

.NET 开发经典名著



# C#高级编程(第8版)

[美] Christian Nagel  
Bill Evjen  
Jay Glynn 著  
Karli Watson  
Morgan Skinner  
李 铭 译  
黄 静 审校

清华大学出版社

北 京

Christian Nagel, Bill Evjen, Jay Glynn, Karli Watson, Morgan Skinner

Professional C# 2012 and .NET 4.5

EISBN: 978-1-118-31442-5

Copyright © 2013 by John Wiley & Sons, 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: 01-2013-7011

Copies of this book sold without a Wiley sticker on the cover are unauthorized and illegal.

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8版)/(美)内格尔(Nagel, C.)等著; 李铭译. —北京: 清华大学出版社, 2013.10  
(.NET开发经典名著)

书名原文: Professional C# 2012 and .NET 4.5

ISBN 978-7-302-33411-8

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

中国版本图书馆 CIP 数据核字(2013)第 180825 号

责任编辑: 王 军 于 平

装帧设计: 牛静敏

责任校对: 成凤进

责任印制:

出版发行: 清华大学出版社

网 址: <http://www.tup.com.cn>, <http://www.wqbook.com>

地 址: 北京清华大学学研大厦 A 座 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](mailto:c-service@tup.tsinghua.edu.cn)

质量反馈: 010-62772015, [zhiliang@tup.tsinghua.edu.cn](mailto:zhiliang@tup.tsinghua.edu.cn)

印 刷 者:

装 订 者:

经 销: 全国新华书店

开 本: 185mm×260mm 印 张: 96.75 字 数: 2663 千字

版 次: 2013 年 10 月第 1 版 印 次: 2013 年 10 月第 1 次印刷

印 数: 1~5000

定 价: 148.00 元

---

产品编号:

# 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。目前, 他主要开发 Windows Store 应用程序来访问 Windows Azure 服务。他具备 Microsoft 技术的深厚功底, 编写了大量图书, 并获得了 Microsoft 认证培训师和专业开发人员证书。Christian 在国际会议上发表演讲(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](http://www.cninnovation.com) 和 [www.thinktecture.com](http://www.thinktecture.com) 可以联系 Christian, 其微博是 @christiannagel。

**Jay Glynn** 开发软件的时间超过 20 年, 使用 PICK Basic 为 PICK 操作系统编写应用程序。到目前为止, 他使用过 Paradox PAL and Object PAL、Delphi、VBA、Visual Basic、C、Java 和 C# 编写软件。他目前是 UL PureSafety 的高级软件工程师, 编写基于 Web 的应用程序。

**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。他现在是一位独立的顾问。

# 技术编辑简介

**David Franson** 自从 20 世纪 90 年代开始, 就成为网络、编程、2D 和 3D 计算机图形领域的专家, 他编写了 *2D Artwork and 3D Modeling for Game Artists*、*The Dark Side of Game Texturing* 和 *Game Character Design Complete*。

**Don Reamey** 是 TIBCO Software 的构建师/首席工程师, 负责 TIBCO Sportfire 商务智能分析软件。在加入 TIBCO 之前, Don 在 Microsoft 做了 12 年的软件开发工程师, 其主要工作是 SharePoint、SharePoint Online 和 InfoPath Forms Service。Don 还用 10 年的时间为资本市场编写财务服务软件。

**Mitchel Sellers** 擅长使用 Microsoft 技术进行软件开发。他是 IowaComputerGurus 公司的 CEO, 与世界范围内的各种公司一起工作。他是一位 Microsoft C# MVP、Microsoft 认证专家, 还是 *Professional DotNetNuke Module Programmig* (Wrox 出版社, 2009) 的作者。Mitchel 常常在网上撰写技术文章, 也经常在用户组和会议上发言。他也是 DotNetNuke 核心团队的一员, .NET 和 DotNetNuke 开发社区的积极参与者。有关 Mitchel 专业经验、认证和出版物的更多信息, 可访问 <http://mitchelsellers.com/>。



# 致 谢

非常感谢 Adaobi Obi Tulton、Maureen Spears 和 Luann Rouff 提高了本书的可读性，非常感谢 Mary James、Jim Minatel 和 Wiley 中为出版本书提供过帮助的每个人，我还要感谢妻子和孩子支持我的写作。你们都是我的力量源泉。

**Christian Nagel**

我想感谢妻儿支持我完成本书，我还要感谢 Wiley 为出版本书付出努力的所有人。

**Jay Glynn**

# 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 4.5。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 4.5 的新增内容。

不要被这个架构名称中的 .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端上。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 8 和 Windows Server 2012，在内核上都有相同的 Windows API 用于 Windows 桌面和服务端应用程序。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 选择这种软件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，人们编写了大量 Windows 第三方软件，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一流的软件,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。

Windows API 自从 1993 年发布的 Windows NT 以来一直在演化和扩展,但自从 2002 年以来,.NET Framework 对程序编写方式进行了重大的修改,2012 年又进行了一次很大的改动。每 10 年就会发生这种改变吗? Windows 8 现在提供了一种新的 API:用于 Windows Store 应用程序的 Windows 运行库(WinRT)。这个运行库是一个本机 API(类似于 Windows API),它没有把.NET 运行库作为其核心,但提供了基于.NET 理念的非常好的新功能。Windows 8 包含这个 API 的第一个版本,可用于现代模式的应用程序。尽管它不基于.NET,但仍可以将.NET 的一个子集应用于 Windows Store 应用程序,用 C#编写该应用程序。这个新的运行库在未来的几年中会演化,随 Windows 的未来版本一起发布。本书也讨论了如何使用 C#和 WinRT 编写 Windows Store 应用程序。

## .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:** 虽然经典 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, 只需要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 2012:** .NET 附带了一个 Visual Studio 2012 开发环境, 它同样可以很好地利用 C++、C#、Visual Basic 2012 和 ASP.NET 或 XML 进行编码。Visual Studio 2012 集成了这个 IDE 所有以前版本中的各种语言专用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 4.5 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, 是该架构的一个主要版本。2.0 版本的主要新特性是 C# 和运行库中对泛型的支持(为泛型修改了 IL 代码)、新类和接口。.NET 3.0 以 2.0 运行库为基础, 引入了创建 UI 的新方式(WPF 和 XAML, 基于矢量的图形替代了基于像素的图形)和一个新的通信技术(WCF)。.NET 3.5 和 C# 3.0 引入了 LINQ, 这是可用于所有数据源的查询语法。.NET Framework 4 是该产品的另一个重要的版本, 也引入了运行库的一个新版本 4.0 和 C# 的新版本 4.0, 提供了动态语言集成和大量用于并行编程的新库。.NET Framework 4.5 基于 4.0 运行库的更新版本, 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# 2012 和 .NET Framework 4.5 中的一些新变化。

### 异步编程

阻塞 UI 对用户并不友好, 如果 UI 不响应, 用户就会不耐烦。也许读者在 Visual Studio 中也有这种经历。而 Visual Studio 在这方面好了许多, 在许多情形下响应得更快。

.NET Framework 总是提供方法的异步调用。但是, 使用同步方法比调用其异步变体容易得多。这在 C# 5.0 版本中有了改变。异步编程与编写同步程序一样容易。新的 C# 关键字基于自从 .NET 4.0 以来就有的 .NET 并行库, 现在该语言提供了高效功能。



## Windows Store 应用程序和 Windows 运行库

Windows Store 应用程序可以使用 C#、Windows 运行库和 .NET Framework 的一个子集编写，Windows 运行库是一个新的本机 API，提供了类似于 .NET 的类、方法、属性和事件。使用语言投射功能，改善了 .NET 运行库。在 .NET 4.5 中，.NET 4.0 运行库进行了就地更新。

## 数据访问的改善

ADO.NET Entity Framework 提供了重要的新功能。其版本从 .NET 4.0 的 4.0 改为 .NET 4.5 的 5.0。 .NET 4.0 发布后，Entity Framework 已经在 4.1、4.2 和 4.3 中接受了更新。现在新功能，例如 Code First、空间类型、使用枚举、表值功能等，都可以使用了。

## WPF 的改善

WPF 进行了改进，以编写 Windows 桌面应用程序。现在可以在非 UI 线程中填充集合，功能区控件现在是架构的一部分，通过事件的弱引用也更容易实现，数据验证可以用 `INotifyDataErrorInfo` 接口异步完成；实时绘图功能可以方便地动态排序、分组修改了的数据。

## ASP.NET MVC

Visual Studio 2010 包含 ASP.NET MVC 2.0。Visual Studio 2012 发布后，就可以使用 ASP.NET MVC 4.0 了。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 应用程序的方式。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。ASP.NET MVC 不是 ASP.NET Web 窗体的替代品，而只是构建应用程序的另一种方式。

## 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 15 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++ 演变成为一种用于在 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 2012，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 2012 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 2012 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 4.5 运行在 Windows Vista/7/8 和服务器操作系统 Windows Server 2008、2008 R2 和 2012 上。要使用.NET 编写代码，需要安装.NET 4.5 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 2012。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 2012。本部分的两章探讨使用工具构建基于 .NET Framework 4.5 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。其中一部分介绍如何使用平台调用和 COM 交互操作功能,与本地代码和程序集进行交互操作。本部分还讨论了 Windows 运行库与 .NET 的区别,以及如何编写 Windows 8 模式的程序。

### 第 IV 部分——数据

本部分介绍如何使用 ADO.NET 访问数据库,学习 ADO.NET Entity Framework。我们可以使用核心 ADO.NET 获得最佳性能,而使用 ADO.NET Entity Framework 可以方便地把对象映射到关系上。还讨论了现在可以使用的 Model First、Database First 和 Code First 编程模型。我们还详细说明 .NET 对 XML 的支持,以及如何使用 LINQ 查询 XML 数据源。

### 第 V 部分——显示

本部分首先阐述如何编写基于 Windows Presentation Foundation 的应用程序,介绍不同的控件类型、样式、资源和数据绑定,以及如何创建固定的和流畅的文档并打印出来。本部分还会介绍如何创建 Windows Store 应用程序,使用图片生成更漂亮的 UI、网格,以及与其他应用程序交互操作的协定。最后讨论 ASP.NET 提供的许多新功能,用 ASP.NET Web 窗体创建 Web 站点、ASP.NET MVC 和动态数据。

### 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,以及使用 WCF 数据服务访问数据。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)和对等网络。

## 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 <http://www.tupwk.com.cn/downpage> 上下载。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 978-1-118-31442-5。

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，如果你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](http://www.wrox.com/misc-pages/booklist.shtml)。

如果在 Book Errata 页面上没有看到你找出的错误，请进入 [www.wrox.com/contact/techsupport.shtml](http://www.wrox.com/contact/techsupport.shtml)，填写表单，发电子邮件，我们会检查你的信息，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](http://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，当论坛上有了新帖子时，会给你发送你选择的主题。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，必须执行下面的步骤：

