PROGRAMMER TO PROGRAMMER™



BEGINNING C#

C#

入门经典

Karli Watson

Marco Bellinaso 等著　康博 译

清华大学出版社

http://www.tup.tsinghua.edu.cn

wrox

PROGRAMMER TO PROGRAMMER™



BEGINNING C#

C# 是 Microsoft 专用在.NET Framework 平台上进行开发的一门新型编程语言。.NET Framework 由托管代码执行的运行时环境，和几乎可以完成所有编程任务的众多类库组成。虽然很多语言都能够编写.NET 代码，但 C# 是惟一针对.NET Framework 而设计的语言，因此在今后几年内，C# 将会成为编写.NET 应用程序的首选。

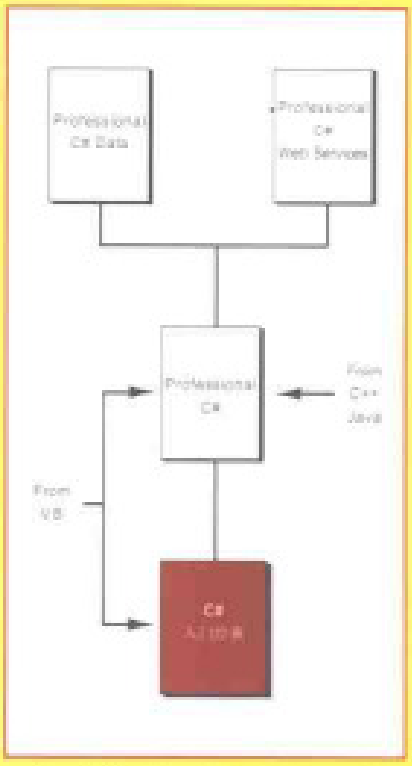
本书是您在学习编写 C# 程序时的必备向导，循序渐进地阐明了 C# 和.NET 中的重要概念和开发技巧。首先讲述 C# 语言的基础知识，然后深入探讨如何使用 C# 进行面向对象的编程，以及如何使用 C# 编写 Windows 应用程序、创建动态 Web 页和 Web 服务。最后的两个案例分析提供了多个完整实用的 C# 应用程序范例，同时展示了利用 C# 编写应用程序的方法。

本书读者对象

本书是一本浅显易懂的 C# 入门手册，适用于初学者、相对缺乏编程经验的程序员，以及那些想从不支持面向对象编程技术的语言转向.NET Framework 平台的程序员。如果您不喜欢阅读那些需要多年 C++ 编程经验的 C# 书籍，那么本书正适合您。

本书内容

- C# 语言基础
- 使用 C# 进行.NET 编程
- 面向对象的编程
- 编写 Windows 应用程序
- 访问数据库
- 用 C# 编写 Web 页和 Web 服务



本书的目标就是让您分享有经验的程序员的知识，提升自己的编程技能。

要获取本书免费源代码和技术支持，请访问：

www.wrox.com

在线讨论：

p2p.wrox.com

Wrox Press Inc., 29 S. LaSalle St. Suite 520 Chicago, Illinois 60603 USA

Wrox Press Ltd., Arden House, 1102 Warwick Road, Acocks Green, Birmingham, B27 6BH, UK.

ISBN 7-302-05333-2

9 787302 053330 >

定价：96.00 元

C#入门经典

Karli Watson
Marco Bellinaso
等著
康　博　译

清华大学出版社

(京) 新登字 158 号
北京市版权局著作权合同登记号：01-2001-4307

内 容 简 介

C#是 Microsoft 专用在.NET Framework 平台上进行开发的一门新型编程语言。.NET Framework 由托管代码执行的运行时环境和几乎可以完成所有编程任务的众多类库组成。虽然很多语言都能够编写.NET 代码，但 C#是惟一针对.NET Framework 而设计的语言，因此在今后几年内，C#将会成为编写.NET 应用程序的首选。

本书是您在学习编写 C#程序时的必备向导，它逐步阐明了 C#和.NET 的关键概念。本书首先介绍 C#语言的基础知识，然后深入探讨如何利用 C#进行面向对象的编程，如何利用 C#编写 Windows 应用程序，以及如何用 C#创建动态 Web 页和 Web 服务。最后的两个案例分析提供了多个完整实用的 C#应用程序范例，同时展示了利用 C#编写应用程序的方法。

本书适用于初学者，以及相对缺乏编程经验、但又想从不支持面向对象编程技术的语言转移到.NET Framework 的程序员。

Karli Watson　Marco Bellinaso et al：Beginning C#
EISBN：1-861004-98-2
Copyright© 2001 by Wrox Press Ltd.
Authorized translation from the English language edition published by Wrox Press Ltd.
All rights reserved. For sale in the People's Republic of China only.
Chinese simplified language edition published by Tsinghua University Press.

本书中文简体字版由清华大学出版社和英国乐思出版公司合作出版。未经出版者书面许可，不得以任何方式复制或抄袭本书内容。

版权所有，翻印必究。
本书封面贴有清华大学出版社激光防伪标签，无标签者不得销售。

书　　名：C＃入门经典
作　　者：Karli Watson　Marco Bellinaso 等著　康博　译
出 版 者：清华大学出版社(北京清华大学学研大厦，邮编 100084)
http://www.tup.tsinghua.edu.cn
责任编辑：李阳
印 刷 者：清华大学印刷厂
发 行 者：新华书店总店北京发行所
开　　本：787×1092 1/16　**印张**：55.75　**字数**：1426 千字
版　　次：2002 年 4 月第 1 版　2002 年 4 月第 1 次印刷
书　　号：ISBN 7-302-05333-2/TP・3132
印　　数：0001～5000
定　　价：96.00 元

出版者的话

随着国际互联网的快速崛起和迅猛发展，计算机之间的互联需求越来越迫切，而目前计算机硬件设备的不兼容性严重束缚了互联网的发展，引发了新一轮的跨平台软件的开发浪潮。软件商纷纷推出新的战略规划和解决方案，Microsoft 提出的.NET 战略就是其中的经典之作。

在经历这场浪潮的洗涤和考验过程中，全球的软件开发人员都迫切需要了解新的软件技术和开发思路。为了满足国内 IT 从业人员的需要，清华大学出版社从 Wrox 出版公司引进了若干套编程系列丛书，“入门经典”系列是其中不可或缺的入门之作。作为世界著名的编程技术图书的出版公司，Wrox 推出的这套“入门经典”系列丛书主要面向编程的初学者、需要了解.NET 策略的程序员，以及需要迅速掌握多门编程工具的程序员。该从书依旧秉承了 Wrox 公司“由程序员为程序员而著（Programmer to Programmer）”的创作理念，每本书均由世界顶级的编程高手执笔。他们站在资深程序员的高度，循序渐进地为初学者讲述了.NET 的理念和构架、编程基本思想、编程语言基础、程序的控制和调试、Windows 应用程序的开发、对象编程技术、数据库访问技术、Web 程序开发和.NET 构架应用等最新的软件开发知识，同时辅以大量操作性强的程序为示例，为读者提供了清晰的编程思路和宝贵的编程经验。

为了保证该系列丛书的质量，清华大学出版社迅速组织了一批位于 IT 开发领域前沿的专家学者进行翻译，经过编辑人员的进一步加工整理后，现陆续奉献给广大读者。

读者可以从 www.wrox.com 网站下载所需的源代码和获取相关的技术支持。同时，也欢迎广大读者参与 p2p.wrox.com 网站上的在线讨论，与世界各地的编程人员交流读书感受和编程体验。

MJ557/02

目　　录

第 1 章　C#简介 …… 1
1.1　什么是.NET Framework …… 1
1.1.1　.NET Framework 的内容 …… 1
1.1.2　如何用.NET Framework 编写应用程序 …… 2
1.2　什么是 C# …… 4
1.2.1　用 C#能编写什么样的应用程序 …… 5
1.2.2　本书中的 C# …… 5
1.3　Visual Studio.NET …… 6
1.4　小结 …… 7

第 2 章　编写 C#程序 …… 8
2.1　Visual Studio.NET 开发环境 …… 8
2.2　控制台应用程序 …… 9
2.3　Windows Forms 应用程序 …… 14
2.4　小结 …… 17

第 3 章　变量和表达式 …… 18
3.1　C#的基本语法 …… 18
3.2　变量 …… 21
3.2.1　简单类型 …… 22
3.2.2　变量的命名 …… 25
3.2.3　字面值 …… 27
3.2.4　变量声明和赋值 …… 28
3.3　表达式 …… 29
3.3.1　数学运算符 …… 29
3.3.2　赋值运算符 …… 33
3.3.3　运算符的优先级 …… 34
3.3.4　名称空间 …… 35
3.4　小结 …… 38
3.5　练习 …… 38

第 4 章　流程控制 …… 40
4.1　布尔逻辑 …… 40
4.1.1　位运算符 …… 42

4.1.2 布尔赋值运算符 …… 45
4.1.3 运算符的优先级更新 …… 47
4.1.4 goto 语句 …… 47
4.2 分支 …… 49
4.2.1 三元运算符 …… 49
4.2.2 if 语句 …… 49
4.2.3 switch 语句 …… 53
4.3 循环 …… 56
4.3.1 do 循环 …… 57
4.3.2 while 循环 …… 60
4.3.3 for 循环 …… 62
4.3.4 循环的中断 …… 66
4.3.5 无限循环 …… 67
4.4 小结 …… 68
4.5 练习 …… 68

第 5 章 变量的更多内容 …… 69
5.1 类型转换 …… 69
5.1.1 隐式转换 …… 69
5.1.2 显式转换 …… 71
5.1.3 使用 Convert 命令进行显式转换 …… 73
5.2 复杂的变量类型 …… 76
5.2.1 枚举 …… 76
5.2.2 结构 …… 81
5.2.3 数组 …… 84
5.3 字符串的处理 …… 90
5.4 小结 …… 95
5.5 练习 …… 95

第 6 章 函数 …… 97
6.1 定义和使用函数 …… 98
6.1.1 返回值 …… 99
6.1.2 参数 …… 101
6.2 变量的作用域 …… 108
6.2.1 其他结构中的变量作用域 …… 111
6.2.2 参数和返回值与全局数据 …… 112
6.3 Main()函数 …… 114
6.4 结构函数 …… 116
6.5 函数的重载 …… 117

6.6 委托 119
6.7 小结 122
6.8 练习 122

第 7 章 调试和错误处理 124
7.1 Visual Studio 中的调试 124
7.1.1 非中断(正常)模式下的调试 125
7.1.2 中断模式下的调试 131
7.2 错误处理 139
7.3 小结 146
7.4 练习 147

第 8 章 面向对象编程简介 148
8.1 什么是面向对象编程 148
8.1.1 什么是对象 149
8.1.2 所有的东西都是对象 151
8.1.3 对象的生命周期 151
8.2 静态和实例类成员 152
8.3 OOP 技术 153
8.3.1 接口 153
8.3.2 继承 154
8.3.3 多态性 156
8.3.4 对象之间的关系 158
8.3.5 运算符重载 159
8.3.6 事件 160
8.3.7 引用类型和值类型 160
8.4 Windows 应用程序中的 OOP 161
8.5 小结 163
8.6 练习 164

第 9 章 定义类 165
9.1 C#中的类定义 165
9.2 System.Object 171
9.3 构造函数和析构函数 172
9.4 Visual Studio.NET 中的 OOP 工具 176
9.4.1 Class View 窗口 176
9.4.2 对象浏览器 178
9.4.3 添加类 179
9.5 类库工程 180

9.6 接口和抽象类 …… 183
9.7 结构类型 …… 185
9.8 阴影和深度复制 …… 187
9.9 小结 …… 187
9.10 练习 …… 188

第 10 章 定义类成员 …… 189
10.1 成员定义 …… 189
10.1.1 定义域 …… 189
10.1.2 定义方法 …… 190
10.1.3 定义属性 …… 191
10.2 VS 成员向导 …… 196
10.2.1 添加方法的向导 …… 197
10.2.2 添加属性的向导 …… 197
10.2.3 添加域的向导 …… 198
10.3 成员属性 …… 199
10.4 类成员的其他议题 …… 199
10.4.1 隐藏基类方法 …… 199
10.4.2 调用重写或隐藏的基类方法 …… 201
10.4.3 嵌套的类型定义 …… 202
10.5 接口的执行 …… 203
10.6 范例应用程序 …… 206
10.6.1 规划应用程序 …… 206
10.6.2 编写类库 …… 207
10.6.3 类库的客户应用程序 …… 211
10.7 小结 …… 213
10.8 练习 …… 213

第 11 章 类的更多内容 …… 214
11.1 集合 …… 214
11.1.1 使用集合 …… 215
11.1.2 定义集合 …… 221
11.2 运算符重载 …… 231
11.2.1 转换运算符 …… 236
11.2.2 升级 CardLib(第二部分) …… 237
11.3 高级转换 …… 242
11.3.1 封箱和拆箱 …… 242
11.3.2 is 运算符 …… 244
11.3.3 as 运算符 …… 247

11.4 深度复制 …… 248
11.5 定制异常 …… 252
11.6 小结 …… 254
11.7 练习 …… 255

第 12 章 事件 …… 256
12.1 什么是事件 …… 256
12.2 使用事件 …… 257
12.2.1 处理事件 …… 257
12.2.2 定义事件 …… 260
12.3 扩展和使用 CardLib …… 268
12.4 小结 …… 276
12.5 练习 …… 277

第 13 章 使用 Windows Form 控件 …… 278
13.1 Windows 窗体设计器 …… 278
13.2 工具箱 …… 279
13.3 控件 …… 280
13.3.1 属性 …… 280
13.3.2 Anchor 和 Dock 属性 …… 281
13.3.3 事件 …… 282
13.4 Button 控件 …… 283
13.4.1 Button 控件的属性 …… 283
13.4.2 Button 控件的事件 …… 284
13.5 Label 和 LinkLabel 控件 …… 286
13.6 TextBox 控件 …… 287
13.6.1 TextBox 控件的属性 …… 287
13.6.2 TextBox 控件的事件 …… 288
13.7 RadioButton 和 CheckBox 控件 …… 297
13.7.1 RadioButton 控件的属性 …… 297
13.7.2 RadioButton 控件的事件 …… 297
13.7.3 CheckBox 控件的属性 …… 298
13.7.4 CheckBox 控件的事件 …… 298
13.7.5 GroupBox 控件 …… 299
13.8 RichTextBox 控件 …… 302
13.8.1 RichTextBox 控件的属性 …… 303
13.8.2 RichTextBox 控件的事件 …… 304
13.9 ListBox 和 CheckedListBox 控件 …… 310
13.9.1 ListBox 控件的属性 …… 310

13.9.2 ListBox 控件的方法 ····· 311
13.9.3 ListBox 控件的事件 ····· 312
13.10 ComboBox 控件 ····· 314
13.10.1 ComboBox 控件的属性 ····· 314
13.10.2 ComboBox 控件的事件 ····· 315
13.11 ListView 控件 ····· 320
13.11.1 ListView 控件的属性 ····· 321
13.11.2 ListView 控件的方法 ····· 323
13.11.3 ListView 控件的事件 ····· 323
13.11.4 ListViewItem ····· 324
13.11.5 ColumnHeader ····· 324
13.11.6 ImageList 控件 ····· 324
13.12 StatusBar 控件 ····· 331
13.12.1 StatusBar 控件的属性 ····· 331
13.12.2 StatusBar 控件的事件 ····· 332
13.12.3 StatusBarPanel 类 ····· 332
13.12.4 StatusBar 范例 ····· 332
13.13 TabControl 控件 ····· 334
13.13.1 TabControl 控件的属性 ····· 335
13.13.2 使用 TabControl 控件 ····· 335
13.14 小结 ····· 337

第 14 章 Windows Forms 的高级功能 ····· 338
14.1 菜单 ····· 338
14.1.1 使用菜单控件 ····· 338
14.1.2 MenuItem 控件 ····· 339
14.1.3 MenuItem 控件的事件 ····· 340
14.1.4 ContextMenu 控件 ····· 341
14.2 工具栏 ····· 345
14.2.1 Toolbar 控件的属性 ····· 346
14.2.2 Toolbar 控件的事件 ····· 346
14.2.3 ToolbarButton 控件的属性 ····· 347
14.3 SDI 和 MDI 应用程序 ····· 352
14.4 创建控件 ····· 364
14.4.1 调试用户控件 ····· 370
14.4.2 扩展 LabelTextbox 控件 ····· 371
14.5 小结 ····· 375
14.6 练习 ····· 375

第 15 章 使用对话框 376
15.1 对话框 376
15.2 如何使用对话框 378
15.3 文件对话框 378
15.3.1 OpenFileDialog 379
15.3.2 SaveFileDialog 391
15.4 打印 396
15.4.1 打印结构 396
15.4.2 打印多个页面 401
15.4.3 页面设置 403
15.4.4 打印对话框 406
15.5 打印预览 410
15.5.1 PrintPreviewDialog 410
15.5.2 PrintPreviewControl 411
15.6 FontDialog 和 ColorDialog 412
15.6.1 FontDialog 412
15.6.2 ColorDialog 414
15.7 小结 415
15.8 练习 415

第 16 章 GDI+简介 417
16.1 图形绘制概述 417
16.1.1 Graphics 类 417
16.1.2 对象的删除 418
16.1.3 坐标系统 419
16.1.4 颜色 424
16.2 使用 Pen 类绘制线条 424
16.3 使用 Brush 类绘制图形 426
16.4 使用 Font 类绘制文本 429
16.5 使用图像进行绘制 433
16.5.1 使用纹理画笔进行绘图 435
16.5.2 使用钢笔绘制图像 437
16.5.3 使用图像绘制文本 438
16.5.4 双倍缓冲 439
16.6 GDI+的高级功能 441
16.6.1 剪切 441
16.6.2 打印 442
16.6.3 打印预览 442

16.6.4 System.Drawing.Drawing2D ····· 443
16.6.5 System.Drawing.Imaging ····· 443
16.7 小结 ····· 443

第 17 章 部署 Windows 应用程序 ····· 445
17.1 部署的含义 ····· 445
17.2 部署工程类型 ····· 446
17.3 Microsoft Windows 安装程序结构 ····· 447
17.3.1 Windows Installer 术语 ····· 447
17.3.2 Windows Installer 的优点 ····· 449
17.4 为简单的编辑器创建一个安装程序软件包 ····· 449
17.4.1 规划安装内容 ····· 450
17.4.2 创建工程 ····· 450
17.4.3 工程属性 ····· 451
17.4.4 安装编辑器 ····· 453
17.4.5 File System 编辑器 ····· 454
17.4.6 File Types 编辑器 ····· 458
17.4.7 Launch Condition 编辑器 ····· 459
17.4.8 User Interface 编辑器 ····· 460
17.5 构建工程 ····· 463
17.6 安装 ····· 464
17.6.1 Welcome ····· 465
17.6.2 Read Me ····· 465
17.6.3 License Agreement ····· 465
17.6.4 Optional Files ····· 466
17.6.5 选择安装文件夹 ····· 467
17.6.6 确认安装 ····· 467
17.6.7 进度 ····· 468
17.6.8 结束安装 ····· 468
17.6.9 运行应用程序 ····· 469
17.6.10 自我修复 ····· 469
17.6.11 卸载 ····· 470
17.7 小结 ····· 471

第 18 章 获取数据 ····· 472
18.1 在 VS 中访问数据 ····· 472
18.1.1 数据访问简史 ····· 472
18.1.2 ADO.NET 引言 ····· 473
18.2 在 VS 中查看数据 ····· 474

18.2.1 连接到数据库 475
18.2.2 数据库表和关系 477
18.3 在应用程序中访问数据库 479
18.4 查看完成的结果 485
18.5 在 DataSet 中导航 487
18.6 添加列表 488
18.7 添加 Datagrid 490
18.7.1 格式化 Datagrid 490
18.7.2 更新数据库 491
18.8 XML 492
18.8.1 XML 文档 492
18.8.2 XML 名称空间 495
18.8.3 格式良好并有效的 XML 496
18.8.4 验证 XML 文档 497
18.8.5 C#中的 XML 自动文档 502
18.9 小结 505
18.10 练习 506

第 19 章 使用 ADO.NET 访问数据 507
19.1 ADO.NET 概述 507
19.1.1 ADO.NET 名称的来源 507
19.1.2 ADO.NET 的设计目标 507
19.2 ADO.NET 类和对象概述 508
19.2.1 提供者对象 509
19.2.2 用户对象 510
19.2.3 使用 System.Data 名称空间 510
19.2.4 在 DataSet 中访问多个表 530
19.3 XML 和 ADO.NET 537
19.4 在 ADO.NET DataSets 中的 XML 支持 537
19.5 ADO.NET 中的 SQL 支持 539
19.5.1 在 DataAdapter 中的 SQL 命令 539
19.5.2 直接执行 SQL 命令 542
19.6 关于 ADO.NET 的进一步说明 544
19.7 小结 544
19.8 练习 545

第 20 章 使用文件 546
20.1 流 546
20.2 用于输入和输出的类 547

20.2.1 File 和 Directory 类 ····· 548
20.2.2 FileInfo 类 ····· 548
20.2.3 DirectoryInfo 类 ····· 549
20.2.4 FileStream 对象 ····· 550
20.2.5 StreamWriter 对象 ····· 556
20.2.6 StreamReader 对象 ····· 558
20.3 高级主题 ····· 563
20.3.1 监控文件结构 ····· 563
20.3.2 异步访问文件 ····· 570
20.4 小结 ····· 575
20.5 练习 ····· 576

第 21 章 .NET 装配件 ····· 577
21.1 装配件简介 ····· 577
21.1.1 组件 ····· 577
21.1.2 组件的优点 ····· 577
21.1.3 组件的简短历史 ····· 578
21.2 自我说明的.NET 装配件 ····· 579
21.3 装配件的结构 ····· 580
21.4 创建装配件 ····· 582
21.4.1 创建类库 ····· 582
21.4.2 Shapes 源代码 ····· 582
21.4.3 编译类库 ····· 583
21.5 查看装配件的内容 ····· 584
21.5.1 将 ILDASM 作为外部工具添加到 VS 中 ····· 584
21.5.2 使用 ILDASM ····· 585
21.5.3 清单 ····· 587
21.6 装配件属性 ····· 589
21.6.1 AssemblyInfo.cs ····· 589
21.6.2 版本号 ····· 591
21.6.3 版本属性 ····· 591
21.6.4 AssemblyVersion 属性 ····· 592
21.6.5 版本兼容性 ····· 592
21.7 调用装配件 ····· 593
21.8 私有和共享装配件 ····· 596
21.8.1 私有装配件 ····· 596
21.8.2 共享装配件 ····· 596
21.8.3 搜索装配件 ····· 599

21.9 小结 600
21.10 练习 600

第 22 章 属性 602
22.1 什么是属性 602
22.2 反射 605
22.3 内置属性 609
22.3.1 System.Diagnostics.ConditionalAttribute 610
22.3.2 System.ObsoleteAttribute 612
22.3.3 System.SerializableAttribute 614
22.3.4 System.Reflection.AssemblyDelaySignAttribute 616
22.4 定制属性 620
22.4.1 TestCaseAttribute 621
22.4.2 System.AttributeUsageAttribute 625
22.5 小结 647

第 23 章 Web 编程基础 648
23.1 引言 648
23.2 Internet 简史 648
23.2.1 World Wide Web 649
23.2.2 Web 的工作原理 649
23.2.3 HTTP – 超文本传输协议 650
23.3 Web 编程 101 652
23.3.1 HTML 652
23.3.2 HTML 窗体 655
23.3.3 静态和活动的 Web 页 659
23.4 Web 的演化过程 665
23.5 .NET Web 编程 665
23.5.1 ASP.NET 666
23.5.2 WebForms 666
23.6 小结 667

第 24 章 ASP.NET 应用程序 668
24.1 建立 Weblog 669
24.1.1 应用程序的基础 669
24.1.2 添加活动元素 672
24.2 Weblog 项目 676
24.2.1 存储 Weblog 项目 676
24.2.2 显示 Weblog 项目 679

24.2.3 显示时间 ····· 685
24.2.4 改善页面的外观 ····· 687
24.2.5 显示项目列表 ····· 692
24.3 创建新 Weblog 项目 ····· 699
24.3.1 编辑 Weblog 项目 ····· 706
24.3.2 给项目排序 ····· 710
24.4 用户会话和 Cookies ····· 713
24.5 小结 ····· 717
24.6 练习 ····· 717

第 25 章 Web 服务 ····· 718
25.1 Web 服务推出之前 ····· 718
25.1.1 远程过程调用 (RPC) ····· 719
25.1.2 SOAP ····· 719
25.2 使用 Web 服务的场合 ····· 720
25.2.1 宾馆旅行社代理应用程序 ····· 721
25.2.2 书籍发布应用程序 ····· 721
25.2.3 客户应用程序的类型 ····· 721
25.2.4 应用程序的体系结构 ····· 721
25.3 Web 服务的体系结构 ····· 722
25.3.1 Web 服务的搜索引擎 ····· 723
25.3.2 Web 服务的发现 ····· 724
25.3.3 可以调用的方法 ····· 725
25.3.4 调用方法 ····· 726
25.4 Web 服务和.NET Framework ····· 728
25.4.1 创建 Web 服务 ····· 728
25.4.2 客户程序 ····· 730
25.5 创建一个简单的 ASP.NET Web 服务 ····· 730
25.5.1 生成的文件 ····· 731
25.5.2 添加 Web 方法 ····· 732
25.6 测试 Web 服务 ····· 733
25.7 执行 Windows 客户程序 ····· 734
25.8 执行 ASP.NET 客户程序 ····· 738
25.9 小结 ····· 740

第 26 章 案例分析 1——调查问卷 Web 站点 ····· 741
26.1 调查问卷 Web 站点 ····· 741
26.2 版本 1 ····· 742
26.2.1 要求 ····· 742

26.2.2 对象模型的设计(事务逻辑层) ····· 742
26.2.3 测试 ····· 747
26.2.4 数据结构(数据层) ····· 749
26.2.5 对象的执行 ····· 750
26.2.6 用户界面(表示层) ····· 758
26.2.7 版本 1 的价值 ····· 766
26.3 版本 2 ····· 768
26.3.1 其他要求 ····· 768
26.3.2 调查结果工具 ····· 768
26.3.3 一人一张选票 ····· 780
26.3.4 调查问卷控件 ····· 783
26.4 三层和调查问卷应用程序 ····· 786
26.5 小结 ····· 786

第 27 章 案例分析 2——在线时事通讯管理器 ····· 788
27.1 应用程序概览 ····· 788
27.2 应用程序的设计 ····· 789
27.3 数据和事务对象 ····· 790
27.3.1 数据库设计 ····· 790
27.3.2 访问数据库的事务类 ····· 793
27.3.3 配置数据管理 ····· 796
27.4 管理用户界面 ····· 801
27.4.1 创建用户控件 ····· 802
27.4.2 Lists 管理器页面 ····· 803
27.4.3 管理订阅者 ····· 814
27.4.4 给订阅者发送电子邮件 ····· 817
27.4.5 Utilities 类 ····· 822
27.4.6 管理新闻归档文件 ····· 822
27.4.7 在线修改设置 ····· 825
27.4.8 在线修改布局 ····· 828
27.4.9 为用户创建订阅窗体 ····· 829
27.4.10 订阅邮件列表 ····· 831
27.4.11 保证应用程序的安全性 ····· 835
27.4.12 编译装配件 ····· 838
27.4.13 运行应用程序 ····· 839
27.4.14 浏览器的兼容问题 ····· 839
27.5 小结 ····· 839

第 28 章　范例——用 C#建立一个在线购物车 ····· 841
28.1　引言 ····· 841
28.2　建立基础 ····· 841
28.2.1　深入理解在线购物车模型 ····· 841
28.2.2　定义需求 ····· 842
28.2.3　生成数据库模型 ····· 842
28.3　开始编码 ····· 844
28.3.1　应用程序的架构 ····· 844
28.3.2　类别列表块 ····· 844
28.3.3　商品的细目 ····· 849
28.3.4　购物车筐 ····· 852
28.3.5　检验系统 ····· 855
28.3.6　跟踪订单 ····· 856
28.4　站点的部署 ····· 857
28.5　改进在线购物车程序 ····· 857
28.5.1　功能的改进 ····· 858
28.5.2　技术方面的改进 ····· 858
28.6　结论 ····· 858

附录 A　设置 PATH 环境变量 ····· 859
A.1　Windows 2000/XP ····· 859
A.2　Windows NT ····· 860
A.3　Windows 95/98/ME ····· 861

附录 B　安装 MSDE ····· 862
B.1　安装 Microsoft Database Engine (MSDE) ····· 862
B.2　把 Northwind 导入 MSDE ····· 866

附录 C　C#编译选项 ····· 870

第1章 C# 简 介

本章将概述 C#和.NET Framework，我们对这些技术的理解，使用它们的原因，以及它们之间的相互关系。

首先讨论一下.NET Framework。这是一种新技术，它包含的许多概念初看起来都不是很容易掌握的(主要因为该框架在应用程序开发环境中引入了执行操作的一种新方式)。也就是说，我们必须在很短的时间里介绍许多新概念，但是，快速浏览这些基础知识对于理解如何在 C#中编程是非常重要的，所以这是不可避免的。本书的后面将详细论述这里提到的许多论题。

之后，本章将讨论 C#本身，包括它的起源和与 C++的类似之处。

最后，介绍本书使用的主要工具：Visual Studio.NET (VS)。

1.1 什么是.NET Framework

.NET Framework 是 Microsoft 为开发应用程序创建的一个富有革命性的新平台。

这句话最有趣的地方是它的含糊不清，但这是有原因的。首先，注意这句话没有说“在 Windows 操作系统上开发应用程序”。尽管.NET Framework 发布的第一个版本运行在 Windows 操作系统上，但以后将推出运行在其他操作系统上的版本，这些操作系统包括 FreeBSD、Linux、Macintosh，甚至个人数字助手(PDA)类设备。使用这项技术的一个主要原因是它可以作为集成各种操作系统的方式。

另外，上面给出的.NET Framework 定义并没有限制应用程序的类型。这是因为本来就没有限制。.NET Framework 可以创建 Windows 应用程序、Web 应用程序、Web 服务和其他各种类型的应用程序。

这样设计.NET Framework，是为了让它可以用于各种语言，包括本书要介绍的 C#语言，以及 C++、Visual Basic、JScript，甚至一些旧的语言，如 COBOL。为此，还出现了这些语言的.NET 版本：Managed C++、Visual Basic.NET 和 JScript.NET 等，目前还在不断推出更多的.NET 版本的语言。所有这些语言都可以访问.NET Framework，它们还可以彼此交互。C#开发人员可以使用 Visual Basic.NET 程序员编写的代码，反之亦然。

所有这些提供了意想不到的多样性，这也是.NET Framework 具有诱人前景的部分原因。

1.1.1 .NET Framework 的内容

.NET Framework 主要包含一个非常大的代码库，这些代码可以在客户语言(如 C#)中通过面向对象编程技术(OOP)来使用。这个库分为不同的模块，这样就可以根据希望得到的结果来选择使用其中的各个部分。例如，一个模块包含 Windows 应用程序的建立块，另一个模块包含连网的代码块，还有一个模块包含 Web 开发的代码块。一些模块还分为更特殊的子模块，例如在 Web 开发模块中，有用于建立 Web 服务的子模块。

其目的是，不同的操作系统可以根据自己的特性，支持其中的部分或全部模块。例如，PDA支持所有的核心.NET 功能，但不需要某些更深奥的模块。

部分.NET Framework 库定义了一些基本类型。类型是数据的一种表达方式，指定其中最基础的部分(例如 32 位带符号的整数)，以便于使用.NET Framework 在各种语言之间交互操作。这称为通用类型系统(Common Type System，CTS)。

除了支持这个库以外，.NET Framework 还包含.NET 公共语言运行时(Common Language Runtime，CLR)，它负责管理用.NET 库开发的所有应用程序的执行。

1.1.2 如何用.NET Framework 编写应用程序

使用.NET Framework 编写应用程序，就是使用.NET 代码库编写代码(使用支持.NET Framework 的任何一种语言)。本书中所有的范例都使用 VS 进行开发，VS 是一种强大的集成开发环境，支持 C#(以及托管和非托管 C++、Visual Basic.NET 和其他一些语言)。这个环境的优点是便于把.NET 功能集成到代码中。我们创建的代码完全是 C#代码，但使用.NET Framework，并在需要时利用 VS 中的其他工具。

为了执行 C#代码，必须把它们转换为目标操作系统能够理解的语言，即本机代码，这种转换称为编译代码，由编译器进行。但在.NET Framework 下，这个过程分为两个部分。

1. MSIL 和 JIT

在编译使用.NET Framework 库的代码时，不是立即创建操作系统特定的本机代码，而是把代码编译为 Microsoft 中间语言(Microsoft Intermediate Language，MSIL)代码，这些代码不专用于任何一种操作系统，也不专用于 C#。其他.NET 语言，如 Visual Basic.NET 也可以在第一阶段编译为这种语言，当使用 VS 开发 C#应用程序时，编译过程就由 VS 完成。

显然，要执行应用程序，必须完成更多的工作，这是 Just-In-Time (JIT)编译器的任务，它把 MSIL 编译为专用于 OS 和目标机器结构的本机代码。这样 OS 才能执行应用程序。这里编译器的名称 Just-In-Time 反映了 MSIL 仅在需要时才编译的事实。

过去，常常需要把代码编译为几个应用程序，每个应用程序用于特定的操作系统和 CPU 结构。这通常是一种优化形式(例如，为了让代码在 AMD 芯片上运行得更快)，但有时是非常重要的(例如对于工作在 Win9x 和 WinNT/2000 环境下的应用程序)。现在就不必要了，因为顾名思义，JIT 编译器使用 MSIL 代码，而 MSIL 代码是独立于机器、操作系统和 CPU 的。目前有几种 JIT 编译器，每种编译器都用于不同的结构，我们总能找到一个合适的编译器创建所需的本机代码。

这样，用户需要做的工作就比较少了。实际上，可以不考虑与系统相关的细节，把注意力放在代码的功能上就够了。

2. 装配件

在编译应用程序时，创建的 MSIL 代码存储在一个装配件中，装配件包括可执行的应用程序文件(这些文件可以直接在 Windows 上运行，不需要其他程序，其扩展名是.exe)和其他应用程序使用的库(其扩展名是.dll)。

除了包含 MSIL 外，装配件还包含元信息(即装配件中包含的数据的信息，也称为元数据)

和可选的资源(MSIL 使用的其他数据，例如声音文件和图片)。元信息允许装配件是完全自我描述的。不需要其他信息就可以使用装配件，也就是说，我们不会遇到下述情形：不能把需要的数据添加到系统注册表中，而这种情形在使用其他平台进行开发时常常出现。

因此，部署应用程序就非常简单了，只需把文件复制到远程计算机上的目录下即可。因为不需要目标系统上的其他信息，所以只需从该目录中运行可执行文件即可(假定安装了.NET CLR)。

当然，不必把运行应用程序所需要的所有信息都安装到一个地方。可以编写一些代码，执行多个应用程序所要求的任务。此时，通常把这些可重用的代码放在所有应用程序都可以访问的地方。在.NET Framework 中，这个地方是全局装配件高速缓冲存储器(Global Assembly Cache ,GAC)，把代码放在这个高速缓冲存储器中是很简单的，只需把包含代码的装配件放在包含该高速缓冲存储器的目录下即可。

3. 托管代码

在把代码编译为 MSIL，再用 JIT 编译器把它编译为本机代码后，CLR 的任务还没有全部完成。用.NET Framework 编写的代码在执行(这个阶段通常称为运行时(runtime))时是托管的。即 CLR 管理着应用程序，其方式是管理内存、处理安全性，以及允许进行跨语言调试等。相反，不在 CLR 控制之下运行的应用程序是非托管的，某些语言如 C++可以用于编写这类应用程序，例如，访问操作系统的低级功能。但是，在 C#中，只能编写在托管环境下运行的代码。我们将使用 CLR 的托管功能，让.NET 自己与操作系统进行交互。

4. 无用存储单元收集

托管代码最重要的一个功能是无用存储单元收集(garbage collection)的概念。这个.NET 方法可确保应用程序不再使用某些内存时，这些内存就会被完全释放。在.NET 推出以前，这项工作主要由程序员负责，代码中的几个简单错误会把大块内存分配到错误的地方，使这些内存神秘失踪。这通常意味着计算机的速度逐渐减慢，最终导致系统崩溃。

.NET 无用存储单元收集会频繁检查计算机内存，从中删除不再需要的内容。它没有设置时间帧，可能一秒钟内会进行上千次的检查，也可能几秒钟检查一次，或者随时进行检查，但可以肯定进行了检查。

这里要给程序员一些提示。因为这项工作在不可预知的时间进行，所以在设计应用程序时，必须记得要进行这样的检查。需要许多内存才能运行的代码应自己执行这样的检查，而不是坐等收集无用存储单元，但这不像听起来那样难。

5. 把它们组合在一起

在继续之前，先总结一下上述创建.NET 应用程序所需要的步骤。

(1) 使用某种.NET 兼容语言(如 C#)编写应用程序代码,如图 1-1 所示。

(2) 代码被编译为 MSIL，存储在装配件中，如图 1-2 所示。



图　1-1

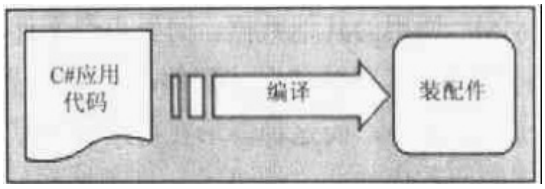


图　1-2

(3) 在执行代码时(如果这是一个可执行文件，就自动运行，或者在其他代码使用它时运行)，首先必须使用 JIT 编译器编译为本机代码，如图 1-3 所示。

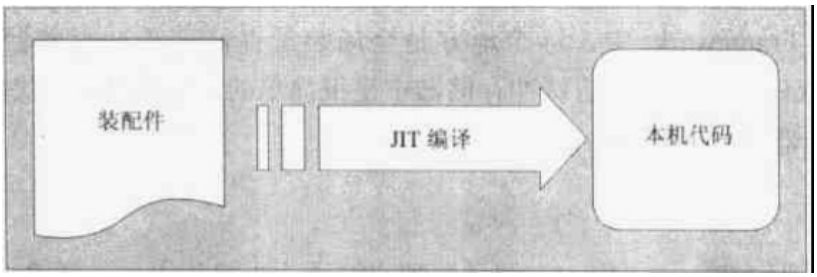


图　1-3

(4) 在托管的 CLR 环境下运行本机代码，以及其他应用程序或过程，如图 1-4 所示。

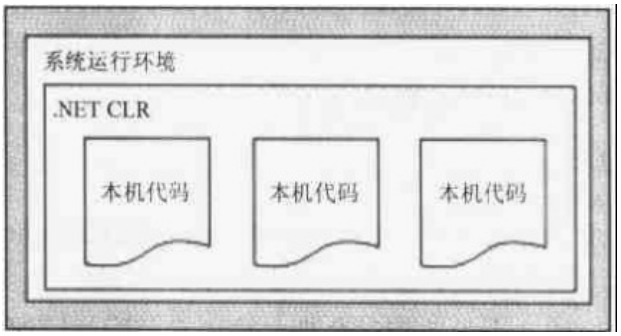


图　1-4

6. 链接

在上述过程中还有一点要注意。在第 2 步中编译为 MSIL 的 C#代码不需要包含在单独的文件中，可以把应用程序代码放在多个源代码文件中，再把它们编译到一个装配件中。这个过程称为链接，是非常有用的。原因是处理几个较小的文件比处理一个大文件简单得多。可以把逻辑上相关的代码分解到一个文件中，以便单独处理它，这也更易于在需要代码时找到它们，让开发小组把编程工作分解为可管理的块，让每个人编写一小块代码，而不会损害已编写好的部分或其他人正在处理的部分。

1.2　什么是 C#

如上所述，C#是可用于创建要运行在.NET CLR 上的应用程序的语言之一，它从 C 和 C++语言演化而来，是 Microsoft 专门为使用.NET 平台而创建的。因为 C#是近期发展起来的，所以

吸取了以前的教训，考虑了其他语言的许多优点，并解决了它们的问题。

使用 C#开发应用程序比使用 C++简单，因为其语法比较简单。但是，C#是一种强大的语言，在 C++中能完成的任务在 C#中也能完成。如前所述，C#中与 C++比较高级的功能等价的功能(例如直接访问和处理系统内存)，只能在标记为“不安全”的代码中使用。这个高级编程技术是非常危险的(正如它的名称)，因为它可能覆盖系统中重要的内存块，导致严重的后果。因此，本书不讨论这个问题。

C#代码常常比 C++略长一些。这是因为 C#是一种类型安全的语言(与 C++不同)。在外行人看来，这表示一旦为某些数据指定了类型，就不能转换为另一个不相关的类型。所以，在类型之间转换时，必须遵守严格的规则。执行相同的任务时，用 C#编写的代码通常比 C++长。但 C#代码更健壮，调试也比较简单，.NET 总是可以随时跟踪数据的类型。在 C#中，不能完成诸如“把 4 字节的内存放在数据中，并把它解释为 X”等的任务，但这并不是一件坏事。

C#只是.NET 开发的一种语言，但在我看来，这是最好的一种语言。C#的优点是，它是惟一为.NET Framework 而设计的语言，是在移植到其他操作系统上的.NET 版本中使用的主要语言。要使语言如 VB.NET 尽可能类似于其以前的语言，且仍遵循 CLR，就不能完全支持.NET 代码库的某些功能。但 C#能使用.NET Framework 代码库提供的每种功能。

1.2.1 用 C#能编写什么样的应用程序

如前所述，.NET Framework 没有限制应用程序的类型。C#使用.NET Framework，所以也没有限制应用程序的类型。这里仅讨论几种常见的应用程序类型。

- **Windows 应用程序** 这些应用程序如 Microsoft Office，有我们很熟悉的 Windows 外观和操作方式，使用.NET Framework 的 Windows Forms 模块就可以生成这种应用程序。Windows Form 模块是一个控件(例如按钮、工具栏、菜单等)库，其中的控件可以用于建立 Windows 用户界面(UI)。
- **Web 应用程序** 这些是 Web 页，可以通过任何 Web 浏览器查看。.NET Framework 包括一个动态生成 Web 内容的强大系统，允许个性化、实现安全性等。这个系统叫作 Active Server Pages.NET (ASP.NET)，我们可以使用 C#通过 Web Forms 创建 ASP.NET 应用程序。
- **Web 服务** 这是创建各种分布式应用程序的新方式，使用 Web 服务可以通过 Internet 虚拟交换数据。无论使用什么语言创建 Web 服务，也无论 Web 服务驻留在什么系统上，都使用一样简单的语法。

这些类型也需要某种形式的数据库访问，这可以通过.NET Framework 的 Active Data Objects.NET(ADO.NET)部分来实现。也可以使用许多其他资源，例如创建连网组件、输出图形、执行复杂数学任务的工具。

1.2.2 本书中的 C#

本书介绍了 C# 语言的语法和用法，但不过分强调.NET Framework。这是必需的，因为我们不能没有一点儿 C# 编程基础，就使用.NET Framework。首先介绍一些比较简单的内容，把面向对象编程(Object of Oriented Programming ,OOP)的问题放在基础知识的后面论述。假定读者没有一点儿编程的知识，这些是首要的规则。

本书还将介绍如何开发各种类型的应用程序，如何进行 Windows Forms 编程，以及其他有趣的.NET 问题(例如访问数据库)，另外还介绍了 Web 应用程序和 Web 服务编程。最后，详细介绍几个用本书前面的内容编写的案例。

1.3 Visual Studio.NET

本书使用 Visual Studio.NET (VS)进行所有的.NET 开发，包括简单的命令行应用程序，以及比较复杂的工程类型。VS 不是开发 C#应用程序所必需的，但使用它可以使任务更简单一些。可以在标准的文本编辑器中处理 C#源代码文件，例如常见的 Notepad，再使用命令行应用程序(是.NET Framework 的一部分)把代码编译到装配件中。但是，为什么要使用功能全面的 VS 呢？

下面列出的一些使 VS 成为.NET 开发的首选的功能。

- VS 可以自动执行编译源代码的步骤，同时可以完全控制重写它们时应使用的任何选项。
- VS 文本编辑器可以配合 VS 支持的语言(包括 C#)，这样就可以智能检测错误，在输入代码时给出合适的推荐代码。
- VS 包括 Windows Forms 和 Web Forms 设计器，允许 UI 元素的简单拖放设计。
- 在 C#中，许多类型的工程都可以用已有的“模板”代码来创建，不需要从头开始。各种代码文件通常已经为我们准备好了，减少了从头开始一个工程所花的时间。
- VS 包括几个可自动执行创建常用任务的向导，它们可以在已有的文件中添加合适的代码，而不需要考虑(在某些情况下)语法的正确性。
- VS 包含许多可视化和导航工程中元素的强大工具，它们可以是 C#源文件代码，也可以是其他资源，例如位图图像或声音文件。
- 除了在 VS 中编写应用程序比较简单外，还可以创建部署工程，以易于为客户提供代码，并方便地安装该工程。
- 在开发工程时，VS 可以使用高级调试技巧，例如能一次调试一行指令，并监视应用程序的状态。

C#还有许多功能，希望读者能掌握它们！

VS 解决方案

在使用 VS 开发应用程序时，可以通过创建解决方案来完成。在 VS 术语中，解决方案不仅仅是一个应用程序，它还包含工程，可以是 Windows Forms 工程、Web Form 工程等。但是，解决方案可以包含多个工程，这样，即使它们最终在硬盘上的多个地方编译为多个装配件，也可以把相关的代码组合到一个地方。

这是非常有用的，因为它可以处理“共享”代码(这些代码放在 Global Assembly Cache 中)，同时，应用程序也使用这段共享代码。在使用惟一的开发环境时，调试代码是非常容易的，因为可以在多个代码块中单步调试指令。

1.4 小结

本章简要介绍了.NET Framework，并讨论了如何轻松地创建各种强大的应用程序。还探讨了把用 C#等语言编写的代码转换为可工作的应用程序所需要做的工作，以及使用在.NET Common Language Runtime 下运行的托管代码有什么优点。

本章还阐述了 C#的实质，以及它与.NET Framework 的关系，描述了进行 C#开发使用的工具——Visual Studio.NET。

第 2 章介绍如何使用 VS 运行 C#代码，给出足够的基础知识，并集中讨论 C#语言本身，而不是过多地讨论 VS 的工作原理。

第2章　编写C#程序

本章用一定的篇幅讨论C#是什么，它是如何适应.NET Framework的，然后编写一些代码。本书主要使用Visual Studio.NET (VS)，所以首先介绍这个开发环境的一些基础知识。VS是一个非常复杂的、庞大的产品，可能会使初学者感到畏惧，但使用它创建简单的应用程序是非常容易的。在本章中，开始使用VS时，不需要了解许多知识，就可以编写C#代码。本书的后面将介绍VS能执行的更复杂的操作，现在仅介绍基础知识。

介绍完了VS后，将把两个简单的应用程序组合在一起。现在不要过多地考虑代码，后面将通过应用程序的创建过程，证明这个过程是有效的。

下面要创建的第一个应用程序是一个简单的控制台应用程序。控制台应用程序没有使用图形化的Windows环境，所以不需要担心按钮、菜单、用鼠标指针进行的交互等，而是在一个命令行窗口中运行应用程序，用更简单的方式与它交互。

第二个应用程序是一个Windows Forms应用程序，其外观和操作方式对Windows用户来说会非常熟悉，而且该应用程序创建起来不需要太多的精力。但所需代码的语法比较复杂，尽管在许多情况下，并不需要考虑细节。

本书后面的两个部分也使用这两种应用程序类型，但开始时略微强调一下控制台应用程序。在学习C#语言时，不需要了解Windows应用程序其他的灵活性能。控制台应用程序的简单性可以让我们集中精力学习语法，而无需考虑应用程序的外观和操作方式。

那么现在就开始吧！

2.1　Visual Studio.NET开发环境

VS在第一次加载时，会立即显示一系列窗口以及一组菜单和工具栏图标，其中的大多数窗口是空的。

如果是第一次运行VS，则屏幕上会为以前使用过这个开发环境的用户显示一个参数列表，这些参数的默认设置就很不错，现在接受它们——其中的参数都是可以修改的。

VS环境布局是完全可定制的，但默认设置很适合我们。其布局如下如图2-1所示。

在VS启动时，主窗口显示了一个介绍性的“开始页面”，该主窗口还会显示所有的代码。这个窗口有几个标签，单击文件名，就可以在文件之间切换。这个窗口也有其他功能：它可以显示图形用户界面，该界面可用于设计工程、纯文本文件、HTML以及各种内置于VS的工具。本书将陆续介绍它们。

在主窗口的上面，有工具栏和VS菜单。这里有几个不同的工具栏，其功能有保存和加载文件，建立和运行工程，以及调试控件等。在需要使用这些工具栏时将会讨论它们。



图　2-1

下面简要描述 VS 最常用的功能：

- 把鼠标放在 Server Explorer 和 Toolbox 上，就会弹出 Server Explorer 和工具箱工具栏，它们提供了许多其他的功能，例如访问服务器设置和服务程序，访问 Windows 应用程序的用户界面建立代码块。
- Solution Explorer 窗口显示当前加载的解决方案的信息。解决方案是一个或多个工程及其配置的 Visual Studio 术语。在 Solution Explorer 窗口中，可以查看解决方案中工程的各种视图，例如工程中包含了什么文件，这些文件中又包含了什么内容。
- Properties 窗口提供了工程内容的更详细视图，可以进行单个元素的其他配置。例如，使用这个窗口可以改变 Windows 窗体中一个按钮的外观。
- 任务列表和 Output 窗口显示了编译工程时的信息，以及已经完成的任务(与 Microsoft Outlook 中的任务列表类似)。这些任务可以是手工输入的，也可以是由 VS 自动生成的。

这似乎要理解很多东西，但不必担心，并不需要太多的时间来习惯这些。下面首先建立第一个范例工程，它将使用上面介绍的许多 VS 元素。

2.2　控制台应用程序

本书将频繁使用控制台应用程序，特别是开始时要使用这类应用程序，所以下面创建一个简单的控制台应用程序：

试试看：创建一个简单的控制台应用程序

(1) 选择 File | New | Project...菜单项，创建一个新的控制台应用程序工程，如图 2-2 所示。

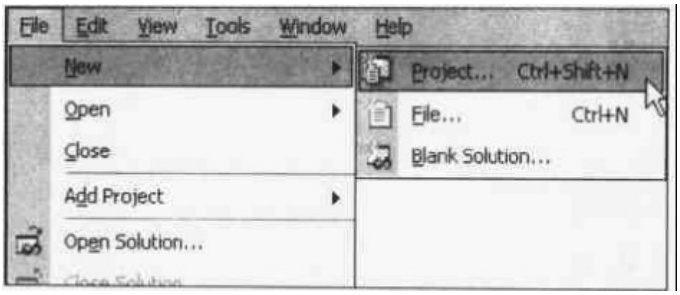


图 2-2

(2) 在所显示窗口的 Project Types:窗格中选择 Visual C# Projects 文件夹，在 Templates:窗格中选择 Console Application 工程类型(需要向下滚动一下)。把 Location:文本框改为 C:\BegCSharp\Chapter2(如果该目录不存在，会自动创建)。Name:文本框中的默认文本不变，如图 2-3 所示。

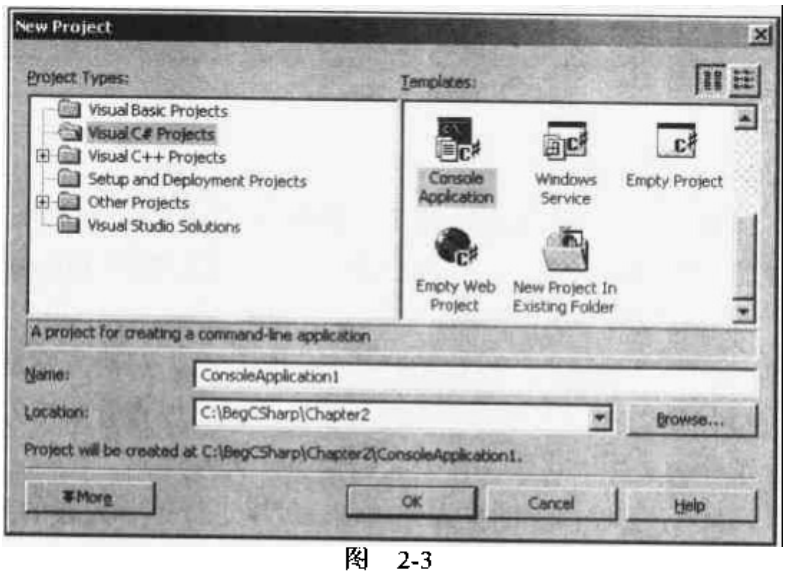


图 2-3

(3) 单击 OK 按钮。

(4) 初始化工程后，在主窗口中显示的文件里添加如下代码行：

```
using System;

namespace ConsoleApplication1
{
    /// <summary>
    /// Summary description for Class1.
    /// </summary>
    class Class1
    {
        static void Main(string[] args)
        {
            //
            // TODO: Add code to start application here
            //          static void Main(string[] args)
```

```
            Console.WriteLine("The first app in Beginning C# Programming!");
        }
    }
}
```

(5) 选择 Debug | Start Without Debugging 菜单项。过一会儿，就应得到如图 2-4 所示的结果。



图　2-4

(6) 按下任一键，退出应用程序。

代码的说明

现在我们不仔细研究这个工程中使用的代码，而关心如何使用 VS 启动和运行代码。可以看出，VS 做了许多工作，简化了编译和执行代码的过程。执行这些简单的步骤还有多种方式。例如，创建一个新工程可以使用 File | New | Project...菜单项，也可以按下 Ctrl+Shift+N，还可以单击工具栏上相应的图标。

同样，代码也可以用几种方式编译和执行。上面使用的方法是选择 Debug | Start Without Debugging 菜单项，也可以按下一个快捷键(Ctrl+F5)，或者使用工具栏中的图标。使用 Debug | Start 菜单项还可以以调试模式运行代码(也可以按下 F5 或单击相应的工具栏图标)，使用 Build | Build，Ctrl+Shift+B 或另一个工具栏图标可以编译工程但不运行它。编译好代码后，在 Windows Explorer 中运行生成的.exe 文件，就可以执行代码，也可以在命令行提示窗口中执行，为此，应打开一个命令行提示窗口，把目录改为 C:\BegCSharp\Chapter2\ ConsoleApplication1\ bin\Debug\，输入 ConsoleApplication1，并按下回车键。

在以后的范例中，我们仅说明“创建一个新的控制台工程”或“执行代码”，用户可以选择自己喜欢的方式执行这些步骤。除非特别声明，否则所有的代码都应在允许调试的情况下运行。

这里要注意的一点是，上例中的“Press any key to continue”仅在不调试的情况下运行代码时出现。如果以调试模式运行工程，它运行的控制台窗口就会在代码执行完毕后立即关闭。一般情况下这是对的，但对于上面的范例来说，有这个提示是有帮助的，这样窗口在弹出后，不会立即消失，否则将很难看到运行的结果。

前面创建了一个工程，现在详细讨论开发环境中的各组成部分。

首先要讨论的窗口是屏幕右上角的 Solution Explorer / Class View 窗口，它显示了两种操作模式(单击窗口底部的标签，就可以切换它们)，如图 2-5 所示。

Solution Explorer 视图显示了组成 ConsoleApplication1 工程的文件，包括在其中添加代码的文件 Class1.cs 和另一个代码文件 AssemblyInfo.cs(所有的 C#代码文件都是这种.cs 文件类型)。AssemblyInfo.cs 文件此时不需要考虑，它包含工程中目前我们不需要关心的其他信息。

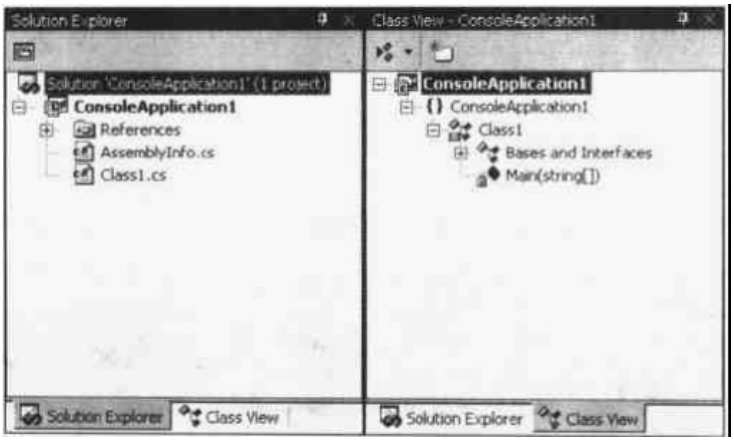


图　2-5

使用这个窗口可以改变主窗口中显示的代码，方法是：双击.cs 文件，或右击这些文件并选择 View Code，或选择它们，单击窗口顶部的工具栏按钮。还可以对代码文件执行其他操作，例如重新命名它们，或从工程中删除它们等。在该窗口中还可以显示其他类型的文件，例如工程资源(资源是工程使用的文件，这些文件可能不是 C#文件，例如位图图像和声音文件等)。可以通过同一界面处理它们。

References 项包含工程中使用的一个.NET 库列表，这个列表在后面介绍，因为标准引用很适合于初学者。

视图 Class View 显示了工程的另一种视图，可以查看刚才创建的代码结构。本书的后面将介绍代码结构，现在选择显示 Solution Explorer 视图。

单击这些窗口中的文件或其他图标，窗口的内容就会发生变化。这是另一个包含多个视图的窗口，其中最重要的是 Properties 视图，如图 2-6 所示。

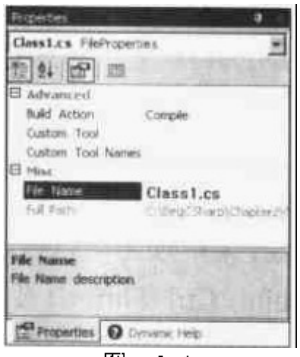


图　2-6

这个窗口显示了在上述窗口中所选项目的其他信息。例如，选择工程中的 Class1.cs 文件，就会显示上面的视图。这个窗口还显示了其他选中项目的信息，例如用户界面组件，就像本章的 Windows 应用程序中的用户界面组件那样。

通常在 Properties 窗口中对项目进行的改变会直接影响代码，添加代码行，或改变文件中的内容。对于一些工程来说，通过这个窗口来操作与手动修改代码所花的时间是相同的。

接着来看 Output 窗口。在执行范例代码时，Output 窗口会先显示一段文本，之后才显示应用程序创建的控制台窗口。在我的计算机上，显示的是：

```
------ Build started: Project: ConsoleApplication1, Configuration: Debug .NET ------

Preparing resources...
Updating references...
Performing main compilation...

Build complete -- 0 errors, 0 warnings
Building satellite assemblies...
```

```
---------- Done ----------

    Build: 1 succeeded, 0 failed, 0 skipped
    Deploy: 0 succeeded, 0 failed, 0 skipped
```

可以看出，这段文本显示了一个文件编译的状态报表。在这个报表中，还将报告编译过程中发生的任何错误。例如，在上一节添加的一行代码中，删除一个分号，重新编译，这次会得到：

```
------ Build started: Project: ConsoleApplication1, Configuration: Debug .NET ------

Preparing resources...
Updating references...
Performing main compilation...
c:\begcsharp\chapter2\consoleapplication1\class1.cs(15,73): error CS1002: ; expected
Build complete -- 1 errors, 0 warnings
Building satellite assemblies...

---------- Done ----------

    Build: 0 succeeded, 1 failed, 0 skipped
    Deploy: 0 succeeded, 0 failed, 0 skipped
```

这次工程不会运行。

注意：

在第 3 章介绍 C#语法时，就会明白分号在大多数代码行的末尾，它是必不可少的。

为了使代码能够执行，VS 自动在任务列表中添加了一个任务，而任务列表和 Output 窗口位于同一个地方，如图 2-7 所示。

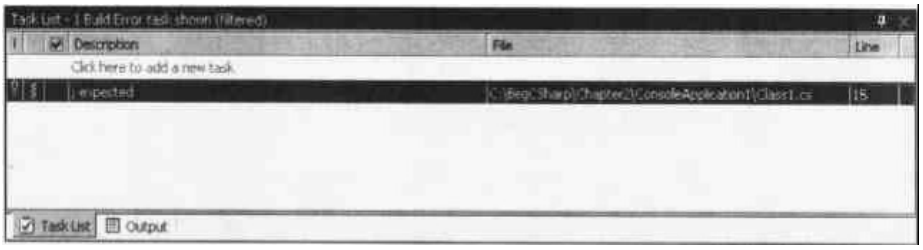


图　2-7

这个窗口有助于根除代码中的错误，因为它会跟踪我们的工作，编译工程。如果双击该窗口中显示的错误，光标就会跳到源代码中出现错误的地方(如果包含错误的源文件没有打开，将被打开)，这样就可以快速更正错误。代码中有错误的一行会出现红色的波浪线，便于我们快速扫描源代码，找出错误。

注意错误的位置用一个行号来指定。在默认情况下，行号不会显示在 VS 文本编辑器中，

但有时需要显示它。为此，需要单击 Tools | Options...菜单项，选中 Options 对话框中相应的复选框。该复选框是 Line numbers，在 Text Editor | C# | General 类别中，如图 2-8 所示。

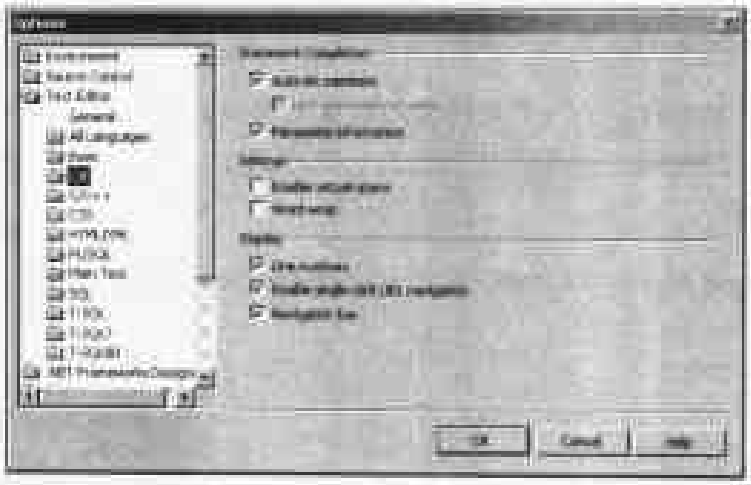


图 2-8

这个对话框中有许多有用的选项，本书将使用其中的几个。

2.3 Windows Forms 应用程序

通常，把代码当作 Windows 应用程序的一部分来说明，要比通过控制台窗口或命令行提示简单一些。下面用用户界面建立块来组合一个用户界面。

现在介绍建立用户界面的基础知识，说明如何启动和运行 Windows 应用程序，但并不详细讨论应用程序实际完成的工作。本书的后面会详细研究 Windows 应用程序。

试试看：创建一个简单的 Windows 应用程序

(1) 在以前的位置(C:\BegCSharp\Chapter2)创建一个类型为 Windows Application 的新工程，其默认名称是 WindowsApplication1。如果第一个工程仍处于打开状态，应确保选择 Close Solution 选项，以便开始一个新解决方案，如图 2-9 所示。

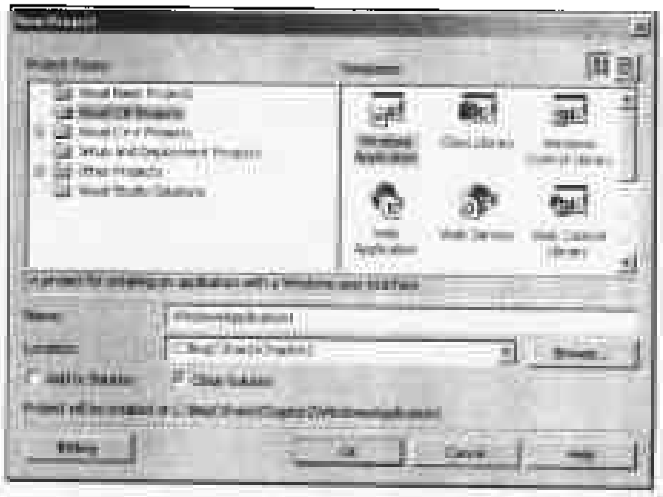


图 2-9

(2) 把鼠标指针移动到屏幕左边的 Toolbox 栏上，然后移到 Windows Forms 选项卡上的 Button 选项，在该选项上双击鼠标左键，以在应用程序的主窗体(Form1)上添加一个按钮，如图 2-10 所示。

(3) 双击刚才添加到窗体的按钮。

(4) 现在应显示 Form1.cs 中的 C#代码。进行如下修改(为了简短起见，这里只显示了文件中的部分代码)：

```
private void button1_Click(object sender, System.EventArgs e)
{
    MessageBox.Show("The first windows app in the book!");
}
```

(5) 运行应用程序。

(6) 单击显示出来的按钮，打开一个消息对话框，如图 2-11 所示。

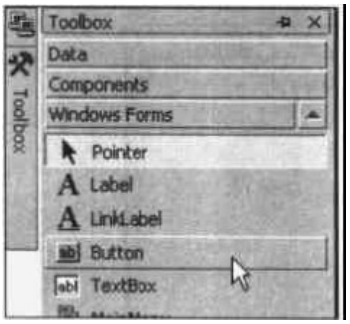


图　2-10



图　2-11

代码的说明

VS 又一次完成了许多工作，使我们能不费吹灰之力就可以完成一个实用的 Windows 应用程序的创建。刚才创建的应用程序与其他窗口的行为方式相同——可以移动、重新设置其大小、最小化等。我们不必编写任何代码，它就可以工作。我们添加的按钮也是这样。双击按钮，VS 就知道我们想添加一些代码，当运行应用程序时，用户单击该按钮，就执行我们编写好的代码。只要提供了这段代码，就可以得到按钮单击的所有功能。

当然，Windows 应用程序不仅限于带有按钮的普通窗体。如果看看从中选择 Button 选项的工具栏，就会看到一整套用户界面建立块，其中一些用户可能很熟悉，但另外一些则不太熟悉。本书在某些地方将使用其中的大多数用户界面建立块，它们使用起来都非常简单，可以节省许多时间和精力。

应用程序的代码在 Form1.cs 中，看起来要比上一节提供的代码复杂得多。但实际上，这些代码看起来比较复杂，原因是：这段代码比较多，这个文件中的许多代码都与窗体上控件的布局有关。这就是为什么可以在设计窗口中查看设计视图中的代码的原因，而设计视图正是这些布局代码的可视化转换。按钮就是可以使用的一种控件，同样，也可以使用 Toolbox 上 Windows Forms 部分中的其他 UI 建立块。

下面把按钮作为一个控件范例，详细解释一下。使用主窗口中的标签，切换回窗体的 DesignView，单击按钮，选择它。此时，屏幕右下角的 Properties 窗口就会显示按钮控件的属性(控件的属性非常类似于上例中文件的属性)，向下滚动到 Text 属性，它目前设置为 button1，把其值改为 Click Me，如图 2-12 所示。



图　2-12

在 Form1 中，按钮上的文本应反映这个变化。

这个按钮有许多属性，从按钮颜色和大小的简单格式化，到某些模糊设置(如数据绑定设置，它可以连接数据库)，应有尽有。如上例所述，改变属性通常会直接改变代码，这也不例外。切换回Form1.cs的代码视图，看看刚才进行的改变。

匆匆一瞥，可能注意不到代码中有什么不同，这是因为 C#代码中处理窗体上控件的布局和格式化的部分被隐藏了(毕竟，如果有了结果的图形显示，就不需要查看代码了)。

VS 使用一个代码突出显示系统来显示这部分 C#代码，可以在如图 2-13 所示的屏幕图中看到。

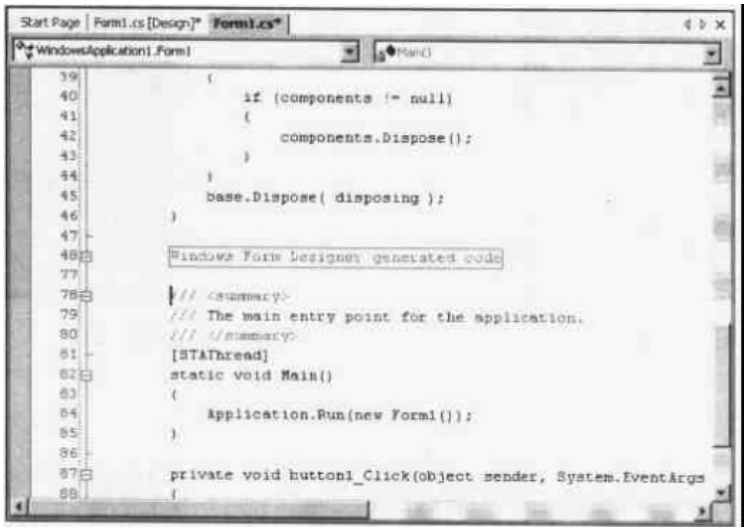


图　2-13

在代码的左边(如果已显示行号，就在行号的旁边)，注意有一些灰色的线条，其上有+和－号的方框。这些方框可用于扩展和收缩代码区域。在文件的中部(在我的机器上是第 48 行，但这可能与用户机器中显示的不同)，有一个带+号的方框，它对应代码主体中的“Windows Form Designer generated code”。这个标签的基本含义是“这里有一些 VS 生成的代码，用户不需知道”。如果愿意，可以查看改变按钮属性的代码。只要单击带有+号的方框，代码就会显示出来，在某处地方应有如下代码：

```
this.button1.Text = "Click Me";
```

不必过多地考虑这行代码，我们可以看到在 Properties 窗口中输入的文本直接显示在代码中。

在编写代码时，这种突出显示代码的方式非常方便。我们可以扩展和收缩其他区域，而不

仅仅是正常情况下隐藏的代码段。查看一本书的目录，有助于快速了解该书的内容；查看一组收缩的代码段，则非常便于浏览大量的 C#代码。

2.4　小结

本章介绍了贯穿本书所使用的一些工具。快速浏览了 Visual Studio.NET 开发环境，并使用它建立了 2 种类型的应用程序。其中比较简单的是控制台应用程序，它足以满足你的大多数需要，并允许你可以集中精力学习 C#编程的基础知识。Windows 应用程序比较复杂，但其可视化程度比较高，对于习惯了 Windows 环境的人来说，使用起来也比较直观。

知道如何创建简单的应用程序，就可以学习 C#了。本书的下面的章节将介绍 C#的基本语法和程序结构，之后讨论更高级的面向对象方法。学习了这些内容后，就可以开始了解如何使用 C#访问.NET Framework 的功能了。

第3章　变量和表达式

计算机程序最基本的描述也许是一系列处理数据的操作，即使是最复杂的范例，这个论述也是正确的，例如 Microsoft Office 套装软件之类大型多功能的 Windows 应用程序。应用程序的用户虽然看不到它们，但这些操作总是在后台上进行。

为了进一步解释它，考虑一下计算机的显示单元。我们常常比较熟悉屏幕上的内容，很难不把它想像为“移动的图片”。但实际上，我们看到的仅是一些数据的显示结果，其最初的形式是存储在计算机内存中的 0 和 1 数据流。因此我们在屏幕上进行的任何操作，无论是移动鼠标指针，单击图标，或在字处理器上输入文本，都会改变内存中的数据。

当然，还有一些不太抽象的情形来说明这一点。如果使用计算器应用程序，就要提供数字，对这些数字执行操作，就像用纸和笔计算数字一样，但使用程序会快得多。

如果计算机程序是在对数据执行操作，这说明我们需要某种存储数据的方式，以及处理它们的一些方法。这两种功能是由变量和表达式提供的，本章将探究它们的含义。

在开始之前，应先了解一下 C#编程的基本语法。

3.1　C#的基本语法

C#代码的外观和操作方式与 C++和 Java 非常类似。初看起来，其语法可能比较混乱，不像英语和其他语言。但是，在 C#编程中，使用的样式是比较清晰的，不用花太多的力气就可以编写出可读性很强的代码。

与其他语言的编译器不同，无论代码中是否有空格、回车符或 tab 字符(这些字符统称为空白字符)，C#编译器都不考虑这些字符。这样格式化代码时就有很大的自由度，但遵循某些规则将有助于使代码易于阅读。

C#代码由一系列语句组成，每个语句都用一个分号表示结束。因为空格被忽略，所以一行可以有多个语句，但从可读性的角度来看，通常在分号的后面加上回车符，这样就不能在一行上放置多个语句了。但一句代码放在多个行上是可以的(也比较常见)。

C#是一个块结构的语言，所有的语句都是代码块的一部分。这些块用花括号来界定("{" 和 "}")，可以包含任意多行语句，或者根本不包含语句。注意花括号字符不需要附带分号。

所以，简单的 C#代码块如下所示：

```
{
    <code line 1, statement 1>;
    <code line 2, statement 2>
        <code line 3, statement 2>;
}
```

其中<code line *x*, statement *y*>部分并不是真正的 C#代码，而是用这个文本作为 C#语句的占位符。注意在这段代码中，第 2、3 行代码行是同一个语句的一部分，因为在第 2 行的末尾没有分号。

在这个简单的代码块中，还使用了缩进格式，使 C#代码的可读性更高。这不是我的发明，而是一个标准规则，实际上在默认情况下 VS 会自动缩进代码。一般情况下，每个代码块都有自己的缩进级别，即它向右缩进了多远。代码块可以互相嵌套(即块中可以包含其他块)，而被嵌套的块要缩进得多一些。

```
{
    <code line 1>;
    {
        <code line 2>;
        <code line 3>;
    }
    <code line 4>;
}
```

前面代码的续行通常也要缩进得多一些，如上面第一个范例中的第 3 行代码。

记住，这种样式并不是强制的。但如果不使用它，读者在阅读本书时会很快陷入迷茫之中。

在 C#代码中，另一个常见的语句是注释。注释并不是严格意义上的 C#代码，但代码最好有注释。注释就是解释，即给代码添加描述性文本(用纯英语[或法语、德语等了])，编译器会忽略这些内容。在开始处理比较长的代码段时，给进行的工作添加提示，例如“这行代码要求用户输入一个数字”，或“这段代码由 Bob 编写”。C#添加注释的方式有两种。可以在注释的开头和结尾放置标记，也可以使用一个标记，其含义是“这行代码的其余部分是注释”。在 C#编译器忽略回车符的规则中，后者是一个例外，但这是一种特殊情况。

要使用第一种方式标记注释，可以在注释的开头加上“/*”，在末尾加上“*/”。这些注释符号可以在单独一行上，也可以在不同的行上，注释符号之间所有内容都是注释。注释中惟一不能输入的是“*/”，因为它会被看作注释结束标记。所以下面的语句是正确的。

```
/* This is a comment */

/* And so...

                ... is this! */
```

但下面的语句会产生错误：

```
/* Comments often end with "*/" characters */
```

注释结束符号后的内容("*/"后面的字符)会被当作 C#代码，因此产生错误。

另一个添加注释的方法是用“//”开始一个注释，其后可以编写任何内容，只要这些内容在一行上即可。下面的语句是正确的：

```
// This is a different sort of comment.
```

但下面的语句会失败，因为第二行代码会解释为 C#代码：

```
// So is this,
  but this bit isn't.
```

这类注释可用于语句的说明，因为它们都放在一行上：

```
<A statement>;              // Explanation of statement
```

在 C#中，还有第三类注释，用于说明代码。它们都是单行注释，用三个"/"符号来开头，而不是两个。

```
/// A special comment
```

在正常情况下，编译器会忽略它们，就像其他注释一样，但可以配置 VS，在编译工程时，提取注释后面的文本，创建一个特殊格式的文本文件，该文件可用于创建文档说明书。具体内容见第 18 章。

一个特别要注意的一点是 C#代码是大小写敏感的。与其他语言不同，必须使用正确的大小写形式输入代码，因为简单地用大写字母代替小写字母会中断工程的编译。

C#控制台应用程序的基本结构

下面看看第 2 章的控制台应用程序范例(ConsoleApplication1)，研究一下它的结构。其代码如下所示：

```
using System;

namespace ConsoleApplication1
{
    /// <summary>
    /// Summary description for Class1.
    /// </summary>
    class Class1
    {
        static void Main(string[] args)
        {
            //
            // TODO: Add code to start application here
            //        static void Main(string[] args)
            Console.WriteLine("The first app in Beginning C# Programming!");
        }
    }
}
```

可以立即看出，上一节讨论的所有语法元素这里都有。其中有分号、花括号、注释和适当的缩进。

在目前看来，代码中最重要的部分如下所示：

```
static void Main(string[] args)
{
    //
    // TODO: Add code to start application here
    //        static void Main(string[] args)
    Console.WriteLine("The first app in Beginning C# Programming!");
}
```

在运行控制台应用程序时，就运行这段代码，更准确地说，是运行花括号中的代码块。其中完成任务的代码行是在自动生成的代码中添加的那行代码，这是代码块中惟一不是注释的代码，这行代码在控制台窗口中输出了一些文本，但目前我们还不需要关心它的具体机制。

在这个范例中我们还没有考虑其他代码，因为本书前几章仅解释 C#的基本语法，至于应用程序进行 Console.WriteLine()调用的具体方式，则不在我们的考虑之内。以后会阐述这些代码的重要性。

3.2　变量

如本章的引言所述，变量关系到数据的存储。实际上，可以把计算机内存中的变量看作架子上的盒子。在这些盒子中，可以放入一些东西，再把它们取出来，或者只是看看盒子里是否有东西。变量也是这样，数据可放在变量中，可以从变量中取出数据或查看它们。

尽管计算机中的所有数据都是相同的东西(一组 0 和 1)，但变量有不同的内涵称为类型。下面再使用盒子来类比，盒子有不同的形状和尺寸，某些东西只能放在特定的盒子中。建立这个类型系统的原因是不同类型的数据需要用不同的方法来处理。变量限定为不同的类型，可以避免混淆它们。例如，组成数字图片的 0 和 1 序列与组成声音文件的 0 和 1 序列，其处理方式是不同的。

要使用变量，需要声明它们。即给变量指定一个名称和一种类型。声明了变量后，就可以把它们用作存储单元，存储声明的数据类型的数据。

声明变量的 C#语法是，指定类型和变量名，如下所示：

```
<type> <name>;
```

如果使用未声明的变量，代码就不会被编译，但此时编译器会告诉我们发生了什么问题，所以这不是一个致命错误。另外，使用未赋值的变量也会产生一个错误，编译器会检测出这个错误。

那么我们可以使用什么类型呢？

实际上，可以使用的类型数量是无限多的。其原因是可以自己定义类型，存储各种复杂的数据。

尽管如此，总有一些数据类型是每个人都要使用的，例如存储数值的变量。因此我们应了解一些简单的预定义类型。

3.2.1 简单类型

简单类型就是组成应用程序中基本组成部件的类型，例如数值和布尔值 (true 或 false)。简单类型还可以组成比较复杂的类型。大多数简单类型都是存储数值的，初看起来有点奇怪，肯定只需要一种类型存储数值吗？

数值类型过多的原因是在计算机内存中，把数字作为一系列的 0 和 1 来存储的机制。对于整数值，用一定的位(单个数字，可以是 0 或 1)来存储，用二进制格式来表示。以 N 位来存储的变量可以表示任何介于 0 到 $(2^N - 1)$之间的数。大于这个值的数则太大，不能存储在这个变量中。

例如，有一个变量存储了 2 位，在整数和表示该整数的位之间的映射应如下所示：

0 = 00
1 = 01
2 = 10
3 = 11

如果要存储更大的数，就需要更多的位(例如，3 位可以存储 0~7 的数)。

这个论点的结论是要存储每个可以想像得到的数，就需要非常多的位，这并不适合 PC。即使我们可以用足够多的位来表示每一个数，变量使用这些位来存储它，其效率也非常低下，例如，如果只需要存储从 0 到 10 之间的数 (因为存储器被浪费了)。其实 4 位就足够了，可以用相同的内存空间存储这个范围内的更多数值。

相反，许多不同的整数类型可以用于存储不同范围的数值，占用不同的内存空间(至多 64 位)，其列表如表 3-1 所示。

注意：

这些类型中的每一种都利用了.NET Framework 中定义的标准类型。如第 1 章所述，使用标准类型可以在语言之间交互操作。在 C#中这些类型的名称是.NET Framework 中定义的别名，表 3-1 列出了这些类型在.NET Framework 库中的名称。

表 3-1

类　型	别　名	允 许 的 值
sbyte	System.SByte	在 －128~127 之间的整数
byte	System.Byte	在 0~255 之间的整数
short	System.Int16	在－32768~32767 之间的整数
ushort	System.UInt16	在 0 ~65535 之间的整数
int	System.Int32	在－2147483648~2147483647 之间的整数
uint	System.UInt32	在 0~4294967295 之间的整数
long	System.Int64	在 －9223372036854775808~9223372036854775807 之间的整数
ulong	System.UInt64	在 0~18446744073709551615 之间的整数

一些变量名称前面的"u"是"unsigned"的缩写，表示不能在这些类型的变量中存储负号，参见该表中的“允许的值”一列。

当然，除了整数以外，还可以存储浮点数，它们不是整数。可以使用的浮点数变量类型有 3 种：float, double 和 decimal。前两种可以用+/－m×2^e等形式存储浮点数，m 和 e 的值随着类型的不同而不同。Decimal 使用另一种形式：+/－m×10^e。这 3 种类型、其 m 和 e 的值，以及它们在实数中的限制如表 3-2 所示。

表　3-2

类　型	别　名	m 的最小值	m 的最大值	e 的最小值	e 的最大值	近似的最小值	近似的最大值
float	System.Single	0	224	－149	104	1.5×10^{-45}	3.4×10^{38}
double	System.Double	0	253	－1075	970	5.0×10^{-324}	1.7×10^{308}
decimal	System.Decimal	0	296	－26	0	1.0×10^{-28}	7.9×10^{28}

除了数值类型外，还有另外 3 种简单类型，如表 3-3 所示。

表　3-3

类　型	别　名	允 许 的 值
Char	System.Char	一个 Unicode 字符，0~65535 之间的整数
bool	System.Boolean	布尔值：true 或 false
string	System.String	一组字符

注意组成 string 的字符数没有上限，因为它可以使用可变大小的内存。

布尔类型 bool 是 C#中最常用的一种变量类型，类似的类型在其他语言的代码中非常丰富。当编写应用程序的逻辑流程时，一个可以是 true 或 false 的变量有非常重要的分支作用。例如，考虑一下有多少问题可以用 true 或 false(或 yes 和 no)来回答。执行变量值之间的比较或检查输入的有效性就是后面使用布尔变量的两个编程范例。

介绍了这些类型后，下面用一个小范例来声明和使用它们。

试试看：使用简单类型的变量

(1) 在目录 C:\BegCSharp\Chapter3 下创建一个新的控制台应用程序 Ch03Ex01。

(2) 给 Class1.cs 添加如下代码(删除注释行)：

```
static void Main(string[] args)
 {
    int myInteger;
    string myString;
    myInteger = 17;
    myString = "\"myInteger\" is";
    Console.WriteLine("{0} {1}.", myString, myInteger);
 }
```

(3) 执行代码(记住在调试模式下运行，会导致在查看结果前，关闭控制台应用程序窗口)，如图 3-1 所示。



图 3-1

代码的说明

我们添加的代码完成了 3 项任务：

- 声明两个变量
- 给这两个变量赋值
- 将两个变量的值输出到控制台

变量声明使用下述代码：

```
int myInteger;
string myString;
```

第一行声明一个类型为 int 的变量 myInteger，第二行声明一个类型为 string 的变量 myString。

提示：

变量的命名是有限制的，不能使用任意的字符序列。本节的后面将介绍命名变量的规则。

接下来的两行代码给变量赋值：

```
myInteger = 17;
myString = "\"myInteger\" is";
```

使用=赋值运算符(在本章的“表达式”一节中详细介绍)给变量分配两个固定的值(在代码中称为字面值)。把整数值 17 赋给 myInteger，把字符串"myInteger"(包括引号)赋给 myString。以这种方式给字符串赋予字符串字面值时，注意必须用双引号把字符串括起来。因此，如果字符串本身包含双引号，就会出现错误，必须用一些表示这些引号字符的其他字符(即转义序列)来替代它们。在本例中，使用序列\"来转义一个双引号：

```
myString = "\"myInteger\" is";
```

如果不使用这些转义序列，而输入如下代码：

```
myString = ""myInteger" is";
```

就会出现编译错误。

注意给字符串赋予字面值时，必须小心换行 —— C#编译器会拒绝分布在多行上的字符串字面值。如果要添加一个换行符，可以在字符串中使用回车换行符的转义序列，即\n。例如，赋值语句：

```
myString = "This string has a\nline break.";
```

会在控制台视图中显示两行，如下所示：

```
This string has a
line break.
```

所有的转义序列都包含一个反斜杠符号，后跟一个小字符组合(详见后面的内容)，因为反斜杠符号的这种用途，它本身也有一个转义序列，即两个连续的反斜杠\\。

下面继续解释代码，还有一行没有说明：

```
Console.WriteLine("{0} {1}.", myString, myInteger);
```

它看起来类似于第一个范例中把文本写到控制台上的简单方法，但本例指定了变量。这里不打算详细讨论这行代码。这是本书第一部分用于给控制台窗口输出文本的一种技巧，知道这一点就足够了。在括号中，有两类参数：

- 一个字符串
- 一个用逗号分隔开的变量列表，这些变量的值将插入到输出字符串中

输出的字符串是"{0} {1}."，它们并没有包含有用的文本。可以看出，这并不是我们运行代码时希望看到的结果，其原因是：字符串实际上是插入变量内容的一个模板，字符串中的每对花括号都是一个占位符，包含列表中每个变量的内容。每个占位符(或格式字符串)用包含在花括号中的一个整数来表示。整数以 0 开始，每次递增 1，占位符的总数应等于列表中指定的变量数，该列表用逗号分隔开，跟在字符串后。把文本输出到控制台时，每个占位符就会用每个变量的值来替代。在上面的范例中，{0}用第一个变量的值 myString 替换，{1}用 myInteger 的内容来替换。

在后面的范例中，就使用这种给控制台输出文本的方式显示代码的输出结果。

3.2.2　变量的命名

如上一节所述，不能把任意序列的字符作为变量名。这并不像第一次听起来那样需要担心什么，因为这种命名系统仍是非常灵活的。

基本的变量命名规则如下：

- 变量名的第一个字符必须是字母、下划线("_")或"@"。
- 其后的字符可以是字母、下划线或数字。

另外，有一些关键字对于 C#编译器而言有特定的含义，例如前面出现的 using 和 namespace 关键字。如果错误地使用其中一个关键字，编译器会产生一个错误，我们马上就会知道做错了，所以不必担心。

例如，下面的变量名是正确的：

```
myBigVar
VAR1
_test
```

但这些变量名不正确：

```
99BottlesOfBeer
```

```
namespace
It's-All-Over
```

记住，C#是区分大小写的，所以必须小心，不要忘了在声明变量时使用正确的大小写。在程序中引用它们时，即使只有一个字母的大小写形式出错，都不能编译成功。

其进一步的结果是得到多个变量，其名称仅有大小写的区别，例如下面的变量都是不同的：

```
myVariable
MyVariable
MYVARIABLE
```

命名约定

变量名是比较常用的，所以有必要用一定的篇幅讨论几种要用到的变量名称。在开始前，要记住这是有争议的。多年以来，出现了不同的系统，一些开发人员拼命维护他们的个人系统。

最近，最流行的系统是所谓的 Hungarian 记号法。这个系统在所有的变量名上加上一个小写形式的前缀，表示其类型。例如，如果变量的类型是 int，就在其名称前加上 i(或 n)，如 iAge。使用这个系统，可以快速看出各个变量是什么类型的。

更现代的语言如 C#灵活地实现了这些系统。与前面介绍的所有类型一样，可以用一两个字母前缀表示变量的类型。但由于可以创建自己的类型，而且在.NET Framework 中有上百种更复杂的类型，所以这种系统很快就失效了。在多人完成的工程中，不同的人很容易遇到易混淆的不同前缀，它们可能导致灾难性的后果。

开发人员现在认识到，最好根据变量的作用来命名它们。如果出现问题，就很容易确定变量的类型。在 VS 中，只需把鼠标指针在变量名上停上足够长的时间，就会弹出一个方框，说明该变量的类型。

目前，在.NET Framework 名称空间中有两种命名约定，称为 PascalCase 和 camelCase。在名称中使用的大小写表示它们的用途。它们都应用到由多个单词组成的名称中，并指定名称中的每个单词除了第一个字母大写外，其余字母都是小写。在 camelCasing 中，还有一个规则，即第一个单词以小写字母开头。

下面是 camelCase 变量名：

```
age
firstName
timeOfDeath
```

下面是 PascalCase 变量名：

```
Age
LastName
WinterOfDiscontent
```

Microsoft 建议：对于简单的变量，使用 camelCase 规则，而比较高级的命名则使用 PascalCase。

最后，注意许多以前的命名系统常常使用下划线字符作为变量名中各个单词之间的分隔符，例如 my_first_variable。这种用法现在已经淘汰了。

3.2.3　字面值

在前面的范例中，有两个字面值的范例：整数和字符串。其他变量类型也有相关的字面值，如表 3-4 所示。其中有许多涉及到后缀，即在字面值的后面添加一些字符，指定想要的类型。一些字面值有多种类型，在编译时由编译器根据它们的上下文确定其类型。

表 3-4

类　型	类　别	后　缀	范例/允许的值
Bool	布尔	无	true 或 false
int, uint, long, ulong	整数	无	100
uint, ulong	整数	u 或 U	100U
long, ulong	整数	l 或 L	100L
ulong	整数	Ul, uL, Ul, UL, lu, lU, Lu 或 LU	100UL
float	实数	f 或 F	1.5F
double	实数	无 d 或 D	1.5
decimal	实数	M 或 M	1.5M
char	字符	无	'a', 或转义序列
string	字符串	无	"a...a",可以包含转义序列

字符串的字面值

在本章的前面，介绍了几个可以在字符串的字面值中使用的转义序列，表 3-5 是这些转义序列的完整列表，以便以后引用。

表 3-5

转 义 序 列	产生的字符	字符的 Unicode 值
\'	单引号	0x0027
\"	双引号	0x0022
\\	反斜杠	0x005C
\0	空	0x0000
\a	警告(产生鸣响)	0x0007
\b	退格	0x0008
\f	换页	0x000C
\n	换行	0x000A
\r	回车	0x000D
\t	水平制表符	0x0009
\v	垂直制表符	0x000B

表 3-5 中的 Unicode 值列是字符在 Unicode 字符集中的 16 进制值。

与上面一样，使用 Unicode 转义序列可以指定 Unicode 字符，该转义序列包括标准的\字符，后跟一个 u 和一个 4 位十六进制值(例如，表 3-5 中的 x 后面的 4 位数字)。

下面的字符串是等价的：

```
"Karli\'s string."
"Karli\u0027s string"
```

显然，Unicode 转义序列还有更多的用途。

也可以逐字地指定字符串，即两个双引号之间的所有字符都包含在字符串中，包括行末字符和需要转义的字符。惟一的例外是，双引号字符的转义，它们必须指定，以避免结束字符串。为此，在该字符串的前面加一个@字符：

```
@"Verbatim string literal."
```

这个字符串可以用一般的方式指定，但需要下面这种方式：

```
@"A short list:
item 1
item 2"
```

逐字指定的字符串在文件名中非常有用，因为文件名中大量使用了反斜杠字符。如果使用一般的字符串，就必须在字符串中使用 2 个反斜杠，例如：

```
"C:\\Temp\\MyDir\\MyFile.doc"
```

而有了逐字指定的字符串字面值，这段代码的可读性就比较高。下面的字符串与上面的等价：

```
@"C:\Temp\MyDir\MyFile.doc"
```

注意：

从本书的后面可以看出，字符串是引用类型，而本章中的其他类型都是值类型。其结果是字符串也可以指定值为 null，即字符串变量不引用字符串。具体内容见本书后面的内容。

3.2.4 变量声明和赋值

快速回忆一下，前面使用变量的类型和名称来声明它们，例如：

```
int age;
```

然后用＝赋值运算符给变量赋值：

```
age = 25;
```

注意：

变量在使用前，必须初始化。上面的赋值语句可以用作初始化语句。

这里还可以做两件事，用户可以在 C#代码中看到。第一是同时声明多个类型相同的变量，方法是：在类型的后面用逗号分隔变量名，如下所示：

```
int xSize, ySize;
```

其中 xSize 和 ySize 都声明为整数类型。

第二个技巧是在声明变量的同时为它们赋值，即把两行代码合并在一起：

```
int age = 25;
```

可以同时使用这两个技巧：

```
int xSize = 4, ySize = 5;
```

xSize 和 ySize 被赋予不同的值：

注意下面的代码：

```
int xSize, ySize = 5;
```

其结果是 ySize 被初始化，而 xSize 仅进行了声明。

3.3 表达式

前面介绍了如何声明和初始化变量，下面该处理它们了。C#包含许多进行这类处理的运算符，包括前面已经使用过的=赋值运算符，把变量和字面值(在使用运算符时，它们都称为操作数)与运算符组合起来，就可以创建表达式，它是计算的基本建立块。

运算符的范围非常广泛，有简单的，也有非常复杂的，其中一些可能只在数学应用程序中使用。简单的操作包括所有的基本数学操作，例如+运算符是把两个操作数加在一起，而复杂的操作则包括通过变量内容的二进制表示来处理它们。还有专门用于处理布尔值的逻辑运算符，和赋值运算符=。

本章主要介绍数学和赋值运算符，而逻辑运算符在第 4 章中介绍，主要论述控制程序流程的布尔逻辑。

运算符大致分为 3 类。

- 一元运算符，处理一个操作数
- 二元运算符，处理两个操作数
- 三元运算符，处理三个操作数

大多数运算符都是二元运算符，只有几个一元运算符和一个三元运算符，即条件运算符(条件运算符是一个逻辑运算符，它返回一个布尔值，详见第 4 章)。

下面先介绍数学运算符，它包括一元运算符和二元运算符。

3.3.1 数学运算符

有 5 个简单的数字运算符，其中 2 个有二元和一元两种形式。在表 3-6 中，列出了这些运

算符，并用一个小范例来说明它们的用法，以及使用简单的数值类型(整数和浮点数)时它们的结果。

表 3-6

运 算 符	类 别	范例表达式	结 果
+	二元	var1 = var2 + var3;	var1 的值是 var2 和 var3 的和
-	二元	var1 = var2 - var3;	var1 是从 var2 的值减去 var3 的值所得的值
*	二元	var1 = var2 * var3;	var1 的值是 var2 和 var3 的乘积
/	二元	var1 = var2 / var3;	var1 是 var2 除以 var3 所得的值
%	二元	var1 = var2 % var3;	var1 是 var2 除以 var3 所得的余数
+	一元	var1 = +var2;	var1 的值等于 var2 的值
-	一元	var1 = - var2;	var1 的值等于 var2 的值除乘以 - 1

上面的范例都使用简单的数值类型作为结果，使用其他简单类型可能描述得不太清晰。如果把两个布尔值加在一起，会得到什么结果？此时，如果对 bool 变量使用+(或其他数学运算符)，编译器不会报告出错。char 变量的相加也会有点让人摸不着头脑。记住，char 变量实际上存储的是数字，所以把两个 char 变量加在一起也会得到一个数字(其类型为 int)。这是一个隐式转换的范例，稍后将详细介绍这个主题和显式转换，因为它也可以应用到 var1, var2 和 var3 都是混合类型的情况。

如前所述，二元运算符+在用于字符串类型变量时也是有意义的。此时，表 3-7 的表项应是：

表 3-7

运 算 符	类 别	范例表达式	结 果
+	二元	var1 = var2 + var3;	var1 的值是把存储在 var2 和 var3 中的字符串连接起来

但其他数学运算符不能用于字符串的处理。

这里应介绍的另外两个运算符是递增和递减运算符，它们都是一元运算符，可以以两种方式使用：放在操作数的前面或后面。表 3-8 简要介绍一下简单表达式的结果，然后讨论它们。

表 3-8

运 算 符	类 别	范例表达式	结 果
++	一元	var1 = ++var2;	var1 的值是 var2 + 1，var2 递增 1
--	一元	var1 = --var2;	var1 的值是 var2 - 1，var2 递减 1
++	一元	var1 = var2++;	var1 的值是 var2，var2 递增 1
--	一元	var1 = var2--;	var1 的值是 var2，var2 递减 1

这里的关键因素是这些运算符总是改变存储在操作数中的值。++总是使操作数加 1，--

总是使操作数减 1。Var1 中存储的结果有区别，其原因是运算符的位置决定了它什么时候发挥作用。把运算符放在操作数的前面，则操作数是在进行任何其他计算前受到运算符的影响，而把运算符放在操作数的后面，则操作数是在完成表达式的计算后受到运算符的影响，

这有益于另一个范例，考虑下面的代码：

```
int var1, var2 = 5, var3 = 6;
var1 = var2++ * --var3;
```

问题是，要把什么值赋予 var1？在表达式计算前，var3 前面的运算符 - -会起作用，把它的值从 6 改为 5。可以忽略 var2 后面的++运算符，因为它是在计算完成后才发挥作用，所以 var1 的结果是 5 与 5 的乘积，即 25。

在许多情况下，这些简单的一元运算符使用起来非常方便，它们实际上是下述表达式的简写形式：

```
var1 = var1 + 1;
```

这类表达式有许多用途，特别适合于在循环中使用，这将在第 4 章讲述。

下面介绍一个范例，说明如何使用数学运算符，并介绍另外两个有用的概念。

试试看：用数学运算符处理变量

(1) 在目录 C:\BegCSharp\Chapter3 下创建一个新控制台应用程序 Ch03Ex02。

(2) 在 Class1.cs 中添加如下代码：

```
static void Main(string[] args)
{
    double firstNumber, secondNumber;
    string userName;
    Console.WriteLine("Enter your name:");
    userName = Console.ReadLine();
    Console.WriteLine("Welcome {0}!", userName);
    Console.WriteLine("Now give me a number:");
    firstNumber = Convert.ToDouble(Console.ReadLine());
    Console.WriteLine("Now give me another number:");
    secondNumber = Convert.ToDouble(Console.ReadLine());
    Console.WriteLine("The sum of {0} and {1} is {2}.", firstNumber,
                      secondNumber, firstNumber + secondNumber);
    Console.WriteLine("The result of subtracting {0} from {1} is {2}.",
                      secondNumber, firstNumber, firstNumber - secondNumber);
    Console.WriteLine("The product of {0} and {1} is {2}.", firstNumber,
                      secondNumber, firstNumber * secondNumber);
    Console.WriteLine("The result of dividing {0} by {1} is {2}.",
                      firstNumber, secondNumber, firstNumber / secondNumber);
    Console.WriteLine("The remainder after dividing {0} by {1} is {2}.",
                      firstNumber, secondNumber, firstNumber % secondNumber);
```

```
}
```

(3) 执行代码，如图 3-2 所示。

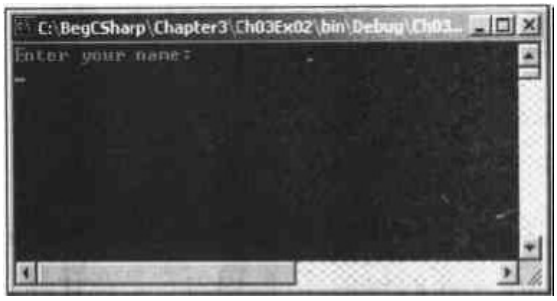


图　3-2

(4) 输入名称，按下回车键，如图 3-3 所示。

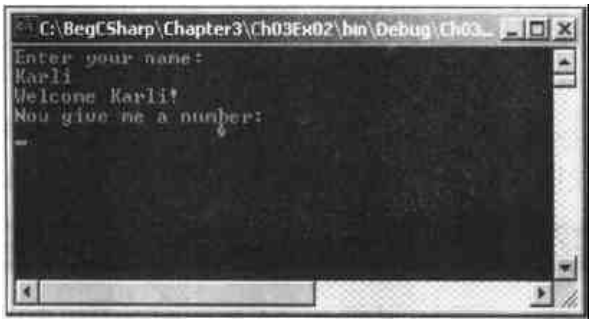


图　3-3

(5) 输入一个数字，按下回车键，再输入另一个数字，按下回车键，如图 3-4 所示。



图　3-4

代码的说明

除了说明数学运算符外，这段代码还引入了两个重要的概念，在以后的范例中将多次用到这些概念。

- 用户输入
- 类型转换

用户输入使用与前面 Console.WriteLine() 命令类似的语法。但这里使用 Console.ReadLine()。这个命令提示用户输入信息，并把它们存储在 string 变量中。

```
string userName;
```

```
Console.WriteLine("Enter your name:");
userName = Console.ReadLine();
Console.WriteLine("Welcome {0}!", userName);
```

这段代码把已赋值变量 userName 的内容写到屏幕上。

这个范例还读取了两个数字，下面略微展开讨论一下。因为 Console.ReadLine()命令生成一个字符串，而我们希望得到一个数字，这就引入了类型转换的问题。第 5 章将详细讨论类型转换，下面先看看本例使用的代码。

首先，声明要存储数字的变量：

```
double firstNumber, secondNumber;
```

接着，给出提示，对 Console.ReadLine()得到的字符串使用命令 Convert.ToDouble()，把字符串转换为 double 类型，把这个数值赋给前面声明的变量 firstNumber：

```
Console.WriteLine("Now give me a number:");
firstNumber = Convert.ToDouble(Console.ReadLine());
```

这个语法是相当简单的，其他的许多转换也用这种方式进行。

其余的代码以相同的方式获取第二个数：

```
Console.WriteLine("Now give me another number:");
secondNumber = Convert.ToDouble(Console.ReadLine());
```

然后输出两个数字的加、减、乘、除的结果，并使用余数运算符(%)显示除操作的余数。

```
Console.WriteLine("The sum of {0} and {1} is {2}.", firstNumber,
                  secondNumber, firstNumber + secondNumber);
Console.WriteLine("The result of subtracting {0} from {1} is {2}.",
                  secondNumber, firstNumber, firstNumber - secondNumber);
Console.WriteLine("The product of {0} and {1} is {2}.", firstNumber,
                  secondNumber, firstNumber * secondNumber);
Console.WriteLine("The result of dividing {0} by {1} is {2}.",
                  firstNumber, secondNumber, firstNumber / secondNumber);
Console.WriteLine("The remainder after dividing {0} by {1} is {2}.",
                  firstNumber, secondNumber, firstNumber % secondNumber);
```

注意我们提供了表达式 firstNumber + secondNumber 等，作为 Console.WriteLine()语句的一个参数，而没有使用中间变量：

```
Console.WriteLine("The sum of {0} and {1} is {2}.", firstNumber,
                  secondNumber, firstNumber + secondNumber);
```

这种语法可以使代码的可读性比较好，减少需要编写的代码量。

3.3.2　赋值运算符

直到现在，我们一直在使用简单的=赋值运算符，其实还有其他赋值运算符，而且它们都

非常有用。

除了＝运算符外，其他赋值运算符都以类似的方式工作。与＝一样，它们都是根据运算符和右边的操作数，把一个值赋给左边的变量。

与以前一样，用表格的方式列出这些运算符及其说明，如表 3-9 所示。

表 3-9

运 算 符	类 别	范例表达式	结 果
=	二元	var1 = var2;	var1 被赋予 var2 的值
+=	二元	var1 += var2;	var1 被赋予 var1 与 var2 的和
-=	二元	var1 -= var2;	var1 被赋予 var1 与 var2 的差
*=	二元	var1 *= var2;	var1 被赋予 var1 与 var2 的乘积
/=	二元	var1 /= var2;	var1 被赋予 var1 与 var2 相除所得的结果
%=	二元	var1 %= var2;	var1 被赋予 var1 与 var2 相除所得的余数

可以看出，这些运算符把 var1 也包括在计算过程中，下面的代码：

```
var1 += var2;
```

与下面的代码结果相同。

```
var1 = var1 + var2;
```

注意：

+=运算符也可以用于字符串，与+一样。

使用这些运算符，特别是在使用长变量名时，可以使代码更容易阅读。

3.3.3 运算符的优先级

在计算表达式时，每个运算符都会按顺序处理。但这并不意味着从左至右地运用这些运算符。

例如，有下面的代码：

```
var1 = var2 + var3;
```

其中+运算符就是在＝运算符之前进行计算的。

在其他一些情况下，运算符的优先级并没有这么明显，例如：

```
var1 = var2 + var3 * var4;
```

其中*运算符先计算，其后是+运算符，最后是=运算符，这是标准的数学顺序，其结果与我们在纸上进行算术运算的结果相同。

像这样的计算，可以使用括号控制运算符的优先级，例如：

```
var1 = (var2 + var3) * var4;
```

括号中的内容先计算，即+运算符在*运算符之前计算。

对于前面介绍的运算符，其优先级如表 3-10 所示，优先级相同的运算符(如*和/)按照从左至右的顺序计算。

表 3-10

优　先　级	运　算　符
优先级由高到低	++, --(用作前缀); +, - (一元)
	*, /, %
	+, -
	=, *=, /=, %=, +=, - =
	++,--(用作后缀)

注意，

括号可用于忽略优先级顺序，如上所述。

3.3.4　名称空间

在继续学习前，应花一定的时间了解一个比较重要的主题——名称空间。它们是.NET 中提供应用程序代码容器的方式，这样，代码及其内容就可以惟一地标识了。名称空间也用作.NET Framework 中给项目分类的一种方式。大多数项目都是类型定义，例如本章描述的简单类型(System.Int32 等)。

在默认情况下，C#代码包含在全局名称空间中。这意味着对于包含在这段代码中的项目，只要按照名称进行引用，就可以由全局名称空间中的其他代码访问它们。可以使用 namespace 关键字为花括号中的代码块显式定义名称空间。如果在该名称空间代码的外部使用名称空间中的名称，就必须写出该名称空间中的限定名称。

限定名称包括它所有的继承信息。基本上，这意味着，如果一个名称空间中的代码需要使用在另一个名称空间中定义的名称，就必须包括对该名称空间的引用。限定名称在不同的名称空间级别之间使用点字符(.)。

例如：

```
namespace LevelOne
{
    // code in LevelOne namespace

    // name "NameOne" defined
}

// code in global namespace
```

这段代码定义了一个名称空间 LevelOne，以及该名称空间中的一个名称 NameOne(注意这

里没有列出其他代码，是为了使我们的讨论一般化，并在定义名称空间的地方添加了一个注释)。在名称空间 LevelOne 中编写的代码可以简单地使用"NameOne"来引用该名称，不需要任何分类信息。但全局名称空间中的代码必须使用分类名称"LevelOne.NameOne"来引用这个名称。

在名称空间中，使用关键字 namespace 还可以定义嵌套的名称空间。嵌套的名称空间通过其层次结构来引用，并使用句点区分层次结构的层次。这最好用一个范例来说明。考虑下面的名称空间：

```
namespace LevelOne
{
    // code in LevelOne namespace

    namespace LevelTwo
    {
        // code in LevelOne.LevelTwo namespace

        // name "NameTwo" defined
    }
}

// code in global namespace
```

其中，在全局名称空间中，NameTwo 必须引用为"LevelOne.LevelTwo.NameTwo"，在 LevelOne 名称空间中，则可以引用为"LevelTwo.NameTwo"，在 LevelOne.LevelTwo 名称空间中，则可以引用为 NameTwo。

要注意的是，名称是由名称空间是惟一定义的。可以在 LevelOne 和 LevelTwo 名称空间中定义名称"NameThree"：

```
namespace LevelOne
{
    // name "NameThree" defined

    namespace LevelTwo
    {
        // name "NameThree" defined
    }
}
```

这定义了两个不同的名称 LevelOne.NameThree 和 LevelOne.LevelTwo.NameThree，它们可以独立使用，互不干扰。

建立了名称空间后，就可以使用 using 语句简化对它们包含的名称的访问。实际上，using 语句的意思是“OK，我们需要这个名称空间中的名称，所以不要每次总是要求对它们分类”。例如，在下面的代码中，LevelOne 名称空间中的代码可以访问 LevelOne.LevelTwo 名称空间中的名称，而无需分类：

```
namespace LevelOne
{
    using LevelTwo;

    namespace LevelTwo
    {
        // name "NameTwo" defined
    }
}
```

LevelOne 名称空间中的代码现在可以简单地使用"NameTwo"引用 LevelTwo.NameTwo。

有时，与上面的 NameThree 范例一样，不同名称空间中的相同名称会产生错误，使系统崩溃(此时，代码是不能编译的)。此时，可以为名称空间提供一个别名，作为 using 语句的一部分。

```
namespace LevelOne
{
    using LT = LevelTwo;

    // name "NameThree" defined

    namespace LevelTwo
    {
        // name "NameThree" defined
    }
}
```

LevelOne 名称空间中的代码可以把 LevelOne.NameThree 引用为"NameThree"，把 LevelOne.LevelTwo.NameThree 引用为"LT.NameThree"。

Using 语句可以应用到包含它们的名称空间，以及该名称空间中包含的嵌套名称空间中。在上面的代码中，全局名称空间不能使用"LT.NameThree"。但如果 Using 语句声明如下：

```
using LT = LevelOne.LevelTwo;

namespace LevelOne
{
    // name "NameThree" defined

    namespace LevelTwo
    {
        // name "NameThree" defined
    }
}
```

这样全局名称空间中的代码和 LevelOne 名称空间就可以使用"LT.NameThree"。

这里有一点要注意：using 语句本身不能访问另一个名称空间中的名称。除非名称空间中的

代码以某种方式链接到工程上，或者代码是在该工程的源文件中定义的，或在链接到该工程的其他代码中定义的，否则就不能访问其中包含的名称。另外，如果包含名称空间的代码链接到工程上，无论是否使用 using，都可以访问其中包含的名称。Using 语句便于我们访问这些名称，减少代码量，使之更合理。

回过头来看看本章开头的 ConsoleApplication1 中的代码，下面的代码应用到名称空间上：

```
using System;

namespace ConsoleApplication1
{
    ...
}
```

第一行使用 using 声明：在这段 C#代码中使用 System 名称空间，而且它可以在该文件的所有名称空间中访问，无需分类。System 名称空间是.NET Framework 应用程序的根名称空间，包含控制台应用程序所需要的所有基本功能。

下一个名称空间 ConsoleApplication1 是为应用程序代码本身声明的。

3.4 小结

本章介绍了创建有效 C#应用程序的许多基础知识，讲述了 C#的基本语法，分析了在创建控制台应用程序工程时，VS 生成的最基础的控制台应用程序代码。

本章的主要内容是变量的使用。我们描述了变量，阐述了如何创建变量，如何给它们赋值，如何处理它们以及它们包含的值。同时，介绍了一些基本的用户交互，描述了如何把文本输出到控制台应用程序上，如何读取用户的输入。这涉及到一些非常基本的类型转换。类型转换是一个复杂的问题，将在第 5 章详细论述。

本章还介绍了如何把运算符和操作数组合为表达式，并说明了这些运算符的执行方式，以及执行它们的顺序。

最后介绍了名称空间。随着本书的介绍，名称空间会显得越来越重要。这里仅以比较抽象的方式介绍了这个主题，完整的论述见后面的内容。

到目前为止，所有的编程工作都是逐行完成的。第 4 章将学习如何使用循环技术和条件分支控制程序执行的流程，使代码的效率更高。

3.5 练习

(1) 在下面的代码中，如何引用名称空间 fabulous 中的名称 great?

```
namespace fabulous
{
    // code in fabulous namespace
```

```
}

namespace super
{
    namespace smashing
    {
        // great name defined
    }
}
```

(2) 下面哪些不是合法的变量名?

a　myVariableIsGood
b　99Flake
c　_floor
d　time2GetJiggyWidIt
e　wrox.com

(3) 字符串"supercalifragilisticexpialidocious"是因为太长了而不能放在 string 变量中吗? 为什么?

(4) 考虑运算符的优先级，列出下述表达式的计算步骤。

```
resultVar += var1 * var2 + var3 % var4 / var5;
```

(5) 编写一个控制台应用程序，要求用户输入 4 个 int 值，并显示它们的乘积。提示：可以考虑使用 Convert.ToDouble()命令，它可以把用户在控制台上输入的数转换为 Double；从 string 转换为 int 的命令是 Convert.ToInt32()。

第4章 流程控制

到目前为止，我们看到的C#代码有一个共同点：程序的执行都是一行接一行、自上而下地进行，不遗漏任何代码。如果所有的应用程序都这样执行，则我们能做的工作就很有限了。

本章介绍控制程序流程的两种方法。程序流程就是C#代码行的执行顺序。这两种方法是：

- 分支——有条件地执行代码。条件取决于计算的结果，例如“如果myVal小于0，就只执行这行代码”。
- 循环——重复执行相同的语句(重复执行一定的次数，或者在满足测试条件后停止执行)。

这两种方法都要用到布尔逻辑。第3章介绍了bool类型，但没有讨论它。本章将在很多地方使用它，所以先讨论布尔逻辑，以便在流程控制环境下使用它。

4.1 布尔逻辑

第3章介绍的bool类型可以有两个值：true或false。这种类型常常用于记录某些操作的结果，以便操作这些结果。特别是bool类型可用于存储比较的结果。

注意：

从历史上看，应记住(并尊重)英国数学家乔治·布尔，他在19世纪中叶为布尔逻辑奠定了基础。

例如，考虑下述情形(本章介绍)：要根据变量myVal是否小于10，来确定是否执行代码。为此，需要确定语句“myVal小于10”的真假，即需要知道比较的布尔结果。

布尔比较需要使用布尔比较运算符(也称为关系运算符)，如表4-1所示。这里var1都是bool类型的变量，var2和var3则可以是各种类型。

表 4-1

运算符	类别	范例表达式	结果
==	二元	var1 = var2 == var3;	如果var2等于var3，var1的值就是true，否则为false
!=	二元	var1 = var2 != var3;	如果var2不等于var3，var1的值就是true，否则为false
<	二元	var1 = var2 < var3;	如果var2小于var3，var1的值就是true，否则为false
>	二元	var1 = var2 > var3;	如果var2大于var3，var1的值就是true，否则为false

(续表)

运 算 符	类 别	范例表达式	结 果
<=	二元	var1 = var2 <= var3;	如果 var2 小于或等于 var3，var1 的值就是 true，否则为 false
>=	二元	var1 = var2 >= var3;	如果 var2 大于或等于 var3，var1 的值就是 true，否则为 false

在代码中，可以对数值使用这些运算符，如下所示：

```
bool isLessThan10;
isLessThan10 = myVal < 10;
```

如果 myVal 存储的值大于 10，这段代码就给 isLessThan10 赋予 true 值，否则赋予 flase 值。
也可以对其他类型使用这些比较运算符，例如字符串：

```
bool isKarli;
isKarli = myString == "Karli";
```

如果 myString 存储的字符串是"Karli"，isKarli 的值就为 true。
也可以对布尔值使用这些运算符：

```
bool isTrue;
isTrue = myBool == true;
```

但只能使用== 和 !=运算符。

注意：
假定 val1 < val2 是 flase，则 val1 > val2 为 true，此时会产生一个错误。因为如果 val1 == val2，则这两个语句都是 false。

在处理布尔值时，还有其他一些布尔运算符，如表 4-2 所示。

表 4-2

运 算 符	类 别	范例表达式	结 果
!	一元	var1 = ! var2;	如果 var2 是 false，var1 的值就是 true，否则为 false(逻辑非)
&	二元	var1 = var2 & var3;	如果 var2 和 var3 都是 true，var1 的值就是 true，否则为 false(逻辑与)
|	二元	var1 = var2 | var3;	如果 var2 或 var3 是 true(或两者都是)，var1 的值就是 true，否则为 false(逻辑或)
^	二元	var1 = var2 ^ var3;	如果 var2 或 var3 中有且仅有一个是 true，var1 的值就是 true，否则为 false (逻辑异或)

上面的代码也可以表述为：

```
bool isTrue;
isTrue = myBool & true;
```

&和|运算符也有两个类似的运算符(见表 4-3)。

表 4-3

运 算 符	类　别	范例表达式	结　果
&&	二元	var1 = var2 && var3;	如果 var2 和 var3 都是 true，var1 的值就是 true，否则为 false (逻辑与)
||	二元	var1 = var2 || var3;	如果 var2 或 var3 是 true(或两者都是)，var1 的值就是 true，否则为 false (逻辑或)

这些运算符的结果与&和|完全相同，但得到结果的方式有一个重要区别：其性能比较好。两者都是检查第一个操作数的值(表 4-3 表中的 var2)，再根据该操作数的值进行操作，可能根本就不需要第二个操作数的值(表 4-3 表中的 var3)。

如果&&运算符的第一个操作数是 false，就不需要考虑第二个操作数的值了，因为无论第二个操作数的值是什么，其结果都是 false。同样，如果第一个操作数是 true，||运算符就返回 true，无需考虑第二个操作数的值。

但上面的&和|运算符却不是这样。它们的操作数总是要计算的。

因为操作数的计算是有条件的，如果使用&&和||运算符来代替&和|，性能会有一定的提高。这在大量使用这些运算符的应用程序中是比较明显的。作为一个规则，尽可能使用&&和||运算符。

4.1.1 位运算符

在上一节的讨论中，读者可能会问，为什么会有&和|运算符。原因是这两个运算符可以用于对数值执行操作。实际上，它们处理的是存储在变量中的一系列位，而不是变量的值。

下面先讨论&。第一个操作数中的每个位都与第二个操作数中相同位置上的位进行比较，在得到的结果中，各个位置上的位如表 4-4 所示。

表 4-4

操作数 1 的位	操作数 2 的位	&的结果位
1	1	1
1	0	0
0	1	0
0	0	0

| 运算符与此类似，但得到的结果是不同的，如表 4-5 所示。

表 4-5

操作数 1 的位	操作数 2 的位	&的结果位
1	1	1
1	0	1
0	1	1
0	0	0

例如，考虑下面代码中的操作：

```
int result, op1, op2;
op1 = 4;
op2 = 5;
result = op1 & op2;
```

这里必须考虑 op1 和 op2 的二进制表示方式，它们分别是 100 和 101。比较这两个表达方式中相同位置上的二进制数字，得出结果，如下所示：

- 如果 op1 和 op2 最左边的位都是 1，result 最左边的位就是 1，否则为 0。
- 如果 op1 和 op2 次左边的位都是 1，result 次左边的位就是 1，否则为 0。
- 继续比较其他的位。

在这个范例中，op1 和 op2 最左边的位都是 1，所以 result 最左边的位就是 1。下一个位都是 0，第 3 个位置上的位分别是 1 和 0，则 result 第 2~3 个位都是 0。最后，结果的二进制值是 100，即结果是 4。

以下是这个过程：

```
    1   0   0     4
&   1   0   1     5
-------------------
    1   0   0     4
```

如果使用|运算符，将进行相同的过程，但如果操作数中相同位置上的位有一个是 1，其结果位就是 1。

```
    1   0   0     4
|   1   0   1     5
-------------------
    1   0   1     5
```

^运算符的用法与此相同。如果操作数中相同位置上的位有且仅有一个是 1，其结果位就是 1，(如表 4-6 所示)。

表 4-6

操作数 1 的位	操作数 2 的位	^的结果位
1	1	0
1	0	1
0	1	1
0	0	0

C#中还可以使用一元位运算符~，它将操作数中的位取反，其结果应是操作数中位为 1 的，在结果中就是 0，反之亦然，如表 4-7 所示。

表 4-7

操作数的位	~的结果位
1	0
0	1

这些位运算符在某些情况下是非常有用的，因为它们可以用变量中的各个位存储信息。例如，颜色可以使用 3 个位来指定红、绿、蓝。我们可以分别设置这些位，改变这 3 个位，进行如下配置，如表 4-8 所示。

表 4-8

位	十 进 制 数	含 义
000	0	黑色
100	4	红色
010	2	绿色
001	1	蓝色
101	5	洋红色
110	6	黄色
011	3	青色
111	7	白色

假定把这些值存储在一个类型为 int 的变量中。首先从黑色开始，即值为 0 的 int 变量，可以执行如下操作：

```
int myColor = 0;
bool containsRed;
myColor = myColor | 2;              // Add green bit, myColor now stores 010
myColor = myColor | 4;              // Add red bit, myColor now stores 110
containsRed = (myColor & 4) == 4;   // Check value of red bit
```

最后一行代码把值 true 赋予 containsRed，因为 myColor 的“红色位”是 1。

这种技术在高效使用信息时非常有效，特别适合于同时检查多个位的值(对于 int 值，是 32 位)。但是，在一个变量中存储额外信息有更好的方式，即利用第 5 章讨论的高级变量类型。

除了这 4 个位运算符外，本节还要介绍另外两个运算符，如表 4-9 所示。

表 4-9

运 算 符	类 别	范例表达式	结 果
>>	二元	var1 = var2 >> var3;	把 var2 的二进制值向右移动 var3 位，就得到 var1 的值
<<	二元	var1 = var2 << var3;	把 var2 的二进制值向左移动 var3 位，就得到 var1 的值

这些运算符通常称为位移运算符，最好用一个范例来说明：

```
int var1, var2 = 10, var3 = 2;
var1 = var2 << var3;
```

结果，var1 的值是 40。具体过程如下：10 的二进制值是 1010，把该数值向左移动两位，得到 101000，即十进制中的 40。实际上，我们执行了多个操作。每向左移动一位，该数都要乘以 2，所以向左移动两位，就是给原来的操作数乘以 4。而每向右移动一位，则是给操作数除以 2，并丢弃余数：

```
int var1, var2 = 10;
var1 = var2 >> 1;
```

在这个范例中，var1 的值是 5，而下面的代码得到的值是 2：

```
int var1, var2 = 10;
var1 = var2 >> 2;
```

在大多数代码中，都不使用这些运算符，但应知道有这样的运算符存在。它们主要用于高度优化的代码，在这些代码中，不能使用其他数学操作。因此它们通常用于设备驱动程序或系统代码。

4.1.2 布尔赋值运算符

本节要介绍的最后一类运算符是把前面的赋值运算符组合起来，非常类似于第 3 章中的数学赋值运算符(+=, *=等)。如表 4-10 所示。

表 4-10

运 算 符	类 别	范例表达式	结 果
&=	二元	var1 &= var2;	var1 的值是 var1 & var2 的结果
|=	二元	var1 |= var2;	var1 的值是 var1 | var2 的结果
^=	二元	var1 ^= var2;	var1 的值是 var1 ^ var2 的结果

这些运算符处理布尔值和数值的方式与&, |和 ^相同。

注意：

&=和 |= 使用 & 和 |，而不是 &&和 ||，与这些较简单的运算符相关联。

位移运算符也有赋值运算符，如表 4-11 所示。

表 4-11

运 算 符	类 别	范例表达式	结 果
>>=	二元	var1 >>= var2;	把 var1 的二进制值向右移动 var2 位，就得到 var1 的值
<<=	二元	var1 <<= var2;	把 var1 的二进制值向左移动 var2 位，就得到 var1 的值

试试看：使用布尔和位运算符

(1) 在目录 C:\BegCSharp\Chapter4 下创建一个新控制台应用程序 Ch04Ex01。

(2) 把下述代码添加到 Class1.cs 中：

```
static void Main(string[] args)
{
    Console.WriteLine("Enter an integer:");
    int myInt = Convert.ToInt32 (Console.ReadLine());
    Console.WriteLine("Integer less than 10? {0}", myInt < 10);
    Console.WriteLine("Integer between 0 and 5? {0}",
                        (0 <= myInt) && (myInt <= 5));
    Console.WriteLine("Bitwise AND of Integer and 10 = {0}", myInt & 10);
}
```

(3) 运行应用程序，出现提示时，输入一个整数，如图 4-1 所示。

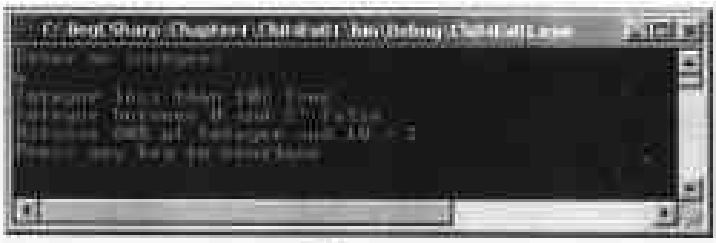


图 4-1

代码的说明

前两行代码使用前面介绍的技术，提示并接受一个整数值：

```
Console.WriteLine("Enter an integer:");
int myInt = Convert.ToInt32(Console.ReadLine());
```

我们使用 Convert.ToInt32()从字符串输入中得到一个整数。Convert.ToInt32()是另一个类型转换命令，与前面使用的 Convert.ToDouble()命令位于同一个系列中。

剩下的 3 行代码对得到的数字进行各种操作，并显示结果。在编写这段代码时，假定用户输入了 6，如图 4-1 所示。

第一个输出是操作 myInt < 10 的结果。如果 myInt 是 6，则它小于 10，因此结果为 true。如果 MyInt 的值是 10 或更大，就会得到 false。

第二个输出涉及的计算比较多：(0 <= myInt) && (myInt <= 5)，其中包含两个比较操作，用于确定 myInt 是否大于或等于 0，且小于或等于 5。接着对结果进行布尔 AND 操作。输入数字 6，则(0 <= myInt)返回 true，而(myInt <= 5)返回 false，最终的结果就是(true) && (false)，即 false，如图 4-1 所示。

最后，对 myInt 的值进行按位 AND 操作。另一个操作数是 10，它的二进制值是 100。如果 myInt 是 6，其二进制值是 110，则这个操作的结果是 10，即十进制中的 2。

```
    0   1   1   0          6
&   1   0   1   0      &   10
    0   0   1   0          2
```

4.1.3 运算符的优先级更新

现在要考虑更多的运算符，所以应更新第3章中的运算符优先级表，以把它们包括在内，如表4-12所示。

表 4-12

优先级	运算符
优先级由高到低	++, - - (用作前缀); (), +, - (一元), !, ~
	*, /, %
	+, -
	<<, >>
	<, >, <=, >=
	==, !=
	&
	^
	|
	&&
	||
	=, *=, /=, %=, +=, -=, <<=, >>=, &=, ^=, |=
	++, - -(用作后缀)

这样表中增加了好几个级别，但它明确定义了下述表达式该如何计算：

```
var1 = var2 <= 4 && var2 >= 2;
```

其中&&运算符在<= 和 >=运算符之后执行。

这里要注意的是，添加括号可以使这样的表达式看起来更清晰。编译器知道用什么顺序执行运算符，但人们常常会忘记这个顺序(有时可能想改变这个顺序)。上面的表达式也可以写为：

```
var1 = (var2 <= 4) && (var2 >= 2);
```

要解决这个问题，可以明确指定计算的顺序，这就是在上一个范例中使用它的原因。

4.1.4 goto 语句

C#允许给代码行加上标签，这样就可以使用 goto 语句直接跳转到这些代码行上。该语句有其优缺点。主要的优点是：这是控制什么时候执行哪些代码的一种非常简单的方式。主要的缺点是：过多地使用这个技巧将很难读懂代码。

下面介绍如何使用这个技巧。

goto 语句的用法如下：

```
goto <labelName>;
```

标签用下述方式定义：

```
<labelName>:
```

例如，下面的代码：

```
int myInteger = 5;
goto myLabel;
myInteger += 10;
myLabel:
Console.WriteLine("myInteger = {0}", myInteger);
```

其执行过程如下：

- myInteger 声明为 int 类型，并赋予值 5。
- goto 语句中断正常的执行过程，把控制转到标为 myLabel:的代码行上。
- myInteger 的值写到控制台上。

下面的第 3 行代码行没有执行。

```
int myInteger = 5;
goto myLabel;
myInteger += 10;
myLabel:
Console.WriteLine("myInteger = {0}", myInteger);
```

实际上，如果在应用程序中加入这段代码，就会发现编译代码时，任务列表会显示一个警告，即“Unreachable code detected”和一个行号。

Goto 语句有它们的用途，但也可能使代码陷入混乱之中。

例如，因使用 goto 语句而非常难懂的代码如下所示：

```
start:
int myInteger = 5;
goto addVal;
writeResult:
Console.WriteLine("myInteger = {0}", myInteger);
goto start;
addVal:
myInteger += 10;
goto writeResult;
```

这是有效的代码，但非常难读懂，读者可以自己试试，看看会发生什么。在此之前，应尝试理解这些代码会完成什么任务，如果理解正确，就可以给自己一点奖励。

后面再讨论这个语句，因为本章的其他一些结构将使用该语句(但最好不要使用它)。

4.2 分支

分支是控制下一步要执行哪些代码的过程。要跳转到的代码行由某个条件语句来控制。这个条件语句使用布尔逻辑，对测试值和一个或多个可能的值进行比较。

本节介绍 C#中的 3 种分支技术：

- 三元运算符
- if 语句
- switch 语句

4.2.1 三元运算符

进行比较最简单的方式是使用第 3 章介绍的三元(或条件)运算符。一元运算符有一个操作数，二元运算符有两个操作数，所以三元运算符有 3 个操作数。其语法如下：

```
<test> ? <resultIfTrue> : <resultIfFalse>
```

其中，计算<test> 可得到一个布尔值，运算符的结果根据这个值来确定是<resultIfTrue>，还是<resultIfFalse>。

三元运算符的范例如下所示：

```
string resultString = (myInteger < 10) ? "Less than 10"
                                       : "Greater than or equal to 10";
```

三元运算符的结果是两个字符串中的一个，这两个字符串都可能被赋给 resultString。把哪个字符串赋给 resultString，取决于 myInteger 的值与 10 的比较。如果 myInteger 的值小于 10，就把第一个字符串赋给 resultString；如果 myInteger 的值大于或等于 10，就把第二个字符串赋给 resultString。例如，如果 myInteger 的值是 4，则 resultString 的值就是字符串“Less than 10”。

这个运算符比较适合于这样的简单赋值语句，但不适合于根据比较结果执行大量代码的情况。此时应使用 if 语句。

4.2.2 if 语句

if 语句的功能比较多，是进行决策的有效方式。与?:语句不同的是，if 语句没有结果(所以不在赋值语句中使用它)，使用该语句是为了有条件地执行其他语句。

if 语句最简单的语法如下：

```
if (<test>)
   <code executed if <test> is true>;
```

先执行<test>(其计算结果必须是一个布尔值，这样才能编译代码)，如果<test>的计算结果是 true，就执行该语句下面的代码。在这段代码执行完毕后，或者因为<test>的计算结果是

false，而没有执行这段代码，将继续执行后面的代码行。

也可以将 else 语句和 if 语句合并使用，指定其他的代码。如果<test>的计算结果是 false，就执行 else 语句：

```
if (<test>)
    <code executed if <test> is true>;
else
    <code executed if <test> is false>;
```

这两段代码可以使用成对的花括号，放在多个代码行上：

```
if (<test>)
{
    <code executed if <test> is true>;
}
else
{
    <code executed if <test> is false>;
}
```

例如，重新编写上一节使用三元运算符的代码：

```
string resultString = (myInteger < 10) ? "Less than 10" : "Greater than 10";
```

因为 if 语句的结果不能赋予一个变量，所以要单独把值赋给变量：

```
string resultString;
if (myInteger < 10)
    resultString = "Less than 10";
else
    resultString = "Greater than or equal to 10";
```

这样的代码尽管比较冗长，但与三元运算符相比，很容易阅读和理解，其灵活性也比较大。下面看一个范例。

试试看：使用 if 语句

(1) 在目录 C:\BegCSharp\Chapter4 下创建一个新控制台应用程序 Ch04Ex02：

(2) 把下列代码添加到 Class1.cs 中：

```
static void Main(string[] args)
{
    string comparison;
    Console.WriteLine("Enter a number:");
    double var1 = Convert.ToDouble(Console.ReadLine());
    Console.WriteLine("Enter another number:");
    double var2 = Convert.ToDouble(Console.ReadLine());
    if (var1 < var2)
```

```
            comparison = "less than";
        else
        {
            if (var1 == var2)
                comparison = "equal to";
            else
                comparison = "greater than";
        }
        Console.WriteLine("The first number is {0} the second number.",
                          comparison);
    }
```

(3) 执行代码，在提示行下输入两个数字，如图 4-2 所示。



图 4-2

代码的说明

代码的第一部分我们已经熟悉了，它从用户输入中得到两个 double 值：

```
string comparison;
Console.WriteLine("Enter a number:");
double var1 = Convert.ToDouble(Console.ReadLine());
Console.WriteLine("Enter another number:");
double var2 = Convert.ToDouble(Console.ReadLine());
```

接着根据 var1 和 var2 的值，把一个字符串赋给 string 变量 comparison。首先看看 var1 是否小于 var2：

```
if (var1 < var2)
    comparison = "less than";
```

如果不是，则 var1 大于或等于 var2。在第一个比较操作中的 else 部分中，需要嵌套第二个比较：

```
else
{
    if (var1 == var2)
        comparison = "equal to";
```

只有在 var1 大于 var2 时，才执行第二个比较操作中的 else 部分：

```
        else
```

```
            comparison = "greater than";
        }
```

最后，把比较操作的值写到控制台上：

```
Console.WriteLine("The first number is {0} the second number.",
                  comparison);
```

这里使用的嵌套只是完成任务的一种方式，还可以编写如下代码：

```
if (var1 < var2)
    comparison = "less than";
if (var1 == var2)
    comparison = "equal to";
if (var1 > var2)
    comparison = "greater than";
```

这个方法的缺点是无论 var1 和 var2 的值是什么，都要执行 3 个比较操作。在第一个比较操作中，如果 var1 < var2 是 true，就只执行一个比较，否则就要执行两个比较操作(还执行了 var1 == var2 比较操作)，这样将使执行的代码行较少。其性能上的差异比较小，但在较重视速度的应用程序中，性能的差异就很明显了。

使用 if 语句执行更多的条件

在上面的范例中，有 3 个条件涉及到 var1 的值，包括了这个变量所有可能的值。有时我们要检查特定的值，例如 var1 是否等于 1, 2, 3 或 4 等。使用上面那样的代码会得到很多恼人的嵌套代码，例如：

```
if (var1 == 1)
{
    // do something
}
else
{
    if (var1 == 2)
    {
        // do something else
    }
    else
    {
        if (var1 == 3 || var1 == 4)
        {
            // do something else
        }
        else
        {
            // do something else
```

```
        }
    }
}
```

注意：

在编写第3个条件时，常常会错误地将代码写为 if (var1 == 3 || 4)。由于运算符有优先级，==运算符先执行，接着用||运算符处理布尔和数值操作数。这就会出现错误。

在这些情况下，就要使用略有不同的缩进模式，缩短 else 代码块(即在 else 块的后面使用一行代码，而不是一个代码块)。这样就得到 else if 语句结构。

```
if (var1 == 1)
{
    // do something
}
else if (var1 == 2)
{
    // do something else
}
else if (var1 == 3 || var1 == 4)
{
    // do something else
}
else
{
    // do something else
}
```

这些 else if 语句实际上是两个独立的语句，它们的功能与上述代码相同。但是这样代码会更易于阅读。

像这样进行多个比较的操作，就应考虑使用另一种分支结构：switch 语句。

4.2.3 switch 语句

switch 语句非常类似于 if 语句，因为它也是根据测试的值来有条件地执行代码。但是，switch 语句可以一次将测试变量与多个值进行比较，而不是仅测试一个条件。这种测试仅限于离散的值，而不是像“大于 X”这样的子句，所以它的用法有点区别，但它仍是一种强大的技术。

switch 语句的基本结构如下：

```
switch (<testVar>)
{
    case <comparisonVal1>:
        <code to execute if <testVar> == <comparisonVal1> >
        break;
    case <comparisonVal2>:
```

```
        <code to execute if <testVar> == <comparisonVal2> >
        break;
    ...
    case <comparisonValN>:
        <code to execute if <testVar> == <comparisonValN> >
        break;
    default:
        <code to execute if <testVar> != comparisonVals>
        break;
}
```

<testVar> 中的值与每个<comparisonValX> 值(在 case 语句中指定)进行比较，如果有一个匹配，就执行为该匹配提供的语句。如果没有匹配，就执行 default 部分中的代码。执行完每个部分中的代码后，还须有另一个语句 break。在执行完一个 case 块后，再执行第二个 case 语句是非法的。

注意：

在这个地方，C#与 C++是有区别的，在 C++中，可以在运行完一个 case 语句后，运行另一个 case 语句。

这里的 break 语句将中断 switch 语句的执行，而执行该结构后面的语句。

在 C#代码中，还有另一种方法可以防止程序流程从一个 case 语句转到下一个 case 语句。可以使用 return 语句，中断当前函数的运行，这远胜于中断 switch 结构的执行(详见第 6 章)。也可以使用 goto 语句(如前所述)，因为 case 语句实际上是在 C#代码中定义标签。例如：

```
switch (<testVar>)
{
    case <comparisonVal1>:
        <code to execute if <testVar> == <comparisonVal1> >
        goto case <comparisonVal2>;
    case <comparisonVal2>:
        <code to execute if <testVar> == <comparisonVal2> >
        break;
    ...
```

一个 case 语句处理完后，不能自由进入下一个 case 语句，但这个规则有一个例外。如果把多个 case 语句放在一起(堆叠它们)，其后加一行代码，实际上是一次检查多个条件。如果满足这些条件中的任何一个，就会执行代码，例如：

```
switch (<testVar>)
{
    case <comparisonVal1>:
    case <comparisonVal2>:
        <code to execute if <testVar> == <comparisonVal1> or
                            <testVar> == <comparisonVal2> >
```

```
        break;
    ...
```

注意这些条件也应用到 default 语句。没有这样的规则：这条语句必须放在比较操作列表的最后。还可以把它和 case 语句放在一起。用 break, goto 或 return 添加一个断点，可以确保在任何情况下，该结构都有一个有效的执行路径。

每个<comparisonValX>都必须是一个常量。一种方法是提供字面值，例如：

```
switch (myInteger)
{
    case 1:
        <code to execute if myInteger == 1 >
        break;
    case -1:
        <code to execute if myInteger == - 1 >
        break;
    default:
        <code to execute if myInteger != comparisons>
        break;
}
```

另一种方式是使用常量。常量与其他变量一样，但有一个重要的区别：它们包含的值是固定不变的。一旦给常量指定一个值后，该常量在代码执行的过程中，其值一直不变。在这里使用常量是很方便的，因为它们通常很容易阅读，在比较时，我们看不到要比较的值。

除了变量类型外，还可以使用关键字 const 声明常量，同时必须给它们赋值，例如：

```
const int intTwo = 2;
```

这行代码是有效的，但如果编写如下代码：

```
const int intTwo;
intTwo = 2;
```

就会产生一个编译错误。如果在最初的赋值之后，试图用任何方式改变常量的值，也会出现编译错误。

下面介绍一个使用常量的 switch 语句的范例。

试试看：使用 switch 语句

(1) 在目录 C:\BegCSharp\Chapter4 下创建一个新控制台应用程序 Ch04Ex03。

(2) 把下述代码添加到 Class1.cs 中：

```
        static void Main(string[] args)
        {
            const string myName = "karli";
            const string sexyName = "angelina";
```

```
        const string sillyName = "ploppy";
        string name;
        Console.WriteLine("What is your name?");
        name = Console.ReadLine();
        switch (name.ToLower ())
        {
            case myName:
                Console.WriteLine("You have the same name as me!");
                break;
            case sexyName:
                Console.WriteLine("My, what a sexy name you have!");
                break;
            case sillyName:
                Console.WriteLine("That's a very silly name.");
                break;
        }
        Console.WriteLine("Hello {0}!", name);
    }
```

(3) 执行代码，输入一个姓名，如图 4-3 所示。

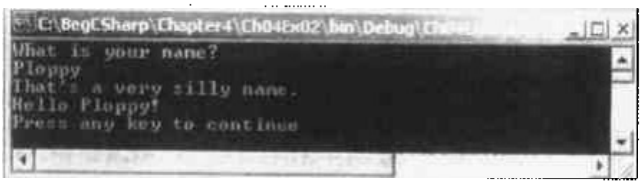


图　4-3

代码的说明

这段代码建立了 3 个常量字符串，接受用户输入的一个字符串，再根据输入的字符串把文本写到控制台上。这里字符串是用户输入的姓名。

在比较输入的姓名(在变量 name 中)和常量时，首先要用 name.ToLower()把输入的姓名转换为小写。name.ToLower()是一个标准方法，主要用于处理所有的字符串变量，在不能确定用户输入的内容时，使用它是很方便的。使用这个技术，字符串"Karli", "kArLi", "karli"等就会与测试字符串"karli"匹配了。

Switch 语句本身尝试把输入的字符串与我们定义的常量相匹配，并输出一个个性化的信息。如果成功，就会问候用户。如果不匹配，则只简单地问候用户。

Switch 语句在 case 部分的数量上没有限制，所以可以扩展这段代码，使之包含自己能想到的每个姓名，但这需要一定的时间。

4.3　循环

循环就是重复执行一些语句。这个技术使用起来非常方便，因为我们可以对操作重复任意

多次(上千次，甚至百万次)，而无需每次都编写相同的代码。

例如，下面的代码计算一个银行账户在10年后的金额，假定不计算每年的利息，且该账户没有其他款项的存取：

```
double balance = 1000;
double interestRate = 1.05; // 5% interest/year
balance *= interestRate;
balance *= interestRate;
balance *= interestRate;
balance *= interestRate;
balance *= interestRate;
balance *= interestRate;
balance *= interestRate;
balance *= interestRate;
balance *= interestRate;
balance *= interestRate;
```

相同的代码编写10次很浪费时间，如果把10年改为其他值，又会如何？我们必须把该代码行手工复制需要的次数，这是多么痛苦的事！

幸运的是，完全不必这样做。使用一个循环就可以对指令执行需要的次数。

循环的另一个重要的类型是一直循环到给定的条件满足为止。这些循环比上面描述的循环略微简单一些(但也是很有效的)，所以首先从这类循环开始。

4.3.1 do 循环

do循环以下述方式执行：执行标记为循环的代码，然后进行一个布尔测试，如果测试的结果为true，就再次执行这段代码。当测试结果为false时，就退出循环。

do循环的结构如下：

```
do
{
    <code to be looped>
} while (<Test>);
```

其中计算*<Test>*会得到一个布尔值。

提示：

while语句后面的分号是必须有的，如果没有它，会产生一个错误。

例如，使用该结构可以把从1到10的数字输出到一列上：

```
int i = 1;
do
{
    Console.WriteLine("{0}", i++);
} while (i <= 10);
```

这里在把 i 的值写到屏幕上后，使用后缀型式的++运算符递增 i 的值，所以需要检查一下 i <= 10 ，把 10 也包含在输出到控制台的数字中。

下面使用这个结构略微修改一下本节引言中的代码。在该段代码中，我们计算了一个账户在 10 年后的结余。这次我们使用一个循环，根据起始的金额和利率，计算该账户的金额要花多长时间才能达到某个指定的数值。

试试看：使用 do 循环

(1) 在目录 C:\BegCSharp\Chapter4 下创建一个新的控制台应用程序 Ch04Ex04。

(2) 把下述代码添加到 Class1.cs 中：

```
static void Main(string[] args)
{
    double balance, interestRate, targetBalance;
    Console.WriteLine("What is your current balance?");
    balance = Convert.ToDouble(Console.ReadLine());
    Console.WriteLine("What is your current annual interest rate (in %)?");
    interestRate = 1 + Convert.ToDouble(Console.ReadLine()) / 100.0;
    Console.WriteLine("What balance would you like to have?");
    targetBalance = Convert.ToDouble(Console.ReadLine());

    int totalYears = 0;
    do
    {
        balance *= interestRate;
        ++totalYears;
    }
    while (balance < targetBalance);
    Console.WriteLine("In {0} year{1} you'll have a balance of {2}.",
                      totalYears, totalYears == 1 ? "" : "s", balance);
}
```

(3) 执行代码，输入一些值，如图 4-4 所示。



图 4-4

代码的说明

这段代码利用固定的利率，对年度计算结余的过程重复必要的次数，直到满足临界条件为止。在每次循环中，递增一个计数器变量，就可以确定需要多少年：

```
int totalYears = 0;
do
{
    balance *= interestRate;
    ++totalYears;
}
while (balance < targetBalance);
```

然后就可以使用这个计数器变量作为输出结果的一部分。

```
Console.WriteLine("In {0} year{1} you'll have a balance of {2}.",
                  totalYears, totalYears == 1 ? "" : "s", balance);
```

注意：

这可能是?: (三元)运算符最常见的用法了——用最少的代码有条件地格式化文本。如果 totalYears 不等于 1，就在 year 后面输出一个 s。

但这段代码并不是完美的，考虑一下目标结余小于当前结余的情况，则结果应如图 4-6 所示。

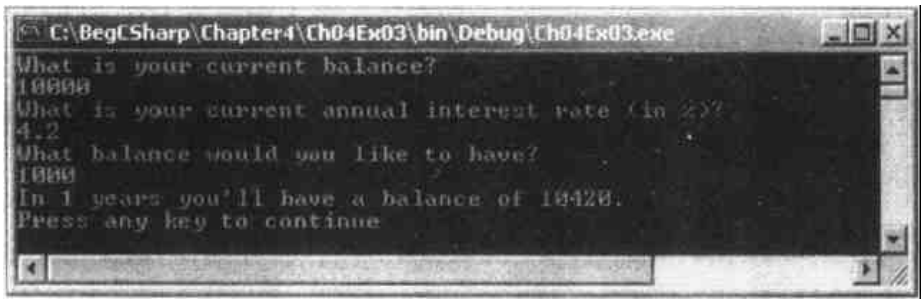


图 4-5

do 循环至少要执行一次。有时(像这种情况)这并不是很理想。当然，可以添加一个 if 语句。

```
int totalYears = 0;
if (balance < targetBalance)
{
    do
    {
        balance *= interestRate;
        ++totalYears;
    }
    while (balance < targetBalance);
}
Console.WriteLine("In {0} year{1} you'll have a balance of {2}.",
                  totalYears, totalYears == 1? "" : "s", balance);
```

但这看起来像是添加了不必要的复杂性。更好的解决方案是使用 while 循环。

4.3.2 while 循环

while 循环非常类似于 do 循环，但有一个重要的区别：While 循环中的布尔测试是在循环开始时进行，而不是最后。如果测试结果为 false，就不会执行循环。程序会直接跳转到循环后面的代码。

while 循环以下述方式指定：

```
while (<Test>)
{
    <code to be looped>
}
```

它使用的方式与 do 循环几乎完全相同，例如：

```
int i = 1;
while (i <= 10)
{
    Console.WriteLine("{0}", i++);
}
```

这段代码的执行结果与前面的 do 循环相同，它在一列中输出从 1 到 10 的数字。

下面使用 while 循环修改上一个范例。

试试看：使用 while 循环

(1) 在 VS 中，打开目录 C:\BegCSharp\Chapter4 下的控制台应用程序 Ch04Ex05。

(2) 修改代码，如下所示(开始时使用 Ch04Ex04 中的代码，记住删除原来 do 循环最后的 while 语句)：

```
static void Main(string[] args)
{
    double balance, interestRate, targetBalance;
    Console.WriteLine("What is your current balance?");
    balance = Convert.ToDouble(Console.ReadLine());
    Console.WriteLine("What is your current annual interest rate (in %)?");
    interestRate = 1 + Convert.ToDouble(Console.ReadLine()) / 100.0;
    Console.WriteLine("What balance would you like to have?");
    targetBalance = Convert.ToDouble(Console.ReadLine());

    int totalYears = 0;
    while (balance < targetBalance)
    {
        balance *= interestRate;
        ++totalYears;
    }
    Console.WriteLine("In {0} year{1} you'll have a balance of {2}.",
```

```
                totalYears, totalYears == 1 ? "" : "s", balance);
        }
```

(3) 再次执行代码，但这次使用小于起始结余的目标结余，如图 4-6 所示。

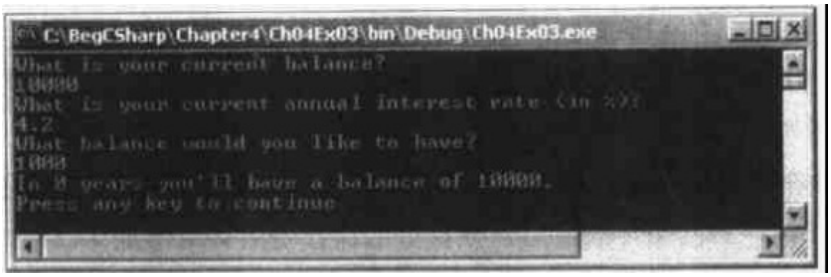


图 4-6

代码的说明

这段代码只是把 do 循环改为 while 循环，就解决了上一个范例中的问题。把布尔测试移到开头，就不需要执行循环，而是直接跳转到输出结果上。

当然，这种情况还有另一个解决方案。例如，可以检查用户输入，确保目标结余大于起始结余。此时可以把用户输入部分放在循环中，如下所示：

```
Console.WriteLine("What balance would you like to have?");
do
{
    targetBalance = Convert.ToDouble(Console.ReadLine());
    if (targetBalance <= balance)
        Console.WriteLine("You must enter an amount greater than " +
                          "your current balance!\nPlease enter another value.");
}
while (targetBalance <= balance);
```

这将拒绝接受无意义的值，得到如图 4-7 所示的结果。

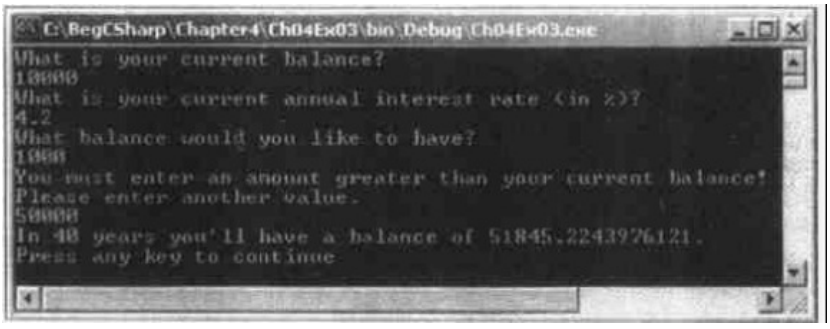


图 4-7

在进行应用程序的设计时，用户输入的有效性检查是一个很重要的问题，本书将介绍更多这方面的范例。

4.3.3 for 循环

本章介绍的最后一类循环是 for 循环。这类循环可以执行指定的次数，并维护它自己的计数器。要定义 for 循环，需要下述信息：

- 初始化计数器变量的一个起始值。
- 继续循环的条件，它应涉及到计数器变量。
- 在每次循环的最后，对计数器变量执行一个操作。

例如，如果要在循环中，使计数器从 1 递增到 10，递增量为 1，则起始值为 1，条件是计数器小于或等于 10，在每次循环的最后，要执行的操作是给计数器加 1。

这些信息必须放在 for 循环的结构中，如下所示：

```
for (<initialization>; <condition>; <operation>)
{
   <code to loop>
}
```

它的工作方式与下述 while 循环完全相同：

```
<initialization>
while (<condition>)
{
   <code to loop>
   <operation>
}
```

但 for 循环的格式使代码更易于阅读，因为其语法是在一个地方涉及循环的全部规则，而不是把几个语句放在代码的不同地方。

前面，我们使用 do 和 while 循环输出了从 1 到 10 的数字。下面看看如何使用 for 循环完成这个任务。

```
int i;
for (i = 1; i <= 10; ++i)
{
   Console.WriteLine("{0}", i);
}
```

计数器变量是一个整数 i，它的起始值是 1，在每次循环的最后递增 1。在每次循环过程中，把 i 的值写到控制台上。

注意当 i 的值为 11 时，将执行循环后面的代码。这是因为在 i 等于 10 的循环末尾，i 会递增为 11。这是在测试条件 i <= 10 之前发生的，此时循环结束。

与 while 循环一样，在第一次执行前，只在条件被测定为 true 时 for 循环才执行，所以循环中的代码可能根本就不会执行。

最后要注意的是，可以把计数器变量声明为 for 语句的一部分，重新编写上述代码，如下所示：

```
for (int i = 1; i <= 10; ++i)
{
    Console.WriteLine("{0}", i);
}
```

但如果这么做，变量 i 就不能在循环外部使用(参见第 5 章中有关变量作用域的部分)。

下面介绍一个使用 for 循环的范例。前面已经有几个使用循环的范例了，所以我把这个范例弄得有趣一些：它将显示一个 Mandelbrot 图像(使用纯文本字符，看起来不会那么吸引人！)

试试看：使用 for 循环

(1) 在目录 C:\BegCSharp\Chapter4 下创建一个新的控制台应用程序 Ch04Ex06。

(2) 把下述代码添加到 Class1.cs 中：

```
static void Main(string[] args)
{
    double realCoord, imagCoord;
    double realTemp, imagTemp, realTemp2, arg;
    int iterations;
    for (imagCoord = 1.2; imagCoord >= -1.2; imagCoord -= 0.05)
    {
        for (realCoord = -0.6; realCoord <= 1.77; realCoord += 0.03)
        {
            iterations = 0;
            realTemp = realCoord;
            imagTemp = imagCoord;
            arg = (realCoord * realCoord) + (imagCoord * imagCoord);
            while ((arg < 4) && (iterations < 40))
            {
                realTemp2 = (realTemp * realTemp) - (imagTemp * imagTemp)
                            - realCoord;
                imagTemp = (2 * realTemp * imagTemp) - imagCoord;
                realTemp = realTemp2;
                arg = (realTemp * realTemp) + (imagTemp * imagTemp);
                iterations += 1;
            }
            switch (iterations % 4)
            {
                case 0:
                    Console.Write(".");
                    break;
                case 1:
                    Console.Write("o");
                    break;
                case 2:
```

```
                    Console.Write("O");
                    break;
                case 3:
                    Console.Write("@");
                    break;
            }
        }
        Console.Write("\n");
    }
}
```

(3) 执行代码，结果如图 4-8 所示。

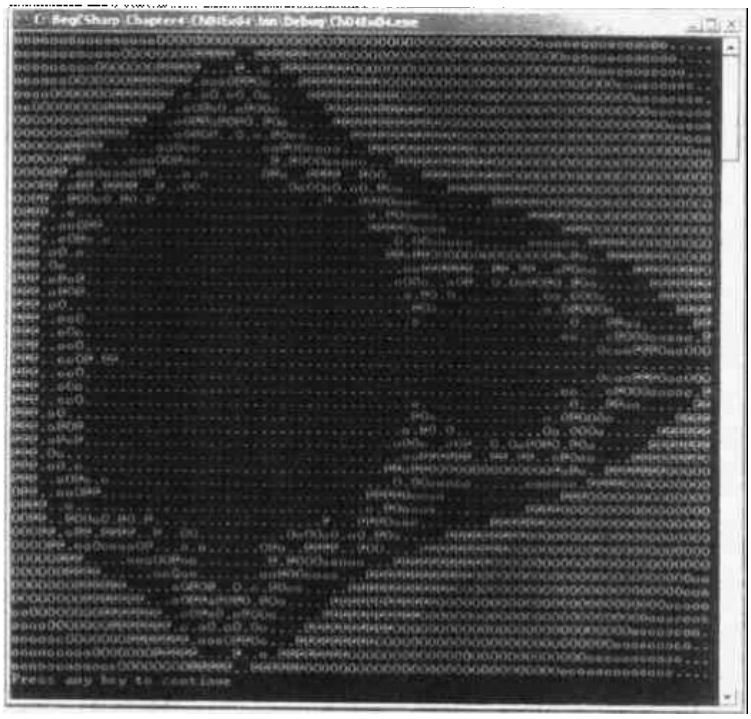


图　4-8

代码的说明

这里不打算详细说明如何计算 Mandelbrot 图像，而是解释为什么需要在这段代码中使用循环。如果你对数学不感兴趣，可以快速浏览下面两段，因为它们对代码的理解非常重要。

Mandelbrot 图像中的每个位置都对应于公式 N = x + y*i 中的一个复数。其实数部分是 x，虚数部分是 y，i 是 -1 的平方根。图像中各个位置的 x 和 y 坐标对应于虚数的 x 和 y 部分。

图像中的每个位置用参数 N 来表示，它是 x*x + y*y 的平方根。如果这个值大于或等于 2，则这个数字对应的位置值是 0。如果参数 N 的值小于 2，就把 N 的值改为 N*N - N(即 N = (x*x - y*y - x) + (2*x*y - y)*i))，并再次测试这个新 N 值。如果这个值大于或大于 2，则这个数字对应的位置值是 1。这个过程一直继续下去，直到我们给图像中的位置赋一个值，或迭代执行的次数多于指定的次数为止。

根据我们给图像中每个点赋予的值，在图形环境下，屏幕上会显示某种颜色的像素。但

是，本例使用的是文本环境，所以屏幕上显示的是一个字符。

下面看看代码，以及其中的循环。

首先声明计算过程中需要的变量：

```
double realCoord, imagCoord;
double realTemp, imagTemp, realTemp2, arg;
int iterations;
```

其中 realCoord 和 imagCoord 是 N 的实数和虚数部分，其他 double 变量是计算过程中的临时信息。Iterations 记录在参数 N(arg)等于或大于 2 之前的迭代次数。

接着是两个 for 循环，迭代图像中所有点的坐标(使用比++ 或--略复杂一些的语法来修改计数器，这是一种常见的强有力的技术)：

```
for (imagCoord = 1.2; imagCoord >= -1.2; imagCoord - = 0.05)
{
    for (realCoord = -0.6; realCoord <= 1.77; realCoord += 0.03)
    {
```

这里选择合适的边界来显示 Mandelbrot 图像的主要部分。如果要放大这个图像，可以放大这些边界。

在这两个循环中，代码处理 Mandelbrot 图像中的一个点，给 N 指定一个值，这段代码执行要求的迭代计算，给定当前点的测试值。

首先初始化一些变量：

```
iterations = 0;
realTemp = realCoord;
imagTemp = imagCoord;
arg = (realCoord * realCoord) + (imagCoord * imagCoord);
```

接着用 while 循环执行迭代。使用 while 循环，而不是 do 循环，是为了防止 N 的初始值大于 2，如果 N 大于 2，iterations = 0 就是我们需要的答案，不再需要进行计算了。

注意这里没有计算参数，而仅获取 x*x + y*y 的值，并检查该值是否小于 4。这样简化了计算，因为 2 是 4 的平方根，不需要我们自己计算平方根。

```
while ((arg < 4) && (iterations < 40))
{
    realTemp2 = (realTemp * realTemp) - (imagTemp * imagTemp)
                - realCoord;
    imagTemp = (2 * realTemp * imagTemp) - imagCoord;
    realTemp = realTemp2;
    arg = (realTemp * realTemp) + (imagTemp * imagTemp);
    iterations += 1;
}
```

这个循环计算上述的数值，其最大迭代数是 40。

把当前点的值存储在 iterations 中后，再使用 switch 语句选择要输出的字符。这里只使用 4 个不同的字符，而不是 40 个，且使用求余运算符(%)，这样 0, 4, 8 等使用一个字符，1, 5, 9 等使用另一个字符，依次类推：

```
switch (iterations % 4)
{
    case 0:
        Console.Write(".");
        break;
    case 1:
        Console.Write("o");
        break;
    case 2:
        Console.Write("O");
        break;
    case 3:
        Console.Write("@");
        break;
}
```

注意这里使用 Console.Write()，而不是 Console.WriteLine()，因为每次输出一个字符时，并不需要从一个新行开始。

在最内层的一个 for 循环结束后，需要结束一行，所以使用前面介绍的转义序列输出一行字符的结束符。

```
        }
        Console.Write("\n");
    }
```

这样，每行都与下一行分隔开来，并进行适当的排列。

这个应用程序的最终结果尽管不是很漂亮，也能给人留下深刻的印象。它说明了循环和分支的用途。

4.3.4 循环的中断

有时我们需要在循环代码的处理上有更精细的控制。C#为此提供了 4 个命令，其中的 3 个我们已经在其他情形中介绍过了：

- break——立即终止循环。
- continue——立即终止当前的循环(继续执行下一次循环)。
- goto——可以跳出循环，到已标记好的位置上(如果希望代码易于阅读和理解，最好不要使用该命令)。
- return——跳出循环及其包含的函数(参见第 6 章)。

break 命令可退出循环，继续执行循环后面的第一行代码，例如：

```
int i = 1;
```

```
while (i <= 10)
{
    if (i == 6)
        break;
    Console.WriteLine("{0}", i++);
}
```

这段代码输出 1~5 的数字，因为 break 命令在 i 的值为 5 时终止了循环。

Continue 仅终止当前的循环，而不是整个循环，例如：

```
int i;
for (i = 1; i <= 10; i++)
{
    if ((i % 2) == 0)
        continue;
    Console.WriteLine(i);
}
```

第 3 个方法使用前面的 goto，例如：

```
int i = 1;
while (i <= 10)
{
    if (i == 6)
        goto exitPoint;
    Console.WriteLine("{0}", i++);
}
Console.WriteLine("This code will never be reached.");
exitPoint:
Console.WriteLine("This code is run when the loop is exited using goto.");
```

注意使用 goto 退出循环是合法的(但会有点杂乱)，但使用 goto 从外部进入循环则是非法的。

4.3.5 无限循环

可以通过编写错误代码或错误的设计，定义永远不终止的循环，即所谓的无限循环。例如，下面的代码：

```
while (true)
{
    // code in loop
}
```

有时这种代码也是有用的，使用 break 语句总是可以退出这样的循环。

但是，当这种情形偶然出现时，就会出问题。考虑下面的循环。它与上一节的 for 循环非常类似：

```
int i = 1;
```

```
while (i <= 10)
{
    if ((i % 2) == 0)
        continue;
    Console.WriteLine("{0}", i++);
}
```

i 是在循环的最后一行代码执行完后才递增的，即在 continue 语句执行完后递增。如果执行到这个 continue 语句(此时 i 为 2)，下一个循环使用相同的 i 值，继续循环，测试这个 i 值，继续循环，一直这样下去。这就冻结了应用程序。注意仍可以用正常方式退出已冻结的应用程序，所以此时不必重新启动计算机。

4.4 小结

本章介绍了可以在代码中使用的各种结构，扩展了编程知识。在开始编写更复杂的应用程序时，这些结构的正确使用是非常重要的，本书将多次使用它们。

首先我们用一定的篇幅介绍了布尔逻辑，以及一些按位逻辑的知识。在学习了本章的其他内容后，再回过头来看看这些逻辑，可以确信，在谈到执行程序中的分支和循环代码时，这个问题是非常重要的。熟悉本节讨论的操作和技术是很有必要的。

分支结构可以有条件地执行代码，当分支与循环一起使用时，可以在 C#代码中创建出比较复杂的结构。把循环嵌套起来，再放在 if 结构中，就会发现代码的缩进是非常有用的。如果把所有的代码都放在屏幕的左端，就很难分析它们了，甚至更难调试。此时应确保代码的缩进——读者以后会赞赏它的！OK，VS 为此做了大量的工作，但最好在输入代码时进行缩进。

4.5 练习

(1) 如果两个整数存储在变量 var1 和 var2 中，该进行什么样的布尔测试，看看其中的一个(但不是两个)是否大于 10?

(2) 编写一个应用程序，其中包含练习 1 中的逻辑，让用户输入两个数字，并显示它们，但拒绝接受两个数字都大于 10 的情况，并要求用户重新输入。

(3) 下面的代码有什么错误?

```
int i;
for (i = 1; i <= 10; i++)
{
if ((i % 2) - 0)
    continue;
Console.WriteLine(i);
}
```

(4) 修改 Mandelbrot 图像应用程序，要求用户输入图像的边界，显示选中的图像部分。当前代码输出的字符应正好能放在控制台应用程序的一行上。考虑如何使每个选中的图像正好占据大小相同的空间，以最大化可视区域。

第5章　变量的更多内容

前面介绍了C#语言的一些内容，现在该回过头来讨论与变量相关的更多论题了。

首先要讨论的主题是类型的转换，即把数值从一种类型转换为另一种类型。前面已经描述了其中的一些信息，这里则要正式进行讨论。掌握这个论题可以更好地理解表达式中混合使用的类型(有意或无意)，更好地控制处理数据的方式。这有助于我们理解代码，避免不必要的误解。

介绍完了类型转换后，我们将阐述另外一些类型的变量：

- 枚举——变量类型，用户定义了一组可能的数值，这些数值可以用人类能理解的方式使用。
- 结构——合成的变量类型，由用户定义的一组其他变量类型组成。
- 数组——包含一种类型的多个变量，可以用下标访问各个数值。

这些类型比前面使用的简单类型复杂一些，但可以使工作更容易完成。

介绍完这些内容后，就要学习另一个与字符串相关的主题——基本字符串处理。

5.1　类型转换

本书的前面论述了，无论类型是什么，所有的数据都是一系列的位，即一组0和1。变量的含义是通过数据的这种解释方式来传达的。最简单的范例是char类型，这种类型用一个数字表示Unicode字符集中的一个字符。实际上，这个数字与ushort的存储方式相同——它们都存储0到65535之间的数字。

但是，一般情况下，不同类型的变量使用不同的模式来表示数据。这意味着，即使可以把一系列的位从一种类型的变量移动到另一种类型的变量中(也许它们占用的存储空间相同，也许目标类型有足够的存储空间包含所有的源数据位)，结果也可能与期望的不同。

这并不是数据位从一个变量到另一个变量的一对一映射，而是需要对数据进行类型转换。

类型转换有两种形式：

- 隐式转换 - 从类型A到类型B的转换可以在所有的情况下进行，执行转换的规则非常简单，可以让编译器进行。
- 显式转换 - 从类型A到类型B的转换只能在某些情况下进行，转换的规则比较复杂，应进行某种类型的处理。

下面逐个解释它们。

5.1.1　隐式转换

隐式转换不需要我们做任何工作，也不需要编写代码。考虑下面的代码：

```
var1 = var2;
```

如果 var2 的类型可以隐式地转换为 var1 的类型，这个赋值语句就涉及到一个隐式转换。但它只能处理相同类型的两个变量，而且不需要隐式转换。

下面看一个范例。

ushort 和 char 的值是可以互换的，因为它们都可以存储 0 到 65535 之间的数字，在这两个类型之间可以进行隐式转换，例如下面的代码：

```
ushort destinationVar;
char sourceVar = 'a';
destinationVar = sourceVar;
Console.WriteLine("sourceVar val: {0}", sourceVar);
Console.WriteLine("destinationVar val: {0}", destinationVar);
```

这里存储在 sourceVar 中的值放在 destinationVar 中。在用两个 Console.WriteLine()命令输出变量时，得到如下结果：

```
sourceVar val: a
destinationVar val: 97
```

即使两个变量存储的是相同的信息，使用不同的类型解释它们时，方式也是不同的。

简单类型有许多隐式转换：bool 和 string 没有隐式转换，但数值类型有一些隐式转换。表 5-1 列出了编译器可以隐式执行的数值转换(记住，char 存储的是数值，所以 char 被当作一个数值类型)。

表 5-1

类　型	可以安全地转换为
Byte	short, ushort, int, uint, long, ulong, float, double, decimal
Sbyte	short, int, long, float, double, decimal
Short	int, long, float, double, decimal
Ushort	int, uint, long, ulong, float, double, decimal
Int	long, float, double, decimal
Uint	long, ulong, float, double, decimal
Long	float, double, decimal
Ulong	float, double, decimal
Float	double
Char	ushort, int, uint, long, ulong, float, double, decimal

不要担心——不需要记住这个表格，因为很容易看出编译器可以执行的隐式转换。第 3 章中的一个表列出了每种简单数字类型的取值范围。这些类型的隐式转换规则是：*任何类型 A*，只要其取值范围完全包含在类型 B 的取值范围内，就可以隐式转换为类型 B。

其原因是很简单的。如果要把一个值放在变量中，而该值超出了变量的取值范围，就会出

问题。例如，short 类型的变量可以存储 0~32676 的数字，而 byte 可以存储的最大值是 255，所以如果要把一个 short 值转换为 byte 值，就会出问题。如果 short 包含的值在 256~32767 之间，该数值就不能放在 byte 中。

但是，如果 short 类型变量中的值小于 255，就应能转换这个值，对吗？

答案是可以。略复杂的答案是可以，但必须使用显式转换。执行显式转换有点类似于“OK，我知道你对我这么做提出了警告，但我将对其后果负责”。

5.1.2　显式转换

顾名思义，在明确要求编译器把数值从一种数据类型转换为另一种数据类型时，就是在执行显式转换。因此，这需要编写额外的代码，代码的格式将随着转换方法的不同而不同。在学习显式转换代码前，先看看如果不添加任何显式转换代码，会发生什么情况。

例如，下面对上一节的代码进行修改，试着把 short 值转换为 byte：

```
byte destinationVar;
short sourceVar = 7;
destinationVar = sourceVar;
Console.WriteLine("sourceVar val: {0}", sourceVar);
Console.WriteLine("destinationVar val: {0}", destinationVar);
```

如果编译这段代码，就会产生如下错误：

```
Cannot implicitly convert type 'short' to 'byte'
```

幸运的是，C#编译器可以检测出没有进行显式转换！

为了成功编译这段代码，需要添加代码，进行显式转换。最简单的方式是把 short 变量转换为 byte。数据类型转换就是强迫数据从一种类型转换为另一种类型，其语法比较简单：

```
(destinationType)sourceVar
```

这将把 sourceVar 中的值转换为 destinationType。

注意：

这只在某些情况下是可行的。彼此之间没有什么关系的类型不能进行数据类型转换。

因此可以使用这个语法修改范例，把 short 变量强迫转换为 byte：

```
byte destinationVar;
short sourceVar = 7;
destinationVar = (byte)sourceVar;
Console.WriteLine("sourceVar val: {0}", sourceVar);
Console.WriteLine("destinationVar val: {0}", destinationVar);
```

得到如下结果：

```
sourceVar val: 7
destinationVar val: 7
```

在试图把一个值转换为不合适的变量时，会发生什么呢？修改代码，如下所示：

```
byte destinationVar;
short sourceVar = 281;
destinationVar = (byte)sourceVar;
Console.WriteLine("sourceVar val: {0}", sourceVar);
Console.WriteLine("destinationVar val: {0}", destinationVar);
```

结果如下：

```
sourceVar val: 281
destinationVar val: 25
```

会发生什么？如果看看这两个数字的二进制表示，以及可以存储在 byte 中的最大值 255：

```
281 = 100011001
 25 = 000011001
255 = 011111111
```

可以看出，源数据的最左边一位丢失了。这会立即出问题：如何确定事情是何时发生的？显然，有时需要显式地把一种数据类型转换为另一种数据类型，最好知道是否有数据丢失了。如果不知道这些，就会产生严重的问题，例如计帐应用程序。

一种方式是简单地检查源变量的值，把它与目标变量的取值范围进行比较。第 6 章在介绍应用程序逻辑和程序流程控制时，将详细说明该过程。还有另一个技术，迫使系统特别注意运行期间的转换。在把一个值放在一个变量中时，如果该值过大，不能放在该类型的变量中，就会导致溢出，这就是我们要检查的情况。

这里要用到两个关键字 checked 和 unchecked，称为表达式的溢出检查内容。以下述方式使用这两个关键字：

```
checked(expression)
unchecked(expression)
```

下面对上一个范例进行溢出检查：

```
byte destinationVar;
short sourceVar = 281;
destinationVar = checked((byte)sourceVar);
Console.WriteLine("sourceVar val: {0}", sourceVar);
Console.WriteLine("destinationVar val: {0}", destinationVar);
```

在执行这段代码时，程序会崩溃，并显示如图 5-1 所示错误信息(在工程 OverflowCheck 中编译这段代码)。

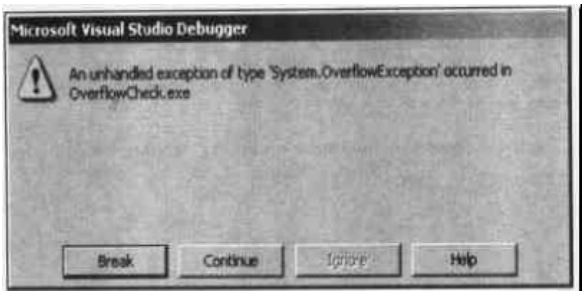


图　5-1

但是，在这段代码中，如果用 unchecked 替代 checked，就会得到与以前一样的结果，不会出现错误。这与前面的默认操作是一样的。

除了这两个关键字以外，还可以配置应用程序，让这种类型的表达式都包含 checked 关键字，除非表达式明确使用 unchecked 关键字(换言之，我们可以改变默认设置，进行溢出检查)。为此，应修改工程的属性：在 VS 中右击 Solution Explorer 窗口中的工程，选择 Properties 选项。单击窗口左边的 Configuration Properties 文件夹，打开有 3 个子项(Build, Debugging 和 Advanced)的列表。要修改的属性是一个编译器设置，所以应选择 Build 子项。在属性的顶部(Code Generation)，有一个选项 Check for Arithmetic Overflow/Underflow，如图 5-2 所示。在默认情况下，这个设置是 False，把它改为 True，就会进行如上所述的 checked 操作。

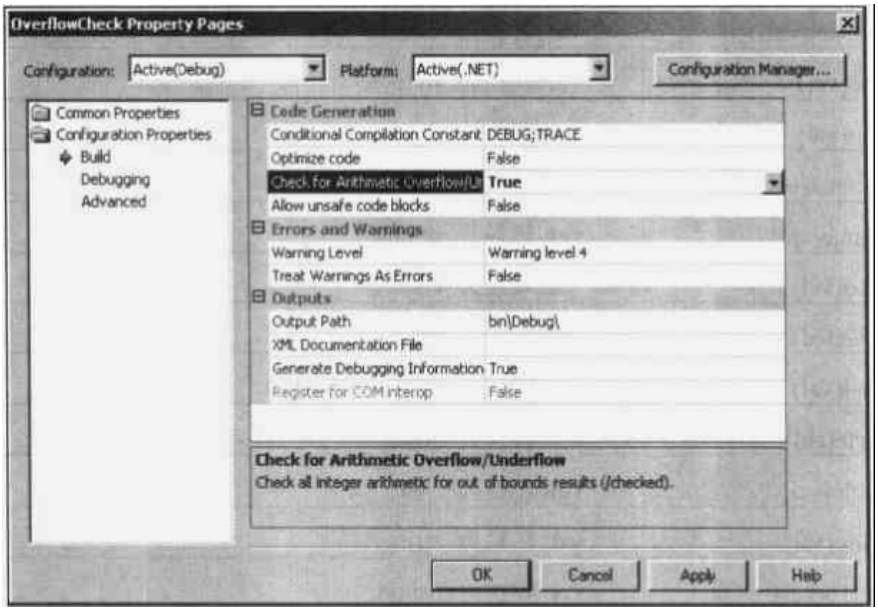


图　5-2

5.1.3　使用 Convert 命令进行显式转换

本书的许多“试试看”范例中使用的显式类型转换，与本章的范例有一些区别。前面使用 Convert.ToDouble()等命令把字符串值转换为数值，显然，这种方式并不适合于所有的字符串。

例如，如果使用 Convert.ToDouble()把字符串 Number 转换为一个 double 值，执行代码，就会得到如图 5-3 所示对话框。

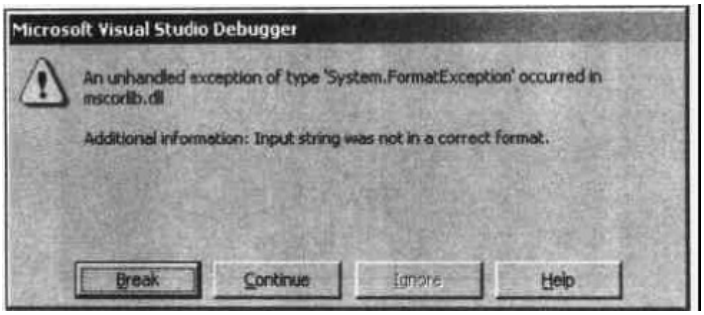


图　5-3

可以看出，执行失败。为了成功执行这种类型的转换，所提供的字符串必须是数字的有效表达方式，该数还必须是不会溢出的数。数的有效表达方式是：首先是一个可选符号(加号或减号)，然后是 0 位或多位数字，一个圆点后跟一位或多位数字，接着是一个可选的 e 或 E，句点后跟一位或多位数字(在这个序列之前或之后)和空格。利用这些可选的额外数据，就可以看出 -1.2451e-24 这样复杂的字符串是一个数值。

以这种方式可以进行许多显式转换，如表 5-2 所示。

表　5-2

命　令	结　果
Convert.ToBoolean(val)	val 转换为 bool
Convert.ToByte(val)	val 转换为 byte
Convert.ToChar(val)	val 转换为 char
Convert.ToDecimal(val)	val 转换为 decimal
Convert.ToDouble(val)	val 转换为 double
Convert.ToInt16(val)	val 转换为 short
Convert.ToInt32(val)	val 转换为 int
Convert.ToInt64(val)	val 转换为 long
Convert.ToSByte(val)	val 转换为 sbyte
Convert.ToSingle(val)	val 转换为 float
Convert.ToString(val)	val 转换为 string
Convert.ToUInt16(val)	val 转换为 ushort
Convert.ToUInt32(val)	val 转换为 uint
Convert.ToUInt64(val)	val 转换为 ulong

其中 val 可以是各种类型的变量(如果这些命令不能处理该类型的变量，编译器就会告诉用户)。

但如表 5-2 所示，转换的名称略不同于 C#类型名称，例如，要转换为 int，应使用 Convert.ToInt32()。这是因为这些命令来自于.NET Framework 的 System 名称空间，而不是 C#本身。这样它们就可以在除 C#以外的其他.NET 兼容语言中使用。

这些转换要注意的一个问题是它们总是要进行溢出检查，checked 和 unchecked 关键字以

及工程属性设置没有什么作用。

下面的范例包括本节介绍的许多转换类型。

试试看：类型转换的实践

(1) 在目录 C:\BegCSharp\Chapter5 下创建一个新控制台应用程序 Ch05Ex01。

(2) 把下述代码添加到 Class1.cs 中：

```
static void Main(string[] args)
{
    short  shortResult, shortVal = 4;
    int    integerVal = 67;
    long   longResult;
    float  floatVal = 10.5F;
    double doubleResult, doubleVal = 99.999;
    string stringResult, stringVal = "17";
    bool   boolVal = true;

    Console.WriteLine("Variable Conversion Examples\n");

    doubleResult = floatVal * shortVal;
    Console.WriteLine("Implicit, -> double: {0} * {1} -> {2}", floatVal,
        shortVal, doubleResult);

    shortResult = (short)floatVal;
    Console.WriteLine("Explicit, -> short:  {0} -> {1}", floatVal,
        shortResult);

    stringResult = Convert.ToString(boolVal) +
        Convert.ToString(doubleVal);
    Console.WriteLine("Explicit, -> string: \"{0}\" + \"{1}\" -> {2}",
        boolVal, doubleVal, stringResult);

    longResult = integerVal + Convert.ToInt64(stringVal);
    Console.WriteLine("Mixed,      -> long:   {0} + {1} -> {2}",
        integerVal, stringVal, longResult);
}
```

(3) 执行代码,如图 5-4 所示。

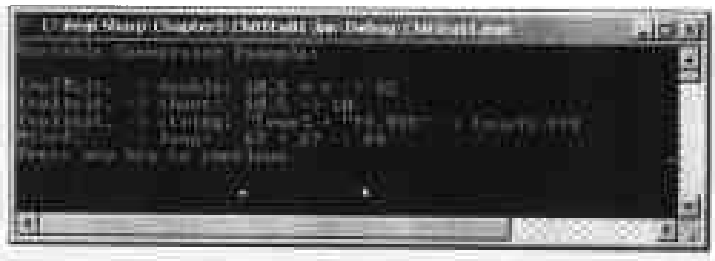


图 5-4

代码的说明

这个范例包含前面介绍的所有转换类型，既有像前面小代码范例中的简单赋值，也有在表达式中进行的转换。我们必须考虑这两种情况，因为每个非一元运算符的处理都可能要进行类型转换，而不仅仅是赋值运算符。例如：

```
shortVal * floatVal
```

其中把一个 short 值与一个 float 值相乘。在这样的指令中，没有指定显式转换，应在可能的情况下进行隐式转换。在这个范例中，惟一有意义的隐式转换是把 short 值转换为 float(因为把 float 值转换为 short 需要显式转换)，所以这是一个我们使用的隐式转换。

也可以重新编写这个过程，使用下述代码：

```
shortVal * (short)floatVal
```

注意：

这并不表示这个操作将返回一个 short。因为两个 short 相乘的结果很可能大于 32767(这是 short 可以包含的最大值)，所以这个操作的结果实际上是 int。

使用这个数据类型转换语法执行显式转换，其运算符的优先级与其他一元运算符一样，如++(用作前缀)——优先级中的最高级。

如果语句涉及到混合的转换，就根据运算符的优先级，在处理每个运算符时执行转换。这意味着可能出现“中间”转换，例如：

```
doubleResult = floatVal + (shortVal * floatVal);
```

要处理的第一个运算符是*，如上所述，它将把 shortVal 转换为 float。接着处理＋运算符，它不需要进行任何转换，因为这是把两个 float 值相加 (floatVal 和 shortVal * floatVal 的 float 结果)。最后，在处理＝运算符时，这个计算的 float 结果转换为 double。

这个转换过程初看起来比较复杂，但只要按照运算符的优先级，把表达式分解为不同的部分，就可以弄明白这个过程。

5.2 复杂的变量类型

到目前为止，所看到的都是 C#提供的简单变量类型。还有 3 个略复杂(但非常有用)的变量，如下所示：

- 枚举
- 结构
- 数组

5.2.1 枚举

到目前为止，我们见到的每种类型(除了 string 外)都有明确的取值范围。诚然，有些类型(如 double)的取值范围非常大，可以看作是连续的，但其中的数值总是固定的。最简单的范例是 bool

类型，它只能是两个值：true 或 false。

有时我们希望变量提取的是一个固定集合中的值。例如，orientation 类型可以存储 north, south, east 或 west 中的一个值。

此时，就可以使用枚举类型。枚举就可以完成这个 orientation 类型的任务：它们允许定义一个类型，提取我们提供的限定值集合中的一个值。

所以，我们需要创建自己的枚举类型 orientation，它可以从上述的 4 个值中提取一个值。注意这是一个附加的步骤——不是仅仅声明一个给定类型的变量，而是声明和描述一个用户定义的类型，再声明这个新类型的变量。

定义枚举

枚举可以使用 enum 关键字来定义，如下所示：

```
enum typeName
{
    value1,
    value2,
    value3,
    ...
    valueN
}
```

接着声明这个新类型的变量：

```
typeName varName;
```

并赋值：

```
varName = typeName.value;
```

枚举使用一个基本类型来存储。枚举类型可以提取的每个值都存储为该基本类型的一个值，在默认情况下，该类型为 int。在枚举声明中添加类型，就可以指定其他基本类型：

```
enum typeName : underlyingType
{
    value1,
    value2,
    value3,
    ...
    valueN
}
```

枚举的基本类型可以是 byte, sbyte, short, ushort, int, uint, long 和 ulong。

在默认情况下，每个值都会根据定义的顺序(从 0 开始)，自动赋给对应的基本类型值。这意味着 value1 的值是 0，value2 的值是 1，value3 的值是 2，等等。可以重写这个赋值过程：使用＝运算符，并指定每个枚举的实际值：

```
enum typeName : underlyingType
{
    value1 = actualVal1,
    value2 = actualVal2,
    value3 = actualVal3,
    ...
    valueN = actualValN
}
```

另外，还可以使用一个值作为另一个枚举的基础值，为多个枚举指定相同的值：

```
enum typeName : underlyingType
{
    value1 = actualVal1,
    value2 = value1,
    value3,
    ...
    valueN = actualValN
}
```

没有赋值的任何值都会自动获得一个初始值，这里使用的值是从比最后一个明确声明的值大 1 开始的序列。例如，在上面的代码中，value3 的值是 value1 + 1。

注意这可能会产生预料不到的问题，在一个定义后指定的值可能与其他值相同，例如 *value2* = value1。例如，在下面的代码中，value4 的值与 value2 相同。

```
enum typeName : underlyingType
{
    value1 = actualVal1,
    value2,
    value3 = value1,
    value4,
    ...
    valueN = actualValN
}
```

当然，如果这正是我们希望的结果，则代码就是正确的。

还要注意，以迂回方式赋值可能会产生一个错误，例如：

```
enum typeName : underlyingType
{
    value1 = value2,
    value2 = value1
}
```

下面看一个范例。

试试看：使用枚举

(1) 在目录 C:\BegCSharp\Chapter5 下创建一个新控制台应用程序 Ch05Ex02。

(2) 把下述代码添加到 Class1.cs 中：

```
namespace Ch05Ex02
{
    enum orientation : byte
    {
        north = 1,
        south = 2,
        east  = 3,
        west  = 4
    }
    /// <summary>
    /// Summary description for Class1.
    /// </summary>
    class Class1
    {
        static void Main(string[] args)
        {
            orientation myDirection = orientation.north;
            Console.WriteLine("myDirection = {0}", myDirection);
        }
    }
}
```

(3) 运行应用程序，控制台上应得到如图 5-5 所示的结果。



图 5-5

(4) 退出应用程序，修改代码，如下所示：

```
byte directionByte;
string directionString;
orientation myDirection = orientation.north;
Console.WriteLine("myDirection = {0}", myDirection);
directionByte = (byte)myDirection;
directionString = Convert.ToString(myDirection);
Console.WriteLine("byte equivalent = {0}", directionByte);
Console.WriteLine("string equivalent = {0}", directionString);
```

(5) 再次运行应用程序，如图 5-6 所示。



图　5-6

代码的说明

这段代码定义并使用了一个枚举类型 orientation。首先要注意的是，类型定义代码放在名称空间 Ch05Ex02 中，而没有与其他代码放在一起，这是因为在运行期间，定义代码并不是像执行应用程序中的代码行那样一行一行地执行。应用程序是从我们已经习惯的位置开始执行的，并可以访问新类型，因为它属于同一个名称空间。

这个范例的第一个迭代说明了创建新类型的变量、给它赋值以及把它输出到屏幕上的基本方法。

接着修改代码，把枚举值转换为其他类型。注意这里必须使用显式转换。即使 orientation 的基本类型是 byte，仍必须使用(byte)数据类型转换，把 myDirection 的值转换为 byte 类型：

```
directionByte = (byte)myDirection;
```

如果要把 byte 类型转换为 orientation，也需要进行显式转换。例如，可以使用下述代码把 byte 变量 myByte 转换为 orientation，并把这个值赋给 myDirection：

```
myDirection = (orientation)myByte;
```

当然，这里必须小心，因为并不是所有 byte 类型变量的值都可以映射为已定义的 orientation 值。Orientation 类型可以存储其他 byte 值，所以不会直接产生一个错误，但会在应用程序的后面违反逻辑。

要获得枚举的字符串值，可以使用 Convert.ToString()：

```
directionString = Convert.ToString(myDirection);
```

使用(string)数据类型转换是不行的，因为需要进行的处理并不仅仅是把存储在枚举变量中的数据放在 string 变量中，而是更复杂一些。

另外，还可以使用变量本身的 ToString()命令。下面的代码与使用 Convert.ToString()的效果相同：

```
directionString = myDirection.ToString();
```

也可以把 string 转换为枚举值，但其语法略复杂一些。有一个特定的命令用于这种类型的转换，即 Enum.Parse()，其使用方式如下：

```
(enumerationType)Enum.Parse(typeof(enumerationType), enumerationValueString);
```

它使用了另一个运算符 typeof，可以得到操作数的类型。对 orientation 类型使用这个命令，

如下所示：

```
string myString = "north";
orientation myDirection = (orientation)Enum.Parse(typeof(orientation),
    myString);
```

当然，并不是所有的字符串值都会映射为一个 orientation 值。如果一个值不能映射为枚举值中的一个，就会产生一个错误。与 C#中的其他值一样，这些值是区分大小写的，所以如果字符串与一个值相同，但大小写不同(例如，myString 设置为"North"，而不是"north")，就会产生一个错误。

5.2.2　结构

下一个要介绍的变量类型是结构(struct，structure 的简写)。结构就是由几个数据组成的数据结构，这些数据可能有不同的类型。根据这个结构，可以定义您自己的变量类型。例如，假定要存储从起点开始到某一位置的路由，其中路由由一个方向和一个距离值(英里)组成。为了简单起见，假定该方向是指南针上的一点(这样，方向就可以用上一节的 orientation 枚举来表示)，距离值可以用一个 double 类型表示。

下面就通过前面的代码，用两个不同的变量来表示该路由：

```
orientation myDirection;
double      myDistance;
```

像这样使用两个变量，是没有错误的，但在一个地方存储这些信息应该更为简单(特别是在需要多个路由时，就更为简单)。

定义结构

结构使用 struct 关键字来定义，如下所示：

```
struct <typeName>
{
    <memberDeclarations>
}
```

<memberDeclarations> 部分包含变量的定义(称为结构的数据成员)，其格式与往常一样。每个成员的声明采用如下形式：

```
<accessibility> <type> <name>;
```

要让调用结构的代码访问该结构的数据成员，可以对<accessibility>使用关键字 public，例如：

```
struct route
{
    public orientation direction;
    public double      distance;
```

```
}
```

定义了结构类型后，就可以定义新类型的变量，来使用该结构：

```
route myRoute;
```

通过句点字符访问这个组合变量中的数据成员：

```
myRoute.direction = orientation.north;
myRoute.distance = 2.5;
```

下面把这个类型放在一个范例中。

试试看：使用结构

(1) 在目录 C:\BegCSharp\Chapter5 下创建一个新控制台应用程序 Ch05Ex03。

(2) 把下述代码添加到 Class1.cs 中：

```
namespace Ch05Ex03
{
    enum orientation : byte
    {
        north = 1,
        south = 2,
        east  = 3,
        west  = 4
    }
    struct route
    {
        public orientation direction;
        public double      distance;
    }
    /// <summary>
    /// Summary description for Class1.
    /// </summary>
    class Class1
    {
        static void Main(string[] args)
        {
            route myRoute;
            int myDirection = -1;
            double myDistance;
            Console.WriteLine("1) North\n2) South\n3) East\n4) West");
            do
            {
                Console.WriteLine("Select a direction:");
                myDirection = Convert.ToInt32(Console.ReadLine());
```

```
            }
            while ((myDirection < 1) || (myDirection > 4));
            Console.WriteLine("Input a distance:");
            myDistance = Convert.ToDouble(Console.ReadLine());
            myRoute.direction = (orientation)myDirection;
            myRoute.distance = myDistance;
            Console.WriteLine("myRoute specifies a direction of {0} and a " +
                "distance of {1}", myRoute.direction, myRoute.distance);
        }
    }
}
```

(3) 执行代码，选择一个方向，输入一个距离值，如图 5-7 所示。

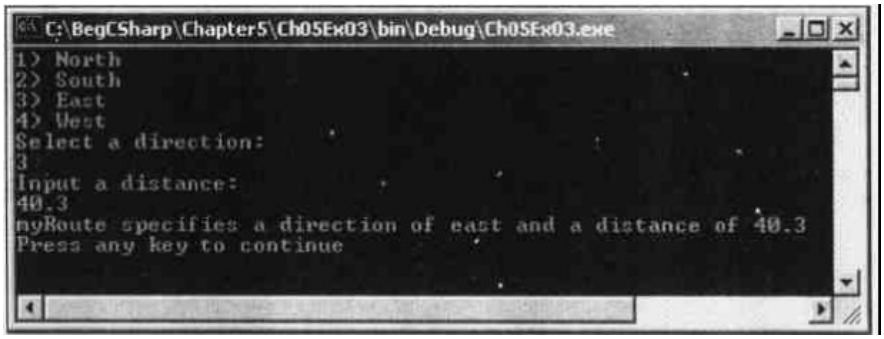


图　5-7

代码的说明

结构和枚举一样，是在代码的主体外部声明的。我们在名称空间声明中声明 route 结构和它使用的 orientation 枚举：

```
enum orientation : byte
{
    north = 1,
    south = 2,
    east  = 3,
    west  = 4
}
struct route
{
    public orientation direction;
    public double      distance;
}
```

代码的主体结构与前面的一些范例类似，要求用户输入一些信息，并显示它们。把方向选项放在 do 循环中，对用户的输入进行有效性检查，如果输入的信息不是一个 1~4 之间的整数(选择了这样的值后，它们就会映射到枚举成员上，以易于赋值)，就拒绝它。

注意，在引用路由的成员时，处理它们的方式与成员类型相同的变量完全一样。赋值语句

如下所示：

```
myRoute.direction = (orientation)myDirection;
myRoute.distance = myDistance;
```

可以直接把输入的值放到 myRoute.distance 中，而不会有负面效果，如下所示：

```
myRoute.distance = Convert.ToDouble(Console.ReadLine());
```

还应进行有效性检查，但这段代码没有这一步。

对结构成员的任何访问都以相同的方式处理。structVar.memberVar 形式的表达式可计算类型为 memberVar 的变量。

5.2.3 数组

前面的所有类型有一个共同点：它们都只存储一个值(结构中存储一组值)。有时，需要存储许多数据，这样就不方便了。有时需要同时存储类型相同的几个值，而不是每个值使用不同的变量。

例如，假定要对所有朋友的姓名执行一些操作。可以使用简单的字符串变量，如下所示：

```
string friendName1 = "Robert Barwell";
string friendName2 = "Mike Parry";
string friendName3 = "Jeremy Beacock";
```

但这看起来需要很多努力，特别是需要编写不同的代码来处理每个变量。例如，不能在循环中迭代这个字符串列表。

另一种方式是使用数组。数组是一个变量的下标列表，存储在数组类型的变量中。例如，有一个数组 friendNames 存储上述的 3 个名字。在方括号中指定下标，就可以访问该数组中的各个成员，如下所示：

```
friendNames[<index>]
```

这个下标是一个整数，第一个条目的下标是 0，第二个条目的下标是 1，依次类推。这样就可以使用循环遍历所有的元素，例如：

```
int i;
for (i = 0; i < 3; i++)
{
    Console.WriteLine("Name with index of {0}: {1}", i, friendNames[i]);
}
```

数组有一个基本类型，数组中的各个条目都是这种类型。FriendNames 数组的基本类型是字符串，因为它要存储 string 变量。

数组的条目常称为元素。

1. 声明数组

数组以下述方式声明：

```
<baseType>[] <name>;
```

其中，<baseType>可以是任何变量类型，包括本章前面介绍的枚举和结构类型。

数组必须在访问之前初始化，不能像下面这样访问数组或给数组赋值：

```
int[] myIntArray;
myIntArray[10] = 5;
```

数组的初始化有两种方式。可以以字面形式指定数组的完整内容，也可以指定数组的大小，再使用关键字 new 初始化所有的数组元素。

使用字面值指定数组，只需要提供一个用逗号分隔开的元素值列表，该列表放在花括号中，例如：

```
int[] myIntArray = {5, 9, 10, 2, 99};
```

其中，myIntArray 有 5 个元素，每个元素都被赋予了一个整数值。

另一种方式需要使用下述语法：

```
int[] myIntArray = new int[5];
```

这里使用关键字 new 显式地初始化数组，用一个常量定义其大小。这种方式会给所有的数组元素赋予同一个默认值，对于数值类型来说，其默认值是 0。也可以使用非常量的变量来进行初始化，例如：

```
int[] myIntArray = new int[arraySize];
```

还可以使用这两种初始化方式的组合：

```
int[] myIntArray = new int[5] {5, 9, 10, 2, 99};
```

使用这种方式，数组大小必须与元素个数相匹配。例如，不能编写如下代码：

```
int[] myIntArray = new int[10] {5, 9, 10, 2, 99};
```

其中数组定义为有 10 个元素，但却只定义了 5 个元素，所以编译会失败。如果使用变量定义其大小，该变量必须是一个常量，例如：

```
const int arraySize = 5;
int[] myIntArray = new int[arraySize] {5, 9, 10, 2, 99};
```

如果省略了关键字 const，这段代码就会失败。

与其他变量类型一样，不需要在声明的同时进行初始化。下面的代码是合法的：

```
int[] myIntArray;
myIntArray = new int[5];
```

下面看一个范例，试着编写一些代码。

试试看：使用数组

(1) 在目录 C:\BegCSharp\Chapter5 下创建一个新控制台应用程序 Ch05Ex04。

(2) 把下述代码添加到 Class1.cs 中：

```
static void Main(string[] args)
{
    string[] friendNames = {"Robert Barwell", "Mike Parry",
                            "Jeremy Beacock"};
    int i;
    Console.WriteLine("Here are {0} of my friends:",
                      friendNames.Length);
    for (i = 0; i < friendNames.Length; i++)
    {
        Console.WriteLine(friendNames[i]);
    }
}
```

(3) 执行代码，如图 5-8 所示。



图 5-8

代码的说明

这段代码用 3 个值建立了一个 string 数组，并在 for 循环中把它们列在控制台上。注意我们使用 friendNames.Length 来确定数组中的元素个数：

```
Console.WriteLine("Here are {0} of my friends:", friendNames.Length);
```

这是获取数组大小的方便方法。

在 for 循环中输出值容易出错。例如，把<改为<=，如下所示：

```
for (i = 0; i <= friendNames.Length; i++)
{
    Console.WriteLine(friendNames[i]);
}
```

编译代码，就会弹出如图 5-9 所示对话框。

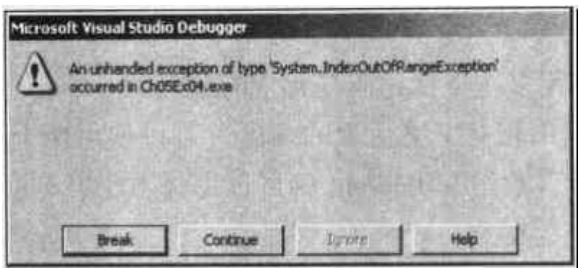


图　5-9

这里我们试图访问 friendNames[3]。记住，数组的下标从 0 开始，所以最后一个元素是 friendNames[2]。如果试图访问超出数组大小的元素，代码就会失败。

还有一个更有弹性的方法可以访问数组的所有元素，即使用 foreach 循环：

2. foreach 循环

foreach 循环可以使用简单的语法定位数组中的每个元素：

```
foreach (<baseType> <name> in <array>)
{
    // can use <name> for each element
}
```

这个循环会迭代每个元素，依次把每个元素放在变量<name>中，且没有访问非法元素的危险。不需要考虑数组中有多少个元素，并可以确保将在循环中使用每个元素。使用这个循环，可以修改上一个范例中的代码，如下所示：

```
static void Main(string[] args)
{
    string[] friendNames = {"Robert Barwell", "Mike Parry",
                            "Jeremy Beacock"};
    Console.WriteLine("Here are {0} of my friends:", friendNames.Length);
    foreach (string friendName in friendNames)
    {
        Console.WriteLine(friendName);
    }
}
```

这段代码的输出结果与前面的范例完全相同。

使用这种非法和标准的 for 循环的主要区别是 foreach 循环对数组内容进行只读访问，所以不能改变任何元素的值。例如，不能编写如下代码：

```
foreach (string friendName in friendNames)
{
    friendName = "Rupert the bear";
}
```

如果编译这段代码，就会失败。但如果使用简单的 for 循环，就可以给数组元素赋值。

3. 多维数组

实际上，多维数组是使用多个下标访问其元素的数组。

例如，假定要确定一座山相对于某位置的高度，可以使用两个坐标 x 和 y 来指定一个位置。把这两个坐标用作下标，数组 hillHeight 就可以用每对坐标来存储高度，这就要使用多维数组了。

像这样的二维数组可以声明如下：

```
<baseType>[,] <name>;
```

多维数组只需要更多的逗号，例如：

```
<baseType>[,,,] <name>;
```

这声明了一个四维数组。

赋值也使用类似的语法，用逗号分隔下标。要声明和初始化上面讨论的二维数组 hillHeight，其基本类型是 double，x 的值是 3，y 的值是 4，则需要：

```
double[,] hillHeight = new double[3,4];
```

另外，还可以使用字面值进行初始的赋值。这里使用嵌套的花括号，用逗号分隔开，例如：

```
double[,] hillHeight = {{1, 2, 3, 4}, {2, 3, 4, 5}, {3, 4, 5, 6}};
```

这个数组的大小与前面的相同，但其值是显式定义的。

要访问多维数组中的每个元素，只需指定它们的下标，并用逗号分隔开，例如：

```
hillHeight[2,1]
```

接着就可以像其他元素那样处理它了。

这个表达式将访问上面定义的第 3 个嵌套数组中的第 2 个元素(其值是 4)。记住，下标从 0 开始，第一个数字是嵌套的数字。换言之，第一个数字指定花括号对，第 2 个数字指定该对花括号中的元素。用图 5-10 来表示这个数组。

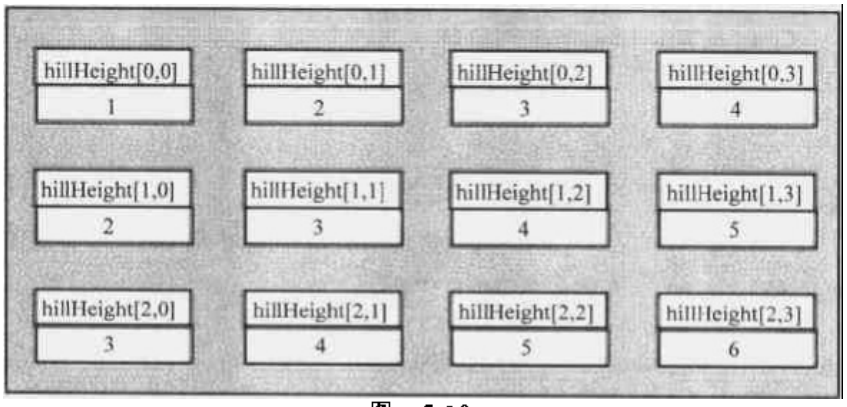


图　5-10

foreach 循环可以访问多维数组中的所有元素，其方式与一维数组相同，例如：

```
double[,] hillHeight = {{1, 2, 3, 4}, {2, 3, 4, 5}, {3, 4, 5, 6}};
foreach (double height in hillHeight)
{
    Console.WriteLine("{0}", height);
}
```

元素的输出顺序与赋予字面值的顺序相同：

```
hillHeight[0,0]
hillHeight[0,1]
hillHeight[0,2]
hillHeight[0,3]
hillHeight[1,0]
hillHeight[1,1]
hillHeight[1,2]
```

4. 数组的数组

上一节讨论的多维数组可称为矩形数组，这是因为每一行的元素个数都相同。使用上一个范例，任何一个 x 坐标都可以对应 0~2 的 y 坐标。

也可以使用直角数组，其中每行都有不同的元素个数。为此，需要有这样一个数组，其中的每个元素都是另一个数组。也可以有数组的数组的数组，甚或更复杂的数组。但是，注意这些数组都必须有相同的基本类型。

声明数组的数组，其语法要在数组的声明中指定多个方括号对，例如：

```
int[][] jaggedIntArray;
```

但初始化这样的数组不像初始化多维数组那样简单，不能进行这样的声明：

```
jaggedIntArray = new int[3][4];
```

即使这样做了，也不是很有效，因为这只能得到与多维数组相同的结果。也不能使用下面的代码：

```
jaggedIntArray = {{1, 2, 3}, {1}, {1, 2}};
```

有两种方式：可以初始化包含其他数组的数组(为了清晰起见，称之为子数组)，然后依次初始化子数组：

```
jaggedIntArray = new int[2][];
jaggedIntArray[0] = new int[3];
jaggedIntArray[1] = new int[4];
```

也可以使用上述字面值赋值的一种改进形式：

```
jaggedIntArray = {new int[] {1, 2, 3}, new int[] {1}, new int[] {1, 2}};
```

对直角数组可以使用 foreach 循环，但通常需要嵌套该循环，才能得到实际的数据。例如，

假定下述直角数组包含 10 个子数组，每个子数组又包含一个整数数组，其元素是 1~10 的约数：

```
int[][] divisors1To10 = {new int[] {1},
                         new int[] {1, 2},
                         new int[] {1, 3},
                         new int[] {1, 2, 4},
                         new int[] {1, 5},
                         new int[] {1, 2, 3, 6},
                         new int[] {1, 7},
                         new int[] {1, 2, 4, 8},
                         new int[] {1, 3, 9},
                         new int[] {1, 2, 5, 10}};
```

下面的代码会失败：

```
foreach (int divisor in divisors1To10)
{
    Console.WriteLine(divisor);
}
```

这是因为数组 divisors1To10 包含 int[]元素，而不是 int 元素。我们必须循环每个子数组和该数组本身：

```
foreach (int[] divisorsOfInt in divisors1To10)
{
    foreach(int divisor in divisorsOfInt)
    {
        Console.WriteLine(divisor);
    }
}
```

可以看出，使用直角数组的语法要复杂得多！在大多数情况下，使用矩形数组比较简单，这是一种比较简单的存储方式。但是，有时必须使用直角数组，而工作效率不会有损失！

5.3 字符串的处理

到目前为止，对字符串的使用还仅限于把字符串写到控制台上，从控制台上读取字符串，以及使用＋运算符连接字符串。下面用几页的篇幅介绍 C#中比较常用的字符串处理技巧。

首先要注意，string 类型变量可以看作是 char 变量的只读数组。这样，就可以使用下面的语法访问每个字符：

```
string myString = "A string";
char myChar = myString[1];
```

但是，不能用这种方式指定各个字符。

为了获得一个可写的 char 数组，可以使用下面的代码，其中使用了数组变量的 ToCharArray() 命令。

```
string myString = "A string";
char[] myChars = myString.ToCharArray();
```

接着就可以用标准方式处理 char 数组了。

也可以在 foreach 循环中使用字符串，例如：

```
foreach (char character in myString)
{
    Console.WriteLine("{0}", character);
}
```

与数组一样，还可以使用 myString.Length 获取元素的个数，这将给出字符串中字符的个数，例如：

```
string myString = Console.ReadLine();
Console.WriteLine("You typed {0} characters.", myString.Length);
```

其他的基本字符串处理技巧以与这个<string>.ToCharArray()方法类似的格式使用命令。有两个简单但很有效的命令是<string>.ToLower()和<string>.ToUpper()。它们可以把字符串转换为大写或小写形式。要明白为什么它们非常有用，可以考虑下面的情形：要检查用户的某个响应，例如字符串 yes。如果可以把用户输入的字符串转换为小写形式，就也能检查字符串"YES", "Yes", "yeS"等。

```
string userResponse = Console.ReadLine();
if (userResponse.ToLower() == "yes")
{
    // act on response
}
```

注意，这个命令与本节的其他命令一样，并没有把字符串改成它应用的形式。把这个命令与字符串合并使用，就会创建一个新的字符串，以便与另一个字符串进行比较(如上所述)，或者赋给另一个变量。该变量可能与当前操作的变量相同，例如：

```
userResponse = userResponse.ToLower();
```

这是一个要点，因为只写出下面的代码是没用的：

```
userResponse.ToLower();
```

下面看看在简化用户输入方面还可以做什么。如果用户无意间在输入内容的前面或后面添加了额外的空格，会怎样？此时，上述代码就不工作了。这就需要删除输入字符串中的空格，此时可以使用<string>.Trim()命令来处理。

```
string userResponse = Console.ReadLine();
```

```
userResponse = userResponse.Trim();
if (userResponse.ToLower() == "yes")
{
    // act on response
}
```

使用该命令，还可以检测下述字符串：

```
"  YES"
"Yes  "
```

也可以使用这些命令删除其他字符，只要在一个 char 数组中指定这些字符即可，例如：

```
char[] trimChars = {' ', 'e', 's'};
string userResponse = Console.ReadLine();
userResponse = userResponse.ToLower();
userResponse = userResponse.Trim(trimChars);
if (userResponse == "y")
{
    // act on response
}
```

这将从字符串的前面或后面删除所有的空格、字母 e 和 s。如果字符串中没有其他字符，结果就是检测以下字符串：

```
"Yeeeees"
" y"
```

还可以使用<string>.TrimStart()和<string>.TrimEnd()命令。它们可以把字符串的前面或后面的空格删掉。这些命令也需要指定 char 数组。

还有另外两个字符串命令可以处理字符串的空格：<string>.PadLeft() 和<string>.PadRight()。它们可以在字符串的左边或右边添加空格，使字符串达到指定的长度。其语法如下：

```
<string>.PadX(<desiredLength>);
```

例如：

```
myString = "Aligned";
myString = myString.PadLeft(10);
```

这将在 myString 中把 3 个空格添加到单词"Aligned"的左边。这些方法可以用于在列中对齐字符串，特别适合于在其他字符串的下面放置一定数量的字符串。

与删除命令一样，还可以以第二种方式使用这些命令，即提供要添加到字符串上的字符，这需要一个 char，而不是像删除命令那样的 char 数组。例如：

```
myString = "Aligned";
```

```
myString = myString.PadLeft(10, '-');
```

这将在 myString 的开头加上 3 个短横线。

还有许多这样的字符串处理命令，其中一些只用于非常特殊的情况。将在后面的章节中遇到它们时进行讨论。在继续下面内容之前，介绍一个前几章涉及到(特别是本章)的 VS 特性。

试试看：VS 中的语句自动完成功能

(1) 在目录 C:\BegCSharp\Chapter5 下创建一个新控制台应用程序 Ch05Ex05。

(2) 在 Class1.cs 中输入下述代码：

```
static void Main(string[] args)
{
    string myString = "This is a test.";
    char[] separator = {' '};
    string[] myWords;
    myWords = myString.
}
```

(3) 输入最后的句点时，注意会弹出如图 5-11 所示窗口。

(4) 不要移动光标，输入 s，弹出的窗口就会改变，显示一个黄色的工具提示窗口，如图 5-12 所示。



图 5-11

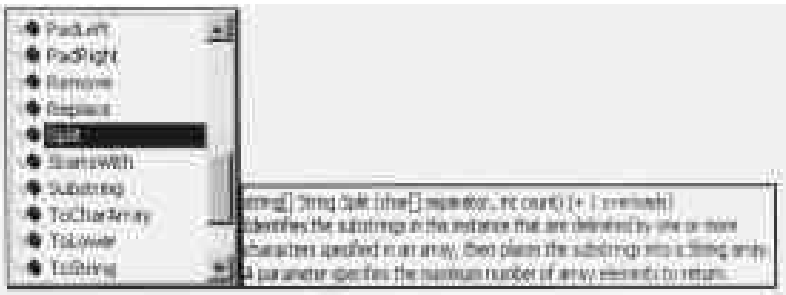


图 5-12

(5) 输入字符串"(separator);"。代码如下所示，弹出的窗口也会消失。

```
static void Main(string[] args)
{
    string myString = "This is a test.";
    char[] separator = {' '};
    string[] myWords;
    myWords = myString.Split(separator);
}
```

(6) 添加下述代码，注意弹出的窗口。

```
static void Main(string[] args)
{
    string myString = "This is a test.";
```

```
        char[] separator = {' '};
        string[] myWords;
        myWords = myString.Split(separator);
        foreach (string word in myWords)
        {
            Console.WriteLine("{0}", word);
        }
    }
```

(7) 执行代码，如图 5-13 所示。



图　5-13

代码的说明

在这段代码中，要注意两点。第一点是所使用的新字符串命令，第二点是使用了 VS 中的自动完成功能。下面一起讨论它们。

使用命令<string>.Split()把 string 转换为 string 数组，把它在指定的位置分隔开。这些位置采用 char 数组的形式，在本例中该数组只有一个元素，即空格字符：

```
char[] separator = {' '};
```

把字符串在每个空格处分解开时，下面的代码会得到其子字符串，即得到单个单词的数组：

```
string[] myWords;
myWords = myString.Split(separator);
```

接着使用 foreach 循环迭代这个数组中的单词，并把这些单词写到控制台上：

```
foreach (string word in myWords)
{
    Console.WriteLine("{0}", word);
}
```

注意：

得到的每个单词都没有空格，单词的内部和两端也没有空格。在使用 split()时，删除了分隔符。

接着，看看自动完成功能。VS 是一个非常智能化的程序包，在用户输入代码时会提供许多关于代码的信息。在 myString 的后面输入句点时，VS 知道 myString 是一个字符串，也知道你要指定一个字符串命令，并显示所有可用的选项。此时可以停止输入，使用上下箭头键选择需要的命令。在这些可用命令中移动时，VS 会告诉你当前选中命令的含义，以及使用它的语法。

在开始输入更多的字符时，VS 会把选中的命令移动到命令的顶部，以便自动输入该命令。一旦它显示出需要的命令，就可以继续输入，就像输入完整的名称一样，所以输入"("会直接跳到指定的位置上，在该位置上，有某些命令需要的额外信息——VS 甚至会告诉用户该信息必须采用的格式，显示这些接受各种信息的命令选项。

VS 的这个功能使用起来非常方便，可以轻松地查找奇怪类型的信息。看看 string 类型的所有命令是很有趣的，这不会使计算机崩溃的，试试看吧！

5.4　小结

本章用一定的篇幅扩展了变量的知识，填补以前的一些空白。本章最重要的问题是类型转换。掌握本章介绍的概念，会使以后的学习容易得多。

另外还介绍了几个变量类型，它们可以以更友好的方式存储数据。本章阐述了枚举如何使数值更容易辨识，从而使代码的可读性更高，结构如何在一个地方合并多个相关的数据元素，如何把类似的数据组合到数组中。本书的其他地方常常要用到这些类型。

最后，主要介绍了字符串的处理，讨论了一些基本技巧和规则。C#提供了许多字符串命令，但这里只介绍了其中的几个，还说明了如何学习 VS 中可用的命令。使用这些技巧可以尝试许多工作。至少下面的范例可以使用本章没有讨论的一个或多个字符串命令来完成，但将不会告诉你该使用哪些命令！

5.5　练习

(1) 下面的转换哪些不是隐式转换？

a) int 转换为 short

b) short 转换为 int

c) bool 转换为 string

d) byte 转换为 float

(2) short 类型包含彩虹的颜色，再加上黑色和白色，据此编写 color 枚举的代码。这个枚举可以使用 byte 类型吗？

(3) 修改第 4 章的 Mandelbrot 图像生成范例，使用下面的结构表示复杂的数字：

```
struct imagNum
{
    public double real, imag;
}
```

(4) 下面的代码可以成功编译吗？如果不能，为什么？

```
string[] blab = new string[5]
string[5] = 5th string.
```

(5) 编写一个控制台应用程序，它接收用户输入的一个字符串，将其中的字符以与输入相反的顺序输出。

(6) 编写一个控制台应用程序，它接收一个字符串，用"yes"替换字符串中所有的"no"。

(7) 编写一个控制台应用程序，给字符串中的每个单词加上双引号。

第6章 函 数

到目前为止所看到的代码都是以一个代码块的形式出现的，其中有一些重复执行的循环代码，和有条件地执行的分支语句。如果需要对数据执行某种操作，就应把需要的代码放在合适的地方。

这种代码结构的作用是有限的。某些任务常常需要在一个程序中执行好几次，例如查找数组中的最大值。此时可以把相同(或几乎相同)的代码块按照需要放在应用程序中，但这样做也会有问题。在某个常见任务中，即使进行非常小的改动(例如，修改某个代码错误)，也需要修改多个代码块，这些代码块可能分布在整个应用程序中。忘了修改其中的一个代码块就会产生很大的影响，导致整个应用程序失败。另外，应用程序也比较长。

这个问题的解决方法是使用函数。在C#中，函数是一种方法，可提供在应用程序中的任何一处执行的代码块。

提示：

本章介绍的特定类型的函数称为方法。但是，这个术语在.NET编程中有非常特殊的含义，本书的后面会详细讨论它，所以现在不使用这个术语。

例如，有一个函数返回数组中的最大值，可以在代码的任何位置使用这个函数，且在每个地方都使用相同的代码行。因为只需要提供一次这段代码，所以对代码的任何修改都只影响使用该函数进行的计算。这个函数可以看作包含可重用的代码。

函数还可以使代码的可读性更高，因为可以使用函数把相关的代码组合在一起。如果这么做，应用程序主体就会非常短，因为代码的内部工作部分被分散了。这类似于在VS中使用大纲视图把代码分解为各个区域，得到应用程序更富有逻辑的结构。

函数还可以用于创建多用途的代码，让它们对不同的数据执行相同的操作。可以以参数的形式为一个函数提供信息，以返回值的形式得到函数的结果。在上面的范例中，参数就是一个要搜索的数组，而返回值就是数组中的最大值。这意味着每次可以使用同一个函数处理不同的数组。函数的参数和返回值共同定义了函数的签名。

本章首先介绍定义和使用简单函数的方法，该函数不接受或返回任何数据。之后，论述在函数之间传送数据的方式。

接着，讨论变量作用域的问题，这涉及到C#应用程序中如何把数据定位到特定代码区域，在把代码分解到多个函数中时，这个问题特别重要。

然后，深入探讨C#应用程序中的一个重要函数：Main()。将论述如何使用这个函数的内置功能来利用命令行参数，以便在运行应用程序时，把信息传送给它们。

其后，讨论第5章介绍的结构类型的另一个特性，即可以把函数作为结构类型的成员。

最后探讨两个比较高级的主题：函数的重载和委托。函数重载技术可以提供名称相同但签

名不同的多个函数，委托是一种变量类型，可以间接地使用函数。同一个委托可用于调用匹配于特定签名的任何函数，提供在运行期间在几个函数之间进行选择的能力。

6.1 定义和使用函数

本节介绍如何把函数添加到应用程序中，以及如何在代码中使用(调用)它们。首先从基础知识开始，看看不交换任何数据的简单函数以及调用它们的代码，然后介绍更高级的函数用法。

首先看一个范例。

试试看：定义和使用基本函数

(1) 在目录 C:\BegCSharp\Chapter6 下创建一个新控制台应用程序 Ch06Ex01。

(2) 把下述代码添加到 Class1.cs 中：

```
class Class1
{
    static void Write()
    {
        Console.WriteLine("Text output from function.");
    }

    static void Main(string[] args)
    {
        Write();
    }
}
```

(3) 执行代码，如图 6-1 所示。



图 6-1

代码的说明

下面的 4 行代码定义了函数 Write()：

```
static void Write()
{
    Console.WriteLine("Text output from function.");
}
```

这些代码把一些文本输出到控制台窗口中。但此时这些并不重要。而更关心函数定义和使用的机制。

函数定义由以下几部分组成：

- 两个关键字：static 和 void
- 函数名后跟圆括号，如 Write()
- 一个要执行的代码块，放在花括号中

定义 Write()函数的代码看起来非常类似于应用程序中的其他代码：

```
static void Main(string[] args)
{
    ...
}
```

这是因为，到目前为止所编写的代码(除了类型定义之外)都是函数的一部分。函数 Main()是(如第 3 章所述)控制台应用程序的入口点函数。当执行一个 C#应用程序时，就会调用它包含的入口点函数，这个函数执行完后，应用程序就终止了。所有的 C#可执行代码都必须有一个入口点。

Main()函数和 Write()函数的惟一区别(除了它们包含的代码))是在函数名 Main 后面的圆括号中还有一些代码，这是我们指定参数的方式，详见后面的内容。

如上所述，Main()函数和 Write()函数都是使用关键字 static 和 void 定义的。关键字 static 与面向对象的概念相关，本书在后面讨论。现在只需记住，在本节的应用程序中所使用的所有函数必须使用这个关键字。

另一方面，void 解释起来就要简单得多。这个关键字表明函数没有返回值。在本章的后面，将讨论函数有返回值时需要编写什么代码。

继续下去，调用函数的代码如下所示：

```
Write();
```

输入函数名，后跟空括号即可。在程序执行到这行代码时，就会运行 Write()函数中的代码。

注意：

在函数定义和调用函数时，必须使用圆括号。如果删除它们，代码就不能编译。

6.1.1 返回值

通过函数进行数据交换的最简单方式是利用返回值。有返回值的函数会计算这个值，其方式与在表达式中使用变量计算它们包含的值完全相同。与变量一样，返回值也有数据类型。

例如，有一个函数 getString()，其返回值是一个字符串，可以在代码中使用该函数，如下所示：

```
string myString;
myString = getString();
```

另外，还有一个函数 getVal()，它返回一个 double 值，可以在一个数学表达式中使用它。

```
double myVal;
```

```
double multipler = 5.3;
myVal = getVal() * multiplier;
```

当函数返回一个值时，可以用两种方式修改函数。

- 在函数声明中指定返回值的类型，但不使用关键字 void。
- 使用 return 关键字结束函数的执行，把返回值传送给调用代码。

在代码的术语中，控制台应用程序中的下述代码看起来像是前面见过的函数类型：

```
static <returnType> <functionName>()
{
    ...
    return <returnValue>;
}
```

这里惟一的限制是<returnValue>必须是一个值，其类型可以是<returnType>，也可以隐式转换为该类型。但是，<returnType>可以是任何类型，包括前面介绍的较复杂的类型。

这段代码可以很简单：

```
static double getVal()
{
    return 3.2;
}
```

但是，返回值通常是函数执行的一些处理的结果，如上所示，返回值是使用 const 变量得到的。

在执行到 return 语句时，程序会立即返回调用代码。这个语句后面的代码都不会执行。但是，这并不意味着 return 语句只能放在函数体的最后一行。可以在前边的代码里使用 return，也可能在执行了分支逻辑之后使用。把 return 语句放在 for 循环中、if 块中，或其他结构中，会使该结构立即终止，函数也立即终止。例如：

```
static double getVal()
{
    double checkVal;
    // checkVal assigned a value through some logic.
    if (checkVal < 5)
        return 4.7;
    return 3.2;
}
```

根据 checkVal 的值，将返回两个值中的一个。

这里惟一的限制是 return 语句必须在函数的闭合花括号 } 之前处理。下面的代码是不合法的：

```
static double getVal()
{
```

```
    double checkVal;
    // checkVal assigned a value through some logic.
    if (checkVal < 5)
        return 4.7;
}
```

如果 checkVal>= 5，就不会执行到 return 语句，这是不允许的。所有的处理路径都必须最终汇总至一处。

最后要注意的是，return 可以用在通过 void 关键字声明的函数中(没有返回值)。如果这么做，函数就会立即终止。以这种方式使用 return 语句时，在 return 关键字和其后的分号之间提供返回值是错误的。

6.1.2 参数

当函数接受参数时，就必须指定下述内容：

- 函数在其定义中指定接受的参数列表，以及这些参数的类型。
- 在每个函数调用中匹配的参数列表。

这涉及到下述代码：

```
static <returnType> <functionName>(<paramType> <paramName>, ...)
{
    ...
    return <returnValue>;
}
```

其中可以有任意多个参数，每个参数都有一个类型和一个名称。参数用逗号分隔开。每个参数都在函数的代码中用作一个变量。

例如，下面是一个简单的函数，带两个参数，并返回它们的乘积：

```
static double product(double param1, double param2)
{
    return param1 * param2;
}
```

下面看一个比较复杂的范例。

试试看：用函数交换数值

(1) 在目录 C:\BegCSharp\Chapter6 下创建一个新控制台应用程序 Ch06Ex02。

(2) 把下述代码添加到 Class1.cs 中：

```
class Class1
{
    static int MaxValue(int[] intArray)
    {
        int maxVal = intArray[0];
```

```
        for (int i = 1; i < intArray.Length; i++)
        {
            if (intArray[i] > maxVal)
                maxVal = intArray[i];
        }
        return maxVal;
    }

    static void Main(string[] args)
    {
        int[] myArray = {1, 8, 3, 6, 2, 5, 9, 3, 0, 2};
        int maxVal = MaxValue(myArray);
        Console.WriteLine("The maximum value in myArray is {0}", maxVal);
    }
}
```

(3) 执行代码如图 6-2 所示。



图　6-2

代码的说明

这段代码包含一个函数，它执行的任务就是本章引言中范例函数所完成的任务。该函数的参数是一个整数数组，返回该数组中的最大值。该函数的定义如下所示：

```
static int MaxValue(int[] intArray)
{
    int maxVal = intArray[0];
    for (int i = 1; i < intArray.Length; i++)
    {
        if (intArray[i] > maxVal)
            maxVal = intArray[i];
    }
    return maxVal;
}
```

函数 MaxValue()定义了一个参数，即 int 数组 intArray，它还有一个 int 类型的返回值。最大值的计算是很简单的。局部整型变量 maxVal 初始化为数组中的第一个值，然后把这个值与数组中后面的每个元素依序进行比较。如果一个元素的值比 maxVal 大，就用这个值代替 maxVal 的当前值。循环结束后，maxVal 就包含数组中的最大值，用 return 语句返回。

Main()中的代码声明并初始化一个简单的整数数组，用于 MaxValue()函数：

```
int[] myArray = {1, 8, 3, 6, 2, 5, 9, 3, 0, 2};
```

调用 MaxValue()，把一个值赋给 int 变量 maxVal：

```
int maxVal = MaxValue(myArray);
```

接着，使用 Console.WriteLine()把这个值写到屏幕上：

```
Console.WriteLine("The maximum value in myArray is {0}", maxVal);
```

1. 参数匹配

在调用函数时，必须使参数与函数定义中指定的参数完全匹配，这意味着要匹配参数的类型、个数和顺序。例如，下面的函数：

```
static void myFunction(string myString, double myDouble)
{
    ...
}
```

不能使用下面的代码调用：

```
myFunction (2.6, "Hello");
```

这里试图把一个 double 值作为第一个参数传递，把 string 值作为第二个参数传递，参数的顺序与函数声明中定义的顺序不匹配。

也不能使用下面的代码：

```
myFunction("Hello");
```

这里仅传送了一个 string 参数，而该函数需要两个参数。

使用上述两个函数调用都会产生编译错误，因为编译器要求必须匹配函数的签名。

再回过头来看看这个范例，MaxValue()只能用于获取整数数组中的最大值 int。如果用下面的代码替换 Main()中的代码：

```
static void Main(string[] args)
{
    double[] myArray = {1.3, 8.9, 3.3, 6.5, 2.7, 5.3};
    double maxVal = MaxValue(myArray);
    Console.WriteLine("The maximum value in myArray is {0}", maxVal);
}
```

这段代码就不能编译，因为参数类型是错误的。

在本章的后面将介绍解决这个问题的一个有效技术。

2. 参数数组

C#允许为函数指定一个(只能指定一个)特定的参数，这个参数必须是函数定义中的最后一个参数，称为参数数组。参数数组可以使用个数不定的参数调用函数，它可以使用 params 关键字来定义。

参数数组可以简化代码，因为不必从调用代码中传递数组，而可以传递可在函数中使用的一个数组中相同类型的几个参数。

定义使用参数数组的函数时，需要使用下述代码：

```
static <returnType> <functionName>(<p1Type> <p1Name>, ... ,
                                   params <type>[] <name>)
{
   ...
   return <returnValue>;
}
```

使用下面的代码可以调用该函数。

```
<functionName>(<p1>, ... , <val1>, <val2>, ...)
```

其中<val1>, <val2>等都是类型为<type>的值，用于初始化<name>数组。在可以指定的参数个数方面没有限制。惟一的限制是它们都必须是<type>类型。甚至可以根本不指定参数。

这一点使参数数组特别适合于为在处理过程中要使用的函数指定其他信息。例如，假定有一个函数 getWord()，它的第一个参数是一个 string 值，并返回字符串中的第一个单词。

```
string firstWord = getWord("This is a sentence.");
```

其中 firstWord 被赋予字符串"This"。

可以在 getWord()中添加一个 params 参数，以根据其下标选择另一个要返回的单词：

```
string firstWord = getWord("This is a sentence.", 2);
```

假定第一个单词计数为 1，则 firstWord 就被赋予字符串"is"。

也可以在第 3 个参数中限制返回的字符个数，同样通过 params 参数来实现：

```
string firstWord = getWord("This is a sentence.", 4, 3);
```

其中 firstWord 被赋予字符串"sen"。

下面看一个完整的范例。

试试看：通过函数交换数据(2)

(1) 在目录 C:\BegCSharp\Chapter6 下创建一个控制台应用程序 Ch06Ex03。

(2) 把下述代码添加到 Class1.cs 中：

```
class Class1
{
   static int sumVals(params int[] vals)
   {
      int sum = 0;
      foreach (int val in vals)
      {
```

```
            sum += val;
        }
        return sum;
    }

    static void Main(string[] args)
    {
        int sum = sumVals(1, 5, 2, 9, 8);
        Console.WriteLine("Summed Values = {0}", sum);
    }
}
```

(3) 执行代码，如图 6-3 所示。



图　6-3

代码的说明

在这个范例中，函数 sumVals()是用关键字 params 定义的，可以接受任意个 int 参数(或不接受任何参数)：

```
static int sumVals(params int[] vals)
{
    ...
}
```

这个函数中的代码通过 vals 数组中的值迭代，把这些值加在一起，返回其结果。

在 Main()中，用 5 个整型参数调用这个函数：

```
int sum = sumVals (1, 5, 2, 9, 8);
```

也可以用 0、1、2 或 100 个整型参数调用这个函数——参数的个数没有限制。

3. 引用和值参数

到目前为止，本章定义的所有函数都带有值参数。其含义是，在使用参数时，是把一个值传递给函数使用的一个变量。对函数中此变量的任何修改都不影响函数调用中指定的参数。例如，下面的函数使传递过来的参数值加倍，并显示出来：

```
static void showDouble(int val)
{
    val *= 2;
    Console.WriteLine("val doubled = {0}", val);
}
```

参数 val 在这个函数中被加倍，如果以下面的方式调用它：

```
int myNumber = 5;
Console.WriteLine("myNumber = {0}", myNumber);
showDouble(myNumber);
Console.WriteLine("myNumber = {0}", myNumber);
```

输出到控制台上的文本如下所示：

```
myNumber = 5
val doubled = 10
myNumber = 5
```

把 myNumber 作为一个参数，调用 showDouble()并不影响 Main()中 myNumber 的值，即使分配给 val 的参数被加倍，myNumber 的值也不变。

这很不错，但如果要改变 myNumber 的值，就会有问题。可以使用一个给 myNumber 返回新值的函数，并使用下面的代码调用它：

```
int myNumber = 5;
Console.WriteLine("myNumber = {0}", myNumber);
myNumber = showDouble(myNumber);
Console.WriteLine("myNumber = {0}", myNumber);
```

但这段代码一点也不直观，且不能处理参数的多个变量值的改变(因为函数只有一个返回值)。

此时可以通过引用传递参数。即函数处理的变量与函数调用中使用的变量相同，而不仅仅是值相同的变量。因此，对这个变量进行的任何改变都会影响用作参数的变量值。为此，只需使用 ref 关键字指定参数：

```
static void showDouble(ref int val)
{
    val *= 2;
    Console.WriteLine("val doubled = {0}", val);
}
```

在函数调用中(这是必须的，因为 ref 参数是函数签名的一部分)：

```
int myNumber = 5;
Console.WriteLine("myNumber = {0}", myNumber);
showDouble(ref myNumber);
Console.WriteLine("myNumber = {0}", myNumber);
```

输出到控制台上的文本如下所示：

```
myNumber = 5
val doubled = 10
myNumber = 10
```

这次，myNumberhas 被 showDouble()修改了。

用作 ref 参数的变量有两个限制。首先，函数可能会改变引用参数的值，所以必须在函数调用中使用变量。所以，下面的代码是非法的：

```
const int myNumber = 5;
Console.WriteLine("myNumber = {0}", myNumber);
showDouble(ref myNumber);
Console.WriteLine("myNumber = {0}", myNumber);
```

其次，必须使用初始化过的变量。C#不允许假定 ref 参数在使用它的函数中初始化，下面的代码也是非法的：

```
int myNumber;
showDouble(ref myNumber);
Console.WriteLine("myNumber = {0}", myNumber);
```

4. 输出参数

除了根据引用传递值之外，还可以使用 out 关键字，指定给定的参数是一个输出参数。out 关键字的使用方式与 ref 关键字相同(在函数定义和函数调用中用作参数的修饰符)。实际上，它的执行方式与引用参数完全一样，因为在函数执行完毕后，该参数的值将返回给函数调用中使用的变量。但是，这里有一些重要区别。把未赋值的变量用作 ref 参数是非法的，但可以把未赋值的变量用作 out 参数。另外，在函数使用 out 参数时，该参数必须看作是还未赋值。即可以把已赋值的变量用作 out 参数，在函数调用中，存储在该变量中的值会在函数执行时丢失。

例如，考虑前面返回数组中最大值的 MaxValue()函数，对它进行扩展，略微修改该函数，获取数组中最大值的元素下标，为了简单起见，如果数组中有多个元素的值都是这个最大值，只提取第一个最大值的下标。为此，通过修改函数，添加一个输出参数，如下所示：

```
static int MaxValue(int[] intArray, out int maxIndex)
{
    int maxVal = intArray[0];
    maxIndex = 0;
    for (int i = 1; i < intArray.Length; i++)
    {
        if (intArray[i] > maxVal)
        {
            maxVal = intArray[i];
            maxIndex = i;
        }
    }
    return maxVal;
}
```

可以用下述方式使用该函数：

```
int[] myArray = {1, 8, 3, 6, 2, 5, 9, 3, 0, 2};
int maxIndex;
Console.WriteLine("The maximum value in myArray is {0}",
                  MaxValue(myArray, out maxIndex));
Console.WriteLine("The first occurrence of this value is at element {0}",
                  maxIndex + 1);
```

结果是：

```
The maximum value in myArray is 9
The first occurrence of this value is at element 7
```

要注意的一个要点是，必须在函数调用中使用 out 关键字，就像 ref 关键字一样。

注意：

在屏幕上显示结果时，给返回的 maxIndex 的值加上 1。这样可以使下标更容易读懂，因此数组的第一个元素指元素 1，而不是元素 0。

6.2 变量的作用域

在上一节中，读者可能想知道为什么需要利用函数交换数据。原因是 C#中的变量仅能从定位的代码区域访问。给定的变量有一个作用域，访问该变量要通过这个作用域来实现。

变量的作用域是一个重要的主题，最好用一个范例来说明。

试试看：定义和使用基本函数

(1) 对 Class1.cs 中的 Ch06Ex01 进行如下修改：

```
class Class1
{
    static void Write()
    {
        Console.WriteLine("myString = {0}", myString);
    }

    static void Main(string[] args)
    {
        string myString = "String defined in Main()";
        Write();
    }
}
```

(2) 编译代码，注意显示在任务列表中的错误和警告：

```
The name 'myString' does not exist in the class or namespace 'Ch06Ex01.Class1'
```

```
The variable 'myString' is assigned but its value is never used
```

