

PROGRAMMER TO PROGRAMMER™

C# Class Design Handbook

C# 类设计手册

Richard Conway

Teun Duynstee

杨 浩

等著

译



清华大学出版社

C# Class Design Handbook

C# 是 Microsoft 的一种新的面向对象语言，它充分利用了 .NET 类型系统，在 C# 中，.NET 编程的核心任务是设计类。所有的 C# 代码都涉及到了 .NET 类型系统，因为所有的代码都包含在类和参与构建 .NET Framework 类层次结构的其他类型中。本书将指导用户高效地设计这些类，学习如何控制类，以及 C# 如何把这些类转换为可执行的代码。能设计出不必重复访问和重复修正的类可以说是一种艺术。本书将帮助读者深入理解在设计类时应做出的每个决策，更好地利用 C# 的功能，创建出健壮、灵活和可重用的类。

本书主要内容

- 类型在 .NET 中的作用
- 可以用 C# 创建的各种类型
- 作为程序逻辑包装器的方法的基本功能
- C# 的特殊方法语法：属性、索引器和运算符
- 构造函数的作用和用法
- 对象的清理和释放
- .NET 基于委托的事件系统的工作原理
- 如何在类型中控制和利用继承功能
- 通过命名空间和程序集组织逻辑代码和物理代码

本书读者对象

本书不是一本纯理论的图书，也不论述 UML 对象建模，而是提供日常问题的解决方案，并列举了大量的代码示例。其目标是帮助 C# 开发人员尽快创建出更好、更整洁、可重用性更高、更健壮的类，避免以后的重复访问和重复设计。

ASP Today
www.asptoday.com

C# Today
www.csharp.today.com

VB Today
www.vbtoday.com

p2p.wrox.com
The programmer's resource center

ISBN 7-302-06832-1



9 787302 068327 >

定价：42.00 元

Recommended
Computer Book
Categories

Programming

ASP.NET, .NET

wrox direct

C#类设计手册

Richard Conway

Teun Duynstee 等著

杨 浩 译

清华大学出版社

北 京

内 容 简 介

C#遵循面向对象编程语言设计的传统方式,具有很好的继承性,并提供了许多令人惊异的性能。本书主要讲解用C#设计类时的一些具体细节。首先描述类型的含义,如何将类关联到.NET类框架,然后讨论组成类型的类型成员。本书的大部分内容介绍C#为设计类型成员(方法、构造函数、属性、运算符和事件)提供的不同机制,最后研究如何将类型集中在一起组成程序集。

本书适用于希望深入理解.NET平台的C#开发人员,通过对本书的学习,可以进一步理解C#为设计类而提供的各种机制。

EISBN: 1-86100-828-7

C# Class Design Handbook

Richard Conway Teun Duynstee, et al.

Copyright©2002 by Wrox Press Ltd.

Original English Language Edition Published by Wrox Press Ltd.

All Rights Reserved.

本书中文简体字版由英国乐思出版公司授权清华大学出版社出版。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。

北京市版权局著作权合同登记号 图字 01-2003-0820

图书在版编目(CIP)数据

C#类设计手册/(英)康威等著;杨浩译. —北京:清华大学出版社, 2003.7

书名原文: C# Class Design Handbook

ISBN 7-302-06832-1

I. C... II. ①康... ②杨... III. ①C语言—程序设计 IV. TP312

中国版本图书馆 CIP 数据核字(2003)第 050898 号

出 版 者: 清华大学出版社

地 址: 北京清华大学学研大厦

<http://www.tup.com.cn>

邮 编: 100084

社 总 机: 010-62770175

客户服务: 010-62776969

组稿编辑: 曹康

文稿编辑: 于平

封面设计: 康博

版式设计: 康博

印 刷 者: 北京鑫海金澳胶印有限公司

发 行 者: 新华书店总店北京发行所

开 本: 787×1092 1/16 印 张: 20.75 字 数: 430 千字

版 次: 2003 年 7 月第 1 版 2003 年 7 月第 1 次印刷

书 号: ISBN 7-302-06832-1/TP·5073

印 数: 1~4000

定 价: 42.00 元

前言

C#遵循编程语言设计的传统方式,它深受C++和Java的影响,甚至还受到Delphi 和 Visual Basic 的影响,因此具有很好的继承性。它提供了许多开发人员都很熟悉的功能,还提供了许多令人惊异的性能。

对于具有 Visual Basic 6 或 ASP VBScript 背景、不熟悉面向对象、使用花括号的 C 系列语言的程序员来说,常常觉得 C#中的面向对象特性令人畏缩。而使用其他面向对象平台(如 C++或 Java)的程序员,则觉得 C#的某些附加功能令人惊讶或迷惑,有些程序员则认为,看似熟悉的语法却以完全不同的方式使用。

本书将揭开 C#面向对象模型的秘密,展示如何使用 C#语言来创建类(以及其他类型)。在 C#中编写的任何代码都是类,所有的逻辑都是类的方法,程序的状态总是与内存中类实例的字段值相关。因此,要理解如何创建这些类,最好的方法就是养成良好的 C#编程习惯。

本书将介绍 C#中可以用于在代码中声明类和类成员的选项,以及这些选项对使用类的代码的影响。阐述类在用于不同目的(如用于展示、在程序员自己的代码中使用)时的不同编码方式,如何确保类以设计好的方式使用,如何以一致、富有逻辑、可预测和用户友好的方式展示类的功能,以便其他代码使用。

本书读者对象

本书面向希望探索.NET 平台全部功能的 C#开发人员。如果希望定义自己的数据类型,建立自己的类层次结构,或建立带有健壮接口的类,就需要深入理解 C#为设计类提供的机制。这也是本书的主题。

本书假定读者已经使用 C#编写过代码,熟悉其基本语法,而且编写出了能运行的代码。读者还应该熟悉所选择的开发工具,知道如何编译和运行 C#代码。

读者也应了解.NET 基本的面向对象机制——例如,哪些对象是类的实例,如何实例化对象,如何访问对象的方法和属性。然而,我们会在讨论时扼要讲述大多数 C#关键字的含义和语法。

本书主要内容

每次利用 C# 编写代码时，都是在编写一个类——这是无法避免的。本书讨论程序员在实际编写程序时所做的一些决定，也就是对设计类所做的一些决定。因此，在编写一个方法，并决定把它定义为 `static`、`public` 还是 `private`，以及它应该接受什么参数等时，本书将从这些决定对类设计的影响方面来研究它们。

本书并不讨论我们每天都在编写的那些代码，并搞清楚它在做什么。不要只从效果方面考虑每个 C# 关键字或语法结构，而要考虑它如何达到那个效果。本书会介绍代码如何编译成 .NET 类型，如何定义类成员，如何继承类成员，如何把类聚合到程序集中，如何控制类实例的创建，以及有效地编写类等。

本书不包括的内容

本书不讨论面向对象的分析和设计、UML 建模或设计模式，尽管我们肯定会遇到这些问题，对此读者可参阅其他书籍。本书没有介绍如何提出业务方面的问题，以及应该编写哪些类来解决这些问题。它重点讨论实现的问题是：如何编写一个类来提供特定的功能。

本书也不是对面向对象的基本介绍，因为任何 C# 程序员都熟悉这样的思想：即使不定义自己的类，也可以拥有一个对象的实例，调用它的方法并访问属性。如果您使用对象很轻松，那么本书不一定适合您。

本书具体内容

本书将自上而下地介绍在 .NET 中究竟由什么组成一个类。首先描述类型是什么，类如何关联到 .NET 类框架，接下来讨论组成类型的类型成员。本书的大部分内容介绍 C# 为设计类型成员(方法、构造函数、属性、运算符和事件)提供的不同机制，最后研究如何将类型集中在一起组成程序集。

以下是各章的主要内容：

第 1 章——定义类型

阐述什么是类型，类型在 .NET 中发挥什么作用，以及存在哪些类型。并分析可以在 C# 中声明的不同类型，以及它们如何映射为 .NET 类型。

第 2 章——类型成员

介绍类型成员：什么是类型成员，如何定义类型成员，如何用 C# 关键字修饰它们。还要介绍由 .NET 根类 `System.Object` 派生出来的每个类型所继承的类型成员。

第 3 章——方法

方法是 .NET 应用程序的重要组成部分，它包括所有程序逻辑。本章研究对所有方法通用的行为，以及在 C# 中如何定义方法。还要讨论参数如何传递给方法，方法如何返回值或生成异常，从而与调用它们的代码通信。

第 4 章——属性和运算符

属性(标量属性和索引属性)是一种机制，它允许创建指定的方法以访问属于类型的数据。运算符是特殊的方法，它允许类型使用者通过基于运算符的简洁语法把它们合并起来。本章研究如何实现属性，索引属性如何工作，以及运算符的创建和使用。

第 5 章——构造函数和对象生命周期

构造函数是比较特殊的方法，调用它可以初始化类型的新实例。本章介绍如何编写这些特殊的方法，如何使用它们来控制可以创建类型实例的代码。还要讨论对象的克隆，运算符的转换，以及一些控制类实例创建的常见编码技巧。

第 6 章——事件和委托

C# 中最复杂的类型成员是事件，C# 类型中最复杂的是委托。事件建立在委托的基础之上，把这二者结合起来常常是程序员感到相当困难的工作。本章阐述委托如何工作，.NET 如何通过委托字段和指定方法来提供它的事件基础结构。

第 7 章——继承和多态性

类型并不仅是它的成员总和，而且还包括从超类中继承的所有成员。本章研究 .NET 类型继承的工作机制，成员何时可以继承，何时不能继承，以及如何使用 C# 控制和利用继承。还论述接口和抽象类的作用和用法。

第 8 章——代码组织和元数据

在 C# 中编写类时，必须决定在何处放置这个类，这是指在逻辑上该类要放在命名空间结构里，在物理上还要放在源文件里，最终要放在 .NET 程序集里。本章将讨论这些问题。还要了解如何使用 .NET 元数据把数据添加到类中，这些数据可能对使用该类的其他代码有用。如何给类编写文档，为其他程序员提供如何使用该类的信息。

使用本书的前提

在学习本书前，读者应该能够编译和执行在 C# 中编写的代码。这意味着读者需要：

- .NET Framework SDK, 可从 Microsoft 的 MSDN 站点 (<http://msdn.microsoft.com>) 的 Software Development Kits 类别中获得。在本书出版时，下载页面可通过下面的 URL 获得：

[http://msdn.microsoft.com/downloads/sample.asp?
url=/msdn-files/027/000/976/msdncompositedoc.xml](http://msdn.microsoft.com/downloads/sample.asp?url=/msdn-files/027/000/976/msdncompositedoc.xml)

- 带有 Visual C# .NET 的 Visual Studio .NET 版本。

Visual C# .NET IDE 的 2002 版下面的 Microsoft 产品提供了：

Microsoft Visual C# .NET Standard

Microsoft Visual Studio .NET Enterprise Architect

Microsoft Visual Studio .NET Enterprise Developer

Microsoft Visual Studio .NET Professional

产品主页是 <http://msdn.microsoft.com/vstudio/>。

其他平台的 .NET 版本正在开发中，Mono 工程(<http://www.go-mono.com/>)支持在 Linux、UNIX 和 Windows 平台上的 C# 编译。Mono 代码不能访问所有的 Microsoft .NET 类库，但遵循与 Microsoft 的 C# 相同的语法规则，所以本书中的代码也可以应用于 Mono。

目 录

第 1 章	定义类型	1
1.1	类型	1
1.1.1	.NET 类型系统	2
1.1.2	值类型和引用类型	4
1.2	值类型	8
1.2.1	基本类型	8
1.2.2	用户定义的值类型(结构)	14
1.2.3	枚举	19
1.3	引用类型	24
1.3.1	类类型	25
1.3.2	委托	32
1.3.3	数组	32
1.3.4	字符串	35
1.4	把值类型用作引用类型	38
1.5	接口	41
1.6	小结	42
第 2 章	类型成员	44
2.1	常量	45
2.2	字段	50
2.3	属性	53
2.4	方法	56
2.5	属性与方法	57
2.6	静态类型成员	57
2.7	事件和委托	61
2.8	运算符	62
2.9	构造函数	62
2.10	使用 System.Object	63
2.10.1	GetType()	63
2.10.2	ToString()	64

2.10.3	Equals()	66
2.10.4	GetHashCode()	68
2.10.5	Equals()和 GetHashCode()	70
2.11	小结	70
第3章	方法	72
3.1	方法的调用	72
3.2	方法的作用域和可见性	74
3.3	方法类型	74
3.3.1	实例和静态方法的例子	74
3.3.2	实例和静态方法的最佳做法	76
3.4	参数和参数化方法	76
3.4.1	参数类型	76
3.4.2	按引用类型传送与按值类型传送	81
3.4.3	可变长度的参数列表	90
3.4.4	传送字符串——不变的对象	91
3.4.5	传送数组和上限检查	92
3.4.6	传送枚举值	92
3.5	方法重载	93
3.6	异常的处理	96
3.6.1	什么是异常	97
3.6.2	Try...Catch...Finally	98
3.6.3	抛出自己代码中的异常	99
3.7	方法和 MSIL 代码	102
3.8	设计小结	108
第4章	属性和运算符	109
4.1	C#中的属性	109
4.1.1	C#属性的语法	110
4.1.2	标量属性	111
4.1.3	索引器	124
4.2	C#中的运算符	135
4.2.1	表达式中的运算符	135
4.2.2	运算符重载的语法	136
4.2.3	运算符和类	137
4.2.4	运算符和结构	149

4.2.5	运算符重载的最佳做法	154
4.2.6	运算符小结	154
4.3	小结	155
第 5 章	构造函数和对象生命周期	156
5.1	对象的创建	158
5.2	销毁对象	162
5.3	使用构造函数	166
5.3.1	把构造函数链接在一起	167
5.3.2	调用基类构造函数	168
5.3.3	私有构造函数	170
5.3.4	静态构造函数	172
5.4	转换运算符	173
5.5	对象的克隆	177
5.5.1	使用复制构造函数	177
5.5.2	ICloneable 接口	179
5.6	串行化和并行化	181
5.7	设计模式	184
5.7.1	Singleton 模式	185
5.7.2	Abstract Factory 模式	187
5.7.3	懒惰的初始化	191
5.7.4	Copy-On-Write	192
5.8	小结	193
第 6 章	事件和委托	194
6.1	委托	194
6.1.1	创建和使用简单委托	195
6.1.2	创建和使用多播委托	201
6.1.3	创建和使用异步委托	210
6.2	事件	217
6.2.1	事件体系结构	218
6.2.2	事件的发布和订阅	219
6.2.3	事件在 .NET Framework 的类型接口中的作用	234
6.3	小结	236
第 7 章	继承和多态性	237
7.1	继承	238

7.1.1	继承的层次结构	239
7.1.2	重写基类的操作	240
7.1.3	抽象类和接口	240
7.1.4	创建类层次结构	244
7.2	基于接口的编程	257
7.2.1	定义接口	258
7.2.2	实现接口	258
7.2.3	多态性和接口	260
7.2.4	多重接口继承	262
7.2.5	IL 中的接口	262
7.2.6	界定接口方法的作用域	264
7.2.7	转换为不同的接口类型	265
7.2.8	名称的多义性和遮蔽	266
7.3	小结	267
第 8 章	代码组织和元数据	269
8.1	使用命名空间建立应用程序的结构	269
8.2	理解程序集	287
8.2.1	什么是程序集	287
8.2.2	元数据	288
8.3	把应用程序部署为程序集	298
8.3.1	部署单程序集应用程序	299
8.3.2	使用私有程序集部署应用程序	300
8.3.3	部署共享程序集	304
8.4	为程序集生成文档	310
8.4.1	使用程序集级别的属性	311
8.4.2	XML 文档	313
8.5	小结	318

第1章 定义类型

C#是一种面向对象编程语言，它的一个设计指导原则是类型的安全性。在面向对象的分析和设计中，我们要确定系统中最重要的对象，考虑它们之间的关系。在C#中编程时，类是其主要机制，用于定义在程序运行期间才存在的对象的行为。C#还提供了许多方式来封装定义应用程序的代码，而不仅仅是在类中封装代码。

只要在C#中编码，所编写的代码就是类型，这些类型表示对象的行为和数据存储需求的组合。当程序运行时，将创建类型的实例(用以分配所需的数据存储空间)，并使类的功能可用。在.NET环境中，类型安全的代码只能访问它允许访问的内存位置，此时，类型以定义好的、允许的方式交互。这对于编写出安全、稳定的应用程序来说非常重要，可以确保编写得不好的代码不会产生很大的破坏，而是按照分配好的角色来运作。

本书旨在帮助C#开发人员深入而细致地理解如何创建出设计优秀的类，确保它在.NET Framework内部正确、一致地运行。通过研究一些例子，给出设计和开发阶段所做决策的结果，并指出C#与其他面向对象语言，如C++和Java之间的异同点。

本书首先讨论什么是类型。本章将研究.NET的类型系统，以及开发人员可以使用的类型。

1.1 类型

在编程中，我们使用术语“类型”来描述某类值。例如，C++和Java程序员比较熟悉如int、float和double这样的类型。对于每种类型，编译器都必须了解下面的信息：

- 创建这种类型的值时，应该分配多少内存
- 使用这个值可以执行什么操作

类型的概念是“强类型”编程语言的基础，包括所有.NET语言。在强类型语言中，存储在每个变量里的值的类型在编译时是已知的，因此编译器可以预测使用每个变量的方式，这样，如果弄错了，编译器可以通知我们。

类型是一个契约。某个类型的变量将保证，它包含给定类型的值所期望拥有的数据，而且能以这种类型的值所希望的方式来处理它。类型的这种契约有时称为接口。

C++用户注意：

如果习惯在头文件中定义C++类的接口，就应注意C#没有头文件。C#类型的定义

作为元数据包含在已编译的程序集内。C#编译器也不考虑源代码中类型定义的顺序。

对计算机来说，所有数据都只是 1 和 0 字符串。如果程序中拥有一个变量，这个变量保存的只是某个二进制数字而已。因此，当计算机把这个变量显示到屏幕上，对它执行计算，或检索变量的某个属性时，计算机需要知道变量包含的类型，以确定如何解释它的值，并响应用户的请求。例如，整型和单精度浮点数都可以存储在 4 个字节的二进制数据中。例如下面的 4 个字节：

00110110 11011011 10001010 01110100

如果这个值解释为整型，就表示数字 920357492；解释为单精度浮点值，它的近似值为 6.5428267E-6。因此，如果变量包含这个二进制数字，让.NET 为它加 1，则结果不仅依赖于变量里的值，还依赖这个变量的类型。

类型描述了内存中 1 和 0 字符串的含义。它允许比较两个整型值，判断一个值是否大于另一个值，获取表示值的字符串，或以特定方式修改值。

1.1.1 .NET 类型系统

.NET Framework 包含大量的类型，来帮助我们用.NET 语言进行软件开发。我们在自己的代码中定义和使用的所有类型都必须遵循.NET Framework 标准，以确保它们在运行时环境中正确运行。定义这些标准的是两个重要的规范：Common Type System (CTS，通用类型系统)和 Common Language Specification (CLS，公共语言规范)。

1. Common Type System (CTS)

CTS 告诉编译器的编写人员如何声明和使用用于.NET 运行时中的类型。所有的.NET 语言都必须遵循它定义的规则，生成的编译代码才能在 Common Language Runtime (CLR，公共语言运行时)内运行。CTS 提供了运行各个.NET 语言的面向对象架构。有这个通用的架构是至关重要的，它能确保运行期间的类型安全和代码安全，还能保证跨语言的集成。实际上，CTS 是.NET Framework 的核心。

图 1-1 显示了 CTS 的结构。

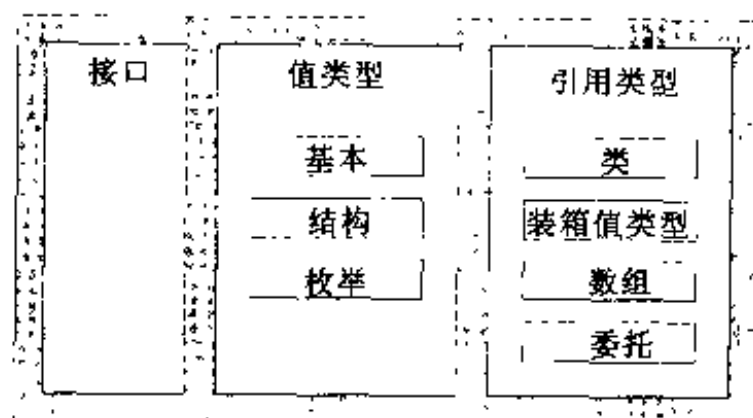


图 1-1

图 1-1 说明了, CTS 清楚地区分了值类型和引用类型(如下所述)。它还允许定义纯粹的接口, 接口是契约的简单定义, 由另一个类型(或一些类型)来实现——它区分开了契约的定义和契约的实现。其他类型都是由这两部分组成的。

C++用户注意:

接口与纯粹的抽象类相似。与 C++不同, C#类只能继承一个基类, 但可以继承多个接口。

Java 用户注意:

与 Java 相同, C#支持类的单一继承和接口的多重继承。但与 Java 不同的是, C#允许显式地实现接口, 避免了命名冲突问题。稍后讨论这个问题。

2. Common Language Specification (CLS)

CTS 定义了类型在运行时环境中的创建和管理方式, 而 CLS 主要关注的是语言的交互操作性。CLS 描述了任何针对 .NET 运行时的编译器都必须支持的最小功能集。对于 C#开发人员, 理解 .NET 语言的无关性的意义是非常重要的。

通过 CLS 很容易在同一个应用程序中使用几种不同的编程语言。只要在与 CLS 兼容的语言中编译源代码, 就可以把它编译成标准的 .NET Framework 字节代码格式, 称作 Microsoft 中间语言(MSIL)。CLS 确保无论使用哪种语言, 某些类型和语言结构都将编译为相同的 MSIL。因此, 只要在代码中声明的所有公共接口中只使用 CLS 兼容类型, 就可以在同一个应用程序中混合使用 C#、Visual Basic .NET、Managed Extensions for C++、Visual J#, 或者 JScript。为此, 可以把 C#编译器构建为一种能检查代码, 并能在我们违反规则时提出警告的编译程序。

使用中间字节代码格式对 Java 开发人员来说很熟悉。Java 在托管的环境(Java 虚拟机)下运行之前, 通常编译为字节代码, C#和其他 .NET 语言也是这样, 在 .NET CLR 运行代码之前, 要编译为 MSIL。一个区别是 Java 允许字节代码在运行期间进行解释, 而不是编译为本机码, 而 MSIL 总是由 Just In Time (JIT, 即时)编译器编译, 或作为预 JIT 编译的程序集从 Global Assembly Cache (GAC, 全局程序集缓存)中加载。另一个重要的区别是 MSIL 的语言中立性自 .NET Framework 建立第一天起就内置其中了。如前所述, CLS 制定了一系列规则, 定义了不同语言编译为 MSIL 的方式, 这对语言的交互操作非常有利。

这里需要澄清一个问题。我们常常假定 C#语言与 .NET 运行时的联系非常紧密, CLS 兼容性是该语言的一个固有特性。而实际上, C#本身是一个类型安全、功能丰富、面向对象的语言, 它非常擅长于编写与 CLS 不兼容的代码。例如, 它的一些基本类型就与 CLS 不兼容。本书主要讨论 .NET 类型系统, 也会讨论 C#编译器在编译源代码时针对 .NET 运行时的方式, 例如, C#中的一些非 CLS 功能如何展现给其他与 CLS 兼容的

语言。

还要指出的是，语言的互操作性不仅仅是一个广告噱头。CTS 和 .NET 运行时一起支持的功能要比 CLS 定义的子集功能多得多。不同的语言会展示这些功能的不同子集，尽管它们包含的功能时常重叠。可以说，某种语言比较适合一个应用程序的某些部分。所有的语言都共享 CLS 定义的公共功能集，所以这些语言之间肯定有一个接口。

除了 .NET 语言之外，还有第三方供应商提供的许多 CLS 兼容语言，例如 COBOL (Fujitsu)、Perl 和 Python (ActiveState)，以及 Smalltalk (Quasar Knowledge Systems)。

在详细介绍值类型和引用类型之前，要知道 .NET 中的所有类型都是含义明确的。这包括枚举和接口。已编译的 .NET 程序集(.exe 或.dll)包含元数据，其中给出了程序集使用和定义的类型的所有信息。对于程序集中定义的那些类型，可以使用反射机制来了解它们的定义。这在开发阶段是非常有益的，而不需要用头文件或类型库来确定对象有什么属性和方法。在运行阶段，这些信息也是十分重要的，此时 CLR 使用这些信息动态地解析 JIT 编译过程中的方法调用。这些信息存储在一个已编译的 .NET 程序集中，由 CLR 扩展使用。第 8 章将详细论述元数据。

1.1.2 值类型和引用类型

值类型一般表示本质上是数字的实体，如日期和时间、颜色、屏幕坐标等等。这种实体可以用固定长度的简单二进制数字表示——惟一复杂的是对这些数字含义的解释。值类型通常比较小，其行为也比较简单，只提供一个用于读写基础值的接口。

1. 值类型和堆栈

简单值类型的一个例子是一个 8 字节长的整数，它可以用来表示一个范围很大的日期和时间。这个数字可以解释为距离固定时间的时间间隔偏移量。.NET 有 64 位 `DateTime` 类型，它表示某一时刻自公元 1 世纪 1 月 1 日 00:00:00 以来的 tick 数。tick 是时间单位，它是 100 个纳秒，或微秒的十分之一。用这种值表示时间的不同实例非常有效。于是，这种类型提供的功能允许从简单的二进制值中以可用的格式提取年、月、日、时、分、秒等。

这样的值类型实际上由非常少的数据构成，很容易在程序中传递 `DateTime` 信息。如果创建一个变量 `aDate` 来保存 `DateTime`，它将需要 8 个字节的存储空间。如果创建另一个变量 `bDate` 来保存 `DateTime`，它也需要 8 个字节的存储空间。现在，如果编写代码 `bDate = aDate;`，那么 `aDate` 中 8 个字节的数据可以很快复制并放入 `bDate` 中。

值类型在用作局部变量或成员字段时，就要在栈上分配空间。栈是内存中一个完全用于存储局部变量(在方法体内部声明的变量)的高效(但大小有限制)区域。上述操作可以用下面的方式来表示。首先，声明两个 `DateTime` 值类型，表示不同的日期：

```
DateTime aDate = new DateTime(1994, 08, 28);    // 28th August 1994
DateTime bDate = new DateTime(1996, 07, 23);    // 23rd July 1996
```

接着，把一个日期复制到另一个变量中：

```
aDate = bDate;    // copy date
```

运行线程的栈，其结果显示在图 1-2 中。左边是复制前的情况，右边是复制后的情况。

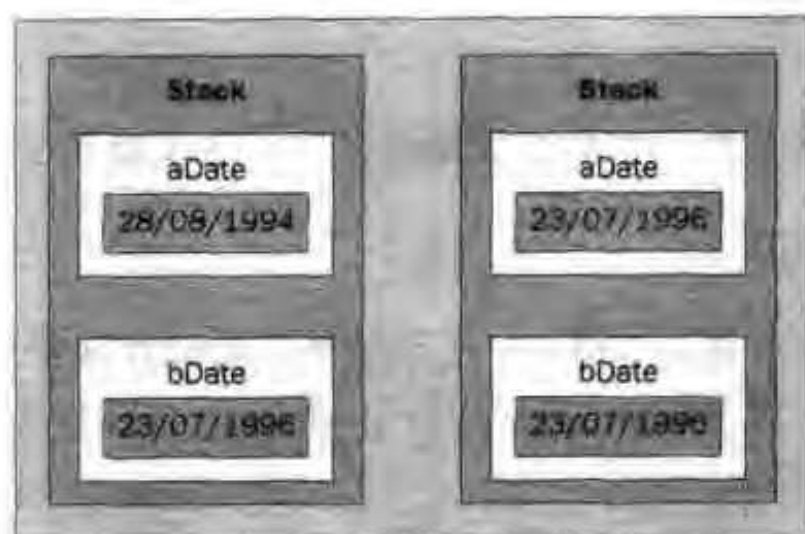


图 1-2

值类型采用按值复制的方式：在把一个值类型对象复制到另一个值类型对象中时，编译器对对象执行逐个字节的复制。当不需要考虑所创建对象的副本数目，而只需关注其值时，可以使用值类型。在把一个值类型对象传递给一个方法时，编译器会创建该对象的一个本地副本。在方法返回时，值类型对象的本地副本所使用的内存就会自动释放。也就是说，在使用值类型时有一些很重要的限制。

2. 引用类型和托管堆

由于值类型的数据大小固定，因此与复杂类型，如数组，相比，它们复制和处理的速度比较快，本章的最后将详细论述数组，现在只需知道数组没有固定的大小即可。在创建变量 `myArray` 来保存一个整型数组时，它需要多少字节的存储空间我们并不清楚——这依赖于放入其中的数组大小。但是，当 .NET 创建变量时，.NET 需要知道应分配给它多少空间。因此，.NET 使用固定大小的数据，即引用，来解决这个问题。引用值总是相同数量的字节，并指向系统内存的某个部分(即托管堆)中的一个位置。

托管堆所占的内存空间比栈大得多，但访问速度较慢。这类似于 C 运行时堆，但托管堆只用于分配内存；我们从来不专门考虑托管堆中内存空间的释放。CLR 会自动处理内存的释放问题(所以称为托管堆)。

如果创建一个有 8 个整数的数组(将需要 32 个字节的存储空间来存储数组的成员)，就会在托管堆中分配 32 字节。把这个数组放入变量 `myArray` 中，就是把数组在托管堆中的位置(对数组的引用)放入变量中。包含这个引用的变量 `myArray`(与其他局部变量一

样)存储在堆栈上。该引用指向的内存则位于托管堆上。

如果创建另一个变量 `myArray2` 保存一个整型数组，然后编写 `myArray2 = myArray;`，这个引用值就复制到 `myArray2` 中，这样 `myArray2` 指向的整型数组就和 `myArray` 完全相同。数组与类一样，都是引用类型。

下面用图表的方式来说明。首先，定义两个数组变量，初始化其中的一个，使它指向托管堆上的一个新实例：

```
int[] myArray = new int[8];  
int[] myArray2;
```

接着，把一个数组赋予另一个数组变量。记住，这仅复制了引用：

```
MyArray2 = myArray;
```

这些操作的结果显示在图 1-3 中。左边是赋值之前运行线程的栈和托管堆，右边是赋值之后的情况。注意引用变量 `myArray` 和 `myArray2` 现在都指向同一个数组实例。

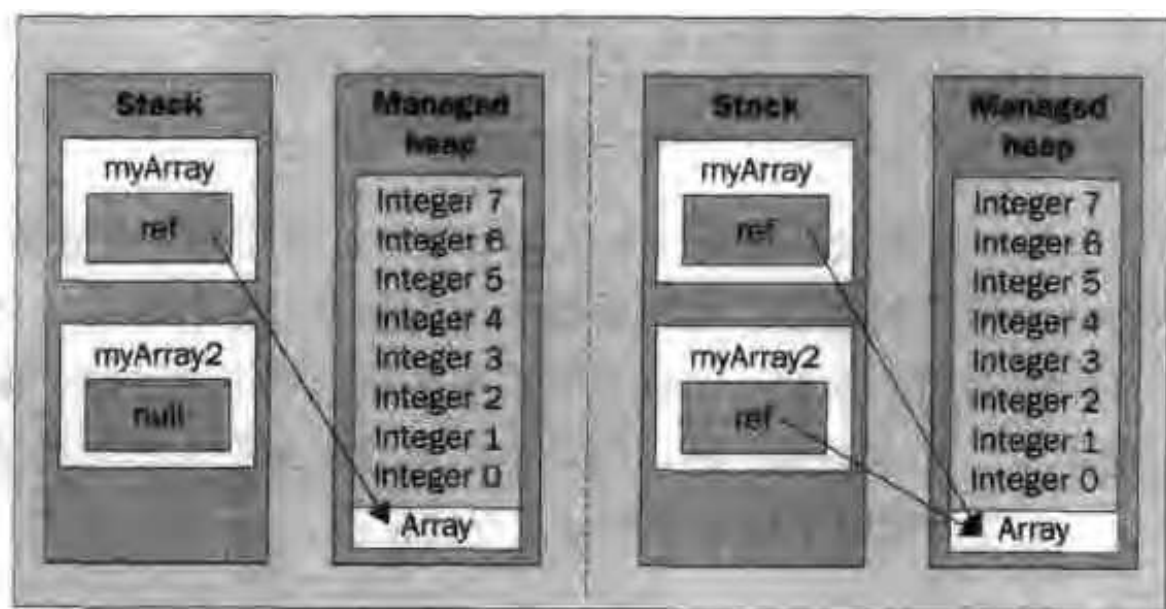


图 1-3

引用类型表示拥有惟一身份的实体——它们存在于托管堆的某个地方，由使用该实体的变量引用。例如，假定定义一个整型数组，并创建它的一个实例，CLR 就会给该对象分配内存，并返回一个对该新对象的引用。把这个数组复制到另一个变量中时，将只复制引用。这样，数组实例仍然只有一个，但有两个变量指向它。引用类型采用按引用复制的方式。同样，把引用类型传递给一个方法时，方法接收对象引用的副本，而不是其值的副本。

3. 值类型和托管堆

在许多 .NET 文献中，常常有“值类型存储在栈中”这句话，这会令人误解。例如，数组是一个引用类型，所以它存储在托管堆中。但如果声明一个含有值类型实体的数组，这些值会存储在数组中，所以它们实际上是作为值类型存储在托管堆上的。当从

数组中检索这些值时，不会获得对堆中值位置的引用，而将获得可在本地使用的来自堆栈值的副本。值类型和引用类型的区别是：引用类型存储在托管堆的惟一位置中，而值类型存储在使用它们的地方。

4. 理解.NET 内存的使用方式

现在，Java 开发人员会认为，值类型类似于基本数据类型，而引用类型类似于类，并在堆上分配内存空间；C++开发人员则认为，值类型是结构类型，引用类型是动态分配内存的引用变量或指针。这两种说法都不完全正确，但其中有一些要点应考虑。

对于引用类型，C++开发人员不应把它们当作指针。尽管在后台有时是这样，但在应用程序的运行过程中，CLR 会在托管堆上移动对象，随时调整引用。在运行期间我们看不到这些，从源代码中也看不出，如果编写代码时要在 C#和 C++之间来回转换，就会产生混淆。例如，在 C#中，访问值类型的语法与访问引用类型的语法没有区别。在 C++中，可以编写如下代码：

```
SomeClass *pClassOnHeap = new SomeClass();  
pClassOnHeap->DoSomething();  
delete pClassOnHeap;
```

但是在 C#中，为了保证向后兼容，“->”语法仅用于不安全的代码块。在正常情况下，“.”应只用于访问类型的方法，如下所示：

```
SomeClass pClassOnHeap = new SomeClass();  
pClassOnHeap.DoSomething();  
pClassOnHeap = null;
```

还要注意，在使用完变量后，并不显式删除在堆上分配了内存的变量，而只是使引用为空，剩下的事就由垃圾回收器来完成。一旦垃圾回收器确定对象不再能通过存储在其他位置的引用来访问，就会删除该对象。这对于 Java 和 Visual Basic 开发人员来说很熟悉，但 C++开发人员会觉得这是多余的代码。第 5 章在介绍对象的生命周期时将详细讨论。

初始化变量

创建了任何类型的变量后，在访问该变量之前，都必须给它初始化一个值。如果没有给局部变量指定该值，就试图在计算表达式中使用此变量，C#编译器就会生成一个错误。这与 C++和 Visual Basic 完全不同。在 C++中，程序员负责解决这个问题，而在 Visual Basic 中，局部变量会自动初始化为相应的默认值。

C#编译器自动初始化变量的情况只有一种：初始化在类或结构中声明为字段的那些变量。这个简单的编译期间检查可以节省许多调试时间，因为它可以防止用户无意中从其他程序遗留下的内存中检索无用的值。

所有的实体都是对象

记住，.NET 环境中的任何类型，无论是值类型还是引用类型，都是对象。也就是说，每个类型都是从 `System.Object` 中显式继承、隐式继承，或间接继承的。这与 Java 的情形类似，在 Java 中，有一个根类结构层次，每个类都以某种方式从 `java.lang.Object` 继承。但在 .NET 中，这还应用于非引用类型。本书将展示这种继承方式的一些结果。

1.2 值类型

.NET 值类型有 3 种。本节将深入讨论每一种值类型：

- 基本类型

所有编程语言都定义了基本类型，如整型、浮点型等等。在 .NET 中，这些类型是值类型。稍后将讨论 C# 中的基本类型，并介绍这些类型如何映射为 Microsoft 中间语言 (MSIL) 的数据类型。

- 用户定义的值类型

可以定义自己的值类型，在应用程序中表示简单的对象或小段数据。在 C# 中，结构就是用户定义的值类型——使用 `struct` 关键字定义。.NET Framework 以类似方式定义了定制的值类型，如 `System.DateTime` 和 `System.Drawing.Point`。

- 枚举类型

枚举类型是一种特殊的值类型，它表示一个拥有一小列允许值的类型。例如，枚举类型可以指定值 `Yes`、`No` 和 `Maybe`。这些值一般都由一个整数表示，但是定义枚举类型允许为特定的一组整数值指定含义。

与 Java 不同，C++ 和 Visual Basic 都支持枚举。C# 中的一个较大区别是，.NET 环境中的枚举都是强类型的。例如，`HairColor` 枚举的允许值是 `Blonde`、`Red`、`Brown`，和 `Black`，而 `EyeColor` 枚举的允许值是 `Blue`、`Green` 和 `Brown`。这样，不但可以编写出可读的代码，同时确保不会无意中把某人描绘成蓝色的头发和红色的眼睛。

1.2.1 基本类型

C# 定义了 15 种基本类型来表示整数、浮点数、布尔值，和字符。其中的 11 种基本类型由 CLS 定义，确保与其他 CLS 兼容的编程语言互操作。剩下的 4 种基本类型与 CLS 不兼容，只能用于 C# 应用程序的私有部分，或不需要与其他语言互操作的公共接口。

每个基本类型实际上都是 .NET Framework 的 `System` 命名空间中标准类型的同义词。自己定义的值类型和这些基本类型之间没有实质上的区别。不过，这些类型确实

受益于 C#语言的一些特殊支持。

- 字面量语法：可以使用字面量语法创建基本值。例如，编写代码 `float pi = 3.142f` 时，可以使用字面量指定浮点值。使用 `3.142d` 表示双精度值，或使用某个后缀表示其他数字类型。其他基本类型也有类似的记号，例如 `true` 和 `false` 表示布尔字面量。
- 运算符支持：可以使用特殊运算符组合基本类型。例如，可以使用加法运算符 (+) 把两个数值加在一起，或使用 “&” 或 “|” 运算符组合布尔值。在 C# 中，还可以为自己的类型定义运算符，详见第 4 章。

表 1-1 显示了 C# 基本类型和 System 命名空间中等价结构之间的映射关系。该表还显示了 C# 编译器在编译时如何把这些类型翻译成 Microsoft 中间语言 (MSIL) 数据类型。与 CLS 不兼容的类型用星号标记。

表 1-1

基 本 类 型	等价的 .NET 结构	等价的 MSIL 数据类型	说 明
bool	System.Boolean	bool	True/False 值
byte	System.Byte	unsigned int8	无符号的 8 位整数
char	System.Char	char	16 位 Unicode 字符
decimal	System.Decimal	System.Decimal	128 位十进制值
double	System.Double	float64	IEEE 64 位浮点数
float	System.Single	float32	IEEE 32 位浮点数
int	System.Int32	int32	32 位带符号的整数
long	System.Int64	int64	64 位带符号的整数
object	System.object	object	所有类型的基类型
sbyte*	System.SByte	int8	8 位带符号的整数
short	System.Int16	int16	16 位带符号的整数
string	System.String	string	Unicode 字符串
uint*	System.UInt32	unsigned int32	32 位无符号的整数
ulong*	System.UInt64	unsigned int64	64 位无符号的整数
ushort*	System.UInt16	unsigned int16	16 位无符号的整数

注意 C# 把 `string` 和 `object` 都列为基本类型，尽管它们都是引用类型，不是值类型。`object` 是整个 .NET 类层次结构的根，第 2 章将进一步讨论 `System.Object`。大多数编程语言都把一些形式的字符串作为基本类型，而 .NET Framework 采用了略微不同的方式。本节把这些基本类型当做值类型看待，稍后在讨论引用类型时再讨论字符串。

C 和 C++ 开发人员要注意，表 1-1 中对 C# 基本类型的描述与 .NET Framework 保持

一致。在 C# 中，int 总是 32 位。而在 C/C++ 中，int 的字节数取决于平台，但这常常被忽视。同样，在 C# 中，long 是 64 位，而在 C++ 中，long 表示“字节数大于或等于 int 类型的字节数的整数类型”。这些定义显然应用于所有的 .NET 语言，在多语言的环境中操作时，这将使出错的可能性降低。

Visual Basic 用户注意：

C# 中的数值类型包括带符号和不带符号两种版本，VB 不支持这个功能。在混合使用这些类型时要小心，特别是在比较数字时，就更要注意。另外，C# 不会自动转换表达式中的数值类型，所以何时舍入就是一个很重要的问题。例如，float f = 1 / 3 返回 0，而 float f = 1.0f / 3.0f 返回 0.33333。

.NET 中的所有类型都是对象，所以可以在字面量上调用方法。这似乎很奇怪，用户可能从来没有想过这么做，但如果编写下面的代码 `string s = 32.ToString();`，编译并运行之，将有助于在我们脑子中巩固“所有实体都是对象”的概念。

下面的例子显示了一个简单的控制台应用程序，来说明基本类型在 C# 中的用法：

```
using System;

class MyClass
{
    static void Main()
    {
        int i = 100;    // use a primitive C# type
        Int32 j = i;    // use the equivalent .NET Framework type
    }
}
```

可以使用基本的 C# 数据类型(如 int)，它们可以与等价的 .NET Framework 类型(如 System.Int32)进行互换。为了简单起见，应该在代码中坚持使用一组声明。

注意：

Microsoft 强烈提倡使用任何 .NET Framework 语言组合的混合语言编程。如果开发一个多语言解决方案，可以显式地使用 .NET Framework 结构类型，来强调跨语言的通用性。例如，C# 中的 short，与 Visual Basic .NET 和 Managed Extensions for C++ 中的 short 相同，在所有语言中都等价的 .NET Framework 类型是 System.Int16。

```
Console.WriteLine("int:      {0}", typeof(int).FullName);
Console.WriteLine("Int32:    {0}", typeof(Int32).FullName);
```

使用 typeof 运算符在运行期间获得数据类型的信息，并把这个信息写到控制台上。下面的代码要求用户输入一个分子和一个分母，然后计算它们的商：


```
Console.Write("\nEnter a double: ");

string input = Console.ReadLine();
double num = Double.Parse(input);
Console.Write("Enter another double: ");
input = Console.ReadLine();
double denom = Double.Parse(input);

double res = num / denom;
if (Double.IsNaN(res))
    Console.WriteLine("Not a Number.");
else if (Double.IsPositiveInfinity(res))
    Console.WriteLine("Positive infinity.");
else if (Double.IsNegativeInfinity(res))
    Console.WriteLine("Negative infinity.");
else
    Console.WriteLine("Result is {0}.", res);
}
}
```

应用程序使用在 `double` 类型中定义的多种方法，来读取和处理 `double` 值。 `Parse` 方法从 `string` 中提取 `double` 值， `IsNaN()` 测试 “is not a number”， `IsPositiveInfinity()` 测试正无穷大(例如，100 除以 0)， `IsNegativeInfinity()` 测试负无穷大(例如，-100 除以 0)。

在应用程序运行时，会将 `int` 和 `Int32` 类型显示为 `System.Int32`，这样可以确保 C# 中的 `int` 类型只不过是 `System.Int32` 的另一个名称。应用程序也会要求我们输入两个浮点数。如果输入两个任意值，就可以在控制台上看到输出。

把代码保存到文件 `primitive_types.cs` 中，并编译它。在命令行提示中输入下面的内容：

```
C:\Class Design\Ch01> primitive_types
int:      System.Int32
Int32:    System.Int32

Enter a double: 432.33
Enter another double: 4576.33
Result is 0.0944708969851387.
```

检查编译器的输出

.NET Framework SDK 包含几个有用的工具，可以用来检查建立项目时生成的文件。最重要的一个工具是 `MSIL Disassembler`(MSIL 反汇编程序)，即 `ildasm.exe`。利用这个工具可以了解编译器如何把 C# 源代码翻译成 MSIL 字节代码。它还能查看类型的详细元数据，帮助理解公共语言运行时是如何工作的，进而更有效地使用 C#。第 8 章将详

细介绍元数据。

一些开发人员由于认为 MSIL Disassembler 无关紧要，因而拒绝接受。事实上并非如此。本书将广泛使用 MSIL Disassembler，来研究 C#编译器如何编译代码。

要运行 MSIL Disassembler 工具，先打开命令提示窗口(如果使用的是 Visual Studio .NET，要先启动 Visual Studio .NET 命令提示窗口)，然后进入包含可执行文件的文件夹中，运行 ildasm，如下所示：

```
> ildasm assembly-filename
```

可执行文件的名称和位置依赖于建立应用程序的方式：

- 如果使用 Visual Studio .NET 建立应用程序，可执行文件将与项目同名，但文件扩展名是.exe 或.dll，放在 bin\Debug 或 bin\Release 子文件夹中。另外，Visual Studio .NET 还添加一个命名空间，它的名称和项目名称相同。
- 如果使用命令行 C#编译器建立应用程序，可执行文件将与源文件同名，但文件扩展名是.exe 或.dll，放在和源文件相同的文件夹中。

例如，如果使用命令行编译器建立 primitive_types 应用程序，这个应用程序将出现在 MSIL Disassembler 窗口中。扩展 MyClass 图标，MSIL Disassembler 窗口就会显示如下信息，如图 1-4 所示。



图 1-4

双击 Main 图标，打开 Main()方法的 MSIL 代码视图，如图 1-5 所示。



图 1-5

在 MSIL 代码中, Main()方法由 MSIL 的 managed 关键字进行标记,表明代码运行在 .NET Framework 公共语言运行时提供的托管环境中。在 C#中编写的所有代码都是托管代码,其中的各种局部变量都用 MSIL 数据类型描述,如 float64、int32 和 string。

如图 1-6 所示是 Main()方法的 IL 代码的末尾。



图 1-6

每行 IL 代码都包含一个命令,以及命令需要操作的数据。程序处理的数据存储在栈中,使用开头为 ld 的 IL 加载命令,把项加载到栈中。栈中的每个变量都占用其类型定义的固定内存空间。对于引用类型对象,栈包含对托管堆中对象的实际存储位置的引用。

屏幕显示的第一行代码把一个字符串的引用加载到栈中。下一行把变量 V_5 的内容(包含除法操作的结果)加载到栈上。当一个项放置到栈上时,它将位于前面任何栈项的上方。从栈中删除项时,栈顶的项最先删除。

暂时忽略 box 命令,先解释一下 call 命令。该命令通知 .NET 去调用方法 WriteLine(),这个方法属于 mscorlib.dll 程序集中的类 System.Console,它把 string 和 object 作为参数。.NET 查询这个方法,然后从栈顶取两个值,传递给被调用的方法。栈顶的项是一个浮点值,该值是执行除法的结果。它不是 object 类型,而是值类型。

稍后深入探讨装箱,因为其中有一些重要的性能问题。现在只需知道 IL 代码中的 box 指令提取栈顶的项,把它复制到托管堆,把对装箱值的引用放在栈的顶部。这允许把该值视为 object,并传递给这个方法调用。因此调用方法时,栈顶的项是一个装箱值,然后是一个字符串,正是这两个值将传递给 Console.WriteLine 方法。

说明:

完成之后,关闭 MSIL Disassembler 窗口。如果忘记关闭 MSIL Disassembler 窗口,EXE 文件将会一直由 MSIL Disassembler 锁定。如果试图开着这些窗口重编译应用程序,会得到一个编译器错误,因为 EXE 文件无法被改写。

1.2.2 用户定义的值类型(结构)

应用程序经常要用类型来封装实质是数字，但无法用基本类型表示的数值型数量，如货币、屏幕坐标和温度。在这些情况下使用类很像是用锤子砸碎坚果。在运行期间对这些简单对象进行垃圾收集时的系统开销会很高。

.NET Framework 提供用户自定义的值类型来解决这个问题。在 C# 中，值类型编写成结构。与值类型一样，结构的实例存储在使用它们的任何地方。

值类型是杂乱的，因为它们到处留下自己的副本。不过，它们也很容易被清除。.NET 不需要跟踪值的每个副本——如果需要在其他地方使用值，只要把一个副本发送到那里即可。因此，如果值类型不再使用，它占用的内存会立即释放。但对于引用类型，就需要利用垃圾回收器来清除。引用的所有副本可能会超出作用域或被改写，但被引用的对象仍在托管堆上占用一定的内存。稍后介绍垃圾回收器如何处理这个问题。

从值实例传递的方式来看，值类型应该很小。如果定义大的值类型，那么在应用程序中的方法之间传递值实例时，效率会非常低，因为要复制到方法里和从方法复制出的数据量会很大。大的值类型会减慢对象的内存分配、管理，以及使用它们的对象和堆栈的清除。

本节将介绍如何在 C# 中有效地定义和使用值类型，以及如何对值类型使用继承。

1. 定义和使用值类型

定义值类型的规则实质上 and 定义类相同。例如，值类型可以有字段、属性、常量、事件、方法和构造函数。如第 3 章和第 4 章所述，还可以像类那样，重写方法和运算符，提供索引器。

值类型或结构是简化了的类。不过，它们还有一些重要的区别：

- 结构至少要声明一个字段或事件。
- 结构自动拥有一个默认的空参构造函数。如果试图定义自己的空参构造函数，编译器就会生成一个错误。这个构造函数用来执行所声明的字段的默认初始化。数值字段初始化为 0，布尔标记初始化为 false，而对象引用设置为 null。
- 不能定义自己的空参构造函数，但可以(并且应该)提供带有参数的构造函数。适当提供几个带有参数的构造函数，这样，值类型的用户就可以用各种有效的方式来初始化他们的对象。
- 不能在结构中提供字段的初始化程序，必须在构造函数中执行初始化(这和类不同，类可以在声明时初始化字段)。不过，可以提供 const 字段的初始化程序，也就是说，可以在定义时初始化 const 字段。下一章介绍 const 时，会详细讲解其原因。
- 结构对象有一个比类对象简单得多的取消内存分配的机制。垃圾回收器清除类

对象，然后在对象消失之前调用对象的析构函数。结构对象在超出作用域时取消分配给它的内存，或用另一个值来改写，而不是由垃圾回收器回收。因此，不允许自定义析构函数，但可以用与类相同的方式定义一个 `Dispose` 方法。只是由于不能读类那样保证会调用该方法，最好不要建立需要清除其资源的值类型。稍后将介绍一个例子，但对象的清除要在第 5 章才介绍。

- 许多例子都使用了带 `public` 字段的结构。这似乎同面向对象开发的基本规则“不要把数据声明为公有”相矛盾。这实际上是速度与封装的折衷。`public` 数据能进行更快的访问，因为它避免了调用方法获取数据的开销，但破坏了结构的封装。如果对此有怀疑，就把所有的字段都声明为 `private` 试试。如果私有字段展示为属性，而访问程序代码通常是由编译器逐行编译的，肯定会得到有效的执行。不要以牺牲可维护性作为代价来优化代码，让编译器进行优化比较好。
- .NET Framework 不允许继承结构。因此，在结构中定义的方法不能由子类中的方法重写。根据这个规则，不能把结构中的任何成员声明为 `virtual`、`abstract` 或 `sealed`。但是，编译器可以准确地预测在代码中使用结构对象时调用的是哪个方法。所以，编译器可以优化方法的调用，提高效率。例如，编译器可以选择在内部扩展方法体，而不是执行传统的方法调用。最终结果是结构对象的方法调用开销比类对象的更少。
- 结构作为值类型，总是在使用它们的地方分配内存。在定义一个包含结构的变量时，是否使用 `new` 运算符并没有什么区别。对于引用类型，如果声明变量时没有使用 `new` 运算符，运行时就不会给它分配托管堆上的存储空间，而是在栈上给它分配一个包含 `null` 值的引用。对于值类型，运行时会给它分配栈上的空间，并调用默认的构造函数，来初始化对象的状态。

下面的例子 `value_types.cs` 说明了这些规则：

```
using System;

struct Money
{
    // private instance field
    private int centsAmount;

    // private class field
    private const string currencySymbol = "$";

    // public constructor
    public Money(int dollars, int cents)
    {
```

```
        centsAmount = (dollars * 100) + cents;
    }

    // another public constructor
    public Money(double amount)
    {
        centsAmount = (int)((amount * 100.0) + 0.5);
    }
}

class MyClass
{
    static void Main()
    {
        Money freebie;
        Money salary = new Money(20000, 0);
        Money carPrice = new Money(34999.95);
    }
}
```

在本例中应注意以下内容：

- 结构中的字段声明为 `private`，来最大化封装。
- `currencySymbol` 变量在声明时初始化。可以这么做，是因为 `currencySymbol` 是一个 `const` 字段。
- 结构中有两个构造函数，用两种不同的方式初始化结构。注意在第二个构造函数中，从 `double` 强制转换为 `int` 时需要进行舍入。
- 类 `MyClass` 中的 `Main()` 方法创建了 3 个结构对象，说明如何调用可用的构造函数(包括编译器生成的无参数构造函数)。

此刻，这个程序还不能进行任何工作。下一节将为它添加更多的功能。

2. 对值类型使用继承

在 C# 中定义结构时，不能显式地指定基类。所有值类型都隐式地继承自 `System.ValueType`，这个类是 .NET Framework 库中的标准类型。`System.ValueType` 继承自 `System.Object`，并重写了 `System.Object` 的一些方法(`System.ValueType` 不提供任何额外的方法)。

在定义自己的值类型时，可以重写派生自 `System.ValueType` 或 `System.Object` 的一些方法。最常重写的一个方法是 `ToString()`，它返回对象的字符串表示。`System.Object` 的更多信息请参阅第 2 章。

结构不能用作其他类的基类；它们不能通过继承来扩展。结构是非常精密复杂的类

型，但它们不是类。这个语言的限制允许编译器最大程度地减少为支持结构而生成的管理代码。

尽管结构不能显式地继承自任何一个类，但它们能实现接口。例如，结构一般可以实现标准的.NET Framework 接口，如 `Comparable`。`Comparable` 允许指定对象应该如何进行比较，以便于排序。值类型经常需要实现这个接口，来和.NET Framework 中的其他类互操作。

本章的后面将详细探讨接口。下面的代码说明了如何修改 `Money` 值类型，以实现一个接口，重写从 `System.Object` 继承的 `ToString()` 方法：

```
// value_type_inheritance.cs
using System;

struct Money : Comparable
{
    // private fields
    private int centsAmount;
    private const string currencySymbol = "$";

    // public constructors
    public Money(int dollars, int cents)
    {
        centsAmount = (dollars * 100) + cents;
    }

    public Money(double amount)
    {
        centsAmount = (int)((amount * 100.0) + 0.5);
    }

    // compare with another Money
    public int CompareTo(object other)
    {
        Money m2 = (Money)other;
        if (centsAmount < m2.centsAmount)
            return -1;
        else if (centsAmount == m2.centsAmount)
            return 0;
        else
            return 1;
    }
}
```

```
    }

    // return value as a string
    public override string ToString()
    {
        return currencySymbol + (centsAmount / 100.0).ToString();
    }
}
```

Money 结构现在实现了 IComparable 接口。IComparable 接口指定的 CompareTo() 方法把这个 Money 实例的值和另一个 Money 实例进行比较，然后返回一个整数，说明比较的结果。

Money 还重写了 ToString() 方法，这个方法在基类 System.Object 中定义。ToString() 返回这个 Money 实例的字符串表示。下面是这些修改的结果：

```
class MyClass
{
    static void Main()
    {
        // create an array of 5 items
        Money[] salaries = new Money[5];
        salaries[0] = new Money(9.50);
        salaries[1] = new Money(4.80);
        salaries[2] = new Money(8.70);
        salaries[3] = salaries[2];
        salaries[4] = new Money(6.30);

        // display unsorted array
        Console.WriteLine("Unsorted array:");
        foreach (Money salary in salaries)
        {
            Console.WriteLine("{0}", salary);
        }

        // sort the array
        Array.Sort(salaries);

        // display sorted array
        Console.WriteLine("Sorted array:");
        foreach (Money salary in salaries)
```

```
        {  
            Console.WriteLine("{0}", salary);  
        }  
    }  
}
```

在上面的例子中，Main()方法创建了一个 Money 实例的数组。给数组元素 3 赋予元素 2 的值，就表示它包含该值的副本。数组中的每个 Money 实例都用 Console.WriteLine 来显示，该方法隐式地调用了重写的 ToString()方法。Array.Sort()方法对数组进行排序。为此，数组元素必须实现 IComparable 接口。Array.Sort()对数组元素反复调用 CompareTo()方法，以特定的顺序对它们进行排序。最后，排过序的数组显示在控制台上。

应用程序显示如下输出：

```
C:\Class Design\Ch01> valuetype_inheritance
```

```
Unsorted array:
```

```
$9.5
```

```
$4.8
```

```
$8.7
```

```
$8.7
```

```
$6.3
```

```
Sorted array:
```

```
$4.8
```

```
$6.3
```

```
$8.7
```

```
$8.7
```

```
$9.5
```

1.2.3 枚举

枚举是.NET 值类型，表示有限的一组允许值的整数类型。它们还可以用于把位标记映射为一个整数类型，以便使用一个变量来表示选项的组合。枚举在许多编程语言中都有，在.NET 中它们也是面向对象的。也就是说，开发人员可以使用其他语言中没有的功能。

1. 枚举类型

要声明一个枚举类型，可使用 enum 关键字，并指定符号名称表示允许值。也可以

为枚举指定基本的整型数据类型(byte、short、int 或 long)，或者给每个名称赋予一个指定的数字。

下面的例子 enumerations.cs 声明了一个简单的枚举来表示一场比赛中的奖牌。

```
// enumerations.cs
using System;

enum Medal : short
{
    Gold,
    Silver,
    Bronze
}
```

枚举隐式地继承了 System.Enum，也继承了它的所有成员。定义了一个枚举类型后，就可以在代码中使用它，如下所示：

```
class MyClass
{
    static void Main()
    {
        Medal myMedal = Medal.Bronze;
        Console.WriteLine("My medal: " + myMedal.ToString());
    }
}
```

这里创建了一个枚举实例 myMedal，并给它赋值 Medal.Bronze。枚举成员的作用域限制为枚举声明体。

如果编译并运行这段代码，则写到控制台上的输出就包含符号名称 Bronze，而不是对应的数字。这是 .NET 枚举优越于传统 C++ 或 VB 的一个地方。Visual Studio .NET IDE 在开发和调试过程中也利用了这个优点，给开发人员显示更有意义的图形。

除了普遍使用的 ToString 方法之外，还可以使用在 System.Enum 中定义的静态方法 Format()，把数值转换为符号名称，而不需要创建枚举类型的实例。例如，下面的语句会生成与上一个例子相同的输出：

```
Console.WriteLine("My medal: " + Enum.Format(typeof(Medal), 2, "G"));
```

System.Enum 实现了 IFormattable 接口，所以可以把格式选项和其他 .NET 类组合在一起使用，访问枚举类型的详细信息。例如，可以创建一个数组，其中每个元素对应一个符号名称，把数值和符号名称一起显示出来：

```
Medal[] medals = (Medal[])Enum.GetValues(typeof(Medal));
foreach (Medal m in medals)
{
    Console.WriteLine("{0:D}\t{1:G}", m, m);
}
```

把前面例子中的 Main()方法体用上述代码替换，会得到下面的输出：

```
C:\Class Design\Ch01> enumerations2
0      Gold
1      Silver
2      Bronze
```

如果需要以另一种方式来转换，例如让用户在运行期间输入名称，以这些名称初始化枚举类型，就可以使用 Parse()方法。用下面的代码替换 Main()方法：

```
Console.Write("\nEnter a medal: ");
string input = Console.ReadLine();
Medal myMedal = (Medal)Enum.Parse(typeof(Medal), input, true);
Console.WriteLine("You entered: " + myMedal.ToString());
```

Parse()方法中的布尔参数控制大小写的区分。在本例中，不检查大小写。用户输入一个匹配的名称后，就给用户输出响应：

```
C:\Class Design\Ch01> enumerations3
Enter a medal: gold
You entered: Gold
```

注意，输入一个与所定义的符号名称不匹配的字符串，会生成一个 ArgumentException，在运行期间使用 try...catch 块可以捕获该异常。也可以使用 System.Enum 中的 IsDefined()方法，在继续之前验证用户的输入。

2. 位标记

许多开发人员都很熟悉位标记。实际上，位标记是指，首先定义一个数值类型，例如包含 8 位的字节，再使用位掩码(bit-mask)(另一个二进制数字)和一些逻辑运算符来设置、取消设置，或查询变量中的各个位。这样，一个字节中可以有至多 8 个开关设置，这是存储空间的一种紧凑用法。

在 C#中，可以定义一个枚举类型来完成类似的工作，而且更容易读取。为此，使用 [Flags] 属性来定义枚举，给每个符号名称指定位的特定组合。例如，下面的代码就定义了一个字节的枚举类型，其中包含 4 个不同的选项：

```
using System;

[Flags]
enum Permit : byte
{
    Create = 0x01,
    Read    = 0x02,
    Update = 0x04,
    Delete = 0x08
}
```

C#编译器允许使用[Flags]或[FlagsAttribute]，它们是同义词。下面的代码说明了如何使用这种枚举类型：

```
class TestBitFlag
{
    static void Main()
    {
        Permit perm1 = Permit.Create;
        Console.WriteLine(perm1.ToString());
        Permit perm = Permit.Create | Permit.Read;
        Console.WriteLine(perm.ToString());
        perm |= Permit.Delete;
        Console.WriteLine(perm.ToString());
    }
}
```

注意使用按位 OR 运算符(|)可以设置变量中的各个位。也可以使用 ToString()方法生成变量的可读表示。编译并运行这段代码，生成的输出如下：

```
C:\Class Design\Ch01> bitflags
Create
Create, Read
Create, Read, Delete
```

一旦用[Flags]属性标识了枚举类型，System.Enum 类型就会产生不同的输出。与前面一样，可以使用 Parse()方法，但这次要使用一组用逗号分隔开的选项，利用用户的输入初始化变量。

所定义的符号名称不必映射各个位。而可以引入一组表示位组合的值，和 System.Enum 所提供的功能保持一致。例如，重新定义枚举类型，如下所示：


```
[Flags]
enum Permit : byte
{
    Create = 0x01,
    Read   = 0x02,
    Update = 0x04,
    Delete = 0x08,
    AllExceptUpdate = 0x0B
}
```

现在编译并运行这个例子，得到如下输出：

```
C:\Class Design\Ch01> bitflags2
Create
Create, Read
AllExceptUpdate
```

[Flags]属性影响了 `System.Enum` 中的 `ToString()`、`Parse()`和 `Format()`方法的工作方式，但结果仍是有意义的。如果枚举类型中的位设置组合正好匹配某个符号名称，就输出该名称；否则就把符号名称的组合输出到用逗号分隔开的列表中。但这里要小心，因为可以用同一个值指定多个符号名称。此时，`System.Enum` 在转换后的输出中将使用这些名称中的一个，但所选的名称是不确定的。

3. 在枚举类型的内部

下面是两个在代码中使用枚举的理由：

- 枚举使用有意义的名称表示允许值。这些名称更易于其他开发人员理解这些值的用途。
- 枚举是强类型的。不能把整型值赋给枚举实例，除非使用显式的类型转换，否则就必须使用枚举成员名称。这将有助于避免因使用原来的整型值而出现的编程错误。

现在，C++ 和 Visual Basic 开发人员已经很熟悉枚举了，但这些语言中的对应类型并不是强类型的。如果用户是 C#新手，就会发现枚举在调试过程中对开发人员是很友好的。原因是 .NET 枚举是值类型，直接从 `System.Enum` 派生。在源代码中看不到这种关系，但如果用 IL 反汇编程序查看一下 MSIL，就会很清楚地看到这种关系。例如，图 1-7 是本节开始时编写的第一个样例应用程序。

Java 没有枚举结构。Java 开发人员一般把类用作类型，把类的各种静态实例用作变量。在上面的 ILDasm 输出中，可以看出，C#中的枚举并不是什么新东西，只是创建专用值类型的一种简化方法，这种值类型提供了定义好的、有限的行为。但与其他值

类型不同的是，在枚举中不允许定义方法、属性或事件。



图 1-7

有时，枚举并不像自定义的值类型那样正好满足用户的需要。可以想像，通过这个类型，可以大大改进枚举提供的功能。可以给枚举添加方法，甚至可以用不同的枚举成员来实现不同的操作。

这里的要点是仔细考虑自己的设计，特别是编写较长的 switch 语句，而该语句基于枚举中的值时，就更要好好考虑。

.NET 枚举的最后一个问题是，可以把基本整数类型的任意值转换为枚举类型——不论它表示枚举值是否合法。

因此，最好不要假定传递给方法的枚举值肯定是允许值中的一个。一定要对枚举值执行有效性检查。

1.3 引用类型

下面是通用类型系统中的引用类型。本节将详细说明这些类型：

- 类类型

在一般的应用程序中定义的大部分类型都是类类型。类为应用程序中的重要对象指定了数据和方法。

- 委托

委托表示指向方法的类型安全的指针。可以用委托来指定回调方法，为图形用户界面(GUI)事件注册处理程序，如按钮单击。

- 数组

数组在托管堆上被分配内存，由引用访问。数组可以是一维的或多维的。也可以使用数组再包含数组，即允许锯齿状数组结构。

- 字符串

.NET Framework 对处理字符串内置了全面的支持。这种支持扩展到了 CLR，包括使编写出的应用程序正确运行在不同的运行时平台上，并使用不同的字符集。

1.3.1 类类型

在面向对象的术语中，类是用来定义系统中特定类型对象的方法和数据。面向对象社团中的许多开发人员在谈论类时会使用短语“抽象数据类型”。而在 .NET 中要小心，因为类只是一种抽象数据类型——这个术语也可应用于结构。另外，.NET 使用“抽象”这个词来表示完全不同的内容。

在 .NET Framework 中，类是引用类型，对象位于公共语言运行时的托管堆中，这些对象的生存期由运行时控制。本节主要讨论如何定义类，在讨论期间，记住，在 .NET Framework 中，类总是引用类型。

1. 用 C# 定义类

类使用 `class` 关键字来定义，这与 C++ 和 Java 相同。实际上，声明 C# 类的基本语法与 Java 完全相同。例如。下面的代码在 C# 和 Java 中都可以编译通过：

```
public class MyClass
{
    private static int refCount = 0;
    public int someNumber;

    public void DoSomething()
    {
    }
}
```

这与 C++ 的区别也不大，在 C++ 中，上述代码可以编写为：

```
class MyClass
{
public:
    int someNumber;
    void DoSomething()
    {
    }
}
```

```
private:
    static int refCount;
}

int MyClass::refCount = 0;
```

C++用户注意:

在 C# 中, 没有全局变量或全局函数。所有的对象都必须在类或结构中定义。另外, 如果没有为类提供默认的构造函数, C# 编译器就会生成一个, 把所有的成员字段都初始化为 0。

类可以包含数据和函数。表 1-2 说明了可以在类中定义的数据成员种类。

表 1-2

数据成员种类	说 明
字段	字段是存储在类的对象中的一块数据。也可以使用 static 关键字来定义由类的所有对象共享的数据
方法	方法是一个函数, 为类的对象提供某个方面的行为。也可以使用 static 关键字来定义应用于类本身而不是一个特定对象的方法
属性	对于使用类的用户来说, 属性和字段类似, 但是它的值必须通过类中定义的 get 和 set 方法获取和修改。属性所呈现的功能就好像它是数据
索引器	类型索引器是一种属性, 允许使用[]标记来访问类。它一般用于允许访问集合中的项
常量	常量是只读的值, 它由类的所有对象共享
事件	事件是一个消息, 对象可以把这个消息发送给感兴趣的侦听器对象。在 .NET Framework 中, 术语“事件源”表示可以引发事件的对象, 术语“事件接收者”表示可以接收事件的对象。事件定义为类型的一个成员, 标识该类型的所有对象都是事件的源
类型	类型可以包含嵌套的类型。这是惟一推荐使用的特殊功能, 详见后面的“嵌套类”
构造函数	构造函数是一个初始化方法。每次创建类的一个对象时, 都要调用构造函数来初始化对象的状态
析构函数	一个类可以有一个结束方法。CLR 在垃圾回收器删除对象之前调用这个方法。这个方法适合执行在对象消失之前的清理操作

C++用户注意:

与 C++ 一样, 析构函数用 ~ 符号指定。析构函数只能为引用类型声明, 且不能重

写。与 C++ 不同的是，析构函数不能显式调用，因为垃圾回收器管理着对象的生存期。对象层次结构中的每个析构函数都会在对象的内存恢复之前自动调用。

对于构造函数，有一点要注意：在 C# 中，可以定义两种类型的构造函数：实例构造函数和静态构造函数。前者我们很熟悉，后者又称为类型初始化程序或类型构造函数。静态构造函数类似于 Java 中的静态初始化块，它的声明与实例构造函数相同，只是要添加 `static` 关键字。类至多只能有一个无参的静态构造函数。静态构造函数定义后，会在类的静态方法的第一次调用之前和类的实例第一次创建之前调用。静态构造函数的讨论详见第 5 章。

下面的类定义说明了这些成员。不能在同一个类中定义所有的成员，这里列出它们只是为了说明有哪些类型的成员。后面的章节将讨论每种成员的设计。

```
// class_members.cs - compile with /t:library option
class MyClass
{
    // Field (usually private, for encapsulation)
    private int aField;

    // Property (usually public, for ease of use)
    public int AProperty
    {
        get { return aField; }
        set { aField = value; }
    }

    // Constant (class-wide, read-only field)
    private const int aConstant = 43;

    // Delegate (defines a method signature)
    public delegate void ADelegate(int aParameter);

    // Event (to alert event receiver objects)
    public event ADelegate AnEvent;

    // Method
    public void AMethod()
    {
        // Method implementation code
    }
}
```

```
// Instance constructor (to initialize new objects)
public MyClass(int aValue)
{
    aField = aValue;
}

// Destructor method (to tidy up unmanaged resources)
~MyClass()
{
    // Finalization code
}

// Nested class definition
private class aNestedClass
{
    // class definition
}
}
```

2. 定义类的可访问性

定义类型时，可以指定它的可访问性。可以使类型专用于它的声明上下文，或对同一个程序集中的其他代码可访问，或对任何程序集中的代码都可访问。

注意：

程序集是 .NET Framework 中的关键概念。最简单的情况是，程序集是一个文件 (.exe 或 .dll)。编写应用程序时，可以决定是否要在同一个程序集中定义所有的类(或其他类型)，或为部署考虑而把类放在不同的程序集中。关于程序集的更多信息，包括如何在程序集中组织类定义的讨论，请参阅第 8 章。

为类型定义可访问性，在大型系统中是一个重要的考虑事项，在这种系统中，类型的数量很容易就达到上百个。此时就要尽可能地限制可访问性。把类型相对于系统的其他部分隐藏起来，会使系统的耦合度更低，更易维护。系统的其他部分对这个类型的依赖性越低，类型就越容易修改。

下面的例子说明了如何使用 C# 指定类的可访问性。

```
class MyClass1
{
    // members
}
```


在上面的代码中没有出现访问修饰符，MyClass1 就隐式声明为私有，这就意味着同一程序集中的其他类以及其他程序集都不能访问它。另一方面，MyClass2 对所有程序集显式声明为公有：

```
public class MyClass2
{
    // members
}
```

默认的可访问性取决于定义它的上下文。通过声明可访问性修饰符来使可访问性显式，可以避免其他代码修改可访问性，出现意想不到的负效应。例如，把 MyClass1 放在命名空间的声明中，就会把默认的可访问性从 private 改为 internal。

程序集中类的公有和私有的可访问性类似于 Java 的约定，但默认的可访问性是 private，而不是 package。不过，C#中有几个访问修饰符可以用于在类中定义的成员，包括嵌套类。

有 3 个访问修饰符可以用于控制类(嵌套类或其他类)或其成员的可访问性，如表 1-3 所示。

表 1-3

访问修饰符	说 明
internal	成员或类只能在同一个程序集中访问
protected	成员或类只能在其父类或从该父类派生出来的其他类型中访问
protected internal	成员或类可以在同一个程序集的任何地方访问，也可以从其父类或从该父类派生出来的其他类型中访问

可访问性详见第 2 章。

3. 嵌套类

在类中声明的所有内容实际上都嵌套在它的内部。在类中可以嵌套任何类型，包括其他类。例如，可以声明只在某个类内部有用的数据结构，最简单的方法是使用嵌套类。嵌套类不但是一个自包含的类(这一点与其他类一样)，而且可以访问在其父类中定义的所有成员。

下面的小例子使用了上一节介绍的访问修饰符：

```
// class_visibility.cs
public class MyClass
{
```



```
class MyPrivateClass
{
    // members
}

internal class MyInternalClass
{
    // members
}

protected class MyProtectedClass
{
    // members
}

protected internal class MyProtIntClass
{
    // members
}
}
```

把这个例子编译进 DLL，在 MSIL 反汇编程序中查看一下 MSIL 代码，如图 1-8 所示。



图 1-8

在图 1-8 中，注意以下几点：

- MyClass 用 MSIL 关键字 public 限定
- MyInternalClass 用 MSIL 关键字 assembly 限定
- MyPrivateClass 用 MSIL 关键字 private 限定

- MyProtectedClass 用 MSIL 关键字 family 限定
- MyProtIntClass 用 MSIL 关键字 famorassem 限定

protected 访问修饰符的使用在第 7 章“继承和多态性”讲述。C++和 Java 开发人员应很熟悉 public、private 和 protected 访问修饰符,但这些术语的含义最初可能容易混淆;internal 则是一个新的访问修饰符。

Java 用户注意:

public 和 private 的含义与 Java 中的一样。internal 的访问范围是程序集,而不是命名空间(这比较类似于 Java)。internal protected 与 Java 中的 protected 访问修饰符相同,C#中的 protected 访问修饰符等价于 Java 的 private protected 访问修饰符,该修饰符在 Java 中已废弃。

Java 开发人员很熟悉两种类型的嵌套类:内部类和静态的嵌套类。在 C#中,嵌套类类似于 Java 的静态嵌套类。C#中没有直接对应于 Java 内部类的内容,但只要创建一个嵌套类,使之包含对其父类的一个引用,再加上一个传送给其构造函数的实例,就可以得到相同的效果。其优点是嵌套类和其父实例之间的关系在代码中很明显。Java 开发人员还应注意,C#中没有与匿名内部类的对应类,C#类不能定义在方法体中。

MSDN 中提供了一些有用的规则,可以帮助决定何时使用嵌套类。这些规则包括:

- 如果类在逻辑上包含在另一个类中,且没有独立的意义,就可以把它实现为一个嵌套类。
- 如果类的成员需要访问包含它的对象的私有成员字段,就可以把它实现为一个嵌套类。
- 如果类需要实现一个接口,返回另一个类型的对象,来实现另一个接口(例如 IEnumerable.GetEnumerator()),就可以把该接口的实现代码创建为嵌套类,通过需要的方法返回对象。本章的最后将详细介绍接口。
- 如果其他类需要访问,或需要收集或包含类,就不要把它实现为一个嵌套类。由其他几个类访问的类应是独立的。
- 一般情况下,不要实现公共的嵌套类。嵌套类应主要在包含类的内部使用。如果实现了一个公共的嵌套类,该类在逻辑上就应属于包含类型,而且使用很频繁。如果嵌套类的一个实例展示为包含类的一个属性,就会出现异常。

4. 类对象的创建和处理

与其他面向对象的语言一样,可以在类中编写构造函数,在创建对象时初始化对象。如果在类中不定义任何构造函数,C#编译器会生成一个无参构造函数。有关静态构造函数的知识详见第 5 章。

公共语言运行时跟踪托管堆中的对象,当应用程序不再需要它们时,就把它们标记

为删除。垃圾回收机制可防止内存泄露，使我们更容易管理应用程序中复杂的对象关系。

在 C# 中可以编写一个析构方法，在对象被释放之前清理这些对象。析构函数类似 C++ 中的析构函数，或 Visual Basic 6 中的类终止方法。不过，与 Java 中的结束方法一样，不能控制这个事件的发生时间。由于垃圾回收器将执行结束代码，因此析构函数还可能导致重大的运行期间系统开销。垃圾收集过程必须维护所有需要结束的无效对象列表，并确保这些对象在适当的时候删除。由于对象释放的不确定性，我们无法确定结束代码什么时候运行。

1.3.2 委托

在 .NET Framework 中，委托是通用类型系统的重要部分。委托很像一个类型安全的指针，指向类中的方法。我们定义一个委托，来指定希望通过委托调用的方法签名，因为委托本身没有实现代码。然后创建一个委托对象，把它绑定到其签名和这个委托相匹配的方法上。

C++ 开发人员可以把委托看作是类型安全的函数指针。Java 开发人员可以使用接口来实现类似的效果。在 C# 中，使用委托可以使函数指针方法变成类型安全的(并包含多个方法)，而利用委托，调用方法时不需要内部类，这样就改进了接口。

在下面几种情况中委托很有用：

- 用另一个对象把一个方法注册为回调方法时。当对象发生重要事情时，它可以调用回调方法。这种用法的一个例子是，在图形用户界面(GUI)控件，如按钮和文本字段上，为事件定义回调方法。
- 当从一系列方法中选择一个方法(有相同的签名)用于一个算法时。例如，让方法计算角的正弦、余弦和正切值，可以使用委托来指定该算法使用这些方法中的哪几个。

定义和使用委托有 3 个必需的步骤：

- 声明委托
- 创建委托对象，把它绑定到特定方法上
- 使用委托对象调用这个方法

第 6 章将详细研究委托。

1.3.3 数组

如前所述，与其他许多编程语言不同，.NET 中的数组是引用类型。它们不仅仅是

包含另一种类型的多个副本的内存块，而且是由 CLR 在托管堆上分配内存的对象。所有的数组都有隐式的基类 `System.Array`，它提供了一些有用的功能。下面看看在 C# 中如何声明数组。

1. 数组的声明

C# 允许声明一维数组(有时称为矢量)、多维数组和锯齿状数组。锯齿状数组就是包含数组的数组，每个数组都有不同的维数。与 C++ 和 Java 一样，C# 中的数组用 `[]` 语法声明：

```
// one-dimensional array
int[] oneDec = new int[10];

// two-dimensional rectangular array
int[,] twoDec = new int[10, 5];

// jagged array
int[][] jaggy = new int[3][];
jaggy[0] = new int[5];
jaggy[1] = new int[10];
jaggy[2] = new int[20];
```

