Server 对象。然后我们读取属性值，使用 Convert 类的静态方法 ToInt32 来转换数据类型并返回转换结果。

使用配置文件来避免在程序中硬编码常量字符串或其他常量是一种好习惯。除了像上面的例子一样自定义一个特殊格式的配置文件外，我们还有另一种方法来从配置文件中读取程序中需要使用的常量——使用 web.config 文件。web.config 文件的具体信息在第十章中应用程序的配置部分详细介绍。该文件是一个 XML 文件，在根元素<configuration>下可以存在一个<appSettings>子元素，该元素下可以定义一个或多个<add>元素，每个<add>元素都是空元素，并具有 key 和 value 两个属性。例如我们的利润配置在 web.config 文件中的格式可以是：

```
<configuration>
    <appSettings>
        <add key=" Sonier Music Co., Ltd." value="5">
    </appSettings>
</configuration>
```

Web 应用程序和 Web Services 可以在程序代码中使用 ConfigurationSettings 对象的 AppSettings 属性来访问各个属性值。ConfigurationSettings 对象处于 System.Configuration 名字空间中，该名字空间是 Page 类自动引入的，但对于 Web Services 则需要使用 using 来引入它。AppSettings 属性是一个集合属性，返回一个 NameValueCollection 对象。可以通过<add>标记中的 key 属性来访问相应的 value 属性值。代码如下：

```
using System;
using System.Configuration;
// …
    private int GetProfit()
    {
```

```
                try
                {
                    string stringvalue = ConfigurationSettings.AppSettings
                    ["Sonier Music Co., Ltd."];
                    return (Convert.ToInt32(stringvalue));
                }
                catch (Exception exception)
                {
                    return DefaultProfit;
                }
            }
        // …
```

另外，也可以用 HttpContext 对象的静态属性 Current 来获得当前 HTTP 请求的上下文对象（HttpContext 对象），向该对象的 GetConfig 方法传递字符串“appSettings”来获得一个 NameValueCollection 对象，这个对象包含了<appSettings>元素下定义的所有属性和相应值。然后我们就可以像处理 AppSettings 属性对象一样访问需要的属性的值了。HttpContext 对象存在于 System.Collection.Specialized 名字空间中，而 HttpContext 对象存在于 System.Web 名字空间中。在 Web Services 中，我们必须显式地引入这些名字空间。由于 GetConfig 方法返回的是一个 object 类型，为了得到 NameValueCollection 类型的对象，还必须使用 C#的 unboxing（其实我们在前面的例子中已经用过）。

```
        using System;
        using System.Web;
        using System.Configuration.Specialized;
        // …
            private int GetProfit()
            {
                try
                {
                    NameValueCollection appSetting = (NameValueCollection)

        HttpContext.Current.GetConfig("appSettings");
                    string stringvalue = appSetting["Sonier Music Co.,
                    Ltd."];
                    return (Convert.ToInt32(stringvalue));
                }
                catch (Exception exception)
                {
                    return DefaultProfit;
                }
            }
        // …
```

在获取唱片列表时，零售商 Web Service 不仅要向调用者返回从供应商获取的数据，还可能需要在返回结果中增加额外的列来标示该唱片的供应商，因为不同的供应商可能会供应同一张唱片，因此我们在 GetList 方法中必须完成这样的工作。因为 DataSet 对象可以容纳多个表，利用这一点，我们可以将从不同供应商获得的供应列表存放在不同的表中，并设置表名为供应商的标识字符串。这一部分代码如下：

```
        [WebMethod(Description="Get the album list that all the suppliers
    supplies")]
        public DataSet GetList()
        {

            try
```

```
        {
            // instantiate the WebService class for Supplier
            DBServices Supplier = new DBServices();

            string Company = Supplier.GetCompany();

            // get the supply list from DBServices
            DataSet ds = Supplier.GetAlbums(Company);

            // reset the price
            int ProfitPerAlbum = GetProfit(Company);
            foreach (DataTable eachtable in ds.Tables)
                foreach (DataRow eachrow in ds.Tables[Company].Rows)
                    eachrow["Price"] = (Decimal)eachrow["Price"] +
                    ProfitPerAlbum;

            // Get other suppliers' supply list
            // fill them into ds with each table name as the supplier's
                identification string
            // reset the price accordingly

            // clear the profit information
            ds.AcceptChanges();

            return ds;
        }
        catch (Exception exception)
        {
            return new DataSet();
        }
    }
```

这段代码中只从一个供应商的 Web Services 获得供应列表，但仍然将表放置到 DataSet 中，对其他供应商供应列表的访问代码可以放置在注释所示的地方。在获得表后，我们利用前面实现的私有函数来读取对特定供应商商品追加的价格，并在每项唱片记录中迭代，逐项修改记录中的价格（“Price”）字段：

```
int ProfitPerAlbum = GetProfit(Company);
foreach (DataTable eachtable in ds.Tables)
    foreach (DataRow eachrow in ds.Tables[Company].Rows)
        eachrow["Price"] = (Decimal)eachrow["Price"] + ProfitPerAlbum;
```

正如第七章提到的，DataSet 对象会自动记录包含的数据项的变动，因此修改前的价格会存在于对象的内存空间中，并且在传输时，这些价格仍然被包含在 XML 格式的 DataSet 表示中。

提示：如果读者对这两个问题仍有疑问，请回顾第 7 章的讨论。

为了验证这一点，我们把 MusicCityServices 类的 GetList()方法中的以下行注释掉：

```
ds.AcceptChanges();
```

然后从浏览器访问 MusicCityServices Web Services。在地址栏中敲入 http://localhost/MusicCity/MusicCityServices.asmx（假设该 Web Services 存在于本地机器的 MusicCity 虚拟目录下）。在首页中选择 GetList 方法，出现图 9.11 的界面。

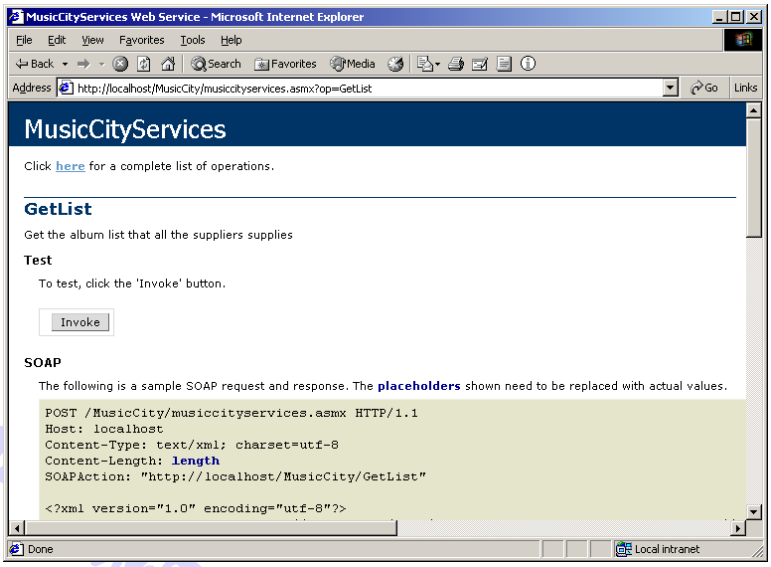


图 9.11　选择 GetList 方法后出现的界面

因为该方法没有要求参数，直接单击“Invoke”。该方法会调用 DBServices Web Service 的 GetAlbums 接口方法。返回的结果会显示在新弹出的浏览器窗口中，如图 9.12。

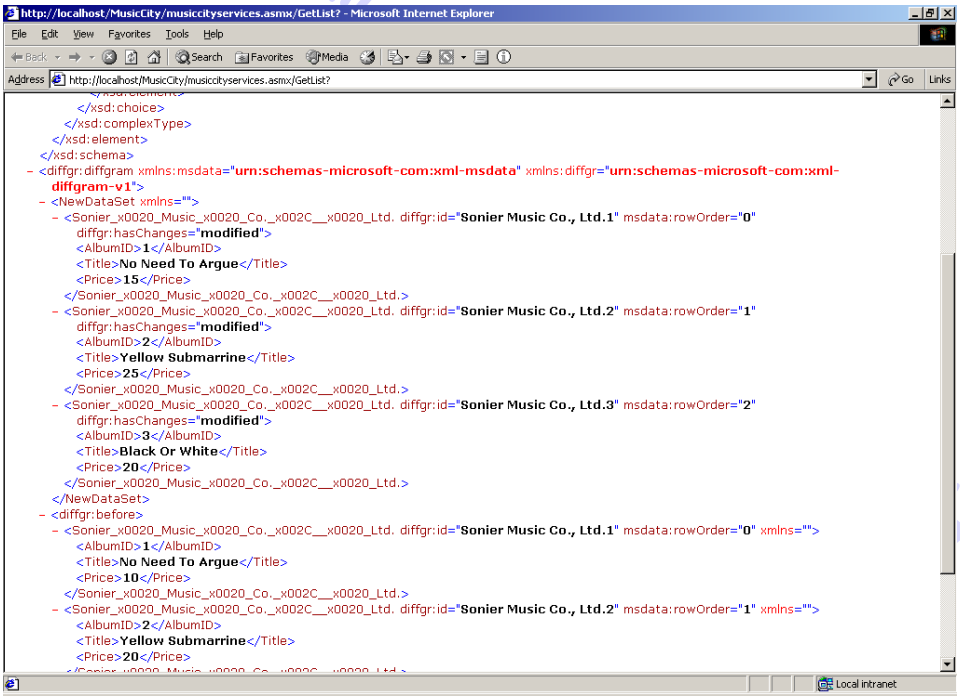


图 9.12　调用 DBServices Web Service 的 GetAlbums 接口方法后显示的文档

可以看到这是一个标准的 XML 文档，这就是返回的 DataSet 对象的 XML 表示。在根元素 DataSet 下有两个子元素：<xsd:schema>和<diffgr:diffgram>，前者是 DataSet 对象中的数据组织模式的 XSD Schema 表示，后者包括了实际的数据。向下卷动滚动条，可以清楚

地看到，<diffgr:diffgram>标记下有两个子元素：<NewDataSet>和<diffgr:before>，仔细阅读标记下的子标记数据，会发现前者中就是修改后的唱片记录，后者则是修改前的唱片记录。例如唱片 No Need To Argue 前者中的价格为 15，而后者中则是 10。

显然，我们不希望将这样的信息作为服务的返回信息透露给外界。刚才注释掉的一行代码可以实现隐藏这样的商业秘密的想法。DataSet 对象 AcceptChanges()方法的调用，可以清除 DataSet 对象中数据修改的痕迹。当然，调用过该方法后这些数据的修改就不可恢复了，并且调用该方法会触发若干个 DataSet 对象的事件。这个方法并不仅仅用在这种情形下，请读者自行查阅手册来获取其他关于 AcceptChanges 方法的信息。

解除这行代码的注释后，可以用同样的方法在浏览器中检查 GetList 方法返回的 DataSet 对象内容。供应商的供应价格将不会包含在其中。

3. 零售商网站页面对 Web Services 的调用

然后，我们会在界面逻辑中使用 GetList 接口方法，来获得零售商可以提供的唱片列表。我们构建首页的 aspx 文件如下：

程序清单 9-9：MusicCity.aspx

```
<%@ Page Language="C#" ClassName="Music"%>
<%@ Import Namespace="System.Data" %>

<script runat="server">
</script>

<html>
<head>
    <title>
        Welcome to Music City
    </title>
    <link rel="stylesheet" HREF="Album.css" TYPE="text/css">
<head>
<body>
<form runat="server">
    <asp:Button Text="List All Album" id="btnList" runat="server">
    </asp:Button>

    </p>
    <asp:Label Text="Album Title">
    </asp:Label>
    <asp:TextBox id="AlbumTitle" runat="server">
    </asp:TextBox>
    </p>
    <asp:Button Text="Search Album" id="btnSearch" runat="server">
    </asp:Button>

    </p>

   <asp:Repeater id="repAlbumList" runat="server">

    <HeaderTemplate>

        <table height="148">
            <tr>
```

```
                    <td class="BackgroundColor1" valign="bottom"
                    align="center" height="56">
                    <H4><font face="Arial" color="#313179"
                    >AlbumID</font></H4>
                    </td>

                    <td class="BackgroundColor2" valign="bottom"
                    align="center" height="56">
                        <H4><font face="Arial" color="#313179">
                        Title</font></H4>
                    </td>

                        <td class="BackgroundColor1" valign="bottom"
                        align="center" height="56">
                    <H4><font face="Arial" color="#313179"
                    >price</font></H4>
                    </td>

                        <td class="BackgroundColor2" valign="bottom"
                        align="center" height="56">
                        <H4><font face="Arial" color="#313179"
                        >Supplier</font></H4>
                        </td>

                    </tr>
            </HeaderTemplate>

            <ItemTemplate>
                <tr>
                    <td>
                    <%# DataBinder.Eval(Container.DataItem,
                    "AlbumID") %>
                </td>

                <td>
                    <a href='AlbumDetail.aspx?AlbumTitle=<%#
                    DataBinder.Eval(Container.DataItem, "Title")
                    %>&Supplier=<%# DataBinder.Eval
                    (Container.DataItem, "Supplier")%>'>
                        <%# DataBinder.Eval(Container.DataItem
                        ,"Title") %>
                        <br>
                  </a>
                </td>

                    <td>
                    <%# DataBinder.Eval(Container.DataItem,"Price") %>
                    </td>

                    <td>
                    <%# DataBinder.Eval(Container.DataItem,"Supplier") %>
                    </td>
                </tr>
            </ItemTemplate>

            <AlternatingItemTemplate>
                <tr>
                    <td class="BackgroundColor6">
```

```
                        <%# DataBinder.Eval(Container.DataItem,"AlbumID") %>
                        </td>

                        <td class="BackgroundColor5">
                    <a href='AlbumDetail.aspx?AlbumTitle=<%# DataBinder
                    .Eval(Container.DataItem,"Title")%>&Supplier=<%#
                    DataBinder.Eval(Container.DataItem, "Supplier")%>'>
                        <%# DataBinder.Eval(Container.DataItem,"Title") %>
                        <br>
                        </a>
                        </td>

                        <td class="BackgroundColor6">
                        <%# DataBinder.Eval(Container.DataItem,"Price") %>
                        </td>

                        <td class="BackgroundColor5">
                        <%# DataBinder.Eval(Container.DataItem,"Supplier") %>
                        </td>
                    </tr>
                </AlternatingItemTemplate>

                <FooterTemplate>
                    </table>
                </FooterTemplate>
            </asp:Repeater>

        </form>
        </body>
        <html>
```

这个页面使用了 Repeater 控件和数据绑定技术，使用模板和 CSS 类来定义列的输出样式。关于这两种技术，请参考本书前面几章的讨论。

我们现在编写按钮 btnList 的单击事件处理函数。首先在按钮的标记中增加 OnClick 属性如下：

```
        <asp:Button Text="List All Album" id="btnList" runat="server"
    OnClick="btnList_Click">
        </asp:Button>
```

然后在<script>标记中编写 btnList_Click 函数：

```
        void btnList_Click(object Sender, EventArgs e)
        {
            try
            {
                MusicCityServices server = new MusicCityServices();

                DataSet ds = server.GetList();
                DataTable table = new DataTable();
                table.Columns.Add("Supplier");

                foreach (DataTable eachtable in ds.Tables)
                {
                    foreach (DataColumn eachcolumn in eachtable.Columns)
                    {
                        if (!table.Columns.Contains
                        (eachcolumn.ColumnName))
                            table.Columns.Add(eachcolumn.ColumnName,
                            eachcolumn.DataType);
```

```
                    }
                    table.Columns["Supplier"].DefaultValue =
                    eachtable.TableName;

                    foreach (DataRow eachrow in eachtable.Rows)
                    {
                        DataRow newrow = table.NewRow();
                        foreach (DataColumn eachfield in eachtable.Columns)
                        {
                            newrow[eachfield.ColumnName] = eachrow
                            [eachfield.ColumnName];
                        }
                        table.Rows.Add(newrow);
                    }
                }

                repAlbumList.DataSource = table.DefaultView;
                repAlbumList.DataBind();
            }
            catch (Exception exception)
            {
                Response.Write(exception.ToString());
            }
        }
```

在这个函数中，首先利用代理类调用 MusicCityServices 服务的 GetList 方法，获得一个 DataSet 对象，如前面所述，该对象中每个表储存了每个供应商的供应列表，表名为供应商名称。因此我们要先建立一个表对象作为存放所有供应唱片列表的表，这个表除了包含各个供应商产品列表的列和数据外，还应增加一个“供应商”字段，来表示每一条唱片记录的供应者。这个工作由下面的代码完成：

```
DataTable table = new DataTable();
table.Columns.Add("Supplier");
```

接着我们在表集合中迭代每一个表，每次迭代，我们首先检查供应商产品列表中是否有结果表中的列，并将新出现的列添加到结果表中。这样我们的代码将具有更强的可扩展性，因为我们在界面逻辑中无法预计也不应该假设会有几个和哪些供应商提供产品（这个工作应该由后台的 Web Services 来完成），因此各个供应商返回的表结构可能是千差万别的。通过动态添加列，可以将代码对表结构的依赖减小到更低的程度。尽管这里只有一个供应商，我们仍然热衷于编写需要更少的修改、更易于扩展的代码。

接下来，我们设置表中的新列“Supplier”的缺省值为表名（也即供应商名称）。

```
table.Columns["Supplier"].DefaultValue = eachtable.TableName;
```

设置缺省值后，表中新加入的行相应字段如果没有给出，则表对象会用该字段的缺省值来填充它，注意设置的缺省值只对设置后加入或修改的行起作用，因此不必担心对第二个供应商的该字段设置会影响已加入结果表的第一个供应商的数据。

最后我们迭代供应商表中每一条数据记录，将记录添加到结果表中。每次迭代，我们先建立一个新行，然后为新的行每一个字段赋值。注意这里迭代的是供应商表中的列，而不是结果表中的列，因为结果表中的列有可能是某个供应商表特有而加入的。如果按结果表的列来迭代，访问供应商表记录时就可能访问到不存在的列了。

```
foreach (DataRow eachrow in eachtable.Rows)
{
```

```
            DataRow newrow = table.NewRow();
            foreach (DataColumn eachfield in eachtable.Columns)
            {
                newrow[eachfield.ColumnName] = eachrow
                [eachfield.ColumnName];
            }
            table.Rows.Add(newrow);
        }
```

最后，我们的 btnList_Click 函数会将结果表的缺省视图作为 Repeater 控件的数据源，将 Repeater 控件绑定到这个结果表上：

```
        repAlbumList.DataSource = table.DefaultView;
        repAlbumList.DataBind();
```

这样就完成了对显示所有唱片的功能的实现。可以看到，对后台 Web Service 的调用是相当简单的，我们绝大多数的代码是在前台处理获得的数据。

另外我们在第一页还可以实现搜索唱片的功能。搜索列表仍然由 Repeater 控件按一定样式给出。这个功能的代码和上面完成的列表代码很相近，同样是从后台的 Web Service 获得一个 DataSet 对象，该对象中每个表包含每个可以提供该唱片的供应商的搜索结果。获得该 DataSet 对象后，同样是将各个表中的数据合并到一个表中，并添加一个列来表示供应商。主要的不同在于调用的 Web Service 而已。我们不再详细分析这些代码，请读者自己研究。

```
        <asp:Button Text="Search Album" id="btnSearch" runat="server"
    OnClick="btnSearch_Click">
        </asp:Button>

        void btnSearch_Click(object Sender, EventArgs e)
        {
            try
            {
                MusicCityServices server = new MusicCityServices();
                DataSet ds = server.SearchAlbum(AlbumTitle.Text);
                DataTable table = new DataTable("Results");
                table.Columns.Add("Supplier");

                // merge each table.
                // a single table contains search result from a single supplier
                foreach (DataTable eachtable in ds.Tables)
                {
                    // prepare
                    foreach (DataColumn eachcolumn in eachtable.Columns)
                    {
                        if (!table.Columns.Contains
                        (eachcolumn.ColumnName))
                            table.Columns.Add(eachcolumn.ColumnName,
                            eachcolumn.DataType);
                    }
                    table.Columns["Supplier"].DefaultValue =
                    eachtable.TableName;

                    foreach (DataRow eachrow in eachtable.Rows)
                    {
                        // create a new row
                        DataRow newrow = table.NewRow();
                        // fill the data
```

```
                        foreach (DataColumn eachfield in eachtable.Columns)
                        {
                            newrow[eachfield.ColumnName] =
                            eachrow[eachfield.ColumnName];
                        }
                        // add the new row into the table
                        table.Rows.Add(newrow);
                    }
                }
                repAlbumList.DataSource = table.DefaultView;
                repAlbumList.DataBind();
            }
            catch (Exception exception)
            {
                Response.Write(exception.ToString());
            }
        }
```

第一页只使用 Repeater 控件显示唱片的简单信息，为了让用户获得某张唱片的详细信息，我们在 ItemTemplate 中使用了数据绑定来构造特殊的 URL。这种技术在 ASP 时代就已经广泛使用，目标页面使用 Request 方法来从 URL 提取出参数，这是 HTTP-GET 方法的请求方式。

```
        <a href='AlbumDetail.aspx?AlbumTitle=<%#
    DataBinder.Eval(Container.DataItem,"Title") %>&Supplier=<%#
    DataBinder.Eval(Container.DataItem, "Supplier") %>'>
        <%# DataBinder.Eval(Container.DataItem,"Title") %>
        <br>
        </a>
```

然后我们构造 AlbumDetail.aspx 页面：

程序清单 9-10：AlbumDetail.aspx

```
        <%@ Page Language="C#" %>
        <%@ Import Namespace="System.Data" %>

        <script runat="server">
            void Page_Load(Object Sender, EventArgs e)
            {
                MusicCityServices server = new MusicCityServices();
                DataSet ds = server.SearchAlbum(Request["AlbumTitle"]);
                repAlbumList.DataSource = ds.Tables[Request["Supplier"]]
                .DefaultView;
                repAlbumList.DataBind();
            }
        </script>
        <html>
        <head>
            <title>
            Detail Information About the Album
            </title>
            <link rel="stylesheet" HREF="Album.css" TYPE="text/css">
        </head>
        <body>
        <form runat="server">
            <asp:Repeater id="repAlbumList" runat="server">
                <HeaderTemplate>
                <table height="148">
                    <tr>
```

```
    <td class="BackgroundColor1" valign="bottom" align
    ="center" height="56">
<H4><font face="Arial" color="#313179">AlbumID
</font></H4>
</td>

<td class="BackgroundColor2" valign="bottom" align
="center" height="56">
    <H4><font face="Arial" color="#313179">Title
    </font></H4>
</td>

<td class="BackgroundColor1" valign="bottom" align
="center" height="56">
    <H4><font face="Arial" color="#313179">Style
    </font></H4>
</td>

<td class="BackgroundColor1" valign="bottom" align
="center" height="56">
    <H4><font face="Arial" color="#313179">Artist
    </font></H4>
</td>

<td class="BackgroundColor1" valign="bottom" align
="center" height="56">
    <H4><font face="Arial" color="#313179">Artist Type
    </font></H4>
</td>

<td class="BackgroundColor1" valign="bottom" align
="center" height="56">
    <H4><font face="Arial" color="#313179">Artist
    Description</font></H4>
</td>

<td class="BackgroundColor1" valign="bottom" align
="center" height="56">
    <H4><font face="Arial" color="#313179">Album
    Description</font></H4>
</td>

    <td class="BackgroundColor1" valign="bottom" align
    ="center" height="56">
    <H4><font face="Arial" color="#313179">price
    </font></H4>
</td>

    <td class="BackgroundColor2" valign="bottom" align
    ="center" height="56">
    <H4><font face="Arial" color="#313179">Supplier
    </font></H4>
    </td>

    <td class="BackgroundColor2" valign="bottom" align
    ="center" height="56">
    <H4><font face="Arial" color="#313179">Order
    </font></H4>
```

```
                    </td>
                </tr>
            </HeaderTemplate>

            <ItemTemplate>
                <tr>
                    <td>
                    <%# DataBinder.Eval(Container.DataItem,"AlbumID")
                    %>
                </td>

                <td>
                <%# DataBinder.Eval(Container.DataItem,"Title") %>
                </td>

                    <td>
                    <%# DataBinder.Eval(Container.DataItem,"Style") %>
                    </td>

                    <td>
                    <%# DataBinder.Eval(Container.DataItem,"Artist")
                    %>
                </td>

                    <td>
                    <%#
DataBinder.Eval(Container.DataItem,"TypeDescription") %>
                </td>

                    <td>
                    <%#
DataBinder.Eval(Container.DataItem,"ArtistDescription") %>
                </td>

                    <td>
                    <%#
DataBinder.Eval(Container.DataItem,"AlbumDescription") %>
                </td>

                    <td>
                    <%# DataBinder.Eval(Container.DataItem,"Price") %>
                    </td>

                    <td>
                    <%=Request["Supplier"] %>
                </td>

                <td>
                <a
href='Order.aspx?Supplier=<%=Request["Supplier"]%>&AlbumTitle=<%#
DataBinder.Eval(Container.DataItem,"Title") %>'>
                <font color="red">Order It!</font>
                </a>
                </td>
                </tr>
            </ItemTemplate>

            <AlternatingItemTemplate>
                <tr>
```

```
                    <td class="BackgroundColor6">
                    <%# DataBinder.Eval(Container.DataItem,"AlbumID")
                    %>
                </td>

                <td class="BackgroundColor5">
                <%# DataBinder.Eval(Container.DataItem,"Title") %>
                </td>

                    <td class="BackgroundColor6">
                    <%# DataBinder.Eval(Container.DataItem,"Style") %>
                    </td>

                    <td class="BackgroundColor5">
                    <%# DataBinder.Eval(Container.DataItem,"Artist")
                    %>
                </td>

                    <td class="BackgroundColor6">
                    <%#
DataBinder.Eval(Container.DataItem,"TypeDescription") %>
                    </td>

                    <td class="BackgroundColor5">
                    <%#
DataBinder.Eval(Container.DataItem,"ArtistDescription") %>
                </td>

                    <td class="BackgroundColor6">
                    <%#
DataBinder.Eval(Container.DataItem,"AlbumDescription") %>
                </td>

                    <td>
                    <%# DataBinder.Eval(Container.DataItem,"Price") %>
                    </td>

                    <td class="BackgroundColor5">
                    <%=Request["Supplier"] %>
                </td>

                <td class="BackgroundColor6">
                <a
href='Order.aspx?Supplier=<%=Request["Supplier"]%>&AlbumTitle=<%#
DataBinder.Eval(Container.DataItem,"Title") %>'>
                <font color="red">Order It!</font>
                </a>
                </td>
                </tr>
            </AlternatingItemTemplate>

            <FooterTemplate>
                </table>
            </FooterTemplate>
        </asp:Repeater>

    </form>
    </body>
    <html>
```

这个页面的界面逻辑代码很少，因为用户选择的唱片只可能有一张，因此其供应商也是确定的，因此在上一个页面中将供应商的名称加入到构造的跳转链接中。在这个页面加载时，用 Request 对象提取出唱片标题并搜索所有供应商供应列表，然后直接使用 Request 对象提取出来的供应商名称从返回的 DataSet 对象中找到结果表并在 Repeater 控件中显示出来。

为了让用户订购唱片，页面上还必须有个链接来允许用户跳转到订购的页面上。我们仍然用前一个页面使用过的方法来向订购页面传递参数。

```
<a
href='Order.aspx?Supplier=<%=Request["Supplier"]%>&AlbumTitle=<%#
DataBinder.Eval(Container.DataItem,"Title") %>'>
```

对于订购页面 Order.aspx，其代码对读者来说相信已经不再神秘：实例化一个后台 MusicCityServices Web Service 类对象（当然，实际上是一个代理类对象，即便该 Web Service 就处在同一服务器上的同一目录下也是如此），然后调用该对象的 Order 方法并根据用户输入传递参数。

程序清单 9-11：Order.aspx

```
<%@ Page Language="C#" %>
<script runat="server">
    void Page_Load(Object Sender, EventArgs e)
    {
        if (!IsPostBack)
        {
            Information.Text = "You have ordered the Album " +
                Request.QueryString["AlbumTitle"];
        }
    }
    void btnSummit_Click(Object Sender, EventArgs e)
    {
        try
        {
            string AlbumTitle = Request.QueryString["AlbumTitle"];
            string Supplier = Request.QueryString["Supplier"];
            int Number = Convert.ToInt32(OrderNumber.Text);
            string Customer = Costumer.Text;
            MusicCityServices server = new MusicCityServices();
            if (server.Order(AlbumTitle, Number, Supplier,
            Customer))
                Information.Text = "Ordered successfully";
            else
                Information.Text = "Some error occurs.  Check you
                order and try again.";
        }
        catch (Exception exception)
        {
            Response.Write(exception.ToString());
            Information.Text = "Some error occurs.  Check you order
            and try again.";
        }
    }
</script>
<html>
<form runat="server">
    <asp:Label id="Information" runat="server" />
```

```
        <br>
        Enter the number of items you want
        <asp:TextBox id="OrderNumber" runat="server" Text="1" />
        </p>
        <br>
        Enter the address you want the item be delivered to
        <asp:TextBox id="Costumer" runat="server" />
        </p>
        <asp:Button id="Summit" runat="server" Text="Summit"
        OnClick="btnSummit_Click" />
    </form>
    </html>
```