代码的说明

那么，什么地方出错？在应用程序主体(Main()函数)中定义的变量 myString 不能在 Write()函数中访问。

原因是变量有一个作用域，在这个作用域中，变量才是有效的。这个作用域包括定义变量的代码块和直接嵌套在其中的代码块。函数中的代码块与调用它们的代码块是分离的。在 Write()中，没有定义 myString，在 Main()中定义的 myString 则超出了作用域—— 它只能在 Main()中使用。

实际上，在 Write()中有一个完全独立的变量 myString，修改代码，如下所示：

```
class Class1
{
    static void Write()
    {
        string myString = "String defined in Write()";
        Console.WriteLine("Now in Write()");
        Console.WriteLine("myString = {0}", myString);
    }

    static void Main(string[] args)
    {
        string myString = "String defined in Main()";
        Write();
        Console.WriteLine("\nNow in Main()");
        Console.WriteLine("myString = {0}", myString);
    }
}
```

这段代码就可以编译，结果如图 6-4 所示。



图 6-4

这段代码执行的操作如下：

- Main()定义和初始化一个字符串变量 myString。
- Main() 把控制权传送给 Write()。
- Write()定义和初始化一个字符串变量 myString，它与 Main()中定义的 myString 变量完全不同。
- Write()把一个字符串输出到控制台上，该字符串包含在 Write()中定义的 myString 的值。

- Write()把控制权传送回 Main()。
- Main()把一个字符串输出到控制台上，该字符串包含在 Main()中定义的 myString 的值。

作用域以这种方式覆盖一个函数的变量称为局部变量。还有一种全局变量，其作用域可覆盖几个函数。修改代码，如下所示：

```
class Class1
{
    static string myString;

    static void Write()
    {
        string myString = "String defined in Write()";
        Console.WriteLine("Now in Write()");
        Console.WriteLine("Local myString = {0}", myString);
        Console.WriteLine("Global myString = {0}", Class1.myString);
    }

    static void Main(string[] args)
    {
        string myString = "String defined in Main()";
        Class1.myString = "Global string";
        Write();
        Console.WriteLine("\nNow in Main()");
        Console.WriteLine("Local myString = {0}", myString);
        Console.WriteLine("Global myString = {0}", Class1.myString);
    }
}
```

结果如图 6-5 所示。



图　6-5

这里添加了另一个变量 myString，这次进一步加深了代码中的名称层次。这个变量定义如下：

```
static string myString;
```

注意这里也需要 static 关键字。在这种形式的控制台应用程序中，必须使用 static 或 const 关键字，来定义这种形式的全局变量。如果要修改全局变量的值，就需要使用 static，因为 const 禁止修改变量的值。

为了区分这个变量和 Main() 与 Write()中同名的局部变量，必须用一个完整限定的名称为变量名分类，参见第 3 章。这里把全局变量称为 Class1.myString。注意在全局变量和局部变量同名时，这是必需的。如果没有局部 myString 变量，就可以使用 myString 表示全局变量，而不需要使用 Class1.myString。如果局部变量和全局变量同名，全局变量就会被隐藏。

全局变量的值在 Main()中设置：

```
Class1.myString = "Global string";
```

在 Write()中访问：

```
Console.WriteLine("Global myString = {0}", Class1.myString);
```

现在，你可能要问，为什么不能使用这个技术通过函数交换数据，而要使用前面介绍的参数来交换数据？有时，这确实是一种交换数据的首选方式，但在许多情况下不应使用这种方式。是否使用全局变量取决于函数的位置。使用全局变量的问题是，它们一般不适合于“一般用途”的函数，这些函数能处理我们所提供的数据，而不仅限于处理特定全局变量的数据。详见后面的内容。

6.2.1 其他结构中的变量作用域

在继续之前，应先注意一下上一节的一个要点总结了上述内容和超出了函数之间的变量作用域。前面说过，变量的作用域包含定义它们的代码块和直接嵌套在其中的代码块。这也可以应用到其他代码块上，例如分支和循环结构的代码块。考虑下面的代码：

```
int i;
for (i = 0; i < 10; i++)
{
    string text = "Line " + Convert.ToString(i);
    Console.WriteLine("{0}", text);
}
Console.WriteLine("Last text output in loop: {0}", text);
```

字符串变量 text 是 for 循环的局部变量，这段代码不能编译，因为在该循环的外部调用的 Console.WriteLine()试图使用该变量 text，这超出了循环的范围。修改代码，如下所示：

```
int i;
string text;
for (i = 0; i < 10; i++)
{
    text = "Line " + Convert.ToString(i);
    Console.WriteLine("{0}", text);
}
Console.WriteLine("Last text output in loop: {0}", text);
```

这段代码也会失败，原因是变量必须在使用前声明和初始化，而 text 是在 for 循环中初始化的。赋给 text 的值在循环块退出时就丢失了。但是还可以进行如下修改：

```
int i;
string text = "";
for (i = 0; i < 10; i++)
{
    text = "Line " + Convert.ToString(i);
    Console.WriteLine("{0}", text);
}
Console.WriteLine("Last text output in loop: {0}", text);
```

这次 text 是在循环外部初始化的，可以访问它的值。这段简单代码的结果如图 6-6 所示。



图　6-6

在循环中最后赋给 text 的值可以在循环外部访问。

可以看出，这个主题的内容需要花一点时间来掌握。在前面的范例中，循环之前赋给 text 空字符串，而在循环之后的代码中，该 text 就不会是空字符串了，其原因不是立即可以看出。

这种情况的解释涉及到分配给 text 变量的内存空间，实际上任何变量都是这样。只声明一个简单的变量类型，并不会引起其他的变化。只有在给变量赋值后，这个值才占用一块内存空间。如果这种占据内存空间的行为在循环中发生，该值实际上定义为一个局部值，超出了循环的范围。即使变量本身没有本地化到循环上，循环所包含的值也本地化到该循环上。但是，在循环外部赋值可以确保该值是主体代码的局部值，仍在循环的范围内。这意味着变量在退出主体代码块之前是没有超出范围的，所以可以在循环外部访问它的值。

幸运的是，C#编译器就检测变量作用域的问题，它生成的响应错误信息可以帮助我们理解变量作用域的问题。

最后一个要注意的问题是，应采用“最好的惯例”。一般情况下，最好在声明和初始化所有的变量后，再在代码块中使用它们。一个例外是把循环变量声明为循环块的一部分，例如：

```
for (int i = 0; i < 10; i++)
{
    ...
}
```

其中 i 本地化了循环代码块中，但这是可以的，因为我们很少需要在外部代码中访问这个计数器。

6.2.2　参数和返回值与全局数据

本节详细介绍如何利用函数，通过全局数据以及参数和返回值，来交换数据。为了更新，

将讨论代码之间的区别，例如：

```
class Class1
{
    static void showDouble(ref int val)
    {
        val *= 2;
        Console.WriteLine("val doubled = {0}", val);
    }

    static void Main(string[] args)
    {
        int val = 5;
        Console.WriteLine("val = {0}", val);
        showDouble(ref val);
        Console.WriteLine("val = {0}", val);
    }
}
```

注意：

这段代码略微不同于本章前面的代码，在前面的范例中，在 Main()中使用了变量名 myNumber，这说明了局部变量可以有相同的名称，且不会相互干涉。这里列出的两个代码范例比较类似，以便我们集中精力研究它们的区别，而无需担心变量名。

和：

```
class Class1
{
    static int val;

    static void showDouble()
    {
        val *= 2;
        Console.WriteLine("val doubled = {0}", val);
    }

    static void Main(string[] args)
    {
        val = 5;
        Console.WriteLine("val = {0}", val);
        showDouble();
        Console.WriteLine("val = {0}", val);
    }
}
```

这两个函数的结果是相同的。

现在，使用哪种方法并没有什么硬性而快速的规则，这两种方法都是有效的。但是，有一些规则是要考虑的。

首先，在第一次讨论这个问题时，使用全局值的 showDouble()版本只使用全局变量 val。为了使用这个版本，必须使用这个全局变量。这会对该函数的多样性有轻微的限制，如果要存储结果，就必须继续把这个全局变量值复制到其他变量中。另外，全局数据可以在应用程序的其他地方由代码修改，这会导致预料不到的结果(即使我们没有认识到这一点，值也是可以改变的)。

但是，损失了多样性常常是有好处的。我们常常希望把一个函数只用于一个目的，使用一个全局数据存储减少在函数调用中犯错的可能性，也许是把它传递给错误的变量。

当然，也可以说，这种简化实际上使代码更难理解。明确指定参数可以一眼看出什么发生了改变。如果看看一个读取 myFunction(val1, out val2)的调用，就会知道 val1 和 val2 都是要考虑的重要变量，在函数执行结束后，val2 就会被赋予一个新值。反之，如果这个函数不带参数，就不能做出它处理了什么数据的任何假设。

最后，记住并不总是能使用全局数据。本书的后面将介绍在不同的文件中编写的代码，以及不同的名称空间的代码如何通过函数彼此通信。像这样的情况，代码常常要分开编写，以致于无法选择全局存储位置。

总之，可以自由选择使用哪种技术来交换数据。一般情况下，最好使用参数，而不使用全局数据，但有时使用全局数据更合适，使用这个技术并没有错。

6.3 Main()函数

前面介绍了创建和使用函数时涉及的大多数简单技术，下面详细论述 Main()函数。

前面，Main()是 C#应用程序的入口点，执行这个函数就是执行应用程序。这个函数有一个参数 string[] args，但还没有说明这个参数的含义。本节将介绍该参数，以及如何使用它。

注意：

Main 函数可以使用四种签名：

```
static void Main()
static void Main(string[] args)
static int Main()
static int Main(string[] args)
```

如果需要，可以忽略这里讨论的 args。直到现在还在使用这个参数的原因，就是在 VS 中创建控制台应用程序时自动生成的 Main()版本。

上面的第三、四个版本返回一个 int 值，它们可以用于表示应用程序如何终止，通常用作一种错误提示(但这不是强制的)，一般情况下，返回 0 反映了“正常”的终止(即应用程序执行完毕，并安全地终止)。

Main()的参数 args 是从应用程序的外部接受信息的方法，这些信息在运行期间指定，其形式是命令行参数。

前面已经遇到了命令行参数，在从命令行上执行应用程序时，通常可以直接指定信息，如在执行应用程序时加载一个文件。例如，考虑 Windows 中的 Notepad 应用程序。通过在命令行窗口中输入 notepad，或者在 Windows 的 Start 菜单中选择 Run 选项时，在打开的窗口中输入 notepad，可以运行该应用程序。也可以在这些地方输入 notepad "myfile.txt"，结果是 Notepad 在运行时将加载文件 myfile.txt，如果该文件不存在，Notepad 也会创建该文件。这里 myfile.txt 是一个命令行参数。通过利用 args 参数，可以编写以相同的方式工作的控制台应用程序。

在执行控制台应用程序时，指定的任何命令行参数都放在这个 args 数组中，接着可以根据需要在应用程序中使用这些参数。

下面用一个范例来说明。

试试看：命令行参数

(1) 在目录 C:\BegCSharp\Chapter6 下创建一个新控制台应用程序 Ch06Ex04。

(2) 把下述代码添加到 Class1.cs 中：

```
class Class1
{
    static void Main(string[] args)
    {
        Console.WriteLine("{0} command line arguments were specified:",
                          args.Length);
        foreach (string arg in args)
            Console.WriteLine(arg);
    }
}
```

(3) 打开工程的属性页面(在 Solution Explorer 窗口中右击 Ch06Ex04 工程名称，选择 Properties)。

(4) 选择 Configuration Properties | Debugging 页面，在 Command Line Arguments 设置中添加希望的命令行参数如图 6-7 所示。

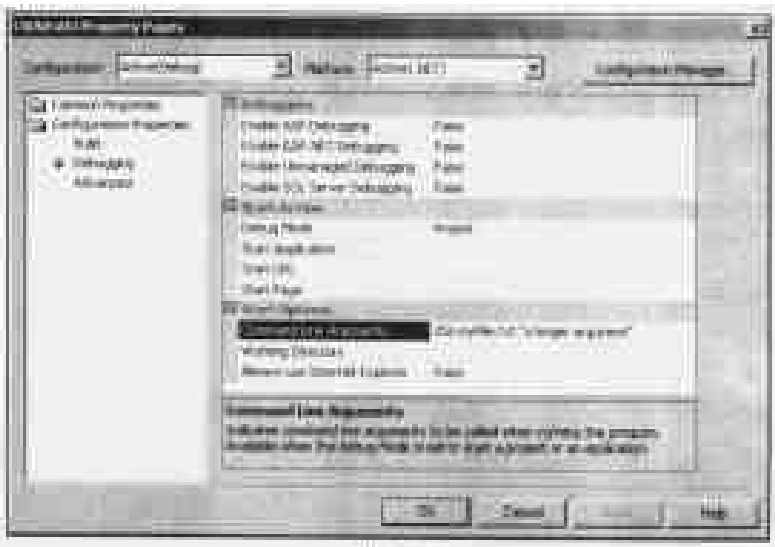


图 6-7

(5) 运行应用程序，如图 6-8 所示。



图　6-8

代码的说明

这里使用的代码非常简单：

```
Console.WriteLine("{0} command line arguments were specified:",
                  args.Length);
foreach (string arg in args)
    Console.WriteLine(arg);
```

使用 args 参数与使用其他字符串数组类似。我们没有对参数进行任何异样的操作，只是把指定的信息写到屏幕上。

在本范例中，通过 VS 中的 Project Properties 对话框提供参数，这是一种很方便的方式，只要在 VS 中运行应用程序，就可以使用相同的命令行参数，无需每次都在命令行提示窗口中输入它们。在工程输出所在的目录(C:\BegCSharp\Chapter6\Ch06Ex04\bin\Debug)下打开命令行窗口，输入下述内容，也可以得到相同的结果：

```
Ch06Ex04 256 myFile.txt "a longer argument"
```

注意，每个参数都用空格分隔开，如果参数比较长，也可以用双引号把参数括起来(如果参数包含空格，这就是必须的，这样才不会把这个参数解释为多个参数)。

6.4　结构函数

第 5 章介绍了结构类型，它可在一个地方存储多个数据元素，结构可以做的工作远不止此。一个重要的功能就是结构可以包含函数和数据。这初看起来很奇怪，但实际上是非常有用的。

例如，考虑下面的结构：

```
struct customerName
{
    public string firstName, lastName;
}
```

如果变量的类型是 customerName，并且要在控制台上输出一个完整的名称，就必须从其组件部分建立该名称。例如，CustomerName 变量 customer 可以使用下述语法：

```
customerName myCustomer;
myCustomer.firstName = "John";
```

```
myCustomer.lastName = "Franklin";
Console.WriteLine("{0} {1}", customer.firstName, customer.lastName);
```

把函数添加到结构中，就可以集中处理常见任务，简化这个过程。可以把合适的函数添加到结构类型中，如下所示：

```
struct customerName
{
    public string firstName, lastName;

    public string Name ()
    {
        return firstName + " " + lastName;
    }
}
```

看起来这很类似于本章前面的其他函数，但没有使用 static 修饰符。原因本书在后面论及，现在知道该关键字不是结构函数所必须的即可。这个函数的用法如下所示：

```
customerName myCustomer;
myCustomer.firstName = "John";
myCustomer.lastName = "Franklin";
Console.WriteLine(customer.Name());
```

这个语法比前面的语法简单得多，也更容易理解。

注意，Name()函数可以直接访问 firstName 和 lastName 结构成员，在 customerName 结构中，它们可以看作是全局成员。

6.5 函数的重载

本章的前面介绍了在调用函数时，必须匹配函数的签名。这表示，需要让多个函数操作不同类型的变量。运算符重载认为这是一个问题，并提供了创建同名的多个函数的能力，但这些函数可使用不同的参数类型。

例如，前面使用了下述代码，其中包含一个函数 MaxValue()：

```
class Class1
{
    static int MaxValue(int[] intArray)
    {
        int maxVal = intArray[0];
        for (int i = 1; i < intArray.Length; i++)
        {
            if (intArray[i] > maxVal)
                maxVal = intArray[i];
        }
```

```
        return maxVal;
    }

    static void Main(string[] args)
    {
        int[] myArray = {1, 8, 3, 6, 2, 5, 9, 3, 0, 2};
        int maxVal = MaxValue(myArray);
        Console.WriteLine("The maximum value in myArray is {0}", maxVal);
    }
}
```

这个函数只能用于处理 int 数组，现在来为不同的参数类型提供不同名称的函数，可以把上述函数重新命名为 IntArrayMaxValue()，添加函数 DoubleArrayMaxValue()处理其他类型。另外，还可以在代码中添加如下函数：

```
...
static double MaxValue(double[] doubleArray)
{
    double maxVal = doubleArray[0];
    for (int i = 1; i < doubleArray.Length; i++)
    {
        if (doubleArray[i] > maxVal)
            maxVal = doubleArray[i];
    }
    return maxVal;
}
...
```

这里的区别是使用了 double 值。函数名称 MaxValue()是相同的，但其签名是不同的。用相同的名称和签名定义两个函数是错误的，但因为这两个函数有不同的签名，所以是可行的。

现在有两个版本的 MaxValue()，它们的参数是 int 和 double 数组，分别返回一个 int 或 double 最大值。

这种代码的优点是不必明确指定要使用哪个函数。只需提供一个数组参数，就可以根据使用的参数类型执行相应的函数。

此时，应注意 VS 的另一个智能化功能。如果在应用程序中有上述两个函数，而且要在 Main()中输入函数的名称，VS 就可以显示出函数的可用重载函数。如果输入下面的代码：

```
double result = MaxValue(
```

VS 提供了 MaxValue()两个版本的信息，使用上下箭头键可以在它们之间滚动如图 6-9 所示。

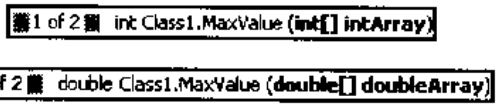


图　6-9

在重载函数时，应包括函数签名的所有方面。例如有两个不同的函数，它们分别带有值参数和引用参数：

```
static void showDouble(ref int val)
{
    ...
}

static void showDouble(int val)
{
    ...
}
```

选择使用哪个版本纯粹是根据函数调用是否包含 ref 关键字来确定。下面的代码是调用引用版本：

```
showDouble(ref val);
```

下面的代码是调用值版本：

```
showDouble(val);
```

另外，函数还可以根据参数的个数等来区分。

6.6 委托

委托是一种可以把引用存储为函数的类型。这听起来相当棘手，但其机制是非常简单的。委托最重要的用途在本书介绍到事件和事件处理时才能解释清楚，但这里也将介绍有关委托的许多内容。在本书的后面使用它们时，这些内容有助于理解一些比较复杂的问题。

委托的声明非常类似于函数，但不带函数体，且要使用 delegate 关键字。委托的声明指定了一个函数签名，其中包含一个返回类型和参数列表。在定义了委托后，就可以声明该委托类型的变量。接着初始化这个变量为与委托有相同签名的函数引用。之后，就可以使用委托变量调用这个函数，就像该变量是一个函数一样。

有了一个引用函数的变量后，还可以执行不能用其他方式完成的操作。例如，可以把委托变量作为参数传递给一个函数，这样，该函数就可以使用委托调用它引用的任何函数，而且在运行之前无需知道调用的是哪个函数。

下面看一个范例。

试试看：使用委托来调用函数

(1) 在目录 C:\BegCSharp\Chapter6 下创建一个新控制台应用程序 Ch06Ex05。

(2) 把下述代码添加到 Class1.cs 中：

```
class Class1
```

```
{
    delegate double processDelegate(double param1, double param2);

    static double Multiply(double param1, double param2)
    {
        return param1 * param2;
    }

    static double Divide(double param1, double param2)
    {
        return param1 / param2;
    }

    static void Main(string[] args)
    {
        processDelegate process;
        Console.WriteLine("Enter 2 numbers separated with a comma:");
        string input = Console.ReadLine();
        int commaPos = input.IndexOf(',');
        double param1 = Convert.ToDouble(input.Substring(0, commaPos));
        double param2 = Convert.ToDouble(input.Substring(commaPos + 1,
                                                input.Length - commaPos - 1));
        Console.WriteLine("Enter M to multiply or D to divide:");
        input = Console.ReadLine();
        if (input == "M")
            process = new processDelegate(Multiply);
        else
            process = new processDelegate(Divide);
        Console.WriteLine("Result: {0}", process(param1, param2));
    }
}
```

(3) 执行代码，如图 6-10 所示。

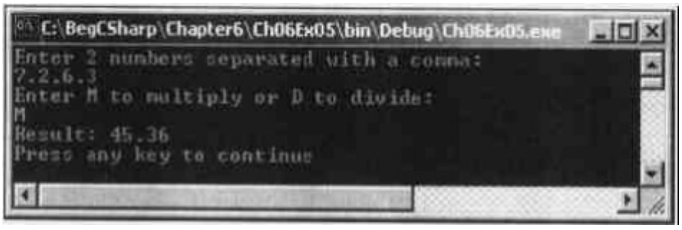


图 6-10

代码的说明

这段代码定义了一个委托 processDelgate，其签名匹配于两个函数(Multiply() 和 Divide())的签名。委托的定义如下所示：

```
delegate double processDelegate(double param1, double param2);
```

delegate 关键字指定该定义是用于委托的，而不是用于函数的(该定义所在的位置与函数定义相同)。接着，用一个签名指定 double 返回类型和两个 double 参数。实际使用的名称可以是任意的，所以任意调用委托类型和参数名。这里委托名是 processDelgate，double 参数名是 param1 和 param2。

Main()中的代码首先使用新的委托类型声明一个变量：

```
static void Main(string[] args)
{
    processDelegate process;
```

接着用一些相当标准的 C#代码请求由逗号分开的两个数字，并把这些数字放在两个 double 变量中：

```
Console.WriteLine("Enter 2 numbers separated with a comma:");
string input = Console.ReadLine();
int commaPos = input.IndexOf(',');
double param1 = Convert.ToDouble(input.Substring(0, commaPos));
double param2 = Convert.ToDouble(input.Substring(commaPos + 1,
                                                 input.Length - commaPos - 1));
```

注意：

为了说明问题，这里没有包含用户输入的有效性检查。如果这些是“现实中的”代码，就应花更多的时间来确定在局部变量 param1 和 param2 中得到有效的值。

接着，询问用户是要相乘，还是相除这两个数字：

```
Console.WriteLine("Enter M to multiply or D to divide:");
input = Console.ReadLine();
```

根据用户的选择，初始化 process 委托变量：

```
if (input == "M")
    process = new processDelegate(Multiply);
else
    process = new processDelegate(Divide);
```

要把一个函数引用赋给委托变量，需要使用略显古怪的语法。这个过程比较类似于给数组赋值，必须使用 new 关键字创建一个新委托。在这个关键字的后面，指定委托类型，提供一个引用函数的参数，该函数是 Console.WriteLine()。注意这个参数与委托类型或目标函数的参数不匹配，这是委托赋值的一个独特语法，参数是要使用的函数名，且不加括号。

最后，使用该委托调用所选的函数。无论委托引用的是什么函数，该语法都是有效的：

```
    Console.WriteLine("Result: {0}", process(param1, param2));
}
```

这里把委托变量看作一个函数名。但与函数不同，我们还可以对这个变量执行更多的操作，例如通过参数把它传递给一个函数，这个函数的一个简单范例如下：

```
static void executeFunction(processDelegate process)
{
    process(2.2, 3.3);
}
```

就像选择一个要使用的“插件”一样，把它们传递给函数委托，就可以控制函数的执行。例如，一个函数要对字符串数组按照字母进行排序。对列表排序有几个不同的方法，它们的性能取决于要排序的列表特性。使用委托可以把一个排序算法函数委托传递给排序函数，指定要使用的方法。

委托有许多用途，但如上所述，它们的大多数常见用途主要与事件处理有关，详见第12章。

6.7 小结

本章相当全面地介绍了C#代码中函数的使用。函数提供的许多其他特性(特别是委托)比较抽象，我们将在面向对象编程中讨论它们，面向对象编程将在不久后讨论。

本章的主要内容总结如下：

- 在控制台应用程序中定义和使用函数
- 通过返回值和参数，利用函数交换数据
- 参数数组
- 按引用或按值传递值
- 为其他返回值指定输出参数
- 变量作用域的概念
- Main()函数的细节，包括命令行参数的用法
- 在结构类型中使用函数
- 函数的重载
- 委托

6.8 练习

(1) 下面两个函数有错误，它们是什么？

```
static bool Write()
{
    Console.WriteLine("Text output from function.");
}
```

```
static void myFunction(string label, params int[] args, bool showLabel)
{
    if (showLabel)
        Console.WriteLine(label);
    foreach (int i in args)
        Console.WriteLine("{0}", i);
}
```

(2) 编写一个应用程序，该程序使用两个命令行参数，分别把值放在一个字符串和一个整型变量中，然后显示这些值。

(3) 创建一个委托，在请求用户输入时，使用它模拟 Console.ReadLine()函数。

(4) 修改下面的结构，使之包含一个返回订单总价格的函数。

```
struct order
{
    public string itemName;
    public int    unitCount;
    public double unitCost;
}
```

(5) 在 order 结构中添加另一个函数，该结构返回一个格式化的字符串，以合适的值替换用尖括号括起来的斜体条目。

第7章　调试和错误处理

本书到目前为止，介绍了在 C#中进行简单编程的所有基础知识。在讨论本书后面章结的面向对象编程之前，先看看 C#代码中的调试和错误处理问题。

代码中有错误是在所难免的。无论程序员有多么优秀，程序总是会有一些问题，好一点的程序员会找出其中一部分错误，并更正它们。当然，这些都是不影响应用程序执行的小问题，例如按钮上的拼写错误等，它们也可能是导致应用程序完全失败的大错误(通常称为致命错误)，致命错误包括妨碍代码编译的简单错误(语法错误)，但更多的错误比较棘手，它们只在运行期间发生。另外，错误可能会更微妙。如果遗漏了一个请求的字段，应用程序就不能给数据库添加一个记录，或者在其他有限制的环境中把错误的数据添加到记录中，在应用程序逻辑在某些方面有瑕疵时，就会产生这样的错误，这类错误称为语义错误(也称为逻辑错误)。

你可能听说过，当应用程序的用户抱怨说程序没有正常工作时，就发生了比较微妙的错误。此时需要跟踪代码，试着确定发生了什么问题，应如何修改代码，使之按希望的那样工作。

在这样的情况下，VS 的调试功能就可以大显身手了。本章的第一部分就介绍一些调试技巧，并把它们应用到常见的一些问题上。

另外，还将讨论 C#中的错误处理技术。利用它们，可以对可能发生错误的地方采取预防措施，并编写有足够弹性的代码来处理可能会致命的错误。这些是 C#语言的一部分，而不是 VS 的调试功能，但 VS 也提供了一些工具来帮助我们进行错误处理。

7.1　Visual Studio 中的调试

在调试模式下运行程序时，并不仅仅是运行编写好的代码。调试程序包含了应用程序的符号信息，例如 VS 知道执行每行代码时发生了什么。符号信息意味着跟踪(例如)未编译代码中使用的变量名，这样，它们就可以匹配已编译的机器码应用程序中现有的值，而机器码程序不包含人类易读的信息。这些信息包含在.pdb 文件中，这些文件位于计算机的 Debug 目录下。它们可以执行许多有用的操作，包括(但不限于这些操作)：

- 给 VS 输出调试信息
- 在执行应用程序的过程中，查看(并编辑)变量的值
- 暂停程序、重新启动程序
- 在代码的某个位置自动暂停程序的执行
- 一次执行程序中的一行代码
- 在应用程序的执行过程中，监视变量内容的修改
- 在运行期间修改变量的内容
- 测试函数的测试调用

本节介绍上述技巧，以及如何使用它们标识没有按希望执行的那些代码，并修改它们，这个过程一般称为调试。

按照这些技术的使用方法把它们分为两个部分。一般情况下，可以先中断程序的执行，再进行调试，或者注上标记，以便以后进行分析。 在 VS 术语中，应用程序可以处于运行状态，也可以处于中断模式，即暂停正常的执行。下面首先介绍非中断模式(运行期间或正常执行)。

7.1.1　非中断(正常)模式下的调试

本书常常使用的一个命令是 Console.WriteLine()函数，它可以把文本输出到控制台上。在开发应用程序时，这个函数可以方便地获得操作的额外反馈，例如：

```
Console.WriteLine("MyFunc() Function about to be called.");
MyFunc ("Do something.");
Console.WriteLine("MyFunc() Function execution completed.");
```

这段代码说明了如何给出 MyFunc()函数的额外信息。

这么做完全正确，但控制台的输出结果会比较混乱。作为一种替代方法，可以把文本输出到另一个位置上——VS 中的 Output 窗口。

第 2 章简要介绍了 Output 窗口，在默认情况下，该窗口位于 VS 开发环境的底部，与 Task List 窗口共享空间。第 2 章介绍了这个窗口如何显示与代码的编译和执行相关的信息，包括在编译过程中遇到的错误等，还可以通过将自定义信息直接写到窗口中，来使用这个窗口显示自定义信息。在屏幕上，该窗口如图 7-1 所示。

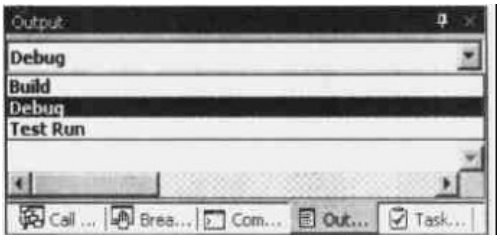


图　7-1

注意：

这个窗口有 3 种模式，使用其中的下拉列表框就可以选择这些模式。可以在 Build, Debug 和 Test Run 模式之间切换。Build 和 Debug 模式分别显示编译和运行期间的信息。本节提到“写入 Output 窗口”时，实际上是指“写入 Output 窗口的 Debug 模式视图”。

另外，还可以创建一个日志文件，在运行应用程序时，会把信息添加到该日志文件中。把信息写入日志文件所使用的技巧与把文本写到 Output 窗口上所使用的技巧相同，但需要理解如何从 C#应用程序中访问文件系统。现在，把这个功能放在书后的附录中，因为即使不太了解文件访问技巧，也可以把信息写入日志文件。

输出调试信息

在运行期间把文本写入 Output 窗口是非常简单的。只要用需要的调用替代 Console.WriteLine()调用，就可以把文本写到希望的地方。此时可以使用两个命令：

```
Debug.WriteLine()
Trace.WriteLine()
```

这两个命令函数的使用方法几乎完全相同，但有一个关键的区别。第一个命令仅在调试模式下运行，而第二个命令则还可用于发布程序。实际上，Debug.WriteLine()命令甚至不能编译为可发布的程序，这肯定有其优点(首当其冲的是，编译好的代码文件比较小)。实际上，一个源文件可以创建出两个版本的应用程序。调试版本显示所有的额外诊断信息，而发布版本没有这个功能，也不给用户显示信息，否则，用户会觉得这非常讨厌。

注意这两个函数的用法与 Console.WriteLine()是不同的。它们使用的惟一字符串参数是为了消息的输出，而不需要使用{X}语法插入变量值。这意味着必须使用+运算符在字符串中插入变量值。它们还可以有第二个字符串参数，用于显示输出文本的类别，这样，如果应用程序的不同地方输出了类似的消息，我们就可以马上确定出 Output 窗口中显示的是哪个输出信息。

这些函数的一般输出如下所示：

```
<category>: <message>
```

例如，下面的语句把"MyFunc"作为可选的类别参数：

```
Debug.WriteLine("Added 1 to i", "MyFunc");
```

其结果为：

```
MyFunc: Added 1 to i
```

下面看一个范例。

试试看：把文本输出到 Output 窗口

(1) 在目录 C:\BegCSharp\Chapter7 下创建一个新的控制台应用程序 Ch07Ex01。

(2) 修改代码，如下所示：

```
using System;
using System.Diagnostics;

namespace Ch07Ex01
{
    /// <summary>
    /// Summary description for Class1.
    /// </summary>
    class Class1
    {
        static void Main(string[] args)
        {
            int[] testArray = {4, 7, 4, 2, 7, 3, 7, 8, 3, 9, 1, 9};
            int[] maxValIndices;
            int maxVal = Maxima(testArray, out maxValIndices);
```

```
    Console.WriteLine("Maximum value {0} found at element indices:",
                      maxVal);
    foreach (int index in maxValIndices)
    {
        Console.WriteLine(index);
    }
}

static int Maxima(int[] integers, out int[] indices)
{
    Debug.WriteLine("Maximum value search started.");
    indices = new int[1];
    int maxVal = integers[0];
    indices[0] = 0;
    int count = 1;
    Debug.WriteLine("Maximum value initialized to " + maxVal +

                    ", at element index 0.");
    for (int i = 1; i < integers.Length; i++)
    {
        Debug.WriteLine("Now looking at element at index " + i + ".");
        if (integers[i] > maxVal)
        {
            maxVal = integers[i];
            count = 1;
            indices = new int[1];
            indices[0] = i;
            Debug.WriteLine("New maximum found. New value is " + maxVal +
                            ", at element index " + i + ".");
        }
        else
        {
            if (integers[i] == maxVal)
            {
                count++;
                int[] oldIndices = indices;
                indices = new int[count];
                oldIndices.CopyTo(indices, 0);
                indices[count - 1] = i;
                Debug.WriteLine("Duplicate maximum found at element index " +
                                i + ".");
            }
        }
    }
    Trace.WriteLine("Maximum value " + maxVal + " found, with " + count +
```

```
                                " occurrences.");
            Debug.WriteLine("Maximum value search completed.");
            return maxVal;
        }
    }
}
```

(3) 在 Debug 模式下执行代码，如图 7-2 所示。



图　7-2

(4) 中断应用程序的执行，查看 Output 窗口中的内容(在调试模式下)：

```
'DefaultDomain': Loaded 'c:\winnt\microsoft.net\framework\v1.0.2901\mscorlib.dll', No symbols loaded.
'Ch07Ex01': Loaded 'C:\BegCSharp\Chapter7\Ch07Ex01\bin\Debug\Ch07Ex01.exe',
Symbols loaded.
'Ch07Ex01.exe': Loaded 'c:\winnt\assembly\gac\system\1.0.2411.0__b77a5c561934e089\system.dll', No
symbols loaded.
'Ch07Ex01.exe': Loaded
'c:\winnt\assembly\gac\system.xml\1.0.2411.0__b77a5c561934e089\system.xml.dll', No symbols loaded.
Maximum value search started.
Maximum value initialized to 4, at element index 0.
Now looking at element at index 1.
New maximum found. New value is 7, at element index 1.
Now looking at element at index 2.
Now looking at element at index 3.
Now looking at element at index 4.
Duplicate maximum found at element index 4.
Now looking at element at index 5.
Now looking at element at index 6.
Duplicate maximum found at element index 6.
Now looking at element at index 7.
New maximum found. New value is 8, at element index 7.
Now looking at element at index 8.
Now looking at element at index 9.
New maximum found. New value is 9, at element index 9.
Now looking at element at index 10.
Now looking at element at index 11.
Duplicate maximum found at element index 11.
Maximum value 9 found, with 2 occurrences.
Maximum value search completed.
The program '[2056] Ch07Ex01.exe' has exited with code 0 (0x0).
```

(5) 使用标 S+andard 准工具栏上的下拉列表框，切换为 Release 模式，如图 7-3 所示。



图　7-3

(6) 再次运行程序，这次是在 Release 模式下运行，并在执行中止时，再查看一下 Output 窗口：

```
'DefaultDomain': Loaded 'c:\winnt\microsoft.net\framework\v1.0.2901\mscorlib.dll', No symbols loaded.
'Ch07Ex01': Loaded 'C:\BegCSharp\Chapter7\Ch07Ex01\bin\Release\Ch07Ex01.exe', No symbols loaded.
'Ch07Ex01.exe': Loaded 'c:\winnt\assembly\gac\system\1.0.2411.0__b77a5c561934e089\system.dll', No
symbols loaded.
'Ch07Ex01.exe': Loaded
'c:\winnt\assembly\gac\system.xml\1.0.2411.0__b77a5c561934e089\system.xml.dll', No
symbols loaded.
Maximum value 9 found, with 2 occurrences.
The program '[1792] Ch07Ex01.exe' has exited with code 0 (0x0).
```

代码的说明

这个应用程序是第 6 章中一个范例的扩展版本，它使用一个函数计算整数数组中的最大值。这个版本也返回数组中，其中包括最大值的下标，以便调用代码能处理这些元素。下面详细分析一下代码。

首先，注意在代码的开头使用了一个额外的 using 语句：

```
using System.Diagnostics;
```

这简化了本例前面讨论的函数访问，因为它们包含在 System.Diagnostics 名称空间中。没有这个 using 语句，如下所示：

```
Debug.WriteLine("Bananas");
```

就需要进一步限定，重新编写这行语句，如下所示：

```
System.Diagnostics.Debug.WriteLine("Bananas");
```

Using 语句使代码更简单，缩短了代码的长度。

Main()中的代码仅初始化一个测试用的整数数组 testArray，并声明了另一个整数数组 maxValIndices，以存储 Maxima()的下标结果(执行计算操作的函数)，接着调用这个函数。函数返回后，代码就输出了结果。

Maxima()略为复杂一些，但没有使用前面介绍的那么多的代码。在数组中进行搜索的方式与第 6 章的 MaxVal()函数类似，但要用一个记录存储最大值的下标。

也许这段代码中最重要的一点(不是输出调试信息的那些代码)是，用于跟踪下标的方法。Maxima()并没有返回一个足以存储源数组中每个下标的数组(需要与源数组有相同的维数)，而

是返回一个正好能容纳搜索到的下标的数组。这可以在搜索过程中连续重建不同长度的数组来实现。这是必要的，因为数组一旦创建好，就不能重新设置长度。

开始搜索时，假定源数组中的第一个元素(称为本地 integers)就是最大值，且数组中只有一个最大值。因此可以为 maxVal(函数的返回值，即搜索到的最大值)和 indices(out 参数数组，存储发现的最大值的下标)设置值。MaxVal 被赋予的值是 integers 中第一个元素的值，indices 被赋予一个值 0，即数组中第一个元素的下标。在变量 count 中存储所发现的最大值的个数，以跟踪 indices 数组。

函数的主体是一个循环，它迭代 integers 数组中的各个值，但忽略第一个值，因为它已经处理过了。每个值都与 maxVal 的当前值进行比较，如果 maxVal 更大，就忽略该值。如果当前处理的值比 maxVal 大，就修改 maxVal 和 indices，反映这个情况。如果当前处理的值与 maxVal 相等，就递增 count，用一个新数组替代 indices。这个新数组就是一个比旧 indices 数组更大的元素，包含搜索到的新下标。

最后一个功能的代码如下所示：

```
if (integers[i] == maxVal)
{
    count++;
    int[] oldIndices = indices;
    indices = new int[count];
    oldIndices.CopyTo(indices, 0);
    indices[count - 1] = i;
    Debug.WriteLine("Duplicate maximum found at element index " +
                    i + ".");
}
```

注意这是把旧 indices 数组备份到 if 代码块的 oldIndices 整型数组中。还要注意使用 <array>.CopyTo() 函数把 oldIndices 中的值复制到新的 indices 数组中。这个函数只提取一个目标数组和一个用于复制第一个元素的下标，并把所有的值都粘贴到目标数组中。

在代码中，各个文本部分都是使用 Debug.WriteLine() 和 Trace.WriteLine()函数输出的，在调试模式下运行时，其最终结果是一个完整的记录，它记述了在循环中计算出结果所采取的步骤。在发布模式下，仅能看到计算的最终结果，因为没有执行 Debug.WriteLine()函数。

除了这些 WriteLine()函数外，还需要注意其他一些问题。首先是 Console.Write()的等价函数：

- Debug.Write()
- Trace.Write()

这两个函数使用与 WriteLine()函数相同的语法(一个或两个参数，即一个消息和可选的类别)，但它们是有区别的，因为它们没有添加行尾字符。

还有下述命令：

- Debug.WriteLineIf()
- Trace.WriteLineIf()
- Debug.WriteIf()

- Trace.WriteIf()

这些方法的参数都与非 if 方法的参数相同，但增加了一个必选参数，且该参数放在列表参数的最前面。这个参数的值为布尔值(或者计算结果为布尔值的表达式)，如果这个值为 true，函数的结果就是只写文本。使用这些函数可以有条件地把文本输出到 Output 窗口中。

例如，在某些情况下，只需要输出调试信息，所以代码中有许多 Debug.WriteLineIf()语句，它们都取决于具体的条件。如果没有这个条件，就不显示它们，以防 Output 窗口显示多余的信息。

7.1.2　中断模式下的调试

调试技术的剩余内容是在中断模式下工作。这个模式可以用几种方式进入，这些方式都可以暂停程序的执行。本节首先说明如何进入中断模式，接着介绍进入中断模式后，可以得到什么好处。

1. 进入中断模式

进入中断模式的最简单方式是在运行应用程序时，单击 VS 中的暂停按钮。这个暂停按钮在 Debug 工具栏上，应把该按钮添加到 VS 默认情况下显示的工具栏中。为此，右击工具栏区域，选择 Debug 工具栏，如图 7-4 所示。



图　7-4

这个工具栏如图 7-5 所示。



图　7-5

在这个工具栏上，前四个按钮可以手工控制中断。在图 7-5 上，其中的 3 个按钮显示为灰色，因为在程序没有运行时，它们是不能工作的。还有一个按钮 Start 是可以使用的，这个按钮与标准工具栏上的 Start 按钮相同。在后面的章节需要其他的按钮时，再介绍它们。

运行一个应用程序时，工具栏就将如图 7-6 所示。



图　7-6

现在，以前显示为灰色的 3 个按钮就可以使用了。它们可以：

- 暂停应用程序的执行，进入中断模式
- 完全停止应用程序的执行(不进入中断模式，而是退出应用程序)
- 重新启动应用程序

暂停应用程序是进入中断模式的最简单方式，但这并不能更好地控制停止运行的位置。我

们可能会很自然地停止运行应用程序，例如要求用户输入信息。还可以在长时间的操作或循环过程中进入中断模式，但停止的位置可能相当随意。

一般情况下，最好使用断点。

断点

断点是源代码中自动进入中断模式的一个标记。它们可以配置为：

- 在遇到断点时，立即进入中断模式
- 在遇到断点时，如果布尔表达式的值为 true，就进入中断模式
- 遇到某断点一定的次数后，进入中断模式
- 在遇到断点时就进入中断模式，且自从上次遇到断点以来变量的值发生了变化

注意上述仅能用于调试程序。如果编译发布程序，就会忽略所有的断点。

添加断点有三种方法。要添加一个断点，当遇到该断点所在的代码行时，就中断执行，可以单击该代码行左边的灰色区域，或者右击该代码行，选择 Insert Breakpoint 菜单项，如图 7-7 所示。

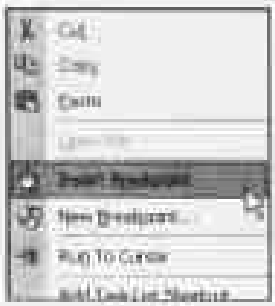
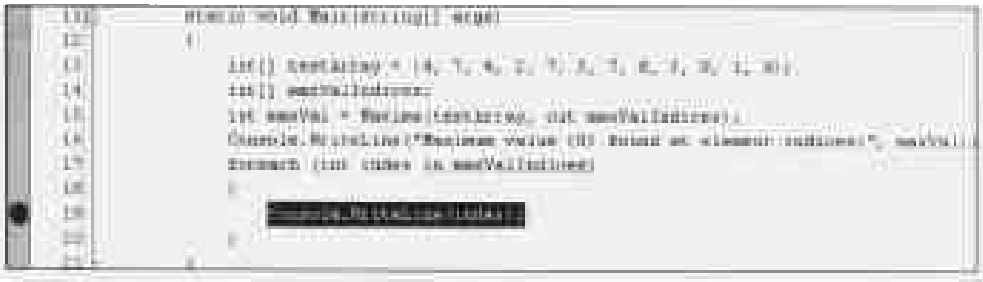


图 7-7

断点在该代码行的旁边显示为一个红色的圆，而该行代码也突出显示，如图 7-8 所示。


图 7-8

使用 Breakpoints 窗口还可以查看文件中的断点信息。首先需要选择 Debug | Windows | Breakpoints 菜单项，激活该窗口。图 7-9 的窗口显示在屏幕的底部，其位置与 Task List 和 Output 窗口相同。


图 7-9

在该窗口中，可以禁用断点(删除描述信息左边的记号；禁用的断点用未填充的红色圆圈来表示)，删除断点，编辑断点的属性。

这个窗口中显示的属性 Condition 和 Hit Count 是惟一的两个可用属性，但它们是非常有用的。右击断点，选择 Properties 菜单项就可以编辑它们(在代码或这个窗口中)。还可以利用三个

标签 Function, File 和 Address 改变断点的位置(Address 标签可以指定断点的绝对内存地址，这里不讨论相关的主题)，Condition... 和 Hit Count...按钮可以改变上述的两个属性。

选择 Condition 按钮将弹出如图 7-10 所示对话框。

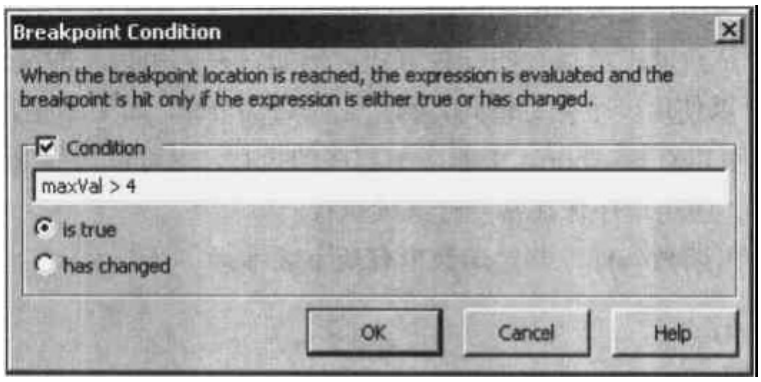


图　7-10

在该对话框中可以输入任意布尔表达式，该表达式则可以包含断点涉及的任何变量。上面的屏幕图显示了一个断点，在遇到这个断点，且 maxVal 的值大于 4 时，就会触发该断点。现在看看这个表达式是否有变化，如果有变化，断点才会被触发(在上面的情况中，如果在遇到断点时，maxVal 从 2 改为 6，就会触发该断点)。

选择 Hit Count 按钮将弹出如图 7-11 所示对话框。

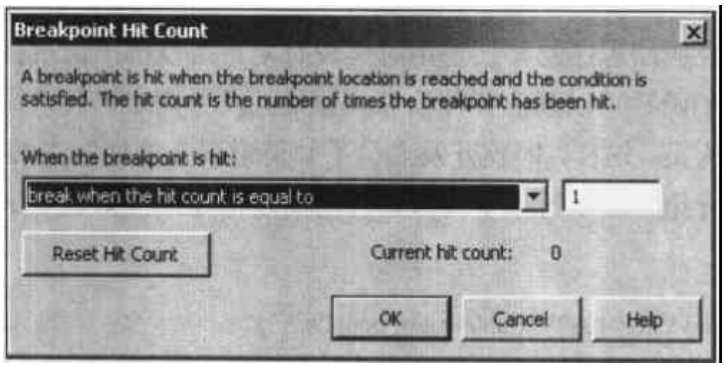


图　7-11

在这个窗口中可以指定在触发一个断点前，要遇到该断点多少次。下拉列表提供了如图 7-12 所示选项。

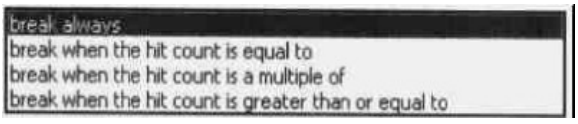


图　7-12

所选的选项与在下边的文本框中输入的值一块确定断点的行为。

这个 Hit Count 按钮在比较长的循环中很有用，例如在执行了前 5000 次循环后需要中断。如果不这么做，中断再重启动 5000 次是很痛苦的。

进入中断模式的其他方式

进入中断模式还有两种方式。一种是在产生一个未处理的异常时选择进入该模式。这种方式在本章后面讨论到错误处理时论述。另一种方式是生成一个判定语句时中断。

判定语句是可以用用户定义的消息中断应用程序的指令。它们常常用于应用程序的开发过程中，作为测试程序是否能平滑运行的一种方式。例如，在应用程序的某一处要求给定的变量值小于 10，此时就可以使用一个判定语句，确定它是否为 true，如果不是，就中断程序的执行。当遇到判定语句时，可以选择 Abort，中断应用程序的执行，也可以选择 Retry，进入中断模式，还可以选择 Ignore，让应用程序像往常一样继续执行。

与前面的调试输出函数一样，判定函数也有两个版本：

- Debug.Assert()
- Trace.Assert()

其调试版本也是仅用于编译调试程序。

这两个函数带三个参数。第一个参数是一个布尔值，其值为 false 会触发判定语句。第二、三个参数是两个字符串参数，分别把信息写到弹出的对话框和 Output 窗口中。上面的范例需要一个函数调用，如下所示：

```
Debug.Assert(myVar < 10, "myVar is 10 or greater.",
                "Assertion occurred in Main().");
```

判定语句通常在应用程序的早期使用时比较有用。可以推出应用程序的一个发布程序，其中包含 Trace.Assert()函数，以列出各种信息。如果触发了判定语句，用户就会收到通知，把这些消息传递给开发人员。这样，即使开发人员不知道错误是如何发生的，也可以改正这个错误。

例如，用在指令第一个字符串中提供错误的简短描述，在第二个字符串中提供下一步该如何操作的指示：

```
Trace.Assert(myVar < 10, "Variable out of bounds.",
                "Please contact vendor with the error code KCW001.");
```

如果触发了这个判定语句，用户就会看到如图 7-13 所示信息。

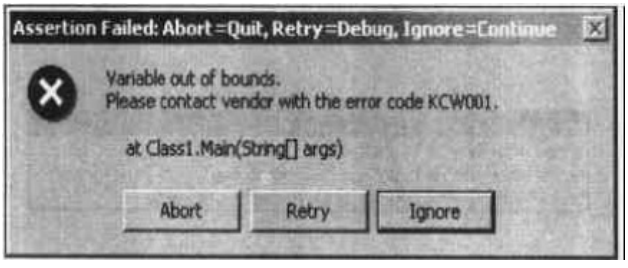


图　7-13

如果用户安装了 VS，并单击了发布程序的 Retry 按钮(他们不想查看代码)，就会看到应用程序的汇编语言指令，这些指令并没有什么启迪作用。下面是上一节中范例的汇编代码：

```
00000196  nop
00000197  pop         ebx
00000198  pop         esi
00000199  pop         edi
0000019a  mov         esp,ebp
0000019c  pop         ebp
0000019d  ret         4
```

这并不容易理解，只有有汇编语言经验的人才能理解它们。

下一个要论述的主题是应用程序中断，以及进入中断模式后，我们可以作什么。一般情况下，进入中断模式的目的是跟踪代码中的错误(或确保程序工作正常)。一旦进入中断模式，就可以使用各种技巧分析代码，分析应用程序在暂停处的状态。

2. 监视变量的内容

监视变量的内容值是 VS 帮助我们使工作变得简单的一个范例。查看变量值的最简单方式是在中断模式下，使鼠标指向源代码中的变量名，此时就会出现一个黄色的工具提示，显示该变量的信息，其中包括该变量的当前值。在图 7-14 中，把鼠标指向 index 变量，可以看到下面的结果。



图　7-14

还可以高亮显示整个表达式，以相同的方式得到该表达式的结果。但这个技巧是有限制的，例如不能显示数组的内容。

目前，你可能注意到，在通过 VS 运行应用程序时，环境中各个窗口的布局发生了变化，在默认情况下，运行期间会发生：

- Properties 窗口消失
- Running Documents 标签添加到 Solution Explorer 窗口中，显示工程当前使用的文档。
- Output 窗口的大小发生了变化，屏幕底部的一半被一个新窗口代替

新显示出来的窗口在调试时非常有用，它允许在中断模式在出应用程序中的变量值上保留标签，它有 3 个标签，如下所示：

Autos—— 当前使用的变量和前面的语句

Locals——在范围内的所有变量

Watch N——可定制的变量和表达式(其中 N 从 1 到 4)

新屏幕布局如图 7-15 所示。

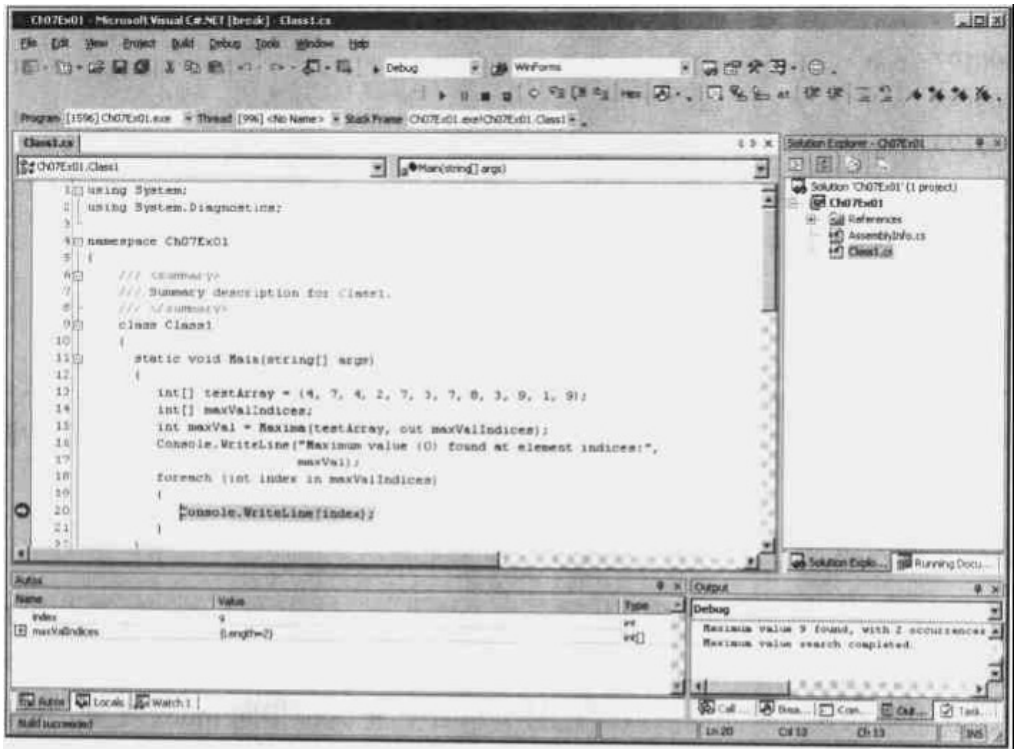


图　7-15

这些窗口的工作方式或多或少有些类似，并根据它们的特定功能添加各种附加特性。一般情况下，每个窗口都包含一个变量列表，有变量的名称、值和类型等信息。更复杂的变量(例如数组)可以使用变量名左边的＋和－扩展/收缩符号进一步查看，它们的内容可以用树状视图的方式显示。例如，在前面的范例中，在 Maxima()调用的后面放置一个断点，得到如图 7-16 所示的窗口。

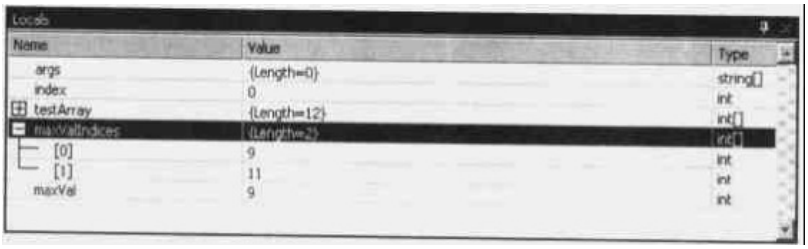


图　7-16

在该窗口中，扩展了一个数组变量 maxValIndices 的视图。

在这个视图中，还可以编辑变量的值。这有效地绕过了前面代码中的其他变量赋值。为此，只需在 Value 列中为要编辑的变量输入一个新值即可。也可以把这种技巧用于其他情况，例如需要修改代码才能编辑变量值的情况。

Watch 窗口至多可以有 4 个，它可以监视特定变量的或涉及特定变量的表达式。要使用这个窗口，只需在 Name 列中输入变量名或表达式，就可以查看它们的结果，注意并不是应用程序中的所有变量在任何时候都可以用，在 Watch 窗口中对变量做出标记。例如，图 7-17 显示了一个 Watch 窗口，其中包含几个样例变量和表达式。这里还是使用了上一个范例的代码，该程序在函数 Maxima()处暂停。



图　7-17

TestArray 数组对于 Main()来说是局部数组，所以在该图中没有值，而是显示一个信息，告诉我们这个变量不能访问。

要在 Watch 窗口添加变量，还可以把变量从源代码拖动到该窗口中。

要添加更多的窗口，可以使用 Debug | Windows | Watch | Watch *N* 菜单选项打开或关闭 Watch 的四个窗口。每个窗口都可以包含变量和表达式的一组观察结果，所以可以把相关的变量组合在一起，以便于访问。

除了这些 Watch 窗口外，还有一个 QuickWatch 窗口，它能快速提供源代码中某个变量的详细信息。要使用这个窗口，可以右击要查看的变量，选择 QuickWatch 菜单选项。但在大多数情况下，使用标准的 Watch 窗口就足够了。

Watch 窗口的一个要点是，它们是在应用程序执行的过程中维护的。如果中断应用程序，再重新运行，就不必再次添加 Watch 窗口了，VS 会记住上次使用的 Watch 窗口。

3. 单步执行代码

前面介绍了如何在中断模式下查看应用程序的运行情况，下面论述如何在中断模式下使用 VS 单步执行代码，查看代码的执行结果，人们的思维速度不会比计算机运行得更快，所以这是一个极有价值的技巧。

进入中断模式后，在代码视图的左边，正在执行的代码旁边会出现一个光标(如果使用断点进入中断模式，该指针最初应显示在断点的红色圆中)，如图 7-18 所示。



图　7-18

这显示了在进入中断模式时程序所执行到的位置。在这个位置上，可以选择逐行执行。为此，使用前面看到的另外一些 Debug 工具栏按钮，如图 7-19 所示。



图　7-19

第 6、7、8 个图标控制了中断模式下的程序流。它们依次是：

- Step Into—— 执行并移动到下一个 要执行的语句上
- Step Over——同上，但不进入嵌套的代码块
- Step Out——执行到代码块的末尾，在执行完该语句块后，重新进入中断模式

如果要查看应用程序执行的每个操作，可以使用 Step Into 按顺序执行指令，这包括在函数中执行，例如上面范例中的 Maxima()。当光标指到第 15 行，调用 Maxima()时，单击这个图标，会使光标移动到 Maxima()函数内部的第一行代码上。而如果光标移到第 15 行时单击 Step Over，就会使光标移动到第 16 行，不必执行 Maxima()中的代码(但仍执行这段代码)。如果单步执行到不感兴趣的函数，可以单击 Step Out，返回到调用该函数的代码。

在单步执行代码时，变量的值可能会发生变化。注意观察上一节讨论的监视窗口，可以看到变量值的变化情况。

在含有语义错误的代码中，这个技巧也许是最有效的。可以单步执行代码，当执行到有错误的代码时，错误会像正常运行程序那样发生。在这个过程中，可以监视数据，看看什么地方出错。本章的后面将使用这个技巧查看范例应用程序的执行情况。

还有其他几个窗口这里没有讨论。在调试过程中，Task List / Output 窗口还将显示另外两个标签：Command Window 和 Call Stack。

4. Immediate 命令

Command Window 有两种模式：Command 和 Immediate。Command 模式可以在不使用菜单栏的情况下执行 VS 操作，Immediate 模式可以执行目前正在执行的标准代码行之间的代码，计算表达式。在 Command 模式下，窗口在每行代码前有一个大于号(>)。在这个窗口中输入"immed"，并按回车键，就可以切换到 Immediate 模式，而输入">cmd"，并按回车键，就可以切换回 Command 模式。

下面详细讨论 Immediate 模式，因为 Command 模式仅适用于复杂的操作。

这个窗口最简单的用法是计算表达式，有点像 Watch 窗口的“one shot”。为此，只需输入一个表达式，并按回车键即可。接着就会显示请求的信息，如图 7-20 所示。



图　7-20

也可以在这里修改变量的内容，如图 7-21 所示。



图　7-21

在大多数情况下，使用前面介绍的变量监视窗口更容易得到相同的效果，但这个技巧对于常常发生变化的变量值仍很方便，也适合于测试以后不感兴趣的表达式。

5. Call Stack 窗口

这是最后一个要讨论的窗口，它描述了程序是如何执行到当前位置的。简言之，该窗口显示了当前函数、调用它的函数，以及调用函数的函数(即一个嵌套的函数调用列表)。调用的位置也被记录下来。

在前面的范例中，在执行到 Maxima()时进入中断模式，或者使用代码单步执行功能，移动到这个函数的内部，得到如图 7-22 所示窗口。

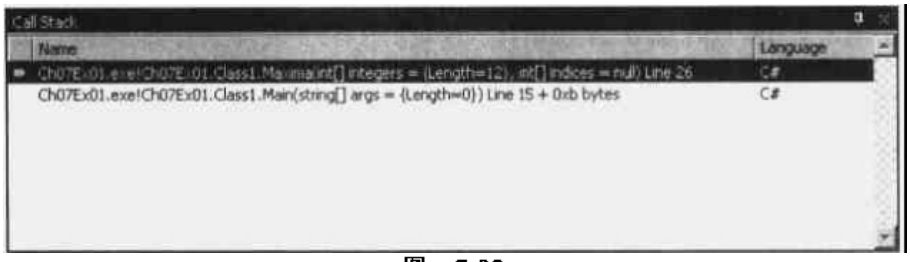


图　7-22

在第一次检测错误时，这个窗口非常有用，因为它们可以用来查看错误发生前的情况。对于常用函数中出现的错误，它有助于找到错误的源头。

注意：

有时这个窗口会显示一些非常杂乱的信息，例如有时因为以错误的方式使用了外部函数，错误在应用程序的外部发生，就会出现这种情况。此时，这个窗口中会列出一个很长的列表，其中只有一两个选项是我们熟悉的。

7.2　错误处理

本章的第一部分讨论如何在应用程序的开发过程中查找和改正错误，使这些错误不会在发布的代码中出现。但有时，我们知道可能会有错误发生，但不能 100%地肯定它们不会发生。此时，最好预料错误，编写足够强壮以处理这些错误的代码，而不必中断程序的执行。

错误处理就是用于这个目的的，下面还将介绍异常和处理它们的方式。

异常

异常是代码中产生的错误，或者在运行期间由代码调用的函数中产生的错误。这里的“错误”定义要比以前更含糊，因为异常可能是在函数等中手工产生。例如，如果函数的一个字符串参数不是以 a 开头，就产生一个异常。这并不是严格意义上的函数外部错误，但把它作为函数外部错误，由调用该函数的代码来处理。

本书在前面已经遇到几次异常了。最简单的范例是试图定位一个超出范围的数组元素，例如：

```
int[] myArray = {1, 2, 3, 4};
int myElem = myArray[4];
```

这会产生如下异常信息，并中断应用程序的执行：

```
An unhandled exception of type 'System.IndexOutOfRangeException' occurred in <file>.exe
```

其中<file> 是包含异常的文件名。

异常在名称空间中定义，大多数异常的名称清晰地说明了它们的用途。在这个范例中，产生的异常叫做 System.IndexOutOfRangeException，说明我们提供的 myArray 数组下标不在允许使用的下标范围内。

在异常未处理时，这个信息才会显示出来，应用程序也才会中断执行。那么如何处理异常呢？

1. try...catch...finally

C#语言包含结构化异常处理(Structured Exception Handling ，SEH)的语法。关键字可以标记出能处理异常的代码和指令，如果发生异常，就使用这些指令处理异常。用于这个目的的三个关键字是 try, catch 和 finally。它们都有一个关联的代码块，必须在连续的代码行中使用。其基本结构如下：

```
try
{
    ...
}
catch (<exceptionType> e)
{
    ...
}
finally
{
    ...
}
```

也可以只有 try 块和 finally 块，而没有 catch 块，或者有一个 try 块和好几个 catch 块。如果有一个或多个 catch 块，finally 块就是可选的，否则就是必须有的。

这些代码块的用法如下：

- try——包含产生异常的代码，在谈到异常时，其中的“产生”在 C#语言中也可以是“生成”或“导致”。
- catch——包含在产生异常时要执行的代码。catch 块可以使用<exceptionType>，设置为只响应特定的异常类型(例如 System.IndexOutOfRangeException)，以便提供多个 catch 块。还可以完全省略这个参数，让一般的 catch 块响应所有的异常。
- Finally——包含总是会执行的代码，如果没有产生异常，则在 try 块之后执行，如果处理了异常，就在 catch 块后执行，或者在未处理的异常中断应用程序之前执行(此时处理这个块，

其原因是存在这个块，否则也可以在这个块的后面放置代码)。

在 try 块的代码中出现异常后， 发生的事件依次是：

- try 块在发生异常的地方中断程序的执行。
- 如果有 catch 块，就检查该块是否匹配于已发生的异常类型。如果没有 catch 块，就执行 finally 块(如果 catch 块，就一定要有 finally 块)。
- 如果有 catch 块，但它与已发生的异常类型不匹配，就检查是否有其他 catch 块。
- 如果有 catch 块匹配于已发生的异常类型，就执行它包含的代码，再执行 finally 块(如果有)。
- 如果 catch 块不匹配于已发生的异常类型，就执行 finally 块(如果有)。

下面用一个范例来说明异常的处理。

试试看：把文本写到 Output 窗口中

(1) 在目录 C:\BegCSharp\Chapter7 下创建一个新控制台应用程序 Ch07Ex02。

(2) 修改代码，如下所示：

```
class Class1
{
    static string[] eTypes = {"none", "simple", "index", "nested index"};

    static void Main(string[] args)
    {
        foreach (string eType in eTypes)
        {
            try
            {
                Console.WriteLine("Main() try block reached.");            // Line 18
            Console.WriteLine("ThrowException(\"{0}\") called.", eType);
                                                                           // Line 19
                ThrowException(eType);
                Console.WriteLine("Main() try block continues.");          // Line 21
            }
            catch (System.IndexOutOfRangeException e)                      // Line 23
            {
                Console.WriteLine("Main() System.IndexOutOfRangeException catch"
                                  + " block reached. Message:\n\"{0}\"",
                                  e.Message);
            }
            catch                                                          // Line 29
            {
                Console.WriteLine("Main() general catch block reached.");
            }
            finally
            {
```

```
            Console.WriteLine("Main() finally block reached.");
        }
        Console.WriteLine();
    }
}

static void ThrowException(string exceptionType)
{
                                                                    // Line 43
    Console.WriteLine("ThrowException(\"{0}\") reached.", exceptionType);
    switch (exceptionType)
    {
        case "none" :
            Console.WriteLine("Not throwing an exception.");
            break;                                                  // Line 48
        case "simple" :
            Console.WriteLine("Throwing System.Exception.");
            throw (new System.Exception());                         // Line 51
            break;
        case "index" :
            Console.WriteLine("Throwing System.IndexOutOfRangeException.");
            eTypes[4] = "error";                                    // Line 55
            break;
        case "nested index" :
            try                                                     // Line 58
            {
                Console.WriteLine("ThrowException(\"nested index\") " +
                                  "try block reached.");
                Console.WriteLine("ThrowException(\"index\") called.");
                ThrowException("index");                            // Line 63
            }
            catch                                                   // Line 65
            {
                Console.WriteLine("ThrowException(\"nested index\") general"
                                  + " catch block reached.");
            }
            finally
            {
                Console.WriteLine("ThrowException(\"nested index\") finally"
                                  + " block reached.");
            }
            break;
    }
  }
}
```

(3) 运行应用程序，如图 7-23 所示。



图　7-23

代码的说明

这个应用程序在 Main()中有一个 try 块，它调用函数 ThrowException()。这个函数会根据它使用的参数产生异常：

- ThrowException("none")——不产生异常。
- ThrowException("simple")——产生一般异常。
- ThrowException("index")——产生 System.IndexOutOfRangeException 异常。
- ThrowException("nested index")——包含它自己的 try 块，其中的代码用每个可能的参数调用 ThrowException("index")，产生一个 System.IndexOutOfRangeException 异常。

这些 string 参数存储在全局数组 eTypes 中，在 Main()函数中迭代，用每个可能的参数调用 ThrowException()。在迭代过程中，会把各种信息写到控制台上，说明发生了什么情况。

这段代码可以使用本章前面介绍的代码单步执行技巧。在执行代码的过程中，一次执行一行代码可以确切地了解代码的执行进度。

在代码的第 18 行添加一个断点(用默认的属性)，该行代码如下：

```
Console.WriteLine("Main() try block reached.");
```

注意：

这里使用行号，因为它们显示在这段代码的下载版本中。如果关闭了行号，记住可以通过 Tools | Options...菜单选项和 Text Editor | C# | General 选项打开它们。上述代码包含注释，这样读者可以阅读文字，而无需打开文件。

在调试模式下运行应用程序。

程序立即进入中断模式，此时光标在第 18 行上。如果选择变量监视窗口中的 Locals 标签，就会看到 eType 当前是"none"。使用 Step Into 图标处理第 18 和 19 行，看看第一行文本是否已经写到控制台上。接着使用 Step Into 按钮单步执行第 20 行的 ThrowException()函数。

执行到 ThrowException()函数(第 43 行)后，Locals 窗口会发生变化。eType 和 args 不再能访问(因为它们是 Main()的局部变量)，我们看到的是 exceptionType 局部变元，它当然是"none"。继续单击 Step Into，到达 switch 语句，检查 exceptionType 的值，执行代码，把字符串"Not throwing an exception"写到屏幕上。在执行第 48 行上的 break 语句时，将退出函数，继续处理 Main()中的第 21 行代码。因为没有产生异常，所以继续执行 try 块。

接着处理 finally 块。再单击 Step Into 几次，执行完 finally 块和 foreach 的第一次循环。下次执行到第 20 行时，使用另一个参数"simple"调用 ThrowException()。

继续使用 Step Into 单步执行 ThrowException()，最终会执行到第 51 行：

```
throw (new System.Exception());
```

这里使用 C# throw 关键字产生一个异常，这个关键字需要把新初始化的异常作为其参数，产生一个异常，这里使用名称空间 System 中的另一个异常 System.Exception。

在使用 Step Into 执行这个语句时，将从第 29 行开始执行一般的 catch 块。因为不能与从前面第 23 行开始的其他 catch 块相匹配，所以执行这个一般的 catch 块。单步执行这段代码，然后执行 finally 块，最后返回另一个循环周期，该循环在第 20 行用一个新参数调用 ThrowException()，这次的参数是"index"。

这次 ThrowException()在第 25 行产生一个异常：

```
eTypes[4] = "error";
```

eTypes 数组是一个全局数组，所以可以在这里访问它。但是这里我们试图访问数组中的第 5 个元素(记住其下标从 0 开始计数)，这会产生一个 System.IndexOutOfRangeException 异常。

这次 Main()中有一个匹配的 catch 块，单步执行该 Catch 块，从第 23 行开始。

这个块中调用的 Console.WriteLine()使用 e.Message 输出存储在异常中的信息(可以通过 catch 块的参数访问异常)。之后再次单步执行 finally 块(而不是第二个 catch 块，因为异常已经处理完了)。返回循环，再次调用第 20 行的 ThrowException()。

在执行到 ThrowException()中的 switch 结构时，这次输入一个新的 try 块，从第 58 行开始。在执行到第 63 行时，将遇到 ThrowException()的一个嵌套调用，这次是使用 "index"参数。可以使用 Step Over 按钮跳过其中的代码行单步执行，因为前面已经单步执行过了。与前面一样，这个调用产生 System.IndexOutOfRangeException 异常。但这次异常在 ThrowException()中的嵌套 try...catch...finally 结构中处理。这个结构没有明确匹配于这种异常的 catch 块，所以执行一般的 catch 块(从第 65 行开始)。

与前面的异常处理一样，现在单步执行这个 catch 块，以及关联的 finally 块，最后返回到函数调用的末尾。但是它们有一个重大的区别。产生的异常是由 ThrowException()中的代码处理的。这就是说，异常并没有留给 Main()处理，所以直接进入 finally 块，之后应用程序中断执行。

2. 列表和配置异常

.NET 框架包含许多异常类型，可以在代码中自由产生和处理这些类型的异常，甚至可以在代码中产生异常，让它们在比较复杂的应用程序中被捕获。VS 提供了一个对话框，可以检查

和编辑可用的异常，该对话框可以使用 Debug | Exceptions...菜单选项(或按下 Ctrl+Alt+E)打开，如图 7-24 所示。

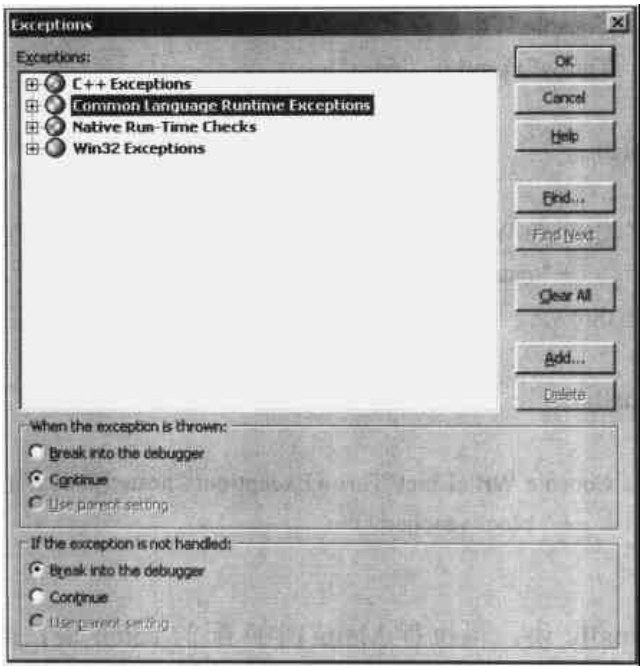


图　7-24

异常按照类别和.NET 库名称空间列出。扩展 Common Language Runtime Exceptions 选项，再扩展 System 选项，就可以看到 System 名称空间中的异常，这个列表包括上面使用的异常 System.IndexOutOfRangeException。

每个异常都可以使用对话框底部的单选按钮来配置。大多数异常在默认情况下都设置为 Use parent setting，使用类别级的选项(它们都在上面的屏幕图中显示出来了)。可以使用第一个选项 When the exception is thrown，中断调试器，即使是对于要处理的异常，也是这样。第二个选项可以忽略未处理的异常，这样做对结果会有影响。

在大多数情况下，使用默认设置就足够了。

3. 异常处理的注意事项

注意，必须在更一般的异常捕获之前为比较特殊的异常提供 catch 块。如果执行了错误的 catch 块，应用程序就会编译失败。

还要注意可以在 catch 块中产生异常，方法是使用上一个范例中的方式，或使用下述表达式：

```
throw;
```

这个表达式会再次产生 catch 块处理过的异常。

如果以这种方式产生异常，该异常就不会由当前的 try...catch...finally 块处理，而是由上一级的代码处理(但嵌套结构中的 finally 块仍会执行)。

例如，如果修改 ThrowException()中的 try...catch...finally 块，如下所示：

```
try
```

```
{
    Console.WriteLine("ThrowException(\"nested index\") " +
                      "try block reached.");
    Console.WriteLine("ThrowException(\"index\") called.");
    ThrowException("index");
}
catch
{
    Console.WriteLine("ThrowException(\"nested index\") general"
        + " catch block reached.");
    throw;
}
finally
{
    Console.WriteLine("ThrowException(\"nested index\") finally"
        + " block reached.");
}
```

则首先执行其中的 finally 块，再执行 Main()中匹配的 catch 块，得到的控制台输出则如图 7-25 所示。



图　7-25

在这个屏幕图中，Main()函数输出了额外的几行，因为这个函数捕获了 System.IndexOutOfRangeException 异常。

7.3　小结

本章主要论述了调试应用程序所使用的技巧。有许多这方面的技巧，其中的大部分可用于各种类型的工程，而不仅仅是控制台应用程序。

本章的主要内容如下：

- 使用 Debug.WriteLine() 和 Trace.WriteLine() 把文本输出到 Output 窗口中
- 中断模式，如何进入中断模式，包括各种断点
- VS 中的调试信息窗口
- 单步执行代码

- 使用 try...catch...finally 处理异常

前面介绍了创建简单控制台应用程序的所有内容，以及调试它们的各种方法。本书的下一部分讨论面向对象编程的强大技术。

7.4　练习

(1) “使用 Trace.WriteLine() 要优于使用 Debug.WriteLine()，因为调试版本仅能用于调试程序。”你同意这个观点吗？为什么？

(2) 为一个简单的应用程序编写代码，其中包含一个循环，该循环在运行 5000 次后产生一个错误。使用断点在第 5000 次循环出现错误前进入中断模式(注意错误是试图访问一个不存在的数组元素，例如在一个有 100 个元素的数组中，访问 myArray[1000])。

(3) “如果 catch 块没有执行，就执行 finally 代码块”对吗？

(4) 下面定义了一个枚举数据类型 orientation。编写一个应用程序，使用结构化异常处理(SEH)把 byte 类型的变量安全地强制转换为 orientation 类型变量。注意可以使用 checked 关键字产生异常，下面是一个范例。这段代码可以在应用程序中使用：

```
enum orientation : byte
{
    north = 1,
    south = 2,
    east  = 3,
    west  = 4
}

myDirection = checked((orientation)myByte);
```

第8章　面向对象编程简介

本书前面介绍了C#语法和编程的所有基础知识。现在我们已经可以写出能使用的控制台应用程序了。但是，要了解 C#语言和.NET Framework 的强大功能，还需要使用面向对象编程(Object-Oriented Programming，OOP)技术。实际上，前面已经在使用这些技术，但为了使学习任务简单一些，在列出代码范例时没有重点讲述该技术。

本章先不考虑代码，而主要探讨 OOP 的规则。OOP 会很快把我们领回 C#语言，因为它与 OOP 是一种共生关系。本章介绍的所有概念在后面的章节中都会再次讨论，并用演示性的代码来说明。所以，如果你在第一次阅读本章时没有掌握所有的内容，不必惊慌。

首先看看 OOP 的基础知识，包括回答最基本的问题“什么是对象？”。很快你就会发现有许多术语与 OOP 有关，这些术语最初很容易混淆，它们使用了许多解释性的语言。使用 OOP 需要以另一种方式来看待编程。

除了讨论 OOP 的一般规则外，本章还将进入一个对于全面理解 OOP 非常重要的领域：Windows Forms 应用程序。这些类型的应用程序(它们使用 Windows 环境和诸如菜单、按钮等特性)有许多描述性的区域，在 Windows Forms 环境中可以有效地说明 OOP 要点。

注意：

本章中的 OOP 是.NET OOP，这里讲述的一些技术不能应用于其他 OOP 环境。在学习 C#时，这有助于更容易地掌握后面的内容。

8.1　什么是面向对象编程

面向对象编程是创建计算机应用程序的一种相当新的方法，它解决了所谓的传统编程技巧带来的问题。前面介绍的编程方法称为函数(或过程)化编程，常常会导致所谓的单一应用程序，即所有的功能都包含在几个代码模块中(常常是一个代码模块)。而使用 OOP 技术，常常要使用许多代码模块，每个模块都提供特定的功能，每个模块都是孤立的，甚至与其他模块完全独立。这种编程方法提供了非常大的多样性，大大增加了代码的重用机会。

要进一步说明这个问题，假定计算机上的一个高性能应用程序是一辆一流赛车。如果使用传统的编程技巧，这辆赛车就是一个单元。如果要改进该车，就必须替换整个单元，把它卖给厂商，让汽车专家升级它。此时的代码一般与使用各种数学和逻辑方式处理数据相关。处理方法通常比较简单，使用基本的数据类型，例如整型和布尔值，建立比较复杂的数据表达方式。

而使用 OOP，事情就不是这么直接了。尽管可以获得相同的效果，但其实现方式是完全不同的。OOP 技术以结构、数据的含义以及数据和数据之间的交互操作为基础。这通常意味着要把更多的精力放在工程的设计阶段，但工程的可扩展性比较高。一旦对某种类型的数据的表达

方式达成一致，这种表达方式就会应用到应用程序以后的版本中，甚至是全新的应用程序中。这种一致的表达方式可以大大减少开发时间。这就是上述赛车范例的工作原理。这里的一致是“引擎”的代码是结构化的，这样新代码(即新引擎)就能很容易地构建出来，而不需要找厂商帮忙。

除了数据表达方式的一致外，OOP 编程常常可以简化任务，因为较抽象实体的构建和使用也是一致的。例如，不仅把输出结果发送给设备(例如打印机)所使用的数据格式是一致的，而且与该设备交换数据的方法也是一致的，这包括它理解的指令等等。

顾名思义，OOP 技术要使用对象，那么，什么是对象呢？

8.1.1　什么是对象

对象就是 OOP 应用程序的一个组成部件。这个组成部件封装了部分应用程序，这部分程序可以是一个过程、一些数据或一些更抽象的实体。

简单地说，对象非常类似于本书前面讨论的结构类型，包含变量成员和函数类型。它所包含的变量组成了存储在对象中的数据，其中包含的函数可以访问对象的功能。略为复杂的对象可能不包含任何数据，而是只包含函数，表示一个过程。例如，可以使用表示打印机的对象，其中的函数可以控制打印机(允许打印文档、测试页等)。

C#中的对象是从类型中创建的，就像前面的变量一样。对象的类型在 OOP 中有一个特殊的名称：类。可以使用类定义实例化对象，这表示创建该类的一个实例。“类的实例”和对象表示相同的含义，注意“类”和“对象”是完全不同的东西。

本章将使用通用建模语言(Universal Modeling Language，UML)语法研究类和对象。UML 是为应用程序建模而设计的，从组成应用程序的对象，到它们执行的操作，到我们希望有的用法，应有尽有。这里只使用这个语言的基本部分，在使用它们的过程中进行解释，但不考虑比较复杂的部分。

提示：

本章中的图都是由 Microsoft Visio 创建的，VS 的企业版本附带有该软件。

图 8-1 是打印机类 Printer 的 UML 表示方法。

类名显示在这个框的顶部(后面论述下面两个区域)。

图 8-2 是这个 Printer 类的一个实例 myPrinter。



图　8-1

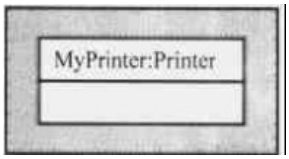


图　8-2

在顶部，实例名显示在前面，后面是类名。这两个名称用一个冒号分隔开。

1. 属性和字段

属性和字段可以访问对象中包含的数据。这个对象数据是区分不同的对象，因为同一个类

的不同对象可能在属性和字段中存储了不同的值。

下面要介绍另一个术语。包含在对象中的不同数据统称为对象的状态。

假定一个对象类表示一杯咖啡，叫做 CupOfCoffee。在实例化这个类(即创建这个类的对象)时，必须提供对于类有意义的状态。此时可以使用属性和字段，让代码能通过该对象设置要使用的咖啡品牌，咖啡中是否加牛奶或方糖，咖啡是否即溶等。给定的咖啡杯对象也有一个指定的状态，例如 Columbian filter coffee with milk and two sugars'。

字段和属性都可以输入，所以可以把信息存储在字段和属性中，作为 string 变量、int 变量等。但是，属性与字段是不同的，因为属性不能直接访问数据。对象能让用户不考虑数据的细节，不需要在属性中用一对一的方式表示。如果在 CupOfCoffee 实例中使用一个字段表示方糖的数量，用户就可以在该字段中放置自己喜欢的数值，但如果使用一个属性来表示，就可以限制这个值，例如为 0 到 2 之间的一个数字。

一般情况下，在访问状态时最好提供属性，而不是字段，因为这样可以更好地控制整个过程，而使用它们的语法是相同的。对属性的读写访问也可以由对象来明确定义。某些属性是只读的，只能查看它们的值，而不能改变它们(至少不能直接改变)。这常常是同时读取几个状态的一个有效技巧。CupOfCoffee 类有一个只读属性 Description，在请求它时，就返回一个字符串，表示该类的一个实例的状态(例如前面给出的字符串)。也可以通过查看几个属性，把相同的数据组合起来，但这样的属性可以节省时间和精力。还可以有只写的属性，其操作方式是类似的。

除了对属性的读写访问外，还可以为字段和属性指定另一种访问，这种可访问性确定了什么代码可以访问这些成员，它们是可用于所有的代码(公共)，还是只能用于类中的代码(私有)，或者更复杂的模式(详见本章后面的内容)。常见的情况是把字段设置为私有，通过公共属性访问它们。这样，类中的代码就可以直接访问存储在字段中的数据，而公共属性禁止外部用户访问这些数据，以防他们在其中放置无效的内容。公共成员是类可以访问的成员。

要更清楚地说明这个问题，可以把成员与变量的作用域等同起来。例如，私有字段和属性可以看作是拥有他们的对象的本地成员，而公共字段和属性的作用域也包括对象外部的代码。

在类的 UML 表示方法中，用第二部分显示属性和字段，如图 8-3 所示。

这是 CupOfCoffee 类的表示方式，前面为它定义了 5 个成员(属性或字段，在 UML 中，它们没有区别)。每个成员都包含下述信息：

- 可访问性：+号用于表示公共成员，-号用于表示私有成员。但一般情况下，本章的图中不显示私有成员，因为这些信息是类内部的信息。至于读/写访问，则不提供任何信息。
- 成员名
- 成员的类型

冒号用于分隔成员名和类型。

2. 方法

“方法”这个术语用于表示对象中的函数。这些函数调用的方式与其他函数相同，使用返回值和参数的方式也相同(详见第 6 章)。

方法用于访问对象的功能。与字段和属性一样，方法可以是公共的或私有的，按照需要限

制外部代码的访问。它们常常使用对象状态影响它们的操作，在需要时访问私有成员，如私有字段。例如，CupOfCoffee 类定义了一个方法 AddSugar()，与设置相应的 Sugar 属性相比，该方法对递增方糖数属性提供了比设置相应的 Sugar 属性更易读的语法。

在 UML 的对象框中，方法显示在第三部分，如图 8-4 所示。

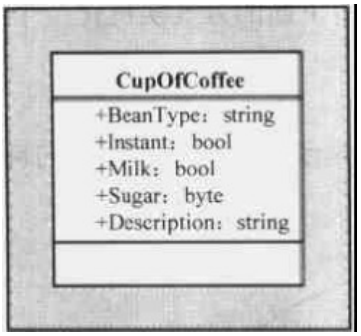


图　8-3

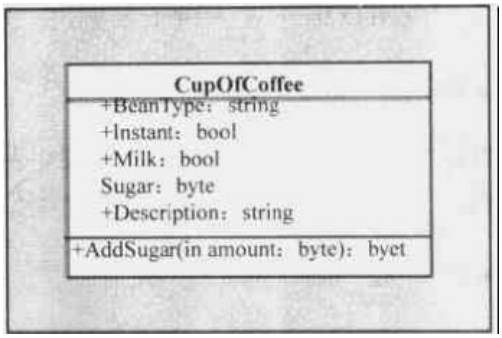


图　8-4

其语法类似于字段和属性，但最后显示的类型是返回类型，在这一部分，还显示了方法的参数。在 UML 中，每个参数都带有下述辨识符之一：in, out 或 inout。它们用于表示数据流的方向，其中 out 和 inout 大致对应于第 6 章讨论的 C#关键字 out 和 ref。in 大致对应于 C#中不使用这两个关键字的情况。

8.1.2　所有的东西都是对象

本书一直在使用对象、属性和方法。实际上，C#和.NET Framework 中的所有东西都是对象。控制台应用程序中的 Main()函数就是类的一个方法。前面介绍的每个变量类型都是一个类。前面使用的每个命令都是一个属性或方法，例如<String>.Length 和 <String>.ToUpper()等。句点字符把对象实例名和属性或方法名分隔开来。

对象无处不在，使用它们的语法也比较简单，这使我们可以集中精力讨论 C#中一些比较基础的方面。

从现在开始详细介绍对象。记住这里讨论的概念都有着深远的影响。它们甚至可以应用到简单的 int 变量上。幸运的是可以使用它来理解这些概念。

8.1.3　对象的生命周期

每个对象都一个明确定义的生命周期，即从使用类定义开始一直到删除它为止。在对象的生命周期中，除了“正在使用”的正常状态之外，还有两个重要的阶段：

- 构造阶段——对象最初进行实例化的时期。这个初始化过程称为构造阶段，由构造函数完成。
- 析构阶段——在删除一个对象时，常常需要执行一些清理工作，例如释放内存，由析构函数完成。

1. 构造函数

对象的初始化过程是自动完成的。我们不需要找一个适合于存储新对象的内存空间。但是，在初始化对象的过程中，有时需要执行一些额外的工作。例如，需要初始化对象存储的数据。

构造函数就是用于初始化数据的函数。

所有的对象都有一个默认的构造函数，该函数没有参数，与类本身有相同的名称。一个类定义可以包含几个构造函数，它们有不同的签名，代码可以使用这些签名实例化对象。带有参数的构造函数通常用于给存储在对象中的数据提供初始值。

在 C#中，构造函数用 new 关键字来调用。例如，可以用下面的方式实例化一个 string 对象：

```
string myString = new string();
```

对象还可以用非默认的构造函数来创建。与默认的构造函数一样，非默认的构造函数与类同名，但它们还带有参数，使用的方式则是类似的，例如：

```
string myString = new string('a', 10);
```

这个构造函数建立了一个新字符串，它的初始值是"aaaaaaaaaa"，即 10 个字母 a。

构造函数与字段、属性和方法一样，可以是公共或私有的。在类外部的代码不能使用私有构造函数实例化对象，而必须使用公共构造函数。这样，就可以要求类的用户使用非默认的构造函数。

一些类没有公共的构造函数，外部的代码就不可能实例化它们，但是，这些类并不是完全没有用的，详见后面的内容。

2. 析构函数

析构函数由.NET Framework 用于清理对象。一般情况下，不需要提供解构方法的代码，而是由默认的析构函数执行操作。但是，如果在删除对象实例前，需要完成一些重要的操作，就应提供特定的析构函数。

记住，并不是在停止使用对象后，立即就调用对象的析构方法。

例如，如果变量超出了范围，代码就不能访问它，但该变量仍存在于计算机内存的某个地方。只有在.NET 运行程序执行其无用存储单元收集器，进行清理时，该实例才被彻底删除。

这就是说，我们不应依赖析构函数释放对象实例使用的资源，因为在不再使用某个对象后，该资源会长时间被该对象占用。如果所使用的资源非常重要，这样做就有可能出问题。有一个解决方法，参阅本章后面的“一次性的对象”一节。

8.2 静态和实例类成员

属性、方法和字段等成员是对象实例所特有的，除此之外，还有静态成员(也称为共享成员)，例如静态方法、静态属性或静态字段。静态成员可以在类的实例之间共享，所以它们可以看作是类的全局对象。静态属性和静态字段可以访问独立于任何对象实例的数据，静态方法可以执行与对象类型相关、但不是特定实例的命令，在使用静态成员时，甚至不需要实例化类型的对象。

例如，前面使用的 Console.WriteLine()和 Convert.ToString()方法就是静态的，根本不需要实例化 Console 或 Convert 类(如果试着进行这样的实例化，操作会失败，因为这些类的构造

函数不是可公共访问的，如前所述。）

在许多情况下，静态属性和方法有很好的效果。

在 UML 语法中，类的静态成员用下划线表示，如图 8-5 所示。

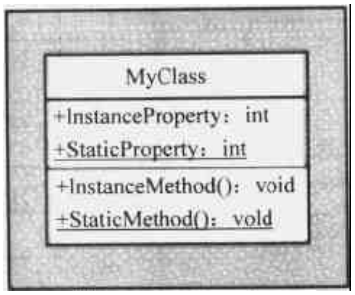


图 8-5

8.3 OOP 技术

前面介绍了一些基础知识，知道对象是什么，以及对象的工作原理，下面讨论对象的其他一些特性，包括：

- 接口
- 继承
- 多态性
- 对象之间的关系
- 运算符重载
- 事件

8.3.1 接口

接口是把隐式公共方法和属性组合起来，以封装特定功能的一个集合。一旦定义了接口，就可以在类中执行它。这样，类就可以支持接口所指定的所有属性和成员。

注意，接口不能单独存在。不能像实例化一个类那样实例化接口。另外，接口不能包含执行其成员的任何代码，而只能定义成员本身。执行过程必须在执行接口的类中实现。

在前面的咖啡范例中，可以把较一般用途的属性和方法例如 AddSugar(), Milk, Sugar 和 Instant 组合到一个接口中，这个接口称为 IHotDrink(接口的名称一般用大写字母 I 开头)。然后就可以在其他对象上使用该接口，例如 CupOfTea 类。所以可以用类似的方式处理这些对象，而对象仍保有自己的属性(例如 CupOfCoffee 仍有属性 BeanType，CupOfTea 仍有属性 LeafType)。

在 UML 中，给对象执行的接口用“棒棒糖”语法来表示。在图 7-6 中，用与类相似的语法把 IHotDrink 的成员放在一个单独的框中(但 Visio 的当前版本不允许接口拥有字段或属性)。

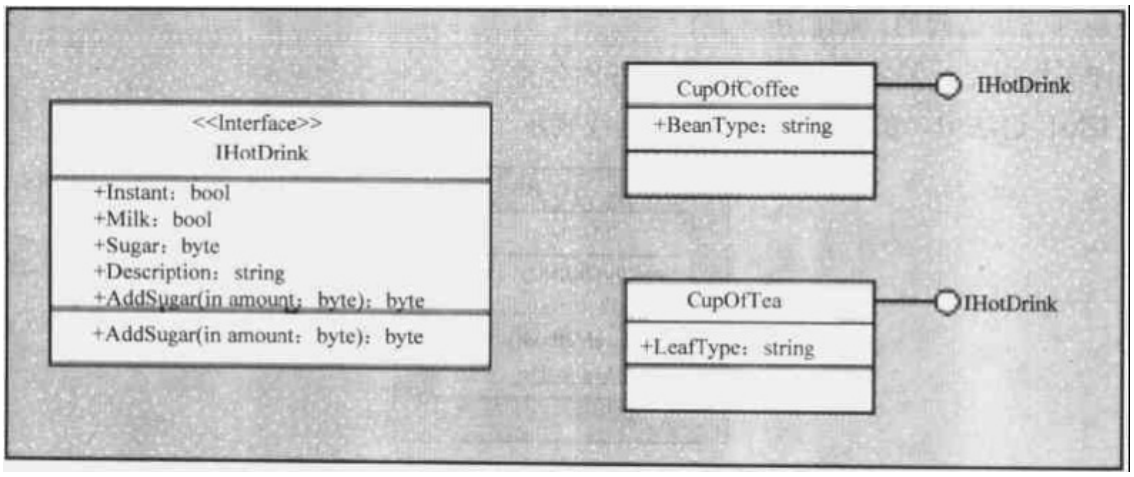


图　8-6

一个类可以支持多个接口，多个类也可以支持相同的接口。所以接口的概念让用户和其他开发人员更容易理解其他人的代码。例如，有一些代码使用一个带某接口的对象。假定不使用这个对象的其他属性和方法，就可以用另一个对象代替这个对象(例如，使用上述 IHotDrink 接口的代码可以处理 CupOfCoffee 和 CupOfTea 实例)。另外，该对象的开发人员可以提供该对象的更新版本，只要它支持已经在使用的接口，就可以在代码中使用该对象。

可删除的对象

IDisposable 接口特别有趣。支持 IDisposable 接口的对象必须执行 Dispose()方法，即它们必须提供这个方法的代码。当不再需要某个对象(例如在对象超出范围之前)时，就调用这个方法，释放重要的资源，否则，该资源会等到对无用存储单元收集器调用析构方法时才释放。这样可以更好地控制对象所使用的资源。

C#允许使用一种可以优化使用这个方法的结构。Using 关键字可以在代码块中初始化使用重要资源的对象，Dispose()方法会在这个代码块的末尾自动调用，用法如下：

```
using (<ClassName> <VariableName> = new <ClassName>())
{
    ...
}
```

其中变量<VariableName>可以在这个代码块中使用，并在代码块的末尾自动删除(调用 Dispose())。

8.3.2　继承

继承是 OOP 最重要的特性之一。任何类都可以从另一个类继承，这就是说，这个类拥有它继承的类的所有成员。在 OOP 中，被继承(也称为派生)的类称为父类(也称为基类)。注意 C#中的对象仅能派生于一个基类。

继承性可以从一个基类扩展或创建更多的特定类。例如，考虑一个代表农场家畜的类(80 多岁的一流开发人员 Old MacDonald 在他的家畜应用程序中使用)。这个类叫做 Animal，拥有方法如 EatFood() 或 Breed()，我们可以创建一个派生类 Cow，支持所有这些方法，它也有自

己的方法，如 Moo() 和 SupplyMilk()。还可以创建另一个派生类 Chicken，该类有 Cluck() 和 LayEgg()方法。

在 UML 中，用箭头表示继承，如图 8-7 所示。

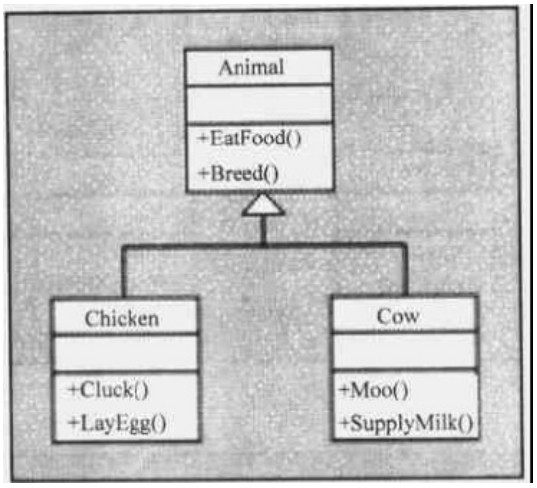


图　8-7

提示：

为了简洁，这里省略了成员的返回类型。

在继承一个基类时，成员的可访问性就成为一个重要的问题。派生类不能访问基类的私有成员，但可以访问其公共成员。不过，派生类和外部的代码都可以访问公共成员。这就是说，只使用这两个可访问性，不仅可以让一个成员被基类和派生类访问，而且也能够被外部的代码访问。

为了解决这个问题，C#提供了第三种可访问性：protected，只有派生类才能访问一个 protected 成员。考虑到外部代码，这个可访问性与私有成员一样，基类和外部代码都不能访问 protected 成员。

除了成员的保护级别外，我们还可以为成员定义其继承行为。基类的成员可以是虚拟的，也就是说，成员可以由继承它的类重写。派生类可以提供成员的其他执行代码。这种执行代码不会删除原来的代码，仍可以在类中访问原来的代码，但外部代码不能访问它们。如果没有提供其他执行方式，外部代码就访问基类中成员的执行代码。

注意虚拟成员不能是私有成员，因为这样会自相矛盾——不能说成员可以由派生类重写，同时派生类又不能访问它。

在前面的家畜范例中，可以把 EatFood()变成虚拟成员，为它提供派生类的新执行代码，例如为 Cow 类提供新执行代码，如图 8-8 所示。

这里显示了 Animal 和 Cow 类的 EatFood()方法，说明它们有自己的执行代码。

基类还可以定义为抽象类。抽象类不能直接实例化。要使用抽象类，必须继承这个类，抽象类可以有抽象成员，这些成员在基类中没有这些代码，所以这些执行代码必须在派生类中提供。

如果 Animal 是一个抽象类，UML 就会如图 8-9 所示。

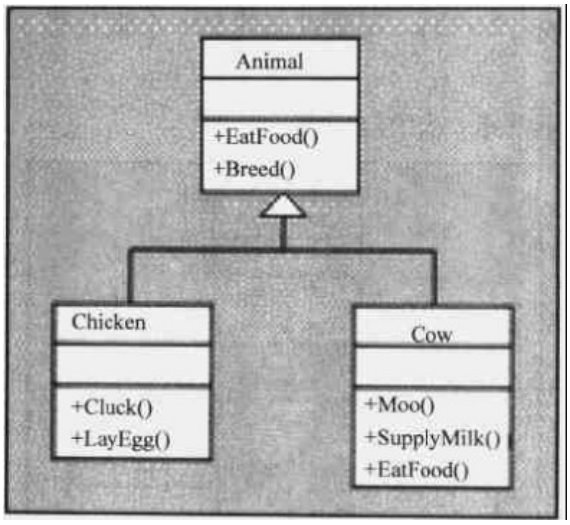


图 8-8

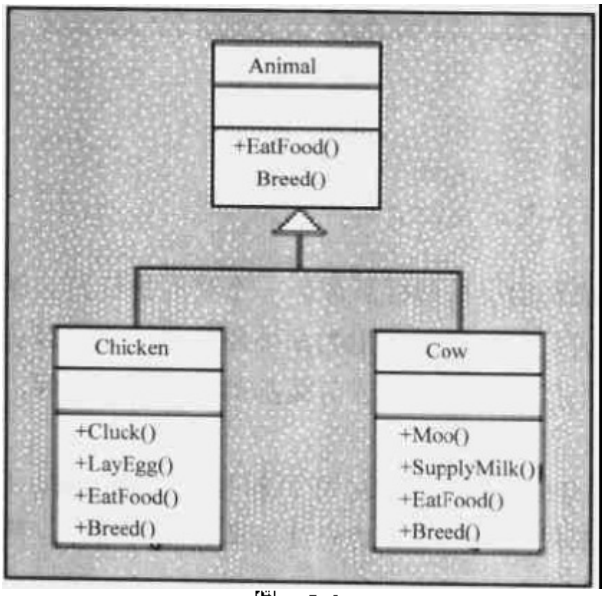


图 8-9

抽象类以斜体名称显示(有时它们的方框有一个短横线)。

最后，类可以是密封的。密封的类不能用作基类，所以也没有派生类。

在 C#中，所有的对象都有一个共同的基类 object(在.NET Framework 中，其别名是 System.Object)。第 9 章将详细介绍这个类。

注意：

如本章前面所述，接口也可以派生于其他接口。与类不同的是，接口可以派生于多个基接口(以类可以支持多个接口的方式派生)。

8.3.3 多态性

继承的一个结果是派生于基类的类在方法和属性上有一定的重叠，因此，可以使用相同的语法处理从同一个基类实例化的对象。例如，如果基类 Animal 有一个方法 EatFood()，则从派

生于它的类 Cow 和 Chicken 中调用这个方法，其语法是类似的：

```
Cow myCow = new Cow();
Chicken myChicken = new Chicken();
myCow.EatFood();
myChicken.EatFood();
```

多态性则更推进了一步。可以把某个基本类型的变量赋予其派生类型的变量，例如：

```
Animal myAnimal = myCow;
```

这不需要进行数据强制类型转换，然后就可以通过这个变量调用基类的方法：

```
myAnimal.EatFood();
```

结果是调用派生类中的 EatFood()执行代码。注意，不能以相同的方式调用派生类上定义的方法，下面的代码不能运行：

```
myAnimal.Moo();
```

但是，可以把基本类型的变量转换为派生类变量，调用派生类的方法，如下所示：

```
Cow myNewCow = (Cow)myAnimal;
myNewCow.Moo();
```

如果原始变量的类型不是 Cow 或派生于 Cow 的类型，这个数据强制类型转换就会产生一个异常。有许多方式说明对象的类型是什么，详见下一章。

在派生于同一个类的不同对象上执行任务时，多态性是一种极为有效的技巧，其使用的代码最少。

注意并不是只有共享同一个父类的类才能利用多态性。只要子类和孙子类在继承层次结构中有一个相同的类，它们就可以用相同的方式利用多态性。

还要注意，在 C#中，所有的类都派生于同一个类 object，object 是继承层次结构中的根。所以可以把所有的对象看作是类 object 的实例。这就是在建立字符串时，Console.WriteLine()可以处理无数多种参数组合的原因。第一个参数后面的每个参数都可以看作是一个 object 实例，所以可以把任何对象的输出结果写到屏幕上。为此，需要调用方法 ToString()(object 的一个成员)，我们可以重写这个方法，为自己的类提供合适的执行代码，或者使用默认的执行代码，返回类名(根据它所在的名称空间，返回类的修饰名)。

接口的多态性

前面介绍的接口概念是组合相关的属性和方法。尽管不能像对象那样实例化接口，但可以建立接口类型的变量，然后就可以在支持该接口的对象上，使用这个变量访问该接口提供的方法和属性。

例如，假定不使用基类 Animal 提供 EatFood()方法，而是把该方法放在接口 IConsume 上。Cow 和 Chicken 类也支持这个接口，惟一的区别是它们必须提供 EatFood()方法的执行代码(因为接口不包含执行代码)，接着就可以使用下述代码访问该方法了：

```
Cow myCow = new Cow();
Chicken myChicken = new Chicken();
IConsume consumeInterface;
consumeInterface = myCow;
consumeInterface.EatFood();
consumeInterface = myChicken;
consumeInterface.EatFood();
```

这就提供了以相同方式访问不同对象的方式，且不依赖于一个公共的基类。在这段代码中，调用 consumeInterface.EatFood()的结果是调用 Cow 或 Chicken 类的方法，这取决于哪个实例被赋予了接口类型的变量。

8.3.4 对象之间的关系

继承是对象之间的一种简单关系，可以让派生类完整地获得基类的特性，而派生类也可以访问基类内部的一些工作代码(通过受保护的成员)。对象之间还有其他一些比较重要的关系。

本节简要讨论下述关系：

- 包含关系——一个类包含另一个类。这类似于继承关系，但包含类可以控制对被包含类的成员的访问，甚至在使用被包含类的成员前进行其他处理。
- 集合关系——一个类用作另一个类的多个实例的容器。这类似于对象数组，但有另一个作用域，包括索引、排序、重新设置大小等。

1. 包含关系

用一个成员字段包含对象实例，就可以实现包含关系。这个成员字段必须是公共字段，此时与继承关系一样，容器对象的用户就可以访问它的方法和属性，但不能像继承关系那样，通过派生类访问类的本机代码。

另外，可以让被包含的成员对象变成私有成员。如果这么做，用户就不能直接访问其成员，即使这些成员是公共的，也不能访问。但可以使用包含类的成员访问这些私有成员。也就是说，可以完全控制被包含的类有什么成员，如果有成员，还可以在访问被包含类的成员前，在包含类的成员上进行其他处理。

例如，Cow 类包含一个 Udder 类，它有一个公共方法 Milk。Cow 对象可以按照要求调用这个方法，作为其 SupplyMilk()方法的一部分，但 Cow 对象的用户看不到这些细节。

在 UML 中，被包含类可以用关联线条来表示。对于简单的包含关系，可以用带有 1 的线条说明一对一的关系(一个 Cow 实例包含一个 Udder 实例)。为了简单起见，也可以把被包含的 Udder 类实例表示为 Cow 类的私有字段，如图 8-10 所示。

2. 集合关系

第 5 章讨论了如何使用数组存储相同类型的多个变量。这也适用于对象(记住，前面使用的变量类型实际上是对象，这没有什么可惊讶的)。例如：

```
Animal[] animals = new Animal[5];
```

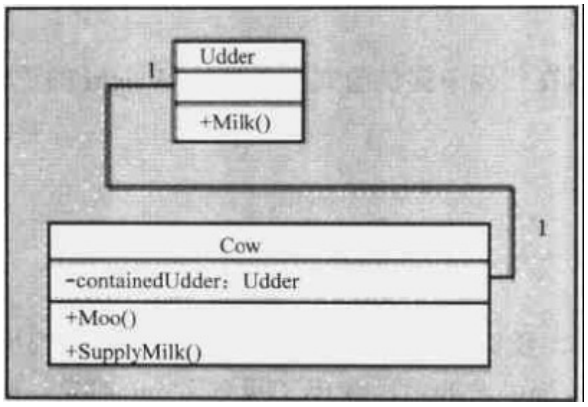


图　8-10

集合基本上是一个带有鸣响的数组，集合以与其他对象相同的方式执行为类。它们通常以所存储的对象名称的复数形式来命名，例如类 Animals 就包含 Animal 对象的一个集合。

数组与集合的主要区别是，集合通常执行额外的功能，例如 Add() 和 Remove()方法可添加和删除集合中的项目。而集合通常有一个 Item 属性，它根据对象的索引返回一个对象。不仅如此，这个属性还允许以更复杂的访问方式来执行。例如，如果设计一个 Animals，Animal 对象就可以根据其名称来访问。

UML 中，用图 8-11 来表示。

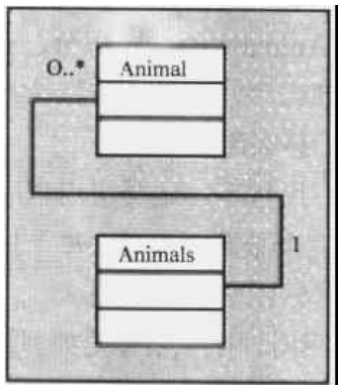


图　8-11

这里遗漏了成员，因为这里描述的是关系。连接线末尾的数字表示一个 Animals 对象可以包含 0 或多个 Animal 对象。

第 11 章详细论述集合。

8.3.5　运算符重载

本书在前面介绍了如何使用运算符处理简单的变量类型。有时也可以把运算符用于从类实例化而来的对象，因为类可以包含运算符如何运算的指令。

例如，给 Animal 添加一个新属性 Weight。接着使用下述代码比较家畜的重量：

```
if (cowA.Weight > cowB.Weight)
{
    ...
```

```
}
```

使用运算符重载，可以在代码中提供隐式使用属性 Weight 的逻辑，例如下面的代码：

```
if (cowA > cowB)
{
   ...
}
```

大于运算符>被重载了。我们为重载运算符编写代码，把它们用作类定义的一部分，而该运算符作用于这个类。在上面的范例中，使用了两个 Cow 对象，所以运算符重载定义包含在 Cow 类中。也可以重载运算符，以相同的方式处理不同的类，其中一个(或两个)类定义包含达到这一目的的代码。

注意只能用这种方式重载现有的 C#运算符，不能创建新的运算符。但是，可以为一元和二元运算符(如+)提供执行代码。

详见第 11 章。

8.3.6 事件

对象可以激活事件，作为它们处理的一部分。事件是非常重要的，可以在代码的其他部分起作用，类似于(但功能更强大)异常。例如，可以在把 Animal 对象添加到 Animals 集合中时执行特定的代码，而这部分代码不是 Animals 类的一部分，也不是调用 Add()方法的代码的一部分。为此，需要给代码添加事件处理程序，这是一种特殊类型的函数，在事件发生时调用。还需要配置这个处理程序，以监听我们感兴趣的事件。

使用事件可以创建事件驱动的应用程序，这类应用程序比读者此时所能想到的多得多。例如，许多基于 Windows 的应用程序完全依赖于事件。每个按钮单击或滚动条拖动操作都是通过事件处理实现的，其中事件是通过鼠标或键盘触发的。

本章的后面将介绍在 Windows 应用程序中事件是如何工作的，第 12 章将深入讨论事件。

8.3.7 引用类型和值类型

在 C#中，数据根据变量的类型以两种方式中的一种存储在一个变量中。变量的类型分为两种：引用类型和值类型，其区别如下：

- 值类型在内存的一个地方(称为堆栈)存储它们自己和它们的内容。
- 引用类型在内存的一个地方(称为堆)存储一个引用，而在另一个地方存储内容。

实际上，在使用 C#时，不必过多地考虑这个问题。到目前为止，所使用的 string 变量(这是引用类型)与使用其他简单变量(大多数是值类型，例如 int)的方式完全相同。

只有一个简单类型是引用类型，即 string 和 object，但数组也是隐式的引用类型。我们创建的每个类都是引用类型，这就是我在这里说明这一点的原因。

结构

还有一个要点：结构类型和类的关键区别是，结构类型是值类型。

你可能认为结构类型和类非常类似，特别是第 6 章介绍了如何在结构类型上使用函数。第 9 章还将探讨这个问题。

8.4　Windows 应用程序中的 OOP

第 2 章介绍了如何在 C#中创建简单的 Windows 应用程序。Windows 应用程序非常依赖于 OOP 技术，本节将论述 OOP 技术，说明本章的一些论点。下面用一个范例来说明。

试试看：使用对象

(1) 在目录 C:\BegCSharp\Chapter8 下调用 Ch08Ex01，创建一个新的 Windows 应用程序。

(2) 使用 Toolbox 栏添加一个新的按钮控件，使之位于 Form1 的中央，如图 8-12 所示。

(3) 双击按钮，为鼠标单击事件添加代码，修改代码，如下所示：

```
        /// <summary>
        /// The main entry point for the application.
        /// </summary>
        [STAThread]
        static void Main()
        {
            Application.Run(new Form1());
        }

        private void button1_Click(object sender, System.EventArgs e)
        {
            ((Button)sender).Text = "Clicked!";
            Button newButton = new Button();
            newButton.Text = "New Button!";
            newButton.Click += new EventHandler(newButton_Click);
            Controls.Add(newButton);
        }

        private void newButton_Click(object sender, System.EventArgs e)
        {
            ((Button)sender).Text = "Clicked!!";
        }
    }
}
```

(4) 运行应用程序，如图 8-13 所示。

(5) 单击标记为 button1 的按钮(见图 8-14)。

(6) 单击标记为 New Button!的按钮，如图 8-15 所示。

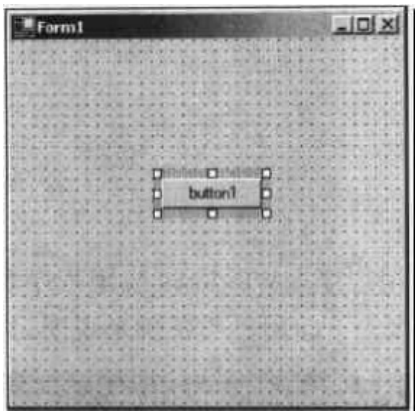


图 8-12

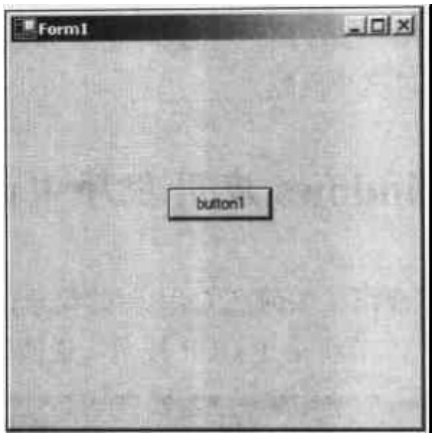


图 8-13

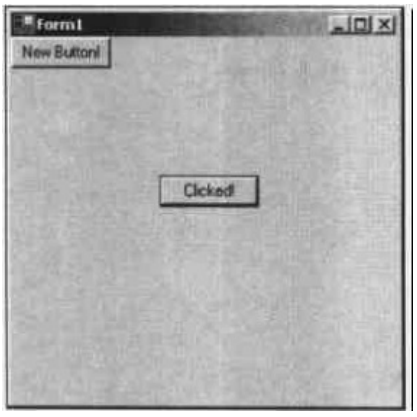


图 8-14

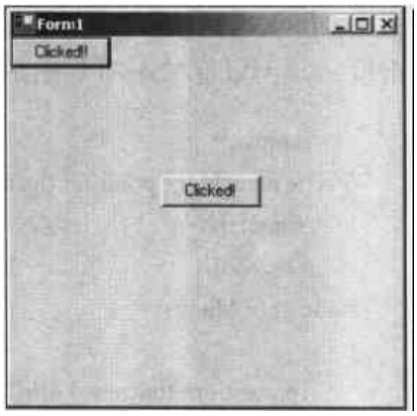


图 8-15

代码的说明

添加几行代码，就创建了一个可以完成某项任务的 Windows 应用程序，同时说明 C#中的一些 OOP 技术。即使在谈到 Windows 应用程序时，谚语“任何东西都是对象”也是正确的。从运行的窗体，到窗体上的控件，都需要使用 OOP 技术。在这个范例中，重点说明本章前面介绍的一些概念，解释如何把它们组合在一起。

在应用程序中，首先是在 Form1 窗体上添加一个新按钮，这个按钮是一个对象，称为 Button。接着双击按钮，添加一个事件处理程序，监听 Button 对象产生的 Click 事件。这个事件处理程序是为封装应用程序的 Form 对象添加到代码中的，是一个私有方法：

```
private void button1_Click(object sender, System.EventArgs e)
{
}
```

这段代码使用 C#关键字 private 作为修饰符。现在不要考虑这个关键字，第 9 章将详细解释本章提及的 OOP 技术。

我们添加的第一行代码改变了按钮上的文本。它利用了本章前面讨论的多态性。表示按钮的 Button 对象作为一个 object 参数发送给事件处理程序，该事件处理程序把参数强制转换为 Button 类型(这可以是，因为 Button 对象继承于 System.Object，System.Object 是一个.NET 类，

object 是其别名)。然后修改对象的 Text 属性，改变显示的文本：

```
((Button)sender).Text = "Clicked!";
```

接着用 new 关键字创建一个新 Button 对象(注意在这个工程中设置了名称空间，因此可以使用这个简单的语法，否则，就需要使用这个对象的完整限定名 System.Windows.Forms.Button)：

```
Button newButton = new Button();
newButton.Text = "New Button!";
```

在代码的其他地方添加一个新的事件处理程序，以响应新按钮产生的 Click 事件：

```
private void newButton_Click(object sender, System.EventArgs e)
{
    ((Button)sender).Text = "Clicked!!";
}
```

接着，使用一些重载运算符语法，把这个事件处理程序注册为 Click 事件的监听程序。同时使用非默认的构造函数创建一个新的 EventHandler 对象，其名称是新事件处理函数的名称：

```
newButton.Click += new EventHandler(newButton_Click);
```

最后，利用窗体的 Controls 集合对象，通过 Add()方法把新按钮添加到窗体上：

```
Controls.Add(newButton);
```

这个小范例几乎使用了本章介绍的所有技术。可以看出，OOP 编程并不复杂——只需要用不同的角度来看待它即可。

8.5　小结

本章完整地描述了面向对象技术。我们在 C#编程环境中进行论述，但主要是用范例来说明。本章介绍的 OOP 大都适用于任何语言。

首先介绍基础知识，例如术语对象的含义，对象如何成为类的实例。接着讨论对象有各种成员，例如字段、属性和方法。这些成员的可访问性都有一定的限制，然后解释了公共和私有成员。之后，说明成员也可以是受保护的，还可以是虚拟和抽象的(抽象方法只能存在于抽象类中)，另外还解释了静态(共享)和实例成员的区别。

接下来，简要介绍了对象的生命周期，包括如何使用构造函数创建对象，如何使用析构函数删除对象。在说明了接口如何组合对象后，介绍了更高级的对象删除方式：支持 IDisposable 接口的删除对象。

本章的其他部分介绍了 OOP 的特性，其中有许多特性将在随后的章节中详细讨论。我们论述了继承(类可以继承基类)，两个版本的多态性(即基类和共享接口)，对象如何用于包含一个或多个其他对象(通过包含和集合关系)。最后，介绍了运算符重载如何用于简化使用对象的语法，对象如何引发事件。

本章的最后一部分用一个 Windows 应用程序范例描述了本章的许多理论。

第 9 章介绍如何使用 C#定义类。

8.6 练习

(1) 下述哪个项目在 OOP 中有真实级别的可访问性？

- 友元
- 公共
- 安全
- 私有
- 受保护的
- 松散的
- 通配符

(2) “必须手动调用对象的析构函数，否则就会浪费资源”，对吗?

(3) 需要创建一个对象，以调用其类的静态方法吗？

(4) 绘制一个类似于本章中的 UML 图，其中包含下述类和接口：

- 抽象类 HotDrink，它有方法 Drink(), AddMilk()和 AddSugar()，以及属性 Milk 和 Sugar。
- 接口 ICup，它有方法 Refill()和 Wash()，以及属性 Color 和 Volume。
- 派生于 HotDrink 的类 CupOfCoffee，支持 ICup 接口，还有一个属性 BeanType。
- 派生于 HotDrink 的类 CupOfTea 支持 ICup 接口，还有一个属性 LeafType。

(5) 为一个函数编写一些代码，接受上述范例的两个杯子对象中的一个，作为一个参数。该函数应可以为它传送的任何杯子对象调用 AddMilk()，Drink()和 Wash()方法。

第9章 定 义 类

第 8 章介绍了面向对象编程(OOP)的特性，本章则要把理论付诸实践，看看如何在 C#中定义类。

本章并不讨论如何定义类的成员，而重点讨论了如何定义类本身。这听起来有一定的限制，但不必担心，本章的内容足够读者认真学习的了。

首先看看基本的类定义语法、用于确定类可访问性的关键字，以及指定继承的方式。我们还将介绍接口的定义，因为它们在许多方面都类似于类的定义。

本章的其他部分介绍在 C#中定义类时涉及到的其他主题，包括：

- System.Object 类
- Visual Studio.NET (VS)提供的帮助工具
- 类库
- 接口和抽象类的比较
- 结构类型
- 复制对象

下面就开始学习基础知识。

9.1 C#中的类定义

C#使用 class 关键字来定义类。其基本结构如下：

```
class MyClass
{
    // class members
}
```

这段代码定义了一个类 MyClass。定义了一个类后，就可以在能访问该定义的工程的其他地方对该类进行实例化。在默认情况下，类声明为内部的，即只有当前工程中的代码才能访问它。可以用 internal 访问修饰符关键字显式指定，如下所示(但这是不必要的)：

```
internal class MyClass
{
    // class members
}
```

另外，还可以指定类是公共的，应可以由其他工程中的代码来访问。为此，要使用关键字 public。

```
public class MyClass
{
   // class members
 }
```

注意：

以这种方式声明的类不能是私有或受保护的。可以把这些声明类的修饰符用于声明类成员，详见下一章。

除了这两个访问修饰符关键字外，还可以指定类是抽象的(不能实例化，只能继承，可以有抽象成员)或密封的(sealed，不能继承)。为此，可以使用两个互斥的关键字 abstract 或 sealed。所以，抽象类必须用下述方式声明：

```
public abstract class MyClass
{
   // class members, may be abstract
}
```

其中 MyClass 是一个公共抽象类，也可以是内部抽象类。

密封类的声明如下所示：

```
public sealed class MyClass
{
   // class members
}
```

与抽象类一样，密封类也可以是公共或内部的。

还可以在类定义中指定继承。为此，要在类名的后面加上一个冒号，其后是基类名，例如：

```
public class MyClass : MyBase
{
   // class members
}
```

注意，在 C#的类定义中，只能有一个基类，如果继承了一个抽象类，就必须执行所继承的所有抽象成员(除非派生类也是抽象的)。

编译器不允许派生类的可访问性比其基类更高。也就是说，内部类可以继承于一个公共类，但公共类不能继承于一个内部类。因此，下述代码是合法的：

```
public class MyBase
{
   // class members
}

internal class MyClass : MyBase
{
```

```
    // class members
}
```

但下述代码不能编译：

```
internal class MyBase
{
    // class members
}

public class MyClass : MyBase
{
    // class members
}
```

如果没有使用基类，则被定义的类就只继承于基类 System.Object(它在 C#中的别名是 object)。毕竟，在继承层次结构中，所有类的根都是 System.Object，稍后将详细介绍这个基类。

除了以这种方式指定基类外，还可以在冒号的后面指定支持的接口。如果指定了基类，它必须紧跟在冒号的后面，之后才是指定的接口。如果没有指定基类，则接口就跟在冒号的后面。使用逗号分隔基类名(如果有基类)和接口名。

例如，给 MyClass 添加一个接口，如下所示：

```
public class MyClass : IMyInterface
{
    // class members
}
```

所有的接口成员都必须在支持该接口的类中执行，但如果不想使用给定的接口成员，就可以提供一个“空”的执行方式(没有函数代码)。

下面的声明是无效的，因为基类 MyBase 不是继承列表中的第一个项：

```
public class MyClass : IMyInterface, MyBase
{
    // class members
}
```

指定基类和接口的正确方式如下：

```
public class MyClass : MyBase, IMyInterface
{
    // class members
}
```

可以指定多个接口，所以下面的代码是有效的：

```
public class MyClass : MyBase, IMyInterface, IMySecondInterface
{
```

```
    // class members
}
```

表 9-1 是类定义中可以使用的访问修饰符组合。

表 9-1

修 饰 符	含 义
none or internal	类只能在当前工程中访问
Public	类可以在任何地方访问
abstract or internal abstract	类只能在当前工程中访问，不能实例化，只能继承
public abstract	类可以在任何地方访问，不能实例化，只能继承
sealed or internal sealed	类只能在当前工程中访问，不能派生，只能实例化
public sealed	类可以在任何地方访问，不能派生，只能实例化

接口的定义

接口声明的方式与声明类的方式相似，但使用的是关键字 interface，而不是 class，例如：

```
interface IMyInterface
{
    // interface members
}
```

访问修饰符关键字 public 和 internal 的使用方式是相同的，所以要使接口的访问是公共的，必须使用 public 关键字：

```
public interface IMyInterface
{
    // interface members
}
```

关键字 abstract 和 sealed 不能在接口中使用，因为这两个修饰符在接口定义中是没有意义的(它们不包含执行代码，所以不能直接实例化，且必须是可以继承的)。

接口的继承也可以用与类继承的类似方式来指定。主要的区别是可以使用多个基接口，例如：

```
public interface IMyInterface : IMyBaseInterface, IMyBaseInterface2
{
    // interface members
}
```

接口继承 System.Object 的方式与类相同。这是通过接口多态性的机制实现的。但是，如上

所述，不能用实例化类的方式来实例化接口。

下面看一个类定义的范例，其中有一些代码使用了这些类。

试试看：定义类

(1) 在目录 C:\BegCSharp\Chapter9 下创建一个新的控制台应用程序 Ch09Ex01。

(2) 修改 Class1.cs 中的代码，如下所示：

```
namespace Ch09Ex01
{
    public abstract class MyBase
    {
    }

    internal class MyClass : MyBase
    {
    }

    public interface IMyBaseInterface
    {
    }

    internal interface IMyBaseInterface2
    {
    }

    internal interface IMyInterface : IMyBaseInterface, IMyBaseInterface2
    {
    }

    internal sealed class MyComplexClass : MyClass, IMyInterface
    {
    }

    /// <summary>
    /// Summary description for Class1.
    /// </summary>
    class Class1
    {
        static void Main(string[] args)
        {
            MyComplexClass myObj = new MyComplexClass();
            Console.WriteLine(myObj.ToString ());
        }
    }
}
```

(3) 执行工程，如图 9-1 所示。



图　9-1

代码的说明

这个工程在下面的继承层次结构中定义了类和接口，如图 9-2 所示。

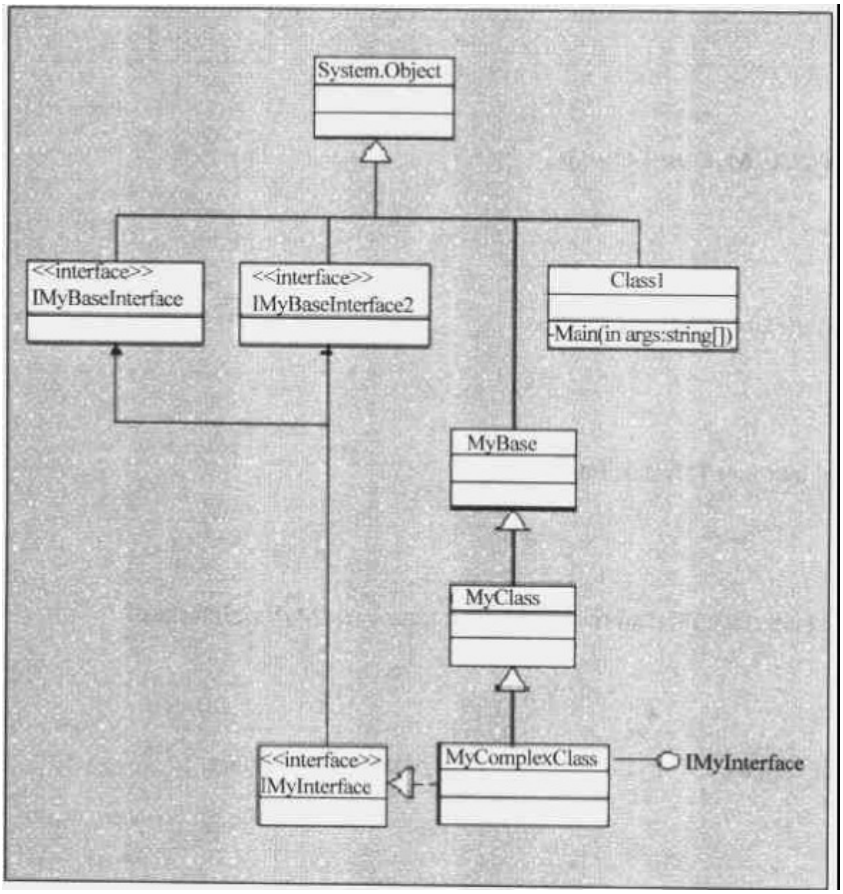


图　9-2

这里包含 Class1，是因为这个类的定义方式与其他类的定义方式相同，而它不是主要类层次结构中的一部分。与本书前面一样，这个类处理的 Main()方法是应用程序的入口点。

MyBase 和 IMyBaseInterface 被定义为公共的，所以它们可以在其他工程中使用。其他类和接口都是内部的，只能在这个工程中使用。

代码调用 myObj 的 ToString()方法，即 MyComplexClass 的一个实例：

```
MyComplexClass myObj = new MyComplexClass();
Console.WriteLine(myObj.ToString());
```

这是继承 System.Object 的一种方法(图中没有显示，因为我省略了这个类的成员)，并把对象的类名作为一个字符串返回，该类名用所有相关的名称空间来限定。

9.2 System.Object

因为所有的类都继承于 System.Object，所以这些类都可以访问该类中受保护的和公共的成员。下面看看可以使用的成员有哪些。System.Object 包含的方法如表 9-2 所示。

表 9-2

方 法	返回类型	虚 拟	静 态	描 述
Object()	N/A	无	无	System.Object 类的构造函数，由派生类的构造函数自动调用
~Object()（也称为 Finalize()，参见下一节）	N/A	无	无	System.Object 类的析构函数，由派生类的构造函数自动调用，不能手动调用
Equals(object)	bool	Yes	无	把调用该方法的对象与另一个对象相比，如果它们相等，就返回 true。以默认的执行方式进行检查，以查看对象的参数是否引用了同一个对象(因为对象是引用类型)。如果想以不同的方式来比较对象，可以重写该方法，(如果这两个对象有相同的值)
Equals (object, object)	bool	无	有	这个方法比较传送给它的两个对象，看看它们是否相等。检查时使用了 Equals (object) 方法。注意，如果两个对象都是空引用，这个方法会返回 true
ReferenceEquals (object,object)	Bool	无	有	这个方法比较传送给它的两个对象，看看它们是否是同一个实例的引用
ToString()	String	有	无	返回一个对应于对象实例的字符串。在默认情况下，这是一个类的修饰名(参见前面的范例)，但它可以被重写，以便给类提供合适的实现方式
Memberwise Clone()	object	无	无	通过创建一个新对象实例并复制成员，以复制该对象。注意成员拷贝不会得到这些成员的新实例。新对象的任何引用类型成员都将引用与源类相同的对象，这个方法是受保护的，所以只能在类或派生的类中使用
GetType()	System.Type	无	无	以 System.Type 对象的形式返回对象的类型
GetHashCode()	int	有	无	用作对象的散列函数，这是一个必选函数，返回一个以压缩形式标识的对象的值

这些方法是.NET Framework中对象类型必须支持的基本方法，但我们可以从不使用它们(或者在特殊情况下使用，例如 GetHashCode())。

在利用多态性时，GetType()是一个有用的方法，允许根据对象的类型来执行不同的代码，而不是像通常那样，对所有的对象都执行相同的代码。例如，如果函数接受一个 Object 类型的参数(表示可以给该函数传送任何信息)，就可以在遇到某些对象时执行额外的任务。联合使用 GetType() 和 typeof()(这是一个 C#运算符，可以把类名转换为 System.Type 对象)，就可以进行比较，如下所示：

```
if (typeof(myObj) == typeof(MyComplexClass))
{
   // myObj is an instance of the class MyComplexClass
}
```

返回的 System.Type 对象可以做更多的工作，这里不讨论它们。反射详见第 22 章。

重写 ToString()方法也是非常有效的，特别是在对象的内容中可以用一个人类所能够理解的字符串表示时，就更是如此。

后面的章节将讨论这些 System.Object 方法，所以现在就讨论到这，后面在需要时再详细讨论。

9.3 构造函数和析构函数

在 C#中定义类时，常常不需要定义相关的构造函数和析构函数，因为基类 System.Object 提供了一个默认的实现方式。但是，如果需要，可以提供我们自己的构造函数和析构函数，以便初始化对象和清理对象。

使用下述语法把简单的构造函数添加到一个类中：

```
class MyClass
{
   public MyClass()
   {
      // Constructor code
   }
}
```

这个构造函数与包含它的类同名，且没有参数(使之成为类的默认构造函数)，这是一个公共函数，所以类的对象是可以实例化的(详见第 8 章)。

也可以使用私有的默认构造函数，即这个类的对象实例不能用这个构造函数来创建(详见第 8 章)：

```
class MyClass
{
   private MyClass ()
```

```
    {
        // Constructor code
    }
}
```

也可以用相同的方式给类添加非默认的构造函数，方法是提供参数，例如：

```
class MyClass
{
    public MyClass()
    {
        // Default constructor code
    }

    public MyClass(int myint)
    {
        // Non-default constructor code (uses myInt)
    }
}
```

可提供的构造函数的个数是没有限制的。

析构函数使用略微不同的语法来声明。在.NET 中使用的析构函数(由 System.Object 类提供)叫作 Finalize()，但这不是我们用于声明析构函数的名称。使用下面的代码，而不是重写 Finalize()：

```
class MyClass
{
    ~MyClass()
    {
        // destructor body
    }
}
```

当进行无用存储单元收集时，就执行析构函数中的代码，释放资源。在调用这个析构函数后，还将隐式地调用基类的析构函数，包括 System.Object 根类中的 Finalize()调用。这个技术可以让.NET Framework 确保调用 Finalize()，因为重写 Finalize()是指基类调用需要显式地执行，这是有潜在危险的(第 10 章将详细讨论如何调用基类的方法)。

构造函数的执行序列

如果在类的构造函数中执行多个任务，把这些代码放在一个地方是非常方便的，这与本书前面论述的把代码放在函数中有相同的优势。使用一个方法就可以把代码放在一个地方(详见第 10 章)，而 C#提供了一个更好的方式。任何构造函数都可以进行配置，以便在执行自己的代码前调用其他构造函数。

在讨论构造函数前，先看看在默认情况下，创建类的实例时会发生什么情况。

为了实例化派生的类，必须实例化它的基类。而要实例化这个基类，又必须实例化这个基类的基类，这样一直到实例化 System.Object 为止。结果是无论使用什么构造函数实例化一个类，总是要先调用 System.Object.Object()。

如果对一个类使用非默认的构造函数，默认的情况是在其基类上使用匹配于这个构造函数签名的构造函数。如果没有找到这样的构造函数，就使用基类的默认构造函数(根类 System.Object 总是要使用默认的构造函数，因为这个类没有非默认的构造函数)。下面介绍一个小范例，说明一系列事件。考虑下面的对象结构：

```
public class MyBaseClass
{
    public MyBaseClass()
    {
    }

    public MyBaseClass(int i)
    {
    }
}

public class MyDerivedClass : MyBaseClass
{
    public MyDerivedClass()
    {
    }

    public MyDerivedClass(int i)
    {
    }

    public MyDerivedClass(int i, int j)
    {
    }
}
```

如果以下面的方式实例化 MyDerivedClass：

```
MyDerivedClass myObj = new MyDerivedClass();
```

则发生下面的一系列事件：

- 执行 System.Object.Object()构造函数。
- 执行 MyBaseClass.MyBaseClass()构造函数。
- 执行 MyDerivedClass.MyDerivedClass()构造函数。

另外，如果使用下面的语句：

```
MyDerivedClass myObj = new MyDerivedClass(4);
```

则发生下面的一系列事件：

- 执行 System.Object.Object()构造函数。
- 执行 MyBaseClass.MyBaseClass(int i) 构造函数。
- 执行 MyDerivedClass.MyDerivedClass(int i) 构造函数。

最后，如果使用下面的语句：

```
MyDerivedClass myObj = new MyDerivedClass(4, 8);
```

则发生下面的一系列事件：

- 执行 System.Object.Object()构造函数。
- 执行 MyBaseClass.MyBaseClass()构造函数。
- 执行 MyDerivedClass.MyDerivedClass(int i, int j) 构造函数。

这个系统会正常工作，并确保派生类中的构造函数可以访问继承的成员。但是，有时需要对发生的事件进行更多的控制。例如，在上面的实例化范例中，需要有下面的事件序列：

- 执行 System.Object.Object()构造函数。
- 执行 MyBaseClass.MyBaseClass(int i) 构造函数。
- 执行 MyDerivedClass.MyDerivedClass(int i, int j) 构造函数。

使用这个序列可以编写在 MyBaseClass(int i)中使用 int i 参数的代码，即 MyDerivedClass(int i, int j)构造函数要做的工作比较少，只需要处理 int j 参数(假定 int i 参数在两种情况下有相同的含义，虽然事情并非总是如此，但实际上我们常常做这样的安排)。只要愿意，C#就可以指定这种操作。

为此，只需指定在派生类的构造函数定义中所使用的基类的构造函数，如下所示：

```
public class MyDerivedClass : MyBaseClass
{
    ...

    public MyDerivedClass(int i, int j) : base(i)
    {
    }
}
```

其中，base 关键字指定.NET 实例化过程，以使用基类中匹配指定签名的构造函数。这里使用了一个 int 参数，所以应使用 MyBaseClass(int i)。这么做将不调用 MyBaseClass()，而是执行本例前面列出的事件序列——也就是我们希望执行的事件序列。

也可以使用这个关键字指定基类构造函数的字面值，例如使用 MyDerivedClass 的默认构造函数调用 MyBaseClass 非默认的构造函数：

```
public class MyDerivedClass : MyBaseClass
{
    public MyDerivedClass() : base(5)
    {
```

```
    }

    ...
}
```

这段代码将执行下述序列：

- 执行 System.Object.Object()构造函数。
- 执行 MyBaseClass.MyBaseClass(int i) 构造函数。
- 执行 MyDerivedClass.MyDerivedClass()构造函数。

除了 base 关键字外，这里还可以使用另一个关键字 this。这个关键字指定在调用指定的构造函数前，.NET 实例化过程对当前类使用非默认的构造函数。例如：

```
public class MyDerivedClass : MyBaseClass
{
    public MyDerivedClass() : this(5, 6)
    {
    }

    ...

    public MyDerivedClass(int i, int j) : base(i)
    {
    }
}
```

这段代码将执行下述序列：

- 执行 System.Object.Object()构造函数。
- 执行 The MyBaseClass.MyBaseClass(int i)构造函数。
- 执行 The MyDerivedClass.MyDerivedClass(int i, int j)构造函数。
- 执行 The MyDerivedClass.MyDerivedClass()构造函数。

惟一的限制是使用 this 或 base 关键字只能指定一个构造函数。但是，如上一个范例所示，这并不是一个很严厉的限制，因为我们仍可以构造相当复杂的执行序列。

本书的后面将讨论这个技术。

9.4 Visual Studio.NET 中的 OOP 工具

OOP 在.NET Framework 中是一个非常基础的主题，VS 提供了几个工具来帮助开发 OOP 应用程序。本节就介绍其中的一些工具。

9.4.1 Class View 窗口

第 2 章介绍了 Solution Explorer 窗口与 Class View 窗口共用相同的空间。这个窗口显示了

应用程序中的类层次结构，可以查看我们使用的类的特性。查看这些信息有几个不同的模式，默认的模式是 Sort By Type。对于上一节的范例工程，其视图如图 9-3 所示。

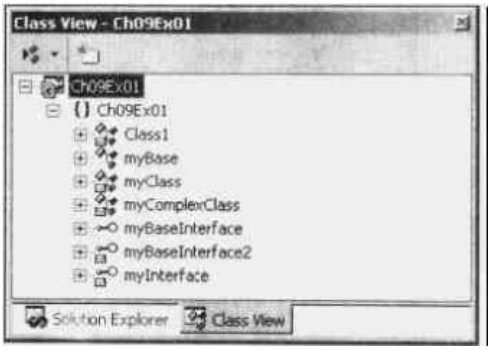


图 9-3

这里使用了许多符号，包括：

- ——工程
- {}——名称空间
- ——类
- ——接口
- ——方法
- ——属性
- ——域
- ——结构
- ——枚举
- ——枚举项目
- ——事件

注意其中一些图标用于类型定义，而不是类定义，例如枚举和结构类型。

其中一些项目还可以在它们的下面放置其符号，表示它们的访问级别(公共项目没有这样的符号)：

- ——私有
- ——受保护的
- ——内部

没有符号用于表示抽象、密封和虚拟项目。

所有的可用模式基本上以相同的方式工作，使用标准树形视图控件就可以展开类的定义。展开类和接口，一直到 System.Object，则得到图 9-4 所示的结果。

在该窗口中，可以看到工程中的所有类信息。

在这里除了可以查看信息外，还可以访问许多项目的相关代码。双击某个项目，或者右击该项目，并选择 Go to Definition，就可以查看到工程中用于定义该项目的代码(此时代码是可以查看的)。如果不能查看代码，例如不能访问基类 System.Object 中的代码，就应考虑使用 Object Browser 视图。

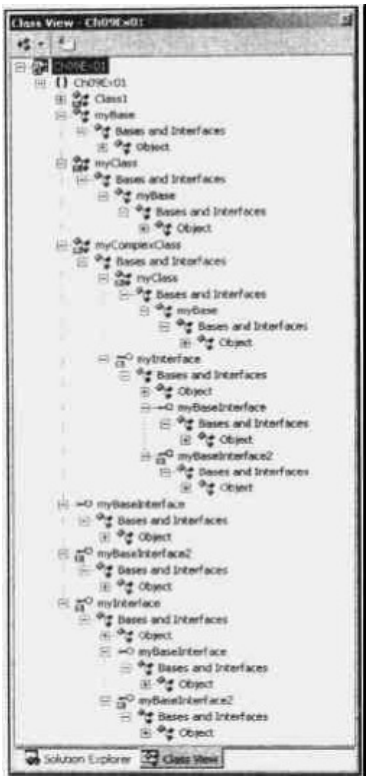


图　9-4

9.4.2　对象浏览器

对象浏览器(Object Browser)是 Class View 窗口的扩展版本，可以查看工程中能使用的其他类，甚至可以查看外部的类。可以自动(例如上一节的情况)或手动(通过 View | Other Windows | Object Browser)进入这个窗口。这个视图显示在主窗口中，可以用与 Class View 窗口相同的方式浏览该视图，如图 9-5 所示。

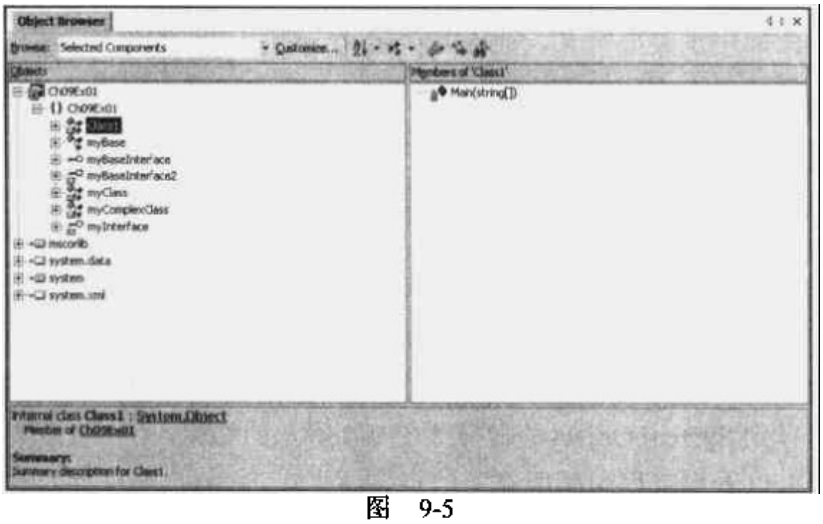


图　9-5

与 Class View 不同，这个窗口在不同的地方显示了类和类的成员，包括工程引用的所有.NET 模块。例如，我们可以使用这个工具浏览 System 名称空间中的项目。

在选择了一个项目后，就在窗口的下面显示该项目的信息，包括访问级别、基类和 Class1 的名称空间。也可以使用这些信息进行导航——例如，单击 System.Object 可以直接查看这个类

的信息。这里还可以查看一些 Summary 信息，它们是由代码中的 XML 文档注释生成的：

```
/// <summary>
/// Summary description for Class1.
/// </summary>
class Class1
{
    static void Main(string[] args)
    {
        MyComplexClass myObj = new MyComplexClass();
        Console.WriteLine(myObj.ToString());
    }
}
```

第 18 章将详细讨论 XML 文档注释。

9.4.3 添加类

VS 包含可以加速执行某些常见任务的工具，其中一些可以应用于 OOP。有一个工具可以给工程快速添加新类，且需要的键入量最少。

该工具的访问方式是单击 File | Add New Item...菜单项或在 Solution Explorer 窗口中右击工程，选择相应的项目。采用这两种方式，都会打开一个对话框，如图 9-6 所示，在该对话框中，可以选择要添加的项目类型。要添加一个类，可以在窗口的右边选择 Class 项目，为包含类的文件提供一个文件名，再单击 Open。所创建的类就以所选择的文件名命名。

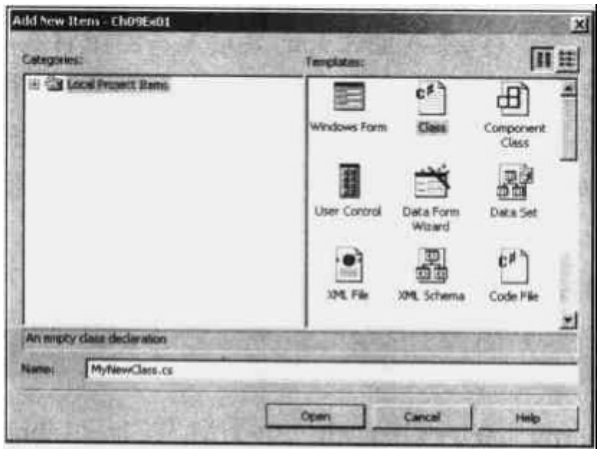


图 9-6

在本章前面的范例中，我们在 Class1.cs 文件中手动添加类定义。把类放在不同的文件中，常常可以更轻松地跟踪类。

打开 Ch09Ex01 工程后，在上面的对话框中输入信息，这样就会在 MyNewClass.cs 中生成下述代码：

```
using System;
```

```
namespace Ch09Ex01
{
    /// <summary>
    /// Summary description for MyNewClass.
    /// </summary>
    public class MyNewClass
    {
        public MyNewClass()
        {
            //
            // TODO: Add constructor logic here
            //
        }
    }
}
```

这个类 MyNewClass 定义在入口点类 Class1 所在的名称空间中，所以可以在代码中使用它，就像它们是在相同的文件中定义一样。

从代码中可以看出(或者更明确地说，是代码中的注释)，生成的类包含了一个默认的构造函数。

9.5 类库工程

除了在工程中把类放在不同的文件中之外，还可以把它们放在完全不同的工程中。如果一个工程什么都不包含，只包含类(以及其他相关的类定义，但没有入口点)，该工程就称为类库。

类库工程编译为.dll 装配件，在其他工程中添加对类库工程的引用，就可以访问它的内容(这是同一个解决方案的一部分，但这不是必须的)。这将扩展对象提供的封装性，因为类库可以进行修改和更新，而不会影响使用它们的其他工程，以升级类提供的服务(这会影响多个用户应用程序)。

下面看一个类库工程的范例和一个利用该工程包含的类的工程。

试试看：使用类库

(1) 在目录 C:\BegCSharp\Chapter9 下创建一个类型为 Class Library 的新工程 Ch09ClassLib，如图 9-7 所示。

(2) 把文件 Class1.cs 重命名为 MyExternalClass.cs(也可以在 Solution Explorer 窗口中右击该文件，选择 Rename 来重命名该名件)。

(3) 修改 MyExternalClass.cs 中的代码，以反映类名的改变：

```
public class MyExternalClass
{
    public MyExternalClass()
    {
```

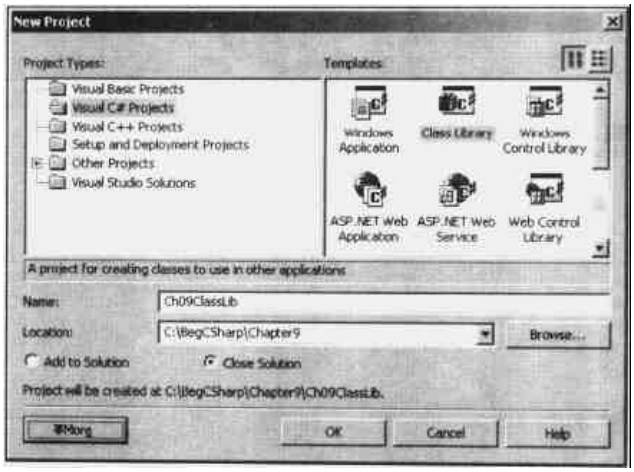


图 9-7

```
            //
            // TODO: Add constructor logic here
            //
        }
    }
```

(4) 使用文件名 MyInternalClass.cs 给工程添加一个新类。

(5) 修改代码，使类 MyInternalClass 成为内部类：

```
internal class MyInternalClass
{
    public MyInternalClass()
    {
        //
        // TODO: Add constructor logic here
        //
    }
}
```

(6) 编译工程(注意这个工程没有入口点，所以不能像往常那样运行它——可以选择 Build | Build 菜单项来构建它)。

(7) 在目录 C:\BegCSharp\Chapter9 下创建一个新的控制台应用程序工程 Ch09Ex02。

(8) 选择 Project | Add Reference...菜单项，或者在 Solution Explorer 窗口中右击 References，选择相同的选项。

(9) 单击 Browse...按钮，导航到 C:\BegCSharp\Chapter9\Ch09ClassLib\bin\Debug\，双击 Ch09ClassLib.dll。

(10) 单击 OK 按钮。

(11) 完成了上述操作后，检查引用是否已被添加到 Solution Explorer 窗口中，如图 9-8 所示。



图 9-8

(12) 打开 Object Browser 窗口，检查新引用，看看其中包含的对象，其结果如图 9-9 所示。

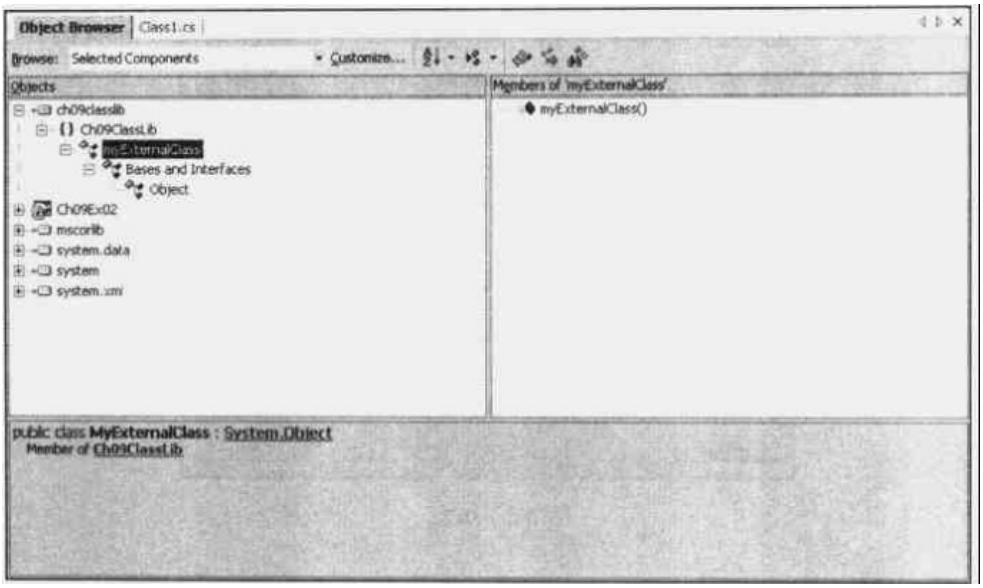


图　9-9

(13) 修改 Class1.cs 中的代码，如下所示：

```
using System;
using Ch09ClassLib;

namespace Ch09Ex02
{
    /// <summary>
    /// Summary description for Class1.
    /// </summary>
    class Class1
    {
        static void Main(string[] args)
        {
            MyExternalClass myObj = new MyExternalClass();
            Console.WriteLine(myObj.ToString());
        }
    }
}
```

(14) 运行应用程序，其结果如图 9-10 所示。



图　9-10

代码的说明

在这个范例中，创建了两个工程，一个是类库工程，另一个是控制台应用程序工程。类库工程 Ch09ClassLib 包含两个类 MyExternalClass(可公开访问)和 MyInternalClass(只能在内部访

问)。控制台应用程序 Ch09Ex02 包含利用类库工程的简单代码。

为了使用 Ch09ClassLib 中的类，在应用程序 Ch09Ex02 中添加了一个由类库工程所生成的 DLL 的引用。该引用指向相关的 DLL。对于这个范例，该引用是指向类库的输出文件，但这仅仅是把这个文件复制到 Ch09Ex02 的本地位置上，以便继续开发类库，而不影响控制台应用程序。为了用新类库工程替换旧版本的装配件，只需用新生成的 DLL 文件覆盖旧文件即可。

在添加了引用后，就可以使用对象浏览器查看可用的类。因为两个类中的一个(MyInternalClass)是内部的，所以在对象浏览器窗口中看不到这个类——它不能由外部的工程访问。但是，另一个类(MyExternalClass)是可以访问的，这是我们在控制台应用程序中使用的类。

可以把控制台应用程序中的代码替换为使用内部类的代码，如下所示：

```
static void Main(string[] args)
{
    MyInternalClass myObj = new MyInternalClass();
    Console.WriteLine(myObj.ToString());
}
```

如果试图编译这段代码，就会产生如下编译错误：

```
C:\BegCSharp\Chapter9\Ch09Ex02\Class1.cs(13): 'Ch09ClassLib.MyInternalClass' is inaccessible due to its protection level
```

利用外部装配件中的类的技术是使用 C#和.NET Framework 编程的关键。实际上，在使用.NET Framework 中的任何类时，也就是在利用外部装配件中的类，因为它们的处理方式是相同的。

9.6 接口和抽象类

本章介绍了如何创建接口和抽象类(现在不考虑其成员，下一章会论述类的成员)。这两种类型在许多方面都很类似，所以应看看它们的相似和不同之处，看看哪些情况应使用什么技术。首先讨论它们的类似之处。

抽象类和接口都包含可以由派生类继承的成员。接口和抽象类都不能直接实例化，但可以声明它们的变量。如果这样做，就可以使用多态性把继承这两种类型的对象指定给它们的变量。接着通过这些变量来使用这些类型的成员，但不能直接访问派生对象的其他成员。

下面看看它们的区别。

派生类只能继承一个基类，即只能直接继承一个抽象类(但可以用一个继承链包含多个抽象类)。相反，类可以使用任意多个接口。但这不会产生太大的区别——这两种情况得到的效果是类似的。只是采用接口的方式略有不同。

抽象类可以拥有抽象成员(没有代码体，且必须在派生类中执行，否则派生类本身必须也是抽象的)和非抽象成员(它们拥有代码体，也可以是虚拟的，这样就可以在派生类中重写)。另一方面，接口成员必须都在使用接口的类上执行——它们没有代码体。另外，接口成员被定义为公共的(因为它们倾向于在外部使用)，但抽象类的成员也可以是私有的(只要它们不是抽象的)、

受保护的、内部的或受保护的内部成员(其中受保护的内部成员只能在应用程序的代码或派生类中访问)。此外，接口不能包含字段、构造函数、析构函数、静态成员或常量。

这说明这两种类型用于完全不同的目的。抽象类主要用作对象系列的基类，共享某些主要特性，例如共同的目的和结构。接口则主要由类来使用，其中这些类在基础水平上有所不同，但仍可以完成某些相同的任务。

例如，假定有一个对象系列表示火车，基类 Train 包含火车的核心定义，例如车轮的规格和引擎的类型(可以是蒸汽发动机、柴油发动机等)。但这个类是抽象的，因为并没有“一般的”火车。为了创建一辆实际的火车，需要给该火车添加特性。为此，派生一些类，例如：Passenger Train, FreightTrain 和 424DoubleBogey 等，如图 9-11 所示。

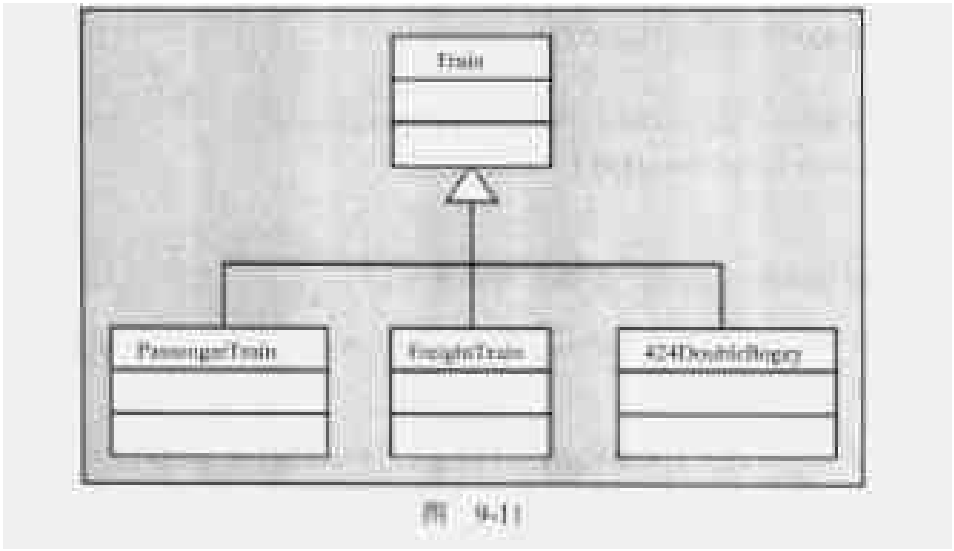


图 9-11

汽车对象系列也可以用相同的方式来定义，使用 Car 抽象基类，其派生类有 Compact, SUV 和 PickUp。Car 和 Train 甚至可以派生于一个相同的基类 Vehicle，如图 9-12 所示。

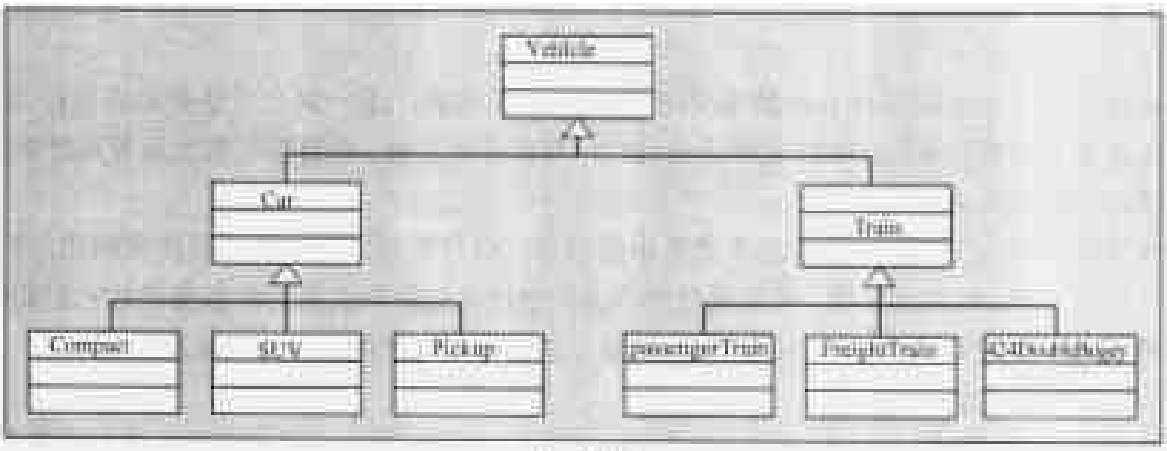


图 9-12

现在，层次结构中的一些类共享相同的特性，这是因为它们的目的是相同的，而不是因为它们派生于相同的基类。例如，PassengerTrain, Compact, SUV 和 Pickup 都可以运送乘客，所以它们都拥有 IPassengerCarrier 接口，FreightTrain 和 PickUp 可以运送货物，所以它们都拥有 IHeavyLoadCarrier 接口，如图 9-13 所示。

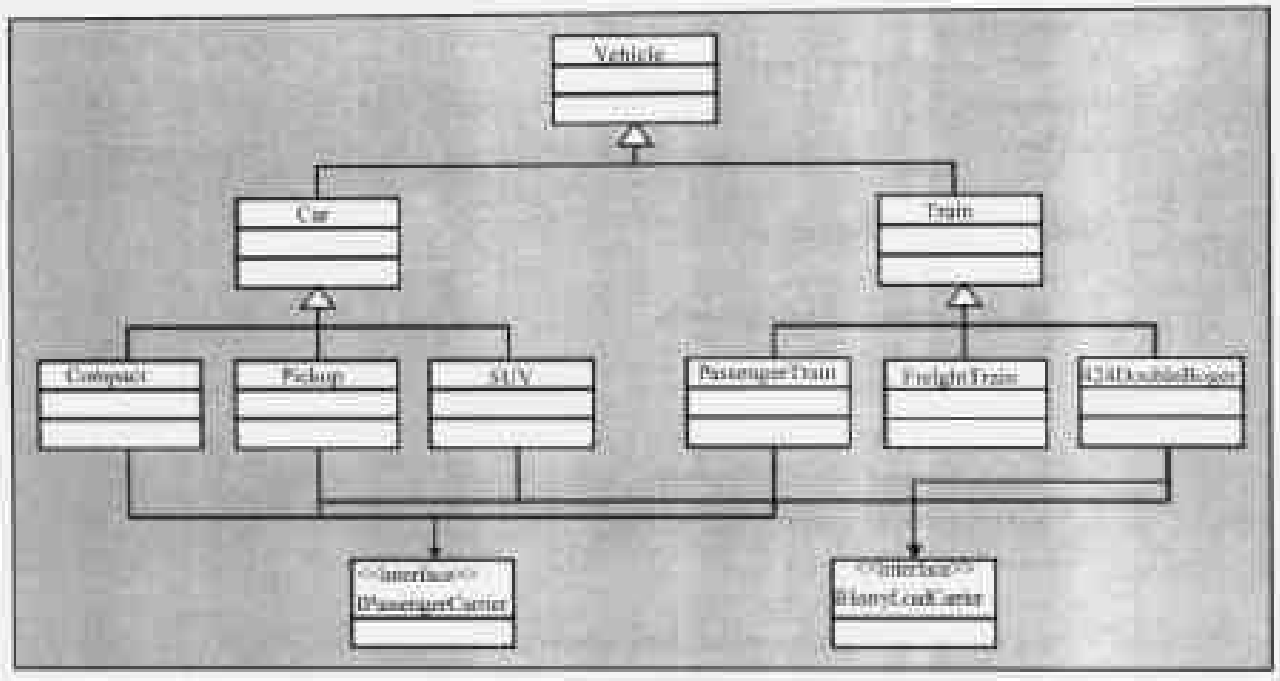


图 9-13

在进行更详细的分工前，把对象系统以这种方式进行分解，可以清楚地看到哪种情形适合使用抽象类，哪种情形适合使用接口。只使用接口或只使用抽象继承，就得不到这个范例的结果。

9.7 结构类型

第 8 章提到过结构和类非常相似，但结构是值类型，而类是引用类型。这意味着什么？最简单的方式是用一个范例来说明。

试试看：类和结构

(1) 在目录 C:\BegCSharp\Chapter9 下创建一个新控制台应用程序工程 Ch09Ex03。

(2) 修改代码，如下所示：

```
namespace Ch09Ex03
{
    class MyClass
    {
        public int val;
    }

    struct myStruct
    {
        public int val;
    }
```

```
    /// <summary>
    /// Summary description for Class1.
    /// </summary>
    class Class1
    {
        static void Main(string[] args)
        {
            MyClass objectA = new MyClass();
            MyClass objectB = objectA;
            objectA.val = 10;
            objectB.val = 20;
            myStruct structA = new myStruct();
            myStruct structB = structA;
            structA.val = 30;
            structB.val = 40;
            Console.WriteLine("objectA.val = {0}", objectA.val);
            Console.WriteLine("objectB.val = {0}", objectB.val);
            Console.WriteLine("structA.val = {0}", structA.val);
            Console.WriteLine("structB.val = {0}", structB.val);
        }
    }
}
```

(3) 运行应用程序，其结果如图 9-14 所示。



图　9-14

代码的说明

这个应用程序包含两个类型定义。一个是结构 myStruct 的定义，它有一个公共 int 字段 val，另一个是类 MyClass 的定义，它包含一个相同的字段(第 10 章介绍类的成员，如字段，现在只要知道它们的语法是相同的即可)。接着对这两种类型的实例执行相同的操作：

- 声明类型的变量。
- 在这个变量中创建该类型的实例。
- 声明类型的第二个变量。
- 把第一个变量赋给第二个变量。
- 在第一个变量的实例中，给 val 字段赋一个值。
- 在第二个变量的实例中，给 val 字段赋一个值。
- 显示两个变量的 val 字段值。

尽管对两种类型的变量执行了相同的操作，但结果是不同的。在显示 val 字段的值时，两

个 object 类型有相同的值，而结构类型有不同的值。

为什么会这样？

对象是引用类型。在把对象赋给变量时，实际上是把带有一个指针的变量赋给了该指针所指向的对象。在实际代码中，指针是内存中的一个地址。在这种情况下，地址是内存中该对象所在的一个位置。在把第一个对象引用赋给类型为 MyClass 的第二个变量时，实际上是复制了这个地址。这样两个变量就包含同一个对象的指针。

结构是值类型。其变量并不是包含结构的指针，而是包含结构本身。在把第一个结构赋给类型为 myStruct 的第二个变量时，实际上是把第一个结构的所有信息复制到另一个结构中。这个过程与本书前面介绍的简单变量类型如 int 是一样的。最终的结果是两个结构类型变量包含不同的结构。

使用指针的技术隐藏在托管 C#代码中，使代码更简单。使用 C#中的不安全代码可以访问低级操作，例如指针操作，但这是一个比较高级的论题，这里不讨论。

9.8 阴影和深度复制

从一个变量到另一个变量按值复制对象，而不是按引用复制对象(即以与结构相同的方式复制)可能非常复杂。因为一个对象可能包含许多对象的引用，例如字段成员等，这将涉及许多烦人的处理操作。把每个成员从一个对象复制到另一个对象中可能不会成功，因为其中一些成员可能是引用类型。

.NET Framework 考虑了这个问题。按照成员复制简单的对象可以通过派生于 System.Object 的 MemberwiseClone()方法来完成，这是一个受保护的方法，但很容易在对象上定义一个调用该方法的公共方法。这个方法提供的复制功能称为阴影复制，因为它没有考虑引用类型成员。因此，新对象中的引用成员就会指向与源对象中相同成员的对象，在许多情况下这并不理想。如果要创建成员的新实例，此时应复制值，而不复制引用，就需要使用深度复制。

可以执行一个接口 ICloneable，以标准的方式来进行。如果使用这个接口，就必须执行它包含的一个方法 Clone()。这个方法返回一个类型为 System.Object 的值。我们可以采用各种处理方式，执行所选的任何一个方法体得到这个对象。如果愿意，就可以进行深度复制(但执行过程不是必选的，所以可以按照需要执行阴影复制)。

详见第 11 章。

9.9 小结

本章讨论了如何在 C#中定义类和接口，把第 8 章的理论以更具体的方式表达出来。我们论述了基本声明所需要的 C#语法和可以使用的可访问关键字、继承接口和其他类的方式，如何定义抽象和密封类以控制这种继承，以及如何定义构造函数和析构函数。

本章介绍了 System.Object，它是我们所定义的所有类的根基类。这个类提供了几个方法，其中一些是虚拟的，所以可以重写它们的执行代码。这个类还可以把任何对象实例当作这个类

的实例，对任意对象应用多态性。

我们还研究了 VS.NET 为 OOP 开发提供的一些工具，包括 Class View 窗口、Object Browser 窗口，以及给工程添加新类的快速方法。在扩展“多文件”这个概念时，我们还介绍了如何创建装配件，不能执行装配件，但它包含可以在其他工程中使用的类定义。

接着深入探讨了抽象类和接口，理解它们的共同和不同之处，以及使用它们的场合。

最后，讨论了引用类型和值类型，略详细地介绍了结构(对象的值类型)。这引出了阴影复制和深度复制对象的讨论，该主题将在本书的后面再次讨论。

第 10 章将介绍如何定义类成员，例如属性和方法，以便在 C#中利用 OOP 创建真正的应用程序。

9.10 练习

(1) 下面的代码有什么错误？

```
public sealed class MyClass
{
   // class members
}

public class myDerivedClass : MyClass
{
   // class members
}
```

(2) 如何定义不能创建的类？

(3) 为什么不能创建的类仍旧有用？如何利用它们的功能？

(4) 在类库工程 Vehicles 中编写代码，执行本章在“接口和抽象类”一节中讨论的对象系列 Vehicle，其中有 9 个对象和 2 个接口需要执行。

(5) 创建一个控制台应用程序工程 Traffic，它引用 Vehicles.dll(在第 4 题中创建)，其中包括一个函数 AddPassenger()，它接受任何带有 IPassengerCarrier 接口的对象。要证明代码可以运行，使用支持这个接口的每个对象实例调用该函数，在每个对象上调用派生于 System.Object 的 ToString()方法，并把结果写到屏幕上。

第10章　定义类成员

本章继续讨论在C#中如何定义类，主要介绍的是如何定义域、属性和方法等类成员。

首先介绍每种类型需要的代码，以及如何使用VS向导生成这段代码的结构。我们还将论述如何通过编辑成员的属性，来快速修改这些成员。

在介绍完成员定义的基本知识后，就将讨论一些比较高级的成员技术：隐藏基类成员、调用重写的基类成员和嵌套的类型定义。

最后，我们将理论付诸实践，创建一个类库，以便在后面的章节中使用它。

10.1　成员定义

在类定义中，也提供了该类中所有成员的定义，包括域、方法和属性。所有的成员都有自己的访问级别，用下面的关键字之一来定义：

- public——成员可以由任何代码访问。
- private——成员只能由类中的代码访问(如果没有使用任何关键字，就默认使用这个关键字)。
- internal——成员只能由定义它的工程(装配件)内部的代码访问。
- protected——成员只能由类或派生类中的代码访问。

最后两个关键字可以合并使用，所以也有protected internal成员。它们只能由工程(更准确地说，是装配件，详见第21章)中派生类的代码来访问。

域、方法和属性都可以使用关键字static来声明，这表示它们是属于类的静态成员，而不是对象实例的成员，详见第8章。

10.1.1　定义域

域用标准的变量声明格式和前面介绍的修饰符来声明(可以进行初始化)，例如：

```
class MyClass
{
    public int MyInt;
}
```

注意：

.NET Framework中的公共域用PascalCasing来命名，而不是camelCasing。这里使用这种命名方法。这就是上面的域叫作MyInt而不是myInt的原因。这仅是推荐使用的命名模式，但它对于我的意义非常重大。私有域没有推荐的命名模式，它们通常使用camelCasing来命名。

域也可以使用关键字 readonly，表示这个域只能在执行构造函数的过程中赋值，或由初始化赋值语句赋值。例如：

```
class MyClass
{
    public readonly int MyInt = 17;
}
```

如本章的前言所述，域可以使用 static 关键字声明为静态，例如：

```
class MyClass
{
    public static int MyInt;
}
```

静态域可以通过定义它们的类来访问(在上面的范例中，是 MyClass.MyInt)，而不是通过这个类的对象实例来访问。

另外，可以使用关键字 const 来创建一个常量。按照定义，Const 成员也是静态的，所以不需要用 static 修饰(实际上，用 static 修饰会产生一个错误)。

10.1.2 定义方法

方法使用标准函数格式，以及可访问性和可选的 static 修饰符来声明。例如：

```
class MyClass
{
    public string GetString()
    {
        return "Here is a string.";
    }
}
```

注意：

.NET Framework 中的公共方法(如域)用 PascalCasing 命名，而不是 camelCasing。

注意，如果使用了 static 关键字，这个方法就只能通过类来访问，不能通过对象实例来访问。

也可以在方法定义中使用下述关键字：

- virtual——方法可以重写。
- abstract——方法必须重写(只用于抽象类中)。
- override——方法重写了一个基类方法(如果方法被重写，就必须使用该关键字)。
- extern——方法定义放在其他地方。

下面的代码是方法重写的一个范例：

```
public class MyBaseClass
{
```

```
    public virtual void DoSomething()
    {
        // Base implementation
    }
}

public class MyDerivedClass : MyBaseClass
{
    public override void DoSomething()
    {
        // Derived class implementation, overrides base implementation
    }
}
```

如果使用了 override，也可以使用 sealed 指定在派生类中不能对这个方法作进一步的修改，即这个方法不能由派生类重写。例如：

```
public class MyDerivedClass : MyBaseClass
{
    public override sealed void DoSomething()
    {
        // Derived class implementation, overrides base implementation
    }
}
```

使用 extern 可以提供方法在工程外部使用的执行代码。这是一个高级论题，这里不做详细讨论。

10.1.3　定义属性

属性定义的方式与域定义的方式类似，但包含的内容比较多。如前所述，属性涉及的内容比域多，是因为它们在修改状态前还要执行额外的操作。属性拥有两个类似于函数的块，一个块用于获取属性的值，另一个块用于设置属性的值。

这两个块分别用 get 和 set 关键字来定义，可以用于控制对属性的访问级别。可以忽略其中的一个块来创建只读或只写属性(忽略 get 块创建只写属性，忽略 set 块创建只读属性)。当然，这仅适用于外部代码，因为类中的代码可以访问这些块能访问的数据。属性至少要包含一个块，才是有效的(既不能读取也不能修改的属性没有任何用处)。

属性的基本结构包括标准访问修改关键字(public, private 等)、后跟类名、属性名和 get 块(或 set 块，或者 get 块和 set 块，其中包含属性处理代码)，例如：

```
public int MyIntProp
{
    get
    {
        // Property get code
```

```
        }
        set
        {
            // Property set code
        }
    }
```

注意：

.NET Framework 中的公共属性也用 PascalCasing 来命名，而不是 camelCasing，与域和方法一样，这里使用 PascalCasing。

定义代码中的第一行非常类似于定义域的代码。区别是行末没有分号，而是一个包含嵌套 get 和 set 块的代码块。

Get 块必须有一个属性类型的返回值，简单的属性一般与一个私有域相关联，以控制对这个域的访问，此时 get 块可以直接返回该域的值，例如：

```
// Field used by property
private int myInt;

// Property
public int MyIntProp
{
    get
    {
        return myInt;
    }
    set
    {
        // Property set code
    }
}
```

注意类外部的代码不能直接访问这个 myInt 域，因为其访问级别是私有的。必须使用属性来访问该域。

Set 函数以类似的方式把一个值赋给域。这里可以使用关键字 value 引用用户提供的属性值：

```
// Field used by property
private int myInt;

// Property
public int MyIntProp
{
    get
    {
        return myInt;
```

```
        }
        set
        {
            myInt = value;
        }
    }
```

value 等于类型与属性相同的一个值，所以如果域使用相同的类型，就不必担心数据类型转换了。

这个简单的属性只是不能直接访问 myInt 域。在对操作进行更多的控制时，属性的真正作用才能发挥出来。例如，使用下面的代码执行 set 块：

```
        set
        {
            if (value >= 0 && value <= 10)
                myInt = value;
        }
```

如果赋给属性的值在 0 到 10 之间，就修改 myInt。此时，要做一个重要的设计选择：如果使用了无效的值，该怎么办？有 4 种选择：

- 什么也不做 (如上述代码所示)。
- 给域赋默认值。
- 继续执行，就好像没有发生错误一样，但记录下该事件，以备将来分析。
- 抛出一个异常。

一般情况下，最后两个选择比较好，它们取决于如何使用类，以及给类的用户授予多少控制权。抛出异常给用户提供的控制权相当大，可以让他们知道发生了什么情况，并作出合适的响应。对此可以使用标准的 System 异常，例如：

```
        set
        {
            if (value >= 0 && value <= 10)
                myInt = value;
            else
                throw (new ArgumentOutOfRangeException("MyIntProp", value,
                            "MyIntProp must be assigned a value between 0 and 10."));
        }
```

这可以在使用属性的代码中通过 try...catch...finally 逻辑来处理，详见第 7 章。

记录数据，例如记录到文本文件中，对产品代码会比较有效，因为产品代码不应发生错误。他们允许开发人员检查性能，如果需要，还可以调试现有的代码。

属性可以使用 virtual, override 和 abstract 关键字，就像方法一样，但这几个关键字不能全部用于域。

试试看：使用域、方法和属性

(1) 在目录 C:\BegCSharp\Chapter10 下创建一个新控制台应用程序工程 Ch10Ex01。

(2) 使用 VS 快捷方式在文件 MyClass.cs 中添加一个新类 MyClass。

(3) 修改 MyClass.cs 中的代码，如下所示：

```
public class MyClass
{
    public readonly string Name;
    private int intVal;

    public int Val
    {
        get
        {
            return intVal;
        }
        set
        {
            if (value >= 0 && value <= 10)
                intVal = value;
            else
                throw (new ArgumentOutOfRangeException("Val", value,
                    "Val must be assigned a value between 0 and 10."));
        }
    }

    public override string ToString()
    {
        return "Name: " + Name + "\nVal: " + Val;
    }

    private MyClass() : this("Default Name")
    {
    }

    public MyClass(string newName)
    {
        Name = newName;
        intVal = 0;
    }
}
```

(4) 修改 Class1.cs 中的代码，如下所示：

```
static void Main(string[] args)
{
    Console.WriteLine("Creating object myObj...");
```

```
        MyClass myObj = new MyClass("My Object");
        Console.WriteLine("myObj created.");
        for (int i = -1; i <= 0; i++)
        {
            try
            {
                Console.WriteLine("\nAttempting to assign {0} to myObj.Val...",
                                  i);
                myObj.Val = i;
                Console.WriteLine("Value {0} assigned to myObj.Val.", myObj.Val);
            }
            catch (Exception e)
            {
                Console.WriteLine("Exception {0} thrown.", e.GetType().FullName);
                Console.WriteLine("Message:\n\"{0}\"", e.Message);
            }
        }
        Console.WriteLine("\nOutputting myObj.ToString()...");
        Console.WriteLine(myObj.ToString());
        Console.WriteLine("myObj.ToString() Output.");
    }
```

(5) 运行应用程序，其结果如图 10-1 所示。

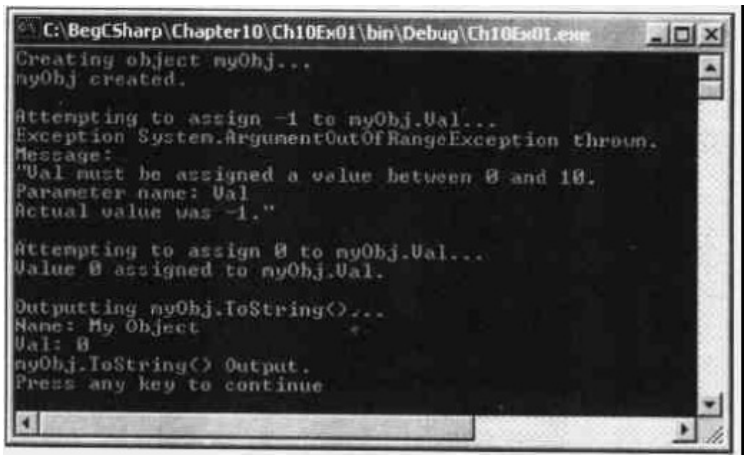


图　10-1

代码的说明

Main()中的代码创建并使用在 MyClass.cs 中定义的 MyClass 类的实例。实例化这个类必须使用非默认的构造函数来进行，因为 MyClass 类的默认构造函数是私有的：

```
private MyClass() : this("Default Name")
{
}
```

注意这里用 this("Default Name")来保证，如果调用了该构造函数，Name 就获取一个值。如果这个类用于派生一个新类，这就是可能的。没有给 Name 域赋值会在后面产生错误的根源。

所使用的非默认构造函数把值赋给 readonly 域 name(只能在域声明或在构造函数中给它赋值)和私有域 intVal。

接着，Main()试着给 myObj(MyClass 的实例)的 Val 属性赋值。for 循环在两次循环中赋值-1 和 0，try...catch 结构用于检查抛出的异常。把-1 赋给属性时，会抛出 System.ArgumentOutOfRangeException 类型的异常，catch 块中的代码会把该异常的信息输出到控制台窗口中。在下一个循环中，值 0 成功地赋给了 Val 属性，通过这个属性再把值赋给私有域 intVal。

最后，使用重写的 ToString()方法输出一个格式化的字符串，表示对象的内容：

```
public override string ToString()
{
    return "Name: " + Name + "\nVal: " + Val;
}
```

这个方法必须使用 override 关键字来声明，因为它重写了基类 System.Object 的虚拟方法 ToString()。此处的代码直接使用属性 Val，而不是私有域 intVal，没有理由不以这种方式使用类中的属性，但这可能会对性能产生比较小的影响(对性能的影响非常小，我们不可能察觉到)。当然，使用属性也可以在属性中进行固有的有效性验证，这对类中的代码也是有益的。

10.2　VS 成员向导

第 9 章介绍了如何使用 VS 中的快捷方式创建一个新类，而且这个类有自己的.cs 文件。而用于修改类的快捷方式(如向导)还有许多，特别是有一些向导可以给类添加属性、方法和域。本节将介绍这些向导。

所有的成员向导都以相同的方式被访问。在 Class View 窗口中，显示了工程中定义的类及其成员。右击一个类，并从 Add 子菜单中选择一个选项，如图 10-2 所示。

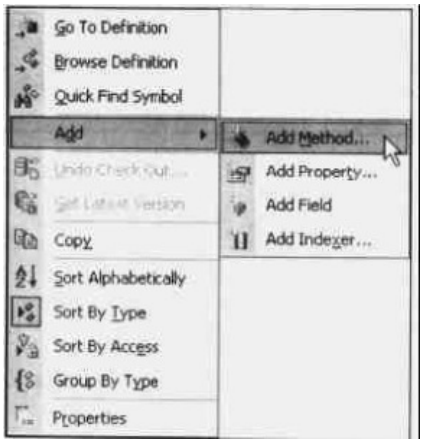


图　10-2

这里有 4 个选项，但这里只讨论其中的 3 个，第 11 章介绍索引符。

10.2.1　添加方法的向导

这个向导允许我们可以给类添加方法。这是通过图 10-3 的对话框来实现的(其中已经填充了几个域)。

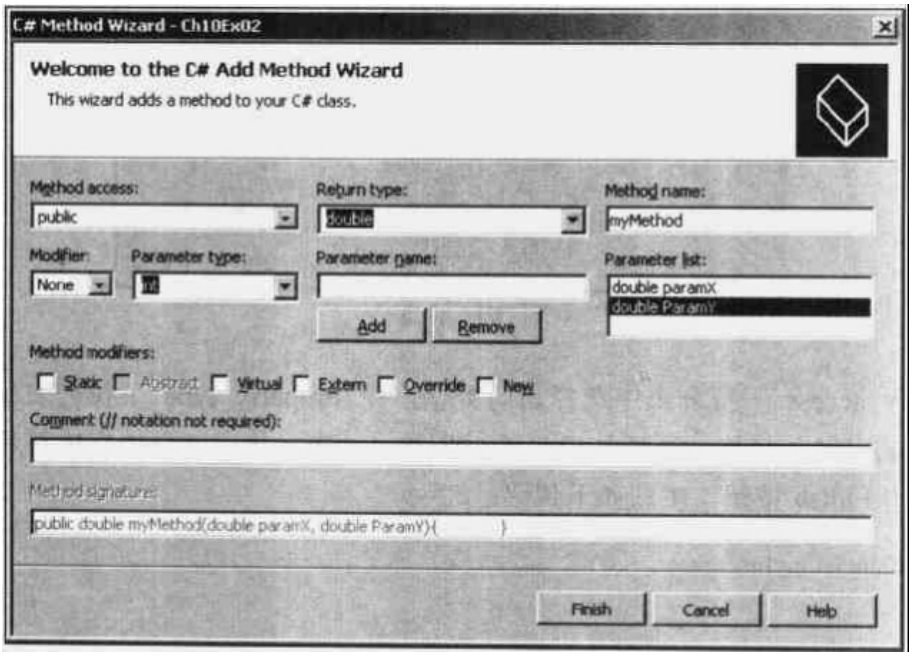


图　10-3

通过这个对话框可以访问所有的方法特性。通过 Method access 下拉列表可以改变访问级别，用 Return type 下拉列表可以改变返回类型(或者在这个域中键入自己的类型)，用 Method name 可以改变名称，用 Method modifiers 可以改变其他修饰符。这些修饰符复选框只能使用有效的组合，只允许用于抽象类中的抽象方法。也可以使用对话框中第二行的 Parameter type 和 Parameter name 域添加参数。Modifier 框可以指定参数的类型：一般(None)、ref 或 out。单击 Add 按钮后，添加的参数就显示在 Parameter list 视图中，还可以使用 Remove 按钮从这个列表中删除参数。

这个对话框还可以用 Comment 域在方法定义的前面添加注释，显示在 Method signature 框中生成的代码预览：

设置好以后(如上所示)，单击 Finish 按钮，把下述代码添加到类中：

```
public double myMethod(double paramX, double ParamY)
{
    return 0;
}
```

显然，这个向导没有提供方法的执行代码，但提供了基本结构，肯定可以减少键入错误！

10.2.2　添加属性的向导

Add Property 向导使用图 10-4 所示的对话框(这里也填充了一些值)。

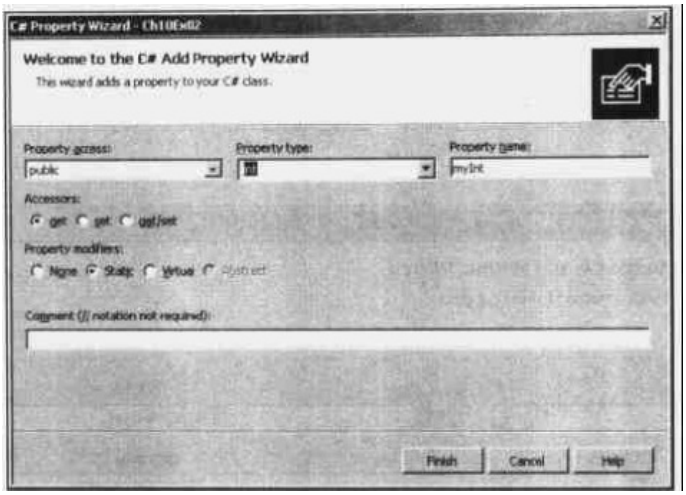


图　10-4

在 Property access 下拉列表中选择访问类型，在 Property type 下拉列表中选择类型，在 Property name 域中键入名称，选择是否要 get 或/和 set 块，并可以添加修饰符和注释。完成上述选择后，单击 Finish 按钮，生成如下代码：

```
public static int myInt
{
    get
    {
        return 0;
    }
}
```

注意该对话框没有提供完整的执行代码，包括为简单的属性匹配一个带域的属性，但提供了基本结构。

10.2.3　添加域的向导

最后，Add Field 向导的对话框比较简单，如图 10-5 所示。

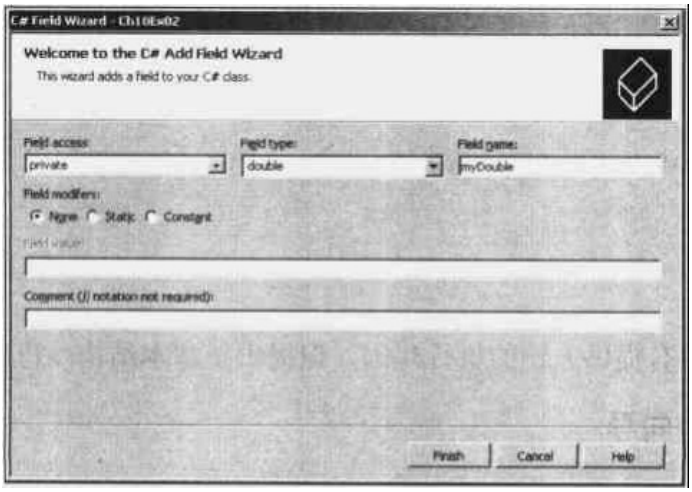


图　10-5

这基本上与添加属性的对话框相同，但不必选择 get 或 set 块，且不能使用 virtual 或 abstract 关键字。但是可以把域定义为常量，此时可以使用 Field value 提供该域的值：

生成的代码比较简单，且是自我描述的：

```
private double myDouble;
```

10.3　成员属性

这里要讨论的最后一个问题是使用 Properties 窗口修改成员属性。如果从 Class View 窗口中选择一个成员，就可以在 Properties 窗口中看到该成员的属性，如图 10-6 所示。



图　10-6

可以直接在这个窗口中修改许多属性，例如通过 Access 属性修改访问级别(图 10-6 中突出显示的属性)。如果这么做，代码也将自动修改，不需要我们进行任何键入操作。

注意 CanOverride 定义了成员是否为 virtual，IsShared 定义了成员是否为 static。

10.4　类成员的其他议题

前面讨论了成员定义的基本知识，下面该讨论一些比较高级的成员议题了。本节主要研究：

- 隐藏基类方法。
- 调用重写或隐藏的基类方法。
- 嵌套的类型定义。

10.4.1　隐藏基类方法

当从基类继承一个(非抽象的)成员时，也就继承了其执行代码。如果继承的成员是虚拟的，就可以用 override 关键字重写这段执行代码。无论继承的成员是否为虚拟，都可以隐藏这些执行代码。这是很有用的，例如，继承的公共成员不像我们希望的那样工作时，就可以隐藏它。

使用下面的代码就可以隐藏：

```
public class MyBaseClass
{
    public void DoSomething()
    {
        // Base implementation
    }
}

public class MyDerivedClass : MyBaseClass
{
    public void DoSomething()
    {
        // Derived class implementation, hides base implementation
    }
}
```

尽管这段代码正常运行，但它会产生一个警告，说明隐藏了一个基类成员。如果是偶然地隐藏了一个需要使用的成员，此时就可以改正错误。如果确实要隐藏该成员，就可以使用 new 关键字显式地说明，这是我们要隐藏的成员：

```
public class MyDerivedClass : MyBaseClass
{
    new public void DoSomething()
    {
        // Derived class implementation, hides base implementation
    }
}
```

其工作方式是完全相同的，但不会显示警告。

此时应注意隐藏基类成员和重写它们的区别。考虑下面的代码：

```
public class MyBaseClass
{
    public virtual void DoSomething()
    {
        Console.WriteLine("Base imp");
    }
}

public class MyDerivedClass : MyBaseClass
{
    public override void DoSomething()
    {
        Console.WriteLine("Derived imp");
    }
}
```

其中重写方法将替换基类中的执行代码，这样下面的代码就将使用新版本，即使这是通过基类进行的，情况也是这样(使用多态性)：

```
MyDerivedClass myObj = new MyDerivedClass();
MyBaseClass myBaseObj;
myBaseObj = myObj;
myBaseObj.DoSomething();
```

结果如下：

```
Derived imp
```

另外，还可以使用下面的代码隐藏基类方法：

```
public class MyBaseClass
{
    public virtual void DoSomething()
    {
        Console.WriteLine("Base imp");
    }
}

public class MyDerivedClass : MyBaseClass
{
    new public void DoSomething()
    {
        Console.WriteLine("Derived imp");
    }
}
```

基类方法不必是虚拟的，但结果是一样的，上面的代码只需要修改一行。对于基类的虚拟方法和非虚拟方法来说，其结果如下：

```
Base imp
```

尽管隐藏了基类的执行代码，但仍可以通过基类访问它。

10.4.2　调用重写或隐藏的基类方法

无论是重写成员还是隐藏成员，都可以在类的内部访问基类成员。这在许多情况下都是很有用的，例如：

- 要对派生类的用户隐藏继承的公共成员，但仍能在类中访问其功能。
- 要给继承的虚拟成员添加执行代码，而不是简单地用新的重写的执行代码替换它。

为此，可以使用 base 关键字，它表示包含在派生类中的基类的执行代码(在控制构造函数中，其用法是类似的，如第 9 章所述)，例如：

```
public class MyBaseClass
```

```
{
    public virtual void DoSomething()
    {
        // Base implementation
    }
}

public class MyDerivedClass : MyBaseClass
{
    public override void DoSomething()
    {
        // Derived class implementation, extends base class implementation
        base.DoSomething();
        // More derived class implementation
    }
}
```

这段代码执行包含在 MyBaseClass 中的 DoSomething()版本，MyBaseClass 是 MyDerived Class 的基类，而 DoSomething()版本包含在 MyDerivedClass 中。

Base 使用对象实例，所以在静态成员中使用它会产生错误。

This 关键字

除了使用第 9 章的 base 关键字外，还可以使用 this 关键字。与 base 一样，this 也可以用在类成员的内部，且该关键字也引用对象实例。由 this 引用的对象实例是当前的对象实例(即不能在静态成员中使用 this 关键字，因为静态成员不是对象实例的一部分)。

This 关键字最常用的功能是把一个当前对象实例的引用传递给一个方法，例如：

```
public void doSomething()
{
    MyTargetClass myObj = new MyTargetClass();
    myObj.DoSomethingWith(this);
}
```

其中，实例化的 MyTargetClass 有一个方法 DoSomethingWith()，该方法带一个参数，其类型与包含上述方法的类兼容。这个参数类型可以是类的类型，而类的类型由这个类继承，是由这个类或 System.Object 执行的一个接口。

10.4.3 嵌套的类型定义

除了在名称空间中定义类之外，还可以在其他类中定义这些类。如果这么做，就可以在定义中使用各种访问修饰符，而不仅仅是 public 和 internal，也可以使用 new 关键字隐藏继承于基类的类型定义。

例如，下面的代码定义了 MyClass，也定义了一个嵌套的类 myNestedClass：

```
public class MyClass
```

```
{
    public class myNestedClass
    {
        public int nestedClassField;
    }
}
```

如果要在 MyClass 的外部实例化 myNestedClass，就必须限定名称，例如：

```
MyClass.myNestedClass myObj = new MyClass.myNestedClass();
```

但是，如果声明嵌套的类为私有或其他与执行该实例化的代码不兼容访问级别，就不能这么做。

存在这个功能的主要原因是定义一个其包含类是私有的类，这样名称空间中的其他代码就不能访问它。

10.5　接口的执行

在继续前，先讨论一下如何定义和执行接口。第 9 章介绍了接口定义的方式与类相似，使用的代码如下：

```
interface IMyInterface
{
    // interface members
}
```

接口成员的定义与类成员的定义相似，但有几个重要的区别：

- 不允许使用访问修饰符 (public, private, protected 或 internal)，所有的接口成员都是公共的。
- 接口成员不能包含代码体。
- 接口不能定义域成员。
- 接口成员不能用关键字 static, virtual, abstract 或 sealed 来定义。
- 类型定义成员是禁止的。

但要隐藏继承了基接口的成员，可以用关键字 new 来定义它们，例如：

```
interface IMyBaseInterface
{
    void DoSomething();
}

interface IMyDerivedInterface : IMyBaseInterface
{
    new void DoSomething();
}
```

其执行方式与隐藏继承的类成员一样。

在接口中定义的属性可以确定访问块 get 和/或 set 中的哪一个能用于该属性，例如：

```
interface IMyInterface
{
    int MyInt
    {
        get;
        set;
    }
}
```

其中 int 属性 MyInt 有 get 和 set 访问程序块。对于访问级别有更严限制的属性来说，可以省略它们中的任一个。

但要注意，接口没有指定属性应如何存储。接口不能指定域，例如用于存储属性数据的域。

最后，接口与类一样，可以定义为类的成员(但与接口的其他成员不同，因为接口不能包含类型定义)。

在类中执行接口

执行接口的类必须包含该接口所有成员的执行代码，且必须匹配指定的签名(包括匹配指定的 get 和 set 块)，并且必须是公共的。可以使用关键字 virtual 或 abstract 来执行接口成员，但不能使用 static 或 const，例如：

```
public interface IMyInterface
{
    void DoSomething();
    void DoSomethingElse();
}

public class MyClass : IMyInterface
{
    public void IMyInterface.DoSomething()
    {
    }

    public void DoSomethingElse()
    {
    }
}
```

接口成员还可以在基类上执行，例如：

```
public interface IMyInterface
{
    void DoSomething();
```

```
    void DoSomethingElse();
}

public class MyBaseClass
{
    public void DoSomething()
    {
    }
}

public class MyDerivedClass : MyBaseClass, IMyInterface
{
    public void DoSomethingElse()
    {
    }
}
```

继承一个执行给定接口的基类，就意味着派生类隐式地支持这个接口，例如：

```
public interface IMyInterface
{
    void DoSomething();
    void DoSomethingElse();
}

public class MyBaseClass : IMyInterface
{
    public virtual void DoSomething()
    {
    }

    public virtual void DoSomethingElse()
    {
    }
}

public class MyDerivedClass : MyBaseClass
{
    public override void DoSomething()
    {
    }
}
```

如上所述，在基类中把执行代码定义为虚拟，派生类就可以替换该执行代码，而不是隐藏它们。如果要使用 new 关键字隐藏一个基类成员，而不是重写它，则方法 IMyInterface.DoSomething()就总是引用基类版本，即使派生类通过这个接口来访问，也是这样。

显式执行接口成员

接口成员也可以由类显式的执行。如果这么做，该成员就只能通过接口来访问，不能通过类来访问。上一节的代码中使用的隐式成员可以通过类和接口来访问。

例如，如果类 MyClass 隐式地执行接口 IMyInterface 的方法 DoSomething()，如上所述，则下面的代码就是有效的：

```
MyClass myObj = new MyClass();
myObj.DoSomething();
```

即：

```
MyClass myObj = new MyClass();
IMyInterface myInt = myObj;
myInt.DoSomething();
```

另外，如果 MyDerivedClass 显式执行 DoSomething()，就只能使用后一种技术。其代码如下：

```
public class MyClass : IMyInterface
{
    void IMyInterface.DoSomething()
    {
    }

    public void DoSomethingElse()
    {
    }
}
```

其中 DoSomething()是显式地执行的，而 DoSomethingElse()是隐式地执行的。只有后者可以直接通过 MyClass 的对象实例来访问。

10.6 范例应用程序

为了解释前面使用的一些技术，下面开发一个类模块，以便在后面的章节中使用。这个类模块包含两个类：

- Card—— 表示一个标准的扑克牌，包含梅花、方块、红心和黑桃，其顺序是从 A 到 K。
- Deck—— 表示一副完整的 52 张扑克牌，在扑克牌中可以按照位置访问各张牌，并可以洗牌。

再开发一个简单的客户程序，确保程序正常工作，但在整个扑克牌应用程序中不使用扑克牌。

10.6.1 规划应用程序

这个应用程序的类库 Ch10CardLib 包含类。在开始编写代码前，应规划一下需要的结构和类的功能。

1. Card 类

Card 类基本上是两个只读域 suit 和 rank 的容器。把域指定为只读的原因是“空白”的牌是没有意义的，牌在创建好后也不能修改。为此，要把默认的构造函数指定为私有，并提供另一个构造函数，从给定的 suit 和 rank 中建立一副扑克牌。

此外，Card 类要重写 System.Object 的 ToString()方法，这样才能获得人们可以理解的字符串，以表示扑克牌。为使编码简单一些，为两个域 suit 和 rank 提供枚举。

Card 类如图 10-7 所示。

2. Deck 类

Deck 类包含 52 个 Card 对象。我们为这些对象使用一个简单的数组类型。这个数组不能直接访问，因为对 Card 对象的访问要通过 GetCard()方法来实现，该方法返回指定下标的 Card 对象。

这个类也有一个 Shuffle()方法，重新安排数组中的牌，所以它应如图 10-8 所示。

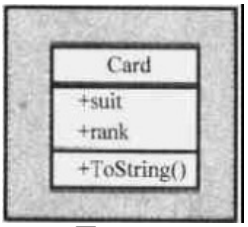


图　10-7

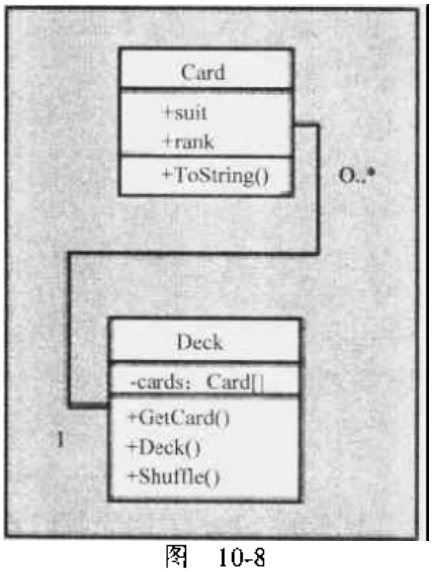


图　10-8

10.6.2　编写类库

对于本例，假定读者对 VS 比较熟悉，所以不再使用标准的“试试看”方式，不显式地列出步骤。

这两个类和枚举都包含在一个类库工程 Ch10CardLib 中。这个工程将包含两个.cs 文件，Card.cs 包含 Card 类的定义以及 Suit 和 Rank 枚举，而 Deck.cs 包含 Deck 类的定义。

1. Card.cs

本节将分析 Card.cs 的代码。首先是通常的 using 语句和名称空间声明：

```
using System;

namespace Ch10CardLib
{
```

接着是 Suit 枚举定义：

```
public enum Suit
{
    Club,
    Diamond,
    Heart,
    Spade
}
```

然后是 Rank 枚举定义，它用 1 来表示 A：

```
public enum Rank
{
    Ace = 1,
    Deuce,
    Three,
    Four,
    Five,
    Six,
    Seven,
    Eight,
    Nine,
    Ten,
    Jack,
    Queen,
    King
}
```

这两个枚举类型都是公共的，因为我们要在这个类库外部使用它们。
下面分析这个文件的主要部分，即 Card 类的定义：

```
public class Card
{
```

Card 定义中的第一部分代码定义了两个只读域，它们使用文件开始处定义的枚举类型：

```
    public readonly Suit suit;

    public readonly Rank rank;
```

重写的 ToString()方法非常简单，因为它们使用的枚举类型非常简单：

```
    public override string ToString()
    {
        return "The " + rank + " of " + suit + "s";
    }
```

以这种方式把域名放在字符串中，可以把所存储的枚举值的字符串表示写到要求的字符串中。接着是构造函数。首先是私有的默认构造函数：

```
        private Card()
        {
        }
```

接着，构造函数必须用于创建 Card 实例，它带有 Suit 和 Rank 参数，初始化只读域：

```
        public Card(Suit newSuit, Rank newRank)
        {
            suit = newSuit;
            rank = newRank;
        }
    }
}
```

这就分析完这个文件的代码。

2. Deck.cs

这个文件的开头与 Card.cs 的开头相同，是标准的代码：

```
using System;

namespace Ch10CardLib
{
```

这里除了要定义的 Deck 类外，没有其他类，所以我们直接讨论类的定义：

```
    public class Deck
    {
```

定义的第一个成员是 Card 对象的私有数组 cards。

```
        private Card[] cards;
```

接着是构造函数，它在 cards 域中创建 52 张牌，并给它们赋值。对两个枚举的所有组合进行迭代，每次迭代都创建一张牌。这将使 cards 包含一个有序的扑克牌列表：

```
        public Deck()
        {
            cards = new Card[52];
            for (int suitVal = 0; suitVal < 4; suitVal++)
            {
                for (int rankVal = 1; rankVal < 14; rankVal++)
                {
                    cards[suitVal * 13 + rankVal - 1] = new Card((Suit)suitVal,
                                                                 (Rank)rankVal);
```

```
        }
    }
}
```

然后，执行 GetCard()方法，为指定的下标返回 Card 对象，或者以与前面相同的方式抛出一个异常：

```
public Card GetCard(int cardNum)
{
    if (cardNum >= 0 && cardNum <= 51)
        return cards[cardNum];
    else
        throw (new System.ArgumentOutOfRangeException("cardNum", cardNum,
                "Value must be between 0 and 51."));
}
```

最后，执行 Shuffle()方法。这个方法创建一个临时的扑克牌数组，并把扑克牌从现有的 cards 数组随机复制到这个数组中。这个函数的主体是一个从 0 到 51 的循环，在每次循环时，都会使用.NET Framework 中 System.Random 类的实例生成一个 0 到 51 之间的随机数。进行了实例化后，这个类的对象使用方法 Next(X)生成 0 到 X 之间的一个随机数。有了一个随机数后，就可以使用它作为临时数组中 Card 对象的下标，以便复制 cards 数组中的 Card 对象。

为了记录已赋值的扑克牌，我们还有一个 bool 变量的数组，在复制每张牌时，把该数组中的值指定为 true。在生成随机数时，检查这个数组，看看是否已经把一张牌复制到临时数组中由随机数指定的位置上了，如果已经复制好了，就将生成另一个随机数。

这不是完成该任务的最高效的方式，因为许多随机数都可能在找到一个空位置以复制扑克牌之前生成。但是，它仍能完成任务，因为 C#代码执行得很快，我们几乎觉察不到延迟。

代码如下：

```
public void Shuffle()
{
    Card[] newDeck = new Card[52];
    bool[] assigned = new bool[52];
    for (int i = 0; i < 52; i++)
    {
        int destCard = 0;
        bool foundCard = false;
        Random sourceGen = new Random();
        while (foundCard == false)
        {
            destCard = sourceGen.Next(52);
            if (assigned[destCard] == false)
                foundCard = true;
        }
        assigned[destCard] = true;
```

```
                newDeck[destCard] = cards[i];
            }
            cards = newDeck;
        }
    }
}
```

这就是类库代码。

10.6.3　类库的客户应用程序

为了简单起见，可以在包含类库的解决方案中添加一个客户控制台应用程序。为此，只需确保在创建工程 Ch10CardClient 时选择 Add to Solution 选项，如图 10-9 所示。

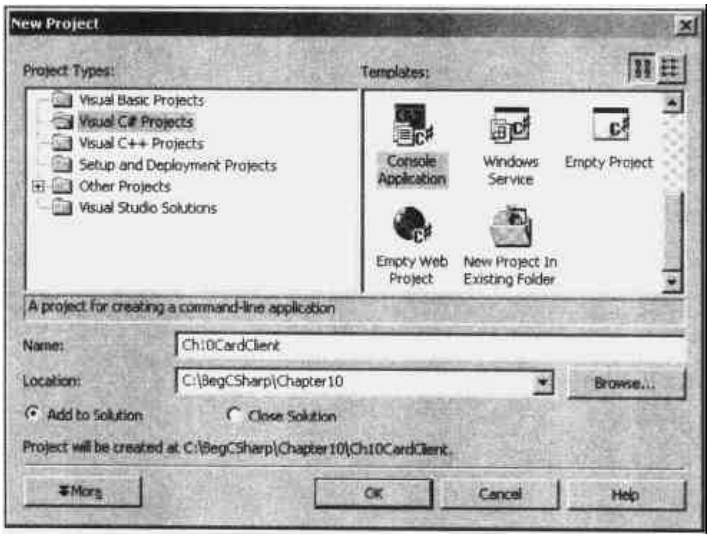


图　10-9

为了使用在这个新的控制台应用程序工程中创建的类库，只需添加一个对 Ch10CardLib 类库工程的引用。一旦创建了工程，就可以通过 Add Reference 对话框的 Projects 选项卡来添加引用，如图 10-10 所示。

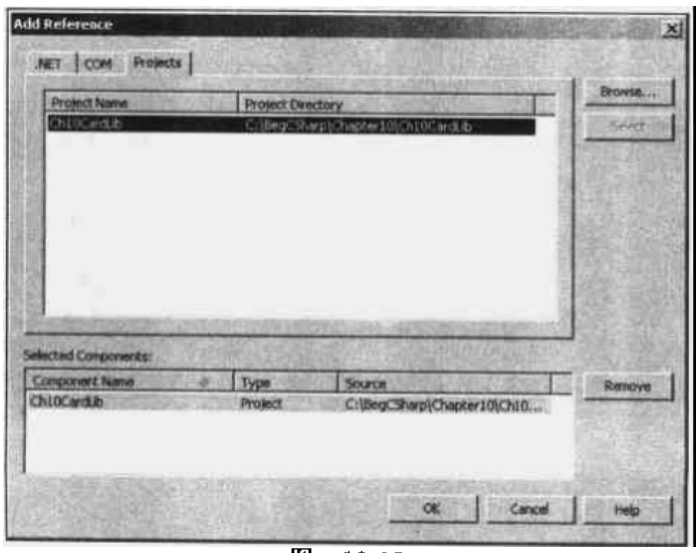


图　10-10

选择工程，单击 Select，再单击 OK 按钮，这样就添加了引用。

因为这个新工程是第二个创建的，所以还需要指定该工程为解决方案的启动工程，即在单击运行按钮后，将执行这个工程。为此，在 Solution Explorer 窗口中右击该工程名，选择 Set as StartUp Project 菜单项。

然后需要添加使用新类的代码，这些代码不需要做什么特别的任务，所以添加下面的代码就可以：

```
using System;
using Ch10CardLib;

namespace Ch10CardClient
{
    class Class1
    {
        static void Main(string[] args)
        {
            Deck myDeck = new Deck();
            myDeck.Shuffle();
            for (int i = 0; i < 52; i++)
            {
                Card tempCard = myDeck.GetCard(i);
                Console.Write(tempCard.ToString());
                if (i != 51)
                    Console.Write(", ");
                else
                    Console.WriteLine();
            }
        }
    }
}
```

其结果如图 10-11 所示。



图　10-11

52 张扑克牌是随机放置的。

后面的章节将继续开发和使用这个类库。

10.7　小结

本章完成了定义基类的讨论。仍有许多内容没有包含进来，但前面涉及到的技术已经足够我们创建相当复杂的应用程序了。

本章介绍了如何定义域、方法和属性，接着讨论了各种访问级别和修饰关键字。为了进行改进，我们介绍了 VS 向导和工具，以便用快捷的方式来创建一个新类。

介绍过这些基本主题后，我们详细讨论了继承行为，主要内容是如何用 new 关键字隐藏不想要的继承成员，扩展基类成员，而不是用 base 关键字替代它们的执行代码。我们还论述了嵌套的类定义。

之后，详细研究了接口的定义和执行，包括显式和隐式执行的概念。

最后，我们开发和使用了一个表示扑克牌的简单类库，后面的章节还将进一步使用这个库。

第 11 章介绍一些生成类的较高级方式，例如集合、运算符重载和深度复制等。

10.8　练习

(1) 编写代码，用虚拟方法 GetString()定义一个基类 MyClass。这个方法应返回存储在受保护的域 myString 中的字符串，该域可以通过只写公共属性 ContainedString 来访问。

(2) 从类 MyClass 派生一个类 MyDerivedClass。重写 GetString()方法，使用该方法的基类执行代码从基类中返回一个字符串，但在返回的字符串中添加文本“(output from derived class)”。

(3) 编写一个类 MyCopyableClass，该类可以使用方法 GetCopy()返回它本身的一个副本。这个方法应使用派生于 System.Object 的 MemberwiseClone()方法。给该类添加一个简单的属性，并且编写使用该类检查任务是否成功执行的客户代码。

(4) 为 Ch10CardLib 库编写一个控制台客户程序，从搅乱的 Deck 对象中一次取出 5 张牌。如果这 5 张牌都是相同的花色，客户程序就应在屏幕上显示这 5 张牌，以及文本“Flush!”，否则就显示 50 张牌以及文本“No flush”，并退出。

第11章　类的更多内容

前面讨论了 C#中所有的基本 OOP 技术，读者还应熟悉一些比较高级的技术。本章的主要内容如下：

- 集合—— 一组对象，它们包含其他对象数组，还包含控制访问这些对象的功能。
- 运算符重载—— 对类的实例使用运算符(如+)，以配置该类。
- 高级转换—— 一些高级的 C#类型转换功能。
- 深度复制—— 确保复制对象不包含对存储在其源对象中的数据的引用。
- 定制异常—— 创建自己的异常，为捕获该异常的代码提供其他信息。

11.1　集合

第 5 章介绍了如何使用数组创建包含许多对象或值的变量类型。但数组有一定的限制。最大的限制是一旦创建好数组，它们的大小就是固定的，不能在现有数组的末尾添加新项目，除非创建一个新的数组。这常常意味着用于处理数组的语法比较复杂。OOP 技术可以创建在内部进行这些处理的类，因此简化了使用项目列表或数组的代码。

C#中的数组是作为 System.Array 类的实例来执行的，它们只是集合类中的一种。集合类一般用于处理对象列表，其功能比简单数组要多，这些功能是通过执行 System.Collections 名称空间中的接口而实现的，因此集合的语法已经标准化了。这个名称空间还包含其他一些有趣的东西，例如以与 System.Array 不同的方式执行这些接口的类。

集合的功能(包括基本函数，例如用[index]语法访问集合中的项目)可以通过接口来实现，该接口不仅没有限制我们使用基本集合类，例如 System.Array。相反我们还可以创建自己的定制集合类。这些集合可以专用于要枚举的对象(即要从中建立集合的对象)。这么做的一个优点是定制的集合类可以是强类型化的。也就是说，在从集合中提取项目时，不需要把它们转换为正确的类型。

在 System.Collections 名称空间中有许多接口都提供了基本的集合功能：

- IEnumerable 提供了循环集合中项目的功能。
- ICollection (继承于 IEnumerable)可以获取集合中项目的个数，并能把项目复制到一个简单的数组类型中。
- IList (继承于 IEnumerable 和 ICollection)提供了集合的项目列表，并可以访问这些项目，以及其他一些与项目列表相关的功能。
- IDictionary (继承于 IEnumerable 和 ICollection)类似于 IList，但提供了可通过键码值而不是索引访问的项目列表。

System.Array 类继承了 IList, ICollection 和 IEnumerable，但不支持 IList 的一些更高级的功

能，表示大小固定的一个项目列表。

11.1.1　使用集合

Systems.Collections 名称空间中的一个类 System.Collections.ArrayList 也执行 IList, ICollection 和 IEnumerable 接口，但执行的方式比 System.Array 更复杂。数组的大小是固定的(不能增加或删除元素)，而这个类可以用于表示大小可变的项目列表。为了更准确地理解这样一个高级集合的功能，下面介绍一个使用这个类和一个简单数组的范例。

试试看：数组和高级集合

(1) 在目录 C:\BegCSharp\Chapter11\下创建一个新的控制台应用程序 Ch11Ex01。

(2) 用 Add Class 向导在文件 Animal.cs 中给工程添加一个新类 Animal。

(3) 修改 Animal.cs 中的代码，如下所示：

```
namespace Ch11Ex01
{
    public abstract class Animal
    {
        protected string name;

        public string Name
        {
        get
            {
                return name;
            }
            set
            {
                name = value;
            }
        }

        public Animal()
        {
            name = "The animal with no name";
        }

        public Animal(string newName)
        {
            name = newName;
        }

        public void Feed()
        {
```

```
            Console.WriteLine("{0} has been fed.", name);
        }
    }
    public class Cow : Animal
    {
        public void Milk()
        {
            Console.WriteLine("{0} has been milked.", name);
        }

        public Cow(string newName) : base(newName)
        {
        }
    }

    public class Chicken : Animal
    {
        public void LayEgg()
        {
            Console.WriteLine("{0} has laid an egg.", name);
        }

        public Chicken(string newName) : base(newName)
        {
        }
    }
}
```

(4) 修改 Class1.cs 中的代码，如下所示：

```
using System;
using System.Collections;

namespace Ch11Ex01
{
    /// <summary>
    /// Summary description for Class1.
    /// </summary>
    class Class1
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Create an Array type collection of Animal " +
                "objects and use it:");

            Animal[] animalArray = new Animal[2];
```

```
        Cow myCow1 = new Cow("Deirdre");
        animalArray[0] = myCow1;
        animalArray[1] = new Chicken("Ken");

        foreach (Animal myAnimal in animalArray)
        {
            Console.WriteLine("New {0} object added to Array collection, " +
                "Name = {1}", myAnimal.ToString(), myAnimal.Name);
        }

        Console.WriteLine("Array collection contains {0} objects.",
            animalArray.Length);
        animalArray[0].Feed();
        ((Chicken)animalArray[1]).LayEgg();
        Console.WriteLine();

        Console.WriteLine("Create an ArrayList type collection of Animal " +
            "objects and use it:");
        ArrayList animalArrayList = new ArrayList();
        Cow myCow2 = new Cow("Hayley");
        animalArrayList.Add(myCow2);
        animalArrayList.Add(new Chicken("Roy"));

        foreach (Animal myAnimal in animalArrayList)
        {
            Console.WriteLine("New {0} object added to ArrayList collection," +
                " Name = {1}", myAnimal.ToString(), myAnimal.Name);
        }
        Console.WriteLine("ArrayList collection contains {0} objects.",
            animalArrayList.Count);
        ((Animal)animalArrayList[0]).Feed();
        ((Chicken)animalArrayList[1]).LayEgg();
        Console.WriteLine();

        Console.WriteLine("Additional manipulation of ArrayList:");
        animalArrayList.RemoveAt(0);
        ((Animal)animalArrayList[0]).Feed();
        animalArrayList.AddRange(animalArray);
        ((Chicken)animalArrayList[2]).LayEgg();
        Console.WriteLine("The animal called {0} is at index {1}.",
            myCow1.Name, animalArrayList.IndexOf(myCow1));
        myCow1.Name = "Janice";
        Console.WriteLine("The animal is now called {0}.",
            ((Animal)animalArrayList[1]).Name);
    }
```

```
        }
    }
```

(5) 运行该应用程序，其结果如图 11-1 所示。



图　11-1

代码的说明

这个范例创建了两个对象集合，第一个集合使用 System.Array 类(这是一个简单的数组)，第二个集合使用 System.ArrayList 类。这两个集合都是 Animal 对象，在 Animal.cs 中被定义。Animal 类是抽象类，所以不能进行实例化。但通过多态性，可以使集合中的项目成为派生于 Animal 类的 Cow 和 Chicken 类的实例。

这些数组在 Class1.cs 的 Main()方法中创建好后，就可以显示其特性和功能。有几个处理操作可以应用到 Array 和 ArrayList 集合上，但它们的语法略有区别。但也有一些操作使用更高级的 ArrayList 类型。

下面首先通过比较这两种集合的代码和结果，讨论一下这两种集合类似的操作。

首先集合的创建——对于简单的数组来说，必须用固定的大小来初始化数组，这样才能使用它。下面使用第 5 章介绍的标准语法创建数组 animalArray：

```
Animal[] animalArray = new Animal[2];
```

另一方面，ArrayList 集合不需要初始化其大小，所以可以使用下面的代码创建列表 animalArrayList：

```
ArrayList animalArrayList = new ArrayList();
```

这个类还有另外两个构造函数可以使用。第一个构造函数把现有的集合作为一个参数，把现有集合的内容复制到新实例中；而另一个构造函数也通过一个参数设置集合的容量(capacity)。这个容量用一个 int 值指定，设置集合中可以包含的项目总数。但是这并不是真实的容量，因为如果集合中的项目个数超过了这个值，容量就会自动增大一倍。

因为数组是引用类型的数组(例如 Animal 和 Animal 派生的对象)，所以用一个长度初始化数组并没有初始化它所包含的项目。要使用一个指定的项目，该项目还需要初始化，即需要给这个项目赋予初始化了的对象：

```
Cow myCow1 = new Cow("Deirdre");
```

```
animalArray[0] = myCow1;
animalArray[1] = new Chicken("Ken");
```

这段代码以两种方式完成该初始化任务：用现有的 Cow 对象来赋值，或者通过创建一个新的 Chicken 对象来赋值。主要的区别是前者引用了数组中的对象——我们在代码的后面就使用了这种方式。

对于 ArrayList 集合，它没有现成的项目，也没有 null 引用的项目。这样就不能以相同的方式给索引赋一些新实例。我们使用 ArrayList 对象的 Add()方法添加新项目：

```
Cow myCow2 = new Cow("Hayley");
animalArrayList.Add(myCow2);
animalArrayList.Add(new Chicken("Roy"));
```

除了语法略有不同外，还可以用相同的方式把新对象或现有的对象添加到集合中。

以这种方式添加完项目后，就可以使用与数组相同的语法来重写它们，例如：

```
animalArrayList[0] = new Cow("Alma");
```

但不能在这个范例中这么做。

第 5 章介绍了如何使用 foreach 结构迭代一个数组。这是可以的，因为 System.Array 类执行 IEnumerable 接口，这个接口的惟一方法 GetEnumerator()可以迭代集合中的项目。后面将详细讨论这一点。在代码中，我们写出了数组中每个 Animal 对象的信息：

```
foreach (Animal myAnimal in animalArray)
{
    Console.WriteLine("New {0} object added to Array collection, " +
        "Name = {1}", myAnimal.ToString(), myAnimal.Name);
}
```

这里使用的 ArrayList 对象也支持 IEnumerable 接口，并可以与 foreach 一起使用，此时语法是相同的：

```
foreach (Animal myAnimal in animalArrayList)
{
    Console.WriteLine("New {0} object added to ArrayList collection, " -
        "Name = {1}", myAnimal.ToString(), myAnimal.Name);
}
```

接着，使用数组的 Length 属性，在屏幕上输出数组中项目的个数：

```
Console.WriteLine("Array collection contains {0} objects.",
    animalArray.Length);
```

也可以使用 ArrayList 集合得到相同的结果，但要使用 Count 属性，该属性是 ICollection 接口的一部分：

```
Console.WriteLine("ArrayList collection contains {0} objects.",
```

```
animalArrayList.Count);
```

集合——无论是简单的数组，还是比较复杂的集合——都用得不多，除非它们可以访问属于它们的项目。简单数组是强类型化的，可以直接访问它们所包含的项目类型。所以可以直接调用项目的方法：

```
animalArray[0].Feed();
```

数组的类型是抽象类型 Animal，因此不能直接调用由派生类所提供的方法。而必须使用数据类型转换：

```
((Chicken)animalArray[1]).LayEgg();
```

ArrayList 集合是 System.Object 对象的集合(通过多态性赋给 Animal 对象)，所以必须对所有的项目进行数据类型转换：

```
((Animal)animalArrayList[0]).Feed();
((Chicken)animalArrayList[1]).LayEgg();
```

代码的剩余部分利用的一些 ArrayList 集合功能超出了 Array 集合的功能范围。

首先，可以使用 Remove() 和 RemoveAt()方法删除项目，这两个方法是在 ArrayList 类中执行的 IList 接口的一部分。它们分别根据项目引用或索引把项目从数组中删除。在本例中，我们使用后一个方法删除添加到列表中的第一个项目，即 Name 属性为“Hayley”的 Cow 对象：

```
animalArrayList.RemoveAt(0);
```

另外，还可以使用：

```
animalArrayList.Remove(myCow2);
```

因为这个对象已经有一个本地引用了，所以可以在数组中添加一个现有的引用，而不是创建一个新对象。

无论采用哪种方式，集合中惟一剩下的项目是 Chicken 对象，可以用下面的方式访问它：

```
((Animal)animalArrayList[0]).Feed();
```

对 ArrayList 对象中的项目进行的任何修改都会使数组中剩下 N 个项目，其实现方式与保留从 0 到 N-1 的索引相同。例如，删除索引为 0 的项目，会使其他项目在数组中移动一个位置，所以应使用索引 0 来访问 Chicken 对象，而不是 1。不再有索引为 1 的项目了(因为集合中最初只有两个项目)，所以如果试图执行下面的代码，就会抛出异常：

```
((Animal)animalArrayList[1]).Feed();
```

ArrayList 集合可以用 AddRange()方法一次添加好几个项目。这个方法接受带有 ICollection 接口的任何对象，该接口包含前面的代码所创建的 animalArray 数组：

```
animalArrayList.AddRange(animalArray);
```

为了确定这是否有效，可以试着访问集合中的第三个项目，它将是 animalArray 中的第二个项目：

```
((Chicken)animalArrayList[2]).LayEgg();
```

AddRange()方法不是 ArrayList 提供的任何接口的一部分。这个方法专用于 ArrayList 类，论证了可以在集合类中执行定制操作，而不仅仅是前面介绍的接口要求的操作。这个类还提供了其他有趣的方法，例如 InsertRange()，它可以把数组对象插入到列表中的任何位置，还有用于给数组排序和重新排序的方法。

最后，再回过头来看看对同一个对象进行多个引用。使用 IList 接口中的 IndexOf()方法，不仅可以看出，myCow1(最初添加到 animalArray 中的一个对象)现在是 animalArrayList 集合的一部分，而且还可以看到它的索引如下：

```
Console.WriteLine("The animal called {0} is at index {1}.",
    myCow1.Name, animalArrayList.IndexOf(myCow1));
```

例如，接下来的两行代码通过对象引用重新命名了对象，并通过集合引用显示了新名称：

```
myCow1.Name = "Janice";
Console.WriteLine("The animal is now called {0}.",
    ((Animal)animalArrayList[1]).Name);
```

11.1.2　定义集合

前面介绍了使用高级集合类能完成什么任务，下面讨论如何创建自己的、强类型化的集合。一种方式是手动执行需要的方法，但这比较花时间，在某些情况下也非常复杂。我们还可以从一个类派生自己的集合，例如 System.Collections.CollectionBase 类，这个抽象类提供了集合类的许多执行方式。这是推荐使用的方式。

CollectionBase 类有接口 IEnumerable, ICollection 和 IList，但只提供了一些要求的执行代码，特别是 IList 的 Clear()和 RemoveAt()方法，以及 ICollection 的 Count 属性。如果要使用提供的功能，就需要自己执行其他代码。

为了方便地完成任务，CollectionBase 提供了两个受保护的属性，它们可以访问存储的对象本身。我们可以使用 List 和 InnerList，List 可以通过 IList 接口访问项目，InnerList 则是用于存储项目的 ArrayList 对象。

例如，存储 Animal 对象的集合类可以定义如下(稍后介绍一个比较完整的执行代码)：

```
public class Animals : CollectionBase
{
    public void Add(Animal newAnimal)
    {
        List.Add(newAnimal);
    }

    public void Remove(Animal oldAnimal)
```

```
    {
        List.Remove(oldAnimal);
    }

    public Animals()
    {
    }
}
```

其中，Add()和 Remove()方法被执行为强类型化的方法，使用 IList 接口中用于访问项目的标准 Add()方法。该方法现在只用于处理 Animal 类或派生于 Animal 的类，而前面介绍的 ArrayList 执行代码可处理任何对象。

CollectionBase 类可以对派生的集合使用 foreach 语法。例如，可以使用下面的代码：

```
Console.WriteLine("Using custom collection class Animals:");
Animals animalCollection = new Animals();
animalCollection.Add(new Cow("Sarah"));
foreach (Animal myAnimal in animalCollection)
{
    Console.WriteLine("New {0} object added to custom collection, " +
        "Name = {1}", myAnimal.ToString(), myAnimal.Name);
}
```

但不能使用下面的代码：

```
animalCollection[0].Feed();
```

要以这种方式通过索引访问项目，就需要使用索引符。

1. 索引符

索引符是一种特殊类型的属性，可以把它添加到一个类中，以提供类似于数组的访问。实际上，可以通过一个索引符提供更复杂的访问，因为我们可以定义和使用复杂的参数类型和方括号语法。它最常见的一个用法是对项目执行一个简单的数字索引。

在 Animal 对象的 Animals 集合中添加一个索引符，如下所示：

```
public class Animals : CollectionBase
{
    ...
    public Animal this[int animalIndex]
    {
        get
        {
            return (Animal)List[animalIndex];
        }
        set
        {
```

```
            List[animalIndex] = value;
        }
    }
}
```

this 关键字与方括号中的参数一起使用，但这看起来类似于其他属性。这个语法是富有逻辑的，因为在访问索引符时，将使用对象名，后跟放在方括号中的索引参数(例如 MyAnimals[0])。

这段代码对 List 属性使用一个索引符(即可以访问 CollectionBase 中的 ArrayList 的 IList 接口，ArrayList 存储了项目)：

```
return (Animal)List[animalIndex];
```

这里需要进行显式数据类型转换，因为 IList.List 属性返回一个 System.Object 对象。

注意，我们为这个索引符定义了一个类型。使用该索引符访问一个项目时，就可以得到这个类型，即可以编写下述代码：

```
animalCollection[0].Feed();
```

而不是：

```
((Animal)animalCollection[0]).Feed();
```

这是强类型化的定制集合的另一个方便特性。下面讨论最后一个范例，实践一下该特性。

试试看：执行 Animals 集合

(1) 在目录 C:\BegCSharp\Chapter11\下创建一个新控制台应用程序 Ch11Ex02。

(2) 在 Solution Explorer 窗口中右击工程名，选择 Add | Add Existing Item...选项，如图 11-2 所示。



图 11-2

(3) 从目录 C:\BegCSharp\Chapter11\Ch11Ex01 下选择 Animal.cs 文件，单击 Open 按钮，如图 11-3 所示。

(4) 修改 Animal.cs 中的名称空间声明，如下所示：

```
namespace Ch11Ex02
```



图 11-3

(5) 使用 Add Class 向导添加一个新类 Animals，并把它存储在 Animals.cs 中。

(6) 修改 Animal.cs 中的代码，如下所示：

```
using System;
using System.Collections;

namespace Ch11Ex02
{
    public class Animals : CollectionBase
    {
        public void Add(Animal newAnimal)
        {
            List.Add(newAnimal);
        }

        public void Remove(Animal newAnimal)
        {
            List.Remove(newAnimal);
        }

        public Animals()
        {
        }

        public Animal this[int animalIndex]
        {
            get
            {
```

```
            return (Animal)List[animalIndex];
        }
        set
        {
            List[animalIndex] = value;
        }
    }
}
```

(7) 修改 Class1.cs，如下所示：

```
static void Main(string[] args)
{
    Animals animalCollection = new Animals();
    animalCollection.Add(new Cow("Jack"));
    animalCollection.Add(new Chicken("Vera"));
    foreach (Animal myAnimal in animalCollection)
    {
        myAnimal.Feed();
    }
}
```

(8) 执行应用程序，其结果如图 11-4 所示。

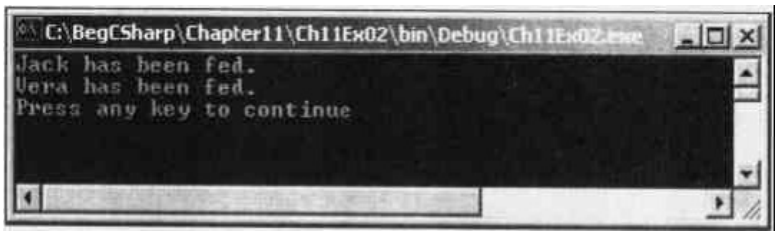


图　11-4

代码的说明

这个范例使用上一节详细介绍的代码，执行类 Animals 中强类型化的 Animal 对象集合。Main()中的代码仅实例化了一个 Animals 对象 animalCollection，添加了两个项目(它们分别是 Cow 和 Chicken 的实例)，再使用 foreach 循环调用这两个对象继承于基类 Animal 的 Feed()方法。

添加索引符向导

VS 包含另一个用于 C# OOP 的向导，即 Add Indexer 向导，它与添加属性、方法和字段的向导在相同的位置上(在 Class View 窗口中右击工程名，选择 Add | Add Indexer...)。该向导将打开如图 11-5 所示的对话框。

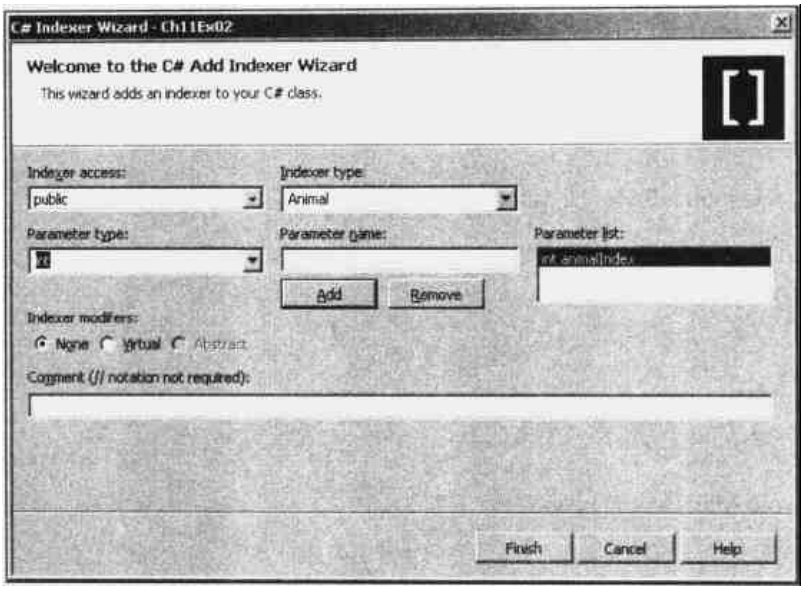


图　11-5

该向导可以指定保护级别(public，表示索引符可以在集合的外部利用)和索引符的类型(这里是 Animal，因为我们希望索引符是强类型化的，并返回 Animal 类的项目)。还可以指定参数的类型和名称；对于本例，只有一个 int 参数，其名称是 animalIndex。最后，可以指定是否为索引符添加修饰符——可以指定索引符是虚拟的或是抽象的(对于抽象类而言)。如果索引符声明为虚拟的，就可以在派生于索引类的任何类上重写该索引符，就像第 10 章的虚拟方法一样。抽象的索引符与虚拟的索引符类似，但我们不提供执行它们的代码——这是由派生于索引类的类提供的。

上述项目将给出用于前面 Animals 类索引符的结构：

```
public Animal this[int animalIndex]
{
    get
    {
        return null;
    }
    set
    {
    }
}
```

注意，我们必须自己填充 get 和 set 代码块，提供执行代码。

2. 关键字值集合和 IDictionary

除了 IList 接口外，集合还可以执行类似的 IDictionary 接口，允许项目通过一个关键字值(例如字符串名)进行索引，而不是通过一个索引。

这也可以使用索引符来完成，但这次的索引符参数是与存储的项目相关联的一个关键字，而不是一个 int 索引，这样集合的用户友好性就更高了。

与索引的集合一样，我们可以使用一个基类简化 IDictionary 接口的执行，这个基类就是

DictionaryBase，它也执行 IEnumerable 和 ICollection 接口，提供了对任何集合都相同的集合处理功能。

DictionaryBase 与 CollectionBase 一样，执行通过其支持的接口获得的一些成员(但不是全部成员)。DictionaryBase 也执行 Clear()和 Count()，但不执行 RemoveAt()。这是因为 RemoveAt()是 IList 接口上的一个方法，不是 IDictionary 接口上的一个方法。但是，Dictionary 有一个 Remove()方法，这是一个应执行基于 DictionaryBase 的定制集合类的方法。

