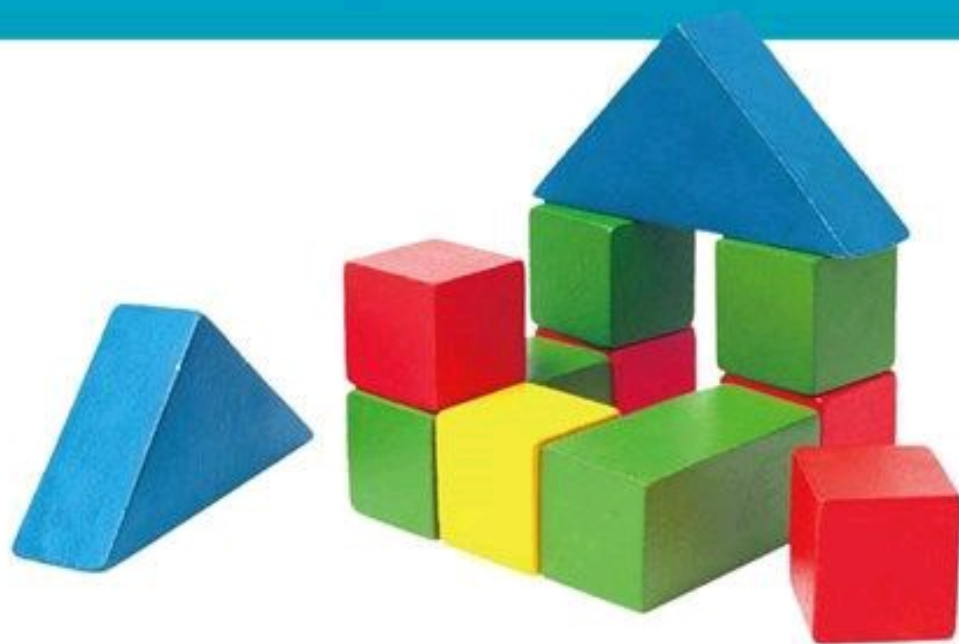


初学者指南

C# A Beginner's Tutorial

[加拿大] Jayden Ky 著 李强 吴戈 译



目录

- [版权信息](#)
- [版权声明](#)
- [内容提要](#)
- [前言](#)
- [.NET Framework概述](#)
- [面向对象编程的概述](#)
- [OOP的优点](#)
- [OOP很难吗](#)
- [关于本书](#)
- [下载和安装.NET Framework](#)
- [选择一个IDE](#)
- [下载程序示例](#)
- [第1章 初识C#](#)
 - [1.1 第一个C#程序](#)
 - [1.1.1 启动IDE](#)
 - [1.1.2 编写C#程序](#)
 - [1.1.3 编译和运行C#程序](#)
 - [1.2 C# 编码惯例](#)
 - [1.3 小结](#)
- [第2章 语言基础](#)
 - [2.1 ASCII和Unicode](#)
 - [2.2 内建类型和通用类型系统](#)
 - [2.3 变量](#)
 - [2.4 常量](#)
 - [2.5 直接量](#)
 - [2.5.1 整型直接量](#)
 - [2.5.2 浮点型直接量](#)
 - [2.5.3 布尔型直接量](#)
 - [2.5.4 字符型直接量](#)
 - [2.6 基本类型转换](#)
 - [2.6.1 宽化转换](#)
 - [2.6.2 窄化转换](#)
 - [2.7 运算符](#)

- [2.7.1 一元运算符](#)
- [2.7.2 算术运算符](#)
- [2.7.3 关系运算符](#)
- [2.7.4 条件运算符](#)
- [2.7.5 位移运算符](#)
- [2.7.6 赋值运算符](#)
- [2.7.7 整型位运算符 & | ^](#)
- [2.7.8 逻辑运算符 & | ^](#)
- [2.7.9 运算符优先级](#)
- [2.7.10 提升](#)
- [2.8 注释](#)
- [2.9 小结](#)
- [第3章 语句](#)
- [3.1 C#语句概览](#)
- [3.2 if语句](#)
- [3.3 while语句](#)
- [3.4 do-while语句](#)
- [3.5 for语句](#)
- [3.6 break语句](#)
- [3.7 continue语句](#)
- [3.8 switch语句](#)
- [3.9 小结](#)
- [第4章 对象与类](#)
- [4.1 C#对象是什么](#)
- [4.2 C#类](#)
 - [4.2.1 字段](#)
 - [4.2.2 方法](#)
 - [4.2.3 main方法](#)
 - [4.2.4 构造函数](#)
 - [4.2.5 UML类图中的类成员](#)
- [4.3 创建对象](#)
- [4.4 null关键字](#)
- [4.5 内存中的对象](#)
- [4.6 C#命名空间](#)
- [4.7 封装和类的访问控制](#)
- [4.8 关键字this](#)
- [4.9 使用其他类](#)

[4.10 静态成员](#)

[4.11 变量作用字段](#)

[4.12 方法重载](#)

[4.13 小结](#)

[第5章 核心类](#)

[5.1 System.Object](#)

[5.2 System.String](#)

[5.2.1 字符串连接](#)

[5.2.2 比较两个字符串](#)

[5.2.3 字符串直接量](#)

[5.2.4 转义特定字符](#)

[5.2.5 String类的属性](#)

[5.2.6 String类的方法](#)

[5.3 System.Text.StringBuilder](#)

[5.3.1 StringBuilder类的构造函数](#)

[5.3.2 StringBuilder类的属性](#)

[5.3.3 StringBuilder类的方法](#)

[5.4 数组](#)

[5.4.1 遍历数组](#)

[5.4.2 改变数组的大小](#)

[5.4.3 为Main传递一个字符串数组](#)

[5.5 System.Console](#)

[5.6 小结](#)

[第6章 继承](#)

[6.1 继承概述](#)

[6.1.1 扩展一个类](#)

[6.1.2 is-a关系](#)

[6.2 可访问性](#)

[6.3 方法覆盖](#)

[6.4 调用基类的构造函数](#)

[6.5 调用基类的隐藏成员](#)

[6.6 类型转换](#)

[6.7 密封类](#)

[6.8 关键字is](#)

[6.9 小结](#)

[第7章 结构](#)

[7.1 结构概述](#)

[7.2 .NET结构](#)

[7.3 编写一个结构](#)

[7.4 可为空的类型](#)

[7.5 小结](#)

[第8章 错误处理](#)

[8.1 捕获异常](#)

[8.2 没有catch的try语句和using语句](#)

[8.3 System.Exception类](#)

[8.4 从方法中抛出异常](#)

[8.5 异常处理中的最后注意事项](#)

[8.6 小结](#)

[第9章 数字和日期](#)

[9.1 数字解析](#)

[9.2 数字格式化](#)

[9.3 System.Math类](#)

[9.4 使用Date和Time](#)

[9.4.1 System.DateTime](#)

[9.4.2 System.TimeSpan](#)

[9.5 小结](#)

[第10章 接口和抽象类](#)

[10.1 接口的概念](#)

[10.2 从技术角度看接口](#)

[10.3 实现System.IComparable](#)

[10.4 抽象类](#)

[10.5 小结](#)

[第11章 枚举](#)

[11.1 枚举概览](#)

[11.2 类中的枚举](#)

[11.3 switch语句中的枚举](#)

[11.4 小结](#)

[第12章 泛型](#)

[12.1 为什么要使用泛型](#)

[12.2 泛型介绍](#)

[12.3 应用限制](#)

[12.4 编写泛型类型](#)

[12.5 小结](#)

[第13章 集合](#)

[13.1 概述](#)

[13.2 List类](#)

[13.2.1 重要的方法](#)

[13.2.2 List示例](#)

[13.3 HashSet类](#)

[13.3.1 有用的方法](#)

[13.3.2 HashSet示例](#)

[13.4 Queue类](#)

[13.4.1 有用的方法](#)

[13.4.2 Queue示例](#)

[13.5 Dictionary 类](#)

[Dictionary示例](#)

[13.6 小结](#)

[第14章 输入输出](#)

[14.1 文件和目录的处理与操作](#)

[14.1.1 创建和删除文件](#)

[14.1.2 创建和删除一个目录](#)

[14.1.3 操作File和Directory的属性](#)

[14.1.4 列出目录下的文件](#)

[14.1.5 复制和移动文件](#)

[14.2 输入/输出流](#)

[14.3 读取文本（字符）](#)

[14.4 写入文本（字符）](#)

[14.5 读取和写入二进制数据](#)

[14.6 小结](#)

[第15章 WPF](#)

[15.1 概述](#)

[15.2 应用程序和窗口](#)

[15.2.1 简单的WPF应用程序1](#)

[15.2.2 简单的WPF应用程序2](#)

[15.3 WPF控件](#)

[15.4 面板和布局](#)

[15.5 事件处理](#)

[15.6 XAML](#)

[15.7 小结](#)

[第16章 多态](#)

[16.1 定义多态](#)

[16.2 多态的应用](#)

[16.3 一个绘图程序中的多态](#)

[16.4 小结](#)

[第17章 ADO.NET](#)

[17.1 介绍ADO.NET](#)

[17.2 访问数据的5个步骤](#)

[17.2.1 安装数据提供者](#)

[17.2.2 获取一个数据库连接](#)

[17.2.3 创建一个DbCommand对象](#)

[17.2.4 创建一个DbDataReader](#)

[17.3 连接到SQL Server的示例](#)

[17.4 小结](#)

[附录A Visual Studio Express 2012 for Windows Desktop](#)

[A.1 硬件和软件的要求](#)

[A.2 下载和安装](#)

[A.3 注册Visual Studio Express 2012](#)

[A.4 创建一个项目](#)

[A.5 创建一个类](#)

[A.6 运行一个项目](#)

[附录B Visual C# 2010 Express](#)

[B.1 硬件和软件的要求](#)

[B.2 下载和安装](#)

[B.3 注册Visual C# 2010 Express](#)

[B.4 创建一个项目](#)

[B.5 创建一个类](#)

[B.6 运行一个项目](#)

[附录C SQL Server 2012 Express](#)

[C.1 下载SQL Server 2012 Express](#)

[C.2 安装SQL Server 2012 Express](#)

[C.3 连接到SQL Server并创建一个数据库](#)

[欢迎来到异步社区！](#)

版权信息

书名：C#初学者指南

ISBN：978-7-115-35290-3

本书由人民邮电出版社发行数字版。版权所有，侵权必究。

您购买的人民邮电出版社电子书仅供您个人使用，未经授权，不得以任何方式复制和传播本书内容。

我们愿意相信读者具有这样的良知和觉悟，与我们共同保护知识产权。

如果购买者有侵权行为，我们可能对该用户实施包括但不限于关闭该帐号等维权措施，并可能追究法律责任。

• 著 [加] Jayden Ky

译 李 强 吴 戈

责任编辑 陈冀康

• 人民邮电出版社出版发行 北京市丰台区成寿寺路11号

邮编 100164 电子邮件 315@ptpress.com.cn

网址 <http://www.ptpress.com.cn>

• 读者服务热线：(010)81055410

反盗版热线：(010)81055315

版权声明

Simplified Chinese translation copyright ©2014 by Posts and
Telecommunications Press ALL RIGHTS RESERVED

C#: A Beginner's Tutorial, by Jayden Ky

Copyright © 2013 Brainy Software Inc.