注意，与 C++ 不同，`[]` 的后面必须是类型名，而不是变量名。这与 Java 中最常用的语法相同。还要注意，数组必须使用 `new` 运算符初始化，因为它是引用类型。

C++ 用户注意：

与 C++ 中基于 C 的数组相比，C# 中的数组在可预测和有用的数据类型方面都要好得多。它的大小和类型在运行期间可以确定，如果数组是多维的，其维数也是可以确定的。NET 数组类型由元素类型和维数来定义，而不是由数组的大小来确定。因此，方法的参数不能指定为两个整型数组，只能指定为一个整型数组。

2. 数组元素的初始化

在创建数组时，所有的元素都自动初始化为它们的默认值。另外，对于所有的数组(锯齿状数组除外)，都可以使用数组初始化程序来加载预设置的数据。此时，不需要指定数组的维数，因为它可以在编译期间确定。

```
int[] oneDec = new int[] {4, 4, 8, 2, 5, 9};
int[,] twoDec = new int[,] {{1, 3}, {9, 4}, {4, 4}, {8, 2}, {5, 9}};
```

C# 还允许使用略微复杂的语法，其效果相同：

```
int[] oneDee = {4, 4, 8, 2, 5, 9};
int[,] twoDee = {{1, 3}, {9, 4}, {4, 4}, {8, 2}, {5, 9}};
```

其中，每个被花括号括起来的项分别与索引为[x,0]和[x,1]的项相关，其中 x 的范围是 0~4。

锯齿状数组中的每个元素本身都是一个数组引用，所以隐式初始化为 null。这就是不能为锯齿状数组指定数组初始化程序，但必须先创建每个子数组的原因：

```
int[][] jaggy = new int[3][];
jaggy[0] = new int[] {2, 8, 8, 9, 4};
jaggy[1] = new int[] {2, 3, 0, 7, 9, 6};
jaggy[2] = new int[] {1, 8, 1, 1, 1, 9, 5, 9};
```

注意，在这种情况下，不能使用简化的初始化语法，该语法只能在声明中使用。

3. 数组的使用

各个数组元素可以使用[]标记来访问，其方法与数组的声明相似。例如，可以用如下方式访问已初始化数组中的元素：

```
int numa = oneDee[2];    // numa now contains 2
int numb = twoDee[0, 1]; // numb now contains 3
int numc = jaggy[1][3];  // numc now contains 7
```

所有的开发人员都要注意，C#中的数组在默认情况下都是基于 0 的。Visual Basic 开发人员应注意，在 C#中要使用方括号来访问数组元素，而不是圆括号。

数组继承了 System.Array 中定义的所有功能，所以可以访问一些有用的属性和方法，来查询和操作数组及其元素。对于其中的一些方法，元素类型必须支持特定的接口。本章的最后将讨论这些接口。表 1-4 是 System.Array 中可以使用的几个公共成员。

表 1-4

成 员	说 明
Rank	这是一个只读属性，返回数组的维数
GetLength()	返回指定维中的元素个数
Sort()	给一个数组、两个数组中的元素，或一个数组中的一部分中的元素排序。注意数组元素类型必须实现 IComparer 或 IComparable 接口
BinarySearch()	使用二进制搜索算法搜索已排序数组中的一个元素。与 Sort() 一样，数组元素类型必须实现 IComparer 或 IComparable 接口
IndexOf()	返回一维数组中第一个匹配元素的索引

除了这些成员之外，在代码中访问数组元素时，CLR 还确保数组的索引在运行期间是有效的。也就是说，不能无意或有意地超越数组的边界，访问超出允许范围的内存。这不仅对跟踪错误很有效，还可以堵上潜在的安全漏洞。

4. 数组的类型转换

C#和.NET 运行时允许对数组转换其引用类型，但有几个限制。要进行类型转换，数组类型必须有相同的维数，元素类型之间的隐式或显式转换也必须可行。CLR 不允许隐式转换包含值类型的数组，但 `System.Array` 定义了一个 `Copy()` 方法，它可以实现这一点。

把数组转换为另一种类型类似于任意两个类型之间的转换。如果两个引用类型之间存在有效的转换，则这些引用类型的数组也应可以进行相同的转换。这称为数组的协变性，因此，CLR 可以在运行期间检查是否给某个引用数组的元素赋予了有效的类型。例如，尽管把字符串数组转换为对象数组是合法的，但下面例子中的第三行代码将抛出一个异常：

```
string[] someStrings = new string[4];    // ok
object[] someObjects = someStrings;      // ok
someObjects[2] = 42;                     // throws exception!
```

如果考虑转换值类型的数组，这会更有效。假定有一个包含屏幕坐标的 `int` 数组，现在要执行某个需要浮点数精度的算法，可以使用 `Copy()` 方法快速而有效地在整个数组中执行转换：

```
int[] points = {13, 145, 87, 209};
double[] work = new double[points.Length];
Array.Copy(points, work, points.Length);
```

下一节将介绍更多的数组处理的例子。

1.3.4 字符串

标准.NET 引用类型的最后一节探讨字符串。.NET 中对字符串处理的支持与 CLR 紧密集成在一起，是非常全面的。于是，C#把字符串看作是基本类型，但它们实际上是引用类型，由 `System.String` 类定义。本节将介绍如何声明和使用字符串，并讨论一些功能。更多的内容可参阅清华大学出版社引进并出版的《C#字符串和正则表达式参考手册》。

1. 字符串的声明

字符串被看作基本类型，因此可以用许多方式显示它。例如，可以使用字面量值来初始化一个字符串：

```
string s = "Hello World";           // Ok
string s = new String("Hello world"); // Error!
```

可以使用构造函数来初始化字符串，但只能通过 `char` 或 `char` 数组中传送(如果编写不安全的、与 CLS 不兼容的代码，就可以在指向字节数组的内存指针中传送)。C# 把 `string` 关键字定义为其基本状态，这常常是使 C# 新手出现混淆的一个原因。实际上，`string` 和 `System.String` 是同义词，可以在代码中互换，但最好一直使用一个标准。

C++ 开发人员可以放心的一点是，C# 中指定提供 `@` 的字符串语法有时称为逐字字符串(VerbatimString)。在字面量字符串值的前面加上 `@` 前缀，可以确保不处理字符串中的任意转义序列，因此不需要编写两次转义字符。下面两条语句声明了相同的字符串，看看自己是否弄明白了：

```
string s1 = "C:\\Winnt\\System32\\Write.exe";
string s2 = @"C:\Winnt\System32\Write.exe";
```

C# 允许使用 C 和 C++ 开发人员非常熟悉的转义序列。例如，可以使用 `\t` 表示制表符，用 `\n` 表示换行，用 `\r` 表示回车。但是，定义换行字符有一种更好的方式：因为 `System.Environment` 类定义了一个独立于平台的静态属性 `NewLine`。在 Windows 下，这个属性包含 `"\r\n"`，由其他 .NET Framework 类使用，例如 `Console.WriteLine()` 方法。

2. 字符串的不可变性

在 .NET 中，最重要的问题之一是，字符串是不可变的。这与 Java 中的 `java.lang.String` 类相同，即字符串一旦初始化，就不能改变。它不能缩短、增大，或改变其中的字符。

只要在字符串上执行操作，就会返回一个新的实例，原来的字符串则保持不变。如果编写下面的语句，在字符串的一个临时副本上进行测试，该临时副本会由垃圾回收器删除：

```
string s = @"C:\Winnt\System32\Write.exe";
if (s.ToLower().EndsWith("dll"))
{
    // Do something
}
```

可变性在第 2 章进行了详细介绍。

3. 字符串的使用

最后用一个包含数组的例子来结束字符串的简要讨论。首先使用数组初始化程序定义一个字符串数组：

```
using System;

public class StringClass
{
    public static void Main()
    {
        // Initialize at declaration
        string[] chapters = {"Managed heap", "Array handling",
                            "Reference types", "Year 2000"};
```

字符串和数组都是引用类型，所以这里定义的是一个栈变量 `chapters`，它包含对托管堆上一个数组的引用。该数组包含对托管堆上其他地方的字符串的引用。如果试图改变这些数组元素，原来的元素就会被指向新字符串实例的引用替代。之后，定义第二个字符串数组，但这次在代码中初始化各个元素：

```
        // Initialize by element
        string[] sections = new string[8];
        sections[0] = "Value types";
        sections[1] = "Language features";
        sections[2] = "Implementing interfaces";
```

使用 `+` 运算符可以连接字符串：

```
        // Concatenate strings
        sections[3] = "User-defined " + sections[0];
```

这仍不会改变它操作的字符串对象，而是创建一个全新的字符串对象，它是对这个对象的引用，该对象位于 `sections` 的第三个元素中。现在使用在 `System.Array` 中定义的 `CopyTo()` 方法，在两个数组之间复制字符串。这个过程执行得很快，因为字符串是引用类型，只有引用需要被复制。CLR 不需要复制字符串本身。因为字符串是不可变的，如果对原来数组中的字符串执行了操作，也没有关系，因为字符串是不会改变的。其结果是：即使字符串是引用类型，它的行为方式也与值类型相同。这种行为方式非常有用，也是引用类型的代码永不可变的原因，详见下一章。

```
        // Copy elements between arrays
        chapters.CopyTo(sections, 4);
```

使用 `System.Array.Sort()` 方法，可以对整个数组排序，也可以像本例中那样，仅对一部分元素排序。`String` 类实现了 `IComparable` 接口，该接口是用 `System.Array` 排序所必不可少的。实际上，`String` 类中有一系列比较方法，它们可以满足区分大小写的要求，并利用调用线程的 `CurrentCulture`，来确保不同语言中字符的正确比较。例如，在德语中，字符串 `'Strasse'` 和 `'Straße'` 是等价的。

```
// Partial string sort
Array.Sort(sections, 0, 4);
```

最后，`System.Array` 实现了 `IEnumerable` 接口，所以可以使用 C# 的 `foreach` 关键字遍历数组中的元素，把它们显示在控制台上：

```
// Enumerate array
foreach (string s in sections)
{
    Console.WriteLine(s);
}
}
```

把这个文件保存为 `string_array.cs`，并编译它。应用程序会生成如下输出：

```
C:\> string_array
Implementing interfaces
Language features
User-defined Value types
Value types
Managed heap
Array handling
Reference types
Year 2000
```

1.4 把值类型用作引用类型

有时希望在需要引用类型的地方使用值类型。例如把值类型的实例传送给一个需要对象参数的方法，比如说把一个值实例插入到 .NET Framework 集合中。

装箱和取消装箱

把值类型的实例转换为引用类型的实例，就称为装箱，其逆过程称为取消装箱。这对于许多转而使用.NET的大多数开发人员来说，都是一个新概念。在C#中，这种情况常常在我们未意识到的情况下发生。但是，理解发生了什么是很重要的，所以下面用一些篇幅来详细介绍这两个过程。

1. 把值类型转换为对象

考虑下面的代码段：

```
// Create an ArrayList object
System.Collections.ArrayList payroll =
    new System.Collections.ArrayList();

// Add some Money objects
payroll.Add(new Money(30500.0));
payroll.Add(new Money(54000.0));
payroll.Add(new Money(27900.0));
```

ArrayList 是一个标准的.NET Framework 集合类，与其他集合类一样，ArrayList 包含一个对象的引用的集合。这些对象必须是引用类型，所以必须位于托管堆上。ArrayList 类的 Add()方法需要一个对象作为参数。

如果要把一个值类型的实例(Money 结构，在本章前面的“值类型的定义和使用”一节中定义)添加到 ArrayList 中，CLR 就会在托管堆上自动创建 Money 实例的一个装箱副本。此时，装箱表示 CLR 给值实例添加了一个包装器，使它的显示和行为类似于一个引用类型。这在 C#中是自动进行的。

装箱的对象包含值对象中的数据副本。如果装箱的对象修改了，不会影响原来的值对象。同样，如果原来的值对象修改了，也不会影响装箱的对象。这正是值类型所期望的。

下面的代码从 ArrayList 中检索一个元素：

```
Money myCash = (Money)payroll[0];
```

在 C#中，ArrayList 类包含一个 Item 属性，它用作该类的索引器。这允许使用[]语法返回托管堆上对象的引用。在使用显式类型转换把它赋予 Money 实例时，CLR 就会从装箱的对象中提取该值，把它复制到本地的 Money 实例中。这样就有了该值的一个副本，修改它不会影响原来的值和堆上的装箱值。这就是取消装箱的过程，在 C#中它也是自动进行的。但是，由于 C#的类型安全性，在取消装箱时必须使用显式的类型转

换。如果 ArrayList 中的对象类型不正确，运行期间就会抛出一个异常。

注意：

前面使用 ArrayList 来介绍装箱过程，但要注意，当值类型在使用 [] 语法定义的一般 C# 数组中声明时，是不会发生装箱过程的。在这种情况下，应在托管堆上分配内存，来包含值类型数组的元素，但这些元素并未装箱。

C# 自动执行装箱过程，在取消装箱时，除了提供显式的类型转换之外，不需要编写任何代码来完成该过程。这是非常好的，因为装箱和取消装箱在运行期间会增加系统开销。

2. 性能问题

为了理解这种系统开销对应用程序的影响，以及可能发生的情况，下面深入探讨一下装箱和取消装箱过程，因为理解这些问题有助于确定应把某个类型实现为结构还是类。下面首先看看在上述代码中，当把 Money 值类型添加到 ArrayList 中时发生的情况：

(1) 首先利用 new 运算符，创建一个基于栈的 Money 类型。调用相关的构造函数初始化该对象的字段。

(2) C# 编译器认为，ArrayList 类的 Add() 方法需要一个引用类型，而不是值类型，所以调用 MSIL 代码执行装箱操作。

(3) 在托管堆上分配内存，这包括 Money 结构中的字段需要的内存，以及把装箱的类型建立为引用类型所需的额外内存。例如，要为一个方法表指针分配内存，而这与值类型并不相关。

(4) 基于栈的 Money 实例是利用 new 运算符创建的，其上的字段要逐个字节地复制到新分配的堆内存上。

(5) 最后，返回堆上保存的对象的地址，并传送给 ArrayList 类的 Add() 方法。

这个过程要重复 3 次，即为添加到 ArrayList 中的每个 Money 实例重复一次。对于取消装箱，其过程正好相反：

(1) 首先，在栈上为 myCash 变量分配内存，再调用 Money 的默认构造函数来初始化成员字段。

(2) 接着，检查装箱的引用，看看它是否为 null。如果是，就抛出一个 NullReferenceException 异常。

(3) 然后，根据在源代码的显式类型转换过程中声明的类型，检查装箱对象的类型。在本例中，如果对象不是 Money 类型，就抛出一个 InvalidCastException 异常。

(4) 最后，从托管堆上获得装箱值类型的一个指针，把 Money 字段的内容逐个字节地复制到基于栈的变量 myCash 中。

可以看出，如果应用程序执行许多装箱和取消装箱过程，那么性能和内存使用方面的

开销就会很大。

3. 其他装箱的情况

上面的例子是把值类型作为参数传送给一个需要引用类型的方法。只要隐式或显式地把值类型的实例转换为引用类型，也会发生装箱操作。例如，下面的代码就创建了 `int` 类型的一个装箱版本：

```
int number = 43;           // define a value type
object o = number;         // boxes a copy of number
int anotherNumber = (int)o; // unboxes the copied value
```

在把值类型转换为接口时，也会发生装箱操作。根据定义，接口是引用类型。所以，如果要从 `Money` 结构中获得 `Comparable` 接口，下面的代码就会引发装箱操作：

```
Money someMoney = new Money(4.0); // define a value type
Comparable iface = someMoney;      // get reference to an interface
```

另一种发生装箱的情况是在检查对象的类型时：

```
Money someMoney = new Money(4.5); // define a value type
Type t = someMoney.GetType();      // access type information
```

在这种情况下，`someMoney` 也装箱了。这是因为 `Money` 没有直接实现 `GetType()`，因为它继承自 `System.ValueType`。为了实现这个方法调用，CLR 需要有一个指向 `Money()` 方法表的指针，它只能通过装箱 `someMoney` 来获得。

最好限制装箱和取消装箱的次数。如果应用程序进行许多的装箱和取消装箱操作，则把数据类型定义为类比定义为结构更有效。

注意：

`Managed Extensions for C++` 较难实现装箱和取消装箱操作。在这种情况下，必须使用 `__box` 和 `dynamic_downcast` 运算符显式装箱和取消装箱值实例。其代码较难读写，但至少知道进行了装箱和取消装箱操作，这也有其优点。

1.5 接口

前面介绍了值类型和引用类型在 C# 中如何工作，下面把注意力转到接口上。接口和类很相似，只是接口中的方法没有任何实现代码。接口只指定一组必须由其他类和结构实现的相关方法。由于接口不包含实现代码，所以不能实例化接口。在面向对象的设计中，接口类似于纯粹的抽象类。

注意:

接口也可以包含属性、索引器和事件。这类成员的更多信息请参阅第2章。

接口在面向对象的设计中发挥着非常重要的作用。可以定义接口来表示契约的职责,而不用担心应用程序中的类和结构如何完成这些职责。接口允许把对象能做什么和如何做分离开。

编写一个把接口类型作为参数的一般方法,再在其他不相关的类中实现该接口,就可以提供一种类型安全的方式,生成多个类共享的功能。如果通过接口来访问对象,该对象的类型就不重要。例如,只需在自己的类中实现 `Comparable` 接口,这些类就可以使用 `Array.Sort()` 方法在集合中正确排序。`System.Array` 类只使用这个接口访问类的实例,不关心它是对字符串、整数、苹果、梨、橙子,还是其他对象排序。

在系统设计中使⤵用接口,有几点好处:

- 可以推迟决定如何实现某个行为。在开发过程中,可以迟一些编写类和结构的实现代码。
- 允许为同一个接口提供几种可选择的实现方式。
- 为将来代码的扩展提供了一个契机。如果需要,可以在6个月之内重新考虑应用程序,定义新的实现类,而不用修改依赖它们的代码,因为接口不变。

C++和Java开发人员应很熟悉这个概念,但Visual Basic开发人员将是第一次听到。

按照约定,.NET Framework中所有接口的名称都由字母I开头。这易于把接口与类和结构区分开。例如,.NET Framework类库有`IDisposable`、`ICloneable`、`Comparable`等接口。

1.6 小结

本章介绍了.NET Framework如何定义所有.NET编程语言的通用类型系统。还研究了基本类型、类、结构、枚举、接口和委托在C#应用程序中的使用。

- 所有基本类型(除了字符串和对象之外)映射为.NET Framework类库中预定义的值类型。例如,如果在C#代码中声明一个`int`变量,编译器会把这个变量映射为`System.Int32`。然后这个变量会转换成MSIL代码中的`int32`类型。
- 类是引用类型。当创建类对象时,它放在公共语言运行时的托管堆中。类对象采用按引用复制的方式。当类对象不再引用时,这个对象就会被垃圾回收器回收。
- 结构是值类型。值的实例可以存储在堆中或托管的对象中。值的实例采用按值复制的方式。当值的实例超出作用域时,它被弹出栈并删除。值实例不受垃圾回收的影响。

- 枚举是整型数据类型，它有一个受限制的允许值的集合。在.NET Framework 中，枚举是值类型。
- 委托表示指向方法的指针。可以创建一个委托来指定所要调用的方法，然后在恰当的时候调用这个方法。在.NET Framework 中，委托构成了事件机制的基础。
- 接口指定可以在其他类和结构中实现的契约。接口可以在设计中分离“做什么”和“如何做”。

介绍了如何在 C# 中定义类型后，后面的章节将研究如何正确有效地设计和实现这些类型。

第2章 类型成员

在面向对象的编程中，最基本的组成块之一是类。除了接口和结构之外，为.NET平台编写的应用程序代码的主要部分都封装在类中。要成为一位高效的类设计师，就必须成为一位成功的面向对象的程序员。

高效的类设计还要简化维护和版本工作。一旦类设计好之后，客户开始使用它，就很难修改或升级版本了。如果从已发布的类中删除功能，就会违反与客户签订的协议，客户的代码就会失效。如果修改已有的类，可能会伤害所有使用该类的客户。这就像建一座房子，原来的设计和实现过程对于避免以后的修改是非常重要的。重新设计一座已建成的房子，比从头开始建一座新房子要困难和昂贵得多。同样，在类进入生产环境后，再修改类结构的设计也是很困难的。

在类型中定义的成员，或者叫类型成员，是用来在.NET Framework 里建立类的基本编程构造。.NET Framework 中的类型成员包括：

- 常量
- 变量 (也叫做字段)
- 属性
- 方法
- 构造函数
- 事件

所有的面向对象编程语言(C++、Java、C#和 VB.NET)都对类型成员提供了不同程度的支持。例如，C++和 Java 编程语言不支持属性，而是利用 get 和 set 方法。但是，在.NET Framework 中，所有的语言都支持 CTS。CTS 有一组通用的规则和要求来描述类型成员。

本章介绍 C#描述的.NET 类型成员。理解其他面向.NET平台的编程语言(如 VB.NET)所提供的类似类型成员是很重要的。另外，一旦编译为.NET 的内部中间语言，就很难确定代码最初是用什么语言编写的了。

我们还将讨论.NET 为类型成员提供的不同级别的可访问性，以限制这些类型成员的可见性。类常常需要限制类型成员对不同代码块的可见性。例如，类可能需要一个只能在同一个命名空间中调用的函数。应用正确的访问修饰符可以灵活地进行这种限制。

最后讨论类设计人员可用的内置类型成员，介绍.NET Framework 的根类 System.Object 提供的功能。这些内置的类型成员或默认的方法总是可以使用的，这是因为每个类都派生于 System.Object。除了可以使用默认的方法之外，还可以重写这些

类型成员，以提供特定的功能。System.Object 定义了客户代码可以在任何对象上调用的有效方法，例如 Equals()方法指定两个对象是否包含了相同的值。本章将探讨这些方法，说明如何使用其中最重要的方法。

2.1 常量

常量提供了一种非常基本但极为有用的功能。常量是不变的值的符号表示。不变的值不是硬编码为数值或字符串值，而是给它们赋予常量。在代码中不会改变的数值常常称为幻数(magic number)。这些幻数可以完成单调乏味的任务，也容易把错误引入代码。

Java 用户注意：

Java 没有常量类型，但通过使用声明为静态的公共变量，可以获得类似的效果。

为了说明常量，下面创建一个简单的猜数游戏(magicnumbers.cs)。把用于显示的代码放在主类 MagicNumbers 中。但为了封装游戏的规则，把游戏逻辑放在另一个类 GuessTheNumber 中：

```
using System;

public class GuessTheNumber
{
    public string tryGuess(int guess)
    {
        Random generator = new Random();
        int randomNumber=generator.Next(1,10);
        if (guess == randomNumber)
            return "You guessed it!";
        else
            return "Sorry, the number was " + randomNumber.ToString();
    }
}

class MagicNumbers
{
    [STAThread]
    static void Main(string[] args)
    {
```

```
Console.WriteLine("We're going to play a guessing game!");
Console.WriteLine("Please enter a number between 1 and 10:");

try
{
    int myGuess=int.Parse(Console.ReadLine());
    if ((myGuess > 10) || (myGuess <1))
        Console.WriteLine("Value must be between 1 and 10");
    else
    {
        GuessTheNumber gtn=new GuessTheNumber();
        Console.WriteLine(gtn.tryGuess(myGuess));
    }
}
catch
{
    Console.WriteLine("You need to enter a number!");
}
}
```

尽管这个例子有点做作，但说明了幻数如何导致问题的出现。在本例中，需要在两个类中存储猜测的最小和最大范围。查看一下 `MagicNumbers` 类，问题就变得比较复杂了。这里将多次提及显示给用户的值。

如果由于某种原因需要修改游戏规则，例如把规则改得难一些，让玩家猜测 1 到 20 之间的某个数，就需要找到幻数的所有实例，并替换它们。当然，这在本例中是一个相对容易的任务，但假定要在一个有上千行代码的程序中进行这样的替换，就比较困难了。在上述代码中，进行这样的替换可能会出错，因为漏掉一个数字的可能性很大。

下面修改本例，来看看常量的作用。我们的目标是为在运行期间不发生变化的值提供常量。最好把相关的常量都组合在一起。猜数游戏中的最大数和最小数就应使用常量。下面的代码可以在 `magicnumbers_revised.cs` 中找到：

```
using System;

public class GuessTheNumber
{
    public const int LOW=1;
    public const int HIGH=10;
```

```
public string tryGuess(int guess)
{
    Random generator = new Random();
    int randomNumber=generator.Next(LOW,HIGH);
    if (guess == randomNumber)
        return "You guessed it!";
    else
        return "Sorry, the number was " + randomNumber.ToString();
}
}

class MagicNumbers
{
    [STAThread]
    static void Main(string[] args)
    {
        GuessTheNumber numberGame=new GuessTheNumber();

        Console.WriteLine("We're going to play a guessing game!");
        Console.WriteLine("Please enter a number between "
            + GuessTheNumber.LOW + " and "
            + GuessTheNumber.HIGH + ":");

        try
        {
            int myGuess=int.Parse(Console.ReadLine());
            if ((myGuess > GuessTheNumber.HIGH) ||
                (myGuess < GuessTheNumber.LOW))
                Console.WriteLine("Value must be between "
                    + GuessTheNumber.LOW + " and "
                    + GuessTheNumber.HIGH);
            else
            {
                Console.WriteLine(numberGame.tryGuess(myGuess));
            }
        }
        catch
        {
            Console.WriteLine("You need to enter a number!");
        }
    }
}
```

```

    }
}
}

```

注意，这里实际上是引用类 `GuessTheNumber` 来访问常量。不是为类的每个实例定义常量，而是只定义一次。还要把常量定义为公共的。如果只在当前类中需要常量，就可以把它声明为私有的。本章的后面将介绍其他访问修饰符。

常量的实际值是在编译时确定的。代码在从 C# 编译为 MSIL 时，编译器会把常量的值插入到最终的可执行文件的元数据中。使用常量除了使代码更易于整理之外，另一个优点是直接把常量值插入到元数据中，是不需要分配内存的。但是，由于值是直接插入 MSIL 的，常量只能是基本类型。

下面看看 `MagicNumbers` 类中 `Main()` 方法的最终 MSIL。在编译代码后，使用 .NET Framework 中的 `ildasm.exe` 实用工具。该工具允许查看 MSIL 和与 .NET 程序集相关的元数据。下面的代码取自 `Main()` 方法：

```

class magicnumbers
{
    [STAThread]
    static void Main(string[] args)
    {
        GuessTheNumber numberGame=new GuessTheNumber();

        Console.WriteLine("We're going to play a guessing game!");
        Console.WriteLine("Please enter a number between "
            + GuessTheNumber.LOW + " and " + GuessTheNumber.HIGH + ":");
    }
}

```

这些代码编译为 MSIL，如图 2-1 所示。注意 `GuessTheNumber` 类中的 `HIGH` 和 `LOW` 常量的相关值嵌入 `magicnumber` 类的 MSIL。

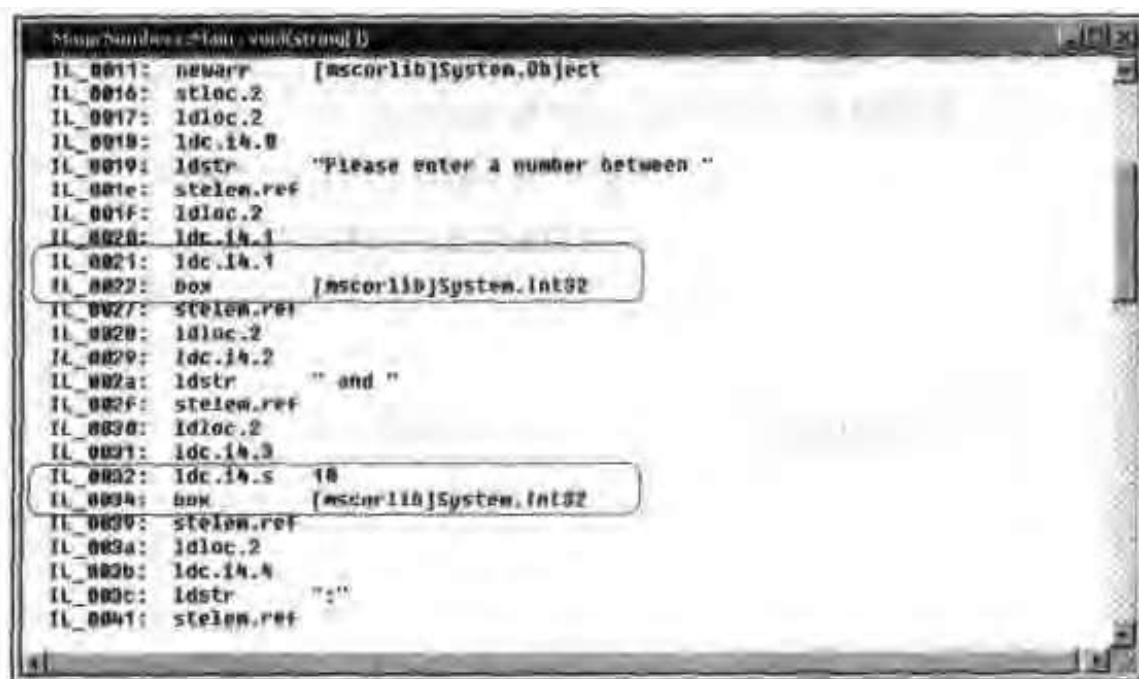


图 2-1 编译后的 Main 方法

.NET 把常量嵌入到可执行代码中，会带来一个潜在的严重缺陷。假定常量是在另一个程序集(如 DLL)中定义的，根据该 DLL 编译代码。则在编译时，常量的值要从该 DLL 中读取，并直接放在代码中。

如果 DLL 中的常量发生了变化，那么代码也需要重新编译。否则，代码不知道这个变化。这样，代码和它所依赖的 DLL 会继续执行，但有一个严重的错误，即常量不再是不变的。

假定编写一个类，它使用一个连接字符串来连接数据库。我们不是在整段代码中键入相同的字符串，而是把连接字符串编写为常量，这样就把代码与这个连接字符串联系在一起了。如果这个连接字符串的任何部分发生变化(如数据库名、IP 地址、协议、数据提供者等)，重新编译和部署使用它的所有代码是相当困难的，但只要固定这个连接字符串就能解决这些困难。

Not-so 常量

常量的值是在编译时确定的。因此常量只能是基本类型。另外，由于在编译时确定常量的值，如果常量发生变化，则依赖该常量的所有其他类也必须重新编译。

有一种方式可以获得与常量类似的功能，即把变量声明为静态的只读变量。常量是在编译时确定其值，而静态只读变量是在运行期间确定其值。使用静态只读变量有两个重要的优点：

- 与常量不同，静态只读变量在发生变化后，依赖它们的类不需要重新编译。
- 静态只读变量可以赋值并键入。其类型也不仅限于基本类型。

使用静态只读变量也有一个缺点，至少与常量相比是这样。常量是直接置换入 MSIL 的。如前所述，即使常量在类之间引用时，也是这样。而静态只读变量必须与对应的类有一个链接。

从语法上看，创建和使用静态只读变量与定义和访问常量没有什么区别：

```
public class GuessTheNumber
{
    public static readonly int LOW=1;
    public static readonly int HIGH=10;
    .....
    //code omitted for brevity
    .....
}
```

但是，对应的 MSIL 有较大的区别。静态只读变量不是简单地取代常量值，而必须建立一个对包含只读变量的类的引用，见图 2-2。

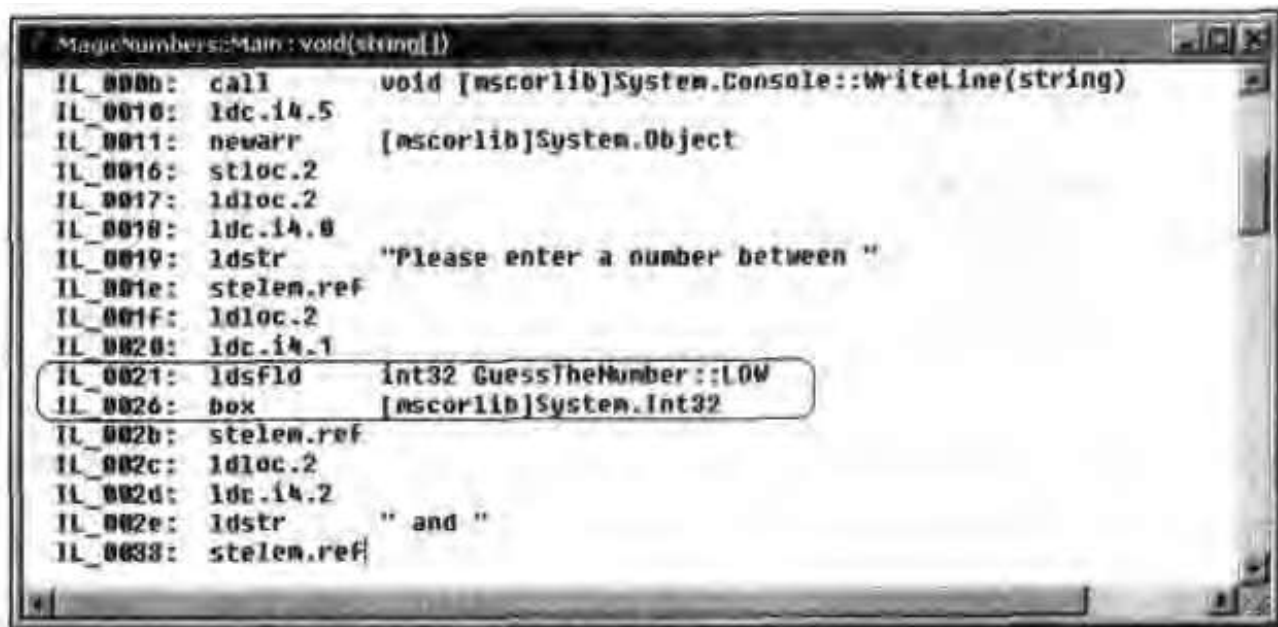


图 2-2

第三种选择是使用应用程序配置文件，这里不详细论述。在.NET Framework 中，应用程序配置是非常灵活的，是用于应用程序中信息的一种极佳机制。有关应用程序配置文件的更多内容请参阅 MSDN。

配置文件非常适合于存储面向部署的信息。例如，数据库连接信息就应存储在配置文件中。当然，由于静态只读变量是在运行期间确定的，可以用于访问和存储配置文件中的值。

下面总结一下本节学习的内容，帮助理解何时使用常量、静态只读字段，或应用程序配置文件：

- 一般情况下，只有不发生改变的量才应展示给外部的类。当然，常量只在类的内部用于最小化幻数时，也是一种非常有效的机制。在这种情况下，值即使频繁变化，也不是什么大问题，因为这些变化会应用于每次重新编译的过程中。
- 如果变量必须在运行期间确定，或必须包含对复杂类型的引用，而不是像常量那样仅使用基本类型，就应使用静态只读变量。
- 最后，配置文件可用于存储与应用程序的安装、配置和部署相关的信息。常量在.NET 中的作用非常大，但理解在任何一种编程语言中，何时使用常量，何时采用其他方式来解决问题是非常重要的。

2.2 字段

字段常常称作实例变量或类变量，是大多数类和结构中最基本的代码块。简单地说，字段存储了类和结构的数据和引用。字段就是在类中定义的变量。

字段如果使用不正确，就会出现难以处理的故障。后面在介绍可访问性时会讨论一些故障。字段的可访问性有 5 种不同的级别：

- public

访问不受限制。类的用户可以在类本身不知道改变的情况下自由读取和修改任何公共字段。使用公共字段要特别小心。

- **internal**

对内部字段的访问只限于在类所在的命名空间中进行。实际上,就是“只有该命名空间中的其他类型才能直接读取或修改这个字段”。与用公共字段一样,类对这些修改也不太清楚。也要小心使用内部字段。

- **protected**

只能访问从当前类继承和扩展的类,当然也能访问当前类。OO 设计的一个重要部分是从其他类中继承的能力。也就是说,只有当前类和从该类型派生的类型能访问这个字段。

- **protected internal**

这是惟一一个双关键字的有效访问修饰符。它把 **protected** 和 **internal** 访问修饰符的含义组合起来,表示“本命名空间中的当前类其他类型,或派生于该类的类型能访问这个字段”。

- **private**

这是访问声明中限制最多的一个修饰符。私有字段表示“只有当前类能访问这个字段。本命名空间中的其他类型和派生自当前类型的类型都不能访问该字段”。

目前,重要的是理解类设计中使用正确的可访问性是非常必要的。例如,如果字段是公共的,类的用户就可以修改它的内容,而不必让类知道进行了这个修改。

字段只是在类中定义的一个变量。当.NET 运行时创建类的实例时,就会为它分配内存,保存实例字段的内容。由于字段是在运行期间创建的,有自己的内存空间,所以字段没有像常量那样的版本问题。常量的值是在编译期间确定的。如果常量发生了变化,依赖它的类也必须重新编译。但如果字段发生了变化,依赖它的类不受影响,因为它们只是保存了字段的一个引用。

下面演示一下字段的外观。另外,还要说明关注类字段上的访问修饰符的原因。下面的程序(fields_publicbankaccount.cs)创建了一个类型为 **Account** 的对象。类 **Account** 是银行账号的一个基本表示方式,它目前只有一个字段来存储账号的余额。由于 **balance** 字段是一个 **double**,所以编译器把它初始化为 0。默认值和类的初始化详见第 5 章。

```
using System;

public class Account
{
    public double balance;
}
```



```
class AtTheBank
{
    [STAThread]
    static void Main(string[] args)
    {
        Account myAccount = new Account();

        Console.WriteLine("You have " + myAccount.balance + "
                           remaining.");
        myAccount.balance=10000;
        Console.WriteLine("You have " + myAccount.balance + "
                           remaining.");
    }
}
```

编译并运行上面的代码，生成如下输出：

```
C:\Class Design\Ch 02>fields_publicbankaccount.exe
You have 0 remaining.
You have 10000 remaining.
```

查看一下为 Account 类创建的 MSIL，注意 balance 字段获得了它自己的存储位置。MSIL 只确认 balance 变量的类型是 float64(double)，其访问修饰符是 access，其名称是 balance。见图 2-3。

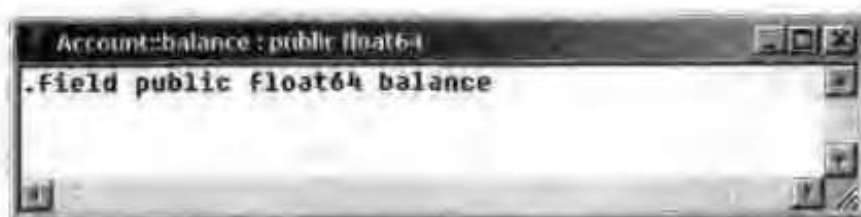


图 2-3

把这个字段声明为公共的，就使它成为类接口的一部分。也就是说，如果以后需要修改该字段名，就会损害二进制兼容性。另外，如上所述，把这个字段声明为 public，就可以不受限制地访问实例化该类的其他类。在一些情况下，这是可以接受的，甚至是我们希望的。但在大多数情况下，在这个银行账号的例子中，不希望其他人修改这个字段的状态。毕竟，让银行客户确定他们自己的余额是不合适的。

公共字段使用起来很简单，但非常危险，最好不要使用。这有两个原因：

- 公共字段是不受限制的

在本例中，AtTheBank 类可以把 balance 的值修改为任何 float64 值。如果要求 balance 不小于 0，则使用 Account 类的所有代码在设置公共字段 balance 时，都必须遵守这个规则。换言之，就是允许客户在不理解所有业务规则的情况下修改该字段。

- 公共字段使类很难升级

部署以后，就很难修改 `balance` 字段的名称，因为要考虑到二进制兼容性。也不能把公共字段转换为属性，从而避免损害二进制兼容性。

实际上，应尽量避免使用公共字段，除非进行特殊的调整，才允许使用它们。例如，有时，可以让类或结构只存储要在其他类之间传送的状态。但是，作为规则，任何要展示到公共接口上的数据采用属性的形式会比较灵活。

2.3 属性

属性表示类中的状态。但是，属性与字段完全不同。字段存储的是数据，而属性提供了对该数据的一种特定访问。属性可以限制输入的值，提供了只读和只写的数据成员，从而扩展了字段的概念。

Java 和 C++ 用户注意：

Java 和 C++ 都没有属性的概念。但 Java 以等同的方式为展示的字段定义了一个很好的模式。在 Java 中，标准方式是创建 `getFieldName()` 和 `setFieldName()` 方法，执行与属性类似的任务。.NET 中的属性可以看做是这种模式的一种正规实现。

在上一个例子中，如果业务要求 `balance` 的值不能小于 0，则在设置属性时就可以应用这一规则。在下一个例子中，如果参数违反了规则，就抛出一个 `ArgumentOutOfRangeException` 异常。下面的程序(`properties_bankaccount.cs`)演示了这个过程：

```
using System;

public class Account
{
    private double balance;
    public double Balance
    {
        get
        {
            return balance;
        }
        set
        {
            if (value < 0)
```

```

        throw
        new ArgumentOutOfRangeException("value", "Balance must
                                   be greater than 0");

        balance=value;
    }
}

```

注意，我们引用了值，但没有声明它。在设置属性时，隐式地设置了 `value` 关键字。例如，`myAccount.Balance=500` 会把 500 加载到 `set` 方法中引用的值参数中：

```

class AtTheBank
{
    [STAThread]
    static void Main(string[] args)
    {
        Account myAccount = new Account();

        try
        {
            Console.WriteLine("You have " + myAccount.Balance + "
                               remaining.");
            myAccount.Balance=10000;
            Console.WriteLine("You have " + myAccount.Balance + "
                               remaining.");
            myAccount.Balance=-500;
            Console.WriteLine("You have " + myAccount.Balance + "
                               remaining.");
        }
        catch(ArgumentOutOfRangeException e)
        {
            Console.WriteLine("Bad value!");
        }
    }
}

```

属性在把数据展示给外部类的方式上有更大的灵活性。属性提供了数据封装。在本例中，用户不再能直接访问账号的余额，而只能通过 `Balance` 属性来访问它。`Balance` 属性根据一个基本业务规则来设置账号的余额。在本书的其他部分，`balance` 是作为公共字段来访问的，但目前只有 `Account` 类负责执行业务规则。

在类的实现代码需要修改时，使用数据封装的优势就体现出来了。数据封装允许控制用户能把哪些值赋予属性，以防止出现无效的状态，因为如果用户破坏了对对象的状态，可以过滤和拒绝输入的值。在本例中，如果以后修改了规则，允许输入负余额，但最大只能输入 500 美元，就可以修改 Account 类，而不破坏其他依赖于 Account 的类。属性还提供了比公共字段更多的值，因为属性可以定义为只读或只写。

在 C# 中，把属性定义为只读或只写只需删除相应的设置程序和获取程序。如果试图获取只写属性的值或设置只读属性的值，编译器在编译时就会产生一个错误，这包括在表达式的右端或语句中访问只写属性。

在 .NET 环境中，属性与其他类型成员有所不同。在 MSIL 中，属性分解为 `get_` 和 `set_` 方法。例如，读写属性 `Balance` 有 `get_Balance` 和 `set_Balance` 访问器方法，它们是在 Account 类的 MSIL 中定义的。编译器还会把一个属性定义发送给 MSIL，以提供属性与 `get_` 和 `set_` 访问器方法之间的关联。下面回到 Account `Balance` 属性上：

```
using System;
public class Account
{
    private double balance;

    public double Balance
    {
        get
        {
            return balance;
        }
        set
        {
            balance=value;
        }
    }
}
```

但是，问题在于，为什么要使用属性？如果属性的目的就是重定向到其他访问器，那么为什么不使用属性，而只添加两个方法，作为私有字段的访问器？

最简单的答案是属性可以大大提高可读性，下面的两个方法都设置了账号的余额：

```
Account myAccount = new Account();
double myBalance;

myAccount.Balance=1000;
```

```
myBalance=myAccount.Balance;
```

和

```
myAccount.setBalance(1000);  
myBalance=myAccount.getBalance();
```

这两个函数都可以使用,对大多数人来说,第一个函数比较容易理解,也比较自然。另外, Visual Studio .NET 等开发工具使用属性 MSIL 块,为开发人员提供类的更多信息。例如,在 Visual Studio .NET 提供自动完成编码功能时,可以区分方法和属性,给属性提供不同的图标,用于显示。

但是,使用属性的第一种方法也有一个缺点。前面讨论了字段可以设置为不同的访问级别,属性也可以设置。缺点就是可访问性级别只能应用于属性,不能应用于 `get` 和 `set` 访问器。如果因某种原因,访问器必须有不同的可见性,例如公共获取访问器和受保护的设置访问器,则属性就不再是最佳的解决方案了。当需要不同级别的可访问性时,应定义自己的获取和设置方法。属性详见第 4 章。

2.4 方法

方法是 C# 中设计类的基础,它们表示与类相关的动作。方法通常执行操纵对象状态的动作。但并不总是这样,例如,可以把方法声明为静态方法。静态方法可以提供特殊的功能,但不能处理对象的状态。与其他静态对象一样,静态方法属于类,而不属于类的某个实例。

所有方法都有一个名称、一个参数列表,以及一个返回类型。还可以定义有相同名称的多个方法,只要它们包含不同的签名。下一章将详细讨论方法。

方法的重载

方法的签名包含方法的名称、参数的数量和类型。例如,如果两个方法有相同的名称,仅参数的数量和类型不同,那么 .NET Framework 就认为这两个方法是不同的。用相同的名称和不同的签名创建方法的方式称作方法重载。如果在同一个类或结构中存在两个方法,它们的签名相同,编译器就会在编译时生成一个错误。

注意,方法的签名中不包含返回类型。这是因为 C# 中的方法不能仅根据它们的返回类型来重载。.NET 平台不遵守这个规则,但 C# 语言遵守。CLR 支持两个或多个仅返回类型不同的方法,但 C# 语言(和许多其他的 .NET 语言,包括 VB.NET)不支持。如果同一个类中的两个方法仅返回类型不同,就会生成一个编译器错误。每种语言都有

这个限制，但.NET运行时没有。实际上，问题是在类设计中很少有限制因素。

2.5 属性与方法

属性和方法非常类似。实际上，属性是定义方法的语法快捷方式。方法和属性都可以用来执行动作，但属性应用来表示数据，而方法应用来执行动作。通常，属性只用来修改(获取或设置)类中一个变量的状态。方法则常用于执行修改对象全部状态的动作。换言之，方法执行对象的业务功能。

从.NET的角度来看，属性和方法没有区别。在.NET中，属性就可以看做是方法。属性的获取和设置访问器总会编译到它们自己的方法中。

设计类时，肯定要选择是使用属性还是使用方法。此时最好问问自己，这个程序的目的是设置或检索数据，还是执行更普通的动作。用属性来设置和检索数据，而用方法来执行动作。

命名属性和方法时，坚持用名词来命名属性，以表示数据，而对方法名称用动词来命名，以表示动作。例如，如果扩展 Account 类，使之包含客户信息，则 Name、Address 和 Telephone 就是理想的属性名称，因为它们可以说明数据。而 Withdraw()、MakeDeposit()和 Transfer()是理想的方法名称，因为它们描述一个动作。

更多命名约定的指导原则，请参阅在.NET SDK 的 Design Guidelines for Class Library Developers 中提出的规范。.NET SDK 可以从 MSDN 的 <http://msdn.microsoft.com> 免费下载。

2.6 静态类型成员

本章前面介绍的字段、属性和方法仅在对类的实例有一个引用时可以被访问，但是，并不是所有的功能都适合于类的实例。.NET 允许定义可以直接从类中调用、不需要创建类实例的类型成员。字段、方法和属性都可以以这种方式声明。

如果某个动作与类的实例无关，但其功能可以独立使用，就可以使用静态类型成员。例如，Time 类定义了一个方法 GetCurrentTime()，它返回当前的系统时间。只是为调用 GetCurrentTime()而创建 Time 的一个实例并不好，因为所有的类返回的值都相同。而这个方法对 Time 类是公共的，不是专门用于某个实例的，所以最好作为静态成员来实现。

静态类型成员在.NET 基类库中是很常见的。例如 System.DateTime 类就有静态的类型成员，System.DateTime.Now 是一个返回系统日期的静态函数，如 static_time_example.cs 所示。注意它直接使用了 DateTime 类，不需要创建 DateTime 类的实例来调

用 Now 类型成员。DateTime 类中的 Now 属性声明为 static:

```
using System;

public class static_time_example
{
    [STAThread]
    static void Main(string[] args)
    {
        Console.WriteLine(DateTime.Now);
    }
}
```

一般说来,静态成员属于类。非静态的,或实例成员属于对象或类的实例。这就是常量和静态只读字段的实现方式。实际上,常量是一种特殊类型的静态字段。

字段是对象的数据的存储位置。系统会为对象的每个实例创建一个新的存储位置。如果字段是静态的,就会有所区别,静态字段只表示一个存储位置。无论创建了多少个类的实例,静态字段都只有一个副本。

下面对 Account 类作一个小的改动,进一步说明静态字段如何影响程序。下面的代码在 static_bankaccount.cs 中:

```
using System;

public class Account
{
    private static double balance;

    public double Balance
    {
        get
        {
            return balance;
        }
        set
        {
            balance=value;
        }
    }
}
```

```
class AtTheBank
{
    [STAThread]
    static void Main(string[] args)
    {
        Account mySavings= new Account();
        Account myChecking=new Account();

        mySavings.Balance=500;

        Console.WriteLine("Savings balance: " + mySavings.Balance);
        Console.WriteLine("Checking balance: " + myChecking.Balance);
    }
}
```

对 Account 类的惟一修改是定义 balance 字段的方式:

```
private static double balance;
```

然而, 尽管这种改动很小, 但对程序的影响非常显著。再看看 Account、mySavings 和 myChecking 的两个实例:

```
Account mySavings= new Account();
Account myChecking=new Account();

mySavings.Balance=500;

Console.WriteLine("Savings balance: " + mySavings.Balance);
Console.WriteLine("Checking balance: " + myChecking.Balance);
```

如果 balance 是实例变量, mySavings 和 myChecking 就有自己的 balance 字段。上述程序的输出应显示 mySavings 实例的 balance 字段是 500, 而 myChecking 的 balance 字段是默认值 0。但是, balance 不是实例字段, 而是静态字段, 所以 balance 的值由 Account 的所有实例共享。

如果方法声明为静态, 则该方法就在类上执行动作, 而不是在类的实例上执行动作。静态成员只能访问和修改静态字段。下面再次修改 account 类, 说明静态方法的使用。下面的代码在 static_bankaccount2.cs 中:

```
using System;

public class Account
```



```
{
    private static double balance;

    public double Balance
    {
        //...omitted for brevity
    }

    public static void addTen()
    {
        balance=balance+10;
    }
}

class AtTheBank
{
    [STAThread]
    static void Main(string[] args)
    {
        Account mySavings= new Account();
        Account myChecking=new Account();

        mySavings.Balance=500;
        Account.addTen();

        Console.WriteLine("Savings balance: " + mySavings.Balance);
        Console.WriteLine("Checking balance: " + myChecking.Balance);
    }
}
```

静态方法在类上执行动作，而不是在类的实例上执行动作。静态方法可以访问静态字段、静态属性和其他静态方法。同样，不能通过 Account 类的 mySavings 或 myChecking 实例来调用 AddTen()，而必须直接在 Account 类上调用。但是，从输出中可以看出，该方法的调用仍会影响每个实例的 balance 字段。这是因为 AddTen() 添加了静态 balance 字段。下面是上述程序的输出：

```
C:\Class Design\Ch 02>static_bankaccount2.exe
Savings balance: 500
Checking balance: 500
Savings balance: 510
```

Checking balance: 510

在使用静态方法和字段时必须小心。只有当字段必须由类的所有实例共享时，才应标记为静态。同样，只有当方法和属性必须作用在静态字段上时，才应标记为静态。使用静态方法和字段的理想情况是和只读属性一起使用。

2.7 事件和委托

事件是相当简单的。前面说明了类应做什么工作。而方法调用还会有反馈。也就是说，如果要取款，就告诉 Account 类去取款。如果想知道余额，就可以询问 Account 类。利用事件，可以让类告诉我们何时发生了什么事。

假定现在有一个 BankManager 类，银行经理想知道何时有人从账号中提取了超过 10000 美元的款项。一种效率不太高的方法是让 BankManager 类询问每个 Account 实例，看看是否有大额取款。另一种方法是让每个 Account 实例保存一个对 BankManager 的引用。当 Account 接收一笔大额取款时，就可以告诉银行经理。这种方式比较好，至少目前看来是如此。只要银行雇用一位审计员来了解所有的取款情况，就需要重新访问 Account，添加另一个引用，这次是对审计员的引用。

一种更简单的方法是让对此信息感兴趣的所有类监视 Account。这样，审计员就不必监视和注意每笔取款了。而且，银行经理可以只留意大额取款。从某种程度上讲，银行经理和审计员都是一个主题，Account，的观察员。

事件的核心是另一个类型成员——委托。委托基本上是对另一个类的方法的引用。如果读者有 C 或 C++ 编程背景，就知道通过使用函数指针就可以获得类似的能力。但函数指针的一大缺陷是类型不安全。换言之，它们可能会试图访问无效的内存地址。而利用委托，.NET 改变了这一切。现在有一流的成员用于回调函数，更好地确保运行期间的正常工作。

事件对于设计类是非常强大和有效的机制。我们不必连续不断地监控对象发生了什么变化，而可以通过引发事件来了解对象的变化。另外，以后添加更多的订阅者也很容易。用这种方式，发布的对象不与订阅者关联起来。

但是，事件和委托应只用于最适合使用它们的情况。它们是设计工具箱中的另一个工具，而不是惟一的工具。事件的一大缺点是不必对它们执行动作。我们并不要求类必须捕获其他类产生的事件。这样订阅者就不会被请求监听类所生成的特定事件。同样，也不应生成对类的正常操作至关重要的事件。例如，不能把 Account 类设计成依赖 Auditor 类来响应事件。这是因为只使用事件，并不能确保 Auditor 会订阅该事件。第 6 章将详细论述事件和委托。

2.8 运算符

与大多数编程语言一样，C#也允许定义运算符的实现方式。尽管这会在动作方面产生混淆，但运算符重载是一项非常有效的功能，可以用于在类中提供定制的算术或比较操作。

注意：

创建自己的运算符的功能，在C++中是非常强的。Java没有内置对这种功能的支持，因为这需要预处理器。但是，也有软件包允许利用Java进行这种功能的开发。

几乎所有的一元和二元运算符都可以重载其默认的行为。一些运算符还要求其他运算符也实现重载。例如，如果定义了小于运算符(<)，则必须定义大于运算符(>)。

定义运算符还有几个规则。首先，运算符的签名必须是 `public static`。其次，只有值类型能传送给运算符。试图给运算符传送引用类型会生成一个编译期间错误。运算符详见第4章。

2.9 构造函数

当创建一个对象时会给它一个默认的初始状态。字段在声明时可以设置一个初始值，或者接受其声明类型的默认值。通常情况下，需要对初始状态有更多的控制，创建对象的代码可以执行无法用简单的赋值语句完成的初始化例程。

构造函数是在引用返回给调用程序之前，用来初始化对象状态的类型成员。它们类似于方法，因为它们可以带0个或多个参数。构造函数没有返回类型，在实例状态字段被赋予了初始值之后执行。与方法一样，构造函数可以重载。

构造函数不同于普通方法，因为只在创建对象时隐式调用它。创建完对象之后，就不能调用构造函数了，它也不位于对象的声明空间。如果类没有指定构造函数，仍会获得一个默认的空参数构造函数。

如果业务规则要求，对象从一开始就需要有某些值，就不要提供空的构造函数。

与其他类型成员一样，也可以对构造函数设置可访问级别。有时把构造函数设置为 `protected` 或 `internal`，则派生类或同一个应用程序中的类就可以用一个方法创建对象，用另一个构造函数把对象重新设置为对象刚开始创建时必须设置的默认值。

在扩展基类时，构造函数还有一个重要的作用。在构造对象时，派生的对象可以把初始化的某部分交给其基类来完成。这部分内容详见第5章。

析构函数

构造函数是在对象创建时调用，析构函数则在删除实例时调用。析构函数不能有任何参数，也不能被重载。类至多有一个析构函数。还要注意，析构函数不能继承。

C++用户注意：

在 C++ 中，程序员需要编写一个析构函数，来管理内存的释放，执行清理工作。在 Java 中，没有析构函数，而让垃圾回收器来处理清理工作。在 C# 中，综合了这两种方式的优点。可以利用垃圾回收器来管理与实例删除相关的所有任务，也可以编写自己的析构函数，在对实例进行垃圾回收时调用。

析构函数详见第 5 章。

2.10 使用 System.Object

.NET Framework 内部的所有类最终都派生于.NET 的根类 System.Object。如果类不继承于另一个类，该类将隐式地继承自 System.Object。编译器会强制执行这个继承，因此通常不必编写下面的代码：

```
public class Account: System.Object
```

因为这是隐式的。由于所有的类都直接或间接继承自 System.Object，则很容易忘记 System.Object 提供的功能。System.Object 的几个成员是虚方法。可以重写这些虚方法，提供自己的实现代码。重写 System.Object 的成员可以让类与.NET 基本类库里的那些类有更好的一致性。大多数.NET 类都可以重写 System.Object 的方法，以提供适合这个类的功能。下述 System.Object 方法通常在类中实现：

- ToString()
- Equals(object)
- GetHashCode()

下一节将探讨这些函数的工作方式，以及如何在 Account 类中重写它们。但首先要讨论 System.Object 中另一个非常有用的方法。

2.10.1 GetType()

GetType()方法是 System.Object 类的一个非常简单但非常有用的成员。考虑下面的程序：

```
using System;

public class Account
{
    // Code omitted for brevity
}

class AtTheBank
{
    [STAThread]
    static void Main(string[] args)
    {
        Account myAccount= new Account();

        Console.WriteLine("myAccount is an object of type : " +
                           myAccount.GetType());
    }
}
```

编译并运行上面的代码，得到如下输出：

```
C:\Class Design\Ch 02>sysobject_gettype_bankaccount.exe
myAccount is an object of type : Account
```

可以看出，`GetType()`返回实例的类型。在本例中，返回的值是 `Account`，因为 `myAccount` 是 `Account` 对象的一个实例。

2.10.2 ToString()

`ToString()`一般是默认类成员。例如，假定有下面的函数调用：

```
Console.WriteLine(Object);
```

它等效于：

```
Console.WriteLine(Object.ToString());
```

这是因为，如果没有指定类成员，就默认调用 `ToString()`。这个调用返回当前对象的字符串表示。在默认情况下，`System.Object.ToString()`方法只返回实例的类型名称。下面的代码位于 `sysobject_tostring_bankaccount.cs` 中：

```
using System;
```

```
public class Account
{
    private double balance;

    public Account(double startingBalance)
    {
        balance=startingBalance;
    }
}

class AtTheBank
{
    [STAThread]
    static void Main(string[] args)
    {
        Account myAccount= new Account(500);

        Console.WriteLine("ToString:" + myAccount.ToString());
        Console.WriteLine("Same as above:" + myAccount);
    }
}
```

编译并运行上面的代码，得到如下结果：

```
C:\Class Design\Ch 02>sysobject_tostring_bankaccount.exe
ToString:Account
Same as above:Account
```

在大多数情况下，ToString()的默认实现方式不是很有用。毕竟，调用 GetType()也可以获得相同的信息，利用默认的 ToString()方法，无论对象的实例状态是什么，类的每个实例都会返回相同的结果。我们需要的是一个对象实例专用的描述字符串。重写 System.Object.ToString()可以为实例定制字符串表示，如下面的 sysobject_tostring_bankaccount2.cs 所示：

```
using System;

public class Account
{
    private double balance;

    public Account(double startingBalance)
```



```
        {  
            balance=startingBalance;  
        }  
  
        public override string ToString()  
        {  
            return "Object: " + GetType() + " balance: " + balance;  
        }  
    }  
  
    class AtTheBank  
    {  
        [STAThread]  
        static void Main(string[] args)  
        {  
            Account myAccount= new Account(500);  
  
            Console.WriteLine("ToString:" + myAccount.ToString());  
        }  
    }
```

编译并运行这个代码，将生成如下输出：

```
C:\Class Design\Ch 02>sysobject_tostring_bankaccount2.exe  
ToString:Object: Account balance: 500
```

2.10.3 Equals()

我们经常需要比较对象的等价性。System.Object.Equals()就提供了这样的功能。 .NET 定义了两种形式的等价，引用等价和值等价。默认情况下，System.Object.Equals() 将测试引用等价。当两个引用指向同一个基本对象时，就是引用等价。下面的例子 sysobject_equals_bankaccount.cs 说明了这点：

```
using System;  
  
public class Account  
{  
    private double balance;  
  
    public Account(double startingBalance)
```

```

        {
            balance=startingBalance;
        }
    }

class atthebank
{
    [STAThread]
    static void Main(string[] args)
    {
        Account myAccount = new Account(500);
        Account mySavings = new Account(500);

        Console.WriteLine(myAccount.Equals(mySavings));
        mySavings=myAccount;
        Console.WriteLine(myAccount.Equals(mySavings));
    }
}

```

第一个等价测试返回 false, mySavings 和 myAccount 有相同的值, 但它们没有引用相同的实例。把 mySavings 设置为 myAccount 后, 下一个等价测试就返回 true。得到的输出如下:

```

C:\Class Design\Ch 02>sysobject_equals_bankaccount.exe
False
True

```

许多实际的对象都需要测试值等价。如果两个对象的状态相同, 其值就是相等的。重写 System.Object.Equals() 方法允许执行自定义的值等价测试。下面的代码保存在 sysobject_equals_bankaccount2.cs 中, 它是一个重写默认 Equals() 实现代码的例子:

```

using System;

public class Account
{
    private double balance;

    public Account(double startingBalance)
    {
        balance=startingBalance;
    }
}

```


在下面的前两个语句中，进行了一个检查，确保没有传送空对象，且传送的对象具有相同的类型。如果这两个测试都不是 true，那么继续比较就没有任何意义。否则，就把对象的类型转换为 Account 类型，与 balance 实例字段进行等值比较。类型转换是必须的，因为默认的对象没有 balance 字段。

```
public override bool Equals(object obj)
{
    if ((obj == null) || (GetType() != obj.GetType()))
        return false;
    Account acct=(Account)obj;
    return acct.balance == balance;
}

class AtTheBank
{
    [STAThread]
    static void Main(string[] args)
    {
        Account myAccount = new Account(500);
        Account mySavings = new Account(500);

        Console.WriteLine(myAccount.Equals(mySavings));
    }
}
```

执行这段代码，返回的结果是两个对象等价。这是因为我们为 Equals() 提供了自己的实现代码。该实现代码测试的是值的等价比较。还要注意，这个比较不太牢靠，因为没有确定两个账号是否有相同的值。但是，它说明了如何建立自己的等价测试方式。

在编译上面的例子时，它可以工作，但会出现一个警告：未重写 GetHashCode() 方法。下面就介绍 GetHashCode() 的用法。

2.10.4 GetHashCode()

GetHashCode() 返回一个散列码。这个函数用于散列算法和一些数据结构，如散列表。 .NET Framework 中包括了以 Hashtable 作为开发人员可用的基本集合对象。Hashtable 根据对象的整数散列码存储对象。Hashtable 类使用散列码在内部排序对象，提供了更快的排序和搜索方法。

按照散列码来存储, 允许 Hashtable 用最小的耗费在较大的集中查找到某个对象或某个对象集。散列码从何而来呢? 为了给类提供散列码, 必须重写 `System.Object.GetHashCode()`。

`System.Object.GetHashCode()` 的默认实现代码只保证每次为相同的实例返回相同的散列码, 而不能保证不同的实例有不同的散列码, 也不能保证相同值的不同实例具有相同的散列码。重写 `System.Object.GetHashCode()` 可以让程序员根据对象的状态返回一致的散列码。

重写 `GetHashCode()` 方法时, 应遵循下面几个原则:

- 如果两个对象的类型相同, 值也相同, 就必须返回相同的散列码。
- 散列码应建立在不变成员的基础之上。不变成员的值在创建后就不能改变。假定为一个对象指定了散列码, 存储了该对象, 然后改变了散列码所依据的值, 这样就不可能检索到该对象, 因为用于查找该对象的散列码已经发生了改变。
- 确保散列值是随机分布的。随机分布的散列值会有较好的搜索性能。
- 散列码的计算应很容易。也就是说, 计算实例的散列码不需要很长的时间。

如果没有重写 `GetHashCode()` 方法, 就不要依赖散列码来保存对象。.NET 的不同版本会为同一个实例生成不同的散列码, 另外, 也不能保证所使用的散列码与所保存的对象保持一致, 例如在把状态保存到数据库中时, 主键总是有效的。

下面为 `Account` 类实现一个简单的 `GetHashCode()`, 该代码在 `sysobject_gethashcode_bankaccount.cs` 中:

```
using System;

public class Account
{
    private double balance;
    private int pin;

    public Account(double startingBalance, int PIN)
    {
        balance=startingBalance;
        pin=PIN;
    }

    public override int GetHashCode()
    {
        return (pin ^ Convert.ToInt32(balance));
    }
}
```

```
class AtTheBank
{
    [STAThread]
    static void Main(string[] args)
    {
        Account myAccount = new Account(500,1234);

        Console.WriteLine(myAccount.GetHashCode());
        Console.ReadLine();
    }
}
```

注意，上面的实现代码远不是健全的散列码实现方式。在现实世界中，账号的 PIN 是不变的(至少在一定的时间内不变)，因此就有一个明显的问题，因为在一个大银行中，许多账号可能有相同的 PIN 值。

2.10.5 Equals()和 GetHashCode()

同时实现 Equals()和 GetHashCode()是很重要的。Hashtable 类使用 GetHashCode()来确定如何对数据排序，使用 Equals()来验证对象是否等价。如果忽略了 Equals()方法，Hashtable 就不能验证两个对象是否等价。如果忽略了 GetHashCode()方法，Hashtable 就不能有效地排序。Hashtable 类在它的设计中同时使用 Equals()和 GetHashCode()。忽略任何一个都会产生意想不到的结果。

无论在应用程序中是否使用 Hashtable 类，重写 Equals()和 GetHashCode()都会使类接口更直观，可预测性更强，便于开发人员使用该类。许多算法都实现了排序和比较。实现 Equals()和 GetHashCode()允许类参与许多.NET 的预建对象(如 Hashtable)，开发人员也可以编写排序和搜索算法，生成自己的过程，让对象利用这些成员。

2.11 小结

类型成员是用来建立类或结构的编程构造。本章讨论了每个类型成员，以及和它们相关的使用规则。

常量主要用于读取，允许对已知值进行一次性的修改。默认情况下常量是共享成员，只要保证不修改值，就可以使用常量。

字段是用来在类或结构中保存状态的变量。要提供数据封装，通常用 private 访问

修饰符来创建字段。Public 字段把类接口和数据锁定在一起，在不破坏兼容性的情况下修改代码是比较困难的。

属性表示类展示的数据。属性在类的内部和外部表示之间添加了一个抽象层，由此扩展了字段的概念。属性允许设计者修改类中状态的管理方式，同时不影响状态在类外部的管理方式。只要去除获取或设置访问器，就可以把属性设计为只读或只写，以防止有害的属性用法。

方法执行类的功能。多个方法可以拥有相同的名称和不同的签名，以提供多种实现代码。这种做法叫做方法重载。

字段、属性和方法都可以声明为静态的。静态类型成员可以在没有对象引用的情况下访问，一般用于处理与类本身相关的操作，而不是与某个实例相关的操作。

事件为类提供了一个方法，以便在动作发生时通知侦听器。由于不需要处理事件，它们通常用来发送对类的关键操作不重要的信息。

构造函数允许自定义对象的创建过程。如果没有指定构造函数，C#编译器会把一个默认构造函数插入到类的 MSIL 中。

类型成员有可访问性。公共成员没有访问方面的限制。受保护的成员只能由派生类从外部访问。内部成员可以由同一个命名空间中的代码访问。私有成员只能在包含类中访问。

System.Object 是 .NET Framework 中每个类型的根类。因为所有类型都继承自这个基类，所有类型都具有可用于它们的特定方法。

第3章 方 法

方法是 C# 类设计和 C# 编译器生成的 MSIL 代码的基础。本章将介绍这两个领域，说明如何在类中设计和编写有效的方法。

方法表示与类相关的动作。方法是封装离散的任务或过程的一组语句。在运行期间，客户代码看不到方法的内部工作情况，但可以调用类中的方法，而无需了解方法的实际工作情况。在面向对象的编程术语中，这称为封装。实际上，方法表示类的行为。

方法的名称应描述它们执行的任务。选择方法的名称是类设计过程中的一个重要任务，因为方法名称构成了类设计人员和使用类的客户代码之间的部分契约。C# 是一种区分大小写的语言，所以有可能创建只是方法名的大小写不同的两个方法。这是应避免的，因为这会引起混淆。

3.1 方法的调用

本章将说明不同类型的方法、参数类型，和在 C# 中设计类时要考虑的其他问题。首先，考虑一下调用方法时 .NET CLR 内部发生了什么。方法的调用包括许多预处理活动或后期处理活动。首先，CLR 通过创建一个栈框架，给方法分配内存。栈框架是一个基于栈的内存块，它的大小足以容纳传送给方法的参数和在方法内部定义的局部变量。栈框架是一个临时的内存块，因为方法执行完后，该框架就立即从栈上弹出，释放内存。

所有的应用程序，无论是简单的控制台应用程序，还是多层分布式系统，都是通过一系列方法调用而建立的。在调用方法序列时，系统就会为每个方法创建一个栈框架，并添加到调用栈上。这个过程建立了一个栈跟踪。栈跟踪是栈框架的一个有序集合，它实际上提供了应用程序中方法调用序列的详细记录。每个栈框架都包含传送给方法的参数和方法中的所有变量。

栈框架和堆跟踪信息在调试时非常有用，因为调试时需要知道从应用程序的一个区域到下一个区域的调用序列。还需要知道给每个方法传送什么参数，这些参数在方法内部发生了什么变化。

.NET Framework 允许通过 `System.Diagnostics` 命名空间中的 `StackFrame` 和 `StackTrace` 类来访问栈框架和栈跟踪。下面的例子说明了如何使用这两个类，源代码在 `stack_trace.cs` 中：

```
using System;
using System.Diagnostics;

class Foo
{
    static void Main()
    {
        Bar.BarMethod(42);
    }
}

class Bar
{
    public static void BarMethod(int x)
    {
        StackTrace st = new StackTrace();

        for(int i=0; i < st.FrameCount; i++)
        {
            StackFrame sf = new StackFrame();
            sf = st.GetFrame(i);
            Console.WriteLine(sf.GetMethod());
        }
    }
}
```

这个例子调用了—个静态方法 `Bar.BarMethod()`。这个方法通过迭代由 `StackTrace.GetFrame()`方法返回的栈框架对象的内部集合，把—个简单的栈跟踪放在控制台窗口上。下面的输出显示在控制台窗口上：

```
C:\Class Design\Ch 03>stack_trace
Void BarMethod(Int32)
Void Main()
```

可以看到组成栈跟踪的方法，最近调用的方法在最前面。这里显示的信息与创建自己的应用程序时 Visual Studio .NET 产生的一些调试消息类似。`System.Diagnostics` 命名空间中的类提供了一种调试应用程序中方法的有效方式。使用上面例子中的规则，就可以创建自己定制的栈跟踪，找出调用自己方法的那些方法。

3.2 方法的作用域和可见性

与所有其他类型成员一样，可以用熟悉的關鍵字 `public` 和 `private` 等来指定方法的作用域和可见性。这些关键字称为访问修饰符，且必须放在方法声明中的最前面。它们的工作方式与第 2 章介绍的其他类成员一样。

在设计类时，应花一些时间来确定所有类成员的可见性的正确级别。为了遵循封装、数据隐藏和抽象的规则，只有构成外部契约的一部分的那些成员才应声明为 `public`。在类的内部使用的实用工具和帮助例程应总是具有类级别范围内的 `private` 可见性，或程序集级别范围内的 `internal` 可见性。

控制 `protected` 成员的规则比较复杂，由内在的设计要求来确定。`protected` 成员的讨论详见第 7 章。

3.3 方法类型

根据类的设计模式，一些方法在类的特定实例上执行动作，而另外一些方法的目的比较中立，它执行的动作不针对类的某个特定实例。这两种类型的方法分别称为实例方法和静态方法。

实例方法执行针对类的某个实例的动作。在一个对象上调用方法时，该方法不对其他对象直接起作用。如果实例方法共享公共资源，如数据库，就会对其他对象有间接的影响。实例方法也可以使用 `this` 关键字。`this` 变量是在其上调用，方法的对象实例的引用。`this` 可以用于访问其他对象实例成员，例如私有方法和私有字段。

当然，在用户定义的值类型中，仍可以定义实例方法，也可以使用 `this` 关键字。但此时它不是对在其上调用方法的对象实例的引用；它在非引用类型上是没有意义的。在值类型上调用方法时，值本身复制到新的栈框架上，使用名称 `this` 可以访问该值，然后在方法执行完后，值就复制回最初的位置。也就是说，在值类型的实例方法中，可以把一个新值赋予 `this`，并在方法执行完后把这个值复制回原来的位置。

当方法不应用于类的特定实例时，可以使用静态方法。与实例方法不同，静态方法不能访问 `this`，只能访问类中的其他静态成员。

3.3.1 实例和静态方法的例子

下面的例子 `method_types.cs` 说明了上述要点。该例子以 `Account` 类为基础来表示一个银行账号。该类支持两个方法 `GetAllAccounts()` 和 `DepositMoney()`：

```
using System;
using System.Data;

public class Account
{
    public static DataSet GetAllAccounts()
    {
        return (new DataSet());
    }

    public void DepositMoney(decimal amount, string accountNumber)
    {
        if (this.validateAccount(accountNumber))
        {
            // Deposit Money to the correct account
        }
    }

    private bool validateAccount(string accountNumber)
    {
        return true; // Assume Account Number is ok
    }
}

public class Payments
{
    static void Main()
    {
        DataSet dsAccountList = Account.GetAllAccounts();
        Account account = new Account();
        account.DepositMoney((decimal)249.99, "56329SVZ");
    }
}
```

下面是这个例子中的要点：

- GetAllAccounts()是一个静态方法。调用静态方法时，必须先指定类型名称，再指定方法名。在本例中使用的是 Account.GetAllAccounts()。
- DepositMoney()是一个实例方法。只有创建了对象的实例，才能访问实例方法 account.DepositMoney()。DepositMoney 利用隐藏的 this 参数 this.validate

Account(), 调用一个私有的实例方法 validateAccount()。

3.3.2 实例和静态方法的最佳做法

在设计方法类型时, 下面的规则将有助于设计更好的实例和静态方法:

- 在设计类时, 最好不要在类中既建立许多静态方法, 又建立许多实例方法。如果出现这种情况, 应把实例方法和静态方法拆分到两个类中, 使设计更清晰。带有静态方法的类称为组抽象。
- 静态方法在调用时, 必须先指定类型名称。这与 C++不同, C++允许从对象实例中调用静态方法。C#编译器要求先指定类名, 从静态方法的本质和目的来看是一件好事, 这样可以确保程序员的意图总是能从代码中反映出来。
- 如果带有静态方法的类用于多线程的代码中, 就应考虑并发问题。因为每个线程都将访问同一个静态方法, 因而访问同一个静态字段, 如果访问是并发的, 静态数据和共享资源就会出现不稳定。使用实例方法不会出现这个问题, 因为实例在线程之间不共享。
- 如果一个实例方法调用另一个实例成员, 应使用 `this` 关键字, 使代码更清晰。

3.4 参数和参数化方法

实现不接受参数的方法是非常受限制的。大多数方法将以某种方式处理数据, 因此应将数据(或数据的引用指针)传递给方法。传递给方法的变量通常叫做参数。在方法中引用这些参数的值称为参数。接受参数的方法叫做参数化方法。

3.4.1 参数类型

把参数传递给参数化方法通常是一个重要的考虑, 必须考虑要传送的数据类型、数据在方法中如何处理, 以及如何把处理结果传送回来或让客户代码看到这些处理结果。还需要考虑参数在传送前是否需要初始化, 方法需要多少个参数, 参数的数量是否是任意的。

在把参数传送给方法时, 对参数如何传送主要有两个选择: 可以传送参数的实际值(按值传送), 也可以传送参数的一个引用(按引用传送), 按值传送参数是 C#中的默认方式, 也是面向 .NET Framework 的所有语言的默认方式。

1. 按值传送

按值传送参数会把数据的一个物理副本从客户代码(参数)复制到方法(参数)中。客户代码和方法都有保存该数据副本的变量。在调用方法的初期,这个数据是相同的,但方法接着会执行后续的语句,在方法内部改变这个数据的值。对参数值的改变不会影响参数的值,它是独立的。

按值传送参数在方法声明中不需要额外的关键字。下面称为 `passing_by_val.cs` 的例子说明了如何按值传送数据:

```
using System;

namespace parameter_types
{
    public class Calculator
    {
        public static decimal CalculateInterest(decimal balance)
        {
            decimal interest = (decimal)45.11;
            balance = balance + interest;
            return (balance);
        }
    }

    class BatchRun
    {
        static void Main()
        {
            decimal balance = (decimal)4566.54;

            Console.WriteLine("Balance = " + balance);
            Console.WriteLine("Balance + Interest = " +
                Calculator.CalculateInterest(balance));
            Console.WriteLine("Balance = " + balance);
        }
    }
}
```

这个例子计算银行账号中余额的利息。简单的 `CalculateInterest()` 方法带有一个 `balance` 参数,并加上利息。得到下面的结果:

```
Balance = 4566.54
Balance + Interest = 4611.65
Balance = 4566.54
```

注意在调用 `CalculateInterest()` 后，账号的余额没有变化。这是因为参数的值复制到方法参数中。参数和参数的值是彼此独立的。

2. 按引用传送

按引用传送参数是复制数据的引用，而不是复制数据本身。可以把这个过程看作给方法传送一个指针，该指针指向存储数据的内存位置。客户代码和方法存储了同一数据的不同引用。所以，对客户代码来说，方法对参数的修改是可见的。方法所做的修改可能会影响客户代码，这种修改称作“副作用”。参数按引用传送时，可以把参数看作是参数的一个别名。