9.4　ASP.NET Web Service 的高级话题

9.4.1　状态管理

Web Servie 是基于标准的 HTTP 等 Web 标准协议的，由于这些协议本身是无连接的，因此同 Web Forms 一样，Web Services 也是无状态的（Stateless）。也就是说，同 Web Services 之间的每一次对话，都是重新建立起来的请求（可能是 HTTP 请求，也可能是 SOAP 请求，等等。而后者同样是基于 HTTP 等无连接协议的）。然而，在应用环境中，开发人员往往需要开发与状态有关的 Web Services。例如，开发人员非常可能将在电子商务网站中经常使用的购物车设计移植到 Web Services 上来。一种自然的思路是，将需要保留的状态数据封装在数据包中（例如 SOAP 包），然后在 Web Services 服务器和 Web Services 客户端之间来回传输时将这些数据作为传输数据的一部分。很明显，当这些数据很大时，将它们放在 Web Services 所在的服务器上会比将这些数据封装在 SOAP 包或 HTTP 包中来回传递更为现实一点。事实上，在设计 Web Services 接口方法时，我们建议尽量利用 ASP.NET Web Services 对按引用传递特性，将相关的数据放在一个接口方法中，这样调用一次接口方法就可获得相关的一系列数据，这是因为每一次 Web Service 调用的开销都可能很大。

ASP.NET 已经为 Web Services 提供了类似于 Web Forms 技术的状态管理机制。说到“类似于 Web Forms”，读者可能已经联想到 Application 和 Session 对象了。是的，开发人员可以同样在 Web Services 中使用这两个对象来管理状态。为了做到这一点，开发人员必须使自己的 Web Service 类从.NET Class Library 中的 WebService 类继承而来，成为其子类。这是因为 Application 对象和 Session 对象是 WebService 类的成员。WebService 类处于 System.Web.Services 名字空间中。

与 Web Forms 技术中的 Page 类成员 Application 对象一样，WebService 中的 Application 对象用来存放整个应用程序范围的状态数据，例如 WebService 被引用的次数。WebService 类中的 Session 对象用来存放那些与单个对话有关的状态数据，例如某个客户的购物车状态。

和 Web Forms 技术一样，Web Services 也是通过两种方法实现 Session 支持的。当客户端支持 cookies 时，Session 对象将使用 cookies 来标识不同客户；否则，Web Services 会自动的修改所有 URL，使之包含一个特殊的字符串，服务器通过解析出该字符串来分辨不同

的会话。

因为这个原因，缺省情况下，对 Session 对象的支持是被关闭的，为了在接口方法中访问 Session 对象，不仅要使 Web Service 类从 System.Web.Services 名字空间中的 WebService 类继承而来，还必须在 Web Method 的声明中将 EnableSession 属性赋值为 true：

```
[WebMethod(EnableSession=true)]
public void SomeMethod()
{
    // …
}
```

WebService 派生类的 Web Method 可以访问 Application 对象，而不必作任何工作。不作为接口方法的成员函数也可以访问 Application，这是因为 Application 对象存储的是整个应用程序域的状态，服务器可以惟一确定它；而要访问 Session 对象，需要调用这些成员函数的接口方法是可以访问 Session 对象的。访问 Session 对象的代码存在于普通成员函数中并不会引起编译时错误。但当 EnableSession 属性不为 true 的 Web Method 调用它们时，则会有运行时错误。

可以使用下面的代码来验证上面的说法。注意这些代码没有在 Web Farms 环境或 Web Garden 环境中测试过，截至到本书撰稿时为止，Beta 2 SDK 文档中也没有关于 Web Services 在这两种环境中的状态管理的额外说明。

程序清单 9-12：StateManagementSample.asmx

```
<%@ WebService Language="C#" Class="StateManagement.Sample" %>

using System.Web.Services;

namespace StateManagement
{
    public class Sample : WebService
    {

        [WebMethod(EnableSession=true)]
        public string CanAccessSession()
        {
            return AccessSession();
        }

        [WebMethod]
        public string CannotAccessSession()
        {
            return AccessSession();
        }

        [WebMethod]
        public string CanAccessApplication()
        {
            return AccessApplication();
        }

        private string AccessApplication()
        {
            Application["test"] = "Application Enabled";
            return (string)Application["test"];
```

```
        }

        private string AccessSession()
        {
            Session["test"] = "Session Enabled";
            return (string)Session["test"];
        }

    }   // end of class
}   // end of namespace
```

可以在浏览器中访问这个 Web Service 来测试各个接口方法能否工作。私有成员函数 AccessSession()包含了对 Session 对象的访问代码，但如前所述，这不会引发编译时错误，因此 Web Service 缺省生成的 Helper 页面和 CannotAccessSession 接口方法的 Helper 页面都是可以正常显示的。但当在 CannotAccessSession 接口方法的 Helper 页面中单击 Invoke 按钮后，IE 浏览器会显示“The page cannot be displayed”，因为 CannotAccessSession 方法不具有 EnableSession=true 属性。使用 Invoke 按钮调用其他两个接口方法则不会引发这个错误，包括 CanAccessSession 方法，因为它具有 EnableSession=true 属性。

9.4.2　ASP.NET Web Services 支持的数据类型

所有的程序设计语言都有特定的数据类型。各种程序设计语言中的简单的基础类型都是极为相似。然而不同语言间的数据类型并无法简单地相互映射起来，因为这些类型从细节上来说似乎不相同的。作为一个 Web Service，它可能被任意类型的客户端程序访问，因此必须为 Web Services 提供一套统一的、具有广泛意义的数据类型。

为了做到这一点，ASP.NET Web Services 使用了 XML Schema Definition 语言草案中定义的数据类型，因为这些数据类型可以被 XML 编码和序列化。这些数据类型和 C++语言及.NET CLR 中类型的对应关系如表 9-7。

我们已经在 9.3 节的例子中看到，在 Web Services 请求过程中可以传递参数，我们使用过的参数类型从简单的 string 和 int 到复杂的 DataSet 对象都有。然而，正如你猜到的，尽管 Web Services 技术已经提供了极为丰富的数据类型，并不是所有类型的数据都可以在这个过程中传递的。在使用 SOAP 协议传输时，Web Service 接口方法可以支持下列的数据类型：

表 9-7　Web Service 接口方法可以支持的数据类型

数据类型	说明
基本类型	包括 String, Char, Byte, Boolean, Int16, Int32, Int64, UInt16, UInt32, UInt64, Single, Double, Guid, Decimal, DateTime（XML 中的 timeInstant）, DataTime（XML 中的 date）, DateTime（XML 中的 time）和 XmlQualifiedName（XML 中的 Qname）
Enum	枚举类型
class 和 struct	带有公共 field 和 properties 成员的类和结构体。其中类和结构体的公共 field 和 properties 成员会被序列化

（续表）

数据类型	说明
DataSet	DataSet 对象。DataSet 对象同样可以作为类或结构体的 field 成员
XmlNode	XmlNode 对象。同样可以作为类或结构体的 field 成员
上述类型构成的数组	例如 String[]，DataSet[]，XmlNode[]等等

无论使用 SOAP 协议还是 HTTP-POST 或 HTTP-GET 协议传输时，Web Services 都支持上表中的所有类型作为接口方法的返回值。

对于方法参数，情况比较复杂。SOAP 协议支持上述所有类型作为接口方法参数类型，并且同时支持按值传输（by value）和按引用传输（by reference），按引用传输的参数既可以是上传到服务器，也可以是从服务器回传到客户。当使用 HTTP-POST 或 HTTP-GET 协议时，按引用传输是不支持的，而按值传输时，支持的参数类型也是有限的，只有大部分的基本类型（包括 Int32，String，Int16，Int64，Boolean，Single，Double，Decimal，DateTime，UInt16，UInt32，UInt64 and Currency）和枚举类型，以及这些被支持的类型的数组。从客户端的角度看，这些基本类型会全部转换成字符串；而枚举类型则会被转换成一个类，而其中的枚举值被转换成静态常量字符串成员。

9.4.3　对事务的支持

在商业逻辑中，事务支持是一个必不可少的特性。

所谓事务是指一些相关的任务序列，序列中的任务或者全部成功完成，或者全部不完成。例如，当你购买一张唱片时，你会向商家付款，商家则会向你提交唱片，这两个任务要么全部完成，要么全部不做。因为无论是你付款后没有收到合乎购买协议（哪一张唱片、购买的数量、交货日期……都是协议的内容）的唱片，还是你拿到唱片后商家发现你付的是假钞或者信用卡发行者拒绝支付，都会有一方不高兴而引起麻烦。

技术上讲，数据库产品对事务的支持已经是一个必不可少的特性。一个简单的例子是，你接到客户订单后，会希望向订单表中添加订单记录，并修改库存表中的库存数据。这两个数据库任务必须被视为一个操作，如果在添加订单记录后数据库服务器刚好崩溃（例如有个家伙踢到了电源线），则在数据库恢复时程序必须能够决定要么继续下一个操作（修改库存），要么将订单表恢复到添加前的状态。否则，很容易预见到，你的数据将不具有一致性。事务技术确保同一事物中的面向数据的资源不会被永久性地改变，直到事务中的所有操作都成功完成。

对事务技术的支持有两种模式：手动模式和自动模式。在手动模式中，由程序员显式地决定何时开始一个事务，控制事务边界中的各种资源，决定事务的结果是确认还是取消，并显式地中止事务。Microsoft ActiveX Data Objects（ADO）、OLE DB、ODBC 和 Microsoft Message Queue（MSMQ）资源 API 支持手动模式的事务操作。在自动模式中，程序员使用标记和属性来将对象标记为参与事务操作，并使用属性的值来控制对象在事务中的行为。对象可以被标记声明为一个新事务的起点，或者标记为参与到某个以存在的或正在进行中的事务中去，或者标记为从不参与到一个事务中。Microsoft Transaction Server（MTS）、COM+ 1.0 和.NET CLR 支持相同的自动的分布式事务模型。

.NET 支持两种事务处理模式。在自动模式中，可以通过标记方式来决定事务行为的对象有 ASP.NET Page 对象、Web Service 接口方法（Web Service Method）和.NET Framework class。当这些事务对象访问数据资源时，与对象关联的事务会自动移交给相应的资源管理者。例如当你的 Page 对象被声明为处理事务后，访问 OLE DB 数据库时，OLE DB 会查询对象上下文环境中的事务并将该事务移交给 Distributed Transaction Coordinator（DTC）。这些物理上的事务操作都是自动发生的。

ASP.NET Web Services 允许程序员控制代码使之运行在一个自动事务的范围内。为了使 Web Services 支持自动事务，可以设置 WebMethod 的 TransactionOption 属性。注意 Web Services 对事务的支持是以 WebMethod 为单位的，一个 WebMethod 只可能以事务的起点参与到事务中。对于一个作为事务起点的 Web Service 接口方法，客户调用该方法后这个方法对数据资源的访问将会遵循事务在关键任务应用程序中的基本角色——ACID，即原子性（Atomicity）、一致性（Consistency）、隔离（Isolation）和可靠性（Durability）。

为什么接口方法只能是一个事务的起点呢？本质上来说，这是因为 Web Services 并不是面向连接的。我们已经知道，Web Services 基于松散耦合的 Web 标准协议，对 Web Services 方法的调用实质是向 Web 服务器发送一个 HTTP 请求，该请求可能是一个 SOAP 协议消息包，也可能使用 HTTP-GET/POST。而 HTTP 请求是无连接的。形象地说，作为 Web 服务器，它并不能从本质上确定一个 HTTP 请求是否是另一个 HTTP 请求的后继请求，尽管我们在比 HTTP 更高的层面上可以做到这一点（例如构造特殊的 URL 并在其中加入标识字符串，或者将标识字符串包含在 HTTP 报文中），HTTP 本身是无法实现这个目标的。因此，运行 Web Services 的服务器并不具有识别不同次的 Web Method 调用的调用者的本质能力。如果一个 Web Method 作为事务的一部分而可以不是事务的起点，那么当调用者请求这个 Web Method 时，服务器如何将它和已存在的事务关联起来呢？举个例子，假设接口方法可以不作为事务的起点而包含在某个事务中时，在由于某个调用者通过某种方式使该事务开始后，另一个调用者调用了这个方法时，服务器如何确定不应该将后者引发的数据资源访问代码包含在前一个调用者引发的事务空间中呢？本质上说，不能！作为现代分布式应用程序健壮性的基础的事务技术并不能容忍 HTTP 的无连接性带来的这种缺陷。

将 Web Method 当作事务的起点，意味着当调用者调用该接口方法时，便会引起一个新的事务。这个事务是与调用者对应的，因此当不同的调用者请求同一接口方法时，他们引发的是不同的事务，Web 服务器和相应的事务管理器（例如 SQL Server、MSMQ）会确保这些事务的独立。同一个 Web Method 对象的代码处于同一事务中，因此一个 Web Method 对象就是事务的原子。

当一个 Web Method 成为一个事务原子后，运行于其中的访问数据资源的代码必须全部成功完成任务，如果任一个环节出了错误，该事务便会失败，其结果是对数据资源的所有变动都会回退到事务开始前的状态（称为 Rollback），Web Method 则会抛出一个异常来通知调用者事务的失败。

为了使一个接口方法成为一个新的事务的起点，必须为其 WebMethod 对象属性的 TransactionOption 属性赋上 RequiresNew 值。RequeiresNew 值是 TransactionOption 枚举类型的一个成员值，TransactionOption 枚举类型的其他成员值如表 9-8。

表 9-8　TransactionOption 枚举类型的其他成员

成员名	说明
Disabled	忽略运行环境上下文中的任何事务
NotSupported	将当前组件构建到没有事务的上下文环境中
Required	如果运行上下文中存在一个事务的话，将当前组件构建到该事务中，否则创建一个新事务并使组件成为新事务的起点
RequiresNew	为组件创建一个新事务，新事务的起点为该组件，忽略运行环境上下文中的其他事务的状态
Supported	如果运行环境上下文存在一个事务的话，将当前组件构建到该事务中

所有.NET 中支持事务的组件（包括 Page 组件、WebMethod 组件和.NET Framework class 组件三种）都使用这个枚举类型来控制组件的事务行为。对于 Web Services 来说，因为每个接口方法（WebMethod 组件对象）都只能是新事务的起点，因此只有 RequiresNew 这个成员值可以应用到 Web Services 中。其他的枚举值可以在其他支持事务的组件中使用，本书不会再讨论它们。

为 Web Services 接口方法添加事务支持时，只须在 WebMethod 声明中加入 TransactionOption 属性。由于 TransactionOption 枚举类型存在于 System.EnterpriseServices 名字空间中，必须在 Web Services 的 asmx 文件中用@Assembly 指令加入对 System.EnterpriseServices 的引用，以便在编译时连接 System.EnterpriseServices.dll 文件。并在源文件中引入 System.EnterpriseServices 名字空间：

例如，在 sample.asmx 中加入：

```
<%@ Assembly Name="System.EnterpriseServices" %>
```

在 sample.asmx 中或 sample.asmx 的 Code-Behind 代码中加入：

```
using System.EnterpriseServices

// …

[WebMethod(TransactionOption=TransactionOption.RequiresNew)]
public void SomeMethod()
{
    // …
}
```

于是，我们可以修改原有的 DBServices Web Services，使之支持事务，从而确保数据的安全性。由于只有 Order 接口方法会对后台数据资源（在这里是 SQL Server 数据库）做出改变，因此只需为 Order 方法定义其 TransactionOption 属性：

```
// …
[WebMethod(Description="Order a specific Album",
           TransactionOption=TransactionOption.RequiresNew)]
public bool Order(string AlbumTitle, int Number, string Customer)
{
    // …
}
```

注意到，使用标记来定义 WebMethdo 的多个属性时，可以使用逗号“,”来分隔各个配对的“属性＝值”字符串，并且可以分成多行。

这样，在调用 Order 方法后，代码运行中如果出现任何意外情况，例如服务器崩溃，则数据库的数据将会退回调用前状态，调用者则会接到一个异常。

9.4.4　使用定制的 SOAP

在前面我们看到，ASP.NET Web Services 技术缺省使用 SOAP 协议来传输远程调用信息。SOAP 协议可以支持比 HTTP-GET/POST 多得多的数据类型作为参数和方法返回值，因此更为灵活。

当使用 SOAP 协议调用远程的 Web Services 接口方法时，传输的信息遵循一个标准的格式。这些信息用一个通过编码存在于 HTTP 报文中的 XML 文档来表示。这个 XML 文档有一个名为<Envelope>的根标记，这个根标记包含了一个必须的<Body>子标记和一个可选的<Header>子标记。<Body>标记中记录了有关方法调用的实际信息，例如方法名、参数、返回值（如果该方法有返回值的话）。<Header>标记这是可选的，包含了和方法调用没有直接关联的信息，这些信息以子节点的形式组织，每个子节点被称为一个 SOAP Header（SOAP 头）。程序员可以利用 SOAP Header 来传输额外的信息。

在 ASP.NET Web Services 的方法调用发生的背后，隐含了 SOAP 消息的格式控制过程和消息的传输过程。SOAP 协议指定了 SOAP 消息的内容使用 XML 语言，但该协议并没有严格的规范化 XML 的编码方式。ASP.NET 提供了基于属性标记的机制，来允许开发人员选择在 SOAP 报文内部使用何种编码格式的 XML。另外，ASP.NET 还提供了一个机制来允许对 SOAP 的传输过程进行相当精确的控制，这个机制也同样是基于属性标记的。

尽管 SOAP Header 可以包含和方法调用有关的数据，但由于 SOAP 规范并没有严格地定义 SOAP Header 的内容，它们往往被用来包含被基础结构处理的信息。一个典型的 SOAP Header 的应用是身份验证，如果对某个方法的调用需要身份验证，则可以由呼叫方将信任状信息放置在 SOAP Header 来传输。这样，不是每个 Web Service 的接口方法都需要处理这些 SOAP Header，它们可以忽略掉报文中的 Header 信息，而将身份验证的工作交给某些基础的实现代码来完成。

.NET Class Library 中，提供了一个 SoapHeader 类来抽象化和对象化 SOAP 协议中的 SOAP Header。在 ASP.NET 中，定义和操纵 SOAP Header 是通过继承该类来实现的。

首先，程序员在 asmx 文件或其 Code-Behind 源文件中创建一个 SoapHeader 类的子类，子类的名称会成为<Header>标记的子标记名。例如当用下面的 C#代码创建子类。

```
public class CustomHeader :
System.Web.Services.Protocols.SoapHeader
{
}

[WebService(Namespace="http://www.somesite.com")]
public class SampleService : System.Web.Services.WebService
{
// …
}
```

在 SOAP 消息中的<Envelope>中将会包含这样的<Header>标记：

```
<soap:Header xmlns="http://schemas.xmlsoap.org/soap/envelope/">
    <CustomHeader xmlns="http://www.somesite.com">
```

```
        </CustomHeader>
    </soap:Header>
```

然后，程序员可以在 SoapHeader 子类中定义特定的共有 field 或 property 成员，这些成员的数据类型必须符合前一节列举的 ASP.NET 支持的数据类型。例如我们为 CustomHeader 类定义成员：

```
public class CustomHeader :
System.Web.Services.Protocols.SoapHeader
{
    public string User;
    public string Pass;
}
```

则产生的 SOAP Header 为：

```
<soap:Header xmlns="http://schemas.xmlsoap.org/soap/envelope/">
    <CustomHeader xmlns="http://www.somesite.com">
        <User>something</User>
        <Pass>somethingelse</Pass>
    </CustomHeader>
</soap:Header>
```

这里的 something 和 somethingelse 是这两个 field 的值。

现在我们已经可以传输 SOAP Header 了，为了在 ASP.NET Web Service 中处理这些 SOAP Header，我们首先在自定义的 Web Serivce 类中添加一个共有的对象成员，该成员的类型为自定义的 SoapHeader 子类。例如：

```
[WebService(Namespace="http://www.somesite.com")]
public class SampleService : System.Web.Services.WebService
{
    public CustomHeader myCustomHeader;
}
```

然后为需要访问该 SOAP Header 的接口方法定义 SoapHeader 属性。SoapHeader 是 WebMethod 的一个对象属性，程序员需要将其 MemberName 属性值置为前面定义的 SoapHeader 类成员的对象变量名。例如对于 SampleService 类，实现一个需要访问 myCustomHeader 成员的接口方法如下。为了方便，我们引入 SoapHeader 对象属性的类型 SoapHeaderAttribute 所在的名字空间 System.Web.Services.Protocols。

```
using System.Web.Services.Protocols;
// …
[WebMethod]
[SoapHeader("myCustomHeader")]
public void MyMethod()
{
    string userid = myCustomHeader.User;
    // do something
}
```

对于没有使用属性标记方式声明对 SoapHeader 类型（实际上是 SoapHeader 子类）对象成员的引用的接口方法，SOAP 消息报中的 Header 则会被方法忽略。

对于客户端，在调用接受 SOAP Header 数据的接口方法时和调用那些忽略 SOAP Header 的接口方法时没什么两样的，除非该接口方法指定了 SOAP Header 数据是必须的，我们会在以后讨论这是如何实现的。客户端在调用接口方法时，如何在请求的 SOAP 包中包含<Header>信息呢？请注意，ASP.NET Web Services 已经将 SOAP Header 对象化为

SoapHeader 对象了，为了在接口方法中访问 SOAP Header，我们的 Web Service 类已经包含了 SoapHeader 派生类型的对象成员。在生成 Web Service 类的代理类时，代理类中将包含了相应的对象成员。因此，客户端可以操纵代理类中的对象成员，代理类的代码会自动将合适的对象成员的值放置到 SOAP 消息包中，或者从消息包中解析出 SOAP Header 数据放置到对象成员中。需要注意的是，使用 wsdl.exe 命令行工具生成的代理类会为 Web Services 类中的 SoapHeader 成员的每一种子类类型生成一个相应的对象，生成的对象名称为 SoapHeader 子类类型名后加上“Value”。例如，对于前面的 SampleService 类，wsdl.exe 生成的代码类会包括一个这样的成员：

```
public CustomHeader CustomHeaderValue;
```

并且包括了 CustomHeader 的定义：

```
public class CustomHeader : SoapHeader
{
    // …
}
```

注意到在代理类的构造函数中，并没有实例化自定义的 SoapHeader 子类对象。因此在客户端为这些对象赋值的时候，必须首先创建一个对象的实例，再把该对象实例赋给代理类的相应成员。因此，使用 SampleService 的自定义 SoapHeader 成员的代码会是这样的：

```
SampleService server = new SampleService();
CustomHeader myCustomHeader = new CustomHeader();
myCustomHeader.User = "Administrator";
myCustomHeader Pass = "password";
server.CustomHeaderValue = myCustomHeader;
server.MyMethod();
```

读者可能会有疑问：为什么 wsdl.exe 生成的代理类的 SoapHeader 成员名称会是按 SoapHeader 子类的类型来命名的呢？为什么不在代理类构造函数中实例化这些成员呢？用代码来描述的话，这两个问题实际上是说，为什么不让我们这样来写客户端代码呢：

```
SampleService server = new SampleService();
server.myCustomHeader.User = "Administrator";
server.myCustomHeader Pass ="password";
server.MyMethod();
```

这样的代码不是更简洁吗？我们会在后面讨论这样为什么不行。请留意，这个话题在 SDK 文档中是没有触及的。现在，请读者先把问题记在心里。

缺省情况下，当一个 Web Service 接口方法被赋予了 SoapHeader 属性时，SOAP Header 是由客户端发送给 Web Service 的接口方法的。然而，同采用 SOAP 协议传输的 Web Service 接口方法的参数一样，SOAP Header 也可以由接口方法回传给客户端，或者可以双向传输，后者类似于按引用传输。为了指明 SOAP Header 的传输方向，可以为接口方法的 SoapHeader 对象属性设定 Direction 属性。该属性的类型为 SoapHeaderDirection，这是一个枚举类型，枚举值有三个：In、Out 和 InOut，分别指定了传输方向为从客户端到接口方法、从接口方法到客户端和双向。

例如，如果前面的 MyMethod 接口方法需要从客户端接受 SOAP Header 并修改其内容后回传给客户端时，可以用下面的代码来将这个接口方法的 SoapHeader.SoapHeaderDirection 属性声明为 InOut：

```
[WebMethod]
[SoapHeader("myCustomHeader",
```

```
Direction=SoapHeaderDirection.InOut)]
    public void MyMethod()
    {
        // …
    }
```

当将 SoapHeader 的 Direction 属性设为 In 或 InOut 时，缺省情况下传向接口方法的 SOAP Header 是必须的，如果客户端没有在发送的 SOAP 消息中包含 SOAP Header，则会引发一个异常。Web Services 的设计者可以采取措施来避免这种情况，使得当不包含相应的 SOAP Header 的 SOAP 消息包到达这些接口方法时，ASP.NET 会将成员对象的值赋为 null，而不会抛出异常。程序员可以在方法的实现代码中通过判断 SoapHeader 成员是否为 null 来得知客户端有否传输 SOAP Header 信息。

具体的属性设置方法如下：

```
    [WebMethod]
    [SoapHeader("myCustomHeader",
Direction=SoapHeaderDirection.InOut), Required=false]
    public void MyMethod()
    {
        if (null == myCustomHeader)
        {
            // do something
        }
        else
        {
            // do something else
        }
        // …
    }
```

一个 Web Service 类可以拥有多个同类型或不同类型的 SoapHeader 派生类对象成员，但一个接口方法不能引用同类型的两个或两个以上 SoapHeader 派生类对象，这是因为 ASP.NET 将派生类名作为 SOAP Header 标记的名称，而同一名称的子标记在<Header>标记中是不允许的，ASP.NET Web Services 只是使用对象成员的名称来使从方法代码中访问相应的 SOAP Header 成为可能。接口方法可以引用多个不同类型的 SOAP Header，只要它们的类型两两不同。例如：

```
    public class Header1 : SoapHeader
    {
        //…
    }

    public class Header2 : SoapHeader
    {
        //…
    }

    [WebService(Namespace="www.somesite.com")]
    public class MutipleHeadersService : WebService
    {
        public Header1 myheader1;
        public Header2 myheader2;
        public Header2 anotherheader2;

        [WebMethod]
[SoapHeader("myheader1")]
```

```
    [SoapHeader("myheader2")]
    public void ValidMethod1()
    {
        // It's OK to refer to the soap header members
        // that are of different types
    }

    [WebMethod]
    [SoapHeader("anotherheader2")]
    public void ValidMethod2()
    {
        // it's OK to refer to some of the soap header members
        // and ignore the other soap headers
}

    [WebMethod]
    [SoapHeader("myheader2")]
    [SoapHeader("anotherheader2")]
    public void InvalidMethod()
    {
        // this function cannot be parsed
        // since it refers to two members, myheader2 and
            anotherheader2,
        // both of which are of the same type, Header2
    }
}
```

好了，还记得前面我们提到的关于 wsdl.exe 工具生成的代理类的使用的问题吗？现在我们已经有足够的理由来解决它了。我们的问题是这样的：如果 Web Service 类中包含了自定义的 SoapHeader 派生类，则 wsdl.exe 生成代理类的时候，是按 SoapHeader 派生类的子类类型来生成代理类成员的。并且，在代理类构造器中，wsdl.exe 没有为这些成员对象加入实例化的代码。因此，我们上面的 MultipleHeadersService 类的代理类的成员大致是这样的：

```
public class : MutipleHeaderService : ….
{
    // …
    public Header1 Header1Value;
    public Header2 Header2Value;
    // …
    public MutipleHeaderService()
    {
        this.Url = "http://...";
        // the constructor does not do the following jobs:
        /*
            Header1Value = new Header1();
            Header2Value = new Header2();
        */
}
}
```

为什么这样不行呢？

首先看看第一个子问题，为什么 wsdl.exe 是按 SoapHeader 子类类型来生成代理类的 SoapHeader 子类成员的呢？如果原来的 Web Service 类包含了多个同一类型的 SoapHeader 子类成员时，不就和原来的 Web Service 类不相互对应了吗？为了理解 wsdl.exe 这样做的理由，先看看如果不这样做会有什么情况。如果为同一类型的多个 SoapHeader 子类生成多个

对象成员，则我们的 MultipleHeaderService 的代理类会变成这样：

```
public class : MutipleHeaderService : ….
{
    // …
    public Header1 myheader1;
    public Header2 myheader2;
    public Header2 anotherheader2;
    // …
}
```

我们已经说过，ASP.NET 在产生 SOAP 消息时，会在 SOAP 消息包中包含 SOAP Header，按照 SOAP 规范，这些 SOAP Header 被放置在<Header>标记下，每个 SOAP Header 作为一个子元素，ASP.NET 在生成 SOAP 消息时，将 SoapHeader 子类对象成员的类型名作为元素名。因此，如果使用按 Web Service 中的 SoapHeader 对象成员名生成代理类的对象成员，则代理类在客户端生成传输到服务器的 SOAP 包时，将可能混淆这些对象成员。当一个接口方法要求某个 SOAP Header 时，这里不妨假设 MyMethod 接口方法接受一个 Header1 类型的 SOAP Header，那么我们在客户端写出的这两个版本的代码：

版本 1：

```
Header2 instance = new Header2();
MultipleHeaderService server = new MultipleHeader();
server.myheader2 = instance;
server.MyMethod();
```

版本 2：

```
Header2 instance = new Header2();
MultipleHeaderService server = new MultipleHeader();
server.anotherheader2 = instance;
server.MyMethod();
```

调用 MyMethod 的效果都是一样的，因为调用 MyMethod 之前的代码会产生同样的 SOAP Header：

```
<soap:Header xmlns="http://schemas.xmlsoap.org/soap/envelope/">
    <Header2 xmlns="http://www.somesite.com">
        <info></info>
    </Header2>
</soap:Header>
```

然而，在编程语言的层面上，例如从 C#的角度看，这两个 SOAP Header 成员是不同的对象，这两个版本的代码是“不应该总有”相同的作用结果的。因此，为了避免语言层面上的尴尬，wsdl.exe 的设计者决定，在生成 Web Service 代理类的时候，将 Web Service 类中同一子类类型的所有 SoapHeader 成员映射为同一个成员，并在子类类型名后加上“Value”作为成员名称。这样，程序员在同封装过的代理类打交道时，对同一类型的 SOAP Header 只能操纵同一个 SoapHeader 成员，从而避免了混淆。

