

Join the discussion @ p2p.wrox.com



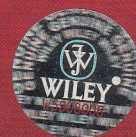
Wrox Programmer to Programmer™



Professional C# 4 and .NET 4

C#高级编程

(第7版)



(美) Christian Nagel

Bill Evjen

等著

Jay Glynn

李 铭

译

黄 静

审校



本书源代码与
第48~57章(电子书)

清华大学出版社

C#高级编程

(第7版)

Christian Nagel
(美) Bill Evjen 等著
Jay Glynn
李 铭 译
黄 静 审校

清华大学出版社

北 京

Christian Nagel, Bill Evjen, Jay Glynn, et al

Professional C# 4 and .NET 4

EISBN: 978-0-470-50225-9

Copyright © 2010 by Wiley Publishing, Inc.

All Rights Reserved. This translation published under license.

本书中文简体字版由 Wiley Publishing, Inc. 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。

北京市版权局著作权合同登记号 图字: 01-2010-2103

本书封面贴有 Wiley 公司防伪标签, 无标签者不得销售。

版权所有, 侵权必究。侵权举报电话: 010-62782989 13701121933

图书在版编目(CIP)数据

C#高级编程(第7版)/(美)内格尔(Nagel, C.), (美)埃夫琴(Evjen, B.)等著; 李铭 译; 黄静 审校.

—北京: 清华大学出版社, 2010.11

书名原文: Professional C# 4 and .NET 4

ISBN 978-7-302-23937-6

I. C… II. ①内… ②埃… ③李… ④黄… III. C 语言—程序设计 IV. TP312

中国版本图书馆 CIP 数据核字(2010)第 195700 号

责任编辑: 王 军 谢晓芳

装帧设计: 孔祥丰

责任校对: 成凤进

责任印制: 王秀菊

出版发行: 清华大学出版社

地 址: 北京清华大学学研大厦 A 座

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

邮 编: 100084

社 总 机: 010-62770175

邮 购: 010-62786544

投稿与读者服务: 010-62776969, c-service@tup.tsinghua.edu.cn

质 量 反 馈: 010-62772015, zhiliang@tup.tsinghua.edu.cn

印 刷 者: 清华大学印刷厂

装 订 者: 三河市新茂装订有限公司

经 销: 全国新华书店

开 本: 185×260 印 张: 94.25 字 数: 2529 千字

附光盘 1 张

版 次: 2010 年 11 月第 1 版 印 次: 2010 年 11 月第 1 次印刷

印 数: 1~5000

定 价: 148.00 元

产品编号: 035929-01

作者简介

CHRISTIAN NAGEL 是 Microsoft 区域董事、Microsoft MVP, thinktecture 的合作伙伴, CN 革新技术的拥有者, 他是一位软件架构师和开发人员, 为开发 Microsoft .NET 解决方案提供培训和咨询服务。他具备超过 25 年的软件开发经验。Christian 从 PDP 11 和 VAX/VMS 系统开始其计算机生涯, 熟悉各种语言和平台。自从 2000 年以来, (那时 .NET 还只是一个技术框架) 他就开始使用各种 .NET 技术构建大量 .NET 解决方案。他具备 Microsoft 技术的深厚功底, 编写了大量 .NET 图书, 并获得了 Microsoft 认证培训师和专业开发人员证书。Christian 在国际会议发表演讲(如 TechEd 和 Tech Days)并创立 INETA Europe, 以支持 .NET 用户组。通过 Web 站点 www.cninnovation.com 和 www.thinktecture.com 可以联系 Christian, 在 www.twitter.com/christiannagel 上可以找到他。

BILL EVJEN 是 .NET 技术和基于社团的 .NET 学习活动的积极倡导者, 他自从 .NET 在 2000 年第一次发布以来就积极涉足 .NET。同年, Bill 成立了 St. Louis .NET 用户组(www.stlnet.org), 这是世界上第一个这样的用户组, Bill 还是国际 .NET 协会(www.ineta.org)的奠基人和前任执行主管, 该协会在全世界有 500 000 多位成员。

Bill 住在密苏里州的圣路易斯, 是 ASP.NET 和 Web 服务的一位著名作者和演说家。他编写或与他人合作编写了 20 多本书, 包括 *Professional ASP.NET 4*、*Professional VB 2008*、*ASP.NET Professional Secrets*、*XML Web Services for ASP.NET* 和 *Web Services Enhancements: Understanding the WSE for Enterprise Applications* (均由 Wiley 出版)。除了写作之外, Bill 还在许多会议上发表演讲, 包括 DevConnections、VSLive 和 TechEd。除了这些活动之外, Bill 还与 Microsoft 联系紧密, 是 Microsoft 区域董事和 MVP。

Bill 是国际新闻及财务服务公司(www.thomsonreuters.com) Thomson Reuters, 即 Lipper 的全球平台架构师。他毕业于华盛顿州贝灵翰姆的西华盛顿大学, 获得了俄语学位。他闲暇时通常在芬兰的 Toivakka 度假。在 Twitter 网站上可以通过 @billevjén 联系到 Bill。

JAY GLYNN 是 PureSafety 的首席构架师, PureSafety 是一家为劳动力的安全和健康提供结果驱动的软件和信息解决方案的业界领先的提供商。Jay 开发软件的时间有 25 余年, 使用过各种语言和技术, 包括 PICK Basic、C、C++、Visual Basic、C# 和 Java。Jay 目前与妻儿住在田纳西州的富兰克林。

KARLI WATSON 是 Infusion Development (www.infusion.com) 的顾问, Boost.net (www.boost.net) 的技术架构师和 IT 自由撰稿专业人士、作家和开发人员。他主攻 .NET (尤其是 C# 和后来的 WPF), 在这个领域编写了许多图书。他擅长以通俗易懂的方式激情澎湃地阐述复杂的理念, 并花了大量的时间研究新技术, 寻求可教给其他人的新技术。

在不工作时(这种时间似乎没有), Karli 希望到山上滑雪, 或者尝试发表他的小说。他喜欢穿颜色鲜亮的衣服, 可以在 www.twitter.com/karlequin 网站上找到他, 也许有一天他自己会建立一个网站。

MORGAN SKINNER 年轻时对 Sinclair ZX80 很感兴趣, 在校期间就开始了计算机生涯, 当时他对教师编写的一些代码不感兴趣, 便开始用汇编语言编程。从此以后他使用各种语言和平台, 包括 VAX 宏汇编程序、Pascal、Modula2、Smalltalk、X86 汇编语言、PowerBuilder、C/C++、VB 和目前的 C#, 自从 2000 年发布 PDC 以来, 他就用 .NET 编程, 而且非常喜欢 .NET, 于是在 2001 年加入了 Microsoft。他现在是开发人员的主要支持人员, 而且花了大多数时间帮助客户使用 C#。在 www.morganskinner.com 上可以联系到 Morgan。

前言

对于开发人员,把 C#语言及其相关联的 .NET Framework 环境描述为多年来最重要的新技术一点都不夸张。 .NET 提供了一种环境。在这个环境中,可以开发在 Windows 上运行的几乎所有应用程序,而 C#是专门用于 .NET Framework 的编程语言。例如,使用 C#可以编写动态 Web 页面、Windows Presentation Foundation 应用程序、XML Web 服务、分布式应用程序的组件、数据库访问组件、传统的 Windows 桌面应用程序,甚或可以联机/脱机运行的新型智能客户端应用程序。本书介绍 .NET Framework 4。如果读者使用以前的版本编码,本书的一些章节就不适用。本书将标注出专用于 .NET Framework 4 的新增内容。

不要被这个 Framework 名称中的 .NET 所迷惑,认为这是一个只关注 Internet 的架构。这个名称中的 .NET 仅强调 Microsoft 相信分布式应用程序是未来的趋势,即处理过程分布在客户端和服务器的上。理解 C#不仅仅是编写 Internet 或与网络能识别的应用程序的一种语言也很重要。它还提供了一种编写 Windows 平台上几乎任何类型的软件或组件的方式。另外, C#和 .NET 都对开发人员编写程序的方式进行了革新,更易于实现在 Windows 上的编程。

那么, .NET 和 C#有什么优点?

.NET 和 C#的重要性

为了理解 .NET 的重要性,了解一下过去 18 年来出现的许多 Windows 技术的本质,会有一定的帮助。尽管所有 Windows 操作系统在表面上看来完全不同,但从 Windows 3.1(1992 年引入)到 Windows 7 和 Windows Server 2008 R2,在内核上都有相同的 Windows API。在我们转而使用 Windows 的新版本时,虽然 API 中增加了非常多的新功能,但这是一个演化和扩展 API 的过程,并非替换它。

开发 Windows 软件所使用的许多技术和架构也是这样。例如,组件对象模型(Component Object Model, COM)源自对象链接和嵌入(Object Linking and Embedding, OLE)。最初,因为它在很大程度上仅把不同类型的 Office 文档链接在一起,所以利用它,例如,可以把一个小型 Excel 电子表格放在 Word 文档中。之后,它逐步演化为 COM、DCOM(Distributed COM, 分布式组件对象模型)和最终的 COM+。COM+是一种复杂的技术,它是几乎所有组件通信方式的基础,实现了事务处理、消息传输服务和对象池。

Microsoft 选择这种革新方法的原因非常明显:它关注后向兼容性。在过去的这些年中,第三方软件编写了大量 Windows 软件,如果 Microsoft 每次都引入一项不遵循现有基本代码的新技术,Windows 就不会获得今天的成功。

后向兼容性是 Windows 技术的极其重要的功能,也是 Windows 平台的一个长处。但它有一个很大的缺点:每次某项技术更新换代,增加了新功能后,它都会比它以前更复杂。

很明显,对此必须进行改进。Microsoft 不可能一直扩展相同的开发工具和语言,总是使它们越

来越复杂,既要保证能跟上最新硬件的发展步伐,又要与20世纪90年代初开始流行的Windows产品向后兼容。如果要得到一系列简单而专业的语言、环境和开发工具,让开发人员轻松地编写最新的软件,就需要一个新的开端。

这就是C#和.NET的作用。粗略地说,.NET是一种在Windows平台上编程的架构——一种API。C#是一种从头开始设计的用于.NET的语言,它可以利用.NET Framework及其开发环境中的所有新增功能,以及在最近25年来出现的面向对象的编程方法。

在继续介绍前,必须先说明,后向兼容性并没有在这个演化进程中丧失。现有的程序仍可以使用,.NET也兼容现有的软件。现在,在Windows上软件组件之间的通信几乎都使用COM实现。因此,.NET能够提供现有COM组件的包装器(wrapper),以便.NET组件与之通信。

我们不需要学习了C#才能给.NET编写代码,因为Microsoft已经扩展了C++,还对Visual Basic进行了很多改进,把它转变成了功能更强大的语言,并允许把用这些语言编写的代码用于.NET环境。但其他这些语言都因有多年演化的遗留痕迹,并非一开始就用现在的技术来编写,导致它们不能用于.NET环境。

本书将介绍C#编程技术,同时提供.NET体系结构工作原理的必要背景知识。我们不仅会介绍C#语言的基础,还会给出使用各种相关技术的应用程序对应的示例,包括数据库访问、动态的Web页面、高级的图形和目录访问等。

.NET 的优点

前面阐述了.NET的优点,但并没有说它会使开发人员的工作更易完成。本节将简要讨论.NET的改进功能。

- **面向对象编程:** .NET Framework和C#从一开始就完全基于面向对象的原则。
- **优秀的设计:** 一个基类库,它以一种非常直观的方式设计出来。
- **语言无关性:** 在.NET中,Visual Basic、C#和托管C++等语言都可以编译为通用的中间语言(Intermediate Language)。这说明,语言可以用以前没有的方式交互操作。
- **对动态Web页面更好的支持:** 虽然ASP具有很大的灵活性,但效率不是很高,这是因为它使用了解释性的脚本语言,且缺乏面向对象的设计,从而导致ASP代码比较混乱。.NET使用ASP.NET,为Web页面提供了一种集成支持。使用ASP.NET,可以编译页面中的代码,这些代码还可以使用.NET能识别的高级语言来编写,如C#或Visual Basic 2010。.NET现在还添加了对最新Web技术的重要支持,如Ajax和jQuery。
- **高效的数据访问:** 一组.NET组件,统称为ADO.NET,提供了对关系数据库和各种数据源的高效访问。这些组件也可用于访问文件系统和目录。尤其是,.NET内置了XML支持,可以处理从非Windows平台导入或导出的数据。
- **代码共享:** .NET引入了程序集的概念,替代了传统的DLL,可以完美无暇地改进代码在应用程序之间的共享方式。程序集是解决版本冲突的正式设备,程序集的不同版本可以并存。
- **增强的安全性:** 每个程序集还可以包含内置的安全信息,这些信息可以准确地指出谁或哪种类型的用户或进程可以调用什么类的哪些方法。这样就可以非常准确地控制用户部署的程序集的使用方式。
- **对安装没有任何影响:** 有两种类型的程序集,分别是共享程序集和私有程序集。共享程序集是可用于所有软件的公共库,而私有程序集只用于特殊软件。由于私有程序集完全自包

含，所以安装过程非常简单。没有注册表项，只需把相应的文件放在文件系统的相应文件夹中即可。

- **Web 服务的支持：**.NET 完全集成了对开发 Web 服务的支持，用户可以轻松地开发任何类型的应用程序。
- **Visual Studio 2010：**.NET 附带了一个 Visual Studio 2010 开发环境，它同样可以很好地利用 C++、C#、Visual Basic 2010 和 ASP.NET 或 XML 进行编码。Visual Studio 2010 集成了这个 IDE 所有以前版本中的各种语言专用的环境中的所有最佳功能。
- **C#：**是使用 .NET 的一种面向对象的强大且流行的语言。

第 1 章将详细讨论 .NET 体系结构的优点。

.NET Framework 4 中的新增特性

.NET Framework 的第 1 版(1.0 版)在 2002 年发布，赢得了许多人的喝彩。.NET Framework 2.0 在 2005 年发布，认为它是该架构的一个主要版本。.NET Framework 4 是该产品另一个重要的版本，包含了许多重要的新功能。

对于 .NET Framework 的每个版本，Microsoft 总是试图确保对已开发出的代码进行尽可能少的不兼容的更改。到目前为止，Microsoft 在这方面做得很成功。

下面将详细描述 C# 2010 和 .NET Framework 4 中的一些新变化。

动态类型

编程界在动态语言(如 JavaScript、Python 和 Ruby)方面的进步非常快。由于这类编程越来越流行，Microsoft 在 C# 中发布了一个新的动态类型功能。并不总是可以以静态方式确知对象最终是什么类型。现在不使用 object 关键字和从这个类型生成的所有对象，而可以让动态语言运行库(Dynamic Language Runtime, DLR)在运行期间动态地确定对象的类型。

使用 C# 新增的动态功能，可以更好地进行交互操作。我们可以与各种动态语言交互操作，更容易地使用 DOM。甚至现在使用 Microsoft Office COM API 也更容易。

在 .NET Framework 4 这个版本中，Microsoft 包含了动态语言运行库。DLR 建立在公共语言运行库(Common Language Runtime, CLR)的基础上，提供了把所有动态语言交互操作连接起来的功能。

C# 使用新的 dynamic 关键字访问新的 DLR。这对于编译器是一个标记，只要遇到这个关键字，编译器就认为它是一个动态调用，而不是一般的静态调用。

可选参数和命名参数

虽然可选参数和命名参数在 Visual Basic 中已存在了一段时间了，但在 .NET 4 发布之前，它们不能在 C# 中使用。可选参数允许为方法的一些参数提供默认值，并允许使用者重载类型，因此，即使只有一个方法，也能处理所有变体。下面是一个例子：

```
public void CreateUser(string firstname, string lastname,
    bool isAdmin, bool isTrialUser)
{
}
```

如果要重载这个方法,并为两个 `bool` 对象提供默认值,就很容易得到好几个方法,为使用者填充这些值,然后调用主方法,以实际创建用户。现在通过可选参数,就可以编写下面的代码:

```
public void CreateUser(string firstname, string lastname,
    bool isAdmin = false, bool isTrialUser = true)
{
}
```

查看这段代码, `firstname` 和 `lastname` 参数没有设置默认值,而 `isAdmin` 和 `isTrialUser` 参数设置了默认值。使用者现在可以编写如下代码:

```
myClass.CreateUser("Bill", "Evjen");
myClass.CreateUser("Bill", "Evjen", true);
myClass.CreateUser("Bill", "Evjen", true, false);
myClass.CreateUser("Bill", "Evjen", isTrialUser: false);
```

上一个例子使用了命名参数,这也是在 .NET Framework 的这个版本中 C# 的一个新功能。命名参数会潜在地改变编写代码的方式。这个新功能能使代码更容易阅读和理解。例如,看一下 `System.IO` 名称空间中的 `File.Copy()` 方法,它一般构建为:

```
File.Copy(@"C:\myTestFile.txt", @"C:\myOtherFile.txt", true);
```

在这行代码中,这个简单的方法使用 3 个参数,但实际传递给 `Copy()` 方法的是什麼内容?除非知道这个方法的前前后后,否则仅看一眼该方法,很难判断出该方法会执行何种操作。而通过命名参数,就可以在提供参数值之前使用代码中的参数名,如下面的示例所示:

```
File.Copy(sourceFileName: @"C:\myTestFile.txt",
    destFileName: @"C:\myOtherFile.txt", overwrite: true);
```

现在通过命名参数,就很容易阅读和理解这行代码将执行的操作。使用命名参数对最终的编译没有影响,命名参数仅用在应用程序的编码中。

协变和抗变

虽然在 .NET Framework 的以前版本中包含协变和抗变,但它们在 .NET 4 中进行了扩展,当处理泛型、委托等时,它们会执行得更好。例如,在 .NET 的以前版本中,可以对对象和数组使用抗变,但不能对泛型接口使用抗变。而在 .NET 4 中,就可以对泛型接口使用抗变。

ASP.NET MVC

ASP.NET MVC 是 ASP.NET 最新的主要新增内容,它为开发团队带来了许多惊喜。ASP.NET MVC 提供了许多开发人员期待的、使用模型-视图-控制器来创建 ASP.NET 的方式。ASP.NET MVC 在开发人员构建的应用程序中提供了可测试性、灵活性和可维护性。记住,ASP.NET MVC 不是每个人都知悉和喜欢的 ASP.NET 的替代品,而只是构建应用程序的另一种方式。

ASP.NET 的这个版本允许使用这个新模型构建应用程序,它完全内嵌在 Framework 和 Visual Studio 中。

C#的优点

C#在某种程度上可以看作是.NET 面向 Windows 环境的一种编程语言。在过去的十几年中, Microsoft 给 Windows 和 Windows API 添加了许多功能, Visual Basic 2010 和 C++也进行了许多扩展。虽然 Visual Basic 和 C++最终已成为非常强大的语言, 但这两种语言也存在问题, 因为它们保留了原来的一些遗留内容。

对于 Visual Basic 6 及其早期版本, 它的主要优点是很容易理解, 许多编程工作都很容易完成, 从很大程度上对开发人员隐藏了 Windows API 和 COM 组件结构的详细信息。其缺点是因为 Visual Basic 从来没有实现真正意义上的面向对象, 所以大型应用程序很难分解和维护。另外, 因为 Visual Basic 的语法继承自 BASIC 的早期版本(BASIC 主要是为了让刚入门的程序员更容易理解, 而不是为了编写大型商业应用程序), 所以不能真正成为结构良好或面向对象的编程语言。

另一方面, C++基于 ANSI C++语言定义。它与 ANSI 不完全兼容, 因为 Microsoft 在 ANSI 定义标准化之前编写其 C++编译器, 但它已经相当接近。但是, 这导致了两个问题。首先, ANSI C++是在十几年前的技术条件下开发的, 因此它不支持现在的概念(如 Unicode 字符串和生成 XML 文档), 某些古老的语法结构是为以前的编译器设计的(如成员函数的声明和定义是分开的)。其次, Microsoft 同时还试图把 C++演变为一种用于在 Windows 上执行高性能任务的语言, 为此不得不在语言中添加大量 Microsoft 专用的关键字和各种库。其结果是在 Windows 上, 该语言非常杂乱。让 C++开发人员描述字符串有多少种定义就可以证明这一点: `char*`、`LPTSTR`、`string`、`CString`(MFC 版本)、`CString`(WTL 版本)、`wchar_t*`、`OLECHAR*`等。

现在进入.NET 时代——一种全新的环境, 它对这两种语言都进行了新的扩展。Microsoft 给 C++添加了许多 Microsoft 专用的关键字, 并把 Visual Basic 演变为 Visual Basic 2010, 保留了一些基本的 Visual Basic 语法, 但在设计上完全不同于原始 Visual Basic, 从实际应用的角度来看, Visual Basic 2010 是一种新语言。

在这里, Microsoft 决定给开发人员提供另一个选择——专门用于.NET、具有新起点的一种语言, 即 C#。Microsoft 在正式场合把 C#描述为一种简单、现代、面向对象、类型非常安全、派生自 C 和 C++的编程语言。大多数独立的评论员对 C#的描述改为“派生自 C、C++和 Java”。这种描述在技术上非常准确, 但没有表达出该语言的真正优点。从语法上看, C#非常类似于 C++和 Java, 许多关键字都相同, C#也使用类似于 C++和 Java 的块结构, 并用花括号({})来标记代码块, 用分号分隔各行语句。对 C#代码的第一印象是它非常类似于 C++或 Java 代码。但在这些表面的类似性后面, C#学习起来要比 C++容易得多, 但比 Java 难一些。其设计比其他语言更适合现代开发工具, 它同时具有 Visual Basic 的易用性, 以及 C++的高性能、低级内存访问。C#包括以下一些功能:

- 完全支持类和面向对象编程, 包括接口和实现继承、虚函数和运算符重载。
- 一致且定义完善的基本类型集。
- 对自动生成 XML 文档的内置支持。
- 自动清理动态分配的内存。
- 可以用用户定义的属性来标记类或方法。这可以用于文档, 对编译有一定的影响(例如, 把方法标记为只在调试版本中编译)。
- 可以完全访问.NET 基类库, 并易于访问 Windows API(如果实际需要它, 这就不常见)。

- 可以使用指针和直接访问内存,但C#语言可以在没有它们的条件下访问内存。
- 以 Visual Basic 的风格支持属性和事件。
- 改变编译器选项,可以把程序编译为可执行文件或.NET 组件库,该组件库可以用与 ActiveX 控件(COM 组件)相同的方式由其他代码调用。
- C#可以用于编写 ASP.NET 动态 Web 页面和 XML Web 服务。

应该指出,对于上述大多数功能,Visual Basic 2010 和 Managed C++也具备。事实上,虽然C#从一开始就使用.NET,但对.NET 功能的支持不仅更完整,而且在比其他语言更合适的语法环境中提供了这些功能。C#语言本身非常类似于Java,但其中有一些改进,尤其是,Java并不应用于.NET 环境。

在结束这个主题前,还要指出C#的两个局限性。一方面是该语言不适用于编写时间紧迫或性能非常高的代码,例如一个要占用1000或1050个机器周期的循环,并在不需要这些资源时,立即清理它们。在这方面,C++可能仍是所有低级语言中的佼佼者。另一方面是C#缺乏性能极高的应用程序所需要的关键功能,包括能够指定那些保证在代码的特定地方运行的内联函数和析构函数。但这类应用程序非常少。

编写和运行C#代码的环境

.NET Framework 4 运行在 Windows XP/2003/7 和最新的 Windows Server 2008 R2 上。要使用.NET 编写代码,需要安装.NET 4 SDK。

此外,除非要使用文本编辑器或其他第三方开发环境来编写C#代码,否则用户几乎肯定也希望使用 Visual Studio 2010。运行托管代码不需要安装完整的 SDK,但需要.NET 运行库。需要把.NET 运行库和代码分布到还没有安装它的客户端上。

本书内容

本书首先在第1章介绍.NET 的整体体系结构,给出编写托管代码所需要的背景知识,此后本书分几部分介绍C#语言及其在各个领域中的应用。

第I部分——C#语言

本部分给出C#语言的背景知识。尽管这一部分假定读者是有经验的编程人员,但它没有假设读者拥有任何特殊语言的知识。首先介绍C#的基本语法和数据类型,再介绍C#的面向对象功能,之后是C#中的一些高级编程主题。

第II部分——Visual Studio

本部分介绍全世界C#开发人员都使用的主要IDE: Visual Studio 2010。本部分的两章探讨使用工具构建基于.NET Framework 4 的应用程序的最佳方式,另外,本部分还讨论项目的部署。

第III部分——基础

本部分介绍在.NET 环境中编程的规则。特别是安全性、线程、本地化、事务、构建 Windows

服务的方式, 以及将自己的库生成为程序集的方式等主题。

第IV部分——数据

本部分介绍如何使用 ADO.NET 和 LINQ 访问数据库, 以及与目录和文件的交互。我们还详细说明 .NET 对 XML 的支持、对 Windows 操作系统的支持, 以及 SQL Server 2008 的 .NET 功能。

第V部分——显示

本部分讨论传统 Windows 应用程序的构建, 在 .NET 中这种应用程序称为 Windows 窗体。Windows 窗体是应用程序的胖客户端版本, 使用 .NET 构建这些类型的应用程序是实现该任务的一种快捷、简单的方式。本部分还阐述如何编写基于 Windows Presentation Foundation 和 Silverlight 的应用程序, 如何编写在 Web 站点上运行的组件, 如何编写网页。其中包括 ASP.NET 和 ASP.NET MVC 提供的许多新功能。

第VI部分——通信

这一部分介绍通信, 主要论述独立于平台使用 Windows Communication Foundation(WCF)进行通信的服务。通过消息队列, 揭示了断开连接的异步通信。本部分还介绍如何利用 Windows Workflow Foundation(WF)、对等网络, 以及创建联合源。

第VII部分——附录

这一部分介绍如何为 Windows 7 和 Windows Server 2008 R2 开发应用程序。

光盘所附章节

即使用这样一本厚书, 也不能涵盖 C# 以及使用这种语言和其他 .NET 技术的所有内容, 于是我们在本书附赠光盘放了全书的源代码和另外 10 章内容。这些章节包括各种主题的信息: GDI+ (这种技术用于构建包含高级图形的应用程序)、在 .NET 客户端和服务端之间通信的 .NET Remoting、用于后台服务的 Enterprise Services 和 Managed Add-In Framework(MAF)。光盘中的其他主题包括 VSTO 开发和使用 LINQ to SQL。

如何下载本书的示例代码

在读者学习本书中的示例时, 可以手工输入所有的代码, 也可以使用本书附带的源代码文件。本书使用的所有源代码都可以从本书合作站点 <http://www.wrox.com/> 和本书附赠光盘上找到。登录到站点 <http://www.wrox.com/> 上, 使用 Search 工具或书名列表就可以找到本书。接着单击本书细目页面上的 Download Code 链接, 就可以获得所有的源代码。

注释:

许多图书的书名都很相似, 所以通过 ISBN 查找本书是最简单的, 本书的 ISBN 是 978-0-470-50225-9。

在下载了代码后, 只需用自己喜欢的解压缩软件对它进行解压缩即可。另外, 也可以进入

<http://www.wrox.com/dynamic/books/download.aspx> 上的 Wrox 代码下载主页, 查看本书和其他 Wrox 图书的所有代码。

勘误表

尽管我们已经尽了各种努力来保证文章或代码中不出现错误, 但是错误总是难免的, 如果您在本书中找到了错误, 例如拼写错误或代码错误, 请告诉我们, 我们将非常感激。通过勘误表, 可以让其他读者避免受挫, 当然, 这还有助于提供更高质量的信息。

要在网站上找到本书的勘误表, 可以登录 <http://www.wrox.com>, 通过 Search 工具或书名列表查找本书, 然后在本书的细目页面上, 单击 Book Errata 链接。在这个页面上可以查看 Wrox 编辑已提交和粘贴的所有勘误项。完整的图书列表还包括每本书的勘误表, 网址是 www.wrox.com/misc-pages/booklist.shtml。

如果在 Book Errata 页面上没有看到您找出的错误, 请进入 www.worx.com/contact/techsupport.shtml, 填写表单, 发电子邮件, 我们会检查您的信息, 如果是正确的, 就在本书的勘误表中粘贴一个消息, 我们将在本书的后续版本中采用。

p2p.wrox.com

P2P 邮件列表是为作者和读者之间的讨论而建立的。读者可以在 p2p.wrox.com 上加入 P2P 论坛。该论坛是一个基于 Web 的系统, 用于传送与 Wrox 图书相关的信息和相关技术, 与其他读者和技术用户交流。该论坛提供了订阅功能, 当论坛上有新帖子时, 会给您发送您选择的主题。Wrox 作者、编辑和其他业界专家和读者都会在这个论坛上进行讨论。

在 <http://p2p.wrox.com> 上有许多不同的论坛, 帮助读者阅读本书, 在读者开发自己的应用程序时, 也可以从这个论坛中获益。要加入这个论坛, 须执行下面的步骤:

- (1) 进入 p2p.wrox.com, 单击 Register 链接。
- (2) 阅读其内容, 单击 Agree 按钮。
- (3) 提供加入论坛所需的信息及愿意提供的可选信息, 单击 Submit 按钮。
- (4) 然后就可以收到一封电子邮件, 其中的信息描述了如何验证账户, 完成加入过程。

提示:

不加入 P2P 也可以阅读论坛上的信息, 但只有加入论坛后, 才能发送自己的信息。

加入论坛后, 就可以发送新信息, 回应其他用户的帖子。可以随时在 Web 上阅读信息。如果希望某个论坛给自己发送新信息, 可以在论坛列表中单击该论坛对应的 Subscribe to this Forum 图标。

对于如何使用 Wrox P2P 的更多信息, 可阅读 P2P FAQ, 了解论坛软件的工作原理, 以及许多针对 P2P 和 Wrox 图书的常见问题解答。要阅读 FAQ, 可以单击任意 P2P 页面上的 FAQ 链接。

目 录

第 I 部分 C# 语言

第 1 章 .NET 体系结构	3
1.1 C#与.NET 的关系	3
1.2 公共语言运行库	4
1.2.1 平台无关性	4
1.2.2 提高性能	4
1.2.3 语言的互操作性	5
1.3 中间语言	6
1.3.1 面向对象和接口的支持	6
1.3.2 不同的值类型和引用类型	7
1.3.3 强数据类型化	8
1.3.4 通过异常处理错误	12
1.3.5 特性的使用	13
1.4 程序集	13
1.4.1 私有程序集	14
1.4.2 共享程序集	14
1.4.3 反射	14
1.4.4 并行编程	15
1.5 .NET Framework 类	15
1.6 名称空间	16
1.7 用 C#创建.NET 应用程序	16
1.7.1 创建 ASP.NET 应用程序	16
1.7.2 创建 Windows 窗体	18
1.7.3 使用 WPF	18
1.7.4 Windows 控件	19
1.7.5 Windows 服务	19
1.7.6 WCF	19
1.7.7 Windows WF	19
1.8 C#在.NET 企业体系结构 中的作用	19
1.9 小结	21

第 2 章 核心 C#	23
2.1 第一个 C#程序	23
2.1.1 代码	24
2.1.2 编译并运行程序	24
2.1.3 详细介绍	25
2.2 变量	26
2.2.1 变量的初始化	27
2.2.2 类型推断	28
2.2.3 变量的作用域	29
2.2.4 常量	31
2.3 预定义数据类型	32
2.3.1 值类型和引用类型	32
2.3.2 CTS 类型	33
2.3.3 预定义的值类型	33
2.3.4 预定义的引用类型	36
2.4 流控制	38
2.4.1 条件语句	38
2.4.2 循环	42
2.4.3 跳转语句	45
2.5 枚举	46
2.6 名称空间	47
2.6.1 using 语句	49
2.6.2 名称空间的别名	49
2.7 Main()方法	50
2.7.1 多个 Main()方法	50
2.7.2 给 Main()方法传递参数	52
2.8 有关编译 C#文件的更多内容	52
2.9 控制台 I/O	54
2.10 使用注释	56
2.10.1 源文件中的内部注释	56
2.10.2 XML 文档	56

2.11 C#预处理器指令.....	58	4.2.4 抽象类和抽象函数.....	100
2.11.1 #define 和 #undef.....	59	4.2.5 密封类和密封方法.....	100
2.11.2 #if, #elif, #else 和 #endif.....	59	4.2.6 派生类的构造函数.....	101
2.11.3 #warning 和 #error.....	60	4.3 修饰符.....	106
2.11.4 #region 和 #endregion.....	61	4.3.1 可见性修饰符.....	106
2.11.5 #line.....	61	4.3.2 其他修饰符.....	106
2.11.6 #pragma.....	61	4.4 接口.....	107
2.12 C#编程规则.....	62	4.4.1 定义和实现接口.....	108
2.12.1 关于标识符的规则.....	62	4.4.2 派生的接口.....	111
2.12.2 用法约定.....	63	4.5 小结.....	113
2.13 小结.....	68	第5章 泛型.....	115
第3章 对象和类型.....	69	5.1 概述.....	115
3.1 类和结构.....	69	5.1.1 性能.....	116
3.2 类.....	70	5.1.2 类型安全.....	117
3.2.1 数据成员.....	70	5.1.3 二进制代码的重用.....	117
3.2.2 函数成员.....	71	5.1.4 代码的扩展.....	117
3.2.3 只读字段.....	83	5.1.5 命名约定.....	118
3.3 匿名类型.....	84	5.2 创建泛型类.....	118
3.4 结构.....	85	5.3 泛型类的功能.....	122
3.4.1 结构是值类型.....	86	5.3.1 默认值.....	123
3.4.2 结构和继承.....	87	5.3.2 约束.....	123
3.4.3 结构的构造函数.....	87	5.3.3 继承.....	126
3.5 部分类.....	87	5.3.4 静态成员.....	127
3.6 静态类.....	89	5.4 泛型接口.....	127
3.7 Object 类.....	89	5.4.1 协变和抗变.....	128
3.7.1 System.Object()方法.....	90	5.4.2 泛型接口的协变.....	129
3.7.2 ToString()方法.....	90	5.4.3 泛型接口的抗变.....	130
3.8 扩展方法.....	92	5.5 泛型结构.....	131
3.9 小结.....	93	5.6 泛型方法.....	134
第4章 继承.....	95	5.6.1 泛型方法示例.....	134
4.1 继承的类型.....	95	5.6.2 带约束的泛型方法.....	135
4.1.1 实现继承和接口继承.....	95	5.6.3 带委托的泛型方法.....	136
4.1.2 多重继承.....	95	5.6.4 泛型方法规范.....	137
4.1.3 结构和类.....	96	5.7 小结.....	138
4.2 实现继承.....	96	第6章 数组.....	139
4.2.1 虚方法.....	97	6.1 简单数组.....	139
4.2.2 隐藏方法.....	98	6.1.1 数组的声明.....	139
4.2.3 调用函数的基类版本.....	99	6.1.2 数组的初始化.....	139

6.1.3 访问数组元素	140	第 8 章 委托、Lambda 表达式 和事件	197
6.1.4 使用引用类型	141	8.1 委托	197
6.2 多维数组	142	8.1.1 声明委托	198
6.3 锯齿数组	143	8.1.2 使用委托	199
6.4 Array 类	144	8.1.3 简单的委托示例	202
6.4.1 创建数组	145	8.1.4 Action<T>和 Func<T> 委托	204
6.4.2 复制数组	146	8.1.5 BubbleSorter 示例	204
6.4.3 排序	147	8.1.6 多播委托	207
6.5 数组作为参数	150	8.1.7 匿名方法	210
6.5.1 数组协变	151	8.2 Lambda 表达式	211
6.5.2 ArraySegment<T>	151	8.2.1 参数	212
6.6 枚举	152	8.2.2 多行代码	212
6.6.1 IEnumerable 接口	152	8.2.3 Lambda 表达式外部的 变量	213
6.6.2 foreach 语句	153	8.3 事件	214
6.6.3 yield 语句	153	8.3.1 事件发布程序	214
6.7 元组	158	8.3.2 事件侦听器	216
6.8 结构比较	159	8.3.3 弱事件	217
6.9 小结	162	8.4 小结	220
第 7 章 运算符和类型强制转换	163	第 9 章 字符串和正则表达式	221
7.1 运算符	163	9.1 System.String 类	221
7.1.1 运算符的简化操作	165	9.1.1 创建字符串	222
7.1.2 运算符的优先级	169	9.1.2 StringBuilder 成员	225
7.2 类型的安全性	169	9.1.3 格式字符串	226
7.2.1 类型转换	170	9.2 正则表达式	231
7.2.2 装箱和拆箱	173	9.2.1 正则表达式概述	232
7.3 比较对象的相等性	174	9.2.2 RegularExpressions Playaround 示例	233
7.3.1 比较引用类型的相等性	174	9.2.3 显示结果	235
7.3.2 比较值类型的相等性	175	9.2.4 匹配、组合和捕获	237
7.4 运算符重载	176	9.3 小结	238
7.4.1 运算符的工作方式	177	第 10 章 集合	239
7.4.2 运算符重载的示例: Vector 结构	178	10.1 集合接口和类型	239
7.5 用户定义的类型强制转换	185	10.2 列表	240
7.5.1 实现用户定义的类型 强制转换	186	10.2.1 创建列表	241
7.5.2 多重类型强制转换	192	10.2.2 只读集合	250
7.6 小结	195		

10.3	队列	250	11.3	并行 LINQ	306
10.4	栈	254	11.3.1	并行查询	306
10.5	链表	256	11.3.2	分区器	307
10.6	有序列表	261	11.3.3	取消	307
10.7	字典	262	11.4	表达式树	308
10.7.1	键的类型	263	11.5	LINQ 提供程序	311
10.7.2	字典示例	264	11.6	小结	311
10.7.3	Lookup 类	268	第 12 章	动态语言扩展	313
10.7.4	有序字典	269	12.1	DLR	313
10.8	集	269	12.2	dynamic 类型	313
10.9	可观察的集合	271	12.3	包含 DLR ScriptRuntime	318
10.10	位数组	272	12.4	DynamicObject 和 ExpandoObject	321
10.10.1	BitArray 类	273	12.4.1	DynamicObject	321
10.10.2	BitVector32 结构	275	12.4.2	ExpandoObject	323
10.11	并发集合	277	12.5	小结	324
10.12	性能	279	第 13 章	内存管理和指针	325
10.13	小结	281	13.1	后台内存管理	325
第 11 章	LINQ	283	13.1.1	值数据类型	325
11.1	LINQ 概述	283	13.1.2	引用数据类型	327
11.1.1	列表和实体	283	13.1.3	垃圾回收	328
11.1.2	LINQ 查询	287	13.2	释放非托管的资源	330
11.1.3	扩展方法	288	13.2.1	析构函数	330
11.1.4	推迟查询的执行	289	13.2.2	IDisposable 接口	331
11.2	标准的查询操作符	291	13.2.3	实现 IDisposable 接口 和析构函数	332
11.2.1	筛选	293	13.3	不安全的代码	334
11.2.2	用索引筛选	293	13.3.1	用指针直接访问内存	334
11.2.3	类型筛选	294	13.3.2	指针示例: PointerPlayground	343
11.2.4	复合的 from 子句	294	13.3.3	使用指针优化性能	347
11.2.5	排序	295	13.4	小结	350
11.2.6	分组	296	第 14 章	反射	351
11.2.7	对嵌套的对象分组	297	14.1	自定义特性	351
11.2.8	连接	298	14.1.1	编写自定义特性	352
11.2.9	集合操作	300	14.1.2	自定义特性示例: WhatsNewAttributes	355
11.2.10	合并	301			
11.2.11	分区	302			
11.2.12	聚合操作符	303			
11.2.13	转换	304			
11.2.14	生成操作符	305			

14.2	反射	358	16.5	小结	424
14.2.1	System.Type 类	358	第 17 章	部署	425
14.2.2	TypeView 示例	360	17.1	部署的规划	425
14.2.3	Assembly 类	362	17.1.1	部署选项	426
14.2.4	完成 WhatsNew Attributes 示例	364	17.1.2	部署要求	426
14.3	小结	368	17.1.3	部署 .NET 运行库	427
第 15 章	错误和异常	369	17.2	简单的部署选项	427
15.1	异常类	369	17.2.1	Xcopy 部署	428
15.2	捕获异常	371	17.2.2	Xcopy 和 Web 应用程序	428
15.2.1	实现多个 catch 块	373	17.2.3	发布 Web 站点	429
15.2.2	在其他代码中捕获异常	376	17.3	Visual Studio 2010 安装 和部署项目	429
15.2.3	System.Exception 属性	376	17.3.1	Windows Installer	430
15.2.4	没有处理异常时所发生 的情况	377	17.3.2	创建安装程序	430
15.2.5	嵌套的 try 块	378	17.4	ClickOnce	437
15.3	用户定义的异常类	379	17.4.1	ClickOnce 操作	437
15.3.1	捕获用户定义的异常	380	17.4.2	发布 ClickOnce 应用程序	438
15.3.2	抛出用户定义的异常	382	17.4.3	ClickOnce 设置	438
15.3.3	定义用户定义的异常类	385	17.4.4	ClickOnce 文件的 应用程序缓存	439
15.4	小结	387	17.4.5	应用程序的安全性	439
第 II 部分 Visual Studio					
第 16 章	Visual Studio 2010	391	17.5	Visual Studio 2010 高级选项	440
16.1	使用 Visual Studio 2010	391	17.5.1	文件系统编辑器	440
16.1.1	创建项目	395	17.5.2	注册表编辑器	440
16.1.2	解决方案和项目的区别	401	17.5.3	文件类型编辑器	440
16.1.3	Windows 应用程序代码	403	17.5.4	用户界面编辑器	441
16.1.4	项目的浏览和编码	404	17.5.5	自定义动作编辑器	442
16.1.5	生成项目	411	17.5.6	Launch Conditions 编辑器	443
16.1.6	调试代码	415	17.6	小结	444
16.2	重构工具	418	第 III 部分 基 础		
16.3	面向多个版本的 .NET Framework	420	第 18 章	程序集	447
16.4	WPF、WCF、WF 等	421	18.1	程序集的含义	447
16.4.1	在 Visual Studio 2010 中构建 WPF 应用程序	421	18.1.1	程序集的功能	448
16.4.2	在 Visual Studio 2010 中构建 WF 应用程序	423			

18.1.2	程序集的结构	448	19.1.2	后置条件	480
18.1.3	程序集清单	449	19.1.3	常量	481
18.1.4	名称空间、程序集 和组件	449	19.1.4	接口的协定	481
18.1.5	私有程序集和共享 程序集	449	19.2	跟踪	483
18.1.6	附属程序集	450	19.2.1	跟踪源	484
18.1.7	查看程序集	450	19.2.2	跟踪开关	485
18.2	创建程序集	451	19.2.3	跟踪侦听器	486
18.2.1	创建模块和程序集	451	19.2.4	筛选器	488
18.2.2	程序集的特性	452	19.2.5	相关性	489
18.2.3	动态加载和创建 程序集	454	19.3	事件日志	492
18.3	应用程序域	457	19.3.1	事件日志体系结构	493
18.4	共享程序集	461	19.3.2	事件日志类	494
18.4.1	强名	462	19.3.3	创建事件源	494
18.4.2	使用强名获得完整性	462	19.3.4	写入事件日志	495
18.4.3	全局程序集缓存	463	19.3.5	资源文件	496
18.4.4	创建共享程序集	463	19.4	性能监控	500
18.4.5	创建强名	464	19.4.1	性能监控类	500
18.4.6	安装共享程序集	465	19.4.2	性能计数器生成器	500
18.4.7	使用共享程序集	465	19.4.3	添加 Performance Counter 组件	503
18.4.8	程序集的延迟签名	466	19.4.4	perfmon.exe	505
18.4.9	引用	467	19.5	小结	506
18.4.10	本机映像生成器	468	第 20 章	线程、任务和同步	507
18.5	配置.NET 应用程序	469	20.1	概述	507
18.5.1	配置类别	469	20.2	异步委托	508
18.5.2	绑定程序集	470	20.2.1	投票	508
18.6	版本问题	471	20.2.2	等待句柄	509
18.6.1	版本号	472	20.2.3	异步回调	510
18.6.2	通过编程方式获取版本	472	20.3	Thread 类	512
18.6.3	绑定到程序集版本	473	20.3.1	给线程传递数据	513
18.6.4	发行者策略文件	474	20.3.2	后台线程	514
18.6.5	运行库的版本	475	20.3.3	线程的优先级	515
18.7	小结	476	20.3.4	控制线程	515
第 19 章	检测	477	20.4	线程池	516
19.1	代码协定	477	20.5	任务	517
19.1.1	前提条件	479	20.5.1	启动任务	517
			20.5.2	连续的任务	518
			20.5.3	任务层次结构	519

20.5.4 任务的结果.....	520	21.1.1 标识和 Principal.....	565
20.6 Parallel 类.....	521	21.1.2 角色.....	567
20.6.1 用 Parallel.For().....		21.1.3 声明基于角色的安全性.....	567
方法循环.....	521	21.1.4 客户端应用程序服务.....	568
20.6.2 使用 Parallel.ForEach().....		21.2 加密.....	573
方法循环.....	523	21.2.1 签名.....	575
20.6.3 通过 Parallel.Invoke().....		21.2.2 交换密钥和安全传输.....	576
方法调用多个方法.....	524	21.3 资源的访问控制.....	579
20.7 取消架构.....	524	21.4 代码访问安全性.....	582
20.7.1 Parallel.For()方法.....		21.4.1 第 2 级安全透明性.....	582
的取消.....	524	21.4.2 权限.....	583
20.7.2 任务的取消.....	526	21.5 使用证书发布代码.....	588
20.8 线程问题.....	527	21.6 小结.....	588
20.8.1 争用条件.....	527		
20.8.2 死锁.....	530	第 22 章 本地化.....	591
20.9 同步.....	532	22.1 System.Globalization.....	
20.9.1 lock 语句和线程安全.....	532	名称空间.....	591
20.9.2 Interlocked 类.....	538	22.1.1 Unicode 问题.....	592
20.9.3 Monitor 类.....	539	22.1.2 区域性和区域.....	592
20.9.4 SpinLock 结构.....	540	22.1.3 使用区域性.....	596
20.9.5 WaitHandle 基类.....	540	22.1.4 排序.....	600
20.9.6 Mutex 类.....	541	22.2 资源.....	602
20.9.7 Semaphore 类.....	542	22.2.1 创建资源文件.....	602
20.9.8 Events 类.....	544	22.2.2 资源文件生成器.....	602
20.9.9 Barrier 类.....	547	22.2.3 ResourceWriter.....	603
20.9.10 ReaderWriterLockSlim.....		22.2.4 使用资源文件.....	604
类.....	549	22.2.5 System.Resources.....	
20.10 Timer 类.....	552	名称空间.....	607
20.11 基于事件的异步模式.....	554	22.3 使用 Visual Studio 的.....	
20.11.1 BackgroundWorker.....		Windows 窗体本地化.....	607
类.....	554	22.3.1 通过编程方式修改.....	
20.11.2 启用取消功能.....	557	区域性.....	612
20.11.3 启用进度功能.....	558	22.3.2 使用自定义资源文件.....	613
20.11.4 创建基于事件的异步.....		22.3.3 资源的自动回退.....	614
组件.....	559	22.3.4 外包翻译.....	615
20.12 小结.....	563	22.4 用 ASP.NET 本地化.....	615
第 21 章 安全性.....	565	22.5 用 WPF 本地化.....	617
21.1 身份验证和授权.....	565	22.5.1 用于 WPF 的 .NET 资源.....	618
		22.5.2 XAML 资源字典.....	619

22.6	自定义资源读取器	622	24.2.2	使用代理	666
22.6.1	创建 DatabaseResource Reader 类	623	24.2.3	异步页面请求	666
22.6.2	创建 DatabaseResource Set 类	625	24.3	把输出结果显示为 HTML 页面	667
22.6.3	创建 DatabaseResource Manager 类	625	24.3.1	从应用程序中进行 简单的 Web 浏览	667
22.6.4	DatabaseResourceReader 的客户端应用程序	626	24.3.2	启动 Internet Explorer 实例	669
22.7	创建自定义区域性	626	24.3.3	给应用程序提供更多 的 IE 类型特性	669
22.8	小结	628	24.3.4	使用 WebBrowser 控件 打印	674
第 23 章	System.Transactions	629	24.3.5	显示请求页面的代码	674
23.1	概述	629	24.3.6	WebRequest 类 和 WebResponse 类的 层次结构	676
23.1.1	事务处理阶段	630	24.4	实用工具类	676
23.1.2	ACID 属性	630	24.4.1	URI	676
23.2	数据库和实体类	631	24.4.2	IP 地址和 DNS 名称	677
23.3	传统的事务	633	24.5	较低层的协议	679
23.3.1	ADO.NET 事务	633	24.5.1	使用 SmtClient	680
23.3.2	System.Enterprise Services	634	24.5.2	使用 TCP 类	682
23.4	System.Transactions	635	24.5.3	TcpSend 和 TcpReceive 示例	682
23.4.1	可提交的事务	636	24.5.4	TCP 和 UDP	684
23.4.2	事务处理的升级	638	24.5.5	UDP 类	684
23.4.3	依赖事务	640	24.5.6	Socket 类	685
23.4.4	环境事务	642	24.6	小结	689
23.5	隔离级别	649	第 25 章	Windows 服务	691
23.6	自定义资源管理器	650	25.1	Windows 服务	691
23.7	Windows 7 和 Windows Server 2008 的事务	656	25.2	Windows 服务的体系结构	692
23.8	小结	660	25.2.1	服务程序	692
第 24 章	网络	661	25.2.2	服务控制程序	694
24.1	WebClient 类	661	25.2.3	服务配置程序	694
24.1.1	下载文件	662	25.2.4	Windows 服务的类	694
24.1.2	基本的 WebClient 示例	662	25.3	创建 Windows 服务程序	694
24.1.3	上传文件	663	25.3.1	创建服务的核心功能	695
24.2	WebRequest 类和 WebResponse 类	664	25.3.2	QuoteClient 示例	698
24.2.1	身份验证	666	25.3.3	Windows 服务程序	699

25.3.4	线程和服务	703	26.4.3	创建类型库	741
25.3.5	服务的安装	704	26.4.4	COM 互操作特性	743
25.3.6	安装程序	704	26.4.5	COM 注册	745
25.4	服务的监视和控制	708	26.4.6	创建 COM 客户端	
25.4.1	MMC 管理单元	708	应用程序	746	
25.4.2	net.exe 实用程序	709	26.4.7	添加连接点	747
25.4.3	sc.exe 实用程序	710	26.4.8	用 sink 对象创建	
25.4.4	Visual Studio Server		客户端	748	
	Explorer	710	26.5	平台调用	750
25.4.5	编写自定义		26.6	小结	754
	ServiceController 类	710	第 27 章	核心 XAML	755
25.5	故障排除和事件日志	718	27.1	概述	755
25.6	小结	719	27.1.1	元素映射到.NET	
第 26 章	互操作性	721	对象上	756	
26.1	.NET 和 COM	721	27.1.2	使用自定义.NET 类	757
26.1.1	元数据	722	27.1.3	把特性用作属性	759
26.1.2	释放内存	722	27.1.4	把特性用作元素	759
26.1.3	接口	722	27.1.5	基本的.NET 类型	760
26.1.4	方法的绑定	724	27.1.6	集合	760
26.1.5	数据类型	724	27.1.7	构造函数	761
26.1.6	注册	724	27.2	依赖属性	761
26.1.7	线程	725	27.2.1	创建依赖类型	762
26.1.8	错误处理	726	27.2.2	强制值回调	763
26.1.9	事件	727	27.2.3	值变更回调和事件	764
26.2	编组	727	27.2.4	事件的冒泡和隧道	764
26.3	从.NET 客户端中使用		27.3	附加属性	767
COM 组件	728	27.4	标记扩展	770	
26.3.1	创建 COM 组件	728	27.5	创建自定义标记扩展	770
26.3.2	创建 RCW	734	27.6	XAML 定义的标记扩展	772
26.3.3	使用 RCW	735	27.7	读写 XAML	772
26.3.4	使用 COM 服务器和		27.8	小结	773
动态语言扩展	736	第 28 章	Managed Extensibility		
26.3.5	线程问题	737	Framework	775	
26.3.6	添加连接点	737	28.1	MEF 的体系结构	775
26.4	从 COM 客户端中使用		28.2	协定	782
.NET 组件	739	28.3	导出	783	
26.4.1	CCM	740	28.3.1	导出属性和方法	787
26.4.2	创建.NET 组件	740	28.3.2	导出元数据	789