在按引用传送参数的过程中，把值复制到方法中，就像按值传送那样，当方法执行完后，该值又复制回原来的位置，同时复制了方法在执行过程中对该值的修改。

在按引用传送参数时，需要在方法声明和调用客户代码中同时使用 `ref` 关键字。在调用程序中没有使用 `ref` 关键字，会产生一个编译时错误：

```
Argument 'n': cannot convert from '<type>' to 'ref <type>'.
```

`ref` 关键字重写了默认的按值传送方式。下面的例子说明了如何按引用传送数据：

```
using System;

namespace parameter_types
{
    public class Calculator
    {
        public static decimal CalculateInterest(ref decimal balance)
        {
            decimal interest = (decimal)45.11;
            balance = balance + interest;
            return (balance);
        }
    }

    class BatchRun
    {
        static void Main()
        {
```

```
decimal balance = (decimal)4566.54;

Console.WriteLine("Balance = " + balance);
Console.WriteLine("Balance + Interest = " +
    Calculator.CalculateInterest(ref balance));
Console.WriteLine("Balance = " + balance);
    }
}
}
```

CalculateInterest()方法修改为使用 ref 关键字接受按引用传送的参数。调用客户代码也修改为使用 ref 关键字按引用传送参数。也就是说，参数和参数包含对同一个值的引用。

注意：

使用 ref 关键字按引用传送的参数必须在传送给方法之前初始化。参数未初始化会产生下面的编译期间错误：Use of unassigned local variable <variable name>。

运行程序，就会显示下面的结果：

```
Balance = 4566.54
Balance + Interest = 4611.65
Balance = 4611.65
```

注意，账号余额的修改是调用 CalculateInterest()方法的一个副作用。在本例中，CalculateInterest()方法还返回新的账号余额。这是多余的，因为账号余额会因按引用传送而自动改变。这样就有两个选择：

- 按值传送参数，从方法中获得一个返回值
- 按引用传送参数，把方法声明为 void，没有返回值

这里影响我们选择的最重要因素是：如果需要方法返回几个值，就可以使用按引用传送的方式。一个方法只能返回一个值，而按引用传送的方式可以对所传送的所有参数施加上述的副作用。

如果希望按引用传送参数会改变几个参数，可参阅下一节讨论的输出参数。输出参数有时可以为这个问题提供更好、更合适的解决方案。

3. 输出参数

输出参数解决了下述问题：让方法模拟操作和传送回多个值。与 ref 参数相似，输出参数也是按引用传送的，out 参数也可以看做是参数的一个别名，但它与 ref 参数有一个微妙而重要的区别。

注意:

使用 out 关键字按引用传送的参数不需要在传送给方法之前进行初始化。

其含义是,如果参数没有值(或是一个 null 引用),方法就不能对参数进行任何修改。在方法中,也可以把输出参数看做是未赋值的;因此,带有 out 参数的方法必须在返回之前初始化所有的 out 参数。

下面的例子 parameter_types.cs 说明了如何使用 out 参数按引用传送数据:

```
using System;

public class Calculator
{
    public static decimal CalculateInterest(out decimal balance)
    {
        decimal interest = (decimal)45.11;
        balance = (decimal)4566.54;
        balance += interest;
        return (balance);
    }
}

class BatchRun
{
    static void Main()
    {
        decimal balance;

        Console.WriteLine("Balance + Interest = " +
            Calculator.CalculateInterest(out balance));
    }
}
```

运行这个程序,就会在控制台窗口中显示下面的结果:

```
C:\Class Design\Ch 03>parameter_types
Balance + Interest = 4611.65
```

这个程序的要点是:

- 账号余额(balance)在传送给 CalculateInterest()之前没有经过初始化。
- 输出参数总是假定为未经过初始化; CalculateInterest()必须在返回之前初始化

balance 参数。

- 如果输出参数在传送给方法之前就已经初始化了，方法则必须忽略其初始值，并赋予它自己的值。在本例中，首先给 balance 赋值，然后再加上利息的操作会产生一个编译期错误，因为 balance 必须在方法返回之前初始化。
- 调用程序和方法声明都显式使用了 out 关键字，其方式与 ref 参数一样。

注释：

在 .NET Framework 中，输出参数目前只有 C# 才支持。Visual Basic .NET 不支持输出参数的概念。C# 输出参数的处理规则与其他 .NET 编程语言使用的常规按引用传送参数相同。因此，C# 的 out 参数和 ref 参数与 Visual Basic .NET 的 ByRef 参数相同。

输出参数非常适合于需要返回多个值的方法。因为要把 out 参数看作是未进行初始化的，所以方法在返回之前，必须给所有的 out 参数赋值。这与返回一个值的方法的规则相同：方法必须赋值，并返回正确类型的值。

C# 编译器要求在客户代码和方法签名中都必须使用 out 关键字，清楚地告诉开发人员该参数在方法中的目的。

3.4.2 按引用类型传送与按值类型传送

选择是按引用(ref 或 out)传送参数还是按值传送，不仅要考虑方法的执行方式、方法的副作用、性能和返回值，还要考虑传送给方法的数据类型。

.NET 类型要么是值类型，要么是引用类型。如第 1 章所述，值类型包含基本数据类型、结构和枚举。把一个值类型变量赋予另一个值类型变量，会使这两个变量包含不同的数据。实际上，所有的值类型赋值，包括给方法参数按值传送参数，都会把数据复制到新变量中。值类型的生存周期可以直接由方法或客户代码控制，因为在方法返回时，所有的值类型都超出了其作用域，它们的内存位置也会立即释放。当值类型变量超出其作用域时，变量及其数据都会从栈中删除，并立即释放内存。

引用类型可以是任意对象类型(包括数组和字符串)，它们存储在堆上。引用类型的变量包含引用。这些引用就是在调用方法时按值或按引用传送的值。

引用类型可以有許多指向它的引用。当多个基于栈的变量指向同一个基于堆的引用类型时，就会有多个指向它的引用。引用类型在堆上不会超出其作用域(因此可以由垃圾回收器回收)，直到所有的引用从栈上删除为止。垃圾回收器控制着堆，对象在垃圾回收器把它从堆上删除之前都占用了内存。这些问题将在第 5 章详细论述对象的生存周期时讨论。

要比较传送引用类型和传数值类型之间的区别，还需要重新考虑一下上一节介绍的参数类型。把参数传送给方法实际上有 4 个基本组合：

- 按值传送值类型
- 按引用(ref 或 out)传送值类型
- 按值传送引用类型
- 按引用(ref 或 out)传送引用类型

下一节将详细介绍这些组合。下面是这 4 个组合的具体内容。

注释:

下面几节没有显式使用输出参数(out)，它们实际上是 ref 参数的一种变体。

1. 按值传送值类型

在提及值类型时，首先就会想到数字，例如整数。结构也是值类型，它非常适合于创建复杂的值类型实体，例如表示分数的类型、图中的坐标，或信用卡信息。这些例子通常表示为一组结构化的数字。在处理不能用一个基本类型表示的、但相对简单的实体对象时，常常选择结构而不选择类。

在按值传送一个结构时，就会把该结构的一个副本复制到方法参数中。参数和参数都包含了该结构的独立副本。方法对结构副本的任何修改都不会反映到客户代码中。

按值传送值类型是默认方式，这也是一种合法的默认方式。除非要满足下一节列出的某个条件，否则最好不要改变这种默认方式。

在下面的例子中，演示了如何按值传送值类型。这个例子以表示为结构的 CreditCard 实体为基础，实现 GetBalance()方法，它从数据库中检索信用卡的余额信息。由于信用卡信息非常敏感，GetBalance()方法中不能出现意外修改信用卡信息的错误。下面的例子在源代码文件中命名为 val_by_val.cs:

```
using System;

public struct CreditCard
{
    double cardNumber;
    DateTime expiryDate;

    public double CardNumber
    {
        get
        {
            return cardNumber;
        }
        set
```

```
{
    cardNumber = value;
}

public DateTime ExpiryDate
{
    get
    {
        return expiryDate;
    }
    set
    {
        expiryDate = value;
    }
}

public class Authorization
{
    public static decimal GetBalance(CreditCard creditCard)
    {
        creditCard.ExpiryDate = creditCard.ExpiryDate.AddMonths(12);
        return (decimal)845.21;
    }
}

class Payments
{
    static void Main()
    {
        CreditCard card = new CreditCard();
        Decimal balance;

        card.CardNumber = 1111222333444;
        card.ExpiryDate = Convert.ToDateTime("01/03/2003");

        balance = Authorization.GetBalance(card);

        Console.WriteLine("Card Number - " + card.CardNumber);
    }
}
```



```
        Console.WriteLine("Expiry Date - " +  
            card.ExpiryDate.ToShortDateString());  
        Console.WriteLine("Balance      - " + balance);  
    }  
}
```

运行此程序，在控制台窗口中会显示下面的输出结果：

```
C:\Class Design\Ch 03>val_by_val  
Card Number - 1111222333444  
Expiry Date - 01/03/2003  
Balance      - 845.21
```

注意，在 `GetBalance()` 方法中特意修改了信用卡的终止日期，这不会永久地影响信用卡结构的状态。这是按值传送值类型的一个不错的例子，因为不希望 `GetBalance()` 方法有任何副作用。

2. 按引用传送值类型

在使用 `ref` 关键字传送结构时，会把该结构的引用传送给方法，而不是传送该结构本身的副本。此时，参数和参数在引用方面是等价的。也就是说，方法对参数进行的修改对客户代码来说是可见的。

使用按引用传送值类型的场合如下：

- 参数以简单的形式表示，例如表示为数字或简单的结构。
- 方法需要修改结构，且这些修改要反映到客户代码中。
- 要把几个结构传送给方法，而且方法要修改所有这些结构。这就解决了方法只能给客户代码返回一个值的问题。

为了说明何时使用按引用传送值类型的方式，把下面的 `RenewCard()` 方法添加到 `Authorization` 类中。新文件名为 `val_by_ref.cs`：

```
public static void RenewCard(ref CreditCard creditCard)  
{  
    creditCard.ExpiryDate = creditCard.ExpiryDate.AddMonths(12);  
}
```

下面修改 `Payments` 类：

```
class Payments  
{  
    static void Main()  
    {
```

```
CreditCard card = new CreditCard();
Decimal balance;

card.CardNumber = 1111222333444;
card.ExpiryDate = Convert.ToDateTime("01/03/2003");

balance = Authorization.GetBalance(card);

Console.WriteLine("Card Number - " + card.CardNumber);
Console.WriteLine("Expiry Date - " +
    card.ExpiryDate.ToShortDateString());
Console.WriteLine("Balance      - " + balance);

Authorization.RenewCard(ref card);

Console.WriteLine();
Console.WriteLine("Card Number - " + card.CardNumber);
Console.WriteLine("Expiry Date - " +
    card.ExpiryDate.ToShortDateString());
Console.WriteLine("Balance      - " + balance);
}
}
```

运行程序，在控制台窗口中会显示下面的输出：

```
C:\Class Design\Ch 03>val_by_ref
Card Number - 1111222333444
Expiry Date - 01/03/2003
Balance      - 845.21

Card Number - 1111222333444
Expiry Date - 01/03/2004
Balance      - 845.21
```

注意信用卡的终止日期增加了一年。这是按引用传送值类型的一个好例子，因为 `RenewCard()` 方法的副作用要对客户代码可见。

3. 按值传送引用类型

在按值传送引用类型时，需要传送对象的值。对象的值不是它的一个属性值，也不是由它的一个方法返回的值。方法参数实际上接收对象在堆上存储位置的值。

按值传送类的副作用与按引用传送结构的相同。但 CLR 的作用不同。在按引用传送结构时，该结构从调用的方法中引用。参数和参数包含对同一个结构的引用。而按值传送类实例时，引用复制到调用的方法中。参数和参数包含的是对堆上同一个对象的引用。

CLR 采用的路由可能不同，但效果相同。如果对象是可变的，就可以修改按值传送的对象状态，这些修改对客户代码也是可见的。

按值传送引用类型是默认方式，这是很自然的，在正常情况下不要改变这种方式。

把引用类型的实例传送给方法时，引用仍由客户代码保存。如果把引用的副本放在类变量中，它引用的对象与调用代码的引用所指向的对象相同。在方法返回之后，它们仍引用同一个对象。所以对于引用类型，不仅调用的方法可以修改客户代码传送给它的对象状态，客户代码也可以在把对象传送给方法之后修改对象，且其副作用仍存在。有时这是很理想的，有时则不是。如果希望避免引用类型的这些问题，应使类型不可变。

下面的例子 `ref_by_val.cs` 基于一个雇员实体。雇员表示为一个类，因为雇员实体一般比较复杂。注意这个例子仅使用 `Salary` 属性，占用的空间比较少。接着实现 `Taxation` 类和 `Calculate()` 方法，来计算雇员的工资：

```
using System;

public class Employee
{
    int salary;

    public int Salary
    {
        get
        {
            return salary;
        }
        set
        {
            salary = value;
        }
    }
}

public class Taxation
{
```

```
public static void Calculate(Employee employee)
{
    employee.Salary = employee.Salary - (employee.Salary*22/100);
}

class Calculate
{
    static void Main(string[] args)
    {
        Employee employee = new Employee();

        employee.Salary = 40000;
        Console.WriteLine("Gross salary before tax - {0}",
            employee.Salary);

        Taxation.Calculate(employee);

        Console.WriteLine("Net salary after tax    - {0}",
            employee.Salary);
    }
}
```

运行程序，在控制台窗口中就会显示下面的输出结果：

```
C:\Class Design\Ch 03>ref_by_val
Gross salary before tax - 40000
Net salary after tax    - 31200
```

注意即使 Employee 是按值传送的，工资也会改变。这是因为 Employee 是一个类，所以是引用类型。

在 Calculate()方法中给 Employee 参数临时重新赋值，就可以避免这个行为。新的代码文件是 ref_by_val2.cs。修改 Calculate()方法，如下所示：

```
public class Taxation
{
    public static void Calculate(Employee employee)
    {
        Employee temp_employee = new Employee();

        temp_employee.Salary = employee.Salary;
```

```
temp_employee.Salary = temp_employee.Salary - ((temp_employee.Salary/100)*22);

employee = temp_employee;
}
}
```

运行程序，这次在控制台窗口中应显示下面的输出结果：

```
C:\Class Design\Ch 03>ref_by_val2
Gross salary before tax - 40000
Net salary after tax    - 40000
```

注意在方法调用后，工资保持不变。使用对象的一个临时实例，其类型与参数相同，再复制每个属性，就可以在临时对象上执行相同的计算。即使在方法执行的最后把参数重新赋给临时对象，对 Salary 属性的修改也不会出现在客户代码中。对按值传送的引用类型的重新赋值不是永久性的，客户代码看不到副作用。

4. 按引用传送引用类型

要把一个对象传送给方法，应传送包含对象引用的变量。如果方法参数声明为 ref 或 out，该参数就是一个对对象引用的引用。这样就有了两次引用：方法参数是对调用参数的引用，参数则是对堆上对象的引用。

按引用传送引用类型与按值传送引用类型的区别很微妙，但很重要。与按值传送引用类型不同，按引用传送引用类型允许永久地把参数和参数重新赋予另一个对象实例。这可以提高给修改对象状态的方法传送对象的安全性。参数化的方法可以声明为接收对象作为参数，以便方法创建同一类型的第二个新对象，修改其状态，如果条件允许，参数对象还可以重新赋予第二个对象。

使用按引用传送引用类型的场合如下：

- 参数必须表示为一个类
- 方法需要有条件地修改参数的状态
- 方法需要永久地把参数重新赋予同一类型的新实例

除了允许调用的方法把引用重新赋予同一类型的新实例之外，还可以取消引用的赋值，即把它的值设置为 null。

为了说明这些，按下面的方式修改 Calculate()方法：

```
public class Taxation
{
    public static void Calculate(ref Employee employee)
    {
```

```
Employee temp_employee = new Employee();

temp_employee.Salary = employee.Salary;

temp_employee.Salary = temp_employee.Salary -
    ((temp_employee.Salary*22/100));

employee = temp_employee;
}
}
```

下面修改 Calculate 类:

```
class Calculate
{
    static void Main()
    {
        Employee employee = new Employee();

        employee.Salary = 40000;
        Console.WriteLine("Gross salary before tax - {0}",
            employee.Salary);

        Taxation.Calculate(ref employee);

        Console.WriteLine("Net salary after tax    - {0}",
            employee.Salary);
    }
}
```

运行程序，在控制台窗口中会显示下面的输出结果:

```
C:\Class Design\Ch 03>ref_by_ref
Gross salary before tax - 40000
Net salary after tax    - 31200
```

注意在调用 Calculate() 方法之后，工资改变了。在这个例子中，即使把参数赋予新实例，按引用传送引用类型也总是在客户代码中出现副作用。

3.4.3 可变长度的参数列表

有时，在设计类时，不知道方法应接受多少个参数。在.NET Framework 中，有这种情况的类很多。Console.WriteLine()方法就是一个不错的例子。这个方法接受类型为System.Object 的不定数量的参数，它们用作字符串格式的参数。

一种称为“参数数组”的技术可用于编写能接受不定数量的参数的方法。params 关键字可在方法声明中实现这种行为。下面的文件(var_para.cs)演示了这一点：

注意：

params 关键字后面的参数必须声明为一维数组- []。

```
using System;
class Statistics
{
    public static double AverageScores(params int[] scores)
    {
        int total = 0;
        double average = 0;

        foreach(int score in scores)
        {
            total += score;
        }

        average = total*1.0 / scores.Length;

        return average;
    }
}
```

创建客户代码，执行 AverageScores()方法：

```
public class params_example
{
    public static void Main()
    {
        Console.WriteLine("Average - " +
            Statistics.AverageScores(7, 2, 5, 8, 9, 6, 8, 1));
        Console.WriteLine("Average - " +
```

```
        Statistics.AverageScores(3, 4, 2, 5));  
        Console.ReadLine();  
    }  
}
```

运行这段代码的输出如下：

```
C:\Class Design\Ch 03>var _para  
Average - 5.75  
Average - 3.5
```

CLR 解决参数数组方法调用的方式与一般的解决方案不同，在一般的解决方案中，参数的个数与参数列表是匹配的。当参数的个数与参数个数不匹配时，编译器就会检查方法声明中是否有 `params` 关键字。接着编译器创建一个所声明类型的参数的临时内部栈，直到有足够的参数接受所有的参数为止。另外，数组中的每个元素都是按引用传送的。因此如果操作值，要确保知道是否有副作用。

由于使参数与参数匹配的过程是动态处理的，参数数组必须是方法声明中的最后一个参数。例如，下面的代码就不能编译：

```
public static double AverageScores(params int[] Scores, int PassMark)  
{  
    //  
}
```

注意：

`params` 参数必须是方法的最后一个参数。

使用参数数组会在调用方法之前执行一个额外的小操作：参数需要打包到数组中，该数组必须在堆上分配内存空间，但与参数数组提供的灵活性相比，它在性能上不会造成很大的降低。

3.4.4 传送字符串——不变的对象

`string` 对象是引用类型，是 `System.String` 类的实例。这个类重写了引用类型的一些常见行为，使字符串的操作方式与值类型相似(但不完全相同)。

字符串对象是不变的，也就是说，它们的初始值不能修改。简单的字符串操作似乎改变了字符串的值，但实际上返回了另一个字符串对象，而不是原对象修改后的版本。这在一些大量处理字符串参数的方法(如 XML 和 HTML 建立器)中需要重点考虑，因为这些处理会用尽系统资源。由于字符串是引用类型，修改字符串实际上是在堆上生成

两个字符串，即原字符串和修改后的字符串。字符串变量会在新赋值之后引用新的字符串对象。如果只有原字符串的引用来自于修改后的变量，堆上就会出现一个孤立的字符串，垃圾回收器必须找到该字符串并删除它。显然，如果这个过程进行了许多次，就会出现性能问题。

注意：

大量的字符串操作应使用 `System.Text.StringBuilder` 类来进行，它限制了为表示字符串而必须创建的对象数量。

注释：

清华大学出版社引进并出版的《C#字符串和正则表达式参考手册》介绍了如何使用 `StringBuilder` 类和其他类进行高效的文本处理。

3.4.5 传送数组和上限检查

数组在 .NET Framework 中是引用类型，“按值传送引用类型”和“按引用传送引用类型”两节中介绍的规则也适用于数组，甚至适用于只包含值类型的数组。

对按值传送的数组成员进行的修改对客户代码是可见的，因为可以通过对同一数组的引用来访问它们。但是，按值传送的数组不能重新赋值为另一个数组引用，所以数组的大小保持不变。如果希望从方法中获得 8 个字节的数据，就可以把它按值传送为一个 8 字节数组，让方法填充该数组。在方法处理完后，该数组肯定仍是 8 个字节，但每个元素的内容可能已由调用的方法修改过了。

为了防止客户代码访问方法对数组进行的任何操作，可以在方法中创建一个临时数组。有了这个值类型的数组，就可以对数组进行 `Clone()` 操作，因为这会创建一个新数组，其中包含相同值的新副本。接着把这些值处理为核心内容，而不会影响客户代码的数组。如果数组包含引用类型，这样处理是不行的。`Clone()` 方法执行数组的阴影复制，所以新数组中的引用指向堆上的相同对象，这与参数数组中的引用相同。如果还要复制数组中的元素，就需要做更多的工作，来确保数组的内容也进行了复制。

3.4.6 传送枚举值

枚举提供了把参数传递给方法的极好方式，在这种方式中参数有受限的一组允许值。枚举用来说明在内部表示为数值的数据列表，如验证表和查找列表。枚举还可以看作常量的集合。

调用带有枚举参数的方法时，可充分利用 Visual Studio .NET IDE 的 IntelliSense。

开发人员可以从 `CloseDownOptions` 列表中选择一个有效的项，或传送声明为与 `CloseDownOptions` 相同类型的变量。枚举使代码更清晰更简明，因为它们不需要幻数(硬编码的值)和多个常量值，从而减少了给方法传送不正确值的几率。

3.5 方法重载

这是一个基本的面向对象设计原则，方法名称应该反映方法实际所做的工作。通常，几个方法需要做相似的事情，但却接收不同的参数。在一些编程语言中，使用复杂的方法名称，如 `PrintForm`、`PrintReport` 和 `Print` 方法来处理这种情况。在面向 .NET Framework 的语言，如 C# 中，可以把所有这些方法都叫做 `Print()`，因为这是它们完成的工作。

重载在 .NET Framework 基类库中使用很广泛，在 Visual Studio.NET 中编写方法调用时，可以在所显示的 IntelliSense 信息中识别重载。方法的第一个版本用“1 of n”显示，后跟方法的签名。微调按钮控件或上/下箭头键可以用来导航至方法的每一个版本。

重载在实用程序类型的类中非常有用，每个方法都可以实现两个(或更多)版本。例如，有一个类可以抽象数据访问组件。这个类可以使用 `GetCustomer()` 方法从数据库中检索客户。这个方法两个版本会被编写为分别接受一个 `int` 值的客户 ID，和一个嵌入到 XML 节点中的用户 ID。

重载方法允许指定同一种方法的不同版本，这些方法有相同的名称，仅由方法的签名来区别。方法的签名(有时称为原型)是名称和参数列表的组合。

下面的方法 `DisplayCalculatedCost()` 接受两个参数：一个 `Order` 对象和一个 `Double`：

```
private void DisplayCalculatedCost(Order order, double orderCost)
{
    // Display the details
}
```

它有如下签名：

```
DisplayCalculatedCost(Order, Double)
```

而 `CalculateCost()` 方法：

```
private double CalculateCost(double orderCost, double deliveryFee)
{
    // Calculate order cost
}
```

有如下签名:

```
CalculateCost(Double, Double)
```

可以看到, 返回类型并不是方法签名的一部分。这定义了一条重载规则:

两个或更多方法如果只是返回类型不同, 它们之间不能互相重载。

再看一下由 `params` 参数类型产生的签名。前面讨论过, 编译器为函数的每一个可能输出创建一个签名引用。在参数数组中, 这个列表应该是无限长的。如下所示:

```
public static int AverageScores(params int[] scores)
{
    int total = 0;
    int average = 0;

    foreach(int score in scores)
    {
        total += score;
    }

    average = total / scores.Length;

    return average;
}
```

产生如下签名:

```
AverageScores(ref int)
AverageScores(ref int, ref int)
AverageScores(ref int, ref int, ref int)
...and so on
```

重载方法时, 要应用许多规则:

- 每一个重载必须有如下所示的一个(或多个)不同之处:
 - 参数数量
 - 参数数据类型
 - 参数顺序
- 每一个方法应该使用相同的方法名称, 否则, 就是创建一个全新的方法。
- 不能重载带有相同属性名称的方法, 反之亦然。

只修改参数的顺序, 就能重载方法, 如下所示(`argument_order.cs`):

```
using System;

public class MethodOverload
{
    public int Add(int number_one, double number_two)
    {
        return (int)(number_one + number_two);
    }

    public int Add(double number_two, int number_one)
    {
        return (int)(number_two - number_one);
    }
}

public class Test
{
    public static void Main(string[] args)
    {
        MethodOverload overload_meth = new MethodOverload();

        Console.WriteLine("Add (int, double) "+ overload_meth.Add(4,3.2));
        Console.WriteLine("Add (double, int) "+ overload_meth.Add(4.2,3));
    }
}
```

这里定义了两个 Add()方法，这两个方法都有两个参数，一个 int 和一个 double。惟一不同的是参数传递的顺序。

- Add(int, double)
- Add(double, int)

第一个方法用来增加数量，而第二个方法减去数量：

```
C:\ClassDesign\Chapter03>argument_order
Add (int, double) 7
Add (double, int) 1
```

只要变换传递参数的顺序，就可以调用不同的方法。我们可能永远不会进行这么糟糕的设计，但它说明了只有参数顺序不同的重载方法通常是很糟糕的。它会使代码难以阅读，使调用这个方法的人感觉困惑。

下面的例子用一种更有效的方式来重载函数：

```
private double CalculateCost(double orderCost)
{
    // Calculate total cost
    return (this.CalculateCost(orderCost, 0, 0));
}

private double CalculateCost(double orderCost, double deliveryFee)
{
    // Calculate total cost
    return (this.CalculateCost(orderCost, deliveryFee, 0));
}

private double CalculateCost(double orderCost, double deliveryFee,
                             double handlingFee)
{
    // Calculate total cost
    return (orderCost + deliveryFee + handlingFee);
}
```

这些方法有下面的签名：

- CalculateCost(Double)
- CalculateCost(Double, Double)
- CalculateCost(Double, Double, Double)

这些方法计算一些货物的价值、可选的运输费用，以及手续费。注意使用 **this** 关键字可以识别出所调用的重载方法的本地版本。前两个方法调用第三个重载方法，并传送相应数量的 0。

重载方法有一些优点。方法重载应用于提供完成几乎相同的任务的不同方法。只要签名不同，就可以在一个类型中定义同名的多个方法。应仔细考虑与创建两个仅签名不同的两个方法相比，给两个方法指定不同的名称是否会使类接口更简明清晰。要使用前面的 **Add()** 例子，可以利用这个方法的几个版本，它们都采用不同类型的成员作为参数，返回传入的两个(或更多)值的总和。但是如果创建一个方法，它连接字符串，或组合非数字值，或对数字做一些数学运算，而不是把它们加在一起，就不应给它们指定相同的名称。

3.6 异常的处理

在设计方法时，总是要有预防措施。也就是说，必须假定方法在运行过程中会遇到

问题，迫使方法停止运行。类的设计人员或开发人员应以适当的方式处理这些异常。这是因为异常的处理是类和其客户之间的契约中的一个主要部分。异常处理是指或者在内部处理异常，并采取相应的措施，或者捕获异常，给客户代码抛出一个更有意义的异常。如果方法缺乏异常处理机制，在方法执行过程中抛出的任何异常都会传送给客户代码。不处理异常，就是把处理异常的责任交给调用方法的代码来承担。

未预料到方法中可能抛出异常，会在开发客户代码时增加额外的工作量。代码中没有异常处理机制，客户代码本身就必须参与异常的处理，再次猜测调用方法时什么地方出了错。这违反了面向对象编程的第一个规则：封装。方法只应抛出已注释过的异常，让客户代码知道会发生什么。

例如，考虑一个存储客户信息的方法。它对数据库进行更新时要存储客户的信息。在这个过程中会有许多问题，例如数据库可能不可用，或违反了外键约束。如果以后要修改这个方法的实现代码，让它能调用 Web 服务，就会遇到另一个类型的问题：网络失败或 XML 解析错误等。如果把这些问题的责任交给客户代码来处理，并假定只需要处理数据库异常，客户代码就不能处理这些新的异常。结果是，让异常渗透到方法中，会展示类的内部工作情况，违背了封装的初衷。如果出了问题，调用程序一般不知道异常的深层次原因。在这个例子中，客户代码只需要知道没有存储客户信息。接着采取相应的措施来处理这个问题。没有异常处理机制的应用程序常常会在用户的屏幕上显示一些无意义的错误消息，更糟糕的是，应用程序会突然崩溃，而不说明原因。

3.6.1 什么是异常

方法在遇到未预料到的问题或处理错误时，就会产生异常，这将使方法的执行永久性终止。例如“文件未找到”或“除数为 0”的异常。一些编程语言会生成无意义的错误数字，而 .NET CLR 则抛出一个异常，这是一种特殊类型的对象。

异常是一个从 `System.Exception` 最终派生出来的对象，`System.Exception` 是所有异常类的基类，提供了所有异常类都支持的核心功能。虽然 `System.Exception` 是异常的超类，但是异常还有两个主要的子类，一个是派生所有 .NET Framework Class Library 异常的 `System.SystemException`，另一个是派生所有用户定义的应用程序异常的 `System.ApplicationException`。

`System.Exception` 有许多有用的只读属性，所有的异常类都共享它们。这些属性都可以检查和报告，改进了调试过程：

- **HelpLink:**

`HelpLink` 属性设置或获取链接了相关帮助文件的 URN 或 URL。在应用程序的用户界面需要给用户显示有意义的其他信息时，就可以使用这个属性。

- **InnerException:**

这个属性可以用于创建嵌套的异常。当处理原异常时生成了另一个异常类，就可以使用这个属性。原异常可以嵌入 `InnerException` 中，创建一个嵌套的异常。

- **Message:**

`Message` 属性用于包含有意义的错误消息，以描述异常的原因。

- **Source:**

`Source` 属性保存了导致异常的对象或应用程序的名称。如果没有设置这个属性，就使用程序集的名称。

- **StackTrace:**

这个属性返回调用栈的字符串表示。它可以用于跟踪导致异常的方法调用的序列。

- **TargetSite:**

当 `StackTrace` 属性不是空引用时，这个属性包含异常的 `System.Reflection.MethodBase`。

这些属性构成了所有异常类的基础，而专业化的异常扩展了这个属性列表，包含专用于某异常类的信息。例如，`System.ArgumentException` 类还包含 `ParamName` 属性，该属性用于表示给方法传送无效参数时的参数名。

在抛出异常时，方法会永久地终止执行。方法不会给客户代码返回一个值，可能会改变或不改变按引用传送的参数。异常的源代码会立即查找异常处理程序。根据异常的类型是否匹配该类型的异常处理程序，就可以找到合适的异常处理程序。

3.6.2 Try...Catch...Finally

`try`、`catch` 和 `finally` 语句块构成了结构化异常处理机制的基础。具体做法是把方法组织为如下形式：

- 可能抛出异常的代码(`try` 块)
- 处理异常的代码(`catch` 块)
- 执行清理工作或释放系统资源的代码(`finally` 块)

`try` 块用于分离出测试异常的代码块。用于测试异常的代码包装在 `try {}` 语句中，`try` 块的后面必须跟有 `catch` 或 `finally` 块，或者同时跟有两个代码块。

`catch` 块用于捕获特定的异常类型。尽管 `catch` 块并不是必需的，但 `try` 块后面通常至少跟有一个 `catch` 块。要捕获不同类型的异常，可以使用多个 `catch` 块。

如果存在 `finally` 块，无论是否抛出异常或处理异常，都要执行这个块。`finally` 块也不是必需的，除非没有 `catch` 块。但是，由于 `finally` 块总是在 `try` 和 `catch` 块之后执行，因此它们应执行必要的清理工作，完成异常的处理。这包括释放应用程序资源，例如数据库连接等。`finally` 块非常有用，因为即使抛出了异常，它们也总是会执行。因此，`finally` 块总是要使用的。下面是使用 `try`、`catch` 和 `finally` 块的一些规则：

- 在需要捕获异常的代码处使用 `try` 块。

- 只捕获特定类和方法负责的那些异常。其他异常则传送回客户代码。
- 在应用程序的根(通常是用户界面)包含一个捕获 `System.Exception` 的通用异常处理程序, 它可以用于处理来自应用程序较低层的异常。
- 捕获异常应从最特殊的异常开始, 接着是较一般的异常处理程序。`catch` 块根据异常的顺序确定优先级, 如果其优先级低于通用的异常处理程序, 就会忽略正确的异常处理程序。
- 在应用程序分为几个逻辑和物理层时, 不要把从较低层传递的所有异常都显示到用户界面上。在这样的环境中, 恢复相应异常处理程序的栈跟踪所需要的性能开销会非常大。如果使用了 `finally` 代码块, 问题也会比较复杂。
- 在无论是否抛出了异常都必须执行的代码处使用 `finally` 块。这包括释放系统资源的代码。

3.6.3 抛出自己代码中的异常

使用 `throw` 语句可以在自己的代码中抛出异常。方法中的处理代码遇到不能忽略的终止问题, 例如传送给方法的无效参数时, 就可以抛出自己的异常。

```
throw (new System.ArgumentException("Error Message",  
                                     "Parameter Name"));
```

在抛出新的异常时, 通常要在一个异常类构造函数中指定异常的信息。`System.ArgumentException` 允许指定错误消息和参数名称。这些信息可以通过 `System.ArgumentException` 类的属性生成。这些在本节的前面已经讨论过了, 但从这个例子中可以看出, `ParamName` 属性添加到 `System.ArgumentException` 类中, 给调用程序提供了其他信息。

如果客户代码提供了无效的参数, 就会抛出一个异常。客户代码负责确保传送有效的数据, 如果没有做到这一点, 方法就不会正常执行。此时应抛出异常, 再传送给客户代码。

从类中抛出异常时, 要考虑该异常对客户代码是否有意义。如果客户代码希望调用一个简单的方法, 把两个数字加在一起, 就不应抛出 `FileNotFoundException` 异常, 因为代码记录了所有的相加操作, 且日志文件找不到了。客户代码不希望出现这个异常, 又不能对它进行任何处理。此时, 应在代码中捕获 `FileNotFoundException` 异常, 再抛出另一个对调用程序有意义的异常。

`System.ArgumentException` 内置于 .NET Framework 类库, 您也可以创建自己的异常类, 方法是创建一个派生于已有异常类的类。由于已有的异常类都隐式或显式从 `System.Exception` 派生出来, 所以我们自己的异常类也从它派生出来。因此, 该异常类

必定支持 `System.Exception` 提供的最小属性集；其中的一些属性可以重写，也可以添加新的属性，专业化自己的异常类。

根据异常类表示的错误类型，通常最好从 `System.ApplicationException` 中派生自己的异常。异常也可以建立在已有 .NET Framework 异常的基础之上，但如果 `catch` 块不够清楚或顺序有错误，这么做就意味着无法区分自己的异常。

要创建自己的异常类，可执行下面的步骤：

- 创建从已有异常类型继承的类，`System.ApplicationException` 通常是最佳的选择。
- 自己的异常应支持所有异常类中都有的 3 个常见构造函数。这些构造函数分别设置 `Message` 和 `InnerException` 属性、自己的 `Message` 属性，和一个不设置属性的默认构造函数。如果自己的异常类派生于不是 `System.ApplicationException` 的另一个异常类，就需要提供其他构造函数。
- 在这 3 个构造函数中，调用对应的基类构造函数，并传送 `Message` 和/或 `InnerException` 参数，或默认的构造函数。
- 包含把值添加到异常类型的其他属性。这些属性应是只读型的，因为它们的值只能通过构造函数来设置。
- 包含允许设置其他属性值的其他构造函数。
- 通过指定 `[Serializable]` 属性，使异常类可以串行化。这将允许类跨 `AppDomain` 和 CPU 处理范围抛出异常。

下面的例子是银行账号的异常类，代码文件是 `InsufficientFundsException.cs`，它编译为 `.dll` 文件。如果取款时账号中没有足够的金额，就会抛出该异常。

```
[Serializable]
public class InsufficientFundsException : System.ApplicationException
{
    private decimal balance;

    public InsufficientFundsException() : base()
    {
    }

    public InsufficientFundsException(string message) : base(message)
    {
    }

    public InsufficientFundsException(string message,
        Exception innerException) : base(message, innerException)
    {
    }
```

```
    }

    public InsufficientFundsException(string message,
        Exception innerException, decimal balance)
        : base(message, innerException)
    {
        this.balance = balance;
    }

    public InsufficientFundsException(string message, decimal balance)
        : base(message)
    {
        this.balance = balance;
    }

    public decimal Balance
    {
        get { return this.balance; }
    }
}
```

除了遵循创建自己的异常类所需要的必要步骤之外，还有很多很好的实例可仿效。在创建自己的异常类时，应遵循下面的规则：

- 在异常类的后面加上后缀 `Exception`，在本例中就是 `InsufficientFundsException`，这将有助于把异常类与其他类区分开来。
- 在描述性的命名空间中创建自己的异常类，在该命名空间中应包含公司名和应用程序名。这将允许在必要时完整地修饰自定义异常类，有助于把内置的 .NET Framework Class Library 异常类和自己的应用程序异常类区分。这也适合于在应用程序中按功能区域组合异常类。
- 仔细考虑自己的异常类派生于哪个已有的异常基类。所有的异常类都隐式或显式派生于基类 `System.Exception`。最好的选择是显式派生于 `System.ApplicationException`，除非需要专业化已有的异常类型。
- 自定义异常类可以重写基类的一些成员，例如 `Message` 属性，也可以通过提供其他的属性和相应的构造函数，来扩展基类。在本例中，把当前账号 `Balance` 作为一个属性包括，它通过返回所抛出异常的其他原因，把值添加到 `InsufficientFundsException` 类中。
- 在捕获异常并抛出另一个异常时，原来的异常应嵌套在 `InnerException` 属性中，该属性返回对原异常类型的一个引用。这个技术用于在需要时为客户代码提供额外的信息。

- 不要为想到的每个异常创建异常类。应考虑为应用程序的功能区域创建一般化的异常，并使用有意义的错误消息。最好首先生成一个可用于应用程序中每个逻辑层的异常类型。
- 不要使创建的异常层次结构过于复杂。至多创建两层或三层异常结构。这将简化 catch 块，减少 catch 块的顺序出现错误的可能性。
- 对于传送回给用户界面的异常类，应赋予 HelpLink 属性，该属性可用于超链接到一个 HTML 帮助文件，提供更多的帮助。

3.7 方法和 MSIL 代码

本章的最后一个主题是讨论 MSIL 代码。MSIL 是所有面向 .NET Framework 的编译器生成的低级编程指令集。

查看一下 C#编译器在本章生成的 MSIL 代码是很重要的，可以看出不同类型的方法如何编译为 MSIL 代码，以及其他章节中介绍的 MSIL 方法的重要性。这将改变考虑所生成的、精致、有时比较复杂的面向对象的 C#代码的方式。

看一下简单的方法如何编译为 MSIL 代码，应创建下面的程序，它包含主要类型的方法示例。这个程序也可以在 Notepad 中创建，保存在源代码文件 msil.cs 中：

```
using System;

class MSILMethods
{
    static void Main()
    {
        int result = 0;
        int total;
        int net = 0;

        MethodExamples examples = new MethodExamples();

        result = examples.ByValueMethod(7);
        examples.ByRefMethod(ref result);
        examples.OutputMethod(out total, result);
        result = result + examples.OverloadMethod(total);
        total = examples.OverloadMethod(result, total);
        net = examples.ParamsMethod(result, total);
    }
}
```

```
        Console.WriteLine(net);
    }
}

class MethodExamples
{
    public int ByValueMethod(int a)
    {
        return a++;
    }

    public void ByRefMethod(ref int b)
    {
        b = b * 2;
    }

    public void OutputMethod(out int c, int d)
    {
        c = d / 4;
    }

    public int OverloadMethod(int e)
    {
        return e + 2;
    }

    public int OverloadMethod(int e, int f)
    {
        return e + f;
    }

    public int ParamsMethod(params int[] g)
    {
        int total = 0;

        foreach(int num in g)
        {
            total = total + num;
        }
    }
}
```

```
        return total + 1;  
    }  
}
```

编译 msil.cs 并检查实用工具 ildasm 检验 C#编译器生成的 MSIL 代码:

C:\Class Design\Ch03>ildasm msil.exe

一旦 ildasm 加载, 首先扩展 MSIL 方法节点, 选择 msil.exe, 单击 View, 再单击 Show Token Values。接着扩展 msil 节点, 就会显示如图 3-1 所示的屏幕, 其中显示了 MSILMethods 类中的所有成员。

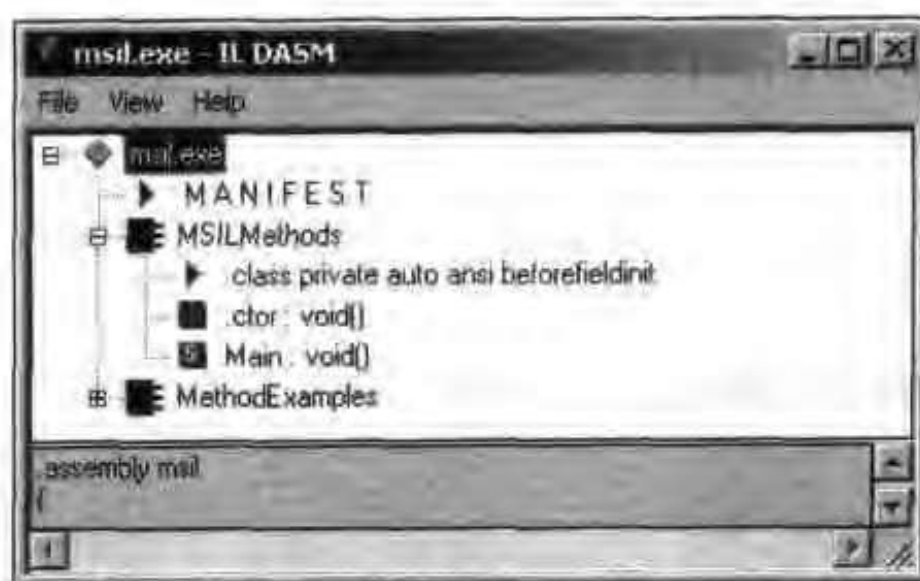


图 3-1

双击 .ctor : void(), 显示的屏幕如图 3-2 所示。

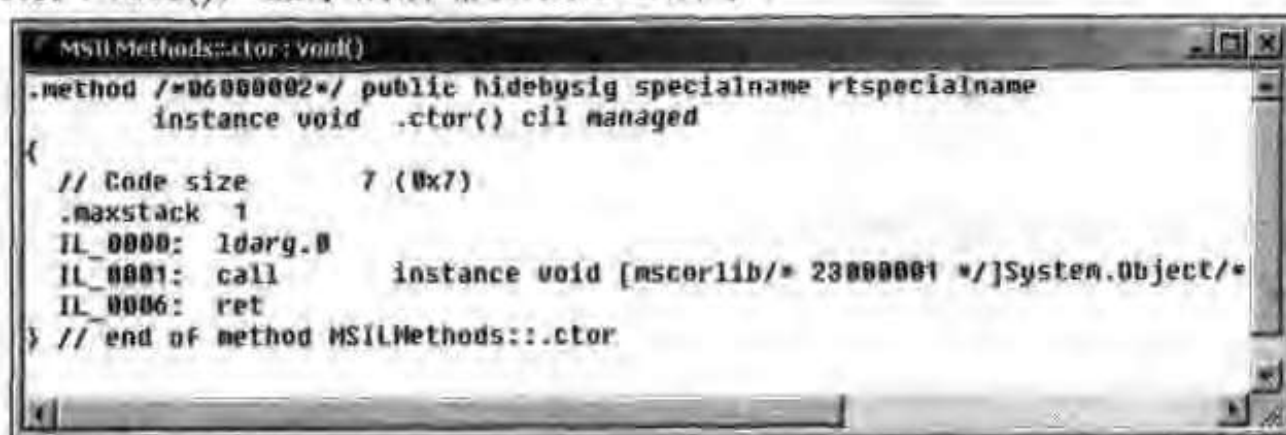


图 3-2

从 C#编译器生成的 MSIL 代码中, 可以注意到 MSILMethods 类的隐式构造函数有如下要点:

- 对于 MSILMethods 类, 这个构造函数实际上是 .method 声明中的一个方法。构造函数与方法一样也编译为 MSIL 代码。这是很重要的, 因为在考虑构造函数的设计 and 使用时, 构造函数就是方法, 只是在类设计时方法要根据其目的变得更加专业化。
- 构造函数方法(.ctor)调用 System.Object 构造函数, 因此可以看出类型的基类。

在原来的.cs 文件中, MSILMethods 类没有显式派生于任何类型, 但隐式派生于 System.Object。这就是调用 System.Object 构造函数的原因。

关闭.ctor 窗口, 双击 Main()方法节点, 将显示如图 3-3 所示的窗口。



图 3-3

在 Main()方法中, 注意下面的要点:

- 从 MSIL 代码中可以看出, Main()通过方法声明中的 cil 托管标记, 作为.NET 托管代码实现。
- 因为创建了一个控制台应用程序, 所以 Main()也是程序的入口。这可以从.entrypoint 指令中看出来。这是把.exe 文件和.dll 文件区分开来的方式。所有的.exe 文件都有一个带有.entrypoint 指令的方法。
- 在“方法的调用”一节中, 讨论了如何创建栈框架, 为参数和局部变量分配栈上的一个固定内存块。.maxstack 指令确定了执行这个方法所需要的基于栈的最多变量数。在本例中, 有 4 个基于栈的变量 total、result、net (它们的类型都是 int), 和 example(这是一个引用指针, 指向类型为 MethodExamples 的基于栈的对象)。栈框架的大小由 C#编译器计算, 并传送到 MSIL 代码中。.maxstack 指令的值进行了优化, 允许在变量不再需要或超出其作用域时, 重复使用栈框架的某些部分。.maxstack 的优化计算基于执行方法所需要的栈的最大空间, 它不是方法中必需的变量个数。优化计算取决于许多因素, 包括变量类型(int、bool、引用等), 传送给其他方法的最多参数数和变量的作用域。

接着, 关闭 Main()窗口, 扩展 MethodExamples 节点。这个节点包含 MethodExamples 类的所有成员, 如图 3-4 所示。



图 3-4

双击第一个 OverloadMethod 成员，就会显示如图 3-5 所示的屏幕。



图 3-5

在.method 指令和 public 访问修饰符之间，有一个标记 06000007。标记在程序集中唯一地标识这个成员。现在关闭这个窗口，双击 MethodExamples 的第二个 OverloadMethod 成员，显示的屏幕如图 3-6 所示。



图 3-6

在这个屏幕中，OverloadMethod()的第二个重载有一个不同的标记 06000006。因为标记在程序集中是惟一的，所以这表示重载的方法实际上编译为完全独立的方法，标记用于解析要调用哪个重载方法。通过重新查看 MSILMethods 类的 Main()方法，就可以证明这个解析过程是在编译期间进行的。见图 3-7。



图 3-7

可能需要滚动窗口，才能看到 Main()调用的标记。

检查 C#编译器生成的 MSIL 代码，可以推断出下面的结论：

- 类成员，包括同名的重载成员，在 MSIL 代码中都实现为独立的成员。C#编译器通过比较方法签名和参数列表，来解析重载的成员。
- C#编译器计算调用方法所需要的每个栈框架的大小。这个计算过程优化为使用最少的内存空间。

- 构造函数也实现为方法。属性和运算符也是这样，对它们的讨论详见下一章。

3.8 设计小结

概况本章，在设计方法时要注意一些问题：

- 按引用传递引用类型(ref)允许重新给对象赋值。方法参数实际上是一个双重指针，首先指向客户代码变量引用，然后指向对象本身。
- 使用值的枚举列表作为方法参数，而不是整型幻数，有助于使代码更清晰，并帮助防止 bug。枚举列表在 IntelliSense 中显示，允许程序员正确地选择适当的值。在评估方法内部的方法参数时，还将提供一定程度的安全性。
- 设法标准化方法的重载。在开发客户代码时，重载的方法可能容易混淆，因为同一个方法有好几个实现代码，每个实现代码接受一组不同的参数。当这些方法都返回不同的值类型时，就可能出现混淆。用这种方式编写的重载方法应该编写为单独的方法，因为它们都有不同的目的。如果方法在语义上是相关的，应使用重载。方法的每个重载都应根据一组不同的参数达到相同的目的。
- 在调用静态方法时，要引用类的名称而不是对象实例变量。静态方法属于类本身，可以由不同类类型的其他对象来使用，不需要实例化一个实际的对象。
- 异常是类的契约的一部分。只处理方法所负责的那些异常，其他异常则自动传送回客户代码。

第4章 属性和运算符

本章将介绍属性和运算符在 C# 类设计中的作用。我们将用许多例子来说明属性如何提高类型的封装性、使用性和健壮性。本章还将介绍索引器，这是一种特殊的属性。讨论运算符重载如何通过允许把自己的类型和算术或布尔运算符结合使用(就像基本类型那样)，以简化应用程序的开发。运算符的显式和隐式转换详见第 7 章。

学习完本章后，读者就会理解如何创建有意义的属性，何时和如何为类和结构实现运算符，还会理解属性和运算符之间的联系。在开始阅读本章的内容之前，最好看一下上一章有关“方法”的内容。本章将使用上一章介绍的许多技术。

4.1 C#中的属性

属性(Property)表示类的特性(attribute)，这与方法不同，方法表示类的动作。例如，Person 类有特性 Name 和 Salary。Person 类还有 Update 和 CalculateBonus 这样的动作。在编写实现代码时，必须确定如何表示 Name 和 Salary 特性。一种方法是把 Name 和 Salary 定义为 Person 类中的私有字段，但这将阻止在程序的其他地方访问信息。把 Name 和 Salary 声明为公共字段也不好，因为这破坏了类的封装，而且直接展示数据，这样，程序的其他部分能读取和修改它们，但为数据的隐藏带来了许多问题。属性则解决了这个问题，它通过 get 和 set 访问器提供了对私有变量的间接访问。

属性访问器只是允许有条件地抽象访问底层数据的方法。在其他语言，如 Visual Basic .NET 中，属性的 get 和 set 访问器有时称为属性过程。

注意：

属性表示类的特性。它们通过 get 和 set 访问器提供了对私有变量的间接访问。

从使用类的程序员的角度来看，属性可以像类中的公共字段那样访问。类的用户可以直接访问属性名，就好像类真的包含该名称的公共字段一样。惟一的区别是可以把属性设置为只读或只写，而这对于公共字段是不可能的。

属性允许把数据展示给客户代码，同时维护了数据在类中的表示方式的封装性。例如，可以在 set 访问器中编写有效性检查代码，确保用户不会把不合法的值赋予属性。同样，还可以在 get 访问器中执行计算，根据请求重新计算属性的值。在讨论 C# 编译器为属性生成的 MSIL 代码时将讨论访问器，现在应知道访问器是类的一个入口，就像

方法一样。

4.1.1 C#属性的语法

属性与方法一样，也是在类中实现的。第3章讨论的 `public` 和 `private` 访问修饰符也可应用于属性，其规则也相同。属性也可以是实例和静态类成员，这些区别稍后讨论。

首先定义属性在 C# 中的实现方式。下面的例子是一个标量属性，但 `get` 和 `set` 访问器的原则与索引器相同：

```
public class Person
{
    private string name;

    public string Name
    {
        get { return name; }
        set { name = value; }
    }
}
```

在这个例子中，有一个私有字段 `name`。可以通过 `Name` 属性的 `get` 和 `set` 访问器来访问此字段。属性可以忽略 `get` 或 `set` 访问器，但不能两个都忽略：

- 如果省略 `set` 访问器，属性就是只读的。这对每次使用时都要重新计算的属性来说很有用，如雇员在公司工作的天数。
- 如果省略 `get` 访问器，属性就是只写的。这对安全对象上的密码属性很有用。使属性可读并不是不好(我们永远不知道什么时候有人需要查询它)，但这种情况并不普遍。

Java 用户注意：

Java 属性也定义为 `get` 和 `set` 方法，但它们必须作为 `get` 和 `set` 方法被访问。

.NET Framework 中的 CLS 支持两种不同的属性：

- 标量属性：

标量属性表示单个类特性。这个属性可以是一个基本值，如 `int`，或引用类型，如 `string`，或较复杂的类型，如 `DateTime`、`Color` 或 `BankAccount` 类。与 Visual Basic .NET 不同的是，C# 标量属性不能参数化。但是 C# 和 Visual Basic .NET 都以索引器的形式提供更好的解决方案。

- 索引器(索引的属性)

索引器表示一个结构,它允许在类上使用像数组那样的语法。客户代码使用数组语法来访问私有集合或数组里的特定值。

本章将详细介绍这两种属性。在讨论时,注意属性是.NET Framework 中的一个标准特征。这意味着只要在 C#类中定义的属性与 CLS 兼容,用任何 CLS 兼容的语言编写的客户代码都可以使用它。

4.1.2 标量属性

首先介绍标量属性。下面的例子占据了几页的篇幅,显示了 `Person` 类中一个简单的读/写属性 `Name`。其中的语法进行了一些压缩,以说明属性如何为编写访问器代码提供一种简洁的方式。在列出代码之后将解释这个属性如何使用。这个例子的源代码位于文件 `simple_scalar_property.cs` 中:

```
public class Person
{
    private string name;

    public string Name
    {
        get { return name; }
        set { name = value; }
    }
}
```

注意这个例子中的下述要点:

- `Person` 类有一个字段 `name`,它保存人名。这个字段声明为私有,来阻止客户代码的直接访问。面向对象开发的最重要目的之一就是保持类的封装。
- `Person` 类有一个属性 `Name`,用来获取和设置人名。这个属性作为 `name` 字段的包装器。`Name` 属性声明为 `public`,用在客户代码中。大多数属性都是 `public`,因为属性的本质就是为类提供一个方便的公有接口。不过,有时也需要 `private`、`protected` 或 `internal` 属性。例如,可以定义一个只能由同一个类中的其他成员访问的 `private` 属性。`Name` 属性的类型是 `string`,`get` 访问器还必须返回 `string` 类型的值。底层的 `private` 字段也是这个类型,如本例所示。
- `set` 访问器包含一个隐藏的参数 `value`,该参数包含从客户代码传送过来的值。

注意:

在定义属性时, 应在访问修饰符和属性名之间包含类型, 指定属性的类型。在本例中, Name 属性是 string 类型, 但也可以是其他可用的类型。

重要的是设计出能区别类中字段和属性的通用命名模式。这将使代码更一目了然, 更易于编写代码, 减少故障数量。而且, 代码将更容易维护, 因为维护程序员更清楚我们的目的。

在 C# 中, 定义属性和字段的命名模式进行了简化, 因为 C# 是区分大小写的。换言之, name 与 Name 是不同的。以多种方式表示数据常常是必不可少的, 对于属性就更是如此, 一般情况下, 字段是在内部使用, 而公共属性用于外部。开发人员需要使用有意义的字段和属性名称来表示相同的含义。公共属性及其底层类型最好使用相同的名称, 因为它们之间的联系将很清晰。

在本章中, 字段使用 camel Case (xxXxx), 例如 name 或 dateOfBirth, 而属性使用 Pascal Case (XxXxx), 例如 Name 或 DateOfBirth。一些开发人员喜欢在字段的开头使用下划线(例如 _Name)。属性也应使用名词来命名。

选择什么命名约定没有关系, 只要在类的内部和所有类中都使用这个约定就行。

下面的代码片段显示了如何在 Person 类中使用 Name 属性:

```
public class SimpleScalarProperty
{
    static void Main()
    {
        Person person = new Person();
        person.Name = "James";

        Console.WriteLine(person.Name);
    }
}
```

在 Main() 方法中注意以下几点:

- 客户代码不能直接访问 name 字段, 对这个数据的所有访问都必须通过 Name 属性间接进行。这意味着 Person 类的开发人员保留了对数据如何在类内部私有实现的控制。这称为封装。
- 客户代码使用类似字段的语法来访问属性。这比方法调用语法更方便。例如, 如果 Person 定义了一对方法 GetName() 和 SetName()(类似于 Java), 客户代码就必须使用更麻烦的语法来获取和设置个人的名字:

```
person.SetName("James")
```

```
Console.WriteLine(person.GetName());
```

了解了如何定义和使用简单属性后,下面讨论在类中正确使用属性的设计和实现问题。首先从 C#编译器生成的 MSIL 代码开始。

1. 把标量属性编译为 MSIL

第3章讨论了 MSIL 代码的作用,理解了不同类型的方法如何编译为 MSIL 代码。在编译属性时,C#编译器生成的 MSIL 代码也为属性生成了 MSIL .method 例程。这是因为属性是特殊类型的方法,这是它们编译为 MSIL 代码的方式。

在类中定义一个读/写属性时,C#编译器会在 MSIL 代码中生成一对方法。例如,如果属性叫做 Name,编译器在 MSIL 代码中生成的方法就叫做 get_Name()和 set_Name()。只要在客户代码中使用这些属性,编译器就隐式地生成代码,来调用 get_Name()和 set_Name() MSIL 方法。

可以使用 MSIL Disassembler 工具 ildasm 来演示这个过程。见图 4-1。



图 4-1

注意 Person 类的下述成员见表 4-1。

表 4-1

成员名称	说 明
m_name	这是一个 private string 字段,用来保存人名
get_Name	这个方法由编译器生成,获取 string 类型的 Name 属性
set_Name	这个方法由编译器生成,把 Name 属性设置为 string
Name	这是 Name 属性本身,它包含 get 和 set 方法

在图 4-1 所示的屏幕上双击 Name 属性,就会出现图 4-2 所示的窗口,其中说明了 Name 属性在 MSIL 代码中如何实现。

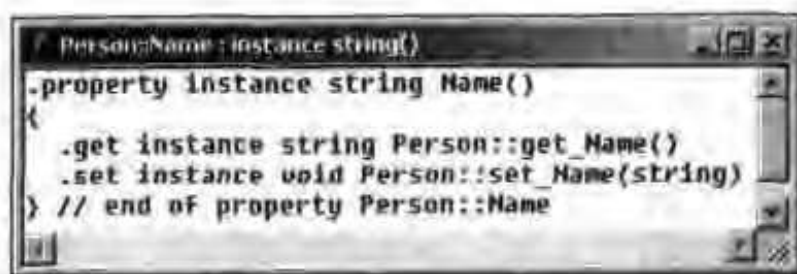


图 4-2

注意, Name 属性包含识别 get 和 set 方法的两个 MSIL 语句。下面是为 Main() 方法生成的 MSIL 见图 4-3。

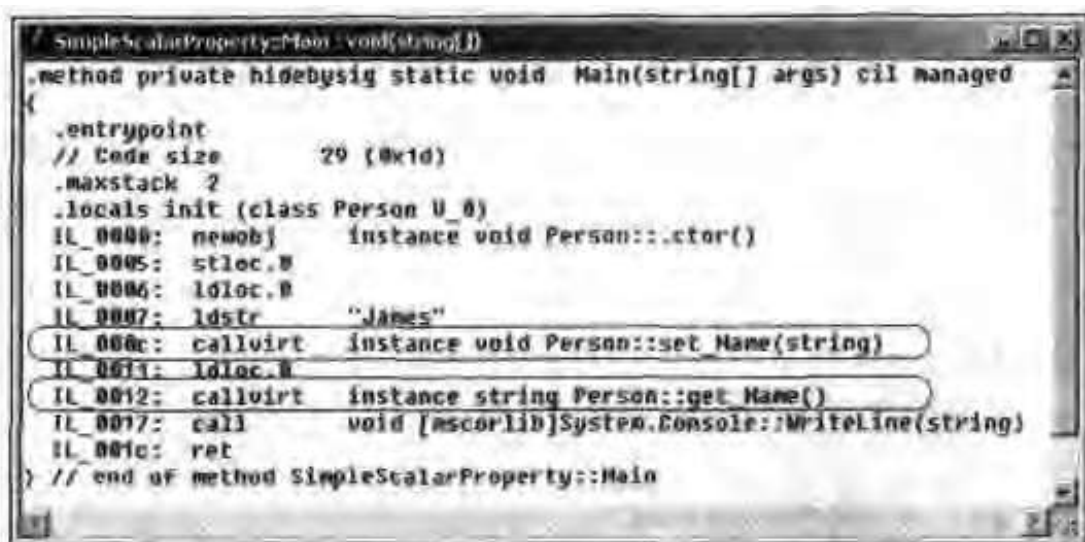


图 4-3

特别要注意以下语句:

- 语句 IL_000c 设置 Name 属性。为了达到这个效果, 编译器生成代码来调用 set_Name()方法。
- 语句 IL_0012 获取 Name 属性。为了达到这个效果, 编译器生成代码来调用 get_Name()方法。

在上面的屏幕中标记的语句通过设置编译器生成的代码, 分别调用 set_Name()和 get_Name()方法, 来设置和获取 Name 属性。

理解了如何将代码编译为 MSIL, 就可以在代码中有效并适当地使用语言功能。例如, 刚才论述了在 MSIL 代码中属性如何实现为方法。只要客户代码使用属性, 编译器都使用一个方法调用。

一些 .NET Framework 语言对属性映射到方法的方式更加明确。例如 Managed Extensions for C++, 为了创建读/写属性 Name, 可以显式地定义属性函数 get_Name() 和 set_Name()。不过, 在把代码编译为 MSIL 时, 这些源代码级的区别就会消失。不管使用哪种编程语言来编写源代码, MSIL 代码看起来都一样。

稍后将介绍 MSIL 代码在运行期间的情况。JIT 编译器编译这些代码, 根据它对已加载代码的理解来进行一些优化。这意味着, 属性引入的指令会生成许多行 MSIL 指令, JIT 编译器能优化一些操作, 生成紧凑而快速的代码。实际上, 在许多情况下, JIT 编译器可以使属性的访问比字段的访问更快, 这听起来是不可能的。而方法调用可以有

效地优化，对同一实例中成员数据的访问也可以比远程访问其他实例中的成员数据更快。因此，属性是快速访问对象状态的一种极佳技术。

2. 读/写、只读和只写属性

前面例子中的 `Name` 属性是读/写属性。也可以定义只读属性和只写属性。只读属性非常有用，它们能展示数据，但不允许客户代码修改数据。例如，可以提供一只读属性来获取个人的生日。也可以使用只写属性来设置安全密码。

下面的例子说明了读/写属性、只读属性和只写属性。`Person` 类有以下属性：

- `Name` 属性，获取和设置个人的名字。个人的名字允许修改，例如，如果这个人结婚了，或者不喜欢一出生就有的名字，就可以修改其名字。
- `DOB` 属性，获取个人的生日。个人的生日必须在构造函数中设置，此后不能修改(一个人的生日不能在他出生后修改)。
- `EmailAlias` 属性，设置个人的 e-mail 别名。e-mail 别名表示雇员的 e-mail 地址中域名前的部分。e-mail 别名可以修改，但是不能获取(客户程序可以获取完整的 e-mail 地址)。
- `EmailAddress` 属性，获取个人的完整 e-mail 地址。在每次请求 e-mail 地址时，都要把公司的域名附加到个人的 e-mail 别名上。

这个例子的源代码位于文件 `readable_and_writable.cs` 中：

```
using System;

public class Person
{
    private string name;
    private DateTime dob;
    private string emailAlias;

    public Person(string name, DateTime dOB)
    {
        name = name;
        dob = dOB;
    }

    public string Name
    {
        get { return name; }
        set { name = value; }
    }
}
```



```
        public DateTime DOB
        {
            get { return dob; }
        }

        public string EmailAlias
        {
            set { emailAlias = value; }
        }

        public string EmailAddress
        {
            get { return emailAlias + "@MyCompany.com"; }
        }
    }

    class ReadableAndWritable
    {
        static void Main()
        {
            Person person = new Person("Steve", new DateTime(1972, 8, 22));

            person.EmailAlias = "Steve.S";

            Console.WriteLine("Name - " + person.Name);
            Console.WriteLine("DOB - " + person.DOB.ToShortDateString());
            Console.WriteLine("Email Address - " + person.EmailAddress);

        }
    }
```

运行这个程序，将在控制台窗口中获得以下输出：

```
C:\Class Design\Ch 04>readable_and_writable.exe
Name - Steve
DOB - 22 August 1972
Email Address - Steve.S@MyCompany.com
```

可以使用 MSIL Disassembler 工具来研究只读和只写属性如何编译为 MSIL 代码。在 `readable_and_writable.exe` 上运行 MSIL Disassembler，将显示如图 4-4 所示的

内容。



图 4-4

在 MSIL Disassembler 窗口中应注意以下几点：

- `get_Name()`和 `set_Name()`方法获取和设置 `Name` 属性
- `get_DOB()`方法获取 `DOB` 属性
- `set_EmailAlias()`方法设置 `EmailAlias` 属性
- `get_EmailAddress()`方法获取 `EmailAddress` 属性
- C#编译器不生成 `set_DOB()`方法，因为 `DOB` 属性是只读的
- C#编译器不生成 `set_EmailAddress()`方法，因为电子邮件地址派生于 `EmailAlias`，这是一个只读属性

3. 静态属性

静态属性表示类型范围内的一个信息，我们希望将之作为类的一部分，而不是作为特定实例的一部分展示。与上一章讨论的静态方法一样，客户代码通过类，而不是类的实例来访问静态属性。

下面的例子说明了如何定义一个静态属性 `Domain`，来表示用作个人的 e-mail 地址一部分的域名。这里假定应用程序中的所有入都有同样的域名，因此域名不是针对 `Person` 类的实例的，而是针对整个 `Person` 类的。

这个例子的源代码位于文件 `static_properties.cs` 中：

```
using System;

public class Person
{
    private string name;
    private DateTime dob;
    private string emailAlias;
    private static string domain;

    public Person(string Name, DateTime dOB)
    {
        name = Name;
        dob = dOB;
    }

    public string Name
    {
        get { return name; }
        set { name = value; }
    }

    public DateTime DOB
    {
        get { return dob; }
    }

    public string EmailAlias
    {
        set { emailAlias = value; }
    }

    public static string Domain
    {
        get { return domain; }
        set { domain = value; }
    }

    public string EmailAddress
    {
        get { return emailAlias + "@" + domain; }
    }
}
```

```
    }  
}  
  
class ReadableAndWritable  
{  
    static void Main()  
    {  
        Person person = new Person("Steve", new DateTime(1972, 8, 22));  
  
        person.EmailAlias = "Steve.S";  
        Person.Domain = "AnotherCompany.com";  
  
        Console.WriteLine("Name - " + person.Name);  
        Console.WriteLine("DOB - " + person.DOB.ToShortDateString());  
        Console.WriteLine("EMail Address - " + person.EmailAddress);  
    }  
}
```

对这个例子要注意以下几点：

- Domain 属性是静态的
- EmailAlias 属性使用域名为个人生成完整的 e-mail 地址
- Main 子例程使用类名(Person)，而不是实例(如 person)来设置 Domain 属性

4. 编写 Get 过程的原则

定义读/写或只读属性时，需要考虑如何编写 get 访问器。

get 访问器所做的修改不应以任何方式影响对象的状态。这是因为调用 get 访问器在逻辑上与访问公共字段类似。如果在 get 访问器中更新了状态，就应该把它实现为方法。方法可能而且经常会有一些副作用。使用 get 访问器的两种情形如下：

- 要获取类中现有字段的值。在这种情况下，get 函数只作为类中已有的一个私有字段的包装器。get 函数直接返回值。
 - 要计算类中的派生值。在这种情况下，get 访问器使用类中的数据来计算新值。
- 在 UML 中，对于面向对象的分析和设计来说，这称作派生特性。

5. 编写 Set 过程的原则

定义一个读/写或只写属性时，需要考虑如何编写 set 访问器。最重要的决定是如何处理非法值。一种方法是抛出一个异常，表示客户代码不能接受非法值。由于值传送给属性 set 访问器的方式与参数传送给参数化方法的方式相同，所以可以抛出一个

`System.ArgumentException` 异常，例如 `ArgumentOutOfRangeException` 或 `ArgumentNullException`，因为值超出了作用域，或值为 `null`。

注意：

结构化的异常处理机制详见第 3 章。在属性中，异常可以像在方法中那样抛出和处理。

6. 标量属性的完整例子

下面的例子说明了本节前面介绍的标量属性的所有问题。在这个例子中，定义了一个类 `FootballTeam`，表示一个足球队。对于其他成员，这个类有表示队名、队服颜色，以及目前该队得分(赢得 3 分，平得 1 分，输得 0 分)的属性。

通读代码，了解在类中如何定义属性。`System.Drawing` 类用于 `Color` 类，而 `System.IO` 用于文件处理类。这个例子的源代码位于 `scalar_properties_complete.cs` 中：

```
using System;
using System.Drawing;
using System.IO;

class FootballTeam
{
    private string name;
    private Color jerseyColor;
    private short wins, draws, defeats;
    private bool logging;

    public FootballTeam(string teamName, Color teamColor)
    {
        name = teamName;
        jerseyColor = teamColor;
    }

    public string Name
    {
        get { return name; }
    }

    public int Points
    {
        get { return ((wins * 3) + (draws * 1)); }
    }
}
```

```
    }

    public Color JerseyColor
    {
        get { return jerseyColor; }
        set {
            if(value.Equals(Color.Black))
            {
                throw(new ArgumentException(
                    "Teams cannot have Black jerseys"));
            }
            else
            {
                jerseyColor = value;
            }
        }
    }

    public bool Logging { set { logging = value; } }

    private FileStream LogStream
    {
        get {
            try
            {
                return (new FileStream(name + ".log", FileMode.Append,
                    FileAccess.Write));
            }
            catch (System.IO.IOException)
            {
                return (new FileStream("Default.log", FileMode.Append,
                    FileAccess.Write));
            }
        }
    }

    public void PlayGame(string opponent, short goalsFor,
        short goalsAgainst)
    {
        if (goalsFor > goalsAgainst)
```



```
        wins++;
    else if (goalsFor == goalsAgainst)
        draws++;
    else
        defeats++;

    if (logging)
    {
        StreamWriter sw = new StreamWriter(LogStream);
        sw.WriteLine("{0} {1}-{2} {3}", name, goalsFor,
            goalsAgainst, opponent);
        sw.Flush();
        sw.Close();
    }
}
```

在这个 `FootballTeam` 类中，注意以下几个设计要点：

- 构造函数显式地初始化 `name` 和 `jerseyColor`。其他字段进行隐式的初始化。`wins`、`draws` 和 `defeats` 隐式初始化为 0，`logging` 隐式初始化为 `False`。通常，在合适时使用隐式初始化是比较好的做法。在本例中对 `wins`、`draws`、`defeats` 和 `logging` 的隐式初始化就很好。
- `Name` 属性获取球队的名称。`Name` 没有 `set` 访问器，因为球队名称一旦创建，就不会改变(这和美式足球队不同，当特许权卖给另一个城市时，它的名称会改变)。
- `Points` 属性计算并返回目前球队所得的分数。这个属性不存储到对象中：它总是在请求时重新计算。如果经常访问 `Points` 属性，另一个策略是使用每次比赛时都会更新的 `points` 字段。不过，如果 `Points` 属性很少使用，更新该值可能要花费很大的系统开销。本例的设计假定只在请求时重新计算 `Points` 值。
- `JerseyColor` 属性获取和设置队服的颜色。`set` 访问器包括有效性验证，以阻止球队有黑色队服(只有裁判才能穿黑色)。此时会抛出一个 `ArgumentException` 异常。
- `Logging` 属性是一个只写属性的例子。这个属性让客户代码允许或禁止把结果记录到文件中。如果这个属性是 `True`，就把该球队参加的每场比赛结果写入一个日志文件中。`Logging` 属性没有 `get` 访问器，客户程序应该永远不需要查询日志是启用还是禁用的。
- `LogStream` 属性是一个 `Private` 属性，只能在 `FootballTeam` 类中使用。该属性

的用途是创建和返回一个 `FileStream` 对象，该对象可以用来把结果写入到一个日志文件中。该访问器封装了用来确定使用哪个文件来记录的逻辑。此时，属性比字段更合适。

- `PlayGame()` 方法递增 `wins`、`draws` 或 `defeats` 的值。这个子例程也测试 `Logging` 属性，来弄清是否启用日志。如启用了日志，该子例程就使用 `LogStream` 属性来获取一个 `FileStream` 对象，用来把结果记录到文件中。

我们可以使用 `FootballTeam` 属性(及其他成员)，如下面的客户代码所示：

```
class Season
{
    static void Main()
    {
        FootballTeam myTeam = new FootballTeam("Wolves", Color.Gold);

        myTeam.Logging = true;

        myTeam.PlayGame("Portsmouth", 2, 1);
        myTeam.PlayGame("Manchester United", 2, 3);
        myTeam.PlayGame("West Bromwich Albion", 4, 0);
        myTeam.PlayGame("Stoke City", 3, 2);
        myTeam.PlayGame("West Ham", 1, 1);

        Console.WriteLine(myTeam.Name + "
                           (" + myTeam.JerseyColor.Name + ") - " +
                           myTeam.Points.ToString());
    }
}
```

有关 `Main()` 方法应注意以下几点：

- 使用 `Name` 只读属性来显示球队名称
- 使用 `JerseyColor` 读/写属性来修改球队队服的颜色，然后显示新的队服颜色
- 使用 `Logging` 只写属性来启用文件日志
- 使用 `Points` 只读属性来计算球队所得的分数

运行这个应用程序，将在控制台窗口中显示以下输出：

```
C:\Class Design\Ch 04>scalar_properties_complete.exe
Wolves (Gold) - 10
```

该应用程序还把足球比赛结果记录到文件 `Wolves.log` 中，该文件包含以下内容：

Wolves 2-1 Portsmouth
Wolves 2-3 Manchester United
Wolves 4-0 West Bromwich Albion
Wolves 3-2 Stoke City
Wolves 1-1 Newcastle United

4.1.3 索引器

知道了如何使用标量属性后，下面介绍索引器。

C#不支持参数化的标量属性。在支持参数化标量属性的其他语言中，这些标量属性常常与私有数组或数据集合一起使用。例如，如果 **Person** 类有一个属性 **Child**，该类就必须支持几个子类。**Child** 参数用于引用数组中的各个元素。

这个技术在 C# 中不支持，但 C# 支持索引器，这是一个解决此问题的较佳方式。

C++用户注意：

C#索引器在 C++ 中称为索引属性。

前面考虑了 **Person** 类有属性 **Child** 的情况，但如果需要处理许多人的信息，而不是一个人的信息，该怎么办？此时就需要一个 **Person** 对象的层次结构，其中每个 **Person** 对象都有子对象。索引器允许把像 **Person** 这样的类型当做数组来处理：

- 索引器的声明与标量属性相同。
- 索引器可以使用 **get** 和 **set** 访问器间接访问底层的私有字段。
- 索引器的 **set** 访问器有一个隐藏的隐式参数 **value**。
- 索引器可以是任何值类型或引用类型。

但与标量属性不同的是，索引器有下述特性：

- 索引器允许把类当作使用 **[]** 运算符的数组来对待
- 索引器是参数化的，并接受参数
- 索引器总是可以间接访问隐藏的集合或类型数组
- 索引器属性必须声明为 **this**
- 索引器给类添加一个隐藏的属性 **Item**
- 索引器的行为与默认属性类似，因为客户代码不需要引用索引器的名称
- 类只能有一个索引器，因为一个类只能有一个默认的类型成员

当类展示一个索引器时，有时就称为包含类，因为它就像其他类型的集合的容器一样。这些类型也可以是其他类型的容器，从而创建一个对象类型的层次结构。在 .NET Framework 中，**DataSet** 对象就是这样，该对象是 **Table** 对象的容器，而 **Table** 对象又是 **Row** 和 **Column** 对象的集合。

索引器用于创建像这样的对象层次结构时,对象之间的关系与数据库中的一对多或父子关系类似。

1. 在类中定义一个索引器

索引器可以间接访问私有变量,这与标量属性相同。但与标量属性不同的是,索引器所访问的私有数据总是某种集合,例如类或结构的数组。这是因为我们一般要对整个实体进行索引,而不是只对该实体的一个特性进行索引。

例如,如果在 `Person` 类中创建一个索引器属性,来表示某人的姓名,可以编写下面的代码:

```
Person[1] = "James";  
Person[2] = "Steve";
```

这就使用了类似数组的语法,但只能用于其他类属性,例如 `Email`、`Address` 等。不能添加其他索引器,因为索引器是默认的属性,一个类中只能有一个默认属性。

要使用索引器,需要对整个 `Person` 实体进行索引,而不是只对某人的姓名进行索引。为此,需要一个包含类和类型为 `Person` 的索引器。这样才能索引 `Person`,而不只是索引 `Person` 的一个特性。

下面的例子(`indexers_simple.cs`)演示了上述内容:

```
using System;  
  
public class Person  
{  
    private string name;  
    public Person(string Name)  
    {  
        name = Name;  
    }  
  
    public string Name  
    {  
        get { return name; }  
    }  
}  
  
public class Authors  
{  
    private Person[] persons = new Person[5];
```

```
        public Person this [int index]
        {
            get { return persons[index]; }
            set { persons[index] = value; }
        }
    }

    public class SingleIndexer
    {
        static void Main()
        {
            Authors authors = new Authors();
            authors[0] = new Person("James");
            authors[1] = new Person("Roger");
            authors[2] = new Person("Ben");
            authors[3] = new Person("Richard");
            authors[4] = new Person("Teun");

            for(int i=0; i<5; i++)
                Console.WriteLine(authors[i].Name);
        }
    }
}
```

编译并运行此程序，就会在控制台窗口中显示下面的内容：

```
C:\Class Design\Ch 04>indexers_simple.exe
James
Roger
Ben
Richard
Teun
```

下面列出第一个索引器的要点：

- **Person** 实体表示为一个类，该类的属性和对应的私有字段可以存储和访问 **Person** 的特性。
- **Authors** 类是 **Person** 的容器类。之所以称为容器类，是因为它包含一个类型为 **Person** 的隐藏私有数组，而该数组包含索引器。该索引器的类型也是 **Person**。
- 索引器属性是参数化的，接受一个类型为 **int** 的参数。与标量属性一样，索引器也可以间接访问私有字段。但是，私有字段是一个 **Person** 对象的数组。**int** 参数

表示数组中的序数值。

- 索引器用 `this` 关键字创建，这将隐式创建一个默认的属性 `Item`。在 C# 客户代码中访问索引器时，无需指定属性的名称 `Item`，因为它是默认属性。
- `SingleIndexer` 客户代码有非常直观的编码样式。这里使用了容器类(`Authors`)和数组语法，但每个元素的类型都是 `Person`
- `Person` 类的数组的管理在客户代码中不可见。这称为封装或数据隐藏，因为 `person` 数组只能通过 `Authors` 类索引器访问。