- (1) 进入 [p2p.wrox.com](http://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	2	
1.1 C#与.NET 的关系	2	
1.2 公共语言运行库	3	
1.2.1 平台无关性	3	
1.2.2 提高性能	3	
1.2.3 语言的互操作性	4	
1.3 中间语言	5	
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	6	
1.3.3 强数据类型化	7	
1.3.4 通过异常处理错误	11	
1.3.5 特性的使用	12	
1.4 程序集	12	
1.4.1 私有程序集	13	
1.4.2 共享程序集	13	
1.4.3 反射	14	
1.4.4 并行编程	14	
1.4.5 异步编程	14	
1.5 .NET Framework 类	14	
1.6 名称空间	15	
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 使用 WPF	18	
1.7.3 Windows 8 应用程序	18	
1.7.4 Windows 服务	18	
1.7.5 WCF	19	
1.7.6 Windows WF	19	
1.8 C#在.NET 企业体系结构中的作用	19	
1.9 小结	20	
第 2 章 核心 C#	22	
2.1 C#基础	23	
2.2 第一个 C#程序	23	
2.2.1 代码	23	
2.2.2 编译并运行程序	23	
2.2.3 详细介绍	24	
2.3 变量	26	
2.3.1 变量的初始化	26	
2.3.2 类型推断	27	
2.3.3 变量的作用域	28	
2.3.4 常量	30	
2.4 预定义数据类型	31	
2.4.1 值类型和引用类型	31	
2.4.2 CTS 类型	32	
2.4.3 预定义的值类型	33	
2.4.4 预定义的引用类型	35	
2.5 流控制	37	
2.5.1 条件语句	37	
2.5.2 循环	41	
2.5.3 跳转语句	44	
2.6 枚举	45	
2.7 名称空间	46	
2.7.1 using 语句	47	
2.7.2 名称空间的别名	48	
2.8 Main()方法	49	
2.8.1 多个 Main()方法	49	
2.8.2 给 Main()方法传递参数	50	
2.9 有关编译 C#文件的更多内容	51	
2.10 控制台 I/O	53	
2.11 使用注释	54	
2.11.1 源文件中的内部注释	54	
2.11.2 XML 文档	55	

2.12	C#预处理器指令	57	4.3.2	隐藏方法	97
2.12.1	#define 和 #undef	57	4.3.3	调用函数的基类版本	98
2.12.2	#if、#elif、#else 和 #endif	58	4.3.4	抽象类和抽象函数	99
2.12.3	#warning 和 #error	59	4.3.5	密封类和密封方法	99
2.12.4	#region 和 #endregion	59	4.3.6	派生类的构造函数	100
2.12.5	#line	60	4.4	修饰符	105
2.12.6	#pragma	60	4.4.1	可见性修饰符	105
2.13	C#编程规则	60	4.4.2	其他修饰符	106
2.13.1	关于标识符的规则	60	4.5	接口	106
2.13.2	用法约定	61	4.5.1	定义和实现接口	107
2.14	小结	67	4.5.2	派生的接口	110
第3章	对象和类型	68	4.6	小结	112
3.1	创建及使用类	68	第5章	泛型	113
3.2	类和结构	69	5.1	泛型概述	113
3.3	类	69	5.1.1	性能	114
3.3.1	数据成员	70	5.1.2	类型安全	115
3.3.2	函数成员	70	5.1.3	二进制代码的重用	115
3.3.3	只读字段	82	5.1.4	代码的扩展	116
3.4	匿名类型	83	5.1.5	命名约定	116
3.5	结构	84	5.2	创建泛型类	116
3.5.1	结构是值类型	85	5.3	泛型类的功能	120
3.5.2	结构和继承	86	5.3.1	默认值	121
3.5.3	结构的构造函数	86	5.3.2	约束	122
3.6	弱引用	86	5.3.3	继承	124
3.7	部分类	88	5.3.4	静态成员	125
3.8	静态类	89	5.4	泛型接口	125
3.9	Object 类	89	5.4.1	协变和抗变	126
3.9.1	System.Object()方法	90	5.4.2	泛型接口的协变	127
3.9.2	ToString()方法	91	5.4.3	泛型接口的抗变	128
3.10	扩展方法	92	5.5	泛型结构	129
3.11	小结	93	5.6	泛型方法	132
第4章	继承	94	5.6.1	泛型方法示例	132
4.1	继承	94	5.6.2	带约束的泛型方法	133
4.2	继承的类型	94	5.6.3	带委托的泛型方法	134
4.2.1	实现继承和接口继承	94	5.6.4	泛型方法规范	135
4.2.2	多重继承	95	5.7	小结	136
4.2.3	结构和类	95	第6章	数组	137
4.3	实现继承	95	6.1	同一类型和不同类型的 多个对象	137
4.3.1	虚方法	96			

6.2 简单数组 .....	138	7.6.2 多重类型强制转换 .....	189
6.2.1 数组的声明 .....	138	7.7 小结 .....	193
6.2.2 数组的初始化 .....	138	<b>第 8 章 委托、Lambda 表达式</b>	
6.2.3 访问数组元素 .....	139	和事件 .....	194
6.2.4 使用引用类型 .....	140	8.1 引用方法 .....	194
6.3 多维数组 .....	141	8.2 委托 .....	195
6.4 锯齿数组 .....	142	8.2.1 声明委托 .....	195
6.5 Array 类 .....	143	8.2.2 使用委托 .....	196
6.5.1 创建数组 .....	143	8.2.3 简单的委托示例 .....	199
6.5.2 复制数组 .....	144	8.2.4 Action<T>和 Func<T>	
6.5.3 排序 .....	145	委托 .....	201
6.6 数组作为参数 .....	148	8.2.5 BubbleSorter 示例 .....	202
6.6.1 数组协变 .....	149	8.2.6 多播委托 .....	204
6.6.2 ArraySegment<T> .....	149	8.2.7 匿名方法 .....	208
6.7 枚举 .....	150	8.3 Lambda 表达式 .....	209
6.7.1 IEnumerator 接口 .....	150	8.3.1 参数 .....	209
6.7.2 foreach 语句 .....	151	8.3.2 多行代码 .....	210
6.7.3 yield 语句 .....	151	8.3.3 闭包 .....	210
6.8 元组 .....	156	8.3.4 使用 foreach 语句的闭包 .....	211
6.9 结构比较 .....	157	8.4 事件 .....	212
6.10 小结 .....	160	8.4.1 事件发布程序 .....	212
<b>第 7 章 运算符和类型强制转换</b> .....	161	8.4.2 事件侦听器 .....	214
7.1 运算符和类型转换 .....	161	8.4.3 弱事件 .....	215
7.2 运算符 .....	161	8.5 小结 .....	219
7.2.1 运算符的简化操作 .....	163	<b>第 9 章 字符串和正则表达式</b> .....	220
7.2.2 运算符的优先级 .....	167	9.1 System.String 类 .....	221
7.3 类型的安全性 .....	168	9.1.1 创建字符串 .....	222
7.3.1 类型转换 .....	168	9.1.2 StringBuilder 成员 .....	225
7.3.2 装箱和拆箱 .....	172	9.1.3 格式字符串 .....	226
7.4 比较对象的相等性 .....	172	9.2 正则表达式 .....	231
7.4.1 比较引用类型的相等性 .....	172	9.2.1 正则表达式概述 .....	231
7.4.2 比较值类型的相等性 .....	173	9.2.2 RegularExpressionsPlayaround	
7.5 运算符重载 .....	174	示例 .....	232
7.5.1 运算符的工作方式 .....	175	9.2.3 显示结果 .....	235
7.5.2 运算符重载的示例:		9.2.4 匹配、组合和捕获 .....	236
Vector 结构 .....	176	9.3 小结 .....	238
7.6 用户定义的类型强制转换 .....	182	<b>第 10 章 集合</b> .....	239
7.6.1 实现用户定义的类型		10.1 概述 .....	239
强制转换 .....	184		



10.2	集合接口和类型 .....	240	11.2.6	分组 .....	302
10.3	列表 .....	241	11.2.7	对嵌套的对象分组 .....	303
10.3.1	创建列表 .....	242	11.2.8	内连接 .....	304
10.3.2	只读集合 .....	251	11.2.9	左外连接 .....	305
10.4	队列 .....	251	11.2.10	组连接 .....	306
10.5	栈 .....	255	11.2.11	集合操作 .....	309
10.6	链表 .....	256	11.2.12	合并 .....	310
10.7	有序列表 .....	262	11.2.13	分区 .....	311
10.8	字典 .....	263	11.2.14	聚合操作符 .....	312
10.8.1	键的类型 .....	264	11.2.15	转换操作符 .....	314
10.8.2	字典示例 .....	265	11.2.16	生成操作符 .....	315
10.8.3	Lookup 类 .....	269	11.3	并行 LINQ .....	316
10.8.4	有序字典 .....	270	11.3.1	并行查询 .....	316
10.9	集 .....	270	11.3.2	分区器 .....	317
10.10	可观察的集合 .....	272	11.3.3	取消 .....	317
10.11	位数组 .....	273	11.4	表达式树 .....	318
10.11.1	BitArray 类 .....	274	11.5	LINQ 提供程序 .....	320
10.11.2	BitVector32 结构 .....	276	11.6	小结 .....	321
10.12	并发集合 .....	278	第 12 章	动态语言扩展 .....	322
10.12.1	创建管道 .....	279	12.1	DLR .....	322
10.12.2	使用 Blocking- Collection .....	282	12.2	dynamic 类型 .....	323
10.12.3	使用 Concurrent- Dictionary .....	283	12.3	包含 DLR ScriptRuntime .....	327
10.12.4	完成管道 .....	285	12.4	DynamicObject 和 ExpandoObject .....	330
10.13	性能 .....	286	12.4.1	DynamicObject .....	330
10.14	小结 .....	288	12.4.2	ExpandoObject .....	332
第 11 章	LINQ .....	289	12.5	小结 .....	333
11.1	LINQ 概述 .....	289	第 13 章	异步编程 .....	334
11.1.1	列表和实体 .....	289	13.1	异步编程的重要性 .....	334
11.1.2	LINQ 查询 .....	293	13.2	异步模式 .....	335
11.1.3	扩展方法 .....	294	13.2.1	同步调用 .....	342
11.1.4	推迟查询的执行 .....	295	13.2.2	异步模式 .....	343
11.2	标准的查询操作符 .....	297	13.2.3	基于事件的异步模式 .....	344
11.2.1	筛选 .....	299	13.2.4	基于任务的异步模式 .....	345
11.2.2	用索引筛选 .....	299	13.3	异步编程的基础 .....	347
11.2.3	类型筛选 .....	300	13.3.1	创建任务 .....	347
11.2.4	复合的 from 子句 .....	300	13.3.2	调用异步方法 .....	348
11.2.5	排序 .....	301	13.3.3	延续任务 .....	348