到此为止，你可能在嘀咕了：“既然这样，似乎将 SoapHeader 子类类型名作为 SOAP Header 的标记名是万恶之源嘛”。是的，看起来如此。如果我们在设计代理类时，为每一个 SoapHeader 对象生成一个 SOAP Header 标记，并将 SoapHeader 对象名作为标记名，问题不就在更根本的层面上解决了吗？这样一来，wsdl.exe 的作者就不必考虑刚才的问题，而使用 wsdl.exe 工具生成的代理类的我们也可以用更自然的代码去操纵各个 SoapHeader 成员对象了啊……

等一下，世界上的事情总没有我们想象中的完美。我们考虑一下下面的情况，你就会打消刚才的念头了。

假设我们在 Web Service 定义 Header1 和 Header2 的成员是完全一样的：

```
public class Header1 : SoapHeader
{
    public string info;
}

public class Header2 : SoapHeader
{
    public string info;
}
```

则生成的 SOAP 消息包中的 Header 标记会是这样的：

```
<soap:Header xmlns="http://schemas.xmlsoap.org/soap/envelope/">
    <Header1 xmlns="http://www.somesite.com">
        <info>something</info>
    </Header1>
    <Header2 xmlns="http://www.somesite.com">
        <info>somethingelse</info>
    </Header2>
</soap:Header>
```

如果我们使用对象名来代替这里的 Header1 和 Header2 作为 SOAP Header 的标记名，那么下面的 Web Service 类：

```
public class PuzzlingHeaderService
{
    public Header1 infoA;
    public Header2 infoB;
}
```

它的代理类的传输过程中的 SOAP Header 会差不多是这样：

```
<soap:Header xmlns="http://schemas.xmlsoap.org/soap/envelope/">
    <infoA xmlns="http://www.somesite.com">
        <info>something</info>
    </infoA>
    <infoB xmlns="http://www.somesite.com">
        <info>somethingelse</info>
    </infoB>
</soap:Header>
```

可以想象，要区分这样的两个只有标记名称不同的 SOAP Header 的不同类型，解析器（无论是服务器端的 Web Service 类还是客户端的代理类）的难度将会很大。更重要的是，SOAP 是一个基于 XML 的协议，它的设计目的是成为 Web 上的远程调用机制，在这个过程中，描述对象的类型是十分关键的一步。如果将对象名称作为 SOAP Header 的标记名，这种描述类型的能力将丧失殆尽。

对于没有声明了 SoapHeader 对象属性的接口方法，是否就不能访问调用该方法的 SOAP 消息包中的 SOAP Header 信息了呢？ASP.NET 提供了一种方法，来方便程序员访问那些他们在编写代码时没有声明需要访问的 SOAP Header。这是因为，技术上讲，从客户端传到服务器端的 SOAP Header 实际上可以是任意的，这是 SOAP 协议允许的；从需求上讲，SOAP 协议给予的这个松散性也是必要的，例如，服务器端可能需要客户端提交某些信息，而这些信息的数量是不确定的，这就好像在高级程序语言中往往允许的函数的不定

参数一样（例如 C++中的…符号）。另外，SOAP 规范允许为某个 SOAP Header 指定一个 MustUnderstand 属性，如果处理程序没有处理这个 SOAP Header，则会抛出一个异常。在 ASP.NET 中，如果没有为 WebMethod 设定相应的属性来引用某些 SOAP Header，那么如果这些没有被声明引用的 SOAP Header 带有 MustUnderstand 属性时，就会有异常产生。相反，如果为它们声明了引用，即便在方法实现代码中没有处理它们，也不会有异常出现。因此，ASP.NET 提供了处理未知的 SOAP Header 的机制，以便设计者可以访问它们，也可以避免这些异常。

为了达到这个目的，Web Service 设计者首先为 Web Service 类添加一个类型为 SoapUnknownHeader 或者 SoapHeader 的对象成员，或者元素类型为这两种类型之一的对象数组成员。例如：

```
public class UnknownHeaderService
{
    public SoapUnknownHeader[] unknownheaders;
    // …
}
```

为需要处理未知的 SOAP Header 的接口方法添加一个 SoapHeader 属性，指明 Name 属性为对象名称，在这里是 unknownheaders，然后将它的 Required 属性置为 false。任何引用到 SoapHeader 类型或 SoapUnknownHeader 类型的 SoapHeader 属性都必须带有值为 false 的 Required 属性，如果没有指明该属性的值或置为 true，会有异常抛出。实现这一步的代码示例如下：

```
[WebMethod]
[SoapHeader("unknownheaders", Required=false]
public void MyMethod()
{
    // …
}
```

接着，在接口方法的实现代码中，可以通过判断引用的对象是否为 null 来得知相应的 SOAP Header 是否被传递。

SoapUnknowHeader 类是 SoapHeader 的一个派生类，它拥有一个 Element 属性，该属性为一个 XmlElement 对象，表示了对应的 SOAP Header 在 SOAP 请求包或应答包中的 XML 表示。由此，可以通过该 XmlElement 对象的 Name 属性来访问到 SOAP Header 的标记名称。另外，还可以像访问一般的 XML 节点一样来获取这个 SOAP Header 的信息。

例如：

```
[WebMethod]
[SoapHeader("unknownheaders", Required=false]
public void MyMethod()
{
    foreach (SoapUnknowHeader eachheader in unknownheaders)
    {
        if (eachheader.Name =="SomeHeaderName")
        {
            // do something…
        }
    }
}
```

接着，为了避免由于不识别某个 SOAP Header 而引发异常，可以将对应于该 SOAP

Header 的 SoapUnknownHeader 对象的 DidUnderstand 属性置为 true。对于无法识别而处理过的 SOAP Header，应该将该属性置为 false。例如：

```
[WebMethod]
[SoapHeader("unknownheaders", Required=false]
public void MyMethod()
{
    foreach (SoapUnknowHeader eachheader in unknownheaders)
    {
        if (eachheader.Name ==”SomeHeaderNameIknow”)
        {
            // …
            eachheader.DidUnderstand = true;
        }
        else
        {
            eachheader.DidUnderstand = false;
        }
    }
}
```

需要注意的是，DidUnderstand 属性在实际的 SOAP Header 的 XML 表示中是不存在的，它只是 ASP.NET Web Services 技术的一个用来和接口方法通讯的内部机制，并不是 SOAP 规范的一部分。另外，当一个 Web Service 类中存在 SoapUnknownHeader 类型的成员时，使用 wsdl.exe 生成的代理类中是不存在这些成员的。如果 Web Service 客户端代码需要往 SOAP 请求中添加这些 SOAP Header，客户端程序员必须修改代理类代码，为其添加一个成员变量。

在处理 SOAP Header 发生错误时，抛出的异常类型是 SoapHeaderException。程序员也可以在自己的 Web Service 逻辑中抛出 SoapHeaderException 来提示客户端调用者出错。

9.5　本章小结

本章知识是对 ASP.NET 技术的提高，介绍了 ASP.NET 技术的一个非常重要内容 Web Services。重点介绍了如何在 AS.NET 中编写和调用 Web Services。Web Servoces 作为一个新概念，正在迅速地被接受。ASP.NET 不仅具有创建 Web Services 的能力，而且使这种创建工具更为简单，开发人员可迅速地掌握这种新技术，并将其运用到可编程 Web 时代的网络应用程序中去。

第 10 章　ASP.NET 的设置、跟踪和安全

本章的内容覆盖面比较广，从如何设置 ASP.NET 应用程序到跟踪调试 ASP.NET 页面或是整个应用程序。其中，Global.asax 文件提供了高级事件的处理函数，Web.Config 文件记录了关于 Web 应用程序的各种默认配置，还可以根据需要自己修改 Web.Config 文件。本章将详细介绍其中的重要设置。本章还介绍了 ASP.NET 的一个非常有用的特性——跟踪。使用跟踪功能，可以快速地发现应用程序的问题所在。跟踪日志还是管理应用程序的重要参考。在本章的最后，介绍了 ASP.NET 中的安全验证和授权机制。ASP.NET 的身份验证和授权方式是可以根据需要定制的。

10.1　ASP.NET 的全局应用文件

10.1.1 什么是 Global.asax 文件

除了编写 UI 码，开发人员还可以为网络应用程序增加处理应用程序级的事件处理代码。这部分代码不会处理生成 UI。一般响应单独页的请求时不会调用这部分代码，也就是说，对于外部用户这部分代码是不可见的。这部分代码只会作为处理更高级别的应用程序事件时的响应，这些事件包括 Application_Start，Application_End，Session_Start，Session_End 等等。开发人员将这部分代码写成 Global.asax 文件，即全局应用文件。

Global.asax 文件位于指定的网络应用程序虚拟目录下的根目录中。如果 Global.asax 位于没有被标记为应用程序的目录中，那么将视为没有指定的 Global.asax 文件。

ASP.NET 会自动地将 Global.asax 文档解析和编译为继承自 HttpApplication 基类的动态.NET 框架类。

正如上面所说的，对 Global.asax 直接的 URL 请求会被自动地被拒绝，Global.asax 文档是不能被外部用户下载、阅读和修改的。

Global.asax 和 ASP 中的 Global.asa 文件类似，只是指令和事件有所不同。

```
<script language="C#" runat="server">

void Application_Start()
{
  // 处理应用程序开始事件的代码
}
</script>
```

这段代码的形式很像 ASP 中的 Global.asa 文档，但是在 Global.asax 中，可以使用下面的写法来指向源代码的另一个位置。

```
<script src="[位置]">
```

Global.asax 和 Globalx.asx 两个文档是可以共存而不互相影响的。因为两个文档的应用对象不同，分别是 ASP.NET 应用程序和 ASP。这样也使得 ASP 和 ASP.NET 的应用程序无法共享数据和事件。

可以使用 WYSIWYG 页面编辑器或是 Notepad 创建 Global.asax 文档，或是将 Global.asax 作为编译过的类放在“\bin”目录中并在另一个 Global.asax 文档中加以引用。

当应用程序的第一个请求到来时，Global.asax 将被编译，然后将应用程序的 Global.asax 实例用于每个请求的生存周期内的应用程序事件和状态的响应。如果修改 Global.asax 文档，那么 ASP.NET 将结束当前的应用程序所有请求作为响应，发出 Application_OnEnd 事件，然后重新开始新的应用程序，并在下一个请求到来时重新解析和编译 Global.asax 文档，发出 Application_OnStart 事件，然后根据新的 Global.asax 调用合适的应用程序启动事件。对于改动前的应用程序，终端浏览器用户还可以继续使用。这样作使得应用程序重新启动的过程成为无停机时间的过程。

10.1.2　如何阅读和编写 Global.asax 文件

Global.asax（全局应用文件），由以下四个部分组成：

- 应用指令（Application Directives）：指定了 ASP.NET 处理器处理 ASP.NET 文档时的可选设置。
- 代码声明（Code Declaration Blocks）（Application Directives）：定义了要被编译产生 Page 类的成员变量和方法。
- 服务器端对象标记（Sever-side Object Tags）：声明和创建新的对象实例。
- 服务器端包含指令（Server-side Include Directives）：添加指定文档的原始内容到 global.asax 文档。

现在，我们逐一介绍这四个部分的写法。

1. 应用指令

应用指令的格式为：

```
<%@ directive attribute=value [ attribute=value … ] %>
```

其中，directive 是指令名，attribute 是属性名，value 是属性的取值。每一条指令可能包含一个或多个属性与属性值的配对。

指令包括@Application，@Import，@Assembly 三种类型。其中，@Application 指令定义了 ASP.NET 应用程序编译器使用的应用程序特有的属性。@Import 指令用来为应用程序引入新的名字空间。@Assembly 指令在解析时为应用程序连接一个组合（Assembly）。

（1）@Application

@Application 指令支持的属性包括：

- INHERITS：INHERITS 属性值是要扩展的类的名字。通过 INHERITS 属性，你可以指定 Global.asax 要继承的类。

```
<% @Application INHERITS="Myapp.Application"%>
```

- DESCRIPTION：DESCRIPTION 属性负责对 INHERITS 属性进行描述，描述的内容会被当成文本注释内容，不被解析和编译。

```
<%@Application INHERITS="Myapp.Application" DESCRIPTION="My application"%>
```

（2）@Assembly

编译器会在编译时引用组合（Assembly），并允许早期的数据绑定。当请求页编译结束后，组合就被载入应用程序的域中，此时允许晚期数据绑定。

位于网络应用程序的“\Bin”目录中的组合会自动连接应用程序中的网页，不需要@Assembly 指令。“\Bin”目录中存放的是编译过的组合。如果将“Web.config”文档中<Assembly>部分中：<add assembly="*"/>一行去掉，那么就不会自动连接这些编译过的组合了。

@Assembly 指令的用法为：

```
<%@Assembly Name="组合名"%>
<%@Assembly Src="路径名"%>
```

组合名指定了要在文档中连接的组合，路径名则指定了要动态编译和连接的对象位置。路径名不能是组合的位置。

注意：@Assembly 指令的两个属性不可以在一条指令中同时出现，也就是说需要使用两个属性的时候，要像示例一样分成若干条@Assembly 指令。

除了使用@Assembly 指令，还可以在 Web.Config 文档中完成将组合连接到应用程序的设置，具体内容可以参见 ASP.NET 的配置一节。

（3）@Import

@Import 指令可以引入新的名字空间。关于@Import 指令的用法，可以参见本书 4.4.1 节的介绍。在这里我们就不再赘述了。

2. 代码声明

代码声明部分定义了成员变量、方法和事件处理函数。声明代码段的格式是：

```
<script runat="server" language="语言名" src="外部文件名">
  代码
</script>
```

代码段的语言可以是 C#，Visual Basic 或是 Jscript。属性“src”指定了要包含的外部文件，这些文件含有需要载入的代码段。

如果没有指定“language”属性，ASP.NET 会将默认值设为应用程序使用的语言。

下面的例子演示了怎样使用<script runat=“server”>段在 ASP.NET 应用程序文档中定义事件处理函数。

```
<script language="C#" runat="server">
void Application_Start(Object Sender, EventArgs e)
    {
        //代码
    }
void Application_End(Object Sender, EventArgs e)
    {
        //代码
    }
</script>
```

这里，我们为事件 Application_Start 和 Application_End 指定了处理函数。您可以自己完善示例中的代码。

3. 服务器端对象标记

当 ASP.NET 处理器在 ASP.NET 应用程序文档中遇到服务器端对象标记时，会产生一个“get/set”属性，使用标记的“ID”属性作为属性名。“get”方法使用来创建第一次使用的应用程序或是对话对象。

对象标记的语法为：

```
<object id="编号" runat=server class="类名">
<object id="编号" runat=server progid="组件名" />
<object id="编号" runat=server classid="组件名" />
```

其中，实例名是指 COM 组件创建的新实例名。

在<object>标记间的属性还有：

- Scope：指明对象声明的范围。可能的取值有“Pipeline/Application/Session”三种。“Pipeline”表明对象只有 HttpPipeline 类的实例可见。“Application”表明对象用于 HttpApplicationState 状态集。“Session”表明对象用于会话状态。Scope 属性的默认值是“Pipeline”。
- Class：指明创建对象的类型。

“class”、“progid”、“classid”这些属性都是互相排斥的，在一个服务器端对象标记内只能有一个这样的属性，否则会出错。

下面是一个在网页表单页中创建 Stack 对象的示例。

```
<object id="mystack" class="System.Collections.Stack"
runat="server" scrop-="appinstance" />
```

您可以通过“id”属性来调用这个对象。

4. 服务器端包含指令

包含指令包含的文档是先于任何动态代码执行的。

指令的格式是：

```
<! -- #include 路径类型＝文件名 -->
```

路径类型属性可以为：

- File：说明路径是相对路径，说明要包含的文档位于当前目录或是当前目录的子目录中。这里，当前目录就是“#include”指令所在的文档所处的目录。
- Virtual：指明使用的是网站的虚拟目录全路径。

要注意的是，文件名需要用双引号括起来。比如：

```
<! -- #include File= "MyHeader.inc" -->
```

我们使用服务器端包含指令包含了名为“MyHeader.inc”到表单页。

ASP.NET 提供几个模板用于在 Global.asax 文档中处理请求和事件。你也可以自己创建新的模板，或是扩展修改已有的模板。修改过的或新创建的模板应包含应用程序对于 HTTP 请求的处理信息。这些模板也必须执行 IHttpModule 接口的标准。

来看一个自定义模板的例子：

```
<Script language="C#" runat="server">
    void Session_OnStart()
     {
```

```
        //处理会话开始的代码
    }
    void Session_OnEnd()
    {
        //处理会话结束的代码
    }
    void Security_OnAuthentication(Object Source,
    AuthenticationEventArgs Details)
    {
        //处理 Summit 事件的代码
    }
</script>
```

在示例中，我们为 ASP.NET 网络应用程序创建了一个自定义的验证模板用来处理被扩展的 OnAuthentication 事件。这个事件包括会话开始、结束和 OnAuthentication 三个部分，分别用 Session_OnStart、Session_OnEnd 和 Security_OnAuthentication 函数处理。

注意到示例中将事件处理代码写成：

模板名_事件名(事件，参数)

这样做是因为要符合 HttpModule 接口的要求。

10.2　应用程序的 Web.Config 文件

无论 Web 应用程序的开发者还是应用程序的管理员都需要了解 Web 应用程序的设置，并进行必要的修改。在 ASP 中，这是通过 IIS 设置来实现的，而在 ASP.NET 中这样的设置信息是存放在一个称为 Web.Config 的配置文件中的。每一个 Web 应用程序的根目录下都有这样一个 Web.Config 文件。

ASP.NET 会对 IIS 加以设置，使得 Web.Config 文件不可以直接从浏览器中读取，从而保证了 Web.Config 文件的隐秘性。任何对 Web.Config 文件的请求都会得到“403:Access Forbidden”这样的错误信息。

ASP.NET 对 Web 应用程序的设置是分层次的，可以在应用程序级、网站级和服务器级几个等级来设置 Web 应用程序的参数。ASP.NET 允许将这些设置存储在静态文本中、动态页面中和应用程序单独的目录中。用户或是管理员只需建立一个虚拟目录就可以实现一个 Web 应用程序。

在 ASP.NET 中，各种设置是分为不同的配置段处理函数放在 Web.Config 文档的<system.web>段中。Web.Config 文档是可以被人读写的，是采用文本形式存储的。使用一个文本编辑器或 XML 解析器就可以阅读和修改更新这些设置了。

系统会自动检测配置文件的改动并且应用新的配置文件，和对 Global.asax 的改动类似，应用新的设置是不需要用户参与的，也就是说，不需要重新启动应用程序，甚至重启服务器。

Web.Config 文件是采用 XML 格式书写的。设置是通过一个个配置段来具体加以指定的。每一个配置段的标记和属性等的写法要保证格式良好，并且注意，XML 对于字母的大小写是区分的。所以在 Web.Config 文件中，对于标记名和属性名的写法是有规定的：在标

记名和属性名中，第一个字母是小写的，如果后面有连接的字，那么这个字的首字母是要大写的。比如：“<appSetting>”。如果属性的取值是一个枚举类型的值，那么第一个字母和连接的字的首字母都要大写。只有“true”和“false”两个枚举属性值例外，是小写的。

ASP.NET 的默认配置段包括：

- <httpModules>：对 HTTP 模板的设置
- <httpHandlers>：将自请求的 URL 映射到 IHttpHandler 类
- <sessionState>：设置 HTTP 模板的会话状态
- <globalization>：设置全局参数设定
- <compilation>：关于所有 ASP.NET 的编译设置
- <trace>：关于 ASP.NET 的跟踪特性的设置
- <processModel>：关于 IIS Web 服务器的 ASP.NET 进程模板设置
- <browserCaps>：对客户浏览器的相关设置

下面我们来看一些常用的配置段的说明。

（1）<appSetting>

指定应用程序的自定义设置，应用的范围是：硬件环境设置、网站设置、应用程序设置和对子目录的设置。使用<appSetting>段，我们可以集中地定义所有的应用程序配置。

<appSetting>段的格式为：

```
<appSetting>
    <add key="设置名" value-"设置值" />
</appSetting>
```

其中“key”属性指定了加入 hash 表中的设置名，“value”属性则给出了要设置的值。

比如：

```
<configuration>
   <appSettings>
      <add key="dsn" value="localhost;uid=sa;pwd="/>
   </appSettings>
</configuration>
```

增加了一个数据源名称（DSN）的配置到 hash 表中。

（2）<authentication>段

<authentication>段存放了有关 ASP.NET 验证的设置。

<authentication>段只有一个名为“mode”属性用来控制应用程序的默认验证模式。“mode”属性的取值有“Windows/Forms/Passport/None”四种。其中，“Forms”值表示默认的验证模式，是基于 ASP.NET 表单的验证模式，值“Passport”表示是使用微软的“Microsoft Passport”验证模式。值“Windows”指定了使用基于 Windows 的验证模式。使用这种模式是通过 Basic/Digest/NTLM/Kerberos 这样的 IIS 形式验证或是根据证书（certificates）验证信息有效性的。取值为“None”时，不使用任何验证模式，将允许匿名用户或是应用程序本身的事件处理函数可以验证信息。

<authentication>段还可以包含两个子标记元素，<forms>和<credentials>。

<forms>支持的属性有：

- name：指定用来验证的 HTTP cookie，默认为“*.ASPXAUTH”
- loginUrl：指定 cookie 不可用时的重定位页面的 URL，默认为“default.aspx”
- protection：指定应用程序的数据保护方式。可取“All/None/Encryption/Validation”，推荐取“All”值，同时采取“Encryption”和“Validation”两种方式。
- timeout：指明 cookie 的有效期，默认为 30 秒。
- path：指明 cookie 的路径位置，默认为“/”。

<forms>还支持子标记元素<credentials>，用来指定用户密码的加密方式。其形式为：

```
<credentials passwordFormat="Clear/SHA1/MD5">
    <user name="用户名" password="密码"/>
</credentials>
```

属性 passwordFormat 指定了子元素<user>中密码的加密方式，可以选择 MD5 或 SHA1 两种加密算法，还有 Clear 方式，即使用密码不会进行加密。

（3）<authorization>段

在<authorization>段设置了对 URL 资源访问权限。子标记<allow>指明可以得到 URL 资源的用户。比如：

```
<authorization>
   <allow roles="Admins" />
   <deny users="*" />
</authorization>
```

指明了允许“Admins”组的成员得到 URL 资源，拒绝所有用户的请求。

注意：对于用户来说，适用第一个符合的允许或拒绝的规则。如上例，“Admins”组的成员会先适用允许规则，然后适用拒绝规则，其结果是可以得到 URL 资源。

还可以通过用户名建立规则，比如：

```
<authorization>
   <allow roles="?" />
</authorization>
```

建立了允许匿名用户得到资源的规则。“?”还可以使用逗号相隔的用户名列表代替。在 ASP.NET 中，定义了“GET、HEAD、POST、DEBUG”四种传输方法。根据这些方法也可以建立规则。这需要使用“verbs”属性。比如：

```
<authorization>
   <allow verbs="GET" />
</authorization>
```

在 machine.config 中的默认设置是<allow users=“*”/>。

（4）<browserCaps>段

<browserCaps>段是对客户浏览器的配置。比如在无法判断客户端浏览器类型时，指定关于默认浏览器的配置。我们可以通过 HttpBrowserCapablities 类的属性得到客户端浏览器的属性。

（5）<compilation>段

在<compilation>段中，是对 ASP.NET 编译设置的说明。<compilation>段支持的属性有：

- debug：可能的取值是“true/false”，决定生成的二进制文档是否包含调试信息。

- defaultLanguage：指定动态编译文档的语言。比如“C#”。
- explicit：可能的取值是“true/false”。决定 Visual Basic 的 explicit 编译器是否可用。
- batch：可能的取值是“true/false”。决定是否支持 batch。
- batchTimeout：指定 batch 编译的时间，编译超出这个规定的时间，将转为单编译（single compilation）模式。
- numRecompilesBeforeAppRestart：指定在应用程序重新开始前，资源动态重编译的数量。这个属性在全局和应用程序级的范围内有效。
- strict：可能的取值是“true/false”。决定 Visual Basic 的 strict 编译器是否可用。
- <compilation>段还包括了三个子标记元素用来指定关于编译器、assembly（组合）和名字空间的信息。使用“<compiler>和<compilers>”标记设置关于编译器的信息，包括：

language	指明动态编译所用的语言。比如“C#;JScript;VB”。
extension	动态编译码文档的扩展名。比如“.cls;.pl”。
type	指明使用的已编译的 assembly 和.NET 框架的类。这些 assembly 和类是用来编译由指定的语言和文件扩展名确定的待编译文件的。
warningLevel	指明编译时警告的等级。
compilerOptions	指定编译时执行的其他设置。

对于 assembly，我们使用<assemblies>标记的子标记加以设置。增加 assembly 引用，使用<add>标记。<add>标记的值是 assembly 的名字，不能是 DLL 路径。<remove>标记指明了去除<add>标记增加的 assembly 引用。<remove>标记声明去除当前的和从指定的 Web.config 文档重继承到的所有 assembly 引用。

对于名字空间也可以使用“<add>、<clear>、<remove>”这三个标记完成与 assembly 类似的增加名字空间、去除指定名字空间和去除所有名字空间的操作。

我们通过一个应用程序的编译设定的例子，来看看<compilation>段的写法。

```
<configuration>
  <system.web>
    <compilation defaultLanguage="C#"
      debug="true"
      numRecompilesBeforeAppRestart="10">
      <compilers>
        <compiler language="C#;Csharp"
          extension=".cs"
          type="Microsoft.CSharp. CSharpCodeProvider,System" />
         <compiler language="VB;VBScript"
          extension=".cls"
          type="Microsoft.VB. VBCodeProvider,System" />

      </compilers>

      <assemblies>
        <add assembly="*" />
      </assemblies>

      <namespaces>
```

```
                    <add namespace="System.Web" />
                    <add namespace="System.Web.UI" />
                    <add namespace="System.Web.UI.WebControls" />
                    <add namespace="System.Web.UI.HtmlControls" />
                  </namespaces>

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

有多个<compiler>标记时，要使用<compilers>标记作为父标记，就像例子中的写法。

（6）<httpHandlers>

<httpHandlers>段根据请求中的 URL 和 HTTP 的动词，将请求映射到合适的 IHttpHandler 类和 IHttpHandlerFactory 类。<httpHandlers>支持三个子标记。

- <add>：指明将动词/通配符映射到 IHttpHandler 类和 IHttpHandlerFactory 类。其中，“verb”属性是 HTTP 动词列表（如 GET、PUT、POST）或是通配符“*”。“path”属性包含的是单 URL 路径或是像“*.aspx”这样含有通配符的字符串。“type”属性指明了使用的类和 assembly。ASP.NET 会先寻找应用程序私有的“/bin”目录中的 assembly DLL，然后在系统的 assembly 缓冲区中寻找。
- <remove>：删除<add>标记增加的映射。不能使用“*”通配符。<remove>可以使用属性“verb”和“path”指定删除具有指定动词和路径的处理程序映射。
- clear：删除所有的 IHttpHandler 映射。包括当前的映射和继承到的映射。

<add>标记指令是按照自上而下的顺序执行的。当有多个<add>标记指定相同的动词/路径时，以最后出现的<add>标记为准。

看一个例子：

```
<configuration>
  <system.web>
    <httpHandlers>
      <add verb="*" path=" StartHandler.Add"
        type="StartHandler.New, StartHandler.dll" />
      <add verb="*" path=" StartHandler.Trc"
        type=" StartHandler.Trc, StartHandler.dll" />
    </httpHandlers>
  <system.web>
</configuration>
```

在上面的示例中，所有的 HTTP 请求都映射到 StartHandler.Add 类和 StartHandler.Trc 类，assembly StartHandler 位于 StartHandler.dll 文档中。

（7）<sessionState>段

关于会话状态的设置，是放在<sessionState>段中的。比如：

```
<configuration>
  <system.web>
    <sessionState
      mode="Inproc"
      cookieless="false"
      timeout="20" />
    </sessionState>
  </system.web>
</configuration>
```

<sessionState>段支持的属性有：

- mode：可能的取值为“Off/Inproc/StateServer/SqlServer”。指定了储存会话状态的位置。“Off”说明会话状态不可用。取值为“Inproc”表示会话状态在本地储存，“StateServer”表示会话状态在远程服务器存储，“SqlServer”表示会话状态存储在 SQL Server 上。
- cookieless：可能的取值有“true/false”。表示没有 cookie 时，会话是否可用。默认值是 false，即没有 cookie 作为标识键，不能进行会话。
- timeout：以分钟为单位，设置会话的空闲时间。默认值是 20。
- connectionString：设置存储会话状态的远程服务器名和端口。当“mode”属性取“StateServer”时，属性才可用。取值可以为“127.0.0.1:42424”这样的形式。
- sqlConnectionString：设置存储会话状态的 SQL Server 的连接字符串。比如：“data source=127.0.0.1;use id=sa;password= ”。在“mode”属性取“SqlServer”时，属性可用。

（8）<webServices>段

在<webServices>段中参数设置比较复杂，我们来看在<webServices>段中最重要的设置，关于 Web Service 中模板页的设置：