当包含索引器的 C# 类通过非 C# 客户代码访问时，默认属性 `Item` 是非常重要的。因为 `Item` 是一个默认属性，它从不会在 C# 客户程序中引用。但其他语言，如 Visual Basic .NET 使用 C# 的 `Person` 类时，可以显式引用 `Item` 属性。比较下述 Visual Basic .NET 例子和上面的 C# 客户代码：

`indexers_simple.vb:`

```
Module Module1
    Sub Main()

        Dim i As Integer
        Dim authors As New Authors()

        authors.Item(0) = New Person("James")
        authors.Item(1) = New Person("Roger")
        authors.Item(2) = New Person("Ben")
        authors.Item(3) = New Person("Richard")
        authors.Item(4) = New Person("Teun")

        For i = 0 To 4
            Console.WriteLine(authors.Item(i).Name)
        Next i

    End Sub
End Module
```

在这里，Visual Basic .NET 客户代码可以显式引用 `Item` 索引器属性。如果 C# 类可以由 Visual Basic .NET 客户代码使用，就可以重写索引器的默认属性名。修改 `Authors` 类，如下所示：(完整的代码在 `indexers_indexer_name.cs` 中)。

```
public class Authors
{
```

```
Person[] persons = new Person[5];

[System.Runtime.CompilerServices.IndexerName("Person")]
public Person this [int index]
{
    get { return persons[index]; }
    set { persons[index] = value; }
}
}
```

应用上面的特性，将允许 Visual Basic.NET 客户代码引用 **Person** 属性，而不是 **Item** 属性：

```
authors.Person(0) = New Person("James")
authors.Person(1) = New Person("Roger")
authors.Person(2) = New Person("Ben")
authors.Person(3) = New Person("Richard")
authors.Person(4) = New Person("Teun")
```

下面重述要点：

- 讨论了如何在类中定义一个索引器，创建与数据库的一对多关系类似的父子关系。
- 介绍了术语“容器”类。容器类是父子关系中的父。
- 容器类拥有与索引器类型相同的对象的私有数组。在本例中，容器类或父类是 **Authors** 类，子类是 **Person** 类，**Authors** 中的私有数组也是 **Person** 类型。
- 子类本身可以包含索引器，从而成为其他子类的父类。这与 ADO.NET 的 **DataSet** 对象类似，该对象包含 **Table**，而 **Table** 又包含 **Columns** 和 **Rows**。
- 索引器实际上是一个默认属性 **Item**，它在 C# 客户代码中不可见。C# 客户代码在使用数组[]运算符时，不引用 **Item** 属性。
- Visual Basic .NET 客户代码可以引用 **Item** 属性。
- **Item** 的默认索引器属性名可以使用一个特性 `[System.Runtime.CompilerServices.IndexerName("DefaultPropertyName")]` 来修改。因此 Visual Basic.NET 客户代码比较易读。
- 索引器与标量属性不同，因为索引器是参数化的，但其参数用作子类的标识符。在本例中，参数就是私有数组的序数位置。索引器可以接受多个不同的参数，如下所述。

2. 把索引器编译为 MSIL

为了深入理解索引器的工作方式，编译上面的 C# 例子，使用 `ildasm` 查看运行的输出结果。在命令行提示窗口中，输入下面的命令：

```
C:\Class Design\Ch 04>ildasm indexer_single_vb.dll
```

在扩展 `Authors and Person` 节点后，应显示如图 4-5 所示的屏幕。

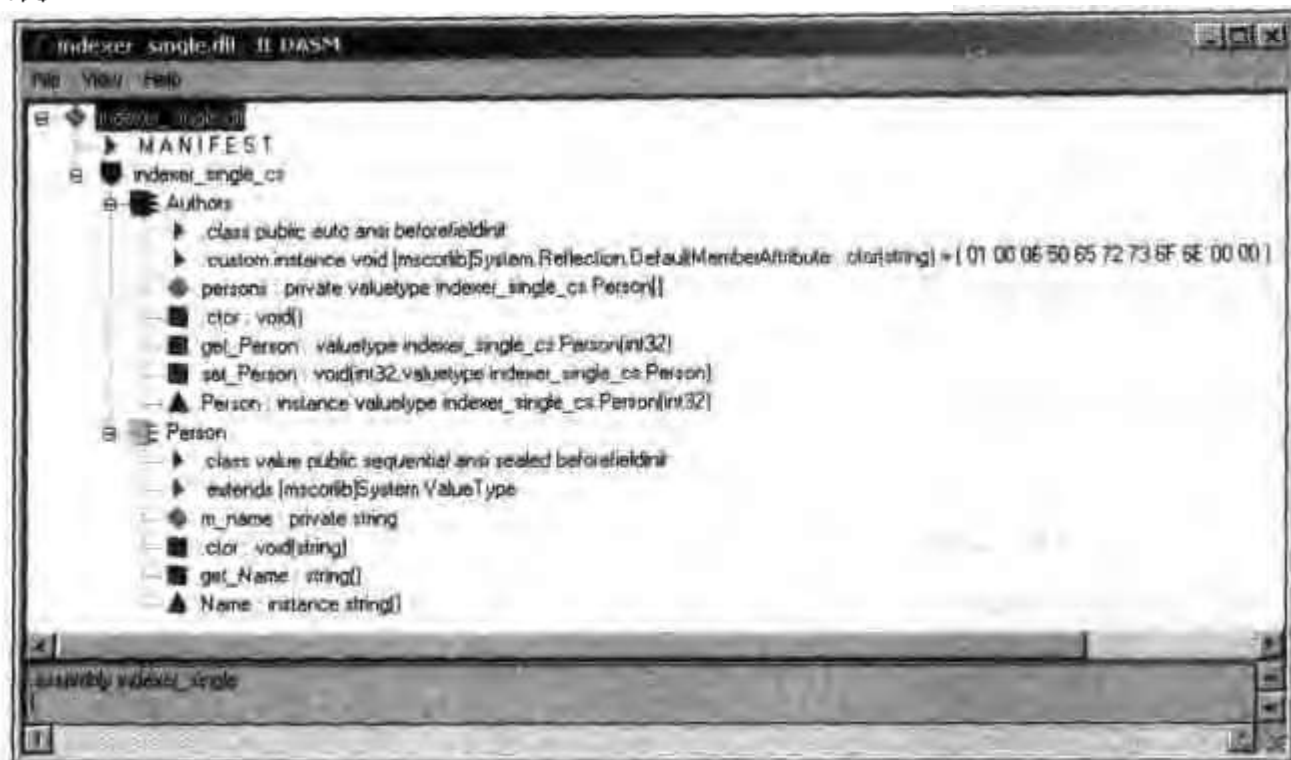


图 4-5

C# 编译器生成的 MSIL 代码证明了上述内容：

- 可以看到 `Authors` 类中定义的索引器。其定义与标量属性相同，也有 `get` 和 `set` 访问器。
- 注意索引器叫做 `Person`，而不是 `Item`，这是因为索引器的默认名称是重写的。
- 索引器实际上就是默认属性，`Authors` 类包含一个 `DefaultMemberAttribute`，它允许索引器(默认属性 `Person`)通过 C# 客户代码隐式引用。
- 在 Visual Basic .NET 中，默认属性 `Person` 可以显式引用。

3. 使用索引器和 ArrayList

前面的例子很好地演示了索引器，但缺乏产品组件的许多功能。例子中的数组其长度是固定的，即 5 个成员。数组的边界存储在 `Authors` 类内部，不能添加或删除 `Authors` 类中的 `Person` 类。

需要给 `Authors` 类添加以下的功能使其更加有用：

- 一个添加 `Person` 类的方法
- 一个删除 `Person` 类的方法
- 一个给 `Person` 类计数的方法

注意：

在类中使用索引器时，应熟悉 `Systems.Collections` 命名空间。这个命名空间提供了在数组和集合上非常有用的许多变体。其中一个最有用的是 `ArrayList`，本主题将主要使用它。

把上述功能列表包含进来，扩展上面的例子。

修改 `indexers_simple.cs` 文件中的 `Authors` 类，如下所示。完整的代码在 `indexers_simple_arraylist.cs` 中：

```
public class Authors
{
    private ArrayList persons = new ArrayList();

    public Person this [int index]
    {
        get { return ((Person)persons[index]); }
        set { persons[index] = value; }
    }

    public void Add(Person person)
    { persons.Add(person); }

    public void Remove(Person person)
    { persons.Remove(person); }

    public int Count { get { return persons.Count; } }
}
```

`Authors` 类现在的功能就强大多了。我们不是使用 `Person` 类的一个简单数组，而是使用 `Person` 类的 `ArrayList`。`ArrayList` 对象在 `System.Collections` 命名空间中，它支持使 `Authors` 类正常工作所需要的所有功能，因此现在可以通过索引器为 `ArrayList` 对象提供一个强类型的包装器。`Authors` 类是强类型的，因为 `ArrayList` 对象存储了 `object` 类型的元素。`Authors` 类可以接受和返回 `Person` 类型的值，这就是索引器 `get` 访问器能把 `ArrayList` 元素显式转换为 `Person` 的原因。

注意：

因为 `ArrayList` 元素的类型是 `Object`，所以这些元素被隐式地装箱了。

为了测试新 `Authors` 类，添加下面的客户代码：

```
public class SingleIndexer
{
    static void Main()
    {
        Authors authors = new Authors();
        authors.Add(new Person("James"));
        authors.Add(new Person("Roger"));
        authors.Add(new Person("Ben"));
        authors.Add(new Person("Richard"));
        authors.Add(new Person("Teun"));

        for(int i=0; i<authors.Count; i++)
            Console.WriteLine(authors[i].Name);
    }
}
```

程序的结果仍与本主题中前面的例子相同,即使完全修改了 Authors 类的实现代码,也是如此。要注意的区别是:

- Authors 类包含一个私有 ArrayList 对象,它用于存储每个 Person 对象。
- Authors 类的 Add()方法现在替代了索引器的 set 访问器,用于把一个新的 Person 类添加到 Authors 类中。索引器的 set 访问器仍作为通过 Authors 类更新 Person 对象的机制。
- 索引器在 for 循环中使用,把 Person 类中 Name 属性的顺序输出显示到控制台窗口上。

本节使用 ArrayList 类改进 Authors 类,提供了许多 .NET Framework 类中都有的 Add()、Remove()和 Count()方法。ArrayList 类本身提供了这个功能,所以可以为 ArrayList 类提供一个强类型的包装器,并使索引器属性支持顺序读取和更新访问。

4. 重载索引器

有时需要对集合的值建立索引,而不是对其序数值进行索引。本章前面使用的索引器都是基于 ArrayList 中 Person 类的序数值。但现在要根据每个人的姓名对集合建立索引。索引器可以像方法那样重载。方法重载的规则(详见第3章)可以应用于重载索引器,即多个索引器可以有相同的名称,只要其签名不同即可。

为了创建这个例子,需要扩展 Person 类,以存储其他一些数据。重载的索引器允许按姓名访问 Person 对象。Age 属性也要添加到 Person 对象中,说明客户代码在工作,新的索引器查找到正确的 Person 对象。完整的代码在 overloading_indexer.cs 中:


```
using System;
using System.Collections;

public class Person
{
    string name;
    short age;

    public Person(string name, short age)
    {
        this.name = name;
        this.age = age;
    }

    public string Name
    {
        get { return name; }
        set { name = value; }
    }

    public short Age
    {
        get { return age; }
        set { age = value; }
    }
}

public class Authors
{
    private ArrayList persons = new ArrayList();

    public Person this [int index]
    { get { return ((Person)persons[index]); } }
}
```

下面提供索引器的重载，允许客户代码根据 `author`(类型为 `Person`)的 `Name` 和 `ArrayList` 中的序数值来建立索引：

```
public Person this [string name]
{
    get
```

```
{
    foreach(Person person in persons)
    {
        if (person.Name == name)
            return person;
    }
    return (Person)persons[0];
}
```

新(重载)的索引器与原索引器的主要区别是, get 访问器必须搜索正确的 Person 类。这是通过一个 foreach 循环完成的。注意这个例子仅在每个人只有一个名字的情况下工作。如果 ArrayList 包含同名的多个 Person 对象, 就只能找到第一个 Person 对象。使用原索引器(它按照元素的序数值来访问 ArrayList)仍可以访问同名的其他 Person 对象。

注意:

ArrayList 类还支持方法 BinarySearch 和 Contains, 它们在索引器的 get 访问器中提供了一个高效的搜索机制。

```
public void Add(Person person)
{ persons.Add(person); }

public void Remove(Person person)
{ persons.Remove(person); }

public int Count { get { return persons.Count; } }
}

public class OverloadedIndexer
{
    static void Main()
    {
        Authors authors = new Authors();
        authors.Add(new Person("James", 31));
        authors.Add(new Person("Roger", 21));
        authors.Add(new Person("Ben", 21));
        authors.Add(new Person("Richard", 21));
        authors.Add(new Person("Teun", 21));
        Person author = authors["James"];
        Console.WriteLine(author.Name + " - " + author.Age.ToString());
    }
}
```

```
        author.Age = 21;  
        author = authors["James"];  
        Console.WriteLine(author.Name + " - " + author.Age.ToString());  
    }  
}
```

两个索引器都忽略了 set 访问器，因此它们都是只读的，这是因为与 get 访问器搜索并更新 Person 类一样，必须为 set 访问器编写一些难以使用的代码。比较简单的解决方案是在 Person 类的内部提供 set 访问器。因为两个索引器的 get 访问器都返回对 Person 类型的引用，所以可以通过对 Person 类型的引用把所有的更新内容都传送给已有的 Person 对象。

注意：

当内部数组或集合包含大量的数据时，为了不降低性能，就需要一个更高效的搜索例程。如果 ArrayList 中有许多元素，顺序循环和比较这些对象的效率就很低，此时应考虑使用 BinarySearch 或 Contains 方法。

5. 索引器小结

索引器通过使用[]运算符语法，使类的行为与数组类似。索引器属性实际上是一个隐藏的默认属性 Item。这个 Item 属性从来不在 C#代码中引用，因为它是默认的类型成员。因为一个类只能有一个默认类型成员，所以一个类只能有一个索引器。但是，索引器可以重载，根据不同的内部数据对类建立索引。

索引器可以用于索引简单的类型，例如字符串或数字，而在索引比较复杂的类型，如定制类(如前面的 Person 类)时，索引器的效率最高。要索引定制类，可以创建一个容器类，把相同类型的索引器展示为要索引的定制类。容器类应包含一个私有内部数组或集合，来存储索引的类型，还应包含公共方法来添加、删除或计数索引的类型。私有的 ArrayList 对象已经包含 Add()、Remove()和 Count()成员，所以容器类就成为私有 ArrayList 的包装器，只是有一个索引器属性，允许客户代码像处理数组那样处理定制类。

可以提供重载的索引器，根据类上的一个属性而不是 ArrayList 中的序数值来索引定制类，并利用专门的代码搜索 ArrayList 中需要的对象。为了在索引器 get 访问器中启用搜索例程，以处理重复的对象，应使用惟一的标识符。这样，索引器就可以以与关系数据库访问非常类似的方式使用。

索引器是通过重载[]运算符来使类类似于数组的。

4.2 C#中的运算符

运算符是在一个或多个类型之间执行操作的符号和字符。例如+运算符可以用于把两个数字加在一起，并返回其结果。它还可以用于把两个字符串连接在一起，返回一个连接后的字符串。

运算符的行为是在类型级中定义的。这称为运算符重载，因为这是把运算符重载来对特定的类型执行特定的行为。

运算符的行为建立在类型的基础之上，这些行为包含在运算符重载方法中。与其他类型的方法一样，这些封装的代码执行特殊的任务。但在运算符重载方法中，方法包含了运算符在用于特定类型时所应用的必要逻辑和行为。因此，运算符也可以用于封装表达式。

4.2.1 表达式中的运算符

CLR 没有提供运算符的内容。在使用运算符时，C#语言编译器会生成执行操作的必要 MSIL 代码。运算符被当做特殊的方法，编译为 MSIL 代码。由 C#编译器为运算符生成的 MSIL 代码在本节的后面介绍。

.NET Framework 中的许多类型都支持运算符；这些类型一般是基本值类型，例如数字和日期。对于这些类型，应确定它们支持哪些运算符，其行为是什么。在设计自己的类型时，也要考虑这些问题。类设计人员必须考虑哪些运算符与要设计的类相关，每个运算符对于每个定制类型应实现什么行为。

运算符重载对 C++开发人员来说很熟悉，但 Java 和 Visual Basic 开发人员，包括 Visual Basic .NET 开发人员可能不了解运算符重载。本节将说明如何和何时使用运算符重载。

本章将介绍如何为自己的定制类型定义运算符。但这些基本上是类，而运算符与结构的关系比较密切，本节将说明其原因。表 4-2 列出了可以重载的所有运算符。

表 4-2

运算符的类型	运 算 符	说 明
一元运算符	+, -, !, ~, ++, --, true, false	一元运算符带一个参数，例如 i++;
二元运算符	+, -, *, /, %, &, , ^, <<, >>	二元运算符带两个参数，例如 int x = 7; int z = x + y;

(续表)

运算符的类型	运 算 符	说 明
比较运算符	<code>==, !=, <, >, <=, >=</code>	比较运算符总是带两个参数，返回一个布尔值，例如： <pre>int a = 7; int b = 10; if (a > b){...}</pre> 还要注意，比较运算符必须成对重载，例如如果重载了<code>></code>，就必须重载<code><</code>

注意：条件逻辑运算符`&&`和`||`、数组索引运算符`[]`、类型转换运算符`()`、赋值运算符`=`和`+=`，以及其他运算符 `new` 和 `typeof` 不能重载。

在详细介绍运算符重载之前，考虑下面的内容：

- 不能定义新的运算符：可重载的运算符列表是有限的，如表 4-1 所示。
- 运算符有优先级：在表达式中使用运算符时，必须记住，乘法运算符`(*)`的优先级比加法运算符`(+)`高。

4.2.2 运算符重载的语法

下面介绍一个 `Employee` 类型的例子，它表示带有 `Name` 和 `Salary` 特性(attribute)的 `Employee` 对象。还有一个 `Bonus` 类型，表示年度奖金。这个类型有一个特性 `Amount`。在应用程序中，需要在客户代码中使用此表达式奖励雇员奖金：

```
employee = employee + bonus;
```

本节在后面研究这些类型的实现代码。现在声明一个允许执行该表达式的运算符重载，如下所示：

```
public static Employee operator+(Employee employee, Bonus bonus)
{
    employee.Salary = employee.Salary + bonus.Amount;
    return employee;
}
```

运算符重载的声明与参数化方法相同。这个运算符是一个静态方法，其可见性是 `public`。方法声明中的这些指令会使代码看上去与其他类型成员一致，因为所有的运算符重载都必须是 `public` 和 `static`。该运算符的作用是创建简单、直观的客户代码表达式：

运算符重载必须是 `public`，才能在客户代码中使用运算符。推广使用运算符重载的原因是这样将允许运算符用于类型的所有实例。我们不希望运算符只能用于类型的某些实例，所以运算符重载也必须是 `static`。

声明的下一部分是运算符的返回类型。如果用 `+` 运算符把两个整数加在一起，返回值(和)也就是整数类型。在本例中要返回 `Employee` 类型。

下面是参数：一个 `Employee` 类型和一个 `Bonus` 类型。这些类型的例子如下所示。在创建运算符重载方法时，参数的个数和顺序是非常重要的。见表 4-3。

表 4-3

一元运算符	必须声明为接受一个参数，返回一个结果。例如， <code>++</code> 运算符必须接受一个参数，返回参数递增 1 后的结果
二元运算符	必须接受两个参数，表示表达式的左边和右边。二元运算符重载并返回一个相应的结果。例如 <code>*</code> 运算符必须接受乘号左右两边的乘数，返回它们的积
比较运算符	必须接受两个参数，表示表达式的左边和右边，返回一个布尔值。例如，如果左边的参数大于右边的参数， <code>></code> 运算符就必须返回 <code>true</code>

本例基于二元运算符 `+`。该运算符接受两个参数，分别表示表达式的左右两边。下面是客户代码表达式：

```
employee = employee + bonus;
```

下面是方法签名：

```
(Employee employee, Bonus bonus)
```

运算符重载方法把第一个参数 `Employee` 作为表达式左边的内容接受。第二个参数的类型是 `Bonus`，它是表达式右边的内容。

该方法包含当把 `Employee` 类型和 `Bonus` 类型加在一起时 `+` 运算符应完成的所有工作内容。在本例中，使用 `+` 运算符计算 `Employee` 的工资，具体方法是把 `Bonus.Amount` 属性和 `Employee.Salary` 属性加在一起。

与方法和属性一样，也可以为结构和类定义运算符。这是一个重要的内容，因为结构与类不同，更适合于使用运算符重载。

4.2.3 运算符和类

为了说明运算符的功能，下面比较一个有运算符重载的例子和一个没有使用运算符的例子。

下面的简单例子没有使用运算符重载。其代码在 `operators_no_operators.cs` 中：

```
using System;

class Bonus
{
    private int bonus;

    public int Amount
    {
        get { return bonus; }
        set { bonus = value; }
    }
}

class Employee
{
    private string name;
    private int salary;

    public string Name
    {
        get { return name; }
        set { name = value; }
    }
    public int Salary
    {
        get { return salary; }
        set { salary = value; }
    }
}

class CalculateBonus
{
    static void Main()
    {
        Employee employee = new Employee();
        Bonus bonus = new Bonus();

        employee.Name = "James";
```

```
employee.Salary = 8000;

bonus.Amount = 1000;

Console.WriteLine(employee.Name + " - " + employee.Salary);

employee.Salary = employee.Salary + bonus.Amount;

Console.WriteLine(employee.Name + " - " + employee.Salary);
}
}
```

运行程序，就会在控制台窗口中显示下面的输出：

```
C:\Class Design\Ch 04>operators_no_operators.exe
James - 8000
James - 9000
```

这个例子没有什么特殊的地方。所使用的所有技术都在本章介绍过了。该例子计算雇员的新工资：给原工资再加上奖金。这是在客户代码的下述表达式中完成的：

```
employee.Salary = employee.Salary + bonus.Amount;
```

这种样式的编程在功能上讲是可行的，但有两个小缺点：

- 与其他.NET Framework 代码的工作方式不一致，特别是在与基本类型比较时，就更是如此。例如，把两个整数加在一起，就不能使用下面的表达式：

```
numberone.Value = numberone.Value + numbertwo.Value
```

- 计算雇员的新工资时使用的业务逻辑是表达式。它可以包装在一个方法中以提供封装性，但在把两个数字加在一起时，该方法没有提供许多开发人员都能接受的对应级别的抽象性。

运算符重载允许使用在一般情况下与基本数据类型相关的简单表达式，用于自己的类型。例如，在 Employee 类型例子中重载+运算符，就可以编写下面的代码：

```
employee = employee + bonus;
```

为此，需要在 Employee 类中创建一个运算符重载方法。修改 Employee 类，如下所示，完整的代码在 operators_overload.cs 中：

```
public static Employee operator+(Employee employee, Bonus bonus)
{
```



```
        employee.Salary = employee.Salary + bonus.Amount;  
        return employee;  
    }
```

这里使用一个运算符重载方法重载了+运算符。运算符重载其实是特殊的方法。它们可以改变参数，返回参数的值。这是参数化方法的一般特性。

要测试运算符重载，可以修改 Calculate Bonus 中的客户代码，如下所示：

```
static void Main()  
{  
    Employee employee = new Employee();  
    Bonus bonus = new Bonus();  
  
    employee.Name = "James";  
    employee.Salary = 8000;  
  
    bonus.Amount = 1000;  
  
    Console.WriteLine(employee.Name + " - " + employee.Salary);  
  
    employee = employee + bonus;  
  
    Console.WriteLine(employee.Name + " - " + employee.Salary);  
}
```

运行程序，就会在控制台窗口中显示下面的输出：

```
C:\Class Design\Ch 04>operators_overload.exe  
James - 8000  
James - 9000
```

现在客户代码可以使用简单的表达式计算雇员的奖金了。这个表达式在语义上与把两个数字加在一起或连接两个字符串的表达式相同。

通过+运算符重载使用简单的表达式，将使客户代码更简单。运算符重载执行了原来很复杂的表达式。我们只是把比较复杂的表达式从客户代码移动到 Employee 类中。这是比较好的设计方式，因为雇员新工资的计算过程封装在 Employee 类中。

这个例子很有意义，因为表达式 Employee = Employee + Bonus; 很直观。

警告：

但重载一个执行 Employee = Employee + Employee; 表达式的运算符没有意义，因为把两个雇员加在一起没有任何意义。

注意：

在考虑运算符重载时，最好先考虑一下客户代码中的什么表达式有意义。一个比较好的方法是使用伪代码。以这种方式考虑运算符重载，有助于创建对客户代码开发人员有意义的直观的运算符重载。

在使用之前预先计划，运算符重载就可以使编程更直观，客户代码的表达式更简单。但运算符重载本身通常包含没有使用运算符重载时在客户代码中使用的表达式。应仔细权衡为类创建运算符重载所节省的精力。如果只有一两行客户代码使用较复杂的非运算符重载表达式，就不应创建运算符重载。

1. 把运算符编译为 MSIL

下面查看一下 C# 编译器为运算符重载生成的 MSIL 代码。在例子 `operators_overload.exe` 上运行 `ildasm`，扩展 `Bonus`、`CalculateBonus` 和 `Employee` 节点后，就会显示如图 4-6 所示的屏幕。



图 4-6

可以看到 `Employee +` 运算符重载和 `Employee` 类的其他成员。这是一个方法，双击它：

```
op_Addition : class Employee(class Employee, class Bonus)
```

应显示如图 4-7 所示的屏幕。

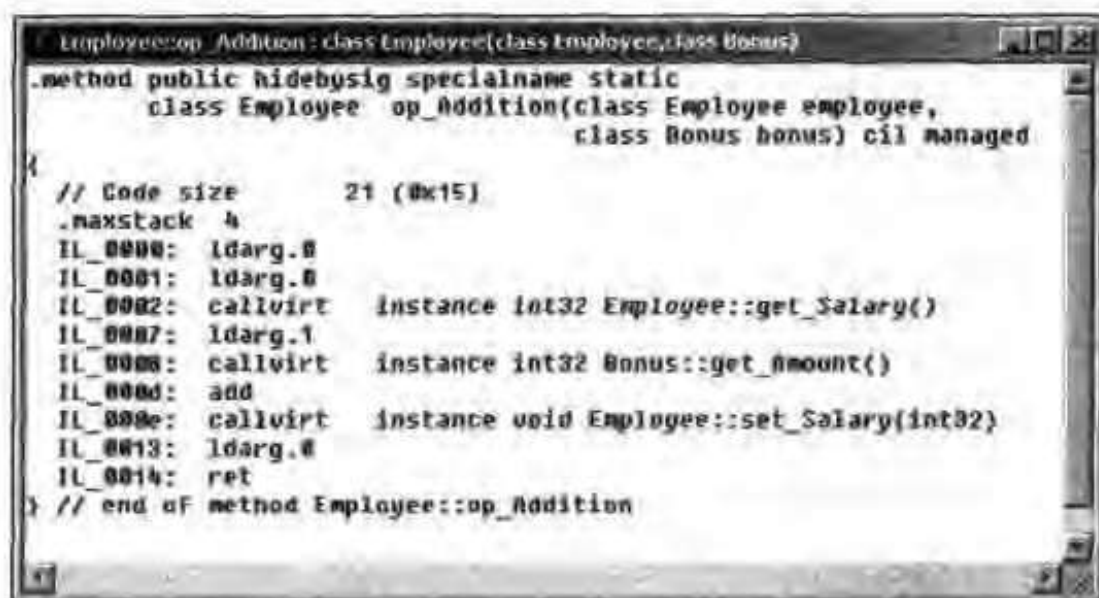


图 4-7

C#编译器生成了这个方法，它的外观和行为与其他参数化方法相同。该方法接收 Employee 和 Bonus 类型的参数，调用 Employee get_Salary()方法，之后调用 Bonus get_Amount()方法。然后执行相加操作，最后调用 Employee set_Salary()方法。get() 和 set() 方法是属性的访问器，属性则是特殊的方法。

简而言之，运算符重载在 MSIL 代码中表示为参数化的方法。

2. 运算符重载和跨语言支持

如前所述，运算符重载对 Visual Basic 和 Java 开发人员来说是新内容。C++开发人员则很熟悉运算符重载。C#支持运算符重载，但 Visual Basic .NET 不支持。如果用 Visual Basic .NET 编写的程序不能创建带有运算符重载的类型，这两种.NET 语言之间的区别就是一个大问题。Visual Basic .NET 客户代码如何调用带有运算符重载的 C#类？

下面的 Visual Basic .NET 客户程序说明了如何使用在 C#类中编写的运算符重载，且在 Visual Basic .NET 客户代码中消费。下面的例子在 operator_overload.vb 中：

```
//while compiling, use the /r: - reference switch for //operators_overload.cs file. Remember to
change the status of the //Classes Employee and Bonus to public
```

```
Module CalculateBonus
```

```
Sub Main()
```

```
Dim emp As New Employee()
```

```
Dim bonus As New Bonus()
```

```
emp.Name = "James"
```

```
emp.Salary = 8000
```

```
bonus.Amount = 1000
```

```
emp = Employee.op_Addition(emp, bonus)
```

```
Console.WriteLine(emp.Name + " - " + emp.Salary.ToString())
```

```
End Sub
```

```
End Module
```

Visual Basic .NET 仍可以使用带有运算符重载的 C# 类型，但其方式不直观，因为要用不熟悉的名字调用方法。而且在访问 C# 类成员时，这个方法在 IntelliSense 中不可见。在 Visual Basic .NET 中，不能以与 C# 客户程序相同的方式使用运算符重载表达式。

Visual Basic .NET 处理运算符重载的方式与处理其他参数化方法一样，前面在查看 C# 编译器生成的 MSIL 代码中就提到了这一点。因为 Visual Basic .NET 不支持编写带有运算符重载的类型，也就不支持在 C# 客户代码中允许使用的定制表达式。

如果 C# 类由非 C# 客户程序使用，最好用一个公共方法执行与运算符重载相同的行为。这要比调用隐藏的 `op_Addition()` 方法(或其他运算符方法)更好，因为用于此目的的方法在 IntelliSense 中可用，其名称也比较恰当。

3. 对称的运算符重载

前面的例子编写了下面的表达式：

```
employee = employee + bonus;
```

这个表达式把 `Bonus` 类型加到 `Employee` 类型上，返回一个 `Employee` 类型，并把奖金加到雇员的工资中。在 `employee` 类上为 `+` 运算符创建一个运算符重载，就可以让这个客户代码表达式工作。

但是，还必须考虑下面的表达式是否有效：

```
employee = bonus + employee;
```

这两个表达式是相同的，惟一的区别是 `Employee` 和 `Bonus` 类型的顺序。我们希望这两个表达式执行相同的行为。

因为运算符重载实际上是特殊的方法，传送给方法的参数的顺序(也就是使用表达式的顺序)就非常重要。已有的运算符重载方法不支持下面的表达式：

```
employee = bonus + employee;
```

在这种情况下，需要创建对称的运算符重载，用于在对两个不同的类型使用二元运算符重载时创建对称的表达式。对称的部分来自于要对称的表达式的需要，即：

```
employee = employee + bonus;
```

与下面的表达式相同:

```
employee = bonus + employee;
```

注意:

对相同类型使用二元运算时, 不需要对称的运算符重载。

创建对称的运算符重载时, 可以先创建第一个运算符, 再用新方法重载它, 执行相同的计算, 但参数的顺序是相反的。方法重载的内容详见第 3 章。当方法的签名不同时, 就可以重载方法。下面的代码(在 `operators_symmetrical.cs` 中)说明了如何重载运算符, 来创建对称的表达式。把下面的运算符+重载方法添加到 `Employee` 类中:

```
public static Employee operator+(Bonus bonus, Employee employee)
{
    employee.Salary = employee.Salary + bonus.Amount;
    return employee;
}
```

把下面的内容添加到 `CalculateBonus` 类中:

```
...
employee = bonus + employee;

Console.WriteLine(employee.Name + " - " + employee.Salary);
}
```

4. 类型比较和相等

在使用自己的类型时, 常常需要比较相同类型的两个实例, 看看它们是否相等。这是因为这种比较的结果不需要依赖于类型成员的变量赋值的值, 而且引用类型(如类)和值类型(如结构)之间在行为上的比较是区别的。这种区别与引用类型或值类型传送给方法参数时的区别一样。传送参数和使用参数与使用=运算符的变量赋值是相同的。

在下面的表达式中, `employee_one` 和 `employee_two` 是前面例子中使用的 `Employee` 类型:

```
employee_one = employee_two;
```

由于 `Employee` 类型是一个类, 因此是一个引用类型, 上面的赋值语句创建 `Employee`

对象的一个阴影拷贝，得到对同一实例的两个引用。使用`==`运算符或`Equals()`方法来比较这两个变量，总会返回`true`，因为它们指向相同的实例。这就是引用相等，因为等价比较基于存储在两个变量中的引用，对于引用类型来说，它们总是相等的。

`Employee`类型的两个不同实例之间的比较总是返回`false`，即使它们包含相同的成员数据。因为它们不是引用相等，两个变量指向堆上不同的`Employee`实例。没有对`Employee`类型成员数据进行比较。

如果把`Employee`类型改为结构，它就变成了值类型，赋值语句会创建结构的深度拷贝，也就是创建同一结构的两个完全独立的拷贝。对一个变量的修改不会影响另一个变量，所以它们的比较通常返回`false`，因为这两个结构中的数据是不同的。

从上面的规则可以看出，进行任何类型的比较都会生成可预测的结果。如果有两个对同一对象的引用，它们之间的比较结果就是引用相等，因为这两个引用都指向堆上的同一对象，在它们之间进行比较时，数据总是相等的。如果有两个同一引用类型的实例，就必须进行逐个成员的比较，测试它们的值是否相等。

注意：

重载`==`运算符将进行值相等的比较，为比较同一引用类型的两个不同实例提供一种可预测结果的方式。

为此，需要重载`==`运算符和`!=`运算符，提供自己的定制比较。当成员比较是相同的，则`==`运算符应返回`true`，否则，如果成员比较是不相同的，`!=`运算符应返回`false`。这就是值相等的测试。也可以使用`Equals()`方法比较一个类型的两个实例：

```
If (employee_one.Equals(employee_two))
{
    ...
}
```

`Equals()`方法是`System.Object`的一个成员，因此可以隐式用于所有的类型，包括值类型和引用类型。它必须和`==`及`!=`运算符重载一起重写，在同一引用类型的两个实例之间提供一种完全的比较机制。实际上，在重载`==`及`!=`运算符时，C#编译器会强制重写`Equals`方法。

本书的第2章说过，我们还应熟悉`GetHashCode()`方法。这个方法通过提供对象的惟一散列值，允许类型在散列表上正确工作。`GetHashCode()`方法也应重写，来获取类型的定制散列值。

下面的代码在`operators_equality.cs`中：

```
using System;
```



```
class Employee
{
    private string name;
    private int salary;

    public string Name
    {
        get { return name; }
        set { name = value; }
    }

    public int Salary
    {
        get { return salary; }
        set { salary = value; }
    }

    public static bool operator==(Employee left_employee,
                                   Employee right_employee)
    {
        if (left_employee.Name == right_employee.Name)
            return true;
        else
            return false;
    }

    public static bool operator!=(Employee left_employee, Employee right_employee)
    {
        if (left_employee.Name != right_employee.Name)
            return true;
        else
            return false;
    }

    public override bool Equals(object type)
    {
        if (!(type is Employee))
            return false;
        else
            return this ==(Employee)type;
    }
}
```

```
}

public override int GetHashCode()
{
    return this.Name.GetHashCode() + this.Salary.GetHashCode();
}
}

class CalculateBonus
{
    static void Main()
    {
        Employee employee_one = new Employee();
        Employee employee_two = new Employee();
        Employee employee_three = new Employee();

        employee_one.Name = "James";
        employee_two.Name = "Steve";

        employee_three = employee_one;

        if (employee_one == employee_two)
            Console.WriteLine("Employee One is the same as Employee Two");
        else
            Console.WriteLine("Employee One is NOT the same as Employee
                                Two");

        if (employee_one == employee_three)
            Console.WriteLine("Employee One is the same as Employee
                                Three");
        else
            Console.WriteLine("Employee One is NOT the same as Employee
                                Three");

        employee_three.Name = "Steve";

        if (employee_two == employee_three)
            Console.WriteLine("Employee Two is the same as Employee
                                Three");
        else
```



```
        Console.WriteLine("Employee Two is NOT the same as Employee  
        Three");  
    }  
}
```

运行程序，将生成如下输出：

```
C:\Class Design\Ch 04>operators_equality.cs  
Employee One is NOT the same as Employee Two  
Employee One is the same as Employee Three  
Employee Two is the same as Employee Three
```

下面是这个程序的要点：

- `==`运算符重载为在 `Employee Name` 属性上执行值相等(而不是引用相等)的比较。如果 `Employee` 类型的两个实例名是相同的，它们的值就是相等的。
- `!=`运算符重载为在 `Employee Name` 属性比较不相同返回 `false`。在重载 `==`运算符时，也必须重载 `!=`运算符，因为运算符 `==`和 `!=`是成对工作的。
- `Equals()`方法重写为与 `==`运算符执行相同的操作。如果传送了无效的参数，该方法不会抛出异常，这是因为如果参数是无效的，比较将总是 `false`。因此，`Equals()`方法首先会检查参数的类型，之后比较 `Employee` 类型成员。
- `GetHashCode()`重载为返回 `Employee Name` 和 `Salary` 属性的组合散列码。
- 创建了 `Employee` 类的三个新实例：
 - `employee_one`，其 `Name` 属性赋值为值 `James`。
 - `employee_two`，其 `Name` 属性赋值为值 `Steve`。
 - `employee_three`，其 `Name` 属性未赋值。
- 在 `employee_one` 和 `employee_two` 之间的比较返回 `false`，因为它们有不同的名称。
- 把 `employee_three` 赋予 `employee_one`；`employee_one` 和 `employee_three` 现在包含同一 `Employee` 实例的不同引用。它们之间的比较返回 `true`，因为它们的 `Name` 属性值隐式相同。
- `employee_three` 的 `Name` 属性赋予了 `Steve` 值，它在与 `employee_two`(它是 `Employee` 类型的一个独立实例)比较时，比较结果为 `true`，即使这两个实例的引用并不相等，也是如此。`==`运算符重载认为它们是相同的，因为它们的 `Name` 属性值是相同的。

简言之，要注意类型比较和相等的下述要点：

- 重载 `==`和 `!=`运算符可以用于测试值相等，而这两个运算符重写用于测试引用相等的默认行为。在比较引用类型时，一般行为是检查引用是否相等，并返回 `true` 或 `false`。

- 重载`==`和`!=`运算符时，类型成员的比较应在两个同类型的实例之间进行。这样可以重写测试引用相等的默认行为，如果某些或所有类型成员是相同的，就返回`true`。
- 除了重载`==`和`!=`运算符，还应重载`Equals()`和`GetHashCode()`方法。`Equals()`方法也用于测试相等，`GetHashCode()`方法用于生成类型实例的散列值。

4.2.4 运算符和结构

前面的例子都基于表示 `Employee` 的类。为像 `Employee` 这样的类提供运算符重载的优点是有限的。这是因为可以重载的运算符数量是有限的。

前面建立的规则是：只重载那些客户代码表达式有意义的运算符。前面的代码都基于 `Employee` 类，我们使用这些例子：

- 表达式 `employee = employee + bonus` 是有意义的，因为可以直观地猜测出雇员获得了奖金。
- 表达式 `employee = bonus + employee` 也是有意义的，因为这是上述表达式的对称表达式。
- 表达式 `employee_one == employee_two` 是有意义的，因为这是同一类型的两个实例之间的布尔比较。
- 表达式 `employee = employee + employee` 没有意义，因为雇员不能和另一个雇员相加。

在考虑可以为 `Employee` 类重载的其他运算符时，也可以看出这些运算符在应用于 `Employee` 类时没有什么意义。

- `Employee++`
- `Employee * Employee`
- `Employee >= Employee`
- `Employee < Employee`

在讨论运算符用于基本类型的方式时，应该说运算符对简单的值类型来说有更大的价值。例如，使用`++`、`*`、`<`、`>`，或`>=`这样的运算符对数字(如整数)有很大的意义。

注意：

程序员应识别运算符，了解它们用于基本类型和字符串类型时会出现什么行为。

结构非常适合于创建复杂的值类型。如果考虑一下图中的坐标和分数，就可以看出它们是已有基本类型的较复杂版本。在应用程序中表示像这样的实体时，值类型的结构(struct)要比引用类型的类更好，原因是：

- 它们是轻型的

- 它们有确定的生存周期，不进行垃圾回收

结构总是比类更适合于运算符重载。为了说明结构、类，和其他可以进行运算符重载的对象之间的区别，下面创建一个基于分数的应用程序，这是一种复杂的值类型。

有许多运算符可以重载，以创建对分数有意义的客户代码表达式。首先，创建处理加和减的运算符。

在下面的例子中可以看到这些优势，该例子在 `operators_structs.cs` 中：

```
using System;

public struct Fraction
{
    private int numerator;
    private int denominator;

    public int Numerator
    {
        get { return numerator; }
        set { numerator = value; }
    }

    public int Denominator
    {
        get { return denominator; }
        set { denominator = value; }
    }

    private static int lcd(int x, int y)
    {
        int r, t = x * y;

        while (true)
        {
            r = x % y;
            if (r == 0) break;
            x = y; y = r;
        }

        return (t / y);
    }
}
```

```
    }  
    public static Fraction operator+(Fraction lh_fraction,  
        Fraction rh_fraction)  
    {  
        Fraction result = new Fraction();  
        result.Denominator = lcd(lh_fraction.Denominator,  
            rh_fraction.Denominator);  
        result.Numerator = lh_fraction.Numerator *  
            (result.Denominator / lh_fraction.Denominator) +  
            rh_fraction.Numerator *  
            (result.Denominator / rh_fraction.Denominator);  
        return result;  
    }  
  
    public static Fraction operator-(Fraction lh_fraction,  
        Fraction rh_fraction)  
    {  
        Fraction result = new Fraction();  
        result.Denominator = lcd(lh_fraction.Denominator,  
            rh_fraction.Denominator);  
        result.Numerator = lh_fraction.Numerator *  
            (result.Denominator / lh_fraction.Denominator) -  
            rh_fraction.Numerator *  
            (result.Denominator / rh_fraction.Denominator);  
        return result;  
    }  
  
    public override string ToString()  
    { return (numerator.ToString() + "/" + denominator.ToString()); }  
}  
  
class FractionCalculator  
{  
    static void Main()  
    {  
        Fraction fraction1 = new Fraction();  
        fraction1.Numerator = 2;  
        fraction1.Denominator = 5;  
  
        Fraction fraction2 = new Fraction();
```

```
fraction2.Numerator = 1;
fraction2.Denominator = 7;

Console.WriteLine("Fraction 1 = " + fraction1.ToString());
Console.WriteLine("Fraction 2 = " + fraction2.ToString());
Console.WriteLine();

Console.WriteLine("Fraction 1 + Fraction 2 = " + (fraction1 +
    fraction2).ToString());
Console.WriteLine("Fraction 1 - Fraction 2 = " + (fraction1 -
    fraction2).ToString());
    }
}
```

执行此程序，将得到如下结果：

```
C:\Class Design\Ch 04>operators_structs.exe
Fraction 1 = 2/5
Fraction 2 = 1/7

Fraction 1 + Fraction 2 = 19/35
Fraction 1 - Fraction 2 = 9/35
```

下面是这个例子的要点：

- 把 Fraction 类型声明为结构而不是类。因此它是值类型。
- 值类型更适合于运算符重载。+和-运算符在客户代码中是有意义的。还可以使用运算符*、/、<和>编写其他有意义的表达式。
- Fraction 类型重载了 ToString()方法，创建了 Fraction 的一个可读字符串表示，这对应用程序是有帮助的。
- Fraction 结构包含一个静态方法(lcd)，使用 Euclid 的算法计算最小公分母。

成对重载运算符

一些运算符是成对使用的。例如<和>运算符。在重载这些运算符时，必须重载两个运算符，而不是只重载其中一个。在使用其他人编写的类型时，将很难为类型重载一对运算符中的一个。但 C#编译器要求必须重载一对运算符。

添加下面的代码，扩展 Fraction 类型。这些代码在 operators_pairs.cs 中：

```
public static bool operator>(Fraction lh_fraction, Fraction rh_fraction)
{
    int lowest;
```

```

lowest = lcd(lh_fraction.Denominator,
             rh_fraction.Denominator);

if (lh_fraction.Numerator *
    (lowest / lh_fraction.Denominator) >
    rh_fraction.Numerator *
    (lowest / rh_fraction.Denominator))
    return true;
else
    return false;
}

public static bool operator<(Fraction lh_fraction, Fraction rh_fraction)
{
    return !(lh_fraction > rh_fraction);
}

```

在 FractionCalculator 的客户代码中添加下面的代码：

```

.....
Console.WriteLine();

Console.WriteLine("Fraction 1 > Fraction 2? " + (fraction1 > fraction2).ToString());
Console.WriteLine("Fraction 1 < Fraction 2? " + (fraction1 < fraction2).ToString());
Console.WriteLine("Fraction 2 > Fraction 1? " + (fraction2 > fraction1).ToString());
Console.WriteLine("Fraction 2 < Fraction 1? " + (fraction2 < fraction1).ToString());
}
}

```

程序现在应生成下面的输出：

```

C:\Class Design\Ch 04>operators_pairs
Fraction 1 = 2/5
Fraction 2 = 1/7

```

```

Fraction 1 + Fraction 2 = 19/35
Fraction 1 - Fraction 2 = 9/35

```

```

Fraction 1 > Fraction 2? True
Fraction 1 < Fraction 2? False

```



```
Fraction 2 > Fraction 1? False
```

```
Fraction 2 < Fraction 1? True
```

Fraction 例子现在几乎完成了。该例子可以执行加减操作，比较大小。客户代码表达式也是有意义的。还可以更进一步，实现前面讨论的==和!=运算符，而实现<=和>=运算符显然与实现<和>运算符类似。

在重载运算符时，应重载所有可用的运算符。例如，如果重载了<和>运算符，就应该也重载<=和>=运算符。

4.2.5 运算符重载的最佳做法

前面的一些例子讨论了何处和如何为类和结构使用运算符重载。运算符重载有时可能会以错误的方式使用。下面是不应使用运算符重载的情况：

- 在所得的表达式没有意义时，不要创建运算符重载。例如，`type_x = type_y + type_z`。
- 结构一般用于表示复杂的值类型，比类更适合于运算符重载，而类一般用于表示复杂的业务实体。在自己的类型上应限制使用运算符，而使用结构表示复杂的值类型数据。
- 运算符<和>、==和!=、>=和<=必须成对重载。C#编译器不允许只重载一对运算符中的一个。在成对重载时，要在第一个运算符中实现比较逻辑，在第二个运算符中返回与第一个运算符相反的结果(NOT \!)。
- 重载==和!=运算符可以用于在同一类型的两个实例之间测试其值是否相等。这特别适合于比较两个引用类型。而两个引用类型之间一般应进行引用相等的比较。在重载这些运算符时，要重写 Equals() 和 GetHashCode()方法。
- 要重载二元运算符如+运算符，把两个不同的类型加起来，可以创建该运算符的重载，但掉换参数的顺序。这将允许在客户代码中使用对称的表达式。

4.2.6 运算符小结

运算符可以用于在客户代码中创建简单、有意义的表达式。这些表达式对客户代码开发人员来说应非常直观，从应用程序的业务目标来看是有意义的。运算符实际上只是特殊的方法，这与标量属性和索引器一样。C#编译器生成的 MSIL 代码可以验证这一点。

下面是使用运算符重载设计自己的类型时要考虑的一些问题：

- 首先从客户代码的角度来考虑运算符重载。这将有助于先用伪代码来创建希望

的表达式。这种技术可以帮助设计有意义、直观的运算符重载。

- 运算符重载可以用于封装在应用程序中频繁使用的表达式。实际上，运算符重载常常包含在没有运算符重载的情况下客户代码使用的同一个表达式。
- 开发客户代码时，只有在有明显的优势时才使用运算符重载。如果只有一两行客户代码可以用简化的表达式重新编写，就不要使用运算符重载。
- 因为运算符重载是方法，所以它们可以像其他方法一样重载，只要方法的签名不同即可。为不同类型创建同一个运算符的不同重载，可以使客户代码更一致。重载也可以用于创建对称的表达式。

4.3 小结

本章介绍了如何定义标量属性，讨论了属性、字段和方法之间的折衷。接着论述了如何定义读/写、只读和只写属性。还讨论了这些技术的规则和最佳做法。索引器允许把类型表示为数组。

运算符是用于对一个或多个类型执行操作的符号或字符。它们可以重载，以封装复杂的表达式。本章还讨论了运算符重载在 C# 中如何工作，提高代码的可读性。我们介绍了运算符重载的最佳做法，说明了运算符如何在类和结构上重载，如何重载 `==` 和 `!=` 运算符，来重写引用类型上的引用相等行为。下一章将讨论构造函数和它们在 C# 中设计类的作用。

第5章 构造函数和对象生命周期

有其他面向对象(如 Java)语言基础的 C#新手不应对 C#在托管堆上创建对象的支持级别感到惊讶。Java 开发人员总是认为,内存管理是 JVM 的专利,构造和析构是托管的过程,不应由开发人员来处理。而 C++开发人员则必须自己管理内存,但这常常出问题。

在 C#中,可以使用托管的和未托管的代码,这将允许我们决定是在托管的代码上创建对象,而不必担心对象的生命周期(因为 CLR 会处理它),还是在未托管的代码上创建对象,并自己销毁它,回收内存。本章将分析 C#中创建和销毁对象的所有方式,其中一些方式是安全的,另外一些则不太安全。为 CLR 做的宣传是有道理的,因为 CLR 内存分配和取消分配过程的效率远比我们自己能做到的高得多。

本章将介绍对象的生命周期,主要讨论如下内容:

- 对象如何在托管堆上创建,如何定义和使用构造函数
- 对象如何销毁,包括如何修改删除模式
- 如何编写高效的对象构造代码
- 如何创建对象的深拷贝和浅拷贝
- 如何使用串行化和并行化构造和销毁对象
- 最后分析在对象构造过程中使用的各种设计模式

内存的自动管理概念对 CLR 来说是最基本的,也就是在堆上为托管的对象分配内存,并使用垃圾回收器(GC)在代码的托管执行过程中解除对象的内存分配——通过托管的执行过程,可以引用对象的 CLR 的生命周期(对象的创建和删除)管理功能,来代替自己的堆内存管理。在编写托管代码时,内存漏洞(在 C++中很常见)就会消失,因为我们不再需要释放内存了。

在执行托管的代码时,会创建一个托管的堆,这样就可以分配内存,来创建类实例(托管堆只是 CLR 和 GC 管理的一块内存区域)。在 C++中创建对象时,一般可以选择两个内存区域。一个是栈,在栈上,每个线程都有有限的内存量。第二个是堆,在堆上创建对象比较可行。与栈空间不同的是,堆空间没有 OS 定义为固定数量的空间大小限制,它的空间大小由机器硬件和操作环境(其他过程和内存消耗)来定义。但是,对象的创建和删除要比在栈上访问内存慢一些。

在对象创建之后,CLR 通过使用指向下一个可用地址的指针来管理堆地址,优化了对象的创建。其优点是 CLR 将根据下一个可用的地址空间来分配内存,递增指针,因此知道下一个可分配的内存存在什么地方。这个系统要比未托管堆快得多,几乎和在

栈上创建对象一样快。这是因为 CLR 不需要搜索一块空闲的内存来放置对象，在创建了新对象之后，托管的堆指针会直接移动到下一个内存块上，省去了未托管代码给对象分配内存时需要的搜索时间。另一个优点是对象是连续存储的，应用程序定位对象就会比把指针指向不连续的内存区域快得多。

GC 则是对象管理器，专门监视对象的使用情况，并通过删除对象空间，释放内存，把它标记为可重复使用。GC 还会连续不断地压缩托管堆，来优化空间，并连续存储对象。在代码中为对象创建的任何引用都由运行对管理，使对象在托管堆上的位置对开发人员完全透明。在 C# 和 C++ .NET 中，堆上的内存分配和压缩过程对使用不安全代码创建的对象来说是一个问题，不安全的代码会直接访问内存，返回指针指向用于新建对象的堆内存。运行时不能更新对未托管对象的引用，在进行压缩时，终止指针指向不正确的内存位置。有一些技术可用于修正对象位置，不让 GC 把它从内存中删除，这个过程称为 pinning。在下面的例子中，char* 变量 pre 就被固定住了——这可以用于所有未托管的指针。这个文件称为 string.cs，必须用 /unsafe 选项编译：

```
using System;

class preString
{
    static unsafe string AddPreString(string s)
    {
        fixed(char* pre = "A")
        {
            return (s + *pre);
        }
    }

    [STAThread]
    static void Main()
    {
        Console.WriteLine(preString.AddPreString("PostString "));
    }
}
```

运行程序，将得到如下输出：

```
C:\Class Design\Ch 05>string
PostString A
```

在运行时发现许多有 null 引用的对象时，就会进行压缩。这些 null 对象不再由应

用程序使用，而且不能由 GC 处理。GC 利用了代(generation)的概念，可以给托管堆释放内存。.NET Framework 文档定义了下面的 3 个高级规则，把托管堆分为不同的代优化：

- 内存压缩过程压缩的是托管堆的一个子集，而不是整个堆，所以速度很快。这是因为它只作用一代的对象(GC 使用的术语“代”在本节的后面介绍)。
- 较新的对象(在执行过程中创建的对象，不是在执行开始时创建的)有较短的生命周期，而较旧的对象有较长的生命周期。
- 较新的对象一般彼此相关，因此它们的存储空间是连续的，也总是在最低的代上释放。

这些规则是 GC 使用“代”的基本原因。在创建一个对象时，它将在 0 代上创建，但是，当托管堆上没有空间分配给 0 代的其他对象时，GC 就会试图回收 0 代上的内存，或者压缩足够的内存，来允许应用程序继续创建对象。未回收的集合就会升级到 1 代和 2 代上(它们仅是托管堆上的不同区域)，且其生命周期比较长，而垃圾回收在这些代上进行得比较少。无论因采用 GC 而使内存的解除分配多么高效，GC 清理托管堆的速度都有可能(尽管可能性不大)慢于为对象分配内存的速度。此时，GC 就不能为新对象分配内存，应用程序就会抛出 `OutOfMemoryException` 异常。

尽管这个过程是托管的，我们不能直接控制它，但理解可以为该过程做什么仍是有益的。弱点实际上存在于未由 GC 托管的资源中，例如，文件句柄和数据库连接。本章的后面将介绍一些工具，它们可以编写高效的代码，来释放这些类型的未托管对象的引用。

5.1 对象的创建

在 C# 中创建对象与其他语言没有区别。C++、Java 和 VB 开发人员总是使用 `new` 关键字在堆上创建新对象(C++ 开发人员可以使用 `malloc()`、`heapalloc()` 等，但这些函数没有隐式调用对象构造函数)。一般情况下，对象中的所有字段要么在创建时赋值，如果没有进行赋值，就由运行时指定一个隐式值(这与 C++ 不同，在 C++ 中，如果没有设置默认值，就会生成警告和异常)。在 C# 中，赋予的默认值取决于字段是对值类型的引用，还是对引用类型的引用。如果未赋值，引用类型在默认情况下初始化为 `null`。值类型，如 `int`，在默认情况下初始化为 0。`System.ValueType` 类(值类型将继承它)有一个默认的构造函数，由派生的值类型类(如 `int`)调用，并初始化类实例的状态。记住，`System.String` 不是值类型，所以在构造过程中不会初始化为空字符串。

`Employee` 类演示了创建对象和初始化值时涉及到的问题。仅 `dept` 字段赋予了字段值，其他字段都赋予了默认值，`string` 类型的字段赋予 `null`，`int` 字段赋予 0。把这些私有字段展示为属性值，如下面的代码所示。下面的代码在文件 `object_creation.cs` 中：

```
using System;
class Employee
{
    private string name;
    private string dept = "IT";
    private int salary;

    public string Name { get {return name;} }
    public string Dept { get {return dept;} }
    public int Salary { get {return salary;} }
}

public class emp_class
{
    static void Main(string[] args)
    {
        Employee emp = new Employee();
        Console.WriteLine("Name: {0}\nDept: {1}\nSalary: {2}",
                           emp.Name, emp.Dept, emp.Salary);
    }
}
```

上面的代码提出了一个警告：Employee.name 和 Employee.salar 字段并未赋值，而是分别指定了默认值 null 和 0。运行程序，将得到如下输出：

```
C:\Class Design\Ch 05>object_creation
Name:
Dept: IT
Salary: 0
```

给字段赋予默认值，有助于避免代码中连续出现异常，因为对于应用程序语言，如 Java 和 VB 来说，默认值不在可接受的值范围之内，而如果使用的字段没有赋值，即使没有赋予默认值，程序也会编译，但会生成 null 引用异常(在 VB 中，则声明“对象变量或块未设置”)。无论是否赋予了默认值，最好总是初始化字段值，因为如果代码没有以后的某个阶段给字段赋值并使用它们，运行时赋予的值就不可能是可接受的值。

这里的一个要点是，类的字段值一般在设计期间是未知的，创建类的代码需要初始化它们。因此，C#与其他面向对象的语言一样，允许定义类的构造函数。如果这个构造函数存在，在对象创建代码使用 new 关键字创建对象时，就可以调用它。构造函数的调用是初始化进程的一部分。C#支持本章后面讨论的重载构造函数的概念。构造函

数对于在运行期间初始化字段的值非常重要——这是对象创建过程的一部分，在对象的引用返回客户代码之前，可以设置字段的默认值。与常规方法不同的是，构造函数不能显式调用，在创建对象时，如果构造函数存在，就调用它，且构造函数没有返回类型。每次创建对象时，都要隐式调用构造函数，初始化对象的状态。

为了解决初始化的一些问题，可以创建不同签名的构造函数，这种构造函数包含不同的参数列表和参数类型，在创建对象时只能调用其中的一个(除非其他的构造函数在内部调用——这是很常见的，本章在后面论述)。Employee 类需要不同类型的构造函数用于不同的目的。例如，在创建对象时，需要填充 3 个字段，或只需要填充两个字段，因为 dept 的默认值是 IT，所创建的 Employee 属于 IT 部门。另外，也可能不需要填充任何对象的字段值，这就需要创建一个没有参数的构造函数，提供不涉及字段的其他形式的初始化——当对象负责填充它自己的字段值，而使用其他技术初始化字段时，例如组合使用属性和串行化，就可以使用这种类型的构造函数。下面的代码在 constructor_parameters.cs 中：

```
public class Employee
{
    public Employee()
    {
        Console.WriteLine("Default constructor called with no params");
    }

    public Employee(string name, string dept, int salary)
    {
        this.name = name;
        this.dept = dept;
        this.salary = salary;
        Console.WriteLine("Constructor called with three params");
    }

    public Employee(string name, int salary)
    {
        this.name = name;
        this.salary = salary;
        Console.WriteLine("Constructor called with two params");
    }
}
```

现在需要添加一个 Main() 方法。在下面的例子中，创建了 3 个 Employee 对象，以利用新的构造函数。如果在 Visual Studio .NET 中使用断点查看执行的顺序，就会看到，

字段(专指 dept 字段)在调用构造函数之前进行了初始化。这样,就可以使用构造函数重写默认的字段的值,在设计期间指定字段的默认值,在运行期间,则在返回类实例之前指定最后的字段值。

```
public class TestEmployee
{
    static void Main(string[] args)
    {
        Employee emp1 = new Employee();
        Console.WriteLine("Name: {0}\nDept: {1}\nSalary: {2}",
                           emp1.Name, emp1.Dept, emp1.Salary);
        Console.WriteLine("-----");
        Employee emp2 = new Employee("Jones", "Operations", 30000);
        Console.WriteLine("Name: {0}\nDept: {1}\nSalary: {2}",
                           emp2.Name, emp2.Dept, emp2.Salary);
        Console.WriteLine("-----");
        Employee emp3 = new Employee("Smith", 35000);
        Console.WriteLine("Name: {0}\nDept: {1}\nSalary: {2}",
                           emp3.Name, emp3.Dept, emp3.Salary);
    }
}
```

这个程序的输出如下所示,与我们希望的相符。调用默认的构造函数,不会改变字段的初始值:

```
C:\Class Design\Ch 05>emp_constructors
Default Constructor
Name:
Dept: IT
Salary: 0
-----
Constructor called with three params
Name: Jones
Dept: Operations
Salary: 30000
-----
Constructor called with two params
Name: Smith
Dept: IT
Salary: 35000
```


5.2 销毁对象

对象的部分生命周期涉及到运行时销毁对象和回收内存, 尽管这个过程是自动进行的, 我们仍对这个控制感到惊讶。把对象引用设置为 `null`, 并注意不再有对对象的其他引用, 就可以显式销毁对象; 或者, 如果对象超出了其作用域, 运行时可以隐式销毁它。在这两种情况下, 引用数都做了修改, 当对象不再有引用时, 垃圾回收器就可以释放内存了。

如果应用程序中的对象数目非常小, 不需要 GC 释放内存, 而新对象的创建也不太多, GC 就不会压缩和释放内存。对象的销毁过程由 GC 执行, 完全取决于应用程序的内存可用性和需要。

在销毁对象时, 有一些机制让对象自己负责取消对内部对象的引用。这对于下放 CLR 不管理的底层资源的处理权非常有益。只要 GC 回收由对象占用的内存, 就会调用其 `Finalize()` 方法, 该方法会释放不由 CLR 管理的所有资源。可以把这个方法看做是运行时的中途拦截, 它阻止 GC 的跟踪, 在对象销毁和解除内存分配之前调用代码。

注意:

Java 开发人员很熟悉这个语法, 因为 Java 语言和 JVM 自 JDK 1.0 起就编写了这个机制。

重写 `System.Object` 中基类方法的实现, 就可以为该方法提供自己的实现。对于所有的 .NET 语言, 一般都可以通过重载名称和签名都相同的基类方法, 使用 `Finalize()` 方法, 但在 C# 和 C++ 中不行。在 .NET Framework SDK 的早期测试版本中, `Finalize()` 方法以前用于替代 C# 中的析构函数, 但现在已经废弃了。

`Finalize()` 方法的概念与 C# 和 Managed C++ 相同, 但使用的语法不同。C# 使用的语法称为析构函数, 用于清理资源, 这些资源一般应是未托管的资源, 例如文件句柄和数据库连接。例如在 `Employee` 类中, 如果创建了一个数据库连接, 在回收 `Employee` 时要关闭它, 取消对它的引用, 就可以创建一个析构函数, 由 GC 在销毁对象时调用。

下面的例子中创建了一个未赋值的成员变量。它表示要使用的文件句柄, 析构函数会在以后清理它:

```
using System;
using System.IO;

public class Employee
{
    FileStream fStream;
```

在无参数的构造函数中，添加下面的代码，写入到一个文件中。在对象的生命周期的任何阶段，都不会关闭或取消对 `fStream` 字段的引用：

```
public Employee()
{
    fStream = File.CreateText("C:\\temp.txt");
    StreamWriter sw = new StreamWriter(fStream);
    sw.Write("Default constructor called with no params");
}
```

为了调用这个新构造函数，可以重复使用上面的代码，作为 `Main()` 方法的一部分，该方法创建了 3 个 `employee` 对象。终止应用程序，将根据 GC 为每个 `employee` 对象释放内存的顺序，陆续为每个 `employee` 对象调用析构函数。

下面的析构函数代码放在 `try...catch` 块中，如果对象是使用其他构造函数创建的，`fStream` 就会设置为 `null`，所以需要捕获异常：

```
~Employee()
{
    try
    {
        fStream.Close();
        fStream = null;
    }
    catch {}
}
```

我们对是否需要对所有 3 个对象有少量的控制权。例如，不需要为不在文件中记录任何信息的对象调用析构函数，因为这会生成一个异常，终止析构函数的执行。在参数化的构造函数代码中，可以添加下面的代码，以确保 GC 不为这个对象实例调用析构函数。但是，在产品应用程序中，应在析构函数内部进行适当的 `null` 引用检查。

```
GC.SuppressFinalize(this);
```

有时需要修改这个策略，让 GC 提供一种等价的反方法，允许析构函数在回收对象时调用。

```
GC.ReRegisterForFinalize(this);
```

尽管 GC 知道何时回收托管堆上所有的对象，但有时 GC 不执行必要的回收操作。应用程序可能有很少的对象，但这些对象包含对未托管资源的许多引用。此时可以让 GC 执行回收操作，释放更多的内存，否则就不要这么做。这是通过调用 `Collect()` 方法

来完成的，但在大多数情况下，不需要这么做，因为这常常会对应用程序的性能有负面的影响。但是，也有一些特殊的情况：在托管堆上没有分配许多内存，几个较大的对象包含对未托管资源的许多引用，但没有在需要回收内存时强制执行回收操作。在这种情况下，可以调用 `GC.Collect()`，提高应用程序的性能。

`GC.Collect()`

最好不要使用析构函数，因为它很耗费性能，但在对象被回收之前，仍需要清理对象使用的资源。GC 建立了一个对象图，其中包含所有还未取消引用的对象，也就是说，这些对象的引用数大于 0，因此仍由应用程序代码使用。GC 没有记录在图中的所有对象，只要没有析构函数，都会被销毁。如果它们有析构函数，就会移动到一个特殊的队列 `freachable (F-Reachable)` 中，`F-Reachable` 是 `Finalization Reachable Queue` 的缩写形式，其中有一个专用的线程管理着所有的对象，在把它们标记为垃圾之前调用它们的析构函数。此时，在回收过程开始时，把带有析构函数的对象标记为垃圾后，对象的内存不会立即回收。因此，应用程序中的大量析构函数就会对性能有负面的影响，因为一个单独的线程会顺序调用每个析构函数，内存的回收就需要较长的时间。

处理对象销毁的首选方式是利用 `IDisposable` 接口。要使用 `IDisposable`，必须实现 `Dispose()` 方法，该方法可以以与 `Finalize()` 或析构函数方法相同的方式来执行，但不会中断垃圾回收过程，也不需要以相同的方式排队和管理。实际上，这两种清理资源方法的根本区别是，`Dispose()` 方法可以由调用代码随时调用，而不会对应用程序的性能有任何影响。因此最好使用 `Dispose()` 方法释放所有的资源，在涉及释放未托管资源的类中，也应习惯于实现 `IDisposable` 接口。

`Dispose()` 是一种约定，一般用于使用了许多未托管资源的对象，例如对 `OleDb` 库的调用或文件句柄。通常 `Dispose()` 方法声明为 `virtual`，并在派生类中重写。在派生类中，调用基类方法可确保派生于基类的所有资源也被销毁。

下面的例子根据一个图形定义了一个非常简单的继承层次结构。这里的要点是使用了 `Dispose()` 方法。在基类 `BasicShape` 中，实现了 `IDisposable` 接口，并为它的一个方法 `Dispose()` 编写了实现代码。调用该方法，就会释放类所使用的 IO 资源。

我们没有直接创建 `BasicShape`，而是创建了一个派生类 `Square`，它重写了 `GetCoords()` 和 `Dispose()` 方法。有了这两个派生的类方法，`GetCoords()` 和 `Dispose()`，`Square` 类就有了更完整的类定义，包括专用于派生类中 `GetCoords()` 方法的代码——使用示例 `a` 和 `b` 字段的对象（如果 `x` 和 `y` 表示一个点，则 `a` 和 `b` 表示正方形的对角点）。派生类上的 `Dispose()` 方法可以用于清理所有的资源。这些代码包含在 `basic_shape.cs` 中：

```
using System;  
using System.IO;
```

```
public abstract class BasicShape : IDisposable
{
    protected int x;
    protected int y;
    private FileStream fStream;

    public virtual void GetCoords()
    {
        //possibly open a file handle here
        try
        {
            StreamReader sr =
                new StreamReader((fStream = File.OpenRead("C:\\temp.txt")));
            int x = Convert.ToInt32(sr.ReadLine());
            int y = Convert.ToInt32(sr.ReadLine());
        }
        catch {}
    }

    public virtual void Dispose()
    {
        if(fStream!=null)
        {
            fStream.Close();
            fStream = null;
        }
    }
}

public class Square : BasicShape
{
    int a = 0;
    int b = 0;

    public override void GetCoords()
    {
        /** Do something associated with this class and then call the base
            class method which will change inherited fields **/
        base.GetCoords();
    }
}
```

```
public override void Dispose()
{
    //clean up any resources associated with this object
    base.Dispose();
}
}
```

取消对派生类上资源的引用时，对象一般不应抛出异常，因为这会妨碍对类层次结构上的每个 `Dispose()` 方法的调用。在调用 `BasicShape` 类的基方法之前，以及在可能出现可预测异常的地方，必须为 `Square` 类编写清理代码，这样整个过程才不会失败。对于非常大的继承层次结构，为每个基类调用 `Dispose()` 方法是非常重要的，如果 `Dispose()` 的调用链被中断，就会有一些资源未被释放。下面是在 `Main()` 方法中用于测试这些情况的代码：

```
Square square = new Square();
square.GetCoords();
square.Dispose();
square = null;
```

MSDN 上的许多例子组合使用了析构函数和 `IDisposable` 接口，来确保调用 `Dispose()` 方法(但实际上不必使用析构函数)。如果我们也要这样做，可以在基类的 `Dispose()` 方法的最后插入一行 `GC.SuppressFinalize(this);`，把析构函数设置为调用 `Dispose()`。此时，如果没有调用 `Dispose()`，析构函数就会调用它。如果调用了 `Dispose()`，就不会调用析构函数，内存就可以在垃圾回收之后立即收回。

方法 `Dispose()` 可以用许多不同的技术来调用，例如可以编写自己的托管类 `GenericFile` 来实现 `IDisposable`。可以使用 `CreateFile` API，通过使用 `P/Invoke` 获取未托管代码中的文件句柄，再在调用 `Dispose()` 的类中实现 `Close()` 方法。`Dispose()` 会使用 `CloseHandle` API 释放未托管的资源。这是文件生命周期的一部分，所以，使用类的用户将显式调用 `Close()`，这样就调用了 `Dispose()`，释放所有的资源。实际上，.NET Framework 类库中的所有类都按这样的方式操作，自动调用 `Dispose()` 方法。

5.3 使用构造函数

可以使用许多方式来设计类，使之使用构造函数，同时使要编写的代码量最少，功能最强，类的实现也比较简单。前面说过，可以显式编写一个构造函数，初始化字段。无论是否在构造函数中初始化字段的值，运行时都会在调用构造函数之前把字段初始

化为其默认值。如果不为类编写构造函数，编译器就会创建一个默认的构造函数，运行时会在创建新对象时调用它。

5.3.1 把构造函数链接在一起

在编写构造函数代码时使用标准方式，可以使要编写的代码量减到最少，代码的可读性比较高，并避免在同一个类中重复类似的代码块。与提供和使用一个构造函数的实现代码相比，把构造函数链接在一起是建立高级接口和多个重载构造函数的一种高效方式。

稍后详细论述的 `Manager` 类是使用构造函数链从一个构造函数中调用另一个构造函数的类的一种典型实现方式。有 3 个 `Manager` 构造函数，最后一个构造函数包含两个参数，是惟一带有实现代码的构造函数。其他两个构造函数首先要做的是调用第三个构造函数。这是使用当前对象的 `this` 自我引用指针来实现的。在 C# 中，这是新的语法，但 C++、JScript 和 Java 开发人员应很熟悉 `this` 的用法。在调用两个参数的构造函数时，会用指定的默认值填充字段，接着调用构造函数代码块中的其他实现代码。在默认的无参数构造函数中还有其他代码，以确保每次创建一个没有名字的 `Manager` 类时都会做记录(项添加到事件日志中)。下面显示的文件是 `chaining_const.cs`：

```
using System;

public class Manager
{
    private int grade;
    private string name;

    public int Grade
    {
        get { return grade; }
        set { grade = value; }
    }

    public string Name
    {
        get { return name; }
        set { name = value; }
    }
}
```

```
public Manager() : this("N/A", 0)
{
    Console.WriteLine("Manager created without name!");
}

public Manager(string name) : this(name, 0) {}

public Manager(string name, int grade)
{
    Name = name;
    Grade = grade;
}

public override string ToString()
{
    return "Name: " + this.Name + ", Grade: " + this.Grade.ToString();
}
}
```

在下面的 Main() 中，调用了包含一个参数的构造函数，并给实现构造函数传送名称 Jones 和默认的等级 0：

```
public static void Main()
{
    Manager mgr_Jones = new Manager("Jones");
    Manager mgr_Blank = new Manager();
    Manager mgr_Name_Grade = new Manager("Jones", 4);
    Console.WriteLine(mgr_Jones.ToString());
    Console.WriteLine(mgr_Blank.ToString());
    Console.WriteLine(mgr_Name_Grade.ToString());
}
```

5.3.2 调用基类构造函数

每个派生类都隐式或显式地调用基类的构造函数。实际上，如果我们没有调用基类的构造函数，CLR 就会调用它(CLR 会调用无参数的构造函数)。基类构造函数只能从派生类的构造函数中调用，在调用它之后，就会执行派生类的构造函数代码块中的实现。

可以在前面创建的 `BasicShape` 抽象类中添加两个构造函数，把它们标记为受保护的，这样就只有派生类能调用它们。还可以把它们标记为公共的，这没有什么区别，因为基类是抽象类，不能实例化。

```
public abstract class BasicShape : IDisposable
{
    protected BasicShape()
    {
        x = 1;
        y = 1;
    }

    protected BasicShape(int x, int y)
    {
        this.x = x;
        this.y = y;
    }
    ...
}
```

在派生类中，可以添加一些代码，调用基类的构造函数。这些方法会在 `base` 关键字的前面加上冒号，来调用基类构造函数，如下所示。这个构造函数会在调用派生类的构造函数之前调用。在下面的例子中，不必总是在基类上调用与派生类的构造函数有相同签名的构造函数。没有参数的 `Square()` 构造函数将调用有两个参数的基类构造函数：

```
public Square() : base(4, 4)
{
    this.a = 1;
    this.b = 1;
}

public Square(int a, int b) : base(4, 4)
{
    this.a = a;
    this.b = b;
}
```


5.3.3 私有构造函数

私有构造函数初看起来是无意义的，但在.NET Framework 类中，它们的功能是相当多的。它们一般用在包含许多静态方法的类中，这些静态方法用作一个库，而不是对象。添加构造函数将确保类不能在外部创建。

创建私有构造函数有两个原因，有时在使用公共构造函数时，不能使用类创建代码。一种情况是，如果不提供初始化参数，创建对象是没有意义的，例如 `FileStream` 类。

第二个原因是创建某种对象设计模式，例如工厂模式，即类上的公共静态方法要调用私有构造函数，并把对象的一个实例返回到对象创建代码，还有单一模式，即构造函数只能调用一次。本章的后面将详细论述这些问题。

在下面的例子中，`Teacher` 类使用一个公共构造函数来创建，如果把小于 3 的等级传送给该构造函数，就会抛出一个异常。它不使用内联代码或另一个方法，而是使用私有构造函数来生成异常。这个例子进行了大量的简化，因为私有构造函数包含额外的初始化代码。在这个例子中，外部的代码不能使用无参数的构造函数创建 `Teacher` 类的实例。

```
using System;

public class Teacher
{
    int grade = 0;

    public Teacher(int grade)
    {
        if(grade < 3)
            new Teacher();
        else
            throw new ArgumentException("Grade must be less than 3");
        this.grade= grade;
    }

    private Teacher()
    {
        // Initialization code
    }
}
```

```
public class TestTeacher
{
    static void Main()
    {
        Teacher teach = new Teacher(3)
    }
}
```

前面介绍过，无论指定与否，派生类总是要调用基类的构造函数。这表示，带有私有构造函数的类不能用作基类，因为它们的构造函数对派生类来说不可见。

当另一个类嵌套在带有私有构造函数的类中时，包含私有构造函数的类就可以由其他对象创建。无论构造函数是不是私有的，嵌套的构造函数总是可以创建其容器类的实例。

下面的 `NestedWorker` 类嵌套在 `Worker` 类中。`NestedWorker` 类有一个公共构造函数，`Worker` 类有一个私有构造函数，`Worker` 类会为它创建一个引用，并在本地存储。所以，可以在被嵌套的类 `NestedWorker` 中创建并调用容器类 `Worker` 的公共构造函数，如下面的 `Main()` 方法所示，这些代码在 `nested_const.cs` 中：

```
using System;

public class Worker
{
    public class NestedWorker
    {
        Worker wk;
        public NestedWorker()
        {
            wk = new Worker();
        }
    }

    private Worker()
    {
        Console.WriteLine("private constructor called!");
    }

    static void Main(string[] args)
    {
        NestedWorker nw = new NestedWorker();
    }
}
```



```
    }  
}
```

5.3.4 静态构造函数

静态构造函数可以在 C# 中用于初始化类数据，其方式与用于初始化实例数据的实例构造函数一样。静态构造函数与实例构造函数在使用规则方面有一些区别。与实例构造函数不一样，静态构造函数不能重载，所以可用的静态构造函数只有一个默认的无参静态构造函数。静态构造函数也不能显式调用，不能在派生类中继承，但在创建基类类型时可以调用。

下面是 .NET 在使用静态构造函数时的几个原则：

- 静态构造函数在创建类的实例之前调用，因此在所有实例构造函数之前调用。
- 静态构造函数在创建类的第一个实例之前调用。
- 静态构造函数在引用静态字段之前调用。

为了检查静态构造函数和实例构造函数的调用顺序，可以给上一个例子中创建的 `Teacher` 类添加一个静态构造函数和一个公共的实例构造函数。使用 `static` 关键字定义静态构造函数，把它与实例构造函数区分开来。这个静态构造函数允许初始化类中的所有静态成员。Java 的静态构造函数与 C# 有类似的原则，即在声明时不需要引用类名，所以静态构造函数的声明一般是在 `static` 关键字的后面跟有代码块。下面的例子是 `static_const.cs`：

```
...  
public class Teacher  
{  
    public Teacher()  
    {  
        Console.WriteLine("Base class instance created");  
    }  
  
    static Teacher()  
    {  
        Console.WriteLine("Class loaded");  
    }  
...  
}
```