28.4	导入	791
28.5	容器和出口提供程序	794
28.6	类别	797
28.7	小结	798
第 29 章	文件和注册表操作	799
29.1	管理文件系统	799
29.1.1	表示文件和文件夹 的.NET 类	800
29.1.2	Path 类	803
29.1.3	FileProperties 示例	803
29.2	移动、复制和删除文件	808
29.2.1	FilePropertiesAndMovement 示例	808
29.2.2	FilePropertiesAndMovement 示例的代码	809
29.3	读写文件	812
29.3.1	读取文件	812
29.3.2	写入文件	814
29.3.3	流	815
29.3.4	缓存的流	816
29.3.5	使用 FileStream 类读写 二进制文件	816
29.3.6	读写文本文件	821
29.4	映射内存的文件	827
29.5	读取驱动器信息	829
29.6	文件的安全性	831
29.6.1	从文件中读取 ACL	831
29.6.2	从目录中读取 ACL	832
29.6.3	添加和删除文件中 的 ACL 项	833
29.7	读写注册表	835
29.7.1	注册表	835
29.7.2	.NET 注册表类	837
29.8	读写独立存储器	839
29.9	小结	844

第IV部分 数 据

第 30 章	核心 ADO.NET	847
30.1	ADO.NET 概述	847

30.1.1	名称空间	848
30.1.2	共享类	848
30.1.3	数据库专用的类	849
30.2	使用数据库连接	850
30.2.1	管理连接字符串	851
30.2.2	高效地使用连接	852
30.2.3	事务	854
30.3	命令	855
30.3.1	执行命令	856
30.3.2	调用存储过程	860
30.4	快速数据访问: 数据读取器	862
30.5	管理数据和关系: DataSet 类	865
30.5.1	数据表	866
30.5.2	数据列	866
30.5.3	数据关系	871
30.5.4	数据约束	872
30.6	XML 架构: 用 XSD 生成代码	875
30.7	填充 DataSet 类	881
30.7.1	用数据适配器填充 DataSet	882
30.7.2	从 XML 中填充 DataSet 类	883
30.8	持久化 DataSet 类的修改	883
30.8.1	通过数据适配器进行 更新	883
30.8.2	写入 XML 输出结果	886
30.9	使用 ADO.NET	887
30.9.1	分层开发	887
30.9.2	生成 SQL Server 的键	888
30.9.3	命名约定	891
30.10	小结	892

第 31 章	ADO.NET Entity Framework	893
31.1	ADO.NET Entity Framework 概述	893

31.2 Entity Framework 映射	894	32.5 小结	938
31.2.1 逻辑层	895	第 33 章 处理 XML	939
31.2.2 概念层	897	33.1 .NET 支持的 XML 标准	940
31.2.3 映射层	898	33.2 System.Xml 名称空间	940
31.3 Entity Client	899	33.3 使用 System.Xml 类	941
31.3.1 连接字符串	900	33.4 读写流格式的 XML	942
31.3.2 Entity SQL	900	33.4.1 使用 XmlReader 类	942
31.4 实体	901	33.4.2 使用 XmlReader 类进行 验证	946
31.5 对象上下文	904	33.4.3 使用 XmlWriter 类	947
31.6 关系	906	33.5 在 .NET 中使用 DOM	949
31.6.1 一个层次结构一个表	907	33.6 使用 XPathNavigator 类	954
31.6.2 一种类型一个表	908	33.6.1 System.Xml.XPath 名称空间	954
31.6.3 懒惰加载、延迟加载 和预先加载	909	33.6.2 System.Xml.Xsl 名称空间	959
31.7 对象查询	910	33.6.3 调试 XSLT	963
31.8 更新	913	33.7 XML 和 ADO.NET	965
31.8.1 对象跟踪	913	33.7.1 将 ADO.NET 数据 转换为 XML 文档	965
31.8.2 改变信息	914	33.7.2 把 XML 文档转换为 ADO.NET 数据	970
31.8.3 附加和分离实体	916	33.8 在 XML 中序列化对象	972
31.8.4 存储实体的变化	916	33.9 LINQ to XML 和 .NET	982
31.9 LINQ to Entities	917	33.10 使用不同的 XML 对象	982
31.10 小结	918	33.10.1 XDocument 对象	982
第 32 章 数据服务	919	33.10.2 XElement 对象	983
32.1 概述	919	33.10.3 XNamespace 对象	984
32.2 包含 CLR 对象的自定义 宿主	920	33.10.4 XComment 对象	986
32.2.1 CLR 对象	920	33.10.5 XAttribute 对象	986
32.2.2 数据模型	922	33.11 使用 LINQ 查询 XML 文档	987
32.2.3 数据服务	923	33.11.1 查询静态的 XML 文档	987
32.2.4 驻留服务	923	33.11.2 查询动态的 XML 文档	989
32.2.5 其他服务操作	924	33.12 XML 文档的更多查询 技术	990
32.3 HTTP 客户端应用程序	925		
32.4 使用 WCF 数据服务和 ADO.NET Entity Framework	929		
32.4.1 ASP.NET 宿主和 EDM	929		
32.4.2 使用 System.Data .Service.Client 的 .NET 应用程序	931		

33.12.1	读取 XML 文档	990	35.1.1	名称空间	1025
33.12.2	写入 XML 文档	992	35.1.2	类层次结构	1027
33.13	小结	993	35.2	形状	1028
第 34 章	.NET 编程和 SQL		35.3	几何图形	1030
	Server	995	35.4	变换	1032
34.1	.NET 运行库的宿主	995	35.5	画笔	1033
34.2	Microsoft.SqlServer.Server	997	35.5.1	SolidColorBrush	1033
34.3	用户定义的类型	998	35.5.2	LinearGradientBrush	1034
34.3.1	创建 UDT	998	35.5.3	RadialGradientBrush	1034
34.3.2	通过 SQL 使用 UDT	1003	35.5.4	DrawingBrush	1035
34.3.3	从客户端代码中使用 UDT	1004	35.5.5	ImageBrush	1035
34.4	用户定义的聚合函数	1005	35.5.6	VisualBrush	1036
34.4.1	创建用户定义的 聚合函数	1006	35.6	控件	1037
34.4.2	使用用户定义的 聚合函数	1007	35.6.1	简单控件	1037
34.5	存储过程	1007	35.6.2	内容控件	1038
34.5.1	创建存储过程	1008	35.6.3	带标题的内容控件	1039
34.5.2	使用存储过程	1009	35.6.4	项控件	1040
34.6	用户定义的函数	1010	35.6.5	带标题的项控件	1041
34.6.1	创建用户定义的函数	1010	35.6.6	修饰	1041
34.6.2	使用用户定义的函数	1010	35.7	布局	1042
34.7	触发器	1011	35.7.1	StackPanel	1042
34.7.1	创建触发器	1011	35.7.2	WrapPanel	1043
34.7.2	使用触发器	1012	35.7.3	Canvas	1044
34.8	XML 数据类型	1013	35.7.4	DockPanel	1044
34.8.1	包含 XML 数据的表	1013	35.7.5	Grid	1045
34.8.2	读取 XML 值	1014	35.8	样式和资源	1046
34.8.3	数据的查询	1017	35.8.1	样式	1046
34.8.4	XML 数据修改语言 (XML DML)	1019	35.8.2	资源	1048
34.8.5	XML 索引	1020	35.8.3	系统资源	1049
34.8.6	强类型化的 XML	1021	35.8.4	从代码中访问资源	1050
34.9	小结	1022	35.8.5	动态资源	1050
			35.8.6	资源字典	1051
第 V 部分	显 示		35.9	触发器	1052
第 35 章	核心 WPF	1025	35.9.1	属性触发器	1053
35.1	概述	1025	35.9.2	多触发器	1054
			35.9.3	数据触发器	1055
			35.10	模板	1056
			35.10.1	控件模板	1057
			35.10.2	数据模板	1060

35.10.3 样式化列表框	1061	36.4 DataGrid	1118
35.10.4 ItemTemplate	1062	36.4.1 自定义列	1120
35.10.5 列表框元素的控件 模板	1064	36.4.2 行的细节	1121
35.11 动画	1066	36.4.3 用 DataGrid 进行分组	1121
35.11.1 时间轴	1066	36.5 小结	1124
35.11.2 非线性动画	1069	第 37 章 用 WPF 创建文档	1125
35.11.3 事件触发器	1069	37.1 文本元素	1125
35.11.4 关键帧动画	1072	37.1.1 字体	1125
35.12 可见状态管理器	1073	37.1.2 TextEffect	1127
35.13 3-D	1076	37.1.3 内联	1128
35.13.1 模型	1077	37.1.4 块	1130
35.13.2 照相机	1079	37.1.5 列表	1132
35.13.3 光线	1079	37.1.6 表	1132
35.13.4 旋转	1079	37.1.7 块的锚定	1134
35.14 小结	1080	37.2 流文档	1135
第 36 章 用 WPF 编写业务 应用程序	1083	37.3 固定文档	1136
36.1 数据绑定	1083	37.4 XPS 文档	1140
36.1.1 BooksDemo 应用程序	1084	37.5 打印	1141
36.1.2 用 XAML 绑定	1086	37.5.1 用 PrintDialog 打印	1142
36.1.3 简单对象的绑定	1088	37.5.2 打印可见元素	1142
36.1.4 更改通知	1090	37.6 小结	1144
36.1.5 对象数据提供程序	1092	第 38 章 Silverlight	1145
36.1.6 列表绑定	1094	38.1 WPF 和 Silverlight 的比较	1145
36.1.7 主从绑定	1096	38.2 创建 Silverlight 项目	1146
36.1.8 多绑定	1097	38.3 导航	1148
36.1.9 优先绑定	1099	38.4 网络	1152
36.1.10 值的转换	1100	38.4.1 创建 ADO.NET Entity Data Model	1153
36.1.11 动态添加列表项	1102	38.4.2 为 Silverlight 客户端 创建 WCF 服务	1153
36.1.12 数据模板选择器	1103	38.4.3 调用 WCF 服务	1155
36.1.13 绑定到 XML 上	1105	38.4.4 使用 WCF 数据服务	1158
36.1.14 绑定的验证	1107	38.4.5 使用 System.Net 访问 服务	1160
36.2 Commanding	1111	38.5 浏览器集成	1162
36.2.1 定义命令	1112	38.5.1 调用 JavaScript	1162
36.2.2 定义命令源	1113	38.5.2 JavaScript 调用 Silverlight	1163
36.2.3 命令绑定	1113		
36.3 TreeView	1114		

38.6	在浏览器外运行的 Silverlight 应用程序.....	1164
38.7	小结	1167
第 39 章	Windows 窗体	1169
39.1	创建 Windows 窗体 应用程序	1169
39.2	Control 类	1175
39.2.1	大小和位置	1175
39.2.2	外观	1176
39.2.3	用户交互操作	1176
39.2.4	Windows 功能	1177
39.2.5	其他功能	1177
39.3	标准控件和组件	1178
39.3.1	Button 控件	1178
39.3.2	CheckBox 控件	1178
39.3.3	RadioButton 控件	1179
39.3.4	ComboBox 控件、 ListBox 控件和 CheckedListBox 控件	1179
39.3.5	DataGridView 控件	1180
39.3.6	DateTimePicker 控件	1188
39.3.7	ErrorProvider 组件	1188
39.3.8	ImageList 组件	1189
39.3.9	Label 控件	1189
39.3.10	ListView 控件	1189
39.3.11	PictureBox 控件	1189
39.3.12	ProgressBar 控件	1190
39.3.13	TextBox 控件、 RichTextBox 控件与 MaskedTextBox 控件	1190
39.3.14	Panel 控件	1191
39.3.15	FlowLayoutPanel 和 TableLayoutPanel 控件	1191
39.3.16	SplitContainer 控件	1192
39.3.17	TabControl 控件和 TabPage 控件	1192
39.3.18	ToolStrip 控件	1193
39.3.19	MenuStrip 控件	1194
39.3.20	ContextMenuStrip 控件	1194
39.3.21	ToolStripMenuItem 控件	1194
39.3.22	ToolStripManager 类	1194
39.3.23	ToolStripContainer 控件	1195
39.4	窗体	1195
39.4.1	Form 类	1195
39.4.2	多文档界面	1199
39.4.3	创建自己的用户控件	1200
39.5	小结	1200
第 40 章	核心 ASP.NET	1201
40.1	ASP.NET 概述	1201
40.1.1	ASP.NET 文件的 处理方式	1202
40.1.2	Web 站点和 Web 应用程序	1202
40.1.3	ASP.NET 中的状态 管理	1203
40.2	ASP.NET Web 窗体	1203
40.2.1	ASP.NET 代码模型	1207
40.2.2	ASP.NET 服务器控件	1208
40.3	ADO.NET 和数据绑定	1222
40.3.1	更新事件登记 应用程序	1222
40.3.2	数据绑定的更多内容	1229
40.4	应用程序配置	1234
40.5	小结	1236
第 41 章	ASP.NET 的功能	1237
41.1	用户控件和自定义控件	1238
41.1.1	用户控件	1238
41.1.2	PCSDemoSite 中的 用户控件	1242
41.1.3	自定义控件	1243
41.2	母版页	1247

41.2.1 在 Web 页面中 访问母版页	1248	41.12.1 ScriptManager 控件	1280
41.2.2 嵌套的母版页	1248	41.12.2 使用 UpdatePanel 控件	1281
41.2.3 PCSDemoSite 中 的母版页	1249	41.12.3 使用 UpdateProgress	1283
41.3 站点导航	1250	41.12.4 使用扩展控件	1284
41.3.1 添加站点地图文件	1251	41.13 使用 AJAX 库	1286
41.3.2 PCSDemoSite 中 的导航	1252	41.13.1 给 Web 页面添加 JavaScript	1286
41.4 安全性	1253	41.13.2 全局实用程序函数	1287
41.4.1 使用 Security Setup 添加 Forms 身份验证 功能	1254	41.13.3 使用 AJAX 库 JavaScript OOP 扩展	1287
41.4.2 实现登录系统	1255	41.13.4 PageRequestManager 对象和 Application 对象	1289
41.4.3 Web 登录服务器控件	1255	41.13.5 JavaScript 的调试	1292
41.4.4 保护目录	1256	41.13.6 异步调用 Web 方法	1293
41.4.5 PCSDemoSite 中的 安全性	1257	41.13.7 ASP.NET 应用程序 服务	1294
41.5 主题	1259	41.14 小结	1294
41.5.1 把主题应用于页面	1259		
41.5.2 定义主题	1260	第 42 章 ASP.NET 动态数据 和 MVC	1297
41.5.3 PCSDemoSite 中的 主题	1260	42.1 路由	1298
41.6 Web 部件	1263	42.1.1 查询字符串参数	1298
41.6.1 Web 部件应用程序 组件	1263	42.1.2 定义路由	1300
41.6.2 Web 部件示例	1264	42.1.3 使用路由参数	1303
41.7 ASP.NET AJAX	1270	42.2 动态数据	1305
41.8 Ajax 的概念	1271	42.2.1 创建动态数据网站	1305
41.9 ASP.NET AJAX	1273	42.2.2 定制动态数据网站	1310
41.9.1 核心功能	1273	42.2.3 进一步开发	1314
41.9.2 ASP.NET AJAX Control Toolkit	1275	42.3 MVC	1314
41.10 ASP.NET AJAX 网站 示例	1276	42.3.1 MVC 的含义	1314
41.11 支持 ASP.NET AJAX 的 网站配置	1278	42.3.2 ASP.NET MVC 的 含义	1315
41.12 添加 ASP.NET AJAX 功能	1279	42.3.3 简单的 ASP.NET MVC 应用程序	1315
		42.3.4 定制 ASP.NET MVC 应用程序	1321

42.3.5 进一步开发.....	1329	44.2.1 If 活动.....	1366
42.4 小结.....	1330	44.2.2 InvokeMethod 活动.....	1367
第VI部分 通 信			
第43章 WCF.....	1333	44.2.3 Parallel 活动.....	1367
43.1 WCF 概述.....	1333	44.2.4 Delay 活动.....	1368
43.1.1 SOAP.....	1335	44.2.5 Pick 活动.....	1368
43.1.2 WSDL.....	1335	44.3 自定义活动.....	1369
43.1.3 REST.....	1335	44.3.1 活动的验证.....	1370
43.1.4 JSON.....	1336	44.3.2 设计器.....	1371
43.2 简单的服务和客户端.....	1336	44.3.3 自定义复合活动.....	1373
43.2.1 服务协定.....	1337	44.4 工作流.....	1375
43.2.2 服务的实现.....	1338	44.4.1 实参和变量.....	1376
43.2.3 WCF 服务宿主和 WCF 测试客户端.....	1338	44.4.2 WorkflowApplication.....	1377
43.2.4 自定义服务宿主.....	1340	44.4.3 WorkflowServiceHost.....	1381
43.2.5 WCF 客户端.....	1342	44.4.4 驻留设计器.....	1386
43.2.6 诊断.....	1343	44.5 小结.....	1391
43.3 协定.....	1345	第45章 对等网络.....	1393
43.3.1 数据协定.....	1345	45.1 P2P 网络概述.....	1393
43.3.2 版本问题.....	1346	45.1.1 客户端-服务器体系 结构.....	1393
43.3.3 服务协定.....	1346	45.1.2 P2P 体系结构.....	1394
43.3.4 消息协定.....	1347	45.1.3 P2P 体系结构的挑战.....	1395
43.4 服务的实现.....	1348	45.1.4 P2P 术语.....	1396
43.4.1 以编程方式创建 客户端.....	1351	45.1.5 P2P 解决方案.....	1396
43.4.2 错误处理.....	1352	45.2 Microsoft Windows Peer-to-Peer Networking.....	1396
43.5 绑定.....	1353	45.2.1 PNRP.....	1396
43.6 宿主.....	1356	45.2.2 People Near Me.....	1399
43.6.1 自定义宿主.....	1356	45.3 构建 P2P 应用程序.....	1400
43.6.2 WAS 宿主.....	1357	45.3.1 System.Net.PeerToPeer.....	1400
43.6.3 预配置的宿主类.....	1357	45.3.2 System.Net.PeerToPeer .Collaboration.....	1405
43.7 客户端.....	1358	45.4 小结.....	1408
43.8 双工通信.....	1360	第46章 消息队列.....	1409
43.9 小结.....	1362	46.1 概述.....	1409
第44章 Windows WF 4.....	1363	46.1.1 使用消息队列的场合.....	1410
44.1 Hello World 示例.....	1363	46.1.2 消息队列功能.....	1411
44.2 活动.....	1365	46.2 Message Queuing 产品.....	1412
		46.3 消息队列体系结构.....	1412

46.3.1 消息.....	1413	47.3 联合源的示例.....	1444
46.3.2 消息队列.....	1413	47.4 小结.....	1449
46.4 Message Queuing		第VII部分 附 录	
管理工具.....	1414	附录 A	1453
46.4.1 创建消息队列.....	1414		
46.4.2 消息队列属性.....	1415	***以下内容见随书附赠光盘***	
46.5 消息队列的编程实现.....	1415	第 48 章 使用 GDI+绘图.....	E1
46.5.1 创建消息队列.....	1415	48.1 理解绘图规则.....	E1
46.5.2 查找队列.....	1416	48.1.1 GDI 和 GDI+	E2
46.5.3 打开已知队列.....	1417	48.1.2 绘制图形.....	E3
46.5.4 发送消息.....	1418	48.1.3 使用 OnPaint()方法	
46.5.5 接收消息.....	1421	绘制图形	E6
46.6 课程订单应用程序.....	1423	48.1.4 使用剪切区域.....	E7
46.6.1 课程订单类库.....	1423	48.2 测量坐标和区域.....	E9
46.6.2 课程订单消息		48.2.1 Point 和 PointF 结构.....	E9
发送程序.....	1424	48.2.2 Size 和 SizeF 结构.....	E10
46.6.3 发送优先级和可恢复		48.2.3 Rectangle 和 RectangleF	
的消息.....	1425	结构.....	E11
46.6.4 课程订单消息接收		48.2.4 Region	E12
程序.....	1426	48.3 调试须知.....	E13
46.7 接收结果	1429	48.4 绘制可滚动的窗口.....	E14
46.7.1 确认队列.....	1430	48.5 世界、页面和设备坐标.....	E18
46.7.2 响应队列.....	1430	48.6 颜色.....	E19
46.8 事务队列	1431	48.6.1 RGB 值.....	E19
46.9 消息队列和 WCF.....	1432	48.6.2 命名颜色	E20
46.9.1 带数据协定的实体类.....	1432	48.6.3 图形显示模式和安全	
46.9.2 WCF 服务协定	1433	的调色板	E20
46.9.3 WCF 消息接收		48.6.4 安全调色板.....	E21
应用程序.....	1434	48.7 画笔和钢笔.....	E21
46.9.4 WCF 消息发送		48.7.1 画笔.....	E22
应用程序.....	1437	48.7.2 钢笔.....	E23
46.10 消息队列的安装.....	1438	48.8 绘制图形和线条.....	E23
46.11 小结.....	1438	48.9 显示图像.....	E25
第 47 章 Syndication	1441	48.10 处理图像时的问题.....	E27
47.1 System.ServiceModel.		48.11 绘制文本.....	E28
Syndication 名称空间			
概述	1441		
47.2 读取联合源的示例.....	1442		

48.12	简单的文本示例	E29	50.1.5	生命周期	E84
48.13	字体和字体系列	E30	50.1.6	版本问题	E85
48.14	示例: 枚举字体系列	E31	50.2	插件示例	E86
48.15	编辑文本文档:		50.2.1	插件协定	E86
	CapsEditor 示例	E33	50.2.2	计算器插件视图	E87
	48.15.1 Invalidate()方法	E37	50.2.3	计算器插件适配器	E88
	48.15.2 计算项的大小和文档		50.2.4	计算器插件	E90
	的大小	E38	50.2.5	计算器宿主视图	E91
	48.15.3 OnPaint()方法	E39	50.2.6	计算机宿主适配器	E91
	48.15.4 坐标转换	E41	50.2.7	计算器宿主	E93
	48.15.5 响应用户的输入	E42	50.2.8	其他插件	E97
48.16	打印	E45	50.3	小结	E97
48.17	小结	E50			
第 49 章	VSTO	E51	第 51 章	Enterprise Services	E99
49.1	VSTO 概述	E51	51.1	使用 Enterprise Services	E99
49.1.1	项目类型	E52	51.1.1	简史	E100
49.1.2	项目功能	E54	51.1.2	使用 Enterprise Services	
49.2	VSTO 项目基础	E55		场合	E100
49.2.1	Office 对象模型	E55	51.1.3	重要功能	E101
49.2.2	VSTO 名称空间	E55	51.2	创建简单的 COM+ 应用	
49.2.3	宿主项和宿主控件	E56		程序	E103
49.2.4	基本的 VSTO 项目		51.2.1	ServicedComponent 类	E103
	结构	E57	51.2.2	程序集的属性	E103
49.2.5	Globals 类	E60	51.2.3	创建组件	E104
49.2.6	事件处理	E60	51.3	部署	E106
49.3	构建 VSTO 解决方案	E61	51.3.1	自动部署	E106
49.3.1	管理应用程序级插件	E62	51.3.2	手工部署	E106
49.3.2	与应用程序和文档		51.3.3	创建安装软件包	E106
	交互操作	E63	51.4	组件服务管理器	E107
49.3.3	UI 的自定义	E64	51.5	客户端应用程序	E108
49.4	示例应用程序	E68	51.6	事务	E109
49.5	小结	E78	51.6.1	事务的特性	E109
第 50 章	MAF	E79	51.6.2	事务的结果	E110
50.1	MAF 体系结构	E79	51.7	示例应用程序	E111
50.1.1	管道	E80	51.7.1	实体类	E111
50.1.2	发现	E81	51.7.2	OrderControl 组件	E113
50.1.3	激活和隔离	E82	51.7.3	OrderData 组件	E114
50.1.4	协定	E83	51.7.4	OrderLineData 组件	E116
			51.7.5	客户端应用程序	E117

51.8 集成 WCF 和 Enterprise Services	E118	52.5 账户管理	E152
51.8.1 WCF 服务外观	E118	52.5.1 显示用户信息	E152
51.8.2 客户端应用程序	E122	52.5.2 创建用户	E153
51.9 小结	E123	52.5.3 重置密码	E153
第 52 章 目录服务	E125	52.5.4 创建组	E154
52.1 Active Directory 的 体系结构	E126	52.5.5 在组中添加用户	E154
52.1.1 Active Directory 的 功能	E126	52.5.6 查找用户	E154
52.1.2 Active Directory 的 概念	E126	52.6 DSML	E155
52.1.3 Active Directory 数据 的特征	E129	52.6.1 System.DirectoryServices. Protocols 名称空间中 的类	E156
52.1.4 指定架构	E130	52.6.2 用 DSML 搜索 Active Directory 对象	E156
52.2 Active Directory 的 管理工具	E131	52.7 小结	E157
52.2.1 Active Directory Users and Computers 工具	E131	第 53 章 C#、Visual Basic、 C++/CLI 和 F#	E159
52.2.2 ADSI Edit 工具	E132	53.1 名称空间	E160
52.3 Active Directory 编程	E133	53.2 定义类型	E161
52.3.1 System.DirectoryServices 名称空间中的类	E134	53.2.1 引用类型	E161
52.3.2 绑定到 Directory Services	E134	53.2.2 值类型	E162
52.3.3 获取目录项	E138	53.2.3 类型推断	E163
52.3.4 对象集合	E140	53.2.4 接口	E164
52.3.5 缓存	E141	53.2.5 枚举	E165
52.3.6 创建新对象	E141	53.3 方法	E166
52.3.7 更新目录项	E142	53.3.1 方法的参数和返回 类型	E166
52.3.8 访问本地 ADSI 对象	E143	53.3.2 参数修饰符	E167
52.3.9 在 Active Directory 中搜索	E144	53.3.3 构造函数	E168
52.4 搜索用户对象	E148	53.3.4 属性	E170
52.4.1 用户界面	E148	53.3.5 对象初始值设定项	E171
52.4.2 获取架构命名上下文	E149	53.3.6 扩展方法	E171
52.4.3 获取 User 类的属性名	E149	53.4 静态成员	E172
52.4.4 搜索用户对象	E150	53.5 数组	E173
		53.6 控制语句	E174
		53.6.1 if 语句	E174
		53.6.2 条件操作符	E174
		53.6.3 switch 语句	E175
		53.7 循环	E176

53.7.1	for 语句	E176	54.5	.NET Remoting 体系结构	E204
53.7.2	while 和 do...while 语句	E177	54.5.1	信道	E204
53.7.3	foreach 语句	E178	54.5.2	格式化程序	E207
53.8	异常处理	E178	54.5.3	ChannelServices 和 RemotingConfiguration	E207
53.9	继承	E180	54.5.4	对象的激活	E209
53.9.1	访问修饰符	E180	54.5.5	消息接收器	E212
53.9.2	关键字	E181	54.5.6	在远程方法中传递对象	E213
53.10	资源管理	E183	54.5.7	生命周期管理	E216
53.10.1	IDisposable 接口的 实现	E183	54.6	配置文件	E218
53.10.2	using 语句	E184	54.6.1	知名对象的服务器 配置	E220
53.10.3	重写 Finalize()方法	E184	54.6.2	知名对象的客户端 配置	E220
53.11	委托	E186	54.6.3	客户端激活的对象的 服务器配置	E221
53.12	事件	E188	54.6.4	客户端激活的对象的 客户端配置	E222
53.13	泛型	E190	54.6.5	使用配置文件的 服务器代码	E222
53.14	LINQ 查询	E192	54.6.6	使用配置文件的 客户端代码	E223
53.15	C++/CLI 混合本地代码 和托管代码	E192	54.6.7	客户端信道的延迟 加载	E223
53.16	C#的特殊功能	E193	54.6.8	调试配置	E224
53.17	小结	E194	54.6.9	配置文件中的生命 周期服务	E224
第 54 章	.NET Remoting	E195	54.6.10	格式化程序提供程序	E225
54.1	使用.NET Remoting 的 原因	E195	54.7	在 ASP.NET 中驻留远程 服务器	E225
54.2	.NET Remoting 术语详解	E197	54.8	类、接口和 Soapsuds	E227
54.2.1	客户端通信	E198	54.8.1	接口	E227
54.2.2	服务器端通信	E199	54.8.2	Soapsuds	E227
54.3	上下文	E199	54.9	异步远程调用	E228
54.3.1	激活	E200	54.9.1	使用委托和 .NET Remoting	E228
54.3.2	特性和属性	E200	54.9.2	OneWay 特性	E229
54.3.3	上下文之间的通信	E201	54.10	.NET Remoting 的 安全性	E229
54.4	远程对象、客户端和 服务器	E201			
54.4.1	远程对象	E201			
54.4.2	简单的服务器应用 程序	E202			
54.4.3	简单的客户端应用 程序	E203			

54.11 远程处理和事件.....	E230	56.3.2 通过自定义对象和 LINQ 查询.....	E275
54.11.1 远程对象.....	E231	56.3.3 通过查询限制所调用 的列.....	E276
54.11.2 事件参数.....	E232	56.3.4 使用列名.....	E277
54.11.3 服务器.....	E233	56.3.5 创建自己的 DataContext 对象.....	E278
54.11.4 服务器配置文件.....	E233	56.4 自定义对象和 O/R 设计器.....	E279
54.11.5 事件接收器.....	E234	56.5 查询数据库.....	E280
54.11.6 客户端.....	E234	56.5.1 使用查询表达式.....	E280
54.11.7 客户端配置文件.....	E235	56.5.2 查询表达式.....	E281
54.11.8 运行程序.....	E236	56.5.3 使用表达式筛选.....	E282
54.12 调用上下文.....	E236	56.5.4 执行连接.....	E282
54.13 小结.....	E237	56.5.5 分组项.....	E284
第 55 章 Web 服务和 ASP.NET.....	E239	56.6 存储过程.....	E285
55.1 SOAP.....	E240	56.7 小结.....	E286
55.2 WSDL.....	E241	第 57 章 WPF 3.0.....	E287
55.3 Web 服务.....	E242	57.1 Hello World 示例.....	E287
55.3.1 提供 Web 服务.....	E242	57.2 活动.....	E288
55.3.2 使用 Web 服务.....	E246	57.2.1 IfElseActivity.....	E289
55.4 扩充事件登记示例.....	E248	57.2.2 ParallelActivity.....	E290
55.4.1 事件登记 Web 服务.....	E249	57.2.3 CallExternalMethod Activity.....	E291
55.4.2 事件登记客户端.....	E253	57.2.4 DelayActivity.....	E291
55.5 使用 SOAP 标题交换 数据.....	E256	57.2.5 ListenActivity.....	E292
55.6 小结.....	E261	57.2.6 活动执行模型.....	E293
第 56 章 LINQ to SQL.....	E263	57.3 自定义活动.....	E293
56.1 LINQ to SQL 和 Visual Studio 2010.....	E264	57.3.1 活动的验证.....	E295
56.1.1 调用 Products 表.....	E265	57.3.2 主题和设计器.....	E296
56.1.2 添加 LINQ to SQL 类.....	E265	57.3.3 ActivityToolboxItem 和图标.....	E297
56.1.3 O/R 设计器概述.....	E266	57.3.4 自定义复合活动.....	E299
56.1.4 创建 Product 对象.....	E267	57.4 工作流.....	E305
56.2 对象如何映射到 LINQ 对象上.....	E269	57.4.1 顺序工作流.....	E305
56.2.1 DataContext 对象.....	E270	57.4.2 状态机工作流.....	E305
56.2.2 Table<TEntity>对象.....	E274	57.4.3 给工作流传递参数.....	E307
56.3 脱离 O/R 设计器工作.....	E274	57.4.4 从工作流中返回结果.....	E308
56.3.1 创建自己的自定义 对象.....	E274		

57.4.5 将参数绑定到活动上.....	E309	57.10.1 把活动代码提取到 服务中.....	E320
57.5 workflow运行库.....	E310	57.10.2 删除代码活动.....	E320
57.6 workflow服务.....	E310	57.10.3 同时运行 WF 3.x 和 4.....	E321
57.6.1 持久性服务.....	E312	57.10.4 考虑把状态机迁移到 流程图上.....	E321
57.6.2 跟踪服务.....	E313	57.11 小结.....	E321
57.6.3 自定义服务.....	E315		
57.7 与 WCF 集成.....	E316		
57.8 驻留 workflow.....	E318		
57.9 workflow设计器.....	E319		
57.10 从 WF 3.X 迁移到 WF 4.....	E320		

第 I 部分

C# 语言

- 第 1 章 .NET 体系结构
- 第 2 章 核心 C#
- 第 3 章 对象和类型
- 第 4 章 继承
- 第 5 章 泛型
- 第 6 章 数组
- 第 7 章 运算符和类型强制转换
- 第 8 章 委托、Lambda 表达式和事件
- 第 9 章 字符串和正则表达式
- 第 10 章 集合
- 第 11 章 LINQ
- 第 12 章 动态语言扩展
- 第 13 章 内存管理和指针
- 第 14 章 反射
- 第 15 章 错误和异常

第 1 章

.NET 体系结构

本章内容:

- 编译和运行面向.NET 的代码
- Microsoft 中间语言(Microsoft Intermediate Language, MSIL 或简称为 IL)的优点
- 值类型和引用类型
- 数据类型化
- 理解错误处理和特性
- 程序集、.NET 基类和名称空间

整本书都将强调, C#语言不能孤立地使用, 而必须和 .NET Framework 一起考虑。C#编译器专门用于.NET, 这表示用 C#编写的所有代码总是在 .NET Framework 中运行。对于 C#语言来说, 可以得出两个重要的结论:

- (1) C#的结构和方法论反映了.NET 基础方法论。
- (2) 在许多情况下, C#的特定语言功能取决于.NET 的功能, 或依赖于.NET 基类。

由于这种依赖性, 在开始编写 C#程序前, 了解.NET 的体系结构和方法论就非常重要, 这就是本章的目的所在。

1.1 C#与.NET 的关系

C#是一种相当新的编程语言, C#的重要性体现在以下两个方面:

- 它是专门为与 Microsoft 的 .NET Framework 一起使用而设计的(.NET Framework 是一个功能非常丰富的平台, 可开发、部署和执行分布式应用程序)。
- 它是一种基于现代面向对象设计方法的语言, 在设计它时, Microsoft 还吸取了其他所有类似语言的经验, 这些语言是近 20 年来面向对象规则得到广泛应用后才开发出来的。

有一个很重要的问题要弄明白: C#就其本身而言只是一种语言, 尽管它是用于生成面向.NET 环境的代码, 但它本身不是.NET 的一部分。.NET 支持的一些特性, C#并不支持。而 C#语言支持另一些特性, .NET 却不支持(如运算符重载)!

但是, 因为 C#语言和.NET 一起使用, 所以如果要使用 C#高效地开发应用程序, 理解 Framework 就非常重要, 所以本章将介绍.NET 的内涵。

1.2 公共语言运行库

.NET Framework 的核心是其运行库执行环境,称为公共语言运行库(CLR)或.NET 运行库。通常将在 CLR 控制下运行的代码称为托管代码(managed code)。

但是,在 CLR 执行编写好的源代码(在 C#中或其他语言中编写的代码)之前,需要编译它们。在.NET 中,编译分为两个阶段:

- (1) 把源代码编译为 Microsoft 中间语言(IL)。
- (2) CLR 把 IL 编译为平台专用的代码。

这个两阶段的编译过程非常重要,因为 Microsoft 中间语言是提供.NET 的许多优点的关键。

Microsoft 中间语言与 Java 字节码共享一种理念:它们都是低级语言,语法很简单(使用数字代码,而不是文本代码),可以非常快速地转换为本地机器码。对于代码,这种精心设计的通用语法有很重要的优点:平台无关性、提高性能和语言的互操作性。

1.2.1 平台无关性

首先,这意味着包含字节码指令的同一文件可以放在任一平台中,运行时编译过程的最后阶段可以很轻松地完成,这样代码就可以运行在特定的平台上。换言之,编译为中间语言就可以获得.NET 平台无关性,这与编译为 Java 字节码就会得到 Java 平台无关性是一样的。

注意.NET 的平台无关性目前只是停留在理论范畴,因为在编写本书时,.NET 的完整实现只能用于 Windows 平台,但是人们正在积极准备,使它可以用于其他平台(参见 Mono 项目,它用于实现.NET 的开放源代码,参见 <http://www.go-mono.com/>)。

1.2.2 提高性能

前面把 IL 和 Java 做了比较,实际上,IL 比 Java 字节码的作用还要大。IL 总是即时编译的(称为 JIT 编译),而 Java 字节码常常是解释性的。Java 的一个缺点是,在运行应用程序时,把 Java 字节码转换为内部可执行代码的过程会导致性能的损失(但在最近,Java 在某些平台上能进行 JIT 编译)。

JIT 编译器并不是把整个应用程序一次编译完(这样会有很长的启动时间),而是只编译它调用的那部分代码(这是其名称由来)。代码编译过一次后,得到的本地可执行程序就存储起来,直到退出该应用程序为止,这样在下次运行这部分代码时,就不需要重新编译了。Microsoft 认为这个过程要比一开始就编译整个应用程序代码的效率高得多,因为任何应用程序的大部分代码实际上并不是在每次运行期间都执行。使用 JIT 编译器,从来都不会编译这种代码。

这解释了为什么托管 IL 代码几乎和本地机器代码的执行速度一样快,但是并没有说明为什么 Microsoft 认为这会提高性能。其原因是编译过程的最后一部分是在运行时进行的,JIT 编译器确切地知道程序运行在什么类型的处理器上,可以利用该处理器提供的任何特性或特定的机器代码指令来优化最后的可执行代码。

传统的编译器会优化代码,但它们的优化过程是独立于运行代码的特定处理器的。这是因为传统的编译器是在发布软件之前编译为本地机器可执行的代码。即编译器不知道运行代码的处理器类型,例如该处理器是兼容 x86 的处理器还是 Alpha 处理器,这超出了基本操作的范围。

1.2.3 语言的互操作性

使用 IL 不仅支持平台无关性,还支持语言的互操作性。简而言之,就是能将任何一种语言编译为中间语言,编译为中间语言的代码可以与从其他语言编译过来的代码进行交互操作。

那么除了 C# 之外,还有什么语言可以通过 .NET 进行交互操作呢?下面就简要讨论其他常见语言如何与 .NET 交互操作。

1. Visual Basic 2010

Visual Basic 6 在升级到 Visual Basic .NET 2002 时,经历了一番脱胎换骨的变化,才集成到 .NET Framework 的第 1 版中。Visual Basic 语言对 Visual Basic 6 进行了很大的演化,也就是说,Visual Basic 6 并不适合运行 .NET 程序。例如,它与 COM(Component Object Model, 组件对象模型)的高度集成,并且只把事件处理程序作为源代码显示给开发人员,大多数代码隐藏不能用作源代码。另外,它不支持继承的实现,Visual Basic 6 使用的标准数据类型也与 .NET 不兼容。

Visual Basic 6 在 2002 年升级为 Visual Basic .NET,对 Visual Basic 进行的改变非常大,完全可以把 Visual Basic 当作是一种新语言。已有的 Visual Basic 6 代码不能编译为当前的 Visual Basic 2010 代码(或 Visual Basic .NET 2002、2003、2005 和 2008 代码),把 Visual Basic 6 程序转换为 Visual Basic 2010 时,需要对代码进行大量的改动。但大多数修改工作都可以由 Visual Studio 2010(Visual Studio 的升级版本,用于与 .NET 一起使用)自动完成。如果把 Visual Basic 6 项目读到 Visual Studio 2010 中,Visual Studio 2010 就会自动升级该项目,也就是说把 Visual Basic 6 源代码重写为 Visual Basic 2010 源代码。虽然这意味着其中的工作大大减轻,但用户仍需要检查新的 Visual Basic 2010 代码,以确保项目仍可按预期方式正确工作,因为这种转换并不能达到完美无缺的程度。

这种语言升级的一个副作用是不能再把 Visual Basic 2010 编译为本地可执行代码了。Visual Basic 2010 只编译为中间语言,就像 C# 一样。如果需要继续使用 Visual Basic 6 编写程序,就可以这么做,但生成的可执行代码会完全忽略 .NET Framework,如果继续把 Visual Studio 作为开发环境,就需要安装 Visual Studio 6。

2. Visual C++ 2010

Visual C++ 6 有许多 Microsoft 对 Windows 的特定扩展。Visual C++ .NET 又新增了更多的扩展内容,来支持 .NET Framework。现有的 C++ 源代码会继续编译为本地可执行代码,而不会有修改,但它会独立于 .NET 运行库运行。如果让 C++ 代码在 .NET Framework 中运行,就可以在代码的开头添加下述命令:

```
#using <mscorlib.dll>
```

还可以把标记 /clr 传递给编译器,这样编译器假定要编译托管代码,因此会生成中间语言,而不是本地机器码。C++ 的一个有趣的问题是在编译成托管代码时,编译器可以生成包含内嵌本地可执行程序的 IL。这表示在 C++ 代码中可以把托管类型和非托管类型合并起来,因此托管 C++ 代码:

```
class MyClass
{
```

定义了一个普通的 C++ 类,而代码:

```
ref class MyClass  
{
```

生成了一个托管类,就好像使用 C#或 Visual Basic 2010 编写类一样。实际上,托管 C++代码比 C#代码更优越的一点是可以在托管 C++代码中调用非托管 C++类,而不必采用 COM 互操作功能。

如果在托管类型上试图使用 .NET 不支持的特性(例如,模板或类的多继承),编译器就会出现一个错误。另外,在使用托管类时,还需要使用非标准 C++功能。

因为 C++允许低级指针操作,C++编译器不能生成可以通过 CLR 内存类型安全测试的代码。如果 CLR 把代码标识为内存类型安全是非常重要的,就需要使用其他一些语言编写源代码(如 C#或 Visual Basic 2010)。

3. COM 和 COM+

从技术上讲,COM 和 COM+并不是面向 .NET 的技术,因为基于它们的组件不能编译为 IL(但如果原来的 COM 组件是用 C++编写的,那么使用托管 C++在某种程度上可以这么做)。但是,COM+ 仍然是一个重要工具,因为其特性没有在 .NET 中完全实现。另外,COM 组件仍可以使用——.NET 集成了 COM 的互操作性,从而使托管代码可以调用 COM 组件,COM 组件也可以调用托管代码(见第 26 章)。一般情况下,把新组件编写为 .NET 组件,大多是为了方便,因为这样可以利用 .NET 基类和托管代码的其他优点。

1.3 中间语言

如前所述,Microsoft 中间语言显然在 .NET Framework 中起着非常重要的作用。现在应详细讨论一下 IL 的主要特征,因为面向 .NET 的所有语言在逻辑上都需要支持 IL 的主要特征。

下面就是中间语言的主要特征:

- 面向对象和使用接口
- 值类型和引用类型之间的显著差异
- 强数据类型化
- 使用异常来处理错误
- 使用特性(attribute)

下面详细讨论这些特征。

1.3.1 面向对象和接口的支持

.NET 的语言无关性还有一些实际的限制。中间语言在设计时就打算实现某些特殊的编程方法,这表示面向它的语言必须与编程方法兼容,Microsoft 为 IL 选择的特定道路是传统的面向对象的编程,带有类的单一继承性。



不熟悉面向对象概念的读者应参考第 53 章(见随书附赠光盘中对应的章节),获得更多的信息。

除了传统的面向对象编程外,中间语言还引入了接口的概念,在带有 COM 的 Windows 下第一次实现了接口。用 .NET 建立的接口与 COM 接口不同,它们不需要支持任何 COM 基础结构,例如,它们不是派生自 IUnknown,也没有对应的 GUID。但它们与 COM 接口共享下述理念:提供一个契约,实现给定接口的类必须提供该接口指定的方法和属性的实现方式。

前面介绍了使用 .NET 意味着要编译为中间语言,即需要使用传统的面向对象的方法来编程。但这并不能提供语言的互操作性。毕竟, C++ 和 Java 都使用相同的面向对象的范例,但它们仍不是可交互操作的语言。下面需要详细探讨一下语言互操作性的概念。

首先,需要了解一下语言互操作性的准确含义。

毕竟, COM 允许以不同语言编写的组件一起工作,即可以调用彼此的方法。这就足够了吗? COM 是一个二进制标准,允许组件实例化其他组件,调用它们的方法或属性,而无须考虑编写相关组件的语言。但为了实现这个功能,每个对象都必须通过 COM 运行库来实例化,通过接口来访问。根据相关组件的线程模型,需要在不同线程的内存空间和运行组件之间编组数据,这可能造成很大的性能损失。在极端情况下,组件保存为可执行文件,而不是 DLL 文件,还必须创建单独的进程来运行它们。重要的是组件仅能通过 COM 运行库与其他组件通信。使用不同语言的组件无法通过 COM 直接彼此通信,或者创建彼此的实例——系统总将 COM 作为中间件。不仅如此, COM 体系结构还不允许利用继承实现,即它丧失了面向对象编程的许多优势。

一个相关的问题是,在调试时,仍必须单独调试使用不同语言编写的组件。不可能在调试器上交替调试不同语言的代码。语言互操作性的真正含义是用一种语言编写的类应能直接与用另一种语言编写的类通信。特别是:

- 用一种语言编写的类应能继承用另一种语言编写的类。
- 一个类应能包含另一个类的实例,而不管两个类是使用什么语言编写的。
- 一个对象应能直接调用用其他语言编写的另一个对象的方法。
- 对象(或对象的引用)应能在方法之间传递。
- 在不同的语言之间调用方法时,应能在调试器中交替调试这些方法调用,即调试不同语言编写的源代码。

这是一个雄心勃勃的目标,但令人惊讶的是, .NET 和中间语言已经实现了这个目标。在调试器上交替调试方法时, Visual Studio IDE(Integrated Development Environment, 集成开发环境)提供了这样的工具(不是 CLR 提供的)。

1.3.2 不同的值类型和引用类型

与其他编程语言一样,中间语言提供了许多预定义的基本数据类型。它的一个特性是值类型和引用类型之间有明显的区别。对于值类型(value type),变量直接存储其数据,而对于引用类型(reference type),变量仅存储地址,对应的数据可以在该地址中找到。

在 C++ 中,使用引用类型类似于通过指针来访问变量,而在 Visual Basic 中,与引用类型最相似的是对象, Visual Basic 6 总是通过引用来访问对象。中间语言也有数据存储的规范:引用类型的实例总是存储在一个名为“托管堆”的内存区域中,值类型一般存储在堆栈中(但如果值类型在引用类型中声明为字段,它们就内联存储在堆中)。第 2 章讨论堆栈和堆,及其工作原理。

1.3.3 强数据类型化

中间语言的一个重要方面是它基于强数据类型化。所有的变量都清晰地标记为属于某个特定数据类型(在中间语言中没有 Visual Basic 和脚本语言中的 Variant 数据类型)。特别是中间语言一般不允许对模糊的数据类型执行任何操作。

例如, Visual Basic 6 开发人员习惯于传递变量,而无需考虑它们的类型,因为 Visual Basic 6 会自动进行所需的类型转换。C++开发人员习惯于在不同类型之间转换指针类型。执行这类操作将极大地提高性能,但破坏了类型的安全性。因此,这类操作只能在某些编译为托管代码的语言中的特殊情况下进行。确实,指针(相对于引用)只能在标记了的 C#代码块中使用,但在 Visual Basic 中不能使用(但一般在托管 C++中允许使用)。在代码中使用指针会立即导致 CLR 提供的内存类型安全性检查失败。注意,一些与 .NET 兼容的语言,例如 Visual Basic 2010,在类型化方面的要求仍比较宽松,但这是可以的,因为编译器在后台确保在生成的 IL 上强制类型安全。

尽管强迫实现类型的安全性似乎会降低性能,但在许多情况下,我们从 .NET 提供的、依赖于类型安全的服务中获得的好处更多。这些服务包括:

- 语言的互操作性
- 垃圾收集
- 安全性
- 应用程序域

下面讨论强数据类型化对 .NET 的这些功能非常重要的原因。

1. 语言互操作性中强数据类型化的重要性

如果类派生自其他类,或包含其他类的实例,它就需要知道其他类使用的所有数据类型,这就是强数据类型化非常重要的原因。实际上,过去由于缺少用于指定这类信息的一致系统,从而成为语言继承和交互操作的真正障碍。这类信息并未在标准的可执行文件或 DLL 中出现。

假定将 Visual Basic 2010 类中的一个方法定义为返回一个 Integer——Visual Basic 2010 可以使用的标准数据类型之一。但 C#没有该名称的数据类型。显然,只有编译器知道如何把 Visual Basic 2010 的 Integer 类型映射为 C#定义的某种已知类型,才可以从该类派生,使用这个方法,并在 C#代码中使用返回的类型。这个问题在 .NET 中是如何解决的?