```
<configuration>
    <system.web>
        <WebServices>
            ……
            <sdlHelpGenerator href="DefaultSdlHelpGenerator.aspx"
            />
            ……
        </webServices>
    <system.web>
</configuration>
```

通过属性“href”，我们将 DefaultSdlHelpGenerator.aspx 页作为模板页。关于在 Web Service 中 DefaultSdlHelpGenerator.aspx 页的介绍，可以参考本书第九章关于 Web Service 的说明。

最后，通过加入<location>的 allowOverride 属性可以实现是否拒绝对 Web.Config 文件中的设置段的修改。

上面我们已经将常用的配置做了详尽的介绍，在 Web.Config 文件中，还有其他的配置段对 ASP.NET 进行设置，这些内容并不常用，我们就不在这里详述了。你可以参考其他相关的资料。

10.3　ASP.NET 的跟踪和完善

10.3.1　ASP.NET 的优化

在当今的网络商务时代，在 B2B（business-to-business）和 B2C（business-to-consumers）模式中，Web 应用程序的性能显得至关重要。对一个商务公司来说，Web 应用程序性能低

下带来的不仅仅是资源的浪费，还会失去潜在的消费者。性能对于应用程序的开发人员和应用程序的系统管理维护人员都是及其重要的。

ASP.NET 具有的特性和提供的工具可以提高 Web 应用程序的性能。从一个改善的进程模型和自动的缓冲机制到性能计算工具和 Web 应用程序的测试工具，使用 ASP.NET 可以设计、开发高效的 Web 应用程序。这正是高访问量、高负荷网站所需要的技术。

开发高效的 ASP.NET 应用程序，我们看重的是 Web 应用程序的响应时间和吞吐量。下面我们讲一些在开发时要注意的问题。这些往往是性能降低的重要因素。

（1）及时地结束会话状态

不是所有的应用程序或是网页都需要为每一个用户开始一个会话状态的，应该及时的将不必要的会话状态结束掉。

要结束网页的会话状态，我们可以在@Page 指令中将 EnableSessionState 属性置为 false。如果网页需要访问会话变量（session variables），但不会创建和修改这些变量，那么可以将 EnableSessionState 属性置为 ReadOnly。比如：

```
<%@ Page EnableSessionState="false" %>.
```

结束 Web 应用程序的会话状态，我们可以在应用程序的 web.config 文件的<sessionstate>段加以设置。比如：

```
<sessionstate mode="off" />
```

（2）选择合适的方式存储会话状态

ASP.NET 提供了三种不同的存储应用程序会话状态的方法：在本地服务器上、在远程服务器上和在 SQL Server 上。每一种方法都有相应的优势，比如：本地服务器存储会话状态的速度是最快的，在储存少量的数据时是比较好的选择，而在远程服务器上存储会话状态有利于多进程的或是运行于多台服务器的应用程序的执行，因为数据不会因为一个进程或是一台服务器重起而丢失。

（3）避免不必要的服务器访问

一般来说，服务器访问只用在应用程序存储和调用数据的请求。多数的数据操作可以在客户端和这些访问之间的阶段完成。比如检查 HTML 表单中的用户输入是否正确，可以在用户向服务器提交表单之前完成。如果不需要向服务器提交要存入数据库的信息，那么应该避免使用对服务器的访问。

如果开发自己的服务器控件，可以考虑将控件放入客户端代码中，高级的浏览器是支持这样做的（支持 ECMAScript）。这样就可以减少不必要的服务器访问了。

（4）使用 Page.IsPostBack 减少执行不必要的访问进程

如果代码中有服务器控件 postback 进程的处理函数，那么有时会需要根据页面的第一次请求和 postback 的页面两种情况编写不同的代码。使用 Page.IsPostBack 属性可以有条件地完成判断这两种情况。比如：

```
void Page_Load(Object sender, EventArgs e)
    {
       if (!Page.IsPostBack)
        {
           info.Text = "Log in first please";
       }
```

```
}
```

我们对每次 Page_Load 事件的请求根据 Page.IsPostBack 属性判断是否请求的是 postback 的页面，如果不是，那么会提示“Log in first please”信息。

除了上面提到的一些经验和注意事项外，你还可以在开发过程中总结自己的心得体会，以利于提高应用程序的性能。从某种程度上讲，优化 ASP.NET 应用程序和开发应用程序同等重要。

在 ASP.NET 中，还可以提供一系列方法和工具对应用程序进行测试和管理。微软公司提供了 Web Application Stress（WAS）工具。使用这个工具，可以模拟多 HTTP 客户端对网站的请求。在 ASP.NET 中，还有多种性能计算器可以跟踪应用程序的执行过程。

我们可以使用 WAS 在图形界面中，控制客户端的载入、连接数量、cookie 格式、HTTP 响应的头信息等参数。在一次测试结束后，WAS 会提供一份包括响应时间、吞吐量等由多个计算器得到数据报告。通过这个工具，我们可以测试应用程序在高负荷运行时的最大吞吐量和 CPU 占用率。

多数的 ASP.NET 的性能计算器的实例是应用于单个应用程序的。通过“ASP.NET Applications”执行对象可以调用这些计算器。如果服务器上有多个应用程序，那么需要为性能计算器指定其中一个要计算性能的应用程序。或者使用_Total_应用程序实例计算所有应用程序性能计算器。

通过 ASP.NET System 执行对象，我们可以使用全局性能计算器（global-only counter）计算整个服务器上所有的应用程序性能。

ASP.NET 的跟踪特性使得我们可以获取一连串跟踪输出语句被执行时所记录下的时间参数，这些时间参数是十分重要的。除此之外，我们还可以获得服务器代码执行时控制权的传递顺序。关于 ASP.NET 程序的跟踪，来看下面的介绍。

10.3.2　跟踪

在网页和应用程序的调试、检查运行情况和处理错误等方面，ASP.NET 提供了称为“跟踪”的新技术。这和在 ASP 中采用 Response.Write 指令对变量进行的跟踪是完全不同的。通过跟踪得到信息有助于开发人员处理错误和调试应用程序。

跟踪的结果是在页面的最后增加一张表格。在表中列出了各种指定的性能数据和自定义的错误处理信息。这些数据和信息可以输出到发出请求的浏览器，或输出到位于应用程序根目录下的 trace.axd 文件中。

在 ASP.NET 中，内置了一个 Trace 对象允许我们输出整页或是整个应用程序的调试信息。ASP.NET 使用 TraceContext 类来存储有关请求的信息、控制层次和跟踪信息。这些跟踪信息包括一个页请求的生存周期在内的任何你所需要的信息。通过 Page.Trace 类的属性，可以调用 TraceContext 类。

跟踪特性只有在激活的状态下，才会显示跟踪到的信息。我们可以控制是否将跟踪激活以及将跟踪信息输出到页面或是到文件。在默认的情况下，是不输出跟踪信息的。但是，通过@Page 指令可以激活跟踪，将跟踪信息显示出来。比如：

```
<%@ Page Trace="true" %>
<script language="C#" runat="server">
    void Page_Load()
    {
        Trace.Write ("Trace test", "Page Load Completed");
    }
</script>
<HTML>
    <H2 align="center">Example for Trace </H2>
</HTML>
```

只需在@Page 指令中，将“Trace”属性置为“true”，就可以输出这些跟踪信息了。如图 10.1 所示。

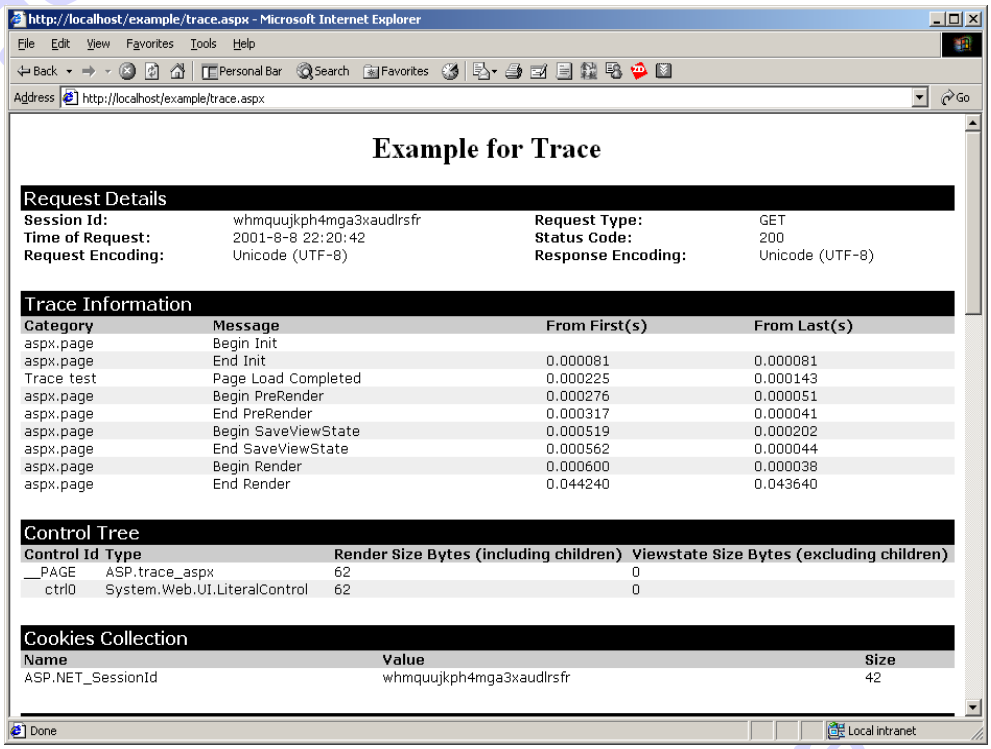


图 10.1　输出跟踪信息

TraceContext 类提供了两种方式来输出跟踪信息：写（write）和警告（warn）。这两种方法的区别仅仅在于以警告方式看到的跟踪信息是红色的。使用 TraceContext.Write 和 TraceContext.Warn 方法输出到跟踪日志中的内容包括跟踪信息、跟踪指定的内容得到信息和错误信息。

1. 页面级跟踪

使用页面级跟踪，我们可以使用 TraceContext 对象将调试信息写到发给客户端浏览器的页面末尾处。ASP.NET 还插入了一些有用的内容，包括记录方法的生存周期，开始时间和结束时间，比如记录 Load 和 Dispose 方法。因为可以指定是否跟踪页面，所以跟踪模块存在于产品代码中，但不用担心，跟踪模块不会对代码的执行产生任何影响。

对于跟踪到的信息，可以选择按照执行顺序输出或是按照目录字母表的顺序输出。

输出跟踪信息到浏览器的方法是在@Page 指令中将“Trace”属性置为“true”，如上面的例子。同时，属性“TraceContext.IsEnabled”的值也会相应地变为“true”。

跟踪信息的两种排序方法是通过 TraceMode 属性指定的。比如按照目录字母表的先后顺序输出的指令为：

```
<%@ Page Language="C#" Trace="True" TraceMode="SortByCategory" %>
```

“TraceMode”还有另一个值“SortByTime”，指定按照时间顺序输出跟踪信息。省略这个属性时，会按照时间顺序输出信息，比如前面的例子。

激活跟踪时，会将这一页的跟踪信息显示于任何请求该页面的浏览器中。这些信息对于产品客户是不必要的，所以在应用程序开发结束后应取消对页面的跟踪。

2. 应用程序级跟踪

除了跟踪页面，ASP.NET 还提供了跟踪整个应用程序的功能，即应用程序级跟踪。

与页面跟踪不同的是，应用程序级跟踪结果除非指定，否则不会在浏览器中显示。通常，开发人员是通过一个基于网页的跟踪视窗程序来观察跟踪信息的。这个视窗会显示跟踪给应用程序的一系列请求的信息，所以在跟踪模块激活时，每个请求都要保存在内存中。ASP.NET 会跟踪指定数目内的每个请求，默认最多跟踪 10 个请求。

使用应用程序级跟踪会自动地对应用程序的每一页进行页面级跟踪，除非对指定的页使用@Page 指令的“Trace”属性禁止跟踪。要在网页中观察跟踪信息，需要将 PageOutput 属性置为“true”。这个属性位于应用程序的 Web.Config 文件中。

在 Web.Config 文件中，是使用“<trace>”标记完成对跟踪特性的设置的。“<trace>”标记支持的属性有：

- enabled：决定是否跟踪应用程序，可取值为“true/false”。默认值是“false”，即不跟踪应用程序。
- pageOutput：可取值为“true/false”。当取“true”时，跟踪信息会输出到浏览器和*.axd 文件中。默认值是“false”。参见上面的说明。
- requestLimit：最大跟踪请求数。默认值是 10 个。超出此数的请求将不被跟踪。
- traceMode：可能取值有“SortByTime/SortByCategory”，默认值是“SortByTime”。可以参考页面级跟踪中@Page 指令的说明。
- localOnly：可能的取值有“true/false”，取默认值“true”时，跟踪视窗（trace.axd）只在应用程序所在的服务器上可见（参见图 10.2）。

看一个例子：

```
<configuration>
 <system.web>
  <trace enabled="true" requestLimit="30" pageOutput="false"/>
 </system.web>
</configuration>
```

示例的设置为最多跟踪 30 个请求，跟踪信息不会在浏览器中显示出来。

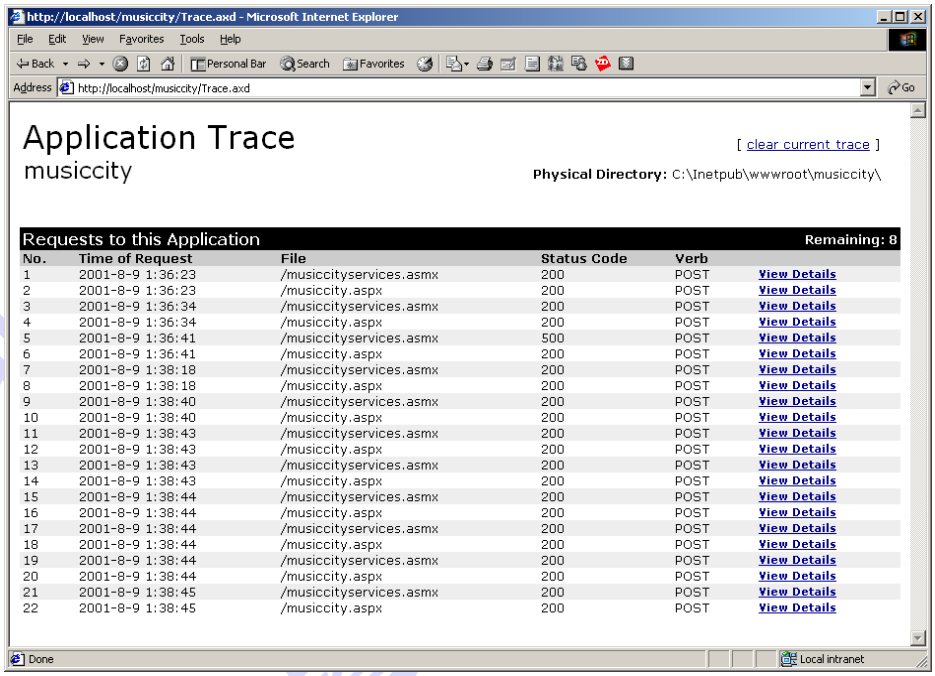


图 10.2

当激活跟踪时，应用程序级跟踪得到的信息会自动输出到 trace.axd 文件中。trace.axd 文件也是位于应用程序根目录下的。通过 trace.axd 文件，还可以跟踪应用程序的每一个页面。如果 Web 应用程序的 URL 是“http://localhost/application”，那么我们可以通过浏览器由 URL“http://localhost/application/trace.axd”访问跟踪到的信息，如图 10.3 所示。

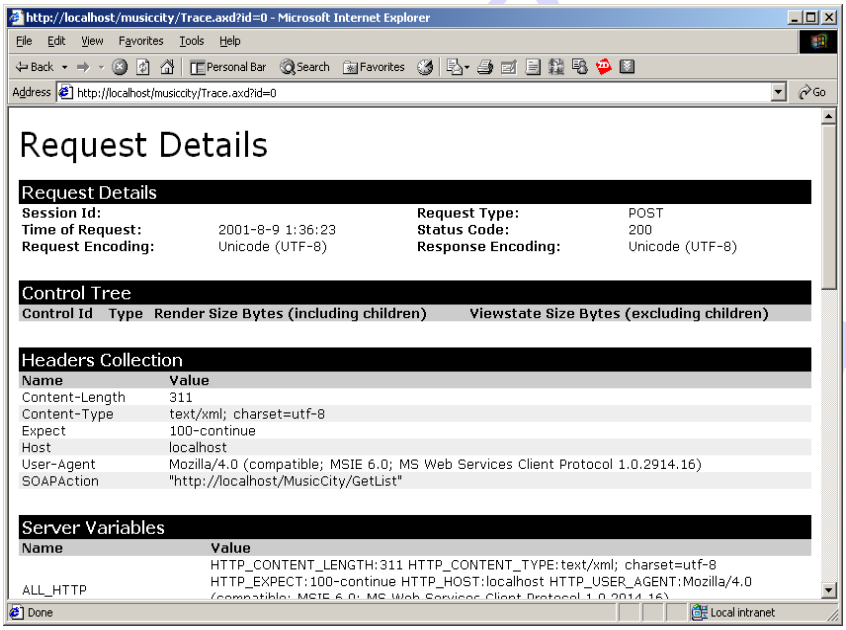


图 10.3

其中，每一行都是一个页面请求的跟踪信息，单击 View Details 链接可以看到详细的请求跟踪信息。超出最大跟踪请求数时，将不会再记录新的请求了。

通过前面的介绍，我们已经可以得到详细的页面或是应用程序的跟踪日志了，可是怎么阅读这份详细的报告呢？我们来逐段地分析这份日志。

一份应用程序级跟踪的请求跟踪信息和对该页面进行单独的页面级跟踪得到信息是相同的。因为应用程序级跟踪会默认地跟踪对每一页的请求。跟踪日志有六部分组成：

- 请求细节（Request Details）
- 跟踪信息（Trace Information）
- 控件树（Control Tree）
- cookie 集合（Cookies Collection）
- HTML 头信息（Headers Collection）
- 服务器变量（Server Variables）

首先来看第一部分——请求细节，如图 10.4 所示。

Request Details

Session Id:	gw3f1s45nw2hyvz3rvysex3g	Request Type:	GET
Time of Request:	2001-8-9 2:08:09	Status Code:	200
Request Encoding:	Unicode (UTF-8)	Response Encoding:	Unicode (UTF-8)

图 10.4

这一部分内容比较简单，是关于详细的请求参数，如会话编号（Session ID）等，我们就不多说了。其中请求类型（Request Type）可以为 GET 或是 POST。如果有问题，可以参见本书第九章关于 Web Service 的介绍。

然后是跟踪信息，如图 10.5 所示。

Trace Information

Category	Message	From First(s)	From Last(s)
aspx.page	Begin Init		
aspx.page	End Init	0.000116	0.000116
aspx.page	Begin PreRender	0.000270	0.000154
aspx.page	End PreRender	0.000324	0.000054
aspx.page	Begin SaveViewState	0.000893	0.000568
aspx.page	End SaveViewState	0.001045	0.000152
aspx.page	Begin Render	0.001088	0.000043
aspx.page	End Render	0.028081	0.026993

10.5

跟踪信息包括的是跟踪由 Write 或是 Warn 方法指定的事件或是其他内容得到的信息。目录（Category）部分是要跟踪的内容名称，可以根据字母顺序排序。“Form First”和“Form Last”两列是从第一条信息算起的显示时间和从最近的信息算起的显示时间。

下面是控件树部分，如图 10.6 所示（截取一部分）。

Control Tree

Control Id	Type	Render Size Bytes (including children)	Viewstate Size Bytes (excluding children)
_PAGE	ASP.Music	652	24
ctrl1	System.Web.UI.LiteralControl	150	0
ctrl0	System.Web.UI.HtmlControls.HtmlForm	484	0
ctrl2	System.Web.UI.LiteralControl	5	0
btnList	System.Web.UI.WebControls.Button	74	0
ctrl3	System.Web.UI.LiteralControl	70	0
AlbumTitle	System.Web.UI.WebControls.TextBox	55	0
ctrl4	System.Web.UI.LiteralControl	14	0
btnSearch	System.Web.UI.WebControls.Button	76	0
ctrl5	System.Web.UI.LiteralControl	29	0

图 10.6

在这部分里给出了控件的层次关系。对于每一个控件，则指明了 ID、类型、Render 大小和 Viewstate 大小。其中，类型给出的是控件类型的全名；Render 大小和 Viewstate 大小是以字节（byte）计算的。

接下来是 cookie 集合，如图 10.7 所示。

Cookies Collection

Name	Value	Size
ASP.NET_SessionId	gw3f1s45nw2hyvz3rvysex3g	42

图 10.7

其中包括 cookie 名、cookie 值和以字节计算的 cookie 大小。

在 cookie 集合之后是 HTML 头信息，如图 10.8 所示。

Headers Collection

Name	Value
Connection	Keep-Alive
Accept	*/*
Accept-Encoding	gzip, deflate
Accept-Language	zh-cn
Host	localhost
User-Agent	Mozilla/4.0 (compatible; MSIE 6.0b; Windows NT 5.0; .NET CLR 1.0.2914)

图 10.8

最后是服务器变量列表，如图 10.9（截取一部分）所示。

Server Variables

Name	Value
ALL_HTTP	HTTP_CONNECTION:Keep-Alive HTTP_ACCEPT:*/* HTTP_ACCEPT_ENCODING:gzip, deflate HTTP_ACCEPT_LANGUAGE:zh-cn HTTP_HOST:localhost HTTP_USER_AGENT:Mozilla/4.0 (compatible; MSIE 6.0b; Windows NT 5.0; .NET CLR 1.0.2914)
ALL_RAW	Connection: Keep-Alive Accept: */* Accept-Encoding: gzip, deflate Accept-Language: zh-cn Host: localhost User-Agent: Mozilla/4.0 (compatible; MSIE 6.0b; Windows NT 5.0; .NET CLR 1.0.2914)
APPL_MD_PATH	/LM/W3SVC/1/Root/musiccity
APPL_PHYSICAL_PATH	C:\Inetpub\wwwroot\musiccity\
AUTH_TYPE	
AUTH_USER	
AUTH_PASSWORD	
LOGON_USER	

图 10.9

10.4　ASP.NET 的安全机制

安全是许多 Web 应用程序的重要部分，因为常常会根据用户的权限不同来决定哪些资源是可用的。判断用户权限的过程称为验证（authentication）；根据不同的权限决定可否得到请求的资源的过程称为授权（authorization）。

一般来说，每个用户都会有一个身份以及相应的权限的标记，比如说通过用户名和密码来区分不同的用户。当身份得到确认之后，会根据与身份相应的权限判断是否可以得到请求的资源。ASP.NET 和 IIS 的验证和授权的过程是联系在一起的。比如，要使一个 IIS 应用程序使用 Basic 验证方式，那么必须使用 Internet Service Manager 来设置应用程序使用 Basic 验证方式。

一些应用程序需要根据请求者的身份和权限动态地改变可访问得到的内容。比如，应用程序会检查当前用户是否是拥有管理员权限，如果是的话，就在普通信息的基础上增加相应的管理信息。对于没有管理员权限的一般用户，只可以访问普通信息，这些管理信息是不可见的。ASP.NET 应用程序就可以实现这种动态地决定当前身份是否可以得到所有资源，比如普通信息之外的管理信息。

ASP.NET 提供了三种验证方式：

- 基于 Basic、Digest 和 Windows 的验证。
- 基于表单的验证。这种方式是使用 cookie 来验证用户身份的，并且允许应用程序使用自己的证书来确认身份。
- 使用微软的“Microsoft Passport”的验证方式。

在 Web.Config 文件中可以设置要使用的验证方式，我们在 10.2 节中已经介绍过了。

ASP.NET 提供了两种类型的授权服务：

- 检查是否存在资源的 ACL（access-control list）或是许可（permission）来决定已确认身份的用户是否可以访问相应的资源。
- 通过 URL 来授权，根据不同的身份决定是否有权限访问的相应的 Web 空间。

我们用一个例子来区分这两种方法。设想一种情形，应用程序允许匿名用户使用“TO_ALL”权限。那么匿名用户请求一个 ASP.NET 页，比如“/default.aspx”，其授权过程是检查“default.aspx”文件的 ASL，看“TO_ALL”权限是否拥有访问“default.aspx”文件的许可。如果这样的权限存在，那么可以授权对“default.aspx”的访问，也就是说，匿名用户可以浏览“default.aspx”页了。

通过 URL 授权的过程是根据 ASP.NET 应用程序的设置，来决定匿名用户是否可以访问请求的 URL。如果请求的 URL 可以访问，那么就是授权的访问。在这种情况下，ASP.NET 检查匿名用户是否可以访问“/default.aspx”。其中的区别是，授权与否是 URL 本身决定的，不是由与 URL 相应的文件决定的。

这种差别看起来不大，但是这就允许应用程序使用两种验证方式：像基于表单的验证方式这样的指定的验证方式，或是使用 Microsoft Passport 验证方式。后一种方式是不需要

检查权限的。这也允许对虚资源进行授权。所谓虚资源是指没有真正的物理页对应的资源。比如，一个应用程序可以将对以“stk”结尾的文件请求映射到相应的处理函数。这样，没有“stk”来检查 ACL，所以只能使用 URL 授权来控制对虚资源的请求。

文件的 ACL 是放入文件或是文件夹中的。这些文件和文件夹在资源管理器的属性页中 Security 标签内指定。

我们将着重介绍基于表单的验证方式。因为 ASP.NET 提供的这种方式允许应用程序提供自己的登录 UI，确认自己的证书。ASP.NET 会将没有经过身份验证的用户重定位到登录页面，然后进行必须的 cookie 管理。这种技术在现在的网站非常流行。

在介绍 Web.Config 文件时，我们讲到了<authentication>标记。设置应用程序采用基于表单的验证方式就是将“mode”属性置为“Forms”。比如，下面的这个例子就是采用基于表单的验证方式的。

```
<configuration>
  <system.web>
    <authentication mode="Forms"/>
    <authorization>
        <allow roles="Admins" />
       <deny users="?" />
    </authorization>
  </system.web>
</configuration>
```

在设置了验证方式之后，我们还需要一个默认的登录页面。在下面的例子中，会看到一个简单的登录页面示例。当用户填入用户名和密码后，会对用户名和密码进行确认。通过确认之后，应用程序调用：

```
FormsAuthentication.RedirectFromLoginPage(UserEmail.Value,
PersistCookie.Checked);
```

重定位网页的默认开始页（default.aspx）。

关于应用程序使用基于表单的验证方式，我们来看一个例子，在例子中我们会结合实际介绍 ASP.NET 验证方式：

程序清单 10-1：example-10-3-login.asax

```
<%@ Import Namespace="System" %>
<%@ Import Namespace="System.IO" %>
<%@ Import Namespace="System.Data" %>
<%@ Import Namespace="System.Xml" %>
<%@ Import Namespace="System.Web.Security " %>
<%@ Import Namespace="System.Web.UI.WebControls" %>

<html>
  <script language="C#" runat=server>

    void Login_Click(Object sender, EventArgs E)
    {
         //create a reader
         XmlTextReader reader = null;
         try
         {
             bool Valid = false;
             reader = new XmlTextReader("C:\\Inetpub\\wwwroot\\
              example\\Login.xml");
```

```
                reader.WhitespaceHandling = WhitespaceHandling.None;

                //search the data file.
                while (reader.Read())
                {
                   if(reader.MoveToContent() == XmlNodeType.Element
                         && reader.Name == "Name")
                     {
                       //authenticate user's identity.
                       if(reader.ReadElementString() == UserName.Value
                          &&
                          reader.ReadElementString() == UserPass.Value)
                       {

FormsAuthentication.RedirectFromLoginPage(UserName.Value,
                          PersistCookie.Checked);
                          Valid = true;   //validated the credentials,
                       }
                   }
                 }
              if(!Valid)
              {
               Msg.Text = "Invalid Credentials: Please try again";
              }
            }
           finally
           {
               if (reader!=null)
                   reader.Close();
           }
       }
      void Register_Click(Object sender, EventArgs E)
      {
          DataSet ds = new DataSet();

          // open and read Login.xml
          FileStream fs = new FileStream(Server.MapPath("Login.xml"),
               FileMode.Open, FileAccess.Read, FileShare.ReadWrite);
          StreamReader reader = new StreamReader(fs);
          ds.ReadXml(reader);
          fs.Close();

          // append a row
          try
          {
              DataRow newUser = ds.Tables[0].NewRow();
              newUser["Name"] = UserName.Value;
                newUser["Password"] = UserPass.Value;

                ds.Tables[0].Rows.Add(newUser);
          }
          catch(Exception)
          {
              Msg.Text = "Register failed.Please try again.";
              return;
          }

          // rewrite the data file
```

```
            fs = new FileStream(Server.MapPath("Login.xml"),
                  FileMode.Create, FileAccess.ReadWrite, FileShare
                  .ReadWrite);
            TextWriter writer = new StreamWriter(fs);
            writer = TextWriter.Synchronized(writer);
            ds.WriteXml(writer);
            writer.Close();

            LoadData();
        }

        void LoadData()
        {
            // read the data from Login.xml
            DataSet ds = new DataSet();

            FileStream fs = new FileStream(Server.MapPath("Login.xml"),
                 FileMode.Open,FileAccess.Read);
            StreamReader reader = new StreamReader(fs);
            ds.ReadXml(reader);
            fs.Close();

             MyDataGrid.DataSource = ds.Tables[0].DefaultView;
                MyDataGrid.DataBind();

         }
</script>

<body>

  <form runat=server>
     <h2><font color="red">Here are available users and their
      passwords.</font></h2>
    <ASP:DataGrid id="MyDataGrid" runat="server"
      Width="300"
      BackColor="lightyellow"
      BorderColor="black"
      CellSpacing="0"
      Font-Name="Verdana"
      Font-Size="10pt"
      HeaderStyle-BackColor="#abcdef"
    />