13.3.4	同步上下文	349	15.3.2	TypeView 示例	392
13.3.5	使用多个异步方法	349	15.3.3	Assembly 类	393
13.3.6	转换异步模式	350	15.3.4	完成 WhatsNewAttributes 示例	395
13.4	错误处理	351	15.4	小结	398
13.4.1	异步方法的异常处理	352	第 16 章	错误和异常	399
13.4.2	多个异步方法的 异常处理	352	16.1	简介	399
13.4.3	AggregateException 类	353	16.2	异常类	400
13.5	取消	353	16.3	捕获异常	401
13.5.1	开始取消任务	354	16.3.1	实现多个 catch 块	403
13.5.2	使用框架特性取消任务	354	16.3.2	在其他代码中捕获异常	407
13.5.3	取消自定义任务	355	16.3.3	System.Exception 属性	407
13.6	小结	355	16.3.4	没有处理异常时所 发生的情况	408
第 14 章	内存管理和指针	356	16.3.5	嵌套的 try 块	408
14.1	内存管理	356	16.4	用户定义的异常类	410
14.2	后台内存管理	356	16.4.1	捕获用户定义的异常	411
14.2.1	值数据类型	357	16.4.2	抛出用户定义的异常	412
14.2.2	引用数据类型	358	16.4.3	定义用户定义的异常类	415
14.2.3	垃圾回收	359	16.5	调用者信息	417
14.3	释放非托管的资源	361	16.6	小结	418
14.3.1	析构函数	361	第 II 部分	Visual Studio	
14.3.2	IDisposable 接口	362	第 17 章	Visual Studio 2012	420
14.3.3	实现 IDisposable 接口和 析构函数	363	17.1	用 Visual Studio 2012 进行 工作	420
14.4	不安全的代码	365	17.1.1	项目文件的改进	423
14.4.1	用指针直接访问内存	365	17.1.2	Visual Studio 的版本	423
14.4.2	指针示例: PointerPlayground	374	17.1.3	Visual Studio 设置	424
14.4.3	使用指针优化性能	378	17.2	创建项目	424
14.5	小结	381	17.2.1	面向多个版本的 .NET Framework	425
第 15 章	反射	382	17.2.2	选择项目类型	427
15.1	在运行期间处理和检查代码	382	17.3	浏览并编写项目	430
15.2	自定义特性	383	17.3.1	Solution Explorer	430
15.2.1	编写自定义特性	383	17.3.2	用代码编辑器进行工作	436
15.2.2	自定义特性示例: WhatsNewAttributes	386	17.3.3	学习和理解其他窗口	438
15.3	反射	389	17.3.4	排列窗口	442
15.3.1	System.Type 类	389			

17.4	构建项目 .....	442	第 18 章	部署 .....	473
17.4.1	构建、编译和生成 .....	442	18.1	部署是应用程序生命周期的 一部分 .....	473
17.4.2	调试版本和发布版本 .....	443	18.2	部署的规划 .....	473
17.4.3	选择配置 .....	445	18.2.1	部署选项 .....	474
17.4.4	编辑配置 .....	445	18.2.2	部署要求 .....	474
17.5	调试代码 .....	446	18.2.3	部署.NET 运行库 .....	475
17.5.1	设置断点 .....	446	18.3	传统的部署选项 .....	475
17.5.2	使用数据提示和调试器 可视化工具 .....	447	18.3.1	xcopy 部署 .....	476
17.5.3	监视和修改变量 .....	448	18.3.2	xcopy 和 Web 应用程序 .....	476
17.5.4	异常 .....	449	18.3.3	Windows Installer .....	476
17.5.5	多线程 .....	450	18.4	ClickOnce .....	477
17.5.6	IntelliTrace .....	451	18.4.1	ClickOnce 操作 .....	477
17.6	重构工具 .....	451	18.4.2	发布 ClickOnce 应用程序 .....	477
17.7	体系结构工具 .....	453	18.4.3	ClickOnce 设置 .....	479
17.7.1	依赖项关系图 .....	453	18.4.4	ClickOnce 文件的 应用程序缓存 .....	481
17.7.2	层关系图 .....	454	18.4.5	应用程序的安装 .....	481
17.8	分析应用程序 .....	456	18.4.6	ClickOnce 部署 API .....	482
17.8.1	序列图 .....	456	18.5	Web 部署 .....	483
17.8.2	探查器 .....	457	18.5.1	Web 应用程序 .....	483
17.8.3	Concurrency Visualizer .....	459	18.5.2	配置文件 .....	483
17.8.4	Code Analysis .....	459	18.5.3	创建 Web Deploy 包 .....	484
17.8.5	Code Metrics .....	460	18.6	Windows 8 应用程序 .....	485
17.9	单元测试 .....	461	18.6.1	创建应用程序包 .....	486
17.9.1	创建单元测试 .....	461	18.6.2	Windows App Certification Kit .....	487
17.9.2	运行单元测试 .....	462	18.6.3	旁加载 .....	487
17.9.3	预期异常 .....	463	18.6.4	Windows 部署 API .....	488
17.9.4	测试全部代码路径 .....	463	18.7	小结 .....	490
17.9.5	外部依赖 .....	464			
17.9.6	Fakes Framework .....	467			
17.10	Windows 8、WCF、WF 等 .....	468			
17.10.1	使用 Visual Studio 2012 生成 WCF 应用程序 .....	468			
17.10.2	使用 Visual Studio 2012 生成 WF 应用程序 .....	470			
17.10.3	使用 Visual Studio 2012 生成 Windows Store 应用程序 .....	470			
17.11	小结 .....	472			

### 第Ⅲ部分 基 础

第 19 章	程序集 .....	492
19.1	程序集的含义 .....	492
19.1.1	程序集的功能 .....	493
19.1.2	程序集的结构 .....	493
19.1.3	程序集清单 .....	494
19.1.4	名称空间、程序集和 组件 .....	494

19.1.5	私有程序集和共享程序集.....	495	20.2.2	后置条件.....	528
19.1.6	附属程序集.....	495	20.2.3	不变量.....	529
19.1.7	查看程序集.....	495	20.2.4	纯粹性.....	529
19.2	构建程序集.....	496	20.2.5	接口的协定.....	529
19.2.1	创建模块和程序集.....	496	20.2.6	简写.....	531
19.2.2	程序集的特性.....	497	20.2.7	协定和遗留代码.....	531
19.2.3	创建和动态加载程序集.....	499	20.3	跟踪.....	532
19.3	应用程序域.....	502	20.3.1	跟踪源.....	533
19.4	共享程序集.....	506	20.3.2	跟踪开关.....	534
19.4.1	强名.....	506	20.3.3	跟踪侦听器.....	535
19.4.2	使用强名获得完整性.....	507	20.3.4	筛选器.....	537
19.4.3	全局程序集缓存.....	508	20.3.5	相关性.....	538
19.4.4	创建共享程序集.....	509	20.3.6	使用 ETW 进行跟踪.....	541
19.4.5	创建强名.....	509	20.4	事件日志.....	542
19.4.6	安装共享程序集.....	510	20.4.1	事件日志体系结构.....	542
19.4.7	使用共享程序集.....	511	20.4.2	事件日志类.....	544
19.4.8	程序集的延迟签名.....	512	20.4.3	创建事件源.....	545
19.4.9	引用.....	513	20.4.4	写入事件日志.....	546
19.4.10	本机映像生成器.....	513	20.4.5	资源文件.....	546
19.5	配置.NET 应用程序.....	514	20.5	性能监视.....	550
19.5.1	配置类别.....	515	20.5.1	性能监视类.....	550
19.5.2	绑定程序集.....	516	20.5.2	性能计数器生成器.....	550
19.6	版本问题.....	517	20.5.3	添加 PerformanceCounter 组件.....	553
19.6.1	版本号.....	517	20.5.4	perfmon.exe.....	555
19.6.2	通过编程方式获取版本.....	518	20.6	小结.....	556
19.6.3	绑定到程序集版本.....	518	第 21 章	任务、线程和同步.....	557
19.6.4	发行者策略文件.....	519	21.1	概述.....	557
19.6.5	运行库的版本.....	521	21.2	Parallel 类.....	558
19.7	在不同的技术之间共享程序集.....	522	21.2.1	用 Parallel.For()方法循环.....	559
19.7.1	共享源代码.....	522	21.2.2	使用 Parallel.ForEach()方法循环.....	562
19.7.2	可移植类库.....	523	21.2.3	通过 Parallel.Invoke()方法调用多个方法.....	563
19.8	小结.....	524	21.3	任务.....	563
第 20 章	诊断.....	525	21.3.1	启动任务.....	563
20.1	诊断概述.....	525	21.3.2	Future——任务的结果.....	566
20.2	代码协定.....	526			
20.2.1	前提条件.....	527			

21.3.3	连续的任务	567	22.2.4	声称	612
21.3.4	任务层次结构	568	22.2.5	客户端应用程序服务	613
21.4	取消架构	568	22.3	加密	618
21.4.1	Parallel.For()方法的取消	569	22.3.1	签名	619
21.4.2	任务的取消	570	22.3.2	交换密钥和安全传输	621
21.5	线程池	571	22.4	资源的访问控制	624
21.6	Thread 类	573	22.5	代码访问安全性	627
21.6.1	给线程传递数据	574	22.5.1	第 2 级安全透明性	627
21.6.2	后台线程	575	22.5.2	权限	628
21.6.3	线程的优先级	576	22.6	使用证书发布代码	632
21.6.4	控制线程	576	22.7	小结	633
21.7	线程问题	577	第 23 章	互操作	634
21.7.1	争用条件	577	23.1	.NET 和 COM 技术	634
21.7.2	死锁	579	23.1.1	元数据	635
21.8	同步	581	23.1.2	释放内存	635
21.8.1	lock 语句和线程安全	582	23.1.3	接口	636
21.8.2	Interlocked 类	587	23.1.4	方法绑定	637
21.8.3	Monitor 类	588	23.1.5	数据类型	637
21.8.4	SpinLock 结构	589	23.1.6	注册	638
21.8.5	WaitHandle 基类	589	23.1.7	线程	638
21.8.6	Mutex 类	590	23.1.8	错误处理	640
21.8.7	Semaphore 类	591	23.1.9	事件	640
21.8.8	Events 类	593	23.1.10	封送	640
21.8.9	Barrier 类	596	23.2	在.NET 客户端中使用 COM 组件	641
21.8.10	ReaderWriterLockSlim 类	598	23.2.1	创建一个 COM 组件	641
21.9	Timer 类	601	23.2.2	创建运行库可调用包装	647
21.10	数据流	602	23.2.3	使用 RCW	648
21.10.1	使用动作块	602	23.2.4	通过动态语言扩展使用 COM 服务	649
21.10.2	源和目标数据块	603	23.2.5	线程问题	650
21.10.3	连接块	604	23.2.6	添加连接点	650
21.11	小结	606	23.3	在 COM 客户端中使用 .NET 组件	652
第 22 章	安全性	608	23.3.1	COM 可调用包装	653
22.1	概述	608	23.3.2	创建.NET 组件	653
22.2	身份验证和授权	609	23.3.3	创建一个类型库	654
22.2.1	标识和 Principal	609	23.3.4	COM 互操作特性	656
22.2.2	角色	610			
22.2.3	声明基于角色的安全性	611			