通用类型系统(CTS)

此类数据类型问题在 .NET 中使用通用类型系统(CTS)得到了解决。CTS 定义了可以在中间语言中使用的预定义数据类型,所有面向 .NET Framework 的语言都可以生成最终基于这些类型的编译代码。

对于上面的例子,Visual Basic 2010 的 Integer 实际上是一个 32 位有符号的整数,它实际映射为中间语言类型 Int32。因此在中间语言代码中就指定这种数据类型。C#编译器可以使用这种类型,所以就不会有问题了。在源代码中,C#用关键字 int 来表示 Int32,所以编译器就认为 Visual Basic 2010 方法返回一个 int 类型的值。

CTS 不仅指定了基本数据类型,还定义了一个内容丰富的类型层次结构,其中包含设计合理的位置,在这些位置上,代码允许定义它自己的类型。CTS 的层次结构反映了中间语言的单一继承的面向对象方法,如图 1-1 所示。

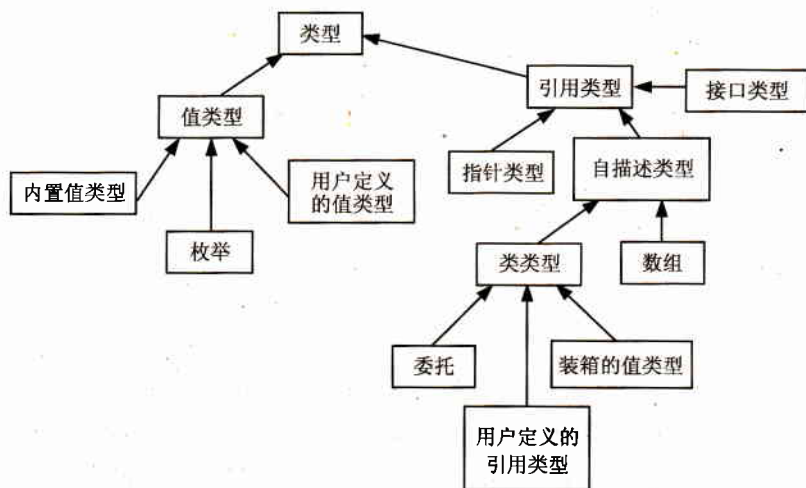


图 1-1

这里没有列出内置的所有值类型，因为第 3 章将详细介绍它们。在 C# 中，编译器识别的每个预定义类型都映射为一个 IL 内置类型。这与 Visual Basic 2010 一样。

公共语言规范(CLS)

公共语言规范(Common Language Specification, CLS)和通用类型系统一起确保语言的互操作性。CLS 是一个最低标准集，所有面向 .NET 的编译器都必须支持它。因为 IL 是一种内涵非常丰富的语言，大多数编译器的编写人员有可能把给定编译器的功能限制为只支持 IL 和 CTS 提供的一部分功能。只要编译器支持已在 CLS 中定义的内容，这就很不错。

下面的一个例子是有关区分大小写字母的。IL 是区分大小写的语言。使用这些语言的开发人员常常利用区分大小写所提供的灵活性来选择变量名。但 Visual Basic 2010 是不区分大小写的语言。CLS 通过指定 CLS 兼容代码不使用任何只根据大小写来区分的名称，解决了这个问题。因此，Visual Basic 2010 代码可以与 CLS 兼容代码一起使用。

这个例子说明了 CLS 的两种工作方式。

(1) 各个编译器的功能不必强大到支持 .NET 的所有功能，这将鼓励人们为其他面向 .NET 的编程语言开发编译器。

(2) 如果限制类只能使用 CLS 兼容的特性，就要保证用其他兼容语言编写的代码可以使用这个类。

这种方法的优点是使用 CLS 兼容特性的限制只适用于公共和受保护的类成员和公共类。在类的私有实现方式中，可以编写非 CLS 代码，因为其他程序集(托管代码的单元，参见本章后面的内容)中的代码不能访问这部分代码。

这里不深入讨论 CLS 规范。一般情况下，CLS 对 C# 代码的影响不会太大，因为 C# 中的非 CLS 兼容特性非常少。



编写非 CLS 兼容代码是完全可以接受的。只是在编写了这种代码后，就不能保证编译好的 IL 代码完全支持语言的互操作性。

2. 垃圾回收

垃圾回收器(garbage collector)用来在.NET 中进行内存管理,特别是它可以恢复正在运行的应用程序需要的内存。到目前为止,Windows 平台已经使用了两种技术来释放进程向系统动态请求的内存:

- 完全以手工方式使用应用程序代码完成这些工作。
- 让对象维护引用计数。

让应用程序代码负责释放内存是低级高性能的语言使用的技术,例如 C++。这种技术很有效,并且一般情况下可以让资源在不需要时就释放,但其最大的缺点是频繁出现错误。请求内存的代码还必须显式通知系统它什么时候不再需要该内存。但这是很容易被遗漏的,从而导致内存泄漏。

尽管现代的开发环境提供了帮助检测内存泄漏的工具,但它们很难跟踪错误,因为直到内存已大量泄漏从而使 Windows 拒绝为进程提供资源时,它们才会发挥作用。到那个时候,由于对内存的需求很大,会使整个计算机变得相当慢。

维护引用计数是 COM 对象采用的一种技术,其方法是每个 COM 组件都保留一个计数,记录客户端目前对它的引用数。当这个计数下降到 0 时,组件就会删除自己,并释放相关的内存和资源。它带来的问题是仍需要客户端通知组件它们已经完成了内存的使用。只要有一个客户端没有这么做,对象就仍驻留在内存中。在某些方面,这是比 C++内存泄漏更为严重的问题,因为 COM 对象可能存在于它自己的进程中,从来不会被系统删除(在 C++内存泄漏问题上,系统至少可以在进程中断时释放所有的内存)。

.NET 运行库采用的方法是垃圾回收器,这是一个程序,其目的是清理内存。方法是所有动态请求的内存都分配到堆上(所有的语言都是这样处理的,但在.NET 中,CLR 维护它自己的托管堆,供.NET 应用程序使用)。每隔一段时间,当.NET 检测到给定进程的托管堆已满,需要清理时,就调用垃圾回收器。垃圾回收器处理目前代码中的所有变量,检查对存储在托管堆上的对象的引用,确定哪些对象可以从代码中访问——即哪些对象有引用。没有引用的对象就不再认为可以从代码中访问,因而被删除。Java 就使用与此类似的垃圾回收系统。

之所以在.NET 中使用垃圾回收器,是因为中间语言已用来处理进程。其规则要求,第一,不能引用已有的对象,除非复制已有的引用。第二,中间语言是类型安全的语言。在这里,其含义是如果存在对对象的任何引用,该引用中就有足够的信息来确定对象的类型。

垃圾回收机制不能和诸如非托管 C++的语言一起使用,因为 C++允许指针自由地转换数据类型。

垃圾回收的一个重要方面是它的不确定性。换言之,不能保证什么时候会调用垃圾回收器:CLR 决定需要它时,就可以调用它。但可以重写这个过程,在代码中调用垃圾回收器。

垃圾回收过程的详细信息可参见第 13 章。

3. 安全性

.NET 很好地弥补了 Windows 提供的安全机制,因为它提供的安全机制是基于代码的安全性,而 Windows 仅提供了基于角色的安全性。

基于角色的安全性建立在运行进程的账户的身份基础上,换言之,就是谁拥有和运行进程。另一方面,基于代码的安全性建立在代码实际执行的任务和代码的可信程度上。由于中间语言提供了

强大的类型安全性,因此 CLR 可以在运行代码前检查它,以确定是否有需要的安全权限。.NET 还提供了一种机制,使代码可以在运行前,预先指定需要什么安全权限。

基于代码的安全性非常重要,原因是它降低了运行来历不明的代码的风险(如代码是从 Internet 上下载的)。即使代码运行在管理员账户下,也有可能使用基于代码的安全性,来确定这段代码是否仍不能执行管理员账户一般允许执行的某些类型的操作,例如读写环境变量、读写注册表或访问 .NET 反射特性。

安全问题详见第 21 章。

4. 应用程序域

应用程序域是 .NET 中的一个重要技术改进,它用于减少运行应用程序的系统开销,这些应用程序需要与其他程序分离开来,但仍需要彼此通信。典型的例子是 Web 服务器应用程序,它需要同时响应许多浏览器请求。因此,要有许多组件实例同时响应这些同时运行的请求。

在 .NET 问世之前,可以让这些实例共享同一个进程,但此时一个运行的实例就有可能导致整个网站的崩溃;也可以把这些实例孤立在不同的进程中,但这样做会增加相关性能的系统开销。

到现在为止,孤立代码的唯一方式是通过进程来实现的。在启动一个新的应用程序时,它会在一个进程环境内运行。Windows 通过地址空间把进程分隔开来。这样,每个进程有 4GB 的虚拟内存来存储其数据和可执行代码(4GB 对应于 32 位系统,64 位系统要用更多的内存)。Windows 利用额外的间接方式把这些虚拟内存映射到物理内存或磁盘空间的一个特殊区域中。每个进程都会有不同的映射,虚拟地址空间块映射的物理内存之间不重叠,如图 1-2 所示。

一般情况下,任何进程都只能通过指定虚拟内存中的一个地址来访问内存——即进程不能直接访问物理内存,因此一个进程不可能访问分配给另一个进程的内存。这样就可以确保任何执行出错的代码不会损害其地址空间以外的数据(注意在 Windows 95/98 上,这些保护措施不像在 Windows NT/2000/XP/2003/Vista/7 上那样完备,所以理论上存在应用程序因写入不正确的内存而导致 Windows 崩溃的可能性)。

进程不仅是运行代码的实例相互隔离的一种方式,在 Windows NT/2000/XP/2003/Vista/7 系统上,它们还可以构成分配了安全权限和许可的单元。每个进程都有自己的安全标识,明确地表示 Windows 允许该进程可以执行的操作。

进程对确保安全有很大的帮助,而它们的一大缺点是性能。许多进程常常在一起工作,因此需要相互通信。一个常见的例子是进程调用一个 COM 组件,而该 COM 组件是可执行的,因此需要在它自己的进程上运行。在 COM 中使用代理时也会发生类似的情况。因为进程不能共享任何内存,所以必须使用一个复杂的编组过程在进程之间复制数据。这对性能有非常大的影响。如果需要使组件一起工作,但不希望性能有损失,唯一的方法是使用基于 DLL 的组件,让所有的组件在同一个地址空间中运行——其风险是执行出错的组件会影响其他组件。

应用程序域(application domain)是分离组件的一种方式,它不会导致因在进程之间传送数据而产生的性能问题。其方法是把任何一个进程分解到多个应用程序域中。每个应用程序域大致对应一个应用程序,执行的每个线程都运行在一个具体的应用程序域中,如图 1-3 所示。

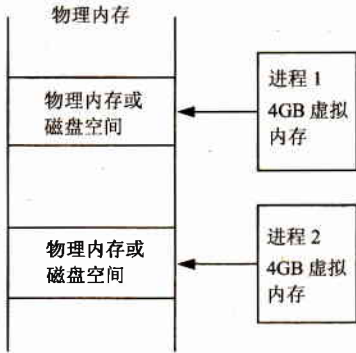


图 1-2

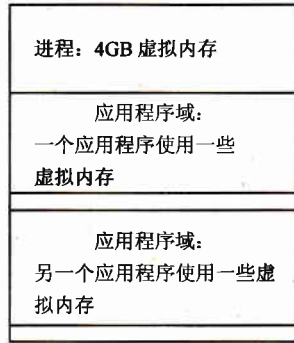


图 1-3

如果不同的可执行文件都运行在同一个进程空间中，显然它们就能轻松地共享数据，因为理论上它们可以直接访问彼此的数据。虽然在理论上这是可以实现的，但是 CLR 会检查每个正在运行的应用程序的代码，以确保这些代码不脱离它自己的数据区域，保证不发生直接访问其他进程的数据的情况。这初看起来是不可能的，不真正运行程序，如何告诉程序要做什么工作？

实际上，这么做通常是可能的，因为中间语言拥有强大的类型安全功能。在大多数情况下，除非代码明确使用不安全的特性，例如指针，否则它使用的数据类型可以确保内存不会被错误地访问。例如，.NET 数组类型执行边界检查，以确保禁止执行超出边界的数组操作。如果运行的应用程序的确需要与运行在不同应用程序域中的其他应用程序通信或共享数据，就必须调用.NET 的远程服务。

被验证不能访问超出其应用程序域的数据(而不是通过明确的远程处理机制)的代码就是内存类型安全的代码。这种代码与运行在同一个进程中但应用程序域不同的类型安全代码一起运行是安全的。

1.3.4 通过异常处理错误

.NET Framework 可以与 Java 和 C++ 使用相同的基于异常的机制处理错误情况。C++ 开发人员应注意到，由于 IL 有非常强大的类型系统，因此在 IL 中不像 C++ 那样存在因使用异常带来的相关性能问题。另外，.NET 和 C# 也支持 `finally` 块，这是许多 C++ 开发人员长久以来的愿望。

第 15 章会详细讨论异常。简要地说，代码的某些领域被看作是异常处理例程，每个例程都能处理某种特殊的错误情况(例如，找不到文件，或拒绝执行某些操作)。这些条件可以定义得很宽或很窄。异常结构确保在发生错误情况时，执行进程立即跳到最有针对性的异常处理例程上，来处理错误情况。

异常处理的结构还提供了一种简便的方式，可以将包含异常情况的准确信息的对象传递给错误处理例程。这个对象包括给用户提供的相应信息和在代码的什么地方检测到错误的确切信息。

大多数异常处理结构，包括异常发生时的程序流控制，都是由高级语言处理的，例如 C#、Visual Basic 2010 和 C++，任何中间语言中的命令都不支持它。例如，C# 使用 `try{}、catch{} 和 finally{} 代码块来处理它，详见第 15 章。`

但.NET 提供了一种基础结构，让面向.NET 的编译器支持异常处理。特别是它提供了一组.NET 类来表示异常，语言的互操作性则允许异常处理代码解释被抛出的异常对象，无论异常处理代码使用什么语言编写，都是这样。语言的无关性没有体现在 C++ 和 Java 的异常处理中，但在 COM 的错误处理机制中有一定限度的体现。COM 的错误处理机制包括从方法中返回错误代码以及传递错误

对象。在不同的语言中，异常的处理是一致的，这是促进多语言开发的重要一环。

1.3.5 特性的使用

特性(attribute)是使用 C++编写 COM 组件的开发人员很熟悉的一个功能(在 Microsoft 的 COM 接口定义语言(Interface Definition Language, IDL)中使用特性)。特性最初是为了在程序中提供与某些项相关的额外信息，以供编译器使用。

.NET 支持特性，因此现在 C++、C#和 Visual Basic 2010 也支持特性。但在 .NET 中，对特性的革新是可以在源代码中定义自己的自定义特性。这些用户定义的特性将和对应数据类型或方法的元数据放在一起，这对于文档记录十分有用，它们和反射技术一起使用，以根据特性执行编程任务。另外，与 .NET 的语言无关性的基本原理一样，特性也可以在一种语言的源代码中定义，而被用另一种语言编写的代码读取。

第 14 章将详细介绍特性。

1.4 程序集

程序集(assembly)是包含编译好的、面向 .NET Framework 的代码的逻辑单元。本章不详细论述程序集，而在第 18 章中论述，下面概述其中的要点。

程序集是完全自描述性的，它是一个逻辑单元而不是物理单元，可以存储在多个文件中(动态程序集的确存储在内存中，而不是存储在文件中)。如果一个程序集存储在多个文件中，其中就会有一个包含入口点的主文件，该文件描述了程序集中的其他文件。

注意可执行代码和库代码使用相同的程序集结构。唯一的区别是可执行的程序集包含一个主程序入口点，而库程序集不包含。

程序集的一个重要特性是它们包含的元数据描述了对应代码中定义的类型和方法。程序集也包含描述程序集本身的元数据，这种程序集元数据包含在一个称为“清单(manifest)”的区域中，可以检查程序集的版本及其完整性。



ildasm 是一个基于 Windows 的实用程序，可以用于检查程序集的内容，包括程序集清单和元数据。第 18 章将介绍 ildasm。

程序集包含程序的元数据，表示调用给定程序集中的代码的应用程序或其他程序集不需要引用注册表或其他数据源，就能确定如何使用该程序集。这与以前的 COM 有很大的区别，以前，组件和接口的 GUID 必须从注册表中获取，在某些情况下，方法和属性的详细信息也需要从类型库中读取。

把数据分散在 3 个以上的不同位置上，可能会出现信息不同步的情况，从而妨碍其他软件成功地使用该组件。有了程序集后，就不会发生这种情况，因为所有的元数据都与程序的可执行指令存储在一起。注意，即使程序集存储在几个文件中，数据也不会出现不同步的问题。这是因为包含程序集入口的文件也存储了其他文件的细节、散列和内容，如果一个文件被替换，或者被篡改，系统肯定会检测出来，并拒绝加载程序集。

程序集有两种类型：私有程序集和共享程序集。

1.4.1 私有程序集

私有程序集是最简单的一种程序集类型。私有程序集一般附带在某个软件上,且只能用于该软件。附带私有程序集的常见情况是,以可执行文件或许多库的方式提供应用程序,这些库包含的代码只能用于该应用程序。

系统可以保证私有程序集不被其他软件使用,因为应用程序只能加载位于主执行文件所在文件夹或其子文件夹中的程序集。

用户一般会希望把商用软件安装在它自己的目录下,这样软件包不存在覆盖、修改或在无意间加载另一个软件包的私有程序集的风险。私有程序集只能用于自己的软件包,这样,用户对什么软件使用它们就有了更大的控制权。因此,不需要采取安全措施,因为这没有其他商用软件用某个新版本的程序集覆盖原来的私有程序集的风险(但软件是专门执行怀有恶意的损害性操作的情况除外)。名称也不会有冲突。如果私有程序集中的类正巧与另一个人的私有程序集中的类同名,是不会有问题的,因为给定的应用程序只能使用它自己的一组私有程序集。

因为私有程序集完全是自包含的,所以部署它的过程就很简单。只需把相应的文件放在文件系统的对应文件夹中即可(不需要注册表项),这个过程称为“0 影响(xcopy)安装”。

1.4.2 共享程序集

共享程序集是其他应用程序可以使用的公共库。因为其他软件可以访问共享程序集,所以需要采取一定的保护措施来防止以下风险:

- 名称冲突,另一个公司的共享程序集执行的类型与自己的共享程序集中的类型同名。因为客户端代码理论上可以同时访问这些程序集,所以这是一个严重的问题。
- 程序集被同一个程序集的不同版本覆盖——新版本与某些已有的客户端代码不兼容。

这些问题的解决方法是把共享程序集放在文件系统的一个特定的子目录树中,称为全局程序集缓存(GAC)。与私有程序集不同,不能简单地把共享程序集复制到对应的文件夹中,而需要专门安装到缓存中,可以用许多.NET 工具完成这个过程,其中包含对程序集的检查、在程序集缓存中设置一个小的文件夹层次结构,以确保程序集的完整性。

为了避免名称冲突,应根据私钥加密法为共享程序集指定一个名称(而对于私有程序集,只需要指定与其主文件名相同的名称即可)。该名称称为强名(strong name),并保证其唯一性,它必须由要引用共享程序集的应用程序来引用。

与覆盖程序集的风险相关的问题,可以通过在程序集清单中指定版本信息来解决,也可以通过同时安装来解决。

1.4.3 反射

因为程序集存储了元数据,包括在程序集中定义的所有类型和这些类型的成员的细节,所以可以编程访问这些元数据。这个技术称为反射,第 14 章详细介绍了它们。该技术很有趣,因为它表示托管代码实际上可以检查其他托管代码,甚至检查它自己,以确定该代码的信息。它们常常用于获取特性的详细信息,也可以把反射用于其他目的,例如作为实例化类或调用方法的一种间接方式(前提是将这些类或方法的名称指定为字符串)。这样,就可以选择类来实例化方法,以便在运行时调用,

而不是在编译时调用,例如根据用户的输入来调用(动态绑定)。

1.4.4 并行编程

.NET Framework 4 允许利用目前出现的所有双处理器和 4 处理器。新的并行计算能力提供了分隔工作活动、并在多个处理器上运行这些活动的方式。现在可用的、新的并行编程 API 使得编写安全的多线程代码变得十分简单,但要注意,仍需要考虑竞态条件和锁。

新的并行编程功能提供了一个新的 Task Parallel Library 和 PLINQ Execution Engine,并行编程的详细内容请参见第 20 章。

1.5 .NET Framework 类

至少从开发人员的角度来看,编写托管代码的最大好处是可以使用 .NET 基类库。 .NET 基类是一个内容丰富的托管代码类集合,它可以完成以前要通过 Windows API 来完成的绝大多数任务。这些类沿用中间语言使用的对象模型,也基于单一继承性。可以从任何适用的 .NET 基类实例化对象,也可以从它们派生自己的类。

.NET 基类的一个主要优点是它们非常直观和易用。例如,要启动一个线程,可以调用 Thread 类的 Start()方法。要禁用 TextBox,应把 TextBox 对象的 Enabled 属性设置为 false。 Visual Basic 和 Java 开发人员非常熟悉这种方式。它们的库都很容易使用,但对于 C++开发人员这是极大的解脱,因为他们多年来一直在使用诸如 GetDIBits()、RegisterWndClassEx()和 IsEqualIID()这样的 API 函数,以及大量需要传递的 Windows 句柄的函数。

另一方面,C++开发人员总是很容易访问整个 Windows API,而 Visual Basic 6 和 Java 开发人员只能访问其语言所能访问的基本操作系统功能。 .NET 基类的新增内容就是把 Visual Basic 和 Java 库的易用性和 Windows API 函数较为丰富的功能结合起来。但 Windows 仍有许多功能不能通过基类来使用,而需要调用 API 函数。但一般情况下,这只限于比较复杂的特性。基类库足以应付日常工作的使用。如果需要调用 API 函数, .NET 提供了所谓的“平台调用”,来确保对数据类型进行正确的转换,这样无论是使用 C#、C++或 Visual Basic 2010 进行编码,该任务都不会比直接从已有的 C++代码中调用函数更困难。



WinCV 是一个基于 Windows 的实用程序,它可以用于浏览基类库中的类、结构、接口和枚举。第 16 章将介绍 WinCV。

第 3 章主要介绍基类。概述了 C#语言语法后,本书的其余内容将主要说明如何使用 .NET Framework 4 的 .NET 基类库中的各种类,即各种基类是如何工作的。 .NET 3.5 基类大致包括以下范围:

- IL 提供的核心功能(例如,通用类型系统中的基本数据类型,详见第 3 章)
- Windows GUI 支持和控件(参见第 35 章和第 39 章)
- Web 窗体(ASP.NET,参见第 40 章和第 41 章)
- 数据访问(ADO.NET,参见第 30 章、第 33 章和第 34 章)
- 目录访问(见随书附赠光盘中的第 52 章)

- 文件系统和注册表访问(参见第 29 章)
- 网络和 Web 浏览(参见第 24 章)
- .NET 特性和反射(参见第 14 章)
- 访问 Windows 操作系统的各个方面(如环境变量等, 参见第 21 章)
- COM 互操作性(参见第 26 章和第 51 章)

附带说一下, 根据 Microsoft 源文件, 大部分 .NET 基类实际上都是用 C# 编写的!

1.6 名称空间

名称空间是 .NET 避免类名冲突的一种方式。例如, 名称空间可以避免下述情况: 定义一个类来表示一个顾客, 称此类为 `Customer`, 同时其他人也在做相同的事(很可能出现这种情况, 拥有客户的企业所占的比例很高)。

名称空间不过是数据类型的一种组合方式, 但名称空间中所有数据类型的名称都会自动加上该名称空间的名字作为其前缀。名称空间还可以相互嵌套。例如, 大多数用于一般目的的 .NET 基类位于名称空间 `System` 中, 基类 `Array` 在这个名称空间中, 所以其全名是 `System.Array`。

.NET 需要在名称空间中定义所有的类型, 例如, 可以把 `Customer` 类放在名称空间 `YourCompanyName` 中, 则这个类的全名就是 `YourCompanyName.Customer`。



如果没有显式提供名称空间, 类型就添加到一个没有名称的全局名称空间中。

在大多数情况下, Microsoft 建议都至少要提供两个嵌套的名称空间名, 第一个是公司名, 第二个是技术名称或软件包的名称, 而类是其中的一个成员, 例如 `YourCompanyName.SalesServices.Customer`。大多数情况下, 这么做可以保证类名不会与其他组织编写的类名冲突。

第 2 章将详细介绍名称空间。

1.7 用 C# 创建 .NET 应用程序

C# 可以用于创建控制台应用程序: 仅使用文本、运行在 DOS 窗口中的应用程序。在进行单元测试类库、创建 UNIX/Linux 守护进程时, 就要使用控制台应用程序。但是, 我们常使用 C# 创建利用许多与 .NET 相关的技术的应用程序, 下面简要论述可以用 C# 创建的不同类型的应用程序。

1.7.1 创建 ASP.NET 应用程序

最初引入的 ASP.NET 1.0 基本改变了 Web 编程模型。ASP.NET 4 是该产品的一个主要版本, 它建立在以前改进的基础之上。ASP.NET 4 采取了一系列重要的革新步骤, 来提高效率。ASP.NET 的主要目标是使用最少的代码建立强大、安全、动态的应用程序。由于本书是关于 C# 的, 所以有许多章节介绍了如何使用这种语言建立最新的 Web 应用程序。

下面讨论 ASP.NET 的重要功能, 详细信息参见第 40~42 章。

1. ASP.NET 的功能

首先, 也是最重要的是, ASP.NET 页面是结构化的。这就是说, 每个页面都是一个继承了 .NET 类 `System.Web.UI.Page` 的类, 可以改写在 `Page` 对象的生存期中调用的一系列方法, (可以把这些事件看成是页面所特有的, 对应于原 ASP 的 `global.asa` 文件中的 `OnApplication_Start` 和 `OnSession_Start` 事件)。因为可以把一个页面的功能放在有明确含义的事件处理程序中, 所以 ASP.NET 比较容易理解。

ASP.NET 页面的另一个优点是可以在 Visual Studio 2010 中创建它们, 在该环境下, 可以创建 ASP.NET 页面使用的业务逻辑和数据访问组件。Visual Studio 2010 项目(也称为解决方案)包含了与应用程序相关的所有文件。而且, 也可以在编辑器中调试传统的 ASP 页面, 在以前使用 Visual InterDev 时, 把 InterDev 和项目的 Web 服务器配置为支持调试常常是一个让人头痛的问题。

最清楚的是, ASP.NET 的代码隐藏功能允许进一步采用结构化的方式。ASP.NET 允许把页面的服务器端功能单独放在一个类中, 随其他页面把该类编译为 DLL, 并把该 DLL 放在 HTML 部分下面的一个目录中。放在页面顶部的 `@Page` 指令将把该文件与一个类关联起来。当浏览器请求该页面时, Web 服务器就会在页面的类文件中引发类中的事件。

最后, ASP.NET 可以显著提高性能。传统的 ASP 页面和每个页面请求一起解释, 而 Web 服务器在编译后缓存 ASP.NET 页面。这表示以后对 ASP.NET 页面的请求就比 ASP 页面第一次执行的速度快得多。

ASP.NET 还易于编写通过浏览器显示窗体的页面, 这在内联网环境中会使用。传统的方式是基于窗体的应用程序提供一个功能丰富的用户界面, 但较难维护, 因为它们运行在非常多的不同计算机上。因此, 当用户界面是必不可少的, 并可以为用户提供扩展支持时, 人们就会依赖基于窗体的应用程序。

2. Web 窗体

为了简化 Web 页面的结构, Visual Studio 2010 提供了 Web 窗体。它们允许以创建 Visual Basic 6 或 C++ Builder 窗口的方式图形化地建立 ASP.NET 页面; 换言之, 就是把控件从工具箱拖放到窗体上, 再考虑窗体的代码, 为控件编写事件处理程序。在使用 C# 创建 Web 窗体时, 就是创建一个继承自 `Page` 基类的 C# 类, 并把这个类看作是代码隐藏的 ASP.NET 页面。当然不必使用 C# 创建 Web 窗体, 而可以使用 Visual Basic 2010 或另一种 .NET 语言来创建。

过去, Web 开发的难度使一些开发小组不愿意使用 Web。为了成功地进行 Web 开发, 必须了解非常多的不同技术, 如 VBScript、ASP、DHTML、JavaScript 等。把窗体概念应用于 Web 页面, Web 窗体就可以简化 Web 开发。

3. Web 服务器控件

用于添加到 Web 窗体上的控件与 ActiveX 控件并不是同一种控件, 它们是 ASP.NET 名称空间中的 XML 标记。当请求一个页面时, Web 浏览器会动态地把它们转换为 HTML 和客户端脚本。Web 服务器能以不同的方式显示相同的服务器端控件, 产生一个对应于请求者特定 Web 浏览器的转换。这意味着现在很容易为 Web 页面编写相当复杂的用户界面, 而不必担心如何确保页面运行在可用的任何浏览器上, 因为 Web 窗体会完成这些任务。

可以使用 C#或 Visual Basic 2010 扩展 Web Form 工具箱。创建一个新服务器端控件, 仅是执行.NET 的 System.Web.UI.WebControls.WebControl 类而已。

4. XML Web 服务

目前, HTML 页面解决了 World Wide Web 上的大部分通信问题。有了 XML, 计算机就可以用一种独立于设备的格式, 在 Web 上彼此通信。将来, 计算机可以使用 Web 和 XML 交流信息, 而不是专用的线路和专用的格式, 如 EDI (Electronic Data Interchange)。XML Web 服务是为面向服务的 Web, 即远程计算机彼此提供可以分析和重新格式化的动态信息, 最后显示给用户。XML Web 服务是计算机给 Web 上的其他计算机以 XML 格式显示信息的一种便利方式。

在技术上, .NET 上的 XML Web 服务是给请求的客户返回 XML 而不是 HTML 的 ASP.NET 页面。这种页面有代码隐藏的 DLL, 它包含了派生自 WebService 类的类。Visual Studio 2010 IDE 提供的引擎简化了 Web 服务的开发。

公司选择使用 XML Web 服务主要有两个原因。首先是因为它们依赖于 HTTP, 而 XML Web 服务可以把现有的网络(HTTP)用作传输信息的媒介。其次是因为 XML Web 服务使用 XML, 该数据格式是自描述的、非专用的且独立于平台。

1.7.2 创建 Windows 窗体

C#和.NET 非常适合于 Web 开发, 它们还为所谓的“胖客户端”或“瘦客户端”应用程序提供了极好的支持, 这种“胖客户端”或“瘦客户端”应用程序必须安装在最终用户的计算机上, 来处理大多数操作, 这种支持来源于 Windows 窗体。

Windows 窗体是 Visual Basic 6 窗体的.NET 版本, 要设计一个图形窗口界面, 只需把控件从工具箱拖放到 Windows 窗体上即可。要确定窗口的行为, 应为该窗体的控件编写事件处理例程。Windows Form 项目编译为可执行文件, 可执行文件必须与.NET 运行库一起安装在最终用户的计算机上。与其他.NET 项目类型一样, Visual Basic 2010 和 C#都支持 Windows Form 项目。第 39 章将详细介绍 Windows 窗体。

1.7.3 使用 WPF

有一种最新的技术叫做 Windows Presentation Foundation(WPF)。WPF 在建立应用程序时使用 XAML。XAML 表示可扩展的应用程序标记语言(Extensible Application Markup Language)。这种在 Microsoft 环境下创建应用程序的新方式在 2006 年引入, 是 .NET Framework 3.0、3.5 和 4 的一部分。要运行 WPF 应用程序, 需要在客户机上安装 .NET Framework 3.0、3.5 或 4。WPF 应用程序可用于 Windows 7、Windows Vista、Windows XP、Windows Server 2003 和 Windows Server 2008(只有这些操作系统能安装 .NET Framework 3.0、3.5 或 4)。

XAML 是用于创建窗体的 XML 声明, 它代表 WPF 应用程序的所有可视化部分和操作。虽然可以编程利用 WPF 应用程序, 但 WPF 是迈向声明性编程的一步, 而声明性编程是编程业的趋势。声明性编程是指, 不是利用编译语言, 如 C#、VB 或 Java, 通过编程来创建对象, 而是通过 XML 类型的编程来声明所有元素。第 35 章详细介绍了如何使用 XAML 和 C#建立这些新类型的应用程序。

1.7.4 Windows 控件

虽然 Web 窗体和 Windows 窗体的开发方式基本相同,但应为它们添加不同类型的控件。Web 窗体使用 Web 服务器控件,Windows 窗体使用 Windows 控件。

Windows 控件比较类似于 ActiveX 控件。在执行 Windows 控件后,它会编译为必须安装到客户机上的 DLL。实际上,.NET SDK 提供了一个实用程序,为 ActiveX 控件创建包装器,以便把它们放在 Windows 窗体上。与 Web 控件一样,Windows 控件的创建需要派生于特定的类 `System.Windows.Forms.Control`。

1.7.5 Windows 服务

Windows 服务(最初称为 NT 服务)是一个在 Windows NT/2000/XP/2003/Vista/7(但没有 Windows 9x)后台运行的程序。当希望程序连续运行,并在用户没有明确启动操作时响应事件,就应使用 Windows 服务。例如 Web 服务器上的 World Wide Web 服务,它们监听来自客户的 Web 请求。

用 C#编写服务非常简单。`System.ServiceProcess` 名称空间中的 .NET Framework 基类可以处理许多与服务相关的样本任务。另外,Visual Studio 2010 允许创建 C# Windows Service 项目,为基本 Windows 服务编写 C#源代码。第 25 章将详细介绍如何编写 C# Windows 服务。

1.7.6 WCF

通过基于 Microsoft 的技术,可以采用许多方式将数据和服务从一处移动到另一处。例如,可以使用 ASP.NET Web 服务、.NET Remoting、Enterprise Services 和用于初学者的 MSMQ。应采用哪种技术?这要考虑具体要达到的目标,因为每种技术都适合于不同的场合。

因此,Microsoft 把所有这些技术集成在一起,放在 .NET Framework 3.0、3.5 和 4 中。现在只有一种移动数据的方式——Windows Communication Foundation(WCF)。WCF 允许建立好服务后,只要修改配置文件,就可以用多种方式提供该服务(甚至在不同的协议下)。WCF 是一种连接各种系统的强大的新方式。第 43 章将详细介绍 WCF。

1.7.7 Windows WF

Windows Workflow Foundation(WF)实际上是在 .NET Framework 3.0 中引入的,但经过全面修订,现在许多人都发现它更容易使用了。Visual Studio 2010 在使用 WF 方面有了长足的进步,并使 workflow 构建变得更简单。WF 有一个新的流程控制类 `Flowchart`,还有一些新活动,例如 `DoWhile`、`ForEach` 和 `ParallelForEach`。

WF 参见第 44 章。

1.8 C#在.NET 企业体系结构中的作用

C#需要 .NET 运行库,在几年内大多数客户机——特别是大多数家用计算机——就可以安装 .NET 了。而且,安装 C#应用程序也意味着安装 .NET 可重新分布的组件。因此,企业环境中会有

许多 C# 应用程序。实际上, C# 为希望建立稳健的 n 层客户机/服务器应用程序的公司提供了一个最佳的机会。

C# 与 ADO.NET 合并后, 就可以快速而经常地访问数据存储库了, 如 SQL Server 和 Oracle 数据库。返回的数据集很容易通过 ADO.NET 对象模型或 LINQ 来处理, 并自动显示为 XML, 以便通过办公室内联网来传输。

一旦为新项目建立了数据库模式, C# 就会为执行一层数据访问对象提供一个极好的媒介, 每个对象都能提供对不同数据库表的插入、更新和删除访问。

因为这是第一个基于组件的 C 语言, 所以 C# 非常适合于执行业务对象层。它为组件之间的通信封装了杂乱的信息, 让开发人员把注意力集中在如何把数据访问对象组合在一起, 在方法中精确地强制执行公司的业务规则。而且使用特性, C# 业务对象可以配备方法级的安全检查、对象池和由 COM+ 服务提供的 JIT 活动。另外, .NET 附带的实用程序允许新的 .NET 业务对象与原来的 COM 组件交互。

要使用 C# 创建企业应用程序, 可以为数据访问对象创建一个类库项目, 为业务对象创建另一个类库项目。在开发时, 可以使用 Console 项目测试类上的方法。喜欢编程的人可以建立能自动从批处理文件中执行的 Console 项目, 对工作代码进行单元测试, 以便确定代码是否中断。

注意, C# 和 .NET 都会影响物理封装可重用类的方式。过去, 许多开发人员把许多类放在一个物理组件中, 因为这样安排会使部署容易得多; 如果有版本冲突问题, 就知道在何处进行检查。因为部署 .NET 企业组件仅是把文件复制到目录中, 所以现在开发人员可以把他们的类封装到逻辑性更高的离散组件中, 而不会遇到 “DLL Hell”。

最后, 用 C# 编写的 ASP.NET 页面构成了用户界面的绝妙媒介。ASP.NET 页面是编译过的, 所以执行得比较快。它们可以在 Visual Studio 2010 IDE 中调试, 所以十分健壮。它们支持所有的语言功能, 例如早期绑定、继承和模块化, 所以用 C# 编写的 ASP.NET 页面是很整洁的, 很容易维护。

经验丰富的开发人员对大做广告的新技术和语言都持非常怀疑的态度, 不愿意利用新平台, 这仅仅是因为他们不愿意。如果读者是一位 IT 部门的企业开发人员, 或者通过 World Wide Web 提供应用程序服务, 即使一些比较新奇的功能(如 XML Web 服务和服务器端控件)不算在内, 也可以确保 C# 和 .NET 至少提供了 4 个优点:

- 组件冲突将很少见, 部署工作将更容易, 因为同一组件的不同版本可以在同一台计算机上并行运行, 而不会发生冲突。
- ASP.NET 代码不再难懂。
- 可以利用 .NET 基类中的许多功能。
- 利用 C# 可以很容易编写需要 Windows 窗体用户界面的应用程序。

在某种程度上, 因为 Web 窗体和基于 Internet 的应用程序的出现, 以前 Windows 窗体并未受到重视。但如果用户缺乏 JavaScript、ASP 或相关技术的专业知识, Windows 窗体仍是方便而快速地创建用户界面的一种可行选择。记住管理好代码, 使用户界面逻辑与业务逻辑和数据访问代码分隔开来, 这样才能在将来需要的时候把应用程序迁移到浏览器上。另外, Windows 窗体还为家用应用程序和一些小公司长期保留了重要的用户界面。Windows 窗体的新智能客户特性(很容易以在线和离线方式工作)将能开发出新的、更好的应用程序。

1.9 小结

本章介绍了许多基础知识，简要回顾了 .NET Framework 的重要方面以及它与 C# 的关系。首先讨论了所有面向 .NET 的语言如何编译为中间语言(之后由公共语言运行库进行编译和执行)，接着讨论了 .NET 的下述特性在编译和执行过程中的作用：

- 程序集和 .NET 基类
- COM 组件
- JIT 编译
- 应用程序域
- 垃圾收集

图 1-4 简要说明了这些特性在编译和执行过程中如何发挥作用。

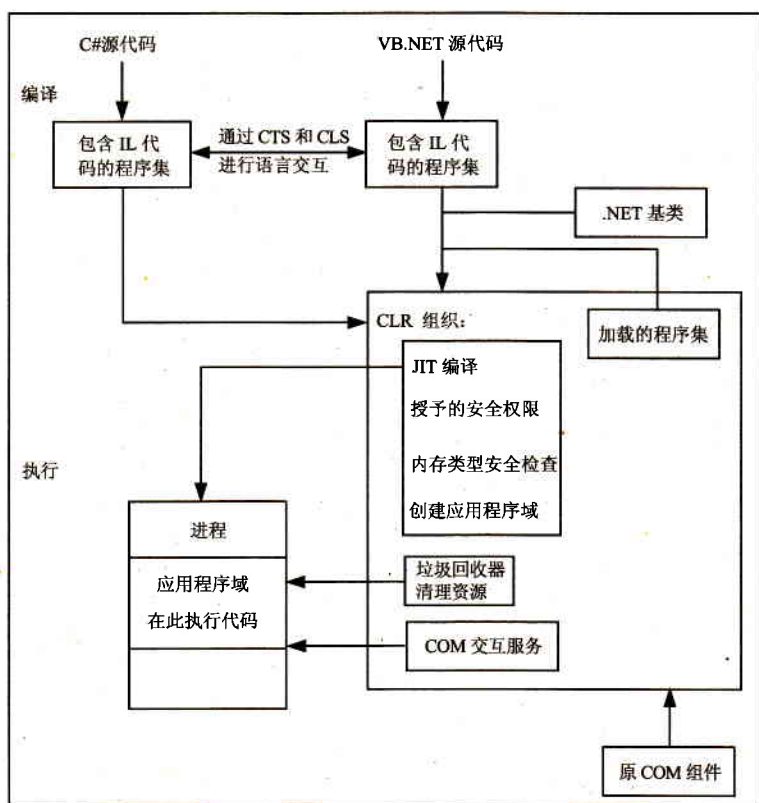


图 1-4

本章还讨论了 IL 的特征，特别是其强数据类型化和面向对象的特征。探讨了这些特征如何影响面向 .NET(包括 C#)的语言，并阐述了 IL 的强类型本质如何支持语言的互操作性，以及 CLR 服务，如垃圾收集 and 安全性。还讨论了用于帮助处理语言互操作性的 CLS 和 CTS。

本章最后讨论了 C# 如何用作基于几种 .NET 技术(包括 ASP.NET)的应用程序的基础。

第 2 章将介绍如何用 C# 语言编写代码。

第 2 章

核 心 C#

本章内容:

- 声明变量
- 变量的初始化和作用域
- C#的预定义数据类型
- 在 C#程序中使用条件语句、循环和跳转语句指定执行流
- 枚举
- 名称空间
- Main()方法
- 基本的命令行 C#编译器选项
- 使用 System.Console 执行控制台 I/O
- 使用内部注释和文档编制功能
- 预处理器指令
- C#编程的推荐规则和约定

理解了 C#的用途后, 就可以学习如何使用它了。本章将介绍 C#的基础知识, 本章的内容也是后续章节的基础, 好的开端等于成功的一半。阅读完本章后, 读者就有足够的 C#知识编写简单的程序了, 但还不能使用继承或其他面向对象的特征。这些内容将在后面的几章中讨论。

2.1 第一个 C#程序

下面编译并运行最简单的 C#程序, 这是一个简单的控制台应用程序, 它由把某条消息写到屏幕上的一个类组成。



后面几章会介绍许多代码示例。编写 C#程序最常用的技巧是使用 Visual Studio 2010 生成一个基本项目, 再添加自己的代码。但是, 第 I 部分的目的是讲授 C#语言, 为了简单起见, 在第 16 章之前避免涉及 Visual Studio 2010。我们使代码显示为简单的文件, 这样就可以使用任何文本编辑器输入它们, 并在命令行上编译。

2.1.1 代码

在文本编辑器(如 Notepad)中输入下面的代码,把它保存为后缀名为.cs 的文件(如 First.cs)。Main()方法如下所示(更多信息参见 2.7 节):



```
using System;

namespace Wrox
{
    public class MyFirstClass
    {
        static void Main()
        {
            Console.WriteLine("Hello from Wrox.");
            Console.ReadLine();
            return;
        }
    }
}
```

代码段 First.cs

2.1.2 编译并运行程序

对源文件运行 C# 命令行编译器(csc.exe), 编译这个程序:

```
csc First.cs
```

如果使用 csc 命令在命令行上编译代码, 就应注意.NET 命令行工具(包括 csc)只有在设置了某些环境变量后才能使用。根据安装.NET(和 Visual Studio 2010)的方式, 这里显示的结果可能与您计算机上的结果不同。



如果没有设置环境变量, 有两种解决方法。第 1 种方法是在运行 csc 之前, 从命令提示符窗口上运行批处理文件 %Microsoft Visual Studio 2010%\Common7\Tools\vsvars32.bat。其中 %Microsoft Visual Studio 2010% 是 Visual Studio 2010 的安装文件夹。第 2 种方法(更简单)是使用 Visual Studio 2010 命令提示符代替通常的命令提示符窗口。Visual Studio 2010 命令提示符在菜单“开始”|“程序”| Microsoft Visual Studio 2010 | Microsoft Visual Studio Tools 子菜单下。它只是一个命令提示符窗口, 打开时会自动运行 vsvars32.bat。

编译代码, 会生成一个可执行文件 First.exe。在命令行或 Windows Explorer 上, 像运行任何可执行文件那样运行该文件, 得到如下结果:

```
csc First.cs
Microsoft (R) Visual C# 2010 Compiler version 4.0.20506.1
```

Copyright (C) Microsoft Corporation. All rights reserved.

First.exe

Hello from Wrox.

2.1.3 详细介绍

首先对 C#语法作几个一般性的解释。在 C#中, 与其他 C 风格的语言一样, 大多数语句都以分号(;)结尾, 语句可以写在多个代码行上, 不需要使用续行字符。用花括号({})把语句组合为块。单行注释以两个斜杠字符开头(/), 多行注释以一条斜杠和一个星号(*)开头, 以一个星号和一条斜杠(*)结尾。在这些方面, C#与 C++和 Java 一样, 但与 Visual Basic 不同。分号和花括号使 C#代码与 Visual Basic 代码有差异很大的外观。如果您以前使用的是 Visual Basic, 就应特别注意每条语句结尾的分号。对于使用 C 风格语言的新用户, 忽略分号常常是导致编译错误的一个最主要的原因。另一个方面是, C#区分大小写, 也就是说, 变量 myVar 与 MyVar 是两个不同的变量。

在上面的代码示例中, 前几行代码与名称空间有关(如本章后面所述), 名称空间是把相关类组合在一起的方式。namespace 关键字声明了应与类相关的名称空间。其后花括号中的所有代码都被认为是在这个名称空间中。编译器在 using 语句指定的名称空间中查找没有在当前名称空间中定义但在代码中引用的类。这非常类似于 Java 中的 import 语句和 C++中的 using namespace 语句。

```
using System;

namespace Wrox
{
```

在 First.cs 文件中使用 using 指令的原因是下面要使用一个库类 System.Console。using System 语句允许把这个类简写为 Console(System 名称空间中的其他类也与此类似)。如果没有 using, 就必须完全限定对 Console.WriteLine()方法的调用, 如下所示:

```
System.Console.WriteLine("Hello from Wrox.");
```

标准的 System 名称空间包含了最常用的 .NET 类型。在 C#中做的所有工作都依赖于 .NET 基类, 认识到这一点非常重要; 在本例中, 我们使用了 System 名称空间中的 Console 类, 以写入控制台窗口。C#没有用于输入和输出的内置关键字, 而是完全依赖于 .NET 类。



几乎所有的 C#程序都使用 System 名称空间中的类, 所以假定本章所有的代码文件都包含 “using System;” 语句。

接着, 声明一个类 MyFirstClass。但是, 因为该类位于 Wrox 名称空间中, 所以其完整的名称是 Wrox.MyFirstCSharpClass:

```
class MyFirstCSharpClass
{
```

所有的 C#代码都必须包含在一个类中。类的声明包括 class 关键字, 其后是类名和一对花括号。

与类相关的所有代码都应放在这对花括号中。

下面声明方法 `Main()`。每个 C# 可执行文件(如控制台应用程序、Windows 应用程序和 Windows 服务)都必须有一个入口点——`Main()`方法(注意 M 大写):

```
public static void Main()
{
```

在程序启动时调用这个方法。该方法要么没有返回值(`void`)，要么返回一个整数(`int`)。注意，在 C# 中方法的定义如下所示:

```
[modifiers] return_type MethodName([parameters])
{
    // Method body. NB. This code block is pseudo-code.
}
```

第一个方括号中的内容表示可选关键字。修饰符(modifiers)用于指定用户所定义的方法的某些特性，如可以在什么地方调用该方法。在本例中，有两个修饰符 `public` 和 `static`。修饰符 `public` 表示可以在任何地方访问该方法，所以可以在类的外部调用它。修饰符 `static` 表示方法不能在类的实例上执行，因此不必先实例化类再调用。这非常重要，因为我们创建的是一个可执行文件，而不是类库。把返回类型设置为 `void`，在本例中，不包含任何参数。

最后，看看代码语句。

```
Console.WriteLine("Hello from Wrox.");
Console.ReadLine();
return;
```

在本例中，我们只调用了 `System.Console` 类的 `WriteLine()` 方法，把一行文本写到控制台窗口上。`WriteLine()` 是一个静态方法，在调用之前不需要实例化 `Console` 对象。

`Console.ReadLine()` 读取用户的输入，添加这行代码会让应用程序等待用户按回车键，之后退出应用程序。在 Visual Studio 2010 中，控制台窗口会消失。

然后调用 `return` 退出该方法(因为这是 `Main` 方法，所以也退出了程序)。在方法头中指定 `void`，因此没有返回值。

对 C# 基本语法有了大致的认识后，下面就详细讨论 C# 的各个方面。因为没有变量不可能编写出重要的程序，所以首先介绍 C# 中的变量。

2.2 变量

在 C# 中声明变量使用下述语法:

```
datatype identifier;
例如:
```

```
int i;
```

该语句声明 `int` 变量 `i`。编译器不允许在表达式中使用这个变量，除非用一个值初始化了该变量。

声明 `i` 之后, 就可以使用赋值运算符(=)给它赋值:

```
i = 10;
```

还可以在一行代码中声明变量, 并初始化它的值:

```
int i = 10;
```

如果在一条语句中声明和初始化了多个变量, 那么所有的变量都具有相同的数据类型:

```
int x = 10, y = 20; // x and y are both ints
```

要声明不同类型的变量, 需要使用单独的语句。在多个变量的声明中, 不能指定不同的数据类型:

```
int x = 10;
bool y = true;           // Creates a variable that stores true or false
int x = 10, bool y = true; // This won't compile!
```

注意上面例子中的“//”和其后的文本, 它们是注释。“//”字符串告诉编译器, 忽略该行后面的文本, 这些文本仅为了让人更好地理解程序, 它们并不是程序的一部分。本章后面会详细讨论代码中的注释。

2.2.1 变量的初始化

变量的初始化是 C# 强调安全性的另一个例子。简单地说, C# 编译器需要用某个初始值对变量进行初始化, 之后才能在操作中引用该变量。大多数现代编译器把没有初始化标记为警告, 但 C# 编译器把它当作错误来看待。这就可以防止我们无意中从其他程序遗留下来的内存中获取垃圾值。

C# 有两个方法可确保变量在使用前进行了初始化:

- 变量是类或结构中的字段, 如果没有显式初始化, 创建这些变量时, 其默认值就是 0 (类和结构在后面讨论)。
- 方法的局部变量必须在代码中显式初始化, 之后才能在语句中使用它们的值。此时, 初始化不是在声明该变量时进行的, 但编译器会通过方法检查所有可能的路径, 如果检测到局部变量在初始化之前就使用了它的值, 就会产生错误。

例如, 在 C# 中不能使用下面的语句:

```
public static int Main()
{
    int d;
    Console.WriteLine(d); // Can't do this! Need to initialize d before use
    return 0;
}
```

注意在这段代码中, 演示了如何定义 `Main()`, 使之返回一个 `int` 类型的数据, 而不是 `void`。在编译这些代码时, 会得到下面的错误消息:

```
Use of unassigned local variable 'd'
```

考虑下面的语句:

```
Something objSomething;
```

在 C# 中, 这行代码仅会为 `Something` 对象创建一个引用; 但这个引用还没有指向任何对象。对该变量调用方法或属性会导致错误。

在 C# 中实例化一个引用对象需要使用 `new` 关键字。如上所述, 创建一个引用, 使用 `new` 关键字将该引用指向存储在堆上的一个对象:

```
objSomething = new Something(); // This creates a Something on the heap
```

2.2.2 类型推断

类型推断(type inference)使用 `var` 关键字。声明变量的语法有些变化。编译器可以根据变量的初始化值“推断”变量的类型。例如:

```
int someNumber = 0;
```

就变成:

```
var someNumber = 0;
```

即使 `someNumber` 从来没有声明为 `int`, 编译器也可以确定, 只要 `someNumber` 在其作用域内, 就是一个 `int`。编译后, 上面两个语句是等价的。

下面是另一个小例子:



```
using System;
```

```
namespace Wrox
```

```
{
    class Program
```

```
{
    static void Main(string[] args)
    {
        var name = "Bugs Bunny";
        var age = 25;
        var isRabbit = true;

        Type nameType = name.GetType();
        Type ageType = age.GetType();
        Type isRabbitType = isRabbit.GetType();

        Console.WriteLine("name is type " + nameType.ToString());
        Console.WriteLine("age is type " + ageType.ToString());
        Console.WriteLine("isRabbit is type " + isRabbitType.ToString());
    }
}
```

代码段 Var.cs

这个程序的输出如下:

```
name is type System.String
age is type System.Int32
isRabbit is type System.Bool
```

需要遵循一些规则：

- 变量必须初始化。否则，编译器就没有推断变量类型的依据。
- 初始化器不能为空。
- 初始化器必须放在表达式中。
- 不能把初始化器设置为一个对象，除非在初始化器中创建了一个新对象。

第3章在讨论匿名类型时将详细探讨。

声明了变量，推断出了类型后，就不能改变变量类型了。变量的类型确定后，就遵循其他变量类型遵循的强类型化规则。

2.2.3 变量的作用域

变量的作用域是可以访问该变量的代码区域。一般情况下，确定作用域遵循以下规则：

只要类在某个作用域内，其字段(也称为成员变量)也在该作用域内。

- 局部变量存在于表示声明该变量的块语句或方法结束的右花括号之前的作用域内。
- 在 for、while 或类似语句中声明的局部变量存在于该循环体内。

1. 局部变量的作用域冲突

大型程序在不同部分为不同的变量使用相同的变量名很常见。只要变量的作用域是程序的不同部分，就不会有问题，也不会产生多义性。但要注意，同名的局部变量不能在单一作用域内声明两次，所以不能使用下面的代码：

```
int x = 20;
// some more code
int x = 30;
```

考虑下面的代码示例：



可从
wrox.com
下载源代码

```
using System;

namespace Wrox.ProCSharp.Basics
{
    public class ScopeTest
    {
        public static int Main()
        {
            for (int i = 0; i < 10; i++)
            {
                Console.WriteLine(i);
            } // i goes out of scope here

            // We can declare a variable named i again, because
            // there's no other variable with that name in scope
        }
    }
}
```

```

for (int i = 9; i >= 0; i -- )
{
    Console.WriteLine(i);
} // i goes out of scope here.
return 0;

```

代码段 Scope.cs

这段代码使用两个 for 循环打印 0~9 的数字，再逆序打印 0~9 的数字。重要的是在同一个方法中，代码中的变量 i 声明了两次。可以这么做的原因是在两次声明中，i 都是在循环内部声明的，所以变量 i 对于各自的循环来说是局部变量。

下面是另一个例子：



可从
wrox.com
下载源代码

```

public static int Main()
{
    int j = 20;
    for (int i = 0; i < 10; i++)
    {
        int j = 30; // Can't do this - j is still in scope
        Console.WriteLine(j + i);
    }
    return 0;
}

```

代码段 ScopeBad.cs

如果试图编译它，就会产生如下错误：

```

ScopeTest.cs(12,15): error CS0136: A local variable named 'j' cannot be declared in this
scope because it would give a different meaning to 'j', which is already used in a 'parent
or current' scope to denote something else.