下面的代码是上一节 Animals 类的另一个版本，这次该类派生于 DictionaryBase。执行代码可用于 Add(), Remove()和一个通过关键字访问的索引符：

```
public class Animals : DictionaryBase
{
    public void Add(string newID, Animal newAnimal)
    {
        Dictionary.Add(newID, newAnimal);
    }

    public void Remove(string animalID)
    {
        Dictionary.Remove(animalID);
    }

    public Animals()
    {
    }

    public Animal this[string animalID]
    {
        get
        {
            return (Animal)Dictionary[animalID];
        }
        set
        {
            Dictionary[animalID] = value;
        }
    }
}
```

这些成员的区别如下：

- Add()——有两个参数，一个关键字和一个值，存储在一起。目录集合有一个继承于 DictionaryBase 的成员 Dictionary，这个成员是一个 IDictionary 接口，有自己的 Add()方法，该方法带两个 object 参数。我们的执行代码带一个 string 值作为关键字和一个 Animal 对象。
- Remove()——带一个关键字参数，而不是对象引用。带有指定关键字值的项目被删除。

● Indexer——使用一个字符串关键字值，而不是一个索引，该索引用于通过 Dictionary 的继承成员来访问存储的项目，这里仍需要进行数据类型转换。

基于 DictionaryBase 的集合和基于 CollectionBase 的集合之间的另一个区别是 foreach 的工作方式略有区别。上一节的集合可以直接从集合中提取 Animal 对象。使用 foreach 和 DictionaryBase 派生类可以提供 DictionaryEntry 结构，这是在 System.Collections 名称空间中定义的另一个类型。要得到 Animal 对象本身，就必须使用这个结构的 Value 成员，也可以使用结构的 Key 成员得到相关的关键字。要使代码等价于前面的代码：

```
foreach (Animal myAnimal in animalCollection)
{
    Console.WriteLine("New {0} object added to custom collection, " +
        "Name = {1}", myAnimal.ToString(), myAnimal.Name());
}
```

需要使用下面的代码：

```
foreach (DictionaryEntry myEntry in animalCollection)
{
    Console.WriteLine("New {0} object added to custom collection, " +
        "Name = {1}", myEntry.Value.ToString(),
        ((Animal)myEntry.Value).Name);
}
```

要重写这段代码，可以直接通过 foreach 提取 Animal 对象，这个问题非常复杂，本书不详细论述。

注意：

Karli Watson 撰写的一篇有关该问题的文章发表在 CSharpToday.com Web 站点上。

3. 升级 CardLib(第一部分)

第 10 章创建了一个类库工程 Ch10CardLib，它包含一个表示扑克牌的 Card 类和一个表示一幅扑克的 Deck 类，这个 Deck 类是 Card 类的集合，且执行为一个简单的数组。

本章给这个库添加一个新类，并把类库重命名为 Ch11CardLib。这个新类 Cards 是 Card 对象的一个定制集合，并拥有本章前面介绍的各种功能。这个类的代码在 Cards.cs 中，如下所示(其中，修改了一些向导生成的代码，这些代码被突出显示出来)：

```
using System;
using System.Collections;

namespace Ch11CardLib
{
    /// <summary>
    /// Summary description for Cards.
    /// </summary>
```

```
public class Cards : CollectionBase
{
    public void Add(Card newCard)
    {
        List.Add(newCard);
    }

    public void Remove(Card oldCard)
    {
        List.Remove(oldCard);
    }

    public Cards()
    {
    }

    public Card this[int cardIndex]
    {
        get
        {
            return (Card)List[cardIndex];
        }
        set
        {
            List[cardIndex] = value;
        }
    }

    // Check to see if the Cards collection contains a particular card.
    // This calls the Contains method of the ArrayList for the collection,
    // which we access through the InnerList property.
    public bool Contains(Card card)
    {
        return InnerList.Contains(card);
    }
}
}
```

然后，需要修改 Deck.cs，以利用这个新集合，而不是数组：

```
using System;

namespace Ch11CardLib
{
    public class Deck
    {
```

```
        private Cards cards = new Cards();

        public Deck()
        {
            // line of code removed here.
            for (int suitVal = 0; suitVal < 4; suitVal++)
            {
                for (int rankVal = 1; rankVal < 14; rankVal++)
                {
                    cards.Add(new Card((Suit)suitVal, (Rank)rankVal));
                }
            }
        }

        public Card GetCard(int cardNum)
        {
            if (cardNum >= 0 && cardNum <= 51)
                return cards[cardNum];
            else
                throw (new System.ArgumentOutOfRangeException("cardNum", cardNum,
                    "Value must be between 0 and 51."));
        }

        public void Shuffle()
        {
            Cards newDeck = new Cards();
            bool[] assigned = new bool[52];
            for (int i = 0; i < 52; i++)
            {
                int sourceCard = 0;
                bool foundCard = false;
                Random sourceGen = new Random();
                while (foundCard == false)
                {
                    sourceCard = sourceGen.Next(52);
                    if (assigned[sourceCard] == false)
                        foundCard = true;
                }
                assigned[sourceCard] = true;
                newDeck.Add(cards[sourceCard]);
            }
            cards = newDeck;
        }
    }
}
```

注意：

在把 Ch10CardLib 中的源文件复制到 Ch11CardLib 中时，必须修改名称空间声明，以引用 Ch11CardLib，所以上面的名称空间代码是突出显示的。Card.cs 文件和 Ch10CardClient 控制台应用程序也是这样。

这里不需要进行很多修改。其中的大多数修改都涉及到改变洗牌逻辑，才能把 Cards 中随机的一张牌添加到新 Cards 集合 newDeck 的开头，而不是把 cards 集合中顺序位置的一张牌添加 newDeck 集合的随机位置上。

Ch10CardLib 解决方案的客户控制台应用程序 Ch10CardClient 可以使用这个新库得到与以前相同的结果，因为 Deck 的方法签名没有改变。这个类库的客户程序现在可以使用 Cards 集合类，而不是依赖于 Card 对象数组，例如在扑克牌游戏应用程序中定义一手牌。

11.2　运算符重载

本章要讨论的下一个主题是运算符重载(operator overloading)。可以通过我们设计的类使用标准的运算符，例如+, >等。这称为重载，因为在使用特定的参数类型时，我们为这些运算符提供了自己的执行代码，其方式与重载方法相同，方法的重载是为同名的方法提供不同的参数。

运算符重载非常有用，因为我们可以在运算符重载中执行需要的任何操作，这并不像“把这两个操作数相加”这么简单。稍后介绍一个进一步升级 CardLib 库的范例。我们将提供比较运算符的执行代码，比较两张牌，看看在一圈(扑克牌游戏中的一局)中哪张牌会赢。因为在许多扑克牌游戏中，一圈取决于牌的花色，这并不像比较牌上的数字那样直接。如果第二张牌与第一张牌的花色不同，则无论其点数是什么，第一张牌都会赢。考虑两个操作数的顺序，就可以实现这种比较。也可以考虑“王牌”的花色，而王牌可以胜过其他的花色，即使该王牌的花色与第一张牌不同，也是如此。也就是说，card1 > card2 是 true(这表示如果 card1 是第一个出牌，则 card1 胜过了 card2)，并不意味着 card2 > card1 是 false。如果 card1 和 card2 都不是王牌，且属于不同的花色，则这两个比较都是 true。

但我们先看看运算符重载的基本语法。

要重载运算符，可给类添加运算符类型成员(它们必须是 static)。一些运算符有多种用途，(例如 - 运算符就有一元和二元两种功能)，因此我们还指定了要处理多少个操作数，以及这些操作数的类型。一般情况下，操作数的类型与定义运算符的类相同，但也可以定义处理混合类型的运算符，详见后面的内容。

例如，考虑一个简单的类 AddClass1，如下所示：

```
public class AddClass1
{
    public int val;
}
```

这仅是 int 值的一个包装器(wrapper)，但可以用于说明规则。

对于这个类，下面的代码不能编译：

```
AddClass1 op1 = new AddClass1();
op1.val = 5;
AddClass1 op2 = new AddClass1();
op2.val = 5;
AddClass1 op3 = op1 + op2;
```

其错误是+运算符不能应用于 AddClass1 类型的操作数，因为我们还没有定义要执行的操作。

下面的代码则可执行，但得不到希望的结果：

```
AddClass1 op1 = new AddClass1();
op1.val = 5;
AddClass1 op2 = new AddClass1();
op2.val = 5;
bool op3 = op1 == op2;
```

其中，使用 == 二元运算符来比较 op1 和 op2，看看它们是否引用的是同一个对象，而不是验证它们的值是否相等。在上述代码中，即使 op1.val 和 op2.val 相等，op3 也是 false。

要重载+运算符，可使用下述代码：

```
public class AddClass1
{
    public int val;

    public static AddClass1 operator +(AddClass1 op1, AddClass1 op2)
    {
        AddClass1 returnVal = new AddClass1();
        returnVal.val = op1.val + op2.val;
        return returnVal;
    }
}
```

可以看出，运算符重载看起来与标准静态方法声明类似，但它们使用关键字 operator 和运算符本身，而不是一个方法名。

现在可以成功地使用+运算符和这个类，如上面的范例所示：

```
AddClass1 op3 = op1 + op2;
```

重载所有的二元运算符都是一样的，一元运算符看起来也是类似的，但只有一个参数：

```
public class AddClass1
{
    public int val;

    public static AddClass1 operator +(AddClass1 op1, AddClass1 op2)
    {
        AddClass1 returnVal = new AddClass1();
```

```
        returnVal.val = op1.val + op2.val;
        return returnVal;
    }

    public static AddClass1 operator  - (AddClass1 op1)
    {
        AddClass1 returnVal = new AddClass1();
        returnVal.val = -op1.val;
        return returnVal;
    }
}
```

这两个运算符处理的操作数的类型与类相同，返回值也是该类型，但考虑下面的类定义：

```
public class AddClass1
{
    public int val;

    public static AddClass3 operator +(AddClass1 op1, AddClass2 op2)
    {
        AddClass3 returnVal = new AddClass3();
        returnVal.val = op1.val + op2.val;
        return returnVal;
    }
}

public class AddClass2
{
    public int val;
}

public class AddClass3
{
    public int val;
}
```

下面的代码就可以执行：

```
AddClass1 op1 = new AddClass1();
op1.val = 5;
AddClass2 op2 = new AddClass2();
op2.val = 5;
AddClass3 op3 = op1 + op2;
```

在合适时，可以用这种方式混合类型。但要注意，如果把相同的运算符添加到 AddClass2 中，上面的代码就会失败，因为它将不知道要使用哪个运算符。因此应注意不要把签名相同的

运算符添加到多个类中。

还要注意，如果混合了类型，操作数的顺序必须与运算符重载的参数的顺序相同。如果使用了重载的运算符和顺序错误的操作数，操作就会失败。所以不能像这样使用运算符：

```
AddClass3 op3 = op2 + op1;
```

当然，除非提供了另一个重载运算符和倒序的参数：

```
public static AddClass3 operator +(AddClass2 op1, AddClass1 op2)
{
    AddClass3 returnVal = new AddClass3();
    returnVal.val = op1.val + op2.val;
    return returnVal;
}
```

下述运算符可以重载：

- 一元运算符：　+, -, !, ~, ++, --, true, false
- 二元运算符：　+, -, *, /, %, &, |, ^, <<, >>
- 比较运算符：　==, !=, <, >, <=, >=

注意：

如果重载 true 和 false 运算符，就可以在布尔表达式中使用类，例如 if (op1) {}。

不能重载赋值运算符，例如+=，但这些运算符使用它们的对应运算符，例如+，所以不必担心它们。重载+表示+=按照希望的那样执行。=运算符包含在其中，它对于重载这个运算符没有任何意义，因为它有一个基本的用途。但这个运算符与用户定义的转换运算符相关，详见下一节。

也不能重载&& 和 ||，但它们可以在计算中使用对应的运算符& 和 |，所以重载& 和 |就足够了。

一些运算符如< 和> 必须成对重载。这就是说，不能重载 <，除非也重载了> 。在许多情况下，可以在这些运算符中调用其他运算符，以减少需要的代码(和可能发生的错误)，例如：

```
public class AddClass1
{
    public int val;

    public static bool operator >=(AddClass1 op1, AddClass1 op2)
    {
        return (op1.val >= op2.val);
    }

    public static bool operator <(AddClass1 op1, AddClass1 op2)
    {
        return !(op1 >= op2);
    }
```

```
    // Also need implementations for <= and > operators
}
```

在比较复杂的运算符定义中，这可以减少代码，且只要修改一个执行代码，其他运算符也会被修改。

这同样适用于==和!=，但对于这些运算符，常常需要重写 Object.Equals()和 Object.GetHash Code()，因为这两个函数也可以用于比较对象。重写这些方法，可以确保无论类的用户使用什么技术，都能得到相同的结果。这不太重要，但应增加进来，以保证其完整性。它需要下述非静态重写方法：

```
public class AddClass1
{
    public int val;

    public static bool operator ==(AddClass1 op1, AddClass1 op2)
    {
        return (op1.val == op2.val);
    }

    public static bool operator !=(AddClass1 op1, AddClass1 op2)
    {
        return !(op1 == op2);
    }

    public override bool Equals(object op1)
    {
        return val == ((AddClass1)op1).val;
    }

    public override int GetHashCode()
    {
        return val;
    }
}
```

注意 Equals()使用 object 类型参数。我们需要使用这个签名，否则就将重载这个方法，而不是重写它。类的用户仍可以访问默认的执行代码。这样就必须使用数据类型转换得到需要的结果(虽然这里或许要添加更多的代码，使之比上面的范例更健全，但如果 op1 参数不是一个 AddClass1 实例，或者是这个类的派生类实例，那么该范例就可能得不到预期的结果)。

GetHashCode()可根据其状态，获取对象实例的一个惟一的 int 值。这里使用 val 就可以了，因为它也是一个 int 值。

11.2.1 转换运算符

除了重载如上所述的数学运算符之外，还可以定义类型之间的隐式和显式转换。如果要在不相关的类型之间转换，这是必须的，例如，如果在类型之间没有继承关系，也没有共享接口，这就是必须的。

下面定义 convclass1 和 convclass2 之间的隐式转换，即编写下述代码：

```
convclass1 op1 = new convclass1();
convclass2 op2 = op1;
```

另外，还可以定义一个显式转换，在下面的代码中调用：

```
convclass1 op1 = new convclass1();
convclass2 op2 = (convclass2)op1;
```

例如，考虑下面的代码：

```
public class convclass1
{
    public int val;

    public static implicit operator convclass2(convclass1 op1)
    {
        convclass2 returnVal = new convclass2();
        returnVal.val = op1.val;
        return returnVal;
    }
}

public class convclass2
{
    public double val;

    public static explicit operator convclass1(convclass2 op1)
    {
        convclass1 returnVal = new convclass1();
        checked {returnVal.val = (int)op1.val;};
        return returnVal;
    }
}
```

其中， convclass1 包含一个 int 值，convclass2 包含一个 double 值。因为 int 值可以隐式转换为 double 值，所以可以在 convclass1 和 convclass2 之间定义一个隐式转换。但反过来就不行了，应把 convclass1 和 convclass2 之间的转换定义为显式。

在代码中，用关键字 implicit 和 explicit 来指定这些转换，如上所示。

对于这些类，下面的代码就很好：

```
convclass1 op1 = new convclass1();
op1.val = 3;
convclass2 op2 = op1;
```

但反方向的转换需要下述显式数据类型转换：

```
convclass2 op1 = new convclass2();
op1.val = 3e15;
convclass1 op2 = (convclass1)op1;
```

注意，如果在显式转换中使用了 checked 关键字，则上述代码将产生一个异常，因为 op1 的 val 属性太大，不能放在 op2 的 val 属性中。

11.2.2　升级 CardLib(第二部分)

现在再次升级 Ch11CardLib 工程，给 card 类添加运算符重载。但首先给 card 类添加额外的字段，给王牌指定花色，使 A 有更高的级别。把这些字段指定为静态，因为设置了它们后，它们就可以应用到所有的 card 对象上：

```
public class Card
{
    // Flag for trump usage. If true, trumps are valued higher
    // than cards of other suits.
    public static bool useTrumps = false;

    // Trump suit to use if useTrumps is true.
    public static Suit trump = Suit.Club;

    // Flag that determines whether Aces are higher than Kings or lower
    // than deuces.
    public static bool isAceHigh = true;
```

完成后，就要给 Deck 类再添加几个构造函数，用不同的特性初始化扑克牌：

```
    public Deck()
    {
        for (int suitVal = 0; suitVal < 4; suitVal++)
        {
            for (int rankVal = 1; rankVal < 14; rankVal++)
            {
                cards.Add(new Card((Suit)suitVal, (Rank)rankVal));
            }
        }
    }

    // Non-default constructor. Allows aces to be set high.
    public Deck(bool isAceHigh) : this()
```

```
{
    Card.isAceHigh = isAceHigh;
}

// Non-default constructor. Allows a trump suit to be used.
public Deck(bool useTrumps, Suit trump) : this()
{
    Card.useTrumps = useTrumps;
    Card.trump = trump;
}

// Non-default constructor. Allows aces to be set high and a trump suit
// to be used.
public Deck(bool isAceHigh, bool useTrumps, Suit trump) : this()
{
    Card.isAceHigh = isAceHigh;
    Card.useTrumps = useTrumps;
    Card.trump = trump;
}
```

每个构造函数都使用第 9 章介绍的: this()来定义，这样，无论如何，默认的构造函数总是会在非默认的构造函数之前调用，初始化扑克牌：

接着，给 Card 类添加运算符重载(和推荐的重写运算符)：

```
public Card(Suit newSuit, Rank newRank)
{
    suit = newSuit;
    rank = newRank;
}

public static bool operator ==(Card card1, Card card2)
{
    return (card1.suit == card2.suit) && (card1.rank == card2.rank);
}

public static bool operator !=(Card card1, Card card2)
{
    return !(card1 == card2);
}

public override bool Equals(object card)
{
    return this == (Card)card;
}
public override int GetHashCode()
```

```
{
    return 13*(int)rank + (int)suit;
}

public static bool operator >(Card card1, Card card2)
{
    if (card1.suit == card2.suit)
    {
        if (isAceHigh)
            return (card1.rank > card2.rank) ^ (card2.rank == Rank.Ace);
        else
            return (card1.rank > card2.rank);
    }
    else
    {
        if (useTrumps && (card2.suit == Card.trump))
            return false;
        else
            return true;
    }
}

public static bool operator <(Card card1, Card card2)
{
    return !(card1 >= card2);
}

public static bool operator >=(Card card1, Card card2)
{
    if (card1.suit == card2.suit)
    {
        if (isAceHigh)
            return (card1.rank >= card2.rank) ^ (card2.rank == Rank.Ace);
        else
            return (card1.rank >= card2.rank);
    }
    else
    {
        if (useTrumps && (card2.suit == Card.trump))
            return false;
        else
            return true;
    }
}
```

```
public static bool operator <=(Card card1, Card card2)
{
    return !(card1 > card2);
}
```

这段代码没有什么需要特别注意的，但要注意布尔运算符 XOR (^)的使用略微复杂一些。该运算符用于>运算符重载中。如果单步执行这段重载运算符的代码，就可以看到它的执行情况，明白为什么需要这个运算符。

比较两张牌 card1 和 card2，其中 card1 假定为第一张出的牌。如前所述，在使用王牌时，这是很重要的，因为王牌胜过其他牌，即使非王牌比较大，也是这样。当然，如果两张牌的花色相同，则王牌是否也是该花色就不重要了，所以这是我们要进行的第一个比较：

```
public static bool operator >(Card card1, Card card2)
{
    if (card1.suit == card2.suit)
    {
```

如果静态 isAceHigh 标记为 true，就不能直接通过 Rank 枚举中的值比较牌的点数了。因为 A 的级别在这个枚举中是 1，比其他牌都小。此时就可以利用^运算符了。我们进行了两个比较，一个比较是牌的点数，另一个比较是看看该牌是否为 A。可能的结果如下所示：

- 如果 card1 比 card2 大，且 card2 不是 A，则 card1 较大。
- 如果 card1 比 card2 大，且 card2 是 A，则 card2 较大。
- 如果 card1 比 card2 小，且 card2 不是 A，则 card2 较大。
- 如果 card1 比 card2 小，且 card2 是 A，则 card1 较大。

用一个表来表示，如表 11-1 所示。

表　11-1

card1 > card2	card2.rank == Rank.Ace	card1 比 card2 大?
True	False	True
True	True	False
False	False	False
False	True	True

而 XOR 运算符的执行情况如表 11-2 所示。

表　11-2

op1	op2	op1 ^ op2
True	false	True
True	true	false
False	false	false
False	true	true

其中的逻辑是相同的，所以可以使用该运算符得到需要的结果：

```
            if (isAceHigh)
                return (card1.rank > card2.rank) ^ (card2.rank == Rank.Ace);
```

但如果 isAceHigh 为 false，就可以根据牌的点数来简化操作：

```
            else
                return (card1.rank > card2.rank);
        }
```

代码的其他部分主要考虑的是 card1 和 card2 花色不同的情况。其中静态 useTrumps 标记是非常重要的。如果这个标记是 true，且 card2 是王牌，则可以肯定，card1 不是王牌(因为这两张牌有不同的花色)，王牌总是胜出，所以 card2 比较大：

```
        else
        {
            if (useTrumps && (card2.suit == Card.trump))
                return false;
```

如果 card2 不是王牌(或者 useTrumps 是 false)，则 card1 胜出，因为它是最先出的牌：

```
            else
                return true;
        }
    }
```

还有一个运算符(>=)使用与此类似的代码，其他代码是非常简单的，所以不需要详细分析它们。

下面的简单客户代码测试这些运算符(把它放在客户工程的 Main()函数中进行测试，就像前面 CardLib 范例的客户代码那样)：

```
Card.isAceHigh = true;
Console.WriteLine("Aces are high.");
Card.useTrumps = true;
Card.trump = Suit.Club;
Console.WriteLine("Clubs are trumps.");

Card card1, card2, card3, card4, card5;
card1 = new Card(Suit.Club, Rank.Five);
card2 = new Card(Suit.Club, Rank.Five);
card3 = new Card(Suit.Club, Rank.Ace);
card4 = new Card(Suit.Heart, Rank.Ten);
card5 = new Card(Suit.Diamond, Rank.Ace);

Console.WriteLine("{0} == {1} ? {2}",
    card1.ToString(), card2.ToString(), card1 == card2);
Console.WriteLine("{0} != {1} ? {2}",
```

```
            card1.ToString(), card3.ToString(), card1 != card3);
        Console.WriteLine("{0}.Equals({1}) ? {2}",
            card1.ToString(), card4.ToString(), card1.Equals(card4));
        Console.WriteLine("Card.Equals({0}, {1}) ? {2}",
            card3.ToString(), card4.ToString(), Card.Equals(card3, card4));
        Console.WriteLine("{0} > {1} ? {2}",
            card1.ToString(), card2.ToString(), card1 > card2);
        Console.WriteLine("{0} <= {1} ? {2}",
            card1.ToString(), card3.ToString(), card1 <= card3);
        Console.WriteLine("{0} > {1} ? {2}",
            card1.ToString(), card4.ToString(), card1 > card4);
        Console.WriteLine("{0} > {1} ? {2}",
            card4.ToString(), card1.ToString(), card4 > card1);
        Console.WriteLine("{0} > {1} ? {2}",
            card5.ToString(), card4.ToString(), card5 > card4);
        Console.WriteLine("{0} > {1} ? {2}",
            card4.ToString(), card5.ToString(), card4 > card5);
```

其结果如图 11-6 所示。



图　11-6

在两种情况下，考虑到特定的规则，可应用这些运算符。这在结果的最后四行中很明显，说明了王牌总是胜过非王牌。

11.3　高级转换

现在我们知道如何定义转换运算符，下面再介绍几个 C#语言的特性，完成类型转换的内容。本节主要介绍：

- 封箱(boxing)和拆箱(unboxing)——引用和值类型之间的转换。
- is 运算符——用于查看变量，看看它是否是给定的类型，或者是否与该类型兼容。
- as 运算符——用于把变量转换为指定的类型，这与数据类型转换略有不同。

11.3.1　封箱和拆箱

第 8 章讨论了引用和值类型之间的区别，第 9 章通过比较结构(值类型)和类(引用类型)进行了说明。封箱(boxing)是把值类型转换为 System.Object 类型，或者转换为由值类型执行的接口类型。拆箱(unboxing)是相反的转换过程。

例如，下面的结构类型：

```
struct MyStruct
{
    public int Val;
}
```

可以把这种类型的结构放在 object 类型的变量中，以封箱它：

```
MyStruct valType1 = new MyStruct();
valType1.Val = 5;
object refType = valType1;
```

其中创建了一个类型为 MyStruct 的新变量(valType1)，并把一个值赋予这个结构的 Val 成员，然后把它封箱在 object 类型的变量(refType)中。

以这种方式封箱变量而创建的对象，包含值类型变量的一个副本的引用，而不包含源值类型变量的引用。要进行验证，可以修改源结构的内容，把对象中包含的结构拆箱到新变量中，检查其内容：

```
valType1.Val = 6;
MyStruct valType2 = (MyStruct)refType;
Console.WriteLine("valType2.Val = {0}", valType2.Val);
```

这段代码将得到如下结果：

```
valType2.Val = 5
```

但在把一个引用类型赋予对象时，将执行不同的操作。要进行验证，可以把 MyStruct 改为一个类(不考虑这个类名不再合适的情况)：

```
class MyStruct
{
    public int Val;
}
```

如果不修改上面的客户代码(再次忽略名称错误的变量)，就会得到如下结果：

```
valType2.Val = 6
```

也可以把值类型封箱到一个接口中，只要它们执行这个接口即可。例如，假定 MyStruct 类执行 IMyInterface 接口，如下所示：

```
interface IMyInterface
{
}

struct MyStruct : IMyInterface
{
```

```
    public int Val;
}
```

接着把结构封箱到一个 IMyInterface 类型中，如下所示：

```
MyStruct valType1 = new MyStruct();
IMyInterface refType = valType1;
```

然后使用一般的数据类型转换语法拆箱它：

```
MyStruct ValType2 = (MyStruct)refType;
```

从这些范例中可以看出，封箱是在没有用户干涉的情况下进行的(即不需要编写任何代码)，但拆箱一个值需要进行显式转换，即需要进行数据类型转换(封箱是隐式的，所以不需要进行数据类型转换)。

读者可能想知道为什么要这么做。封箱非常有用，有两个非常重要的原因。首先，它允许使用集合中的值类型(例如 ArrayList)，集合中项目的类型是 object。其次，有一个内部机制允许在值类型上调用 object，例如 int 和结构。

最后要注意的是，在访问值类型的内容前，必须进行拆箱。

11.3.2 is 运算符

is 运算符可以检查未知的变量(该变量可能用作对象参数，传送给一个方法)是否可以转换为给定的类型，如果可以进行转换，该值就是 true。在对对象调用方法前，可以使用该运算符，查看执行该方法的对象的类型。

提示：

is 运算符不会检查两个类型是否相同，但可以检查它们是否兼容。

is 运算符的语法如下：

```
<operand> is <type>
```

这个表达式的结果如下：

- 如果 <type> 是一个类，而<operand>也是该类型，或者它继承了该类型，或者它封箱到该类型中，则结果为 true。
- 如果 <type> 是一个接口类型，而<operand>也是该类型，或者它是执行该接口的类型，则结果为 true。
- 如果 <type> 是一个值类型，而<operand>也是该类型，或者它被拆箱到该类型中，则结果为 true。

下面用几个范例说明如何使用该运算符。

试试看：使用 is 运算符

(1) 在目录 C:\BegCSharp\Chapter11\下创建一个新控制台应用程序 Ch11Ex03。

(2) 修改中 Class1.cs 的代码，如下所示：

```
namespace Ch11Ex03
{
    class Checker
    {
        public void Check(object param1)
        {
            if (param1 is ClassA)
                Console.WriteLine("Variable can be converted to ClassA.");
            else
                Console.WriteLine("Variable can't be converted to ClassA.");

            if (param1 is IMyInterface)
                Console.WriteLine("Variable can be converted to IMyInterface.");
            else
                Console.WriteLine("Variable can't be converted to IMyInterface.");

            if (param1 is MyStruct)
                Console.WriteLine("Variable can be converted to MyStruct.");
            else
                Console.WriteLine("Variable can't be converted to MyStruct.");
        }
    }

    interface IMyInterface
    {
    }

    class ClassA : IMyInterface
    {
    }

    class ClassB : IMyInterface
    {
    }
    class ClassC
    {
    }

    class ClassD : ClassA
    {
    }

    struct MyStruct : IMyInterface
```

```
    {
    }

    /// <summary>
    /// Summary description for Class1.
    /// </summary>
    class Class1
    {
        static void Main(string[] args)
        {
            Checker check = new Checker();
            ClassA try1 = new ClassA();
            ClassB try2 = new ClassB();
            ClassC try3 = new ClassC();
            ClassD try4 = new ClassD();
            MyStruct try5 = new MyStruct();
            object try6 = try5;
            Console.WriteLine("Analyzing ClassA type variable:");
            check.Check(try1);

            Console.WriteLine("\nAnalyzing ClassB type variable:");
            check.Check(try2);
            Console.WriteLine("\nAnalyzing ClassC type variable:");
            check.Check(try3);
            Console.WriteLine("\nAnalyzing ClassD type variable:");
            check.Check(try4);
            Console.WriteLine("\nAnalyzing MyStruct type variable:");
            check.Check(try5);
            Console.WriteLine("\nAnalyzing boxed MyStruct type variable:");
            check.Check(try6);
        }
    }
}
```

(3) 运行代码，其结果如图 11-7 所示。



图　11-7

代码的说明

这个范例说明了使用 is 运算符的各种可能的结果。其中定义了 3 个类、一个接口和一个结构，并把它们用作类的方法的参数，使用 is 运算符确定它们是否可以转换为 ClassA 类型、接口类型和结构类型。

只有 ClassA 和 ClassD(继承了 ClassA)类型与 ClassA 兼容。如果类没有继承一个类，就不会与该类兼容。

ClassA、ClassB 和 MyStruct 类都执行 IMyInterface，所以它们都与 IMyInterface 类兼容，ClassD 继承了 ClassA，所以它们两个也兼容。因此，只有 ClassC 是不兼容的。

最后，只有 MyStruct 类型的变量本身和该类型的封箱变量与 MyStruct 兼容，因为不能把引用类型转换为值类型(当然，我们不能拆箱以前的封箱变量)。

11.3.3　as 运算符

as 运算符使用下面的语法，把一种类型转换为指定的引用类型：

```
<operand> as <type>
```

这只适用于下列情况：

- <operand> 的类型是<type>类型
- <operand> 可以隐式转换为<type>类型
- <operand> 可以封箱到类型 type>中

如果可以从 <operand> 显式转换为<type> ，则表达式的结果就是 null。例如，在上面的代码范例中，可以在 ClassA 和派生于 ClassA 的 ClassD 之间进行显式转换。下面的代码：

```
ClassA obj1 = new ClassA();
ClassD obj2 = obj1 as ClassD;
```

就使 obj2 的结果为 null。

但如果 ClassA 类型的变量包含 ClassD 类型的变量实例，则这个运算符就会起作用：

```
ClassD obj1 = new ClassD();
ClassA obj2 = obj1;
ClassD obj3 = obj2 as ClassD;
```

其中 obj3 包含与 obj1 相同的对象引用，而不是 null。

因此，as 运算符非常有用，因为下面的代码会抛出一个异常：

```
ClassA obj1 = new ClassA();
ClassD obj2 = (ClassD)obj1;
```

但上面的 as 表达式只会把 null 赋予 obj2。这表示，下面的代码(使用本章前面开发的两个类 Animal 和派生于 Animal 的一个类 Cow)在 C#应用程序中是很常见的：

```
public void MilkCow(Animal myAnimal)
{
    Cow myCow = myAnimal as Cow;
    if (myCow != null)
    {
        myCow.Milk();
    }
    else
    {
        Console.WriteLine("{0} isn't a cow, and so can't be milked.",
            myAnimal.Name);
    }
}
```

这要比检查异常的代码简单得多！

11.4 深度复制

第 9 章介绍了如何使用受保护的方法 System.Object.MemberwiseClone()进行阴影复制。使用一个方法，如下所示：

```
public class Cloner
{
    public int Val;

    public Cloner(int newVal)
    {
        Val = newVal;
    }

    public object GetCopy()
    {
        return MemberwiseClone();
    }
}
```

假定有引用类型的域，而不是值类型的域(例如，对象)：

```
public class Content
{
    public int Val;
}

public class Cloner
```

```
{
    public Content MyContent = new Content();

    public Cloner(int newVal)
    {
        MyContent.Val = newVal;
    }

    public object GetCopy()
    {
        return MemberwiseClone();
    }
}
```

此时，通过 GetCopy() 得到的阴影复制有一个域，它引用的对象与源对象相同。
下面的代码使用这个类来说明这一点：

```
Cloner mySource = new Cloner(5);
Cloner myTarget = (Cloner)mySource.GetCopy();
Console.WriteLine("myTarget.MyContent.Val = {0}", myTarget.MyContent.Val);
mySource.MyContent.Val = 2;
Console.WriteLine("myTarget.MyContent.Val = {0}", myTarget.MyContent.Val);
```

第 4 行把一个值赋给 mySource.MyContent.Val，源对象中公共域 MyContent 的公共域 Val 也改变了 myTarget.MyContent.Val 的值。这是因为 mySource.MyContent 引用了与 myTarget.MyContent 相同的对象实例。上述代码的结果如下：

```
myTarget.MyContent.Val = 5
myTarget.MyContent.Val = 2
```

为了说明这个过程，需要执行深度复制。修改上面的 GetCopy()方法就可以进行深度复制，但最好使用.NET Framework 的标准方式。为此，执行 ICloneable 接口，该接口有一个方法 Clone()，这个方法不带参数，返回一个对象类型，其签名和上面使用的 GetCopy() 方法相同。
修改上面使用的类，可以使用下面的深度复制代码：

```
public class Content
{
    public int Val;
}

public class Cloner : ICloneable
{
    public Content MyContent = new Content();

    public Cloner(int newVal)
```

```
    {
        MyContent.Val = newVal;
    }

    public object Clone()
    {
        Cloner clonedCloner = new Cloner(MyContent.Val);
        return clonedCloner;
    }
}
```

其中使用包含在源 Cloner 对象 (MyContent)中的 Content 对象的 Val 字段，创建一个新 Cloner 对象 。这个域是一个值类型，所以不需要深度复制。

使用与上面类似的代码测试阴影复制，但使用 Clone() 而不是 GetCopy()，得到如下结果：

```
myTarget.MyContent.Val = 5
myTarget.MyContent.Val = 5
```

这次包含的对象是独立的。

注意有时在比较复杂的对象系统中，调用 Clone()是一个递归过程。例如，如果 Cloner 类的 MyContent 域也需要深度复制，就要使用下面的代码：

```
public class Cloner : ICloneable
{
    public Content MyContent = new Content();

    ...

    public object Clone()
    {
        Cloner clonedCloner = new Cloner();
        clonedCloner.MyContent = MyContent.Clone();
        return clonedCloner;
    }
}
```

这里调用了默认的构造函数，简化了创建一个新 Cloner 对象的语法。为了使这段代码能正常工作，还需要在 Content 类上执行 ICloneable 接口。

升级 CardLib(第三部分)

下面把上述内容付诸于实践：使用 ICloneable 接口，复制 Card、Cards 和 Deck 对象，这在某些扑克牌游戏中是有用的，因为在这些游戏中不需要让两副扑克牌引用一组相同的 Card 对象，但肯定会使一副扑克牌中的牌序与另一副牌的牌序相同。

在 Ch11CardLib 中，对 Card 类执行复制操作是很简单的，因为只需进行阴影复制(Card 只

包含值类型的数据，其形式为域)。我们只需对类定义进行如下修改：

```
public class Card : ICloneable
{
    public object Clone()
    {
        return MemberwiseClone();
    }
```

注意 ICloneable 接口的这段执行代码只是一个阴影复制，无法确定在 Clone()方法中执行了什么操作，而这正是我们的目的。

接着，需要对 Cards 集合类执行 ICloneable 接口。这个过程略复杂一些，因为涉及到源集合中的每个 Card 对象，所以需要进行深度复制：

```
public class Cards : CollectionBase, ICloneable
{
    public object Clone()
    {
        Cards newCards = new Cards();
        foreach (Card sourceCard in List)
        {
            newCards.Add((Card)sourceCard.Clone());
        }
        return newCards;
    }
```

最后，需要在 Deck 类上执行 ICloneable 接口。这里有一个问题：因为没有洗牌，Deck 类无法修改它包含的扑克牌。例如无法修改有给定牌序的 Deck 实例。为了说明这一点，为 Deck 类定义一个新的私有构造函数，在实例化 Deck 对象时，该函数可以传送指定的 Cards 集合。所以，在这个类中执行复制的代码如下所示：

```
public class Deck : ICloneable
{
    public object Clone()
    {
        Deck newDeck = new Deck((Cards)cards.Clone());
        return newDeck;
    }

    private Deck(Cards newCards)
    {
        cards = newCards;
    }
```

再次用一些简单的客户代码进行测试(与以前一样，这应放在客户工程的 Main()方法中，进

行测试)：

```
Deck deck1 = new Deck();
Deck deck2 = (Deck)deck1.Clone();
Console.WriteLine("The first card in the original deck is: {0}",
      deck1.GetCard(0));
Console.WriteLine("The first card in the cloned deck is: {0}",
      deck2.GetCard(0));
deck1.Shuffle();
Console.WriteLine("Original deck shuffled.");
Console.WriteLine("The first card in the original deck is: {0}",
      deck1.GetCard(0));
Console.WriteLine("The first card in the cloned deck is: {0}",
      deck2.GetCard(0));
```

其结果如图 11-8 所示。



图　11-8

这里要注意的是，当前的扑克牌规则是在 Card 类的静态成员中定义的，应用于每个 Deck 中所有的 Card 对象上。遵循不同规则的每个 Deck 不可能包含相同的牌。但这适用于这个类库，因为可以假定如果一个应用程序要使用不同的规则，就可以包含这些规则，例如在切换牌时，设置 Card 的静态成员。

11.5　定制异常

在本书的前面，讨论了异常，以及如何使用 try...catch...finally 块处理它们。我们还论述了几个标准的.NET 异常，包括异常 System.Exception 的基类。有时在应用程序中，也可以从这个基类中派生自己的异常类，并使用它们，而不是使用标准的异常。这样就可以把更特殊的信息发送给捕获该异常的代码，让处理异常的捕获代码更专业。例如，可以给异常类添加一个属性，允许访问某些基本信息，对异常的接收代码进行必要的改变，或者仅给出异常情况的更多信息。

定义了异常类后，就可以使用 Debug | Exceptions... 菜单项，把它添加到 VS 可以识别的异常列表中。在这个对话框中，可以控制在抛出这种类型的未处理异常时 VS 的响应方式。该对话框可以让 VS 停止执行，启动调试程序，或者继续执行，首先是抛出了这种类型的异常，其次是该异常是否未处理，如图 11-9 所示。

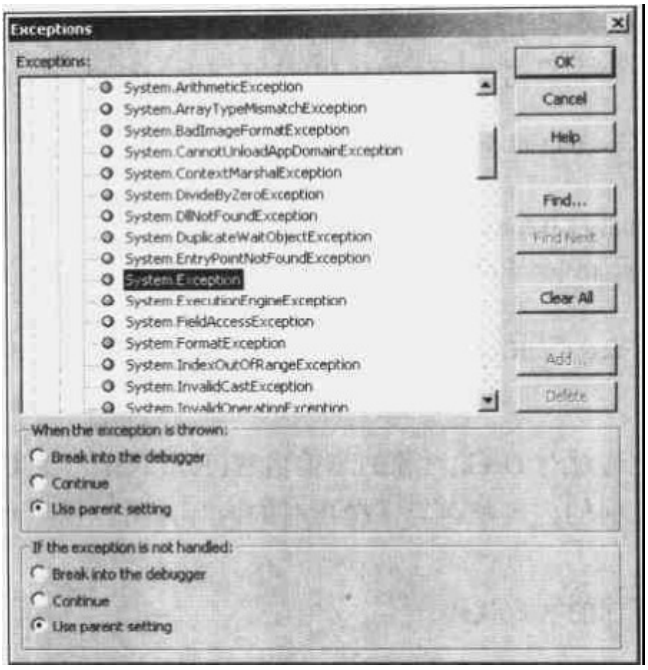


图　11-9

升级 CardLib(第四部分)

定制异常的用法也可以通过升级 Ch11CardLib 工程来说明。Deck.GetCard()方法目前抛出了一个标准的.NET 异常，但下面改为使用一个定制异常。

首先，需要定义该异常。方法是使用在新文件 Exceptions.cs 中定义的一个新类：

```
public class CardOutOfRangeException : Exception
{
    private Cards deckContents;

    public Cards DeckContents
    {
        get
        {
            return deckContents;
        }
    }

    public CardOutOfRangeException(Cards sourceDeckContents) :
            base("There are only 52 cards in the deck.")
    {
        deckContents = sourceDeckContents;
    }
}
```

这个类必须用 Cards 类的一个实例来创建，它可以通过 DeckContents 属性来访问这个 Cards 对象，为 Exception 构造函数提供合适的错误信息，使该错误信息可以通过类的 Message 属性

得到。

接着，在 Deck.cs 中添加抛出该异常的代码(替换原来的标准异常)：

```
public Card GetCard(int cardNum)
{
    if (cardNum >= 0 && cardNum <= 51)
        return cards[cardNum];
    else
        throw new CardOutOfRangeException((Cards)cards.Clone());
}
```

DeckContents 属性是通过对 Deck 对象的当前内容(其形式是一个 Cards 对象)进行深度复制来初始化的。这表示，此时的内容是在异常抛出时的内容，所以以后对 Deck 内容的修改不会丢失这些信息。

要进行测试，使用下面的客户代码：

```
Deck deck1 = new Deck();
try
{
    Card myCard = deck1.GetCard(60);
}
catch (CardOutOfRangeException e)
{
    Console.WriteLine(e.Message);
    Console.WriteLine(e.DeckContents[0]);
}
```

结果如图 11-10 所示。



图　11-10

其中捕获代码把异常的 Message 属性写到屏幕上。我们还通过 DeckContents 显示了 Cards 对象中的第一张牌，以证明可以通过定制异常对象访问 Cards 集合。

11.6　小结

本章介绍的许多技巧都可以使 OOP 应用程序更强大、更有趣。尽管这些技巧要花一定的时间来掌握，但它们可以使类更容易使用，简化了编写其他代码的任务。

本章介绍的每个论题都有许多用途。读者可能在应用程序中见过表格形式的集合，如果要处理类型相同的一组对象，则创建类型安全的集合可以使任务更容易完成。介绍了集合后，我们又介绍了如何添加索引符，以访问集合中的对象。

我们还介绍了运算符重载，运算符重载允许我们的定义运算符(如+和－)处理类的方式。也论述了深度复制，这是避免 OOP 中一个最常见错误的重要方面——把内部数据的引用传送给用户。最后介绍的是如何执行自己的异常对象，把更详细的信息传送给异常处理程序。

第 12 章介绍.NET 中 OOP 最后一个非常重要的方面——事件。

11.7　练习

(1) 创建一个集合类 People，它是下述 Person 类的集合，该集合中的项目可以通过一个字符串索引符来访问，该字符串索引符是人的姓名，与 Person.Name 属性相同：

```
public class Person
{
    private string name;
    private int age;

    public string Name
    {
        get
        {
            return name;
        }
        set
        {
            name = value;
        }
    }

    public int Age
    {
        get
        {
            return age;
        }
        set
        {
            age = value;
        }
    }
}
```

(2) 扩展上一题中的 Person 类，重载>,<, >= 和<= 运算符，比较 Person 实例的 Age 属性。

(3) 给 People 类添加 GetOldest()方法，使用上面定义的重载运算符，返回其 Age 属性的值为最大的 Person 对象数组(1 个或多个对象，因为对于这个属性而言，多个项目可以有相同的值)。

(4) 在 People 类上执行 ICloneable 接口，提供深度复制功能。

第12章　事　　件

本章主要讨论.NET 中最常用的 OOP 技术：事件。

像往常一样，先介绍基础知识：事件到底是什么。之后讨论几个简单的事件，看看使用它们可以做什么。然后论述如何创建和使用自己的事件。

本章的第二部分介绍如何给 CardLib 类库添加一个事件，使该类库更完整。另外，因为这是在介绍一些更高级论题之前的最后一部分，所以举了一些范例。我们还将创建一个使用该类库的扑克牌游戏应用程序。

首先介绍事件。

12.1　什么是事件

事件类似于异常，因为它们都由对象引发，我们可以提供代码来处理事件。但它们也有几个重要的区别。最重要的区别是并没有与 try...catch 类似的结构来处理事件，而必须订阅(subscribe)它们。订阅一个事件的含义是提供代码，在事件发生时执行这些代码，它们称为事件处理程序。

事件可以有许多订阅的处理程序，在该事件发生时，这些处理程序都会被调用，其中包括引发该事件的对象所在的类的事件处理程序，但事件处理程序也可能在其他类中。

事件处理程序本身都是简单的函数。对事件处理函数的惟一限制是它必须匹配于事件所要求的签名(返回类型和参数)。这个签名是事件定义的一部分，由一个委托指定。

提示：

使用在事件中的委托是非常有用的。因此第 6 章对此进行了论述，读者可以温习这一部分，复习一下委托是什么和如何使用它们。

处理过程如下所示：

首先，应用程序创建一个可以引发事件的对象。例如，假定应用程序是一个即时信息传送(instant messaging)应用程序，它创建的对象表示一个远程用户的连接。当接收到通过该连接从远程用户传送来的信息时，这个连接对象会引发一个事件，如图 12-1 所示。

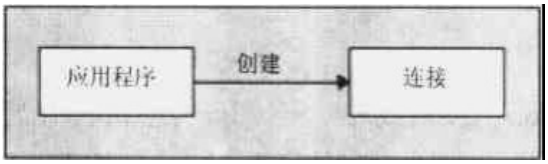


图　12-1

接着，应用程序订阅事件。为此，即时信息传送应用程序将定义一个函数，该函数可以与事件指定的委托类型一起使用，把这个函数的一个引用传送给事件，而事件的处理函数可以是另一个对象的方法，假定是表示显示设备的对象，当接收到信息时，该方法将显示即时信息，如图 12-2 所示。

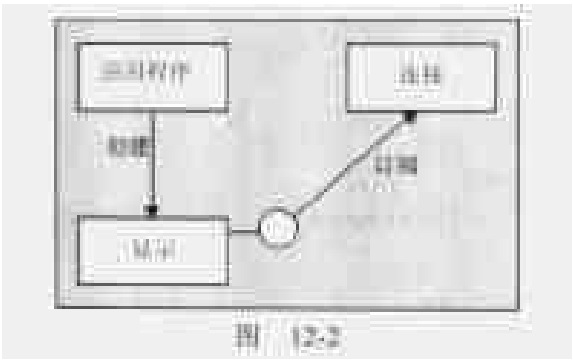


图 12-2

引发事件后，就通知订阅器。当接收到通过连接对象传送过来的即时信息时，就调用显示设备对象上的事件处理方法。因为我们使用的是一个标准方法，所以引发事件的对象可以通过参数传送任何相关的信息，这样就大大增加了事件的通用性。在本例中，参数是即时信息的文本，使用显示设备对象的事件处理程序可以显示它，如图 12-3 所示。

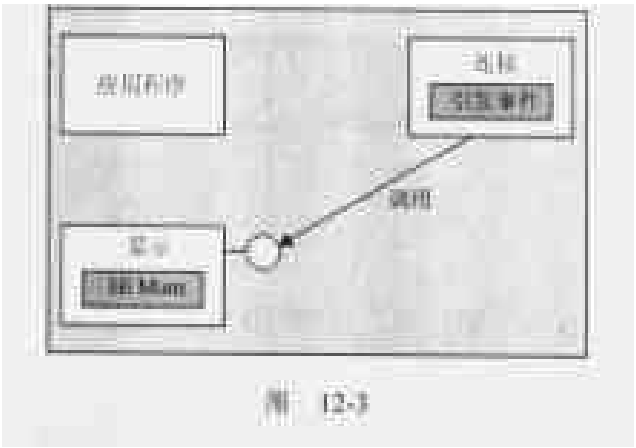


图 12-3

12.2 使用事件

本节介绍处理事件所需要的代码，然后讨论如何定义和使用自己的事件。

12.2.1 处理事件

如前所述，要处理事件，需要提供一个事件处理函数，订阅事件，该函数的签名应匹配于事件指定的委托签名。下面介绍的范例使用一个简单的计时器对象引发事件，调用一个处理函数。

试试看：处理事件

(1) 在目录 C:\BegCSharp\Chapter12\下创建一个新的控制台应用程序 Ch12Ex01。

(2) 修改 Class1.cs 中的代码，如下所示：

```
using System;
```

```
using System.Timers;

namespace Ch12Ex01
{
    class Class1
    {
        static int counter = 0;

        static string displayString =
                                    "This string will appear one letter at a time. ";

        static void Main(string[] args)
        {
            Timer myTimer = new Timer(100);
            myTimer.Elapsed += new ElapsedEventHandler(WriteChar);
            myTimer.Start();
            Console.ReadLine();
        }

        static void WriteChar(object source, ElapsedEventArgs e)
        {
            Console.Write(displayString[counter++ % displayString.Length]);
        }
    }
}
```

运行应用程序(启动后，按回车键将终止程序的执行)，其结果如图 12-4 所示。

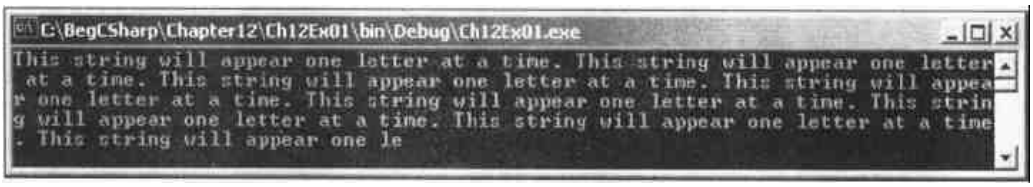


图　12-4

代码的说明

用于引发事件的对象是 System.Timers.Timer 类的一个实例。使用一个时间段(以毫秒为单位)来初始化该对象，当使用 Start()方法启动 Timer 对象时，就引发一系列事件，根据指定的时间段来引发事件。因为 Main()用 100 毫秒初始化 Timer 对象，所以在启动该对象后，1 秒钟内将引发 10 次事件：

```
static void Main(string[] args)
{
    Timer myTimer = new Timer(100);
```

Timer 对象有一个 Elapsed 事件，如果使用 Object Browser 窗口，就可以看到它，如图 12-5

所示。

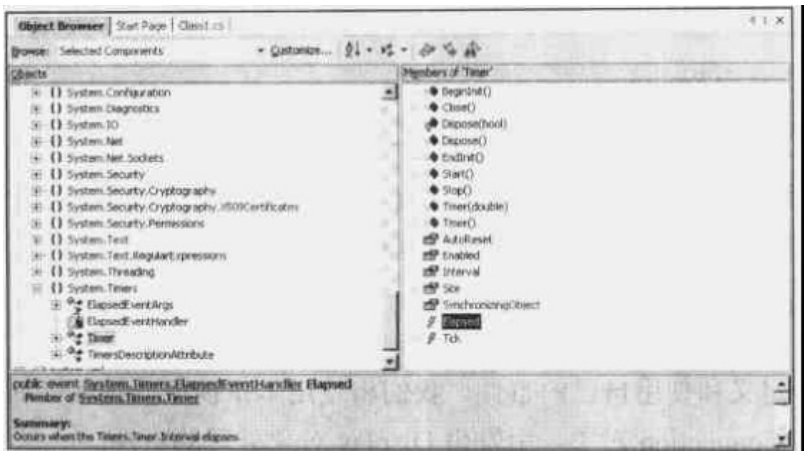


图 12-5

这个事件要求的事件处理程序签名是 System.Timers.ElapsedEventHandler 委托类型，该委托是在.NET Framework 中定义的标准委托之一，用于与下述签名匹配的函数：

```
void functionName(object source, ElapsedEventArgs e);
```

Timer 对象的第一个参数是它本身的引用，第二个参数则是 ElapsedEventArgs 对象的一个实例。现在可以不考虑这些参数，后面将论述它们。

在代码中，有一个匹配该签名的方法：

```
static void WriteChar(object source, ElapsedEventArgs e)
{
    Console.Write(displayString[counter++ % displayString.Length]);
}
```

这个方法使用 Class1 的两个静态字段 counter 和 displayString，显示一个字符。每次调用方法时，显示的字符都不相同。

下一个任务是把这个处理程序与事件关联起来——即订阅它。为此，可以使用+=运算符，给事件添加一个处理程序，其形式是用事件处理程序方法初始化的一个新委托实例：

```
static void Main(string[] args)
{
    Timer myTimer = new Timer(100);
    myTimer.Tick += new EventHandler(WriteChar);
```

这个命令(使用略微古怪的语法，专用于委托)在列表中添加一个处理程序，当引发 Elapsed 事件时，就会调用该处理程序。可以给这个列表添加任意多个处理程序，只要它们满足指定的条件即可。当引发事件时，每个处理程序都会被依次调用。

剩下的任务是启动计时器的 Main()：

```
    myTimer.Start();
```

我们不想在处理完任何事件前终止应用程序，所以要让 Main()函数一直执行。最简单的方式是请求用户输入，因为这个命令要在用户输入一行文本和/或按下回车键后，才会停止处理。

```
        Console.ReadLine();
    }
```

在这里，Main()中的处理会停止，但 Timer 对象中的处理将继续。当该对象引发事件时，就调用 WriteChar()方法，同时该方法运行 Console.ReadLine()语句。

12.2.2 定义事件

接着论述如何定义和使用自己的事件。我们将使用本章前面介绍的即时信息传送应用程序范例，并创建一个 Connection 对象，引发由 Display 对象处理的事件。

试试看：定义事件

(1) 在目录 C:\BegCSharp\Chapter12\下创建一个新控制台应用程序 Ch12Ex02。

(2) 添加一个新类 Connection，存储在 Connection.cs 中：

```
using System;
using System.Timers;

namespace Ch12Ex02
{
    public delegate void MessageHandler(string messageText);

    public class Connection
    {
        public event MessageHandler MessageArrived;

        private Timer pollTimer;

        public Connection()
        {
            pollTimer = new Timer(100);
            pollTimer.Elapsed += new ElapsedEventHandler(CheckForMessage);
        }

        public void Connect()
        {
            pollTimer.Start();
        }

        public void Disconnect()
        {
            pollTimer.Stop();
```

```
        }

        private void CheckForMessage(object source, ElapsedEventArgs e)
        {
            Console.WriteLine("Checking for new messages.");
            Random random = new Random();
            if ((random.Next(9) == 0) && (MessageArrived != null))
            {
                MessageArrived("Hello Mum!");
            }
        }
    }
}
```

(3) 添加一个新类 Display，存储在 Display.cs 中：

```
using System;

namespace Ch12Ex02
{
    public class Display
    {
        public void DisplayMessage(string message)
        {
            Console.WriteLine("Message arrived: {0}", message);
        }
    }
}
```

(4) 修改 Class1.cs 中的代码，如下所示：

```
using System;

namespace Ch12Ex02
{
    class Class1
    {
        static void Main(string[] args)
        {
            Connection myConnection = new Connection();
            Display myDisplay = new Display();
            myConnection.MessageArrived +=
                    new MessageHandler (myDisplay.DisplayMessage);
            myConnection.Connect();
            Console.ReadLine();
        }
```

```
        }
    }
```

(5) 运行应用程序，其结果如图 12-6 所示。



图　12-6

代码的说明

在这个应用程序中，完成大部分工作的是 Connection 类。这个类的实例使用如本章第一个范例中所示的 Timer 对象，在类的构造函数中初始化它，并通过 Connect() 和 Disconnect()访问它的状态(可访问和禁止访问)：

```
public class Connection
{
    private Timer pollTimer;

    public Connection()
    {
        pollTimer = new Timer(100);
        pollTimer.Elapsed += new ElapsedEventHandler(CheckForMessage);
    }

    public void Connect()
    {
        pollTimer.Start();
    }

    public void Disconnect()
    {
        pollTimer.Stop();
    }

    ...
}
```

在构造函数中，我们还以与第一个范例相同的方式注册了 Elapsed 事件的一个事件处理程序。每调用这个处理程序方法 CheckForMessage()10 次后，就会引发一个事件。在分析它的代码前，先看看事件的定义。

在定义事件前，必须先定义一个委托类型，以用于该事件，这个委托类型指定了事件处理方法必须遵循的签名。为此，我们使用标准的委托语法，在 Ch11Ex02 名称空间中将该委托定义为公共类型，使该类型可供外部代码使用：

```
namespace Ch12Ex02
{
   public delegate void MessageHandler(string messageText);
```

这个委托类型称为 MessageHandler，是 void 函数的签名，它有一个 string 参数。使用这个参数可以把 Connection 对象接收过来的即时信息发送给 Display 对象。

定义了委托后(或者定位合适的现有委托)，就可以把事件本身定义为 Connection 类的一个成员：

```
public class Connection
{
   public event MessageHandler MessageArrived;
```

给事件命名(这里使用名称 MessageArrived)，用 event 关键字和要使用的委托类型(前面定义的 MessageHandler 委托类型)声明它。

以这种方式声明了事件后，就可以引发它，方法是按名称来调用它，就好像它是一个其签名是由委托指定的方法一样。例如，使用下面的代码引发这个事件：

```
MessageArrived("This is a message.");
```

如果定义该委托不包含任何参数，就可以使用下面的代码：

```
MessageArrived();
```

如果定义了较多的参数，就需要比较多的代码来引发事件。

CheckForMessage()方法如下所示：

```
private void CheckForMessage(object source, ElapsedEventArgs e)
{
   Console.WriteLine("Checking for new messages.");
   Random random = new Random();
   if ((random.Next(9) == 0) && (MessageArrived != null))
   {
      MessageArrived("Hello Mum!");
   }
}
```

使用前面几章中的 Random 类实例，以生成一个 0～9 之间的随机数，如果该随机数为 0，就引发一个事件，它的发生几率为 10%。这类似于轮流检测连接，看看是否接收到信息，不可

能每次检测时，都没有接收到信息。

注意，这里还提供了其他逻辑。如果表达式 MessageArrived != null 为 true，就只引发一个事件。这个表达式也使用了委托语法，但语法略有不同，其含义是“事件是否有订阅器？”。如果没有订阅器，MessageArrived 就是 null，也就不会引发事件。

订阅事件的类是 Display，它包含一个方法 DisplayMessage()，其定义如下所示：

```
public class Display
{
    public void DisplayMessage(string message)
    {
        Console.WriteLine("Message arrived: {0}", message);
    }
}
```

这个方法匹配于委托类型方法的签名(而且是个公共方法，如果类不是生成该事件的类，则其事件处理程序的要求就有所不同)，所以可以使用它响应 MessageArrived 事件。

剩下的是 Main()中的代码初始化了 Connection 和 Display 类的实例，把它们关联起来，开始执行任务。这里需要的代码类似于第一个范例中的代码：

```
static void Main(string[] args)
{
    Connection myConnection = new Connection();
    Display myDisplay = new Display();
    myConnection.MessageArrived +=
            new MessageHandler(myDisplay.DisplayMessage);
    myConnection.Connect();
    Console.ReadLine();
}
```

再次调用 Console.ReadLine()，当开始执行 Connection 对象的 Connect()方法时，暂停 Main()的处理。

1. 多用途的事件处理程序

前面的 Timer.Elapse 事件的签名包含了事件处理程序常见的两个参数，如下所示：

- object source——引发事件的对象的引用
- ElapsedEventArgse——由事件传送的参数

在这个事件(以及许多其他的事件)中使用 Object 类型参数的原因是，我们常常要为由不同对象引发的几个相同事件使用同一个事件处理程序，但仍要指定哪个对象引发了事件。

要说明这一点，下面扩展上一个范例。

试试看：使用多用途的事件处理程序

(1) 在目录 C:\BegCSharp\Chapter12\下创建一个新控制台应用程序 Ch12Ex03。

(2) 把 Ch12Ex02 中的代码复制到 Class1.cs、Connection.cs 和 Display.cs 中，并把每个文件

中的 Ch12Ex02 名称空间改为 Ch12Ex03。

(3) 添加一个新类 MessageArrivedEventArgs，将它存储在 MessageArrivedEventArgs.cs 中：

```
using System;

namespace Ch12Ex03
{
    public class MessageArrivedEventArgs : EventArgs
    {
        private string message;

        public string Message
        {
            get
            {
                return message;
            }
        }

        public MessageArrivedEventArgs()
        {
            message = "No message sent.";
        }

        public MessageArrivedEventArgs(string newMessage)
        {
            message = newMessage;
        }
    }
}
```

(4) 修改 Connection.cs，如下所示：

```
namespace Ch12Ex03
{
    public delegate void MessageHandler(Connection source,
                                        MessageArrivedEventArgs e);

    public class Connection
    {
        public event MessageHandler MessageArrived;

        private string name;

        public string Name
        {
```

```
            get
            {
                return name;
            }
            set
            {
                name = value;
            }
        }

        ...

        private void CheckForMessage(object source, EventArgs e)
        {
            Console.WriteLine("Checking for new messages.");
            Random random = new Random();
            if ((random.Next(9) == 0) && (MessageArrived != null))
            {
                MessageArrived(this, new MessageArrivedEventArgs("Hello Mum!"));
            }
        }

        ...

    }
}
```

(5) 修改 Display.cs，如下所示：

```
        public void DisplayMessage(Connection source, MessageArrivedEventArgs e)
        {
            Console.WriteLine("Message arrived from: {0}", source.Name);
            Console.WriteLine("Message Text: {0}", e.Message);
        }
```

(6) 修改 Class1.cs，如下所示：

```
        static void Main(string[] args)
        {
            Connection myConnection1 = new Connection();
            myConnection1.Name = "First connection.";
            Connection myConnection2 = new Connection();
            myConnection2.Name = "Second connection.";
            Display myDisplay = new Display();
            myConnection1.MessageArrived +=
                                new MessageHandler(myDisplay.DisplayMessage);
```

```
        myConnection2.MessageArrived -=
                        new MessageHandler(myDisplay.DisplayMessage);
        myConnection1.Connect();
        myConnection2.Connect();
        Console.ReadLine();
    }
```

(7) 运行应用程序，其结果如图 12-7 所示。



图　12-7

代码的说明

发送一个引发事件的对象引用，将其作为事件处理程序的一个参数，就可以为不同的对象定制处理程序的响应。利用该引用可以访问源对象，包括它的属性。

通过发送包含在派生于 System.EventArgs(与 ElapsedEventArgs 相同)的类中的参数，就可以将其他必要信息作为参数(例如 MessageArrivedEventArgs 类上的 Message 参数)。

另外，这些参数也将得益于多态性。可以为 MessageArrived 事件定义一个处理程序，如下所示：

```
public void DisplayMessage(object source, EventArgs e)
{
    Console.WriteLine("Message arrived from: {0}",
                      ((Connection)source).Name);
    Console.WriteLine("Message Text: {0}",
                      ((MessageArrivedEventArgs)e).Message);
}
```

修改 Connection.cs 中的委托定义，如下所示：

```
public delegate void MessageHandler(object source, EventArgs e);
```

这个应用程序将像以前那样执行，但 DisplayMessage()函数的通用性更高(至少在理论上是这样 —— 需要用更多的执行代码提高产品的质量)。这个处理程序还可以处理其他事件，例如 Timer.Elapsed 事件，但必须修改处理程序的内部代码，这样，在引发这个事件时，发送过来的

参数才会正确地被处理(以这种方式把它们转换为 Connection 和 MessageArrivedEventArgs 对象，会抛出一个异常，所以这里应使用 as 运算符)。

在继续前，注意这个范例的输出结果是引发了两个 Connection 对象的事件系列。这是因为 Random 类产生随机数的方式所造成的。在实例化 Random 类型的对象时，是用一个随机数种子启动的。该种子用于从某个复杂的等式中生成一组伪随机数(计算机不能生成真正的随机数)。我们可以在构造函数中为 Random 实例指定一个随机数种子，但如果没有指定(如本例所示)，就使用当前时间作为种子。在连续的几行代码中实例化和启动两个 Connection 对象后，当响应 Timer.Elapsed 事件时，可能会得到两个相同的随机数种子。所以，如果其中一个随机数种子生成了一个信息，另一个种子也会生成该信息。这就是一个指令处理的速度问题了。要说明这个问题，必须以不同的方式设置随机数种子值。一种解决方案是(这里没有采用)提供在两个 Connection 对象中都可以访问的 Random 类实例。然后使用相同的随机数序列，就不会产生同步的信息。

2. 返回值和事件处理程序

前面的所有的事件处理程序都有 void 类型的返回值。可以为事件提供返回类型，但这会出问题。这是因为引发给定的事件，可能会调用好几个事件处理程序。如果这些处理程序都返回一个值，那么我们该使用哪个返回值？

系统处理这个问题的方式是，只允许访问由事件处理程序最后返回的那个值，也就是最后一个订阅该事件的处理程序返回的值。

这个功能在某些情况下是有用的，但这里不讨论它们。最后使用 void 类型的事件处理程序，且避免使用 out 类型的参数。

12.3 扩展和使用 CardLib

前面介绍了事件的定义和使用，下面添加第 11 章开发的类库 Ch11CardLib。与以前一样，使用类库工程 Ch12CardLib，它最初的代码与 Ch11CardLib 相同(但名称空间是 Ch12CardLib，而不是 Ch11CardLib)。

当使用 GetCard 获得 Deck 对象中的最后一个 Card 对象，就将引发添加到该类库中的事件 LastCardDrawn。这个事件允许订阅器重新洗牌，停止客户要求的处理。为这个事件(LastCardDrawnHandler)定义的委托需要为 Deck 对象提供一个引用，这样无论处理程序在什么地方，都可以在该处理程序中访问 Shuffle()方法。

```
namespace Ch12CardLib
{
   public delegate void LastCardDrawnHandler(Deck currentDeck);
```

定义和引发事件的代码比较简单，如下所示：

```
public event LastCardDrawnHandler LastCardDrawn;
```

```
public Card GetCard(int cardNum)
{
    if (cardNum >= 0 && cardNum <= 51)
    {
        if ((cardNum == 51) && (LastCardDrawn != null))
            LastCardDrawn(this);
        return cards[cardNum];
    }
    else
        throw new CardOutOfRangeException((Cards)cards.Clone());
}
```

这是把事件添加到 Deck 类定义所需要的所有代码。下面只需要使用它就可以了。

CardLib 的扑克牌游戏客户程序

在开发了 CardLib 库后，就可以使用它了。在结束讲述 C#和.NET Framework 中 OOP 技术的这个部分前，我们将编写扑克牌应用程序的基本代码，使用我们熟悉的扑克牌类。

与前面的章节一样，我们将在 Ch12CardLib 解决方案中添加一个客户控制台应用程序，添加一个 Ch12CardLib 工程的引用，使之成为启动工程。这个应用程序叫作 Ch12CardClient。

首先，在 Ch12CardClient 的一个新文件 Player.cs 中创建一个新类 Player，这个类包含一个私有 Cards 域 hand，一个私有字符串域 name，以及两个只读属性 Name 和 PlayHand。这些属性只能访问私有域。注意，即使 PlayHand 属性是只读的，但仍可以访问返回域 hand 的引用，修改玩家手中的扑克牌。

我们还把默认的构造函数设置为私有，以隐藏它，并提供了一个公共的非默认构造函数，接受 Player 实例中属性 Name 的初始值。

最后，提供一个 bool 类型的方法 HasWon()。如果玩家手中的扑克牌都是相同的花色(一个简单的取胜条件，但并没有什么意义)，该方法就返回 true。

Player.cs 的代码如下所示：

```
using System;
using Ch12CardLib;

namespace Ch12CardClient
{
    public class Player
    {
        private Cards hand;
        private string name;

        public string Name
        {
            get
            {
```

```
            return name;
        }
    }

    public Cards PlayHand
    {
        get
        {
            return hand;
        }
    }

    private Player()
    {
    }

    public Player(string newName)
    {
        name = newName;
        hand = new Cards();
    }

    public bool HasWon()
    {
        bool won = true;
        Suit match = hand[0].suit;
        for (int i = 1; i < hand.Count; i++)
        {
            won &= hand[i].suit == match;
        }
        return won;
    }
  }
}
```

接着定义一个处理扑克牌游戏的类 Game，这个类在 Ch12CardClient 工程的 Game.cs 文件中。这个类有 4 个私有成员字段：

- playDeck——Deck 类型的变量，包含要使用的一副扑克牌
- currentCard——一个 int 值，用作下一张要翻开的扑克牌的指针
- players——一个 Player 对象数组，表示玩家
- discardedCards——Cards 集合，表示玩家扔掉的扑克牌，但还没有放回整副牌中。

这个类的默认构造函数初始化了存储在 playDeck 中的 Deck，并洗牌，把 currentCard 指针变量设置为 0(playDeck 中的第一张牌)，并引发 playDeck.LastCardDrawn 事件的处理程序 Reshuffle()。这个处理程序将洗牌，包括 discardedCards 中的牌，并把 currentCard 重置为 0，准

备从新的一副牌中读取扑克牌。

Game 类还包含两个实用程序方法：SetPlayers()可以设置游戏的玩家(Player 对象数组)，DealHands()可以处理玩家手中的牌(每个玩家有 7 张牌)。玩家的数量限制为 2 到 7 人，确保每个玩家有足够的牌。

最后，PlayGame()方法包含游戏的逻辑。在分析了 Class1.cs 中的代码后介绍这个函数，Game.cs 的剩余代码如下所示：

```
using System;
using Ch12CardLib;

namespace Ch12CardClient
{
    public class Game
    {
        private int currentCard;
        private Deck playDeck;
        private Player[] players;
        private Cards discardedCards;

        public Game()
        {
            currentCard = 0;
            playDeck = new Deck(true);
            playDeck.LastCardDrawn += new LastCardDrawnHandler(Reshuffle);
            playDeck.Shuffle();
        }

        private void Reshuffle(Deck currentDeck)
        {
            currentDeck.Shuffle();
            discardedCards = new Cards();
            currentCard = 0;
        }

        public void SetPlayers(Player[] newPlayers)
        {
            if (newPlayers.Length > 7)
                throw new ArgumentException("A maximum of 7 players may play this" +
                                            " game.");

            if (newPlayers.Length < 2)
                throw new ArgumentException("A minimum of 2 players may play this" +
                                            " game.");
```

```
            players = newPlayers;
        }

        private void DealHands()
        {
            for (int p = 0; p < players.Length; p++)
            {
                for (int c = 0; c < 7; c++)
                {
                    players[p].PlayHand.Add(playDeck.GetCard(currentCard++));
                }
            }
        }

        public int PlayGame()
        {
            // Code to follow.
        }
    }
}
```

Class1.cs 包含 Main() 函数，它启动和运行游戏。这个函数执行下述步骤：

- 显示引导画面。
- 给用户显示 2 到 7 个玩家。
- 建立一个 Player 对象数组。
- 给每个玩家起个名字，用于初始化数组中的 Player 对象。
- 创建一个 Game 对象，使用 SetPlayers()方法指定玩家。
- 使用 PlayGame()方法启动游戏。
- PlayGame() 的 int 返回值用于显示一个警告信息(返回的值是 Player 对象数组中获胜的玩家索引)。

这个函数的代码(为了整洁起见，加上了一些注释)如下所示：

```
        static void Main(string[] args)
        {
            // Display introduction.
            Console.WriteLine("KarliCards: a new and exciting card game.");
            Console.WriteLine("To win you must have 7 cards of the same suit in" +
                              " your hand.");
            Console.WriteLine();

            // Prompt for number of players.
            bool inputOK = false;
            int choice = -1;
            do
```

```
{
    Console.WriteLine("How many players (2-7)?");
    string input = Console.ReadLine();
    try
    {
        // Attempt to convert input into a valid number of players.
        choice = Convert.ToInt32(input);
        if ((choice >= 2) && (choice <= 7))
            inputOK = true;
    }
    catch
    {
        // Ignore failed conversions, just continue prompting.
    }
} while (inputOK == false);

// Initialize array of Player objects.
Player[] players = new Player[choice];

// Get player names.
for (int p = 0; p < players.Length; p++)
{
    Console.WriteLine("Player {0}, enter your name:", p + 1);
    string playerName = Console.ReadLine();
    players[p] = new Player(playerName);
}

// Start game.
Game newGame = new Game();
newGame.SetPlayers(players);
int whoWon = newGame.PlayGame();

// Display winning player.
Console.WriteLine("{0} has won the game!", players[whoWon].Name);
}
```

接着看看应用程序的主体 PlayGame()。这里不打算详细讲解这个方法，而只是对它进行一定的注释，使之更容易理解。实际上，这些代码都不复杂，仅是比较多而已。

每个玩家都可以查看手中的牌和桌面上的一张翻开的牌。他们可以拾取这张牌，或者翻开一张新牌。在拾取一张牌后，玩家必须扔掉一张牌，如果他们拾取了桌面上的那张牌，就必须用另一张牌替换桌面上的那张牌，或者把扔掉的那张牌放在桌面上那张牌的上面(把桌面上原来的那张牌添加到 discardedCards 集合中)。

在分析这段代码时，一个关键的问题是处理 Card 对象的方式。必须清楚，这些对象定义为引用类型，而不是值类型(使用结构)。给定的 Card 对象似乎同时存在于多个地方，因为引

用可以存在于 Deck 对象、Player 对象的 hand 字段、discardedCards 集合和 playCard 对象(桌面上的当前牌)中。这样就很容易跟踪扑克牌，特别是可以用于从一副牌中拾取一张新牌。如果牌不在任何玩家的手中，不在桌面上，也不是 discardedCards 集合中，才能接受该牌。

代码如下所示：

```
public int PlayGame()
{
    // Only play if players exist.
    if (players == null)
        return -1;

    // Deal initial hands.
    DealHands();

    // Initialize game vars, including an initial card to place on the
    // table: playCard.
    bool GameWon = false;
    int currentPlayer;
    Card playCard = playDeck.GetCard(currentCard++);

    // Main game loop, continues until GameWon == true.
    do
    {
        // Loop through players in each game round.
        for (currentPlayer = 0; currentPlayer < players.Length;
             currentPlayer++)
        {
            // Write out current player, player hand, and the card on the
            // table.
            Console.WriteLine("{0}'s turn.", players[currentPlayer].Name);
            Console.WriteLine("Current hand:");
            foreach (Card card in players[currentPlayer].PlayHand)
            {
                Console.WriteLine(card);
            }
            Console.WriteLine("Card in play: {0}", playCard);

            // Prompt player to pick up card on table or draw a new one.
            bool inputOK = false;
            do
            {
                Console.WriteLine("Press T to take card in play or D to " +
                                  "draw:");
                string input = Console.ReadLine();
                if (input.ToLower() == "t")
```

```
    {
        // Add card from table to player hand.
        Console.WriteLine("Drawn: {0}", playCard);
        players[currentPlayer].PlayHand.Add(playCard);
        inputOK = true;
    }
    if (input.ToLower() == "d")
    {
        // Add new card from deck to player hand.
        Card newCard;
        // Only add card if it isn't already in a player hand.
        bool cardIsInPlayerHand;
        do
        {
            newCard = playDeck.GetCard(currentCard++);
            cardIsInPlayerHand = false;
            // Loop through all player hands to see if newCard is
            // already in a hand.
            foreach (Player testPlayer in players)
            {
                cardIsInPlayerHand |=
                    testPlayer.PlayHand.Contains(newCard);
            }
        } while (cardIsInPlayerHand);
        // Add the card found to player hand.
        Console.WriteLine("Drawn: {0}", newCard);
        players[currentPlayer].PlayHand.Add(newCard);
        inputOK = true;
    }
} while (inputOK == false);

// Display new hand with cards numbered.
Console.WriteLine("New hand:");
for (int i = 0; i < players[currentPlayer].PlayHand.Count; i++)
{
    Console.WriteLine("{0}: {1}", i + 1,
                      players[currentPlayer].PlayHand[i]);
}

// Prompt player for a card to discard.
inputOK = false;
int choice = -1;
do
{
    Console.WriteLine("Choose card to discard:");
```

```
                string input = Console.ReadLine();
                try
                {
                    // Attempt to convert input into a valid card number.
                    choice = Convert.ToInt32(input);
                    if ((choice > 0) && (choice <= 8))
                        inputOK = true;
                }
                catch
                {
                    // Ignore failed conversions, just continue prompting.
                }
            } while (inputOK == false);

            // Place reference to removed card in playCard (place the card
            // on the table), then remove card from player hand.
            playCard = players[currentPlayer].PlayHand[choice - 1];
            players[currentPlayer].PlayHand.RemoveAt(choice - 1);
            Console.WriteLine("Discarding: {0}", playCard);

            // Space out text for players
            Console.WriteLine();

            // Check to see if player has won the game, and exit the player
            // loop if so.
            GameWon = players[currentPlayer].HasWon();
            if (GameWon == true)
                break;
        }
    } while (GameWon == false);

    // End game, noting the winning player.
    return currentPlayer;
}
```

玩这个游戏，确保花一定的时间研究它。应尝试在 Reshuffle()方法中设置一个断点，由 7 个玩家来玩这个游戏。如果在拾取了扑克牌后，马上扔掉它，不需要太长的时间就要重新洗牌了，因为 7 个玩家玩这个游戏时，就只富余 3 张牌。这样就可以注意 3 张牌何时重新翻开，验证程序是否正常执行。

12.4 小结

本章介绍了事件和事件处理的重要论题。尽管事件是相当微妙的，刚开始很难理解，但它

的代码是非常简单的，读者肯定会在本书的其他地方使用事件处理程序。

除了讨论事件和事件处理的一些简单范例外，我们还给前几章中的 CardLib 类库添加了一些内容。完成后，就使用这个类库创建了一个简单的客户扑克牌游戏程序。这个程序可以作为本书第一部分介绍的所有技术的一个实用范例。

本章不仅完成了 OOP 作为 C#编程技术的讨论，还完成了 C#语言的讨论。从现在开始，我们将在比较复杂的场合中使用这些知识，例如创建 Windows 和 Web 应用程序，更多地利用.NET Framework 中的内容。

12.5 练习

(1) 查看事件处理程序的代码，这些代码使用了一般用途(object sender, EventArgs e)的语法，这些语法也用于本章前面的 Timer.Elapsed 或 Connection.MessageArrived 事件。处理程序应输出一个表示接收了什么类型事件的字符串，并根据引发的事件，输出 MessageArrived EventArgs 参数的 Message 属性或 ElapsedEventArgs 参数的 SignalTime 属性。

(2) 修改扑克牌游戏，设置流行拉米扑克牌的更有趣的取胜条件。即一个玩家要取胜，手中的牌必须包含两套牌，一套由 3 张牌组成，另一套由 4 张牌组成。一套牌应是同花顺(例如 3H, 4H, 5H, 6H)或者几张同点的套牌(例如 2H, 2D, 2S)。

第13章　使用Windows Form控件

近几年来，Visual Basic 在允许程序员使用工具，通过直观的窗体设计器创建高质量的用户界面，编程语言的易学易用，为快速开发应用程序提供尽可能好的环境方面赢得了广泛的好评。Visual Basic 和其他快速应用程序开发工具(如 Delphi)也提供了许多预制控件，开发人员可以使用它们快速建立应用程序的用户界面(UI)。

大多数 Visual Basic Windows 应用程序的核心是窗体设计器。创建用户界面时，把控件从工具箱拖放到窗体上，把它们放在应用程序运行时需要的地方，再双击该控件，添加控件的处理程序。Microsoft 提供的控件和用一定的金钱购买来的定制控件，为程序员提供了空前巨大的重用代码池，以及仅通过鼠标单击就可以完全测试的代码。通过 Visual Studio.NET ，Visual Basic 的这个核心现在也可以用于 C#程序员。

在.NET 出现以前，大多数控件都是专用的 COM 对象，称为 ActiveX 控件。它们常常可以在设计和运行期间显示自己。每个控件都有许多属性，程序员可以利用这些属性完成一定的定制，例如设置背景色、标题和在窗体上的位置。本章介绍的控件与 ActiveX 控件的外观和操作方式相同，但它们不是 ActiveX 控件，而是.NET 装配件。用户仍可以使用为 Visual Studio 旧版本设计的控件，但它们会有较小的性能开销，因为在使用这些控件时，.NET 封装了它们。由于众所周知的原因，在 Microsoft 设计.NET 时，并没有打算放弃现有的控件，所以提供了使用旧控件的方式，即使将来的控件设计为纯粹的.NET 组件，也是这样。

这些.NET 装配件的设计方式允许我们在任何 Visual Studio 语言中使用它们，组件行业肯定会日益增长，并开始生产纯粹的.NET 组件。第 14 章介绍如何创建控件。

注意：

第 21 章深入探讨了.NET 装配件。如果读者想了解装配件，可以参阅第 21 章。

本书前面已经在范例中介绍了窗体设计器，本章将详细介绍它，特别是如何使用各种控件，这些控件都来自于 Visual Studio.NET。在本书中全面介绍 Visual Studio.NET 中的控件是不可能的，所以这里只介绍最常用的控件，包括标签、文本框、列表视图和状态栏等。

13.1　Windows 窗体设计器

首先简要介绍一下 Windows 窗体设计器。在给用户界面布局时，这是一个主要的操作区域。设计窗体也可以不使用 Visual Studio.NET，但在 Notepad 中设计界面需要有丰富的经验。

下面看看我们要使用的环境。启动 Visual Studio.NET，选择 File | New | Project，创建一个新的 C# Windows 应用程序工程。在显示的对话框中，单击左边树图中的 Visual C# Projects，再选择右边列表中的 Windows Application。现在使用 Visual Studio 的默认名称，单击 OK 按钮。

这将打开一个窗口，如图 13-1 所示。

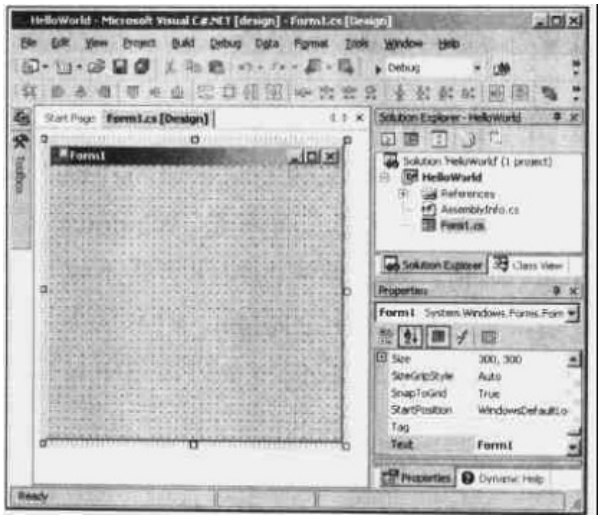


图　13-1

如果读者对 Visual Basic 中的窗体设计器比较熟悉，就会注意到它们的类似之处：显然，设计器非常成功，因此也可以在其他 Visual Studio 语言中使用它。如果读者不熟悉 Visual Basic 设计器，则上述屏幕图中有许多元素，下面将逐个介绍它们。

在屏幕的中心是我们设计的窗体。可以把控件从工具箱拖放到窗体上。图 13-1 中工具箱是折叠的，但如果把鼠标指针放在屏幕左边的 Toolbox 标签上，工具箱就会打开。然后单击面板右上角的大头针图标，把它钉住。这将重新安排工作区域的位置，工具箱现在将总是显示在最上面，不会被窗体遮挡住。后面将详细介绍工具箱及其包含的控件。

在屏幕的左边，Server Explorer 也是折叠的 ——用 Toolbox 标签顶部的计算机图标来表示。可以把它看作是 Windows 控制面板的小版本。在 Server Explorer 中，可以浏览网络上的计算机，添加和删除数据库连接等。

窗口的右边有两个面板。右上方的面板是 Solution Explorer 和 Class View。在 Solution Explorer 中，可以查看所有打开的工程及其相关文件。单击 Solution Explorer 底部的标签，就可以激活 Class Viewer。在 Class Viewer 中，可以浏览工程中的所有类及其所有的派生类。

在屏幕的右下方是 Properties 面板。该面板包含选中项目的所有属性，以便引用和编辑。本章将频繁使用这个面板。

在这个面板上，也可以查看 Dynamic Help 标签页。这个面板显示了选中对象的帮助提示，在键入代码时，还将显示代码的帮助提示。如果计算机使用的是一个旧微处理器，或者 RAM 较小，则最好在不需要该面板时删除它，因为搜索帮助会大大降低性能。

13.2　工具箱

下面详细讨论工具箱。如果屏幕上没有工具箱，可以把鼠标指针移动到屏幕左边的工具箱上，单击面板右上角的大头针图标，把它钉在前景上，如图 13-2 所示。

提示：

如果不小心单击 X，关闭了工具箱，可以选择 View 菜单中的 Toolbox，或按下 Ctrl-Alt-X，重新打开它。



图　13-2

工具箱包含.NET 开发人员可以使用的所有控件集。特别是它提供了对于 Windows Application 开发人员来说非常重要的控件集。如果选择创建一个 Web Forms 工程，而不是 Windows Application，就应使用不同的工具箱。可以使用的控件集并不仅限于此。我们可以定制工具箱，使之满足自己的需要。但本章主要讨论图 13-2 中显示的控件集——实际上，我们将介绍该图中显示的大多数控件。

知道工具箱所在的位置后，就该了解工具箱中的一般性的控件了。

13.3　控件

.NET 中的大多数控件都派生于 System.Windows.Forms.Control 类。这个类定义了控件的基本功能，这就是控件中的许多属性和事件都相同的原因。许多类本身就是其他控件的基类，图 13-3 中的 Label 和 TextBoxBase 类就是这样。

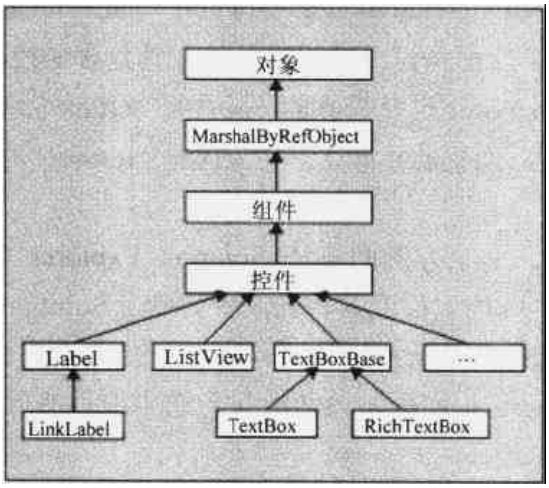


图　13-3

提示：

还有一些控件称为定制或用户控件，派生于另一个类 System.Windows.Forms.UserControl。这个类本身派生于 Control 类，提供了创建控件所需要的功能。第 14 章将介绍它们。顺便说一下，用于设计 Web 用户界面的控件派生于另一个类 System.Web.UI.Control。

13.3.1　属性

所有的控件都有许多属性，用于处理控件的操作。大多数控件的基类 Control 有许多属性，

其他控件要么直接继承了这些属性，要么重写它们，提供某些定制的操作。

表 13-1 列出了 Control 类最常见的属性。这些属性在本章介绍的大多数控件中都有，所以后面将不再详细解释它们，除非属性的操作对于某个控件来说进行了改变。注意这个表并不完整：如果要查看类的所有属性，请参阅 MSDN 库。

表 13-1

名　称	可 用 性	描　述
Anchor	读写	使用这个属性，可以指定当控件的容器大小发生变化时，该控件如何响应。参见下面对这个属性的详细解释
BackColor	读写	控件的背景色
Bottom	读写	设置这个属性，可以指定控件的底部距离窗口的顶部有多远。这与指定控件的高度不同
Dock	读写	可以使控件靠在窗口的边界上。参见下面对这个属性的详细解释
Enabled	读写	把 Enabled 设置为 true 通常表示该控件可以接收用户的输入。把 Enabled 设置为 False 通常表示不能接收用户的输入
ForeColor	读写	控件的前景色
Height	读写	控件从底部到顶部的距离
Left	读写	控件的左边界到窗口左边界的距离
Name	读写	控件的名称。这个名称可以在代码中用于引用该控件
Parent	读写	控件的父控件
Right	读写	控件的右边界到窗口左边界的距离
TabIndex	读写	控件在容器中的标签顺序号
TabStop	读写	指定控件是否可以用 Tab 键访问
Tag	读写	这个值通常不由控件本身使用，而是在控件中存储该控件的信息。当通过 Windows Form 设计器给这个属性赋一个值时，就只能给它赋一个字符串值
Top	读写	控件的顶部距离窗口顶部的距离
Visible	读写	指定控件是否在运行期间可见
Width	读写	控件的宽度

13.3.2　Anchor 和 Dock 属性

在设计窗体时，这两个属性特别有用。如果用户认为改变窗口的大小并不容易，可应确保窗口看起来不显得很乱，并编写许多代码行来达到这个目的。许多程序解决这个问题时，都是禁止给窗口重新设置大小，这显然是解决问题最简单的方法，但不是最好的方法。.NET 引入了 Anchor 和 Dock 属性，就是为了在不编写任何代码的情况下解决这个问题。

Anchor 属性用于指定在用户重新设置窗口的大小时控件该如何响应。可以指定如果控件重新设置了自己的大小，就根据控件自己的边界锁定它，或者其大小不变，但应根据窗口的边界

来锚定它的位置。

Dock 属性与 Anchor 属性是相关的。可以使用该属性指定控件应停放在容器的边框上。如果用户重新设置了窗口的大小，该控件将继续停放在窗口的边框上。例如，如果指定控件停放在容器的底部边界上，则无论窗口的大小如何改变，该控件都将改变大小，确保总是位于屏幕的底部，控件在过程中不会改变大小，只是一直停靠在窗口的边界上。

参见本章后面的文本框范例，了解 Anchor 属性的用法。

13.3.3 事件

在用户单击一个按钮时，应用程序的程序员就会被告知发生了什么。为此，控件使用事件。Control 类定义了本章控件比较常见的许多事件。表 13-2 描述了许多这类事件。这个表仅列出了最常见的事件；如果需要查看完整的列表，请参阅 MSDN 库：

表 13-2

名　称	描　述
Click	在单击控件时引发。在某些情况下，这个事件也会在用户按下回车键时引发
DoubleClick	在双击控件时引发。处理某些控件上的 Click 事件，例如 Button 控件，表示永远不会调用 DoubleClick 事件
DragDrop	在完成拖放操作时引发。换言之，当一个对象被拖到控件上，然后用户释放鼠标按钮后，引发该事件
DragEnter	在被拖动的对象进入控件的边界时引发
DragLeave	在被拖动的对象移出控件的边界时引发
DragOver	在被拖动的对象放在控件上时引发
KeyDown	当控件有焦点时，按下一个键时引发该事件，这个事件总是在 KeyPress 和 KeyUp 之前引发
KeyPress	当控件有焦点时，按下一个键时发生该事件，这个事件总是在 KeyDown 之后引发，在 KeyUp 之前引发。KeyDown 和 KeyPress 的区别是 KeyDown 传送被按下的键的键盘码，而 KeyPress 传送被按下的键的 char 值
GotFocus	在控件接收焦点时引发。不要用这个事件执行控件的有效性验证，而应使用 Validating 和 Validated
LostFocus	在控件失去焦点时引发。不要用这个事件执行控件的有效性验证，而应使用 Validating 和 Validated
MouseDown	在鼠标指针指向一个控件，且鼠标按钮被按下时引发。这与 Click 事件不同，因为在按钮被按下之后，且未被释放之前引发 MouseDown
MouseMove	在鼠标滑过控件时引发
MouseUp	在鼠标指针位于控件上，且鼠标按钮被释放时引发
Paint	绘制控件时引发

(续表)

名　称	描　述
Validated	当控件的 CausesValidation 属性设置为 true，且该控件获得焦点时，引发该事件。它在 Validating 事件之后发生，表示有效性验证已经完成
Validating	当控件的 CausesValidation 属性设置为 true，且该控件获得焦点时，引发该事件。注意被验证有效性的控件是失去焦点的控件，而不是获得焦点的控件

本章的其他部分提供了使用这些事件的许多范例。

下面准备开始讨论控件本身，首先从前面章节中使用过的一个控件开始讨论，即 Button 控件。

13.4　Button 控件

在考虑按钮时，可能会把它想像为一个矩形按钮，单击该按钮，就可以执行某项任务。但在技术上，Visual Studio.NET 中有 3 种按钮。这是因为单选按钮(顾名思义)和复选框也都是按钮。所以，Button 类不是直接派生于 Control 类，而是派生于另一个类 ButtonBase，它派生于 Control 类。本节主要讨论 Button 控件，后面再介绍单选按钮和复选框。

按钮控件存在于几乎所有的 Windows 对话框中。按钮主要用于执行 3 类任务：

- 用某种状态关闭对话框(例如 OK 和 Cancel 按钮)。
- 给对话框上输入的数据执行操作(例如，输入一些搜索条件后，单击 Search)。
- 打开另一个对话框或应用程序(例如 Help 按钮)。

对按钮控件的处理是非常简单的。通常是在窗体上添加控件，再双击它，给 Click 事件添加代码，这对于大多数应用程序来说就足够了。

下面介绍该控件的常用属性和事件，明白该如何操作它。之后，创建一个小范例，举例说明按钮的基本属性和事件。

13.4.1　Button 控件的属性

表 13-3 把属性作为 Buttonbase 的成员列出来，但从技术上讲，它们都是在 ButtonBase 基类中定义的。这里只解释最常用的属性。完整的列表请参阅 MSDN。

表　13-3

名　称	可　用　性	描　述
FlatStyle	读写	按钮的样式可以用这个属性改变。如果把样式设置为 PopUp，则该按钮就显示为平面，直到用户再把鼠标指针移动到它上面为止。此时，按钮会弹出，显示为正常的 3D 外观
Enabled	读写	这个属性派生于 Control，但这里仍讨论它，因为这是一个非常重要的属性。把 Enabled 设置为 false，则该按钮就会灰显，单击它，不会起任何作用
Image	读写	可以指定一个在按钮上显示的图像(位图，图标等)
ImageAlign	读写	使用这个属性，可以设置按钮上的图像在什么地方显示

13.4.2 Button 控件的事件

到目前为止，按钮最常用的事件是 Click。只要用户单击了按钮，即当鼠标指向该按钮时，按下鼠标左键，再释放它，就会引发该事件。如果在按钮上单击了鼠标左键，然后把鼠标移动到其他位置，再释放鼠标，将不会引发 Click 事件。同样，在按钮得到焦点，且用户按下了回车键时，也会引发 Click 事件。如果窗体上有一个按钮，就总是要处理这个事件。

下面看一个范例。创建一个带有 3 个按钮的对话框。其中两个按钮把语言从英语改为丹麦语(也可以使用其他语言)，最后一个按钮关闭对话框。

试试看：按钮的测试

(1) 打开 Visual Studio.NET，创建一个新的 C# Windows 应用程序。把它命名为 ButtonTest。

(2) 钉住工具箱，双击 Button 控件 3 次。然后移动按钮，重新设置窗体的大小，如图 13-4 所示。

(3) 右击一个按钮，选择 Properties，在 Properties 面板上选择 Name 编辑字段和输入文本，修改每个按钮的 Name 属性，如图 13-4 所示。

(4) 与 Name 字段一样，修改每个按钮的 Text 属性，但不修改前 3 个字母(btn)。

(5) 我们要在文本的前面显示一个标志，清晰地表示出每个按钮的作用。选择 English 按钮，找到 Image 属性。单击右边的(...)，打开一个对话框，在该对话框中选择一个图像。我们要显示的标志图标来自于 Visual Studio.NET。如果安装到默认位置(即英语安装)，这些图标将位于 C:\Program Files\Microsoft VisualStudio.NET\Common7\Graphics\icons\ Flags。选择图标 flguk.ico。对 Danish 按钮重复这个操作，但选择 flgden.ico 文件(如果要使用其他的标志，可以从这个目录中选择其他标志)。

(6) 注意按钮文本和图标彼此遮挡，所以需要改变图标的对齐方式。对于 English 和 Danish 按钮，把 ImageAlign 属性改为 MiddleLeft。

(7) 此时，可以调整按钮的宽度，使文本不从图像的右边开头。为此，可以选择每个按钮，向右拖动。

(8) 最后，单击窗体，把 Text 属性改为“Do you speak English?”。

这就是对话框的用户界面，如图 13-5 所示。

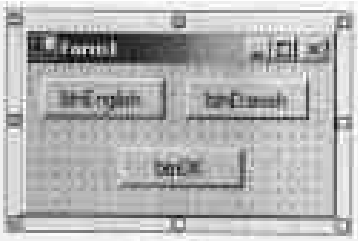


图 13-4

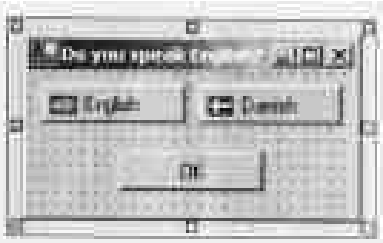


图 13-5

下面准备给对话框添加事件处理程序。双击 English 按钮，以便直接双击处理程序。Click 事件是按钮的默认事件，在双击该按钮时，就会引发该事件。其他控件有其他默认事件。

添加事件处理程序

在双击控件时，窗体的代码编码会发生两个变化。首先，在 InitializeComponent 方法中创建订阅事件的代码：

```
this.btnEnglish.Click += new System.EventHandler(this.btnEnglish_Click);
```

如果要订阅不是默认事件的其他事件，就需要自己编写订阅代码，详见本章的其他内容。记住，每次从设计模式切换到代码模式时，InitializeComponent 方法中的代码都会被覆盖。所以，不要在这个方法中编写事件订阅代码。而应使用类的构造函数。

其次是添加了事件处理程序本身：

```
private void btnEnglish_Click(object sender, System.EventArgs e)
{
    this.Text = "Do you speak English?";
}
```

方法名是把控件名、下划线和要处理的事件名连接在一起。第一个参数 object sender 包含单击的控件。在这个范例中，控件总是由方法名来标识，但在其他情况下，许多控件可能使用同一个方法来处理事件，此时就要通过查看这个值，来确定是哪个控件调用了该方法。本章后面的文本框范例说明了多个控件如何使用同一个方法，另一个参数 System.EventArgs e 包含所发生事件的信息。在本例中，不需要这些信息。

回忆一下本书前面的内容，this 关键字表示类的当前实例。因为我们操作的类由这个实例来表示，所以可以通过该关键字访问它包含的属性和控件。在上面的代码中，设置 this 的 Text 属性，就是设置窗体上当前实例的 Text 属性。

返回窗体设计器，双击 Danish 按钮，编写这个按钮的事件处理程序，下面是代码：

```
private void btnDanish_Click(object sender, System.EventArgs e)
{
    this.Text = "Taler du dansk?";
}
```

这个方法与 btnEnglish_Click 相同，但 Danish 按钮的文本有所不同。最后，以相同的方式添加 OK 按钮的事件处理程序。其代码则有一些不同之处：

```
private void btnOK_Click(object sender, System.EventArgs e)
{
    Application.Exit();
}
```

使用这段代码，就可以退出应用程序。这就是第一个范例。编译这个范例，运行它，单击按钮，将得到如图 13-6 所示的结果。



图　13-6

13.5 Label 和 LinkLabel 控件

Label 控件也许是最常用的控件。在任何 Windows 应用程序中，都可以在对话框中见到它们。标签是一个简单的控件，其用途只有一个：显示标题或简短提示，给用户解释窗体的某些内容。

Visual Studio.NET 包含两个标签控件，它们可以用两种截然不同的方式来显示它们自己：

- Label 是标准的 Windows 标签。
- LinkLabel 类似于标准标签(派生于标准标签)，但以 Internet 链接的方式显示(超链接)。

这两个控件都在 Window Forms 标签页的控件面板顶部。在图 13-7 中，拖放了这两种类型的标签，以说明它们之间在外观上的区别。

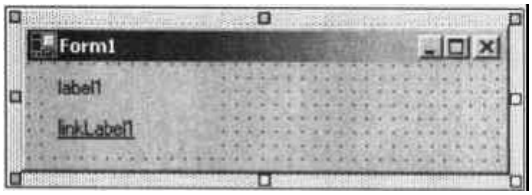


图　13-7

注意：

如果读者使用过 Visual Basic，就会注意到 Text 属性用于设置显示的文本，而不是 Caption 属性。.NET 的所有内置控件都使用 Text 来表示控件的主要文本。在.NET 出现之前，Caption 和 Text 是可以互换使用的。

这就是 Label 控件的主要用途。标准的 Label 通常不需要添加任何事件处理代码。但如果使用 LinkLabel 控件，希望用户可以单击它，进入文本中显示的网页，就需要添加其他代码。

Label 控件有非常多的可以设置的属性。大多数属性都派生于 Control，但有一些属性是新增的。表 13-4 列出了最常见的属性。如果没有特别说明，这些属性存在于 Label 和 LinkLabel 控件中。

表　13-4

名　称	可 用 性	描　述
BorderStyle	读写	可以指定标签边框的样式。默认为无边框
DisabledLinkColor	读写	(只用于 LinkLabel)用户单击 LinkLabel 后控件的颜色
FlatStyle	读写	控制显示控件的方式。把这个属性设置为 PopUp，表示控件一直显示为平面样式，直到用户把鼠标指针移动到该控件上面，此时，控件显示为弹起样式
Image	读写	这个属性可以指定要在标签上显示的图像(位图，图标等)
ImageAlign	读写	(读写) 图像显示在标签的什么位置
LinkArea	读写	(只用于 LinkLabel)文本中显示为链接的部分
LinkColor	读写	(只用于 LinkLabel) 链接的颜色

(续表)

名　称	可 用 性	描　述
Links	只读	(只用于 LinkLabel) LinkLabel 可以包含多个链接。利用这个属性可以查找需要的链接。控件会跟踪显示在文本中的链接
LinkVisited	只读	(只用于 LinkLabel) 返回链接是否可访问
Text	读写	显示在 Label 上的文本
TextAlign	读写	文本显示在控件的什么地方

13.6　TextBox 控件

在希望用户输入程序员在设计阶段不知道的文本(例如用户的姓名)时，应使用文本框。文本框的主要用途是让用户输入文本，用户可以输入任何字符，也可以限制用户只输入数值。

.NET 内置了两个基本控件来提取用户输入的文本：TextBox 和 RichTextBox(本章后面讨论 RichTextBox)。这两个控件都派生于基类 TextBoxBase，而 TextBoxBase 派生于 Control。

TextBoxBase 提供了在文本框中处理文本的基本功能，例如选择文本、剪切和从剪切板上粘贴，以及许多事件。这里不讨论什么对象派生于什么类，而是先介绍两个控件中比较简单的一个：TextBox。下面创建一个范例，说明 TextBox 的属性，并在此基础上说明后面的 RichTextBox 控件。

13.6.1　TextBox 控件的属性

如本章前面所述，列出所有的属性是不可能的，所以这里仅列出最常见的属性，如表 13-5 所示。

表　13-5

名　称	可 用 性	描　述
CausesValidation	读写	当控件的属性设置为 true，且该控件获得了焦点时，会引发两个事件：Validating 和 Validated。可以处理这些事件，以便验证失去焦点的控件中数据的有效性。这可能使控件永远都不能获得焦点。下面会讨论相关的事件
CharacterCasing	读写	这个值表示 TextBox 是否会改变输入的文本的大小写。可能的值有： • Lower: 文本框中输入的所有文本都转换为小写 • Normal: 不对文本进行任何转换 • Upper: 文本框中输入的所有文本都转换为大写
MaxLength	读写	这个值指定输入到 TextBox 中的文本最多的字符长度。把这个值设置为 0，表示最大字符长度仅受限于可用的内存

(续表)

名　称	可 用 性	描　述
Multiline	读写	表示该控件是否是一个多行控件。多行控件可以显示多行文本
PasswordChar	读写	指定是否用密码字符替换在单行文本框中输入的字符。如果 Multiline 属性为 true，这个属性就不起作用
ReadOnly	读写	这个 Boolean 值表示文本是否为只读
ScrollBars	读写	指定多行文本框是否显示滚动条
SelectedText	读写	在文本框中选择的文本
SelectionLength	读写	在文本中选择的字符数。如果这个值设置得比文本中总字符数大，则控件会把它重新设置为字符总数减去 SelectionStart 的值
SelectionStart	读写	文本框中被选中文本的开头
WordWrap	读写	指定在多行文本框中，如果一行的宽度超出了控件的宽度，其文本是否应自动换行

13.6.2 TextBox 控件的事件

在窗体上，对 TextBox 控件中文本的有效性验证会使一些用户很高兴，使另一些用户很生气。

当用户单击了 OK 按钮后，对话框只验证其内容，此时用户会非常生气。这种验证数据有效性的方式通常会显示一个信息框，告诉用户“第三个 TextBox”中的数据不正确。接着继续单击 OK 按钮，直到所有的数据都正确为止。显然这不是验证数据有效性的好方法，那么我们还能怎么做呢？

答案取决于 TextBox 控件提供的有效性验证事件。如果要确保文本框中不输入无效的字符，或者只输入某个范围内的数值，就需要告诉控件的用户：输入的值是否有效。

TextBox 控件提供了表 13-6 所示的事件(所有的事件都派生于 Control)。

表 13-6

名　称	描　述
Enter GotFocus Leave Validating Validated LostFocus	这 6 个事件按照列出的顺序引发。它们统称为“焦点事件”，当控件的焦点发生改变时引发，但有两个例外。Validating 和 Validated 仅在控件接收了焦点，且其 CausesValidation 属性设置为 true 时引发。接收焦点的控件引发事件的原因是有时即使焦点改变了，我们也不希望验证控件的有效性。它的一个范例是用户是否单击了 Help 按钮

（续表）

名　称	描　述
KeyDown KeyPress KeyUp	这 3 个事件称为“键事件”。它们可以监视和改变输入到控件中的内容 KeyDown 和 KeyUp 接收与按下键对应的键码，这样就可以确定是否按下了特殊的键 Shift 或 Control 和 F1 另一方面，KeyPress 接收与键对应的字符。这表示字母 a 的值与字母 A 的值不同 如果要排除某个范围内的字符，例如只允许输入数值，这是很有用的
Change	只要文本框中的文本发生了改变，无论发生什么改变，都会引发该事件

试试看：TextBoxTest

下面创建一个对话框，在该对话框中可以输入姓名、地址、职业和年龄。这个范例的目的是为处理属性和使用事件打下基础，而不是创建什么特别有用的东西。

下面先建立用户界面：

(1) 选择 Files | New，创建一个新的 C# Windows Application 工程。把该工程命名为 TextBoxTest。

(2) 创建图 13-8 所示的窗体，把标签、文本框和按钮拖放到设计界面上。在重新设置两个文本框 txtAddress 和 txtOutput 的大小时，必须把它们的 Multiline 属性设置为 true。为此，右击控件，选择 Properties。

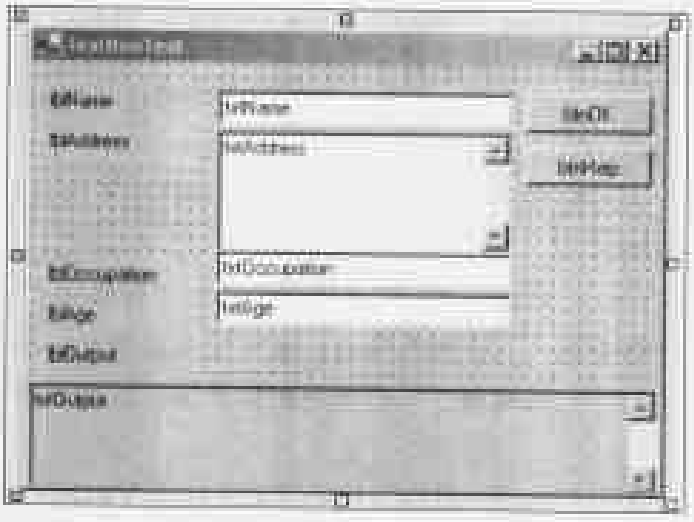


图 13-8

(3) 给控件命名，如图 13-8 所示。

(4) 把每个文本框的 Text 属性设置为空字符串，这表示在应用程序第一次运行时它们是空的。

(5) 把其他控件的 Text 属性设置为控件的名称，但不改变前 3 个字符。把窗体的 Text 属性设置为图中的标题。

(6) 把两个控件 txtOutput 和 txtAddress 的 Scrollbars 属性设置为 Vertical。

(7) 把 txtOutput 控件的 ReadOnly 属性设置为 true。

(8) 把按钮 btnHelp 的 CausesValidation 属性设置为 false。在前面讨论 Validating 和 Validated 事件时，把这个属性设置为 false，就可以让用户单击这个按钮，而不必考虑输入的无效数据。

(9) 改变窗体的大小，使之适合于控件的大小，然后就锚定控件，这样它们就可以在重新设置窗体的大小时正确地响应。设置 Anchor 属性，如表 13-7 所示。

表 13-7

控　件	Anchor 值
所有的标签	Top, Bottom, Left
所有的文本框	Top, Bottom, Left, Right
两个按钮	Top, Right

注意：

可以从一个控件中把 Anchor 属性文本复制并粘贴到值相同的其他控件上。用户不需要使用下拉按钮。

锚定 txtOutput 而不是让它停放在窗体底部的原因是，在拖动窗体时，要重新设置输出文本区域的大小。如果要把控件停放在窗体的底部，就要把它移动到底部，但不会重新设置大小。

(10) 最后要设置的是，在窗体上，找到 Size 和 MinSize 属性。如果窗体设置得比现在的小，就没有什么意义了，因此应把 MinSize 属性值设置得与 Size 属性值一样大。

设置窗体的可见部分现在已经完成了。如果运行它，则单击按钮或输入文本，将不会发生什么。但如果最大化或拖动对话框，控件就会按照希望的那样在用户界面上位于正确的位置，并重新设置其大小，以填充整个对话框。如果用户以前曾经用过其他语言例如 Visual Basic 6 来完成相同的任务，就知道刚才的任务量减少了多少。

下面该看看代码了。右击窗体，选择 View Code。如果钉住了工具箱，就应去掉该大头钉，给代码窗口留出更多的空间。

编辑器中的代码非常少。在类的顶部，定义了控件，但只有展开了 Windows Form Designer Generated Code 区域，才能看到代码在什么地方。这里要强调的是，本节不应编辑任何代码。下次改变设计器中的内容时，它会被覆盖，甚至更糟糕的是，改变了内容后，可能会使窗体设计器不再显示窗体。

再花点时间复习一下本节的内容，明白为什么可以不使用 Visual Studio.NET 创建 Windows 应用程序。本节的内容都可以在 Notepad 或类似的文本编辑器中输入和编译。用户还应明白为什么不推荐这样做。在最有利的情况下跟踪其中的内容也是很困难的，而很容易引入错误，而且因为不能看到这些代码的效果，把控件放在窗体上的合适位置也是一件烦琐的任务。但这为第三方软件厂商编写自己的编程环境，以对抗 Visual Studio.NET 打开了大门，因为用于创建窗体的编译器包含在.NET Framework 中，而不是在 Visual Studio.NET 中。

添加事件处理程序

在添加代码前，可以查看一下在窗体设计器中最后做的工作。返回窗体设计器(单击文本编

辑器顶部的标签)，双击按钮 btnOK，对其他按钮重复这个过程。与本章前面的按钮范例一样，这会创建按钮的 Click 事件处理程序。单击 OK 按钮后，我们要把输入文本框中的文本传送到只读的输出框中。

下面是两个 Click 事件的代码：

```
private void btnOK_Click(object sender, System.EventArgs e)
{
    // No testing for invalid values are made, as that should
    // not be necessary

    string output;

    // Concatenate the text values of the four TextBoxes
    output = "Name: " + this.txtName.Text + "\r\n";
    output += "Address: " + this.txtAddress.Text + "\r\n";
    output += "Occupation: " + this.txtOccupation.Text + "\r\n";
    output += "Age: " + this.txtAge.Text;

    // Insert the new text
    this.txtOutput.Text = output;
}

private void btnHelp_Click(object sender, System.EventArgs e)
{
    // Write a short description of each TextBox in the Output TextBox
    string output;

    output = "Name = Your name\r\n;
    output += "Address = Your address\r\n";
    output += "Occupation = Only allowed value is 'Programmer'\r\n";
    output += "Age = Your age";

    // Insert the new text
    this.txtOutput.Text = output;
}
```

在这两个函数中，使用了文本框的 Text 属性，在 btnOK_Click()函数中获取或设置该属性，或在 btnHelp_Click()函数中设置了该属性。

我们插入用户输入的信息，无需检查该信息是否正确。这就是说，必须在其他地方进行检查。在本例中，必须满足许多条件，其值才是正确的。

- 用户名不能为空。
- 用户的年龄必须是一个大于或等于 0 的数字。
- 用户的职业必须是“程序员”或为空。
- 用户的地址不能为空。

从中可以看出，对两个文本框(txtName 和 txtAddress)进行的检查是相同的。并应禁止用户在 Age 框中输入无效的数值，最后必须检查用户是否是程序员。

为了防止用户在输入完信息之前单击 OK 按钮，要先在窗体的构造函数中，把 OK 按钮的 Enabled 属性设置为 false，在调用 InitializeComponent()中生成的代码之后，才能设置这个属性：

```
public Form1()
{
    //
    // Required for Windows Form Designer support
    //
    InitializeComponent();
    this.btnOK.Enabled = false;
}
```

下面创建这两个文本框的处理程序，检查一下它们是否为空。为此，订阅文本框的 Validating 事件。通知控件该事件应由方法 txtBoxEmpty_Validating()来处理。

还需要知道控件的状态。为此，使用文本框的 Tag 属性。回忆一下本章前面关于这个属性的讨论，在窗体设计器中的 Tag 属性的赋值只能是字符串值。但由于 Tag 属性是在代码中赋值，所以可以对它进行其他处理，这里应该输入一个布尔值。

在构造函数中，添加下述语句：

```
this.btnOK.Enabled = false;

// Tag values for testing if the data is valid
this.txtAddress.Tag = false;
this.txtAge.Tag = false;
this.txtName.Tag = false;
this.txtOccupation.Tag = false;

// Subscriptions to events
this.txtName.Validating += new
System.ComponentModel.CancelEventHandler(this.txtBoxEmpty_Validating);
this.txtAddress.Validating += new
        System.ComponentModel.CancelEventHandler(this.txtBoxEmpty_Validating);
```

注意：

如果对事件还没有完全理解，请参阅第 12 章中对事件的完整论述。

与前面的按钮事件处理程序不同，Validating 事件的处理程序是标准处理程序 System.EventHandler 的一个专业化版本。这个事件需要专业化处理程序的原因是，如果有效性验证失败，就必须有一种方式防止进行任何进一步的处理。如果要取消进一步的处理，就表示在输入有效的数据前，不能退出该文本框。在这个范例中不需要进行任何操作。

提示：

在使用 GotFocus 和 LostFocus 事件执行控件的有效性验证时，Validating 和 Validated 事件和

CausesValidation 属性一起导致了错误的产生。当 GotFocus 和 LostFocus 事件被连续引发时，就产生了这个错误。因为有效性验证代码试图在控件之间移动焦点，这将产生一个无限循环。

添加下面的事件处理程序：

```
private void txtBoxEmpty_Validating(object sender,
                                    System.ComponentModel.CancelEventArgs e)
{
    // We know the sender is a TextBox, so we cast the sender object to that
    TextBox tb = (TextBox)sender;

    // If the text is empty we set the background color of the
    // Textbox to red to indicate a problem. We use the tag value
    // of the control to indicate if the control contains valid
    // information.
    if (tb.Text.Length == 0)
    {
        tb.BackColor = Color.Red;
        tb.Tag = false;

        // In this case we do not want to cancel further processing,
        // but if we had wanted to do this, we would have added this line:
        // e.Cancel = true;
    }
    else
    {
        tb.BackColor = System.Drawing.SystemColors.Window;
        tb.Tag = true;
    }

    // Finally, we call ValidateAll which will set the value of
    // the OK button.
    ValidateAll();
}
```

因为有多个文本框使用这个方法来处理事件，所以我们不知道哪个控件调用了函数，但知道无论是哪个控件调用了方法，其结果是一样的，所以可以把发送对象的参数传送给文本框，对它进行操作即可。

如果文本框中的文本长度是 0，就把背景色设置为红色，把 Tag 设置为 false。如果不是，就把背景色设置为窗口的标准 Windows 颜色。

注意：

在设置控件的标准颜色中，总是应使用 System.Drawing.SystemColors 枚举中的颜色。如果把颜色设置为白色，而用户修改了默认的颜色设置，应用程序看起来就会很古怪。

ValidateAll()函数在本例的最后介绍。

与 Validating 事件相对照，下一个添加的处理程序是 Occupation 文本框的处理程序。这个过程与前面两个处理程序完全相同，但有效性验证代码有所不同，因为职业必须是程序员或一个空字符串。所以要在构造函数中添加一行代码：

```
this.txtOccupation.Validating += new
    System.ComponentModel.CancelEventHandler(this.txtOccupation_Validating);
```

然后是处理程序本身：

```
private void txtOccupation_Validating(object sender,
                                      System.ComponentModel.CancelEventArgs e)
{
    // Cast the sender object to a textbox
    TextBox tb = (TextBox)sender;

    // Check if the values are correct
    if (tb.Text.CompareTo("Programmer") == 0 || tb.Text.Length == 0)
    {
        tb.Tag = true;
        tb.BackColor = System.Drawing.SystemColors.Window;
    }
    else
    {
        tb.Tag = false;
        tb.BackColor = Color.Red;
    }

    // Set the state of the OK button
    ValidateAll();
}
```

倒数第二个要做的操作是处理年龄文本框。我们希望用户只输入正数(包括 0，这样可以使检测简单一些)。为此，使用 KeyPress 事件，在不想要的字符在文本框中显示出来之前就删除它们。

首先，订阅 KeyPress 事件。其方式与构造函数的其他事件处理程序一样：

```
this.txtAge.KeyPress += new
        System.Windows.Forms.KeyPressEventHandler(this.txtAge_KeyPress);
```

这个事件处理程序也是专业化的。其中提供了 System.Windows.Forms.KeyPressEventHandler，因为事件需要按下键的信息。

然后添加事件处理程序本身：

```
private void txtAge_KeyPress(object sender,
                             System.Windows.Forms.KeyPressEventArgs e)
{
    if ((e.KeyChar < 48 || e.KeyChar > 57) && e.KeyChar != 8)
```

```
        e.Handled = true; // Remove the character
    }
```

0 到 9 之间数字的 ASCII 值是 48 到 57，所以应保证字符在这个范围内。但有一个例外。ASCII 值 8 表示退格键，为了编辑方便，允许跳过它。

把 KeyPressEventArgs 的 Handled 属性设置为 true，告诉控件不应对字符进行任何操作，所以不显示该字符。

现在控件没有标记为有效或无效，这是因为需要进行另一个检查，看看是否输入了所有的信息。这是很简单的，因为前面已经编写了执行该检查的方法，并在构造函数中添加如下一行代码，订阅 Age 控件的 Validating 事件：

```
this.txtAge.Validating += new
        System.ComponentModel.CancelEventHandler(this.txtBoxEmpty_Validating);
```

最后必须对所有的控件进行处理。如果用户在所有的文本框中输入了有效的文本，接着修改了某些文本，使它们不再有效，则 OK 按钮仍是可用的。所以必须为所有的文本框编写最后一个事件处理程序：Change 事件，如果任何文本字段包含无效的数据，它就禁用 OK 按钮。

当控件中的文本发生了改变，就激发 Change 事件。在构造函数中添加如下一行代码，订阅该事件：

```
this.txtName.TextChanged += new System.EventHandler(this.txtBox_TextChanged);
this.txtAddress.TextChanged += new
                                    System.EventHandler(this.txtBox_TextChanged);
this.txtAge.TextChanged += new System.EventHandler(this.txtBox_TextChanged);
this.txtOccupation.TextChanged += new
                                    System.EventHandler(this.txtBox_TextChanged);
```

Change 事件使用 Click 事件的标准事件处理程序。最后，添加该事件本身：

```
private void txtBox_TextChanged(object sender, System.EventArgs e)
{
    // Cast the sender object to a Textbox
    TextBox tb = (TextBox)sender;

    // Test if the data is valid and set the tag and background
    // color accordingly.
    if (tb.Text.Length == 0 && tb != txtOccupation)
    {
        tb.Tag = false;
        tb.BackColor = Color.Red;
    }
    else if (tb == txtOccupation &&
            (tb.Text.Length != 0 && tb.Text.CompareTo("Programmer") != 0))
    {
        // Don't set the color here, as it will color change while the user
        // is typing
```

```
            tb.Tag = false;
        }
        else
        {
            tb.Tag = true;
            tb.BackColor = SystemColors.Window;
        }

        // Call ValidateAll to set the OK button
        ValidateAll();
    }
```

这次，必须知道哪个控件调用了事件处理程序，因为在用户开始输入时，我们不希望把 Occupation 文本框的背景色改为红色。为此，应检查文本框的 Name 属性，该属性放在发送对象参数中。

最后一件事：激活或禁用 OK 按钮的 ValidateAll 方法：

```
    private void ValidateAll()
    {
        // Set the OK button to enabled if all the Tags are true
        this.btnOK.Enabled = ((bool)(this.txtAddress.Tag) &&
                              (bool)(this.txtAge.Tag) &&
                              (bool)(this.txtName.Tag) &&
                              (bool)(this.txtOccupation.Tag));
    }
```

如果所有的 Tag 属性都是 true，这个方法就把 OK 按钮的 Enabled 属性设置为 true。需要把 Tag 属性的值转换为布尔值，因为它存储为一个对象类型。

如果现在测试该程序，就会得到如图 13-9 所示的结果。

注意，在文本框中输入了无效的数据，但背景色没有改为红色时，可以单击 Help 按钮。

与本章前面的范例相比，刚完成的范例比较长，这是因为我们将以这个范例为基础。

注意：

记住可以从 www.wrox.com 上下载本书的源代码。

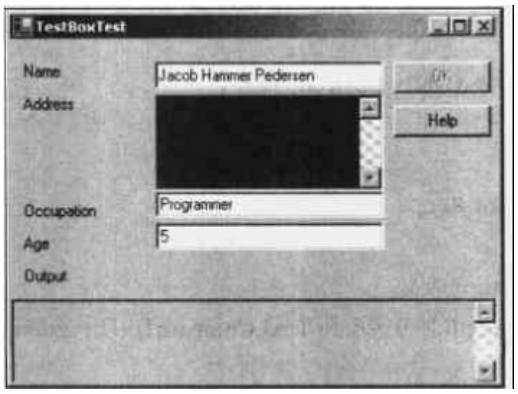


图　13-9

13.7　RadioButton 和 CheckBox 控件

如前所述，RadioButton 和 CheckBox 控件与按钮控件有相同的基类，但它们的外观和用法大不相同。

传统上，单选按钮显示为一个标签，左边是一个圆点，该点可以是选中或未选中。在要给用户提供几个互斥选项时，就可以使用单选按钮。例如，询问用户的性别。

把单选按钮组合在一起，使它们创建一个逻辑单元，此时必须使用 GroupBox 控件。首先在窗体上拖放一个组框，再把需要的 RadioButton 按钮放在组框的边界之内，RadioButton 按钮知道如何改变自己的状态，以反映组框中惟一被选中的选项。如果不把它们放在组框中，则在任意时刻，窗体上只有一个 RadioButton 被选中。

传统上，CheckBox 显示为一个标签，左边是一个带有标记的小方框。在希望用户可以选择一个或多个选项时，就应使用复选框。例如询问用户要使用的操作系统(如 Windows 95, Windows 98, Linux 和 Max OS X 等)。

下面介绍这两个控件的重要属性和事件，从 RadioButton 开始，然后用一个小范例说明它们的用法。

13.7.1　RadioButton 控件的属性

这个控件派生于 ButtonBase，前面已经有一个使用按钮的范例了，所以要描述的属性只有几个，如表 13-8 所示。与往常一样，如果用户要查看完整的列表，请参阅 MSDN 库。

表　13-8

名　称	可 用 性	描　述
Appearance	读写	RadioButton 可以显示为一个圆形选中标签，放在左边、中间或右边，或者显示为标准按钮。当它显示为按钮时，控件被选中时显示为按下状态，否则显示为 3D 外观
AutoCheck	读写	如果这个属性为 true，用户单击单选按钮时，会显示一个选中标记。如果该属性为 false，默认状态下不显示选中标记
CheckAlign	读写	使用这个属性，可以改变单选按钮的对齐形式，它可以是左对齐、居中和右对齐
Checked	读写	表示控件的状态。如果控件有一个选中标记，它就是 true，否则为 false

13.7.2　RadioButton 控件的事件

在处理 RadioButtons 时，通常只使用一个事件，但与往常一样，还可以订阅许多其他事件。本章只介绍两个事件，介绍第二个事件的原因是它们之间有微妙的区别，如表 13-9 所示。

表 13-9

名　称	描　述
CheckChanged	当 RadioButton 的选中改变时，引发这个事件。如果窗体或组框中有多个 RadioButton 控件，这个事件会引发两次。第一次是为刚才选中、但现在变成未选中的控件而引发，第二次是为现在变成选中的控件而引发
Click	每次单击 RadioButton 时，都会引发该事件。这与 Change 事件是不同的，因为连续单击 RadioButton 两次或多次只改变 Checked 属性一次，且只改变以前未选中的控件的 Checked 属性

13.7.3 CheckBox 控件的属性

可以想像，这个控件的属性和事件非常类似于 RadioButton 控件的属性和事件，但有两个新属性，如表 13-10 所示。

表 13-10

名　称	可 用 性	描　述
CheckState	读写	与 RadioButton 不同，CheckBox 有 3 种状态： Checked、Indeterminate 和 Unchecked。复选框的状态是 Indeterminate 时，控件旁边的复选框通常是灰显的，表示复选框的当前值是无效的，或者在当前环境下没有意义。例如，如果在 Windows Explorer 中选择几个文件，查看它们的属性。如果一些文件是只读的，而另一些文件不是，就选中只读复选框，但它们是灰显的– indeterminate
ThreeState	读写	这个属性为 false 时，用户就不能把复选框的状态改为 Indeterminate。但仍可以在代码中把复选框的状态改为 Indeterminate

13.7.4 CheckBox 控件的事件

一般只使用这个控件的一两个事件。注意，即使 RadioButton 和 CheckBox 控件都有 CheckChanged 事件，但其结果是不同的，如表 13-11 所示。

表 13-11

名　称	描　述
CheckedChanged	当复选框的 Checked 属性改变时，就引发该事件。注意在复选框中，当 ThreeState 为 true 时，单击复选框不会改变 Checked 属性。在复选框从 Checked 变为 indeterminate 状态时，就会出现这种情况
CheckedStateChanged	当 CheckedState Checked 属性改变时，引发该事件。CheckedState 属性的值可以是 Checked 和 Unchecked。只要 Checked 属性改变了，就引发该事件。另外，当状态从 Checked 变为 indeterminate 时，也会引发该事件

前面总结了 RadioButton 和 CheckBox 控件的事件和属性。在使用它们之前，先介绍一下前面提及的 GroupBox 控件。

13.7.5　GroupBox 控件

在介绍范例之前，先看看组框控件。这个控件常常和 RadioButton 及 CheckBox 控件一起使用，显示一个框架，其上有一个标题，其中的一系列控件以某种方式逻辑链接起来。

组框的用法非常简单，把它拖放到窗体上，再把所需的控件拖放到组框中即可(但其他方式不能用，不能使组框与一些已有的控件重叠)。其结果是父控件是组框，而不是窗体，所以在任意时刻，可以选择多个 RadioButton。但在组框中，只能选择一个 RadioButton。

这里需要解释一下父控件和子控件的关系。把一个控件放在窗体上时，窗体就是该控件的父控件，所以该控件是窗体的一个子控件。而把一个 GroupBox 放在窗体上时，它就成为窗体的一个子控件。而组框本身可以包含控件，所以它就是这些控件的父控件，其结果是移动 GroupBox 时，其中的所有控件也会移动。

把控件放在组框上的另一个结果是可以改变其中所有控件的某些属性，方法是在组框上设置这些数据。例如，如果要禁用组框中的所有控件，只需把组框的 Enabled 属性设置为 false 即可。

下面用一个范例说明 GroupBox 的用法。

试试看：RadioButton 和 CheckBox 范例

下面修改用于说明文本框用法的范例。在该范例中，惟一可能的职业是程序员。下面不强迫用户填写程序员，而是把这个文本框改成复选框。

为了说明 RadioButton 的用法，我们将要求用户再提供一条信息：性别。

把文本框范例改为：

(1) 删除 blOccupation 标签和文本框 txtOccupation。

(2) 重新设置窗体的大小，使之能放下一个组框，其中包含用户的性别信息，并命名这些新控件，如图 13-10 所示。

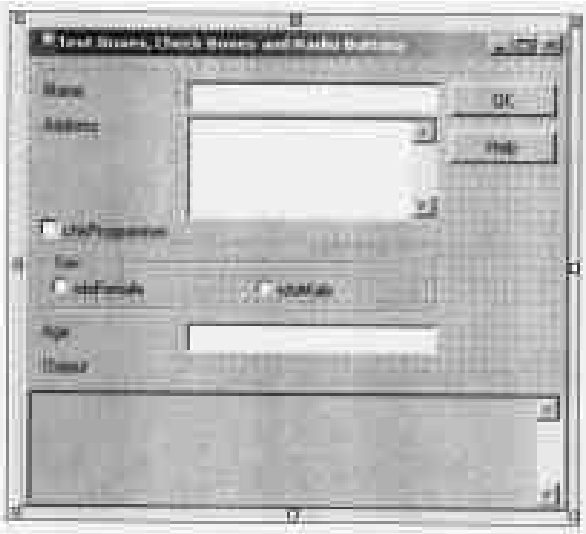


图 13-10

(3) RadioButton 和 CheckBox 控件的文本应与该控件相同(前 3 个字符不算)。

(4) 把 chkProgrammer 复选框的 Checked 属性设置为 true。

(5) 把 rdoMale 或 rdoFemale 的 Checked 属性设置为 true。注意不能把它们两个同时设置为 true。否则，另一个 RadioButton 的值会自动变为 false。

对这个范例的可见部分不再需要更多的修改，但代码要进行许多修改。首先，需要删除所有对已删除文本框的引用。进入代码，完成下述步骤。

(6) 在窗体的构造函数中，删除引用 txtOccupation 的两行代码，这包括对 Validating 和 TextChanged 事件的订阅，以及把 txtBox 的 Tag 属性设置为 false 的代码行。

(7) 彻底删除方法 txtOccupation_Validating。

提示：

这个范例在本章下载的 RadioButtonAndCheckBox Visual Studio.NET 工程的代码中。

添加事件处理程序

txtBox_TextChanged 方法可以进行测试，看看调用的控件是否为 txtOccupation 文本框。我们现在知道该控件并不是 txt Occupation 文本框，所以要修改该方法，即删除 else if 块，修改 if 测试代码，如下所示：

```
private void txtBox_TextChanged(object sender, System.EventArgs e)
{
    // Cast the sender object to a Textbox
    TextBox tb = (TextBox)sender;

    // Test if the data is valid and set the tag background
    // color accordingly.
    if (tb.Text.Length == 0)
    {
        tb.Tag = false;
        tb.BackColor = Color.Red;
    }
    else
    {
        tb.Tag = true;
        tb.BackColor = SystemColors.Window;
    }

    // Call ValidateAll to set the OK button
    ValidateAll();
}
```

最后在 ValidateAll()方法中检查已删除的文本框的值，彻底删除检查代码，则最终代码变成：

```
private void ValidateAll()
```

```
{
    // Set the OK button to enabled if all the Tags are true
    this.btnOK.Enabled = ((bool)(this.txtAddress.Tag) &&
        (bool)(this.txtAge.Tag) &&
        (bool)(this.txtName.Tag));
}
```

这里使用的是复选框，而不是文本框，所以用户不会输入无效的信息，因为用户要么是一个程序员，要么不是。

我们也知道用户要么是男性，要么是女性，因为前面把一个 RadioButton 的属性设置为 true，这样用户就不会选择无效的值。因此，下面只需要修改帮助文本和输出。我们在按钮事件处理程序中完成它：

```
private void btnHelp_Click(object sender, System.EventArgs e)
{
    // Write a short description of each TextBox in the Output TextBox
    string output;

    output = "Name = Your name\r\n";
    output += "Address = Your address\r\n";
    output += "Programmer = Check 'Programmer' if you are a programmer\r\n";
    output += "Sex = Choose your sex\r\n";
    output += "Age = Your age";

    // Insert the new text
    this.txtOutput.Text = output;
}
```

下面只剩下修改帮助文本了，所以不要对帮助方法感到惊讶。在 OK 方法中，它显得稍微有趣一点：

```
private void btnOK_Click(object sender, System.EventArgs e)
{
    // No testing for invalid values are made, as that should
    // not be neccessary

    string output;

    // Concatenate the text values of the four TextBoxes
    output = "Name: " + this.txtName.Text + "\r\n";
    output += "Address: " + this.txtAddress.Text + "\r\n";
    output += "Occupation: " + (string)(this.chkProgrammer.Checked ?
            "Programmer" : "Not a programmer") + "\r\n";
    output += "Sex: " + (string)(this.rdoFemale.Checked ? "Female" :
                                                    "Male") + "\r\n";
```

```
        output += "Age: " + this.txtAge.Text;

        // Insert the new text
        this.txtOutput.Text = output;
    }
```

新增代码进行了突出显示。其中的第一行打印出了用户的职业。考察一下复选框的 Checked 属性，如果它是 true，就写入字符串“Programmer”，如果它是 false，就填写 Not a programmer。

第二行代码检查单选按钮 rdoFemale。如果该控件的 Checked 属性是 true，则该用户是一位女性。如果它是 false，则该用户是一位男性。因为这里只有两个选项，所以不需要选中其他单选按钮，该按钮的 Checked 属性总是与第一个单选按钮相反。

如果使用多于两个的单选按钮，就必须循环所有的按钮，直到找到一个 Checked 属性是 true 的按钮为止。

现在运行范例，得到如图 13-11 所示的结果。

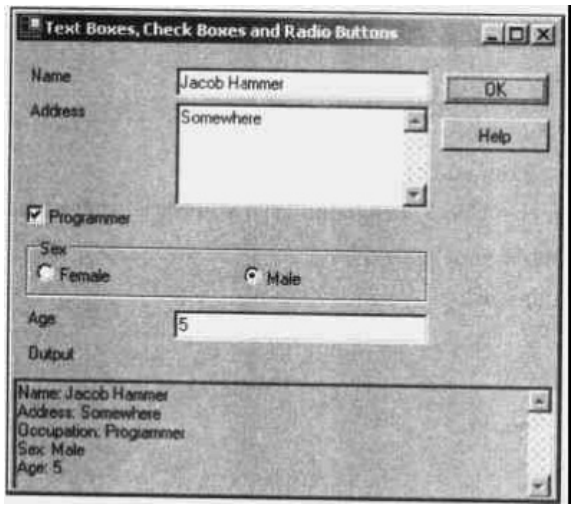


图　13-11

13.8　RichTextBox 控件

与一般的 TextBox 一样，RichTextBox 控件派生于 TextBoxBase。所以，它与 TextBox 共享许多功能。但许多功能是不同的。TextBox 常用于从用户处获取短文本字符串，而 RichTextBox 用于显示和输入格式化的文本(例如，黑体、下划线和斜体)。它使用标准的格式化文本，称为 Rich Text Format (富文本格式)或 RTF。

在上面的范例中，我们使用了标准的 TextBox。也可以使用 RichTextBox 来完成该任务。实际上，如后面的范例所示，可以删除 txtOutput 文本框，在它的位置上插入一个同名的 RichTextBox，这个范例还会像以前那样运行。

13.8.1　RichTextBox 控件的属性

如果这种文本框比上一节介绍的文本框更高级，我们就会期望它有一些新属性。表 13-12 列出了 RichTextBox 的常用属性。

表　13-12

名　称	可 用 性	描　述
CanRedo	只读	如果某个任务还没有完成，这个属性就是 true，否则就是 false。它可以重复应用
CanUndo	只读	如果可以在 RichTextBox 上执行撤销操作，这个属性就是 true，否则就是 false
RedoActionName	只读	这个属性包含操作名称，该操作可用于重做已经在 RichTextBox 中撤销的动作
DetectUrls	读写	把这个属性设置为 true，可以使控件检测 URL，并格式化它们(在浏览器中是带有下划线的部分)
Rtf	读写	它对应于 Text 属性，但包含 RTF 格式的文本
SelectedRtf	读写	使用这个属性可以获取或设置控件中被选中的 RTF 格式的文本如果把这些文本复制到另一个应用程序中，例如 MS Word ，该文本会保存所有的格式化信息
SelectedText	读写	与 SelectedRtf 一样，可以使用这个属性获取或设置被选中的文本。但与该属性的 RTF 版本不同，所有的格式化信息都会丢失
SelectionAlignment	读写	它表示选中文本的对齐方式，可以是 Center, Left 或 Right
SelectionBullet	读写	使用这个属性可以确定选中的文本是否格式化为项目符号的格式，或使用它插入或删除项目符号
BulletIndent	读写	使用这个属性可以指定项目符号的缩进像素值
SelectionColor	读写	可以修改选中文本的颜色
SelectionFont	读写	可以修改选中文本的字体
SelectionLength	读写	使用这个属性可以设置或获取选中文本的长度
SelectionType	只读	这个属性包含了选中文本的信息。它可以确定是选择了一个或多个 OLE 对象，还是仅选择了文本
ShowSelectionMargin	读写	如果把这个属性设置为 true，在 RichTextBox 的左边就会出现一个页边距，这将使用户更易于选择文本。
UndoActionName	只读	如果用户选择撤销某个动作，该属性将获取该动作的名称
SelectionProtected	读写	把这个属性设置为 true，可以指定不修改文本的某些部分

从上面的列表可以看出，大多数新属性都与选中的文本有关。这是因为在用户处理其文本时，对它们应用的任何格式化操作都是对用户选择出来的文本进行的。万一没有选择出文本，格式化操作就从光标所在的位置开始应用，该位置称为插入点。

13.8.2 RichTextBox 控件的事件

RichTextBox 使用的大多数事件与 TextBox 使用的事件相同，表 13-13 中有几个有趣的新事件。

表 13-13

名　称	描　述
LinkedClick	在用户单击文本中的链接时，引发该事件
Protected	在用户尝试修改已经标记为受保护的文本时，引发该事件
SelectionChanged	在选中文本发生变化时，引发该事件。如果因某些原因不希望用户修改选中文本，就可以在该事件中禁止修改

试试看：RichTextBox 范例

在这个范例中，将创建一个非常基本的文本编辑器。它说明了如何修改文本的基本格式，如何加载和保存 RichTextBox 中的文本。为了简单起见，这个范例加载和保存到固定的文件中。

与往常一样，首先设计窗体：

(1) 创建一个新的 C# Windows Application 工程，命名为 RichTextBoxTest。

(2) 创建窗体，如图 13-12 所示。文本框 txtSize 应是一个 TextBox 控件。文本框 rtfText 应是一个 RichTextBox 控件。

(3) 如图所示命名控件，清除 rtfText 和 txtSize 的 Text 属性。

(4) 除了文本框以外，把其他控件的 Text 属性设置为其控件名称(前 3 个字符不算)。

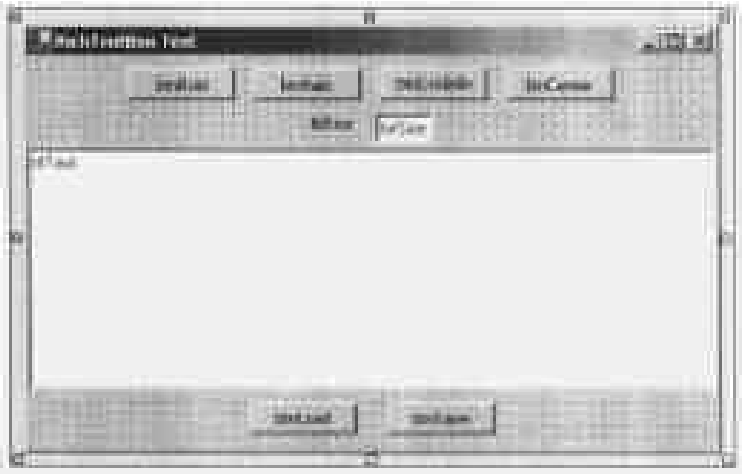


图 13-12

(5) 把 txtSize 文本框的 Text 属性改为 10。

(6) 锚定控件，如表 13-14 所示。

表　13-14

控 件 名 称	Anchor 值
btnLoad 和 btnSave	Bottom
RtfText	Top, Left, Bottom, Right
其他控件	Top

(7) 把窗体的 MinimumSize 属性值设置为 Size 属性的值。

添加事件处理程序

前面是该范例的可见部分，下面添加代码。双击 Bold 按钮，在代码中添加 Click 事件处理程序。下面是该事件的代码：

```
private void btnBold_Click(object sender, System.EventArgs e)
{
    Font oldFont;
Font newFont;

    // Get the font that is being used in the selected text
    oldFont = this.rtfText.SelectionFont;

    // If the font is using bold style now, we should remove the
    // Formatting
    if (oldFont.Bold)
        newFont = new Font(oldFont, oldFont.Style & ~FontStyle.Bold);
    else
        newFont = new Font(oldFont, oldFont.Style | FontStyle.Bold);

    // Insert the new font and return focus to the RichTextBox
    this.rtfText.SelectionFont = newFont;
    this.rtfText.Focus();
}
```

首先获取当前选中文本使用的字体，并把它赋给一个局部变量。然后检查一下选中文本是否为黑体。如果是，就去除黑体设置；否则就设置黑体。使用 oldFont 作为原型，创建一个新字体，但根据需要添加或删除黑体格式。

最后，把新字体赋给选中文本，把焦点返回给 RichTextBox。

提示：

Font 对象的描述参见第 16 章。

btnItalic 和 btnUnderline 的事件处理程序的代码与上面的代码相同，但要修改其对应的样式。双击 Italic 和 Underline 两个按钮，添加下面的代码：

```
private void btnItalic_Click(object sender, System.EventArgs e)
```

```
{
    Font oldFont;
    Font newFont;

    // Get the font that is being used in the selected text
    oldFont = this.rtfText.SelectionFont;

    // If the font is using Italic style now, we should remove it
    if (oldFont.Italic)
        newFont = new Font(oldFont, oldFont.Style & ~FontStyle.Italic);
    else
        newFont = new Font(oldFont, oldFont.Style | FontStyle.Italic);

    // Insert the new font
    this.rtfText.SelectionFont = newFont;
    this.rtfText.Focus();
}

private void btnUnderline_Click(object sender, System.EventArgs e)
{
    Font oldFont;
    Font newFont;
    // Get the font that is being used in the selected text
    oldFont = this.rtfText.SelectionFont;

    // If the font is using Underline style now, we should remove it
    if (oldFont.Underline)
        newFont = new Font(oldFont, oldFont.Style & ~FontStyle.Underline);
    else
        newFont = new Font(oldFont, oldFont.Style | FontStyle.Underline);

    // Insert the new font
    this.rtfText.SelectionFont = newFont;
    this.rtfText.Focus();
}
```

双击最后一个格式化按钮 Center，添加下面的代码：

```
private void btnCenter_Click(object sender, System.EventArgs e)
{
    if (this.rtfText.SelectionAlignment == HorizontalAlignment.Center)
        this.rtfText.SelectionAlignment = HorizontalAlignment.Left;
    else
        this.rtfText.SelectionAlignment = HorizontalAlignment.Center;
    this.rtfText.Focus();
}
```

这里必须检查一下另一个属性 SelectionAlignment，看看选中的文本是否已经居中对齐。HorizontalAlignment 是一个枚举，其值可以是 Left, Right, Center, Justify 和 NotSet。在本例中，只检查一下是否设置了 Center，如果已经设置了。就把对齐方式设置为 Left。如果不是，就设置为 Center。

文本编辑器能进行的最后一个格式化操作是设置文本的大小。为文本框 Size 添加两个事件处理程序，一个处理程序控制输入，另一个处理程序检测用户输入完一个值的时间。

在窗体的构造函数中添加下面的代码行：

```
public Form1()
{
    InitializeComponent();

    // Event Subscription
    this.txtSize.KeyPress += new
        System.Windows.Forms.KeyPressEventHandler(this.txtSize_KeyPress);
    this.txtSize.Validating += new
        System.ComponentModel.CancelEventHandler(this.txtSize_Validating);
}
```

在前面的范例中，有两个事件处理程序。它们都使用一个帮助方法 ApplyTextSize，该方法带一个字符串参数，表示文本的大小：

```
private void txtSize_KeyPress(object sender,
                              System.Windows.Forms.KeyPressEventArgs e)
{
    // Remove all characters that are not numbers, backspace and enter
    if ((e.KeyChar < 48 || e.KeyChar > 57) &&
                                        e.KeyChar != 8 && e.KeyChar != 13)
    {
        e.Handled = true;
    }
    else if (e.KeyChar == 13)
    {
        // Apply size if the user hits enter
        TextBox txt = (TextBox)sender;

        if (txt.Text.Length > 0)
            ApplyTextSize(txt.Text);
        e.Handled = true;
        this.rtfText.Focus();
    }
}

private void txtSize_Validating(object sender,
                                System.ComponentModel.CancelEventArgs e)
```

```
{
    TextBox txt = (TextBox)sender;

    ApplyTextSize(txt.Text);
    this.rtfText.Focus();
}

private void ApplyTextSize(string textSize)
{
    // Convert the text to a float because we'll be needing a float shortly
    float newSize = Convert.ToSingle(textSize);
    FontFamily currentFontFamily;
    Font newFont;

    // Create a new font of the same family but with the new size
    currentFontFamily = this.rtfText.SelectionFont.FontFamily;
    newFont = new Font(currentFontFamily, newSize);

    // Set the font of the selected text to the new font
    this.rtfText.SelectionFont = newFont;
}
```

我们感兴趣的是帮助方法 ApplyTextSize。它首先把文本的大小从字符串转换为浮点数，我们只允许用户输入整数，但在创建新字体时，需要使用浮点数，所以把它转换为正确的数据类型。

之后，获取字体所属的字体系列，从该系列中创建一个带有新字号的新字体。最后，把选中文本的字体设置为新字体。

这就是我们所能进行的所有格式化操作，有一些操作可以由 RichTextBox 本身处理。如果现在尝试运行这个范例，就可以把文本设置为黑体、斜体和下划线，还可以居中文本。这就是我们期望的操作，但还有一些比较有趣的操作。试着在文本中键入一个网址，例如 www.wrox.com，该文本就被控件识别为一个 Internet 地址，加上下划线，当把鼠标指针移到该文本的上面时，鼠标指针就会变成手的形状。单击该文本，就会打开一个网页。我们需要处理用户单击链接时引发的事件：LinkClicked。

为此，在构造函数中订阅如下事件：

```
this.rtfText.LinkClicked += new
        System.Windows.Forms.LinkClickedEventHandler(this.rtfText_LinkedClick);
```

我们以前没有见过这个事件处理程序。它用于提供单击链接的文本，处理程序非常简单，如下所示：

```
private void rtfText_LinkedClick(object sender,
                                 System.Windows.Forms.LinkClickedEventArgs e)
{
```

```
        System.Diagnostics.Process.Start(e.LinkText);
    }
```

这段代码打开了默认的浏览器(如果浏览器没有打开)，并导航到该链接指向的站点。

应用程序的编辑部分就完成了。剩下的只有加载和保存控件的内容。这里使用一个固定的文件。

双击 Load 按钮，添加下面的代码：

```
private void btnLoad_Click(object sender, System.EventArgs e)
{
    // Load the file into the RichTextBox
    try
    {
        rtfText.LoadFile("../../Test.rtf");
    }
    catch (System.IO.FileNotFoundException)
    {
        MessageBox.Show("No file to load yet");
    }
}
```

这就完成了，不需要做其他工作。因为我们处理的是文件，所以总是有可能遇到异常，必须处理这些异常。在 Load 方法中，处理了文件不存在而抛出的异常。保存文件也是这样，双击 Save 按钮，添加下面的代码：

```
private void btnSave_Click(object sender, System.EventArgs e)
{
    // Save the text
    try
    {
        rtfText.SaveFile("../../Test.rtf");
    }
    catch (System.Exception err)
    {
        MessageBox.Show(err.Message);
    }
}
```

现在运行范例，格式化一些文本，再单击 Save 按钮。清空文本框，单击 Load 按钮，刚才保存的文本就会再次显示出来。

这是一个 RichTextBox 范例。运行它时，应得到如图 13-13 所示的结果。

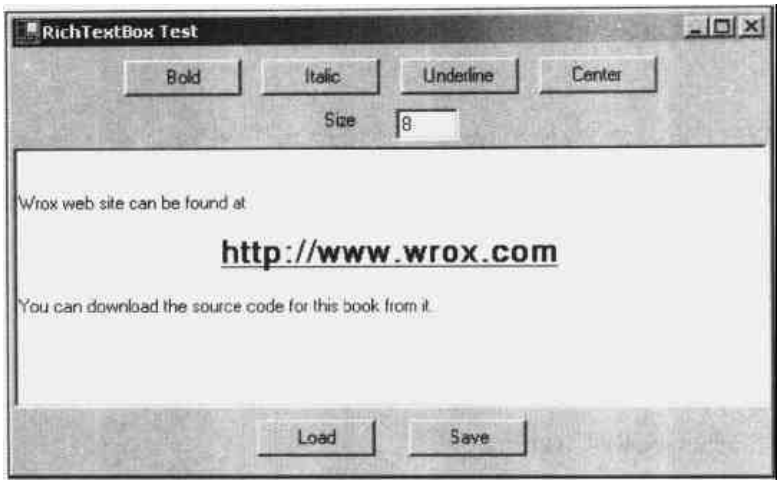


图　13-13

13.9　ListBox 和 CheckedListBox 控件

列表框用于显示一组字符串，可以一次从中选择一个或多个选项。与复选框和单选按钮一样，列表框也提供了要求用户选择一个或多个选项的方式。在设计期间，如果不知道用户要选择的数值个数，就应使用列表框。(例如一起工作的工人列表框)。即使在设计期间知道所有可能的值，但列表中的值非常多，也应考虑使用列表框。

ListBox 类派生于 ListControl 类。后者提供了两种 Visual Studio.NET 内置列表类型控件的基本功能。另一个控件 ComboBox 在本章后面介绍。

另一种类型的列表框由 Visual Studio.NET 提供，称为 CheckedListBox，派生于 ListBox 类。它提供的列表类似于 ListBox，但除了文本字符串以外，每个列表选项还附带有一个选中标记。

13.9.1　ListBox 控件的属性

除非显式声明，表 13-15 中列出的所有属性都可用于 ListBox 类和 CheckedListBox 类。

表　13-15

名　称	可 用 性	描　述
SelectedIndex	读写	这个值表示列表框中选中选项的基于 0 的索引。如果列表框可以一次选择多个选项，这个属性就包含选中列表中的第一个选项
ColumnWidth	读写	在包含多个列的列表框中，这个属性指定列的宽度
Items	只读	Items 集合包含列表框中的所有选项，使用这个集合的属性可以增加和删除选项
MultiColumn	读写	列表框可以有多个列。使用这个属性可以获取或设置列表框中列的个数

(续表)

名　称	可 用 性	描　述
SelectedIndices	只读	这个属性是一个集合，包含列表框中选中选项的所有基于 0 的索引
SelectedItem	读写	在只能选择一个选项的列表框中，这个属性包含选中的选项。在可以选择多个选项的列表框中，这个属性包含选中选项中的第一个选项
SelectedItems	只读	这个属性是一个集合，包含当前选中的所有选项
SelectionMode	读写	在列表框中，可以使用 4 种选择模式 • None: 不能选择任何选项 • One: 一次只能选择一个选项 • MultiSimple: 可以选择多个选项 • MultiExtended: 可以选择多个选项，用户还可以使用 Ctrl, Shift 和箭头键进行选择
Sorted	读写	把这个属性设置为 true 会使列表框对它包含的选项按照字母顺序排序
Text	读写	许多控件都有 Text 属性。但这个 Text 属性与其他控件的 Text 属性大不相同。如果设置列表框控件的 Text 属性，它将搜索匹配该文本的选项，并选择该选项。如果获取 Text 属性，返回的值是列表中第一个选中的选项。如果 SelectionMode 是 None，就不能使用这个属性
CheckedIndicies	只读	(只用于 CheckedListBox)这个属性是一个集合，包含 CheckedListBox 中状态是 checked 或 indeterminate 的所有选项
CheckedItems	只读	(只用于 CheckedListBox)这是一个集合，包含 CheckedListBox 中状态是 checked 或 indeterminate 的所有选项
CheckOnClick	读写	(只用于 CheckedListBox) 如果这个属性是 true，则选项就会在用户单击它时改变它的状态
ThreeDCheckBoxes	读写	(CheckedListBox only) 设置这个属性，就可以选择平面或正常的 CheckBoxes

13.9.2　ListBox 控件的方法

为了高效地操作一个列表框，读者应了解许多可以调用列表框的方法。表 13-16 列出了最常用的方法。除非特别声明，否则这些方法均属于 ListBox 和 CheckedListBox 类。

表 13-16

名　称	描　述
ClearSelected	清除列表框中的所有选项
FindString	查找列表框中第一个以指定字符串开头的字符串，例如 FindString("a")就是查找列表框中第一个以 a 开头的字符串
FindStringExact	与 FindString 类似，但必须匹配整个字符串
GetSelected	返回一个表示是否选择一个选项的值
SetSelected	设置或清除选项
ToString	返回当前选中的选项
GetItemChecked	(只用于 CheckedListBox) 返回一个表示选项是否被选中的值
GetItemCheckState	(只用于 CheckedListBox) 返回一个表示选项的选中状态的值
SetItemChecked	(只用于 CheckedListBox) 设置指定为选中状态的选项
SetItemCheckState	(只用于 CheckedListBox) 设置选项的选中状态

13.9.3　ListBox 控件的事件

正常情况下，在处理 ListBox 和 CheckedListBox 时，使用的事件都与用户选中的选项有关，如表 13-17 所示。

表 13-17

名　称	描　述
ItemCheck	(只用于 CheckedListBox)在列表框中一个选项的选中状态改变时发生该事件
SelectedIndexChanged	在选中选项的索引改变时引发该事件

试试看：ListBox 范例

下面用 ListBox 和 CheckedListBox 创建一个小范例。用户可以查看 CheckedListBox 中的选项，然后单击一个按钮，把选中的选项移动到一般的 ListBox 中。创建的对话框如图 13-14 所示。

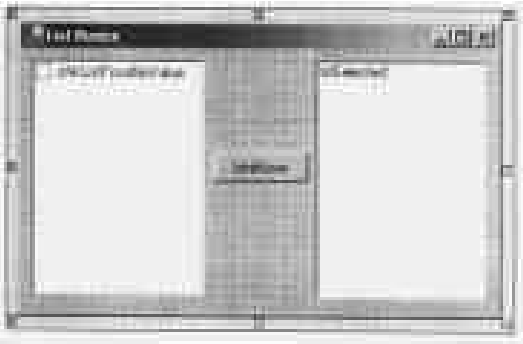


图 13-14

(1) 在 Visual Studio.NET 打开一个新工程 Lists，在窗体上添加一个 ListBox、一个 CheckedListBox 和一个按钮，改变其名称，如图 13-14 所示。

(2) 把按钮的 Text 属性改为 Move。

(3) 把 CheckedListBox 的属性 CheckOnClick 改为 true。

添加事件处理程序

现在准备添加一些代码。当用户单击 Move 按钮时，要查找被选中了的选项，再把它们复制到 Selected 列表框中。

双击该按钮，输入下面的代码：

```
private void btnMove_Click(object sender, System.EventArgs e)
{
    // Check if there are any checked items in the CheckedListBox
    if (this.chkListPossibleValues.CheckedItems.Count > 0)
    {
        // Clear the ListBox we'll move the selections to
        this.lstSelected.Items.Clear();

        // Loop through the CheckedItems collection of the CheckedListBox
        // and add the items in the Selected ListBox
        foreach (string item in this.chkListPossibleValues.CheckedItems)
        {
            this.lstSelected.Items.Add(item.ToString());
        }

        // Clear all the checks in the CheckedListBox
        for (int i = 0; i < this.chkListPossibleValues.Items.Count; i++)
            this.chkListPossibleValues.SetItemChecked(i, false);
    }
}
```

首先查看一下 CheckedItems 集合的 Count 属性。如果集合中有选中的选项，该属性就会大于 0。接着清除 Selected 列表框中的所有选项，循环 CheckedItems 集合，把每个选项添加到 Selected 列表框中。最后，删除 CheckedItems 中的所有选中标记。

现在需要移动 CheckedListBox 中的内容。可以在设计模式时添加选项，方法是选择属性面板上的 Items 属性，在其中添加选项。另外，还可以通过编写代码来添加选项。在窗体的构造函数中添加下面的代码：

```
public Form1()
{
    //
    // Required for Windows Form Designer support
    //
    InitializeComponent();
```

```
        // Fill the CheckedListBox
        this.chkListPossibleValues.Items.Add("One");
        this.chkListPossibleValues.Items.Add("Two");
        this.chkListPossibleValues.Items.Add("Three");
        this.chkListPossibleValues.Items.Add("Four");
        this.chkListPossibleValues.Items.Add("Five");
        this.chkListPossibleValues.Items.Add("Six");
        this.chkListPossibleValues.Items.Add("Seven");
        this.chkListPossibleValues.Items.Add("Eight");
        this.chkListPossibleValues.Items.Add("Nine");
        this.chkListPossibleValues.Items.Add("Ten");
    }
```

如果要通过属性面板输入值，就可以在运行期间使用 Items 集合添加选项。

这是一个列表框范例，如果现在运行它，将得到如图 13-15 所示的结果。

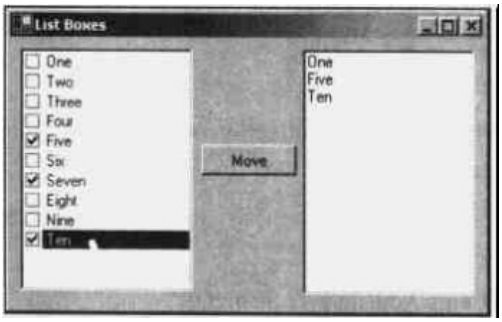


图　13-15

13.10　ComboBox 控件

顾名思义，组合框合并了许多控件，特别是 TextBox, Button 和 ListBox 控件。与 ListBox 不同，组合框从来都不能在列表中选择多个选项，但可以在 ComboBox 的 TextBox 部分输入新选项。

一般情况下，ComboBox 控件可以节省对话框中的空间，因为组合框中可见的部分只有文本框和按钮部分。当用户单击文本框右边的箭头按钮时，就会打开一个列表框，用户可以在其中进行选择。只要完成了选择，列表框就会消失，回到原来的显示状态。

下面介绍该控件的属性和事件，然后创建一个使用 ComboBox 和两个 ListBox 控件的范例。

13.10.1　ComboBox 控件的属性

因为 ComboBox 控件包含 TextBox 和 ListBox 控件的功能，该控件的许多属性也属于 TextBox 和 ListBox 控件。所以，ComboBox 有非常多的属性和事件，这里仅介绍最常见的属性和事件如表 13-18 所示。要查看完整的列表，请参阅 MSDN 库。

表　13-18

名　称	可 用 性	描　述
DropDownStyle	读写	组合框可以用 3 种不同的样式来显示： • DropDown: 用户可以编辑控件的文本框部分，且必须单击箭头按钮，才能显示控件的列表部分 • Simple: 与 DropDown 相同，但控件的列表部分总是可见的，类似于一般的 ListBox • DropDownList: 用户不能编辑控件的文本框部分，且必须单击箭头按钮，才能显示控件的列表部分
DroppedDown	读写	表示控件的列表部分是否可以下拉。如果把这个属性设置为 true，则列表将打开
Items	只读	这个属性是一个集合，包含组合框中包含的所有列表选项
MaxLength	读写	把这个属性设置为非 0 值，可以控制输入到控件文本框部分的最大字符个数
SelectedIndex	读写	表示列表中当前选中的选项索引
SelectedItem	读写	表示列表中当前选中的选项
SelectedText	读写	表示在控件的文本框部分中被选中的文本
SelectionStart	读写	在控件的文本框部分，这个属性表示选中的第一个字符的索引
SelectionLength	读写	在控件的文本框部分，被选中文本的长度
Sorted	读写	把这个属性设置为 true，会使控件对列表部分的选项按字母排序
Text	读写	如果把这个属性设置为 null，则删除控件中列表部分的任何选项。如果把这个属性设置为一个值，该值位于控件的列表部分，就选择该值。如果该值不在列表中，就显示文本框部分的文本

13.10.2　ComboBox 控件的事件

在处理组合框时，应注意 3 个主要操作：

- 选项的改变
- 下拉状态的改变
- 文本的改变

要处理这 3 个动作，一般应订阅下面的一个或多个事件，如表 13-19 所示。

表 13-19

名　称	描　述
DropDown	在下拉控件的列表部分时引发
SelectedIndexChanged	在改变了控件的列表部分中的选项时引发
KeyDown KeyPress KeyUp	当控件的文本框部分获得焦点时，如果按下了一个键，就会引发这些事件，请参阅本章前面文本框一节中对这些事件的介绍
TextChanged	在 Text 属性改变时引发

试试看：ComboBox 范例

再次使用 TextBox 范例。在 CheckBoxes 范例中，我们把用于职业的 TextBox 改为 CheckBox。注意用户可以有除程序员以外的其他职业，所以应再次修改该对话框，包容更多的职业。为此，使用一个组合框，其中包含两个职业：顾问和程序员，并允许用户在文本框中添加自己的职业。

因为用户每次使用该应用程序时，不必手动输入其职业，所以在每次关闭对话框时，都应把 ComboBox 中的选项保存到一个文件中，而在每次启动应用程序时，都应加载该文件。

下面先修改对话框的外观：

(1) 删除复选框 chkProgrammer。

(2) 添加一个标签和一个组合框，命名它们，如图 13-16 所示。

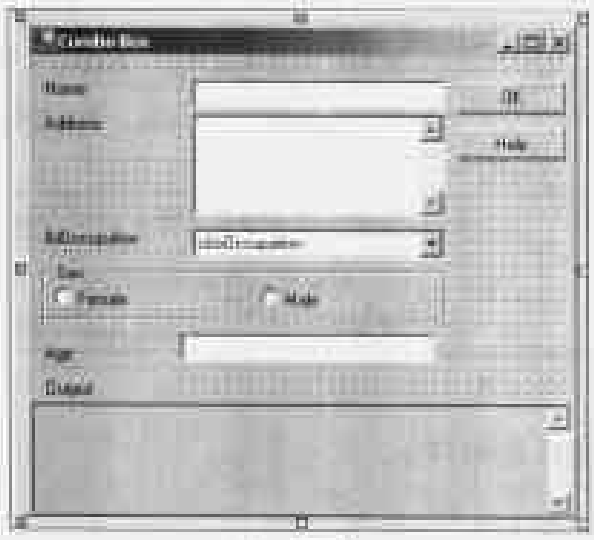


图 13-16

(3) 把标签的 Text 属性改为 Occupation，清空组合框的 Text 属性。

(4) 对窗体的其他内容不需要修改。仅要用下面的两行代码创建一个文本文件 Occupations.txt:

```
Consultant
Programmer
```

注意：

这个范例的代码在本章的 ComboBox Visual Studio.NET 工程的下载代码中。

添加事件处理程序

现在准备修改代码。在开始编写新代码时，修改 btnOK_Click 事件处理程序，以匹配窗体的改变：

```
private void btnOK_Click(object sender, System.EventArgs e)
{
    // No testing for invalid values are made, as that should
    // not be neccessary

    string output;

    // Concatenate the text values of the controls
    output = "Name: " + this.txtName.Text + "\r\n";
    output += "Address: " + this.txtAddress.Text + "\r\n";
    output += "Occupation: " + this.cboOccupation.Text + "\r\n";
    output += "Sex: " + (string)(this.rdoFemale.Checked ? "Female" : "Male") +
              "\r\n";
    output += "Age: " + this.txtAge.Text;

    // Insert the new text
    this.txtOutput.Text = output;
}
```

这里没有使用复选框，而使用了组合框，所以在选择组合框中的一个选项时，它就会显示在控件的可编辑部分中。所以我们感兴趣的值总是在组合框的 Text 属性中。

下面准备编写新代码，首先要创建一个方法，加载文件中已有的值，并把它们插入到组合框中：

```
private void LoadOccupations()
{
    try
    {
        // Create a StreamReader object. Change the path to where you put
        // the file
        System.IO.StreamReader sr =
                    new System.IO.StreamReader("../../Occupations.txt");

        string input;

        // Read as long as there are more lines
        do
```

```
        {
            input = sr.ReadLine();

            // Add only if the line read contains any characters
            if (input != "")
                this.cboOccupation.Items.Add(input);
        } while (sr.Peek() != -1);
                                    // Peek returns -1 if at the end of the stream

        // Close the stream
        sr.Close();
    }
    catch (System.Exception)
    {
        MessageBox.Show("File not found");
    }
}
```

流读取器在第 20 章介绍，所以这里不讨论这段代码的本质。总之，我们使用它打开文本文件 Occupations.txt，一次一行地为组合框读取选项。对 Items 集合使用 Add()方法，把文件中的每个选项都添加到组合框中。

用户应能在组合框中输入新选项，所以应添加检查回车键是否被按下的代码。如果组合框 Text 属性中的文本在 Items 集合中不存在，就给组合框添加新选项：

```
private void cboOccupation_KeyDown(object sender,
                                   System.Windows.Forms.KeyEventArgs e)
{
    int index = 0;
    ComboBox cbo = (ComboBox)sender;

    // We only want to do something if the enter key was pressed
    if (e.KeyCode == Keys.Enter)
    {
        // FindStringExact searches for the string and is not
        // case-sensitive, which
        // is exactly what we need, as Programmer and programmer is the same.
        // If we find a match we'll move the selection in the ComboBox to
        // that item.
        index = cbo.FindStringExact(cbo.Text);
        if (index < 0) // FindStringExact return -1 if nothing was found.
            cbo.Items.Add(cbo.Text);
        else
            cbo.SelectedIndex = index;

        // Signal that we've handled the key down event
```

```
            e.Handled = true;
        }
    }
```

ComboBox 对象的 FindStringExact()方法搜索完全匹配的字符串(不考虑字符串的大小写)。这是非常有利的，因为我们不希望在集合中添加相同的职业。

如果在 Items 中找不到匹配文本的选项，就添加一个新选项。添加新选项会自动把当前选中的选项设置为新选项。如果找到一个匹配，就选择集合中已有的选项。

还需要订阅事件，这在窗体的构造函数中完成，如下所示：

```
this.txtAge.TextChanged += new System.EventHandler(this.txtBox_TextChanged);
this.cboOccupation.KeyDown += new
            System.Windows.Forms.KeyEventHandler(this.cboOccupation_KeyDown);
```

在用户关闭对话框时，应保存组合框中的选项，这在另一个方法中完成，如下所示：

```
private void SaveOccupation()
{
    try
    {
        System.IO.StreamWriter sw = new
                                System.IO.StreamWriter("../../Occupations.txt");
    foreach (string item in this.cboOccupation.Items)
        sw.WriteLine(item); // Write the item to the file
    sw.Flush();
    sw.Close();
    catch (System.Exception)
    {
        MessageBox.Show("File not found or moved");
    }
}
```

StreamWriter 类在第 20 章介绍，所以这里不讨论代码的细节。但我们再次把文件 IO 代码封装到 try...catch 块中，作为预防措施，以防在打开窗体时有人不小心删除或移动了文本文件。循环 Items 集合中的选项，把每个选项写到文件中。

最后，必须调用刚才定义的 LoadOccupation() 和 SaveOccupation()方法。我们分别在窗体的构造函数和 Dispose 方法中调用它们：

```
public Form1()
{
    .
    .
    .
    this.cboOccupation.KeyDown += new
        System.Windows.Forms.KeyEventHandler(this.cboOccupation_KeyDown);
```

```
    // Fill the ComboBox
    LoadOccupations();
}

public override void Dispose()
{
    // Save the items in the ComboBox
    SaveOccupation();

    base.Dispose();
    if(components != null)
        components.Dispose();
}
```

这是一个 ComboBox 范例，运行这个范例，会得到如图 13-17 所示的结果。

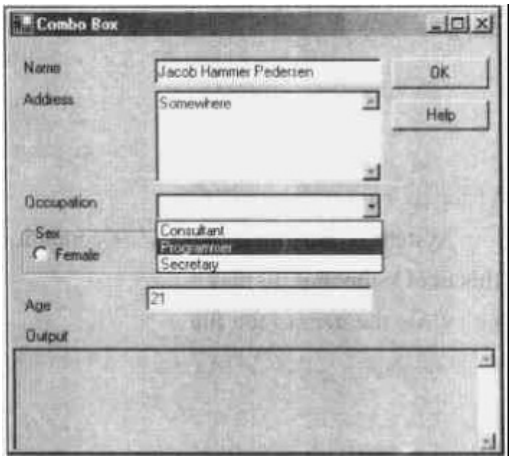


图　13-17

13.11　ListView 控件

在 Windows 中，可以从一个列表中选择要在标准对话框中打开的文件，这个列表就是 ListView 控件。在标准列表视图对话框中可以进行的各种查看操作(大图标、详细视图等)，都可以在 Visual Studio.NET 提供的 ListView 中进行，如图 13-18 所示。

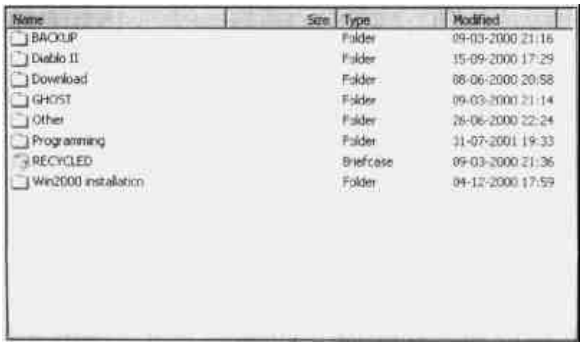


图　13-18

列表视图通常用于显示数据，用户可以对这些数据和显示方式进行某些控制。还可以把包含在控件中的数据显示为列和行(像网格那样)，或者显示为一列，或者显示为图标表示。最常用的列表视图就是图 13-18 中用于导航计算机中文件夹的视图。

ListView 控件是本章中最复杂的一个控件，它包括了超出本书范围的内容，这里仅编写一个范例，使用 ListView 控件中最重要的功能，为用户打下坚实的基础，并全面介绍可以使用的许多属性、事件和方法。本章还将讨论 ImageList 控件，它用于存储在 ListView 控件中使用的图像。

13.11.1　ListView 控件的属性

ListView 的属性如表 13-20 所示。

表　13-20

名　称	可 用 性	说　明
Activation	读写	使用这个属性，可以控制用户在列表视图中激活选项的方式。不要改变默认设置，除非有很好的理由，因为这可能会改变用户为自己的系统设置的选项 可能的值如下： • Standard:这个设置是用户为自己的机器选择的值 • OneClick: 单击一个选项，激活它 • TwoClick: 双击一个选项，激活它
Alignment	读写	这个属性可以控制列表视图中的选项对齐的方式。有 4 个可能的值： • Default: 如果用户拖放一个选项，它将仍位于拖动前的位置 • Left: 选项与 ListView 控件的左边界对齐 • Top: 选项与 ListView 控件的顶边界对齐 • SnapToGrid: ListView 控件包含一个不可见的网格，选项都放在该网格中
AllowColumn Reorder	读写	如果把这个属性设置为 true，就允许用户改变列表视图中列的顺序。如果这么做，就应确保即使改变了列的属性顺序，填充列表视图的例程也能正确插入选项
AutoArrange	读写	如果把这个属性设置为 true，选项会自动根据 Alignment 属性排序。如果用户把一个选项拖放到列表视图的中央，且 Alignment 是 Left，则选项会自动左对齐。只有在 View 属性是 LargeIcon 或 SmallIcon 时，这个属性才有意义
CheckBoxes	读写	如果把这个属性设置为 true，列表视图中的每个选项会在其左边显示一个复选框。只有在 View 属性是 Details 或 List 时，这个属性才有意义

（续表）

名　称	可 用 性	说　明
CheckedIndices CheckedItems	只读	利用这两个属性分别可以访问索引和选项的集合，该集合包含列表中被选中的选项
Columns	只读	列表视图可以包含列。通过这个属性可以访问列集合，通过该集合，可以增加或删除列
FocusedItem	只读	这个属性包含列表视图中有焦点的选项。如果没有选择任何选项，该属性就为 null
FullRowSelect	读写	这个属性为 true 时，单击一个选项，该选项所在的整行文本都会突出显示。如果该属性为 false，则只有选项本身会突出显示
GridLines	读写	把这个属性设置为 true，则列表视图会在行和列之间绘制网格线。只有 View 属性为 Details 时，这个属性才有意义
HeaderStyle	读写	可以控制列标题的显示方式，有 3 种样式： Clickable：列标题显示为一个按钮 NonClickable：列标题不响应鼠标单击 None：不显示列标题
HoverSelection	读写	这个属性设置为 true 时，用户可以通过把鼠标指针放在列表视图的一个选项上，以选择它
Items	只读	列表视图中的选项集合
LabelEdit	读写	这个属性设置为 true 时，用户可以在 Details 视图下编辑第一列的内容
LabelWrap	读写	如果这个属性是 true，标签就会重叠，因为需要有许多行显示所有的文本
LargeImageList	读写	这个属性包含 ImageList，而 ImageList 包含大图像。这些图像可以在 View 属性为 LargeIcon 时使用
MultiSelect	读写	这个属性设置为 true 时，用户可以选择多个选项
Scrollable	读写	这个属性设置为 true，就显示滚动条
SelectedIndices SelectedItems	只读	这两个属性分别包含选中索引和选项的集合
SmallImageList	读写	当 View 属性为 SmallIcon 时，这个属性包含了 ImageList，其中 ImageList 包含了要使用的图像
Sorting	读写	可以让列表视图对它包含的选项排序，有 3 种可能的模式： • Ascending • Descending • None

(续表)

名　称	可 用 性	说　明
StateImageList	读写	ImageList 包含图像的蒙板，这些图像蒙板可用作 LargeImageList 和 SmallImageList 图像的覆盖图，表示定制的状态
TopItem	只读	返回列表视图顶部的选项
View	读写	列表视图可以用 4 种不同的模式显示其选项： • LargeIcon: 所有的选项都在其旁边显示一个大图标(32x32)和一个标签 • SmallIcon: 所有的选项都在其旁边显示一个小图标(16x16)和一个标签 • List: 只显示一列。该列可以包含一个图标和一个标签 • Details: 可以显示任意数量的列。只有第一列可以包含图标

13.11.2　ListView 控件的方法

对于像列表视图这样复杂的控件来说，专用的方法非常少。表 13-21 列出了这些方法。

表　13-21

名　称	描　述
BeginUpdate	调用这个方法，将告诉列表视图停止更新，直到调用 EndUpdate 为止。当一次插入多个选项时使用这个方法很有用，因为它会禁止视图闪烁，并可以大大提高速度
Clear	彻底清除列表视图，删除所有的选项和列
EndUpdate	在调用 BeginUpdate 之后调用这个方法。在调用这个方法时，列表视图会显示出其所有的选项
EnsureVisible	在调用这个方法时，列表视图会滚动，以显示指定索引的选项
GetItemAt	返回列表视图中位于 x,y 的选项

13.11.3　ListView 控件的事件

表 13-22 中所示的是要处理的 ListView 控件事件。

表　13-22

名　称	描　述
AfterLabelEdit	在编辑了标签后，引发该事件
BeforeLabelEdit	在用户开始编辑标签前，引发该事件
ColumnClick	在单击一个列时，引发该事件
ItemActivate	在激活一个选项时，引发该事件

13.11.4 ListViewItem

列表视图中的选项总是 ListViewItem 类的一个实例。ListViewItem 包含要显示的信息，例如文本和图标的索引。ListViewItems 有一个集合 SubItems，其中包含另一个类 ListViewSubItem 的实例。如果 ListView 控件处于 Details 模式下，这些子选项就会显示出来。每个子选项表示列表视图中的一个列。子选项和主选项之间的区别是，子选项不能显示图标。

通过 Items 集合把 ListViewItems 添加到 ListView 中，通过 ListViewItem 上的 SubItems 集合把 ListViewSubItems 添加到 ListViewItem 中。

13.11.5 ColumnHeader

要使列表视图显示列标题，需要把类的实例 ColumnHeader 添加到 ListView 的 Columns 集合中。当 ListView 控件处于 Details 模式下时，ColumnHeaders 为要显示的列提供一个标题。

13.11.6 ImageList 控件

ImageList 控件提供了一个集合，可以用于存储在窗体的其他控件中使用的图像。可以在图像列表中存储任意大小的图像，但在每个控件中，每个图像的大小必须相同。对于 ListView，则需要两个 ImageList 控件，才能显示大图像和小图像。

ImageList 是本章介绍的第一个不在运行期间显示它本身的控件。在把它拖放到正在开发的窗体上时，它并不是放在窗体本身上，而是在它的下面，其中包含所有的组件。这个功能可以防止不是用户界面一部分的控件妨碍窗体设计器。这个控件的处理方式与其他控件相同，但不能移动它。

可以在设计和运行期间给 ImageList 添加图像。如果知道在设计期间需要显示哪些图像，就可以单击 Images 属性右边的按钮，添加这些图像。这会打开一个对话框，在该对话框中，可以浏览要插入的图像。如果选择在运行期间添加图像，就可以通过 Images 集合添加它们。

请参阅下面的 ListView 范例，了解如何使用这个控件。

试试看：ListView 范例

学习使用 ListView 控件及其相关的图像列表的最好方式是利用一个范例。下面创建一个对话框，其中有一个 ListView 和两个 ImageList。ListView 显示硬盘上的文件和文件夹。为了简单起见，我们不提取文件和文件夹中的正确图标，而使用文件夹的标准文件夹图标和文件的信息图标。

双击文件夹，就可以浏览文件夹树，后退按钮可以在文件夹树中向上移动。4 个单选按钮用于在运行期间改变列表视图的模式。如果双击了一个文件，就可以执行它。

与往常一样，首先创建用户界面：

(1) 创建一个新 Visual Studio.NET 工程 ListView。在窗体上添加一个列表视图、一个按钮、一个标签和一个组框。然后在组框中添加 4 个单选按钮，此时窗体应如图 13-19 所示：

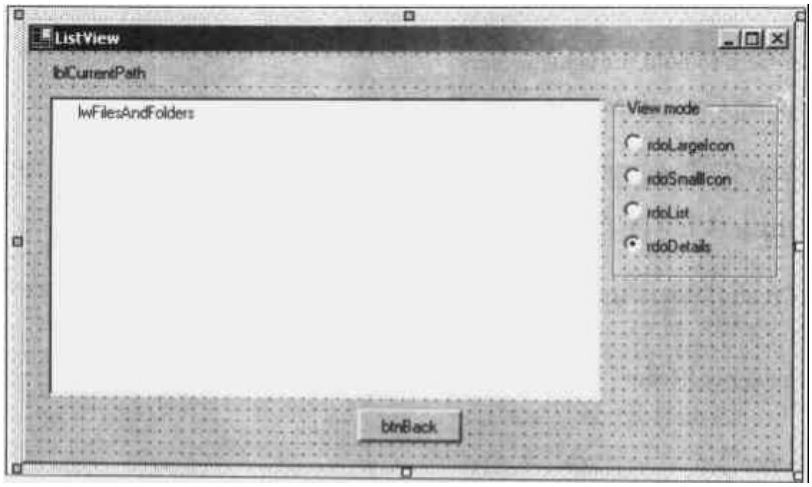


图　13-19

(2) 如图 13-19 所示命名控件。ListView 不在图中显示其名称，并添加了一个选项，提供其名称。读者不应该这么做。

(3) 把单选按钮的 Text 属性的值改为其名称，但没有前 3 个字符。

(4) 清除标签的 Text 属性。

(5) 在工具箱中双击 ImageList 控件的图标(要滚动工具箱才能找到该图标)，在窗体中添加两个图像列表。把它们重新命名为 ilSmall 和 ilLarge。

(6) 把图像列表 ilLarge 的 Size 属性值改为 32, 32。

(7) 单击 ilLage 图像列表的 Images 属性右边的按钮，打开一个对话框，从中可以浏览要插入的图像。

(8) 单击 Add，浏览 Visual Studio.NET 下包含图像的文件夹。这些文件是：

<Drive>:\Program Files\Visual Studio.NET\Common7\Graphics\Icons\Win95\clsdfold.ico 和 <Drive>:\Program Files\Visual Studio.NET\Common7\Graphics\Icons\Computer\msgbox04.ico

(9) 确保文件夹图标位于列表的顶部。

(10) 对另一个图像列表 ilSmall 重复第(7)、(8)步。

(11) 把单选按钮 rdoDetails 的 Checked 属性设置为 true.

(12) 设置列表视图的下述属性：

MultiSelect = true

LargeImageList = ilLarge

SmallImageList = ilSmall

View = Details

修改窗体和框架的 Text 属性，如图 13-19 所示。

添加事件处理程序

这是我们的用户界面，下面可以添加代码了。首先，需要一个字段，以包含前面浏览的文件夹，在单击后退按钮时，就可以返回这些文件夹。我们将存储文件夹的绝对路径，所以选择使用 StringCollection：

```
public class Form1 : System.Windows.Forms.Form
```

```
{
    // Member field to hold previous folders
    private System.Collections.Specialized.StringCollection folderCol;
```

在窗体设计器中没有创建任何列标题，现在要在代码中创建。我们使用一个方法CreateHeadersAndFillListView()来创建它们：

```
private void CreateHeadersAndFillListView()
{
    ColumnHeader colHead;

    // First header
    colHead = new ColumnHeader();
    colHead.Text = "Filename";
    this.lwFilesAndFolders.Columns.Add(colHead); // Insert the header

    // Second header
    colHead = new ColumnHeader();
    colHead.Text = "Size";
    this.lwFilesAndFolders.Columns.Add(colHead); // Insert the header

    // Third header
    colHead = new ColumnHeader();
    colHead.Text = "Last accessed";
    this.lwFilesAndFolders.Columns.Add(colHead); // Insert the header
}
```

先声明一个变量 colHead，用于创建 3 个列标题。这 3 个标题都是用 new 关键字创建的，在把它添加到 ListView 的 Columns 集合中之前，将文本赋给它。

在第一次显示窗体时，进行的最后一个初始化工作是用硬盘上的文件和文件夹填充列表视图。这通过另一个方法来完成：

```
private void PaintListView(string root)
{
    try
    {
        // Two local variables that is used to create the items to insert
        ListViewItem lvi;
        ListViewItem.ListViewSubItem lvsi;

        // If there's no root folder, we can't insert anything
        if (root.CompareTo("") == 0)
            return;

        // Get information about the root folder.
        System.IO.DirectoryInfo dir = new System.IO.DirectoryInfo(root);
```

```
// Retrieve the files and folders from the root folder.
DirectoryInfo[] dirs = dir.GetDirectories(); // Folders
FileInfo[] files = dir.GetFiles();                    // Files

// Clear the ListView. Note that we call the Clear method on the
// Items collection rather than on the ListView itself.
// The Clear method of the ListView remove everything, including column
// headers, and we only want to remove the items from the view.
this.lwFilesAndFolders.Items.Clear();

// Set the label with the current path
this.lblCurrentPath.Text = root;

// Lock the ListView for updates
this.lwFilesAndFolders.BeginUpdate();

// Loop through all folders in the root folder and insert them
foreach (System.IO.DirectoryInfo di in dirs)
{
    // Create the main ListViewItem
    lvi = new ListViewItem();
    lvi.Text = di.Name; // Folder name
    lvi.ImageIndex = 0; // The folder icon has index 0
    lvi.Tag = di.FullName; // Set the tag to the qualified path of the
                           // folder

    // Create the two ListViewSubItems.
    lvsi = new ListViewItem.ListViewSubItem();
    lvsi.Text = ""; // Size - a folder has no size and so this column
                    // is empty
    lvi.SubItems.Add(lvsi); // Add the sub item to the ListViewItem

    lvsi = new ListViewItem.ListViewSubItem();
    lvsi.Text = di.LastAccessTime.ToString(); // Last accessed column
    lvi.SubItems.Add(lvsi); // Add the sub item to the ListViewItem

    // Add the ListViewItem to the Items collection of the ListView
    this.lwFilesAndFolders.Items.Add(lvi);
}

// Loop through all the files in the root folder
foreach (System.IO.FileInfo fi in files)
{
    // Create the main ListViewItem
```

```
                lvi = new ListViewItem();
                lvi.Text = fi.Name; // Filename
                lvi.ImageIndex = 1; // The icon we use to represent a folder has
                // index 1
                lvi.Tag = fi.FullName; // Set the tag to the qualified path of the
                // file

                // Create the two sub items
                lvsi = new ListViewItem.ListViewSubItem();
                lvsi.Text = fi.Length.ToString(); // Length of the file
                lvi.SubItems.Add(lvsi); // Add to the SubItems collection

                lvsi = new ListViewItem.ListViewSubItem();
                lvsi.Text = fi.LastAccessTime.ToString(); // Last Accessed Column
                lvi.SubItems.Add(lvsi); // Add to the SubItems collection

                // Add the item to the Items collection of the ListView
                this.lwFilesAndFolders.Items.Add(lvi);
            }

            // Unlock the ListView. The items that have been inserted will now
            // be displayed
            this.lwFilesAndFolders.EndUpdate();
        }
        Catch (System.Exception err)
        {
            MessageBox.Show("Error: " + err.Message);
        }
    }
```

在第一个 foreach 块中，对 ListView 控件调用了 BeginUpdate()。记住 ListView 控件上的 BeginUpdate()方法告诉 ListView 控件，停止更新其可见区域，直到调用了 EndUpdate()为止。如果没有调用这个方法，列表视图的填充就会进行得很慢，列表可能在填充选项时闪烁。在第二个 foreach 块的后面调用了 EndUpdate()，就可以使 ListView 控件显示出填充到它里面的内容。

这两个 foreach 块包含了我们感兴趣的代码。首先创建 ListViewItem 的一个新实例，再把 Text 属性设置为要插入的文件名或文件夹名。ListViewItem 的 ImageIndex 表示 ImageList 中一个选项的索引。所以两个 ImageList 中的图标有相同的索引是非常重要的。使用 Tag 属性保存文件夹和文件的完全限定路径，在用户双击选项时，将使用该路径。

然后创建两个子选项，将要显示的文本赋给这两个子选项，再把它们添加到 ListViewItem 的 SubItems 集合中。

最后，把 ListViewItem 添加到 ListView 的 Items 集合中。ListView 非常聪明，知道如果视图模式不是 Details，就应忽略子选项。所以，现在无论视图模式是什么，都可以增加子选项。

剩下的就是列表视图应显示根文件夹，为此，在窗体的构造函数中调用两个函数。同时用

根文件夹实例化 folderCol StringCollection：

```
InitializeComponent();

// Init ListView and folder collection
folderCol = new System.Collections.Specialized.StringCollection();
CreateHeadersAndFillListView();
PaintListView(@"C:\");
folderCol.Add(@"C:\");
```

为了允许用户双击 ListView 中的选项，浏览文件夹，需要订阅 ItemActivate 事件。在构造函数中添加事件的订阅：

```
this.lwFilesAndFolders.ItemActivate += new
                    System.EventHandler(this.lwFilesAndFolders_ItemActivate);
```

对应的事件处理程序如下所示：

```
private void lwFilesAndFolders_ItemActivate(object sender, System.EventArgs e)
{
    // Cast the sender to a ListView and get the tag of the first selected
    // item.
    System.Windows.Forms.ListView lw = (System.Windows.Forms.ListView)sender;
    string filename = lw.SelectedItems[0].Tag.ToString();

    if (lw.SelectedItems[0].ImageIndex != 0)
    {
        try
        {
            // Attempt to run the file
            System.Diagnostics.Process.Start(filename);
        }
        catch
        {
            // If the attempt fails we simply exit the method
            return;
        }
    }
    else
    {
        // Insert the items
        PaintListView(filename);
        folderCol.Add(filename);
    }
}
```

选中选项的标记包含被双击的文件或文件夹的完全限定路径。索引为 0 的图像是一个文件

夹，所以查看索引就可以确定哪个选项是文件，哪个选项是文件夹。如果选项是一个文件，就试着加载它。

如果选项是一个文件夹，就通过新文件夹调用 PaintListView，再把新文件夹添加到 folderCol 集合中。

在讨论单选按钮前，先给 Back 按钮添加单击事件，提供完整的浏览功能。双击该按钮，给事件处理程序添加如下代码：

```
private void btnBack_Click(object sender, System.EventArgs e)
{
    if (folderCol.Count > 1)
    {
        PaintListView(folderCol[folderCol.Count-2].ToString());
        folderCol.RemoveAt(folderCol.Count-1);
    }
    else
    {
        PaintListView(folderCol[0].ToString());
    }
}
```

如果 folderCol 集合中有多个选项，我们就不在浏览器的根文件夹下。对该路径调用 PaintListView，进入上面的文件夹。folderCol 集合中的最后一个选项是当前文件夹，这就是需要第二次提取最后一个选项的原因。然后删除集合中的最后一个选项，使前面一个选项成为当前文件夹。如果该集合中只有一个选项，就只需对该选项调用 PaintListView。

剩下的是修改列表视图的查看类型。双击每个单选按钮，添加如下代码：

```
private void rdoLarge_CheckedChanged(object sender, System.EventArgs e)
{
    RadioButton rdb = (RadioButton)sender;
    if (rdb.Checked)
        this.lwFilesAndFolders.View = View.LargeIcon;
}

private void rdoList_CheckedChanged(object sender, System.EventArgs e)
{
    RadioButton rdb = (RadioButton)sender;
    if (rdb.Checked)
        this.lwFilesAndFolders.View = View.List;
}

private void rdoSmall_CheckedChanged(object sender, System.EventArgs e)
{
    RadioButton rdb = (RadioButton)sender;
    if (rdb.Checked)
```

```
        this.lwFilesAndFolders.View = View.SmallIcon;
    }

    private void rdoDetails_CheckedChanged(object sender, System.EventArgs e)
    {
        RadioButton rdb = (RadioButton)sender;
        if (rdb.Checked)
            this.lwFilesAndFolders.View = View.Details;
    }
```

检查单选按钮，看看它是否改为 Checked。如果是，就设置 ListView 的 View 属性。这就是 ListView 范例。运行它，将得到如图 13-20 所示的结果。

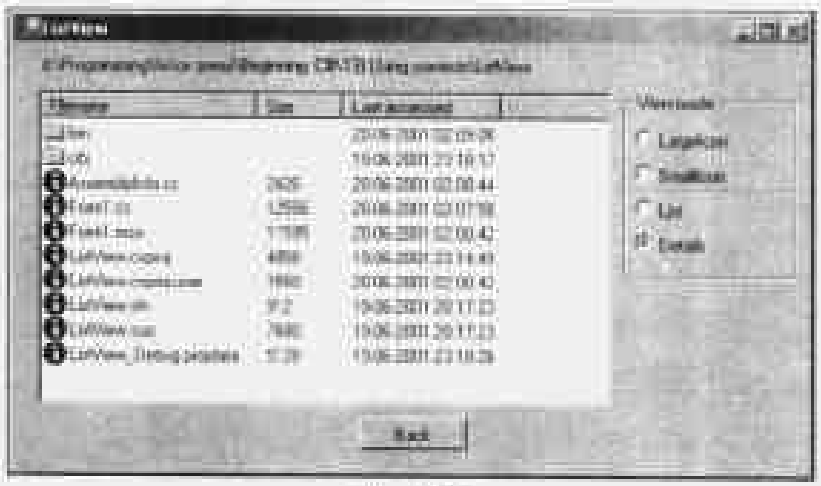


图 13-20

13.12　StatusBar 控件

状态栏常用于在对话框中提供选中选项的提示或当前执行的操作的信息。一般情况下，StatusBar 放在屏幕的底部，如在 MS Office 应用程序中，但它也可以位于其他地方。Visual Studio.NET 提供的状态栏可以用于显示一个文本，或者给它添加面板，来显示文本，或者创建自己的例程，以显示面板的内容，如图 13-21 所示。



图 13-21

图 13-21 显示的是 MS Word 中的状态栏。状态栏中的面板可以用凹陷下去的部分来区分。

13.12.1　StatusBar 控件的属性

如上所述，可以设置 StatusBar 控件的 Text 属性，给用户显示简单的文本，也可以创建面板，使用它们达到相同的效果，StatusBar 的属性如表 13-23 所示。

表 13-23

名　称	可 用 性	描　述
BackgroundImage	读写	可以给状态栏赋予一个图像，显示在背景上
Panels	只读	这是状态栏上的面板集合。使用这个集合可以添加和删除面板
ShowPanels	读写	如果要显示面板，这个属性必须设置为 true
Text	读写	如果不使用面板，这个属性就存储显示在状态栏中的文本

13.12.2 StatusBar 控件的事件

状态栏没有很多新事件，但如果手动绘制一个面板，DrawItem 事件就是一个非常重要的事件，如表 13-24 所示。

表 13-24

名　称	描　述
DrawItem	在设置了 OwnerDraw 样式的面板需要重新绘制时，引发该事件。如果要自己绘制面板的内容，必须订阅这个事件
PanelClick	在单击面板时发生

13.12.3 StatusBarPanel 类

状态栏中的每个面板都是 StatusBarPanel 类的一个实例。这个类包含 Panels 集合中所有面板的信息。这些信息可以是简单文本和文本队列，也可以是要显示的图标和面板的样式。

如果要自己绘制面板，必须把面板的 Style 属性设置为 OwnerDraw，并处理 StatusBar 的 DrawItem 事件。

13.12.4 StatusBar 范例

下面修改前面创建的 ListView 范例，说明 StatusBar 控件的用法。我们将删除用于显示当前文件夹的标签，把信息移动到状态栏的面板上。还要显示第二个面板，其中包括列表视图的当前查看模式。

(1) 删除标签 lblCurrentFolder。

(2) 双击工具箱中的 StatusBar 控件，把它添加到窗体上(放在列表的底部)。新控件会自动停放在窗体的底部边界。

(3) 把 StatusBar 的名称改为 sbInfo，并清除 Text 属性。

(4) 找到 Panels 属性，双击它右边的按钮，打开一个添加面板的对话框。

(5) 单击 Add，给集合添加一个面板。把 AutoSize 属性设置为 Spring，这表示该面板与 StatusBar 中的其他面板共用相同的空间。

(6) 再次单击 Add，把 AutoSize 属性改为 Contents，这表示该面板会把其大小改为它包含

的文本大小。把 MinSize 属性设置为 0。

(7) 单击 OK 按钮，关闭该对话框。

(8) 把 StatusBar 上的 ShowPanels 属性设置为 true。

注意：

这个范例的代码可以在本章下载的 Visual Studio.NET 工程 StatusBar 中找到。

添加事件处理程序

这就是用户界面，下面讨论代码。首先在 PaintListView 方法中设置当前路径。删除设置标签中文本的一行代码，并在该位置插入下述代码：

```
this.sbInfo.Panels[0].Text = root;
```

第一个面板的索引为 0，并设置它的 Text 属性，就像设置标签的 Text 属性一样。最后，修改 4 个单选按钮的 CheckedChanged 事件，设置第二个面板的文本：

```
private void rdoLarge_CheckedChanged(object sender, System.EventArgs e)
{
    RadioButton rdb = (RadioButton)sender;
    if (rdb.Checked)
    {
        this.lwFilesAndFolders.View = View.LargeIcon;
        this.sbInfo.Panels[1].Text = "Large Icon";
    }
}

private void rdoList_CheckedChanged(object sender, System.EventArgs e)
{
    RadioButton rdb = (RadioButton)sender;
    if (rdb.Checked)
    {
        this.lwFilesAndFolders.View = View.List;
        this.sbInfo.Panels[1].Text = "List";
    }
}

private void rdoSmall_CheckedChanged(object sender, System.EventArgs e)
{
    RadioButton rdb = (RadioButton)sender;
    if (rdb.Checked)
    {
        this.lwFilesAndFolders.View = View.SmallIcon;
        this.sbInfo.Panels[1].Text = "Small Icon";
    }
```

```
}

private void rdoDetails_CheckedChanged(object sender, System.EventArgs e)
{
    RadioButton rdb = (RadioButton)sender;
    if (rdb.Checked)
    {
        this.lwFilesAndFolders.View = View.Details;
        this.sbInfo.Panels[1].Text = "Details";
    }
}
```

面板上的文本设置与在上面的 PaintListView 中设置完全相同。

这就是 StatusBar 范例，运行它，应得到如图 13-22 所示结果。

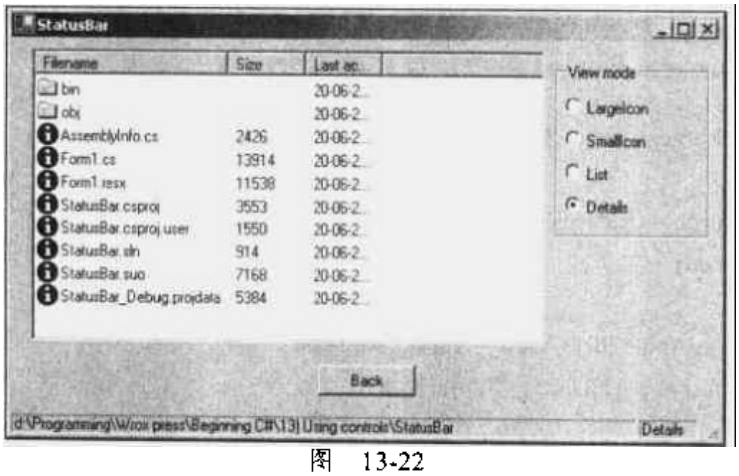


图　13-22

13.13　TabControl 控件

TabControl 提供了一种简单的方式，可以把对话框组织为富有逻辑的部分，以便根据控件顶部的标签来访问。TabControl 包含 TabPages，它的工作方式与 GroupBox 控件非常类似，但它比较复杂，如图 13-23 所示。

上面的屏幕图是 MS Word 2000 中一般配置的 Options 对话框。注意对话框顶部的两行标签。单击其中的每一个标签都会在对话框的剩余空间中显示不同的控件集合。它清楚地说明了如何使用标签控件来组合相关信息，使用户易于查找需要的信息。

标签控件的使用是非常简单的。可以在控件的 TabPages 集合中添加任意数量的标签，再把要显示的控件拖放到各个页面上。

在运行期间，可以通过控件的属性浏览这些标签。

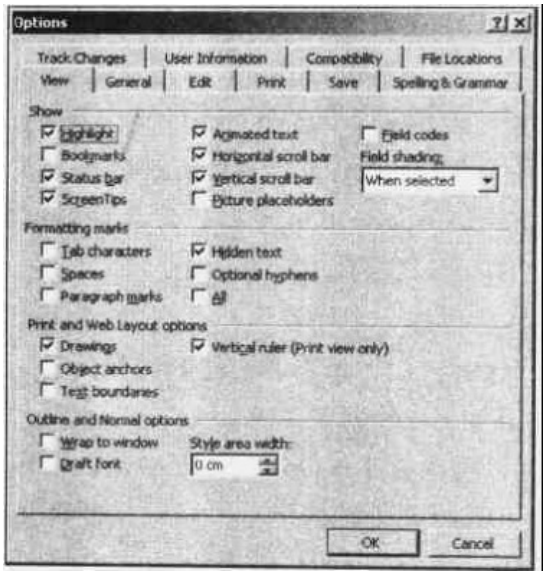


图　12-23

13.13.1　TabControl 控件的属性

TabControl 的属性(如表 13-25 所示)一般用于控制 TabPage 容器的外观，特别是正在显示的选项卡。

表　13-25

名　称	可　用　性	描　述
Alignment	读写	控制标签在标签控件的什么位置。默认的位置为控件的顶部
Appearance	读写	控制标签的显示方式。标签可以显示为一般的按钮或带有平面样式
HotTrack	读写	如果这个属性设置为 true，则当鼠标指针滑过控件上的标签时，其外观就会改变
Multiline	读写	如果这个属性设置为 true，就可以有几行标签
RowCount	只读	返回当前显示的标签行数
SelectedIndex	读写	返回或设置选中标签的索引
TabCount	只读	返回标签的总数
TabPages	只读	这是控件中的 TabPages 集合。使用这个集合可以添加和删除 TabPages

13.13.2　使用 TabControl 控件

TabControl 的工作方式与前面的控件有一些区别。在把该控件拖放到窗体上时，会看到一个灰色的矩形，它与上图中显示的控件有些差别。另外，在属性面板的下面，还有两个按钮，看起来似乎是链接，其标题分别是 Add Tab 和 Remove Tab。单击 Add Tab 会在控件上插入一个新的标签，且该控件可以识别。显然，还可以用 Remove Tab 链接按钮删除该标签。

上面的过程可以让用户很快使用和运行该控件。另一方面，如果要改变标签的操作方式或

样式，就应使用 TabPages 对话框，在选择属性面板上的 TabPages 时，可以通过按钮访问该对话框。

TabPages 属性也是用于访问标签控件上各个页面的集合。下面创建一个范例，说明该控件的基本内容。这个范例还说明了如何把控件放在标签控件的不同页面上：

(1) 创建一个新的 C# Windows Application 工程，命名为 TabControl。

(2) 把一个 TabControl 控件从工具箱拖放到窗体上。

(3) 单击 Add Tab，在控件上添加一个标签。

(4) 找到 TabPages 属性，选择它后，单击它右边的按钮。

(5) 单击 Add，在控件中添加另一个标签。

(6) 把标签页的 Text 属性分别改为 Tab One 和 Tab Two。

(7) 单击控件顶部的标签，选择要处理的选项卡。选择标有 Tab One 的选项卡。在控件上拖放一个按钮。确保把该按钮放在 TabControl 的框架中。如果把它放在框架的外部，则该按钮就被放在窗体上，而不是标签控件上。

(8) 把按钮的名称改为 btnShowMessage，其 text 属性改为 Show Message。

(9) 单击 Text 属性为 Tab Two 的标签，把一个文本框控件拖放到 TabControl 上。把这个控件命名为 txtMessage，并清除 Text 属性。

(10) 单击对话框中的 OK 按钮，返回窗体。这两个标签应如屏幕图 13-24 和 13-25 所示。

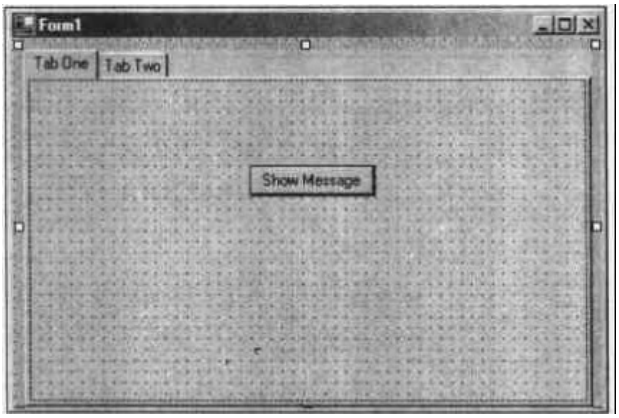


图　13-24

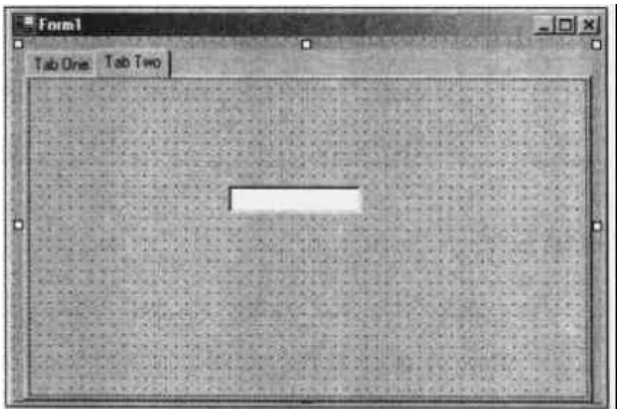


图　13-25

添加事件处理程序

下面准备访问控件。如果运行代码，就会看到选项卡正确显示出来了，为了说明标签控件的用法，剩下要做的工作是添加一些代码，在用户单击标选项卡上的 Show Message 按钮时，在另一个标签页中输入的文本将显示在信息框中。首先，双击第一个选项卡上的按钮，为 Click 事件添加一个处理程序，再添加下述代码：

```
private void btnShowMessage_Click(object sender, System.EventArgs e)
{
    // Access the TextBox

    MessageBox.Show(this.txtMessage.Text);
}
```

在标签页上访问一个控件，与访问窗体上的其他控件是一样的。获取文本框的 Text 属性，在信息框中显示它。

在本章的前面，我们介绍了在窗体中一次只能选择一个单选按钮(除非把它们放在组框中)。TabPages 与组框的工作方式是完全相同的，所以可以在不同的选项卡上放置多组单选按钮，而不需要使用组框。

要能处理标签控件，最后要注意的是如何确定当前显示的是哪个选项卡。这可以使用两个属性：SelectedTab 和 SelectedIndex，顾名思义，SelectedTab 返回 TabPage 对象，如果没有选择标签，就返回 null。而 SelectedIndex 返回标签的索引，如果没有选择标签，就返回 - 1。

13.14　小结

本章介绍了创建 Windows 应用程序时最常用的一些控件，并讨论了如何使用它们创建简单而强大的用户界面。我们还论述了这些控件的属性和事件，给出了使用它们的范例。

本章讨论的控件包括：

- Label
- Button
- RadioButton
- CheckBox
- ComboBox
- ListBox
- ListView
- GroupBox
- RichTextBox
- StatusBar
- ImageList
- TabControl

第 14 章将讨论更复杂的控件，例如菜单和工具栏，并使用它们开发一个多文档界面(*Multi-Document Interface*，MDI)的 Windows 应用程序。我们还将说明如何创建用户控件，该控件组合了本章介绍的简单控件的功能。

第14章 Windows Forms 的高级功能

上一章介绍了 Windows 应用程序开发中一些最常用的控件。使用第 13 章介绍的控件，可以创建界面十分友好的对话框，但 Windows 应用程序的用户界面很少只包含一个对话框。这些应用程序使用单一文档界面(Single Document Interface，SDI)或者多文档界面(Multiple Document Interface，MDI)。这两种类型的应用程序通常会频繁使用菜单和工具栏，第 13 章没有讨论它们，但本章讨论它们。

像第 13 章那样，本章在介绍控件时，首先介绍菜单控件，再介绍工具栏，说明如何把工具栏上的按钮与特定的菜单项链接起来，或把特定的菜单项与工具栏上的按钮链接起来。接着创建 SDI 和 MDI 应用程序，主要讨论 MDI 应用程序，因为 SDI 应用程序基本上是 MDI 应用程序的子集。

到目前为止，我们仅使用了 Visual Studio.NET 提供的控件。这些控件非常强大，提供了许多功能，但有时候使用它们是不够的。所以需要创建定制控件，本章的最后介绍如何创建定制控件。

14.1 菜单

有多少个 Windows 应用程序不包含菜单？这个数字可能接近于 0。在每个为 Windows 操作系统编写的应用程序中，菜单可能是最重要的部分。为了帮助用户创建应用程序的菜单，Visual Studio.NET 提供了一个控件，使用它不必做太多的工作，就可以快速创建简单的菜单和细化菜单。

14.1.1 使用菜单控件

这里要使用的菜单控件叫作 MainMenu，与 Visual Studio.NET 提供的其他两个菜单控件一样，该控件也派生于基类 Menu。

在简要讨论完 MainMenu 后，还要讨论另一种菜单，即 ContextMenu，它用于在用户右击一个选项时弹出一个菜单。从技术上讲，还有第三种菜单类型，即 MenuItem，它也派生于 Menu 类。MenuItem 表示显示在菜单中的各个选项。在图 14-1 中，Type Here 字段表示一个 MenuItem。稍后详细讨论 MenuItem。

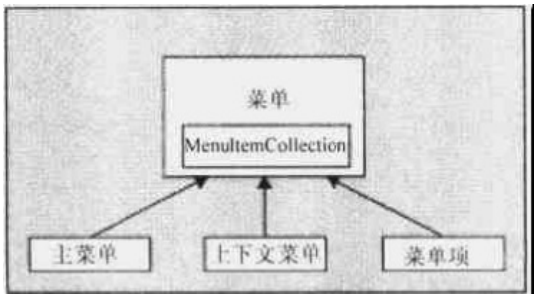


图 14-1

从工具箱中把 MainMenu 控件拖放到设计界面上时，这个控件的行为与状态栏非常类似，因为它位于窗体上，也位于控制盘上。但与状态栏不同的是，菜单控件可以直接在窗体上编辑。要创建新菜单项，只需把指针放在 Type Here 框上，如图 14-2 所示。

在突出显示的框中键入菜单的标题，在要用作该菜单项助记符的字母前面加上一个宏字符(&)(在菜单项中，该字母显示为下划线形式，如图 14-3 所示，可以使用 Alt+Key 组合键来选择该菜单项)。

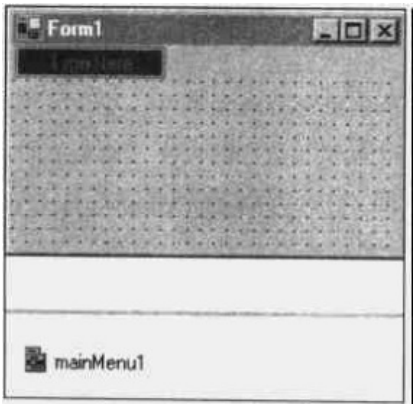


图　14-2

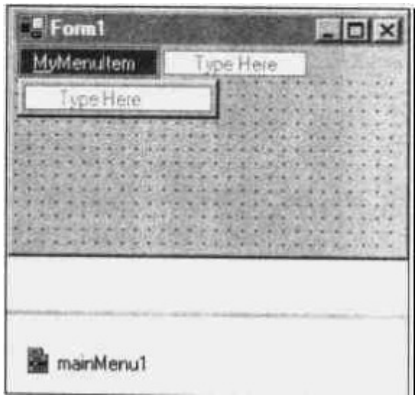


图　14-3

注意可以在一个菜单中用相同的助记符创建好几个菜单项，其规则是每个弹出菜单只能使用一次该字符(例如，在 Files 弹出菜单中使用一次，在 View 菜单中使用一次等)。如果不小心把同一个助记符赋予了一个弹出菜单中的多个菜单项，则只能最靠近顶部的那个菜单项会响应该字符。

要在菜单中创建一个分隔符，可以键入一个短横线(-)，作为文本。注意不能对顶级菜单项进行上述操作，只能在子菜单中使用分隔符，稍后再讨论它。

在单击突出显示的方框时，属性面板上的属性会发生改变。这是因为菜单可以使用的大多数属性也可以应用于各个菜单项。

14.1.2　MenuItem 控件

如果在窗体的控件中选择一个菜单项，就可以使用几个属性控制该菜单项的外观和操作方式。每个菜单项都可以包含其他菜单项，这样就可以为主菜单中的菜单项创建子菜单。在基类 Menu 中定义了一个集合 MenuItemCollection，该集合包含所有的 MenuItem 对象，如图 14-3 中的类所示。

表 14-1 介绍 MenuItem 提供的属性，其中一些属性继承于 Menu。这个列表并不全面，如果需要查看完整的列表，请参阅 MSDN 库。

表　14-1

名　称	可 用 性	描　述
BarBreak	读写	设置这个值，指定菜单应显示在新行上。使用这个属性，可以创建多行菜单，设置了这个属性后，子菜单项将放在新列上。这个属性在 Properties 窗口中不可见

(续表)

名　称	可 用 性	描　述
Checked	读写	表示菜单是否被选中
DefaultItem	读写	默认的菜单项用黑体显示。如果用户双击了包含子菜单项的 MenuItem，且其中一个菜单项是默认菜单项，就选中这个菜单项
Enabled	读写	把 Enabled 设置为 false，菜单项就会灰显，不能被选中
MdiList	读写	表示在 MDI 中，菜单是否用一系列子窗口来填充
MergeOrder	读写	这个属性表示一个菜单与另一个菜单合并时应放在什么位置。在本章后面讨论 MDI 应用程序时，将讨论菜单的合并
MergeType	读写	当要合并一个菜单，但它的 MergeOrder 与另一个菜单相同时，这个属性控制该菜单的操作方式
Mnemonic	只读	这个属性返回菜单项附带的助记符。助记符是紧跟在宏字符(&)后面的那个字符。这个属性在 Properties 窗口中不可见
OwnerDraw	读写	如果把这个属性设置为 true，就由用户绘制所有的菜单项。如果要在菜单中添加图像，就必须把这个属性设置为 true，并执行菜单项的 DrawItem 和 MeasureItem 事件处理程序
RadioCheck	读写	如果把这个属性设置为 true，就把选中标记显示为单选按钮，而不是复选标记
Shortcut	读写	快捷键是直接访问菜单项的键组合(如通过 Ctrl+S 可选中 Save 选项)
ShowShortcut	读写	把这个属性设置为 true，表示快捷键的文本显示在 MenuItem 中
Text	读写	MenuItem 的文本，包括字符串中表示助记符前面的宏字符(&)

14.1.3 MenuItem 控件的事件

MenuItem 需要多少个事件？只需要一个——Click 事件。实际上，可以订阅的事件有 5 个，如表 14-2 所示，如果要自己绘制菜单项，就必须使用其中的两个，下面介绍这些事件，然后创建一个使用该控件的范例。

表 14-2

名　称	描　述
Click	在用户单击菜单项时，引发该事件。也可以调用 MenuItem 的 PerformClick() 方法，手动引发该事件
DrawItem	在需要绘制菜单项时发发。如果把 OwnerDraw 属性设置为 true，就必须处理这个事件
MeasureItem	在绘制菜单项之前引发。如果要手动绘制菜单项，就应在该事件中计算和返回菜单项的宽度和高度

(续表)

名　称	描　述
Popup	在显示子菜单的菜单项时引发该事件。可以使用这个事件验证菜单项的可访问性，并设置它们的状态
Select	在用户把鼠标指针放在菜单项上或使用键盘移动焦点时引发该事件

14.1.4　ContextMenu 控件

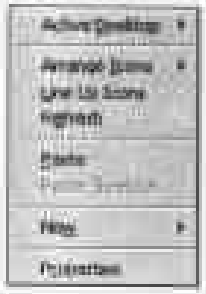


图　14-4

当用户右击窗体上的一个选项时，就会打开上下文菜单。图 14-4 中的上下文菜单是右击计算机的桌面后打开的菜单。

把上下文菜单控件拖放到窗体上，并给它添加菜单项，就可以创建一个上下文菜单，其方式与 MainMenu 控件完全相同。惟一的区别是上下文菜单上没有顶级菜单项。为了把上下文菜单绑定到窗体的控件上，需要选择该控件，设置它的 ContextMenu 属性，使之指向刚才创建的上下文菜单。

试试看：Menu 范例

下面准备创建一个使用菜单的范例。这个范例仍非常简单，因为在后面讨论 MDI 应用程序时，将介绍一些更高级的菜单范例。在这个范例中，要创建一个主菜单和一个上下文菜单：

(1) 首先创建一个新 Windows 应用程序，命名为 MenuExample。

(2) 双击工具箱中的控件，在窗体上添加一个 MainMenu 控件和一个 ContextMenu 控件。把它们分别命名为 MainMenuFiles 和 ContextMenuFonts。

(3) 选择 MainMenu 控件，从 menuItemFiles 开始，把表 14-3 所示的文本输入到菜单项中，把它们添加到 MainMenu 控件中。菜单 menuItemFiles 是顶级菜单。记住在菜单中键入一个短横线，创建一个分隔符，在要选择的字母前面加上一个宏字符(&)，使之成为助记符。

表　14-3

名　称	文　本
MenuItemFiles	&Files
MenuItemNew	&New
MenuItemOpen	&Open
MenuItemSave	&Save
MenuItemSeperator0	-
MenuItemExit	E&xit

(4) 现在选择上下文菜单，像上面的菜单一样创建表 14-4 所示的菜单项。

表 14-4

名　称	文　本
menuItemBold	&Bold
menuItemItalic	&Italic
menuItemUnderline	&Underline

(5) 现在给窗体添加一个 RichTextBox 控件，命名为 rtfText，并把它的 Dock 属性设置为 Fill，填充整个窗体，再清除其 Text 属性。

(6) 选择 RichTextBox 控件，再选择 ContextMenu 属性。从下拉列表中，选择 ContextMenuFonts，把上下文菜单绑定到 RichTextBox 控件上。

现在的窗体应如图 14-5 所示。

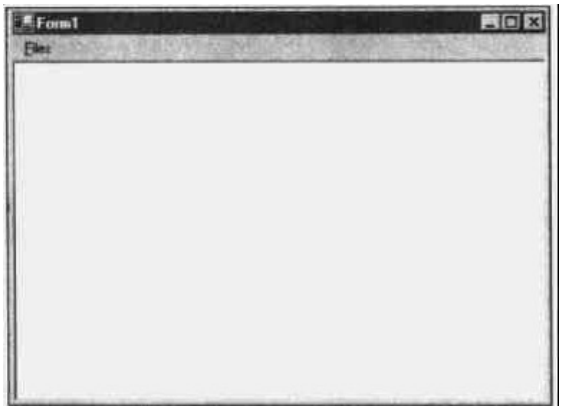


图 14-5

添加事件处理程序

下面准备给菜单添加代码。本例中打开、保存和新建方法仍很简单，再使用一个固定的文件来说明。要添加菜单项的 Click 事件处理程序，可以双击该菜单项，就像按钮和其他控件一样。这里将为 New, Open, Save 和 Exit 菜单项添加处理程序。不能单击分隔符，在用户单击文件菜单时，什么也不做，所以不需要给这个菜单项添加 Click 事件处理程序。

先双击 New 菜单项：

```
private void menuItemNew_Click(object sender, System.EventArgs e)
{
    this.rtfText.Clear();
}
```

显然它非常简单。Sender 参数是被单击的菜单项的引用。在本例中，这是一个 New 菜单项，但事件处理程序可以被赋给许多菜单项，此时该参数就可以标识要使用的菜单。

对 Open, Save 和 Exit 菜单项重复这个过程：

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

```
    // Load the file
    try
    {
        this.rtfText.LoadFile("../../test.rtf");
    }
    catch (System.Exception err)
    {
        MessageBox.Show("Error while loading:\n" + err.Message);
    }
}

private void menuItemSave_Click(object sender, System.EventArgs e)
{
    // Save the file
    try
    {
        this.rtfText.SaveFile("../../test.rtf");
    }
    catch (System.Exception err)
    {
        MessageBox.Show("Error while saving file:\n" + err.Message);
    }
}

private void menuItemExit_Click(object sender, System.EventArgs e)
{
    // Exit the application
    Application.Exit();
}
```

注意这里还添加了错误处理代码，处理文本文件不存在的情况。进行这种检查还有另一种方式。MenuItemFiles 选项包含主菜单中的其他菜单项，所以每次要求菜单项显示其成员时，都会引发一个 Popup 事件。如果订阅这个事件，就可以检查文件是否存在，如果该文件不存在，就禁用 Open 菜单项。把该事件添加到窗体的构造函数中，订阅该事件：

```
public Form1()
{
    //
    // Required for Windows Form Designer support
    //
    InitializeComponent();

    this.menuItemFiles.Popup += new EventHandler(this.menuItemFiles_Popup);
}
```

接着创建事件处理程序本身：

```
private void menuItemFiles_Popup(object sender, System.EventArgs e)
{
    // Check to see if the file exist by setting the Enabled property to the
    // return value of the File.Exists method.
    this.menuItemOpen.Enabled = System.IO.File.Exists("../../test.rtf");
}
```

File 类的静态方法 Exists 返回一个布尔值，如果文件存在，该方法就返回 true，否则返回 false，这就是我们希望 Open 菜单项作出的响应，所以把 Enabled 属性的值设置为这个函数的返回值即可。

处理完主菜单后，就要处理上下文菜单了。像处理主菜单那样添加 Click 事件——双击菜单项，添加下述代码：

```
private void menuItemBold_Click(object sender, System.EventArgs e)
{
    Font newFont = new Font(rtfText.SelectionFont,
                            (rtfText.SelectionFont.Bold ?
                             rtfText.SelectionFont.Style & ~FontStyle.Bold :
                             rtfText.SelectionFont.Style | FontStyle.Bold));
                             rtfText.SelectionFont = newFont;
    }

private void menuItemItalic_Click(object sender, System.EventArgs e)
{
    Font newFont = new Font(rtfText.SelectionFont,
                            (rtfText.SelectionFont.Italic ?
                             rtfText.SelectionFont.Style & ~FontStyle.Italic :
                             rtfText.SelectionFont.Style | FontStyle.Italic));
                             rtfText.SelectionFont = newFont;
    }

private void menuItemUnderline_Click(object sender, System.EventArgs e)
{
Font newFont = new Font(rtfText.SelectionFont,
                        (rtfText.SelectionFont.Underline ?
                         rtfText.SelectionFont.Style & ~FontStyle.Underline :
                         rtfText.SelectionFont.Style | FontStyle.Underline));
                         rtfText.SelectionFont = newFont;
}
```

在这 3 个函数中，从 RichTextBox 的选中文本当前使用的字体中创建一个新字体。如果字体的样式已经设置好了，就从新字体中删除它，否则就在字体中包含该样式。接着把选中文本的字体设置为新字体。

创建对象的一行代码看起来可能有点问题，下面解释一下。如果读者不熟悉三元运算符，请参阅第 4 章。在 menuItemBold_Click 方法中使用该行代码：

```
1 Font newFont = new Font(rtfText.SelectionFont,
2                         (rtfText.SelectionFont.Bold ?
3                          rtfText.SelectionFont.Style & ~FontStyle.Bold :
4                          rtfText.SelectionFont.Style | FontStyle.Bold));
5                          rtfText.SelectionFont = newFont;
```

在标记为 2 的一行代码中，确定选中字体的 Bold 属性是否为 true。如果是，就必须为该字体创建一个新字体样式，其中除黑体之外，保留了当前使用的其他样式，这是第 3 行代码中?和:之间的内容。冒号后面是第 4 行，如果 Bold 属性的值是 false，就运行这行代码，即添加黑体样式，而不是删除它。最后，前面讨论的整个事件序列都放在括号中，这两个括号通常用于把三元运算符的逻辑与其他代码区分开来，并提供可读性，但在上面的范例中，读者有可能认为这些括号并没有什么帮助。

这就是菜单的第一个范例。运行代码，将得到如图 14-6 所示的结果。

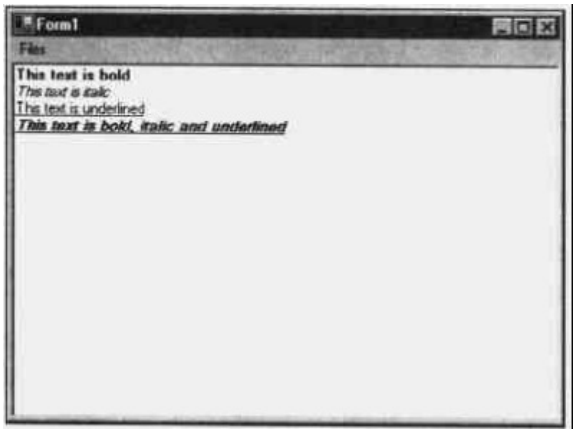


图　14-6

14.2　工具栏

通过菜单可以访问应用程序中的大多数功能，把一些菜单项放在工具栏中和放在菜单中有相同的作用。这些菜单项一般是用户常用的选项，例如 Open 和 Save。工具栏提供了单击访问程序中功能的方式，当然，通过菜单栏访问程序的功能主要比通过菜单来访问程序中的功能要少一些，但用户单击一个一直可见的按钮，要比在菜单中导航方便得多。

屏幕图 14-7 显示了 MS Word 中可见的工具栏部分。



图　14-7

工具栏上的按钮通常包含一个图片，不包含文本，但它可以既包含图片又包含文本。例如 MS Word 中的工具栏按钮就不包含文本(参见图 14-7)。包含文本的工具栏按钮有 MS Internet Explorer 中的工具栏。如果把鼠标指针停留在工具栏的一个按钮上，就会显示一个工具提示，给出该按钮的用途信息，特别是只显示图标时，这是很有帮助的。

与刚才讨论的菜单控件不同，工具栏控件不仅仅是其他对象的容器。还可以对该控件本身设置属性，例如它位于屏幕的什么地方。但工具栏上的所有的按钮都是对象。Toolbar 控件上的每个按钮都是一个 ToolbarButton 对象，它包含了要显示的图像、和图标一起显示的文本样式以及工具提示等信息。

下面先介绍 Toolbar 控件的属性和事件，再介绍 ToolbarButton。

14.2.1 Toolbar 控件的属性

Toolbar 控件的属性管理着控件的显示方式和显示位置。它们还管理着控件中按钮的一些显示设置，这些设置对于控件中包含的所有按钮来说都相同，如表 14-5 所示。

表 14-5

名　称	可 用 性	描　述
Appearance	读写	这个属性控制着控件中包含的所有按钮的外观。可以把这个属性设置为 Flat 或 Normal，Normal 设置会在按钮周围加上一个 3D 边框
AutoSize	读写	把这个属性设置为 false，就可以改变控件的大小。默认值为 true，可以把控件的大小设置为给图像和文本都预留空间
Buttons	只读	返回控件中包含的按钮集合
ButtonSize	读写	这个属性控制按钮的大小。如果 AutoSize 属性是 true，这个属性就不起作用
Divider	读写	如果这个属性设置为 true，控件就会在 Toolbar 控件的顶部绘制一个边框
DropDownArrows	读写	如果工具栏按钮提供了一个下拉列表，这个属性就确定是否在按钮的右边绘制一个箭头
ImageList	读写	一个图像列表，控件中的按钮从该图像列表中提取图像
ShowToolTips	读写	把这个属性设置为 true，控件就会为它包含的每个按钮显示工具提示
Wrappable	读写	如果这个属性设置为 true，且工具栏不够长，不能在一行上显示它包含的所有按钮，就换到下一行

14.2.2 Toolbar 控件的事件

Toolbar 类有两个事件不是派生于 Control 类，当单击了下拉列表中的按钮或向下的箭头时，就引发这两个事件。读者可能认为，按钮本身应引发这些事件，但 Microsoft 选择了另一种方式。

结果是一个事件处理程序必须处理工具栏上所有按钮的单击事件，所以能区分这些按钮，如表 14-6 所示。

表　14-6

名　称	描　述
ButtonClick	单击了工具栏中的按钮时，引发该事件。要确定单击了哪个按钮，应使用发送给事件处理程序的 ToolBarButtonClickEventArgs 参数的 Button 属性。查看这个属性就可以确定单击了哪个按钮
ButtonDropDown	在单击了工具栏中的一个按钮，该按钮的 Style 属性设置为 DropDownButton，或者单击了该按钮对应的箭头时，引发该事件。ToolBarButtonClickEventArgs 可以用于确定单击了哪个按钮，以及该执行什么操作

14.2.3　ToolbarButton 控件的属性

ToolbarButton 类管理着工具栏中每个按钮的各种设置，包括图像、样式和文本，以及工具提示，如表 14-7 所示。 ToolbarButton 的一个有趣功能是如果按钮的样式设置为 Drop Down Button，它就可以包含一个菜单 ——更准确地说，是包含要显示的菜单项的 MenuItem。

表　14-7

名　称	可　用　性	描　述
DropDownMenu	读写	使用这个属性可以把 MenuItem 赋给按钮。如果该按钮的 Style 属性设置为 DropDownButton (参见下面的内容)，当用户单击该按钮时，MenuItem 就会显示出来
Enabled	读写	把这个属性设置为 false，用户就不能访问这个按钮。注意如果在按钮上使用了位图图像，该按钮就不能绘制出彩色图像，而只能绘制黑白图像
ImageIndex	读写	要使用的图像索引。图像列表被赋给 Toolbar 控件
PartialPush	读写	在工具栏按钮的样式设置为 ToggleButton 时，就可以使用这个属性。把这个属性设置为 true，则按钮就会灰显。当 Enabled 属性设置为 fasle 时，这个设置会使整个按钮表面都变模糊
Pushed	读写	如果 ToolbarButton 的样式设置为 ToggleButton，再把这个属性设置为 true 就会使按钮显示为按下状态
Style	读写	可以使用 4 种不同的样式： • PushButton: 显示为一般的按钮 • ToggleButton: 选择这种样式时，按钮显示为按下状态，并一直保持这种状态，直到再次按下按钮为止，这种按钮的一个范例是在 MS Word 的工具栏上使用的 Bold 按钮

(续表)

名　称	可 用 性	描　述
		• Separator: 这种样式的按钮没有绘制出来，它在按钮之间创建了一个小空间。 • DropDownButton: 这种样式的按钮可以被赋给一个MenuItem。如果 MenuItem 包含子菜单项，它们就可以用于创建一个下拉菜单，当用户单击该按钮时，会打开该菜单
Text	读写	在工具栏按钮上要显示的文本
ToolTipText	读写	按钮的工具提示文本

试试看：Toolbar 范例

下面扩展讨论菜单时建立的范例，使之包含一个工具栏。这个简单的工具栏包含三个按钮，每个按钮对应于前面范例中上下文菜单的 3 个菜单项 —— 黑体、斜体和下划线。还有第四个按钮，这是个下拉按钮，可以从中选择一种字体。

注意：

在与 Visual Studio.NET 一起安装的图像中没有字体按钮的位图。用户可以使用自己喜欢的任何位图，或者从www.wrox.com上下载本书的源代码，其中可以找到这里所使用的图像。

首先从本章前面的 MenuExample 代码开始，进行如下修改：

(1) 要添加工具栏，应把 RichTextBox 的 Dock 属性改为 None，并把 Anchor 属性设置为 Left, Top, Right 和 Bottom。

(2) 双击工具栏中的控件，在窗体上添加一个 ImageList 和一个 MainMenu。把它们命名为 ImageListToolbar 和 MainMenuFonts。ImageList 包含要在按钮上使用的图像，MainMenu 是要在下拉列表按钮中使用的菜单。

(3) 把 3 个 MenuItem 添加到 MainMenuFonts 控件上，设置如表 14-8 所示的属性。

表 14-8

名　称	文　本
MenuItemFonts	
menuItemMS	MS Sans Serif
menuItemTimes	Times New Roman

(4) 单击 ImageList 上 Images 属性右边的按钮，给 ImageList 添加 4 个图像。其中 3 个位图图像可以在 Visual Studio 安装目录的 Common7 目录下找到：

<VS.NET install directory>\Common7\Graphics\bitmaps\Tlbr_W95

其名称分别是 BLD.BMP, ITL.BMP 和 UNDRLN.BMP。

最后一个菜单项可以使用自己喜欢的任何位图，或者从 Wrox 上下载源代码，获得本书使

用的位图(Font.bmp)。

(5) 双击工具箱中的 Toolbar 控件，把它添加到窗体上，命名为 toolbarFonts。工具栏现在与 RichTextBox 的一部分重叠了。为了纠正这种情况，单击 RichTextBox，把它移动到工具栏的下面。

(6) 把工具栏的 ImageList 属性设置为 ImageListToolbar。

(7) 选择工具栏的 Button 属性，单击它右边的按钮，给 Toolbar 控件添加按钮。

(8) 在工具栏中添加 5 个按钮，清除它们的 Text 属性。设置这 5 个按钮的属性，如表 14-9 所示。

表　14-9

ToolbarButton	属　性
第一个按钮	Name = toolBarButtonBold ImageIndex =黑体图像的索引(B) Style = ToggleButton
第二个按钮	Name = toolBarButtonItalic ImageIndex = 斜体图像的索引(I) Style = ToggleButton
第三个按钮	Name = toolBarButtonUnderline ImageIndex = 下划线图像的索引(U) Style = ToggleButton
第四个按钮	Name = toolBarButtonSeperator Style = Seperator
第五个按钮	Name = toolBarButtonFonts ImageIndex =最后一个图像的索引 Style = DropDownButton DropDownMenu = MenuItemFonts

窗体现在应如图 14-8 所示。

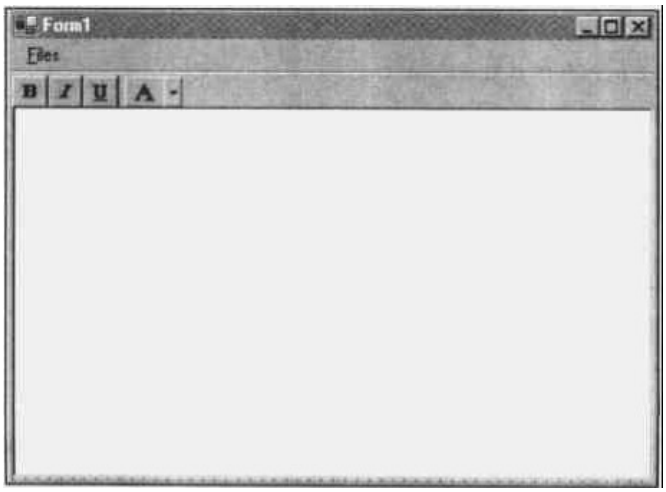


图　14-8

注意：

这个工程是本书 Visual Studio.NET 工程 ToolBars 下载代码的一部分。

添加事件处理程序

下面准备为 Toolbar 的 ButtonClick 事件处理程序添加代码。双击控件，就可以添加事件处理程序：

```
private void toolBarFonts_ButtonClick(object sender,
                                System.Windows.Forms.ToolBarButtonClickEventArgs e)
{
    Font newFont;

    // Switch on the index of the button in the Buttons collection of the
    // ToolBar
    switch (toolBarFonts.Buttons.IndexOf(e.Button))
    {
    case 0: // Bold
        if (e.Button.Pushed)
         // Create a new font with Bold face
            newFont = new Font(rtfText.SelectionFont,
                                rtfText.SelectionFont.Style | FontStyle.Bold);
        else
            // Create a new font without bold face
            newFont = new Font(rtfText.SelectionFont,
                                rtfText.SelectionFont.Style & ~FontStyle.Bold);
        rtfText.SelectionFont = newFont;
        break;
    case 1: // Italic
        if (e.Button.Pushed)
            // Create a new font with italic
            newFont = new Font(rtfText.SelectionFont,
                                rtfText.SelectionFont.Style | FontStyle.Italic);
        else
            // Create a new font without italic
            newFont = new Font(rtfText.SelectionFont,
                                rtfText.SelectionFont.Style & ~FontStyle.Italic);
        rtfText.SelectionFont = newFont;
        break;
    case 2: // Underline
        if (e.Button.Pushed)
            // Create a new font with underline
            newFont = new Font(rtfText.SelectionFont,
                                rtfText.SelectionFont.Style | FontStyle.Underline);
        else
```

```
            // Create a new font without underline
            newFont = new Font(rtfText.SelectionFont,
                               rtfText.SelectionFont.Style & ~FontStyle.Underline);
        rtfText.SelectionFont = newFont;
        break;
    }
}
```