23.3.5	COM 注册	658	24.10	小结	713
23.3.6	创建 COM 客户端 应用程序	659	第 25 章	事务处理	714
23.3.7	添加连接点	660	25.1	简介	714
23.3.8	使用 sink 对象创建 客户端	661	25.2	概述	715
23.3.9	平台调用	663	25.2.1	事务处理阶段	715
23.4	小结	667	25.2.2	ACID 属性	716
第 24 章	文件和注册表操作	668	25.3	数据库和实体类	716
24.1	文件和注册表	668	25.4	传统的事务	718
24.2	管理文件系统	669	25.4.1	ADO.NET 事务	718
24.2.1	表示文件和文件夹的 .NET 类	670	25.4.2	System.Enterprise- Services	719
24.2.2	Path 类	672	25.5	System.Transactions	720
24.2.3	FileProperties 示例	672	25.5.1	可提交的事务	722
24.3	移动、复制和删除文件	677	25.5.2	事务处理的升级	724
24.3.1	FilePropertiesAndMovement 示例	677	25.5.3	依赖事务	726
24.3.2	FilePropertiesAndMovement 示例的代码	678	25.5.4	环境事务	728
24.4	读写文件	681	25.6	隔离级别	735
24.4.1	读取文件	681	25.7	自定义资源管理器	736
24.4.2	写入文件	683	25.8	文件系统事务	742
24.4.3	流	684	25.9	小结	746
24.4.4	缓存的流	686	第 26 章	网络	747
24.4.5	使用 FileStream 类读写 二进制文件	686	26.1	网络	747
24.4.6	读写文本文件	691	26.2	WebClient 类	748
24.5	映射内存的文件	697	26.2.1	下载文件	748
24.6	读取驱动器信息	698	26.2.2	基本的 WebClient 示例	748
24.7	文件的安全性	700	26.2.3	上传文件	750
24.7.1	从文件中读取 ACL	700	26.3	WebRequest 类和 WebResponse 类	750
24.7.2	从目录中读取 ACL	701	26.3.1	身份验证	752
24.7.3	添加和删除文件中的 ACL 项	703	26.3.2	使用代理	752
24.8	读写注册表	704	26.3.3	异步页面请求	753
24.8.1	注册表	705	26.4	把输出结果显示为 HTML 页面	753
24.8.2	.NET 注册表类	706	26.4.1	从应用程序中进行简单的 Web 浏览	754
24.9	读写独立存储器	709	26.4.2	启动 Internet Explorer 实例	755

26.4.3	给应用程序提供更多 IE 类型的功能	756	27.4.3	sc.exe 实用程序	801
26.4.4	使用 WebBrowser 控件打印	761	27.4.4	Visual Studio Server Explorer	801
26.4.5	显示所请求页面的代码	761	27.4.5	编写自定义 ServiceController 类	801
26.4.6	WebRequest 类和 WebResponse 类的层次结构	762	27.5	故障排除和事件日志	809
26.5	实用工具类	763	27.6	小结	810
26.5.1	URI	763	<b>第 28 章 本地化</b>		<b>811</b>
26.5.2	IP 地址和 DNS 名称	764	28.1	全球市场	811
26.6	较低层的协议	766	28.2	System.Globalization 名称空间	812
26.6.1	使用 SmtplibClient	767	28.2.1	Unicode 问题	812
26.6.2	使用 TCP 类	768	28.2.2	区域性和区域	813
26.6.3	TcpSend 和 TcpReceive 示例	769	28.2.3	使用区域性	817
26.6.4	TCP 和 UDP	771	28.2.4	排序	823
26.6.5	UDP 类	771	28.3	资源	824
26.6.6	Socket 类	772	28.3.1	创建资源文件	824
26.6.7	Websocket	775	28.3.2	资源文件生成器	825
26.7	小结	779	28.3.3	ResourceWriter	825
<b>第 27 章 Windows 服务</b>		<b>780</b>	28.3.4	使用资源文件	826
27.1	Windows 服务	780	28.3.5	System.Resources 名称空间	830
27.2	Windows 服务的体系结构	781	28.4	使用 Visual Studio 的 Windows Forms 本地化	831
27.2.1	服务程序	782	28.4.1	通过编程方式修改区域性	836
27.2.2	服务控制程序	783	28.4.2	使用自定义资源消息	837
27.2.3	服务配置程序	783	28.4.3	资源的自动回退	838
27.2.4	Windows 服务的类	783	28.4.4	外包翻译	838
27.3	创建 Windows 服务程序	783	28.5	ASP.NET Web Forms 的本地化	839
27.3.1	创建服务的核心功能	784	28.6	用 WPF 本地化	841
27.3.2	QuoteClient 示例	787	28.6.1	用于 WPF 的 .NET 资源	842
27.3.3	Windows 服务程序	791	28.6.2	XAML 资源字典	843
27.3.4	线程化和服务	795	28.7	自定义资源读取器	846
27.3.5	服务的安装	795	28.7.1	创建 DatabaseResource-Reader 类	847
27.3.6	安装程序	795	28.7.2	创建 DatabaseResource-Set 类	848
27.4	Windows 服务的监控和控制	799			
27.4.1	MMC 管理单元	800			
27.4.2	net.exe 实用程序	801			

28.7.3 创建 DatabaseResource-Manager 类.....	849	30.2.1 使用属性的 MEF.....	876
28.7.4 DatabaseResourceReader 的客户端应用程序.....	850	30.2.2 基于约定的部件注册 .....	881
28.8 创建自定义区域性.....	850	30.3 定义协定.....	883
28.9 用 Windows Store 应用程序进行本地化 .....	852	30.4 导出部件.....	884
28.9.1 使用资源.....	852	30.4.1 创建部件.....	884
28.9.2 使用多语言应用程序工具集进行本地化 .....	853	30.4.2 导出属性和方法 .....	889
28.10 小结 .....	854	30.4.3 导出元数据 .....	891
<b>第 29 章 核心 XAML .....</b>	<b>855</b>	30.4.4 使用元数据进行惰性加载 .....	893
29.1 XAML 的作用.....	855	30.5 导入部件.....	894
29.2 概述 .....	855	30.5.1 导入连接 .....	896
29.2.1 元素如何映射到.NET 对象上.....	856	30.5.2 部件的惰性加载 .....	898
29.2.2 使用自定义.NET 类 .....	857	30.5.3 用惰性实例化的部件读取元数据 .....	898
29.2.3 把属性用作特性 .....	859	30.6 容器和出口提供程序.....	901
29.2.4 把属性用作元素.....	859	30.7 类别.....	903
29.2.5 基本的.NET 类型 .....	860	30.8 小结.....	905
29.2.6 使用集合和 XAML .....	860	<b>第 31 章 Windows 运行库 .....</b>	<b>906</b>
29.2.7 用 XAML 代码调用构造函数.....	861	31.1 概述.....	906
29.3 依赖属性 .....	861	31.1.1 .NET 与 Windows 运行库的比较 .....	907
29.3.1 创建依赖属性.....	862	31.1.2 名称空间 .....	907
29.3.2 强制值回调 .....	863	31.1.3 元数据 .....	909
29.3.3 值变更回调和事件.....	864	31.1.4 语言投射 .....	910
29.3.4 事件的冒泡和隧道.....	864	31.1.5 Windows 运行库中的类型 .....	912
29.4 附加属性 .....	867	31.2 Windows 运行库组件 .....	913
29.5 标记扩展 .....	870	31.2.1 集合.....	913
29.6 创建自定义标记扩展.....	870	31.2.2 流.....	914
29.7 XAML 定义的标记扩展.....	872	31.2.3 委托与事件 .....	915
29.8 读写 XAML.....	872	31.2.4 异步操作 .....	915
29.9 小结 .....	873	31.3 Windows 8 应用程序 .....	916
<b>第 30 章 Managed Extensibility Framework.....</b>	<b>874</b>	31.4 Windows 8 应用程序的生命周期 .....	918
30.1 概述 .....	874	31.4.1 Windows 8 应用程序的执行状态 .....	919
30.2 MEF 的体系结构 .....	875	31.4.2 SuspensionManager.....	920
		31.4.3 导航状态 .....	921



31.4.4	测试暂停	922	32.10	使用 ADO.NET	971
31.4.5	页面状态	922	32.10.1	分层开发	971
31.5	Windows 8 应用程序的设置	924	32.10.2	生成 SQL Server 的键	972
31.6	摄像头功能	926	32.10.3	命名约定	974
31.7	小结	928	32.11	小结	976
<b>第IV部分 数 据</b>					
第 32 章	核心 ADO.NET	930	第 33 章	ADO.NET Entity Framework	977
32.1	ADO.NET 概述	930	33.1	用 Entity Framework 编程	977
32.1.1	名称空间	931	33.2	Entity Framework 映射	979
32.1.2	共享类	932	33.2.1	逻辑层	979
32.1.3	数据库专用类	932	33.2.2	概念层	981
32.2	使用数据库连接	933	33.2.3	映射层	983
32.2.1	管理连接字符串	934	33.2.4	连接字符串	983
32.2.2	高效地使用连接	935	33.3	实体	984
32.2.3	事务	937	33.4	对象上下文	988
32.3	命令	938	33.5	关系	990
32.3.1	执行命令	939	33.5.1	一个层次结构一个表	990
32.3.2	调用存储过程	942	33.5.2	一种类型一个表	992
32.4	快速数据访问: 数据读取器	944	33.5.3	懒惰加载、延迟加载和 预先加载	993
32.5	异步数据访问: 使用 Task 和 await	947	33.6	查询数据	994
32.6	管理数据和关系: DataSet 类	949	33.6.1	Entity SQL	994
32.6.1	数据表	949	33.6.2	对象查询	995
32.6.2	数据列	950	33.6.3	LINQ to Entities	998
32.6.3	数据关系	955	33.7	把数据写入数据库	999
32.6.4	数据约束	956	33.7.1	对象跟踪	999
32.7	XML 架构: 用 XSD 生成代码	959	33.7.2	改变信息	1000
32.8	填充 DataSet 类	965	33.7.3	附加和分离实体	1001
32.8.1	用数据适配器填充 DataSet	965	33.7.4	存储实体的变化	1002
32.8.2	从 XML 中填充 DataSet 类	967	33.8	使用 POCO 对象	1003
32.9	持久化 DataSet 类的修改	967	33.8.1	定义实体类型	1003
32.9.1	通过数据适配器 进行更新	967	33.8.2	创建数据上下文	1004
32.9.2	写入 XML 输出结果	969	33.8.3	查询和更新	1004
			33.9	使用 Code First 编程模型	1005
			33.9.1	定义实体类型	1005
			33.9.2	创建数据上下文	1006
			33.9.3	创建数据库, 存储实体	1006
			33.9.4	数据库	1007
			33.9.5	查询数据	1007