```

其原因是：变量 j 是在 for 循环开始前定义的，在执行 for 循环时应处于其作用域内，在 Main() 方法结束执行后，变量 j 才超出作用域，第 2 个 j(不合法)则在循环的作用域内，该作用域嵌套在 Main() 方法的作用域内。因为编译器无法区分这两个变量，所以不允许声明第 2 个变量。

2. 字段和局部变量的作用域冲突

某些情况下，可以区分名称相同(尽管其完全限定的名称不同)、作用域相同的两个标识符。此时编译器允许声明第 2 个变量。原因是 C# 在变量之间有一个基本的区分，它把在类型级别声明的变量看作字段，而把在方法中声明的变量看作局部变量。

考虑下面的代码：



可从
wrox.com
下载源代码

```

using System;

namespace Wrox
{
    class ScopeTest2
    {

```

```

    {
        static int j = 20;
        public static void Main()
        {
            int j = 30;
            Console.WriteLine(j);
            return;
        }
    }
}

```

代码段 ScopeTest2.cs

虽然在 `Main()` 方法的作用域内声明了两个变量 `j`，这段代码也会编译——在类级上定义的 `j`，在该类删除前是不会超出作用域的(在本例中，当 `Main()` 方法中断，程序结束时，才会删除该类)；以及在 `Main()` 中定义的 `j`。此时，在 `Main()` 方法中声明的新变量 `j` 隐藏了同名的类级变量，所以在运行这段代码时，会显示数字 30。

但是，如果要引用类级变量，该怎么办？可以使用语法 `object.fieldname`，在对象的外部引用类或结构的字段。在上面的例子中，我们访问静态方法中的一个静态字段(静态字段详见下一节)，所以不能使用类的实例，只能使用类本身的名称：

```

...
public static void Main()
{
    int j = 30;
    Console.WriteLine(j);
    Console.WriteLine(ScopeTest2.j);
}
...

```

如果要访问一个实例字段(该字段属于类的一个特定实例)，就需要使用 `this` 关键字。

2.2.4 常量

顾名思义，常量是其值在使用过程中不会发生变化的变量。在声明和初始化变量时，在变量的前面加上关键字 `const`，就可以把该变量指定为一个常量：

```
const int a = 100; // This value cannot be changed.
```

常量具有如下特点：

- 常量必须在声明时初始化。指定了其值后，就不能再改写了。
- 常量的值必须能在编译时用于计算。因此，不能用从一个变量中提取的值来初始化常量。如果需要这么做，应使用只读字段(详见第3章)。

- 常量总是静态的。但注意，不必(实际上，是不允许)在常量声明中包含修饰符 `static`。

在程序中使用常量至少有 3 个好处：

- 由于使用易于读取的名称(名称的值易于理解)替代了较难读取的数字或字符串，常量使程序变得更易于阅读。

- 常量使程序更易于修改。例如，在 C# 程序中有一个 `SalesTax` 常量，该常量的值为 6%。如果以后销售税率发生变化，把新值赋给这个常量，就可以修改所有的税款计算结果，而不必查找整个程序去修改税率为 0.06 的每个项。
- 常量更容易避免程序出现错误。如果在声明常量的位置以外的某个地方将另一个值赋给常量，编译器就会报告错误。

2.3 预定义数据类型

前面介绍了如何声明变量和常量，下面要详细讨论 C# 中可用的数据类型。与其他语言相比，C# 对其可用的类型及其定义有更严格的描述。

2.3.1 值类型和引用类型

在开始介绍 C# 中的数据类型之前，理解 C# 把数据类型分为两种非常重要：

- 值类型
- 引用类型

下面几节将详细介绍值类型和引用类型的语法。从概念上看，其区别是值类型直接存储其值，而引用类型存储对值的引用。

这两种类型存储在内存的不同地方：值类型存储在堆栈中，而引用类型存储在托管堆上。注意区分某个类型是值类型还是引用类型，因为这种存储位置的不同会有不同的影响。例如，`int` 是值类型，这表示下面的语句会在内存的两个地方存储值 20：

```
// i and j are both of type int
i = 20;
j = i;
```

但考虑下面的代码。这段代码假定已经定义了一个类 `Vector`，`Vector` 是一个引用类型，它有一个 `int` 类型的成员变量 `Value`：

```
Vector x, y;
x = new Vector ();
x.Value = 30; // Value is a field defined in Vector class
y = x;
Console.WriteLine(y.Value);
y.Value = 50;
Console.WriteLine(x.Value);
```

要理解的重要一点是在执行这段代码后，只有一个 `Vector` 对象。`x` 和 `y` 都指向包含该对象的内存位置。因为 `x` 和 `y` 是引用类型的变量，声明这两个变量只保留了一个引用——而不会实例化给定类型的对象。两种情况下都不会真正创建对象。要创建对象，就必须使用 `new` 关键字，如上所示。因为 `x` 和 `y` 引用同一个对象，所以对 `x` 的修改会影响 `y`，反之亦然。因此上面的代码会显示 30 和 50。



C++开发人员应注意,这个语法类似于引用,而不是指针。我们使用.(句点)符号,而不是->来访问对象成员。在语法上,C#引用看起来更类似于C++引用变量。但是,抛开表面的语法,实际上它类似于C++指针。

如果变量是一个引用,就可以把其值设置为 `null`,表示它不引用任何对象:

```
y = null;
```

如果将引用设置为 `null`,显然就不可能对它调用任何非静态的成员函数或字段,这么做会在运行期间抛出一个异常。

在C#中,基本数据类型如 `bool` 和 `long` 都是值类型。如果声明一个 `bool` 变量,并给它赋予另一个 `bool` 变量的值,在内存中就会有二个 `bool` 值。如果以后修改第一个 `bool` 变量的值,第二个 `bool` 变量的值也不会改变。这些类型是通过值来复制的。

相反,大多数更复杂的C#数据类型,包括我们自己声明的类都是引用类型。它们分配在堆中,其生存期可以跨多个函数调用,可以通过一个或几个别名来访问。CLR 执行一种精细的算法,来跟踪哪些引用变量仍是可以访问的,哪些引用变量已经不能访问了。CLR 会定期删除不能访问的对象,把它们占用的内存返回给操作系统。这是通过垃圾收集器实现的。

把基本类型(如 `int` 和 `bool`)规定为值类型,而把包含许多字段的较大类型(通常在有类的情况下)规定为引用类型,C#设计这种方式的原因是可以得到最佳性能。如果要把自己的类型定义为值类型,就应把它声明为一个结构。

2.3.2 CTS 类型

如第1章所述,C#认可的基本预定义类型并没有内置于C#语言中,而是内置于.NET Framework 中。例如,在C#中声明一个 `int` 类型的数据时,声明的实际上是.NET 结构 `System.Int32` 的一个实例。这听起来似乎很深奥,但其意义深远:这表示在语法上,可以把所有的基本数据类型看作是支持某些方法的类。例如,要把 `int i` 转换为 `string`,可以编写下面的代码:

```
string s = i.ToString();
```

应强调的是,在这种便利语法的背后,类型实际上仍存储为基本类型。基本类型在概念上用.NET 结构表示,所以肯定没有性能损失。

下面看看C#中定义的内置类型。我们将列出每个类型,以及它们的定义和对应.NET 类型(CTS 类型)的名称。C#有15个预定义类型,其中13个是值类型,两个是引用类型(`string` 和 `object`)。

2.3.3 预定义的值类型

内置的CTS 值类型表示基本类型,如整型和浮点类型、字符类型和布尔类型。

1. 整型

C#支持8个预定义整数类型,如表2-1所示。

表 2-1

名 称	CTS 类 型	说 明	范 围
sbyte	System.SByte	8 位有符号的整数	-128~127 ($-2^7 \sim 2^7 - 1$)
short	System.Int16	16 位有符号的整数	-32 768~32 767 ($-2^{15} \sim 2^{15} - 1$)
int	System.Int32	32 位有符号的整数	-2 147 483 648~2 147 483 647 ($-2^{31} \sim 2^{31} - 1$)
long	System.Int64	64 位有符号的整数	-9 223 372 036 854 775 808~9 223 372 036 854 775 807 ($-2^{63} \sim 2^{63} - 1$)
byte	System.Byte	8 位无符号的整数	0~255 ($0 \sim 2^8 - 1$)
ushort	System.UInt16	16 位无符号的整数	0~65535 ($0 \sim 2^{16} - 1$)
uint	System.UInt32	32 位无符号的整数	0~4 294 967 295 ($0 \sim 2^{32} - 1$)
ulong	System.UInt64	64 位无符号的整数	0~18 446 744 073 709 551 615 ($0 \sim 2^{64} - 1$)

一些 C#类型的名称与 C++和 Java 类型一致，但其定义不同。例如，在 C#中，int 总是 32 位带符号的整数。而在 C++中，int 是带符号的整数，但其位数取决于平台(在 Windows 上是 32 位)。在 C#中，所有的数据类型都以与平台无关的方式定义，以备将来 C#和.NET 迁移到其他平台上。

byte 是 0~255(包括 255)的标准 8 位类型。注意，在强调类型的安全性时，C#认为 byte 类型和 char 类型完全不同，它们之间的编程转换必须显式写出。还要注意，与整数中的其他类型不同，byte 类型在默认状态下是无符号的，其有符号的版本有一个特殊的名称 sbyte。

在.NET 中，short 不再很短，现在它有 16 位长。Int 类型更长，有 32 位。long 类型最长，有 64 位。所有整数类型的变量都能被赋予十进制或十六进制的值，后者需要 0x 前缀：

```
long x = 0x12ab;
```

如果对一个整数是 int、uint、long 或是 ulong 没有任何显式的声明，则该变量默认为 int 类型。为了把输入的值指定为其他整数类型，可以在数字后面加上如下字符：

```
uint ui = 1234U;
long l = 1234L;
ulong ul = 1234UL;
```

也可以使用小写字母 u 和 l，但后者会与整数 1 混淆。

2. 浮点类型

C#提供了许多整型数据类型，也支持浮点类型，如表 2-2 所示。

表 2-2

名 称	CTS 类 型	说 明	位 数	范围(大致)
float	System.Single	32 位单精度浮点数	7	$\pm 1.5 \times 10^{-45} \sim \pm 3.4 \times 10^{38}$
double	System.Double	64 位双精度浮点数	15/16	$\pm 5.0 \times 10^{-324} \sim \pm 1.7 \times 10^{308}$

float 数据类型用于较小的浮点数，因为它要求的精度较低。double 数据类型比 float 数据类型大，

提供的精度也大一倍(15 位)。

如果在代码中没有对某个非整数值(如 12.3)硬编码, 则编译器一般假定该变量是 `double`。如果想指定该值为 `float`, 可以在其后加上字符 `F`(或 `f`):

```
float f = 12.3F;
```

3. decimal 类型

`decimal` 类型表示精度更高的浮点数, 如表 2-3 所示。

表 2-3

名 称	CTS 类型	说 明	位 数	范围(大致)
<code>decimal</code>	<code>System.Decimal</code>	128 位高精度十进制数表示法	28	$\pm 1.0 \times 10^{-28} \sim \pm 7.9 \times 10^{28}$

CTS 和 C# 一个重要的优点是提供了一种专用类型进行财务计算, 这就是 `decimal` 类型, 使用 `decimal` 类型提供的 28 位的方式取决于用户。换言之, 可以用较大的精确度(带有美分)来表示较小的美元值, 也可以在小数部分用更多的舍入来表示较大的美元值。但应注意, `decimal` 类型不是基本类型, 所以在计算时使用该类型会有性能损失。

要把数字指定为 `decimal` 类型, 而不是 `double`、`float` 或整型, 可以在数字的后面加上字符 `M`(或 `m`), 如下所示。

```
decimal d = 12.30M;
```

4. bool 类型

C# 的 `bool` 类型用于包含布尔值 `true` 或 `false`, 如表 2-4 所示。

表 2-4

名 称	CTS 类型	说 明	位 数	值
<code>bool</code>	<code>System.Boolean</code>	表示 <code>true</code> 或 <code>false</code>	NA	<code>true</code> 或 <code>false</code>

`bool` 值和整数值不能相互隐式转换。如果变量(或函数的返回类型)声明为 `bool` 类型, 就只能使用值 `true` 或 `false`。如果试图使用 0 表示 `false`, 非 0 值表示 `true`, 就会出错。

5. 字符类型

为了保存单个字符的值, C# 支持 `char` 数据类型, 如表 2-5 所示。

表 2-5

名 称	CTS 类型	值
<code>char</code>	<code>System.Char</code>	表示一个 16 位的(Unicode)字符

`char` 类型的字面量是用单引号括起来的, 如 `'A'`。如果把字符放在双引号中, 编译器会把它看作字符串, 从而产生错误。

除了把 char 表示为字符字面量之外，还可以用 4 位十六进制的 Unicode 值(如'\u0041')、带有数据类型转换的整数值(如(char) 65)或十六进制数('\x0041')表示它们。它们还可以用转义序列表示，如表 2-6 所示。

表 2-6

转 义 序 列	字 符
'\'	单引号
'\"'	双引号
'\\'	反斜杠
'\0'	空
'\a'	警告
'\b'	退格
'\f'	换页
'\n'	换行
'\r'	回车
'\t'	水平制表符
'\v'	垂直制表符

2.3.4 预定义的引用类型

C#支持两种预定义的引用类型，如表 2-7 所示。

表 2-7

名 称	CTS 类 型	说 明
object	System.Object	根类型，CTS 中的其他类型都是从它派生而来的(包括值类型)
string	System.String	Unicode 字符串

1. object 类型

许多编程语言和类结构都提供了根类型，层次结构中的其他对象都从它派生而来。C#和.NET 也不例外。在 C#中，object 类型就是最终的父类型，所有内置类型和用户定义的类型都从它派生而来。这样，object 类型就可以用于两个目的：

- 可以使用 object 引用绑定任何子类型的对象。例如，第 7 章将说明如何使用 object 类型把堆栈中的一个值对象装箱，再移动到堆中。object 引用也可以用于反射，此时必须有代码来处理类型未知的对象。
- object 类型执行许多一般用途的基本方法，包括 Equals()、GetHashCode()、GetType()和 ToString()。用户定义的类需要使用一种面向对象技术——重写(见第 4 章)，提供其中一些方

法的替代执行代码。例如,重写 `ToString()` 时,要给类提供一个方法,给出类本身的字符串表示。如果类中没有提供这些方法的实现代码,编译器就会使用 `object` 类型中的实现代码,它们在类中的执行不一定正确。

后面的章节将详细讨论 `object` 类型。

2. string 类型

C# 有 `string` 关键字,在编译为 .NET 类时,它就是 `System.String`。有了它,像字符串连接和字符串复制这样的操作就很简单了:

```
string str1 = "Hello ";
string str2 = "World";
string str3 = str1 + str2; // string concatenation
```

尽管这是一个值类型的赋值,但 `string` 是一个引用类型。`String` 对象被分配在堆上,而不是栈上。因此,当把一个字符串变量赋予另一个字符串时,会得到对内存中同一个字符串的两个引用。但是,`string` 与引用类型在常见的操作上有一些区别。例如,字符串是不可改变的。修改其中一个字符串,就会创建一个全新的 `string` 对象,而另一个字符串不发生任何变化。考虑下面的代码:



可从
wrox.com
下载源代码

```
using System;

class StringExample
{
    public static int Main()
    {
        string s1 = "a string";
        string s2 = s1;
        Console.WriteLine("s1 is " + s1);
        Console.WriteLine("s2 is " + s2);
        s1 = "another string";
        Console.WriteLine("s1 is now " + s1);
        Console.WriteLine("s2 is now " + s2);
        return 0;
    }
}
```

代码段 StringExample.cs

其输出结果为:

```
s1 is a string
s2 is a string
s1 is now another string
s2 is now a string
```

改变 `s1` 的值对 `s2` 没有影响,这与我们期待的引用类型正好相反。当用值 "a string" 初始化 `s1` 时,就在堆上分配了一个新的 `string` 对象。在初始化 `s2` 时,引用也指向这个对象,所以 `s2` 的值也是 "a string"。但是当现在要改变 `s1` 的值时,并不会替换原来的值,堆上会为新值分配一个新对象。`s2` 变量仍指向原来的对象,所以它的值没有改变。这实际上是运算符重载的结果,运算符重载详见第 7

章。基本上，`string` 类实现为其语义遵循一般的、直观的字符串规则。

字符串字面量放在双引号中("`...`")；如果试图把字符串放在单引号中，编译器就会把它当作 `char`，从而引发错误。C# 字符串和 `char` 一样，可以包含 Unicode 和十六进制数转义序列。因为这些转义序列以一个反斜杠开头，所以不能在字符串中使用这个非转义的反斜杠字符，而需要用两个反斜杠字符(`\\`)来表示它：

```
string filepath = "C:\\ProCSharp\\First.cs";
```

即使用户相信自己可以在任何情况下都记住要这么做，但输入两个反斜杠字符会令人迷惑。幸好，C# 提供了另一种替代方式。可以在字符串字面量的前面加上字符 `@`；在这个字符后的所有字符都看作是其原来的含义——它们不会解释为转义字符：

```
string filepath = @"C:\ProCSharp\First.cs";
```

甚至允许在字符串字面量中包含换行符：

```
string jabberwocky = @"'Twas brillig and the slithy toves  
Did gyre and gimble in the wabe.";
```

那么 `jabberwocky` 的值就是：

```
'Twas brillig and the slithy toves  
Did gyre and gimble in the wabe.
```

2.4 流控制

本节将介绍 C# 语言的重要语句：控制程序流的语句，它们不是按代码在程序中的排列位置顺序执行的。

2.4.1 条件语句

条件语句可以根据条件是否满足或根据表达式的值控制代码的执行分支。C# 有两个控制代码分支的结构：`if` 语句，测试特定条件是否满足；`switch` 语句，它比较表达式和多个不同的值。

1. `if` 语句

对于条件分支，C# 继承了 C 和 C++ 的 `if...else` 结构。对于用过程语言编程的人，其语法非常直观：

```
if (condition)
    statement(s)
else
    statement(s)
```

如果在条件中要执行多个语句，就需要用花括号(`{ ... }`)把这些语句组合为一个块(这也适用于其他可以把语句组合为一个块的 C# 结构，如 `for` 和 `while` 循环)。

```
bool isZero;
```

```

if (i == 0)
{
    isZero = true;
    Console.WriteLine("i is Zero");
}
else
{
    isZero = false;
    Console.WriteLine("i is Non-zero");
}

```

还可以单独使用 if 语句，不加最后的 else 语句。也可以合并 else if 子句，测试多个条件。



可从
wrox.com
下载源代码

```

using System;

namespace Wrox
{
    class MainEntryPoint
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Type in a string");
            string input;
            input = Console.ReadLine();
            if (input == "")
            {
                Console.WriteLine("You typed in an empty string.");
            }
            else if (input.Length < 5)
            {
                Console.WriteLine("The string had less than 5 characters.");
            }
            else if (input.Length < 10)
            {
                Console.WriteLine("The string had at least 5 but less than 10  
Characters.");
            }
            Console.WriteLine("The string was " + input);
        }
    }
}

```

代码段 Elseif.cs

添加到 if 子句中的 else if 语句的个数不受限制。

注意在上面的例子中，我们声明了一个字符串变量 `input`，让用户在命令行上输入文本，把文本填充到 `input` 中，然后测试该字符串变量的长度。代码还说明了在 C# 中如何进行字符串处理。例如，要确定 `input` 的长度，可以使用 `input.Length`。

对于 if，要注意的一点是如果条件分支中只有一条语句，就无需使用花括号：

```

if (i == 0) Let's add some brackets here.
    Console.WriteLine("i is Zero");           // This will only execute if i == 0

```

```
Console.WriteLine("i can be anything");// Will execute whatever the  
// value of i
```

但是,为了保持一致,许多程序员只要使用 if 语句,就加上花括号。

前面介绍的 if 语句还演示了用于比较数值的一些 C#运算符。特别注意, C#使用 “=” 对变量进行等于比较。此时不要使用 “=”, 一个 “=” 用于赋值。

在 C#中, if 子句中的表达式必须等于布尔值。不能直接测试整数(如从函数中返回的值), 而必须明确地把返回的整数转换为布尔值 true 或 false, 例如, 将值与 0 或 null 进行比较:

```
if (DoSomething() != 0)  
{  
    // Non-zero value returned  
}  
else  
{  
    // Returned zero  
}
```

2. switch 语句

switch...case 语句适合于从一组互斥的分支中选择一个执行分支。其形式是 switch 参数的后面跟一组 case 子句。如果 switch 参数中表达式的值等于某个 case 子句旁边的某个值, 就执行该 case 子句中的代码。此时不需要使用花括号把语句组合到块中; 只需使用 break 语句标记每段 case 代码的结尾即可。也可以在 switch 语句中包含一条 default 子句, 如果表达式不等于任何 case 子句的值, 就执行 default 子句的代码。下面的 switch 语句测试 integerA 变量的值:

```
switch (integerA)  
{  
    case 1:  
        Console.WriteLine("integerA =1");  
        break;  
    case 2:  
        Console.WriteLine("integerA =2");  
        break;  
    case 3:  
        Console.WriteLine("integerA =3");  
        break;  
    default:  
        Console.WriteLine("integerA is not 1,2, or 3");  
        break;  
}
```

注意 case 的值必须是常量表达式; 不允许使用变量。

C 和 C++程序员应很熟悉 switch...case 语句, 而 C#的 switch...case 语句更安全。特别是它禁止几乎所有 case 中的失败条件。如果激活了块中靠前的一条 case 子句, 后面的 case 子句就不会被激活, 除非使用 goto 语句特别标记也要激活后面的 case 子句。编译器会把没有 break 语句的 case 子句标记为错误:

Control cannot fall through from one case label ('case 2:') to another

在有限的几种情况下,这种失败是允许的,但在大多数情况下,我们不希望出现这种失败,而且这会导致出现很难察觉的逻辑错误。让代码正常工作,而不是出现异常,这样不是更好吗?

但在使用 goto 语句时,会在 switch...cases 中重复出现失败。如果确实想这么做,就应重新考虑设计方案了。下面的代码说明了如何使用 goto 模拟失败,得到的代码会非常混乱:

```
// assume country and language are of type string
switch(country)
{
    case "America":
        CallAmericanOnlyMethod();
        goto case "Britain";
    case "France":
        language = "French";
        break;
    case "Britain":
        language = "English";
        break;
}
```

但有一种例外情况。如果一条 case 子句为空,就可以从这个 case 跳到下一条 case 上,这样就可以用相同的方式处理两条或多条 case 子句了(不需要 goto 语句)。

```
switch(country)
{
    case "au":
    case "uk":
    case "us":
        language = "English";
        break;
    case "at":
    case "de":
        language = "German";
        break;
}
```

在 C# 中,switch 语句的一个有趣的地方是 case 子句的排放顺序是无关紧要的,甚至可以把 default 子句放在最前面!因此,任何两条 case 都不能相同。这包括值不同的不同常量,所以不能这样编写:

```
// assume country is of type string
const string england = "uk";
const string britain = "uk";
switch(country)
{
    case england:
    case britain: // this will cause a compilation error
        language = "English";
        break;
}
```

上面的代码还说明了 C# 中的 `switch` 语句与 C++ 中的 `switch` 语句的另一个不同之处：在 C# 中，可以把字符串用作测试的变量。

2.4.2 循环

C# 提供了 4 种不同的循环机制(`for`、`while`、`do...while` 和 `foreach`)，在满足某个条件之前，可以重复执行代码块。

1. `for` 循环

C# 的 `for` 循环提供的迭代循环机制是在执行下一次迭代前，测试是否满足某个条件，其语法如下：

```
for (initializer; condition; iterator)
    statement(s)
```

其中：

- **initializer** 是指在执行第一次循环前要计算的表达式(通常把一个局部变量初始化为循环计数器)；
- **condition** 是在每次迭代执行新循环前要测试的表达式(它必须等于 `true`，才能执行下一次迭代)；
- **iterator** 是每次迭代完要计算的表达式(通常是递增循环计数器)。

当 **condition** 等于 `false` 时，迭代停止。

`for` 循环是所谓的预测试循环，因为循环条件是在执行循环语句前计算的，如果循环条件为假，循环语句就根本不会执行。

`for` 循环非常适合于一个语句或语句块重复执行预定的次数。下面的例子就是 `for` 循环的典型用法，这段代码输出从 0~99 的整数：

```
for (int i = 0; i < 100; i = i+1) // this is equivalent to
                                // For i = 0 To 99 in VB.
{
    Console.WriteLine(i);
}
```

这里声明了一个 `int` 类型的变量 `i`，并把它初始化为 0，用作循环计数器。接着测试它是否小于 100。因为这个条件等于 `true`，所以执行循环中的代码，显示值 0。然后给该计数器加 1，再次执行该过程。当 `i` 等于 100 时，循环停止。

实际上，上述编写循环的方式并不常用。C# 在给变量加 1 时有一种简化方式，即不使用 `i = i+1`，而简写为 `i++`：

```
for (int i = 0; i < 100; i++)
{
    //etc.
}
```

也可以上面的例子中给循环变量 `i` 使用类型推断功能。使用类型推断功能时，循环结构变成：

```
for (var i = 0; i < 100; i++)
```

```
...
```

嵌套的 `for` 循环非常常见，在每次迭代外部的循环时，内部循环都要彻底执行完毕。这种模式通常用于在矩形多维数组中遍历每个元素。最外部的循环遍历每一行，内部的循环遍历某行上的每个列。下面的代码显示数字行，它还使用另一个 `Console` 方法 `Console.Write()`，该方法的作用与 `Console.WriteLine()` 相同，但不在输出中添加回车换行符：



可从
wrox.com
下载源代码

```
using System;

namespace Wrox
{
    class MainEntryPoint
    {
        static void Main(string[] args)
        {
            // This loop iterates through rows
            for (int i = 0; i < 100; i+=10)
            {
                // This loop iterates through columns
                for (int j = i; j < i + 10; j++)
                {
                    Console.Write(" " + j);
                }
                Console.WriteLine();
            }
        }
    }
}
```

代码段 NestedFor.cs

尽管 `j` 是一个整数，但它会自动转换为字符串，以便进行连接。

上述例子的结果是：

```
0 1 2 3 4 5 6 7 8 9
10 11 12 13 14 15 16 17 18 19
20 21 22 23 24 25 26 27 28 29
30 31 32 33 34 35 36 37 38 39
40 41 42 43 44 45 46 47 48 49
50 51 52 53 54 55 56 57 58 59
60 61 62 63 64 65 66 67 68 69
70 71 72 73 74 75 76 77 78 79
80 81 82 83 84 85 86 87 88 89
90 91 92 93 94 95 96 97 98 99
```

尽管在技术上，可以在 `for` 循环的测试条件中计算其他变量，而不计算计数器变量，但这不太常见。也可以在 `for` 循环中忽略一个表达式(甚或所有表达式)。但此时，要考虑使用 `while` 循环。

2. while 循环

与 for 循环一样, while 也是一个预测试循环。其语法是类似的, 但 while 循环只有一个表达式:

```
while(condition)
    statement(s);
```

与 for 循环不同的是, while 循环最常用于以下情况: 在循环开始前, 不知道重复执行一个语句或语句块的次数。通常, 在某次迭代中, while 循环体中的语句把布尔标志设置为 false, 结束循环, 如下面的例子所示。

```
bool condition = false;
while (!condition)
{
    // This loop spins until the condition is true.
    DoSomeWork();
    condition = CheckCondition(); // assume CheckCondition() returns a bool
}
```

3. do...while 循环

do...while 循环是 while 循环的后测试版本。该循环的测试条件要在执行完循环体之后执行。因此 do...while 循环适用于至少要将循环体执行一次的情况:

```
bool condition;
do
{
    // This loop will at least execute once, even if Condition is false.
    MustBeCalledAtLeastOnce();
    condition = CheckCondition();
} while (condition);
```

4. foreach 循环

foreach 循环可以迭代集合中的每一项。现在不必考虑集合的概念, 第 10 章将详细介绍集合。知道集合是一种包含一系列对象的对象即可。从技术上看, 要使用集合对象, 就必须支持 IEnumerable 接口。集合的例子有 C# 数组、System.Collection 名称空间中的集合类, 以及用户定义的集合类。从下面的代码中可以了解 foreach 循环的语法, 其中假定 arrayOfInts 是一个整型数组:

```
foreach (int temp in arrayOfInts)
{
    Console.WriteLine(temp);
}
```

其中, foreach 循环每次迭代数组中的一个元素。它把每个元素的值放在 int 型的变量 temp 中, 然后执行一次循环迭代。

这里也可以使用类型推断功能。此时, foreach 循环变成:

```
foreach (var temp in arrayOfInts)
```

temp 的类型推断为 int, 因为这是集合项的类型。

注意, foreach 循环不能改变集合中各项(上面的 temp)的值, 所以下面的代码不会编译:

```
foreach (int temp in arrayOfInts)
{
    temp++;
    Console.WriteLine(temp);
}
```

如果需要迭代集合中的各项, 并改变它们的值, 就应使用 for 循环。

2.4.3 跳转语句

C#提供了许多可以立即跳转到程序中另一行代码的语句, 在此, 先介绍 goto 语句。

1. goto 语句

goto 语句可以直接跳转到程序中用标签指定的另一行(标签是一个标识符, 后跟一个冒号):

```
goto Label1;
    Console.WriteLine("This won't be executed");
Label1:
    Console.WriteLine("Continuing execution from here");
```

goto 语句有两个限制。不能跳转到像 for 循环这样的代码块中, 也不能跳出类的范围, 不能退出 try...catch 块后面的 finally 块(第 15 章将介绍如何用 try...catch...finally 块处理异常)。

goto 语句的名声不太好, 在大多数情况下不允许使用它。一般情况下, 使用它肯定不是面向对象编程的好方式。

2. break 语句

前面简要提到过 break 语句——在 switch 语句中使用它退出某个 case 语句。实际上, break 也可以用于退出 for、foreach、while 或 do...while 循环, 该语句会使控制流执行循环后面的语句。

如果该语句放在嵌套的循环中, 就执行最内部循环后面的语句。如果 break 放在 switch 语句或循环外部, 就会产生编译错误。

3. continue 语句

continue 语句类似于 break, 也必须在 for、foreach、while 或 do...while 循环中使用。但它只退出循环的当前迭代, 开始执行循环的下一迭代, 而不是退出循环。

4. return 语句

return 语句用于退出类的方法, 把控制权返回方法的调用者。如果方法有返回类型, return 语句必须返回这个类型的值; 如果方法返回 void, 应使用没有表达式的 return 语句。

2.5 枚举

枚举是用户定义的整数类型。在声明一个枚举时，要指定该枚举的实例可以包含的一组可接受的值。不仅如此，还可以给值指定易于记忆的名称。如果在代码的某个地方，要试图把一个不在可接受范围内的值赋予枚举的一个实例，编译器就会报告一个错误。

从长远来看，创建枚举可以节省大量时间，减少许多麻烦。使用枚举比使用无格式的整数至少有如下 3 个优势：

- 如上所述，枚举可以使代码更易于维护，有助于确保给变量指定合法的、期望的值。
- 枚举使代码更清晰，允许用描述性的名称表示整数值，而不是用含义模糊、变化多端的数来表示。
- 枚举也使代码更易于键入。在给枚举类型的实例赋值时，Visual Studio .NET IDE 会通过 IntelliSense 弹出一个包含可接受值的列表框，减少了按键次数，并能够让我们回忆起可选的值。

可以定义如下的枚举：

```
public enum TimeOfDay
{
    Morning = 0,
    Afternoon = 1,
    Evening = 2
}
```

本例在枚举中使用一个整数值，来表示一天的每个阶段。现在可以把这些值作为枚举的成员来访问。例如，`TimeOfDay.Morning` 返回数字 0。使用这个枚举一般是把合适的值传送给方法，并在 `switch` 语句中迭代可能的值。

```
class EnumExample
{
    public static int Main()
    {
        WriteGreeting(TimeOfDay.Morning);
        return 0;
    }

    static void WriteGreeting(TimeOfDay timeOfDay)
    {
        switch(timeOfDay)
        {
            case TimeOfDay.Morning:
                Console.WriteLine("Good morning!");
                break;
            case TimeOfDay.Afternoon:
                Console.WriteLine("Good afternoon!");
                break;
            case TimeOfDay.Evening:
```

```

        Console.WriteLine("Good evening!");
        break;
    default:
        Console.WriteLine("Hello!");
        break;
    }
}

```

在 C# 中, 枚举的真正强大之处是它们在后台会实例化为派生于基类 `System.Enum` 的结构。这表示可以对它们调用方法, 执行有用的任务。注意因为 .NET Framework 的执行方式, 在语法上把枚举当做结构是不会造成性能损失。实际上, 一旦代码编译好, 枚举就成为基本类型, 与 `int` 和 `float` 类似。

可以获取枚举的字符串表示, 例如使用前面的 `TimeOfDay` 枚举:

```

TimeOfDay time = TimeOfDay.Afternoon;
Console.WriteLine(time.ToString());

```

会返回字符串 `Afternoon`。

另外, 还可以从字符串中获取枚举值:

```

TimeOfDay time2 = (TimeOfDay) Enum.Parse(typeof(TimeOfDay), "afternoon", true);
Console.WriteLine((int)time2);

```

这段代码说明了如何从字符串获取枚举值, 并转换为整数。要从字符串中转换, 需要使用静态的 `Enum.Parse()` 方法, 这个方法带 3 个参数。第 1 个参数是要使用的枚举类型, 其语法是关键字 `typeof` 后跟放在括号中的枚举类名。`typeof` 运算符将在第 7 章详细论述。第 2 个参数是要转换的字符串, 第 3 个参数是一个 `bool`, 指定在进行转换时是否忽略大小写。最后, 注意 `Enum.Parse()` 方法实际上返回一个对象引用——我们需要把这个字符串显式转换为需要的枚举类型(这是一个拆箱操作的例子)。对于上面的代码, 将返回 1, 作为一个对象, 对应于 `TimeOfDay.Afternoon` 的枚举值。在显式转换为 `int` 时, 会再次生成 1。

`System.Enum` 上的其他方法可以返回枚举定义中的值的个数或列出值的名称等。详细信息参见 MSDN 文档。

2.6 名称空间

如前所述, 名称空间提供了一种组织相关类和其他类型的方式。与文件或组件不同, 名称空间是一种逻辑组合, 而不是物理组合。在 C# 文件中定义类时, 可以把它包括在名称空间定义中。以后, 在定义另一个类(在另一个文件中执行相关操作)时, 就可以在同一个名称空间中包含它, 创建一个逻辑组合, 该组合告诉使用类的其他开发人员: 这两个类是如何相关的以及如何使用它们:

```

namespace CustomerPhoneBookApp
{
    using System;

```

```
public struct Subscriber
{
    // Code for struct here...
}
```

把一个类型放在名称空间中，可以有效地给这个类型指定一个较长的名称，该名称包括类型的名称空间，名称之间用句点(.)隔开，最后是类名。在上面的例子中，Subscriber 结构的全名是 CustomerPhoneBookApp.Subscriber。这样，有相同短名的不同的类就可以在同一个程序中使用。全名常常称为完全限定的名称。

也可以在名称空间中嵌套其他名称空间，为类型创建层次结构：

```
namespace Wrox
{
    namespace ProCSharp
    {
        namespace Basics
        {
            class NamespaceExample
            {
                // Code for the class here...
            }
        }
    }
}
```

每个名称空间名都由它所在名称空间的名称组成，这些名称用句点分隔开，开头是最外层的名称空间，最后是它自己的短名。所以 ProCSharp 名称空间的全名是 Wrox.ProCSharp，NamespaceExample 类的全名是 Wrox.ProCSharp.Basics.NamespaceExample。

使用这个语法也可以组织自己的名称空间定义中的名称空间，所以上面的代码也可以写为：

```
namespace Wrox.ProCSharp.Basics
{
    class NamespaceExample
    {
        // Code for the class here...
    }
}
```

注意不允许在另一个嵌套的名称空间中声明多部分的名称空间。

名称空间与程序集无关。同一个程序集中可以有不同的名称空间，也可以在不同的程序集中定义同一个名称空间中的类型。

应在开始一个项目之前就计划定义名称空间的层次结构。一般可接受的格式是 CompanyName.ProjectNameSystemSection。所以在上面的例子中，Wrox 是公司名，ProCSharp 是项目，对于本章，Basics 是部分名。

2.6.1 using 语句

显然, 名称空间相当长, 输入起来很繁琐, 用这种方式指定某个类也不总是必要的。如本章开头所述, C#允许简写类的全名。为此, 要在文件的顶部列出类的名称空间, 前面加上 `using` 关键字。在文件的其他地方, 就可以使用其类型名称来引用名称空间中的类型了:

```
using System;
using Wrox.ProCSharp;
```

如前所述, 几乎所有的 C#源代码都以语句 `using System;` 开头, 这仅是因为 Microsoft 提供的许多有用的类都包含在 `System` 名称空间中。

如果 `using` 语句引用的两个名称空间包含同名的类型, 就必须使用完整的名称(或者至少较长的名称), 确保编译器知道访问哪个类型。例如, 假如类 `NamespaceExample` 同时存在于 `Wrox.ProCSharp.Basics` 和 `Wrox.ProCSharp.OOP` 名称空间中。如果要在名称空间 `Wrox.ProCSharp` 中创建一个类 `Test`, 并在该类中实例化一个 `NamespaceExample` 类, 就需要指定使用哪个类:

```
using Wrox.ProCSharp.OOP;
using Wrox.ProCSharp.Basics;
namespace Wrox.ProCSharp
{
    class Test
    {
        public static int Main()
        {
            Basics.NamespaceExample nSEx = new Basics.NamespaceExample();
            // do something with the nSEx variable.
            return 0;
        }
    }
}
```



因为 `using` 语句在 C# 文件的开头, 而 C 和 C++ 也把 `#include` 语句放在这里, 所以从 C++ 迁移到 C# 的程序员常把名称空间与 C++ 风格的头文件相混淆。不要犯这种错误, `using` 语句在这些文件之间并没有建立物理链接。C# 也没有对应于 C++ 头文件的部分。

公司应花一些时间开发一种名称空间模式, 这样其开发人员才能快速定位他们需要的功能, 而且公司内部使用的类名也不会与现有的类库相冲突。本章后面将介绍建立名称空间模式的规则和其他命名约定。

2.6.2 名称空间的别名

`using` 关键字的另一个用途是给类和名称空间指定别名。如果名称空间的名称非常长, 又要在代码中多次引用, 但不希望该名称空间的名称包含在 `using` 指令中(例如, 避免类名冲突), 就可以给该

名称空间指定一个别名，其语法如下：

```
using alias = NamespaceName;
```

下面的例子(前面例子的修订版本)给 `Wrox.ProCSharp.Basics` 名称空间指定别名 `Introduction`，并使用这个别名实例化了一个 `NamespaceExample` 对象，这个对象是在该名称空间中定义的。注意名称空间别名的修饰符是“`::`”。因此将先从 `Introduction` 名称空间别名开始搜索。如果在相同的作用域中引入了一个 `Introduction` 类，就会发生冲突。即使出现了冲突，“`::`”运算符也允许引用别名。`NamespaceExample` 类有一个方法 `GetNamespace()`，该方法调用每个类都有的 `GetType()` 方法，以访问表示类的类型的 `Type` 对象。下面使用这个对象来返回类的名称空间名：

```
using System;
using Introduction = Wrox.ProCSharp.Basics;
class Test
{
    public static int Main()
    {
        Introduction::NamespaceExample NSEx =
            new Introduction::NamespaceExample();
        Console.WriteLine(NSEx.GetNamespace());
        return 0;
    }
}

namespace Wrox.ProCSharp.Basics
{
    class NamespaceExample
    {
        public string GetNamespace()
        {
            return this.GetType().Namespace;
        }
    }
}
```

2.7 Main()方法

本章的开头提到过，C#程序是从方法 `Main()` 开始执行的。这个方法必须是类或结构的静态方法，并且其返回类型必须是 `int` 或 `void`。

虽然显式指定 `public` 修饰符是很常见的，因为按照定义，必须在程序外部调用该方法，但我们给该入口点方法指定什么访问级别并不重要，即使把该方法标记为 `private`，它也可以运行。

2.7.1 多个 Main()方法

在编译C#控制台或 Windows 应用程序时，默认情况下，编译器会在类中查找与上述签名匹配的 `Main()` 方法，并使这个类方法成为程序的入口点。如果有多个 `Main()` 方法，编译器就会返回一个错

误消息。例如，考虑下面的代码 DoubleMain.cs:



可从
wrox.com
下载源代码

```
using System;

namespace Wrox
{
    class Client
    {
        public static int Main()
        {
            MathExample.Main();
            return 0;
        }
    }

    class MathExample
    {
        static int Add(int x, int y)
        {
            return x + y;
        }

        public static int Main()
        {
            int i = Add(5,10);
            Console.WriteLine(i);
            return 0;
        }
    }
}
```

代码段 DoubleMain.cs

上述代码包含两个类，它们都有一个 Main() 方法。如果按照通常的方式编译这段代码，就会得到下述错误:

csc DoubleMain.cs

Microsoft (R) Visual C# 2010 Compiler version 4.0.20506.1

Copyright (C) Microsoft Corporation. All rights reserved.

DoubleMain.cs(7,25): error CS0017: Program

'DoubleMain.exe' has more than one entry point defined:

'Wrox.Client.Main()'. Compile with /main to specify the type that contains the entry point.

DoubleMain.cs(21,25): error CS0017: Program

'DoubleMain.exe' has more than one entry point defined:

'Wrox.MathExample.Main()'. Compile with /main to specify the type that contains the entry point.

但是，可以使用 /main 选项，其后跟 Main() 方法所属类的全名(包括名称空间)，明确告诉编译器把哪个方法作为程序的入口点:

```
csc DoubleMain.cs /main:Wrox.MathExample
```

2.7.2 给 Main()方法传递参数

前面的例子只介绍了不带参数的 Main()方法。但在调用程序时,可以让 CLR 包含一个参数,将命令行参数传递给程序。这个参数是一个字符串数组,传统上称为 args(但 C#可以接受任何名称)。在启动程序时,程序可以使用这个数组,访问通过命令行传送过来的选项。

下面的例子 ArgsExample.cs 是在传送给 Main()方法的字符串数组中循环,并把每个选项的值写入控制台窗口:



可从
wrox.com
下载源代码

```
using System;

namespace Wrox
{
    class ArgsExample
    {
        public static int Main(string[] args)
        {
            for (int i = 0; i < args.Length; i++)
            {
                Console.WriteLine(args[i]);
            }
            return 0;
        }
    }
}
```

代码段 ArgsExample.cs

通常使用命令行就可以编译这段代码。在运行编译好的可执行文件时,可以在程序名的后面加上参数,例如:

```
ArgsExample /a /b /c
/a
/b
/c
```

2.8 有关编译 C#文件的更多内容

前面介绍了如何使用 csc.exe 编译控制台应用程序,但其他类型的应用程序如何编译? 如果要引用一个类库,该怎么办? MSDN 文档详细介绍了 C#编译器的所有编译选项,这里只介绍其中最重要的选项。

要回答第一个问题,应使用/target 选项(常简写为/t)来指定要创建的文件类型。文件类型可以是表 2-8 所示的类型中的一种。

表 2-8

选 项	输 出
/t:exe	控制台应用程序 (默认)
/t:library	带有清单的类库
/t:module	没有清单的组件
/t:winexe	Windows 应用程序 (没有控制台窗口)

如果想得到一个可由 .NET 运行库加载的非可执行文件(如 DLL), 就必须把它编译为一个库。如果把 C# 文件编译为一个模块, 就不会创建任何程序集。虽然模块不能由运行库加载, 但可以使用 /addmodule 选项编译到另一个清单中。

另一个需要注意的选项是 /out, 该选项可以指定由编译器生成的输出文件名。如果没有指定 /out 选项, 编译器就会使用输入的 C# 文件名, 加上目标类型的扩展名来确定输出文件名(如 .exe 表示 Windows 或控制台应用程序, .dll 表示类库)。注意 /out 和 /t(或 /target) 选项必须放在要编译的文件名前面。

默认状态下, 如果在未引用的程序集中引用类型, 可以将 /reference 或 /r 选项与程序集的路径和文件名一起使用。下面的例子说明了如何编译类库, 并在另一个程序集中引用这个库。它包含两个文件:

- 类库
- 控制台应用程序, 该应用程序调用库中的一个类

第一个文件 MathLibrary.cs 包含 DLL 的代码, 为了简单起见, 它只包含一个公共类 MathLib 和一个方法, 该方法把两个 int 类型的数据加在一起:



namespace Wrox

```
{
    public class MathLib
    {
        public int Add(int x, int y)
        {
            return x + y;
        }
    }
}
```

代码段 MathLibrary.cs

使用下述命令把这个 C# 文件编译为 .NET DLL:

```
csc /t:library MathLibrary.cs
```

控制台应用程序 MathClient.cs 将简单地实例化这个对象, 调用其 Add() 方法, 在控制台窗口中显示结果:



可从
wrox.com
下载源代码

```
using System;

namespace Wrox
{
    class Client
    {
        public static void Main()
        {
            MathLib mathObj = new MathLib();
            Console.WriteLine(mathObj.Add(7,8));
        }
    }
}
```

代码段 MathClient.cs

使用/r 选项编译这个文件，使之指向新编译的 DLL:

```
csc MathClient.cs /r:MathLibrary.dll
```

当然，下面就可以像往常一样运行它了：在命令提示符上输入 **MathClient**，其结果是显示数字 15——加运算的结果。

2.9 控制台 I/O

现在，读者应基本熟悉了 C# 的数据类型以及控制线程如何执行操作这些数据类型的程序。本章还要使用 **Console** 类的几个静态方法来读写数据，这些方法在编写基本的 C# 程序时非常有效，下面就详细介绍它们。

要从控制台窗口中读取一行文本，可以使用 **Console.ReadLine()** 方法，它会从控制台窗口中读取一个输入流（在用户按回车键时停止），并返回输入的字符串。写入控制台也有两个对应的方法，前面已经使用过它们：

- **Console.Write()** 方法将指定的值写入控制台窗口。
- **Console.WriteLine()** 方法类似，但在输出结果的最后添加一个换行符。

所有预定义类型（包括 **object**）都有这些方法的各种形式（重载），所以在大多数情况下，在显示值之前不必把它们转换为字符串。

例如，下面的代码允许用户输入一行文本，并显示该文本：

```
string s = Console.ReadLine();
Console.WriteLine(s);
```

Console.WriteLine() 还允许用与 C 的 **printf()** 函数类似的方式显示格式化的输出结果。要以这种方式使用 **WriteLine()**，应传入许多参数。第一个参数是花括号中包含标记的字符串，在这个花括号中，要把后续参数插入到文本中。每个标记都包含一个基于 0 的索引，表示列表中参数的序号。例如，**{0}** 表示列表中的第一个参数，所以下面的代码：

```
int i = 10;
```

```
int j = 20;
Console.WriteLine("{0} plus {1} equals {2}", i, j, i + j);
```

会显示:

```
10 plus 20 equals 30
```

也可以为值指定宽度, 调整文本在该宽度中的位置, 正值表示右对齐, 负值表示左对齐。为此可以使用格式{*n,w*}, 其中*n*是参数索引, *w*是宽度值。

```
int i = 940;
int j = 73;
Console.WriteLine(" {0,4}\n+{1,4}\n —— \n {2,4}", i, j, i + j);
```

结果如下:

```
    940
+   73
-----
   1013
```

最后, 还可以添加一个格式字符串以及一个可选的精度值。这里没有列出格式字符串的完整列表, 因为如第9章所述, 我们可以定义自己的格式字符串。但用于预定义类型的主要格式字符串如表2-9所示。

表 2-9

字 符 串	说 明
C	本地货币格式
D	十进制格式, 把整数转换为以 10 为基数的数, 如果给定一个精度说明符, 就加上前导 0
E	科学计数法(指数)格式。精度说明符设置小数位数(默认为 6)。格式字符串的大小写(e 或 E)确定指数符号的大小写
F	固定点格式, 精度说明符设置小数位数, 可以为 0
G	普通格式, 使用 E 或 F 格式取决于哪种格式较简单
N	数字格式, 用逗号表示千分符, 例如 32 767.44
P	百分数格式
X	十六进制格式, 精度说明符用于加上前导 0

注意除 e/E 之外, 格式字符串都不需要考虑大小写。

如果要使用格式字符串, 应把它放在给出参数个数和字段宽度的标记后面, 并用一个冒号把它们分隔开。例如, 要把 decimal 值格式化为货币格式, 且使用计算机上的地区设置, 其精度为两位小数, 则使用 C2:

```
decimal i = 940.23m;
decimal j = 73.7m;
Console.WriteLine(" {0,9:C2}\n+{1,9:C2}\n —— \n {2,9:C2}", i, j, i + j);
```

在美国, 其结果是:

```

    $940.23
+   $73.70
-----
    $1,013.93

```

最后一个技巧是，可以使用占位符来代替这些格式字符串，例如：

```

double d = 0.234;
Console.WriteLine("{0:#.00}", d);

```

其结果为.23，因为如果在符号(#)的位置上没有字符，就会忽略该符号(#)，如果在 0 的位置上有一个字符，就用这个字符代替 0，否则就显示 0。

2.10 使用注释

本节的内容是给代码添加注释，该主题表面看来十分简单，但实际可能很复杂。

2.10.1 源文件中的内部注释

本章开头提到过，C#使用传统的C风格注释方式：单行注释使用//...，多行注释使用/*...*/：

```

// This is a singleline comment
/* This comment
   spans multiple lines */

```

单行注释中的任何内容，即从//开始一直到行尾的内容都会被编译器忽略。多行注释中“/*”和“*/”之间的所有内容也会被忽略。显然不能在多行注释中包含“*/”组合，因为这会被当作注释的结尾。

实际上，可以把多行注释放在一行代码中：

```

Console.WriteLine(/*Here's a comment! */ "This will compile.");

```

像这样的内联注释在使用时应小心，因为它们会使代码难以理解。但这样的注释在调试时是非常有用的，例如，在运行代码时要临时使用另一个值：

```

DoSomething(Width, /*Height*/ 100);

```

当然，字符串面值中的注释字符会按照一般的字符来处理：

```

string s = "/* This is just a normal string */";

```

2.10.2 XML 文档

如前所述，除了C风格的注释外，C#还有一个非常出色的功能，本章将讨论这一功能：根据特定的注释自动创建XML格式的文档说明。这些注释都是单行注释，但都以3条斜杠(///)开头，而不是通常的两条斜杠。在这些注释中，可以把包含类型和类型成员的文档说明的XML标记放在代码中。

编译器可以识别表 2-10 所示的标记。

表 2-10

标 记	说 明
<>	把行中的文本标记为代码, 例如<c>int i = 10;</c>
<code>	把多行标记为代码
<example>	标记为一个代码示例
<exception>	说明一个异常类(编译器要验证其语法)
<include>	包含其他文档说明文件的注释(编译器要验证其语法)
<list>	把列表插入到文档中
<param>	标记方法的参数(编译器要验证其语法)
<paramref>	表示一个单词是方法的参数(编译器要验证其语法)
<permission>	说明对成员的访问(编译器要验证其语法)
<remarks>	给成员添加描述
<returns>	说明方法的返回值
<see>	提供对另一个参数的交叉引用(编译器要验证其语法)
<seealso>	提供描述中的“参见”部分(编译器要验证其语法)
<summary>	提供类型或成员的简短小结
<value>	描述属性

要了解它们的工作方式, 可以在上一节的 MathLibrary.cs 文件中添加一些 XML 注释。我们给类及其 Add() 方法添加一个 <summary> 元素, 也给 Add() 方法添加一个 <returns> 元素和两个 <param> 元素:

```
// MathLib.cs
namespace Wrox
{
    ///<summary>
    ///  Wrox.Math class.
    ///  Provides a method to add two integers.
    ///</summary>
    public class Math
    {
        ///<summary>
        ///  The Add method allows us to add two integers
        ///</summary>
        ///<returns>Result of the addition (int)</returns>
        ///<param name="x">First number to add</param>
        ///<param name="y">Second number to add</param>
        public int Add(int x, int y)
        {
            return x + y;
        }
    }
}
```

C#编译器可以把 XML 元素从特定的注释中提取出来, 并使用它们生成一个 XML 文件。要让编译器为程序集生成 XML 文档, 需在编译时指定/doc 选项, 后跟要创建的文件名:

```
csc /t:library /doc:MathLibrary.xml MathLibrary.cs
```