本书中文简体版由**Brainy Software Inc.** 授权人民邮电出版社出版。未经出版者书面许可，对本书的任何部分不得以任何方式或任何手段复制和传播。

版权所有，侵权必究。

内容提要

C#是一种简单易学的、成熟的编程语言，作为.NET Framework的一部分，C#语言得到非常广泛的应用。

本书是一本C#语言的初学者的教程，涵盖了C#和.NET Framework语言中最重要的主题。全书共包括16章和3个附录，依次介绍了 C#程序语言、面向对象编程和.NET Framework类库3个方面的知识和技术。附录部分简单介绍了Visual Studio Express和SQL Server Express等常用工具。

本书内容全面，示例丰富，浅显易懂，可以帮助读者掌握C#编程基础知识，以完成中级C#程序员的日常任务。本书适合C#语言初学者和对C#编程感兴趣的读者阅读，也可以作为相关专业的教学参考书或培训教材。

前言

欢迎阅读本书。

C#（读作“c sharp”）是一种易学的、成熟的编程语言。同时，它也是.NET Framework的一部分。.NET Framework是很大的一个技术集合，它包罗万象，以至于初学者往往不知从何入手。如果你也是一名初学者，那么本书非常适合你，因为本书就是专门为.NET初学者所编写的教程。

作为初学者的教程，本书并不会介绍.NET Framework中的每一种技术。相反，本书涵盖C#和.NET Framework语言中最重要的主题，掌握了这些内容，你才能够自学其他的技术。但是本书的内容很全面，在完全理解各章的内容后，你就能很好地完成中级C#程序员的日常任务。

本书介绍了以下三个主题，它们是专业的C#程序员必须要掌握的。

- C#程序语言
- C#的面向对象编程（Object-Oriented Programming, OOP）
- .NET Framework类库

设计一门高效的C#课程，其困难之处就在于，这三个主题实际上是彼此相关的。一方面，C#就是OOP语言，所以如果你已经了解OOP，那么学习C#的语法就较容易。另一方面，诸如继承、多态、数据封装等OOP的特性，我们最好是用真实案例来讲解。可是，理解真正的C#程序，却需要我们具有.NET Framework类库的知识。

因为这三个主题相互依赖，所以我们不能把它们划分为三个独立的部分。相反，讨论一个主题的章节会和讨论另一个主题的章节交织在一起。例如，在介绍多态之前，本书要确保我们已经熟悉某些.NET Framework的类，以便能给出真实的案例。另外，如果不能很好地理解一组特定的类，我们就很难理解泛型这样的语言特性，而这组特定的类又是在讨论完支持类后才介绍的。

本书中也会有一个主题在两三个地方重复出现的情况。例如，for和while循环语句是一个基本的语言特性，应该在前边的章节中介绍它。

用foreach循环遍历一个数组或集合，却只能在介绍过数组和集合类型后再讲解。因此，循环语句首先会出现在第3章，然后在第5章介绍数组和第13章介绍集合时，会再次出现。

本前言接下来的内容会给出.NET Framework的高级概述、OOP的介绍、每章的简单介绍以及.NET Framework的安装指南。

.NET Framework概述

.NET Framework是一种编程环境的常用名称，其规范的叫法为通用语言基础架构（Common Language Infrastructure, CLI）。CLI是微软开发的并且通过了ISO和ECMA的认证标准。ISO和ECMA都是国际标准化机构。

.NET Framework引人注目的地方之一，就是支持多种编程语言。实际上，最新的统计结果表明，有超过30种语言可以使用.NET Framework，包括Visual Basic、C#和C++。这就意味着，如果习惯于使用Visual Basic，可以继续用这种语言编程。如果你是一名C++程序员，也不必为了充分利用.NET Framework所提供的优势而去学习一种新的语言。

但是，多语言支持并不是.NET Framework的唯一特性。它还提供了一整套技术，使软件开发更迅速且应用程序更加健壮和安全。多年来，.NET Framework成为首选的技术，因为它具有以下优点。

- 跨语言集成
- 易用性
- 平台独立性
- 一个可以加速应用程序开发的庞大的类库
- 安全性
- 可扩展性
- 广泛的行业支持

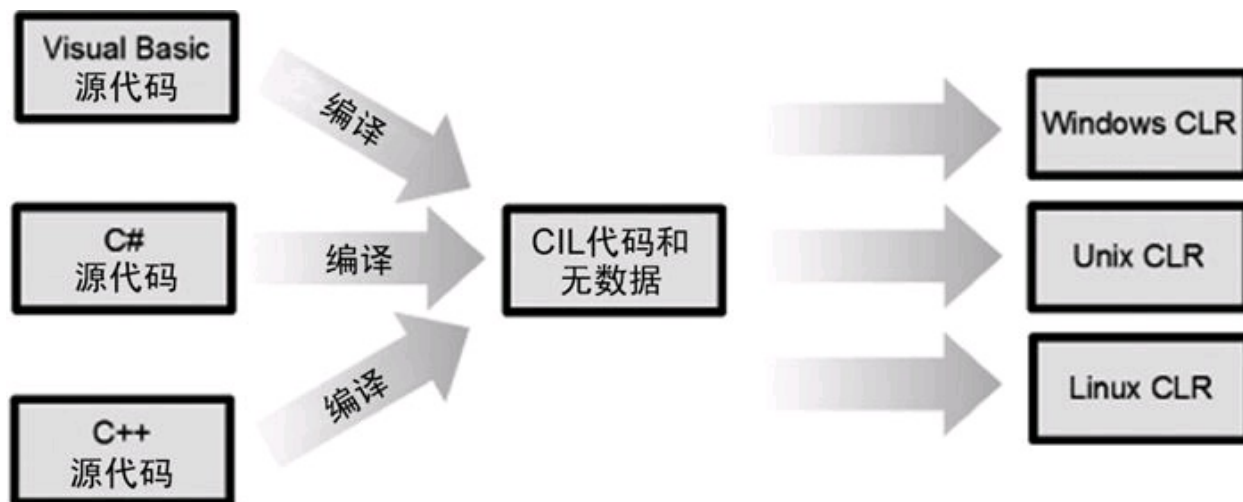
.NET Framework不像传统的编程环境。在传统的编程中，源代码要编译成可执行代码。这个可执行代码对于目标平台来说是本地的，因为它只能在原计划运行的平台上运行。换句话说，在Windows上编写和编译的代码，只能在Windows上运行；在Linux上编写的代码，只能在

Linux上运行，依次类推，如图I-1所示。



图I-1 传统的编程范式

相反，.NET Framework程序则编译成通用中间语言（Common Intermediate Language, CIL，读作“sil”或“kil”）代码。如果你熟悉Java，CIL代码相当于Java的字节码。CIL代码此前叫作微软中间语言（Microsoft Intermediate Language）或MSIL代码，只能运行在公共语言运行时（Common Language Runtime, CLR）上。CLR是解释CIL代码的一个本地应用程序。因为CLR可用于多个平台，同样的CIL代码也变成了跨平台的代码。如图I-2所示，我们可以用支持的任何语言编写一个.NET程序并且把它编译成CIL代码。同样CIL代码可以运行在任意已经开发了CLR的操作系统上。除了CIL代码以外，.NET编译器还生成了元数据以描述CIL代码中的类型。这个元数据在术语上叫作清单（manifest）。把CIL代码和相应的清单一起打包成一个.dll或.exe文件，叫作程序集。



图I-2 .NET编程模型

目前，微软提供了CLR针对Windows的实现，但是随着来自Project Mono（<http://www.mono-project.com>）和DotGNU Portable.NET（<http://dotgnu.org/pnet.html>）的其他实现，CIL代码已经能

够在Linux、Mac OS X、BSD、Sony PlayStation 3和Apple iPhone上运行了。

.NET术语中把只能在CLR之上运行的代码称为托管代码（Managed code）。另一方面，一些.NET语言，诸如C#和C++，既可以生成托管代码又可以生成非托管代码。非托管代码运行在运行时之外。本书只介绍托管代码。

当用C#或其他.NET语言编程时，我们总是使用通用类型系统（Common Type System, CTS）来工作。在解释CTS前，我们要确定你已经了解了什么是类型。那么，什么是类型呢？在计算机编程中，类型决定了值的种类，例如一个数字或一段文本。对于编译器来说，类型信息特别有用。例如，它使得的 $3*2$ 这个乘法运算有意义，因为3和2都是数字。但是，我们如果在C#代码中写下VB *C#，编译器将认为它无效，因为不能把两段文本相乘，至少，在C#中不允许这样做。

CTS中有5种类型。

- 类
- 结构
- 枚举
- 接口
- 委托

在本书中，我们会逐一介绍这些类型。

面向对象编程的概述

OOP通过对真实世界中的对象建模应用来工作。OOP有三种主要特性：封装、继承和多态。

OOP的好处是很实际的。这也是为什么包括C#在内的大部分现代编程语言都是OO的。我们甚至可以引用为了支持OOP而进行语言转换的两个著名的例子：C语言演变成了C++，而Visual Basic则升级为Visual Basic.NET。

本节介绍OOP的好处并且对学习OOP的难易程度给出一个评估。

OOP的优点

OOP的优点包括代码易维护、代码可复用和可扩展性。下面，我们对这些优点作更详尽的介绍。

1. 易维护

当前的应用软件往往非常大。很久以前，一个“大”系统包含几千行代码。而现在，甚至100万行代码都不算是大系统。当系统变得更大的时候，就开始产生一些问题。C++之父Bjarne Stroustrup曾经说过类似于下面的话。小程序可以随便写，即使不是很容易，但最终你还是能让它工作。但是，大程序却截然不同。如果你没有使用“好的编程”技术，新的错误会随着你修正老错误的步伐而不断地产生。

原因就在于，一个大程序的不同部分会相互影响。当我们修改程序的某一部分内容时，可能不会意识到这种变化可能会影响到其他部分。OOP使应用程序更容易模块化，而且模块化使得维护不再头疼。模块化在OOP中是内在的，因为类（它是对象的模板）本身就是一个模块。好的设计应该允许一个类包含相似的功能和相关的数据。在OOP中，一个经常用到的重要的相关术语是耦合，它表示两个模块之间有一定程度的交互。各部分之间的松耦合使得代码更容易实现复用，而复用是OOP的另一个优点。

2. 可复用性

可复用性意味着，如果对于最初编写的代码有相同的功能需求，那么代码的编写者或其他人可以复用它们。一种OOP语言通常带有一套现成的库，这并不奇怪。以C#为例，这门语言是.NET Framework的一部分，它提供了一套详细设计并经过测试的类库。编写和发布自己的库也很容易。编程平台中对可复用性的支持是非常吸引人的，因为它会缩短开发时间。

类的可复用性的最大挑战之一就是为类库创建好的文档。程序员如何才能快速地找到提供了他（或她）想要的功能的那个类？找到这样一个类会比从头编写一个新的类更快吗？好在，.NET Framework类库有大量的文档。

可复用性不仅适用于编码阶段类或其他类型的复用，当我们在OO

系统中设计一个应用时，OO设计问题的解决方案也可以复用。这些解决方案叫作设计模式。为了更容易地指明每一种解决方案，人们给每种模式都起了一个名称。在经典的设计模式图书《*Design Patterns: Elements of Reusable Object-Oriented Software*》中，我们可以找到可复用的设计模式的最早的名录，该书作者是Erich Gamma、Richard Helm、Ralph Johnson和John Vlissides。

3. 可扩展性

每个应用都是独一无二的，都有自己的需求和规格。说到可复用性，有时我们无法找到一个已有的类能够提供应用程序所需的确切功能。但是，我们可能会找到一个或两个类，它们提供了这些功能的一部分。可扩展性意味着，我们仍然可以使用这些类，通过扩展它们来满足我们的需求。我们还节省了时间，因为不需要从头编写代码。

在OOP中，可扩展性通过继承来实现。我们可以扩展一个已有的类，为它添加一些方法或数据，或者修改不喜欢的方法行为。如果知道该基本功能会在许多示例中使用，但是又不希望类提供非常具体的功能，那么我们可以提供一个通用的类，以后可以扩展它来为应用程序提供具体的功能。

OOP很难吗

C#程序员需要熟练掌握OOP。如果你曾经使用过过程式语言，诸如C或Pascal，你会发现有很大的不同。鉴于此，这既是好消息又是坏消息。

我们先说坏消息。

研究人员一直针对在学校教授OOP的最好的方式而争论不休。有些人认为，最好的方法是在介绍OOP前先教授过程式语言。我们发现，在许多学校，OOP课程往往安排在接近大学最后1年。

但是，最新的研究表明，拥有过程式编程技能的人和OO程序员的视角以及解决问题的模式差异迥然。当熟悉过程式编程的人需要学习OOP时，他所面临的障碍就是模式的转变。据说，从过程式转变到面向对象模式，这种观念的转变需要6~18个月的时间。其他的研究也表明，没有学习过过程式编程的学生，不会觉得OOP很难。

我们再来说好消息。

C#是学习OOP的最简单的语言之一。例如，我们不需要担心指针，也不需要花费宝贵的时间去解决由于没有释放不用的对象而引起的内存泄漏的问题。最重要的是，.NET Framework有一个非常广泛的类库，在其早期的版本中，bug相对来说很少。一旦我们掌握了OOP的基本要点，用C#编程是很容易的。

关于本书

本书每一章的内容可以概述如下。

第1章，“初识C#”。本章编写了一个简单的C#程序，然后用csc工具编译并运行它。另外，本章还给出关于编码惯例和集成开发环境的一些建议。

第2章，“语言基础”介绍了C#语言的语法，还介绍了字符集、基本类型、变量和运算符等。

第3章，“语句”，介绍了C#中的for、while、do-while、if、if-else、switch、break和continue等语句。

第4章，“对象与类”，是本书中的第一节OOP课程。本章通过解释什么是C#对象以及如何在内存中存储它开始了对OOP的学习，然后继续介绍了类、类成员以及两个OOP的概念（抽象和封装）。