33.9.6 定制数据库的生成 .....	1008
33.10 小结 .....	1009
<b>第 34 章 处理 XML .....</b>	<b>1010</b>
34.1 XML .....	1010
34.2 .NET 支持的 XML 标准 .....	1011
34.3 System.Xml 名称空间 .....	1011
34.4 使用 System.Xml 类 .....	1012
34.5 读写流格式的 XML .....	1013
34.5.1 使用 XmlReader 类 .....	1013
34.5.2 使用 XmlReader 类 进行验证 .....	1017
34.5.3 使用 XmlWriter 类 .....	1019
34.6 在 .NET 中使用 DOM .....	1020
34.7 使用 XPathNavigator 类 .....	1025
34.7.1 System.Xml.XPath 名称空间 .....	1025
34.7.2 System.Xml.Xsl 名称空间 .....	1030
34.7.3 调试 XSLT .....	1034
34.8 XML 和 ADO.NET .....	1036
34.8.1 将 ADO.NET 数据转换为 XML 文档 .....	1036
34.8.2 把 XML 文档转换为 ADO.NET 数据 .....	1042
34.9 在 XML 中序列化对象 .....	1044
34.10 LINQ to XML 和 .NET .....	1053
34.11 使用不同的 XML 对象 .....	1053
34.11.1 XDocument 对象 .....	1053
34.11.2 XElement 对象 .....	1054
34.11.3 XNamespace 对象 .....	1055
34.11.4 XComment 对象 .....	1057
34.11.5 XAttribute 对象 .....	1058
34.12 使用 LINQ 查询 XML 文档 .....	1059
34.12.1 查询静态的 XML 文档 .....	1059
34.12.2 查询动态的 XML 文档 .....	1060
34.13 XML 文档的更多 查询技术 .....	1062
34.13.1 读取 XML 文档 .....	1062
34.13.2 写入 XML 文档 .....	1063
34.14 小结 .....	1065
<b>第 V 部分 显 示</b>	
<b>第 35 章 核心 WPF .....</b>	<b>1068</b>
35.1 理解 WPF .....	1069
35.1.1 名称空间 .....	1069
35.1.2 类层次结构 .....	1070
35.2 形状 .....	1072
35.3 几何图形 .....	1073
35.4 变换 .....	1075
35.5 画笔 .....	1077
35.5.1 SolidColorBrush .....	1077
35.5.2 LinearGradientBrush .....	1077
35.5.3 RadialGradientBrush .....	1078
35.5.4 DrawingBrush .....	1078
35.5.5 ImageBrush .....	1079
35.5.6 VisualBrush .....	1079
35.6 控件 .....	1081
35.6.1 简单控件 .....	1081
35.6.2 内容控件 .....	1081
35.6.3 带标题的内容控件 .....	1083
35.6.4 项控件 .....	1084
35.6.5 带标题的项控件 .....	1084
35.6.6 修饰 .....	1085
35.7 布局 .....	1086
35.7.1 StackPanel .....	1086
35.7.2 WrapPanel .....	1086
35.7.3 Canvas .....	1087
35.7.4 DockPanel .....	1088
35.7.5 Grid .....	1089
35.8 样式和资源 .....	1090
35.8.1 样式 .....	1090
35.8.2 资源 .....	1092
35.8.3 系统资源 .....	1093
35.8.4 从代码中访问资源 .....	1093

35.8.5	动态资源	1094	36.4.1	BooksDemo 应用程序	
35.8.6	资源字典	1095		内容	1132
35.9	触发器	1096	36.4.2	用 XAML 绑定	1133
35.9.1	属性触发器	1096	36.4.3	简单对象的绑定	1135
35.9.2	多触发器	1098	36.4.4	更改通知	1137
35.9.3	数据触发器	1098	36.4.5	对象数据提供程序	1140
35.10	模板	1100	36.4.6	列表绑定	1142
35.10.1	控件模板	1100	36.4.7	主从绑定	1145
35.10.2	数据模板	1103	36.4.8	多绑定	1145
35.10.3	样式化列表框	1105	36.4.9	优先绑定	1147
35.10.4	ItemTemplate	1106	36.4.10	值的转换	1149
35.10.5	列表框元素的 控件模板	1107	36.4.11	动态添加列表项	1150
35.11	动画	1108	36.4.12	动态添加选项卡中的 项	1151
35.11.1	时间轴	1109	36.4.13	数据模板选择器	1152
35.11.2	非线性动画	1112	36.4.14	绑定到 XML 上	1154
35.11.3	事件触发器	1112	36.4.15	绑定的验证	1156
35.11.4	关键帧动画	1115	36.5	TreeView	1164
35.12	可见状态管理器	1116	36.6	DataGrid	1168
35.12.1	可见的状态	1117	36.6.1	自定义列	1170
35.12.2	变换	1118	36.6.2	行的细节	1171
35.13	3-D	1119	36.6.3	用 DataGrid 进行分组	1171
35.13.1	模型	1120	36.6.4	实时成型	1174
35.13.2	照相机	1121	36.7	小结	1180
35.13.3	光线	1122	第 37 章	用 WPF 创建文档	1181
35.13.4	旋转	1122	37.1	简介	1181
35.14	小结	1123	37.2	文本元素	1182
第 36 章	用 WPF 编写业务 应用程序	1124	37.2.1	字体	1182
36.1	概述	1124	37.2.2	TextEffect	1183
36.2	菜单和功能区分件	1125	37.2.3	内联	1184
36.2.1	菜单控件	1125	37.2.4	块	1186
36.2.2	功能区控件	1126	37.2.5	列表	1188
36.3	Commanding	1128	37.2.6	表	1188
36.3.1	定义命令	1129	37.2.7	块的锚定	1189
36.3.2	定义命令源	1130	37.3	流文档	1191
36.3.3	命令绑定	1130	37.4	固定文档	1195
36.4	数据绑定	1131	37.5	XPS 文档	1199
			37.6	打印	1200

37.6.1	用 PrintDialog 打印 .....	1201	39.2.3	JavaScript 和 jQuery .....	1245
37.6.2	打印可见元素 .....	1201	39.3	托管和配置 .....	1246
37.7	小结 .....	1203	39.4	处理程序和模块 .....	1248
<b>第 38 章</b>	<b>Windows 8 应用程序 .....</b>	<b>1204</b>	39.4.1	创建自定义处理程序 .....	1249
38.1	概述 .....	1204	39.4.2	ASP.NET 处理程序 .....	1250
38.2	Windows 8 的现代 UI 设计 .....	1204	39.4.3	创建自定义模块 .....	1251
38.2.1	内容, 不是 chrome 设计 .....	1205	39.4.4	通用模块 .....	1253
38.2.2	快速流畅 .....	1206	39.5	全局的应用程序类 .....	1254
38.2.3	可读性 .....	1206	39.6	请求和响应 .....	1254
38.3	示例应用程序的核心功能 .....	1206	39.6.1	使用 HttpRequest 对象 .....	1255
38.3.1	文件和目录 .....	1207	39.6.2	使用 HttpResponseMessage 对象 .....	1256
38.3.2	应用程序数据 .....	1208	39.7	状态管理 .....	1256
38.3.3	应用程序页面 .....	1213	39.7.1	视图状态 .....	1257
38.4	应用程序工具栏 .....	1218	39.7.2	cookie .....	1258
38.5	启动与导航 .....	1219	39.7.3	会话 .....	1259
38.6	布局的变化 .....	1222	39.7.4	应用程序状态 .....	1261
38.7	存储 .....	1225	39.7.5	缓存 .....	1262
38.7.1	定义数据协定 .....	1226	39.7.6	配置文件 .....	1263
38.7.2	写入移动数据 .....	1227	39.8	成员和角色 .....	1267
38.7.3	读取数据 .....	1229	39.8.1	配置成员 .....	1267
38.7.4	写入图像 .....	1230	39.8.2	使用成员 API .....	1269
38.7.5	读取图像 .....	1232	39.8.3	启用角色 API .....	1270
38.8	选择器 .....	1233	39.9	小结 .....	1270
38.9	共享协定 .....	1234	<b>第 40 章</b>	<b>ASP.NET Web Forms .....</b>	<b>1271</b>
38.9.1	共享源 .....	1234	40.1	概述 .....	1271
38.9.2	共享目标 .....	1237	40.2	ASPX 页面模型 .....	1272
38.10	Tile .....	1239	40.2.1	添加控件 .....	1272
38.11	小结 .....	1241	40.2.2	使用事件 .....	1273
<b>第 39 章</b>	<b>核心 ASP.NET .....</b>	<b>1242</b>	40.2.3	使用回送 .....	1274
39.1	用于 Web 应用程序的 .NET Framework .....	1242	40.2.4	使用自动回送 .....	1275
39.1.1	ASP.NET Web Forms .....	1243	40.2.5	回送到其他页面 .....	1275
39.1.2	ASP.NET Web Pages .....	1243	40.2.6	定义强类型化的 跨页面回送 .....	1276
39.1.3	ASP.NET MVC .....	1244	40.2.7	使用页面事件 .....	1277
39.2	Web 技术 .....	1244	40.2.8	ASPX 代码 .....	1278
39.2.1	HTML .....	1244	40.2.9	服务器端控件 .....	1280
39.2.2	CSS .....	1245	40.3	母版页 .....	1281

40.3.1	创建母版页	1281	41.2.2	路由约束	1319
40.3.2	使用母版页	1283	41.3	创建控制器	1320
40.3.3	在内容页中定义 母版页内容	1284	41.3.1	动作方法	1321
40.4	导航	1285	41.3.2	参数	1321
40.4.1	站点地图	1286	41.3.3	返回数据	1322
40.4.2	Menu 控件	1286	41.4	创建视图	1323
40.4.3	菜单路径	1287	41.4.1	向视图传递数据	1325
40.5	验证用户输入	1287	41.4.2	Razor 语法	1325
40.5.1	使用验证控件	1287	41.4.3	强类型视图	1326
40.5.2	使用验证摘要	1289	41.4.4	布局	1327
40.5.3	验证组	1289	41.4.5	部分视图	1330
40.6	访问数据	1290	41.5	从客户端提交数据	1334
40.6.1	使用 Entity Framework	1291	41.5.1	模型绑定器	1335
40.6.2	使用 Entity Data Source	1291	41.5.2	注释和验证	1337
40.6.3	排序和编辑	1293	41.6	HTML Helper	1338
40.6.4	定制列	1294	41.6.1	简单的 Helper	1338
40.6.5	在网格中使用模板	1295	41.6.2	使用模型数据	1339
40.6.6	定制对象上下文的 创建过程	1297	41.6.3	定义 HTML 特性	1340
40.6.7	对象数据源	1298	41.6.4	创建列表	1340
40.7	安全性	1299	41.6.5	强类型化的 Helper	1341
40.7.1	启用表单身份验证	1299	41.6.6	编辑器扩展	1342
40.7.2	登录控件	1300	41.6.7	创建自定义 Helper	1342
40.8	Ajax	1301	41.6.8	模板	1343
40.8.1	ASP.NET AJAX 的 概念	1302	41.7	创建数据驱动的应用程序	1344
40.8.2	ASP.NET AJAX 网站示例	1305	41.7.1	定义模型	1344
40.8.3	支持 ASP.NET AJAX 的网站配置	1308	41.7.2	创建控制器和视图	1345
40.8.4	添加 ASP.NET AJAX 功能	1308	41.8	动作过滤器	1350
40.9	小结	1315	41.9	身份验证和授权	1352
第 41 章	ASP.NET MVC	1316	41.9.1	登录模型	1352
41.1	ASP.NET MVC 概述	1316	41.9.2	登录控制器	1352
41.2	定义路由	1318	41.9.3	登录视图	1354
41.2.1	添加路由	1319	41.10	ASP.NET Web API	1355
			41.10.1	使用 Entity Framework Code-First 进行 数据访问	1355
			41.10.2	为 ASP.NET Web API 定义路由	1357
			41.10.3	控制器实现	1357