如果 XML 注释没有生成格式正确的 XML 文档, 编译器就生成一个错误。

上面的代码会生成一个 XML 文件 Math.xml, 如下所示。



可从
wrox.com
下载源代码

```
<?xml version="1.0"?>
<doc>
  <assembly>
    <name>MathLibrary</name>
  </assembly>
  <members>
    <member name="T:Wrox.MathLibrary">
      <summary>
        Wrox.MathLibrary class.
        Provides a method to add two integers.
      </summary>
    </member>
    <member name=
      "M:Wrox.MathLibrary.Add(System.Int32,System.Int32)">
      <summary>
        The Add method allows us to add two integers.
      </summary>
      <returns>Result of the addition (int)</returns>
      <param name="x">First number to add</param>
      <param name="y">Second number to add</param>
    </member>
  </members>
</doc>
```

代码段 MathLibrary.xml

注意, 编译器自行完成了一些工作——它创建了一个<assembly>元素, 并为该文件中的每个类型或类型成员添加一个<member>元素。每个<member>元素都有一个 name 特性, 该特性的值是成员的全名, 前面有一个字母, 含义如下: "T:"表示一个类型, "F:" 表示一个字段, "M:" 表示一个成员。

2.11 C#预处理器指令

除了前面介绍的常用关键字外, C#还有许多名为“预处理器指令”的命令。这些命令从来不会转化为可执行代码中的命令, 但会影响编译过程的各个方面。例如, 使用预处理器指令可以禁止编译器编译代码的某一部分。如果计划发布两个版本的代码, 即基本版本和拥有更多功能的企业版本, 就可以使用这些预处理器指令。在编译软件的基本版本时, 使用预处理器指令可以禁止编译器编译与额外功能相关的代码。另外, 在编写提供调试信息的代码时, 也可以使用预处理器指令。实际上, 在销售软件时, 一般不希望编译这部分代码。

预处理器指令的开头都有符号#。



C++开发人员应知道，在C和C++中预处理器指令非常重要，但是，在C#中，并没有那么多的预处理器指令，它们的使用也不太频繁。C#提供了其他机制来实现许多C++指令的功能，如定制特性。还要注意，C#并没有一个像C++那样的独立预处理器，所谓的预处理器指令实际上是由编译器处理的。尽管如此，C#仍保留了一些预处理器指令名称，因为这些命令会让人觉得就是预处理器。

下面简要介绍预处理器指令的功能。

2.11.1 #define 和 #undef

#define 的用法如下所示：

```
#define DEBUG
```

它告诉编译器存在给定名称的符号，在本例中是DEBUG。这有点类似于声明一个变量，但这个变量并没有真正的值，只是存在而已。这个符号不是实际代码的一部分，而只在编译器编译代码时存在。在C#代码中它没有任何意义。

#undef 正好相反——它删除符号的定义：

```
#undef DEBUG
```

如果符号不存在，#undef 就没有任何作用。同样，如果符号已经存在，则#define 也不起作用。必须把#define 和#undef 命令放在C#源文件的开头位置，在声明要编译的任何对象的代码之前。#define 本身并没有什么用，但与其他预处理器指令(特别是#if)结合使用时，它的功能就非常强大了。



这里应注意一般C#语法的一些变化。预处理器指令不用分号结束，一般一行上只有一条命令。这是因为对于预处理器指令，C#不再要求命令使用分号进行分隔。如果它遇到一条预处理器指令，就会假定下一条命令在下一行上。

2.11.2 #if, #elif, #else 和 #endif

这些指令告诉编译器是否要编译某个代码块。考虑下面的方法：

```
int DoSomeWork(double x)
{
    // do something
    #if DEBUG
        Console.WriteLine("x is " + x);
    #endif
}
```

这段代码会像往常那样编译,但 `Console.WriteLine` 命令包含在 `#if` 子句内。这行代码只有在前面的 `#define` 命令定义了符号 `DEBUG` 后才执行。当编译器遇到 `#if` 语句后,将先检查相关的符号是否存在,如果符号存在,就编译 `#if` 子句中的代码。否则,编译器会忽略所有的代码,直到遇到匹配的 `#endif` 指令为止。一般是在调试时定义符号 `DEBUG`,把与调试相关的代码放在 `#if` 子句中。在完成了调试后,就把 `#define` 语句注释掉,所有的调试代码会奇迹般地消失,可执行文件也会变小,最终用户不会被这些调试信息弄糊涂(显然,要做更多的测试,确保代码在没有定义 `DEBUG` 的情况下也能工作)。这项技术在 C 和 C++ 编程中十分常见,称为条件编译(conditional compilation)。

`#elif` (=else if) 和 `#else` 指令可以用在 `#if` 块中,其含义非常直观。也可以嵌套 `#if` 块:

```
#define ENTERPRISE
#define W2K

// further on in the file

#if ENTERPRISE
    // do something
    #if W2K
        // some code that is only relevant to enterprise
        // edition running on W2K
    #endif
#elif PROFESSIONAL
    // do something else
#else
    // code for the leaner version
#endif
```

与 C++ 中的情况不同,使用 `#if` 不是有条件地编译代码的唯一方式, C# 还通过 `Conditional` 特性提供了另一种机制,详见第 14 章。

`#if` 和 `#elif` 还支持一组逻辑运算符 “!”、“—”、“!=” 和 “||”。如果符号存在,就被认为是 `true`,否则为 `false`,例如:

```
#if W2K && (ENTERPRISE==false) // if W2K is defined but ENTERPRISE isn't
```

2.11.3 #warning 和 #error

另两个非常有用的预处理器指令是 `#warning` 和 `#error`,当编译器遇到它们时,会分别产生警告或错误。如果编译器遇到 `#warning` 指令,会给用户显示 `#warning` 指令后面的文本,之后编译继续进行。如果编译器遇到 `#error` 指令,就会给用户显示后面的文本,作为一条编译错误消息,然后会立即退出编译,不会生成 IL 代码。

使用这两条指令可以检查 `#define` 语句是不是做错了什么事,使用 `#warning` 语句可以让自己回顾一下进行了哪些操作:

```
#if DEBUG && RELEASE
```

```

    #error "You've defined DEBUG and RELEASE simultaneously! "
#endif

#warning "Don't forget to remove this line before the boss tests the code! "
    Console.WriteLine("I hate this job*");

```

2.11.4 #region 和 #endregion

#region 和 **#endregion** 指令用于把一段代码标记为有给定名称的一个块，如下所示。

```

#region Member Field Declarations
    int x;
    double d;
    Currency balance;
#endregion

```

这看起来似乎没有什么用，它不影响编译过程。这些指令的优点是它们可以被某些编辑器识别，包括 Visual Studio .NET 编辑器。这些编辑器可以使用这些指令使代码在屏幕上更好地布局。第 16 章会详细介绍它们。

2.11.5 #line

#line 指令可以用于改变编译器在警告和错误信息中显示的文件名和行号信息。这条指令用得并不多。如果编写代码时，在把代码发送给编译器前，要使用某些软件包改变输入的代码，就可以使用这个指令，因为这意味着编译器报告的行号或文件名与文件中的行号或编辑的文件名不匹配。**#line** 指令可以用于还原这种匹配。也可以使用语法 **#line default** 把行号还原为默认的行号：

```

#line 164 "Core.cs" // we happen to know this is line 164 in the file
                    // Core.cs, before the intermediate
                    // package mangles it.

// later on

#line default      // restores default line numbering

```

2.11.6 #pragma

#pragma 指令可以抑制或还原指定的编译警告。与命令行选项不同，**#pragma** 指令可以在类或方法级别执行，对抑制警告的内容和抑制的时间进行更精细的控制。下面的例子禁止“字段未使用”警告，然后在编译 **MyClass** 类后还原该警告。

```

#pragma warning disable 169
public class MyClass
{
    int neverUsedField;
}
#pragma warning restore 169

```

2.12 C#编程规则

本节介绍编写 C#程序时应该遵循的准则。

2.12.1 关于标识符的规则

本节将讨论变量、类、方法等的命名规则。注意本节所介绍的规则不仅是准则，也是 C#编译器强制使用的。

标识符是给变量、用户定义的类型(如类和结构)和这些类型的成员指定的名称。标识符区分大小写，所以 `interestRate` 和 `InterestRate` 是不同的变量。确定在 C#中可以使用什么标识符有两条规则：

- 尽管可以包含数字字符，但它们必须以字母或下划线开头。
- 不能把 C#关键字用作标识符。

C#包含如表 2-11 所示的保留关键字。

表 2-11

<code>abstract</code>	<code>do</code>	<code>in</code>	<code>protected</code>	<code>true</code>
<code>as</code>	<code>double</code>	<code>int</code>	<code>public</code>	<code>try</code>
<code>base</code>	<code>else</code>	<code>interface</code>	<code>readonly</code>	<code>typeof</code>
<code>bool</code>	<code>enum</code>	<code>internal</code>	<code>ref</code>	<code>uint</code>
<code>break</code>	<code>event</code>	<code>is</code>	<code>return</code>	<code>ulong</code>
<code>byte</code>	<code>explicit</code>	<code>lock</code>	<code>sbyte</code>	<code>unchecked</code>
<code>case</code>	<code>extern</code>	<code>long</code>	<code>sealed</code>	<code>unsafe</code>
<code>catch</code>	<code>false</code>	<code>namespace</code>	<code>short</code>	<code>ushort</code>
<code>char</code>	<code>finally</code>	<code>new</code>	<code>sizeof</code>	<code>using</code>
<code>checked</code>	<code>fixed</code>	<code>null</code>	<code>stackalloc</code>	<code>virtual</code>
<code>class</code>	<code>float</code>	<code>object</code>	<code>static</code>	<code>volatile</code>
<code>const</code>	<code>for</code>	<code>operator</code>	<code>string</code>	<code>void</code>
<code>continue</code>	<code>foreach</code>	<code>out</code>	<code>struct</code>	<code>while</code>
<code>decimal</code>	<code>goto</code>	<code>override</code>	<code>switch</code>	
<code>default</code>	<code>if</code>	<code>params</code>	<code>this</code>	
<code>delegate</code>	<code>implicit</code>	<code>private</code>	<code>throw</code>	

如果需要把某一保留字用作标识符(例如，访问一个用另一种语言编写的类)，那么可以在标识符的前面加上前缀符号@，告知编译器其后的内容是一个标识符，而不是 C#关键字(所以 `abstract` 不是有效的标识符，`@abstract` 才是)。

最后，标识符也可以包含 Unicode 字符，用语法 `uXXXX` 来指定，其中 `XXXX` 是 Unicode 字符的 4 位十六进制编码。下面是有效标识符的一些例子：

- `Name`
- `überfluß`

- `_Identifier`
- `\u005fIdentifier`

最后两个标识符完全相同,可以互换(因为 005f 是下划线字符的 Unicode 代码),所以这些标识符在同一个作用域内不要声明两次。注意虽然从语法上看,在标识符中可以使用下划线字符,但大多数情况下最好不要这么做,因为它不符合 Microsoft 的变量命名规则,这种命名规则可以确保开发人员使用相同的命名约定,易于阅读他人编写的代码。

2.12.2 用法约定

在任何开发语言中,通常有一些传统的编程风格。这些风格不是语言自身的一部分,而是约定,例如,变量如何命名,类、方法或函数如何使用等。如果使用某语言的大多数开发人员都遵循相同的约定,不同的开发人员就很容易理解彼此的代码,这一般有助于程序的维护。约定主要取决于语言和环境。例如,在 Windows 平台上编程的 C++ 开发人员一般使用前缀 `psz` 或 `lpsz` 表示字符串: `char *pszResult; char *lpszMessage;` 但在 UNIX 系统上,则不使用任何前缀: `char *Result; char *Message;`。

从本书中的示例代码中可以总结出, C# 中的约定是命名变量时不使用任何前缀: `string Result; string Message;`。



变量名用带有前缀字母来表示某种数据类型,这种约定称为 Hungarian 表示法。这样,其他阅读该代码的开发人员就可以立即从变量名中了解它代表什么数据类型。在有了智能编辑器和 IntelliSense 之后,人们普遍认为 Hungarian 表示法是多余的。

但是,在许多语言中,用法约定是从语言的使用过程中逐渐演变而来的, Microsoft 编写的 C# 和整个 .NET Framework 都有非常多的用法约定,详见 .NET/C# MSDN 文档。这说明,从一开始, .NET 程序就有非常高的互操作性,开发人员可以以此来理解代码。用法规则还得益于 20 年来面向对象编程的发展,因此相关的新闻组已经仔细考虑了这些用法规则,而且已经为开发团体所接受。所以我们应遵守这些约定。

但要注意,这些规则与语言规范不同。用户应尽可能遵循这些规则。但如果有很好的理由不遵循它们,也不会有什么問題。例如,不遵循这些用法约定,也不会出现编译错误。一般情况下,如果不遵循用法规则,就必须有一个充分的理由。规则应是一个正确的决策,而不是一种束缚。在阅读本书的后续内容时,应注意到在本书的许多示例中,都没有遵循该约定,这通常是因为某些规则适用于大型程序,而不适合于本书中的小示例。如果编写一个完整的软件包,就应遵循这些规则,但它们并不适合于只有 20 行代码的独立程序。在许多情况下,遵循约定会使这些示例难以理解。

编程风格的规则非常多。这里只介绍一些比较重要的规则,以及最适合于用户的规则。如果用户要让代码完全遵循用法规则,就需要参考 MSDN 文档。

1. 命名约定

使程序易于理解的一个重要方面是给对象选择命名的方式，包括变量、方法、类、枚举和名称空间的命名方式。

显然，这些名称应反映对象的功能，且不与其他名称冲突。在 .NET Framework 中，一般规则也是变量名要反映变量实例的功能，而不反映数据类型。例如，`height` 就是一个比较好的变量名，而 `integerValue` 就不太好。但是，这种规则是一种理想状态，很难达到。在处理控件时，大多数情况下使用 `confirmationDialog` 和 `chooseEmployeeListBox` 等变量名比较好，这些变量名说明了变量的数据类型。

名称的约定包括以下几个方面：

(1) 名称的大小写

在许多情况下，名称都应使用 **Pascal** 大小写形式。**Pascal** 大小写形式指名称中单词的首字母大写，如 `EmployeeSalary`、`ConfirmationDialog`、`PlainTextEncoding`。注意，名称空间和类，以及基类中的成员等的名称都应遵循该规则，最好不要使用带有下划线字符的单词，即名称不应是 `employee_salary`。其他语言中常量的名称常常全部大写，但在 C# 中最好不要这样，因为这种名称很难阅读，而应全部使用 **Pascal** 大小写形式的命名约定：

```
const int MaximumLength;
```

我们还推荐使用另一种大小写模式：**camel** 大小写形式。这种形式类似于 **Pascal** 大小写形式，但名称中第一个单词的首字母不大写，如 `employeeSalary`、`confirmationDialog`、`plainTextEncoding`。有 3 种情况可以使用 **camel** 大小写形式。

- 类型中所有私有成员字段的名称都应是 camel 大小写形式：

```
private int subscriberId;
```

但要注意成员字段的前缀名常常用一条下划线开头：

```
private int _subscriberId;
```

- 传递给方法的所有参数的名称都应是 camel 大小写形式：

```
public void RecordSale(string salesmanName, int quantity);
```

- **camel** 大小写形式也可以用于区分同名的两个对象——比较常见的情况是属性封装一个字段：

```
private string employeeName;

public string EmployeeName
{
    get
    {
        return employeeName;
    }
}
```

如果这么做，则私有成员总是使用 **camel** 大小写形式，而公有的或受保护的成员总是使用 **Pascal** 大小写形式，这样使用这段代码的其他类就只能使用 **Pascal** 大小写形式的名称了(除了参数名以外)。

还要注意大小写问题。C#区分大小写，所以在 C# 中，仅大小写不同的名称在语法上是正确的，如上面的例子。但是，有时可能从 Visual Basic .NET 应用程序中调用程序集，而 Visual Basic .NET 不区分大小写，如果使用仅大小写不同的名称，就必须使这两个名称不能在程序集的外部访问(上例是可行的，因为仅私有变量使用了 camel 大小写形式的名称)。否则，Visual Basic .NET 中的其他代码就不能正确使用这个程序集。

(2) 名称的风格

名称的风格应保持一致。例如，如果类中的一个方法被命名为 ShowConfirmationDialog()，另一个方法就不能被命名为 ShowDialogWarning()或 WarningDialogShow()，而应是 ShowWarningDialog()。

(3) 名称空间的名称

名称空间的名称非常重要，一定要仔细考虑，以避免一个名称空间的名称与其他名称空间同名。记住，名称空间的名称是 .NET 区分共享程序集中对象名的唯一方式。如果软件包的名称空间使用的名称与另一个软件包相同，而这两个软件包都安装在一台计算机上，就会出问题。因此，最好用自己的公司名创建顶级的名称空间，再嵌套技术范围较窄、用户所在小组或部门或者类所在软件包的名称空间。Microsoft 建议使用如下的名称空间：<CompanyName>. <TechnologyName>，例如：

```
WeaponsOfDestructionCorp.RayGunControllers
WeaponsOfDestructionCorp.Viruses
```

(4) 名称和关键字

名称不应与任何关键字冲突，这非常重要。实际上，如果在代码中，试图给某一项指定与 C# 关键字同名的名称，就会出现语法错误，因为编译器会假定该名称表示一条语句。但是，由于类可能由其他语言编写的代码访问，所以不能使用其他 .NET 语言中的关键字作为对应的名称。一般来说，C++ 关键字类似于 C# 关键字，不太可能与 C++ 混淆，只有 Visual C++ 常用的关键字则以两个下划线字符开头。与 C# 一样，C++ 关键字都是小写字母，如果要遵循公有类和成员使用 Pascal 风格的名称的约定，则在它们的名称中至少有一个字母大写，因此不会与 C++ 关键字冲突。另一方面，Visual Basic .NET 的问题会多一些，因为 Visual Basic .NET 的关键字要比 C# 的多，而且它不区分大小写，不能依赖于 Pascal 风格的名称来区分类和成员。

表 2-12 列出了 Visual Basic .NET 中的关键字和标准函数调用，无论对 C# 公有类使用什么大小写组合，这些名称都不应使用。

表 2-12

Abs	Do	Loc	RGB
Add	Double	Local	Right
AddHandler	Each	Lock	Rmdir
AddressOf	Else	LOF	Rnd
Alias	Elseif	Log	RTrim
And	Empty	Long	SaveSettings
Ansi	End	Loop	Second
AppActivate	Enum	LTrim	Seek

(续表)

Append	EOF	Me	Select
As	Erase	Mid	SetAttr
Asc	Err	Minute	SetException
Assembly	Error	MIRR	Shared
Atan	Event	MkDir	Shell
Auto	Exit	Module	Short
Beep	Exp	Month	Sign
Binary	Explicit	MustInherit	Sin
BitAnd	ExternalSource	MustOverride	Single
BitNot	False	MyBase	SLN
BitOr	FileAttr	MyClass	Space
BitXor	FileCopy	Namespace	Spc
Boolean	FileDateTime	New	Split
ByRef	FileLen	Next	Sqrt
Byte	Filter	Not	Static
ByVal	Finally	Nothing	Step
Call	Fix	NotInheritable	Stop
Case	For	NotOverridable	Str
Catch	Format	Now	StrComp
CBool	FreeFile	NPer	StrConv
CByte	Friend	NPV	Strict
CDate	Function	Null	String
CDbl	FV	Object	Structure
CDec	Get	Oct	Sub
ChDir	GetAllSettings	Off	Switch
ChDrive	GetAttr	On	SYD
Choose	GetException	Open	SyncLock
Chr	GetObject	Option	Tab
CInt	GetSetting	Optional	Tan
Class	GetType	Or	Text
Clear	GoTo	Overloads	Then
CLng	Handles	Overridable	Throw
Close	Hex	Overrides	TimeOfDay
Collection	Hour	ParamArray	Timer
Command	If	Pmt	TimeSerial

(续表)

Compare	Iif	PPmt	TimeValue
Const	Implements	Preserve	To
Cos	Imports	Print	Today
CreateObject	In	Private	Trim
CShort	Inherits	Property	Try
CSng	Input	Public	TypeName
CStr	InStr	Put	TypeOf
CurDir	Int	PV	UBound
Date	Integer	QBColor	UCase
DateAdd	Interface	Raise	Unicode
DateDiff	Ipmt	RaiseEvent	Unlock
DatePart	IRR	Randomize	Until
DateSerial	Is	Rate	Val
DateValue	IsArray	Read	Weekday
Day	IsDate	ReadOnly	While
DDB	IsDBNull	ReDim	Width
Decimal	IsNumeric	Remove	With
Declare	Item	RemoveHandler	WithEvents
Default	Kill	Rename	Write
Delegate	Lcase	Replace	WriteOnly
DeleteSetting	Left	Reset	Xor
Dim	Lib	Resume	Year
Dir	Line	Return	

2. 属性和方法的使用

类中出现混乱的一个方面是某个特定数量是用属性还是方法来表示。这没有硬性规定，但一般情况下，如果该对象的外观像一个变量，就应使用属性来表示它(属性详见第3章)，即：

- 客户端代码应能读取它的值，最好不要使用只写属性，例如，应使用 `SetPassword()` 方法，而不是 `Password` 只写属性。
- 读取该值不应花太长的时间。实际上，如果它是一个属性，通常表明读取过程花的时间相对较短。
- 读取该值不应有任何细微的和不希望的负面效应。设置属性的值，不应有与该属性不直接相关的负面效应。设置对话框的宽度会改变该对话框在屏幕上的外观，这是可以的，因为它与有问题的属性相关。
- 可以按照任何顺序设置属性。尤其在设置属性时，最好不要因为还没有设置另一个相关的属性而抛出一个异常。例如，如果为了使用访问数据库的类，需要设置 `ConnectionString`、`UserName` 和 `Password`，应确保已经实现了该类，这样用户才能按照任何顺序设置它们。

- 顺序读取属性也应有相同的效果。如果属性的值可能会出现预料不到的改变, 就应把它编写为一个方法。在监控汽车的运动的类中, 把 `speed` 设置为属性就不合适, 而应使用 `GetSpeed()` 方法; 另一方面, 应把 `Weight` 和 `EngineSize` 设置为属性, 因为对于给定的对象, 它们是不变的。

如果要编码的相关项满足上述所有条件, 就把它设置为属性, 否则就应使用方法。

3. 字段的用法

字段的用法非常简单。字段应总是私有的, 但在某些情况下也可以把常量或只读字段设置为公有。原因是如果把字段设置为公有, 就不利于在以后扩展或修改类。

遵循上面的规则就可以培养良好的编程习惯, 而且这些规则应与面向对象编程的风格一起使用。

最后要记住以下有用的备注: **Microsoft** 在保持一致性方面相当谨慎, 在编写 .NET 基类时遵循了它自己的规则。在编写 .NET 代码时应很好地遵循这些规则, 对于基类来说, 就是要弄清楚类、成员、名称空间的命名方式和类层次结构的工作方式等。类与基类之间的一致性有助于提高可读性和可维护性。

2.13 小结

本章介绍了一些 C# 基本语法, 包括编写简单的 C# 程序需要掌握的内容。我们讲述了许多基础知识, 但其中有许多是熟悉 C 风格语言(甚至 JavaScript)的开发人员能立即领悟的。

C# 语法与 C++/Java 语法非常类似, 但仍存在一些细微区别。在许多领域, 将这些语法与功能结合起来会提高编码速度, 如高质量的字符串处理功能。C# 还有一个已定义的强类型系统, 该系统基于值类型和引用类型的区别。第 3 章和第 4 章将介绍 C# 的面向对象编程特性。

第 3 章

对象和类型

本章内容:

- 类和结构的区别
- 类成员
- 按值和按引用传送参数
- 方法重载
- 构造函数和静态构造函数
- 只读字段
- 部分类
- 静态类
- Object 类, 其他类型都从该类派生而来

到目前为止, 我们介绍了组成 C#语言的主要模块, 包括变量、数据类型和程序流语句, 并简要介绍了一个只包含 Main()方法的完整小例子。但还没有介绍如何把这些内容组合在一起, 构成一个完整的程序, 其关键就在于对类的处理。这就是本章的主题。第 4 章将介绍继承以及与继承相关的特性。



本章将讨论与类相关的基本语法, 但假定您已经熟悉了使用类的基本原则, 例如, 知道构造函数或属性的含义, 因此本章主要阐述如何把这些原则应用于 C#代码。

3.1 类和结构

类和结构实际上都是创建对象的模板, 每个对象都包含数据, 并提供了处理和访问数据的方法。类定义了类的每个对象(称为实例)可以包含什么数据和功能。例如, 如果一个类表示一个顾客, 就可以定义字段 CustomerID、FirstName、LastName 和 Address, 以包含该顾客的信息。还可以定义处理在这些字段中存储的数据的功能。接着, 就可以实例化表示某个顾客的类的对象, 为这个实例设置相关字段的值, 并使用其功能。

```
class PhoneCustomer
{
    public const string DayOfSendingBill = "Monday";
```

```

    public int CustomerID;
    public string FirstName;
    public string LastName;
}

```

结构与类的区别是它们在内存中的存储方式、访问方式(类是存储在堆(heap)上的引用类型,而结构是存储在栈(stack)上的值类型)和它们的一些特征(如结构不支持继承)。较小的数据类型使用结构可提高性能。但在语法上,结构与类非常相似,主要的区别是使用关键字 `struct` 代替 `class` 来声明结构。例如,如果希望所有的 `PhoneCustomer` 实例都分布在栈上,而不是分布在托管堆上,就可以编写下面的语句:

```

struct PhoneCustomerStruct
{
    public const string DayOfSendingBill = "Monday";
    public int CustomerID;
    public string FirstName;
    public string LastName;
}

```

对于类和结构,都使用关键字 `new` 来声明实例:这个关键字创建对象并对其进行初始化。在下面的例子中,类和结构的字段值都默认为 0:

```

PhoneCustomer myCustomer = new PhoneCustomer(); // works for a class
PhoneCustomerStruct myCustomer2 = new PhoneCustomerStruct(); // works for a struct

```

在大多数情况下,类要比结构常用得多。因此,我们先讨论类,然后指出类和结构的区别,以及选择使用结构而不使用类的特殊原因。但除非特别说明,否则就可以假定用于类的代码也适用于结构。

3.2 类

类中的数据和函数称为类的成员。Microsoft 的正式术语对数据成员和函数成员进行了区分。除了这些成员外,类还可以包含嵌套的类型(如其他类)。成员的可访问性可以是 `public`、`protected`、`internal`、`protected`、`private` 或 `internal`。第 5 章将详细解释各种可访问性。

3.2.1 数据成员

数据成员是包含类的数据——字段、常量和事件的成员。数据成员可以是静态数据。类成员总是实例成员,除非用 `static` 进行显式的声明。

字段是与类相关的变量。前面的例子已经使用了 `PhoneCustomer` 类中的字段。

一旦实例化 `PhoneCustomer` 对象,就可以使用语法 `Object.FieldName` 来访问这些字段,如下例所示:

```

PhoneCustomer Customer1 = new PhoneCustomer();
Customer1.FirstName = "Simon";

```

常量与类的关联方式同变量与类的关联方式。使用 `const` 关键字来声明常量。如果把它声明为 `public`，就可以在类的外部访问它。

```
class PhoneCustomer
{
    public const string DayOfSendingBill = "Monday";
    public int CustomerID;
    public string FirstName;
    public string LastName;
}
```

事件是类的成员，在发生某些行为(如改变类的字段或属性，或者进行了某种形式的用户交互操作)时，它可以让对象通知调用方。客户可以包含所谓“事件处理程序”的代码来响应该事件。第8章将详细介绍事件。

3.2.2 函数成员

函数成员提供了操作类中数据的某些功能，包括方法、属性、构造函数和终结器(`finalizer`)、运算符以及索引器。

- 方法是与某个类相关的函数，与数据成员一样，函数成员默认为实例成员，使用 `static` 修饰符可以把方法定义为静态方法。
- 属性是可以从客户端访问的函数组，其访问方式与访问类的公共字段类似。C#为读写类中的属性提供了专用语法，所以不必使用那些名称中嵌有 `Get` 或 `Set` 的方法。因为属性的这种语法不同于一般函数的语法，在客户端代码中，虚拟的对象被当做实际的东西。
- 构造函数是在实例化对象时自动调用的特殊函数。它们必须与所属的类同名，且不能有返回类型。构造函数用于初始化字段的值。
- 终结器类似于构造函数，但是在 CLR 检测到不再需要某个对象时调用它。它们的名称与类相同，但前面有一个“~”符号。不可能预测什么时候调用终结器。第13章将介绍终结器。
- 运算符执行的最简单的操作就是加法和减法。在两个整数相加时，严格地说，就是对整数使用“+”运算符。C#还允许指定把已有的运算符应用于自己的类(运算符重载)。第7章将详细论述运算符。
- 索引器允许对象以数组或集合的方式进行索引。

1. 方法

注意，正式的 C# 术语区分函数和方法。在 C# 术语中，“函数成员”不仅包含方法，而且也包含类或结构的一些非数据成员，如索引器、运算符、构造函数和析构函数等，甚至还有属性。这些都不是数据成员，字段、常量和事件才是数据成员。

(1) 方法的声明

在 C# 中，方法的定义包括任意方法修饰符(如方法的可访问性)、返回值的类型，然后依次是方法名和输入参数的列表(用圆括号括起来)和方法体(用花括号括起来)。

```
[modifiers] return_type MethodName([parameters])
{
```

```
// Method body
}
```

每个参数都包括参数的类型名和在方法体中的引用名称。但如果方法有返回值，`return` 语句就必须与返回值一起使用，以指定出口点，例如：

```
public bool IsSquare(Rectangle rect)
{
    return (rect.Height == rect.Width);
}
```

这段代码使用了一个表示矩形的 .NET 基类 `System.Drawing.Rectangle`。

如果方法没有返回值，就把返回类型指定为 `void`，因为不能省略返回类型。如果方法不带参数，仍需要在方法名的后面包含一对空的圆括号 `()`。此时 `return` 语句就是可选的——当到达右花括号时，方法会自动返回。注意方法可以包含任意多条 `return` 语句：

```
public bool IsPositive(int value)
{
    if (value < 0)
        return false;
    return true;
}
```

(2) 调用方法

在下面的例子中，`MathTest` 说明了类的定义和实例化、方法的定义和调用的语法。除了包含 `Main()` 方法的类之外，它还定义了类 `MathTest`，该类包含两个方法和一个字段。



可从
wrox.com
下载源代码

```
using System;

namespace Wrox
{
    class MainEntryPoint
    {
        static void Main()
        {
            // Try calling some static functions.
            Console.WriteLine("Pi is " + MathTest.GetPi());
            int x = MathTest.GetSquareOf(5);
            Console.WriteLine("Square of 5 is " + x);

            // Instantiate a MathTest object
            MathTest math = new MathTest(); // this is C#'s way of
                                           // instantiating a reference type

            // Call nonstatic methods
            math.value = 30;
            Console.WriteLine(
                "Value field of math variable contains " + math.value);
            Console.WriteLine("Square of 30 is " + math.GetSquare());
        }
    }
}
```

```
// Define a class named MathTest on which we will call a method
class MathTest
{
    public int value;

    public int GetSquare()
    {
        return value*value;
    }

    public static int GetSquareOf(int x)
    {
        return x*x;
    }

    public static double GetPi()
    {
        return 3.14159;
    }
}
```

代码段 MathTest.cs

运行 mathTest 示例，会得到如下结果：

```
Pi is 3.14159
Square of 5 is 25
Value field of math variable contains 30
Square of 30 is 900
```

从代码中可以看出，MathTest 类包含一个字段和一个方法，该字段包含一个数字，该方法计算该数字的平方。这个类还包含两个静态方法，一个返回 pi 的值，另一个计算作为参数传入的数字的平方。

这个类有一些功能并不是设计 C# 程序的好例子。例如，GetPi() 通常作为 const 字段来执行，而好的设计应使用目前还没有介绍的概念。

(3) 给方法传递参数

参数可以通过引用或通过值传递给方法。在变量通过引用传递给方法时，被调用的方法得到的就是这个变量，所以在方法内部对变量进行的任何改变在方法退出后仍旧有效。而如果变量通过值传送给方法，被调用的方法得到的是变量的一个相同副本，也就是说，在方法退出后，对变量进行的修改会丢失。对于复杂的数据类型，按引用传递的效率更高，因为在按值传递时，必须复制大量的数据。

在 C# 中，除非特别说明，所有的参数都通过值来传递。但是，在理解引用类型的含义时需要注意。因为引用类型的变量只包含对象的引用，将要复制的正是这个引用，而不是对象本身，所以对底层对象的修改会保留下来。相反，值类型的对象包含的是实际数据，所以传递给方法的是数据本身的副本。例如，int 通过值传递给方法，对应方法对该 int 的值所做的任何改变都没有改变原 int 对象的值。但如果把数组或其他引用类型(如类)传递给方法，对应的方法就会使用该引用改变这个

数组中的值，而新值会反射在原始数组对象上。

下面的例子 `ParameterTest.cs` 说明了这一点：



可从
wrox.com
下载源代码

```
using System;

namespace Wrox
{
    class ParameterTest
    {
        static void SomeFunction(int[] ints, int i)
        {
            ints[0] = 100;
            i = 100;
        }

        public static int Main()
        {
            int i = 0;
            int[] ints = { 0, 1, 2, 4, 8 };
            // Display the original values.
            Console.WriteLine("i = " + i);
            Console.WriteLine("ints[0] = " + ints[0]);
            Console.WriteLine("Calling SomeFunction.");

            // After this method returns, ints will be changed,
            // but i will not.
            SomeFunction(ints, i);
            Console.WriteLine("i = " + i);
            Console.WriteLine("ints[0] = " + ints[0]);
            return 0;
        }
    }
}
```

代码段 `ParameterTest.cs`

结果如下：

```
ParameterTest.exe
i = 0
ints[0] = 0
Calling SomeFunction ...
i = 0
ints[0] = 100
```

注意，`i` 的值保持不变，而在 `ints` 中改变的值在原始数组中也改变了。

注意字符串的行为方式有所不同，因为字符串是不可变的(如果改变字符串的值，就会创建一个全新的字符串)，所以字符串无法采用一般引用类型的行为方式。在方法调用中，对字符串所做的任何改变都不会影响原始字符串。这一点将在第 9 章详细讨论。

(4) `ref` 参数

如前所述，通过值传送变量是默认的，也可以迫使值参数通过引用传送给方法。为此，要使用

ref 关键字。如果把一个参数传递给方法，且这个方法的输入参数前带有 ref 关键字，则该方法对变量所做的任何改变都会影响原始对象的值：

```
static void SomeFunction(int[] ints, ref int i)
{
    ints[0] = 100;
    i = 100; // The change to i will persist after SomeFunction() exits.
}
```

在调用该方法时，还需要添加 ref 关键字：

```
SomeFunction(ints, ref i);
```

最后，C#仍要求对传递给方法的参数进行初始化，理解这一点也非常重要。在传递给方法之前，无论是按值传递，还是按引用传递，任何变量都必须初始化。

(5) out 参数

在 C 风格的语言中，函数常常能从一个例程中输出多个值，这使用输出参数实现，只要把输出的值赋予通过引用传递给方法的变量即可。通常，变量通过引用传递的初值并不重要，这些值会被函数重写，函数甚至从来没有使用过它们。

如果可以在 C#中使用这种约定，就会非常方便。但 C#要求变量在被引用前必须用一个初值进行初始化。尽管在把输入变量传递给函数前，可以用没有意义的值初始化它们，因为函数将使用真实、有意义的值初始化它们，但是这样做是没有必要的，有时甚至会引起混乱。但有一种方法能够简化 C#编译器所坚持的输入参数的初始化。

编译器使用 out 关键字来初始化。在方法的输入参数前面加上 out 前缀时，传递给该方法的变量可以不初始化。该变量通过引用传递，所以在从被调用的方法中返回时，对应方法对该变量进行的任何改变都会保留下来。在调用该方法时，还需要使用 out 关键字，与在定义该方法时一样：

```
static void SomeFunction(out int i)
{
    i = 100;
}

public static int Main()
{
    int i; // note how i is declared but not initialized.
    SomeFunction(out i);
    Console.WriteLine(i);
    return 0;
}
```

(6) 命名参数

参数一般需要按定义的顺序传送给方法。命名参数允许按任意顺序传递。所以下面的方法：

```
string FullName(string firstName, string lastName)
{
    return firstName + " " + lastName;
}
```

下面的方法调用会返回相同的全名：

```
FullName("John", "Doe");  
FullName(lastName: "Doe", firstName: "John");
```

如果方法有几个参数，就可以在同一个调用中混合使用位置参数和命名参数。

(7) 可选参数

参数也可以是可选的。必须为可选参数提供默认值。可选参数还必须是方法定义的最后一个参数。所以下面的方法声明是不正确的：

```
void TestMethod(int optionalNumber = 10, int notOptionalNumber)  
{  
    System.Console.Write(optionalNumber + notOptionalNumber);  
}
```

要使这个方法正常工作，就必须在最后定义 `optionalNumber` 参数。

(8) 方法的重载

C#支持方法的重载——方法的几个版本有不同的签名(即，方法名相同，但参数的个数和/或类型不同)。为了重载方法，只需声明同名但参数个数或类型不同的方法即可：

```
class ResultDisplayer  
{  
    void DisplayResult(string result)  
    {  
        // implementation  
    }  
  
    void DisplayResult(int result)  
    {  
        // implementation  
    }  
}
```

如果不能使用可选参数，就可以使用方法重载来达到此目的：

```
class MyClass  
{  
    int DoSomething(int x) // want 2nd parameter with default value 10  
    {  
        DoSomething(x, 10);  
    }  
  
    int DoSomething(int x, int y)  
    {  
        // implementation  
    }  
}
```

在任何语言中，对于方法重载，如果调用了错误的重载方法，就有可能出现运行错误。第 4 章将讨论如何使代码避免这些错误。现在，知道 C#在重载方法的参数方面有一些小限制即可：

- 两个方法不能仅在返回类型上有区别。
- 两个方法不能仅根据参数是声明为 `ref` 还是 `out` 来区分。

2. 属性

属性(property)的概念是: 它是一个方法或一方法, 在客户端代码看来, 它(们)是一个字段。例如 Windows 窗体的 `Height` 属性。假定有下面的代码:

```
// mainForm is of type System.Windows.Forms
mainForm.Height = 400;
```

执行这段代码时, 窗口的高度设置为 400, 因此窗口会在屏幕上重新设置大小。在语法上, 上面的代码类似于设置一个字段, 但实际上是调用了属性访问器, 它包含的代码重新设置了窗体的大小。

在 C# 中定义属性, 可以使用下面的语法:

```
public string SomeProperty
{
    get
    {
        return "This is the property value.";
    }
    set
    {
        // do whatever needs to be done to set the property.
    }
}
```

`get` 访问器不带任何参数, 且必须返回属性声明的类型。也不应为 `set` 访问器指定任何显式参数, 但编译器假定它带一个参数, 其类型也与属性相同, 并表示为 `value`。例如, 下面的代码包含一个属性 `Age`, 它设置了一个字段 `age`。在这个例子中, `age` 表示属性 `Age` 的后备变量。

```
private int age;

public int Age
{
    get
    {
        return age;
    }
    set
    {
        age = value;
    }
}
```

注意这里所用的命名约定。我们采用 C# 的区分大小写模式, 使用相同的名称, 但公有属性采用 Pascal 大小写形式命名, 并且如果存在一个等价的私有字段则它采用 camel 大小写形式命名。一些开发人员喜欢使用前缀为下划线的字段名, 如 `_foreName`, 这会为识别字段提供极大的便利。

(1) 只读和只写属性

在属性定义中省略 `set` 访问器，就可以创建只读属性。因此，如下代码把 `Name` 变成只读属性：

```
private string name;

public string Name
{
    get
    {
        return name;
    }
}
```

同样，在属性定义中省略 `get` 访问器，就可以创建只写属性。但是，这是不好的编程方式，因为这可能会使客户端代码的作者感到迷惑。一般情况下，如果要这么做，最好使用一个方法替代。

(2) 属性的访问修饰符

C# 允许给属性的 `get` 和 `set` 访问器设置不同的访问修饰符，所以属性可以有公有的 `get` 访问器和私有或受保护的 `set` 访问器。这有助于控制属性的设置方式或时间。在下面的代码示例中，注意 `set` 访问器有一个私有访问修饰符，而 `get` 访问器没有任何访问修饰符。这表示 `get` 访问器具有属性的访问级别。在 `get` 和 `set` 访问器中，必须有一个具备属性的访问级别。如果 `get` 访问器的访问级别是 `protected`，就会产生一个编译错误，因为这会使两个访问器的访问级别都不是属性。

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

(3) 自动实现的属性

如果属性的 `set` 和 `get` 访问器中没有任何逻辑，就可以使用自动实现的属性。这种属性会自动实现后备成员变量。前面 `Age` 示例的代码如下：

```
public int Age {get; set;}
```

不需要声明 `private int age`。编译器会自动创建它。

使用自动实现的属性，就不能在属性设置中验证属性的有效性。所以在上面的例子中，不能检查是否设置了无效的年龄。但必须有两个访问器。尝试把该属性设置为只读属性，就会出错：

```
public int Age {get;}
```

但是，每个访问器的访问级别可以不同。因此，下面的代码是合法的：

```
public int Age {get; private set;}
```

(4) 内联

一些开发人员可能会担心,在上一节中,我们列举了许多情况,其中标准 C# 编码方式导致了大材小用,例如,通过属性访问字段,而不是直接访问字段。这些额外的函数调用是否会增加系统开销,导致性能下降?其实,不需要担心这种编程方式会在 C# 中带来性能损失。C# 代码会编译为 IL,然后在运行时 JIT 编译为本地可执行代码。JIT 编译器可生成高度优化的代码,并在适当的时候随意地内联代码(即,用内联代码来替代函数调用)。如果实现某个方法或属性仅是调用另一个方法,或返回一个字段,则该方法或属性肯定是内联的。但要注意,在何处内联代码完全由 CLR 决定。我们无法使用像 C++ 中 `inline` 这样的关键字来控制哪些方法是内联的。

3. 构造函数

声明基本构造函数的语法就是声明一个与包含的类同名的方法,但该方法没有返回类型:

```
public class MyClass
{
    public MyClass()
    {
    }
    // rest of class definition
```

没有必要给类提供构造函数,在本书的例子中没有提供这样的构造函数。一般情况下,如果没有提供任何构造函数,编译器会在后台创建一个默认的构造函数。这是一个非常基本的构造函数,它只能把所有的成员字段初始化为标准的默认值(例如,引用类型为空引用,数值数据类型为 0, `bool` 为 `false`)。这通常就足够了,否则就需要编写自己的构造函数。

构造函数的重载遵循与其他方法相同的规则。换言之,可以为构造函数提供任意多的重载,只要它们的签名有明显的区别即可:

```
public MyClass() // zeroparameter constructor
{
    // construction code
}
public MyClass(int number) // another overload
{
    // construction code
}
```

但注意,如果提供了带参数的构造函数,编译器就不会自动提供默认的构造函数。只有在没有定义任何构造函数时,编译器才会自动提供默认的构造函数。在下面的例子中,因为定义了一个带单个参数的构造函数,编译器会假定这是可用的唯一构造函数,所以它不会隐式地提供其他构造函数:

```
public class MyNumber
{
    private int number;
    public MyNumber(int number)
    {
        this.number = number;
    }
}
```

```

    }
}

```

上面的代码还说明,一般使用 `this` 关键字区分成员字段和同名的参数。如果试图使用无参数的构造函数实例化 `MyNumber` 对象,就会得到一个编译错误:

```
MyNumber numb = new MyNumber(); // causes compilation error
```

注意,可以把构造函数定义为 `private` 或 `protected`,这样不相关的类也不能访问它们:

```
public class MyNumber
{
    private int number;
    private MyNumber(int number) // another overload
    {
        this.number = number;
    }
}
```

这个例子没有为 `MyNumber` 定义任何公有的或受保护的构造函数。这就使 `MyNumber` 不能使用 `new` 运算符在外部代码中实例化(但可以在 `MyNumber` 中编写一个公有静态属性或方法,以实例化该类)。这在下面两种情况下是有用的:

- 类仅用作某些静态成员或属性的容器,因此永远不会实例化它
- 希望类仅通过调用某个静态成员函数来实例化(这就是所谓对象实例化的类工厂方法)

(1) 静态构造函数

C#的一个新特征是也可以给类编写无参数的静态构造函数。这种构造函数只执行一次,而前面的构造函数是实例构造函数,只要创建类的对象,就会执行它。

```
class MyClass
{
    static MyClass()
    {
        // initialization code
    }
    // rest of class definition
}
```

编写静态构造函数的一个原因是,类有一些静态字段或属性,需要在第一次使用类之前,从外部源中初始化这些静态字段和属性。

.NET 运行库没有确保什么时候执行静态构造函数,所以不应把要求在某个特定时刻(例如,加载程序集时)执行的代码放在静态构造函数中。也不能预计不同类的静态构造函数按照什么顺序执行。但是,可以确保静态构造函数至多运行一次,即在代码引用类之前调用它。在 C#中,通常在第一次调用类的任何成员之前执行静态构造函数。

注意,静态构造函数没有访问修饰符,其他 C#代码从来不调用它,但在加载类时,总是由 .NET 运行库调用它,所以像 `public` 或 `private` 这样的访问修饰符就没有任何意义。出于同样原因,静态构造函数不能带任何参数,一个类也只能有一个静态构造函数。很显然,静态构造函数只能访问类的静态成员,不能访问类的实例成员。

注意,无参数的实例构造函数与静态构造函数可以在同一个类中同时定义。尽管参数列表相同,但这并不矛盾,因为在加载类时执行静态构造函数,而在创建实例时执行实例构造函数,所以何时执行哪个构造函数不会有冲突。

如果多个类都有静态构造函数,先执行哪个静态构造函数就不确定。此时静态构造函数中的代码不应依赖于其他静态构造函数的执行情况。另一方面,如果任何静态字段有默认值,就在调用静态构造函数之前指定它们。

下面用一个例子来说明静态构造函数的用法,该例子的思想基于包含用户首选项的程序(假定用户首选项存储在某个配置文件中)。为了简单起见,假定只有一个用户首选项——BackColor,它表示要在应用程序中使用的背景色。因为这里不想编写从外部数据源中读取数据的代码,所以假定该首选项在工作日的背景色是红色,在周末的背景色是绿色。程序仅在控制台窗口中显示首选项——但这足以说明静态构造函数是如何工作的。

```
namespace Wrox.ProCSharp.StaticConstructorSample
{
    public class UserPreferences
    {
        public static readonly Color BackColor;

        static UserPreferences()
        {
            DateTime now = DateTime.Now;
            if (now.DayOfWeek == DayOfWeek.Saturday
                || now.DayOfWeek == DayOfWeek.Sunday)
                BackColor = Color.Green;
            else
                BackColor = Color.Red;
        }

        private UserPreferences()
        {
        }
    }
}
```

这段代码说明了颜色首选项如何存储在静态变量中,该静态变量在静态构造函数中进行初始化。把这个字段声明为只读类型,这表示其值只能在构造函数中设置。本章后面将详细介绍只读字段。这段代码使用了 Microsoft 在 Framework 类库中支持的两个有用的结构 System.DateTime 和 System.Drawing.Color。DateTime 结构实现了静态属性 Now 和实例属性 DayOfWeek, Now 属性返回当前时间,DayOfWeek 属性计算出某个日期是星期几。Color(详见随书附赠光盘中的第 48 章)用于存储颜色,它实现了各种静态属性,如本例使用的 Red 和 Green,本例返回常用的颜色。为了使用 Color 结构,需要在编译时引用 System.Drawing.dll 程序集,且必须为 System.Drawing 名称空间添加一条 using 语句:

```
using System;
using System.Drawing;
```

用下面的代码测试静态构造函数:

```
class MainEntryPoint
{
    static void Main(string[] args)
    {
        Console.WriteLine("User-preferences: BackColor is: " +
                           UserPreferences.BackColor.ToString());
    }
}
```

编译并运行这段代码，会得到如下结果：

```
User-preferences: BackColor is: Color [Red]
```

当然，如果在周末执行上述代码，颜色设置就是 **Green**。

(2) 从构造函数中调用其他构造函数

有时，在一个类中有几个构造函数，以容纳某些可选参数，这些构造函数包含一些共同的代码。

例如，下面的情况：

```
class Car
{
    private string description;
    private uint nWheels;
    public Car(string description, uint nWheels)
    {
        this.description = description;
        this.nWheels = nWheels;
    }

    public Car(string description)
    {
        this.description = description;
        this.nWheels = 4;
    }
    // etc.
}
```

这两个构造函数初始化了相同的字段，显然，最好把所有的代码放在一个地方。C#有一个特殊的语法，称为构造函数初始化器，可以实现此目的：

```
class Car
{
    private string description;
    private uint nWheels;

    public Car(string description, uint nWheels)
    {
        this.description = description;
        this.nWheels = nWheels;
    }

    public Car(string description): this(description, 4)
    {
    }
}
```

```

}
// etc

```

这里, **this** 关键字仅调用参数最匹配的那个构造函数。注意, 构造函数初始化器在构造函数的函数体之前执行。现在假定运行下面的代码:

```
Car myCar = new Car("Proton Persona");
```

在本例中, 在带一个参数的构造函数的函数体执行之前, 先执行带两个参数的构造函数(但在本例中, 因为在带一个参数的构造函数的函数体中没有代码, 所以没有区别)。

C#构造函数初始化器可以包含对同一个类的另一个构造函数的调用(使用前面介绍的语法), 也可以包含对直接基类的构造函数的调用(使用相同的语法, 但应使用 **base** 关键字代替 **this**)。初始化器中不能有多个调用。

3.2.3 只读字段

常量的概念就是一个包含不能修改的值的变量, 常量是 C# 与大多数编程语言共有的。但是, 常量不必满足所有的要求。有时可能需要一些变量, 其值不应改变, 但在运行之前其值是未知的。C# 为这种情形提供了另一种类型的变量: 只读字段。

readonly 关键字比 **const** 灵活得多, 允许把一个字段设置为常量, 但还需要执行一些计算, 以确定它的初始值。其规则是可以在构造函数中给只读字段赋值, 但不能在其他地方赋值。只读字段还可以是一个实例字段, 而不是静态字段, 类的每个实例可以有不同的值。与 **const** 字段不同, 如果要把只读字段设置为静态, 就必须显式声明它。

如果有一个用于编辑文档的 MDI 程序, 因为要注册, 所以需要限制可以同时打开的文档数。现在假定要销售该软件的不同版本, 而且顾客可以升级他们的版本, 以便同时打开更多的文档。显然, 不能在源代码中对最大文档数进行硬编码, 而是需要一个字段表示这个最大文档数。这个字段必须是只读的——每次启动程序时, 从注册表键或其他文件存储中读取。代码如下所示:

```

public class DocumentEditor
{
    public static readonly uint MaxDocuments;

    static DocumentEditor()
    {
        MaxDocuments = DoSomethingToFindOutMaxNumber();
    }
}

```

在本例中, 字段是静态的, 因为每次运行程序的实例时, 只需存储最大文档数一次。这就是在静态构造函数中初始化它的原因。如果只读字段是一个实例字段, 就要在实例构造函数中初始化它。例如, 假定编辑的每个文档都有一个创建日期, 但不允许用户修改它(因为这会覆盖过去的日期)。注意, 该字段也是公有的, 我们不需要把只读字段设置为私有, 因为按照定义, 它们不能在外部修改(这条规则也适用于常量)。

如前所述, 日期用基类 **System.DateTime** 表示。下面的代码使用带有 3 个参数(年份、月份和月份中的日)的 **System.DateTime** 构造函数, 可以从 MSDN 文档中找到这个构造函数和其他 **DateTime**

构造函数的更多信息。

```
public class Document
{
    public readonly DateTime CreationDate;

    public Document()
    {
        // Read in creation date from file. Assume result is 1 Jan 2002
        // but in general this can be different for different instances
        // of the class
        CreationDate = new DateTime(2002, 1, 1);
    }
}
```

在上面的代码段中，`CreationDate` 和 `MaxDocuments` 的处理方式与任何其他字段相同，但因为它们是只读的，所以不能在构造函数外部赋值：

```
void SomeMethod()
{
    MaxDocuments = 10; // compilation error here. MaxDocuments is readonly
}
```