接着创建派生于 `Teacher` 类的 `SupplyTeacher` 类，首先调用静态构造函数，再调用基类的实例构造函数：

```
public class SupplyTeacher : Teacher
{
    public SupplyTeacher ()
    {
        Console.WriteLine("Derived class instance created");
    }
}

public class TestStaticConstructor
{
    static void Main(string[] args)
    {
        SupplyTeacher teach = new SupplyTeacher();
        SupplyTeacher teach1 = new SupplyTeacher();
    }
}
```

在创建 `SupplyTeacher` 类的实例时，会生成下面的输出，确认每个构造函数类型的调用顺序。在本例中，创建了两个 `SupplyTeacher` 对象，来说明在相同的对象创建模式中，静态构造函数只调用一次，而基类的实例构造函数调用了两次。

```
C:\Class Design\Ch 05>static_const
Class loaded
Base class instance created
Derived class instance created
Base class instance created
Derived class instance created
```

5.4 转换运算符

前面介绍了对象的生命周期和创建与删除对象的不同方式，特别定义了实现构造函数和析构函数的方式。还有另一种类型的对象构造方式需要讨论。本节就讨论不同引用类型之间的运算符转换和它们对调用构造函数的影响。

在值类型之间有一种转换机制，允许把 `int` 转换为 `long` 类型，或把 `short` 转换为 `int` 类型等。转换运算符可以在值类型和引用类型之间进行转换，这种转换是隐式的还是显式的，取决于需要的转换类型。隐式转换与显式转换的区别是，隐式转换不需要用圆括号指定类型的转换。下面的转换是隐式的，因为其中使用了赋值语句，把类型从

Bus 转换为 BigVehicle，但没有直接在代码中指定类型转换。

```
Bus bus;  
BigVehicle big = bus;
```

而在显式转换中，必须显示定义两个类型之间的转换。下面的例子把 BigVehicle 结构类型转换为 Bus 类类型(类和结构将在本节的最后讨论):

```
BigVehicle big;  
Bus bus = (Bus)big;
```

为了清晰起见，下面的例子演示了隐式和显式转换运算符的用法，但我们更感兴趣的是显式转换运算符，因为使用这种机制可以定义一种新类型的对象构造方式。转换运算符也可以用于在类和结构值之间转换，反之亦然。

为了说明转换运算符的用法，下面回到 Teacher 类上，添加一个简单的结构 BigVehicle，再添加转换运算符，这样就可以在这两种类型之间进行隐式和显式转换了。

在用户定义的转换中需要使用两个新的关键字：implicit 和 explicit。这些方法必须创建为静态方法，必须用 operator 关键字指定重写了语言运算符的行为。implicit 运算符将创建一个新的 BigVehicle 结构，并把所有的字段值从 Bus 类实例复制到结构中，这是使用 bus 参数参数实现的。explicit 运算符则使用 big 参数，调用 Bus 类的构造函数，通过构造函数的初始化过程，把 BigVehicle 值参数复制到 Bus 字段中，来创建 Bus 类的新实例。下面的代码在 explicit_conversion.cs 中，转换运算符则在程序清单的最后：

```
using System;  
  
public struct BigVehicle  
{  
    public string NumberPlate;  
    public bool inService;  
    public int seats;  
}  
  
public class Bus  
{  
    string numberPlate;  
    bool inService;  
    int seats;  
  
    public string NumberPlate  
    {
```

```
        get { return numberPlate;}
        set { numberPlate = value; }
    }

    public bool InService
    {
        get { return inService;}
        set { inService = value; }
    }

    public int Seats
    {
        get { return seats;}
        set { seats = value; }
    }

    public Bus(string numberPlate, int seats, bool inService)
    {
        this.numberPlate = numberPlate;
        this.seats = seats;
        this.inService = inService;
    }

    public static implicit operator BigVehicle(Bus bus)
    {
        BigVehicle big;
        big.inService = bus.NumberPlate;
        big.seats = bus.Seats;
        big.inService = bus.inService;
        return big;
    }

    public static explicit operator Bus(BigVehicle big)
    {
        return new Bus(big.NumberPlate, big.seats, big.inService);
    }
}
```

要在 Bus 对象和 BigVehicle 结构值之间转换,并调用隐式运算符,可以使用 Main() 方法中的下述代码。BigVehicle 结构在代码中创建,并用 Bus 对象的属性填充其字段值:

```
Bus bus1 = new Bus("Eazy Rider", 20, true);  
BigVehicle big1 = bus1;
```

同样,调用显式转换运算符,就可以在 `BigVehicle` 和 `Bus` 之间进行显式的类型转换。在调用运算符时,会创建 `Bus` 类的一个新实例,并把 `BigVehicle` 值传送给带三个参数的构造函数(值从 `BigVehicle` 字段中提取)。以这种方式使用显式运算符,可以把 `Bus` 构造函数定义为私有。也就是说,不能显式地创建 `Bus` 类的新实例,但可以提供一个转换接口,这样,创建新 `Bus` 对象的惟一方式就是把 `BigVehicle` 转换为 `Bus`(或把其他类型转换为定义了显式运算符的 `Bus`)。

```
Bus bus2;  
BigVehicle big2;  
big2.inService = true;  
big2.NumberPlate = "Eazy Rider";  
big2.seats = 20;  
bus2 = (Bus)big2;
```

运算符的使用不怎么受限制,这就意味着,可以定义一个代码块,把任意值类型转换为任意引用类型,或把任意引用类型转换为任意值类型。但是,每个类型只能定义一个运算符,因为编译器不能确定应使用哪个运算符进行指定的转换。

在结构和类之间进行隐式的类型转换,允许为引用类型定义值类型语义。如果要定义一个 `Code` 类型,它采用一个字母后跟一个数字的格式,就可以使用下面的代码来定义这种类型:

```
using System;  
  
public class Code  
{  
    private string code = String.Empty;  
  
    public Code(string code)  
    {  
        if(code!=null && code.Length == 2  
            && char.IsLetter(code[0]) && char.IsNumber(code[1]))  
            this.code = code.ToUpper();  
        else  
            throw new ArgumentException("Not a valid code");  
    }  
    public static implicit operator Code(string address)  
    {
```

```
        return new Code(address);
    }

    public static explicit operator string(Code code)
    {
        return code.code;
    }

    public override string ToString()
    {
        return (string)this;
    }
}
```

可以为这个类型显式输入代码(`Code myCode = "C3";`)，如果在集合中使用这个类型，就不必执行任何有效性检查，来确保代码的格式正确。重写显式的字符串转换运算符和 `ToString()` 方法，这个类型就会像我们希望的那样用作字符串。但这么做应小心，因为这个类型太像字符串了，可能会使代码的用户混淆它们。

转换运算符还可以在基类中用于在两个类型之间转换，也可以用于派生类(但因为它们都是静态的，不能继承，所以派生类不能使用基类的转换运算符)。下面的代码假定，`BigBus` 类派生于 `Bus` 类，但使用 `Bus` 类型对它的引用。即使 `BigBus` 类有自己的转换运算符，也会使用 `Bus` 类的隐式转换运算符。

```
Bus bus = new BigBus(40);
BigVehicle i = bus;
```

5.5 对象的克隆

克隆对象要求在不影响原对象及其内容的情况下，在对象的副本中顺利工作。系统以许多不同的方式使用克隆，但克隆一般用于把对象初始化为默认状态。本节主要讨论克隆对象的不同方式，以及它与对象构造的关系。

5.5.1 使用复制构造函数

克隆对象的需要导致了复制构造函数的诞生，这个构造函数把它自己的类类型作为参数。这样，类实例参数就可以用于填充对象当前实例的字段值。

下面编写的 `Teacher` 类包含一个复制构造函数，在类的每个新实例中，该构造函数都可以传送一个对象引用，来初始化 `Teacher` 类的所有新的类实例。这里可以采用许多不同的方式：类实例可以用于设置所有后续类实例的初始化；而在 `Teacher` 类中，我们创建几个类实例，每个实例用于一个部门，并把它们传送给要创建的 `Teacher` 对象的子集的复制构造函数。不必把该构造函数用作模板，而可以只把它用于复制类的某些数据，这些数据对把特定对象初始化为默认状态非常重要。

```
public class Teacher
{
    private int grade;
    private int salary;
    private string dept = String.Empty;

    public Teacher() {}

    public Teacher(int grade, int salary, string dept)
    {
        this.grade = grade;
        this.salary = salary;
        this.dept = dept;
    }
    public Teacher(Teacher teach)
    {
        grade = teach.grade;
        salary = teach.salary;
        dept = teach.dept;
    }
}
```

要调用复制构造函数，可以在 `Main()` 方法中使用下面的代码。这个例子的代码文件是 `copy_ctors.cs`。它首先用要复制的默认值创建一个 `Teacher` 对象，再把对象引用传送给复制构造函数，并使用值在还未初始化的类实例中填充字段。

```
Teacher teach1 = new Teacher(3, 23000, "IT");
Teacher teach2 = new Teacher(teach1);
```

在 .NET Framework 中引入了反射机制，所以很容易开发一般的复制构造函数，通过识别类型中的可用属性值，来反射一个类型，并提取某个对象的值。可以使用反射机制动态地查找类型的公共接口的所有信息，并把值添加到类的当前实例中。要使用反射机制，必须给对象添加公共属性值，或把字段声明为公共的。这里选择前一种

方式:

```
public int Grade { get { return grade; } set { grade = value; }}
public int Salary { get { return salary; } set { salary = value; }}
public string Dept { get { return dept; } set { dept = value; }}
```

Teacher 类的复制构造函数的当前实现代码可以由下面的代码替代。这些代码允许进一步抽象,从类的一个实例中提取出所有的属性值,并填充到当前实例的属性值中。这样,只要有可公共访问的属性或字段值,Teacher 类上的接口就可以修改(即可以给类添加更多的属性,但不再更新代码,因为其余属性都是动态的),而不必更新复制构造函数。下面使用的类可通过引用 System.Reflection 命名空间来获得:

```
public Teacher(Teacher teach)
{
    PropertyInfo[] pInfoArray = teach.GetType().GetProperties();
    foreach(PropertyInfo pInfo in pInfoArray)
    {
        Type ty = pInfo.PropertyType;
        object myvalue =
            teach.GetType().InvokeMember(pInfo.Name,
                                           BindingFlags.GetProperty, null,
                                           teach, null);
        teach.GetType().InvokeMember(pInfo.Name,
                                      BindingFlags.SetProperty, null,
                                      this, new object[] {myvalue});
    }
}
```

在新的 Teacher 类实例上调用 get 属性访问器,将值从一个对象复制到另一个对象上,但没有指定字段或属性的命名:

```
Console.WriteLine("Details:{0}, {1}, {2}",
    teach2.Grade, teach2.Dept, teach2.Salary);
```

5.5.2 ICloneable 接口

.NET Framework 中的许多类都实现了 ICloneable 接口,它可以用于替代复制构造函数,来获取对象的克隆。ICloneable 提供了一个必须实现的 Clone()方法。下面通过

实现 `ICloneable` 类，以修改 `Teacher` 类的定义：

```
public class Teacher : ICloneable
```

现在必须实现 `Clone()` 方法，返回 `Teacher` 类的一个复制了的实例。为了符合 `ICloneable` 接口，`Clone()` 方法必须返回对象类型。使用 `ICloneable` 是非常有用的，因为使用 `ICloneable` 可以建立能测试是否支持克隆的类，即 `ICloneable` 接口，然后以普通的方式转换它的类型。之后，应用程序就可以通过一个接受 `ICloneable` 接口类型(而不是具体的类类型)的方法返回对象。

```
    public object Clone()
    {
        return new Teacher(grade, salary, dept) as object;
    }
```

调用 `Clone()` 方法会返回一个 `object` 类型的 `Teacher` 对象，该对象的类型可以转换回 `Teacher`。

```
Teacher teach3 = (Teacher)teach2.Clone();
```

上面就是一个深拷贝的例子。它比相等浅拷贝的代价高一些。深拷贝会拷贝整个填充的对象，包括该对象中其他引用类型和值类型的值。而浅拷贝只复制了一个对象中的所有引用，也就是复制了在另一个对象中创建、初始化和使用的类型的引用。它没有这些值的拷贝，通过引用它们的其他对象的引用来共享它们。对类型的每个引用都会使该类型的引用数递增 1，因此过度使用浅拷贝将禁止 GC 删除对象。每个对象都从 `System.Object` 中继承了一个方法 `MemberWiseClone()`，该方法返回对象的一个浅拷贝。使用对象的浅拷贝的主要局限是，如果源对象中的值发生了变化，就会在内存中对所有相同的值进行第二次复制引用，而且所有的改变都是隐式的，并影响克隆的对象。成员克隆会导致浅拷贝，但某些内容是按值拷贝的，这与值类型一样。

```
    public object Clone()
    {
        return this.MemberwiseClone();
    }
```

深拷贝是一个非常昂贵的操作，应尽量避免使用，因为深拷贝会按值拷贝所有的成员字段。这涉及到许多对象的创建，代价非常高，而实际上可以组合使用这两种拷贝类型。对于包含对许多其他对象或对象集合的引用的对象，使用深拷贝效果比较好。

5.6 串行化和并行化

串行化是 Java 开发人员非常熟悉的一个概念，它用于创建一个对象的表示，说明对象的状态可以在什么地方保存和传输；对象的格式化表示可以是字节流，或 XML 文档。可以把对象串行化为许多类型的 Stream 成员，例如 FileStream 或 NetworkStream。对象在串行化后，还可以在以后通过并行化过程恢复。并行化是使用格式化的值，重新构造它，来构成一个 .NET 对象。格式化程序决定串行化的规则，来指定要串行化的对象有什么外观。实际上，我们很少需要除 .NET Framework 提供之外的其他格式化程序，.NET Framework 提供的格式化程序有二进制格式化程序、XML 格式化程序，和 SOAP 格式化程序。二进制格式化程序在串行化一个对象时，会把该对象的所有字段值转换为可以写入一个文件或网络终端点的字节流。XML 格式化程序则把对象格式化为可以写入任何流中的 XML 文档表示。SOAP 格式化程序把对象串行化为一个基于 XML 的协议，即简单对象访问协议(目前的标准是 1.1 版本)，该协议可用于调用 Web 服务方法。

使用并行化过程，可以从串行化的源(例如磁盘上的文件或网络流)中创建同一对象的几个副本。从概念上讲，对象串行化为流，会使编程模型变得简单而抽象，这对于对象在网络上传输非常有利。使用套接字，可以在客户端通过使用二进制格式化程序格式化对象，在服务器上重新创建相同的对象。

这个例子要把一个对象串行化到磁盘文件上，然后使用并行化操作重新构造对象。必须使用下面的命名空间来串行化对象。为了访问文件和流，还必须使用 System.IO 命名空间。本例将包含 SOAP，说明串行化时会产生什么输出，但实际上，也可以根据具体的情况进行选择。例如，二进制格式化程序会使对象表示所包含的字节非常少，在需要提高性能时就可以选择二进制格式化程序。SOAP 格式化的对象是一个标准，可以用另一种编程语言，如 Java，重新构造，这将使对象串行化的交互操作性更强。把下面的内容添加到 copy_ctors.cs 文件中。新的代码文件称为 serializable_deserializable.cs:

```
using System.Runtime.Serialization;
/*we won't be using the Binary Formatter but if you need to use it you must define the following
using statement*/
using System.Runtime.Serialization.Formatters.Binary;
using System.Runtime.Serialization.Formatters.Soap;
```

在前面讨论成员复制时，创建了 teach 引用：

```
Teacher teach3 = (Teacher)teach2.Clone();
```

可以在 Teacher 类的定义中添加 Serializable 属性，指定可以串行化的类。这是必

须的，因为在使用格式化程序串行化对象时，必须检查对象中是否有这个属性，如果该属性不存在，就抛出一个 `SerializationException` 异常。

`[Serializable()]`

```
public class Teacher : ICloneable
```

下面的代码包含在 `Main()`方法中，如果 `teacher.xml` 文件不存在，就创建它，并返回一个 `FileStream` 对象，把 `Teacher` 对象串行化到磁盘上。这个过程是使用 `SoapFormatter` 对象完成的。把 `FileStream` 和 `Teacher` 对象作为参数，调用 `SoapFormatter` 类的 `Serialize()` 方法。

```
FileStream fTeacher = File.Open("C:\\teacher.xml",  
                                FileMode.OpenOrCreate,  
                                FileAccess.ReadWrite);  
  
SoapFormatter sfTeacher = new SoapFormatter();  
sfTeacher.Serialize(fTeacher, teach3);
```

在本例中，对象的串行化和并行化使用了相同的流，也就是说，必须确保流指针重新设置为指向开始，否则就会抛出一个异常。

```
fTeacher.Position = 0;
```

`Deserialize()`方法使用打开的 `FileStream`，从流中读取所有的内容，并重新创建为一个 `Teacher` 对象。然后打印出其字段值，看看对象是否正确创建。

```
Teacher teach4 = (Teacher)sfTeacher.Deserialize(fStream);  
Console.WriteLine("Details:{0}, {1}, {2}",  
                  teach4.Grade, teach4.Dept, teach4.Salary);
```

整个过程都可以用于从磁盘的文件中重新构造 `Teacher` 对象实例。从上面的代码中可以看出，不必使用构造函数重新创建 `Teacher` 对象，而可以调用 `Deserialize()`方法从 `FileStream` 中读取，重新转换为 `Teacher` 类型。这个技术广泛应用于恢复应用程序的状态，以便定期获取屏幕快照，保存对象，因为使用对象数据库保存对象在开发团体中获得了广泛的应用，与可串行化的 Java EJB Entity Beans 一样。使用串行化不仅仅是克隆的一种替代方式，也是对克隆的一种补充，克隆可以用于随时复制对象实例，而串行化可以用于保存或传输对象实例，并在应用程序的外部重新构建它。

我们对串行化过程的控制权和克隆过程相同，并可以方便地在可串行化的类中实现。字段值可以标记为是否应串行化或是否不应串行化。其方法非常简单，只需把另一个属性添加到不应串行化的字段上，如下所示：

`[NonSerialized]`

```
private string dept = String.Empty;
```

这里串行化的 Teacher 对象是以串行化的形式重新构建。包含在其中的许多信息超出了本章的范围，与串行化的理解也无关。这里重新构建它是为了说明在上述 dept 字段中使用的 NonSerialized 属性的作用。在下面的 SOAP XML 中有一个标记 al:Teacher，在这个标记中还嵌套了两个标记 grade 和 salary，它们包含这些字段的值。注意 dept 字段没有重新构建，因为它标记为 NonSerialized。要生成这些 XML 代码，可编译并运行 serializable_deserializable.cs 文件。在命令提示窗口中输入下面的内容：

```
C:\Class Design\Ch 05> type \teacher.xml
<SOAP-ENV:Envelope
  xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:SOAP-ENC=http://schemas.xmlsoap.org/soap/encoding/
  xmlns:SOAP-ENV=http://schemas.xmlsoap.org/soap/envelope/
  xmlns:clr="http://schemas.microsoft.com/soap/encoding/clr/1.0"

  SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
<SOAP-ENV:Body>
  <al:Teacher id="ref-1"
    xmlns:al=
      "http://schemas.microsoft.com/clr/nsassem/
        CloningAndSerialization/CloningAndSerialization
        %2C%20Version%3D1.0.1056.43019%2C%20Culture
        %3Dneutral%2C%20PublicKeyToken%3Dnull">
    <grade>3</grade>
    <salary>23000</salary>
  </al:Teacher>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

除了这些之外，.NET Framework 还提供了一个丰富的 XML 串行化模型，它的使用方式与对象串行化相同。

下面是一个非常简单的 XML 文档，它保存到文件 computer.xml 中。从这个文档中可以推断出 XML 模式，它可以看做是 XML 文档的模板。同样，可以推断出 XML 模式和 XML 文档之间的关系与类和对象的关系类似。

```
<COMPUTER>
  <HARDDRIVE>MAXTOR</HARDDRIVE>
  <GRAPHICSCARD>ATI</GRAPHICSCARD>
  <SOUNDCARD>CREATIVE LABS</SOUNDCARD>
```

```
</COMPUTER>
```

.NET Framework 提供了一个工具 `xsd.exe`，它可以用于推断 XML 模式，再从该模式中生成一个 C# 类文件：

```
C:\Class Design\Ch 05>xsd computer.xml
```

从模式中可以使用 `/c` 选项派生 C# 类：

```
C:\Class Design\Ch 05>xsd /c computer.xsd
```

上面创建一个类 `COMPUTER`，它是通过下面自动生成的代码从文件 `computer.cs` 中创建的。类声明上面的属性与 `Serializable` 属性同义。

```
using System.Xml.Serialization;

[System.Xml.Serialization.XmlTypeAttribute(
    Namespace="http://tempuri.org/computer.xsd")]
[System.Xml.Serialization.XmlRootAttribute(
    Namespace="http://tempuri.org/computer.xsd", IsNullable=false)]
public class COMPUTER
{
    public string HARDDRIVE;
    public string GRAPHICSCARD;
    public string SOUNDCARD;
}
```

要使用 `computer.xml` 中存储的对象，可以使用 `System.Xml.Serialization.XmlSerializer` 类把文件的内容并行化为一个类型为 `COMPUTER` 的对象。要检查这个过程是否成功进行，可以输出一个字段值，如下所示：

```
XmlSerializer xs = new XmlSerializer(typeof(COMPUTER));
COMPUTER cmp =
    (COMPUTER)xs.Deserialize(File.Open("computer.xml",
    FileMode.Open));
Console.WriteLine("Graphics card is: {0}", cmp.GRAPHICSCARD);
```

5.7 设计模式

前面讨论了对对象构建的基本原则，如何改变使用对象创建过程，包括使用运算符、构造函数重载，构造函数链和静态构造函数，来初始化类构造函数。对对象创建原则

有了大致的了解后，下面阐述一个比较抽象的问题。

设计模式是一个可重复使用的设计，它可以把解决问题的面向对象方式变成一个模板。这些模式可用于为设计创建抽象层，它们一般应与面向对象开发技术一起学习，发展好的规则，摒弃不良的开发规则。

许多人都提出过设计模式，而发起者是 Erich Gamma、Richard Helm、Ralph Johnson 和 John Vlissides，即所谓的 Gang of Four。Gang of Four 的模式在计算机领域应用非常广泛，常常缩写为 Gof Patterns。Gang of Four 创建了一系列可重用的面向对象设计模式，由开发团体和大多数面向对象的项目采用。更多的信息可参阅 Addison & Wesley 出版的“Design Patterns: Elements of Reusable Object-Oriented Software” (ISBN: 0-201-63361-2)，该书由 Gang of Four 编著，也常称为 GOF 图书。在深入探讨几个可以在本书的代码中使用的设计模式时，显然对所有精通 .NET Framework 类库或 Java 类库的开发人员来说，这些设计模式可以在整个库代码中使用。本节介绍的模式描述为创建模式，因为它们根据对象的创建来表达设计思想，而不是根据结构或行为模式来表达，结构或行为模式是根据类模型或对象之间的通信来表示设计思想。

5.7.1 Singleton 模式

Singleton 设计模式在大多数类库和面向对象模型中使用。Singleton 设计模式确保类只有一个实例是可用的，这是由一个方法定义的，该方法通过使用类中的静态帮助方法来控制。Singleton 设计模式在操作系统的设计中使用得很多，以确保在请求使用某个硬件设备时，使用该硬件设备对象的一个实例，而不是新建的实例。

下面的例子有一个类 HeadTeacher，它不能直接使用 new 关键字创建。只有指定一个私有的构造函数才能创建它，所以只有嵌套的类或静态方法才能创建 HeadTeacher 类的实例。前面已经讨论了嵌套的类方式，但这里在定义 Singleton 时使用静态方法。

静态字段 headTeacher 存储了 HeadTeacher 对象实例。通过 GetInstance() 静态方法来控制这个对象实例的创建，即检查它是否为 null。如果是，这个方法就创建并返回 HeadTeacher 的一个新实例，否则就返回静态字段引用。这样，就只存在一个 HeadTeacher 的对象，所有的客户程序都有该实例的一个引用。

```
using System;
```

```
public class HeadTeacher
{
    private static HeadTeacher headTeacher;
    private int grade;
    private string name;
```



```
private HeadTeacher() {}

public static HeadTeacher GetInstance()
{
    if(headTeacher == null)
    {
        Console.WriteLine("HeadTeacher instance created!");
        headTeacher = new HeadTeacher();
        return headTeacher;
    }
    else
    {
        Console.WriteLine("HeadTeacher instance already exists!");
        return headTeacher;
    }
}

public int Grade
{
    get { return grade; }
    set { grade = value; }
}

public string Name
{
    get { return name; }
    set { name = value; }
}
}
```

使用 `Main()` 方法检查是否创建了对象的一个实例，通过调用 `GetInstance()` 方法来创建一个 `HeadTeacher` 对象。可以把 `Grade` 属性设置为 10，它在两个对象引用中是相同的，因为它们引用的是同一个对象。包含所有源代码的文件是 `singleton.cs`：

```
public class mainHeadTeacher
{
    static void Main()
    {
        HeadTeacher ht1 = HeadTeacher.GetInstance();
```

```
HeadTeacher ht2 = HeadTeacher.GetInstance();  
ht1.Grade = 10;  
  
Console.WriteLine("ht1 Grade:{0}", ht1.Grade);  
Console.WriteLine("ht2 Grade:{0}", ht2.Grade);  
}  
}
```

其结果如下所示:

```
C:\Class Design\Ch 05> Singleton  
HeadTeacher instance created!  
HeadTeacher instance already exists!  
ht1 Grade:10  
ht2 Grade:10
```

通过修改上面的内容,应用第二个设计模式 Double-Checked Locking Pattern,以确保 Singleton 类是线程安全的。为了使所有的内容都是线程安全的,必须确保在任何时候都只有一个线程能进入方法。考虑下面的情况:一个线程进入 GetInstance()方法,而另一个线程在对象创建之前、在检查包含 HeadTeacher 对象的局部变量为 null 之后进入该方法。在控制权传送回原线程之前,另一个线程已经创建了对对象。假定调用代码是多线程的,则 h1 和 h2 就会引用不同的 HeadTeacher 对象。

可以通过下面的命名空间,使用.NET 内置的同步功能:

```
using System.Runtime.CompilerServices;
```

现在可以在MethodImpl 属性上使用MethodImplOptions.Synchronized 枚举成员,以确保这个方法一次仅由一个线程调用。新文件叫做 locking_pattern.cs。也可以使用锁定、监控器,或互斥方式来代替该枚举成员,但这些线程问题超出了本书的范围。

```
[MethodImpl(MethodImplOptions.Synchronized)]  
public static HeadTeacher getInstance()  
{  
    ...  
}
```

5.7.2 Abstract Factory 模式

Abstract Factory 模式与 Singleton 一样,可以在整个.NET Framework 类库中使用。Factory 这个名称对 Java 开发人员来说很熟悉,在 Java 中,Factory()方法用于整个 Java

类库，COM 开发人员也可以从用于创建 COM 类实例的 ClassFactory 对象中看到这个术语。

在使用 Abstract Factory 模式时，不需要直接通过构造函数来创建类，而可以展示 Factory，它会创建类。实际上不需要指定具体的类的系列，因为 Factory 知道应返回什么类，该类还可以扩展，返回一个默认类。只要需要扩展类的系列，就可以修改 Factory 类(它是抽象的)中的代码，而不需要直接修改 HospitalWorker 中的代码。

abstract factory 的最佳定义是 GoF 图书中给出的定义：

abstract factory 的作用是提供一个接口，以创建一系列相关对象，而不需要指定它们的具体类。

在本例中创建两种类型的医院工作者：医生和护士。护士和医生有一些共同的特点，例如，他们都具有行医资格或还是学生身份的。为了建立他们的模型，可以创建抽象的基类 Doctor 和 Nurse，基类包含底层类型 HospitalWorkers 的基本实现，本例中不需要考虑该实现，因为必须使用一个具体的类来创建医生或护士对象。但医生和护士的属性是相同的。首先创建一个抽象类 MedicalStaff，让两个抽象类 Doctor 和 Nurse 继承该类。

```
public abstract class MedicalStaff
{
    protected int salary;
    protected int grade;
    protected string hospital;

    public int Pay
    {
        get { return this.salary; }
        set { this.salary = value; }
    }

    public int Grade
    {
        get { return this.grade; }
        set { this.grade = value; }
    }

    public string Hospital
    {
        get { return this.hospital; }
        set { this.hospital = value; }
    }
}
```

```
public abstract class Doctor : MedicalStaff {}  
public abstract class Nurse : MedicalStaff {}
```

医生和护士还有两种不同的类型，即具有行医资格的或是学生，所以要派生新类，表示护士或医生的学生身份或有行医资格的身份。它们的实现并不重要，重要的是理解设计模式：

```
public class StudentNurse : Nurse {}  
public class QualifiedNurse : Nurse {}  
public class QualifiedDoctor : Doctor {}  
public class StudentDoctor : Doctor {}
```

继续这个设计模式，必须创建一个基类，以创建具有学生身份或有行医资格的护士或医生。基类叫做 HospitalWorker，它有两个方法，用于返回护士或医生，这些应在派生类中实现：

```
public abstract class HospitalWorkerFactory  
{  
    public abstract Nurse GetNurse() {}  
    public abstract Doctor GetDoctor() {}  
}
```

两个派生类是 StudentFactory 和 QualifiedFactory，它们根据所调用的方法，分别返回学生身份的医生或护士，以及有行医资格的护士或医生。

```
public class StudentFactory : HospitalWorkerFactory  
{  
    public Nurse GetNurse()  
    {  
        return new StudentNurse();  
    }  
  
    public Doctor GetDoctor()  
    {  
        return new StudentDoctor();  
    }  
}  
  
public class QualifiedFactory : HospitalWorkerFactory  
{
```

```
public Nurse GetNurse()
{
    return new QualifiedNurse();
}

public Doctor GetDoctor()
{
    return new QualifiedDoctor();
}
}
```

基类 `HospitalWorkerFactory` 是以模式命名的 `Abstract Factory` 类，因为它是一个接口。使用它的创建代码是一个 `Factory`，它利用具体的名称创建和返回了对象。类别的每种类型，无论是学生身份的还是有行医资格的医院工作者，都必须有派生于 `abstract factory` 的具体的实现。

在创建代码中，可以添加另一个类 `HospitalWorker`，把合适的 `factory` 作为一个方法参数，返回护士或医生。`HospitalWorkerFactory` 类是 `StudentFactory` 类型或是 `QualifiedFactory` 类型，因此可以确定正确的类，确定正确的选择。

```
public class HospitalWorker
{
    public static Nurse GetNurse(HospitalWorkerFactory hwf)
    {
        return hwf.GetNurse();
    }

    public static Doctor GetDoctor(HospitalWorkerFactory hwf)
    {
        return hwf.GetDoctor();
    }
}
```

有其他类型的 `Factory` 方法(抽象的 `factory` 方法只是其中的一个)，但它们都直接使用另一个类(`Factory` 类)来控制对象的创建。这里创建的所有类型都有相同的属性，但在这个例子中，可以为 `Doctor` 和 `Nurse` 抽象类提供完全不同的属性和方法。`.NET Framework` 提供了 `factory` 的许多例子，例如，`WebRequest` 类或 `SymmetricAlgorithm` 类，它们一般不是使用抽象的 `factory` 类参数，而是使用字符串参数，来表示类的字面量名称，让静态 `Create()` 方法返回适合的类实例。

5.7.3 懒惰的初始化

上面两种模式基于对象创建的抽象化，避免直接创建代码。在 Abstract Factory 模式中，创建所需的对象时，并不直接指定具体类的名称，但通过 factory 方法推迟对象的创建。而在 Singleton 模式中，使用一个类来管理对象的一个实例，该实例或者已经存在，或者需要使用一个静态方法创建。懒惰初始化可确保字段值仅在代码第一次访问时设置，之后就会返回同一个值。懒惰初始化是作为设计模式来进行的，但可以看做是一种优化技术，来避免资源的消耗。

利用上一节的 Doctor 和 Nurse，就可以返回一个当前尚未初始化的字段值。懒惰初始化模型仅在需要时填充字段值，因为实现代码很少使用它们。为了得到更好的性能，不要在构建时设置这个值，以防耗费过多的时间。懒惰初始化指定 get 属性访问器包含从资源中获取信息的所有必要代码，而且仅在填充底层字段值时调用。

在下面的 Lazy 类中，有两个字段 id 和 name。数字参数是一个整数值，在对象第一次创建和赋予 id 字段时，会把该数字参数传送给构造函数。在创建对象时，name 字段没有赋值，而在请求 Name 的 get 访问器从文件中填充 name 字段时，使用 id 值。这样，才能在使用时有效地进行初始化。

```
public class Lazy
{
    private int id;
    private string name;

    public Lazy(int number)
    {
        this.id = number;
    }

    public int ID
    {
        get { return id; }
    }

    public string Name
    {
        get
        {
            if(this.name==null)
```

```
        {  
            StreamReader sr;  
            try  
            {  
                sr = new  
                    StreamReader("C:\\name"+id.ToString("0000")+".txt");  
                name = sr.ReadToEnd();  
            }  
            finally  
            {  
                if(sr!=null)  
                    sr.Close();  
            }  
            return this.name;  
        }  
    }  
}
```

5.7.4 Copy-On-Write

Copy-on-write 是一般设计模式，如代理模式，的一个应用。这与对象构建的讨论有关，因为它直接与通过优化(其方式与懒惰初始化相同)来构建对象有关。本章的前面讨论过，深拷贝要比浅拷贝的代价更高，所以从效率的角度来看，应使用浅拷贝。这里的问题是，浅拷贝依赖对象的共享引用，只要对象不需要某些不同的字段值，使用浅拷贝就比较好，实际上，在应用程序中，它们总是必须在某个阶段使用。

Copy-on-write 解决了如下问题：确定何时写入对象值，如果是，它就获取引用对象的副本，进行相应的修改。这样，它不像深拷贝那样昂贵，因为第二个对象引用只需要在整个应用程序中应用读取操作，而不是一个几乎相同的副本。像引用计数、与 copy-on-write 相关的同步等问题，会使其实现更强壮。

Copy-on-write 可以内置于类中，在一些.NET 对象中继承。例如，在下面的代码中，第二个变量 world 是第一个变量 hello 的引用拷贝。world 变量应修改底层的字符串，这样在引用和写入屏幕时，这两个引用应生成表达式 Hello World!

但是，在这种情况下，当字符串使用 copy-on-write 技术时，并不是按值创建字符串的副本。因为重载的运算符会检查字符串上的引用数。如果该引用数大于 1，对象值已修改，就创建一个新副本，并把已修改的新副本的引用存储在第二个变量中。

```
string hello = "Hello";
```

```
string world = hello;  
world += " World!";  
Console.WriteLine("{0}\\{1}", hello, world);
```

这个优化结果可以使用运算符重载的类似技术在对象上应用。在上面的字符串例子中，对字符串的每个引用都可以调用隐式的转换运算符，返回一个对当前字符串的引用，给引用数递增 1。在使用 += 运算符时，这将检查引用数是否大于 1，如果是，就按值复制字符串，返回一个追加了附加字符串参数的数字字符串，并把引用数设置为 1。copy-on-write 模式的使用有许多争论，因为它需要同步，进行引用计数和其他检查也会耗费比原来的按值拷贝更多的资源。

5.8 小结

本章主要讨论了创建和删除对象所需要了解的许多技术和原则，包括托管堆和垃圾收集器的知识，以及链接和重载构造函数所必须的语义。我们还介绍了初始化和创建的不同形式，包括使用对象克隆、并行化和运算符转换。本章最后介绍了创建对象的设计模式，说明了使用它如何修改对象创建过程，使之更高效和抽象，优化对象的创建，提高其性能。

第6章 事件和委托

基于事件的编程是 .NET Framework 的基石，它是包括 Windows Forms 和 ASP.NET 页面在内的所有 .NET 用户界面代码的基础。在应用程序中的任何位置都可以使用事件，解除一个对象与依赖它的其他对象之间的耦合。在一个对象中展示和激活事件，能够使别的对象来响应在编写类代码时没有考虑到的动作。因此，在任何一种 C# 类型的接口中，事件都是非常有用的组件。

.NET Framework 中的事件机制使用了委托(或回调)。第 1 章在讨论 .NET Framework 的通用类型系统时，简要介绍了委托。委托封装了函数或子例程的签名。在 C/C++ 中，它类似于函数指针。函数指针就是内存地址，编译器对它指向方法的签名不了解。因此，函数指针不是类型安全的(如果传送错误的类型，编译器不会察觉)，不能作为托管应用程序的一部分。委托定义为代表在应用程序中需要“回调”的方法签名。然后，创建委托的实例，并将其绑定到指定的方法上。这样，就可以利用委托调用前面所选择的方法了。因为委托只能指向有特定签名的方法，所以是类型安全的。

本章将讨论委托机制的一些细节问题。学习如何把委托编译为 MSIL 代码，帮助理解委托内部的工作原理。然后，讨论一些高级的委托问题，例如多播委托和异步委托。

在学习了委托的使用方法之后，将讨论 .NET Framework 的事件模型。大多数程序员都熟悉某种事件模型，但是 .NET Framework 中的事件与 Visual Basic 或 Java 中的事件的工作原理大相径庭。我们将讲述事件模型如何使用委托，以及如何在用户界面代码中使用事件。同时，还将学习如何将事件作为一种常用的方法来解除事件源对象(生成事件的对象)与事件接收器对象(接收事件通知的对象)之间的耦合。

6.1 委托

要理解事件在 C# 中的工作方式，首先要了解委托的使用方式。事件模型利用委托来标识应用程序中的事件处理程序方法。除了在事件机制中的作用之外，委托本身也有非常重要的作用。

委托类似于应用程序中指向对象中方法的指针。如前所述，C 和 C++ 中使用的函数指针不是类型安全的，因为编译器不知道所调用的函数签名。所以，如果程序员出了错，就可能把错误类型的参数传送给函数。Java 的设计人员在语言规范中禁止使用函数指针，从而解决了这一问题。

对于一般用回调函数解决的编程问题(把指针传送给函数),Java 程序员求助于定义一个专用的回调接口。调用者定义一个可以调用的类型(接口或类)。在编写必须调用的代码时,必须实现这个接口,方法名还必须与调用者定义的相同。与使用委托一样,这个技术允许编写一个调用对象,而不必知道最终要调用什么对象。但是,所调用的对象必须包含相当多的代码来实现接口,还必须遵循调用对象的设计原则。委托允许开发人员拥有与不安全的函数指针相同的灵活性,但具有类型安全性。

.NET Framework 的设计人员在克服函数指针的问题时,把它们封装在一个特殊类型的类中,每个类对应一个函数签名。这些类称为委托类或委托。定义委托类,可以指定要通过委托调用的方法签名。接着创建一个委托对象,并将其绑定到其签名同委托签名匹配的方法上。

在下面的几种情况中,可以使用委托:

- 将某一对象的方法注册为另一个对象的回调方法

对该对象而言,发生了一些很重要的事情时,就可以调用回调方法。调用委托的对象并不需要了解自己所引用的对象或该方法。委托中包含了哪个对象调用哪个方法时的全部信息。因此,即使将来定义了带有新方法名称的新对象的类型,调用委托的对象仍然能够继续正常工作。

- 在算法中选择使用具有相同签名的一系列方法中的一个方法。

例如,第1章学习了如何定义一个代表例如 `Math.Sin`、`Math.Cos` 和 `Math.Tan` 等数学函数的委托。

- 把方法的选择与调用分离开来。

可以创建委托来表示需要调用的方法,但是,只有在准备就绪后,才能执行该方法。例如,在一个实时应用程序中,创建一个表示每隔 10 秒调用一次的方法的委托。然后,创建一个独立的线程,并利用委托每隔 10 秒调用所需的方法。

6.1.1 创建和使用简单委托

在应用程序中使用委托,需要按照以下 3 步进行:

- 声明委托类型
- 创建委托对象,并将该对象绑定到指定的方法上
- 使用委托对象来调用方法

下面的例子声明了一个委托,利用该委托在 Windows 窗体中绘制不同类型的图形。这是一个完全可见的例子,在此没有列出所有的窗体代码。本例的源代码在下载代码文件夹 `DelegatesEvents\UsingDelegates.cs` 中。

窗体的外观如图 6-1 所示。

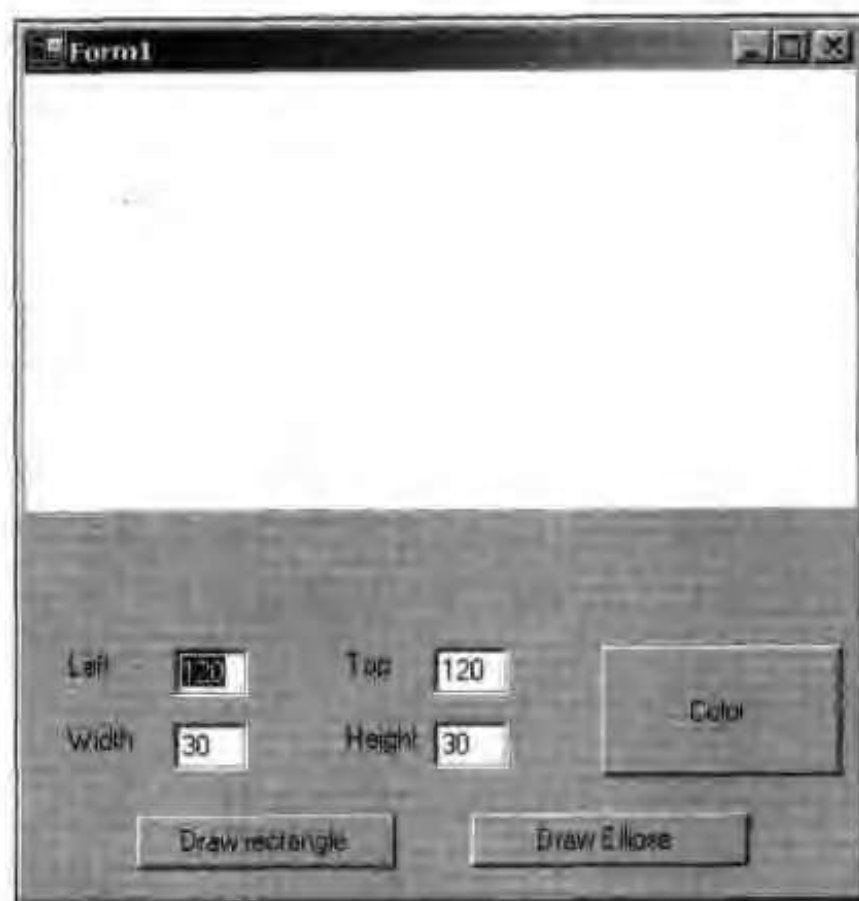


图 6-1

当用户单击 Draw rectangle 按钮时，应用程序将根据指定的位置、大小和颜色显示矩形。与此类似，当用户单击 Draw ellipse 按钮时，应用程序则根据指定的特征显示椭圆。

为了实现上述功能，可以编写两个完全独立的按钮单击事件处理程序方法，分别处理 Draw rectangle 和 Draw Ellipse 单击事件。但是，由于每个方法都需要执行下面的任务，将产生大量重复代码：

- 确保填写了所有的文本框，指定形状的位置和尺寸
- 确保图形在屏幕上的绘图区显示
- 利用指定的颜色创建画笔
- 绘制相应的形状

为了避免编写重复代码，可以利用委托来代表所需的绘图操作。该委托或者指向 FillRectangle()方法，或者指向 FillEllipse()方法，这两个方法均在 .NET Framework 类库的 Graphics 类中定义。然后，编写一个通用的方法，该方法把该委托作为参数输入，利用委托调用相应的绘图操作。

下面是应用程序中的部分代码：

```
// omitted the delegate declaration here
```

```
public class UsingDelegates : System.Windows.Forms.Form
{
    //Declare a Rectangle field, to indicate available drawing area
    private Rectangle mRect;
```

```
#region Windows Form Designer generated code
// omitted for brevity. This region contains some event
// related code as well. We will look at that later.
#endregion

//When the form receives a paint event,
//color the available drawing area white
private void UsingDelegates_Paint(object sender,
    System.Windows.Forms.PaintEventArgs e)
{
    mRect = new Rectangle(0, 0, Width, Height / 2);
    e.Graphics.FillRectangle(new SolidBrush(Color.White), mRect);
}

private void btnColor_Click(object sender, System.EventArgs e)
{
    ColorDialog dlgColor = new ColorDialog();
    dlgColor.ShowDialog(this);
    btnColor.BackColor = dlgColor.Color;
}

// Delegate-related code (see later)...
}
```

在上面这段代码中需要注意以下几点内容：

- 声明一个名为 `mRect` 的 `private` 字段，表示窗体中的绘图区。
- 当窗体绘制自身时，初始化 `mRect` 字段，并将矩形区域绘制为白色。
- 程序为窗体中的 `Color` 按钮提供了单击事件处理程序方法。该方法显示一个标准的颜色对话框，这样用户可以选择下一个形状的颜色。为了表明用户选择的颜色，利用所选的颜色重新绘制 `Color` 按钮。注意，这不是指定处理 `btnColor` 中的 `Click` 事件的方法名。方法和事件的联系在省略了的 `Forms Designer` 代码中。本章的第二部分将详细讨论这些代码。

现在，继续看一下委托专用的代码。在类定义的前面，加上一个委托定义，如下所示：

```
//Define a Delegate type, to indicate signature of
//method to call
delegate void DrawShape(Brush aBrush, Rectangle aRect);
```

委托的定义与类的定义类似，它们都指定了一个新类型。它不是 `UsingDelegates` 类的一部分，但构成了一个独立的类。该委托代表了返回空值的方法，该方法带两个参数：

- 一个 `Brush` 对象，用来指定形状的颜色
- 一个 `Rectangle` 对象，用来指定形状的位置和尺寸

这个委托对应于 `Graphics` 类中 `FillRectangle()` 和 `FillEllipse()` 方法的签名。下一步是创建该类型的实例，把它绑定到特定对象的特定方法上。下面的例子说明了如何创建一个 `DrawShape` 实例，再把它绑定到 `Graphics` 对象的 `FillRectangle()` 方法上：

```
//Handle the Click event for the btnDrawRect button
private void btnDrawRect_Click(object sender, System.EventArgs e)
{
    // Create a Graphics object (we need its FillRectangle method)
    Graphics aGraphics = CreateGraphics();

    // Declare a DrawShape variable
    DrawShape DrawRectangleMethod;

    // Create a delegate object, and bind to the FillRectangle method
    DrawRectangleMethod = new DrawShape(aGraphics.FillRectangle);

    // Call MyDrawShape, and pass the delegate as a parameter
    MyDrawShape(DrawRectangleMethod);
}
```

在上面的 `btnRectangle_Click()` 方法中需要注意以下内容：

- 由于绘图方法(例如 `FillRectangle()` 和 `FillEllipse()`)定义为 `Graphics` 类的实例方法，因此需要创建 `Graphics` 对象。这样，需要一个 `Graphics` 类的实例来调用上面的方法。`CreateGraphics()` 是从 `Control` 类中继承的方法(窗体是一个子类)。
- 声明一个 `DrawShape` 类型的局部变量。这样强化了委托是一种数据类型的概念。本章的后面将讲述 `DrawShape` 数据类型的 MSIL 代码。
- 创建一个 `DrawShape` 类型的实例，并将其绑定到 `Graphics` 对象的 `FillRectangle()` 方法上。注意传送方法名时没有加圆括号。这个语法只能用于委托的构造函数。

注意：

也可以将委托绑定到静态方法上。为了实现委托与静态方法的绑定，而不是与实例方法的绑定，需要在方法前指定类名(例如：`Math.Sin`)，而不是在方法前指定对象名(例如：`aGraphics.FillRectangle`)。

- 给另一个方法 `MyDrawShape()` 传送 `DrawShape` 实例。`MyDrawShape()` 使用委托调用指定的绘图操作(`aGraphics.FillRectangle`)

现在, 为处理 `btnDrawEll` 创建一个类似的方法。通常, 委托实例在创建后立即传送给方法。在下面的代码中使用委托会产生混淆。`btnDrawEll_Click()` 代码的功能与 `btnDrawRect_Click()` 的功能类似, 读者可以自己比较一下。

```
// Handle the Click event for the btnDrawEll button
private void btnDrawEll_Click(object sender, System.EventArgs e)
{
    // Use a shorter but more cryptic syntax.
    MyDrawShape(new DrawShape(CreateGraphics().FillEllipse));
}
```

最后一个任务是在 `MyDrawShape()` 方法中使用委托, 来调用需要的绘图操作。下面是 `MyDrawShape()` 方法的代码:

```
// MyDrawShape uses a delegate to indicate which method to call
private void MyDrawShape(DrawShape theDelegate )
{
    // Are any text fields blank?
    if (txtLeft.Text.Length == 0 || txtTop.Text.Length == 0 ||
        txtWidth.Text.Length == 0 || txtHeight.Text.Length == 0)
    {
        MessageBox.Show("Please fill in all text boxes", "Error",
            MessageBoxButtons.OK,
            MessageBoxIcon.Error);

        return;
    }

    // Get the coordinate values entered in the text fields
    Rectangle aRect = new Rectangle(Int32.Parse(txtLeft.Text),
                                    Int32.Parse(txtTop.Text),
                                    Int32.Parse(txtWidth.Text),
                                    Int32.Parse(txtHeight.Text));

    // Make sure the coordinates are in range
    if (mRect.Contains(aRect))
    {
        // Get the color of the btnColor button
        Brush aBrush = new SolidBrush(btnColor.BackColor);
        // Call the delegate, to draw the specified shape
    }
}
```

```
        theDelegate(aBrush, aRect);
    }
    else
    {
        // Display error message, and return immediately
        MessageBox.Show("Coordinates are outside drawing area", "Error",
            MessageBoxButtons.OK, MessageBoxIcon.Error);
    }
}
```

在 `MyDrawShape()` 方法中需要注意以下内容：

- `MyDrawShape()` 使用 `DrawShape` 类型的委托。委托对象指定了所需的绘图操作，还指定了绘图操作中要使用的 `Graphics` 对象。有一点必须清楚：`MyDrawShape()` 并不了解在委托中指定的方法。因此，将来更便于扩展方法，例如引进新的绘图操作。
- `MyDrawShape()` 执行大量的管理任务，例如获得用户输入的尺寸、位置和颜色信息。
- `MyDrawShape()` 利用委托调用指定的绘图方法。委托的调用方式与方法一样。传送参数时，必须使用委托定义中指定的类型。例如：`DrawShape` 带一个 `Brush` 参数和一个 `Rectangle` 参数。如果目标方法有带不同参数的重载方法，就不能使用这个委托调用重载方法。

要编译应用程序，可以使用命令行编译器或图形化的 IDE，例如 Visual Studio .NET。要运行应用程序，可以输入下面的命令：

```
C:\Class Design\Ch 06> UsingDelegates.exe
```

此时，运行应用程序，然后，根据用户选择的颜色创建矩形和椭圆形。

应用程序的结果很有意思，而了解应用程序如何编译成 MSIL 代码更为有用。关闭应用程序，在命令提示中输入下面的命令(确保 ILDASM 在系统路径下)：

```
C:\Class Design\Ch 06> ildasm UsingDelegates.exe
```

MSIL Disassembler 窗口中显示了下面的信息(在图 6-2 中，展开了我们感兴趣的节点，从而显示了每个类型的详细信息)。

正如在 Windows 应用程序中所期望的那样，这里有许多信息。其中最重要的部分是 `DrawShape` 的 MSIL 代码。注意下面与 `DrawShape` 相关的要点：



图 6-2

- DrawShape 是一个类。只要在应用程序中定义了一个新的委托类型，编译器都将生成一个类，来封装委托的签名信息。
- MSIL 语法 `extends[mscorlib]System.MulticastDelegate`，表示生成的 DrawShape 类是从一个标准 .NET Framework 类 MulticastDelegate 继承而来的。本章的后面将解释多播委托的含义。MulticastDelegate 在逻辑上位于 System 命名空间中，在物理上则位于 mscorlib 程序集中。本书将在第 7 章中详细讨论继承，第 8 章学习命名空间和程序集。
- 除了从父类 MulticastDelegate 继承而来的方法外，DrawShape 还定义了几个方法。首先是 BeginInvoke() 和 EndInvoke() 方法，它们允许在需要时通过委托异步调用方法。换句话说，可以在后台线程中利用委托调用方法；应用程序的主线程能够连续执行，而无需等待方法完成。本章后面将讲述异步委托的示例。
- DrawShape 还有一个 Invoke() 方法，它允许通过委托同步调用方法。编译器根据定义 DrawShape 类型时指定的签名生成 Invoke() 方法。在调用委托时，编译器会生成在委托实例上调用 Invoke() 方法的 IL 代码。如果试图在自己的代码上调用 Invoke() 方法，C# 编译器就会生成一个错误 “Invoke() cannot be called directly on a delegate”。在其他的许多 .NET 语言中，程序员必须自己调用 Invoke() 方法。

6.1.2 创建和使用多播委托

.NET Framework 支持两种类型的委托：

- 单播委托：允许在一个对象上调用一个方法。上一节介绍了这种委托的工作

方式。

- 多播委托：允许在不同的对象上调用一系列方法。多播委托维护一个调用列表，以牢记在哪个对象上调用哪个方法。调用多播委托时，委托将按顺序调用指定对象的指定方法。

注意：

实际上，所有的委托都是多播的。单播委托是调用列表上只有一个方法的委托。但单播委托(通过构造函数)和多播委托(组合已有的委托)的创建大不相同，所以要把它当成不同的实体。

如果需要对一个对象的集合执行相同的操作，或者需要对同一个对象执行一系列操作，或者是上面这两种情况的组合时，就可以使用多播委托。利用多播委托可以隐式地将需要执行的所有方法和执行这些方法的所有对象组成一个集合。把同一委托类型的实例合并起来，就创建了多播委托。我们可以通过合并相同委托类型的实例创建多播委托。

创建和使用多播委托的步骤是：

(1) 定义一个委托类型，表示需要通过委托调用的方法的签名。多播委托只能用于执行签名相同的方法，这些方法的签名必须和委托指定的强数据类型一致。

(2) 编写具有相同签名的方法作为委托。

(3) 创建委托对象，并将它绑定到需要通过委托调用的第一个方法上。

(4) 创建另一个委托对象，将其绑定到下一个需要调用的方法上。

(5) 调用 `System.Delegate` 类的 `Combine()` 方法，将上面创建的两个委托合并成一个集成的多播委托。方法 `Combine()` 返回一个新的委托，该委托的调用列表中包含了所有的委托。

(6) 重复执行上面两个步骤，根据需要创建多个委托，并将它们合并成一个集合的多播委托。

(7) 如果需要从多播委托中删除委托，可以调用 `System.Delegate` 类中定义的 `Remove()` 方法。`Remove()` 方法返回一个新的委托，其调用列表中不包含已经删除的委托。如果调用列表中只包含刚刚删除的委托，那么 `Remove()` 方法返回 `null`。

(8) 当调用多播委托指定的方法时，只需像前面那样调用委托。即可按调用列表中的顺序调用方法，只返回调用列表中最后一个方法的结果。

除了 `System.Delegate` 类上的静态方法 `Combine()` 和 `Remove()` 之外，C# 还提供了另一个语法来添加或删除多播委托中的委托：使用加号和减号运算符(+和-)。可以把两个委托加在一起，创建一个新的多播委托。下面的两行代码是等价的：

```
// Combining DelegateA and DelegateB to CombinedDelegate
System.Delegate CombinedDelegate =
```

```
System.Delegate.Combine(DelegateA, DelegateB);
```

```
// Idem
```

```
System.Delegate CombinedDelegate = DelegateA + DelegateB;
```

与往常一样，这些运算符可以与赋值运算符组合为+= 和 -=。这是在 C#中合并委托最常见的方法。运算符的语法看起来常常比使用 Combine() 和 Remove()方法更简洁。C#编译器把加号运算符转换回静态方法的调用(这不是在 Delegate 类中重载的运算符)。

为了说明这些概念，演示多播委托的 C#语法，下面这个完整的例子将使用多播委托，在 Windows Forms 应用程序中绘制任意数量的子窗体。该应用程序包含一个主窗体，允许用户创建新的子窗体，随时改变这些子窗体的颜色。主窗体使用多播委托在每一个子窗体中调用 Repaint()方法。

本例的源代码保存在下载文件夹 DelegatesEvents\MulticastDelegates 中。在学习源代码之前，首先看看应用程序的执行情况。应用程序中的主窗体如图 6-3 所示。



图 6-3

用户在对话框中输入屏幕坐标，然后单击 Add Window，来创建子窗体。在应用程序的这个阶段，创建了新的委托，并将它绑定到新建子窗体的 Repaint()方法上。将这个委托加入到主窗体的多播委托中，跟踪所有打开的子窗体。

主窗体在其状态栏中显示一个消息，表示多播委托的调用列表中的项数。这说明当前打开的子窗体的个数。

如果多次单击 Add windows 按钮，屏幕上将显示多个子窗体。多播委托将跟踪这些打开的子窗体。更准确地说，在多播委托的调用列表中，保存着每个子窗体的 Repaint()方法，如图 6-4 所示。



图 6-4

当用户单击 Change Color 选择新颜色时, 就调用多播委托重新绘制所有的子窗体。换句话说讲, 就是调用多播委托, 从而调用每一个子窗体的 Repaint() 方法。每个窗体按照指定的颜色重新绘制, 如图 6-5 所示。

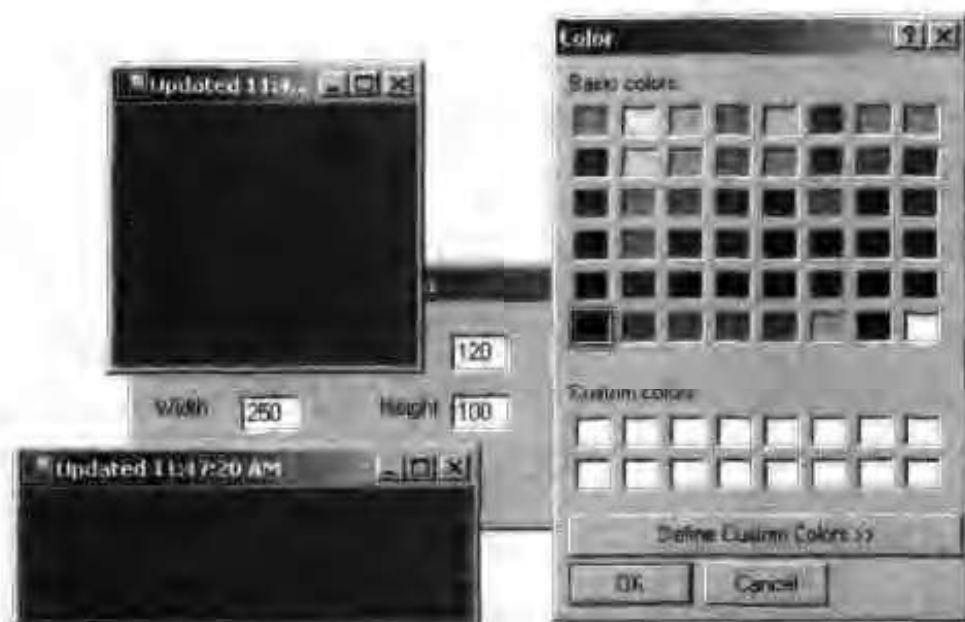


图 6-5

这里还需要考虑另外一个问题。如果用户关闭一个子窗体, 则需要从主窗体的多播委托中删除相应的委托。为此, 主窗体中包含一个 ChildFormClosing() 方法, 在子窗体关闭之前调用该方法。在 ChildFormClosing() 方法中, 从多播委托中删除了该子窗体的委托, 因为以后不需要重新绘制该子窗体。

- 当委托所指向的对象不再存在时, 最好从调用列表中删除该委托。也就是说, 对象已超出其作用域, 不久会被垃圾回收。但委托不仅指向对象上的方法, 还保存该对象本身的引用。因此, 只要委托仍在调用列表中, 对象就应保留在内存中。

理解了应用程序的工作原理后, 下面看看其代码。子窗体的源代码比较简单, 因为这个类中没有与委托相关的代码, 子窗体也不了解应用程序中的其他子窗体。ChildForm 类的源代码在 ChildForm.cs 中提供:

```
public class ChildForm : System.Windows.Forms.Form
{
    private System.ComponentModel.Container components = null;
    // omitted the constructor

    #region Windows Form Designer generated code
    // omitted for brevity
    #endregion

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

```

        // Display the current time in the window's title bar
        this.Text = "Created " + DateTime.Now.ToLongTimeString();
    }

    // This method will be called via multicast delegate in main form
    public string Repaint(Color theColor)
    {
        // Set the color for this form, and update the caption bar
        this.BackColor = theColor;
        this.Text = "Updated " + DateTime.Now.ToLongTimeString();
        return this.Text;
    }

    // Handle the Cancel event for this form
    private void ChildForm_Closing(object sender,
        System.ComponentModel.CancelEventArgs e)
    {
        // Tell the main form we are closing, so the main form can
        // remove us from its multicast delegate
        Mainform MyOwner = (Mainform)this.Owner;
        MyOwner.ChildFormClosing(this);
    }
}

```

对 ChildForm 类需要注意以下几点：

- ChildForm_Load()方法在标题栏中显示窗体的创建时间。
- Repaint()方法有一个参数 Color，它告诉窗体采用什么颜色重新绘制自身。当用户选择新颜色后，主窗体的多播委托就调用该方法。
- ChildForm_Cancel()方法通知主窗体，子窗体即将关闭。使用 Owner 属性得到对主窗体的引用，然后调用其 ChildFormClosing()方法。

下面是应用程序中主窗体的代码。主窗体包含了所有与委托相关的程序。首先，定义委托类型，并声明一个引用多播委托对象的字段：

```

// define the Delegate type (and it's signature)
delegate string ChangeColorDelegate(Color aColor);

public class Mainform : System.Windows.Forms.Form
{

```

```

        // Declare a ChangeColorDelegate field,
        // to refer to the multicast delegate
        private ChangeColorDelegate mAllRepaintMethods;
    }

```

在上面定义的 MainForm 类中需要注意以下内容：

- **ChangeColorDelegate** 定义了需要通过该委托类型调用的方法的签名。委托的签名和 ChildForm 类的 Repaint() 方法的签名相匹配。

在 C# 中，定义单播委托类型和多播委托类型的方式是没有区别的。事实上，C# 中的每一个委托类型都是一个隐式的多播委托。至于是使用委托代表一个还是多个方法由用户自己决定。

- **mAllRepaintMethods** 是一个指向多播委托的字段。换句话说，mAllRepaintMethods 将引用 ChangeColorDelegate 实例。最初时，由于没有创建子窗体，mAllRepaintMethods 是一个空引用。当创建了第一个子窗体后，将创建一个新的 ChangeColorDelegate 实例，并把它赋予 mAllRepaintMethods。

现在来看如何创建新的子窗体，并将它的 Repaint() 方法加入多播委托：

```

        // Handle the click event on the btnAddWindow button
        // the registration of this method for the Click event is done
        // elsewhere

        private void btnAddWindow_Click(object sender, System.EventArgs e)
        {
            // Are any text fields blank?
            if (txtLeft.Text.Length == 0 || txtTop.Text.Length == 0 ||
                txtWidth.Text.Length == 0 || txtHeight.Text.Length == 0)
            {
                MessageBox.Show("Please fill in all text boxes",
                                "Error",
                                MessageBoxButtons.OK,
                                MessageBoxIcon.Error);
                return;
            }

            ChildForm aChildForm = new ChildForm();
            aChildForm.Owner = this;
            aChildForm.DesktopBounds = new Rectangle(
                Int32.Parse(txtLeft.Text),
                Int32.Parse(txtTop.Text),

```

```
        Int32.Parse(txtWidth.Text),
        Int32.Parse(txtHeight.Text));
    aChildForm.Show();

    // Create a new delegate for the child form's Repaint method
    ChangeColorDelegate newDelegate = new ChangeColorDelegate
        (aChildForm.Repaint);

    // Combine new delegate into the multicast delegate
    mAllRepaintMethods = (ChangeColorDelegate)System.Delegate.Combine(
        mAllRepaintMethods, newDelegate);

    // Use multicast delegate to count the child forms
    sbStatus.Text = "Created child form " +
        mAllRepaintMethods.GetInvocationList().Length + ".";
}
```

在上面的 `btnAddWindow_Click()` 方法中需要注意下面的内容：

- 首先验证用户是否在所有的文本字段中都输入了值。如果是，就根据指定的位置和尺寸创建新的子窗体，并在屏幕中显示该窗体。在此，指定子窗体的 `Owner` 为主窗体。
- 创建一个新的 `ChangeColorDelegate` 实例，并将其绑定到新子窗体的 `Repaint()` 方法上。
- 使用 `System.Delegate` 上的静态方法 `Combine()`，把 `newDelegate` 实例添加到多播委托的调用列表 `mAllRepaintMethods` 上。最初当 `mAllRepaintMethods` 还没有初始化时，`Combine()` 方法返回传送过来的 `newDelegate` 实例。
- 委托有一个 `GetInvocationList()` 方法，它返回一个委托对象的数组，表示多播委托中的所有项。可以使用属性 `Length` 确定这个数组的长度。在本例中，这个属性表示当前已打开的子窗体的个数。

下面的代码把新委托和已有的委托合并到一个新的多播委托中：

```
mAllRepaintMethods += newDelegate;
```

在生成的 MSIL 代码中，这看起来是相同的。一旦用户创建了子窗体，单击 `Change Color`，可以改变这些子窗体的颜色。该功能通过下面的代码实现：

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

```
if(mAllRepaintMethods == null)
{
    MessageBox.Show("There are no child forms to change.",
        "Error changing color",
        MessageBoxButtons.OK,
        MessageBoxIcon.Error);
}
else
{
    // Ask user to choose a color
    ColorDialog dlgColor = new ColorDialog();
    dlgColor.ShowDialog();

    // Invoke multicast delegate, to repaint all the child forms
    mAllRepaintMethods (dlgColor.Color );

    // Use multicast delegate to count the child forms
    sbStatus.Text = "Updated " +
        mAllRepaintMethods.GetInvocationList().Length +
        " child form(s).";
}
}
```

在上面所示的 `btnColors_Click` 方法中，需要注意下面的内容：

- 如果 `mAllRepaintMethods` 为 `null`，就表示没有打开的子窗体。因此，显示一个错误消息，并立刻返回。利用多播委托来跟踪是否存在子窗体非常简单。
- 如果 `mAllRepaintMethods` 不为 `null`，就提示用户为已有的窗体选择新的颜色。然后，调用多播委托，从而调用所有子窗体的 `Repaint()` 方法。利用多播委托来遍历子窗体，并重新绘制它们十分方便。
- 在主窗体的状态栏中显示一个消息，说明重新绘制了多少个子窗体。同前面一样，调用多播委托的 `GetInvocationList` 方法就可以得到该信息。

最后，看看 `MainForm` 类中的 `ChildFormClosing()` 方法。子窗体在关闭前调用该方法，让主窗体跟踪其子窗体。`ChildFormClosing()` 方法的代码如下：

```
public void ChildFormClosing(ChildForm aChildForm )
{
    // Create a dummy delegate for the ChildForm that is closing
    ChangeColorDelegate unneededDelegate =
        new ChangeColorDelegate ( aChildForm.Repaint);
```

```
// Remove the delegate from the multicast delegate
mAllRepaintMethods = (ChangeColorDelegate)System.Delegate.Remove(
    mAllRepaintMethods, unneededDelegate);

// If multicast delegate is Nothing, there are no child forms left
if (mAllRepaintMethods == null)
{
    sbStatus.Text = "Final child form has been closed.";
}
else
{
    // Use multicast delegate to count the child forms
    sbStatus.Text = "Child form closed, " +
        mAllRepaintMethods.GetInvocationList().Length +
        " form(s) remaining.";
}
}
```

在上面所示的 `ChildFormClosing()` 方法中，需要注意下面的内容：

- `ChildFormClosing()` 方法接收一个 `ChildForm` 参数，来表示即将关闭的子窗体。
- 创建新的委托，并将它绑定到即将关闭的子窗体的 `Repaint()` 方法上。然后，调用 `System.Delegate` 类的 `Remove()` 方法，从多播委托中删除该委托。`Remove()` 方法返回两个可能的值中的一个：
 - 如果从多播委托中删除了最后一个委托，多播委托将不再存在。此时，`Remove()` 方法返回 `null`。
 - 相反，`Remove()` 方法返回一个不包含刚刚删除的委托的多播委托。
- 注意委托显示值的语法。即使创建了指向方法的新委托，仍可以使用它来删除指向同一方法的另一个委托。这两个委托是等价的。如果列表中的多个委托指向同一个对象的同一个方法，则只删除第一个委托。
- 在主窗体的状态栏中显示相关消息，指示剩余的子窗体的个数。

编写下面的代码，从调用列表中删除一个委托，这行代码要短得多：

```
// Remove the delegate from the multicast delegate
mAllRepaintMethods -= unneededDelegate;
```

示例应用程序的源代码已经全部结束。要运行该应用程序，输入下面的命令：

```
C:\Class Design\Ch 06> MainForm.exe
```


此时, 应用程序的外观如图 6-6 所示。通过该窗体能够打开子窗体, 并设置子窗体的颜色。



图 6-6

当发生重要情况时, 利用多播委托将便于维护需要一起通知的对象的集合。当调用多播委托时, 将隐式地遍历其调用列表中的所有项, 从而对每一个指定对象调用指定的方法。

6.1.3 创建和使用异步委托

前面的例子都是通过直接调用委托来同步调用方法。在同步调用方法时, 必须在调用的方法返回后, 才能继续执行程序。

还可以使用委托异步调用方法。这样, 就可以在后台线程中调用方法, 同时, 主线程可以继续工作。当有大量任务需要执行时, 使用异步委托是很有用的。

为了支持通过委托异步调用方法, C#编译器在委托类中生成了下面两个方法(以及支持同步调用方法的 `Invoke()` 方法):

- **BeginInvoke()**

调用 `BeginInvoke()` 方法在后台线程中异步调用所需的方法。

- **EndInvoke()**

当指定的方法执行结束后, 可以调用 `EndInvoke()` 得到方法的返回值。该方法还可以访问方法的任何输出参数。例如: 如果方法的参数是 `ByRef`, 在方法运行过程中修改了这些参数, 就可以使用 `EndInvoke()` 获得修改后的值。如果方法还没有执行完, 应用程序就会等待(暂停), 直到返回值可用为止, 所以调用 `EndInvoke()` 应小心。

根据需要, 使用异步委托有几种不同的方法。下面是几种可能发生的情况, 按照问题的复杂性排列:

- 调用不返回结果的方法。
- 调用返回结果的方法, 并在有限的时间内等待返回结果。
- 调用返回结果的方法, 并当生成返回结果时, 定义一个回调方法接收返回结果。

为了对这些可能的情况有一定的了解, 下面看一个示例应用程序, 该程序使用这 3 种可能性, 显示每种情况的发生时间。这个例子的源代码在下载文件夹 `DelegatesEvents\AsyncDelegates` 中。在介绍代码之前, 先看看应用程序的功能。该应用

程序只有一个窗体。在编译应用程序并运行它时,结果如图 6-7 所示。



图 6-7

单击其中任何一个按钮,应用程序都开始从 Internet 上下载 3 个 XML 文件(测试这个例子需要一个能工作的网络连接)。它们包含 Slashdot 网站上 3 个类目中的主要文章。当文件完全下载,并由 `System.Xml.XmlDocument` 类分析后,就显示自单击按钮之后的时间(单位是毫秒)。在下载了 3 个文件后,屏幕应如图 6-8 所示。

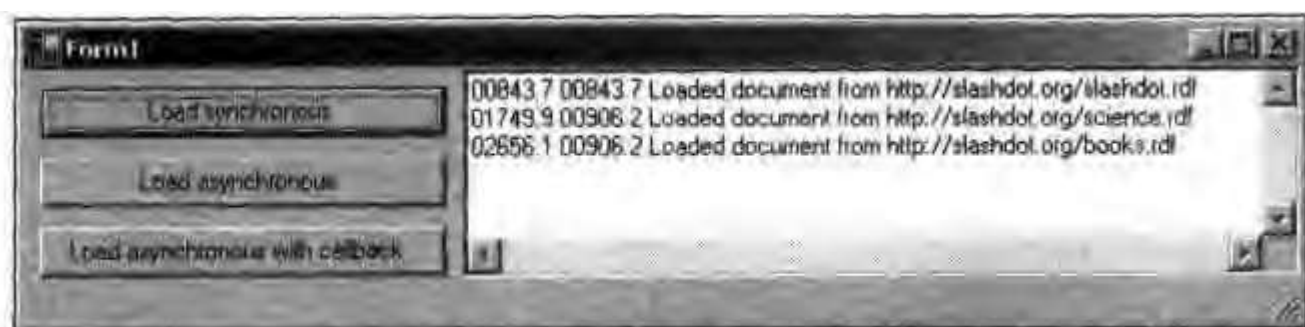


图 6-8

每次下载都需要大约 900 毫秒(在读者自己的环境中,该时间可能或短或长,这取决于读者的连接)。因为文件的大小大致相等,且从同一个服务器上访问,所以上述正是我们想要的结果。

首先看看同步代码。在用户单击 Load Synchronous 按钮时调用这些代码:

```
public class MainForm : System.Windows.Forms.Form
{
    private ArrayList mUrlList;

    private void MainForm_Load(object sender, System.EventArgs e)
    {
        // Initialising a list of three Urls containing valid Xml content
        mUrlList = new ArrayList();
        mUrlList.Add("http://slashdot.org/slashdot.rdf");
        mUrlList.Add("http://slashdot.org/science.rdf");
        mUrlList.Add("http:// slashdot.org/books.rdf");
    }

    private void btnSync_Click(object sender, System.EventArgs e)
```



```
{
    // Clear the output
    ClearLog();
    // Initialise an array of XmlDocument objects
    XmlDocument[] Documents = new XmlDocument[mUrlList.Count];
    for (int i = 0; i < mUrlList.Count; i++)
    {
        Documents[i] = new XmlDocument();
        // Both the loading of the content over the network
        // and the parsing of this content happens in the next line.
        Documents[i].Load((string)mUrlList[i]);
        AppendLog("Loaded document from " + mUrlList[i]);
    }
}
}
```

注意以下几点：

- Mainform_Load()方法初始化要下载的 URL 的 ArrayList, 这些 URL 是硬编码的。
- 在 btnSync_Click()方法中, 使用了 ClearLog()和 AppendLog()方法。这些方法的代码没有列出, 但可以从下载代码中获得。它们负责把已用的毫秒数记录到窗体的文本框中。
- btnSync_Click()本身相当简单: 在 for 循环中, mUrlList 中的所有 URL 都用于从 Internet 上加载 XmlDocument 对象。XmlDocument 上的 Load()方法会等待下载完所有的远程资源, 并进行分析, 且没有出现错误。

如果以前没有使用过 System.Xml 命名空间中的类, 没有关系, 因为它与这里的内容无关。我们只使用 XmlDocument.Load()方法, 作为执行时间很长的一个操作例子。

网络请求需要等待 3 x 900 毫秒, 是因为带宽和网络的延迟。先发出 3 个请求(这几乎不需要任何带宽), 再等待所有的请求是比较高效的。传统上, 这需要生成其他线程, 让每个线程执行请求, 并等待响应。多线程编程方式很难做得很好, 错误也难以调试。现在使用委托的异步调用, 就可以获得多线程编程的许多功能, 而不会费太大的劲。在许多情况下, 委托实际上进行的是多线程编程, 即:

- 调用执行时间很长的操作, 而不会冻结用户接口。
- 同时调用多个操作, 更有效地使用资源。

比较复杂的多线程情形(例如有一个监控资源的工作者线程)不能用委托来编程, 更多的信息可参阅 System.Threading 命名空间的文档说明。注意, 这种调用会创建一个消耗资源的新线程。在循环中异步调用委托 10000 次会降低系统的速度。

下面看看异步调用的代码。首先定义一个可用于在 XmlDocument 上调用 Load() 方法的委托:

```
delegate void DocumentLoad(string fileUrl);
```

现在不是直接调用方法,而是创建一个 DocumentLoad 类型的委托,让委托调用方法。要异步调用方法,应使用委托上的 BeginInvoke() 方法。最简单的方式是:

```
private void btnAsync_Click(object sender, System.EventArgs e)
{
    ClearLog();
    XmlDocument[] Documents = new XmlDocument[mUrlList.Count];
    for (int i = 0; i < mUrlList.Count; i++)
    {
        XmlDocument Document[i] = new XmlDocument();
        DocumentLoad MyDelegate = new DocumentLoad(Document[i].Load);
        MyDelegate.BeginInvoke((string)mUrlList[i], null, null);
        AppendLog("Loaded document from " + mUrlList[i]);
    }
}
```

稍后将说明给委托的 BeginInvoke() 方法传送的两个额外的参数。在本例中,只传送了 null。如果使用的是 Visual Studio .NET,就应注意 IntelliSense 不支持 BeginInvoke() 和 EndInvoke() 方法。

这段代码为 3 个 XmlDocument 调用了 Load() 方法。在调用 BeginInvoke() 方法时,该方法会立即返回,但仍在委托创建的另一个线程上加载文档。所以在调用 AppendLog() 方法时,就不能确保是否已完成加载(实际上加载肯定没有完成)。

在某些情况下,我们并不太在意方法何时完成,只是希望启动该方法,然后继续处理主要的任务。在这些情况下,调用 BeginInvoke() 会比较好。但一般要知道执行操作的结果。此时必须利用委托,从 BeginInvoke() 中返回值。

在方法执行计算时,不能获取返回值,此时方法在另一个线程上运行。要从异步方法调用中获取返回值,可以使用 EndInvoke() 方法。在调用这个方法时,委托会继续执行,直到底层方法的原调用结束为止。注意不要调用 EndInvoke(),除非要等待方法执行完毕。EndInvoke() 方法将返回从底层方法中返回的值。EndInvoke() 方法的返回类型总是等于委托本身的返回值。

如果调用的方法使用 ref 或 out 参数,就可以通过 EndInvoke() 方法获取这些参数的新值。这里不详细讨论它们,使用 ILDASM.exe 查看委托,就可以找到这些参数的顺序。

EndInvoke() 总是期望对象把 IAsyncResult 实现为一个参数。对于这个参数,必须使用从 BeginInvoke() 中获取的返回值。这个对象类似于一个接收器,通过 EndInvoke() 接收返回

值。这个机制是必须的，因为可以通过 `BeginInvoke()` 多次调用同一个委托(可能使用不同的参数)。如果调用 `EndInvoke()`，委托就必须确保指定要结束对 `BeginInvoke()` 的哪一个调用；如果不打算调用 `EndInvoke`，就不需要 `IAsyncResult` 对象。