第5章，“核心类”介绍了.NET Framework类库中重要的类：System.Object、System.String、System.Text.StringBuilder和System.Console，还介绍了数组。本章非常重要，因为本章所介绍的类是.NET Framework中最常用到的一些类。

第6章，“继承”，介绍了OOP的特性之一——继承，它使得代码可以扩展。本章介绍了如何扩展一个类、影响子类的可访问性以及覆盖方法等内容。

第7章，“结构”，介绍了CTS的第二种类型——结构。本章强调了引用类型和值类型之间的不同，介绍了.NET Framework类库中经常用到

的一些结构。本章还介绍了如何编写自己的结构。

毋庸置疑，错误处理在任何编程语言中都是一项重要特性。作为一门成熟的语言，C#有非常健壮的错误处理机制，它能防止bug四处蔓延。第8章“错误处理”详细介绍了这种机制。

第9章，“数字和日期”，介绍了在使用数字和日期时所要处理的三个问题：解析、格式化和操作。本章还介绍了可以帮助我们完成这些任务的.NET类型。

第10章，“接口和抽象类”，解释了接口远不只是没有实现的类那么简单。接口定义了服务提供者和客户之间的一个契约。本章还介绍了如何使用接口和抽象类。

第11章，“枚举”，介绍了如何使用关键字enum来声明一个枚举类型。本章还描述了如何在C#程序中使用枚举。

第12章，“泛型”，介绍了泛型。

第13章，“集合”，介绍了如何使用System.Collections.Generic命名空间的成员来组织对象和操作它们。

第14章，“输入和输出”，介绍了流的概念，而且介绍了如何使用流来执行输入和输出的操作。

你会发现第15章“WPF”的内容很有趣，因为我们将学习编写有漂亮用户界面和易用控件的桌面应用程序。

多态是OOP的主要支柱之一。当一个对象的类型在编译时不为人知的时候，多态是非常有用的。第16章“多态”介绍了这种特性并且提供了有用的示例。

访问数据库并且操作数据，这是商业应用程序中最重要的一项任务。目前有许多种不同的数据库服务器，访问不同的数据库需要不同的技能。在第17章“ADO.NET”中，我们介绍如何访问数据库以及操作数据库中的相关数据。

附录A，“Visual Studio Express 2012 for Windows Desktop”，介绍了一款免费的集成开发环境（Integrated Development Environment，

IDE），它能帮助我们更有效地编写代码。Visual Studio Express 2012 for Windows Desktop运行在Windows 7和Windows 8上，如果你使用这类操作系统，应该考虑使用它。如果你使用更早版本的Windows，那么可以选择Visual C# 2010 Express作为IDE，我们会在附录B“Visual C# 2010 Express”中介绍它。

最后，附录C介绍了如何安装SQL Server 2012 Express这款免费的软件并创建了一个数据库。

下载和安装.NET Framework

在开始编译和运行C#程序前，我们需要下载和安装.NET Framework软件。

默认情况下，在.NET Framework出现后发布的Windows操作系统会包含某个版本的.NET Framework软件。Windows 7附带的是.NET Framework 3.5。因此，如果你需要版本4或4.5，那就需要单独安装它。如果你计划使用Visual Studio，那么你很幸运，因为它已经包含了某个版本的.NET Framework，不需要再单独安装。否则，你可以通过以下链接下载4.5版本。

http://msdn.microsoft.com/en-us/library/5a4x27ek.aspx

要在命令行进行编译，我们需要把包含csc.exe文件（C#编译器）的路径添加到PATH环境变量中。这个路径是C:\Windows\Microsoft.NET\Framework\v4.x.y，其中x和y是版本号。x和y的实际值要根据所安装的版本来决定。例如，作者计算机上的版本是4.0.30319。

如果你使用的是64位的计算机，那么路径可能如下所示：
C:\Windows\Microsoft.NET\Framework64\v4.x.y。

要给PATH变量添加一个路径，我们首先需要用鼠标右键点击桌面上的“My Computer”图标，选择“Properties”菜单项。其次在弹出的对话框中，点击位于“Advanced”标签页或“Advanced system settings”标签页上的“Environment Variables”按钮。最后在弹出的对话框中，把上述路径添加到System变量列表框中当前的Path变量的末尾。请注意，每条路径

之间必须要用分号隔开。

选择一个IDE

IDE是每一位程序员都要使用的工具。大多数现代IDE通过帮助程序员更早地找到bug、debug并跟踪程序，从而显著地提高生产效率。

就.NET Framework开发而言，它有很多IDE可用，但是Microsoft Visual Studio显然是赢家。好在，精简版的Visual Studio，Visual Studio Express 2012 for Windows Desktop（针对Windows 7和Windows8 用户）和Visual C# 2010 Express（针对较早的Windows版本）是免费的。如果你还没有IDE，那么应该现在就去下载并安装它。在第一次使用后，你还需要注册软件，以便继续使用它。

注册是免费的。

下载程序示例

本书的程序示例以及每章中问题的答案可以从以下网址下载。

http://books.brainysoftware.com/download/csharp.zip

首先将这个zip文件解压缩到一个工作路径中。现在可以开启你的C#编程之旅了。

第1章 初识C#

开发一款C#程序，包括编写代码、把它编译成通用中间语言（Common Intermediate Language，CIL）编码以及运行CIL编码。作为一名C#程序员，你会不断地重复这个过程，而熟悉和习惯这个过程也是至关重要的。因此，本章的主要目标是，帮助你体验在Visual Studio Express 2012 for Windows Desktop 或Visual C# 2010 Express这两种免费的微软IDE中使用C#来进行软件开发的过程。

编写的代码不仅能工作，而且要易读和可维护，这一点是很重要的。本章将介绍C#编码惯例。

本章及以后章节的示例代码都假设用Visual Studio Express 2012 for Windows Desktop或Visual C# 2010 Express 开发。

1.1 第一个C#程序

本节重点介绍C#开发的步骤：编写程序、把它编译成CIL编码并且运行CIL编码。这里你将会用到Visual Studio Express 2012 for Windows Desktop 或Visual C# 2010 Express，可以通过微软的官方网站免费下载它们。如果你还没有安装IDE，请先安装IDE。Visual Studio Express 2012 for Windows Desktop适合运行在Windows 7和Windows 8上，如果你使用这类操作系统，应该考虑使用它；否则，请下载和安装Visual C# 2010 Express，可以参见附录A或附录B。

1.1.1 启动IDE

启动IDE。打开程序后，你会看到如图1-1或1-2所示的界面。如果软件无法打开，那是因为还没有注册，你应该马上去注册。注册是免费的，而且很简单，更多信息请参见附录A和附录B。



图1-1 Visual Studio Express 2012的启动界面

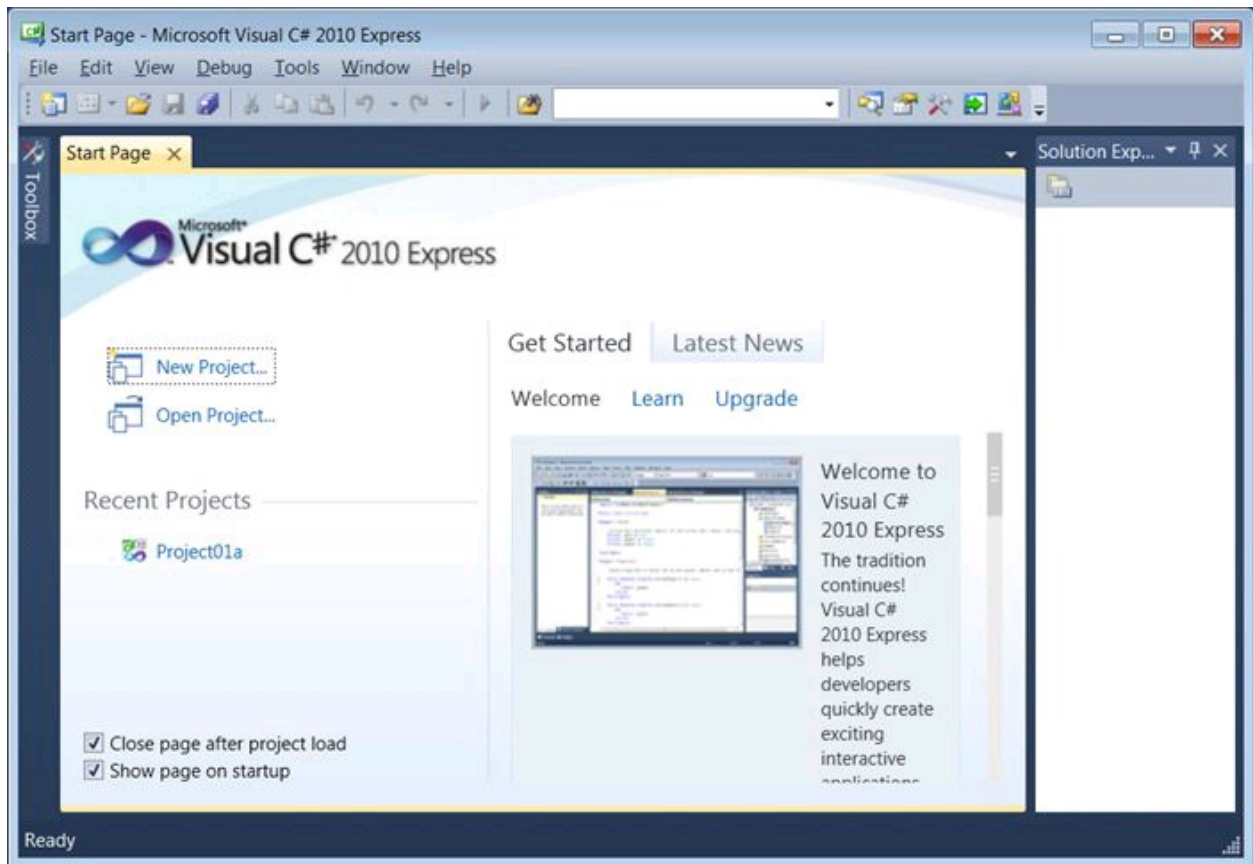


图1-2 Visual C# 2010 Express已准备就绪

两个IDE窗口看上去不同，但是都提供了类似的功能。因此下文我们只介绍在Visual C# 2010 Express的截屏图。

点击“New Project”图标创建新的项目，然后选择“Console Application”，如图1-3所示。

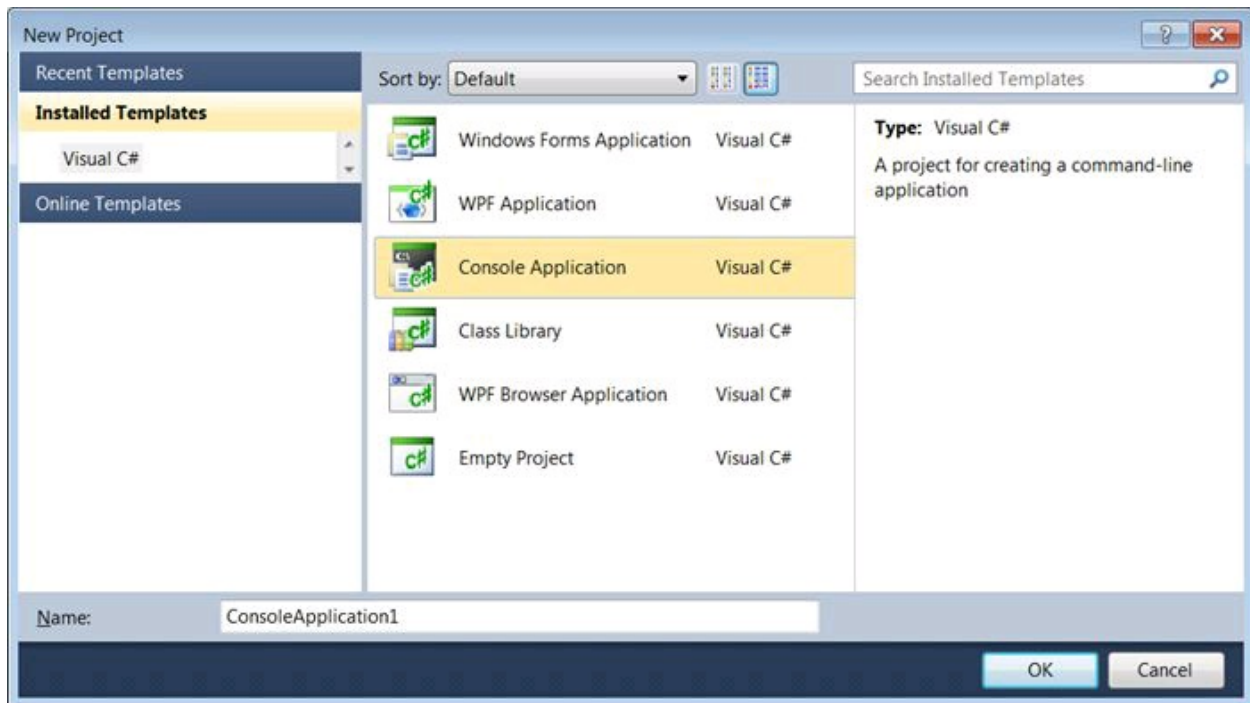


图1-3 创建一个新的项目

接受ConsoleApplication1作为解决方案和项目名称，然后点击“OK”按钮，你就会看到所创建的项目和解决方案，如图1-4所示。更棒的是，Visual C# 2010 Express还创建了一个附带一些样板代码的程序文件，如图1-3所示。注意，项目就是一个便于管理应用的容器。它包含C#源代码文件、图片和视频文件等其他的资源文件以及描述应用的记录文档。当创建一个项目时，Visual C# 2010 Express还创建一个解决方案。解决方案是另一种容器，它可以包含一个或多个项目。