    <h3><font face="Verdana">Login Page</font></h3>

    <table>
      <tr>
        <td>Name:</td>
        <td><input id="UserName" type="text" runat=server/></td>
        <td><ASP:RequiredFieldValidator ControlToValidate
         ="UserName"
                Display="Static" ErrorMessage="*" runat=
                server/></td>
      </tr>
      <tr>
        <td>Password:</td>
        <td><input id="UserPass" type=password runat=server/></td>
        <td><ASP:RequiredFieldValidator ControlToValidate
         ="UserPass"
```

```
                    Display="Static" ErrorMessage="*" runat=
                    server/></td>
                </tr>
                <tr>
                  <td>Persistent Cookie:</td>
                  <td><ASP:CheckBox id=PersistCookie runat="server" /> </td>
                  <td></td>
                </tr>
              </table>
              <p>
                 <asp:button text="Login" OnClick="Login_Click" runat=
                  server/>
             <asp:button text="Register" OnClick="Register_Click" runat=
              server/>
              <p>
             <asp:Label id="Msg" ForeColor="red" Font-Size="20" runat=server
              />
             </form>
            </body>
          </html>
```

程序清单 10-2：example-10-3-default.aspx

```
<%@ Import Namespace="System.Web.Security " %>
<html>
  <script language="C#" runat=server>

    void Page_Load(Object Src, EventArgs E )
    {
       Welcome.Text = "Hello, " + User.Identity.Name;
    }

    void Signout_Click(Object sender, EventArgs E)
    {
      FormsAuthentication.SignOut();
      Response.Redirect("login.aspx");
    }

  </script>

  <body>
    <h3 align="center">
        <font face="Verdana">Done! We are using cookie
         authentication.</font>
     </h3>
    <form runat=server>
      <h3 align="center"><asp:label id="Welcome" runat=
       server/></h3>

     <p align="center">
      <asp:button text="Signout" OnClick="Signout_Click" runat=
       server/>
     </p>

    </form>
  </body>
</html>
```

程序清单 10-3：example-10-3-web.cinfig

```
<configuration>
```

```
    <system.web>
        <authentication mode="Forms">
          <forms name=".SECURITY" loginUrl="login.aspx" protection=
           "All" timeout="30" />
        </authentication>
        <authorization>
          <deny users="?" />
        </authorization>
      <globalization requestEncoding="UTF-8" responseEncoding=
       "UTF-8" />
    </system.web>
</configuration>
```

在这个例子中，我们实现了将根据用户名和密码来确认用户身份，并授权用户相应访问权限。只有用户名和密码匹配输入时，才可以得到访问权限。当输入不正确时，如图 10.10 所示。

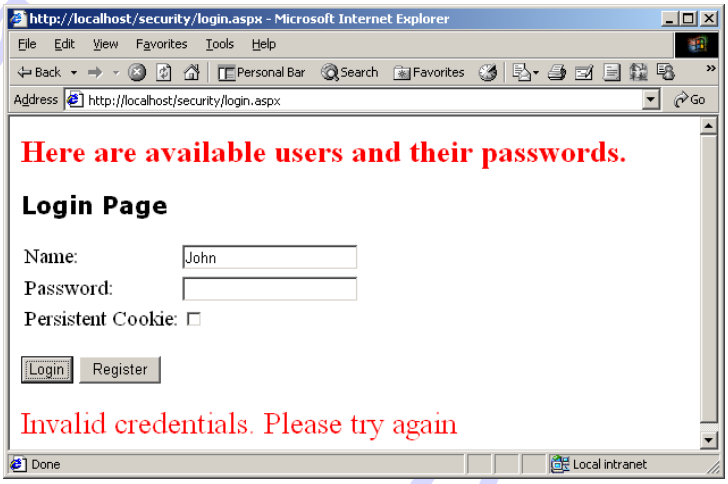


图 10.10　登录界面

会显示“Invalid credentials. Please try again.”不能登录得到应用程序的 default.aspx 页面。我们使用 XML 文档作为数据库来存储用户名和密码。这里有一个假设，服务器是安全的，尽管这不可能是绝对的。因为用户名和密码都是不经加密地存储在 Login.xml 文档中的。可以打开这个文件，得到所有的用户名和密码。

我们使用一个 XmlTextReader 类的实例指针 reader 访问 Login.xml 文档。通过这个指针返回的值与表单中得到值进行比较。

```
if(reader.MoveToContent() == XmlNodeType.Element
        && reader.Name == "Name")
        {
           //authenticate user's identity.
           if(reader.ReadElementString() == UserName.Value &&
        reader.ReadElementString() == UserPass.Value)
          {

FormsAuthentication.RedirectFromLoginPage(UserName.Value,
             PersistCookie.Checked);
                 Valid = true;   //validated the credentials,
          }
```

```
}
```

只有用户名和密码都匹配时，才会重定位到用户请求的 default.aspx 页面。这里我们使用了一点技巧。由于 Login.xml 文档中只有用户名和密码两种子标记，并且匹配的用户名和密码是相邻的，那么使用 ReadElementString 方法可以避免对访问指针的复杂操作。调用第一个 ReadElementString 方法得到用户名后，指针会自动移到下一个标记，在这里就是密码标记。如果不匹配的话，将读取下一个<name>标记，进行新的一组用户名和密码的匹配判断。如果匹配，那么重定位页面并将布尔变量 Valid 置为真。这样就不会提示确认错误了。

除了 FormsAuthentication.RedirectFromLoginPage 方法重定位页面以外，ASP.NET 还提供了 GetAuthCookie 和 SetAuthCookie 方法避免这种重定位网页。这种技术对于将登录部分嵌入网站首页的登录方式非常有用。

对于新用户，可以将登录页当成注册页，在用户名和密码处填写要注册的用户名（Smith）和相应的密码（Smith），单击 Register 按钮，会看到如图 10.11 所示界面。

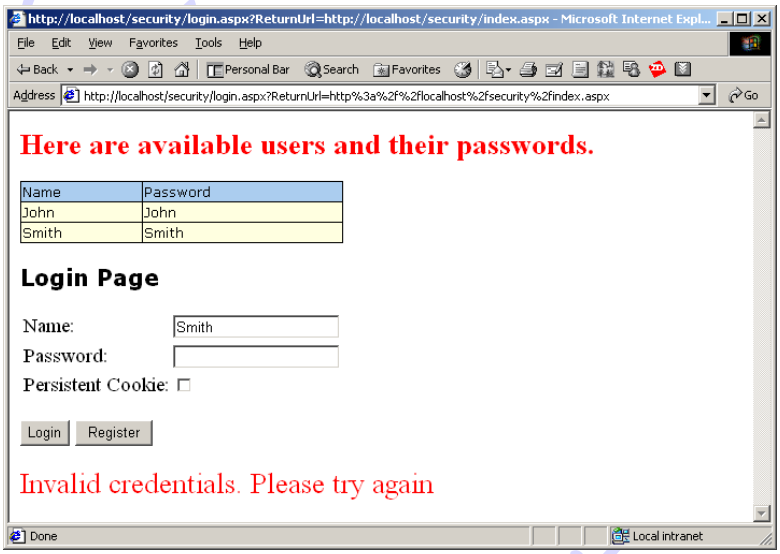


图 10.11　注册页面

这里，为了提示可用的用户名和密码，我们将 XML 文档中的数据公开写在页面上。这是完全没有必要的，这里只是方便说明。

提示：在这里，我们使用和用户名相同的字符串作为密码作例子，但是在实际应用中，这种弱密码是非常容易被破解的。一个足够复杂的密码应该包括数字、字母、标点和符号。弱密码会使所采取安全措施彻底失效。

此时，无论使用已有的用户名“John”，还是刚刚注册的用户名“Smith”都可以登录，由 ASP.NET 重定位到 default.aspx，如图 10.12 所示。

图 10.12

当用户单击“Signout”按钮退出时，我们使用 FormsAuthentication.SignOut 方法删除或宣布当前使用 cookie 失效，从而结束本次登录，然后由 Redirect 方法重定位到登录界面 Login.aspx，如图 10.10 所示。

```
void Signout_Click(Object sender, EventArgs E)
    {
      FormsAuthentication.SignOut();
      Response.Redirect("login.aspx");
    }
```

最后，我们介绍一下 cookie 的存在周期。注意到登录页面有一个单选框，如图 10.13 所示。

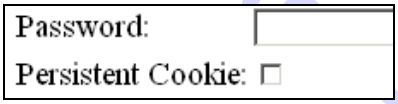


图 10.13　登录页面的单选框

登录时，勾选这个单选框表示使用持久 cookie 登录。这样做浏览器会保存登录的 cookie，并且每一次浏览器会话都会送回 cookie。不勾选这个单选框的话，cookie 只存在于当前的会话时段，关闭浏览器后 cookie 会丢失。但是，使用了 SignOut 方法会清除所有的 cookie，无论是不是持久 cookie。

基于表单的验证方式还可以根据有效的证书验证用户身份。这需要在 Web.Config 文件中加以设置。

```
<authentication>
    <credentials passwordFormat="SHA1" >
        <user name="Smith" password="GASDFSA9823598ASDBAD"/>
        <user name="John" password="ZASDFADSFASD23483142"/>
    </credentials>
</authentication>
```

应用程序调用 FormsAuthentication.Authenticate 方法，将证书代替用户名和密码，由 ASP.NET 确认证书。证书有三种存储形式：SHAI 形式、MD5 形式和 clear，即明文存储。

基于表单的验证方法很灵活。除了前面的验证方法以外，还可以在 Web.Config 文档的<authentication>配置段中通过子标记<deny>和<allow>来建立新的验证和授权规则。这种规则优先于用户名和密码的验证方法。

10.5　本章小结

本章内容较丰富，主要包括四个部分，一个是 ASP.NET 的全局应用文件 Global.asax；第二个是应用程序的 Web.config 文件；第三个部分则是 ASP.NET 的跟踪和调试；最后是 ASP.NET 的安全机制。

第 11 章　ASP.NET 的缓冲机制

本章将介绍 ASP.NET 技术的缓冲机制。我们先谈谈 ASP.NET 缓冲机制的全貌，以及新技术所具有的优势。同时引出 ASP.NET 的三种缓冲方法——输出缓冲、部分缓冲和数据缓冲。在 11.2 节中，将具体的介绍每一种缓冲方法。你将在新的缓冲技术与现有技术的对比中，体会到 ASP.NET 所体现的新的观念、新的思想。在本章的最后，作为前面说明的补充，我们将介绍一些 ASP.NET 直观认识缓冲技术的例子，以突出新技术的特性，并帮助你进一步理解缓冲机制。

11.1　缓冲的机制

缓冲（caching）技术在现在的网站建设中的作用显得相当重要。由于目前的技术大多是在客户端浏览器发出请求时，动态的生成可浏览网页，如静态 HTML 网页，然后将根据请求生成的网页传回浏览器。关于动态生成网页的优势、长处，想必你也有所了解。但是我们现在要考虑的是如何降低这种技术带来的对性能的更高要求。

这种动态生成技术的流行使得对网站的服务器性能要求较高，是因为需要实时的根据浏览器的请求生成网页，在生成复杂网页或是频繁生成相同的页面时，问题显得尤为严重。但是，这笔服务器的开销是可以通过将动态生成的页面保存起来的方法降低的。当用户下次请求生成网页时，只需将已保存的静态网页传给客户端浏览器而不需再生成新的相同的页面了。这样，降低了对服务器的性能要求；或是在相同配置下，降低了服务器的工作负荷。

这样的作法也有局限性，比如说，在用户请求与保存的页面完全相同的浏览页时，才能向浏览器传回已保存的网页。这样的方法处理生成如商品的介绍这类网页比较有效，因为网页的内容与浏览器的请求没有比较直接的关系，换句话说，对于不同的请求可能生成的页面是完全相同的浏览页面。

同时，将动态生成的静态页面保存在磁盘上，也不如放在内存中以获得更快的读取速度。事实上，ASP.NET 就是在内存中开辟缓冲区的。

在 ASP.NET 技术中，有三个缓存的方法分别为：页面输出缓存（Page Output caching）、页面部分缓冲（Page Fragment caching）和页面数据缓冲（Page Data caching）。

输出缓冲提供了对于请求的动态反映，一般用来存放完整的浏览页。当有浏览器请求时，将缓冲区中的页面传给浏览器。前面提到了在需要频繁的生成浏览页时，比如访问量极大的网站，将频繁访问的页面放入输出缓冲区中，无疑可以大大降低了处理频繁的生成请求的负担。对于放入缓冲区的页面，可以根据多个请求并行的传给浏览器。这样省去了大量的执行代码的时间，提高了工作效率。

如果将整个网页放入缓冲区是不现实的，我们可以采取第二种缓冲方法——部分缓冲。所谓部分缓冲是相对于在输出缓冲区中存放整个页面而言的。部分缓冲是将网页中复杂的、

难以生成的部分放入缓冲区，供不同的请求调用；而对于网页中一定要动态生成的部分单独处理，从而减少了要执行的代码的数量。在实际的应用中，采用部分缓冲常常是值得的。通常在部分缓冲的过程中，要先确定哪些部分是不用随着每个请求改变的，然后在创建一次整个网页之后，就可以将这些部分分离出来，放入部分缓冲区中，在一定的时间内供浏览器请求调用，超出这个时间，就要面临更新缓冲区的问题了。关于部分缓冲，请参阅下一节的介绍。

如何选择页面或是部分页面在缓冲区中停留的时间是很关键的。可选的方法也有很多，比如：可以定时更新缓冲区，即为缓冲区内的所有已生成的片段（网页）指定有效期。当超出有效期时，就更新缓冲区。在浏览器发出页面浏览请求时，也只需判断是否在有效期内，若在有效期内，则直接将输出缓冲区的指定内容，否则就需要重新生成这些请求的网页。

数据缓冲是将对象放入内存缓冲区中，由 ASP.NET 提供的一系列的类、属性和方法来控制这些缓冲区内的对象。

ASP.NET 支持文件和缓冲关键字相互依赖，允许开发人员通过外部文件或其他的缓冲内容建立自己的缓冲内容。

11.2　ASP.NET 的缓冲方式

ASP.NET 为网络应用程序提供了三种缓冲方式，利用在内存中开辟的缓冲区将难以动态生成的页面和访问频率较高的页面或是重要的内容放入缓冲区，当网络应用程序发出请求时，将保存的内容传给浏览器或是应用程序，从而降低了服务器的工作负荷。

先来看看怎样使用页面输出缓冲。

11.2.1　页面输出缓冲

页面输出缓冲是通过保存已生成的动态页面来提高请求和响应的吞吐量。页面缓冲是默认允许的，但是对于已给定的响应，输出是不加以缓冲的，除非是有直接的代码说明为响应提供页面输出缓冲。

要为响应提供输出缓冲，需指定缓冲区中内容的有效期和建立有访问权限的公有缓冲。这可以通过 OutputCache API 实现或是使用@OutputCache 指令来完成。当缓冲区建立以后，第一次的 GET 请求会为缓冲区开辟一个入口，以后的 GET 请求或是 HEAD 请求都利用这个入口来并行地利用输出缓冲的，只要缓冲区中留有请求的内容。

输出缓冲是利用有效期来管理缓冲区中的页面的。假设在缓冲区中有一个页面被标记上有效期并且有效期是 60 分钟，那么当这个页面在缓冲区中停留的时间超过 60 分钟时，就会被清理出缓冲区。如果此时请求这个页面，那么这个被清理出缓冲区的页面就会根据请求重新创建，然后又一次放入缓冲区，标记上新的有效期。这种类型的有效期就是绝对有效期，也就是页面在缓冲区中被保留的时间。

使用指令@OutputCache 声明有效期是：

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

其中，参数“Duration”是以秒为单位的，“Duration = ‘60’”指定了有效期是 60 秒。参数“VaryByParam”指明页面不会被 GET 或是 POST 指令改变。

声明有效期的方法还可以通过 HttpCachePolicy 类来实现。由 HttpResponse.Cache 属性可以得到关于缓冲内容的说明，比如有效期、可否被修改等。

如果在网页的开始使用@OutputCache 指令建立页面缓冲，ASP.NET 会使用 Page.InitOutputCache 方法将@OutputCache 指令转换为 HttpCachePolicy 类的方法来实现。

使用了页面输出缓冲的网页，在 GET 请求初始化网页时会将网页的响应加入到输出缓冲中去。只要在有效期内，输出缓冲可以并行的为 GET、POST 和 HEAD 请求输出所请求的网页。明确了@OutputCache 指令的参数“VaryByParam”的值后，响应 GET 请求的查询串得到的结果和 POST 请求得到的结果都是可以放入缓冲中的。

对于 GET 请求中关键字和取值成对匹配的请求，在参数值相同时可以不考虑参数的传递顺序，直接根据有效期来输出或创建位于输出缓冲中内容。只有当参数值有改变时，ASP.NET 才会认为请求的是不同的内容，然后调用不同的缓冲区中的页面或是创建新的页面，作为请求的响应。

设定有效期的方法还可以 HttpCachePolicy 类的方法来实现。我们改写前面的例子，通过 HttpCachePolicy 类来指定有效期，如下：

```
Response.Cache.SetExpires (DateTime.Now.AddSeconds (60));
Response.Cache.SetCacheability (HttpCacheability.Public);
```

ASP.NET 是通过 Response.Cache 方法来调用 HttpCachePolicy 类的方法的。当请求的页面超出了有效期时，会再次创建更改页面并为其指定新的有效期，然后放入页面输出缓冲中等待下次的请求。

页面输出缓冲的机制就是利用有效期这个概念，使用两种方法来实现根据有效期得到的操作。下面来看重要的类——HttpCachePolicy 类。

HttpCachePolicy 类提供一系列方法来指定 Http 头的关于缓冲的信息和 ASP.NET 中页面输出缓冲的控制。在本节中，我们将介绍与页面输出缓冲有关的方法。通过这些方法来控制输出缓冲。

SetExpires 方法用来确定有效期的时间。SetExpires 的参数是一个 Datetime 类的实例，一个表示时间的参数。比如：

```
Response.Cache.SetExpires (DateTime.Now.AddSeconds (60));
```

使用了 Datetime 类的方法“Now.AddSeconds”确定了有效期为 60 秒。又比如：

```
Response.Cache.SetExpires (DateTime.Now.AddHours (2));
```

指定了有效期为两个小时。类似的，还有 AddDays（天）、AddMinutes（分钟）、AddMonth（月）等方法分别用来根据不同的时间单位来确定有效期。

Datetime 还有其他的方法来指定有效期。

```
Response.Cache.SetExpires(DateTime.Parse("10:00:00AM"));
```

通过使用 Parse 方法将字符串“10:00:00AM”转换为本地时间，从而说明了有效期是到本地时间当天的上午 10 点之前。

SetCacheability 方法设定了 http 头的 Cache-Control 部分，用来说明文档在网络中的缓冲方式。下面是 SetCacheability 方法的调用方式：

参数是 HttpCacheablility 指定枚举值。比如：

```
Response.Cache.SetCacheability (HttpCacheability.Server);
```

HttpCacheablility 的取值一共四种：Server、Public、Private 和 NoCache。其中，Private 是默认值，指明请求的响应在客户端缓冲。Public 说明响应在客户端和代理服务器都是可缓冲的。Server 说明只能在最初的服务器缓冲。

SetSlidingExpiration 方法的参数是布尔值。当参数为真时，每一次请求都会更新 http 头 Cache-Control 部分。否则，设置将会被保留下来。

```
Response.Cache.SetSlidingExpiration(false);
```

例子中，参数是 false，所以 Cach-Control 部分不会被实时更新。最后，我们来看一个演示 ASP.NET 页面缓冲的例子。

程序清单 11-1：example-11-1.aspx

```
<%@ OutputCache Duration="20" VaryByParam= "none" %>

<html>
  <head>
    <title>输出缓冲示例</title>
  </head>
  <script language="C#" runat="server">

    void Page_Load(Object sender, EventArgs e)
    {
        NowTime.Text = "本页创建时间是" + DateTime.Now.ToString();
    }

  </script>

  <body>

    <center>
     <h3><font face="Verdana">使用输出缓冲的示例</font></h3>
    <p><p><p><p>
    <asp:label id="NowTime" runat="server"/>
    </center>

  </body>

</html>
```

我们设定有效期是 20 秒，这样在 20 秒内的请求都会得到缓冲区中的页面，所以图 11.1 和图 11.2 相同，而在 20 秒后，得到是新的页面了，如图 11.3。

注意：页面的生成时间是不同的。

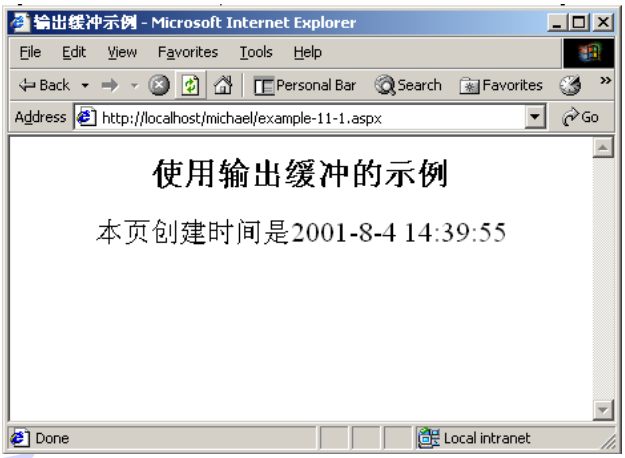


图 11.1

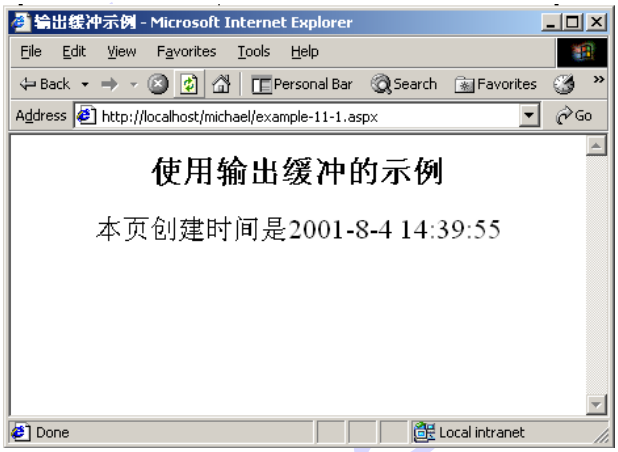


图 11.2

图 11.3

11.2.2　页面部分缓冲

在将网页整页地放入缓冲区变得不现实时，ASP.NET 提供了将网页的一部分放入缓冲区的方法，页面部分缓冲。事实上，页面部分缓冲也可以看成页面输出缓冲的一种。使用页面部分缓冲，首先要将网页中放入缓冲区的部分分离出来。在网页中使用 user control 来说明需要缓冲的部分，然后使用@OutputCache 指令标记出这些部分，为这些部分确定有效期等属性。

user control 部分的介绍，可以参考“网页表单”一章中的详细说明，下面我们来看看@OutputCache 指令的说明。

@OutputCache 指令一共有 6 个属性：

- Duration:以秒为单位，指定有效期的时间长度。这是必须的属性，因为任何输出缓冲中的内容都要指定有效期，部分缓冲也是一样。
- Location:取值限于枚举集 OutputCacheLocation 的 ANY、Client、Downstream、Server 和 None 五种枚举值，默认值是 ANY。当输出缓冲区中的 ASP.NET 页面和用户控件（user control）时，这个属性是必须的。

取 Client 值时，输出缓冲定位于发送请求的客户浏览器。取 Server 值时，定位于处理请求的服务器。取 Downstream 时，输出缓冲定位于处理请求的服务器的下传流。

如果取 ANY，那么上面三种情况都是允许的，取 None 则请求页面的输出缓冲没有被激活，也就是说没有输出缓冲可用。

- VaryByCustom：当 VaryByCustom 取“browser”值时，根据浏览器的名字和主要的版本信息改变缓冲区内容。如果属性值为自定义的字符串，就必须在应用程序的 Global.asax 文档中重定义 HttpApplication.GetVaryByCustomString 方法。
- VaryByHeader：根据 http 头信息来改变缓冲区内容。当有多重头信息时，输出缓冲中会为每个指定的 HTTP 头信息保存不同的页面文档。VaryByHeader 属性可以应用于缓冲所有 HTTP/1.1 的缓冲内容，不仅仅是 ASP.NET 的缓冲。

但是，VaryByHeader 属性不支持@OutputCache 指令对 user control 的标记。我们在这里只是提一下这个属性，已使你能获得对@OutputCache 指令的全面认识。

- VaryByParam：使用一个用分号间隔的字符串列表来改变输出缓冲。在默认的情况下，这些字符串是对应 GET 方法的查询串，或者是 POST 方法的参数。当属性设为多重参数时，输出缓冲会为每一个指定的参数保存其请求的文档。VaryByParam 属性可能的取值包括“none/*”和任何有效的查询串以及 POST 方法的参数名。

VaryByParam 属性允许 user control 的编写者控制 ASP.NET 缓冲储存位于服务器上的输出缓冲的多个实例。比如：

```
http://localhost/startpage.aspx?categoryid=foo&selectedid=001
http://localhost/startpage.aspx?categoryid=foo&selectedid=002
```

请求的是两个不同的用户控件，id 分别为 001 和 002。通过这种方法，可以选择所需的部分页面。

- VaryByControl：使用字符串来改变输出缓冲的内容，这些字符串是 user control 的

属性全名。VaryByControl 也不支持 ASP.NET 页面的@OutputCache 指令。

页面部分缓冲的过程是在创建网页表单时，使用 user control 来描述需要分离的部分。使用@OutputCache 指令将 user control 描述的部分标记为需放入缓冲区中的内容。

当浏览器请求该页面时，创建表单后，将需要缓冲的部分放入缓冲区。再次请求时，会判断请求页面中是否有缓冲区中的页面部分，如果判断结果是肯定的，那么将创建请求页面的其他部分，再加上缓冲区中的内容合成请求页面，作为请求的响应将页面传回浏览器。如果请求的页面在缓冲区中没有部分缓冲页面，那么将再次创建请求页面传回浏览器，并且再次根据@OutputCache 指令将 user control 部分放入缓冲区中。

对于如何判断请求的页面是否包括缓冲区中内容，可以使用前面介绍的属性，比如 VaryByParam 属性。

11.2.3　数据缓冲

除了上述的两种缓冲方法以外，ASP.NET 还提供一种强大的，易于操作的缓冲机制——数据缓冲。数据缓冲可以在缓冲区中为每个应用程序保存各种对象，这些对象可以根据 http 的请求被调用。缓冲区对于各个不同的应用程序来说是私有的。

在 ASP.NET 中，数据缓冲是通过 Cache 类来实现的。每个应用程序的缓冲区实际上就是 Cache 类的一个实例。每个应用程序的生存周期就是这个缓冲实例的生存周期。当应用程序重新运行时，Cache 类实例会重建。通过 Cache 类的方法，可以将数据对象放入缓冲区中，然后再根据关键字匹配寻找到并加以利用这些对象。

Cache 类提供了一个接口，通过这个接口可以完全控制要缓冲的内容。并且，可以规定缓冲的时间和方式。一个简单的添加缓冲对象的例子是：

```
Cache["keyname"] = keyValue;
```

然后重新找到这个对象，对这个对象的访问是：

```
keyValue = Cache["keyname"];
if(keyValue != null ) {
    DisplayData(keyValue);
}
```

在 ASP.NET 中，主要有三种数据缓冲方式：

（1）清除废物(Scavenging)

当系统的内存资源紧张时，将最不常用的和不重要的内容清除出缓冲区，把内存用来处理频繁访问的方法称为清除废物的缓冲方式。这种方法在缓冲技术中很常见，比如内存的缓冲（Cache）。这种方法保证了在缓冲区中的都是重要的和常用的内容。

使用这种方法需要在将对象放入缓冲区时，为这些对象标记优先级和必需的访问频率。当需要清除废物时，根据对象的优先级来判断哪些是不重要的内容，根据指定的访问频率判断哪些内容的访问频率达不到指定的频率，然后将这些内容清除出内存缓冲区。

实现上，在使用 Cache.Add 和 Cache.Insert 方法向缓冲区中加入对象时，可以指定其优先级。优先级一共有 6 种，由小到大依次为：Low、BelowNormal、Normal、AboveNormal、High 和 NotRemoveable。默认值是 Normal。

```
Cache.Insert("String", Example, null, DateTime.Now.AddSeconds(20),
TimeSpan.Zero, CacheItemPriority.AboveNormal,
```

```
    CacheItemPriorityDecay.Slow, onRemove);
```

这行代码指明了优先级是“AboveNormal”，关于 Insert 方法我们下面会讲到。

指定的访问频率一共是四种，依次为：Fast、Medium、Slow 和 Never，其中标为 Fast 的内容最容易被清出缓冲区。前面的例子中，标记的访问频率是“Slow”。

（2）有效期

在使用 Add 和 Instert 方法时，还可以为添加的内容标上有效期。通过有效期来判断哪些内容应该被清出缓冲区。这和输出缓冲的方法类似，超过有效期的内容将会被清除出去。Add 和 Insert 方法有两个参数来说明这个方法：

DateTime 指定了有效期的时间，可以是绝对时间，比如每天 1：00；也可以是相对时间，比如从创建开始的 30 秒。

TimeSpan 决定了创建对象的时间间隔。当对象被清出缓冲区时，所有对该对象的请求都会得到 null，直到对象再次创建并加入到缓冲区中。

有些数据是有规则地更新的，比如商品报价、比赛比分等，可以将有效期设为每次数据更新的时间间隔长度。对于一个网站，每天更新一次商品的报价，那么可以将报价的数据有效期设为一天，类似对于网站每 3 分钟采集、更新一次的比赛比分，其数据有效期是 3 分钟。通过有效期的方法管理缓冲区，可以有效地控制类似上述情况下的数据缓冲。

（3）关键字匹配

ASP.NET 允许根据外部文件、目录或其他的缓冲内容来确定要缓冲的内容。这样就造成了一种依赖关系，可以是依赖外部文件或是依赖关键字。当这种依赖关系改变时，缓冲区中的内容就是无效的，需要被清出内存。这种方法可以有效地更新缓冲区中的内容，尤其是当缓冲内容的数据源被修改时，使用这种方法可以根据这种依赖关系来更新缓冲内容，将修改前的数据作为无效内容清除出内存。

三种方法各有所长，你可以根据不同的需要来选择数据缓冲的方法。下面，我们详细介绍实现数据缓冲的类 Cache 类中的 Add 和 Insert 方法。

Add 方法的使用示例：