在这个函数中，首先检查被单击按钮的索引。这是必需的，因为在 ToolBar 的 Buttons 集合中，所有的按钮都使用这个事件表示单击。在找到被单击的按钮后，就要测试按钮的 Pushed 属性，看看是否应设置或删除字体样式，并因此创建一个新字体。最后，把 rtfText 控件的 SelectionFont 设置为该新字体。

ToolbarButton 控制着字体的样式，对这些按钮进行的最后一个操作是它们应反映当前选项使用的字体。为此，给 RichTextBox 控件添加 SelectionChanged 事件。首先在构造函数中订阅该事件：

```
public Form1()
{
    InitializeComponent();

    this.menuItemFiles.Popup += new EventHandler(this.menuItemFiles_Popup);
    this.rtfText.SelectionChanged += new
                                     EventHandler(this.rtfText_SelectionChanged);
}
```

再添加事件处理程序本身：

```
private void rtfText_SelectionChanged(object sender, System.EventArgs e)
{
    // Set the toolbar buttons to the correct state of pushed or not
    this.toolBarButtonBold.Pushed = rtfText.SelectionFont.Bold;
    this.toolBarButtonItalic.Pushed = rtfText.SelectionFont.Italic;
    this.toolBarButtonUnderline.Pushed = rtfText.SelectionFont.Underline;
}
```

RichTextBox 控件可以确定当前字体的样式，所以把相应的属性赋给工具栏按钮的 Enabled 属性即可。

剩下的工作就是处理 DropDownButton 样式按钮中两个菜单项的单击事件。为此，给菜单项添加 Click 事件，就像本章前面的菜单范例那样(双击该菜单项)：

```
private void menuItemMS_Click(object sender, System.EventArgs e)
{
    // Create a new font with the correct font family.
    Font newFont = new Font("MS Sans Serif", rtfText.SelectionFont.Size,
                            rtfText.SelectionFont.Style);
```

```
        rtfText.SelectionFont = newFont;
    }

    private void menuItemTimes_Click(object sender, System.EventArgs e)
    {
        // Create a new font with the correct font family.
        Font newFont = new Font("Times New Roman", rtfText.SelectionFont.Size,
                                rtfText.SelectionFont.Style);
        rtfText.SelectionFont = newFont;
    }
```

从要使用的字体系列中创建一种新字体，再把 SelectionFont 设置为新字体。

注意：

如果用户使用的是 Visual Studio.NET 的 Beta 2，在编译这个范例时就会出现许多警告信息。这并不是因为代码有什么问题，而是因为 Microsoft 的工作有一些问题。问题是由 Visual Studio.NET 为 ToolbarButton 生成的代码包含一行设置 ItemData 属性的代码。Microsoft 把这个属性标记为过时，它会在将来的某个时候删除。警告信息要求我们使用 Tag 属性。不要删除这些代码行 —— 只要打开设计窗口，编辑器就会重新插入它们。

现在编译并运行应用程序，得到如图 14-9 所示的结果。

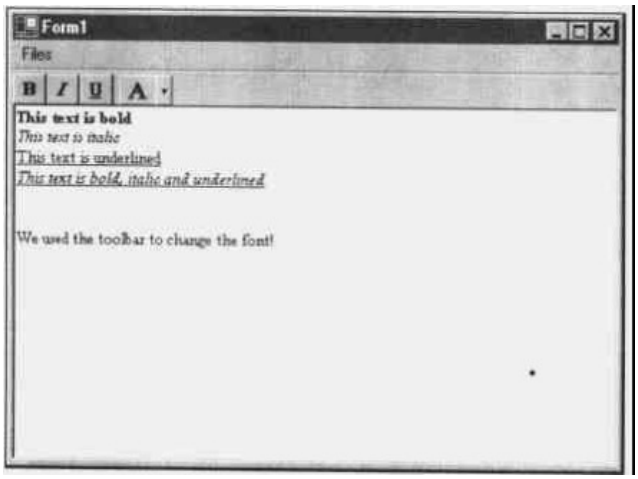


图　14-9

14.3　SDI 和 MDI 应用程序

传统上，可以为 MS Windows 编写 3 种应用程序，它们是：

- 基于对话框的应用程序，它们给用户显示一个对话框，该对话框提供了所有的功能。
- 单一文档界面(SDI)，这些应用程序显示给用户一个菜单、一个或多个工具栏和一个窗口，在该窗口中，用户可以执行某些任务。
- 多文档界面(MDI)，这些应用程序的执行方式与 SDI 相同，但可以同时打开多个窗口。

● 基于对话框的应用程序通常用途比较单一，它们可以完成用户输入量非常少的特定任务，或者专门处理某一类型的数据。这种应用程序的范例是 MS Windows 中的计算器，如图 14-10 所示。

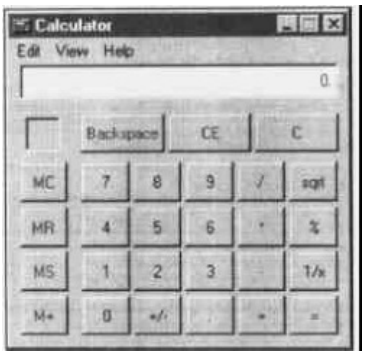


图　14-10

单一文档界面通常用于完成一个特定任务，也允许用户把一个要处理的单一文档加载到应用程序中。但这个任务通常涉及到许多用户交互操作，用户也常常希望能保存或加载工作的结果。SDI 应用程序的范例是 WordPad 和 MS Paint，它们都是 MS Windows 附带的程序，如图 14-11 所示。

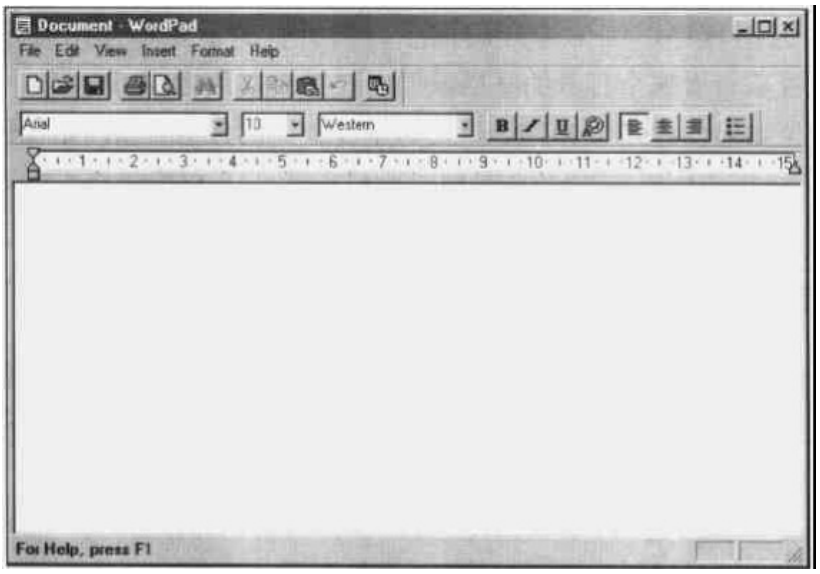


图　14-11

但一次只能处理一个文档，所以如果用户要打开第二个文档，就必须打开一个新的 SDI 应用程序实例，它与第一个实例没有关系，对一个实例的任何配置都不会影响第二个实例。例如，在 MS Paint 的一个实例中，可以把绘图颜色设置为红色，如果打开 MS Paint 的第二个实例，绘图颜色仍是默认的黑色。

多文档界面非常类似于 SDI 应用程序，但它可以在不同的窗口中保存多个已打开的文档，用户可以在任一时刻打开多个窗口。MDI 的标识符包含在菜单栏右边的 Windows 菜单中，该菜单在 Help 的前面。MDI 应用程序的范例有 Adobe Acrobat Reader 和 MS Word 97，如图 14-12 所示：

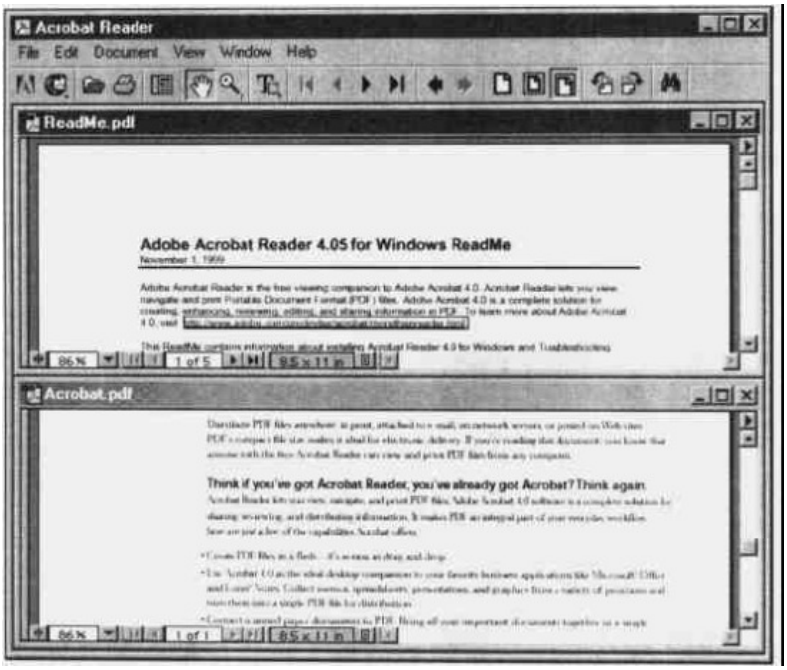


图　14-12

第四种应用程序在 MS Office 2000 中被引入。这种类型的应用程序介于 SDI 和 MDI 之间，显示给用户的窗口没有占用相同的区域，每个窗口都在任务栏中显示出来。最重要的是，应用程序本身是 MDI 应用程序，因为主应用程序直到所有的窗口都关闭时才退出，可以使用 Windows 菜单项选择要查看哪个打开的文档，但用户界面本身与 SDI 一样。

本章将主要讨论如何创建一个 MDI，以及其中涉及到的任务。原因是任何 SDI 应用程序基本上是 MDI 的一个子集，所以如果能创建 MDI，也就能创建 SDI。实际上，第 15 章就将创建一个简单的 SDI 应用程序，用于说明如何使用 Windows 命令对话框。

建立 MDI 应用程序

创建 MDI 会涉及到什么问题？首先，希望用户能完成的任务应是他需要一次打开多个文档的任务。例如，文本编辑器或上图显示的文本查看器。第二，应在应用程序中提供最常用功能的工具栏，例如把字体样式设置为黑体，加载和保存文档等。第三，应提供一个包含 Window 菜单项的菜单，让用户可以重新定位打开的窗口(平铺和层叠)，显示所有已打开窗口的一个列表。MDI 应用程序的另一个功能是如果打开了一个窗口，该窗口包含一个菜单，则该菜单就应集成到应用程序的主菜单上。

MDI 应用程序至少要由两个截然不同的窗口组成。创建的第一个窗口叫作 MDI 容器(Container)，可以在容器中显示的窗口叫作 MDI 子窗口。我们把 MDI 窗器既可以叫“MDI 容器”也可以叫“主窗口”，把 MDI 子容器既可以叫“MDI 子容器”又可以叫“子窗口”。

要创建 MDI 应用程序，首先要像创建其他应用程序那样，在 Visual Studio 中创建一个 Windows 应用程序。要把应用程序的主窗口从一个窗体改为 MDI 容器，只需把窗体的 IsMdiContainer 属性设置为 true 即可。改变窗体的背景色，使之表示该窗体现在只有一种背景色，不应放置任何可见的控件，但也可以放置控件，在某些情况下这也是合理的。

要创建一个子窗口，可以选择 Project | Add New Item，在打开的对话框中选择 Windows

Form，给工程添加一个新窗体。把这个窗体的 MdiParent 属性设置为主窗口的一个引用，该窗体就变成一个子窗口了。不能通过 Properties 面板设置这个属性，只能通过代码来设置。

在 MDI 应用程序可以以最基本的模式显示它本身之前，还有两件事要做：必须告诉 MDI 容器显示哪个窗口，再显示它们。为此，创建要显示窗体的一个新实例，再对它调用 Show()。要显示为子窗口的窗体构造函数应与父容器关联，方法是把它的 MdiParent 属性设置为 MDI 容器的实例。

下面介绍一个小范例，引领我们完成这些步骤，之后再执行更复杂的任务。

试试看：创建一个 MDI 应用程序

(1) 创建一个新的 Windows 应用程序，命名为 MdiBasic。

(2) 选择窗体，设置如表 14-10 所示属性。

表　14-10

属　性	值
Name	frmContainer
IsMdiContainer	True
Text	MDI Basic
WindowState	Maximized

(3) 从 Project | Add New Item 菜单中选择 Windows Form，给解决方案添加一个新窗体，命名为 frmChild。

显示子窗体所需要的代码放在窗体的构造函数中。首先看看子窗口的构造函数：

```
public frmChild(MdiBasic.frmContainer parent)
{
    InitializeComponent();

    // Set the parent of the form to the container
    this.MdiParent = parent;
}
```

为了把子窗体绑定到 MDI 容器上，子窗体必须注册到容器中。为此，设置该窗体的 MdiParent 属性，如上面的代码所示。注意我们使用的构造函数包括参数 parent。

因为 C#没有为定义了自己构造函数的类提供默认的构造函数，所以上面的代码可以防止我们创建没有绑定到 MDI 容器上的窗体实例。

最后，显示窗体，这是在 MDI 容器的构造函数中完成的：

```
public frmContainer()
{
    InitializeComponent();

    // Create a new instance of the child form
```

```
        MdiBasic.frmChild child = new MdiBasic.frmChild(this);

        // Show the form
        child.Show();
    }
```

创建子类的一个新实例，把 this 参数传递给构造函数，其中 this 表示 MDI 容器类的当前实例。然后对子窗体的新实例调用 Show()，就可以了。如果要显示多个子窗口，只需对每个窗口重复上面代码中突出显示的几行代码即可。

现在运行代码，得到如图 14-13 所示的结果。

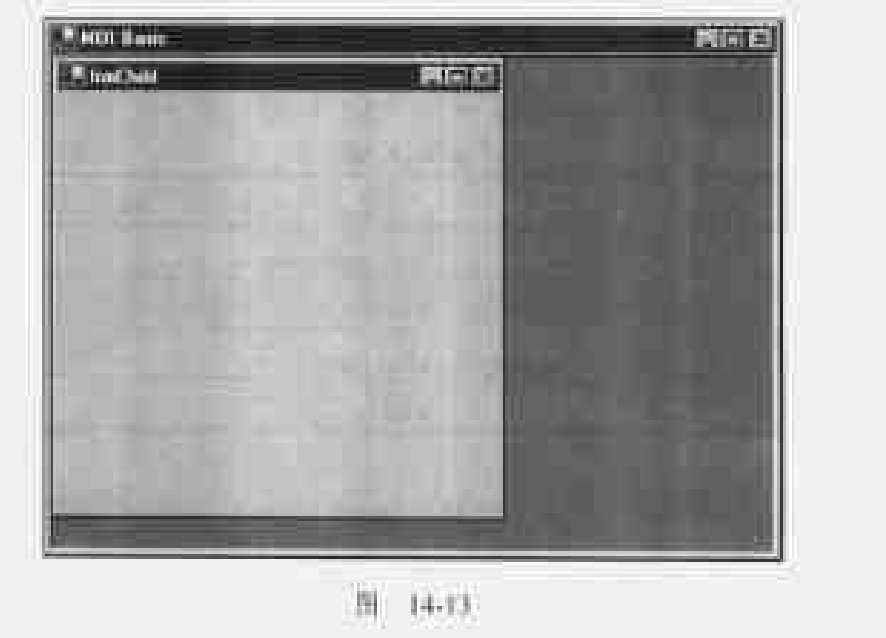


图 14-13

这并不是最吸引人的用户界面，但它是一个好的开始。下一个范例是一个简单的文本编辑器，根据本章前面介绍的菜单和工具栏创建。

试试看：创建一个 MDI 文本编辑器

下面先创建一个基本工程，再讨论其中的内容：

(1) 创建一个新的 Windows Application 工程，命名为 SimpleTextEditor。在窗体上设置下述属性，如表 14-11 所示。

表 14-11

属　性	值
Name	FrmContainer
IsMdiParent	True
Text	Simple Text Editor
WindowState	Maximized

(2) 把包含窗体的文件(文件本身，不包含属性)重命名为 frmContainer.cs，修改 Main()中的

下述代码：

```
static void Main()
{
    Application.Run(new frmContainer());
}
```

(3) 在工程中插入一个现有的选项 (Project | Add Existing Item)。浏览至工具栏范例所在的位置，插入窗体(其名为 Form1.cs)。

(4) 把新插入的窗体重命名为 frmEditor。把包含它的文件重命名为 frmEditor.cs。把 Text 属性设置为 Editor。

这些步骤非常类似于 MDI 基本应用程序，但使用的是现有的窗体，而不是创建新窗体。下面的内容就比较有趣了。

(5) 进入编辑器窗体的代码。把名称空间改为 SimpleTextEditor，把 Form1 的所有实例改为 frmEditor。

(6) 找到编辑器窗体的 Main 方法，删除它。

(7) 修改编辑器窗体的构造函数：

```
public frmEditor(SimpleTextEditor.frmContainer parent)
{
    InitializeComponent();

    // Bind to the parent
    this.MdiParent = parent;
}
```

(8) 修改窗体 frmContainer 的构造函数 ：

```
public frmContainer()
{
    InitializeComponent();

    SimpleTextEditor.frmEditor newForm = new frmEditor(this);
    newForm.Show();
}
```

现在运行应用程序，得到如图 14-14 所示的结果。

注意这里有一些技巧。我们没有为容器创建任何菜单，但在左上角出现了菜单 File。为什么？答案是前面为编辑器窗体创建了一个菜单，而这个窗体现在是容器的一个子窗口，该菜单就被合并到主菜单中，在容器中没有任何菜单，所以只能看到一个菜单项。

如果菜单应包含在子窗口中，则菜单就是这个窗口所特有的。File 菜单在所有窗口中应该都有的，不应包含在子窗口中。原因很明显，如果关闭编辑器窗口，File 菜单就会消失！为了改变这种情况，需要执行下面的步骤：

(9) 右击编辑器窗体中的菜单 MainMenuFiles，选择 Delete，在弹出的信息框中选择 Yes，删除该菜单。

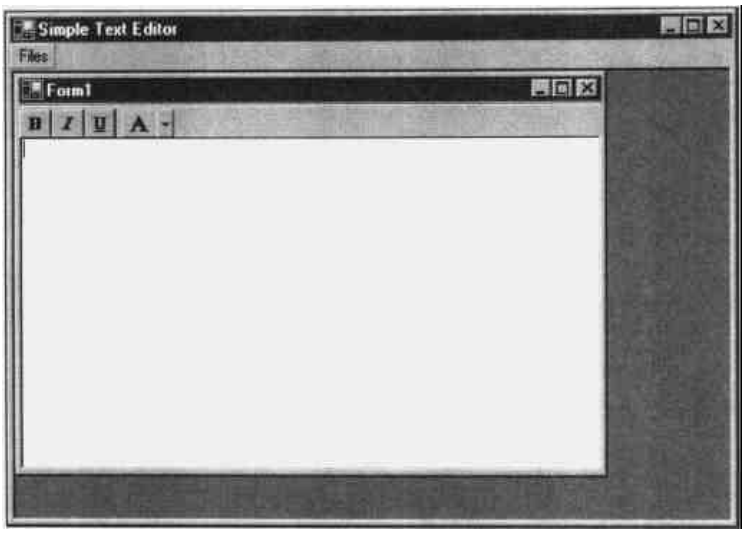


图 14-14

(10) 在代码中，删除该菜单的事件处理程序：

- menuItemFiles_Popup
- menuItemSave_Click
- menuItemNew_Click
- menuItemOpen_Click
- menuItemExit_Click

(11) 在构造函数中，删除订阅 menuItemFiles_Popup 事件的代码。

更适合于编辑器窗体的菜单是 Edit。用下面的步骤创建该菜单：

(12) 在编辑器窗体中添加一个新的 MainMenu，把该控件命名为 MainMenuEdit。

(13) 在其中添加 7 个菜单项，设置如表 14-12 所示的属性。

表 14-12

名　称	文　本	MergeOrder
menuItemEdit	&Edit	0
menuItemUndo	&Undo	N/A
menuItemRedo	&Redo	N/A
menuItemSeperatorEdit0	-	N/A
menuItemCut	Cu&t	N/A
menuItemCopy	&Copy	N/A
menuItemPaste	&Paste	N/A

(14) 把编辑器窗体的 Menu 属性设置为 MainMenuEdit，这个菜单现在应与容器的菜单合并在一起。

现在该创建无论选择哪个窗口都会显示的菜单了。按照下面的步骤给容器添加菜单：

(15) 给窗体 frmContainer 添加一个 MainMenu，命名为 MainMenuContainer。

(16) 添加表 14-13 所示的菜单项。注意 MenuItemWindow 应是一个顶级菜单项，即它应在

File 菜单的右边。

表　14-13

名　称	文　本	MdiList	Top level	MergeOrder
MenuItemFiles	&Files	False	Yes	0
MenuItemNew	&New	False	No	N/A
MenuItemOpen	&Open	False	No	N/A
MenuItemClose	&Close	False	No	N/A
MenuItemSepFiles0	-	False	No	N/A
MenuItemSave	&Save	False	No	N/A
MenuItemSaveAll	Save &All	False	No	N/A
MenuItemSepFiles1	-	False	No	N/A
MenuItemExit	E&xit	False	No	N/A
MenuItemWindow	&Window	False	Yes	1
MenuItemTile	&Tile	False	No	N/A
MenuItemCascade	&Cascade	False	No	N/A
menuItemSepWindow0	-	False	No	N/A
menuItemWindowsOpen	Open Windows	True	No	N/A

现在运行应用程序，Edit 菜单就会位于 File 菜单和 Window 菜单之间。原因是 Window 菜单项的 MergeOrder 属性设置为 1，而 Edit 菜单的 MergeOrder 属性设置为 0。MergeType 是 Add，如果 MergeOrder 的值小于或等于 Edit 菜单的 MergeOrder 属性，Edit 菜单就会添加到所有顶级菜单的右边，如图 14-15 所示。

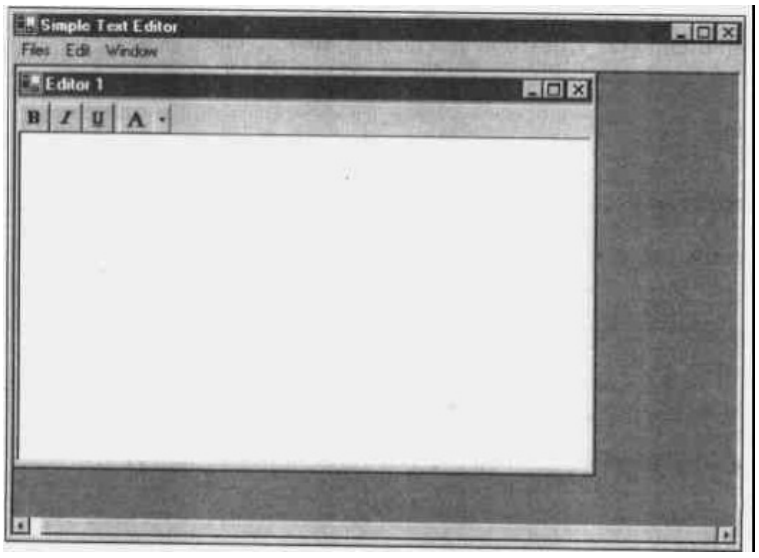


图　14-15

另一个要注意的问题是 Open Windows 菜单项包含一个带有复选标记的子菜单项，如图 14-16 所示。该子菜单项的文本与编辑器窗体的文本相同。在这个菜单中，列出了所有打开的窗口，我们不必编写代码。这是因为把菜单项 MenuItemWindowsOpen 的属性 MdiList 设置为 true 的结果。

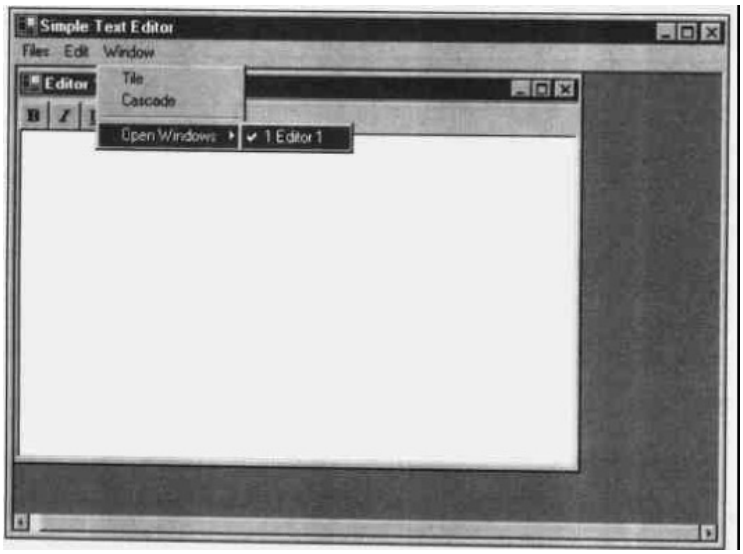


图　14-16

添加事件处理程序

下面添加代码。首先，为 Edit 菜单创建菜单处理程序。双击该菜单中的所有菜单项(除了分隔符和顶级菜单以外)，添加下述代码：

```
private void menuItemUndo_Click(object sender, System.EventArgs e)
{
    rtfText.Undo();
}

private void menuItemRedo_Click(object sender, System.EventArgs e)
{
    rtfText.Redo();
}

private void menuItemCut_Click(object sender, System.EventArgs e)
{
    rtfText.Cut();
}

private void menuItemCopy_Click(object sender, System.EventArgs e)
{
    rtfText.Copy();
}
```

```
private void menuItemPaste_Click(object sender, System.EventArgs e)
{
    rtfText.Paste();
}
```

RichTextBox 控件提供了所有菜单项的对应方法，所以只需调用这些函数即可。与前面的 Menu 范例一样，需要检查一下菜单是应启用，还是禁用。这可以在 Popup 事件处理程序中完成。首先，把该事件添加到窗体 frmEditor 的构造函数中：

```
public frmEditor(SimpleTextEditor.frmContainer parent)
{
    InitializeComponent();

    // Bind to the parent
    this.MdiParent = parent;

    // Subscribe to the popup event of the Edit menu
    this.menuItemEdit.Popup += new EventHandler(this.menuItemEdit_Popup);
}
```

然后添加事件处理程序：

```
private void menuItemEdit_Popup(object sender, System.EventArgs e)
{
    // If there is no text selected, we cannot cut it.
    this.menuItemCut.Enabled = rtfText.SelectedText.Length > 0 ? true : false;

    // If there is no text selected, we cannot copy it.
    this.menuItemCopy.Enabled = rtfText.SelectedText.Length > 0 ? true : false;

    // The CanPaste method of the RichTextBox tells us if there's anything to
    // paste
    this.menuItemPaste.Enabled =
                rtfText.CanPaste(DataFormats.GetFormat(DataFormats.Rtf));

    // The CanUndo property of the RichTextBox tells us if we can undo an
    // action
    this.menuItemUndo.Enabled = rtfText.CanUndo;

    // The CanRedo property of the RichTextBox tells us if we can redo an
    // action
    this.menuItemRedo.Enabled = rtfText.CanRedo;
}
```

RichTextBox 控件再次帮助了我们。调用一个函数或查询一个属性，就可以检查所有的菜

单项，看看它们是处于启用状态还是禁用状态。需要进一步解释的惟一方法是 CanPaste()。这个方法提取要粘贴的文本，如果可以进行粘贴，就返回布尔值 true。

下面看看 MDI 容器中的菜单。首先从 File 菜单中的 New 菜单项开始。单击 New 菜单项时，要创建另一个窗口。前面已经说明了如何创建一个新窗口，因为这是在构造函数中进行的。有一个问题，如果所有的窗口都有相同的标题 Editor，就不可能在创建的 MdiList 中区分它们了。为此，给 Editor 窗体的构造函数添加一个新参数，在 Editor 窗体的构造函数中，传送应在新窗口标题中显示的文本：

```
public frmEditor(SimpleTextEditor.frmContainer parent, string caption)
{
    InitializeComponent();

    // Bind to the parent
    this.MdiParent = parent;
    // Set the caption
    this.Text = caption;

    // Subscribe to the popup event of the Edit menu
    this.menuItemEdit.Popup += new EventHandler(this.menuItemEdit_Popup);
    this.rtfText.SelectionChanged += new
                        EventHandler(this.rtfText_SelectionChanged);
}
```

这种改变意味着，必须在容器的构造函数中修改要构造的调用：

```
public frmContainer()
{
    InitializeComponent();



    SimpleTextEditor.frmEditor newForm = new frmEditor(this, "Editor 1");
    newForm.Show();
}
```

下面准备创建新窗口。双击 New 菜单项，添加下述代码：

```
private void menuItemNew_Click (object sender, System.EventArgs e)
{
    string caption = "Editor " + nextFormNumber++; // The caption

    // Create the new window
    SimpleTextEditor.frmEditor newForm = new SimpleTextEditor.frmEditor(this,
                                caption);
    newForm.Show(); // Show the form
```

```
}
```

首先，创建新标题。NextFormNumber 在类的顶部定义为：

```
private int nextFormNumber = 2;
```

如果选择使用窗体数组(MdiChildren)中的当前窗口数，且打开了两个窗口，就会出问题，数字为 1 的窗口会随即关闭。所以每次在打开新窗体时，要给这个数字加一。下一个窗体的数字是以 2 开始，而不是以 1 开始，原因是构造函数使用文本“Editor 1”初始化第一个窗口。

之后创建编辑器窗体的一个新实例，并显示它。

如果运行代码，并添加一个新窗口，Edit 菜单中的事件就会作用于正确的窗口——这很好，因为我们不需要做任何工作，就可以得到这个结果。事件处理程序是为每个窗体实例定义的，所以每次事件都会传送到正确的地方。在 Open Windows 列表中显示了两个窗口，如图 14-17 所示。



图　14-17

接着要再次关闭窗口。frmContainer 窗体有一个属性 ActiveMdiChild，它可以区分开要关闭的子窗口。双击 Close 菜单，添加如下代码：

```
private void menuItemClose_Click(object sender, System.EventArgs e)
{
    // Get the active MDI child
    SimpleTextEditor.frmEditor frm =
                (SimpleTextEditor.frmEditor)this.ActiveMdiChild;
    if (frm != null) // Make sure the child is valid before using it
    {
        frm.Close(); // Close the window
    }
}
```

首先，把包含在 ActiveMdiChild 属性中的窗体转换为 SimpleTextEditor 类。然后确保在对

窗体进行任何操作前，其实例不是空，之后对该窗口调用 Close()。

这里不对 File 的其他菜单项如 Open, Save 和 SaveAll 等进行这样的处理。这些菜单项使用标准的 Open 和 Save 对话框(详见第 15 章)。Edit 菜单项使用前面范例中的代码。记住可以从 www.wrox.com 上下载所有的代码。

下面解释 Window 菜单上剩余的两个菜单项 Tile 和 Cascade。平铺打开的文档有两种方式——垂直平铺和水平平铺。本例使用其中一种方式，水平平铺窗口。层叠多文档窗口只有一个选项。双击 Tile 和 Cascade 菜单项，添加如下代码：

```
private void menuItemTile_Click(object sender, System.EventArgs e)
{
    this.LayoutMdi(MdiLayout.TileHorizontal);
}

private void menuItemCascade_Click(object sender, System.EventArgs e)
{
    this.LayoutMdi(MdiLayout.Cascade);
}
```

容器窗口的 LayoutMdi 方法可以改变所有 MDI 子窗口的布局。

这就是 SDI 和 MDI 应用程序的讨论。如果运行代码，应能得到如图 14-18 所示的结果(图中显示了 3 个打开文档的平铺布局)。

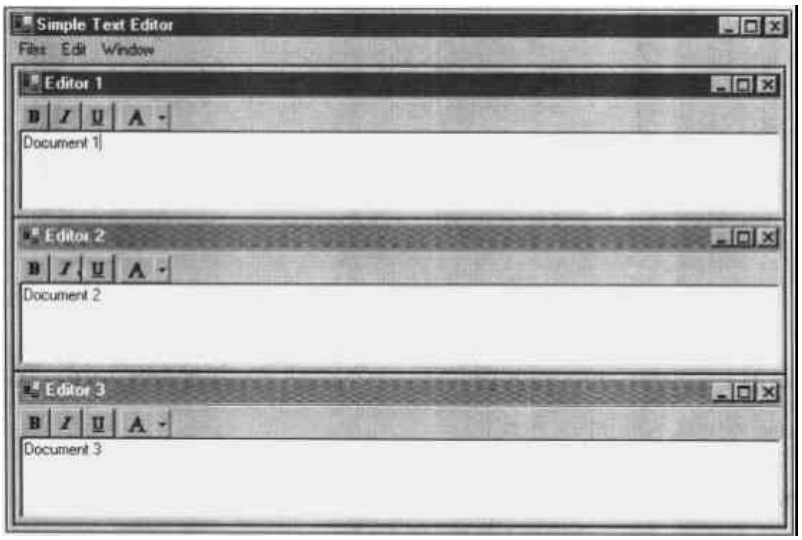


图　14-18

14.4　创建控件

有时 Visual Studio.NET 提供的控件不能满足我们的需要。原因是多方面的，控件不能以希望的方式绘制自己，或者控件只能以某种方式使用，我们希望把控件的功能和界面一起封装，或者需要的控件不存在。为此，Microsoft 提供了创建满足需要的控件的方式。Visual Studio.NET

提供了一个工程类型 Windows Control Library，使用它可以创建自己的控件。

可以开发两种不同类型的自定义控件：用户控件(或组合控件)和定制控件：

- 用户或组合控件：这种控件是根据现有的控件的功能创建一个新控件。这类控件一般用于把控件的用户界面和功能封装在一起，或者把几个其他控件组合在一起，改善控件的界面。
- 定制控件：当没有控件可以满足要求时，就创建这类控件，即从头开始创建控件。它要自己绘出用户界面，在创建控件的过程中没有现有的控件可以使用。当想要创建的控件的用户界面与其他可用的控件都不同时，一般需要创建这样的控件。

本章主要讨论用户控件，因为从头开始设计和绘制定制控件超出了本书的范围。第 16 章介绍 GDI+时，给出了自己绘制选项的方式，用户应能很容易绘制出定制控件。

注意：

在 Visual Studio 6 中使用的 ActiveX 控件放在扩展名为 ocx 的特殊文件子集中，这些文件实际上是 COM DLL。在.NET 中，控件存在的方式与其他装配件一样，这是因为没有 ocx 扩展名了，控件存在于 DLL 中。

用户控件继承于一个类 UserControl。这个基类提供的控件具有.NET 中控件应具有的所有基本功能——我们只需创建控件即可。实际上，任何对象都可以创建为一个控件，包括设计俏皮的标签和功能全面的网格控件，如图 14-19 所示。

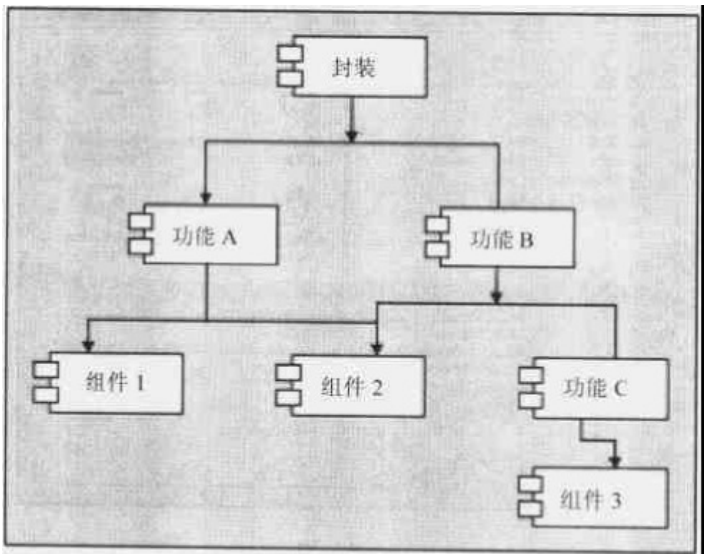


图　14-19

注意：

与用户控件不同，定制控件通常派生于 Control 类，而不是 UserControl 类。

在处理控件时，要考虑许多问题。如果控件不满足这些条件，人们就不会使用它。这些条件是：

- 在设计期间，控件的操作方式应尽可能接近运行期间的操作方式。如果将一个包含标签的控件和一个文本框合并创建了一个 LabelTextbox 控件，标签和文本框就都要在设计期间显示出来，为标签输入的文本也要在设计期间显示出来。在上面的范例中，这是相当简单的，但在

比较复杂的情况下，就会出问题，此时需要采取一种折中的方法。

● 访问控件的属性应可以在窗体设计器中以逻辑的方式进行。例如，ImageList 控件显示了一个对话框，用户可以在该对话框中浏览要包含的图像，导入了图像后，它们就显示在对话框的一个列表中。

在下面的几页中，我们将通过一个范例说明如何创建控件。这个范例创建 LabelTextbox 控件，并说明创建用户控件工程、创建属性和事件，以及调试控件的基础知识。

试试看：LabelTextbox 范例

从这个控件的名称可以看出，这个控件是使用两个现有控件来创建的，一步执行一个任务在 Windows 编程中非常常见：给窗体添加一个标签，再在该窗体中添加一个文本框，把文本框与标签关联起来。下面看看这个控件的用户可以执行什么操作：

● 用户可以把文本框放在标签的右边或下边，如果文本框放在标签的右边，就可以指定该控件的左边界与文本框之间的固定距离，使文本框对齐。

● 用户应能使用文本框和标签的常用属性和事件。

现在我们知道要做什么了，启动 Visual Studio，创建一个新工程：

(1) 在 Visual Studio 中创建一个新的 C#工程。选择 Windows Control Library 工程，把新工程命名为 LabelTextbox，如图 14-20 所示。

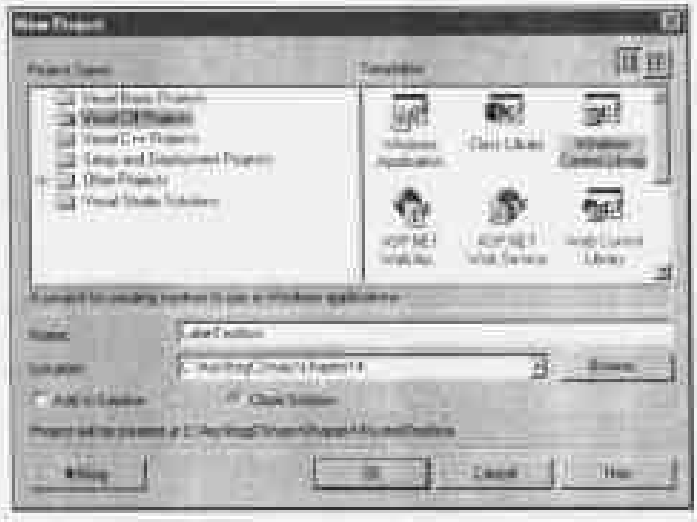


图 14-20

窗体设计器显示了一个设计界面，它看起来与我们经常使用的界面有一些区别。首先，这个界面比一般的小，其次它看起来根本不像是对话框。读者不应因为这个新界面而泄气，其工作方式是一样的。主要的区别是前面都是把控件放在窗体上，现在则是创建一个要放在窗体上的控件，如图 14-21 所示。

(2) 单击设计界面，打开控件的属性。把控件的名称改为 ctlLabelTextbox。

(3) 双击工具箱中的标签，把它添加到用户控件中。把它的 Name 属性改为 lblTextBox，把 Text 属性设置为 Label，把 AutoSize 属性设置为 true。这就将标签控件的大小设置为文本的大小。

(4) 双击工具箱中的文本框，把它的 Name 属性改为 txtLabelText，清除其 text 属性。

在设计期间，我们不知道用户会如何放置这些控件，所以要编写代码，给标签和文本框定

位。这些代码确定了在把 LabelTextbox 控件放在窗体上时控件的位置。

控件的设计是很鼓舞人心的，不仅文本框可随意浮动在设计界面上，而且其界面也太大了。但这并没有负面作用，因为与我们习惯使用的其他控件不同，我们看到的并不是最后得到的结果。给该控件添加的代码会改变控件的外观，但只有在把控件添加到窗体时，才改变它的外观，如图 14-22 所示。



图 14-21

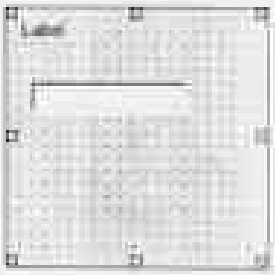


图 14-22

首先，要确定控件彼此的位置。用户应能决定控件如何定位，为此，要给控件添加两个属性，而不是一个属性。一个属性叫作 Position，允许用户选择两个选项：Right 和 Below。如果用户选择了 Right，就使用另一个属性，这个属性叫作 TextboxMargin，是一个 int，表示控件左边界到文本框的像素数。如果用户指定该属性为 0，文本框的右边界就与控件的右边界对齐。

添加属性

为了让用户可以选择 Right 和 Below，先用这两个值定义一个枚举。返回控件工程，进入代码编辑器，添加如下代码：

```
public class ctlLabelTextbox : System.Windows.Forms.UserControl
{
    // Enumeration of the two possible positions
    public enum PositionEnum
    {
        Right,
        Below
    }
```

这是一个第 5 章介绍的一般枚举。现在有一些技巧：我们要把这个位置设置为一个属性，用户可以通过代码和设计器实现这一设置。为此，给 ctlLabelTextbox 类添加一个属性。但首先创建两个成员字段，包含用户要选择的值：

```
// Member field that will hold the choices the user makes
private PositionEnum mPosition = PositionEnum.Right;
private int mTextboxMargin = 0;

public ctlLabelTextbox()
```

```
{
    ...
```

然后添加 Position 属性，如下所示：

```
public PositionEnum Position
{
    get
    {
        return mPosition;
    }
    set
    {
        mPosition = value;
        MoveControls();
    }
}
```

像添加其他属性那样添加这个属性。如果要返回该属性，就返回 mPosition 成员字段，如果要修改 Position，就把值赋给 mPosition，并调用 MoveControls()方法。稍后会介绍 MoveControls()方法，现在知道这个方法可以通过检查 mPosition 和 mTextboxMargin 的值，来定位两个控件就可以了。

TextboxMargin 属性是一样的，但它处理的是一个整数：

```
public int TextboxMargin
{
    get
    {
        return mTextboxMargin;
    }
    set
    {
        mTextboxMargin = value;
        MoveControls();
    }
}
```

添加事件处理程序

在测试两个属性前，还要添加两个事件处理程序。把该控件放在窗体上时，将调用 Load 事件。使用这个事件可以初始化控件和该控件使用的所有资源。我们处理这个事件是为了移动控件，设置它的大小，使之正好包容它包含的两个控件。另一个要添加的事件是 SizeChanged。当改变控件大小时，就引发这个事件。处理这个事件是为了让控件正确地绘制它自己。在控件的构造函数中订阅这两个事件：

```
// Handle the SizeChanged event
this.SizeChanged += new System.EventHandler(this.OnSizeChanged);

// Handle the Load event
this.Load += new EventHandler(this.OnLoad);
```

然后添加事件处理程序：

```
private void OnLoad(object sender, EventArgs e)
{
    lblTextBox.Text = this.Name; // Add a text to the label
    // Set the height of the control
    this.Height = txtLabelText.Height + lblTextBox.Height;
    MoveControls(); // Move the controls
}

private void OnSizeChanged(object sender, System.EventArgs e)
{
    MoveControls();
}
```

再调用 MoveControls()，定位控件。在再次测试控件之前，该介绍这个方法了：

```
private void MoveControls()
{
    switch (mPosition)
    {
    case PositionEnum.Below:
        // Place the top of the Textbox just below the label
        this.txtLabelText.Top = this.lblTextBox.Bottom;
        this.txtLabelText.Left = this.lblTextBox.Left;

        // Change the width of the Textbox to equal the width of the control
        this.txtLabelText.Width = this.Width;        this.Height = txtLabelText.Height +
lblTextBox.Height;
        break;
    case PositionEnum.Right:
        // Set the top of the textbox to equal that of the label
        txtLabelText.Top = lblTextBox.Top;

        // If the margin is zero, we'll place the textbox next to the label
        if (mTextboxMargin == 0)
        {
            int width = this.Width-lblTextBox.Width-3;
            txtLabelText.Left = lblTextBox.Right + 3;
            txtLabelText.Width = width;
```

```
            }
            else
            {
                // If the margin isn't zero, we place the textbox where the user
                // has specified
                txtLabelText.Left = mTextboxMargin;
                txtLabelText.Width = this.Right-mTextboxMargin;
            }
            break;
        }
    }
```

转换 mPosition 的值，确定应把文本框放在标签的下边还是右边。如果用户选择 Below，就把文本框的顶边移动到标签的底边上。然后把文本框的左边界移动到控件的左边界上，把它的宽度设置为控件的宽度。

如果用户选择了 Right，就有两种可能性。如果 TextboxMargin 是 0，就先确定控件中文本框的宽度，然后把文本框的左边界设置为标签文本的右边界，把剩余的空间设置为文本框的宽度。如果用户指定了边距，就把文本框的左边界放在该位置上，再次设置宽度。

下面准备测试这个控件，在开始测试前，先编译控件。

14.4.1　调试用户控件

调试用户控件与调试 Windows 应用程序大不相同。一般情况下，可以在某个位置添加断点，按下 F5，看看发生什么情况。如果读者仍对调试不熟悉，应参阅第 7 章中的详细论述。

控件需要一个容器来显示它本身，我们必须提供一个这样的容器，为此，创建一个 Windows Application 工程：

试试看：调试用户控件

(1) 打开 LabelTextbox 工程，选择 Files | New Project，创建一个新的 Windows Application 工程，命名为 LabelTextboxTest。在继续前，单击 Add to Solution，再单击 OK。

在 Solution Explorer 中，应可以看到已打开了两个工程。第一个工程是我们创建的 LabelTextbox，以黑体字显示。如果要运行解决方案，调试程序就会把控件工程用作启动工程。这将会失败，因为控件不是一种独立的工程。为了更正这个问题，右击新工程名 LabelTextboxTest，选择 Set as Startup Project。如果现在运行解决方案。就会运行 Windows 应用程序工程，且不会产生错误。

(2) 为了让 Windows 应用程序工程知道有 LabelTextbox 控件，需要添加一个对它的引用。在 Solution Explorer 中，右击工程 LabelTextboxTest 下的 References 标题，再选择 Add Reference...。

(3) 选择 Projects 选项卡，其中列出了控件工程。单击 Select，再单击 OK，添加对该工程的引用。

(4) 双击控件 ctlLabelTextbox，把它添加到窗体上。注意，如果在双击 ctlLabelTextbox 时产生了错误，可以试试下面的步骤：右击 Solution Explorer 中的 LabelTextBox，再选择 Remove。

重复步骤(3)，不是选择 Projects 选项卡上的工程，而是单击 Browse，浏览至 LabelTextbox.dll 所在的位置，选择它，再单击 OK。

(5) 进入代码编辑器，找到 InitializeComponent()方法中的一行：

```
this.MyControl = new LabelTextbox.ctlLabelTextbox();
```

可能需要展开 Windows Form Designer generated code 区域。

(6) 在该行上放置一个断点。

(7) 运行代码，代码会停止在我们设置断点的位置上。现在跟踪代码(如果使用默认的键盘映射，就可以按下 F11)。在跟踪代码时，将进入新控件的构造函数，这正是调试组件的地方，也可以设置断点。按下 F5，运行应用程序。

14.4.2　扩展 LabelTextbox 控件

试试看：LabelTextbox 范例(续)

最后，准备测试控件的属性。在包含 LabelTextbox 控件的一个 Visual Studio 新实例中创建一个工程，把控件拖放到窗体上。我们可以看到显示控件名称的标签和占据控件剩余空间的文本框。还要注意在把 LabelTextbox 控件添加到窗体上时，其中的控件会移动到正确的位置上，如图 14-23 所示。



图 14-23

添加更多的属性

现在还不能对该控件进行什么操作，因为它还不能改变标签和文本框中的文本。下面添加两个属性：LabelText 和 TextboxText。属性的添加方式与以前添加属性的方式相同：

```
public string LabelText
{
    get
    {
        return lblTextBox.Text;
    }
```

```
        set
        {
            lblTextBox.Text = value;
            MoveControls(); // Call MoveControls to make the size the textbox if
                            // needed
        }
    }

    public string TextboxText
    {
        get
        {
            return txtLabelText.Text;
        }
        set
        {
            txtLabelText.Text = value;
        }
    }
```

把文本赋给标签和文本框的 Text 属性，如果要插入文本，就返回 Text 属性的值。如果要改变标签的文本，需要调用 MoveControls()，因为标签文本可能会影响文本框的位置。另一方面，插入到文本框中的文本并没有移动控件，且文本比文本框长，超出文本框的部分就不会显示出来。

添加更多的事件处理程序

现在该考虑控件应提供的事件了。因为该控件派生于 UserControl 类，所以继承了该类的许多功能，我们不需要担心它们。但有许多事件我们不希望以标准的方式交给用户。例如 KeyDown, KeyPress 和 KeyUp 事件。需要修改这些事件的原因是用户希望在文本框中按下一个键时就引发这些事件。现在，只有在控件本身获得焦点，且用户按下的一个键时，才会引发这些事件。

要改变其操作方式，必须处理文本框引发的事件，把它们发送给用户。首先在窗体的构造函数中订阅事件：

```
// Handle the SizeChanged event
this.SizeChanged += new System.EventHandler(this.OnSizeChanged);

// Textbox Keyboard events
this.txtLabelText.KeyDown += new KeyEventHandler(this.txtLabelText_KeyDown);
this.txtLabelText.KeyUp += new KeyEventHandler(this.txtLabelText_KeyUp);
this.txtLabelText.KeyPress += new
                              KeyPressEventHandler(this.txtLabelText_KeyPress);
```

接着，添加 3 个事件处理程序：

```
private void txtLabelText_KeyDown(object sender, KeyEventArgs e)
{
    OnKeyDown(e);
}

private void txtLabelText_KeyUp(object sender, KeyEventArgs e)
{
    OnKeyUp(e);
}

private void txtLabelText_KeyPress(object sender, KeyPressEventArgs e)
{
    OnKeyPress(e);
}
```

调用 OnKeyXXX 方法会执行订阅事件的任何方法。

添加定制的事件处理程序

在创建基类中不存在的事件时，需要做更多的工作。下面创建一个事件 PositionChanged，当 Position 属性改变时，将引发该事件。

为了创建这个事件，需要做 3 件事：

- 需要一个合适的委托，用于调用用户赋给事件的方法。
- 用户必须把一个方法赋给事件，以订阅该事件。
- 必须调用用户赋给事件的方法。

要使用的委托是由.NET Framework 提供的 EventHandler 委托。如第 12 章所述，这是一种特殊的委托，它由其关键字 event 声明。下面的代码声明了一个事件，允许用户订阅该事件：

```
public event System.EventHandler PositionChanged;

// Constructor
public ctlLabelTextbox()
{
```

现在只剩下引发该事件了。当改变 Position 属性时，将引发该事件。所以在 Position 属性的 set 存取器中引发该事件：

```
public PositionEnum Position
{
    get
    {
        return mPosition;
    }
    set
    {
        mPosition = value;
```

```
            MoveControls();
            if (PositionChanged != null) // Make sure there are subscribers
            {
                // Get the list of methods to call
                System.Delegate[] subscribers = PositionChanged.GetInvocationList();
                // Loop through the methods
                foreach (System.EventHandler target in subscribers)
                {
                    target(this, new EventArgs()); // Call the method
                }
            }
        }
    }
```

首先，确保检查 PositionChanged 是否为 null，看看有没有订阅者。如果没有，就调用 GetInvocationList，获取要调用的委托列表。然后循环所有的委托，逐个调用它们。

定制事件在 LabelTextboxTest 工程的构造函数中订阅：

```
        public Form1()
        {
            //
            // Required for Windows Form Designer support
            //
            InitializeComponent();

            myControl.PositionChanged +=
                                    new EventHandler(this.myControl_PositionChanged);
            }
        }
```

现在可以编译和运行包含用户控件的测试应用程序了。该应用程序还包含一个按钮，使用该按钮可以设置 Position 属性，并显示一个信息框，以切换控件的外观，如图 14-24 所示。

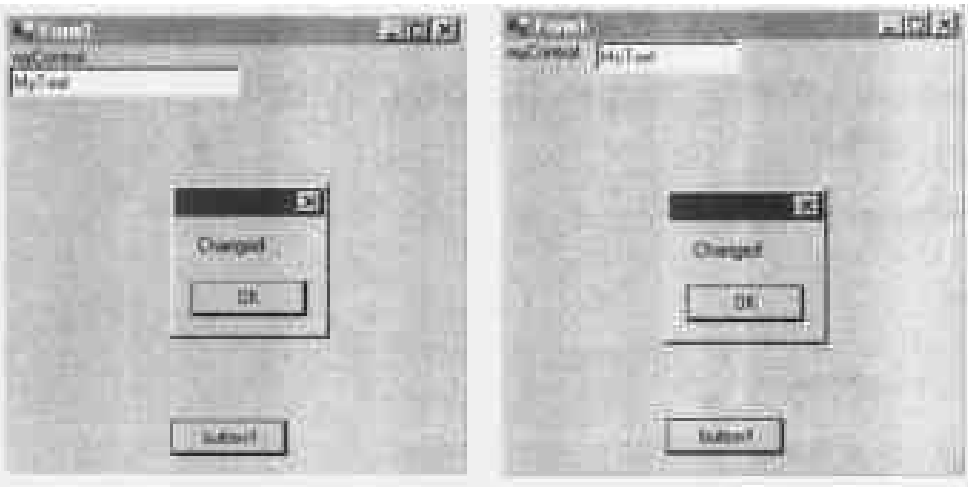


图 14-24

与往常一样，完整的工程可以从 Wrox Web 站点上下载。

这个范例到此就完成了。还可以细化它，这留给读者作为练习。

14.5　小结

本章继续第 13 章的内容，介绍了 MainMenu 和 Toolbar 控件。我们讨论了如何创建 MDI 和 SDI 应用程序，如何在这些应用程序中使用菜单和工具栏。接着论述如何创建自己的控件，设计该控件的属性、用户界面和事件。第 15 继续讨论 Windows Forms，介绍一种特殊类型的窗体 Windows Common Dialog。

14.6　练习

以 LabelTextbox 范例为基础，创建一个新属性 MaxLength，以存储文本框中可以输入的最大字符数，然后创建两个新事件 MaxLengthChanged 和 MaxLengthReached。MaxLengthChanged 事件应在修改 MaxLength 属性时引发，MaxLengthReached 事件应在用户输入一个字符后，使文本框中的文本长度等于 MaxLength 属性值时引发。

第15章　使用对话框

第 14 章介绍了如何实现菜单、工具栏、SDI 和 MDI 窗体等，如何显示简单的信息框，从用户处获取信息，还介绍了如何创建较复杂的定制对话框，要求用户输入特定的信息。但如果可能，应使用预先编写好的对话框类，而不是定制的对话框来执行命令操作，例如打开和保存文件、打印等。这样做，不仅需要编写的代码量比较少，而且还可以使用我们熟悉的标准 Windows 对话框。.NET Framework 提供了与 Windows 对话框相关的类，可以打开和保存文件、访问打印机、选择颜色和字体等。使用这些对话框而不是定制对话框，用户就不需要学习复杂的方法，从头开始编写这些功能需要的代码。

本章将讨论如何使用这些标准对话框类，主要内容如下：

- 使用 OpenFileDialog 和 SaveFileDialog 类。
- 学习.NET 打印类层次结构，并使用 PrintDialog, PageSetupDialog 和 PrintPreviewDialog 类进行打印和打印预览。
- 介绍改变字体和颜色的 FontDialog 和 ColorDialog。

15.1　对话框

对话框是一个在另一个窗口中显示的窗口。在继续执行程序流之前，可以让用户输入一些数据。通用对话框常常用于从用户处获取一般性的信息，它是 Windows 操作系统的一部分。

在.NET Framework 中，有如图 15-1 所示的对话框类。

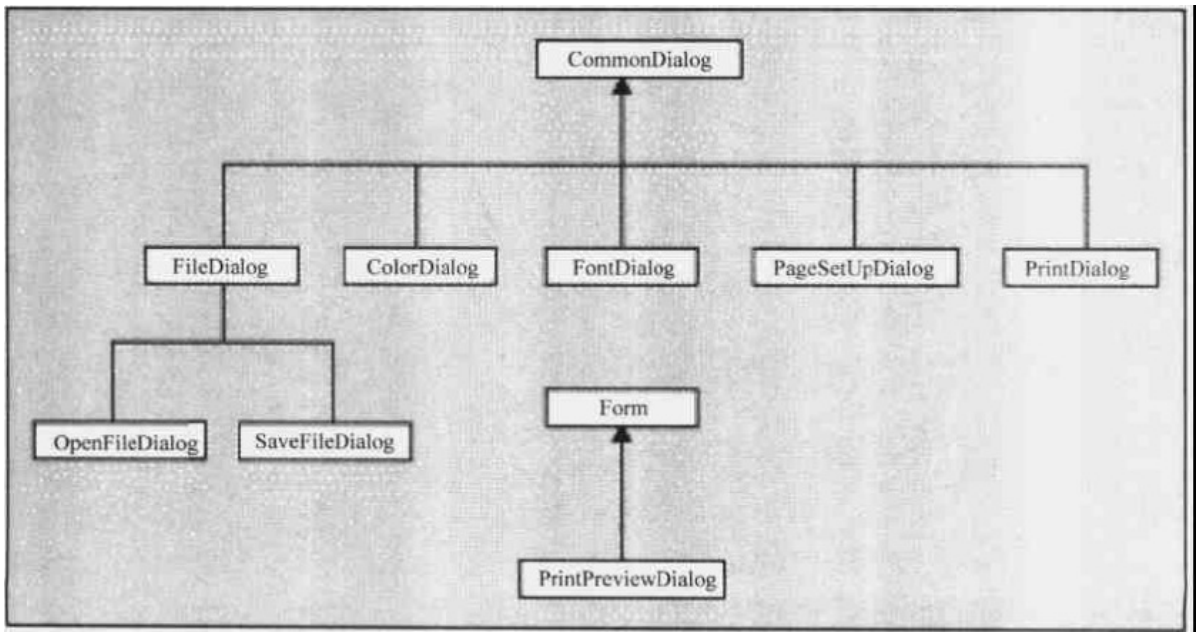


图　15-1

除了 PrintPreviewDialog 外，其他对话框类都派生于抽象基类 CommonDialog，这个基类的方法可以管理 Windows 通用对话框。

CommonDialog 定义了表 15-1 所示的方法和事件，它们对于所有的通用对话框类来说都是通用的。

表　15-1

公共实例方法和事件	描　　述
ShowDialog()	这个方法由派生的类执行，显示一个通用对话框
Reset()	每个派生的对话框类都执行 Reset()方法，把对话框类的所有属性设置为它们的默认值
HelpRequest	当用户单击通用对话框上的 Help 按钮时，就引发该事件

所有这些对话框类都封装了一个 Windows 通用对话框，使该对话框可用于.NET 应用程序。PrintPreviewDialog 是一个例外，因为它把自己的元素添加到 Windows Form 上，控制打印预览，所以它并不是一个真正的对话框。OpenFileDialog 和 SaveFileDialog 类派生于抽象基类 FileDialog，该基类提供了对于打开和关闭文件对话框来说都相同的文件功能。

下面简要介绍一下不同的对话框所适用的场合：

- 要让用户选择和浏览要打开的文件，应使用 OpenFileDialog。这个对话框可以配置为只允许选择一个文件，或可以选择多个文件。
- 使用 SaveFileDialog，用户可以为要保存的文件指定一个文件名和浏览的路径。
- PrintDialog 用于选择一个打印机，并设置打印选项。
- 要配置页面的边距，通常使用 PageSetupDialog。
- PrintPreviewDialog 是在屏幕上进行打印预览的一种方法，并有一些选项如缩放。
- FontDialog 列出了所有已安装的 Windows 字体、样式和字号，以及各字体的预览效果，以便选择字体。
- ColorDialog 类用于选择颜色。

注意：

我见过一些公司开发的应用程序，其中不仅没有重用通用对话框，而且也没有使用建立定制对话框的样式规则。这些对话框的开发导致了功能的不一致，其中一些按钮和列表框可以在其他地方找到，例如 OK 和 Cancel 按钮被倒置。

有时这种不一致也会在一个应用程序中出现。这会使用户感到沮丧，延长了完成任务的时间。

一定要使建立和使用的对话框保持一致！使用通用对话框，很容易达到这种一致性，在用户熟悉的许多应用程序中，都使用了通用对话框。

15.2 如何使用对话框

CommonDialog 是对话框类的基类，所以所有的对话框类的使用方式都是类似的。公共实例方法是 ShowDialog() 和 Reset()。ShowDialog()调用受保护的 RunDialog()实例方法来显示对话框，并利用用户与对话框交互操作的信息，最终返回一个 DialogResult 实例。另一方面，Reset()把对话框类的属性设置为它们的默认值。

下面的代码段是一个说明如何使用对话框类的范例。后面将详细介绍每一步，现在先概述使用对话框的整体概念。

在下面的代码段中，可以看出：

- 首先创建一个对话框类的新实例。
- 接着必须把一些属性设置为启用/禁用可选功能，并设置对话框的状态。在本例中，把 Title 属性设置为“Sample”，ShowReadOnly 属性设置为 true。
- 调用 ShowDialog()方法，显示对话框，等待并响应用户的输入。
- 如果用户按下 OK 按钮，就关闭对话框，比较对话框的结果和 DialogResult.OK 之后，通过查询特定的属性值，获取用户输入的值。在本例中，把 FileName 属性的值存储在 fileName 变量中：

```
OpenFileDialog dlg = new OpenFileDialog();
dlg.Title = "Sample";
dlg.ShowReadOnly = true;

if (dlg.ShowDialog() == DialogResult.OK)
{
    string fileName = dlg.FileName;
}
```

就是这么简单！当然，每个对话框都有自己的可配置选项，详见后面的内容。

如果在 Windows Forms 应用程序中使用对话框，就要比这里编写几行代码更简单。Windows Forms 设计器会创建代码以实例化一个新实例，设置属性值。我们仅需要调用 ShowDialog()，输入改变的值即可，如下面所述。

15.3 文件对话框

使用文件对话框，用户可以选择一个驱动器，浏览文件系统，选择一个文件。我们仅需要用户输入一个文件名即可。

使用 OpenFileDialog，可以选择要打开的文件名，而使用 SaveFileDialog，可以指定要保存的文件名。这些对话框类都是非常简单的，因为它们派生于同一个抽象基类，但每个类都有一些独有的属性。本节首先介绍 OpenFileDialog 的功能，再介绍它与 SaveFileDialog 的区别。我们将开发一个使用这两个类的样例应用程序。

15.3.1　OpenFileDialog

OpenFileDialog 类可以选择要打开的文件。从上面的范例可以看出，OpenFileDialog 类的新实例是在调用 ShowDialog()方法前创建的。

```
OpenFileDialog dlg = new OpenFileDialog();
dlg.ShowDialog();
```

运行这两行代码，得到如图 15-2 所示的对话框。



图　15-2

可以看出，我们在调用 ShowDialog()之前设置了这个类的属性，而该方法改变了对话框的操作方式和外观，或限制可以打开的文件。下一节将介绍可以进行的修改。

注意：

在控制台应用程序中使用 OpenFileDialog 时，必须引用装配件 System.Windows.Forms，且必须包含名称空间 System.Windows.Forms。而对于在 Visual Studio .NET 中编写的 Windows Forms 应用程序，这已经由应用程序向导完成了。

1. 对话框的标题

OpenFileDialog 的默认标题是 Open。设置 Title 属性可以改变对话框的标题。Open 并不总是最好的标题，例如在应用程序中，要分析日志文件，查看一些信息，对它们进行计算，或者获取文件的大小，在执行完需要的处理后，直接关闭文件。此时，文件就不是打开的，所以使用 Analyze Files 标题就比较好。Visual Studio .NET 本身为文件打开对话框就设置了不同的标题，以区分打开的文件类型：Open Project, Open File, Open Solution 和 Open File from Web。

下面的代码说明了如何设置不同的标题：

```
OpenFileDialog dlg = new OpenFileDialog();
dlg.Title = "Open File";
dlg.ShowDialog();
```

2. 指定目录

在默认情况下，对话框将打开用户在上一次运行应用程序时打开的目录，并显示这个目录中的文件。设置 InitialDirectory 属性可以改变这种设置。InitialDirectory 的默认值是一个空字符串，表示用户的“我的文档”目录，第一次在应用程序中使用这个对话框时，就显示“我的文档”目录下的文件。第二次打开该对话框时，显示的目录就与上一次打开的文件所在目录相同。OpenFileDialog 调用的 Windows 通用对话框使用注册表来定位上一次打开的文件名。

在调用 ShowDialog()前，把 InitialDirectory 属性设置为一个目录字符串，就可以改变这种设置方式。在应用程序中，不要使用硬编码的目录字符串，因为这个目录可能在用户的系统上不存在。为了获取特定的系统文件夹，可以使用 System.Environment 类的静态方法 GetFolderPath()。该方法接受一个 Environment.SpecialFolder 枚举，其中可以定义要返回路径的哪个系统目录。

在下面的代码范例中，把 InitialDirectory 设置为通用用户目录，把该目录作为模板的目录：

```
string dir = Environment.GetFolderPath(Environment.SpecialFolder.Templates);
dlg.InitialDirectory = dir;
```

3. 设置文件过滤器

文件过滤器定义了用户可以选择打开的文件类型。简单的过滤器字符串如下所示：

```
Text Documents (*.txt)|*.txt|All Files|*.*
```

过滤器用于显示 Files of type：列表框中的选项，如图 15-3 所示。Microsoft WordPad 显示了 3 个选项。



图 15-3

过滤器可以分为多个部分，每个部分用一个竖杠字符(|)分隔开。两个字符串总是放在一起，这样部分的个数总是一个偶数。第一个字符串定义要在列表框中显示的文本，第二个字符串用于指定要在对话框中显示的文件扩展名。用 Filter 属性设置过滤器字符串，如下面的代码所示：

```
dlg.Filter = "Text documents (*.txt)|*.txt|All Files|*.*";
```

设置错误的 Filter 值会产生一个运行异常 System.ArgumentException 和错误信息“The provided filter string is invalid”。过滤器前后也不允许有空格。

FilterIndex 属性指定列表框中的默认选项。在 WordPad 中，默认值是 Rich Text Format (*.rtf)，在屏幕图 15-3 中为突出显示状态，它们可能很难阅读，但如果打开 WordPad，就可以看到它们。如果可以选择多个文件类型，就可以把 FilterIndex 设置为默认的文件类型。注意 FilterIndex 是

基于 1 的。

4. 有效性验证

在试图打开一个文件之前，OpenFileDialog 可以自动执行对文件的有效性验证。使用 ValidateNames，就会检查用户输入的文件名，看看它是否是一个有效的 Windows 文件名。在对话框中输入无效的文件名，并单击 OK 按钮，会显示图 15-4 所示的对话框，在更正文件名之后，才能退出 OpenFileDialog。无效的文件名包含无效字符，例如\\, /或 :。

输入无效的文件名 xxx\\xxx 会得到如图 15-4 所示的错误信息。

CheckFileExists 和 CheckPathExists 是有效性验证的另外两个属性。使用 CheckPathExists 会验证路径的有效性，而 CheckFileExists 会验证文件的有效性。如果文件不存在，单击 OK 按钮后，会显示如图 15-5 所示的对话框。

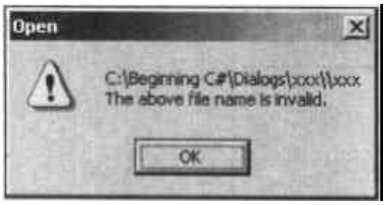


图　15-4

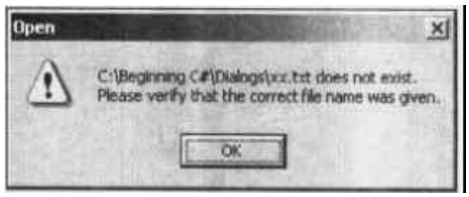


图　15-5

这 3 个属性的默认值都是 true，所以有效性验证会自动进行。

5. 帮助

OpenFileDialog 类提供了一个帮助按钮，在默认状态下，该按钮是不可见的。把 ShowHelp 属性设置为 true，就会使该按钮可见，也可以给 HelpRequest 事件添加一个事件处理程序，用以给用户显示帮助信息。

6. 结果

OpenFileDialog 类的 ShowDialog()方法返回一个 DialogResult 枚举。该枚举定义了成员 Abort、Cancel、Ignore、No、None、OK、Retry 和 Yes。None 是默认值，只要用户没有关闭对话框，就设置这个值。根据单击的按钮，将返回对应的结果，对于 OpenFileDialog，则只返回 DialogResult.OK 和 DialogResult.Cancel。

如果用户单击 OK 按钮，选中的文件名可以使用 FileName 属性来访问。如果用户取消了对话框，FileName 就是一个空字符串。如果 Multiselect 属性设置为 true，用户可以选择多个文件，通过访问 FileName 属性，可以得到所有选中的文件名，返回一个字符串数组。

下面的代码说明了如何从 OpenFileDialog 中获取多个文件名：

```
OpenFileDialog dlg = new OpenFileDialog();
dlg.Multiselect = true;

if (dlg.ShowDialog() == DialogResult.OK)
{
    foreach (string s in dlg.FileNames)
    {
```

```
            Console.WriteLine(s);
        }
    }
```

ShowDialog()方法可以打开对话框。设置 Multiselect 属性为 true，用户可以选择多个文件，如果一切就绪，单击对话框中的OK 按钮，就会关闭该对话框，返回DialogResult.OK。使用 foreach 语句，可以遍历从 FileNames 属性中返回的字符串数组中的所有字符串，显示每个选中的文件。

7. OpenFileDialog 的属性

总之，在图 15-6 中，列出了 OpenFileDialog 和所有可以打开 Windows 元素的属性。在该图中，可以看出哪些属性会影响哪些用户界面元素。

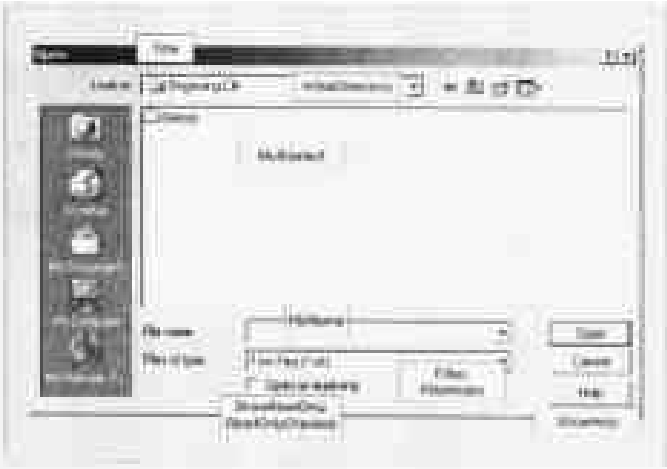


图 15-6

试试看：创建一个 Windows 应用程序

(1) 要使用打开文件和保存文件对话框，下面创建一个简单的 Windows 应用程序 SimpleEditor，如图 15-7 所示。

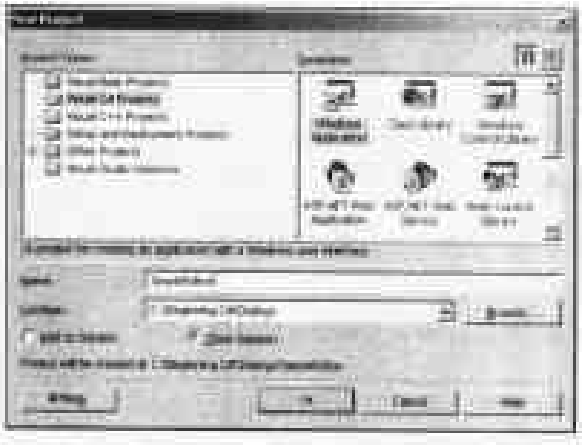


图 15-7

(2) 把生成的文件 Form1.cs 重命名为 SimpleForm.cs，类 Form1 重命名为 SimpleEditorForm，把名称空间改为 Wrox.Editor。

在修改类的名称时，还必须改变 Main()的执行代码，反映名称的改变，因为这不是通过设

置 Form 类的 Name 属性而自动改变的：

```
[STAThread]
static void Main()
{
    Application.Run(new SimpleEditorForm());
}
```

(3) 把窗体的 Text 属性设置为“Simple Editor”。这里要用一个多行文本框来读取和修改文件的数据，所以从工具箱中把一个 TextBox 添加到 Windows Forms 设计器中。该文本框应是多行的，应覆盖应用程序的整个区域，所以把这些属性设置为指定的值，如表 15-2 所示。

表　15-2

属　性	值
(Name)	TextBoxEdit
Text	<空>
Multiline	True
Dock	Fill
ScrollBars	Both
AcceptsReturn	True
AcceptsTab	True

(4) 接着，给应用程序添加一个 MainMenu，该主菜单应有一个 File 菜单项，其中包含子菜单 New, Open, Save 和 Save As，如图 15-8 所示。

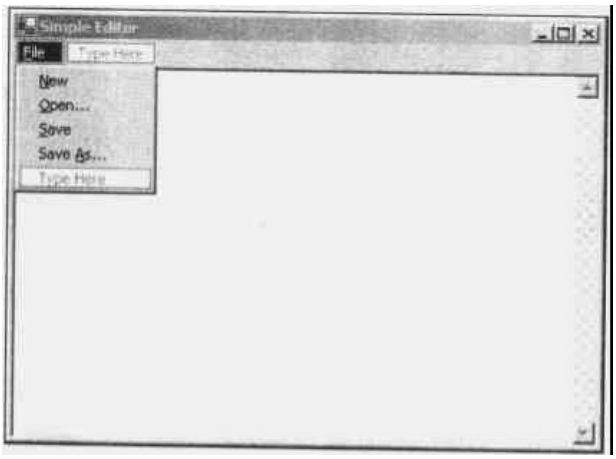


图　15-8

为了提高代码的可读性，把名称改为表 15-3 中的值。Text 属性显示在菜单中。&字符指定其后带有下划线的字符，如果使用键盘访问该菜单，就应使用<ALT>+该下划线字符。Open 和 Save As 菜单项的 Text 属性中的...表示用户在执行操作前要打开一个对话框，输入一些数据。选择 File, New 和 Save 按钮，就会执行操作，不需要额外的操作。

表 15-3

菜单项名称	文 本
MiFile	&File
MiFileNew	&New
MiFileOpen	&Open...
MiFileSave	&Save
MiFileSaveAs	Save &As...

注意：

下面是添加子菜单项的方法：只要在主菜单项中输入文本&File，就会在它的下面打开一个子菜单，在 Type Here 方框中就可以输入子菜单项。

注意：

在第 14 章“Windows forms 的高级功能”中详细讨论了菜单。

(5) 菜单项&New 的处理程序应通过调用 TextBox 的 Clear()方法清除文本框的数据：

```
private void miFileNew_Click(object sender, System.EventArgs e)
{
    fileName = "Untitled";
    textBoxEdit.Clear();
}
```

(6) fileName 成员变量应再次被设置为“Untitled”。必须在 SimpleEditorForm 类中声明和初始化这个成员变量：

```
private string fileName = "Untitled";
```

在 Open 菜单中，要显示打开文件对话框，其步骤如下所示。

试试看：读取文件

在 SimpleEditor 范例中，当启动应用程序时，可以把文件名作为一个参数，传递过来的文件名应用于打开文件，并在文本框中显示它。

(1) 修改 Main()方法中的执行代码，传递一个参数：

```
[STAThread]
static void Main(string[] args)
{
    string fileName = null;
    if (args.Length != 0)
        fileName = args[0];
    Application.Run(new SimpleEditorForm(fileName));
}
```

(2) 现在还必须修改 SimpleEditorForm 构造函数中的执行代码，以使用一个字符串：

```
public SimpleEditorForm(string fileName)
{
    //
    // Required for Windows Form Designer support
    //
    InitializeComponent();

    if (fileName != null)
    {
        this.fileName = fileName;
        OpenFile();
    }
}
```

(3) 还必须执行 OpenFile() 方法，打开一个文件，并用文件中的数据填充文本框：

```
protected void OpenFile()
{
    try
    {
        using (StreamReader reader = File.OpenText(fileName))
        {
            textBoxEdit.Clear();
            textBoxEdit.Text = reader.ReadToEnd();
        }
    }
    catch (IOException ex)
    {
        MessageBox.Show(ex.Message, "Simple Editor",
            MessageBoxButtons.OK, MessageBoxIcon.Exclamation);
    }
}
```

注意：

这里使用 StreamReader 和 File 类读取文件。要使用这两个类，而不必每次增加名称空间，必须在程序的开头添加 using System.IO;语句。

(4) 现在就可以在命令行上传递一个文件名，运行应用程序。该文件会立即打开，显示在文本框中，如图 15-9 所示。

代码的说明

把 string[]作为参数添加到 Main()方法中，在启动应用程序时，就可以使用用户提供的任意命令行参数：



图 15-9

```
static void Main(string[] args)
```

在 Main()方法中，使用 Length 属性检查是否传递了参数。如果传递的第一个参数设置为 fileName 变量，就使用该变量把参数传送给 SimpleEditorForm 的构造函数：

```
{
    string fileName = null;
    if (args.Length != 0)
        fileName = args[0];
    Application.Run(new SimpleEditorForm(fileName));
}
```

注意：

如第 7 章所述，在 Visual Studio .NET 中，还可以为调试程序定义命令行参数。在 Solution Explorer 中，选择一个工程，再选择 Project | Properties 菜单。如果选择了左边树中的 Configuration Properties | Debugging，就会弹出图 15-10 所示的对话框，在该对话框中，就可以输入命令行参数。

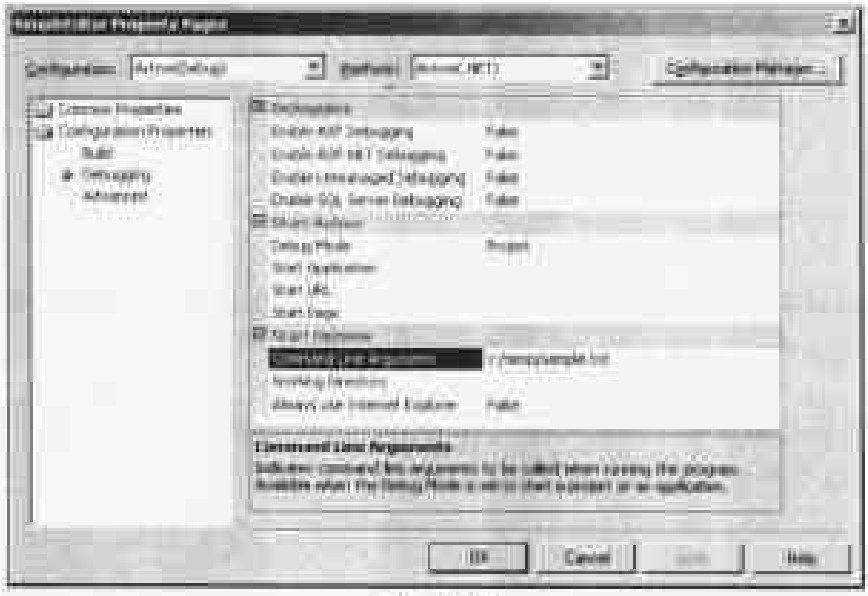


图 15-10

在 SimpleEditorForm 构造函数中，检查一下文件名变量是否已经设置了一个值。如果是，就设置成员变量 fileName，并调用 OpenFile()方法，打开文件。这里使用一个 OpenFile()方法，没有编写打开文件的调用代码，直接在类的构造函数中填充文本框，因为 OpenFile()可以在程序的其他部分再次使用。

```
if (fileName != null)
{
    this.fileName = fileName;
    OpenFile();
}
```

在 OpenFile()方法中，使用了 File 类的静态方法 OpenText()打开一个文件，并返回一个 StreamReader。接着使用 StreamReader，通过 ReadToEnd()读取文件，把文本加载为一个字符串，传递给 TextBox 对象。StreamReader 应在使用完受管制和未托管资源后关闭。这是使用 using 语句完成的。Using 在代码块的末尾调用 Dispose()。StreamReader 类的 Dispose()执行代码调用 Close()关闭文件：

```
using (StreamReader reader = File.OpenText(fileName))
{
    textBoxEdit.Clear();
    textBoxEdit.Text = reader.ReadToEnd();
}
```

因为文件操作很容易产生异常，如果用户没有文件的正确访问许可，或者文件本身不存在，代码就会封装到一个 try 块中。在产生 IO 异常时，会显示一个信息框，通知用户出了问题，但应用程序仍在运行：

```
try
{
    //...
}
catch (IOException ex)
{
    MessageBox.Show(ex.Message, "Simple Editor",
        MessageBoxButtons.OK, MessageBoxIcon.Exclamation);
}
```

在启动应用程序时，如果输入一个不存在的文件名，就会显示如图 15-11 所示的信息框。



图　15-11

试试看：添加一个打开文件对话框

现在就可以启动应用程序，输入一个文件名，用简单的编辑器读取文件了，当然我们更愿意使用一个通用对话框类，下面就把通用对话框类添加到应用程序中。

在工具箱的 Windows Forms 类别中，可以找到 OpenFileDialog 组件。把这个组件添加到 Windows Forms 设计器中。这里仅修改了 3 个属性：dlgOpen 的实例名，把 Filter 属性设置为下列字符串，并把 FilterIndex 的属性设置为 2，使 Wrox 文档成为默认选项：

```
Text Documents (*.txt)|*.txt|Wrox Documents (*.wroxtext)|*.wroxtext|All Files|*.*.
```

代码的说明

把 OpenFileDialog 组件添加到 Windows Forms 设计器中，将把一个新的私有成员添加到 SimpleEditorForm 类中：

```
public class SimpleEditorForm : System.Windows.Forms.Form
{
    private System.Windows.Forms.TextBox textBoxEdit;
    private System.Windows.Forms.MenuItem miFile;
    private System.Windows.Forms.MenuItem miFileNew;
    private System.Windows.Forms.MenuItem miFileOpen;
    private System.Windows.Forms.MenuItem miFileSave;
    private System.Windows.Forms.MenuItem miFileSaveAs;
    private System.Windows.Forms.MainMenu mainMenu;
    private System.Windows.Forms.OpenFileDialog dlgOpenFile;
```

在 Windows Forms 的设计器代码中，在 InitializeComponent()中创建这个 OpenFileDialog 类的一个新实例，并设置了指定的属性。为了在代码编辑器中查看下面的代码，应单击 Windows Forms Designer generated code 一行上的+字符，再单击 private void InitializeComponent()一行上的+字符：

```
    private void InitializeComponent()
    {
        this.textBoxEdit = new System.Windows.Forms.TextBox();
        this.mainMenu = new System.Windows.Forms.MainMenu();
        this.miFile = new System.Windows.Forms.MenuItem();
        this.miFileNew = new System.Windows.Forms.MenuItem();
        this.miFileOpen = new System.Windows.Forms.MenuItem();
        this.miFileSave = new System.Windows.Forms.MenuItem();
        this.miFileSaveAs = new System.Windows.Forms.MenuItem();
        this.dlgOpenFile = new System.Windows.Forms.OpenFileDialog();
        // ...
        //
        // dlgOpenFile
        //
        this.dlgOpenFile.Filter =
```

```
        "Text Documents (*.txt)|*.txt|Wrox Documents (*.wroxtext)|*.wroxtext|All Files|*.*";
    this.dlgOpenFile.FilterIndex = 2;
```

有了 Windows Forms 设计器的支持，就创建了 OpenFileDialog 的一个新实例，并设置了属性。下面要显示该对话框。

试试看：显示 OpenFileDialog

给 Open 菜单项的单击事件添加处理程序，用下面的代码显示对话框，读取选中的文件：

```
private void miFileOpen_Click(object sender, System.EventArgs e)
{
    if (dlgOpenFile.ShowDialog() == DialogResult.OK)
    {
        fileName = dlgOpenFile.FileName;
        OpenFile();
    }
}
```

代码的说明

ShowDialog()方法显示打开文件对话框，返回用户单击的按钮。如果单击按下了除 OK 外的其他按钮，我们就什么也不做。这就是在 if 语句中要检查 DialogResult.OK 的原因。如果用户取消了对话框，也是什么也不做：

```
if (dlgOpenFile.ShowDialog() == DialogResult.OK)
{
```

接着，通过访问 OpenFileDialog 类的 FileName 属性，把成员变量 fileName 设置为这个值，获取选中的文件名。OpenFile()方法就使用这个值。还可以使用 OpenFileDialog 类，直接打开文件，具体方法是调用 dlgOpenFile.OpenFile()，该方法已经返回一个 Stream 对象，但因为前面使用 OpenFile()方法打开和读取文件，所以我们使用它：

```
    fileName = dlgOpenFile.FileName;
    OpenFile();
```

试试看：启动简单编辑器

现在可以启动简单编辑器程序了。此时只有 New 和 Open...菜单项可以使用。Save 和 Save As...的执行在下一节介绍。

选择菜单项 File | Open...，如图 15-12 所示，OpenFileDialog 就会出现，从中可以选择一个文件。假定当前没有扩展名为.wroxtext 的文件。到现在为止，我们还不能保存文件，所以在对话框编辑器中选择另一个文件类型，打开一个文件，或者把一个文本文件复制到扩展名为.wroxtext 的文件中，如图 15-13 所示。

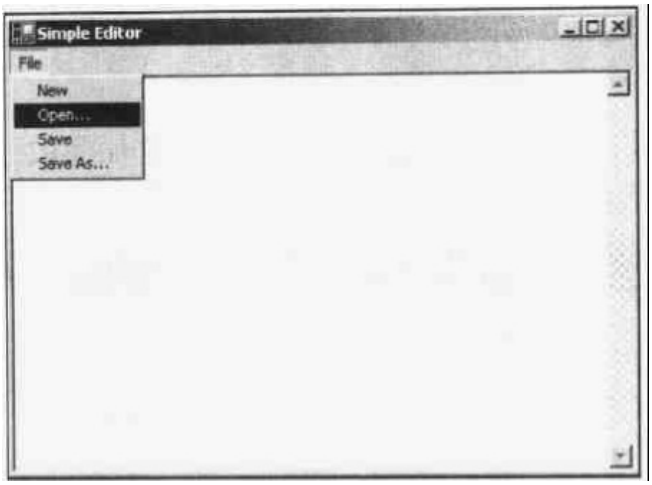


图　15-12



图　15-13

选择一个文本文件，单击 Open 按钮，文本就显示在对话框的文本框中。

这里选择了一个本地系统中的样例文本文件 GlobalKnowledge.txt，如图 15-14 所示。

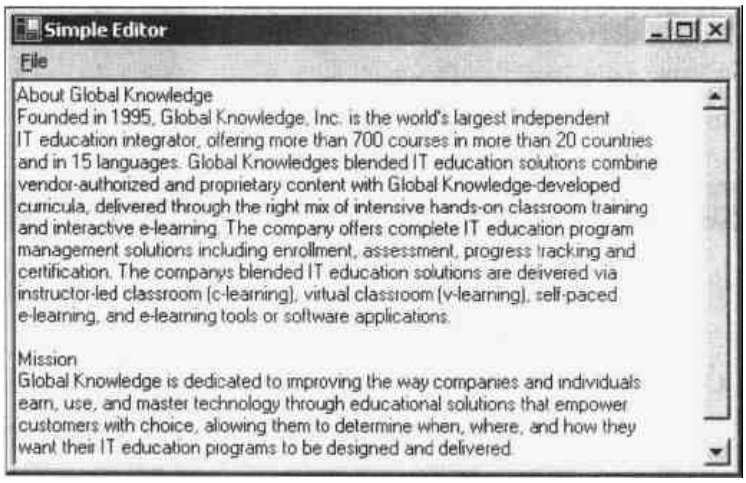


图　15-14

此时可以读取现有的文件。也可以创建新文件，修改现有文件，下面使用 SaveFileDialog

来完成这些任务。

15.3.2　SaveFileDialog

SaveFileDialog 类非常类似于 OpenFileDialog，它们有许多相同的属性。这里不讨论与打开对话框使用方式相同的属性，而主要论述保存对话框特有的属性，以及命令属性应用的区别。

1. 标题

使用 Title 属性，可以设置对话框的标题，这与 OpenFileDialog 类似。如果没有设置标题，默认标题是 Save As。

2. 文件扩展名

文件扩展名用于把文件和应用程序关联起来。最好给文件添加一个扩展名，否则 Windows 就不知道打开该文件时应使用哪个应用程序，最终很可能用户也会忘记应使用什么应用程序。

AddExtension 是一个布尔属性，它定义了文件扩展名是否应自动添加到用户输入的文件名上。如果用户已经输入了一个文件扩展名，就不会添加其他扩展名了。所以如果用户输入了文件名 test，就保存文件 test.txt，如果输入了 test.txt，保存的文件仍是 test.txt，而不是 test.txt.txt。

如果用户没有输入文件扩展名，就使用 DefaultExt 属性设置的文件扩展名。如果这个属性为空，就使用当前选择的 Filter 中定义的文件扩展名。如果设置了 Filter 和 DefaultExt，则不论 Filter 是什么，都使用 DefaultExt。

3. 有效性验证

为了自动验证文件名的有效性，应使用属性 ValidateNames, CheckFileExists 和 CheckPathExists，这一点与 OpenFileDialog 相同。OpenFileDialog 和 SaveFileDialog 的区别是对于 SaveFileDialog，CheckFileExists 的默认值是 false，表示可以提供新文件名，进行保存。

4. 覆盖已有的文件

可以看出，文件名的有效性验证类似于 OpenFileDialog。但是，对于 SaveFileDialog 类来说，要进行更多的检查，设置更多的属性。首先，如果 CreatePrompt 属性设置为 true，就会询问用户是否要创建一个新文件。如果 OverwritePrompt 属性设置为 true，就会询问用户是否真的想覆盖已有的文件。OverwritePrompt 的默认设置为 true，而 CreatePrompt 的默认设置为 false。使用这些设置，如果用户要保存已有的文件，就显示图 15-15 所示的对话框。

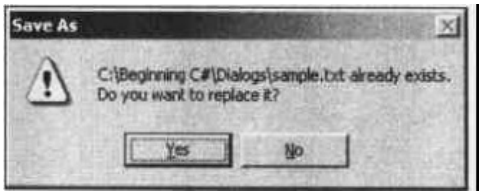


图　15-15

5. ileSaveDialog 属性

与前面的对话框一样，图 15-16 总结了 FileSaveDialog 的所有属性：

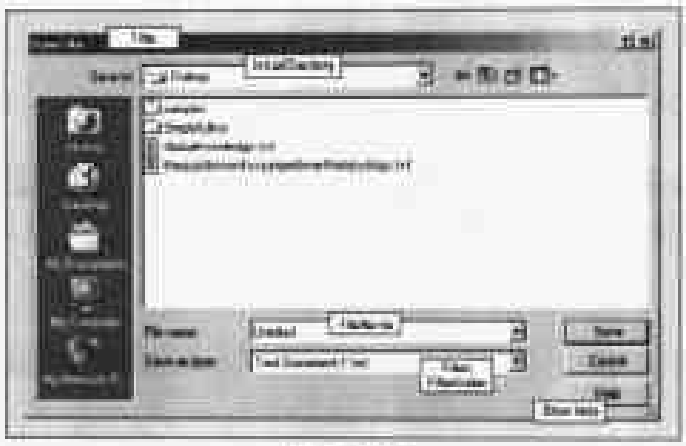


图 15-15

试试看：添加 SaveFileDialog

(1) 以与 OpenFileDialog 类似的方式，把 SaveFileDialog 添加到 Windows Forms 设计器中：从工具箱中选择 SaveFileDialog 组件，把它拖放到窗体上。把其名称改为 dlgSaveFile，把 FileName 设置为 Untitled，FilterIndex 设置为 2，Filter 属性设置为下述字符串，就像前面处理 OpenFileDialog 那样。因为这允许这个编辑器存储扩展名为.txt 和.wroxtext 的文件，所以应删除 *.* ：

```
Text Document (*.txt)|*.txt|Wrox Documents (*.wroxtext)|*.wroxtext.
```

(2) 给 Save As 菜单项的 Click 事件添加处理程序，并添加下列代码。在其代码中，用 ShowDialog()方法显示 SaveFileDialog。与 OpenFileDialog 一样，我们只对用户单击 OK 按钮的结果感兴趣。调用 SaveFile()方法，把文件存储到磁盘上，这个方法必须在下一步中执行：

```
private void miFileSaveAs_Click(object sender, System.EventArgs e)
{
    if (dlgSaveFile.ShowDialog() == DialogResult.OK)
    {
        fileName = dlgSaveFile.FileName;
        SaveFile();
    }
}
```

(3) 在文件中添加 SaveFile()方法，如下所示：

```
protected void SaveFile()
{
    try
    {
        Stream stream = File.OpenWrite(fileName);
        using (StreamWriter writer = new StreamWriter(stream))
        {
            writer.Write(textBoxEdit.Text);
        }
```

```
        }
        catch (IOException ex)
        {
            MessageBox.Show(ex.Message, "Simple Editor",
                MessageBoxButtons.OK, MessageBoxIcon.Exclamation);
        }
    }
```

与 OpenFile()方法类似，使用 File 类打开文件，但现在打开文件的目的是用 OpenWrite()进行写入访问。OpenWrite()返回一个 Stream 对象，该对象传递给 StreamWriter 类的构造函数。StreamWriter 类的 Write()方法把 textBox 中的所有数据写到文件中。在 using 块的末尾，关闭 StreamWriter。Stream 对象与 writer 关联在一起，所以 stream 也被关闭，不需要为 stream 编写额外的 Close()。

用于 IO 的类详见第 20 章。

(4) 建立了工程后，就可以使用 Visual Studio .NET 的 Debug | Start 菜单项启动应用程序。在文本框中写入一些文本，选择菜单 File | Save As...，如图 15-17 所示。

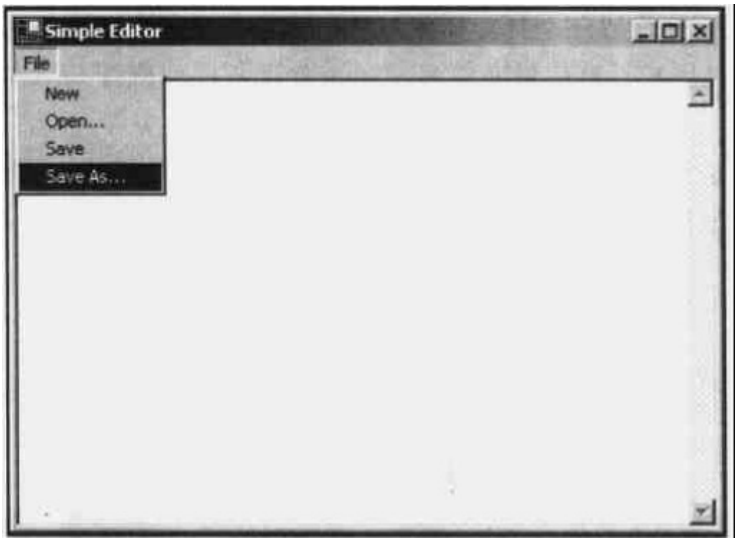


图　15-17

弹出如图 15-18 所示的 SaveFileDialog。现在可以保存文件，再打开它，进行更多的修改。



图 15-18

试试看：执行 Save 处理程序

可以执行 Save As，但此时不能使用简单的 Save。给 Save 菜单项的 Click 事件添加一个处理程序，并添加如下代码：

```
private void miFileSave_Click(object sender, System.EventArgs e)
{
    if (fileName == "Untitled")
    {
        miFileSaveAs_Click(sender, e);
    }
    else
    {
        SaveFile();
    }
}
```

代码的说明

对于 Save 菜单，文件应在不打开任何对话框的情况下保存。这个规则有一个例外，如果创建了一个新文档，且用户没有提供文件名，Save 处理程序就应像 Save As 处理程序那样，显示保存文件对话框。

使用 fileName 成员变量，可以检查文件是否打开或者在创建新文档后，文件名是否仍设置为初始值 Untitled。如果 if 语句返回 true，就调用 miFileSaveAs_Click()处理程序，执行与 Save As 菜单相同的操作。

注意：

当文件已经打开，且用户选择了 Save 时，执行线程会进入 else 块。此时可以使用以前执行的 SaveFile()方法。

试试看：设置窗体的标题

使用 Notepad, Word 和其他 Windows 应用程序，当前编辑的文件名会显示在应用程序的标题中。这个范例也添加这个功能。

创建一个新的成员函数 SetFormTitle()，添加如下执行代码：

```
protected void SetFormTitle()
{
    FileInfo fileinfo = new FileInfo (fileName);
    this.Text = fileinfo.Name + " - Simple Editor";
}
```

FileInfo 类用于获取文件名，但不包括存储在 fileName 变量中的路径。FileInfo 类和 StreamWriter 类在第 20 章介绍。

设置了成员变量 fileName 后，在 miFileNew_Click()、miFileOpen_Click()和 miFileSaveAs_

Click()处理程序中给这个方法添加一个调用，如下面的代码段所示：

```
 private void miFileNew_Click(object sender, System.EventArgs e)
{
    fileName = "Untitled";
    SetFormTitle();
    textBoxEdit.Clear();
}

private void miFileOpen_Click(object sender, System.EventArgs e)
{
    if (dlgOpenFile.ShowDialog() == DialogResult.OK)
    {
        fileName = dlgOpenFile.FileName;
        SetFormTitle();
        OpenFile();
    }
}

private void miFileSaveAs_Click(object sender, System.EventArgs e)
{
    if (dlgSaveFile.ShowDialog() == DialogResult.OK)
    {
        fileName = dlgSaveFile.FileName;
        SetFormTitle();
        SaveFile();
    }
}
```

代码的说明

每次修改文件名时，窗体的 text 属性都会改为文件名后跟应用程序名。

现在启动应用程序，得到如图 15-19 所示的结果。这里在编辑文件 sample.txt，这些信息会显示在窗体的标题中。

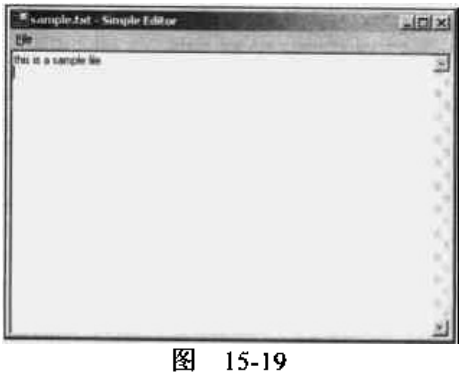


图　15-19

这就完成了一个简单的编辑器：可以打开、创建和保存文件。工作全部完成了吗？没有！因为无纸办公还不存在，我们还应添加一些打印功能。

15.4　打印

对于打印，需要考虑打印机的选择、页面设置和如何打印多页面。System.Drawing.Printing 名称空间中的.NET 类可以帮助我们完成这些任务。

PrintDialog 类可以选择打印机，在介绍它之前，必须先简要讨论一下.Net 如何进行打印。打印的基类是 PrintDocument，它的一个方法 Print()会启动一系列调用，最后调用 OnPrintPage()，把输出结果传送给打印机，但是，在深入讨论如何执行打印代码前，先详细论述.NET 打印类。

15.4.1　打印结构

图 15-20 以图形方式显示了打印结构的主要部分，以及类和一些属性及方法之间的关系。

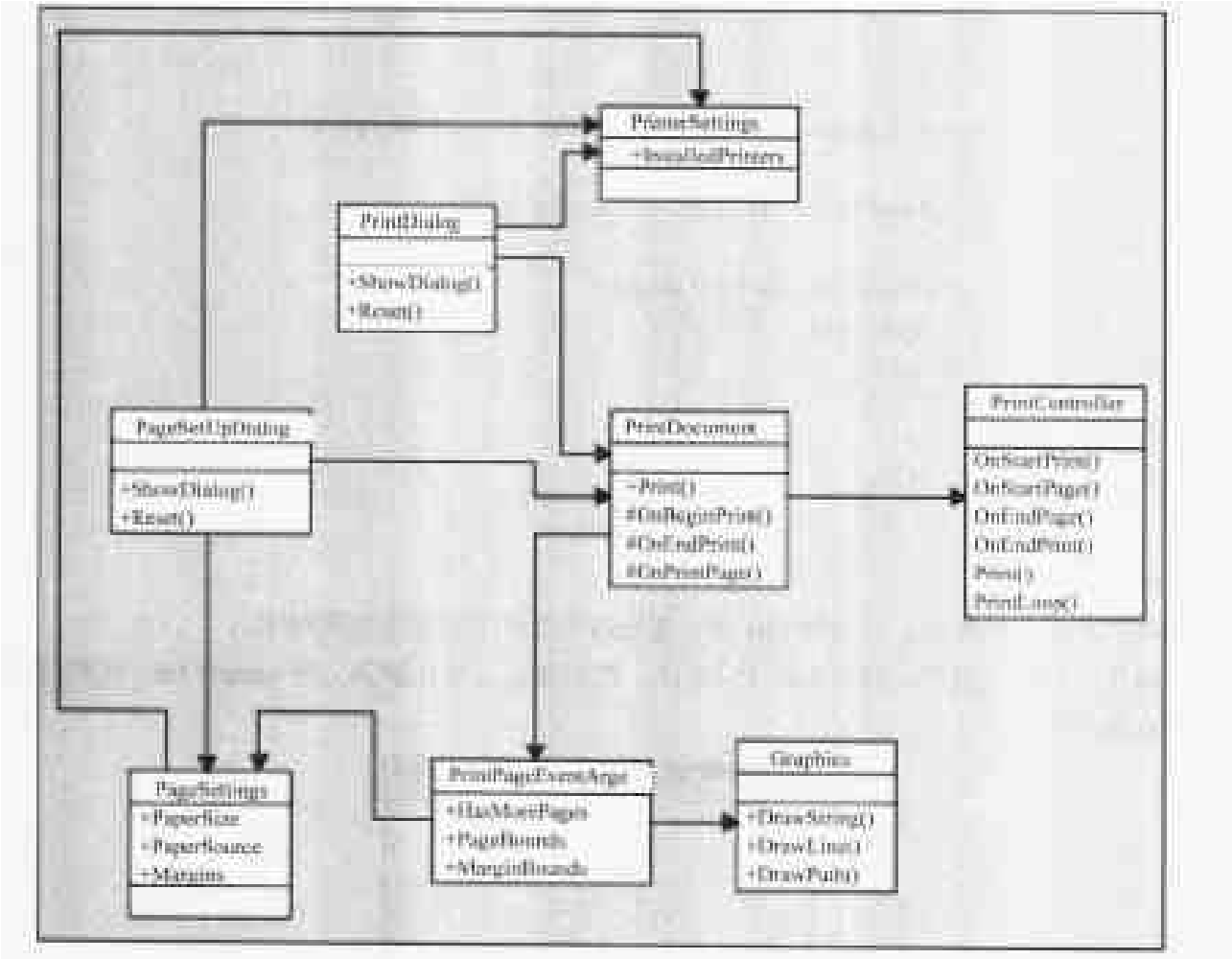


图　15-20

下面看看这些类的功能：

- 首先介绍最重要的类 PrintDocument。在图 15-20 中，几乎所有的类都与这个类有关系。

要打印一个文档，需要 PrintDocument 的一个实例。在图 15-21 中，将介绍这个类启动的打印序列。但我们先介绍其他类。

- PrintController 类控制打印任务流。启动打印任务，打印控制器提供了打印开始、每个页面和打印结束的事件。这个类是抽象类，因为其执行代码不同于正常的打印和打印预览。
- 使用 PrinterSettings 类，可以获取和设置打印机配置，例如双面打印、横向或纵向，打印份数等。
- PrintDialog 类确定使用哪个打印机进行打印和如何配置 PrinterSettings。与前面介绍的其他对话框类一样，这个类派生于 CommonDialog，所以读者对它的使用应有初步的认识。
- PrinterSettings 类指定页面的大小和边界，页面是彩色还是黑白的。这个类的配置由 PageSetupDialog 类进行，PageSetupDialog 也是一个 CommonDialog。

1. 打印序列

前面介绍了打印体系结构中类的作用，下面讨论一下主要的打印序列。图 15-21 显示了主要的内容：要编写的一个应用程序，PrintDocument 类的一个实例和即时序列中的一个 PrintController。

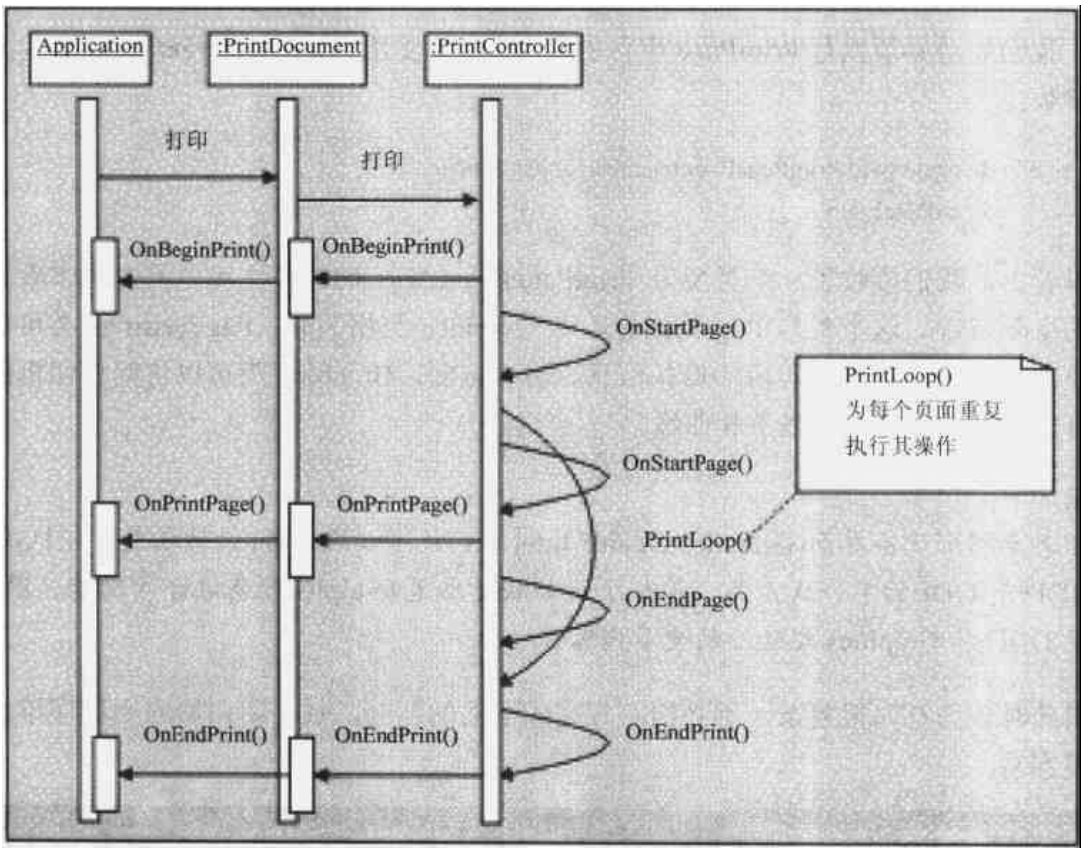


图　15-21

应用程序必须调用 PrintDocument 的 Print()方法，启动调用序列。因为 PrintDocument 本身并不负责打印流，所以由 PrintController 通过调用这个类的 Print()方法完成打印。

打印控制器现在执行操作，通知 PrintDocument，通过调用 OnBeginPrint()开始打印。如果应用程序应在打印任务的开始执行某些操作，就必须在 PrintDocument 中注册一个事件处理程

序，这样我们才会在应用程序类中接到该通知。在图 15-20 中，假定注册了处理程序 OnBeginPrint()，所以这个处理程序在 PrintDocument 类中调用。

在开始阶段完成后，PrintController 就进入 PrintLoop()，为每个要打印的页面调用 PrintDocument 类中的 OnPrintPage()方法。OnPrintPage()调用所有的 PrintPage 事件处理程序。必须对每个页面执行这个处理程序，否则就不会打印任何内容。在图 15-21 中，该处理程序叫作 OnPrintPage()。

在打印完最后一页后，PrintController 就调用 PrintDocument 类中的 OnEndPrint()。也可以执行要在这里调用的处理程序。

总之，最重要的是：

提示：

可以在 PrintDocument.PrintPage 处理程序中执行打印代码。每个要打印的页面都要调用该处理程序。如果打印任务只需要执行一次打印代码，就必须执行 BeginPrint 和 EndPrint 处理程序。

2. PrintPage 事件

现在我们知道必须执行 PrintPage 事件的处理程序。委托 PrintPageEventHandler 定义了处理程序的参数：

```
public delegate void PrintPageEventHandler(object sender,
    PrintPageEventArgs e);
```

可以看出，我们接收了一个类型为 PrintPageEventArgs 的对象。可以回过头来看看类图中这个类的主要属性。这个类与 PageSettings 和 Graphics 类相关联；PageSettings 类可以设置页面的大小、页边距，获取打印机的设备信息。另一方面，Graphics 类可以访问打印机的设备内容，给打印机发送字符串、线条和曲线。

提示：

GDI 表示图形设备界面(Graphics Device Interface)，可以给设备(如屏幕或打印机)输出图形信息。GDI+是 GDI 的下一代产品，添加了诸如渐变画笔和 alpha 颜色混合等功能。第 16 章介绍了使用 GDI+和 Graphics 类绘图的更多内容。

如果此时认为打印很复杂，不必担心！完成下面的范例，就会明白给应用程序添加打印功能其实很容易。

试试看：添加 Print 菜单项

在添加 PrintDialog 之前，必须添加一些打印菜单项。添加两个分隔符和 Print, Print Preview, Page Setup 及 Exit 菜单项。

其中，新菜单项的 Name 和 Text 属性如表 15-4 所示。

表　15-4

菜单项名称	文　本
MiFilePrint	&Print...
MiFilePrintPreview	Print Pre&view...
MiFilePageSetup	Page Set&up...
MiFileExit	E&xit

菜单如图 15-22 所示。

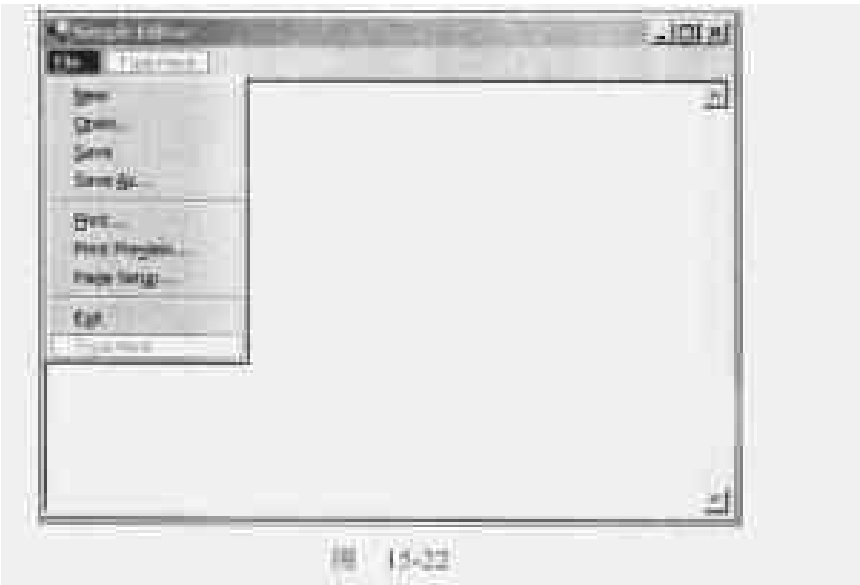


图　15-22

试试看：添加 PrintDocument 组件和处理程序

(1) 把 PrintDocument 组件从工具箱拖放到窗体上，把其名称改为 printDocument，选择 Properties 窗口中的 Events 按钮，给 PrintPage 事件添加事件处理程序 OnPrintPage()。然后把下面的代码添加到该事件处理程序中：

```
private void OnPrintPage(object sender,
                System.Drawing.Printing.PrintPageEventArgs e)
{
    char[] param = {'\n'};
    string[] lines = textBoxEdit.Text.Split(param);

    int i = 0;
    char[] trimParam = {'\r'};
    foreach(string s in lines)
    {
        lines[i++] = s.TrimEnd(trimParam);
    }

    int x = 20;
```

```
        int y = 20;
        foreach (string line in lines)
        {
            e.Graphics.DrawString(line, new Font("Arial", 10),
                Brushes.Black, x, y);
            y += 15;
        }
    }
```

(2) 接着把一个处理程序添加给 Print 菜单的 Click 事件，以调用 PrintDocument 类的 Print() 方法：

```
private void menuItemFilePrint_Click(object sender,
    System.EventArgs e)
{
    printDocument.Print();
}
```

(3) 现在就可以建立和启动应用程序，打印文档了。当然，要使这个范例得到预期的结果，必须安装一台打印机。

代码的说明

printDocument 对象的 Print()方法在 PrintController 类的帮助下，执行 printDocument 的 PrintPage 事件：

```
printDocument.Print();
```

在 OnPrintPage() 处理程序中，使用 String.Split()方法和换行字符\n 逐行分解文本框中的文本。得到的字符串写到变量 lines 中：

```
private void OnPrintPage(object sender,
    System.Drawing.Printing.PrintPageEventArgs e)
{
    char[] param = {'\n'};
    string[] lines = textBoxEdit.Text.Split(param);
```

根据创建文本文件的方式，文本行用字符\n 和\r 分隔开。使用字符串类的 TrimEnd()方法，从每个字符串中删除字符\r：

```
    int i = 0;
    char[] trimParam = {'\r'};
    foreach(string s in lines)
    {
        lines[i++] = s.TrimEnd(trimParam);
    }
```

在下面代码的第二个 foreach 语句中，遍历所有的文本行，调用 e.Graphics.DrawString(),

把每行文本发送给打印机，其中 e 是 PrintPageEventArgs 类型的一个变量，其属性连接到打印机关联文本中。打印机关联文本可以写到打印机设备上；Graphics 类的一些方法可以绘制该关联文本。

注意：

在第 16 章中，介绍了 Graphics 类的方法和绘图的更多内容。

因为我们还没有选择打印机，所以使用默认打印机，其细节存储在 Windows 注册表中。

通过 DrawString()方法，可以为打印输出使用字号为 10 点的 Arial 字体和黑色的画笔。输出结果的位置用变量 x 和 y 定义。水平位置固定为 20 像素，垂直位置则是每行都会递增：

```
int x = 20;
int y = 20;
foreach (string line in lines)
{
    e.Graphics.DrawString(line, new Font("Arial", 10),
                                Brushes.Black, x, y);
    y += 15;
}
```

前面完成的打印并不是完美无瑕的：

- 就像用户使用控制面板设置的那样，打印输出发送给默认打印机。应用程序最好允许用户选择打印机。我们使用 PrintDialog 类来解决这个问题。
- 字体是固定的。要允许用户选择字体，可以使用 FontDialog 类(详见后面的内容)。
- 在程序中，页面的边界被固定为硬编码值。要让用户设置其他页面边界的值，可以使用 PageSetupDialog 类。
- 不能打印多个页面。如果要打印的文档分布在多个页面上，则只能打印第一页。最好还能打印页眉(例如文件名)和页脚(例如页号)。

下面继续介绍打印过程，解决上述问题。

15.4.2　打印多个页面

每个要打印的页面都要调用 PrintPage 事件。必须把 PrintPageEventArgs 类的 HasMorePages 属性设置为 true，告诉 PrintController 当前打印的页面不是最后一页，

试试看：通过修改 OnPrintPage 打印多个页面

(1) 还必须在 SimpleEditorForm 类中声明一个 string[]类型的成员变量 lines 和一个 int 类型的变量 linesPrinted：

```
private string[] lines;
private int linesPrinted;
```

(2) 修改 OnPrintPage()处理程序。在前面 OnPrintPage()的执行代码中，我们把文本分成多行。因为每个页面都会调用 OnPrintPage()方法，把文本分成多行仅在打印操作的开始进行一次。

删除 OnPrintPage()中的所有代码，用下面的新执行代码替换它们：

```
private void OnPrintPage(object sender,
                System.Drawing.Printing.PrintPageEventArgs e)
{
    int x = 20;
    int y = 20;

    while (linesPrinted < lines.Length)
    {
        e.Graphics.DrawString (lines[linesPrinted++],
                new Font("Arial", 10), Brushes.Black, x, y);
        y += 15;
        if (y >= e.PageBounds.Height  - 80)
        {
            e.HasMorePages = true;
            return;
        }
    }

    linesPrinted = 0;
    e.HasMorePages = false;
}
```

(3) 给 printDocument 对象的 BeginPrint 事件添加事件处理程序。每个打印任务只调用 OnBeginPrint()一次，这里创建 lines 数组：

```
private void OnBeginPrint(object sender,
                System.Drawing.Printing.PrintEventArgs e)
{
    char[] param = {'\n'};
    lines = textBoxEdit.Text.Split(param);

    int i = 0;
    char[] trimParam = {'\r'};
    foreach (string s in lines)
    {
        lines[i++] = s.TrimEnd(trimParam);
    }
}
```

(4) 建立工程后，就可以启动多页面文档的打印任务了。

代码的说明

用 PrintDocument 的 Print()方法启动打印任务，该方法会调用 BeginPrint()一次，再对每个页面调用 OnPrintPage()。

在 BeginPrint()中，把文本框中的文本分解到一个字符串数组中。数组中的每个字符串都表示一行文本，因为文本是用换行字符(\n)分解的，并删除了回车字符(\r)，如前所述：

```
char[] param = {'\n'};
lines = textBoxEdit.Text.Split(param);

int i = 0;
char[] trimParam = {'\r'};
foreach (string s in lines)
{
    lines[i++] = s.TrimEnd(trimParam);
}
```

OnPrintPage()是在 BeginPrint()之后调用的。只要已打印的行数小于我们希望打印的行数，我们就将继续打印下去。lines.Length 属性返回数组 lines 中的字符串个数。LinesPrinted 变量的值会随着发送给打印机的各行文本而递增：

```
while (linesPrinted < lines.Length)
{
    e.Graphics.DrawString(lines[linesPrinted++],
                new Font("Arial", 10), Brushes.Black, x, y);
```

打印一行后，就要检查新计算出来的垂直位置是否超过了页面的边界，同时给边界减去 80 个像素，因为我们并不想真的打印到页面的终结处，许多打印机都不会打印到页面的终结处。如果到达这个位置，PrintPageEventArgs 类的 HasMorePages 属性就会设置为 true，通知控制器，必须再次调用 OnPrintPage()方法。PrintController 的一个方法 PrintLoop()有一个用于每个要打印的页面的序列。如果 HasMorePages 是 false，PrintLoop()就会停止。HasMorePages 属性的默认值是 false，所以打印会停止：

```
    y += 15;
    if (y >= e.PageBounds.Height - 80)
    {
        e.HasMorePages = true;
        return;
    }
```

15.4.3　页面设置

到目前为止，页面的边距都是在程序中硬编码的。下面修改应用程序，允许用户在页面上设置页边距。为此，应使用另一个对话框类：PageSetupDialog。

这个类可以配置页面大小和纸张来源、方向、页边距，因为这些选项都依赖于打印机，所以在这个对话框中也可以选择打印机。

1. 页面

AllowPaper 属性的值为 true，则用户就可以选择页面大小和纸张来源。PageSetupDialog.

PageSettings.PaperSize 属性返回一个 PaperSize 实例，用它的属性 Height、Width 和 PaperName 可以读取高度、宽度和纸张名称。PaperName 指定诸如 Letter 和 A4 等名称。Kind 属性返回一个枚举，从该枚举中可以获取 PaperKind 枚举的一个值，它可以是纸张大小 European, American 或 Japanese 3 个值中的一个。

PageSetupDialog.PageSettings.PaperSource 属性返回一个 PaperSource 实例，其中可以读取打印机纸张来源的名称和相应的纸张类型(只要打印机用该打印机设置进行了正确配置)。

2. 页边距

把 AllowMargins 属性设置为 true，将允许用户设置打印输出的页边距值。指定 MinMargins 属性，可以为用户定义可以输入的最小页边距值。要读取页边距，可以使用 PageSetupDialog.PageSettings.Margins 属性。返回的 Margins 对象有 Bottom, Left, Right 和 Top 属性。

3. 方向

AllowOrientation 属性定义用户是否可以选择纵向和横向打印方式。查看 PageSetupDialog.PageSettings.Landscape 的值可以读取选定的值，这是一个布尔属性，其值为 true 时，指定横向打印，其值为 false 时，指定纵向打印。

4. 打印机

AllowPrinter 属性指定用户是否可以选择打印机。根据这个属性的值，可以启用或禁用 Printer 按钮。这个按钮的处理程序会打开 PrintDialog，后面将介绍这个对话框。

5. PageSetupDialog 属性

图 15-23 描述了可以启用或禁用这个对话框中特殊选项的属性，以及哪些属性可以用于访问这些值。

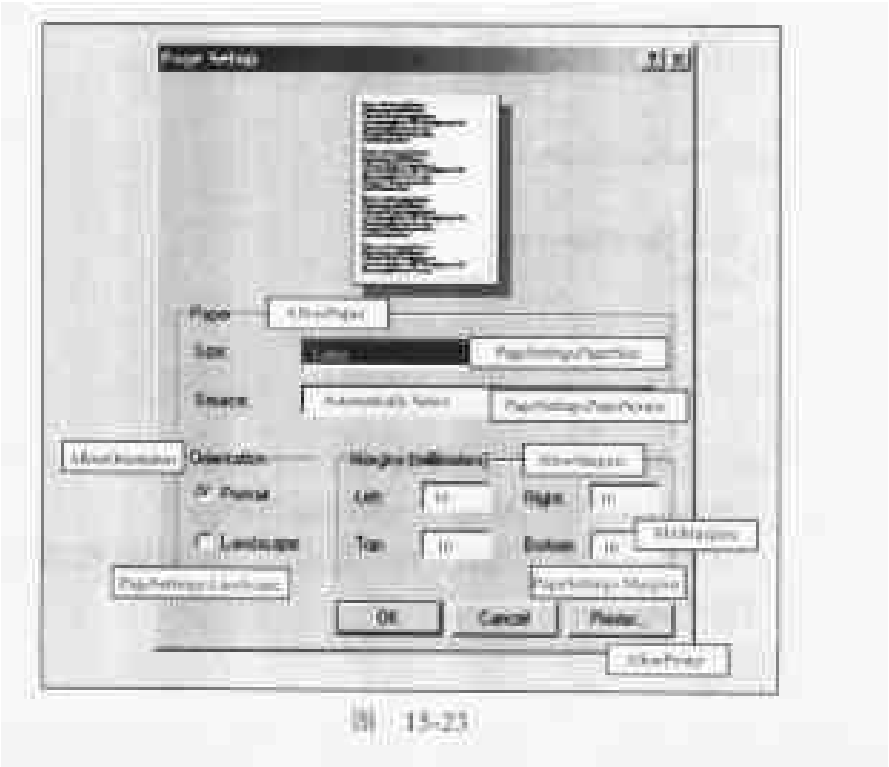


图 15-23

试试看：添加页面设置对话框

(1) 把工具箱中的 PageSetupDialog 组件添加到 Windows Forms 设计器中。把属性 Name 设置为 dlgPageSetup，Document 设置为 printDocument，把要打印的文档与对话框关联起来。

然后给 Page Setup 菜单项添加一个单击事件处理程序，添加代码，使用 ShowDialog()方法显示该对话框。这里不需要检查 ShowDialog()的返回值，因为 OK 单击事件处理程序的执行代码在关联的 PrintDocument 对象中设置了新值：

```
private void menuItemFilePageSetup_Click(object sender,
    System.EventArgs e)
{
    dlgPageSetup.ShowDialog();
}
```

(2) 现在修改 OnPrintPage()的执行代码，使用由 PageSetupDialog 设置的页边距。在代码中，x 和 y 变量设置为 PrintPageEventArgs 类的属性 MarginBounds.Left 和 MarginBounds.Top。用 MarginBounds.Bottom 检查页面的边界：

```
private void OnPrintPage(object sender,
    System.Drawing.Printing.PrintPageEventArgs e)
{
    int x = e.MarginBounds.Left;
    int y = e.MarginBounds.Top;

    while (linesPrinted < lines.Length)
    {
        e.Graphics.DrawString(lines[linesPrinted++],
            new Font("Arial", 10), Brushes.Black, x, y);

        y += 15;
        if (y >= e.MarginBounds.Bottom)
        {
            e.HasMorePages = true;
            return;
        }
    }

    linesPrinted = 0;
    e.HasMorePages = false;
}
```

(3) 现在可以建立工程，运行应用程序了。选择 File | Page Setup，显示如图 15-24 所示的对话框。可以修改边界，用配置好的边界进行打印。

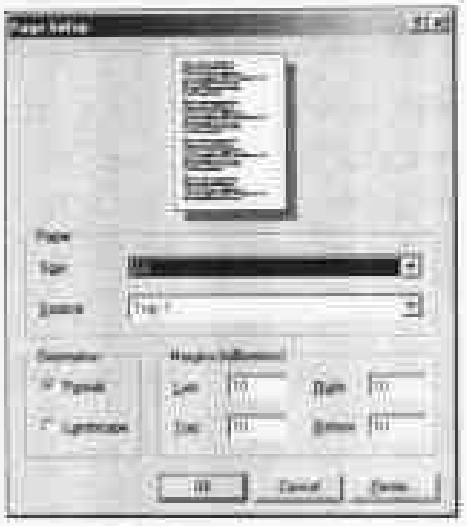


图 15-24

提示：

如果 PageSetupDialog 的显示失败，就会抛出一个 System.ArgumentException 类型的异常，这可能是因为忘记把 PrintDocument 对象与 PageSetupDialog 关联起来了。PageSetupDialog 需要一个关联的 PrintDocument，才能在对话框中设置和查询要显示的各个值。

15.4.4 打印对话框

PrintDialog 类允许用户从已安装的打印机中选择一台打印机，选择打印的份数和其他一些打印机设置，例如布局和打印机的纸张来源。因为 PrintDialog 非常容易使用，所以下面就给 Editor 应用程序添加一个 PrintDialog。

试试看：添加 PrintDialog

(1) 从工具箱中把 PrintDialog 组件添加到窗体中。把其名称设置为 dlgPrint，把这个对象的属性 Document 设置为 printDocument。

修改 Print 按钮的单击事件处理程序的执行代码，如下所示：

```
private void miFilePrint_Click(object sender, System.EventArgs e)
{
    if (dlgPrint.ShowDialog() == DialogResult.OK)
    {
        printDocument.Print();
    }
}
```

(2) 建立并运行应用程序。选择 File | Print，打开 PrintDialog，如图 15-25 所示，现在就可以选择打印机来打印文档了。

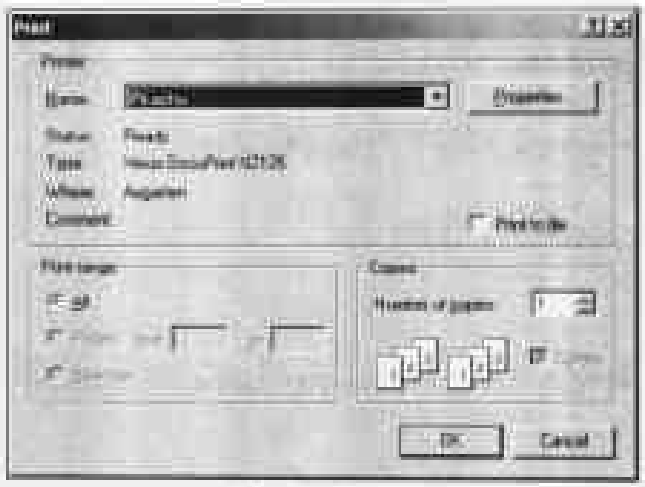


图 15-25

1. 打印对话框中的选项

在 SimpleEditor 程序中，没有修改 PrintDialog 的任何属性。但这个对话框也有一些选项。在上面的对话框中，有 3 个选项组：Printer, Print range 和 Copies。

- 在 Printer 选项组中，不仅可以选择打印机，还有一个 Print to File 选项。在默认情况下，这个选项是启用的，但没有选中。选择这个复选框可以让用户把输出结果打印到一个文件中，而不是打印机上。把 AllowPrintToFile 属性设置为 false，可以禁用这个选项。

如果用户选择了这个选项，rintDocument.Print()就会调用图 15-26 所示的对话框，要求用户键入存储打印结果的文件名。



图 15-26

- 在对话框的 Print Range 选项组中，只能选择 All——在默认状态下，Pages 和 Selection 是禁用的。后面的章节将介绍如何执行这些选项。
- Copies 选项组可以让用户选择要打印的份数。

2. 打印选中的文本

如果只打印选中的文本，可以把属性 AllowSelection 设置为 true，启用这个单选按钮。也可以修改打印代码，只打印选中的文本。

试试看：添加打印选项

把下面突出显示的代码添加到 Print 按钮的单击事件处理程序中：

```
private void miFilePrint_Click(object sender, System.EventArgs e)
{
    if (textBoxEdit.SelectedText != "")
```

```
        {
            dlgPrint.AllowSelection = true;
        }
        if (dlgPrint.ShowDialog() == DialogResult.OK)
        {
            printDocument.Print();
        }
    }
```

在程序中，所有要打印的文本行都在 OnBeginPrint()处理程序中设置了。修改这个方法的执行代码：

```
    private void OnBeginPrint(object sender,
        System.Drawing.Printing.PrintEventArgs e)
    {
        char[] param = {'\n'};

        if (dlgPrint.PrinterSettings.PrintRange == PrintRange.Selection)
        {
            lines = textBoxEdit.SelectedText.Split(param);
        }
        else
        {
            lines = textBoxEdit.Text.Split(param);
        }

        int i = 0;
        char[] trimParam = {'\r'};
        foreach (string s in lines)
        {
            lines[i++] = s.TrimEnd(trimParam);
        }
    }
```

现在就可以建立并启动程序了。打开一个文件，选择一些文本，用菜单 File | Print 启动打印对话框。从 Print Range 选项组中选择 Selection 选项按钮，之后，单击 OK 按钮，就只打印选中的文本，如图 15-27 所示。

代码的说明

只有在选择了一些文本之后，才能把 AllowSelection 属性设置为 true。在显示 PrintDialog 前，必须检查是否已经选择了一些文本，即确保编辑框的 SelectedText 属性值不是 null。如果选择了一些文本，就把 AllowSelection 属性设置为 true：

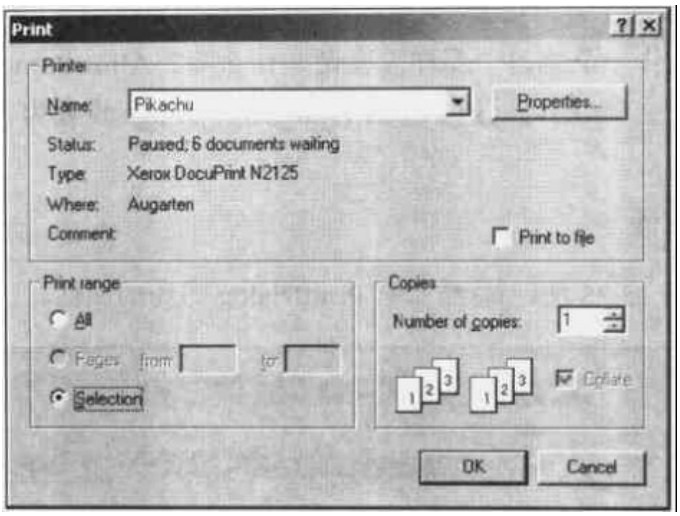


图　15-27

```
if (textBoxEdit.SelectedText != "")
{
    dlgPrint.AllowSelection = true;
}
```

在每个打印任务的开始都要调用 OnBeginPrint()，访问 printDialog.PrinterSettings.PrintRange 属性，可以获得用户是否选择了 Selection 选项的信息，PrintRange 属性返回一个 PrintRange 枚举，该枚举包含 AllPages, Selection 或 SomePages 值：

```
if (printDialog.PrinterSettings.PrintRange == PrintRange.Selection)
{
```

如果选项是 Selection，就可以用文本框的 SelectedText 属性获取选中的文本字符串，这个字符串的分解方式与完整的文本的分解方式相同：

```
    lines = textBoxEdit.SelectedText.Split(param);
}
```

使用 PrintRange 枚举时，必须在程序的开头包含名称空间 System.Drawing.Printing：

```
using System.Drawing.Printing;
```

3. 打印页面的范围

打印某个范围的页面可以以与打印选中文本类似的方式进行。把 AllowSomePages 属性设置为 true，就可以启用该选项按钮。用户现在可以选择要打印的页面范围。但是，Simple Editor 中的页面边界在哪里？哪一页是最后一页？设置 PrintDialog.PrinterSettings.ToPage 属性就可以设置最后一页。用户如何知道要打印的页码？在诸如 Microsoft Word 这样的文档处理应用程序中，这是没有问题的，因为在 Microsoft Word 中，可以选择在屏幕上的 Print Layout 作为视图，但这不能在 Simple Editor 应用程序中使用的简单 TextBox 来实现。这就是我们没有在该应用程

序中执行该功能的原因。

当然，可以把这当作一个练习。那样需要做些什么呢？AllowSomePages 属性必须设置为 true。在显示 PrintDialog 之前，还可以把 PrinterSettings.FromPage 设置为 1，PrinterSettings.ToPage 设置为最大的页码数。

4. PrintDialog 的属性

下面再用一个图(图 15-28 所示)总结影响 PrintDialog 布局的属性。

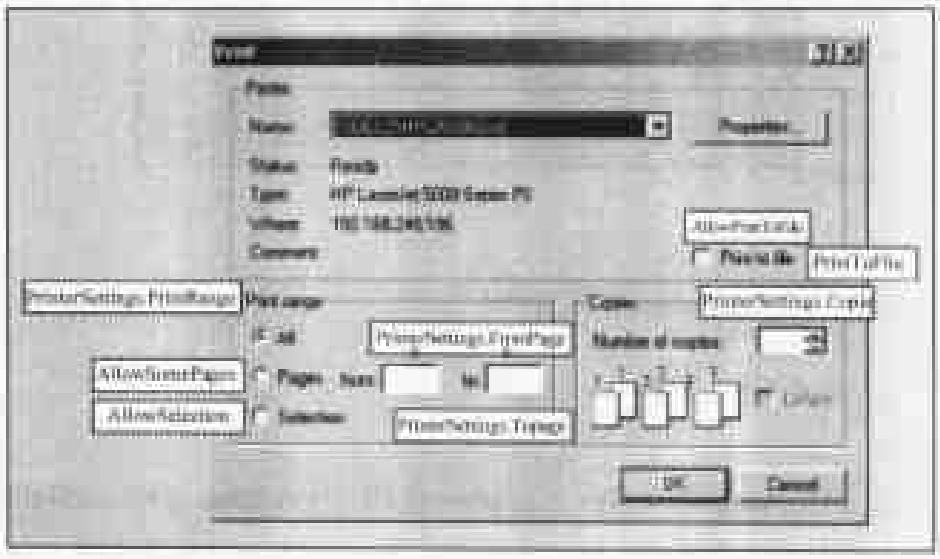


图 15-28

15.5 打印预览

也许用户想查看打印结果。这就是使用打印预览的原因。在.NET 中执行 Print Preview 是很简单的。可以使用 PrintPreviewControl 类，该类可以用于在窗体中预览文档，说明如何打印它。PrintPreviewDialog 是一个封装了一些控件的对话框。

15.5.1 PrintPreviewDialog

查看一下 PrintPreviewDialog 类的 MSDN 说明提供的属性和继承列表，很容易看出这是一个窗体，如图 15-29 所示，不是一个封装的通用对话框—— 该类派生于 System.Windows.Forms.Form，使用它的方式与使用前面几章创建的窗体的方式相同。

在 Simple Editor 应用程序中添加一个 PrintPreviewDialog 类。

试试看：添加打印预览对话框

给 Windows Forms 设计器添加一个 PrintPreviewDialog 组件，把其名称设置为 dlgPrintPreview，Document 属性设置为 printDocument。

添加并执行 Print Preview 菜单项的 Click 事件处理程序，得到如图 15-29 所示的结果。

```
private void miFilePrintPreview_Click(object sender,
    System.EventArgs e)
{
    dlgPrintPreview.ShowDialog();
}
```

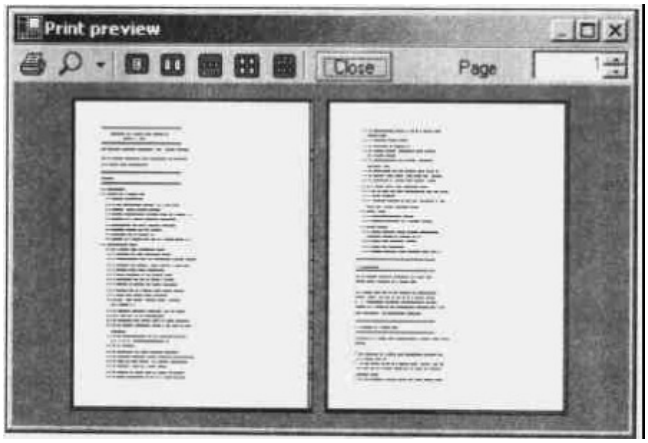


图　15-29

15.5.2　PrintPreviewControl

Microsoft Word 和 WordPad 中的 Print Preview 与 PrintPreviewDialog 不同，因为在这些应用程序中，预览不在自己的对话框中显示，而是在应用程序的主窗口中显示。

为此，可以在窗体上放置 PrintPreviewControl 类。Document 属性必须设置为 PrintDocument 对象，Visible 属性必须设置为 false。在处理程序中，可以把 Visible 属性设置为 true，以查看打印预览。然后把 PrintPreviewControl 放在其他控件的前面，如图 15-30 所示：

可以从标题和 File 菜单项中看出，这里显示的是 Simple Editor 应用程序的主窗口。仍需要添加一些元素，控制 PrintPreviewControl 类进行缩放、打印和显示 2、3、4 或 6 页中的文本。一个特殊的工具栏可以用于激活这些功能。PrintPreviewDialog 类已经实现了这些功能，如图 15-31 显示了 4 页。

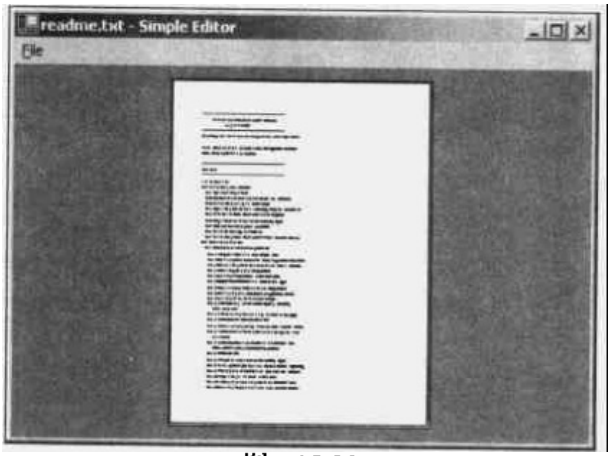


图　15-30

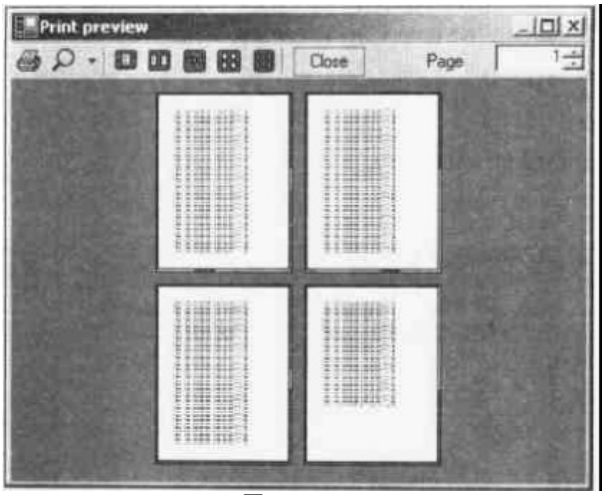


图　15-31

15.6　FontDialog 和 ColorDialog

本章要介绍的最后一个对话框是 FontDialog 和 ColorDialog。

注意：
这里仍只讨论设置字体和颜色的对话框，不讨论 Font 和 Color 类，这两个类详见第 16 章。

15.6.1　FontDialog

FontDialog 让应用程序的用户选择字体。用户可以改变字体、样式、字号和字体的颜色。图 15-32 描述了修改对话框中各元素的属性。

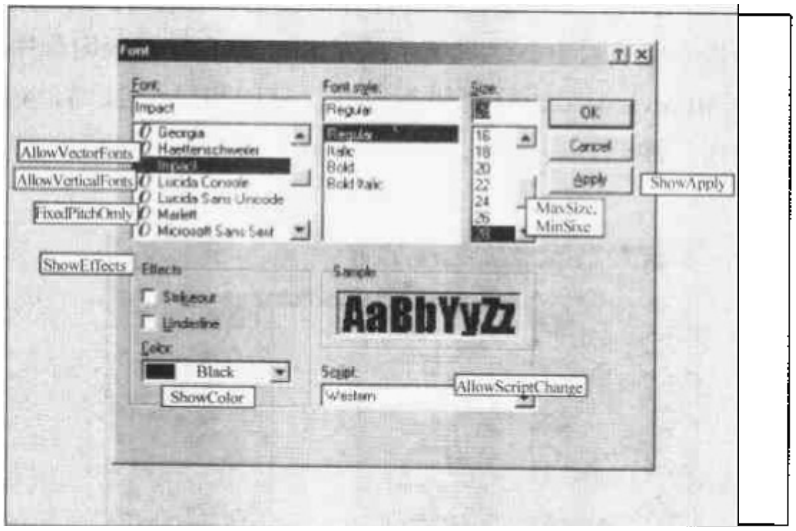


图　15-32

1. 如何使用 FontDialog

这个对话框的使用方式与前面的对话框相同。在 Windows Forms 设计器中，可以将对话框从工具箱拖放到窗体上，创建一个 FontDialog 的实例。

使用 FontDialog 的代码如下所示：

```
if (dlgFont.ShowDialog() == DialogResult.OK)
{
    textBoxEdit.Font = dlgFont.Font;
}
```

调用 ShowDialog()方法就可以显示 FontDialog。如果用户单击 OK 按钮，该方法就返回 DialogResult.OK。使用 FontDialog 类的 Font 属性，可以读取选中的字体；接着把这个字体传送给 TextBox 的 Font 属性。

2. FontDialog 的属性

图 13-32 中已列出了 FontDialog 类的属性，现在看看这些属性的用途。

表 15-5

属　性	描　述
AllowVectorFonts	布尔值，定义是否可以在字体列表中选择矢量字体
AllowVerticalFonts	布尔值，定义是否可以在字体列表中选择垂直字体(Vertical font)。垂直文本用于远东国家，系统上可能没有安装垂直字体
FixedPitchOnly	设置属性 FixedPitchOnly 将只在字体列表中显示固定大小的字体。使用固定大小的字体，会使每个字符有相同的大小。其默认值是 false
MaxSize	给 MaxSize 属性指定一个值，定义用户可以选择的最大字号
MinSize	类似于 MaxSize，使用 MinSize 能设置用户可以选择的最小字号
ShowApply	如果应显示 Apply 按钮，必须把 ShowApply 属性设置为 true。按下 Apply 按钮，用户可以在应用程序中查看更新的字体，而无需退出字体对话框
ShowColor	在默认状态下，Color 选项不会显示在对话框中。如果希望用户能在字体对话框中选择字体的颜色，只需把 ShowColor 属性设置为 true
ShowEffects	在默认状态下，用户可以选择 Strikeout 和 Underline 复选框来处理字体。如果不希望显示这些选项，就必须把 ShowEffects 属性设置为 false
AllowScriptChange	把 AllowScriptChange 属性设置为 false，可以防止用户改变字体的脚本。可用的脚本依赖于选中的字体，例如，字体 Arial 支持 Western, Hebrew, Arabic, Greek, Turkish, Baltic, Central European, Cyrillic 和 Vietnamese 脚本

3. 激活 Apply 按钮

FontDialog 与其他对话框的一个有趣区别是，FontDialog 支持 Apply 按钮，但在默认状态下不显示它。如果用户单击 Apply 按钮，对话框会一直打开，但字体会应用到文本上。

在 Windows Forms 设计器中选择 FontDialog 之后，就可以在 Properties 窗口中把 ShowApply 属性设置为 true。但如果用户现在单击了 Apply 按钮，该怎么办？对话框仍是打开的，所以 ShowDialog()方法没有返回。我们需要给 FontDialog 类的 Apply 事件添加一个处理程序，为此，

可以单击 Properties 窗口中的 Events 按钮，并给 Apply 事件起一个处理程序名。

从下面的代码可以看出，这里输入了名称 OnApplyFontDialog。在这个处理程序中，可以使用 FontDialog 类的成员变量访问 FontDialog 的选中字体：

```
private void OnApplyFontDialog(object sender, System.EventArgs e)
{
    textBoxEdit.Font = dlgFont.Font;
}
```

15.6.2 ColorDialog

配置 ColorDialog 并不像 FontDialog 那样麻烦。如果用户不想使用给出的任何颜色，使用 ColorDialog 可以让用户配置定制颜色，为此，需要设置 AllowFullColor 属性。使用 FullColor 属性，也可以自动展开该对话框的定制颜色配置部分。SolidColorOnly 指定只能选择单色。CustomColors 属性可以用于获取和设置配置的定制颜色值，如图 15-33 所示。

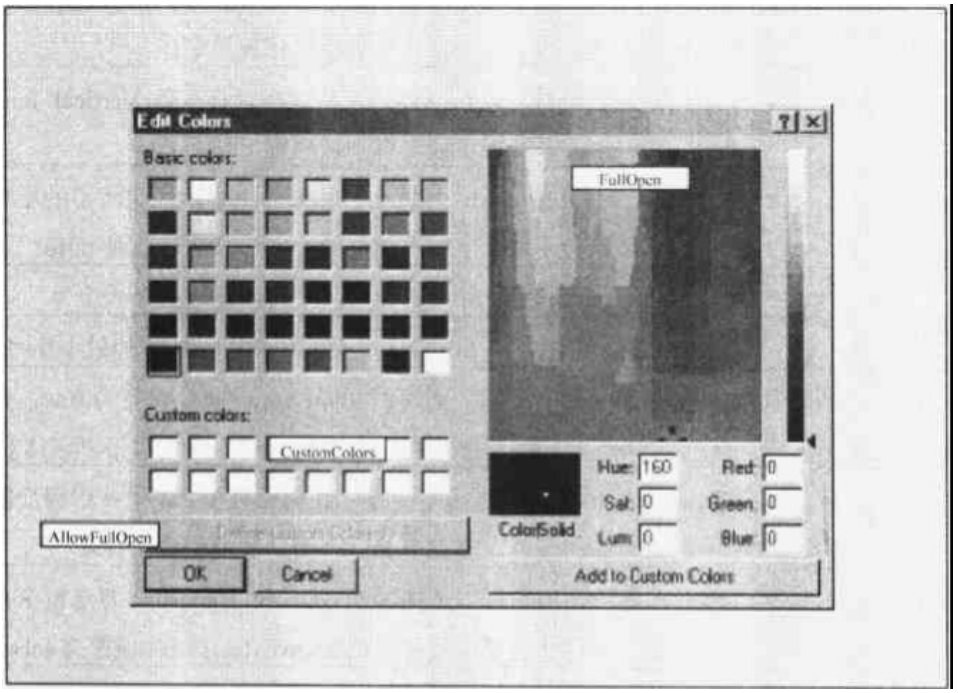


图　15-33

1. 如何使用颜色对话框

与其他对话框一样，ColorDialog 可以从工具箱拖放到 Windows Forms 设计器的窗体上。ShowDialog()显示该对话框，用户单击 OK 或 Cancel 按钮后，该对话框关闭。通过访问对话框的 Color 属性可以读取选中的颜色，如下面的代码范例所示：

```
if (dlgColor.ShowDialog() == DialogResult.OK)
{
    textBoxEdit.ForeColor = dlgColor.Color;
}
```

2. 颜色对话框的属性

影响该对话框外观的属性如表 15-6 所示。

表　15-6

属　性	描　述
AllowFullOpen	把这个属性设置为 false，将不允许用户定义定制颜色，因为按钮 Define Custom Colors 是禁用的。这个属性的默认值是 true
FullOpen	在显示对话框前，把 FullOpen 属性设置为 true，会自动打开定期制颜色选项的对话框
AnyColor	把这个属性设置为 true，将在基本颜色列表中显示所有可用的颜色
CustomColors	使用 CustomColors 属性可以预置一个定制颜色数组，并可以读取用户定义的定制颜色
SolidColorOnly	把 SolidColorOnly 属性设置为 true，用户就只能选择单色

15.7　小结

本章介绍了如何在应用程序中使用对话框类。我们讨论了如何打开和保存文件，在复习了.NET Framework 的打印类后，论述了如何给应用程序添加打印功能。总之，在 Simple Editor 应用程序中，使用了下述对话框类：

- FileOpenDialog 要求用户指定要打开的文件。
- FileSaveDialog 要求用户提供一个文件名，以保存数据。
- PrintDialog 可以选择要用于打印的打印机和打印配置。
- PageSetupDialog 修改打印页码的页边距。
- PrintPreviewDialog 预览打印结果，让用户事先了解打印的结果。

我们还介绍了 FontDialog 和 ColorDialog 类的基础知识。请读者练习把这两个类添加到 Simple Editor 应用程序中。

15.8　练习

因为 FontDialog 和 ColorDialog 的使用方式与本章介绍的其他对话框类似，所以很容易把这些对话框添加到 Simple Editor 应用程序中：

(1) 允许用户改变文本框的字体。为此，在主菜单中添加一个新的菜单项 F&ormat 和一个子菜单 Format: &Font...。给这个菜单项添加一个处理程序。利用 Windows Forms 设计器把 FontDialog 添加到应用程序中。在菜单的处理程序中显示这个对话框，把文本框的 Font 属性设置为选中的字体。

还需要修改方法 OnPrintPage()的执行代码，为打印输出使用选中的字体，在前面的执行代码中，我们在 Graphics 对象的 DrawString()方法中创建一个新 Font 对象。现在通过访问 Font 属性，使用 textBoxEdit 的字体。如果用户选择了一个大字号，还必须注意字体的位置。为了避免一行文本会部分覆盖其上或其下的文本，修改固定值，改变文本行的垂直位置。最好使用字体的字号来递增文本行的垂直位置：使用 Font 类的 Height 属性。

(2) 对 Simple Editor 应用程序的另一个比较大的改进是改变字体的颜色。在 Format 菜单项中添加第二个子菜单 Color...。给这个菜单项添加处理程序，打开一个 ColorDialog。如果用户单击了 OK 按钮，就把文本框的 ForeColor 属性设置为 ColorDialog 的选中颜色。

在方法 OnPrintPage()中，确保选中的颜色只能在打印机支持该颜色时使用。使用 PrintPageEventArgs 参数的 PageSettings.Color 属性，就可以查看打印机支持哪些颜色。使用下面的代码，可以利用文本框的颜色创建一个画笔对象：

```
Brush brush = new SolidBrush(textBoxEdit.ForeColor);
```

这个画笔可以用作 DrawString()方法中的一个参数，代替前面范例中使用的黑色画笔。

第16章 GDI+简介

本章简要介绍如何使用 Graphics Device Interface 类(GDI+)编程。映射应用程序、游戏、计算机辅助设计/计算机辅助制造技术(CAD/CAM)、绘图程序、表格程序和许多其他类型的应用程序都要求开发人员为 Windows Forms 应用程序编写图形代码。编写定制控件也需要图形代码。利用这个最新的类库，Microsoft 使图形代码的编写比以前任何时候都简单。

编写图形代码是最愉快的编程任务之一。修改代码后，立即能在可见的窗体上看到结果。无论是编写定制图形窗口，以另一种新颖的方式在应用程序中显示某些信息，还是编写一个定制控件，都能使应用程序更漂亮、可用性更高，一般用户都可以接受这些应用程序。

首先，解释使用 GDI+进行绘图的机制，再编写几个简单的图形程序范例。然后深入探讨 GDI+的扩展功能，包括：

- 剪切
- 打印
- 打印预览
- Drawing2D 名称空间
- Imaging 名称空间

在讨论完上述各个主题后，将介绍用于执行这些功能的类，以及可以在何处找到有关绘图的更多资料。知道能做什么，理解类的层次结构仅掌握了一半内容。

16.1 图形绘制概述

编写图形代码，首先要知道如果有关窗口被其他窗口遮挡住，Microsoft Windows 并没有存储每个已打开窗口的外观。如果被遮挡的窗口置于前台，让它可见，Windows 就会告诉应用程序“你的窗口(或一部分窗口)现在是可见的，需要绘制它吗？”我们只需绘制窗口的内容，Windows 本身会考虑窗口的边框、标题栏和其他窗口要素。

在编程术语中，当创建一个窗口，并在该窗口中进行绘图时，一般要声明一个派生于 System.Windows.Forms.Form 的类。如果要编写一个定制控件，就要声明一个派生于 System.Windows.Forms.UserControl 的类。在这两种情况下，都重写了虚拟函数 OnPaint()。只要窗口的任何一部分需要重新绘制，Windows 都会调用这个函数。

在这个事件中，PaintEventArgs 类是一个参数。在 PaintEventArgs 中有两个重要的信息：Graphics 对象和 ClipRectangle 对象。下面介绍 Graphics 对象。本章的最后将介绍剪切。

16.1.1 Graphics 类

Graphics 类封装了一个 GDI+绘图界面。有 3 种基本类型的绘图界面：

- Windows 和屏幕上的控件
- 要发送给打印机的页面
- 内存中的位图和图像

Graphics 类提供了可以在这些绘图界面上绘图的功能。在其他功能中，我们可以使用它绘制圆弧、曲线、Bezier 曲线、椭圆、图像、线条、矩形和文本。

给窗口获得 Graphics 对象有两种不同的方式。首先是重写 OnPaint()事件，该事件是一个 Form 类继承 Control 类的虚拟方法。下面从利用该事件传送的 PaintEventArgs 中获取 Graphics 对象：

```
protected override void OnPaint(PaintEventArgs e) {
    Graphics g = e.Graphics;

    // do our drawing here

}
```

有时，需要直接在窗口中绘图，而无需等待 OnPaint()事件。例如要编写代码，选择窗口中的某些图形对象(类似于在 Windows Explorer 中选择图标)，或者用鼠标拖动一些对象，就是这种情况。在窗体上调用 CreateGraphics()方法就可以获得一个 Graphics 对象，这是 Form 类继承 Control 类的另一个方法：

```
protected void Form1_Click (object sender, System.EventArgs e)
{
    Graphics g = this.CreateGraphics();

    // do our drawing here

    g.Dispose();    // this is important
}
```

建立一个处理拖放操作的应用程序是一件有点棘手的事，超出了本书的范围。无论如何，这是一个不太常用的技术。我们做的几乎所有工作主要是为了响应 OnPaint()事件。

获得 Graphics 对象还有其他方式，详见后面的内容。

16.1.2 对象的删除

提示：

Graphics 对象总是要调用 Dispose()。

每个人都很熟悉 Windows 在用尽资源时的状态。Windows 开始运行得非常缓慢，有时应用程序不能正确绘制。操作正确的应用程序会在用完资源后释放它们。在使用.NET Framework 进行开发时，有几种数据类型对于调用 Dispose()是非常重要的，否则一些资源就不能释放。这些类执行 IDisposable 接口。Graphics 是这些类中的一个。但是，只有在调用 CreateGraphics()获得 Graphics 对象时，调用 Dispose()才是重要的。如果是从 OnPaint()事件中获得的 Graphics

对象，而不是创建了该对象，就不应调用 Dispose()，但在上面的范例中，应调用 Dispose()。

对于执行 IDisposable 的许多类来说，Dispose()方法会在析构函数中自动调用。读者可能认为，这样我们就不必调用 Dispose()了，但事实并非如此。原因是只有在无用存储单元收集器(GC)调用了析构函数时，我们才不必调用 Dispose()，但不能确保 GC 何时运行。特别是，在内存很大的 Windows 9X 操作系统中，GC 可能运行次数非常少，所有的资源都在运行 GC 之前就用尽了。而内存用尽后会触发 GC，但资源用尽后则不会触发 GC。Windows 2000 对资源用尽的情况并不是特别敏感。根据规范，Windows 2000 对这些类型的资源没有任何限制，但当打开了太多的应用程序时，这个操作系统就会出错。关闭一些应用程序会很快恢复正确的操作。无论如何，最好编写代码，以一种省时的方式，手动删除占用大量资源的对象。

在.NET Framework 的 Beta 2 中引入了一个新结构，它可以用于强迫 Dispose()运行。在对象超出作用域时，Using 结构会自动调用 Dispose()。下面的代码演示了 using 关键字在这种情况下的正确用法：

```
using (Graphics g = this.CreateGraphics())
{
    g.DrawLine(Pens.Black, new Point(0, 0),
        new Point(3, 5));
}
```

根据上面的说明，其代码应等价于：

```
Graphics g = this.CreateGraphics();
try {
    g.DrawLine(Pens.Black, new Point(0, 0),
        new Point(3, 5));
}
finally {
    if (g != null)
        ((IDisposable)g).Dispose();
}
```

不要把这个 using 关键字和 using 指令混淆，using 指令可以为名称空间创建别名，或者允许在一个名称中空间中使用一些类，这样就不需要完全限定该类的用法了。这与 using 关键字的用法完全不同。

本章的范例将使用这两种方式处理对 Dispose()的调用。有时直接调用 Dispose()，有时则使用 using 结构。后者是一种比较清晰的解决方案，这从上面的代码段可以看出，但哪种方式更好则没有定论。

在介绍第一个范例前，绘图还有其他两个方面需要介绍——坐标系统和颜色。

16.1.3　坐标系统

在设计一个绘制复杂图形的程序时，最重要的是代码能绘制出我们想要的图形，而不绘制我们不希望有的图形。一个放错地方的像素会对图形有负面影响，所以在执行绘图操作时，理

解应绘制哪些像素是非常重要的。在创建定制控件时，常常要绘制许多矩形、水平线和垂直线，所以这也很重要。一条线长了一个像素，或者短了一个像素都是很显眼的。但是，这对于曲线、对角线和其他图形操作并不重要。

GDI+的坐标系统建立在通过像素中心的假想数学直线上，这些直线从 0 开始，其左上角的交点是 X=0, Y=0。X=1, Y=2 的简短记号是点(1,2)。用于绘图的每个窗口都有自己的坐标。如果要创建一个可以在其他窗口中使用的定制控件，这个定制控件本身就有自己的坐标。换言之，在绘制该定制控件时，它的左上角是点 0,0。不必担心定制控件会放在其包含窗口的什么地方。

在绘制线条时，GDI+会把绘制出来的像素在指定的数学直线上对中。在绘制整数坐标的水平线时，可以认为每个像素的一半落在假想数学直线的上半部分，而另一半落在假想数学直线的下半部分。在从点 1,1 到点 5,1 绘制一个像素宽的水平线时，会得到如图 16-1 所示的像素。

在从点 2,1 到点 2,4 绘制一个像素宽、4 个像素长的垂直线时，会得到如图 16-2 所示的像素。

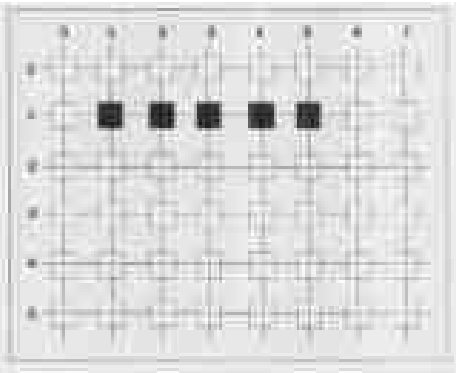


图 16-1

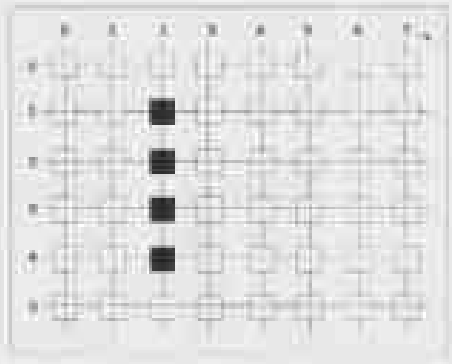


图 16-2

在绘制从点 1,0 到点 4,3 的对角线时，会得到如图 16-3 所示的像素。

在绘制一个矩形，其左上角为 1,0，右下角为 6,4 时，矩形应如图 16-4 所示。

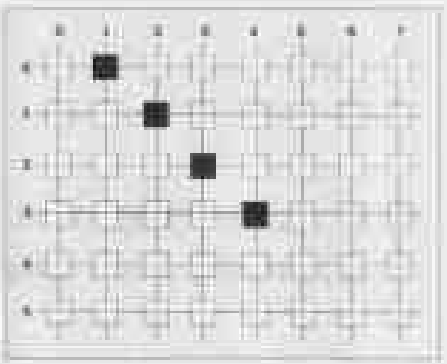


图 16-3

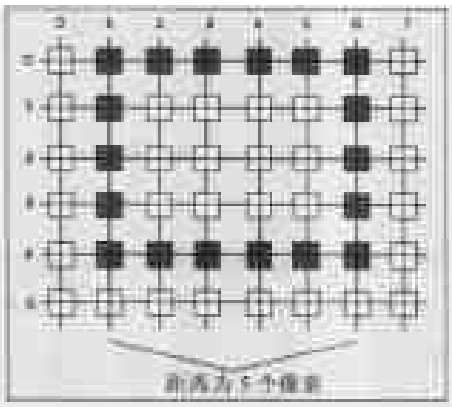


图 16-4

注意这里有一些很有趣的地方。如果指定了宽度为 5，就会在水平方向上绘制 6 个像素。但如果考虑到数学直线通过像素中心，该矩形就只有 5 个像素宽，绘制的线条有一半像素落在假想数学直线的外面，一半像素则落在假想数学直线的里面。

还不仅如此。如果使用图形保真技术进行绘图，其他像素就会上一半的颜色，创建出光滑的线条，部分避免了对角线的“台阶”外观。

图 16-5 中所示的是没有使用图形保真技术绘制的线条。

图 16-6 中所示的是使用了图形保真技术绘制的线条。

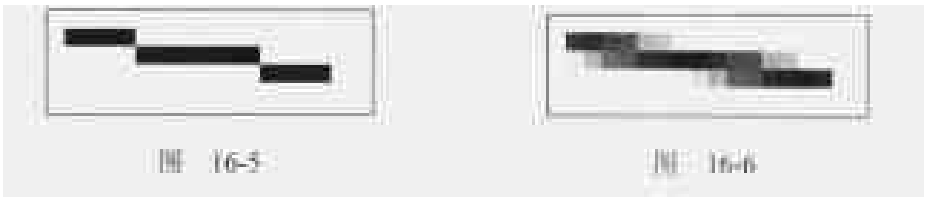


图 16-5　　图 16-6

在高分辨率下观察时，这个线条比较光滑，没有台阶效果。

理解了传送给绘图函数的坐标系统之间的关系之后，查看在绘图界面上得到的结果，将很容易确定哪些像素受到调用绘图函数的影响。

在绘图时，常常用 3 种结构指定坐标：Point, Size 和 Rectangle。

1. Point

GDI+使用 Point 表示一个点。这是一个二维平面上的点——一个像素的表示方式。许多 GDI+函数例如 DrawLine()，以 Point 作为其参数。声明和构造 Point 的代码如下所示：

```
Point p = new Point(1, 1);
```

有一些公共属性可以获得和设置 Point 的 X 和 Y 坐标。

2. Size

GDI+使用 Size 表示一个尺寸(像素)。Size 结构包含宽度和高度。声明和构造 Size 的代码如下所示：

```
Size s = new Size(5, 5);
```

有一些公共属性可以获得和设置 Size 的宽度和高度。

3. Rectangle

GDI+在许多不同的地方使用这个结构，以指定矩形的坐标。Point 结构定义矩形的左上角，Size 定义其大小。Rectangle 有两个构造函数。一个构造函数的参数是 X 坐标、Y 坐标、宽度和高度，另一个构造函数的参数是 Point 和 Size 结构，声明和构建 Rectangle 的两个范例如下所示：

```
Rectangle r1 = new Rectangle(1, 2, 5, 6);

Point p = new Point(1, 2);
Size s = new Size(5, 6);
Rectangle r2 = new Rectangle(p, s);
```

有一些公共属性可以获得和设置 Rectangle 的 4 个点和大小。另外，还有其他属性和方法可以完成诸如测试矩形是否为空，确定矩形是否与另一个矩形相交，提取两个矩形的相交部分，合并两个矩形等工作。详细内容参见.NET Framework 参考资料中的 Rectangle 结构。

4. GraphicsPaths

还有两个更重要的数据类型可以用作 GDI+中许多绘图函数的参数。GraphicsPath 类表示一系列连接的线条和曲线。在构造一条路径时，可以添加线条、Bezier 曲线、圆弧、饼形图、多边形和矩形等。在构造一条复杂的路径后，可以用一个操作绘制路径：调用 DrawPath()。可以调用 FillPath()填充路径。

使用一个点数组和 PathTypes 构造 GraphicsPath，PathTypes 是一个 byte 数组，其中的每个元素对应于点数组中的每一个元素，并给出了路径如何通过这些点来构造的其他信息。例如，如果点是路径的起始点，那么这个点的路径类型就是 PathPointType.Start。如果点是两个线条的连接点，那么这个点的路径类型就是 PathPointType.Line。如果点用于构造一条从前一点到后一点之间的 Bezier 曲线，路径类型就是 PathPointType.Bezier。

下面的代码用 4 条线段创建一个图形路径：

```
GraphicsPath path;
path = new GraphicsPath(new Point[]{
        new Point(10, 10),
        new Point(150, 10),
        new Point(200, 150),
        new Point(10, 150),
        new Point(200, 160)
        }, new byte[] {
        (byte)PathPointType.Start,
        (byte)PathPointType.Line,
        (byte)PathPointType.Line,
        (byte)PathPointType.Line,
        (byte)PathPointType.Line
        });
e.Graphics.DrawPath(Pens.Black, path);
```

使用这条路径绘制的线段如图 16-7 所示。

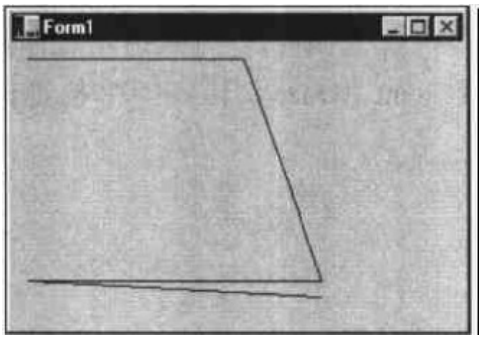


图　16-7

构造这条路径的代码有点复杂。GraphicsPath 的构造函数带有两个参数。第一个参数是一个构造路径的 Point 数组，它使用 C#语法声明数组和初始化数组。给每个点调用构造函数：

```
new Point[]{
```

```
        new Point(10, 10),
        new Point(150, 10),
        new Point(200, 150),
        new Point(10, 150),
        new Point(200, 160)
    }
```

第二个参数是一条构建路径的 byte 数组：

```
    new byte[] {
        (byte)PathPointType.Start,
        (byte)PathPointType.Line,
        (byte)PathPointType.Line,
        (byte)PathPointType.Line,
        (byte)PathPointType.Line
    }
```

最后，在上面的范例中调用 DrawPath()方法：

```
    e.Graphics.DrawPath(Pens.Black, path);
```

5. Regions

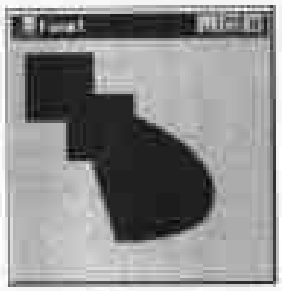
Region 类是一个复杂的图形，由矩形和路径组成。在构造了一个 Region 后，就可以使用 FillRegion()方法绘制该区域。

下面的代码创建一个区域，给它添加一个 Rectangle 和一个 GraphicsPath，再用蓝色填充该区域：

```
    Rectangle r1 = new Rectangle(10, 10, 50, 50);
    Rectangle r2 = new Rectangle(40, 40, 50, 50);
    Region r = new Region(r1);
    r.Union(r2);

    GraphicsPath path = new GraphicsPath(new Point[] {
            new Point(45, 45),
            new Point(145, 55),
            new Point(200, 150),
            new Point(75, 150),
            new Point(45, 45)
            }, new byte[] {
            (byte)PathPointType.Start,
            (byte)PathPointType.Bezier,
            (byte)PathPointType.Bezier,
            (byte)PathPointType.Bezier,
            (byte)PathPointType.Line
            });
```

```
r.Union(path);
e.Graphics.FillRegion(Brushes.Blue, r);
```



该代码将绘制如图 16-8 所示的图形。

构造区域的代码有一些复杂，但上述范例中最复杂的部分是如何构造组成区域的路径，上面的范例已经说明了如何构造这些路径。构造区域包括构造矩形和路径，之后调用 Union()方法。如果决定使矩形和路径相交，就可以用 Intersection()方法替代 Union()方法。

在 GDI+的简介中没有路径和区域的更多信息，所以本章不深入讨论它们。至于更多的信息，可参阅.NET Framework 参考资料中的 GraphicsPath 类和 Region 类。

16.1.4 颜色

GDI+中的许多绘图操作都涉及到颜色。在绘制线条或矩形时，都需要指定使用什么颜色。

在 GDI+中，颜色封装在 Color 结构中。把红、绿、蓝色值传送给 Color 结构的一个函数，就可以创建一种颜色，但几乎从来不需要创建颜色。Color 结构包含大约 150 个属性，提供了大量的预置颜色。忘掉红、绿、蓝、黄和黑色吧！如果需要以 LightGoldenrodYellow 或 LavenderBlush 颜色绘制图形，都有预定义的颜色供我们使用！声明一个 Color 类型的变量，用 Color 结构中的一种颜色初始化它，如下所示：

```
Color redColor = Color.Red;
Color anotherColor = Color.LightGoldenrodYellow;
```

下面几乎已经准备好绘图了，但绘图前有两点需要注意。

颜色可以用两种不同的方式来表示，一种是 RGB，如上所述。另一种是把颜色分解为 3 种组件：色调、饱和度和亮度。Color 结构包含完成分解颜色的实用方法：GetBrightness(), GetHue()和 GetSaturation()。

可以使用 Paint 应用程序来使用颜色。从菜单中选择 Colors | Edit Colors，单击 Define Custom Colors 按钮，打开一个对话框，在其中可以使用鼠标来选择颜色，查看该颜色的 RGB 值。也可以获得该颜色的色调、饱和度和亮度值，还可以直接输入 RGB 值，查看得到的颜色。

GDI+中的颜色还有第 4 个组件：Alpha 组件。使用这个组件可以设置颜色的不透明度，以便创建淡入/淡出效果，例如 Windows 2000 中的菜单效果。使用 Alpha 组件超出了本书的范围。

16.2 使用 Pen 类绘制线条

第一个范例是绘制线条。我们使用 Pen 类绘制线条，该类定义了代码绘图时的颜色、线宽和线条的样式。颜色和线宽属性是很显然的，而线条的样式表示该线条是否为实线，或者由短线和点组成。Pen 类在 System.Drawing 名称空间中。

试试看：Pen 范例

启动 Visual Studio.NET，创建一个新的 C# Windows Application。创建新工程时，Visual

Studio 还会为工程创建一个空白的新窗体 Form1。如前面几章所述，模块 Form1.cs 包含这个窗体的代码。右击该窗体，从弹出的菜单中选择 View Code。另外，还可以在 Solution Explorer 中右击 Form1.cs，从弹出的菜单中选择 View Code。在 Form1.cs 中添加如下代码：

```
protected override void OnPaint(PaintEventArgs e)
{
    Graphics g = e.Graphics;

    using (Pen blackPen = new Pen(Color.Black, 1))
    {
        for (int y = 0; y < ClientRectangle.Height;
                y += ClientRectangle.Height / 10)
        {
            g.DrawLine(blackPen, new Point(0, 0),
                    new Point(ClientRectangle.Width, y));
        }
    }
}
```

现在按下 F5 键，编译并运行代码，得到如图 16-9 所示的窗口。

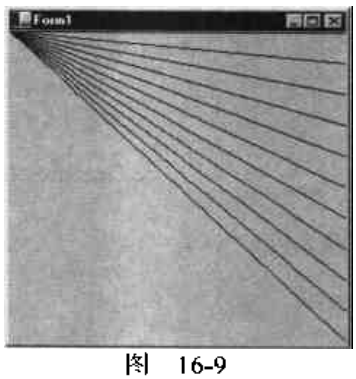


图　16-9

代码的说明

本章前面介绍了 Graphics 类。在这段代码中，首先要在这个方法中声明 Graphics 类的一个实例，初始化 PaintEventArgs 类中的变量：

```
Graphics g = e.Graphics;
```

注意，因为传送了 Graphics 对象的引用，而没有创建它，所以不需要(也不应)对它手动调用 Dispose()。这个范例使用了资源占用量非常大的 Pen 对象，所以要把其余代码封装到 using 块中，如上所述，确保只要有可能，就删除该对象。

在构造钢笔时，把钢笔的颜色和宽度作为参数传送给构造函数。在本例中，颜色为黑色，线宽为 1。下面这行代码构造了钢笔：

```
using (Pen blackPen = new Pen(Color.Black, 1))
```

所绘图的每个窗口都有一个客户区域，它是一个矩形，位于边框的内部，定义了绘图的区域。可以利用 ClientRectangle 获取客户区域，ClientRectangle 是窗体的一个只读的公共属性(继承于 Control)。它包含窗口中所绘制的客户区域的大小(即宽度和高度)。下面的代码引用了 ClientRectangle 属性：

```
for (int y = 0; y < ClientRectangle.Height;
        y += ClientRectangle.Height / 10)
```

在绘制每个线条时，都要传送刚才创建的钢笔以及线条的起始点和终止点：

```
g.DrawLine(blackPen, new Point(0, 0),
        new Point(ClientRectangle.Width, y));
```

提示：

对 Pen 对象应总是调用 Dispose()。

与 Graphics 对象一样，在用完 Pen 对象后，就应调用 Dispose()，或者使用 using 结构，这是很重要的，否则应用程序就可能耗尽 Windows 资源。

在本例中，构造了一个 Pen。但获得 Pen 有更简单的方式。Pens 类包含的属性可以创建大约 150 种钢笔，每个钢笔都有前面介绍的预定义颜色。下面的范例版本与前面的范例工作方式一样，但这次是构造一个 Pen，我们是在 Pens 类中创建该 Pen 的：

```
protected override void OnPaint(PaintEventArgs e)
{
   for (int y = 0; y < ClientRectangle.Height;
         y += ClientRectangle.Height / 10)
   {
      e.Graphics.DrawLine(Pens.Black, new Point(0, 0),
                          new Point(ClientRectangle.Width, y));
   }
}
```

在这里，没有创建 Pen，所以不需要调用 Dispose()。

Pen 类有许多功能。可以创建一个绘制虚线的钢笔，创建线宽比 1 个像素宽的钢笔。Pen 类有一个属性 Alignment，该属性可以定义钢笔是否从左向右(或从上到下)绘制指定的线条。设置 StartCap 和 EndCap 属性可以指定线条的末端有箭头、菱形、方框或圆形。甚至可以使用 CustomStartCap 和 CustomEndCap 属性编写定制的起始端和终止端形状。在学习了图像后，就可以了解如何用 Pen 指定 Brush，以便使用位图来绘制图像，而不是使用一种单色。更多的信息参见.NET Framework 参考资料中的 Pen 类。

16.3 使用 Brush 类绘制图形

下一个范例使用 Brush 类绘制图形，例如矩形、椭圆、饼形图和多边形。Brush 类是一个抽象的基类。要实例化一个 Brush 对象，应使用派生于 Brush 的类，例如 SolidBrush, TextureBrush 和 LinearGradientBrush。Brush 类在 System.Drawing 名称空间中，但 TextureBrush 和 LinearGradientBrush 类在 System.Drawing.Drawing2D 名称空间中。下面是这些类的作用：

- SolidBrush 用一种单色填充图形。
- TextureBrush 用一个位图填充图形。在构造这个画笔时，还指定了一个边框矩形和一个填充模式。边框矩形指定画笔使用位图的哪一部分——可以不使用整个位图。填充模式有许多选项，包括平铺纹理的 Tile、TileFlipX, TileFlipY 和 TileFlipXY，它们指定连续平铺时翻转图像。使用 TextureBrush 可以创建出非常有趣和富有想像力的效果。
- LinearGradientBrush 封装了一个画笔，该画笔可以绘制两种颜色渐变的图形，其中第一种颜色以指定的角度逐渐过渡到第二种颜色。角度则可以根据程度来指定。0° 表示颜色从左向右过渡。90° 表示颜色从上到下过渡。

还有一种画笔 PathGradientBrush，它可以创建精细的阴影效果，其中阴影从路径的中心趋向路径的边界。这种画笔可以让人想起用彩笔绘制的阴影地图，当我们还是孩子时，会在不同的州或国家之间的边界上涂上较暗的颜色。这种画笔的更多信息可参见.NET Framework 参考资料中的 PathGradientBrush 类。

提示：

总是要对 Brush 对象调用 Dispose()。

与 Graphics 对象和 Pen 对象一样，对创建的 Brush 对象调用 Dispose()，或者使用 using 结构是非常重要的，否则应用程序就会耗尽 Windows 资源。

试试看：Brush 范例

与以前一样，启动 Visual Studio.NET，创建一个新的 C# Windows Application 工程。右击窗体，从弹出的菜单中选择 View Code。找到 Form1 类的构造函数，在对 InitializeComponent() 的调用的后面添加一个对 SetStyle()的调用。SetStyle()是 Form 类的一个函数。修改的构造函数如下所示：

```
public Form1()
{
    //
    // Required for Windows Form Designer support
    //
    InitializeComponent();
    SetStyle(ControlStyles.Opaque, true);

    //
    // TODO: Add any constructor code after InitializeComponent call
    //
}
```

这段代码修改了 Form 类的作用，使之不能自动绘制窗口的背景。如果包括这行突出显示的代码，但不手动绘制窗口的背景，那么在创建窗口时进行的操作就会显示出来，而这并不是我们希望得到的。

在文件的开头添加一个 using 语句，因为在这个范例中，将使用 LinearGradientBrush，它在 Drawing2D 名称空间中。在给 Drawing2D 名称空间添加了 using 指令后，其代码如下所示：

```
using System;
using System.Drawing;
using System.Drawing.Drawing2D;
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;
using System.Data;
```

现在，给类添加一个 OnPaint()方法：

```
protected override void OnPaint(PaintEventArgs e)
{
    Graphics g = e.Graphics;
    g.FillRectangle(Brushes.White, ClientRectangle);
    g.FillRectangle(Brushes.Red, new Rectangle(10, 10, 50, 50));

    Brush linearGradientBrush = new LinearGradientBrush(
            new Rectangle(10, 60, 50, 50), Color.Blue, Color.White, 45);
    g.FillRectangle(linearGradientBrush, new Rectangle(10, 60, 50, 50));

    // Manually call Dispose()
    linearGradientBrush.Dispose();

    g.FillEllipse(Brushes.Aquamarine, new Rectangle(60, 20, 50, 30));
     g.FillPie(Brushes.Chartreuse, new Rectangle(60, 60, 50, 50),
90, 210);
    g.FillPolygon(Brushes.BlueViolet, new Point[] {
                                    new Point(110, 10),
                                    new Point(150, 10),
                                    new Point(160, 40),
                                    new Point(120, 20),
                                    new Point(120, 60),
                                    });
}
```



图　16-10

按下 F5 键，编译和运行代码，得到如图 16-10 所示的窗口。

代码的说明

与 Pens 类一样，Brushes 类包含的属性可以创建大约 150 种画笔，每个画笔都可以使用预定义的颜色。使用这个类可以创建本例中的大多数画笔，但 LinearGradientBrush 例外，因为它是我们自己创建的。

首先调用 FillRectangle()，绘制窗口中客户区域的背景：

```
g.FillRectangle(Brushes.White, ClientRectangle);
```

创建 LinearGradientBrush 要使用一个矩形指定矩形的大小，用两种颜色进行渐变，以及一个角度(本例中该角度为 45°)：

```
Brush linearGradientBrush = new LinearGradientBrush(
        new Rectangle(10, 60, 50, 50), Color.Blue, Color.White, 45);
g.FillRectangle(linearGradientBrush, new Rectangle(10, 60, 50, 50));
linearGradientBrush.Dispose();
```

在指定画笔的矩形时，使用了一个宽为 50、高为 50 的矩形，它的大小与定义画笔时使用的矩形相同。其结果是画笔的区域正好与要填充的矩形相符合。试着改变在创建画笔时定义的矩形，使其宽度和高度都是 10，看看会发生什么情况。另外，再试着把角度(在上面的代码中

定义为 45°)改成另一个值，看看结果有什么变化。

16.4 使用 Font 类绘制文本

下一个范例使用 Font 类绘制文本。Font 类封装了字体的 3 个主要特性：字体系列、字体大小和字体样式。Font 类在 System.Drawing 名称空间中。

根据.NET 文档说明，字体系列“把一组有类似基本设计的类型提炼出来”，这是一种把字体系列看作是像 Courier, Arial 或 Times New Roman 这样的东西的奇异方式。

字体的 Size 属性表示字体类型的大小。但是，在.NET Framework 中，这个 Size 并不仅仅是点的大小。它可以是点的大小，但通过 Unit 属性可以改变 GraphicsUnit 属性，Unit 定义了字体的测量单位。记住，一个点等于 1/72 英寸，所以 10 点的字体有 10/72 英寸高。在 GraphicsUnit 枚举中，可以把字体的大小指定为：

- 点的大小
- 显示大小(1/75 英寸)
- 文档 (1/300 英寸)
- 英寸
- 毫米
- 像素

这表示在指定字体的大小时有很大的灵活性。如果要编写一个文本绘制例程，该例程需要以可接受的方式在高分辨率的显示器、低分辨率的显示器和打印机上工作，就可以利用这个特性。

在绘制文本时，如果指定了一个字体以及绘图界面，常常需要知道文本字符串的宽度(像素)，显然，这就是为什么不同的字体会对字符串的宽度(像素)有不同的影响的原因：小字体的宽度需要使用比较少的像素。了解绘图界面也是很重要的，因为不同绘图界面的像素分辨率是不同的。一般情况下，屏幕上每英寸有 72 个像素。打印机上每英寸有 300 个像素、600 个像素，甚至更多。使用 Graphics 对象的 MeasureString()方法可以计算出给定字体的字符串宽度。下面的一些样例代码说明了如何获取字符串的宽度和高度，之后绘制一个黑色矩形，再在矩形中绘制蓝色的字符串：

```
String str = "This is a string";
SizeF size = g.MeasureString(str, Font);
g.DrawRectangle(Pens.Black, 0, 0, size.Width, size.Height);
g.DrawString(str, Font, Brushes.Blue, new RectangleF(0, 0,
        size.Width, size.Height));
```

字体的 Style 属性表示该类型是否为斜体、黑体、删除线或下划线。

提示：

对 Font 对象总是要调用 Dispose()。

对创建的 Font 对象调用 Dispose()或使用 using 结构是很重要的，否则应用程序就可能耗尽

Windows 资源。

在绘制文本时，使用一个 Rectangle 指定要绘制文本的边框坐标。一般情况下，这个矩形的高度应是字体的高度或字体高度的倍数。只有在使用剪切的文本绘制某些特殊效果的文本时，高度才有可能是其他值。

StringFormat 类封装了文本布局信息，包括对齐和行间距信息。下面的范例说明了如何使用 StringFormat 类使文本向右对齐和居中。

试试看：字体范例

与以前一样，启动 Visual Studio.NET，创建一个新的 C# Windows Application 工程，右击窗体，从弹出的菜单中选择 View Code。找到 Form1 类的构造函数，在对 InitializeComponent() 的调用的后面添加一个对 SetStyle()的调用。还需要改变窗口的边框，给要显示的文本以足够的空间。修改后的构造函数如下所示：

```
public Form1()
{
    //
    // Required for Windows Form Designer support
    //
    InitializeComponent();
    SetStyle(ControlStyles.Opaque, true);
    Bounds = new Rectangle(0, 0, 500, 300);

    //
    // TODO: Add any constructor code after InitializeComponent call
    //
}
```

现在。给类添加一个 OnPaint()方法：

```
protected override void OnPaint(PaintEventArgs e)
{
    Graphics g = e.Graphics;
    int y = 0;

    g.FillRectangle(Brushes.White, ClientRectangle);

    // Draw left justified text
    Rectangle rect = new Rectangle(0, y, 400, Font.Height);
    g.DrawRectangle(Pens.Blue, rect);
    g.DrawString("This text is left justified.", Font,
                 Brushes.Black, rect);
    y += Font.Height + 20;
```

```
        // Draw right justified text
        Font aFont = new Font("Arial", 16, FontStyle.Bold | FontStyle.Italic);
        rect = new Rectangle(0, y, 400, aFont.Height);
        g.DrawRectangle(Pens.Blue, rect);

        StringFormat sf = new StringFormat();
        sf.Alignment = StringAlignment.Far;
        g.DrawString("This text is right justified.", aFont, Brushes.Blue,
                     rect, sf);
        y += aFont.Height + 20;

        // Manually call Dispose()
        aFont.Dispose();

        // draw centered text
        Font cFont = new Font("Courier New", 12, FontStyle.Underline);
        rect = new Rectangle(0, y, 400, cFont.Height);
        g.DrawRectangle(Pens.Blue, rect);
        sf = new StringFormat();
        sf.Alignment = StringAlignment.Center;
        g.DrawString("This text is centered   and underlined.", cFont,
                     Brushes.Red, rect, sf);
        y += cFont.Height + 20;

        // Manually call Dispose()
        cFont.Dispose();

        // Draw multiline text
        Font trFont = new Font("Times New Roman", 12);
        rect = new Rectangle(0, y, 400, trFont.Height * 3);
        g.DrawRectangle(Pens.Blue, rect);
        String longString = "This text is much longer, and drawn ";
        longString += "into a rectangle that is higher than ";
        longString += "one line, so that it will wrap.   It is ";
        longString += "very easy to wrap text using GDI+.";
        g.DrawString(longString, trFont, Brushes.Black, rect);

        // Manually call Dispose()
        trFont.Dispose();
    }
```

按下 F5 键，编译和运行代码，得到如图 16-11 所示的结果。

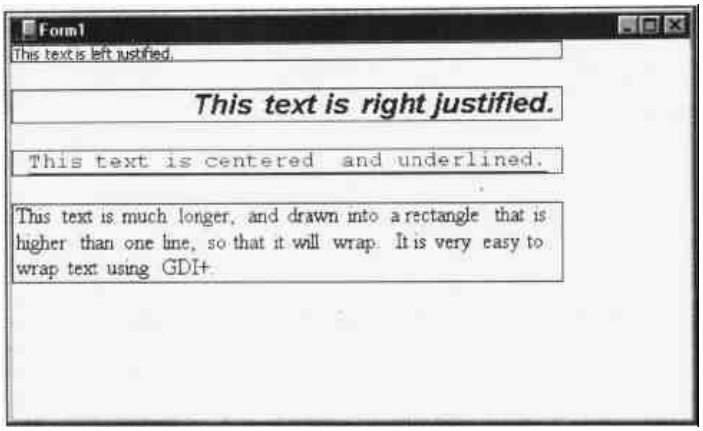


图 16-11

代码的说明

这个范例包含几个最常见的文本绘制操作。

与往常一样，为简单起见，把对 Graphics 对象的引用赋给一个局部变量，并把窗口的背景设置为白色。

在绘制文本时，计算文本的边框矩形。使用 Height 属性获得字体的高度。为了便于说明，把这个矩形绘制为蓝色，这样文本的边框矩形就非常清晰。在绘制文本时，需要提供文本、字体、画笔和边框矩形：

```
// Draw left justified text
Rectangle rect = new Rectangle(0, y, 400, Font.Height);
g.DrawRectangle(Pens.Blue, rect);
g.DrawString("This text is left justified.", Font,
             Brushes.Black, rect);
```

我们仅指定了文本的边框矩形。字体的基线是使用该字体的大多数字符所在的假想线。GDI+和字体本身决定基线在什么位置。我们不能控制基线。

在绘制文本时，把一个画笔传送给 DrawString()函数。在本例中，仅传送了一个单色画笔。也可以传送其他类型的画笔，例如渐变画笔。在下一节介绍图像后，就将解释如何使用 TextureBrush 绘制文本。

在本例中第一次绘制文本时，使用了该窗体的默认字体。这个字体在继承了 Control 的 Font 属性中指定：

```
g.DrawString("This is a left justified string.", Font,
             Brushes.Black, rect);
```

在本例中，第二次绘制文本时，我们创建了 Font 的一个新实例：

```
Font aFont = new Font("Arial", 16, FontStyle.Bold | FontStyle.Italic);
```

这个范例不仅说明了如何创建字体的新实例，还说明了如何给字体指定样式。此时样式可以是黑体或斜体。

这个范例还说明了如何创建 StringFormat 对象，以便绘制右对齐和居中的文本。在 GDI+

中，右对齐的文本是指 Far 对齐方式的文本，左对齐文本是指 Near 对齐方式的文本：

```
StringFormat sf = new StringFormat();
sf.Alignment = StringAlignment.Far;
```

最后，绘制一些多行文本。使用 GDI+时，这个任务再容易不过了。我们只需要指定一个矩形，其宽度小于字符串的长度(像素)，而高度则足以绘制多行文本即可。本例要使其高度等于字体高度的 3 倍。

16.5　使用图像进行绘制

图像在 GDI+中有许多用途。当然，可以在窗口中绘制图像，也可以用图像创建画笔(TextureBrush)，再绘制用该图像填充的图形。还可以从 TextureBrush 中创建钢笔，使用图像绘制线条。在绘制文本时，可以提供一个 TextureBrush，这样就可以使用该图像绘制文本了。Image 类在 System.Drawing 名称空间中。

图像另一个非常重要的用途是双倍缓冲的图形编程技巧。有时要创建的图形非常精细复杂，即使使用目前运行速度最快的机器，也需要很长时间才能绘制出来。观察图形在屏幕中一点一点地绘制出来，并不是一件令人愉快的事。这类应用程序有映射应用程序和复杂的 CAD/CAM 应用程序。在这个技巧中，并不在把图形绘制在窗口中，而是绘制到一个图像中。在完成了图像的绘制后，再把该图像绘制到窗口中。这个技巧就称为双倍缓冲。一些其他的绘图技巧还涉及到在多个图层上绘图，即首先绘制背景，再在背景的上面绘制对象，最后在对象的上面绘制文本。如果这个图形直接在屏幕上绘制，用户就会看到一个闪烁的效果。双倍缓冲可以消除这种闪烁效果。

Image 本身是一个抽象类，它有两个子类：Bitmap 和 Metafile。

Bitmap 类用于一般的图像，有高度和宽度属性。本节的范例就使用这个类。从文件中加载一个 Bitmap 图像，并绘制它。也可以从该类中创建画笔，再使用该画笔创建一个钢笔，以绘制线条，也可以使用该画笔绘制文本。

在本章的最后将介绍 GDI+的高级功能，此时会详细论述 Metafile 类。

提示：

对 Image 对象总是要调用 Dispose()。

对创建的 Image 对象调用 Dispose()或使用 using 结构是很重要的，否则应用程序就可能耗尽 Windows 资源。

位图有几个可能的来源。可以从文件中加载位图，位图也可以来自打开的流，还可以从另一个现有的图像中创建位图。位图可以创建为空白的图像，以便在其上绘图。在从文件中读取图像时，该图像可以是 JPEG, GIF 或 BMP 格式。

试试看：图像范例

下面是一个非常简单的范例，从文件中读取一个图像，并在窗口中绘制它。与以前一样，

启动 Visual Studio.NET ，创建一个新的 C# Windows Application 工程。右击窗体，从弹出的菜单中选择 View Code。需要在 Form1 类中声明一个私有变量，在从文件中读取图像后，就把该图像保存到私有变量中。声明了组件变量后，添加 theImage 声明，如下所示：

```
public class Form1 : System.WinForms.Form
{
    /// <summary>
    ///    Required designer variable.
    /// </summary>
    private System.ComponentModel.Container components;
    private Image theImage;
```

修改构造函数，如下所示：

```
public Form1()
{
    //
    // Required for Windows Form Designer support
    //
    InitializeComponent();
    SetStyle(ControlStyles.Opaque, true);
    theImage = new Bitmap("swipes.jpg");
    //
    // TODO: Add any constructor code after InitializeComponent call
    //
}
```

给类添加 OnPaintEvent()方法：

```
protected override void OnPaint(PaintEventArgs e)
{
    Graphics g = e.Graphics;

    g.DrawImage(theImage, ClientRectangle);
}
```

最后，需要删除存储在类的一个成员变量中的 Image。修改 Form 类的 Dispose()方法，如下所示：

```
protected override void Dispose( bool disposing )
{
    if( disposing )
    {
        theImage.Dispose();
        if (components != null)
        {
            components.Dispose();
```

```
        }
    }
    base.Dispose( disposing );
}
```

在建立这个类之前，必须获取图像，并把它放在与可执行文件相同的目录下。在默认情况下，这个目录是 Visual Studio 工程所在目录的孙目录。例如，如果在目录 ImageExample 下建立这个范例，可执行文件就在 ImageExample\bin\Debug 目录下。我们应在该目录下放置 JPG 文件。

现在，按下 F5 键，编译和运行代码，得到如图 16-12 所示的结果。



图 16-12

代码的说明

在构造函数中，我们实例化了一个 Bitmap 对象，并把它赋给前面声明的 Image 变量。

接着在 OnPaint()事件中绘制图像。在绘制图像时，把 Rectangle 作为一个参数传递给 DrawImage()方法。如果图像的大小与传递给 DrawImage()的矩形大小不同，GDI+会自动重新设置图像大小，使其与指定的矩形大小相同。迫使 GDI+不重新设置图像大小的一种方式是从 Width 和 Height 属性中获取图像的大小，再把图像的大小传递给 DrawImage()方法。

16.5.1　使用纹理画笔进行绘图

下面修改范例，从图像中创建一个 TextureBrush，用该画笔绘制一个椭圆。我们将使用上一个范例，对它进行一些修改。

试试看：纹理画笔(范例 1)

在 Form1 类中添加另一个 Image 变量的声明：

```
public class Form1 : System.WinForms.Form
{
    /// <summary>
    ///     Required designer variable.
    /// </summary>
```

```
private System.ComponentModel.Container components;
private Image theImage;
private Image smallImage;
```

在构造函数中，从该 theImage 中创建一个 smallImage。在创建 small Image 时，指定一个矩形，其大小为 smallImage 的 1/10。这将创建原图像的一个小版本：

```
public Form1()
{
    //
    // Required for Windows Form Designer support
    //
    InitializeComponent();
    SetStyle(ControlStyles.Opaque, true);
    theImage = new Bitmap("swipes.jpg");
    smallImage = new Bitmap(theImage,
                    new Size(theImage.Width / 10, theImage.Height / 10));

    //
    // TODO: Add any constructor code after InitializeComponent call
    //
}
```

用下面的代码替换 OnPaint() 事件：

```
protected override void OnPaint(PaintEventArgs e)
{
    Graphics g = e.Graphics;

    g.FillRectangle(Brushes.White, ClientRectangle);

    Brush tBrush = new TextureBrush(smallImage, new Rectangle(0, 0,
                        smallImage.Width, smallImage.Height));
    g.FillEllipse(tBrush, ClientRectangle);
    tBrush.Dispose();
}
```

最后，需要删除存储在类的成员变量中的两个图像。修改 Form 类的 Dispose()方法，如下所示：

```
protected override void Dispose( bool disposing )
{
    if( disposing )
    {
        theImage.Dispose();
        smallImage.Dispose();
```

```
            if (components != null)
            {
                components.Dispose();
            }
        }
        base.Dispose( disposing );
    }
```

运行这个应用程序，得到如图 16-13 所示的窗口。

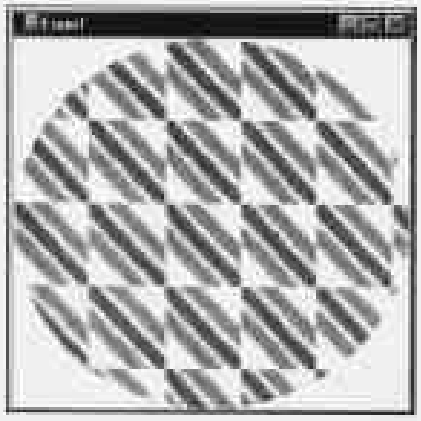


图　16-13

代码的说明

在创建 TextureBrush 时，把一个矩形传递给构造函数，以指定图像的哪一部分用于创建该画笔。在本例中，指定使用整个图像。无论使用 TextureBrush 绘制什么图像，都使用这个位图来绘制，而不是使用单色。

这个范例的大多数代码已经在本章前面解释过了。不同的地方是调用了 Graphics 类的 FillEllipse()方法，传递新创建的纹理画笔，ClientRectangle 在窗口中绘制出椭圆：

```
g.FillEllipse(tBrush, ClientRectangle);
```

16.5.2　使用钢笔绘制图像

试试看：纹理画笔(范例 2)

前面已经创建了一个 TextureBrush，还可以使用该画笔创建一个钢笔。在 TextureBrush 范例 1 中，修改其中的 OnPaint()事件，如下所示：

```
protected override void OnPaint(PaintEventArgs e)
{
    Graphics g = e.Graphics;

    g.FillRectangle(Brushes.White, ClientRectangle);

    Brush tBrush = new TextureBrush(smallImage, new Rectangle(0, 0,
```

```
                    smallImage.Width, smallImage.Height));
        Pen tPen = new Pen(tBrush, 15);
        g.DrawRectangle(tPen, 20, 20, 240, 240);
        tPen.Dispose();
        tBrush.Dispose();
    }
```

运行程序，得到如图 16-14 所示的结果。

这是一个使用钢笔绘图的范例，其中使用画笔创建钢笔，而画笔是使用一个图像创建的。

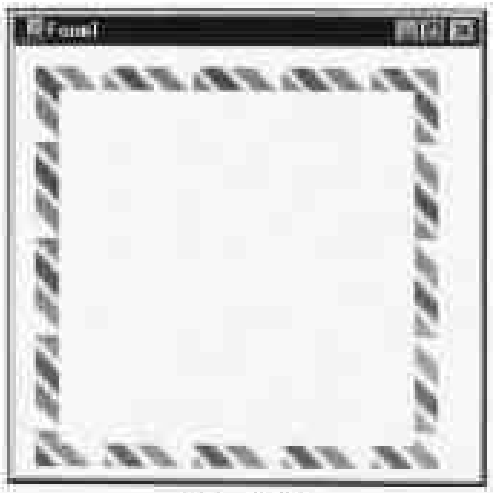


图 16-14

16.5.3 使用图像绘制文本

试试看：纹理画笔(范例 3)

还可以使用 TextureBrush 绘制文本。修改 OnPaint 事件，如下所示：

```
protected override void OnPaint(PaintEventArgs e)
{
    Graphics g = e.Graphics;

    g.FillRectangle(Brushes.White, ClientRectangle);

    Brush tBrush = new TextureBrush(smallImage, new Rectangle(0, 0,
                    smallImage.Width, smallImage.Height));
    Font trFont = new Font("Times New Roman", 32,
                    FontStyle.Bold | FontStyle.Italic);
    g.DrawString("Hello Image", trFont, tBrush, ClientRectangle);
    tBrush.Dispose();
    trFont.Dispose();
}
```

运行程序，得到如图 16-15 所示的结果。



图 16-15

对 DrawString()方法的调用类似于以前对这个方法的调用。它所带的参数是文本、字体、纹理画笔和边框矩形：

```
g.DrawString("Hello Image", trFont, tBrush, ClientRectangle);
```

16.5.4　双倍缓冲

在绘制文本时，会涉及到一个问题，就是用户必须等待相当长的时间才能看到绘制好的图形。如前所述，解决方案是先把文本绘制到图像上，再在完成了所有的绘图操作后，把整个图像绘制到窗口中。

试试看：不使用双倍缓冲的范例

下面的 OnPaint()例程用随机颜色绘制了许多线条。在运行该例程时，可以看到绘制的过程(假定机器的速度不是很快)：

```
protected override void OnPaint(PaintEventArgs e)
{
    Graphics g = e.Graphics;
    Random r = new Random();

    g.FillRectangle(Brushes.White, ClientRectangle);

    for (int x = 0; x < ClientRectangle.Width; x++)
    {
        for (int y = 0; y < ClientRectangle.Height; y += 10)
        {
            Color c = Color.FromArgb(r.Next(255), r.Next(255), r.Next(255));
            using (Pen p = new Pen(c, 1))
            {
                g.DrawLine(p, new Point(0, 0), new Point(x, y));
            }
        }
```

```
        }
    }
```

运行代码，在全部的绘图操作完成后，窗口如图 16-16 所示。



图 16-15

试试看：使用双倍缓冲的范例

如果用下面的代码替换 OnPaint()过程，图形会在一两秒钟内绘制完：

```
protected override void OnPaint(PaintEventArgs e)
{
    Graphics displayGraphics = e.Graphics;
    Random r = new Random();
    Image i = new Bitmap(ClientRectangle.Width, ClientRectangle.Height);
    Graphics g = Graphics.FromImage(i);

    g.FillRectangle(Brushes.White, ClientRectangle);

    for (int x = 0; x < ClientRectangle.Width; x++)
    {
        for (int y = 0; y < ClientRectangle.Height; y += 10)
        {
            Color c = Color.FromArgb (r.Next(255), r.Next(255), r.Next(255));
            Pen p = new Pen(c, 1);
            g.DrawLine(p, new Point(0, 0), new Point(x, y));
            p.Dispose();
        }
    }
    displayGraphics.DrawImage(i, ClientRectangle);
    i.Dispose();
}
```

代码的说明

使用下面的代码，创建高度和宽度与 ClientRectangle 相同的新图像：

```
Image i = new Bitmap(ClientRectangle.Width, ClientRectangle.Height);
```

再使用下面的代码从图像中获取一个 Graphics 对象：

```
Graphics g = Graphics.FromImage(i);
```

所有的绘图操作与前面的代码一样，但它们现在是绘制在图像中，而不是直接绘制在窗口中。最后，在函数的尾部，把图像绘制到窗口中：

```
displayGraphics.DrawImage(i, ClientRectangle);
```

因为线条首先绘制到不可见的图像中，所以必须等待一小段时间，才能看到该图像。

16.6　GDI+的高级功能

前面仅介绍了 GDI+的一部分功能，它还有许多其他的功能，仅一章的篇幅是不能完全包含它们的。但是，为使本章比较完整，下面介绍这些高级功能中的几个。

16.6.1　剪切

剪切功能非常重要，有 3 个方面的原因。

首先，在调用 OnPaint()事件时，除了 Graphics 对象外，该事件还传递了一个剪切矩形。对于简单的绘图例程来说，不需要过多地考虑剪切的矩形，但如果执行一个非常精细的、需要很长时间的绘图例程，在绘图前测试一下这个剪切矩形，就可以减少绘图时间。我们需要绘制任何图形的边框矩形。如果这个边框矩形与剪切矩形不相交，就可以跳过该绘图操作。

图 16-17 显示的一个窗口包含了一个条形的表格，该表格被另一个窗口部分遮挡住了。

在关闭计算器，Windows 操作系统绘制了窗口的边框后，条形表格窗口应如图 16-18 所示。

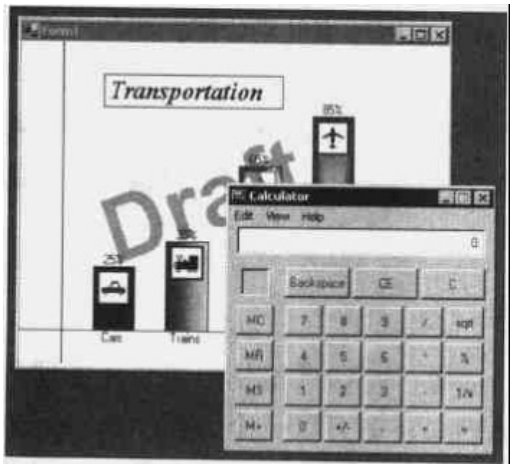


图　16-17

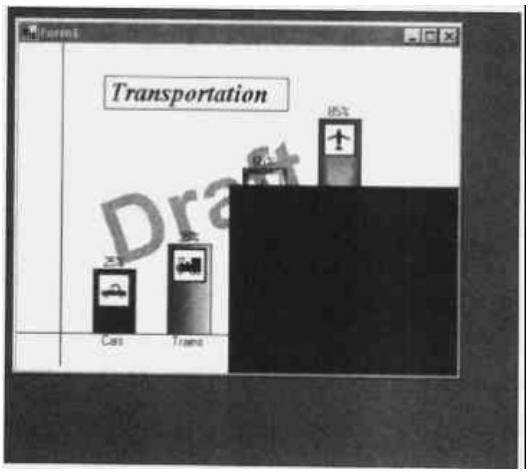


图　16-18

此时，调用条形表格窗口的 OnPaint()事件，把剪切矩形设置为已关闭窗口覆盖的区域，即

图 16-18 中黑色的区域。条形表格窗口现在需要绘制以前被上层窗口覆盖的部分。不需要重新绘制汽车或火车条，实际上，即使 OnPaint()事件试图绘制没有被覆盖的窗口区域，也是不可能的。它绘制的任何图形都会被忽略。条形表格窗口知道汽车栏的边框矩形，可以确定这个矩形是否与窗口的重叠部分相交。在确定了它们不相交之后，绘图例程就不会重新绘制汽车栏这部分窗口。

有时在绘制时，如果需要绘制一个图形的一部分，则绘制整个图形，再把想要的部分剪切下来会比较方便。如果有一个图像，要绘制该图像的一部分，则应设置剪切矩形，再绘制要显示的图像部分，通过剪切区域显示出来，而不应创建一个包含原图像一部分的新图像。在创建一个选取框时，我们使用的就是剪切功能。连续改变绘制文本的区域，同时设置一个剪切区域，就可以创建水平移动文本的效果。

最后，还有一个技巧：即在一个较大图形中，创建一个视口，用户可以移动这个视口，方法是在图形上拖动鼠标，或者操作滚动条。视口也可以根据用户执行的其他操作编程移动。此时，应选择一个剪切区域，并绘制我们要通过该区域查看的窗口部分。

更多的信息请参见 Graphics 类的 Clip 属性。

16.6.2 打印

如第 15 章所述，GDI+完全支持打印，打印类封装在 System.Drawing.Printing 名称空间中。为打印编写图形代码非常类似于为在屏幕上显示图形而编写的代码。我们使用相同的对象 Graphics, Point, Size, Rectangle, Pen, Brush 和 Image 等。但是，其机制有点不同，我们并不是响应 OnPaint 事件，而是创建一个事件处理程序(可以给它随意起一个名字)，把它注册为打印页面的事件处理程序：

```
PrintDocument pd = new PrintDocument(); //Assumes the default printer
pd.PrintPage += new PrintPageEventHandler(this.pd_PrintPage);
```

注册了处理程序后，调用 PrintDocument 的 Print()方法：

```
pd.Print();
```

PrintDocument 类重复调用前面创建的事件处理程序。调用该事件后，把正确的页面绘制到打印机上，这并不是打印技术的所有内容，却是一般情况。

还有其他一些问题需要考虑。打印机的像素分辨率与屏幕的像素分辨率是不同的，在绘图时需要考虑分辨率的问题。一些打印机支持颜色，另一些打印机不支持。可以编写灵活的代码把这些因素都考虑在内，智能地处理各种情况。

16.6.3 打印预览

还可以创建一个打印预览窗口，显示打印页面的内容。打印预览使用的事件处理程序与打印图形的处理程序相同。一旦用户打开了打印预览窗口，就可以一个页面一个页面地查看整个文档。用户还可以缩放页面的任何一部分，查看其细节。屏幕图 16-19 显示的就是打印预览窗口。

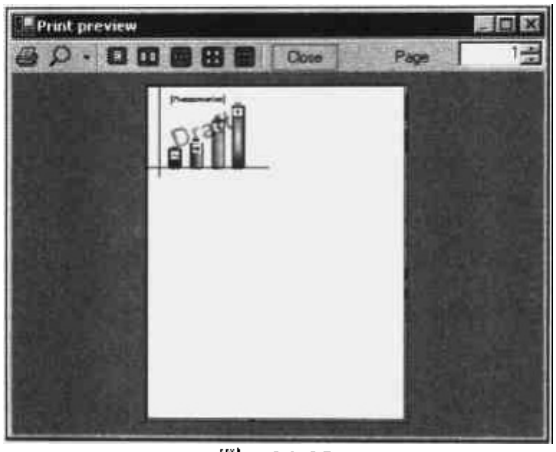


图　16-19

这仅是第 15 章中所介绍的一个主题的总结。

16.6.4　System.Drawing.Drawing2D

这个名称空间中的类提供了二维图形和矢量图形的高级功能。我们可以使用这些类建立复杂的图形和图像处理程序。

矢量图形是一种技巧，程序员利用这种技巧根本不能定位其中的像素，但可以记录多个矢量，表示如从一点画到另一点、在某个位置绘制矩形等操作。开发人员可以应用按比例缩放、旋转和其他转换。应用了这些转换后，所有的操作结果都会立即显示在窗口中。

这个名称空间包括一些高级画笔。前面介绍了 LinearGradientBrush 和 PathGradientBrush，还有一个 HatchBrush，它可以使用一个影线填充模式、前景色和背景色进行绘制。

这个名称空间也包含 Matrix 类，它定义了几何变换。使用这个类可以变换绘图操作，例如使用 Matrix 类可以绘制一个倾斜的椭圆。

这个名称空间还包含 GraphicsPath 类，前面已经介绍过了。使用这个类可以定义一个复杂的路径，一次绘制出整个路径。

16.6.5　System.Drawing.Imaging

这个名称空间中的类提供了高级的图像支持。两个主要的功能是元文件和编码/解码类。

元文件描述了一系列图形操作，这些操作可以记录下来，以便以后回放。Encoder 和 Decoder 类可以扩展 GDI+，以支持任何图像格式。更多的信息请参阅.NET Framework 参考资料中的 System.Drawing.Imaging 选项。

16.7　小结

本章介绍了 System.Drawing 名称空间中的一些类，讨论了 Graphics 类如何封装绘图界面，复习了绘图的机制，在窗口需要重新绘制时，就应调用 OnPaint 事件。

我们探讨了颜色和坐标系统，介绍了用于在绘图界面指定位置和大小的 Point, Size 和 Rectangle 结构。接着列举了一些绘制线条、图形、文本和图像的范例。

总之，本章的主要内容如下：

- Graphics 类
- Color 结构
- 使用 Pen 类绘制线条
- 使用 Brush 类绘制图形
- 使用 Font 类绘制文本
- 使用 Bitmap 类绘制图像
- 绘制到图像上(双倍缓冲)

在处理某些类时，调用 Dispose()是非常重要的，这些类包括：

- Graphics
- Pen
- Brush
- Font
- Image

最后，概述了.NET Framework 中的其他图形功能，并说明了应在.NET Framework 参考文档的什么地方查找这些功能的更多信息。

第17章　部署Windows应用程序

.NET 一个非常突出的特征就是其安装可以通过使用简单的 xcopy 来完成。装配件仅仅包含许多文件，不再需要注册表来存储装配件配置，因此对多个文件进行复制就可以完成安装。

但是，很快就可以知道，有一些原因导致不能为完成 Windows 应用程序的安装而使用 xcopy。对于部署 Windows 应用程序来说，xcopy 仅可以用于最简单的应用程序。对于仅仅安装在少数系统中的小型应用程序而言，xcopy 可以完成许多工作，但是对于大型应用程序或安装在许多系统中的应用程序而言，我们就需要考虑一种更好的安装机制。xcopy 不注册或验证装配件的位置，它也不能使用 Windows Installer Zero Administration Windows(ZAW)功能，这就意味着文件可以随意被重写，而且也没有内置的卸载程序。

注意：

Microsoft 的 Zero Administration Initiative 方便了系统管理员的工作，使得他们可以自动在客户机系统上更新和安装应用程序，并集中管理应用程序。

对于 Windows 应用程序而言，Windows Installer(Windows 安装程序)具有如下优点：

- 安装时，用户可以选择要安装的功能。通过使用通知(advertisement)，也可以在首次需要使用功能时再安装它。
- 如果安装失败，则回滚(rollback)，这样系统就可以返回到安装之前的状态。
- 如果文件损坏，则可以很容易地修复应用程序(即仅仅重新安装损坏的组件)。
- 在 Windows 安装程序文件中，我们可以读取如下的信息：安装什么文件，完成什么样的注册表配置。
- 应用程序可以完全卸载。

在本章，我们要介绍以下内容：

- Visual Studio.NET 中的部署工程类型
- Windodws 安装程序的功能
- 使用 Visual Studio.NET 创建 Windows 安装程序软件包

17.1　部署的含义

部署是在目标系统上安装应用程序的过程。

Microsoft Solutions Framework(http://www.microsoft.com/business/services/mcsmsf.asp)的进程模型定义了开发循环中 4 个阶段的进程模型：展望阶段，阐述了设计解决方案的思想；规划阶段主要用于分析和设计解决方案；开发阶段处理开发工作；巩固阶段不引入新功能，但是却要调试代码，并将产品的测试版和最终版本提交到顾客的站点，如图 17-1 所示。

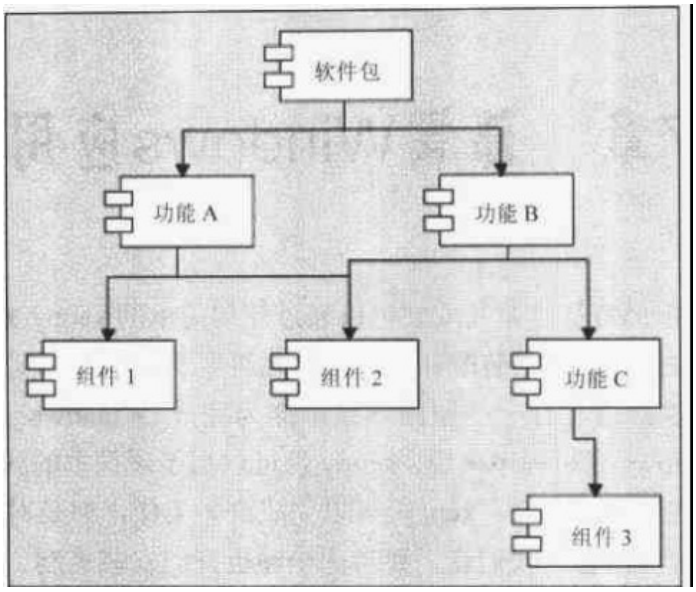


图　17-1

在这 4 个阶段之间，可以看到 4 个主要的标志。这些主要标志并不形成序列，但是对于今天的应用程序开发工作而言，顾客的需求可能变化，在后面阶段，您可能会发现某些程序并不能达到预期的效果。因此在规划阶段也要做一定的开发工作，以获取要使用的原型，这样在开发阶段还可以改变设计方案。

提示：

在进程循环中较早地考虑工程的部署问题很有好处，因为部署战略可以影响应用程序的设计。

17.2　部署工程类型

打开 Visual Studio.NET 的 Add New Project 对话框，选择了 Setup and Deployment Projects 之后，就可以看到如图 17-2 所示的的窗口。

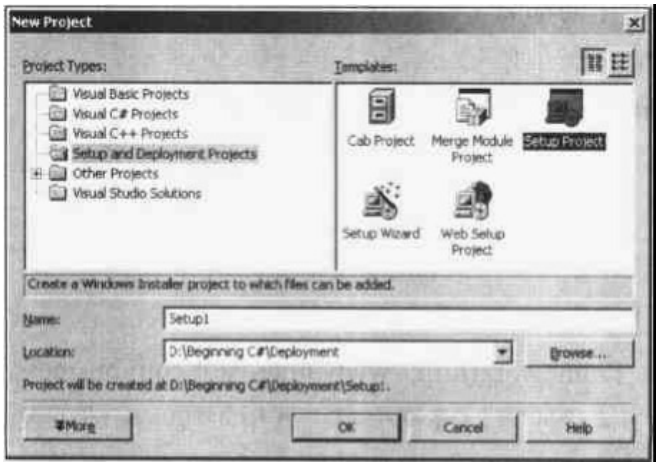


图　17-2

窗口中就有许多工程类型以及它们的作用：

• 有了 Cab Project 模板，就可以创建归档文件(cabinet file)。归档文件可以用于将多个装配件合并到单一的文件中，并进行压缩。因为归档文件可以压缩，所以 Web 客户机可以从服务器上下载较小的文件。

注意：

创建组件超出了本书的范围，因此我们不介绍归档工程的创建问题。您可以阅读《C#高级编程》(由清华大学出版社出版)，查询如何创建.NET 组件，用于从 Web 服务器下载内容的信息。

• Merge Module Project 模板用于创建 Windows Installer 合并模块。合并模块是安装程序文件，可以包括在多个 Microsoft Installer 安装软件包中。对于随多个安装程序一起安装的组件而言，可以创建一个合并模块，以在安装软件包中包括此模块。合并模块的一个范例是.NET 运行时本身：它被发送到合并模块中，因此就可以在应用程序的安装程序软件包中包括.NET 运行时。在范例应用程序中我们要使用一个合并模块。

• 我们要使用 Setup Project 模板。此模板用于创建 Windows 安装程序软件包，这样它就是部署 Windows 应用程序的最好方法。

• Setup Wizard 是可以逐步选择其他模板的。需要自我回答的第一个问题是：您希望创建一个安装程序，以安装应用程序或可以重新分布的软件包吗？实际可以创建 Windows 安装程序软件包、合并模块或 CAB File，这要取决于您的选择。

在此列表中的最后一个模板是 Web Setup Project。在本章我们不会使用此模板，因为本书的主题是开发 Windows 应用程序。对于 Web 应用程序的部署问题，您可以参考《Beginning ASP.NET with C#》(ISBN 1861006152，由 Wrox Press 出版)。

17.3 Microsoft Windows 安装程序结构

在 Windows Installer 之前，程序员必须创建定制的安装程序。不仅仅是构建安装程序有大量烦琐的工作，而且许多程序并不遵循 Windows 规则。通常重写旧版本的系统 DLL，因为安装程序不会检查版本。除了这些之外，在其中复制应用程序文件的目录通常是错误的，例如使用硬编码的目录字符串 C:\Program Files。如果系统管理员改变默认驱动器字母，或使用了操作系统的国际化版本(其中此目录的命名方式各不相同)，安装就会失败。

Windows Installer 的第一个版本随 Microsoft Office 97 发布，它也可以作为可分布式软件包发布，这些可分布式软件包可以包含在其他应用程序软件包中。版本 1.1 新增了对注册 COM+ 组件的支持，版本 1.2 支持 Windows ME 的文件保护机制。版本 1.5 新增了对.NET 装配件和 Windows 64 位版本的支持。而在 Visual Studio.NET 中有 Windows Installer 的新版本：2.0。

17.3.1 Windows Installer 术语

使用 Windows Installer 时，我们必须理解一些 Windows Installer 技术中所使用的术语：软件包、功能和组件。

要注意的第一个要点就是组件，它用于安装程序。

提示：

在 Windows Installer 的环境中，组件与我们在.NET Framework 中所使用的术语“组件”不同。Windows Installer 组件仅仅是一个单一的文件(或者是在逻辑上一致的多个文件)。这些文件是可执行文件、DLL 或简单的文本文件。

从图 17-3 中可以看到，软件包包含一个或多个功能块。软件包是单一的 Microsoft Installer(MSI)数据库。功能是用户对产品功能的观点，由多个功能和组件构成。组件是开发人员对安装的看法；它是安装的最小单元，由多个文件组成。区分功能和组件的原因是单一的组件可以包含多个功能，如图 17-3 中的 Component 2。一个功能不能包含在多个功能中。

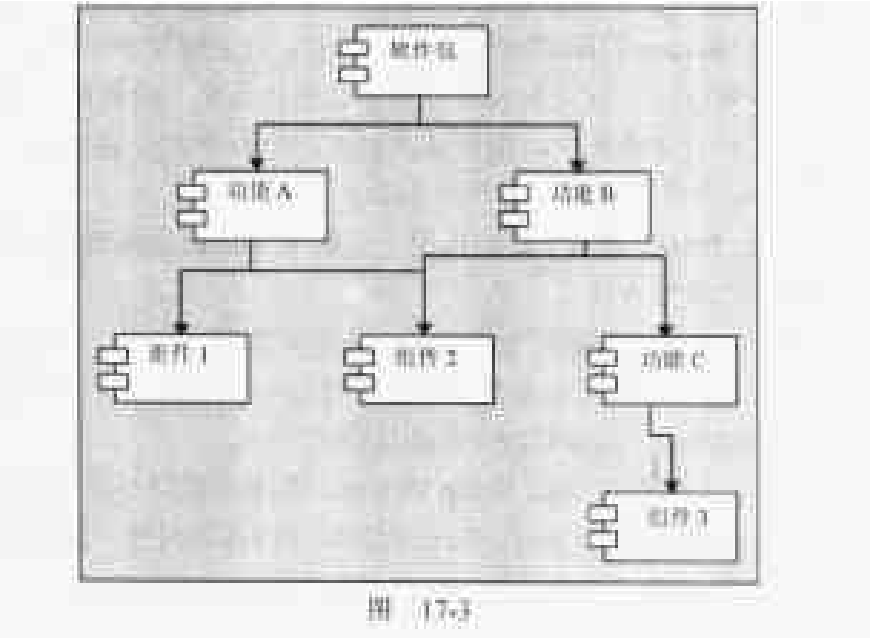


图 17-3

下面介绍一个功能的实际范例：Visual Studio.NET。通过使用在 Control Panel 中的 Add/Remove Programs 选项，我们可以改变在安装之后的 Visual Studio.NET 的安装特征，为此还要单击 Change/Remove 按钮，如图 17-4 所示。



图 17-4

通过单击 Change/Remove 按钮，就打开了 Visual Studio.NET Maintenance 向导。这是研究功能用法的好方法。从图 17-5 中可以看到，Visual Studio.NET 软件包包括功能 Language Tools，.NET Framework SDK，Crystal Reports for Visual Studio.NET，Tools for Redistributing Apps 和 Server Components。功能 Language Tools 具有子功能 Visual Basic.NET，Visual C++ .NET 和 Visual C# .NET 等。

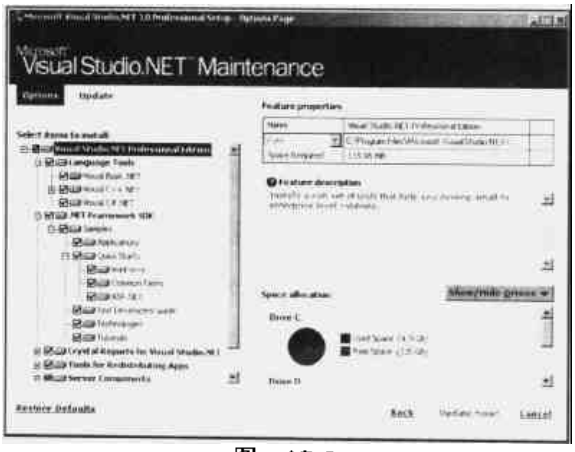


图　17-5

17.3.2　Windows Installer 的优点

Windows Installer 的优点如下：

- 可以安装功能，卸载或通知。有了通知，软件包的功能会在首次使用时安装。或许您在使用 Microsoft Word 时已经注意到了 Windows Installer 的用法。如果未安装 Word 的广告功能，只要您使用此功能，就会自动安装 Word。
- 如果应用程序损坏，它还可以自我修复，为此要使用 Windows Installer 软件包的修复功能。
- 如果安装失败，就会自动回滚。安装失败之后，所有的内容都保持原样：在系统上没有附加的注册表项，没有文件等。
- 使用卸载功能，可以清除所有的文件，注册表项等。这样就可以完全卸载应用程序。也不会留下临时文件，注册表就可以恢复原样。

通过阅读 MSI 数据库文件的表，就可以获取以下信息：复制了什么文件，写入了什么注册表项等。

17.4　为简单的编辑器创建一个安装程序软件包

我们要使用在第 15 章的范例解决方案(Simple Editor)来使用 Visual Studio.NET 创建 Windows 安装程序软件包。当然，您也可以使用按要求开发的其他 Windows Forms 应用程序；您只需要相应地改变一个所使用的名称即可。

17.4.1 规划安装内容

在开始构建安装程序之前，我们必须规划安装内容。首先要考虑一些问题：

- 应用程序需要什么文件？当然是可执行文件和一些组件装配件。无需您标识这些项之间的依赖关系，因为这种依赖关系会自动包括。或许还需要其他一些文件。如文档文件、readme.txt、许可文件、文档模板、图像、配置文件等。我们必须了解所有需要的文件。

对于 Simple Editor 而言，我们所需要的是可执行文件，我们还要包含文件 readme.rtf，license.rtf 和 Wrox Press 的位图，将其显示在安装对话框中。

- 应该使用什么目录？应用程序文件应该安装在 Program Files\Application name 中。Program Files 目录的命名对于操作系统的各个语言变体各有不同。而且，管理员也可以为此应用程序选择不同的路径。无需知道此目录的位置，因为 API 函数调用可以获取此目录。有了此安装程序，我们就可以使用一个特定预定义的文件夹在 Program Files 目录中放置文件。

提示：

任何情况下目录都不应该是硬编码的。有了国际版本，这些目录可以不同的命名！即使应用程序支持 Windows 的 English 版本(实际上不会这样)，System Administrator 也可以将这些目录移动到不同的驱动器中。

Simple Editor 将可执行文件放置在默认的应用程序目录中，除非安装用户选择了不同的路径。

- 用户如何访问应用程序？我们可以在 Start 菜单中为可执行文件设置快捷方式，在桌面放置图标等。如果希望在桌面放置图标，就应该首先考虑用户是否乐意。对于 Windows XP，其原则是尽可能地使桌面干净。

可以从 Start 菜单访问 Simple Editor。

- 什么是分布式媒体？我们希望将安装软件包放置在 CD，软盘或网络共享中吗？
- 我们会对用户提出什么要求？用户会接受许可信息，显示 Readme 文件，并询问安装路径吗？安装要求一些选项吗？

Visual Studio.NET Installer 提供的默认对话框足以满足此范例的要求。我们会请求安装程序的目录(用户可以选择不同于默认路径的路径)，显示 Readme 文件，并请求接受许可协议。

17.4.2 创建工程

在了解了安装软件包的内容之后，我们可以使用 Visual Studio.NET 安装程序来创建安装程序工程，并添加所有应该安装的文件。在第一步中，我们会使用 Project 向导并配置工程。

试试看：创建 Windows Installer 工程

(1) 打开在第 15 章创建的 Simple Editor 工程的解决方案文件。在现有的解决方案中添加安装工程。如果在第 15 章未创建解决方案，可以从 http://www.wrox.com 的 Wrox 网站上下载它。

(2) 使用 File | New | Project 菜单，在解决方案中添加 Setup Project。选择 *Add to Solution* 单选按钮，并单击 OK 按钮，如图 17-6 所示。

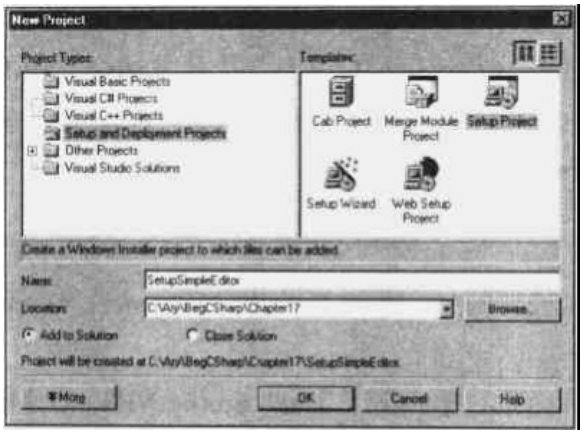


图　17-6

17.4.3　工程属性

到目前为止，我们仅仅拥有一个工程文件可以用于安装解决方案。必须定义要安装的文件。但是我们还必须配置工程属性。为此，就必须知道 Packaging 和 Bootstrapper 选项的含义。

1. 打包

MSI 是启动安装的数据库，但是我们可以定义如何使用如图 17-7 所示的 3 个选项定义要安装的文件。如果在 Visual Studio.NET 菜单中选择 Project | Properties，就打开此对话框。

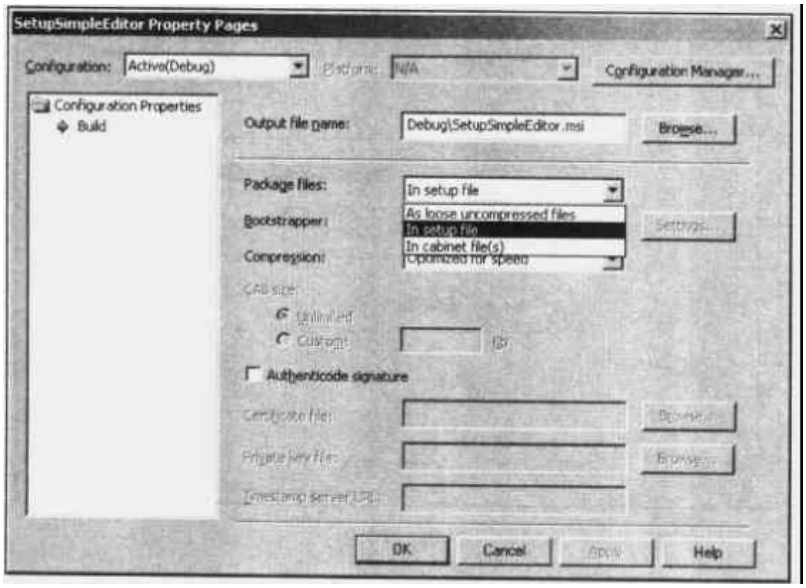


图　17-7

- 有了松散非压缩文件(loose uncompressed file)，所有的程序和数据文件都可以原样存储。不会进行文件压缩。
- In Setup file 意味着所有的文件都被合并和压缩到 MSI 文件中。此选项可以在软件包中被单一的组件重写。如果将所有的文件放置到单一的 MSI 文件中，您就必须注意安装程序要适合于希望使用的工具，比如 CD 或软盘。如果安装的内容太多，超过了单一的软盘的容量，可以试试改变压缩速率，为此在 Compression 列表框中选择 Optimized for size 选项。如果其容量

仍然不适合，则可以选择打包的下一个选项。

● 对文件打包的第三种方法是使用归档文件。在此方法中，MSI 文件用于加载和安装 CAB 文件。使用 CAB 文件，才有可能设置文件大小，使得可以在 CD 或软盘上进行安装(对于从软盘的安装，我们可以设置 1440KB 的安装容量)。

2. Bootstrapper

在同一个对话框中可以设置的另一个选项是 bootstrapper。Bootstrapper 是必须在实际的应用程序运行之前执行的程序。

在要安装应用程序的目标系统上，随 Visual Studio.NET 创建的安装程序软件包要求使用 Windows Installer 版本 1.5。Windows Installer 的 1.5 版本最初是随 Windows XP 发布的。如果您的程序部署在较早版本的系统上，您就需要 bootstrapper，它可以安装 Windows Installer 的新版本。选择 Windows Installer Bootstrapper 就可以在安装程序中包括 bootstrapper，安装时，它就在开始安装目标程序之前安装 Windows Installer 1.5。这种选择所需要的容量大约是 3MB。

如果使用 Web Bootstrapper，我们就必须在 Web 站点上放置 bootstrapper。当在 Properties Editor 中选择了此选项时，就需要下载 URL。安装应用程序的用户就可以使用此 URL，随此选项安装 bootstrapper，在安装软件包中您不需要额外的空间，但是安装应用程序的用户必须可以访问 Internet，如图 17-8 所示。

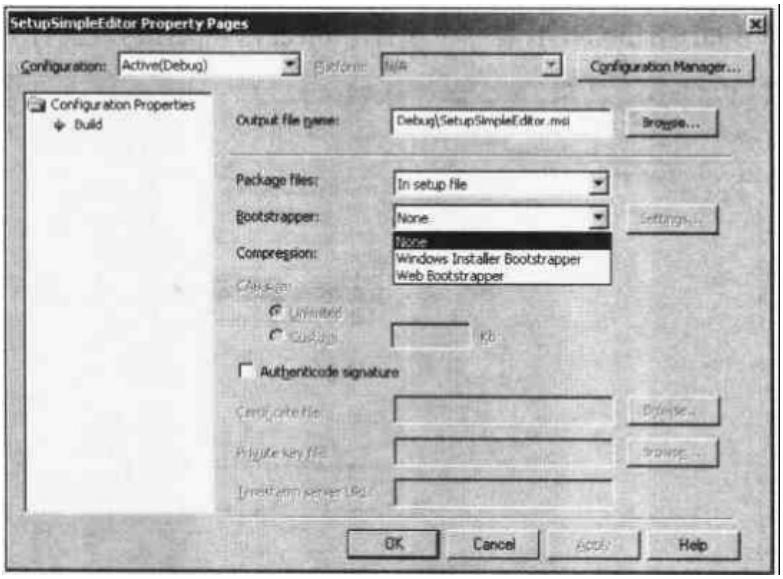


图　17-8

如果选择了 None 选项不安装 bootstrapper，而且在目标系统上未安装 Windows Installer 1.5，则就会在安装时得到图 17-9 所示的的错误消息。