还要注意，在构造函数中不必给只读字段赋值。如果没有赋值，它的值就是其特定数据类型的默认值，或者在声明时给它初始化的值。这适用于只读的静态字段和实例字段。

3.3 匿名类型

第 2 章讨论了 `var` 关键字，它用于表示隐式类型化的变量。`var` 与 `new` 关键字一起使用时，可以创建匿名类型。匿名类型只是一个继承自 `Object` 且没有名称的类。该类的定义从初始化器中推断，类似于隐式类型化的变量。

如果需要一个对象包含某个人的姓氏、中间名和名字，则声明如下：

```
var captain = new {FirstName = "James", MiddleName = "T", LastName = "Kirk"};
```

这会生成一个包含 `FirstName`、`MiddleName` 和 `LastName` 属性的对象。如果创建另一个对象，如下所示：

```
var doctor = new {FirstName = "Leonard", MiddleName = "", LastName = "McCoy"};
```

`Captain` 和 `doctor` 的类型就相同。例如，可以设置 `captain = doctor`。

如果所设置的值来自于另一个对象，就可以简化初始化器。如果已经有一个包含 `FirstName`、`MiddleName` 和 `LastName` 属性的类，且有该类的一个实例(`person`)，`captain` 对象就可以初始化为：

```
var captain = new {person.FirstName, person.MiddleName, person.LastName};
```

`person` 对象的属性名应投射到新对象名 `captain`。所以 `captain` 对象应有 `FirstName`、`MiddleName` 和 `LastName` 属性。

这些新对象的类型名未知。编译器为类型“伪造”了一个名称，但只有编译器才能使用它。我们不能也不应使用新对象上的任何类型反射，因为这不会得到一致的结果。

3.4 结构

前面介绍了类如何封装程序中的对象，也介绍了如何将它们存储在堆中，通过这种方式可以在数据的生存期上获得很大的灵活性，但性能会有一些的损失。因为托管堆的优化，这种性能损失比较小。但是，有时仅需要一个小的数据结构。此时，类提供的功能多于我们需要的功能，由于性能原因，最好使用结构。看看下面的例子：

```
class Dimensions
{
    public double Length;
    public double Width;
}
```

上面的代码定义了类 `Dimensions`，它只存储了某一项的长度和宽度。假定编写一个布置家具的程序，让人们试着在计算机上重新布置家具，并存储每件家具的维度。使字段变为公共字段，就会违背编程规则，但我们实际上并不需要类的全部功能。现在只有两个数字，把它们当作一对来处理，要比单个处理方便一些。既不需要很多方法，也不需要从类中继承，也不希望.NET 运行库在堆中遇到麻烦和性能问题，只需存储两个 `double` 类型的数据即可。

为此，只需修改代码，用关键字 `struct` 代替 `class`，定义一个结构而不是类，如本章前面所述：

```
struct Dimensions
{
    public double Length;
    public double Width;
}
```

为结构定义函数与为类定义函数完全相同。下面的代码说明了结构的构造函数和属性：

```
struct Dimensions
{
    public double Length;
    public double Width;

    public Dimensions(double length, double width)
    {
        Length=length;
        Width=width;
    }

    public double Diagonal
    {
        get
        {
            return Math.Sqrt(Length*Length + Width*Width);
        }
    }
}
```

```

    }
}

```

结构是值类型，不是引用类型。它们存储在栈中或存储为内联(**inline**)(如果它们是存储在堆中的另一个对象的一部分)，其生存期的限制与简单的数据类型一样。

- 结构不支持继承。
- 对于结构构造函数的工作方式有一些区别。尤其是编译器总是提供一个无参数的默认构造函数，它是不允许替换的。
- 使用结构，可以指定字段如何在内存中的布局(第 14 章在介绍属性时将详细论述这个问题)。

因为结构实际上是把数据项组合在一起，有时大多数或者全部字段都声明为 **public**。严格来说，这与编写 .NET 代码的规则相反——根据 Microsoft，字段(除了 **const** 字段之外)应总是私有的，并由公有属性封装。但是，对于简单的结构，许多开发人员都认为公有字段是可接受的编程方式。

下面几节将详细说明类和结构之间的区别。

3.4.1 结构是值类型

虽然结构是值类型，但在语法上常常可以把它们当作类来处理。例如，在上面的 **Dimensions** 类的定义中，可以编写下面的代码：

```

Dimensions point = new Dimensions();
point.Length = 3;
point.Width = 6;

```

注意，因为结构是值类型，所以 **new** 运算符与类和其他引用类型的工作方式不同。**new** 运算符并不分配堆中的内存，而是只调用相应的构造函数，根据传送给它的参数，初始化所有的字段。对于结构，可以编写下述完全合法的代码：

```

Dimensions point;
point.Length = 3;
point.Width = 6;

```

如果 **Dimensions** 是一个类，就会产生一个编译错误，因为 **point** 包含一个未初始化的引用——不指向任何地方的一个地址，所以不能给其字段设置值。但对于结构，变量声明实际上是为整个结构在栈中分配空间，所以就可以为它赋值了。但要注意下面的代码会产生一个编译错误，编译器会抱怨用户使用了未初始化的变量：

```

Dimensions point;
Double D = point.Length;

```

结构遵循其他数据类型都遵循的规则：在使用前所有的元素都必须进行初始化。在结构上调用 **new** 运算符，或者给所有的字段分别赋值，结构就完全初始化了。当然，如果结构定义为类的成员字段，在初始化包含的对象时，该结构会自动初始化为 0。

结构是会影响性能的值类型，但根据使用结构的方式，这种影响可能是正面的，也可能是负面的。正面的影响是为结构分配内存时，速度非常快，因为它们将内联或者保存在栈中。在结构超出了作用域被删除时，速度也很快。负面影响是，只要把结构作为参数来传递或者把一个结构赋予另

一个结构(如 $A=B$, 其中 A 和 B 是结构), 结构的所有内容就被复制, 而对于类, 则只复制引用。这样就会有性能损失, 根据结构的大小, 性能损失也不同。注意, 结构主要用于小的数据结构。但当把结构作为参数传递给方法时, 应把它作为 `ref` 参数传递, 以避免性能损失——此时只传递了结构在内存中的地址, 这样传递速度就与在类中的传递速度一样快了。但如果这样做, 就必须注意被调用的方法可以改变结构的值。

3.4.2 结构和继承

结构不是为继承设计的。这意味着: 它不能从一个结构中继承。唯一的例外是对应的结构(和 C# 中的其他类型一样)最终派生于类 `System.Object`。因此, 结构也可以访问 `System.Object` 的方法。在结构中, 甚至可以重写 `System.Object` 中的方法——如重写 `ToString()` 方法。结构的继承链是: 每个结构派生自 `System.ValueType` 类, `System.ValueType` 类又派生自 `System.Object`。`ValueType` 并没有给 `Object` 添加任何新成员, 但提供了一些更适合结构的实现方式。注意, 不能为结构提供其他基类: 每个结构都派生自 `ValueType`。

3.4.3 结构的构造函数

为结构定义构造函数的方式与为类定义构造函数的方式相同, 但不允许定义无参数的构造函数。这看起来似乎没有意义, 其原因隐藏在 .NET 运行库的实现方式中。下述情况非常少见: .NET 运行库不能调用用户提供的自定义无参数构造函数, 因此 Microsoft 采用一种非常简单的方式, 禁止在 C# 的结构内使用无参数的构造函数。

前面说过, 默认构造函数把数值字段都初始化为 0, 把引用类型字段初始化为 `null`, 且总是隐式地给出, 即使提供了其他带参数的构造函数, 也是如此。提供字段的初始值也不能绕过默认构造函数。下面的代码会产生编译错误:

```
struct Dimensions
{
    public double Length = 1; // error. Initial values not allowed
    public double Width = 2; // error. Initial values not allowed
}
```

当然, 如果 `Dimensions` 声明为一个类, 这段代码就不会有编译错误。

另外, 可以像类那样为结构提供 `Close()` 或 `Dispose()` 方法。

3.5 部分类

`partial` 关键字允许把类、结构或接口放在多个文件中。一般情况下, 一个类全部驻留在单个文件中。但有时, 多个开发人员需要访问同一个类, 或者某种类型的代码生成器生成了一个类的某部分, 所以把类放在多个文件中是有益的。

`partial` 关键字的用法是: 把 `partial` 放在 `class`、`struct` 或 `interface` 关键字的前面。在下面的例子中, `TheBigClass` 类驻留在两个不同的源文件 `BigClassPart1.cs` 和 `BigClassPart2.cs` 中:

```
//BigClassPart1.cs
partial class TheBigClass
{
    public void MethodOne()
    {
    }
}

//BigClassPart2.cs
partial class TheBigClass
{
    public void MethodTwo()
    {
    }
}
```

编译包含这两个源文件的项目时，会创建一个 `TheBigClass` 类，它有两个方法 `MethodOne()` 和 `MethodTwo()`。

如果声明类时使用了下面的关键字，这些关键字就必须应用于同一个类的所有部分：

- `public`
- `private`
- `protected`
- `internal`
- `abstract`
- `sealed`
- `new`
- 一般约束

在嵌套的类型中，只要 `partial` 关键字位于 `class` 关键字的前面，就可以嵌套部分类。在把部分类编译到类型中时，属性、XML 注释、接口、泛型类型的参数属性和成员会合并。有如下两个源文件：

```
//BigClassPart1.cs
[CustomAttribute]
partial class TheBigClass: TheBigBaseClass, IBigClass
{
    public void MethodOne()
    {
    }
}

//BigClassPart2.cs
[AnotherAttribute]
partial class TheBigClass: IOtherBigClass
{
    public void MethodTwo()
    {
    }
}
```

```

    }

```

编译后, 等价的源文件变成:

```

[CustomAttribute]
[AnotherAttribute]
partial class TheBigClass: TheBigBaseClass, IBigClass, IOtherBigClass
{
    public void MethodOne()
    {
    }

    public void MethodTwo()
    {
    }
}

```

3.6 静态类

本章前面讨论了静态构造函数和它们可以如何初始化静态的成员变量。如果类只包含静态的方法和属性, 该类就是静态的。静态类在功能上与使用私有静态构造函数创建的类相同。不能创建静态类的实例。使用 `static` 关键字, 编译器可以检查用户是否不经意间给该类添加了实例成员。如果是, 就生成一个编译错误。这可以确保不创建静态类的实例。静态类的语法如下所示:

```

static class StaticUtilities
{
    public static void HelperMethod()
    {
    }
}

```

调用 `HelperMethod()` 不需要 `StaticUtilities` 类型的对象。使用类型名即可进行该调用:

```

StaticUtilities.HelperMethod();

```

3.7 Object 类

前面提到, 所有的.NET 类都派生自 `System.Object`。实际上, 如果在定义类时没有指定基类, 编译器就会自动假定这个类派生自 `Object`。本章没有使用继承, 所以前面介绍的每个类都派生自 `System.Object` (如前所述, 对于结构, 这个派生是间接的: 结构总是派生自 `System.ValueType`, `System.ValueType` 又派生自 `System.Object`)。

其实际意义在于, 除了自己定义的方法和属性等外, 还可以访问为 `Object` 定义的许多公有的和受保护的成员方法。这些方法可用于自己定义的所有其他类中。

3.7.1 System.Object()方法

下面将简要总结每个方法的作用，下一节详细论述 ToString()方法。

- ToString()方法：是获取对象的字符串表示的一种便捷方式。当只需要快速获取对象的内容，以进行调试时，就可以使用这个方法。在数据的格式化方面，它几乎没有提供选择：例如，在原则上日期可以表示为许多不同的格式，但 DateTime.ToString()没有在这方面提供任何选择。如果需要更复杂的字符串表示，例如，考虑用户的格式化首选项或文化(区域)，就应实现 IFormattable 接口(详见第 9 章)。
- GetHashCode()方法：如果对象放在名为映射(也称为散列表或字典)的数据结构中，就可以使用这个方法。处理这些结构的类使用该方法确定把对象放在结构的什么地方。如果希望把类用作字典的一个键，就需要重写 GetHashCode()方法。实现该方法重载的方式有一些相当严格的限制，这些将在第 10 章介绍字典时讨论。
- Equals()(两个版本)和 ReferenceEquals()方法：如果把 3 个用于比较对象相等性的不同方法组合起来，就说明 .NET Framework 在比较相等性方面有相当复杂的模式。这 3 个方法和比较运算符“=”在使用方式上有微妙的区别。而且，在重写带一个参数的虚 Equals()方法时也有一些限制，因为 System.Collections 名称空间中的一些基类要调用该方法，并希望它以特定的方式执行。第 7 章在介绍运算符时将探讨这些方法的使用。
- Finalize()方法：第 13 章将介绍这个方法，它最接近 C++风格的析构函数，在引用对象作为垃圾被回收以清理资源时调用它。Finalize()方法的 Object 实现方式实际上什么也没有做，因而被垃圾收集器忽略。如果对象拥有对未托管资源的引用，则在该对象被删除时，就需要删除这些引用，此时一般要重写 Finalize()。垃圾收集器不能直接删除这些对未托管资源的引用，因为它只负责托管的资源，于是它只能依赖用户提供的 Finalize()。
- GetType()方法：这个方法返回从 System.Type 派生的类的一个实例。这个对象可以提供对象成员所属类的更多信息，包括基本类型、方法、属性等。System.Type 还提供了 .NET 的反射技术的入口点。这个主题详见第 14 章。
- MemberwiseClone()方法：这是 System.Object 中唯一没有在本书的其他地方详细论述的方法。不需要讨论这个方法，因为它在概念上相当简单，它只复制对象，并返回对副本的一个引用(对于值类型，就是一个装箱的引用)。注意，得到的副本是一个浅表复制，即它复制了类中的所有值类型。如果类包含内嵌的引用，就只复制引用，而不复制引用的对象。这个方法受保护的，所以不能用于复制外部的对象。该方法不是虚方法，所以不能重写它的实现代码。

3.7.2 ToString()方法

第 2 章已经提到了 ToString()方法，它是快速获取对象的字符串表示的最便捷的方式。

例如：

```
int i = 50;  
string str = i.ToString(); // returns "-50"
```

下面是另一个例子：

```
enum Colors {Red, Orange, Yellow};
// later on in code...
Colors favoriteColor = Colors.Orange;
string str = favoriteColor.ToString(); // returns "Orange"
```

`Object.ToString()`声明为虚方法, 在这些例子中, 实现该方法的代码都是为 C#预定义数据类型重写的代码, 以返回这些类型的正确字符串表示。`Colors` 枚举是一个预定义的数据类型, 它实际上实现为一个派生自 `System.Enum` 的结构, 而 `System.Enum` 有一个相当智能的 `ToString()`重写方法, 它处理用户定义的所有枚举。

如果不在自己定义的类中重写 `ToString()`, 该类将只继承 `System.Object` 的实现方式——它显示类的名称。如果希望 `ToString()`返回一个字符串, 其中包含类中对象的值信息, 就需要重写它。下面用一个例子 `Money` 来说明这一点。在该例子中, 定义一个非常简单的类 `Money`, 它表示美元数。`Money` 只是 `decimal` 类的包装器, 但它提供了一个 `ToString()`方法。注意, 这个方法必须声明为 `override`, 因为它将替代(重写)`Object`提供的 `ToString()`方法。第4章将详细讨论重写。该例子的完整代码如下所示(注意它还说明了如何使用属性封装字段):

```
using System;

namespace Wrox
{
    class MainEntryPoint
    {
        static void Main(string[] args)
        {
            Money cash1 = new Money();
            cash1.Amount = 40M;
            Console.WriteLine("cash1.ToString() returns: " + cash1.ToString());
            Console.ReadLine();
        }
    }

    public class Money
    {
        private decimal amount;

        public decimal Amount
        {
            get
            {
                return amount;
            }
            set
            {
                amount = value;
            }
        }

        public override string ToString()
        {
            return "$" + Amount.ToString();
        }
    }
}
```

```

    }
}

```

这个例子仅说明了 C# 的语法特性。C# 已经有表示货币量的预定义类型 `decimal`。所以在现实生活中，不必编写这样的类来重复该功能，除非要给它添加其他各种方法。在许多情况下，由于格式化要求，也可以使用 `String.Format()` 方法(详见第 8 章)来表示货币字符串，而不是 `ToString()`。

在 `Main()` 方法中，先实例化一个 `Money` 对象，再调用 `ToString()`，执行该方法的重写版本。运行这段代码，会得到如下结果：

```
cash1.ToString() returns: $40
```

3.8 扩展方法

有许多扩展类的方式。如果有类的源代码，继承(如第 4 章所述)就是给对象添加功能的好方法。但如果没有源代码，该怎么办？此时可以使用扩展方法，它允许改变一个类，但不需要该类的源代码。

扩展方法是静态方法，它是类的一部分，但实际上没有放在类的源代码中。假定上例中的 `Money` 类需要一个方法 `AddToAmount(decimal amountToAdd)`。但是，由于某种原因，程序集最初的源代码不能直接修改。此时必须做的所有工作就是创建一个静态类，把方法 `AddToAmount()` 添加为一个静态方法。对应的代码如下：

```

namespace Wrox
{
    public static class MoneyExtension
    {
        public static void AddToAmount(this Money money, decimal amountToAdd)
        {
            money.Amount += amountToAdd;
        }
    }
}

```

注意 `AddToAmount()` 方法的参数。对于扩展方法，第一个参数是要扩展的类型，它放在 `this` 关键字的后面。这告诉编译器，这个方法是 `Money` 类型的一部分。在这个例子中，`Money` 是要扩展的类型。在扩展方法中，可以访问所扩展类型的所有公有方法和属性。

在主程序中，`AddToAmount()` 方法看起来像是另一个方法。它没有显示第一个参数，也不能对它进行任何处理。要使用新方法，需要执行如下调用，这与其他方法相同：

```
cash1.AddToAmount(10M);
```

即使扩展方法是静态的，也要使用标准的实例方法语法。注意这里使用 `cash1` 实例变量来调用 `AddToAmount()`，而没有使用类型名。

如果扩展方法与类中的某个方法同名，就从来不会调用扩展方法。类中已有的任何实例方法优先。

3.9 小结

本章介绍了 C# 中声明和处理对象的语法，论述了如何声明静态和实例字段、属性、方法和构造函数。还讨论了 C# 中新增的且其他语言的 OOP 模型中没有的新特性：例如，静态构造函数提供了初始化静态字段的方式，利用结构可以定义高性能的类型，不需要使用托管的堆。我们还阐述了 C# 中的所有类型最终都派生自类 `System.Object`，这说明所有的类型都开始于一组基本的实用方法，包括 `ToString()`。

本章多次提到了继承，第 4 章将介绍 C# 中的实现(implementation)继承和接口继承。

第 4 章

继 承

本章内容:

- 继承的类型
- 实现继承
- 访问修饰符
- 接口

第 3 章介绍了如何使用 C# 中的各个类,其重点是如何定义方法、属性、构造函数和单个类(或单个结构)中的其他成员。我们指出,所有的类最终都派生于 `System.Object` 类,但并没有说明如何创建继承类的层次结构。继承是本章的主题。我们将讨论 C# 和 .NET Framework 如何处理继承。

4.1 继承的类型

首先介绍 C# 在继承方面支持和不支持的功能。

4.1.1 实现继承和接口继承

在面向对象的编程中,有两种截然不同的继承类型:实现继承和接口继承。

- **实现继承:**表示一个类型派生于一个基类型,它拥有该基类型的所有成员字段和函数。在实现继承中,派生类型采用基类型的每个函数的实现代码,除非在派生类型的定义中指定重写某个函数的实现代码。在需要给现有的类型添加功能,或许多相关的类型共享一组重要的公共功能时,这种类型的继承非常有用。
- **接口继承:**表示一个类型只继承了函数的签名,没有继承任何实现代码。在需要指定该类型具有某些可用的特性时,最好使用这种类型的继承。

C# 支持实现继承和接口继承。它们都内置于语言和架构中,因此可以根据应用程序的体系结构选择合适的继承。

4.1.2 多重继承

一些语言(如 C++) 支持所谓的“多重继承”,即一个类派生自多个类。使用多重继承的优点是有争议的:一方面,毫无疑问,可以使用多重继承编写非常复杂、但很紧凑的代码,如 C++ ATL 库。

另一方面,使用多重实现继承的代码常常很难理解和调试(这也可以从 C++ ATL 库中看出)。如前所述,简化健壮代码的编写工作是开发 C# 的重要设计目标。因此, C# 不支持多重实现继承。而 C# 又允许类型派生自多个接口——多重接口继承。这说明, C# 类可以派生自另一个类和任意多个接口。更准确地说,因为 `System.Object` 是一个公共的基类,所以每个 C# 类(除了 `Object` 类之外)都有一个基类,还可以有任意多个基接口。

4.1.3 结构和类

第 3 章区分了结构(值类型)和类(引用类型)。使用结构的一个限制是结构不支持继承,但每个结构都自动派生自 `System.ValueType`。实际上还应更仔细一些:不能编码实现类型层次的结构,但结构可以实现接口。换言之,结构并不支持实现继承,但支持接口继承。事实上,定义结构和类可以总结为:

- 结构总是派生自 `System.ValueType`,它们还可以派生自任意多个接口。
- 类总是派生自用户选择的另一个类,它们还可以派生自任意多个接口。

4.2 实现继承

如果要声明派生自另一个类的一个类,就可以使用下面的语法:

```
class MyDerivedClass: MyBaseClass
{
    // functions and data members here
}
```

这个语法非常类似于 C++ 和 Java 中的语法,但是, C++ 程序员习惯于使用公共和私有继承的概念,要注意 C# 不支持私有继承,因此在基类名上没有 `public` 或 `private` 限定符。支持私有继承只会大大增加语言的复杂性,实际上私有继承在 C++ 中也很少使用。

如果类(或结构)也派生自接口,则用逗号分隔列表中的基类和接口:

```
public class MyDerivedClass: MyBaseClass, IInterface1, IInterface2
{
    // etc.
}
```

对于结构,语法如下:

```
public struct MyDerivedStruct: IInterface1, IInterface2
{
    // etc.
}
```

如果在类定义中没有指定基类，C#编译器就假定 `System.Object` 是基类。因此下面的两段代码生成相同的结果：

```
class MyClass: Object // derives from System.Object
{
    // etc.
}
```

和

```
class MyClass // derives from System.Object
{
    // etc.
}
```

第二种形式比较常用，因为它较简单。

C#支持 `object` 关键字，它用作 `System.Object` 类的假名，所以也可以编写下面的代码：

```
class MyClass: object // derives from System.Object
{
    // etc.
}
```

如果要引用 `Object` 类，就可以使用 `object` 关键字，智能编辑器(如 Visual Studio)会识别它，因此便于编辑代码。

4.2.1 虚方法

把一个基类函数声明为 `virtual`，就可以在任何派生类中重写该函数：

```
class MyBaseClass
{
    public virtual string VirtualMethod()
    {
        return "This method is virtual and defined in MyBaseClass";
    }
}
```

也可以把属性声明为 `virtual`。对于虚属性或重写属性，语法与非虚属性相同，但要在定义中添加关键字 `virtual`，其语法如下所示：

```
public virtual string ForeName
{
    get { return foreName;}
    set { foreName = value;}
}
private string foreName;
```

为了简单起见，下面的讨论将主要集中于方法，但其规则也适用于属性。

C#中虚函数的概念与标准 OOP 的概念相同：可以在派生类中重写虚函数。在调用方法时，会

调用该类对象的合适方法。在 C# 中，函数在默认情况下不是虚拟的，但(除了构造函数以外)可以显式地声明为 `virtual`。这遵循 C++ 的方式，即从性能的角度来看，除非显式指定，否则函数就不是虚拟的。而在 Java 中，所有的函数都是虚拟的。但 C# 的语法与 C++ 的语法不同，因为 C# 要求在派生类的函数重写另一个函数时，要使用 `override` 关键字显式声明：

```
class MyDerivedClass: MyBaseClass
{
    public override string VirtualMethod()
    {
        return " This method is an override defined in MyDerivedClass. " ;
    }
}
```

重写方法的语法避免了 C++ 中很容易发生的潜在运行错误：当派生类的方法签名无意中与基类版本略有差别时，该方法就不能重写基类的方法。在 C# 中，这会出现一个编译错误，因为编译器会认为函数已标记为 `override`，但没有重写其基类的方法。

成员字段和静态函数都不能声明为 `virtual`，因为这个概念只对类中的实例函数成员有意义。

4.2.2 隐藏方法

如果签名相同的方法在基类和派生类中都进行了声明，但该方法没有分别声明为 `virtual` 和 `override`，派生类方法就会隐藏基类方法。

在大多数情况下，是要重写方法，而不是隐藏方法，因为隐藏方法会造成对于给定类的实例调用错误方法的危险。但是，如下面的例子所示，C# 语法可以确保开发人员在编译时收到这个潜在错误的警告，从而使隐藏方法(如果这确实是用户的本意)更加安全。这也是类库开发人员得到的版本方面的好处。

假定有一个类 `HisBaseClass`：

```
class HisBaseClass
{
    // various members
}
```

在将来的某一刻，要编写一个派生类，用它给 `HisBaseClass` 添加某个功能，特别是要添加该基类中目前没有的方法——`MyGroovyMethod()`：

```
class MyDerivedClass: HisBaseClass
{
    public int MyGroovyMethod()
    {
        // some groovy implementation
        return 0;
    }
}
```

一年后，基类的编写者决定扩展基类的功能。为了保持一致，他也添加了一个名为

`MyGroovyMethod()`的方法, 该方法的名称和签名与前面添加的方法相同, 但并不完成相同的工作。在使用基类的新方法编译代码时, 程序在应该调用哪个方法上就会有潜在的冲突。这在 C# 中完全合法, 但因为 `MyGroovyMethod()` 与基类的 `MyGroovyMethod()` 不相关, 运行这段代码就可能会产生意外的结果。C# 可以很好地处理这种冲突。

此时, 编译时系统会发出警告。在 C# 中, 要隐藏一个方法应使用 `new` 关键字声明, 如下所示:

```
class MyDerivedClass: HisBaseClass,
{
    public new int MyGroovyMethod()
    {
        // some groovy implementation
        return 0;
    }
}
```

但是, 新添加的 `MyGroovyMethod()` 没有声明为 `new`, 所以编译器会认为它隐藏了基类的方法, 但没有显式声明, 因此系统会发出一个警告(这也适用于是否把 `MyGroovyMethod()` 声明为 `virtual`)。如果愿意, 就可以给新方法重命名。最好这么做, 因为这会避免许多冲突。但是, 如果觉得重命名方法不可能(例如, 已经针对其他公司把软件发布为一个库, 所以无法修改方法的名称), 则所有的已有客户端代码仍能正确运行, 方法是选择新添加的 `MyGroovyMethod()`。这是因为访问这个方法的任何已有代码必须通过对 `MyDerivedClass`(或进一步派生的类)的引用进行选择。

已有的代码不能通过对 `HisBaseClass` 类的引用访问这个方法, 因为在对 `HisBaseClass` 类的早期版本进行编译时, 会产生一个编译错误。这个问题只会发生在将来编写的客户端代码上。C# 会发出一个警告, 告诉用户在将来的代码中可能会出问题——用户应注意这个警告, 不要试图在将来添加的代码中通过对 `HisBaseClass` 的引用调用新的 `MyGroovyMethod()` 方法, 但所有已有的代码仍会正常工作。这是比较微妙的, 但它很好地说明了 C# 如何处理类的不同版本。

4.2.3 调用函数的基类版本

C# 有一种特殊的语法用于从派生类中调用方法的基类版本: `base.<MethodName>()`。例如, 假定派生类中的一个方法要返回基类的方法 90% 的返回值, 就可以使用下面的语法:

```
class CustomerAccount
{
    public virtual decimal CalculatePrice()
    {
        // implementation
        return 0.0M;
    }
}

class GoldAccount: CustomerAccount
{
    public override decimal CalculatePrice()
    {
        return base.CalculatePrice() * 0.9M;
    }
}
```

```
}
```

注意, 可以使用 `base.<MethodName>()` 语法调用基类中的任何方法, 不必从同一个方法的重载中调用它。

4.2.4 抽象类和抽象函数

C# 允许把类和函数声明为 **abstract**。抽象类不能实例化, 而抽象函数不能直接实现, 必须在非抽象的派生类中重写。显然, 抽象函数本身也是虚拟的(尽管也不需要提供 **virtual** 关键字, 实际上, 如果提供了该关键字, 就会产生一个语法错误)。如果类包含抽象函数, 则该类也是抽象的, 也必须声明为抽象的:

```
abstract class Building
{
    public abstract decimal CalculateHeatingCost(); // abstract method
}
```



C++ 开发人员还要注意术语上的细微差别: 在 C++ 中, 抽象函数常常描述为纯虚函数, 而在 C# 中, 仅使用抽象这个术语。

4.2.5 密封类和密封方法

C# 允许把类和方法声明为 **sealed**。对于类, 这表示不能继承该类; 对于方法, 这表示不能重写该方法。

```
sealed class FinalClass
{
    // etc
}
class DerivedClass: FinalClass // wrong. Will give compilation error
{
    // etc
}
```

在把类或方法标记为 **sealed** 时, 最可能的情形是: 如果要对库、类或自己编写的其他类作用域之外的类或方法进行操作, 则重写某些功能会导致代码混乱。也可以因商业原因把类或方法标记为 **sealed**, 以防第三方以违反授权协议的方式扩展该类。但一般情况下, 在把类或成员标记为 **sealed** 时要小心, 因为这么做会严重限制它的使用方式。即使认为它不能对继承自一个类或重写类的某个成员发挥作用, 仍有可能在将来的某个时刻, 有人会遇到我们没有预料到的情形, 此时这么做就很有用。 .NET 基类库大量使用了密封类, 使希望从这些类中派生出自己的类的第三方开发人员无法访问这些类。例如, **string** 就是一个密封类。

把方法声明为 **sealed** 也可以实现类似的目的, 但很少这么做。

```

class MyClass: MyClassBase
{
    public sealed override void FinalMethod()
    {
        // etc.
    }
}

class DerivedClass: MyClass
{
    public override void FinalMethod() // wrong. Will give compilation error
    {
    }
}

```

要在方法或属性上使用 `sealed` 关键字, 必须先从基类上把它声明为要重写的方法或属性。如果基类上不希望有重写的方法或属性, 就不要把它声明为 `virtual`。

4.2.6 派生类的构造函数

第3章介绍了单个类的构造函数是如何工作的。这样, 就产生了一个有趣的问题, 在开始为层次结构中的类(这个类继承了其他也可能有自定义构造函数的类)定义自己的构造函数时, 会发生什么情况?

假定没有为任何类定义任何显式的构造函数, 这样编译器就会为所有的类提供默认的初始化构造函数, 在后台会进行许多操作, 但编译器可以很好地解决类的层次结构中的所有问题, 每个类中的每个字段都会初始化为对应的默认值。但在添加了一个我们自己的构造函数后, 就要通过派生类的层次结构高效地控制构造过程, 因此必须确保构造过程顺利进行, 不要出现不能按照层次结构进行构造的问题。

为什么派生类会有某些特殊的问题? 原因是在创建派生类的实例时, 实际上会有多个构造函数起作用。要实例化的类的构造函数本身不能初始化类, 还必须调用基类中的构造函数。这就是为什么要通过层次结构进行构造的原因。

为了说明为什么必须调用基类的构造函数, 下面是手机公司 MortimerPhones 开发的一个例子。这个例子包含一个抽象类 `GenericCustomer`, 它表示顾客。还有一个(非抽象)类 `Nevermore60Customer`, 它表示采用特定付费方式(称为 `Nevermore60` 付费方式)的顾客。所有的顾客都有一个名字, 它由一个私有字段表示。在 `Nevermore60` 付费方式中, 顾客前几分钟的电话费比较高, 需要一个字段 `highCostMinutesUsed`, 它详细说明了每个顾客该如何支付这些较高的电话费。抽象类 `GenericCustomer` 的定义如下所示:



可从
wrox.com
下载源代码

```

abstract class GenericCustomer
{
    private string name;
    // lots of other methods etc.
}

class Nevermore60Customer: GenericCustomer
{
    private uint highCostMinutesUsed;
}

```

```
// other methods etc.  
}
```

代码段 MortimerPhones.cs

不要担心在这些类中实现的其他方法，因为这里仅考虑构造过程。如果下载了本章的示例代码，就会发现类的定义仅包含构造函数。

下面看看使用 `new` 运算符实例化 `Nevermore60Customer` 时，会发生什么情况：

```
GenericCustomer customer = new Nevermore60Customer();
```

显然，成员字段 `name` 和 `highCostMinutesUsed` 都必须在实例化 `customer` 时进行初始化。如果没有提供自己的构造函数，而是仅依赖默认的构造函数，那么 `name` 会初始化为 `null` 引用，`highCostMinutesUsed` 初始化为 0。下面详细讨论其过程。

`highCostMinutesUsed` 字段没有问题：编译器提供的默认 `Nevermore60Customer` 构造函数会把它初始化为 0。

那么 `name` 呢？看看类定义，显然，`Nevermore60Customer` 构造函数不能初始化这个值。字段 `name` 声明为 `private`，这意味着派生的类不能访问它。默认的 `Nevermore60Customer` 构造函数甚至不知道存在这个字段。唯一知道这个字段的代码项是 `GenericCustomer` 的其他成员，这意味着如果对 `name` 进行初始化，就必须在 `GenericCustomer` 的某个构造函数中进行。无论类层次结构有多大，这种情况都会一直延续到最终的基类 `System.Object` 上。

理解了上面的问题后，就可以明白实例化派生类时会发生什么样的情况了。假定默认的构造函数一直在使用：编译器首先找到它试图实例化的类的构造函数，在本例中是 `Nevermore60Customer`，这个默认 `Nevermore60Customer` 构造函数首先要做的是为其直接基类 `GenericCustomer` 运行默认构造函数，然后 `GenericCustomer` 构造函数为其直接基类 `System.Object` 运行默认构造函数，`System.Object` 没有任何基类，所以它的构造函数就执行，并把控制权返回给 `GenericCustomer` 构造函数。现在执行 `GenericCustomer` 构造函数，把 `name` 初始化为 `null`，再把控制权返回给 `Nevermore60Customer` 构造函数，接着执行这个构造函数，把 `highCostMinutesUsed` 初始化为 0，并退出。此时，`Nevermore60Customer` 实例就已经成功地构造和初始化了。

所有操作的最终结果是，构造函数的调用顺序是先调用 `System.Object`，再按照层次结构由上向下进行，直到到达编译器要实例化的类为止。还要注意在这个过程中，每个构造函数都初始化它自己的类中的字段。这是它的一般工作方式，在开始添加自己的构造函数时，也应尽可能遵循这条规则。

注意构造函数的执行顺序。总是最先调用的正是基类的构造函数。也就是说，派生类的构造函数可以在执行过程中调用它可以访问的任何基类方法、属性和任何其他成员，因为基类已经构造出来了，其字段也初始化了。这也意味着，如果派生类不喜欢初始化基类的方式，但要访问数据，就可以改变数据的初始值。但是，好的编程方式几乎总是应尽可能避免这种情况，让基类构造函数来处理其字段。

理解了构造过程后，就可以开始添加自己的构造函数了。

1. 在层次结构中添加无参数的构造函数

首先讨论最简单的情况，在层次结构中用一个无参数的构造函数来替换默认的构造函数后，看

看会发生什么情况。假定要把每个人的名字初始化为字符串"<no name>", 而不是 null 引用。就可以修改 `GenericCustomer` 中的代码, 如下所示:

```
public abstract class GenericCustomer
{
    private string name;
    public GenericCustomer()
    : base() // We could omit this line without affecting the compiled code.
    {
        name = "<no name>";
    }
}
```

添加这段代码后, 代码运行正常。`Nevermore60Customer` 仍有自己的默认构造函数, 所以上面描述的事件的顺序保持不变, 但编译器会使用自定义 `GenericCustomer` 构造函数, 而不是生成默认的构造函数, 所以 `name` 字段按照需要总是初始化为"<no name>"。

注意, 在定制的构造函数中, 在执行 `GenericCustomer` 构造函数前, 添加了一个对基类构造函数的调用, 使用的语法与前面解释如何让构造函数的不同重载版本互相调用时使用的语法相同。唯一的区别是, 这次使用的关键字是 `base`, 而不是 `this`, 表示这是基类的构造函数, 而不是要调用的当前类的构造函数。在 `base` 关键字后面的圆括号中没有参数, 这非常重要, 因为没有给基类构造函数传送任何参数, 所以编译器必须调用无参数的构造函数。其结果是编译器会插入要调用 `System.Object` 构造函数的代码, 这正好与默认情况相同。

实际上, 可以省略这行代码, 只加上为本章中大多数构造函数编写的代码:

```
public GenericCustomer()
{
    name = " < no name > ";
}
```

如果编译器没有在左花括号的前面找到对另一个构造函数的任何引用, 它就会假定我们要调用基类的构造函数——这符合默认构造函数的工作方式。

`base` 和 `this` 关键字是调用另一个构造函数时允许使用的唯一关键字, 其他关键字都会产生编译错误。还要注意只能指定唯一一个其他的构造函数。

到目前为止, 这段代码运行正常。但是, 要通过构造函数的层次结构把进度弄乱的最好方法是把构造函数声明为私有:

```
private GenericCustomer()
{
    name = "<no name>";
}
```

如果试图这样做, 就会产生一个有趣的编译错误, 如果不理解构造是如何按照层次结构由上而下的顺序工作的, 这个错误就会让人摸不着头脑。

```
'Wrox.ProCSharp.GenericCustomer.GenericCustomer()' is inaccessible due to its protection level
```

有趣的是, 该错误没有发生在 `GenericCustomer` 类中, 而是发生在 `Nevermore60Customer` 派生类

中。编译器试图为 `Nevermore60Customer` 生成默认的构造函数，但又做不到，因为默认的构造函数应调用无参数的 `GenericCustomer` 构造函数。把该构造函数声明为 `private`，它就不可能访问派生类了。如果为 `GenericCustomer` 提供一个带参数的构造函数，但同时没有提供一个无参数的构造函数，也会发生类似的错误。在本例中，编译器不能为 `GenericCustomer` 生成默认构造函数，所以当编译器试图为任何派生类生成默认构造函数时，它会再次发现它不能做到这一点，因为没有无参数的基类构造函数可调用。这个问题的解决方法是为派生类添加自己的构造函数——实际上不需要在这些构造函数中做任何工作，这样，编译器就不会为这些派生类生成任何默认构造函数了。

前面介绍了所有的理论知识，下面用一个例子来说明如何给类的层次结构添加构造函数。下一节为 `MortimerPhones` 示例添加带参数的构造函数。

2. 在层次结构中添加带参数的构造函数

首先是带一个参数的 `GenericCustomer` 构造函数，它仅在顾客提供其姓名时才实例化顾客：

```
abstract class GenericCustomer
{
    private string name;
    public GenericCustomer(string name)
    {
        this.name = name;
    }
}
```

到目前为止，代码正常运行，但刚才说过，在编译器试图为派生类创建默认构造函数时，会产生一个编译错误，因为编译器为 `Nevermore60Customer` 生成的默认构造函数会试图调用一个无参数的 `GenericCustomer` 构造函数，但 `GenericCustomer` 没有这样的构造函数。因此，需要为派生类提供一个构造函数，来避免这个编译错误：

```
class Nevermore60Customer: GenericCustomer
{
    private uint highCostMinutesUsed;
    public Nevermore60Customer(string name)
        : base(name)
    {
    }
}
```

现在，`Nevermore60Customer` 对象的实例化只有在提供了包含顾客姓名的字符串时才能进行，这正是我们需要的。有趣的是 `Nevermore60Customer` 构造函数对这个字符串所做的处理。它本身不能初始化 `name` 字段，因为它不能访问基类中的私有字段，但可以把顾客姓名传送给基类，以便 `GenericCustomer` 构造函数处理。具体方法是，把先执行的基类构造函数指定为把顾客姓名当作参数的构造函数。除此之外，它不需要执行任何操作。

下面讨论如果要处理不同的重载构造函数和一个类的层次结构，会发生什么情况。最终，假定 `Nevermore60` 的顾客通过朋友联系到 `MortimerPhones`，即 `MortimerPhones` 公司中有一个人是朋友，因此通过与朋友签约可以获得折扣。这表示在构造一个 `Nevermore60Customer` 时，还需要传递联系人的姓名。在现实生活中，构造函数必须利用该姓名去完成更复杂的工作，如处理折扣等，但这里只是把联系人的姓名存储到另一个字段中。

此时, `Nevermore60Customer` 的定义如下所示:

```
class Nevermore60Customer: GenericCustomer
{
    public Nevermore60Customer(string name, string referrerName)
    : base(name)
    {
        this.referrerName = referrerName;
    }

    private string referrerName;
    private uint highCostMinutesUsed;
```

该构造函数将姓名作为参数, 把它传递给 `GenericCustomer` 构造函数进行处理。referrerName 是一个需要声明的变量, 这样构造函数才能在其主体中处理这个参数。

但是, 并不是所有的 `Nevermore60Customers` 都有联系人, 所以还需要有一个不需此参数的构造函数(或为它提供默认值的构造函数)。实际上, 我们指定如果没有联系人, `referrerName` 字段就设置为 "<None>", 使用如下带一个参数的构造函数:

```
public Nevermore60Customer(string name)
: this(name, "<None>" )
{
}
```

这样就正确建立了所有的构造函数。执行下面的代码行时, 检查事件链很有益:

```
GenericCustomer customer = new Nevermore60Customer(cArabel Jones");
```

编译器认为它需要带一个字符串参数的构造函数, 所以它确认的构造函数就是刚才定义的最后—一个构造函数, 如下所示。

```
public Nevermore60Customer(string Name)
: this(Name, "<None>" )
```

在实例化 `customer` 时, 就会调用这个构造函数。之后立即把控制权传递给对应的 `Nevermore60Customer` 构造函数, 该构造函数带两个参数, 分别是 "Arabel Jones" 和 "<None>". 在这个构造函数中, 把控制权依次传递给 `GenericCustomer` 构造函数, 该构造函数带有 1 个参数, 即字符串 "Arabel Jones". 然后这个构造函数把控制权传送给 `System.Object` 默认构造函数。现在才能执行这些构造函数, 首先执行 `System.Object` 构造函数。接着执行 `GenericCustomer` 构造函数, 它初始化 `name` 字段。然后带有两个参数的 `Nevermore60Customer` 构造函数得到控制权, 把联系人的姓名初始化为 "<None>". 最后, 执行 `Nevermore60Customer` 构造函数, 该构造函数带有 1 个参数——这个构造函数什么也不做。

这个过程非常简洁, 设计也很合理。每个构造函数都负责处理变量的初始化。在这个过程中, 正确地实例化了类, 以备使用。如果在为类编写自己的构造函数时遵循同样的规则, 就会发现, 即便是最复杂的类也可以顺利地初始化, 并且不会出现任何问题。

4.3 修饰符

前面已经遇到许多所谓的修饰符，即应用于类型或成员的关键字。修饰符可以指定方法的可见性，如 `public` 或 `private`；还可以指定一项的本质，如方法是 `virtual` 或 `abstract`。C#有许多访问修饰符，下面讨论完整的修饰符列表。

4.3.1 可见性修饰符

表 4-1 中的修饰符确定了是否允许其他代码访问某一项。

表 4-1

修 饰 符	应 用 于	说 明
<code>public</code>	所有类型或成员	任何代码均可以访问该项
<code>protected</code>	类型和内嵌类型的所有成员	只有派生的类型能访问该项
<code>internal</code>	所有类型或成员	只能在包含它的程序集中访问该项
<code>private</code>	类型和内嵌类型的所有成员	只能在它所属的类型中访问该项
<code>protected internal</code>	类型和内嵌类型的所有成员	只能在包含它的程序集和派生类型的任何代码中访问该项

注意，类型定义可以是内部或公有的，这取决于是否希望在类型包含的程序集外部访问它：

```
public class MyClass
{
    // etc.
```

不能把类型定义为 `protected`、`private` 和 `protected internal`，因为这些修饰符对于包含在名称空间中的类型没有意义。因此这些修饰符只能应用于成员。但是，可以用这些修饰符定义嵌套的类型(即，包含在其他类型中的类型)，因为在这种情况下，类型也具有成员的状态。于是，下面的代码是合法的：

```
public class OuterClass
{
    protected class InnerClass
    {
        // etc.
    }
    // etc.
}
```

如果有嵌套的类型，则内部的类型总是可以访问外部类型的所有成员。所以，在上面的代码中，`InnerClass` 中的代码可以访问 `OuterClass` 的所有成员，甚至可以访问 `OuterClass` 的私有成员。

4.3.2 其他修饰符

表 4-2 中的修饰符可以应用于类型的成员，而且有不同的用途。在应用于类型时，其中的几个

修饰符也是有意义的。

表 4-2

修 饰 符	应 用 于	说 明
new	函数成员	成员用相同的签名隐藏继承的成员
static	所有成员	成员不作用于类的具体实例
virtual	仅函数成员	成员可以由派生类重写
abstract	仅函数成员	虚拟成员定义了成员的签名，但没有提供实现代码
override	仅函数成员	成员重写了继承的虚拟或抽象成员
sealed	类、方法和属性	对于类，不能继承自密封类。对于属性和方法，成员重写已继承的虚拟成员，但任何派生类中的任何成员都不能重写该成员。该修饰符必须与 <code>override</code> 一起使用
extern	仅静态[DllImport]方法	成员在外部用另一种语言实现

4.4 接口

如前所述，如果一个类派生自一个接口，声明这个类就会实现某些函数。并不是所有的面向对象语言都支持接口，所以本节将详细介绍 C# 接口的实现。

下面列出 Microsoft 预定义的一个接口 `System.IDisposable` 的完整定义。`IDisposable` 包含一个方法 `Dispose()`，该方法由类实现，用于清理代码：

```
public interface IDisposable
{
    void Dispose();
}
```

上面的代码说明，声明接口在语法上与声明抽象类完全相同，但不允许提供接口中任何成员的实现方式。一般情况下，接口只能包含方法、属性、索引器和事件的声明。

不能实例化接口，它只能包含其成员的签名。接口既不能有构造函数(如何构建不能实例化的对象?)也不能有字段(因为这隐含了某些内部的实现方式)。接口定义也不允许包含运算符重载，尽管这不是因为声明它们在原则上有什么问题，而是因为接口通常是公共协定，包含运算符重载会引起一些与其他 .NET 语言不兼容的问题，如 Visual Basic .NET，因为它不支持运算符重载。

在接口定义中还不允许声明关于成员的修饰符。接口成员总是公有的，不能声明为虚拟或静态。如果需要，就应由实现的类来声明，因此最好实现执行的类来声明访问修饰符，就像本节的代码那样。

例如，`IDisposable`。如果类希望声明为公有类型，以便它实现方法 `Dispose()`，该类就必须实现 `IDisposable`。在 C# 中，这表示该类派生自 `IDisposable` 类。

```
class SomeClass: IDisposable
{
    // This class MUST contain an implementation of the
    // IDisposable.Dispose() method, otherwise
```

```
// you get a compilation error.
public void Dispose()
{
    // implementation of Dispose() method
}
// rest of class
}
```

在这个例子中，如果 `SomeClass` 派生自 `IDisposable` 类，但不包含与 `IDisposable` 类中签名相同的 `Dispose()` 实现代码，就会得到一个编译错误，因为该类破坏了实现 `IDisposable` 的一致协定。当然，编译器允许类有一个不派生自 `IDisposable` 类的 `Dispose()` 方法。问题是其他代码无法识别出 `SomeClass` 类支持 `IDisposable` 特性。



`IDisposable` 是一个相当简单的接口，它只定义了一个方法。大多数接口都包含许多成员。

4.4.1 定义和实现接口

下面开发一个遵循接口继承规范的小例子来说明如何定义和使用接口。这个例子建立在银行账户的基础上。假定编写代码，最终允许在银行账户之间进行计算机转账业务。许多公司可以实现银行账户，但它们都是彼此赞同表示银行账户的所有类都实现接口 `IBankAccount`。该接口包含一个用于存取款的方法和一个返回余额的属性。正是这个接口还允许外部代码识别由不同银行账户实现的各种银行账户类。我们的目的是允许银行账户彼此通信，以便在账户之间进行转账业务，但还没有介绍这个功能。

为了使例子简单一些，我们把本例子的所有代码都放在同一个源文件中，但实际上不同的银行账户类不仅会编译到不同的程序集中，而且这些程序集位于不同银行的不同机器上。但这些内容对我们的目的过于复杂了。为了保留一定的真实性，我们为不同的公司定义不同的名称空间。

首先，需要定义 `IBankAccount` 接口：

```
namespace Wrox.ProCSharp
{
    public interface IBankAccount
    {
        void PayIn(decimal amount);
        bool Withdraw(decimal amount);
        decimal Balance
        {
            get;
        }
    }
}
```

注意，接口的名称为 `IBankAccount`。接口名称通常上以字母 `I` 开头，以便知道这是一个接口。



如第2章所述,在大多数情况下,.NET 的用法规则不鼓励采用所谓的 Hungarian 表示法,在名称的前面加一个字母,表示所定义对象的类型。接口是 Hungarian 表示法推荐采用的几种名称之一。

现在可以编写表示银行账户的类了。这些类不必彼此相关,它们可以是完全不同的类。但它们都表示银行账户,因为它们都实现了 `IBankAccount` 接口。

下面是第一个类,一个由 Royal Bank of Venus 运行的存款账户:



可从
wrox.com
下载源代码

```
namespace Wrox.ProCSharp.VenusBank
{
    public class SaverAccount: IBankAccount
    {
        private decimal balance;
        public void PayIn(decimal amount)
        {
            balance += amount;
        }
        public bool Withdraw(decimal amount)
        {
            if (balance >= amount)
            {
                balance -= amount;
                return true;
            }
            Console.WriteLine("Withdrawal attempt failed.");
            return false;
        }
        public decimal Balance
        {
            get
            {
                return balance;
            }
        }
        public override string ToString()
        {
            return String.Format("Venus Bank Saver: Balance = {0,6:C}", balance);
        }
    }
}
```

代码段 BankAccounts.cs

实现这个类的代码的作用一目了然。其中包含一个私有字段 `balance`, 当存款或取款时就调整这个字段。如果因为账户中的金额不足而取款失败, 就会显示一条错误消息。还要注意, 因为我们要使代码尽可能简单, 所以不实现额外的属性, 如账户持有人的姓名。在现实生活中, 这是最基本的

信息,但对于本例不必要这么复杂。

在这段代码中,唯一有趣的一行是类的声明:

```
public class SaverAccount: IBankAccount
```

SaverAccount 派生自一个接口 **IBankAccount**, 我们没有明确指出任何其他基类(当然这表示 **SaverAccount** 直接派生自 **System.Object**)。另外,从接口中派生完全独立于从类中派生。

SaverAccount 派生自 **IBankAccount**, 表示它获得了 **IBankAccount** 的所有成员,但接口实际上并不实现其方法,所以 **SaverAccount** 必须提供这些方法的所有实现代码。如果缺少实现代码,编译器就会产生错误。接口仅表示其成员的存在性,类负责确定这些成员是虚拟还是抽象的(但只有在类本身是抽象的,这些函数才能是抽象的)。在本例中,接口的任何函数不必是虚拟的。

为了说明不同的类如何实现相同的接口,下面假定 **Planetary Bank of Jupiter** 还实现一个类 **GoldAccount** 来表示其银行账户中的一个:

```
namespace Wrox.ProCSharp.JupiterBank
{
    public class GoldAccount: IBankAccount
    {
        // etc
    }
}
```

这里没有列出 **GoldAccount** 类的细节,因为在本例中它基本上与 **SaverAccount** 的实现代码相同。**GoldAccount** 与 **VenusAccount** 没有关系,它们只是碰巧实现相同的接口而已。

有了自己的类后,就可以测试它们了。首先需要一些 **using** 语句:

```
using System;
using Wrox.ProCSharp;
using Wrox.ProCSharp.VenusBank;
using Wrox.ProCSharp.JupiterBank;
```