下面看看异步下载的代码，在这段代码中，直到下载完成后，才记录其持续的时间。

```
private void btnAsync_Click(object sender, System.EventArgs e)
{
    ClearLog();
    // As we will now be working with several objects at the same time,
    // we declare arrays of XmlDocument, Delegate and the IAsyncResult
    XmlDocument[] Documents = new XmlDocument[mUrlList.Count];
    DocumentLoad[] Delegates = new DocumentLoad[mUrlList.Count];

    // VS.NET does not support arrays of interface type with
    // Intellisense, but this code is valid C# and compiles fine
    IAsyncResult[] Tickets = new IAsyncResult[mUrlList.Count];

    for (int i = 0; i < mUrlList.Count; i++)
    {
        Documents[i] = new XmlDocument();
        Delegates[i] = new DocumentLoad(Documents[i].Load);
        // The next line starts the loading of the XmlDocument on a
        // different thread. We don't have to wait for its
        // completion now. The two null parameters at the end
        // are for use with callback functions.
        Tickets[i] = Delegates[i].BeginInvoke(
            (string)mUrlList[i], null, null);
        AppendLog("Started loading document from " + mUrlList[i]);
    }
    for (int i = 0; i < mUrlList.Count; i++)
    {
        // Force to wait here until the call is completed. We
        // could have called BeginInvoke multiple times on the
        // same delegate instance, so we need the ticket (of
        // type IAsyncResult) to specify which call we mean exactly.
        Delegates[i].EndInvoke(Tickets[i]);
        AppendLog("Loaded document from " + mUrlList[i]);
    }
}
```

首先循环 URL 的列表, 对于每个 URL, 创建一个 XmlDocument 实例和一个 DocumentLoad 实例, 后者指向 XmlDocument 对象的 Load()方法。XmlDocument 实例和指向它们的委托都存储在数组中。在结束异步操作时, 还需要使用它们。接着在委托上调用 BeginInvoke(), 在第三个数组上存储返回的对象。

在启动 3 个委托后, 在理论上就可以执行其他一些处理器密集的工作, 因为在收到第一个响应之前, 有上百个毫秒的时间。但在这个应用程序中没有执行任何任务, 只是进入第二个循环, 等待三个线程结束。按照开始的顺序, 在委托上调用 EndInvoke()方法, 给它传送合适的参数。现在执行暂停, 直到委托调用的方法结束为止。之后, 就记录时间, 继续执行下一个委托。

在这种情况下, 从记录的时间中可以看出什么? 对于第一个请求, 看不出什么区别。可能会因创建线程和处理线程的开销而产生小的延迟。单击第二个按钮, 会得到如图 6-9 所示的结果。

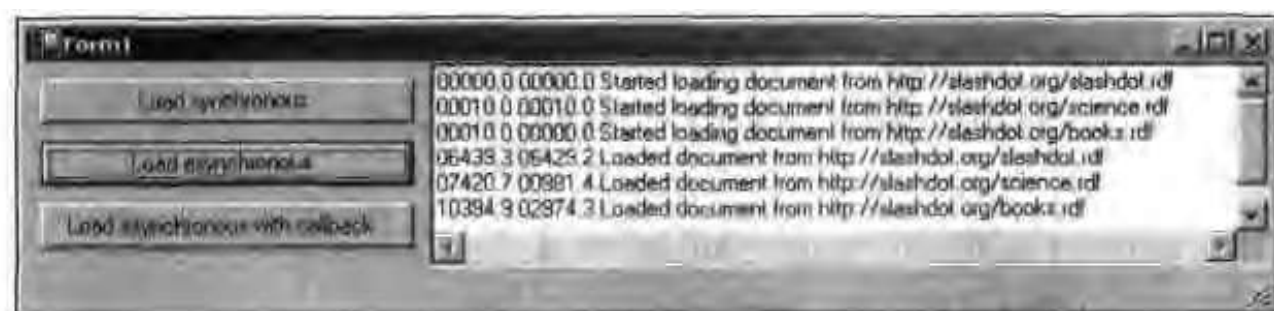


图 6-9

只要需要结果, 就调用 EndInvoke()方法, 在知道操作需要执行多长时间的情况下, 这是一个很好的解决方案。有时只是等待并不合适。有时希望应用程序即使没有获得操作的结果也开始工作(可以禁用需要结果才能正常工作的所有功能, 一旦获得了需要的信息, 就启用它们)。在这些情况下, 可以使用回调函数。此时, 必须执行一个有预定义签名的方法, 告诉委托在执行完毕之后, 立即调用回调方法。

在本例中, 只使用回调方法在日志中创建一行, 也可以从回调方法中调用 EndInvoke(), 访问所执行方法的返回值。其结果如图 6-10 所示。



图 6-10

注意, 3 个下载的开始顺序不必与完成它们的顺序相同, 下面是代码:

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

```
ClearLog();

XmlDocument[] Documents = new XmlDocument[mUrlList.Count];

for (int i = 0; i < mUrlList.Count; i++)
{
    Documents[i] = new XmlDocument();
    DocumentLoad TheDelegate = new DocumentLoad(Documents[i].Load);
    AsyncCallback CallbackDelegate = new
        AsyncCallback(this.ReadyLoading);

    // We start the loading of the XmlDocument, passing it
    // a delegate for calling the ReadyLoading method when
    // it is ready. A reference to the current XmlDocument
    // is also passed in. This will be passed to the callback
    // method as the AsyncState property of the IAsyncResult.

    TheDelegate.BeginInvoke((string)mUrlList[i],
        CallbackDelegate, Documents[i]);
    AppendLog("Started loading document from " + mUrlList[i]);
}

}

private void ReadyLoading(IAsyncResult r)
{
    if (r.AsyncState is XmlDocument)
    {
        XmlDocument doc = (XmlDocument)r.AsyncState;
        AppendLog("Loaded " + doc.BaseURI);
    }
}
```

在底部，可以看到回调函数。它的签名由标准委托 `System.AsyncCallback` 定义。如果所传送对象的 `IAsyncResult.AsyncState` 属性是 `XmlDocument` 类型，就在日志中添加一行。`AsyncState` 属性由启用异步调用的代码来填充。

查看一下启动执行过程的代码，可以看出现在使用了两个委托，一个指向开始执行过程的方法，另一个(类型是 `System.AsyncCallback`)指向回调方法。这个委托作为 `BeginInvoke()` 的第二个参数传送。第三个参数是可选的，它传送的内容会作为 `IAsyncResult.AsyncState` 传送给回调函数。在本例中传送的是 `XmlDocument` 实例本身。

我们在回调方法中使用它记录已加载文档的 URL。

所以，调用委托实例有三种方式：

- 同步调用，等待方法执行完毕。这是委托最常见的用法。
- 异步调用，等待方法执行完毕(通过调用 `EndInvoke()` 方法来完成)，但在方法执行完毕之前调用。
- 异步调用，注册一个回调方法，一旦执行完毕，就给出通知。

每种调用方式适合不同的情形，这取决于方法结果的获取难度和等待的时间长度。注意不能在多播委托上使用异步调用，此时委托会抛出一个异常。

6.2 事件

本章的第一部分学习了如何利用委托调用对象的方法，讲述了如何使用单播委托调用单个方法，以及如何利用多播委托在调用列表中存储多个方法，并按顺序调用它们。还讨论了在有些情况下，如何使用委托来异步调用方法。

本章的第二部分将主要学习事件。.NET Framework 事件模型根据委托来指出当特定事件发生时需要调用的方法。在 .NET Framework 中，事件的实现是使用委托存储订阅的事件处理方法，在事件发生时调用它们。

大多数程序员都很熟悉图形用户界面(GUI)应用程序中事件的使用方法。用户界面元素和应用程序代码之间的交互操作是事件最重要的用法之一，但事件也可以用于其他许多情形。.NET Framework 中的事件是 Observer 设计模式的一种实现。在这种模式中，所监视的对象拥有的方法能注册和注销感兴趣的监视者对象，并给出通知。在 C# 中编写对象时，将自动创建这个功能。

当用户与窗体中的控件进行交互时，产生事件，同时调用事件处理程序来处理该事件。但是，事件并不局限于 GUI 编程，当在应用程序中发生了重要情况时，还可以利用事件作为一种通知相关对象的通用方法。见图 6-11。

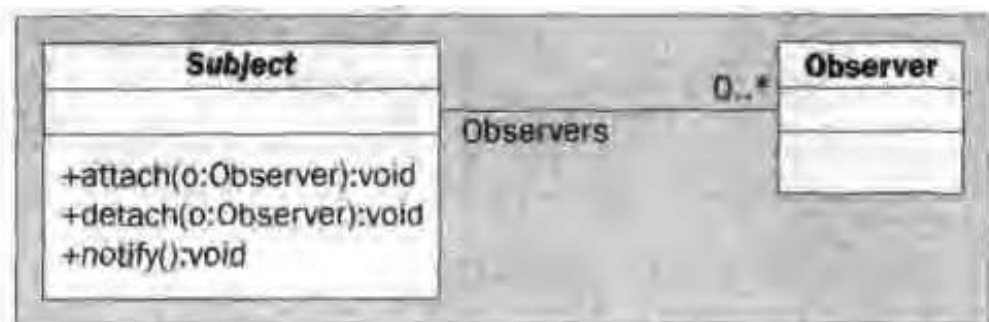


图 6-11

本节将学习以下内容：

- .NET Framework 中的事件体系结构
- 发布事件的方式。

- 预订事件的方式。
- 引发事件的方式。

6.2.1 事件体系结构

下面是对.NET Framework 的事件体系结构的概要总结。同时还提供了与.NET 中的事件处理相关的一些关键术语的正式定义。

引发事件的对象叫做事件源。例如，由于 Button 对象能够引发 Click、MouseEnter 等事件，因此它是一个事件源。事件源发布它可以引发的事件；在 C# 中，使用关键字 `event` 在类定义中发布事件。事件的类型总是委托类型，这个委托类型的签名定义了处理该事件的方法的签名。

定义事件处理程序方法的对象称为事件接收方。事件接收方预订要在事件源上处理的事件。事件接收方提供的事件处理程序方法必须与为该事件预订的方法签名一致。事件源利用多播委托跟踪这些事件处理程序方法。

当发生重要情况时，事件源引发事件。例如，当用户单击屏幕上的按钮控件时，Button 对象就引发 Click 事件。当事件引发后，事件源自动利用它的多播委托调用在每个事件接收方上指定的事件处理程序方法。

图 6-12 说明了事件源对象与事件接收方对象之间的关系。

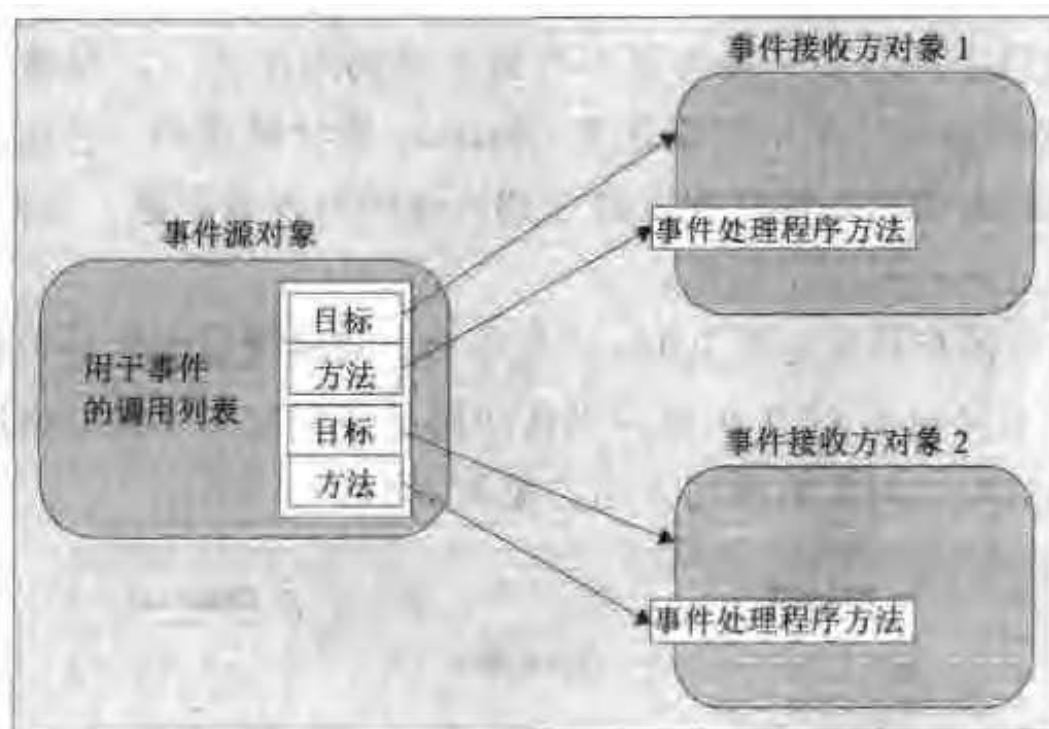


图 6-12

在这个简单的示例中，事件源对象仅仅引发一种事件。事件源对象为该事件保存一个调用列表，以此跟踪预订了该事件的事件接收方。调用列表中的每一项中保存了下面两条信息：

- 目标：事件接受方对象的引用

- 方法：对事件接收方对象的事件处理方法的引用

当事件源对象引发事件时，事件源对象利用调用列表中的信息调用每一个注册的事件处理方法。

注意：

图 6-12 中只显示了事件接收方的事件处理方法。事实上，事件接受方对象还有其他的方法、属性和字段等。

.NET 使用多播委托为维护调用列表中的目标和方法提供了底层结构。

理论问题的介绍到此结束，下面看看如何在 C# 代码中使用事件。首先学习在类中定义事件。

6.2.2 事件的发布和订阅

要在 C# 中发布事件，必须在事件源类中声明事件(Event)成员。对每一个事件，必须指定下面两项内容：

- 事件名。

例如，Button 类发布的事件名有 Click、MouseOver 等。需要为类的实例可能会发生的、影响到其他对象的事情声明事件成员。

- 事件的签名(参数列表)。

当事件源对象引发事件时，它将提供所需的参数值。这些参数值将传递给事件接收方对象的事件处理程序。因此，事件处理程序必须与事件具有相同的签名。事件的签名通过声明事件的类型来指定。这个类型必须是一个委托。

SimpleEvent 项目中的 BankAccount 类是一个展示为事件的类的简单例子。这个项目在源代码下载的 DelegatesEvents\SimpleEvent 文件夹中。

1. 发布事件

首先看看 BankAccount 类中发布事件的代码结构：

```
using System;

public class BankAccount
{
    public BankAccount(string name)
    {
        mBalance = 0;
        mAccountName = name;
    }
}
```



```
    }

    public event EventHandler Overdrawn;

    private double mBalance;
    private string mAccountName;

    public string Name { get { return mAccountName; } }
    public double Balance { get { return mBalance; } }

    public void Credit(double Amount)
    {
    }

    public void Debit(double Amount)
    {
    }
}
```

这个类相当简单。它有两个私有字段 `mBalance` 和 `mAccountName`。它们都可以通过只读属性来访问。账号名只能通过构造函数来设置,而余额可以调用 `Debit()`和 `Credit()`方法来修改。但这两个方法现在还没有提供实现(稍后在处理引发的事件时,将介绍其实现)。惟一要注意的代码行是定义事件成员的代码。

本例把 `System.EventHandler` 用作事件的类型。`System.EventHandler` 是一个内置的委托,专门用于不需要给事件接收方传送任何额外信息的简单事件。`Framework` 中的许多事件都使用它(例如 `Button` 中的 `Click` 事件)。

事件是类的成员,因此可以有访问修饰符。默认的可访问性是 `public`,因为事件的本质是把信息传播给应用程序中的其他对象。但是,如果需要,也可以限制事件的可访问性:

- 可以将事件声明成 `private`,这就意味只有类自身的方法才能注册为事件处理程序。利用该方法能够在类中触发行为。例如:在后台线程中,每隔 1 秒触发一次 `Tick` 事件,在类中定义事件处理方法,来每隔 1 秒执行任务。这种模式是很少见的,但是利用该模式能够生成一流的代码。
- 将事件声明为 `internal`,这就意味着只有同一个程序集中的方法才能注册为事件处理程序方法。在跨多个程序集的大型应用程序中可以使用该方法。为了减少程序集之间的依赖性,可以将类成员的访问性设为 `internal`,这样,它们就不能在其他程序集中访问了。采用这种方法,在一个程序集中修改代码时,不会对其他程序集中的代码产生不利影响。第 8 章将详细学习程序集。

- 将事件声明为 `protected`，这就意味着类自身的方法，或者从该类继承的类中的方法都可以注册为事件处理程序方法。

本例中事件名为 `Overdrawn`。本章后面的部分，将学习当顾客需要从账户中提取过多的现金时如何引发这个事件。

事件不能有返回类型。事件只有从事件源对象到事件接收方对象的单向信息流。在事件定义中指定返回类型是错误的。同样，当事件使用委托时，很明显可以使用并不是由事件提供的委托的方法。只使用事件的底层结构，而忽略委托的其他功能是不对的。

代码看起来很简单，但完成了好几项工作：

- 为事件接收方发布事件名和签名
- 为注册和注销事件接收方创建注册方法
- 跟踪在调用列表中注册的事件接收方

把文件编译为 `.dll` 文件，使用 `ILDASM.exe` 查看可执行文件，可以看到在类中有下面 3 个功能见图 6-13。

```
C:\Class Design\Ch 06\DelegatesEvents\SimpleEvent>csc /t:library BankAccount.cs
```

```
Microsoft (R) Visual C# .NET Compiler version 7.00.9466
```

```
for Microsoft (R) .NET Framework version 1.0.3705
```

```
Copyright (C) Microsoft Corporation 2001. All rights reserved.
```

```
C:\Class Design\Ch 06\DelegatesEvents\SimpleEvent>ildasm BankAccount.dll
```



图 6-13

在编译好的 MSIL 代码中，有 4 个与事件声明相关的成员：

- **EventHandler 类型的私有字段 OverDrawn**

该字段是 `System.EventHandler` 的一个实例，用于把所有已注册的事件处理程序存储为一个多播委托。在实例化对象时，该字段设置为 `null`。注意即使事件本身是公共的，这个字段也总是私有的。

- **公共方法 `add_Overdrawn()`**

该方法由编译器生成，用来为 `Overdrawn` 事件添加新的事件处理程序。当其他的事件接收方预订该事件时调用该方法。注意，`add_Overdrawn()` 方法接收一个类型为 `EventHandler` 的参数。该参数是 `EventHandler` 委托类型的实例，为 `Overdrawn` 事件指定一个新的事件接收方对象和事件处理程序方法。

打开 IL，查看 `IL_0002`、`IL_0007` 和 `IL_0008` 语句。这些语句在 `BankAccount` 对象中将新的委托实例加入到 `Overdrawn` 多播委托成员中。注意，MSIL 调用 `System.Delegate::Combine` 方法，将新委托添加到多播委托中。

- **公共方法 `remove_Overdrawn`**

该方法由编译器生成，用于为 `Overdrawn` 事件删除一个现有的事件处理程序。当事件接收方取消订阅这个事件时调用该方法。查看 IL，可以发现 `remove_Overdrawn` 和 `add_Overdrawn` 很相似，二者之间的差别在于 `remove_Overdrawn` 调用 `System.Delegate::Remove` 方法删除指定的委托。

- **事件声明 `OverDrawn`**

该成员仅在使用该类的类编译过程中使用，用来为 `Overdrawn` 事件标识新添加或删除的事件处理程序的方法名。它的格式与第 5 章介绍的属性成员类似，只不过属性成员指向 `getter` 和 `setter` 方法。`Overdrawn` 的 MSIL 代码如图 6-14 所示。

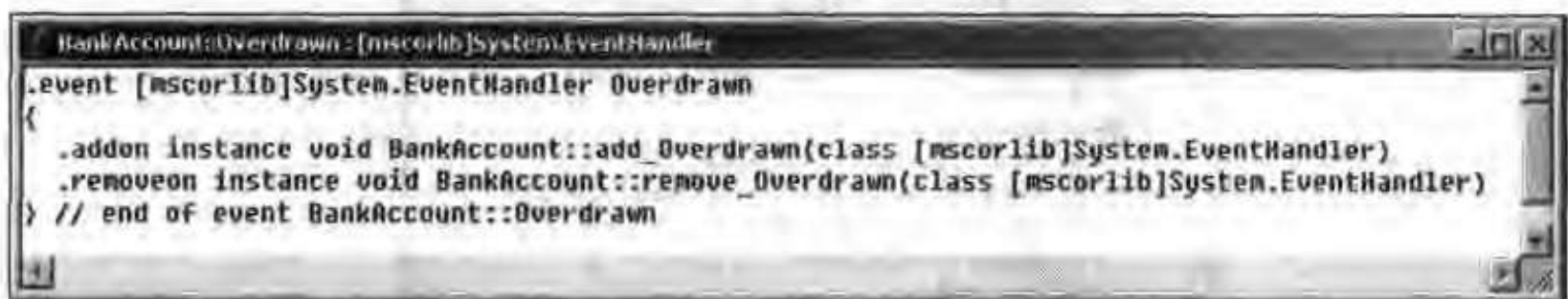


图 6-14

显然，C#编译器在事件名的前面加上 `add_` 和 `remove_`，来创建注册方法。也就是说，不能在自己的方法中使用这些名称。下面的代码就不能编译：

```
public event EventHandler Overdrawn;
public void add_Overdrawn(EventHandler e)
{
}
}
```

在编译这段代码时，会产生一个异常 `'Class 'BankAccount' already defines a member`

called 'add_Overdrawn' with the same parameter types'.

2. 注册已发布的事件

类发布了事件后，就要把一个方法注册为事件处理程序。为了说明注册的方式，下面看看同一个项目中的 `BankAccountForm` 类的代码。

除了通常的窗体代码之外，还有如下方法：

```
private BankAccount mBankAccount;
public BankAccountForm()
{
    //
    // Required for Windows Form Designer support
    //
    InitializeComponent();

    mBankAccount = new BankAccount("Teun Duynstee");
    RefreshTitle();
    mBankAccount.Overdrawn += new
        EventHandler(this.OverdrawnHandlerMethod);
}

private void RefreshTitle()
{
    this.Text = String.Format("Account {0}, balance: {1}",
        mBankAccount.Name, mBankAccount.Balance );
}

private void btnCredit_Click(object sender, System.EventArgs e)
{
    mBankAccount.Credit(Double.Parse(txtAmount.Text));
    RefreshTitle();
}

private void btnDebit_Click(object sender, System.EventArgs e)
{
    mBankAccount.Debit(Double.Parse(txtAmount.Text));
    RefreshTitle();
}

private void OverdrawnHandlerMethod(object sender,
    System.EventArgs e)
{
    MessageBox.Show("The bank account has been overdrawn. " +
        "The current balance is " + mBankAccount.Balance,
```

```
        "Overdrawn",  
        MessageBoxButtons.OK,  
        MessageBoxIcon.Information);  
    }
```

注意下面的要点:

- 在窗体的构造函数中, 创建了 `BankAccount` 的一个实例, 并把方法 `OverdrawnHandlerMethod()` 设置为 `Overdrawn` 事件的处理程序添加到 `Overdrawn` 事件中。其语法很烦琐, 创建了一个由事件指定其类型的委托。这个委托指向处理程序方法。使用 `+=` 运算符, 把这个委托添加到事件的已注册的事件接收方列表中。`+=` 语法是 C# 中把处理程序方法和事件连接起来的惟一方式。在其他 .NET 语言中, 语法有所不同, 但在 MSIL 中, 结果是相同的。因此, 使用其他 .NET 语言的程序员可以注册这个事件, 不会出问题, 也不需要特别的考虑。
- 方法 `OverdrawnHandlerMethod()` 仅在消息框中显示消息, 并没有使用所传送的两个参数。第一个参数是对引发事件的对象的引用。在本例中, 就是 `mBankAccount` 对象。已经有一个对这个对象的引用了, 所以本例不需要把所传送的参数的类型转换为 `BankAccount`。在第二个参数中传送的是 `System.EventArgs` 对象, 它也没有什么用。在比较复杂的事件中, 这个对象包含事件的其他附加信息。本章的后面将讨论这些事件参数。
- `RefreshTitle`、`btnCredit_Click` 和 `btnDebit_Click` 方法使用 `BankAccount` 上可用的方法和属性, 允许用户在 `BankAccount` 对象上执行操作。

下面看看标记为 Windows Form Designer generated code 的部分代码:

```
// A lot of other controls and properties have been omitted here  
this.btnCredit = new System.Windows.Forms.Button();  
this.btnDebit = new System.Windows.Forms.Button();  
  
// btnCredit  
this.btnCredit.Text = "Credit";  
this.btnCredit.Click += new  
                        System.EventHandler(this.btnCredit_Click);  
  
// btnDebit  
this.btnDebit.Text = "Debit";  
this.btnDebit.Click += new System.EventHandler(this.btnDebit_Click);
```

在实例化按钮并设置属性后, 创建一个 `EventHandler` 委托实例, 指向窗体上的方法。接着把委托实例添加到按钮的 `Click` 事件中。

在图形化开发环境中使用窗体的编辑器时, 窗体上的每个事件一般只有一个事件处

理程序，任何事件处理程序方法都只处理一个事件。

3. 引发事件

为了使这个简单的例子启动，需要引发事件。显然，Overdrawn 事件应在 BankAccount 类的 Debit() 方法上引发。前面让 Debit() 和 Credit() 方法的实现暂时为空，下面在 BankAccount 类中给它们添加如下代码：

```
public void Credit(double Amount)
{
    mBalance += Amount;
}

public void Debit(double Amount)
{
    mBalance -= Amount;
    if (mBalance < 0)
    {
        if (Overdrawn != null)
        {
            Overdrawn(this, new EventArgs());
        }
    }
}
```

Credit() 和 Debit() 对 mBalance 字段进行了相应的修改，但在 Debit() 中，还要检查结果是否小于 0。如果是，就要引发 Overdrawn 事件。Overdrawn 事件实现为一个多播委托，所以引发事件会调用委托。这里要注意两点：

- 在注册事件之前，Overdrawn 字段等于 null。如果试图调用 null，就会抛出一个运行时异常，在 C# 中引发事件时，必须先检查所调用的委托是否为 null。
- 与所有的预定义事件处理程序类一样，System.EventHandler 的签名返回 void，但有两个参数。在第一个参数中，必须传送对事件引发对象(this)的引用。第二个参数必须是 System.EventArgs 的一个实例。System.EventArgs 类似于哑元类。它可能根本不包含任何信息，用于不传送任何信息的事件(例如 Overdrawn)，以及这类事件的基类。

下一节(创建传送信息的事件)将介绍它。在代码中，调用 new EventArgs() 创建 System.EventArgs 的一个实例，EventArgs() 还有一个静态属性 Empty，它返回类的一个新实例。有的人比较喜欢使用 EventArgs.Empty 语法，而不喜欢 new 语法。

注意:

Debit()和 Credit()方法还应检查负数,只要出现了负数,就抛出一个异常,但这超出了本例的范围。

这就完成了这个简单的事件例子。本例创建了一个带有 Overdrawn 事件的 BankAccount 类。在一些特殊的情况下(调用 Debit()方法,且返回的余额小于 0 时),类就会引发事件。在示例应用程序中,有一个主窗体,它使用类的实例执行银行交易事务。在加载窗体时,它会创建 BankAccount 的一个新实例,并注册事件。在事件发生时,窗体就显示一个消息框,如图 6-15 所示,说明所发生的情况。

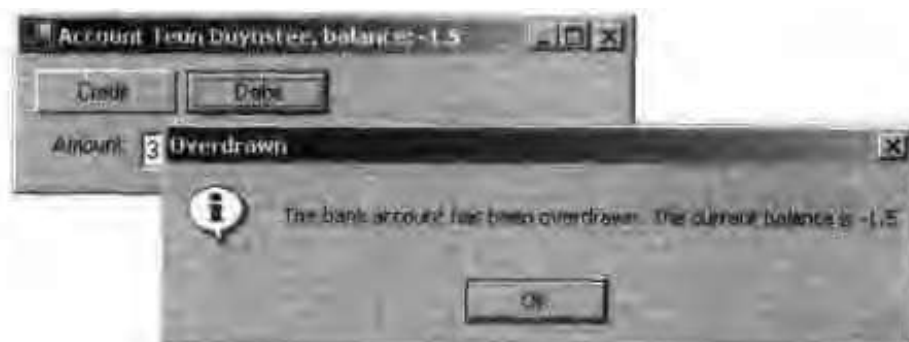


图 6-15

4. 创建传送信息的事件

介绍了简单的事件后,下面介绍另一个示例应用程序: DelegatesEvents\Advanced Events。这个应用程序用于显示几个高级事件功能。该应用程序有一个较大的银行管理窗体,用于创建许多小的银行账号窗体(与 SimpleEvent 中的窗体类似)。银行经理可以选择在事件日志屏幕上记录了事件的那些激活账号。除了 Overdrawn 事件之外,银行账号还会为较大的存款引发事件(存款大于阈值时)。当经理修改了大额存款阈值后,所有的银行账号窗体都会收到这个通知。

运行这个应用程序,结果如图 6-16 所示。

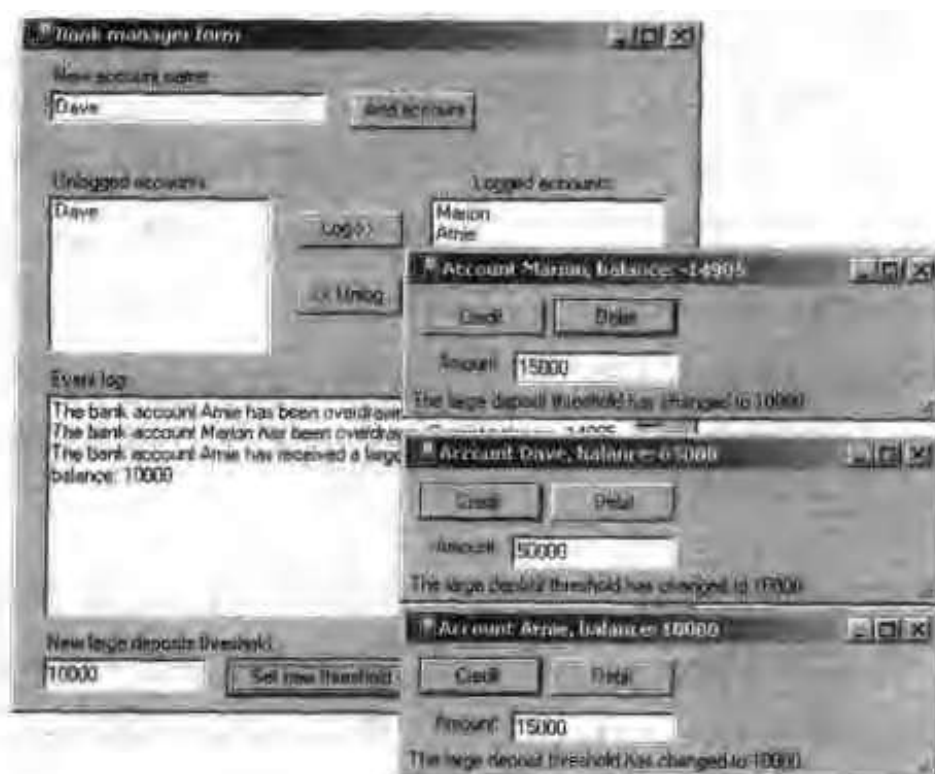


图 6-16

首先定义给事件目的地传送信息的事件。在大多数情况下,仅传送通知是不够的。在本例中,有一个事件叫做 `LargeDeposit`。当一些对象对 `BankAccount` 实例上进行的大额存款感兴趣时,它们就希望知道该存款额有多大。`System.EventHandler` 类型不允许把这个存款额传送给事件接收方,但可以定义另一个委托来完成这一任务。本章的第一部分介绍了具体的做法。有了这个经验,就可以使用下面的代码:

```
// Bad practice, do not do this
public event MyDelegate LargeDeposit;

public delegate void MyDelegate(BankAccount b, double Amount);
```

可以声明一个委托,其签名不仅包含引发事件的 `BankAccount`, 还包含一个包含存款额的 `double`。这种方式是可行的,但违反了 Microsoft 的事件、委托及其参数的命名和签名的约定。

委托类型的名称应遵循 `XxxEventHandler` 格式。签名是标准化的,返回值应为 `void`, 且有两个参数:

- 第一个参数是引发事件的对象引用。它常常声明为一个对象,允许任何类型的对象使用事件。但如果能肯定事件只由 `BankAccount()` 引发,也可以输入它,避免许多不必要的类型转换。
- 第二个参数包含 `System.EventArgs` 的一个子类的实例。这个类应包含必须传送给事件处理程序的所有信息。类的名称应与委托名相同,但后缀应是 `EventArgs`, 而不是 `EventHandler`。

遵循这个约定,其他程序员就更容易理解代码,还有一些比较直接的优点。事件处理程序方法的签名遵循相同的格式,所以可以使用一个方法,专门为其他事件类型处理 `System.EventHandler` 类型的事件。

首先看看声明,它应遵循事件命名约定,稍后解释各个部分:

```
public delegate void BankTransactionEventHandler
    (BankAccount sender, BankTransactionEventArgs args);

public class BankTransactionEventArgs : EventArgs
{
    public BankTransactionEventArgs(double amount)
    {
        mAmount = amount;
    }
    private double mAmount;
    public double Amount
```



```
{
    get{return mAmount;}
}

public class BankAccount
{
    public event BankTransactionEventHandler LargeDeposit;
    // Many other members omitted
}
```

最前面的是委托类型的声明，它命名为 `BankTransactionEventHandler`。

接着是 `BankTransactionEventArgs` 类的定义。它是 `EventArgs` 的一个子类，并定义了一个属性 `Amount`，这个属性是只读的，只能通过构造函数来设置。

最后是 `BankAccount` 中的 `LargeDeposit` 事件声明为 `BankTransactionEventHandler` 类型。

现在，在进行大额存款时，需要引发这个事件。下面看看其代码：

```
private static double mThreshold = 10000;

public void Credit(double Amount)
{
    mBalance += Amount;
    if (Amount >= mThreshold)
    {
        if (LargeDeposit != null)
        {
            LargeDeposit(this, new BankTransactionEventArgs(Amount));
        }
    }
}
```

其中添加了两个内容：

- 一个私有静态字段，它保存了大额存款的阈值。这显然是相当随意的，但至少是对所有 `BankAccount` 的实例都相等的值。后面会添加修改这个值的功能和通知进行了修改的功能。
- 添加了调用委托的代码，该委托保存了已注册的事件处理程序。当然，还必须检查委托是否为 `null`。在调用它时，给它传送 `BankTransactionEventArgs` 的实例。对它实例化时，在构造函数中传送存款额。事件处理程序会使用这个对象获取

存款额。使 EventArgs 对象保持不变(使属性为只读)是非常重要的,因为已注册的事件处理程序会按顺序获取所传送的同一个实例。我们不知道为这个事件注册了什么处理程序,也不知道它们试图对 EventArgs 实例执行什么操作。

遵循这个约定,将允许其他人定义可以处理任何事件的事件处理程序。如果一个方法可以处理 System.EventHandler 类型的事件,就可以用于处理所有的事件(这些事件都遵循这个约定)。处理方法需要两个参数,一个是 object 类型(以传送任何对象),另一个是 EventArgs 类型。BankTransactionEventArgs 是 EventArgs 的一个子类,所以方法也接受这个类型。

遵循事件签名的约定,会使事件更易于使用,因为其他程序员可以从名称中理解类的用途,用于更广泛的用途,因为它们可以与标准事件处理程序关联起来。

5. 定义静态的事件

前面把事件当做对象的实例成员,也可以把事件定义为静态的(类型的成员,而不是实例的成员)。静态事件比实例事件少得多,其原因在下一节介绍如何订阅事件时说明。

首先研究一下 AdvancedEvents 示例的 BankAccount 类。在这个类中有如下代码:

```
public static event EventHandler ThresholdChanged;
private static double mThreshold = 10000;
public static double Threshold
{
    set
    {
        mThreshold = value;
        if (ThresholdChanged != null)
        {
            ThresholdChanged(null, EventArgs.Empty);
        }
    }
    get{return mThreshold;}
}
```

这里定义了一个新事件,除了新添的 static 关键字之外,与一般的事件定义是完全相同的。我们还定义了一个静态属性 Threshold,以展示 mThreshold 字段。在设置新值时,会调用 ThresholdChanged 委托(在检查它不是 null 之后)。

如果查看一下注册 ThresholdChanged 事件的 BankAccountForm 类,就会发现注册代码有一些不同。窗体的构造函数包含如下代码:

```
mBankAccount = new BankAccount(accountName);  
BankAccount.ThresholdChanged += new  
    EventHandler(this.AlertThresholdChanged);
```

注意:

注册的事件不是 mBankAccount 实例的一个成员, 而是 BankAccount 类本身的成员。实际上, 要注册这个事件, 并不需要类的实例。

在 bankmanagerform 窗体中, 可以通过 BankAccount 上的静态属性 Threshold 修改阈值:

```
private void btnSetThreshold_Click(object sender, System.EventArgs e)  
{  
    BankAccount.Threshold = Double.Parse(txtThreshold.Text);  
}
```

只要银行经理单击了 btnSetThreshold 按钮, BankAccount 上的静态字段就修改值, 改变 BankAccount 的所有实例的行为。修改阈值后, 所有的 BankAccountForm 实例都会通过 AlertThresholdChanged()方法的调用收到这个事件的通知。如果在 ildasm 中查看可执行文件, 就会看到一个静态事件(看起来与正常的事件相同)生成了一个私有委托的字段、两个注册方法和一个事件成员。只是这些对象都标记为静态, 如图 6-17 所示。



图 6-17

6. 动态注册和注销

事件处理程序是在对象的创建期间注册的，在对象的整个生存周期中都有效。在使用窗体构建器关联事件时，窗体上控件的事件处理程序总是这样。但是，这不是必须的，可以在程序的运行期间，在接收某些通知的过程中注册或注销事件。例如，在示例应用程序中，允许银行经理选择通过日志窗口监视的银行账号。对于这些被选中的账号来说，Overdrawn 和 LargeDeposit 事件会在日志文本框中生成一行文本。

为此，要使用两个方法，一个用于记录 Overdrawn 事件，一个用于记录 LargeDeposit 事件。在创建 BankAccountForm(它又创建了 BankAccount 对象)时，把新建的 BankAccount 添加到 lstUnlogged 列表框中。只有用户单击了按钮 Log >>，才把所选 BankAccount 上的事件关联到相应的方法上。

```
private void btnLog_Click(object sender, System.EventArgs e)
{
    if (lstUnlogged.SelectedItem is BankAccount)
    {
        BankAccount b = (BankAccount)lstUnlogged.SelectedItem;
        b.Overdrawn += new EventHandler(this.OverdrawnHandler);
        b.LargeDeposit += new
            BankTransactionEventHandler(this.LargeDepositHandler);
        lstLogged.Items.Add(b);
        lstUnlogged.Items.Remove(b);
    }
}
```

为了避免运行期间的类型转换异常，首先要测试所选的列表项是否为 BankAccount 的一个实例(在这个应用程序中，惟一可能返回的是 null)。如果是，就把它的类型转换为 BankAccount，并把 OverdrawnHandler() 方法注册到 Overdrawn 事件上，把 LargeDepositHandler() 方法注册到 LargeDeposit 事件上。之后，把所选的 BankAccount 从未记录的列表项移动到已注册的列表项上。

其逆过程非常类似，如下：

```
private void btnUnlog_Click(object sender, System.EventArgs e)
{
    if (lstLogged.SelectedItem is BankAccount)
    {
        BankAccount b = (BankAccount)lstLogged.SelectedItem;
        b.Overdrawn -= new EventHandler(this.OverdrawnHandler);
        b.LargeDeposit -= new
```

```
BankTransactionEventHandler(this.LargeDepositHandler);
```

```
lstLogged.Items.Remove(b);  
lstUnlogged.Items.Add(b);  
}  
}
```

可以想像在这两个事件处理程序方法中会发生什么，所以这里不再说明。在“创建传送信息的事件”一节的屏幕图中，说明了如何创建 3 个账号，而银行经理窗体中的事件只来自于这 3 个账号中的两个。账号 Dave 仍在未记录账号的列表中。

事件概念的优点在这个例子中显露无遗：接收事件的对象确定感兴趣的事件(或何时引发这些事件)。方法仅为这些事件实现。引发事件的对象不过问谁应接收哪个通知，只是调用事件委托，并确保调用所有已注册的事件处理程序。事件处理程序的管理部分隐藏在两个方法 `add_eventname()` 和 `remove_eventname()` 中。

7. 定义自己的注册方法

因为注册的管理是隐藏的，所以可以避免管理错误。但有时需要修改生成的 MSIL，完成不同的工作。为此，可以为事件定义自定义的添加和删除方法。

C#提供了一种特殊的语法，非常类似于在属性上定义访问器方法的语法。事件声明会生成方法和一个委托类型的私有字段。如果选择定义自定义的访问器方法，就必须定义 `add()` 方法、`remove()` 方法，和调用列表的存储位置(可以选择另一个实现，而不使用多播委托)。

在示例应用程序 `DelegatesEvents\CustomAddRemove` 中，演示了如何修改注册方法。该应用程序有两个按钮，一个用于创建新的事件目的地，把它注册到事件源上，另一个按钮用于从事件源中引发事件。所有的对象都可以访问用于记录的文本框。

- 事件源在引发事件之前记录一行。
- 事件接收方在接收到一个事件通知时记录一行。
- 事件源在发现注册或注销了一个事件接收方时记录一行。这也是要演示的一个功能。把记录代码放在事件注册方法中。

应用程序运行结果如图 6-18 所示。

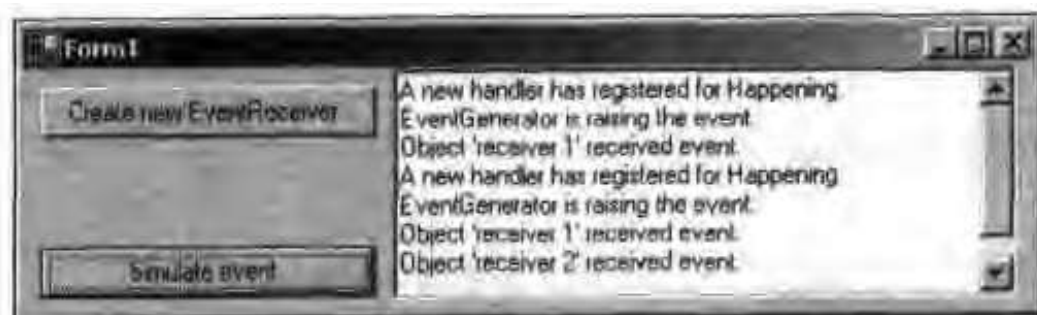


图 6-18

这里只列出了事件声明的代码。读者现在应很熟悉其他功能了。

```
private EventHandler mHappening;

public event EventHandler Happening
{
    add
    {
        mHappening = (EventHandler)Delegate.Combine(mHappening, value);
        mTextBox.Text += "A new handler has registered for
                           Happening\r\n";
    }
    remove
    {
        mHappening = (EventHandler)Delegate.Remove(mHappening, value);
        mTextBox.Text += "A new handler has unregistered for
                           Happening\r\n";
    }
}
```

注意在事件声明中，声明了两个代码块。add 后面的代码块最终编译为一个方法 `add_Happening()`，而在 remove 后面的代码在 MSIL 中编译为 `remove_Happening()` 方法。除了在日志文本框中添加一些文本的代码行之外，这里的实现与默认的实现完全相同。我们声明了一个 `EventHandler` 类型的私有字段，来保存所有的事件处理程序方法。而对于方法的管理，则只在 add 方法中调用了 `Delegate.Combine()`，在 remove 方法中调用了 `Delegate.Remove()`。这两个方法都需要一个 `EventHandler` 的实例，通过 value 关键字就可以访问该实例。

另一个区别是命名。在使用默认的实现时，编译器用与事件名相同的名称生成了一个私有字段(在本例中，私有字段称为 `Happening`)。在 MSIL 中这是允许的，事件和私有字段可以有相同的名称。但在 C# 中，这会出现命名冲突：编译器会抛出一个异常 `The class 'CustomAddRemove.EventGenerator' already contains a definition for 'Happening'`。所以必须为字段选择另一个名称(这里选择 `mHappening`)，在引发事件时，也必须使用这个名称。

```
public void SimulateEvent()
{
    mTextBox.Text += "EventGenerator is raising the event.\r\n";
    if (mHappening != null)
    {
```



```
        mHappening(this, EventArgs.Empty);  
    }  
}
```

大多数开发人员不需要定义定制的注册方法，也不需要添加一些日志，因此，这看起来似乎并不是一个非常可信的应用。而且，在一些情况下，这种技术可以使代码更简洁和快速。例如在 Framework 库中，大多数控件(包括 Windows 窗体和 ASP.NET)都使用定制的注册方法。

System.Windows.Forms.Control 类是窗体中几乎所有对象的基类：所有的控件，如按钮、标签、复选框，和窗体本身。控件有 58 个事件。在一般的窗体中，有许多控件，其中的大多数都非常简单，大多数事件也很少被使用。另外，控件的每个实例会占用一些内存来存储私有的委托字段(以跟踪事件处理程序)。为了使控件轻型，Framework 的实现人员决定不给每个事件使用一个委托实例，而是有一个集合来存储控件的所有事件处理程序。在控件需要引发某个事件时，它知道如何从集合中获取正确的委托，并调用它。如果根本没有使用任何事件，由于集合是空的，控件就仅使用内存。

注释：

实际上，这个方法在控件本身上无法实现，只在其基类 System.ComponentModel.Component 上实现。这个类展示了受保护的属性 Events，所有的子类都可以使用该属性存储较少使用的事件。

6.2.3 事件在 .NET Framework 的类型接口中的作用

.NET Framework 中的事件与属性和方法一样都是成员。子类从其基类中继承了所有的事件，包括处理程序方法。甚至接口也可以包含事件。

在建立展示事件的类时，一般把事件及其实现看做一个实体，一个成员。但在 MSIL 中，事件、存储调用列表的私有字段和注册方法是不同的项。事件只是发布的名称，和它对两个注册方法的引用。在编译过的 .NET 程序集中，注册方法与其他方法一样。另外，保存多播委托的私有字段和与事件无关的私有字段一样。.NET 运行时不知道它与事件有关。在继承和重写时可以解释事件的行为。

如果在基类中，事件声明为虚拟，就可以重写它。子类中的声明如下所示(这段代码在示例应用程序中没有出现)：

```
// Valid code, but will not work as you would expect  
public override event EventHandler Happening;
```

从上面的注释可以看出，这段代码可以编译，但不能像期望的那样工作。重写的事

件生成了事件发布的新版本和注册方法的新版本。这些都重写了原来的实现，还创建一个新的私有字段，来保存多播委托。现在，当其他类注册事件时，事件处理程序就会存储在子类的新私有字段中。但是，如果基类试图引发事件，就会调用基类的私有字段中的委托。这是因为事件的默认实现把字段创建为私有，基类和子类不能引用同一个委托。因此这种形式的重写是没有什么用的。

要实现可以重写的事件，需要定制存储字段的实现。可以保留 `add()` 和 `remove()` 方法的默认实现，但把委托字段声明为受保护的。然后在子类中，就可以有效地重写注册方法，同时使用受保护的字段来存储和引发事件。

```
class BaseClass
{
    protected EventHandler mHappening;

    public virtual event EventHandler Happening
    {
        add
        {
            mHappening = (EventHandler)Delegate.Combine(mHappening, value);
        }
        remove
        {
            mHappening = (EventHandler)Delegate.Remove(mHappening, value);
        }
    }
}
class SubClass : BaseClass
{
    // No field declaration here
    public override event EventHandler Happening
    {
        add
        {
            mHappening = (EventHandler)Delegate.Combine(mHappening, value);
            mTextBox.Text += "A new handler has registered for  
Happening\r\n";
        }
        remove
        {
            mHappening = (EventHandler)Delegate.Remove(mHappening, value);
        }
    }
}
```



```
mTextBox.Text += "A new handler has unregistered for  
Happening\r\n";  
    }  
}  
}
```

如果基类使用默认的实现，就不能实现新的注册方法，且仍能访问原字段。

6.3 小结

本章中讲述的内容很多，介绍了一些新概念和编程方式。即使建立非常简单的 Windows Forms 应用程序和 ASP.NET Web Forms 应用程序(一般称为网站)，很好地理解 .NET Framework 中事件的工作方式也是至关重要的。

本章介绍了如何在一般编程任务中使用委托，例如指定要调用哪个方法，但在准备好之后再执行它，或者在一个单独的线程上执行任务。另外，委托可以使用事件机制，也可以不使用事件机制。

.NET Framework 中的事件使用了多播委托。可以定义委托类型，指定事件处理程序方法的方法签名。事件机制隐式地创建了委托对象，使用这些委托对象可以跟踪每个事件的事件处理程序。

本章还通过许多示例，说明了在 C# 中事件和委托的最好实践方法。同时，还利用一些篇幅分析了 MSIL 代码，说明了事件和委托的内部工作原理。这对如何在应用程序中更有效地使用事件和委托是很有帮助的。

第7章 继承和多态性

继承和多态性是进行面向对象设计和开发的两个重要概念。传统上，所有的面向对象语言都支持这两个概念，C#中的语法类似于 C++。

继承允许构建类层次结构。类层次结构是一组相关的类，因为它们共享某些内容。例如，可以为哺乳动物定义一个类层次结构，其中人、猴子和老鼠是三种不同类型的哺乳动物。每种哺乳动物都是不同的，但它们的基本类型都是 `mammal`。因此，可以创建一个类层次结构，建模哺乳动物的行为，在该层次结构的最上层是 `Mammal` 类，它是一个基类，其他三个类是其派生类。派生类不仅继承了基类的所有成员(包括其属性、字段和方法，只要它们不是基类的私有成员即可)，还可以给所继承的 `Mammal` 类添加新行为。例如，`Human` 类有一些属性或方法是人所独有的，例如 `Job Title` 或 `Telephone Number`，`Monkey` 类有一个 `Tree` 属性是猴子独有的，指定了猴子所在树的位置。简言之，所有的这些类都有哺乳动物共有的元素，也有表示其独有功能和特性的专用元素。

继承避免了代码的重复，允许重用代码，也就是说，每个类不需要复制 `Mammal` 类的实现。因此，可以创建一个类层次结构，用于在代码中描述类的关系。可以使用继承创建事物的类别和关系，把复杂的问题分解为更易于管理的小块。

多态性由继承支持。因为所有的派生类都继承了基类的基本功能，所以可以把派生类的实例看做基类的实例。这就是取代的原则——可以提取出基类的一个实例，用派生类的一个实例替换它，而代码仍能继续工作——仍可以调用方法，并访问实例中原本应在基类上访问的属性。把一种类型的对象视作另一种类型的实例就叫做多态性。

继承为派生类提供的另一个功能是它们可以重写基类的默认实现实例，并提供属性和方法的定制实现。在这些情况下，用于访问基类功能的代码会发现基类的实例被另一个派生类的实例所替代，而调用基类上的方法，会执行派生类上的代码。这个过程在 .NET 中由虚方法分派技术支持，本章的后面将讨论其工作原理。有时需要防止派生类重写功能，有时让派生类具有重写功能又是至关重要的。本章将解释如何控制继承。

.NET 与大多数面向对象语言一样，一般把父类称为基类，子类称为派生类。多态性可以用两种方式来支持：定义一个由派生类实现的接口，或实现继承和方法重写，激活继承方法的新实现。其他编程语言对此使用不同的术语，例如，Java 把基类称为超类，派生类称为子类。C++开发人员把继承描述为从基类中派生，扩展其操作和属性。

注意：

本章将互换使用术语“操作”和“属性”以及“方法”和“字段”，操作和属性是

概念上的术语，而方法和字段是在代码中描述这些概念的实现。

下面的几节将介绍继承和多态性中的一些关键概念，但没有使用代码，以便读者从概念上更完整地理解它们。

7.1 继承

为了说明继承和多态性的用法，下面使用 UML(统一建模语言)图来解释类的关系。

UML 的参考书有 Wrox 出版社出版的“Instant UML”(ISBN 1-86100-087-1)。在.NET 中使用 UML 的详细信息，也可参阅最近出版的“Professional UML with Visual Studio .NET”(ISBN 1-86100-795-7)。

UML 可以用于描述整个应用程序，说明类之间的关系。这里要提及的是 OO 开发中的两种关系，一种是 is a 关系，一种是 has a 关系。类是应派生于另一个类，还是应包含另一个类总是有规则可循的(这是 C#中使用的 is 关键字的背景，本章稍后讨论)。

这些关系最好根据前面讨论的 Mammal 类层次结构来理解。Monkey 是一种哺乳动物，因为它可以完成哺乳动物能完成的所有动作。Monkey 还有一个 Tree。Monkey 不能完成 Tree 能完成的动作，但从 Monkey 中获取 Tree，就可以访问 Tree 的功能。

is a 关系是多态性的基础。如果 Monkey 是 Mammal，则应用于 Mammal 的所有代码就都可以应用于 Monkey。因此可以用 Monkey 代替 Mammal，并应用多态性。在.NET 中，多态性是通过继承来提供的，因此.NET 中的 is a 关系就可以用继承来建模，并应使 Monkey 类继承 Mammal。

has a 关系也对继承有影响。如果 Monkey 有 Tree，则任何派生于 Monkey 的类就都有 Tree。同样，如果 Mammal 有 LatinName，而 Monkey 是 Mammal，Monkey 也应有 LatinName。

图 7-1 所示的 UML 图表示 Mammal 和派生于 Mammal 类的所有类之间的基本关系。其中的箭头说明了继承的关系。在这个例子中，Mammal 类是基类，派生类是 Human、Monkey 和 Mouse。

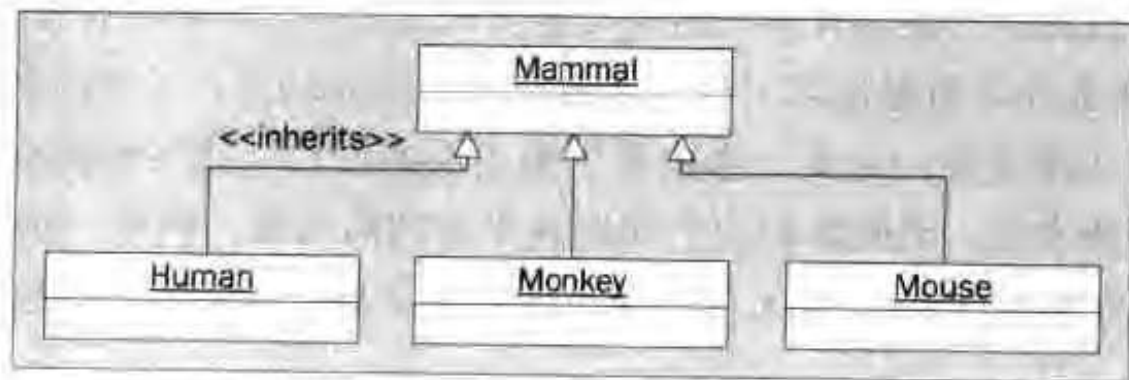


图 7-1

7.1.1 继承的层次结构

猴子和人都有不同的类型，它们的属性也互不相同，所以，可以把派生类当做基类，将继承层次结构扩展到其他类上。下面扩展这个继承层次结构，建立两种不同类型的 Monkey，一种是 Ape，另一种是 Chimpanzee。猿猴和黑猩猩应具有猴子的特性，例如 Climb() 方法，但没有人的特性。为这两种新类型的猴子提供不同的操作和属性，例如，猿猴要占领领土，所以有一个 Fight() 方法，而黑猩猩没有这个方法，但有一个 Flee() 方法。

下面修订过的模型显示了这个新派生类的关系，如图 7-2 所示，其中有第二层派生关系。

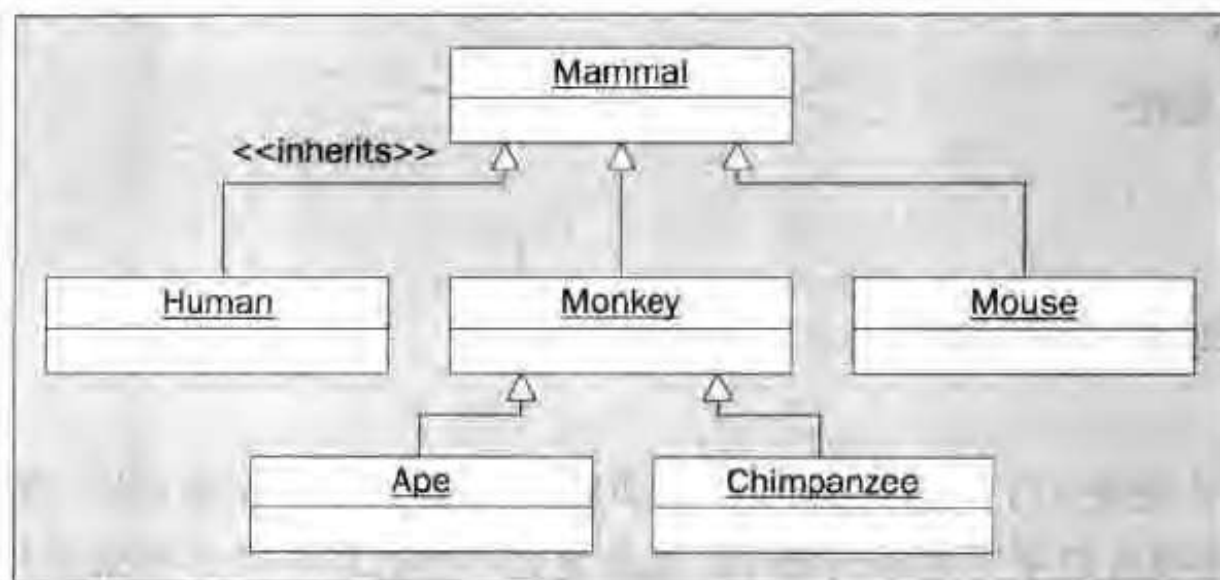


图 7-2

Ape 是 Mammal，也是 Monkey，它的实现代码与 Mammal 不同。注意这个模型仍缺乏一些区分每个类与基类的描述。在 UML 中可以使用类符号添加更多的信息，例如类的操作和属性。顶部的图形表示类名，第二部分包含类的属性名，最后一部分包含操作。见图 7-3。

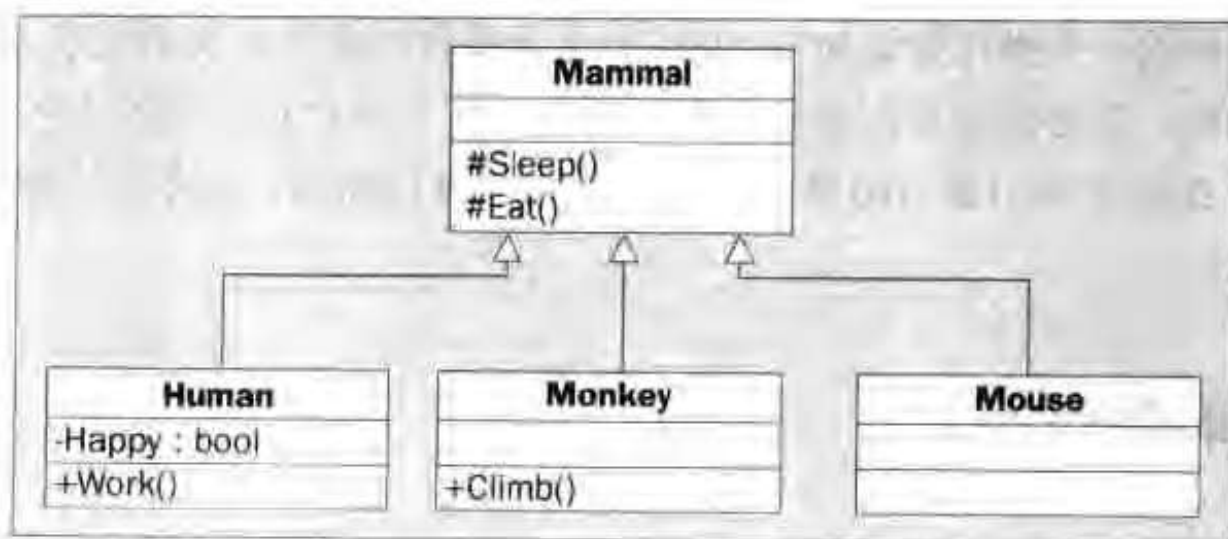


图 7-3

在 `Mammal` 类中, 成员前面的符号#表示 `Sleep()` 和 `Eat()` 是类的受保护成员, 即它们只能由类本身包含的代码和派生类来访问。符号-表示私有成员, 符号+表示公共成员。在图 7-3 中, 给 `Human` 类添加了一些成员: `Work()` 操作, 因为人是惟一能工作的哺乳动物, 一个属性表示较高级的人类情感, 它是一个布尔值, 表示高兴或悲伤。同样, `Monkey` 的 `Climb()` 方法表示猴子爬树, 并不是所有的哺乳动物都会爬树, 所以不能在基类 `Mammal` 中实现它。老鼠没有额外的操作或属性, 因为老鼠是一个典型的哺乳动物, 除了睡眠和吃食之外不需要其他活动, 但这里也为这个类提供了一个扩展, 即 `RunRoundWheel()` 方法。

不需要重复定义基类固有的操作和属性, 因为在定义类之间的继承关系时, 这在 UML 模型中是隐式的。在图 7-3 中, 假定人能完成三个操作:

- `Sleep()`
- `Eat()`
- `Work()`

还有一个属性:

- `Happy`

7.1.2 重写基类的操作

基类给派生类展示了一系列操作(方法和属性), 它们是可以重写的。把基类定义为 `virtual`, 派生类就可以重写基类的操作, 也就是说, 所有这些操作在派生类中都可以有新的实现。标记为非虚的方法不能重写, 因此不能向基类提供新的实现, 但可以用相同的签名定义一个操作, 隐藏相应的基类操作。在 `Mammal` 类层次结构中, `Sleep()` 操作应是虚拟的, 因为每个类都有不同的实现, 例如, 老鼠比人或猴子需要更多的睡眠。因此, 应根据这些类的区别和是否能毫不修改地继承基类的实现, 把不同的方法与每个派生类关联起来。在 UML 类图中, 要把基类中的一个操作定义为 `virtual`, 只需把与基类方法同名的方法添加到派生类图上。

如果基类中的一个操作标记为 `virtual`, 派生类就可以重写基类的实现。如果派生类选择实现该操作, 就必须使用与基类完全相同的签名。本章的后面将讨论 C# 中具体的做法。但是, 在派生类中重写的每个方法签名必须进行标记, 表示将在派生类中提供一个新的实现。

7.1.3 抽象类和接口

继承中的一个重要概念是具体的类。可以实例化的类都是具体的类。要拥有类的实例, 该实例在其接口的每个方法中必须有定义好的实现。满足这些条件的类就称为具

体类。如果只能定义具体类，这种区分就是不必要的，但在.NET中，还可以使用两种机制定义没有定义实现的类：接口和抽象类。

抽象类至少包含一个抽象方法。抽象方法是声明了签名，但没有提供任何实现的方法。在创建从抽象类中派生的类，并添加实现时，才提供抽象方法的实现代码。如果从抽象类中派生的类没有为抽象方法定义实现，派生的类也是抽象的。抽象类与具体类正好相反，它不能实例化，在生成类继承层次结构，提供更具体的继承模型时，使用抽象类。

接口就像是抽象类的一种纯粹形式。抽象类可以包含一些方法的定义，而接口只能包含签名的定义。因为接口不能定义实现，所以继承系统对它们特别对待，稍后讨论。

抽象类用作在需要时提供实现代码的基类，由所有的类继承。可以把Mammal声明为抽象类，创建Mammal类型的对象是没有意义的，更不用说创建Mammal的子类型的对象了。在继承模型中，最好把抽象类作为基类。

每个在抽象类中声明的抽象操作都隐式声明为virtual，因为必须在派生类中重写这个方法，或声明为abstract。对非抽象方法来说，这不是必须的，因为派生类可以在需要时使用抽象基类方法的实现。为了重写非抽象方法，应在基类中把它们声明为virtual。

抽象类和抽象方法在UML中使用斜体文本表示。Mammal模型只包含一个方法，如果基类有多个抽象和非抽象方法，可以组合使用斜体文本和非斜体文本来表示，见图7-4。

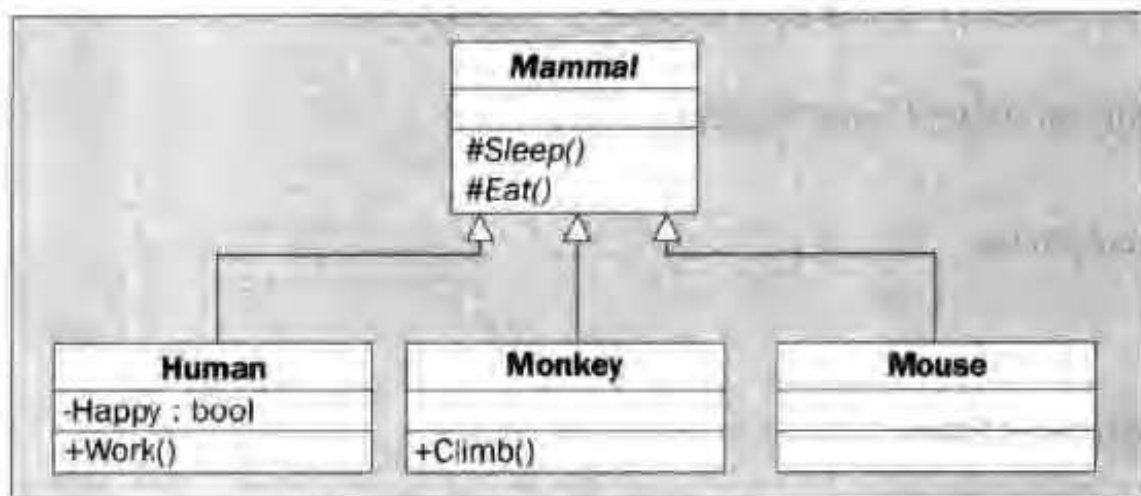


图 7-4

1. 继承的类型

继承有两种不同的方式，第一种方式是基类包含实现时的实现继承方式。如果操作声明为virtual，就可以重写该实现。如果操作声明为abstract，就必须重写该实现。这是前面介绍的继承类型。第二种类型是接口继承。实现继承需要接口继承，如果继承了一个方法的实现，也就通过所调用的实现继承了该方法的签名。但对于抽象方法，可以继承提供实现的责任，而不继承实现。这就是接口的继承。使用纯粹的接口类型，可以声明一组方法，类必须实现这组方法，才能继承接口。每个派生类都必须实现接口指定的操作。

图 7-5 中的 Mouse 类就实现了 ISmallAnimal 接口。

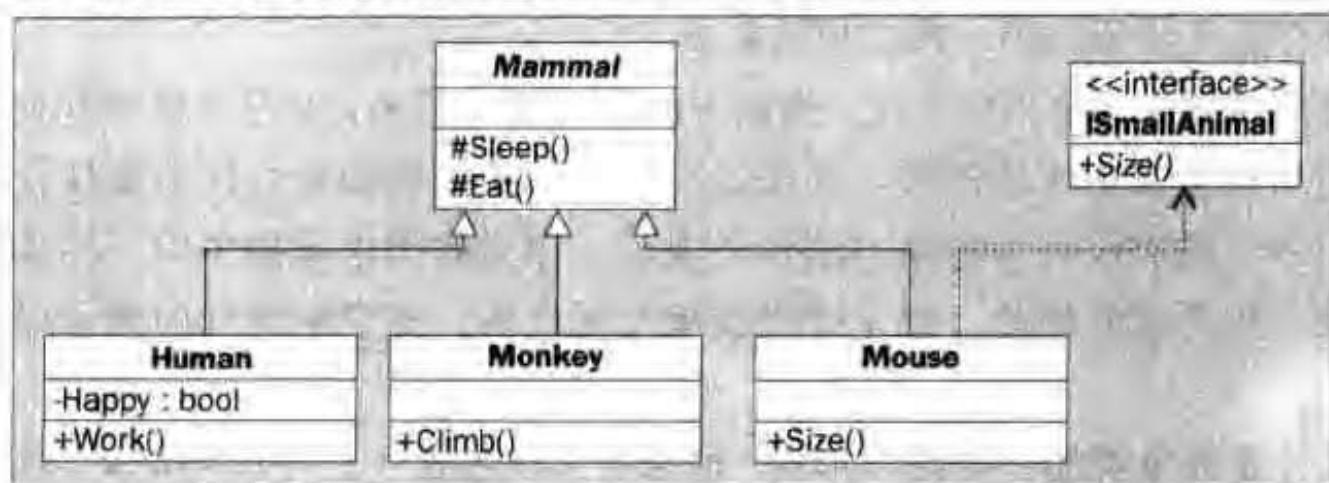


图 7-5

接口的 UML 图与类的 UML 图略有不同。在上面的例子中，使用<<interface>>指定 ISmallAnimal 是一个接口，而不是类。接口不能包含任何实现，只能包含操作签名。因此，接口不声明属性。可以使用虚线来表示接口和其他接口或类之间的关系。从图 7-5 中可以看出，Mouse 类提供了 Size()操作的实现。

接口提供了指定方法契约的方式，而不是方法契约的实现。使用接口继承可以提供类中接口方法的不同实现。Framework 类库中的集合类使用接口继承来定义许多方法的实现，例如 Sort()、Reverse()和 Add()。下面的例子使用了 ArrayList，在创建 ArrayList 时，实现了 ICollection，因此其类型是 ICollection。可以使用 Hashtable 类来代替 ArrayList，它们的方法没有什么区别，因为它们都实现了 ICollection：

```
public static int ReturnCount(ICollection col)
{
    return col.Count;
}

public static void Main()
{
    ArrayList ar = new ArrayList();
    int count = ReturnCount(ar);
}
```

2. 多重继承

Microsoft CLR 实现采用的 CLS 要求，一个类只能继承一个基类，因此不会有两个直接基类。.NET 避免了多重继承的复杂性，少了许多麻烦，在这样的环境下，要把两个类的功能合并为一个类，可以使用其他机制。在类的实例继承了多个基类(有多个 is a 关系)时，可以用多个接口继承来表示，这些内容稍后讨论。多重继承的方式非常类似于 Java 语言设计人员采用的方式，但与 C++大不相同。C++支持多重实现的继承，但

C#不支持。多重实现的继承只能使编码更容易一些,如图7-6所示。

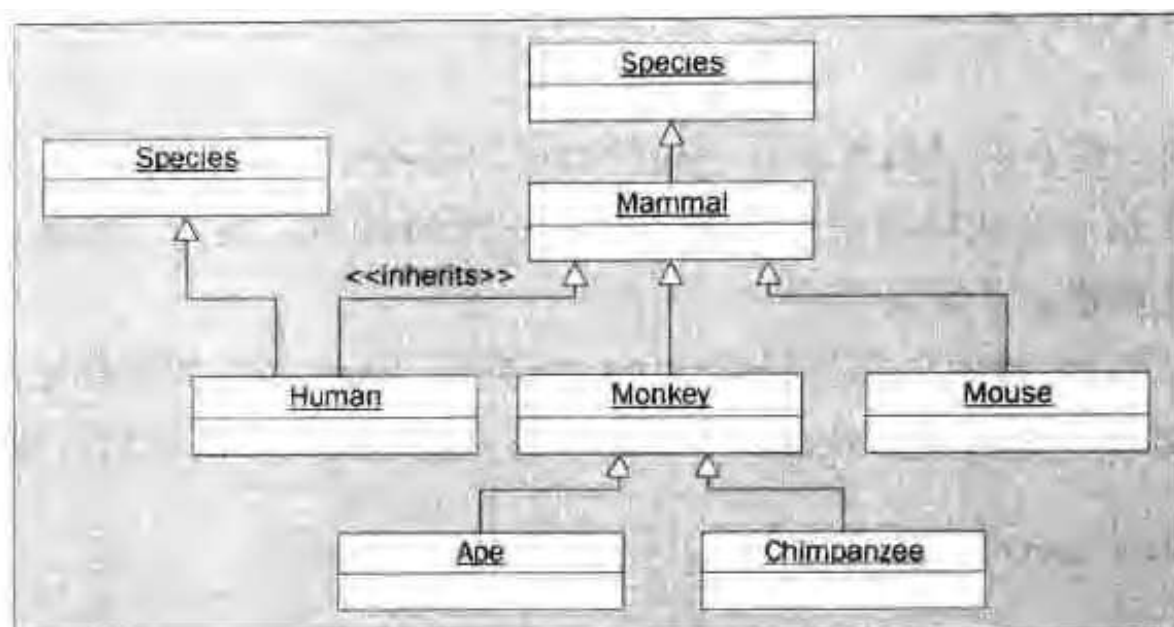


图 7-6

类 Human 继承了 Mammal 和 Species。Mammal 本身继承自 Species。在解析所有的链接时, Human 会继承两次 Species 的方法。但 Mammal 类不会重写 Species 中的一些方法,此时,如何解析对 Human 类的方法的调用? 这些在 CLR 中都避免了,因此就简单多了。

如果仅继承接口,而不继承实现,就不会出现相同的问题。继承接口就是继承提供实现的责任。这个责任可以继承几次,即可以从两个不同的接口继承相同责任,并对两个接口提供实现。所以,实现的多重继承比较复杂,但接口的多重继承则有助于解决许多问题。

下面的例子使用了 .NET Framework 类库中与集合类相关的两个技术。例如, ArrayList 类实现了 IList、ICollection、IEnumerable 和 ICloneable 接口,因此为枚举、克隆和标准集合类的方法,如 Add(), 提供了它自己的实现。Windows Forms 控件 DomainUpDown 使用的 DomainUpDownItemsCollection 派生于 ArrayList,强调在同一个层次结构中使用实现和接口的继承。可以在层次结构的同一层定义接口和基类,但下面的实现把它们显示在另一层上。见图 7-7。

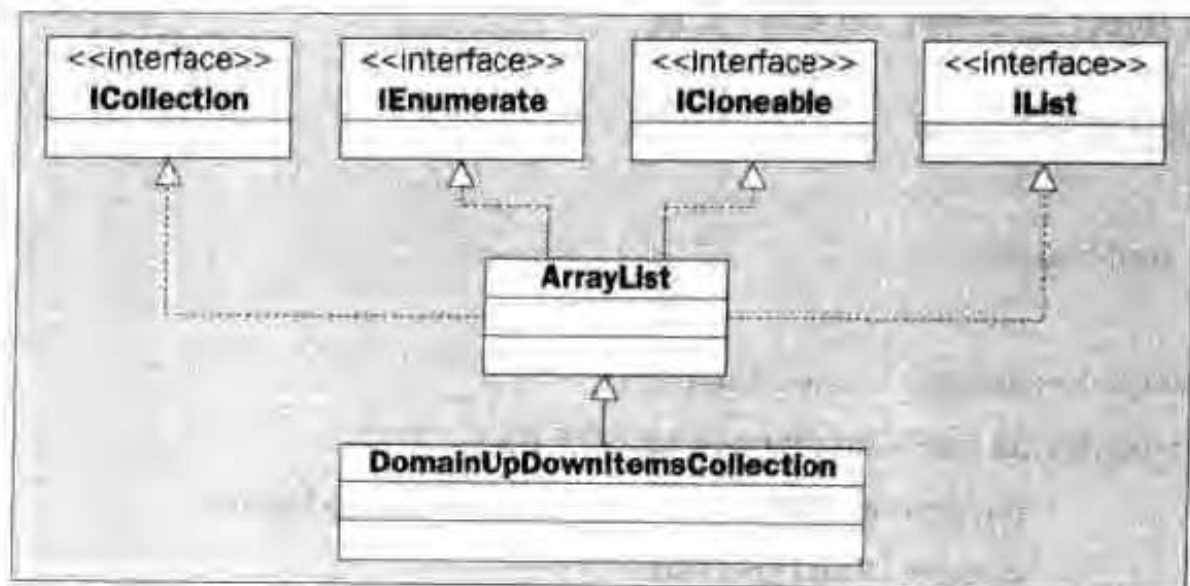


图 7-7

7.1.4 创建类层次结构

本节介绍的一些小例子创建了一个新的类层次结构，其中包括 UML 一节中介绍的许多概念。这些例子在代码中演示了继承和多态性的原则，创建了抽象类和具体类，还为多个接口继承建立了实现类。

为了演示，可以创建两个类类型和一个结构类型。Example1 类没有显式继承其他类，而 Example2 类继承了 System.Object 类。Example3 结构也没有显式继承其他类。

```
public class Example1
{
    //some code
}

public class Example2 : System.Object
{
    //some code
}

public struct Example3
{
    // some code
}
```

```
Example2 example2 = new Example2();
Console.WriteLine("Class type instance is {0}
                  and derives from {1}", example2.GetType().Name,
                  example2.GetType().BaseType);

Example3 example3 = new Example3();
Console.WriteLine("Class type instance is {0}
                  and derives from {1}", example3.GetType().Name,
                  example3.GetType().BaseType);
    }
}
```

上面的代码 `example_inheritance.cs` 执行后，会生成如下输出：

```
C:\Class Design\Ch 07> example_inheritance
Class type instance is Example1 and derives from System.Object
Class type instance is Example2 and derives from System.Object
Class type instance is Example3 and derives from System.ValueType
```

编译器不允许通过继承 `System.ValueType` 来创建自己的值类型。CLR 没有直接区分类和结构(即引用类型和值类型)，而使用继承方式来区分值类型的两个优化结果。CLR 可以优化处理值类型的方式，因为它们是密封的。密封类不能继承或派生。

1. 类的定义

通过 .NET Framework 工具 ILDASM 来查看 MSIL 输出，允许检查这种相互关系，确保三个类型分别继承自 `System.Object` 或 `System.ValueType`。注意，结构型值类型 `Example3` 在 IL 定义中标记为密封的，也就是说，不能把这个类型用作基类。在默认情况下，所有的值类型都包含这个定义，表示值类型不能使用任何形式的多态性，CLR 知道这些，就可以优化结构的内存分配的行为。对象实例需要一个方法表，并间接使用这个表，来支持实现的继承和接口继承，而不需要利用查询工具来确定应使用什么方法，即基类方法或重写的方法，因为运行时知道它只能有一个指定的值，而使用值类型可以避免系统开销，因为它是密封的。见图 7-8。

2. 设计基类

前面介绍了如何构建类层次结构，把 `Mammal` 用作基类，然后派生其他类，这些类继承了 `Mammal` 的方法，并为此类的新方法提供了定义。从一开始就计划构建类层次结构是非常重要的，因为基类和可见接口上的任何修改都会导致对继承该基类的所有类产生修改。