图　17-9

试试看：配置工程

(1) 将属性页上的 Bootstrapper 选项(View | Property Pages)改变为 Windows Installer Bootstrapper，这样应用程序就可以安装在没有 Windows Installer 1.5 的系统上。作者还将输出文件名称改变为 WroxSimpleEditor.msi，如图 17-10 所示。

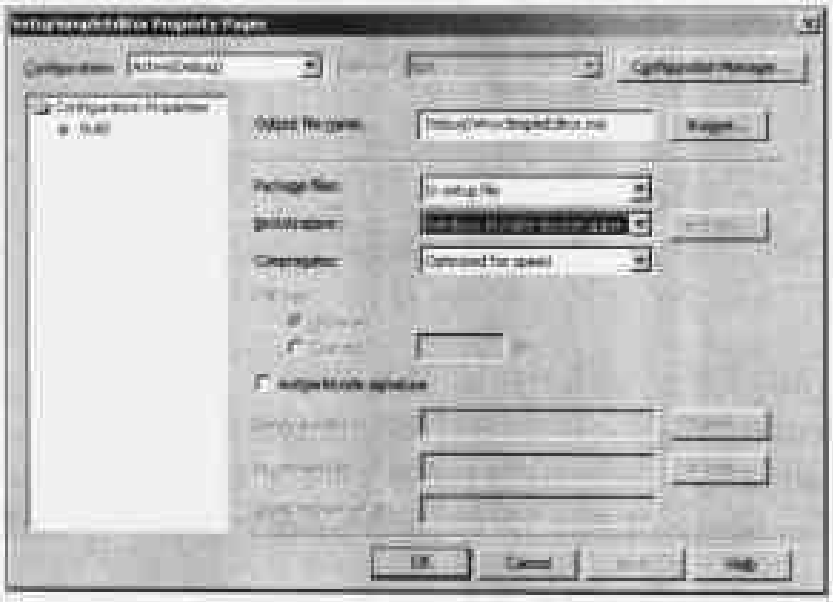


图 17-10

(2) 将工程属性设置为表 17-1 中的值。

表 17-1

属　性	值
Author	Christian Nagel
Description	要打印和编辑文本文件的 Simple Editor
Keywords	Installer，Wrox Press 和 Simple Editor
Manufacturer	Wrox Press
ManufacturerUrl	http://www.wrox.com
Product Name	Simple Editor
SupportUrl	http://p2p.wrox.com
Title	Simple Editor 的安装数据库
Version	1.0.0

17.4.4　安装编辑器

有了 Visual Studio.NET Setup Project，我们就可以得到 6 个编辑器。选择编辑器的方式如下：打开部署工程，选择菜单 View | Editor，如图 17-11 所示。

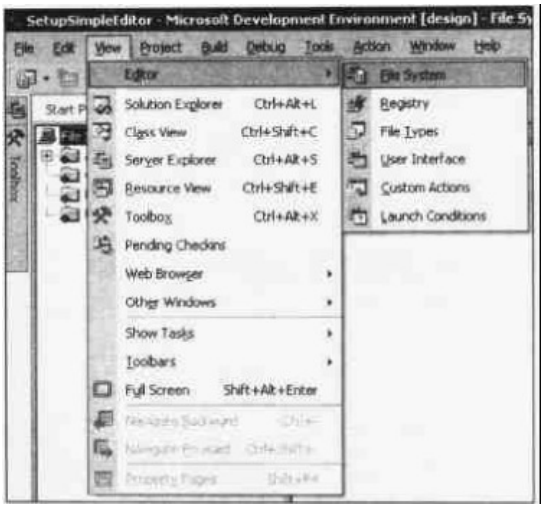


图　17-11

- File System 编辑器用于向安装软件包中添加文件。
- 通过使用 Registry 编辑器，可以为应用程序创建注册表项。
- File Types 编辑器允许注册应用程序的具体文件扩展名(file extentions)。
- 通过使用 User Interface 编辑器，我们可以添加和配置对话框，在安装程序期间显示这些对话框。
- Custom Actions 编辑器允许它在安装和卸载期间启动定制的程序。
- 通过使用 Launch Conditions 编辑器，我们可以规定对应用程序的要求，比如，必须具有.NET 运行时。

17.4.5　File System 编辑器

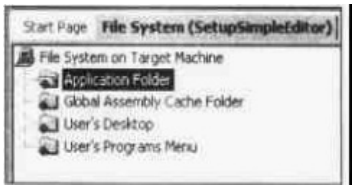


图　17-12

通过使用 File System 编辑器，我们可以向安装软件包中添加文件，并配置安装它们的位置。在 Visual Studio.NET 中的 View | Editor | File System 菜单上可以打开此编辑器。还会自动打开一些预定义的特殊文件夹，如图 17-12 所示。

- Application Folder 用于存储可执行文件和库。其位置被定义为[ProgramFilesFolder][Manufacturer][ProductName]。在 English 语言系统上，[ProgramFilesFolder]被解析为 C:\Program Files。[Manufacturer]和[ProductName]的目录由 Manufacturer 和 ProductName 工程属性定义。
- 在 Global Assembly Cache Folder 中，我们可以安装共享装配件。Global Assembly Cache 用于应该在多个应用程序之间共享的装配件中。在第 21 章介绍了更多的共享装配件的知识，如需要更进一层地学习这些知识，可以参考《C#高级编程》(由清华大学出版社出版)。
- 如果希望在桌面上放置图标，可以使用 User Desktop 选项。此文件夹的默认路径是 C:\Documents and Settings\[username]\Desktop 或 C:\Documents and Settings\All Users\Desktop，这取决于安装是用于单个用户还是所有的用户。
- 用户通常在 User Programs Menu 中启动程序。默认路径是 C:\Documents and Settings\[username]\Start Menu\[Programs]。在此菜单中您可以把快捷方式放置到应用程序。快

捷方式具有名称，它包括公司和应用程序的名称，这样用户就可以很容易地识别应用程序，比如 Microsoft Excel。

注意：

一些应用程序创建了子菜单，由此可以启动多个应用程序，比如 Microsoft Visual Studio.NET 7.0。根据 Windows Guidelines，许多程序却由于错误的原因而列举了大量不需要的程序：在此菜单中不应该放置未安装的程序，因为此功能可以在 Control Panel 的 Add/Remove Programs 中获取，卸载程序应该在控制面板中进行。此菜单中也不应该放置帮助文件，因为帮助文件可以直接从应用程序中获取。因此对于许多应用程序而言，在 User's Programs Menu 中直接放置应用程序的快捷方式就足矣。这些限制的目的是不要在 Start 菜单中放置太多的项，使得它过于拥挤。

此信息的较好的参考位于“Microsoft Windows 2000 桌面应用程序规范”中。您可以从 http://msdn.microsoft.com 下载此文件，并选择 Partners & Certification | Windows Logo Program：

- User Personal Data Folder 是用户的默认文件夹，其中存储了文档。C:\Documents and Settings\[username]\My Documents 是默认路径。此路径是 Visual Studio.NET 使用的默认目录，用于存储工程。
- 当选择文件时，添加 User Send To Menu 的快捷方式可以扩展 Send To 上下文菜单。使用此上下文菜单，用户通常可以将文件发送到目标程序，比如软驱、邮件接受者或 My Documents 文件夹。

1. 向特殊文件夹添加项

选择文件夹并选择菜单 Action | Add，我们可以从列表中作出选择，向特殊文件夹添加项，如图 17-13 所示。

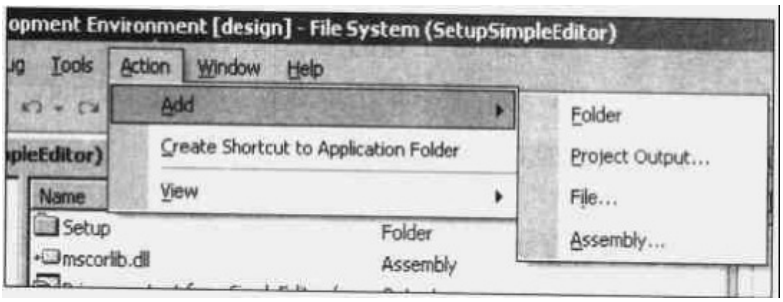


图　17-13

我们可以选择子文件夹、文件装配件或工程输出。选择工程输出，文件夹会自动添加生成的输出文件，是.DLL 或.EXE，这取决于所添加的工程是组件库还是应用程序。选择一个菜单 Project Output 或 Assembly 可以自动添加对文件夹的所有依赖关系(所有引用的装配件)。

2. 文件属性

通过在文件夹中选择文件的属性，我们可以设置表 17-2 中所示的属性。取决于文件类型，一些属性不会得到应用，还有一些属性未列举出来。

表 17-2

属　性	说　明
Condition	有了此属性，当选择应该安装的文件时，就定义条件。如果您仅仅希望在具体的操作系统版本上添加文件，或者添加一些用户必须在对话框中选择的内容时，该属性就非常有用
Exclude	如果不安装此文件，则设置为 True。这样文件就可以保留在工程中而无需安装。如果可以肯定它不是相关文件或它存在于应用程序已经被破坏的系统中，则就可以排除文件
PackageAs	有了 PackageAs，我们就可以重写文件添加到安装程序软件包的方式的默认设置；例如，如果工程配置是 In Setup File，我们就可以使用此选项将具体的文件的软件包配置改变为 Loose，这样此文件就不会被添加到 MSI 数据库文件中。如果我们希望添加的文件是在开始安装之前用户应该读取的 Readme 文件，这就非常有用。显然，即使压缩了其他文件，我们也不会压缩此文件
Permanent	将此属性设置为 True 意味着在卸载之后文件仍会保留在目标计算机上。这可以用于配置文件。下面给出一个范例，即安装 Microsoft Outlook 的新版本时：如果配置了 Microsoft Outlook，则在卸载并再次安装它时，就不需要再次配置它，因为上一次的安装配置还没有被删除
ReadOnly	此属性在安装时设置只读文件特性
Vital	此属性意味着此文件对于此产品的安装至关重要。如果此文件的安装失败，就无法完成完整的安装进程，只能返回

试试看：向安装程序软件包添加文件

(1) 向安装程序工程添加 Simple Editor 工程的主要输出，为此使用 Project | Add | Project Output 菜单。在 Add Project Output Group 对话框中，选择 Primary Output，如图 17-14 所示。

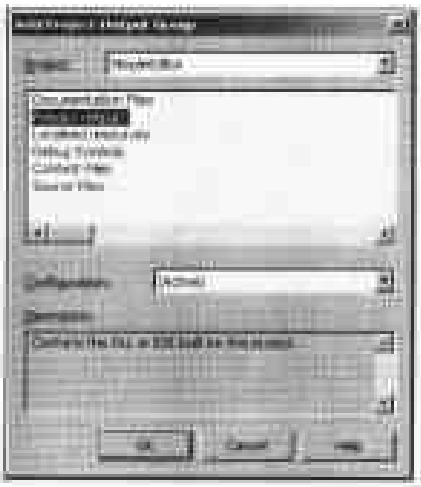


图 17-14

单击 OK 按钮，可以向在自动打开的 File System Editor 中的 Application Folder 添加

SimpleEditor 工程的主要输出结果。在此，主要的输出就是 SimpleEditor.exe。在 Solution Explorer (View | Solution Explorer)中可以看到，相关的文件自动受到检测；并向工程添加了合并模块 dotnetfxredist_x86_enu.msm。这个合并模块包括.NET 运行时的所有的文件。如果.NET 运行时未安装在目标系统上，则它就随应用程序安装。

在安装程序中，我们要求必须首先安装运行时。选择文件 dotnetfxredist_x86_edu.msm，并将 Exclude 属性更改为 True。运行时大约需要 15MB 的空间，因此我们的安装文件要比 15MB 稍微小一点。将 Exclude 属性设置为 True 安装就会成功，即使在目标系统上未安装.NET 运行时，但是却不会启动 SimpleEditor.exe。最好的方法是，如果在安装期间得到了错误消息，就不会安装运行时。我们可以通过定义.NET Framework Launch Condition 来完成。稍后我们介绍这些内容。

(2) 我们希望添加的附加文件有标志、许可和 readme 文件。在 File System Editor 中，要在 Application Folder 中创建名称为 Setup 的子目录。这可以通过选择 Application Folder 并选择菜单 Action | Add | Folder 来实现。

提示：

在 Visual Studio.NET 中，仅当选择了在 Setup Editor 中的项才可以得到 Action 菜单。如果选择了在 Solution Explorer 或 Class View 中的项，就不会得到 Action 菜单。

(3) 将文件 wroxlogo.bmp，wroxsetuplogo.bmp，readme.rtf 和 license.rtf 拖放到 Setup 文件夹中。在本书从 http://www.wrox.com 下载的范例中可以得到这些文件，但是您可以自己很容易地创建这些文件。可以使用许可和 Readme 信息填充文本文件。无需改变这些文件的属性。这些文件会用在安装程序的对话框中。

(4) 将文件 readme.txt 拖放到应用程序文件夹中。我们希望在开始安装之前用户可以读取这些文件。将属性 PackageAs 设置为 vsdpaLoose，这样此文件就不会被压缩到 Installer 软件包中。还要将属性 ReadOnly 设置为 True，这样就不会改变文件。

(5) 将文件 demo.wroxtext 拖放到 User's Desktop 文件夹中。只有在询问过用户是否希望安装之后，才能安装此文件。因此，将此文件的 Condition 属性设置为 CHECKBOXDEMO。其值必须大写。如果此条件设置为 TRUE，才可以安装此文件。稍后，我们要定义一个对话框，并在其中设置此属性。

注意：

我们更希望通过快捷方式而不是文件来演示 Condition 属性，因为在 User's Desktop 中最好使用快捷方式。但是在写入时，Visual Studio.NET 的 Installer 就有一个程序错误，即不能为快捷方式设置 Condition 属性，而只能是来自 MSDN 文档的信息。

我们希望从 Start | Programs 菜单中可以得到程序。我们需要 SimpleEditor 程序的一个快捷方式。

(6) 在 Application Folder 中选择 Primary output from SimpleEditor 项，并打开菜单 Action | Create Shortcut to Primary output from SimpleEditor。将所生成的快捷方式的 Name 属性设置为 Wrox Simple Editor，并将此快捷方式拖放到 User's Programs Menu。

17.4.6 File Types 编辑器



图 17-15

如果应用程序使用了定制的文件类型，而且您希望注册文件扩展名(file extentions)，当用户双击文件或使用上下文菜单时这些文件扩展名就启动应用程序，您可以使用 File Types Editor。此编辑器可以使用 View | Editor | File Types 启动，如图 17-15 所示。

有了 File Type Editor，您就可以配置文件扩展名，在应用程序中可以处理这些扩展名。文件扩展名具有表 17-3 所示的属性。

表 17-3

属　性	说　明
Name	在此您应该添加有用的名称，描述文件类型。此名称显示在 File Type Editor 中，并写入到注册表中。名称应该是惟一的。.doc 文件类型的范例是 Word.Document.8。无须像在 Word 范例中那样使用 ProgID，也可以为.dochtml 文件扩展名使用简单的文本，比如 wordhtmlfile
Command	通过使用 Command 属性，您可以规定可执行文件，当用户使用此类型打开文件时，就会启动这个可执行文件
Description	在此您可以添加说明
Extensions	应用程序可以进行注册的文件扩展名 会在注册表的部分中注册文件扩展名
Icon	为文件扩展名规定要显示的图标

创建动作

在 File Type Editor 中创建了文件类型之后，您就可以添加动作(action)。自动添加的默认动作是 Open。您可以添加附加动作，比如 New 和 Print，或者是程序使用文件时所需要的任何动作。对于动作，必须定义 Arguments 和 Verb 属性。Argument 属性规定传递给程序的参数，已经为文件扩展名注册了此程序；例如"%1"的意思是，文件名称被传递给应用程序。Verb 属性规定要发生的动作。如果应用程序支持的话，对于打印动作可以添加/print。

下面给 SimpleEditor 安装程序添加动作。我们希望注册文件扩展名，因此在 Windows Explorer 中使用 SimpleEditor 应用程序，以打开带有扩展名.wroxtext 的文件。在注册后，我们可以双击这些文件打开它们，这就会自动启动 Simple Editor 应用程序。

试试看：设置文件扩展名

通过 View | Editor | File Types 启动文件。使用带有下面的属性组的菜单 Action | Add File Type 添加新文件类型。我们不希望改变.txt 文件扩展名的 Notepad 的所属权，因此就使用.wroxtext 文件扩展名，如表 17-4 所示。

表　17-4

属　性	值
(Name)	Wrox.SimpleEditor.Text
Command	SimpleEditor 的主要输出
Description	Wrox 文本文档
Extensions	wroxtext

您也可以设置 Icon 属性，为打开的文件定义图标。

使 Open 动作的属性为默认值，因此要将文件名称作为应用程序参数传递。

17.4.7　Launch Condition 编辑器

通过 Launch Condition 编辑器，我们可以在安装之前对目标系统作出一些规定。通过选择菜单 View | Editor | Launch Conditions 可以启动 Launch Condition 编辑器，如图 17-16 所示。

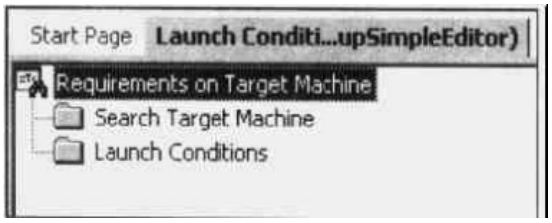


图　17-16

编辑器有两部分可以规定要求：Search Target Machine 和 Launch Conditions。在第一部分中，我们规定对具体的文件，注册表项等的搜索内容和方式，如果搜索失败，则在第二部分中定义错误消息。

下面研究一些通过使用 Action 菜单定义的启动条件，如图 17-17 所示。

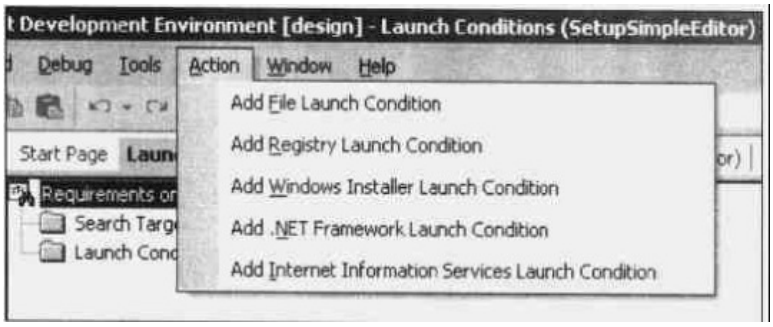


图　17-17

- 使用 File Launch Condition，我们可以在目标系统上搜索已经安装的文件。
- Registry Launch Condition 允许在安装之前查找注册表项。
- Windows Installer Launch Condition 可以搜索 Windows Installer 文件。
- NET Framework Launch Condition 检查目标系统上是否已经安装了.NET Framework。
- Internet Information Services Launch Condition 检查已经安装的 Internet Information

Server。当然，您也可以检查此服务器的版本。

在我们的范例中，我们使用.NET Framework Launch Condtion 启动条件。

.NET Framework 启动条件

对 Launch Condition 编辑器添加.NET Framework Launch Condition 启动条件，仅可以添加一个条件，在此条件中，给预定义的值设置属性：Condition 被设置为 MsiNetAssemblySupport，Message 被设置为[VSDNETMSG]。如果在系统上安装了 Common Language Runtime，则在安装时 Windows Installer 1.5 就将属性 MsiNetAssemblySupport 设置为 true。如果未安装.NET 运行时，则消息[VSDNETMSG]就显示安装时默认的错误消息。

试试看：添加.NET Framework 启动条件

在 File System Editor 中的文件中，我们已经设置排除了.NET 运行时，因此，必须在目标系统上检查.NET 运行时：

(1) 如果还没有使用 View | Editor | Launch Conditions 进行这样的处理，就可以启动 Launch Condition Editor。

(2) 使用 Action | Add .NET Framework Launch Condition 添加.NET Launch Condition。可以将 Name 属性设置为您喜欢的名称，比如是 DOTNETCondition。

在安装应用程序时，Installer 检查.NET 运行时。如果未安装它，用户就会看到图 17-18 所示的错误消息，然后停止安装。



图 17-18

17.4.8　User Interface 编辑器

使用 User Interface Editor，我们可以定义用户可以看到的对话框，以配置安装。通过它可以通知用户许可协议，询问安装路径和配置应用程序的其他信息。

试试看：启动 User Interface 编辑器

通过选择 View | Editor | User Interface 启动 User Interface Editor。可以使用 User Interface 编辑器为预定义的对话框设置属性。可以看到带有自动生成的对话框的视图和两个安装模式，如图 17-19 所示。



图　17-19

1. 安装和有管理的安装

从图 17-19 中可以看到，我们有两种安装模式：Install 和 Administrative Install。Install 模式是在目标系统上安装应用程序的典型安装模式。而使用 Administrative 安装，我们可以在网络共享上安装应用程序的图像。然后用户就可以在网络上安装应用程序。

2. 默认的对话框

在两种安装模式中，有 3 种序列可以显示对话框：Start，Progress 和 End。下面介绍默认的对话框：

- Welcome 对话框给用户显示欢迎消息。可以使用自己的消息代替默认的欢迎文本。用户仅仅删除安装或单击下一个按钮即可。
- 通过第二个对话框 Installation Folder，用户可以选择文件夹，在其中可以安装应用程序。如果添加了定制的对话框，就必须在此对话框之前添加对话框。
- Confirm Installation 对话框是启动安装之前的最后一个对话框。
- Progress 对话框显示进度控件，使得用户可以看到安装进度。
- 安装结束后会显示 Finished 对话框。

我们所介绍的默认对话框会自动显示，即使从来没有打开 User Interface 编辑器。但是我们必须配置这些对话框，以便显示对编辑器应用程序有用的消息。

试试看：配置默认对话框

(1) 选择 Welcome 对话框。在属性窗口中，可以看到可以更改此对话框的 3 种属性：BannerBitmap，CopyrightWarning 和 WelcomeText。在复选框中选中(Browse...)可以选择 BannerBitmap 属性，选择文件 wroxsetuplogo.bmp，我们将此文件放置在文件夹 Application Folder\Setup 中。此位图会显示在对话框的顶部。

属性 CopyrightWarning 的默认文本为：

警告：

此计算机程序受到版权法和国际公约保护。未经授权，复制或销售此程序，或其中的一部

分，都会受到法律许可的最大限度的民事或刑事处罚。

此文本也会显示在 Welcome 对话框中。如果不符合您的需要，可以改变此文本。

在此对话中的另一个文本由属性 WelcomeText 定义：

安装程序会引导您完成在计算机上安装[ProductName]的全部步骤。

您也可以改变此文本。字符串[ProductName]自动被属性 ProductName 代替，我们使用工程的属性定义了属性 ProductName。

(2) 选择对话框 Installation Folder。此对话框仅有一个属性：BannerBitmap。可以按改变 Welcome 对话框的同样的方式改变此属性。对于显示位图的每一个对话框而言，每一个文件夹都具有此属性。也可以在所有其他的对话框中改变此属性。

3. 附加对话框

仅仅具有 Visual Studio.NET 安装程序，我们还不能设计定制对话框，并将其添加到安装序列。完成这些任务就需要更复杂的工具，比如 Windows 的 InstallShield 或 Wise。但是有了 Visual Studio.NET 安装程序，我们可以在 Add Dialog 窗口中添加和定制许多预定义的对话框。

通过在 User Interface 编辑器中选择 Start 序列和选择菜单 Action | Add Dialog，可以显示 Add Dialog，如图 17-20 所示。所有这些对话框都是可以配置的。

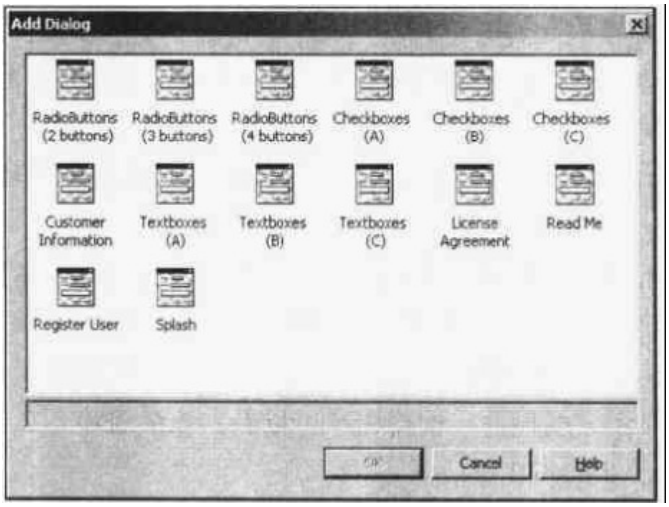


图　17-20

我们具有的对话框，其中可以显示 2，3 或 4 个单选按钮，复选框对话框可以显示 4 个复选框，文本框可以显示 4 个文本框。我们可以通过设置属性而配置这些对话框的 Windows 控件。我们使用 Checkboxes 对话框来演示如何使用这些对话框。

Customer Information 对话框询问用户的名称、公司和序列号码。如果没有产品的序列号码，可以通过将 ShowSerialNumber 设置为 False 而隐藏序列号码。使用 License Agreement 对话框，用户可以在开始安装之前接受一个许可。许可文件通过 LicenseFile 属性定义。使用 Register User 对话框，用户可以单击 Register Now 按钮，其中会启动使用 Executable 属性定义的程序。定制程序可以向 FTP 服务器发送数据，或通过电子邮件传输数据。Splash 对话框在开始安装之前显示一个醒目的窗口，其中具有 SpashBitmap 属性所规定的位图。

试试看：添加其他对话框

(1) 可以使用 Action | Add Dialog 在 Start 序列中添加 Read Me，License Agreement 和 Checkboxes 对话框。通过以下面的方式拖放和定义开始序列中的顺序：

Welcome – Read Me – License Agreement – Checkboxes - Installation Folder – Confirm Installation

提示：

所有定制的对话框必须放置在 Installation Folder 对话框之前；否则构建安装软件包时就会出现警告。

(2) 我们必须按以前的方式为所有的对话框配置属性 BannerBitmap。对于 Read Me 对话框，将属性 ReadmeFile 设置为 readme.rtf，即我们在前面向 Application Folder\Setup 添加的文件。

(3) 对于 License Agreement 对话框，将 LicenseFile 属性设置为 license.rtf。

(4) Checkbox 对话框用于询问用户是否安装我们放置在 User's Desktop 中的文件 demo.wroxtext。根据表 17-5 来改变此对话的属性。

表　17-5

属　性	值
BannerText	Optional Files
BodyText	Installation of optional files
Checkbox1Label	Do you want a demo file put on to the desktop?
Checkbox1Property	CHECKBOXDEMO
Checkbox2Visible	False
Checkbox3Visible	False
Checkbox4Visible	False

属性 Checkbox1Property 被设置为与文件 demo.wroxtext 的 Condition 属性具有相同的值，当我们使用 File System 编辑器向软件包中添加文件时设置了文件 demo.wroxtext。如果用户选择此复选框，则值 CHECKBOXDEMO 就是 true，就安装文件；如果未选择复选框，值是 false，就不安装文件。

其他复选框的 CheckboxXVisible 属性被设置为 False，因为我们仅需要一个复选框。

17.5　构建工程

现在开始构建安装程序工程。

试试看：构建工程

(1) 选择菜单 Build | Build Solution，创建 Microsoft Installer Package。如果构建成功，则可以在 Debug 或 Release 目录中找到图 17-21 所示的文件(取决于构建设置)。

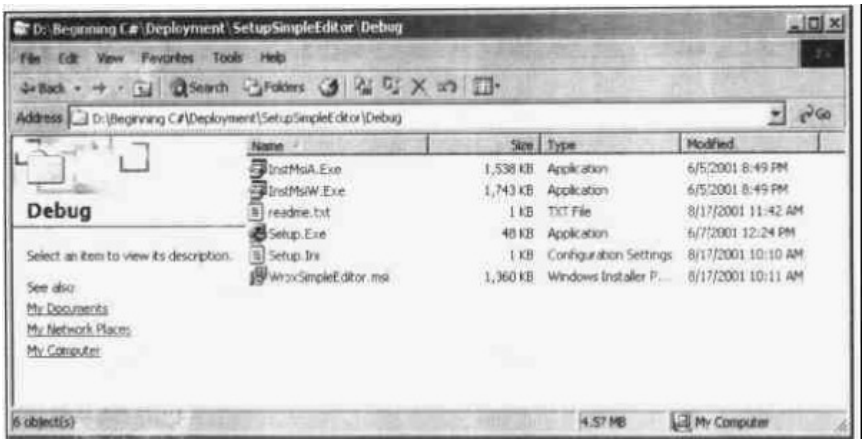


图　17-21

注意：

Setup.exe 开始安装 MSI 数据库文件 WroxSimpleEditor.msi。向安装程序工程添加的所有文件(只有一个例外)都被合并和压缩到 MSI 文件中，因为我们已经将工程属性设置为 Package Files in Setup File。其中有一个例外是文件 readme.txt。有了此文件，我们已经改变了 PackageAs 属性，以便在安装应用程序之前可以马上读取它。Windows Installer 本身出现在文件 InstMsiA.exe 和 InstMsiW.exe 中；并给出 Windows 98/ME 的 ASCII 版本和 Windows NT4/2000/XP 的 Unicode 版本。

17.6　安装

现在可以安装 Simple Editor 应用程序。可以在 setup.exe 文件上双击，或选择文件 WroxSimpleEditor.msi 并按鼠标右键，打开上下文菜单，选择 Install 选项，或在 Visual Studio.NET 中完成，Visual Studio.NET 具有在 Solution Explorer 中打开的安装工程，如图 17-22 所示。

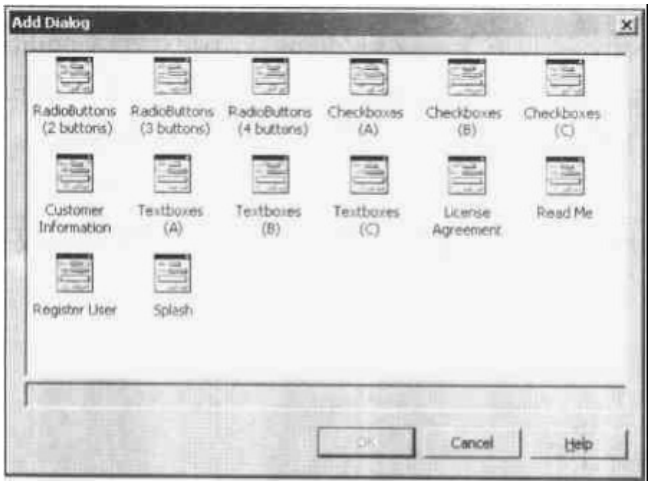


图　17-22

从下面的屏幕图中可以看到，在可配置的文件中出现了所有具有 Wrox 标志的对话框、插入的 ReadMe 和 License Agreement 对话框。

下面按顺序介绍安装序列。

17.6.1　Welcome

可以看到的第一个对话框是 Welcome 对话框。在此对话框中，我们可以看到 Wrox 标志，可以通过设置 BannerBitmap 属性的值而插入它。所看到的文本由属性 WelcomeText 和 CopyrightWarning 定义。此对话框的标题可以引出属性 ProductName，我们已经使用工程属性设置了 ProductName 属性，如图 17-23 所示。

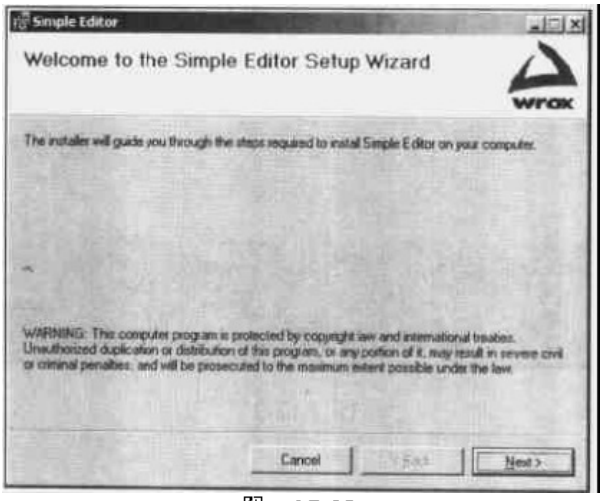


图　17-23

17.6.2　Read Me

单击 Next 按钮之后，我们可以看到配置的 Read Me 对话框。它显示富文本文件 readme.rtf，可以通过设置属性 ReadMeFile 而配置它，如图 17-24 所示。

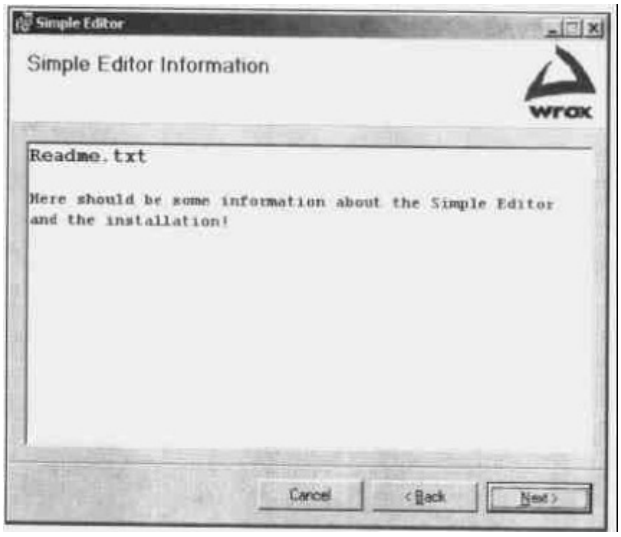


图　17-24

17.6.3　License Agreement

所出现的第 3 个对话框是许可对话框。在此我们也是仅仅配置 BannerBitmap 和 LicenseFile

属性。符合此许可的单选按钮也会自动被添加。在图 17-25 中可以看到，Next 按钮还不可以使用，直到选中 I Agree 按钮为止。此对话框的这个功能是自动完成的。



图 17-25

17.6.4 Optional Files

同意许可信息并单击 Next 按钮可以显示 Checkbox 对话框。有了此对话框，我们不仅可以设置位图，而且还可以为属性 BannerText，BodyText 和 Checkbox1Label 设置文本。所有其他的复选框都不可见，因为特定的 CheckboxNVisible 属性被设置为 False。

选择复选框就会在桌面上安装文件 demo.wroxtext，如图 17-26 所示。



图 17-26

17.6.5　选择安装文件夹

有了此对话框，用户可以选择安装应用程序的路径。此对话框允许我们设置属性 BannerBitmap。默认路径是[Program Files]\[manufacturer]\[Application Name]。

用户也可以选择是为每一个人安装应用程序，还是仅为当前登录的用户安装应用程序。根据此选项，程序文件的快捷方式要放置在针对用户的目录，或放置在 All Users 目录中，如图 17-27 所示。

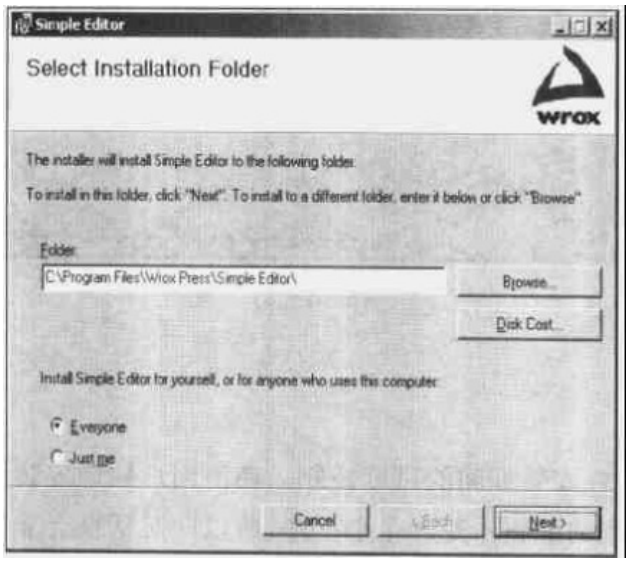


图　17-27

Disk Cost

单击 Disk Cost 按钮可以打开图 17-28 所示的一个对话框，其中显示所有磁盘的磁盘空间，并计算每一个磁盘所要求的空间。这帮助用户选择安装应用程序的磁盘。

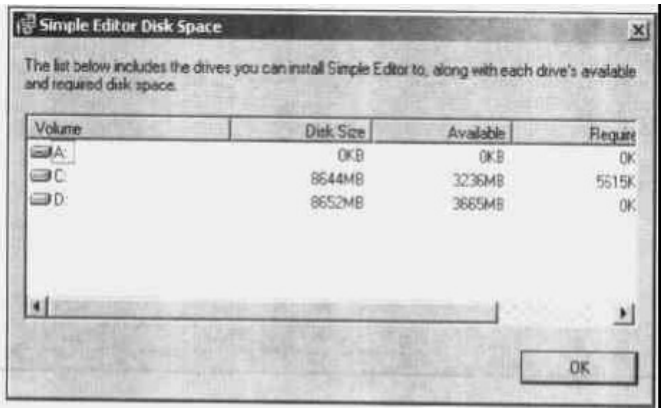


图　17-28

17.6.6　确认安装

下面的对话框是最终开始安装的最后一个对话框。在此没有什么问题，它仅仅是在实际安装之前可以删除安装的最后一个位置，如图 17-29 所示。

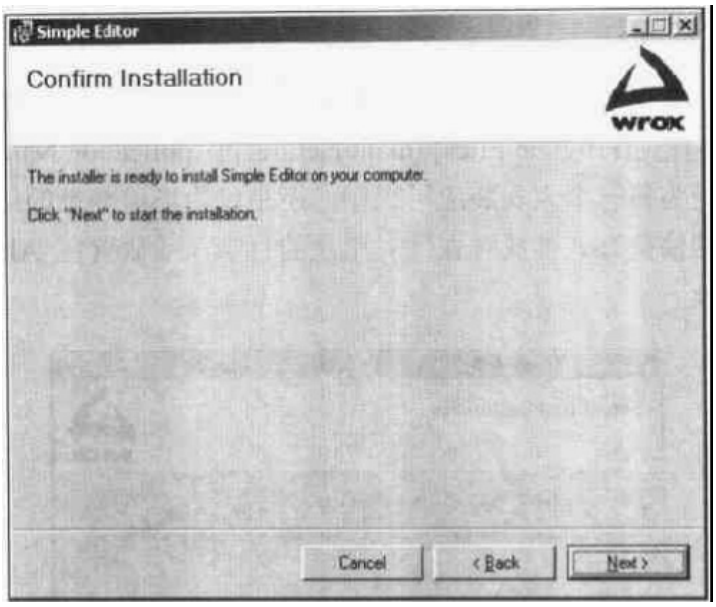


图　17-29

17.6.7　进度

Progress 对话框显示在安装期间的进度控制，使得用户知道安装还在进行，并大致估算安装所需要的时间。因为我们的编辑器是一个小程序，所以此对话框会非常迅速地结束，如图 17-30 所示。

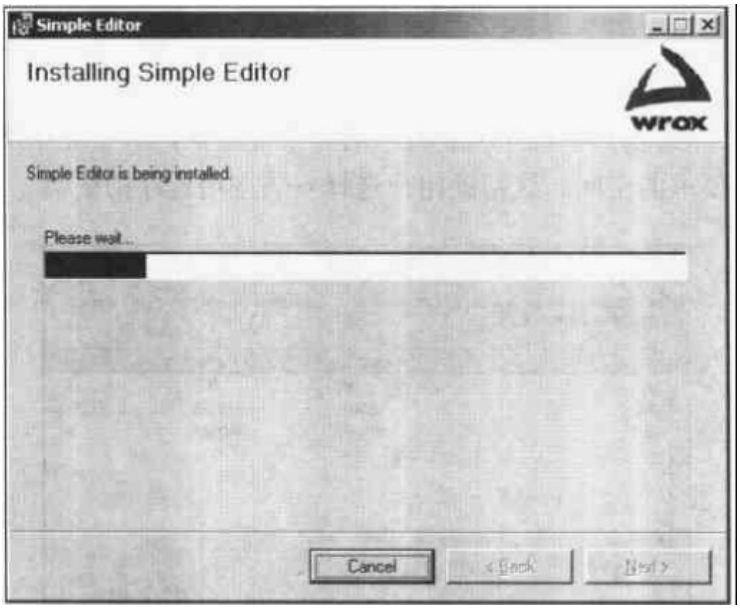


图　17-30

17.6.8　结束安装

安装成功后，我们可以看到最后一个对话框：Installation Complete 对话框，如图 17-31 所示。

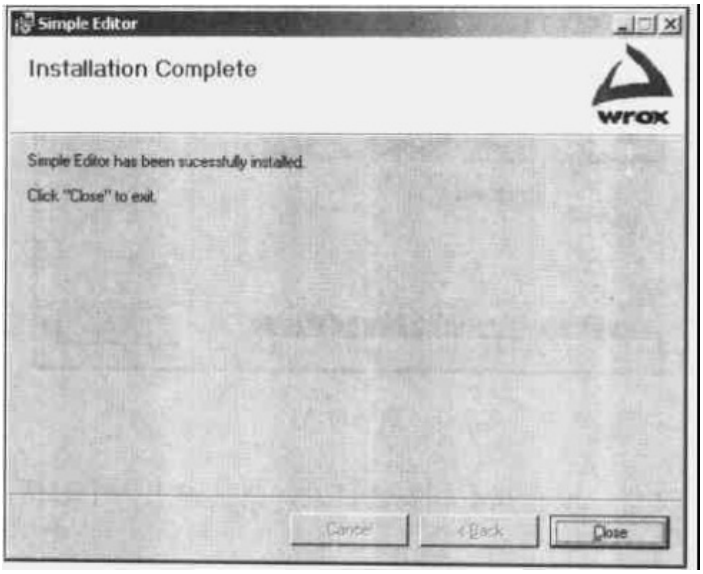


图　17-31

17.6.9　运行应用程序

从菜单项 Start | Programs | Wrox Simple Editor 中可以启动 Editor，如图 17-32 所示。

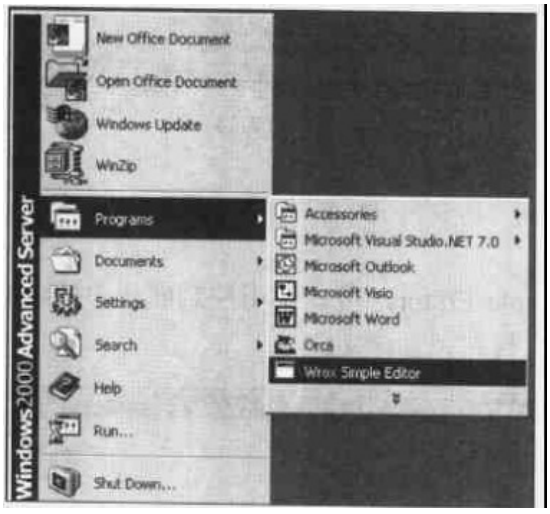


图　17-32

因为我们注册了文件扩展名，所以还有一种办法可以启动应用程序：双击带有文件扩展名.wroxtext 的文件。如果选择了带有 Optional Files 对话框的复选框，就可以在桌面上找到 demo.wroxtext，否则您就可以使用刚才安装的工具 Wrox Simple Editor 创建一个文件。

17.6.10　自我修复

现在已经安装了 Simple Editor。假如文件损坏了。为了完成此练习，我删除了文件 C:\Program Files\Wrox Press\Simple Editor\SimpleEditor.exe。现在我们可以选择 msi 文件的上下文菜单并选择 Repair 菜单。还可以使用一个更好的方法：在.wroxtext 文件上双击，或使用 Start | Programs | Wrox Simple Editor 启动编辑器。这就自动启动修复，因为丢失了一个必须的文件！

如果在完成安装的同一个位置找到了安装文件，当进行修复时，就可以看到如图 17-33 所示的对话框。

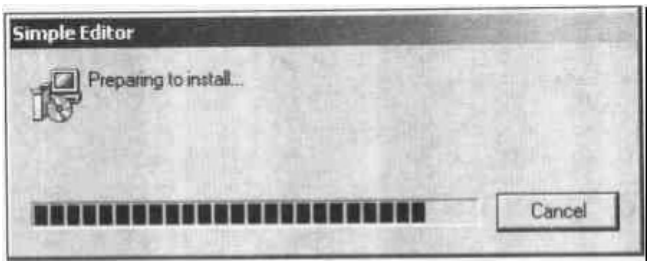


图　17-33

如果从 CD 上安装文件，或者找不到安装目录，就会弹出一个对话，询问安装文件，该对话框如图 17-34 所示。

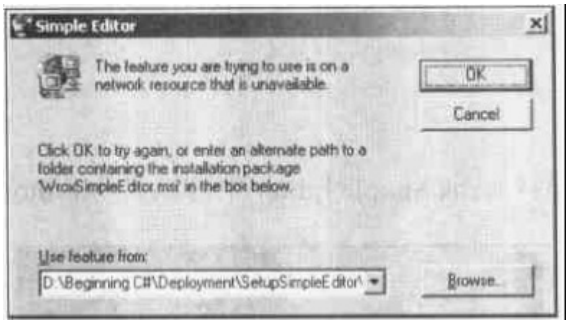


图　17-34

17.6.11　卸载

如果希望清除 Wrox Simple Editor，可以使用控制面板中的 Add / Remove Programs，并在 Simple Editor 中单击 Remove 按钮，如图 17-35 所示。



图　17-35

17.7　小结

在本章，我们介绍了 Windows Installer 的功能以及如何使用 Visual Studio.NET 创建 Installer 软件包。Windows Installer 更易于进行标准化的安装、卸载和修复。

Visual Studio.NET Installer 在功能上受到限制，不能提供 Windows Installer 的全部功能，但是对于许多应用程序而言，Visual Studio.NET Installer 的功能就足够了。我们有许多编辑器，其中可以配置 Windows Installer 文件。有了 File System Editor，我们可以规定所有的文件和快捷方式，Launch Conditions 编辑器可以定义一些必须的前提条件，File Types 编辑器用于为应用程序注册文件扩展名，User Interface 编辑器更易于修改用于安装的对话框。

当开发 Windows 应用程序时，最好的方法是构建 Windows Installer 软件包，有了 Visual Studio .NET，这就很容易完成。

第18章 获取数据

尽管本章的章名是“获取数据”，但是我们自第 3 章(在第 3 章引入了变量和字面量)以来一直在处理数据。但是，究竟什么是字符串或数字，而不是一段数据呢？到目前为止，我们一直使用的是硬编码到程序中的数据。如果改变这些数据，我们就必须改变程序本身。在本章我们要介绍的内容是，如何访问保留在程序之外的数据——在数据库中，电子表格中的数据等。

任何商业应用程序的核心内容是数据。想像一下公司保留的描述雇员细节的数据，比如薪水、业绩和顾客细节等。您所编写的大多数程序会以某种方式访问外部数据，不管它是简单的数据输入应用程序(使得人事部门可以在数据库中输入和编辑雇员的细节)，还是成熟的电子商务 Web 站点(可以读取产品目录和顾客细节，比如信用卡号码以及来自后端数据库的运送地址)。

.NET Framework 在 System.Data 名称空间中提供一组特殊的对象，System.Data 名称空间使得我们可以相对容易地访问所有的数据。从总体上来说，这些对象就是 ADO.NET(此名称源于较早的数据访问技术 ActiveX Data Objects)。在第 19 章我们会详细介绍 ADO.NET，但是在本章，我们要学习如何使得 Visual Studio.NET 可以为我们完成所需要的全部工作。我们要使用 VS 的强大功能编写简单的数据驱动应用程序。这使得我们可以学习使用数据库的更重要的概念，而无需担心这些代码。对于喜欢在代码页面之上编写页面，最大程度地简化工作的人来说，本章并不适合！但是对于希望使得工作完成得又快又好的人来说，本章正是为你们准备的！

18.1 在 VS 中访问数据

ADO.NET 题目太大，本身就可以成书，虽然我们在第 19 章才详细介绍它，但是在此我们要介绍其基本知识，使得您有所认识，并可以使用它。ADO.NET 构建并可以使用较早的数据访问技术，因此我们首先回顾.NET 出现之前的情况。

18.1.1 数据访问简史

当编写第一个数据库系统时，比如 Oracle 和 IBM 的 DB2 时，希望他们的程序可以访问数据库中的数据的开发人员需要使用一组专用于该数据库系统的函数。每一个系统都有它们自己的函数库，比如用于 Oracle 的 Oracle Call Interface(OCI)，用于 Sybase 的 SQL Server(后来被 Microsoft 收购)的 DBLib。这些函数库使得程序可以快速访问数据，因为他们的程序直接与数据库通信。这表示程序员必须熟悉他们所使用的数据库的每一种库，因此编写数据驱动的应用程序的任务非常复杂。同时还表示，如果公司改变他们所使用的数据库系统，则就必须重新编写他们的应用程序。

这个问题被 Open Database Connectivity(开放式数据库连接性，ODBC)解决了。ODBC 由 Microsoft 与其他公司在 20 世纪 90 年代早期联合开发。ODBC 提供了开发人员可以用于任何数

据库系统的公共函数集。这些函数被驱动程序翻译为具体的函数调用，用于特定的数据库系统。

这解决了专用的数据库的主要问题——开发人员仅需要知道如何使用一组函数(ODBC 函数)即可，如果公司改变他们的数据库系统，在应用程序中所需要改变的就是连接到数据库的代码。但是还有一个问题。虽然“无纸办公”还只是一个理想，公司确实有非常巨量的电子数据存储在不同的位置——电子邮件、Web 页面和 Project 2000 文件等。ODBC 可以用于在传统数据库中访问数据，但是却不能访问其他类型的数据，这些数据不是存储在整齐的行和列中，也不需要具有相干的结构。

OLE DB 提供了解决此问题的答案。OLE DB 与 ODBC 的工作方式相同，都是在数据库和需要访问数据的应用程序之间提供一个抽象层。客户机应用程序通过该数据源的 OLE DB 提供者与数据源通信，数据源可以是传统的数据库或任何其他存放数据的位置。任何数据源的数据都以表格格式提供给应用程序——就像它来自数据库一样。OLE DB 允许访问现有的 ODBC 驱动程序提供的数据，因此它可以用于访问所有 ODBC 支持的数据库。稍后就可以知道，ADO.NET 以非常类似的方式支持 OLE DB 和 ODBC。

最后一项“旧”的数据访问技术是 ActiveX Data Object(ADO)。ADO 仅仅是一个薄层，位于 OLE DB 的顶部，允许以高级语言(比如 Visual Basic)编写的程序访问 OLE DB 数据。

18.1.2 ADO.NET 引言

尽管 ADO.NET 从 ADO 导出其名称，但是二者实际上差别很大。按其体系结构来说，ADO.NET 实际上更类似于 OLE DB。ADO.NET 包含 System.Data 名称空间中的一组对象，System.Data 名称空间可以通过.NET 数据提供者(provider)与数据库通信。ADO.NET 对象允许我们连接到数据库，在数据库中检索、编辑、删除和插入数据，并在程序中处理数据。

1 .NET 数据提供者

ADO.NET 有两个重要的组成部分——DataSet 和.NET 数据提供者。DataSet 用于以表格格式在程序中放置一组数据；它不关心数据的来源。数据提供者包含许多针对数据源的组件，这些组件允许我们连接到单独的数据源上，并与之通信。每一个数据提供者都位于 System.Data 名称空间内自己的名称空间中。在编写本书时，可以使用 3 种数据提供者：

- 用于 SQL Server 的数据提供者。它位于 System.Data.SqlClient 名称空间，用于连接到 SQL Server 7.0 或更高版本以及 MSDE 数据库(MSDE 是 SQL Server 的简缩版本)。SQL Server 提供者作为.NET Framework 的一部分安装。如果需要连接到 SQL Server 6.5 或更早的数据库版本，就需要使用 OLE DB 提供者。
- 用于 OLE DB 的数据提供者。它用于通过 OLE DB 连接到数据源，位于 System.Data.OleDb 名称空间。如同 SQL Server 提供者一样，它也随.NET Framework 安装。OLE DB 提供者不用于通过 ODBC 驱动程序连接到数据库。
- 用于 ODBC 的数据提供者。ODBC 提供者可以用于连接到具有 ODBC 驱动程序的数据库。ODBC 数据提供者的名称空间是 System.Data.Odbc。它可以从 Microsoft 的 Web 站点上下载，即 http://download.microsoft.com/download/dasdk/Beta/1.0.2914.58/W98NT42KMeXP /EN-US/ odbc_net.exe。

尽管此时我们仅仅可以直接连接到 SQL Server 和 MSDE 数据库，但是可以使用的数据提供者的数量却希望得到增多，因为有更多的人转而使用.NET。尤其是，必须尽快提供用于 Oracle 的数据提供者。但是当前，如果希望连接到其他的数据源，还是需要使用 OLE DB 或 ODBC。

数据提供者包含 4 个主要组件，如表 18-1 所示。

表 18-1

组件名称	说明
Connection	用于连接到数据库或其他数据源
Command	用于在数据库中检索、编辑、删除或插入数据
DataReader	从数据源提供数据流。这些数据只读(不可修改)，我们只能前向经过这些数据
DataAdapter	用于使用来自数据源的数据填充 DataSet ，并使用 DataSet 中的更改来更新数据源

但是，当使用这些对象时，实际上不是通过其名称引用它们。每一个提供者都有自己的类的实现。针对提供者的类实现相同的界面，因此它们都具有相同的方法和属性(尽管不能阻止具体的类添加它们自己的方法和属性)。例如，如果我们希望创建到 SQL Server 数据库的连接，则我们可以使用 SqlConnection 对象，或者如果我们希望连接到 ODBC 数据源，就可以使用 OdbcConnection。这两种对象都实现 IDbConnection，因此它们都具有相同的方法和属性集。

2. DataSet

在创建了到数据库的连接并从中检索了数据之后，就可以使用 DataSet 对象在程序内处理该数据。DataSet 实际上是 DataTable 对象的集合。如同其名称所暗示的，DataTable 以表的格式存储数据；即在列和行中存储数据。DataSet 不会成为数据提供者的一部分(不管数据是来自 SQL Server，还是来自 OLE DB 数据源，DataSet 对象类型只有一种)，因此 DataSet 不维护到数据源的连接。这表示当在 DataSet 中处理数据时，我们仅仅是在本地机器上的内存中使用数据的副本。这缓解了数据库服务器和网络的压力，因为只是在首次获取数据和编辑完数据并将其回传到数据库时才连接到数据源。虽然这种非连接数据结构具有优点，但是还有一些问题。当处于完全断开的环境中时，用户不会知道其他用户对数据所作的改动。如果使用屏幕上所看到的数据决策，而其他用户正在编辑这些数据，则我们就只能得到落伍的信息。我们已经介绍了足够多的抽象的理论。现在我们启动 VS.NET，在 BegCSharp\Chapter18 文件夹中创建新的 C# Windows 应用程序，命名为 Ch18Ex01，并连接到数据库中！

18.2 在 VS 中查看数据

首先，我们需要一个要连接的数据库！在此，我们要连接到 SQL Server Northwind 数据库。如果没有 SQL Server 也不要担心。随 Office 和 Access 2000 安装的 Microsoft Data Engine (MSDE) 类似于 SQL Server 引擎的简缩版本，您同样可以访问它，就好像已经安装了完整的 SQL Server

版本一样。在默认情况下，不会安装MSDE，但是如果您的系统上未安装它，可以逐步安装它，并输入附录B的Northwind数据库。

18.2.1 连接到数据库

打开VS.NET之后，再打开Server Explorer，为此要将鼠标悬浮于窗口右侧的标签上(默认地，Server Explorer标签位于Toolbox标签之上)。此窗口显示在本地机器以及其他可以访问的机器上可以得到的服务。我们所感兴趣的是Server Explorer树视图中的顶部节点，因为它允许我们在工程中添加Data Connections，如图18-1所示。

右击Data Connections节点，选择Add Connection...。这就会弹出Data Link Properties窗口，如图18-2所示。

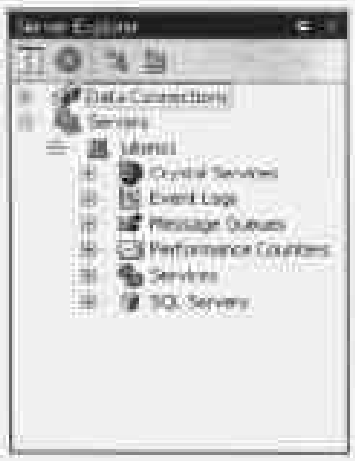


图 18-1

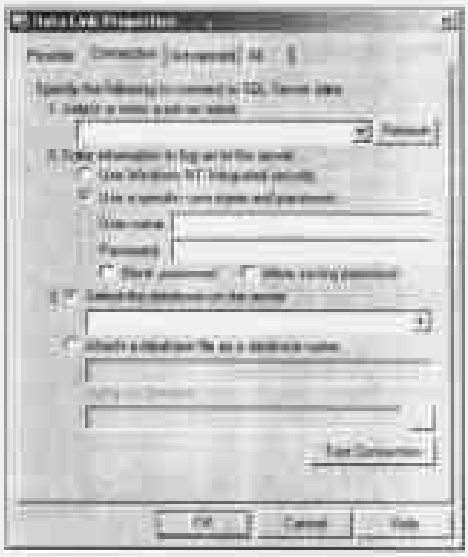


图 18-2

默认地，连接提供者假定我们要访问SQL Server数据库，因此所显示的所有的设置就是用于SQL Server的OLE DB Provider。在此，我们要访问在MSDE或SQL Server中的Northwind范例数据库，因此我们希望保留这些设置。但是，如果您希望使用另一个数据库系统，就需要回到Provider标签，为数据库选择合适的提供者，如图18-3所示。

单击Provider标签会弹出您的系统上安装的一列提供者。但是在此我们希望使用默认的SQL Server提供者，因此仅仅单击Next >>按钮即可。

提示：

在本章，我们使用SQL Server/MSDE和Northwind范例数据库。如果希望使用另一种数据库，仅仅为您的数据库系统选择合适的OLE DB提供者即可(例如，连接到Access数据库的Microsoft Jet提供者)。但是不要将它与.NET数据提供者相混淆。Data Link不允许我们选择.NET数据提供者，因此我们必须使用OLE DB的数据提供者，即使对于SQL Server也是如此。VS会识别出它连接到了SQL Server数据库，并为我们创建SqlConnection对象。

单击Next >>按钮，可以回到Connection标签。在此，我们需要填写VS需要的所有信息才

能连接到数据库。首先，在顶部文本框中，要么输入安装了 SQL Server 或 MSDE 的服务器的名称，要么从下拉列表中选择服务器名称。

接下来，您需要规定连接到数据库所需要使用的用户帐户。如果选择了 Use Windows NT Integrated Security，VS 就会试图使用您登录到 Windows 所使用的同一个用户帐户连接到数据库。如果您使用了不同的用户名和密码连接到 SQL Server，可以选择下一个选项(Use a specific username and password)，并将它们输入到合适的框中。随 MSDE 和 SQL Server 安装的默认用户帐户的用户名称是 sa，密码为空。

最后，在 Select the database on the server 下拉列表中选择 Northwind 数据库。如果使用的是 MSDE，则 Northwind 不会出现在列表中。关于如何输入数据库的详细知识，请参见附录 B。

当输入了这些信息之后，单击 Test Connection 按钮。现在您就有希望看到一个对话框，告诉您，连接成功，该对话框如图 18-4 所示。

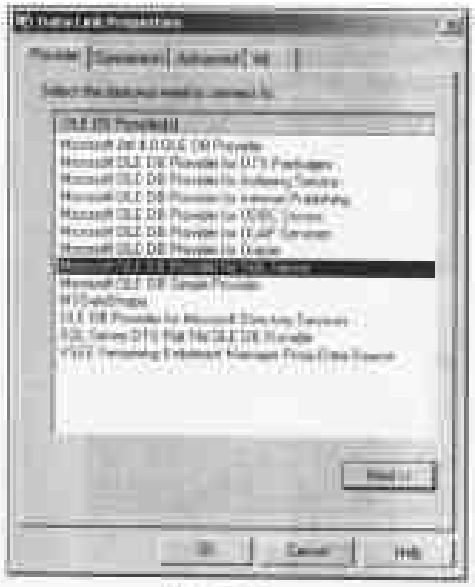


图 18-3

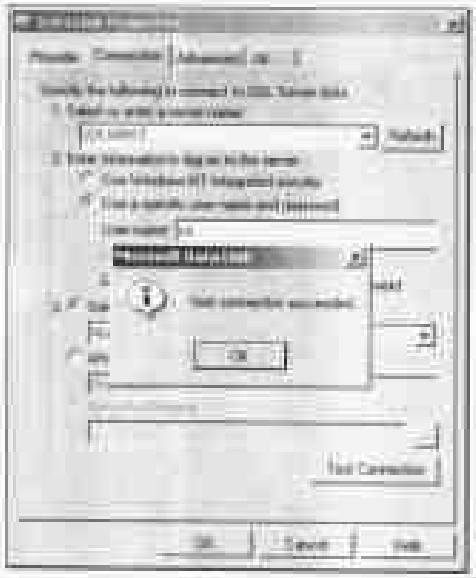


图 18-4

祝贺您，您已经成功地添加了第一个 ADO.NET 数据库连接！现在可以看到它嵌套在 Server Explorer 的 Data Connections 部分下，如图 18-5 所示。



图 18-5

现在我们可以访问数据库以及其中的数据，而且没有离开 VS！下面介绍一下这些数据，看一下这些数据的结构。这有助于我们在第 19 章编写访问数据的代码。

18.2.2　数据库表和关系

在 Server Explorer 中，打开刚才添加的连接之下的 Tables 节点。可以看到在数据库中的一列表，如图 18-6 所示。



图　18-6

关系数据库，比如 SQL Server 和 Oracle，在一系列关联的表中存储数据。这些表包含列和行；每一行代表数据库中的一条记录，列代表每一条记录的单独的域。为了直观地显示，请在 Server Explorer 中的 Customers 节点上单击，并从 Table 中选择 Retrieve Data。现在 VS 就会从 Northwind 数据库中的 Customers 表中加载和显示数据，如图 18-7 所示。



图　18-7

Northwind 范例数据库包含虚拟的食物批发商的数据，这些批发商可以提供各种饭店和食品店数据。Customers 表包含每一个顾客的细节。此表中的每一行都代表 Northwind 所提供的具体的公司，每一列包含关于公司的具体的数据段，比如公司名称、地址和在公司中 Northwind

的联系人的名称。

在表中的每一行由惟一的 5 字符 ID 代码区分，这些代码存储在 CustomerID 域中。这些独特的域就是主键码 (primary key)，在将 Customers 表与数据库中的其他表关联时起关键作用。为了学习其工作方式，可以右击 Orders 节点，并再次从 Table 中选择 Retrieve Data，如图 18-8 所示。

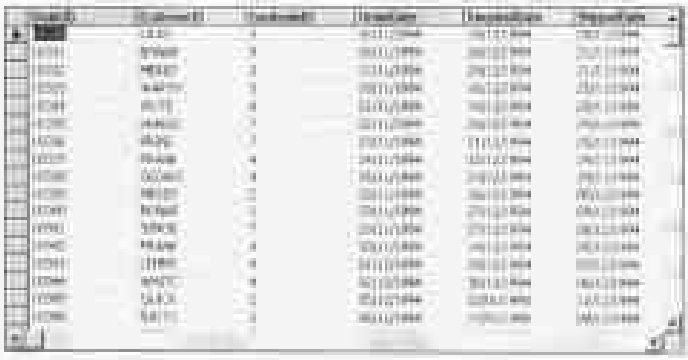


图 18-8

此表表示 Northwind 公司接收到的订单。每一行表示一个订单。注意，此表也具有一个 CustomerID 域，它带有与 Customers 表中的 CustomerID 域相同的 5 字符 ID 代码。在此列中的值用作在 Customers 表中的行的指针，Customers 表中有关于顾客的更详细的信息。这种类型的列就是外键码(foreign key)。

VS 提供了工具，可以在数据库中以图形化方式表示主键码和外键码之间的关系。为了直观演示这一点，在 Server Explorer 中右击 Northwind 连接的 Database Diagrams 节点，选择 New Diagram。VS 会询问，希望在图中添加哪一个表，如图 18-9 所示。



图 18-9

在列表框中选择所有的表，单击 Add 按钮。然后单击 Close 按钮，清除 Add Table 对话框。此时，VS 就会创建一个图表，显示表以及表之间的关系，如图 18-10 所示。

注意：

如果您使用的是 MSDE，并使用 Import and Export Data 向导导入 Northwind 数据库，则在图表中就不会显示其关系，因为向导不会导入主键码和外键码信息。

为了在一个屏幕上显示完整的图表，可以右击图表的空白区域，从弹出的菜单中选择 Zoom | To Fit。但是您可能不能分辨所有的列名称！其关系由表之间的线表示，线两端有相应的符号。表旁边的键码符号表示在此表中的主键码与线另一端的表中的外键码相关。相反，无穷大符号表示外键码，它指向相关表中的主键码。

但是，您并没有购买 VS，将其用作数据库的前端，因此下面我们介绍如何使用 C#程序中的连接！

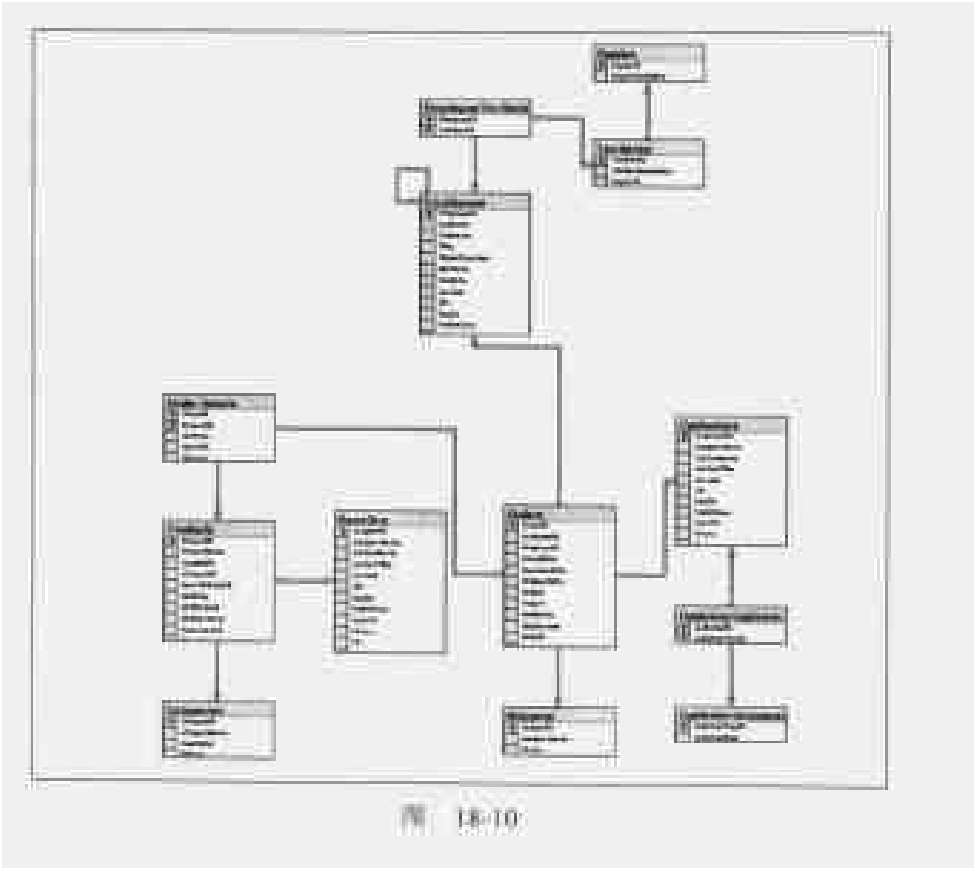


图　18-10

18.3　在应用程序中访问数据库

在 Ch18Ex01 工程中，返回到表单的设计视图，添加文本框和标签，并改变这些对象的属性，如表 18-2 所示。

表　18-2

控　件	属　性	值
表单	Name	frmMain
	Text	Data_Example1
文本框	Name	txtCustID
	Text	(blank)
标签	Name	lblCustID
	Text	Customer ID

最后的结果如图 18-11 所示。

现在我们在工程中添加数据库连接。打开 Toolbox 窗口，选择 Data 部分。在此菜单中，我们可以看到所有可以使用的与数据相关的控件，如图 18-12 所示。

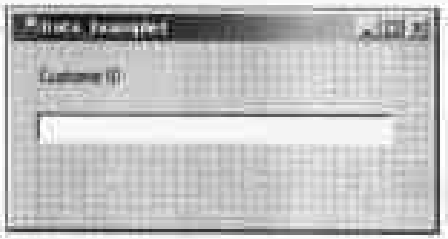


图 18-11



图 18-12

可以看到，对于 SQL Server 和 OLE DB 数据提供者，分别有 DataAdapter，Connection 和 Command 对象。SQL Server 对象带有 Sql 前缀，而 OLE DB 对象的前缀是 OleDb。在此，当我们访问 SQL Server 数据库时，我们可以使用 SQL Server 提供者。在 SqlDataAdapter 上双击，会打开 Data Adapter Configuration 向导。

数据适配器配置向导

在显示屏幕之后，向导会询问我们希望使用哪一种连接，如图 18-13 所示。

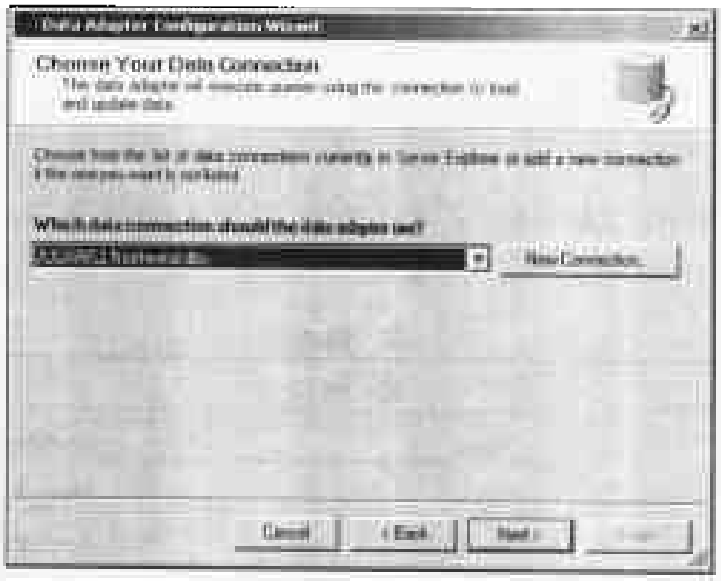


图 18-13

我们仅仅配置了一种连接(我们在前面建立的连接，以便访问 Northwind 数据库)，因此在列表中只能得到这一种连接。如果我们还没有构建连接，也可以选择 New Connection...，然后添

加它。

接下来，向导会要求我们选择希望选择要检索的数据的方式，如图 18-14 所示。

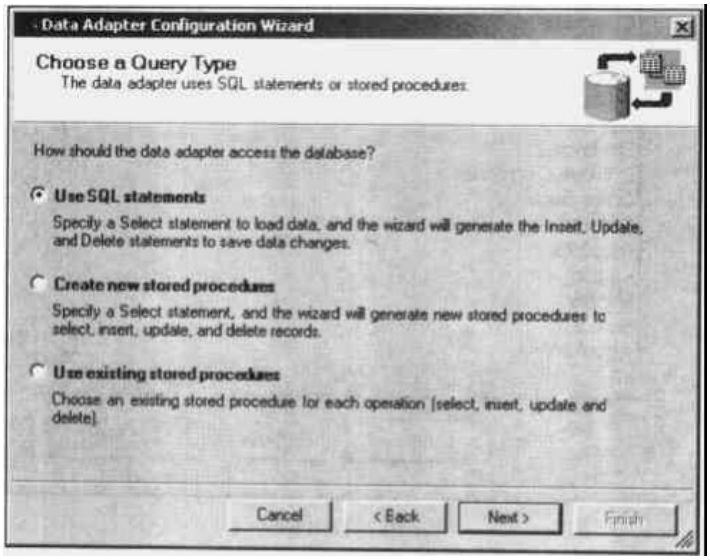


图　18-14

在此范例中选择 Use SQL Statements 选项。此选项允许我们使用 Structured Query Language(SQL，结构化查询语言)命令选择要检索的数据。SQL(不要与 SQL Server 相混淆)是一门特殊语言，用于与关系数据库通信，稍后我们要介绍这些知识。

单击 Next >按钮；我们就可以进入一个界面，其中可以输入希望用来检索数据的 SQL 语句。但是我们不必自己编写 SQL 语句，而是由 VS 来完成编写工作，如图 18-15 所示。

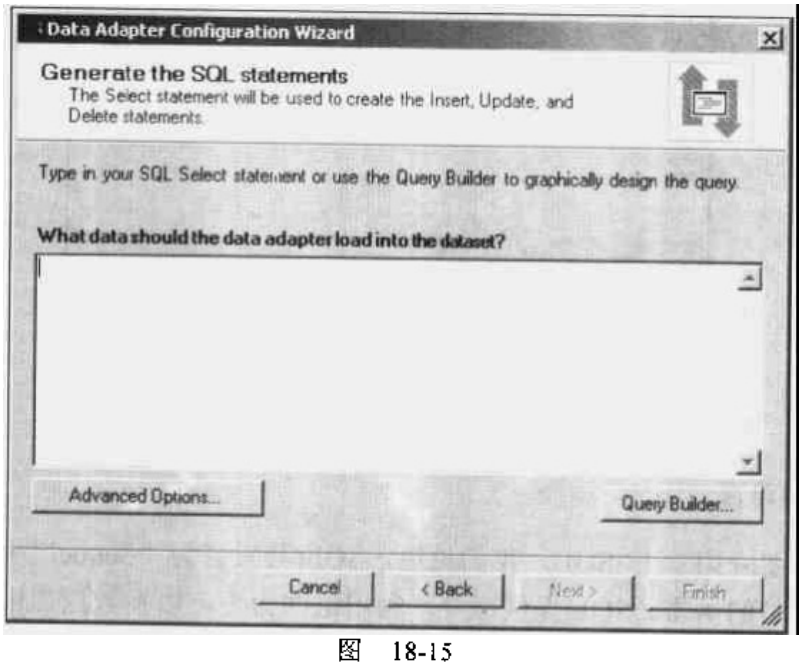


图　18-15

为了使得 VS 可以构建 SQL 语句，可以单击 Query Builder...按钮，如图 18-16 所示。

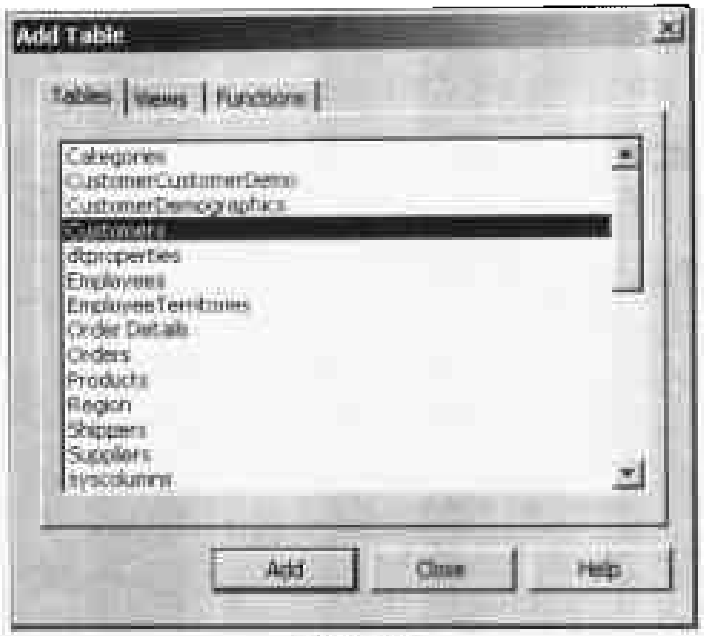


图 18-16

现在就在 Northwind 数据库中给出一个列表。在 Add Table 窗口中，选择 Customers。在 Add 按钮上单击，可以向 Query Builder 中添加 Customers 表，然后单击 Close 按钮。现在我们就可以在 Customers 表中选择单个的列，如图 18-17 所示。

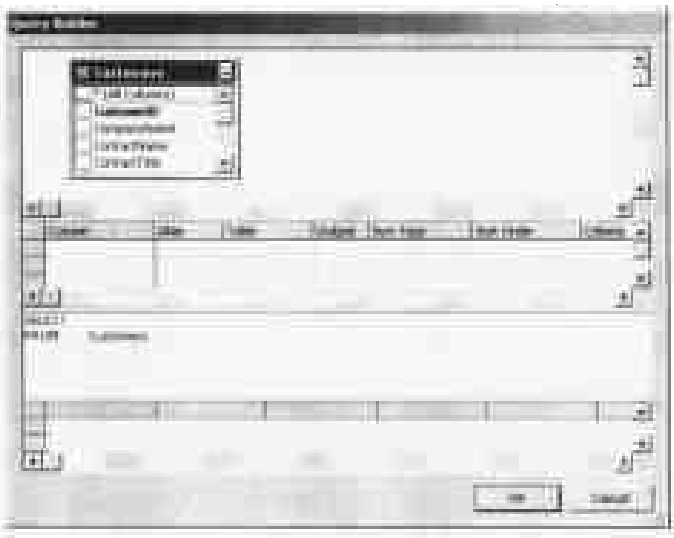


图 18-17

1. 结构化查询语言

此时，我们需要介绍一下 SQL。我们说过，SQL(其发音为“Sequel”或“ess-queue-ell”)是用于与数据库通信的语言。SQL 提供命令，利用这些命令可以在数据库中检索数据、插入或删除数据，并修改数据。在第 19 章我们要更详细地介绍 SQL 语言，但是现在，我们只介绍用来从数据库中检索数据的命令。

SELECT 语句的形式为 SELECT column1, column2, ... columnx FROM table；例如：

```
SELECT CompanyName, ContactName FROM Customers
```

此命令为在 Customers 表的每一行检索 CompanyName 和 ContactName 列的值。

我们也可以限制被检索的行数，为此在 SELECT 语句中添加 WHERE 子句。WHERE 子句允许我们为具体的列规定值，在数据库中仅仅会检索带有该列的值的行。例如：

```
SELECT CompanyName FROM Customers WHERE Region = 'WA'
```

此查询检索在 Washington 地区的公司名称。

我们也可以星号代替所有的列名称。这会选择表中的所有列。例如：

```
SELECT * FROM Customers
```

这就会选择完整的 Customers 表。

提示：

当选择“*”选项时要小心。虽然可以使用此选项，而且这也极富吸引力，但是几乎所有的数据库管理员或数据库程序员都建议不要使用它。如果使用了*，在执行时就会检索表中的所有列，而不仅仅是构建工程时使用的那些列。如果有人希望在表中添加域，应用程序就会选择它们，以及希望在应用程序中使用的那些域。所有要被传送的附加数据都会对应用程序的性能产生严重的影响。

2. 生成 SQL 语句

在此，我们选择了 3 列——CompanyName，ContactName 和 CustomerID。在 Query Builder 的 Customers 框中选择这些列，单击 OK 按钮。Query Builder 就会为我们创建 SQL 语句，如图 18-18 所示。

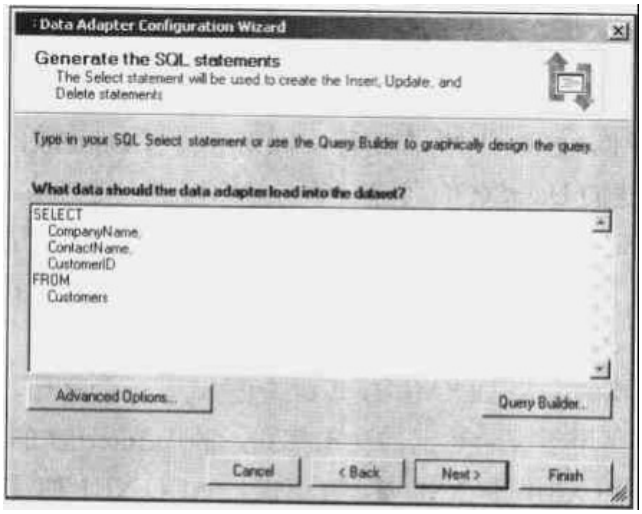


图　18-18

选择 Next >按钮。数据适配器就会验证所有的设置，并配置自己，以便进一步在应用程序中使用。结束时，单击 Finish 按钮。现在可以看到在表单之下添加了两个对象，即 SqlDataAdapte 和 SqlConnection，如图 18-19 所示。

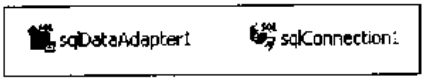


图　18-19

数据适配器包含我们希望从数据库中返回的数据的基本信息。可以将数据适配器看作是DataSet 与数据库之间的桥接器。数据适配器是.NET 数据提供者的一个组件，用于将来自数据库的数据填充到 DataSet 中，并将在 DataSet 中的变化返回到数据库中。SqlConnection 对象包含 ADO.NET 用来连接到数据库的所有信息。

现在有了数据适配器和到数据库的连接，我们就希望添加 DataSet，这样我们就可以在应用程序中使用数据。右击刚才创建的 sqlDataAdapter1，选择 Generate Dataset...。在随后出现的对话中，我们可以规定是将现有的 DataSet 关联到数据适配器，还是创建新的 DataSet，再与数据适配器相关联，如图 18-20 所示。

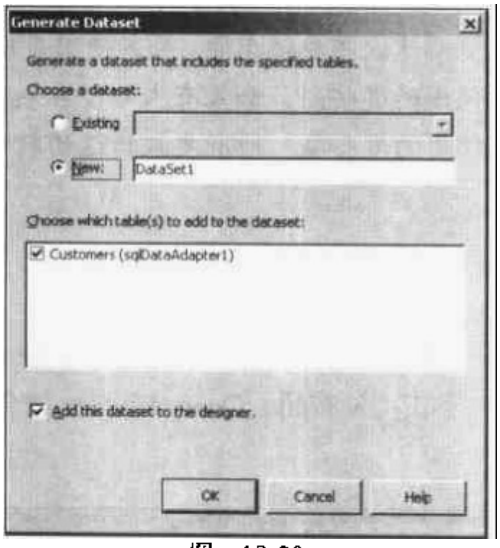


图　18-20

在此，我们创建了新 DataSet。因为在工程中原来并没有 DataSet，所以默认选择了 New。改变在 dsCustomers 中创建的 DataSet 的名称，单击 OK 按钮。

现在有一个新的 Data Set 对象紧邻着连接和数据适配器。您可能还没有注意到的是，在 Solution Explorer 中添加了新文件。如果您注意一下，就会发现在工程中添加了新文件，称为 dsCustomers.xsd，如图 18-21 所示。

此文件是 DataSet 的模式。XSD(XML 模式定义)模式是一个文档，通过它，我们可以验证 XML 文档的结构。XML 是用于表示数据的文本格式，随着 Internet 的发展，XML 变得日益重要。在本章后面，我们会介绍 XML 和 XSD 模式。模式是 ADO.NET 的重要的工具，因为 DataSets 在后台使用 XML 组织数据。此文件由 Visual Studio 自动生成，它规定了数据集、每一个表和表之间的关系的结构。

创建了 DataSet，我们就可以把表单上的文本框绑定到来自数据库的数据上。在 txtCustID 文本框上单击，在文本框的属性上翻屏，直到看到 DataBindings 细目为止。如果单击 DataBindings 之下的 Text 部分，就会注意到一个包含 dsCustomers1 和在该表中的所有列的组合框。通过在该组合框中选择任意的域，就可以将该域绑定到文本框。这仅仅意味着，当滚屏经过数据组时，

该列的值在文本框中可见。用户在文本框中所作的任何变化都可以反映到 DataSet 中。DataSet 不维护对数据源的永久性的连接，但是如果我们不特意请求的话，数据源本身不会更新。

在此，我们希望将文本框绑定到 CustomerID 列，因此就要从组合框中选择 CustomerID，如图 18-22 所示。



图 18-21



图 18-22

如果现在保存并运行工程，您就会看到一个空白文本框。为什么会是这样？难道我们在最后几页辛勤处理的数据还不能使用？其答案为否。要记住，到目前为止，我们仅仅定义了预计来自表的数据的结构。我们还没有定义实际的数据，或告诉应用程序获取一些数据。因此我们需要在范例中添加一些代码(事实上，这只是在本章所使用的少数代码行而已)。

我们需要通知数据适配器填充 DataSet。双击表单的背景。现在就可以进入代码编辑窗口，光标就会位于表单加载事件的处理程序中。在表单加载事件处理程序中，输入下面的代码：

```
private void frmMain_Load(object sender, System.EventArgs e)
{
    // Fill the dataset with the data in the Northwind database,
    // Customers table
    this.sqlDataAdapter1.Fill(this.dsCustomers1,1,0,"Customers");
}
```

这些代码执行数据适配器的 Fill()方法。Fill()方法使用来自数据源的数据填充 DataSet 中的 DataTable。此方法带有 4 个参数。第一个参数规定希望填充的 DataSet，在此就是 dsCustomers1 DataSet。第二个参数规定，应该从第一个记录开始。第三个参数指示我们希望返回多少记录。如果在此设为 0，则就返回所有的记录。最后一个参数规定生成数据的表的名称。

现在，保存并再次运行应用程序。可以看到第一条记录的 CustomerID 值，如图 18-23 所示。

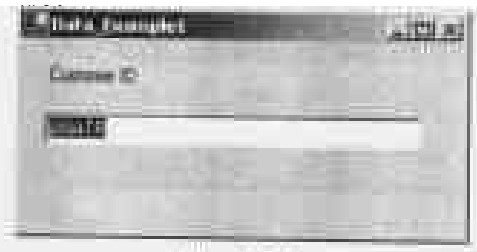


图 18-23

18.4　查看完成的结果

能够看到第一条记录非常不错，但是其用处不大。我敢肯定用户还希望能看到对应于 ID

的顾客和联系人的姓名，因此就在此表单中添加更多的文本框，如表 18-3 所示。

表　18-3

控　件	属　性	值
标签	Name	lblCustName
	Text	Customer Name
文本框	Name	txtCustName
	Text	(blank)
	DataBindings/Text	dsCustomers1 - Customers.CompanyName
标签	Name	lblContactName
	Text	Contact Name
文本框	Name	txtContactName
	Text	(blank)
	DataBindings/Text	dsCustomers1 - Customers.ContactName

注意：

用于这两个新文本框的 DataBindings 属性的设置方式几乎完全等同于设置原来的 txtCustID 文本框的方式——仅仅从组合框中选择合适的列即可，这些组合框紧邻在文本框的 DataBindings 属性之下的 Text 细目。

表单应该如图 18-24 所示。

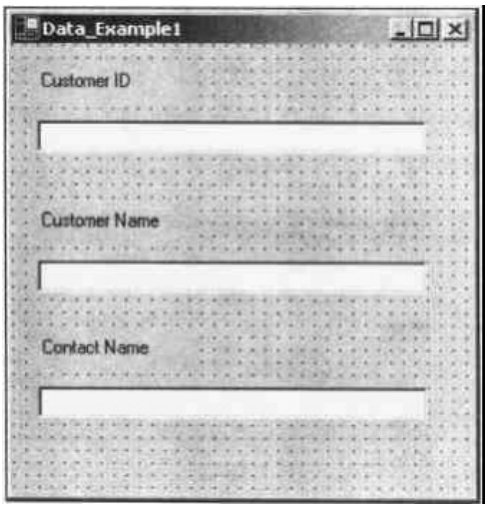


图　18-24

如果现在运行此表单，就可以看到数据库的附加信息。此时，您就可以随意添加域。在此，我们仅仅包括了来自数据适配器的表的 3 列。如果希望显示更多的内容，可以右击数据适配器，选择 Configure Data Adapter...选项。这就会弹出与我们前面使用的相同的 Configure Data Adapter

对话框，使得我们可以修改希望在数据源检索的列数。

18.5　在 DataSet 中导航

现在，我们已经显示了第一条记录，但是用户希望看到下一个记录。我敢打赌，一旦他们看到了第二条记录，就还希望看到第三条记录。他们永远不会满足的。

把玩笑撇开，在数据组中导航也是非常重要的。.NET 的每一方面都是以对象为基础的，因此 DataSets 也是对象，它以表、列和行作为集合。因此，导航这些对象非常类似于导航集合——对表示集合中的当前位置的计数器累加即可。

在表单中添加表 8-4 中的对象，其结果如图 18-25 所示。

表　18-4

控　件	属　性	值
按钮	Name	cmdBack
	Text	Back
按钮	Name	cmdNext
	Text	Next

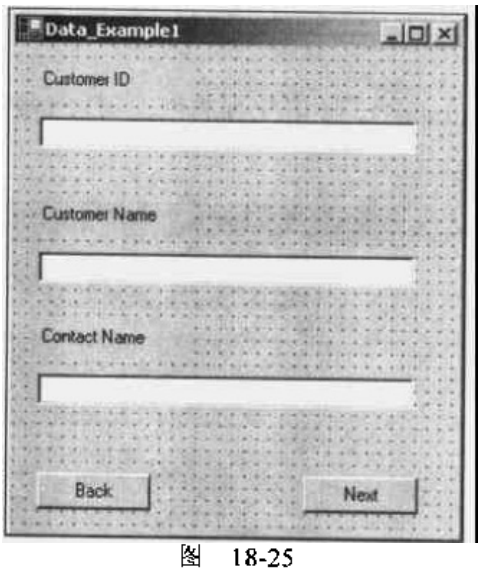


图　18-25

双击 cmdNext 按钮，在 Click 事件处理程序中添加下面的代码：

```
private void cmdNext_Click(object sender, System.EventArgs e)
{
    // Move to the next record in the DataSet, Customers table
    this.BindingContext[this.dsCustomers1,"Customers"].Position++;
}
```

这些代码使用表单的 BindingContext 改变当前的位置。所有的 Windows Form 都有

BindingContext 对象，它可以管理在表单上的数据绑定的控件。通过为 DataSet 的 Customers 表累加 BindingContext 的 Position 属性，我们可以移到到表中的下一条记录，并在所有的文本框中显示新数据。

我们也可以按同样的方式反向经过数据。在 cmdBack 单击事件中输入下面的代码即可：

```
private void cmdBack_Click(object sender, System.EventArgs e)
{
    // Move to the previous record in the dataset, customer table
    this.BindingContext[this.dsCustomers1,"Customers"].Position--;
}
```

18.6　添加列表

列表框可以像文本框那样绑定到 DataSet。在表单中添加表 18-5 中所示的对象。

表　18-5

控　件	属　性	值
列表框	Name	LstCustID
	DataSource	dsCustomers1.Customers
	DisplayMember	CustomerID

如果希望看到如何为 CustomerIDs 使用此列表框，可以向前推进一步，清除 Customer ID 文本框。表单应该如图 18-26 所示。

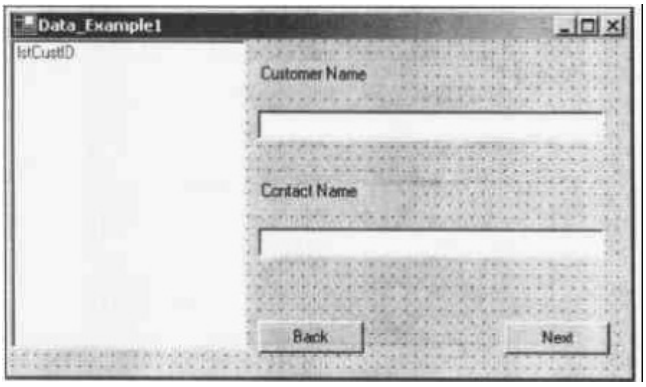


图　18-26

现在保存并运行这个应用程序。

现在在数据集中就有了一个列表框，列举了所有的 CustomerIDs。不知您是否注意到了，在列表框中选择项并没有移动在 BindingContext 中的指针，因此也没有改变在文本框中的数据。要解决这个问题开始感觉有点畏惧，但是它可以较好地练习解决面向对象的编程问题。

我们以所了解的内容开始。我们所知道的就是用户仅仅可以从列表框中选择一项。我们还知道对象集合有索引——数字索引。对象代表在列表框中的每一个项，其中列表框作为容器，

而每一项都有对应的索引号码。如果按在 DataSet 表中的记录的顺序生成列表框，则我们就知道每一条记录的索引号匹配列表框项的索引号。为了在 DataSet 中移动指针，就需要规定新位置。因此，如果可以把为被选择的列表框项的索引传递到 DataSet，我们就可以移动指针，并在文本框中显示该记录。

这确实有效，当处于设计模式中时，我们双击列表框，所访问的第一个事件处理程序就用于 SelectedIndexChanged 事件。这就是选择项时引发的事件。在列表框上双击并向事件处理程序添加代码：

```
private void lstCustID_SelectedIndexChanged(object sender,System.EventArgs e)
{
    // Pass the new index number to the bindingcontext
    this.BindingContext[this.dsCustomers1,"Customers"].Position =
        this.lstCustID.SelectedIndex;
}
```

我们再次使用 BindingContext 对象设置 Position 属性。但是在这一次不是规定我们正在累加或累减当前的索引号码，而是仅仅传递在列表框中选择的项的索引。

还需要作出两个更多的变动。命令按钮不会改变在列表框中选择的项。尽管移动了在 DataSet 中的指针，列表框却不反映这种移动。我们可以按与在 DataSet 中移动位置的同样的方式改变在列表框中选择的项的索引。按如下方式修改 cmdNext 按钮的单击事件：

```
private void cmdNext_Click(object sender, System.EventArgs e)
{
    // Move to the next record in the DataSet, Customers table
    this.BindingContext[this.dsCustomers1,"Customers"].Position++;

    // Synchronize the pointers of the listbox and DataSet
    this.lstCustID.SelectedIndex =
        this.BindingContext[this.dsCustomers1,"Customers"].Position;
}
```

我们所添加的这段代码会把在列表框中选择项的索引设置为在 DataSet 中的新位置。换句话来说，我们选择对应于在 DataSet 中的当前记录的列表框项。我们可以对 cmdBack 按钮进行完全相同的修改：

```
private void cmdBack_Click(object sender, System.EventArgs e)
{
    // Move to the previous record in the dataset, Customers table
    this.BindingContext[this.dsCustomers1,"Customers"].Position--;

    // Synchronize the pointers of the listbox and DataSet
    this.lstCustID.SelectedIndex =
        this.BindingContext[this.dsCustomers1,"Customers"].Position;
}
```

这样我们就可以确定用户界面了，运行工程就可以看到此界面。

18.7 添加 Datagrid

但是还需要添加最后一个对象。尽管通常不在我们所制作的完整表单中添加内容，但是数据网格是非常快速和方便的对象，可以添加到 Windows Form，显示数据，甚至提高用于变化的用户界面。

在表单中添加下面的数据网格，如表 18-6 所示。

表 18-6

控　件	属　性	值
Datagrid	Name	dgCustomers
	DataSource	dsCustomers1.Customers

所生成的表单如图 18-27 所示。

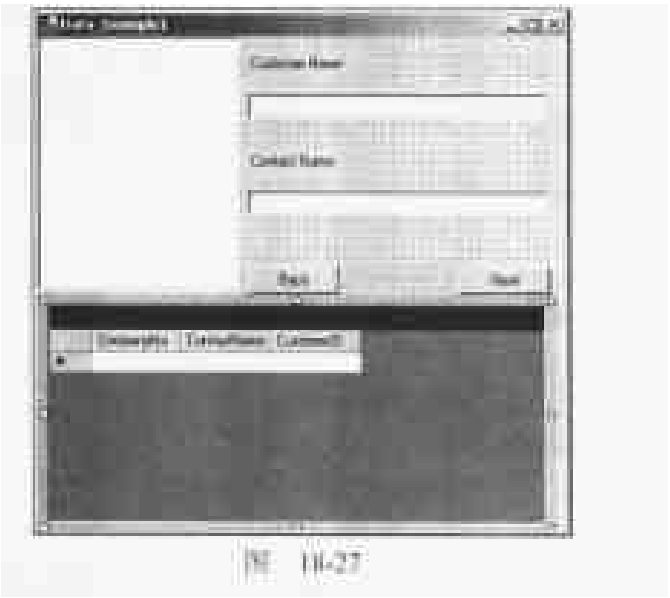


图 18-27

使用数据网格实现数据绑定时，所需要处理的附加改变就是设置它的 DataSource 属性——在此，即指 dsCustomers1.Customers。例如，不需要修改其他属性。当然，如果所使用的数据网格没有其他绑定的控件，那么就还需要配置连接和 DataSet，正如在本章前面所处理的那样，您还需要通过调用它的 Fill()方法生成 DataSet。

保存并运行该应用程序。Datagrid 应该随完整的表内容而生成。

18.7.1 格式化 Datagrid

甚至您还可以改变数据网格的外观，使得您的应用程序更为完美。最简单的方法是使用构建在对象中的一个模板。例如，如果不喜欢蓝色和灰色，可以修改模板：可以右击数据网格并选择 Auto Format，如图 18-28 所示。

在 Auto Format 对话框中，您可以选择任一个预定义的模板，以修整数据网格。但是，如果您更愿意手动改变外观，比如如果具有更愿意匹配的颜色，您可以调整背景颜色或字体样式的所有属性。

例如，可以将标题背景颜色从蓝色更改为红色。对数据网格的属性翻屏，直到找到 CaptionBackColor 为止。如果单击当前值，就会看到列表，该列表有 3 个标签：Custom，Web 和 System，如图 18-29 所示。

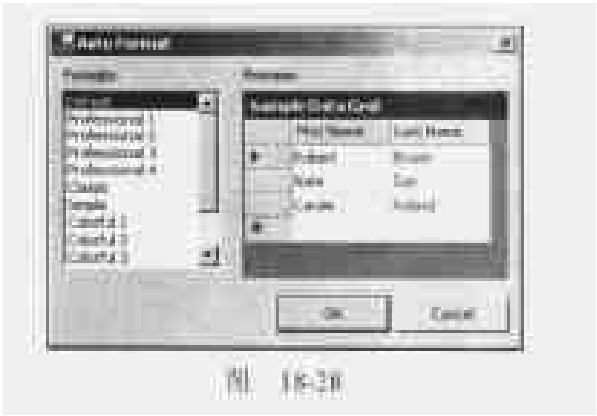


图 18-28



图 18-29

System 具有预定义的颜色，由 Windows 提供。例如，如果从 System 中选择了颜色 Desktop，而且用户改变了桌面颜色，则此颜色就会改变。有时这会导致难以预期的副作用，比如难于阅读。因此，改变这些值时要小心。现在，我们可以在 Web 选项卡上选择 Firebrick 颜色。窗格顶部的颜色就会变为深红色。

18.7.2　更新数据库

至此，我们可以从数据库中读取数据，可以在文本框和数据网格中改变数据，但是如果我们不保存对数据库本身所作的变动，则这种做法就没有什么用处。因此，我们可以立即看到如何使用在 DataSet 中的数据更新在数据库中的数据。对于这一节而言，我们会使用一点代码，而不是使用向导。将命令按钮添加到名称为 cmdUpdate 的表单中，则表单就会如图 18-30 所示。



图 18-30

现在给 cmdUpdate_Click 事件添加下面的代码：

```
private void cmdUpdate_Click(object sender, System.EventArgs e)
{
    // Pass the DataSet back to the database
    this.sqlDataAdapter1.Update(this.dsCustomers1.Customers);
}
```

当单击按钮时，对 DataSet 所作的任何变动都会传递到数据库。在第 19 章我们会更详细地介绍更新数据库的内容。

提示：

在编写本书时，如果使用了用于 SQL Server 的.NET 数据提供者，则 Update()仅仅可以与数据绑定的控件一起可靠地使用 —— 当对 OLE DB 提供者调用 Update()时，就会生成错误。

18.8 XML

到目前为止，我们一直在重点介绍关系数据库，但是在结束本章之前，我们还必须介绍一下其他数据格式 ——Extensible Markup Language(可扩展标记语言，XML)。XML 是一种以简单文本格式存储数据的方式，这意味着它可以被任何计算机读取。在后续章节介绍 Web 编程技术时可以看到，XML 是在 Internet 上传输数据的极好的格式。甚至对于人来说，它也很容易理解！

XML 的细节非常复杂，因此在此我们不介绍其所有的细节。但是其基本格式却非常简单，大多数时间您不需要了解 XML 的详细知识，因为 VS 通常会为您处理这些工作 —— 您基本上不必手动编写 XML 文档。我们已经说过，XML 在.NET 领域非常重要，因为它是传输数据的默认格式，因此理解其基本知识至关重要。

注意：

如果需要全面理解 XML，请参见《XML 入门经典》(ISBN 1-861003-41-2，由 Wrox Press 出版)。

18.8.1 XML 文档

在 XML 中的完整数据集就是 XML 文档。XML 文档可以是计算机上的物理文件，或是内存中的字符串。但是其本身必须是完整的，必须遵循一定的规则(稍后我们会介绍这些知识)。XML 文档由许多不同的部分组成。其中最重要的部分是 XML 元素，它包含文档的实际数据。

1. XML 元素

XML 元素包含一个开标记(被尖括号包围的元素名称，比如<myElement>)、元素中的数据和闭标记(与开标记相同，但是在左括号后有一个斜线：</myElement>)。

例如，我们可以定义一个放置图书书名的元素：

```
<book>Tristram Shandy</book>
```

如果您了解一些 HTML，就可能认为这非常类似——非常正确！事实上，HTML 和 XML 共享许多相同的语法。最大的不同是，XML 没有任何预定义元素——我们选择了自己元素的名称，因此对于可以拥有的元素的个数没有限制。要记住的最重要的一点是 XML——不管它使用什么样的名称——实际上不是语言。相反，它只是定义语言的标准(即 XML 应用)。每一种语言有自己独特的词汇库——这是一组特定的元素，可以用于文档和采用这些元素的结构。稍后会介绍，我们可以显式地限制在 XML 文档中允许使用的元素。另外，我们也允许任何元素以及使用文档的程序给出其本身的结构。

元素名称区分大小写，因此<book>和<Book>被认为是不同的元素。这表示，如果试图通过使用不明确区分大小写的闭标记关闭<book>元素，则 XML 文档就是非法的。XML 分析程序读取 XML 文档并将它们分析为单独的元素，它们会拒绝任何包含非法 XML 的文档。

元素也可以包含其他元素，因此我们可以修改此<book>元素，以便通过添加两个子元素包括作者和标题：

```
<book>
   <title>Tristram Shandy</title>
   <author>Lawrence Sterne</author>
</book>
```

但是却不允许重叠元素，因此在父元素的闭标记之前必须关闭所有的子元素。例如，我们不能完成下面的工作：

```
<book>
   <title>Tristram Shandy
   <author>Lawrence Sterne
   </title></author>
</book>
```

这是非法的，因为在<title>元素内打开了<author>元素，但是闭标记</title>却位于闭标记</author>之前。

对于所有元素必须具有闭元素的规定有一个例外。可以具有“空”元素，其中没有内嵌的数据或文本。在此，我们可以直接在开元素之后添加闭标记，如上所示，或者可以使用简短语法，在开元素末尾添加闭元素的斜线：

```
<book />
```

这等同于完整的语法：

```
<book></book>
```

2. 属性

与在元素体内存储数据一样，我们也可以在属性之内存储数据，属性被添加到元素的开标记之内。属性的形式为：

```
name="value"
```

其中属性值必须包含在单引号或双引号内。例如：

```
<book title="Tristram Shandy"></book>
```

或者：

```
<book title='Tristram Shandy'></book>
```

这都是合法的，但是下面的语句不合法：

```
<book title=Tristram Shandy></book>
```

此时，您可能要问，为什么我们在 XML 中需要两种方式存储数据。下面二者的区别是什么：

```
<book>
   <title>Tristram Shandy</title>
</book>
```

和：

```
<book title="Tristram Shandy"></book>
```

最坦白的答案是，二者并没有太大的区别。也无需优先使用哪一种方式。如果需要添加对后面出现的数据片断的更多说明信息，最好选择使用元素 —— 总是可以给元素添加子元素或属性，但是对属性就不能进行这样的操作。值得争论的是，元素是否更易于读取，更简洁(这只能根据个人不同的爱好来决定)。另一方面，如果未经压缩就在网络上传输文档，则属性会占用更少的带宽(即使压缩了这也没有什么区别)，而且如果放置对于文档的每一位用户而言无关紧要的信息，属性也是非常方便的。也许最好的选择是同时使用二者，可以根据自己的爱好选择使用任何一种方式来存储特定的数据项。但是确实没有硬性和简短的规则。

3. XML 声明

除了元素和属性之外，XML 文档还包含许多组成部分(XML 文档单独部分就是节点 —— 因此元素、元素内的文本和属性都是 XML 文档的节点)。如果您确实需要深入研究 XML，则这些都是非常重要的。但是只有一种类型的节点存在于几乎所有的 XML 文档中。这就是 XML 声明，如果包括了它，则它就必须作为文档的第一个节点。

XML 声明的格式类似于元素的格式，但是在尖括号内具有问号。它一贯的名称是 xml，并具有名称为 version 的属性；现在，其惟一可能的值是“1.0”。因此最简单的 XML 声明形式为：

```
<?xml version="1.0"?>
```

另外，它还可以包含属性 encoding(带有的值表示用于读取文档的字符集，比如“UTF-16”表示文档使用 16 位 Unicode 字符集)和 standalone(带有值“yes”或“no”，表示 XML 文档是否依赖于任何其他文件)。但是这些属性并不是必须的，您可以仅在 XML 文件中包括 version 属性。

4. XML 文档的结构

XML 最重要的一点就是它提供了一种结构化的组织数据的方式，它非常不同于关系数据库。可以看到，最新型的数据库系统在表中存储数据，表通过在各列中的值而相互关联。每一个表在行和列中存储数据——每一行表示一个单一的记录，每一列表示关于该记录的特定的数据项。相反，XML 数据是分层组织的，有点类似于 Windows Explorer 中的文件夹和文件。每一个文档必须具有单一的根元素，其中包含所有的元素和文本数据。如果在文档的顶级结点中有多个元素，那么文档就是不合法的 XML 文档。但是我们可以在顶级包括其他 XML 节点——通常是 XML 声明。所以下面的 XML 文档是合法的：

```
<?xml version="1.0"?>
<books>
   <book>Tristram Shandy</book>
   <book>Moby Dick</book>
   <book>Ulysses</book>
</books>
```

但是下面的文档就不合法：

```
<?xml version="1.0"?>
<book>Tristram Shandy</book>
<book>Moby Dick</book>
<book>Ulysses</book>
```

在此根元素之下，组织数据时具有极大的灵活性。不像关系数据(其中每一行具有相同的列数)，元素所包含的子元素数没有限制。尽管经常以与关系数据类似的方式组织 XML 文档(每一个记录具有一个元素)，XML 文档确实无需具有任何预定义的结构。这使得 XML 比关系数据库更适合于存储不规则的数据。例如，XML 可以用于标记文本文档。Web 标记语言，HTML 等，尽管不是严格的 XML 应用，但是它们确实紧密相关，在第 23 章我们会介绍这些内容。

18.8.2 XML 名称空间

正如每一个人都可以定义自己的 C#类一样，每一个人也可以定义自己的 XML 元素；这就导致了几乎相同的问题——我们如何知道哪一个元素属于哪一个词汇表？如果注意到本节的标题，就可以认识到可以以类似的方式回答这个问题。如同我们定义名称空间以组织 C#类型一样，我们使用 XML 名称空间定义 XML 词汇表。这使得我们将来自不同词汇表的元素包含到单一的 XML 文档中，却不必冒误解释元素的风险，因为(例如)两个不同的词汇表定义了一个<customer>元素。

XML 名称空间非常复杂，因此我们在此不详细介绍它，但是其基本语法非常简单。我们可以通过使用前缀(其后跟随有冒号)将具体的元素或属性关联到具体的名称空间。例如，<wrox:book>表示寄存在 wrox 名称空间中的<book>元素。但是如何知道名称空间 wrox 所表示的内容呢？为了使得此方法有效，我们必须保证每一个名称空间都是惟一的。最简单的办法是将前缀映射到某些独特的事物上。如下所示：在 XML 文档的某个位置，我们需要将我们所使

用的名称空间前缀关联到 Uniform Resource Identifier(惟一资源标识符，URI)。URI 有几种种类，但是最常见的类型是 Web 地址，比如“http://www.wrox.com”。

为了使用具体的名称空间标识前缀，我们可以使用元素内的 xmlns:<prefix>属性，将其值设置为可以标识名称空间的惟一的 URI。然后前缀就可以用于该元素的任何位置，包括任何内嵌的子元素。例如：

```
<?xml version="1.0"?>
<books>
   <book xmlns:wrox="http://www.wrox.com">
      <wrox:title>Beginning C#</wrox:title>
      <wrox:author>Karli Watson</wrox:author>
   </book>
</books>
```

在此我们使用带有<title>和<author>元素的 wrox:前缀，因为它们位于<book>元素内，其中定义了前缀。但是如果我们试图将此前缀添加到<books>元素，XML 就是非法的，因为并没有为此元素定义前缀。

我们也可以为使用 xmlns 属性的元素定义一个默认的名称空间：

```
<?xml version="1.0"?>
<books>
   <book xmlns="http://www.wrox.com">
      <title>Beginning C#</title>
      <author>Karli Watson</author>
      <html:img src="begcsharp.gif"
                xmlns:html="http://www.w3.org/1999/xhtml" />
   </book>
</books>
```

在此，将<book>元素的默认名称空间定义为“http://www.wrox.com”。因此在此元素内的所有内容都属于此名称空间，除非我们通过添加不同的名称空间前缀而显式地请求了其他内容，比如对<img>元素所做的处理那样(我们将它设置到与 XML 兼容的 HTML 文档所使用的名称空间中)。

18.8.3 格式良好并有效的 XML

到现在为止，我们已经介绍了“合法”XML。事实上，有两种合法形式的 XML。遵循 XML 标准的所有规则的文档就是“格式良好(well-formed)”的 XML。如果 XML 文档的格式不好，分析程序就不能正确地解释它，并拒绝该文档。为了有一个格式良好的 XML，对文档的要求如下：

- 有且只能具有一个根元素
- 对每一个元素都有闭标记(上面提到的简短语法除外)
- 不能有重叠元素——所有的子元素必须完全嵌套在父元素内

- 所有的属性必须包围在引号内

这些并不完全，但是却强调了 XML 新手容易犯下的一些错误。

完全遵循这些规则的 XML 文档仍然可以是无效的。记住我们在前面说过，XML 本身不是语言，而是定义 XML 应用程序的标准。格式良好的 XML 文档仅仅符合 XML 标准；为了使之有效，它们还必须符合对 XML 应用程序所规定的任何规则。并不是所有的分析程序都要检查文档是否有效；进行这种检查的分析程序就是验证分析程序。但是为了检查一个文档是否符合应用程序的规则，我们首先还需要一个方法可以规定它们的内容。

18.8.4 验证 XML 文档

XML 支持两种方法，定义在文档中可以放置那些元素和属性，以及其放置顺序——Document Type Definitions 和 Schemas。DTD 使用从 XML 的父继承的非 XML 语法，并逐渐地被模式所代替。DTD 不允许我们规定元素和属性的数据类型，因此在.NET Framework 的环境中使用得相对不灵活，也相对少一些。另一方面，模式非常灵活——它们允许我们规定数据类型，并写入与 XML 兼容的语法。但是，遗憾的是，模式非常复杂，定义它们需要不同的格式——即使在.NET 中也是如此！

模式

.NET 支持的模式具有两种不同的格式——XML Schema Definition 语言(XSD)和 XML-Data Reduced 模式(XDR)。模式可以包括在 XML 文档内，也可以放置在单独的文件中。这些格式互不兼容，您确实需要在编写 XML 文档之前非常熟悉 XML，因此我们在此不详细介绍它。但是，在模式中识别主要元素时它非常有用，因此我们要解释一下基本原则。为此，我们要介绍这个简单的 XML 文档的范例 XSD 和 XDR 模式，它们包含了两本 Wrox 的 C#图书的细节：

```
<?xml version="1.0"?>
<books>
   <book>
      <title>Beginning C#</title>
      <author>Karli Watson</author>
      <code>4982</code>
   </book>
   <book>
      <title>Professional C#</title>
      <author>Simon Robinson</author>
      <code>4990</code>
   </book>
</books>
```

XSD 模式

在 XSD 模式中的元素必须属于名称空间“http://www.w3.org/2001/XMLSchema”。如果未包括此名称空间，则就不会识别模式元素。

为了在另一文件中将 XML 文档与 XSD 模式关联，我们需要在根元素中添加 schemalocation

元素：

```
<?xml version="1.0"?>
<books schemalocation="file://C:\CSharp\XML\books.xsd">
  ...
</books>
```

下面简要介绍范例 XSD 模式：

```
<schema xmlns="http://www.w3.org/2001/XMLSchema">
  <element name="books">
    <complexType>
      <choice maxOccurs="unbounded">
        <element name="book">
          <complexType>
            <sequence>
              <element name="title" />
              <element name="author" />
              <element name="code" />
            </sequence>
          </complexType>
        </element>
      </choice>
      <attribute name="schemalocation" />
    </complexType>
  </element>
</schema>
```

在此首先需要注意的是，我们设置的默认名称空间为 XSD 名称空间。这就告诉分析程序，在文档中的所有元素属于模式。如果我们不规定此名称空间，分析程序就认为元素仅是普通的 XML 元素，认识不到需要使用它们进行验证。

完整的模式包含在称为<schema>的元素内(使用小写“s”——记住，大小写非常重要！)在文档中的每个元素都必须由<element>元素来表示。此元素具有指示元素名称的 name 属性。如果元素包含嵌套子元素，我们就必须在<complexType>元素内为这些子元素包含<element>标记。在<complexType>元素中，我们规定子元素发生的方式。例如，我们可以使用<choice>元素规定对子元素进行选择操作，或使用<sequence>规定子元素必须以它们在模式中列举的顺序出现。如果一个元素可能出现多次(如<book>元素)，就需要在其父元素内包括 maxOccurs 属性。将其设置为“unbounded”，表示元素可以按我们的需要的次数出现。最后，任何属性都必须由<attribute>元素表示，包括 schemalocation 属性，它可以告诉分析程序模式所在的位置。我们将其放置在子元素列表的结尾处。

XDR 模式

为了将外部的 XDR 模式应用到 XML 文档上，我们可以为带有值“x-schema: <schema_filename>”的文档规定一个名称空间：

```
<?xml version="1.0"?>
<books xmlns="x-schema:books.xdr">
    ...
</books>
```

下面的模式是我们刚才介绍的 XSD 模式的 XDR 模式。可以看到，它们的区别非常大：

```
<Schema xmlns="urn:schemas-microsoft-com:xml-data">
    <ElementType name="title" content="textOnly" />
    <ElementType name="author" content="textOnly" />
    <ElementType name="code" content="textOnly" />
    <ElementType name="book" content="eltOnly">
        <group order="seq">
            <element type="title" />
            <element type="author" />
            <element type="code" />
        </group>
    </ElementType>
    <ElementType name="books" content="eltOnly">
        <element type="book" />
    </ElementType>
</Schema>
```

我们再一次设置默认的名称空间，以告诉分析程序，文档中的所有元素属于模式定义：这一次的名称空间是“urn:schemas-microsoft-com:xml-data”。注意，它与 XSD 模式不同，此模式是专用格式，所以不可用于非 Microsoft 产品。事实上，当与 SQL Server 和 Microsoft 的数据库服务器一起使用时，XDR 模式非常有用，因为它对 XDR 的内置支持。

这一次我们的根元素是带有大写字母“S”的<Schema>。这个根元素再次包含了完整的模式定义(记住 XML 文档必须具有单一的根元素)。但是之后，就有了较大的区别 —— 在文档中出现的元素按反序出现！其原因是，在文档中的每一个元素由模式中的<ElementType>元素表示，它为每一个子元素包含了一个<element>元素(注意在此是小写“e”)。在<element>标记内，我们设置 type 属性，指向<ElementType>元素 —— 这必须是已经定义好的。如果我们希望限制子元素出现的方式，就可以在<ElementType>元素内使用<group>元素，并设置其 order 属性。在此，我们将其设置为“seq”，规定元素按在模式中相同的顺序发生 —— 与在 XSD 模式中的<sequence>标记相同。

试试看：在 VS 中创建 XML 文档

现在我们已经介绍了 XML 幕后的基本理论，接下来就可以创建 XML 文档。幸运的是，VS 为我们完成了大部分工作，甚至基于 XML 文档创建了 XSD 模式，我们无需编写一行代码！

(1) 打开 VS，从菜单中选择 File | New | File...(不需要打开一个工程)，出现如图 18-31 所示的对话框。

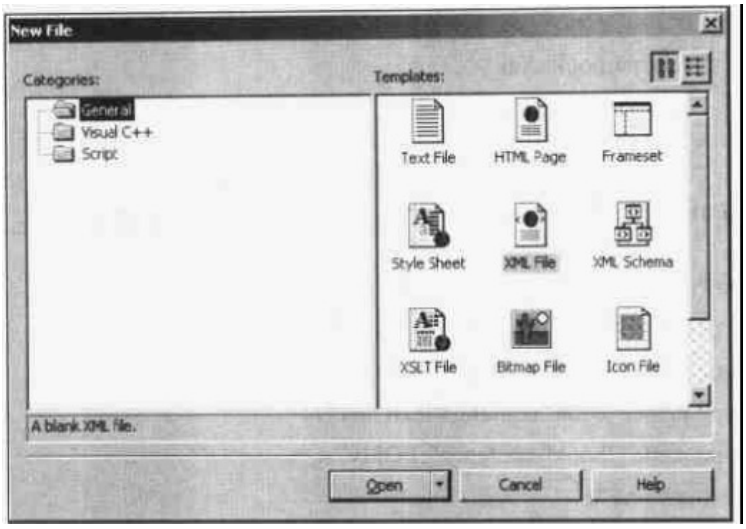


图　18-31

(2) 在 New File 菜单中，选择 XML File，单击 Open。VS 会为我们创建一个新的 XML 文档。注意 VS 添加 XML 声明的方式，以 encoding 属性结束(其属性和元素也是彩色的，但是在黑白输出中其效果不太好)，其结果如图 18-32 所示。

```
<?xml version="1.0" encoding="utf-8" ?>
```

图　18-32

(3) 通过按下 Ctrl+S 组合键或者在菜单中选择 File | Save XMLFile1 而保存文件。VS 会询问要将文件保存的位置，以及文件的名称；将其保存在 BegCSharp\Chapter18 文件夹中，其名称为 GhostStories.xml。

(4) 将光标移动到 XML 声明下面的代码行，输入文本<stories>。注意当我们输入大于号关闭开标记时，VS 自动置入结束标记的方式，如图 18-33 所示。

```
<?xml version="1.0" encoding="utf-8" ?>
<stories></stories>
```

图　18-33

(5) 输入下面的 XML 文件：

```
<?xml version="1.0" encoding="utf-8" ?>
<stories>
    <story>
        <title>A House in Aungier Street</title>
        <author>
            <name>Sheridan Le Fanu</name>
            <nationality>Irish</nationality>
        </author>
        <rating>eerie</rating>
    </story>
```

```
    <story>
        <title>The Signalman</title>
        <author>
            <name>Charles Dickens</name>
            <nationality>English</nationality>
        </author>
        <rating>atmospheric</rating>
    </story>
    <story>
        <title>The Turn of the Screw</title>
        <author>
            <name>Henry James</name>
            <nationality>American</nationality>
        </author>
        <rating>a bit dull</rating>
    </story>
</stories>
```

(6) 在代码窗口中右击 XML，从弹出菜单中选择 View Data。这使得 VS 可以按列表形式显示 XML 文件的数据，似乎这些数据来自关系数据库，如图 18-34 所示。

(7) 我们实际上可以在此表中编辑数据，因此我们可以修改 XML 文档，而无需输入标记。在网格底部的空白行中的 title 列上的小框中单击，输入 Number 13。再移动到 rating 框，输入 mysterious。这就输入了一个新内容，但是我们仍然需要输入作者。为此，单击新行旁边的加号。就会出现<author>元素的链接，如图 18-35 所示。



图 18-34



图 18-35

(8) 单击此链接，会显示另一个表，其中我们可以输入作者的姓名和国籍。在两列中输入 MR James 和 English(要确保输入国籍之后按下回车键，否则数据就会丢失)，其结果如图 18-36 所示。

(9) 在数据网格上右击，选择 XML Source。这就会在闭标记</stories>之前添加一个新的<story>元素：

```
<story>
    <title>Number 13</title>
    <rating>mysterious</rating>
    <author>
        <name>MR James</name>
```

```
        <nationality>English</nationality>
    </author>
</story>
```

(10) 最后，我们使 VS 为此 XML 文档创建 XSD 模式。右击网格，并选择 Create Schima。VS 会创建一个 XSD 模式，但是它还创建图表，可视化地显示模式，如图 18-37 所示。

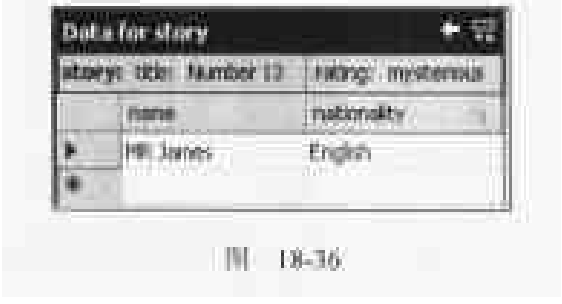


图 18-36

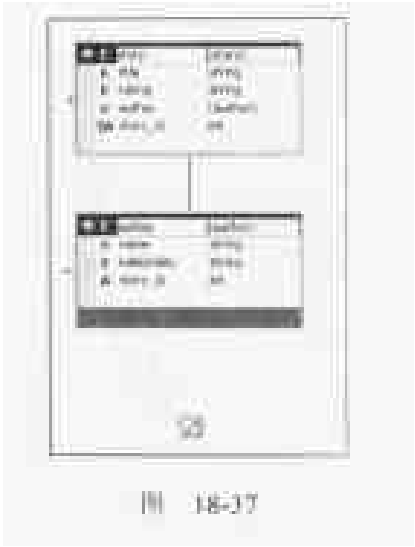


图 18-37

注意：

VS 是如何添加 story_Id 域，以标识每一个<story>元素的。如果我们希望将 XML 文档转换为关系数据，这就非常有用，因为数据库需要一种方法来识别哪一个作者创作了哪一个故事 —— 在 XML 中，我们将<author>元素放置在<story>元素之下。

(11) 要查看实际的模式，在图表上右击，选择 View XML Source。

18.8.5 C#中的 XML 自动文档

至此，我们以非常抽象的方式介绍了 XML，您可能要问，“如何在 C#中使用 XML？XML 可以做什么工作？”在本书剩余章节我们会介绍 XML 更实际的用法，但是只有一种 XML 用法是针对 C#使用的。VS 允许我们在 C#程序文件中创建具体的注释，我们可以摘录 C#程序文件并将其编译进单独的文件中。这个文件就是 XML 文档。这意味着 XML 分析程序(比如构建到 VS.NET 的分析程序)，可以很容易地从此文件中摘录数据。VS 会格式化此数据，以显示为 Web 页 —— 因此我们仅仅从代码中的少数注释就可以得到外观时髦的文档！

VS 自动地在代码中放置了少量的特定注释，因此您可能注意到了它们。开始是 3 条斜线(///)，而不是通常的两条斜线。例如，VS 为 Class1 类在定义之前添加代码行，当创建 Windows 或控制台应用程序时，VS 自动将 Class1 类添加到工程：

```
/// <summary>
/// Summary description for Class1.
/// </summary>
```

现在，您可以迅速地将<summary>和</summary>识别为 XML 元素的开标记和闭标记。为

了演示如何从这个简单的元素进到自动文档中，我们可以在第 4 章的 Mandelbrot 应用程序中添加一些注释。

试试看：对类进行文件管理

(1) 打开第 4 章中的 Ch04Ex06 工程。

(2) 首先在<summary>元素中添加一些更有用的内容，VS 为 Class1 创建了<summary>元素。按如下方式改变标记之间的文本(不要删除首行的 3 条斜线)：

```
/// <summary>
/// This class generates Mandelbrot sets in the console window!
/// </summary>
```

(3) 接下来，为 Main()方法添加一些文档。在方法定义的开始处添加空白行，并在其中输入 3 条斜线。如图 18-38 所示。



图 18-38

VS 为<summary>元素添加 XML 标记(与为 Class1 所添加的标记相同)，但是它还检查下面的代码行，确认它是方法定义，并检查其参数。然后为每一个参数创建一个<param>元素(在此仅有一个参数——即称为 args 的字符串数组，当执行控制台应用程序时，我们可以用来读取命令行的数据)。参数的名称存储在 name 属性中。

(4) 在<sunmary>和<param>元素中添加一些描述性文本：

```
/// <summary>
/// This is the Main() method for Class1 -
/// this is where we call the Mandelbrot generator!
/// </summary>
/// <param name="args">
/// The args parameter is used to read in
/// arguments passed from the console window
/// </param>
```

顺便还要注意 VS 知道您位于 autodoc 注释中间的方式，当到达屏幕的边缘并按下回车键时，它自动在新代码行上添加 3 条斜线！

(5) 接下来将这些注释转化为文档。在菜单中，选择 Tools | Build Comment Web Pages...，其对话框如图 18-39 所示。

如果 VS 解决方案中包含多个工程，我们在此可以选择是为完整的解决方案构建文档，还是为具体的工程构建文档。在这个解决方案中仅有一个工程，因此选择 Build for entire Solution。

我们还可以选择目录，其中可以保存所生成的 Web 页面。

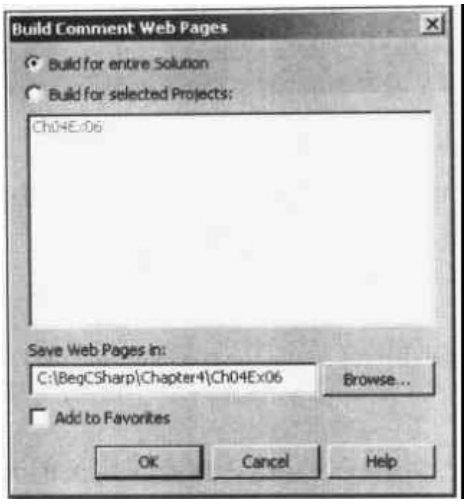


图　18-39

(6) 单击 OK 按钮，使得 VS 生成 Web 页面。因为 Internet Explorer 已被集成到 VS 中，因此当我们单击 OK 按钮时，VS 会自动显示解决方案文档的主页，如图 18-40 所示。

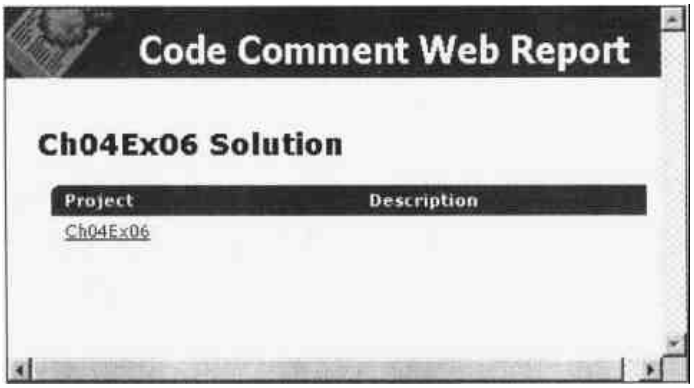


图　18-40

(7) 单击 Ch04Ex06 链接，为工程打开文档(在解决方案中仅有一个文档)，其结果如图 18-41 所示。

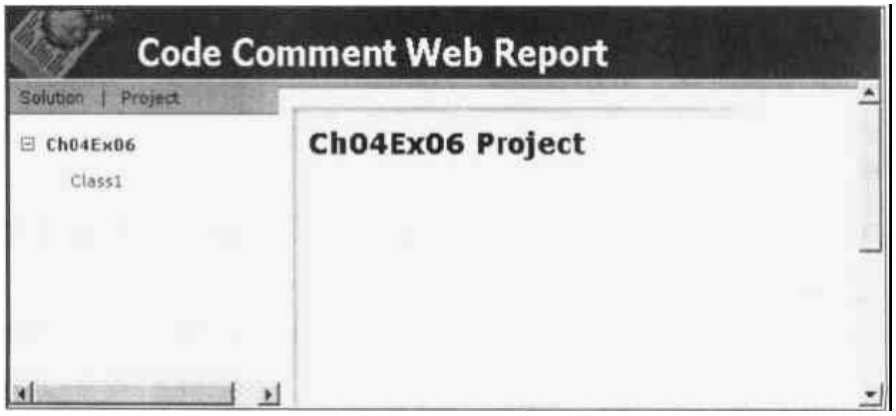


图　18-41

(8) 为 Class1 类打开文档，为此单击紧邻 Ch04Ex06 链接的加号，然后单击在其下出现的 Class1 链接，其结果如图 18-42 所示。

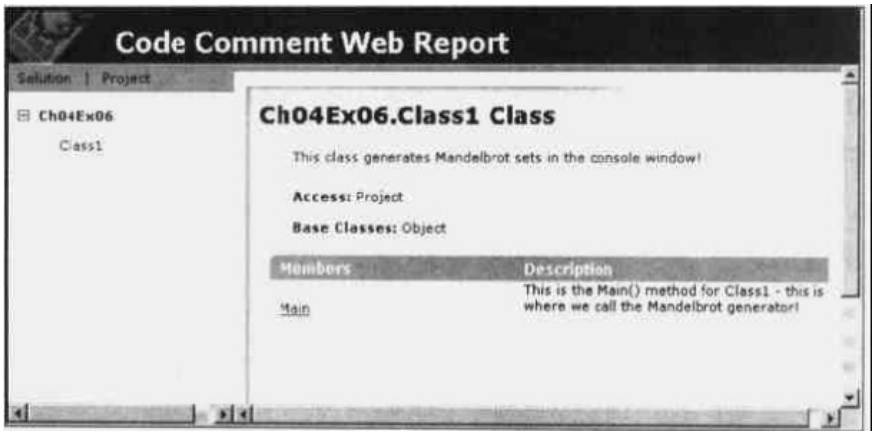


图　18-42

此处我们可以看到从注释中生成的实际页面！文档化的类的页面具有所输入的简短的说明，一列类的成员，以及它们的简要说明。它还给出 VS 说明其自身的信息——类的完全名称，包括名称空间、类的访问级别和从其中派生的类。

(9) 最后，单击 Main 链接，查看 Main()方法的文档。此页面显示方法的说明及其参数，以及方法的完整签名，如图 18-43 所示。

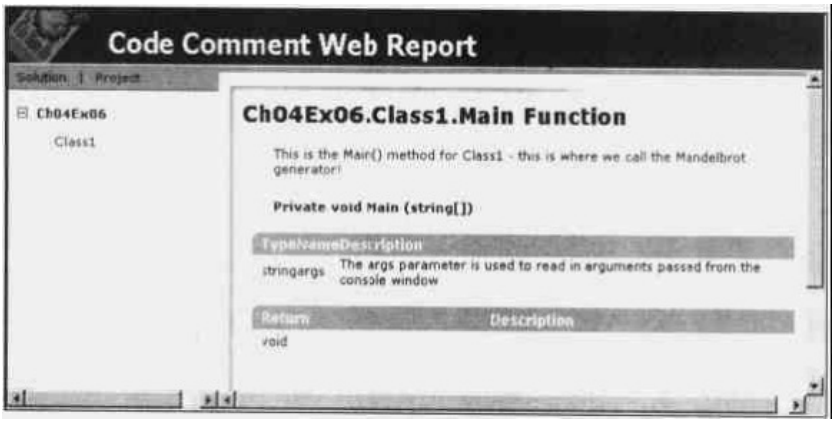


图　18-43

遗憾的是，此图的格式并不好，因为 args 参数表稍微有点弯曲。

18.9　小结

Visual Studio.NET 给我们提供了一套完整的工具，可以快速地创建连接到数据库的应用程序。而且几乎不涉及到代码。纵览本章，我们介绍了显示数据的几种机制。从基本的文本框，到更高级的数据网格，它们都依赖于 ADO.NET 和所有的 Windows Form 内生的 BindingContext 的强大功能和灵活性。

在本章，我们学习了如何使用这些工具连接到数据源，并使用少量代码创建数据驱动的应

用程序。但是自动生成的代码从不像手动编写的代码那样有效，因此在第 19 章我们将要学习如何以本章介绍的知识为基础进行构建，并继续编写自己的 ADO.NET 代码。

在本章最后一节，我们讨论了 Extensible Markup Language(XML)，它是存储和检索数据的一种文本格式。我们还介绍了需要遵循的规则，以保证 XML 文档有良好的格式，我们介绍了如何针对 XSD 和 XDR 模式来验证它们。在此，我们还学习了 VS 如何为我们完成硬编码工作，甚至从 XML 文档中创建 XSD 模式。最后，我们介绍了 C#以 XML 格式从特定的注释中为代码创建文档的功能。为了使您清楚 XML 的作用，在第 19 章我们将要介绍 XML 和 ASO.NET 是如何复杂地相互链接的。

18.10　练习

(1) 修改第 1 个范例，在 sqlDataAdapter1 中添加列 ContactTitle。

(2) 在显示 ContactTitle 信息的表单中添加文本框。

(3) 创建新应用程序，显示 Northwind 数据库中的 Products 表的数据。

(4) 修改“在 DataSet 中导航”范例代码，当在最后一条记录上单击 Next 按钮时翻屏回到 DataSet 的开端，当在第一条记录上单击 Back 按钮时，翻屏回到 DataSet 的结束处。

(5) 为 Ch18Ex01 应用程序的最后版本创建一组 XML 自动文档的 Web 页面。

第19章　使用ADO.NET访问数据

在前面章节中，我们学习了如何使用 Visual C#中的向导和代码生成工具生成可以访问关系数据的用户界面和控件。

在本章，我们要学习如何使用直接由程序员编写的 C#代码完成相同的数据访问工作，这次不同于前面章节中所使用的向导生成的代码方法。我们要使用 ADO.NET 数据访问类来完成数据访问。首先，我们概要介绍 ADO.NET。然后要学习 ADO.NET 中类的结构，最后要构建一些使用 ADO.NET 的代码范例。

19.1　ADO.NET 概述

ADO.NET 是与 C#和 Microsoft.NET Framework 一起使用的类集的名称，用于以关系型的、面向表的格式访问数据。这包括关系数据库，比如 Microsoft Access 和 SQL Server，以及其他数据库，甚至还包括非关系数据源。ADO.NET 被集成到.NET Framework，用于.NET 语言，尤其是 C#。

ADO.NET 位于 System.Data.dll 装配件(关于装配件的内容在第 21 章介绍)。从某种意义上来说，ADO.NET 就是 System.Data.dll 装配件，因为从定义上来说，在此装配件中所包含的任何类都是 ADO.NET 的一部分。这包括所有的 System.Data 名称空间，和来自 System.Xml 名称空间的一个类。在第 18 章我们介绍了 XML，本章我们会接着介绍 ADO.NET 对 XML 的支持。

19.1.1　ADO.NET 名称的来源

您可能要问，为什么这一部分.NET Framework 被称为 ADO.NET 呢？为什么不直接称之为 System.Data 并按此名称使用它呢？ADO.NET 的名称起源于 ADO(ActiveX Data Objects)，这是用途广泛的类组，用于在以往的 Microsoft 技术中访问数据。之所以使用 ADO.NET 名称，是因为 Microsoft 希望表明，这是在.NET 编程环境中优先使用的数据访问接口。

ADO.NET 的作用与 ADO 相同，提供易于使用的类集，以访问数据，ADO.NET 的功能得到更新和增强，可以用于.NET 编程环境。虽然它的作用与 ADO 相同，但是在 ADO.NET 中的类、属性和方法与 ADO 有很大的不同。

19.1.2　ADO.NET 的设计目标

下面我们简要介绍一下 ADO.NET 的设计目标。它们是：简单地访问关系和非关系数据，统一 XML 和关系数据访问，支持 Internet 上的多层应用程序，与上一代技术相比，它可以扩充以支持更多的数据源。

1. 简单地访问关系数据

ADO.NET 的主要目标是提供对关系数据的简单访问功能。坦白地说，易于使用的类描述关系数据库中的表、列和行。另外，ADO.NET 引入了 DataSet 类，它代表来自封装在一个单元中的关联表中的一组数据，维持它们之间完整的关系。这是在 ADO.NET 中的新概念，可以显著地扩展数据访问接口的功能。

2. 可扩展性

ADO.NET 可以扩展 ——它为插件.NET 数据提供者(也称为可管理提供者)提供了框架，这些提供者被构建，以便从任何数据源读取和写入数据。ADO.NET 提供了两种内置的.NET 数据提供者，一种用于 OLE DB 数据源，另一种用于 Microsoft SQL Server。可以通过 OLE DB 访问数据格式(比如 Microsoft Access)、第三方数据库和非关系数据。另外，Microsoft 最近预演了用于 ADO.NET 的 ODBC.NET 数据提供者，它允许.NET 访问更多的旧的数据格式和第三方数据库。

3. 支持多层应用程序

ADO.NET 用于多层应用程序。这是当今商业和电子商务应用程序最常见的体系结构。在多层体系结构中，应用逻辑的不同部分运行在多个服务器或进程中，每一部分就称为一层。

ADO.NET 使用开放的 Internet 标准 XML 格式在层之间通信，允许数据通过 Internet 防火墙来传递，并允许以非 Microsoft 技术来实现一层或多层。

4. 统一 XML 和关系数据访问

ADO.NET 另一个重要的目标是沟通在行、列和 XML 文档中的关系数据，其中 XML 文档具有分层的数据结构。结合 XML 和 ADO.NET 构建的.NET 技术可以扩展.NET 的用法。

我们已经介绍了 ADO.NET 的目标，下面就介绍实际的 ADO.NET 类。

19.2　ADO.NET 类和对象概述

图 19-1 显示了 ADO.NET 中的基本类。注意，这不是一个继承图表，而是给出了最常用的类之间的关系。



图 19-1

可以把类分为.NET 数据提供者对象和用户对象。提供者对象专用于每一种类型的数据源；

专用于提供者的用户对象完成在数据源中实际的读取和写入工作。用户对象是将数据读入到内存中后用来访问和操纵数据的对象。用户对象以非连接方式使用；甚至在数据库连接关闭之后，也可以使用内存中的数据。提供者对象要求活动的连接，您可以使用它们首先读取数据，然后根据需要，通过用户对象使用内存中的数据，和/或通过使用提供者对象更新数据源中的数据，并将变动写回到数据源中。

19.2.1　提供者对象

即在每一个.NET 数据提供者中定义的对象，其名称前带有专用于提供者的名称。因此，用于 OLE DB 提供者的实际连接对象就是 OleDbConnection；用于 SQL Server.NET 提供者的类就是 SqlConnection，ODBC.NET 提供者的连接类就是 OdbcConnection。

1. 连接对象

连接对象是您要使用的第一个对象，被要求用于任何其他 ADO.NET 对象之前。它提供了到数据源的基本连接。如果您所使用的是要求用户名和密码的数据库，或者是位于远程网络服务器上的数据库，则连接对象就可以提供建立连接并登录的细节。

注意：

如果您非常熟悉传统的 ADO，您可能注意到，在传统的 ADO 中起类似作用的 Connection 和其他对象具有与 ADO.NET 对应对象相类似的名称。

2. 命令对象

可以使用此对象发出命令，比如对数据源的 SQL 查询，如“SELECT * FROM Customers”查询在 Customers 表中的数据。专用于提供者的名称包括用于 SQL Server 的 SqlCommand 和用于 OLE DB 的 OleDbCommand。

3. CommandBuilder 对象

此对象用于构建 SQL 命令，可以在基于单一表查询的对象中进行数据修改。当我们学习如何更新数据时，会更详细地介绍该对象。专用于提供者的名称包括用于 SQL Server 的 SqlCommandBuilder 和用于 OLE DB 的 OleDbCommandBuilder。

4. DataReader 对象

这是一个快速而易于使用的对象，可以从数据源中读取仅能前向和只读的数据流(比如所发现的顾客集合)。此对象具有最好的功能，可以简单地读取数据；关于此对象的用法可以参见下面的范例。专用于提供者的名称包括用于 SQL Server 的 SqlDataReader 和用于 OLE DB 的 OledbDataReader。

5. DataAdapter 对象

这是一个通用的类，可以执行针对数据源的各种操作，包括更新变动的数据，填充数据集以及其他操作，在下面的范例中我们会介绍这些知识。专用于提供者的名称包括用于 SQL Server

的 SqlDataAdapter 和用于 OLE DB 的 OledbAdapter。

19.2.2 用户对象

还有用于定义 ADO.NET 的断开的、用户端的对象：

1. DataSet 对象

此对象表示一组相关表，在应用程序中这些表作为一个单元被引用。例如，Customers，Orders 和 Products 都是表示每一位顾客或他们从公司中定购的产品的 DataSet 中的表。有了此对象，您就可以快速从每一个表中获取所需要的数据，当与服务器断开时检查并修改数据，然后在一次操作中就使用这些修改的数据更新服务器。

DataSet 具有属性，使得您可以访问低级对象，这些对象代表单独的表、行、列和关系。这些对象是：

2. DataTable 对象

此对象代表在 DataSet 中的一个表。例如，Customers，Orders 或 Products。

3. DataColumn 对象

此对象代表表中的一列，比如 OrderID 或 CustomerName。

4. DataRow 对象

此对象代表来自表的关联数据的一行；例如某位顾客的 CustomerID、姓名和地址等。

5. DataRelation 对象

此对象代表通过共享列而发生关系的两个表之间的关系；例如 Orders 表具有 CustomerID 列，标识发出订单的顾客。创建 DataRelation 对象来代表共享列 CustomerID 的 Customers 和 Orders 表的关系。

现在我们了解了在 ADO.NET 中的对象的总体结构。还有更多的对象，但是我们要跳过这些对象的细节，介绍一些范例，理解此对象的工作原理。

19.2.3 使用 System.Data 名称空间

在 C#代码中使用 ADO.NET 的第一步是引用 System.Data 名称空间，其中含有所有的 ADO.NET 类。将下面的 using 指令放置在使用 ADO.NET 的程序的开端：

```
using System.Data;
```

1. SQL Server .NET 数据提供者

接下来，我们需要为将要使用的具体数据源引用.NET 数据提供者。如果您所使用的是 SQL Server(版本 7 或更高级的版本)或 MSDE，则通过使用 SQL Server 专用的.NET 数据提供者就可以获得最好的性能和对基础性的功能的最直接的访问，按如下方式使用 using 指令可以引用 SQL Server 专用的.NET 数据提供者：

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

2. OLE DB .NET 提供者

对于不是 SQL Server 的大多数数据源(Microsoft Access，Oracle 以及其他数据源)而言，可以使用 OLE DB.NET 数据提供者，按如下方式使用 using 指令引用它：

```
using System.Data.OleDb;
```

如果有适合您的数据库的.NET 数据提供者，则您可能反而希望使用这种.NET 数据提供者，它有自己的 using 指令。SQL Server.NET 数据提供者是一个范例性的针对产品的.NET 数据提供者。

3. ODBC .NET 数据提供者

如果您的数据源没有自己的或 OLE DB 提供者(比如 PostgrsSQL 或其他一些第三方数据库)，则可以使用 ODBC.NET 数据提供者。它可以不随 Visual Studio.NET 下载(见 http://msdn.microsoft.com/downloads/) —— 它位于 ODBC .NET 数据提供者测试版的.NET Framework 之下。安装之后，它就被放置在 System.Data.Odbc.dll 装配件中；当编译时它就要求具体的引用。ODBC 提供者可以通过下面的 using 指令来引用：

```
using System.Data.Odbc;
```

试试看：简单的单一表数据访问范例

在此范例中，我们要从一个表中获取一些数据，比如在 SQL Server/MSDE Northwind 范例数据库中的 Customers 表 —— 这个数据库与我们在第 18 章介绍的数据库相同。

此表的行和列包含关于 Northwind 商行的顾客的数据。在 Visual Studio.NET 中此表如图 9-2 所示。



图　19-2

我们可以创建一个程序，通过使用 ADO.NET 查询此表的 CustomerID 和 CompanyName 列。

此程序和本章后面的范例的创建方式与创建 Visual Studio.NET 中的控制台应用程序工程的方式相同；请参照本章中的此范例和其他范例的编译和执行步骤。

启动 Visual Studio.NET，创建一个新的 C#工程：在主菜单中选择 File | New | Project...，然后从 Visual C# Projects 菜单中选择 Console Application 模板。可以将此命名为

DataReaderExample，如图 19-3 所示。

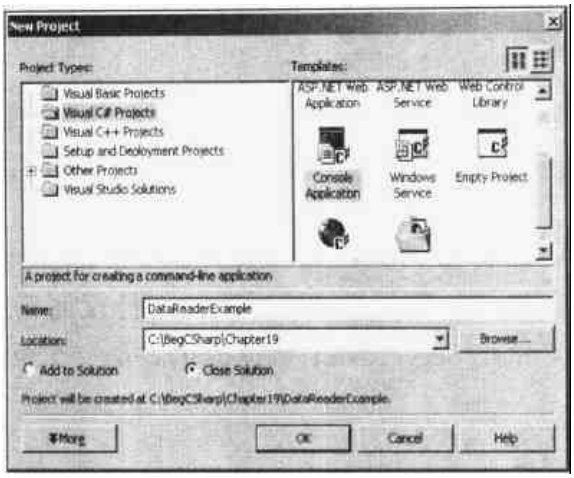


图　19-3

将文件 Class.cs 重命名为 DataReaderExample.cs，然后输入下面完整的 C#程序，此程序并不太长：

```
using System;
using System.Data;                  // Use ADO.NET namespace
using System.Data.SqlClient;        // Use SQL Server data provider namespace

class DataReaderSqlExample
{
    public static void Main()
    {
        // Specify SQL Server-specific connection string
        SqlConnection thisConnection = new SqlConnection(
            @"Data Source=(local);Integrated Security=SSPI;" +
            "Initial Catalog=northwind");

        // Open connection
        thisConnection.Open();

        // Create command for this connection
        SqlCommand thisCommand = thisConnection.CreateCommand();

        // Specify SQL query for this command
        thisCommand.CommandText =
            "SELECT CustomerID, CompanyName from Customers";

        // Execute DataReader for specified command
        SqlDataReader thisReader = thisCommand.ExecuteReader();

        // While there are rows to read
```

```
        while (thisReader.Read())
        {
            // Output ID and name columns
            Console.WriteLine("\t{0}\t{1}",
            thisReader["CustomerID"], thisReader["CompanyName"]);
        }

        // Close reader
        thisReader.Close();

        // Close connection
        thisConnection.Close ();
    }
}
```

编译此程序(F7)并运行它(Ctrl-F5)。可以看到一列顾客 ID 和公司名称，如下所示：

```
ALFKI    Alfreds Futterkiste
ANATR    Ana Trujillo Emparedados y helados
ANTON    Antonio Moreno Taquería
AROUT    Around the Horn
BERGS    Berglunds snabbköp
BLAUS    Blauer See Delikatessen
...
```

我们将逐行研究此范例，详细介绍其工作原理。

代码的说明

第一步是引用前面描述的 System.Data 名称空间和提供者，在这些范例中，我们要使用 SQL Server .NET 提供者，因此我们需要在程序的开端使用下面的代码行：

```
using System.Data;
using System.Data.Sql;
```

连接到数据源

接下来需要连接到数据源。通过创建一个使用连接字符串的连接对象就可以完成。连接字符串包含希望连接到的数据库的提供者的名称、登录信息(数据库用户、密码等)以及希望使用的具体数据库的名称。我们再介绍一下这个连接字符串的具体元素；但是要记住，在不同的数据提供者之间的连接字符串区别非常大，因此，如果您所使用的数据提供者与本范例中所使用的数据提供者不同，则还需要查询所使用的提供者的具体的连接信息(在本章后面的范例中给出了一点这方面的信息)。

我们创建连接对象的代码行如下：

```
SqlConnection thisConnection = new SqlConnection(
    @"Data Source=(local);Integrated Security=SSPI;" +
    "Initial Catalog=northwind");
```

SqlConnection 是一个用于 SQL .NET 数据提供者的连接对象的名称；如果我们正在使用 OLE DB，则可以创建一个 OleDbConnection。连接字符串包含由分号分隔的命名项；我们逐个对它们进行介绍。第一个项是：

```
Data Source=(local);
```

这仅仅是您正在访问的 SQL Server 的名称。(local)是一个非常方便的 SQL Server 简短名称，它引用运行在当前机器上的服务器实例。如果在您的计算机上已经运行了 SQL Server，则此名称就可以起到这样的作用。

.NET Framework SDK 范例可以有选择地安装 SQL Server 桌面引擎(也称为 MSDE)，以运行范例。此实例的默认名称是(local)\NetSDK。因此可以试用的另一个服务器名称是：

```
Data Source=(local)\NetSDK;
```

注意连接字符串前面的@符号表示一个字符串字面量，使得此名称中的反斜线可以发挥作用；否则就需要双反斜线(\\)将反斜线字符转义到 C#字符串中。在此也可以规定实际计算机的名称：

```
Data Source=ROADRUNNER\NetSDK;
```

如果安装了正在使用的 SQL Server，则就会知道 SQL Server 实例的名称。否则，就必须查询 SQL Server 或询问网络管理员，才能知道所使用的名称。

连接字符串的下一部分规定如何登录到数据库；这一个短语规定使用 Windows 登录和 SQL Server 登录的内置集成安全，这样就不需要规定单独的用户和密码：

```
Integrated Security=SSPI;
```

SSPI 即 Security Support Provider Interface(安全支持提供者接口)；它用于规定 SQL Server 和 Windows 的标准内置安全。最后规定了您希望使用的具体数据库，在此即 Northwind 范例：

```
Initial Catalog=northwind
```

尽管处于节省空间的原因，许多数据库管理员选择忽略 Northwind 范例，但是在默认的 SQL Server 安装中，总是会出现 Northwind 范例。程序的其余部分的作用与前面范例的程序相同，因此我们就不再进行解释。

现在，就有了一个为我们的计算机和数据库配置的连接对象(但是连接还未激活；因此必须打开连接)。

Open 方法

有了连接对象之后，第一件工作就是打开它，这就建立了到数据库的连接：

```
thisConnection.Open();
```

如果 Open()方法未成功，比如未发现 SQL Server，就会抛出 SqlException 异常，您可以看到下面的消息：

```
Unhandled Exception: System.Data.SqlClient.SqlException: Timeout expired.
Possible reasons:  the timeout period elapsed prior to completion of the operation,
the server is not responding, or the maximum pool size was exceeded.  Please see
the documentation for further details.
   at System.Data.SqlClient.SqlConnection.Open()
   at DataReaderSqlExample.Main() in c:\begcsharp\chapter19\datareaderexample\
datareaderexample.cs:line 15
```

这段消息表示，在规定的合理时间范围内，程序不能连接到 SQL Server。检查在连接字符串中的服务器名称是否正确。

发出查询命令(SQL 命令)

下一步是创建命令对象，并给它提供 SQL 命令，执行数据库操作(比如检索数据)。其代码如下：

```
SqlCommand thisCommand = thisConnection.CreateCommand();
thisCommand.CommandText = "SELECT CustomerID, CompanyName from Customers";
```

连接对象具有 CreateCommand()方法，可以创建与此连接相关联的命令，因此我们使用它可以得到命令对象。命令本身被指派给命令对象的 CommandText 属性。我们可以从 Nrothwind 数据库中获取一列顾客 ID 和公司名称，使之成为 SQL 查询命令的基础：

```
SELECT CustomerID, CompanyName from Customers
```

SELECT 命令是可以从一个或多个表中获取数据的 SQL 命令。常见的错误是错误地拼写一个表的名称，导致另一个异常：

```
thisCommand.CommandText = "SELECT CustomerID, CompanyName from Customer";
```

唉！我忘记了在“Customers”中的“s”。因此就会抛出下面的异常：

```
Unhandled Exception: System.Data.SqlClient.SqlException: Invalid object name
'Customer'...
```

您可能注意到了，生成错误的方法是 ExecuteReader()；我们需要创建一个 DataReader 对象，并实际运行命令，以得到结果。接下来我们介绍其具体实现。

使用 DataReader 读取结果

DataReader 是轻量级的对象，可以迅速获取查询的结果。它是只读的，因此不能使用它更新数据 —— 在结束此范例之后，我们再介绍此内容。在上一节中我们看到，可以使用所创建的最后一个对象中方法、命令对象，来创建接下来所需要的对象的相关实例 —— 在此，即 DataReader：

```
SqlDataReader thisReader = thisCommand.ExecuteReader();
```

ExecuteReader()在数据库中运行 SQL 命令，因此也在此生成所有的数据库错误；它还创建读取器对象，用于读取生成的结果。将带有所包含的结果的该读取器传递给我们；我们将其指

派给 thisReader。

读取并获得方法

有几种方法可以从读取器中获取结果，但是下面介绍的过程确实最常见的过程。DataReader 的 Read()方法从查询中读取单独的一行数据，如果有多个数据要读取，则返回 true，如果没有，则返回 false。因此，我们建立 while 循环，使用 Read()方法读取数据，当迭代它们时，输出结果：

```
while (thisReader.Read())
{
    Console.WriteLine("\t{0}\t{1}",
    thisReader["CustomerID"], thisReader["CompanyName"]);
}
```

因此，当 Read()返回 true 时，Console.WriteLine("\t{0}\t{1}"写出两行带有由标签字符(\t)分隔的数据片断。OleDbDataReader 对象提供索引符属性(见第 11 章对索引符的讨论)。索引符允许我们引用列，作为按列名称的数组引用：thisReader["CustomerID"], thisReader["CompanyName"]。其输出结果是 CustomerID 和 CompanyName：

```
ALFKI    Alfreds Futterkiste
ANATR    Ana Trujillo Emparedados y helados
  ...
```

当 Read()在最后的结果处返回 false 时，while 循环就终结。

关闭读取器和连接

最后一步是关闭打开的对象，包括读取器对象和连接对象。每一个对象都有 Close()方法，在退出程序之前可以调用这些方法：

```
thisReader.Close();
thisConnection.Close();
```

这就是访问单一表的全部内容！

试试看：访问范例

稍作修改，就可以编写同样的程序，从而使用此数据库(nwind.mdb)的 Microsoft Access 版本。它位于目录 C:\Program Files\Microsoft Visual Studio.NET\Common7\Tools\Bin\，但是不要直接使用此程序，对其进行复制(在临时目录中，比如在 C:\tmp\nwind.mdb 中)，这样您总是可以返回到初始程序处。而且，此路径名对于 Visual Studio 的早期版本是正确的，但是现在必须作出修改才行，如果在指定的位置未发现此程序，可以使用系统上的 Search 菜单搜索。

除了连接字符串的细节外，所作的变动也可以用于其他 OLE DB 数据源。下面是一个范例(即从 Wrox Web 站点下载的代码中的 DataReaderAccessExample)：

```
using System;
using System.Data;          // Use ADO.NET namespace
using System.Data.OleDb;    // Use namespace for OLE DB .NET Data Provider
```

```
class DataReaderAccessExample
{
    public static void Main ()      // everything is in the Main function
    {
        // Create connection object for Microsoft Access OLE DB Provider;
        // note @ sign prefacing string literal so backslashes in path name;
        // work
        OleDbConnection thisConnection = new OleDbConnection(
              @"Provider=Microsoft.Jet.OLEDB.4.0;Data Source=C:\tmp\nwind.mdb");

        // Open connection object
        thisConnection.Open();

        // Create SQL command object on this connection


OleDbCommand thisCommand = thisConnection.CreateCommand();

        // Initialize SQL SELECT command to retrieve desired data
        thisCommand.CommandText =
              "SELECT CustomerID, CompanyName FROM Customers";

        // Create a DataReader object based on previously defined command object
        OleDbDataReader thisReader = thisCommand.ExecuteReader();

        while (thisReader.Read())
        {
            Console.WriteLine("\t{0}\t{1}",
            thisReader["CustomerID"], thisReader["CompanyName"]);
        }
        thisReader.Close();
        thisConnection.Close();
        }
    }
}
```

惟一变动的是连接、命令和读取器对象的名称以及连接字符串的内容。

而且，还把规定数据提供者变化的 using 指令从：

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

变为：

```
using System.Data.OleDb;
```

代码的说明

我们所创建的不是 SqlConnection，SqlCommand 和 SqlDataReader 对象，而是创建了 OleDbConnection，OleDbCommand 和 OleDbDataReader 对象。这些对象的作用基本上与 OLE DB 相应的对象的作用相同；其具体区别请参见 Visual Studio.NET 文档。

其他的惟一区别就是连接字符串，我们需要彻底改变。OLE DB 连接字符串的第一部分即 Provider 子句，该子句规定了用于这种类型的数据库的 OLE DB 提供者的名称。对于 Microsoft Access 数据库而言，总是采用下面的名称(“Jet”是包括在 Access 中的数据库引擎的名称)：

```
Provider=Microsoft.Jet.OLEDB.4.0;
```

如果为不同的数据库或数据格式使用了不同的 OLE DB 提供者，则可以在 Provider 子句中规定该提供者的名称。

连接字符串的第二部分是 Data Source 子句，在使用 OLE DB/Microsoft Access 的情况下，它简单地规定将要打开的 Microsoft Access 数据库文件(.mdb 文件)的名称：

```
Data Source=C:\tmp\nwind.mdb
```

注意程序中连接字符串之前的@符号，它规定了在 C#中的字符串字面量，这样路径名称中的反斜线才会发挥作用；否则，需要使用双反斜线(\\)才可以在 C#中对文件名称转义。

试试看：更新数据

现在我们已经可以从数据库中读取数据了，但是我们如何修改数据呢？我们将再次给出一个非常简单的范例，此范例仅使用了一个表，同时引入在第 20 章才会介绍的几个新对象。

我们可以设想一位顾客 Bottom-Dollar Markets，其名称改变为 Acme, Inc。我们需要在数据库中改变公司名称。我们在此还要使用 Northwind 数据库的 SQL Server/MSDE 版本。

此范例完整的源代码如下(即下载代码中的 DataUpdateExample)：

```
using System;
using System.Data;                  // Use ADO.NET namespace
using System.Data.SqlClient;   // Use SQL Server .NET Data Provider

class DataUpdateExample
{
    public static void Main()
    {
        // Specify SQL Server-specific connection string
        SqlConnection thisConnection = new SqlConnection(
              @"Data Source=(local);Integrated Security=SSPI;" +
              "Initial Catalog=northwind");

        // Open connection
        thisConnection.Open();
```

```
        // Create DataAdapter object for update and other operations
        SqlDataAdapter thisAdapter = new SqlDataAdapter(
            "SELECT CustomerID, CompanyName FROM Customers", thisConnection);

        // Create CommandBuilder object to build SQL commands
        SqlCommandBuilder thisBuilder = new SqlCommandBuilder(thisAdapter);

        // Create DataSet to contain related data tables, rows, and columns
        DataSet thisDataSet = new DataSet();

        // Fill DataSet using query defined previously for DataAdapter
        thisAdapter.Fill(thisDataSet, "Customers");

        // Show data before change
        Console.WriteLine("name before change: {0}",
            thisDataSet.Tables["Customers"].Rows[9]["CompanyName"]);

        // Change data in Customers table, row 9, CompanyName column
        thisDataSet.Tables["Customers"].Rows[9]["CompanyName"] = "Acme, Inc.";

        // Call Update command to mark change in table
        thisAdapter.Update(thisDataSet, "Customers");

        Console.WriteLine("name after change: {0}",
            thisDataSet.Tables["Customers"].Rows[9]["CompanyName"]);

        thisConnection.Close();
    }
}
```

当运行程序时，其输出结果如下：

```
name before change: Bottom-Dollar Markets
name after change: Acme, Inc.
```

下面我们具体研究源代码。

代码的说明

程序的第一部分类似于前面的范例；我们可以使用连接字符串创建连接对象并打开连接：

```
SqlConnection thisConnection = new SqlConnection(
    @"Data Source=(local);Integrated Security=SSPI;" +
    "Initial Catalog=northwind");
thisConnection.Open();
```

然后我们使用下面的语句创建一种新对象 —— a DataAdapter：

```
SqlDataAdapter thisAdapter = new SqlDataAdapter(
    "SELECT CustomerID, CompanyName FROM Customers", thisConnection);
```

DataAdapter 比 DataReader 对象更通用一些：它用于对数据进行的许多不同种类的操作，包括查询、更新和删除。我们可以使用 SQL 查询初始化它，在前面的范例中，我们使用了同样的 SQL 查询初始化命令对象。

CommandBuilder 对象

在更新单一表的简单情况下，我们不需要知道如何编写 SQL 语句，以完成更新；我们可以使用方便的简单对象 CommandBuilder，它可以为我们创建正确的 SQL 语句，并将其自动与 DataAdapter 相关联：

```
SqlCommandBuilder thisBuilder = new SqlCommandBuilder(thisAdapter);
```

注意，我们将 thisAdapter 作为参数传递给 CommandBuilder 构造函数。当创建 CommandBuilder 对象时，构造函数就生成 SQL 命令，并将其与传递进来的 DataAdapter 关联。稍后在本章我们还要介绍其他的 SQL 语句；但是现在我们只能学习 SQL。

DataSet 对象

下面我们介绍一个对象，随着深入学习，我们会逐渐认识到其重要性。该对象就是 DataSet 对象：

```
DataSet thisDataSet = new DataSet();
```

DataSet 是 ADO.NET 中的关键对象；所有复杂的操作都要使用它。DataSet 包含一组相关的 DataTable 对象，代表要使用的数据库表。每一个 DataTable 对象都有子 DataRow 和 DataColumn 对象，分别代表数据库表的行和列。通过这些对象，您可以获取表、行和列的所有元素，这些内容稍后介绍。

DataAdapter 填充方法

在使用 DataSet 之前，我们必须使用来自数据库的数据填充它。可以使用 DataAdapter 中的 Fill()方法来完成这种工作。为什么使用的是 DataAdapter 的 Fill()方法而不是 DataSet 呢？这是因为 DataSet 是内存中数据的抽象表示，而 DataAdapter 是一个对象，它将 DataSet 联系到某个数据库上。我们给 Fill()传递的第一个参数是希望用来填充对象的数据组，第二个参数是 DataSet 中的 DataTable 的名称，其中我们希望加载 DataSet 的数据：

```
thisAdapter.Fill(thisDataSet, "Customers");
```

在此，第二个参数的名称是 Customers 表，因此就使用此名称在 DataSet 中调用相关的 DataTable。填充了 DataSet 之后，我们就可以访问单独的行和列。

在 DataSet 中访问表、行和列

在变动数据之前，我们看一下改变之前的数据：

```
Console.WriteLine("name before change: {0}",
```

```
thisDataSet.Tables["Customers"].Rows[9]["CompanyName"]);Z
```

在此我们做了什么工作呢？我们在 Customers 表中输出了索引号为 9 的行的 CompanyName 列，如下所示：

```
name before change: Bottom-Dollar Markets
```

以 thisDataSet 开始的复杂表达式具有值“Bottom-Dollar Markets”。下面我们详细介绍其各个组成部分：

```
thisDataSet.Tables["Customers"].Rows[9]["CompanyName"]
```

DataSet 对象具有 Tables 属性，它包含一个全部 DataTable 对象的集合。此集合是一个有序的列表，意味着您可以以表名作为索引访问每一个 DataTable，就好像它们是数组中的一个元素一样：

```
Tables["Customers"]
```

在每一个 DataTable 中，有一个 Rows 属性，该属性是单独的 DataRow 对象的一个集合；它也是一个有序的列表，可以按行号索引。在此我们稍作简化：假定我们只希望知道索引号为 9 的行(实际上是第 10 行，因为索引号以 0 开始的 —— 第一行是 Rows[0]，行号为 0)：

```
Rows[9]
```

在实际的程序中，我们很可能还要在 SQL 查询中添加一些限定符，选择感兴趣的行，而不是仅仅使用行的索引号。在下面的范例中，我们就讨论如何找到感兴趣的行。

最后我们要介绍的是，DataRow 对象具有索引符属性，使得您可以通过名称访问单独的列。我们使用此索引符可以得到 Rows[9]: ["CompanyName"]的 CompanyName 列。

表、行和列的 DataSet 结构

上面所讨论的结构有点抽象，下面我们给出一张图表演示该结构，如图 19-4 所示。

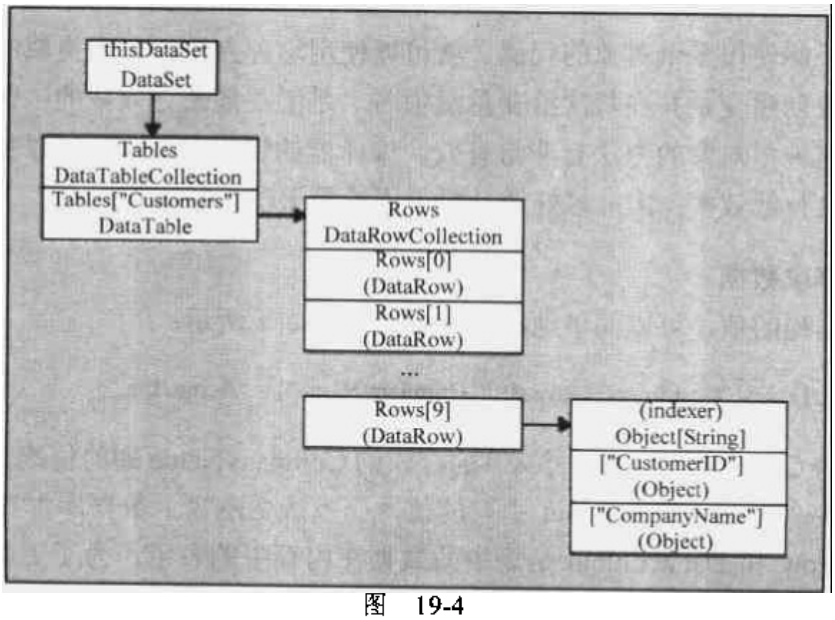


图　19-4

变量 thisDataSet 具有 DataSet 的类型，我们已经对其进行了声明。DataSet 具有类型 DataTableCollection 的属性 Tables；此集合的单独成员具有 DataTable 的类型；可以通过表名称访问它们，在此 Tables["Customers"]就是所讨论的成员。DataTable 具有类型 DataRowCollection 的属性 Rows；此集合的成员具有 DataRow 的类型，可以通过行号来访问。

您可能希望 DataRow 具有类型 DataColumnCollection 的属性，但是实际并非如此简单，因为我们希望根据各自的数据类型访问每一行中的单独列，因此包含字符数据的列就变为字符串，包含整型数据的列变为整型对象，如此等等。

为此，我们要使用 DataRow 的索引符属性，DataRow 是通过列名称(是字符串)索引的通用对象(System.Object)的数组。相反，为了使用索引符引用数据，您可以在 DataRow 对象上使用数组操作符。在此处的范例代码中，Rows[9]是 DataRow 对象，因此 Rows[9]["CompanyName"]是对代表行号为 9 的 CompanyName 列的对象的索引符引用。

理解这种结构的另一种方式是将这个范例分解之后，对表达式的各个组成元素逐个研究：

```
// Example using multiple objects
DataTable customerTable = thisDataSet.Tables["Customers"];
DataRow rowNine = customerTable.Rows[9];
Object companyName = rowNine["CompanyName"];
Console.WriteLine("name before change: {0}", companyName);
```

在此，我们将 customerTable 声明为 DataTable，并在 thisDataSet 的 Tables 属性中指定 Customers 表。将 rowNine 声明为 DataRow，并把 customerTable 的 Rows 属性的第 9 个元素赋给它。最后，我们将 companyName 声明为 Object，并使用 rowNine 的索引符属性给它指定 CompanyName 域。

当跟踪关联对象的链时，此范例可以帮助我们跟踪其进程，但是使用得出同样结果的单行表达式要更简单些：

```
Console.WriteLine("name before change: {0}",
    thisDataSet.Tables["Customers"].Rows[9]["CompanyName"]);
```

如果您十分了解使用多个对象的代码，就可以使用这种方法。对于类似的一次引用方法，它在为每一个对象创建变量并在每次给变量赋值时，是潜在地缺乏效率的；但是，如果准备多次使用对象，则这种多对象的方法有非常有效。编译器的优化器可以通过以每次引用为基础编程的方法而补偿这种低效率；因此最好还是要使用最易于理解的编码方式。

在数据集中修改数据

为了改变数据列的值，可以简单地给它赋值即可，如下所示：

```
thisDataSet.Tables["Customers"].Rows[9]["CompanyName"] = "Acme, Inc.";
```

这一行代码将 Customers 的索引号为 9 的行中的 CompanyName 列的值改变为“Acme, Inc.”。但是，它仅仅改变在内存中的 DataSet 中列的值，而不改变磁盘上数据库本身的值。DataSet，DataTable，DataRow 和 DataColumn 是表中的数据在内存中的表示。为了更新数据库，我们还需要调用 Update()方法。

DataAdapter 更新方法

Update()是 DataAdapter 对象的方法。为了调用它，可以规定以其为基础进行更新的 DataSet，以及在 DataSet 中用于更新的 DataTable 的名称。需要指出的是，当调用前面提到的 Fill()方法时，DataTable 名称(“Customers”)必须与我们所使用的名称相匹配：

```
thisAdapter.Update(thisDataSet, "Customers");
```

Update()方法自动遍历 DataTable 中的行，以检查需要对数据库作出的变动。在 Rows 集合中的每一个 DataRow 对象都具有属性 RowState，可以跟踪此行是否已经被删除、添加、修改，还是未作变动。所作的任何变化都会反映到此数据库中。

结束

现在通过输出数据的“更新后”的状态，我们可以确认所作的变动：

```
Console.WriteLine("name after change: {0}",
    thisDataSet.Tables["Customers"].Rows[9]["CompanyName"]);
```

最后，关闭连接：

```
thisConnection.Close();
```

这就是全部内容！

试试看：添加行

在前面的范例中我们更新了现有的行的值，但是如果我们希望添加新行，或删除现有的行，该怎么办呢？通过几个 DataRow 对象的方法就可以很容易地完成此任务。我们可以修改程序，在 Customers 表中创建新行；使用下面突出显示的代码行代替 Fill()和 Update()方法调用之间的代码(即下载代码中的 DataAddRowExample)：

```
thisAdapter.Fill(thisDataSet, "Customers");

Console.WriteLine("# rows before change: {0}",
                  thisDataSet.Tables["Customers"].Rows.Count);

DataRow thisRow = thisDataSet.Tables["Customers"].NewRow();
thisRow["CustomerID"] = "ZACZI";
thisRow["CompanyName"] = "Zachary Zithers Ltd.";
thisDataSet.Tables["Customers"].Rows.Add(thisRow);

Console.WriteLine("# rows after change: {0}",
                  thisDataSet.Tables["Customers"].Rows.Count);

thisAdapter.Update(thisDataSet, "Customers");
```

此范例的输出结果为：

```
# rows before change: 91
# rows after change: 92
```

下面我们详细研究这些代码。

代码的说明

首先看一下引入 Rows 集合的新属性(Count)之前的情况。这使得我们可以计算在此表中有多少行：

```
Console.WriteLine("# rows before change: {0}",
    thisDataSet.Tables["Customers"].Rows.Count);
```

其次，我们可以通过使用 DataTable 对象的 NewRow()方法创建新行对象：

```
DataRow thisRow = thisDataSet.Tables["Customers"].NewRow();
```

注意这就使用与 Customers 表相同的列创建了一个新行对象，但是并不将其实际添加到 DataSet 中；我们需要在完成之前给列赋一些值：

```
thisRow["CustomerID"] = "ZACZI";
thisRow["CompanyName"] = "Zachary Zithers Ltd.";
```

现在我们可以使用 Rows 集合的 Add()方法实际添加行：

```
thisDataSet.Tables["Customers"].Rows.Add(thisRow);
```

如果在调用 Add()方法之后选取了 Count 属性，就可以看到已经实际添加了一行：

```
Console.WriteLine("# rows after change: {0}",
    thisDataSet.Tables["Customers"].Rows.Count);
```

输出结果显示了 92 行，比变动之前的输出结果多出了一行。对于前面的范例，还需要调用 Update()方法才能在磁盘上的数据库中实际添加新行：

```
thisAdapter.Update (thisDataSet, "Customers");
```

记住，DataSet 是内存中，非连接的数据副本；它实际上是 DataAdapter 连接到磁盘上的数据库，因此需要 DataAdapter 的 Update()方法才能使 DataSet 中的内存数据与磁盘上的数据库同步。

如果我们在运行了此程序之后，看一下数据库中的表，就会发现已经成功地添加了一行，如图 19-5 所示。

CustomerID	CompanyName	ContactName	ContactTitle	Address	City
WARTH	Wartian Herkku	Pirkko Koskitalo	Accounting Manager	Torikatu 38	Oulu
WELLI	Wellington Importadora	Paula Parente	Sales Manager	Rua do Mercado, 12	Resende
WHITC	White Clover Markets	Karl Jablonski	Owner	305 - 14th Ave. S. Suite 3B	Seattle
WILMK	Wilman Kala	Matti Karttunen	Owner/Marketing Assistant	Keskuskatu 45	Helsinki
WOLZA	Wolski Zajazd	Zbyszek Piestrzeniewicz	Owner	ul. Filtrowa 68	Warszawa
ZACZI	Zachary Zithers Ltd.	<NULL>	<NULL>	<NULL>	<NULL>

图　19-5

注意，仅仅填充了 Customer ID 和 Company Name 列，因为我们在程序中所使用的就是这两个列。其余的列都是空白的(实际上，根据 SQL 它们也包含值 NULL)。您可能认为填充 Contact Name 仅仅是在程序代码中添加下面的这一行代码的问题：

```
thisRow["ContactName"] = "Zylla Zithers";
```

但是，这并不是您所要完成的所有工作。在前面，当我们进行首次查询时，我们构建了 DataSet，规定了两行：CustomerID 和 CompanyName：

```
SqlDataAdapter thisAdapter = new SqlDataAdapter(
  "SELECT CustomerID, CompanyName FROM Customers", thisConnection);
```

对 ContactName 的引用会导致错误，因为在所构建的 DataSet 中并没有这样的列。可以通过添加 ContactName 列进行修正：

```
SqlDataAdapter thisAdapter = new SqlDataAdapter(
    "SELECT CustomerID, ContactName, CompanyName FROM Customers",
    thisConnection);
```

或者可以使用下面的命令从 Customers 中选择所有的列：

```
SqlDataAdapter thisAdapter = new SqlDataAdapter(
    "SELECT * FROM Customers", thisConnection);
```

在第 18 章我们谈到，在 SQL SELECT 命令中的星号(*)代表了表中的所有列；根据这种变动，您可以为数据库中的所有列添加值。但是，当仅仅需要使用两列或三列时，却得到了所有的列，这有点浪费；这种问题要注意避免。

试试看：找到行

如果您试图多次运行前面的范例，就可以看到下面的消息：

```
Unhandled Exception: System.Data.SqlClient.SqlException:
Violation of PRIMARY KEY constraint 'PK_Customers'.
Cannot insert duplicate key in object 'Customers'.
```

这表示 Add()失败了，因为它原本应该创建复制行。Customers 表的定义要求 CustomerID 域包含惟一值，当向列指派主键码时就要求这样。当我们试图第二次运行代码时，就会给出值“ZACZI”，因为它已经被放置在第一次运行的范例的表中。

我们可以改变其逻辑，这样我们就可以在添加行之前首先对其进行搜索。DataTable Rows 集合提供了称为 Find()的方法，此方法在完成这种任务时非常有用；我们可以改写围绕行加法的逻辑，以便使用 Find()，而不是计算行。下面是完整的源代码(在下载的代码中的 DataAddFindExample)；在给出代码之后我们会详细对其进行讨论：

```
using System;
using System.Data;                  // use ADO.NET namespace
using System.Data.SqlClient;        // use SQL Server .NET Data Provider
```

```
class DataAddFindExample
{
    public static void Main()
    {
        // Specify SQL Server-specific connection string
        SqlConnection thisConnection = new SqlConnection(
            @"Data Source=(local);Integrated Security=SSPI;" +
            "Initial Catalog=northwind");

        // Open connection
        thisConnection.Open();

        // Create DataAdapter object for update and other operations
        SqlDataAdapter thisAdapter = new SqlDataAdapter(
            "SELECT CustomerID, CompanyName FROM Customers", thisConnection);

        // Create CommandBuilder object to build SQL commands
        SqlCommandBuilder thisBuilder = new SqlCommandBuilder(thisAdapter);

        // Create DataSet to contain related data tables, rows, and columns
        DataSet thisDataSet = new DataSet();

        // Fill DataSet using query defined previously for DataAdapter
         thisAdapter.Fill(thisDataSet, "Customers");

        Console.WriteLine("# rows before change: {0}",
            thisDataSet.Tables["Customers"].Rows.Count);

        // Set up keys object for defining primary key
        DataColumn[] keys = new DataColumn[1];
        keys[0] = thisDataSet.Tables["Customers"].Columns["CustomerID"];
        thisDataSet.Tables["Customers"].PrimaryKey = keys;

        DataRow findRow = thisDataSet.Tables["Customers"].Rows.Find("ZACZI");

        if (findRow == null)
        {
            Console.WriteLine("ZACZI not found, will add to Customers table");

            DataRow thisRow = thisDataSet.Tables["Customers"].NewRow();
            thisRow["CustomerID"] = "ZACZI";
            thisRow["CompanyName"] = "Zachary Zithers Ltd.";
            thisDataSet.Tables["Customers"].Rows.Add(thisRow);
            if ((findRow =
```

```
                thisDataSet.Tables["Customers"].Rows.Find("ZACZI")) != null)
            {
                Console.WriteLine("ZACZI successfully added to Customers table");
            }
        }
        else
        {
            Console.WriteLine("ZACZI already present in database");
        }

        thisAdapter.Update(thisDataSet, "Customers");

        Console.WriteLine("# rows after change: {0}",
        thisDataSet.Tables["Customers"].Rows.Count);
        thisConnection.Close();
    }
}
```

代码的说明

程序从开始直到 Fill()方法调用都与前面的范例的程序相同。我们使用 Count 属性输出当前行数，然后使用 Find()检查我们所添加的行是否已经存在。

在使用 Find()之前，我们需要构建主键码。主键码就是用于搜索的键码；它由表的一列或多列构成，包含可以在表中惟一地标识行的值或值集合，因此当我们通过键码搜索时，可以找到一行，而且仅仅可以找到一行。在 Northwind 数据库中的 Customers 表将 CustomerID 列用作其主键码：

```
DataColumn[] keys = new DataColumn[1];
keys[0] = thisDataSet.Tables["Customers"].Columns["CustomerID"];
thisDataSet.Tables["Customers"].PrimaryKey = keys;
```

首先我们创建 DataColumns 数组。因为键码就是一列或多列，所以很自然地要使用数组这种结构；我们称 DataColumn 数组为 Keys。接下来我们将键码数组 keys[0]的第一个元素赋给 Customers 表中的 CustomerID 列的值。最后，我们将 Keys 赋给 Customers DataTable 对象的 PrimaryKey 属性。

注意：

另一种办法是从数据库中加载主键码，这不会默认完成。但是，您可以显式地告诉 ADO.NET 加载主键码的信息，方法是在填充 DataSet 之前设置 DataAdapter MissingSchemaAction 属性，如下所示：

```
thisAdapter.MissingSchemaAction = MissingSchemaAction.AddWithKey;
thisAdapter.Fill(thisDataSet, "Customers");
```

通过隐式初始化 DataTable 的 PromaryKey 属性，同样可以完成主键码的构建。

不管使用哪种方法，我们都可以找到行！

```
DataRow findRow = thisDataSet.Tables["Customers"].Rows.Find("ZACZI");
DataRow findRow = thisDataSet.Tables["Customers"].Rows.Find("ZACZI");
```

Find()返回一个 DataRow，因此我们建立名称为 findRow 的 DataRow 对象就可以获取结果。Find()带有的参数是要查询的值；这可以是多值键码的对象数组，但是在此它仅有一个值，我们可以传递一个包含值“ZACZI”的字符串，它是我们要查询的 CustomerID。

如果 Find()定位了一个匹配行，则它就返回匹配该行的 DataRow；如果未发现匹配行，则返回 null 引用，下面就可以看到这样的引用：

```
if (findRow == null) {
    Console.WriteLine("ZACZI not found, will add to Customers table");

    DataRow thisRow = thisDataSet.Tables["Customers"].NewRow();
    thisRow["CustomerID"] = "ZACZI";
    thisRow["CompanyName"] = "Zachary Zithers Ltd.";
    thisDataSet.Tables["Customers"].Rows.Add(thisRow);
    if ((findRow = thisDataSet.Tables["Customers"].Rows.Find("ZACZI")) != null)
    {
        Console.WriteLine("ZACZI successfully added to Customers table");
    }
}
else
{
    Console.WriteLine("ZACZI already present in database");
}
```

如果 findRow 是 null，我们就可以继续向下进行，添加行，这同前面范例相同。为了保证 Add()成功，我们必须在知道添加操作成功后马上调用 Find()方法。

在本节前面我们已经提到，使用 Find()方法的这个版本可以重用；您可以正确地多次运行它。但是，一旦在数据库中找到“ZACZI”行之后，它就不再运行 Add()代码。下面我们学习如何重复这一部分程序。

试试看：删除行

您猜想到我要介绍这部分内容了吗？添加了行之后，从逻辑上来讲，我们就应该进行相反的操作，即删除行。

DataRow 对象具有可以删除当前行的 Delete()方法。我们的新代码改变了在 findRow 上的 if 语句的作用，这样就可以测试 findRow 不等于 null(换句话来说，要找到所搜索的行)。然后我们就可以删除行，为此要在 findRow 上调用 Delete(即在下载的代码中的 DataDeleteExample)：

```
using System;
using System.Data;                  // use ADO.NET namespace
using System.Data.SqlClient;        // use SQL Server .NET Data Provider
```

```
class DataDeleteExample
{
    public static void Main()
    {
        // Specify SQL Server-specific connection string
        SqlConnection thisConnection = new SqlConnection(
            @"Data Source=(local);Integrated Security=SSPI;" +
            "Initial Catalog=northwind");

        // Open connection
        thisConnection.Open();

        // Create DataAdapter object for update and other operations
        SqlDataAdapter thisAdapter = new SqlDataAdapter(
            "SELECT CustomerID, CompanyName FROM Customers", thisConnection);

        // Create CommandBuilder object to build SQL commands
        SqlCommandBuilder thisBuilder = new SqlCommandBuilder(thisAdapter);

        // Create DataSet to contain related data tables, rows, and columns
        DataSet thisDataSet = new DataSet();

        // Fill DataSet using query defined previously for DataAdapter
        thisAdapter.Fill(thisDataSet, "Customers");

        Console.WriteLine("# rows before change: {0}",
            thisDataSet.Tables["Customers"].Rows.Count);

        // Set up keys object for defining primary key
        DataColumn[] keys = new DataColumn[1];
        keys[0] = thisDataSet.Tables["Customers"].Columns["CustomerID"];
        thisDataSet.Tables["Customers"].PrimaryKey = keys;

        DataRow findRow = thisDataSet.Tables["Customers"].Rows.Find("ZACZI");

        if (findRow != null)
        {
            Console.WriteLine("ZACZI already in Customers table");
            Console.WriteLine("Removing ZACZI  . . .");

            findRow.Delete();

            thisAdapter.Update(thisDataSet, "Customers");
        }
```

```
            Console.WriteLine("# rows after change: {0}",
                thisDataSet.Tables["Customers"].Rows.Count);

            thisConnection.Close();
        }
    }
```

代码的说明

注意当调用 Delete()时，在调用 Update 改变数据库之前，数据库中的行是不会被删除的。

注意：

实际上 Delete()方法并不执行删除操作，它仅仅标记要删除的行。在 Rows 集合中的每一个 DataRow 对象都有属性 RowState，它跟踪记录此行是被删除、添加、修改，还是不改变。Delete()方法将行的 RowState 设置为 Deleted，然后 Update()就在数据库中将在 Rows 集合中标记为 Deleted 的所有行删除。

Visual Studio.NET 在线文档会提示您在执行了 Delete()方法之后调用 DataSet 的 AcceptChanges()方法；但是，这仅能删除 DataSet 中的行，却不影响在数据库中的行，因为 Update()仅影响在 Rows 集合中找到的行，遗漏的行就被忽略掉。如果希望在数据库中删除行，则不要在调用 Update()之前调用 AcceptChanges()方法。

这个问题也适用于 Remove()方法；仅当希望从 DataSet 的 Rows 集合中，而不是从数据库中删除行时才调用它。

19.2.4　在 DataSet 中访问多个表

ADO.NET 模型与原来的数据访问模型相比，有一个最大优点，就是 DataSet 对象可以记录多个表和它们之间的关系。这表示您可以在一个操作的不同程序段之间传递完整的相关数据集，体系结构内在地维护数据之间关系的完整性。

DataRelation 对象用于描述在 DataSet 中的多个 DataTables 之间的关系。每一个 DataSet 都有 DataRelations 的 Relations 集合，这使得您可以确认和操纵相关的表。

首先我们研究 Customers 和 Orders 表。每一位顾客可能发出几个订单；我们如何才能看到由每一位顾客发出的订单呢？Orders 表中的每一行都包含发出订单的顾客的 CustomerID；您可以匹配所有的订单行，这些行包含了一个在 Customers 表中该顾客行的具体 CustomerID，如图 19-6 所示。

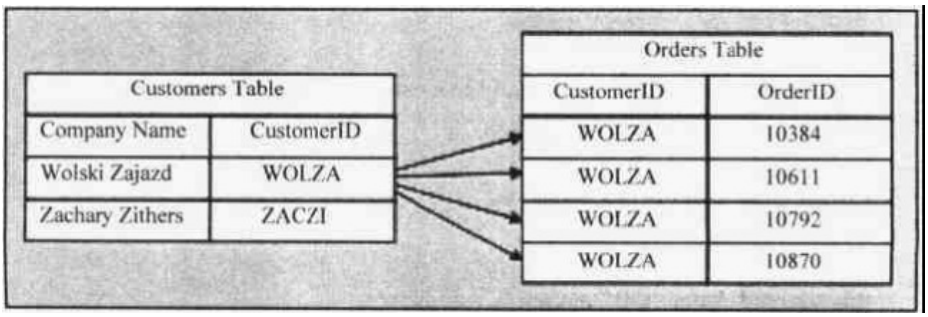


图　19-6

在两个表中的匹配的 CustomerID 域定义 Customers 表和 Orders 表之间的一对多的关系。我们可以在 ADO.NET 中使用这种关系，为此要创建一个代表这种关系的 DataRelation 对象。

1. 导航关系

为了使用这种关系，我们需要从一张表的行进入到另一张表的关联行。这就是对关系导航。通常导航是指从第一张表的父行进入其他表的子行。在图 19-6 中，在 Customers 表中的行可以被认为是父行，在 Orders 表中的相关行被认为是子行。导航也可以按相反的方向进行。我们再回到前面，此时我们准备在代码中使用导航这个概念。首先建立程序。

试试看：DataRelation 范例

下面是一段完整的程序(DataRelationExample)：在学习了这段完整的程序之后，我们要具体讨论这段程序：

```
using System;
using System.Data;                  // use ADO.NET namespace
using System.Data.SqlClient;    // use OLE DB .NET Data Provider

class DataRelationExample
{
    public static void Main()
    {
        // Specify SQL Server-specific connection string
        SqlConnection thisConnection = new SqlConnection(
            @"Data Source=(local);Integrated Security=SSPI;" +
            "Initial Catalog=northwind");

        // Open connection
        thisConnection.Open();

        // Create DataAdapter object for update and other operations
        SqlDataAdapter thisAdapter = new SqlDataAdapter(
            "SELECT CustomerID, CompanyName FROM Customers", thisConnection);

        // Create CommandBuilder object to build SQL commands
        SqlCommandBuilder thisBuilder = new SqlCommandBuilder(thisAdapter);

        // Create DataSet to contain related data tables, rows, and columns
        DataSet thisDataSet = new DataSet();

        // Set up DataAdapter objects for each table and fill
        SqlDataAdapter custAdapter = new SqlDataAdapter(
            "SELECT * FROM Customers", thisConnection);
        SqlDataAdapter orderAdapter = new SqlDataAdapter(
            "SELECT * FROM Orders", thisConnection);
```

```
            custAdapter.Fill(thisDataSet, "Customers");
            orderAdapter.Fill(thisDataSet, "Orders");

            // Set up DataRelation between customers and orders
            DataRelation custOrderRel = thisDataSet.Relations.Add("CustOrders",
                thisDataSet.Tables["Customers"].Columns["CustomerID"],
                thisDataSet.Tables["Orders"].Columns["CustomerID"]);

            // Print out nested customers and their order ids
            foreach (DataRow custRow in thisDataSet.Tables["Customers"].Rows)
            {
                Console.WriteLine("Customer ID: " + custRow["CustomerID"] +
                                  " Name: " + custRow["CompanyName"]);
                foreach (DataRow orderRow in custRow.GetChildRows(custOrderRel))
                {
                    Console.WriteLine("    Order ID: " + orderRow["OrderID"]);
                }
            }
            thisConnection.Close();
        }
    }
```

代码的说明

在构造 DataRelation 之前，我们需要创建 DataSet 对象，并链接我们希望通过 DataSet 使用的数据库表，如下所示：

```
DataSet thisDataSet = new DataSet();
SqlDataAdapter custAdapter = new SqlDataAdapter(
    "SELECT * FROM Customers", thisConnection);
SqlDataAdapter orderAdapter = new SqlDataAdapter(
    "SELECT * FROM Orders", thisConnection);
custAdapter.Fill(thisDataSet, "Customers");
orderAdapter.Fill(thisDataSet, "Orders");
```

我们要为引用的每一张表创建一个 DataAdapter 对象。然后使用来自我们准备使用的列的数据填充 DataSet；在此我们不必担心效率问题，所以我们要使用所有的列(SELECT * FROM <table>)。

创建 DataRelation 对象

接下来创建 DataRelation 对象并将其链接到 DataSet 上：

```
DataRelation custOrderRel = thisDataSet.Relations.Add("CustOrders",
    thisDataSet.Tables["Customers"].Columns["CustomerID"],
    thisDataSet.Tables["Orders"].Columns["CustomerID"]);
```

DataTable 具有 Relations 属性，它是在 DataSet 中表示表之间关系的所有 DataRelation 对象

的集合。

为了创建新 DataRelation，我们可以使用 Relations 的 Add()方法，它为关系接收字符串名称(在此范例中即“CustOrders”)，和两个 DataColumns，以及来自 Customers 的父列 CustomerID，Customers 后面跟有来自 Orders 的子列 CustomerID。父列必须位于子列之前。这就是其全部内容。

使用 GetChildRows

现在我们可以找到顾客及其订单。首先，建立 foreach 循环，显示每一位顾客的顾客信息：

```
foreach (DataRow custRow in thisDataSet.Tables["Customers"].Rows)
{
    Console.WriteLine("Customer ID: " + custRow["CustomerID"] +
                      " Name: " + custRow["CompanyName"]);
```

我们刚刚循环了 Customers 表的 Rows 集合，输出了每一位顾客的 CustomerID 和 CompanyName。显示了顾客信息之后，我们就希望显示该顾客的相关订单。

为此，我们需要添加一个嵌套的 foreach 循环，通过调用 DataRow 的 GetChildRows()方法对其进行初始化。我们将所创建的 DataRelation 对象传递给 GetChildRows()，这就会返回 DataRowCollection，它仅仅包含此顾客在 Orders 表中的相关行：

```
    foreach (DataRow orderRow in custRow.GetChildRows(custOrderRel))
    {
        Console.WriteLine("    Order ID: " + orderRow["OrderID"]);
    }
}
```

现在我们就可以为每一位顾客重复执行此进程。我们在 OrderID 的显示中添加了一些前导空白，因此每位顾客的订单的显示位置比顾客信息缩进一些：

```
...
Customer ID: WILMK Name: Wilman Kala
  Order ID: 10615
  Order ID: 10673
Customer ID: WOLZA Name: Wolski Zajazd
  Order ID: 10374
  Order ID: 10611
  Order ID: 10792
Customer ID: ZACZI Name: Zachary Zithers Ltd.
```

有了这种缩进显示图，您就会可以发现每一位顾客和他们所发出的订单之间的关系更明显了。Customer ID “Zachary Zithers Ltd.” 没有 Orders，因为我们刚刚在前面的范例中将它添加到表中了。

试试看：更多的表

下面我们进一步扩展此程序，研究每一位顾客在订单中所定购的具体产品，以及具体产品的名称。这些信息可以通过 Northwind 数据库中的其他表获得。我们回顾一下它们之间的关系；了解它们之间关系的最简单的办法是为 Northwind 数据库创建一个图表，Northwind 数据库在 VS 中包含的所有的表，这同第 18 章的作法一样，如图 19-7 所示。

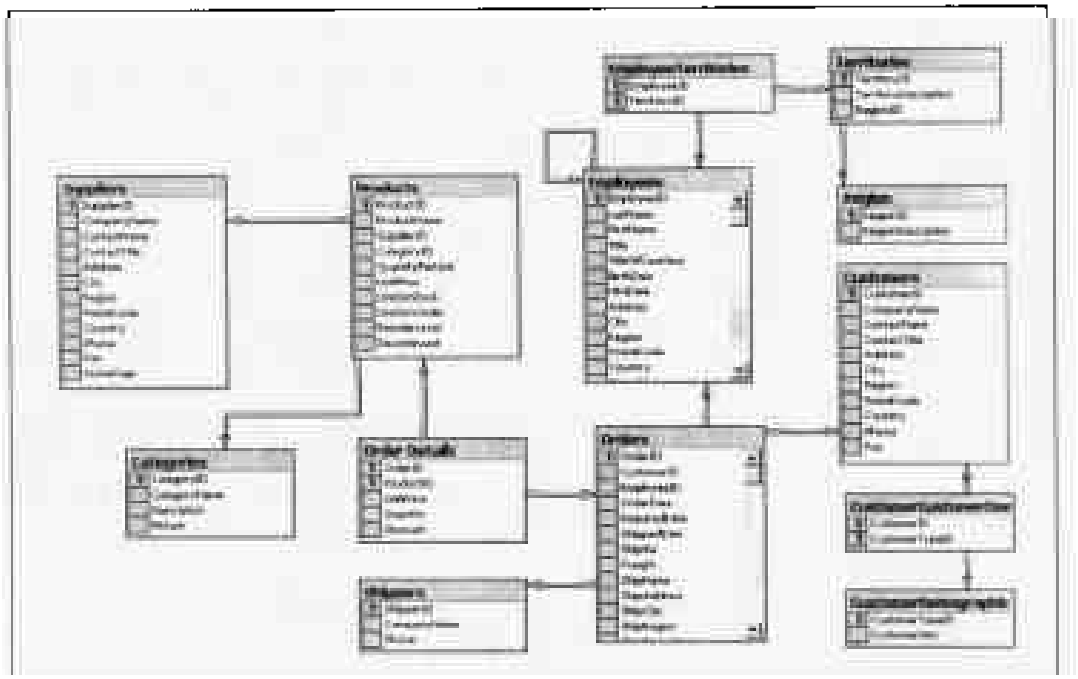


图　19-7

如果使用的是 MSDE，而且未输入关系，则可以参考此处的图表。表之间的线表示关系，指向列的每一端的线以标识关系。主键码和外键码之间的关系通过下面的方式表示，即箭头符号代表父列，无穷大符号代表子列。

我们希望能显示每一位包括产品名称的顾客订单细节，为此要遵循图 19-7 中 4 个表之间的关系：Customers，Orders，Order Details 和 Products 表。程序首先初始化连接，然后为每一个表创建一个 DataAdapter 对象：

```
DataSet thisDataSet = new DataSet();

SqlDataAdapter custAdapter = new SqlDataAdapter(
      "SELECT * FROM Customers", thisConnection);
custAdapter.Fill(thisDataSet, "Customers");

SqlDataAdapter orderAdapter = new SqlDataAdapter(
      "SELECT * FROM Orders", thisConnection);
orderAdapter.Fill(thisDataSet, "Orders");

SqlDataAdapter detailAdapter = new SqlDataAdapter(
      "SELECT * FROM [Order Details]", thisConnection);
```

```
detailAdapter.Fill(thisDataSet, "Order Details");

SqlDataAdapter prodAdapter = new SqlDataAdapter(
    "SELECT * FROM Products", thisConnection);
prodAdapter.Fill(thisDataSet, "Products");
```

接下来，为 4 个表之间的每一种关系构建 DataRelation 对象：

```
DataRelation custOrderRel = thisDataSet.Relations.Add("CustOrders",
    thisDataSet.Tables["Customers"].Columns["CustomerID"],
    thisDataSet.Tables["Orders"].Columns["CustomerID"]);

DataRelation orderDetailRel = thisDataSet.Relations.Add("OrderDetail",
    thisDataSet.Tables["Orders"].Columns["OrderID"],
    thisDataSet.Tables["Order Details"].Columns["OrderID"]);
```

第一种关系与前面范例的关系完全相同。第二种关系增加了 Orders 和 Order Details 之间的关系，为此使用 OrderID 作为链接列。

最后一种关系是 Order Details 和 Products 之间的关系，使用 ProductID 作为链接列。注意 Products 是父表(第二个参数)，因为它是一对多关系中的“一”端(一个 Product 可以出现在多个 Orders 中)：

```
DataRelation orderProductRel = thisDataSet.Relations.Add("OrderProducts",
    thisDataSet.Tables["Products"].Columns["ProductID"],
    thisDataSet.Tables["Order Details"].Columns["ProductID"]);
```

现在，我们已经建立了关系，接下来就可以使用这些关系执行处理工作。其基本结构还是一个嵌套的 foreach 循环，但是这一次具有三级嵌套：

```
foreach (DataRow custRow in thisDataSet.Tables["Customers"].Rows)
{
    Console.WriteLine("Customer ID: " + custRow["CustomerID"]);

    foreach (DataRow orderRow in custRow.GetChildRows(custOrderRel))
    {
        Console.WriteLine("\tOrder ID: " + orderRow["OrderID"]);
        Console.WriteLine("\t\tOrder Date: " + orderRow["OrderDate"]);

        foreach (DataRow detailRow in orderRow.GetChildRows(orderDetailRel))
        {
            Console.WriteLine("\t\tProduct: " +
                detailRow.GetParentRow(orderProductRel)["ProductName"]);
            Console.WriteLine("\t\tQuantity: " + detailRow["Quantity"]);
        }
    }
}
```

同前面一样，我们为父行输出数据，然后使用 GetChildRows()获取与此父行关联的子行。外循环与前面范例中的循环相同。接下来，将 OrderDate 附加细节输出到 OrderID，然后得到此 OrderID 的 OrderDetails。

最内层的循环却不同；为了获取 Product 行，我们可以调用 GetParentRow()，它可以获取父对象，从关系的“多”端进入“一”端。有时这种从子对象到父对象的导航也被称为“逆流”导航，相比之下，正常的父对象到子对象的导航称为“顺流”导航。逆流导航需要调用 GetParentRow()。

这样就可以结束程序，只是还要关闭连接，这同前面的范例相同。当编译和运行此程序时，我们可以看到下面的结果(下面的内容被简缩，但是显示了出现在屏幕的输出的最后一部分内容)：

```
Customer ID: WOLZA
    Order ID: 10906
        Order Date: 3/27/1996 12:00:00 AM
        Product: Sirop d'érable
        Quantity: 15
    Order ID: 10998
        Order Date: 5/3/1996 12:00:00 AM
        Product: Guaraná Fantástica
        Quantity: 12
        Product: Sirop d'érable
        Quantity: 7
        Product: Longlife Tofu
        Quantity: 20
        Product: Rhönbräu Klosterbier
        Quantity: 30
    Order ID: 11044
        Order Date: 5/23/1996 12:00:00 AM
        Product: Tarte au sucre
        Quantity: 12
Customer ID: ZACZI
```

我们可以看到为每一位顾客处理的订单的细节，其缩进结构表示父与子的分层结构。在此，Customer ID“ZACZI”还是没有订单，因为我们刚刚在前面的范例中向表中添加了订单。

2. ChildRelations 和 ParentRelations 集合

DataTable 具有属性 ChildRelations，它保存所有 DataRelations 的集合，当前的表以父关系的身份加入到 DataRelations 中。您可以使用此属性，处理表中的所有关系：

```
foreach(DataRelation thisRelation in thisTable.ChildRelations){
    foreach(DataRow thisParentRow in thisTable.Rows){
        foreach (DataRow thisChildRow in
                thisParentRow.GetChildRows(thisRelation))
        {
```

```
            // . . . process child rows . . .
        }
    }
}
```

在 DataTable 上还有 ParentRelations 集合，可以按类似的方式使用 DataTable。在目前的范例中不需要真正使用它，因为不能一次为一个表处理多个父关系或子关系。

19.3　XML 和 ADO.NET

在前面说过，XML 支持是 ADO.NET 的一个主要设计目标，它对于 ADO.NET 内部实现是至关重要的。因此，ADO.NET 具有对内置在对象模型中的 XML 强大的支持意义重大。因为在第 18 章中引入了 XML，所以现在我们介绍在 ADO.NET 中对其的支持。

19.4　在 ADO.NET DataSets 中的 XML 支持

在 ADO.NET 中的 XML 支持主要用于 DataSet 对象，因为 XML 主要关注所有的关系和分层的结构化数据。DataSet 具有 7 个方法可以用于处理 XML，其中最易于使用的方法是 WriteXml()，它将数据集的内容以 XML 文档的形式写出。

为了使用 WriteXml()，可以在现有的数据中构造一个 DataSet，为此要使用与上一个范例中相同的代码，并使用带有 Fill()方法的 DataAdapter 加载数据，为关系定义 DataRelation 对象等。然后在所构造的 DataSet 上调用 WriteXml()即可：

```
thisDataSet.WriteXml("nwinddata.xml");
```

WriteXml()可以将 XML 写到不同的目标上；该方法的此版本仅仅可以将 XML 写到文件中。将 XML 作为输入格式接收的外部程序可以很容易地读取和处理 XML。

ReadXml()方法也可以用于将 XML 文件的内容读取到 DataSet 中。

试试看：XML 范例

下面的程序可以构造 DataSet 并写出其内容；这类似于在上一个范例中使用的程序，但是所不同的是使用上面所述的对 WriteXml()的单一调用代替了复杂的嵌套的 foreach 循环。在运行此程序之前，要保证已经具有了名为 C:\tmp 的目录。

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

class DataXmlExample
{
    public static void Main()
```

```
    {
        SqlConnection thisConnection = new SqlConnection(
            @"Data Source=(local);Integrated Security=SSPI;" +
            "Initial Catalog=northwind");

        thisConnection.Open();
        DataSet thisDataSet = new DataSet();

        SqlDataAdapter custAdapter = new SqlDataAdapter(
            "SELECT * FROM Customers", thisConnection);
        custAdapter.Fill(thisDataSet, "Customers");

        SqlDataAdapter orderAdapter = new SqlDataAdapter(
            "SELECT * FROM Orders", thisConnection);
        orderAdapter.Fill(thisDataSet, "Orders");

        SqlDataAdapter detailAdapter = new SqlDataAdapter(
            "SELECT * FROM [Order Details]", thisConnection);
        detailAdapter.Fill(thisDataSet, "Order Details");

        DataRelation custOrderRel = thisDataSet.Relations.Add("CustOrders",
            thisDataSet.Tables["Customers"].Columns["CustomerID"],
            thisDataSet.Tables["Orders"].Columns["CustomerID"]);
        custOrderRel.Nested = true;

        DataRelation orderDetailRel = thisDataSet.Relations.Add("OrderDetail",
            thisDataSet.Tables["Orders"].Columns["OrderID"],
            thisDataSet.Tables["Order Details"].Columns["OrderID"]);
        orderDetailRel.Nested = true;

        thisDataSet.WriteXml(@"c:\tmp\nwinddata.xml");
        Console.WriteLine(
            @"Successfully wrote XML output to file c:\tmp\nwinddata.xml");

        thisConnection.Close();
    }
}
```

代码的说明

DataRelation 对象的 Nested 属性通知 WriteXml()方法，将订单细节和订单嵌套在 XML 输出中的每个父顾客之下。文件 nwinddata.xml 包含数据库中的所有数据(包括所有的列，因为当填充 DataSet 时，我们规定为 SELECT * FROM)。这是人类易于理解、易于分析的 XML 格式。您可以在 Microsoft Internet Explorer 中直接浏览其内容，如图 19-8 所示。

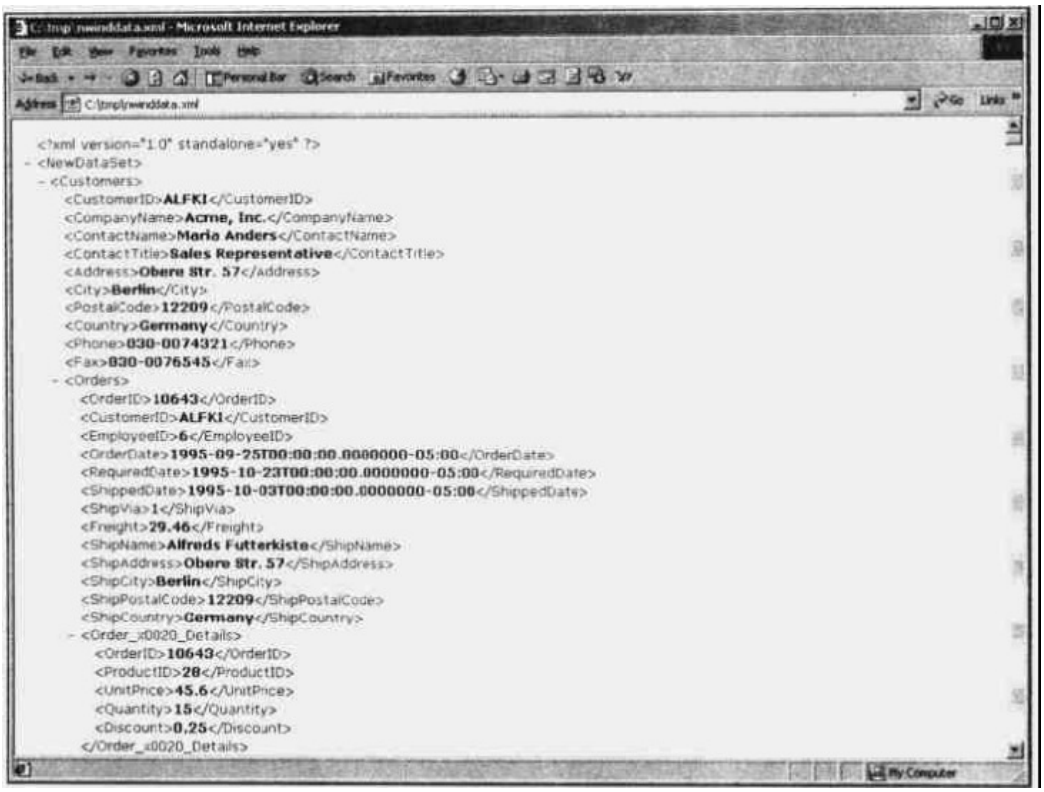


图　19-8

19.5　ADO.NET 中的 SQL 支持

在本章，我们已经介绍了基本的 ADO.NET 操作的全部内容，但是却没有介绍 SQL 数据库查询语言的内容。从数据源读取和写入的所有 ADO.NET 命令都被转换为执行初始数据库操作的 SQL 命令。

在现实的使用环境中，实际使用 ADO.NET 还需要具备 SQL 的知识；如需要更详细地了解 SQL，可以参考关于 SQL 的教材《Beginning SQL Programming》或《Beginning SQL Server 2000 Programming》。

下面我们就介绍关于 SQL 的基本知识。

19.5.1　在 DataAdapter 中的 SQL 命令

在前面给出的范例中，我们曾经使用 SQL SELECT 命令，它返回表的所有行，例如：

```
SqlDataAdapter thisAdapter = new SqlDataAdapter(
    "SELECT * FROM Customers", thisConnection);
```

当调用 Fill()方法时，这个 SELECT 命令返回顾客表的所有行和列，并把它们加载到程序的内存中。这非常适合于小型表，比如 Northwind Customers，它仅有 11 列，少于 100 行的数据，但是对于那些有 100 000，甚至是 1 000 000 行的商业应用程序中的大型表而言，这就不适用了。

您需要构造 SELECT 命令，这样就可以给出确实需要处理的数据。其中一种方法是，如果

您的程序确实仅与一些列交互，则限制所使用的列数，通过 SELECT 语句仅仅规定那些确实需要的列，比如：

```
SELECT CustomerID, CompanyName FROM Customers
```

但是，当添加行时，通常您就不需要这样处理了，因为您会规定所有列的值。

1. 带有 SELECT 的 WHERE 的用法

尽量减少加载到内存中的数据量的方法是在 SQL SELECT 语句上规定 WHERE 子句，这就限制了所选择的行数。例如，下面的语句：

```
SELECT * FROM Customers WHERE CustomerID = "ZACZI"
```

会仅仅将包含 ZACZI 的一行加载到内存中，使用一小部分内存就加载了整个表。也可以使用 WHERE 子句规定一个范围，如下所示的语句：

```
SELECT * FROM Orders WHERE OrderID BETWEEN 10900 AND 10999
```

就会仅加载在上述范围之中带有 OrderID 的行。

您总是可以使用 WHERE 子句限制从大型表中加载的行数。决不要将表的所有行加载到 DataSet 中，然后再使用 foreach 循环搜索它们；相反，要使用带有 WHERE 的 SELECT 语句完成这种搜索。

我们的设计目标就是确立下面两方面之间最具效率的均衡，即在运行 ADO.NET 程序的客户机上处理本机数据，在运行 SQL 的服务器上处理搜索任务。ADO.NET 对象模型和 C#比 SQL 更适合于处理复杂运算或导航逻辑。使用希望处理的表的数据填充 DataSet，并在客户机上执行这种逻辑。但是使用合适的条件限制每一张表中被选择的行数会极大地提高性能，减少所使用的内存。

2. 查看 SQL SELECT，UPDATE，INSERT 和 DELETE 命令

SQL 使用 4 种命令，用于在表中查询、更新、添加和删除行。这些命令分别是 SELECT，UPDATE，INSERT 和 DELETE 命令。在前面的范例中我们已经使用了 CommandBuilder 对象，以创建用于更新数据库的 SQL 命令：

```
SqlDataAdapter thisAdapter =
    new SqlDataAdapter("SELECT CustomerID from Customers", thisConnection);

SqlCommandBuilder thisBuilder = new SqlCommandBuilder(thisAdapter);
```

命令构建器生成 SQL 命令，用于以 SELECT 命令为基础修改数据(UPDATE，INSERT 和 DELETE)。

试试看：显示 SQL 范例

在下面创建的程序中，我们可以看到所生成的命令，这些命令带有 CommandBuilder 对象的 GetUpdateCommand()，GetInsertCommand()和 GetDeleteCommand()方法：

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

class ShowSQLExample
{
    public static void Main()
    {
        OleDbConnection thisConnection = new OleDbConnection(
                    "Provider=Microsoft.Jet.OLEDB.4.0;Data Source=NWIND.MDB");
        thisConnection.Open();

        OleDbDataAdapter thisAdapter = new
          OleDbDataAdapter("SELECT CustomerID from Customers", thisConnection);

        OleDbCommandBuilder thisBuilder = new OleDbCommandBuilder(thisAdapter);

        Console.WriteLine("SQL SELECT Command is:\n{0}\n",
                          thisAdapter.SelectCommand.CommandText);

        OleDbCommand updateCommand = thisBuilder.GetUpdateCommand();
        Console.WriteLine("SQL UPDATE Command is:\n{0}\n",
                          updateCommand.CommandText);

        OleDbCommand insertCommand = thisBuilder.GetInsertCommand();
        Console.WriteLine("SQL INSERT Command is:\n{0}\n",
                          insertCommand.CommandText);

        OleDbCommand deleteCommand = thisBuilder.GetDeleteCommand();
        Console.WriteLine("SQL DELETE Command is:\n{0}",
                          deleteCommand.CommandText);

        thisConnection.Close();
    }
}
```

此范例的输出为：

```
SQL SELECT Command is:
SELECT CustomerID from Customers

SQL UPDATE Command is:
UPDATE 'Customers' SET 'CustomerID' = ? WHERE ( 'CustomerID' = ? )

SQL INSERT Command is:
INSERT INTO 'Customers'( 'CustomerID' ) VALUES ( ? )
```

```
SQL DELETE Command is:
DELETE FROM   'Customers' WHERE ( 'CustomerID' = ? )
```

代码的说明

注意 UPDATE 和 DELETE 命令使用由 CommandBuilder 对象生成的 WHERE 子句。

问号(？)是参数的标记，其中 ADO.NET 运行时代替实际的值进入命令中；例如使用 Delete() 方法删除包含 CustomerID "ZACZI"的行，同时调用 Update()，并执行命令：

```
DELETE FROM   'Customers' WHERE ( 'CustomerID' = 'ZACZI'   )
```

清除 ZACZI 行。

注意为了输出 SELECT 命令，我们可以使用 SelectCommand 属性直接从 DataAdapter 中获取命令。DataAdapter 还具有 UpdateCommand，InsertCommand 和 DeleteCommand 属性，可以在更新时直接获取或设置 SQL 命令。熟悉 SQL 的开发人员可以优化这些命令，这些命令要比由 CommandBuilder 自动生成的命令更易于执行，尤其是当将所有的命令包括在 SQL SELECT 语句中时更是如此。

19.5.2　直接执行 SQL 命令

如果程序需要执行面向集合的操作，比如删除或更新所有满足某一条件的行，则直接使用单一的 SQL 命令要比使用在 C#代码中扩展的 SQL 命令要更有效，尤其对于大型表来说更是如此。

ADO.NET 提供 SqlCommand 或 OleDbCommand 对象，以执行 SQL 命令。这些对象可以提供 Execute()方法直接执行 SQL 命令。在本章开始，当介绍 DataReader 对象时，我们使用了 ExecuteReader 方法。下面我们介绍其他的 Execute()方法。

试试看：SqlCommand 的 ExecuteScalar 方法

同第一个范例相同，下面考虑一个程序，它获取表的行数；这类似于 DataReader 范例，但是使用了不同的 SQL 语句和 SQL Execute 方法。此范例使用 SqlCommand 的 ExecuteScalar()方法执行查询；此方法用于执行仅仅返回一个标量的 SQL 命令(标量即单一值，相对于 ExecuteReader()所返回的多行)。

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

class DataExecuteScalarExample
{
    public static void Main()
    {
        SqlConnection thisConnection = new
        SqlConnection("Data Source=(local);" +
```

```
                "Integrated Security=SSPI;Initial Catalog=northwind");
        thisConnection.Open();

        SqlCommand thisCommand = thisConnection.CreateCommand();
        thisCommand.CommandText = "SELECT COUNT(*) FROM Customers";
        Object countResult = thisCommand.ExecuteScalar();
        Console.WriteLine("Count of Customers = {0}", countResult);
        thisConnection.Close();
    }
}
```

代码的说明

此程序使用 SQL.NET 数据提供者(System.Data.SqlClient)。程序的命令解释程序与本章第一个范例相同，在具有 SSPI 安全和 Northwind 数据库的本机上打开到 SQL Server 的连接。

我们创建 SqlCommand 对象，并指定 SELECT COUNT(*)显示为命令文本。COUNT()是 SQL 函数，可以返回符合 WHERE 条件的行数。然后我们调用 SqlCommand 的 ExecuteScalar 方法执行查询，检索其数量。显示数量后退出。当再次执行 Northwind 数据库时，程序就显示：

```
Count of Customers = 91
```

这与前面范例中所使用的方法相同，即将 Customers 表加载到 DataTable 对象，并在 Rows 对象上调用 COUNT()。为什么要采用这种方法处理此工作？这是因为它取决于数据的结构和在程序中完成的其他工作。如果仅有少量数据，或者由于某种原因在 DataSet 中加载了所有的行，则可以只使用 DataTable.Rows.Count。但是，如果您希望在有 1 000 000 行的大型表中计算精确的行数，则使用带有 ExecuteScalar()方法的 SELECT COUNT(*)查询要比在内存中加载 1 000 000 行的效率要高得多。

试试看：SqlCommand 的 ExecuteNonQuery 方法

我们再介绍一个范例。SqlCommand 的 ExecuteNonQuery()方法执行根本就不返回任何数据的命令。类似 SQL INSERT，UPDATE 和 DELETE 的数据修改操作不返回任何数据；我们对这些命令感兴趣的是受影响的行数，这是此方法要返回的内容。

假设一位供应商使其产品的价格上涨了 5%。下面的程序就显示如何使用 SqlCommand 对象执行 SQL UPDATE 命令，以使得该供应商的产品的价格全部上涨 5%：

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

class DataSqlNonQueryExample
{
    public static void Main()
    {
        SqlConnection thisConnection = new SqlConnection(
```

```
            "Data Source=(local);" +
            "Integrated Security=SSPI;Initial Catalog=northwind");
        thisConnection.Open();

        SqlCommand thisCommand = thisConnection.CreateCommand();
        thisCommand.CommandText = "UPDATE Products SET " +
            "UnitPrice=UnitPrice*1.05 WHERE SupplierId=12";
        int rowsAffected = thisCommand.ExecuteNonQuery();
        Console.WriteLine("Rows Updated = {0}", rowsAffected);
        thisConnection.Close();
    }
}
```

代码的说明

同前面的范例一样，此程序打开了一个连接。我们可以创建 SqlCommand 对象，指派 UPDATE 命令显示为命令文本。然后调用 SqlCommand 的 ExecuteNonQuery 方法执行查询，然后返回数据库中受到影响的行数。显示行数并退出。当再次执行 Northwind 数据库时，程序显示：

```
Rows Updated = 5
```

表示有 5 种产品调整了价格。

19.6　关于 ADO.NET 的进一步说明

到此我们已经介绍完了 ADO.NET 概况。如果希望深入学习，可以参考 Microsoft .NET Framework SDK 在线文档。选中“Search Titles Only”选项，对 ADO.NET 进行搜索，就可以得到一列 ADO.NET 综述文章。搜索类似 DataTable 的单个对象名称可以在该特定的对象上找到引用信息，包括在此没有介绍的方法和属性。

19.7　小结

我们已经学习了 ADO.NET 是.NET Framework 的组成部分，.NET Framework 使得您可以访问关系数据库和其他数据源。设计 ADO.NET 的目的就是可以提供简单而灵活的架构，用于数据访问；它被设计用于多层应用程序体系结构，并与 XML 集成了关系数据。

ADO.NET 类包含在 System.Data 名称空间中。我们回顾了 ADO.NET 的对象模型，并学习了其主要对象的作用，包括 Connection，Command，DataReader，DataSet，DataTable，DataRow 和 DataColumn 对象。

我们学习了.NET 数据提供者可以访问具体的数据源，ADO.NET 可以通过为新数据源编写.NET 数据提供者而扩展到新数据源。我们介绍了包括在 ADO.NET 中，分别用于 Microsoft SQL Server 和 OLE DB 数据源的.NET 数据提供者，并学习了它们分别包含在 System.