然后需要一个 **Main()** 方法:

```
namespace Wrox.ProCSharp
{
    class MainEntryPoint
    {
        static void Main()
        {
            IBankAccount venusAccount = new SaverAccount();
            IBankAccount jupiterAccount = new GoldAccount();
            venusAccount.PayIn(200);
            venusAccount.Withdraw(100);
            Console.WriteLine(venusAccount.ToString());
            jupiterAccount.PayIn(500);
            jupiterAccount.Withdraw(600);
            jupiterAccount.Withdraw(100);
            Console.WriteLine(jupiterAccount.ToString());
        }
    }
}
```

这段代码(如果下载本例子, 就会发现它在 `BankAccounts.cs` 文件中)的执行结果如下:

```
C: > BankAccounts
Venus Bank Saver: Balance = £100.00
Withdrawal attempt failed.
Jupiter Bank Saver: Balance = £400.00
```

在这段代码中, 要点是把两个引用变量声明为 `IBankAccount` 引用的方式。这表示它们可以指向实现这个接口的任何类的任何实例。但我们只能通过这些引用调用接口的一部分方法——如果要调用由类实现的但不在接口中的方法, 就需要把引用强制转换为合适的类型。在这段代码中, 我们调用了 `ToString()`(不是 `IBankAccount` 实现的), 但没有进行任何显式的强制转换, 这只是因为 `ToString()` 是一个 `System.Object` 方法, 因此 C# 编译器知道任何类都支持这个方法(换言之, 从任何接口到 `System.Object` 的数据类型强制转换是隐式的)。第 7 章将介绍强制转换的语法。

接口引用完全可以看做是类引用——但接口引用的强大之处在于, 它可以引用任何实现该接口的类。例如, 我们可以构造接口数组, 其中数组的每个元素都是不同的类:

```
IBankAccount[] accounts = new IBankAccount[2];
accounts[0] = new SaverAccount();
accounts[1] = new GoldAccount();
```

但注意, 如果编写了如下代码, 就会生成一个编译错误:

```
accounts[1] = new SomeOtherClass(); // SomeOtherClass does NOT implement
// IBankAccount: WRONG!!
```

这会导致一个如下所示的编译错误:

```
Cannot implicitly convert type 'Wrox.ProCSharp.SomeOtherClass' to
'Wrox.ProCSharp.IBankAccount'
```

4.4.2 派生的接口

接口可以彼此继承, 其方式与类的继承方式相同。下面通过定义一个新的接口 `ITransferBankAccount` 来说明这个概念, 该接口的功能与 `IBankAccount` 相同, 只是又定义了一个方法, 把资金直接转到另一个账户上。

```
namespace Wrox.ProCSharp
{
    public interface ITransferBankAccount: IBankAccount
    {
        bool TransferTo(IBankAccount destination, decimal amount);
    }
}
```

因为 `ITransferBankAccount` 派生自 `IBankAccount`, 所以它拥有 `IBankAccount` 的所有成员和它自己的成员。这表示实现(派生自) `ITransferBankAccount` 的任何类都必须实现 `IBankAccount` 的所有方法

和在 `ITransferBankAccount` 中定义的新方法 `TransferTo()`。没有实现所有这些方法就会产生一个编译错误。

注意, `TransferTo()` 方法对于目标账户使用了 `IBankAccount` 接口引用。这说明了接口的用途: 在实现并调用这个方法时, 不必知道转账的对象类型, 只需知道该对象实现 `IBankAccount` 即可。

下面说明 `ITransferBankAccount`: 假定 `Planetary Bank of Jupiter` 还提供了一个当前账户。 `CurrentAccount` 类的大多数实现代码与 `SaverAccount` 和 `GoldAccount` 的实现代码相同(这仅是为了使例子更简单, 一般是不会这样的), 所以在下面的代码中, 我们仅突出显示了不同的地方:

```
public class CurrentAccount: ITransferBankAccount
{
    private decimal balance;
    public void PayIn(decimal amount)
    {
        balance += amount;
    }
    public bool Withdraw(decimal amount)
    {
        if (balance >= amount)
        {
            balance -= amount;
            return true;
        }
        Console.WriteLine("Withdrawal attempt failed.");
        return false;
    }
    public decimal Balance
    {
        get
        {
            return balance;
        }
    }
    public bool TransferTo(IBankAccount destination, decimal amount)
    {
        bool result;
        result = Withdraw(amount);
        if (result)
        {
            destination.PayIn(amount);
        }
        return result;
    }
    public override string ToString()
    {
        return String.Format("Jupiter Bank Current Account: Balance = {0,6:C}",
            balance);
    }
}
```

可以用下面的代码验证该类:



可从
wrox.com
下载源代码

```
static void Main()
{
    IBankAccount venusAccount = new SaverAccount();
    ITransferBankAccount jupiterAccount = new CurrentAccount();
    venusAccount.PayIn(200);
    jupiterAccount.PayIn(500);
    jupiterAccount.TransferTo(venusAccount, 100);
    Console.WriteLine(venusAccount.ToString());
    Console.WriteLine(jupiterAccount.ToString());
}
```

代码段 CurrentAccounts.cs

这段代码(CurrentAccount.cs)的结果如下所示,可以验证,其中说明了正确的转账金额:

```
C: > CurrentAccount
Venus Bank Saver: Balance = £300.00
Jupiter Bank Current Account: Balance = £400.00
```

4.5 小结

本章介绍了如何在 C#中进行继承代码。C#支持多接口继承和单一实现继承,还提供了许多有用的语法结构,以使代码更健壮,如 `override` 关键字,它表示函数应在何时重写基类函数, `new` 关键字表示函数在何时隐藏基类函数,构造函数初始化器的硬性规则可以确保构造函数以健壮的方式进行交互操作。

第 5 章

泛 型

本章内容:

- 泛型概述
- 创建泛型类
- 泛型类的特性
- 泛型接口
- 泛型结构
- 泛型方法

.NET 自从 2.0 版本开始就支持泛型。泛型不仅是 C#编程语言的一部分,而且与程序集中的 IL(Intermediate Language, 中间语言)代码紧密地集成。有了泛型,就可以创建独立于被包含类型的类和方法了。我们不必给不同的类型编写功能相同的许多方法或类,只创建一个方法或类即可。

另一个减少代码的选项是使用 Object 类,但 Object 类不是类型安全的。泛型类使用泛型类型,并可以根据需要用特定的类型替换泛型类型。这就保证了类型安全性:如果某个类型不支持泛型类,编译器就会出现错误。

泛型不仅限于类,本章还将介绍用于接口和方法的泛型。用于委托的泛型参见第 8 章。

5.1 概述

泛型并不是一个全新的结构,其他语言中有类似的概念。例如,C++模板就与泛型相似。但是,C++模板和.NET 泛型之间有一个很大的区别:对于 C++模板,在用特定的类型实例化模板时,需要模板的源代码。相反,泛型不仅是 C#语言的一种结构,而且是 CLR 定义的。所以,即使泛型类是在 C# 中定义的,也可以在 Visual Basic 中用一个特定的类型实例化该泛型。

下面几节介绍泛型的优点和缺点,尤其是:

- 性能
- 类型安全性
- 二进制代码重用
- 代码的扩展
- 命名约定

5.1.1 性能

泛型的一个主要优点是性能。第 10 章介绍了 `System.Collections` 和 `System.Collections.Generic` 名称空间的泛型和非泛型集合类。对值类型使用非泛型集合类，在把值类型转换为引用类型，和把引用类型转换为值类型时，需要进行装箱和拆箱操作。



装箱和拆箱详见第 7 章，这里仅简要复习一下这些术语。

值类型存储在栈上，引用类型存储在堆上。C# 类是引用类型，结构是值类型。NET 很容易把值类型转换为引用类型，所以可以在需要对象(对象是引用类型)的任意地方使用值类型。例如，`int` 可以赋予一个对象。从值类型转换为引用类型称为装箱。如果方法需要把一个对象作为参数，同时传递一个值类型，装箱操作就会自动进行。另一方面，装箱的值类型可以使用拆箱操作转换为值类型。在拆箱时，需要使用类型强制转换运算符。

下面的例子显示了 `System.Collections` 名称空间中的 `ArrayList` 类。`ArrayList` 存储对象，`Add()` 方法定义为需要把一个对象作为参数，所以要装箱一个整数类型。在读取 `ArrayList` 中的值时，要进行拆箱，把对象转换为整数类型。可以使用类型强制转换运算符把 `ArrayList` 集合的第一个元素赋予变量 `i1`，在访问 `int` 类型的变量 `i2` 的 `foreach` 语句中，也要使用类型强制转换运算符：

```
var list = new ArrayList();
list.Add(44); // boxing — convert a value type to a reference type

int i1 = (int)list[0]; // unboxing — convert a reference type to
                      // a value type
foreach (int i2 in list)
{
    Console.WriteLine(i2); // unboxing
}
```

装箱和拆箱操作很容易使用，但性能损失比较大，遍历许多项时尤其如此。

`System.Collections.Generic` 名称空间中的 `List<T>` 类不使用对象，而是在使用时定义类型。在下面的例子中，`List<T>` 类的泛型类型定义为 `int`，所以 `int` 类型在 JIT 编译器动态生成的类中使用，不再进行装箱和拆箱操作：

```
var list = new List<int>();
list.Add(44); // no boxing — value types are stored in the List<int>

int i1 = list[0]; // no unboxing, no cast needed
foreach (int i2 in list)
{
    Console.WriteLine(i2);
}
```

5.1.2 类型安全

泛型的另一个特性是类型安全。与 `ArrayList` 类一样，如果使用对象，就可以在这个集合中添加任意类型。下面的例子在 `ArrayList` 类型的集合中添加一个整数、一个字符串和一个 `MyClass` 类型的对象：

```
var list = new ArrayList();
list.Add(44);
list.Add("mystring");
list.Add(new MyClass());
```

如果这个集合使用下面的 `foreach` 语句迭代，而该 `foreach` 语句使用整数元素来迭代，编译器就会编译这段代码。但并不是集合中的所有元素都可以强制转换为 `int`，所以会出现一个运行异常：

```
foreach (int i in list)
{
    Console.WriteLine(i);
}
```

错误应尽早发现。在泛型类 `List<T>` 中，泛型类型 `T` 定义了允许使用的类型。有了 `List<int>` 的定义，就只能把整数类型添加到集合中。编译器不会编译这段代码，因为 `Add()` 方法的参数无效：

```
var list = new List < int > ();
list.Add(44);
list.Add("mystring"); // compile time error
list.Add(new MyClass()); // compile time error
```

5.1.3 二进制代码的重用

泛型允许更好地重用二进制代码。泛型类可以定义一次，并且可以用许多不同的类型实例化。不需要像 C++ 模板那样访问源代码。

例如，`System.Collections.Generic` 名称空间中的 `List<T>` 类用一个 `int`、一个字符串和一个 `MyClass` 类型实例化：

```
var list = new List<int>();
list.Add(44);

var stringList = new List<string>();
stringList.Add("mystring");

var myClassList = new List<MyClass>();
myClassList.Add(new MyClass());
```

泛型类型可以在一种语言中定义，在任何其他 .NET 语言中使用。

5.1.4 代码的扩展

在用不同的特定类型实例化泛型时，会创建多少代码？

因为泛型类的定义会放在程序集中，所以用特定类型实例化泛型类不会在 IL 代码中复制这些

类。但是，在 JIT 编译器把泛型类编译为本地代码时，会给每个值类型创建一个新类。引用类型共享同一个本地类的所有相同的实现代码。这是因为引用类型在实例化的泛型类中只需要 4 个字节的内存地址(32 位系统)，就可以引用一个引用类型。值类型包含在实例化的泛型类的内存中，同时因为每个值类型对内存的要求都不同，所以要为每个值类型实例化一个新类。

5.1.5 命名约定

如果在程序中使用泛型，在区分泛型类型和非泛型类型时就会有一定的帮助。下面是泛型类型的命名规则：

- 泛型类型的名称用字母 T 作为前缀。
- 如果没有特殊的要求，泛型类型允许用任意类替代，且只使用了一个泛型类型，就可以用字符 T 作为泛型类型的名称。

```
public class List<T> { }

public class LinkedList<T> { }
```

- 如果泛型类型有特定的要求(例如，它必须实现一个接口或派生自基类)，或者使用了两个或多个泛型类型，就应给泛型类型使用描述性的名称：

```
public delegate void EventHandler<TEventArgs>(object sender, TEventArgs e);

public delegate TOutput Converter<TInput, TOutput>(TInput from);

public class SortedList<TKey, TValue> { }
```

5.2 创建泛型类

首先介绍一个一般的、非泛型的简化链表类，它可以包含任意类型的对象，以后再把这个类转化为泛型类。

在链表中，一个元素引用下一个元素。所以必须创建一个类，它将对象封装在链表中，并引用下一个对象。类 `LinkedListNode` 包含一个属性 `Value`，该属性用构造函数初始化。另外，`LinkedListNode` 类包含对链表中下一个元素和上一个元素的引用，这些元素都可以从属性中访问。



可从
wrox.com
下载源代码

```
public class LinkedListNode
{
    public LinkedListNode(object value)
    {
        this.Value = value;
    }

    public object Value { get; private set; }

    public LinkedListNode Next { get; internal set; }
    public LinkedListNode Prev { get; internal set; }
}
```

代码段 `LinkedListObjects/LinkedListNode.cs`

`LinkedList` 类包含 `LinkedListNode` 类型的 `first` 和 `last` 属性,它们分别标记了链表的头尾。`AddLast()` 方法在链表尾添加一个新元素。首先创建一个 `LinkedListNode` 类型的对象。如果链表是空的, `first` 和 `last` 属性就设置为该新元素; 否则, 就把新元素添加为链表中的最后一个元素。通过实现 `GetEnumerator()` 方法时, 可以用 `foreach` 语句遍历链表。`GetEnumerator()` 方法使用 `yield` 语句创建一个枚举器类型。



可从
wrox.com
下载源代码

```
public class LinkedList: IEnumerable
{
    public LinkedListNode First { get; private set; }
    public LinkedListNode Last { get; private set; }

    public LinkedListNode AddLast(object node)
    {
        var newNode = new LinkedListNode(node);
        if (First == null)
        {
            First = newNode;
            newNode.Prev = Last;
            Last = First;
        }
        else
        {
            LinkedListNode previous = Last;
            Last.Next = newNode;
            Last = newNode;
            Last.Prev = previous;
        }
        return newNode;
    }

    public IEnumerator GetEnumerator()
    {
        LinkedListNode current = First;
        while (current != null)
        {
            yield return current.Value;
            current = current.Next;
        }
    }
}
```

代码段 `LinkedListObjects/LinkedListNode.cs`



yield 语句参见第 6 章。

现在可以对于任意类型使用 `LinkedList` 类了。在下面的代码段中, 实例化了一个新 `LinkedList` 对象, 添加了两个整数类型和一个字符串类型。整数类型要转换为一个对象, 所以执行装箱操作, 如

前面所述。通过 `foreach` 语句执行拆箱操作。在 `foreach` 语句中，链表中的元素被强制转换为整数，所以对于链表中的第 3 个元素，会发生一个运行异常，因为把它强制转换为 `int` 时会失败。



可从
wrox.com
下载源代码

```
var list1 = new LinkedList();
list1.AddLast(2);
list1.AddLast(4);
list1.AddLast("6");

foreach (int i in list1)
{
    Console.WriteLine(i);
}
```

代码段 `LinkedListObjects/Program.cs`

下面创建链表的泛型版本。泛型类的定义与一般类类似，只是要使用泛型类型声明。之后，泛型类型就可以在类中用作一个字段成员，或者方法的参数类型。`LinkedListNode` 类用一个泛型类型 `T` 声明。属性 `Value` 的类型是 `T`，而不是 `object`。构造函数也变为可以接受 `T` 类型的对象。也可以返回和设置泛型类型，所以属性 `Next` 和 `Prev` 的类型是 `LinkedListNode<T>`。



可从
wrox.com
下载源代码

```
public class LinkedListNode<T>
{
    public LinkedListNode(T value)
    {
        this.Value = value;
    }

    public T Value { get; private set; }
    public LinkedListNode<T> Next { get; internal set; }
    public LinkedListNode<T> Prev { get; internal set; }
}
```

代码段 `LinkedListSample/LinkedListNode.cs`

下面的代码把 `LinkedList` 类也改为泛型类。`LinkedList<T>` 包含 `LinkedListNode<T>` 元素。`LinkedList` 中的类型 `T` 定义了类型 `T` 的属性 `first` 和 `last`。`AddLast()` 方法现在接受类型 `T` 的参数，并实例化 `LinkedListNode<T>` 类型的对象。

除了 `IEnumerable` 接口，还有一个泛型版本 `IEnumerable<T>`。`IEnumerable<T>` 派生自 `IEnumerable`，添加了返回 `IEnumerator<T>` 的 `GetEnumerator()` 方法，`LinkedList<T>` 实现泛型接口 `IEnumerable<T>`。



枚举与接口 `IEnumerable` 和 `IEnumerator` 详见第 6 章。



可从
wrox.com
下载源代码

```
public class LinkedList<T> : IEnumerable<T>
{
    public LinkedListNode<T> First { get; private set; }
    public LinkedListNode<T> Last { get; private set; }

    public LinkedListNode<T> AddLast(T node)
    {

```

```

        var newNode = new LinkedListNode(node);
        if (First == null)
        {
            First = newNode;
            newNode.Prev = Last;
            Last = First;
        }
        else
        {
            LinklistNode previous = Last;
            Last.Next = newNode;
            Last = newNode;
            Last.Prev = previous;
        }
        return newNode;
    }

    public IEnumerator<T> GetEnumerator()
    {
        LinkedListNode<T> current = First;

        while (current != null)
        {
            yield return current.Value;
            current = current.Next;
        }
    }

    IEnumerator IEnumerable.GetEnumerator()
    {
        return GetEnumerator();
    }
}

```

代码段 LinkedListSample/LinkedList.cs

使用泛型类 `LinkedList<T>`，可以用 `int` 类型实例化它，且无需装箱操作。如果不使用 `AddLast()` 方法传递 `int`，就会出现一个编译错误。使用泛型 `IEnumerable<T>`，`foreach` 语句也是类型安全的，如果 `foreach` 语句中的变量不是 `int`，就会出现一个编译错误。



可从
wrox.com
下载源代码

```

var list2 = new LinkedList<int>();
list2.AddLast(1);
list2.AddLast(3);
list2.AddLast(5);

foreach (int i in list2)
{
    Console.WriteLine(i);
}

```

代码段 LinkedListSample/Program.cs

同样，可以对于字符串类型使用泛型 `LinkedList<T>`，将字符串传递给 `AddLast()` 方法。

```

var list3 = new LinkedList<string>();
list3.AddLast("2");
list3.AddLast("four");
list3.AddLast("foo");

foreach (string s in list3)
{
    Console.WriteLine(s);
}

```



每个处理对象类型的类都可以有泛型实现方式。另外，如果类使用了层次结构，泛型就非常有助于消除类型强制转换操作。

5.3 泛型类的功能

在创建泛型类时，还需要一些其他 C# 关键字。例如，不能把 `null` 赋予泛型类型。此时，如下一节所述，可以使用 `default` 关键字。如果泛型类型不需要 `Object` 类的功能，但需要调用泛型类上的某些特定方法，就可以定义约束。

本节讨论如下主题：

- 默认值
- 约束
- 继承
- 静态成员

首先介绍一个使用泛型文档管理器的示例。文档管理器用于从队列中读写文档。先创建一个新的控制台项目 `DocumentManager`，并添加 `DocumentManager<T>` 类。`AddDocument()` 方法将一个文档添加到队列中。如果队列不为空，`IsDocumentAvailable` 只读属性就返回 `true`。



可从
wrox.com
下载源代码

```

using System;
using System.Collections.Generic;

namespace Wrox.ProCSharp.Generics
{
    public class DocumentManager<T>
    {
        private readonly Queue<T> documentQueue = new Queue<T>();

        public void AddDocument(T doc)
        {
            lock (this)
            {
                documentQueue.Enqueue(doc);
            }
        }
    }
}

```

```

        public bool IsDocumentAvailable
        {
            get { return documentQueue.Count > 0; }
        }
    }
}

```

代码段 DocumentManager/DocumentManager.cs

5.3.1 默认值

现在给 `DocumentManager<T>` 类添加一个 `GetDocument()` 方法。在这个方法中，应把类型 `T` 指定为 `null`。但是，不能把 `null` 赋予泛型类型。原因是泛型类型也可以实例化为值类型，而 `null` 只能用于引用类型。为了解决这个问题，可以使用 `default` 关键字。通过 `default` 关键字，将 `null` 赋予引用类型，将 `0` 赋予值类型。

```

public T GetDocument()
{
    T doc = default(T);
    lock (this)
    {
        doc = documentQueue.Dequeue();
    }
    return doc;
}

```



`default` 关键字根据上下文可以有多种含义。switch 语句使用 `default` 定义默认情况。在泛型中，根据泛型类型是引用类型还是值类型，泛型 `default` 用于将泛型类型初始化为 `null` 或 `0`。

5.3.2 约束

如果泛型类需要调用泛型类型中的方法，就必须添加约束。对于 `DocumentManager<T>`，文档的所有标题应在 `DisplayAllDocuments()` 方法中显示。`Document` 类实现带有 `Title` 和 `Content` 属性的 `IDocument` 接口：



可从
wrox.com
下载源代码

```

public interface IDocument
{
    string Title { get; set; }
    string Content { get; set; }
}

public class Document: IDocument
{
    public Document()
    {

```

```

    }

    public Document(string title, string content)
    {
        this.Title = title;
        this.Content = content;
    }

    public string Title { get; set; }
    public string Content { get; set; }
}

```

代码段 DocumentManager/Document.cs

要使用 `DocumentManager<T>` 类显示文档，可以将类型 `T` 强制转换为 `IDocument` 接口，以显示标题：



可从
wrox.com
下载源代码

```

public void DisplayAllDocuments()
{
    foreach (T doc in documentQueue)
    {
        Console.WriteLine(((IDocument)doc).Title);
    }
}

```

代码段 DocumentManager/DocumentManager.cs

问题是，如果类型 `T` 没有实现 `IDocument` 接口，这个类型强制转换就会导致一个运行异常。最好给 `DocumentManager<TDocument>` 类定义一个约束：`TDocument` 类型必须实现 `IDocument` 接口。为了在泛型类型的名称中指定该要求，将 `T` 改为 `TDocument`。 `where` 子句指定了实现 `IDocument` 接口的要求。

```

public class DocumentManager<TDocument>
    where TDocument: IDocument
{

```

这样就可以编写 `foreach` 语句，从而使类型 `TDocument` 包含属性 `Title`。 `Visual Studio IntelliSense` 和编译器都会提供这个支持。

```

    public void DisplayAllDocuments()
    {
        foreach (TDocument doc in documentQueue)
        {
            Console.WriteLine(doc.Title);
        }
    }
}

```

在 `Main()` 方法中，用 `Document` 类型实例化 `DocumentManager<T>` 类，而 `Document` 类型实现了需要的 `IDocument` 接口。接着添加和显示新文档，检索其中一个文档：



可从
wrox.com
下载源代码

```
static void Main()
{
    var dm = new DocumentManager<Document>();
    dm.AddDocument(new Document("Title A", "Sample A"));
    dm.AddDocument(new Document("Title B", "Sample B"));

    dm.DisplayAllDocuments();

    if (dm.IsDocumentAvailable)
    {
        Document d = dm.GetDocument();
        Console.WriteLine(d.Content);
    }
}
```

代码段 DocumentManager/Program.cs

DocumentManager 现在可以处理任何实现了 IDocument 接口的类。

在示例应用程序中，介绍了接口约束。泛型支持几种约束类型，如表 5-1 所示。

表 5-1

约 束	说 明
where T: struct	对于结构约束，类型 T 必须是值类型
where T: class	类约束指定类型 T 必须是引用类型
where T: IFoo	指定类型 T 必须实现接口 IFoo
where T: Foo	指定类型 T 必须派生自基类 Foo
where T: new()	这是一个构造函数约束，指定类型 T 必须有一个默认构造函数
where T1: T2	这个约束也可以指定，类型 T1 派生自泛型类型 T2。该约束也称为裸类型约束



只能为默认构造函数定义构造函数约束，不能为其他构造函数定义构造函数约束。

使用泛型类型还可以合并多个约束。where T: IFoo, new() 约束和 MyClass<T> 声明指定，类型 T 必须实现 IFoo 接口，且必须有一个默认构造函数。

```
public class MyClass<T>
    where T: IFoo, new()
{
    //...
```



在 C# 中，where 子句的一个重要限制是，不能定义必须由泛型类型实现的运算符。运算符不能在接口中定义。在 where 子句中，只能定义基类、接口和默认构造函数。

5.3.3 继承

前面创建的 `LinkedList<T>` 类实现了 `IEnumerable<out T>` 接口：

```
public class LinkedList<T> : IEnumerable<out T>
{
    //...
```

泛型类型可以实现泛型接口，也可以派生自一个类。泛型类可以派生自泛型基类：

```
public class Base<T>
{
}

public class Derived<T> : Base<T>
{
}
```

其要求是必须重复接口的泛型类型，或者必须指定基类的类型，如下例所示：

```
public class Base<T>
{
}

public class Derived<T> : Base<string>
{
}
```

于是，派生类可以是泛型类或非泛型类。例如，可以定义一个抽象的泛型基类，它在派生类中用一个具体的类型实现。这允许对特定类型执行特殊的操作：

```
public abstract class Calc<T>
{
    public abstract T Add(T x, T y);
    public abstract T Sub(T x, T y);
}

public class IntCalc: Calc<int>
{
    public override int Add(int x, int y)
    {
        return x + y;
    }

    public override int Sub(int x, int y)
    {
        return x - y;
    }
}
```

5.3.4 静态成员

泛型类的静态成员需要特别关注。泛型类的静态成员只能在类的一个实例中共享。下面看一个例子，其中 `StaticDemo<T>` 类包含静态字段 `x`：

```
public class StaticDemo<T>
{
    public static int x;
}
```

由于同时对一个 `string` 类型和一个 `int` 类型使用了 `StaticDemo<T>` 类，所以存在两组静态字段：

```
StaticDemo<string>.x = 4;
StaticDemo<int>.x = 5;
Console.WriteLine(StaticDemo<string>.x); // writes 4
```

5.4 泛型接口

使用泛型可以定义接口，在接口中定义的方法可以带泛型参数。在链表的示例中，就实现了 `IEnumerable<out T>` 接口，它定义了 `GetEnumerator()` 方法，以返回 `IEnumerator<out T>`。 .NET 为不同的情况提供了许多泛型接口，例如 `Comparable<T>`、`ICollection<T>` 和 `IExtensibleObject<T>`。同一个接口常常存在比较老的非泛型版本，例如，.NET 1.0 有基于对象的 `Comparable` 接口。`Comparable<in T>` 基于一个泛型类型：

```
public interface Comparable<in T>
{
    int CompareTo(T other);
}
```

比较老的非泛型接口 `Comparable` 需要一个带 `CompareTo()` 方法的对象。这需要强制转换为特定的类型，例如，`Person` 类要使用 `LastName` 属性，就需要使用 `CompareTo()` 方法：

```
public class Person: Comparable
{
    public int CompareTo(object obj)
    {
        Person other = obj as Person;
        return this.LastName.CompareTo(other.LastName);
    }
}
```

实现泛型版本时，不再需要将 `object` 的类型强制转换为 `Person`：

```
public class Person: Comparable < Person >
{
    public int CompareTo(Person other)
    {
        return this.LastName.CompareTo(other.LastName);
    }
}
```

```

    }
    //...

```

5.4.1 协变和抗变

在.NET 4 之前, 泛型接口是不变的。 .NET 4 通过协变和抗变为泛型接口和泛型委托添加了一个重要的扩展。协变和抗变指对参数和返回值的类型进行转换。例如, 可以给一个需要 `Shape` 参数的方法传送 `Rectangle` 参数吗? 下面用示例说明这些扩展的优点。

在.NET 中, 参数类型是协变的。假定有 `Shape` 和 `Rectangle` 类, `Rectangle` 派生自 `Shape` 基类。声明 `Display()` 方法是为了接受 `Shape` 类型的对象作为其参数:

```
public void Display(Shape o) { }
```

现在可以传递派生自 `Shape` 基类的任意对象。因为 `Rectangle` 派生自 `Shape`, 所以 `Rectangle` 满足 `Shape` 的所有要求, 编译器接受这个方法调用:

```
Rectangle r = new Rectangle { Width= 5, Height=2.5};
Display(r);
```

方法的返回类型是抗变的。当方法返回一个 `Shape` 时, 不能把它赋予 `Rectangle`, 因为 `Shape` 不一定总是 `Rectangle`。反过来是可行的: 如果一个方法像 `GetRectangle()` 方法那样返回一个 `Rectangle`,

```
public Rectangle GetRectangle();
```

就可以把结果赋予某个 `Shape`:

```
Shape s = GetRectangle();
```

在.NET Framework 4 版本之前, 这种行为方式不适用于泛型。在 C# 4 中, 扩展后的语言支持泛型接口和泛型委托的协变和抗变。下面开始定义 `Shape` 基类和 `Rectangle` 类:



可从
wrox.com
下载源代码

```
public class Shape
{
    public double Width { get; set; }
    public double Height { get; set; }

    public override string ToString()
    {
        return String.Format("Width: {0}, Height: {1}", Width, Height);
    }
}
```

代码段 Variance/Shape.cs

```
public class Rectangle: Shape
{
}
```

代码段 Variance/Rectangle.cs

5.4.2 泛型接口的协变

如果泛型类型用 `out` 关键字标注，泛型接口就是协变的。这也意味着返回类型只能是 `T`。接口 `IIndex` 与类型 `T` 是协变的，并从一个只读索引器中返回这个类型：

```
public interface IIndex<out T>
{
    T this[int index] { get; }
    int Count { get; }
}
```

代码段 Variance/IIndex.cs



如果对接口 `IIndex` 使用了只读索引器，就把泛型类型 `T` 传递给方法，并从方法中检索这个类型。这不能通过协变来实现——泛型类型必须定义为不变的。不使用 `out` 和 `in` 标注，就可以把类型定义为不变的。

`IIndex<T>` 接口用 `RectangleCollection` 类来实现。`RectangleCollection` 类为泛型类型 `T` 定义了 `Rectangle`：



可从
wrox.com
下载源代码

```
public class RectangleCollection: IIndex<Rectangle>
{
    private Rectangle[] data = new Rectangle[3]
    {
        new Rectangle { Height=2, Width=5},
        new Rectangle { Height=3, Width=7},
        new Rectangle { Height=4.5, Width=2.9}
    };

    public static RectangleCollection GetRectangles()
    {
        return new RectangleCollection();
    }

    public Rectangle this[int index]
    {
        get
        {
            if (index < 0 || index > data.Length)
                throw new ArgumentOutOfRangeException("index");
            return data[index];
        }
    }

    public int Count
    {
        get
        {
```

```
return data.Length;
```

```
}
```

```
}
```

```
}
```

代码段 Variance/RectangleCollection.cs

`RectangleCollection.GetRectangle()`方法返回一个实现 `IIndex<Rectangle>`接口的 `RectangleCollection`类, 所以可以把返回值赋予 `IIndex<Rectangle>`类型的变量 `rectangle`。因为接口是协变的, 所以也可以把返回值赋予 `IIndex<Shape>`类型的变量。Shape 不需要 `Rectangle` 没有提供的内容。使用 `shapes` 变量, 就可以在 `for` 循环中使用接口中的索引器和 `Count` 属性:



可从
wrox.com
下载源代码

```
static void Main()
{
    IIndex<Rectangle> rectangles = RectangleCollection.GetRectangles();
    IIndex<Shape> shapes = rectangles;

    for (int i = 0; i < shapes.Count; i++)
    {
        Console.WriteLine(shapes[i]);
    }
}
```

代码段 Variance/Program.cs

5.4.3 泛型接口的抗变

如果泛型类型用 `in` 关键字标注, 泛型接口就是抗变的。这样, 接口只能把泛型类型 `T` 用作其方法的输入:



可从
wrox.com
下载源代码

```
public interface IDisplay<in T>
{
    void Show(T item);
}
```

代码段 Variance/IDisplay.cs

`ShapeDisplay` 类实现 `IDisplay<Shape>`, 并使用 `Shape` 对象作为输入参数:



可从
wrox.com
下载源代码

```
public class ShapeDisplay: IDisplay<Shape>
{
    public void Show(Shape s)
    {
        Console.WriteLine("{0} Width: {1}, Height: {2}", s.GetType().Name,
            s.Width, s.Height);
    }
}
```

代码段 Variance/ShapeDisplay.cs

创建 `ShapeDisplay` 的一个新实例, 会返回 `IDisplay<Shape>`, 并把它赋予 `shapeDisplay` 变量。因为 `IDisplay<T>` 是抗变的, 所以可以把结果赋予 `IDisplay<Rectangle>`, 其中 `Rectangle` 派生自 `Shape`。

这次接口的方法只能把泛型类型定义为输入，而 `Rectangle` 满足 `Shape` 的所有要求：



可从
wrox.com
下载源代码

```
static void Main()
{
    //...

    IDisplay<Shape> shapeDisplay = new ShapeDisplay();
    IDisplay<Rectangle> rectangleDisplay = shapeDisplay;
    rectangleDisplay.Show(rectangles[0]);
}
```

代码段 Variance/Program.cs

5.5 泛型结构

与类相似，结构也可以是泛型的。它们非常类似于泛型类，只是没有继承特性。本节介绍泛型结构 `Nullable<T>`，它由 .NET Framework 定义。

.NET Framework 中的一个泛型结构是 `Nullable<T>`。数据库中的数字和编程语言中的数字有显著不同的特征，因为数据库中的数字可以为空，而 C# 中的数字不能为空。`Int32` 是一个结构，而结构的实现同值类型，所以结构不能为空。这个问题不仅存在于在数据库中，也存在于把 XML 数据映射到 .NET 类型。

这种区别常常令人很头痛，映射数据也要多做许多辅助工作。一种解决方案是把数据库和 XML 文件中的数字映射为引用类型，因为引用类型可以为空值。但这也会在运行期间带来额外的系统开销。

使用 `Nullable<T>` 结构很容易解决这个问题。下面的代码段说明了如何定义 `Nullable<T>` 的一个简化版本。

结构 `Nullable<T>` 定义了一个约束：其中的泛型类型 `T` 必须是一个结构。把类定义为泛型类型后，就没有低系统开销这个优点了，而且因为类的对象可以为空，所以对类使用 `Nullable<T>` 类型是没有意义的。除了 `Nullable<T>` 定义的 `T` 类型之外，唯一的系统开销是 `hasValue` 布尔字段，它确定是设置对应的值，还是使之为空。除此之外，泛型结构还定义了只读属性 `HasValue` 和 `Value`，以及一些操作符重载。把 `Nullable<T>` 类型强制转换为 `T` 类型的操作符重载是显式定义的，因为当 `hasValue` 为 `false` 时，它会抛出一个异常。强制转换为 `Nullable<T>` 类型的操作符重载定义为隐式的，因为它总是能成功地转换：

```
public struct Nullable<T>
    where T: struct
{
    public Nullable(T value)
    {
        this.hasValue = true;
        this.value = value;
    }
    private bool hasValue;
    public bool HasValue
```

```

    {
        get
        {
            return hasValue;
        }
    }

private T value;
public T Value
{
    get
    {
        if (!hasValue)
        {
            throw new InvalidOperationException("no value");
        }
        return value;
    }
}

public static explicit operator T(Nullable<T> value)
{
    return value.Value;
}

public static implicit operator Nullable<T>(T value)
{
    return new Nullable<T>(value);
}

public override string ToString()
{
    if (!HasValue)
        return String.Empty;
    return this.value.ToString();
}
}

```

在这个例子中, `Nullable<T>` 用 `Nullable<int>` 实例化。变量 `x` 现在可以用作一个 `int`, 进行赋值或使用运算符执行一些计算。这是因为强制转换了 `Nullable<T>` 类型的运算符。但是, `x` 还可以为空。`Nullable<T>` 的 `HasValue` 和 `Value` 属性可以检查是否有一个值, 该值是否可以访问:

```

Nullable<int> x;
x = 4;
x += 3;
if (x.HasValue)
{
    int y = x.Value;
}
x = null;

```

因为可空类型使用得非常频繁, 所以 C# 有一种特殊的语法, 它用于定义可空类型的变量。定义

这类变量时，不使用泛型结构的语法，而使用“?”运算符。在下面的例子中，变量 x1 和 x2 都是可空的 int 类型的实例：

```
Nullable<int> x1;
int? x2;
```

可空类型可以与 null 和数字比较，如上所示。这里，x 的值与 null 比较，如果 x 不是 null，它就与小于 0 的值比较：

```
int? x = GetNullableType();
if (x == null)
{
    Console.WriteLine("x is null");
}
else if (x < 0)
{
    Console.WriteLine("x is smaller than 0");
}
```

知道了 `Nullable<T>` 是如何定义的之后，下面就使用可空类型。可空类型还可以与算术运算符一起使用。变量 x3 是变量 x1 和 x2 的和。如果这两个可空变量中任何一个的值是 null，它们的和就是 null。

```
int? x1 = GetNullableType();
int? x2 = GetNullableType();
int? x3 = x1 + x2;
```



这里调用的 `GetNullableType()` 只是一个占位符，它对于任何方法都返回一个可空的 int。为了进行测试，简单起见，可以使实现的 `GetNullableType()` 返回 null 或返回任意整数。

非可空类型可以转换为可空类型。从非可空类型转换为可空类型时，在不需要强制类型转换的地方可以进行隐式转换。这种转换总是成功的：

```
int y1 = 4;
int? x1 = y1;
```

但从可空类型转换为非可空类型可能会失败。如果可空类型的值是 null，并且把 null 值赋予非可空类型，就会抛出 `InvalidOperationException` 类型的异常。这就是需要类型强制转换运算符进行显式转换的原因：

```
int? x1 = GetNullableType();
int y1 = (int)x1;
```

如果不进行显式类型转换，还可以使用合并运算符(coalescing operator)从可空类型转换为非可空类型。合并运算符的语法是“??”，为转换定义了一个默认值，以防可空类型的值是 null。这里，如

果 `x1` 是 `null`, `y1` 的值就是 0。

```
int? x1 = GetNullableType();
int y1 = x1 ?? 0;
```

5.6 泛型方法

除了定义泛型类之外,还可以定义泛型方法。在泛型方法中,泛型类型用方法声明来定义。泛型方法可以在非泛型类中定义。

`Swap<T>()`方法把 `T` 定义为泛型类型,该泛型类型用于两个参数和一个变量 `temp`:

```
void Swap<T>(ref T x, ref T y)
{
    T temp;
    temp = x;
    x = y;
    y = temp;
}
```

把泛型类型赋予方法调用,就可以调用泛型方法:

```
int i = 4;
int j = 5;
Swap<int>(ref i, ref j);
```

但是,因为 C#编译器会通过调用 `Swap()`方法来获取参数的类型,所以不需要把泛型类型赋予方法调用。泛型方法可以像非泛型方法那样调用:

```
int i = 4;
int j = 5;
Swap(ref i, ref j);
```

5.6.1 泛型方法示例

下面的例子使用泛型方法累加集合中的所有元素。为了说明泛型方法的功能,下面使用包含 `Name` 和 `Balance` 属性的 `Account` 类:



```
public class Account
{
    public string Name { get; private set; }
    public decimal Balance { get; private set; }

    public Account(string name, Decimal balance)
    {
        this.Name = name;
        this.Balance = balance;
    }
}
```

代码段 `GenericMethods/Account.cs`

其中应累加余额的所有账户操作都添加到 List<Account> 类型的账户列表中:



可从
wrox.com
下载源代码

```
var accounts = new List < Account > ()
{
    new Account("Christian", 1500),
    new Account("Stephanie", 2200),
    new Account("Angela", 1800)
};
```

代码段 GenericMethods/Program.cs

累加所有 Account 对象的传统方式是用 foreach 语句遍历所有的 Account 对象,如下所示。foreach 语句使用 IEnumerable 接口迭代集合的元素,所以 AccumulateSimple()方法的参数是 IEnumerable 类型。foreach 语句处理实现 IEnumerable 接口的每个对象。这样,AccumulateSimple()方法就可以用于所有实现 IEnumerable<Account>接口的集合类。在这个方法的实现代码中,直接访问 Account 对象的 Balance 属性:



可从
wrox.com
下载源代码

```
public static class Algorithm
{
    public static decimal AccumulateSimple(IEnumerable<Account> source)
    {
        decimal sum = 0;
        foreach (Account a in source)
        {
            sum += a.Balance;
        }
        return sum;
    }
}
```

代码段 GenericMethods/Algorithm.cs

Accumulate()方法的调用方式如下:

```
decimal amount = Algorithm.AccumulateSimple(accounts);
```

5.6.2 带约束的泛型方法

第一个实现代码的问题是,它只能用于 Account 对象。使用泛型方法就可以避免这个问题。

Accumulate()方法的第二个版本接受实现了 IAccount 接口的任意类型。如前面的泛型类所述,泛型类型可以用 where 子句来限制。用于泛型类的这个子句也可以用于泛型方法。Accumulate()方法的参数改为 IEnumerable<T>。IEnumerable<T>是泛型集合类实现的泛型接口。



可从
wrox.com
下载源代码

```
public static decimal Accumulate<TAccount> (IEnumerable<TAccount> source)
    where TAccount: IAccount
{
    decimal sum = 0;
    foreach (TAccount a in source)
```

```

    {
        sum += a.Balance;
    }
    return sum;
}

```

代码段 GenericMethods/Algorithm.cs

重构的 Account 类现在为实现接口 IAccount:



可从
wrox.com
下载源代码

```

public class Account: Iaccount
{
    //...
}

```

代码段 GenericMethods/Account.cs

IAccount 接口定义了只读属性 Balance 和 Name:



可从
wrox.com
下载源代码

```

public interface IAccount
{
    decimal Balance { get; }
    string Name { get; }
}

```

代码段 GenericMethods/IAccount.cs

将 Account 类型定义为泛型类型参数, 就可以调用新的 Accumulate()方法:



可从
wrox.com
下载源代码

```

decimal amount = Algorithm.Accumulate<Account>(accounts);

```

代码段 GenericMethods/Program.cs

因为编译器会从方法的参数类型中自动推断出泛型类型参数, 所以以如下方式调用 Accumulate()方法是有效的:

```

decimal amount = Algorithm.Accumulate(accounts);

```

5.6.3 带委托的泛型方法

泛型类型实现 IAccount 接口的要求过于严格。下面的示例提示了, 如何通过传递一个泛型委托来修改 Accumulate()方法。第 8 章详细介绍了如何使用泛型委托, 以及如何使用 Lambda 表达式。

这个 Accumulate()方法使用两个泛型参数 T1 和 T2。第一个参数 T1 用于实现了 IEnumerable<T1> 参数的集合, 第二个参数使用泛型委托 Func<T1, T2, TResult>。其中, 第 2 个和第 3 个泛型参数都是 T2 类型。需要传递的方法有两个输入参数(T1 和 T2)和一个 T2 类型的返回值:



可从
wrox.com
下载源代码

```

public static T2 Accumulate<T1, T2> (IEnumerable<T1> source,
                                     Func<T1, T2, T2> action)
{
    T2 sum = default(T2);
    foreach (T1 item in source)
    {
        sum = action(item, sum);
    }
    return sum;
}

```

```

    {
        sum = action(item, sum);
    }
    return sum;
}

```

代码段 GenericMethods/Algorithm.cs

在调用这个方法时，需要指定泛型参数类型，因为编译器不能自动推断出该类型。对于方法的第一个参数，所赋予的 `accounts` 集合是 `IEnumerable<Account>` 类型。对于第二个参数，使用一个 Lambda 表达式来定义 `Account` 和 `decimal` 类型的两个参数，返回一个小数。对于每一项通过 `Accumulate()` 方法调用这个 Lambda 表达式：



可从
wrox.com
下载源代码

```

decimal amount = Algorithm.Accumulate<Account, decimal>(
    accounts, (item, sum) => sum += item.Balance);

```

代码段 GenericMethods/Program.cs

不要为这种语法伤脑筋。该示例仅说明了扩展 `Accumulate()` 方法的可能方式。第 8 章详细介绍了 Lambda 表达式。

5.6.4 泛型方法规范

泛型方法可以重载，为特定的类型定义规范。这也适用于带泛型参数的方法。`Foo()` 方法定义了两个版本，第 1 个版本接受一个泛型参数，第 2 个版本是用于 `int` 参数的专用版本。在编译期间，会使用最佳匹配。如果传递了一个 `int`，就选择带 `int` 参数的方法。对于任何其他参数类型，编译器会选择方法的泛型版本：



可从
wrox.com
下载源代码

```

public class MethodOverloads
{
    public void Foo<T> (T obj)
    {
        Console.WriteLine("Foo<T> (T obj), obj type: {0}", obj.GetType().Name);
    }

    public void Foo(int x)
    {
        Console.WriteLine("Foo(int x)");
    }

    public void Bar<T>(T obj)
    {
        Foo(obj);
    }
}

```

代码段 Specialization/Program.cs

`Foo()` 方法现在可以通过任意参数类型来调用。下面的示例代码给该方法传递了一个 `int` 和一个 `string`：



```
static void Main()
{
    var test = new MethodOverloads();
    test.Foo(33);
    test.Foo("abc");
}
```

代码段 GenericMethods/Program.cs

运行该程序，可以从输出中看出选择了最佳匹配的方法：

```
Foo(int x)
Foo<T>(T obj), obj type: String
```

需要注意的是，所调用的方法是在编译期间定义的，而不是运行期间。这很容易举例说明：添加一个调用 `Foo()` 方法的 `Bar()` 泛型方法，并传递泛型参数值：

```
public class MethodOverloads
{
    // ...

    public void Bar<T>(T obj)
    {
        Foo(obj);
    }
}
```

`Main()` 方法现在改为调用传递一个 `int` 值的 `Bar()` 方法：

```
static void Main()
{
    var test = new MethodOverloads();
    test.Bar(44);
}
```

从控制台的输出可以看出，`Bar()` 方法选择了泛型 `Foo()` 方法，而不是用 `int` 参数重载的 `Foo()` 方法。原因是编译器是在编译期间选择 `Bar()` 方法调用的 `Foo()` 方法。由于 `Bar()` 方法定义了一个泛型参数，而且泛型 `Foo()` 方法匹配这个类型，所以调用了 `Foo()` 方法。在运行期间给 `Bar()` 方法传递一个 `int` 值不会改变这一点。

```
Foo<T>(T obj), obj type: Int32
```

5.7 小结

本章介绍了 CLR 中一个非常重要的特性：泛型。通过泛型类可以创建独立于类型的类，泛型方法是独立于类型的方法。接口、结构和委托也可以用泛型的方式创建。泛型引入了一种新的编程方式。我们介绍了如何实现相应的算法(尤其是操作和谓词)以用于不同的类，而且它们都是类型安全的。泛型委托可以去除集合中的算法。

本书还将探讨泛型的更多特性和用法。第 8 章介绍了常常实现为泛型的委托，第 10 章论述了泛型集合类，第 11 章讨论了泛型扩展方法。

下一章说明如何对于数组使用泛型方法。

第 6 章

数 组

本章内容:

- 简单数组
- 多维数组
- 锯齿数组
- Array 类
- 作为参数的数组
- 枚举
- 元组
- 结构比较

如果需要使用同一类型的多个对象,就可以使用集合(参见第 10 章)和数组。C#用特殊的记号声明、初始化和使用数组。Array 类在后台发挥作用,它为数组中元素的排序和过滤提供了几个方法。使用枚举器,可以迭代数组中的所有元素。

.NET 4 还引入了一个新类型 Tuple,它用于合并不同类型的多个对象。详细内容参见 6.7 节。

6.1 简单数组

如果需要使用同一类型的多个对象,就可以使用数组。数组是一种数据结构,它可以包含同一类型的多个元素。

6.1.1 数组的声明

在声明数组时,应先定义数组中元素的类型,其后是一对空方括号和一个变量名。例如,下面声明了一个包含整型元素的数组:

```
int[] myArray;
```

6.1.2 数组的初始化

声明了数组后,就必须为数组分配内存,以保存数组的所有元素。数组是引用类型,所以必须给它分配堆上的内存。为此,应使用 new 运算符,指定数组中元素的类型和数量来初始化数组的变量。下面

指定了数组的大小。

```
myArray = new int[4];
```



值类型和引用类型请参见第 3 章。

在声明和初始化数组后，变量 `myArray` 就引用了 4 个整型值，它们位于托管堆上，如图 6-1 所示。

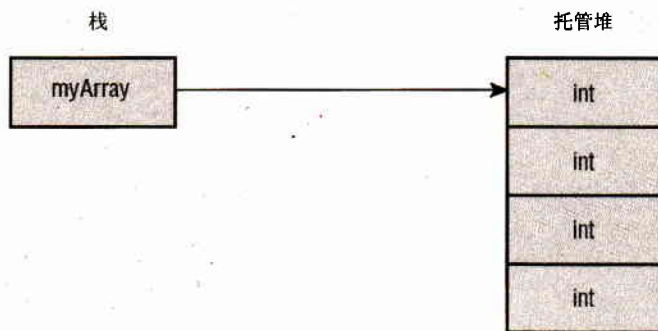


图 6-1



在指定了数组的大小后，如果不复制数组中的所有元素，就不能重新设置数组的大小。如果事先不知道数组中应包含多少个元素，就可以使用集合。集合请参见第 10 章。

除了两个语句中声明和初始化数组之外，还可以在一个语句中声明和初始化数组：

```
int[] myArray = new int[4];
```

还可以使用数组初始化为数组的每个元素赋值。数组初始化器只能在声明数组变量时使用，不能在声明数组之后使用。

```
int[] myArray = new int[4] {4, 7, 11, 2};
```

如果用花括号初始化数组，则还可以不指定数组的大小，因为编译器会自动统计元素的个数：

```
int[] myArray = new int[] {4, 7, 11, 2};
```

使用 C# 编译器还有一种更简化的形式。使用花括号可以同时声明和初始化数组，编译器生成的代码与前面的例子相同：

```
int[] myArray = {4, 7, 11, 2};
```

6.1.3 访问数组元素

在声明和初始化数组后，就可以使用索引器访问其中的元素了。数组只支持有整型参数的索引器。

通过索引器传递元素编号, 就可以访问数组。索引器总是以 0 开头, 表示第一个元素。可以传递给索引器的最大值是元素个数减 1, 因为索引从 0 开始。在下面的例子中, 数组 `myArray` 用 4 个整型值声明和初始化。用索引器对应的值 0、1、2 和 3 就可以访问该数组中的元素。

```
int[] myArray = new int[] {4, 7, 11, 2};
int v1 = myArray[0]; // read first element
int v2 = myArray[1]; // read second element
myArray[3] = 44;     // change fourth element
```



如果使用错误的索引器值(其中不存在对应的元素), 就会抛出 `IndexOutOfRangeException` 类型的异常。

如果不知道数组中的元素个数, 则可以在 `for` 语句中使用 `Length` 属性:

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

除了使用 `for` 语句迭代数组中的所有元素之外, 还可以使用 `foreach` 语句:

```
foreach (var val in myArray)
{
    Console.WriteLine(val);
}
```



`foreach` 语句利用了本章后面讨论的 `IEnumerable` 和 `IEnumerator` 接口。

6.1.4 使用引用类型

除了能声明预定义类型的数组, 还可以声明自定义类型的数组。下面用 `Person` 类来说明, 这个类有自动实现的属性 `Firstname` 和 `Lastname`, 以及从 `Object` 类重写的 `ToString()` 方法:



可从
wrox.com
下载源代码

```
public class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }

    public override string ToString()
    {
        return String.Format("{0} {1}", FirstName, LastName);
    }
}
```

代码段 Sample1/Person.cs

声明一个包含两个 `Person` 元素的数组与声明一个 `int` 数组类似:

```
Person[] myPersons = new Person[2];
```

但是必须注意,如果数组中的元素是引用类型,就必须为每个数组元素分配内存。若使用了数组中未分配内存的元素,就会抛出 `NullReferenceException` 类型的异常。



第 15 章介绍了错误和异常的详细内容。

使用从 0 开始的索引器,可以为数组的每个元素分配内存:

```
myPersons[0] = new Person { FirstName="Ayrton", LastName="Senna" };
myPersons[1] = new Person { FirstName="Michael", LastName="Schumacher" };
```