```
        Cache.Add(key, value, null, DateTime.Now.AddSeconds (15),t,
    CacheItemPriority.High, CacheItemPriorityDecay.Slow, onRemove);
```

我们来逐个的分析参数的含义：

- key　　缓冲内容的关键字，用来引用缓冲内容的关键字
- value　　要放入缓冲的内容
- dependencies　　依赖关系，当依赖关系发生改变，内容就会被标为无效的数据
- 绝对有效期　　这里有效期时间为 15 秒
- 时间间隔　　这里用变量 t 来表示间隔的时间
- 优先级　　这里取 CacheItemPritorty.BelowNormal，即优先级为 BelowNormal
- 访问频率　　这里的频率是“Slow”
- onRemove　　如果采用“onRemove”，那么当这个应用程序在缓冲区中的对象被清出缓冲区时，会通报应用程序。

Insert 方法也有和 Add 方法一样的调用形式，参数的含义也相同。除此之外，Insert 方法还有简单的调用形式：

```
Insert(key, value);
Insert(key, value , CacheDependency);
Insert(key, value, CacheDependency, DateTime, TimeSpan);
```

其中，CacheDependency 是指依赖关系；DateTime 是有效期的时间；TimeSpan 是创建对象的时间间隔。

最后让我们来演示一个输出缓冲的例子，先来看代码：

程序清单 11-2：-11-3.aspx

```
<%@ Import Namespace="System" %>
<%@ Import Namespace="System.IO" %>
<%@ Import Namespace="System.Data" %>
<%@ Import Namespace="System.Web.UI.WebControls" %>

<html>

 <script language="C#" runat="server">

  void Page_Load(Object Src, EventArgs E )
  {

    if(!IsPostBack)
    {
      LoadData();
    }
  }

  void NewRecordBtn_Click(Object sender, EventArgs E)
  {

    if(!Page.IsValid)
    {
        Msg.Text = "请将必要部分填写好";
        return;
    }

    DataSet ds = new DataSet();

    // open and read the XML document
    FileStream fs = new FileStream(Server.MapPath("Records.xml"),
                                   FileMode.Open,
       FileAccess.Read, FileShare.ReadWrite);
    StreamReader reader = new StreamReader(fs);
    ds.ReadXml(reader);
    fs.Close();

    // append a row
    try
    {
        DataRow newRecord = ds.Tables[0].NewRow();
        newRecord["UserID"] = UserID.Text;
       newRecord["Name"] = Name.Text;
       newRecord["Age"] =Age.Text;
       newRecord["Code"] = Code.Text;
       newRecord["Province"] = Province.Text;
```

```
            newRecord["Hobby"] = Hobby.Text;
            newRecord["Mail"] = Mail.Text;
            if(Sex_female.Checked)
                newRecord["Sex"] = Sex_female.Text;
            else
                newRecord["Sex"] = Sex_male.Text;
            ds.Tables[0].Rows.Add(newRecord);
        }
        catch(Exception)
        {
            CacheMsg.Text = "注册失败，请重试。";
            return;
        }

        // rewrite the data file
        fs = new FileStream(Server.MapPath("Records.xml"), FileMode
                                                          .Create,
            FileAccess.ReadWrite, FileShare.ReadWrite);
        TextWriter writer = new StreamWriter(fs);
        writer = TextWriter.Synchronized(writer);
        ds.WriteXml(writer);
        writer.Close();

        Cache.Remove("MyData");
        LoadData();
    }

    void RefreshBtn_Click(Object sender, EventArgs e)
    {
        LoadData();
    }

    void LoadData()
    {
        DataView Source = (DataView)Cache["MyData"];

        if(Source == null)
        {
            // read the data from the XML documents
            DataSet ds = new DataSet();

            FileStream fs = new FileStream(Server.MapPath
              ("Records.xml"),
                FileMode.Open,FileAccess.Read);
            StreamReader reader = new StreamReader(fs);
            ds.ReadXml(reader);
            fs.Close();

            Source = new DataView(ds.Tables[0]);

            // cache it for the same requests
            Cache.Insert("MyData", Source, new
                CacheDependency(Server.MapPath("Records.xml")));

            // we advertise that where the page come from
            CacheMsg.Text = "本页是生成的新页面";
        }
        else
        {
```

```
            CacheMsg.Text = "本页是缓冲区中的页面";
        }

        MyDataGrid.DataSource = Source;
        MyDataGrid.DataBind();
       }

</script>

<body>
   <form runat="server">
    <hr>

    <H3 align="center">
    <font face="Verdana">赶快为注册新成员！</font> 
    </H3>
    <H3 align="center">
     <asp:label id="Msg" runat="server" Font-Size="8"
     Font-Name="Verdana" ForeColor="red" Text="请认真填写以下注册信息
    "></asp:label>
    </H3>
    <p>

    <table align="center">
      <tr>
        <td>用户名:</td>
        <td><ASP:TextBox id=UserID  runat=server/></td>
        <td><ASP:RequiredFieldValidator ControlToValidate=
         "UserID" Display="Static" ErrorMessage="*" runat=
         server/></td>
      </tr>
      <tr>
        <td>姓名:</td>
        <td><ASP:TextBox id=Name  runat=server/></td>
        <td><ASP:RequiredFieldValidator ControlToValidate="Name"
          Display="Static" ErrorMessage="*" runat=server/></td>
      </tr>
      <tr>
        <td>性别:</td>
        <asp:RadioButton id=Sex GroupName="Sex">
        <td><ASP:RadioButton id=Sex_male Text="Male" GroupName=
        "Sex" runat=server/>
        <ASP:RadioButton id=Sex_female Text="Female" GroupName=
         "Sex" runat=server/></td>
      </tr>
      <tr>
        <td>年龄:</td>
        <td><ASP:TextBox id=Age  runat=server/></td>
        <td><ASP:RequiredFieldValidator ControlToValidate="Age"
          Display="Static" ErrorMessage="*" runat=server/></td>
      </tr>
      <tr>
        <td>籍贯:</td>
        <td><ASP:TextBox id=Province  runat=server/></td>
        <td><ASP:RequiredFieldValidator ControlToValidate=
        "Province" ErrorMessage="*" Display="Static" runat=
        "server"/></td>
      </tr>
```

```
            <tr>
              <td>身份证号码:</td>
              <td><ASP:TextBox id=Code  runat=server/></td>
              <td><ASP:RequiredFieldValidator ControlToValidate="Code"
               ErrorMessage="*" Display="Static" runat=server/></td>
            </tr>
            <tr>
              <td>e-mail:</td>
              <td><ASP:TextBox id=Mail  runat=server/></td>
              <td><ASP:RequiredFieldValidator ControlToValidate="Mail"
               ErrorMessage="*" Display="Static" runat="server"/></td>
            </tr>
            <tr>
              <td>我的爱好:</td>
              <td><ASP:TextBox id=Hobby  runat=server TextMode=
               "MultiLine" Rows="3" Columns="30" /></td>
              <td><ASP:RequiredFieldValidator ControlToValidate="Hobby"
               ErrorMessage="*" Display="Static" runat="server"/></td>
            </tr>

          </table>

          <p align="center">
          <asp:button Text="注册" OnClick="NewRecordBtn_Click" runat=
           server/>
          <asp:button Text="刷新" OnClick="RefreshBtn_Click" runat=
           server/>
          </p>

          <hr>

          <p>

          <i><asp:label id="CacheMsg" runat="server"/></i>

          </p>

          <h3><font face="Verdana">已注册的成员名单</font></h3>

          <ASP:DataGrid id="MyDataGrid" runat="server"
            Width="900"
            BackColor="lightyellow"
            BorderColor="black"
            CellPadding=3
            CellSpacing="0"
            Font-Name="Verdana"
            Font-Size="10pt"
            HeaderStyle-BackColor="#abcdef"
          />

        </form>
      </body>
    </html>
```

得到的效果如图 11.4 所示，初次注册页面是动态生成的。

图 11.4　初次注册页面

当单击刷新按钮时，由于请求的是相同的页面，所以将缓冲中的页面调出，传给浏览器，如图 11.5 所示。

图 11.5

当注册了新的成员后，将再次生成新的页面，如图 11.6 所示。

图 11.6

这个例子用到的 ASP.NET 比较简单，可以参考前面的介绍，也可以参考代码中的注释来读懂它。我们的目的是试验 ASP.NET 的缓冲机制，但是，如果将这个例子完善一下，那么一个注册系统就可以使用了。这个系统的数据库是 XML 文档，还可以通过数据接口来访问文档中的内容。

11.3　本章小结

对数据库有所了解的读者对缓冲这个概念一定不陌生，为了保证数据库的安全，数据库对数据的操作都有缓冲机制。同理，ASP.NET 中为了保证数据的安全，也引入了缓冲机制，极大的提高了安全性。

第 12 章　实现简单的分布式信息流支撑系统

在这一章里，我们将本书涉及的技术，包括 Web Forms、Web Services、XML 及其相关技术、以及.NET Framework 的基础架构贯穿起来，并结合现代电子商务技术的发展趋势，使用.NET Framework 来创建一个简单的分布式信息流支持系统。我们不仅会使用到本书前面提到的几乎所有先进技术，还会演示如何使用.NET 架构中对 MSMQ（Microsoft Message Queue）的支持，来为一个现代企业设计适应下一代互联网环境的业务流程的一部分，并提供一个简单而又强大的实现基础。

12.1　下一代互联网环境中的电子商务

诞生于 19 世纪 60 年代的互联网，在 90 年代开始得到迅猛发展，这主要归功于 Web 技术的诞生和浏览器的发明。将信息孤岛联结起来，这是人类多年来的美好愿望。计算机的发明，成为信息世界自从造纸术发明以来最伟大的信息物质载体；计算机网络的出现，使信息的共享成为可能；而 Web 技术和浏览器的出现，则使信息共享和流通的能力延伸到桌面。Web 和浏览器技术是如此易于使用，这种易用性使互联网成为商业世界至今最为重视的技术。

在互联网上发布和获取商业信息，这种做法在 90 年代的短短几年中迅速深入人心。早期的互联网环境的电子商务便是这样，充满了这种简单、快捷而又臃肿的做法。商人们将电子邮箱视为下一代的电话和信箱，开始盲目地往任何人的电子邮箱中派发广告。信息时代信息复制成本几乎为零，这在商务世界中无疑是一个好消息。当你使用电子邮件发送一千条广告时，你所付出的代价和发送一千万条是几乎相等的。精明的商人们很快发现了技术的好处，并且由于这两种技术的易用性，互联网用户人数的指数增长便具有了必要性和可能性，从而成为一种必然。然而，很快人们就发现，虽然每个人都几乎不费成本地发布商业信息，但效果并不如他们期望中的好。因为每个人都会不知不觉地将时间花在浏览他人不分青红皂白给你塞过来的信息上了。

20 世纪 90 年代末，IBM 提出和倡导的电子商务的概念开始风行。越来越多的企业在网站上售卖他们的产品，并且出现了以 Amathon、EBuy 等网站为代表的 B2C、B2B 形式的电子商务企业。然而，很快人们又发现，在 Web 上销售产品的阻力太大，在 Web 上提供服务时的赢收问题也是令人头疼的问题。在网络经济泡沫破灭后，这种探索电子商务的思路开始被否定。

在 IBM 提出的电子商务概念中，其实包括了 e-Business 和 e-Commerce 两种形式。除了广为人知的 e-Commerce 这种利用数字媒介来实现交易行为的方式外，还包括了供应链组成单元间的协作和企业内部的业务流程的信息化。电子商务技术带来的既有效率的提高、成本的降低，还有企业核心竞争力的增值。

在下一代互联网的开发中，对电子商务的支持深入成为互联网技术主要考虑的内容。

下一代的互联网中，XML 语言将成为通用的数据格式。首先，XML 是开放的跨平台标准。并且，它是基于文本的，因此 XML 工具的创建十分简单。另外，XML 具有强大的数据描述能力，它不仅可以描述层次型模式的数据，还可以描述关系型模式的数据并在这两者间相互转化，甚至可以描述二进制数据。使用 XML Schema，还可以定义和描述文档特有的复杂数据类型和文档规范（尤其是行业文档规范）。使用 XSLT 技术，可以使不同格式和规范的 XML 文档相互转换，XPath 和 XPointer 技术允许对 XML 文档进行高度定制的、精确的定位和搜索。因此，XML 语言十分适合在 Web 环境中进行商务信息的传递和处理。

在过去，我们已经有了 DCOM/COM+和 CORBA 等远程调用技术，来实现对远程机器上对象的引用、调用和机器间组件对象的传输。然而，现存的这些技术都必须基于特定的传输协议，很难穿过企业的防火墙，因此它们仅适用于耦合紧密的 Intranet 环境和较为紧密、稳定的 Extranet，而很难和任意合作企业的信息系统建立起联系来。在下一个电子商务时代，由于信息流通的加快，企业间的合作关系将更为松散、灵活，今天甲和乙合作，明天这种合作关系可能就结束了。由此带来的就是信息系统的松散耦合的需求。因此，SOAP 协议和 Web Services 应运而生。SOAP 协议基于标准的 HTTP 传输协议，因此可以透明地穿过合作企业的防火墙。基于 SOAP 协议的 Web Services 技术，采用 WSDL 语言描述自身，不仅可以描述服务的内容，还可以描述服务的调用规范，因此将有能力构造服务的搜索机制（例如正在完善和标准化中的 UDDI），企业通过搜索来获取可以提供某种服务的企业实体列表以及这些服务的具体描述和调用规范，然后根据调用规范来请求服务。

现在已经很容易想象下一代互联网环境中的一种电子商务情形：假设我是一家自行车制造商，我可能将自身的核心竞争力定位在传动系统的开发上，并在互联网上搜索提供轮胎的潜在合作伙伴。合作伙伴的查询服务、订购服务……将以 Web Services 的方式暴露给外界。通过 UDDI 等服务发现方式，我会找到若干家“轮胎订购服务”的提供者，并得到这些服务的描述信息。这些描述信息都是基于 XML 的，因此可以轻易地被人和电脑识别。通过信息系统的自定义规则或者通过责任人的判断，自行车制造商可以做出决策，并按照我们选取的供应商的 Web Services 的描述中提供的调用规范，由信息系统调用这些 Web Services。调用的过程仍然是基于 SOAP 和 XML 的。客户和供应商之间的数据交换使用基于行业规范或符合其他规范格式的 XML 文档，任何一方都可以轻易的将这些交易过程的文档进行转换，成为符合内部文档规范的 XML 文档，并通过 BizTalk Server 这样的系统来驱动这些电子文档形成信息流，在企业业务流程中流动。

由此可见，下一代互联网为电子商务提供了一个绝佳的基础设施。下一代的电子商务将是高度数字化和自动化的。

12.2　一个简化的报价和审核系统

12.2.1　简化的需求分析

现在，让我们来看看，如何使用.NET 技术，来设计和构建一个现代化的企业信息支持系统。

我们假想中的企业是一家生产和销售自行车部件（例如轮胎）的处于供应链中部的中

型企业，我们的合作伙伴主要是各个大型的自行车制造企业，因此我们主要接受大规模的订单。由于行业的特征，这些订单的交货周期往往较长。现在我们需要构造一个报价系统，来向客户提供报价服务。客户提交身份信息、订单需求，我们的企业会评估该订单并为客户提供交易的报价，由客户决定是否发出该订单。

首先设计我们的业务流程。电子商务技术给传统产业带来的冲击是全方位的。采纳电子商务解决方案往往要求对企业进行业务流程再造。在这个假想的例子中，我们设计业务的流程的原则是将决策的权力分散到各个部门，分散的权力往往起到相互制约的作用，从而使报价决策更加精确。我们采用数字手段来将这些部门紧密地联系起来，从而避免权力分散带来的低效决策。

我们设立客户代表，全权负责与客户的交互，客户代表根据个人的意见、经验和对客户的熟悉程度，向后台系统提供建议报价，如果报价通过，客户代表会进一步和客户接触疫促成交易的成功，如果报价未能通过，会负责修改建议报价并再次提交或放弃提交。审核人会负责对建议报价的评估，决定是否对该客户提供的报价的合理与否。

因为我们的客户相对固定，采用客户代表的机制可以更好的在交易行遵循企业的发展战略，从而为不同的客户提供个性化的报价。例如对于某些重要客户，客户代表可能会给出相对低的报价或者折扣来争取这份订单，因为与该客户的长久的合作关系往往比一两笔交易更能为企业带来长久的利益。对于某些企业，客户代表可以根据其信用程度、支付能力甚至生产能力，提供较高的报价，来限制这些企业的订单大小。

设立审核人是为了限制客户代表的权利。客户代表提交的报价必须在审核人权衡各种因素后才有可能通过。审核人对公司内部的信息拥有比客户代表更大的权利，例如可以获得客户关系部门对该客户的评估、获得产品的生产成本核算以及内部战略、政策等信息。审核人在收到客户代表提交的信息后，可以做出初步判断来决定通过报价与否，如果信息不足，审核人可以查询会计部门对相应产品的成本核算和客户关系部门对相应客户的评估信息，从而协助做出决策。

客户关系部门和会计部门是审核人的信息支持部门，负责接收相关信息的需求并做出回复。

这个订单报价审核流程可以由图 12.1 来表示。图中虚箭头表示对相关信息的请求，实箭头表示对请求的回复。客户代表（Client Representative）处理与若干客户的联系，分别用双向箭头表示这种联系过程。

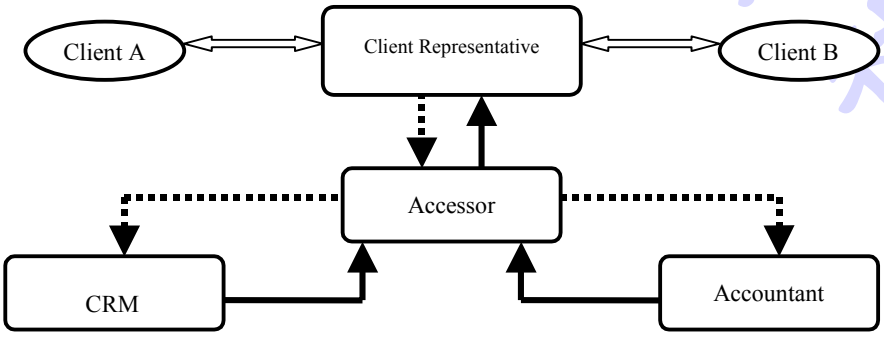


图 12.1

12.2.2　信息系统的基本架构

在我们的企业中，客户代表不是表示某个人，而是表示某个部门。客户询问报价时，可以查询产品的标准价格，这个工作可以由信息系统来自动答复，不须部门人员的干预。当客户希望订购产品时，需要查询该产品的具体报价；客户代表部门的人员按照负载均衡的原则处理这些特定的查询，根据客户身份和订单需求，做出具体的报价。这里所说的负载均衡是指多名客户代表同时接受和处理报价请求，客户提交的报价请求由当前空闲的，不在处理请求状态的客户代表接受。这种情况类似于银行中有多个业务柜台，客户们排在同一条队伍中，当某个柜台出现空闲时，排在最前端的客户便走向前，和柜台人员接洽。其他的客户会留在队伍中，等待下一个出现空闲的柜台。

审核人（Accessor）同样是指一个部门。在处理客户代表部门（CRM）提交的报价审核文件时，审核人不须也不应该关心这份报价是客户代表部门的哪一位成员提交的。审核人做的判断并不需要这个信息，审核人关心的应该只是客户是谁、客户关系部门对客户的评估是怎样的、相对于产品的成本和其他财政因素，这样的报价是否合适。

因此，我们将客户代表和审核人之间的信息沟通渠道规划为两个消息队列，客户代表部门的同事们将其报价建议放入其中一个消息队列，审核部门的人员同样按照负载均衡的原则从队列中提取请求并处理请求，在处理后将对请求的反馈信息放入另一个消息队列，让客户代表人员在他们认为适当的时候从队列中提取这些信息。这样就可以避免客户代表一方的系统对审核部门的某个系统的直接调用，降低系统的耦合程度。另外，使用消息队列的方式还可以使客户代表和审核人的工作异步化，客户代表可以提交某个请求后，接着接受另一个代表的请求或者处理以前提交的请求的反馈信息，而不必停下来等待审核人对当前事务的反馈。这无疑可以大大提高客户代表部门的客户响应度。

在审核人与两个支持子系统之间，同样存在请求信息的消息和回馈信息的消息。我们在审核人和客户关系部门同样设立两个消息队列来实现异步的消息请求。在审核人和成本核算部门（往往是会计部门（Account）的子部门）之间，由于对成本信息的查询通常不须人员干预，审核人往往不必等待很久时间，因此我们可以使用同步的远程方法调用方式。

审核人对成本信息的查询是必须的，因为这个操作开销较小，并且往往是审核人做出决策的必要信息。而对客户关系的信息查询则是可选的，因为这个查询操作开销往往较大，而且许多情况下审核人只需成本信息就可作出决策。

12.2.3　系统的.NET 实施规划

现在，我们希望将这个信息流支撑系统的简化版本用.NET 平台技术来实现。

在消息队列方面，我们无疑会使用 Microsoft 公司的 MSMQ 产品，以便充分利用.NET 平台对该产品的充分支持。

我们会将客户代表与客户的界面做成 Web Service，以便客户可以使用各种终端来获取我们的报价服务。为了方便浏览器用户，我们也会实现一个 Web Forms 界面。

客户通过访问前端的 Web Service 来获取标准的报价信息，这个过程因为是简单的查询，不须人手干预，所以可以是同步的，查询结果会以 Web Service 调用返回值的方法回传

给客户端。定制的报价涉及的部门更多，并且企业内部可能经过多次协商才会得到一个内部可以接受的报价，这个回馈时间是不定的，因此我们会使用异步的方式来应答，例如由客户代表使用 E-mail 来通知客户我们可以提供的优惠价格。客户代表会将报价建议发送到一个消息队列中，我们称队列名称为 Proposal；对建议的回馈则会被发送到 ProposalResponse 消息队列中。

标准的报价信息和产品的成本信息由会计部门提供，标准报价可以实现成 Web Service，供客户和审核人等查询，Web Service 的后端可以直接访问会计部门的数据库；成本信息是敏感信息，在这个系统涉及的部门中，我们只希望审核人可以获取该信息，因此会在 Web Service 接口中加入身份判断。另外，这个 Web Service 还会提供一个查询产品清单的接口方法，以便在前端的 Web Forms 页面中向客户展示所有的产品。

CRM 系统同样是一个审核人决策的信息支撑平台。由于客户关系信息的评估往往需要一段时间，我们同样使用消息队列来传输审核人的请求和对请求的答复。审核人审核某一报价建议时，如果需要客户关系信息，可以向消息队列发送请求消息，我们将这个队列称为 ClientInfoRequest。因为审核人是一个部门，我们有多个审核人员来处理这些报价建议，为了业务的完整性，在向客户关系部门发送请求后对该请求的答复必须能够回到发出请求的审核人员桌面上。为此，在向 CRM 系统发送的请求消息包中必须添加上标示审核人员身份的信息。CRM 系统关心的只是待查询客户的身份，而不关心审核人员的身份信息。在 CRM 系统找到客户的评价后，就将回复信息直接放置到 Proposal 消息队列中。

审核人平台较为复杂，我们会将它实现成一个 Web Forms，每个审核人员在各自的浏览器上处理报价建议。在处理这些建议时，我们将让审核人员主动地从 Proposal 消息队列中“拉”出报价建议。这是因为审核人员考虑每单报价建议所需要的时间往往是变化不定的，让他们在处理完一个建议后主动去提取下一个建议比定时的从消息队列中提取信息更为合理。

审核人员提取的报价建议的直接来源有两种情况：

- 报价建议来自客户代表，还没有处理过
- 报价建议来自 CRM 系统，它曾被处理过，由于需要客户信息而转发到 CRM 系统并被重新发送到 Proposal 队列中。

审核人员在这个 Web Form 上所作的操作会有三种：

- 缺乏足够的客户信息来作出判断。审核人员会单击按钮来向 ClinetInfoRequest 消息队列发送请求。如果客户信息已包含在请求消息包中，这个操作显然不可能被执行。
- 决定接受该建议。会单击页面上的某一个按钮来将消息发送到 ProposalResponse 消息队列中。
- 决定否决该建议。可以单击另一个按钮来将消息发送给 ProposalResponse 队列。同时可以在页面上填写文本来通知客户代表否决建议的原因，并可以提出提高或降低报价的建议。但审核人不应该直接告诉客户代表他认为合适的报价。

最后但也是极为重要的一点是，我们在各子系统之间传递的消息采用 XML 格式，都

是 XML 格式的字符流。

至此为止，我们已经有了一个使用.NET 平台来实施这个报价系统的蓝图。我们会在下一节给出一个简化系统的代码。

12.2.4　实现一个简化的信息流支撑系统

现在，让我们一步一步构建一个简化的信息流支撑系统。我们要构建上述的报价系统的简化版本，会使用到的.NET 技术包括：

- 构造客户交户界面和审核人工作平台的 ASP.NET Web Forms
- 构建接受客户报价、内部成本查询等请求的 ASP.NET Web Services
- 构建、发送和接收异步消息的 MSMQ 消息队列
- 后端的 ADO.NET 数据层支持
- 页面身份验证
- User Server Control

从技术人员的角度，这些技术看来很动人，是不是？不过在接触这个系统的架构代码之前，请牢记一点：企业电子商务信息系统的设计和实施并不能从技术的角度来考虑，而应该将企业的战略规划、业务流程和核心竞争力放在设计时桌面最显眼的位置上，考虑如何达到最适当的效率、技术和成本的组合，而不可片面追求信息系统的技术先进程度。

1. 设计子系统接口

对于会计部门，实施最为简单：实现一个 Web Service 并提供下列几个接口方法：

- 获取产品清单 GetProductList
- 查询产品标准报价 GetProductStandardPrice
- 查询产品成本 GetProductCost
- 为了辅助其他内部系统，我们还提供一个由产品 ID 获取产品名称的接口方法 GetProductName

这个 Web Service 的定义如下：

程序清单 12-1：ProductService.asmx

```
using System.Data;
using System.Web.Services;
public class ProductService
{
    [WebMethod]
    public string GetProductName(string ProductID)
    {
    }
    [WebMethod]
    public Decimal GetProductStandardPrice(string ProductID)
    {
    }
    [WebMethod]
    public Decimal GetProductCost(string ProductID, string
     AccessorID)
    {
```

```
    }
        [WebMethod]
        public DataSet GetProductList()
        {
        }
    }
```

客户代表部门暴露到企业外部的 Web Service 至少应具有下列接口方法：

- 查询标准报价 QueryStandardPrice
- 查询特定订单报价 QueryDetailPrice

其 Web Service 定义为：

程序清单 12-2：ClientRepresentative.asmx

```
public class ClientRepresentative : WebService
{
    [WebMethod]
    public Decimal QueryStandardPrice(string ProductID)
    {
    }
    [WebMethod(EnableSession=true)]
    public void QueryDetailPrice(string ProductID,
                                 int Quantity,
                                 string UserID,
                                 DateTime ShipDate)
    {
    }
}
```

2. 设计消息格式

我们接着设计在系统运作过程中各个子过程的 XML 消息格式。这些 XML 消息会被以字符串流的格式放入消息队列。消息的格式直接影响到子系统间的沟通。

在客户通过客户代表的 Web Service 发出请求后，客户代表 Web Service（ClientRepresentative.asmx）会集中客户提交的信息来生成消息包。消息包的格式用 XSD Schema 表示可以是下面的形式：

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2000/10/XMLSchema">
    <xsd:element name="PriceProposal" type="PriceType">
        <xsd:annotation>
            <xsd:documentation>write something if you like to
             .</xsd:documentation>
        </xsd:annotation>
    </xsd:element>
    <xsd:complexType name="PriceType">
        <xsd:sequence>
            <xsd:element name="Product">
                <xsd:complexType>
                    <xsd:simpleContent>
                        <xsd:extension base="xsd:string">
                            <xsd:attribute name="ProductID" type=
                             "xsd:string" use="Required"
                            />
                        </xsd:extension>
                    </xsd:simpleContent>
                </xsd:complexType>
```

```
                    </xsd:element>
                    <xsd:element name="Quantity" type=
                     "xsd:nonNegativeInteger"/>
                    <xsd:element name="UserID" type="xsd:string"/>
                    <xsd:element name="ShipDate" type="xsd:string"/>
                    <xsd:element name="ProposalPrice">
                        <xsd:complexType>
                            <xsd:sequence>
                                <xsd:element name="Comment" type=
                                 "xsd:string" minOccurs="0"/>
                            </xsd:sequence>
                            <xsd:attribute name="currency">
                                <xsd:simpleType>
                                    <xsd:restriction base="xsd:string">
                                        <xsd:enumeration value="dollar"/>
                                        <xsd:enumeration value="RMB"/>
                                        <xsd:enumeration value="pound"/>
                                    </xsd:restriction>
                                </xsd:simpleType>
                            </xsd:attribute>
                            <xsd:attribute name="price" type=
                             "xsd:nonNegativeInteger"/>
                        </xsd:complexType>
                    </xsd:element>
                </xsd:sequence>
                <xsd:attribute name="RepresentativeID" type=
                 "xsd:string"/>
            </xsd:complexType>
        </xsd:schema>
```