Data.SqlClient 和 System.Data.OleDb 名称空间中。

介绍了如何通过 DataReader 对象对数据进行快速、只读的访问，如何编写同样适用于 SqlClient .NET 数据提供者和 OleDb .NET 数据提供者的程序。我们还学习了如何通过 DataSet，DataAdapter 和 CommandBuilder 对象更新数据和添加行，并学习了如何使用主键码找到行，以及如何删除行。

我们还学习了如何通过 DataRelation 对象在 DataSet 中访问多个表，以及如何生成该数据的 XML 视图。最后我们简要介绍了如何利用在 ADO.NET 中的 SQL 数据库语言支持，包括显示自动生成的 SQL 命令，以及直接执行 SQL 命令。

19.8　练习

(1) 修改第一个范例中的程序，以便在 Northwind 数据库中使用 Employees 表，检索 EmployeeID 和 LastName 列。

(2) 修改显示 Update()方法的第一个程序，将公司名称改变为 Bottom-Dollar Markets。

(3) 编写询问用户 CustomerID 的程序。

(4) 编写程序，为顾客“Zachary Zithers”创建一些订单；使用范例程序查看订单。

(5) 编写程序，显示 Northwind 数据库中的关系层次结构中不同于本章使用过的部分；例如，显示 Products，Suppliers 和 Categories。

(6) 将在前面范例中生成的数据写入 XML 文档。

(7) 改变用于输出所有顾客订单细节和产品的程序，为 Customers 使用 SELECT 语句的 WHERE 子句，限制要处理的顾客数量。

(8) 修改所示的程序，输出 UPDATE，INSERT 和 DELETE 语句，将“SELECT * FROM Customers”用作 SQL SELECT 命令。注意所生成的语句的复杂性。

第20章 使 用 文 件

在本章，您要学习如何读和写文本文件，这是许多.NET 应用程序基础性的工作。我们要讨论用于创建文件的主要类，对它们读和写，支持在 C#代码中用于处理文件系统的类。尽管我们不能详细介绍全部的类，但是我们可以深入介绍，使您可以理解概念和基本理论。

文件可以很方便地用于在应用程序的实例之间存储数据，或者它们也可以用于在应用程序之间传输数据。可以存储用户和应用程序配置，以便在下一次运行应用程序时检索它们。文件的用法可以随不同的应用程序而变化。可以看到，.NET Framework 给我们提供了工具，可以在应用程序中有效地使用文件。

在本章，我们要学习下面的内容：

- 流是什么，以及.NET 如何使用流类访问文件。
- 如何使用 File 对象处理文件结构。
- 如何将数据写入文件。
- 如何从文件中读出数据。
- 如何在文件中读取和写出格式化的数据。
- 如何同步(一次完成)和异步(使用批处理)访问文件。
- 如何监控要变化的文件和目录。

20.1 流

在.NET Framework 中进行的所有的输入和输出工作都要使用到流。流是串行化设备的抽象表示。串行化设备可以以线性方式存储数据，并可以以同样的方式访问：一次访问一个字节。此设备可以是磁盘文件、打印机、内存位置和或任何其他支持以线性方式读写的对象。通过使设备抽象，就可以隐藏流的基础性的目标/源。这种抽象的级别使得代码可以重用，允许我们编写更通用的例程。因此，当应用程序从文件输入流或网络输入流中读取数据时，就可以转换并重用类似的代码。而且，通过使用流，我们还可以忽略每一种设备的物理机制。因此为了从文件流中文件读取，我们无需担心磁盘开销或内存分配问题。

当向某些外部目标写数据时，就要用到输出流。这可以指物理磁盘文件、网络位置、打印机或其他程序。理解流编程技术可以带来许多高级应用。对本章而言，我们仅限于讨论写磁盘文件。

输入流用于将数据读到程序可以访问的内存或变量中。到目前为止，我们所使用过的最常见的输入流形式是键盘。输入流可以来自任何源，在此我们主要关注读磁盘文件。适用于读/写磁盘文件的概念也适用于大多数设备，因此通过本章的学习，我们就会对流有一个基本的认识，并学习可以应用于许多情况的被证明有效的方法。

20.2　用于输入和输出的类

System.IO 名称空间包含我们在本章要介绍的所有的类。System.IO 包含用于在文件中读取和写入数据的类，您必须在 C#应用程序中引用此名称空间才可以访问这些类。在 System.IO 名称空间中确实包含了不少的类，在图 20-1 中就可以看到，但是我们将仅介绍用于文件输入和输出的主要类。

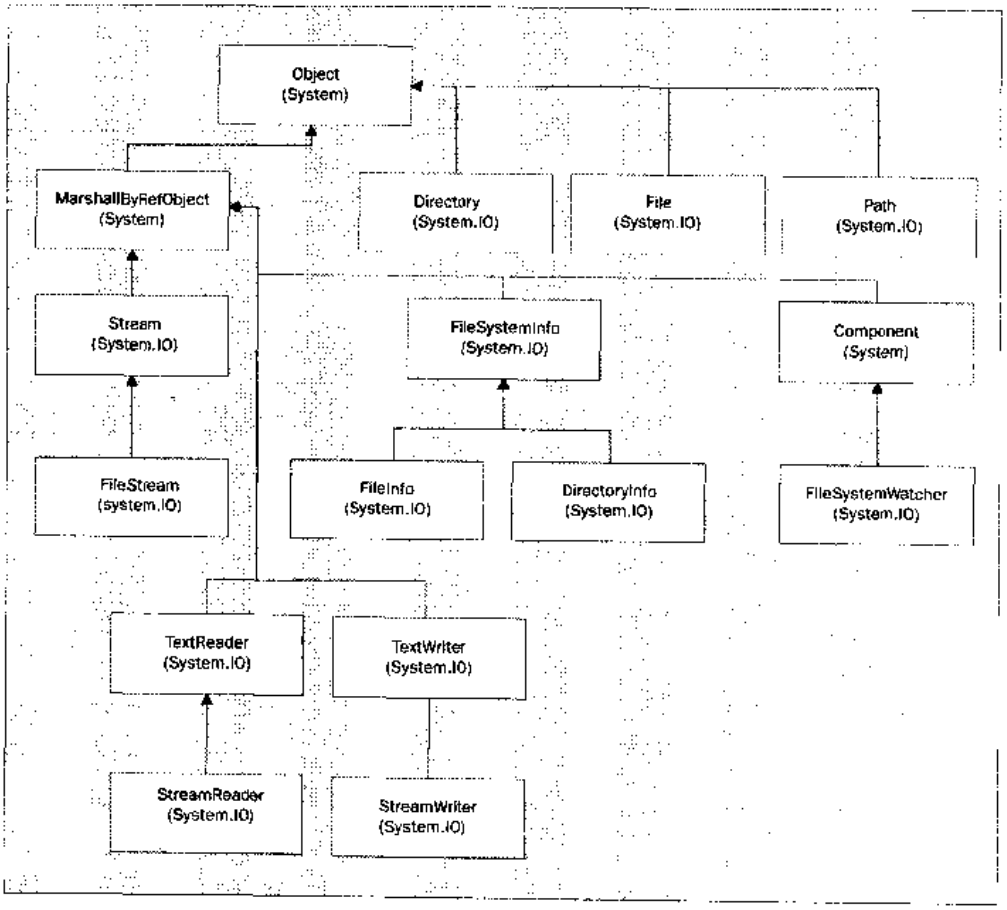


图　20-1

- File ——实用类，提供许多静态方法，用于移动、复制和删除文件。
- Directory ——实用类，提供许多静态方法，用于移动、复制和删除目录。
- Path ——实用类，用于处理路径名称。
- FileInfo ——表示磁盘上的物理文件，具有可以处理此文件的方法。要完成对文件的读取和写入工作，就必须创建 Stream 对象。
- DirectoryInfo ——表示磁盘上的物理目录，具有可以处理此目录的方法。
- FileStream ——表示可以被写或被读，或二者均可的文件。此文件可以同步或异步地读和写。在本章我们要讨论这两种方法。
- StreamReader ——从流中读取字符数据，并可以通过使用 FileStream 被创建为基类。
- StreamWriter ——向流写字符数据，可以通过使用 FileStream 被创建为基类。
- FileSystemWatcher ——FileSystemWatcher 是我们在本章要介绍的最高级的类。它用于监

控文件和目录，并在这些位置发生变化时，给出应用程序可以捕获的事件。在 Windows 编程技术中缺乏此功能，但是现在.NET Framework 使得它易于对文件系统事件作出响应。

20.2.1 File 和 Directory 类

作为实用类，File 和 Directory 类都提供了许多方法，用于处理文件系统以及其中的文件和目录。这些是静态方法，涉及移动文件、查询和更新属性并创建 FileStream 对象。在第 8 章我们学习到，可以在类上调用静态方法，而无需创建它们的实例。

File 类的一些最常用的静态方法如表 20-1 所示。

表 20-1

方　法	说　明
Copy()	将文件复制到规定的位置
Create()	在规定的路径上创建文件
Delete()	删除文件
Open()	在规定的路径上返回 FileStream 对象
Move()	将规定的文件移动到新位置。我们可以在新位置为文件规定不同的名称

Directory 类的一些常用的静态方法如表 20-2 所示。

表 20-2

方　法	说　明
CreateDirectory()	创建具有规定路径的目录
Delete()	删除规定的目录以及其中的所有文件
GetDirectories()	返回表示当前目录之下的目录的 Directory 对象的数组
GetFiles()	返回在当前目录中的 File 对象的数组
Move()	将规定的目录移动到新位置。可以在新位置为文件夹规定一个新名称

20.2.2 FileInfo 类

FileInfo 类不像 File 类，它没有静态方法，仅可用于实例化的对象。FileInfo 对象表示在磁盘或网络位置的文件。注意它不是流。为了读取或写入文件，必须创建 Stream 对象。FileInfo 对象可以帮助您完成这项工作，为此要提供几种返回实例化的 Stream 对象的方法。但是，要首先创建 FileInfo 对象，您就必须提供文件或目录的路径。

```
FileInfo aFile = new FileInfo("C:/Log.txt");
```

FileInfo 类提供的许多方法类似于 File 类的方法，但是因为 File 是静态类，所以它需要字符串参数为每一个方法调用规定文件位置。因此，下面的调用就会完成相同的工作：

```
FileInfo aFile = new FileInfo("Data.txt");
```

```
if (aFile.Exists)
Console.WriteLine("File Exists");
if (File.Exists("Data.txt"))
Console.WriteLine("File Exists");
```

大多数 FileInfo 方法以这种方式反映 File 方法。如果您仅在对象上进行单一方法调用，则可以使用静态 File 类。在此，静态调用要快一些，因为.NET Framework 不必遍历实例化新对象并调用方法的过程。但是，如果应用程序在文件上执行几种操作，则实例化 FileInfo 对象并使用其方法就更好一些。这会节省时间，因为对象将在文件系统上引用正确的文件，而静态类就必须每次都寻找文件。

FileInfo 类也提供了下面的关于基础性的文件的属性，这些属性可以用来更新文件，如表 20-3 所示。

表 20-3

属 性	说 明
Attributes	获取或者设置当前文件的属性
CreationTime	获取当前文件的创建日期和时间
DirectoryName	返回文件目录的路径
Exists	判断是否存在文件
FullName	检索文件的完整路径
Length	获取文件的容量
Name	仅仅返回文件的名称，而不是完整的文件位置路径

20.2.3 DirectoryInfo 类

DirectoryInfo 类的作用类似于 FileInfo 类。它是一个实例化的对象，表示计算机上的单一目录。同 FileInfo 类一样，在 Directory 和 DirectoryInfo 之间可以复制许多方法调用。当使用这二者时，可以使用相同的原则：如果进行单一调用，就使用静态 Directory 类。如果进行一系列调用，则使用实例化的 DirectoryInfo 对象。

DirectoryInfo 类与 FileInfo 类共享几乎所有相同的属性，DirectoryInfo 类作用在无文件的目录上除外。

Pathnames 和相对路径

当在.NET 代码中规定路径名时，您可以使用绝对路径名，也可以使用相对路径名。绝对路径名显式地规定文件或目录来自于哪一个已知的位置——比如是 C: 驱动器。它的一个范例是 C:\Work\LogFile.txt。注意它准确地定义了其位置。

相对路径名相对于应用程序在文件系统上运行的位置。通过使用相对路径名称，无需规定已知的驱动器或位置；当前的目录就是起始点。例如，如果应用程序运行在 C:\Development\FileDemo 目录上，并使用了相对路径“LogFile.txt,”，则该文件就位于 C:\Development\FileDemo\LogFile.txt 中。为了上移目录，则使用..字符。这样，在同一个应用程

序中，路径“..\Log.txt”就指向了在 C:\Development 中的 Log.txt 文件。

在开发过程中，相对路径名需要技巧的部分是，它相对于运行应用程序的位置。当使用 Visual Studio.NET 开发程序时，这就表示应用程序是创建的工程文件夹下的几个目录。它通常位于 ProjectName\bin\Debug。

20.2.4 FileStream 对象

FileStream 对象表示在磁盘或网络路径上指向文件的流。当类提供向文件读取和写入字节的方法时，经常使用 StreamReader 或 StreamWriter 执行这些功能。这是因为 FileStream 类操作字节和字节数组，而 Stream 类操作字符数据。字符数据易于使用，但是我们可以看到有些操作，比如随机文件访问，就必须由 FileStream 对象执行，稍后我们再对此进行介绍。

还有几种方法可以创建 FileStream 对象。构造函数具有许多不同的重载/版本，但是最简单的构造函数仅仅带有两个参数，即文件名和 FileMode 枚举。

```
FileStream aFile = new FileStream("Log.txt", FileMode.OpenOrCreate);
```

FileMode 枚举具有几种成员，如表 20-4 所示，可以规定如何打开或创建文件。这些枚举成员可以联合使用。

表 20-4

FileMode 枚举成员	说　明
Append	如果文件存在，就打开文件，将文件位置移动到文件的末尾，并创建一个新文件。FileMode.Append 仅可以与枚举 FileAccess.Write 联合使用
Create	创建新文件；如果存在这样的文件，就破坏它
CreateNew	创建新文件，但是如果已经存在了此文件，则抛出异常
Open	打开现有的文件。如果不存在所规定的文件，则抛出异常
OpenOrCreate	如果文件存在，则规定打开文件，否则就创建新文件。如果文件已经存在，则保留在文件中的数据
Truncate	打开现有文件，清除其内容。然后我们可以向文件写入全新的数据，但是保留文件的初始创建日期。必须存在有文件，否则就会抛出异常

前面的构造函数默认以只读模式打开文件。需要一个附加参数规定不同的访问级别，此参数即 FileAccess 参数。

```
FileStream aFile = new FileStream("Log.txt", FileMode.OpenOrCreate, FileAccess.Write)
```

此行代码将打开文件，对文件进行写访问。对文件的任何读的企图都会导致抛出异常。仅有 3 种类型的 FileAccess 枚举：Read，ReadWrite 和 Write。因此您可以打开一个文件，只读、只写或进行读写操作。此属性的作用是，基于用户的身份验证级别改变用户对文件的访问。

File 和 FileInfo 类都提供了 OpenRead()和 OpenWrite()方法，更易于创建 FileStream 对象。前者打开了只读访问的文件，后者允许您写入文件。这些都提供了快捷方式，因此您不必提供前面所有的信息。例如，下面的代码行打开了用于只读访问的 Data.txt 文件：

```
FileStream aFile = File.OpenRead("Data.txt");
```

注意下面的代码执行同样的功能：

```
FileInfo aFile = new FileInfo("Data.txt");
FileStream aFile = aFile.OpenRead();
```

1. 文件位置

FileStream 类维护内部文件指针。这会指向文件中的位置，其中会发生下一次的读或写操作。在大多数情况下，当打开文件时，它就指向文件的开始位置，但是此指针可以修改。这允许应用程序在文件的任何位置可以读或写。这允许随机访问文件，或可以直接在文件中查找具体位置。当处理大型文件时，这非常省时，因为马上可以定位到正确的位置。

实现此功能的方法是 Seek()方法：

```
public long Seek(long offset, SeekOrigin origin);
```

第一个参数规定文件指针以字节为单位的移动距离。第二个参数规定开始计算的起始位置。SeekOrigin 枚举包含 3 个值：Begin，Current 和 End。下面的代码行会将文件指针移动到文件的第 8 个字节，其起始位置就是文件的第 1 个字节：

```
aFile.Seek(8,SeekOrigin.Begin);
```

下面的代码行会将指针向前移动 2 个字节，从当前位置开始。如果直接在上面的代码行之后执行下面的代码，则文件指针就指向文件的第 10 个字节：

```
aFile.Seek(2,SeekOrigin.Current);
```

注意当您在文件中读取或写入时，文件位置也会改变。在读取了 10 个字节之后，文件指针就指向被读取的第 10 个指针之后的字节。

您也可以规定负查找位置，这可以与 SeekOrigin.End 枚举一起使用，查找文件末端的位置。下面的代码会查找文件中倒数第 5 个字节：

```
aFile.Seek( - 5, SeekOrigin.End);
```

以这种方式访问的文件有时被称为随机访问文件，因为应用程序可以访问此文件的任何位置。稍后介绍的 Stream 类可以连续地访问文件，而且不允许操作文件指针。

2. 读取数据

使用 FileStream 类读取数据不像使用 StreamReader 和 StreamWriter 类读取数据那样容易，我们在后面会介绍这些内容。这是因为 FileStream 类只能处理原始字节(raw byte)。使用原始字节的功能使得 FileStream 类可以用于任何数据文件，而不仅仅是文本文件。通过读取字节数据，FileStream 对象可以用于读取类似图像和声音的文件。这种灵活性具有代价，即不能使用 FileStream 类将数据直接读入字符串，而使用 StreamWriter 和 StreamReader 类却可以这样处理。但是有几种转换类可以很容易地将字节数组转换为字符数组，或者进行相反的操作。

FileStream.Read()方法是从 FileStream 对象所指向的文件访问数据的主要手段：

```
public int Read(byte[] array, int offset, int count);
```

第一个参数是被传输进来的字节数组，用以接受 FileStream 对象中的数据。第二个参数是字节数组中开始写入数据的位置。它通常是 0，表示从数组的开端的文件中向数组写数据。最后一个参数规定从文件中读出多少字节。

下面的范例演示了从随机访问文件中读取数据。可以读取的文件实际是为此范例创建的类文件。

试试看：从随机访问文件中读取数据

(1) 打开 Visual Studio.NET。

(2) 通过选择 File | New | Project 而创建一个新工程。选择 C# Console Application，将其命名为 ReadFile，如图 20-2 所示。

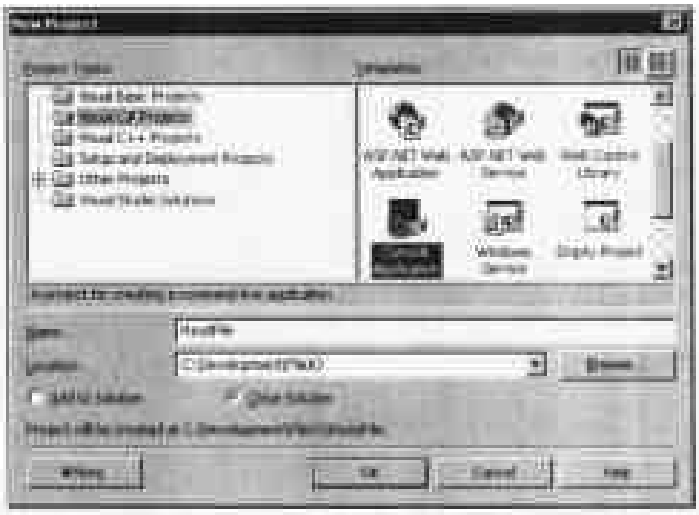


图 20-2

(3) 在 Class1.cs 文件的顶部添加下面的两个名称空间声明。System.IO 用于 FileStream 类，System.Text 用于进行对从文件中读取的字节的转换处理工作。

```
using System;
using System.IO;
using System.Text;
```

(4) 在 Main()方法中添加下面的代码：

```
static void Main(string[] args)
{
    byte[] byData = new byte[100];
    char[] charData = new Char[100];

    try
    {
        FileStream aFile = new FileStream("../../Class1.cs",FileMode.Open);
        aFile.Seek(55,SeekOrigin.Begin);
```

```
            aFile.Read(byData,0,100);
        }
        catch(IOException e)
        {
            Console.WriteLine("An IO exception has been thrown!");
            Console.WriteLine(e.ToString());
            Console.ReadLine();
            return;
        }

        Decoder d = Encoding.UTF8.GetDecoder();
        d.GetChars(byData, 0, byData.Length, charData, 0);

        Console.WriteLine(charData);
        Console.ReadLine();

        return;
    }
```

(5) 运行应用程序。可以看到如图 20-3 的输出结果。注意计算机上的输出可以与下图有很大的不同。

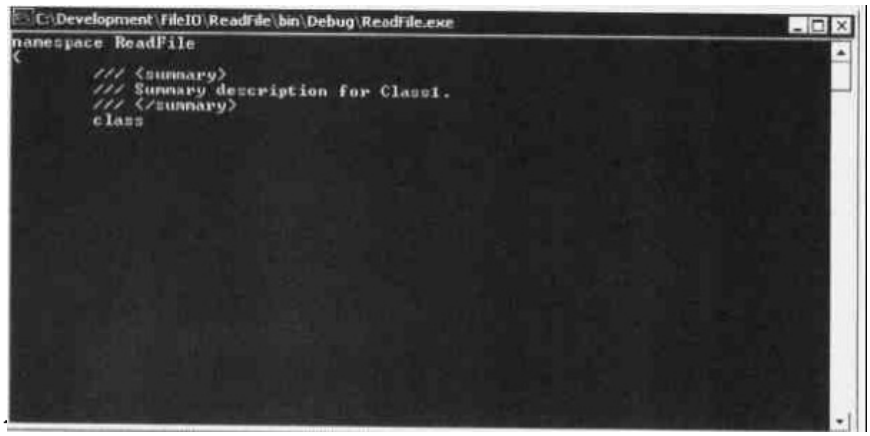


图　20-3

按回车键，关闭控制台。

代码的说明

此应用程序打开自己的.cs 文件，用于读取。它在下面的代码行中使用.. 符号逐级导航文件结构，从而完成这些工作：

```
FileStream aFile = new FileStream("../../Class1.cs",FileMode.Open);
```

实现实际查找工作，并从文件的具体位置读取字节的两行代码，如下所示：

```
try
{
    aFile.Seek(55,SeekOrigin.Begin);
```

```
        aFile.Read(byData,0,100);
    }
    catch(IOException e)
    {
        Console.WriteLine("An IO exception has been thrown!");
        Console.WriteLine(e.ToString());
        Console.ReadLine();
        return;
    }
```

第一行代码将文件指针移动到文件的第 55 个字节。这是名称空间“n”；其前面的 54 个字符是 using 指令的 3 行代码。第二行将接下来的 100 个字节读入到 byData 字节数组中。

注意可以使用在第 7 章介绍的 try-catch 异常处理。文件 IO 涉及到的所有操作都可以抛出类型 IOException 的异常。所有产品代码都必须包含错误处理，尤其是处理文件系统时更是如此。在本章的所有范例都具有错误处理的基本形式。

当从文件中获取了字节数组时，就需要将其转换为字符数组，以便我们可以在控制台显示它。为此，我们可以使用 System.Text 名称空间的 Decoder 类。此类用于将原始字节转换为更有用的项，比如字符：

```
Decoder d = Encoding.UTF8.GetDecoder();
d.GetChars(byData, 0, byData.Length, charData, 0);
```

这些代码基于 UTF8 编码模式创建了 Decoder 对象。这就是 Unicode 编码模式。然后调用 GetChars()方法，此方法带有字节数组，并将字节数组转换为字符数组。完成之后，就可以将字符数组输出到控制台。

3. 写入数据

向随机访问文件中写入的进程与从中读取字节的进程类似。首先需要创建一个字节数组；最简单的办法是首先构建我们希望向文件写入的字符数组。然后使用 Encoder 对象将其转换为字节数组，其用法非常类似于 Decoder。最后调用 Write()方法，将字节数组传送到文件中。

下面我们构建一个简单的范例演示其全部过程。

试试看：将数据写入到随机访问文件

(1) 创建一个新过程。选择 C# Console Application，将其命名为 WriteFile。

(2) 如上所示，在 Class1.cs 文件顶部添加下面的两个名称空间声明：

```
using System;
using System.IO;
using System.Text;
```

(3) 在 Main()方法中添加下面的代码：

```
static void Main(string[] args)
{
```

```
        byte[] byData = new byte[100];
        char[] charData = new Char[100];

        try
        {
            FileStream aFile = new FileStream("Temp.txt",FileMode.OpenOrCreate);
            charData = "Hello World".ToCharArray();
            Encoder e = Encoding.UTF8.GetEncoder();
            e.GetBytes(charData,0,charData.Length, byData,0,true);

            //Move file pointer to beginning of file
            aFile.Seek(0,SeekOrigin.Begin);
            aFile.Write(byData,0,byData.Length);
        }
        catch(IOException ex)
        {
            Console.WriteLine("An IO exception has been thrown!");
            Console.WriteLine(ex.ToString());
            Console.ReadLine();
            return;
        }

        return;
    }
```

(4) 运行该应用程序。稍后就将其关闭。

(5) 导航到应用程序目录——在目录中已经保存了文件，因为我们使用了相对路径。目录位于 WriteFile\bin\Debug folder。打开 Temp.txt 文件。可以在文件中看到如图 20-4 所示的文本。

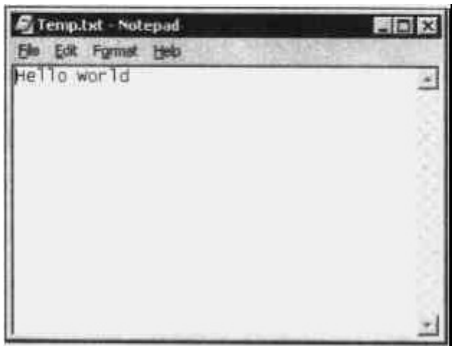


图　20-4

代码的说明

此应用程序在自己的目录中打开文件，并在文件中写入了一个简单的字符串。这与前面范例的做法非常类似。

下面的代码行创建字符数组，为此使用了 String 类的 ToCharArray()静态方法。因为在 C# 中的所有事物都是对象，文本“Hello World”实际上是一个 String 对象，所以甚至可以在字符

串上调用这些静态方法。

```
CharData = "Hello World".ToCharArray();
```

下面的代码行显示了如何将字符数组转换为 FileStream 对象需要的正确字节数组。

```
Encoder e = Endoding.UTF8.GetEncoder();
e.GetBytes(charData,0,charData.Length, byData,0,true);
```

在这一次要基于 UTF8 编码方法来创建 Encoder 对象。我们也可以将 Unicode 用于解码。此时，在写入流之前，需要将字符数据编码为正确的字节格式。在 GetBytes()方法中可以完成这些工作，它可以将字符数组转换为字节数组，并将字符数组作为第一个参数，而将该数组中起始位置的下标作为第二个参数。第三个参数是要转换的字符数量。第四个参数是在其中置入数据的字节数组，第五个参数是在字节数组中开始写入位置的下标。最后一个参数决定在结束后 Encoder 对象是否应该更新其状态，即 Encoder 对象是否仍然保留它原来在字节数组中的内存位置。在随后的调用中维持这种状态可以帮助 Encoder 对象，但是当只进行单一调用时，这就没有什么意义。对 Encoder 的最后的调用必须将此参数设置为 true，以清空其内存，释放无用单元收集的对象。

之后，使用 Write()方法向 FileStream 写入字节数组就非常简单：

```
aFile.Seek(0,SeekOrigin.Begin);
aFile.Write(byData,0,byData.Length);
```

20.2.5 StreamWriter 对象

我们已经看到，使用 FileStream 对象非常困难，您可能要问，还有简单一些的方法吗？恐怕是没有了，因为有了 FileStream 对象，通常都会把它包装在 StreamWriter 或 StreamReader 中，并使用它们的方法来处理文件。如果您不需要将文件指针改变到任意位置，则这些类就使得文件非常容易使用。StreamWriter 类允许我们将字符和字符串写入到文件中，它处理重要的转换和向 FileStream 对象写入工作。

还有许多方法可以创建 StreamWriter 类。如果已经有了 FileStream 对象，则可以使用此对象来创建 StreamWriter 类：

```
FileStream aFile = new FileStream("Log.txt",FileMode.CreateNew);
StreamWriter sw = new StreamWriter(aFile);
```

也可以直接从文件中创建 StreamWriter 对象：

```
StreamWriter sw = new StreamWriter("Log.txt",true);
```

这个构造函数具有文件名和 Boolean 值，这个 Boolean 值规定是附加文件还是创建新文件。如果此值设置为 false，则就创建一个新文件，或者截取现有文件并打开它。如果此值设置为 true，则打开文件，保留原来的数据。如果找不到文件，则创建一个新文件。

注意当创建 FileStream 对象时，您无法得到选项的范围。除了使用 Boolean 值附加或创建

新文件外，我们根本就无法像使用 FileStream 类时那样规定 FileMode 属性。而且，您也无法设置 FileAccess 属性，因此您总是具有对文件的读/写特权。为了使用任何高级参数，您必须在 FileStream 构造函数中规定这些参数，然后在 FileStream 对象中创建 StreamWriter。

试试看：输出流

(1) 创建一个新工程。创建一个新 C# Console Application 并称之为 StreamWrite。

(2) 我们将再次使用 System.IO 名称空间，因此在 Class1.cs 文件顶部添加下面的代码：

```
using System;
using System.IO;
```

(3) 在 Main()方法中添加下面的代码：

```
static void Main(string[] args)
{
    try
    {
        FileStream aFile = new FileStream("Log.txt",FileMode.OpenOrCreate);
        StreamWriter sw = new StreamWriter(aFile);

        //Write data to file
        sw.WriteLine("Hello World");
        sw.Write("This is a ");
        sw.Write("string of characters.");
        sw.Close();
    }
    catch(IOException e)
    {
        Console.WriteLine("An IO exception has been thrown!");
        Console.WriteLine(e.ToString());
        Console.ReadLine();
        return;
    }
    return;
}
```

(4) 构建并运行该工程。如果没有发现错误，则工程被很快运行之后并被关闭。因为我们在控制台上没有显示任何内容，所以在控制台中无法看到退出程序。

(5) 进入应用程序目录，找到 Log.txt 文件，它位于 StreamWrite\bin\Debug 文件夹，这是因为我们使用了相对路径。

(6) 打开文件，可以看到图 20-5 所示的字符。

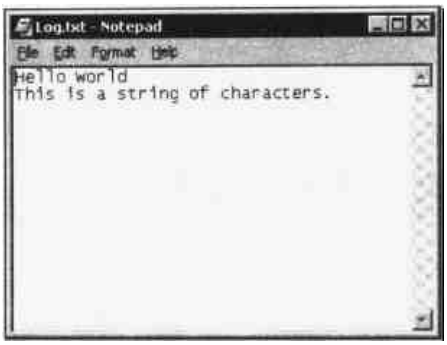


图　20-5

代码的说明

这个简单的应用程序演示了 StreamWriter 类的两个最重要的方法，即 Write()和 WriteLine()方法。这两种方法具有许多重载的版本，可以完成更高级的文件输出，但是在本范例中我们使用基本的字符串输出。

WriteLine()方法会编写传递给它的字符串，其后跟有换行符。在此范例中我们可以看到，这使得在新行上开始一个新的写操作。

Write()方法简单地将传递给它的字符串写入文件，其后却不跟有换行符。这使得我们可以使用多个 Write()语句编写完整的句子或段落。

格式化数据

如同您可以向控制台写入格式化数据一样，您也可以向文件写入格式化数据。这使用了相同的方法：Write()和 WriteLine()方法，但是它们也可以带有格式参数。例如：

```
sw.WriteLine("The date is {0}",System.DateTime.Now.ToShortDateString());
sw.Write("{0},{1},{2}","Kay","Dan","Rob");
```

在此范例中，您可以看到如何编写逗号分隔的文件。在更复杂的范例中，这些数据还可以来自 DataSet 或其他数据源。

20.2.6　StreamReader 对象

输入流用于从外部源读取数据。在很多情况下，数据源可以是磁盘上的文件或网络的某些位置。但是要记住，任何可以发送数据的位置都可以是数据源，比如网络应用程序，Web 服务，甚至是控制台。

我们用来从文件中读取数据的类是 StreamReader。同 StreamWriter 一样，这个类是一个通用类，可以用于任何流。我们会再次围绕 FileStream 对象构造 StreamReader 类，使得它指向正确的文件。

StreamReader 对象的创建方式非常类似于 StreamWriter 对象的创建方式。创建它的最常见方式是使用前面创建 FileStream 对象的方式：

```
FileStream aFile = new FileStream("Log.txt",FileMode.Open);
StreamReader sw = new StreamReader(aFile);
```

同 StreamWriter 一样，StreamReader 类可以直接在包含到具体文件路径的字符串中创建：

```
StreamReader sw = new StreamReader("Log.txt");
```

读取流输入

(1) 在 Visual Studio 中创建一个新工程。选择 C# Console Application，并命名为 StreamRead。

(2) 我们必须再次输入 System.IO 名称空间，因此将下面的代码放置在 Class1.cs 文件的顶部：

```
using System;
using System.IO;
```

(3) 在 Main()方法中添加下面的代码：

```
static void Main(string[] args)
{
    string strLine;

    try
    {
        FileStream aFile = new FileStream("Log.txt",FileMode.Open);
        StreamReader sr = new StreamReader(aFile);
        strLine = sr.ReadLine();
    //Read data in line by line
    while(strLine != null)
        {
            Console.WriteLine(strLine);
            strLine = sr.ReadLine();
        }
        sr.Close();
        Console.ReadLine();
    }
    catch(IOException e)
    {
        Console.WriteLine("An IO exception has been thrown!");
        Console.WriteLine(e.ToString());
        Console.ReadLine();
        return;
    }

    return;
}
```

(4) 在 StreamRead\bin\Debug 目录中创建 log.txt 文件。这可以是一个新文件，或者仅仅是在上一个范例中所创建的日志文件的副本。要保证在应用程序的同一个目录中有一个文件！如果目录中没有文件，则应用程序就会崩溃，因为当 FileStream 构造函数找不到文件时，它就会

抛出异常。

(5) 运行该应用程序——可以看到写入到控制台的文件文本，如图 20-6 所示。



图　20-6

代码的说明

这个应用程序非常类似于前面的应用程序。其明显的区别就是，它是在读取数据，而不是在写入数据。同前面一样，您必须导入 System.IO 名称空间，才能访问所需要的类。

我们使用 ReadLine()方法从文件中读取文本。这个方法读取文本，直到发现回车符为止，并以字符串的形式返回结果文本。当到达文件尾时就返回空值，通过这种方法我们可以测试文件是否已到达了尾部：

```
strLine = sr.ReadLine();
while(strLine != null)
{
    Console.WriteLine(strLine);
    strLine = sr.ReadLine();
}
```

Console.ReadLine()方法用于阻止控制台在我们查看所显示的结果之前迅速关闭。

1. 读取数据

ReadLine()方法不是我们在文件中访问数据的惟一方法。StreamReader 类具有许多可以读取数据的方法。

读取数据最简单的方法是 Read()。此方法将流的下一个字符作为正整数值返回，如果到达了流的结尾处，则返回 -1。这个值可以被转换为字符，为此要使用 Convert 系统实用类。在上面的范例中，程序的主体可以按如下方式编写：

```
int nChar;
nChar = sr.Read();
while(nChar != -1)
{
    Console.Write(Convert.ToChar(nChar));
    nChar = sr.Read();
}
```

对于小型文件，可以使用一种非常方便的方法，即 ReadToEnd()方法。此方法读取完整的

文件，并将其作为字符串返回。在此，前面的应用程序可以被简化为：

```
strLine = sr.ReadToEnd();
Console.WriteLine(strLine);
```

虽然这似乎非常容易和方便，但是还必须小心。通过将所有的数据读取到字符串对象中，会迫使文件中的数据放置到内存中。也许会禁止这样处理，具体要根据数据文件的大小来决定。如果数据文件非常大，最好将数据留在文件中，并使用 StreamReader 方法访问文件。

2. 分隔文件

在文件中存储数据非常常用的方法是使用逗号分隔数据。我们可以看到如何使用 StreamWriter 类编写使用这种方法存储数据的文件。使用逗号分隔的文件也易于读取。String 类提供了一种称为 Split()的方法，可以用于将字符串转换为基于所提供的分隔符字符的数组。如果我们规定逗号为分隔符，则它就会创建尺度合理的字符串数组，其中包含在初始逗号分隔的字符串中的全部数据。下面我们看一个具体范例。

试试看：逗号分隔的值

(1) 在 Visual Studio 中创建新一个工程。选择 C# Console Application，将其命名为 CommaValues。

(2) 将下面的代码放置在 Class1.cs 文件的顶部：

```
using System;
using System.IO;
```

(3) 在 Main()方法中添加下面的代码：

```
static void Main(string[] args)
{
    string strLine;
    string[] strArray;
    char[] charArray = new Char[] {','};

    try
    {
        FileStream aFile = new FileStream("Log.txt",FileMode.Open);
        StreamReader sr = new StreamReader(aFile);

        strLine = sr.ReadLine();
        while(strLine != null)
        {
            //Split row of data into string array
            strArray = strLine.Split(charArray);

            //Loop through string array – print to console
```

```
            for(int x=0;x<=strArray.GetUpperBound(0);x++)
            {
                Console.WriteLine(strArray[x].Trim());
            }
            strLine = sr.ReadLine();
        }

        sr.Close();
        Console.ReadLine();
    }
    catch(IOException e)
    {
        Console.WriteLine("An IO exception has been thrown!");
        Console.WriteLine(e.ToString());
        Console.ReadLine();
        return;
    }

    return;
}
```

(4) 打开记事本，写入下面的文本：

```
Amanda, Kay, Sandy, Dan, Rob
Natalie, Kate, David, Zoe, Andrew
```

(5) 在 CommaValues\bin\Debug 目录中将文件保存为 log.txt，也可以将其保存在可以运行应用程序可执行文件的任何位置。

(6) 运行该应用程序——可以看到写入到控制台的文件文本，如图 20-7 所示。



图　20-7

代码的说明

同前面的范例相同，这个应用程序也逐行地把数据读入 String 对象。但是，因为我们知道这是一个包含使用逗号分隔的文本值的文件，所以我们准备以不同的方式处理它。下面的代码为每一行数据生成字符串数组：

```
strArray = strLine.Split(charArray);
```

此方法接收字符数组，我们在前面已经创建了这些字符数组。注意在此范例中，我们仅仅在数组中有一个单一字符：一个逗号。在数组中可以有多个字符；在此，每次在字符串中出现一个字符时，它会被认为是有效的域分隔符。例如，下面的代码在数组中创建一个带有多个字符的字符数组。当在字符串“Please split?this|string up”上调用 Split 方法时，就会在字符串数组中返回每一个单词，因为所使用的每一个分隔符(空格、问号和垂直线)都被传送回 Split 方法中：

```
char[] charArray = new Char[] {'?','|'};
strArray = "Please split?this|string up".Split(charArray);
```

在我们检索了字符串数组之后，就可以循环数组并将其内容输出到控制台。可以通过简单的 for 循环完成这种工作。注意我们可以调用 String 类的 Trim()方法。此方法清除字符串开头和结束处的所有空白(空格和标记)。进行这种处理工作的原因是，我们只对数据名称本身感兴趣，而对数据起始位置和结束位置处的空格不感兴趣。对于应用程序您可能并不希望这样处理，因为应用程序中的空格具有意义：

```
for(int x=0;x<=strArray.GetUpperBound(0);x++)
{
    Console.WriteLine(strArray[x].Trim());
}
```

可以看到，使用.NET Framework 从逗号分隔的变量(CSV)文件中提取数据非常容易。虽然这只是一个简单范例，但是您可以发现这些原则也适用于企业级别的商业应用程序。甚至 XML 也是一个存储和传输数据的优秀方法，您可以发现 CSV 文件仍然非常普遍，还会使用相当长的时间。分隔文件的优点是非常简洁，因此要比 XML 文件小些。

分隔文件被许多旧系统应用，如果应用程序必须与这种系统交互，就可以发现分隔数据格式使用得更频繁些。注意我们已经介绍了大多数逗号分隔的文件，但是分隔符极可能采用标签——这样就创建了另一种常见的数据格式，即标签分隔数据。还要记住的是，可以在 Excel，Access 以及许多常见的关系数据库系统中输出和输入 CSV 文件。

20.3 高级主题

在这一节，我们要学习监控文件系统，将变动通知给应用程序，并学习异步访问文件，可以在应用程序继续执行其他进程时，在后台加载大型文件等内容。

20.3.1 监控文件结构

有时，应用程序所需要完成的工作不仅仅限于将文件读到文件系统并从文件系统中写出文件。有时，知道修改文件或目录的时间非常重要。.NET Framework 使得创建完成这些任务的定制应用程序变得容易了。

帮助我们完成这些任务的类是 FileSystemWatcher 类。这个类提供几种应用程序可以捕获的事件。这使得应用程序可以对文件系统事件作出响应。

使用 FileSystemWatcher 的基本过程非常简单。首先我们必须设置一些属性，这些属性规定

监控的位置、内容以及 FileSystemWatcher 引发应用程序要处理的事件的时间。然后我们给 FileSystemWatcher 提供定制事件处理程序的地址，因此当发生重要事件时，FileSystemWatcher 就调用这些属性。然后打开 FileSystemWatcher，等待事件。

在启用 FileSystemWatcher 对象之前必须设置的属性如表 20-5 所示。

表 20-5

属　性	说　明
Path	它设置要被监控的文件或目录位置
NotifyFilter	这是 NotifyFilters 枚举值的组合，NotifyFilters 枚举值规定在被监控的文件内要监控哪些内容。这些表示要被监控的文件或文件夹的属性。如果任何规定的属性发生了变化，则就引发事件。可能的枚举值是：Attributes，CreationTime，DirectoryName，FileName，LastAccess，LastWrite，Security 和 Size。注意可以通过二进制 OR 操作符联合使用这些枚举值
Filter	在其上监控文件的过滤器，例如，*.txt

设置之后，您必须为 4 种事件编写事件处理程序，这 4 种事件分别是：Changed，Created，Deleted 和 Renamed。正如我们在第 12 章介绍的，这仅仅需要创建您自己的方法并将此方法赋给对象的事件即可。通过将自己的事件处理程序赋给这些方法，就可以在引发事件时调用方法。当修改与 Path，NotifyFilter 和 Filter 属性匹配的文件或目录时，就引发每一个事件。

在设置了属性和事件之后，将 EnableRaisingEvents 属性设置为 true，就可以开始监控工作。

试试看：监控文件系统

(1) 构建一个更复杂的范例，此范例使用我们在本章所介绍的大多数内容。创建一个新的工程。这一次选择 C# Windows Application，将其命名为 FileWatch。

(2) 使用表 20-6 所示的属性设置各种表单属性。

表 20-6

属　性	设　置
FormBorderStyle	FixedDialog
MaximizeBox	False
MinimizeBox	False
Size	302, 160
StartPosition	CenterScreen
Text	FileMonitor

(3) 使用下面的列表在表单中添加所需要的控件，并设置正确的属性：

- TextBox ——Name：txtLocation；Location：8，24；Size：184，20；Text：nothing
- Button ——Name：cmdBrowse；Location：208，24；Size：64，24；Text：Browse...

- Button ——Name：cmdWatch；Location：88，56；Size：80，32；Text：Watch!；Enabled：false (我们不能在文件被规定之前就监控它)
- Label ——Name：lblWatch；Location：8，104；Size：264，32；Text：nothing
- OpenFileDialog ——Name：FileDialog；Filter：All Files|*.*

结束时，表单应如图 20-8 所示。按钮上的灰色区域是 Label 控件，可以显示应用程序的当前状态。

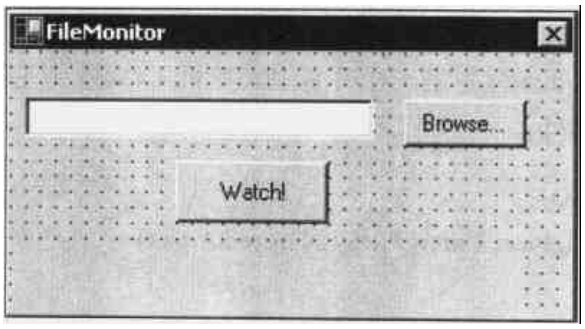


图 20-8

(4) 下面添加一些代码，使得此表单可以正常使用。在 Solution Explorer 中的 Form1.cs 上右击，然后单击 Show Code。通常，需要将 System.IO 名称空间添加到工程中，以便访问需要的类。可以在 Form1.cs 文件的顶部添加下面的代码：

```
using System;
using System.Drawing;
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;
using System.Data;
using System.IO;
```

(5) 接下来我们必须在 Form1 类中添加 FileSystemWatcher 类。在 WinForms 对象的专用声明之下添加下面的代码：

```
private System.Windows.Forms.TextBox txtLocation;
private System.Windows.Forms.Button cmdBrowse;
private System.Windows.Forms.Button cmdWatch;
private System.Windows.Forms.Label lblWatch;
private System.Windows.Forms.OpenFileDialog FileDialog;

//File System Watcher object
private FileSystemWatcher watcher;
```

(6) 在 InitializeComponent() 方法中还需要添加下面的代码。为了可以查看 InitializeComponent() 方法，您首先需要扩展标记为 Windows Form Designer generated code(Windows Form Designer 生成的代码)的区域。为此在屏幕左侧带有加号的小箱上单击即可。您可以看到 InitializeComponent()方法，它还带有由 Visual Studio.NET 所填充的大量现有的代码。

在 InitializeComponent()方法的末尾添加下面的代码。这些代码用来初始化 FileSystemWatcher 对象，并将事件与我们在下面要创建的方法关联起来：

```
this.watcher = new System.IO.FileSystemWatcher ();
this.watcher.Deleted += new System.IO.FileSystemEventHandler(this.OnDelete);
this.watcher.Renamed += new System.IO.RenamedEventHandler(this.OnRenamed);
this.watcher.Changed += new System.IO.FileSystemEventHandler(this.OnChanged);
this.watcher.Created += new System.IO.FileSystemEventHandler(this.OnCreate);
```

(7) 在 InitializeComponent()方法之后添加下面 4 种方法。这些方法用来处理 FileSystemWatcher 引发的事件：

```
// Define the event handlers.
public void OnChanged(object source, FileSystemEventArgs e)
{
    try
    {
        StreamWriter sw = new StreamWriter("C:/FileLogs/Log.txt",true);
        sw.WriteLine("File: {0} {1}", e.FullPath, e.ChangeType.ToString());
        sw.Close();
        lblWatch.Text = "Wrote change event to log";
    }
    catch(IOException ex)
    {
        lblWatch.Text = "Error Writing to log";
    }

}

public void OnRenamed(object source, RenamedEventArgs e)
{
    try
    {
        StreamWriter sw =new StreamWriter("C:/FileLogs/Log.txt",true);
        sw.WriteLine("File renamed from {0} to {1}", e.OldName, e.FullPath);
        sw.Close();
        lblWatch.Text = "Wrote renamed event to log";
    }
    catch(IOException ex)
    {
        lblWatch.Text = "Error Writing to log";
    }
}

public void OnDelete(object source, FileSystemEventArgs e)
{
```

```
    try
    {
        StreamWriter sw = new StreamWriter("C:/FileLogs/Log.txt",true);
        sw.WriteLine("File: {0} Deleted", e.FullPath);
        sw.Close();
        lblWatch.Text = "Wrote delete event to log";
    }
    catch(IOException ex)
    {
        lblWatch.Text = "Error Writing to log";
    }
}

public void OnCreate(object source, FileSystemEventArgs e)
{
    try
    {
        StreamWriter sw = new StreamWriter("C:/FileLogs/Log.txt",true);
        sw.WriteLine("File: {0} Created", e.FullPath);
        sw.Close();
    lblWatch.Text = "Wrote create event to log";
    }
    catch(IOException ex)
    {
        lblWatch.Text = "Error Writing to log";
    }
}
```

(8) 现在我们为浏览按钮添加事件处理程序。最简单的办法是让 IDE 来完成这些工作。单击返回到表单的设计视图。双击 Browse...按钮。下面的代码就会打开 File Open 对话框，这就允许用户选择要监控的文件。ShowDialog()方法可以完成这些工作：

```
private void cmdBrowse_Click(object sender, System.EventArgs e)
{
    if (FileDialog.ShowDialog() != DialogResult.Cancel )
    {
        txtLocation.Text = FileDialog.FileName;
    }
    cmdWatch.Enabled = true;
}
```

ShowDialog()方法返回表示用户如何退出 File Open 对话框的 DialogResult 枚举值。用户可能单击 OK 按钮或者单击 Cancel 按钮。我们需要检查是否用户未单击 Cancel 按钮，因此要在将用户的文件选择保存到 TextBox 之前，将方法调用的结果与 DialogResult.Cancel 枚举值作出比较。最后，将 Watch 按钮的 Enabled 属性设置为 true，以便可以监控文件。

(9) 现在处理最后一点代码。遵循上面处理 Watch 按钮时同样的过程。添加下面的代码启动 FileSystemWatcher：

```
private void cmdWatch_Click(object sender, System.EventArgs e)
{
    watcher.Path =Path.GetDirectoryName(txtLocation.Text);
    watcher.Filter = Path.GetFileName(txtLocation.Text);
    watcher.NotifyFilter = NotifyFilters.LastWrite | NotifyFilters.FileName | NotifyFilters.Size;
    lblWatch.Text = "Watching " + txtLocation.Text;
    // Begin watching.
    watcher.EnableRaisingEvents = true;
}
```

(10) 我们还需要确保 FileLogs 目录存在，以便于我们在其中写入数据。在 Form1 构造函数中添加下面的代码，Form1 构造函数会检查目录是否存在；如果不存在，则创建新目录。

```
public Form1()
{
    //
    // Required for Windows Form Designer support
    //
    InitializeComponent();

    DirectoryInfo aDir = new DirectoryInfo("C:/FileLogs");
    if(!aDir.Exists)
        aDir.Create();
}
```

(11) 构建工程。如果成功地构建了所有内容，则单击 Browse 按钮，并在计算机上选择一个文本文件。

(12) 单击 Watch 按钮，开始监控文件。在应用程序中可以见到的惟一变化是，您可以看到一个显示文件正在被监控的标签控件。

(13) 使用 Windows Explorer 导航到您正在监控的同一个文件。在记事本中打开它，并添加一些文本。保存此文件。

(14) 现在可以检查日志文件，以查看其变化。导航到 C:\FileLogs\Log.txt 文件，在记事本中打开它。可以看到对选择来进行监控的文件所作出的变动的说明，如图 20-9 所示。

代码的说明

此应用程序非常简单，但是它演示了 FileSystemWatcher 的工作原理。试着处理一下您在监控器文本框中所添加的字符串。如果在目录中规定*.*，则它就监控目录中所发生的所有变化。

在应用程序中的大多数代码的作用都围绕建立 FileSystemWatcher 对象，监控正确的位置：

```
watcher.Path = Path.GetDirectoryName(txtLocation.Text);
watcher.Filter = Path.GetFileName(txtLocation.Text);
```

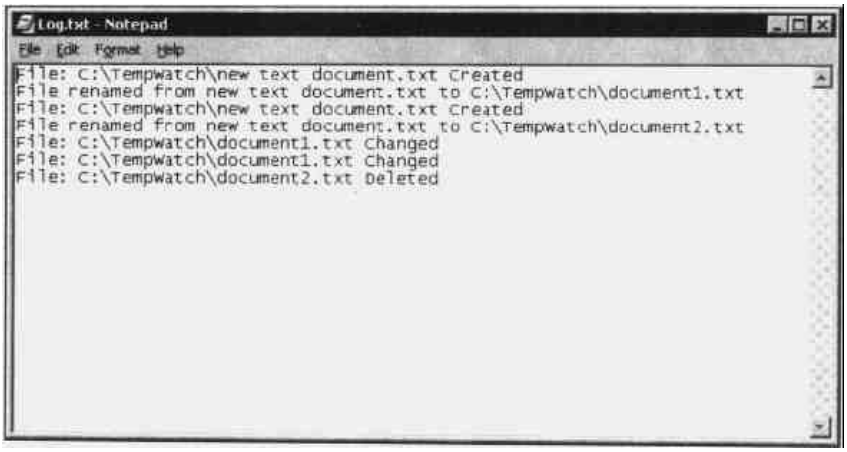


图 20-9

```
watcher.NotifyFilter = NotifyFilters.LastWrite | NotifyFilters.FileName | NotifyFilters.Size;

lblWatch.Text = "Watching " + txtLocation.Text;
// Begin watching.
watcher.EnableRaisingEvents = true;
```

第一行代码设置要监控的目录的路径。使用了我们还未介绍的新对象：System.IO.Path 对象。这是一个静态类，其作用非常类似于静态 File 对象。它给出了许多静态方法，以处理和提取文件位置字符串中的信息。我们首先使用它提取用户在文本框中输入的目录名称，为此要使用 GetDirectoryName()方法。

第二行代码设置对象的过滤器，过滤器可以是一个实际的文件，在这种情况下，它仅仅监控文件；过滤器也可以是类似于*.txt 的文件，表示要监控所规定的目录中的所有.txt 文件。我们也可以使用 Path 静态对象从所提供的文件位置中提取信息。

NotifyFilter 是 NotifyFilters 枚举值的组合，规定组成变化的内容。在此，如果最后写入的时间信息、文件名称或文件大小发生了变化，则它就将此变化通知给我们。在更新了 UI 之后，我们可以将 EnableRaisingEvents 属性设置为 true，开始监控。

但是在这之前，我们还要创建对象，设置事件处理程序。

```
this.watcher = new System.IO.FileSystemWatcher();
this.watcher.Deleted += new System.IO.FileSystemEventHandler(this.OnDelete);
this.watcher.Renamed += new System.IO.RenamedEventHandler(this.OnRenamed);
this.watcher.Changed += new System.IO.FileSystemEventHandler(this.OnChanged);
this.watcher.Created += new System.IO.FileSystemEventHandler(this.OnCreate);
```

这就是我们使用创建的专用方法为监控器对象挂接事件处理程序的方式。在我们自己的方法中决定了如何处理实际事件。注意在事件发生之后我们才得到通知。

在实际的事件处理程序方法中，我们仅仅将事件写入日志文件。显然，这是依赖于我们的应用程序的更复杂的响应。当在目录中添加了文件时，我们可以将其移动到别处，或通过使用信息读取其内容并引发新的进程。其可能的用法是无穷的！

20.3.2 异步访问文件

本章所有的文件访问都是同步的。这表示直到完成了读或写操作，才能继续进行应用程序流。对于大多数应用程序都适用同步文件访问，但是有时您需要写入大量的数据，而您不希望在完成写入工作之前让应用程序等待。或者您需要从网络的某个位置的文件上读取数据，但是网络连接速度却非常慢；在加载数据期间，应用程序可以继续进行其他处理工作或者更新用户界面元素。这种类型的数据访问方式就是异步的。其主要意思是，随着类似于 IO 的进程，应用程序还可以继续处理其他工作，而不必等待进程结束再处理。

FileStream 类给出了几种方法，允许对文件进行异步访问。这是访问文件更困难的方法，但是它却使得您可以在写入或读出文件时完成其他处理工作。读取文件的进程首先调用方法 BeginRead()。此方法接受与基本 Read 方法的参数相类似的参数，但是还具有一些附加参数，可以启动异步进程：

```
IAsyncResult BeginRead(byte[] array, int offset, int numBytes, AsyncCallback userCallback, object
stateObject);
```

前 3 个参数完全等同于 Read 方法的参数。UserCallback 参数是定制的方法，当读取操作结束时，.NET Framework 就会将这种定制方法调用回应用程序中。我们在后面会介绍这些内容。因为此方法并不是必须的，所以您可以给它传递 null，如同在第一个范例中那样处理。最后一个参数是用户提供的状态对象，与回调方法一起使用，以便分隔多个异步操作。因为您可以使用一个回调方法负责许多不同的操作，所以您可以使用此参数来传递规定异步操作的字符串或数据值。因为我们还没有使用回调方法，所以我们可以仅传入 null。

所返回的 IAsyncResult 对象是读取进程的抽象表示。它保护许多属性和方法，允许您查询进程的状态，以及判断进程是否结束了。

为了完成读取进程，就要调用 EndRead()方法，并传入从 BeginRead()调用返回的 IAsyncResult 对象。在下面的范例中我们介绍其工作原理。

试试看：异步读取文件

(1) 在 Visual Studio 中创建一个新工程。选择 C# Console Application，并将其命名为 AsyncRead。

(2) 在 Class1.cs 顶部放置下面的代码：

```
using System;
using System.IO;
using System.Text;
```

(3) 在 Main()方法中添加下面的代码：

```
static void Main(string[] args)
{
    byte[] byData = new byte[100];
    char[] charData = new Char[100];
```

```
try
{
    FileStream aFile = new FileStream("../../Class1.cs ",FileMode.Open);
    aFile.Seek(0,SeekOrigin.Begin);

    //Begin the asynchronous read from the file
    System.IAsyncResult asResult = aFile.BeginRead(byData,0,100,null,null);

    //Do other processing while data is being read
    while (!asResult.IsCompleted)
    {
        Console.WriteLine("Reading from file...");
    }

    aFile.EndRead(asResult);
    Decoder d = Encoding.ASCII.GetDecoder();
    d.GetChars(byData,0,byData.Length,charData,0);
    Console.WriteLine(charData);

    Console.ReadLine();
    return;    }
catch(IOException e)
{
    Console.WriteLine("An IO exception has been thrown!");
    Console.WriteLine(e.ToString());
    Console.ReadLine();
    return;
}
}
```

(4) 运行此应用程序——可以看到，在几行“Reading From File...”代码之后就是写入文件的 Class1.cs 文件的开始代码，如图 20-10 所示。

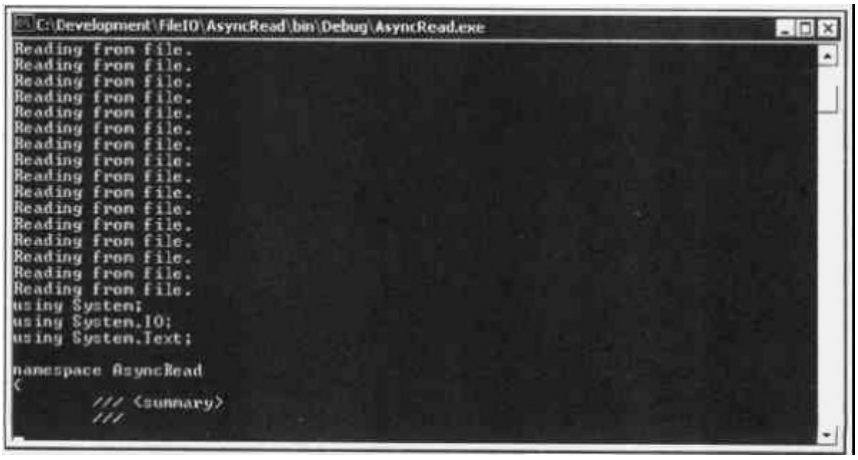


图 20-10

代码的说明

此范例的用意是什么呢？实际上其作用非常多。基本上来说，我们首先以下面的代码行读取文件：

```
System.IAsyncResult asResult = aFile.BeginRead(byData,0,100,null,null);
```

返回 IAsyncResult 对象，这样我们就可以进行查询，看一下读进程是否已结束了。如果读进程还没有结束，我们就可以进行其他处理工作，在后台读取文件：

```
while(!asResult.IsCompleted)
{
    Console.WriteLine("Reading from file...");
}
```

此范例仅仅在等待期间向控制台进行了写入操作，但是您自己的应用程序却可以完成许多复杂的同期处理工作。但是由于我们在等待 Read()结束时仅仅添加了 while 循环，所以还不能完成太多的其他工作。幸运的是，还有一种方法可以弥补这种不足——有时可以加入 BeginRead 参数增加应用范围。

1. 事件驱动异步访问文件

以异步方式访问文件的另一种方法要使用回调方法，当操作结束时，回调方法允许将.NET Framework 调入到应用程序中。按这种方式，一旦读取操作结束，就可以通知应用程序，使得应用程序开始处理其他工作。

为此，您必须创建接受 IAsyncResult 对象的方法。此方法可以是当前类的一部分，或者完全是另一个类。在此，我们使得它成为类的成员，但是这并不是必须规定的。

试试看：事件驱动异步文件读取

(1) 在 Visual Studio 中创建一个新工程。选择 C# Console Application，并将其命名为 AsyncReadEvents。

(2) 将类的名称 Class1 改变为 AsynchRead。其代码应该如下：

```
using System;

namespace AsyncReadEvents
{
    /// <summary>
    /// Summary description for Class1.
    /// </summary>
    class AsynchRead
    {
        static void Main(string[] args)
        {
            //
            // TODO: Add code to start application here
```

```
            //
        }
    }
}
```

(3) 将下面的代码放置在 Class1.cs 的顶部：

```
using System;
using System.IO;
using System.Text;
```

(4) 在此范例中我们所处理的工作稍微有些不同。在 AsynchRead 类中添加下面的代码。

```
public class AsynchRead
{
    byte[] byData;
    char[] charData;
```

(5) 添加下面的代码，将其直接放置在刚刚定义过的变量的下面，以便为 AsynchRead 类添加构造函数：

```
public AsynchRead()
{
    byData = new byte[100];
    charData = new Char[100];

    try
    {
        FileStream aFile = new FileStream("../../Class1.cs",FileMode.Open);
        System.AsyncCallback cb = new AsyncCallback(this.HandleRead);
        System.IAsyncResult aResult =  aFile.BeginRead(byData,0,100,cb,"Read Class1.cs");
    }
    catch(IOException e)
    {
        Console.WriteLine("An IO exception has been thrown!");
        Console.WriteLine(e.ToString());
        Console.ReadLine();
        return;
    }
}
```

(6) 接下来添加我们在上面的 BeginRead()方法中注册的回调方法 HandleRead()。将下面的代码添加到 AsynchRead 类中的某个位置中。

```
private void HandleRead(IAsyncResult ar)
{
    Decoder d = Encoding.UTF8.GetDecoder();
```

```
        d.GetChars(byData, 0, byData.Length, charData, 0);
        Console.WriteLine("Value of state object for this operation: {0}", ar.AsyncState);
        Console.WriteLine(charData);
    }
```

(7) 最后，在 Main()方法中添加下面的代码。这就实例化一个新 AsynchRead 对象，它使得此应用程序开始工作：

```
static void Main(string[] args)
{
    AsynchRead aClass = new AsynchRead();

    for(int x=0;x<200;x++)
    {
        Console.WriteLine(x);
    }

    Console.ReadLine();
    return;
}
```

(8) 运行此应用程序。可以看到如图 20-11 所示的输出结果。

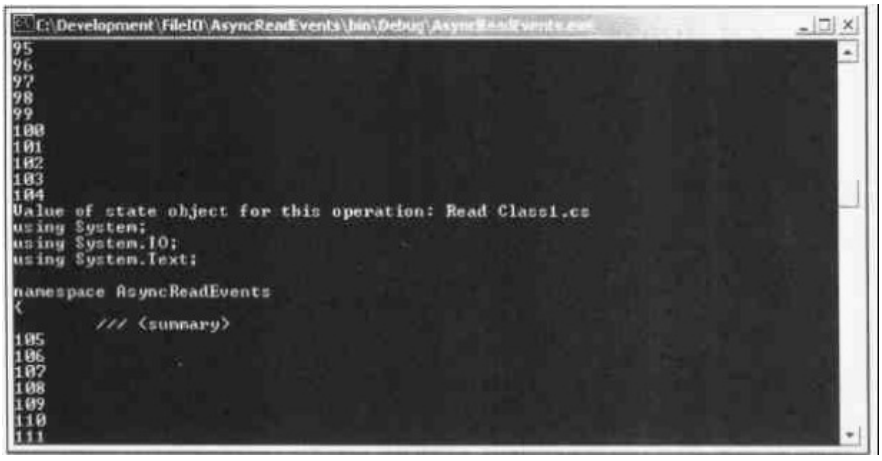


图　20-11

代码的说明

这个范例完全不同于第一个范例。在此范例中，Main()方法实例化一个 Class1 对象，然后继续进行它自己的处理工作——它所要完成的就是向 Console 写入连续数字，但是要注意的是，它根本就不必监控文件读取进程，可以完成其他任何工作。它是基于事件的进程，可以继续进行其他完全与读取进程无关的处理工作。

在 Class1 构造函数中，开始进行异步读取文件，系统注册回调方法。在打开 FileStream 对象之后，下面两行代码就可以完成所有的工作。

```
System.AsyncCallback cb = new AsyncCallback(this.HandleRead);
```

```
System.IAsyncResult aResult = aFile.BeginRead(byData,0,100,cb,"Read Class1.cs");
```

第一行代码创建一个 AsyncCallback 对象，这个对象被要求作为 BeginRead()方法的第四个参数。为此，我们需要构造一个新的 AsyncCallback 对象，传递方法的地址，这个方法将 IAsyncResult 作为其惟一的参数来接受。第二行代码调用 BeginRead()方法，传递我们在前面介绍的标准数据，以及 AsyncCallback 对象和字符串“Read Class1.cs”，这是用户提供的状态对象，在后面的回调方法中我们会使用这个对象。在这里，使用这个对象意义不大，因为我们仅仅有一个异步读取请求，但是如果有多个异步请求，则我们就需要区分每一种请求。

HandleRead()方法不会完成我们在前面介绍的任务。它仅仅将数据解码，将字符数组写入控制台。我们还可以输出 IAsyncResult 对象的 AsyncState 属性值，以演示状态对象已经将该属性值置入了回调方法中。

```
private void HandleRead(IAsyncResult ar)
{
    Decoder d = Encoding.UTF8.GetDecoder();
    d.GetChars(byData, 0, byData.Length, charData, 0);
    Console.WriteLine("Value of state object for this operation: {0}", ar.AsyncState);
    Console.WriteLine(charData);
}
```

有趣的是此方法不是由应用程序调用的。当文件读取进程结束时，它由.NET Framework 调用。这就允许当进行相对慢的文件输入进程时，应用程序可以继续处理其他信息。

2. 异步写入文件

到目前为止，我们仅仅介绍了异步读取文件，但是这些技术也适用于以异步方式写入文件。您还可以选择是否使用事件，使用事件和不使用事件各有利弊。因为我们已经多次介绍了如何写入文件，而且异步访问文件的基本原则相同，所以在本章后面只安排了异步写入文件的练习。

20.4 小结

本章我们学习了流和在.NET Framework 中使用流访问文件和其他串行设备的原因。我们介绍了在 System.IO 名称空间中的基类：File，FileInfo 和 FileStream，以及它们的作用。我们还介绍了 StreamReader 和 StreamWriter 类，以及它们在写入流时的作用。我们学习了使用 FileStream 类读取和写入随机文件的方法。最后我们学习了使用.NET Framework 可以构建的更高级的解决方案。我们还构建了完整的应用程序，可以使用 FileSystemWatcher 类监控文件和目录。

概括地说，我们在本章学习了：

- 打开文件
- 读取文件
- 写入文件
- StreamWriter 和 StreamReader 类与 FileStream 类的区别
- 异步访问文件的优缺点

- 使用 FileSystemWatcher 类监控文件系统

20.5 练习

(1) 必须输入哪一个名称空间才允许应用程序使用文件？

(2) 何时使用 FileStream 对象，而不是使用 StreamWriter 对象写入文件？

(3) StreamReader 类的哪些方法允许从文件中读取数据，每一个方法的具体作用是什么？

(4) FileSystemWatcher 类提供了哪些事件，其作用是什么？

(5) 修改我们在本章构建的 FileWatch 应用程序。添加使得无需退出应用程序就可以打开和关闭文件系统监控功能。

(6) 创建异步向文件写入数据的应用程序。创建大量的随机字节数据，并将其写入文件。

第21章 .NET装配件

21.1 装配件简介

装配件是作为整体发布的.NET 可执行程序(或者是可执行程序的一部分)。装配件是一种用于对 C#程序打包的工具，用于执行和发布 C#程序。当构建 C#程序时，所生成的.exe 文件就是装配件。如果构建了类库，则所生成的 DLL(动态链接库，Dynamic Link Library)文件也是装配件。

提示：

在一个装配件中的所有代码作为整体被构建、发布，并赋予版本号码。装配件导出公共类、属性和方法，其他程序可以看到这些类、属性和方法。程序专用的所有内容都放置在装配件中。读取装配件可以获取更多的细节。可以通过在.NET Framework 在线文档中搜索术语“Assembly”找到附加信息。

虽然所有的 C#程序都被打包为一个装配件，但是装配件的许多功能却是用来便于发布一类特殊的称为组件的程序。理解组件对于理解装配件的优点是至关重要的，因此我们回顾一下组件。

21.1.1 组件

组件是子程序或程序的一部分，它被其他程序所使用。另外，组件还是二进制单元(换句话来说，是可执行代码，而不是源代码)，可以被其他程序所使用，而无需再次编译组件的源代码或使用组件的程序的源代码。这表示第三方提供商无需提供这些组件的源代码。

组件可以按不同方式定义。其最宽松的意思包括二进制子程序。按这种解释，DLL 从定义上来说都是组件，因为它是包含可执行代码的子程序。

较严格的定义要求组件提供一种手段，可以将其内容告诉给其他程序。装配件在.NET 中提供这种广告功能。

最严格的定义要求组件提供已知的接口，释放不再使用的系统资源并提供与设计工具的集成功能。在.NET Framework 中，从严格意义上来说，要求组件实现 System.ComponentModel.IComponent 接口，此接口可以提供这些功能。

要理解装配件的优点，我们需要使用其不太严格的定义，包括广告组件内容的必要条件。

21.1.2 组件的优点

组件可以提高子程序的重用性和灵活性，并便于子程序的发布。另外，重用二进制代码可以节省时间并增加程序的可靠性。

例如，考虑名称为 Shapes 的类，它包含代表圆、三角形或其他图形的对象。它可能包含用于计算图形面积或执行图形的其他操作的方法。许多程序可能要使用 Shapes 类：着色/绘图程序、工程设计、建筑/施工设计、计算机辅助设计、游戏及其他程序。

如果绘制和处理图形的例程仅仅被定义一次，而被许多程序重用，这岂不很好？这就是可重用的好处。

如果实现重用的功能，但却不必为使用此功能的所有程序再次编译和链接 Shapes 类库行不行呢？这可以节省时间并有助于提高可靠性(因为当再次编译并再次链接时，可能引入程序错误)。

更妙的是，可能其他人或公司已经编写出了图形组件，可以完成您所希望的任务；然后您就可以使用这些组件(下载和/或购买组件)，而不必自己编写组件。如果可以在二进制水平上共享组件，则就不必关心开发组件时所使用的编程语言的类别。.NET Framework 和装配件具有所有这一切优点。

21.1.3 组件的简短历史

为了使得不同的程序可以在二进制水平上重用组件，就必须具有用于实现命名类和对象并在二进制水平上使用类和对象的标准方法。在基于 Microsoft 的产品中完成这些任务的标准在不断地演进。

Microsoft Windows 引入了 DLL(Dynamic-Link-Library)，其中一个或多个程序可以使用存储在单独文件中的代码块。如果程序以同种语言编写(通常是 C)，就可以在非常基本的层次上使用 DLL。但是程序需要事先了解它们所使用的 DLL，而且 DLL 不能使得程序相互使用彼此的数据。

为了交换数据，人们开发了 DDE(动态数据交换，Dynamic Data Exchange)。它定义了一种格式和一种机制，用于从一个程序向另一个程序传输数据，但是它却不太灵活。OLE1.0(对象链接和嵌入，Object Linking and Embedding)也可以实现此功能，它使得类似 Word 文档的文档可以实际包含来自另一个程序(比如 Excel)的文档。这有点类似于组件，但是 OLE1.0 实际上并不是通用的组件标准。

Microsoft 使用 COM(组件对象模型，Component Object Model)标准定义了它第一个真正的组件标准，COM 标准于 20 世纪 90 年代中期在 Windows 中实现。OLE 版本 2 以及许多后继技术在 COM 的基础上构建。DCOM(Distributed COM)引入了一种功能，使得 COM 组件可以与网络交互。COM+添加了服务，组件可以在服务上调用，以保证在多层环境中还可以具有很好的性能。

COM 性能非常好，但是却难于掌握(尤其是在 C++中使用时)和使用。COM 需要插入到 Windows 系统注册表中的组件的信息，这使得安装更加复杂，也难于清除组件。

COM 最初用于 C/C++；对其改良之后也可以用于 Visual Basic(“Automation”)，实际上它在 Visual Basic 中使用性能很好，但是 C/C++程序员更难于使组件与 Visual Basic 兼容(例如，您还是不能从用另一种语言定义的类中继承)。

另外，随着用户不断地安装 Microsoft 和其他公司的多种版本的 DLL 和 COM 组件，问题也出现了，因为程序使用了相同的共享 DLL 的不同版本。一种程序可以很容易地安装由另一种程序使用的 DLL 的不同版本，这就使得原来的程序无法运行(这就是“DLL Hell”)。我们需要

检测安装在系统上的不同 DLL 的所有信息，这种负担使得组件难于升级和维护。

而有了.NET 编程模型，就可以定义一种新的标准来解决这种问题。

21.2 自我说明的.NET 装配件

.NET 装配件与其之前的技术的最重要的区别是，它们是完全自我说明的；说明文档包含在装配件中，这样系统或调用程序就不必查询注册表中的信息，或者查询其他位置中关于包含在装配件中的对象的信息。

在.NET 装配件中的自我说明文档不仅仅包含对象和方法的名称，以及参数的数据类型；它还包含对象的版本信息(比如是 Shapes1.0 还是 Shapes1.1，或 Shapes2.0)，并控制被包含的对象的安全。所有这些信息都包含在装配件自身中，因此就无需到别处查找信息。这就使得安装.NET 组件要比安装之前的 Windows 技术更容易，更方便。事实上，安装过程就像将装配件复制到目标系统的磁盘上那样简单。

1. .NET 装配件和.NET Framework 类库

所有的.NET 程序，包括所有的 C#程序，都广泛地使用了.NET Framework 类库。当使用 using System 指令从 System 名称空间中调用方法时都会用到这些类 —— 所有的 System 名称空间(System.Data，System.Drawing 等)都属于.NET Framework 类库。

在此类库中的每一个类都是自我说明的装配件的一部分。例如，绘图类就包含在 System.Drawing.dll 装配件中。如果要在 Visual Studio.NET 中添加对此装配件的引用，在为程序构建装配件时，编译器就会将引用包含在该装配件中。在运行时，CLR 在程序的装配件中读取元数据，以判断程序是否还需要其他装配件，然后定位并加载这些装配件，便于您的程序使用。您的程序所引用的装配件可能会引用其他装配件，因此带有简单的 using 指令的简单程序都可以同时引用几种不同的装配件。在每一种装配件中的自我说明文档可以在您不知晓的情况下跟踪所有这些引用。

注意：

在此我们要澄清一些概念，以免以后引起混淆。名称空间与装配件并不总是一对一地匹配。例如，System.Data.dll 装配件实际上在 System.Data 和 System.Xml 名称空间中实现一些例程，而 System.Xml 名称空间中的其他例程却在 System.Xml.dll 装配件中实现。在您的程序中，当规定 using 指令时，您就可以引用一个名称空间；在 Visual Studio.NET 工程中的引用规定了实际使用的装配件。

2. 跨语言的程序设计

装配件和.NET 的一个附加的优点是，它们可以实现跨语言编程，这是因为可以从任何.NET 语言中调用组件，而不管最初在什么类型的语言中编程。

.NET 提供了大量的功能，以实现跨语言编程技术：

- Common Language Runtime(公共语言运行时，CLR)，它可以管理所有.NET 装配件的

执行。

- MSIL(微软中间语言，Microsoft Intermediate Language)，由所有的.NET 语言编译器生成。这是编译器生成的二进制代码的公共标准，是 CLR 所执行的语言。CLR 也定义存储装配件的元数据的格式，这表示不管使用什么类型的语言编写，所有的装配件都共享存储元数据的公共格式。

- Common Language Specification(公共语言规范，CLS)，其作用是，在所有与 CLS 兼容的语言(C#，Visual Basic，C++或任何其他的.NET 语言)中编写的程序都共享组件，这些组件具有跨各种语言界限的完全继承性。CLS 定义了语言必须支持的功能，以便于支持与其他.NTE 语言的交互性。还有可能使用 CLS 中没有的功能，但是这样就无法保证其他语言也会支持这些功能。

- Common Type System(公共类型系统，CTS)，它定义由所有的.NET 语言使用的基本类型，以及定义我们自己的类的规则。这可以防止以不兼容方式实现语言(比如 string 类型)。

按照 CLS 规范，您就可以用 C#语言编写组件，用其他.NET 语言(比如 Visual Basci.NET)编写的程序就可以使用包含这个组件的装配件，因为 C#和 VB.NET 组件都可以由 CLR 执行。同样，C#程序也可以使用用 VB.NET，Managed C++等语言编写的组件。在装配件级别上，.NET 语言使用的所有类、对象和数据类型都可以共享，因此您可以继承类并充分利用组件，而不管编写它们的语言类别。

3. 用于 COM 和其他旧代码的自我说明装配件

.NET Framework 还允许用 COM 和其他“旧”技术编写的组件或库可以用 C#和其他.NET 语言编写。

通过自我说明装配件，这种机制也可以发挥作用；所发生的情况是，为继承代码创建了包装器装配件(wrapper assembly)，这些旧代码允许装配件在.NET 运行时中自我说明，将 COM 数据类型转换为.NET 数据类型并允许反复调用.NET 语言到旧代码，或者进行相反的操作。当添加对 COM 组件的引用(通过使用 Add Reference 对话框中的 COM 标签)时，Visual Studio.NET 就自动创建包装器装配件。

图 21-1 显示了所使用的一个包装器(也称为运行时可调用包装)。.NET 客户机装配件发出的调用到达包装器，再到达 COM 组件；站在.NET 装配件的角度来看，包装器其实就是组件。

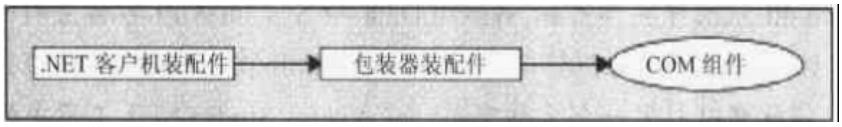


图 21-1

21.3 装配件的结构

装配件的组成部分为.NET 程序提供了一种手段，可以使程序发现其他程序，并解决了程序和组件之间的引用问题。

装配件包含程序或类库的可执行代码，以及元数据(说明其他数据的数据)，这使得其他程序可以查找在装配件中定义的对象的类、方法和属性。元数据以两种方式发挥作用：作为内容的表，说明在装配件中包含的内容；作为参考文献，说明装配件之外对数据的引用。下面我们详细介绍元数据。

单一文件的.NET 装配件具有图 21-2 所示的一般格式：

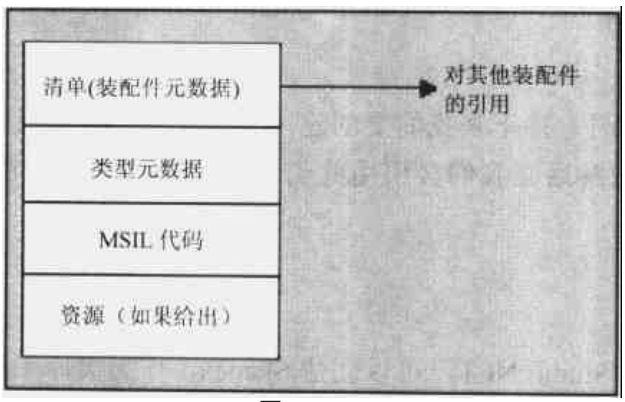


图 21-2

每一个装配件包含一个清单，用于说明装配件的内容(比如装载货物的清单)。清单也称为装配件元数据，因为它描述了装配件本身 ——装配件包含什么模块，装配件引用了其他什么装配件，等。在本章后面，当我们介绍所创建的装配件的内容时，我们会详细介绍清单的内容。之前的组件技术，比如 COM，没有类似于清单的内置概念；清单是内置于.NET 装配件的自我说明文档的核心内容。

当运行程序以便在系统中(比如 System.Console 库，它包含用于输出“Hello, World!”的 WriteLine()方法)引用其他装配件时，.NET 时就在程序的装配件中使用清单。

清单之后就是类型元数据 —— 说明类、属性和方法等，类型元数据包含在装配件中，随它们的参数和返回值的数据类型放置在一起。其后是每一种类型的实际的二进制代码，这些代码作为与机器无关的 Microsoft Intermediate Language(MSIL)代码存储。最后是组成装配件部分的其他资源。这些资源是程序的不可执行部分(在.Resources 文件中规定)，比如图像、图标或消息文件。

虽然装配件通常包含一个文件，但是它也可以包含几种文件，如图 21-3 所示。

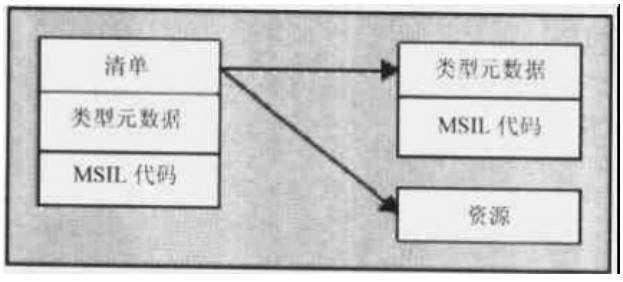


图 21-3

从.NET 运行时的角度来看，具有多文件的装配件是单一逻辑的 DLL 或 EXE，用于包含多个文件。仅有一个文件包含清单。清单指向组成多个文件的装配件的其他文件。包含可执行代码的文件也称为模块；模块包含类型元数据和 MSIL 代码。还可以有不包含可执行代码的资源文件。

多文件装配件通常用于某些高级应用程序。只有当实际运行模块或资源或使用模块或资源时，才加载模块或资源，因此将很少使用的模块和资源存储在单独的文件中可以节省下载时间和内存空间。例如，面向全球发布的应用程序可能具有用不同语言编写的模块或资源；将它们分开保存，就可以在使用时仅仅将语言模块加载到内存中即可。

21.4 创建装配件

我们已经介绍了装配件，接下来我们要创建一个装配件，并介绍其属性。首先，用 C#语言创建一个简单类库，在本章后面我们会引用此类库。我们制作在本章第一节中构思的 Shapes 组件的简单版本。

21.4.1 创建类库

如果使用的是 Visual Studio.NET，可以创建 Shapes，作为类库工程，如图 21-4 所示。

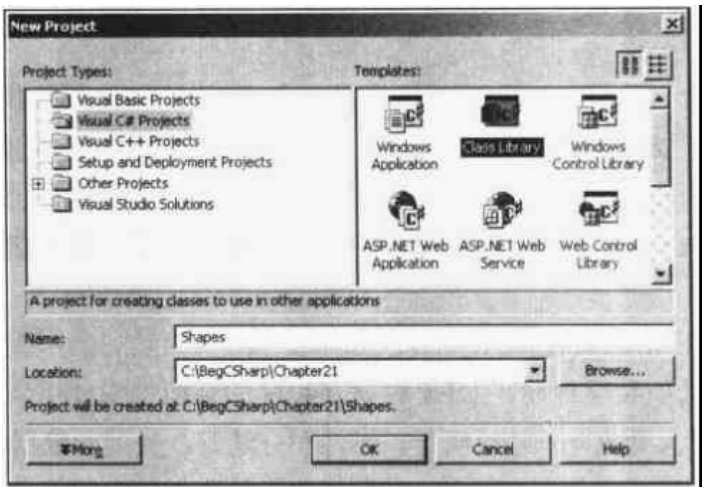


图　21-4

将默认创建的 class1.cs 文件重命名为 shapes.cs，并输入下面所示的源代码。从 shapes.cs 构建的二进制文件就是 shapes.dll，我们将其作为装配件范例。

21.4.2 Shapes 源代码

shapes.dll 装配件的 C#源代码如下：

```
namespace Shapes
{
    public class Circle
    {
        double Radius;

        public Circle()
        {
            Radius = 0;
```

```
        }

        public Circle(double givenRadius)
        {
            Radius = givenRadius;
        }

        public double Area()
        {
            // area = pi r squared
            return System.Math.PI * (Radius * Radius);
        }
    }

    public class Triangle
    {
        double Base;
        double Height;

        public Triangle()
        {
            Base = 0;
            Height = 0;
        }

        public Triangle(double givenBase, double givenHeight)
          {
            Base = givenBase;
            Height = givenHeight;
          }

        public double Area()
        {
            return 0.5F * Base * Height;   // area = 1/2 base * height
        }
    }
}
```

这些代码非常简单，显然并不能仅仅使用一组图形完全实现所有的目标，但是这些代码却可以达到练习的目的。惟一复杂的一点是，Circle 和 Triangle 类各自都有两个构造函数，一个不带参数，另一个带有初始化实例变量的参数。稍后当我们介绍装配件时会再次学习此内容。

21.4.3 编译类库

如果您使用的是 Visual Studio.NET，则选择 Build | Build (Ctrl-Shift-B)就可以构建 Shapes

工程。

如果在命令行使用 C#编译器，则可以使用下面的命令编译 shapes.cs：

```
csc /target:library shapes.cs
```

这就会在当前的目录中创建 shapes.dll。/target:library 选项指示编译器将结果装配件创建为 DLL，而不是可执行文件(.exe)。

注意：

如果从命令行执行 csc 有麻烦，原因可能是您未能正确地设置 PATH 环境变量。设置 PATH 环境变量可以参见附录 A。

21.5 查看装配件的内容

下面我们使用在 shapes.dll 装配件中的自我说明文档查看刚才创建的装配件的内容。我们用来查看装配件内容的工具是 ildasm，即.NET Framework Intermediate Language DisASSeMbler 工具。此工具可以非常方便地查看并理解装配件的内部结构。但是它并不是在日常的 C#程序开发工作中经常使用的工具，因此您就不必记住使用此工具的细节。

21.5.1 将 ILDASM 作为外部工具添加到 VS 中

ildasm 是可以添加到 Visual Studio.NET 环境中的外部工具。要将其添加到 VS 中，可以进入 Visual Studio.NET 中的 Tools | External Tools...菜单。在此对话框中单击 Add 按钮。可以在 Menu Contents 列表中看到[New Tool 1]，以及 Title 项目框；在 Title 项目框中输入 ildasm，然后单击 Command 项目框右侧的浏览按钮(...)。在出现的 Open 对话框中，导航到下面的路径：

```
C:\Program Files\Microsoft.Net\FrameworkSDK\Bin
```

在 Bin 目录中单击 ildasm.exe，然后单击 Open。Ildsam 就会出现在 Menu Contents 列表中，如图 21-5 所示。

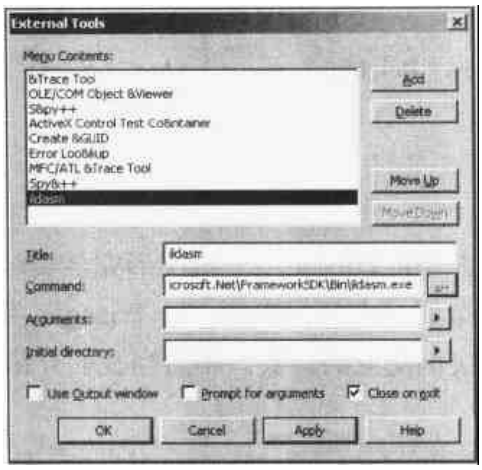


图　21-5

单击 OK 按钮，ildasm 就会作为 Visual Studio.NET 的 Tools 菜单的选项出现。

21.5.2 使用 ILDASM

现在已经在 Visual Studio.NET 中添加了 ildasm，通过选择 Tools | ildasm 就可以打开它。ildasm 就会出现在单独的窗口中，如图 21-6 所示。

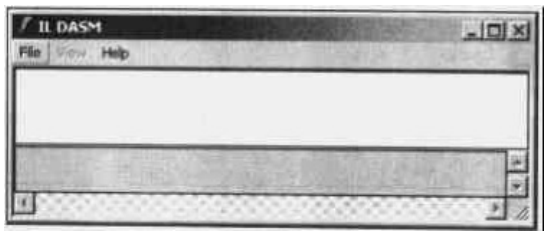


图 21-6

使用 File | Open 菜单可以找到包含 shapes.dll 的目录并打开此目录。如果您使用了命令行编译器编译程序，则 shapes.dll 就会出现在与 shapes.cs 源文件相同的目录中。如果您使用了 Visual Studio.NET，则 shapes.dll 就位于工程的 bin 子目录中，如果您构建了默认的配置，如图 21-7 的 Open 对话框所示，则它就位于 Debug 子目录中。

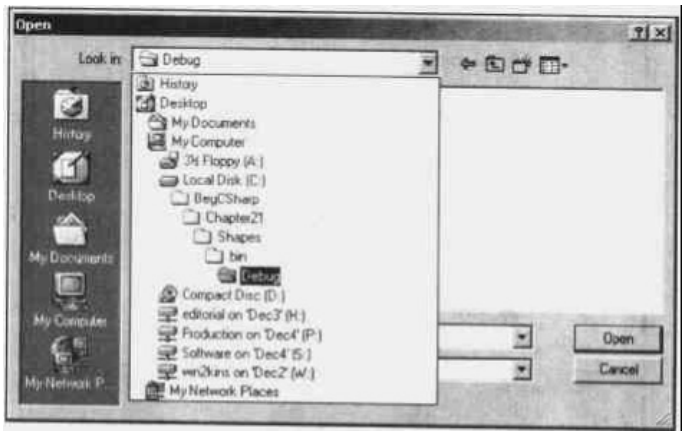


图 21-7

定位了 shapes.dll 之后，单击 Open 按钮。然后在 ildasm 主窗口中就会出现装配件的视图，如图 21-8 所示。

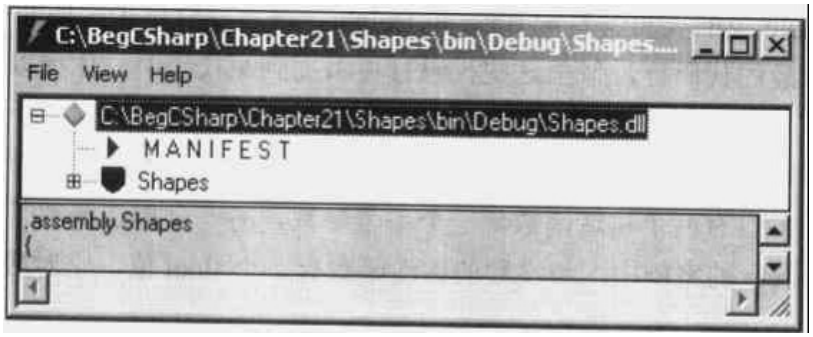


图 21-8

在图中我们可以看到清单和 Shapes 类。清单是装配件信息；稍后我们再对其进行介绍。Shapes 盾形图标表示 Shapes 名称空间；它来源于此装配件的元数据。可以通过单击 Shapes 盾

形图标前面的加号扩展树视图。我们就可以看到在源文件中定义的两个类，即 Circle 和 Triangle，如图 21-9 所示。

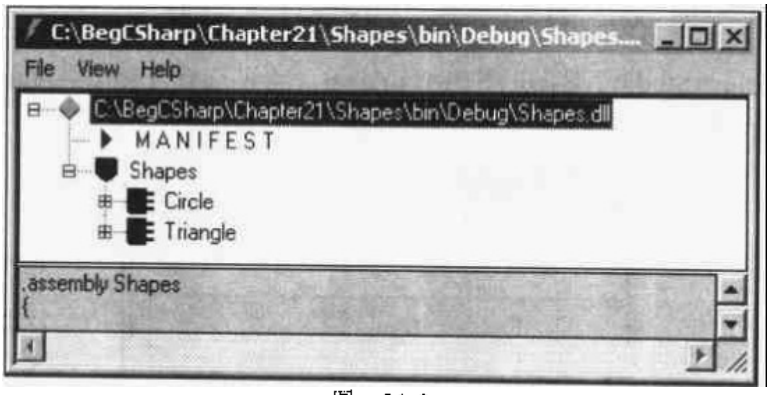


图 21-9

扩展每一种类的视图，为此分别单击 Circle 和 Triangle 前面的加号，如图 21-10 所示。

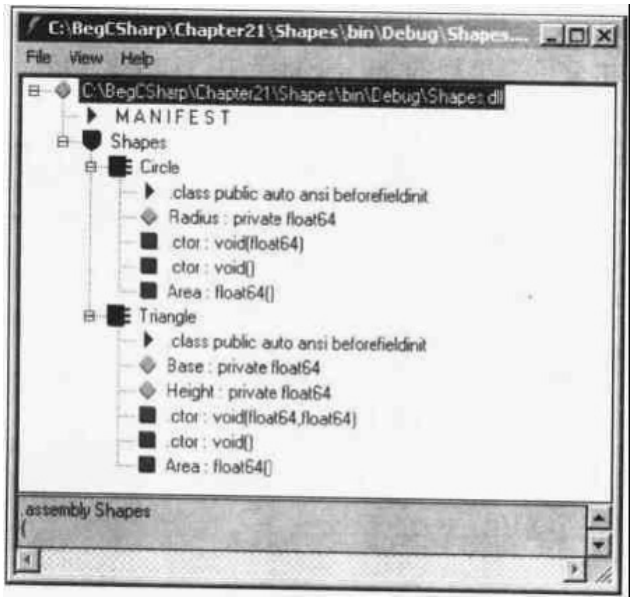


图 21-10

现在我们就可以看到对应于所创建的源代码的内容。可以看到 Circle 类的类实例变量 radius，以及 Triangle 类的类实例变量 Base 和 Height，以及同时用于这所两个类的 Area 方法。我们还可以看到有趣的代码行，似乎这些代码行被添加到源代码中，比如以.ctor 和.class 开始的行。

如果我们再仔细的看一下，就会发现.ctor 行实际上对应于为 Circle 和 Triangle 定义的构造函数。对于这二者各自有两个构造函数，一个不带参数，另一个构造函数带有的参数可以实例化实例变量。在 Circle 的案例中，带参数的构造函数有一个 float 值，它对应的行如下：

```
.ctor void(float32)
```

在装配件的 ildasm 视图中，对于 Triangle 类而言，参数化的构造函数带有两个 float 参数。此构造函数对应的行如下：

```
.ctor(float32, float32)
```

以句号开始的行是 MSIL 中的指令，是 C#语言代码，编译后用于在.NET 时中运行。.ctor 指令是 MSIL 指令，生成一个类构造函数。不需要为了查看装配件的内容而彻底理解 MSIL；我们仅仅在遇到它们时介绍一些感兴趣的方面即可。

我们还在窗口的顶部可以看到标记为 MANIFEST 的行；接下来我们稍微介绍一下清单的内容。

21.5.3 清单

清单描述在装配件中的每一个文件或模块(我们曾经说过，虽然装配件通常只包含一个文件，但是它们也可以包含多个文件)。更重要的是，它还描述此装配件引用了什么样的外部装配件。例如，如果程序使用 System.Data.dll，它就会在清单中被反射，这更易于记录装配件之间的依赖关系，也更易于部署程序和验证程序的安装是否正确。清单还记录装配件的版本号码，更易于升级到程序的新版本。下面我们就介绍一下所创建的装配件的清单。

在 shapes.dll 的 ildasm 列表的顶部双击标记为 MANIFEST 的行。这就会打开新的窗口，如图 21-11 所示，在该窗口中描述了清单的细节。



图 21-11

shapes.dll 清单包含两个.assembly 指令和一个.module 指令。无需关心.assembly 块的内容；即目前要忽略在花括号{}中的内容。第一行是：

```
.assembly extern mscorlib
```

这是对 mscorlib.dll 外部装配件的引用，在 mscorlib.dll 中定义了.NET Framework 中的大多数 System 基类。使用 System 类的所有 C#和.NET 程序都必须使用这种外部引用。.assembly Shapes 行是对图形装配件本身的声明。其后是对 shapes.dll 文件的.module 声明。我们的装配件只有一个文件，因此对该文件就只有一个.module 声明。

我们还可以看一下如果向源文件添加附加引用会发生什么情况。假如我们希望使用 System.Drawing 名称空间的一些方法绘制图形。关闭 ildasm，返回到 Visual Studio.NET 中的 shapes 工程。按如下方式修改 shapes.cs。首先添加引用 System.Drawing 名称空间的 using 指令：

```
namespace Shapes
{
    using System.Drawing;

    public class Circle
    {
```

然后，在 Area()方法之后添加 Circle 的 Draw()方法：

```
public void Draw()
{
    Pen p = new Pen(Color.Red);
}
```

这些代码还不能完成绘制图形的任务，但是这是完成绘制图形任务的开端。

接下来我们需要把对 System.Drawing.dll 装配件的引用添加到工程中。如果不添加引用，就会看到下面的错误消息：

error CS0234: The type or namespace name 'Drawing' does not exist in the class or namespace 'System' (are you missing an assembly reference?)

在 VS 添加引用，可以在菜单中选择 Project | Add Reference...，会出现如图 21-12 所示的 Add Reference 对话框。.NET 标签显示可以获取的.NET 系统装配件；在列表中向下翻屏，选择 System.Drawing.dll 装配件。然后单击 Select 按钮。

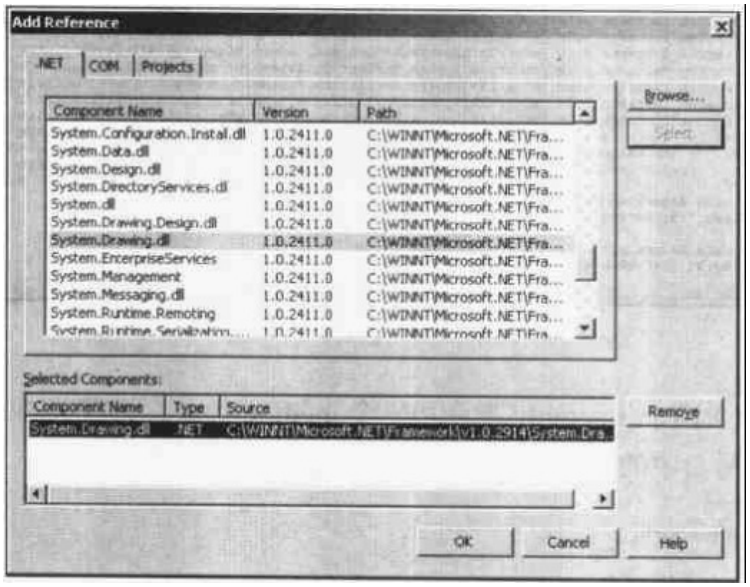


图　21-12

单击 OK 按钮就可以添加引用。这可以在 shapes.dll 文件中添加对 System.Drawing 引用。保存所改动的源文件，关闭 ildasm(否则就会得到一个编译器错误，因为 ildasm 打开了 shapes.dll)，然后根据上面的修改重新编译 shapes.cs。

接下来重新启动 ildasm，打开 shapes.dll。现在可以在 Circles 对象之下看到 Draw()方法，如图 21-13 所示。



图 21-13

再次双击 MANIFEST，可以看到所作出的修改。现在就有了对 System.Drawing 装配件的外部装配件引用，如图 21-14 所示。

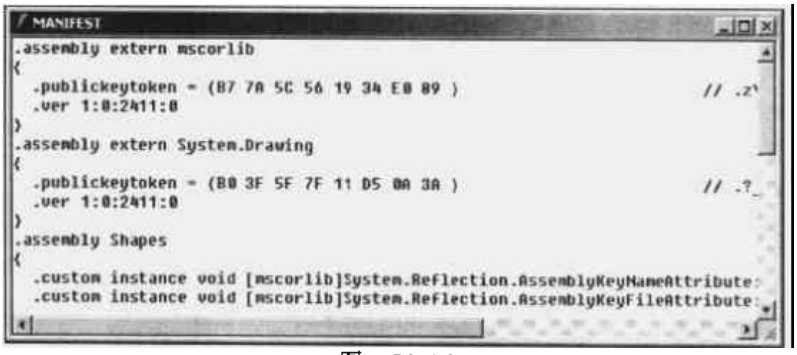


图 21-14

shapes 装配件通知系统，无论何时引用 shapes 时，它都要求 System.Drawing 装配件。稍后我们要介绍其他程序如何利用 Shapes 装配件，但是在看过上面窗口之后，您很可能要问，在前面我们提到在.assembly 指令中被忽略的内容到底是什么。

21.6 装配件属性

除了外部装配件引用之外，装配件清单还包含其他专用于装配件的信息。这些被称为装配件属性。

21.6.1 AssemblyInfo.cs

如果在 Visual Studio.NET 中构建类库工程，则您可能会注意到 Visual Studio.NET 创建了第二个 C#源文件作为工程的一部分。此文件被称为 AssemblyInfo.cs，用于在清单中设置装配件的属性，比如装配件版本号、名称等。接下来我们介绍此文件的内容(为了节省空间，我们已经删除了一些注释)：

```
using System.Reflection;
using System.Runtime.CompilerServices;
```

```
//
// General Information about an assembly is controlled through the following
// set of attributes. Change these attribute values to modify the information
// associated with an assembly.
//
[assembly: AssemblyTitle("")]
[assembly: AssemblyDescription("")]
[assembly: AssemblyConfiguration("")]
[assembly: AssemblyCompany("")]
[assembly: AssemblyProduct("")]
[assembly: AssemblyCopyright("")]
[assembly: AssemblyTrademark("")]
[assembly: AssemblyCulture("")]

//
// Version information for an assembly consists of the following four values:
//
//      Major Version
//      Minor Version
//      Build Number
//      Revision
//
// You can specify all the values or you can default the Revision and
// Build Numbers by using the '*' as shown below:

[assembly: AssemblyVersion("1.0.*")]

[assembly: AssemblyDelaySign(false)]
[assembly: AssemblyKeyFile("")]
[assembly: AssemblyKeyName("")]
```

在方括号中的每一条语句(比如[assembly: AssemblyXxxxx])都是属性，这些是 C#中的特殊语法，在第 22 章我们再深入介绍它们。现在，知道每一条语句设置装配件的某一个属性即可。在每一个属性开始处的单词 assembly:告诉系统，下面的指令用于装配件本身，而不是用于类、方法或程序的其他部分。

VS 提供包含装配件属性的文件，该文件用作模板，您可以在其中填充希望装配件具有的属性。您也可以在使用 csc 编译的源文件中使用这些属性。大多数属性，比如标题、公司等，都是纯粹的信息，可以使用与组件相关的描述性值来填充，比如：

```
[assembly: AssemblyTitle("MyCompany Shapes Class Library")]
[assembly: AssemblyDescription("Classes for Manipulation of Shapes")]
[assembly: AssemblyConfiguration("Enterprise Version")]
[assembly: AssemblyCompany("MyCompany, Inc.")]
[assembly: AssemblyProduct("Shapes")]
```

```
[assembly: AssemblyCopyright("Copyright 2001, MyCompany, Inc.")]
[assembly: AssemblyTrademark("Shapes is a trademark of MyCompany, Inc.")]
```

AssemblyCulture 属性指用于此装配件的国家语言，如果作出了规定，则它就是后面带有国际标准的特殊简缩符。如需更多的信息，可以参见 System.Globalization 名称空间和在.NET Framework 在线文档中的文化主题。除非您要发布组件的不同语言版本，否则您就无需设置文化属性；如果您确实需要设置文化属性，则.NET 运行时会自动搜索与当前文化匹配的装配件的版本。例如，在法国，您要显示法语消息(使用法语资源和/或代码)，在英国，您要显示英语消息(使用英语资源和/或代码)。使用正确的文化属性标记合适的装配件即可。

这些属性位于 AssemblyInfo.cs 文件中，因此它们也就被包括在装配件清单中。例如，如果再次在清单中查找 shapes.dll，就可以在.assembly Shapes 部分中看到下面的代码行：

```
.custom instance void [mscorlib]System.Reflection.AssemblyTitleAttribute::
.ctor(string) = ( 01 00 00 00 00 )
```

这些代码表示装配件包含一个 AssemblyTitle 属性。如果从命令行编译 shapes.cs，却没有 AssemblyInfo.cs 文件，则这些代码就不会出现在清单中。

21.6.2 版本号

.NET 装配件的版本由 4 部分组成，如表 21-1 所示。

表 21-1

Major version	Minor version	Build Number	Revision

如果您使用过用户软件，则就会非常熟习前两部分 ——即主版本号和次版本号，如 Shapes 版本 1.0(1 是主版本号，0 是次版本号)。

剩下的两部分更细致地划分了版本。构建号表示构建装配件的号码；当每次重新构建装配件时，构建号都会变化。

修订版本号划分得更细致，用于装配件的补丁程序，除了此装配件带有补丁程序之外，此装配件与原来的装配件没有什么区别。

21.6.3 版本属性

在 ildasm 中可以看到装配件的版本属性。看一下 shapes.dll 文件的清单，可以发现每一个被引用的装配件都有一个.ver 指令，它放置在.assembly 块中，用于该装配件。

您还可以注意到外部装配件引用(比如对 System.Xxxx 装配件和 CLR(mscorlib)的引用)的版本信息全部相同。这是因为.NET 运行时使用装配件版本号用于兼容性检查。当我们学习版本兼容性问题时再详细讨论这个问题。

如果您使用了命令行 C#编译器(csc)，则 shapes 装配件的.ver 指令全部是 0，如：

```
.ver 0:0:0:0
```

这是因为我们还没有给 shapes 装配件赋予版本号。如果使用的是 Visual Studio.NET，则看到的版本号如下：

```
.ver 1:0:486:7484
```

这是因为 VS 自动添加了一些版本信息。

21.6.4 AssemblyVersion 属性

在 VS 创建的 AssemblyInfo.cs 文件内部，可以使用 AssemblyVersion 属性来设置版本号：

```
[assembly: AssemblyVersion("1.0.*")]
```

这个星号 (*)是什么？AssemblyVersion 属性允许在版本号的最后两部分中规定星号(*)。这就会指示 VS 自动设置构建和修订版本号。您还可以仅仅为修订版本号规定星号(如 1.1.1.*)，但是不能为主版本和次版本号使用星号(不允许有 1.*的形式)。如果使用 ildasm 查看装配件版本号，您就可以看到 VS 实际设置的版本号，如下所示：

```
.ver 1:0:585:24784
```

如果改变了类中的一些代码并重新构建装配件，您就会看到号码会自动改变，如下所示：

```
.ver 1:0:585:25005
```

您也可以直接通过规定具体的号码而设置版本号，而不是使用星号设置版本号：

```
[assembly: AssemblyVersion("1.0.1.2")]
```

这就会使得 VS 生成一个装配件，其版本号为所规定的主版本号、次版本号、构建号(build number)和修订版本号。

如果您开发的程序仅仅为了自用，则无需设置或关心版本号。但是，如果您向其他终端用户发布软件，则如果您在产品中添加了重要新功能，就要修改主版本号和次版本号：1.0 用于产品的第一版，1.1 用于发布稍作修改的版本，2.0 用于做了较大修改的版本，等等。将 VS 自动设置构建/修订版本号非常好，甚至在大多数时间您都无需知道这些号码。

使用 ildasm 检查完整的版本号非常方便。例如，如果终端用户报告了一个程序错误，则您就可以将安装在您的计算机上的装配件版本与安装在终端用户机器上的版本对比。然后您就可以知道用户使用的是什么版本；使用修订版本号和构建号，您甚至都可以知道今天早上构建的装配件与昨天构建的装配件的区别。

21.6.5 版本兼容性

.NET 运行时在加载装配件判断版本兼容性时会检查版本号，这仅适用于共享装配件，稍后再介绍这些内容。但是，在此要介绍使用版本号检查版本的方法。

装配件的清单包含装配件的版本号以及被引用的外部装配件的版本号。当.NTE 运行时加载一个被引用的装配件时，它在该装配件的清单中检查其版本，并将此版本与存储在引用表中的版本比较，确认这些版本是兼容的。

如果装配件的主版本号或次版本号不同，则就认为它们是不兼容的，因此也就不能加载被引用的装配件——例如，Shapes 版本 1.1 与引用 Shapes 版本 1.0，1.2 或 2.0 的程序就不兼容。

如果在一个系统上既有使用 Shapes1.0 的 A 程序，又有使用 Shapes1.1 的 B 程序，该怎么办呢？.NET 运行时可以处理这个问题；.NET 运行时具有一种并列执行的功能，可以确保在同一台计算机上同时安装 Shapes1.0 和 Shapes1.1，并分别将这两种装配件提供给需要它们的程序。

版本号的后两部分更细致地判断版本冲突问题。构建号表示装配件的构建；当每次重新构建装配件时通常都要改变构建号。如果两种装配件具有相同的主版本号和次版本号，但是其构建号却不同，则这两种装配件可能兼容，也可能不兼容；运行时可以假设它们是兼容的，这样就可以加载装配件；如果对装配件做了导致不兼容的修改，最后还需开发人员来修改主版本和次版本。

在前面的范例中，您可能注意到了，装配件 mscorlib 和 System.Drawing 的版本号都是 1.0，其构建号都是 2411，这是作者在编程时系统上的.NET System 装配件的构建号。如果将其中一种库的构建号更改为 2412，甚至是 9999，则.NET 就会使用新构建；但是这种构建号却不能保证是兼容的。

修订版本号又深入一层来判断版本冲突问题，使得您可以规定某构建号非常具体的补丁程序或修订程序。1.0.2411.0 和 1.0.2411.1 被认为是完全兼容的。在装配件并存的情况下，如果两种共存的装配件的主版本和次版本号匹配，但是它们的构建号和/或修订版本号却不同，则系统就会执行新装配件(换句话来说，就是执行具有较高构建/修订版本号的装配件)。

21.7 调用装配件

现在我们看一下当程序引用 Shapes 时，会发生的情况。为 shapes 创建一个简单的客户机程序，将其命名为 shapeuser.cs：

```
namespace ShapeUser
{
    using System;
    using Shapes;

    public class ShapeUser
    {
        public static void Main()
        {
            Circle c = new Circle(1.0F);

            Console.WriteLine("Area of Circle(1.0) is {0}", c.Area());
        }
    }
}
```

使用命令行编译 ShapeUser

如果您正在使用命令行 C#编译器，就可以在与 shapes.cs 和 shapes.dll 相同的目录中创建 shapeuser.cs，然后使用下面的目录编译它：

```
csc /reference:shapes.dll shapeuser.cs
```

这就会在当前的目录中创建 shapeuser.exe，它可以引用 shapes.dll 装配件。

在 VS 中编译 ShapeUser

如果正在使用 VS，可以创建一个新控制台应用程序工程，将其命名为 ShapeUser。将源文件 Class1.cs 重命名为 shapeuser.cs，并输入上面的代码。此工程需要一个对要编译的 Shapes 工程的引用，因此从 Project 菜单中选择 Add Reference...(或者在 Solution Explorer 中的 References 上右击，并选择 Add Reference...)，就会出现 Add Reference 对话框，如图 21-15 所示。

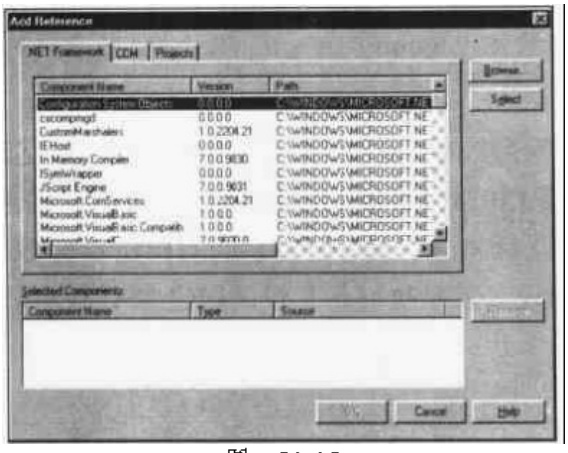


图　21-15

在 Project 标签上单击，然后单击 Browse 按钮(在窗口的右上角)。在 Select Components 对话框中，在 Shapes 工程目录中找到 shapes.dll 文件，单击 Open 按钮。在 Selected Components 列表中就会出现 shapes.dll 文件，如图 21-16 所示。

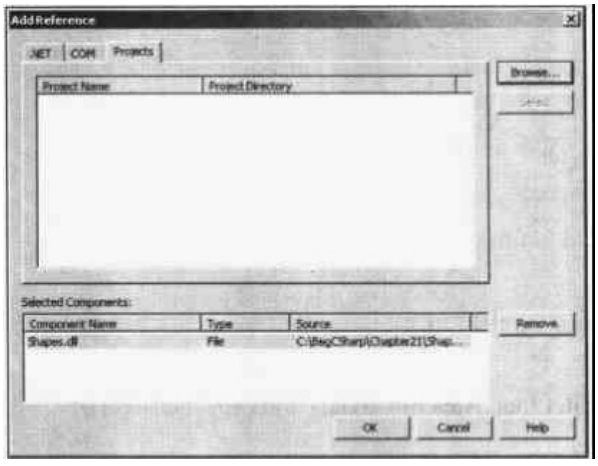


图　21-16

单击 OK 按钮，就会在 ShapesUser 工程引用列表中添加 shapes，如图 21-17 所示。



图 21-17

在 ShapeUser bin\Debug 目录中已经添加了 shapes.dll 的私有副本。在下一节我们会详细讨论它，在此我们继续构建 ShapeUser 应用程序。

使用 ILDASM 进行检查

构建了 ShapeUser.exe 之后，可以使用 ildasm 检查其内容，如图 21-18 所示。



图 21-18

注意即使是一个单独的程序，它也具有如同类库所有的元数据。在 MANIFEST 上双击，查看 shapes 组件的外部引用，如图 21-19 所示。

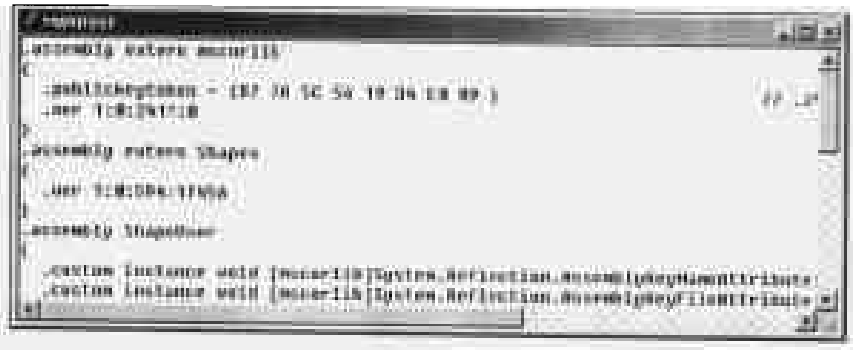


图 21-19

最后，执行 shapeuser.exe，其结果如图 21-20 所示。



图 21-20

21.8 私有和共享装配件

到目前为止，我们学习了私有装配件(private assembly) ——即作为单一应用程序的组成部分部署的装配件。但是.NET Framework 还提供了特殊的工具，以提供共享装配件(shared assembly)，共享装配件可以同时被多个程序使用。

21.8.1 私有装配件

默认情况下，装配件对于工程而言是私有的。私有装配件必须与应用程序位于相同的目录中。

在此的 shapes.dll 装配件是私有的；为了在 ShapeUser 工程中引用它，我们既可以在同一个目录中构建两种工程，也可以在 VS 中添加引用时从 ShapeUser 工程浏览 Shapes 开发目录 ——在这种情况下，VS 生成了 shapes.dll 的私有副本，并将其放置在 ShapeUser 目录中。

复制 shapes.dll 装配件可以保证，即使由于连续的开发工作而无法使用初始的 shapes.dll，也可以运行 ShapeUser。但是复制每一个被引用的 DLL 会导致广泛使用的组件的效率降低，因此.NET Framework 提供了共享装配件。

21.8.2 共享装配件

共享装配件可以用于系统上的所有程序。由于所有的共享装配件都存储在称为 Global Assembly Cache(全局装配件高速缓冲存储器，GAC)的特殊.NET 系统目录中，所以程序无需知道共享装配件的位置。因为在系统范围都可以知道共享装配件，所以.NET 运行时在共享装配件上强制进行几种额外的检查工作，以保证共享装配件可以用于所有请求它们的程序中，比如安全性和版本兼容问题。

1. 安全性和强名

共享装配件必须证明它不会被使用相同名称和版本的另一种装配件代替，而且它也不会被修改，比如被病毒篡改。这就要求在将共享装配件加载到全局装配件高速缓冲存储器之前，在共享装配件上加上密钥。密钥不仅有助于防止出现安全隐患，而且有助于防止出现由于两种组件使用相同的名称和版本号而导致的简单名称/版本冲突问题。如果密钥不同，即使两种组件的名称相同，也认为它们是不同的组件。装配件名称、版本和密钥的独特组合就是强名(strong name)。

2. 全局装配件高速缓冲存储器

Golbal Assembly Cache 是一个特殊目录，位于 WinNT\assembly 目录。所有共享装配件，包括 Microsoft 提供的.NET Framework System 装配件，都放置在此，并从此处加载。如果使用 Windows Explorer 浏览此目录，则特殊的 Windows 命令解释程序扩展(shell extension)就会显示装配件的属性，包括合并到每一个强名中的密钥。

Windows 命令解释程序扩展(即 shfusion.dll)集成到 Windows Explorer 中，其功能被大大扩展，而不仅仅限于普通的列举文件功能。图 21-21 所示窗口显示了 Windows Explorer 中展示的

GAC，图中列举了装配件的名称、版本、密钥和其他属性。

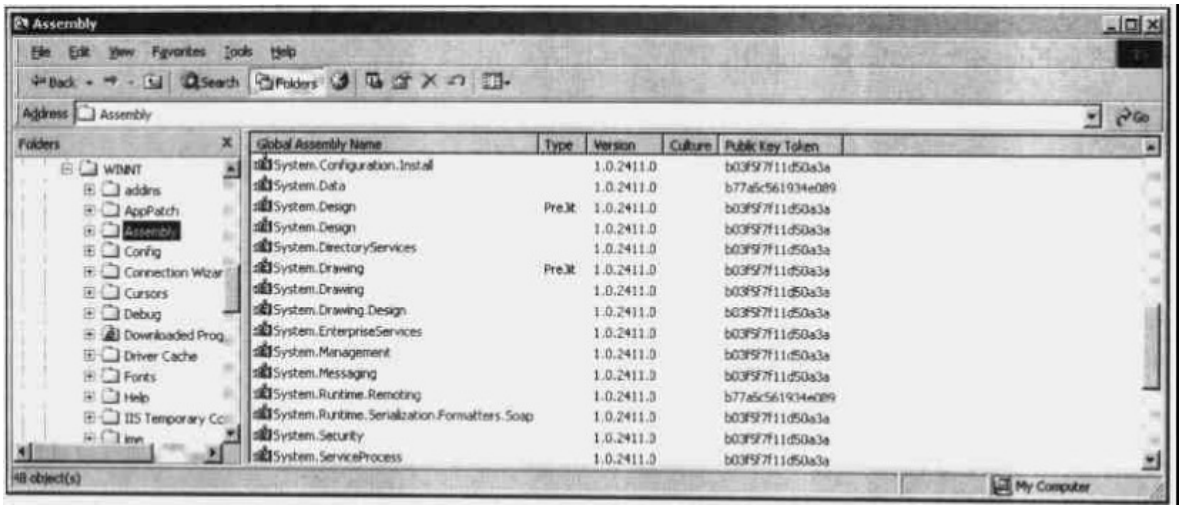


图 21-21

Windows 命令解释程序扩展加强了全局装配件高速缓冲存储器的安全策略，使得装配件可以通过拖放被复制到此目录中，但是复制时必须遵守规则。

您必须具有计算机上的局域管理员特权才可以访问该计算机的全局装配件高速缓冲存储器。

3. 创建共享装配件

为了创建带有强名的共享装配件，您必须生成公共/私有密钥对，此密钥对用于标记装配件。公共/私有密钥加密系统使用的密钥仅为编码消息的发送者所知，而公共密钥却发布给外界。.NET 环境使用这种机制保证被引用的共享装配件确实就是所需要的装配件(例如，不同公司发布的装配件可能具有相同的名称和版本号，或者电脑黑客可以通常创建具有相同名称/版本的装配件而欺骗程序，他们也可能试图篡改现有的装配件)。必须验证对装配件引用中的密钥与(标记的)共享装配件中的密钥是否匹配；如果它们不匹配，则就不能加载共享装配件。

装配件名称、版本和公共密钥的组合必须是独特的；这种组合就是强名。

.NET Framework 提供了一种工具，可以生成称为 sn.exe 的强名(sn 代表 Strong Name)。遗憾的是，强名仅能用于命令提示中，而且您还必须在使用之前设置 PATH 环境变量(具体方法请参见附录 A)。

使用 VS Command Prompt(VS 命令提示)启动一个命令行(可以通过选择 View | Other Windows | Command Window 实现)，或者从 Windows 中打开一个命令窗口。使用下面的命令可以生成一个密钥，给出从装配件中引用的文件名称(带有.snk 扩展名)：

```
sn –k shapes.snk
```

这就可以创建在当前的目录中创建密钥文件 shapes.snk。为了使用此密钥标记装配件，可以在工程的 AssemblyInfo.cs 文件的最后一部分中修改 AssemblyKeyFile 属性：

```
[assembly: AssemblyKeyFile("shapes.snk")]
```

在 VS 中，密钥文件的位置相对于工程中的 obj\debug or obj\release 目录；如果将此文件放置在另一个目录中，比如工程根目录中，就要包括一个相对于此文件的路径，比如[assembly: AssemblyKeyFile(@"..\..\shapes.snk")]：

注意：

如果您使用的是命令行编译器，可以简单地在 shapes.cs 源文件的开端添加 AssemblyKeyFile 代码行，其前是合适的 using 代码行。

```
using System.Reflection;
using System.Runtime.CompilerServices;
[assembly: AssemblyKeyFile("shapes.snk")]
```

现在重新编译 shapes.dll 装配件。如果 VS 不能找到 shapes.snk 文件，就不编译工程，这就会得到一个错误消息，比如：

```
Cryptographic failure while signing assembly 'C:\BegCSharp\Chapter21\Shapes\obj\Debug\Shapes.dll' --
'Error reading key file 'shapes.snk' -- The system cannot find the file specified.
```

这些错误消息告诉我们，VS 正在何处查找密钥文件，我们可以将文件移到该目录中，也可以规定相对于此目录的另一个目录。

当您成功地再次构建了工程之后，就为装配件加了标记。如果查看带有 ildasm 的 shapes.dll 装配件的清单，就可以看到已经生成并嵌入了一个公共密钥，如图 21-22 所示。

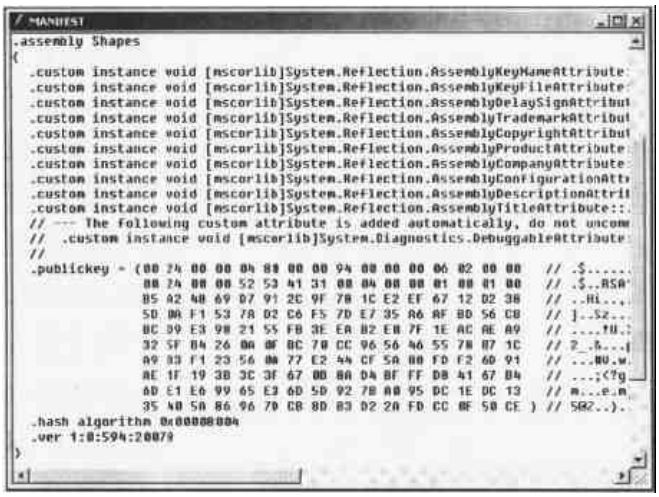


图　21-22

注意，如果我们不重新编译 shapeuser.exe 就试图运行它，则就会得到下面的错误消息，这是因为 shapes.dll 装配件不再匹配(外部引用是对私有装配件的引用，而现在 shapes 是一个共享装配件)：

Unhandled Exception: System.IO.FileLoadException: The located assembly's manifest definition with name 'shapes' does not match the assembly reference. File name: "shapes"

at ShapeUser.ShapeUser.Main()

注意：

如果我们使用 VS 创建 ShapeUser，就不会发生这种错误，因为 VS 创建装配件的私有副本(这种副本不会改变)。

我们需要重新编译 shapeuser.cs，以使用 shapes.dll 标记过的版本在 shapeuser.exe 内更新外部装配件引用。重新编译之后， shapeuser.cs 的作用同原来相同，仍然使用 shapes.dll 的局域副本。

最后一步是将强命名的装配件复制到 Global Assembly Cache 中。仅仅通过将文件拖放到 GAC 文件夹(WinNT\assembly)中就可以完成复制。另外，我们也可以使用称为 gacutil 的.NET 命令行工具(Global Assembly Cache Utility，全局装配件高速缓冲存储器工具)。为了在全局高速缓冲存储器中安装 shapes.dll，可以在 Visual Studio.NET 命令行工具提示中使用带有/i 标志的 gacutil：

```
gacutil /i shapes.dll
```

消息“Assembly successfully added to the cache”表示已成功地安装了装配件。

注意：

对于部署商业应用程序，在全局高速缓冲存储器中添加共享装配件的优选方法是使用 Windows Installer。

为了证实 shapes.dll 已经安装到了高速缓冲存储器中，可以删除在当前的目录中的 shapes.dll 的副本。再运行 shapeuser.exe，就可以看到下面的消息：

```
Area of Circle(1.0) is 3.141596
```

即使缺少了 shapes.dll，它仍然可以运行，这是因为 shapeuser.exe 从 GAC 中加载了图形装配件。为了进一步测试它，可以使用 gacutil 卸载图形装配件：

```
gacutil /u shapes
```

注意卸载选项中没有了.dll 扩展名。

接下来试着运行 shapeuser：

```
C:\shapes>shapeuser
Unhandled Exception: System.IO.FileNotFoundException: File or assembly name shapes, or one of its dependencies, was not found.
```

这表明，原来从 GAC 中加载了 shapes.dll。

21.8.3 搜索装配件

当外部装配件被引用时，.NET 运行时遵循一组预定义的规则，来确定外部装配件的位置。我们已经介绍了如何首先在局域目录中搜索，其次在 GAC 中搜索共享装配件的内容。

对于私有装配件而言，要首先搜索局域目录，然后系统就搜索与装配件具有相同名称的子目录。运行时还要搜索与被请求的装配件具有相同名称的 DLL 或 EXE 文件。对于 Shapes 类而言，组合起来就是要进行下面的搜索：

./shapes.DLL.
./shapes/shapes.dll
./shapes/shapes.exe
./shapes.exe

可以使用配置文件规定附加搜索路径，甚至规定 URL，用于通过 Internet 从远程位置下载装配件。装配件的配置文件是 XML 格式的文件，当搜索装配件时，它规定了应用到.NET 运行时上的规则。配置文件还覆盖用于版本检查的默认行为。配置文件和 XML 的细节稍微复杂一些，其内容超出了本书的介绍范围。如需更详细地学习其内容，可以搜索.NET Framework 在线文档的“装配件配置文件”主题。

21.9 小结

C#程序和类库作为装配件发布，装配件具有许多特性，可以简化 Microsoft .NET Framework 中的组件发布工作。组件用来按二进制方式重用对象。.NET 组件的基本特性就是自我说明特性；这种特性使得.NET 组件与原来的组件具有明显的区别(比如 COM 组件)。

自我说明特性具有许多优点，包括易于安装和与 Common Lauguage Runtime(公共语言运行时，CLR)，以提供跨语言和旧特性以及便于 C#语言开发工作。

我们可以使用.NET 命令行工具和 Visual Studio.NET 创建 C#类库组件，并将其编译进装配件中。然后创建了使用此组件的 C#应用程序，学习如何使用 ildasm 查看装配件的内容。这有助于我们理解装配件的结构。

我们介绍了装配件的各个部分，包括清单、版本号以及其他装配件属性。除了有助于我们理解装配件的结构之外，这还使得我们可以在调试装配件时可以在外部引用中比较版本号。

装配件既可以是专用于应用程序的私有装配件，也可以是共享装配件，共享装配件可以用于系统范围。我们学习了通过创建密钥文件，并标记装配件来创建装配件的强名而创建共享装配件的内容。还学习了检查共享装配件的版本，介绍了存储共享装配件的全局装配件高速缓冲存储器以及用于查看并操作全局装配件高速缓冲存储器的各种工具。

在装配件的兼容性检查、安全性和资源效率等优点与创建和安装装配件的复杂程度之间有一个最优折中方案，在学习了这些内容之后，我们最后简要介绍了如何在.NET 运行时中搜索和解析装配件的引用。

装配件的这些特性使得在.NET 环境中开发应用程序变得更容易了。

21.10 练习

- 在 AssemblyInfo.cs 或 shapes.cs 中修改装配件的版本号和其他装配件属性，并使用 ildasm 在 shapes.dll 中查看修改结果。
- 使用 Square 类和 Circle 以及 Triangle 类创建新的 moreshapes 装配件，然后使用 ildasm 查看其属性。

- 在 shapeuser.exe 中改变装配件引用，以便使用 moreshapes.dll 装配件，然后在 ildasm 查看修改后的属性。
- 为 moreshapes 装配件创建您自己的客户程序，moreshapes 装配件使用了 Square 和 Triangle 对象以及 Circle 对象。使用 ildasm 为此客户机程序检查 moreshapes 装配件。
- 使用/?标志显示 gacutil 的命令选项，然后使用其选项列举系统上所有的全局装配件属性。
- 为 moreshapes 装配件创建强名，并标记该装配件。使用 ildasm 查看结果。试着执行自己的客户程序，然后重新编译此程序，以引用标记的装配件。
- 使用 gacutil 在 Global Assembly Cache 中安装 moreshapes.dll，分别通过在局域目录和/或 GAC 中给出 moreshapes 装配件，和不给出 moreshapes 装配件来试着运行 shapeuser 和您自己的客户程序。

第22章 属 性

本章介绍属性。描述属性的定义及其作用，并给出几个可用于.NET Framework 的属性范例。

我们还将讨论定制属性。定制属性是自己为扩展系统而编写的属性，并提供几个可使用的范例。同时介绍 Intermediate Language Disassembler (ildasm)如何用于掌握现有装配件的属性。

属性是.NET Framework 最有用的特性之一，Microsoft 就常常使用它们。要高效地使用属性，需要花大量的时间研究它们，但这是值得的。在下面的章节中，我们将学习以下内容：

- 调试
- 提供装配件的信息
- 把方法和类标记为过时
- 条件编译
- 数据库访问

我们还会描述如何编写扩展系统的定制属性，并介绍几个定制属性的范例。在本章的最后，您应掌握足够的属性知识，知道如何把它们应用到自己的工程上。

22.1 什么是属性

很难用一句话定义属性，最好通过实践来学习如何使用它们。现在我们把属性定义为可以应用到装配件中代码块上的额外信息，例如类、方法或属性。这些信息可以由使用装配件的其他类访问。

第 21 章讨论了装配件，提到了 AssemblyInfo.cs 文件。要查看这个文件，可在 VS 中创建一个新工程，打开 Solution Explorer (Ctrl+Alt+L)窗口。

得到如图 22-1 所示的结果。

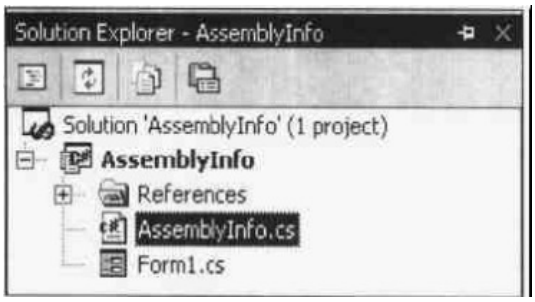


图 22-1

如果双击这个文件，就会看到由 VS 所创建的一些代码，部分代码如下所示：

```
using System.Reflection;
using System.Runtime.CompilerServices;
```

```
//
// General Information about an assembly is controlled through the following
// set of attributes. Change these attribute values to modify
// the information associated with an assembly.
//
[assembly: AssemblyTitle("")]
[assembly: AssemblyDescription("")]
[assembly: AssemblyConfiguration("")]
[assembly: AssemblyCompany("")]
[assembly: AssemblyProduct("")]
[assembly: AssemblyCopyright("")]
[assembly: AssemblyTrademark("")]
[assembly: AssemblyCulture("")]
```

为了简洁起见，这里仅列出了该文件的部分代码。在这个文件中，有许多以“[assembly:”开头的代码行，它们就是属性定义。在编译这个文件时，定义了的任何属性都保存在最后的装配件中，这个过程称为“pickling”。要了解这个过程，可以修改上面的一个属性，例如 AssemblyTitle 属性，并编译装配件：

```
[assembly: AssemblyTitle("Wrox rocks")]
```

编译完成后，在 Windows Explorer 中单击装配件，显示属性表。图 22-2 显示了版本信息选项卡。Description 字段包含 AssemblyTitle 属性中的描述信息。

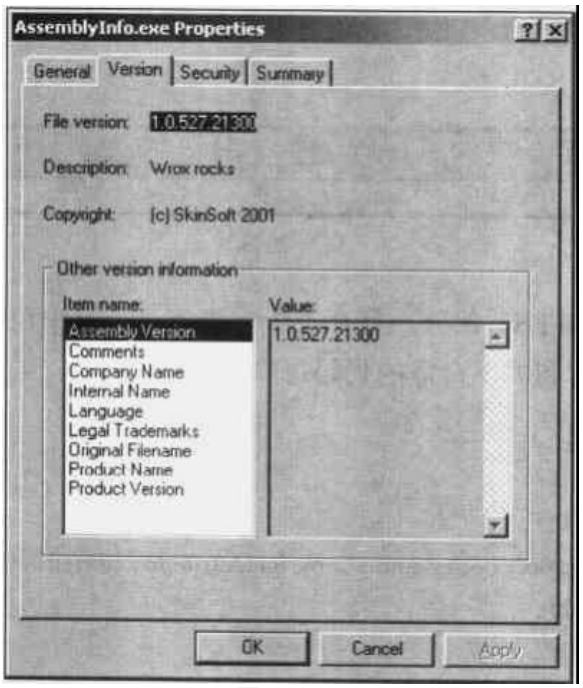


图　22-2

在版本信息选项卡中，该装配件的属性及其对应的名称如表 22-1 所示。

表　22-1

属　性	版 本 信 息
AssemblyTitle	描述
AssemblyDescription	注释
AssemblyCompany	公司名称
AssemblyTrademark	合法商标
AssemblyVersion	装配件版本和产品版本
AssemblyCopyright	版权
AssemblyProduct	产品名称

您可能已经注意到，通过属性表得到的属性要比在装配件中定义的属性少。Microsoft 把一些最常见的属性映射到属性表上，但其他属性就必须编写代码了(参见后面的“反射”一节)，也可以使用 ildasm。

为了查找给定装配件上的所有属性，可以使用 ildasm 检查装配件，查找其中定义的属性。如果还没有把 ildasm 作为外部工具添加到 VS 中，可参阅前面的章节，把它添加到 VS 中。

现在打开 ildasm，并使用 File | Open 选择装配件。选择了一个装配件后，屏幕应如图 22-3 所示。

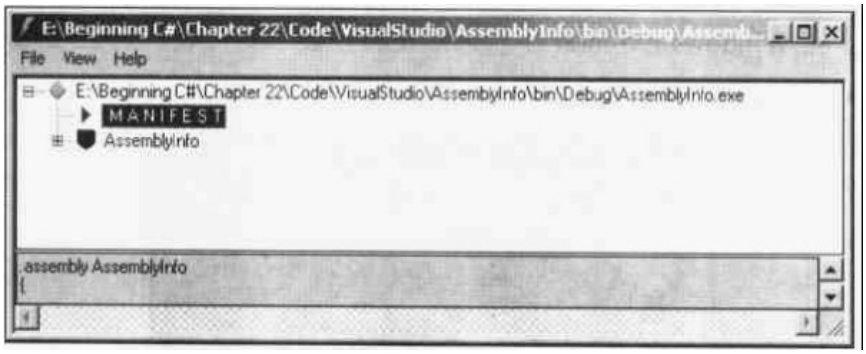


图　22-3

双击突出显示的 MANIFEST 部分，打开一个次级窗口，其中包含第 21 章介绍的装配件清单。向下滚动文件，就会看到如下奇怪的代码：

```
.assembly AssemblyInfo
{
    .custom instance void
    [mscorlib]System.Reflection.AssemblyCopyrightAttribute::.ctor(string)
        = (01 00 00 00 00)
    .custom instance void
    [mscorlib]System.Reflection.AssemblyKeyNameAttribute::.ctor(string)
        = (01 00 00 00 00)

...

    .custom instance void
```

```
    [mscorlib]System.Reflection.AssemblyTitleAttribute::.ctor(string)
      = (01 00 0A 57 72 6F 78 20 72 6F 63 6B 73 00 00) // ...Wrox rocks..

    .hash algorithm 0x00008004
    .ver 1:0:522:37167
}
```

向下查看文件，可以注意到有许多声明类似于类型声明：

```
[mscorlib]System.Reflection.AssemblyTitleAttribute::.ctor(string)
  = (01 00 0A 57 72 6F 78 20 72 6F 63 6B 73 00 00)    // ...Wrox rocks..
```

前面键入的 AssemblyTitle 一直在装配件清单中，如果打开 16 进制/ASCII 转换表，就会看到“01 00 0A”后面的字符集是'Wrox rocks'的 ASCII 码。其中，前面的“01 00”是一个二字节的 ID，“0A”是字符串的长度(10 个字符)。在装配件中存储属性的过程称为“pickling”，如果在 Web 上查看一些.NET 背景资料，就会遇到这个“pickling”。

您可能注意到，在 AssemblyInfo.cs 的代码段中，使用了术语 AssemblyTitle，但在 IL 代码中，则使用 AssemblyTitleAttribute。C#编译器会先查找一个属性类 AssemblyTitle，如果没有找到，就在该词的后面加上 Attribute，并再次搜索。所以无论是键入了完整的类名，还是忽略了最后的 Attribute，都会得到相同的代码。本章将省略后缀 Attribute。

装配件清单中定义的属性声明看起来像是一个对象及其构造函数，方括号中的字节是传递给构造函数的参数。

介绍了背景知识后，就可以定义属性了。

提示：

属性是一个包含装配件中其他数据的类，这些数据涉及到装配件或装配件中的任何类。

既然属性是一个类，在装配件清单中属性以上述格式存储，下面就详细说明本章前面的属性定义：

```
[assembly: AssemblyTitle("Wrox rocks")]
```

该语法与正常的 C#有点区别，用方括号把属性括起来。assembly:标记定义了属性的作用域(详见本章后面的内容)，其他代码声明了属性。AssemblyTitle 属性有一个构造函数，该构造函数只有一个参数——字符串值。编译器把这个值包含在装配件中。要访问该值，可以利用标准的 Windows 属性表，或者在 ildasm 中查看装配件，或者利用反射(详见下一节)编程访问该值。

除了可以使用简单的属性处理装配件信息外，.NET Framework 还定义了近 200 个属性，用于各种处理操作，例如调试、设计时的控制操作和串行化等。在反射一节之后将介绍一些标准属性，再说明如何利用定制属性扩展.NET。

22.2 反射

除非您有一定的 Java 基础，否则反射(reflection)可能是一个新论题，所以这里用一些篇幅

定义反射，并说明如何使用它。

反射可以编程检查装配件，获得该装配件的信息，包括其中包含的所有对象类型。这些信息还包含用户添加到这些类型中的属性。反射对象位于 System.Reflection 名称空间中。

除了读取给定装配件中定义的类型外，还可以使用 System.Reflection.Emit 的服务程序生成自己的装配件和类型。这个论题超出了本书的范围，但如果您对此感兴趣，可以参阅 MSDN 中发送动态装配件的内容。

试试看：查找属性

本节的第一个范例将检查一个装配件，并显示其中定义的所有属性列表，生成一个类似于前面的列表。本节的范例代码都可以从 Wrox Press Web 站点上下载，它们位于 Chapter22/FindAttributes 目录下。完整的源文件如下所示：

```
// FindAttributes.cs
// Import types from System and System.Reflection
using System;
using System.Reflection;

/// <summary>
/// Corresponds to section titled 'Reflection' in Chapter 22
/// </summary>
public class FindAttributes
{
    /// <summary>
    /// Main .exe entry point
    /// </summary>
    /// <param name="args">Command line args - should be an assembly</param>
    static void Main(string[] args)
    {
        // Output usage information if necessary
        if (args.Length == 0)
            Usage ();
        else if ((args.Length == 1) && (args[0] ==  "/?"))
            Usage ();
        else
        {
            // Load the assembly
            string assemblyName = null;

            // Loop through the arguments passed to the console application.
            // I'm doing this as if you specify a full path name including spaces,
            // you end up with several arguments - I'm just stitching them back
            // together again to make one filename...
            foreach (string arg in args)
```

```
        {
            if (assemblyName == null)
                assemblyName = arg;
            else
                assemblyName = string.Format("{0} {1}" , assemblyName , arg);
        }

        try
        {
            // Attempt to load the named assembly
            Assembly a = Assembly.LoadFrom(assemblyName);

            // Now find the attributes on the assembly
            // The parameter is ignored, so I chose true
            object[] attributes = a.GetCustomAttributes(true);

            // If there were any attributes defined...
            if (attributes.Length > 0)
            {
                Console.WriteLine("Assembly attributes for '{0}'...",
                    assemblyName);

                // Dump them out...
                foreach (object o in attributes)
                    Console.WriteLine("   {0}" , o.ToString ());
            }
            else
                Console.WriteLine("Assembly {0} contains no Attributes.",
                    assemblyName);
        }
        catch (Exception ex)
        {
            Console.WriteLine("Exception thrown loading assembly {0}...",
                assemblyName);
            Console.WriteLine ();
            Console.WriteLine(ex.ToString ());
        }
    }
}

/// <summary>
/// Display usage information for the .exe
/// </summary>
static void Usage()
{
```

```
        Console.WriteLine("Usage:");
        Console.WriteLine("    FindAttributes <Assembly>");
    }
}
```

代码的说明

范例代码首先检查传递给命令行的参数，如果没有提供参数，或者用户键入了“FindAttributes /?”，就调用 Usage 方法，显示一个简单的命令用法小结：

```
if (args.Length == 0)
  Usage ();
else if ((args.Length == 1) && (args[0] ==   "/?"))
  Usage ();
```

接着在一个字符串中重新构建命令行参数。其原因是目录名中有空格是很常见的，例如'Program Files'，有这样的一个空格，该目录就会被当作两个参数。所以应迭代所有的参数，把它们组合到一个字符串中，再把该字符串用作要加载的装配件名：

```
foreach (string arg in args)
{
  if (assemblyName == null)
    assemblyName = arg;
  else
    assemblyName = string.Format ("{0} {1}", assemblyName , arg);
}
```

然后尝试加载装配件，检索其中定义的所有定制属性：

```
Assembly   a = Assembly.LoadFrom (assemblyName);

// Now find the attributes on the assembly
// The parameter is ignored, so I chose true
object[]   attributes = a.GetCustomAttributes(true);
```

找到的任何属性都将输出到控制台上。下面在 Visual Studio.NET 中建立可执行文件，如果愿意，可以使用命令行编译器：

```
csc FindAttributes.cs
```

该编译器会编译文件，生成一个控制台可执行文件，以后就可以调用该文件了。

要运行 FindAttributes 应用程序，需要提供要检查的装配件名。现在可以使用 FindAttributes.exe 装配件本身，如图 22-4 所示。

尽管没有指定 DebuggableAttribute，但 C#编译器仍添加了该属性。如果在调试模式下编译可执行文件，就会添加该属性，但在编写过程中，似乎所有的可执行文件都有这个属性。

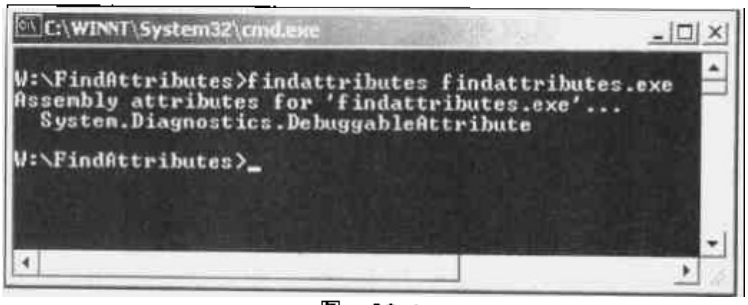


图　22-4

可以适当地修改代码，添加任意多个装配件属性。例如，更新 FindAttributes 的源代码，如下所示：

```
using System;
using System.Reflection;

[assembly: AssemblyTitle("Wrox rocks")]

public class FindAttributes
```

运行时，就可以得到如图 22-5 所示的输出。



图　22-5

本章的后面会继续讨论反射，说明如何检索装配件中定义的类和方法的属性。

22.3　内置属性

前几节讨论了.NET Framework 包含许多属性，例如 DebuggableAttribute 和 AssemblyTitleAttribute 属性。本节将论述一些在.NET Framework 中定义的较常见的属性，并说明何时使用它们。

本节讨论的属性如下：

- System.Diagnostics.ConditionalAttribute
- System.ObsoleteAttribute
- System.SerializableAttribute
- System.Reflection.AssemblyDelaySignAttribute

要了解.NET 附带的更多属性，可以参阅 MSDN，或者在安装.NET 时安装 MSDN，或者使

用http://msdn.microsoft.com上的在线版本。

在使用.NET 时，另一个非常有用的工具是'Reflector'程序，它可以从http://www.aisto.com/roeder/dotnet/上下载。这个程序利用反射来检查装配件。使用它可以在很短的时间内查找派生于 System.Attribute 的所有类。这是一个必须的工具。

22.3.1 System.Diagnostics.ConditionalAttribute

这是最常用的属性之一，因为它允许在编译时，根据符号的定义包含或不包含几个代码块。这个属性包含在 System.Diagnostics 名称空间中，该名称空间包含调试和跟踪输出的类、事件日志、性能计数器和过程信息。下面的代码是使用该属性的一个范例：

```
using System;
using System.Diagnostics;

class TestConditional
{
    static void Main(string[] args)
    {
        // Construct a new TestConditional object
        TestConditional tc = new TestConditional ();

        // Call a method only available if DEBUG is defined...
        tc.DebugOnly ();
    }

    // Class constructor
    public TestConditional ()
    {
    }

    // This method is atributed, and will ONLY be included in
    // the emitted code if the DEBUG symbol is defined when
    // the program is compiled
    [Conditional("DEBUG")]
    public void DebugOnly ()
    {
        // This line will only be displayed in debug builds...
        Console.WriteLine ("This string only displays in Debug");
    }
}
```

这个范例的源代码位于 Chapter22/Conditional 目录下。代码在静态的 Main 函数中构建了一个 TestConditional 类的实例，然后调用 DebugOnly 方法，这是用 Conditional 属性来描述的，这个函数仅在控制台上显示一行文本。

在编译一个 C#源文件时，可以在命令行上定义符号。如果没有给出某个方法的符号，

Conditional 属性将防止调用该方法。

如果在 Visual Studio.NET 中编译一个调试版程序，DEBUG 符号就会自动设置。如果要为某个工程定义或细化符号，就将显示工程属性对话框，并导航到 Configuration Properties 的 Build 部分，如图 22-6 所示。

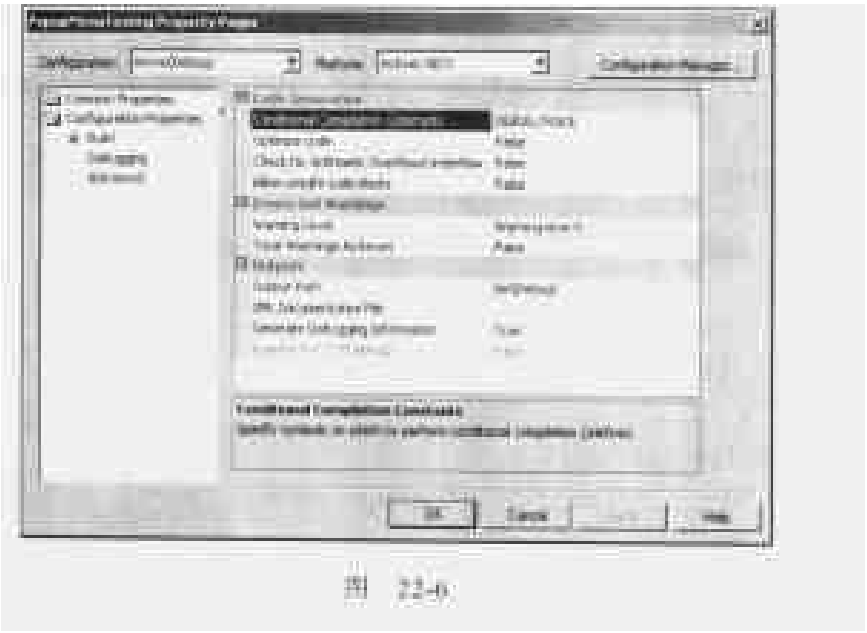


图 22-6

注意 Debug 块的默认设置是定义 DEBUG 和 TRACE。

要在命令行上定义一个符号，可以使用/d:选项(/define:的缩写，也可以键入完整的字符串)：

```
csc /d:DEBUG /r:System.dll conditional.cs
```

如果用上面的命令行编译和运行文件，就会得到输出字符串“This string only displays in Debug”。如果编译时没有在命令行上定义 DEBUG 符号，那么程序将什么也不显示。

要清楚地了解生成的代码中发生了什么情况，可以使用 ildasm 查看生成的代码，如图 22-7 所示。



图 22-7

如果没有定义 DEBUG 符号，为 Main()方法生成的 IL 代码将如下所示：

```
.method private hidebysig static void Main(string[] args) cil managed
{
    .entrypoint
    // Code size    7 (0x7)
```

```
    .maxstack  1
    .locals (class TestConditional V_0)
    IL_0000:  newobj instance void TestConditional::.ctor()
    IL_0005:  stloc.0
    IL_0006:  ret
  } // end of method TestConditional::Main
```

这段代码仅创建了 TestConditional 对象的一个实例(IL_0000)，把它存储在一个局部变量中，并返回。

但如果用/d:DEBUG 选项编译文件，就将产生如下代码：

```
.method private hidebysig static void Main(string[] args) cil managed
{
  .entrypoint
  // Code size  13 (0xd)
  .maxstack  1
  .locals (class TestConditional V_0)
  IL_0000:  newobj instance void TestConditional::.ctor()
  IL_0005:  stloc.0
  IL_0006:  ldloc.0
  IL_0007:  callvirt instance void TestConditional::DebugOnly()
  IL_000c:  ret
} // end of method TestConditional::Main
```

添加了突出显示的两行代码，以调用条件方法。使用 Conditional 将删除对方法的调用，而不是删除方法本身。

提示：

Conditional 属性只能用于无返回值的方法，否则，删除调用就意味着没有返回值；但可以把方法设置为带有 out 或 ref 参数，变量仍保有其原始值。

22.3.2 System.ObsoleteAttribute

在新语言的新书中介绍这个属性似乎有些奇怪，但这说明了 Microsoft 工程人员在.NET Framework 中考虑了许多细节性问题。Obsolete 属性可以把类、方法或装配件中的其他项目标记为不再使用。

这个属性是很有用的，例如发布一个类库。在开发一系列类的过程中，会有一些类、方法或属性被取代，这是不可避免的。这个属性可以让代码的使用者最终抛弃某个功能。

假定在应用程序的版本 10 中，有一个类：

```
public class Developer
{
  public Developer ()
  {
  }
```

```
    public void OriginalMethod ()
    {
    }
}
```

编译并使用这个类，几年后，要加入一些新东西：

```
public void NewMethod ()
{
}
```

自然，要允许库的使用者使用 OriginalMethod()一段时间，就应在编译时显示一个警告信息，告诉他们有一个更新的方法，通知用户 NewMethod()的存在。为此，需要添加一个 Obsolete 属性，如下所示：

```
[Obsolete("Use NewMethod instead.")]
public void OriginalMethod ()
{
}
```

再次为调试或发布而编译时，C#编译器就会给您发送一个警告：现在使用的是一个过时(或迟早要过时)的方法：

```
obsolete.cs(10,3): warning CS0618: 'Developer.OriginalMethod()' is obsolete:
    'Use NewMethod instead'
```

在这个过程中，每个人都不希望每次编译代码时都看到这个警告信息，最终他们(几乎是所有人)都将使用 NewMethod()。我们最后会删除对 OriginalMethod()的支持，此时应给 Obsolete 属性添加一个额外的参数：

```
[Obsolete("You must use NewMethod instead.", true)]
public void CodeInC ()
{
}
```

在用户试图编译这个方法时，编译器会生成一个错误，暂停编译，显示下述信息：

```
obsolete.cs(10,3): error CS0619: 'Developer.OriginalMethod()' is obsolete:
    'You must use NewMethod instead.'
```

使用这个属性，将为类的用户提供帮助，以修改使用该类的应用程序，因为类在不断地演化。

对于二进制类，例如购买没有源代码的组件，这并不是升级版本的好方法，.NET Framework 内置了非常优秀的版本升级功能，本书的其他地方介绍了这个功能。但它提供了一种很好的方式，提示不应再继续使用类的某个功能。

22.3.3 System.SerializableAttribute

串行化是指存储和获取磁盘文件、内存或其他地方中的对象。在串行化时，所有的实例数据都保存到存储介质上，在取消串行化时，对象会被还原，且不能与其原实例区别开来。

以前在用 MFC、ATJ 或 VB 编写代码时，必须考虑如何存储和获取实例数据，现在使用这个属性可以大大减少键入量。假定有一个 C#对象实例，它存储在一个文件中，如下所示：

```
public class Person
{
    public Person ()
    {
    }

    public int Age;
    public int WeightInPounds;
}
```

使用 C#语言(以及.NET Framework 中的任何语言)，可以串行化实例的成员，而几乎不需要编写任何代码。只需给类添加 Serializable 属性，.NET 运行时就会完成其他的工作。

在运行时接收到串行化一个对象的请求时，就会检查对象的类是否执行 ISerializable 接口，如果没有执行该接口，就检查该类是否有 Serializable 属性。这里不打算进一步讨论 ISerializable，它是一个高级论题，在 MSDN 中已经有一些很好的说明了。

如果类上有 Serializable 属性，.NET 就使用反射获取所有的实例数据，是否为公共、受保护的、私有，并把这些数据存储为对象的表示方式。并行化是这个过程的逆过程，数据从存储介质中读取出来，并赋给类的实例变量。

下面是一个标记有 Serializable 属性的类：

```
[Serializable]
public class Person
{
    public Person ()
    {
    }

    public int Age;
    public int WeightInPounds;
}
```

试试看：串行化

这个范例的所有代码都在 Chapter22/Serialize 子目录下。要存储 Person 类的一个实例，可以使用 Formatter 对象，把存储在类中的数据转换为一个字节流。这个系统有两种默认的格式标识符，即 BinaryFormatter 和 SoapFormatter(它们在 System.Runtime.Serialization.Formatters 下有

自己的名称空间)。下面的代码显示了如何使用 BinaryFormatter 存储一个 Person 对象：

```
public static void Serialize ()
{
  // Construct a person object
  Person   me = new Person ();

  // Set the data that is to be serialized
  me.Age = 34;
  me.WeightInPounds = 200;

  // Create a disk file to store the object to...
  Stream   s = File.Open ("Me.dat" , FileMode.Create);

  // And use a BinaryFormatted to write the object to the stream...
  BinaryFormatter bf = new BinaryFormatter ();

  // Serialize the object
  bf.Serialize (s , me);

  // And close the stream
  s.Close ();
}
```

代码的说明

代码首先创建 person 对象，设置 Age 和 WeightInPounds 数据，接着在文件 Me.dat 上构造一个数据流。二进制格式标识符利用该数据流把 person 类的实例存储到 Me.dat 上，最后关闭该数据流。

默认的串行化代码会存储对象的所有内容，在大多数情况下，这正是我们需要的。但在某些情况下，要定义一个或多个不进行串行化的字段，这也很容易：

```
[Serializable]
public class Person
{
   public Person ()
   {
   }

   public int Age;
   [NonSerialized]
   public int WeightInPounds;
}
```

在串行化这个类时，只存储了 Age 成员，没有存储 WeightInPounds 成员，所以不能在并行化时获取这个成员。

并行化基本上是上述串行化代码的逆过程。下面的范例在前面创建的 Me.dat 文件上打开一个数据流，构造一个 BinaryFormatter，以读取对象，调用 Deserialize 方法，以获取对象。接着把它转换为一个 Person，把年龄和体重写到控制台上：

```
public static void DeSerialize ()
{
  // Open the file this time
  Stream    s = File.Open ("Me.dat" , FileMode.Open);

  // And use a BinaryFormatted to read object(s) from the stream
  BinaryFormatter bf = new BinaryFormatter ();

  // Deserialize the object
  object    o = bf.Deserialize (s);

  // Ensure it is of the correct type...
  Person    p = o as Person;

  if (p != null)
    Console.WriteLine ("DeSerialized Person aged: {0} weight: {1}" ,
                       p.Age , p.WeightInPounds);

  // And close the stream
  s.Close ();
}
```

一般情况下，常定义可计算为 NonSerialized 的数据，因为这些数据可以在需要时重新计算。例如，有一个计算素数的类，您可以缓存这些素数，以便在使用类时加快响应速度，但不需要对一组素数进行串行化和并行化的操作，因为它们可以在需要时重新计算。有时成员只与对象的某个特殊的用途有关，例如，在一个表示字处理器文档的对象中，要串行化文档的内容，但通常不串行化插入点的位置，当下次加载该文档时，仅把插入点放在文档的开头即可。

如果还要对对象串行化的方式进行更多的控制，可以执行 ISerializable 接口，这是一个高级论题，本书不讨论。

22.3.4 System.Reflection.AssemblyDelaySignAttribute

反射名称空间提供了许多属性，其中一些属性已经在本章的前面介绍过了。还有一个比较复杂的属性是 AssemblyDelaySign。在第 21 章中，我们学习了如何构建装配件，如何创建全局装配件，以及如何在 Global Assembly Cache (GAC)中注册它们。.NET Framework 还允许给装配件加上“延迟标记”，即可以在 GAC 中注册它，以便在测试时不需要私有密钥。

使用延迟标记的一种场合是在开发商用软件时。在室内开发的每个装配件在销售给客户之前，都需要用公司的私有密钥标记，所以当编译装配件时，应在 GAC 中注册它之前引用密钥文件。

但是，许多组织都不希望它们的私有密钥出现在每个开发人员的机器上。因此运行时允许部分地标记装配件，变换几个设置，使装配件可以在 GAC 中注册。在全部测试完毕后，就可以由拥有私有密钥文件的人来标记它。这可以是 QA 部门、一个或多个可信任的人或市场部门。

在下面的范例中，将说明如何给一般的装配件加上延迟标记，如何在 GAC 中注册它，以进行测试，最后在私有密钥中添加它，以完成标记。

试试看：生成一个密钥文件

生成一个密钥文件

首先需要用实用程序 sn.exe 创建一个密钥文件，sn.exe 一般安装在 Program Files\Microsoft.NET\FrameworkSDK\Bin 目录下。这个密钥文件包含公共密钥和私有密钥，所以我们称之为 Company.Key：

```
sn –k Company.Key
```

接着需要获取开发人员使用的公共密钥部分：

```
sn –p Company.Key Company.Public
```

这个命令会生成一个密钥文件 Company.Public，其中只包含密钥的公共部分。这个公共密钥文件可以复制到所有的机器上，不需要保密，但私有密钥文件需要保密。把 Company.Key 文件存储在某个安全的地方，因为它只需要在最终标记装配件时使用。

为了给装配件加上延迟标记，并在 GAC 中注册它，还需要得到公共密钥记号，该记号基本上是公共密钥的一个缩写版本，在注册装配件时使用它。可以用下述两种方式获取该记号：

- 从公共密钥文件本身获取

```
sn –t Company.Public
```

- 从使用该密钥标记的任何装配件中获取：

```
sn –T <assembly>
```

这两个命令都会显示公共密钥的一个散列版本，且是区分大小写的。我们在注册装配件时再介绍它。

试试看：延迟标记装配件

下面的代码说明了如何给装配件加上延迟标记。这些代码位于 Chapter22/DelaySign 目录下。

```
using System;
using System.Reflection;

// Define the file which contains the public key
[assembly: AssemblyKeyFile ("Company.Public") ]
// And that this assembly is to be delay-signed
[assembly: AssemblyDelaySign (true) ]
```

```
public class DelayedSigning
{
  public DelayedSigning ()
  {
  }
}
```

代码的说明

AssemblyKeyFile 属性定义了应在哪个文件中查找密钥。该文件既可以是公共密钥文件，也可以是由可以信任的人建立的、包含公共密钥和私有密钥的文件。AssemblyDelaySign 属性定义了装配件是完整地标记(false)了，还是做了延迟标记(true)。

如果使用 Visual Studio.NET 编写所有的工程，Visual Studio.NET 就会创建一个文件 AssemblyInfo.cs，它包含各种属性，例如本章前面介绍的版本升级和文件信息属性。它还定义了 AssemblyDelaySign 属性，并用每个新工程创建其默认值。

在编译时，装配件将在清单中包含用于公共密钥的一个项目。实际上，装配件清单还包含用于私有密钥的足够空间，所以重新标记装配件肯定不会改变它(而是把一些额外的字节写到装配件清单中)，如图 22-8 所示。

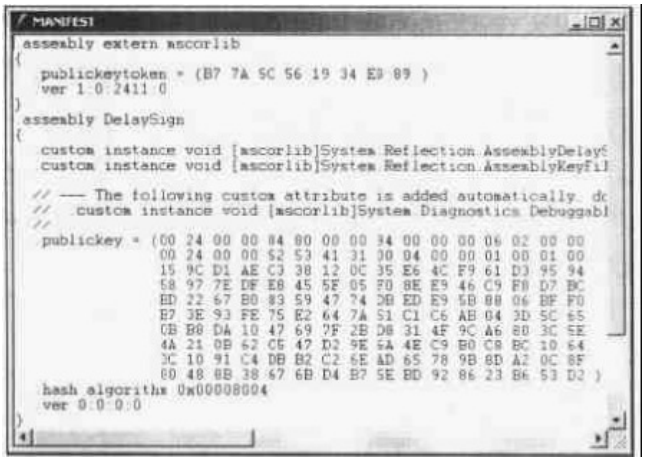


图　22-8

该窗口的底层部分显示了存储在装配件清单中的公共密钥。还要注意，在顶部有一个对 mscorlib.dll 的引用。这个库包含 System(及其派生的)名称空间中的大多数对象。该窗口不仅包括该装配件的版本信息，还包含其公共密钥记号(这是一系列字节)。在第三方装配件引用您自己的装配件时，它们也会包含您的公共密钥的一个散列。

如果使用 ildasm 查看一个装配件，而该装配件引用了一个延迟标记的装配件，在装配件清单中就会有一个项目包含公共密钥记号。要在命令行上添加对另一个装配件的引用，可使用/r 选项，例如在/r:System.Data.dll 中：

```
.assembly extern DelaySign
{
    .publickeytoken = (34 AA A4 14 6E E0 1E 4A)
    .ver 0:0:0:0
```

```
}
```

在 GAC 中注册

试图使用 FrameworkSDK\Bin 目录下的 gacutil 工具，在 GAC 中注册一个延迟标记的装配件，会产生如下错误信息(或者类似的信息)：

```
Microsoft (R) .NET Global Assembly Cache Utility. Version 1.0.2914.16
Copyright (C) Microsoft Corp. 1998-2001. All rights reserved.
Failure adding assembly to the cache: Strong name signature could not be verified. Was the assembly built
delay-signed?
```

此时装配件只进行了部分标记，在默认情况下，GAC 只接受有完整强名的装配件。但对于延迟标记的装配件，可以使用 sn 实用程序指示 GAC 跳过强名的验证。记得前面的公共密钥记号吗？此时就是它大显身手的时候了。

```
sn -Vr *,34AAA4146EE01E4A
```

这将指示 GAC 允许注册公共密钥记号为 34AAA4146EE01E4A4A 的装配件。在命令提示行上键入这个命令，会得到如下信息：

```
Microsoft (R) .NET Framework Strong Name Utility    Version 1.0.2914.16
Copyright (C) Microsoft Corp. 1998-2001. All rights reserved.

Verification entry added for assembly '*,34AAA4146EE01E4A'
```

现在就可以成功地把装配件安装到 GAC 上了。在为装配件添加验证项目时，不需要使用公共密钥值，可以使用下述命令，指定注册的所有装配件：

```
sn –Vr *
```

也可以通过键入装配件的完整名称来指定它，如下所示：

```
sn –Vr DelaySign.dll
```

这些数据会永远保存在所谓的 Verification Skip Table 中，这是一个存储在磁盘上的文件。要得到 Verification Skip Table 中的项目列表，应键入下述命令：

```
C:\> sn -Vl
```

这些命令是区分大小写的。下面是输出：

```
Microsoft (R) .NET Framework Strong Name Utility    Version 1.0.2914.16
Copyright (C) Microsoft Corp. 1998-2001. All rights reserved.

Assembly/Strong Name                    Users
=============================================
*,03689116d3a4ae33                      All users
*,33aea4d316916803                      All users
```

*,34AAA4146EE01E4A	All users
*,631223CD18E5C371	All users
*,b03f5f7f11d50a3a	All users
*,b77a5c561934e089	All users

注意 Users 列，可以定义某个给定的装配件能由一部分用户加载到 GAC 中。参阅 sn.exe 文档说明，了解这方面及其他装配件命名选项的更多内容。

完整的强名

这个过程的最后一项是把公共密钥和私有密钥编译到装配件中，带有这两项的装配件称为有完整的强名，可以在 GAC 中注册，且不必跳过验证项目。

再次使用 sn.exe 实用程序，但这次是使用–R 选项，该选项可以重新标记装配件，并添加私有密钥部分：

```
sn -R delaysign.dll Company.Key
```

显示的结果如下：

```
Microsoft (R) .NET Framework Strong Name Utility    Version 1.0.2914.16
Copyright (C) Microsoft Corp. 1998-2001. All rights reserved.

Assembly 'delaysign.dll' successfully re-signed
```

其他参数还有要重新标记的装配件名，包含公共密钥和私有密钥的密钥文件。

22.4 定制属性

本章的前半部分主要介绍了.NET Framework 包含的一些属性，这并不是所有的属性，我们还可以创建自己的属性。

本章仅涉及处理定制属性较肤浅的知识。本节主要介绍下述属性：

- TestCaseAttribute ——把测试类的代码与类本身链接起来。
- BugFixAttribute ——记录谁修改了源代码中的什么内容，以及修改的时间。
- DatabaseTableAttribute and DatabaseColumnAttribute ——说明如何从.NET 类中创建一个数据库模式。

定制属性只是一个特殊的类，它必须遵循下述两个规范：

- 属性必须派生于 System.Attribute。
- 属性的构造函数只能包含可在编译时解析的类型，例如字符串和整数。

对于属性构造函数的参数来说，其类型的限制源于属性存储在装配件元数据中的方式。在代码中使用属性时，就是在内联中使用属性的构造函数。例如：

```
[assembly: AssemblyKeyFile ("Company.Public") ]
```

这个属性存储在装配件的元数据中，作为调用 AssemblyKeyFileAttribute 的构造函数的一个

指令。该函数带一个字符串参数。在上面的范例中，该字符串是"Company.Public"。如果定义了一个定制属性，该属性的使用者就会把参数写入到类的构造函数中。

第一个范例 TestCaseAttribute 说明了测试类是如何与它们测试的代码耦合在一起的。

22.4.1 TestCaseAttribute

在单元测试软件中，常常要定义一系列测试类来检查类，以确保它们按照希望的那样执行。在回归测试中更是常常使用测试类，因为我们要确保更正一个错误或添加额外的功能，不必破坏其他的代码。

在与受控制的客户一起工作(例如为医药公司编写软件，而这些公司在政府代理机构的严格控制下工作)时，就需要提供代码和测试之间的交叉引用。这里的 TestCaseAttribute 可以帮助我们跟踪类及其测试类。

试试看：创建一个定制属性类

完整的源代码位于 Chapter22/TestCase 目录下。

要创建一个定制属性类，需要：

- 创建一个派生于 System.Attribute 的类。
- 按照需要创建构造函数和公共属性。
- 给类添加属性，以定义可以在什么地方使用定制属性。

下面逐个讨论这些步骤：

(1) 创建定制属性类

这是最简单的一步。只需创建一个派生于 System.Attribute 的类即可。

```
public class TestCaseAttribute : Attribute
{
}
```

(2) 创建构造函数和属性。

如前所述，当用户使用属性时，都会调用该属性的构造函数。对于这个 TestCase 属性，我们要定义用于测试给定类的对象类型，所以使用 System.Type 值：

```
public class TestCaseAttribute : Attribute
{
  /// <summary>
  /// Constructor for the class
  /// </summary>
  /// <param name="testCase">The object which contains
  /// the test case code</param>
  public TestCaseAttribute (System.Type testCase)
  {
     TestCase = testCase;
  }
```

```
    /// <summary>
    /// The test case class
    /// </summary>
    public readonly System.Type TestCase;

    /// <summary>
    /// Perform the test
    /// </summary>
    public void Test ()
    {
      // Create an instance of the class under test
      // The test case object created is assumed to
      // test the object in its' constructor
      object o = Activator.CreateInstance (TestCase);
    }

}
```

这段代码定义了一个构造函数和一个只读的成员变量 TestCase。Test()方法用于实例化 TestCase，因为这个范例在 TestCase 类的构造函数中进行测试。

(3) 标记类的用途

最后需要标记属性类，指定可以在什么地方使用这个属性类。这个 TestCase 属性只在类上有效。我们可以确定创建的属性在什么地方有效。本章的后面会对此进行详细的说明：

```
[AttributeUsage(AttributeTargets.Class,
                AllowMultiple=false,
                Inherited=true)]
public class TestCaseAttribute : Attribute
{
  /// <summary>
  /// Constructor for the class
  /// </summary>
  /// <param name="testCase">The object which contains
  /// the test case code</param>
  public TestCaseAttribute (System.Type testCase)
  {
      TestCase = testCase;
  }

  /// <summary>
  /// The test case class
  /// </summary>
  public readonly System.Type TestCase;

  /// <summary>
```

```
    /// Perform the test
    /// </summary>
    public void Test ()
    {
      // Create an instance of the class under test
      // The test case object created is assumed to
      // test the object in its constructor
       object o = Activator.CreateInstance (TestCase);
    }
}
```

AttributeUsage 属性有一个构造函数，它带有一个 AttributeTargets 枚举中的值(详见本节后面的内容)。这里只说明放置 TestCase 属性惟一有效的地方是在一个类上。使用 OR 可以指定这个枚举中的几个值，所以其他属性在类、构造函数或属性上是有效的。

在定义属性时，我们还利用了属性的两个特性：AllowMultiple 和 Inherited。详见本节的后面。

下面需要一个对象来测试 TestCase。这个类没有什么特殊的地方：

```
[TestCase (typeof(TestAnObject))]
public class SomeCodeOrOther
{
    public SomeCodeOrOther ()
    {
    }

    public int Do ()
    {
        return 999;
    }
}
```

类的前面有 TestCase 属性，使用 typeof()定义用于测试代码的类。要编译这个范例，需要编写一个测试类。这个对象用于检查测试代码的一个实例，如下所示：

```
public class TestAnObject
{
    public TestAnObject ()
    {
        // Exercise the class under test
        SomeCodeOrOther scooby = new SomeCodeOrOther ();

        if (scooby.Do () != 999)
            throw new Exception ("Pesky Kids");
    }
}
```

这个类仅实例化了测试类，调用一个方法。如果返回值与期望的不一样，就抛出一个异常。更完整的 TestCase 将完整地检查测试对象，调用该类上的所有方法，传递不在作用域内的值，检查错误条件，可能要设置用于上下文信息的一些类，如果测试一个访问数据库的类，还要传递一个连接对象。

现在回到主代码上。这个类循环装配件中的所有类型，查找定义了 TestCaseAttribute 的类型。如果找到，就获取该属性，调用 Test()方法：

```
using System;
using System.Reflection;

[AttributeUsage(AttributeTargets.Class,AllowMultiple=false,Inherited=true)]
public class TestCaseAttribute : Attribute
{
    // Code removed for brevity
}

/// <summary>
/// A class that uses the TestCase attribute
/// </summary>
[TestCaseAttribute(typeof(TestAnObject))]
public class SomeCodeOrOther
{
    // Code removed for brevity
}

// Main program class
public class UnitTesting
{
  public static void Main ()
  {
    // Find any classes with test cases in the current assembly
    Assembly   a = Assembly.GetExecutingAssembly ();

    // Loop through the types in the assembly and test them if necessary
    System.Type[] types = a.GetExportedTypes ();

    foreach (System.Type t in types)
    {
      // Output the name of the type...
      Console.WriteLine ("Checking type {0}" , t.ToString ());

      // Does the type include the TestCaseAttribute custom attribute?
      object[]   atts = t.GetCustomAttributes(typeof(TestCaseAttribute),
                                              false);
```

```
            if (1 == atts.Length)
            {
              Console.WriteLine ("   Found TestCaseAttribute: Running Tests");

              // OK, this class has a test case. Run it...
              TestCaseAttribute tca = atts[0] as TestCaseAttribute;

              try
              {
                // Perform the test...
                tca.Test ();
                Console.WriteLine ("   PASSED!");
              }
              catch (Exception ex)
              {
                Console.WriteLine ("   FAILED!");
                Console.WriteLine (ex.ToString ());
              }
            }
          }
        }
      }
```

新的代码部分被突出显示。在运行程序时，程序会通过 Assembly 类的静态方法 GetExecutingAssembly()得到正在运行的装配件。接着在该装配件上调用 GetExportedTypes，查找可以在装配件中公开访问的所有对象类型的列表。

对于装配件中每个导出的类型，程序会检查它是否包含 TestCase 属性。如果包含，就获取该属性(它在内部构造属性实例，把代码中使用的参数传递给对象的构造函数)，并调用 Test() 方法，测试代码。

在运行程序时，程序的输出如下所示：

```
Checking type TestCaseAttribute
Checking type SomeCodeOrOther
    Found TestCaseAttribute: Running Tests
    PASSED!
Checking type TestAnObject
Checking type UnitTesting
```

22.4.2　System.AttributeUsageAttribute

在定义定制属性类时，必须定义属性所使用的类型。在上面的范例中，TestCase 属性仅在类上使用时才有效。为了定义属性可以在什么地方使用，需要添加另一个属性 *AttributeUsage*。

最简单的格式如下所示：

```
[AttributeUsage(AttributeTargets.Class)]
```

它只有一个参数，即表示属性有效的枚举。如果试图用 TestCase 属性标记一个方法，编译器就会产生一个错误信息，无效的用法如下所示：

```
public class TestAnObject
{
    [TestCase (typeof(System.String))]    // Invalid here
    public TestAnObject ()
    {
        etc...
    }
}
```

产生的错误如下：

```
TestCase.cs(54,4): error CS0592: Attribute 'TestCase' is not valid on this
        declaration type. It is valid on 'class' declarations only.
```

AttributeTargets 枚举定义了表 22-2 所示的成员，使用 or 运算符(|)可以把这些成员合并起来，以定义该属性有效的一组元素。

表 22-2

AttributeTargets 值	描　述
All	属性在装配件的任何地方都有效
Assembly	属性在装配件上有效，例如，本章前面的属性 AssemblyKeyFile
Class	属性在类定义上有效。TestCase 属性就使用这个值，另一个范例是 Serializable 属性
Constructor	属性仅在类的构造函数中有效
Delegate	属性仅在委托中有效
Enum	属性可以被添加到枚举值中。例如 System.FlagsAttribute 属性，在应用到一个 enum 上时，该属性定义用户可以使用按位 or 运算符合并枚举中的值。AttributeTargets enum 使用这个属性
Event	属性在事件定义上有效
Field	属性可以放在字段上，例如内部成员变量。它的一个范例是 NonSerialized 属性，它在前面用于指定在串行化一个类时，不应存储某个值
Interface	属性在接口上有效。例如在 System.Runtime.InteropServices 中定义的 GuidAttribute，它允许显式地定义接口的 GUID
Method	定义属性在方法上有效。System.Runtime.Remoting.Messaging 中的 OneWay 属性就使用这个值

(续表)

AttributeTargets 值	描　　述
Module	属性在模块中有效。装配件可以从许多代码模块中创建，所以可以使用这个值把属性放在一个单独的模块上，而不是放在整个装配件上
Parameter	属性可应用于方法定义中的一个参数上
Property	属性可以应用于一个属性
ReturnValue	属性与函数的返回值相关
Struct	属性在结构上有效

1. 属性的作用域

在本章的第一个范例中，介绍了 Assembly*属性，它们的语法如下所示：

```
[assembly: AssemblyTitle("Wrox rocks")]
```

assembly: 字符串指定了属性的作用域，在本例中，是告诉编译器 AssemblyTitle 属性要应用到装配件本身上。只需使用作用域修饰符，编译器就不能在作用域外部工作。例如，如果要把一个属性添加到函数的返回值中：

```
[MyAttribute ()]
public long DoSomething ()
{
    ...
}
```

当编译器执行到这个属性时，就会进行一个有意义的猜测，是否把属性应用到方法本身上，但这里我们不希望把属性应用到方法上，所以用添加了一个修饰符，明确指定属性应关联到什么成员上：

```
[return:MyAttribute ()]
public long DoSomething ()
{
    ...
}
```

如果要定义属性的作用域，可选择下述值：

- assembly ——属性应用到装配件上
- field ——属性应用到枚举或类的字段上
- event ——属性应用到事件上
- method ——属性应用到它前面的方法中
- module ——属性存储在模块中
- param ——属性应用到参数上
- property ——属性存储在属性上

- return ——属性应用到函数的返回值上
- type ——属性应用到类、接口或结构上

许多不经常用作作用域的值一般都很明确。但对于 assembly, module 和 return 这几个值来说，就必须使用作用域标志。如果属性的定义区域有一些模糊，编译器就会选择被赋给属性的对象。当标记函数的返回值时，这是最常见的，如下所示：

```
[SomeAttribute]
public string DoSomething ();
```

编译器猜测属性应用到方法上，而不是返回值上。我们需要以下述方式定义作用域，获得期望的结果：

```
[return:SomeAttribute]
public string DoSomething ();
```

2. AttributeUsage.AllowMultiple

可以定义用户是否能给元素添加一个或多个相同的属性。例如，可以创建一个属性，列出应用到一段代码上的所有错误更正。随着装配件的演化，需要在方法上提供几个错误更正的细节。

下面的代码定义了一个简单的 BugFixAttribute，并使用 AllowMultiple 标志，使属性可以在任意给定的代码块上多次使用：

```
[AttributeUsage (AttributeTargets.Class | AttributeTargets.Property |
                 AttributeTargets.Method | AttributeTargets.Constructor ,
                 AllowMultiple=true)]
public class BugFixAttribute : Attribute
{
    public BugFixAttribute (string bugNumber , string comments)
    {
        BugNumber = bugNumber;
        Comments = comments;
    }

    public readonly string BugNumber;
    public readonly string Comments;
}
```

BugFix 属性的构造函数的参数是一个错误号和一个注释字符串，用 AllowMultiple=true 标记，表示它可以按照如下方式使用：

```
[BugFix("101","Created some methods")]
public class MyBuggyCode
{
```

```
    [BugFix("90125","Removed call to base()")]
    public MyBuggyCode ()
    {
    }

    [BugFix("2112","Returned a non null string")]
    [BugFix("38382","Returned OK")]
    public string DoSomething ()
    {
        return "OK";
    }
}
```

设置 AllowMultiple 标志的语法有点奇怪。AttributeUsage 的构造函数只带一个参数：可以使用属性的标志列表。AllowMultiple 是 AttributeUsage 属性上的一个特性，所以下面的语法表示“构建属性，再把 AllowMultiple 的属性值设置为 true”。

```
[AttributeUsage (AttributeTargets.Class | AttributeTargets.Property |
                 AttributeTargets.Method | AttributeTargets.Constructor ,
                 AllowMultiple=true)]
public class BugFixAttribute : Attribute
{
    ...
}
```

本章后面还将说明一个用于 Inherited 属性的类似方法。如果定制属性有特性，就可以用相同的方式设置它们。例如添加更正错误的人名：

```
    public readonly string BugNumber;
    public readonly string Comments;
    public string Author = null;

    public override string ToString ()
    {
        if (null == Author)
            return string.Format ("BugFix {0} : {1}" ,
                                  BugNumber , Comments);
        else
            return string.Format ("BugFix {0} by {1} : {2}" ,
                                  BugNumber , Author , Comments);
    }
```

这将添加 Author 属性，和一个重写的 ToString()执行代码，如果设置了 Author 属性，该执行代码就会显示所有的细节，否则就只显示错误号和注释。ToString()方法可用于显示给定代码段的更正错误列表 —— 也许会打印到其他地方的文件中。

编写好 BugFix 属性后，就需要以某种方式来报告在类及其成员上所做出的更正。

报告类的错误更正的方法把类的类型(也在 System.Type 中)传递给下面所示的 DisplayFixes 函数，这也使用反射来查找应用到类上的错误更正，再迭代这个类上的所有方法，查找错误更正属性。

这个范例在 Chapter22/BugFix 目录下：

```
public static void DisplayFixes (System.Type t)
{
  // Get all bug fixes for the given type,
  // which is assumed to be a class
  object[]   fixes = t.GetCustomAttributes (typeof (BugFixAttribute) , false);

  Console.WriteLine ("Displaying fixes for {0}" , t);

  // Display the big fix information
  foreach (BugFixAttribute bugFix in fixes)
    Console.WriteLine ("   {0}" , bugFix);

  // Now get all members (i.e. functions) of the class
  foreach (MemberInfo member in t.GetMembers (BindingFlags.Instance |
                                              BindingFlags.Public |
                                              BindingFlags.NonPublic |
                                              BindingFlags.Static))
  {
    // And find any big fixes applied to these too
    object[]   memberFixes = member.GetCustomAttributes (typeof (BugFixAttribute) , false);

    if (memberFixes.Length > 0)
    {
      Console.WriteLine ("   {0}" , member.Name);

      // Loop through and display these
      foreach (BugFixAttribute memberFix in memberFixes)
        Console.WriteLine ("      {0}" , memberFix);
    }
  }
}
```

代码的说明

代码首先从类本身获取所有的 BugFix 属性：

```
object[] fixes = t.GetCustomAttributes (typeof (BugFixAttribute) ,
                                        false);
```

这些都枚举和显示出来了。代码接着使用 GetMembers()方法，迭代类上定义的所有成员：

```
foreach (MemberInfo member in t.GetMembers (
        BindingFlags.Instance | BindingFlags.Public |
        BindingFlags.NonPublic | BindingFlags.Static))
```

GetMembers()从给定的类型中获取属性、方法和字段。要限制返回的成员列表，应使用 BindingFlags 枚举(在 System.Reflection 中定义)。

传递给这个方法的绑定标志指定了我们对哪个成员感兴趣，在本例中无论可见性如何，我们感兴趣的是所有的实例和静态成员，所以指定 Instance 和 Static，以及 Public 和 NonPublic 成员。

在获得了所有的成员之后，就迭代这些成员，查找与特定成员相关的 BugFix 属性，把它们输出到控制台上。要输出给定类的错误更正列表，只需调用静态方法 DisplayFixes()，传递类的类型即可：

```
BugFixAttribute.DisplayFixes (typeof (MyBuggyCode));
```

对于前面给出的 MyBuggyCode 类，将得到如下结果：

```
Displaying fixes for MyBuggyCode
    BugFix 101 : Created some methods
    DoSomething
        BugFix 2112 : Returned a non null string
        BugFix 38382 : Returned OK
    .ctor
        BugFix 90125 : Removed call to base()
```

如果要显示给定装配件中所有类的错误更正，可以使用反射来获取装配件中的所有类型，把每个类型传递给静态方法 BugFixAttribute.DisplayFixes。

3. AttributeUsage.Inherited

在定义定制属性时，通过设置这个标志可以把属性定义为可继承的：

```
[AttributeUsage (AttributeTargets.Class,
                 Inherited = true)]
public class BugFixAttribute { ... }
```

这表示 BugFix 属性可以由使用该属性的任何子类继承，这可能正是我们希望得到的，也可能不是。在 BugFix 属性这个范例中，这种继承性不是我们希望有的，因为错误更正一般会应用到一个类上，而不会应用到派生类上。

假定有一个应用了错误更正的抽象类：

```
[BugFix("38383","Made the class abstract")]
public abstract class GenericRow : DataRow
{
    public GenericRow (System.Data.DataRowBuilder builder)
        : base (builder)
```

```
    {
    }
}
```

如果从这个类创建了一个子类，就不希望把同样的 BugFix 属性反射到子类上——子类没有要更正的错误。但是，如果定义了一组属性，把类的成员链接到数据库表中的字段上，就需要继承这些属性。

在定义数据库模式时，常常会遇到大多数表包含一组标准列的情况，例如 Name 和 Description。可以用这些字段编写一个基类，并包括一个定制属性，该定制属性把类中的一个属性链接到数据库的一个列上。其子类可以适当地添加更多的字段。

下面的范例创建 DatabaseTable 和 DatabaseColumn 属性，它们可以应用到一个类上，让适合于存储该类的数据库表自动生成。

试试看：使用属性生成数据库表

最后一个范例将说明如何使用在.NET 类中的属性生成数据库模式(数据库模式是指一个数据库的设计，包括表、列和数据类型)，以便.NET 对象可以创建自己的数据库表来存储数据。我们将论述如何提取这个模式信息，生成 SQL，以创建数据库中的表，以及构造内存中的 DataTable 对象。

如第 19 章所述，我们使用 DataSet, DataTable, DataRow 和 DataAdapter 对象访问 ADO.NET 中的对象。这些对象的使用应与基本的数据库结构保持同步，这是很重要的。如果数据库结构随时间的改变而变化，就需要确保对表的更新，例如添加新列会反射到访问该数据库的类中。

在这个范例中，我们将创建 DataRow 的子类，它是专门为存储特定数据库表中的数据而设计的。如果基本数据库模式变化得不是很频繁，这可以提供一种访问数据库的高效方式。如果数据库模式变化得很频繁，或者允许用户改变数据库结构，最好能从数据库中请求模式信息，随时建立数据表，动态生成数据表。

图 22-9 说明了 ADO.NET 类和基本数据库表之间的关系。

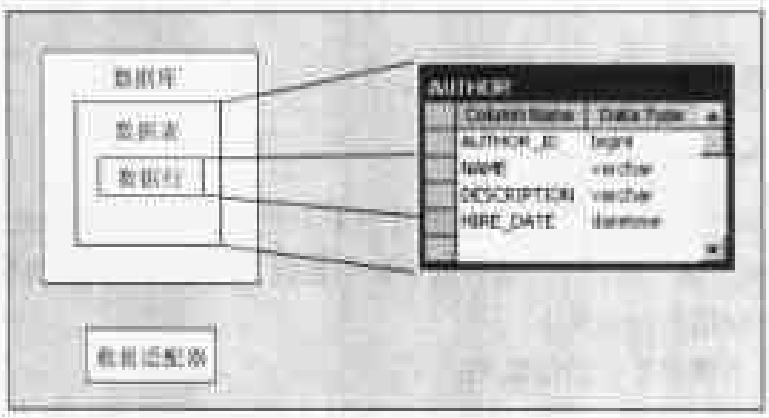


图 22-9

DataSet 由一个或多个 DataTable 对象组成，每个 DataTable 对象都映射到数据库表中的一行上。DataAdapter 用于把数据库中的数据提取到 DataTable 中，以及把 DataTable 中的数据写回数据库。

DataTabl 主要由样板代码组成，所以我们将定义一个基类 DataTable 对象，它可以用作 DataRow 的一般容器。但是 DataRow 需要对数据库中的列进行类型安全的访问，所以要创建它的子类。这个对象和基本表之间的关系如图 22-10 所示。

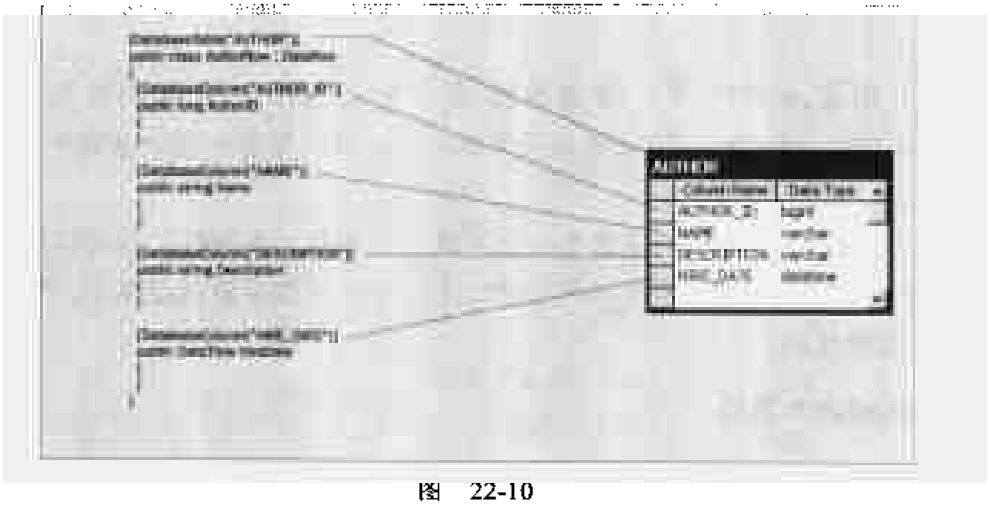


图　22-10

这个范例主要考虑的是 Books 和 Authors 表。它只由这两个表组成，如下面的几页所述。尽管这个范例使用的是 SQL Server，但可以修改代码，使之可用于 Oracle 或其他数据库引擎。

AuthorRow 类派生于 DataRow，且包含 Author 基本表中每个列的属性。DatabaseTable 属性被添加到行类上，每个链接到表中列上的属性现在都有一个 DatabaseColumn 属性。这些属性的一些参数被删除了，使图像能放在屏幕上。完整的内容将在下面几节中论述。

4. DatabaseTable 属性

这个范例的第一个属性用数据库表名标记一个类实例 DataRow，该数据库表将保存 DataRow。范例代码在目录 Chapter22/DatabaseAttributes 下：

```
// Excerpt from DatabaseAttributes.cs
/// <summary>
/// Attribute to be used on a class to define which database table is used
/// </summary>
[AttributeUsage (AttributeTargets.Class , Inherited = false ,
                 AllowMultiple=false)]
public class DatabaseTableAttribute : Attribute
{
  /// <summary>
  /// Construct the attribute
  /// </summary>
  /// <param name="tableName">The name of the database table</param>
  public DatabaseTableAttribute (string tableName)
  {
    TableName = tableName;
  }
```

```
    /// <summary>
    /// Return the name of the database table
    /// </summary>
    public readonly string TableName;
}
```

属性由一个构造函数组成，该函数接受一个表名作为字符串参数，而该属性用 Inherited=false 和 AllowMultiple=false 修饰符来标记。任何子类都不能继承这个属性，且一个类只能链接一个表。

在这个属性类中，我们把表名存储为一个字段，而不是一个属性。这可以随人的不同而不同。在这个范例中，由于没有修改表名的方法，所以一个只读字段就足够了。如果喜欢使用属性，就可以修改范例代码。

5. DatabaseColumn 属性

这个属性放在 DataRow 类的公共属性上，用于定义要链接的属性的列名，以及该列是否可以包含空值：

```
// Excerpt from DatabaseAttributes.cs
/// <summary>
/// Attribute to be used on all properties exposed as database columns
/// </summary>
[AttributeUsage (AttributeTargets.Property , Inherited=true ,
                        AllowMultiple=false) ]
public class DatabaseColumnAttribute : Attribute
{
  /// <summary>
  /// Construct a database column attribute
  /// </summary>
  /// <param name="column">The name of the column</param>
  /// <param name="dataType">The datatype of the column</param>
  public DatabaseColumnAttribute (string column , ColumnDomain dataType)
  {
    ColumnName = column;
    DataType = dataType;
    Order = GenerateOrderNumber ();
  }

  /// <summary>
  /// Return the column name
  /// </summary>
  public readonly string ColumnName;
  /// <summary>
  /// Return the column domain
  /// </summary>
```

```
    public readonly ColumnDomain DataType;
    /// <summary>
    /// Get/Set the nullable flag. A property might be better
    /// </summary>
    public bool Nullable = false;
    /// <summary>
    /// Get/Set the Order number. Again a property might be better.
    /// </summary>
    public int Order;
    /// <summary>
    /// Get/Set the Size of the column (useful for text columns).
    /// </summary>
    public int Size;

    /// <summary>
    /// Generate an ascending order number for columns
    /// </summary>
    /// <returns></returns>
    public static int GenerateOrderNumber ()
    {
      return nextOrder++;
    }

    /// <summary>
    /// Private value used whilst generating the order number
    /// </summary>
    private static int nextOrder = 100;
}

/// <summary>
/// Enumerated list of column data types
/// </summary>
public enum ColumnDomain
{
    /// <summary>
    /// 32 bit
    /// </summary>
    Integer,
    /// <summary>
    /// 64 bit
    /// </summary>
    Long,
```

```
        /// <summary>
        /// A string column
        /// </summary>
        String,
        /// <summary>
        /// A date time column
        /// </summary>
        DateTime
    }
```

这个类也用 AllowMultiple=false 来标记，因为在 DataRow 的一个属性和该属性链接的列之间总是一对一的关系。

把这个属性标记为可继承的，以便为数据库行创建一个类层次结构，因为模式中的每个表都有可能有类似的列。

构造函数带两个参数。第一个参数是要在数据库中定义的列名。第二个参数是 ColumnDomain 枚举中的一个值，在本例中，该枚举有 4 个值，但这对于产品代码来说是不够的。

该属性还有 3 个其他的特性，总结如下：

- Nullable——默认值为 false，当生成的列用于定义数据库值是否设置为 NULL 时，使用这个特性。
- Order——定义表中列的顺序号。生成表时，列以升序的顺序输出，默认为生成一个递增值，这是在构造函数中完成的。可以根据需要重写这个值。
- Size——定义字符串类中的最大字符数。

要定义一个 Name 列，可以使用下面的代码：

```
    [DatabaseColumn("NAME",ColumnDomain.String,Order=10,Size=64)]
    public string Name
    {
        get { return (string) this ["NAME"]; }
        set { this["NAME"] = value; }
    }
```

这就定义了一个字段 NAME，它是一个 VARCHAR(64)，因为列字段被设置为字符串，并设置了 Size 参数。它把顺序号设置为 10，本章后面会详细说明。该列也不允许为空，因为 Nullable 属性的默认值是 False(例如，该列生成为 NON NULL)。

DataRow 有一个索引符，它以一个字段名(或序数)作为其参数，返回一个对象，在上面的 getter 中，该对象在返回之前，会转换为一个字符串。

6. 创建数据库行

这个范例的要点是产生一系列强类型化的 DataRow 对象。在这个范例中，我们将创建两个类 Author 和 Book，它们都派生于同一个基类，因为它们拥有一些相同的字段。

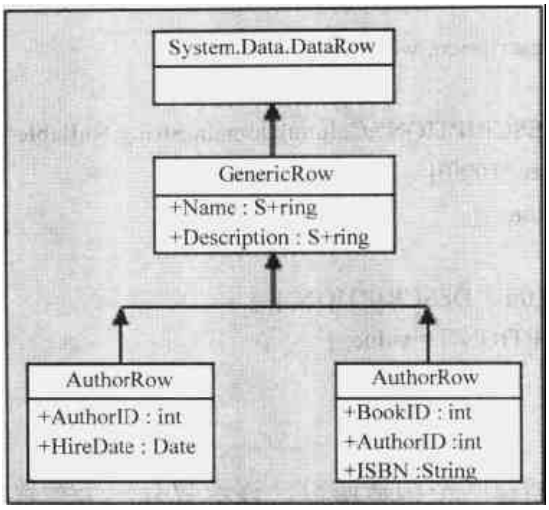


图 22-11

GenericRow 类定义了 Name 和 Description 属性，它的代码如下所示。该类派生于基类 DataRow，这是.NET Framework 中所有数据库行的基类。

在这个范例中，有两个类派生于 GenericRow，分别为 Author (AuthorRow)和 Book (BookRow)。这两个类还包含其他属性，它们链接到数据库的字段上：

```
// Excerpt from DatabaseTables.cs
/// <summary>
/// Base class row - defines Name and Description columns
/// </summary>
public abstract class GenericRow : DataRow
{
  /// <summary>
  /// Construct the object
  /// </summary>
  /// <param name="builder">Passed in from System.Data</param>
  public GenericRow (System.Data.DataRowBuilder builder)
    : base (builder)
  {
  }

  /// <summary>
  /// A column for the record name
  /// </summary>
  [DatabaseColumn("NAME",ColumnDomain.String,Order=10,Size=64)]
  public string Name
  {
    get { return (string) this["NAME"]; }
    set { this["NAME"] = value; }
  }
```

```
    /// <summary>
    /// A column for the description, which may be null
    /// </summary>
    [DatabaseColumn("DESCRIPTION",ColumnDomain.String,Nullable=true,Order=11,
                    Size=1000)]
    public string Description
    {
      get { return (string) this["DESCRIPTION"]; }
      set { this["DESCRIPTION"] = value; }
    }
  }
```

派生于 DataRow 需要创建一个构造函数，该函数有一个参数 DataRowBuilder。这个类在 System.Data 装配件内部。

接着定义两个属性 Name 和 Description，每个属性都有一定的特性。Name 属性的特性如下：

```
[DatabaseColumn("NAME",ColumnDomain.String,Order=10,Size=64)]
```

这把列名定义为 NAME，把它的域定义为长度为 64 的字符串，其顺序号设置为 10。这样做的原因是在创建数据库表时，我总是在定义表中的其他字段前，先定义主键码字段。把这个值设置为 10，可以给更多相同的字段留下一定的空间。在主键码中，如果字段多于 10 个，就需要重新设计！

description 列也有名称、字段和长度。将 Nullable 属性设置为 true，就不必定义一个 description 列了。另一种方式是定义一个“默认的”属性，并把它设置为一个空字符串，避免在数据库中使用 NULL。将顺序号设置为 11，则在生成的模式中，name 和 description 列总是保存在一起的：

```
[DatabaseColumn("DESCRIPTION",ColumnDomain.String,Nullable=true,
                                                    Order=11,Size=1000)]
```

每个属性存取器都为属性值定义了 get 和 set 方法，它们都是强类型化的，因此在字符串列中，会把一个字符串值返回给调用程序：

```
get { return (string) this["NAME"]; }
set { this["NAME"] = value; }
```

这里的代码有一些重复，因为属性定义了列名，所以可以在方法中使用反射提取相应列的值。但是，反射不是 API 最高效的方式，因为这些类用于尽可能快地访问基本的列。要提高这些存取器的性能，可以使用列的许多索引，因为使用字符串将涉及到在一个散列表中查找许多索引值。使用各种索引符时要小心，因为它们维护起来有点困难，特别是在定义了一个子类的实例中，就更是如此。

Author 行的构造过程如下：

```
// Excerpt from DatabaseTables.cs
/// <summary>
```

```
/// Author table, derived from GenericRow
/// </summary>
[DatabaseTable("AUTHOR")]
public class AuthorRow : GenericRow
{
  public AuthorRow (DataRowBuilder builder)
    : base (builder)
  {
  }

  /// <summary>
  /// Primary key field
  /// </summary>
  [DatabaseColumn("AUTHOR_ID",ColumnDomain.Long,Order=1)]
  public long AuthorID
  {
    get { return (long) this["AUTHOR_ID"]; }
    set { this["AUTHOR_ID"] = value; }
  }

  /// <summary>
  /// Date the author was hired
  /// </summary>
  [DatabaseColumn("HIRE_DATE",ColumnDomain.DateTime,Nullable=true)]
  public DateTime HireDate
  {
    get { return (DateTime) this["HIRE_DATE"]; }
    set { this["HIRE_DATE"] = value; }
  }
}
```

这里为 GenericRow 类创建了一个子类，并添加 AuthorID 和 HireDate 属性。注意为 AUTHOR_ID 列所选择的顺序号，设置它之后，就可以在发布的表中把它显示为第一个列。HireDate 属性没有这样的顺序号，所以它的值由属性生成，这些生成的值都是从 100 开始的，因此表的布局是 AUTHOR_ID, NAME, DESCRIPTION，最后是 HIRE_DATE。

BookRow 类也派生于 GenericRow，包含 Name 和 Description 属性。它添加了 BookID、PublishDate 和 ISBN 属性：

```
// Excerpt from DatabaseTables.cs
/// <summary>
/// Table for holding books
/// </summary>
[DatabaseTable("BOOK")]
public class BookRow : GenericRow
{
```

```
    public BookRow (DataRowBuilder builder)
      : base (builder)
    {
    }

    /// <summary>
    /// Primary key column
    /// </summary>
    [DatabaseColumn("BOOK_ID",ColumnDomain.Long,Order=1)]
    public long BookID
    {
      get { return (long) this["BOOK_ID"]; }
      set { this["BOOK_ID"] = value; }
    }

    /// <summary>
    /// Author who wrote the book
    /// </summary>
    [DatabaseColumn("AUTHOR_ID",ColumnDomain.Long,Order=2)]
    public long AuthorID
    {
      get { return (long) this["AUTHOR_ID"]; }
      set { this["AUTHOR_ID"] = value; }
    }

    /// <summary>
    /// Date the book was published
    /// </summary>
    [DatabaseColumn("PUBLISH_DATE",ColumnDomain.DateTime,Nullable=true)]
    public DateTime PublishDate
    {
      get { return (DateTime) this["PUBLISH_DATE"]; }
      set { this["PUBLISH_DATE"] = value; }
    }

    /// <summary>
    /// ISBN for the book
    /// </summary>
    [DatabaseColumn("ISBN",ColumnDomain.String,Nullable=true,Size=10)]
    public string ISBN
    {
      get { return (string) this["ISBN"]; }
      set { this["ISBN"] = value; }
    }
  }
```

7. 生成 SQL

定义了数据库行之后，代码就应从这些类中生成一个数据库模式了。本例把结果输出到控制台上，我们也可以通过在命令提示行上运行.exe，把结果输出到一个文本文件中。

下面的类为每个要创建数据库表的类调用 OutputTable：

```
public class DatabaseTest
{
    public static void Main ()
    {
        OutputTable (typeof (AuthorRow));
        OutputTable (typeof (BookRow));
    }
    public static void OutputTable (System.Type t)
    {
        // Code in full below
    }
}
```

可以利用反射循环装配件中的每个类，检查它是否派生于 GenericRow，并自动输出类。为了简单起见，我们对要生成的表名 AuthorRow 和 BookRow 进行硬编码。

OutputTable()方法如下所示：

```
// Excerpt from Database.cs
/// <summary>
/// Produce SQL Server style SQL for the passed type
/// </summary>
/// <param name="t"></param>
public static void OutputTable (System.Type t)
{
  // Get the DatabaseTable attribute from the type
  object[]  tableAttributes = t.GetCustomAttributes
                  (typeof (DatabaseTableAttribute) , true) ;

  // Check there is one...
  if (tableAttributes.Length == 1)
  {
    // If so output some SQL
    Console.WriteLine ("CREATE TABLE {0}" ,
      ((DatabaseTableAttribute)tableAttributes[0]).TableName);
    Console.WriteLine ("(");
    SortedList columns = new SortedList ();

    // Roll through each property
    foreach (PropertyInfo prop in t.GetProperties ())
    {
```

```
            // And get any DatabaseColumnAttribute that is defined
            object[]   columnAttributes = prop.GetCustomAttributes
              (typeof (DatabaseColumnAttribute) , true);

            // If there is a DatabaseColumnAttribute
            if (columnAttributes.Length == 1)
            {
              DatabaseColumnAttribute dca = columnAttributes[0]
                                                  as DatabaseColumnAttribute;

              // Convert the ColumnDomain into a SQL Server data type
              string   dataType = ConvertDataType (dca);

              // And add this column SQL into the sorted list - I want the
              // columns to come out in ascending order of order number
              columns.Add (dca.Order, string.Format ("   {0, - 31}{1, - 20}{2,8},",
                dca.ColumnName ,
                dataType ,
                dca.Nullable ? "NULL" : "NOT NULL"));
            }
          }
          // Now loop through the SortedList of columns
          foreach (DictionaryEntry e in columns)
            // And output the string...
            Console.WriteLine (e.Value);

          // Then terminate the SQL
          Console.WriteLine (")");
          Console.WriteLine ("GO");
          Console.WriteLine ();
        }
      }
```

这段代码反射了传递过来的类，并查找 DatabaseTable 属性。如果找到了 DatabaseTable 属性，就把 CREATE TABLE 子句写到控制台上，包括属性中的表名。

然后迭代类中的所有属性，以查找 DatabaseColumn 属性。有这个特性是所有属性都会成为已生成表中的一列：

```
            foreach (PropertyInfo prop in t.GetProperties ())
            {
              object[] columnAttributes = prop.GetCustomAttributes (
                                  typeof (DatabaseColumnAttribute) , true);
```

列的字符串表示是通过调用 ConvertDataType()方法来构造的，该方法如下所示。该字符串表示存储在一个排好序的集合中，这样列就会根据属性的 Order 特性值来生成。

在循环了所有的属性，并在排好序的列表中创建项目后，循环这个排好序的列表，并把每个值写到控制台上：

```
foreach (DictionaryEntry e in columns)
    Console.WriteLine(e.Value);
```

最好添加一个闭合括号和一个 GO 命令，指示 SQL Server 执行语句块，并创建表。

这个装配件中的最后一个函数把 ColumnDomain 枚举中的值转换为数据库特定的数据类型。另外，对于字符串列，我们创建了列的表示方式，使之包含该列的长度，例如基类中的 Name 属性构造为 VARCHAR(64)。这个列类表示长度至多为 64 个字符的可变字符数组。

```
// Excerpt from Database.cs
/// <summary>
/// Convert a ColumnDomain to a SQL Server data type
/// </summary>
/// <param name="dca">The column attribute</param>
/// <returns>A string representing the data type</returns>
private static string ConvertDataType (DatabaseColumnAttribute dca)
{
  string dataType = null;

  switch (dca.DataType)
  {
    case ColumnDomain.DateTime:
    {
      dataType = "DATETIME";
      break;
    }
    case ColumnDomain.Integer:
    {
      dataType = "INT";
      break;
    }
    case ColumnDomain.Long:
    {
      dataType = "BIGINT";
      break;
    }
    case ColumnDomain.String:
    {
      // Include the size of the string...
      dataType = string.Format ("VARCHAR({0})" , dca.Size);
      break;
    }
  }
```

```
    return dataType;
  }
```

对于枚举中的每个成员，我们创建一个适合于 SQL Server 的列字符串。这个范例中 Author 和 Book 类的 SQL 如下所示：

```
CREATE TABLE AUTHOR
(
    AUTHOR_ID              BIGINT              NOT NULL,
    NAME                   VARCHAR(64)         NOT NULL,
    DESCRIPTION            VARCHAR(1000)       NULL,
    HIRE_DATE              DATETIME            NULL,
)
GO

CREATE TABLE BOOK
(
    BOOK_ID                BIGINT              NOT NULL,
    AUTHOR_ID              BIGINT              NOT NULL,
    NAME                   VARCHAR(64)         NOT NULL,
    DESCRIPTION            VARCHAR(1000)       NULL,
    PUBLISH_DATE           DATETIME            NULL,
    ISBN                   VARCHAR(10)         NULL,
)
GO
```

这个 SQL 会对一个空白的或已有的 SQL Server 数据库运行代码，创建表。创建出来的 DataRow 类可以提供对这些表中的数据进行类型安全的访问。

要利用派生的 DataRow 类，需要提供如下所示的代码。这个类重写了 DataTable 中的一些函数，并被传递给构造函数中行的类型：

```
// Excerpt from DatabaseTables.cs
/// <summary>
/// Boilerplate data table class
/// </summary>
public class MyDataTable : DataTable
{
  /// <summary>
  /// Construct this object based on a DataRow
  /// </summary>
  /// <param name="rowType">A class derived from DataRow</param>
  public MyDataTable (System.Type rowType)
  {
    m_rowType = rowType;
    ConstructColumns ();
```

```
}

/// <summary>
/// Construct the DataColumns for this table
/// </summary>
private void ConstructColumns ()
{
  SortedList columns = new SortedList ();

  // Loop through all properties
  foreach (PropertyInfo prop in m_rowType.GetProperties ())
  {
    object[]  columnAttributes = prop.GetCustomAttributes
        (typeof (DatabaseColumnAttribute) , true);

    // If it has a DatabaseColumnAttribute
    if (columnAttributes.Length == 1)
    {
      DatabaseColumnAttribute dca = columnAttributes[0]
                                          as DatabaseColumnAttribute;

      // Create a DataColumn
      DataColumn  dc = new DataColumn (dca.ColumnName ,
                                          prop.PropertyType);
      // Set its nullable flag
      dc.AllowDBNull = dca.Nullable;
      // And add it to a temporary column collection
      columns.Add (dca.Order , dc);
    }
  }

  // Add the columns in ascending order
  foreach (DictionaryEntry e in columns)
    this.Columns.Add (e.Value as DataColumn);
}
/// <summary>
/// Called from within System.Data
/// </summary>
/// <returns>The type of the rows that this table holds</returns>
protected override System.Type GetRowType ()
{
  return m_rowType;
}

/// <summary>
```

```
    /// Construct a new DataRow
    /// </summary>
    /// <param name="builder">Passed in from System.Data</param>
    /// <returns>A type safe DataRow</returns>
    protected override DataRow NewRowFromBuilder (DataRowBuilder builder)
    {
      // Construct a new instance of my row type class
      return (DataRow) Activator.CreateInstance (GetRowType(),
                                                 new object[1] { builder });
    }

    /// <summary>
    /// Store the row type
    /// </summary>
    private System.Type m_rowType;
}
```

构造函数中调用了 ConstructColumns()函数，该函数将为 DataTable 生成一个 DataColumns 数组，它们也使用反射来获取。其他方法 GetRowType() 和 NewRowFromBuilder()重写基类 DataTable 中的方法。

有了这个派生的 MyDataTable 类后，就可以在自己的代码中轻松地使用它了。下面的范例给 Author 表添加两个作者记录，并把这些行输出到一个 XML 文件中：

```
DataSet         ds = new DataSet ();
MyDataTable t = new MyDataTable (typeof (AuthorRow));

ds.Tables.Add (t);
AuthorRow       author = (AuthorRow)t.NewRow ();

author.AuthorID = 1;
author.Name = "Me";
author.HireDate = new System.DateTime (2000,12,9,3,30,0);

t.Rows.Add (author);

author = (AuthorRow) t.NewRow ();
author.AuthorID = 2;
author.Name = "Paul";
author.HireDate = new System.DateTime (2001,06,06,23,56,33);

t.Rows.Add (author);

t.DataSet.WriteXml (@".\authors.xml");
```

运行这段代码，得到如下结果：

```
<?xml version="1.0" standalone="yes"?>
<NewDataSet>
  <Table1>
    <AUTHOR_ID>1</AUTHOR_ID>
    <NAME>Me</NAME>
    <HIRE_DATE>2000-12-09T03:30:00.0000000-00:00</HIRE_DATE>
  </Table1>
  <Table1>
    <AUTHOR_ID>2</AUTHOR_ID>
    <NAME>Paul</NAME>
    <HIRE_DATE>2001-06-06T23:56:33.0000000+01:00</HIRE_DATE>
  </Table1>
</NewDataSet>
```

这个范例说明了如何在代码中使用定制属性。如果没有考虑把数据库结构与访问该数据库的类结合起来，这就是一个很好的起点。如果模式改变的不是很频繁，就可以把数据库表与类结合起来，但对于较动态的后端来说，最好使数据访问类与它们访问的数据库表同步起来。

从完整的实现角度来看，我们还可以包含其他属性，来定义下述全部或部分内容：

- 主键码列
- 约束 —— 外键码和检查
- 版本 —— 每个列属性和表属性的版本号，这将简化升级脚本的生成，实际上可以根据属性生成整个升级脚本
- 列的默认值

22.5 小结

本章描述了属性的定义，讨论了.NET Framework 中定义的一些属性。.NET Framework 中还有许多属性类，了解它们的用途的最佳方式是参阅 MDSN 文档说明。

我们讨论了可以放在装配件中的属性，为最终用户详细介绍了文件属性对话框，使用 ildasm 探讨属性如何存储在装配件中。我们还阐述了反射的主题，说明了如何从代码中读取属性，接着讨论了一些内置的属性，例如 Conditional, Obsolete 和 Serializable。

接着介绍了如何创建定制属性，并论述了 AttributeUsage 属性。我们使用前面描述的许多策略创建了一个 BugFix 属性。最后开发了一个比较长的范例，以此来说明从类中生成数据库模式信息。

第23章 Web编程基础

23.1 引言

在接下来的3章中，我们将介绍World Wide Web的编程。这是一个巨大的课题，但本章将提供一些难题的基础知识，以及令人惊异的机会。后面的第24章将深入讨论ASP.NET的开发，但首先回顾一下Internet，特别是World Wide Web的简史。

本章将学习：

- Internet是如何工作的
- HTML的基础和HTML窗体
- 静态和动态Web页之间的区别
- 客户端和服务器端脚本的区别
- 为什么ASP.NET是开发cutting-edge Web解决方案的最佳选择

23.2 Internet简史

Internet最初是在20世纪60年代作为一个美国国防部项目来开发的，该项目叫作ARPANET(高级研究计划局网络，Advanced Research Projects Agency Network)。其目的是把散布在西海岸的各个计算机连接起来，让它们可以彼此通信，即使在遭遇核武器攻击时也能通信。那时，计算机非常少，也非常昂贵，只有大学和政府才买得起。把几台现有的计算机连接在一起，共享信息和数据，可以使每台计算机更有价值。在网络上通信的协议被设计为可以经受住网络的破坏，这样因核武器攻击而导致一个或多个站点不能使用时，网络仍可以发挥作用。

ARPANET开发的最终结果是TCP/IP (传输控制协议/Internet协议，Transmission Control Protocol/Internet Protocol)。这是通过电话线、光纤电缆和卫星信号传送计算机数据和信息的协议。TCP/IP可以让不同的计算机通过网络彼此通信，它为今天的Internet铺平了道路。

在20世纪70和80年代，Internet仍主要是学者和政府的特权。随着计算机价格的不断降低，以及个人计算机进入主流市场，Internet才开始发展壮大。90年代World Wide Web的出现，使Internet呈几何级数增长。目前许多人都使用他们的计算机访问Web，因此大多数公司都在使用Web销售或促销产品和服务，Internet，特别是World Wide Web改变了我们做生意的方式，这并不是一种保守的看法。

23.2.1　World Wide Web

快速测验：　Internet 和 World Wide Web 有什么区别?