图 6-2 显示了 `Person` 数组中的对象在托管堆中的情况。`myPersons` 是存储在栈上的一个变量,该变量引用了存储在托管堆上的 `Person` 元素对应的数组。这个数组有足够容纳两个引用的空间。数组中的每一项都引用了一个 `Person` 对象,而这些 `Person` 对象也存储在托管堆上。

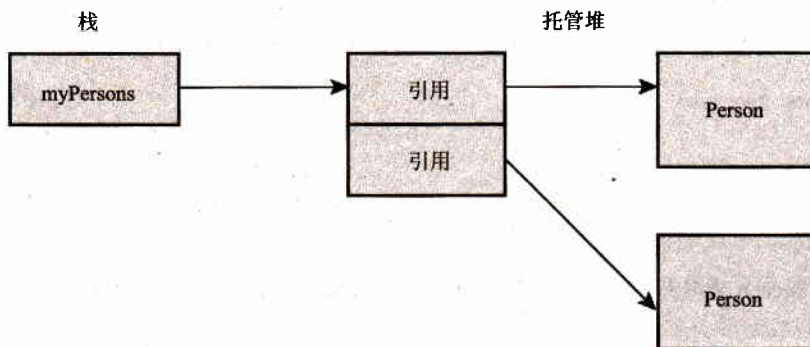


图 6-2

与 `int` 类型一样,也可以对自定义类型使用数组初始化器:

```
Person[] myPersons2 =
{
    new Person { FirstName="Ayrton", LastName="Senna"},
    new Person { FirstName="Michael", LastName="Schumacher"}
};
```

6.2 多维数组

一般数组(也称为一维数组)用一个整数来索引。多维数组用两个或多个整数来索引。

图 6-3 是二维数组的数学表示法,该数组有 3 行 3 列。第 1 行的值是 1、2 和 3,第 3 行的值是 7、8 和 9。

$$a = \begin{bmatrix} 1, & 2, & 3 \\ 4, & 5, & 6 \\ 7, & 8, & 9 \end{bmatrix}$$

在 C# 中声明这个二维数组,需要在方括号中加上一个逗号。数组在初始化时应

图 6-3

指定每一维的大小(也称为阶)。接着,就可以使用两个整数作为索引器来访问数组中的元素:



可从
wrox.com
下载源代码

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

代码段 Sample1/Program.cs



声明数组之后,就不能修改其阶数了。

如果事先知道元素的值,则也可以使用数组索引器来初始化二维数组。在初始化数组时,使用一个外层的花括号,每一行用包含在外层花括号中的内层花括号来初始化。

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



使用数组初始化器时,必须初始化数组的每个元素,不能遗漏任何元素。

在花括号中使用两个逗号,就可以声明一个三维数组:

```
int[, ,] threedim = {
    { { 1, 2 }, { 3, 4 } },
    { { 5, 6 }, { 7, 8 } },
    { { 9, 10 }, { 11, 12 } }
};

Console.WriteLine(threedim[0, 1, 1]);
```

6.3 锯齿数组

二维数组的大小对应于一个矩形,如对应的元素个数为 3×3 。而锯齿数组的大小设置比较灵活,在锯齿数组中,每一行都可以有不同的尺寸。

图 6-4 比较了有 3×3 个元素的二维数组和锯齿数组。图 6-4 中的锯齿数组有 3 行,第 1 行有两个元

素, 第 2 行有 6 个元素, 第 3 行有 3 个元素。



图 6-4

在声明锯齿数组时, 要依次放置左右括号。在初始化锯齿数组时, 只在第 1 对方括号中设置该数组包含的行数。定义各行中元素个数的第 2 个方括号设置为空, 因为这类数组的每一行包含不同的元素个数。之后, 为每一行指定行中的元素个数:



可从
wrox.com
下载源代码

```
int[][] jagged = new int[3][];
jagged[0] = new int[2] { 1, 2 };
jagged[1] = new int[6] { 3, 4, 5, 6, 7, 8 };
jagged[2] = new int[3] { 9, 10, 11 };
```

代码段 Sample1/Program.cs

迭代锯齿数组中所有元素的代码可以放在嵌套的 for 循环中。在外层的 for 循环中迭代每一行, 在内层的 for 循环中迭代一行中的每个元素:

```
for (int row = 0; row < jagged.Length; row++)
{
    for (int element = 0; element < jagged[row].Length; element++)
    {
        Console.WriteLine("row: {0}, element: {1}, value: {2}",
            row, element, jagged[row][element]);
    }
}
```

该迭代结果显示了所有的行和每一行中的各个元素:

```
row: 0, element: 0, value: 1
row: 0, element: 1, value: 2
row: 1, element: 0, value: 3
row: 1, element: 1, value: 4
row: 1, element: 2, value: 5
row: 1, element: 3, value: 6
row: 1, element: 4, value: 7
row: 1, element: 5, value: 8
row: 2, element: 0, value: 9
row: 2, element: 1, value: 10
row: 2, element: 2, value: 11
```

6.4 Array 类

用方括号声明数组是 C# 中使用 Array 类的表示法。在后台使用 C# 语法, 会创建一个派生自抽象基

类 `Array` 的新类。这样, 就可以使用 `Array` 类为每个 C# 数组定义的方法和属性了。例如, 前面就使用了 `Length` 属性, 或者使用 `foreach` 语句迭代数组。其实这是使用了 `Array` 类中的 `GetEnumerator()` 方法。

`Array` 类实现的其他属性有 `LongLength` 和 `Rank`。如果数组包含的元素个数超出了整数的取值范围, 就可以使用 `LongLength` 属性来获得元素个数。使用 `Rank` 属性可以获得数组的维数。

下面通过了解不同的功能来看看 `Array` 类的其他成员。

6.4.1 创建数组

`Array` 类是一个抽象类, 所以不能使用构造函数来创建数组。但除了可以使用 C# 语法创建数组实例之外, 还可以使用静态方法 `CreateInstance()` 创建数组。如果事先不知道元素的类型, 该静态方法就非常有用, 因为类型可以作为 `Type` 对象传递给 `CreateInstance()` 方法。

下面的例子说明了如何创建类型为 `int`、大小为 5 的数组。`CreateInstance()` 方法的第 1 个参数应是元素的类型, 第 2 个参数定义数组的大小。可以用 `SetValue()` 方法设置对应元素的值, 用 `GetValue()` 方法读取对应元素的值:



可从
wrox.com
下载源代码

```
Array intArray1 = Array.CreateInstance(typeof(int), 5);
for (int i = 0; i < 5; i++)
{
    intArray1.SetValue(33, i);
}

for (int i = 0; i < 5; i++)
{
    Console.WriteLine(intArray1.GetValue(i));
}
```

代码段 Sample1/Program.cs

还可以将已创建的数组强制转换成声明为 `int[]` 的数组:

```
int[] intArray2 = (int[])intArray1;
```

`CreateInstance()` 方法有许多重载版本, 可以创建多维数组和不基于 0 的数组。下面的例子就创建了一个包含 2×3 个元素的二维数组。第一维基于 1, 第二维基于 10:

```
int[] lengths = { 2, 3 };
int[] lowerBounds = { 1, 10 };
Array racers = Array.CreateInstance(typeof(Person), lengths, lowerBounds);
```

`SetValue()` 方法设置数组的元素, 其参数是每一维的索引:



可从
wrox.com
下载源代码

```
racers.SetValue(new Person
{
    FirstName = "Alain",
    LastName = "Prost"
}, 1, 10);
racers.SetValue(new Person
{
    // ...
}
```

```

        FirstName = "Emerson",
        LastName = "Fittipaldi"
    }, 1, 11);
racers.SetValue(new Person
{
    FirstName = "Ayrton",
    LastName = "Senna"
}, 1, 12);
racers.SetValue(new Person
{
    FirstName = "Ralf",
    LastName = "Schumacher"
}, 2, 10);
racers.SetValue(new Person
{
    FirstName = "Fernando",
    LastName = "Alonso"
}, 2, 11);
racers.SetValue(new Person
{
    FirstName = "Jenson",
    LastName = "Button"
}, 2, 12);

```

代码段 Sample1/Program.cs

尽管数组不是基于 0，但可以用一般的 C# 表示法将它赋予一个变量。只需注意不要超出边界即可：

```

Person[, ] racers2 = (Person[, ]) racers;
Person first = racers2[1, 10];
Person last = racers2[2, 12];

```

6.4.2 复制数组

因为数组是引用类型，所以将一个数组变量赋予另一个数组变量，就会得到两个引用同一数组的变量。而复制数组，会使数组实现 `ICloneable` 接口。这个接口定义的 `Clone()` 方法会创建数组的浅表副本。

如果数组的元素是值类型，以下代码段就会复制所有值，如图 6-5 所示：

```

int[] intArray1 = {1, 2};
int[] intArray2 = (int[])intArray1.Clone();

```

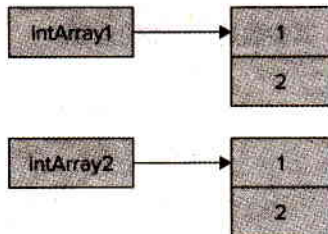


图 6-5

如果数组包含引用类型，则不复制元素，而只复制引用。图 6-6 显示了变量 `beatles` 和 `beatlesClone`，其中 `beatlesClone` 通过从 `beatles` 中调用 `Clone()` 方法来创建。`beatles` 和 `beatlesClone` 引用的 `Person` 对象是相同的。如果修改 `beatlesClone` 中一个元素的属性，就会改变 `beatles` 中的对应对象。



可从
wrox.com
下载源代码

```
Person[] beatles = {
    new Person { FirstName="John", LastName="Lennon" },
    new Person { FirstName="Paul", LastName="McCartney" }
};
Person[] beatlesClone = (Person[])beatles.Clone();
```

代码段 Sample1/Program.cs

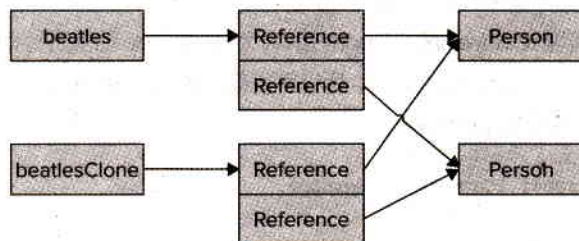


图 6-6

除了使用 `Clone()` 方法之外，还可以使用 `Array.Copy()` 方法创建浅表副本。但 `Clone()` 方法和 `Copy()` 方法有一个重要区别：`Clone()` 方法会创建一个新数组，而 `Copy()` 方法必须传递阶数相同且有足够元素的已有数组。



如果需要包含引用类型的数组的深层副本，就必须迭代数组并创建新对象。

6.4.3 排序

`Array` 类使用 `QuickSort` 算法对数组中元素进行排序。`Sort()` 方法需要数组中的元素实现 `Comparable` 接口。因为简单类型(如 `System.String` 和 `System.Int32`)实现 `Comparable` 接口，所以可以对包含这些类型的元素排序。

在示例程序中，数组 `names` 包含 `string` 类型的元素，这个数组可以排序。



可从
wrox.com
下载源代码

```
string[] names = {
    "Christina Aguilera",
    "Shakira",
    "Beyonce",
    "Gwen Stefani"
};

Array.Sort(names);

foreach (var name in names)
{
    Console.WriteLine(name);
}
```

代码段 SortingSample/Program.cs

该应用程序的输出是排好序的数组:

```
Beyonce
Christina Aguilera
Gwen Stefani
Shakira
```

如果对数组使用自定义类,就必须实现 `Comparable` 接口。这个接口只定义了一个方法 `CompareTo()`,如果要比较的对象相等,该方法就返回 0。如果该实例应排在参数对象的前面,该方法就返回小于 0 的值。如果该实例应排在参数对象的后面,该方法就返回大于 0 的值。

修改 `Person` 类,使之实现 `Comparable<Person>` 接口。对 `LastName` 的值进行比较。`LastName` 是 `string` 类型,而 `String` 类已经实现了 `Comparable` 接口,所以可以使用 `String` 类中 `CompareTo()` 方法的实现代码。如果 `LastName` 的值相同,就比较 `FirstName`:



```
public class Person: Comparable<Person>
{
    public int CompareTo(Person other)
    {
        if (other == null) throw new ArgumentNullException("other");

        int result = this.LastName.CompareTo(
            other.LastName);
        if (result == 0)
        {
            result = this.FirstName.CompareTo(
                other.FirstName);
        }
        return result;
    }
    //...
```

代码段 SortingSample/Person.cs

现在可以按照姓氏对 `Person` 对象对应的数组排序:



```
Person[] persons = {
    new Person { FirstName="Damon", LastName="Hill" },
    new Person { FirstName="Niki", LastName="Lauda" },
    new Person { FirstName="Ayrton", LastName="Senna" },
    new Person { FirstName="Graham", LastName="Hill" }
};

Array.Sort(persons);
foreach (var p in persons)
{
    Console.WriteLine(p);
}
```

代码段 SortingSample/Program.cs

使用 `Person` 类的排序功能，会得到按姓氏排序的姓名：

```
Damon Hill
Graham Hill
Niki Lauda
Ayrton Senna
```

如果 `Person` 对象的排序方式与上述不同，或者不能修改在数组中用作元素的类，就可以实现 `IComparer` 接口或 `IComparer<T>` 接口。这两个接口定义了方法 `Compare()`。要比较的类必须实现这两个接口之一。`IComparer` 接口独立于要比较的类。这就是 `Compare()` 方法定义了两个要比较的参数原因。其返回值与 `IComparable` 接口的 `CompareTo()` 方法类似。

类 `PersonComparer` 实现了 `IComparer<Person>` 接口，可以按照 `firstName` 或 `lastName` 对 `Person` 对象排序。枚举 `PersonCompareType` 定义了可用于 `PersonComparer` 的排序选项：`FirstName` 和 `LastName`。排序方式由 `PersonComparer` 类的构造函数定义，在该构造函数中设置了一个 `PersonCompareType` 值。实现 `Compare()` 方法时用一个 `switch` 语句指定是按 `FirstName` 还是 `LastName` 排序。



可从
wrox.com
下载源代码

```
public enum PersonCompareType
{
    FirstName,
    LastName
}

public class PersonComparer: IComparer<Person>
{
    private PersonCompareType compareType;

    public PersonComparer(PersonCompareType compareType)
    {
        this.compareType = compareType;
    }

    public int Compare(Person x, Person y)
    {
        if (x == null) throw new ArgumentNullException("x");
        if (y == null) throw new ArgumentNullException("y");

        switch (compareType)
        {
            case PersonCompareType.FirstName:
                return x.FirstName.CompareTo(y.FirstName);
            case PersonCompareType.LastName:
                return x.LastName.CompareTo(y.LastName);
            default:
                throw new ArgumentException(
                    "unexpected compare type");
        }
    }
}
```

代码段 `SortingSample/PersonComparer.cs`

现在, 可以将一个 `PersonComparer` 对象传递给 `Array.Sort()` 方法的第 2 个参数。下面按名字对 `persons` 数组排序:



可从
wrox.com
下载源代码

```
Array.Sort (persons,
            new PersonComparer (PersonCompareType.FirstName));
foreach (var p in persons)
{
    Console.WriteLine(p);
}
```

代码段 `SortingSample/Program.cs`

`persons` 数组现在按名字排序:

```
Ayrton Senna
Damon Hill
Graham Hill
Niki Lauda
```



`Array` 类还提供了 `Sort` 方法, 它需要将一个委托作为参数。这个参数可以传递给方法, 从而比较两个对象, 而不需要依赖 `IComparable` 或 `IComparer` 接口。第 8 章将介绍如何使用委托。

6.5 数组作为参数

数组可以作为参数传递给方法, 也可以从方法中返回。要返回一个数组, 只需把数组声明为返回类型, 如下面的方法 `GetPerson()` 所示:



可从
wrox.com
下载源代码

```
static Person[] GetPersons()
{
    return new Person[] {
        new Person { FirstName="Damon", LastName="Hill" },
        new Person { FirstName="Niki", LastName="Lauda" },
        new Person { FirstName="Ayrton", LastName="Senna" },
        new Person { FirstName="Graham", LastName="Hill" }
    };
}
```

代码段 `SortingSample/Program.cs`

要把数组传递给方法, 应把数组声明为参数, 如下面的 `DisplayPerson()` 方法所示:

```
static void DisplayPersons(Person[] persons)
{
    //...
```

6.5.1 数组协变

数组支持协变。这表示数组可以声明为基类，其派生类型的元素可以赋予数组元素。

例如，可以声明一个 `object[]` 类型的参数，给它传递一个 `Person[]`：

```
static void DisplayArray(object[] data)
{
    //...
}
```



数组协变只能用于引用类型，不能用于值类型。

数组协变有一个问题，它只能通过运行时异常来解决。如果把 `Person` 数组赋予 `object` 数组，`object` 数组就可以使用派生自 `object` 的任何元素。例如，编译器允许把字符串传递给数组元素。但因为 `object` 数组引用 `Person` 数组，所以会出现一个运行时异常。

6.5.2 ArraySegment<T>

结构 `ArraySegment<T>` 表示数组的一段。如果某方法应返回数组的一部分，或者给某方法传递数组的一部分，就可以使用数组段。通过 `ArraySegment<T>` 可以只传递一个参数，而不是用 3 个参数传递数组、偏移量和元素个数。在这个结构中，关于数组段的信息(偏移量和元素个数)包含在结构的成员中。

`SumOfSegments()` 方法提取一组 `ArraySegment<int>` 元素，计算该数组段定义的所有整数之和，并返回整数和：



可从
wrox.com
下载源代码

```
static int SumOfSegments(ArraySegment<int>[] segments)
{
    int sum = 0;
    foreach (var segment in segments)
    {
        for (int i = segment.Offset; i < segment.Offset +
            segment.Count; i++)
        {
            sum += segment.Array[i];
        }
    }
    return sum;
}
```

代码段 `ArraySegmentSample/Program.cs`

使用这个方法时，传递了一个数组段。第一个数组元素从 `ar1` 的第一个元素开始，引用了 3 个元素；第二个数组元素从 `ar2` 的第 4 个元素开始，引用了 3 个元素；

```
int[] ar1 = { 1, 4, 5, 11, 13, 18 };
int[] ar2 = { 3, 4, 5, 18, 21, 27, 33 };
```

```
var segments = new ArraySegment<int> [2]
{
    new ArraySegment<int> (ar1, 0, 3),
    new ArraySegment<int> (ar2, 3, 3)
};
var sum = SumOfSegments(segments);
```



数组段不复制原数组的元素，但原数组可以通过 `ArraySegment<T>` 访问。如果数组段中的元素改变了，这些变化就会反映到原数组中。

6.6 枚举

在 `foreach` 语句中使用枚举，可以迭代集合(参见第 10 章)中的元素，且无需知道集合中的元素个数。`foreach` 语句使用了一个枚举器。图 6-7 显示了调用 `foreach` 方法的客户端和集合之间的关系。数组或集合实现带 `GetEnumerator()` 方法的 `IEnumerable` 接口。`GetEnumerator()` 方法返回一个实现 `IEnumerator` 接口的枚举。接着，`foreach` 语句就可以使用 `IEnumerator` 接口迭代集合了。

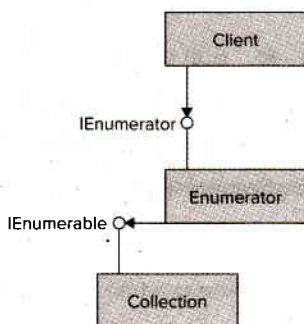


图 6-7



`GetEnumerator()` 方法用 `IEnumerator` 接口定义。`foreach` 语句并不真的需要在集合类中实现这个接口。有一个名为 `GetEnumerator()` 的方法，它返回实现了 `IEnumerator` 接口的对象就足够了。

6.6.1 IEnumerator 接口

`foreach` 语句使用 `IEnumerator` 接口的方法和属性，迭代集合中的所有元素。为此，`IEnumerator` 定义了 `Current` 属性，来返回光标所在的元素，该接口的 `MoveNext()` 方法移动到集合的下一个元素上，如果有这个元素，该方法就返回 `true`。如果集合不再有更多的元素，该方法就返回 `false`。

这个接口的泛型版本 `IEnumerator<T>` 派生自接口 `IDisposable`，因此定义了 `Dispose()` 方法，来清理枚举器占用的资源。



IEnumerator 接口还定义了 **Reset()** 方法, 以与 **COM** 交互操作。许多 **.NET** 枚举器通过抛出 **NotSupportedException** 类型的异常, 来实现这个方法。

6.6.2 foreach 语句

C# 的 **foreach** 语句不会解析为 **IL** 代码中的 **foreach** 语句。**C#** 编译器会把 **foreach** 语句转换为 **IEnumerator** 接口的方法和属性。下面是一条简单的 **foreach** 语句, 它迭代 **persons** 数组中的所有元素, 并逐个显示它们:

```
foreach (var p in persons)
{
    Console.WriteLine(p);
}
```

foreach 语句会解析为下面的代码段。首先, 调用 **GetEnumerator()** 方法, 获得数组的一个枚举器。在 **while** 循环中——只要 **MoveNext()** 返回 **true**——就用 **Current** 属性访问数组中的元素:

```
IEnumerator<Person> enumerator = persons.GetEnumerator();
while (enumerator.MoveNext())
{
    Person p = enumerator.Current;
    Console.WriteLine(p);
}
```

6.6.3 yield 语句

自 **C#** 的第 1 个版本以来, 使用 **foreach** 语句可以轻松地迭代集合。在 **C# 1.0** 中, 创建枚举器仍需要做大量的工作。**C# 2.0** 添加了 **yield** 语句, 以便于创建枚举器。**yield return** 语句返回集合的一个元素, 并移动到下一个元素上。**yield break** 可停止迭代。

下一个例子是用 **yield return** 语句实现一个简单集合的代码。**HelloCollection** 类包含 **GetEnumerator()** 方法。该方法的实现代码包含两条 **yield return** 语句, 它们分别返回字符串 **Hello** 和 **World**。



可从
wrox.com
下载源代码

```
using System;
using System.Collections;

namespace Wrox.ProCSharp.Arrays
{
    public class HelloCollection
    {
        public IEnumerator<string> GetEnumerator()
        {
            yield return "Hello";
            yield return "World";
        }
    }
}
```

代码段 **YieldDemo/Program.cs**



包含 `yield` 语句的方法或属性也称为迭代块。迭代块必须声明为返回 `IEnumerator` 或 `IEnumerable` 接口, 或者这些接口的泛型版本。这个块可以包含多条 `yield return` 语句或 `yield break` 语句, 但不能包含 `return` 语句。

现在可以用 `foreach` 语句迭代集合了:

```
public void HelloWorld()
{
    var helloCollection = new HelloCollection();
    foreach (var s in helloCollection)
    {
        Console.WriteLine(s);
    }
}
```

使用迭代块, 编译器会生成一个 `yield` 类型, 其中包含一个状态机, 如下面的代码段所示。`yield` 类型实现 `IEnumerator` 和 `IDisposable` 接口的属性和方法。在下面的例子中, 可以把 `yield` 类型看作内部类 `Enumerator`。外部类的 `GetEnumerator()` 方法实例化并返回一个新的 `yield` 类型。在 `yield` 类型中, 变量 `state` 定义了迭代的当前位置, 每次调用 `MoveNext()` 时, 当前位置都会改变。`MoveNext()` 封装了迭代块的代码, 并设置了 `current` 变量的值, 从而使 `Current` 属性根据位置返回一个对象。

```
public class HelloCollection
{
    public IEnumerator GetEnumerator()
    {
        return new Enumerator(0);
    }

    public class Enumerator: IEnumerator<string>, IEnumerator, IDisposable
    {
        private int state;
        private string current;

        public Enumerator(int state)
        {
            this.state = state;
        }

        bool System.Collections.IEnumerator.MoveNext()
        {
            switch (state)
            {
                case 0:
                    current = "Hello";
                    state = 1;
                    return true;
                case 1:

```

```

        current = "World";
        state = 2;
        return true;
    case 2:
        break;
    }

    return false;
}

void System.Collections.IEnumerator.Reset()
{
    throw new NotSupportedException();
}

string System.Collections.Generic.IEnumerator < string > .Current
{
    get
    {
        return current;
    }
}

object System.Collections.IEnumerator.Current
{
    get
    {
        return current;
    }
}

void IDisposable.Dispose()
{
}
}
}

```



yield 语句会生成一个枚举器，而不仅仅生成一个包含的项的列表。这个枚举器通过 foreach 语句调用。从 foreach 中依次访问每一项时，就会访问枚举器。这样就可以迭代大量的数据，而无需一次把所有的数据都读入内存。

1. 迭代集合的不同方式

在下面这个比 Hello World 示例略大但比较真实的示例中，可以使用 yield return 语句，以不同方式迭代集合的类。类 MusicTitles 可以用默认方式通过 GetEnumerator() 方法迭代标题，用 Reverse() 方法逆序迭代标题，用 Subset() 方法迭代子集：



可从
wrox.com
下载源代码

```
public class MusicTitles
{
    string[] names = { "Tubular Bells", "Hergest Ridge", "Ommadawn",
                      "Platinum" };

    public IEnumerator<string> GetEnumerator()
    {
        for (int i = 0; i < 4; i++)
        {
            yield return names[i];
        }
    }

    public IEnumerable<string> Reverse()
    {
        for (int i = 3; i >= 0; i -- )
        {
            yield return names[i];
        }
    }

    public IEnumerable<string> Subset(int index, int length)
    {
        for (int i = index; i < index + length; i++)
        {
            yield return names[i];
        }
    }
}
```

代码段 YieldDemo/Program.cs



类支持的默认迭代是定义为返回 `IEnumerator` 的 `GetEnumerator()` 方法。命名的迭代返回 `IEnumerable`。

迭代字符串数组的客户端代码先使用 `GetEnumerator()` 方法，该方法不必在代码中编写，因为这是默认使用的方法。然后逆序迭代标题，最后将索引和要迭代的项数传递给 `Subset()` 方法，来迭代子集：

```
var titles = new MusicTitles();
foreach (var title in titles)
{
    Console.WriteLine(title);
}
Console.WriteLine();

Console.WriteLine("reverse");
foreach (var title in titles.Reverse())
{
    Console.WriteLine(title);
}
```

```

Console.WriteLine();

Console.WriteLine("subset");
foreach (var title in titles.Subset(2, 2))
{
    Console.WriteLine(title);
}

```

2. 用 yield return 返回枚举器

使用 yield 语句还可以完成更复杂的任务，例如，从 yield return 中返回枚举器。

在 TicTacToe 游戏中有 9 个域，玩家轮流在这些域中放置一个“十”字或一个圆。这些移动操作由 GameMoves 类模拟。方法 Cross() 和 Circle() 是创建迭代类型的迭代块。变量 cross 和 circle 在 GameMoves 类的构造函数中设置为 Cross() 和 Circle() 方法。这些字段不设置为调用的方法，而是设置为用迭代块定义的迭代类型。在 Cross() 迭代块中，将移动操作的信息写到控制台上，并递增移动次数。如果移动次数大于 8，就用 yield break 停止迭代；否则，就在每次迭代中返回 yield 类型 circle 的枚举对象。Circle() 迭代块非常类似于 Cross() 迭代块，只是它在每次迭代中返回 cross 迭代器类型。



可从
wrox.com
下载源代码

```

public class GameMoves
{
    private IEnumerator cross;
    private IEnumerator circle;

    public GameMoves()
    {
        cross = Cross();
        circle = Circle();
    }

    private int move = 0;
    const int MaxMoves = 9;

    public IEnumerator Cross()
    {
        while (true)
        {
            Console.WriteLine("Cross, move {0}", move);
            if (++move >= MaxMoves)
                yield break;
            yield return circle;
        }
    }

    public IEnumerator Circle()
    {
        while (true)
        {
            Console.WriteLine("Circle, move {0}", move);
            if (++move >= MaxMoves)
                yield break;
            yield return cross;
        }
    }
}

```

代码段 YieldDemo/GameMoves.cs

在客户端程序中，可以以如下方式使用 `GameMoves` 类。将枚举器设置为由 `game.Cross()` 返回的枚举器类型，以设置第一次移动。在 `while` 循环中，调用 `enumerator.MoveNext()`。第一次调用 `enumerator.MoveNext()` 时，会调用 `Cross()` 方法，`Cross()` 方法使用 `yield` 语句返回另一个枚举器。返回的值可以用 `Current` 属性访问，并设置为 `enumerator` 变量，用于下一次循环：



可从
wrox.com
下载源代码

```
var game = new GameMoves();
IEnumerator enumerator = game.Cross();
while (enumerator.MoveNext())
{
    enumerator = enumerator.Current as IEnumerator;
}
```

代码段 YieldDemo/Program.cs

这个程序的输出会显示交替移动的情况，直到最后一次移动：

```
Cross, move 0
Circle, move 1
Cross, move 2
Circle, move 3
Cross, move 4
Circle, move 5
Cross, move 6
Circle, move 7
Cross, move 8
```

6.7 元组

数组合并了相同类型的对象，而元组合并了不同类型的对象。元组起源于函数编程语言(如 F#)，在这些语言中频繁使用元组。在 .NET 4 中，元组可通过 `NET Framework` 用于所有的 .NET 语言。

.NET 4 定义了 8 个泛型 `Tuple` 类和一个静态 `Tuple` 类，它们用作元组的工厂。这里的不同泛型 `Tuple` 类支持不同数量的元素。例如，`Tuple<T1>` 包含一个元素，`Tuple<T1, T2>` 包含两个元素，以此类推。

方法 `Divide()` 返回包含两个成员的元组 `Tuple<int, int>`。泛型类的参数定义了成员的类型，它们都是整数。元组用静态 `Tuple` 类的静态 `Create()` 方法创建。`Create()` 方法的泛型参数定义了要实例化的元组类型。新建的元组用 `result` 和 `remainder` 变量初始化，返回这两个变量相除的结果：



可从
wrox.com
下载源代码

```
public static Tuple<int, int> Divide(int dividend, int divisor)
{
    int result = dividend / divisor;
    int remainder = dividend % divisor;

    return Tuple.Create<int, int> (result, remainder);
}
```

下面的代码说明了 `Divide()` 方法的调用。可以用属性 `Item1` 和 `Item2` 访问元组的项:

```
var result = Divide(5, 2);
Console.WriteLine("result of division: {0}, reminder: {1}",
    result.Item1, result.Item2);
```

如果元组包含的项超过 8 个, 就可以使用带 8 个参数的 `Tuple` 类定义。最后一个模板参数是 `TRest`, 表示必须给它传递一个元组。这样, 就可以创建带任意个参数的元组了。

下面说明这个功能:

```
public class Tuple<T1, T2, T3, T4, T5, T6, T7, TRest>
```

其中, 最后一个模板参数是一个元组类型, 所以可以创建带任意多项的元组:

```
var tuple = Tuple.Create<string, string, string, int, int, int, double,
    Tuple<int, int>>>(
    "Stephanie", "Alina", "Nagel", 2009, 6, 2, 1.37,
    Tuple.Create<int, int>(52, 3490));
```

6.8 结构比较

数组和元组都实现接口 `IStructuralEquatable` 和 `IStructuralComparable`。这两个接口都是 .NET 4 新增的, 不仅可以比较引用, 还可以比较内容。这些接口都是显式实现的, 所以在使用时需要把数组和元组强制转换为这个接口。`IStructuralEquatable` 接口用于比较两个元组或数组是否有相同的内容, `IStructuralComparable` 接口用于给元组或数组排序。

对于说明 `IStructuralEquatable` 接口的示例, 使用实现 `IEquatable` 接口的 `Person` 类。`IEquatable` 接口定义了一个强类型化的 `Equals()` 方法, 以比较 `FirstName` 和 `LastName` 属性的值:



可从
wrox.com
下载源代码

```
public class Person: IEquatable<Person>
{
    public int Id { get; private set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }

    public override string ToString()
    {
        return String.Format("{0}, {1} {2}", Id, FirstName, LastName);
    }

    public override bool Equals(object obj)
    {
        if (obj == null) throw new ArgumentNullException("obj");
        return Equals(obj as Person);
    }

    public override int GetHashCode()
```

```

    {
        return Id.GetHashCode();
    }

    public bool Equals(Person other)
    {
        if (other == null) throw new ArgumentNullException("other");

        return this.Id == other.Id && this.FirstName == other.FirstName &&
            this.LastName == other.LastName;
    }
}

```

代码段 StructuralComparison/Person.cs

现在创建了两个包含 `Person` 项的数组。这两个数组通过变量名 `janet` 包含相同的 `Person` 对象，和两个内容相同的不同 `Person` 对象。比较运算符 “`!=`” 返回 `true`，因为这其实是两个变量 `persons1` 和 `persons2` 引用的两个不同数组。因为 `Array` 类没有重写带一个参数的 `Equals()` 方法，所以用 “`=`” 运算符比较引用也会得到相同的结果，即这两个变量不相同：



可从
wrox.com
下载源代码

```

var janet = new Person { FirstName = "Janet", LastName = "Jackson" };
Person[] persons1 = {
    new Person
    {
        FirstName = "Michael",
        LastName = "Jackson"
    },
    janet
};
Person[] persons2 = {
    new Person
    {
        FirstName = "Michael",
        LastName = "Jackson"
    },
    janet
};

if (persons1 != persons2)
    Console.WriteLine("not the same reference");

```

代码段 StructuralComparison/Program.cs

对于 `IStructuralEquatable` 接口定义的 `Equals()` 方法，它的第一个参数是 `object` 类型，第二个参数是 `IEqualityComparer` 类型。调用这个方法时，通过传递一个实现了 `IEqualityComparer<T>` 的对象，就可以定义如何进行比较。通过 `EqualityComparer<T>` 类完成 `IEqualityComparer` 的一个默认实现。这个实现检查该类型是否实现了 `IEquatable` 接口，并调用 `IEquatable.Equals()` 方法。如果该类型没有实现 `IEquatable`，就调用 `Object` 基类中的 `Equals()` 方法进行比较。

`Person` 实现 `IEquatable<Person>`，在此过程中比较对象的内容，而数组的确包含相同的内容：

```

if ((persons1 as IStructuralEquatable).Equals(persons2,

```

```

        EqualityComparer<Person>.Default))
    {
        Console.WriteLine("the same content");
    }

```

下面看看如何对元组执行相同的操作。这里创建了两个内容相同的元组实例。当然，因为引用 t1 和 t2 引用了两个不同的对象，所以比较运算符 “!=” 返回 true:

```

var t1 = Tuple.Create<int, string>(1, "Stephanie");
var t2 = Tuple.Create<int, string>(1, "Stephanie");
if (t1 != t2)
    Console.WriteLine("not the same reference to the tuple");

```

Tuple<>类提供了两个 **Equals()**方法: 一个重写了 **Object** 基类中的 **Equals()**方法, 并把 **object** 作为参数, 第二个由 **IEqualityComparer** 接口定义, 并把 **object** 和 **IEqualityComparer** 作为参数。可以给第一个方法传送另一个元组, 如下所示。这个方法使用 **EqualityComparer<object>.Default** 获取一个 **ObjectEqualityComparer<object>**, 以进行比较。这样, 就会调用 **Object.Equals()**方法比较元组的每一项。如果每一项都返回 true, **Equals()**方法的最终结果就是 true, 这里因为 **int** 和 **string** 值都相同, 所以返回 true:

```

if (t1.Equals(t2))
    Console.WriteLine("the same content");

```

还可以使用类 **TupleComparer** 创建一个自定义的 **IEqualityComparer**, 如下所示。这个类实现了 **IEqualityComparer** 接口的两个方法 **Equals()**和 **GetHashCode()**:



可从
wrox.com
下载源代码

```

class TupleComparer: IEqualityComparer
{
    public new bool Equals(object x, object y)
    {
        return x.Equals(y);
    }

    public int GetHashCode(object obj)
    {
        return obj.GetHashCode();
    }
}

```

代码段 StructuralComparison/Program.cs



实现 **IEqualityComparer** 接口的 **Equals()**方法需要 **new** 修饰符或者隐式实现的接口, 因为基类 **Object** 也定义了带两个参数的静态 **Equals()**方法。

使用 **TupleComparer**, 给 **Tuple<T1, T2>**类的 **Equals()**方法传递一个新实例。**Tuple** 类的 **Equals()**方法为要比较的每一项调用 **TupleComparer** 的 **Equals()**方法。所以, 对于 **Tuple<T1, T2>**类, 要调用两次 **TupleComparer**, 以检查所有项是否相等:

```

if (t1.Equals(t2, new TupleComparer()))

```

```
Console.WriteLine("equals using TupleComparer");
```

6.9 小结

本章介绍了创建和使用简单数组、多维数组和锯齿数组的 C# 表示法。C# 数组在后台使用 `Array` 类, 这样就可以用数组变量调用这个类的属性和方法。

我们还探讨了如何使用 `Comparable` 和 `Comparer` 接口给数组中的元素排序, 描述了如何使用和创建枚举器、`IEnumerable` 和 `IEnumerator` 接口, 以及 `yield` 语句。最后介绍了 .NET 4 的一个新特性——元组。

下一章介绍运算符和强制类型转换。

第 7 章

运算符和类型强制转换

本章内容:

- C#中的运算符
- 处理引用类型和值类型时相等的含义
- 基本数据类型之间的数据转换
- 使用装箱技术把值类型转换为引用类型
- 通过类型强制转换在引用类型之间转换
- 重载标准的运算符以支持自定义类型
- 给自定义类型添加类型强制转换运算符

前几章介绍了使用 C#编写程序所需要的大部分知识。本章将首先讨论基本语言元素,接着论述 C#语言的扩展功能。

7.1 运算符

C 和 C++开发人员应很熟悉大多数 C#运算符,这里为新程序员和 Visual Basic 开发人员介绍最重要的运算符,并揭示 C#中的一些新变化。

C#支持表 7-1 中的运算符。

表 7-1

类 别	运 算 符
算术运算符	+ - * / %
逻辑运算符	& ^ ~ && !
字符串连接运算符	+
增量和减量运算符	++ --
移位运算符	<< >>
比较运算符	= != <> <= >=
赋值运算符	= += -= *= /= %= &= = ^= <<= >>=
成员访问运算符(用于对象和结构)	.
索引运算符(用于数组和索引器)	[]
类型转换运算符	()

(续表)

类 别	运 算 符
条件运算符(三元运算符)	?:
委托连接和删除运算符(见第 8 章)	+ -
对象创建运算符	new
类型信息运算符	sizeof is typeof as
溢出异常控制运算符	checked unchecked
间接寻址运算符	[]
名称空间别名限定符(见第 2 章)	::
空合并运算符	??

如表 7-2 所示, 有 4 个运算符(sizeof、*、->和&)只能用于不安全的代码(这些代码忽略了 C#的类型安全性检查), 这些不安全的代码见第 13 章的讨论。还要注意, sizeof 运算符在早期版本的.NET Framework 1.0 和 1.1 中使用, 它需要不安全模式。自从.NET Framework 2.0 版本以来, 就不需要这个运算符了。

表 7-2

类 别	运 算 符
运算符关键字	sizeof(仅用于.NET Framework 1.0 和 1.1)
运算符	*, ->, &

使用 C#运算符的一个最大缺点是, 与 C 风格的语言一样, 对于赋值(=)和比较(==)运算 C#使用不同的运算符。例如, 下述语句表示 “x 等于 3”:

```
x = 3;
```

如果要比较 x 和另一个值, 就需要使用两个等号(==):

```
if (x == 3)
{
}
```

C#非常严格的类型安全规则防止出现常见的 C 错误, 也就是在逻辑语句中使用赋值运算符代替比较运算符。在 C#中, 下述语句会产生一个编译错误:

```
if (x = 3)
{
}
```

习惯使用与字符(&)来连接字符串的 Visual Basic 程序员必须改变这个习惯。在 C#中, 使用加号(+)连接字符串, 而 “&” 符号表示两个不同整数值的按位 AND 运算。“|” 符号则在两个整数之间执行按位 OR 运算。Visual Basic 程序员可能还没有使用过取模(%)运算符, 它返回除运算的余数, 例如, 如果 x 等于 7, 则 x % 5 会返回 2。

在 C#中很少会用到指针, 因此也很少用到间接寻址运算符(->)。使用它们的唯一场合是在不安全的代码块中, 因为只有在此 C#才允许使用指针。指针和不安全的代码见第 13 章。

7.1.1 运算符的简化操作

表 7-3 列出了 C# 中的全部简化赋值运算符。

表 7-3

运算符的简化操作	等 价 于
<code>x++, ++x</code>	<code>x = x + 1</code>
<code>x--, --x</code>	<code>x = x - 1</code>
<code>x += y</code>	<code>x = x + y</code>
<code>x -= y</code>	<code>x = x - y</code>
<code>x *= y</code>	<code>x = x * y</code>
<code>x /= y</code>	<code>x = x / y</code>
<code>x %= y</code>	<code>x = x % y</code>
<code>x >>= y</code>	<code>x = x >> y</code>
<code>x <<= y</code>	<code>x = x << y</code>
<code>x &= y</code>	<code>x = x & y</code>
<code>x = y</code>	<code>x = x y</code>

为什么用两个例子来说明“++”增量和“--”减量运算符？把运算符放在表达式的前面称为前置，把运算符放在表达式的后面称为后置。要点是注意它们的行为方式有所不同。

增量或减量运算符可以作用于整个表达式，也可以作用于表达式的内部。当 `x++` 和 `++x` 单独占一行时，它们的作用是相同的，对应于语句 `x = x + 1`。但当它们用于较长的表达式的内部时，把运算符放在前面(`++x`)会在计算表达式之前递增 `x`，换言之，递增了 `x` 后，在表达式中使用新值进行计算。而把运算符放在后面(`x++`)会在计算表达式之后递增 `x`——使用 `x` 的原始值计算表达式。下面的例子使用“++”增量运算符说明了它们的区别：

```
int x = 5;
if (++x == 6) // true — x is incremented to 6 before the evaluation
{
    Console.WriteLine("This will execute");
}

if (x++ == 7) // false — x is incremented to 7 after the evaluation
{
    Console.WriteLine("This won't");
}
```

判断第一个 `if` 条件得到 `true`，因为在计算表达式之前，`x` 从 5 递增为 6。第二条 `if` 语句中的条件为 `false`，因为在计算完整个表达式(`x == 6`)后，`x` 才递增为 7。

前置运算符 `--x` 和后置运算符 `x--` 与此类似，但它们是递减，而不是递增。

其他简化运算符，如 `+=` 和 `-=` 需要两个操作数，通过对第 1 个操作数执行算术、逻辑或按位运算，从而改变第一个操作数的值。例如，下面两行代码是等价的：

```
x += 5;
```

```
x = x + 5;
```

下面介绍在 C# 代码中频繁使用的基本运算符和类型强制转换运算符。

1. 条件运算符

条件运算符(?:)也称为三元运算符,是 if...else 结构的简化形式。其名称的出处是它带有 3 个操作数。它首先判断一个条件,如果条件为真,就返回一个值;如果条件为假,则返回另一个值。其语法如下:

```
condition ? true_value : false_value
```

其中 condition 是要判断的布尔表达式, true_value 是 condition 为 true 时返回的值, false_value 是 condition 为 false 时返回的值。

恰当地使用三元运算符,可以使程序非常简洁。它特别适合于给被调用的函数提供两个参数中的一个。使用它可以把布尔值转换为字符串值 true 或 false。它也很适合于显示正确的单数形式或复数形式,例如:

```
int x = 1;
string s = x + " ";
s += (x == 1 ? "man": "men");
Console.WriteLine(s);
```

如果 x 等于 1,这段代码就显示 1 man; 如果 x 等于其他数,就显示其正确的复数形式。但要注意,如果结果需要本地化为不同的语言中,就必须编写更复杂的例程,以考虑到不同语言的不同语法。

2. checked 和 unchecked 运算符

考虑下面的代码:

```
byte b = 255;
b++;
Console.WriteLine(b.ToString());
```

byte 数据类型只能包含 0~255 的数,所以递增 b 的值会导致溢出。CLR 如何处理这个溢出取决于许多因素,包括编译器选项,所以只要有未预料到的溢出风险,就需要用某种方式确保得到我们想要的结果。

为此,C#提供了 checked 和 unchecked 运算符。如果把一个代码块标记为 checked,CLR 就会执行溢出检查,如果发生溢出,就抛出 OverflowException 异常。下面修改代码,使之包含 checked 运算符:

```
byte b = 255;
checked
{
    b++;
}
Console.WriteLine(b.ToString());
```

运行这段代码,就会得到一条错误信息:

```
Unhandled Exception: System.OverflowException: Arithmetic operation resulted in an
overflow at Wrox.ProCSharp.Basics.OverflowTest.Main(String[] args)
```



用 `/checked` 编译器选项进行编译, 就可以检查程序中所有未标记代码中的溢出。

如果要禁止溢出检查, 则可以把代码标记为 `unchecked`:

```
byte b = 255;
unchecked
{
    b++;
}
Console.WriteLine(b.ToString());
```

在本例中, 不会抛出异常, 但会丢失数据——因为 `byte` 数据类型不能包含 256, 溢出的位会丢弃, 所以 `b` 变量得到的值是 0。

注意, `unchecked` 是默认行为。只有在需要把几行未检查的代码放在一个显式地标记为 `checked` 的大代码块中, 才需要显式地使用 `unchecked` 关键字。

3. is 运算符

`is` 运算符可以检查对象是否与特定的类型兼容。“兼容”表示对象或者该类型, 或者派生自该类型。例如, 要检查变量是否与 `object` 类型兼容, 可以使用下面的代码:

```
int i = 10;
if (i is object)
{
    Console.WriteLine("i is an object");
}
```

`int` 和所有 C# 数据类型一样, 也从 `object` 继承而来; 表达式 `i is object` 将为 `true`, 并显示相应的消息。

4. as 运算符

`as` 运算符用于执行引用类型的显式类型转换。如果要转换的类型与指定的类型兼容, 转换就会成功进行; 如果类型不兼容, `as` 运算符就会返回 `null` 值。如下面的代码所示, 如果 `object` 引用实际上不引用 `string` 实例, 把 `object` 引用转换为 `string` 就会返回 `null`:

```
object o1 = "Some String";
object o2 = 5;

string s1 = o1 as string; // s1 = "Some String"
string s2 = o2 as string; // s2 = null
```

`as` 运算符允许在一步中进行安全的类型转换, 不需要先使用 `is` 运算符测试类型, 再执行转换。

5. sizeof 运算符

使用 `sizeof` 运算符可以确定栈中值类型需要的长度(单位是字节):

```
Console.WriteLine(sizeof(int));
```

其结果是显示数字 4，因为 `int` 有 4 个字节。

如果对于复杂类型(和非基元类型)使用 `sizeof` 运算符，就需要把代码放在 `unsafe` 块中，如下所示：

```
unsafe
{
    Console.WriteLine(sizeof(Customer));
}
```

第 13 章将详细论述不安全的代码。

6. `typeof` 运算符

`typeof` 运算符返回一个表示特定类型的 `System.Type` 对象。例如，`typeof(string)` 返回表示 `System.String` 类型的 `Type` 对象。在使用反射技术动态地查找对象的相关信息时，这个运算符很有用。第 14 章将介绍反射。

7. 可空类型和运算符

对于布尔类型，可以给它指定 `true` 或 `false` 值。但是，要把该类型的值定义为 `undefined`，该怎么办？此时使用可空类型可以给应用程序提供一个独特的值。如果在程序中使用可空类型，就必须考虑 `null` 值在与各种运算符一起使用时的影响。通常可空类型与一元或二元运算符一起使用时，如果其中一个操作数或两个操作数都是 `null`，其结果就是 `null`。例如：

```
int? a = null;
int? b = a + 4; // b = null
int? c = a * 5; // c = null
```

但是在比较可空类型时，只要有一个操作数是 `null`，比较的结果就是 `false`。即不能因为一个条件是 `false`，就认为该条件的对立面是 `true`，这在使用非可空类型的程序中很常见。例如：

```
int? a = null;
int? b = -5;

if (a >= b)
    Console.WriteLine("a >= b");
else
    Console.WriteLine("a < b");
```



`null` 值的可能性表示，不能随意合并表达式中的可空类型和非可空类型，详见 7.2.1 节的内容。

8. 空合并运算符

空合并运算符(`??`)提供了一种快捷方式，可以在处理可空类型和引用类型时表示 `null` 可能的值。这个运算符放在两个操作数之间，第一个操作数必须是一个可空类型或引用类型；第二个操作数必须与第一个操作数的类型相同，或者可以隐含地转换为第一个操作数的类型。空合并运算符的计算如下：

- 如果第一个操作数不是 `null`，整个表达式就等于第一个操作数的值。

- 如果第一个操作数是 `null`，整个表达式就等于第二个操作数的值。

例如：

```
int? a = null;
int b;

b = a ?? 10; // b has the value 10
a = 3;
b = a ?? 10; // b has the value 3
```

如果第二个操作数不能隐含地转换为第一个操作数的类型，就生成一个编译错误。

7.1.2 运算符的优先级

表 7-4 显示了 C# 运算符的优先级。表 7-4 顶部的运算符有最高的优先级(即在包含多个运算符的表达式中，最先计算该运算符)：

表 7-4

组	运 算 符
初级运算符	<code>() . [] x++ x-- new typeof sizeof checked unchecked</code>
一元运算符	<code>+ - ! ~ ++x --x</code> 和数据类型强制转换
乘/除运算符	<code>* / %</code>
加/减运算符	<code>+ -</code>
移位运算符	<code><< >></code>
关系运算符	<code>< > <= >= is as</code>
比较运算符	<code>== !=</code>
按位 AND 运算符	<code>&</code>
按位 XOR 运算符	<code>^</code>
按位 OR 运算符	<code> </code>
布尔 AND 运算符	<code>&&</code>
布尔 OR 运算符	<code> </code>
条件运算符	<code>?:</code>
赋值运算符	<code>= += -= *= /= %= &= = ^= <<= >>= >>=</code>



在复杂的表达式中，应避免利用运算符优先级来生成正确的结果。使用圆括号指定运算符的执行顺序，可以使代码更整洁，避免出现潜在的冲突。

7.2 类型的安全性

第 1 章提到中间语言(IL)可以对其代码强制实现强类型安全性。强类型化支持 .NET 提供的许多服务，包括安全性和语言的交互性。因为 C# 这种语言会编译为 IL，所以 C# 也是强类型的。此外，这说明数据类型并不总是无缝地可互换的。本节将介绍基元类型之间的转换。



C#也支持不同引用类型之间的轮换, 在与其他类型相互转换时还允许定义所创建的数据类型的行为方式。本章稍后将详细讨论这两个主题。

另一方面, 泛型可以避免对一些常见的情形进行类型转换, 详见第 5 章和第 10 章。

7.2.1 类型转换

我们常常需要把数据从一种类型转换为另一种类型。考虑下面的代码:

```
byte value1 = 10;
byte value2 = 23;
byte total;
total = value1 + value2;
Console.WriteLine(total);
```

在试图编译这些代码时, 会得到一条错误消息:

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

问题是, 我们把两个 byte 型数据加在一起时, 应返回 int 型结果, 而不是另一个 byte。这是因为 byte 包含的数据只能为 8 位, 所以把两个 byte 型数据加在一起, 很容易得到不能存储在单个字节中的值。如果要把结果存储在一个 byte 变量中, 就必须把它转换回一个 byte。C#支持两种转换方式: 隐式转换和显式转换。

1. 隐式转换

只要能保证值不会发生任何变化, 类型转换就可以自动(隐式)进行。这就是前面代码失败的原因: 试图从 int 转换为 byte, 而可能丢失了 3 个字节的数据。编译器不会告诉我们该怎么做, 除非我们明确告诉它这就是我们希望的! 如果在 long 型变量中而不是 byte 型变量中存储结果, 就不会有问题了:

```
byte value1 = 10;
byte value2 = 23;
long total; // this will compile fine
total = value1 + value2;
Console.WriteLine(total);
```

这是因为 long 型变量包含的数据字节比 byte 类型多, 所以没有丢失数据的危险。在这些情况下, 编译器会很顺利地转换, 我们也不需要显式提出要求。

表 7-5 列出了 C#支持的隐式类型转换。

表 7-5

源 类 型	目 的 类 型
sbyte	short、int、long、float、double、decimal、BigInteger
byte	short、ushort、int、uint、long、ulong、float、double、decimal、BigInteger
short	int、long、float、double、decimal、BigInteger