一个示范的消息包会是这样：

```
        <?xml version="1.0" encoding="UTF-8"?>
        <PriceProposal RepresentativeID="RepresentativeSessionID">
            <Product ProductID="021-2651">Product Name</Product>
            <Quantity>10000</Quantity>
            <UserID>Universal Bicycle</UserID>
            <ShipDate>2002-2-1</ShipDate>
            <ProposalPrice currency="dollar" price="3570000">
                <Comment><Comment>
            </ProposalPrice>
        </PriceProposal>
```

根元素名称表示这个消息包的作用，RepresentativeID 用来标示具体的客户代表人员，以便审核部门的答复可以返回到提出该报价建议的人手中从而跟进业务。根元素以下的各层子元素的作用如其标记名所提示的。

审核部门在接受到这样的信息后，如果无法做出判断，可以发出下面格式的 XML 消息作为客户评估请求：

```
        <?xml version="1.0" encoding="UTF-8"?>
        <xsd:schema xmlns:xsd="http://www.w3.org/2000/10/XMLSchema"
         elementFormDefault="qualified">
            <xsd:element name="ClientInfo" type="ClientInfoType">
                <xsd:annotation>
                    <xsd:documentation>write something if you like
                     to .</xsd:documentation>
                </xsd:annotation>
            </xsd:element>
            <xsd:complexType name="ClientInfoType">
```

```
                <xsd:sequence>
                    <xsd:element name="Product">
                        <xsd:complexType>
                            <xsd:simpleContent>
                                <xsd:extension base="xsd:string">
                                    <xsd:attribute name="ProductID" type=
                                     "xsd:string"/>
                                </xsd:extension>
                            </xsd:simpleContent>
                        </xsd:complexType>
                    </xsd:element>
                    <xsd:element name="Quantity" type=
                     "xsd:nonNegativeInteger"/>
                    <xsd:element name="UserID" type="xsd:string"/>
                    <xsd:element name="ShipDate" type="xsd:string"/>
                    <xsd:element name="ProposalPrice">
                        <xsd:complexType>
                            <xsd:sequence>
                                <xsd:element name="Comment" type=
                                 "xsd:string" minOccurs="0"/>
                            </xsd:sequence>
                            <xsd:attribute name="currency">
                                <xsd:simpleType>
                                    <xsd:restriction base="xsd:string">
                                        <xsd:enumeration value="dollar"/>
                                        <xsd:enumeration value="RMB"/>
                                        <xsd:enumeration value="pound"/>
                                    </xsd:restriction>
                                </xsd:simpleType>
                            </xsd:attribute>
                            <xsd:attribute name="price" type=
                             "xsd:nonNegativeInteger"/>
                        </xsd:complexType>
                    </xsd:element>
                </xsd:sequence>
                <xsd:attribute name="RepresentativeID" type=
                 "xsd:string"/>
                <xsd:attribute name="AccessorID" type="xsd:string"/>
                <xsd:attribute name="status" type="xsd:string"/>
            </xsd:complexType>
        </xsd:schema>
```

可以发现，这个消息包的格式和上一个包的格式是十分接近的，仅仅加入了 AccessorID 属性来标示审核人员，这和 RepresentativeID 属性的作用是类似的。其他的改变便只有用来表示包用途的根元素的名称，和与此匹配的 status 属性。在发送客户信息请求时，这个 status 属性的值是 Request。虽然我们不必把订单信息传递给 CRM 系统，但传递了也无妨。相反，如果过滤掉上一个包中的订单信息，则为了在客户信息返回时能够处理它，必须在将订单信息保留在审核人员的事务空间中，这会给我们的系统带来小小的复杂性。为了简化问题，我们把包几乎是整个的转发给 CRM 系统。

上一个包如果经过转发，会变成大概这样子：

```
        <ClientInfo RepresentativeID="RepresentativeSessionID"
    AccessorID="AccessorSessionID" status="Request">
            <Product ProductID="021-2651">Product Name</Product>
            <Quantity>10000</Quantity>
            <UserID> Universal Bicycle </UserID>
            <ShipDate>2002-2-1</ShipDate>
```

```
            <ProposalPrice currency="dollar" price=”3570000”>
                <Comment></Comment>
            </ProposalPrice>
        </ClientInfo>
```

返回时，CRM 系统只需在收到的请求包中加入客户的信息，并修改 status 属性为 Response 即可。例如上个请求包的回复包看起来是这样：

```
        <ClientInfo RepresentativeID="RepresentativeSessionID"
    AccessorID="AccessorSessionID" status="Response">
            <Product ProductID="021-2651">Product Name</Product>
            <Quantity>10000</Quantity>
            <UserID> Universal Bicycle </UserID>
            <ShipDate>2002-2-1</ShipDate>
            <ProposalPrice currency="dollar" price=”3570000”>
                <Comment></Comment>
            </ProposalPrice>
            <Archives>
                <Grade>normal</Grade>
                <Record>
                    <Deal ID="021-2311">
                        <Product>Pen</Product>
                        <Quantity>12500</Quantity>
                        <date>2001-5-6</Date>
                    </Deal>
                </Record>
                <Consultation>This is our second cooperation.
                 </Consultation>
            </Archives>
        </ClientInfo>
```

用 XML Schema 表示如下：

```
        <?xml version="1.0" encoding="UTF-8"?>
        <xsd:schema xmlns:xsd="http://www.w3.org/2000/10/XMLSchema"
         elementFormDefault="qualified">
            <xsd:element name="ClientInfo" type="ClientInfoType">
                <xsd:annotation>
                    <xsd:documentation>write something if you like
                     to .</xsd:documentation>
                </xsd:annotation>
            </xsd:element>
            <xsd:complexType name="ClientInfoType">
                <xsd:sequence>
                    <xsd:element name="Product">
                        <xsd:complexType>
                            <xsd:simpleContent>
                                <xsd:extension base="xsd:string">
                                    <xsd:attribute name="ProductID" type=
                                     "xsd:string"/>
                                </xsd:extension>
                            </xsd:simpleContent>
                        </xsd:complexType>
                    </xsd:element>
                    <xsd:element name="Quantity" type=
                     "xsd:nonNegativeInteger"/>
                    <xsd:element name="UserID" type="xsd:string"/>
                    <xsd:element name="ShipDate" type="xsd:string"/>
                    <xsd:element name="ProposalPrice">
                        <xsd:complexType>
                            <xsd:sequence>
```

```
                        <xsd:element name="Comment" type=
                         "xsd:string" minOccurs="0"/>
                    </xsd:sequence>
                    <xsd:attribute name="currency">
                        <xsd:simpleType>
                            <xsd:restriction base="xsd:string">
                                <xsd:enumeration value="dollar"/>
                                <xsd:enumeration value="RMB"/>
                                <xsd:enumeration value="pound"/>
                            </xsd:restriction>
                        </xsd:simpleType>
                    </xsd:attribute>
                    <xsd:attribute name="price" type=
                     "xsd:nonNegativeInteger"/>
                </xsd:complexType>
            </xsd:element>
<!-- 以下为新加入部分-->
            <xsd:element name="Archives">
                <xsd:complexType >
                    <xsd:sequence>
                        <xsd:element name="Grade" type=
                         "xsd:string"/>
                        <xsd:element name="Record">
                            <xsd:complexType>
                                <xsd:choice maxOccurs=
                                 "unbounded">
                                    <xsd:element name="Deal">
                                        <xsd:complexType>
                                            <xsd:sequence>
                                                <xsd:element
                                                 name="Product"
type="xsd:string" />
                                                <xsd:element
                                                 name="Quantity"
type="xsd:nonNegativeInteger" />
                                                <xsd:element name=
                                                "Date" type="xsd:string" />
                                            </xsd:sequence>
                                            <xsd:attribute name="ID"
                                             type="xsd:string" />
                                        </xsd:complexType>
                                    </xsd:element>
                                </xsd:choice>
                            </xsd:complexType>
                        </xsd:element>
                        <xsd:element name="Consultation" type="xsd:
                         string" />
                    </xsd:sequence>
                </xsd:complexType>
            </xsd:element>
<!--到此为止-->
        </xsd:sequence>
        <xsd:attribute name="RepresentativeID" type="xsd:
          string"/>
        <xsd:attribute name="AccessorID" type="xsd:string"/>
        <xsd:attribute name="status" type="xsd:string"/>
    </xsd:complexType>
```

```
</xsd:schema>
```

与会计部门的信息请求和答复是使用同步的 ASP.NET Web Service 技术实现的。尽管这其中仍是使用 XML 进行传输，我们不必考虑这个细节，ASP.NET 会自动为我们序列化数据。

审核人对客户代表的答复包格式和客户代表的请求包仍是极为类似的。客户代表不须从审核人处了解客户的信息（如果需要，我们可以用同样的方法为客户代表开辟一条和 CRM 部门对话的消息通道，例如消息队列或者 Web Service）。因此审核人会过滤掉 CRM 答复包中的客户信息，增加审核信息。例如，如果拒绝给予客户 Universal Bicycle 的 10000 单位 021-2651 产品 3570000 dollars 的报价，建议客户代表提出更高的报价时，答复消息包可能是下面的 XML 文档：

```
<PriceProposal RepresentativeID="RepresentativeSessionID">
    <Product ProductID="021-2651">Product Name</Product>
    <Quantity>10000</Quantity>
    <UserID>Universal Bicycle</UserID>
    <ShipDate>2001-7-8</ShipDate>
    <ProposalPrice currency="dollar" price="35700">
        <Comment></Comment>
    </ProposalPrice>
    <Decision>
        <Result accept="false" recommendation="higher" />
        <Comment></Comment>
    </Decision>
</PriceProposal>
```

其 Schema 格式为：

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2000/10/XMLSchema"
elementFormDefault="qualified">
    <xsd:element name="PriceProposal" type="PriceProposalType">
        <xsd:annotation>
            <xsd:documentation>write something if you like to
             .</xsd:documentation>
        </xsd:annotation>
    </xsd:element>
    <xsd:complexType name="PriceProposalType">
        <xsd:sequence>
            <xsd:element name="Product">
                <xsd:complexType>
                    <xsd:simpleContent>
                        <xsd:extension base="xsd:string">
                            <xsd:attribute name="ProductID" type=
                             "xsd:string"/>
                        </xsd:extension>
                    </xsd:simpleContent>
                </xsd:complexType>
            </xsd:element>
            <xsd:element name="Quantity" type=
             "xsd:nonNegativeInteger"/>
            <xsd:element name="UserID" type="xsd:string"/>
            <xsd:element name="ShipDate" type="xsd:string"/>
            <xsd:element name="ProposalPrice">
                <xsd:complexType>
                    <xsd:sequence>
                        <xsd:element name="Comment" type=
```

```
                    "xsd:string" minOccurs="0"/>
                </xsd:sequence>
                <xsd:attribute name="currency">
                    <xsd:simpleType>
                        <xsd:restriction base="xsd:string">
                            <xsd:enumeration value="dollar"/>
                            <xsd:enumeration value="RMB"/>
                            <xsd:enumeration value="pound"/>
                        </xsd:restriction>
                    </xsd:simpleType>
                </xsd:attribute>
                <xsd:attribute name="price" type=
                 "xsd:nonNegativeInteger"/>
            </xsd:complexType>
        </xsd:element>
        <xsd:element name="Decision">
            <xsd:complexType>
                <xsd:choice maxOccurs="unbounded">
                    <xsd:element name="Result">
                        <xsd:complexType>
                            <xsd:sequence>
                                <xsd:element name="Comment"
                                 minOccurs="0"/>
                            </xsd:sequence>
                            <xsd:attribute name="accept" type=
                             "xsd:boolean"/>
                            <xsd:attribute name=
                             "recommendation">
                                <xsd:simpleType>
                                    <xsd:restriction base=
                                     "xsd:string">
                                        <xsd:enumeration value=
                                     "higher"/>
                                        <xsd:enumeration value=
                                     "lower"/>
                                    </xsd:restriction>
                                </xsd:simpleType>
                            </xsd:attribute>
                        </xsd:complexType>
                    </xsd:element>
                </xsd:choice>
            </xsd:complexType>
        </xsd:element>
    </xsd:sequence>
    <xsd:attribute name="RepresentativeID" type=
     "xsd:string"/>
  </xsd:complexType>
</xsd:schema>
```

3. 构建消息队列

为了使用 MSMQ 进行消息传递，必须在开发平台和运行平台上安装 MSMQ。为了测试这个微型的系统，我们只需用到本地私有队列。我们以 Windows 2000 平台为例说明如何在.NET 应用程序中使用消息队列。

消息队列是 Windows 2000 操作系统的通讯基础，安装 Windows 2000 时不默认安装消息队列，因此如果在安装 Windows 2000 时没有安装 MSMQ，则必须手动安装。消息队列软件包括多个组件，其中消息队列服务器和 MSMQ Exchange 连接器仅在 Windows 2000

Server 和 Windows 2000 Advanced Server 平台上可用，而消息队列独立客户和从属客户在 Windows 2000 Professional 和前面两个平台上都是可用的。

安装消息队列组件需要有相应的权限，安装步骤和安装时的注意事项，可以参阅 Windows 2000 手册。

在安装好消息队列组件后，就必须使用消息队列管理工具来为我们的开发项目配置消息队列。我们主要的工作是建立程序代码中需要使用的消息队列。

为了建立本地私有队列，打开“计算机管理”单元。可以通过右击桌面上“我的电脑”图标并选择“Management”（管理）菜单来打开“计算机管理”单元。

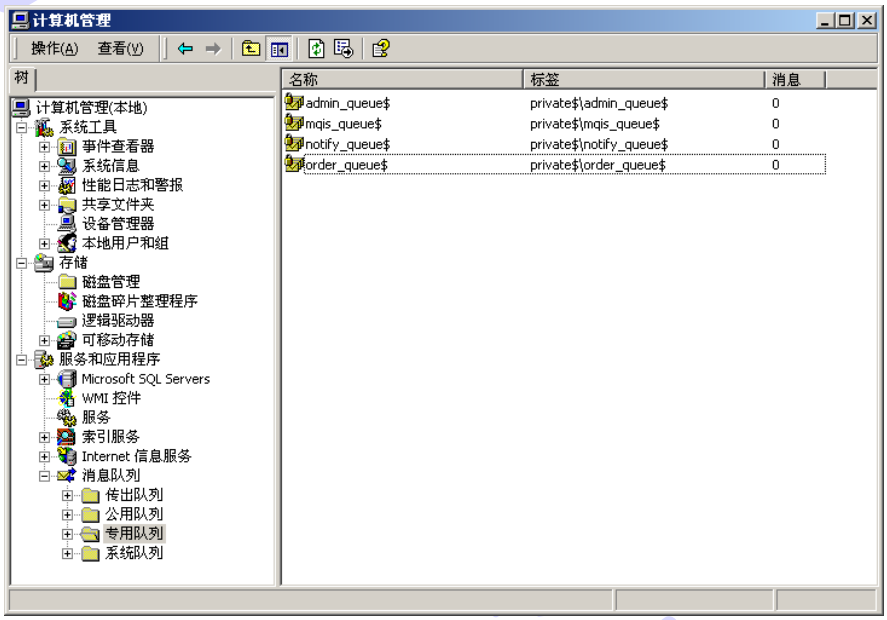


图 12.2

选择“服务和应用程序”下的“消息队列”，定位到“专用队列”。

然后右击左窗格中“专用队列”，在弹出的菜单中选择“新建”→“专用队列”，并键入“Proposal”来建立客户代表发送报价建议消息的队列。

重复上一步工作，建立前面提到的其他队列。

然后我们建立一个通用模块，封装队列的操作：

程序清单 12-3：Common.cs

```
using System;
using System.Messaging;

public class Common
{
    public static MessageQueue GetQueue(string path)
    {
        MessageQueue queue;
        if (MessageQueue.Exists(path))
            queue = new MessageQueue(path);
        else
```

```
                queue = MessageQueue.Create(path);
            return queue;
        }

        public static string PopQueue(string path)
        {
            MessageQueue q = GetQueue(path);
            q.Formatter = new XmlMessageFormatter(new string[]
             {"System.String"});
            return (string) q.Receive().Body;
        }

        public static void PushQueue(string path, string message)
        {
            MessageQueue q = GetQueue(path);
            q.Formatter = new XmlMessageFormatter(new string[]
             {"System.String"});
            q.Send(message);
        }
    }
```

这里我们实现了三个静态函数。PopQueue 和 PushQueue 分别用来向特定的消息队列发送和提取消息，GetQueue 函数首先确认系统中是否存在指定的队列，如果是，则返回表示该队列的一个实例，否则就创建一个队列并返回表示新队列的实例。

这里使用到.NET Class Library 对 MSMQ 的支持类 MessageQueue、Message 和 XmlMessageFormatter 对象。MessageQueue 对象具有许多静态方法，例如 GetQueue 里用到的 Exists 和 Create 方法。前者判断系统中是否存在某个消息队列，后者创建一个消息队列。MessageQueue 对象还包括例如 Send 和 Receive、Peek 等实例方法，Send 用来向 MessageQueue 对象所表示的队列发送消息，Receive 和 Peek 用来从队列中提取消息，两者的不同之处在于，Receive 提取消息后消息会从消息队列中删除，Peek 方法则保留提取出来的消息。因此，如果重复使用 Receive 方法，提取到的消息是不同的，而 Peek 方法则会提取到同一条消息。

在.NET 中，队列消息用 Message 对象来表示，该对象的 Body 属性表示了消息的内容。我们使用 Send、Receive 和 Peek 方法时，发送和提取到的都是 Message 对象。在这段代码中，我们使用 Send 方法时仅传递一个 string 类型变量，Send 方法具有一个重载版本，会先用 string 变量作为 Body 属性构造一个 Message 对象，再调用 Send 方法的标准版本将该对象发送出去。我们收发的都是简单的 string 类型（是一个 XML 文档的字符串表示），因此使用这个简单的重载版本。在 Receive 时，得到的是一个 Message 对象，因此我们使用 Unboxing 将 Body 对象转化为字符串后返回：

```
return (string) q.Receive().Body;
```

MessageQueue 对象还有一个重要属性 Formatter，该属性用来指明如何对消息中的对象进行编码和序列化。缺省来说，MessageQueue 使用 XmlMessageFormatter 对象来序列化 Body 属性中的消息。下面的代码创建了一个 XmlMessageFormatter 对象，使该对象按 String 类型来编码信息，并赋给 MessageQueue 对象 q 的 Formatter 属性，让 q 在传输消息时使用这个 XmlMessageFormatter 对象来完成序列化工作。

```
q.Formatter = new XmlMessageFormatter(new string[]{"System.String"});
```

这些和消息队列有关的支持类存在于.NET 类库的 System.Messaging 名字空间中。请参

阅类库手册来了解更多有关消息队列的操作。

4. 构建示范数据库

需要使用数据库的有至少两个部门：会计部门和 CRM 部门。因此建立的数据库至少要包含记录客户信息、记录历史订单信息、记录产品信息的表。我们使用 SQL Server 2000 来建立数据库。如果你手头没有 SQL Server 2000，可以使用.NET Framework SDK 中包含的 SQL Server 2000 Desktop Engine 来建立示范用的数据库。为了运行我们将建立的简化的系统模型，需要建立的表的基本结构如下：

表 12-1　客户信息表 Clients

列名	数据类型	长度	允许空
ClientID	nvarchar	50	否
Grade	nvarchar	50	否
Consultation	ntext	16	是

表 12-2　订单记录表 Orders

列名	数据类型	长度	允许空
OrderID	nvarchar	50	否
ProductID	nvarchar	50	否
Quantity	nvarchar	50	否
ClientID	char	10	是
Date	datetime	8	是

表 12-3　产品清单表 Products

列名	数据类型	长度	允许空
ProductID	int	4	否
ProductName	nvarchar	50	否
StandardPrice	money	8	否
Description	nvarchar	50	是

由于我们建立的是一个系统模型，我们将精力放在主要的信息上。实际的信息系统的数据库结构自然不会如此简单。

5. 构建会计子系统 Web Service

根据先前确定的ProductService类接口和数据库结构，我们可以编写出这个Web Service的 C#代码。

程序清单 12-4：ProductService.asmx

```
<%@ WebService Language="C#" Class="ProductService" debug="true"%>

using System;
using System.Data;
using System.Data.SqlClient;
using System.Web.Services;

public class ProductService
```

```
{
    private const string ConnectionString = "data source=(local)\\
     NetSDK; database=Bicycle; uid=sa; pwd=";
    private const string QueryStandardPriceString = "SELECT
     StandardPrice FROM Products WHERE ProductID=@ProductID";
    private const string QueryProductNameString = "SELECT
     ProductName FROM Products WHERE ProductID=@ProductID";
    private const string QueryProductListString = "SELECT
     ProductName, ProductID FROM Products;";
    private const string QueryProductCostString = "SELECT Cost FROM
     Products WHERE ProductID=@ProductID";

    private bool IsEntitledAccessorID(string AccessorID)
    {
        return true;
    }

    [WebMethod]
    public string GetProductName(string ProductID)
    {
        SqlConnection conn = new SqlConnection(ConnectionString);
        SqlCommand cmd = new SqlCommand(QueryProductNameString,
                         conn);
        cmd.Parameters.Add("@ProductID", SqlDbType.NVarChar)
         .Value = ProductID;
        string ProductName;
        try
        {
            conn.Open();
            ProductName = (string) cmd.ExecuteScalar();
        }
        catch (Exception e)
        {
            ProductName = "";
        }
        finally
        {
            if (conn.State == ConnectionState.Open)
                conn.Close();
        }
        return ProductName;
    }

    [WebMethod]
    public Decimal GetProductStandardPrice(string ProductID)
    {
        SqlConnection conn = new SqlConnection(ConnectionString);
        SqlCommand cmd = new SqlCommand(QueryStandardPriceString,
                                        conn);
        cmd.Parameters.Add("@ProductID", SqlDbType.NVarChar)
            .Value = ProductID;
        Decimal Price;
        try
        {
            conn.Open();
            Price = (Decimal) cmd.ExecuteScalar();
        }
        catch (Exception e)
        {
```

```
            Price = -1;
        }
        finally
        {
            if (conn.State == ConnectionState.Open)
                conn.Close();
        }
        return Price;
    }

    [WebMethod]
    public Decimal GetProductCost(string ProductID, string
                                    AccessorID)
    {
        if (IsEntitledAccessorID(AccessorID))
        {
            SqlConnection conn = new SqlConnection
                                 (ConnectionString);
            SqlCommand cmd = new SqlCommand(QueryProductCostString,
                                            conn);
            cmd.Parameters.Add("@ProductID", SqlDbType.NVarChar)
                                .Value = ProductID;
            Decimal Cost;
            try
            {
                conn.Open();
                Cost = (Decimal) cmd.ExecuteScalar();
            }
            catch (Exception e)
            {
                Cost = -1;
            }
            finally
            {
                if (conn.State == ConnectionState.Open)
                    conn.Close();
            }
            return Cost;
        }
        else
            return -1;
    }

    [WebMethod]
    public DataSet GetProductList()
    {
        SqlConnection conn = new SqlConnection(ConnectionString);
        SqlCommand cmd = new SqlCommand(QueryProductListString,
                                        conn);
        SqlDataAdapter ad = new SqlDataAdapter();
        ad.SelectCommand = cmd;
        DataSet ds = new DataSet();
        try
        {
            ad.Fill(ds);
        }
        catch (Exception e)
        {
        }
```

```
            finally
            {
                if (conn.State == ConnectionState.Open)
                    conn.Close();
            }
            return ds;
        }
    }
```

可以看到，这个 Web Service 主要的代码都是和 ADO.NET 相关的，而且都是查询代码。代码的算法都基本一样：建立与数据库的连接，用查询字符串和参数对象来建立 Command 对象，然后执行查询，最后使用异常机制的 finally 来确保连接的关闭。

我们用一个私有函数 IsEntitledAccessorID 来判断查询成本信息的人员是否具有相应的权限。这里有多种方法来实现这样的功能，不过我们关心的是消息是如何在系统间流动的，所以忽略了这个函数的具体实现，仅仅简单地返回 true。

6. 实现客户代表 Web Service

下面我们看看 ClientRepresentative.asmx 文件的完整代码：

程序清单 12-5：ClientRepresentative.asmx

```
<%@ WebService Language="C#" Class="ClientRepresentative"%>
<%@ Assembly Name="System.Messaging" %>


using System;
using System.Data;
using System.Data.SqlClient;
using System.Messaging;
using System.Web.Services;
using System.Xml;

[WebService(Namespace="http://www.toyseller.com")]
public class ClientRepresentative : WebService
{

    private Decimal DetermineProposalPrice(string ProductID, int
     Quantity, DateTime ShipDate, string comment)
    {
        return 1000;
    }

    [WebMethod]
    public Decimal QueryStandardPrice(string ProductID)
    {
        ProductService pd = new ProductService();
        return pd.GetProductStandardPrice(ProductID);
    }

    [WebMethod(EnableSession=true)]
    public void QueryDetailPrice(string ProductID, int Quantity,
     string UserID, DateTime ShipDate)
    {
        ProductService pd = new ProductService();
        string ProductName = pd.GetProductName(ProductID);
        string comment = "";
        string message =
```

```
                "<PriceProposal RepresentativeID='" + Session
                                                .SessionID + "'>";
            message += ("<Product ProductID='" + ProductID + "'>" +
                        ProductName + "</Product>");
            message += ("<Quantity>" + Quantity.ToString() +
                        "</Quantity>");
            message += ("<UserID>" + UserID + "</UserID>");
            message += ("<ShipDate>" + ShipDate.ToString() +
                        "</ShipDate>");
            message +=  "<ProposalPrice currency='Dollar' ";
            message +=      ("price='" + DetermineProposalPrice
            (ProductID, Quantity, ShipDate, comment).ToString() + "'>");
            message +=      ("<comment>" + comment + "</comment>");
            message +=  "</ProposalPrice>";
            message += "</PriceProposal>";

            string mqpath = ".\\Private$\\Proposal";
            MessageQueue queue;
            if (!MessageQueue.Exists(mqpath))
                queue = MessageQueue.Create(mqpath);
            else
                queue = new MessageQueue(mqpath);
            queue.Formatter = new XmlMessageFormatter(new string[]
                              {"System.String"});
            queue.Send(message);
        }
    }
```

QueryStandardPrice 接口方法就是一个后台 ProductService Web Service 的封装。稍微有趣的事情发生在 QueryDetailPrice 接口方法中。它首先调用 ProductService 的 GetProductName 来从产品 ID 获取产品名称，然后使用字符串操作来根据接口方法的输入参数构造一个字符串形式的 XML 文档，最后将它发送到 Proposal 消息队列中。发送消息队列的操作和前面构建的 Common 类的静态函数代码是一样的，也可以将 Common 类编译成 Assembly 后在 Web Service 中使用@Assembly 指令和@Import 指令来引入这个 Assembly 和名字空间，就可以使用前面的代码了。具体的做法是，使用编译器编译 Common.cs 文件：

```
csc /t:library common.cs
```

把编译出来的 common.dll 拷贝到 ClientRepresentative.asmx 所在的虚拟目录的 bin 子目录下，然后在 ClientRepresentative.asmx 文件前部加入

```
<%@ Assembly Name="common" %>
```

就可以使用 common.cs 中的静态函数来发送消息了。

另外，我们使用 DetermineProposalPrice 方法来确定提交的建议报价。我们也没有实现这个函数，只是简单地返回一个常量。因为这可以有多种方法来实现。

接下来，我们来实现系统的中心部位，审核人的控制台。

7. 审核部门 Web 应用程序的实现

按照前面的设计，审核部门人员会在 Web 页上工作。他们会使用页面上的按钮来主动地提取 Proposal 队列中的消息，消息会按一定的格式显示在页面上，然后决定是向客户关系部门请求对客户的评估信息还是通过或拒绝报价建议。

首先我们构造工作台页面的界面元素。

程序清单 12-6：Desktop.aspx

```
<html>
<form runat="server">
<asp:Xml id="Proposal"
    TransformSource="message.xsl"
    runat="server"
/>
<p/>
The standard price of the product is:
<asp:Label id="StandardPrice"
    runat="server"
/>
The cost of the product is:
<asp:Label id="Cost"
    runat="server"
/>
<br>
if to deny it, the price should be:
<asp:RadioButton id="rbHigher" GroupName="rbgPriceSuggestion"
    Text="Higher"
    Checked="true"
    runat="server"
/>
<asp:RadioButton id="rbLower" GroupName="rbgPriceSuggestion"
    Text="Lower"
    runat="server"
/>
<p/>
<asp:Button id="QueryClient"
    Text="I want more information about the client"
    OnClick="QueryClient_Click"
    runat="server"
/>
<asp:Button id="AcceptProposalPrice"
    Text="Accept Proposal Price"
    OnClick="Accept_Click"
    runat="server"
/>
<asp:Button id="DenyProposalPrice"
    Text="Deny Proposal Price"
    OnClick="Deny_Click"
    runat="server"
/>

<p/>
Write down you comment to post to the client representative here:
<p/>
<asp:TextBox id="txtComment"
    runat="server"
/>