如果回答没有区别，请继续往下看。Internet 是把全球计算机都连接起来的一个网络，上百万台服务器存储了数以十亿计的 Web 页、电子邮件组、讨论列表和 FTP 站点。Internet 是在线的所有东西，包括电子邮件、FTP、新闻组、Gopher 站点和聊天程序等。World Wide Web 则出现得比较晚，第一个 Web 服务器出现于 1991 年，但它现在已经是 Internet 是一个主要部分。Web 组成了连接的 Web 页，其中包含文本、图形和动画。World Wide Web 的一个关键组件是超链接——文本或图形链接，它允许用户导航到其他 Web 页上，这是通过 HyperText Markup Language(超文本标记语言，HTML)实现的，详见后面的内容。

在 Web 之前，Internet 主要是基于文本的。Web 提供了一种可交互操作、对用户非常友好的环境，因此吸引了绝大多数用户。使用 Internet 获得信息不再是学者和教授的特权。在设计下一个 Web 站点时记住这些！本书主要介绍 World Wide Web 的编程。

23.2.2　Web 的工作原理

Web是客户机－服务器系统的一个范例。Web上的所有内容都被分为服务器或客户机部分。

1. 服务器

World Wide Web 上的所有内容存储在世界某个地方的服务器上。服务器是运行在计算机上的一个应用程序，提供了另一个人、应用程序或计算机需要的信息、文件或数据。在 Web 页上，Web 服务器响应服务器上包含的 Web 页的请求。客户机给 Web 服务器发送页面的请求，Web 服务器再把这个页面发送给客户机，进行浏览。

服务器通过 Uniform Resource Locator(统一资源定位符，URL)定位。这是 Web 页在 World Wide Web 上的特殊地址，它指向存储在服务器上的站点和请求的 Web 页。例如 URL: http://www.microsoft.com/ms.htm指向 Internet 上的服务器www.microsoft.com，以及该 Web 站点的 ms.htmWeb 页。URL 与文件系统的路径相同，可以包含子文件夹，以进一步浏览站点的内容。

所有的 Web 站点都有一个辨识自己的独特的 Internet Protocol (IP)地址，这个地址由 4 个数字序列组成，其格式是 200.123.45.1(它们用句点分隔开，数字可以是 0 到 255 之间的任何数)。幸运的是，我们不需要记忆要访问的 Web 站点的 IP 地址，有一个特殊的服务器 Domain Name Services(域名服务程序，DNS)会自动处理 URL 和 IP 地址的转换。这些 DNS 服务器可以存在于自己的网络中，也可以位于 Internet 上。一个 Web 服务器可以拥有好几个不同的 Web 站点，每个 Web 站点都有不同的 URL，认识到这一点是很重要的。此时，Web 站点的 IP 地址都指向相同的机器。Web 服务器应用程序管理着每个 Web 站点的客户请求。

2. 客户程序

客户程序是运行在计算机上的一个软件。最常用的 Internet 客户程序是一个浏览器，它是一种专用程序，允许用户键入地址，或使用 Web 页上的超链接，导航到新页面上。它处理到 Web 服务器的请求，显示服务器返回的 Web 页。目前，最常用的两个浏览器是 Netscape 的

Navigator 和 Microsoft 的 Internet Explorer。

下面的范例说明了客户请求如何发送给服务器，以及如何返回一个 Web 页响应。这个响应的格式是 HTML 页面，显示在客户机的浏览器上，如图 23-1 所示。

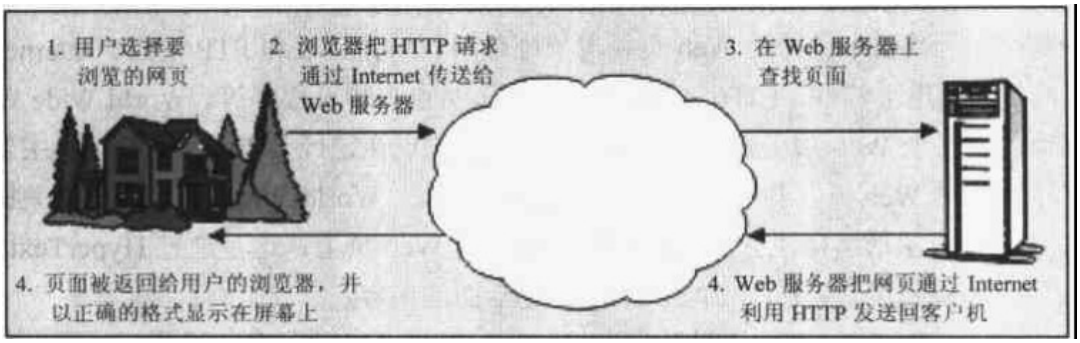


图　23-1

23.2.3　HTTP – 超文本传输协议

对于用户来说，发送和接收数据都是明确处理的。他们看到的是浏览器窗口显示一个 Web 页，如果出了错，就显示错误信息。而对于开发人员来说，就需要更深入地理解处理通信过程。前面讨论了 TCP/IP，这个协议可以通过电话线发送数据，并处理数据发送至目的地的过程。另一个需要记住的缩写词是 HTTP，它表示超文本传输协议，这个协议指定如何处理数据。这是一个应用程序协议，允许浏览器和服务器相互通信，来回传送数据。Web 页的所有请求和服务器发送的所有响应都是在浏览器和服务器之间传送的 HTTP 信息。

1. HTTP 信息的类型

为了从 Web 服务器获取一个资源，例如 Web 页，客户机应给 Web 服务器发送一个 HTTP 请求信息。这个请求信息包含要请求的资源名称(例如 Web 页文件名)，和指定客户机的其他信息的数据(称为标题)。Web 服务器处理该请求，并给客户机编写一个 HTTP 响应信息。这个响应信息包括请求的状态，响应包含的其他标题，以及请求的数据。如果因某种原因请求失败，该失败信息就会在状态代码中报告给客户机，不返回任何数据。

记住，在大多数情况下，我们看不到这些信息。浏览器会创建并传输请求信息，处理服务器的 HTTP 响应信息。但理解可以选择的不同选项的含义是很重要的。

- HTTP GET

最常见的 HTTP 请求是 GET 命令。它用于向服务器请求一个 Web 页文件或其他资源。GET 选项的格式非常简单，它包括要从服务器获取的资源名称和 0 或多个标题行。下面是一个有效 GET 请求的范例：

```
GET /content/booklistings.html HTTP/1.0
From: user@wrox.com
User-Agent: Mozilla/4.0
```

/content/booklistings.html 是从 Web 服务器请求的一个特定资源。服务器文件夹的导航从根文件夹开始，因此，如果这个请求发送至 www.wrox.com，就会获取 URL 为 http: //www.

wrox.com/content/booklisting.html 的 Web 页。From 和 User-Agent 行是与请求信息一起发送的常见 HTTP 标题。

- HTTP POST

客户机发送给 Web 服务器的请求信息的另一个常见形式是 POST 命令，在把数据的有效负载发送给服务器进行处理时，常常使用这个命令。这常常用于传送 HTML 格式的数据，这种数据详见后面的内容。HTTP POST 信息看起来非常类似于 HTTP GET，但大多数情况下，指定的资源是一个服务器应用程序，或活动 Web 页(这是 Web 页的一种特殊类型，它会动态生成 HTTP，以响应传送给它的数据)。请求还必须包含要传送给服务器的数据的有效负载，这与 GET 信息不同。下面是一个发送给 Web 服务器的 POST 请求的范例：

```
POST /content/GetBooks.asp HTTP/1.0
From: user@wrox.com
User-Agent: Mozilla/4.0
Content-Type: text/plain
Content-Length: 33

Here is some data for the server.
```

这个请求被发送给服务器上的动态 Active Server Page (ASP)文件，它处理发送过来的数据，并给客户机返回一个动态生成的 HTML 页面。这个请求在信息中指定了两个额外的 HTTP 标题：Content-Type 和 Content-Length。Web 服务器使用这些标题处理 POST 命令传送过来的数据。传送给服务器的实际数据是字符串“Here is some data for the server”。

- 查询字符串

给服务器应用程序传送数据的另一种方式是使用 GET 命令在查询字符串中传送信息。这是一个简单的 URL 请求，把数据直接放追加在该 URL 的后面。当只有少量文本信息传送给服务器时，这是很有效的。数据以名称/值对的形式放在 URL 的后面，用一个问号把地址和数据分隔开来，用一个&符号把每个名称/值对分隔开来。下面的范例是以查询字符串的格式把数据追加在 HTTP GET 请求后：

```
GET /Books/Book_Details.asp?isbn=1861004982&order=title HTTP/1.0
From: user@wrox.com
User-Agent: Mozilla/4.0
```

这些数据也是使用 URL 中指定的 isbn 和 order 字段被发送到 ASP 文件中的。使用 GET 方法给服务器应用程序发送数据非常简单，也易于完成，但有一定的限制。只能发送文本数据，可以放在查询字符串中的数据量也很少。如果要发送的不仅仅是几个数据字段，就必须使用 POST 命令。另外，查询字符串中的数据在连接时完全可见，所以有安全风险的任何数据(例如信用卡信息或密码)，都不应包含在查询字符串中。

2. HTTP 标题

在前面的范例中介绍了几个 HTTP 标题。可以把 HTTP 标题看作是标准的文档标题：它不是实际页面或文档的一部分，但包含与文档相关的重要元数据信息。只要给客户机或服务器发

送了一个 HTTP 命令，就会有一系列标题与数据一起发送出来。每个标题基本上都是一个指定的属性，它包含标题名和其后的标题值。HTTP 标题的一些范例如表 23-1 所示。

表 23-1

标 题 名	描　述
From	客户机电子邮件标识，在用户的浏览器设置中指定
User-Agent	客户机用于发送 HTTP 请求的软件类型和版本
Content-Type	包含在该 HTTP 信息体中的数据类型
Content-Length	包含在该 HTTP 信息体中的数据长度
Date	编写该 HTTP 信息的日期和时间

23.3 Web 编程 101

这是一本编程书籍，所以首先编写一些代码。我们将开发 Web 页，从最简单的开始，到最终会使用 ASP.NET。它将有助于理解多年来 Web 开发的演变过程，理解为什么 ASP.NET 是这样一个强大的工具。有经验的读者可以跳过自己已经熟悉的章节。

23.3.1 HTML

HTML 即超文本标记语言，是显示 Web 页的语言。它指定了 Web 页的外观和布局：图形放在什么地方，文本的外观如何，页面上还可以有超链接。基本上，在 Web 上看到的每个 Web 页都是 HTML 文档。这些文档位于一个 Web 服务器上，当客户机请求 Web 页时，服务器就通过 HTTP 给客户机发送该 Web 页。

有关以 HTML 编程的书籍非常多，本书不打算详细讨论这个主题。要学习如何使用 HTML 制作绝佳的 Web 页，请参阅 Wrox 出版社的《HTML 4.01 编程指南》(ISBN 1861005334)。

1. 标记

HTML 由一组定义页面布局的标记组成。这些标记用尖括号括起来，以便浏览器轻松地识别他们。浏览器使用标记正确地显示和格式化括起来的文本。例如，下面的代码行会以黑体字显示文本“Hello World”：

```
<b>Hello World</b>
```

对于大多数标记来说，必须有一个起始标记和结束标记。这可以从上面的代码看出，第一个表示黑体文本的开始，第二个标记表示这一行之后的其他文本不是黑体。斜杠字符/用于表示标记的结束。

2. 基本的 HTML

HTML 页面的惟一规则是它们必须包含一个<html>标记。因此，下面的代码行是有效的 Web 页：

```
<html></html>
```

显然，这个 Web 页没有什么用，因为它没有显示任何信息。其他标记必须放在这两个 HTML 标记中间。下面使用一些基本的 HTML 标记，创建一个比较有效、有趣的 Web 页。

试试看：一个简单的 HTML 页面

(1) 打开 Notepad，或合适的 ASCII 文本编辑器。

(2) 在文档中写入如下文本：

```
<html>
    <head>
        <title>An Easy HTML Page</title>
    </head>
    <body>
        <center>
            <h1>Hello Everyone</h1>
            <i>This text is in Italics</i><br>
            <b>Here text is in bold:</b>
            <h3><font color="blue">
                Please comment on this great site:
            </font></h3>
            <hr>
            <br>
        </center>
        <table Border="1">
            <tr>
                <td>Name</td>
                <td><input type="text" name="Name"></td>
            </tr>
            <tr>
                <td>Comments:</td>
                <td>
                    <textarea name="Comments" rows=3 cols=65 wrap></textarea>
                </td>
            </tr>
        </table>
    </body>
</html>
```

(3) 把文件保存为 Comments.htm。

(4) 运行 Internet Explorer 或 Netscape Navigator，打开新保存的文件。其结果如图 23-2 所示：



图　23-2

代码的说明

尽管这是一个非常简单的页面，但说明了许多 HTML 编程的基础知识。文件的整个内容包含在<html></html>块中，告诉浏览器这是一个 HTML 文件。

下面的代码行说明如何设置 Web 页的标题。<title>标记必须放在一对<head>标记中。标题显示在 Web 浏览器窗口的顶部。<head>区域也可以包含其他页面信息，例如创建者及其内容，这些信息可以由搜索引擎或 Web 代理程序使用：

```
<head>
    <title>An Easy HTML Page</title>
</head>
```

页面的其余部分包含在一对<body>标记中。这个标记指定实际页面的内容，即浏览器客户区域中的内容：

```
<center>
    <h1>Hello Everyone</h1>
    <i>This text is in Italics</i><br>
    <b>Here the text is in bold:</b>
```

使用 标记可以使页面元素位于页面的中间。<h1>标记表示一个字体样式，称为“一级标题”。有 6 个标题级别，从 1 到 6，一级标题最大。这里使用<h1>样式来格式化页面的标题。下一行使用<i>标记，把括起来的文本格式化为斜体。<b>标记以类似的方式工作，但把文本格式化为黑体：

```
    <h3><font color=blue>Please comment on this great site:</font></h3>
```

接着使用<font>标记指定文本的颜色。注意文本 color=blue 包含在<font>标记中。许多标记都有可选的属性，它们可以包含在标记中，指定文本的其他信息。本例中属性是 color，指定括起来的文本颜色。这个属性的值可以是一个指定的值，例如 blue，也可以是一个 16 进制数，表示颜色的 RGB 值。每个不同的标记都有许多可选的属性，而设置属性的一般格式如下：

```
    <hr>
    <br>
```

```
</center>
```

下一行用<hr>标记在屏幕上绘制一条水平线，<hr>标记的正式名称是“水平规则(horizontal rule)”。
标记是一个换行符，表示结束当前行，把下一个可打印的字符移动到下一行上。注意<hr>和
都不需要结束标记，它们只有一个，没有匹配的标记，这是因为它们都不标记要显示的文本，仅在屏幕上输出某些内容。然后用 结束标记终止文本的居中：

```
<table border=1>
    <tr>
        <td>Name</td>
        <td><input type="text" name="Name"></td>
    </tr>
    <tr>
        <td>Comments:</td>
        <td><textarea name="Comments" rows=3 cols=65 wrap></textarea></td>
    </tr>
</table>
```

下面的代码行创建一个简单的表，其中表的各行中有一些文本框。下面将详细讨论如何创建一个交互的 HTML 页面。<table>标记告诉浏览器开始绘制一个表。在<table>标记之间有表行<tr>和表数据<td>标记，表行标记指定开始一个新行，表数据指定开始一个新列。处理 HTML 表有一些非常高级的格式化技巧；如果读者对 Web 开发很感兴趣，可以更多地了解它们。

文本框看起来非常类似于第 13 章中设计 WebForm 应用程序时使用的文本框。区别是现在是编写 HTML 来显示它们，而不是创建 TextBox 对象本身。<input>标记有许多属性可以设置，例如 type 确定它是文本框、单选按钮、命令按钮，还是复选框。<textarea>标记会创建一个更大、可滚动的文本框，它也有许多属性，可以设置控件的各种属性、列数和行数，以及文本是否打包等。

在浏览器中查看文件时可以看出，我们创建了一个简单的 Web 页。可以重新设置窗口的大小，在不同的浏览器中查看页面，在两个文本框中输入文本。这里没有提及图形和内容，但我们的目的是说明如何使用 HTML，而不是创建一个完整的 Web 站点。接下来，讨论如何让 Web 页给服务器提交数据。

23.3.2　HTML 窗体

我们的 Web 页已 OK，但用户如何提交页面的内容？目前还无法做到，用户只能在文本框中键入文本。

我们需要有一种把注释从页面的读者发送给我们自己的方式。在 HTML 中处理这个过程的方式是使用窗体。可以把 HTML 窗体看作类似于基于纸张的表格：最终用户必须填写它们包含的字段，并发送给正确的接收人或地址。

窗体的创建类似于大多数其他的 HTML 元素：使用一个标记<form>，这个标记用于把窗体放在页面上，</form>则结束该标记。这个标记不显示任何内容，仅标记窗体的开始和结束。这些标记之间的内容指定窗体的实际字段。

在<form>标记中有几个非常重要的、必须填写的属性，其基本语法如下：

```
<form name="myform" method="post" action="myform.asp">
```

name 属性标识窗体，在服务器中必须是惟一的，这样才能正确地处理窗体。Method 属性指定窗体数据如何发送给服务器。最后，action 属性告诉浏览器把窗体数据发送到什么地方。

Method 属性可以有两个值：GET 或 POST。读者应从前面讨论的 HTTP 请求中熟悉它们。窗体仅告诉浏览器在提交窗体时，给服务器传送了什么格式的 HTTP 请求。GET 方法会把窗体数据追加在资源地址的后面，创建一个用以分析服务器的查询字符串。POST 方法会把窗体数据作为 HTTP 信息的实际信息体发送给服务器。GET 方法对于只有文本的少量数据来说非常有效，而 POST 方法可以用于任何数量、任何格式的数据。

创建窗体需要的最后一个关键信息是指定提交窗体时在服务器上执行什么处理。这是用<form>标记的 action 属性指定的。目前，大多数情况下，这都会指向一个活动的 Web 页，它可以处理提交的数据，给用户输出某种类型的响应。下一节将结束如何把窗体数据直接发送给活动的页面，但现在我们仅通过电子邮件发送窗体数据。

过去，创建真正动态内容的惟一方式是使用 Common Gateway Interface (公共网关接口，CGI)，它允许应用程序驻流在 Web 服务器上，由 Web 页调用。这个应用程序可以用任何编程语言编写，但我们常常使用 C/C++ 或 Perl。窗体应被提交给该应用程序，该应用程序再返回得到的动态内容。但从服务器的角度来看，CGI 应用程序对处理器或资源并不友好，在许多情况下由服务器端的脚本工具替代，详见下一节。在许多 Web 应用程序中，HTML 窗体被提交给一个动态服务器端页面，处理用户的输入，并生成合适的响应页面。

我们指定用<form>标记的 action 属性应用程序或页面的位置，以使给它们提交窗体。这可以引用一个 CGI 应用程序、活动的 Web 页，甚或一个电子邮件地址。在后一种情况下，窗体通过电子邮件提交给指定的地址。

下面编写一些代码，看看这些窗体是如何工作的。

试试看：HTML 窗体

(1) 打开上一个范例中的 Comments.htm。

(2) 在现有的文档中添加如下代码：

```
<html>
    <head>
        <title>An Easy HTML Page</title>
    </head>
    <body>
        <center>
            <h1>Hello Everyone</h1>
            <i>This text is in Italics</i><br>
            <b>Here text is in bold:</b>
            <h3><font color=Blue>Please comment on this great site:</font></h3>
            <hr>
            <br>
```

```
        </center>
        <form method="post" enctype="text/plain"
              action="mailto:yourname@domain.com" name="CommentsForm">
          <table border="1">
            <tr>
              <td>Name</td>
              <td><input type="text" name="Name"></td>
            </tr>
            <tr>
              <td>Comments:</td>
              <td>
                <textarea name="Comments" rows=3 cols=65 wrap></textarea>
              </td>
            </tr>
          </table>
        <br>
        <input type="submit" value="Submit">
        </form>
      </body>
    </html>
```

(3) 在 action 属性中，在 mailto 的后面键入电子邮件地址。

(4) 把文档保存为 FormComments.htm。

(5) 在浏览器中打开 HTML 文件。填写一些数据，单击 Submit 按钮，如图 23-3 所示。



图　23-3

屏幕上什么也没有改变，但如果浏览器配置正确，应收到一封电子邮件。根据浏览器的不同，这封电子邮件应带有一个附件，其中包含了传送过来的数据，或者在电子邮件中包含这些数据。无论如何，得到的数据都应如下所示：

```
Name=Keith
Comments=This site is amazing!
```

代码的说明

可以看出，这个范例仅添加了3行代码，但它的功能却大大改变了。现在，终端用户可以查看Web页，以提供反馈。这是迈向交互Web页的第一步。

下面的代码开始窗体，并设置了在用户提交它时要做的工作：

```
<form method=post enctype="text/plain"
      action="mailto:yourname@domain.com" name="CommentsForm">
```

method属性指定这个窗体把一个HTTP POST命令和HTTP信息体中的窗体数据一起发送回服务器。action属性指定这些数据应发送到什么地方。在本例中，我们把窗体通过电子邮件传送给指定的地址。enctype属性指定在HTTP POST命令中要发送数据的类型。在大多数情况下，这是纯文本。name属性用于在JavaScript中引用窗体元素，稍后详细讨论。

在<form>标记之间，指定了格式化的表，以及现有的<input>和<textarea>标记。注意这两个标记的一个非常重要的属性是name属性，我们以一般格式给它赋值。当把窗体中的数据发送给服务器时，它的形式是一系列名称/值对，其中，名称是窗体字段的name属性，而值是用户输入的数据。因此，访问特定数据元素的惟一方式是必须知道窗体元素的名称。可以看出，在通过电子邮件接收到的窗体数据中，这是如何工作的：

```
Name=Keith
Comments=This site is amazing!
```

再次使用<input>标记创建Submit按钮，但这次把type属性指定为submit，这样，包含它的窗体就会执行其action属性，在本例中就是发送一个电子邮件。value属性说明如何定制按钮的文本，这与定制WinForm按钮是一样的。

窗体元素

在Web页中可以有各种各样的窗体元素。使用基本HTML，要比使用ASP.NET的限制更多，详见第24章。表23-2列出了基本HTML窗体中的不同窗体选项。

表　23-2

窗体元素	HTML标记	说　明
文本框	<input type="text">	它创建的文本框类似于WinForm应用程序中的文本框。用户可以在区域中键入文本，也可以使用标准的剪切和粘贴操作
密码文本框	<input type="password">	这个文本框中的文本显示为一系列隐藏的字符。注意这并不意味着在传输过程中隐藏这些字符，必须使用其他方式确保传输过程的安全性
多行文本框	<textarea>	如果需要多行文本或少量的文本，可以使用这种文本框的特殊形式。这个控件允许输入多行文本，提供标准文本框的编辑功能

(续表)

窗体元素	HTML 标记	说　明
单选按钮	<input type="radio">	如果要给用户显示几个选项，但只有一个选项是有效的，就可以使用单选按钮。用户一次只能选择一个单选按钮
复选框	<input type="checkbox">	复选框的操作方式类似于单选按钮，但允许用户从组中选择多个值。因此，对于“选择所有选项”的问题，复选框就是最佳选择
菜单	<select name="MenuName">	它允许创建一个菜单，让用户从一组选项中选择。每个菜单项都用一个额外的<option>标记来指定
隐藏	<input type="hidden">	隐藏字段永远不会显示给用户，但非常便于存储由服务器处理的数据。第 24 章将讨论 ASP.NET 如何使用隐藏的字段存储当前会话的信息
图像	<input type="image">	这将创建一个图像，单击该图像，就提交窗体。另外，对于其他指定的窗体数据，鼠标的水平和垂直位置也会发送给服务器，这样，服务器就可以根据用户单击图像时鼠标的位置返回内容。在许多 Web 页上很常见的一个范例是公司位置的地图。用户单击特定的城市或州，就可以指向带有该分支信息的页面
Submit 按钮	<input type="submit">	在默认状态下，它显示为一个按钮，但其外观可以改变。单击这个按钮时，就给服务器提交窗体和当前的数据
Reset 按钮	<input type="reset">	这将在窗体元素中删除所有的用户选择和文本，就好像用户刚刚进入该页面一样

注意所有的窗体元素必须有一个 name 属性，应给它赋一个有意义的名称。这样就可以从各个窗体元素中提取数据了。

23.3.3　静态和活动的 Web 页

前面介绍的所有页面都是静态的 Web 页。也就说，这些页面都是服务器上的 HTML 文件，只要 Web 开发人员没有修改 HTML 文件，它们就不会改变。在 World Wide Web 初期，大多数站点都是由一组内部连接的静态页面组成的。当内容需要改变时，Web 开发人员就必须遍历和修改 HTML 文件。对于产品列表要不断更新的站点和新闻站点来说，工作量非常大，几乎不可能使用静态 Web 页创建一个 Web 站点。

人们需要一种方式，这种方式使得 Web 页可随时创建，而每次内容有变化时，Web 开发人

员不需要更新页面。如果在后端数据库系统上有新内容添加到产品列表上，它们就会自动显示在 Web 站点上。当增加了新的新闻时，它们会自动显示出来。另外，把 HTML 用作数据库驱动系统的前端，系统就是通用的。使用浏览器的任何人都可以访问 Web 应用程序，而无需考虑操作系统、机器等。客户机只需要一个 Web 浏览器就可以访问数据，不需要安装客户应用程序。

不久，World Wide Web 流行起来，人们认识到活动 Web 页的重要性，并开始考虑解决方案。创建动态性更强的 Web 页的两种方式分别是利用客户和服务器端脚本。

1. 客户端脚本

要得到交互性和动态性更强的 Web 页，Web 页包含的一些代码块应包含在 HTML 页面中。这些代码不像以前我们所介绍的程序那样编译。脚本语言是在执行时被解释的。有了可以交互和运行在 Web 页上的代码，Web 程序员就有了比静态 Web 页更多的选择。可以在客户的计算机上执行某些有趣的操作，而无需与服务器接触。这非常有助于在把窗体提交给服务器之前，检查 Web 页上 HTML 控件的内容，或者给用户创建更丰富的用户界面。

大多数 Web 应用程序使用的脚本语言是 JavaScript，这种语言可以用于所有的商用浏览器，它提供了丰富的命令和对象。本书不深入讨论 JavaScript；如果读者想学习这种语言，可参阅 Wrox 出版社的《JavaScript 入门》(ISBN 1681004060)。

下面看看上一个 Web 页，添加一些客户端检证，这些检证将在提交 Web 页前执行。

试试看：JavaScript 的客户端脚本

(1) 打开上一个范例的 FormComments.htm。

(2) 在页面的顶部添加下述代码，这将添加一个 JavaScript 函数，验证页面上的窗体元素：

```
<html>
  <head>
    <title>An Easy HTML Page</title>
<script language="JavaScript">
<!--

function ValidateForm()
{
  if (document.CommentsForm.Name.value=="")
  {
    alert("Please enter a name!");
    document.CommentsForm.Name.focus();
    return false;
  }
  if (document.CommentsForm.eMail.value=="" ||
      document.CommentsForm.eMail.value.indexOf("@") == -1)
  {
    alert("Please enter a valid eMail address!");
    document.CommentsForm.eMail.focus();
    return false;
```

```
    }
    if (document.CommentsForm.Comments.value=="")
    {
        alert("Please enter some comments!");
        document.CommentsForm.Comments.focus();
        return false;
    }
    return true;
}

// -->
</script>

    </head>
```

(3) 现在修改下述代码行，以便在提交窗体时包含该 JavaScript 函数的引用：

```
<form method="post" enctype="text/plain"
        action="mailto:yourname@domain.com" name="CommentsForm"
        onSubmit="return ValidateForm()">
```

(4) 下面添加一个电子邮件窗体元素，以使得有效性验证更有趣一些。添加下述代码行，增加一个新的电子邮件文本框：

```
<table border=1>
    <tr>
        <td>Name</td>
        <td><input type="text" name="Name"></td>
    </tr>
    <tr>
        <td>Email</td>
        <td><input type="text" name="eMail"></td>
    </tr>
    <tr>
        <td>Comments:</td>
        <td><textarea name="Comments" rows=3 cols=65 wrap></textarea></td>
    </tr>
</table>
```

(5) 把页面保存为 JavaScriptForm.htm。

(6) 在自己喜欢的浏览器中加载该页面。输入不正确的数据，看看弹出的信息框。用正确的数据填写好 3 个文本框后，该窗体才会被提交，如图 22-4 所示。



图　22-4

代码的说明

JavaScript 用于在客户的计算机上执行任务。这包括有效性验证(如本例所述)，处理页面元素，创建更动态的用户界面等。JavaScript 还带有一些非常酷的工具，但这不是本章的重点，我们仅介绍它的一个简单用法。使用 JavaScript 的优点是它可以省去 Web 服务器的一个循环，节省了时间和带宽，并可以一直在客户的计算机上进行处理，而无需使用服务器的宝贵资源。

在这个范例中，我们在窗体上添加了一些简单的有效性验证。JavaScript 是一种编程语言，其语法类似于 C#，因为它们都来源于 C，所以大多数代码都很容易理解。我们创建了一个函数 ValidateForm()，这个函数检查 3 个文本框，看看它们是否包含有效的数据。注意我们把该函数放在<script>块中，告诉浏览器它应把其中的文本解释为 JavaScript 命令。在 language 属性中设置语言类型，我们还用<!-- 和 // -->符号，把所有的 JavaScript 命令放在一个 HTML 注释块中，确保不能处理 JavaScript 的旧浏览器会忽略括起来的 JavaScript 命令：

```
<script language="JavaScript">
<!--
```

函数本身检查 3 个窗体元素，看看它们是否包含有效的数据。JavaScript 通过 Document Object Model(文档对象模型，DOM)引用浏览器中的所有内容，包括 HTML 页面本身。DOM 指定了一个 16 进制的系统，用以定位页面上的所有元素，允许处理、查询和更新这些元素：

```
if (document.CommentsForm.Name.value=="")
{
    alert("Please enter a name!");
    document.CommentsForm.Name.focus();
    return false;
}
```

if 语句使用 DOM 访问特定的窗体元素。它使用窗体名 CommentsForm 和窗体元素名 Name 缩小范围，直到只有一个元素为止。然后它使用该元素的一个属性 value 获取其文本。把这个

文本与空字符串进行比较，如果返回 true，就显示一个错误信息，使用窗体元素对象的 focus() 方法设置焦点。

另一个比较的操作过程与此类似。电子邮件的有效性验证略复杂一些，因为它要检查文本框是否为空，同时还要检查其中是否有一个@符号。为了便于进行这样的检查，它使用 indexOf() 方法，这个方法返回传送给它的字符位置，如果没有找到该字符，就返回 - 1。我们检查该方法是否返回 - 1，如果是，就显示一个错误信息。

在改变窗体标记，提交窗体时，调用该函数：

```
<form method="post" enctype="text/plain"
action="mailto:yourname@domain.com" name="CommentsForm"
onSubmit="return ValidateForm()">
```

提交窗体后，就执行 onSubmit 动作。在本例中，我们已经定义了它，以便调用刚才所讨论的 ValidateForm()函数。如果 ValidateForm()函数返回 false，那么窗体就不会被提交给服务器，这就防止了在有效性验证中发生错误时，把窗体提交给服务器。

大多数窗体对象都有类似于 onSubmit 的事件，激发该事件后，就执行定制的 JavaScript 代码。其中一些常用的事件如表 23-3 所示，但它们可能不能用于所有的对象。

表　23-3

事 件 名	说　明
OnFocus	当第一次加载浏览器时，以及用户切换到 Windows 中的浏览器时激发该事件。在用户设置控件的焦点时，也激发这个事件
OnLoad	在为页面加载了所有图像和文本后，引发该事件。这个事件可以用于提供任何页面对象的初始化
OnResize	在用户重新设置了浏览器窗口的大小时，引发该事件
onUnload	在用户浏览新页面，或动态脚本开始给用户发送新内容时，引发该事件
onBlur	当用户离开一个控件时，引发该事件
onClick	当用户单击一个按钮时，引发该事件

2. 服务器端脚本

使用客户端脚本仍不能解决创建真正由数据驱动的 Web 站点的基本问题。客户端脚本非常有用，通常用于执行特定的任务，但有时也需要服务器端脚本。

服务器端脚本表示在客户发出一个有助于建立返回页面的请求时，需要执行一些代码。这些代码可以从数据库中获取数据，根据返回的特定数据建立用户界面，再把 HTML 页面发送回客户机。最终的结果是发送回浏览器的一个标准 HTMLWeb 页，这就是创建动态 Web 页的过程，浏览器浏览的 Web 页实际上是在客户机发出请求时，所运行的代码，不是服务器上的一个物理页面。

服务器端脚本的另一个优点是减少了应用程序开发人员需要的维护时间。可以创建每个人都可以访问的 Web 应用程序，而不是给公司的每个桌面分布一个 Windows 应用程序。当应用程序需要更新和修改时，只需要更新一个基于服务器的应用程序，而不需要在每个客户机器上修改。这还允许每个客户在任何时刻都有应用程序的最新版本，因为它位于服务器上。

有许多工具和语言都可以生成活动的 Web 页。其中所使用的一些流行技术如下所示：

- Active Server Pages
- PHP
- ColdFusion
- WebSphere
- Java Server Pages

本书将讨论如何在.NET 环境中进行开发，所以下面介绍 Microsoft 对这个问题的解决方案：Active Server Pages。要 24 章将详细论述 Active Server Pages 的下一代 ASP.NET。

3. Active Server Pages

Active Server Pages 是 Microsoft 在服务器上创建动态 Web 页的一种技术，用于创建页面的语言是 VBScript，它是 Visual Basic 语言的一部分，工作方式非常类似于 Visual Basic，但所有的变量都必须是 variant 数据类型，且没有某些功能，例如文件 IO 功能。另外，因为它是一种脚本语言，这意味着它不进行编译，服务器在读取代码的同时执行它们，在性能方面与编译好的语言相比，有很大的损失。

这是一个很大的论题，我们在第 24 章详细介绍 ASP.NET，所以这里不深入论述它。如果读者希望熟练掌握这个技术，可参阅 Wrox 出版社的《Active Server Pages 3.0 入门》(ISBN 1861003382)。

4. 状态管理

状态是 Web 应用程序中要管理的一个非常复杂的内容。状态这个术语用于描述必须维护的永久性信息，这些信息不仅仅在使用或浏览一次 Web 站点时被维护。因为 HTTP 是一个无状态的协议，所以 Web 应用程序无法说明什么用户在浏览站点，或者这个用户是第一次浏览站点，还是一个回头客，这显然是一个非常重要的区别。

Cookies 是解决这个问题的一种方式。Cookies 是一种在客户的机器上存储信息的技术，这些信息可以是任何内容，例如数据库查询键码、用户喜欢的值或站点可以用于跟踪其用户的其他东西。当用户浏览一个站点时，Web 服务器就会给客户的机器发送 Cookie，并把它保存起来，当用户再次访问这个站点时，浏览器就会给服务器发送以前保存的 Cookie，让站点对内容进行个性化设置，或者获取正确的购物篮，或者完成其他需要完成的操作。Web 站点现在就可以在用户的后续访问中维护该用户的状态了。另一种跟踪站点用户的方式是在一个数据库中记录它们。如果要执行服务器端的有效性验证，看看用户的 ID 和密码是否匹配于存储在用户数据库中的值，这种方式是非常有效的。

23.4　Web 的演化过程

在第一次引入 ASPWeb 页时，它们是非常好的，它们解决了 World Wide Web 如何发挥作用的基础性问题。根据从数据库或其他后端系统中获取的内容创建活动 Web 页的能力是具有革命意义的。许多 Web 站点目前仍在使用这项技术。

但许多人发现，服务器端脚本有一个基本限制。因为这些页面没有编译，在客户机发出请求时执行，响应时间可能比较长。每个页面都要用相当长的时间创建，因此如果一秒内有上千个请求，Web 服务器就跟不上加载页面的速度。

所以开发了另一个体系结构，它把业务逻辑和数据访问分布到许多组件，这些组件以编译好的 COM 组件方式来运行，这称为 3 层设计，因为应用程序现在被分解为 3 个部分：前端/显示层、中间层 COM 组件和后端的数据库。这很快演变为 n 层模型，因为中间层演变为不同级别的 COM 组件和应用程序组合。

在解开这些层的组合时，每个层都可以单独开发，只要层之间的接口保持不变即可。这就把主要的注意力集中到组件的开发上，设计出最佳的组件，对 Active Server Pages 脚本部分的依赖性也达到最小。ASP 仍需要调用这些组件，格式化返回的数据，因为它们是 Web 应用程序的用户界面层。

这个模型在建立可伸缩的、高可用性的企业 Web 应用程序中被证明是非常有效的，如图 23-5 所示。

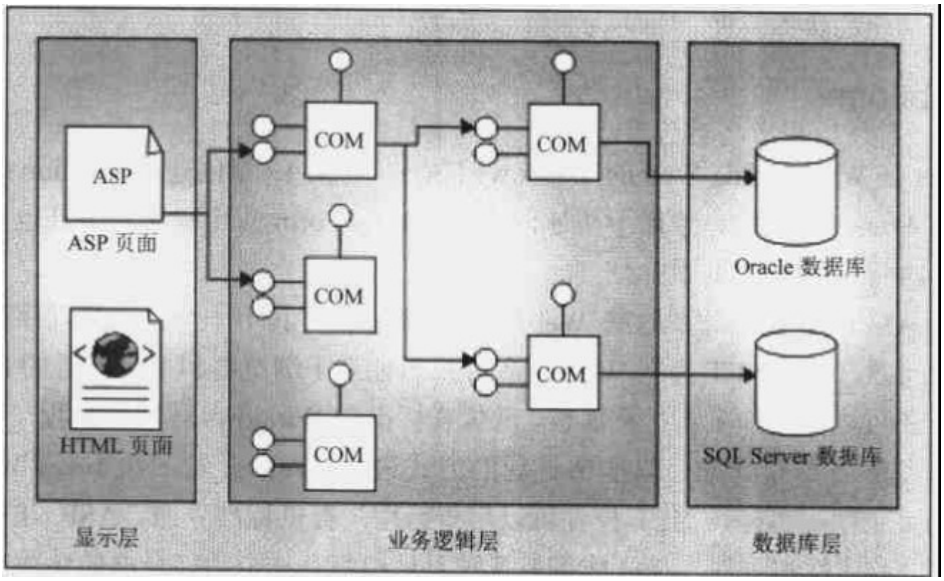


图　23-5

23.5　.NET Web 编程

.NET 和 ASP.NET 最适合在什么地方采用这个模型？.NET Framework 代表着企业 Web 应用程序开发的下一个演化步骤。它主要强调组件和互操作性，是下一代的编程环境。ASP.NET 是.NET Framework 在活动 Web 页上的应用。使用 ASP.NET 可以创建编译好的、数据驱动的

Web 页，通过这些 Web 页可以访问.NET Framework 的各种特性和功能。

对于 Web 开发人员来说，这意味着 ASP 在 n 层应用开发链中不再是比较弱的一环。现在可以开发真正强大、编译好的 Web 应用程序，该程序直接给客户机返回 HTML，且仍可在功能丰富的开发环境中开发。

23.5.1 ASP.NET

与 ASP 不同的是，ASP.NET 没有使用自己的编程语言，ASP.NET 应用程序的建立类似于建立本书的其他程序，使用 C#, VB.NET 或其他.NET Framework 兼容语言。这样，Web 程序员就把内容和布局真正分离开来，因为 ASP.NET 应用程序可以使用 WebForms 显示用户界面，使用本书介绍的所有.NET Framework 工具建立编程后端。

组成 ASP.NET 应用程序的代码是编译好的.NET 程序，所以它的运行比相同的 ASP 应用程序快得多。仅此一项就可以大大提高性能。另外，模型的自动高速缓存和易于使用的状态管理系统，也使该开发环境变得更为强大。

下面是 ASP.NET 的一个简短的优点列表：

- 编译好的代码，可以进行早期绑定、强类型化和即时(JIT)编译为本机代码
- 基于 XML 的配置文件
- 与.NET Framework 的全面集成
- 易于使用的高速缓存模型
- 完全自动化的状态管理

23.5.2 WebForms

WebForms 是 World Wide Web 的 Windows Form。与第 13 章创建的 Windows Form 应用程序一样，.NET Framework 的许多用于快速创建 Windows Form 应用程序的工具也可以用于建立交互操作的、功能丰富的 Web 页。

与标准的 ASPWeb 页相比，这些 Web 页有许多优点。其中一个优点是程序员不必担心浏览器的依赖性问题。ASP.NET 会把 WebForm 转换为独立于浏览器和平台核心的 HTML 版本。Web 开发人员还可以访问许多事件和属性，就像在标准的 Windows 环境中编程一样。这些事件可以在客户机或服务器端处理，以非常直观的方式进行 Web 开发，而不必担心处理和解释 HTML 窗体的复杂性。另外，.NET 控件可以用于给用户提供标准界面，ASP.NET 允许开发人员设置这些控件的属性和事件。最终结果是开发人员最终有一个功能全面的环境来建立高效、可伸缩的 Web 应用程序。下面都是 WebForm 的优点：

- 可以运行在任何浏览器上，并自动显示适合该浏览器的 HTML 版本
- 可以用任何与.NET 兼容的语言编写
- 全面支持 GUI 开发工具
- 丰富的服务器和客户机控件
- 能轻松地把显示代码和业务逻辑代码分离开来

23.6　小结

World Wide Web 已经存在了很长时间，它逐渐演变成一种功能非常健壮、非常丰富的环境。ASP.NET 是利用了.NET Framework 的功能进行 Web 站点开发的一种工具。

本章介绍了 Internet 和 World Wide Web 的简史，讨论了在这个过程中开发人员是如何创建 Web 站点的。我们学习了 HTTP 和 HTML 的基础知识，了解了需要使用客户端和服务器端脚本的几个原因。本章还讨论了 JavaScript 如何用于创建客户端的有效性验证，学习了 Active Server Pages 的作用。

我们还学习了为什么 ASP.NET 是 Web 开发的下一步，使用 ASP.NET 和 WebForm 的优点。第 24 章将深入探讨 ASP.NET，并建立一个工作范例。

第24章 ASP.NET应用程序

如第23章所述，ASP.NET是建立动态Web站点的.NET方式。与ADO.NET一样，ASP.NET的名称来源于上一代技术Active Server Pages(ASP)，但其类似之处并不像它们的名称那样多。ASP页面是用功能有限的脚本语言(例如VBScript和JScript)解释和编写的，而ASP.NET应用程序被编译为MSIL，就像其他.NET应用程序那样。这表示我们可以利用能使用其他.NET对象的强大语言来编写它们，当然本书关心的是(因为这是一本关于C#的书)如何用C#编写ASP.NET应用程序。

ASP.NET的第二个优点是引入了服务器端控件。在ASP中，可以给页面添加HTML控件(例如按钮和其他窗体元素)，但只能手动添加，且必须在客户端的JScript中响应用户对这些控件的输入(例如用户单击一个按钮)，或者重新把页面提交给服务器。这样产生的第一个问题是不同的浏览器以不同的方式执行JScript，几乎不可能编写在Netscape和IE上都能运行的复杂例程，而且一些旧浏览器根本就不支持JScript。如果重新把页面提交给服务器，那么Internet的无状态性就会丢失页面上存储在变量中的信息，除非我们编写了复杂的代码，把它们存储在HTML元素或URL查询字符串中。

ASP.NET通过服务器端控件解决了这些问题。这些控件生成发送给浏览器的HTML代码，显示控件，它们还可以生成JScript函数和隐藏的HTML元素，来存储它们的当前状态。在提交页面时，这些信息就会传送回服务器，控件将自动处理这些信息，改变HTML，以显示控件。简言之，我们可以让文本框记忆用户输入的值，但这项技术更进了一步，允许ASP.NET拥有比ASP更复杂的控件(例如日历控件)。

最后，也许是最重要的，ASP.NET提供了一种创建Web服务的简单方式。现在可以把它们看作是Web的组件，但Web服务是一个复杂的主题，第25章将详细讨论它们。

要理解服务器端的代码，可访问Amazon.com，查找自己喜欢的小说。我们浏览的小说页面并不存在于服务器上，而存在于一个描述页面一般外观的模板上，当用户请求这个页面时，就执行服务器上的代码，从数据库中提取该小说的信息，把它插入到模板上，传送给用户的是HTML，详见后面的内容。

重要的是本章将主要讨论运行在服务器端的代码。Web浏览器不必执行什么操作就可以显示页面。

提示：

从现在开始，只要提到“活动的页面”，就是指可以用典型的ASP和ASP.NET建立的页面。这两个技术都使用相同的方式建立页面，只是ASP.NET功能更强大，用ASP.NET把动态的Web站点组织在一起要容易得多。

下面是一个范例，它看起来是一个普通的Web页，如图24-1所示。



图　24-1

但注意页面底部的版权信息。假定这个信息显示在 Web 站点的每个 Web 页上。现在它设置为 2001，而当进入 2002 年时，就必须浏览站点，把每个页面从 2001 更新为 2001-2002。

有了“活动的页面”，就可以配置页面，让它自己确定年份，自动更新。这就是说，这个简单而烦琐的管理任务将不再有了。

OK，这是一个相当陈腐的范例。在把 Web 站点与某些数据库或其他形式的功能(例如电子商务购物车)连接在一起时，活动页面技术实际上就已经启动了。本章将解释如何建立一个“weblog”。Weblog 等价于一个 Web 日志，它不是写在纸上，锁在抽屉里，而是每个 Web 上的人都可以查看。规则是一样的，有一个日期，在该日期下有当天的想法和事件列表，它们有时称为“blog”，在 www.bloghop.com 上可以查找到它们的一个列表。

24.1　建立 Weblog

为了说明 ASP.NET 的强大功能，下面按照第一个规则开发一个 weblog 应用程序：首先从一个空白的 Web 页开始，一点一点地添加功能，并在此过程中说明各个功能。首先说明如何显示 weblog 项目，一次显示一个项目还是多个项目。然后说明如何给项目排序、添加和编辑项目，再简要说明安全性问题，例如使用注册页面限制对站点的访问和给用户授权。在完成这个工作范例的过程中，读者将学习到下述内容：

- 使用 Web 窗体控件给 Web 页添加动态(活动)的内容
- 使用 C#代码执行每个控件的事件处理程序
- 创建一个视觉效果比较好的用户界面
- 扩展用户界面，以支持其他功能

24.1.1　应用程序的基础

可以使用 ASP.NET Web Application 在 Visual Studio.NET 中建立应用程序，这将创建一个新工程，并自动配置 IIS，以便在建立工程的同时调试它。

首先是在 Web 站点上建立一个简单的静态页面(换言之，没有动态元素的页面)，以便确保所有元素按照希望的那样工作。我们把这个页面创建为 ASP.NET 页面(其扩展名是.aspx)，因为尽管最初这个页面没有动态内容，但它以后会有一些动态的东西。

注意：

在 ASP 页面中，扩展名通常是.asp，表示 Active Server Pages。在 ASP.NET 中，页面的扩

展名是.aspx，最初它是 ASP+的一个变体，ASP+是 ASP.NET 最初的名称。

试试看：创建 Weblog 页面

(1) 打开 Visual Studio.NET，从菜单中选择 File | New | Project。在 Project Types 列表中，选择 Visual C# Projects，并从 Templates 列表中选择 ASP.NET Web Application。在名称框中，输入 Weblog，如图 24-2 所示。

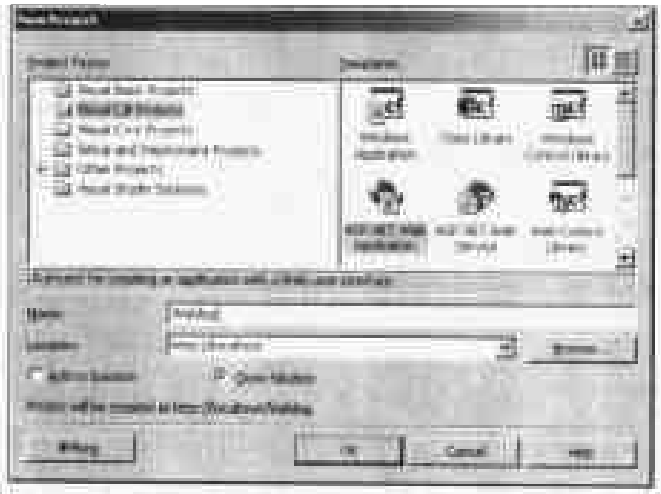


图 24-2

(2) 在单击 OK 按钮以前，注意在 Location 框中，URL 是http://localhost，这表示要创建一个新的 Web 站点，在讨论 Internet 时，localhost 总是表示自己的计算机。

(3) 单击 OK 按钮，创建新工程。

(4) 使用 Solution Explorer，右击 Weblog 工程，选择 Add | Add Web Form，输入名称 Default.aspx，并单击 Open 按钮。

(5) 新页面在设计模式下打开，但如果读者以前使用过 ASP，可以不在该模式下打开页面。如果页面是在设计模式下打开，其结果应如图 24-3 所示。

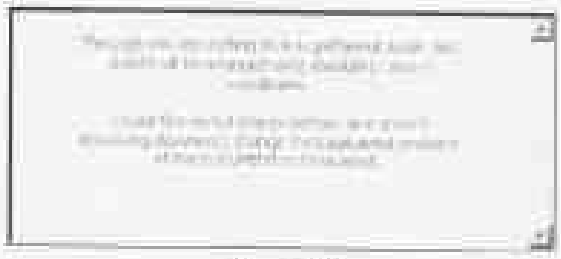


图 24-3

如果显示了非常多的 HTML，可从菜单中选择 View | Design，页面中应有上图中的网格布局。

(6) 在设计器中，可以用两种方式建立页面：GridLayout 模式和 FlowLayout 模式。在我看来，最好使用 FlowLayout 模式。在 Properties 窗口中，把 pageLayout 属性改为 FlowLayout，设计器中的点就会消失。

设计器还提供了一组便于建立 Web 站点的工具。实际上，这表示建立 Web 页就像使用 Microsoft Word 写信一样。但是，设计器提供的工具有许多限制，因为它们主要是针对那些只

对设计 Web 界面感兴趣，而对编程不感兴趣的人的。可以说，读者在阅读到本书的这个地方时，一般喜欢编写计算机软件，所以下面回到原来的题目，建立 HTML 页面。如果不知道如何编写 HTML，请参阅第 23 章，复习基本的 HTML 知识。

(7) 从菜单中选择 View | HTML Source，显示 HTML 编辑器(在代码编辑器的底部，有一些标有 Design 和 HTML 的小按钮，使用这两个按钮可以在视图之间切换)。

(8) 在 HTML 代码框架中，添加下面突出显示的代码：

```
<%@ Page language="c#" Codebehind="Default.aspx.cs" AutoEventWireup="false"
Inherits="WebLog.CDefault" %>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN" >
<html>
    <head>
        <meta name="GENERATOR" Content="Microsoft Visual Studio 7.0">
        <meta name="CODE_LANGUAGE" Content="C#">
        <meta name="vs_defaultClientScript" content="JavaScript (ECMAScript)">
        <meta name="vs_targetSchema"
              content="http://schemas.microsoft.com/intellisense/ie5">
    </head>
    <body>
        <form id="Default" method="post" runat="server">
            <div>
                Disraeli's Weblog
            </div>
            <br>
            <div>
                Copyright &copy; Wrox Press 2001
            </div>
        </form>
    </body>
</html>
```

(9) 在运行工程前，必须告诉 VS，新的 Default.aspx 页面是我们要调试的页面。使用 Solution Explorer，右击 Default.aspx，并选择 Set as Start Page。

像往常一样编译并运行工程，得到如图 24-4 所示的结果。

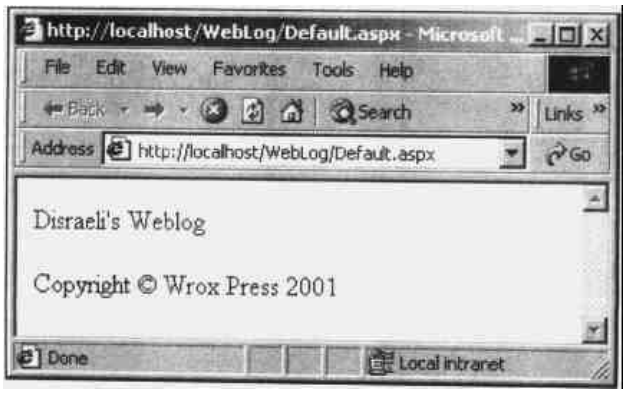


图　24-4

代码的说明

前面创建了一个新工程，并给它添加了一个页面。VS 在后台处理大部分工作。

但是，这个页面上没有任何活动的内容，它仅有我们键入的静态文本。下面看看如何给页面添加活动元素。

24.1.2　添加活动元素

如果读者以前建立过 ASP 页面，或者使用新技术建立过动态 Web 页，此时应抛弃以前了解的建立活动页面的所有东西。ASP.NET 使用的是一种完全不同的方式。

在 Microsoft 看来，Visual Basic 是非常成功的，读者可以用与建立 Windows Form 应用程序相同的方式建立 Web 应用程序。所有的内容都是以控件为中心，用事件驱动。

例如，假定有一个用 VB 或 C#编写的 Windows 应用程序。如果要给窗体添加一个按钮，使之完成某些任务，可以使用工具箱在窗体上绘制该按钮，再双击它，创建一个事件处理程序，添加一些代码。这就是以控件为中心，用事件驱动的编程方式。

以前在 Web 上，没有控件的概念。所有的东西都是 HTML。如果要绘制一个按钮，服务器端的脚本就必须发送 HTML，让浏览器显示该按钮。当按钮被单击时，浏览器会再次请求该页面，但给页面添加一些额外的信息，表示该按钮已单击。服务器端脚本再响应这些额外的信息，给浏览器发送新的或不同的信息。

在.NET 中，这些都是在后台发生的。我在设计器中绘制一个按钮，ASP.NET 把这个控件转换为 HTML，把它发送给浏览器，浏览器再绘制出该按钮。如果按钮被单击，就给服务器发送回一个请求，但我们觉得按钮的单击是可以响应的一个事件，本节就介绍这个过程。

试试看：添加一个版权元素

(1) 如果工程在运行，通过关闭 Internet Explorer 窗口，关闭这个工程。

(2) 从菜单中选择 View | Design。只要想给页面添加活动元素，就必须进入设计视图。

(3) 为了帮助用户布置控件，从菜单中选择 View | Details，得到如图 24-5 所示结果。

(4) 选择并删除版权信息。

(5) 使用工具箱，把一个 Label 控件拖放到两个 div 标记之间，这两个 div 标记描述了版权信息，结果如图 24-6 所示。



图 24-5



图 24-6

小方框 <?> 表示创建的标记不是一个 HTML 标记，它是一个只有 ASP.NET 才能理解的标

记，所以把它显示为一个<?>(可以想像，Microsoft 要在这里绘制一个小图标，表示它的含义，也许它们将在以后的版本中出现)。

(6) 这里要注意，从概念上讲，WebForms 标签控件的使用与 Windows Forms 标签控件的使用没有任何区别。实际上，如果选择这个控件，查看 Properties 窗口，则可以改变它的属性，这就是我们下面要做的工作：

- 把 ID 属性设置为 labelCopyright
- 把 Text 属性设置为(copyright)

(7) 双击页面的背景，这在概念上类似于双击 Windows Form 的背景，创建一个事件处理程序，该程序在加载页面时调用。

(8) 把下面的代码输入到 Page_Load()方法中：

```
private void Page_Load(object sender, System.EventArgs e)
{
    // what year is this?
    int year = DateTime.Now.Year;
    if(year == 2001)
        labelCopyright.Text = "Copyright &copy; Disraeli " + year;
    else
        labelCopyright.Text = "Copyright &copy; Disraeli 2001-" + year;
}
```

编译并运行工程。(不能右击代码编辑器，选择 View in Browser，必须像以前那样运行代码)。得到与以前相同的结果，如图 24-7 所示。

注意：

如果运行 Outlook 或其他支持进度安排的程序，一定要关闭它。我们把系统时钟设置得提前一些，除非要立即显示 12 个月的进度安排，否则就应关闭它。而对于那些对日期变化非常敏感的应用程序来说，也要关闭它。如果不能确定改变日期对应用程序会产生什么影响，就不要改变日期。这并不是什么好玩的事。

现在，把系统时钟设置为未来的某一年，刷新页面，得到如图 24-8 所示的结果。

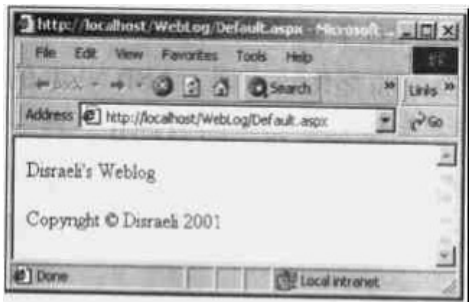


图　24-7

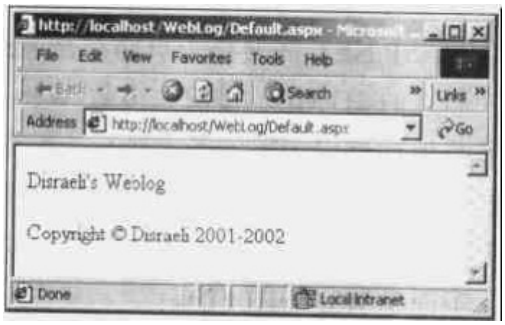


图　24-8

代码的说明

前面的工作证明了标签信息是动态的。在建立这个页面时，使用 DateTime.Now()获取当前

日期，改变版本信息。

但活动的元素如何进入页面？为了弄明白这一点，打开页面的 HTML 编辑器，在这里可以找到这个标记：

```
<%@ Page language="c#" Codebehind="Default.aspx.cs" AutoEventWireup="false"
Inherits="WebLog.CDefault" %>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN" >
<html>
    <head>
        <meta name="GENERATOR" Content="Microsoft Visual Studio 7.0">
        <meta name="CODE_LANGUAGE" Content="C#">
        <meta name="vs_defaultClientScript" content="JavaScript (ECMAScript)">
        <meta name="vs_targetSchema"
              content="http://schemas.microsoft.com/intellisense/ie5">
    </head>
    <body>
        <form id="Default" method="post" runat="server">
            <div>
                Disraeli's Weblog
            </div>
            <br>
            <div>
                <asp:Label id="labelCopyright"
                           runat="server">(copyright)</asp:Label>
            </div>
        </form>
    </body>
</html>
```

在接收到.aspx 页面的一个请求后，ASP.NET 会在处理页面前检查整个页面，以确定页面的哪些部分是静态的，哪些部分是动态的。在上面的代码范例中，突出显示了<asp:Label>标记。ASP.NET 知道这个标记是页面的活动部分。在排除页面的活动部分后，页面的其他部分就都是静态的。

最后，ASP.NET 需要整个页面都是静态的，所以 ASP.NET 的作用主要是把页面的动态部分转换为静态的 HTML。目前大多数 Web 浏览器在功能方面都比较有限。ASP.NET 的方式可以使大多数处理过程在服务器上进行，最终发送回可以在浏览器上显示的 HTML。在一些情况下，ASP.NET 可以把浏览器应执行的代码发送给浏览器，但在以后的几年中，浏览器将能完成越来越多的任务，现在我们仍在服务器上进行所有的处理。这些处理工作也的确是在服务器上进行的。下面是 ASP.NET 代码：

```
<div>
    <asp:Label id="labelCopyright" runat="server">(copyright)</asp:Label>
</div>
```

下面是 IE 实际接收到的代码：

```
<div>
    <span id="labelCopyright">Copyright &copy; Disraeli 2001-2002</span>
</div>
```

使用 Visual Studio.NET 编写 ASP.NET 页面时，页面默认使用的模型把编写的代码分为两部分。第一部分是.aspx 文件，包含页面的模板，定义了活动元素应在什么地方显示。第二部分是.cs 或.vb 文件，包含事件处理代码。在双击页面的背景，访问 Load 事件处理程序时，实际上是打开了一个新文件，如图 24-9 所示。



图　24-9

这个 C#源文件称为后台编码(code-behind)文件，其文件名与.aspx 文件相同，在内部链接到.aspx 文件上。如果查看这个文件，就会注意到它包含一个类定义，这个类派生于 System.Web.UI.Page。

```
/// <summary>
/// Summary description for CDefault.
/// </summary>
public class CDefault : System.Web.UI.Page
```

提示：

ASP.NET 自动给类名加上前缀“C”(表示“类”)。这将减少命名冲突。

这个类提供了一个 Load 事件，在加载页面时，会引发该事件。这是在 ASP.NET 加载.aspx 文件，并检查它，确定页面上存在动态元素后发生的。

在页面上定义的每个动态元素也是类的受保护成员。把标签控件的 ID 属性改为 labelCopyright，该成员就是在这里定义的：

```
/// <summary>
/// Summary description for CDefault.
/// </summary>
public class CDefault : System.Web.UI.Page
{
    protected System.Web.UI.WebControls.Label labelCopyright;
```

这是另一个典型的范例，说明了 WebForms 和 Windows Forms 有多么类似。在 Windows Form 上绘制一个控件时，也添加了同样类型的成员。有了这个成员后，就可以通过代码来访问它：

```
private void Page_Load(object sender, System.EventArgs e)
{
    // what year is this?
```

```
        int year = DateTime.Now.Year;
        if(year == 2001)
            labelCopyright.Text = "Copyright &copy; Disraeli " + year;
        else
            labelCopyright.Text = "Copyright &copy; Disraeli 2001-" + year;
    }
```

下面看看发生了什么情况。此时，ASP.NET 加载了.apsx 文件，并查找动态元素。实际上，ASP.NET 在内存中存储了页面的一个 HTML 代码图像，但在 HTML 代码中为页面上的活动元素插入了占位符。ASP.NET 现在给出页面和页面上的控件，同时把它们的 HTML 代码插入到这些占位符上。设置标签控件的 Text 属性，就是告诉控件，ASP.NET 最终将在什么时候要求 HTML 代码把该控件放在 HTML 代码图像中的 ASP.NET 为它保留的占位符上，则标签控件应提供如下所示的 HTML 代码：

```
<span id="labelCopyright">Copyright &copy; Disraeli 2001-2002</span>
```

换言之，它提取了存储在 text 中的值，并用 和 标记把它括起来。

最后，ASP.NET 决定所有的控件都提供，组成页面的各个部分的 HTML，再把整个 HTML 图像发送给显示它们的浏览器。

24.2 Weblog 项目

现在读者已经理解了使用控件/事件模式建立 ASP.NET 页面的规则，下面看看如何使用这个规则建立一些有用的页面。我们现在要做的是建立一个单独的页面，该页面列出整个 Weblog 项目，这是某一天的事件和想法列表。

建立活动 Web 站点的一个优点是每次要增加或修改内容时，都不必重新编写 HTML 代码。这样不了解 HTML 的人就可以给站点添加内容，而有经验的 HTML 程序员也可以快速而轻松地添加内容。

给站点添加内容的传统方式是显示一个窗体，让用户输入他们的内容。单击 Save Changes 按钮后，新内容就会显示在站点上。这些内容通常存储在某种类型的数据库中。

在这个范例中，我们要把内容存储在 Web 服务器的一个单独的 XML 文件中，而不是存储在数据库中。每个文件都包含一个 weblog 项目，或者某一刻发生的事件。下面介绍如何创建这些项目，再说明如何把它们存储到页面上。

24.2.1 存储 Weblog 项目

如上所述，我们将在服务器上存储包含 weblog 项目的 XML 文件。但是，在此之前，要先介绍几个限制。

读者可能注意到尽管前面已经创建了新工程，但不知道文件要存储在什么地方。这是因为 VS 是通过 FrontPage Extensions 与 Web 服务器通信来存储文件的。

注意：

FrontPage Extensions 有一个简短的历史。其规则是它们在更新 Internet 上的一个远程 Web 站点时，仅把文件从计算机的一个文件夹复制到另一个文件夹上。实际上，系统管理员很难配置 FrontPage Extensions，会出现 Web 站点安全问题，而且有点古怪。我们只能希望.NET 套装软件包含的版本解决这些问题。

这就产生了第一个问题，因为我们需要知道这些文件存储在什么地方，这样才能支持它们。幸运的是，ASP.NET 可以告诉我们文件在哪里。

试试看：确定存储 Web 站点的位置

(1) 打开 Default.aspx 的 HTML 编辑器，添加突出显示的代码(为了简单起见，这里省略了一些代码)：

```
<form id="Default" method="post" runat="server">
    <div>
        Disraeli's Weblog
    </div>
    <br>
    <div>
        <asp:Label id="labelCopyright"
                    runat="server">(copyright)</asp:Label>
    </div>
    <br>
    <div>
    </div>
</form>
```

(2) 从菜单中选择 View|Design，显示设计器。

(3) 使用工具箱，把一个新 Label 控件拖放到两个新的 div 标记之间。把该控件的 ID 属性设置为 labelServerPath，把 Text 属性设置为 (serverpath)，如图 24-10 所示。

(4) 双击页面背景，再次打开 Load 处理程序。添加如下代码：

```
private void Page_Load(object sender, System.EventArgs e)
{
    // what year is this?
    int year = DateTime.Now.Year;
    if(year == 2001)
        labelCopyright.Text = "Copyright &copy; Disraeli " + year;
    else
        labelCopyright.Text = "Copyright &copy; Disraeli 2001-" + year;

    // set the server path...
    labelServerPath.Text = Server.MapPath("");
}
```

运行工程，得到如图 24-11 所示的结果。

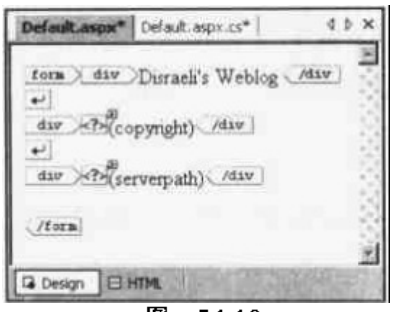


图　24-10



图　24-11

代码的说明

在第一个范例中，给窗体添加了一个新控件。我们要用这个控件报告存储 Web 站点的路径。

Cdefault 类派生于 System.Web.UI.Page，而 System.Web.UI.Page 有许多属性，这些属性可以帮助我们理解页面的内容，访问环境的一些信息。Server 属性包含一个方法 MapPath，它可以把 Web 站点上的一个虚拟路径转换为 Web 服务器上的一个物理路径。在这个范例中，要把它转换为一个空白字符串，告诉它我们要在文件夹中包含 Web 站点的根路径。

```
private void Page_Load(object sender, System.EventArgs e)
{
    // what year is this?
    int year = DateTime.Now.Year;
    if(year == 2001)
        labelCopyright.Text = "Copyright &copy; Disraeli " + year;
    else
        labelCopyright.Text = "Copyright &copy; Disraeli 2001-" + year;

    // set the server path...
    labelServerPath.Text = Server.MapPath("");
}
```

如果使用 Explorer，并导航到 Web 页上给出的路径上，就可以看到这些文件。这里突出显示了.aspx 文件和.cs "code behind" 文件，如图 24-12 所示。



图　24-12

下面创建一个 XML 文件，用于存储 weblog 中的项目。

试试看：创建一个项目 XML 文件

(1) 在包含 Web 站点的文件夹中，创建一个新文件夹 Entries。例如，在我的计算机中，新文件夹是：

```
c:\Inetpub\wwwroot\WebLog\Entries
```

本章的其他部分都把这个文件夹称为“项目文件夹”。

(2) 打开 Notepad，创建这个文件：

```
<?xml version="1.0" ?>
<Entry xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <Title>Hello!</Title>
  <Details>These are the details of the Weblog entry</Details>
</Entry>
```

XML 是区分大小写的，所以应确保输入的文件如上所示，例如，上面的代码是<Title>，就必须输入<Title>，而不能输入<title>或<TItLe>。

把这个文件保存为 Entry.xml，并把它放在刚才创建的项目文件夹中。

24.2.2　显示 Weblog 项目

现在我们有一个包含一个项目的 XML 文件，可以为 Weblog 中每个要显示的项目创建更多的文件，还可以建立一个加载这些文件的类，显示包含在其中的数据，就像属性一样。

下面就说明如何加载文件。

试试看：加载 Weblog 文件

(1) 要加载 XML 文件，需要给由 Visual Studio.NET 创建的类 Global 添加一个静态方法，这个类放置共享的方法和属性，这些方法和属性可以用于 Web 应用程序中所有的页面，它们都在 Global.asax 页面中。

(2) 要查找 Global，可使用 Solution Explorer，右击 Global.asax 文件，并选择 View Code，如图 24-13 所示。

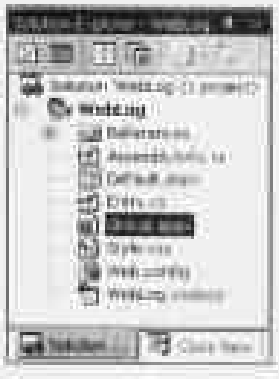


图 24-13

(3) 注意 Global 类派生于 System.Web.HttpApplication。这个类提供了一些事件，当某个应用程序作用域内的事件发生时，就可以响应它们(在 ASP.NET 中，应用程序的作用域可以看作是 Web 站点的作用域)。

(4) 把这个成员添加到 Global 中：

```
/// <summary>
/// Summary description for Global.
```

```
/// </summary>
public class Global : System.Web.HttpApplication
{
    // members...
        public static String EntryFilePath;
```

注意我们使用类名 String(s 大写)，而不是 C#关键字 string(s 小写)。这是因为按照 Common Language Runtime，这个类应可见，但如果使用 C#关键字 string，它就会在内部转换为 System.String ——这样就不必进行额外的处理。

(5) 接着把下面的代码添加到 Application_Start()方法中：

```
protected void Application_Start(Object sender, EventArgs e)
{
    // set the shared entry path member...
    EntryFilePath = Server.MapPath("Entries");
}
```

(6) 使用 Solution Explorer，创建一个新类 Entry。首先在新文件的顶部添加如下 using 语句：

```
using System;
using System.IO;
using System.Xml.Serialization;
```

(7) 然后添加下面的成员和相应的属性：

```
/// <summary>
/// Summary description for Entry.
/// </summary>
public class Entry
{
    // members...

private DateTime _timestamp;
private String _title;
private String _details;

[XmlIgnore()] public DateTime Timestamp
{
    get
    {
        return _timestamp;
    }
    set
    {
        _timestamp = value;
    }
```

```
    }

    public String Title
    {
        get
        {
            return _title;
        }
        set
        {
            _title = value;
        }
    }

    public String Details
    {
        get
        {
            return _details;
        }
        set
        {
            _details = value;
        }
    }
```

注意这些属性的名称都匹配于刚才创建的 XML 文件中的项目的名称，这是经过深思熟虑的！

(8) 现在返回 Global.asax.cs 的代码编辑器，在文件的顶部添加相同的名称空间引用：

```
using System;
using System.Collections;
using System.ComponentModel;
using System.Web;
using System.Web.SessionState;
using System.IO;
using System.Xml.Serialization;
```

(9) 接着把这个静态方法添加到 Global 类中：

```
    // LoadEntry - load an entry from disk...
    public static Entry LoadEntry(String filename)
    {
        // we have the name, but we need the path...
        String filepath = EntryFilePath + "\\" + filename;
```

```
        // open the file...
        FileStream file = new FileStream(filepath, FileMode.Open);

        // create a serializer...
        XmlSerializer serializer = new XmlSerializer(typeof(Entry));
        Entry newEntry = (Entry)serializer.Deserialize(file);

        // close the file...
        file.Close();

        // return the entry...
        return newEntry;
    }
```

(10) 从技术上看，现在可以创建一些新 Entry 对象，用存储在 XML 文件中的数据填充 Title 和 Details 属性。现在需要在 Default.aspx 页面上创建一些控件，以便查看结果。

(11) 打开 Default.aspx 的 HTML 编辑器，添加如下代码：

```
<form id="Default" method="post" runat="server">
    <div>
        Disraeli's Weblog
    </div>
    <br>
    <div>
    </div>
    <div>
    </div>
    <br>
    <div>
        <asp:label id="labelCopyright" runat="server">(copyright)</asp:label>
    </div>
...
</form>
```

(12) 回到 Default.aspx 的设计器中。在两个新 div 标记中，分别拖放一个新标签控件，如图 24-14 所示。

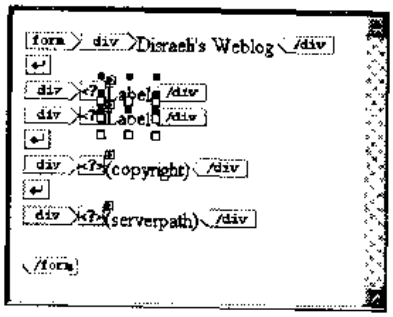


图　24-14

• 对于第一个新控件，把 ID 属性设置为 labelEntryTitle，把 Text 属性设置为(entryTitle)。

• 对于第二个新控件，把 ID 属性设置为 labelEntryDetails，把 Text 属性设置为(entryDetails)

(13) 双击页面的背景，打开 Load 事件处理程序，首先修改最后一行，以使用 Global.EntryFilePath 共享属性：

```
private void Page_Load(object sender, System.EventArgs e)
{
    // what year is this?
    int year = DateTime.Now.Year;
    if(year == 2001)
        labelCopyright.Text = "Copyright &copy; Disraeli " + year;
    else
        labelCopyright.Text = "Copyright &copy; Disraeli 2001-" + year;

    // set the server path...
    labelServerPath.Text = Global.EntryFilePath;
  }
```

(14) 接着，添加如下代码，以在 Global 上使用共享方法，根据存储在 Entry.xml 中的数据创建一个新的 Entry 对象，再填充刚才添加的两个新控件 labelEntryTitle 和 labelEntryDetails：

```
private void Page_Load(object sender, System.EventArgs e)
{
...
    // set the server path...
    labelServerPath.Text = Global.EntryFilePath;

    // load the entry from disk...
    Entry entry = Global.LoadEntry("Entry.xml");
    labelEntryTitle.Text = entry.Title;
    labelEntryDetails.Text = entry.Details;
}
```

运行工程，得到如图 24-15 所示的结果：

(15) 要确信从磁盘上加载了改变的内容，可使用 Notepad 再次编辑 Entry.xml。改变数据，如下所示：

```
<?xml version="1.0"?>
<Entry xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <Title>Hello, again!</Title>
  <Details>I have changed the details in the XML file...</Details>
</Entry>
```

(16) 保存文件，刷新 Internet Explorer 页面，此时就应看到如图 24-16 所示的变化。



图　24-15



图　24-16

代码的说明

在 Global.asax 中，首先再次使用 Server.MapPath，把虚拟的 Entries 目录解析为可以在代码中使用的物理路径。在接收到.aspx 文件的第一个请求时，调用 Application_Start()。这样就有机会设置应用程序(或 Web 站点)的全局数据。在本例中，我们把物理路径存储在共享的 EntryFilePath 成员中：

```
protected void Application_Start(Object sender, EventArgs e)
{
    // set the shared entry path member...
    EntryFilePath = Server.MapPath("Entries");
}
```

现在该处理静态的 LoadEntry()方法了，该方法从项目文件夹中加载 XML 数据文件，并把内容放在我们定义的 Entry 类中。为此，使用 System.Xml.Serialization 名称空间中的一个类 XmlSerializer，调用它的方法 Deserialize()，从文件中提取 XML 字符串，并把它的数据转换(并行化)为一组对象属性，它们存储在 Entry 类的实例中。

```
// LoadEntry - load an entry from disk...
public static Entry LoadEntry(String filename)
{
    // we have the name, but we need the path...
    String filepath = EntryFilePath + "\\" + filename;

    // open the file...
    FileStream file = new FileStream(filepath, FileMode.Open);

    // create a serializer and use it to populate the properties of the
    // a newly created Entry object...
    XmlSerializer serializer = new XmlSerializer(typeof(Entry));
    Entry newEntry = (Entry)serializer.Deserialize(file);

    // close the file...
```

```
    file.Close();

    // return the entry...
    return newEntry;
}
```

在 Default.aspx.cs 中，再次定制 Load 事件处理程序。这次假定 XML 文件叫作 Entry.xml，设置每个标签控件的 Text 属性，如下所示：

```
private void Page_Load(object sender, System.EventArgs e)
{
    // what year is this?
    int year = DateTime.Now.Year;
    if(year == 2001)
        labelCopyright.Text = "Copyright &copy; Disraeli " + year;
    else
        labelCopyright.Text = "Copyright &copy; Disraeli 2001-" + year;

    // set the server path...
    labelServerPath.Text = Global.EntryFilePath;

    // load the entry from disk...
    Entry entry = Global.LoadEntry("Entry.xml");
    labelEntryTitle.Text = entry.Title;
    labelEntryDetails.Text = entry.Details;
}
```

24.2.3　显示时间

前面没有在项目的旁边显示时间。要显示时间，应提取上一次保存文件的时间，并把日期保存在项目的 Timestamp 属性中。接着修改代码，以便在显示细节前显示时间和日期。

试试看：显示时间

(1) 要获取时间，需要使用上一次修改项目的日期。在大多数情况下，这是项目上一次保存到磁盘上的时间。

(2) 使用 Solution Explorer，右击 Global.asax，并选择 View Code，找到 LoadEntry 方法，添加如下代码：

```
// LoadEntry - load an entry from disk...
public static Entry LoadEntry(String filename)
{
...

    // close the file...
    file.Close();
```

```
        // update timestamp...
        newEntry.Timestamp = new FileInfo(filepath).LastWriteTime;

        // return the entry...
        return newEntry;
    }
```

(3) 现在，打开 Default.aspx.cs 的代码编辑器，进行如下修改：

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

        // set the server path...
        labelServerPath.Text = Global.EntryFilePath;

        // load the entry from disk...
        Entry entry = Global.LoadEntry("Entry.xml");
        labelEntryTitle.Text = entry.Title;
        labelEntryDetails.Text = entry.Timestamp.ToString("dddd") + ", " +
            entry.Timestamp.ToLongDateString() + " - " + entry.Details;
        }
```

运行工程，这次就会显示时间了，如图 24-17 所示。



图　24-17

代码的说明

Entry 对象已经包含一个存储时间信息的成员。使用 XmlIgnore 属性可以告诉 XmlSerializer 不必担心它，这就是为什么没有在 XML 文件中包含它的原因。用这个属性标记的任何内容都不会存储到 XML 文件上，也不会从 XML 文件中加载它。

```
/// <summary>
/// Summary description for Entry.
/// </summary>
public class Entry
```

```
{
    // members...
    [XmlIgnore()] public DateTime Timestamp;
    public String Title;
    public String Details;
```

在创建 Entry 对象时，使用了 System.IO.FileInfo 对象，以说明上一次修改文件的日期：

```
// LoadEntry - load an entry from disk...
public static Entry LoadEntry(String filename)
{
...

    // close the file...
    file.Close();

    // update timestamp...
    newEntry.Timestamp = new FileInfo(filepath).LastWriteTime;

    // return the entry...
    return newEntry;
}
```

在显示页面时，使用两种方法把日期转换为字符串。第一种方法是 ToString()，它被配置为显示日期的名称。第二种方法是 ToLongDateString()，使用计算机的本地设置显示日期的内容。如果读者在 US，月份就会写在日期的前面。如果读者是在 UK，日期就会写在月份的前面。最好的方式是使用这个日期函数的本地化版本，否则就会使用户感到困惑。

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

    // set the server path...
    labelServerPath.Text = Global.EntryFilePath;

    // load the entry from disk...
    Entry entry = Global.LoadEntry("Entry.xml");
    labelEntryTitle.Text = entry.Title;
    labelEntryDetails.Text = entry.Timestamp.ToString("dddd") + ", " +
        entry.Timestamp.ToLongDateString()+ " - " + entry.Details;
    }
```

24.2.4　改善页面的外观

注意页面的外观的确非常丑陋。HTML 是(按照目前的说法)一项旧技术，主要用于共享单调的科学文档。Web 的用户主要采用它使页面看起来新颖而时尚。如果页面很新颖而时尚，就

必须自己动手。

有许多技术都可以用于改善页面的设计，但最简单的技术是使用 Cascading Style Sheets (级联式样式表，CSS)。CSS 是一个很大的主题，这里不打算详细介绍。简单地说，CSS 是一个特别格式化的文本文件，包含浏览器需要的所有字体、颜色和样式，能以更好的格式显示 HTML。

试试看：改善页面的外观

(1) 首先需要建立一个站点上所有页面都共享的样式表。为此，在 Solution Explorer 中右击 Weblog 工程，选择 Add | Add New Item。在 Templates 列表中搜索，找到 Style Sheet。选择它，并把名称设置为 style.css，单击 Open 按钮。

(2) 在新样式表中添加如下代码：

```
body
{
    padding-right: 0px;
    padding-left: 0px;
    font-size: 8pt;
    padding-bottom: 0px;
    margin: 0px;
    padding-top: 0px;
    font-family: Verdana, Arial;
}
.header
{
    padding-right: 5px;
    padding-left: 5px;
    padding-bottom: 10px;
    padding-top: 10px;
    background-color: #000099;
    font-weight: bold;
    font-size: 14pt;
    color: white;
}
.normal
{
    padding-right: 5px;
    padding-left: 5px;
    font-size:8pt;
}
.normalHeading
{
    padding-right: 5px;
    padding-left: 5px;
    font-size:12pt;
    font-weight: bold;
```

```
}
.entryTitle
{
    padding-right: 5px;
    padding-left: 5px;
    padding-bottom: 1px;
    padding-top: 1px;
    font-weight: bold;
    font-size: 10pt;
    color: white;
    background-color: #66cc99;
}
.entryDate
{
    font-weight: bold;
    color: #333399;
    font-size: 8pt;
}
.entry
{
    padding-right: 5px;
    padding-left: 5px;
    padding-top: 2px;
    font-size: 8pt;
}
```

如果不想键入这些代码，可以从下载的代码中添加 styles.css 文件；如果该文件还没有包含在工程中，可以在 Solution Explorer 中右击 WebLog 工程节点，选择 Add | Add Existing Item，浏览到 CSS 文件。

(3) 要使用样式表，必须把它关联到一个页面上。打开 Default.aspx 的 HTML 编辑器，把下面一行代码添加进去：

```
<html>
    <head>
        <meta content="Microsoft Visual Studio 7.0" name="GENERATOR">
        <meta content="C#" name="CODE_LANGUAGE">
        <meta content="JavaScript (ECMAScript)" name="vs_defaultClientScript">
        <meta content="http://schemas.microsoft.com/intellisense/ie5"
            name="vs_targetSchema">
        <link rel="stylesheet" href="style.css">
    </head>
```

(4) 现在修改突出显示的代码行，在需要的地方添加代码：

```
<%@ Page language="c#" Codebehind="Default.aspx.cs" AutoEventWireup="false"
Inherits="WebLog.CDefault" %>
```

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN" >
<html>
    <head>
        <meta content="Microsoft Visual Studio 7.0" name="GENERATOR">
        <meta content="C#" name="CODE_LANGUAGE">
        <meta content="JavaScript (ECMAScript)" name="vs_defaultClientScript">
        <meta content="http://schemas.microsoft.com/intellisense/ie5"
            name="vs_targetSchema">
        <link rel="stylesheet" href="style.css">
    </head>
    <body>
        <form id="Default" method="post" runat="server">
            <div class="header">
              · Disraeli's Weblog
            </div>
            <br>
            <div class="entryTitle">
                <asp:label id="labelEntryTitle" runat="server">
                    (entryTitle)</asp:label>
            </div>
            <div class="entry">
                <asp:label id="labelEntryDetails" runat="server">
                    (entryDetails)</asp:label>
            </div>
            <br>
            <hr color="#000000">
            <div class="normal">
                <asp:label id="labelCopyright" runat="server">
                    (copyright)</asp:label>
            </div>
            <br>
            <div class="normal">
                <asp:Label id="labelServerPath" runat="server">
                    (serverpath)</asp:Label>
            </div>
        </form>
    </body>
</HTML>
```

(5) 最后一个要修改的代码在 Default.aspx.cs 中。查看该文件的代码编辑器，对 Page_Load() 进行如下修改：

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

```
        // set the server path...
        labelServerPath.Text = Global.EntryFilePath;

        // load the entry from disk...
        Entry entry = Global.LoadEntry("Entry.xml");
        labelEntryTitle.Text = entry.Title;
        labelEntryDetails.Text = "<font class=entryDate>" +
            entry.Timestamp.ToString("dddd") + ", " +
            entry.Timestamp.ToLongDateString() + "</font> - " +
            entry.Details;
        }
```

编译并运行工程，得到如图 24-18 所示的结果。

代码的说明

CSS 可以定义能应用到页面的任何元素上的样式。我们在.css 文件中建立这些样式，例如刚才创建的 style.css 文件。

对 CSS 的讨论超出了本书的范围，但一般要点如下：可以修改标准的 HTML 元素的样式，例如<body> 和 ，也可以定义新类，在 CSS 术语中，是把样式组合到一个单元中。要修改已有的元素，只需输入元素的名称，添加 CSS 代码即可。下面重新定义了如何显示<body>标记：

```
body
{
    padding-right: 0px;
    padding-left: 0px;
    font-size: 8pt;
    padding-bottom: 0px;
    margin: 0px;
    padding-top: 0px;
    font-family: Verdana, Arial;
}
```

可以给新类指定任意属性名，但必须在前面加一个句点，下面是建立 header 类的过程：

```
.header
{
    padding-right: 5px;
    padding-left: 5px;
    padding-bottom: 10px;
    padding-top: 10px;
    background-color: #000099;
    font-weight: bold;
    font-size: 14pt;
    color: white;
}
```

从技术上看，不必理解 CSS，就可以在 Visual Studio.NET 中编辑它。找到样式表的代码，如果样式表就是当前选中的文档，则从菜单中选择 Windows | Document Outline，就可以打开 CSS 编辑器，如图 24-19 所示。

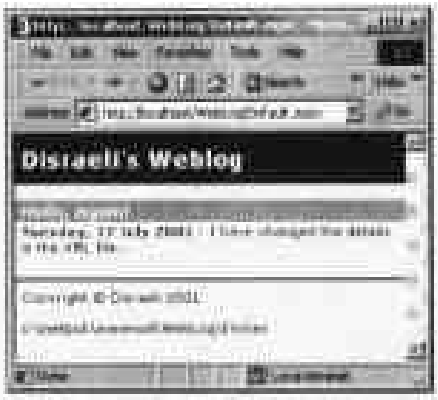


图 24-18



图 24-19

如果右击任一个元素或类，并选择 Build Style，就会显示一个对话框，通过该对话框，可以用直观的合理的方式修改各种 CSS 代码。

建立了样式表之后，就需要把它关联到页面上。这将提示浏览器下载样式，并把它们应用到文档上，用一个 link 元素就可以实现，如下所示：

```
<html>
    <head>
        <meta content="Microsoft Visual Studio 7.0" name="GENERATOR">
        <meta content="C#" name="CODE_LANGUAGE">
        <meta content="JavaScript (ECMAScript)" name="vs_defaultClientScript">
        <meta content="http://schemas.microsoft.com/intellisense/ie5"
              name="vs_targetSchema">
        <link rel="stylesheet" href="style.css">
    </HEAD>
```

最后，要设置页面上的元素所使用的类，可以使用 class 属性告诉浏览器将把什么样式应用到元素上。下面的代码说明如何告诉包含 Disraeli's Weblog 的 div 元素来使用 header 类：

```
<div class="header">
    Disraeli's Weblog
</div>
```

本章的其余部分都使用这个样式表，但注意我们不打算详细介绍它。

24.2.5　显示项目列表

可以显示一个 weblog 项目，但如何在同一个页面上显示许多项目呢？

显示列表是活动 Web 站点常常执行的一个操作，它可以在电子商务 Web 站点上显示销售产品的列表，或者显示数据库中的顾客记录。ASP.NET 有许多方式可以轻松地显示列表，下面就介绍其中的一些方式。

试试看：显示 Weblog 项目列表

(1) 在浏览 Weblog 项目列表之前，需要在 Entries 文件夹中存储多个项目。使用 Explorer 和 Notepad，创建一些项目，可以使用自己喜欢的任何名称，如图 24-20 所示。



图 24-20

(2) 在 Global 类中，需要一个新的静态方法，该方法返回一个包含所有项目的列表。打开 Global.asax.cs 的代码编辑器，添加如下代码：

```
// LoadAllEntries - load all entries from disk...
public static Entry[] LoadAllEntries()
{
    // get the path containing the entries...
    DirectoryInfo entryFolder = new DirectoryInfo(EntryFilePath);

    // get a list of files...
    FileInfo[] files = entryFolder.GetFiles();
    // create an array of entries...
    Entry[] entries = new Entry[files.Length];

    // loop through and load each file...
    int index = 0;
    foreach(FileInfo file in files)
    {
        entries[index] = LoadEntry(file.Name);
        index++;
    }

    // return the list...
    return entries;
}
```

(3) 现在需要删除 Default.aspx 中创建两个标签控件的代码。打开设计器，删除这两个标签控件，得到如图 24-21 所示的结果。

(4) 现在删除两个 div 标记，其结果如图 24-22 所示。

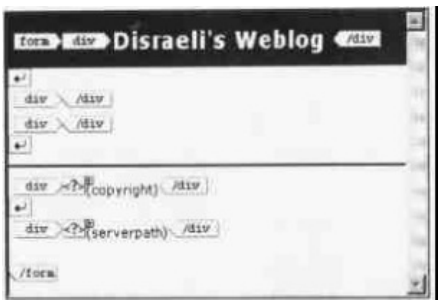


图　24-21

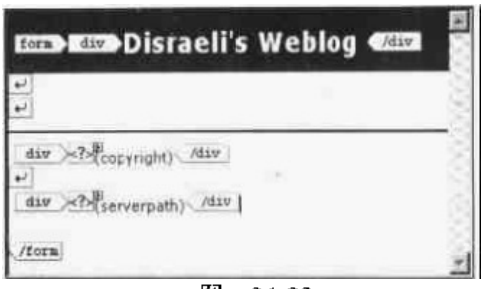


图　24-22

(5) 接着使用工具箱，把一个 DataList 控件拖放到两个回车符之间，得到如图 24-23 所示的结果。

(6) 这里把 DataList 控件描述为“需要有高精度”。它需要非常多的修改和 UI 处理，但它是一个功能非常强大的控件，以一种彻底的方式使用数据绑定，即在 Page_Load()中，必须给控件提供 Entry 对象数组。它将迭代整个列表，每次迭代都会显示我们定义的一系列控件。每个控件都是把数据绑定到 Entry 类的一个属性上。所以可以在 DataList 控件中添加一个 Label 控件，把 Label 控件的 Text 属性绑定到 Title 属性上，等等。

(7) 在进行处理之前，把 DataList 控件的 ID 属性改为 datalistEntries。

(8) 右击 DataList 控件，选择 Edit Template | Item Template，把控件放在一个单独的模式中，在该模式中，可以处理每次对 Entry 对象数组的迭代。

(9) 使用工具箱，把一个 Label 控件拖放到 ItemTemplate 下面的门色区域中，如图 24-24 所示。

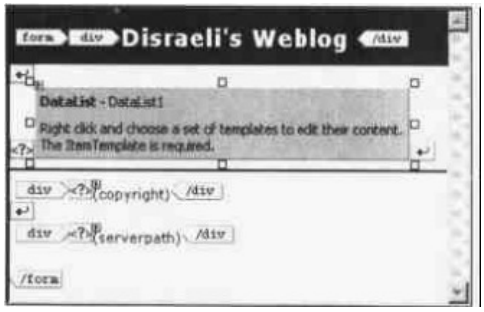


图　24-23



图　24-24

显示在 ItemTemplate 项目中的任何控件都将在数组的每次迭代中显示。所以，如果数组中有 3 个 Entry 项目，就会显示 3 个标签控件。

(10) 范动作选择 Label 控件，使用 Properties 窗口，把 CssClass 属性改为 entryTitle。控件的外观应有所改变，以反映刚才选中的样式。

(11) 现在查找 DataBindings 项目，并选择它。单击它旁边的省略号(...)，打开数据绑定窗口。

左边的列表显示了 Label 控件上可以进行数据绑定的所有属性。应选择 Text 属性。选择了该属性后，对窗口右边的控件进行的任何操作都将配置 Text 属性的数据绑定。

(12) 选择 Custom binding expression 单选按钮，把下面的代码输入到该单选按钮的文本框中：

```
DataBinder.Eval(Container, "DataItem.Title")
```

得到如图 24-25 所示的结果。

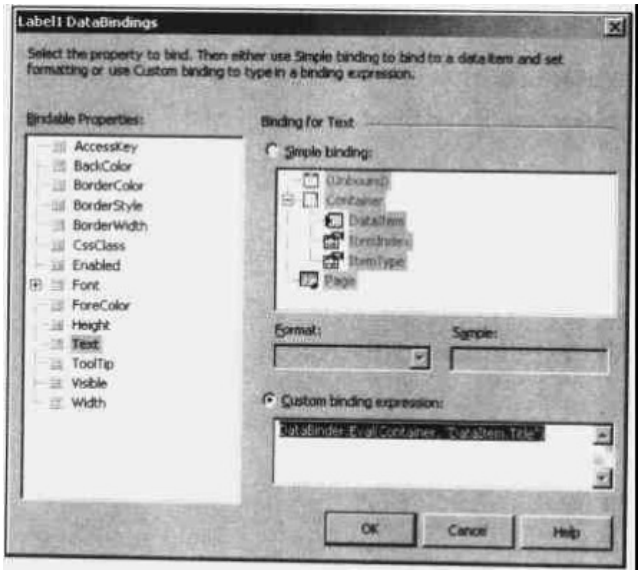


图　24-25

(13) 单击 OK 按钮，保存数据绑定。

(14) 下面要验证这部分工作，再显示日期和其他内容。双击页面的背景，打开 Load 事件处理程序，添加如下代码：

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

        // set the server path...
        labelServerPath.Text = Global.EntryFilePath;

        // load all of the entries from disk...
        Entry[] entries = Global.LoadAllEntries();
        datalistEntries.DataSource = entries;
        datalistEntries.DataBind();
    }
```

现在编译并运行工程，得到如图 24-26 所示的结果。



图　24-26

代码的说明

要在磁盘上创建一个包含所有 Entry 对象的数组，其代码是非常简单的。我们只需请求 System.IO.DirectoryInfo 对象给 Entries 文件夹返回一个 System.IO.FileInfo 对象列表，再迭代列表 LoadEntry 即可：

```
// LoadAllEntries - load all entries from disk...
public static Entry[] LoadAllEntries()
{
    // get the path containing the entries...
```

```
        DirectoryInfo entryFolder = new DirectoryInfo(EntryFilePath);

        // get a list of files...
        FileInfo[] files = entryFolder.GetFiles();

        // create an array of entries...
        Entry[] entries = new Entry[files.Length];

        // loop through and load each file...
        int index = 0;
        foreach(FileInfo file in files)
        {
            entries[index] = LoadEntry(file.Name);
            index++;
        }

        // return the list...
        return entries;
    }
```

有了列表后，在 Page_Load() 上，把它传送给 DataList 控件，并调用 DataBind()：

```
    private void Page_Load(object sender, System.EventArgs e)
    {
...
        // set the server path...
        labelServerPath.Text = Global.EntryFilePath;

        // load all of the entries from disk...
        Entry[] entries = Global.LoadAllEntries();
        datalistEntries.DataSource = entries;
        datalistEntries.DataBind();
    }
```



图 24-27

这段代码会请求 DataList 迭代每个 Entry 对象，且遵循在模板上定义的规则。我们定义的惟一模板规则是在每次迭代时，必须显示一个 Label 控件，这个 Label 控件的 Text 属性值应设置为当前显示的 Entry 对象的 Title 属性值，如图 24-27 所示。

1. 显示细节

我们可以显示标题，但其他信息该怎么办？本节介绍如何显示细节和日期。

试试看：显示 Weblog 项目的其他信息

(1) 回忆一下，前面显示 weblog 项目的日期的方式是相当复杂的。因为 DataList 用于公共属性，所以要在 Entry 上建立一个属性，它可以把日期返回为一个字符串。打开 Entry 的代码编辑器，添加如下代码：

```
public String TimestampAsString
{
    get
    {
        return Timestamp.ToString("dddd") + ", " +
                        Timestamp.ToLongDateString();
    }
}
```

(2) 打开 Default.aspx 的设计器，DataList 控件可能拖放到不能直接编辑的状态下，如图 24-28 所示。

(3) 右击控件，再次选择 Edit Template | Item Template。把光标放在绿色的 Label 控件后面，按住 Shift 键的同时，并按下回车键。这将添加一个换行符，而不是一个回车符。如果添加一个回车符，在标题和项目之间的空间就会太大。

(4) 在已有的绿色 Label 控件下面添加一个 Label 控件。在控件的后面添加一个空格、一个短横线和一个空格之后，再添加一个 Label 控件，如图 24-29 所示。



图　24-28

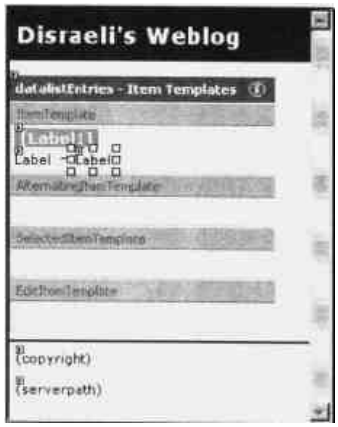


图　24-29

注意：

每个控件左上角的绿色小箭头表示一个控件。

(5) 选择第一个新增的 Label 控件，把它的 CssStyle 属性设置为 entryDate。

(6) 找到并选择 DataBindings 属性。单击省略号，打开 DataBindings 窗口，确保选择了左边列表中的 Text 属性，选中 Custom binding expression 单选按钮，添加如下代码：

```
DataBinder.Eval(Container, "DataItem.TimestampAsString")
```

(7) 单击 OK 按钮，保存绑定。

(8) 选择另一个新增的 Label 控件，把它的 CssStyle 属性设置为 entry。

(9) 找到并选择 DataBindings 属性。单击省略号，打开 DataBindings 窗口，选择左边列表中的 Text 属性，单击 Custom binding expression 单选按钮，添加如下代码：

```
DataBinder.Eval(Container, "DataItem.Details")
```

现在编译并运行工程，得到如图 24-30 所示的结果。

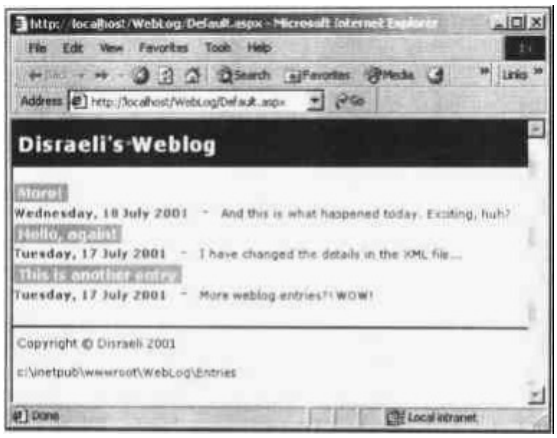


图　24-30

代码的说明

因为 DataList 控件最好使用属性来工作，所以我们创建了一个只读属性 TimestampAsString，它会返回 Timestamp 属性的格式化版本。我们没有给这个属性添加 set 子句，因为这表示对象的用户可能要求我们在字符串之外再建立一个 System.DateTime 值，尽管这是可行的，但这超出了该对象的作用域。

```
public String TimestampAsString
{
    get
    {
        return Timestamp.ToString("dddd") + ", " +
                        Timestamp.ToLongDateString();
    }
}
```

添加了属性后，给项目模板添加更多的标签控件就只是一个重复以前工作的问题了。

2. DataList 和公共成员的重要说明

给 DataList 控件输入到 Custom binding expression 字段中的表达式仅能处理属性，不能处理公共成员。如果输入：

```
public class Entry
{
    // members...
    Public String Title;
```

并试图用下面的表达式绑定到 Title 上：

```
DataBinder.Eval(Container, "DataItem.Title")
```

就会得到一个找不到该属性的错误。此时，惟一的选择是把成员转换为一个属性，如下

所示：

```
public String Title
{
    get
    {
        return _title;
    }
    set
    {
        _title = value;
    }
}
```

24.3　创建新 Weblog 项目

现在，我们需要获取的和显示站点上项目列表的所有功能都已经被添加进去了。但是，此时不使用 Notepad 仍不能创建新项目。最好能在站点上添加一个窗体，用于创建新项目。

试试看：创建 Weblog 项目

(1) 使用 Solution Explorer，右击 Weblog 工程，选择 Add | Add Web Form。输入名称 Edit.aspx。

(2) 打开该页面的设计器，单击背景，使用 Properties 窗口把 pageLayout 属性改为 FlowLayout。

(3) 从菜单中选择 View | HTML Source。添加如下代码：

```
<%@ Page language="c#" Codebehind="Edit.aspx.cs" AutoEventWireup="false" Inherits="WebLog.Edit"
%>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN" >
<html>
    <head>
        <meta name="GENERATOR" Content="Microsoft Visual Studio 7.0">
        <meta name="CODE_LANGUAGE" Content="C#">
        <meta name="vs_defaultClientScript" content="JavaScript (ECMAScript)">
        <meta name="vs_targetSchema"
            content="http://schemas.microsoft.com/intellisense/ie5">
        <link rel="stylesheet" href="style.css">
    </head>
    <body>
        <form id="Edit" method="post" runat="server">
            <div class="header">
                Disraeli's Weblog
            </div>
```

```
        <br>
        <div class="normalHeading">
            Create a New Entry
        </div>
        <br>
        <div class="normal">
            <table cellspacing="0" cellpadding="3">
                <tr>
                    <td class="normal">
                        Title:
                    </td>
                    <td>
                    </td>
                </tr>
                <tr>
                    <td class="normal">
                        Details:
                    </td>
                    <td>
                    </td>
                </tr>
                <tr>
                    <td colspan="2" align="right">
                    </td>
                </tr>
            </table>
        </div>
    </form>
</body>
</html>
```

(4) 从菜单中选择 View | Design，得到如图 24-31 所示的结果。

(5) 使用工具箱，把一个 TextBox 控件拖放到 Title 右边的小方框中。重复这个操作，但这次是把控件拖放到右边的小方框中，如图 24-32 所示。

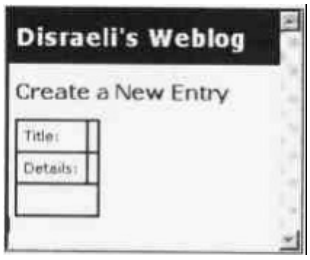


图　24-31

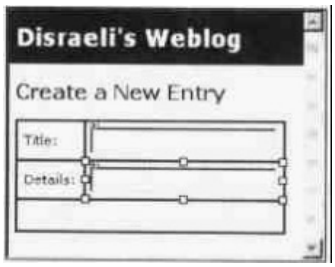


图　24-32

- 把上面 TextBox 控件的 ID 属性改为 textTitle。
- 把下面 TextBox 控件的 ID 属性改为 textTitle，再把 TextMode 属性改为 Multiline。

(6) 使用工具箱，把一个按钮控件拖放到底部空白的长方框中。把它的 ID 属性改为 buttonOk，并把它的 Text 属性改为 Save Changes。

(7) 使两个文本框控件略大一些，如图 24-33 所示。

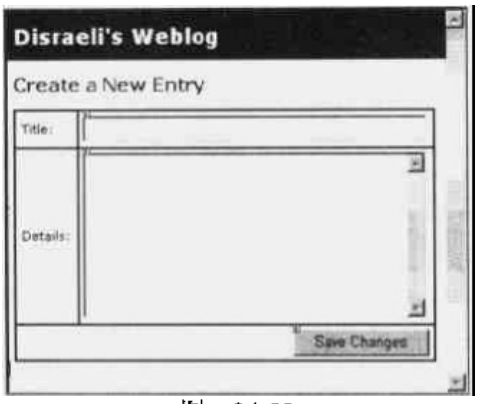


图　24-33

(8) 单击 Save Changes 按钮，除非用户填充了这两个文本框，否则我们不希望创建新项目。可以使用 ASP.NET 的有效性验证功能来自动确保所输入的细节，如果没有输入这些内容，将显示一个信息。

(9) 从工具箱中把一个 RequiredFieldVallDation 控件拖放到 Title 文本框的右边，把它的 ErrorMessage 属性改为 Required，把 ControlToVallDate 属性设置为 textTitle。

(10) 重复这个操作，但这次把新控件拖放到 Details 框的右边。把它的 ErrorMessage 属性改为 Required，把 ControlToVallDate 属性设置为 textTitle。

(11) 使用 Solution Explorer，查看 Entry 的代码编辑器，添加如下成员：

```
/// <summary>
/// Summary description for Entry.
/// </summary>
public class Entry
{
    // members...
    private String _filename;
    private DateTime _timestamp;
    private String _title;
    private String _details;
```

(12) 接着添加如下属性：

```
    [XmlIgnore()] public String Filename
    {
        get
        {
            return _filename;
        }
        set
        {
```

```
            _filename = value;
        }
    }
```

(13) 现在给 Entry 类添加下面的方法：

```
public void Save()
{
    // do we have a filename?
    if(Filename == null)
    {
        // get a filename based on the date...
        Timestamp = DateTime.Now;
        Filename =
            String.Format("{0:d4}{1:d2}{2:d2}_{3:d2}{4:d2}.xml",
                (int)Timestamp.Year, (int)Timestamp.Month,
                (int)Timestamp.Day, (int)Timestamp.Hour,
                (int)Timestamp.Minute);
    }

    // get the whole filename...
    String filepath = Global.EntryFilePath + "\\" + Filename;

    // create a serializer and save...
    FileInfo fileInfo = new FileInfo(filepath);
    if(fileInfo.Exists == true)
        fileInfo.Delete();
    FileStream stream =
            new FileStream(fileInfo.FullName, FileMode.Create);
    XmlSerializer serializer = new XmlSerializer(this.GetType());
    serializer.Serialize(stream, this);
    stream.Close();
}
```

(14) 打开 Edit.aspx.cs 的代码编辑器，找到 Page_Load()方法，添加如下代码：

```
private void Page_Load(object sender, System.EventArgs e)
{
    // is the page being saved?
    if(IsPostBack == true)
    {
        // create a new entry object...
        Entry newEntry = new Entry();

        // set the values...
        newEntry.Title = textTitle.Text;
        newEntry.Details = textDetails.Text;
```

```
            // save it...
            newEntry.Save();

            // show the list...
            Response.Redirect("default.aspx");
        }
    }
```

(15) 接着打开 Default.aspx 的 HTML 编辑器，在页面上添加如下代码，为了简单起见，这里省略了前面给页面添加的一些代码：

```
<asp:DataList id="datalistEntries" runat="server">
...
</asp:DataList>
<br>
<div class="normal">
    <a href="edit.aspx">Create a new entry</a>
</div>
<br>
<hr color="#000000">
<div class="normal">
    <asp:label id="labelCopyright" runat="server">(copyright)</asp:label>
</div>
```

运行工程，单击 Create a new entry 链接，显示窗体，填充该窗体，并单击 Save Changes。该页面将重新加载，显示新项目，如图 24-34 所示。

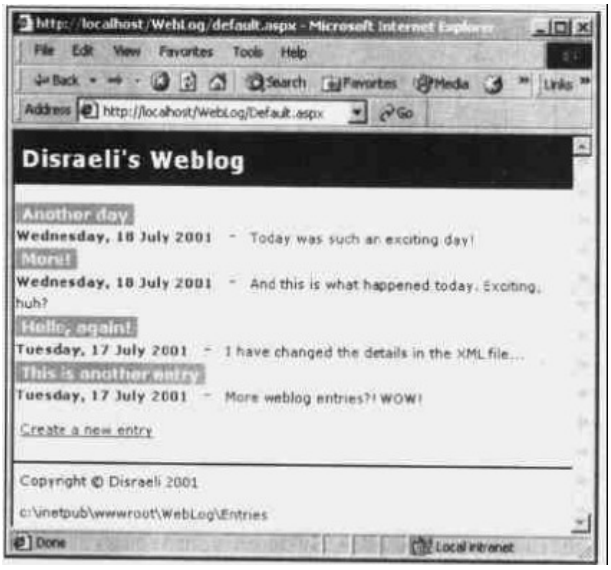


图　24-34

如果查看 Entries 文件夹，则会看到在其中创建了一个新文件，如图 24-35 所示。



图　24-35

代码的说明

在 Edit.aspx 中建立窗体与建立 Default.aspx 页面没有什么不同。在建立这两种不同的对象时，都使用了 HTML 代码和编辑器。

在单击窗体上的按钮时，会发生两种情况：执行有效性验证代码，对添加到页面上的控件进行有效性验证。这两种情况都是针对同一个文本框控件的。单击按钮时，验证要运行的代码，确保填充了控件，如果不验证运行的代码，就会显示一个信息，如图 24-36 所示。

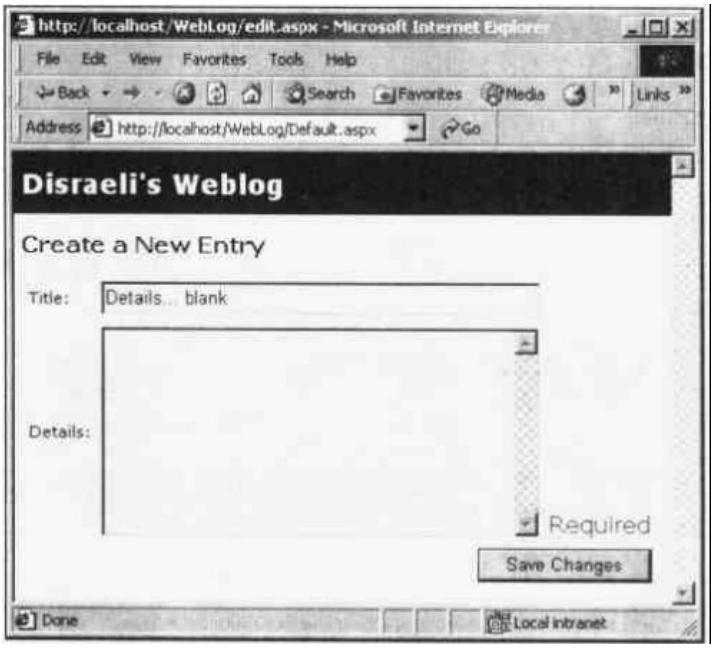


图　24-36

但如果有效性验证是成功的，页面就会被发送回 ASP.NET。此时，再次调用 Page_Load() 方法，但可以使用 IsPostBack 属性，看看页面是否真的已被传送回 ASP.NET：

```
private void Page_Load(object sender, System.EventArgs e)
{
    // is the page being saved?
    if(IsPostBack == true)
    {
```

如果页面被传送回来，就创建一个新的 Entry 对象：

```
        // create a new entry object...
        Entry newEntry = new Entry();
```

传送回页面后，文本框控件的 Text 属性就会自动填充用户所输入的内容，这表示可以：

```
        // set the values...
        newEntry.Title = textTitle.Text;
        newEntry.Details = textDetails.Text;
```

接着，告诉 Entry 对象保存它自己(详见后面的内容)：

```
        // save it...
        newEntry.Save();
```

保存了项目后，告诉 ASP.NET 请求浏览器导航到 default.aspx，查看新项目：

```
        // show the list...
        Response.Redirect("default.aspx");
    }
}
```

在保存项目时，需要一个文件名。文件名必须是惟一的(否则就会删除其他项目)，因此根据项目保存的日期和时间来命名项目：

```
public void Save()
{
    // do we have a filename?
    if(Filename == null)
    {
        // get a filename based on the date...
        Timestamp = DateTime.Now;
        Filename =
        String.Format("{0:d4}{1:d2}{2:d2}_{3:d2}{4:d2}.xml",
                (int)Timestamp.Year, (int)Timestamp.Month,
                (int)Timestamp.Day, (int)Timestamp.Hour,
                (int)Timestamp.Minute);
    }
```

把文件名存储为一个属性，这么做的原因是很显然的，详见后面的内容——要保存它，需要把文件名转换为一个完整的物理路径：

```
        // get the whole filename...
        String filepath = Global.EntryFilePath + "\\" + Filename;
```

之后，使用 XmlSerializer 对象把对象保存到磁盘上。这个串行化方法的参数有一个流对象参数和 Entry 对象，流对象表示要写入的文件，该方法可以把 Entry 对象的属性转换为一个 XML 字符串，与前面 Deserialize()函数的功能正好相反：

```
        // create a serializer and save...
        FileInfo fileInfo = new FileInfo(filepath);
        if(fileInfo.Exists == true)
            fileInfo.Delete();
        FileStream stream =
                new FileStream(fileInfo.FullName, FileMode.Create);
        XmlSerializer serializer = new XmlSerializer(this.GetType());
        serializer.Serialize(stream, this);
        stream.Close();
    }
```

重新加载了 Default.aspx 后，GetAllEntries()就会返回文件的完整列表，包括刚才新建的文件。

24.3.1　编辑 Weblog 项目

现在有了一个能用于创建新项目的窗体。是否可以使用它编辑在磁盘上存储为 XML 文件的现有项目？

本节介绍如何修改应用程序，使之在加载后可编译项目。

试试看：编辑 Weblog 项目

(1) 打开 Default.aspx 的设计器。

(2) 右击 DataList 控件，选择 Edit Templates | Item Template。

(3) 使用工具箱，把一个新 Hyperlink 控件拖放到绿色 Label 控件的旁边，如图 24-37 所示。



图 24-37

(4) 使用 Properties 窗口，选择链接控件的 DataBindings 属性，在左边的列表中选择 NavigateUrl。

(5) 选中 Custom binding expression 单选按钮，输入下述表达式：

```
"edit.aspx?filename=" + DataBinder.Eval(Container, "DataItem.Filename")
```

(6) 单击 OK 按钮，把改变保存到绑定中。

(7) 再次使用 Properties 窗口，把链接控件的 Text 属性改为 Edit，把 CssClass 属性改为 entry。

(8) 打开 Global.asax 的代码编辑器，找到 LoadEntry()方法，进行如下修改：

```
// LoadEntry - load an entry from disk...
public static Entry LoadEntry(String filename)
{
    // we have the name, but we need the path...
    String filepath = EntryFilePath + "\\" + filename;

    // open the file...
    FileStream file = new FileStream(filepath, FileMode.Open);

    // create a serializer...
    XmlSerializer serializer = new XmlSerializer(typeof(Entry));
    Entry newEntry = (Entry)serializer.Deserialize(file);

    // close the file...
    file.Close();

    // update timestamp and filename...
    newEntry.Timestamp = new FileInfo(filepath).LastWriteTime;
    newEntry.Filename = filename;

    // return the entry...
    return newEntry;
}
```

运行工程，单击一个 Edit 链接，显示 Create new Entry 窗体。但是我们真正感兴趣的是 Address 栏。如果看到这一栏，注意包含项目数据的 XML 文件的文件名已被内嵌到 URL 中，如图 24-38 所示。



图　24-38

代码的说明

现在还不能编辑页面，但下一步就是修改 Edit.aspx 和 Edit.asp.cs，以便在 URL 中查找文件名项目，并加载该项目。

这里比较有趣的是数据绑定。在新建的超链接控件上，把 NavigateUrl 属性绑定到下面的表达式上：

```
"edit.aspx?filename=" + DataBinder.Eval(Container, "DataItem.Filename")
```

可以看出，每次迭代 Entry 对象数组时，都要提取对象的 Filename 属性，并把它放在 edit.aspx?filename=字符串的末端，这样就得到了一个完整的 URL。

加载选中的项目

要加载在查看 Edit.aspx 时选中的项目，只需查找可能或不可能包含在 URL 中的 filename 参数即可。

试试看：加载选中的项目

(1) 打开 Entry.aspx.cs 的代码编辑器，找到 Page_Load()方法，添加如下代码：

```
private void Page_Load(object sender, System.EventArgs e)
{
    // is the page being saved?
    if(IsPostBack == true)
    {
        // create a new entry object...
        Entry newEntry = new Entry();

        // do we have a filename to use?
        if(Request.Params["filename"] != null)
            newEntry.Filename = Request.Params["filename"];

        // set the values...
        newEntry.Title = textTitle.Text;
        newEntry.Details = textDetails.Text;

        // save it...
        newEntry.Save();

        // show the list...
        Response.Redirect("default.aspx");
    }
    else
    {
        // did we get a filename?
        String filename = Request.Params["filename"];
        if(filename != null)
        {
            // load the entry object...
            Entry entry = Global.LoadEntry(filename);

            // populate the fields...
            textTitle.Text = entry.Title;
                        textDetails.Text = entry.Details;
        }
    }
}
```

(2) 运行工程。如果单击 Edit 链接，项目就会加载到新页面上。当单击了 Save Changes 时，在窗体上对数据进行的任何修改都将保存到 XML 文件中。

代码的说明

在.NET 中，参数可以以两种方式输入页面：通过窗体变量或通过查询字符串。大体上，ASP.NET 能很好地运用窗体在 Web 上的工作方式，我们惟一需要考虑的是查询字符串变量。

这些类型的变量是 URL 中显示在问号后面的部分。在范例中，有一个变量 filename，其值是 20010718_1419.xml：

```
http://localhost/WebLog/edit.aspx?filename=20010718_1419.xml
```

但这个 URL 有两个变量 a 和 b；a 设置为 Jack，b 设置为 Coffee，这两个变量用一个宏符号(&)分隔开：

```
http://anotherserver/page.aspx?a=Jack&b=Coffee
```

在 Page_Load()中，如果 IsPostBack 返回 false，则表示页面是第一次加载。此时可以把 Text 属性设置为文件中存储的内容(这里省略了一些代码)：

```
private void Page_Load(object sender, System.EventArgs e)
{
    // is the page being saved?
    if(IsPostBack == true)
    {
    ...
    }
    else
    {
        // did we get a filename?
        String filename = Request.Params["filename"];
        if(filename != null)
        {
```

Request.Params 属性被突出显示出来，它可以用于访问传送给页面的参数。在这个范例中，我们要获取 filename 参数。当然，如果单击了 Create new entry 链接，这个参数就是空白的。如果有一个参数，就可以从磁盘上加载项目，并更新每个控件上的 Text 属性：

```
            // load the entry object...
            Entry entry = Global.LoadEntry(filename);

            // populate the fields...
            textTitle.Text = entry.Title;
                    textDetails.Text = entry.Details;
        }
    }
}
```

在传送回页面时，会出现这个问题的另一面。此时需要再查看一下 Request.Params，确定

是否已经有了一个文件名。如果是，就可以使用它，此时 Save()会用包含改变的新文件替换已有的文件。如果没有该文件名，Save()就使用当前的日期和时间创建一个新文件名(这里省略了一些代码)：

```
private void Page_Load(object sender, System.EventArgs e)
{
    // is the page being saved?
    if(IsPostBack == true)
    {
        // create a new entry object...
        Entry newEntry = new Entry();

        // do we have a filename to use?
        if(Request.Params["filename"] != null)
            newEntry.Filename = Request.Params["filename"];

        // set the values...
        newEntry.Title = textTitle.Text;
        newEntry.Details = textDetails.Text;

        // save it...
        newEntry.Save();

        // show the list...
        Response.Redirect("default.aspx");
    }
```

现在可以显示项目列表，使用 Web 站点本身创建新项目，对已有的项目做出修改。

24.3.2　给项目排序

注意在添加更多的项目时，最后一个创建的项目并没有显示在页面的底部。但这一般是 weblog 的工作方式，第一个项目是最新的。

在从目录中读取项目时，需要按照日期对它们排序。幸运的是，.NET 以字母顺序返回文件，这就是为什么使用前面的文件名格式的原因。考虑下面 3 个文件名：

```
20010718_1318.xml
20010719_1003.xml
20010720_1823.xml
```

这里使用的格式是 yyyymmdd_hhmm，其缺点是使用站点每分钟只能创建一个文件。在我们看来，这没有什么大问题，因为我们通常一天只创建一个新文件。这种格式的文件在按字母顺序排序时，会自动从最早排至最迟。我们仅需要对存储在磁盘上文件颠倒顺序即可。

试试看：对项目排序

(1) 首先，使用 Web 工具创建至少两个项目。删除不遵循刚才所述日期格式的已有 XML 文件。假定有 3 个文件，如图 24-39 所示。

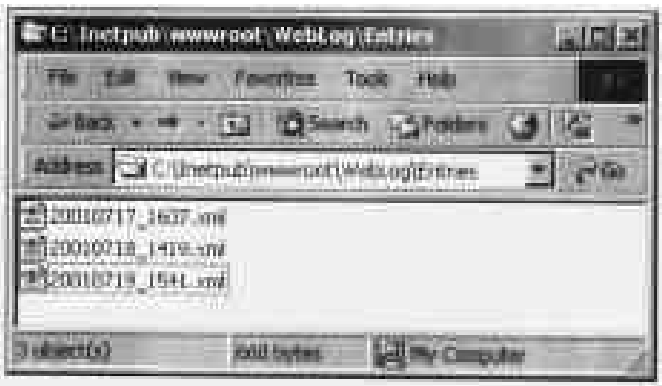


图 24-39

可以看出，这些文件按照字母顺序列出，其顺序与我们需要的正好相反。

(2) 打开 Global.asax 的代码编辑器，找到 LoadAllEntries，对该方法进行如下两处修改：

```
// LoadAllEntries - load all entries from disk...
public static Entry[] LoadAllEntries()
{
    // get the path containing the entries...
    DirectoryInfo entryFolder = new DirectoryInfo(EntryFilePath);

    // get a list of files...
    FileInfo[] files = entryFolder.GetFiles();

    // create an array of entries...
    Entry[] entries = new Entry[files.Length];

    // loop through and load each file...
    int index = files.Length - 1;
    foreach(FileInfo file in files)
    {
        entries[index] = LoadEntry(file.Name);
        index--;
    }

    // return the list...
    return entries;
}
```

编译并运行工程，文件应以正确的顺序显示，如图 24-40 所示。



图　24-40

代码的说明

这里的技巧是以文件在磁盘上显示顺序的逆序来填充数组。在创建数组时，数组的长度应与 GetFiles()返回的 FileInfo 对象数组的长度相同：

```
// LoadAllEntries - load all entries from disk...
public static Entry[] LoadAllEntries()
{
    // get the path containing the entries...
    DirectoryInfo entryFolder = new DirectoryInfo(EntryFilePath);

    // get a list of files...
    FileInfo[] files = entryFolder.GetFiles();

    // create an array of entries...
    Entry[] entries = new Entry[files.Length];
```

在迭代每个 FileInfo 对象时，让 Global.LoadEntry 创建一个新 Entry 对象。把这个新对象存储在数组中，其位置由 index 指定，而 index 从数组的最后一个位置开始，每迭代一次，index 就递减 1，直到迭代完 files 数组为止：

```
    // loop through and load each file...
    int index = files.Length - 1;
    foreach(FileInfo file in files)
    {
        entries[index] = LoadEntry(file.Name);
        index--;
    }

    // return the list...
    return entries;
}
```

24.4　用户会话和 Cookies

这并不是把注意力转移到建立一个允许任何人访问的 Web 站点，创建新 weblog 项目，编辑旧项目。这有一点愚蠢，理论上只有 Web 站点拥有者才能做出修改。

现在需要提供一种方式，让站点拥有者可以验证自己的身份。这样，只要给 Web 站点的用户提供修改站点的机会，就可以检查用户的身份，允许或禁止他修改 Web 站点。

会话的工作方式是，允许 Web 服务器把一组数据关联到特定的 Web 浏览器上。在这种情况下，浏览器会在给定站点的主页请求上加上标签，这个标签是惟一的，在以后的请求中，这个标签的作用就像一把钥匙，用以打开与它相关的数据。

Web 服务器使用一个 Cookie 来标记用户的浏览器(Cookie 是一小段信息，Web 服务器会把它发送给客户机器，只要以后访问该站点，就会读取它)。在对站点的主页进行请求时，Cookie 还不存在，所以 Web 服务器会创建一个新的用户会话，把该会话的 ID 放在一个 Cookie 中，然后把它发送给浏览器。在 ASP.NET 代码中，可以设置各种属性(详见后面的内容)，把信息添加到会话中。

可以在会话中存储许多想要存储的内容，但它们会占用服务器上的内存，这种信息应存储得尽可能少。例如，如果可以在一个会话中存储客户相当小的 ID 时，是否需要在一个会话中把所有的内容都存储在客户的机器上？这样，在需要它们时，就要查找很琐碎的信息。

限制对 Web 站点的访问

在这个范例中，我们将在会话中存储一个布尔值，它告诉我们用户是否可以编辑站点上的信息。

试试看：禁止编辑

(1) 首先，打开 Global.asax 的代码编辑器，找到 Session_Start()方法，添加如下代码：

```
protected void Session_Start(Object sender, EventArgs e)
{
    // configure the session...
    Session["canedit"] = false;
}
```

(2) 接着，打开 Edit.aspx 的代码编辑器，找到 Page_Load()方法，在顶部添加如下代码：

```
private void Page_Load(object sender, System.EventArgs e)
{
    // are we allowed to be edited...
    if((bool)Session["canedit"] == false)
        Response.Redirect("CannotEdit.aspx");
```

(3) 然后使用 Solution Explorer，右击 Weblog 项目，选择 Add | Add Web Form。把新窗体命名为 CannotEdit。打开新页面的 HTML 编辑器，添加如下代码：

```
<%@ Page language="c#" Codebchind="CannotEdit.aspx.cs" AutoEventWireup="false"
Inherits="WebLog.CannotEdit" %>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN" >
<html>
    <head>
        <meta name="GENERATOR" Content="Microsoft Visual Studio 7.0">
        <meta name="CODE_LANGUAGE" Content="C#">
        <meta name="vs_defaultClientScript" content="JavaScript (ECMAScript)">
        <meta name="vs_targetSchema"
              content="http://schemas.microsoft.com/intellisense/ie5">
    </head>
    <body MS_POSITIONING="GridLayout">
        <form id="CannotEdit" method="post" runat="server">
            Sorry, you're not allowed to make changes to the site.
            <br>
            <br>
            <a href="default.aspx">Continue...</a>
        </form>
    </body>
</html>
```

编译并运行工程，注意 Edit 链接和 Create new entry 链接会进入错误的页面，而不是前面访问的页面，如图 24-41 所示。

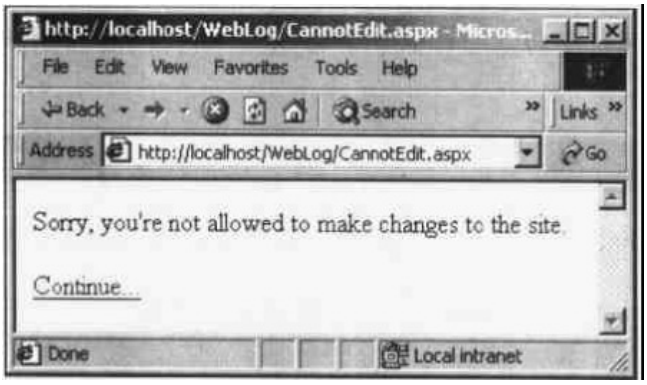


图　24-41

代码的说明

这里的技巧就是会话。会话是 ASP.NET 提供的，用于标识一个用户。用户第一次请求 ASP 或 ASP.NET 站点上的页面时，页面会被分配一个惟一的会话，其他会话都与它们不同。

只要创建了一个新会话，即只要新用户请求站点上的主页时，就调用 Session_Start()。System.Web.HttpApplication 和 System.Web.UI.Page 都支持一个属性 Session，这个属性是一个基本集合，可以在其中存储值。在这个范例中，我们创建了一个值 canedit，并为它存储了值 false：

```
protected void Session_Start(Object sender, EventArgs e)
{
    // configure the session...
```

```
        Session["canedit"] = false;
    }
```

这些值称为会话变量。

可以从站点的任一页面上获取这个值。当获取这个值时，就是在访问会话变量，获取对用户来说是惟一的值。所以，如果有 100 个用户都在使用这个站点，每个用户都会有惟一的会话。在创建会话时，Canedit 变量设置为初始值 false。但是，如果把一个用户的 Canedit 改为 true，这个变化将只影响这一个用户。基本上，对于能提供正确密码的用户，就可以把 Canedit 改为 true。

在 Edit.aspx.cs 中，在加载页面时，检查一下这个值是否为 false，如果是，就把用户重定向为另一个页面：

```
private void Page_Load(object sender, System.EventArgs e)
{
    // are we allowed to be edited...
    if((bool)Session["canedit"] == false)
        Response.Redirect("CannotEdit.aspx");
```

用户的身份验证

在这个练习中，我们将创建一个非常简单的身份验证例程。基本上让用户输入一个密码，如果该密码匹配于我们希望的值，就把 Canedit 设置为 true，该用户就可以编辑页面。

试试看：用户的身份验证

(1) 使用 Solution Explorer，右击 Weblog 工程，选择 Add | Add Web Form，将其命名为 Login.aspx。

(2) 在显示窗体的设计器时，单击页面的背景，使用 Properties 属性窗口把 pageLayout 改为 FlowLayout。

(3) 这里不修改页面的外观，因为这是一个管理工具，读者现在应掌握了如何使页面更吸引人。所以此处只把一个 TextBox 控件和 Button 控件拖放到页面上。

(4) 选择 TextBox 控件，把它的 ID 属性改为 textPassword，TextMode 属性改为 Password。

(5) 选择 Button 控件，把它的 Text 属性改为 Login。

(6) 双击窗体页面的背景，打开 Page_Load()方法，添加如下代码：

```
private void Page_Load(object sender, System.EventArgs e)
{
    // posting back?
    if(IsPostBack == true)
    {
        // do we have the correct password?
        if(textPassword.Text.CompareTo("stringy") == 0)
        {
            // update the session...
            Session["canedit"] = true;
```

```
            // redirect...
            Response.Redirect("default.aspx");
        }
    }
}
```

运行工程，单击一个 Edit 链接，此时应返回起始页面。在 Address 栏中，把 Default.aspx 改为 Login.aspx，单击 Go 按钮，其结果如图 24-42 所示。

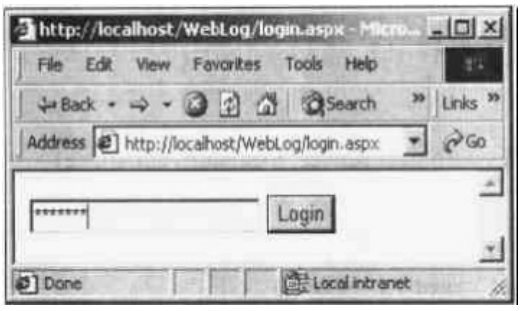


图　24-42

输入密码 stringy(注意大小写)。单击 Login 按钮，则应显示 Default.aspx。现在如果单击 Edit 链接，就能修改它。

代码的说明

只要加载 Edit.aspx，代码就会进行检查，确保 canedit 会话变量为 true。如果不是，就重定向到一个错误页面。

在 Login.aspx 中，当单击 Login 按钮时，就调用 Page_Load()方法。此时，看看页面是否被传送回来。接着看看 textPassword 的 Text 属性，确定输入的密码是否为 stringy：

```
private void Page_Load(object sender, System.EventArgs e)
{
    // posting back?
    if(IsPostBack == true)
    {
        // do we have the correct password?
        if(textPassword.Text.CompareTo("stringy") == 0)
        {
```

如果是，就把 canedit 会话变量改为 true，把用户重定向到主页上：

```
            // update the session...
            Session["canedit"] = true;

            // redirect...
                    Response.Redirect("default.aspx");
        }
    }
}
```

只要单击一个链接，打开 Edit.aspx，就会发现 canedit 设置为 true，因此页面像往常那样显示。然后就可以修改页面了。

24.5　小结

本章介绍了 Active Server Pages.NET(ASP.NET)。这个技术是 Microsoft 继成功的 ASP 技术之后推出的下一代产品，而 ASP 技术是在 1996 年推出的。

首先本章介绍了 ASP.NET 中的控件/事件模型是如何工作的，这非常类似于 Windows Forms 的工作方式。实际上，对于初学者来说，使用 ASP.NET 建立活动 Web 页是相当简单的。但是，坚毅的 Web 开发人员会发现转换有一些困难，因为其规则是完全不同的。

为了说明开发 ASP.NET 所使用的一些技术，我们创建了一个 Web 站点，在该 Web 站点上，可以查看 Web 日志上的项目。这些项目存储为 XML 文件，可以使用 XmlSerializer 读写这些项目。XmlSerializer 是一个非常有用的.NET Framework 类。我们在 Global 类上创建静态方法，其中 Golbal 类可以用于站点上的所有页面，所创建的这些静态方法可以从磁盘上获取一个 Entry 对象或所有 Entry 对象的一个数组。我们使用 DataList 控件和其他内置的 ASP.NET 控件来处理 Entry 对象列表。

最后，说明了如何创建一个窗体，以便让用户直接在 Web 站点上创建和编辑新 Entry 对象。我们还介绍了 Session 集合，学习了一些会话状态。

24.6　练习

(1) Asp.NET 的基本原则是什么？

(2) 在 Visual Studio.NET 中，可以把多少个文件关联到一个.aspx 文件上？

(3) DataList 控件可以做什么？

(4) 什么是样式表？

(5) 本章中 XML 文件的文件名格式有什么特殊的地方？

第25章 Web 服 务

读者以前可能遇到过“Web 服务”这个术语，尽管您可能不知道它们的含义，不知道目前或将来如何把它们用于 Web 操作。Web 服务奠定了下一代 Web 应用程序的基础。无论客户应用程序是 Windows 应用程序，还是 ASP.NET Web Forms 应用程序，无论客户程序运行在 Windows，Pocket Windows 或其他设备上，它们都会通过 Internet 使用 Web 服务定期通信。Web 服务类似于第 21 章介绍的组件，但它们运行在 Internet 上。它们是服务器端的程序，用以监听来自客户应用程序的信息，并传送回特定的信息。这些信息可能来自于 Web 服务本身，同一个域中的其他组件，或其他 Web 服务。尽管在编写本书的过程中，整个 Web 服务的概念在演变，但有几种不同类型的 Web 服务执行不同的功能：一些服务给某个行业(例如制造业和保健业)提供特定的信息，一些服务使用不同提供商提供的服务为特定的主题提供信息；一些服务专门用于一个应用程序，建立可以由许多不同的应用程序使用的块服务。

Web 服务可以合并、共享、交换或插入不同销售商或开发商提供的几个服务，形成新服务或随时定制应用程序，以满足客户的需要。

本章主要介绍：

- Web 服务的前任
- 什么是 Web 服务
- Web 服务使用的协议
- 创建一个 ASP.NET Web 服务
- 测试 Web 服务
- 建立一个使用 Web 服务的客户程序

我们并没有深入到 Web 服务的内部工作中，特别是基于 XML 的 SOAP 和 WSDL 格式，但概述了这些协议的使用场合。在阅读本章后，读者就可以使用 Visual Studio .NET 来创建和使用简单的 Web 服务了。

25.1 Web 服务推出之前

如第 23 章所述，把计算机连接起来传输数据在 1969 年就已经是一个非常重要的概念，那时只有 4 台计算机通过电话线连接起来，形成了 ARPANET。1976 年，发明了 TCP/IP 协议。为了使这个协议易于使用，伯克利大学创建了用于网络编程的套接字模型。

在使用 Sockets API 编程时，客户机必须启动一个到服务器的连接，再发送和接收数据。要在服务器上执行某些操作，得到结果，就需要使用其他协议来描述请求和响应代码。这些所谓的应用程序协议有 FTP、Telnet 和 HTTP。FTP 协议用于从服务器处获得某些文件，并把文件放在服务器上。FTP 协议支持请求代码，例如从客户机发送到服务器的 GET 和 PUT。服务器

分析接收到的数据流，根据请求代码调用相应的方法。第 23 章介绍的 HTTP 协议的工作方式非常类似于 FTP 协议的工作方式。

25.1.1 远程过程调用 (RPC)

用 Sockets API 和 TCP/IP 协议调用服务器上的定制方法，程序员必须创建一种方式，让服务器分析数据流，以调用相应的方法。为了使这个过程更容易实现，一些公司创建了 RPC (远程过程调用，Remote Procedure Calls)协议，例如 Open Software Foundation 公司(OSF)创建的、仍很流行的 DCE-RPC (Distributed Computing Environment – Remote Procedure Calls)协议，该公司后来变成 Open Group (参阅www.opengroup.org站点上的信息)。使用 RPC 可以用 IDL (接口定义语言，Interface Definition Language)格式定义方法，服务器必须执行该方法，客户机也可以调用它。现在不再需要处理定制协议的定义，分析请求代码，以调用方法了，这些工作都由一个特定的程序来完成，该程序称为 stub，由一个接口编译器生成。

RPC 用于调用方法，这表示我们必须进行过程化编程。RPC 技术推出得较晚，那时大多数开发商已经开始使用面向对象的方式编程了。为了在这些技术之间搭起桥梁，又开发出了几个技术，包括 CORBA 和 DCOM。

1. CORBA

Object Management Group (OMG，www.omg.org) 在 1991 年开发出了 CORBA (公用对象请求代理程序体系结构，Common Object Request Broker Architecture)，把面向对象技术添加到网络编程中。许多销售商如 Digital Equipment、HP、IBM 和其他销售商都提供 CORBA 服务器。但因为 OMG 没有定义引用执行代码，只有一个规范，所以这些销售商的服务器都不能交互操作。HP 服务器需要一个 HP 客户机，IBM 服务器需要一个 IBM 客户机，等等。

2. DCOM

在 Windows NT 4 中，Microsoft 用面向对象的功能扩展了 DCE-RPC 协议。DCOM (Distributed COM)可以在网络上调用 COM 组件，并用于 COM+应用程序。几年后，Microsoft 操作系统需要使用 DCOM，Microsoft 就对其他带有 The Active Group 的操作系统开放了该协议。DCOM 可以用于 Unix、VMS 和 IBM 系统，但主要用于 Microsoft 环境，把它用于其他系统所做的努力并不成功。哪个 IBM 大型机管理员愿意在自己的系统中加入一些 Microsoft 技术呢？

3. RMI

Sun 公司对自己的 Java 技术采用了不同的方式。在一个纯粹的 Java 环境中，RMI (远程方法调用，Remote Method Invocation)协议可以用于远程调用对象。Sun 公司添加了一些与 CORBA 和 COM 的桥梁技术，但它的主要目的是让人们使用只有 Java 的解决方案。

25.1.2 SOAP

前面介绍的所有技术都能用于应用程序之间的通信。但如果有 CORBA、DCOM 和 RMI 解决方案，就很难让这些不同的技术彼此通信。这些技术的另一个问题是它们的结构不是为上千个客户想获得 Internet 解决方案所需要的可伸缩性而设计的。

于是几家公司包括 Microsoft 和 Userland Software (www.userland.com)在 1999 年创建了 SOAP (简单对象访问协议，Simple Object Access Protocol)，把它作为通过 Internet 调用对象的全新方式，它建立在已被广泛接受的标准协议的基础之上。SOAP 使用基于 XML 的格式描述方法和参数，在网络上进行远程调用。COM 世界中的 SOAP 服务器可以给 COM 调用传送 SOAP 信息，而 CORBA 世界中的 SOAP 服务器可以给 CORBA 调用传送 SOAP 信息。最初 SOAP 定义使用 HTTP 协议，所以 SOAP 调用可以通过 Internet 实现。

在 SOAP 中，产生了 Web 服务这个术语。

注意：

在 W3C 规范中， SOAP 1.1 在 www.w3.org/TR/SOAP/上。

目前，从广义上看，提供服务的每个应用程序都是一个 Web 服务。Web 服务不一定要使用 SOAP 协议作为信息的格式，用 HTTP 协议在客户机和服务器之间通信，也可以使用其他协议。在 SOAP 规范 1.0 中，需要使用 HTTP 协议，但 1.1 版本就有了改变，可以使用其他传输协议。但要在 Internet 上使用 Web 服务，SOAP 和 HTTP 是主要协议。使用.NET Remoting，可以创建使用二进制信息格式和 TCP 连接的 Web 服务。

本章主要介绍可以用 Visual Studio.NET Project Wizard 创建的 Web 服务，该向导本身可以在后台使用.NET Remoting、SOAP 和 HTTP。

25.2 使用 Web 服务的场合

为了从另一个角度来了解 Web 服务，需要区分用户-应用程序通信和应用程序－应用程序通信之间的区别。

首先看看用户-应用程序通信，从 Web 上获得某些天气信息。有许多网站如 http://www.msnbc.com 和 http://www.weather.com 都提供了天气信息，其格式用户都很容易理解。用户可以直接阅读这些页面。

如果要创建一个客户应用程序来显示天气(应用程序-应用程序通信)，该应用程序必须连接到一个 Web 站点上，其 URL 字符串包含某个城市，我们想知道该城市的天气。接着要分析从 Web 站点返回的 HTML 信息，得到气温和天气信息，最后以合适的格式给客户应用程序显示这些信息。

其工作量很大，而我们只想得到某个城市的气温信息。而且，从 HTML 中获得数据的过程并不是很简单的。这是因为 HTML 数据主要用于显示在 Web 浏览器上，而不是由其他客户端商务应用程序使用。因此，数据内嵌在文本中，并不容易提取，需要重新编写或修改客户应用程序，从同一个 Web 页上提取不同的数据信息(例如下雨)。而使用 Web 浏览器，用户可以立即得到他们需要的数据，忽略不需要的内容。

要解决处理 HTML 数据的问题，Web 服务提供了一种有效的方式，可以只返回请求的数据。只要调用远程服务器上的一个方法，获得需要的信息，客户应用程序就可以直接使用这些信息了。我们不必处理预先格式化对用户界面有意义的文本，因为 Web 服务以 XML 格式显示信息，处理 XML 数据的工具早就有了。客户应用程序只需调用.NET Framework XML 类的一些方法，就可以得到需要的数据。更妙的是，如果用 C#编写.NET Web 服务的客户应用程序，甚至不需

要编写执行这些任务的代码，有一些可以生成 C#代码的工具！

刚才讨论的天气应用程序是使用 Web 服务的一个范例，Web 服务还有许多其他的用途。

25.2.1 宾馆旅行社代理应用程序

如何预订假日宾馆？到一个旅行社代理处去，它可以为您完成所有的工作。您已经在 Internet 上预订了假日宾馆？在某条航线的 Web 站点上，可以查找可能的航线，并预订它们。可以使用 Web 搜索引擎查找某个城市的宾馆。也许您很幸运，找到了去宾馆的地图。在找到该宾馆的主页后，就可以导航到预订窗体页面，在该页面上可以提前预订房间。接着可以搜索一个汽车租赁公司...

现在要做的是利用搜索引擎查找 Web 站点，找到去这些站点的路径。另外，可以创建一个家庭旅行代理应用程序，它使用包含宾馆、航线和汽车租赁公司等信息的 Web 服务，给客户提供了一种方便的方式来处理所有的假日旅行问题，包括提前预订某场音乐会。在假日里使用 Pocket PC，就可以使用相同的 Web 服务得到旅行用的地图，电影院的某些信息等。

25.2.2 书籍发布应用程序

Web 服务还可以用于两个有一些合作的公司。假定书籍出版商要得到书店销售书籍的信息，就可以用一个 Web 服务来实现。可以建立一个使用该 Web 服务的 ASP.NET 应用程序，直接给用户提供这个服务。这个服务的另一个客户应用程序是书店的 Windows 应用程序，首先检查本地书店，再检查分销商的书店。销售员可以立即答出交货日期，无需检查不同应用程序中的不同书店。

25.2.3 客户应用程序的类型

Web 服务的客户应用程序可以是使用 Windows Forms 创建的 Windows 应用程序，也可以是使用 Web Forms 创建的 ASP.NET 应用程序，Windows PC、Unix 系统和 Pocket PC 都可以使用该 Web 服务。在.NET Framework 中，Web 服务可以用在各种应用程序类型中：Windows Forms、Web Forms 或控制台应用程序。

25.2.4 应用程序的体系结构

使用 Web 服务的应用程序究竟是什么样的？无论是开发 ASP.NET 或 Windows 应用程序，还是开发小设备的应用程序，从前面介绍的应用程序可以看出，Web 服务在这些类型的应用程序中都是非常重要的技术。

图 25-1 说明了使用 Web 服务的方式。设备和浏览器通过 Internet 连接到一个用 Web Form 开发的 ASP.NET 应用程序上，这个 ASP.NET 应用程序使用某些本地 Web 服务，以及可以通过网络获得的其他远程 Web 服务：Portal Web 服务、用于某个应用程序的 Web 服务和 Building Block Web 服务。下面的列表详细描述了这些服务类型的含义：

- Portal Web 服务为同一个主题提供不同公司的服务。这将易于使用多个服务的入口点。
- 用于某个应用程序的 Web 服务：仅用于某个应用程序。
- Building Block Web 服务：可以在多个应用程序中使用。Microsoft 的 Hailstorm 就提供了许多这样的服务，例如存储应用程序的配置或者用户的身份验证。

图 25-1 中的 Windows 应用程序可以直接使用 Web 服务，无需通过 ASP.NET 应用程序。

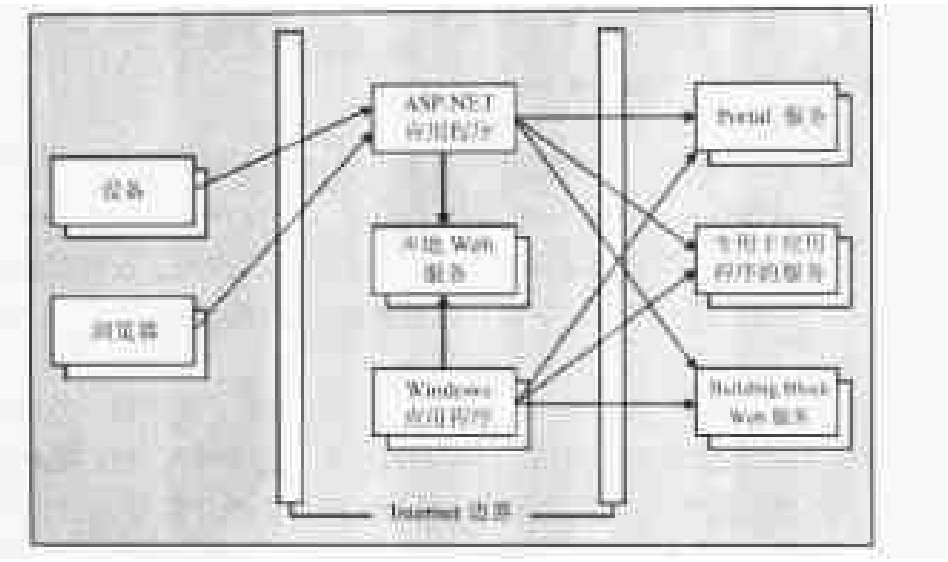


图　25-1

25.3　Web 服务的体系结构

如果要使用已有的 Web 服务，就必须找到一个满足我们需要的 Web 服务。如果知道有这样一个 Web 服务，就必须获得如何与它通信的信息。图 25-2 说明了调用 Web 服务的重要机制。

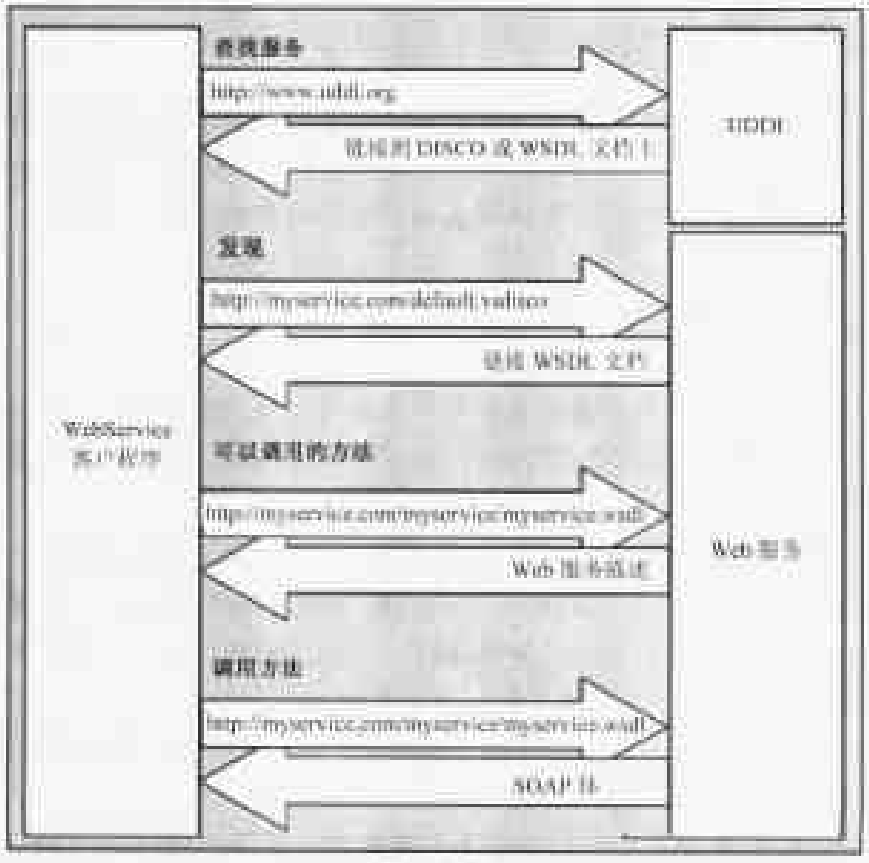


图　25-2

● 首先找到一个已经在注册目录服务中注册了的 Web 服务，这个目录服务返回该 Web 服务的绑定信息。

● 下一个序列称为发现。如果知道保存该 Web 服务的服务器，就可以请求该服务器传送该服务的描述信息。在这个序列中，要使用发现协议(DISCO)。

● 服务的描述信息以 Web Services Description Language (Web 服务描述语言，WSDL)格式显示。该描述信息描述了服务拥有的方法，可以传送的参数类型等。如果使用一个不支持发现、也没有在 UDDI 目录下注册的服务，就会得到一个 WSDL 文档。

● 有了 Web 服务的描述信息后，就知道可以调用什么方法。方法是使用 SOAP 调用的，所以所有包含参数的方法调用都必须转换为 SOAP 协议。

注意：

SOAP 和 WSDL 都是使用 XML 语法定义的。

下面详细介绍这些序列。

25.3.1　Web 服务的搜索引擎

也许可以使用其他公司已经支持的 Web 服务，为了找到已经存在的 Web 服务，Microsoft、IBM 和 Ariba 合作建立了一个带有 UDDI (Universal Description、Discovery and Integration)服务的网站 www.uddi.org。UDDI 是独立于平台的，开放式的架构，可描述服务、发现商务，使用 Web 集成商务服务，它还是一个可操作的注册表。如果公司要发布它的 Web 服务，就可以在 UDDI 中注册它。有了 UDDI 商务注册表和 UDDI API，就可以编程定位 Web 服务的信息了。

在这 3 家公司推出了 UDDI 后，现在已有超过 220 家公司支持 UDDI 工程。其中有 Boeing、BT、Compaq、DataChannel、Dell、Fujitsu、HP、Hitachi、KPMG Consulting、Merrill Lynch、Nortel Networks、Oracle、Rational、SAP、Sun Microsystems 和 VeriSign 等。

要找到某个已有的 Web 服务，可以按照商务名称进行搜索，或者使用下面的类别来浏览：行业分类、ISO 3166 地理分区或 Microsoft 的地理分区。在成功搜索到一个服务后，就可以得到这个 Web 服务的描述信息、分组的分类信息和绑定信息，它们包含在一个发现(DISCO)文件中，也许还是一个 WSDL 文档。

在 Visual Studio.NET 中，可以用菜单 Project | Add Web Reference 搜索 Microsoft 的 UDDI 站点，如图 25-3 所示。

在图中的 Microsoft UDDI Directory 旁边，还有一个 Test Microsoft UDDI Directory。在开发过程中，可以在测试目录中注册服务，另一个目录只能用于产品已经稳定的活动服务。

选择 Microsoft UDDI Directory，可以指定搜索字符串，查找注册了的 Web 服务。如果搜索的结果有绑定信息，就可以使用 Reference 按钮添加该 Web 服务的一个引用。

输入字符串 Continental，列出 Continental Airlines 的一个 Web 服务 Continental Flight Status，如图 25-4 所示。

注意：

在编写本书的过程中，并没有很多的 Web 服务注册到 Microsoft 的 UDDI 站点上，许多已注册的服务也没有功能链接。我希望在发布 Visual Studio. NET 时，这种情况会有所改变。

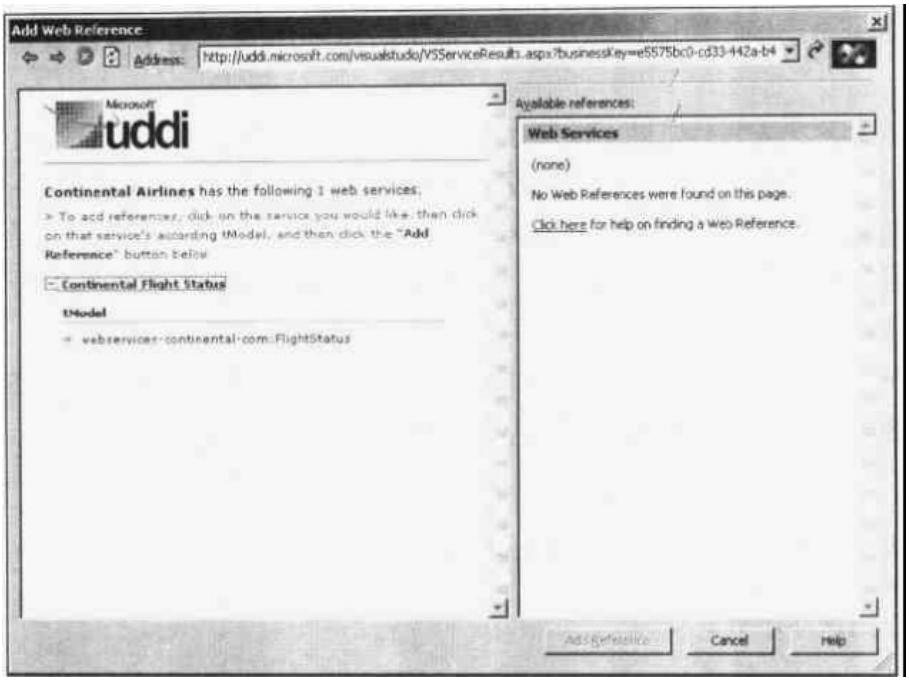


图　25-3

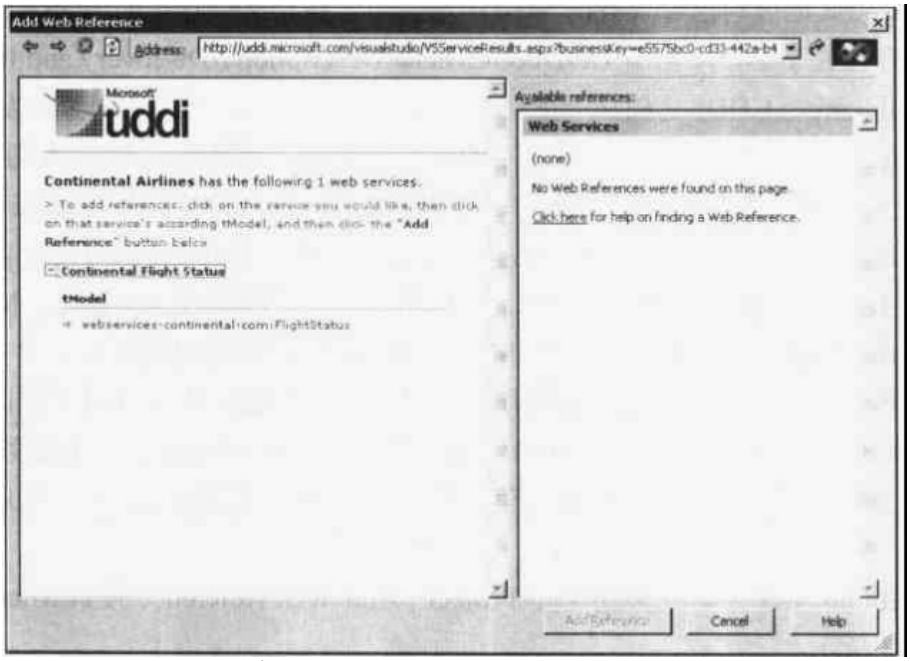


图　25-4

在 Web 页 www.xmethods.net 和 www.gotdotnet.com 中有许多 Web 服务。

25.3.2　Web 服务的发现

只要知道某个服务器支持 Web 服务，发现过程就可以用于获得该服务器上 Web 服务的信息。Web 服务器使用扩展名为 vsdisco 的文件返回 Web 服务的信息。

Vsdisco 文件定义的规则控制着服务器查找 Web 服务的方式。这个文件定义了服务是应通过排除某些目录进行动态查找，还是在某些目录中进行静态查找。

用浏览器请求文件 default.vsdisco，就将用 discoveryRef 元素返回其他 vsdisco 文件的链接。一个<discoveryRef>元素将显示对 http://localhost/WebServiceSample/WebServiceSample.vsdisco 的引用。我使用这个链接获得这个服务的更多信息，如图 25-5 所示。

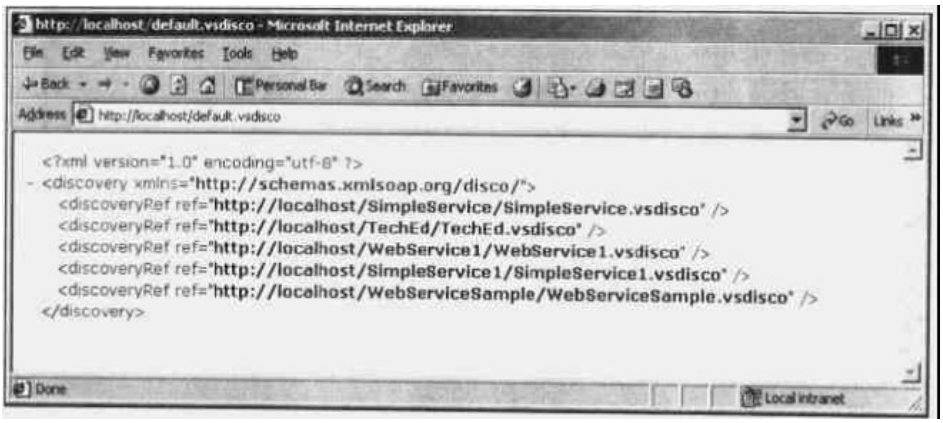


图 25-5

在浏览器的地址栏中输入地址 http://localhost/WebServiceSample/WebServiceSamplevsdisco，会用<contractRef>元素返回服务的 WSDL 信息的一个链接：http://localhost/ WebServiceSample/ Service1.asmx?wsdl，如图 25-6 所示。

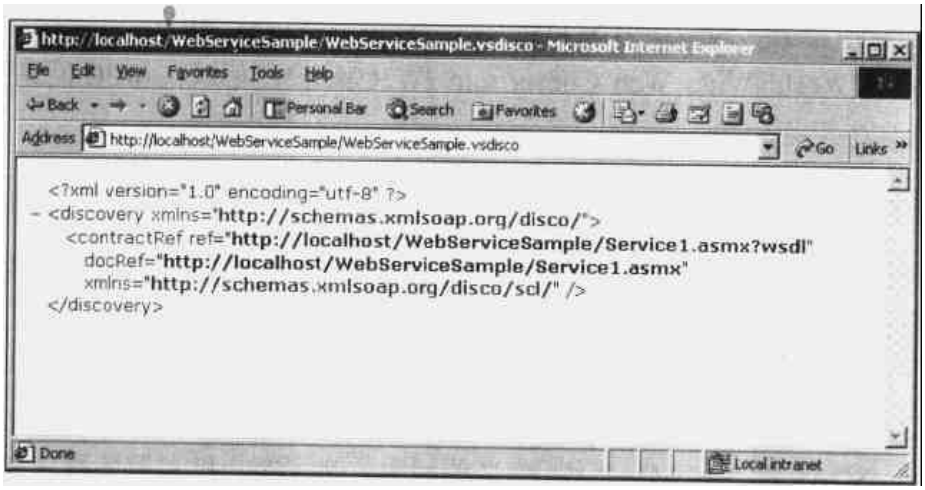


图 25-6

当然，不一定要使用浏览器阅读发现信息，可以直接通过 Visual Studio .NET 工具来来使用发现信息：在 Add Web Reference 对话框中，有一个 Web References on Local Web Server 的超链接，其中本地服务器的发现信息用于查找 Web 服务。也可以在 Address 编辑框中输入 Web 服务的地址。

25.3.3 可以调用的方法

Web 服描述语言(WSDL) 文档包含了下述信息：Web 服务支持什么方法，如何调用这些方法；给服务传送的参数类型，以及从服务返回的参数类型。

通过在发现过程中利用<contractRef>元素得到的链接，可以在浏览器中查看 WSDL 文档，如图 25-7 所示。

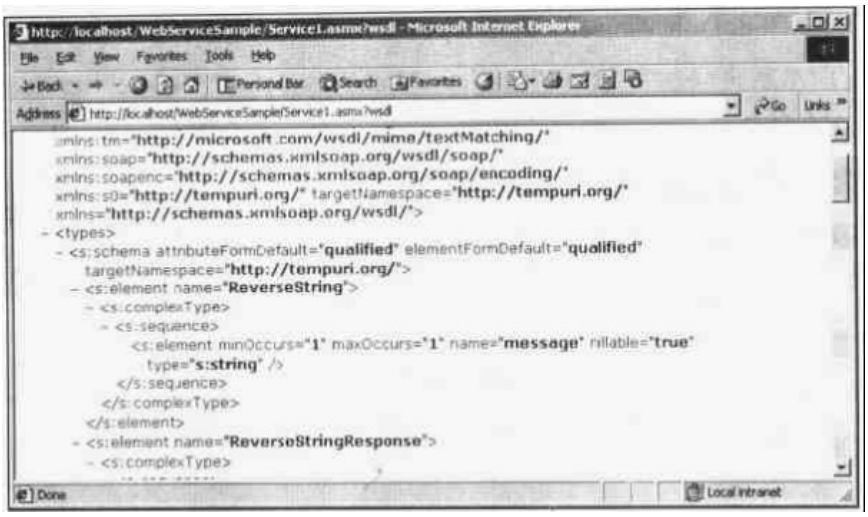


图　25-7

不一定要直接处理这些信息。WSDL 文档是用 WebMethod 属性动态生成的；本章的后面会介绍这个属性。在客户端，使用 Visual Studio.NET 添加 Web Reference 不仅可以得到服务器支持的那些 Web 服务的信息，还可以请求 WSDL 文档。这个 WSDL 文档用于创建带有相同方法和参数的客户代理程序，而利用这个代理程序，客户应用程序就只需在服务器中执行时调用方法，因为代理程序会把它们转换为 SOAP 调用，通过网络进行调用。

WSDL 规范由 World Wide Web Consortium (W3C)维护，可以在 W3C 网站 www. w3.org/TR/wsdl 上阅读这个规范。

25.3.4　调用方法

要调用 Web 服务上的一个方法，该调用必须转换为 SOAP 信息，因为它是在 WSDL 文档中定义的。SOAP 规范定义了如何传送方法名和参数。

SOAP 信息是客户机和服务器之间通信的基本单元。图 25-8 显示了 SOAP 信息的一部分。SOAP 信息包含一个 SOAP 封套，该封套把所有的 SOAP 信息封装在一个块中。SOAP 封套本章由两部分组成：SOAP 标题和 SOAP 体。可选的标题定义了客户机和服务器应如何处理 SOAP 体。必须有 SOAP 体包括要调用的方法块。在 SOAP 体中，客户机发送方法调用、以及所有方法参数的串行化值。SOAP 服务器把返回值发送回 SOAP 信息体。

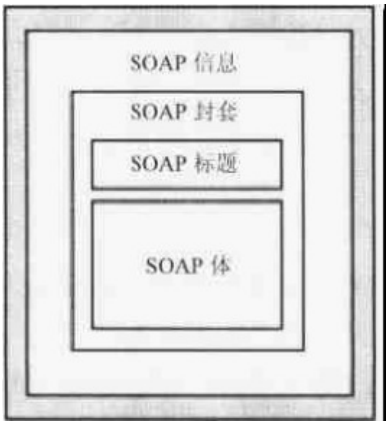


图　25-8

在下面的范例中，说明了 SOAP 信息是如何从客户机发送给服务器的。客户机调用 Web 服务方法 ReverseString()。字符串 Hello World!作为这个方法的参数而被传送。该方法调用在 SOAP 体中，位于 XML 元素<soap:Body>中。信息体本身包含在封套<soap:Envelope>中。在 SOAP 信息的开头，有 HTTP 标题，因为 SOAP 信息用一个 HTTP POST 请求来发送。

不一定要创建这样一个信息，因为它可以由客户代理程序创建的：

```
POST /WebServiceSample/Service1.asmx HTTP/1.1
Host: localhost
Content-Type: text/xml; charset=utf-8
Content-Length: 508
SOAPAction: "http://www.wrox.com/webservice/ReverseString"

<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <ReverseString xmlns="http://www.wrox.com/webservice">
      <message>Hello World!</message>
    </ReverseString>
  </soap:Body>
</soap:Envelope>
```

服务器在响应调用时，返回一个 SOAP 信息，其中包含结果!dlroW olleH；用 XML 元素 ReverseStringResult 就可以查看它：

```
HTTP/1.1 200 OK
Content-Type: text/xml; charset=utf-8
Content-Length: 446

<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <ReverseStringResponse xmlns="http://www.wrox.com/webservice">
      <ReverseStringResult>!dlroW olleH</ReverseStringResult>
    </ReverseStringResponse>
  </soap:Body>
</soap:Envelope>
```

SOAP 规范由 W3C 的 XML Protocol Working Group 维护(参见 www.w3.org/TR/soap 和 www.w3.org/TR/2001/WD-soap12-20010709 版本 1.2)。

SOAP 和防火墙

系统管理员常常询问 SOAP 协议是否超出了防火墙的安全界限，换言之，SOAP 是否违反了防火墙的规则？实际上，对于一般的 Web 服务并没有太多的安全问题。对于一般的 Web 服

务，防火墙的系统管理员会打开 HTTP 端口 80，允许服务器与外界通信。Internet 上的用户可以直接访问这些服务器，即使它们位于防火墙中，也是这样。用户可以用 HTTP 请求来请求一个 HTML 文件，服务器可以返回一个静态页面或用 ASP 或 CGI 脚本动态创建的一个页面。Web 服务仅是服务器端应用程序的另一种类型，它使用 HYYP 进行通信，但不是接收简单的 HTTP GET 或 POST 请求，而是接收一个包含内嵌 SOAP 信息的 HTTP POST 请求，它也不是返回 HTML，而是返回一个包含 SOAP 响应信息的 HTTP 响应。对于防火墙来说，通信是通过 HTTP 进行的，所以允许通过端口 80 来通信。

但是如果这个 Web 服务不像期望的那样执行，例如泄露了机密数据或中断了服务器，就出问题了，但这种问题对于所有的服务器端应用程序来说非常普通，无论它们是传统的 Web 页，还是服务器端的商务对象或 Web 服务。

如果防火墙的系统管理员仍在担心 Web 服务潜在的安全问题，可以使用应用程序过滤器，不允许通过 HTTP 请求进行 SOAP 调用。

25.4 Web 服务和.NET Framework

在.NET Framework 中，很容易创建和定制 Web 服务。下面列出的 4 个名称空间可以处理 Web 服务：

- System.Web.Services 名称空间中的类用于创建 Web 服务。
- 使用名称空间 System.Web.Services.Description，可以通过 WSDL 描述 Web 服务。
- 要使用 DISCO 协议查找 Web 服务，可以使用 System.Web.Services.Discovery 名称空间中的类。
- 使用 System.Web.Services.Protocols ，可以创建 SOAP 请求和响应。

注意：

本书不介绍 System.Web.Services.Description 和 System.Web.Services.Discovery 的用法。本章仅介绍其他两个名称空间。更多的信息可参阅 Wrox 出版的《ASP.NET Programmers Reference》(ISBN 186100530X)。

25.4.1 创建 Web 服务

对于 Web 服务的执行代码，可以从 System.Web.Services.WebService 中派生 Web 服务类。WebService 类提供了对 ASP.NET Application 和 Session 的访问，如表 25-1 所示。

表 25-1

WebService 属性	描　述
Application	获得当前请求的 HttpApplicationState 对象
Context	获得封装 HTTP 特定信息的 HttpContext 对象。在这里，可以读取 HTTP 标题信息
Server	获得 HttpServerUtility 对象。这个类有一些帮助方法，可以进行 URL 编码和解码

(续表)

WebService 属性	描 述
Session	获得 HttpSessionState 对象，以存储客户机的一些状态
User	获得执行接口 IPrincipal 的一个用户对象。使用这个接口可以得到用户名和身份验证类型

1. WebService 属性

The sub class of WebService 的子类应用属性 WebService 来标记。WebServiceAttribute 类有下述属性，如表 25-2 所示。

表 25-2

属 性	描 述
Description	服务的描述信息，可用于 WSDL 文档
Name	获取或设置 Web 服务名称
Namespace	获取或设置 Web 服务的 XML 名称空间。其默认值是 http://tempuri.org，它用于测试，但在开放这个服务前，应修改该名称空间

注意：

第 22 章介绍了属性，其中包含了更多的属性内容。

2. WebMethod 属性

Web 服务中可以使用的所有方法都必须用 WebMethod 属性来标记。当然，服务还可以有其他没有用 WebMethod 标记的方法。这些方法可以从 WebMethod 中调用，但不能在客户机上调用。使用属性类 WebMethodAttribute，方法就可以在远程客户机上调用，并可以定义是否缓存响应，缓存时间有多长，会话状态是否与指定的参数一起存储。表 25-3 列出了 WebMethodAttribute 类的属性。

表 25-3

属 性	描 述
BufferResponse	获取或设置响应是否应缓存的标志。默认值为 true。使用被缓存的响应，仅可以将值完成的软件包传递给客户机
CacheDuration	使用这个属性可以设置结果应缓存的时间长短。如果第二次发出了相同的请求，则在设置这个属性的时间段中，就只在发出的请求中返回缓存的值。默认值为 0，这表示结果不缓存
Description	该描述用于给预期的用户生成服务帮助页面
EnableSession	布尔值，表示会话状态是否有效。默认值是 false，因此 WebService 类的 Session 属性不能用于存储会话状态
MessageName	默认状态下，把信息名设置为方法名
TransactionOption	这个属性表示方法的事务处理支持。默认值是 Disabled

25.4.2 客户程序

要调用一个方法，客户机必须创建一个与 Web 服务所在的服务器之间的连接，并发送一个 HTTP 请求，传送一个 SOAP 信息。方法调用必须转换为 SOAP 信息。这些都是客户代理程序实现的。客户代理程序的执行代码在 SoapHttpClientProtocol 类中。

1. SoapHttpClientProtocol

类 System.Web.Services.Protocols.SoapHttpClientProtocol 是客户代理程序的基类。Invoke() 方法转换参数，建立一个 SOAP 信息，发送给 Web 服务。调用哪个 Web 服务由 Url 属性确定。

SoapHttpClientProtocol 类也支持使用 BeginInvoke() 和 EndInvoke()方法的异步调用。

2. 其他客户协议

如果不使用 SoapHttpClientProtocol 类，还可以使用其他代理程序类，HttpGetClientProtocol 和 HttpPostClientProtocol 仅执行简单的 HTTP GET 或 HTTP POST 请求，没有 SOAP 调用的系统开销。

注意：

如果解决方案在客户机和服务器上使用.NET，就可以使用 HttpGetClientProtocol 和 HttpPostClientProtocol 类。如果要使用不同的技术，就必须使用 SOAP 协议。

比较 HTTP POST 请求和本章前面介绍的 SOAP 调用：

```
POST /WebServiceSample/Service1.asmx/ReverseString HTTP/1.1
Host: localhost
Content-Type: application/x-www-form-urlencoded
Content-Length: length

message=string
```

HTTP GET 请求甚至更短，GET 请求的缺点是发送的参数长度是有限制的。如果其长度超过了 1K，就应考虑使用 POST：

```
GET /WebServiceSample/Service1.asmx/ReverseString?message=string HTTP/1.1
Host: localhost
```

有了 HttpGetClientProtocol 和 HttpPostClientProtocol，则其系统开销比 SOAP 方法小，但缺点是其他平台上的 Web 服务不支持它。

25.5 创建一个简单的 ASP.NET Web 服务

下面用 Visual Studio.NET 创建一个简单的 Web 服务。

试试看：创建一个 Web 服务工程

选择 File | New | Project，再选择 ASP.NET Web 服务模板，创建一个新工程，如图 25-9 所示。把工程名称设置为 WebServiceSample，单击 OK 按钮。

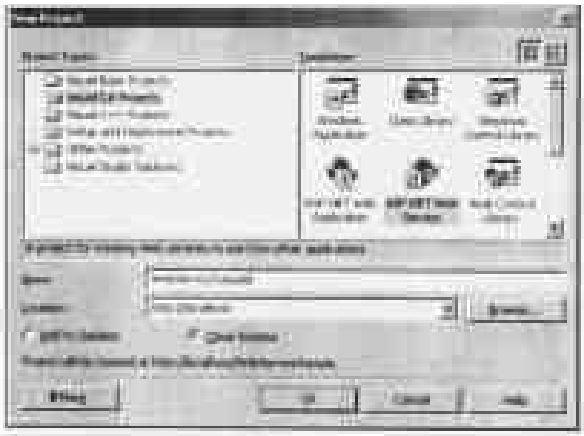


图 25-9

25.5.1　生成的文件

向导生成的文件如下所示：

- AssemblyInfo.cs 用于提供装配件属性，与其他工程类型一样。
- Global.asax：应用程序和会话开始和结束请求，与第 24 章的 Web Forms 应用程序相同。
- Service1.asmx 保存 Web 服务类。所有的 ASP.NET Web 服务都用.asmx 扩展名来标识。包含源代码的文件是 Service1.asmx.cs，因为后台编码功能可以在 Visual Studio .NET 中使用。但是在默认状态下，这个文件在 Solution Explorer 中是不可见的，要查看它，需要单击 Solution Explorer 工具栏中的 Show All Files 按钮。

向导生成了一个派生于 System.Web.Services.WebService 的类 Service1。在 Service1.asmx.cs 文件中，一些范例代码说明了 Web 服务的方法是如何编码的——它应是公共的，用 WebMethod 属性标记：

```
using System;
using System.Collections;
using System.ComponentModel;
using System.Data;
using System.Diagnostics;
using System.Web;
using System.Web.Services;

namespace WebServiceSample
{
    /// <summary>
    /// Summary description for Service1.
    /// </summary>
    public class Service1 : System.Web.Services.WebService
```

```
    {

        // WEB SERVICE EXAMPLE
        // The HelloWorld() example service returns the string
        // Hello World
        // To build、uncomment the following lines then save and build
        // the project
        // To test this web service、press F5

//      [WebMethod]
//      public string HelloWorld()
//      {
//          return "Hello World";
//      }
    }
}
```

- Web.config 是这个应用程序的配置文件。如第 24 章所述，利用这个文件，可以配置跟踪、会话状态和调试模式。
- WebServiceSample.vsdisco 是这个应用程序的发现文件。XML 元素<dynamicDiscovery>表示应在所有的目录中搜索 Web 服务，除了用<exclude>元素排除的目录外。

```
<?xml version="1.0" ?>
<dynamicDiscovery xmlns="urn:schemas-dynamicdiscovery:disco.2000-03-17">
<exclude path="_vti_cnf" />
<exclude path="_vti_pvt" />
<exclude path="_vti_log" />
<exclude path="_vti_script" />
<exclude path="_vti_txt" />
<exclude path="Web References" />
</dynamicDiscovery>
```

25.5.2　添加 Web 方法

接着，应给 Web 服务添加一个方法。这里添加一个简单的方法 ReverseString()，它接收一个字符串，颠倒该字符串中的字符顺序后，返回给客户机。

试试看：添加一个方法

(1) 在 Service1.asmx.cs 文件中添加下述代码：

```
[WebMethod]
public string ReverseString(string message)
{
    char[] arr = message.ToCharArray();
    Array.Reverse(arr);
```

```
        message = new string(arr);

        return message;
    }
```

要在为 Web 服务生成的描述信息中惟一地标识 XML 元素，应添加一个名称空间。在类 Service1 中添加 WebService 属性和名称空间 http://www.wrox.com/webservices。当然，可以使用其他字符串惟一地标识 XML 元素。可以使用公司页面的 URL 链接。Web 链接不一定存在；它仅用于惟一标识。如果使用基于公司网址的名称空间，那么就可以确保其他公司都不能使用这个名称空间。

如果没有输入名称空间，使用的默认名称空间是http://tempuri.org。从学习的角度来看，这个默认的名称空间就已足够了，但不应部署一个使用它的 Web 服务产品。

(2) 修改范例代码，如下所示：

```
[WebService(Namespace="http://www.wrox.com/webservices")]
public class Service1 : System.Web.Services.WebService
{
```

(3) 现在编译工程。

25.6　测试 Web 服务

现在就可以测试服务了。在浏览器中打开文件 Service1.asmx(可以选择 Debug | Start Without Debugging，在 Visual Studio .NET 中启动它)，列出服务中的所有方法，如图 25-10 所示。在该服务中，只有一个方法 ReverseString()。

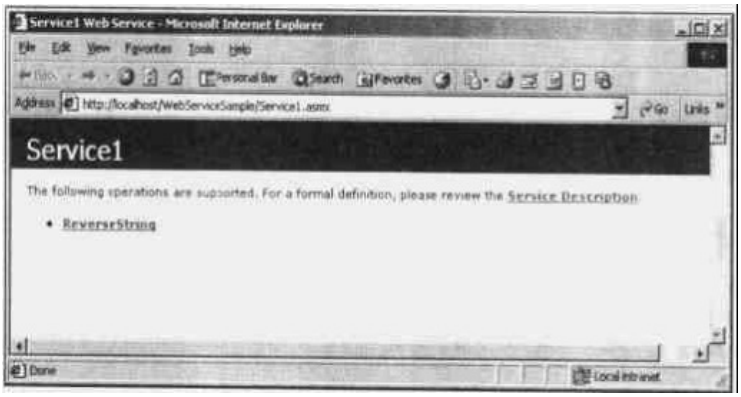


图　25-10

选择 ReverseString 方法的链接，显示一个对话框，以测试 Web 服务。该测试对话框已经编辑了传递给 ReverseString 方法的参数的字段，在本例中，该方法只有一个参数。

在这个页面中，还可以获得如何从客户机中进行 SOAP 调用的信息，以及服务器返回的响应信息。图 25-11 所示是一个带 SOAP、HTTP GET 和 HTTP POST 的范例。



图　25-11

再把字符串 Hello web services!写到文本框中后，单击 Invoke 按钮，服务器就会发送如图 25-12 结果。

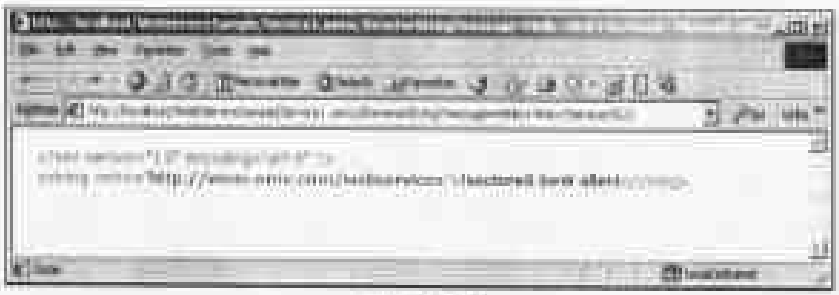


图　25-12

结果是 String 类型，与期望的一样，它的顺序与我们输入的字符串的顺序相反。

25.7　执行 Windows 客户程序

测试工作进行的很顺利，但我们希望创建一个使用 Web 服务的 Windows 客户程序。该客户程序必须创建一个 SOAP 信息，通过 HTTP 通道发送出去。这不一定要我们自己去完成。System.Web.Services.Protocols.SoapHttpClientProtocol 类会在后台完成所有的工作。

试试看：创建一个 Windows 客户程序

(1) 创建一个新的 C#工程，一个 Windows 应用程序，如图 25-13 所示。

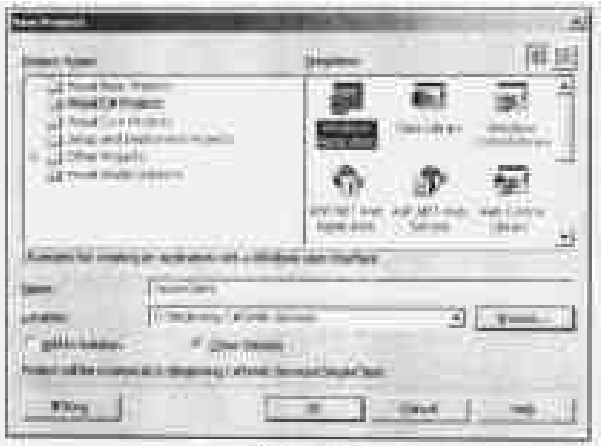


图 25-13

(2) 在窗体上添加两个文本框和一个按钮。利用按钮的单击处理程序，调用该 Web 服务，其结果如图 25-14 所示。

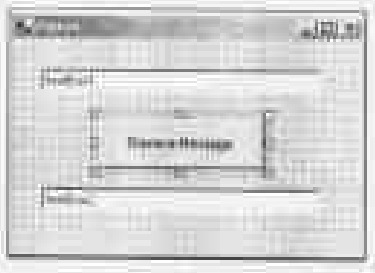


图 25-14

(3) 要添加一个 Web 引用，选择 Project | Add Web Reference 菜单，再选择超链接 Web References on Local Server，浏览到前面生成的 Web 服务上，接着查看合同和文档说明，之后单击 Add Reference 按钮。也可以直接输入 asmx 文件的 URL，而不是浏览服务，如图 25-15 所示。



图 25-15

代码的说明

在 Solution Explorer 中，可以看到 localhost 的一个新 Web 引用，以及已发送给客户机的相应的 DISCO 和 WSDL 文档，如图 25-16 所示。

在 Solution Explorer 中，看不到新创建的一个类，该类可以执行客户代理程序，把方法调用转换为 SOAP 格式。要查看这个类，必须切换到 Class View 窗口。此时将创建一个新名称空间，其中带有服务器的名称，在本例中是 localhost。类 Service1 派生于 System.Web.Services.Protocols.SoapHttpClientProtocol，执行 Web 服务的方法 ReverseString()，如图 25-17 所示。



图　25-16

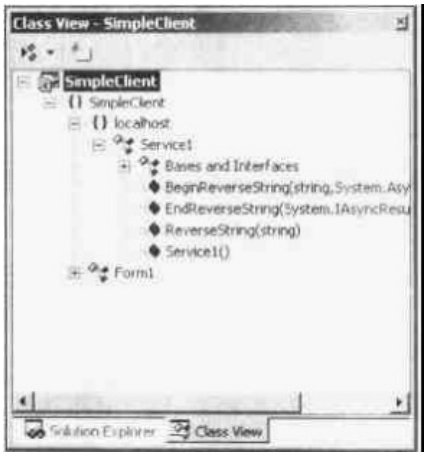


图　25-17

双击 ReverseString()方法，打开自动生成的 service1.cs 文件，下面看看这些向导生成的代码：

Service1 类派生于 SoapHttpClientProtocol 类，这个基类在 Invoke()方法中创建了一个 SOAP 信息。WebServiceBindingAttribute 属性把绑定值设置为 Web 服务：

```
//------------------------------------------------------------------------------
// <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>
//------------------------------------------------------------------------------

namespace SimpleClient.localhost {
    using System.Diagnostics;
    using System.Xml.Serialization;
    using System;
    using System.Web.Services.Protocols;
    using System.Web.Services;
```

```
[System.Web.Services.WebServiceBindingAttribute(Name="Service1Soap",
    Namespace="http://www.wrox.com/webservices")]
public class Service1 :
    System.Web.Services.Protocols.SoapHttpClientProtocol {
```

在构造函数中，属性 Url 设置为 Web 服务。这个属性将用于 SoapHttpClientProtocol 类，请求一个服务：

```
[System.Diagnostics.DebuggerStepThroughAttribute()]
public Service1() {
    this.Url = "http://localhost/WebServiceSample/Service1.asmx";
}
```

最重要的方法是 Web 服务提供的方法：ReverseString()。这里的方法与服务器上执行的方法有相同的参数。ReverseString()客户端版本的执行代码调用基类 SoapHttpClientProtocol 的 Invoke()方法。Invoke()使用方法名 ReverseString 和参数 message 创建一个 SOAP 信息：

```
[System.Diagnostics.DebuggerStepThroughAttribute()]
[System.Web.Services.Protocols.SoapDocumentMethodAttribute(
    http://www.wrox.com/webservices/ReverseString,
    RequestNamespace=http://www.wrox.com/webservices,
    ResponseNamespace="http://www.wrox.com/webservices",
    Use=System.Web.Services.Description.SoapBindingUse.Literal,
    ParameterStyle=
        System.Web.Services.Protocols.SoapParameterStyle.Wrapped)]
public string ReverseString(string message)
{
    object[] results = this.Invoke("ReverseString", new object[]
                    { message });
    return ((string)(results[0]));
}
```

客户程序也可以异步调用 Web 服务。方法 BeginReverseString()仅带有发送到服务器上的参数，EndReverseString()返回结果：

```
[System.Diagnostics.DebuggerStepThroughAttribute()]
public System.IAsyncResult BeginReverseString(string message,
                        System.AsyncCallback callback, object asyncState)
{
    return this.BeginInvoke("ReverseString", new object[] {
                    message}, callback, asyncState);
}

[System.Diagnostics.DebuggerStepThroughAttribute()]
public string EndReverseString(System.IAsyncResult asyncResult)
{
```

```
            object[] results = this.EndInvoke(asyncResult);
            return ((string)(results[0]));
        }
    }
}
```

到现在为止，我们没有给客户程序编写一行代码。我们设计了一个小用户界面，又用 Add Web Reference 菜单创建了一个代理类。现在必须创建它们之间的链接。

(4) 给按钮添加 Click 事件处理程序，添加如下两行代码：

```
private void button1_Click(object sender, System.EventArgs e)
{
    localhost.Service1 ws = new localhost.Service1();
    textBox2.Text = ws.ReverseString(textBox1.Text);
}
```

代码的说明

利用这行代码，创建代理类的一个新实例。在构造函数的执行代码中，Url 属性设置为 Web 服务：

```
localhost.Service1 ws = new localhost.Service1();
```

调用代理类的 ReverseString()方法，把 SOAP 信息传送给服务器，以调用 Web 服务：

```
textBox2.Text = ws.ReverseString(textBox1.Text);
```

运行程序，得到如图 25-18 所示的结果。

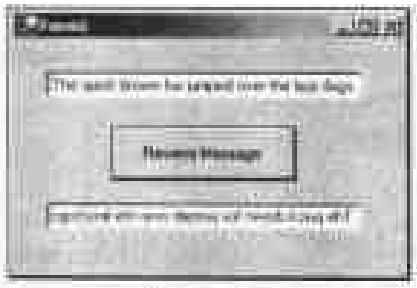


图 25-18

25.8 执行 ASP.NET 客户程序

现在可以在 ASP.NET 客户应用程序中使用相同的服务。引用 Web 服务的方式与下面所述的处理 Windows 应用程序的方式相同。

试试看：创建 ASP.NET 客户应用程序

(1) 创建一个新 C#工程，ASP.NET Web 应用程序，并命名为 ASPNETClient，如图 25-19 所示。

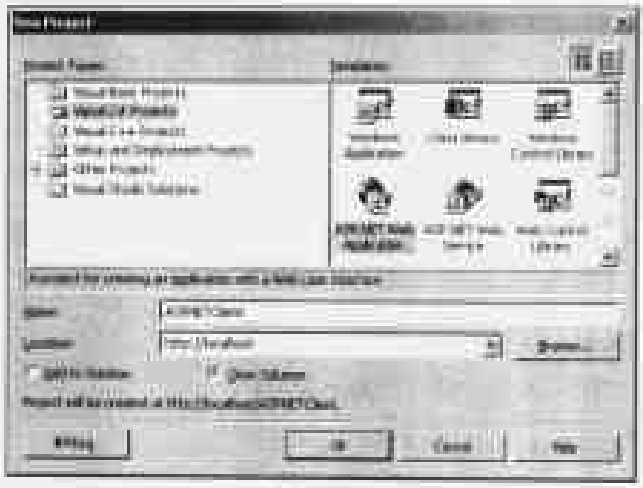


图 25-19

在 Web 窗体中添加两个文本框和一个按钮，如图 25-20 所示。利用按钮的单击处理程序，调用 Web 服务。

(2) 用前面处理 Windows 应用程序的方式，添加一个对 http://localhost/webservicesample/service1.asmx 的 Web 引用。

(3) 添加了 Web 引用后，再次生成一个客户代理类。给按钮添加一个单击处理程序，在该处理程序中输入如下代码：

```
private void Button1_Click(object sender、System.EventArgs e)
{
    ASPNETClient.localhost.Service1 ws =
        new ASPNETClient.localhost.Service1();
    TextBox2.Text = ws.ReverseString(TextBox1.Text);
}
```

(4) 建立该工程。

(5) 使用 Debug | Start，可以启动浏览器，在第一个文本框中输入一个测试信息。单击按钮，调用 Web 服务，在第二个文本框中得到逆序的信息，如图 25-21 所示。



图 25-20



图 25-21

可以看出，在 Web 应用程序中使用 Web 服务与在 Windows 应用程序中使用它一样简单。

25.9 小结

本章介绍了 Web 服务，并概述了 Web 服务使用的协议。要查找和运行 Web 服务，必须执行下述一个或所有的步骤：

(1) 目录 —— 利用 UDDI 查找 Web 服务。

(2) 发现 —— 在已知的服务器上查找 Web 服务

(3) 描述 —— WSDL 描述了方法和参数

(4) 调用 —— 利用 SOAP 协议调用独立于平台的方法

利用 Visual Studio.NET 创建 Web 服务是非常简单的，其中 WebService 类使用 WebMethod 属性定义一些方法。创建使用 Web 服务的客户程序也很简单，只需创建一个 Web 服务，给客户工程添加一个 Web 引用，再使用代理程序即可。客户程序的核心是 SoapHttpClientProtocol 类，它可以把方法调用转换为一个 SOAP 信息。

第26章　案例分析1——调查问卷Web站点

本书非常全面地介绍了.NET 技术，它们都可以用 C#代码来访问。本章将不再讨论理论，而是深入研究一个范例程序，它使用了.NET 中的许多技术。

这个案例先从一个基本应用程序开始，然后扩展它，创建带有其他功能的第二个版本。现实生活总是希望应用程序能满足一些新的需求，显然，把这个过程映射到书中一个案例分析上是非常有效的。这两个版本也可以告诉我们.NET 如何更便于扩展应用程序，这也意味着在设计和建立应用程序时，可以把注意力集中在应用程序的一些较小的部分。

该应用程序主要用于 Web 站点。但是读者在阅读本章的过程中，可以发现该应用程序可以用于各种类型的接口。尽管我们要构建的是一个 Web 站点，但其底层的对象模型也可以用于控制台应用程序或桌面应用程序，甚或一个 Web 服务，记住这一点是很重要的。另外，我们还能明白软件在其生存期中演化时，3 层应用程序模型(即数据层、事务逻辑层和表示层)可以起到什么作用。

26.1　调查问卷 Web 站点

许多 Web 站点都包含一个功能，可以把问题和一组可能的答案一起显示出来。Web 站点的访问者根据自己的观点回答问题，还可以查看此时所有选票的结果。下面是一个范例：

What development language do you primarily use?
C#
C++
Java
JavaScript
Visual Basic

访问者看到的结果如下所示：

10%　C#
2%　C++
34%　Java
12%　JavaScript
42%　Visual Basic

本章要建立的案例分析就是这样一个调查问卷系统，在建立该系统的两个版本时，还要添加一些功能。主要的界面是基于 Web 的，使用 ASP.NET 来建立，另外，我们还将用一个控制台应用程序、Windows 桌面应用程序和一个 Web 服务来开发表示层。

26.2　版本 1

在调查问卷 Web 站点的第一个版本中，创建一段简单的执行代码，在 Web 页上显示一个调查问卷，收集该调查问卷的选票。在开始之前，先介绍一下描述要求的一些语句。

26.2.1　要求

低通信量的 Web 站点必须有一个调查问卷，显示一个问题和许多可能的答案(通常少于 10 个)。在调查问卷中，站点访问者会选择其中的一个答案。选择之后，它们会查看调查问卷中每个答案的票数。

问题、答案和选票数的所有信息都存储在 Web 服务器的一个 XML 文件中，因为没有考虑使用一个数据库，所以其功能也不是很需要。如果管理员要修改调查问卷，就手动编辑 XML 调查问卷文件。应用程序必须输出简单的 HTML，该 HTML 会在两个主要浏览器(Netscape 和 Internet Explorer)中正确地显示出来，如图 26-1 所示。

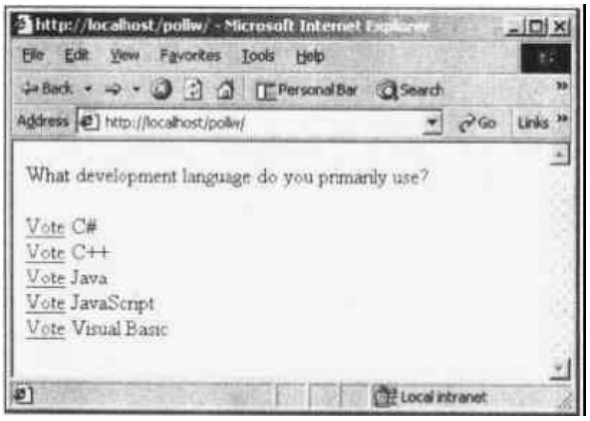


图　26-1

26.2.2　对象模型的设计(事务逻辑层)

对于这个案例分析，我们使用一个 3 层设计，在应用程序的 3 层之间有清晰的间隔：

- 表示层：这是用户界面，对于.NET，它可以包括 ASP.NET 应用程序(Internet、intranet、数字电视、移动电话和手提电话等)、控制台应用程序、Windows Forms 应用程序或 Web 服务。在这个案例分析中，我们使用 ASP.NET、Windows Forms、Web 服务和控制台命令行应用程序创建表示层。
- 事务逻辑层：这是应用程序的核心部分。在此将应用面向对象设计的一般规则，大多数代码一般也放在这里。
- 数据层：在该层可存储数据，它可以存在于 XML 文件中、磁盘上的无格式文件中、SQL Server、Microsoft Access、MSDE、Oracle 数据库中或其他存储数据的地方。许多应用程序还包括管理该层数据访问的类，以及存储介质本身，如图 26-2 所示。

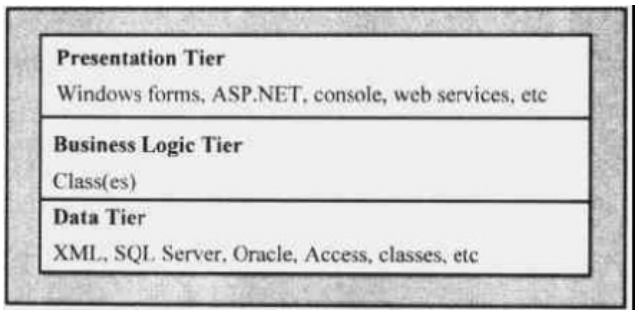


图　26-2

.NET 提供了许多功能强大而灵活的工具，除了这些工具外，我们还要使用一些方法开发能生成正确结果的应用程序。只使用.aspx 文件就可以建立这里讨论的调查问卷 Web 站点应用程序，所有的代码都放在一个地方，.aspx 文件直接与数据库连接。但是，这个案例分析使用了一种更结构化的方法，它可以分解为逻辑块，也很容易扩展。与维护软件相比，建立软件一般不需要很长的时间，这就是为什么使用结构化的方法非常有价值的原因。

对于 3 层应用程序，应谨慎利用需求来建立应用程序的对象模型图。这就是说，建立一个一致的模型，其中包括应用程序完成的主要任务，然后从中创建事务逻辑层。在事务逻辑层中，创建对象，建立以后要处理的问题域中的元素，以这种方式建立它们，就可以提供表示层需要的内容，更合理地使用数据层。通常在中间层设计对象时，必须考虑物理环境(例如性能)，但在本案例分析中，系统非常简单，可以不考虑物理环境，把注意力集中在这些事务逻辑对象上。这样就可以自由创建一个概念上非常清晰的对象模型，当需求改变时，还可以扩展该模型。

1. 案例分析中用到的类

通常利用直观的观察和一些经验来处理对象模型，但随着时间的推移，我们会总结出一个理论基础。首先再次扫描一下需求，把名词列出来，在应用程序中可能找到它们的描述类/对象：

- Web 站点
- 调查问卷
- 问题
- 回答
- 站点访问者
- 选票

看看这些名词，似乎最有用的名词集中在“调查问卷、问题、回答和选票”上。其他两个名词(Web 站点和站点访问者)不是收集选票的应用程序所特有的，但后面会需要它们。

Poll 类

在讨论一次调查问卷时，就是在讨论一些非常特殊的数据：

- 一个问题
- 多个答案
- 计算每个答案的选票

还要记住，在 XML 文件中可以有多次调查问卷，应能创建一个 Poll 对象，表示这些调查问卷。

现在数据需要存储在 Poll 对象中，下面考虑一下 Poll 对象需要提供什么功能：

- 选票——应包含一个方法 Vote()，它把一个答案的选票注册到一次调查问卷中。
- 获取问题——必须能访问调查问卷中的问题，所以把它显示在 Web 页上。
- 获取回答——同样，需要一种方式读取调查问卷的答案列表，所以在页面上显示它们，让访问者从这些答案中选择。
- 获得回答的选票——对于任何一个答案，还必须有一种方式计算出该答案收到多少张选票。我们使用它在访问者投票后，显示每个答案的选票数。

Poll 类包含许多需要的内容，下面看看其他元素能提供什么。

Question 类

什么是问题？从要求来看，问题仅是一个字符串，例如“您使用什么平台？”因为它非常简单，所以不需要把它执行为一个对象，把它当作另一个类上的字符串字段即可。我们将决定何时关注类之间的关系。

Answer 类

答案和问题一样，是一个字符串，但记住每次调查问卷只有一个问题，每个问题和每次调查问卷都有几个答案。我们还必须考虑每个答案还包含一个相关的选票数。如果把答案执行为一个类，就要包含答案文本的属性和选票数，包含一个 Vote()方法。下一节确定答案。

Vote 类

什么是选票？它是 Yes 或 No，没有其他的内容。或者把选票当作一个类，此时它可以包含每个答案的选票数。在这样的环境下，Votes 类不过是一个数组。如果是这样，建立一个实际上是数组的类就没有任何好处。也许以后可以找到建立这个类的理由，但现在不把它建立为一个类。

类之间的关系

显然可以假定 Poll 类是应用程序的核心组件。它包含一个问题，一组答案和选票数，并提供 Vote()方法来注册访问者的选票。

为问题创建一个类是没有意义的，因为它仅是一个字符串，所以把它放在另一个类中，作为一个字符串属性。每个 Poll 对象都包含一个问题，所以 Poll 类应包含一个字符串字段，以存储问题。

现在的问题是如何处理答案和选票数，因为这里可以把答案作为一个“Answer”类，其中包含一个字符串(答案文本)和一个选票数，也可以在 Poll 类中使用一个集合字段存储答案列表，再用一个平行的字段集合存储每个答案的选票数(换言之，选票字段集合的第二个元素给出问题已注册的选票数，该选票数用答案集合中的第二个元素来描述)。从对象模型的角度来看，其结果是注册一个选票，如下所示：

```
myAnswerText = myPoll.Answers[3];
myPoll.Vote(3);
```

这里在 Poll 类上使用 Vote()方法注册选票。我们还有一个属性 Answers，可访问字符串集合(也许是某种形式的数组)。

如果创建一个包含在 Poll 类中的 Answer 类，其代码应如下所示：

```
myAnswerText = myPoll.Answers[3].Text;
myPoll.Answers[3].Vote();
```

也可以写为：

```
myAnswer = myPoll.Answers[3];
myAnswerText = myAnswer.Text;
myAnswer.Vote();
```

上面两个范例的区别是集合现在是 Answer 对象的一个集合，每个 Answer 对象提供一个属性 Text(给出答案文本)和一个方法 Vote()。从使用的角度来看，前者比较简单，不需要建立 Answer 类。另一个好处是在以后扩展应用程序时，我们脑子中要记住的类比较少，应用程序的维护也比较简单。所以我们使用 Poll 类中的字段处理答案和选票数，而不是创建 Answer 类。实际上，如果以后要建立这个类，是完全可行的，但现在是要使程序简单且好用。

2. 对象模型

调查问卷应用程序的对象模型现在只包含一个类 Poll，该类如下所示：

- 方法 ——Vote()
- 属性 ——Question
- 属性 ——Answers[] (集合)
- 属性 ——Votes[] (集合)

假定调查问卷的答案存储在一个属性中，该属性返回一个可以迭代的字符串集合，选票数以类似的方式管理，并返回一个整数。

现在要有一个成功地存储调查问卷 Web 站点的对象，该如何创建一个这样的对象呢？我们需要一种创建 Poll 对象的方式，该对象表示当前进行的调查问卷。为此，可使用 Poll 类上的一个静态方法 Current()，于是可以使用该方法，如下所示：

```
CurrentlyActivePoll = Poll.Current();
```

得到如下所示的更坚固的对象模型：

- 方法 ——Vote()
- 方法 ——Current() (静态)
- 属性 ——Question
- 属性 ——Answers[] (集合)
- 属性 ——Votes[] (集合)

3. Poll 类的方法和属性

下面详细讨论每个方法和属性的签名，说明执行代码是如何工作的。在下一节中，介绍如何在一般环境下使用 Poll 类。

Vote()方法

Vote()方法带一个整数，其值为 1 到 n，n 表示调查问卷中答案的个数。它没有返回值：

```
public void Vote(int answer)
```

Current()静态方法

Current()方法是静态的，可以使用对类名的一个引用来访问它，而不是实例名。它不带参数，返回类 Poll 的一个对象，表示当前进行的调查问卷：

```
public static Poll Current()
```

Question 属性

Question 属性返回一个字符串，该字符串用于显示 Web 站点访问者要回答的问题，例如“您使用什么平台？”

```
public string Question
```

在第一个版本中，我们使用 XML 文件编辑问题，这表示 Question 属性是只读的，只需执行 get 子句，不需要执行 set 子句。

Answers 属性

Answers 属性可以访问一个存储了调查问卷所有答案的集合对象，对于上述的操作系统调查问卷，其答案可以是“Windows 2000”，“Linux”，“MacOS”等。因为我们知道集合的大小且知道每个元素都是相同的类型，所以可以使用内在的数组类型，把答案存储在一个字符串数组中：

```
public string[] Answers
```

与 Question 属性一样，数据是通过修改 XML 文件来手动修改的，所以只需执行 get 子句，这是一个只读属性。

Votes 属性

Votes 属性包含每个调查问卷答案的整数值。该属性返回一个整数数组，它与 Answers 数组类似。在执行代码必须维护的两个数组之间是一对一的映射关系：

```
public int[] Votes
```

存储在 Votes 属性中的值只能由 Vote()方法修改，所以这个属性与 Question 和 Answers 一样，也是只读的，只执行 get 子句。

注意在像这样的类中，有两个必须彼此同步的数组，所以必须确保它们是同步的。在商业环境中，创建一个 Answer 类来存储答案文本和选票数可以更好地把这两个数组关联起来，或者使用其他解决方案。从简洁性来看，我们继续使用最适合我们的两个平行数组。只要在用代码修改集合中的值时仔细一些即可。

4. 一般用法

假定有一个对象模型。如何使用它？在了解方法和属性如何用于解决问题时，常常会发

现设计中的漏洞，特别是模型的工作方式在概念上比较混乱(这称为“用法问题”，在面向对象的编程中是一个重要的元素)。下面介绍一个简单的控制台应用程序，它执行当前的调查问卷，显示它并注册一个选票。从这里可以看出模型在其概念上是否有效：

```
Poll myPoll = Poll.Current();
Console.WriteLine(myPoll.Question);
for(int i=1;i<myPoll.Answers.Length;i++)
{
    Console.Write(myPoll.Answers[i] + ": ");
    Console.WriteLine(myPoll.Votes[i] + " votes");
}
myPoll.Vote(2);
Console.WriteLine("Vote registered against " + myPoll.Answers[2]);
Console.WriteLine(Poll.Votes[2]);
```

OK，这看起来不错；我们可以获取当前的调查问卷，显示它，并注册选票。从这里可以看出，我们使用 myPoll.Answers.Length 确定有多少个答案。这似乎相当混乱，因为不清楚在 Votes 或 Answers 上是否使用了 Length 属性，为了避免这种混乱，添加一个属性 AnswerCount，它返回调查问卷中的答案个数：

```
public int AnswerCount      // Public property on the Poll class
```

这略微改变了使用 Poll 类的方式，提供了一个更清晰的界面：

```
Poll myPoll = Poll.Current();
Console.WriteLine(myPoll.Question);
for(int i=1;i<myPoll.AnswerCount;i++)
{
    Console.Write(myPoll.Answers[i] + ": ");
    Console.WriteLine(myPoll.Votes[i] + " votes");
}
myPoll.Vote(2);
Console.WriteLine("Vote registered against " + myPoll.Answers[2]);
Console.WriteLine(Poll.Votes[2]);
```

26.2.3　测试

在继续之前，用一个小控制台应用程序测试一下 Poll 类。这么做有两个优点。首先，很容易看出 Poll 类是在建立时就起作用了。第二，可以看出如何在控制台应用程序上使用 Poll 类，即使建立它的目的主要是用于 ASP.NET 应用程序，也是如此。

首先在 Visual Studio.NET 中创建一个新的 C#控制台应用程序 PollC (Poll Command Line 的缩写)，如图 26-3 所示。

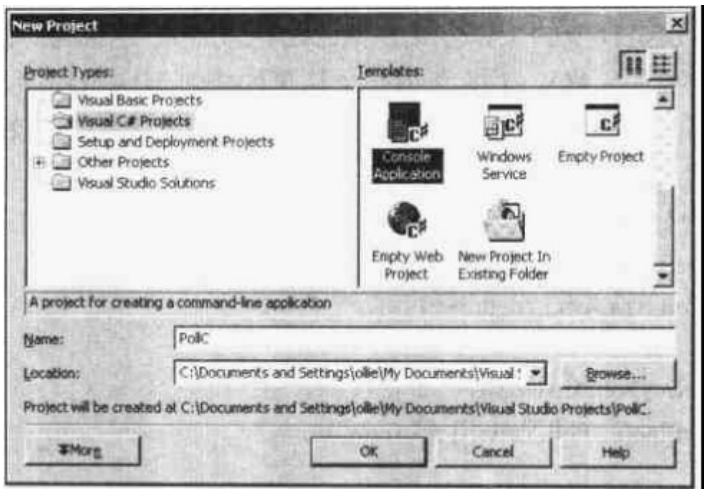


图　26-3

Visual Studio .NET 创建控制台应用程序工程，生成一个文件 Class1.cs，它包含静态方法 Main()，在运行控制台应用程序的.exe 文件时，执行该方法。

```
using System;
namespace PollC
{
    /// <summary>
    /// Summary description for Class1.
    /// </summary>
    class Class1
    {
        static void Main(string[] args)
        {
            //
            // TODO: Add code to start application here
            //
        }
    }
}
```

现在给获取当前进行调查问卷的控制台应用程序添加一些方法，显示它和选票数，注册选票，再显示调查问卷内容。为此，修改 Main()方法，如下所示：

```
static void Main(string[] args)
{
    Poll myPoll = Poll.Current();
    DisplayPoll(myPoll);
    myPoll.Vote(2);
    Console.WriteLine("Vote registered against " + myPoll.Answers[2]);
    DisplayPoll(myPoll);
}
```

这里的代码类似于前面使用 Poll 类的代码。首先，创建一个 Poll 对象 myPoll，接着使用一个 DisplayPoll()方法，把该对象显示在屏幕上(后面介绍这个方法)，之后使用 Poll 类的 Vote()方法注册一个选票，最后再次调用 DisplayPoll()，以显示调查问卷的内容。如果正确地完成了这些任务，就会增加第二个答案的选票数。

DisplayPoll()是一个简单的私有方法，它带一个 Poll 对象，在控制台输出中显示它。它将显示问题、每个答案，以及每个答案接收的选票数：

```
private static void DisplayPoll(Poll aPoll)
{
    Console.WriteLine("===Poll===");
    Console.WriteLine(aPoll.Question);
    for(int i=1;i<=(aPoll.AnswerCount);i++)
    {
        Console.Write(aPoll.Answers[i] + ": ");
        Console.WriteLine(aPoll.Votes[i] + " votes");
    }
    Console.WriteLine("----------");
}
```

DisplayPoll()使用 Question 属性显示问题文本，然后迭代 Answers 和 Votes 属性中的每个项目，显示答案文本和每个答案的选票数。

建立了 Poll 类后，控制台应用程序就应输出如图 26-4 所示的结果。

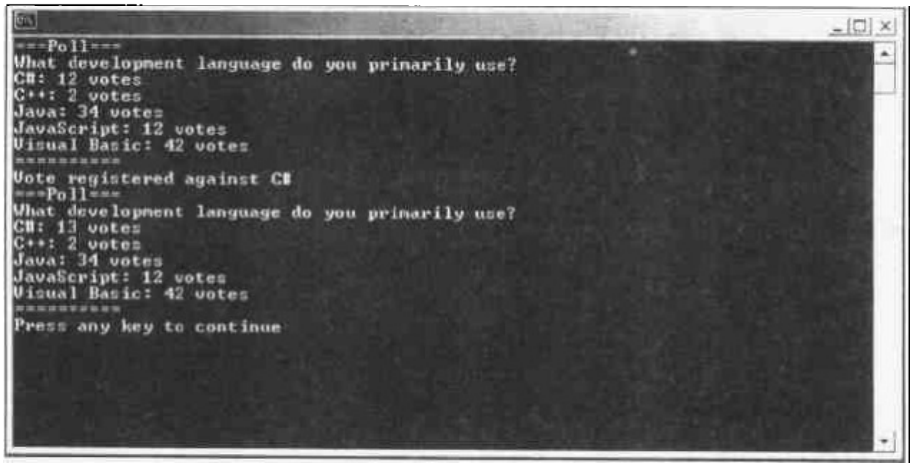


图　26-4

26.2.4　数据结构(数据层)

现在有一个测试工具，可以用于测试代码的运行情况，下面将深入探讨如何在 XML 文件中存储数据。注意，Poll 类和调查问卷数据之间是一对一的映射关系，界面(可以是控制台应用程序、Web Forms 或 Windows Forms)可以与事务逻辑(Poll 类)通信，因此很方便以后移动到不

同的数据存储结构中。以这种方式使用 XML 有许多限制，详见后面的内容。

每个调查问卷都有如下一组数据：

- 问题(字符串)
- 多个答案(字符串)
- 多个选票数(整数)

可以在 C:\驱动器的根目录下创建一个简单的 XML 文件 polls.xml，如下所示：

```
<?xml version="1.0" encoding="utf-8" ?>
<Poll Question="What development language do you primarily use?">
    <Answer Text="C#" Votes="10" />
    <Answer Text="C++" Votes="2" />
    <Answer Text="Java" Votes="34" />
    <Answer Text="JavaScript" Votes="12" />
    <Answer Text="Visual Basic" Votes="42" />
</Poll>
```

注意，如何用一个 Poll 标记存储整个调查问卷过程，并存储一个表示调查问卷问题的属性。在 Poll 标记中，有一系列 Answer 标记，每个标记都包含一个答案文本和一个选票数。

XML 格式非常简单，可以手动编辑它，但以 XML 格式存储数据，就可以使用各种选项把数据读取到事务对象上，并进行处理。XML 文件中为调查问卷答案提供的标记，实际上其结构与它们在数据库中的结构完全相同，一行表示一个答案，它有 Text 和 Votes 列。以后可以看到，通过数据集访问 XML 的方式与访问数据库表的方式完全相同。

26.2.5　对象的执行

记住，我们把一个类 Poll 放在了事务逻辑层中，下面简单描述一下需要编写的一组方法和属性。

1. 方法

- public static Poll Current()
- public void Vote(int answer)

2. 属性

- public int AnswerCount
- public string Question
- public string[] Answers
- public int[] Votes

下面把类组合在一起。首先为 Poll 类创建一个空白的模板文件。在 Visual Studio.NET 的 Project 菜单上，使用 Add Class 选项，键入文件名 Poll.cs，如图 26-5 所示。

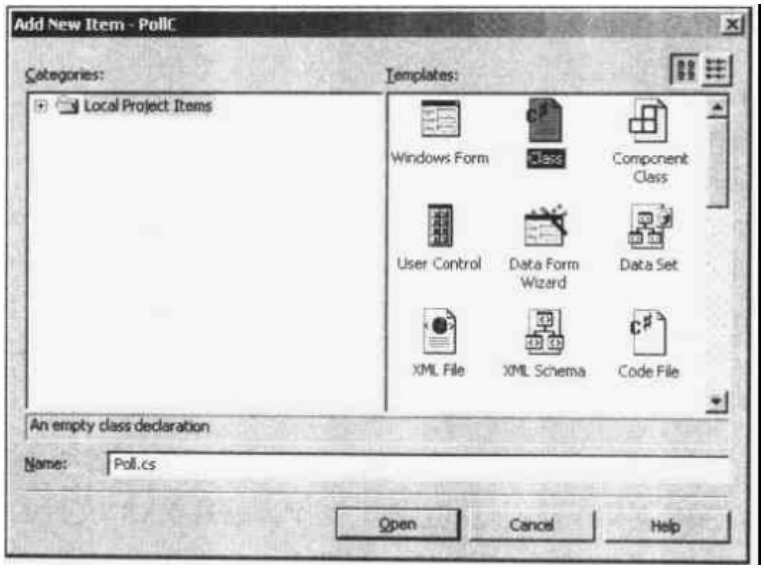


图　26-5

Visual Studio.NET 将创建 Poll.cs 文件，并生成一个启动程序类的轮廓，如下所示：

```
using System;

namespace Poll
{
    /// <summary>
    /// Summary description for Poll.
    /// </summary>
    public class Poll
    {
        public Poll()
        {
            //
            // TODO: Add constructor logic here
            //
        }
    }
}
```

注意该文件是如何添加空白的构造函数(Poll()方法)的，这不同于 PollC.cs 文件，该文件有一个 Main()方法，而不是构造函数。现在有了一个类文件，下面就介绍每个方法，并说明如何把它们组合起来。

在介绍 Poll 类中的方法和属性之前，必须定义一些字段。这里将简要介绍它们，在介绍每个方法时再详细阐述。它们存储选票、答案、问题和答案数(myAnswerCount)，以及存储调查问卷数据的 XML 文件的完整文件名：

```
namespace Polling
{
```

```
public class Poll
{
private int[] myVotes;
private string[] myAnswers;
private string myQuestion;
private int myAnswerCount;
private string myFilename;
```

还要注意的是我们要把名称空间改为 Polling。

3. 方法 —— public static Poll Current()

这个方法是惟一的静态方法，可用于返回 Poll 类的一个实例，该实例包含当前进行的调查问卷数据，类必须从 XML 文件中提取这些数据。这个方法从 XML 文件的<Question>标记中提取问题文本，并把这些文本存储在 myQuestion 属性中。它将迭代<Answer>标记列表，把 Text 值保存在 myAnswers 数组中，把 Votes 值存储在 myVotes 数组中。

因为方法要被创建为 Poll 类的一个方法，所以我选择执行一个私有构造函数，使用 XML 文件创建 Poll 对象，这表示 Current()方法非常简单，因为新构造函数会完成所有的工作：

```
public static Poll Current()
{
    Poll myPoll = new Poll(@"c:\polls.xml");
    return myPoll;
}
```

这里仅使用 New 关键字创建 Poll 的一个新实例，把一个文件名字符串传送给构造函数。注意如何把@字符添加到字符串的前端，以便在文件中使用反斜杠(否则，可以写为“c:\\polls.xml”，两个反斜杠可能较难处理)。创建好新对象后，就把它传送回方法。下面介绍这个新构造函数。

4. 方法 —— private Poll(string filename)

构造函数把一个文件名作为其惟一的参数，在该构造函数中，创建 Poll 对象，并用 XML 文件中的数据配置它。在从 XML 文件中提取数据时，有许多选项。可以一个接一个地读取每个节点，提取需要的数据，或者按照要求读取特定的数据段(例如，先读取 Question，再读取每个 Answer 等)。也可以使用串行化功能把 Poll 对象存储在磁盘上，从磁盘上把这些对象并行化，重新创建它们。第四个选项是把 XML 文件读取到 DataSet 对象中，在该对象中，可以把 XML 文件当作其他数据源来处理。对于这个范例，我选择 DataSet 方式，用 ADO.NET 读取 XML 文件。

从图 26-6 中可以看出，在把 XML 加载到 ADO.NET 数据集中时，创建了两个表(记住 DataSet 可以包含几个相关的表)。表 0 表示标记<Poll>，只包含一行一列。表 1 存储了<Answer>标记数据，在两列中包含调查问卷的答案，每行一个答案。

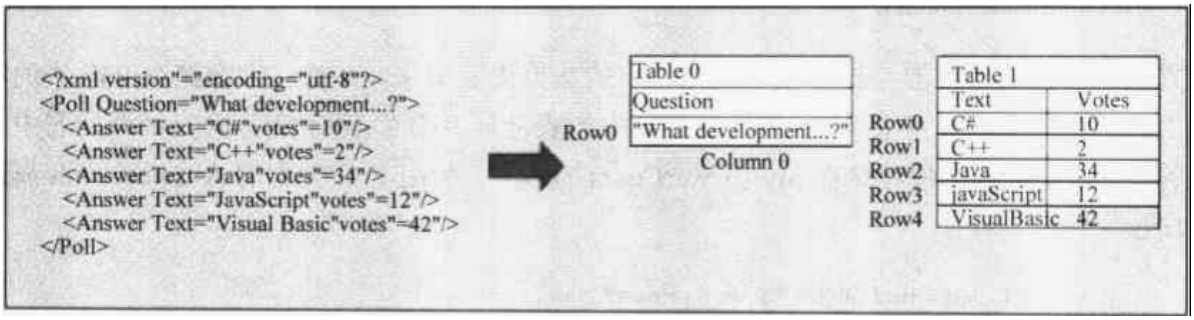


图　26-6

下面一步一步地说明如何在方法中编写代码把 XML 文件加载到数据集中。这里的代码要比前面看到的代码多一些。首先定义构造函数的签名，并说明它包含一个字符串和一个保存调查问卷数据的 XML 文件的文件名：

```
private Poll(string filename)
{
myFilename = filename;
```

接着，创建一个 DataSet 对象存储数据。DataSet 对象在第 19 章已经讨论过了。使用构造函数实例化一个 FileStream 对象，给构造函数传送文件名和 FileMode.Open 枚举，该枚举指定以读取方式打开文件：

```
DataSet myDataSet = new DataSet();
FileStream fsReadXml = new FileStream(filename, FileMode.Open);
```

为了使用这些对象，需要确保在文件的开头引用相应的名称空间：

```
using System.Data;   // For the DataSet object
using System.IO;     // For the FileStream object
```

现在创建一个 XmlTextReader 对象，用它读取 XML 文件中的数据。要创建该对象，仅传送一个 FileStream 对象 fsReadXml 即可：

```
XmlTextReader myXmlReader = new System.Xml.XmlTextReader(fsReadXml);
```

确保引用下面的名称空间：

```
using System.Xml;    // For the XmlTextReader object
```

这行代码完成了许多工作：DataSet 对象上的 ReadXml()方法调用。这个方法把整个 XML 文件读取到 DataSet 对象中，创建表和行，它们不仅要反映数据，还要反映数据的结构。ADO.NET 提供了两个表，第一个表(表 0)只包含一行一列，存储了调查问卷的问题。第二个表包含的每一行都只有一个答案，把答案文本存储在一列上，该答案接收到的选票数存储在另一列上。

```
myDataSet.ReadXml(myXmlReader);
```

下面一行代码关闭文件：

```
myXmlReader.Close();
```

方法的其余部分主要是把存储在数据集中的数据提取出来，并把它们存储在 Poll 类的私有字段上。首先计算表 1 中的行数(记住数据集中的表是从 0 开始索引的)，确定调查问卷中有多少个答案，接着把这个值存储在 myAnswerCount 字段中，使用 Poll 类上的 AnswerCount 属性可以访问该字段：

```
myAnswerCount = myDataSet.Tables[1].Rows.Count;
```

要设置 myQuestion 字段(可 Poll 类上的 Question 属性访问它)，引用表 0，查找表 1 中的第一个项目和该表中包含的惟一一行。ItemArray 属性返回类 object 的一个对象，所以必须使用 ToString()方法(由 object 类提供)把它转换为一个字符串，然后用它设置 myQuestion 字段：

```
myQuestion = myDataSet.Tables[0].Rows[0].ItemArray[1].ToString();
```

在下一个代码块中，初始化 myAnswers 数组，保存问题的答案，通过迭代表 1(表 1 包含了答案的列表)中的每一行，填充该数组。注意数组索引是从 1 开始的(不是 0)，所以用 myAnswerCount+1 来确定数组的维数。Poll 类上的 Votes 和 Answers 属性都是从 1 开始索引的，因为它使表示层有更清晰的界面，把带有“问题 1”的调查问卷引用为第一个问题，而不是第二个问题(如果数组是从 0 开始索引的，就是第二个问题)。这是一个主观的决定，也可以修改代码，使数组从 0 开始索引。

注意如何通过引用表 1 来访问每个答案的文本，表 1 包含了答案的列表，迭代每一行，每次都可以从列 0 中提取出该文本：

```
myAnswers = new string[myAnswerCount+1];
string answer;
for(int i=0;i<myAnswerCount;i++)
{
    answer = myDataSet.Tables[1].Rows[i].ItemArray[0].ToString();
    myAnswers[i+1] = answer;
}
```

下一个代码块几乎与上一个代码块完全相同，但这次是使用包含答案的表中的列 1 填充 myVotes 数组。注意这是一个整数数组，使用 int 对象上的 Parse()方法，把 XML 文件中的字符串转换为一个整数，再把它赋给 myVotes 元素：

```
myVotes = new int[myAnswerCount+1];   // add 1 to use one-indexed
string votes;
for(int i=0;i<myAnswerCount;i++)
{
    votes = myDataSet.Tables[1].Rows[i].ItemArray[1].ToString();
    myVotes[i-1] = int.Parse(votes);
}
```

在这个构造函数的末尾，已初始化了对象，这样 myQuestion, myAnswerCount, myAnswers

和 myVotes 字段都已填充好，并准备使用对象上的 Question, Count, Answers 和 Votes 属性访问它们。创建好对象后，还有另一种(当前)方式可以修改 Poll 对象中的数据，即利用下面将要介绍的 Vote()方法。

在使用 Poll 类上的属性时，并不直接从文件中读取数据，而是从类中的字段上读取数据，认识到这一点是很重要的。设计中包含这个原因是因为它可以避免每次调用类上的属性时，都必须打开 XML 文件。这种方法称为缓存，大大改进了类的性能。

5. 方法 ——public void Vote(int answer)

这个方法带一个整数参数，指定选择哪个答案。该方法必须使用这个值在 Poll 类的 myVotes 字段(缓存的副本)和 XML 文件中为答案增加选票。

在该方法中，首先要检查答案参数中传送过来的值。如果它是 0 或者大于调查问卷中的答案数(存储在 myAnswerCount 中)的一个值，就使用 throw 关键字抛出一个异常。这会导致一个错误，并把该错误显示给用户，除非显式调用了一些检查代码：

```
public void Vote(int answer)
{
if (answer == 0 || answer > myAnswerCount)
{
    throw (new Exception("Invalid choice of answer."));
}
```

要在选票的缓存副本中增加选票数(该副本存储在 Votes 数组中)，可以对数组元素使用＋＋运算符：

```
myVotes[answer]++;
```

下面几行代码与 Poll()构造函数中的开头几行相同，把 XML 文件读取到数据集中：

```
DataSet myDataSet = new DataSet();
FileStream fsReadXml = new FileStream(myFilename, System.IO.FileMode.Open);
XmlTextReader myXmlReader = new System.Xml.XmlTextReader(fsReadXml);
myDataSet.ReadXml(myXmlReader);
myXmlReader.Close();
```

接着在 DataSet 中把表示答案选票数的一个字符串提取出来，该答案是由传送给该方法的 answer 参数指定的。在这个应用程序中要特别小心，因为 DataSet 是从 0 开始索引的，但为了阅读的方便，我们使用从 1 开始的索引。所以在指定 DataSet 中的行时，我们使用“答案 1”，因为前面已经把基于 1 的索引转换为基于 0 的索引值。

记住使用这些“+1” 和“－1”修饰符的原因是我们希望 Poll 类的用户把答案编号为 1,2,3，而不是 0,1,2。尽管这将大大方便表示层代码的编写，但必须做更多的工作。注意这类代码常常会出错，测试是非常重要的，因为常常很难在代码中找出错误：

```
string votes = myDataSet.Tables[1].Rows[answer-1].ItemArray[1].ToString();
```

有了选票数的一个值后，把它转换为一个整数：

```
int votesInt = int.Parse(votes);
```

要修改数据集中各单元格的内容，可提取一个我们感兴趣的特定调查问卷答案的 DataRow。再次注意在对数据集(从 0 开始索引)进行索引时，索引器在末端加上了一个“－1”：

```
DataRow myVotesRow = myDataSet.Tables[1].Rows[answer - 1];
```

要递增选票数，可以给选票变量加 1，把它转换为一个字符串，并把它赋给 DataRow 中的 Votes 列。在执行完这一行后，数据集就包括了新选票数，下面需要把数据集保存回 XML 文件中：

```
myVotesRow["Votes"] = (votesInt+1).ToString();
```

创建一个 StreamWriter 对象，DataSet 可以在其中写入数据，给它传送 XML 的文件名，该文件名保存在 myFilename 字段中：

```
StreamWriter myStream = new StreamWriter(myFilename);
```

为了指示 ADO.NET 把 XML 插回到文件中，调用 DataSet 类上的 WriteXml()方法，该方法把它传送给流，并且把指定的值也插回文件，我们希望输出不带模式信息的 XML：

```
myDataSet.WriteXml(myStream,XmlWriteMode.IgnoreSchema);
```

最后，关闭流，退出方法：

```
myStream.Close();
}
```

这就完成了，我们只需再次保存数据。.NET 会使数据在 XML 和数据集之间移时变得非常简单。

6. 属性 ——public int AnswerCount

AnswerCount 属性返回调查问卷中的答案数。它是只读的，返回存储在私有字段 myAnswerCount 中的值。这个字段只能由 Current()方法初始化，但类的将来版本将允许用其他方式实例化和管理它。例如，如果要编程创建新的调查问卷，需要创建一个空白的调查问卷，在其中配置答案数：

```
public int AnswerCount
{
    get
    {
        return myAnswerCount;
    }
}
```

7. 属性 ——public string Question

这个属性与 AnswerCount 属性相同，但它返回 myQuestion 字段的值，该字段存储了表示调查问卷中问题的字符串：

```
public string Question
{
    get
    {
        return myQuestion;
    }
}
```

8. 属性 ——public string[] Answers

这个属性返回固定长度的数组，该数组包含调查问卷中所有答案的文本。为此，它返回私有字段 myAnswers，该字段用 Current()方法来填充：

```
public string[] Answers
{
    get
    {
        return myAnswers;
    }
}
```

9. 属性 ——public int[] Votes

这个属性返回 myVotes 数组，该数组存储了调查问卷中每个答案的选票数。myVotes 字段用 Current()方法来填充：

```
public int[] Votes
{
    get
    {
        return myVotes;
    }
}
```

注意这个属性返回选票数的一个缓存副本，每次不是从 XML 文件中读取它，如果有多个 Poll 对象更新了选票数，就会出问题。如果需要有几个注册选票的 Poll 对象，就应重新编写这个属性，使之直接从数据源中读取选票数。

10. 名称空间

从完整性的角度来看，并提示我们正在使用.NET Framework 名称空间，这里有一些在 Poll 类中使用的名称空间：

```
using System;
using System.Data;
using System.IO;
using System.Xml;
```

记住，如果在编写代码前添加了名称空间声明，Visual Studio.NET 的 IntelliSense 将有助于编写代码。

11. 测试 Poll 类

记住前面创建了一个控制台应用程序，用于测试 Poll 事务对象。现在必须建立 Poll 类，使用其类文件中的 using Polling;语句，重新建立控制台应用程序。把 XML 文件保存到 C:\盘的根目录下，使用 Visual Studio .NET 添加对 Poll 装配件的一个引用。记住要在 Visual Studio.NET 的解决方案中建立工程，只需从 Build 菜单中选择 Build Solution (Ctrl+Shift+B)即可。要运行控制台应用程序，可右击 PollC 工程，选择 Set As Startup Project，然后从 Debug 菜单中选择 Start Without Debugging(Ctrl+F5)，得到如图 26-7 所示的结果。

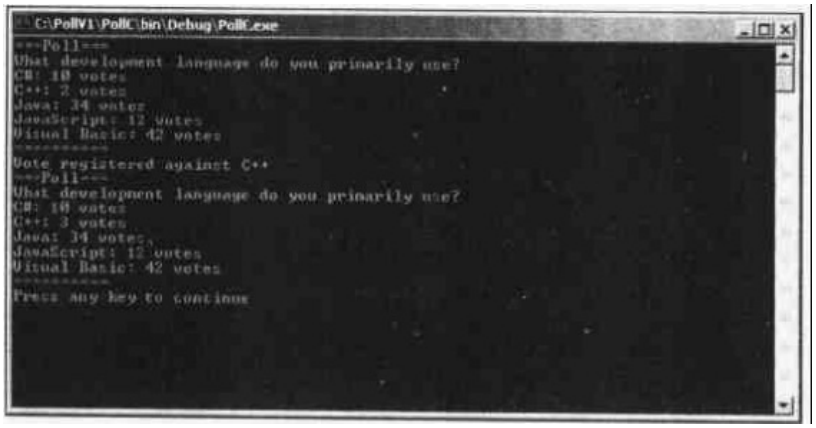


图　26-7

这个测试相当简单，没有检查事务对象的每个方面是否都有效，但它很便于运行一个基本检查。它没有检查的一个区域是数据存储在磁盘上的永久性，以及调查问卷数据是否已存储到 XML 文件中。

许多程序员在开始编写事务对象前，都编写过这样的测试代码，确保测试代码对事务对象进行全面的测试。这样在修改代码时，很容易看出何时会出问题。

26.2.6　用户界面(表示层)

现在已经处理完数据层和事务逻辑层，并在表示层编写了一个简单的控制台应用程序。下面该使用 ASP.NET 为调查问卷建立一个 Web 界面。首先需要在 Visual Studio.NET 的解决方案中添加一个 ASP.NET 工程，称之为 PollW (Poll Web 的缩写)，如图 26-8 所示。

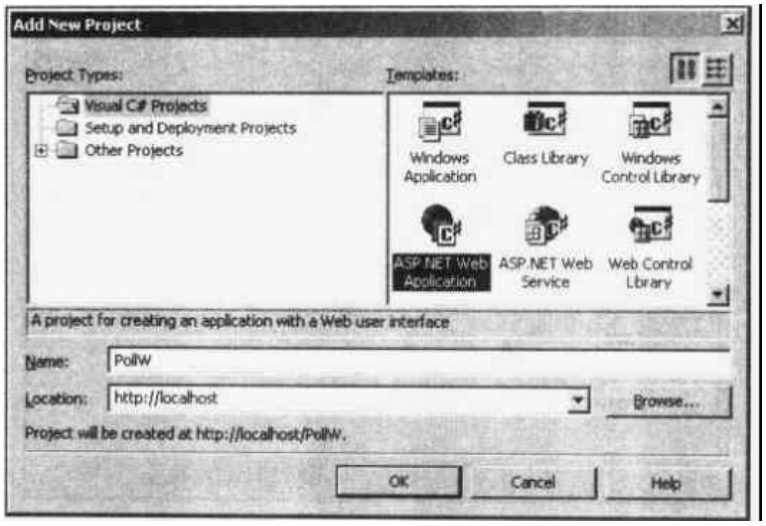


图　26-8

注意在创建新的 Web 应用程序时，Visual Studio.NET 会自动创建几个文件，包括初始 ASP.NET 页面 WebForm1.aspx。把它重命名为 default.aspx，以便把应用程序引用为 http://localhost/PollW/。记住 default.aspx 有一个关联的后台编码的代码文件 default.aspx.cs，后面介绍这个文件。显然，如果 Web 服务器没有运行在本地机器上，就需要指定服务器名，而不是使用 localhost。

在继续前，必须考虑一些重要的问题。在 ASP.NET 中，有一个建立 Web 窗体的奇异系统，它非常类似于 Visual Basic，因为在该系统中可以使用 Visual Studio 中的拖放操作，并使用基于事件的编程方式。但是有一个区别，在 ASP.NET 中，一个页面上只能有一个服务器端窗体。构造调查问卷 Web 站点的逻辑方式是使用一个服务器端窗体，在该窗体中，每个答案有一个单选按钮和一个注册答案的 Vote 按钮。但是，如果采用这种方法，包含调查问卷的任何页面都不能包含其他服务器端窗体，如果把调查问卷放在站点的许多页面上，以后要增加另一个窗体(例如第二个调查问卷或一个搜索控件)，就会受到限制。为了避免这个问题，有机会查看.NET 中很少有人知道的部分，对调查问卷 Web 站点采用的设计不能使用服务器端窗体，而应使用略低级别的 ASP.NET 功能，ASP 3.0 程序员可能很熟悉这个功能。尽管它们在 ASP.NET 上没有普遍使用，但仍能了解它们是如何在这样不适合使用服务器端窗体的情况下发挥作用的。

一般情况下，对于简单的回送窗体，可以使用一个页面，在提交按钮上使用 OnClick 事件，注册一个人的答案。但是，对于这种设计，要避免使用服务器端窗体，应使用两个页面。第一个页面显示问题和一组答案，在每个问题的旁边显示一个 Vote 链接。这个链接把答案数作为一个参数传送给第二个页面，第二个页面再使用传送过来的值注册选票，并显示到目前为止的调查问卷的结果。

图 26-9 所示的是第一个页面 default.aspx。

图 26-10 所示的是第二个页面 vote.aspx。

第一个页面 default.aspx 显示了调查问卷，在使用 Vote 链接选择一个答案时，可以看到 URL 上的 1 传送给了 vote.aspx。这个参数用于注册选票。vote.aspx 显示每个答案的选票数。

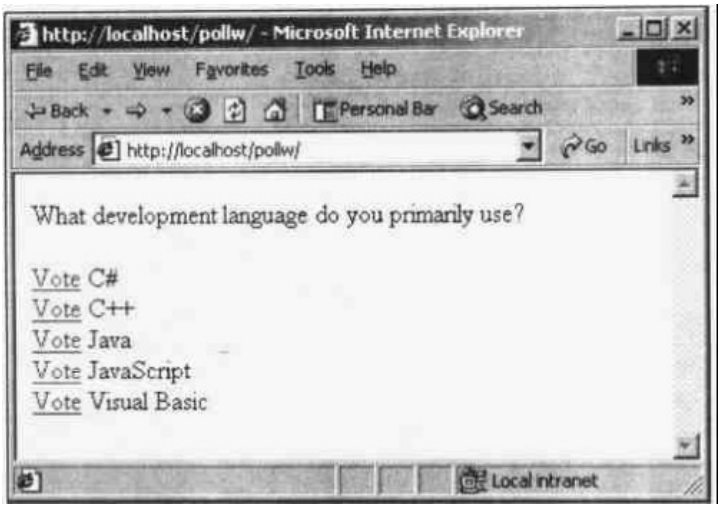


图　26-9

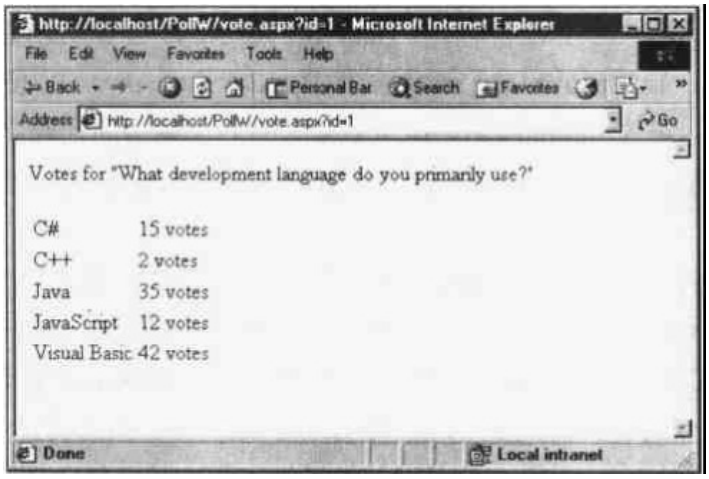


图　26-10

首先看看 ASP.NET 文件 default.aspx，它是由 Visual Studio.NET 创建的，该页面是标准的，提供了几个零碎的东西，例如外部的 HTML 标记、标题和主体标记：

```
<%@ Page language="c#" Codebehind="Default.aspx.cs" AutoEventWireup="false"
Inherits="PollW.WebForm1" %>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN" >
<HTML>
    <HEAD>
        <meta name="GENERATOR" Content="Microsoft Visual Studio 7.0">
    </HEAD>
    <BODY>
        <form id="Form1" method="post" runat="server">
        </form>
    </BODY>
</HTML>
```

注意它还包括一个服务器端窗体标记(<form runat="server">)。首先应删除它，这将用作

个头脑中的提示信息，因为我们不使用服务器端窗体回送。同时，还要插入两个服务器端的 Label 控件，显示调查问卷的问题和答案列表：

```
<%@ Page language="c#" Codebehind="Default.aspx.cs" AutoEventWireup="false"
Inherits="PollW.WebForm1" %>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN" >
<HTML>
    <HEAD>
        <meta name="generator" Content="Microsoft Visual Studio 7.0">
    </HEAD>
    <BODY>
        <asp:Label id="QuestionLabel" runat="server"></asp:Label>
        <asp:Label id="AnswersLabel" runat="server"></asp:Label>
    </BODY>
</HTML>
```

现在有一个空白的 ASP.NET 页面，其中包含放置问题的 Label 控件。接着在它的下面添加答案，但不是在.aspx 文件中添加，而是在后台编码文件 default.aspx.cs 中添加。

下面是 Visual Studio.NET 在 default.aspx.cs 中提供的标准代码，它提供了对事件的基本支持，以及初始化和页面加载的方法框架。注意在默认情况下，Visual Studio.NET 会给 Page_Init() 方法生成一个事件处理程序，这是在创建 Page 对象时调用 Page_Init()方法的方式：

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

namespace PollW
{
    /// <summary>
    /// Summary description for WebForm1.
    /// </summary>
    public class WebForm1 : System.Web.UI.Page
    {
        public WebForm1()
        {
            Page.Init += new System.EventHandler(Page_Init);
        }
```

```
        private void Page_Load(object sender, System.EventArgs e)
        {
            // Put user code to initialize the page here
        }

        private void Page_Init(object sender, EventArgs e)
        {
            //
            // CODEGEN: This call is required by ASP.NET
            //
            InitializeComponent();
        }

        #region Web Form Designer generated code
        /// <summary>
        /// Required method for Designer support - do not modify
        /// the contents of this method with the code editor.
        /// </summary>
        private void InitializeComponent()
        {
            this.Load += new System.EventHandler(this.Page_Load);
        }
        #endregion
    }
}
```

我们进行的第一个修改是给 QuestionLabel 和 AnswersLabel 添加声明，以便在页面的后台编码中访问它们。如果使用 Visual Studio .NET 拖放了标签控件，而不是手动添加它们，那么这些声明就由 Visual Studio .NET 自动完成。下面的代码在 WebForm1 类的顶部，即构造函数的前面：

```
public class WebForm1 : System.Web.UI.Page
{
    protected System.Web.UI.WebControls.Label QuestionLabel;
    protected System.Web.UI.WebControls.Label AnswersLabel;
```

我们主要关心的是 Page_Load()方法，在浏览器请求页面时执行它。在这个方法中，要添加插入问题文本的代码，并给每个答案添加 HTML 控件。下面一步一步地解释方法以及新代码。首先在 Poll 类中创建一个对象，用静态方法 Current()初始化它：

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

```
{
    Poll aPoll = Poll.Current();
```

在下一行上，开始使用 Poll 对象，并利用该对象在标签控件(QuestionLabel)上设置 Text 属性：

```
    QuestionLabel.Text = aPoll.Question;
```

接下来的几行代码构成一个循环，循环每个答案，并给每个答案输出一些 HTML。注意如何使用 Poll.AnswerCount 属性设置 for 循环的循环次数。我们还创建了一个字符串变量，以存储 HTML。在该 HTML 中，将创建和插入两个新换行标记：

```
string HtmlLink = "<br /><br />";
for(int i=1;i<=aPoll.AnswerCount;i++)
{
```

对于每个答案，需要输出一个带有文本“Vote”的文本链接，还要显示答案的文本。必须确保链接指向 vote.aspx，URL 包含表示答案的数字(例如 1 表示 C#)。HTML 如下所示：

```
<a href="vote.aspx?id=1">Vote</a> C#<br />
```

注意在行的末尾还有一个
标记，确保每个答案在屏幕上显示在单独的一行上。也可以使用一个表格使格式更整洁，但现在用比较简单的方式。

“ ”称为非中断的空格，可迫使浏览器在“Vote”文本和答案文本(例如 C#)之间插入一个空格，这样两个文本就用一个字符分隔开。要创建这个 HTML，可以只使用一个字符串(称为 HtmlLink)，再使用变量 i 和 Poll 对象上的 Answers 数组属性，插入 id 值和答案文本。代码分隔为两行，这样它才能放在屏幕上(一般情况下，如果在几行代码中建立字符串，就应使用 StringBuilder 类)：

```
    HtmlLink += "<a href=\"vote.aspx?id="+i+"\">Vote</a>";
    HtmlLink += " " + aPoll.Answers[i] + "<br />";
```

现在有一个字符串变量 HtmlLink，它包含一个答案的 HTML。接着，再次循环，添加另一个答案的 HTML。然后使用 HTML 代码设置第二个标签控件的文本：

```
    }
    AnswersLabel.Text = HtmlLink;
    }
}
```

如果把 default.aspx 加载到浏览器上，调查问卷就会在顶部显示一个问题和一个答案列表(但 Vote 链接还不起作用)。

下面看看 vote.aspx 页面。要创建它，使用 Visual Studio.NET 的 Project 菜单中的 Add Web Form 项目。确保在对话框中选中了 Web Form，并在 Name 字段中键入 vote.aspx，如图 26-11 所示。

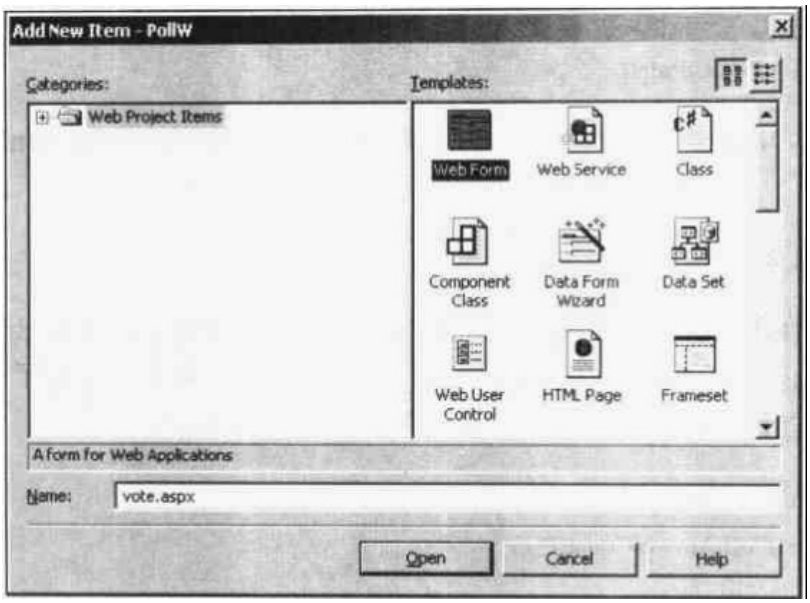


图　26-11

Visual Studio.NET 将生成 vote.aspx 和 vote.aspx.cs (后台编码文件)，打开 vote.aspx 进行编辑。vote.aspx 几乎与 default.aspx 完全相同，它是空白的页面，其中只有一个介绍页面的 Label 标签。注意服务器端窗体标记已经手动删除了，因为我们不需要它们：

```
<%@ Page language="c#" Codebehind="vote.aspx.cs" AutoEventWireup="false" Inherits="PollW.vote"
%>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN" >
<HTML>
    <HEAD>
        <meta name="GENERATOR" Content="Microsoft Visual Studio 7.0">
        <meta name="CODE_LANGUAGE" Content="C#">
        <meta name="vs_defaultClientScript" content="JavaScript (ECMAScript)">
        <meta name="vs_targetSchema"
              content="http://schemas.microsoft.com/intellisense/ie5">
    </HEAD>
    <BODY>
        <asp:Label id="titleLabel" runat="server"></asp:Label>
        <br />
        <br />
        <asp:Panel id="resultsPanel" runat="server"></asp:Panel>
    </BODY>
</HTML>
```

与 default.aspx 一样，需要添加控件的声明：

```
public class vote : System.Web.UI.Page
{
    protected System.Web.UI.WebControls.Label titleLabel;
    protected System.Web.UI.WebControls.Panel resultsPanel;
```

下面介绍 Page_Load()方法，该方法必须插入一个页面的标题，重新迭代调查问卷的问题，再显示每个问题和它的选票数。我们先使用 Current()方法实例化一个 Poll 对象，使该对象包含从 XML 文件提取出来的当前调查问卷数据。

```
private void Page_Load(object sender, System.EventArgs e)
{
    Poll aPoll = Poll.Current();
```

在下一行上，使用 Page 类的 Request 对象(页面继承了该对象)，再使用它的 QueryString 集合访问 URL 上的参数(或者称为查询字符串)：

```
string parameter = Request.QueryString["id"];
```

接着，检查 id 值是否为空，以确定是否有一个参数。如果该值为空，就尝试注册一个选票，这两种方式都会显示目前调查问卷的结果。注意在 int 类上如何使用 Parse()方法把字符串转换为整数。如果传送了一个不能转换为整数的字符串，这个方法调用就会失败。但在本例中对传送给页面的内容有高度的控制，所以捕获这个错误没有什么好处：

```
if (parameter != null)
{
    aPoll.Vote(int.Parse(parameter));
}
```

下一行代码设置标签 titleLabel，使用它的 Text 属性存储页面的介绍，重申问题。注意在字符串中是如何使用反斜杠('\')来包含文本中的'"'字符的：

```
titleLabel.Text = "Votes for \"" + aPoll.Question + "\"";
```

在 Page_Load()方法中，其余代码在单独的一行上显示各个答案，以及接收到的选票数，显示调查问卷的结果。首先实例化一个 Table 对象(存储结果的 HTML 表)，把 AnswerCount 属性作为其限制值，用一个 for 循环进行循环：

```
Table myTable = new Table();
for(int i=1;i<=aPoll.AnswerCount;i++)
{
```

接着用 myTable 和存储在 Poll 对象中的信息把 HTML 表组合起来。创建一个新表行，并把它添加到表中：

```
TableRow myRow = new TableRow();
myTable.Controls.Add(myRow);
```

现在创建一个新的表单元格，使用其 Text 属性把它的内容设置为保存调查问卷的答案，然后把它添加到表行中：

```
TableCell myAnswerCell = new TableCell();
myAnswerCell.Text = aPoll.Answers[i];
```

```
        myRow.Controls.Add(myAnswerCell);
```

对选票数进行相同的操作，创建一个新的表单元格，设置其 Text 属性以存储选票数(在末尾添加一个字符串 votes)，然后把它添加到表行中：

```
        TableCell myVotesCell = new TableCell();
        myVotesCell.Text = aPoll.Votes[i].ToString() + " votes";
        myRow.Controls.Add(myVotesCell);
    }
```

在循环的外部，把该表添加到 resultsPanel 服务器控件的控件集合中，如下所示：

```
    resultsPanel.Controls.Add(myTable);
}
```

这个操作的结果如图 26-12 所示。

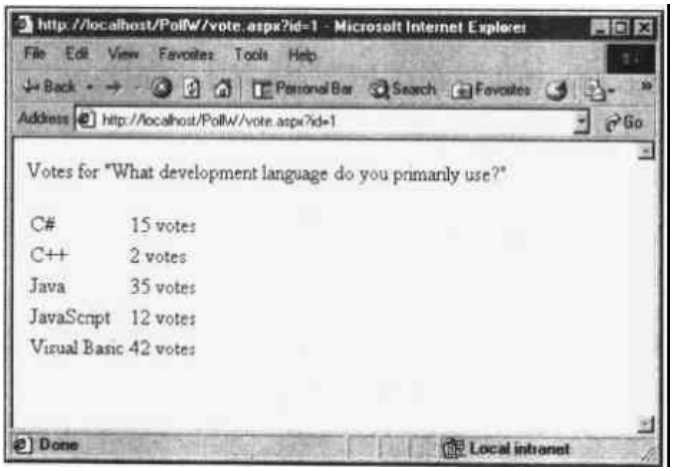


图　26-12

26.2.7　版本 1 的价值

现在有一个小型的 3 层应用程序可跟踪调查问卷的选票，并介绍了在 ASP.NET 和控制台应用程序中访问的事务对象。这个应用程序相当基本，注意它有一些局限性：

- 多个选票：这可能是最重要的缺陷：任何人都可能访问站点，对同一个调查问卷的问题重复投票。如果调查问卷的结果涉及到第三方厂商，它们可能抵挡不住在某个地方添加一些选票的诱惑。
- 可重用性：尽管我们建立的代码是自我包含的，非常简单，可重用，但界面代码分为.aspx 文件和.cs 文件(后台编码文件)。每次要增加一个调查问卷时，就必须把.aspx 文件和 .cs 文件合并到一个新页面上，在该新页面上增加调查问卷。这很适于把调查问卷插入到一个页面上，使用 ASP.NET，其可能性很大。
- 编辑调查问卷：XML 文件虽然很简单，但有时编辑它的人会忘记一个重要的字符，Web 站点于是开始显示错误。有许多方法可以处理这个问题。可以温文地处理该问题，显示一个信息“很抱歉此时该调查问卷不能使用”。但更好的解决方案是提供一个用户界面，站点管理员

可以使用它来编辑调查问卷。该界面可以使用许多技术来实现，但最合适的两个技术是 ASP.NET(编辑调查问卷的 Web 界面)和 Windows Forms 界面。

- 监控调查问卷：在应用程序中，站点管理员无法监控调查问卷。如果收集的数据非常重要，最好进行实时观察，看看选票主要集中在什么地方。要解决这个问题，最好建立另一个管理界面，该界面实时显示调查问卷的结果。这个界面可以使用 ASP.NET 建立，或者扩展表示层，建立 Windows Forms 界面。

- 数据管理：数据存储在 XML 文件中时，必须确保能够查找到这个文件。为此，在备份 Web 服务器时，也要备份 XML 文件。这会增加 Web 站点管理的系统开销，特别是在其他地方(例如数据库或其他文件)也有数据时，就更是这样。如果扩展了 Web 站点，使之运行在多个服务器上，也会出问题。可以把数据文件存储在两个 Web 服务器都可以访问的服务器上，但简单的问题就会很快变得复杂起来。如果考虑这些维护问题，最好把所有的问题都放在一起进行解决，使用一个中央数据库来存储所有的数据，而不是一个 XML 文件。

- 重复的代码：注意在 Current()方法中有一个把 XML 文件加载到 DataSet 中的代码块，这段代码在 Vote()方法中被重复了。在现实世界中，这个代码块应放在一个单独的私有方法中，让 Current()和 Vote()调用都可以从 XML 文件中获取一个 DataSet。无论如何应避免有重复的代码，编程工作才会更容易完成。

- 多个调查问卷：我们的设计仅允许有一个调查问卷。在现实世界中，每个调查问卷都会被新的替代，站点管理员需要编辑 XML 文件，以显示新的调查问卷。如果需要，他们还可以保留旧的 XML 文件的副本，以创建调查问卷数据的归档文件。手工的工作量相当大，实际上，我们可以扩展设计，自动把所有的调查问卷归档到数据库中，并为管理员添加新调查问卷提供一个接口。有了几个调查问卷，还需要指定哪个是当前进行的调查问卷。

- 并发性：我们使用一个 XML 文件存储调查问卷数据，如果 Web 站点很繁忙，一些选票就有可能未被注册，因为访问文件的企图可能会使另一张选票插入到文件中。这在低通信量的站点上并不是个问题，但如果希望注册许多选票，就需要修改数据层，以便更好地处理频率较高的选票。最富逻辑的方式是使用一个数据库服务器，如 SQL Server 或 Oracle，这样就不会有并发性问题了。

- 可视化的外观：调查问卷页面 default.aspx 和 vote.aspx 都缺乏可视化设计的窗体。对于几乎所有的 Web 站点来说，外观是需要设计的，这样才能使之更吸引人。在这方面，焦点主要集中在开发过程和使用的技术上，读者应把页面的外观设计为自己希望的模样。

- 错误处理：这个应用程序提供了一些简单的功能，但它提供的错误处理过于简单。问题是如果发生了什么预料不到的事，应用程序就会生成完全不适合用户查看的信息。例如，如果 XML 文件不能使用，就会抛出一个异常。在投入销售之前，应用程序必须进行扩展，防止它本身及其用户不出现失败。

知道有一些领域需要改进和扩展调查问卷程序后，下面进入第二个版本，添加其他一些功能，消除这些限制中的两个。我们只解决了这两个限制问题，如果读者决定执行类似的调查问卷系统，就应解决其他问题。

26.3 版本 2

没有建立软件，而是在演化。只要一个软件的任何版本在建立过程中，就将计划准备扩展和修改它。在上一节中，可以对调查问卷程序进行许多改进。在它的第二个版本中，我们将解决两个限制问题，改进已有的代码，并特别关注它们是在表示层、事务逻辑层，还是在数据层中进行修改。

26.3.1 其他要求

下面是调查问卷 Web 站点的第二版本的新要求：

- 调查问卷结果工具：站点管理员需要在台式机器上运行一个工具，它可以连续不断地显示当前的选票数。应用程序必须是一个 Windows 程序，并通过 TCP/IP 连接到 Web 站点上。这个工具必须显示问题、每个答案及其选票数，选票的百分比以及选票总数。
- 每人一张选票：站点访问者应只能对每个调查问卷投一次票。
- 调查问卷控件：必须能确保事务对象是可用的，把一些简单的语句添加到.aspx 文件中，轻松地把调查问卷添加到任何 ASP.NET 页面上。

26.3.2 调查结果工具

1. 接口

在 3 个新要求中，首先看看最大的一个要求。它涉及到使用 Windows Forms 创建一个桌面 Windows 应用程序，根据显示调查问卷信息的方式，这些要求是相当松散的，但可以假定所有的内容都显示在一个窗体上。

下面使用的设计在顶部有几个标签，存储问题和选票总数。它们的下面是一组数据行，每行包含一个答案文本及其选票数(百分比)，还有一个可以图形化地显示选票数的栏，如图 26-13 所示。

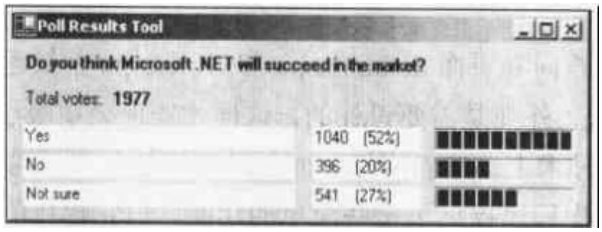


图 26-13

这些都非常简单，但有一个小问题，在设计窗体时，不知道有多少个答案。可以使用一个列表框或另一个长度为动态变化的控件，它们可以使工具看起来很不错，并描述了 Windows Forms 的一些功能，下面将动态地创建每行数据。如果调查问卷有许多答案，窗体上就会有许多数据行，其大小会重新设置，以显示它们，如图 26-14 所示。



图 26-14

2. 调查问卷的 Web 服务

在可能使用这个应用程序的场合中，结果工具不可能运行在与 Web 站点相同的机器上。实际上，我们可以确定这是不可能的，但可以假定两台机器的连接支持 TCP/IP。对于使用 HTTP 通过 TCP/IP 进行通信的对象来说(通常这表示在 Internet 上，因为内部网应用程序一般会远程调用)，我们将使用 Web 服务。Web 服务是通过网络(包括 Internet)进行远程调用的一种方式，并利用 SOAP 和 XML 标准管理该通信。在.NET 中，特别是在 Visual Studio.NET 中，Web 服务执行起来是非常简单的。有许多工作都是在后台完成的，读者可能想了解这些工作，但建立一个有效的 Web 服务是很烦琐的。

首先为调查问卷程序创建一个 Web 服务，它应在 PollW ASP.NET 应用程序中构造，因为 Web 服务要在 Web 站点的一个页面上被访问。在 Web 服务中创建 3 个 Web 方法，这些方法都可以通过 Internet 访问：

- Answers()：返回一个字符串数组，其中包含每个答案的文本。
- Votes()：返回一个整数数组，其中包含每个答案的选票数。
- Question()：返回一个字符串，其中包含调查问卷中的问题。

在理论上，可以把这些 Web 方法放在 Poll 类上，但为了把可通过内部网访问的方法与事务对象区分开来，创建一个新类，这个新类显示了这些 Web 方法，而这些方法利用已有的 Poll 类。应对 Web 服务提供的接口进行完全控制，其方式要独立于底层事务逻辑类的设计。

要在 Visual Studio 中创建一个 Web 服务，只需使用 Project 菜单中的 Add Web Service，把 Web 服务添加到 PollW 工程(Web 应用程序)上。输入 Web 服务文件的名称，这是显示方法的页面。Web 服务使用.asmx 扩展名，但其结构和内容都类似于.aspx 文件，如图 26-15 所示。



图 26-15

创建好.asmx 文件后，还有一个后台编码文件 results.asmx.cs，在该文件中，可以编写执行 Web 方法的代码。如果向下滚动到 results.asmx.cs 文件，就可以看到底部有一个加了注释的样例 Web 方法：

```
// WEB SERVICE EXAMPLE
// The HelloWorld() example service returns the string Hello World
// To build, uncomment the following lines then save and build the project
// To test this web service, press F5
//
//        [WebMethod]
//        public string HelloWorld()
//        {
//            return "Hello World";
//        }
```

从提供的样例中可以看出，创建 Web 方法非常简单，只需在方法声明中添加“[WebMethod]”属性即可。下面是 Web 方法 Question()的代码：

```
[WebMethod(Description="Return the poll question")]
public string Question()
{
    Poll aPoll = Poll.Current();
    return aPoll.Question;
}
```

注意我们给[WebMethod]属性传送了一个参数，给 Web 方法提供一个简单的描述。方法体使用 Poll 类的方式与控制台应用程序和 Web 应用程序一样，它使用 Current()方法实例化一个 Poll 对象，并返回 Question 属性。使这个方法可以在 Internet 上使用的代码都在后台上。

选票数和答案的 Web 方法非常类似：

```
[ WebMethod(Description="Return the results for the poll")]
public int[] Votes()
{
    Poll aPoll = Poll.Current();
    return aPoll.Votes;
}

[ WebMethod(Description="Return the answers in the poll")]
public string[] Answers()
{
    Poll aPoll = Poll.Current();
    return aPoll.Answers;
}
```

在每个方法中，都包含了一个简短描述，并利用 Poll 类提供需要的数据。这就是集中式事务对象的优点，一旦有了提供核心功能的对象，就可以很容易地在几个不同的地方使用它们。

现在在 Web 浏览器上通过 HTTP 打开 results.asmx，就可以看到图 26-16。注意每个 Web 方法都按照名称列了出来，并包括一个描述。利用这些信息将很容易看出每个 Web 方法的作用。

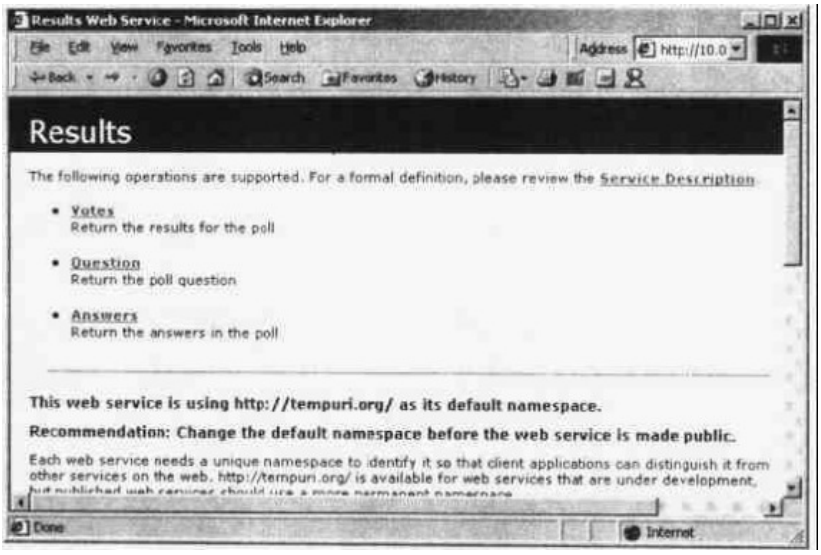


图　26-16

这个人们可以理解的页面由.NET 自动生成，允许我们检验 Web 服务。可以看出，它正确地显示了前面编写的 3 个 Web 方法。页面的其余部分提供了在把 Web 服务面市之前如何配置它的建议。如果选择 Web 方法 Question，就会显示一个页面，介绍该 Web 方法的更多信息，以及如何使用它，如图 26-17 所示。

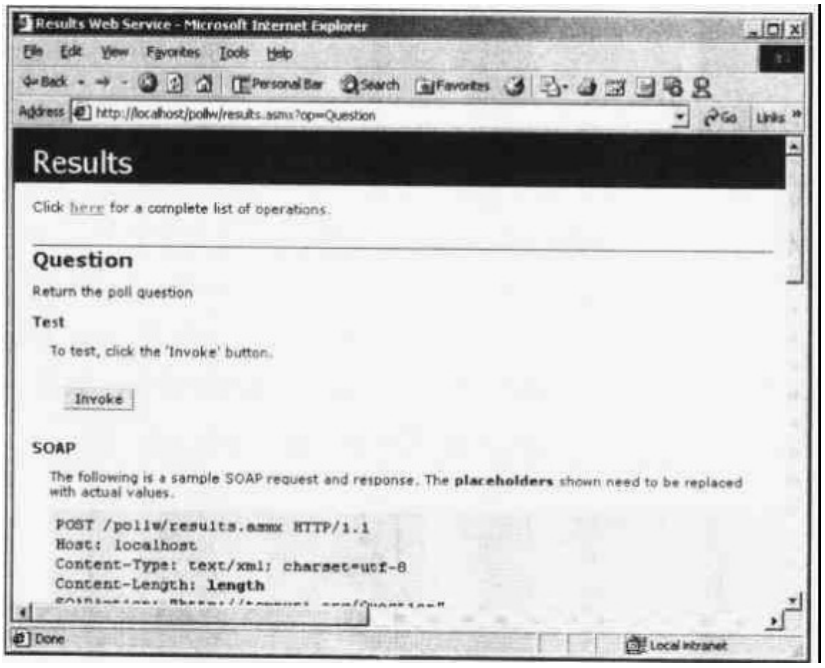


图　26-17

在 Web 方法 Question 的页面上，有一个标记为 Invoke 的按钮，可用于测试 Web 服务。如果单击这个按钮，就可以看到 Web 方法 Question()的 XML/SOAP 输出结果，如图 26-18 所示。



图 26-18

注意访问这个 Web 方法的 URL 是非常简单的，还有一个简单的 XML 文档用于给我们返回结果——调查问卷上的问题。

如果查看一下 Web 方法 Votes() 的结果，将得到类似的结果，但应注意.NET 和 Web 访问处理返回的整数数组是非常容易的。.NET 在后台完成的工作量是很大的，但一般不需要知道这些，这仅意味着我们的工作效率更高，如图 26-19 所示。

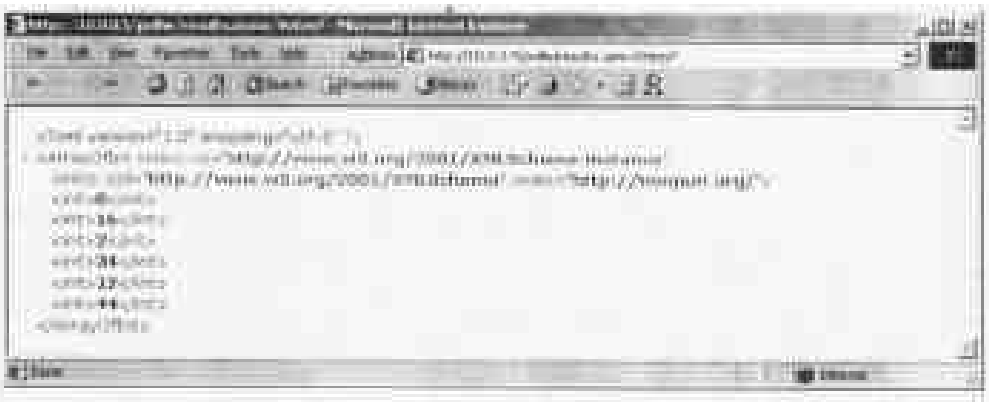


图 25-19

3. Windows 应用程序

显然，建立 Web 方法和使它们可以通过 HTTP 使用是相当容易的。那么该如何使用它们？如何把这个 XML 转换为某些有用的东西？我们不必亲自完成这个任务。.NET 会替我们完成。

现在已经有了一个有效的 Web 服务，它提供的 Web 方法可以访问调查问卷的数据，下面就应把注意力转向 Windows Forms 应用程序。在 File 菜单中选择 Add Project 选项，再选择 Windows Application，键入名称 PollD(Poll Desktop 应用程序的缩写)，如图 26-20 所示。



图 26-20

Visual Studio.NET 将创建一个简单的类文件 Form1.cs，并在设计视图中打开窗体，开始编辑。虽然使用代码可以在窗体上创建大部分控件，但对顶部的 3 个控件用 Visual Studio.NET 中的设计视图来创建。第一个控件是一个标签控件，称为 questionLabel，显示调查问卷中的问题。第二个控件也是一个标签控件“Total votes:”。第三个控件是一个标签控件 totalVotesLabel，显示在调查问卷中注册的选票总数。添加完这 3 个控件后，窗体应如图 26-21 所示。

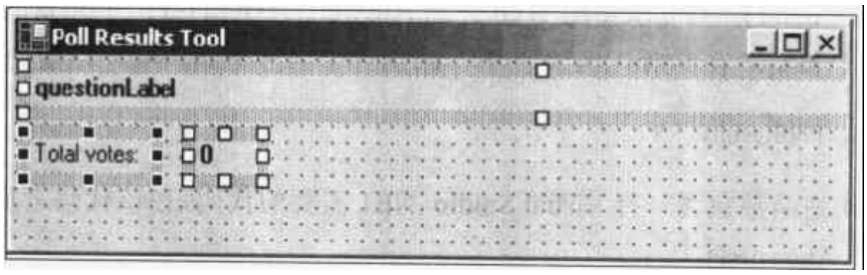


图　26-21

不需要考虑窗体的高度或宽度，因为这些将在后面编程设置。现在有了一个简单的窗体，使用 Web 方法 Question()给它添加一些实际数据。为此，需要在应用程序中增加一个特定的类，叫做代理类。该类包含调用 Web 方法的所有代码，还把 XML 结果打包到窗体中，该 XML 可以在代码中使用(这是通过第 25 章介绍的 SOAP 技术来完成的)。

创建代理类有两种方式。可以使用 wsdl.exe 实用程序或 Visual Studio.NET。后者是一种比较容易的方式，这里就使用它。要在 PollD 工程中添加代理类，右击该工程，选择 Add Web Reference，打开一个对话框，在对话框的顶部有一个字段，在其中键入要使用的 Web 服务的 URL。也可以使用 UDDI，这是一个能在 Internet 上使用的 Web 服务库。这里不使用 UDDI，仅在字段中键入 Web 服务的 URL：http://localhost/PollW/results.asmx。

接着单击回车键或绿箭头按钮，对话框就会显示人们可以理解的 Web 服务页面，这个页面我们在前面见到过。确保定位了正确的 Web 服务后，单击 Add Reference，Visual Studio 就会生成一个代理类，如图 26-22 所示。

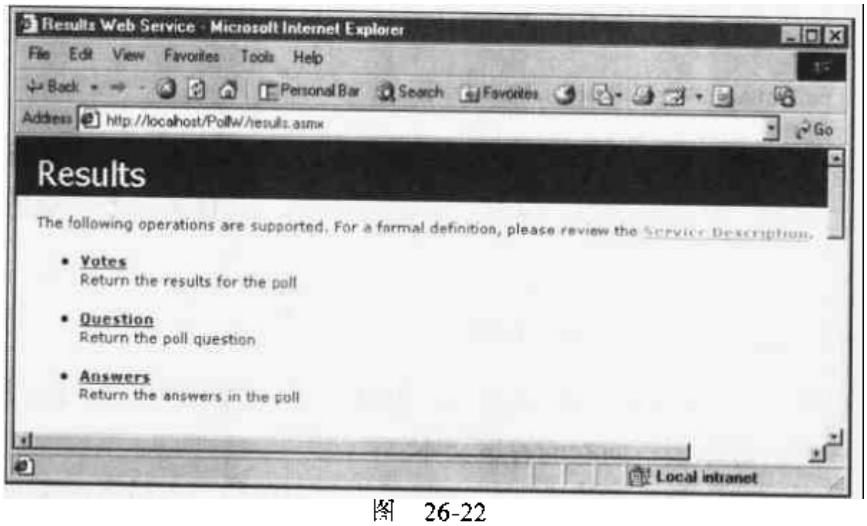


图　26-22

Visual Studio.NET 在 Windows 应用程序中创建一个新的名称空间 PollD.localhost，它将包含 Web 服务提供的所有 Web 方法。在 Solution Explorer 中，右击 localhost，把它重命名为

PollD.PollResults，使名称更其有可读性。下面使用 PollD.PollResults 名称空间中的一个类 Results，该类包含 3 个可以使用的方法：Question(), Votes()和 Answers()。

无论您相信与否，这就是我们需要做的工作。下面回到 Form1.cs，我们还没有任何显示调查问卷中数据的控件。需要动态创建每个控件。但在此之前，要知道调查问卷中有多少个答案，这样才能确定要创建多少个控件，窗体要设置为多大。为了简单起见，这个案例分析没有包含 Web 方法 AnswerCount()，但可以使用 Web 方法 Answers()返回的数组中元素的个数，来确定调查问卷中有多少个答案。在 Form1.cs 文件中，添加一个名称空间声明，以便使用代理类：