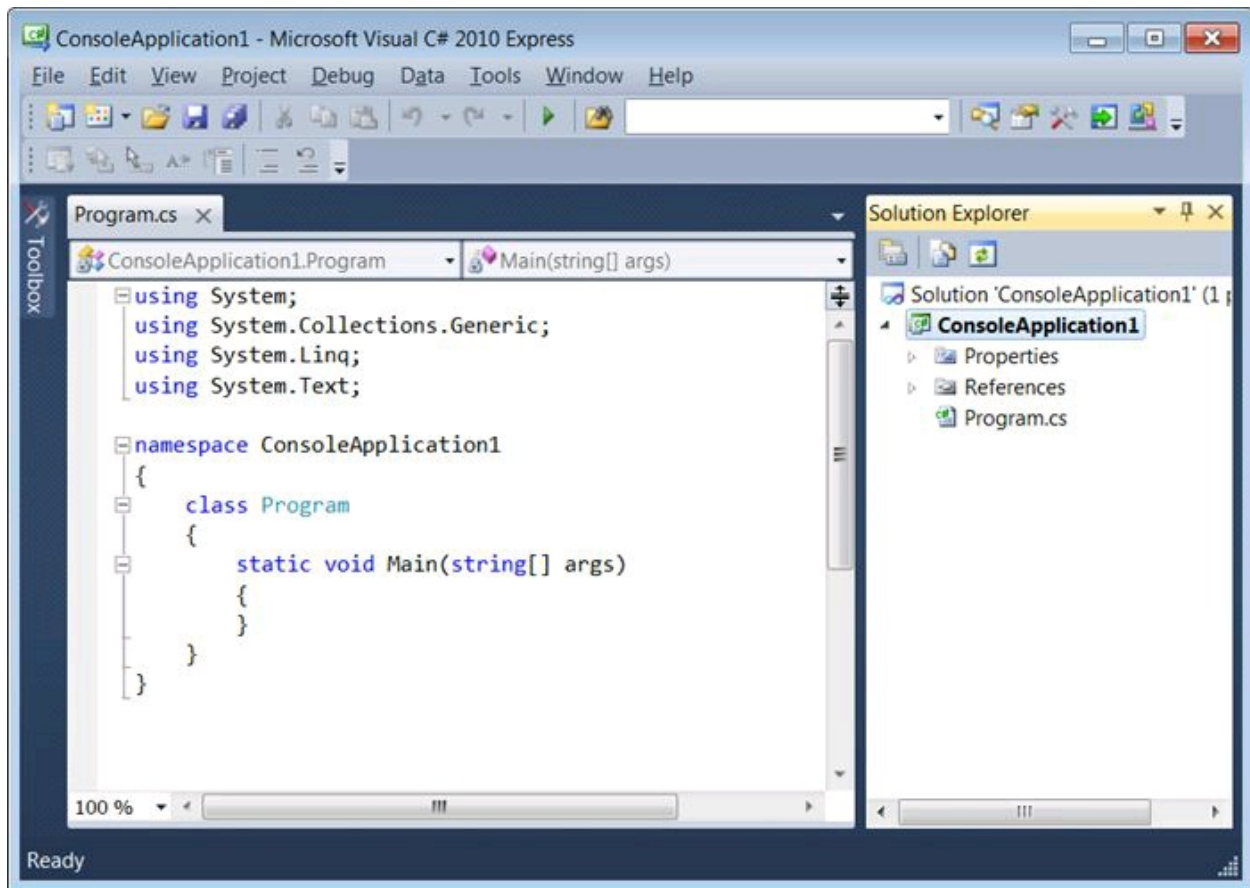


图1-4 创建的解决方案和项目

现在，你可以开始编写代码了。

1.1.2 编写C#程序

在**static void Main(string[] args)**后边的大括号中插入如下两行语句。

```
Console.WriteLine("Hello World!");  
Console.ReadLine();
```

程序清单1.1展现了完整的程序代码，新插入的语句用加粗字体表示。

程序清单1.1 一个简单的C#程序

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace ConsoleApplication1
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Hello World!");
            Console.ReadLine();
        }
    }
}
```

另外，你可以双击本书附带的zip文件中的.sln文件查看，zip文件可以从本书站点下载。

1.1.3 编译和运行C#程序

用Visual C# 2010 Express开发真的非常简单。要编译代码，直接按下“F5”键或者点击工具栏上的Start 按钮即可。Start按钮是绿色的，如图1-5所示。



图1-5 Start按钮

如果程序编译成功，Visual C# 2010 Express将会运行这个程序。你可以在控制台看到文本“Hello World!”，如图1-6所示。

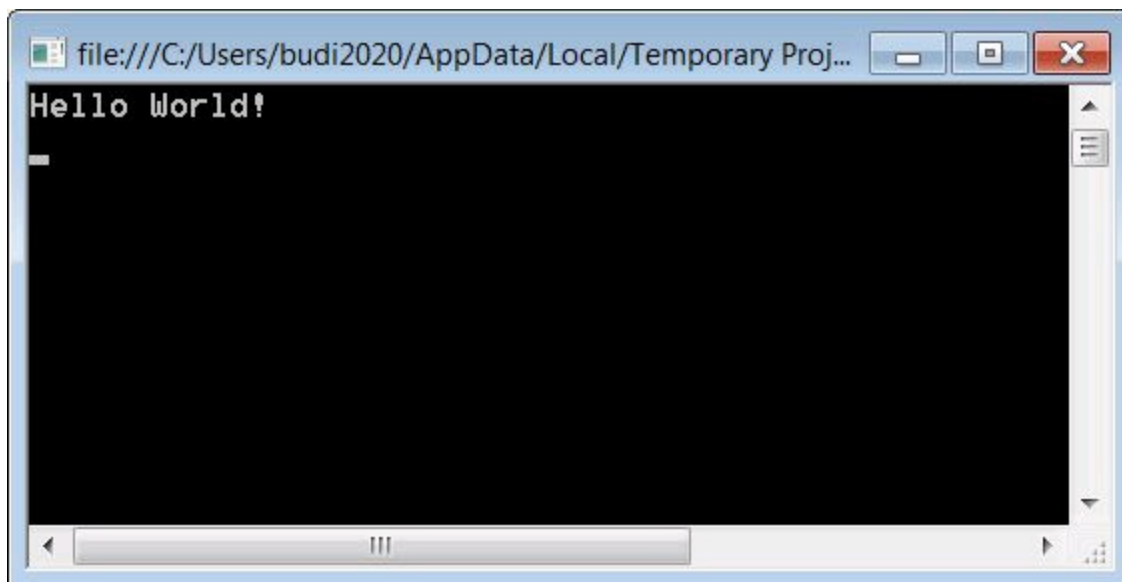


图1-6 运行这个程序

恭喜，你已经成功编写了第一个C#程序。在欣赏完第一个程序后，你可以按下“Enter”键来关闭控制台。本章唯一的目的是让你熟悉编写和编译的过程，我们就不再试图去解释程序是如何工作的了。

1.2 C# 编码惯例

编写能够正确运行的C#程序很重要。但是，代码的易读性和易维护性也是至关重要的。可以说，一款软件80%的生命周期是用在维护上。程序员的流动率是很高的，因此其他人来维护你编写的代码的可能性也是很大的。任何接手你所编写的代码的人，都会因为你编写的源代码干净并且易读而心存感激。

使用一致的编码惯例是使你的代码更易阅读的一种方法（其他方法还包括正确的代码组织和充分的注释）。编码惯例包括文件名、文件组织、缩进、注释、声明、语句、空白以及命名惯例。微软发布了一个文档，描述了微软员工需要遵守的标准。该文档的链接如下。

<http://msdn.microsoft.com/en-us/library/ff926074.aspx>

本书的示例程序都会遵循该文档所建议的惯例。我们也希望你能在编程生涯的第一天就养成习惯，遵守这些惯例，以便在今后能自然而然

地编写出干净的代码。

1.3 小结

本章用Visual Studio Express 2012 for Windows Desktop或Visual C# 2010 Express帮助你编写了第一个C#程序。你已经成功地编写、编译和运行了你的程序。

第2章 语言基础

C#是一种面向对象编程（Object-Oriented Programming, OOP）的语言，因此对于OOP的理解是至关重要的。本书会在第4章开始介绍OOP。但在了解更多OOP的特性和技术前，我们先要学习以下内容，即本章所介绍的基本的编程概念，包括如下主题。

- 编码集：C#支持Unicode字符编码集，程序元素名称不仅局限于ASCII（American Standard Code for Information Interchange，美国信息交换标准代码）字符。几乎可以使用目前所有的人类语言来编写文本。
- 内建类型：在.NET类库中，这些类型中的每一种都有一个缩写名称或别名，例如，int就是System.Int32结构的缩写名称。
- 变量：变量就是能够改变其内容的占位符。变量的类型有很多种。
- 常量：值不能改变的占位符。
- 直接量：直接量是C#编译器所能够理解的数据值的表示。
- 类型转换：将一种数据类型转换为另一种类型。
- 操作符：操作符是所要执行的某种操作的表示。

注意

Java和C++是在C#之前两种常用的编程语言，如果你曾用它们编写过程序，那么在学习C#时会觉得很容易上手，因为C#语法与Java和C++很相似。但是，C#的发明者增加了一些Java和C++中所没有的特性，也删掉了一些Java和C++中已有的特性。

2.1 ASCII和Unicode

传统上，在讲英语的国家计算机只使用ASCII字符集来表示字母和数字字符。ASCII码中每个字符用7位表示，该字符集中一共有128个字符，包括了小写和大写的拉丁字母、数字和标点符号。

ASCII字符集后来又扩展并增加了128个字符，包括像德语字符ä、ö、ü和英国货币符号£。这个字符集叫做扩展ASCII，每个字符用8位来表示。

ASCII和扩展ASCII仅仅是众多可用字符集中的两种。另一种常用的字符集是ISO（国际标准组织）制定的ISO-8859-1标准，也叫做Latin-1。在ISO-8859-1中，每种字符同样是用8位表示。这个字符集包括许多西欧国家语言书写所用到的所有字符，例如德语、丹麦语、荷兰语、法语、意大利语、西班牙语、葡萄牙语，当然也包括英语。每字符8位的字符集很方便，因为一个字节正好是8位。因此，存储和传送8位字符集编写的文本是最高效的。

然而，不是每种语言都使用拉丁字母。中文、韩语和泰语就是使用不同字符集的语言。例如，在中文中，每个字符表示一个汉字，而不是一个字母。有成千上万这样的字符，8位无法表示字符集中所有的字符。日语也使用自己的不同的字符集。总而言之，针对世界上所有的语言，会有成百上千的字符集。这样就会很混乱，因为在一个字符集中表示一个特定字符的代码，在另一个字符集中会表示一个完全不同的字符。

Unicode是由一个叫做Unicode Consortium（www.unicode.org）的非营利组织开发的字符集。它试图把世界上所有语言的所有字符放到一个字符集中。在Unicode中，唯一的一个数字精确地表示一个字符。Unicode用于.NET Framework、Java、XML、ECMAScript和LDAP等语言中，当前的版本是Unicode 6。它也被诸多业内领导者所采用，诸如IBM、Microsoft、Oracle、Google、HP、Apple等。

最初，每个Unicode字符用16位表示，能够表示超过65000个字符。65000个字符足以对世界上主要语言的大部分字符进行编码。然而，Unicode Consortium计划要对超过一百万个字符进行编码。要满足这么大的数量，你就需要比16位更大的存储空间来表示每个字符。事实上，我们认为32位系统可以非常方便地存储Unicode字符。

现在，我们已经发现了一个问题。当Unicode为所有语言中所有用到的字符提供足够的空间，存储和传送Unicode文本就不会像存储和传送ASCII或Latin-1字符那样高效。在互联网世界，这是一个大问题。想象一下传送4倍于ASCII文本数据的数据是什么样子？

好在字符编码可以使Unicode文本的存储和传送更高效。我们可以把字符编码理解成数据压缩。现在，有很多种字符编码可用，Unicode Consortium支持其中的3种。

- UTF-8: 这对于HTML以及将Unicode字符转换为可变长度的字节编码的协议很常见。UTF-8的优势在于，它与人们所熟知的ASCII字符集一致，拥有和ASCII相同的字节值，并且有许多现成软件可以把Unicode字符转换成UTF-8。大多数浏览器都支持UTF-8字符编码。
- UTF-16: 在这个字符编码中，所有常用的字符都存放在一个独立的16位的编码单元中，其他不常用的字符可以通过成对的16位编码单元来访问。.NET Framework使用这种字符编码。
- UTF-32: 这种字符编码为每个字符使用32位。显然，这不是Internet应用的选择。至少，目前不是。

ASCII字符在软件编程中仍然扮演着主要的角色。C#对大部分输入的元素使用ASCII码，除了注释、标识符以及字符和字符串内容以外。对于后者，C#支持Unicode字符。也就是说，你可以使用英语之外的其他语言写注释、标识符和字符串。例如，如果你是住在北京说中文的人，你可以用中文字符作为变量名称。下面这句C#代码声明一个名为password的标识符，它由ASCII字符组成。