<!--
Add validation controls for Number TextControl
-->
</form>
</html>
```

产生的工作台界面大致如图 12.3 所示。

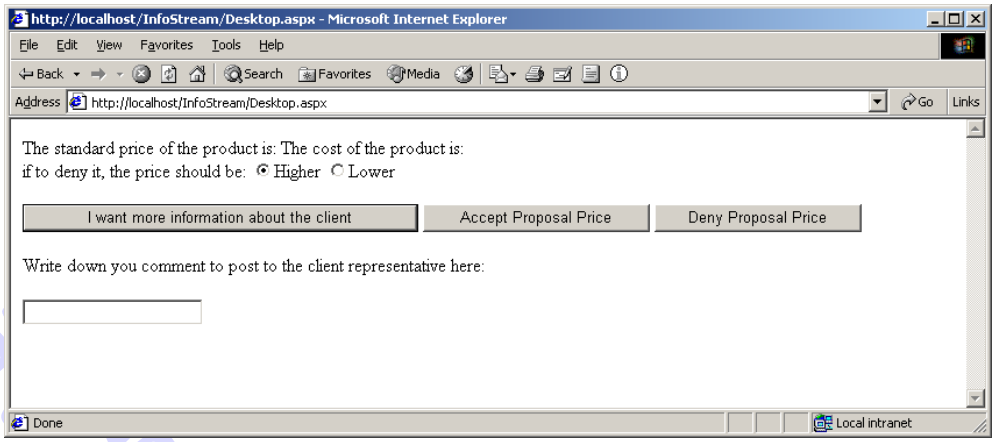


图 12.3

我们主要希望做的是：

1．在页面加载时从消息队列中提取消息，显示到页面中。如果消息包是 ClientInfo 类型，并且 status 属性是 Response 时，表明包中包含了从 CRM 系统得到的客户信息，因此“I want more information about the client”按钮应该无效。否则三个按钮都是有效的。

2．如果审核人单击“I want more information about the client”，将构造 ClinetInfo 包并设置 status=“Request”和一个独一无二的审核人 ID，发送到 ClientInfoRequest 队列中。然后重新加载页面来提取下一条消息。

3．如果单击 Accept 或 Deny 按钮，构造给客户代表的答复包并发送到 ProposalResponse 队列中。同样，重载页面来提取下条信息。构造的答复包中的建议信息和备注内容从页面上的相应 Web Server Control 获取输入。

这个页面中，我们使用了前面构造的 common 类的 Assembly 来简化发送消息。页面的界面逻辑代码如下。

程序清单 12-7：Desktop.aspx

```
<%@ Page Language="C#" Debug="true" %>
<%@ Assembly Name="System.Messaging" %>
<%@ Assembly Name="Common" %>
<%@ Import Namespace="System.Messaging" %>
<%@ Import Namespace="System.Xml" %>
<%@ Import Namespace="System.Data" %>

<script runat="server">

    string GetIndentification()
    {
        // get the id from local machine
        return Session.SessionID;
    }

    void Page_Load(Object Sender, EventArgs e)
    {
        ReceiveNext();
    }
```

```
        //  Get the request
        //  from Client Representative
        //  or CRM Dept.(containing the detail info of client as request)
        void ReceiveNext()
        {
            string path = ".\\Private$\\Proposal";
            string message = Common.PopQueue(path);

            Session["message"] = message;

            Proposal.DocumentContent = message;
            XmlDocument doc = Proposal.Document;

            if (doc.DocumentElement.HasAttribute("AccessorID") &&
                doc.DocumentElement.GetAttribute("AccessorID") ==
                    Session.SessionID)
            // if it's from CRM and is for this Accessor
            {
                if (    doc.DocumentElement.Name == "ClientInfo" &&
                    doc.DocumentElement.GetAttribute("status") ==
                                                        "Response")
                {
                    QueryClient.Enabled = false;
                }
            }
            else
            {
                // push it back for the Accessor who will access it
                Common.PushQueue(path, message);
            }
        }

        void QueryClient_Click(Object Sender, EventArgs e)
        {
            string path = ".\\Private$\\ClientInfoRequest";
            MessageQueue queue = Common.GetMessage(path);

            XmlDocument doc = new XmlDocument();
            XmlElement root = doc.CreateElement("ClientInfo");

            XmlDocument olddoc = new XmlDocument();
            olddoc.LoadXml((string)Session["message"]);

            root.InnerXml = olddoc.DocumentElement.InnerXml;

            root.SetAttribute("RepresentativeID",

    olddoc.DocumentElement.GetAttribute("RepresentativeID"));
            root.SetAttribute("status", "request");
            root.SetAttribute("AccessorID", Session.SessionID);
            string message = doc.InnerXml;

            // send the message to CRM Dept.
            queue.Send(message);

        }

        void Accept_Click(Object Sender, EventArgs e)
```

```
        {
            XmlDocument doc = new XmlDocument();
            XmlElement root = doc.CreateElement("PriceProposal");
            doc.AppendChild(root);

            XmlDocument olddoc = new XmlDocument();
            olddoc.LoadXml((string) Session["message"]);

            root.InnerXml = olddoc.DocumentElement.InnerXml;

            root.SetAttribute("RepresentativeID",

    olddoc.DocumentElement.GetAttribute("RepresentativeID"));

            XmlElement nodeDecision = doc.CreateElement("Decision");
            root.AppendChild(nodeDecision);
            XmlElement nodeResult = doc.CreateElement("Result");
            nodeDecision.AppendChild(nodeResult);
            XmlElement nodeComment = doc.CreateElement("Comment");
            nodeDecision.AppendChild(nodeComment);

            nodeResult.SetAttribute("accept", "true");

            nodeComment.InnerText = txtComment.Text;

            string path = ".\\Private$\\ProposalResponse";
            string message = doc.InnerXml;
            Common.PushQueue(path, message);

        }

        void Deny_Click(Object Sender, EventArgs e)
        {
            XmlDocument doc = new XmlDocument();
            XmlElement root = doc.CreateElement("PriceProposal");
            doc.AppendChild(root);

            XmlDocument olddoc = new XmlDocument();
            olddoc.LoadXml((string) Session["message"]);

            root.InnerXml = olddoc.InnerXml;

            root.SetAttribute("RepresentativeID",

    olddoc.DocumentElement.GetAttribute("RepresentativeID"));

            XmlElement nodeDecision = doc.CreateElement("Decision");
            root.AppendChild(nodeDecision);
            XmlElement nodeResult = doc.CreateElement("Result");
            nodeDecision.AppendChild(nodeResult);
            XmlElement nodeComment = doc.CreateElement("Comment");
            nodeDecision.AppendChild(nodeComment);

            nodeResult.SetAttribute("accept", "false");
            nodeResult.SetAttribute("recommendation", (rbHigher.Checked
                                    ? "higher" : "lower") );

            nodeComment.InnerText = txtComment.Text;
```

```
                string path = ".\\Private$\\ProposalResponse";
                string message = doc.InnerXml;
                Common.PushQueue(path, message);

            }
        </script>
```

在这些代码中，我们重用了从消息队列中提取的消息包。因为消息包的内容是以 string 表示的 XML 文档，并且文档的大小较小，我们使用 XmlDocument 对象来在内存中构造一棵消息对应的 DOM 树（代码中是 olddoc）。然后构造一棵空树（doc），构造根节点，然后用拷贝 InnerXml 属性内容的方法来从源信息的 DOM 树中拷贝其中大部分的子节点。这样我们可以避免在子节点集中迭代遍历的开销。

提示：遍历时的时间复杂度至少为 O（n），而拷贝字符串则是 O（1）!。

接着我们根据需要在新树中重置或添加节点，最后用 InnerXml 属性得到该树的 XML 文档的字符串表示：

```
        string message = doc.InnerXml;
```

至此为止，我们已经可以发送这个 XML 消息了。在页面上显示接收到的 XML 消息时，我们不是从中提取各标记的值后赋给 Web Server Controls，而是使用 Xml Web Server 控件和 XSL 文件来对接收到的 XML 消息进行转换，变成 HTML 代码来显示。如果读者还没有接触过 XSL 语言的编程，现在可以开始感受到这种语言的强大了。XSL 语言还可以用来进行 XML 文档间的转换，即从一种格式的 XML 文档转换成另一种格式的 XML 文档。这在企业间的文档交换中极为有用。

这个页面中使用的 message.xsl 文件如下：

程序清单 12-8：message.xsl

```
        <xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
            <xsl:template match="ClientInfo">
                <html>
                    <body>
                        <table border="3">
                            <tr>
                                <th>Product</th>
                                <th>Quantity</th>
                                <th>ShipDate</th>
                                <th>ProposalPrice</th>
                            </tr>
                            <tr>
                                <td>
                                    <xsl:value-of select="Product"/>
                                </td>
                                <td>
                                    <xsl:value-of select="Quantity"/>
                                </td>
                                <td>
                                    <xsl:value-of select="ShipDate"/>
                                </td>
                                <td>
                                    <xsl:value-of select=
                                                "ProposalPrice"/>
```

```
                        </td>
                    </tr>
                </table>
                <p/>
                <xsl:if test="boolean(//Archives/*)">
                    <xsl:apply-templates select="Archives"/>
                </xsl:if>
            </body>
        </html>
    </xsl:template>
    <xsl:template match="Archives">
        <html>
            <body>
                <hr size="3"/>
                <h4>Archives</h4>
                    <ul>
                        <li>
                            <font  face="arial">Grade:
</font><p/>
                            <xsl:value-of select="Grade"/>
                        </li>
                    </ul>
                    <ul>
                        <li>
                            <font face="arial">Consultation:
                                                </font><p></p>
                            <xsl:value-of select="Consultation"/>
                        </li>
                    </ul>
                    <ul>
                        <li>
                            <font face="arial">Record:</font>
                                                        <p></p>
                            <xsl:apply-templates select="Record"/>
                                        </li>
                    </ul>
                <hr size="3"/>
                </body>
            </html>
        </xsl:template>
    <xsl:template match="Record">
        <xsl:for-each select="Deal">
            <ul>
                <li><xsl:value-of select="@ID"/></li>
            <ul>
            <li>Product:<xsl:value-of select="Product"/></li>
            <li>Quantity:<xsl:value-of select="Quantity"/></li>
            <li>Date:<xsl:value-of select="Date"/></li>
            </ul>
            </ul>
        </xsl:for-each>
    </xsl:template>
</xsl:stylesheet>
```

在装载页面时，我们将得到的 XML 文档传递给 Xml Web Server Control 进行显示，为了方便在重发消息包时重新加载 XML 文档，我们在 Session 对象中保留一个 XML 文档的备份，这样，在重用消息包时，只需使用下面的代码：

```
olddoc.LoadXml((string) Session["message"]);
```

8. 实现自动答复的 CRM 系统引擎

最后一个有趣的东西是我们的 CRM 系统。我们会演示一个控制台程序，这个程序可以自动“监听”消息队列中的消息，如果有消息到达，则会处理消息，否则会等待消息的到来。这是通过消息队列的异步调用来实现的。

使用 Receive/Peek 来从消息队列中提取信息时，这两个方法是同步的，也就是说，如果消息队列是空的，程序流程会阻塞，直至消息队列中出现一条消息。在提取一条信息后，会接着执行下一条语句。

使用消息队列的一个优势是异步，也就是说，提供了和 Receive/Peek 对应的 BeginReceive/BeginPeek 方法，可以使用这两个方法来通知消息队列，当消息到达时消息队列会调用某个函数，从而执行某些特定的操作。执行 BeginReceive/BeginPeek 方法后，程序流程不会阻塞，而是继续执行下一条指令。这有点像回调和 Delegate 机制。在回调函数中，处理完当前消息后，可以再次调用 BeginReceive/BeginPeek 来等待和提取下一条消息。当然，如果消息队列中存在消息，回调函数会立即被调用。

我们实现的 CRM 引擎如下：

程序清单 12-9：CRMEngine.cs

```
using System;
using System.Xml;
using System.Data;
using System.Data.SqlClient;
using System.Messaging;

public class CRMEngine
{
    private const string strQueueRequest = ".\\Private$\\
                                            ClientInfoRequest";
    private const string strQueuePropal = ".\\Private$\\Proposal";
    private const string strConnection = "data source=(local)\\
                            NetSDK; database=Toys; uid=sa; pwd=";
    private const string strGetHistoryOrders = "SELECT * FROM Orders
                                            WHERE ClientID=@ClientID";
    private const string strGetClientInfo = "SELECT * FROM Clients
                                            WHERE ClientID=@ClientID";

    public void Run()
    {
        MessageQueue queueRequest = Common.GetQueue
                                        (strQueueRequest);
        queueRequest.Formatter = new XmlMessageFormatter(new
                                    string[]{"System.String"});
        queueRequest.ReceiveCompleted += new
            ReceiveCompletedEventHandler(OnReceiveCompleted);
        queueRequest.BeginReceive();
        Console.WriteLine("To end monitoring and process request,
                            press ENTER.");
        Console.ReadLine();
    }

    public static void OnReceiveCompleted(
                    Object source,
                    ReceiveCompletedEventArgs asyncResult)
```

```
{
    MessageQueue queueRequest = (MessageQueue)source;
    string msg = (string) queueRequest.EndReceive
                 (asyncResult.AsyncResult).Body;
    XmlDocument doc = new XmlDocument();
    string ClientID;
    string UserGrade;
    string Consultation;
    SqlConnection conn = new SqlConnection(strConnection);;

    try
    {
        Console.WriteLine(msg);
        // analyse the request
        doc.LoadXml(msg);
        if (doc.DocumentElement.GetAttribute("status") !=
                                                "request")
            throw new Exception("XML Parsing Error");
        ClientID = doc.DocumentElement.SelectSingleNode
                   ("//UserID").InnerText;
        Console.WriteLine("ClientID: " + ClientID);
        // process the request
        SqlCommand cmdClientInfo = new SqlCommand
                                   (strGetClientInfo + ";" +
                   strGetHistoryOrders, conn);
        cmdClientInfo.Parameters.Add("@ClientID", SqlDbType.NVarChar)
                                   .Value = ClientID;

        conn.Open();
        SqlDataReader reader = cmdClientInfo.ExecuteReader();

        XmlElement nodeArchives;

        if (reader.Read())
        {
            UserGrade = (string) reader["Grade"];
            Consultation = (string) reader["Consultation"];
            Console.WriteLine("\tGrade" + UserGrade);
        }

        nodeArchives = doc.CreateElement("Archives");
        XmlElement nodeGrade = doc.CreateElement("Grade");
        doc.DocumentElement.AppendChild(nodeArchives);
        nodeArchives.AppendChild(nodeGrade);

        XmlElement nodeRecord = doc.CreateElement("Record");
        ProductService pd = new ProductService();
        if (reader.NextResult())
        {
            while (reader.Read())
            {
                string DealID = (string) reader["OrderID"];
                string ProductID = (string) reader
                                   ["ProductID"];
                string ProductName = pd.GetProductName
                                     (ProductID);
                int Quantity = (int) reader["Quantity"];
                DateTime date = (DateTime) reader["Date"];
                XmlElement nodeDeal = doc.CreateElement
```

```
                                        ("Deal");
                    nodeDeal.SetAttribute("ID", DealID);
                    XmlElement nodeProduct = doc.CreateElement
                                            ("Product");
                    nodeProduct.InnerText = ProductName;
                    XmlElement nodeQuantity = doc.CreateElement
                                            ("Quantity");
                    nodeQuantity.InnerText = Quantity.ToString();
                    XmlElement nodeDate = doc.CreateElement
                                          ("Date");
                    nodeDate.InnerText = date.ToString();
                    nodeDeal.AppendChild(nodeProduct);
                    nodeDeal.AppendChild(nodeQuantity);
                    nodeDeal.AppendChild(nodeDate);
                    nodeRecord.AppendChild(nodeDeal);
                }   // end of while
            }   // end of if
            reader.Close();

            doc.DocumentElement.SetAttribute("status", "response");

            // return the request, put the response into the queue
            msg = doc.InnerXml;
            Common.PushQueue(strQueuePropal, msg);

        }   // end of try
        catch (XmlException e)
        {
            Console.WriteLine(
                "Request cannot be processed due to unexpected XML
                  error.  Ignored.");
            Console.WriteLine("Detail:");
            Console.WriteLine(e.ToString());
        }
        catch (Exception e)
        {
            Console.WriteLine(
                "Request cannot be processed due to unexpected error.
                 Ignored.");
            Console.WriteLine("Detail:");
            Console.WriteLine(e.ToString());
        }
        finally
        {
            if (conn.State == ConnectionState.Open)
                conn.Close();
        }

        Console.WriteLine("Done");
        queueRequest.BeginReceive();        // wait and proccess
                                                next request
    }

    public static void Main()
    {
        CRMEngine crmsys = new CRMEngine();
        crmsys.Run();
    }
}
```

这个引擎是一个控制台程序，可以使用下面的命令行来编译：

```
csc /out:CRMEngine.exe /t:exe CRMEngine.cs common.cs ProductService.cs
```

因为 CRMEngine.cs 中使用到 common 类和 ProductService Web Service，因此必须把 common.cs 和 ProductService 的代理类文件连接起来编译。/out 编译开关用来指定输出的文件名。

编译成功后，可以运行 CRMEnging.exe 就可以开始监听消息队列。

程序开始执行后，使用：

```
queueRequest.ReceiveCompleted += new
            ReceiveCompletedEventHandler(OnReceiveCompleted);
queueRequest.BeginReceive();
```

注册消息到达事件的处理函数 OnReceiveCompleted，然后调用 BeginReceive 方法来监听第一条信息。

在第一条消息到达后，OnReceiveComplete 函数被执行。在这个函数中，首先使用：

```
string msg = (string) queueRequest.EndReceive(asyncResult.AsyncResult).Body;
```

获取消息内容，然后构造一棵 DOM 树来从消息中提取 ClientID 等信息。根据 ClientID，执行一条数据库查询，并用 DataReader 来遍历结果集。为了节省连接开销，我们使用了多表查询，获得两个结果集，然后用 DataReader 的 NextResult 方法来从第一个结果集转到第二个。第一个结果集包含了客户信息表中的该客户基本信息，第二个结果集包含了订单记录表中与该客户的交易情况。我们利用这些信息来构造一棵新的 DOM 树，表示返回的消息包，在构造完后，使用前面的方法得到 string 表示的 XML 文档。然后将消息发送到 Proposal 队列中去。

```
Common.PushQueue(strQueuePropal, msg);
```

为了继续处理下一条消息，在最后我们又一次调用了 BeginReceive 方法。

在 Run 入口中，第一次调用了 BeginReceive 方法后，我们使用 Console 对象的 ReadLine 方法来等待客户端输入回车键。这一行总会执行到，因为 BeginReceive 方法是异步的。如果需要结束，可以敲入回车键。执行完 ReadLine 方法后，程序流就到达末尾了，因此结束整个进程。

CRM 系统除了设计成这种自动回复模式外，也可以设计成类似于审核人的模式，从而便于人员干预和实时地评估。这里设计成自动模式，是假定 CRM 部门会定期或不定期评估客户，并将评估结果放入数据库中，从而减少审核人决策时的等待时间。另一个原因，是我们希望借此机会向读者演示这种异步消息处理技术。当然，在实际系统的实施中，是不能以使用某些技术来作为目标的。尽管技术的进步可能改变企业的业务流程，但一切的一切，都应该从提高企业的核心竞争力出发，业务流程的改变必须符合企业的内部状况、外部环境、战略目标以及企业本身的文化，而不是以技术作为考虑的重点。

9. 实现订单报价系统的 Web 入口

最后，我们来实现一个 Web Forms 构建的 Web 界面，该界面会集成我们提供的报价系统 Web Service，来为客户展示一个可视化的服务。

为了仅向我们认可的客户提供服务，同时也为了确认客户身份以便于后台的 CRM 系统运作，我们为入口页面构造登录页面来进行身份验证。我们采用的是基于 Forms 的验证

机制，并且重用了我们在讲述 User Server Control 技术时创建的 Login 控件。当然，由于当时我们还没有介绍身份验证技术，这个控件是没有使用到 ASP.NET 对身份验证技术的支持的。我们只需修改一处代码，即判断客户 ID 和密码是否匹配的条件语句。修改后的控件代码如下：

程序清单 12-10：example-5-1.ascx

```
<%@Control
    ClassName="usrLoginBox"
    Inherits="CLoginBox"
    src="example-5-1-code.cs"
%>
<%@Import Namespace="System.Web.Security"%>

<table runat="server"
 style="background-color:lightblue;font: 10pt verdana;
        border-width:1;border-style:solid;border-color:black;"
        cellspacing=15>
    <tr>
        <td><b>User: </b></td>
        <td><ASP:TextBox id="User" runat="server"/></td>
    </tr>
    <tr>
        <td><b>Password: </b></td>
        <td><ASP:TextBox id="Pass" TextMode="Password" runat=
         "server"/></td>
    </tr>
    <tr>
        <td></td>
        <td>
            <ASP:Button id="btnSummit"
                Text="Login"
                OnClick="btnSummit_Click"
                runat="server"
            />
        </td>
    </tr>
    <tr>
        <td ColSpan=2><ASP:Label id="lblInfo" runat="server"/>
                                            </td>
    </tr>
</table>
```

程序清单 12-11：example-5-1-code.cs

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

public class UserInfo
{
    public UserInfo(string User, string Pass)
    {
        this.User = User;
        this.Pass = Pass;
    }
    public string User;
```

```
    public string Pass;
}


public delegate void LoginEventHandler(Object Sender, UserInfo e);

public class CLoginBox : UserControl
{
    protected   Button  btnSummit;
    protected   TextBox User;
    protected   TextBox Pass;
    protected   Label   lblInfo;

    public event LoginEventHandler LoginSuccess;
    public event LoginEventHandler LoginFail;

    protected void btnSummit_Click(Object Sender, EventArgs e)
    {
        // initialize the event argument
        UserInfo arg = new UserInfo(User.Text, Pass.Text);

        // verify the user's identification
        if (FormsAuthentication.Authenticate(arg.User, arg.Pass))
        {
            // if the LoginSuccess event is subscribed
            if (null!=LoginSuccess)
                // fire the event
                LoginSuccess(this, arg);
        }
        else
        {
            lblInfo.Text = "Invalid UserID/Password";
            // if the LoginFailed event is subscribed
            if (null!=LoginFail)
                // fire the event
                LoginFail(this, arg);
        }
    }
}
```

为了使身份验证机制工作起来，我们还必须设置相应的 web.config 文件，例如加入两个客户，并使用明文传输密码的设置如下：

```
<authentication mode="Forms">
    <forms loginUrl="/infostream/login.aspx">
        <credentials passwordFormat="Clear">
            <user name="M" password="1"/>
            <user name="Z" password="2"/>
        </credentials>
    </forms>
</authentication>
```

接着我们构建实际的入口的 Web Form。

程序清单 12-12：default.aspx

```
<%@ Page Language="C#" %>
<%@ Import Namespace="System.Data" %>
<%@ Import Namespace="System.Data.SqlClient" %>

<script runat="server">
```

```
        void Page_Load(Object Sender, EventArgs e)
        {
            if (!IsPostBack)
            {
                ProductService pd = new ProductService();
                DataSet ds = pd.GetProductList();
                Products.DataSource = ds;
                Products.DataTextField = "ProductName";
                Products.DataValueField = "ProductID";
                Products.DataBind();
            }
            WelcomeInfo.InnerText = "Welcome! " + Context.User.Identity.
                                    Name + "!";
        }

        void StandardQuery_Click(Object Sender, EventArgs e)
        {
            ClientRepresentative rep = new ClientRepresentative();
            Decimal price = rep.QueryStandardPrice(Products.
                                        SelectedItem.Value);
            ResponseInfo.Text = "The standard price for " +
                Products.SelectedItem.Text +
                " is: " +
                price.ToString();
        }

        void CustomQuery_Click(Object Sender, EventArgs e)
        {
            ClientRepresentative rep = new ClientRepresentative();
            rep.QueryDetailPrice(Products.SelectedItem.Value,
                    Convert.ToInt32(Number.Text),
                    Context.User.Identity.Name,
                    ShipDate.SelectedDate
            );
            ResponseInfo.Text = "Thank you for your interest for our
                                products." +
                    "We'll give you an feasible reply via email very soon";
        }

        void SignOut_Click(Object Sender, EventArgs e)
        {
            FormsAuthentication.SignOut();
            Response.Redirect("login.aspx");
        }

    </script>

    <html>
    <body>
    <form runat="server">
    <span id="WelcomeInfo"
        runat="server"
    />
    <br>
    <asp:Button id="SignOut"
        Text="Sign Out"
        OnClick="SignOut_Click"
        runat="server"
```

```
/>
<br>
<asp:DropDownList id="Products"
    runat="server"
/>
<p/>
<asp:Button id="StandardQuery"
    Text="StandardQuery"
    OnClick="StandardQuery_Click"
    runat="server"
    />

<p/>
<br>
Please tell us the Quantity of the product you may order:
<asp:TextBox id="Number"
    runat="server"
/>
<br>
When would you like the products to be shipped possibly?
<asp:Calendar id="ShipDate"
    runat="server"
/>
<br>
<p/>
<asp:Button id="CustomQuery"
    OnClick="CustomQuery_Click"
    Text="I am interested in the selected product and hope to query
     detail price"
    runat="server"
    />
<p/>

<asp:Label id="ResponseInfo"
    runat="server"
/>

<!--
add validation controls to limit the content of Number TextBox
-->
</form>
</body>
</html>
```

这个页面首次加载时会使用 ProductService Web Service 获取产品清单，然后用数据绑定技术加载到下拉列表控件中。因为有 View State 机制，我们不必在以后页面的加载过程中重新装载数据。SignOut_Click 方法会使用常见的 Forms 验证机制来注销当前用户。StandardQuery_Click 方法调用 ClientRepresentative Web Service 来获取标准报价，然后在页面上显示出来。CustomQuery_Click 方法则调用 ClientRepresentative 来发出查询报价请求，这个方法是异步的，我们的后台系统根据掌握的客户信息，用 E-mail 等多种方式向客户发送最终的报价。页面上还可以加入输入验证控件来确保客户输入的订单中数量范围（例如大于零）和期望中的交货日期范围（例如至少要在当前日期之后）。这里的代码里没有包含这些内容。

如果一切顺利，我们的页面看起来会如图 12.4 所示。

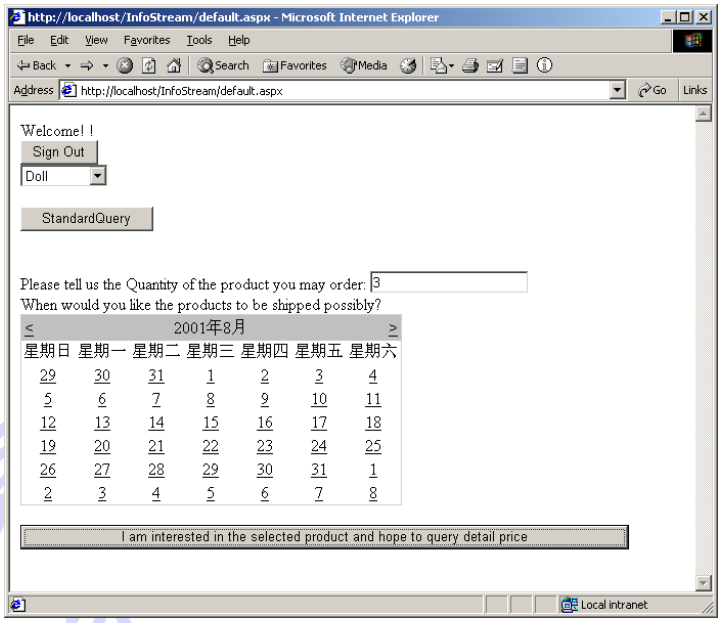


图 12.4

最后，读者会发现，我们这个系统其实并不完善。除了上面提到的几个没有实现的函数外，我们也没有实现客户代表部门提取和处理审核部门回复消息的功能。这些任务将留给读者，用以练习编写健壮的、功能完善的.NET 应用程序。我们相信读者可以将这个系统完善成可以实际运作的信息平台，或者会从这个系统的设计和模型的实现过程中得到某些启示。当然，读者也应该完全有可能、有能力提出自己的设计方案和全新的系统架构。

12.3　本章小结

在上一节中，我们演示了如何用我们前面学过的各种技术来构建一个订单报价系统的模型。我们使用到的技术包括 ASP.NET、ADO.NET、XML、MSMQ，范围覆盖到本书前面介绍的大部分知识点。在这个系统中，我们使用到两种消息通讯机制：同步和异步机制。在.NET Framework 中，对消息机制的支持主要是通过 MSMQ 和 Web Service 来实现的。MSMQ 技术适用于企业 Intranet 中的消息交换，消息的格式、内容相对固定，子系统间、系统模块间的耦合度相对较高。Web Service 技术适合耦合松散的企业间的远程调用和信息交换，并且可以穿透防火墙，企业内部的子系统接口也可以用 Web Service 方式实现，从而便于日后的系统扩展和向外界提供服务。消息队列机制十分适合异步的消息传递，而尽管 Web Service 提供异步的调用方式，它则更擅长于远程方法调用。

在系统内部的消息传递中，XML 的大量采用是必然的趋势。.NET Framework 从框架的底层提供对 XML 的支持，包括 ADO.NET 等技术都可以和 XML 紧密地集成，从而便于系统内部和系统间的不同格式的消息交换和信息系统对消息的自动化处理。.NET 提供了全面的 DOM 对象模型的支持，并提供了类似于 SAX 的 XML 访问机制。

在表现层，.NET 提供了 Web Forms 及相关技术来实现丰富的用户体验和更轻松灵活的开发过程，这些技术包括 Web Server Control、User Server Control、数据绑定技术、和操作系统及 IIS 服务器紧密集成的安全机制、性能卓越的预编译和缓冲机制、调试技术、方便的 Web 应用程序配置和分发等等。

由此可见，构筑下一代基于 Web 的电子商务系统，.NET 无疑是一个极好的平台和开发工具。