图 7-8

下面的 Teacher 类层次结构定义了两个字段值 Salary 和 Grade，以及一个方法 CalculateDaysWorked()。派生类 HeadTeacher、SupplyTeacher 和 StaffTeacher 分别是教务主任、有临时岗位的助教，以及一般教师。教务主任并不教课，而有临时岗位的助教和一般教师必须教课。见图 7-9。

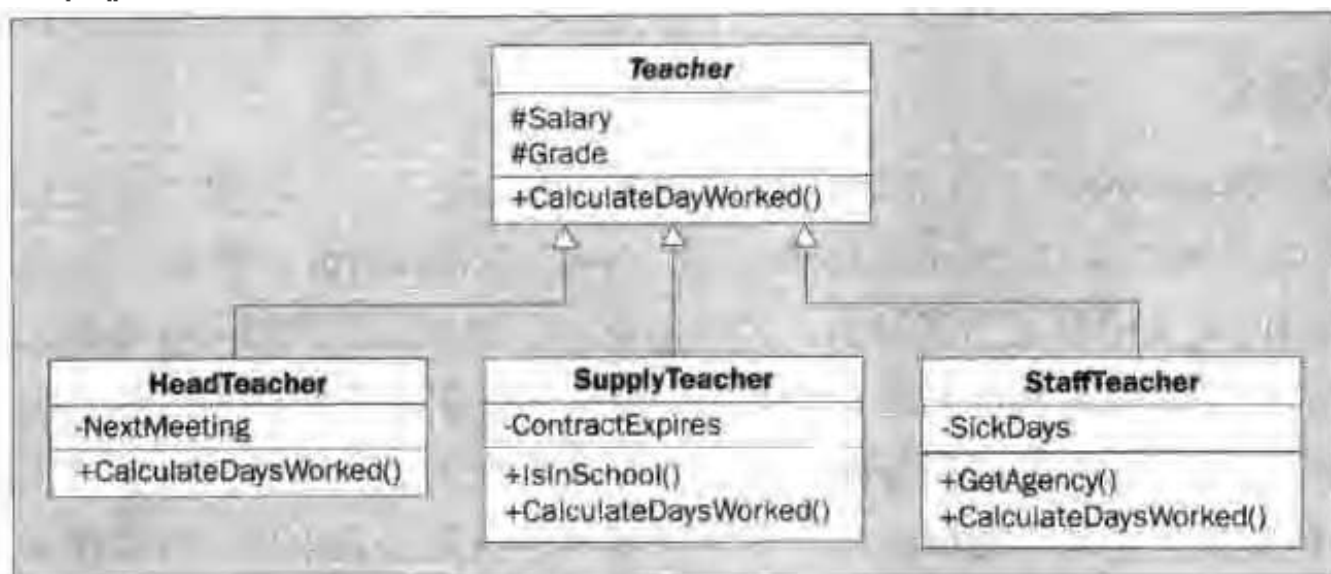


图 7-9

例如，基类 Teacher 中的方法 CalculateDaysWorked() 返回每个教师一年内的工作天数，该方法在三个派生自 Teacher 的具体类中有不同的实现。

假定要利用另一个参数和索引来计算这个值。修改基类中的方法签名会影响所有的派生类，因为派生类必须提供一个实现，以匹配基类的实现。如果基类中有许多派生类，这就是一个很艰巨的任务。使用 internal 修饰符可以避免这个问题，该修饰符表示，只有同一个程序集类能看到它们要派生的类。这种方式确保了程序集中的类可以继承，但其他软件开发人员不能从程序集外部的类中派生自己的类，这样，对新基类的可见修改

就不会破坏软件应用程序。

抽象类和具体类

把基类 `Teacher` 声明为 `abstract`，把所有的字段声明为 `protected`，而不是 `private`，因为它们对派生类是不可见的。派生类可以为字段创建属性访问器，提供一级封装，或在基类中把属性访问器声明为 `protected`，把底层的字段声明为 `private`。这样，如果基类的实现修改了，派生类不会受到影响。在默认情况下，方法不是抽象的。`IsWorking()` 方法可以在派生类上重写，因为它声明为 `virtual`，允许修改私有变量 `isWorking` 的值。

```
using System;
public abstract class Teacher
{
    protected int salary;
    protected int grade;
    private bool isWorking = false;
    public abstract int CalculateDaysWorked ();
    public virtual void Work(bool isWorking)
    {
        //we could add some logging code here
        this.isWorking = isWorking;
    }
    protected bool IsWorking
    {
        get { return isWorking; }
    }
}
```

同样，可以为这个类定义一个构造函数，但最好把它标记为 `protected`，只允许派生类访问它。这个类的另一个实现是展示两个属性访问器，给基类中的私有字段 `Salary` 和 `Grade` 写入信息。这将在基类和派生类之间建立松散耦合，因为所有的派生类都不需要直接依赖字段类型。提供受保护的属性访问器，表示可以修改基类中的实现，在字段声明中反映这种修改。也就是说，属性访问器总是可以有相同的方法签名，但实现代码可以修改，而不会给派生类带来任何开销。

可以为派生类提供一个属性访问器，来获取或设置私有字段 `int` 的值。属性访问器有一个 `long` 型参数，把它转换为 `int` 值(反之亦然)。

```
protected Teacher(int Grade)
{
    this.grade = Grade;
}
```

```
protected long Grade
{
    set
    {this.grade = (int)value;
    }
    get
    {return (long)grade;
    }
}
```

使用虚方法

可以添加另一个通用的虚(Virtual)方法，它可以在需要时重写。还可以添加一个方法 `CalculatePay()`，根据教师的年薪计算其月薪，包括加班费。`salary` 字段包含年薪。如果这个字段是 0，方法就抛出一个异常，否则就返回月薪和加班费。加班费存储在 `overtime` 字段中。

```
protected int overtime;
public virtual double CalculatePay()
{
    if(salary==0)
    {
        // throw new Exception("Unknown salary value");
    }
    return (salary / 12) + (overtime * (salary / (52 * 40)) * 1.5);
}

public class AbstractTeacherExample
{
    public static void Main(string[] args)
    {
    }
}
```

有时需要在派生类中重写这个方法，例如为 `SupplyTeacher` 类提供另一个实现，返回不同的值，因为该类根据等级计算每天的薪水。一般教师的薪水计算方式与虚方法相同，教务主任的计算方式也类似，但要根据他们的级别对加班费和年度补助加倍。

把这个文件保存为 `abstract_teacher.cs`。MSIL 显示了 `Teacher` 类和 `CalculateDaysWorked()` 方法的定义。在 MSIL 中，`Teacher` 类声明为 `abstract`，

CalculateDaysWorked()方法设置为 virtual。没有把它声明为 virtual，是因为所有的抽象方法都默认为虚方法。见图 7-10。

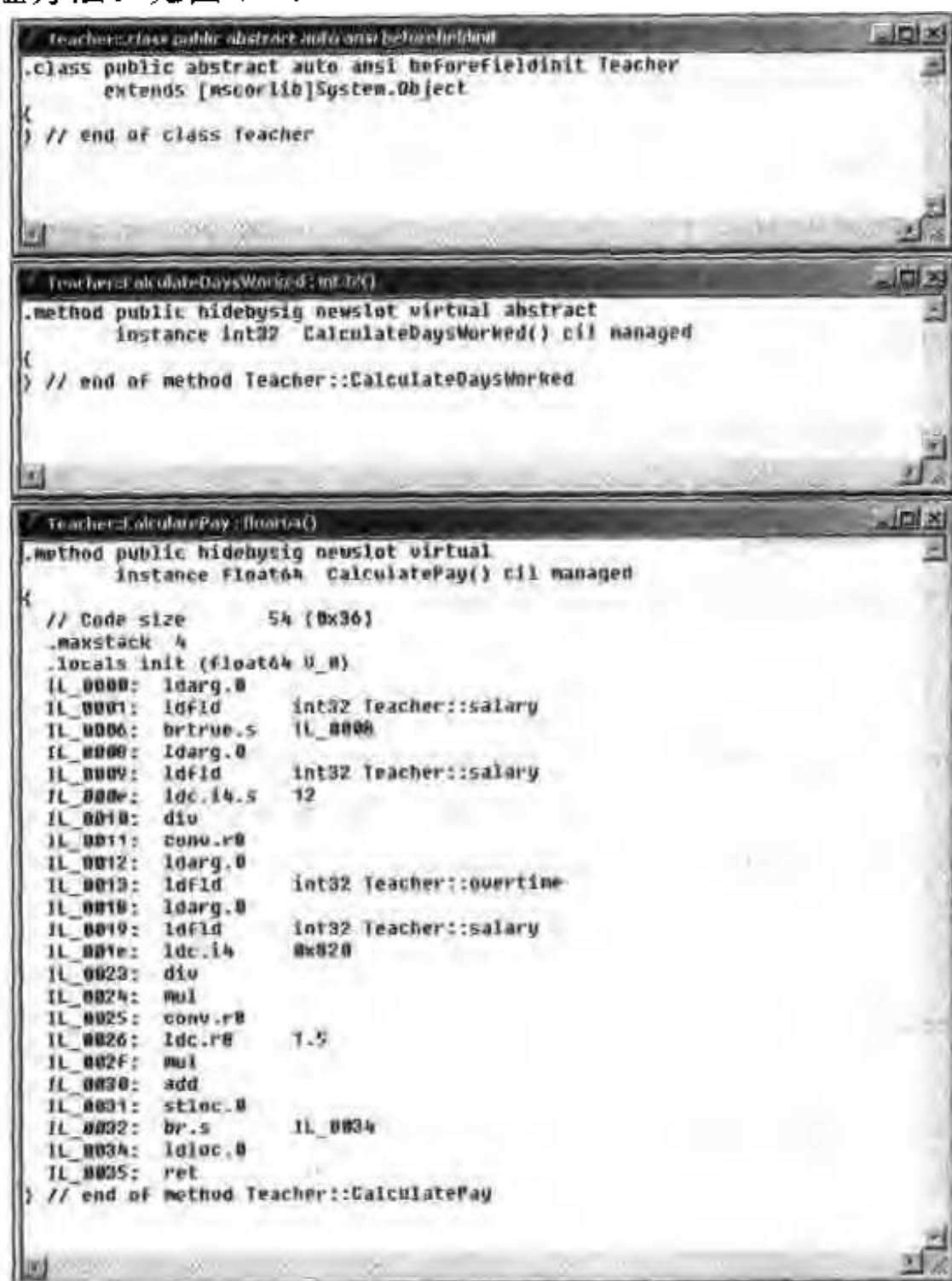


图 7-10

这里有两个新的关键字 hidebysig 和 newslot。关键字 hidebysig 与 virtual 一起使用，表示这个方法应隐藏在派生类中。如果存在签名相同的方法，在调用方法时就可以避免混淆。

newslot 关键字在虚拟函数表中创建了一个新的插槽，虚拟函数表是每个 .NET 对象都包含的一个方法表，其中包含所有继承的方法、重写的方法，以及静态方法。CLR 利用虚拟函数表机制来确定是调用虚方法，还是调用派生类上的重写方法。

密封类

在 .NET Framework 中，另一个重要的概念是使用密封类或方法，避免类的继承或方法的重写。抽象类可确保类用作基类，密封类则防止类用作基类。教务主任由 HeadTeacher 类定义，且只有一种类型，为此，需要创建 HeadTeacher 的定义，把该类密

封起来，防止它用作其他类的基类。如果用下面的密封类定义编译 `HeadTeacher` 类代码，再选择从该类中派生其他类，C#编译器就会生成一个异常。

```
public sealed class HeadTeacher
```

同样，声明 `sealed` 方法将防止在派生类中重写它。MSDN 文档指定密封类也可以用于非继承的类。密封的方法不能先在基类中声明，而必须在派生类中重写，并声明为 `sealed`。继承这个派生类的其他类不能重写 `Sealed` 方法。下面的代码保存在 `sealing_classes.cs` 中：

```
public class SupplyTeacher : Teacher
{
    private int monthDays;
    public override sealed double CalculatePay()
    {
        if(monthDays==0)
            monthDays = GetMonthDays(DateTime.Now.Month,
                                      DateTime.Now.Year);
        return (100 + (this.Grade * 10)) * monthDays;
    }

    protected int MonthDays
    {
        set { monthDays = value; }
    }

    private int GetMonthDays(int month, int year)
    {
        if(month==9||month==4||month==6||month==11)
        {
            return 30;
        }
        if(month==2&&(year%4==0))
        {
            return 29;
        }
        else if(month==2&&(year%4!=0))
        {
            return 28;
        }
        return 31;
    }
}
```

```

    }
}

```

这段代码探讨了基类的一些责任，以及如何使用基类影响派生类中的行为。

在图 7-11 的前两行代码中，注意在 IL 指令中，`abstract` 是 `final` 的对立指令(在 C# 中，也是 `sealed` 的对立指令)，因为前者确保方法必须在派生类中重写，后者禁止方法在派生类中重写。

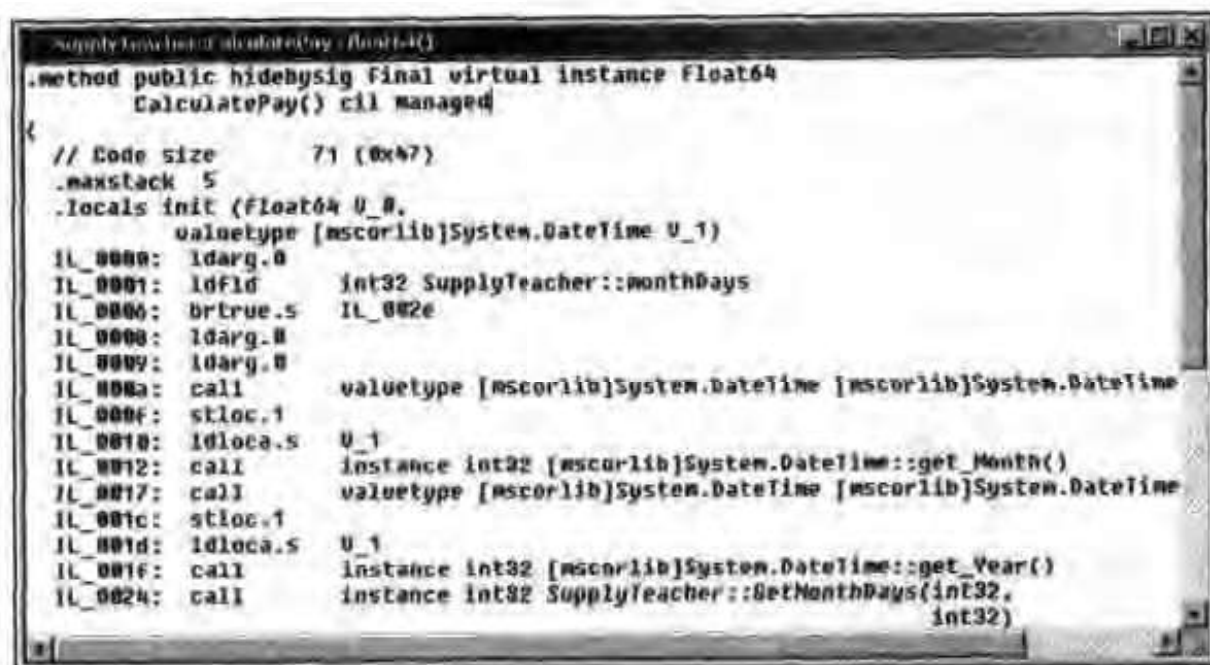


图 7-11

3. 派生类

回到 `Teacher` 模型上。下面要描述内置于这个模型中的三个派生类。在创建派生类的实例时，会调用基类构造函数。这里要注意优先级的顺序，因为基类构造函数会在派生类的构造函数之前调用。这个自顶而下的结构表示，`System.Object` 构造函数会先调用，之后调用中间的派生类的构造函数，最后调用最下面的派生类的构造函数。

`SupplyTeacher` 类把 `int` 型参数传送给它的构造函数，再调用基类(`Teacher`)中签名相同的构造函数。这会给 `grade` 字段填充一个值来替代初始值 0。接着计算 `salary` 字段的值，调用重写的方法 `CalculatePay()` 来填充 `salary` 字段，和计算年薪。注意，如果把 `CalculatePay()` 方法调用放在 `Teacher` 基类的代码块中，即使 `Teacher` 类有自己的方法 `CalculatePay()`，也会调用 `SupplyTeacher` 类的方法 `CalculatePay()`。这是因为我们创建的类型是派生类类型，而不是基类类型，因此运行时知道这个对象的实例化环境，也知道方法已在虚拟函数表中重写了。

```

public SupplyTeacher(int number): base(number)
{
    salary = CalculatePay() * 12;
}
public abstract class Teacher

```



```
{
    protected Teacher(int Grade)
    {
        this.grade = Grade;
    }
    //all the other methods, properties and fields here
}
```

即使没有初始化派生类，使用初始化代码仍可以访问基类的方法和属性。如果回过头来看看 `SupplyTeacher` 类，可以看到，继承的语法冒号(:)指定了 `SupplyTeacher` 继承自 `Teacher`。C++开发人员很熟悉这个语法，而 Java 开发人员可以把它与 `extends` 关键字关联起来，使用 `super` 来代替 `base`，以引用基类。下面的代码在 `deriving_classes.cs` 中显示：

```
public class SupplyTeacher : Teacher
{
    public SupplyTeacher(int number) : base(number)
    {
        Console.WriteLine(base.Salary);
        salary = CalculatePay() * 12;
    }
    //other methods and properties and fields here
}
```

方法的重写和遮蔽

在派生类中指定方法时，签名和方法名一样重要。下面在派生类 `StaffTeacher` 中创建另一个方法 `CalculatePay()`，用于给教师的月薪中加上每月的补助，但 `CalculatePay()` 方法不包含参数。

在基类上定义虚方法的标准模型说明，需要创建 `CalculatePay()` 的一个重载方法，并希望派生类 `StaffTeacher` 重写它。但是，这个方法重载对其他的所有类都是可见的，无论它们是否在派生类上重写了，都是如此。理想情况下，除了 `StaffTeacher` 之外的其他派生类都不应实现这个重载方法。而使用另一个重载的虚方法又不可能实现，此时应创建另一个方法，它的名称和返回类型与派生类 `StaffTeacher` 相同，但参数签名不同。这个方法带有两个重载，其中一个为重写的方法，而另一个包含未在基类中定义的全新签名。

这个技术可以在其他派生类上使用，创建签名不同的重载方法，从而拥有这些方法的重载版本。这个技术称为遮蔽。在下面的例子 `overriding_and_shadowing.cs` 中，允许用或不用 `new` 关键字来声明所有额外的重载方法，因为此方法没有在基类中隐藏惟一

的签名方法。

```
public class StaffTeacher : Teacher
{
    public StaffTeacher(int code) : base(code)
    { }
    public override double CalculatePay()
    {
        return base.CalculatePay();
    }

    public double CalculatePay(int monthlySupplement)
    {
        if(monthlySupplement > 1000) throw new Exception("Supplement out
                                                    of range exception");
        return base.CalculatePay() + monthlySupplement;
    }
}
```

如果省略了 `CalculatePay()` 方法，请求派生类匹配这个签名的所有方法都将由签名相同的虚拟基类方法提供。方法本身可以有自己的实现，包括对基类方法的调用，也就是调用方法的基类虚拟实现。

虚方法的分派

虚方法的分派包含了多态性的概念。如果有对派生类类型实例的基类引用，就可以使用基类引用调用派生类的任意重写方法。这允许在代码中把派生类引用替换为基类引用，从而忽略派生的类型。

即使通过创建 `SupplyTeacher` 类的一个新实例，来返回 `Teacher` 类的引用，代码中会反映 `Teacher` 类的接口(所有的公共方法、属性和字段)。它使用多态性来实现代码。

下面的机制称为虚方法分派，有助于使用 `is` 或 `as` 运算符检查某个类型，确定变量是哪一个类的实例：

```
Teacher st = new SupplyTeacher();
Console.WriteLine(st.CalculatePay());
//this will invoke the method on the derived class SupplyTeacher
```

非虚方法的分派

可以使用非虚方法分派来避免前面使用的多态性。如果创建一个类似于上述虚方法分派例子的派生类实例，就不能调用未重写基类方法的派生类方法，而只能调用基类

方法。

在编译没有 `override` 关键字的代码时，编译器会生成一个警告消息。警告消息说明派生方法的签名与基类方法相同，因此隐藏它。在派生类中重新定义这个方法，添加 `new` 关键字，就可以去除该警告消息。在 `Teacher` 类中，可以声明 `GetTaughtSubjectName()` 方法。

```
protected string subject;
public virtual string GetTaughtSubjectName()
{
    return subject;
}
```

在 `StaffTeacher` 类中，可以用相同的公共签名声明同一个方法：

```
public string GetTaughtSubjectName()
{
    return this.subject;
}
```

使用下面的代码(`nonvirtual_teacher.cs`)为派生类实例返回基类引用，使用该引用调用方法，就会调用基类方法。

```
Teacher st = new SupplyTeacher();
Console.WriteLine(st.GetTaughtSubjectName());
//this will invoke the method on the base class Teacher
```

4. 多态性

多态性的另一个明确实现是把 `Teacher` 引用当做方法的一个参数使用，这样就不必确定使用了哪个派生类，并可以通过替换原则把它用作基类。在方法 `InvokeMemberOnTeacher()` 中，传送一个 `Teacher` 类型的对象引用，它可以引用继承自 `Teacher` 的任何派生类型的对象。例如，派生类 `SupplyTeacher` 就传送给重写方法 `CalculatePay()`，返回一个字符串参数，其中包含要传送的类型的名称，因此类实例实际上是派生类型，它从这个 `GetType` 调用中返回。避免联合使用 `is` 和 `as` 运算符和多态性，因为这违反了多态性的原则，而对于每个添加到类层次结构中的派生类，都必须更新这个方法。

```
public class InvocationClass
{
    public static int InvokeMemberOnTeacher(Teacher teach, out string WhichTeacher)
    {
```

```
        int pay = teach.CalculatePay();
        WhichTeacher = teach.GetType().Name;
        return pay;
    }
    static void Main()
    {
        Teacher st = new SupplyTeacher(6);
        string teacherString = String.Empty;
        Console.WriteLine("Teacher Pay: {0}",
                           InvocationClass.InvokeMemberOnTeacher
                                   (st, out teacherString));
    }
}
```

Main()方法使用 Teacher 基类引用创建了一个 SupplyTeacher 类实例。它调用静态方法 InvokeMemberOnTeacher(), 在控制台窗口上显示输出。调用的结果如下所示:

```
C:\Class Design\Ch 07>polymorphism
Teacher Pay: 4960
```

为 CalculatePay()调用的结果生成的 MSIL 代码包含一个指令 callvirt, 它仅在调用虚方法代替正常的调用指令时才使用。与 callvirt 相反, call 使用当前类型引用的虚拟函数表, 并可以用于前面的非虚方法分派例子。callvirt 获取具体类型的信息, 确定是调用类型的基类上的虚方法, 还是调用类型的重写方法。注意, Teacher 类层次结构在 Inheritance 命名空间中。

虚拟函数表包含对类实例拥有的各种方法的内存指针列表。它为虚方法和非虚方法提供不同的区域, 以区分它们。派生类的方法表和基类的方法表在结构上是类似的, 虚方法声明中的方法对应于这两个方法表。基类和派生类都使用一个虚拟函数表来包含一系列指针, 指向每个方法在内存中的位置。每个虚方法都添加了 MSIL 指令 newslot, 这表示给虚拟函数表添加另一个插槽。这种添加过程仅存在于虚方法部分。

newslot 指令在虚拟函数表的最后添加一个新项, 与最后一个方法指针偏移一定的距离。在派生类中重写基类的虚方法不会添加 newslot 指令, 而是让 CLR 检查该方法签名是否与基类和派生类中的相同。如果不相同, 就添加到虚拟函数表的最后, 因为省略 newslot 指令仅用于提示 CLR 查看基类的虚拟函数表。

如果签名相同, 就在虚拟函数表中替换原来的方法, 为派生类的重写方法添加方法指针。下面的 MSIL 代码显示了 StaffTeacher 的方法 CalculatePay(), 它省略了 newslot 指令, 因此会替换基类的虚方法。见图 7-12。

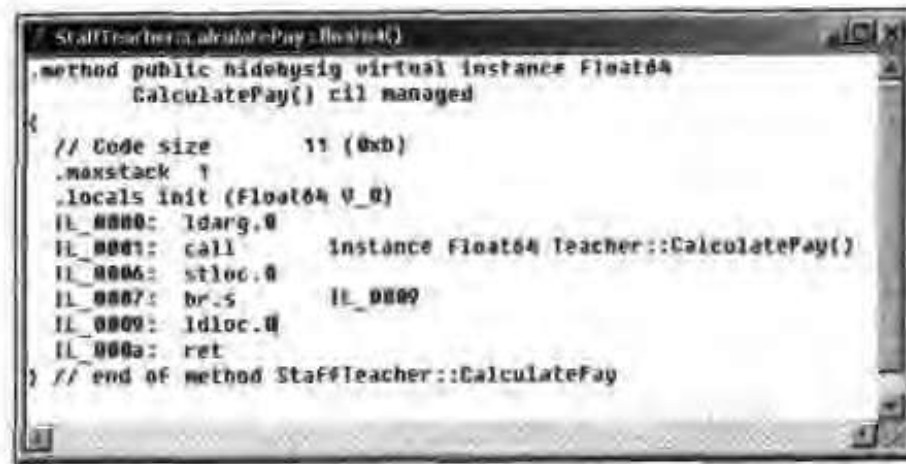


图 7-12

图 7-13 显示了 StaffTeacher 类实例的虚拟函数表布局。在对象的上半部分包含所有的虚方法，在下半部分包含所有的非虚方法，其后是所有的实例方法和静态方法。见图 7-13。

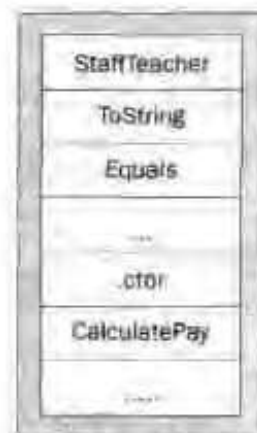


图 7-13

图 7-14 更完整地描述了图 7-13，其中包括支持对继承的虚拟对象和其他虚拟对象的区分。这是必要的，因为重写的方法会替代继承的虚方法中的对应方法。

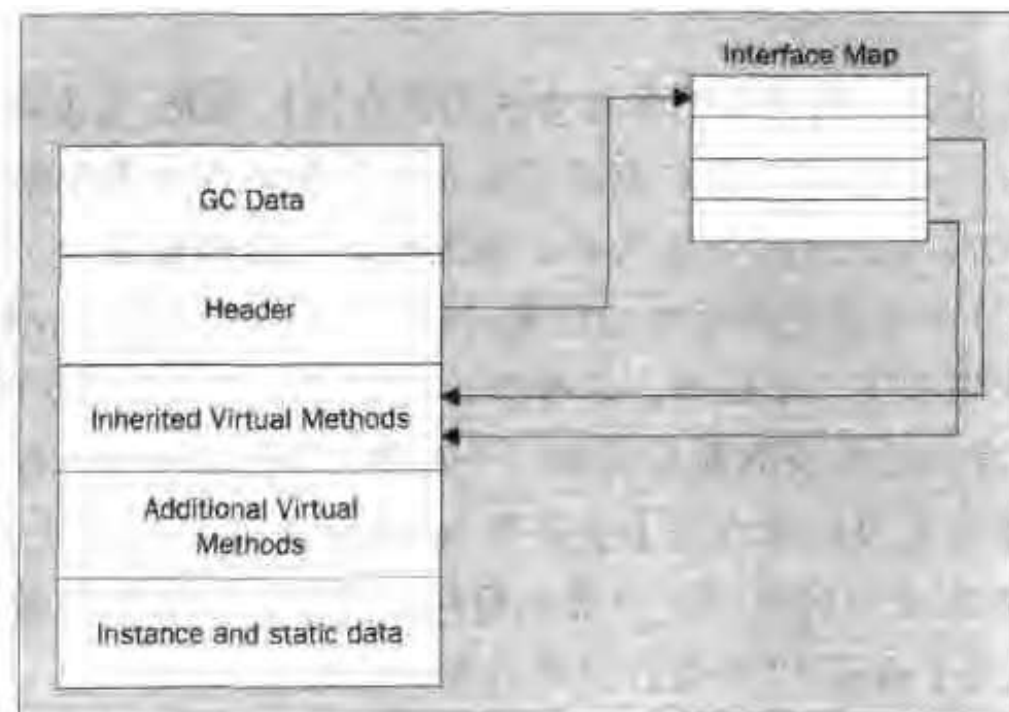


图 7-14

垃圾回收器 GC 的数据部分包含用于清理的信息，包括对象是否标记为删除，生成信息等。Header 包含一个指向内存接口映射的指针，它包含支持接口的一组指针，以支持已实现方法的方法表。这样，任何被支持的接口都可以在实现中找到已实现的方法。

最后, 在 MSIL 中查看一下 StaffTeacher 类的类声明, 可以看到, extends 关键字用于表示继承(在语义上类似于 Java)。如果类不是派生于另一个类, 则 MSIL 就默认它派生于 System.Object。所以, 即使类说明了 Teacher 类是基类, 最终的基类也总是 System.Object。见图 7-15。

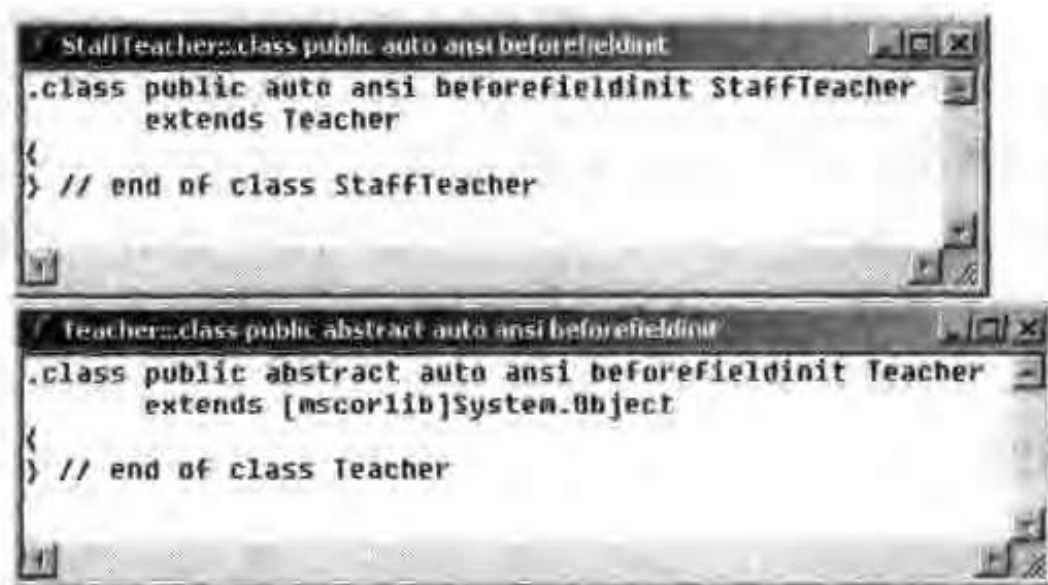


图 7-15

7.2 基于接口的编程

下面介绍如何在 C# 中编写接口, 让类能实现不同接口的成员。本章前面说过, 实现的继承和接口继承可以合并。声明接口有一种可以接受的格式。在扩展 SupplyTeacher 类的实现时, 可以定义两个接口, 其中包含 SupplyTeacher 类必须实现的方法, 因为接口声明只包含方法签名, 而不包含实现。

SupplyTeacher 类继承自 Teacher, 还实现了 ITemporaryWorker 和 IPublicWorker 接口。接口名应总是以 I 开头。这表示, SupplyTeacher 类派生于 Teacher 类, 所以是 Teacher 的一种类型, 但其特征与 StaffTeacher 不同。代码中的顺序常常先是继承的基类, 其后是要实现的接口列表。

```
public class SupplyTeacher : Teacher, ITemporaryWorker, IPublicWorker
```

接口可以帮助完成开发任务, 因为类实现的接口数量是没有限制的(这与实现的继承不同)。利用接口继承, 可以解除具体类与调用代码的耦合, 使代码中的类没有依赖关系。如果接口发生了变化, 类的实现则也需要修改, 以修订具体类中新的实现。一个结构可以实现一个接口, 但不能继承类的实现。

SupplyTeacher 类支持两个接口 ITemporaryWorker 和 IPublicWorker。实现了这些接口, 就可以利用替换原则以相同的方式处理类。这是非常重要的, 因为这个类可以看作是一种 ITemporaryWorker 和 IPublicWorker。也就是说, SupplyTeacher 类必须实

现每个接口指定的所有方法，也可以把它看做是这些接口，而不需要确定类的类型。该类必须实现每个接口定义的所有方法，因为这是类和调用代码之间的契约，否则编译器就会产生异常。

在 OO 开发中，使用多重继承的正确方式是继承多个接口，这是 .NET 和 Java 都支持的编程模型。接口可以用于补充实现继承的使用。可以在 `Teacher` 类上实现一个接口，因为 `Teacher` 是一种 `PublicSectorWorker` 的类型，与其他公共部门的人员共享某些相同的内容。在默认情况下，所有的派生类都继承了所有公共部门人员应共享的公共接口，因为每个派生类也都可以看作是一种 `PublicSectorWorker`。下面的例子扩展了 `Teacher` 类模型，允许实现接口，也就允许在类上使用多态性。

7.2.1 定义接口

为所有公共部门人员创建一个基类并不好，因为这违反了 `is a` 规则。使用接口可以确保所有类型的公共部门人员类都实现 `Load()` 方法。可以定义一个接口 `INonAdminPublicSectorEmployee`，在它的定义中包含 `Load()` 方法。也就是说，实现该接口的所有类都必须实现同一个签名方法。

下面定义的接口支持两个方法，第一个方法 `Load()` 带一个字符串参数，第二个方法 `GetItem()` 带一个 `DateTime` 参数，返回一个字符串。注意定义中没有实现。接口不能包含任何实现代码，只能包含方法、事件和属性的签名(也不能包含静态成员或实例成员)。

每个类都必须以某种方式实现这里定义的两个方法，否则 C# 编译器就会生成异常(除非方法是抽象的)。

```
public interface INonAdminPublicSectorWorker
{
    void Load(string fileName);
    string GetItem(DateTime date);
}
```

7.2.2 实现接口

可以在当前类层次结构中为教师添加一个 `HospitalWorker` 类，它拥有 `INonAdminPublicSectorWorker` 接口方法的定制实现。`HospitalWorker()` 类的 `Load()` 方法的一个简单实现如下所示，它使用一个用逗号分隔的文件，其中包含了某一天与约会项相关的所有日期信息。在这个例子中，要注册的方法名前面没有前缀来说明这是接口签名的实现，因为编译器会自动辨识所实现的方法签名是否与接口中指定的方法签名相同。

运行时使用接口映射来确定类支持的接口，并检查是否有所支持的方法签名。方法表可以检查类所支持的接口的接口映射，以确定该类方法在虚拟函数表中的位置，这些方法会根据接口按顺序存储。因此使用接口会使间接性提高一级。

Load()方法使用日期和时间值创建一个 DateTime 对象，被用作 Hashtable 的一个键，以存储与键相关的所有字符串值。第二个方法 GetItem()也实现了，它提取一个 DateTime 值，用作 Hashtable 的一个键，返回相应的值；或者如果集合中 DateTime()不存在，就返回空字符串。注意：其他实现类中的实现可以采用任何形式。下面的代码文件保存为 implementing_interfaces.cs：

```
public class HospitalWorker : INonAdminPublicSectorWorker
{
    private Hashtable ht = new Hashtable();
    public void Load(string FileName)
    {
        try
        {
            FileStream stream = File.Open(FileName, FileMode.Open,
                                           FileAccess.Read);
            StreamReader reader = new StreamReader(stream);
            string fileContents = reader.ReadToEnd();
            foreach(string line in
                        fileContents.Split(Environment.NewLine.ToCharArray()))
            {
                string[] agenda = line.Split(new Char[] { ',' });
                ht.Add(DateTime.Parse(agenda[0]), agenda[1]);
            }
        }
        catch {}
        finally
        {
            if(reader!=null)
                {reader.Close();}
        }
    }
    static void Main(string[] args)
    {
        HospitalWorker hw = new HospitalWorker();
        hw.Load("C:\\agenda.txt");
    }
}
```



```
public string GetItem(DateTime dt)
{
    return ht[dt].ToString ();
}
}
```

7.2.3 多态性和接口

不需要知道实现类是 `HospitalWorker`、`Teacher`，还是其他实现 `INonAdminPublicSectorWorker` 接口的类。可以修改类的创建语句和调用。然后只需要知道接口是 `INonAdminPublicSectorWorker` 类型，就可以通过多态性实现这个例子中的同一个抽象级别，因为这些是在具体的基类实现中完成的，即使在代码中使用了比较一般的 `Teacher` 类型，CLR 也会自动调用派生类的正确方法。

```
INonAdminPublicSectorWorker hw = new HospitalWorker();
hw.Load("C:\\agenda.txt");
```

使用接口和不同类型的类，可以了解每个类提供的功能子集；如果使用 `INonAdminPublicSectorWorker` 接口来替代上述代码例子中的类，就只能看到接口支持的方法，而看不到类支持的方法。这是类和把调用代码与具体类绑定在一起的契约之间的公共签名。

接口不包含任何实现代码，所以不能像抽象类那样实例化，但可以把接口看做相关类的组合机制。这就是多态性和接口一起使用的原因。例如，可以使用下面的代码把 `HospitalWorker` 对象转换成它实现的接口类型。在类实现了多个接口时，它就可以转换为这些接口的任何一个接口。在 `inaps` 引用上调用 `Load()` 方法，就会在 `HospitalWorker` 类实例上调用需要的方法，如上所述。

```
HospitalWorker hw = new HospitalWorker();
INonAdminPublicSectorWorker inaps = (INonAdminPublicSectorWorker)hw;
inaps.Load("C:\\agenda.txt");
```

使用引用类型代替类类型时要小心，因为不能调用接口不支持的任何对象；这种行为与使用基类类型在派生类实例上调用方法是相同的。例如，使用基于接口的编程方式就是不合法的，因为 `INonAdminPublicSectorWorker` 不支持 `ReturnEverything()` 方法，该方法由 `HospitalWorker` 类支持：

```
inaps.ReturnEverything();
```

解决这个问题的方法是不再转换回 HospitalWorker 类型，ReturnEverything()方法为它定义了另一个接口：

```
if(inaps is HospitalWorker) {  
    HopsitalWorker hw = (HospitalWorker)inaps; }
```

下面扩展模型，定义一个方法，它使用接口类型来代替具体类型，并调用该接口支持的方法。其优点是可以在代码中使用多态性提供深层次的抽象。编写一个静态方法，使用在接口类型中传送的基类-派生类模型，而不是具体类和传送给 Load()方法的议程文件名。只要具体的实现类支持正确的接口，使用什么类就不相关了。

```
public static void InvokeMemberOnNonAdmin(INonAdminPublicSectorWorker inon, string  
fileName)  
{  
    try  
    {  
        inon.Load(fileName);  
    }  
    catch(Exception ex)  
    {  
        //log this exception to a file  
    }  
}
```

为调用这个方法，应使用与前面例子类似的代码(但在代码中应尽量避免使用这么长的方法名)：

```
INonAdminPublicSectorWorker hw = new HospitalWorker();  
InvocationClass.InvokeMemberOnNonAdmin( hw, "C:\\agenda.txt");
```

相同类可以实现许多接口，同一个接口也可以由许多不同的类实现。一个类实现多个接口的概念通常称为多重接口继承。实现代码的继承一般是指从基类中继承，接口继承则是指实现接口。

在自己的类库中可以混合和匹配接口，这与.NET Framework 类似。例如，可以在.NET Framework 中开发实现公共接口的类，如 IList、ICollection、ICloneable、IFormattable 等。

在前面的一个例子中，已经实现了支持 IComparer 接口的类。这个类可以传送给集合类对象的 Sort()方法，多次调用已实现的 Compare()方法，比较集合中的所有成员。这将允许使用.NET Framework 集合类，而不必建立自己的集合类，指定要使用的排序

类。多重接口继承是可行的，因为利用多态性和许多支持的接口，同一个类可以以许多不同的方式使用。

7.2.4 多重接口继承

下面的代码定义了 `ILazyWorker` 接口，它支持一个方法 `WastedTime()`，其现在在实现类的下面。接口可以声明为公共类型或内部类型，默认为公共类型。

```
interface ILazyWorker
{
    int WastedTime();
}
```

可以修改 `HospitalWorker` 类的定义，以支持 `INonAdminPublicSectorWorker` 接口和新的接口 `ILazyWorker`。对象支持的接口数并没有限制，但大多数接口只支持几个方法。

```
public class HospitalWorker : INonAdminPublicSectorWorker, ILazyWorker
```

`HospitalWorker` 支持上面的两个接口，这表示在创建类实例的代码和创建类本身的代码之间有一个契约。对象创建代码知道对象应支持什么接口，因此可以根据接口的定义推断对象可以做什么，而不必了解对象的类型是什么。这就是多态性通过实现类的契约责任发挥作用的方式。见图 7-16。

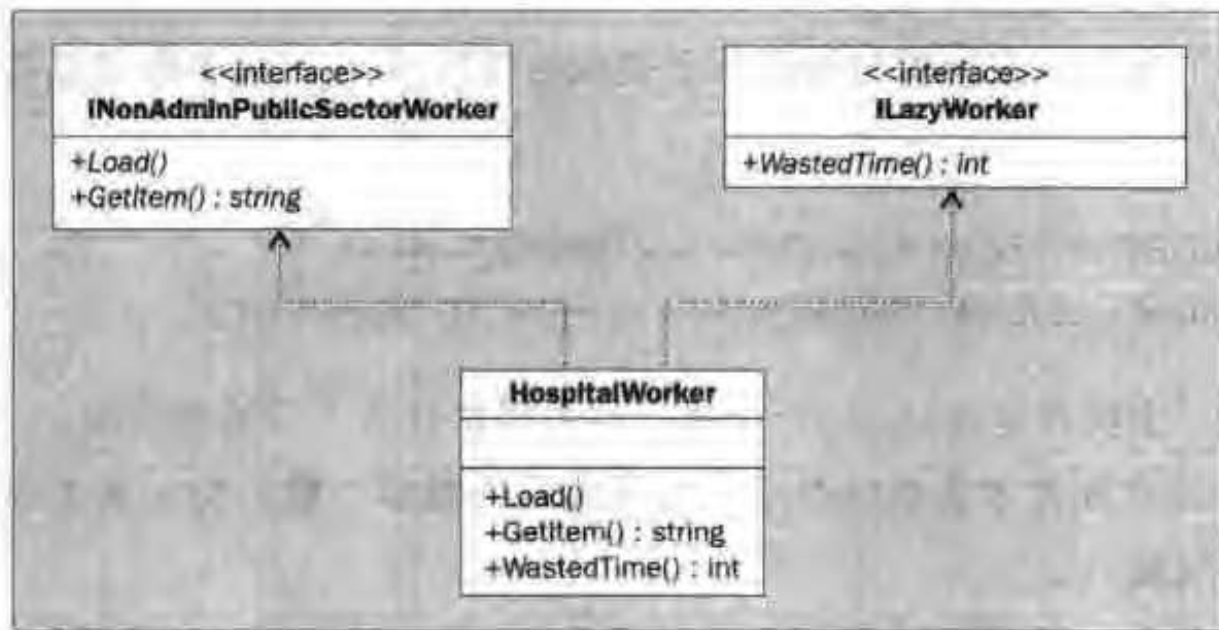
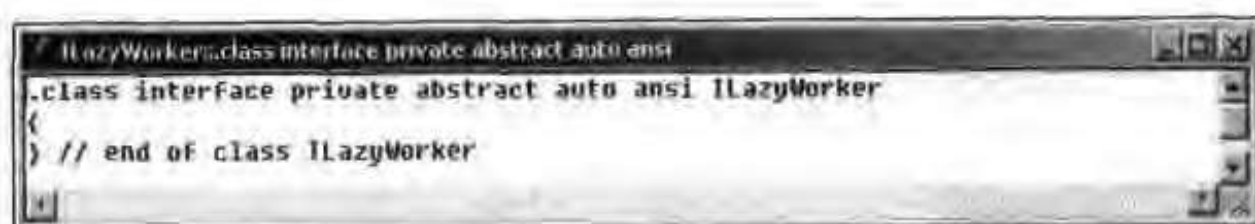


图 7-16

7.2.5 IL 中的接口

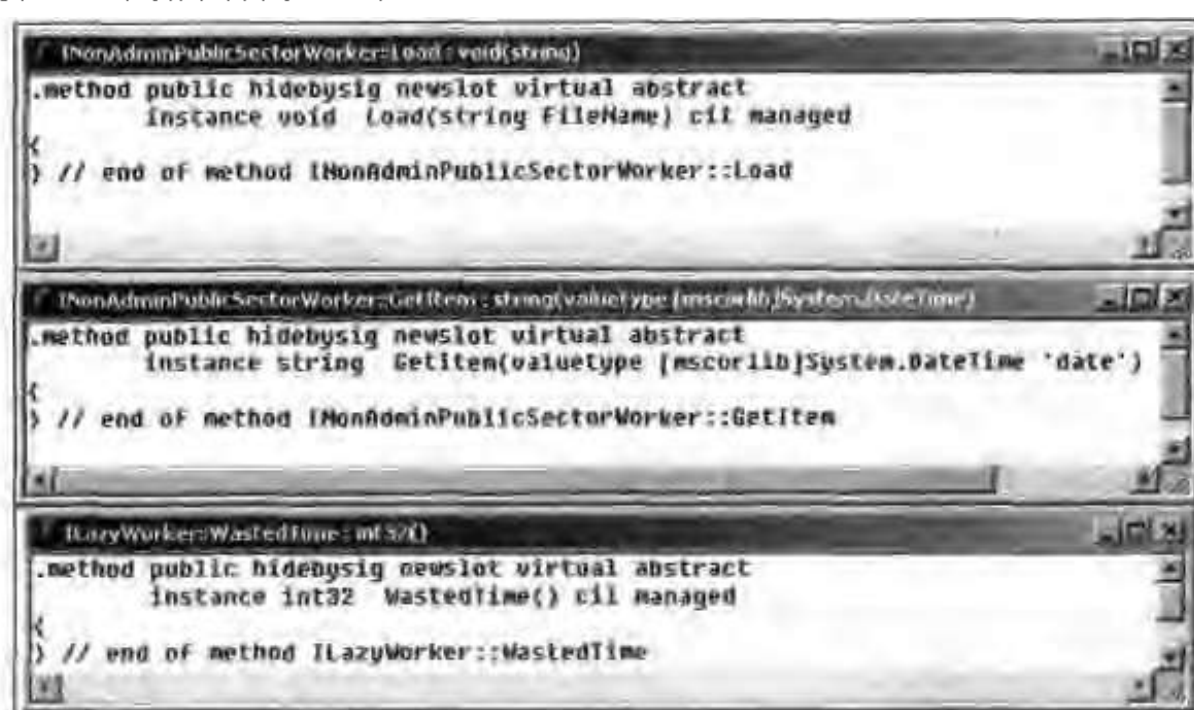
下面看看 `ILazyWorker` 接口的 MSIL。 `interface` 指令允许把一项添加到接口映射中， `abstract` 指令指定接口不能实例化(与抽象类一样)。见图 7-17。



```
.class interface private abstract auto ansi
{
} // end of class ILazyWorker
```

图 7-17

查看 MSIL 会发现，每个方法都使用了 `news slot` 指令，因为在这个接口中，此方法没有任何虚拟实现。没有重写这个定义，所有的内容都添加到接口的虚方法部分。即使 C# 代码中省略了定义，每个方法也都声明为抽象。前面用接口的实现定义了一个契约，其方式与抽象方法的实现完全相同。实现类的开发人员必须实现接口中的所有方法，因此，接口上的所有方法都必须定义为抽象类型。见图 7-18。



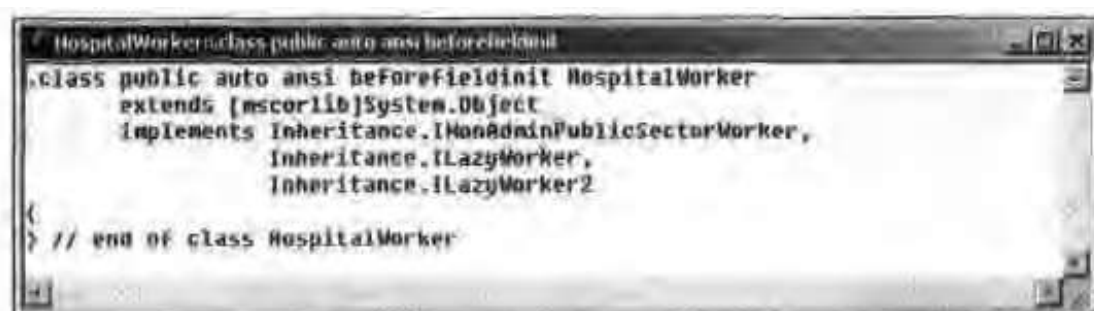
```
.method public hidebysig news slot virtual abstract
instance void Load(string FileName) cil managed
{
} // end of method INonAdminPublicSectorWorker::Load

.method public hidebysig news slot virtual abstract
instance string GetItem(valuetype [mscorlib]System.DateTime 'date')
{
} // end of method INonAdminPublicSectorWorker::GetItem

ILazyWorker::WastedTime: int32()
.method public hidebysig news slot virtual abstract
instance int32 WastedTime() cil managed
{
} // end of method ILazyWorker::WastedTime
```

图 7-18

下面评估一下实现类 `HospitalWorker` 的 MSIL 输出。Java 程序员应很熟悉在 MSIL 中表示接口继承的语法：`implements` 指令告诉 CLR 把这些接口添加到接口映射中，检查具体类是否实现了接口方法。见图 7-19。



```
.class public auto ansi beforefieldinit HospitalWorker
{
extends [mscorlib]System.Object
implements Inheritance.INonAdminPublicSectorWorker,
Inheritance.ILazyWorker,
Inheritance.ILazyWorker2
} // end of class HospitalWorker
```

图 7-19

在 `HospitalWorker` 类中，`INonAdminPublicSectorWorker` 和 `ILazyWorker` 的两个已实现的方法显示在 MSIL 中。MSIL 指令实现表示，该方法是使用 `news slot` 声明的，因为在接口中没有任何虚拟实现。还要注意这个方法声明为 `final`，这表示不能在派生类中重写它。在具体类中把方法声明为 `virtual`，就可以取消这个限制，也就是删除 `final` 指令，允许在派生类中重写它。见图 7-20。



图 7-20

在上面的代码例子中,在具体类 `HospitalWorker` 中实现的所有方法都声明为公共可见。为接口方法提供私有定义,就可以改变其可见性。如果使用接口类型引用,而不是具体类的引用类型,就可以调用私有成员。

7.2.6 界定接口方法的作用域

下面再看看 `ILazyWorker` 接口,注意没有使用访问修饰符:

```
interface ILazyWorker
{
    int WastedTime();
}
```

这个接口的具体实现, `HospitalWorker` 类声明了一个公共方法,来实现 `ILazyWorker` 接口的 `WastedTime()`方法:

```
public int WastedTime()
{
    return -1;
}
```

还声明了 `ILazyWorker.WastedTime()`方法,但没有指定访问修饰符,这将自动把方法的可见性设置为私有,因为类中的所有方法都默认为私有可见,除非进行了修改。

因此,具体类和接口就有了方法,允许调用不同的方法。公共方法返回一个强类型

的类实例，而私有方法返回一个必须转换的对象。另一个用途是，多个接口可以避免方法名和签名的冲突。这个规则要求，如果两个接口有一个方法的名称和签名相同，它们的实现在同一个具体类中就会共存，没有任何含糊不清。

这是通过使用指针来完成的，该指针从接口映射指向虚拟函数表，在虚拟函数表中，私有界定作用域的接口方法或公共方法在每个连续的接口方法映射的方法表中的同一个位置。重要的是使用什么类型的引用来调用方法，这是由 `RuntimeTypeHandle` 决定的。如果 `INonAdminPublicWorker` 接口上有一个对应的 `WastedTime()` 方法，那么就只能通过接口引用来使它可见。在一个具体类上实现许多接口时，可以使用这个方式，因为通过接口实现的多态性可避免名称冲突。

```
int ILazyWorker.WastedTime()
{
    return -2;
}
```

使用下面的代码获得对 `HospitalWorker` 类的一个具体类类型引用，并调用 `WastedTime()` 方法，然后引用 `ILazyWorker` 接口：

```
HospitalWorker h = new HospitalWorker();
Console.WriteLine(h.WastedTime());
ILazyWorker ilw = new HospitalWorker();
Console.WriteLine(ilw.WastedTime());
```

在实现类中声明的方法存储在方法表中，其方式与虚方法、重写方法或新方法存储在派生类(方法表)中的方式一样。接口映射由 CLR 使用，其中包含所有的接口，而不仅仅是当前类支持的接口。在该表中有一系列指向虚方法的指针，由类实现的每个接口支持。

对不支持的接口不能进行任何转换，因为方法表中没有指向类中方法的指针。查找接口映射，会在实现继承的使用上面添加第二层，因此导致较慢的实现。Microsoft 认为，JIT 编译器会对此进行优化。实现继承的使用需要额外的系统开销，因为用 `callvirt` 指令替代了 `call`，但代码的可维护性和私有限定作用域的接口方法的使用可以补偿这种性能的略微下降。

7.2.7 转换为不同的接口类型

多态性的概念是类类型与它是否支持契约、能否替换接口类型不相关。下面的例子说明了如何使用多个接口，把它们从具体类型转换为接口类型。首先调用 `WastedTime()` 方法，创建一个新的 `HospitalWorker` 类实例，`WastedTime()` 方法调用公共签名方法，

因为这里使用的是具体类类型引用。接着把 `HospitalWorker` 转换为支持的接口 `ILazyWorker`。然后，把所得的 `ilw` 接口类型引用字段转换为 `INonAdminPublicSectorWorker` 接口，因为底层的类型可以通过接口映射知道支持什么接口。编译器会过滤掉把具体类转换为接口类型时不支持的接口，并生成一个异常。

```
HospitalWorker worker = new HospitalWorker();
Console.WriteLine(worker.WastedTime());
ILazyWorker ilw = (ILazyWorker)h;
INonAdminPublicSectorWorker inaps2 = (INonAdminPublicSectorWorker)ilw;
Console.WriteLine(inaps2.GetItem(DateTime.Now));
Console.WriteLine(ilw.WastedTime());
```

这段代码会生成如下输出：

```
C:\Class Design\Ch 07>implementing_interfaces.cs
-1
-2
Not implemented!
```

两个接口都调用了对应的接口方法，而没有调用公共签名方法。这段代码的 MSIL 如下所示。在所有的实例中，无论是使用具体引用还是接口引用，每个方法调用都使用了 `callvirt` 指令。可以在支持的接口之间转换，是因为有 `castclass` 指令，它使用底层的基类类型检查是否支持接口。如果支持，就继续转换，这就是在接口之间转换时无需直接使用具体类引用的原因。见图 7-21。



图 7-21

7.2.8 名称的多义性和遮蔽

可以声明第二个接口，它的方法签名与第一个接口相同，但不会出现名称冲突，因

为具体类上只有一个公共签名。所以，WastedTime()方法在两个接口上是相同的。这就满足了公共契约，因为两个接口都支持这个方法。

```
interface ILazyWorker2
{
    int WastedTime();
}
```

要支持接口之间的遮蔽，可以重新定义上面的接口，该接口的方法名称相同，而方法签名略有不同。这样就可以把一个新接口引入具体类的定义，就好像 WastedTime()方法在公共接口上重载一样。另外，还可以像重载方法那样使用该方法，并提供 WastedTime(int)方法的实现，来替代无参数的 WastedTime()方法。这样，无参数的 WastedTime()方法就可以用 0 值调用第二个方法，而不会影响计算结果。这种方式适合于扩展实现模型，同时具有向后兼容性，并为在公共的具体类型接口中重载方法提供了一种安全的方式。

```
interface ILazyWorker2
{
    int WastedTime(int time);
}
```

还可以用下面的方式使用接口之间的继承。这仅是一个声明，与使用这两个接口的多重接口继承一样。实际上，实现类仍必须实现 ILazyWorker 和 ILazyWorker2。

```
interface ILazyWorker : ILazyWorker2
{
    int WastedTime();
}
```

7.3 小结

本章主要讨论了继承在代码中的使用，即通过使用 virtual 关键字声明，使用实现继承编写基类中的虚方法。在派生类中，通过使用 override 关键字重写虚方法。我们还论述了 MSIL，探讨了定义实现继承的许多关键字，包括 newslot 和 callvirt。

本章讨论了使用抽象基类比具体基类优越的地方，最终的基类是 System.Object，甚至结构(派生于 System.ValueType)也把 System.Object 作为最终的基类。

我们阐述了虚拟分派和非虚拟分派，包括 new 关键字的使用，强调了已继承成员的隐藏。介绍了如何在派生类中重写方法，或在对派生类对象实例的基类引用中访问

基类方法，以使用多态性注册继承类的行为。

本章还讨论了遮蔽的概念，如何用与继承或重写方法名相同、但签名不同的方法(或属性)，给类添加成员，允许指定一组重载方法，而无需修改基类上的接口。

每个类都有自己的方法表，其连续的区域存储了继承的虚方法和该类在虚拟函数表中的虚方法。重写方法使用该表改写虚拟函数表中的继承虚方法的内存地址。

我们介绍了如何使用接口和多重接口继承，接口映射如何辨识具体类支持的接口。IL 解释了操作及 CLR 编译和支持 C#代码的方式。讨论了多重接口继承的用法，说明了如何根据是使用接口类型引用还是使用具体类类型引用，通过接口的界定作用域和方法分派来避免名称冲突。

第8章 代码组织和元数据

这是本书的最后一章，主要讨论影响 C#应用程序的逻辑和物理组织的问题。本章的第一部分讨论应用程序如何在编码过程中进行逻辑组织，第二部分论述在准备部署的过程中如何把这种逻辑组织转换为文件的物理分区。

本章将讨论如何使用命名空间，将应用程序划分为相关类型的组。命名空间强调了应用程序的逻辑结构，避免了类型名称的冲突。.NET Framework 类库突出了命名空间的重要性；类库以模块的方式设计，命名空间增强了类库的组织。我们将学习在 C#中定义命名空间的方法，这些信息保存在 Visual Studio .NET 的什么地方，同时提供有助于有效使用命名空间的设计指导。本章主要学习以下内容：

- 讨论命名空间的定义以及如何创建嵌套的命名空间
- 分析程序集和元数据，其中包含了版本信息、运行应用程序所需的文件，和定义的各种类型等
- 讨论如何部署多程序集格式的应用程序
- 学习如何创建共享程序集，并将其放在全局程序集缓存(Global Assembly Cache, GAC)中，方便运行时查找程序
- 如何为代码建立说明文档，为程序集提供相关信息，例如公司名、版权声明，及程序集的描述等

本章中的部分内容使用了 Visual Studio .NET，而不仅仅局限于 Framework SDK。本章首先讨论命名空间。

8.1 使用命名空间建立应用程序的结构

在逻辑上，.NET 应用程序中的所有类型都定义在命名空间中。.NET Framework 类库自身也是一种逻辑上组织为有层次的命名空间结构。表 8-1 描述了.NET Framework 类库中定义的一些命名空间。

表 8-1

命名空间	描 述
System	包括在 .NET 应用程序中广泛使用的基本类、结构、接口、委托类型和事件。例如，System 命名空间包括 Object 类，其他所有类型都由该类继承而来。Object 提供了一些基本方法，例如：Equals、ToString 和 Finalize 等
System.Windows	包含创建基于 Windows 的 GUI 所用的类和其他类型
System.Web	包含创建 ASP.NET 页面和服务端控件时所用的类和其他类型
System.Data	包含数据库查询、更新和管理所用的基本 ADO.NET 类和类型
System.Xml	包含创建和处理 XML 文档所用的类和类型

当创建一个新的 C# 项目时，Visual Studio .NET 会为项目赋予一个默认的命名空间。打开 Visual Studio .NET，创建新的项目（例如，创建 Windows 应用程序）就可以证实这一点。现在，在 Solution Explorer 的项目图标上单击右键，选择快捷菜单中的 Properties，将显示如图 8-1 所示的 Property Pages 对话框。

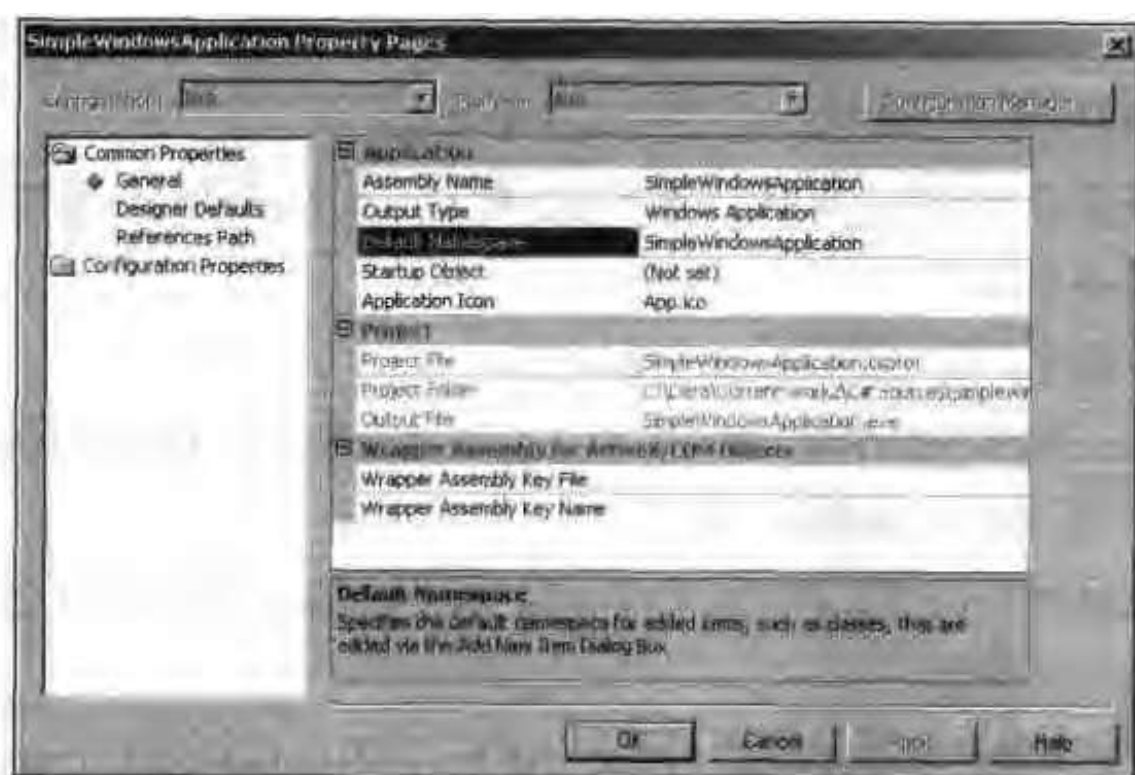


图 8-1

这个项目的默认命名空间是 SimpleWindowsApplication。在默认情况下，该命名空间与项目名称相同，当然也可以根据需要修改此命名空间。例如，当创建了多个不同的项目，希望它们位于相同的命名空间时，就可以将每个项目的默认命名空间分别修改为所需的命名空间。这将确保使用 Visual Studio .NET 向导添加的所有类都放在正确的命名空间中。向导把命名空间声明插入它们生成的所有文件中，就可以使类位于正确的命名空间。这些指令的命名空间名都是从这个属性中获得的。但改变该属性值不会改变项目中已有类的命名空间。另外，不要在项目属性对话框中删除默认的命名空

间项；如果删除了，而不是留下命名空间指令，Visual Studio .NET 就会给用向导添加的类增加一个空白的命名空间，这会导致一个编译错误，除非自己填充该命名空间。

注意：

在 Visual Basic .NET 代码中，代码本身并没有包含命名空间声明，但编译的类型都属于命名空间。VB.NET 有一个类似的项目级别的命名空间属性，但不使用它插入默认的命名空间声明，而是在编译时把所有的类型都封装在指定的命名空间中。所以在默认情况下，所有的 VB.NET 类型都属于与它们在其中声明的项目同名的命名空间。

有时也可以在代码中显式编写 namespace 语句。例如，在此应用程序中，按下面所示编辑 Form1.cs 文件，给 Form1 类添加一个额外的命名空间声明：

```
namespace SimpleWindowsApplication
{
    namespace MyNamespace
    {
        public class Form1 : System.Windows.Forms.Form
        {
            ...
        }
    }
}
```

所以可以定义嵌套的命名空间。在本例中，MyNamespace 的命名空间声明就嵌套在另一个命名空间 SimpleWindowsApplication 的声明中。因此，Form1 类的完全限定名称是 SimpleWindowsApplication.MyNamespace.Form1。

可以直接从 Visual Studio .NET 中建立应用程序，但如果要在命令行上编译这个应用程序，则创建简单窗体所需的代码如下所示，这些代码包含在本书的下载文件 Form1.cs 中：

```
// form1.cs
using System;
using System.Windows.Forms;

namespace SimpleWindowsApplication
{
    namespace MyNamespace
    {
        public class Form1 : System.Windows.Forms.Form
        {
```

```
static void Main()  
{  
    Application.Run(new Form1());  
}  
}  
}
```

如果在 MSIL Disassembler 窗口中查看生成的 IL, 则显示如图 8-2 所示的信息。该图中的窗口可以验证 Form1 类定义在 SimpleWindowsApplication.MyNamespace 命名空间中。



图 8-2

设计和实现命名空间

通过浏览 .NET Framework 可以对命名空间有一定的了解。 .NET Framework 类库中包含了数千个类和其他类型, Microsoft 将这些类和类型组织成下面所示的有层次的命名空间结构:

- 以 System 开始的命名空间包含了所有 .NET 语言通用的类和类型。例如, System.Data 包含 ADO.NET 类型; System.Drawing 中包含图形类型; System.Security 则包含与安全相关的类型等。
- 以 Microsoft 开始的命名空间包含了 Microsoft 专用的类和类型, 包括用于所有 .NET 语言的命名空间。例如, Microsoft.CSharp 包含支持 C# 编译和代码生成的类。 Microsoft.VisualBasic 包含了 Visual Basic .NET 运行时, 以及支持 Visual Basic .NET 的编译和代码生成的类和其他类型。

在项目的设计过程中, 首先需要考虑的是命名空间, 因为命名空间是将大系统划分成独立子系统的一种好方法。

许多组织使用统一建模语言(UML)来建立面向对象系统的模型。关于 UML 的更多

信息, 请参见 Wrox 出版社出版的“Instant UML”(1-86100-087-1)。UML 包含数据包的图形表示法, 其中包括多组在系统建模时需要使用的相关类和类型组。在面向对象设计中, 需要标识哪些类是相互依赖的, 并将这些类放入相同的包中。包可以将相互依赖的类划分成相关的逻辑和功能的单元, 从而使系统更易于理解。

1. 设计命名空间的示例

假设有一个系统可以从图书馆借书。系统中有类似 Member、Book 等的类。图 8-3 描述了该系统的 UML 对象简化模型。

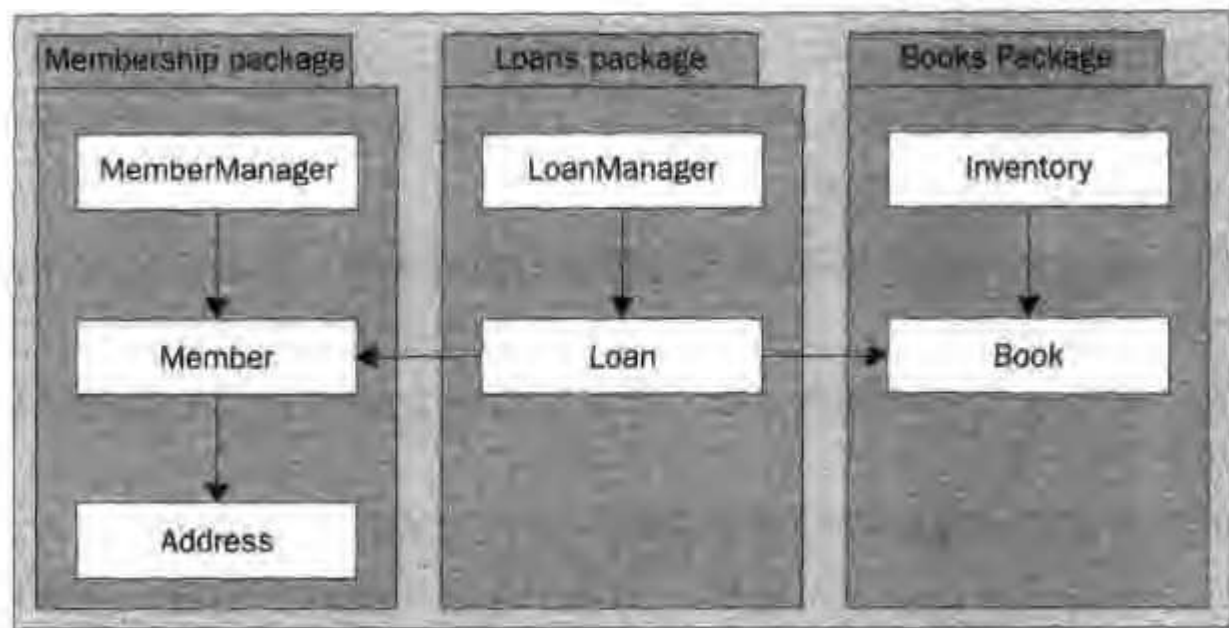


图 8-3

该模型中包含了 3 个包:

- **Membership** 包中包含实现图书馆的成员关系规则所需的所有类和类型。
MemberManager 对象中有 Member 对象的集合, 每个 Member 都有一个 Address。
- **Loans** 包中包含实现借书和还书规则所需的所有类和类型。LoanManager 对象中有 Loan 对象的集合。每个 Loan 对象都有对 Member 对象和 Book 对象的引用, 来表示哪个成员借了哪本书。
- **Books** 包中包含维护图书馆详细目录所需的所有类和类型。每个 Book 对象包含关于书的信息, 如作者、书名等。

无疑, 这只是一个简化的对象模型, 其中忽略了实际图书馆系统中的许多细节问题。但已能看出如何利用包将系统看作是松散耦合的子系统的集合。

在面向对象设计中, 需要将不同包中类之间的依赖性降为最小。这样可以在不影响系统中其他包的前提下, 方便地修改一个包中的实现细节。

2. 实现命名空间的示例

在按照 UML 中的逻辑映射开始编写系统的代码时, 每个包都应该由一个独立的命名空间表示。因此, 图书馆系统的 C# 应用程序将由 3 个命名空间组成, 如表 8-2 所示。

表 8-2

命名空间	命名空间中的类
Membership	MemberManager、Member 和 Address
Loans	LoanManager 和 Loan
Books	Inventory 和 Book

通过将 UML 包映射到代码中的命名空间，可以保持项目设计阶段和建造阶段的连续性。另外，当程序员需要在将来修改程序中的错误或添加新特征时，可以帮助他们理解应用程序是如何构造的。

命名空间定义了系统中类型的逻辑组。在 C#源文件中无需指定编写这些命名空间的具体位置。这里有以下几个选项：

- 在一个 C#源文件中编写所有命名空间中的所有类型。当命名空间很小且类型数量较少时，这是可行的。否则，源文件将包含非常多的类和类型，其尺寸将变得很大，从而导致代码很难浏览。这或多或少会妨碍开发小组同时协作代码的不同部分。
- 在独立的源文件中分别编写每个类型。这对于协作来说灵活性最大，但不同类型中的相关功能的开发会不连接。在这个系统中处理类型时，在不同的命名空间中引用一个类型，与在同一个命名空间中引用该类型一样简单，所以采用这种方式，可能常常需要跨命名空间之间的边界处理类型。
- 把相关的类型组放在同一个源文件中，忽略命名空间之间的边界。例如，可以把所有的实体类(在本例中是 Member、Book 和 Loan)放在一个源文件中，因为需要同时编写它们的代码。但这会使前面 UML 分析中建立的逻辑边界变得模糊。尽管如此，有时把不同命名空间中的两个类型放在同一个文件中是有意义的，这样它们会紧密相关，因为它们构成了两个命名空间中的两个子系统之间的接口。
- 把一个命名空间中的所有类型放在同一个源文件中，这与应用程序的逻辑结构最接近，但仅适用于每个命名空间中的类型比较少少的情况，此时一般需把命名空间分为几个相关类型的组。但是本例在每个命名空间中的类型都比较少，它们的关系也比较紧密。因此可以采用这种模式。

此处采用最后一种方法，创建一个由 4 个文件组成的 C#应用程序，如表 8-3 所示。

表 8-3

文件名	描述
membership.cs	实现 Membership 命名空间中的所有类型
books.cs	实现 Books 命名空间中的所有类型
loans.cs	实现 Loans 命名空间中的所有类型
library_app.cs	包含应用程序的 Main()方法

编写 Membership 命名空间

下面是 Membership 命名空间的全部源代码。代码定义在 membership.cs 文件中。我们将在列出代码后讨论重要的设计问题；首先需要指出的是 Membership 命名空间独立于应用程序中的其他命名空间，它只使用 system 命名空间：

```
using System;
using System.Collections;

namespace Membership
{
    public class MemberManager
    {
        private static MemberManager mInstance;
        private Hashtable mLibraryMembers;

        private MemberManager()
        {
            mLibraryMembers = new Hashtable();
        }

        public static MemberManager Instance
        {
            get
            {
                if (mInstance == null)
                {
                    mInstance = new MemberManager();
                }
                return mInstance;
            }
        }

        public Member CreateMember(string name, string street,
                                   string city)
        {
            Member newMem = new Member(name, street, city);
            mLibraryMembers[name] = newMem;
            return newMem;
        }

        public Member this [string name]
        {

```



```
        get { return (Member)mLibraryMembers[name]; }
    }
}

// ... more to follow
}
```

MemberManager 类是一个单独的类，这意味着该类只有一个实例。为了实现这一点，将构造函数定义为 **private**。另外，还提供了一个静态属性 **Instance**，它返回 **MemberManager** 的单一实例。该属性采用懒惰初始化，只在第一次请求时创建了 **MemberManager** 实例。

MemberManager 类使用 **Hashtable** 保存 **Member** 对象的集合。使用 **factory** 方法 **CreateMember** 创建 **Member** 对象，并添加到集合中。为了从集合中检索 **Member** 对象，可以把成员名作为一个键，创建一个只读索引器。注意这个简化的实现方式假定成员名是惟一的。

文件 **membership.cs** 还包含 **Member** 类的定义：

```
public class Member
{
    private string mName;
    private Address mAddress;

    public Member(string name, string street, string city)
    {
        mName = name;
        mAddress = new Address(street, city);
    }

    public override string ToString()
    {
        return mName + ", address: " + mAddress.ToString();
    }
}
```

Member 类中保存了每个成员的姓名和地址。而地址又封装在 **Address** 类中。每个类都提供了 **ToString** 方法，用来以文本格式检索信息。在 **Membership** 命名空间中添加 **Address** 类后，该命名空间就完整了。**Address** 类如下所示：

```
public class Address
{
```

```
        private string mStreet;
        private string mCity;
    public Address(string street, string city)
        {
            mStreet = street;
            mCity = city;
        }

    public override string ToString()
        {
            return mStreet + ", " + mCity;
        }
    }
}
```

该段源代码阐明了本章和前几章讨论的许多重要问题。最重要的是，它表示了一个可以用于其他基于成员的应用程序的通用成员关系架构。

编写 Books 命名空间

现在再看看 Books 命名空间的源代码。源代码定义在 books.cs 文件中。

```
using System;
using System.Collections;

namespace Books
{
    public class Inventory
    {
        private static Inventory mInstance;
        private Hashtable mLibraryBooks;

        private Inventory()
        {
            mLibraryBooks = new Hashtable();
        }

        public static Inventory Instance
        {
            get
            {

```

```
        if (mInstance == null)
            mInstance = new Inventory();
        return mInstance;
    }
}

public Book CreateBook(string title, string author,
                      string isbn)
{
    Book newBook = new Book(title, author, isbn);
    mLibraryBooks[isbn] = newBook;
    return newBook;
}

public Book this [string ISBN]
{
    get { return (Book)mLibraryBooks[ISBN]; }
}
}

public class Book
{
    private string mTitle;
    private string mAuthor;
    private string mISBN;

    public Book(string title, string author, string isbn)
    {
        mTitle = title;
        mAuthor = author;
        mISBN = isbn;
    }

    public override string ToString()
    {
        return mTitle + " (ISBN: " + mISBN + "), by: " + mAuthor;
    }
}
}
```

Inventory 类与 MemberManager 类之间有惊人的相似之处。首先，这两个类都是单一类，都使用 Hashtable 保存其他对象的集合 (Inventory 保存一组 Book 对象，而 MemberManager 保存 Member 对象的集合)。

另一个相似之处是 Inventory 和 MemberManager 提供对它们各自集合中对象的访问。这两个类都提供了一个 factory 方法来创建对象，并把它们添加到集合中，也提供了一个索引器，来检索项。在这个简化的例子中，Books 用 ISBN 的字符串表示建立索引。(ISBN 中的 N 表示数字，但 ISBN 不只包含数字，还可以以字符 X 结尾)。

请再次注意，Books 命名空间与应用程序中的其他命名空间不存在依赖性。这说明了命名空间之间进行松散耦合的设计目标。

编写 Loans 命名空间

Loans 命名空间包括 LoanManager 和 Loan 类。每次成员借书时，LoanManager 就创建一个新的 Loan 对象以标识书和借书者。同时，LoanManager 允许还书，这是图书馆的一个重要业务规则。