```
string password = "secret";
```

与之相对，如下的标识符用简体中文字符表示。

```
string 密码 = "secret";
```

2.2 内建类型和通用类型系统

C#是一种强类型语言，这意味着每条数据必须有一个类型，对于变量这样的数据占位符也是一样的。C#定义了各种所谓的内建类型。

与此同时，.NET Framework支持多种编程语言，并且用一种语言编写的代码可以与另一种语言编写的代码交互操作。为了促进这种语言的互操作性，.NET Framework设计人员定义了通用类型系统（Common Type System, CTS）。CTS指定了运行时（也就是CLR，公共语言运行时）所支持的所有数据类型。用一种语言编写的程序要被另一种语言编写的程序所调用，这就要求前一个程序只能被转换成CTS兼容的类型。

C#既支持兼容CTS的内建数据类型，也支持不兼容CTS的内建数据类型。但是，转换成不兼容CTS的数据类型的C#代码，可能就无法很好地与其他语言编写的代码交互操作。

表2-1列举了C#的内建类型。

表2-1 C#内建类型

C# 类型	.NET 类型	大小（字节）	值/区间
byte	Byte	1	0~255
char	Char	2	任意Unicode字符
bool	Boolean	1	true或false
sbyte*	SByte	1	-128~127
short	Int16	2	-32 768~32 767
ushort*	UInt16	2	0~65 535
int	Int32	4	-2 147 483 648~2 147 483 647
uint*	UInt32	4	0~4 294 967 295
float	Single	4	-3.402823e38~3.402823e38
double	Double	8	-1.79769313486232e308~1.79769313486232e308
decimal	Decimal	16	$\pm 1.0 \times 10^{28}$ to $\pm 7.9 \times 10^{28}$

long	Int64	4	-9,223,372,036,854,775,808～ 9,223,372,036,854,775,807
ulong*	UInt64	4	0～18446744073709551615
string	String		字符序列
object	Object		所有其他类型的基类型

星号标出的4个内建类型（sbyte、ushort、uint、ulong）都不兼容CTS，这意味着其他语言使用它们可能会有问题。使用不兼容CTS类型要非常慎重。

初学者经常很难选择数据类型，其实作出选择并不难。第一条规则就是必须要确定该类型是数字或非数字。如果不是数字，你可以选择**bool**、**char**或**string**。**bool**类型有两种状态，真或假（是或否）。**char**类型可以包含一个Unicode字符，如“a”、“9”或“&”。Unicode也允许char包含那些在英文字母中不存在的字符，例如日语字符“の”。**string**是一个字符序列，也是编程中最常用到的数据类型。

例如，要表示某个结果是否使用过，**bool**是很好的选择。在C#中，你可以写成如下格式。

```
bool used;
```

但是，如果结果是3种条件之一（新的、用过的、不确定），那么就不能用布尔类型，因为3种状态对于这种数据类型来说太多了。你可以用一个数值类型来替代，如byte。

选择数据类型的第二条规则是使用占空间最小的类型。这条规则尤其适用于数值类型。byte、sbyte和int都能保存整数。但是，如果保存的数字小于10，你可以选用byte或sbyte，因为它们只占用一个字节，而int需要占用4个字节。

另外，如果一个值不能是负数，要选择无符号类型。例如，年龄用byte保存而不用sbyte。

```
byte Age;
```

在这个例子中，`byte`就足够了，因为没有人能活到200岁。但是，如果要表示一个国家的人口数量，我们可能就要用`int`了。稍等，难道`int`不是一个有符号的数据类型吗？一个国家的人口数量怎么会是负数呢？没错，人口数量不会是负数，最接近的可选类型是`uint`，但是它不兼容CTS。我们要尽量避免使用任何不兼容CTS的类型，即使用其他数据类型可能会开销更大。如果要选择一个能够包含数字0到4 000 000的数据类型，我们会用`long`（8个字节宽），而不是用`uint`（4个字节宽），除非内存确实是个大问题。

`byte`、`short`、`int`和`long`数据类型只能保存整数，有小数点的数字我们需要用`float`或`double`。

2.3 变量

变量是数据占位符。C#是强类型语言，因此每个变量必须有一个声明的类型。在C#中，有两种数据类型。

- 引用类型：引用类型的变量提供了对对象的引用。
- 基本类型：基本类型变量保存了一个基本数据。

除了数据类型之外，C#变量也是一个名称或一个标识符。选择标识符有一些基本规则。

（1）标识符是一个长度没有限制的字母和数字组成的序列。标识符必须以字母或下划线开头。

（2）标识符不能是C#关键字（如表2-2所示）、**bool**值或**null**值。

（3）标识符在作用域内必须是唯一的。作用域我们会在第4章中介绍。

表2-2 C#关键字

abstract	do	in	protected	true
-----------------	-----------	-----------	------------------	-------------

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

注意

关键字加上@前缀，就可以当作标识符使用。

下面是一些合法的标识符。

```
salary  
x2  
_x3  
@class  
row_count  
密码
```

下面是一些不合法的标识符。

```
2x  
class  
c#+variable
```

2x是不合法的，因为它是以数字开头的。class是一个关键字。c#+variable是不合法的，因为它包含了一个#符号和一个加号。

这里还要注意名称是区分大小写的。x2和X2是两个不同的标识符。

声明一个变量要先给出类型，然后在名称后加上一个分号。下面是变量声明的一些示例。

```
byte x;  
int rowCount;  
char c;
```

上面示例中声明了3个变量。

- byte类型的变量x;
- int类型的变量rowCount。
- char类型的变量c

x、rowCount和c是变量名称或标识符。

我们也可以在一行中声明相同类型的多个变量，变量之间用逗号隔开。例如以下语句。

```
int a, b;
```

它等同于以下语句。

```
int a;  
int b;
```

但是，我们不建议在一行中写多个声明，因为这样会降低易读性。

最后，我们可以在声明变量的同时给它指定一个值。

```
byte x = 12;  
int rowCount = 1000;  
char c = 'x';
```

2.4 常量

在C#中，可以在一个变量声明前加上关键字**const**，使得这个值成为不可改变的。例如，一年中的月份数目是固定不变的，所以你可以写成如下格式。

```
const int NumberOfMonths = 12;
```

再看一个例子，在一个执行数学运算的类中，你可以声明一个变量pi，它的值等于22/7（一个圆的圆周除以它的直径，在数学中用希腊字母 π 来表示）。

```
const float pi = (float) 22 / 7;
```

一旦赋了值，这个值就不能再改变了。如果你试图改变它，将会导致编译错误。

注意，22/7前边的（float）是用来把除后的值转换成float类型。如果不这样做，结果会返回int类型，pi变量的值就是3，而不是3.142857。

还需要注意的是，既然C#使用Unicode字符，如果你不觉得 π 比pi更难输入的话，那么可以把变量pi直接定义成 π 。

```
const float  $\pi$  = (float) 22 / 7;
```

2.5 直接量

在程序中，经常需要给变量赋一个值，例如把数字2赋予一个int变量，把字符“c”赋予一个char变量。因此，需要用C#编译器能够理解的格式来写出值的表现形式。值的源代码表示就叫作直接量。有3种类型直接量：基本类型直接量、字符串直接量和null直接量。本章只介绍基本类型的直接量。null直接量会在第4章介绍，字符串直接量会在第5章讲述。

基本类型直接量有4种子类型：整型直接量、浮点型直接量、字符型直接量和布尔型直接量。这些子类型说明分别如下。

2.5.1 整型直接量

整型直接量可以写成十进制（有时候也说，以10为基数）、十六进制（以16为基数）和八进制（以8为基数）。例如，一百可以表示成100。下面是十进制的整型直接量。

```
2  
123456
```

下面的另一个示例代码把10赋予int类型的变量x。

```
int x = 10;
```

十六进制整型用前缀0x或0X表示，例如，十六进制数字9E写成0X9E或0x9E。八进制整型用前缀数字0表示，例如，八进制数字567表示为如下格式。

整型直接量用于为byte、short、int和long这些变量类型赋值，要注意的是，所赋的值不能超过变量的容量。例如，byte类型数字最大就是255。下面的代码会导致编译错误，因为500对于byte来说太大了。

```
byte b = 500;
```

注意，long直接量可以加上后缀L。

```
long productId = 9876543210L;
```

2.5.2 浮点型直接量

像0.4、1.23、 $0.5e^{10}$ 这样的数字都是浮点数。浮点数包含以下部分。

- 整数部分
- 小数点
- 小数部分
- 可选的指数

以1.23为例，这个浮点数的整数部分是1，小数部分是23，没有可选的指数部分。 $0.5e^{10}$ 的整数部分是0，小数部分是5，10是指数。

.NET Framework中有两种类型的浮点数：float和double。在float和double中，作为零的整数部分是可选的。换句话讲，也就是0.5可以写成.5。指数部分可以用e或E表示。

要表示float变量，你可以使用如下的格式之一。

```
Digits . [Digits] [ExponentPart] f_或_F  
. Digits [ExponentPart] f_或_F  
Digits ExponentPart f_或_F  
Digits [ExponentPart] f_或_F
```

请注意，括号中的部分是可选的。

F或f表示浮点数直接量是float类型，如果没有这一部分的话，会导致浮点数直接量成为double类型。要更明确表示double直接量，也可以加上后缀D或d。

要表示double类型的直接量，你可以使用如下的格式之一。

```
Digits . [Digits] [ExponentPart] [d_或_D]  
. Digits [ExponentPart] [d_或_D]  
Digits ExponentPart [d_或_D]  
Digits [ExponentPart] [d_或_D]
```

在float和double中，ExponentPart的定义如下所示。

```
ExponentIndicator SignedInteger
```

ExponentIndicator是e或E。SignedInteger是

```
Signopt Digits。
```

Sign是+或-，加号是可选的。

float直接量的示例如下所示。

```
2e1f  
8.f  
.5f  
0f  
3.14f  
9.0001e+12f
```

double直接量的示例如下。

```
2e1  
8.  
.5  
0.0D  
3.14  
9e-9d  
7e123D
```


2.5.3 布尔型直接量

`bool`类型有两个值，用直接量表示就是`true`和`false`。例如，下面的代码声明一个`bool`变量`includeSign`并给它赋值为`true`。