41.10.4 使用 jQuery 的客户端 应用程序 .....	1358	43.4 服务的行为 .....	1398
41.11 小结 .....	1360	43.5 绑定 .....	1401
<b>第 42 章 ASP.NET 动态数据 .....</b>	<b>1361</b>	43.5.1 标准的绑定 .....	1401
42.1 概述 .....	1361	43.5.2 标准绑定的特性 .....	1402
42.2 创建动态数据 Web 应用程序 .....	1362	43.5.3 Web 套接字 .....	1404
42.2.1 配置 Scaffolding .....	1363	43.6 宿主 .....	1407
42.2.2 查看结果 .....	1364	43.6.1 自定义宿主 .....	1407
42.3 定制动态数据网站 .....	1366	43.6.2 WAS 宿主 .....	1408
42.3.1 控制框架 .....	1367	43.6.3 预配置的宿主类 .....	1408
42.3.2 定制模板 .....	1368	43.7 客户端 .....	1410
42.3.3 配置路由 .....	1373	43.7.1 使用元数据 .....	1410
42.4 小结 .....	1374	43.7.2 共享类型 .....	1411
<b>第 VI 部分 通 信</b>		43.8 双工通信 .....	1411
<b>第 43 章 WCF .....</b>	<b>1376</b>	43.8.1 双工通信的协定 .....	1412
43.1 WCF 概述 .....	1376	43.8.2 双工通信的服务 .....	1412
43.1.1 SOAP .....	1378	43.8.3 双工通信的客户端 应用程序 .....	1413
43.1.2 WSDL .....	1378	43.9 路由 .....	1414
43.1.3 REST .....	1379	43.9.1 示例应用程序 .....	1415
43.1.4 JSON .....	1379	43.9.2 路由接口 .....	1416
43.2 创建简单的服务和客户端 .....	1379	43.9.3 WCF 路由服务 .....	1416
43.2.1 定义服务和数据协定 .....	1380	43.9.4 为失败使用路由器 .....	1417
43.2.2 数据访问 .....	1382	43.9.5 改变协定的桥梁 .....	1418
43.2.3 服务的实现 .....	1383	43.9.6 过滤器的类型 .....	1419
43.2.4 WCF 服务宿主和 WCF 测试客户端 .....	1384	43.10 小结 .....	1419
43.2.5 自定义服务宿主 .....	1386	<b>第 44 章 WCF 数据服务 .....</b>	<b>1420</b>
43.2.6 WCF 客户端 .....	1388	44.1 概述 .....	1420
43.2.7 诊断 .....	1390	44.2 包含 CLR 对象的 自定义宿主 .....	1421
43.2.8 与客户端共享协定 程序集 .....	1392	44.2.1 CLR 对象 .....	1422
43.3 协定 .....	1393	44.2.2 数据模型 .....	1424
43.3.1 数据协定 .....	1394	44.2.3 数据服务 .....	1424
43.3.2 版本问题 .....	1394	44.2.4 驻留服务 .....	1425
43.3.3 服务协定 .....	1395	44.2.5 其他服务操作 .....	1426
43.3.4 消息协定 .....	1396	44.3 HTTP 客户端应用程序 .....	1426
43.3.5 错误协定 .....	1396	44.4 使用 WCF 数据服务和 ADO.NET Entity Framework .....	1431

44.4.1	ASP.NET 宿主和 EDM	1432	47.1.1	使用消息队列的场合	1483
44.4.2	使用 WCF 数据服务 客户库	1433	47.1.2	消息队列功能	1484
44.5	小结	1441	47.2	Message Queuing 产品	1485
第 45 章	Windows WF 4	1442	47.3	消息队列体系结构	1486
45.1	工作流概述	1442	47.3.1	消息	1486
45.2	Hello World 示例	1443	47.3.2	消息队列	1486
45.3	活动	1444	47.4	Message Queuing 管理工具	1487
45.3.1	If 活动	1445	47.4.1	创建消息队列	1487
45.3.2	InvokeMethod 活动	1446	47.4.2	消息队列属性	1488
45.3.3	Parallel 活动	1446	47.5	消息队列的编程实现	1489
45.3.4	Delay 活动	1447	47.5.1	创建消息队列	1489
45.3.5	Pick 活动	1447	47.5.2	查找队列	1490
45.4	自定义活动	1448	47.5.3	打开已知队列	1490
45.4.1	活动的验证	1449	47.5.4	发送消息	1492
45.4.2	设计器	1450	47.5.5	接收消息	1494
45.4.3	自定义复合活动	1452	47.6	课程订单应用程序	1496
45.5	工作流	1454	47.6.1	课程订单类库	1496
45.5.1	实参和变量	1455	47.6.2	课程订单消息发送 程序	1499
45.5.2	WorkflowApplication	1455	47.6.3	发送优先级和可恢复 的消息	1501
45.5.3	存放 WCF 工作流	1459	47.6.4	课程订单消息接收 应用程序	1502
45.5.4	工作流的版本	1463	47.7	接收结果	1508
45.5.5	驻留设计器	1464	47.7.1	确认队列	1508
45.6	小结	1468	47.7.2	响应队列	1509
第 46 章	对等网络	1469	47.8	事务队列	1509
46.1	P2P 网络概述	1469	47.9	消息队列和 WCF	1510
46.1.1	客户端-服务器 体系结构	1469	47.9.1	带数据协定的实体类	1511
46.1.2	P2P 体系结构	1470	47.9.2	WCF 服务协定	1512
46.1.3	P2P 体系结构的挑战	1471	47.9.3	WCF 消息接收 应用程序	1513
46.1.4	P2P 术语	1472	47.9.4	WCF 消息发送 应用程序	1515
46.1.5	P2P 解决方案	1472	47.10	消息队列的安装	1517
46.2	PNRP	1472	47.11	小结	1517
46.3	构建 P2P 应用程序	1475			
46.4	小结	1481			
第 47 章	消息队列	1482			
47.1	概述	1482			

# 第 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 异步编程
- 第 14 章 内存管理和指针
- 第 15 章 反射
- 第 16 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)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

本章源代码下载地址([wrox.com](http://wrox.com)):

本章没有可供下载的代码。

### 1.1 C#与.NET 的关系

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。

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(使用 C#或其他语言编写的代码)之前，需要编译它们。在.NET 中，编译分为两个阶段：

(1) 将源代码编译为 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。不过，现在已经有了.NET 的一个部分跨平台实现(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

前面对 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 high 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 2012

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 .NET 当成一种新语言。已有的 Visual Basic 6 代码不能编译为当前的 Visual Basic 2012 代码(或 Visual Basic .NET 2002、2003、2005、2008 和 2012 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 2012(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 2012 中，Visual Studio 2012 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 2012 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 2012 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 2012 编译为本地可执行代码了。Visual Basic 2012 只编译为中间语言，就像 C# 一样。如果需要使用 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++ 2012

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 2012 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。

编写使用 .NET 的 C++ 程序会得到几种不同的互操作场景。使用编译器设置 /clr 启用公共语言运行库支持时，就可以完全混合所有的本地和托管 C++ 功能。其他选项(如 /clr:safe 和 /clr:pure)可以限制 C++ 指针的使用，从而像使用 C# 和 Visual Basic 那样编写安全的代码。

Visual C++ 2012 允许为 Windows 8 的 Windows Runtime(WinRT)创建程序。在这样的程序中，C++ 不使用托管代码，而是本地访问 WinRT。

### 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，因为它包含一些 .NET 不具备的特性。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 23 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。

### 4. Windows 运行库

Windows 8 提供了一种新的运行库，可被新应用程序使用。这个运行库可在 Visual Basic、C#、C++ 和 JavaScript 中使用。用在不同的环境中时，它会发生相应的变化。例如，在 C# 中使用时，它看起来就像 .NET Framework 中的类；在 JavaScript 中使用时，它看起来就像 JavaScript 开发人员所惯用的 JavaScript 库；而在 C++ 中使用时，它又像是一个 C++ 标准库。这种多样性是通过使用语言投影实现的。第 31 章将讨论 Windows 运行库以及在 C# 中如何使用它。

## 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。

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, 而无须考虑它们的类型, 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 执行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 2012 类中的一个方法定义为返回一个 Integer——Visual Basic 2012 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 2012 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 2012 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 2012 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。

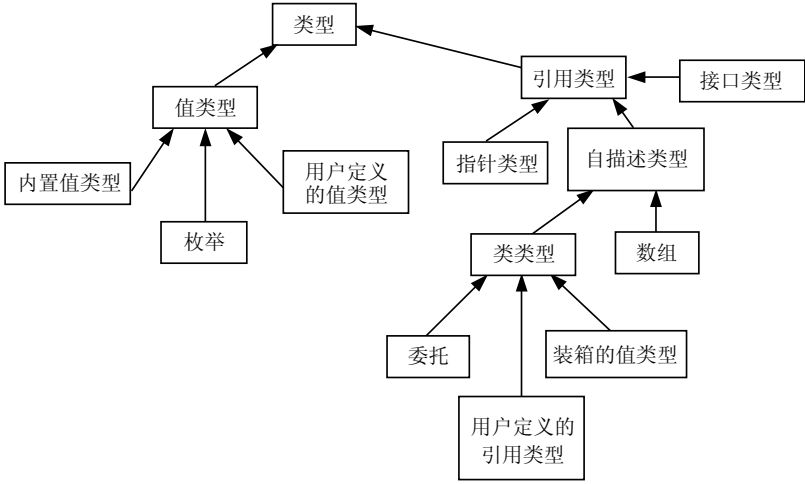


图 1-1

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 2012 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 2012 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 2012 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。这在测试时很有帮助,但是在正常的程序中不应该这么做。

垃圾回收过程的详细信息可参见第 14 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，也可以使用基于代码的安全性，指定这段代码不能执行管理员账户一般可以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 22 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。

进程不仅是运行代码的实例相互隔离的一种方式，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。

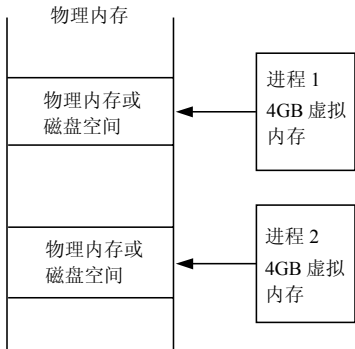


图 1-2

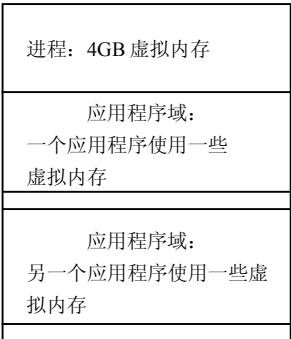


图 1-3

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(除非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 有更严格的强类型系统,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#支持 finally 块,这是许多 C++开发人员长久以来一直希望 C++也能够提供的一种功能。

第 16 章会详细讨论异常。简要地说,代码的某些部分被看成异常处理例程,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 2012 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 16 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 2012 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 19 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 19 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私有程序集。

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。只需要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 15 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 允许利用目前出现的所有多核处理器。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死锁。

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 21 章。

### 1.4.5 异步编程

C# 5.0 以 Task Parallel Library 中的 Task 为基础，提供了新的异步功能。自从 .NET 1.0 以来，.NET Framework 中的许多类都同时提供了同步和异步版本。当用户界面线程在执行需要花费一些时间的任务时，是不应该被阻塞的。如果看到过不响应的程序，就会知道那是很烦人的。但是，那些异步方法的缺点是很难使用。对应的同步版本在编写程序时很方便，所以更加常用。

使用多年鼠标的用户会习惯了延迟。移动对象或者使用滚动条时，延迟是很常见的。但是，对于触摸界面，延迟会造成很糟糕的用户体验。这可以通过调用异步方法来解决。如果 WinRT 中的某个方法需要超过 50ms 才能完成，那么 WinRT 会只提供异步方法调用。

在 C# 5.0 中调用新的异步方法是很简单的。C# 5.0 定义了两个新的关键字：async 和 await。第 13 章将介绍这两个关键字和它们的用法。

## 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，它们的库也都很容易使用，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 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 2012 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。