Loans 命名空间的代码定义在 loans.cs 文件中。在代码列出后将讨论设计问题：

```
using System;
using System.Collections;
using Membership;
using Books;

namespace Loans
{
    public class LoanManager
    {
        private static LoanManager mInstance;
        private Hashtable mLoans;

        private LoanManager()
        {
            mLoans = new Hashtable();
        }

        public static LoanManager Instance
        {
            get
            {
                if (mInstance == null)
                    mInstance = new LoanManager();
            }
        }
    }
}
```

```
        return mInstance;
    }
}

public void BorrowBook(string isbn, string member)
{
    Inventory inv = Inventory.Instance;
    MemberManager mem = MemberManager.Instance;

    Book theBook = inv[isbn];
    Member theMember = mem[member];

    if (theBook != null && theMember != null)
    {
        Loan theLoan = new Loan(theBook, theMember);
        mLoans[isbn] = theLoan;
        Console.WriteLine(theLoan.ToString());
    }
    else
        Console.WriteLine("Cannot borrow book");
}

public void ReturnBook(string isbn)
{
    if (mLoans.ContainsKey(isbn))
    {
        mLoans.Remove(isbn);
        Console.WriteLine("Book {0} has been returned", isbn);
    }
    else
        Console.WriteLine("Cannot return book");
}
}

public class Loan
{
    private Book mTheBook;
    private Member mTheMember;

    public Loan(Book TheBook, Member TheMember)
```

```
{
    mTheBook = TheBook;
    mTheMember = TheMember;
}

public override string ToString()
{
    return "Book: " + mTheBook.ToString() +
        "\nBorrowed by: " + mTheMember.ToString() + "\n";
}
}
```

在这个源文件中需要注意以下内容：

- 源文件中导入了 `Membership` 和 `Books` 命名空间，因此不需要使用诸如 `Membership.MemberManager` 之类的完全限定类名，就可以访问这些命名空间中定义的类。

此时，命名空间之间的依赖性是不可避免的。一定程度上的耦合是必然的；但是最重要的是标识发生耦合的位置，同时尽可能减小这种耦合。

- 在当前库系统中只有一个 `LoanManager`，因此，`LoanManager` 是单一类。
- 成员通过 `BorrowBook()` 方法借书。书和成员都利用惟一标识符标识。本例利用书的 ISBN 作为 `Inventory` 中的查询关键字，得到(具有此 ISBN 的) `Book` 对象的一个引用。同样，在 `MemberManager` 中使用成员名作为查询关键字，得到对此名称的 `Member` 对象的一个引用。显然，在真实的应用程序中，选择惟一标识符时要更小心，因为这些标识符可能表示数据库表中的主键。尽管本例在内部使用 `Hashtables`，仍可以修改 `Inventory` 和/或 `MemberManager` 类，以使用另一个存储技术，且不需要修改其他类。

如果可以定位 `Book` 和 `Member` 对象，则创建新的 `Loan` 对象记住哪个成员借了哪本书。`Loan` 对象存储在 `Hashtable` 中，利用书的 ISBN 作为查询关键字；对于查询关键字，这是一个好的选择，因为一本书一次只能被一个成员借阅。

- 通过 `ReturnBook()` 方法将书归还图书馆。在还书时，惟一需要的信息是书的 ISBN(而不需知道是谁还这本书，至少在当前模型中不需要知道这一点)。

如果能够在 `Hashtable` 中定位书的 ISBN，就意味着此书确实已被借出。此时，从 `Hashtable` 中删除 `Loan`，以表示该书已经归还。

- `Loan` 类引用了一个 `Book` 对象，还引用了一个 `Member` 对象。用 UML 术语说就是 `Loan` 是一个关联对象，它定义了 `Book` 和 `Member` 之间的关联。

为应用程序编写 Main()方法

前面学习了在应用程序中如何实现核心命名空间，余下的问题就是该如何编写 Main()方法使用这些命名空间了。Main()方法的代码如下所示，该方法保存在 library_app.cs 文件中：

```
using System;
using Membership;
using Books;
using Loans;

namespace LibraryApp
{
    public class LibraryApp
    {
        public static void Main()
        {
            Inventory inv = Inventory.Instance;
            inv.CreateBook("Professional C#", "Robinson",
                           "1-86100-704-3");
            inv.CreateBook("Instant UML", "Muller", "1-86100-087-1");
            MemberManager mem = MemberManager.Instance;
            mem.CreateMember("Georgia", "5th Avenue", "New York");
            mem.CreateMember("William", "Park Lane", "London");

            LoanManager loanmgr = LoanManager.Instance;
            loanmgr.BorrowBook("1-86100-704-3", "Georgia");
            loanmgr.BorrowBook("1-86100-087-1", "William");

            loanmgr.ReturnBook("1-86100-704-3");
            loanmgr.ReturnBook("1-86100-087-1");
        }
    }
}
```

上面的代码需要注意以下内容：

- 代码中导入了 Membership、Books 和 Loans 命名空间。该应用程序中使用了所有这些命名空间中的类。
- 在 Main()方法中，引用了 Inventory 对象，并将两本书添加到书库中。另外，还引用了 MemberManager 对象，并创建了两个成员。
- 引用 LoanManager 对象，两次调用 BorrowBook 方法从图书馆中借出两本书。

然后,调用 `ReturnBook` 方法两次,将所借的两本书归还图书馆。在这几种情况中,都是通过书的 ISBN 和成员名标识书和成员。

生成和运行应用程序

生成应用程序的方法之一是打开 .NET Framework 命令提示窗口,输入下面的命令:

```
C:\> csc library_app.cs membership.cs books.cs loans.cs
```

该命令编译了应用程序中的 4 个 C# 源文件,创建了一个程序集 `library_app.exe`。程序集根据编译的文件列表中的第一个源文件命名;如果要为程序集创建不同的名称,可以使用 `/out:filename` 编译器选项。另外一种选择是把每个命名空间编译为一个单独的 DLL。然后使用 `/r` 选项链接这些 DLL。Visual Studio .NET 还可以为编译提供其他不同的选项,根据需要选择最适当的方法。稍后介绍这些选项。

在运行应用程序时,在控制台窗口输出下面的信息:

```
Book: Professional C# (ISBN: 1-86100-704-3), by: Robinson  
Borrowed by: Georgia, address: 5th Avenue, New York
```

```
Book: Instant UML (ISBN: 1-86100-087-1), by: Muller  
Borrowed by: William, address: Park Lane, London
```

```
Book 1-86100-704-3 has been returned  
Book 1-86100-087-1 has been returned
```

3. 嵌套的命名空间

利用嵌套的命名空间可以定义多层体系结构,在应用程序中提供不同层次的抽象。这有助于掌握系统的全局形状,同时提供足够的信息,以有组织的、易于理解的方式为超大模型或复杂系统建模。

.NET Framework 类库是多层命名空间设计的一个很好的示例:

- 顶层命名空间 `System` 和 `Microsoft` 的区别是,前者包含系统范围的类(在 `System` 命名空间中),后者包含特定销售商的类(在 `Microsoft` 命名空间中)
- 嵌套的命名空间,如 `System.Data` 和 `System.Drawing`,将系统范围的类分割成逻辑组。每个命名空间都显示了较强的内聚力(同一个命名空间里的内容都是共同的)和较弱的耦合性(不同命名空间的类之间只有极少的依赖性)
- 更深层的嵌套还提供了更细的粒度。例如,表 8-4 表述了 `System.Data` 下的嵌套命名空间:

表 8-4

命名空间	描述
System.Data.SqlClient	包含访问 SQL Server 7.0(及以后版本)数据库的 ADO.NET 类和类型
System.Data.OleDb	包含访问 SQL Server 6.5(及以后版本)数据库和其他数据库, 如 Oracle 和 Microsoft Access 的 ADO.NET 类和类型
System.Data.Common	包含访问任何数据库的通用 ADO.NET 类和类型。例如, 该命名空间中包括 DataSet 类, DataSet 类表示在断开连接的应用程序中的数据的数据的内存中缓存
System.Data.SqlTypes	包含表示 SQL Server 中内部数据类型的类型

定义嵌套命名空间

在代码中, 可以采用两种不同的方法定义嵌套命名空间。下例分别使用了这两种方法。本例的源代码保存在下载文件夹中, 文件名为 `nested_namespaces.cs`:

```
namespace MyNamespace1
{
    public class MyClassA
    {
        // members
    }

    namespace MyNestedNamespace1
    {
        public class MyClassB
        {
            // members
        }
    }
}

namespace MyNamespace2.MyNestedNamespace2
{
    public class MyClassC
    {
        // members
    }
}
```

上面的代码需要注意以下内容：

- `MyNamespace1` 是外层命名空间，包含 `MyClassA` 类。该类的完全限定类名为 `MyNamespace1.MyClassA`。
- `MyNestedNamespace1` 定义在 `MyNamespace1` 内部，从而创建了嵌套命名空间 `MyNamespace1.MyNestedNamespace1`。该命名空间中包含 `MyClassB` 类，该类的完全限定类名为 `MyNamespace1.MyNestedNamespace1.MyClassB`。
- `MyNamespace2.MyNestedNamespace2` 是一个嵌套命名空间。这再次说明了在 `Namespace` 语句中如何使用“.”语法创建嵌套命名空间。该命名空间中包含 `MyClassC` 类，此类的完全限定类名为 `MyNamespace2.MyNested.Namespace2.MyClassC`。

使用嵌套命名空间

如果需要使用嵌套命名空间中定义的类，可以使用如下的完全限定类名。下面所示的源代码位于 `use_fully_qualified.cs` 文件中。

```
public class UseFullyQualified
{
    public static void Main()
    {
        MyNamespace1.MyClassA a = new MyNamespace1.MyClassA();
        MyNamespace1.MyNestedNamespace1.MyClassB b =
            new MyNamespace1.MyNestedNamespace1.MyClassB();
        MyNamespace2.MyNestedNamespace2.MyClassC c =
            new MyNamespace2.MyNestedNamespace2.MyClassC();
    }
}
```

打开 .NET Framework 命令提示窗口，输入下面的命令，将这些文件编译成一个程序集：

```
C:\> csc use_fully_qualified.cs nested_namespaces.cs
```

然后，在 MSIL Disassembler 中打开该程序集，看看编译器如何形成命名空间和类。MSIL Disassembler 窗口显示如图 8-4 所示的信息。



图 8-4

通过 MSIL Disassembler，可以知道程序集中包含了 3 个命名空间。

如果在代码中多次使用同一个类，那么使用类的完全限定类名就会很繁琐。为了减少键盘输入，可以在代码中按照下面的方法导入所需的命名空间。下面的源代码位于 `use_shortnames.cs` 文件中：

```
using MyNamespace1;
using MyNamespace1.MyNestedNamespace1;
using MyNamespace2.MyNestedNamespace2;

public class UseShortnames
{
    public static void Main()
    {
        MyClassA a = new MyClassA();
        MyClassB b = new MyClassB();
        MyClassC c = new MyClassC();
    }
}
```

编译上面的代码和 `nested_namespaces.cs`，如果在 MSIL Disassembler 中打开该程序集，将看到与前一个示例相同的类和命名空间。注意需要在 `using` 语句中分别指定每个命名空间，嵌套的命名空间也要指定。`using` 语句不会循环搜索嵌套命名空间，来解析类型名。

8.2 理解程序集

前面介绍了如何在设计阶段，把代码组织到命名空间和/或源文件的逻辑组中。下面讨论如何把这些逻辑组转换为文件的物理集，以便于部署。为此，需要理解程序集。

8.2.1 什么是程序集

程序集是 Microsoft 建立的术语，用于描述一组或多组逻辑上相关的文件，这些文件包含类型定义和/或资源。严格地说，程序集中的每个物理文件都称为模块，可以从一个或多个源文件中编译。

1. 单文件程序集

如果使用 Visual Studio .NET，就只能生成单文件(单模块)程序集。此时，解决方案中的每个项目都把一个程序集生成为其主要输出，它是一个 .exe 或 .dll(稍后讨论 ASP.NET)。在 Visual Studio .NET 解决方案中，所有这些程序集(可能是 .exe 和 .dll 的组合)构成了应用程序。

每个程序集都包含一个程序集清单，其中包含版本和安全信息。所以，程序集是可重用的最小单元。

2. 多文件程序集

如果不使用 Visual Studio .NET，就可以生成多文件程序集，这是一个模块的集合，每个模块都放在它自己的文件中。另外，每个模块也可以由不同的 .NET 语言编译器生成。

在这种情况下，程序集中只有一个模块包含程序集清单，该清单列出了组成程序集的其他文件(模块)。保存程序集清单的模块一般有 .exe 或 .dll 扩展名，而其他模块(在默认情况下)的扩展名是 .netmodule。稍后将介绍一个这方面的例子。

3. 逻辑组织和物理组织

无论使用什么方式编译代码，在发布应用程序时，上面的所有文件 .exe's、.dll's 和 .netmodule's 都必须部署在最终用户的机器上。

还要注意源代码的逻辑结构和已部署应用程序的物理结构之间没有关系。可以把多个源文件编译到一个模块中，组成较大程序集的一部分。命名空间可以跨几个程序集，一个模块也可以包含多个命名空间，每个命名空间又包含许多类型。还可以使用不同的编译条件选项，把同一个源文件编译到多个模块中(但不推荐这么做)，之后把它们组合到一个程序集中。

这很容易让人摸不着头脑。除了文件扩展名比较熟悉之外，没有与其他开发工具对应的地方。如果读者具有 C++ 背景，可能会认为 .netmodule 文件类似于 .obj 文件，只是一个模块可以从多个源文件中建立。但实际上还有更多的不同。

8.2.2 元数据

每个程序集，无论是单文件程序集，还是多文件程序集，都包含元数据。所有的 .NET 语言编译器都会生成元数据，元数据描述了代码中的类型和成员，替代了 C++ 头文件，接口定义语言 (IDL) 和 COM 开发人员已使用十年的类型库。这些信息全部存储在已编译的 .exe 或 .dll 文件中。元数据还列出了代码中引用的类型和成员，但没有列出其他程序集中定义的类型和成员。

元数据为公共语言运行时提供了在运行期间加载类型和调用方法所需的所有信息。CLR 还使用元数据加强其安全性。安全系统使用权限来防止未获得授权的代码访问资源。

本节首先学习如何查看单文件程序集中所有可用的元数据。然后学习如何创建多文件程序集，并讨论这样做的好处。

1. 查看单文件程序集中的元数据

查看元数据的最简单方法是使用 MSIL Disassembler 工具 ildasm.exe。前面已多次使用该工具，本章将更广泛地使用它。

下面看看简单控制台应用程序中的简单程序集的元数据。本例的源代码保存在 simple_console.cs 中：

```
using System;

namespace MetadataSimpleApp
{
    class SimpleConsole
    {
        public static void Main()
        {
            Console.WriteLine("Wrox Rocks!");
        }
    }
}
```

指定命名空间为 MetadataSimpleApp，生成该应用程序，使用 /adv 或 /advanced 命令行选项打开 MSIL Disassembler 工具，如下所示：

```
C:\> ildasm /adv simple_console.exe
```

从 MSIL Disassembler 中可以知道：

- MANIFEST 项提供了程序集的信息清单。本章稍后将介绍，程序集清单列出了程序集中的文件和类型，以及本程序集使用的其他程序集中的类型。
- 程序集中包含一个命名空间 `MetadataSimpleApp`。
- `MetadataSimpleApp` 命名空间中包含一个类 `SimpleConsole`。
- `SimpleConsole` 有一个编译器生成的构造函数和用户自己编写的 `Main()` 方法。

以上所示只是 MSIL Disassembler 工具中可以使用的部分信息。由于在运行 MSIL Disassembler 时使用了 `/adv` 选项，因此可以访问程序集中的高级信息，包括全部元数据。要查看该元数据，单击 View，选择 `MetaInfo`，然后选择 `Show!`，就在一个窗口中显示程序集的所有元数据。

该元数据包含了一些重要部分：

- 这个 .NET 模块的元数据
- 这个程序集的元数据。此处，程序集中只包含了一个 .NET 模块
- 这个程序集中定义的地类型的元数据
- 其他程序集中引用类型的元数据
- 其他引用的程序集的元数据

.NET 模块的元数据

元数据中的前两行提供了关于 .NET 模块的信息。 .NET 模块是一个可移植、可执行 (PE) 格式文件，例如 .exe 或 .dll 文件，或者文件扩展名为 .netmodule 的独立模块。前面列举的所有 C# 示例都是单模块程序集，这种程序集只能包含单一文件。本章后面的内容将学习如何创建包含多个 .NET 模块的程序集；此外，还将讨论为什么创建包含多个 .NET 模块的程序集很有用。

现在返回讨论 .NET 模块元数据，可以得到下面的模块元数据：

```
ScopeName : simple_console.exe  
MVID      : {9271A944-5B7E-4F7B-BA3C-97EDA8302130}
```

在 .NET 模块元数据中应该注意以下内容：

- `ScopeName` 是包括路径的 .NET 模块的名称。
- `MVID` 是模块版本 ID，它是编译器生成的 .NET 模块的全局唯一 ID (Globally Unique ID, GUID)。每次生成模块时，编译器都会生成不同的 MVID。

关于此程序集的元数据

在接近元数据列表的末尾，可以看到关于此程序集的元数据，如下所示(下面的内容略经省略，删除了关于自定义属性的元数据)：

Assembly

```
-----  
Token: 0x20000001  
Name : simple_console  
Public Key :  
Hash Algorithm : 0x00008004  
Major Version: 0x00000000  
Minor Version: 0x00000000  
Build Number: 0x00000000  
Revision Number: 0x00000000  
Locale: <null>  
Flags : [SideBySideCompatible] (00000000)
```

下面是部分程序集元数据的一些描述:

- **Token** 是一个 4 位元数据令牌。元数据的每个项都由不同的令牌标识。令牌的顶位表示元数据的类型: 例如, 0x20 表示程序集定义, 0x23 表示程序集引用, 0x02 表示类型定义等。打开 FrameworkSDK\include 子文件夹中的 C++头文件 CorHdr.h, 查找 CorTokenType 枚举的类型, 可以得到令牌类型的完整列表。
- **Name** 是不包括路径和扩展名的程序集名。
- **PublicKey** 是用来标记程序集的公钥。此处由于没有指定任何程序集, 因此它为空。本章稍后讨论如何在计算机中将共享程序集安装到全局程序集缓存(GAC)中时, 将学习标记程序集的方法。
- **Major Version**、**Minor Version**、**Build Number** 和 **Revision Number** 用来识别当前程序集的精确版本。这个 4 部分的版本号将当前程序集与其他版本的程序集区别出来, 并允许其他 .NET 应用程序指定需要使用的程序集版本。

4 部分版本号中的每一个都对 CLR 有特殊的意义, 因为它可以决定要加载的程序集的版本。假设程序集 A 引用另外一个程序集 B 的特定版本。运行时使用下面的规则来加载适当版本的程序集 B。

- 如果存在具有与 A 要求的相同的版本号的 B 程序集, 那么运行时加载该版本的 B 程序集。
- 如果存在具有不同 Build Number 的某版本的 B 程序集, 就表示对程序集进行了可兼容的修改(例如安全处理)。运行时将加载该版本的 B 程序集, 而不进行准确的匹配。
- 如果存在具有不同 Revision Number 的某版本的 B 程序集, 这表示对程序集进行了一个递增修改(例如 Service Pack)。运行时认为该程序集与 A 所需的版本兼容, 因此运行时将加载该版本的 B 程序集, 而不进行准确的匹配。
- 如果存在具有不同 Major Version 或 Minor Version 的某版本的 B 程序集, 这表示

与以前版本的程序集存在很大的区别(例如主要的新产品版本)。运行时将不加载该版本的 B 程序集。

关于此程序集中定义的类型元数据

在程序集中定义类型(例如类和结构)时,程序集会包含每种这些类型的元数据。简单的 C# 应用程序定义了一个类型,它是一个类 SimpleConsole。因此,程序集的元数据包含一个类型定义,如下所示:

TypeDef #1

```
-----
TypeDefName: MetadataSimpleApp.SimpleConsole (02000002)
Flags      : [NotPublic] [AutoLayout] [Class] [AnsiClass] (00100000)
Extends    : 01000001 [TypeRef] System.Object
Method #1 [ENTRYPOINT]
-----
MethodName: Main (06000001)
Flags      : [Public] [Static] [HideBySig] [ReuseSlot] (00000096)
RVA        : 0x00002050
ImplFlags : [IL] [Managed] (00000000)
CallConvtn: [DEFAULT]
ReturnType: Void
No arguments.
```

在该元数据中需要注意以下内容:

- TypeDefName 是类型的名称。该类型的名称为 MetadataSimpleApp.SimpleConsole。括号中的值是这段元数据的元数据令牌(顶位上的 0x02 值表示这是一个类型定义)。
- Flags 提供了 MetadataSimpleApp.MyModule 的更多信息。下面是简要说明: [NotPublic]表示 MyModule 不是一个公共类型; [AutoLayout]表示它的成员布局由 CLR 自动处理; [Class]表示它是一个类; [AnsiClass]表示 LPTSTR 字符串类型作为 ANSI 字符串处理(而不是 Unicode 字符串)。
- Extends 表示该类型从 System.Object 继承而来。
- Method #1 提供了类型中第一个方法的元数据。该元数据表示方法的名称(Main)、其参数(none)、其返回类型(void), 和该方法的其他信息, 例如方法为 public 和 static。由于只定义了一个方法, 则元数据中的另一个方法就是编译器生成的构造函数。

关于其他引用程序集的元数据

程序集中还包含该程序集引用的所有其他程序集的元数据。很明显, CLR 需要这

些信息，因此可以在需要的时候加载这些程序集。当前程序集引用了其他程序集，例如：

```
AssemblyRef #1
-----
Token: 0x23000001
Public Key or Token: b7 7a 5c 56 19 34 e0 89
Name: mscorlib
Major Version: 0x00000001
Minor Version: 0x00000000
Build Number: 0x00000ce4
Revision Number: 0x00000000
Locale: <null>
HashValue Blob:
Flags: [none] (00000000)
```

引用的程序集的元数据需要注意以下内容：

- **Public Key or Token** 表示用来标记被引用的程序集的公钥。
- **Name** 表示被引用的程序集名为 **mscorlib**。
- **Major Version**、**Minor Version**、**Build Number** 和 **Revision Number** 表示需要引用的程序集的精确版本。即使计算机上随后安装了程序集的新版本，也可以保证总是引用正确版本的程序集。
- **Locale** 方便了程序集的定位。可以存在以不同语言(例如英语和法语)编写的包含相同数据副本的多个程序集。可以给每个程序集赋予不同的位置。程序集中的 **Locale** 元数据表示需要引用哪个特定位置的程序集。

引用的程序集中使用的类型的元数据

除了引用的程序集的信息之外，还有代码中使用的这些程序集中的类型信息。例如：

```
TypeRef #4 (01000004)
-----
Token:      0x01000004
ResolutionScope: 0x23000001
TypeRefName:  System.Console
MemberRef #1
-----
Member: (0a000003) WriteLine:
CallConvntn: [DEFAULT]
ReturnType: Void
1 Arguments
  Argument #1: String
```

这表示, 不仅引用 `System.Console` 类型, 还引用了这个类型的一个重载方法 `WriteLine`。在更复杂的应用程序中, 就可以从元数据中看到本程序集使用的其他程序集中的类型和类型成员。

2. 创建多文件程序集

本书谈到的所有例子都使用单文件程序集。单文件程序集由单个 PE 文件(或.NET 模块)组成。PE 文件通常指 .dll 或 .exe 文件, 其中有一个程序集清单和描述程序集中类型的元数据。

有时还需要创建包含多个 .NET 模块的程序集(因此名称为“assembly”)。当编译每个 .NET 模块时, 会得到一个包含 .NET 模块的 MSIL 代码和元数据的 PE 文件。程序集中的一个 .NET 模块必须标识为“main”模块。此 .NET 模块包含标识程序集中其他所有 .NET 模块的清单。主模块的扩展名一般是 .exe 或 .dll, 其余模块的扩展名是 .netmodule。

将程序集实现为独立 .NET 模块的集合, 是因为:

- 可以用不同的 .NET 程序设计语言编写每个模块。这对使用不同语言编写系统不同部分的公司是很有用的。
- 可以将类型组织在独立的模块中, 从而优化将代码加载到公共语言运行时中的方法。可以将相关的类型放在同一模块中, 因此在模块加载到运行时中时, 可以将这些类型一起加载。很少使用的类型可以放在独立的模块中; 运行时只有在应用程序需要这些类型时, 才加载该模块。通过这种方法, 可以将应用程序的占用内存最小化, 并减少加载时间。
- 可以包含只有数据的模块, 这些模块中包含图片、XML 文档、音乐片段等资源。 .NET Framework SDK 包含一个工具 `Assembly Linker(AL.exe)`, 利用该工具将这些非 MSIL 模块链接到程序集中。

多文件程序集示例

本节将学习如何创建包含 3 个独立模块的程序集。模块采用不同的 .NET 程序设计语言编写, 然后将这些模块链接在一起, 来创建一个单一逻辑的程序集。这是根据多个独立模块创建单一程序集(而不是仅仅根据单一模块创建程序集)的一个典型原因。

以下是对本例中 3 个模块的描述:

- 第一个模块为采用 C# 编写的 `CSMod` 模块, 其中包含一个类 `DistanceConverter`, 利用该类将相距单位在英里和公里之间转换。
- 第二个模块是采用 Visual Basic .NET 编写的 `VBMod` 模块。该模块包含一个类 `TempConverter`, 该类将温度单位在摄氏与华氏之间转换。
- 第三个模块是采用 C# 编写的 `MainMod` 模块。该模块将作为程序集的主模块, 也就是说文件中将包含标识程序集中所有其他模块的清单。

模块采用下面的方法形成一个多文件的程序集，如图 8-5 所示。



图 8-5

程序集中的每个模块都包括 MSIL 指令和元数据；元数据描述了模块定义和引用的类型。程序集中的主模块为 MainMod，该模块中包含程序集清单。清单中标识了程序集中使用的所有其他模块(VBMod 和 CSMod)，和在这些模块中定义的公共类型。在讨论整个例子时，将说明元数据的概况。

CLR 需要这些清单信息，因此在应用程序中使用这些类型时，它可以加载适当的模块。

下面来看看如何实现上面所说的内容。实现过程分为以下 3 步：

- 编写 CSMod.cs 的源代码，并将其编译成一个 .NET 模块。
- 编写 VBMod.vb 的源代码，并将其编译成另一个 .NET 模块。
- 编写 MainMod.cs 的源代码，并将其编译到程序集的主 .NET 模块中。在此阶段，需要指定程序集中需要包含其他哪些模块(本例为 VBMod 和 CSMod)。

创建 CSMod 模块

以下是 CSMod.cs 模块的 C#源代码。该模块的源代码(和本例中将使用的所有其他模块的源代码)保存在下载文件夹 MultiFileAssembly 中。

```
public class DistanceConverter
{
    public static double MileToKm(double miles)
    {
        return (miles * 1.6093);
    }
}
```

```
public static double KmToMile(double km)
{
    return (km / 1.6093);
}
```

要将该文件编译成模块(与独立程序集相反), 必须使用 C#.NET 命令行编译器。不能使用 Visual Studio .NET 创建 .NET 模块, 它总是为项目创建单一程序集(该规则对于 Managed Extensions for C++ 项目例外, 对于 Managed Extensions for C++ 可以设置 /NOASSEMBLY 选项, 来创建一个模块而不是一个程序集)。

打开 .NET Framework 命令提示窗口, 按照下面的方式编译代码:

```
C:\> csc /target:module CSMod.cs
```

编译器选项 /target:module 通知编译器生成一个模块(在本例中是 CSMod.netmodule), 而不是程序集。模块中包含代码的已编译 MSIL 指令, 和描述代码中定义和引用类型的元数据。然后, 将该模块与其他模块相链接, 创建多文件程序集。

注意:

/target 编译器选项可以简写为 /t。

创建 VBMod 模块

以下是 VBMod.vb 的 Visual Basic .NET 源代码:

```
Public Class TempConverter
    Public Shared Function CelsiusToFahr(ByVal C As Double) As Double
        Return (C * 9.0 / 5.0) + 32
    End Function

    Public Shared Function FahrToCelsius(ByVal F As Double) As Double
        Return (F - 32) * 5.0 / 9.0
    End Function
End Class
```

要将该文件编译成独立模块, 按照以下的方法运行 Visual Basic .NET 编译器:

```
C:\> vbc /target:module VBMod.vb
```

运行后生成名为 VBMod.netmodule 的 .NET 模块。注意, 在这个阶段, 该模块与前面刚刚创建的模块 CSMod.netmodule 之间并没有链接。

在程序集中创建 Main 模块

以下是 MainMod.cs 的源代码，其中使用了其他两个模块中定义的类型：

```
using System;

public class MyClass
{
    public static void Main()
    {
        Console.WriteLine("Select an option: ");
        Console.WriteLine("  1  C to F");
        Console.WriteLine("  2  F to C");
        Console.WriteLine("  3  Miles to Km");
        Console.WriteLine("  4  Km to Miles");
        Console.WriteLine("=>");

        string input = Console.ReadLine();
        int opt = Int32.Parse(input);

        Console.WriteLine("Value to convert: ");
        input = Console.ReadLine();
        double value = Double.Parse(input);

        double result;
        switch (opt)
        {
            case 1:
                result= TempConverter.CelsiusToFahr(value);
                break;
            case 2:
                result= TempConverter.FahrToCelsius(value);
                break;
            case 3:
                result= DistanceConverter.MileToKm(value);
                break;
            case 4:
                result= DistanceConverter.KmToMile(value);
                break;
            default:
                Console.WriteLine("Invalid option");
        }
    }
}
```

```

        return;
    }

    Console.WriteLine("Result: {0}", Math.Round(result, 2));
}
}

```

输入下面的命令编译源文件：

```

C:> csc /addmodule:CSMod.netmodule /addmodule:VBMod.netmodule
MainMod.cs

```

该命令将 MainMod.cs 编译到 MainMod.exe 中。MainMod.exe 是程序集中的主模块。编译器选项 /addmodule 可以将项添加到程序集的清单中，从而指出 VBMod.netmodule 和 CSMod.netmodule 是程序集的一部分。

注意：

如果忘记使用 /addmodule 选项，那么每次在试图访问在其他模块(未指定的模块)中定义的类型时，都会产生编译器错误。这是因为编译器并不知道它需要在其他模块中查找来解析类型定义。

在 MSIL Disassembler 中，使用 /adv (或 /advanced) 选项打开 MainMod.exe，双击程序集清单图标，查看程序集中的清单信息。下面是程序集清单信息中最重要的部分：

```

.module extern VBMod.netmodule
.module extern CSMod.netmodule
...
.assembly MainMod
{
    .hash algorithm 0x00008004
    .ver 0:0:0:0
}
.file CSMod.netmodule
    .hash = (6C CB 79 30 7E D2 C0 E8 F2 57 4A 3F 0F 84 EC FC
            75 7D A9 B0 )
.file VBMod.netmodule
    .hash = (B3 F0 2D E0 C0 34 AC 48 50 76 CA 72 FC 9C 59 85
            16 C1 2A D7 )
.class extern public DistanceConverter
{
    .file CSMod.netmodule

```



```
.class 0x02000002
}
.class extern public TempConverter
{
    .file VBMod.netmodule
    .class 0x02000002
}
.module MainMod.exe
// MVID: {43979DDE-E7E8-41C0-BD07-3718B82ECC9A}
.imagebase 0x00400000
.subsystem 0x00000003
.file alignment 512
.corflags 0x00000001
// Image base: 0x03090000
```

在清单中需要注意以下内容：

- 清单引用程序集中的其他模块(VBMod.netmodule 和 CSMOD.netmodule)。
- 清单还列出了这些模块中定义的公共类型。当程序集的用户使用其中某个类时，该信息用来通知公共语言运行时加载哪个模块。

说明：

创建多文件程序集时，必须记住要在程序集中部署所有模块。在上面的例子中，必须在程序集 MainMod.exe 中部署 VBMod.netmodule、CSMod.netmodule 和主模块。

8.3 把应用程序部署为程序集

上一节讨论了如何创建由一个或多个模块组成的程序集。如果需要创建多文件程序集，程序集中的主模块包含标识程序集中的所有其他模块的清单信息。

本节将扩展我们学习的范围，考虑一下整个应用程序的物理组织。应用程序存在 3 种部署场景：

- 对简单应用而言，可以将应用程序部署为一个程序集。
- 对于较大应用程序而言，可以将部分功能分割成独立的程序集。把相关的功能组合成同一程序集，这样 CLR 可以从单一程序集中加载它需要的所有信息。将很少使用的功能转移到一个独立的程序集中，只在需要时加载它们。例如，如果查看 .NET Framework 中的类，就会发现 Microsoft 把部分命名空间放在核心

程序集中,而其他部分放在次级程序集中。还可以将资源(例如位图)放在另一个程序集中,方便地实现交互应用。

为了强调应用程序的逻辑组织,可以根据命名空间边界来组织程序集,将每一个命名空间作为独立的程序集实现。

通常,可以在目标计算机上将所有程序集部署在同一文件夹中。还可以提供一个配置文件,为私有的程序集指定不同的子文件夹。

- 如果需要部署几个共享许多常用代码的应用程序,可以创建所有应用程序共享的常用程序集。可以在目标计算机上将常用程序集安装到全局程序集缓存(GAC)中,这样程序集就可以由多个应用程序访问。本节稍后将讨论 GAC。

8.3.1 部署单程序集应用程序

部署单程序集应用程序是3种情况中最简单的一种,此时需要部署由一个程序集组成的应用程序。要部署这样的应用程序,只需将程序集复制到目标计算机上即可。这就是通常所说的 XCopy 部署,该名称以 DOS 命令 XCOPY 命名,用于把完整的文件夹结构从发布介质复制到目标计算机上。当然,也可以使用拖放操作或复制-粘贴技术,在 Explorer 中部署应用程序,但原理是相同的。此时,不需要在系统中注册任何信息,因此在计算机上终止现有的应用程序工作时不存在任何危险。这样排除了计算机上安装传统 Windows 应用程序时引起的“DLL Hell”。这种应用程序总是采用某种方式更新注册表,从而妨碍了现有应用程序的正常运行。

卸载简单.NET Framework 应用程序的方法也很简单。只需删除在安装时复制的文件即可,例如程序集的.exe和.dll文件,以及安装的任何配置文件(本章稍后将讨论配置文件)。将这种方法与卸载传统 Windows 应用程序的方法相比较:可以保证所有的注册项都被清除,而不需中断其他应用程序的工作。卸载传统 Windows 应用程序经常给人留下一一种猜疑,总是怀疑应用程序还残留在硬盘的某个角落。

注意:

还存在其他需要为发布程序而打包.NET 应用程序的情况。例如,如果希望能在 Internet 上下载应用程序,一般应将应用程序放在一个.CAB 文件中,以保证简单地下载。另外,也可以将应用程序打包在一个.MSI 文件中,这样,该应用程序可以利用 Microsoft Windows Installer 安装。例如,用户可以使用 Microsoft Systems Management Server (SMS)安装应用程序。

8.3.2 使用私有程序集部署应用程序

本节学习如何部署包含多个程序集的应用程序。此处假设该应用程序只需要这些程序集，把所有的程序集都安装在与主应用程序程序集相同的文件夹(或子文件夹)中。由于这些程序集只由一个应用程序使用，因此称它们为私有程序集。

注意：

稍后将讨论如何创建共享程序集，为了保证所有应用程序都可使用共享程序集，将其部署在主要的全局程序集缓存中。

1. 在同一个文件夹中部署私有程序集

假设有一个提供多个不同服务的大型应用程序。将应用程序组织成独立的程序集是很有意义的，这样在用户运行应用程序时，公共语言运行时可以只加载需要的程序集。一般情况下，其中的一个程序集为.exe 文件，其他的则为.dll 文件。

在同一个文件夹中部署程序集的示例

下面来看一个简单的例子，在该示例中有一个.exe 程序集文件和一个.dll 程序集文件。DLL 文件中包含了.exe 文件使用的有用函数。此例将执行以下任务：

- 编写 DLL 文件的源代码。
- 将源代码编译成.dll 程序集文件。
- 编写可执行文件的源代码。根据需要使用 DLL 提供的功能。
- 将该源代码编译成.exe 程序集文件。通知编译器引用了哪些其他程序集，让编译器解析这些程序集的引用。
- 部署.exe 文件和.dll 文件。最简单的部署情况是将.exe 文件与.dll 文件安装在相同的文件夹中。

下面是 DLL 文件的源代码。源文件为 my_useful_library.cs：

```
using System;

public class MyUsefulClass
{
    public static void MyUsefulMethod()
    {
        Console.WriteLine("Hello world");
    }
}
```

要将该源文件编译成.DLL 程序集文件,需要在命令提示行中运行 C#.NET 编译器,如下所示:

```
C:\> csc /target:library my_useful_library.cs
```

编译器选项/target:library 通知编译器创建库文件(.dll 文件)。

现在看看可执行文件的源代码。源代码文件为 hello_world.cs:

```
public class HelloWorld
{
    public static void Main()
    {
        MyUsefulClass.MyUsefulMethod();
    }
}
```

要将源文件编译成.exe 程序集文件,按照如下的方式在 .NET Framework 命令提示窗口中运行 C#编译器:

```
C:\> csc /reference:my_useful_library.dll hello_world.cs
```

由于使用了 /reference 编译器选项,因此可执行文件的清单中包含对 my_useful_library.dll 程序集文件的引用。

注意:

/reference 编译器选项可简写为 /r。

在目标计算机上部署应用程序时,最简单的方法是将 hello_world.exe 和 my_useful_library.dll 文件安装在同一文件夹中。对应用程序而言,my_useful_library.dll 是一个私有程序集。

当用户运行 hello_world.exe 时,CLR 在下列位置搜索程序集:

- my_useful_library.dll
- my_useful_library\my_useful_library.dll

如果 CLR 找不到文件扩展名为.dll 的程序集,它将重新查找文件扩展名为.exe 的程序集:

- my_useful_library.exe
- my_useful_library\my_useful_library.exe

如果仍然找不到程序集,CLR 就抛出 System.IO.FileNotFoundException 异常。

创建和引用特定文化的程序集

.NET Framework 允许创建多个不同版本的程序集,以满足不同地域的需要。例如,

可以创建英语版程序集、西班牙语版程序集、法语版程序集等。Microsoft 使用术语“文化”表示程序集的使用场所。本章后面的章节将学习在创建程序集时，如何使用属性来设置文化。

应用程序可以指定需要绑定的程序集的文化。当 CLR 查找特定的程序集时，它将与文化同名的子目录中查找。例如，如果需要 `my_useful_library` 程序集，并指定 `en` 文化(英语)，运行时将在下列位置查找程序集：

- `en\my_useful_library.dll`
- `en\my_useful_library\my_useful_library.dll`
- `en\my_useful_library.exe`
- `en\my_useful_library\my_useful_library.exe`

2. 在不同的子文件夹中部署私有程序集

毫无疑问，应用程序可以很大。将所有的程序集保存在同一文件夹中并不是最好的方法；例如，如果应用程序中包含 12 个不同的程序集，最好将这些程序集组织到反映应用程序中命名空间层次结构的目录结构中。

如果需要在名称不同的子文件夹中部署私有程序集，可以创建一个应用程序配置文件，来通知 CLR 在什么位置查找程序集。配置文件与请求程序集保存在相同的文件夹中，文件扩展名为 `.config`。例如，`MyApp.exe` 的配置文件名应该为 `MyApp.exe.config`。

应用程序配置文件是一个 XML 文件，它通知 CLR 私有程序集的具体位置。应把编写配置文件作为开发的一部分，并与应用程序一起部署配置文件。

注意：

.NET Framework 广泛使用 XML 作为表达配置信息和其他目的的标准方法。例如，当应用程序与数据库断开连接时，XML 用来表示数据(详见 `System.Data.DataSet` 类)，在 ASP.NET 中表示标记，向 Web Service 传递信息，并从中得到信息(详见 `System.WebServices.WebService`)等。

下面的例子创建和使用应用程序配置文件。假设有一个包括两个程序集的应用程序，这两个程序集保存在下面的文件夹中。

```
C:\MyAppFolder\MyApp.exe
C:\MyAppFolder\MySubFolder\MyUsefulLibrary.dll
```

下面是 `MyApp.exe.config` 配置文件的示例。该文件是 `MyModule.exe` 应用程序的配置文件：

```
<configuration>
  <runtime>
```

```
<assemblyBinding
  xmlns="urn:schemas-microsoft-com:asm.v1">
  <probing privatePath="MySubFolder"/>
</assemblyBinding>
</runtime>
</configuration>
```

该配置文件需要注意以下内容：

- 顶层元素必须是 configuration。configuration 元素可以有許多子元素，来配置运行时执行的不同方面。
- runtime 子元素指定公共语言运行时如何处理程序集加载和垃圾回收。
- assemblyBinding 子元素包含关于程序集的位置和版本信息。该元素必须利用 XML 命名空间“urn:schemas-microsoft-com:asm.v1”进行限定，使 CLR 正确识别它。
- assemblyBinding 元素包含子元素 probing。
- probing 元素有 privatePath 属性，该属性指定 CLR 从中查找私有程序集的子文件夹名。为了用户能在相同配置文件中指定多个文件夹(为不同的程序集)，可以指定多个目录，目录之间采用分号隔开。
- 注意此处必须指定了文件夹，不能指定绝对文件夹或相对文件夹，如..\SomeOtherFolder。该规则最小化了程序集在任何计算机上安装时发生冲突的可能性。

说明：

XML 是区分大小写的，因此必须按照上面的例子准确输入元素名和属性名。在配置文件中还可以指定许多其他信息，包括网络设置、加密设置、ASP.NET 设置等。如果需要了解更多的信息，请参阅 Visual Studio .NET 帮助中的<configuration>元素。

当用户运行 MyApp.exe 时，CLR 在下面的路径中查找 MyUsefulLibrary 程序集(按照如下所示的顺序)：

- MyUsefulLibrary.dll
- MyUsefulLibrary\MyUsefulLibrary.dll
- MySubFolder\MyUsefulLibrary.dll
- MySubFolder\MyUsefulLibrary\MyUsefulLibrary.dll

如果 CLR 找不到文件扩展名为.dll 的程序集，它将重新查找文件扩展名为.exe 的程序集：

- MyUsefulLibrary.exe
- MyUsefulLibrary\MyUsefulLibrary.exe

- MySubFolder\MyUsefulLibrary.exe
- MySubFolder\MyUsefulLibrary\MyUsefulLibrary.exe

最后需要指出的是,应用程序配置文件只是简单的 XML 文本,配置文件与应用程序一起部署在用户的计算机上。如果用户需要将私有程序集从它原来所在的文件夹中组织到另一个子文件夹层次结构中,可以随时编辑配置文件,以指明私有程序集的新位置。用户没有必要修改任何注册设置,因此相对 COM 的用户而言,出错的机率有所降低。

8.3.3 部署共享程序集

前面的讨论说明了如何创建私有程序集,以及如何在应用程序的子文件夹中部署私有程序集,以保证应用程序可以访问这些程序集。有时还需要创建共享程序集,并将其添加到全局程序集缓存(GAC)中。GAC 中保存了在目标计算机上运行的所有应用程序都可以访问的共享程序集信息。例如,.NET Framework 类库的基本程序集保存在 GAC 中。

听起来这似乎与 COM 有些类似。我们可以创建包含可重用功能的 COM 组件,然后安装这些组件,从而使用此功能。但是,COM 组件在系统注册表中注册,这可能会与注册表中的其他版本的组件产生冲突。.NET 程序集并不在系统注册表中注册,因此不存在产生注册冲突的可能性。

安装新版本的.NET 共享程序集后,不会引起 GAC 冲突。在 GAC 中添加程序集时,必须给程序集添加强名称。强名称的程序集具有名称、版本、公钥及可选的文化信息;这些信息惟一标识程序集,以免与其他程序集发生冲突。

然而,Microsoft 不推荐应用程序与安装在 GAC 中的程序集相互依赖,而是推荐使用私有程序集。在大多数平台中,磁盘空间是很平常的,即使应用程序位于 GAC 中,保存在不同应用程序域中的应用程序也会独立加载代码,因此不会终止节省内存。将所有应用程序的文件一起保存,避免对 GAC 注册的依赖,可以简化我们的工作。

注意:

在开发使用 COM+服务(也称为 Serviced 组件)的.NET 类时,不能使用 GAC。.NET Framework 提供了使用属性声明 COM+设置的方式,允许使用分布式事务管理和对象池等功能。在 COM+服务器应用程序中使用服务组件的程序集应放在 GAC 中。这是因为程序集需要两个主机过程来定位,即客户过程(绑定程序集的类型)和 COM+代理过程 `dllhost.exe`(加载和执行程序集)。这两个过程不能位于与程序集相同的文件夹中,所以需要在 GAC 中安装它。

应用程序可以提供特定程序集的名称、版本、公钥和可选的文化信息,准确地指定需要使用的程序集。由于其他程序集具有不同的名字、版本、公钥或可选的文化信息,

因此应用程序不会得到其他类似的程序集。这是应用程序部署和配置的主要进步。这标志着 DLL Hell 的末日, 对于我们而言这也是一个很好的解脱。

注意, 只有强名称的程序集才可以插入到 GAC 中; 由于没有强名称的程序集(也就是程序集没有公钥)中包含的信息不够充分, 容易引起冲突, 因此不能将这些程序集添加到 GAC 中。

注意:

利用文化信息可以在不同的地域创建不同的程序集。例如, 可以创建 3 个包含不同文化的程序集, 一个包含英语文本的程序集, 另一个包括法语文本的程序集, 第三个则是包含西班牙语文本的程序集。每个程序集都可以具有相同的名称、版本和公钥, 但不能具有相同的文化标记。

下面对前面所讲内容作一个总结。如果需要创建可以由多个应用程序访问的共享程序集, 必须按照以下的步骤创建:

- 创建强名称程序集的共享程序集
- 创建使用共享程序集的应用程序
- 将共享程序集安装到全局程序集缓存(GAC)中

1. 创建强名称程序集的共享程序集

要创建强名称程序集, 首先必须生成一个公共-私有(public-private)密钥对。出于安全的目的, IT 产业中广泛使用公共-私有密钥对:

- 公司使用私钥来标记其应用程序, 然后将这些应用程序发布给用户。
- 用户必须验证应用程序是否来自于用户信任的公司。它们使用公司的公钥, 检查应用程序的数字签名。组织签名的惟一方法是应用程序是否利用用户所信任公司的私钥创建(当然, 该机制依赖于这样一个事实: 公司的私钥确实是私有的)。

利用公共-私有密钥对可以为程序集创建强名称。编译期间编译器使用私钥, 并将相应的公钥写入程序集清单中。如果需要了解关于公共-私有密钥对的理论, 请参阅 Visual Studio .NET 帮助文件中的“Keys, cryptography”。

不能使用 Visual Studio .NET 创建公共-私有密钥对, 必须使用 .NET Framework SDK 中的 Strong Name 工具。打开 .NET Framework 命令提示窗口, 输入下面的命令:

```
C:\> sn /k C:\classdesign\chapter08\MyKey.snk
```

上面的命令在 C:\MyKeyFolder 文件夹中创建了一个公共-私有密钥对文件 MyKey.snk。利用这个公共-私有密钥对就可以创建强名称的程序集。下载文件夹中包含一个简单示例 signed_assembly.cs, 如下所示:

```
using System;
```



```
namespace MyGlobalAssembly
{
    public class SignedMethod
    {
        public static void MyMethod()
        {
            Console.WriteLine("Hi");
        }
    }
}
```

项目中的文件 `MyGlobalAssembly` 还包含一个源文件 `AssemblyInfo.cs`。该文件定义了一组属性，由编译器在为程序集生成附加元数据时使用，例如版本号和公钥等属性：

```
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: AssemblyCompany("")]
[assembly: AssemblyProduct("")]
[assembly: AssemblyCopyright("")]
[assembly: AssemblyTrademark("")]

// Version information 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 '*'

[assembly: AssemblyVersion("1.0.0.0")]
[assembly: AssemblyDelaySign(false)]
[assembly: AssemblyKeyFile("C:/classdesign/chapter08/MyKey.snk")]
```

```
[assembly: AssemblyKeyName("")]
```

在该示例中, [assembly:AssemblyVersion(...)]用来通知编译器写入程序集中的版本号。而[assembly:AssemblyKeyFile(...)]则用来通知编译器需要使用哪一个公共-私有密钥对文件。这样, 编译器就可以利用这些信息生成强名称的程序集了。

在 Visual Studio .NET 菜单中建立项目, 把程序集创建为 DLL, 在 MSIL Disassembler 中可以查看程序集。在 MSIL Disassembler 窗口中, 双击 MANIFEST 节点可以查看程序集的清单。

注意, 清单应包括一个 publickey 属性, 该属性表示当前程序集是强名称的程序集。此外, ver 属性中包含了在 AssemblyInfo.cs 中指定的版本信息:

```
.assembly MyGlobalAssembly
{
    ...
    .publickey = (00 24 00 00 04 ... AC D4 D2 40 C1 )
    .hash algorithm 0x00008004
    .ver 1:0:0:0
}
```

publickey 属性是一个较长的数字。为了提高效率, Strong Name 工具还生成了一个公钥的缩写版本, 称作公钥令牌(public key token)。按照下面的方法运行 Strong Name 工具, 可以查看公钥令牌:

```
C:\> sn /t C:\classdesign\chapter08\MyKey.snk
```

显示如下的公钥令牌(很明显, 如果读者运行该命令, 应该得到不同的公钥令牌)。

```
Microsoft (R) .NET Framework Strong Name Utility Version 1.0.3705.0
Copyright (C) Microsoft Corporation 1998-2001. All rights reserved.
```

```
Public key token is 070bcd7fade41693
```

2. 创建使用共享程序集的应用程序

前面已经列举了使用强名称程序集的简单应用程序。本节首先说明如何利用 Visual Studio .NET 查看和生成该应用程序, 然后学习利用命令行查看和生成应用程序的方法。这个 Visual Studio 项目的例子叫做 MyApp.csproj, 在 MyApp 文件夹中, 本章其余的代码也在 MyApp 文件夹中。

应用程序包含一个简单的源文件 shared_assembly.cs, 该文件内容为:

```
using MyGlobalAssembly;
```



```
namespace MyApp
{
    class SharedAssembly
    {
        static void Main()
        {
            SignedMethod.MyMethod();
        }
    }
}
```

注意其中的 `Main` 调用 `SignedMethod.MyMethod`。类 `SignedMethod` 定义在 `MyGlobalAssembly` 命名空间中，因此源代码中使用了 `using` 语句来导入该命名空间。

然而，以上只是一部分内容。`MyGlobalAssembly` 命名空间定义在另一个程序集中，因此还必须通知编译器需要使用哪个程序集。在 Visual Studio .NET 中，在 Solution Explorer 中展开 References 项。References 列表包含了对 `MyGlobalAssembly` 的引用。右键单击该程序集名，选择快捷菜单中的 Properties。就会显示如图 8-6 所示的属性对话框。

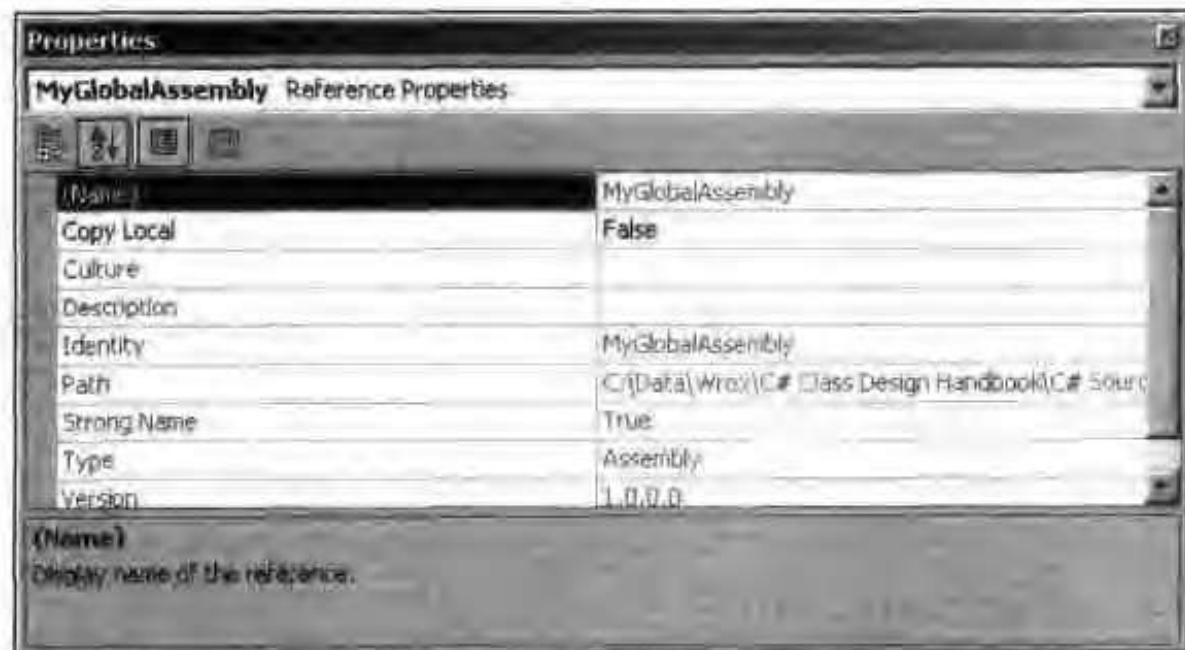


图 8-6

该对话框中指明了引用的程序集的下列信息：

- 程序集的名称为 `MyGlobalAssembly`。
- `Copy Local` 标记为 `False`。它通知编译器不需将指定的程序集复制到应用程序的输出文件夹中。这样，在运行应用程序时，将迫使 CLR 在 GAC 中查找程序集。

说明：

当添加程序集引用时，`Copy Local` 标记默认设置为 `True`。这样，编译器将指定的

程序集复制到应用程序的输出目录中。当运行应用程序时,CLR 将定位并加载程序集的副本,而不是定位和加载 GAC 中的程序集。为了避免这种行为,如果需要使用 GAC 版本,只要将程序集引用添加到项目中,就必须将 Copy Local 设置为 False。

- Path 属性在开发时,通知编译器程序集的当前位置。将该程序集引用添加到项目中时,指定该位置。
- 由于程序集要创建为强名称程序集,因此 Strong Name 属性为 True。
- Version 属性表示强名称程序集的版本号。

如果在 Visual Studio .NET 中生成该应用程序,刚才提到的程序集引用提供了编译器定位其他程序集所需的所有信息。另外,也可以通过命令行编译应用程序,在此例中,编译时必须指定/r 编译器选项,通知编译器有关程序集的信息:

```
C:\> csc /r:C:\classdesign\chapter08\MyGlobalAssembly.dll  
shared_assembly.cs
```

在运行应用程序之前,看起来一切都进展顺利。CLR 试图加载被引用的程序集。运行时在假设(不正确的假设)程序集为一个私有程序集的前提下,查看应用程序所在的文件夹。然而,程序集为一个共享程序集,它并不在该目录中。因此 CLR 抛出 System.IO.FileNotFoundException 异常,应用程序终止。

为了解决该问题,必须将共享程序集安装到 GAC 中。

3. 将共享程序集安装到 GAC 中

.NET Framework 提供了运行时工具 gacutil.exe,利用该工具可以在 GAC 中安装和删除程序集。打开 .NET Framework 命令提示窗口,将当前路径设为 DLL 文件所在的文件夹。输入下面的命令:

```
C:\> gacutil -i MyGlobalAssembly.dll
```

说明:

用户必须具有管理员权限才可将程序集安装到 GAC 中。

GAC 实现为 Windows 文件夹下的 Assembly 文件夹(例如, C:\Windows\Assembly),将程序集安装到 GAC 中时, gacutil 工具将程序集复制到 Assembly 文件夹下指定的子文件夹中。该工具基于程序集的名称、版本、公钥和文化,为程序集的子文件夹设计名称。这样消除了安装程序集时碰到与原有程序集具有相同文件名的可能性。

要检查程序集是否正确添加到 GAC 中,可以使用 .NET Framework Configuration 工具。打开控制面板,双击管理工具,然后双击 Microsoft .NET Framework Configuration。单击 Manage the Assembly Cache 超链接,打开 Assembly Cache 窗口。

在 Assembly Cache 窗口中单击 View List of Assemblies 超链接。显示 GAC 中的所

8.4.1 使用程序集级别的属性

前面的章节中，说明了如何创建共享程序集，并将其安装在 GAC 中。GAC 强调程序集是强名称的，这意味着程序集中必须包含公钥信息。正如前面所述，通过 [assembly: AssemblyKeyFile] 属性提供该公钥。例如：

```
[assembly: AssemblyKeyFile("C:/classdesign/chapter08/MyKey.snk")]
```

除了为公共-私有密钥对文件指定文件名之外，还可以指定如下所示的附加信息：

- 程序集的自由格式标题
- 程序集的描述
- 公司名
- 产品名信息
- 版权信息
- 商标

下例说明了如何定义这些信息。本例的源代码保存在下载文件夹 Documentation 中。文档信息定义在 AssemblyInfo.cs 文件中：

```
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("My Assembly")]
[assembly: AssemblyDescription("My cool assembly")]
[assembly: AssemblyCompany("My Company Inc.")]
[assembly: AssemblyProduct("My Product")]
[assembly: AssemblyCopyright("(c) My Company, 2002")]
[assembly: AssemblyTrademark("TM My Company")]

//
// Version information for an assembly consists of 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 '*' notation
```

```
[assembly: AssemblyVersion("1.0.*")]
```

在本书下载代码的 Documentation 文件夹中，使用命令行编译器建立这个应用程序。输入下面的命令：

```
C:\Documentation>csc /out:Documentation.exe *.cs
```

可以在 Windows 资源管理器中查看已编译程序集的属性。在 Windows 资源管理器中选择程序集文件，选择快捷菜单中的 Properties。例如，在 Windows 2000 中，Properties 窗口中显示如图 8-8 所示的信息。

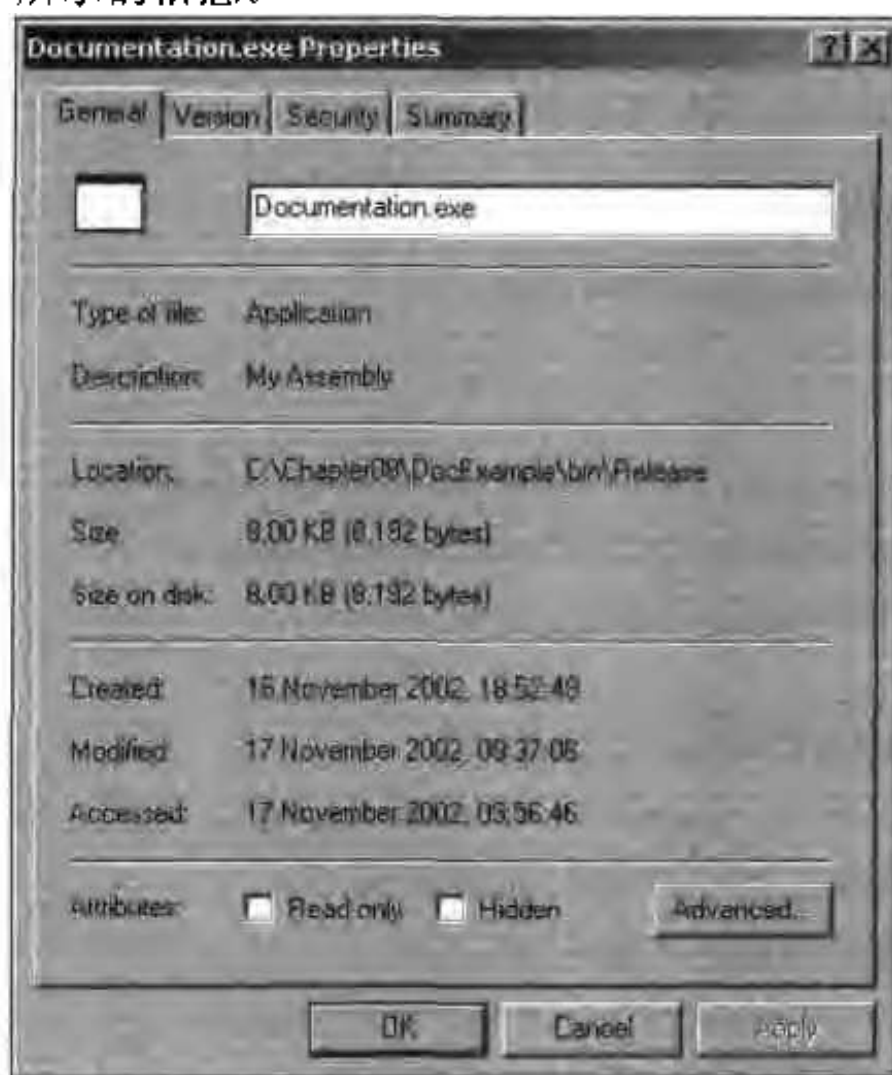


图 8-8

单击对话框中的 Version 选项卡，显示程序集的详细信息。例如，Comments 字段如图 8-9 所示。

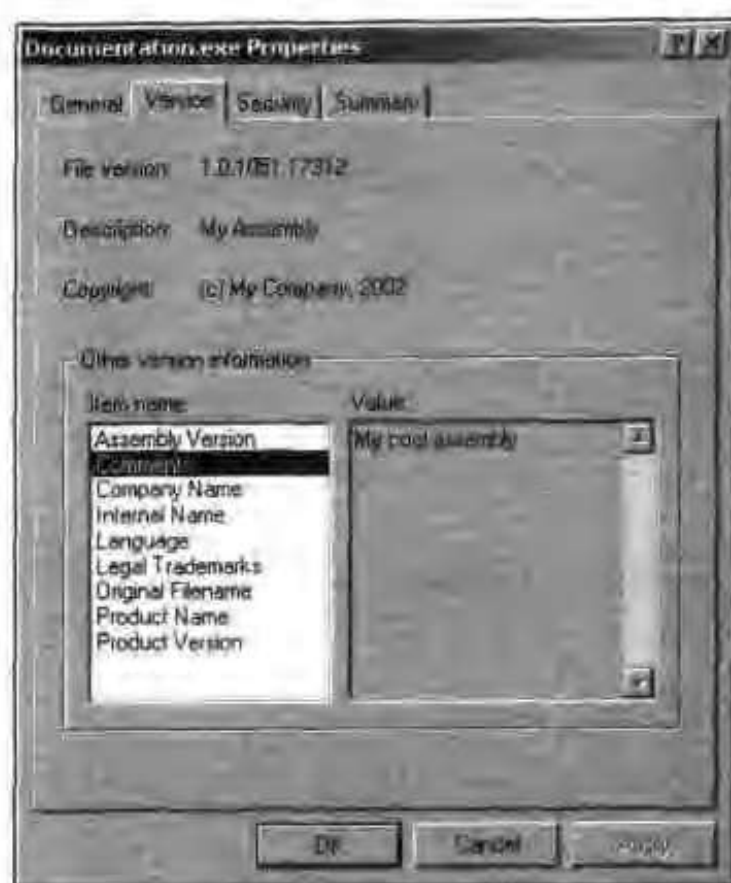


图 8-9

8.4.2 XML 文档

Visual Studio .NET 的 C#编译器提供了一个惟一的命令行选项/doc, 用于在编译代码时生成额外的输出文件。该输出文件是 XML 格式, 包含从源代码中收集来的一些注释。为此, 注释编写采用特殊的格式。在 C#中, 该格式就是使用三个连续的斜杠符号"/", 其后是 XML 标记名称。例如, 在文件 books.cs 定义的类型上添加几个这样的注释:

```
using System.Collections;

namespace Books
{
    /// <summary>
    /// Inventory is a singleton class. It maintains a list of
    /// Books, indexed by ISBN.
    /// </summary>
    /// <remarks>
    /// Use the static property Instance to obtain a reference to
    /// the singleton. Then use the factory method CreateBook
    /// to create and add individual Books.
    /// </remarks>
    public class Inventory
```



```
{  
    /// <summary>  
    /// Private member field containing a reference to the  
    /// singleton instance.  
    /// </summary>  
    private static Inventory mInstance;  
  
    /// <summary>  
    /// Hashtable containing a collection of Book objects.  
    /// </summary>  
    private Hashtable mLibraryBooks;  
  
    // and so on...  
}
```

上面的代码要注意以下几点：

- 在要引用的类型或成员之前添加///注释。
- 如果在 Visual Studio .NET 中输入///, 就会自动添加<summary></summary>标记。

要查看这些注释生成的 XML 文档文件, 可以使用 C#命令行编译器和/doc 选项编译代码, 或配置 Visual Studio .NET 来编译代码。要调用命令行编译器, 可输入下面的命令:

```
C:\> csc /t:library /doc:documentation.xml books.cs
```

books.cs 只包含类型定义, 所以使用/t 选项把它编译为一个库。/doc 选项需要使用一个文件名来编写生成的 XML。在上面的例子中, 指定了文件 documentation.xml。

另外, 也可以在 Visual Studio .NET 中, 右击 Solution Explorer 中的项目, 从菜单中选择 Properties, 再选择 Configuration Properties, 给 XML Documentation File 选项输入 documentation.xml。

在这两种情况下, 编译项目时, 都会在文件 documentation.xml 中生成下面的 XML:

```
<?xml version="1.0"?>  
<doc>  
  <assembly>  
    <name>DocExample</name>  
  </assembly>  
  <members>  
    <member name="T:Books.Inventory">
```

```
<summary>
Inventory is a singleton class. It maintains a list of
Books, indexed by ISBN.
</summary>
<remarks>
Use the static property Instance to obtain a reference to
the singleton. Then use the factory method CreateBook
to create and add individual Books.
</remarks>
</member>
<member name="F:Books.Inventory.mInstance">
<summary>
Private member field containing a reference to the
singleton instance.
</summary>
</member>
<member name="F:Books.Inventory.mLibraryBooks">
<summary>
Hashtable containing a collection of Book objects.
</summary>
</member>
<!-- etc. -->
</members>
</doc>
```

我们输入的<summary> 和 <remarks>标记被提取出来, 和它们引用的 C#类型放在一起。

Visual Studio .NET 的文档定义了许多 XML 标记, IDE 可以理解这些标记, IntelliSense 也会提示它们。另外, <summary>元素中的注释实际上由 IntelliSense 用于在开发过程中描述类中的字段、属性和方法。

在///注释中, 不仅可以使使用由 Visual Studio .NET 注释的标记, 还可以使用任何有效的 XML 标记, 编译器会正确提取这些标记。因此, 可以生成任何格式的 XML, 使用合适的 XSLT 样式表就可以把它们处理为可读的文档。

实际上, Visual Studio .NET 也可以处理, 但在编写本书时, IDE 只能使用 Tools 菜单中的 Build Comment Web Pages 选项正确处理<summary> 和 <remarks>标记, 生成 HTML 格式的文档。使用这个选项和注释过的源代码, 可以生成如图 8-10 所示的 HTML 文档。

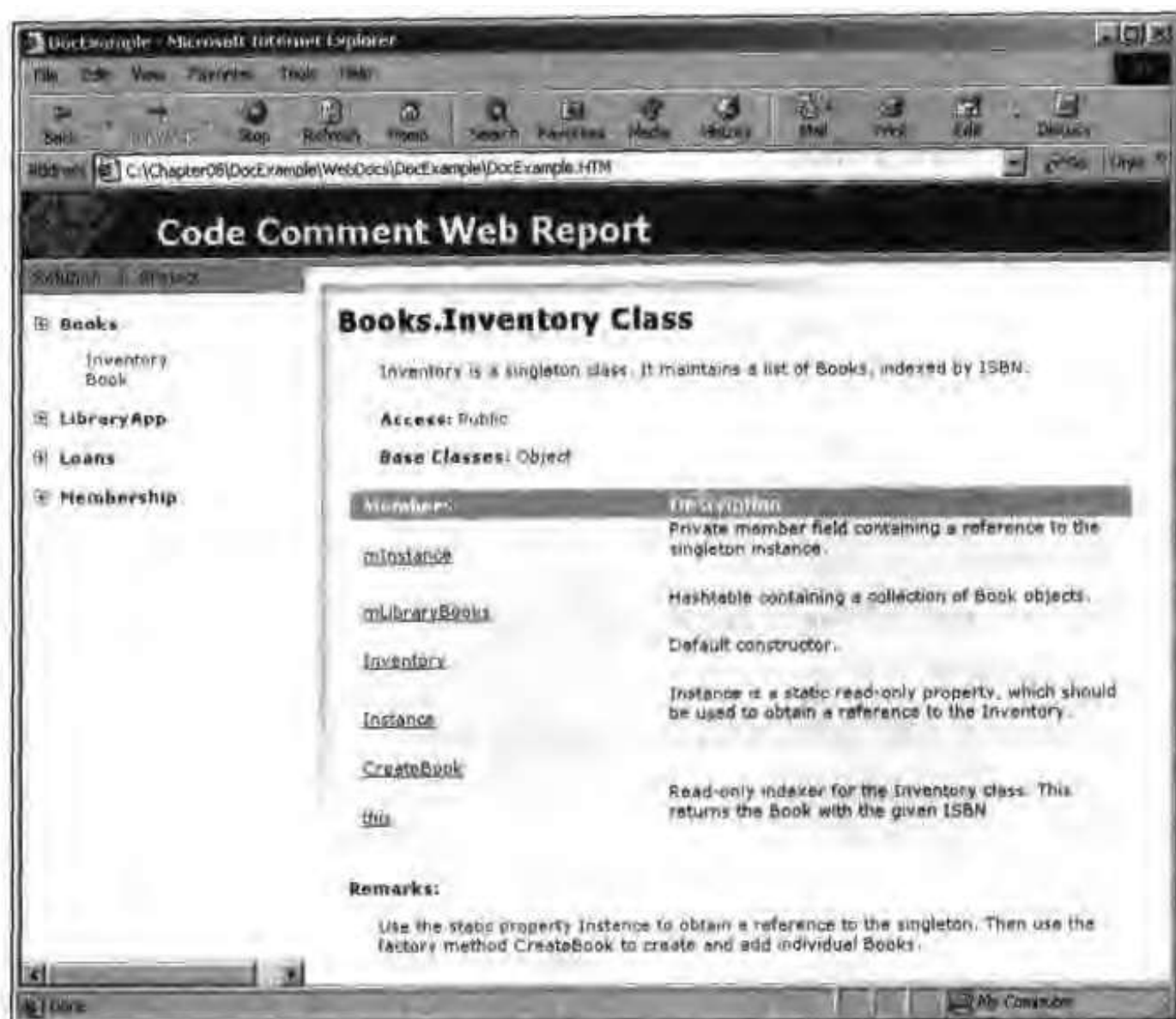


图 8-10

与手工生成文档相比，这是一个巨大的进步，其优点是，该功能是 IDE 的一个组成部分。

Visual Studio .NET 的当前版本不支持所有的 XML 标记。但是，有一些第三方工具可以处理生成的 XML，并创建更完整的文档。其中一种工具就是 NDoc，它可以根据上面的 XML 生成文档。NDoc 可以从 <http://ndoc.sourceforge.net> 下载。下面快速介绍一下这个软件：

- 它是免费的。
- 它是一个开放源代码的产品，可以根据需要进行配置。
- 它完全用 C#编写，使用 XML 标记来生成文档。
- 文档可以用已编译的 HTML 格式(CHM)生成，可以像 MSDN 文档那样可视化地设计。

另一个优点是 NDoc 使用反射机制遍历程序集中定义类层次结构，还可以为继承的成员生成文档。其结果是相等令人满意的。例如，可以加载 NDoc，选择上面使用的 Visual Studio .NET 项目 DocExample。NDoc 会识别解决方案生成的程序集，并提供一些选项，修改已生成文档的格式。见图 8-11。

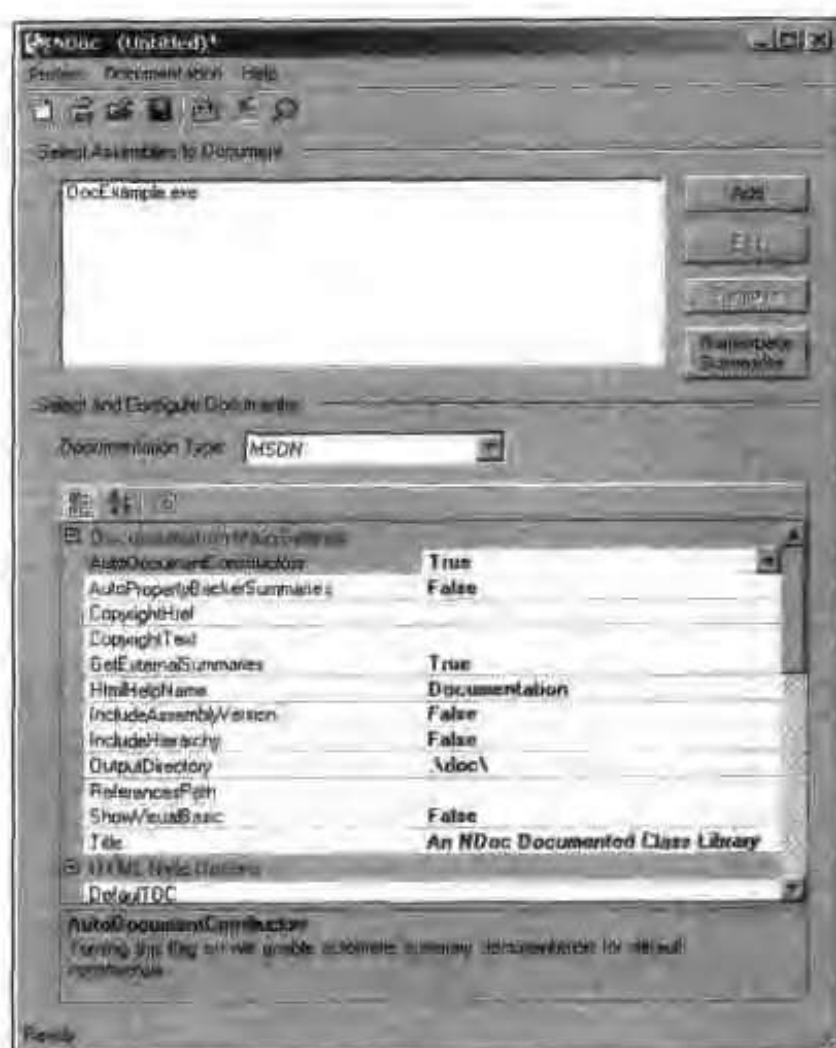


图 8-11

单击几下鼠标，就可以以已编译的 HTML 格式生成文档文件。NDoc 还会为在指定程序集中定义的所有类型遍历其类层次结构，采用类似于 .NET Framework 在 MSDN 文档中的格式建立文档。见图 8-12。

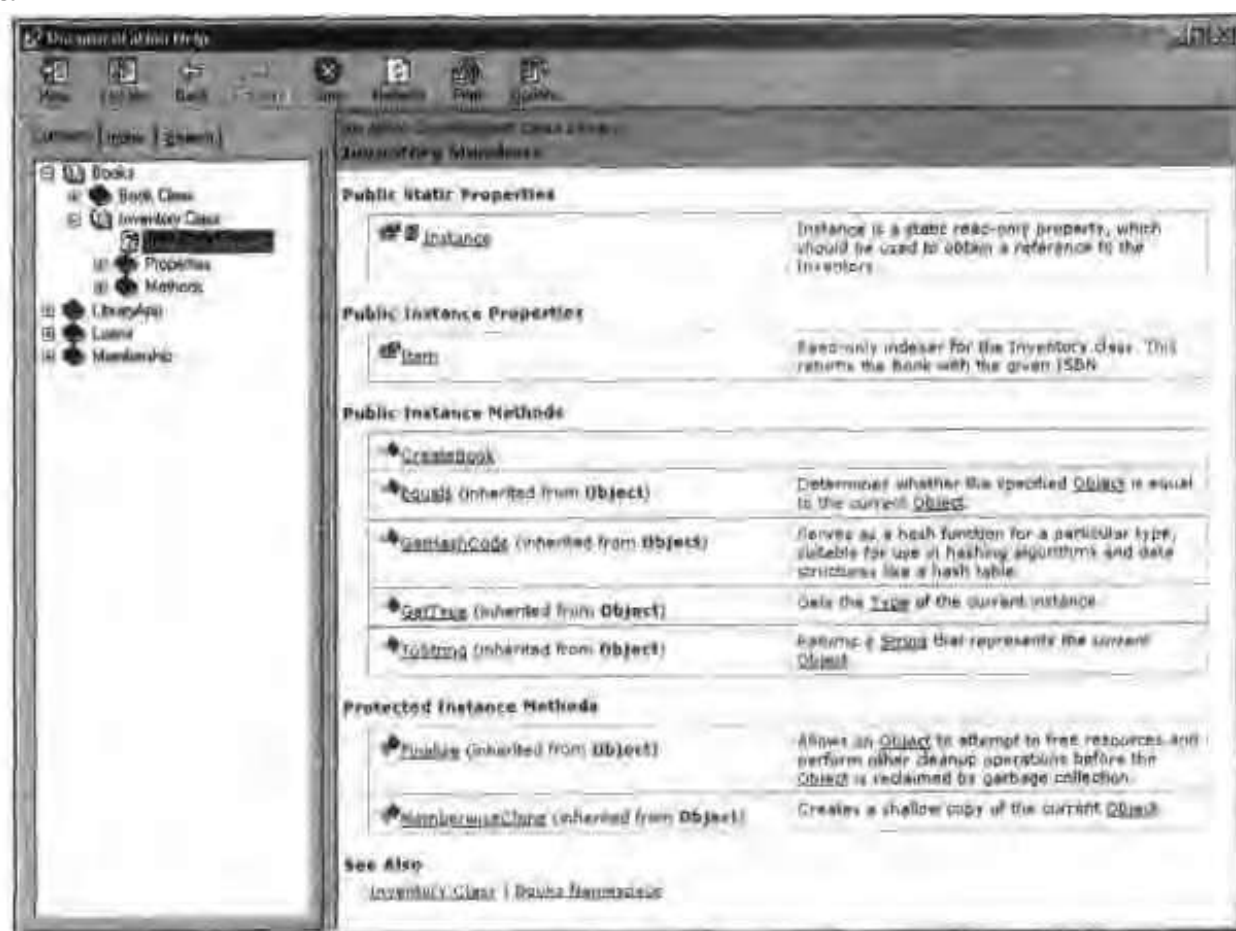


图 8-12

8.5 小结

在本书最后一章中，我们学习了在 C#应用程序中如何控制代码的逻辑和物理组织。具体讲解了以下内容：

- 如何利用命名空间将应用程序分割成相关类型的逻辑组，在 UML 对象模型中如何将它们与包相关联。
- 如何使用程序集和模块来将应用程序分割成可工作的、可重用的、惟一命名的区域，这些区域可以用多种 .NET 语言编写。
- 如何将应用程序编译和部署为独立的程序集，包括使用 DLL、EXE 和模块。
- 如何使用 GAC 创建可由 CLR 轻松定位的共享程序集，GAC 如何排除使用 Windows 注册表时的 DLL Hell。
- 如何把文档内置于编译的程序集中。
- 如何使用 XML 标记产生和维护具有专业外观的文档。

[G e n e r a l I n f o r m a t i o n]

书名= C #类设计手册

作者=

页数= 3 1 8

S S 号= 1 1 1 4 5 9 9 1

出版日期=

封面
书名
版权
前言
目录
正文