```
boolean includeSign = true;
```

2.5.4 字符型直接量

字符型直接量是用单引号括起来的Unicode字符或转义序列。转义序列就是一个Unicode字符的表现形式，这个Unicode字符无法通过键盘输入，或者它在C#中具有特殊的作用。例如，回车和换行字符用于结束一行，它们没有可见形式。要表示换行字符，你就需要对其转义，也就是写出其字符表现形式。单引号字符也需要转义，因为单引号是用来将字符包围起来的。

如下是字符直接量的示例。

```
'a'  
'z'  
'A'  
'Z'  
'ø'  
'ü'  
'%'  
'我'
```

如下是作为转义序列的字符直接量。

'\b'	退格键字符
'\t'	tab键字符
'\\'	反斜线
'\''	单引号
'\"'	双引号
'\n'	换行
'\r'	回车

另外，C#允许转义一个Unicode字符，以便你用ASCII字符序列来表示一个Unicode字符。例如，字符 ☺ 的Unicode编码是2299，你可以用下

面的字符直接量来表示这个字符。

```
'⊙'
```

然而，如果你不能通过键盘来生成那个字符，可以用下面的方式转义它。

```
'\u2299'
```

2.6 基本类型转换

在处理不同数据类型时，我们经常需要执行转换。例如，将一个变量的值赋予另一个变量就涉及转换。如果变量都是相同类型，赋值总是会成功。从一种类型转换成相同类型叫做恒等转换。例如，下列操作能够确保成功。

```
int a = 90;  
int b = a;
```

但是，不同类型的转换就不能确保成功，甚至不可能。这里还有两种其他基本类型的转换，宽化转换和窄化转换。

2.6.1 宽化转换

宽化基本类型转换是指，从一种类型到另一种类型转换时，后者的大小和前者一样大甚至更大，例如从int（32位）到long（64位）。宽化转换在下列情况中是允许的。

- byte到short、int、long、float或double
- short到int、long、float或double
- char到int、long、float或double
- int到long、float或double
- long到float或double
- float到double

从一种整数类型到另一种整数类型的宽化转换是不会丢失信息的。

同样，从float到double的转换也会保留全部信息。但是，从int或long到float的转换，可能会导致准确度降低。

宽化基本转换隐式地进行，我们不需要在代码中做任何事。例如：

```
int a = 10;  
long b = a; // 扩大转换
```

2.6.2 窄化转换

窄化转换是从一种类型到另一种更小的类型的转换，例如，从long(64位)到int(32)位。通常，窄化基本类型转换在下列情况下产生。

- short到byte或char
- char到byte或short
- int到byte、short或char
- long到byte、short或char
- float到byte、short、char、int或long
- double到byte、short、char、int、long或float

和宽化基本类型转换不同，窄化基本类型转换必须是显式的，需要在圆括号中指定目标类型。例如，下面是long到int的窄化转换。

```
long a = 10;  
int b = (int) a; // 窄化转换
```

代码第2行中的(int)告诉编译器要做一个窄化转换。

如果要转换的值大于目标类型的容量，窄化转换可能会导致信息丢失。上一个示例没有出现信息丢失是因为10对于int来说足够小。但是，下面这个转换例子中会有一些信息丢失，因为9876543210L对于int来说太大了。

```
long a = 9876543210L;  
int b = (int) a; // b的值现在是1286608618
```

窄化转换引起信息丢失，会使程序产生缺陷。

2.7 运算符

计算机程序就是实现特定功能的操作的集合。运算有很多种，包括加减乘除和位移。在本节中，我们将学习各种C#操作。

一个运算符带上一个、两个或三个操作数来执行操作。操作数是操作的对象，运算符是表示行为的符号。例如，以下为加法操作。

`x + 4`

这个例子中，x和4是操作数，+是运算符。

运算符可能返回结果，也可能不返回结果。

注意

任何合法的运算符和操作数的组合叫做表达式。例如，**x + 4**就是一个表达式。布尔型表达式的返回结果是true或false。整型表达式生成一个整数。浮点型表达式的结果是浮点数。

需要一个操作数的运算符叫做一元运算符。C#中只有少数几个一元运算符。C#中最常见的运算符类型是二元运算符，它需要两个操作数。另外C#中只有一个三元运算符，即?:，它需要3个操作数。

表2-3中列出了C#运算符。

表2-3 C# 运算符

=	>	<	!	~	?	:				
==	<=	>=	!=	&&		++	--			
+	-	*	/	&		^	%	<<	>>	>>>
+=	-=	*=	/=	&=	=	^=	%=	<<=	>>=	>>>=

C#中有6组运算符。

- 一元运算符
- 算术运算符
- 关系和条件运算符
- 位移和逻辑运算符
- 赋值运算符
- 其他运算符

这些运算符我们会在下文中介绍。

2.7.1 一元运算符

一元运算符操纵一个操作数。有6个一元运算符，本节都会介绍。

一元运算符 -

一元运算符-，返回结果是对操作数取反。操作数必须是一个数值或一个数值变量。例如，下列代码中，y的值是-4.5。

```
float x = 4.5f;  
float y = -x;
```

一元运算符 +

这个运算符返回操作数的值。操作数必须是数值或一个数值基本类型的变量。例如，下面的例子中，y的值是4.5。

```
float x = 4.5f;  
float y = +x;
```

这个运算符不太重要，因为没有它也不会有什么不同。

递增运算符 ++

递增运算符对操纵的操作数加1。操作数必须是数值类型的变量。运算符可以在操作数之前或之后。如果运算符在操作数之前，叫作前缀递增运算符；如果在操作数之后，就称为后缀递增运算符。

下例是前缀递增运算符。

```
int x = 4;
++x;
```

在执行++x之后，x的值是5。上面代码和下面代码是一样的。

```
int x = 4;
x++;
```

在执行x++之后，x的值是5。

然而，如果把递增运算符的结果赋值给同一个表达式中的另一个变量，那么前缀运算符和后缀运算符截然不同，参考例子如下。

```
int x = 4;
int y = ++x;
// y = 5, x = 5
```

前缀递增运算符在赋值之前应用，x会增加到5，然后把这个值复制给y。

但是，注意一下如下的后缀递增运算符的使用。

```
int x = 4;
int y = x++;
// y = 4, x = 5
```

使用后缀递增运算符，操作数（x）在赋值给另一个变量（y）之后，才会增加1。

请注意，递增运算符最常用于int类型。然而，它也能应用于其他数值类型，例如float类型和long类型。

递减运算符--

递减运算符对操纵的操作数减1。操作数必须是数值类型的变量。与递增运算符一样，它也分为前缀递减运算符和后缀递减运算符。例如，下列代码中x先递减，然后赋值给y。

```
int x = 4;
int y = --x;
// x = 3; y = 3
```

在下面的例子中，我们用到了后缀递减运算符。

```
int x = 4;
int y = x--;
// x = 3; y = 4
```

逻辑取反运算符！

这个运算符只能应用于bool基本类型或者System.Boolean的一个实例。如果操作数是false，这个运算符产生的值是true；如果操作数是true，运算符产生的值是false。如下例所示。

```
bool x = false;
bool y = !x;
// 这里，y是true，而x是false
```

按位求补运算符～

这个运算符的操作数必须是整数基本类型或整型变量。结果是对操作数的每位取反。例如：

```
int j = 2;
int k = ~j; // k = -3; j = 2
```

为了更好地理解这个运算符是如何工作的，我们需要先把操作数转换成二进制数字，然后再把所有位取反。这个数字的二进制形式如下所示。

```
0000 0000 0000 0000 0000 0000 0000 0010
```

每一位取反后如下所示。

```
1111 1111 1111 1111 1111 1111 1111 1101
```

得到整数值是-3。

Sizeof运算符

这个一元运算符用来获取一个数据类型的字节长度。例如，sizeof(int)返回4。

2.7.2 算术运算符

有5种算术运算符：加减乘除和取模。下面我们介绍每种算数运算符。

加法运算符 +

加法运算符将两个操作数相加。操作数的类型必须可以转换成数值类型。例子如下。

```
byte x = 3;  
int y = x + 5; // y = 8
```

确保接收加法结果的变量有足够大的容量。例如，在下列代码中，k是-294967296，而不是 4 000 000。

```
int j = 2000000000; // 2,000,000  
int k = j + j; // 没有足够的存储空间，产生bug!
```

而下面代码是正确的。

```
long j = 2000000000; // 2,000,000  
long k = j + j; // k的值是4,000,000
```

减法运算符 -

减法运算符将两个操作数相减。操作数类型必须可以转换成数值基本类型。例子如下。

```
int x = 2;  
int y = x - 1; // y = 1
```

乘法运算符 *

这个运算符把两个操作数相乘。操作数的类型必须可以转换成数值基本类型。例子如下所示。

```
int x = 4;  
int y = x * 4;    // y = 16
```

除法运算符 /

这个运算符把两个操作数相除。左边的操作数是除数，右边的操作数是被除数，除数和被除数都必须可以转换成数值基本类型。如下例所示。

```
int x = 4;  
int y = x / 2;    // y = 2
```

请注意，如果被除数是0，执行除法操作时会出错。

用/操作符得到的除法结果总是一个整数。如果被除数不能除尽除数，将会忽略余数。例子如下所示。

```
int x = 4;  
int y = x / 3;    // y = 1
```

模运算符 %

这个运算符把两个操作数相除，然后返回余数。左边是除数，右边是被除数。除数和被除数都必须可以转换成数值基本类型。例如，下面操作的结果是2。

```
8 % 3
```

相等运算符

有两个相等运算符，==（等于）和!=（不等于），两个操作符都需要两个操作数，操作数可以是整型、浮点型、字符型或布尔型。相等运算符返回的结果是布尔型。

如下例所示，在对比后，c的值是true。

```
int a = 5;  
int b = 5;  
bool c = a == b;
```

另一示例如下。

```
bool x = true;  
bool y = true;  
bool z = x != y;
```

在对比后，z的值是false，因为x等于y。

2.7.3 关系运算符

关系运算符有4个：小于（<）、大于（>）、小于等于（<=）和大于等于（>=）。本节会分别介绍每种运算符。

小于、大于、小于等于、大于等于这些运算符都操纵两个操作数，操作数的类型必须是基本数值类型。关系操作返回结果是布尔型。

小于操作符（<）用于判断左边操作数的值是否小于右边操作数的值。例如，如下的操作返回的结果是false。

```
9 < 6
```

大于操作符（>）符判断左边操作数的值是否大于右边操作数的值。如下例，操作返回true。

```
9 > 6
```

小于等于操作符（<=）判断左边操作数的值是否小于等于右边操作数的值。例如，如下的操作的结果为false。

```
9 <= 6
```

大于等于操作符（>=）判断左边操作数的值是否大于等于右边操作数的值。例如，如下的操作返回true。

```
9 >= 9
```

2.7.4 条件运算符

条件运算符有3种：与运算符（&&），或运算符（||）以及?:运算符。具体内容如下。

&& 运算符

这个运算符把两个表达式作为操作数，两个表达式必须有返回值，而且返回值要转换成布尔型。如果两个操作数结果都为true，返回值为true；否则，返回false。如果左边操作数结果为false，右边操作数不再作判断。下例返回结果是false。

```
(5 < 3) && (6 < 9)
```

|| 运算符

这个运算符把两个表达式作为操作数，两个表达式必须有返回值，而且返回值要转换成布尔型。如果其中一个操作数的结果是true，返回true。如果左边操作数结果为true，右边的操作数不再作判断。下例返回结果是true。

```
(5 < 3) || (6 < 9)
```

?: 运算符

这个运算符需要3个操作数，语法如下。

```
expression1 ? expression2 : expression3
```

expression1 必须返回一个可转换成布尔型的值。如果*expression1*的结果是true，返回*expression2*；否则，返回*expression3*。

例如，下面的表达式把4赋值给x。

```
int x = (8 < 4) ? 2 : 4
```

2.7.5 位移运算符

位移运算符需要两个操作数，操作数必须可转换成整数基本类型。左边操作数表示要位移的值，右边操作数表示位移距离。下面是3种位移运算符。

- 左位移运算符 <<
- 右位移运算符 >>
- 无符号右位移运算符 >>>

左位移运算符 <<

左位移运算符向左位移一个数字，空位补0。 $n \ll s$ 就是将 n 向左位移 s 位。这相当于乘以2的 s 次方。

例如，整数1向左移3位（ $1 \ll 3$ ），结果为8。为了便于计算，我们把操作数转换成二进制数。

0000 0000 0000 0000 0000 0000 0000 0001

向左位移3位，如下所示。

0000 0000 0000 0000 0000 0000 0000 1000

结果等于8（相当于 1×2^3 ）。

另一条规则是：如果左边的操作数是一个int，那么只有前5位的位移距离会用到。换句话说，位移距离的范围必须在0到31之间。如果传递的数字大于31，也只有前5位会使用。这也就是说，如果 x 是int， $x \ll 32$ 与 $x \ll 0$ 是一样的； $x \ll 33$ 和 $x \ll 1$ 是一样的。

如果左边的操作数是long，只有前6位的位移距离会用到。换句话说，位移距离实际可用的范围就是0到63。

右位移运算符 >>

右位移运算符>>把左边的数字向右位移。 $n \gg s$ 的值是 n 向右位移 s 位，结果是 $n/2^s$ 。

例如， $16 \gg 1$ 等于8。为了证明这一点，我们用二进制表示16。