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 4.5 的 .NET 基类库中的各种类，即各种基类是如何工作的。.NET 4.5 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 UI 支持和控件(参见第 35 章~第 38 章)
- 在 ASP.NET 中使用 Web 窗体和 MVC(参见第 39 章~第 42 章)
- 使用 ADO.NET 和 XML 进行数据访问(参见第 32 章~第 34 章)
- 文件系统和注册表访问(参见第 24 章)
- 网络和 Web 浏览(参见第 26 章)
- .NET 特性和反射(参见第 14 章)
- COM 互操作性(参见第 23 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.ProjectName 中，则这个类的全名就是 YourCompanyName.ProjectName.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.Sales-Services.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。在对类库进行单元测试、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 4.5 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 4.5 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 39~42 章。

#### 1. ASP.NET 的功能

ASP.NET 最初被设计出来时，只提供了 ASP.NET Web 窗体，其目标是按照 Windows 应用程序开发人员编写应用程序的方式轻松地创建 Web 应用程序，是不必编写 HTML 和 JavaScript 的。

现在情况发生了变化。HTML 和 JavaScript 重新焕发了生机，再次变得重要起来。相应地，ASP.NET 中有了一个新的框架，不只方便了编写 HTML 和 JavaScript，还基于流行的 MVC 模式提供了代码的分离，从而更便于进行单元测试。这个框架就是 ASP.NET MVC。

重构后的 ASP.NET 为 ASP.NET Web 窗体和 ASP.NET MVC 打下了基础，而且 UI 框架也构建在这个基础之上。



第 39 章将介绍 ASP.NET 打下的基础。

#### 2. ASP.NET Web 窗体

为了简化 Web 页面的结构，Visual Studio 2012 提供了 Web 窗体。它们允许以图形化方式建立 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，并把这个类看成代码隐藏的 ASP.NET 页面。当然不是必须使用 C#创建 Web 窗体，也可以使用 Visual Basic 2012 或另一种.NET 语言来创建。

ASP.NET Web 窗体提供了丰富的功能，使用它的控件不只可以创建简单的代码，还能够利用 JavaScript 和服务端验证逻辑进行输入验证，以及使用网格和数据源来访问数据库。ASP.NET 的 Web 窗体还提供了 Ajax 功能，允许在客户端动态渲染页面的某个部分。



第 40 章将详细讨论 ASP.NET Web 窗体。

### 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 2012 扩展 Web Form 工具箱。创建一个新服务器端控件只需要实现.NET 的 System.Web.UI.WebControls.WebControl 类而已。

### 4. ASP.NET MVC

Visual Studio 自带了 ASP.NET MVC 4。这种技术已经发展到第 4 个版本了。Web 窗体采取的做法是为开发人员抽象掉了 HTML 和 JavaScript，但是随着 HTML 5 和 jQuery 的出现，使用这些技术再次变得重要起来。ASP.NET MVC 将重点放到了在模型和控制器中单独编写服务器端代码，而在使用视图时只用少量服务器端代码从控制器中获取信息。这种功能分离使得编写单元测试变得简单，并且让开发人员能够充分利用 HTML 5 和 JavaScript 库。



第 41 章介绍了 ASP.NET MVC。

### 5. ASP.NET 动态数据

使用 ASP.NET 动态数据(Dynamic Data)可以快速创建数据驱动的 Web 应用程序。借助于 Entity Framework 和基架选项，可以高效快速地创建能够读写数据的窗体。ASP.NET 动态数据并不是创建窗体的一种一站式方法，开发人员还可以定制窗体和窗体字段，并提供用于输入数据的类。



第 42 章介绍了 ASP.NET 动态数据。

### 6. ASP.NET Web API

ASP.NET Web API 是在客户端和服务器之间进行简单通信的一种新方式(REST 风格)。这种新框架基于 ASP.NET MVC，并使用了控制器和路由。客户端可以收到符合开放数据(Open Data)规范的 JSON 或 Atom 数据。

这个新 API 具备的一些特征使得不只 Web 客户端很容易在 JavaScript 中使用它，而且在 Windows 8 应用程序中也很容易使用。



因为 ASP.NET Web API 以 ASP.NET MVC 为基础，所以这种技术也在第 41 章中介绍。



### 1.7.2 使用 WPF

有两种技术可以用于创建 Windows 桌面应用程序：Windows 窗体和 Windows Presentation Foundation(WPF)。Windows 窗体包含的类只是封装了原生 Windows 控件，所以是基于像素图形的。WPF 则是基于矢量图的一种新技术。

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。要运行 WPF 应用程序，需要在客户机上至少安装 .NET Framework 3.0。当然，更新版本的 .NET Framework 会提供新的 WPF 功能。例如，.NET 4.5 中新增了功能区控件和实时造形等功能。

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 29 章介绍了 XAML(XML Paper Specification、Windows Workflow Founding 和 Windows Communication Foundation 也使用了 XAML)。

第 35 章详细介绍了如何使用 XAML 和 C# 构建 WPF 应用程序。第 36 章详细介绍了如何使用 WPF 和 XAML 创建数据驱动的业务应用程序。打印和创建文档是 WPF 的另外一个重要方面，第 37 章将进行讨论。

### 1.7.3 Windows 8 应用程序

Windows 8 用“触摸为先”的应用程序开启了一种新的范式。桌面应用程序通常会提供一个菜单和一个工具栏，用户在应用程序的一个框架中查看下一步可以做什么。Windows 8 应用程序则将重点放到了内容。应用程序的框架应该缩减到最低，只提供用户与内容交互所需的任务，而不是提供他们可以使用的不同选项。关注点应是当前的任务，而不是用户下一步可能要执行的操作。这样一来，用户就会根据内容记住应用程序的用途。“有内容、无框架”是这种技术的口号。

可以使用 C# 和 XAML，结合 Windows Runtime 和 .NET Framework 的一个子集编写 Windows 8 应用程序。Windows 8 应用程序为开发人员提供了广阔的新世界。其主要缺点是只能运行在 Windows 8 或更高版本的操作系统上。



第 38 章将详细介绍 Windows 8 应用程序。

### 1.7.4 Windows 服务

Windows 服务(最初称为 NT 服务)是一个在基于 Windows NT 内核的操作系统上后台运行的程序。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，它们监听来自客户端的 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 .NET 允许创建 C# Windows Service 项目，为基本

Windows 服务编写 C#源代码。第 27 章将详细介绍如何编写 C# Windows 服务。

### 1.7.5 WCF

ASP.NET Web API 可以实现客户端和服务端之间的通信，它使用起来十分简单，但是功能不如 SOAP 协议丰富。

WCF 是一种功能丰富的技术，提供了多种通信选项。使用 WCF 时，既可以使用基于 REST 的通信，也可以使用基于 SOAP 的通信，都能获得标准 Web 服务(如安全性、事务、双向和单向通信、路由、发现等)提供的所有功能。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。第 44 章和第 47 章也会讲到一些基于 WCF 的技术，例如 WCF Data Services 和 Message Queueing with WCF。

### 1.7.6 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，自从 .NET 4 以外许多人都发现它更容易使用了。.NET 4.5 也对它做了一点小改进。Visual Studio 2012 在使用 WF 方面有了长足的进步，并使得使用 C#(原来的版本使用 VB)构建工作流和编写表达式变得更简单。WF 有一个新的状态机设计器和一些新活动。



第 45 章将详细讨论 WF。

## 1.8 C#在.NET 企业体系结构中的作用

新技术总在快速出现。应该为企业应用程序使用哪种技术呢？有许多因素影响所要做出的决定。例如，如果现有应用程序是开发人员根据现有知识开发的，应该怎么办？可以在遗留应用程序中集成新功能吗？根据需要的维护量，可能重新构建一些现有的应用程序来使用新功能更加合理。通常，遗留应用程序和新应用程序是可以共存很长一段时间的。应用程序对客户端有什么需求？需要把 .NET Framework 升级到 4.5 版本吗？还是 2.0 版本就够了？或者，客户端能够使用 .NET 吗？

要做的决定很多，.NET 也提供了很多选项。可以在客户端的 Windows 窗体、WPF 或 Windows 8 应用程序中使用 .NET，也可以在使用 IIS 和 ASP.NET 运行库托管的 Web 服务器上的 ASP.NET Web 窗体或 ASP.NET MVC 中使用 .NET。服务可以运行在 IIS 内，也可以托管在 Windows 服务内。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 和其他带有数据提供程序的数据库。使用 ADO.NET Entity Framework 很容易将数据库关系映射为对象层次结构。这不只适用于 SQL Server，也适用于许多提供了 Entity Framework 提供程序的不同数据库。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 或 JSON，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 C#是第一个基于组件的 C 语言，所以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。

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 2012 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。

在 SOA 和基于服务的编程热潮过后，现在使用服务已经成为了一种业界常规。新的趋势是基于云的编程，Microsoft 为此提供了 Windows Azure。可以在本地服务器或者云中的 ASP.NET Web 窗体、ASP.NET Web API 或 WCF 中运行.NET 应用程序。客户端则可以使用 HTML 5 来接触更多的受众，或者使用 WPF 或 Windows 8 应用程序来实现丰富的功能。.NET 仍然能够跟得上新技术，利用新选项，所以生机是无限的。

## 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
- 垃圾回收

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。

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，如垃圾回收和安全性。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 和 WPF)的应用程序的基础。

第 2 章将介绍如何用 C#语言编写代码。

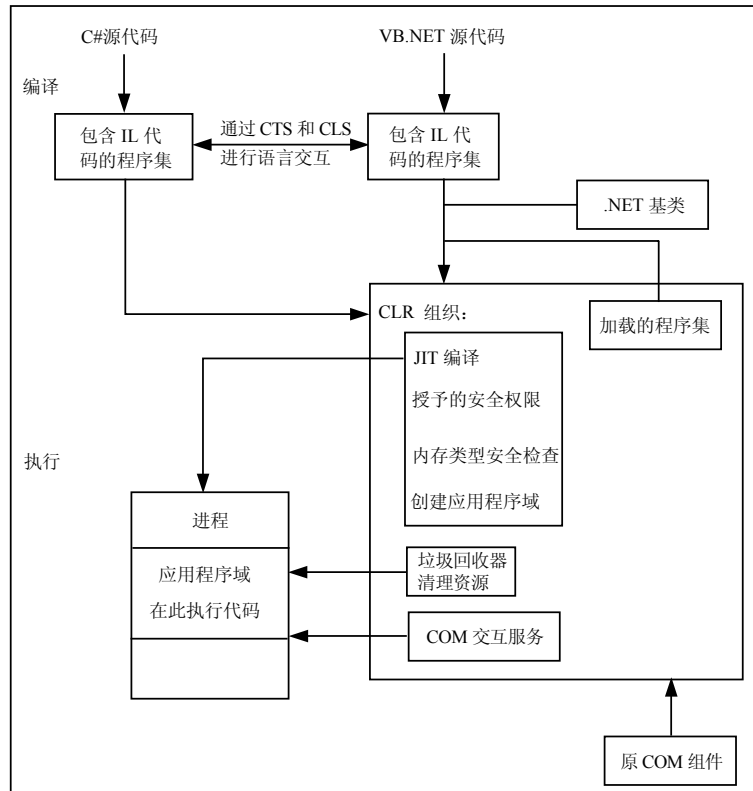


图 1-4

# 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

### 本章源代码下载地址(wrox.com):

打开网页 <http://www.wrox.com/remtitle.cgi?isbn=1118314425>, 单击 Download Code 选项卡即可下载本章源代码。本章代码分为以下几个主要的示例文件:

- ArgsExample.cs
- DoubleMain.cs
- ElseIf.cs
- First.cs
- MathClient.cs
- MathLibrary.cs
- NestedFor.cs
- Scope.cs
- ScopeBad.cs
- ScopeTest2.cs
- StringExample.cs
- Var.c.s

## 2.1 C#基础

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。

## 2.2 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 2012 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 17 章之前避免涉及 Visual Studio 2012。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。

### 2.2.1 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;
        }
    }
}
```

### 2.2.2 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 2011)的方式,这里显示的结果可能与你计算

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 2011%\Common7\Tools\vsvars32.bat`。其中 `%Microsoft Visual Studio 2011%` 是 Visual Studio 2011 的安装文件夹。第 2 种方法(更简单)是使用 Visual Studio 2011 命令提示符代替通常的命令提示符窗口。Visual Studio 2011 命令提示符在菜单“开始”|“程序”|Microsoft Visual Studio 2011|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# Compiler version 4.0.30319.17379
for Microsoft(R) .NET Framework 4.5
Copyright (C) Microsoft Corporation. All rights reserved.
```

```
First.exe
Hello from Wrox.
```

### 2.2.3 详细介绍

首先对 C# 语法做几个一般性的解释。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。如果你以前使用的是 Visual Basic，就应特别注意每条语句结尾的分号。对于新接触 C 风格语言的用户，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 2011 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。

## 2.3 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。

### 2.3.1 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` 关键字把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
```

### 2.3.2 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());
}
}
```

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。

### 2.3.3 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同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：

```
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;
        }
    }
}
```

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：

```
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;
}
```

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：

```
using System;
namespace Wrox
{
    class ScopeTest2
    {
        static int j = 20;
        public static void Main()
        {
            int j = 30;
            Console.WriteLine(j);
            return;
        }
    }
}
```

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()` 方法终止，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。

### 2.3.4 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。

## 2.4 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。

### 2.4.1 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 **两个 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 实现一种精细的算法，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。这是通过垃圾回收器实现的。

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。

## 2.4.2 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**)。

### 2.4.3 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	$-128 \sim 127 (-2^7 \sim 2^7 - 1)$
short	System.Int16	16 位有符号的整数	$-32\,768 \sim 32\,767 (-2^{15} \sim 2^{15} - 1)$
int	System.Int32	32 位有符号的整数	$-2\,147\,483\,648 \sim 2\,147\,483\,647 (-2^{31} \sim 2^{31} - 1)$
long	System.Int64	64 位有符号的整数	$-9\,223\,372\,036\,854\,775\,808 \sim 9\,223\,372\,036\,854\,775\,807 (-2^{63} \sim 2^{63} - 1)$
byte	System.Byte	8 位无符号的整数	$0 \sim 255 (0 \sim 2^8 - 1)$
ushort	System.UInt16	16 位无符号的整数	$0 \sim 65\,535 (0 \sim 2^{16} - 1)$
uint	System.UInt32	32 位无符号的整数	$0 \sim 4\,294\,967\,295 (0 \sim 2^{32} - 1)$
ulong	System.UInt64	64 位无符号的整数	$0 \sim 18\,446\,744\,073\,709\,551\,615 (0 \sim 2^{64} - 1)$

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	$\pm 1.5 \times 10^{245} \sim \pm 3.4 \times 10^{38}$
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)
decimal	System.Decimal	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
bool	System.Boolean	表示 true 或 false	NA	true 或 false

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
char	System.Char	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

2.4.4 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` 类型实现了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实现代码。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` 与引用类型的常见行为有一些区别。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：

```
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;
    }
}
```

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` 类型，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没有经过转义的反斜杠字符，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.
```

## 2.5 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。

### 2.5.1 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。

```
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);
        }
    }
}
```

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，就无须使用花括号：

```

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；只需要使用 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。这包括值相同的不同常量,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。

## 2.5.2 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.
}
```



也可以在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：

```
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();  
            }  
        }  
    }  
}
```

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。

### 2.5.3 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 16 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。

## 2.6 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输入。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递给方法，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, 在语法上把枚举当作结构不会造成性能损失。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。

## 2.7 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..
    }
}
```

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.ProjectName.SystemSection**。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。

### 2.7.1 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 nSE = new Basics.NamespaceExample();
            // do something with the nSE 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。

### 2.7.2 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;
        }
    }
}
```

## 2.8 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。

### 2.8.1 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`：

```
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;
    }
}
```

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
```

## 2.8.2 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：

```
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;
        }
    }
}
```

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
```

## 2.9 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
<code>/t:exe</code>	控制台应用程序 (默认)
<code>/t:library</code>	带有清单的类库
<code>/t:module</code>	没有清单的组件
<code>/t:winexe</code>	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;
        }
    }
}
```

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：

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

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：在命令提示符下输入 `MathClient`，其结果是显示数字 15——加运算的结果。

## 2.10 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。

## 2.11 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。注释有助于阅读代码的其他开发人员理解代码，而且可以用来为开发人员生成代码的文档。

### 2.11.1 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 single-line 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 .*/";
```

## 2.11.2 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
<c>	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
<para>	建立文档的结构
<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)

(续表)

标 记	说 明
<summary>	提供类型或成员的简短小结
<typeparam>	用在泛型类型的注释中以说明一个类型参数
<typepararef>	类型参数的名称
<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 MathLib
    {
        ///<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。

```
<?xml version="1.0"?>
<doc>
  <assembly>
    <name>MathLibrary</name>
  </assembly>
  <members>
    <member name="T:Wrox.MathLibrary">
      <summary>
        Wrox.MathLibrary class.
      </summary>
    </member>
  </members>
</doc>
```

```

        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>

```

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。

## 2.12 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。

### 2.12.1 #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。

### 2.12.2 `#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
```

### 2.12.3 #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` 语句可以提醒自己执行某个操作：

```

#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.*");

```

### 2.12.4 #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 17 章会详细介绍它们。

### 2.12.5 #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
```

### 2.12.6 #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
```

## 2.13 C#编程规则

本节介绍编写 C# 程序时应该遵循的准则。大多数 C# 开发人员都遵守这些规则，所以在这些规则的指导下编写程序可以方便其他开发人员使用程序的代码。

### 2.13.1 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

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

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。

### 2.13.2 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。

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, 但是 Microsoft 编写的 C# 和整个 .NET Framework 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
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

(续表)

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
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



(续表)

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

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明显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。

## 2.14 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
- 弱引用
- Object 类, 其他类型都从该类派生而来

本章源代码下载地址([wrox.com](http://www.wrox.com)):

打开网页 <http://www.wrox.com/remtitle.cgi?isbn=1118314425>, 单击 Download Code 选项卡即可下载本章源代码。本章代码分为以下几个主要的示例文件:

- MathTest
- MathTestWeakReference
- ParameterTest

### 3.1 创建及使用类

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, 但假定你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。

## 3.2 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。

## 3.3 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。

### 3.3.1 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和变量与类的关联方式相同。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。

### 3.3.2 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，虚拟的对象被当作实际的东西。
- 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 14 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，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、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`，该类包含几个方法和一个字段。

```
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;
    }
}
```

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，除非特别指定，所有的引用类型都通过引用传递，所有的值类型都通过值来传递。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` 说明了用作参数的值类型和引用类型的区别：

```
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;
        }
    }
}
```

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，只需要声明同名但参数个数或类型不同的方法即可：

```
class ResultDisplay
{
    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：它是一个方法或一对方法，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作为下划线的字段名，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;}
```

### 内联

一些开发人员可能会担心，前面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，到目前为止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` 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余个调用。

### 3.3.3 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。

### 3.4 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。

## 3.5 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，但性能会有一定的损失。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，并存储每件家具的尺寸。表面看来使字段变为公共字段会违背编程规则，但这里的关键是我们实际上并不需要类的全部功能。现在只有两个数字，把它们当成一对来处理，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布局(第15章在介绍特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。

### 3.5.1 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，不需要等待垃圾回收。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。

### 3.5.2 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`。

### 3.5.3 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，但其原因隐藏在 .NET 运行库的实现方式中。在一些极罕见的情况中，.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 干脆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。第 14 章将讨论 `Dispose()` 方法。

## 3.6 弱引用

在应用程序代码内实例化一个类或结构时，只要有代码引用它，就会形成强引用。例如，如果有一个类 `MyClass()`，并创建了一个变量 `myClassVariable` 来引用该类的对象，那么只要 `myClassVariable` 在作用域内，就存在对 `MyClass` 对象的强引用，如下所示：

```
MyClass myClassVariable = new MyClass();
```

这意味着垃圾回收器不会清理 `MyClass` 对象使用的内存。一般而言这是好事，因为可能需要访问 `MyClass` 对象，但是如果 `MyClass` 对象很大，并且不经常访问呢？此时可以创建对象的弱引用。

弱引用允许创建和使用对象，但是垃圾回收器运行时(第 14 章将介绍垃圾回收)，就会回收对象并释放内存。由于存在潜在的 bug 和性能问题，一般不会这么做，但是在特定的情况下使用弱引用是很合理的。

弱引用是使用 `WeakReference` 类创建的。因为对象可能在任意时刻被回收，所以在引用该对象前必须确认它存在。以前面的 `MathTest` 类为例，这次使用 `WeakReference` 类创建对它的弱引用：

```
static void Main()
{
    // Instantiate a weak reference to MathTest object
    WeakReference mathReference = new WeakReference(new MathTest());
    MathTest math;
    math = mathReference.Target as MathTest;
    if(math != null)
    {
        math.Value = 30;
        Console.WriteLine("Value field of math variable contains " + math.Value);
        Console.WriteLine("Square of 30 is " + math.GetSquare());
    }
    else
    {
        Console.WriteLine("Reference is not available.");
    }

    GC.Collect();

    if(mathReference.IsAlive)
    {
        math = mathReference.Target as MathTest;
    }
    else
    {
        Console.WriteLine("Reference is not available.");
    }
}
```

创建 `mathReference` 时，会向其构造函数传递一个新的 `MathTest` 对象。`MathTest` 对象成为了 `WeakReference` 对象的目标。想要使用 `MathTest` 对象时，就需要先检查 `mathReference` 对象以确保其未被回收。`IsAlive` 属性就用于这个目的。如果 `IsAlive` 为 `true`，就从目标属性得到 `MathTest` 对象的引用。注意，因为 `Target` 属性返回的是 `Object` 类型，所以必须将其强制转换为 `MathTest` 类型。

然后，调用垃圾回收器(`GC.Collect()`)，并尝试再次获得 `MathTest` 对象。这一次，`IsAlive` 属性返回 `false`，如果确实想要使用 `MathTest` 对象，就必须实例化一个新的 `MathTest` 对象。

## 3.7 部分类

`partial` 关键字允许把类、结构、方法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()
    {
    }
}

```

### 3.8 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();
```

### 3.9 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。

### 3.9.1 `System.Object()` 方法

下面将简要总结每个方法的作用，3.9.2 小节详细论述 `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：注意有 3 个用于比较对象相等性的不同方法，这说明 .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。`Object` 中实现的 `Finalize()` 方法实际上什么也没有做，因而被垃圾回收器忽略。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 15 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。这个方法是受保护的，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。

### 3.9.2 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过的代码，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
```

### 3.10 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。

### 3.11 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。