```
using PollD.PollResults;
```

Form1 中的构造函数包含一行 Visual Studio .NET 需要的代码，在该代码之后插入一些代码，以使用 Web 方法 Answers()：

```
public Form1()
{
    //
    // Required for Windows Form Designer support
    //
    InitializeComponent();
```

首先实例化 Results 类，这是一个代理类，通过它可以访问任何远程 Web 方法。

```
    Results pollResults = new Results();
```

在下一行代码上，显示一些文本，当窗体下载数据时，这些文本是可见的。一般情况下，这些文本很快就会被替换：

```
    questionLabel.Text = "Loading poll data...";
```

接着的一行代码调用 Web 方法 Answers()。注意，这是一个一般的方法调用，由 Visual Studio.NET 生成的代理类隐藏了所有的内部工作：

```
    string[] pollAnswers = pollResults.Answers();
```

在 Form1 类的构造函数中添加一个变量 myAnswerCount，该变量保存调查问卷中答案的总数。从 Web 方法 Answers()获得的数组是从 0 开始索引的，而且第一个元素是空的，所以必须从该总数中减 1：

```
    int myAnswerCount = pollAnswers.Length - 1;
```

知道了有多少个答案后，就可以计算窗体的布局，并添加需要的控件了。每行数据应包含：

- 答案标签控件：显示调查问卷的答案文本。
- 选票标签控件：显示答案的选票，它既显示为一个整数，也显示占所有选票的百分比。
- 选票进度条控件：这是该答案的选票数的图形表示。

这个窗体要包含相当多的控件，它们应是分组的，记住这一点是很重要的。例如，在获取选票数据时，必须用一种方式迭代所有的选票标签，把它们的 Text 属性设置为选票数。为了使

这个过程容易完成，可以用一个数组保存已创建的所有标签控件，用一个数组保存答案标签，一个数组保存选票标签，一个数组保存选票进度条。

Form1.cs 中的下一行代码创建了一个存储标签控件的数组，显示调查问卷中的答案(“C#”，“C++”等)。AnswerControls 数组在类的级别上定义，如下所示：

```
private Label[] answerControls
```

在创建数组时，利用 myAnswerCount 字段保存答案的总数：

```
answerControls = new Label[myAnswerCount];
```

有了数组后，就可以开始循环了。每次循环时，都将创建一个标签控件，设置它的位置和几个其他属性，在数组中保存它的一个引用，以便以后访问它。首先 for 语句再次使用 myAnswerCount 字段：

```
for(int i=1;i<=myAnswerCount;i++)
{
```

使用 new 关键字创建一个新的 Windows Forms 标签：

```
    Label answerLabel = new Label();
```

创建一个 Location 对象，用于指定显示标签控件位置。注意指定标签的位置是距离左边 8 个像素，距离下边 37 个像素，每个标签距离窗体的下边界还有 20 个像素(每个控件的周围都有一定的空白，大约 20 个像素)。我们创建的第一个标签是顶部的那个标签，把它向下移动：

```
    answerLabel.Location = new System.Drawing.Point(8, 37+(20*i));
```

创建一个 Size 对象，使用该对象把标签设置为 200 个像素宽，16 个像素高。选择这些值的原因是，它们创建出的页面布局能满足我们的要求，可以自由修改这些值，看看会发生什么：

```
    answerLabel.Size = new System.Drawing.Size(200, 16);
```

把标签的背景色设置为灰白：

```
    answerLabel.BackColor = Color.AntiqueWhite;
```

在这行代码中，将标签控件的一个引用存储在数组中。记住，这并没有做任何工作，而是提供了在设置答案标签的内容时，迭代所有答案标签的一种简单方式：

```
    answerControls[i-1] = answerLabel;
```

最后，在窗体的 Controls 属性上，使用 Add()方法把标签添加到窗体上：

```
    Controls.Add(answerLabel);
}
```

如果建立工程，并运行应用程序，就会自动生成一组标签。注意在后台应用程序与 Web 服务器通信，并获取请求答案的总数。在这个范例中，Windows 应用程序(PollD)和 Web 服务(在

PollID 中)都在同一台机器上，它们也可以移动到其他机器上，而不需要进行很多改变，如图 26-23 所示。

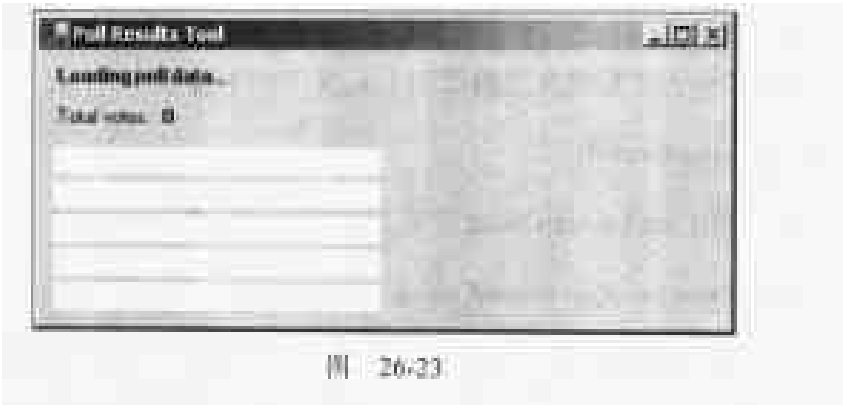


图 26-23

下面继续增加显示选票的标签控件，增加图形进度条，它也显示选票，但显示方式更加形象。下面的代码是 Form1.cs 中构造函数的下一部分，它们几乎与刚才添加的代码完全相同，但标签距离右边更远一些(220 像素，而不是 8)。在数组 voteControls 中添加这些控件的引用，这个数组是在类的级别上定义的，如下所示：

```
private Label[] voteControls.
```

回到构造函数代码，添加更多的标签：

```
voteControls = new Label[myAnswerCount];
for(int i=1;i<=myAnswerCount;i++)
{
    Label votesLabel = new Label();
    votesLabel.Location = new System.Drawing.Point(220, 37+(20*i));
    votesLabel.Size = new System.Drawing.Size(80, 16);
    votesLabel.BackColor = Color.AntiqueWhite;
    voteControls[i-1] = votesLabel;
    Controls.Add(votesLabel);
}
```

添加了这些代码后，窗体上就有两列控件了，其结果如果 26-24 所示。

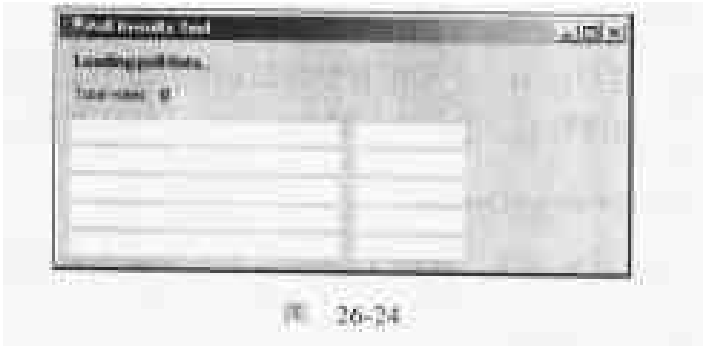


图 26-24

要添加图形指示器，可使用 ProgressBar 控件。首先创建一个数组，用于循环这些控件，以便以后添加它们的内容。VoteBarControls 变量在类的作用域中声明：

```
private ProgressBar[] voteBarControls.
```

在窗体的构造函数中，添加如下代码：

```
voteBarControls = new ProgressBar[myAnswerCount];
```

对每个答案循环一次，与两个标签控件一样：

```
for(int i=1;i<=myAnswerCount;i++)
{
```

给窗体创建一个 ProgressBar 控件：

```
    ProgressBar progressBar = new ProgressBar();
```

创建一个 Point 对象，用于指定窗体上 ProgressBar 的位置。每个进度条都距离左边 310 个像素，距离下边 37 个像素，每个控件周围空白 20 个像素。它们与两列标签对齐：

```
    progressBar.Location = new System.Drawing.Point(310, 37+(20*i));
```

把进度条的最小值设置为 0。以后要把其最大值设置为任一个答案接收到的最大选票数：

```
    progressBar.Minimum = 0;
```

把进度条的宽度设置为 103 个像素，高度设置为 16 个像素：

```
    progressBar.Size = new System.Drawing.Size(103, 16);
```

最后，把对控件的引用添加到进度条数组上，再把进度条添加到窗体上：

```
    voteBarControls[i-1] = progressBar;
    Controls.Add(progressBar);
}
```

如果建立并运行 Windows 应用程序，窗体上就有 3 列控件了。前两列是标签控件，第三列是一组进度条。注意每一列都可以使用我们构造的数组来访问，这样就很容易把数据添加到下面将创建的控件上，如图 26-25 所示。

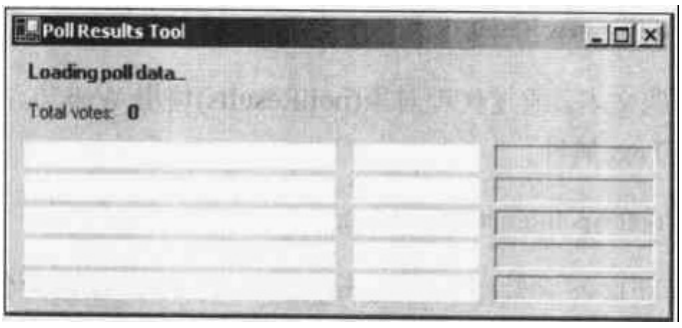


图　26-25

为了确保窗体的大小总是正好能容纳它所包含的控件，还可以添加下面两行设置窗体的宽度和高度的代码：

```
this.Height = 87 + (20*myAnswerCount);
```

```
this.Width = 428;
```

有了这样的窗体后，就需要用数据填充它。但是，我们并不是只填充一次数据，而是一直用最新的数据更新它。为此，建立一个工具，它可以每隔几秒种就从 Web 站点上请求数据。.NET 包含一个对象 Timer，可以用于执行这个任务。在使用对象 Timer 时，要指定一个时间段，和一个要调用的方法。每次到达这个时间段时，就调用这个方法，所以要编写一些代码，让 Timer 对象完成该任务。我们编写的方法叫作 GetPollData()，它可以从 Web 服务中收集数据，并把它们插入到窗体控件上。

首先，需要在代码的顶部添加如下语句：

```
using System.Timers;
```

接着使用 New 关键字实例化 Timer 对象：

```
System.Timers.Timer aTimer = new System.Timers.Timer();
```

然后给它添加一个事件处理程序，当到达这个时间间隔时，该对象知道要调用哪个方法。在这个处理程序中，告诉它应调用 GetPollData()方法，该方法稍后介绍。

```
aTimer.Elapsed += new ElapsedEventHandler(GetPollData);
```

接着设置计时器的时间间隔，并激活它。在本例中，其时间间隔设置为 5 秒，也许在现实情况下这个值应是可以配置的：

```
aTimer.Interval=5000;
aTimer.Enabled=true;
```

现在窗体上有一些控件和一个计时器，它每隔 5 秒就调用一次 GetPollData()方法，填充窗体。在 Form1 中，GetPollData()方法首先必须实例化 Web 服务代理类中的一个对象，这样才能使用 Web 方法：

```
public void GetPollData(object source, ElapsedEventArgs e)
{
    Results pollResults = new Results();
```

要设置窗体上的问题文本，通过代理对象(pollResults)调用 Web 方法 Question，使用它设置 questionLabel 控件上的 Text 属性。

```
    questionLabel.Text = pollResults.Question();
```

接下来的两行代码获取答案数组，并把 myAnswerCount 设置为包含调查问卷中的答案总数：

```
    string[] pollAnswers = pollResults.Answers();
    int myAnswerCount = pollAnswers.Length-1;
```

我们必须计算两个 Web 方法没有提供的值。第一个值是选票的总数。第二个值是某个答案

接收到的最大选票数(这个值用于设置进度条的最大值)。要创建这两个值，使用 Web 方法 Votes() 获取选票数的数组，并循环它，更新 totalVotes 和 highestVote 变量：

```
int totalVotes = 0;
int highestVote = 0;
int[] pollVotes = pollResults.Votes();
for(int i=1;i<=myAnswerCount;i++)
{
    totalVotes = totalVotes + pollVotes[i];
    if (pollVotes[i] > highestVote)
        highestVote = pollVotes[i];
}
```

下面计算选票的百分比，所以需要一个整数变量存储这个值：

```
int percentageVote;
```

在下面的几行代码中，循环某个答案，把这些值赋给窗体上的 3 列控件，首先是答案文本，接着是选票结果(带有百分比)，最后是进度条。用一个 For 循环开始：

```
for(int i=1;i<=myAnswerCount;i++)
{
```

在窗体的第一列上，用标签控件的 Text 属性首先设置答案文本。使用前面创建的答案标签数组访问标签控件。注意因为控件数组是从 0 开始索引的，所以要从索引中减去 1，但答案数组不是基于 0 的：

```
answerControls[i-1].Text = pollAnswers[i];
```

要设置选票字段，可使用类似的过程，但首先需要计算百分比。首先用选票数除以选票总数。注意结果存储在一个浮点数变量中：

```
float voteFraction = ((float)pollVotes[i]/(float)totalVotes);
```

要创建一个百分比值，把它乘以 100：

```
percentageVote = (int)(100 * voteFraction);
```

要增加选票数，把选票数的一个字符串副本赋给选票标签控件数组中控件的 Text 属性：

```
voteControls[i-1].Text = pollVotes[i].ToString()+"   ";
```

接着在文本的后面放置该百分比值，并用方括号括起来，便于阅读，当然要添加%：

```
voteControls[i-1].Text += "(" + percentageVote.ToString() + "%)";
```

要设置进度条的位置，需要设置它可以达到的最大值和实际值。使用答案接收到的最大选票数来设置进度条的最大值，再用 Value 属性设置实际的选票数：

```
            voteBarControls[i-1].Maximum = highestVote;
            voteBarControls[i-1].Value = pollVotes[i];
        }
```

最后，设置 totalVotesLabel 控件的 Text 属性，在窗体上显示选票总数：

```
        totalVotesLabel.Text = totalVotes.ToString();
    }
```

如果建立工程，并运行应用程序，就会显示一个空白的窗体，但很快计时器就会发挥作用，所有的字段都将用 Web 服务器上的调查问卷结果来填充，如图 26-26 所示。

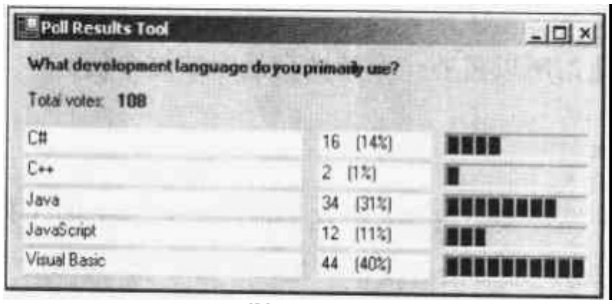


图　26-26

如果运行了控制台应用程序，或者使用 Web 应用程序提交了选票。现在就可以看到，在 5 秒钟内，Poll Results Tool 会正确显示更新了的选票数。

26.3.3　一人一张选票

如果要限制一个人可以在调查问卷中投票的次数，就必须能确定以前的投票情况。投票的主要界面是一个 Web 页，所以可以使用 cookies，它是存储像这样的信息的理想方式。记住 cookie 是一小段信息，当访问者请求访问 Web 站点时，他们的浏览器就会在其硬盘上存储这些信息。每次他们再回到这个 Web 站点时，就会把 cookie 发送回来。如果设置一个 cookie，说明他们已经在调查问卷中投票了，我们就可以忽略他们将来的投票。这里使用 cookie 的一个缺点是如果浏览器关闭了 cookie，这个访问者就可以一次又一次地投票。

首先对 Web 应用程序工程(PollW) 中的 default.aspx 文件进行一个小小的改动，以查看 cookie，并观察代码所做的工作。在代码中标记了的改动(Trace="true")表示，在浏览器中打开 default.aspx 时，会看到几页调试信息，包括页面接收到的 cookie 信息。此时，以前使用过 ASP 3.0 的用户就会很惊讶地发现：

```
<%@ Page language="c#" Trace="true" ... Inherits="PollW.WebForm1" %>
```

如果现在在浏览器中打开 default.aspx，就会看到其中只有一个 cookie，ASP.NET 使用这个 cookie 维护会话信息，如图 26-27 所示。

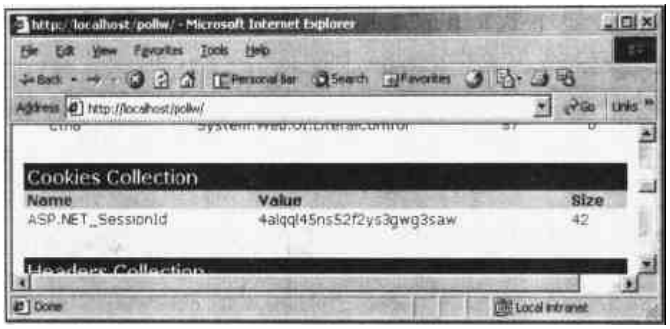


图　26-27

看到 cookies 后，就可以编写保存 cookie 的代码了，看看当执行它时会发生什么情况。下面把注意力转向 Poll 类。当一个人在 Vote()方法的最后注册了选票后，就在他的机器上保存 cookie。记住在该方法的最后，我们把调查问卷的数据保存回 XML 文件，并关闭流：

```
myDataSet.WriteXml(myStream,XmlWriteMode.IgnoreSchema);
myStream.Close();
```

要保存 cookie，在这里再添加一些代码。首先，检查一下该方法是否运行在 HttpContext 中，这种检查可确保 cookie 代码仅在方法从 ASP.NET 页面上调用时执行，在从其他应用程序(例如 Windows Forms 应用程序或控制台应用程序)中使用该方法时，就忽略 cookie 代码。

要使用 HttpContext 类，需要插入一个对 System.Web 名称空间的引用：

```
using System.Web;
```

下面是写入 XML 流之后插入的代码：

```
if (HttpContext.Current != null)
{
```

要创建一个 cookie，需使用 HttpCookie 类实例化一个对象。每个 cookie 都有一个名称和一个值。名称总是相同的(WroxPoll)，但值在各个调查问卷中都是不同的。因为我们要确保在发布一个新的调查问卷时，人们都可以成功地投票：

```
    HttpCookie myCookie = new HttpCookie("WroxPoll");
```

把调查问卷的值设置为对当前调查问卷、调查问卷的问题来说是惟一的值：

```
    myCookie.Value = (myQuestion);
```

最后，使用 HttpResponse 对象获得正确的 HTTP 响应，访问它的 Response 属性，该属性表示返回用户浏览器的数据。使用 Cookies 属性上的 Add()方法，请求把新的 cookie 添加到 HTTP 流中，使用户的浏览器可以把它保存到磁盘上：

```
    HttpContext.Current.Response.Cookies.Add(myCookie);
}
```

如果重新建立工程，打开 default.aspx，注册一个选票，再重新加载 default.aspx，就会发现

跟踪输出中的 cookies 部分将显示我们的 cookie，其结果如图 26-28 所示。

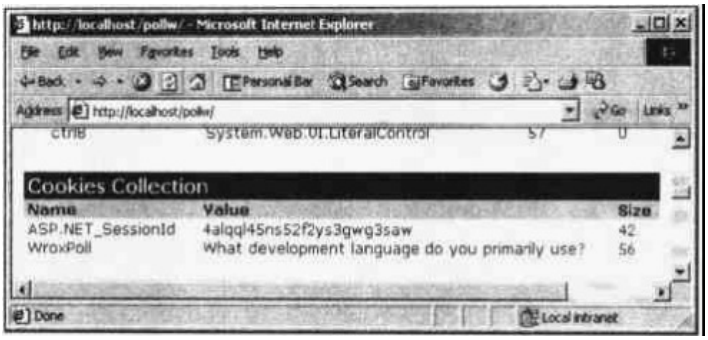


图　26-28

在用户的机器上成功保存了 cookie 中的信息后，就可以确定访问者是否已经在这个调查问卷中投票了。为防止他们再次投票，应在 Vote()方法的开头添加如下代码：

```
if (HttpContext.Current.Request.Cookies["WroxPoll"] != null)
{
    if (HttpContext.Current.Request.Cookies["WroxPoll"].Value == myQuestion
    {
        throw (new Exception("You have already voted in this poll"));
    }
}
```

首先检查 cookie 是否存在，一旦确定它不是空的，就应检查其值是否匹配当前进行的调查问卷中的问题。

在商用应用程序中，可以以更友好的方式处理对用户投票的拒绝，但对于这个范例，如果 cookie 的值匹配于当前调查问卷中的问题文本，那么这个 cookie 就会抛出一个异常，如图 26-29 所示。再次注意在使用 HttpContext 之前应检查它是否为空，仅可以确保不检查非 ASP.NET 应用程序中的 cookie。

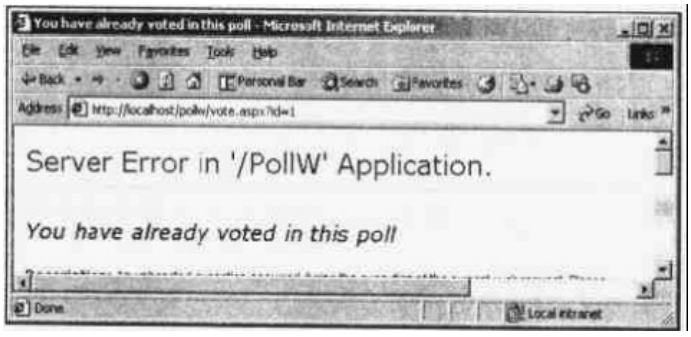


图　26-29

如果进入调查问卷的 XML 文件(c:\polls.xml)，并修改其中的问题，就可以提交另一个选票。这是因为我们检查选票是否已提交，是建立在调查问卷中问题的基础上。改变问题，cookie 值和问题名称不再匹配，所以选票就会被接收。如果注册了选票，Windows Forms 应用程序就会显示更新时的新选票总数(换言之，在少于 5 秒的时间内)。

26.3.4　调查问卷控件

.NET 比以前的技术先进的一个方面是它很容易重用已编写好的软件。本节的任务是打包调查问卷 Web 站点，这样就可以只使用几行代码，把它插入到任何 ASP.NET 页面上。此时，把该调查问卷和另一个 Web 页集成起来需要做许多工作，因为我们必须从.aspx 文件和后台编码文件.aspx.cs 中复制相当多的内容。另一个新增的功能是可以编写控件，让它显示调查问卷，如果用户已经投票了，就要把它编写为只显示答案，不显示投票页面。

要打包调查问卷，可以创建一个定制控件，该控件在继承了.NET Framework 的 Control 的类中执行。首先在 Visual Studio.NET 中右击 Web 应用工程(PollW)，选择 Add，再选择 Add New Item。在对话框中，选择 Web Custom Control，在名称字段中键入 PollControl.cs，其结果如图 26-30 所示。

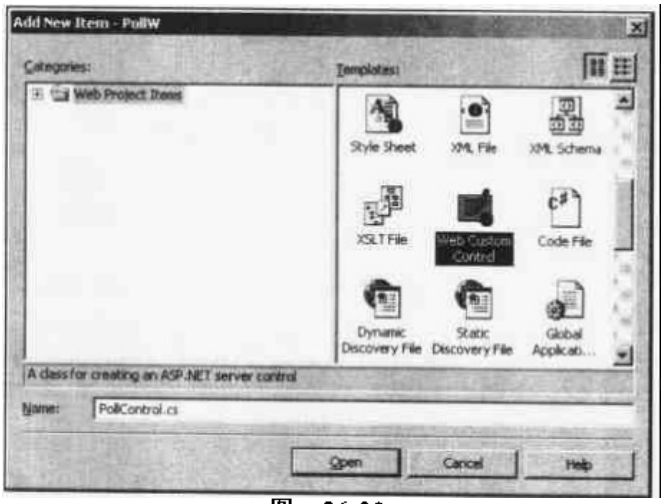


图　26-30

尽管 Visual Studio.NET 会生成几行起始代码(执行控件上的 Text 属性)，但不需要这么做，所以把类删除，得到一个更简单的开头：

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

namespace PollW
{
    /// <summary>
    /// Summary description for PollControl.
    /// </summary>
    public class PollControl : WebControl
    {
        protected override void Render(HtmlTextWriter output)
        {
            output.Write("This is my Poll Control");
        }
```

```
    }
}
```

注意实际上只需创建一个继承了 System.Web.UI.WebControls.WebControl 的类。有了这个类，就可以执行 Render()方法的重写代码，我们在这里生成要输出的 HTML。也可以编写一个控件，把其他控件组合在一起来创建这个控件本身，但在这个范例中，我们要生成 HTML。

现在有了一个简单的定制控件，下面把它插入到一个页面上，看看需要做什么。首先，在工程上添加一个新页面 default2.aspx，把这个页面用作定制控件的容器。为此，应使用 Project 菜单中的 Add Web Form，如图 26-31 所示。

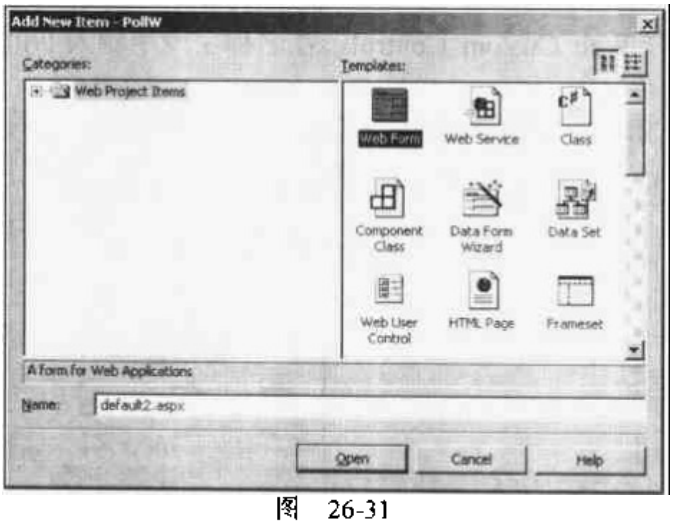


图　26-31

删除 default2.aspx 中的内容，使程序变得简单一些，这样就可以看出将发生的情况：

```
<%@ Page language="c#" Codebehind="default2.aspx.cs" ..... %>
<html>
    <head>
    </head>
    <body>
    </body>
</html>
```

要在页面上插入调查问卷控件，应添加下面两行突出显示的代码：

```
<%@ Page language="c#" Codebehind="default2.aspx.cs" ..... %>
<%@ Register TagPrefix="Wrox" Namespace="PollW" Assembly="PollW" %>
<html>
    <head>
    </head>
    <body>
        <Wrox:PollControl runat="server"></Wrox:PollControl>
```

```
    </body>
</html>
```

添加的第一行代码声明了一个 TagPrefix，叫作 Wrox。这表示如果 ASP.NET 在页面上看到以<Wrox:开头的任何服务器端控件，就会搜索 PollW 装配件中的 PollW 名称空间，找到一个匹配名称的控件。在本例中，指定<Wrox:PollControl>，让 ASP.NET 搜索 PollW 名称空间，找到一个类 PollControl。前面添加的第二行代码是一个服务器端控件，我们在这一行中把调查问卷插入到页面上。注意这里包含了 runat 属性，如果在浏览器中打开 default2.aspx，就可以看到 PollControl Render()方法输出的文本，如图 26-32 所示。

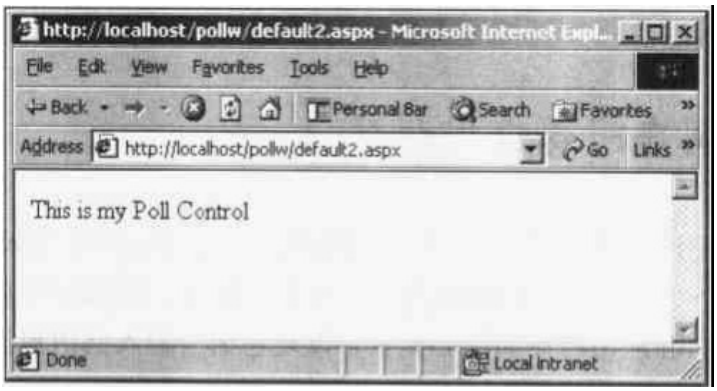


图　26-32

成功插入定制控件后，就要在 PollControl.cs 的 Render()方法中添加一些代码，让控件显示调查问卷。这里的代码几乎和后代编码文件 default.aspx.cs 中的代码完全相同。记住文件需要在其开头加上 using Polling 语句：

```
protected override void Render(HtmlTextWriter output)
{
    string Html = "";
    Poll aPoll = Poll.Current();
    Html += aPoll.Question + "<br /><br />";
    for(int i=1;i<=aPoll.AnswerCount;i++)
    {
        Html += "<a href=\"vote.aspx?id="+i+"\">Vote</a>";
        Html += " " + aPoll.Answers[i] + "<br />";
    }
    output.Write(Html);
}
```

如果重新建立工程，打开 default2.aspx，就会看到与 default.aspx 相同的内容，如图 26-33 所示。但这个版本更容易重用。要在站点上使用它，只需确保 vote.aspx 是可以访问的，并确保可以得到包含 Poll 类和 PollControl 类的装配件，再在 ASP.NET 页面上添加两行代码。实际上，我们只是封装了功能，并把它称为 Web 控件。这是把各种功能组合到一起，以易于重用它们的一种方式。

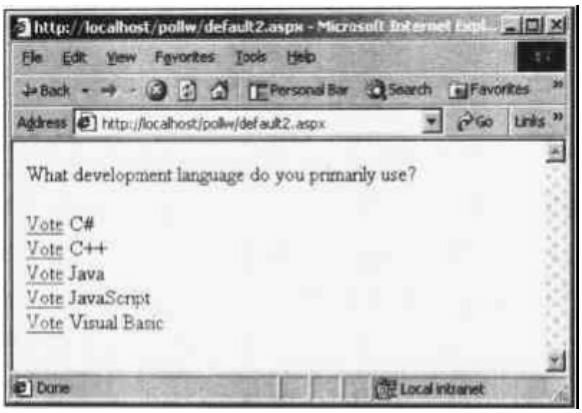


图　26-33

26.4　三层和调查问卷应用程序

前面已经把一个通过控制台测试程序发挥功用的应用程序、一个 Web 服务、一个 Windows Forms 应用程序和一个 ASP.NET Internet 接口组合在一起了。图 26-34 总结了如何把这些程序组合在一起，显示了 Poll 应用程序的核心元素，以及它如何融入到由数据、事务逻辑和表示层组成的 3 层结构中的。

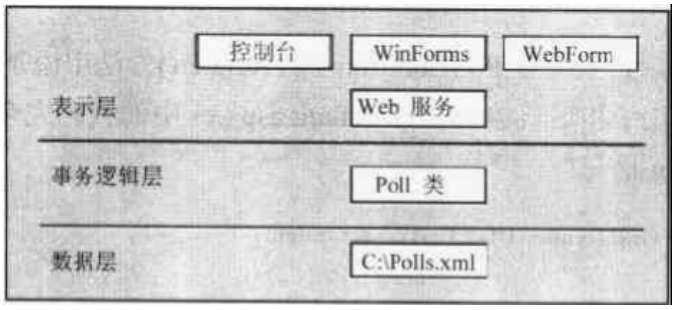


图　26-34

注意:

尽管 Windows Forms 应用程序通过 Web 服务与 Poll 类通信，但是如果它运行在与 Poll 类相同的机器(即 Web 服务器)上，也就不必这么做了。

26.5　小结

这个案例分析说明了如何把本书介绍的 C#和.NET 理论应用到真实环境中，开发一个能通过几个用户界面访问的应用程序。该案例分析以建立应用程序第一个版本的方式模拟真实环境，再把第二个版本添加进去，增加功能。在商用环境下，并不编写软件，它在不断地发展，其中有一些内容是我们要在这里阐述的。

快速回忆一下，这个案例分析介绍了如下技术和技巧：

- 事务对象(Poll 类)
- 抛出异常
- ASP.NET 定制控件
- ASP.NET 服务器端控件
- 类的静态方法
- XML 和 ADO.NET 数据集
- Web 服务和 Web 方法
- System.Timer
- HTTP Cookies
- HttpContext 类
- 支持多个接口的表示层
- 控制台应用程序
- Windows Forms 应用程序
- 3 层开发
- 面向对象的设计
- 软件的测试

本章很便于阅读，其中的信息可以帮助您用 C#创建自己的应用程序。.NET、3 层体系结构和面向对象的设计方法，是开发软件程序的一个非常强大的平台，C#可能是.NET 中最令人激动的语言。

第27章 案例分析2——在线时事通讯管理器

第 26 章介绍了如何使用 C#编程，如何在 Windows 应用程序和简单的 ASP.NET 应用程序中使用 C#。本章将介绍如何把前面学习到的知识组合在一起，建立可用于实际.NETWeb 站点的 Web 应用程序。

下面列出了阅读本案例分析后学习到或更深入理解所学内容的一些论题：

- 如何用 ASP.NET 设计一个完整的 Web 应用程序
- 如何使用一些 ASP.NET 服务器端控件生成内容丰富的用户界面
- 使用后台编码把代码和用户界面区分开来
- 编写一个 Web 配置文件，存储应用程序的定制设置，执行基本的安全功能
- 使用 C#的面向对象功能，创建和使用封装了应用程序的事务逻辑的类
- 在文件、邮件和数据库处理中使用.NET Framework 的一些类
- 用 OLE DB 托管的提供者和 ADO.NET 类读写数据库
- 发布已编译的 ASP.NET 应用程序，通过 IIS 运行它

27.1 应用程序概览

每个 Web 制作人员都知道开发一个成功的 Web 站点需要的关键元素，其中一些如下：

- 好的内容
- 组织 (耐航性)
- 有魅力的设计
- 与用户的交互

成功的 Web 站点需要包含新鲜的、定期更新的有趣内容。电子邮件列表是使订阅者通报新内容的有效方式，提示访问者，激励他定期访问该站点。

管理这样一个列表是很困难的。管理员需要维护订阅者的列表，处理新的订阅请求和取消订阅的请求，跟踪所有已发送的信息。

本章介绍的应用程序有助于自动管理这些邮件列表。这是一个在线的时事通讯管理器，可以自动执行订阅及新闻管理有关的大多数任务，让管理员更轻松地完成其余的任务。该程序：

- 管理多个邮件列表——管理员可以添加、删除或修改列表及其属性。
- 管理每个列表的订阅者，可以设置和修改它们的名称和个人信息。
- 以纯文本或 HTML 格式给订阅者发送电子邮件。所有的电子邮件都可以用用户的名称和地址自动定制，可以包含一个取消订阅的链接，以及列表的其他信息(例如名称和描述)。
- 管理传送给每个列表的旧时事通讯的归档文件，允许管理员读取和复制旧信息。

• 创建一个 HTML 窗体来获取用户的数据，而无需管理员编写任何代码。这个功能的工作方式类似于一个向导——管理员选择他们创建窗体时所需要的列表，并选择几个其他选项，向导就会为他们建立代码。

• 设置选项，例如发送者的姓名和地址，自动添加到每个信息中的签名、默认的标题，以及订阅和取消订阅的信息。

• 能修改应用程序的布局(字体颜色和大小)，无需修改 HTML 代码或上传新文件。

这个解决方案的最大优点是不再需要手动管理订阅者组了，应用程序会在一个数据库中自动添加或删除它们。第二个优点是管理员不论是在世界的哪一处，都可以在线监视 Web 站点订阅者，联系它们，甚至修改 Web 站点的布局，他们只需访问一个带有 Web 浏览器和 Internet 连接的计算机即可。

提示：

这个案例分析还假定使用 Visual Studio.NET 开发应用程序。如果不是，下载的代码包含了如何在命令行上编译这个应用程序的指令。

27.2　应用程序的设计

由数据库驱动的应用程序通常可分解为表示层、事务逻辑层和数据层。这个设计便于维护、并有更好的可伸缩性，因为可以在不同的服务器上部署该应用程序。即使下面将开发的应用程序不需要很高的可伸缩性，也常常采用这种方式进行开发。图 27-1 是该应用程序结构的图形表示。

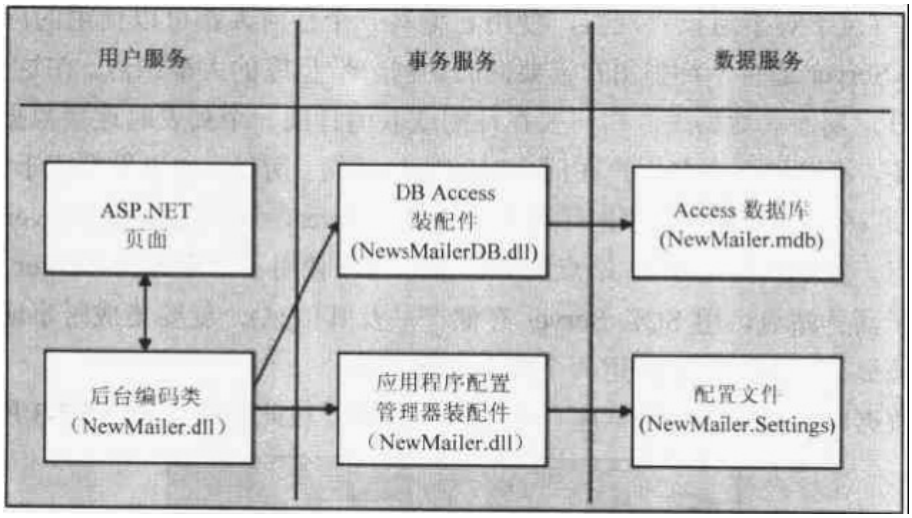


图　27-1

可以看出，表示层(用户服务)是一组 ASP.NET 页面及其后台编码文件(编译到一个装配件中)。事务服务层包含另外两个装配件，可以方便地访问数据。本章将详细论述这些层和装配件的执行情况。

27.3　数据和事务对象

首先讨论数据和事务层。图 27-2 显示了事务层的两个装配件，以及它们包含的公共类。

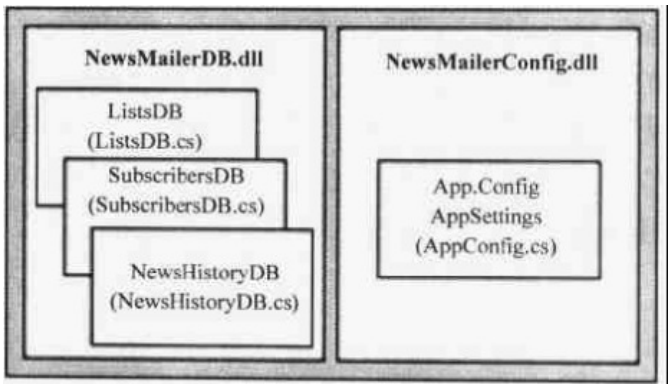


图　27-2

NewsMailerDB.dll 装配件提供了对数据库的访问，其中存储了邮件列表的名称、订阅者和旧信息。NewsMailerConfig.dll 装配件提供了对 XML 设置文件的访问，在该文件中，可以读写所有的应用程序选项。下一节将介绍这些事务库和它们所提供的可访问的数据。

27.3.1　数据库设计

整个应用程序是数据库驱动的，所以先讨论使用的数据库和设计。我们决定使用 Access 数据库，而不是 SQL Server，这有几个原因。Access 更易于配置，可以在大多数 Web 存储规划中使用，因为它比较便宜，而 SQL Server 应用于高级(较贵)Web 存储的软件包中。另外，小的 Web 站点可以得益于时事通讯管理器，使用它编写一个任何人都可以使用的应用程序会比较好。而且，SQL Server 适合于许多用户需要同时访问该数据库的大型站点。在这个应用程序中，只有管理员可以定期连接数据库。用户要在订阅或取消订阅一个列表时连接数据库，但这个操作执行得非常快，不可能有许多用户在同一时间进行订阅。所以，如果没有成千上万个订阅者，使用 Access 就足够了。但是，把数据库移植到 SQL Server，以利用 SQL Server 的强大功能和速度有几个缺点。显然，如果 Web 站点的其他部分已经使用了一个 SQL Server 数据库(例如正在开发一个电子商务站点，用 SQL Server 存储产品及其信息)，就要集成时事通讯管理器，把它的表添加到该数据库中，避免使用两个独立的数据库。

现在看看数据库的设计。该数据库有 4 个表，这些表彼此链接。如图 27-3 所示。

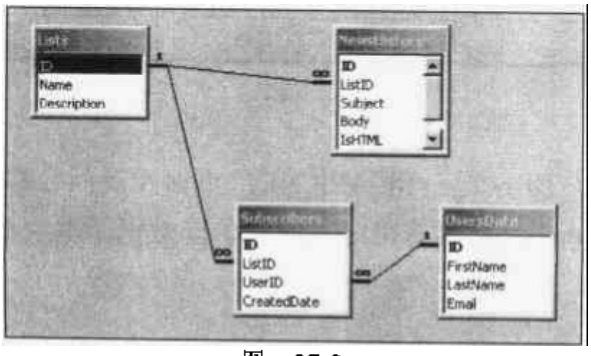


图　27-3

下面就开始描述这些表和它们的关系。

- Lists 表

Lists 表用于存储每个可用邮件列表的信息，它的结构如表 27-1 所示。

表　27-1

列　名	类　型	大　小	可否为空	描　述
ID	AutoNumber		否	邮件列表的惟一 ID
Name	Text	50	否	邮件列表的名称
Description	Text	250	是	邮件列表的可选描述

- UsersData 表

UsersData 表存储了用户的个人信息，其结构如表 27-2 所示。

表　27-2

列　名	类　型	大　小	可否为空	描　述
ID	AutoNumber		否	用户的惟一 ID
FirstName	Text	50	否	用户的名
LastName	Text	50	否	用户的姓
Email	Text	50	否	用户的电子邮件地址

这个表没有存储用户订阅的列表，因为一个用户可以订阅几个列表，最好能避免重复内容。如果要存储用户的信息和他们订阅的所有列表，就需要避免重复。把用户的数据放在一个单独的表中，就意味着这个空间没有浪费，如果要更新用户的信息(例如他们的地址有变化)，就只需更新一个他们订阅许多列表的记录。

- Subscribers 表

Subscribers 表基本上提供了列表和用户之间的链接。它还记录了用户第一次订阅该列表的日期，它的结构如表 27-3 所示。

表　27-3

列　名	类　型	大　小	可否为空	描　述
ID	AutoNumber		否	订阅的惟一 ID
ListID	Integer		否	用户订阅的列表 ID
UserID	Integer		否	用户的 ID，它应匹配于 UserData 表中的一个 ID
CreatedDate	Date		否	创建这个订阅的日期

- NewsHistory 表

NewsHistory 表用于存储给每个邮件列表的订阅者发送的旧邮件，其结构如表 27-4 所示。

表 27-4

列　名	类　型	大　小	可否为空	描述
ID	Auto Number		否	信息的惟一ID
ListID	Text		否	邮件列表的名称，信息就发送给该邮件列表
Subject	Text	200	否	信息的标题
Body	Memo		是	信息体
IsHTML	Boolean		否	如果为 true，就表示信息以 HTML 格式发送
SentDate	Date		否	信息发送的时间

表之间的关系

如果回过头来看看表示各表之间关系的图 27-3，就可以看出，表 Lists 和 Subscribers 之间的关系以及 Lists 和 NewsHistory 之间的关系是一对多的，因为一个列表可以有许多订阅者和许多旧信息。同样，一个用户可以订阅多个列表，所以 UsersData 和 Subscribers 之间的关系也是一对多。

使用 Access 建立这些关系是很简单的。下面看看如何建立。打开 NewsMailer.mdb 文件，选择 Tools | Relations，显示一个对话窗口，它在前景中列出了可用的 4 个表，在背景中列出了一个空窗口：从 4 个对话框中选择每个表，把它们添加到背景窗口中。最后关闭对话框。现在背景中的窗口显示了 4 个表，此时就可以创建它们的关系了。要把表 Lists 与 Subscribers 链接起来，单击 Subscribers 表的 ListID 字段，把它拖放到 Lists 表的 ID 字段上，显示如图 27-4 所示的对话框。

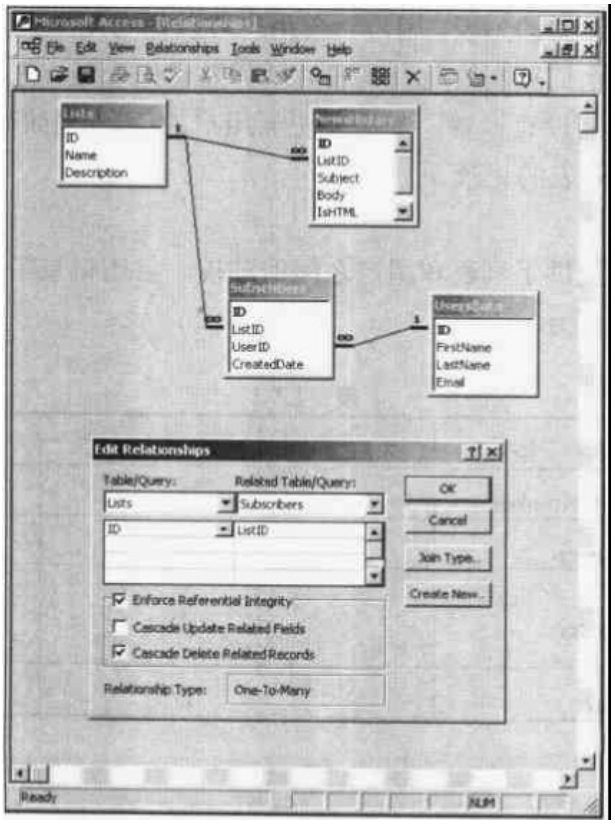


图 27-4

在这个对话框中，可以在两个下拉框中选择要链接的字段。如果执行了拖放操作，选择了正确的字段，底部的描述就将是“一对多”，表示一个邮件列表对应多个订阅者。如果不执行拖放操作，还可以双击主窗口(显示表的窗口)，得到同样的对话框，但其中没有所选择的要链接的字段。

在创建关系前，还有一点要记住：激活 Cascade Delete，应用 Referential Integrity，如图 27-4 所示。这样，当删除一个列表时，链接的订阅者也会自动删除。还有一个 Cascade Update 选项，它可以确保如果 Lists 表中的 ID 有变化时，Subscribers 表中的所有 ListID 都会更新，我们不需要选择它，因为 ID 字段是一个 AutoNumber 字段，在添加记录时不能修改。

现在以同样的方式创建 Lists-NewsHistory 和 Subscribers-UsersData 的关系，根据图 27-4 了解使用哪些字段进行连接。

27.3.2　访问数据库的事务类

处理数据库的代码从逻辑上和物理上可分为几个类。这些类都编译到一个类库 NewsMailerDB 中。其余的代码是应用程序的后台编码类，它构成了一个独立的装配件。这种方式是最好的，因为如果需要修改 DB 设计中的某些内容，或者要升级到 SQL Server，就只需修改和重新编译事务装配件。这种方式也比把所有的内容(事务和后台编码类)都编译到一个装配件中更好，因为可能要在其他客户(如 Windows 应用程序前端)中使用事务装配件。

对于这个应用程序，采用这种方式没有什么大的优势，应用程序的设计相当简单，没有许多大文件要编译，也没有其他客户使用事务类。即使把所有的内容都编译到一个 DLL 中，也不会出现严重的问题。我们将使用独立的 DLL 说明它的简单性，帮助读者养成良好的习惯！

如果从头开始开发一个工程，就应以面向对象的方式来设计工程，通常是把事务类和用户界面代码分隔开，把它们编译到不同的库中。把代码分成几个装配件也可以在以后进行，因为这主要是一个编译问题。但重要的是创建只负责管理 DB 的类，把所有的 UI 代码放在其他类中。

1. 创建工程

如果使用 VS.NET 开发应用程序，下面就开发一个类型为 C# | Class Library 的新工程，命名为 NewsMailerDB。下一节将讨论如何为这个装配件编写类文件。必须通过 Project | Add New Class 命令为每个类创建一个新文件，或者通过 Project | Add Existing Item，从 Wrox 站点的下载数据中导入可用的文件。

提示：

这些事务类的代码在文本中也有。完整的列表参见下载的代码。

2. ListsDB 类

ListsDB 类提供了处理现有邮件列表和添加新列表的所有函数，这个类汇总在表 27-5 中。

GetLists()方法返回一个数据集，其中包含 Lists 表中的所有内容，包括所有的可用列表和它们的属性。另一方面，GetDetails()返回一个由 ID 指定的列表，并作为参数传递。

表 27-5

ListsDB 类

方　法	描　述
public ListsDB(string ConnString)	类的构造函数，其参数是一个连接字符串，在设置文件中定义(后面讨论)
public DataSet GetLists()	获取一个包含所有可用于新闻列表的数据集
public DataRow GetDetails(int ListID)	用指定的 ID 获得列表的细节
public bool Add(string ListName, string ListDescr)	用指定的名称和描述添加一个新列表
public bool Delete(int ListID)	删除指定的列表
bool Update(int ListID, string ListName, string ListDescr)	更新指定列表的 Name 和 Description 属性

在 Delete()方法中，用 ID 指定一个记录，作为参数传送给该方法，然后该方法删除这个记录。Update() 和 Add()方法也是一样的，但参数包含要修改或添加的数据。

3. SubscribersDB 类

SubscribersDB 类提供了对 Subscribers 和 UsersData 表的访问。它的构建方式与 ListsDB 类相似，位于文件 SubscribersDB.cs 中。这个类的方法如表 27-6 所示。

表 27-6

SubscribersDB 类

方　法	描　述
public SubscribersDB(string ConnString)	类的构造函数，其参数是一个连接字符串，在设置文件中定义
public DataSet GetSubscribers(int ListID)	获取指定列表的订阅者列表
public DataRow GetDetails(int SubscrID)	获得指定订阅者的信息
public bool Add(int ListID, string FirstName, string LastName, string Email)	添加新订阅者
public bool Delete(int SubscrID)	删除一个订阅者
public bool Update(int SubscrID, string FirstName, string LastName, string Email)	更新订阅者的个人信息
public int GetUserID(int SubscrID)	用指定的订阅 ID 返回用户 ID
public int GetUserID(string Email)	用指定的电子邮件地址返回用户 ID
public int GetSubscrID(int ListID, string Email)	返回特定列表和特定用户的订阅者 ID，用惟一的电子邮件地址标识

这个类比 ListsDB 更复杂，因为它必须处理两个表，以及这两个表之间的关系。Get

Subscribers()函数返回参数 ID 指定的列表中的所有订阅者，它使用一个包含 INNER JOIN 的 SQL 命令从 UserData 和 Subscribers 表中获取信息：

```
public DataSet GetSubscribers(int ListID)
{
    string sqlCmd = "SELECT FirstName, LastName, Email, ";
    sqlCmd += "ListID, UserID, CreatedDate, Subscribers.ID FROM UsersData ";
    sqlCmd += "INNER JOIN Subscribers ON UsersData.ID=Subscribers.UserID ";
    sqlCmd += "WHERE Subscribers.ListID = " + ListID;

    DataSet subscribers = new DataSet();
    OleDbConnection myConn = new OleDbConnection(connString);
    OleDbDataAdapter myDAdp = new OleDbDataAdapter(sqlCmd, myConn);

    myDAdp.Fill(subscribers, "Subscribers");
    return subscribers;
}
```

GetDetails()几乎与 ListsDB 类中的对应方法完全相同，其区别是 SQL 字符串，它与上面的代码相同。

Add()、Delete() 和 Update()方法可以插入、删除和修改用户对新闻列表的订阅情况。它们的执行过程与 ListsDB 类中的对应方法略有不同。Delete()函数首先删除指定列表中的订阅者，再检查该用户是否还订阅了其他列表。如果不是，就把该用户从 UserData 表中删除。Add()函数不仅把订阅者添加到指定列表中，还检查传送过来的电子邮件地址是否与 UserData 表中已有的电子邮件地址相匹配，如果不相匹配，就假定订阅者是一个新用户，并把它添加到该表中。对于这个工程，电子邮件字段的每个项目都假定表示一个惟一的用户。

接着，是两个 GetUserID()重载方法，它们由 Delete() 和 Add()方法在内部使用。第一个方法带一个 Subscriber ID 参数，返回与订阅相关的 User ID ——Delete()在内部使用它。第二个重载方法的参数是一个电子邮件字符串，返回与该电子邮件地址相关的 User ID ——Add()在内部使用它。

这个类的最后一个函数用指定的电子邮件给某个列表返回订阅者的 ID。GetSubscrID()由 Add()方法使用，确定订阅列表的用户是否已经在 UserData 表中了。

4. NewsHistoryDB 类

NewsHistoryDB 类(位于 NewsMailerDB.cs 文件中)是最简单的类，它只有 4 个方法和一个方法参数化的构造函数，如表 27-7 所示。

表　27-7

NewsHistoryDB 类

方　法	描　述
public NewsHistoryDB(string ConnString)	类的构造函数，其参数是一个连接字符串，在设置文件中定义

（续表）

NewsHistoryDB 类	
public DataSet GetNews(int ListID)	获取指定列表的所有新闻项目
public DataRow GetDetails(int NewsID)	获取标识特定新闻记录的行
public bool Add(int ListID, string Subject, string Body, bool IsHTML)	把新闻记录的内容添加到指定的列表中
public bool Delete(int NewsID)	删除指定的新闻记录

5. 编译装配件

完成了这些文件后，就可以编译这个装配件了。如果在 VS.NET 中执行这些步骤，Solution Explorer 窗口就应如图 27-5 所示。

注意，AssembyInfo.cs 包含一组默认属性，这些默认属性将和源文件一起编译到装配件中。这个文件在默认情况下由 Visual Studio.NET 在建立工程时创建。我们没有对它进行任何修改。

此时 Class View 应如图 27-6 所示。

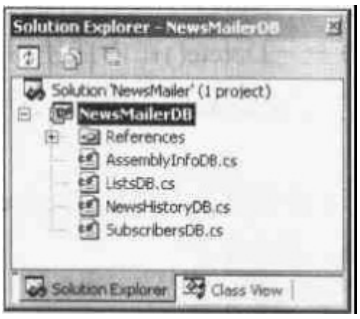


图　27-5



图　27-6

要编译装配件，只需调用 VS.NET 中的 Build 命令即可。

27.3.3　配置数据管理

NewsMailer 可以用特定的确认信息、管理员的名称和电子邮件，以及其他几个选项(稍后将介绍所有的选项)来定制。最好不要在源代码中对这些设置值进行硬编码，因为在管理员需要修改一个选项(例如他的电子邮件地址)时，就要修改源代码，重新编译工程，上载新装配件。这是一个很大的问题，特别是在站点管理员不是程序员时，就更是如此。所以下面把所有的设

置都存储在一个便于读写的外部文件中，并编写一个页面，让管理员轻松地更新这些值，无需重新编译或重新上载任何内容。

1. XML 设置文件

设置一般存储在数据库或文本文件中。如果选择存储在数据库中，就需要创建一个新表，来存储一些设置。可以把所有的设置都存储在一行上(有多个列)，或者把一个设置放在单独的一行上。这种解决方案是很简单的，只需要在 NewsMailerDB 装配件中增加一个新表和一个新类即可，但浪费了数据库资源。更好的解决方案是使用 XML 文件，它的格式很适合于应用程序的设置。而且.NET Framework 提供了一些类，很容易读写所有的设置，并用几行代码把它们显示为一个类的属性。后面将介绍这段代码，下面先解释设置本身。

NewsMailer.Settings 文件用于存储应用程序的所有设置，但没有存储数据库的布局和连接字符串。所有可用的设置如表 27-8 所示。

表　27-8

设置名称	类　型	可否为空	描　述
SenderName	string	是	新闻发送者的名称
SenderEmail	string	否	新闻发送者的电子邮件地址
NewsSubject	string	否	新闻的默认主题。管理员可以直接在页面上为这个默认字符串追加任何文本，编写信息
Signature	string	是	自动追加到信息末尾的签名
SubscriptionSubject	string	否	电子邮件的主题，作为订阅的确认信息发送出去
SubscriptionMessage	string	是	电子邮件的主体，作为订阅的确认信息发送出去
UnSubscriptionSubject	string	否	电子邮件的主题，作为取消订阅的确认信息发送出去
UnSubscriptionMessage	string	是	电子邮件的主体，作为取消订阅的确认信息发送出去
SendSubscriptionEmail	bool	否	如果为 true，新订阅者就会在订阅一个列表后，收到一个确认的电子邮件
SendUnSubscription Email	bool	否	如果为 true，现有的订阅者就会在结束订阅后，收到一个确认的电子邮件
SubscribeURL	string	否	Subscribe.ASP.NET 页面的完整 URL，用于增加或删除列表中的用户

NewsSubject, [Un]SubscriptionSubject 和[Un]SubscriptionMessage 可以包含特殊的标记，例如#SubscriberFirstName# 和 #ListDescription#，在发送每封邮件前，它们分别用各自的值替代。详见本章后面的内容。

在表 27-8 中，我们指定了每个设置的类型。当然，所有的设置都在 XML 文件中存储为字符串。类型表示映射设置值的类的属性类型。“可否为空”列有类似的含义：它描述了哪些设置可以在设置文件中忽略。没有任何模式文件能确保文件中正好有所需的设置 ——用于读取和

修改它们的 ASP.NET 页面可确保文件是有效的。

XML 文件如下所示：

```
<?xml version="1.0" encoding="utf-8"?>
<AppSettings>
    <SenderName>NewsMailer Team</SenderName>
    <SenderEmail>admin@newsmailerserver.com</SenderEmail>
    <NewsSubject>News 4 you: #ListName#</NewsSubject>
    <Signature>- The NewsMailer Team</Signature>
    <SubscrSubject>Welcome to the mailing list #ListName#</SubscrSubject>
    <SubscrMessage>Welcome dear subscriber</SubscrMessage>
    <UnSubscrSubject>Bye bye from #ListName#</UnSubscrSubject>
    <UnSubscrMessage>You've been removed from #ListName#</UnSubscrMessage>
    <SendSubscrEmail>true</SendSubscrEmail>
    <SendUnSubscrEmail>true</SendUnSubscrEmail>
    <SubscribeURL>http://localhost/newsmailer/subscribe.aspx</SubscribeURL>
</AppSettings>
```

2. NewsMailerConfig 装配件

对象的串行化是把对象的状态存储到内存、二进制文件或 ASCII 文件中的过程。对象的状态是它的所有属性、值和引用。另一方面，并行化将创建一个新对象，其状态由存储在一个给定文件中的数据来设置。.NET Framework 通过一组能管理几个不同存储格式的类，使串行化和并行化的过程变得更简单一些。

首先，如果使用的是 VS.NET，就创建一个类型为 Class Library 的新工程，把默认的类文件重命名为 AppConfig.cs。如果用一个文本编辑器编写代码，就应在目录 NewsMailer\Config 下创建这个新文件。这是 AppSettings 类的总结，在串行化一个实例时，它将生成一个 XML 文件，该文件与前面介绍的 XML 文件相同，该文件包含一些公共属性，其名称匹配于 XML 文件中的属性。

AppSettings 类中的所有属性都由 XmlElement 属性标记，这是因为我们要把设置串行化为 XML，为此，使用 System.Xml.Serialization 名称空间中的 XmlSerializer 类。AppSettings 类没有任何串行化或并行化它本身的方法，所以这个任务由 AppConfig 支持类来完成，这个支持类也在 AppConfig.cs 中，如表 27-9 所示。

表 27-9

AppConfig 类

方　法	描　述
public static AppSettings GetSettings(string fileName)	获取包含设置的 XML 文件，并创建包含其数据的一个 AppSettings 对象
public static void SaveSettings(string fileName, AppSettings data)	提取 AppSettings 对象，把它的数据写到给定的 XML 文件中

这两个方法都相当复杂，所以要详细解释它们。首先，下面是 GetSettings()的代码：

```
public static AppSettings GetSettings(string fileName)
{
    HttpContext context = HttpContext.Current;

    AppSettings data = (AppSettings)context.Cache["AppSettings"];

    if (data == null)
    {
        XmlSerializer serializer = new XmlSerializer(typeof(AppSettings));
        try
        {
            // create a filestream to read the XML document
            FileStream fs = new FileStream(fileName, FileMode.Open);
            XmlReader reader = new XmlTextReader(fs);

            // retrieve the oject state
            data = (AppSettings)serializer.Deserialize(reader);
            reader.Close();

            context.Cache.Insert("AppSettings", data,
                                 new CacheDependency(fileName));
        }
        catch (System.IO.FileNotFoundException)
        {
            // if the file is not found, return a new empty class
            data = new AppSettings();
        }
    }
    return data;
}
```

首先，这个方法检查在第一次调用该方法后，带有设置的对象是否已经存储在缓存中，如果对象已经在缓存中，GetSettings()就返回该数据。如果该对象没有在缓存中，就创建一个 XmlSerializer，打开指定为第一个参数的文件，为文件流创建一个 XmlTextReader 对象。最后，XmlSerializer 的 Deserialize()方法获取对象的状态，得到的 AppSettings 对象存储在缓存中，以备将来使用。

这段代码中有几点要注意。第一点，设置文件的路径不是硬编码的，而是作为一个参数来传送。这是必须的，因为这段代码将编译到一个装配件中，可由任意客户使用，而且设置文件可以驻留在不同的位置上。

另一个有趣的地方是设置存储在缓存中的方式，看看下面的代码：

```
context.Cache.Insert("AppSettings", data, new CacheDependency(fileName));
```

第三个参数表示存储在缓存中的项目依赖于作为参数传送的文件。如果该文件被更新，缓存中的项目就不再有效，所以在试图访问设置时，需要再次对文件并行化。自动监视文件是

ASP.NET 支持的一个很好的缓存功能，可以用任何编辑器修改设置文件的内容，重新加载文件，把新设置重新加载到缓存中。

最后一点：当对象状态从文件中恢复时，在 XML 文件中都存储为字符串的设置会自动转换为相应属性的类型。

SaveSettings()方法的构建方式与 GetSettings()类似：它创建 XmlSerializer 的一个实例，在第一个参数指定的位置上创建一个新文件，串行化作为第二个参数的 AppSettings 对象。下面是完整的代码：

```
public static void SaveSettings(string fileName, AppSettings data)
{
    XmlSerializer serializer = new XmlSerializer(typeof(AppSettings));

    // Create an XmlTextWriter using a FileStream
    FileStream fs = new FileStream(fileName, FileMode.Create);
    XmlTextWriter writer = new XmlTextWriter(fs, new UTF8Encoding() );

    // Serialize using the XmlTextWriter
    serializer.Serialize(writer, data);
    writer.Close();
}
```

这里惟一要注意的是，字符如“<”和“>”是编码了的，因为它们在 XML 中有特殊的含义(它们用于把 XML 标记括起来)。编码格式在 XmlTextWriter 对象的构造函数中指定。

3. 编译装配件

如果在 VS.NET 中开发，Solution Explorer 就应如图 27-7 所示。

Class View 应如图 27-8 所示。

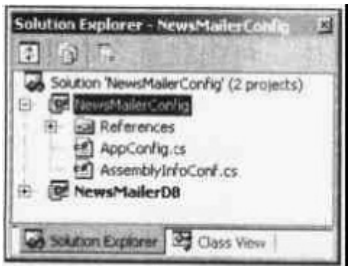


图　27-7



图　27-8

27.4　管理用户界面

完成了数据库设计和事务类后，就完成了相当多的工作。下面要设计应用程序的用户界面，使用事务类读写数据库中的数据。

在解释页面的 ASP.NET 和 C#代码之前，先看看要得到的一部分结果，如图 27-9 所示。

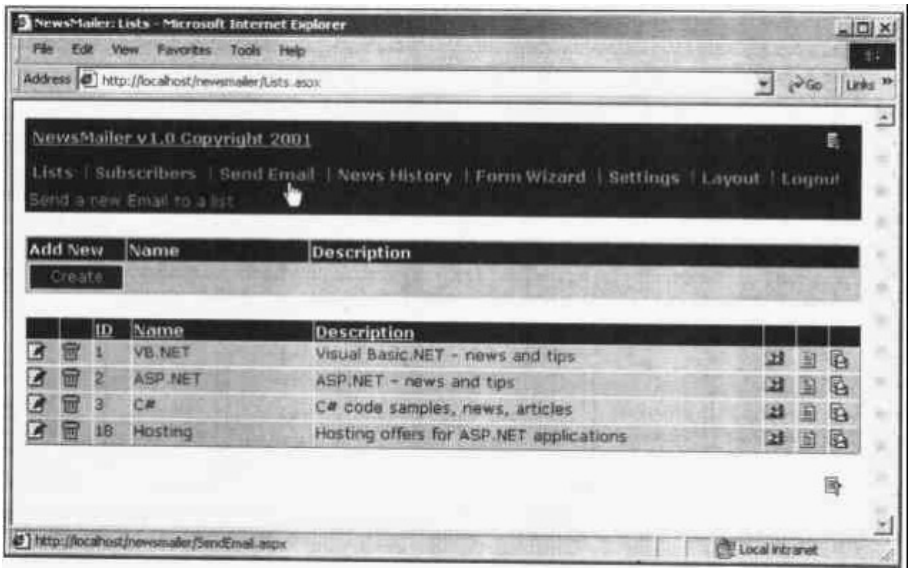


图　27-9

上面的页面允许管理员查看可用的邮件列表，创建、删除或编辑列表、跳转到订阅者列表或应用程序的其他页面上。页面中的表显示了它从数据库中获得的所有列表，显示了它们的属性，并可以按照 ID, Name 或 Description 对它们排序。而且，表中的所有图标(ID 列左边、Description 列右边的图像)都是其他页面的超链接，单击它们可以完成与该行列表相关的其他操作。表 27-10 是网格图标的列表，以及它们执行的操作说明。

表　27-10

图　标	描　述
	编辑记录
	删除记录
	跳转到一个页面上，该页面显示这个邮件列表的订阅者
	跳转到一个页面上，该页面显示发送给这个邮件列表的旧时事通讯
	跳转到一个页面上，以编写新时事通讯，并把它发送给这个邮件列表的订阅者

如果读者以前曾使用传统的 ASP 编程，就会觉得这个网格需要一些复杂的代码来管理所有的超链接、图像、行的其他颜色和数据排序等。如果是这样，读者就会很惊讶地发现使用 ASP.NET 和几行 C#代码很容易完成这些任务。

应用程序由几个 ASP.NET 页面组成，多数页面都是用于管理的。图 27-10 说明了页面的一般构成，页面的完整列表，它们的文件名、各自的后台编码类和文件。

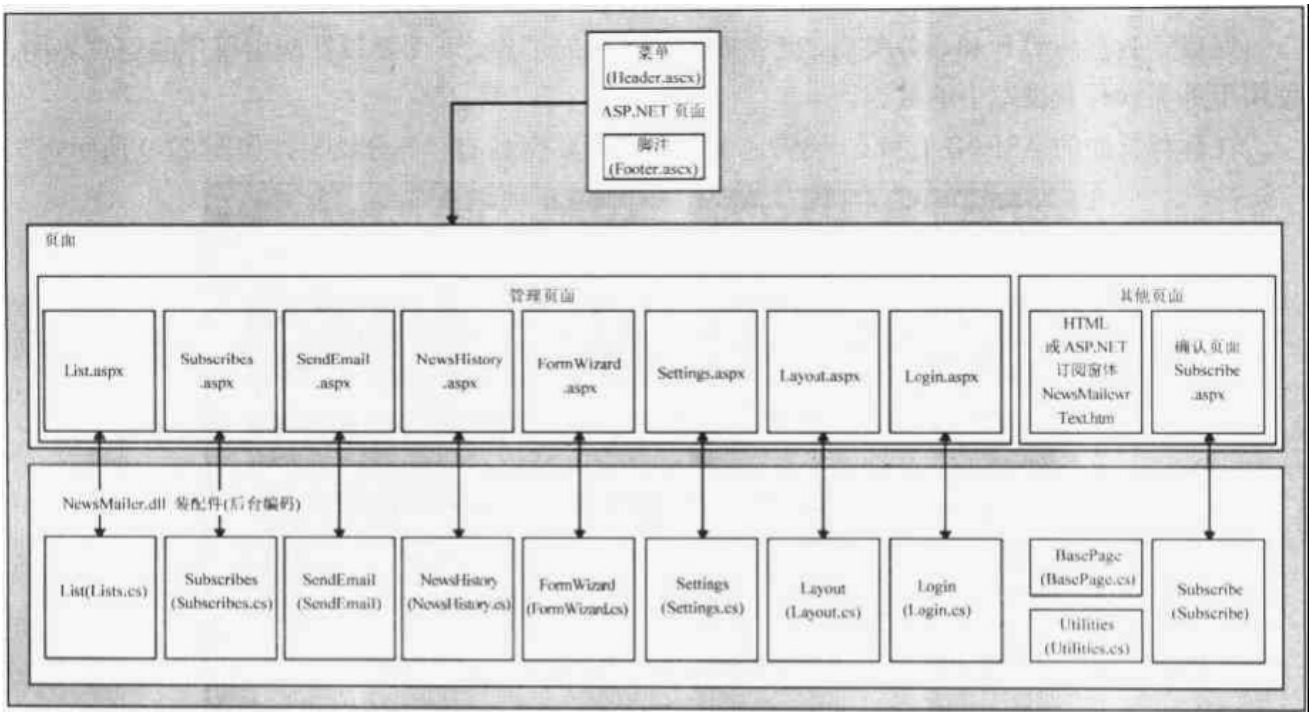


图　27-10

27.4.1　创建用户控件

如果部分用户界面需要用在多个页面上，就应考虑如何重用代码，避免在页面之间复制和粘贴它们。在这个程序中，要共享标题和脚注部分，它们分别是顶部的菜单和网格右下角的小图标，该图标把页面滚动到顶部。在传统的 ASP 中，要用<!--#INCLUDE -->指令包含两个不同的文件，但在 ASP.NET，有另一种更好的方式：用户控件。使用用户控件，可以把共享代码单独放在一个独立的文件中，再在需要时包含它们。给要导入的控件指定一个名称，修改它的公共属性，或调用它包含的公共方即可。换言之，它不是插入到页面上的一段代码，而是作为一个带有名称和属性的图形组件来处理的。用户控件作为一个服务器组件，像其他 ASP.NET 服务器组件一样，是在服务器上处理的，用户浏览器上显示的是与所有浏览器都兼容的标准 HTML。

在工程中有两个用户控件，如表 27-11 所示。

表　27-11

用 户 控 件	描　　述
Header.ascx	为这个 Web 站点上所有的页面添加一个共同的标题，它由 3 部分组成： • 标题部分，格式化为包含文本和一个徽标的表格 • 第二个表，它包含到其他 Web 页的链接 • 一个标签，当鼠标指向它时，显示一个简单的文本信息，描述每个链接
Footer.ascx	只包含一个图像链接，它把用户重定向到页面的顶部

这两个文件都包含在下载的代码中。

在 Web 页上使用用户控件

要在 Web 页上使用用户控件，必须在页面上注册它们，方法是指定它的源文件和一个前缀，把它与其他服务器控件区分开来。在页面的顶部添加下面两行代码：

```
<%@ Register TagPrefix="NewsMailer" TagName="Header" src="Header.ascx" %>
<%@ Register TagPrefix="NewsMailer" TagName="Footer" src="Footer.ascx" %>
```

下面是在正常页面上添加 Header 和 Footer 用户控件的代码：

```
<html>
<head>
    <title>NewsMailer: Subscribers</title>
    <link rel="stylesheet" HREF="Styles.css" TYPE="text/css">
</head>
<body>
    <!-- Insert the menu user control -->
    <NewsMailer:Header id="Menu" runat="server" />

    <b>Some HTML code</b>

    <!-- Insert the footer -->
    <NewsMailer:Footer id="Footer" runat="server" />
</body>
</html>
```

27.4.2　Lists 管理器页面

图 27-9 说明了邮件列表的管理页面刚刚加载时的外观。其外观将根据我们要执行的操作而有所改变。例如，当单击左侧的笔记本和钢笔图标，编辑一个列表时，会得到如图 27-11 所示的结果。

列表的当前 Name 和 Description 都显示在两个文本框中，管理员可以修改这些值，单击绿色的记号图标，就可以更新数据库中的值，单击 X，就可以取消操作。注意 ID 列不是可编辑的，因为 ID 是一个计数列，其值不能修改。列表可以按照 Name 排序，但不能按照 ID 排序。单击列的标题，就可以按照该列排序。

要添加一个新列表，管理员可以单击主页左上角的 Create 按钮，得到如图 27-12 所示的结果。

显示两个空白的文本框，管理员可以用要创建的邮件列表的名称和描述来填充它们，用左边的两个图标可以确认或取消该操作。编辑操作也可以使用这两个图标来确认或取消。文本框是隐藏的，显示返回原来的结果。单击图标编辑新行时，这个操作几乎是自动实现的，但我们需要处理控件的事件，在添加新行时，显示或隐藏文本框。这个过程要比初看起来简单得多，只需编写几行代码，稍后我们会介绍这方面的内容。

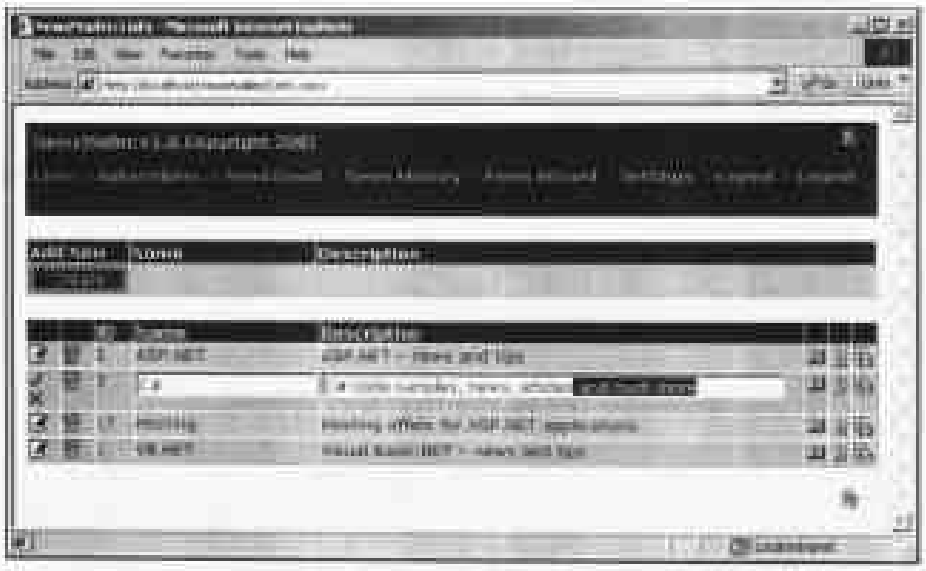


图 27-11



图 27-12

1. Lists.aspx 页面

这个 ASP.NET 页面定义了用户界面。与 ASP 页面一样，这个页面还可以用于存储脚本代码。但是 ASP.NET 页面的一个主要优点是通过使用后台编码文件(codebehind files)，把代码与用户界面分离开来，后台编码文件这个术语在本章中用得非常多(而且，前面已经用了这个术语，但没有定义它)。后台编码文件有一个或多个处理 Web 页所有事件的类。两个层的分离可以让 Web 设计人员修改 Web 页的布局，而无需读取所有有关该页面的混乱的 VBScript 或 JScript 代码以及 HTML 代码。而且，后台编码文件可以编译到一个装配件中，更易于应用程序的部署(因为我们只上传一个 DLL，而不是几个 C#源文件)，这增加了应用程序的安全性，因为即使有人可以访问服务器，读取事务代码也不像使用 ASP 那样快速。

要在 VS.NET 中创建 Lists.aspx 页面，可选择 Project | Add Web Form。如果使用的是其他编辑器，可以在 NewsMailer 文件夹下创建一个新文件。

页面的源代码很长，不能都列出来，所以这里只解释 C#程序员感兴趣的部分。第一行定义用于处理页面事件的基类：

```
<%@ Page language="c#" Inherits="NewsMailer.Lists" %>
```

这个类包含页面加载时执行的所有代码，单击一个按钮、添加、删除或编辑一个邮件列表等。类 Lists 是 NewsMailer 名称空间的一部分，如前所述，它编译到一个独立的装配件 NewsMailer.dll(详见后面的内容)中。该页面会自动查找 Web 根文件夹下的 bin 文件夹，找到这个装配件，并继承该类。包含 C#源代码的文件(本例是 Lists.cs)就是后台编码文件。

移动到页面的主体部分，其中有一个窗体，该窗体包含 Web 页在标题和脚注的中间部分的

代码。首先，看看 Create 按钮，它位于菜单的左下方。单击这个按钮时，就会显示用于确认或取消操作的两个文本框，如图 27-12 所示。界面的其余部分是相同的。下面是这个按钮的代码：

```
<asp:TableRow ID="CreateNewRow">
   <asp:TableCell ColumnSpan="3">
      <asp:Button runat="server" Text="Create"
                  ID="Create" CssClass="Button"
                  OnClick="Create_Click" Width="80px"
      />
   </asp:TableCell>
</asp:TableRow>
```

接着，有一个表行，它在默认状态下是不可见的，但单击 Create 按钮后，它就是可见的。这里可以修改列表的内容，接受或取消所做的修改操作：

```
<asp:TableRow ID="AddNewControlsRow" Visible="false">
   <asp:TableCell>
      <asp:Image ImageURL="./Images/imgSpacer.gif"
                 Width="15px" Height="1px" runat="server" />
      <asp:LinkButton ID="AddNew" runat="server"
                 Text="<img border=0 src=./Images/imgOK.gif
                 Alt='Add new list'>"
                 OnClick="AddNew_Click"
      />
      <asp:Image ImageURL="./Images/imgSpacer.gif"
                 Width="15px" Height="1px" runat="server" />
      <asp:LinkButton ID="CancelAddNew"
                 runat="server" CausesValidation="false"
                 Text="<img border=0 src=./Images/imgCancel.gif
                 Alt='Cancel editing mode'>"
                 OnClick="CancelAddNew_Click"
      />

   </asp:TableCell>
   <asp:TableCell VerticalAlign="Top">
      <asp:TextBox runat="server" Class="TextBox"
                 ID="NewListName" Width="150px" />
      <asp:RequiredFieldValidator Id="ValidateNewName"
                 runat="server" ControlToValidate="NewListName"
                 Display="dynamic"><br>* Name is required
      </asp:RequiredFieldValidator>
   </asp:TableCell>

   <asp:TableCell VerticalAlign="Top">
      <asp:TextBox runat="server" Class="TextBox"
                 ID="NewListDescr" Width="450px" />
```

```
        </asp:TableCell>
    </asp:TableRow>
```

注意需要编程访问的所有服务器控件都有一个 ID 属性，所有的代码都放在一个服务器端窗体中。还要注意，在创建新邮件列表时，需要这个邮件列表的名称，这是由 RequiredFieldValidator 控件来保证的。ControlToValidate 属性的含义一目了然，Display 属性设置为 dynamic，指定显示错误信息的空间是动态创建的。

最有趣的一个服务器控件是数据网格控件，它在本应用程序中用于显示邮件列表。这个控件仅用几行代码就提供了许多功能：交替数据行的不同样式、数据的排序和版面。最重要的是它是完全可配置的，且基于模板。通过一个代码模板，就可以定制每一列的内容，以及显示该列数据的方式。

下面设置了数据网格的属性：

```
<asp:DataGrid id="ListsGrid" runat="server"
        CssClass="Grid_General"
        HeaderStyle-CssClass="Grid_Header"
        ItemStyle-CssClass="Grid_Item"
        AlternatingItemStyle-CssClass="Grid_AlternatingItem"
        AllowSorting="True"
        AutoGenerateColumns ="False"
        DataKeyField="ID"
        OnEditCommand="ListsGrid_Edit"
        OnCancelCommand="ListsGrid_CancelEdit"
        OnUpdateCommand="ListsGrid_Update"
        OnDeleteCommand="ListsGrid_Delete"
        OnSortCommand="ListsGrid_Sort" >
```

设置数据网格中各个组件的显示样式，指定它是可以排序的，为网格设置源表中用作数据主键码的列(DataKeyField 属性)。最后一行代码定义了事件过程，调用它可以处理网格的操作。

接着，需要手动声明每一列如何显示。需要显示下述列：

- 可单击的图像，以编辑行
- 可单击的图像，以删除行
- 显示列表名称的列
- 显示列表描述的列
- 可单击的图像，以跳转到另一个页面上，该页面显示了当前行的列表订阅者
- 可单击的图像，以跳转到另一个页面上，该页面显示了发送给列表的旧信息
- 可单击的图像，以跳转到另一个页面上，把新电子邮件发送给列表的订阅者

本书的前面没有介绍数据网格，下面就详细说明如何使用它。前两列需要的代码如下所示：

```
<Columns>
    <asp:EditCommandColumn
            ItemStyle-Width="25px"
            EditText="<img border=0 Alt='Edit this list'
```

```
                    src=./Images/imgEdit.gif>"
            CancelText="<img border=0 src=./Images/imgCancel.gif>"
            UpdateText="<img border=0 src=./Images/imgOK.gif>"
    />
    <asp:ButtonColumn
            ItemStyle-Width="25px"
            Text="<img border=0 Alt='Delete this list'
                    src=./Images/imgDelete.gif>"
            CommandName="delete"
    />
```

EditCommandButton 是一个特殊的列，其中带有要开始编辑的链接，而 ButtonColumn 创建一般的按钮或链接。必须处理 Click 事件，以执行需要的操作。注意通过把<img>标记设置为该列的 Text 属性，显示按钮。如果编写了一个一般的字符串，而不是 HTML，就会得到一个标准按钮，其标题就是该字符串。

下面的列代码显示了列表的 ID、名称和描述：

```
<asp:BoundColumn HeaderText="ID"   ItemStyle-Width="30px"
                DataField-"ID" ReadOnly="True" SortExpression="ID" />
<asp:TemplateColumn HeaderText="Name" SortExpression="Name"
                    ItemStyle-Width="150px">
  <ItemTemplate>
    <asp:Label runat="server" Text='<%#
                DataBinder.Eval(Container.DataItem, "Name") %>' />
  </ItemTemplate>
  <EditItemTemplate>
    <asp:TextBox ID="EditListName" runat="server"
                Text='<%# DataBinder.Eval(Container.DataItem,
                                        "Name") %>'
                Class="TextBox"
                Width="150px"
    />
    <asp:RequiredFieldValidator Id="ValidateEditName"
                runat="server"
                ControlToValidate="EditListName"
                Display="dynamic">* Name is required
    </asp:RequiredFieldValidator>
  </EditItemTemplate>
</asp:TemplateColumn>
<asp:TemplateColumn HeaderText="Description"
                    SortExpression="Description">
  <ItemTemplate>
    <asp:Label runat="server" Text='<%#
                DataBinder.Eval(Container.DataItem,
                                    "Description") %>' />
```

```
    </ItemTemplate>
    <EditItemTemplate>
        <asp:TextBox ID="EditListDescr" runat="server"
                     Text='<%# DataBinder.Eval(Container.DataItem,
                                                "Description") %>'
                     Class="TextBox"
                     Width="380px"
        />
    </EditItemTemplate>
</asp:TemplateColumn>
```

第一列被定义为一个一般的非编辑列，这里不再讨论它。其他两列比较有趣，因为它们说明了如何创建一个模板列，它用一个标签来显示值，或者使用文本框来编辑它。当进入或退出行的编辑模式时，将自动设置适当的视图。注意标签和文本框的文本由 DataBinder.Eval()方法处理，该方法返回已读取行的列值。如前面的表所示，要添加一个新列表，需要输入该列表的名称。

剩下的 3 列都是链接，通过这些链接可以导航到内容页面上：

```
<asp:HyperLinkColumn
        DataNavigateUrlField="ID"
        DataNavigateUrlFormatString="Subscribers.aspx?ListID={0}"
        DataTextFormatString="<img border=0 Alt='Show
                subscribers' src=./Images/imgAddresses.gif>"
        DataTextField="ID"
/>
<asp:HyperLinkColumn
        DataNavigateUrlField="ID"
        DataNavigateUrlFormatString="NewsHistory.aspx?ListID={0}"
        DataTextFormatString="<img border=0 Alt='Show past
                news for this list' src=./Images/imgMessage.gif>"
        DataTextField="ID"
/>
<asp:HyperLinkColumn
        DataNavigateUrlField="ID"
        DataNavigateUrlFormatString="SendEmail.aspx?ListID={0}"
        DataTextFormatString="<img border=0 Alt='Send a new
                mail to this list' src=./Images/imgMail.gif>"
        DataTextField="ID"
/>
```

它们都是非编辑的超链接，显示为图像，它们指向的 URL 是用在 DataNavigateUrlFormatString 中定义的字符串模板来建立的，每一行的 ID 由标记{0}替代。

2. BasePage 类

在介绍上一节提及的 Lists.cs 后台编码文件之前，先介绍一个基类，这个案例分析的后台

编码文件所包含的所有类都使用这个基类 BasePage，它派生于 System.Web.UI.Page，详见表 27-12。

表 27-12

BasePage 类	
方　法	描　述
Public string GetConnString()	从 web.config 文件中获取数据库的连接字符串，web.config 文件存储了应用程序的所有配置设置
public string GetSettingsFile()	从存储应用程序设置的地方中获取文件名，该文件名也在 web.config 中

这个类有两个公共方法，所有的派生类都继承了这两个方法。这是一个很好的面向对象编程，因为所有这些类都需要连接字符串和设置文件名，最好把存取器的功能放在一个地方，而不是把该功能多次复制和粘贴到每个类中。这两个函数的执行方式非常类似，它们都将在 web.config 的这个部分中查找连接字符串的值和设置文件，再把它们存储在缓存中：

```
<configuration>
    <appSettings>
        <add key="ConnString" value="Provider=Microsoft.Jet.OLEDB.4.0;
            Data Source=E:\Projects\NewsMailer\DB\NewsMailer.mdb" />

        <add key="SettingsFile" value="NewsMailer.Settings" />
    </appSettings>
</configuration>
```

注意必须修改上述 ConnString 键码的值，使之指向数据库文件的正确路径。

3. Lists.aspx 的后台编码文件

定义了基类后，下面介绍后台编码文件 Lists.cs。

4. Lists 类

如果扫描一下前面的 using 指令列表，就会注意到类在 Newsmailer 名称空间中，派生于前面介绍的 BasePage：

```
namespace NewsMailer
{
public class Lists : NewsMailer.BasePage
{
```

在声明了几个受保护的 Web Form 控件后，就要编写页面的功能代码了。下面是在加载页面时运行的处理程序：

```
protected void Page_Load(object sender, EventArgs e)
{
```

```
    if (!Page.IsPostBack)
    {
        // bind the page's controls
        BindGrid();
    }
}
```

在上面的例程中，有选择地调用 BindGrid()方法，它用邮件列表填充网格：

```
protected void BindGrid()
{
    NewsMailer.ListsDB lists = new NewsMailer.ListsDB(GetConnString());
    // get all the lists
    DataView myDV = lists.GetLists().Tables[0].DefaultView;

    // sort the data according to the SortExpression value
    if ( ListsGrid.Attributes["SortExpression"]!= null )
        myDV.Sort = ListsGrid.Attributes["SortExpression"];

    ListsGrid.DataSource = myDV;
    ListsGrid.DataBind();
}
```

如果页面的 IsPostBack 属性是 false，就只调用该函数，此时页面是第一次加载，或者由用户用浏览器的 Refresh/Reload 命令手动刷新，而不是重新加载页面，因为用户把一些数据传送给窗体。实际上，当页面被传送回来时，控件的状态存储在一个隐藏的 HTML 控件中，并自动恢复，而不再需要重新查询数据库。

BindGrid()可能是每个页面中最重要的函数，因为在该方法中，页面从数据库中获取数据，并通过数据网格显示基类。第一行从前面创建的 NewsMailerDB 库中实例化一个 ListsDB 对象：记得吗？其构造函数带一个由它的任意方法使用的连接字符串参数，而该连接字符串由 BasePage 的 GetConnString()方法返回：

```
NewsMailer.ListsDB lists = new NewsMailer.ListsDB(GetConnString());
```

下面一行代码获取一个数据集，在一个私有变量中存储表的数据视图：

```
DataView myDV = lists.GetLists().Tables[0].DefaultView;
```

接着根据一个数据网格属性存储数据视图，这是一个定制属性，在后面的代码中设置它。在第一次调用函数时，这个属性为空，没有存储数据视图。最后两行把数据视图与 DataSource 属性关联起来，调用 DataBind()方法显示数据行。

5. 对列表排序

单击一个列标题时，调用 ListsGrid_Sort 函数，如下所示：

```
protected void ListsGrid_Sort(Object sender,DataGridSortCommandEventArgs e)
```

```
{
    ShowAddNewControls(false);
    ListsGrid.EditItemIndex = -1;
    // set the SortExpression attribute that will be used
    // to actually sort the data in the BindGrid method
    ListsGrid.Attributes["SortExpression"]=e.SortExpression.ToString();
    BindGrid();
}
```

该函数首先隐藏添加新列表的控件，以防它们是可见的。接着禁用数据行的编辑模式(如果数据行可以编辑)。最后，该函数获取排序表达式，对数据和列名进行排序，调用 BindGrid 刷新数据网格控件的内容。

6. 编辑和更新列表

下面执行处理数据行的编辑和更新的方法：

```
protected void ListsGrid_Edit(object sender, DataGridCommandEventArgs e)
{
    ShowAddNewControls(false);
    // start editing
    ListsGrid.EditItemIndex = (int)e.Item.ItemIndex;
    BindGrid();
}

protected void ListsGrid_CancelEdit(object sender,
    DataGridCommandEventArgs e)
{
    ListsGrid.EditItemIndex = -1;
    BindGrid();
}

protected void ListsGrid_Update(object sender,
    DataGridCommandEventArgs e)
{
    if (Page.IsValid)
    {
        // get the new values from the textboxes
        string ListName =
                ((TextBox)e.Item.FindControl("EditListName")).Text;
        string ListDescr =
                ((TextBox)e.Item.FindControl("EditListDescr")).Text;
        int ListID = (int)ListsGrid.DataKeys[e.Item.ItemIndex];

        NewsMailer.ListsDB lists = new NewsMailer.ListsDB(
                GetConnString());
```

```
            // update the values
            lists.Update(ListID, ListName, ListDescr);

            ListsGrid.EditItemIndex = -1;
            BindGrid();
        }
    }
```

第一个事件过程 ListsGrid_Edit()隐藏了添加数据行的控件，把 EditItemIndex 属性设置为要编辑的数据行索引，这个值是从 Item 对象中获取的，它显示为对象 e 的一个属性，作为参数传送给函数。最后，再次调用 BindGrid()，把该行的数据绑定到两个文本框上。单击 X 图标后，仅把 EditItemIndex 设置为－1，刷新网格。

最有趣的部分是数据行的更新，这是在 ListsGrid_Update()过程中完成的。首先注意，只有在页面有效的情况下，即只有遵守了有效性验证控件的规则后，才可以执行代码。此时，Name 文本框必须用一个有效的名称填写。如果这个检查通过了，就会继续执行，并获取两个文本框的当前值。每个文本框控件的引用都由 Item 的 FindControl()方法返回，接着由 Text 属性读取其内容。注意要读取 Text 属性，必须有一个文本框控件，它不是由 FindControl()显式返回的，所以需要把返回的对象手动转换为一个 TextBox。最后一个参数是要更新的数据行的 ID，它是很容易获得的，因为这个列设置为数据网格的主键码，只需读取一个属性值即可。最后，创建一个 ListsDB 对象，调用其 Update()方法，把列表 ID、新名称和描述传送给该方法——数据库被更新，并可以刷新数据网格，以反映这些变化。

7. 删除列表

ListGrid_Delete()事件的过程与前面的事件过程有相同的结构，但它更简单，因为只需把一个参数传送给 ListsDB 对象的 Delete()方法，这个参数就是列表的 ID，它由 DataKeys 枚举属性返回：

```
protected void ListsGrid_Delete(object sender, DataGridCommandEventArgs e)
{
    ShowAddNewControls(false);
    ListsGrid.EditItemIndex = -1;
    NewsMailer.ListsDB lists = new NewsMailer.ListsDB(GetConnString());
    // get the ID of this record and delete it
    lists.Delete( (int)ListsGrid.DataKeys[e.Item.ItemIndex] );
    BindGrid();
}
```

8. 创建列表

最后要介绍的是如何创建一个新数据行。单击 Create 按钮后，就会显示两个文本框和这两个文本框各自的图标，分别表示确认或取消创建操作，同时 Create 按钮不可见。数据网格的 EditItemIndex 属性也设置为－1，禁止执行任何编辑操作。下面是这个事件的代码，以及根据传送过来的布尔参数显示或隐藏控件的函数代码：

```
protected void Create_Click(object sender, EventArgs e)
{
    // show the textboxes and buttons for adding a new record
    ShowAddNewControls(true);
    ListsGrid.EditItemIndex = -1;
    BindGrid();
}

protected void ShowAddNewControls(bool ShowControls)
{
    // show/hide the controls for adding a new record
    NewListName.Text="";
    NewListDescr.Text="";
    AddNewControlsRow.Visible = ShowControls;
    CreateNewRow.Visible = !ShowControls;
}
```

添加新列表的图标是一个按钮，它有一个单击事件过程：

```
protected void AddNew_Click(object sender, EventArgs e)
{
    if (Page.IsValid)
    {
        NewsMailer.ListsDB lists = new NewsMailer.ListsDB(
            GetConnString());
        // add the new record
        lists.Add(NewListName.Text, NewListDescr.Text);
        ShowAddNewControls(false);
        BindGrid();
    }
}
```

如果页面是有效的，就像前面的 Update()事件那样，代码就会获得两个文本框的内容，并把这些字符串传送给 ListsDB 对象的 Add()方法。在本例中，代码比较简单，因为我们可以直接引用两个文本框，理由是没有以它们的名称命名的其他控件。

取消添加操作的图标是一个 ButtonLink 控件，在单击该按钮时，会隐藏代表新列表名称和描述的控件，把它们恢复为默认状态。下面是其 Click 事件的代码：

```
protected void CancelAddNew_Click(object sender, EventArgs e)
{
    ShowAddNewControls(false);
}
```

这里有一个要注意的地方，单击一个按钮后，会把页面传送回来。当传送回页面时，会验证代表列表名称和描述的控件，因此如果遗漏了新列表的名称，就停止传送。为了避免这个问

题，可以把该按钮的 CausesValidation 属性设置为 false，指定这个按钮不引发窗体的有效性验证。

现在这个页面的代码就编码完了。可以看出，执行该页面不需要很多代码，而传统的 ASP 页面需要的代码量较大：一个功能全面的网格在 ASP 中需要上百行(甚至上千行)的复杂代码。

27.4.3 管理订阅者

管理列表订阅者的页面非常类似于刚才分析的页面，所以下面仅讨论它们的一些区别。图 27-13 所示的页面是在添加新订阅者的过程中显示的，用户试图不输入电子邮件地址就成为一个新订阅者，所以有效性验证器会通知用户需要填写电子邮件字段。

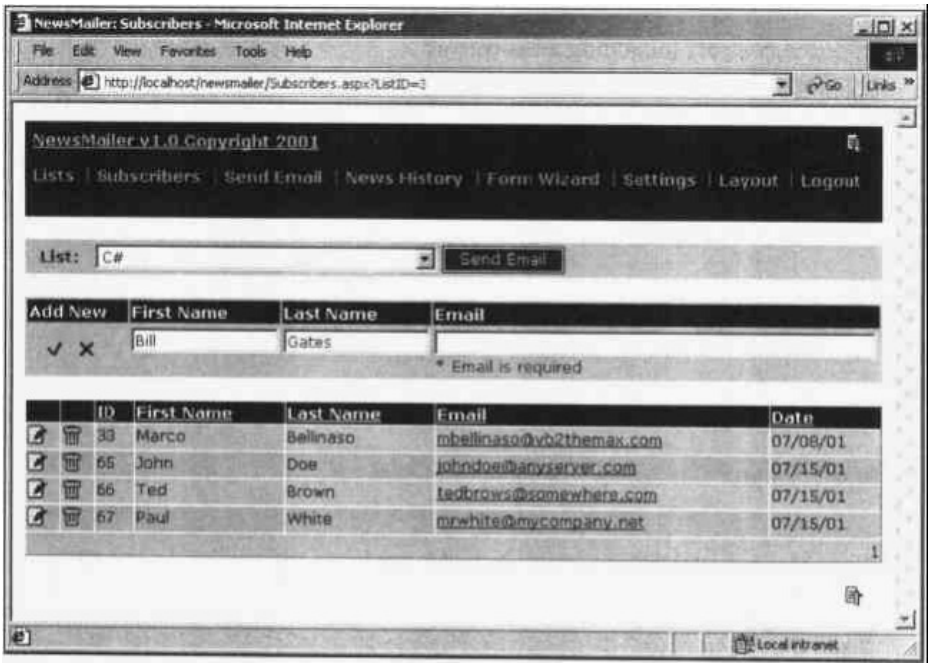


图　27-13

1. Subscribers.aspx 页面

从图 27-13 中可以看出，这个页面与前面的页面是有区别的。首先网格中的列数不同，其中有一个下拉列表，在其中可以选择邮件列表，并查看该列表的订阅者。还有一个可以把用户传送给页面的按钮，给邮件列表发送新信息。

对于列来说，仅增加了几个新模板列，在 DataBinder.Eval()调用中修改了列名，再修改其他非模板列的一些属性。这个页面的代码比较长，下面是第一部分，表示组合框和 Send Email 按钮：

```
<asp:table id="TableLists" runat="server" CssClass="Grid_General">
    <asp:TableRow>
        <asp:TableCell width="50px"
            HorizontalAlign="Center"><b>List:</b></asp:TableCell>
        <asp:TableCell width="280px">
<asp:DropDownList runat="server" ID="ListsDropDown"
                DataTextField="Name" DataValueField="ID" Width="280px"
                Class="TextBox" AutoPostBack="True"
                OnSelectedIndexChanged="ListsDropDown_IndexChanged" />
```

```
            </asp:TableCell>
            <asp:TableCell>
                <asp:Button runat="server" Text="Send Eail"
                    ID="SendEmail" CssClass="Button"
                    OnClick="SendEmail_Click" Width="100px" />
            </asp:TableCell>
        </asp:TableRow>
    </asp:table>
```

注意，下拉列表的 AutoPostBack 属性已被设置为 true，则每次用户修改了选中的项目时，都会把窗体传送回来。这个事件引发了 OnSelectedIndexChanged，调用了 ListsDropDown_IndexChanged() 过程。

页面的其他内容类似于前面的页面。下面看看额外添加到数据网格声明中的几行代码：

```
<asp:DataGrid id="SubscrGrid" runat="server"
    AllowPaging="True"
    PageSize="20"
    PagerStyle-HorizontalAlign="Right"
    PagerStyle-PageButtonCount="20"
    PagerStyle-Mode="NumericPages"
    OnPageIndexChanged="SubscrGrid_PageChanged"
...
```

这些代码行分别表示，数据在每个页面上显示 20 行，用户可以通过右下角的页码在页面之间浏览，并指定了处理修改页面的事件过程。

2. Subscribers.aspx 的后台编码文件

与 ASP.NET 代码一样，Subscribers 页面的后台编码文件与 Lists 页面没有什么太大的区别。只是添加了几行代码，处理新增的控件。下面是在加载页面时执行的代码：

```
protected void Page_Load(object sender, EventArgs e)
{
    // get the ListID from the QueryString
    string ListID = Request.Params["ListID"];
    SubscrGrid.Attributes["ListID"] = ListID;

    if (!Page.IsPostBack)
    {
        // load all the avaible lists in the DropDown control
        NewsMailer.ListsDB lists = new NewsMailer.ListsDB(GetConnString());
        DataView myDV = lists.GetLists().Tables[0].DefaultView;
        ListsDropDown.DataSource = myDV;
        ListsDropDown.DataBind();

        if (ListID!=null)
        {
```

```
            // select the DropDown element according to the ListID
            foreach (ListItem item in ListsDropDown.Items)
                if (item.Value==ListID)
                    ListsDropDown.SelectedIndex =
                        ListsDropDown.Items.IndexOf(item);
        }

        // (re)bind the page's controls
        BindGrid();
    }
}
```

如果还记得 Lists.aspx 页面，就知道在每个网格行的右边都有一个图标，单击该图标，就会把用户引入 Subscribers.aspx 页面，查看该列表的订阅者。为此，图标应指向 Subscribers.aspx 页面，并在查询字符串中添加列表的 ID，其中 ID 作为一个参数。这个值从查询字符串中获取，存储为数据网格控件的一个属性，因为它在代码调用该列表的 GetSubscribers()方法时使用。

继续下去，除了通常的 BindGrid()调用外，这次也有一个用可用的邮件列表来填充的下拉列表。这直接在该过程中完成，具体操作是调用 ListsDB 类的 GetLists()方法，把得到的表格关联为控件的一个数据源。一个接一个地读取刚才添加的项目，选择其 ID 等于查询字符串中 ID 的项目。

这个页面的 BindGrid()方法有一些用于读取列表的 ID 的新代码，以获取该列表的订阅者，再使用 SubscribersDB 类获取需要的数据集(以及表格)：

```
protected void BindGrid()
{
    int ListID;

    // get the ListID value from the Grid's attributes
    if ( SubscrGrid.Attributes["ListID"] != null )
        ListID = int.Parse(SubscrGrid.Attributes["ListID"]);
    else
        ListID = int.Parse(ListsDropDown.SelectedItem.Value);

    // get all the subscribers of this list
    NewsMailer.SubscribersDB subscribers = new NewsMailer.SubscribersDB(
        GetConnString());
    DataView myDV = subscribers.GetSubscribers(
        ListID).Tables[0].DefaultView;

    // sort the records according to the DataGrid's SortExpression attribute
    if ( SubscrGrid.Attributes["SortExpression"]!="" )
        myDV.Sort = SubscrGrid.Attributes["SortExpression"];

    SubscrGrid.DataSource = myDV;
    SubscrGrid.DataBind();
}
```

这个函数从下拉列表中选择一个项目：

```
protected void ListsDropDown_IndexChanged(object sender, EventArgs e)
{
    // reload the page to show the records for the selected list
    Response.Redirect("Subscribers.aspx?ListID=" +
        ListsDropDown.SelectedItem.Value);
}
```

它把浏览器重定向到同一个页面，但指向选中项目的 ID，以便下一次调用 BindGrid()时加载正确的订阅者。

当用户浏览到另一个网格页面上时，需要取消编辑模式，即在重新绑定网格前，必须隐藏添加新数据行的控件，当要添加的订阅者已经在列表上时，隐藏要显示的错误信息。当然，还必须把 CurrentPageIndex 属性设置为新值，修改当前页面：

```
protected void SubscrGrid_PageChanged(Object sender,
    DataGridPageChangedEventArgs e)
{
    SubscrGrid.EditItemIndex = -1;
    AddNewError.Visible = false;
    ShowAddNewControls(false);
    // change the current page and reding the data
    SubscrGrid.CurrentPageIndex = e.NewPageIndex;
    BindGrid();
}
```

最后，处理 Send Email 按钮单击事件的过程，把当前列表的 ID 作为其参数，跳转到 SendEmail.aspx 页面，对下拉列表使用相同的方式处理：

```
protected void SendEmail_Click(object sender, EventArgs e)
{
    // redirect to the page to send an Email,
    // adding the ListId value to the QueryString
    Response.Redirect("SendEmail.aspx?ListID=" +
        ListsDropDown.SelectedItem.Value);
}
```

27.4.4　给订阅者发送电子邮件

现在我们处于应用程序的一个关键地方，必须设计和编写页面的代码，该代码允许管理员编写一个信息，并把该信息发送给列表的所有订阅者，以及把信息存储在数据库中，以备将来查询。与往常一样，先给出图 27-14，让读者对这个过程有一个大致的了解。

通过这个页面，管理员可以编写一个电子邮件，设置对大多数邮件客户来说都相同的所有邮件选项：CC、BCC、电子邮件的格式 (纯文本或 HTML)、优先级、标题和主体。另外，还可以使用几个特殊格式的字符串(它们列在右边)，每个发送出来的电子邮件都会自动用各自的

值替代这些字符串。电子邮件通过与 Windows 2000 Server 一起安装的默认 SMTP 服务发送，所以不需要特定服务器或第三方服务器发送信息。也可以查看信息的预览，但用#分隔的字符串不会转换，因为预览并不是订阅者专用的。图 17-15 是前面屏幕中页面主体的预览。



图 27-14

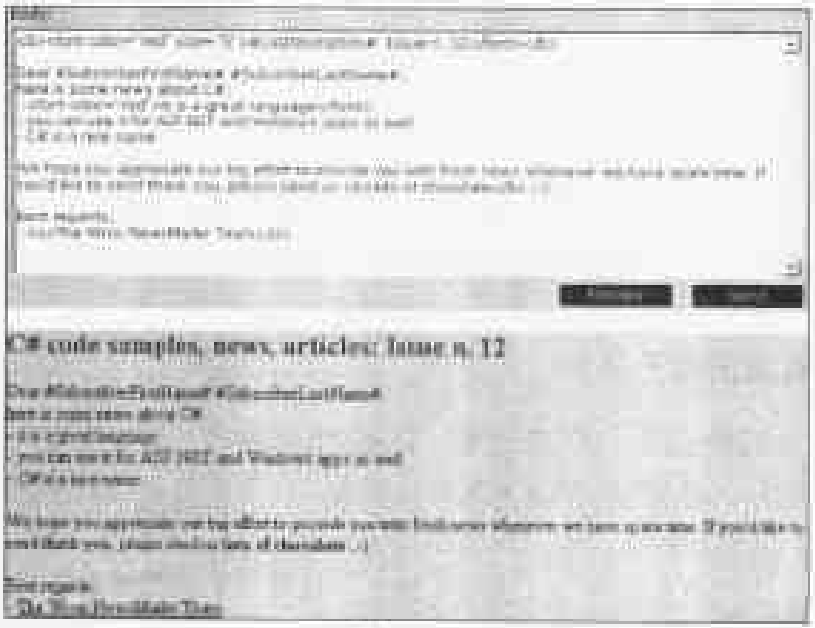


图 27-15

1. SendEmail.aspx 页面

ASP.NET 页面包含的代码类似于前面介绍的代码，这里就不再重复了。这个页面上的文本框控件比较独特，用于显示存储在 XML 设置文件中的 NewsSubject 和 Signature 属性的值。它们实际上是由后台编码文件获取，并发送回页面的。

2. SendEmail.aspx 的后台编码文件

这个页面没有使用数据网格显示数据，所以这个后台编码文件与前面两个文件大不相同。首先看看 Page_Load()代码，其中与前面文件不同的代码突出显示：

```
// Two variables used to store the configuration settings
public string Signature_Value;
public string Subject_Value;

protected void Page_Load(object sender, EventArgs e)
{
    string ListID = Request.Params["ListID"];

    if (!Page.IsPostBack)
    {
        // load all the lists name in the DropDown control
        NewsMailer.ListsDB lists = new NewsMailer.ListsDB(GetConnString());
        DataView myDV = lists.GetLists().Tables[0].DefaultView;
        ListsDropDown.DataSource = myDV;

        // load the signature and news subject from the DB
        NewsMailer.AppSettings settings =
        NewsMailer.AppConfig.GetSettings(Server.MapPath(GetSettingsFile()));
        Signature_Value = settings.Signature;
        Subject_Value = settings.NewsSubject;
        Page.DataBind();

        if (ListID!=null)
        {
            foreach (ListItem item in ListsDropDown.Items)
              if (item.Value==ListID)
                    ListsDropDown.SelectedIndex =
                    ListsDropDown.Items.IndexOf(item);
        }
    }
}
```

NewsMailerConfig 装配件用于从 XML 设置文件中获取所有的设置，签名和新闻标题存储在公共变量中。

注意设置文件名是从 web.config 文件中获取的，或者由 BaseClass 的 GetSettingsFile()函

数从缓存中获取，Server.MapPath()调用把返回值转换为真实的路径，该调用获取了文件的完整路径。

执行 Page.DataBind()(该方法把窗体的所有子控件绑定到数据上)时，两个变量 Signature_Value 和 Subject_Value 的值绑定并显示在文本框中。

下面是 Preview 按钮的代码：

```
protected void Preview_Click(object sender, EventArgs e)
{
    string MsgText = Body.Text;
    int ListID = int.Parse(ListsDropDown.SelectedItem.Value);

    // replace the list's and settings special tags with their values
    MsgText = NewsMailer.Utilities.ProcessListTags(MsgText,
                                                   GetConnString(), ListID);
    MsgText = NewsMailer.Utilities.ProcessSettingsTags(MsgText,
                                                   GetConnString());

    // encode the body if the format is not HTML
    if (FormatDropDown.SelectedIndex==0)
        MsgText = HttpUtility.HtmlEncode(MsgText);
    // replace the new lines with <br>
    MsgText = MsgText.Replace("\n", "<br>");
    // remove carriage returns
    MsgText = MsgText.Replace("\r", "");

    MsgPreview.Text = MsgText;
    MsgPreview.Visible = true;
}
```

这些代码调用 Utilities 类的一些函数(稍后介绍)，用它们各自的值替代特定的标记，如果选中的格式是纯文本格式，就对文本编码(例如<b>编码为<b>)，用
标记替代换行符，删除回车符，因为 HTML 不能识别'\n' 和 '\r'字符。最后，在预览标签中显示信息体。

下面是 Send Email 按钮的代码：

```
protected void Send_Click (object sender, EventArgs e)
{
    Server.ScriptTimeout = 3600;

    string MsgBody = Body.Text;
    int ListID = int.Parse(ListsDropDown.SelectedItem.Value);
    string MsgSubject = NewsMailer.Utilities.ProcessListTags(
                                    Subject.Text, GetConnString(), ListID);

    NewsMailer.AppSettings settings =
    NewsMailer.AppConfig.GetSettings(Server.MapPath(GetSettingsFile()));
```

```
        NewsMailer.NewsHistoryDB news = new NewsMailer.NewsHistoryDB(
                                                    GetConnString());

        // replace the list's and settings special tags with their values
        MsgBody = NewsMailer.Utilities.ProcessListTags(MsgBody,
                                                    GetConnString(), ListID);
        MsgBody = NewsMailer.Utilities.ProcessSettingsTags(MsgBody,
                                                    GetConnString());

        // add a new record in the news archive
        news.Add(ListID, MsgSubject,MsgBody,(FormatDropDown.SelectedIndex==1));

        // create a new mail message
        MailMessage MailMsg = new MailMessage();
        MailMsg.From = settings.SenderName + " <" + settings.SenderEmail + ">";
        MailMsg.Cc = CC.Text;
        MailMsg.Bcc = BCC.Text;
        MailMsg.Priority = (MailPriority)PriorityDropDown.SelectedIndex;
        MailMsg.BodyFormat = (MailFormat)FormatDropDown.SelectedIndex;
        MailMsg.Subject = MsgSubject;

        NewsMailer.SubscribersDB subscribers = new NewsMailer.SubscribersDB(
                                                    GetConnString());
        // get all the subscribers of this list
        DataTable dt = subscribers.GetSubscribers(ListID).Tables[0];
        foreach (DataRow dr in dt.Rows)
        {
            // replace the subscriber's special tags with their values
            MailMsg.To = (string)dr["Email"];
            MailMsg.Body = NewsMailer.Utilities.ProcessSubscriberTags(MsgBody,
                                            GetConnString(), (int)dr["ID"]);
            // send the mail to this subscriber
            SmtpMail.Send(MailMsg );
        }

        // show the number of sent mails
        MsgPreview.Text = string.Format(
                        "<center><b>{0} mails have been sent</b></center>",
                        dt.Rows.Count.ToString());
        MsgPreview.Visible = true;
    }
```

这个例程比较长，但它的执行方式与 Preview_Click 类似，下面的它的执行步骤：

- 列表和一般设置的特定标记用它们的值替代，如果需要，对文本编码。
- 使用 NewsHistory 的 Add()方法，把信息添加到归档文件中。
- 创建一个新 MailMessage 对象，设置对所有订阅者都相同的属性。

- 使用一个 SubscribersDB 类实例，获取当前列表的所有订阅者。
- 对于由查询返回的每个订阅者，把它的电子邮件地址设置为邮件对象的目的地地址，并发送邮件。
- 在预览标签中显示已发送信息的总数。

在这段代码中，有几点要注意。首先，除了为其他页面声明的公共名称空间外，这次还必须声明 System.Web.Mail 名称空间，这个名称空间包含的类可以通过随 Windows 2000 Server 一起安装的默认 SMTP 服务，建立和发送电子邮件。

另一个要注意的是第一行代码：

```
Server.ScriptTimeout = 3600;
```

发送一个信息是很快的，但如果有上千个订阅者，就需要一定的时间，几秒或者更多。已经熟悉传统 ASP 编程的用户都知道，在默认情况下，页面有 90 秒的时间执行脚本代码，把所有已生成的 HTML 代码发送给浏览器。IIS 管理员可以修改这个默认值，也可以由用户把一个值硬编码到页面中——这就是我们要在这里做的。把完成执行脚本代码的最长时间设置为 1 个小时。

27.4.5 Utilities 类

在 SendEmail.cs 后台编码文件中，使用 Utilities 类替代主体和标题的特定标记。表 27-13 是这个类中方法的完整列表。注意它们都是静态方法，这表示使用它们时，不必创建这个类的实例。

表 27-13

Utilities 类

方　法	描　述
public static string ReadTextFile(string FilePath)	获取指定文本文件的内容
public static bool SaveTextFile(string FilePath, string FileText)	把 FileText 字符串保存到指定的文件中，返回一个布尔值，表示操作的成功或失败
public static string ProcessListTags(string Message, string ConnString, int ListID)	用数据库中的值替代用#分隔的列表标记
public static string ProcessSubscriberTags(string Message, string ConnString, int SubscrID)	用数据库中的值替代用#分隔的订阅者标记
public static string ProcessSettingsTags(string Message, string SettingsFilePath)	用数据库中的值替代用#分隔的设置标记

可以参阅下载的代码，了解这些方法的执行情况。

27.4.6 管理新闻归档文件

新闻归档页面显示了邮件列表的所有旧新闻，还显示了纯文本或 HTML 格式的所有旧信息，这取决于发送它们时所使用的格式。图 27-16 是页面的屏幕图。

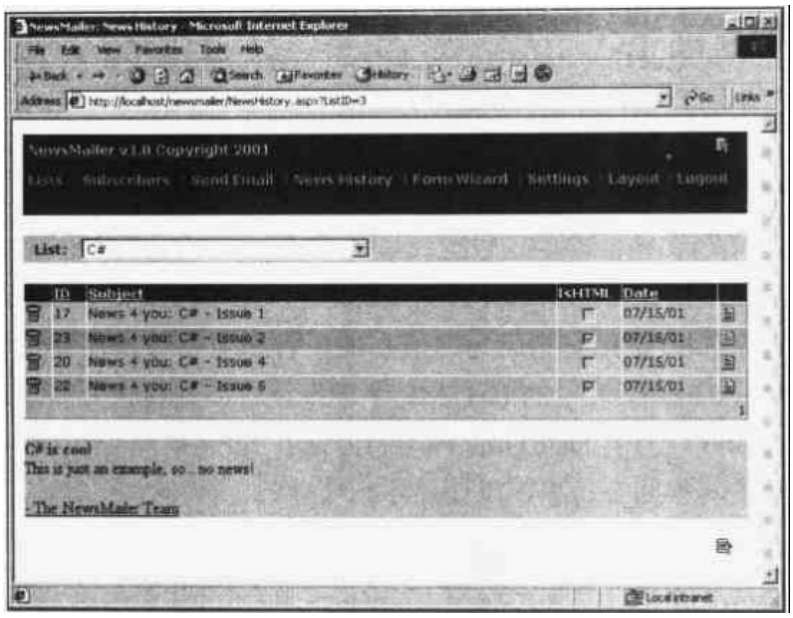


图　27-16

1. NewsHistory.aspx 页面

这个页面使用数据网格来显示数据，但有一些独特的功能。注意选中的行与其他行的颜色不同。还要注意 IsHTML 列上有一个复选框，表示输出结果是否为 HTML 格式，Date 字段表示文章的发送时间。该页面的许多代码都与以前的页面的代码相同，所以这里没有列出所有的代码。下面的代码定义了 IsHTML 和 Date 列：

```
<Columns>
... Code removed for clarity

<asp:BoundColumn HeaderText="Subject" DataField="Subject"
    ReadOnly="True" SortExpression="Subject" />

<asp:TemplateColumn HeaderText="IsHTML"
    ItemStyle-Width="60px" ItemStyle-HorizontalAlign="Center">
    <ItemTemplate>
        <asp:CheckBox runat="server" Enabled="false"
            Checked='<%#
            (bool)DataBinder.Eval(Container.DataItem, "IsHTML")
             %>'
        />
    </ItemTemplate>
</asp:TemplateColumn>

<asp:TemplateColumn HeaderText="Date"
    SortExpression="SentDate"   ItemStyle-Width="90px">
    <ItemTemplate>
        <asp:Label runat="server"
```

```
                Text='<%# string.Format("{0:MM/dd/yy}",
                    DateTime.Parse(DataBinder.Eval(Container.DataItem,
                    "SentDate").ToString()) ) %>'
            />
        </ItemTemplate>
    </asp:TemplateColumn>
```

在 Send Email 页面中，我们把一个公共变量绑定到文本框上，这里 IsHTML 列是一个只读列，包含一个禁用的复选框(禁用它是因为旧信息的内容不能由管理员通过这个页面来修改)，其值根据数据库的 NewsHistory 表中的 IsHTML 字段来设置，并转换为布尔值。

要获取日期值，可以使用 DateTime.Parse()方法，把数据库返回的字符串转换为一个 DateTime 值，再把该值转换回字符串，以便显示。

2. NewsHistory.aspx 的后台编码文件

对于这段代码的解释很短，因为这段代码执行的大多数操作都类似于其他后台编码文件中已讨论的操作。惟一要讨论的新功能是处理数据网格的 OnSelectedIndexChanged 事件的过程。选中一个数据行后，该过程就获取该信息的主体，如果该信息为纯文本，就对它编码，并显示在预览标签中：

```
protected void NewsGrid_SelectionChanged(object sender, EventArgs e)
{
    string Body;
    NewsMailer.NewsHistoryDB news = new NewsMailer.NewsHistoryDB(
        GetConnString());

    // get the ID of the selected row
    int NewsID = int.Parse(
        NewsGrid.DataKeys[NewsGrid.SelectedIndex].ToString());
    // get the Body of the news from the DB
    Body = news.GetDetails(NewsID)["Body"].ToString();

    // if the news format was not HTML, encode the Body text
    if (!Convert.ToBoolean(news.GetDetails(NewsID)["IsHTML"]))
        Body = HttpUtility.HtmlEncode(Body);

    // replace the new lines with <br>, to show the new lines in the Label
    Body = Body.Replace("\n", "<br>");
    Body = Body.Replace("\r", "");

    // show the Body text in the Preview label
    MsgPreview.Text = Body;
    MsgPreview.Visible = true;
}
```

注意这里不需要替换任何特殊标记，因为在把信息添加到数据库的新闻归档文件中之前，就已经替换过它们了。

另外，管理员修改另一个网格页面时，必须取消对选中行的选择，因为这不是自动取消的：

```
protected void NewsGrid_PageChanged(Object sender,
    DataGridPageChangedEventArgs e)
{
    UnselectGridItem();
    // change the current page and reding the data
    NewsGrid.CurrentPageIndex = e.NewPageIndex;
    BindGrid();
}

protected void UnselectGridItem()
{
    // hide the Preview control and unselect the selected row
    NewsGrid.SelectedIndex =  - 1;
    MsgPreview.Text = "";
    MsgPreview.Visible = false;
}
```

27.4.7　在线修改设置

到目前为止，我们使用了存储在 NewsMailer.Settings XML 文件中的一些值，在创建文件时设置它们。Web 页需要对这些设置进行在线修改，而不必坐在 Web 服务器旁边，手动修改数据库。页面如图 27-17 所示。

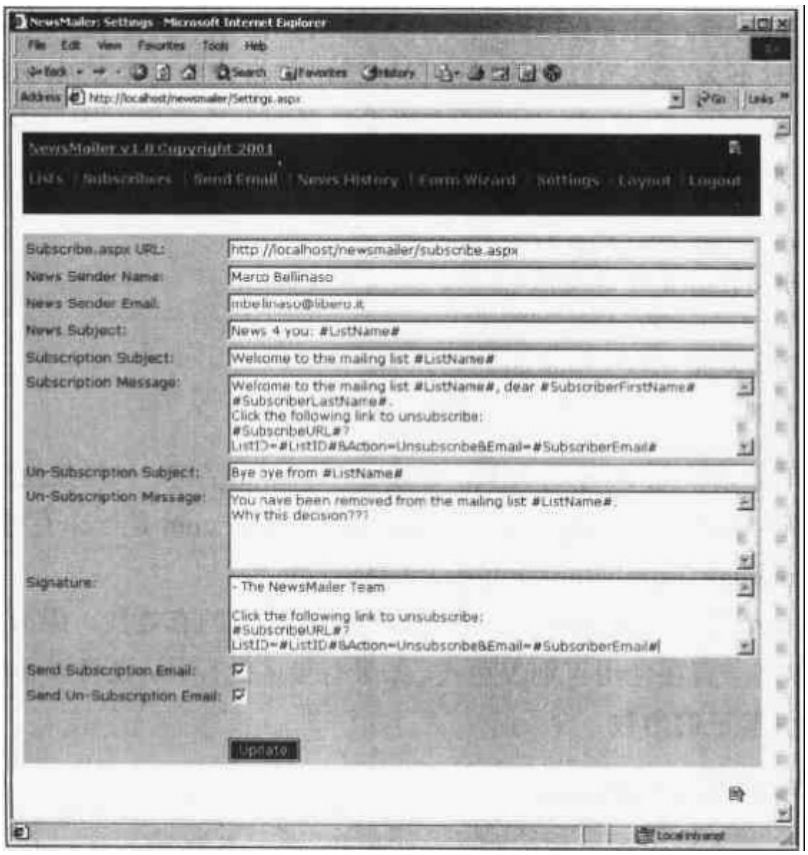


图　27-17

1. Settings.aspx 页面

ASP.NET 页面由一组控件组成，这些控件绑定到包含在 Settings 数据库中的各个设置上。而这些设置绑定到后台编码文件中设置的公共变量中，如前面的 Send Email 页面所述。该页面相当大，但从显示的角度来看，下面的部分代码定义了 Sender Email 字段，并填充了相应的文本框，还进行了有效性验证：

```
<asp:TableRow>
    <asp:TableCell Text="News Sender Email:" />
    <asp:TableCell>
        <asp:TextBox runat="server" Width="500px" ID="SenderEmail"
            Class="TextBox" Text='<%# SenderEmail_Value %>' />
        <asp:RequiredFieldValidator runat="server"
            ControlToValidate="SenderEmail"
            Display="dynamic"><br>* This field is required
        </asp:RequiredFieldValidator>
        <asp:RegularExpressionValidator runat="server"
              validationExpression=".*@.*\..*"
              ControlToValidate="SenderEmail"
              Display="dynamic"><br>* This Email address is not valid
         </asp:RegularExpressionValidator>
    </asp:TableCell>
</asp:TableRow>
```

上面的控件比较特殊，因为它包含两个有效性验证控件，而不是一个。第一个控件确保文本框在提交窗体时不为空，第二个控件确保字符串的格式为有效的电子邮件地址。后一个控件使用了 RegularExpressionValidator，顾名思义，正则表达式检查控件内容的有效性。在这个例子中，表达式如下：

```
<asp:RegularExpressionValidator runat="server"
    validationExpression=".*@.*\..*"
    ControlToValidate="SenderEmail"
    Display="dynamic"><br>* This Email address is not valid
</asp:RegularExpressionValidator>
```

这表示，字符串必须以几个*字符开头，必须包含一个@字符、一些其他字符和一个句点(转义为\.)，最后是另外一些字符。例如，mbellinaso@vb2themax.com 是一个有效的电子邮件地址，而 mbellinaso@vb2themax 或 mbellinaso.libero.it 是错误的地址。

正则表达式是一个非常强大的工具，可以检查控件内容的有效性，因为它们可以非常详细和复杂，实际上，本书一直在使用正则表达式。如果有更多的控件，还可以使用 CustomValidator 编写一个进行有效性验证的函数。

2. Settings.aspx 的后台编码文件

修改设置的所有后台编码文件都由两个过程组成：第一个是 Page_Load()，它使用 AppConfig

和 AppSettings 对象从数据库中获取设置，通过调用 Page.DataBind()绑定不同的服务器控件：

```
protected void Page_Load(object sender, EventArgs e)
{
    if (!IsPostBack)
    {
        // load all the settings
        NewsMailer.AppSettings settings =
            NewsMailer.AppConfig.GetSettings(
                Server.MapPath(GetSettingsFile()));

        SubscribeURL_Value = settings.SubscribeURL;
        SenderName_Value = settings.SenderName;
        SenderEmail_Value = settings.SenderEmail;
        NewsSubject_Value = settings.NewsSubject;
        SubscrSubject_Value = settings.SubscrSubject;
        SubscrMessage_Value = settings.SubscrMessage;
        UnSubscrSubject_Value = settings.UnSubscrSubject;
        UnSubscrMessage_Value = settings.UnSubscrMessage;
        Signature_Value = settings.Signature;
        SendSubscrEmail_Value = settings.SendSubscrEmail;
        SendUnSubscrEmail_Value = settings.SendUnSubscrEmail;

        Page.DataBind();
    }
}
```

第二个过程是 Update_Click()，单击 Update 按钮时调用该过程。它从控件中获取新设置，设置新 AppSettings 对象的属性，最后调用 AppConfig 的 SaveSettings()方法，把 settings 对象串行化为 XML 文件(本例中是 NewsMailer.Settings)：

```
protected void Update_Click(object sender, EventArgs e)
{
    NewsMailer.AppSettings settings = new NewsMailer.AppSettings();
    // set the new properties values and update everything
    settings.SubscribeURL = SubscribeURL.Text;
    settings.SenderName = SenderName.Text;
    settings.SenderEmail = SenderEmail.Text;
    settings.NewsSubject = NewsSubject.Text;
    settings.SubscrSubject = SubscrSubject.Text;
    settings.SubscrMessage = SubscrMessage.Text;
    settings.UnSubscrSubject = UnSubscrSubject.Text;
    settings.UnSubscrMessage = UnSubscrMessage.Text;
    settings.Signature = Signature.Text;
    settings.SendSubscrEmail = SendSubscrEmail.Checked;
```

```
        settings.SendUnSubscrEmail = SendUnSubscrEmail.Checked;
        NewsMailer.AppConfig.SaveSettings(
                                Server.MapPath(GetSettingsFile()), settings);
    }
```

27.4.8　在线修改布局

本章开头说过，所有控件的所有样式都放在一个单独的样式表中，而不是放在页面中。实际上，无论在何时进行修改操作，对颜色和其他布局样式硬编码总是一件非常麻烦的事情，因为必须手动检查每个页面，替换要修改的所有样式。使用一个样式表就简单多了，样式表可以在线编辑，无需在本地计算机上修改文件，再把该文件上传回服务器。执行页面，在线编辑任何文本文件都只需几分钟，也只需几行代码，所以下面就使用样式表。

图 27-18 所示的屏幕图显示了一个简单的样式表编辑器。

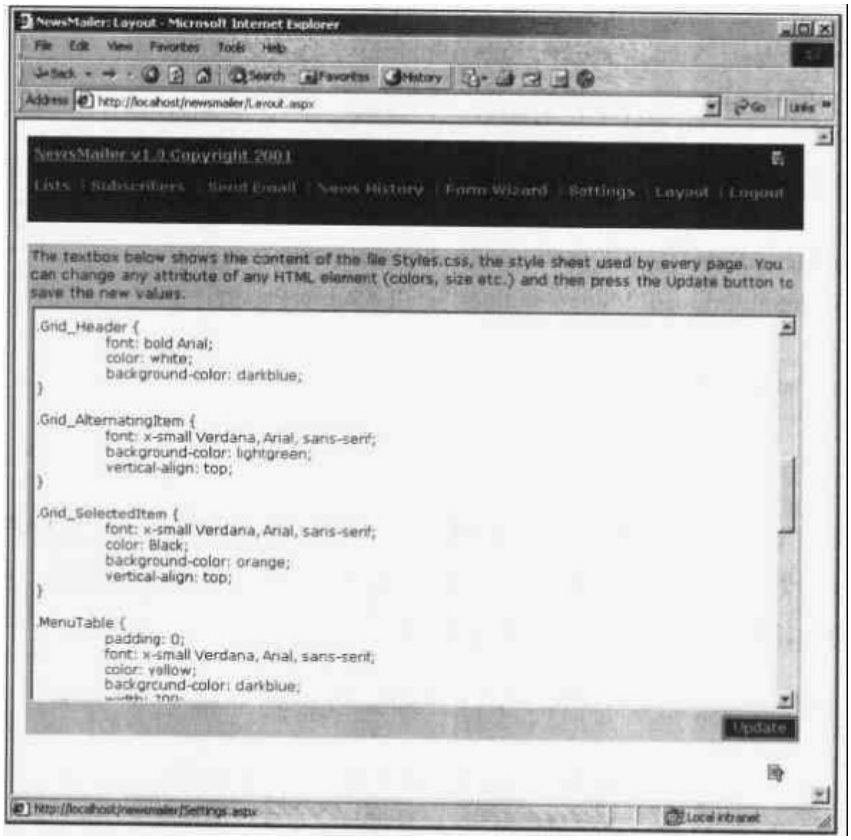


图　27-18

如果读者知道 CSS，就可以修改 ASP.NET 控件的样式，重新设置用户界面的全部样式只需几秒钟。

1. Layout.aspx 页面

这个文件中只有 ASP.NET 控件的代码，所以这里没有给出全部代码。但下面这行代码值得深入研究：

```
<asp:Label ID="FileName" runat="Server" Text="Styles.css" Visible="false" />
```

这行代码声明了一个隐藏的标签，它用于设置要编辑的文件名，这意味着通过修改这个设置就可以编辑任何文本文件。管理员在修改了隐藏控件中的路径后，仍需要重新上传页面，但这要比修改后台编码文件中的值简单得多，也快得多。因为这需要重新编译和重新上传整个装配件。记住，应以编译好的格式发布应用程序，如果硬编码了一个设置，远程工作的站点管理员就不能在没有开发人员的帮助下修改它。如果有源代码，就需要重新编译这些源代码。所以最佳的解决方案是把需要的设置放在 ASP.NET 页面的隐藏字段中，如上所述。

2. Layout.aspx 的后台编码文件

这个页面的代码读取文件的内容，在 Page_Load()事件过程中把它绑定到文本框中，在 Update_Click()事件过程中读取文本框的内容，把它保存到指定的文件中。注意类级变量 FileContent_Value 用于存储文件中的文本：

```
public string FileContent_Value;
void Page_Load(object sender, EventArgs e)
{
    if (!Page.IsPostBack)
    {
        // read the content of the specified file
        FileContent_Value = NewsMailer.Utilities.ReadTextFile(
            Server.MapPath(FileName.Text));
        Page.DataBind();
    }
}

void Update_Click(object sender, EventArgs e)
{
    // save the file with the new text
    NewsMailer.Utilities.SaveTextFile(
    Server.MapPath(FileName.Text), FileContent.Text);
}
```

还要注意读写文本文件的代码在 Utilities 类中，如前所述。ReadTextFile() 和 SaveTextFile() 方法都是静态的，使用它们时不必创建 Utilities 类的实例。

27.4.9　为用户创建订阅窗体

管理邮件列表和订阅者、发送电子邮件信息和修改布局的页面都介绍过了。还有什么？订阅窗体，当然它允许终端用户订阅一个列表。

下面没有创建一个订阅窗体，而是创建了一个 ASP.NET 页面，为指定的邮件列表自动生成一个 HTML 窗体，如图 27-19 所示。

管理员选择要创建订阅窗体的邮件列表，指定是否需要姓名字段(这些字段不是订阅所必需的，所以没有它们窗体也可以工作)，窗体上有两个选项按钮，允许用户订阅和取消订阅。单击 Create 按钮后，就会生成 HTML 代码，并放在底部的文本框中，在页面的中间显示窗体的预览。

此时，管理员可以复制 HTML，并粘贴到任意 HTML 页面上。

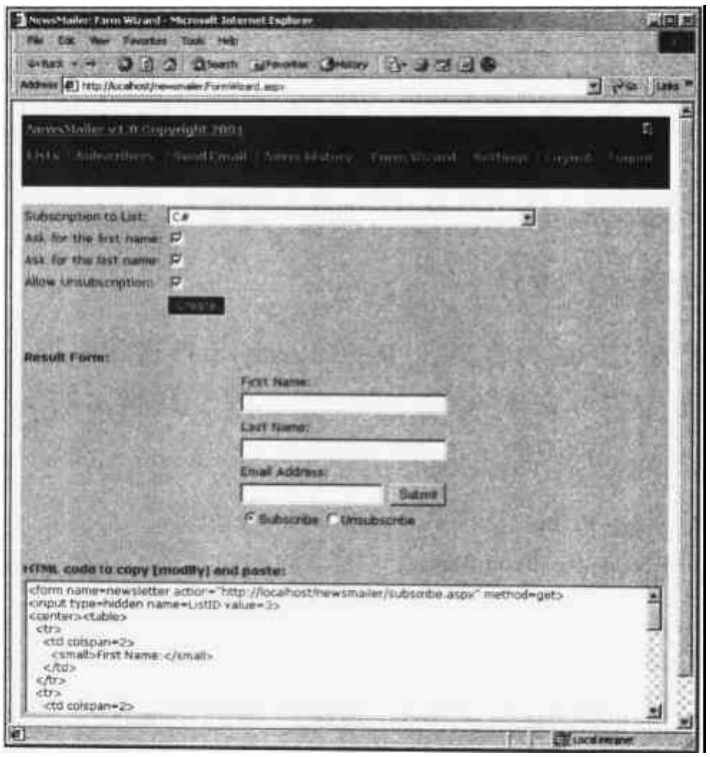


图　27-19

1. FormWizard.aspx 页面

页面的第一部分包含下拉列表和复选框，建立过程与往常一样，并通过一个表格对齐控件。在上面显示的表格中，标有“Result Form:”的第二部分在默认情况下是不可见的，当用户单击 Create 按钮后，这部分才显示出来。这段代码几乎全部由 ASP.NET 控件组成，与前面的代码类似，所以这里没有列出代码。

2. FormWizard.aspx 的后台编码文件

这个页面的后台编码只使用数据库，以可用的邮件列表填充下拉列表，从 Settings 表中获取 Subscribe.aspx 页面的 URL，其余代码根据选中的选项来编写窗体的 HTML。Create_Click() 事件处理程序建立一个 HTML 字符串，其内容取决于管理员在窗体的顶部选择的选项。这段代码没有什么新东西或有趣的内容，这里就不重复了。读者可以从下载的代码中查看它们。

由 Create_Click()事件处理程序创建的 HTML 代码显示在 ResultForm 标签中，允许用户预览结果。这些 HTML 代码并不完整，因为其根元素是<form>标记，但按照需要，很容易把它们从文本框复制到一个单独的 HTML 文件中。

在图 27-20 所示的屏幕图中，显示了由这个页面生成的几个测试窗体。

其中的第一个窗体是把 3 个选项都设置为未选中而生成的。第二个窗体则选中了前两个选项(要求输入姓名)，第 3 个窗体则选中了全部 3 个选项。

显然，因为生成的代码是简单的 HTML，所以可以按照需要定制结果。例如，可以改变颜色、字体、文本框的宽度、添加一个图像作为边框或背景，或者添加一个注释，告诉客户关于所订阅的邮件列表名。一个非常有用的操作是添加一个 JavaScript 函数，以确保电子邮件地址

字段在提交窗体前不为空。

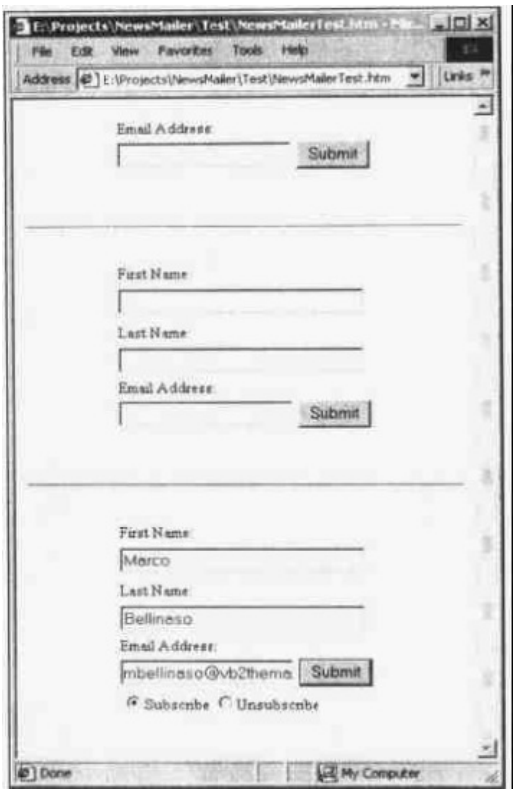


图　27-20

27.4.10　订阅邮件列表

到目前为止，我们都是从管理员的角度介绍应用程序的，那么终端用户呢？访问该 Web 站点并想订阅邮件列表的客户呢？他们要做什么工作？下面就讨论这个问题。带有订阅窗体的页面可以放在任何地方，甚至可以放在另一个服务器上，因为它指向 Subscribe.aspx 的 URL。

现在该测试一下图 27-20 中的订阅窗体页面，订阅一个新用户。我们填写图 27-20 中的第二个窗体。如果在管理员页面上选择 Send Subscription Email 设置(如果使用前面提供的设置文件，它就是默认选项)，订阅者将接收到用他们的信息定制的确认电子邮件信息，如图 27-21 所示。

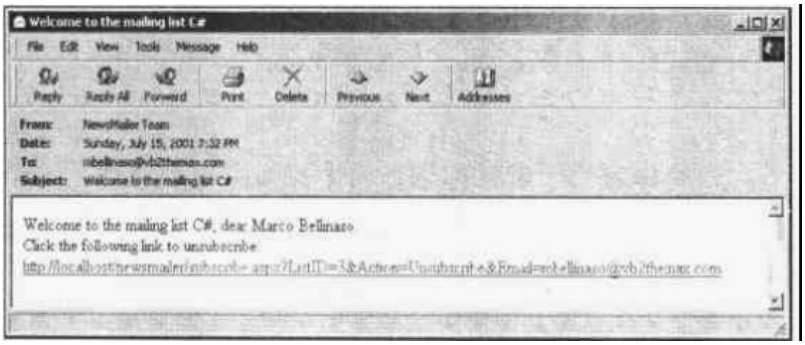


图　27-21

最后一行是一个链接，如果单击该链接，就会从邮件列表中删除这个订阅者。这个链接将替换下述字符串的标记，自动建立：

#SubscribeURL#?ListID=#ListID#&Action=Unsubscribe&Email=#SubscriberEmail#

参阅 XML 设置文件中的相应值。

从读取的字符串中可以猜出，订阅和取消订阅的区别是 Action 参数，在本例中，它设置为 Unsubscribe。订阅的链接使用相同的字符串，但没有 Action 参数，或者把参数设置为除 Unsubscribe 之外的其他字符串。链接中的另一个参数是邮件列表的 ID(存储在窗体的隐藏字段中)和订阅者的电子邮件地址。

只有启用了相应的设置后，才会发送确认电子邮件。但是用户总是会接收到一个确认 Web 页，其中包含与邮件相同的文本。对于取消订阅，所有的内容都一样：管理员可以选择发送一个确认信息，但站点总是会显示一个 Web 页。要测试这个操作，可以打开前面电子邮件信息中的链接，于是打开一个浏览器实例，并显示信息，如 27-22 所示。

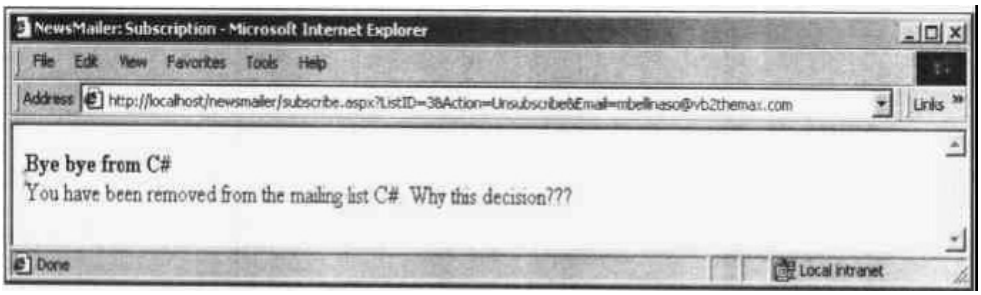


图　27-22

在 HTML 窗体上选择 Unsubscribe 选项，会得到相同的结果。显然，管理员也可以使用 Subscribers.aspx 页面添加或删除订阅者。

1. Subscribe.aspx 页面

这个页面的设计代码是本章中最短的：

```
<body>
    <asp:Label ID="Subject" runat="Server"
        CssClass="SubscrSubject" Visible="false" />
    <asp:Label ID="Message" runat="Server"
        CssClass="SubscrMessage" Visible="false" />
    <asp:Label runat="server" ID="EmailRequired" Visible="false"
        Text="The Email address is required"   Class="Error" />
</body>
```

其中有 3 个标签：一个标签用于确认标题，一个标签用于信息体，一个标签用于因没有指定电子邮件地址而显示的错误信息。这 3 个标签在默认情况下都不可见。在用户订阅或取消订阅后，根据操作的结果，它们就会显示出来。

2. Subscribe.aspx 的后台编码文件

这个 codebehind 类的所有代码都位于 Page_Load()过程中，因为在加载页面后，所有的工作都完成了，没有任何窗体或控件引发其他事件。这个事件处理程序的完整代码如下所示：

```
protected void Page_Load(object sender, EventArgs e)
{
    int SubscrID;
    bool SendEmail;
    // get the ListID, FirstName, LastName, Email and Action
    // from the QueryString generated by the HTML form
    int ListID = int.Parse(Request.Params["ListID"]);
    string FirstName = Request.Params["FirstName"] + "";
    string LastName = Request.Params["LastName"] + "";
    string Email = Request.Params["Email"] + "";
    string Action = Request.Params["Action"] + "";

    if (Email.Trim()=="")
    {
        // if the Email was not specified,
        // show the error message and do not proceed
        EmailRequired.Visible = true;
    }
    else
    {
        string SubjectText;
        string MessageText;

        // create the objects to manage the settings and the subscriber
        NewsMailer.AppSettings settings =
            NewsMailer.AppConfig.GetSettings(
                Server.MapPath(GetSettingsFile()));

        NewsMailer.SubscribersDB subscribers = new
                NewsMailer.SubscribersDB(GetConnString());

        // unsubscribe user
        if (Action.CompareTo("Unsubscribe")==0)
        {
            // get the subject/body of the subscription message
            // and see if a confirmation Email is required
            SendEmail = settings.SendUnSubscrEmail;
            SubjectText = settings.UnSubscrSubject;
            MessageText = settings.UnSubscrMessage;
            // get the ID of the subscriber with this Email
            SubscrID = subscribers.GetSubscrID(ListID, Email);
            // replace the tags of this subscriber
            MessageText = NewsMailer.Utilities.ProcessSubscriberTags(
                MessageText, GetConnString(), SubscrID);
            // delete the subscriber
```

```
        subscribers.Delete(SubscrID);
    }
    else      // subscribe user
    {
        // get the subject/body of the unsubscription message
        // and see if a confirmation Email is required
        SendEmail = settings.SendSubscrEmail;
        SubjectText = settings.SubscrSubject;
        MessageText = settings.SubscrMessage;
        // add the subscriber
        subscribers.Add(ListID, FirstName, LastName, Email);
        // get the ID of the subscriber just added,
        // or of the subscriber already present
        // with the specified Email address
        SubscrID = subscribers.GetSubscrID(ListID, Email);
        // replace the tags of this new subscriber
        MessageText = NewsMailer.Utilities.ProcessSubscriberTags(
            MessageText, GetConnString(), SubscrID);
    }

    // replace the list's and settings tags
    SubjectText = NewsMailer.Utilities.ProcessListTags(
        SubjectText, GetConnString(), ListID);
    MessageText = NewsMailer.Utilities.ProcessListTags(
        MessageText, GetConnString(), ListID);
    MessageText = NewsMailer.Utilities.ProcessSettingsTags(
        MessageText, GetConnString());

    // if a confirmation Email is required
    if (SendEmail)
    {
        // create a new message
        MailMessage MailMsg = new MailMessage();
        MailMsg.From = settings.SenderName + " <" +
            settings.SenderEmail + ">";
        // set the "To" field to the subscriber's Email
        MailMsg.To = Email;
      // set the Subject/body to the subject/body
        // created for the confirmation page
        MailMsg.Subject = SubjectText;
        MailMsg.Body = MessageText;
        // send the mail
        SmtpMail.Send(MailMsg );
    }
```

```
            // show the message that confirms the action
            Subject.Text = SubjectText;
            Message.Text = MessageText;
            Subject.Visible = true;
            Message.Visible = true;
        }
    }
```

在第一行代码中，目标列表的 ID、订阅者的姓名和地址从查询字符串中被提取出来，并保存在局部变量中，这样在后面的过程中就可以方便地使用它们。在继续执行其他操作前，代码检查电子邮件参数是否为空，如果是，就显示 EmailRequired 错误标签，且不执行其他过程。如果提供了电子邮件地址，且 Action 是 Unsubscribe，则代码首先设置信息标题和主体的确认文本，用订阅者的信息替换特定的标记，再从数据库中删除订阅者。订阅者也可能添加到数据库中，再替换标记。标记的替换是后一步完成，因为 Utilities.ProcessSubscriberTags()方法需要把订阅者的 ID 作为一个参数，所以新订阅者必须先存储在数据库中，然后替换邮件列表和设置的特定标记，无论用户是订阅还是取消订阅，这都是必须的。如果设置文件表示应发送一个确认电子邮件(包含标题和主体)，该过程就发送这样一个邮件。

27.4.11　保证应用程序的安全性

现在，这个应用程序的功能都编写全了，也从管理员和终端用户的角度测试了该应用程序，它都工作正常。还剩下什么？安全性。如果不执行任何类型的登录处理，控制管理页面的访问，任何人都可以使用这些页面盗取订阅者的地址，发送假邮件，删除邮件列表等。在设计任何公共 Web 应用程序时，安全性是在设计阶段必须考虑的首要问题。但对于像本章这样的小程序来说，安全性和用户身份认证可以在最后一步进行，没有任何问题，也非常方便。

Web.config 中的安全性设置

在讨论存储连接字符串时，首先遇到的就是 web.config 文件。这个配置文件可以用于存储 Web 页、会话状态管理、应用程序(与前面插入的 ConnString 和 SettingsFile 一样)、跟踪、编译、安全性的设置和其他设置。这个文件非常有用，因为它可以重写 IIS 的设置。下面介绍这个文件中指定的安全性设置。ASP.NET 和这个配置文件提供了一种简便的方式，以实现灵活的用户身份认证(即用户是否能访问该站点)和授权(授予或剥夺用户某些访问权限)。下面的代码完成了大部分工作：

```
<configuration>
    <system.web>
        <authentication mode="Forms">
            <forms name=".NEWSMAILER" loginUrl="Login.aspx"
                protection="All" timeout="60">
                <credentials passwordFormat="Clear" >
                    <user name="Admin" password="AdminOK"/>
                    <user name="Marco" password="MarcoOK"/>
```

```
            </credentials>
        </forms>
    </authentication>

    <authorization>
        <deny users="?" />
    </authorization>
</system.web>

<appSettings>
    <add key="connstring" value="Provider=Microsoft.Jet.OLEDB.4.0;
        Data Source=E:\Projects\NewsMailer\DB\NewsMailer.mdb" />
</appSettings>
</configuration>
```

Forms 模式下的身份认证是指使用一个 Web 窗体进行登录(其他选项有 HTTP Basic 系统和 Microsoft Passport 系统)。<authentication>部分中的代码行指定了登录页面的 URL，如果用户试图访问一个受保护的页面，但没有通过身份认证，该页面就会把用户重定向到其他页面上。Cookie 用于保存登录选项(.NEWSMAILER)和允许访问站点的用户凭证。这个例子的凭证是用户名和密码的一个组合。最后，<authorization>中的<deny users="?" />标记指定匿名用户不允许进入应用程序中受保护的部分。

当然，ASP.NET 中的安全问题要复杂和专业得多，这仅是一个基本用户身份认证和授权系统的例子。如果要使用 web.config 文件，还可以用 Triple DES 对密码加密。另一个选项是把用户名和密码保存在一个数据库表中，编写一个定制函数，检查用户传送过来的登录信息是否有效，或者映射 Windows Active Directory 中的用户。但这些都超出了本章的范围。要了解更多的信息，可参阅.NET Framework SDK 文档说明，或者清华大学出版社的《ASP.NET 高级编程》。

下面介绍 Login.aspx 文件和后台编码的内容。

现在需要编写 Login.aspx 页面，完成后，其页面应如图 27-23 所示。

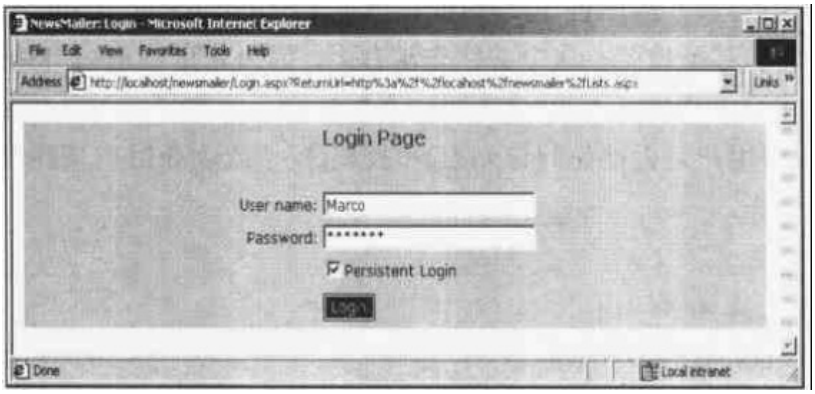


图　27-23

设计该页面的代码没有什么特别之处，前面已经介绍了其他页面的更复杂的代码，所以下面直接讨论处理单击 Login 按钮的代码：

```
protected void LoginUser_Click(object sender, EventArgs e)
```

```
{
    // check the specified user name and password
    if (FormsAuthentication.Authenticate(UserName.Text, Password.Text))
    {
        // if the user is authenticated, make the login persistent
        // (if required by the user) and jump to the page
        // that required authentication
        FormsAuthentication.RedirectFromLoginPage(UserName.Text,
            Persistent.Checked);
    }
    else
    {
        // if the user name or password is wrong, show the error label
        InvalidLogin.Visible = true;
    }
}
```

代码调用 Authenticate()方法，验证用户名和密码的有效性。如果它们无效，就显示 InvalidLogin 标签，否则登录信息就存储到 cookie 中(如果选中了 Persistent Login 复选框)，且浏览器跳转到用户要求登录的页面上。将该页面的 URL 传送到查询字符串中，在上图的地址栏中可以看到该页面的 URL。这个 URL 是 localhost/newsmailer/Lists.aspx，且已经编码。所以，如果现在用户试图加载一个页面，但没有通过身份认证，该用户就会被重定向到登录页面上，在通过身份认证之后，再发送回最初的页面。

应用程序现在似乎已经完成了，但还有一个小问题。当客户发送该窗体，以订阅一个列表时，该怎么办？这个用户没有进行身份认证，会被重定向到登录页面上。这当然是不会发生的，但该如何解决这个问题？一种解决方案是把 Subscribe.aspx 页面移动到另一个目录下，在该目录下，匿名用户可以加载该页面，这个解决方案肯定奏效，但这样就需要两个 web.config 文件，一个存储了根文件夹下的连接字符串，另一个存储了受保护的文件夹下的安全设置。还有一种更简单的解决方案，惟一的要求是在已有的 web.config 文件中添加几行代码，允许把 Subscribe.aspx 页面和其他页面放在一起。使用<location>标记，可以为不同的文件设置不同的安全选项，再让匿名用户只获得 Subscribe.aspx 页面：

```
<configuration>
    <system.web>

...Code omitted for clarity

    </system.web>

    <location path="Subscribe.aspx">
        <system.web>
            <authorization>
```

```
            <allow users="*" />
          </authorization>
        </system.web>
      </location>

      <appSettings>
        <add key="connstring" value="Provider=Microsoft.Jet.OLEDB.4.0;
          Data Source=E:\Projects\NewsMailer\DB\NewsMailer.mdb" />
      </appSettings>
    </configuration>
```

现在，当用户发送窗体，加载 Subscribe.aspx 页面时，ASP.NET 把该用户看作是匿名用户，给它们授予访问页面的权限。但是，如果用户试图访问其他页面(也许正好是页面 URL)，就需要输入登录信息。

还有一个问题：如果要注销、删除 cookie 和登录信息，该怎么办？可以使用 Forms Authentication 对象的 SignOut()方法(包含在 System.Web.Security 名称空间中)，它正好完成了我们需要的工作。如果在查询字符串中，将 Action 参数的值设置为“logout”，这个方法就在 Page_Load()过程中调用：

```
protected void Page_Load(object sender, EventArgs e)
{
    // if the QueryString has the "logout" parameter, log out
    // and delete the cookie for the login page
    if (Request.Params["Action"]=="logout")
        FormsAuthentication.SignOut();
}
```

带有这个参数的页面由 Logout 菜单项调用，它在 Header 用户控件中定义，如下所示：

```
<asp:HyperLink runat="server"
    NavigateUrl="Login.aspx?ReturnUrl=Lists.aspx&Action=logout"
    Text="Logout"   Class="MenuItem" ... />
```

注意，除了“Action=logout”参数外，还有一个 ReturnUrl 参数，将它设置为 Lists.aspx。这是必须的，因为在注销后，就返回登录窗体。如果管理员再次注册，登录页面必须知道要跳转页面的 URL。

27.4.12　编译装配件

在测试应用程序，把它上传到 Web 服务器之前，最后一步是编译前面编写的所有后台编码文件。如果使用的是 VS.NET，Solution Explorer 应如图 27-24 所示。

选择 Build 命令，VS.NET 就会编译工程，并考虑引用两个事务装配件 (NewsMailerConfig 和 NewsMailerDB)。



图　27-24

27.4.13　运行应用程序

编译好装配件后，就需要确保这些装配件位于 Web 应用程序所在根文件夹的 bin 目录下，或者位于通过 IIS 管理员创建的虚拟文件夹下。修改 web.config 文件中的连接字符串，使之指向 Access 数据库(或 DSN)，这样就可以在 Internet Explore 中测试程序了。要查看 Lists 页面，可访问 http://localhost/newsmailer/lists.aspx Web 站点。

27.4.14　浏览器的兼容问题

这个应用程序的所有页面都已经开发好，并在 Internet Explorer 5.5 下进行了测试。如果在 Netscape 6 使用该应用程序，其布局会有所不同，因为 Netscape 对一些样式表设置有不同的解释。在 Netscape 的早期版本中浏览站点，其效果可能更糟。这对于必须能由公众访问的一般 Web 应用程序来说是一个问题。在产品系统中，如果不可能规定用户使用的浏览器(在本例中，至少是管理员使用的浏览器)，开发可以在所有浏览器上工作的页面就非常重要了。

27.5　小结

第一个由数据库驱动的 C#应用程序 ASP.NET 到此就完成了。本案例的目的是展示如何编写 Web 应用程序，如何使用 C#语言来编写，如何利用.NET Framework 中的一些基类。本章还介绍了使用 ASP.NET 服务器控件，可以少编写数百行代码，开发出结构更好、效果更令人满意的页面。

- 这个应用程序可以用于许多 Web 站点，还可以添加更多的功能，使之更强大。下面是一些方法：

- 允许创建和管理订阅窗体，让用户一次订阅多个邮件列表，但只需输入一次电子邮件地址。
- 允许每个用户选择是接收 HTML 格式的时事通讯，还是接收纯文本的时事通讯，并根据该选择发送不同格式的时事通讯。
- 允许在时事通讯中发送附件。
- 允许有多个拥有不同权限的管理员。例如，一个基本的管理员只能查看当前的邮件列表和订阅者，第二个级别的管理员还可以发送新的时事通讯，而高级管理员有全部权限，能添加、删除和修改邮件列表和订阅者。

这里讨论的大多数论题主要针对的是使用 ASP.NET 进行 Web 开发，但许多原则和一些代码(特别是 Utilities 类)也可以用于 Windows 应用程序，例如 IO 管理、面向对象设计、通过 ADO.NET 访问数据库，以及许多.NET 基类。即使读者对 Web 编程不是很感兴趣，也可以利用这些代码。

第28章 范例——用C#建立一个在线购物车

28.1 引言

作为 Web 开发人员，所创建的 Web 站点需要满足各种各样的应用需求。每个开发出来的 Web 站点都是独特的，都要根据每个客户的特定需要添加各种功能。站点的确是为每个有不同需要的客户而构建的，但每个站点也共享一些相同的特性。站点中共享相同特性的这一部分可以当作能重用的独立应用程序。

因此，本章将研究目前站点中一个公用块的开发循环——在线购物车。

本章将从分析购物车的开发，把它作为一组组件进行清晰的描述，再执行它们，之后就可以把这些知识用于非 ASP.NET 领域。在描述了每个组件后，就解释用 C#编写的执行代码。

应用程序被设计为使用一个用于存储数据的 SQL Server 数据库。应用逻辑在 Web Form 中完成，并通过 Web 浏览器显示给用户。核心逻辑使用后台编码技术驻留在一个单独的 C#组件中。它也与.NET Framework Beta 2 兼容。

假定读者已经具备了 C#语言、Web 开发和数据库设计的基础知识。

28.2 建立基础

首先花一些时间理解一个简单的购物车是如何工作的。从这些知识中，可以得到一些结论，并说明我们的要求。在这些要求中，数据库设计就会显现出来。

在完成这些步骤后，我们就有一条清晰的开发路径，准备在 C#语言中执行在线购物车。

28.2.1 深入理解在线购物车模型

首先看看在线购物车的简化图形，如图 28-1 所示。这些是要讨论的功能块。

其中有 4 个基本模块：

商品类别——在此以有组织的方式给客户显示选项。

商品细目——在此给客户显示尽可能多的信息，展示我们的商品。

购物车——在此管理用户选中的商品。

检验系统——在此保存选中的商品和客户的信息，结束交易。

这些是在线商店中要执行的基本块。它包括后面将要详细讨论的、必不可少的功能。这些块可以用更多的特性进一步扩展，详见以后的内容。

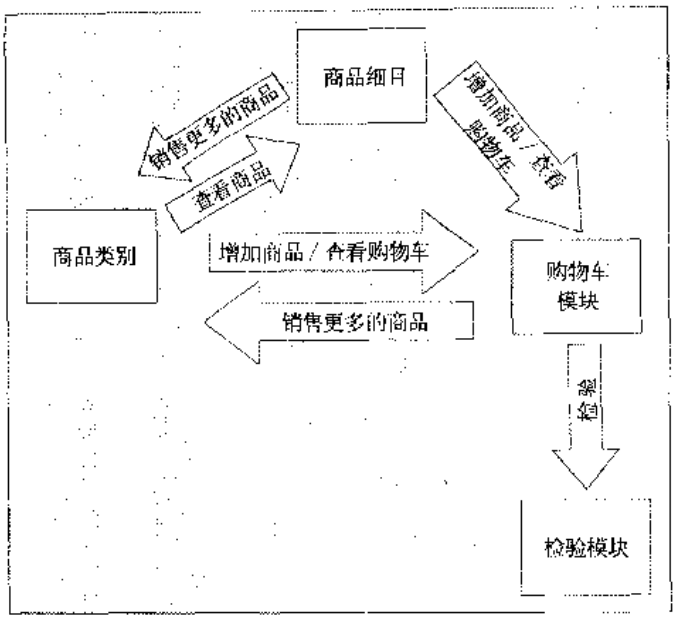


图　28-1

28.2.2　定义需求

与在每个软件开发循环中一样，首先需要定义需求，这样才能设计出满足客户需要的软件。在线购物车应用程序应完成下述任务：

- 有一个类别和子类别的列表。
- 商品应放在其相应的子类别中。
- 商品应可以按照类别和家用来选择。
- 每个商品都应有一个 ID、或短或长的描述、或小或大的图像、库存和价格。
- 用户应能把商品放在车中，或者取出它们。
- 用户应有一个订单号，并能通过订单跟踪系统跟踪它。

28.2.3　生成数据库模型

在我们的需求中，定义数据库模式。图 28-2 显示了一个概念模型，以及这个购物车版本中使用的所有表。

这些表可以分组为：

- 绿色——应用程序的类别部分
- 天蓝色——商品的细目
- 桔黄色——购物车筐
- 黄色——检验系统

1. 类别部分

绿色部分对应于类别部分。它由两个表组成。这是一个非常简单的安排，子类别表继承了 section Id。这样就可以用类别/子类别的方式显示商品了。这种方式在大多数情况下都很有效，读者可以改善它，可以使用递归类别进行非常灵活的设计，该递归类别继承了父 Id。

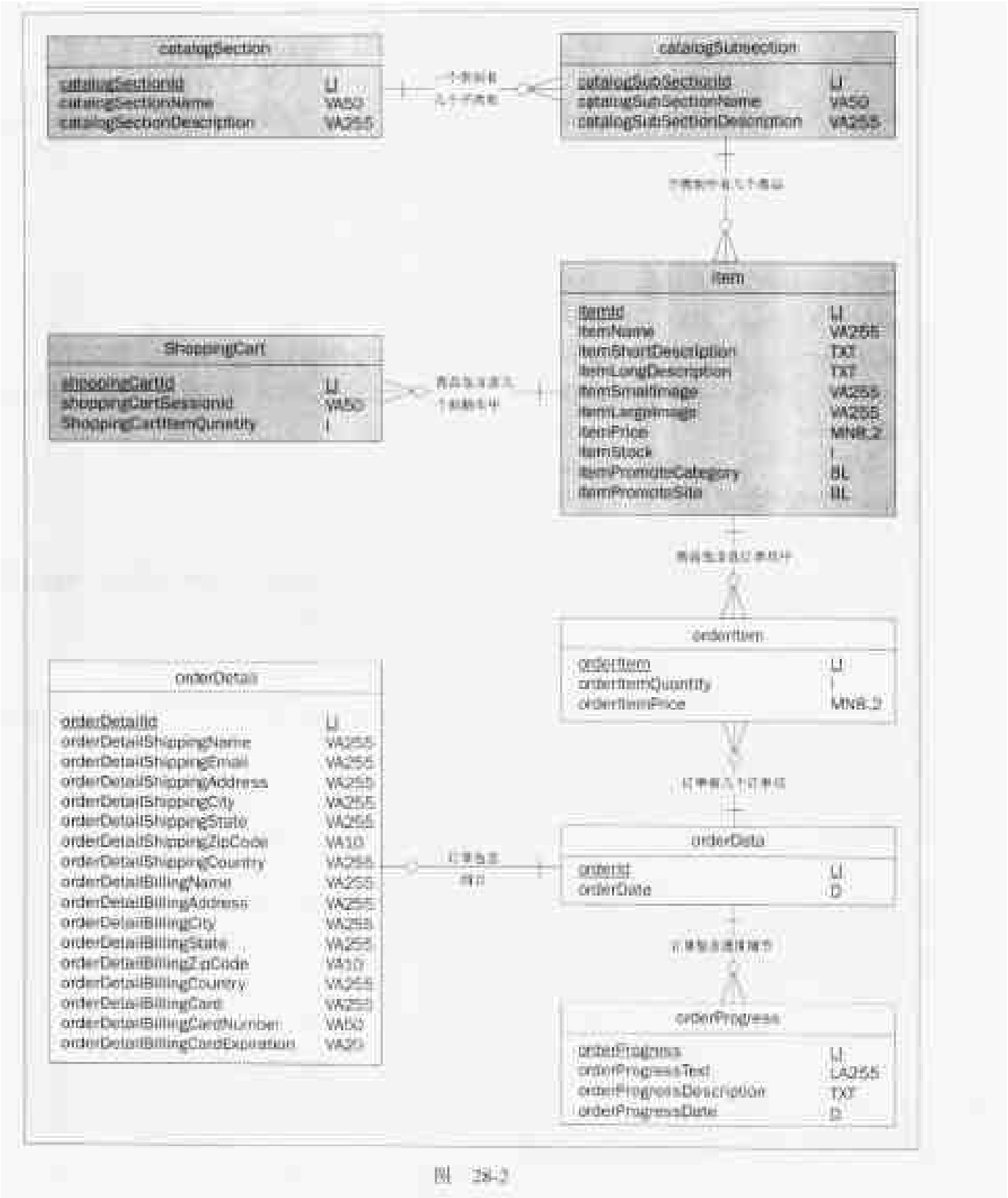


图　28-3

2. 商品的细目

这个模型中的商品细目部分是商品细目设计中的一个删节版本。其中只有一个表，保存了商品的重要信息，例如描述、价格和图像。此外，还有两个布尔值，用于指定商品是显示为家

用还是类别商品。

3. 购物车筐

购物车筐是一个简单的表，用于临时存储用户选择的商品，本例仅存储了 session Id，item id 和 quantity。我们把它作为一个内存中的数据表来实现，在一个会话变量中保存选中的商品。这个表列出了在会话中需要保存的信息。

4. 检验系统

这是最复杂的部分或系统。在检验时，要在订单表上创建一个 order Id。然后把存储在购物车筐中的商品取出来，并保存在 order Items 表中。用户信息存储在 order details 表中。这两个表继承了 order Id。第四个表也继承了 order Id。这是一个 order Progress 表，用于跟踪订单。在购物的过程中，在线购物管理员应在这个表中添加一个记录，表示进度完成的日期。然后由购物者进行检查。

28.3 开始编码

当基础工作的设计完成后，就要开始编码了。代码的体系结构基于简单的 Web Forms，称为定制的用户控件。核心逻辑使用后台编码技术在独立的组件中完成。

28.3.1 应用程序的架构

首先建立基本架构。我们利用 web.config 文件保存重要的应用程序信息，例如连接字符串，该配置文件如下所示：

```
<configuration>
  <appSettings>
    <add key="connString"
value="server=(local)\NetSDK;database=ShoppingCart;Trusted_Connection=yes" />
  </appSettings>
</configuration>
```

要获取这些信息，可使用下面的代码段：

```
String connString = ConfigurationSettings.AppSettings["connString"];
```

可以看出，这是非常简单的，也非常便于存储应用程序的数据。

28.3.2 类别列表块

建立了架构后，首要的任务是显示一个商品列表，如果选择了类别或子类别，就按照这些类别进行筛选。

首先建立 Web Forms 的工作区。设置语言，在代码中添加适当的名称空间。我们把 C#设置为要使用的语言，导入 System 名称空间。接着添加 System.Data 和 System.Data.OleDb，因

为下面要把 SQL Server 用作数据库引擎。一旦把这个应用程序用于实际工作，就应把调试标志设置为 false。

```
<%@ Page Language="C#" Debug="true" %>
<%@ Import Namespace="System" %>
<%@ Import Namespace="System.Data" %>
<%@ Import Namespace="System.Data.SqlClient " %>
```

现在把引用设置为定制的用户控件。

```
<%@ Register TagPrefix="SC" TagName="Site" Src="uc_header.ascx" %>
<%@ Register TagPrefix="SC" TagName="Section" Src="uc_catalog_section.ascx" %>
<%@ Register TagPrefix="SC" TagName="SubSection" Src="uc_catalog_subsection.ascx" %>
<%@ Register TagPrefix="SC" TagName="ProductList" Src="uc_catalog_product_list.ascx" %>
```

把它们嵌入到 aspx 文件中，引用控件。

```
<SC:Site runat="server" />
```

下面是完整的类别 WebForm 的 catalog.aspx 文件。

```
<%@ Page Language="C#" Debug="true" %>
<%@ Register TagPrefix="SC" TagName="Site" Src="uc_header.ascx" %>
<%@ Register TagPrefix="SC" TagName="Section" Src="uc_catalog_section.ascx" %>
<%@ Register TagPrefix="SC" TagName="SubSection" Src="uc_catalog_subsection.ascx" %>
<%@ Register TagPrefix="SC" TagName="ProductList" Src="uc_catalog_product_list.ascx" %>
<html>
<head>
<title>Shopping Cart in C# - Catalog</title>
<link rel="stylesheet" href="style.css">
</head>
<body>
<form runat=server>
<SC:Site runat="server" />
<table width="800" cellspacing="0" cellpadding="3" border="0"
style="border-color:Black;border-width:1px;border-style:solid;">
<tr>
<td width="800" align="right" bgcolor="#CCCCCC"><span class="itemTitle">Item Catalog</span></td>
</tr>
</table>
<table width="800" cellspacing="0" cellpadding="0" border="0">
<tr>
<td width="200" height="100%" rowspan="2" valign="top"><SC:Section runat="server" /></td>
<td width="600" height="10" valign="top"><SC:SubSection runat="server" /></td>
</tr>
<tr>
<td width="600" height="100%"><SC:ProductList runat="server" /></td>
```

```
</tr>
</table>
</form>
</body>
</html>
```

在第一个 Web Form 中使用了 4 个定制用户控件。现在要执行这些定制控件，其中的大多数控件都要在站点上重用。

首先执行最简单的用户控件 uc_header.ascx。这个文件包含站点的徽标和两个链接。我们把这个文件封装在一个控件中，因为它要在整个站点中重用，下面是代码：

```
<table width="800" cellspacing="0" cellpadding="3" border="0"
style="border-color:Black;border-width:1px;border-style:solid;">
<tr>
<td width="400"><a href="catalog.aspx"><span class="mainTitle">Shopping Cart Site</span></a></td>
<td width="400" align="right"><a href="cart.aspx">View Cart</a><span class="mainText">| </span><a
href="tracking.aspx">Track your order</a></td>
</tr>
</table>
```

可以看出，这是一个静态表，包含的仍是旧风格的文件。接着编写一个比较复杂的用户控件 uc_catalog_section.ascx，它显示了可以在站点上使用的类别列表。这是通过把一个数据检索后台编码类绑定到用户控件上实现的。

下面是用户控件的代码：

```
<%@ Control Language="C#" Debug="true" Inherits="后台编码.UcSection" %>
<%@ Import Namespace="System.Data" %>
<%@ Import Namespace="System.Data.SqlClient" %>
<html>
<head>
    <title></title>
     <script language="C#" runat="server"></script>
</head>
<body>

    <font face="Verdana" size="-1">
        <asp:DataList id="MySectionList" runat="server"
            BorderColor="black"
            BorderWidth="1"
            GridLines="Both"
            CellPadding="3"
            Font-Name="Verdana"
            Font-Size="8pt"
            Width="200px"
            HeaderStyle-BackColor="#aaaadd"
            SelectedItemStyle-BackColor="Gainsboro"
```

```
                >
                    <HeaderTemplate>
                      Sections
                    </HeaderTemplate>
                    <ItemTemplate>
                    <a href="catalog.aspx?sectionId=<%#
DataBinder.Eval(Container.DataItem,"catalogSectionId") %>

&sectionIndex=<%#Container.ItemIndex%>">
                                                <%# DataBinder.Eval(Container.DataItem,
"catalogSectionName") %></a>
                    </ItemTemplate>
                    <SelectedItemTemplate>
                      <%# DataBinder.Eval(Container.DataItem, "catalogSectionName") %>
                    </SelectedItemTemplate>
                </asp:DataList>
            </font>

    </body>
    </html>
```

这里把更多的 HTML 放在控件上，并拖放了一个 DataList 对象，以处理信息的显示。<%# DataBinder.Eval(Container.DataItem, "catalogSectionName") %>代码用于从绑定的数据中选择特定的信息，并把它们放在合适的位置上。

在后台编码类中，首先必须把 DataList 对象声明为受保护的对象，接着执行一些数据库的操作。从数据库中获取了需要的信息后，就把它们绑定到 Web 控件 DataList 上。

用户控件继承了后台编码类，这样，它们就可以一起工作了。把共享的变量(例如 Web Form 控件)声明为受保护的变量是非常重要的，这样它才能在后台编码类中访问。

```
public class UcSection : UserControl {
  protected DataList MySectionList;
  protected Label MyLabel;

  protected void Page_Load(object sender, EventArgs e) {
    if (!IsPostBack) {
      Bind_MySectionList();
    }

    if (Request.QueryString["sectionIndex"] != null) {
      MySectionList.SelectedIndex = Int32.Parse(Request.QueryString["sectionIndex"]);
    }
  }

  protected void Bind_MySectionList() {
    String connString = ConfigurationSettings.AppSettings["connString"];
```

```
        SqlConnection myConnection = new SqlConnection(connString);
        SqlDataAdapter myCommand = new SqlDataAdapter("SELECT catalogSectionId,
catalogSectionName
                                        FROM catalogSection ORDER BY catalogSectionName",
myConnection);
        DataSet ds = new DataSet();
        myCommand.Fill(ds, "catalogSection");

        MySectionList.DataSource = new DataView(ds.Tables[0]);
        MySectionList.DataBind();
      }
    }
```

接着要执行子类别用户控件，这个控件完成的任务与类别控件类似，仅增加一个应用到数据库上的查询过滤器。选择对应于选中类别的子类别，其余部分就完全相同了。

```
    "SELECT catalogSubSectionId, catalogSubSectionName FROM catalogSubSection WHERE
catalogSectionId= " +
                                                    sectionId + " ORDER BY
catalogSubSectionName"
```

然后需要显示商品列表，有 3 种类型的商品：

- 站点商品：如果没有选中某个类别，就显示它们。
- 类别商品：如果选择了某个类别，但没有选择子类别，就显示它们。
- 一般的商品：这些商品属于选中的子类别。

需要考虑这三种可能性，创建正确的查询，这是在 blah 类中用下面的代码完成的。在有了查询后，把它绑定到 Web 控件数据网格上，以在一个表中显示商品。

```
    string SQLQuery = "SELECT itemId, itemName, itemShortDescription, itemSmallImage, itemPrice,
itemStock
                                                FROM item WHERE itemPromoteSite=1
ORDER BY itemName";

    int mysectionId = 0;
    if (Request.QueryString["sectionId"] != null) {
      mysectionId = Int32.Parse(Request.QueryString["sectionId"]);
    }
    if (mysectionId != 0) {
      //If we have a section selected we filter products for this section
      SQLQuery = "SELECT item.itemId, item.itemName, item.itemShortDescription, item.itemSmallImage,
item.itemPrice, item.itemStock
          FROM (catalogSection INNER JOIN catalogSubSection ON catalogSection.catalogSectionId =
catalogSubSection.catalogSectionId)
                                            INNER JOIN item ON
catalogSubSection.catalogSubSectionId = item.catalogSubSectionId
```

```
                                    WHERE (((catalogSection.catalogSectionId)=" + mysectionId + ") AND
((item.itemPromoteCategory)=1))";
    }

    int mysubsectionId = 0;
    if (Request.QueryString["subsectionId"] != null) {
      mysubsectionId = Int32.Parse(Request.QueryString["subsectionId"]);
    }
    if (mysubsectionId != 0) {
      //If we have a subsection selected we filter products for this subsection
      SQLQuery = "SELECT itemId, itemName, itemShortDescription, itemSmallImage, itemPrice, itemStock
FROM item WHERE catalogSubSectionId=" + mysubsectionId + " ORDER BY itemName";
    }
```

结果得到类别显示层，它由 4 个主要的用户控件组成：

- 第一个控件负责显示类别列表。
- 第二个控件负责显示子类别列表。
- 第三个控件负责显示一个商品网格。这个商品列表是根据不同的情况，从 3 个 SQL 查询中的一个获取的。
- 最后一个控件显示页面的顶部。

所有的控件共同组成了 catalog.aspx。用户控件利用后台编码类进行数据库访问，以及在页面上绑定 DataList。

catalog.aspx 文件应得到如图 28-3 所示的结果。

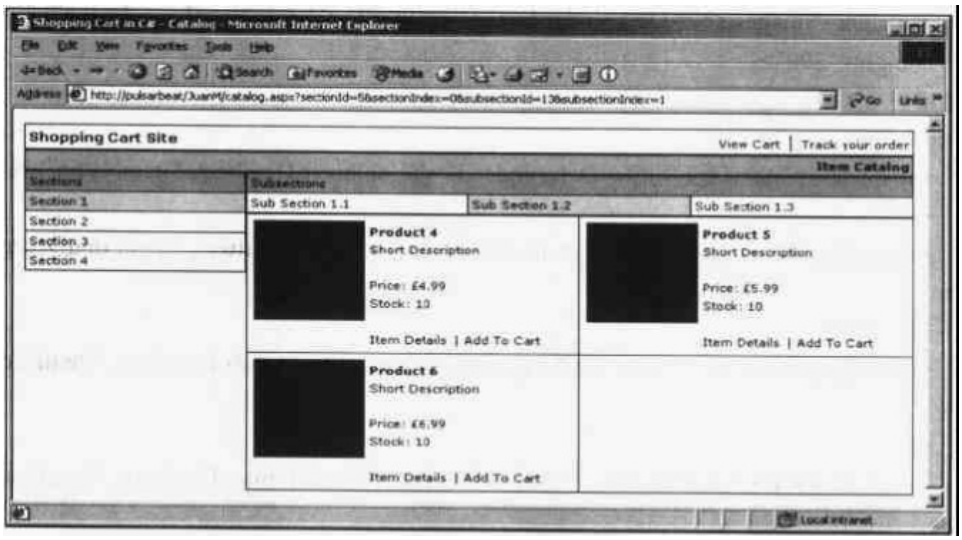


图　28-3

28.3.3　商品的细目

建立购物车的下一步是显示商品的细目。首先用一些代码来显示类别。下面将使用基本的 HTML 架构和 3 个用户控件。标题、类别和子类别控件会重用。

这里的区别是用一个新的商品细目控件替代商品列表控件。我们使用一个数据列表 Web 控

件，以显示商品的特性以及后台编码存储库中的相应类。

这个简单任务的用户控件代码如下所示：

```
<%@ Control Language="C#" Debug="true" Inherits="后台编码.UcItemDetails" %>
<%@ Import Namespace="System.Data" %>
<%@ Import Namespace="System.Data.SqlClient" %>

<script language="C#" runat="server"></script>
<font face="Verdana" size="-1">
<asp:DataList id="MyItemDetails" runat="server"
  BorderColor="black"
  BorderWidth="1"
  GridLines="Both"
  CellPadding="3"
  Font-Name="Verdana"
  Font-Size="8pt"
  Width="600px"
  HeaderStyle-BackColor="#aaaadd"
  SelectedItemStyle-BackColor="Gainsboro"
  RepeatDirection = "Horizontal"
  RepeatColumns = "1"
  >
  <ItemTemplate>
    <table>
    <tr>
    <td valign="top">
      <img src="images/<%# DataBinder.Eval(Container.DataItem, "itemLargeImage") %>"
width="200" height="200">
    </td>
    <td valign="top">
      <span class="itemTitle"><%# DataBinder.Eval(Container.DataItem, "itemName") %></span>
      <br>
      <span class="itemText"><%# DataBinder.Eval(Container.DataItem, "itemLongDescription")
%></span>
      <br><br>
      <span class="itemText">Price: <%# (DataBinder.Eval(Container.DataItem, "itemPrice", "{0:c}"))
%></span>
      <br>
      <span class="itemText">Stock: <%# DataBinder.Eval(Container.DataItem, "itemStock")
%></span>
      <br><br>
      <a
href="catalog.aspx?sectionId=<%=sectionId%>&sectionIndex=<%=sectionIndex%>&subsectionId=<%=subsec
tionId%>

&subsectionIndex=<%=subsectionIndex%>">Keep On Shopping!</a>
      <span class="itemText"> | </span>
      <a href="cart.aspx?cartAction=1&itemId=<%# DataBinder.Eval(Container.DataItem, "itemId")
```

```
%>">Add To Cart</a>
            </td>
            </tr>
            </table>
        </ItemTemplate>
        </asp:DataList>
    </font>
```

后台编码类如下所示：

```
    protected void Bind_ItemDetails() {
        int myItemId = 0;
        if (Request.QueryString["itemId"] != null) {
            myItemId = Int32.Parse(Request.QueryString["itemId"]);
        }
        string SQLQuery = "SELECT itemId, itemName, itemLongDescription, itemLargeImage, itemPrice,
itemStock
                                                                    FROM item WHERE
itemId = " + myItemId;
        String connString = ConfigurationSettings.AppSettings["connString"];
        SqlConnection myConnection = new SqlConnection(connString);
        SqlDataAdapter myCommand = new SqlDataAdapter(SQLQuery, myConnection);
        DataSet ds = new DataSet();
        myCommand.Fill(ds, "item");

        MyItemDetails.DataSource = new DataView(ds.Tables[0]);
        MyItemDetails.DataBind();
    }
```

这个简单的模块还可以进一步扩展。这个窗体显示了存储的商品细目，可以用用户的评论和相关的商品列表来扩展其功能。

得到的屏幕如图 28-4 所示。

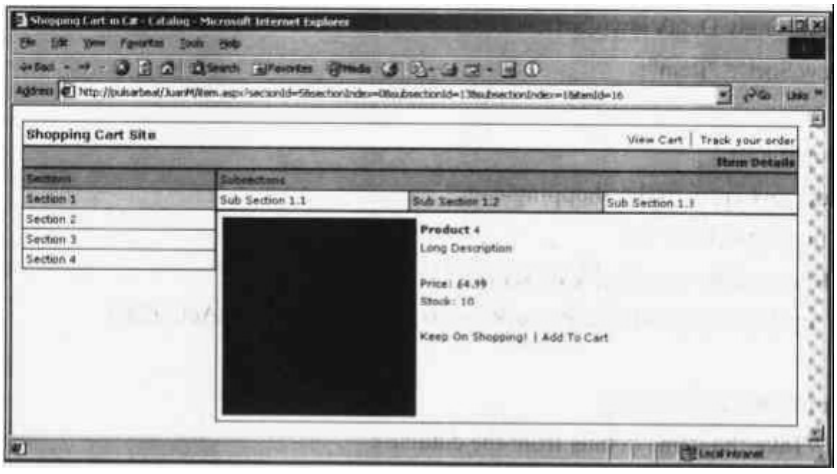


图　28-4

28.3.4 购物车筐

购物车筐执行大部分功能。它应提供添加、编辑和删除商品的方式。车筐仅是一个临时的存储区域，用于把用户感兴趣的商品组合起来。接着需要一些会话管理来识别每个客户在自己的会话中所选择的商品。

与以前的 ASP 应用程序一样，ASP.NET 提供了处理会话所必须的对象。在这里我们将利用 Session 对象。把一个 DataTable 对象存储为 Session 变量，这样所有的购物车筐操作就都在内存中进行。这是一种非常快速的方法，但需要提供足够的内存来处理并发客户。

购物车筐模块由两个基本部分组成。第一个部分管理用户放在筐中的商品，第二部分负责显示筐中当前的商品。下面详细介绍这两个部分。

Web Form 用查询字符串变量把一个商品添加到购物车上。这里仅把商品添加到 DataTable 中。要执行更新和删除方法，需要把 DataTable 绑定到 DataList Web 控件上，并执行操作：

```
void Page_Load(Object Sender, EventArgs E) {
    if (Request.QueryString["cartAction"] != null) {
        this.cartAction = Int32.Parse(Request.QueryString["cartAction"]);
    }

    if (Session["ShoppingCart"] == null) {
        Cart = new DataTable();
        Cart.Columns.Add(new DataColumn("ItemId", typeof(string)));
        Cart.Columns.Add(new DataColumn("Item", typeof(string)));
        Cart.Columns.Add(new DataColumn("Qty", typeof(string)));
        Cart.Columns.Add(new DataColumn("Price", typeof(string)));
        Session["ShoppingCart"] = Cart;
    }
    else {
        Cart = (DataTable)Session["ShoppingCart"];
    }

    CartView = new DataView(Cart);
    CartView.Sort = "Item";

    if (!IsPostBack) {
        //Add new entry to the shopping cart
        int myCartAction = 0;
        if (Request.QueryString["cartAction"] != null) {
            myCartAction = Int32.Parse(Request.QueryString["cartAction"]);
        }
        if (myCartAction==1) {
            //Take the items details from the database
            string myItemId = "0";
            if (Request.QueryString["itemId"] != null) {
                myItemId = Request.QueryString["itemId"];
```

```
            }
            string SQLQuery = "SELECT itemName, itemPrice FROM item WHERE itemId = " +
myItemId;
                String connString = ConfigurationSettings.AppSettings["connString"];

                  SqlConnection myConnection = new SqlConnection(connString);
                     SqlCommand myCommand = new SqlCommand(SQLQuery, myConnection);
                 myConnection.Open();
                SqlDataReader dr2 = myCommand.ExecuteReader();

            if (dr2.Read()) {
                    DataRow dr = Cart.NewRow();
                  dr[0] = myItemId;
                  dr[1] = dr2.GetString(0);
                  dr[2] = "1";
                dr[3] = dr2.GetSqlMoney(1).ToString();
                 Cart.Rows.Add(dr);
            }
        }

        BindCartList();
        BindTotalList();
    }
}
```

更新和删除数据的函数如下所示：

```
    protected void DataCartList_DeleteCommand1(Object Sender, DataListCommandEventArgs e) {
        string item = ((Label)e.Item.FindControl("Label4")).Text;
    CartView.RowFilter = "Item='"+item+"'";
        if (CartView.Count > 0) //item exists in cart
            CartView.Delete(0);
        CartView.RowFilter = "";

        CartList.EditItemIndex = -1;
        BindCartList();
    BindTotalList();
    }


    protected void DataCartList_EditCommand1(Object Sender, DataListCommandEventArgs e) {
        CartList.EditItemIndex = (int)e.Item.ItemIndex;
        BindCartList();
    BindTotalList();
    }
```

```
protected void DataCartList_CancelCommand1(Object Sender, DataListCommandEventArgs e) {
    CartList.EditItemIndex = -1;
    BindCartList();
BindTotalList();
}

protected void DataCartList_UpdateCommand1(Object Sender, DataListCommandEventArgs e) {
string itemId = ((Label)e.Item.FindControl("Label1")).Text;
    string item = ((Label)e.Item.FindControl("Label2")).Text;
    string qty = ((TextBox)e.Item.FindControl("Text1")).Text;
    string price = ((Label)e.Item.FindControl("Label3")).Text;

    // with a database, we'd use an update command.  Since we're using an in-memory
    // DataTable, we'll delete the old row and replace it with a new one
    //remove old entry
    CartView.RowFilter = "Item='"+item+"'";
    if (CartView.Count > 0) //item exists in cart
        CartView.Delete(0);
    CartView.RowFilter = "";

    //add new entry
    DataRow dr = Cart.NewRow();
    dr[0] = itemId;
    dr[1] = item;
    dr[2] = qty;
dr[3] = price;
    Cart.Rows.Add(dr);

    CartList.EditItemIndex = -1;
    BindCartList();
BindTotalList();
}
```

最后需要对订单上的商品进行小计、计算税款和合计。我们使用 BindTotalList()函数生成结果，并把它绑定到页面上的另一个 Web 控件上：

```
protected void BindTotalList() {
    double mySubTotal = 0;
    double myTaxRate = 0.15;
    double myTax = 0;
    double myTotal = 0;

    DataTable CartTable = (DataTable)Session["ShoppingCart"];
    foreach(DataRow myRow in CartTable.Rows) {
        Double tempPrice = Double.Parse(myRow[3].ToString());
```

```
        Int32 tempQty = Int32.Parse(myRow[2].ToString());
        mySubTotal += tempPrice * tempQty;
    }

    myTax = mySubTotal * myTaxRate;
    myTotal = mySubTotal + myTax;

    CartTotal = new DataTable();
        CartTotal.Columns.Add(new DataColumn("SubTotal", typeof(string)));
        CartTotal.Columns.Add(new DataColumn("Tax", typeof(string)));
        CartTotal.Columns.Add(new DataColumn("Total", typeof(string)));

    DataRow drTotal = CartTotal.NewRow();
        drTotal[0] = mySubTotal.ToString();
        drTotal[1] = myTax.ToString();
        drTotal[2] = myTotal.ToString();
    CartTotal.Rows.Add(drTotal);

    CartTotalView = new DataView(CartTotal);
    CartTotalList.DataSource = CartTotalView;
    CartTotalList.DataBind();
}
```

购物车的屏幕如图 28-5 所示。

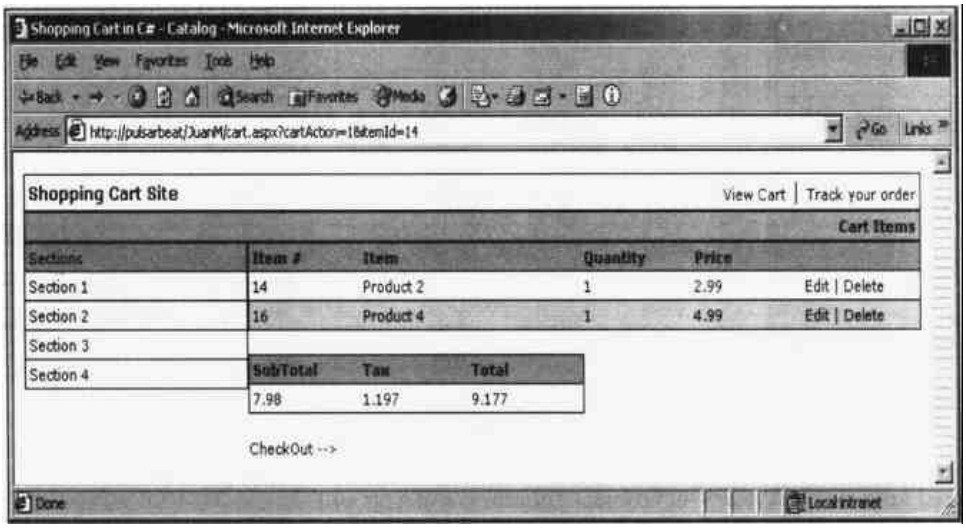


图　28-5

28.3.5　检验系统

检验系统负责保存客户信息和要处理的选中商品。我们使用 3 个表来保存客户信息，如下所示：

- orderData——存储订单的日期，生成 order Id。
- orderItem——已购买的商品列表，它还保存商品购买时的价格。

• orderDetail——存储订单的购物和账单信息。

把这个过程分为 3 步：

• 购物和账单信息——在这里可获取购物信息。有了这些信息，就可以计算购物的费用。这应根据每个存储的需要来进行。我们还验证信用卡信息的有效性。这应使用某个特定的供应商程序如 paypal 来完成。

• 确认——这里显示接收到的信息小结，并显示已购商品的列表，以及订单的商品总数。如果正确，用户就可以提交这些信息了。

• 小结——结束交易，给用户显示订单号。

把信息从第一步传送到第二步。然后使用 Web Form 控件请求确认。如果信息正确，就注册客户的订单。

在发送信息之前，需要进行某些窗体的有效性验证，通过外部提供者(provider)或自己的软件进行信用卡的有效性验证。

28.3.6　跟踪订单

在线购物车程序的最后一部分是跟踪系统。我们执行一个基本的跟踪系统，要求用户输入电子邮件地址和订单号。系统将显示一个在跟踪数据库中注册的里程碑列表。

我们使用一个表存储跟踪信息。它包含订单的 id 和要获取的目标信息。用户输入有效的电子邮件和订单号后，就显示图 28-6 所示的事件的列表。

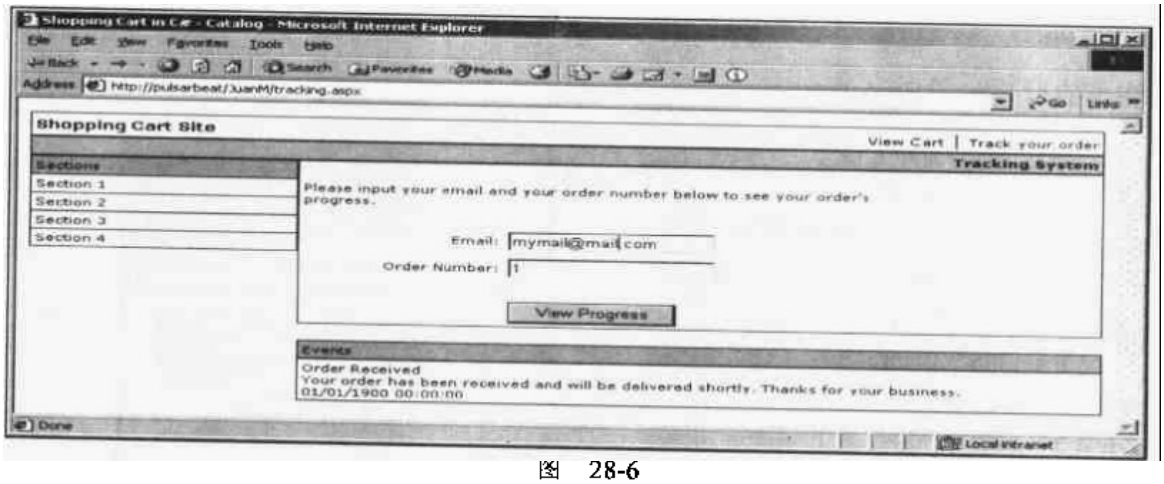


图　28-6

需要的代码如下：

```
protected void ButtonView_Click(object Source, EventArgs e) {
    //Look for order id with provided email and order number
    String connString = ConfigurationSettings.AppSettings["connString"];
    string strSelect = "SELECT orderId FROM orderDetail WHERE orderId=" + orderIdBox.Text + "
AND orderDetailShippingEmail='" + emailBox.Text + "'";
    SqlConnection myConnection = new SqlConnection(connString);
    SqlCommand myCommand = new SqlCommand(strSelect, myConnection);

    myConnection.Open();
    SqlDataReader dr = myCommand.ExecuteReader();
```

```
        int newOrderId = 0;
        if (dr.Read()) {
            //emailBox.Text = "Encontramos!!!";
            Bind_TrackingList();
        }
        myConnection.Close();
    }

    protected void Bind_TrackingList() {
        string SQLQuery = "SELECT orderProgressText, orderProgressDescription, orderProgressDate FROM
orderProgress WHERE orderId = " + orderIdBox.Text;
        String connString = ConfigurationSettings.AppSettings["connString"];

        SqlConnection myConnection = new SqlConnection(connString);
        SqlDataAdapter myCommand = new SqlDataAdapter(SQLQuery, myConnection);

        DataSet ds = new DataSet();
        myCommand.Fill(ds, "orderProgress");

        MyTrackingList.DataSource = new DataView(ds.Tables[0]);
        MyTrackingList.DataBind();
    }
```

28.4　站点的部署

这个应用程序的部署非常简单：

- 在 SQL Server 中创建一个新数据库。
- 创建好数据库后，需要运行生成的数据库创建脚本(database creation scrip)。
- 在 IIS 管理程序中创建一个新站点。
- 把 web.config 文件复制到 www 根文件夹下。
- 把 Web Forms 和用户控件复制到 www 根文件夹下。
- 把后台编码.dll 复制到 wwwroot/bin 目录下(不需要注册)。

28.5　改进在线购物车程序

刚才创建的购物车采用一种基本的方法提供了在线购物车的核心架构。可以对它进行进一步的改进。这可以由读者完成。这些改进应扩展购物车程序，为在线团体提供更好的服务，这里给出一些想法：

28.5.1 功能的改进

- 递归类别达到 n 层
- 商品的评论
- 商品的相关产品
- 用户的账户和管理
- 客户的邮件列表
- 特殊的购物费用计算

28.5.2 技术方面的改进

- 为提高性能和重用代码，建立基于组件的版本
- 使用存储过程，迁移到 SQL Server 数据库上，提高性能
- 在服务器上执行 SSL(这是必须的，但超出了本文的范围)
- 执行实时信用卡处理和付费

28.6 结论

这个系统说明了在线购物车的基本知识，涉及到在线存储的主要方面，包括商品的分类、商品细目模块、购物车筐和检验模块。目前每个在线购物车都必须执行这些基本部分。希望本章能让读者明白如何轻松地建立自己的在线购物车，添加新功能，满足自己的需要，以及改进该购物车程序的一些方法。

在以后的文章中，我们将介绍一些重要的需求——不考虑本章内容的通用购物车(为了简短)，添加一个管理控制台，修改和删除列表、商品和订单、商品的评论、信用卡付费的 SSL，讨论如何使用信用卡处理提供者。

附录A　设置PATH环境变量

为了在 Windows 命令行上使用 C#编译器和其他.NET 工具，需要确保更新 PATH 环境变量。命令提示在当前目录下查找文件；如果要运行另一个目录下的文件，需要提供信息，让 Windows 知道可以在什么地方找到该文件。为此，需要设置 PATH 环境变量。PATH 变量包含一个目录名列表，用分号将目录名分隔开，如果 Windows 在当前目录下没有找到请求的文件，就会在这些目录下查找。我们需要在这个列表中添加两个文件夹，确保 Windows 能找到所有的.NET 工具，第一个文件夹是 C:\Program Files\Microsoft.Net\FrameworkSDK\Bin，它包含了许多.NET 命令行工具，如 ildasm，gacutil 和 sn；第二个文件夹是 C:\WINNT\Microsoft.NET\Framework\v1.0.2914(最后一个子文件夹中的数字可以不同，它取决于机器上安装的.NET Framework 版本)，它包含 C#编译器本身(csc)和 System 类库。

A.1　Windows 2000/XP

要在 Windows 2000 或 XP 机器上设置 PATH 变量，可从控制面板上打开 System 小程序，单击 Advanced 标签，打开 Advanced 选项卡，如图 A-1 所示。

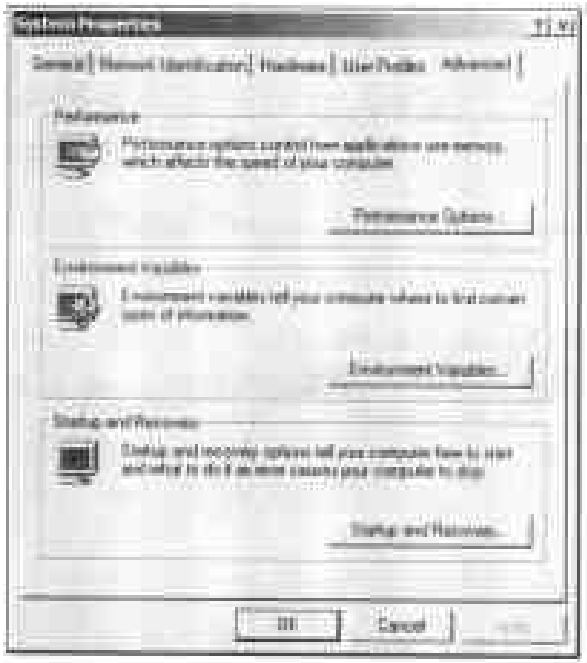


图 A-1

在 Advanced 选项卡中，单击 Environment Variables... 按钮，在 System variables 列表框(如图 A-2 所示)中向下移动滚动条，找到 Path 变量。



图 A-2

选择它，单击列表框下面的 Edit...按钮，打开如图 A-3 所示的对话框。



图 A-3

在 Variable Value 文本框中，填写最终要在 PATH 变量中包含的目录的完整路径。PATH 变量是一个由分号分开的列表，所以需要在第一个路径前面和两个路径之间添加分号。例如，使用.NET Framework 的当前版本，需要在字符串的末尾添加如下文本：

```
;C:\Program Files\Microsoft.Net\FrameworkSDK\Bin;C:\WINNT\
Microsoft.NET\Framework\v1.0.2914
```

单击 OK 按钮，关闭 Edit System Variable 对话框，再关闭 Environment Variables 和 System Properties 对话框。PATH 变量就被更新了。如果已经打开了一个命令窗口，就需要重新打开它，因为新值不会应用到已有的窗口上。

A.2 Windows NT

在 Windows NT4 下设置环境变量的过程略有区别。与 Windows 2000 相同，从控制面板中打开 System 小程序，接着单击 Environment 标签，打开 Environment 选项卡，如图 A-4 所示。

从 System Variables 列表框中选择 Path 变量。这个变量的当前值现在显示在 Value 列表框中，与 Windows 2000 一样，把.NET Framework 路径添加到这个值的末尾(用分号分隔开)。单击 Set 按钮更新该值，单击 OK 按钮，接受新值。这个值现在就会应用到新打开的窗口上。

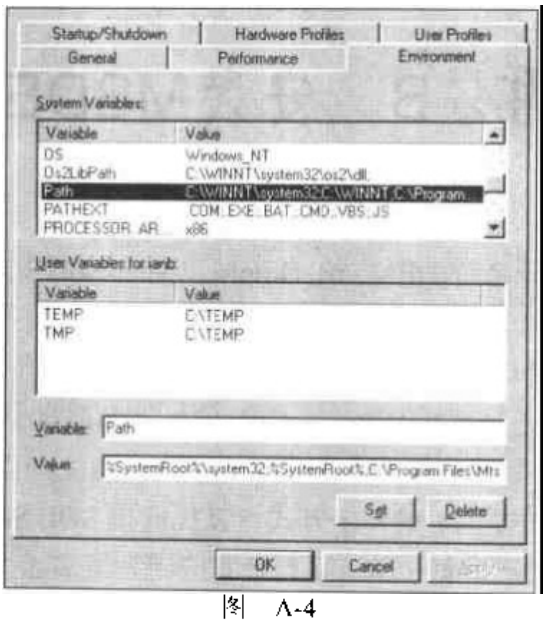


图　A-4

A.3　Windows 95/98/ME

如果运行的是 Windows 9x 或 Windows ME，则需要更新 autoexec.bat 文本文件来设置 PATH 变量。这个文本文件包含用于启动 Windows 时所使用的配置信息，它位于硬盘的根目录下(例如 C:\autoexec.bat)。在 Notepad 或其他文本编辑器中打开这个文件，添加如下一行代码：

```
SET PATH=%PATH%;C:\Windows\Microsoft.NET\Framework\v1.0.2914
```

根据系统和.NET Framework 版本的需要调整路径。这会把路径添加到现有的 PATH 变量中。注意在 Windows 9x 或 ME 机器上，并不是所有的工具都将和.NET Framework 一起安装，最后，重新启动 Windows，使这种改变发挥作用。

附录B　安装MSDE

高效率的SQL数据库管理系统(DBMS)如Oracle和Microsoft SQL Server支持每分钟成千上万个事务处理，同时能在最短的停机时间内进行维护和更新。它们是一个非常大的现代计算过程，但拥有这个功能所需的费用是比较高的。首先是DBMS的起价是几千美元，容量越大，价格越高。业界水准的DBMS还要再加上内存、CPU的性能和磁盘空间的费用。

作为开发人员，不能在膝上型计算机中安装像这样一个功能全面的系统，来测试SQL语句。正如本章开头所述，Microsoft提供了另一个方式可以提取出SQL Server产品，把它与所有的管理和开发工具分隔开来，仅留下数据引擎。接着把它应用到最多5个用户和2GB的数据上。最终的产品称为MSDE (Microsoft Data Engine)。可以把它用作一个后端，再把SQL Server CD中包含的客户工具无缝地关联到书中，创建一个小规模的SQL Server安装程序。

B.1　安装 Microsoft Database Engine (MSDE)

MSDE在Office 2000附带光盘的Disk One\SQL\x86\ SETUP\SetupSQL.exe目录下。

如果有 Visual Products 的一个许可，就可以从http://msdn.microsoft. com/vstudio/ msde/default.asp上下载 MSDE，或者在该站点上请求把一张 CD 发送给你。安装很简单，需要大约50MB的磁盘空间和数据。尽管这是MSDE，不是Microsoft SQL Server，应用程序仍应安装在C:\MSSQL7文件夹下。在该文件夹中有一个Data文件夹，其中包含默认状态下建立的数据库。

(1) 整个安装过程所需的时间不到15分钟。首先关闭所有其他的应用程序。

(2) 选择Start | Run | browse，浏览到SetupSQL.exe可执行文件，并运行它。单击OK按钮。退出浏览窗口，单击OK按钮，运行安装文件。

(3) 启动一个安装向导，该向导会提示您选择一种安装方法。在这里选择Local Install，如图B-1所示，单击Next按钮。



图　B-1

(4) 安装向导现在向您显示一个带有警告信息的欢迎屏幕，要求用户退出所有其他的应用程序，同时显示一些版权信息，如图 B-2 所示。单击 Next 按钮。

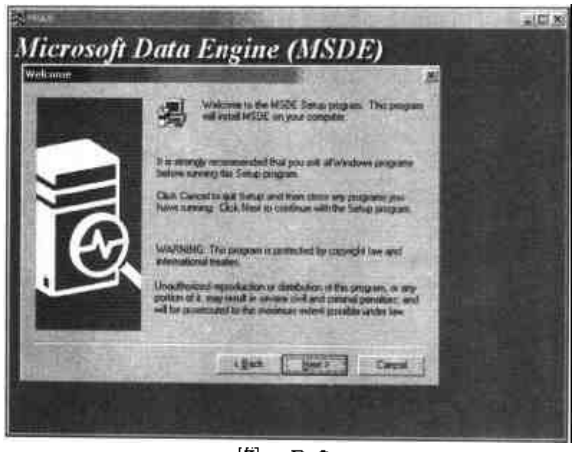


图　B-2

(5) 安装向导要求输入用户和公司信息，如图 B-3 所示，单击 Next 按钮。

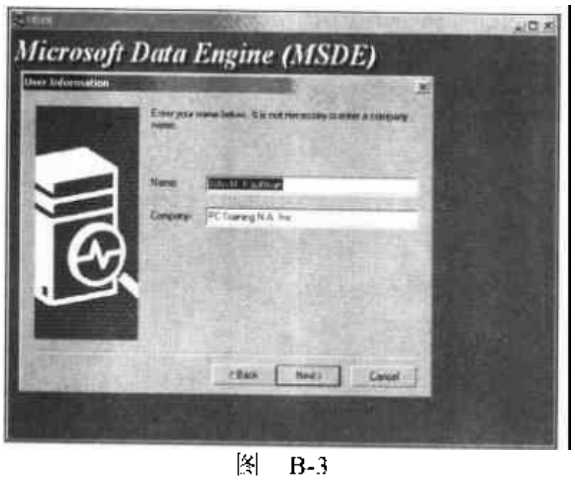


图　B-3

(6) 紧接着会出现一个 Setup Type 窗口，可以在其中修改文件的位置。与完整的 SQL Server 产品一样，可以选择把数据放在一个数据文件驱动器上，而不是保存在程序文件的驱动器上，如图 B-4 所示。单击 Next 按钮。

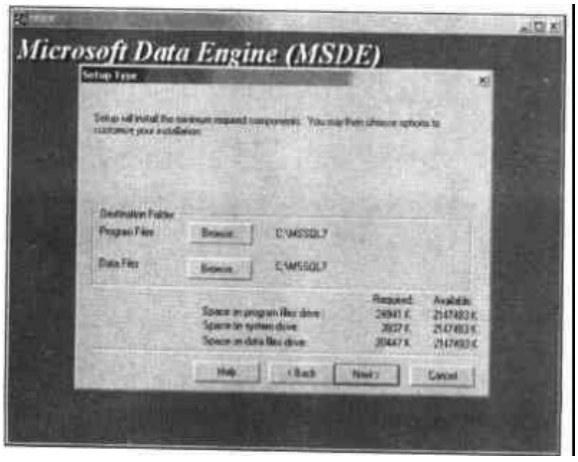


图　B-4

(7) MSDE 需要知道默认的字符集以及其他一些专用语言的设置。这些设置都是自我解释性的，如图 B-5 所示。一般情况下可接受这些默认设置。单击 Next 按钮。

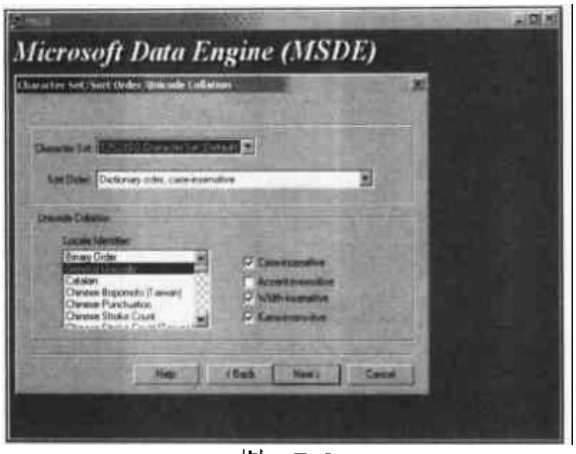


图　B-5

(8) MSDE 提供了安装几个网络链接库的选项，如图 B-6 所示。对于最简单的情况，接受默认设置，只安装 TCP/IP Sockets 和 Multi-Protocol。单击 Next 按钮。

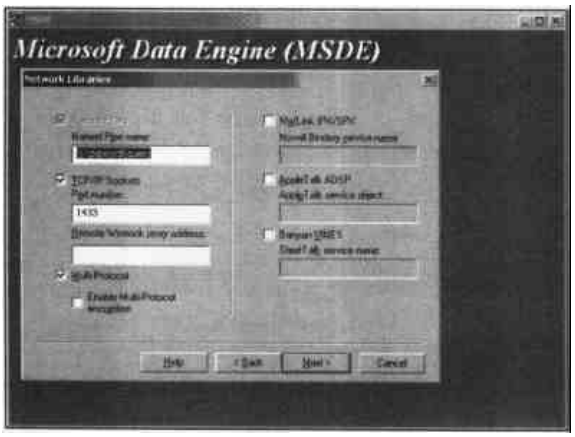


图　B-6

(9) 向导收集要使用的账户管理类型的信息，如图 B-7 所示。对于最简单的情况，选择 Use the same account for each service 和 use local system account。单击 Next 按钮。

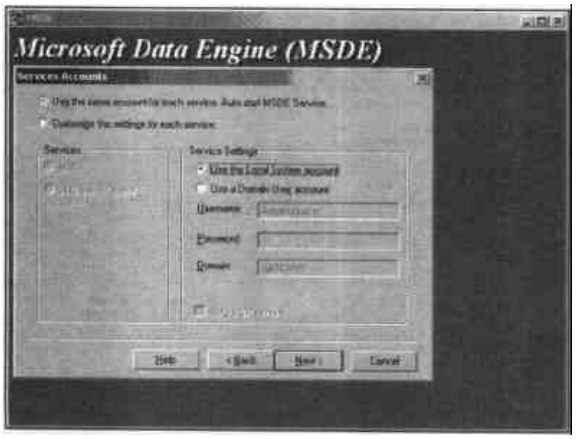


图　B-7

(10) 安装向导现在显示一个屏幕，开始复制，单击 Next 按钮。

(11) 接着显示一个信息框，说明正在复制的文件，如图 B-8 所示。之后就显示过程信息。

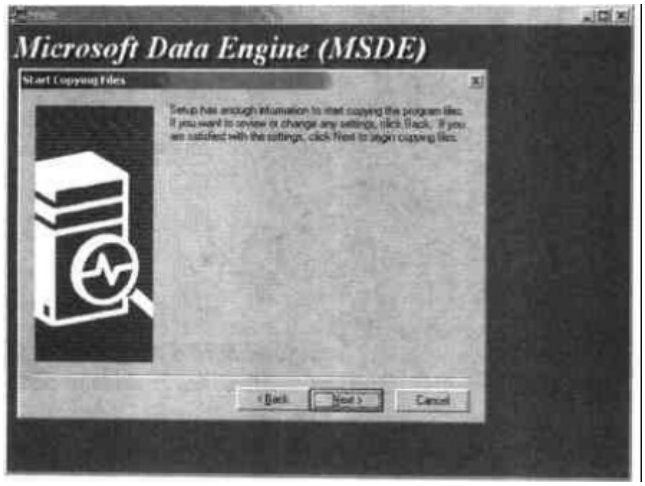


图　B-8

(12) 安装向导显示一个 setup complete 窗口，如图 B-9 所示，单击 Finish 按钮。

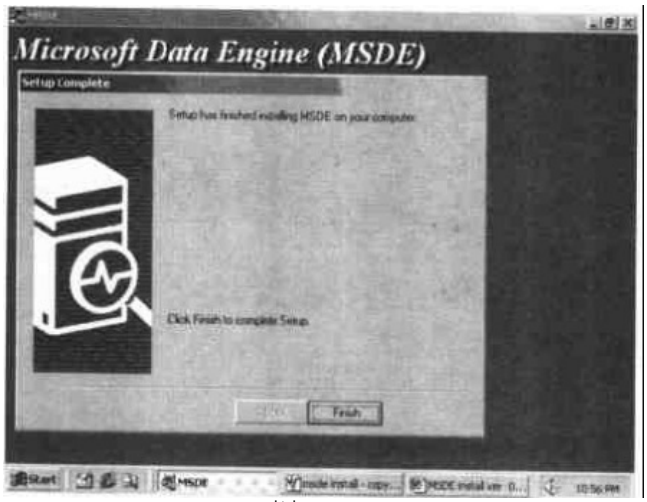


图　B-9

(13) 可能通过选择 Start | programs | MSDE | Service Manager，启动 MSDE。

注意：

MSDE 的服务管理器与 SQL Server 产品的服务管理器没有区别。

(14) 如果需要，可选择自动启动。

注意：

MSDE 大约要占用 5.3 MB 的磁盘空间，而服务管理器要占用 2.3MB 的磁盘空间。所以如果不打算使用它，自动启动会占用更多的空间，所以应尽可能手动启动。

(15) 单击带有绿色箭头的 Start 选项，启动 MSDE 服务。

运行服务管理器时，在系统盘的右端有一个服务器图标。当运行 MSDE 时，该图标上会有一个绿色箭头，当停止运行 MSDE 时，该图标就会显示一个红色框，如图 B-10 所示。

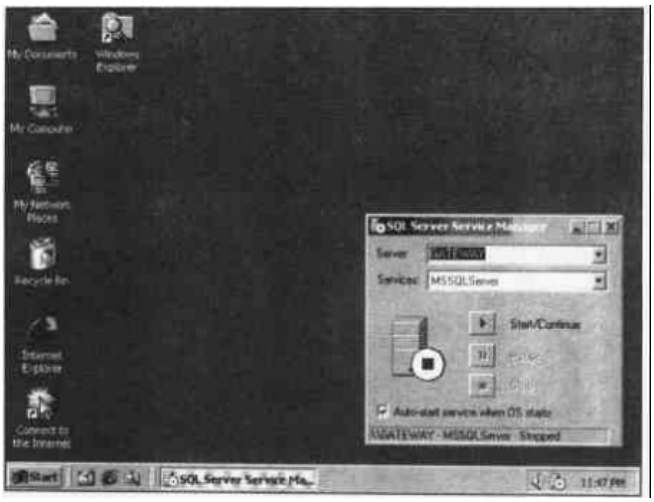


图 B-10

要查看版本，右击系统盘上的 MSDE 服务器图标，选择 About。此时可看到 SQL Services Manager 的版本是 7，如图 B-11 所示。

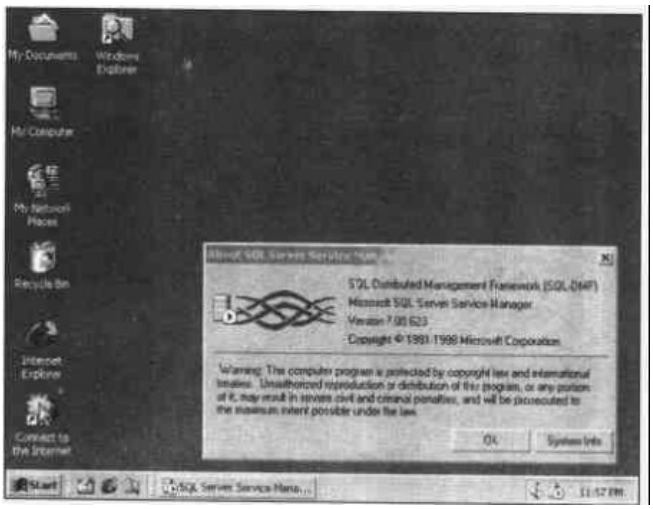


图 B-11

B.2 把 Northwind 导入 MSDE

MSDE 没有安装任何样例数据库，所以需要导入该数据库，以运行第 18 和 19 章中的范例。MSDE 和 SQL Server 附带了一个导入导出数据的实用程序，这样就可以使用该实用程序把普通的 Northwind 数据库(nwind.mdb)导入 MSDE，该数据库由 Access 和 Visual Basic 6 提供。

(1) 搜索 Northwind 数据库，选择 Start | Search | Files and folders | File，在 Named:框中键入 nwind.mdb。把文件复制(不是移动)到文件夹 C:\BegCSharp\AppendixB 中(该文件夹应自己创建)。

(2) 选择 Start/programs/MSDE 或 MS SQL Server/Import and Export Data。查看显示出来的 Data Transformation Wizard 屏幕，如图 B-12 所示。

(3) 单击 Next 按钮，进入 Source 屏幕，如图 B-13 所示，把 Data Source 设置为 Microsoft

Access，File name 设置为 nwind.mdb(需要找到 Northwind 的副本，该副本在 C:\BegCSharp\AppendixB 目录下)。

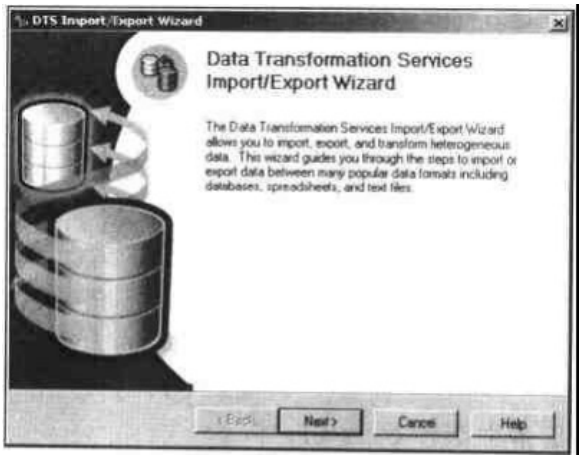


图　B-12

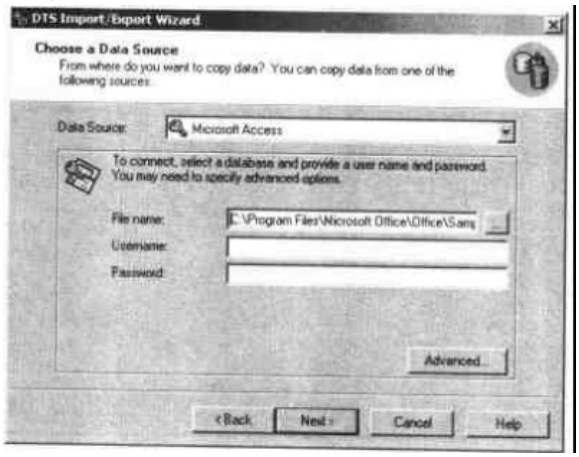


图　B-13

(4) 单击 Next 按钮，进入 Destination 屏幕，如图 B-14 所示，把 Destination 设置为 Microsoft OLEDB Provider for SQL Server，Server 设置为(local)，选中 SQL Server Authentication 单选按钮，将 Username 设置为 sa (不需要密码)。

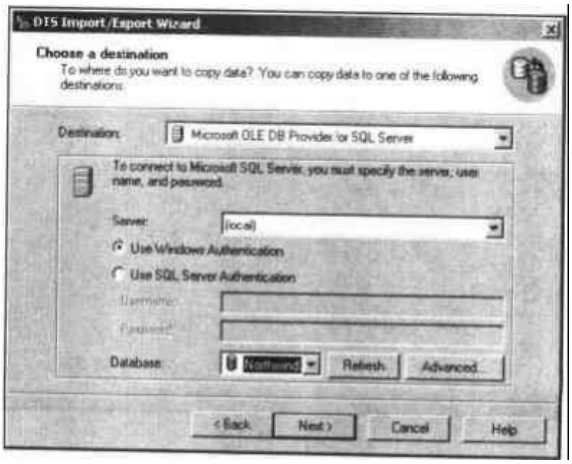


图　B-14

单击 Next 按钮前，选择<new> 作为数据库，打开 Create Database 对话框，如图 B-15 所示，在这里输入名称(我们使用 Northwind)，把 Data file size 设置为 4 MB，Log file size 设置为 1 MB。

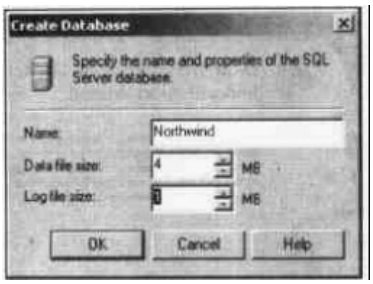


图 B-15

(5) 单击 Next 按钮，进入到 Method 屏幕，选中 copy table 单选按钮，如图 B-16 所示。单击 Next 按钮，再在下一个屏幕上选择 copy all。

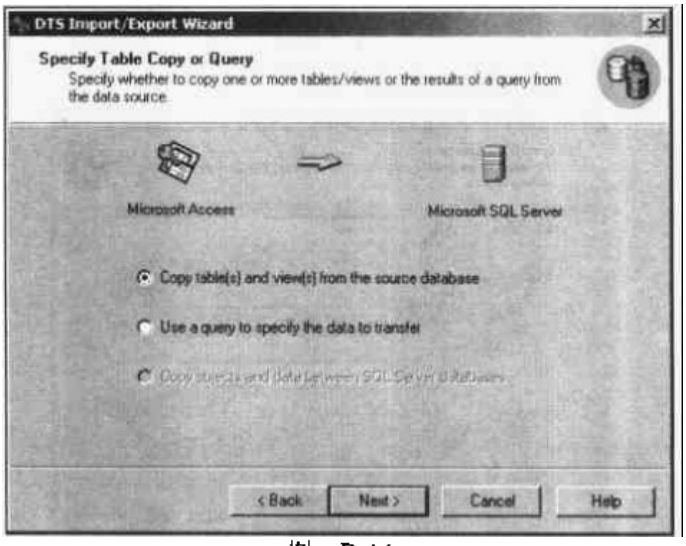


图 B-16

(6) 单击 Next 按钮，选中 Runimmediately 复选框，立即运行 DTS，如图 B-17 所示。

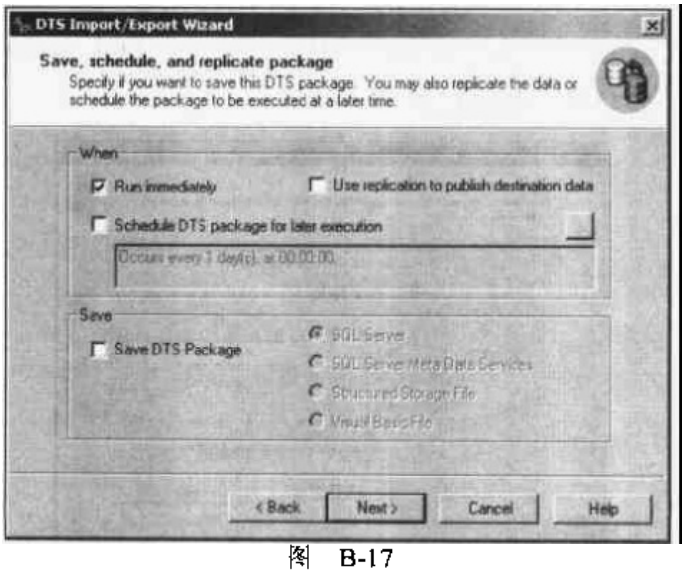


图 B-17

(7) 再次单击 Next 按钮，显示 Summary 列表框，如图 B-18 所示，单击 Finish 按钮。

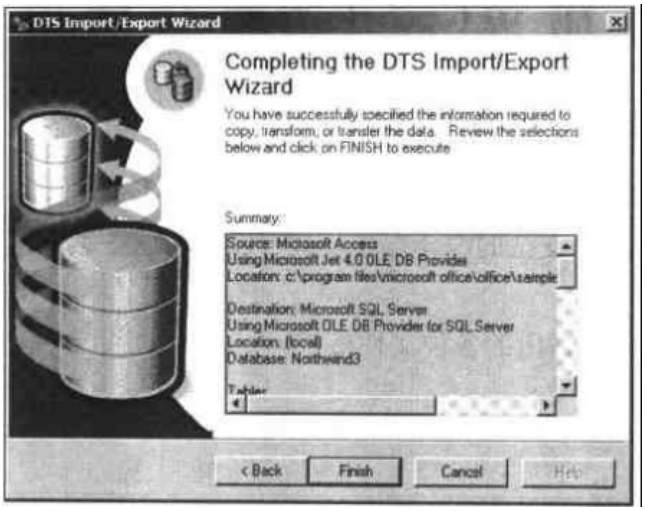


图　B-18

(8) 查看成功注意事项，再单击 Done 按钮。

(9) 注意查询、报表等还没有复制。

(10) 打开 Windows Explorer，注意此时，在 C:\MSSQL7\Data 下有一个 Northwind.mdf(不是.mdb)文件。

附录C　C#编译选项

本附录列出了各个C#编译器选项，如果编译C#工程时没有使用Visual Studio.NET，或者要执行Visual Studio.NET不支持的编译操作，就可以使用这些编译选项。它们根据类别的不同放在不同的列表中。

表C-1列出了编译器能输出的各种文件格式。

表　C-1

选　项	用　途
/doc:<filename>	处理XML文档注释(用3个斜杠///表示)，并输出为指定的XML文件
/nooutput	编译代码，但不创建输出文件；当控制台显示警告和错误时，该选项用于调试
/out:<filename>	指定输出文件的名称。如果没有指定，编译器就会生成一个标准的.exe文件。其名称与源文件名相同(但扩展名不同)
/target:<option> /t:<option>	指定输出文件的格式，有4个选项： • exe: 生成一个标准的可执行文件(默认设置) • library: 生成一个代码库(DLL) • module: 生成一个代码模块(没有清单的装配件)，该模块以后会添加到装配件中(使用/addmodule) • winexe: 创建Windows可执行文件

除非指定了/target:module选项，否则编译器就会给创建的EXE添加一个装配件清单(如果没有创建EXE，就添加第一个DLL)。注意/target可缩写为/t。

表C-2解释了决定编译器优化的命令行选项。

表　C-2

选　项	用　途
/optimize<+|->　/o<+|->	启用或禁用编译器进行优化，创建较小、较快、较高效的输出，优化在默认情况下是禁用的。要启用优化，可使用语法： /optimize or /optimize+ 用禁用优化，可使用语法： /optimize -

表C-3描述的选项可在创建和引用.NET装配件时使用。

表 C-3

选 项	用 途
/addmodule:<module>	指定在特定装配件中包含一个或多个模块。如果指定了多个装配件，就应使用分号来分隔它们。整个选项不能应用于 Visual Studio.NET
/nostdlib<+|->	指定是否导入标准库(mscorlib.dll)，默认值是导入该库。 如果要执行自己的 System 名称空间和类，可以让编译器不加载标准库。其语法是： /nostdlib 或/nostdlib+ 导入它的语法是： /nostdlib -
/reference:<assembly> /r:<assembly>	从装配件文件中导入元数据。可以指定装配件的完整路径，或者可以指定的 Path 环境变量来决定路径，或者可以指定启动当前工程的相对路径。如果指定了多个文件，应使用分号分隔它们

表 C-4 列出了应用于调试和错误检查的选项。

表 C-4

选 项	用 途
/bugreport:<filename>	创建指定的文件，该文件中包含由编译器生成的错误信息。文件包含的内容有： • 源代码的副本 • 编译器选项的列表 • 编译器版本、操作系统等的信息 • 编译器的输出 • 问题和可能的解决方案(可选) • 这个选项不能用于 Visual Studio.NET
/checked<+|->	指定整数溢出是否会引发一个运行错误。它仅能应用于 checked 和 unchecked 块作用域之外的代码。在默认状态下它是禁用的。溢出检查的语法是： /checked 或 /checked+ 要禁用溢出检查，可使用下面的语法： /checked -
/debug<+|-> /debug:<option>	生成调试信息。要使用这个选项，使用下面的语法： /debug 或/debug+ 要禁用这个选项，使用下面的语法： /debug - 在默认状态下，调试是禁用的。如果指定应输出调试信息，就可以使用两个选项，指定生成的调试信息类型： • /debug:full: 可以把调试程序关联到执行的程序上 • /debug:pdbonly: 在调试程序中启动程序时，允许调试源代码，但当运行程序关联到调试程序上时，仅显示装配件

(续表)

选　项	用　途
/fullpaths	指定包含错误的文件的完整路径。这个选项不能用于 Visual Studio.NET
/nowarn:<number>	禁止编译器生成特定的警告。<number>选项指定禁用的警告号。如果指定了多个警告号，就用逗号分隔开。这个选项不能用于 Visual Studio.NET
/warn:<option> /w:<option>	设置要显示的最低警告级别。其选项是： • 禁止所有的警告 • 仅显示严重警告 • 显示严重警告和中等级别的警告 • 显示严重警告和中低等级别的警告 • 显示所有的警告，包括信息性的警告
/warnaserror<+ | - >	把所有的警告当作错误。为此，可使用下面的语法： /warnaserror 或/warnaserror+ 要禁用这个选项，可使用下面的语法： /warnaserror - 该选项在默认状态下是禁用的

表 C-5 说明了如何设置预处理器指令。

表　C-5

选　项	用　途
/define:<name> /d:<name>	定义由<name>指定的预处理器符号

表 C-6 列出了与外部资源相关联的选项。

表　C-6

选　项	用　途
/linkresource:<filename> /linkres:<filename>	创建指定.NET 资源的链接。两个可选的其他参数(用逗号分开)如下： • identifier: 资源的逻辑名。该名称用于加载资源(默认情况下是文件名) • mimetype: 表示资源的媒介类型的字符串(默认为空) 这个选项不能用于 Visual Studio.NET
/resource:<filename> /res:<filename>	把.NET 指定的资源内嵌到输出文件中。两个可选的其他参数(用逗号分开)如下： • identifier: 资源的逻辑名。该名称用于加载资源(默认情况下是文件名) • mimetype: 表示资源的媒介类型的字符串(默认为空)
/win32icon:<filename>	把指定的 Win32 图标 (.ico) 文件插入到输出文件中
/win32res:<filename>	把指定的 Win32 资源(.res) 文件插入到输出文件中。这个选项不能用于 Visual Studio.NET

表 C-7 列出了各种杂项编译器选项。

表　C-7

选　项	用　途
@<filename>	指定一个文件，该文件包含所有的编译器选项和由编译器处理的源文件，就好像它们是在命令行中输入的
/baseaddress:<address>	指定首选的基地址，在该地址上可加载 DLL。<address>的值可以是十进制、十六进制或八进制
/codepage:<id>	指定用于编译过程中所有源代码的代码页面(在<id>选项中传送的值)。如果在 C#文件中使用的不是系统默认的字符集，就可以使用这个选项。该选项不能用于 Visual Studio.NET
/help /?	给标准输出列出编译器选项。该选项不能用于 Visual Studio.NET
/incremental<+ | - > /incr<+ | - >	允许增加源代码文件的编译，它只编译自从上次编译以来有变化的那些函数。上次编译的状态信息存储在两个文件中：存储调试信息的.dbg 文件(如果指定了/debug ，就是.pdb)，和存储状态信息的.incr 文件。要使用这个选项，可使用下面的语法： /incremental or /incremental+ 要禁用这个选项，可使用下面的语法： /incremental - 在默认状态下这个选项是禁用的
/main:<class>	如果源代码中存在多个 Main()方法，该选项就指定 Main()方法的位置
/nologo	禁止输出编译器标题信息。这个选项不能用于 Visual Studio.NET
/recurse:<dir\file>	搜索要编译的源文件的子目录。有两个选项： • dir (可选):开始所在搜索的目录或子目录。如果没有指定，该目录或子目录就是当前工程所在的目录 • file: 要搜索的文件。可以使用通配符
/unsafe	允许在编译代码的过程中使用 unsafe 关键字