```
0000 0000 0000 0000 0000 0000 0001 0000
```

然后，把它向右移一位，如下所示。

```
0000 0000 0000 0000 0000 0000 0000 1000
```

结果等于8。

无符号右移 \gg

$n \gg s$ 的值依赖于 n 是正值还是负值。对于正值 n ，结果和 $n \gg s$ 是一样的。

如果 n 是负值，结果要看 n 的类型。如果 n 是 `int`，值是 $(n \gg s) + (2 \ll \sim s)$ 。如果 n 是 `long`，值是 $(n \gg s) + (2L \ll \sim s)$ 。

2.7.6 赋值运算符

赋值运算符有12个。

```
= += -= *= /= %= <<= >>= >>>= &= ^= |=
```

赋值运算符需要两个操作数，操作数必须是整数基本类型。左边操作数必须是一个变量。例如：

```
int x = 5;
```

除了赋值运算符 `=`，其他运算符工作方式都一样，我们可以把它们看成是两个运算符组合在一起。例如，`+=` 实际上就是 `+` 和 `=`。赋值运算符 `<<=` 有两个运算符，即 `<<` 和 `=`。

两部分赋值运算符先对两个操作数应用第一个运算符，然后把结果赋值给左边的操作数。例如，`x+=5` 和 `x=x+5` 是一样的。

`x -= 5` 相当于 `x = x - 5`。

$x \ll= 5$ 等于 $x = x \ll 5$ 。

$x \&= 5$ 和 $x = x \&= 5$ 可以得到同样的结果。

2.7.7 整型位运算符 $\& \mid \wedge$

位运算符 $\& \mid \wedge$ 对两个操作数逐位执行，操作数类型必须可以转换为 `int` 类型。 $\&$ 表示与运算符， \mid 是或运算符， \wedge 是异或运算符。例子如下所示。

```
0xFFFF & 0x0000 = 0x0000
0xF0F0 & 0xFFFF = 0xF0F0
0xFFFF | 0x000F = 0xFFFF
0xFFF0 ^ 0x00FF = 0xFF0F
```

2.7.8 逻辑运算符 $\& \mid \wedge$

逻辑运算符 $\& \mid \wedge$ 对两个操作数执行逻辑运算，操作数必须可转换成 `boolean` 类型。 $\&$ 表示与运算符， \mid 是或运算符， \wedge 是异或运算符。例子如下所示。

```
true & true = true
true & false = false
true | false = true
false | false = false
true ^ true = false
false ^ false = false
false ^ true = true
```

2.7.9 运算符优先级

在大多程序中，表达式中经常会出现乘法运算符，例子如下所示。

```
int a = 1;
int b = 2;
int c = 3;
int d = a + b * c;
```

以上代码执行后，`d` 的值是多少呢？如果答案是9，那么你错了。实

实际上正确答案是7。

乘法运算符*优先级高于加法运算符+，因此乘法运算总是在加法运算前执行。如果想要先执行加法，那么我们就需要用到圆括号。

```

int d = (a + b) * c;

```

这个表达式会使d的值得9。

表2-4列出了所有运算符的优先级顺序。同一栏中的运算符具有相同的优先级。

表2-4 运算符的优先级

运 算 符	
后缀运算符	[] . (params) expr++ expr--
一元运算符	++expr --expr +expr -expr ~ !
创建或类型转换	new (type)expr
乘法	* / %
加法	+ -
位移	<< >> >>>
关系	< > <= >=
相等	== !=
按位与	&

按位异或	\wedge
按位或	$ $
逻辑与	$\&\&$
逻辑或	$ $
条件	$?:$
赋值	$= \ += \ -= \ *= \ /= \ \% = \ \& = \ \wedge = \ = \ \ll = \ \gg = \ \>\> =$

请注意，圆括号的优先级最高，也能使表达式更清晰。例如，我们来看下面的代码。

```
int x = 5;
int y = 5;
boolean z = x * 5 == y + 20;
```

在比较后，z的值是true。但是，这个表达式非常不清晰。

我们用圆括号来重写最后一行代码。

```
bool z = (x * 5) == (y + 20);
```

结果没有变化，因为*和+比==拥有更高的优先级，但这样会使表达式看起来更清晰。

2.7.10 提升

有些一元运算符（如+、-和~）和二元运算符（如+、-、*、/）会导致自动提升，也就是提升到拥有更大存储空间的数据类型，如从byte类型提升到int类型。我们来看以下代码。


```
byte x = 5;  
byte y = -x; // 错误
```

虽然byte类型可以容纳-5，但第2行代码还是产生了一个奇怪的错误。错误的原因就在于一元运算符-会导致-x的结果提升为int。为了解决这个问题，需要把y的类型改为int类型，或做一个显式的窄化转换。

```
byte x = 5;  
byte y = (byte) -x;
```

对于一元运算符，如果操作数是byte、short或char，操作的结果就会提升为int。

对于二元运算符，提升的规律如下。

- 如果运算符是byte或short，两种运算符都会转换成int，结果也是int；
- 如果运算符中有一个是double，那么另一个运算符也转换成double，结果也是double；
- 如果运算符中有一个是float，那么另一个运算符也转换为float，结果也是float；
- 如果运算符中有一个是long，那么另一个运算符也转换成long，结果也是long。

例如，下列代码会产生一个编译错误。

```
short x = 200;  
short y = 400;  
short z = x + y;
```

我们可以把z改为int来修正，也可以对x+y执行显式的窄化转换。例子如下所示。

```
short z = (short) (x + y);
```

请注意，在x+y外边的圆括号是必需的。

2.8 注释

在所有代码中都加上注释，对每个类所提供的功能、每个方法所做的事情、每个字段包含的内容等做充分的解释，这是一种很好的做法。

C#中有两种类型的注释，它们都与C和C++的注释语法很相似。

- 传统注释：用/*和*/把注释内容包围起来。
- 行末注释：用双斜杠（//），在行中//后的剩余内容会被编译器所忽略。

例如，以下是描述一个方法的注释。

```
/*  
    toUpperCase 把字符串对象中的字符转换成大写  
*/  
public void toUpperCase(String s) {
```

以下是一个行末注释。

```
public int rowCount; // 数据库中的行数
```

传统注释不允许嵌套，这意味着以下例子中格式是不合法的。

```
/*  
/* comment 1 */  
comment 2 */
```

因为在第一个/*出现以后，第一个*/会结束注释。所以上面的注释中会有一个多余的“**comment 2 */**”，而这样会产生一个编译错误。

另外，行末注释可以包含任何内容，也包括字符/*和*/的序列，如下所示。

```
// /*这样注释是可以的*/
```

2.9 小结

本章介绍了C#的语言基础、基本概念和进一步学习前需要掌握的一些主题。这些主题包括：字符集、变量、基本类型、常量、运算符、运算符优先级和注释。

第3章我们将介绍语句，C#语言中另一个重要内容。

第3章 语句

计算机程序就是称之为语句的指令的汇编。C#中有许多类型的语句，其中有一些诸如if、while、for和switch这样的条件语句，它们能够决定程序的流程。本章将介绍C#语句，先作一个概述，然后详细介绍每一种语句。Return语句是跳出方法的语句，我们会在第4章介绍。

3.1 C#语句概览

在程序中，语句就是做事的指令。语句控制着程序执行的顺序。例如，给一个变量赋值就是一条语句。

```
x = z + 5;
```

甚至，一个变量声明也是一条语句。

```
long secondsElapsed;
```

相对来说，表达式就是运算符和操作数组合在一起来求值。例如， $z + 5$ 就是一个表达式。

在C#中，我们用分号来结束语句，而且多条语句可以写在同一行。

```
x = y + 1; z = y + 2;
```

但是，我们不推荐将多条语句写在同一行，因为这会使得代码的易读性变差。

注意

在C#中，空语句是合法的，它不做任何操作，例子如下所示。

```
;
```

有些表达式后边加上结束语句的分号，可以作为语句来使用。例如，`x++`是一个表达式，而如下所示就是一条语句。

```
x++;
```

语句可以组织为语句块。根据定义，语句块就是在花括号中的以下程序单元的序列。

- 语句
- 局部类声明
- 局部变量声明语句

我们可以用标签来标记一条语句和一个语句块。标签的命名与C#标识符命名遵循同样的规则，并且用冒号来结束。例如，下面语句就是用来标记`sectionA`的。

```
sectionA: x = y + 1;
```

下面是标记一个块的示例。

```
start:
{
    // 语句
}
```

用标签标记一条语句或者一个块的目的，就是能在`goto`语句中引用它。通常，我们认为用`goto`是一种不好的做法，本书也不加以讨论。

3.2 if语句

`if`是一个条件分支语句。`if`语句的语法有以下两种方式。

```
if (booleanExpression)
{
    statement(s)
}

if (booleanExpression)
```

```
{
    statement(s)
}
else
{
    statement(s)
}
```

如果booleanExpression的值为true，紧跟在if语句后的语句块中的语句将会执行；如果值为false，if语句块中的语句不会执行；如果booleanExpression值为false并且有else语句块，则会执行else语句块中的语句。

例如，在下面的if语句中，如果x大于4，将会执行if语句块。

```
if (x > 4)
{
    // 语句
}
```

在下面示例中，如果a大于3，会执行if语句块；否则，会执行else语句块。

```
if (a > 3)
{
    // 语句
} else
{
    // 语句
}
```

注意，好的编码风格要求语句块中要有缩进。

在if语句中，如果要计算一个bool，不一定非要像下面这样使用==符号。

```
boolean fileExist = ...
if (fileExist == true)
```

我们也可以简写为以下格式。

```
if (fileExists)
```

同样，如下语句所示。

```
if (fileExists == false)
```

我们也可以写成以下格式。

```
if (!fileExists)
```

如果要计算的表达式写在一行太长，我们建议在后边的行中使用两个字符的缩进，例子如下所示。

```
if (numberOfLoginAttempts < numberOfMaximumLoginAttempts  
    || numberOfMinimumLoginAttempts > y)  
{  
    y++;  
}
```

但是，这样可能会导致所谓的悬挂else（dangling else）问题。

```
if (a > 3)  
    a++;  
else  
    a = 3;
```

我们来看如下的示例（注意：在C#代码中，**System.Console.WriteLine**用于打印字符串和值）。

```
if (a > 0 || b < 5)  
    if (a > 2)  
        System.Console.WriteLine("a > 2");  
    else  
        System.Console.WriteLine("a < 2");
```

因为else语句要对应哪个if语句不够明确，所以else语句“悬挂”了起来。else语句通常是与紧挨着的if语句对应的。使用花括号就会使代码更清楚。

```
if (a > 0 || b < 5)
{
    if (a > 2)
    {
        System.Console.WriteLine("a > 2");
    }
    else
    {
        System.Console.WriteLine("a < 2");
    }
}
```

如果有多个选项，我们也可以使用带有一系列else语句的if语句。

```
if (booleanExpression1)
{
    // 语句
}
else if (booleanExpression2)
{
    // 语句
}
...
else
{
    // 语句
}
```

例子如下所示。

```
if (a == 1)
{
    System.Console.WriteLine("one");
}
else if (a == 2)
{
    System.Console.WriteLine("two");
}
else if (a == 3)
{
    System.Console.WriteLine("three");
}
else
{
    System.Console.WriteLine("invalid");
}
```



```
}
```

这个例子中，后面紧跟着一个if的else语句没有使用括号。在本章的3.8节中，我们还会看到关于switch语句的讨论。

3.3 while语句

在很多情况下，我们可能想要在一行中多次执行一个动作。换句话说，如果想重复执行一个语句块，最直接的实现方式，就是重复代码行。例如，使用以下代码来实现蜂鸣声。[\[1\]](#)

```
System.Console.Beep();
```

然后，使用如下这条语句来等待半秒钟。

```
System.Threading.Thread.Sleep(500);
```

因此，要产生3次蜂鸣，每次蜂鸣间隔500毫秒，我们可以直接重复相同的代码。

```
System.Console.Beep();  
System.Threading.Thread.Sleep(500);  
System.Console.Beep();  
System.Threading.Thread.Sleep(500);  
System.Console.Beep();
```

但是，有些情况是没办法用重复代码来实现的，例如以下情况。

- 重复的次数多于5次，这意味着代码行数至少增加5倍。即使只需要修改语句块中的一行代码，我们也必须要修改相同行的副本。
- 重复的次数无法预先确定。

更聪明的方法是把要重复的代码放到一个循环中。这样，我们只需要编写代码一次，然后通知C#执行代码任意多次就好。创建循环的一种方法是使用while语句，这是本节要讨论的主题。另一种方法是使用for语句，我们将在3.5节介绍。

While语句的语法如下。

```
while (booleanExpression)
{
    statement(s)
}
```

这里，当booleanExpression值为true时，statement (s) 会执行。如果花括号中只有一条语句，我们可以省略花括号。但是为了更清晰，即使是一条语句，我们也应该使用花括号。

下面是while语句的示例，当i小于3时，代码打印输出整数i。

```
int i = 0;
while (i < 3)
{
    System.Console.WriteLine(i);
    i++;
}
```

注意，循环中的代码的执行取决于值i，它在每次循环后加1，直到达到3为止。

生成间隔500毫秒的3次蜂鸣的代码如下。

```
int j = 0;
while (j < 3) {
    System.Console.Beep();
    try
    {
        Thread.currentThread().sleep(500);
    }
    catch (Exception e)
    {
    }
    j++;
}
```

有时，我们会使用值总是为true的表达式（例如布尔常量true）来计算，然后通过break语句来跳出循环。

```
int k = 0;
```

```
while (true)
{
    System.Console.WriteLine(k);
    k++;
    if (k > 2) {
        break;
    }
}
```

我们会在3.6节中介绍break语句。

3.4 do-while语句

do-while语句与while语句类似，只不过相应的语句块至少要执行一次。它的语法如下。

```
do
{
    statement(s)
} while (booleanExpression);
```

在do-while语句中，我们把要执行的语句放在关键字do的后边。与while语句一样，如果只有一条语句，我们也可以省略掉花括号，但是为了清楚起见，我们最好还是使用花括号。

以下是do-while语句的示例。

```
int i = 0;
do
{
    System.Console.WriteLine(i);
    i++;
} while (i < 3);
```

打印输出的内容如下。

```
0
1
2
```

下面的do-while语句展示：即使使用j的初始值来测试表达式j < 3的结果是false，do语句块中的代码至少也会执行一次。

```
int j = 4;
do
{
    System.Console.WriteLine(j);
    j++;
} while (j < 3);
```

打印输出的内容如下。

```
4
```

3.5 for语句

for语句和while语句类似，即用它封装那些需要多次执行的代码。但是，for要比while更复杂。

for语句以一个初始值开始，随后是每次循环都会计算的一个表达式，最后是表达式结果为true时会执行的一个语句块。每次循环迭代语句块执行之后，也会执行一条更新语句。

for语句语法如下。

```
for ( init ; booleanExpression ; update ) {
    statement(s)
}
```

init是一个初始值，它会在第一次循环迭代前执行；booleanExpression是一个布尔表达式，当它的结果为true时，将会执行语句；update是在语句块执行完后执行的一条语句。init、expression和update都是可选的。

for语句只会在满足以下条件之一的时候停止。

- booleanExpression结果为false
- 执行break或continue语句

- 产生运行时错误

for 语句通常会在初始化部分声明一个变量，然后给它赋一个值。声明的变量可以在expression和update部分看到，也可以在语句块中看到。

例如，下面for语句循环5次，每次都打印输出i的值。

```
for (int i = 0; i < 3; i++)  
{  
    System.Console.WriteLine(i);  
}
```

for语句一开始声明了一个int类型的名称为i的变量，并把0赋值给它。

```
int i = 0;
```

然后它表达式 $i < 3$ ，因为i等于0，其结果为true，因此，会执行语句块，并且会打印输出i的值。然后执行更新语句 $i++$ ，i增加到1。第1次循环结束。

for语句然后再次判断 $i < 3$ ，因为i等于1，结果又是true，执行语句块，打印输出1到控制台，然后，执行更新语句 $i++$ ，i增加到2。第2次循环结束。

接下来，for语句判断表达式 $i < 3$ ，因为i等于2，结果为true，执行语句块，打印输出2到控制台，然后，执行更新语句 $i++$ ，使得i等于3。第3次循环结束。

此时，for语句再次判断表达式 $i < 3$ ，结果为false，循环结束。

控制台的打印输出如下。

```
0  
1  
2
```

注意，变量i在其他地方是不可见的，因为它声明于for循环中。

还要注意的，如果for语句块中只有一条语句，我们可以去掉花括号，所以上面for语句的示例可以重写为以下格式。

```
for (int i = 0; i < 3; i++)  
    System.Console.WriteLine(i);
```

然而，即使只有一条语句我们也建议使用花括号，这会使代码更加清楚。

下面是for语句的另一个示例。

```
for (int i = 0; i < 3; i++)  
{  
    if (i % 2 == 0)  
    {  
        System.Console.WriteLine(i);  
    }  
}
```

它会循环3次，每次循环都会测试i的值。如果i是偶数，它的值会打印输出。for循环的结果如下。

```
0  
2
```

下面的for循环与上一示例很相似，但是使用i+=2作为更新语句。结果它只循环了两次，即当i等于0和2时。

结果如下所示。

```
0  
2
```

变量递减的语句也会经常用到。我们来看下面的for循环。

```
for (int i = 3; i > 0; i--)  
{  
    System.Console.WriteLine(i);  
}
```

打印输出如下。

```
3
2
1
```

for语句的初始化部分是可选的。在下面的for循环中，变量j在循环之外声明，所以我们可以将for语句块之外的其他地方的代码中使用j。

```
int j = 0;
for ( ; j < 3; j++) {
    System.Console.WriteLine(j);
}
// j在这里是可见的
```

如前所述，更新语句是可选的。下面的for语句把更新语句放到了语句块的末尾，结果是相同的。

```
int k = 0;
for ( ; k < 3; )
{
    System.Console.WriteLine(k);
    k++;
}
```

理论上，我们甚至可以忽略booleanExpression部分。例如，下面的for语句中就没有booleanExpression部分，循环只有在执行break语句时才能终止。更多信息请参见3.6节。

```
int m = 0;
for ( ; ; )
{
    System.Console.WriteLine(m);
    m++;

    if (m > 4)
    {
        break;
    }
}
```

如果比较for和while，我们会发现while语句通常可以用for语句来替

换，如下例所示。

```
while (expression)
{
    ...
}
```

它通常会写成以下格式。

```
for ( ; expression; )
{
    ...
}
```

注意

另外，foreach能够遍历一个数组或一个集合，参见第5章和第13章对foreach的介绍。

3.6 break语句

break语句用于从封闭的do、**while**、for或switch语句中跳转。除此之外，在任何地方使用break都会产生编译错误。

例如，我们来看如下代码。

```
int i = 0;
while (true)
{
    System.Console.WriteLine(i);
    i++;
    if (i > 3) {
        break;
    }
}
```

其结果如下所示。

```
0
1
2
```


注意，`break`跳出循环并且没有执行语句块中剩下的语句。

下面是`break`的另一个示例，这次是在`for`循环中。

```
int m = 0;
for ( ; ; )
{
    System.Console.WriteLine(m);
    m++;
    if (m > 4) {
        break;
    }
}
```

3.7 `continue`语句

`continue`语句和`break`语句类似，但是它只能终止当前迭代的执行，然后重新开始下一次迭代。

例如，下面的代码打印输出数字0到9，但是5除外。

```
for (int i = 0; i < 10; i++)
{
    if (i == 5) {
        continue;
    }
    System.Console.WriteLine(i);
}
```

当`i`等于5时，`if`语句的表达式计算结果为`true`，触发调用`continue`语句。结果，下面打印输出`i`的值的语句没有执行，而是继续下一次循环，也就是`i`等于6。

3.8 `switch`语句

本章的最后，我们要介绍一系列`else if`语句中的一种替代形式，即

switch语句。Switch允许我们根据一个表达式的返回值，从可选代码中选择一个块语句来执行。switch语句中的表达式，必须返回int、String或枚举类型的值。

注意

String类我们会在第5章中介绍，枚举类型的值会在第10章介绍。

switch语句的语法如下。

```
switch(expression)
{
    case value_1 :
        [statement(s);]
        [break | goto label;]
    case value_2 :
        [statement(s);]
        [break | goto label;]
    .
    .
    .
    case value_n :
        [statement(s);]
        [break | goto label;]
    default:
        [statement(s);]
        [break | goto label;]
}
```

在每个case后的每条语句都可以用break跳出循环或用goto跳转到一个标签。这些语句和跳转语句都是可选的。

下面是一个switch语句的示例，如果i的值是1，会打印输出“One player is playing this game”；如果值是2，会打印输出“Two players are playing this game.”；如果值是3，会打印输出“Three players are playing this game.”；对于任意其他值，都会打印输出“You did not enter a valid value.”。

```
int i = ...;
switch (i)
{
    case 1 :
```

```
        System.Console.WriteLine(
            "One player is playing this game.");
        break;
    case 2 :
        System.Console.WriteLine(
            "Two players are playing this game.");
        break;
    case 3 :
        System.Console.WriteLine(
            "Three players are playing this game.");
        break;
    default:
        System.Console.WriteLine(
            "You did not enter a valid value.");
        break;
}
```

3.9 小结

C#程序的执行顺序是通过语句控制的。在本章中，我们学习了以下控制语句：if、while、do-while、for、break、continue和switch。理解如何使用这些语句对于编写正确的程序至关重要。

[1] What this line of code and the following lines of code do will become clear after you read Chapter 4.

在学习了第4章后，你就会明白这行代码及后面的代码行。

第4章 对象与类

本章介绍对象与类。如果你是OOP的初学者，可能需要认真阅读本章，因为对OOP的充分理解，是编写高质量程序的关键。

本章首先介绍什么是对象以及类是由哪些部分组成的。然后，我们会讲述在C#中如何用关键字new来创建对象，如何在内存中存储对象，如何将类组织到命名空间中，如何用访问控制来实现封装以及C#如何管理没有使用过的对象。另外，我们还会介绍方法重载和静态类成员。

4.1 C#对象是什么

当用OOP语言开发应用程序时，我们需要创建一个模块来模拟一个解决问题的真实情景。以一个公司工资单应用程序为例，它能计算雇员的税后工资和应缴纳的个人所得税。这样的应用程序，通常会用一个Company对象来表示公司，一个Employee对象表示为公司工作的雇员，一个Tax对象表示每个雇员需缴纳的所得税，依次类推。在开始编写这样的应用程序前，我们需要先了解C#对象是什么以及如何创建C#对象。

我们先来看看生活中的对象。对象无处不在，包括生命（人、宠物等等）和非生命（汽车、房子、街道等等），实体（书、电视等）和抽象体（爱情、知识、税率、规则等）。每个对象都有两个特征，即属性和行为。例如，汽车的一些属性如下所示。

- 颜色
- 轮胎数
- 车牌号

另外，汽车能够执行以下行为。

- 行驶
- 刹车

我们再来看一个例子，狗有以下属性：颜色、年龄、品种、体重等，它也能吠、跑、撒尿和嗅等。

C#对象也有属性以及它能执行的行为。在C#中，属性叫作字段（field），行为叫作方法（method）。其他编程语言中可能有不同的叫法。例如，方法也经常叫作函数。

字段和方法都是可选的，也就是说，有些C#对象可能没有字段但是有方法，而其他一些对象可能有字段但是没有方法。当然，有些对象既有属性也有方法，而有些对象则属性和方法都没有。

如何在C#中创建一个对象呢？创建一个对象，首先需要有一个类，它是对象的蓝图。接下来我们介绍类。

4.2 C#类

类是创建相同类型对象的蓝图或模板。如果有一个Employee类，那么我们就能够创建任意多个Employee对象。要创建Street对象，我们首先需要有一个Street类。类决定了能够得到何种对象。例如，如果我们创建一个有Age字段和Position字段的Employee类，那么所有通过Employee类创建的Employee对象，也都会有Age字段和Position字段。字段不会多也不会少。类决定了对象。

总之，类是一种OOP工具，能使程序员创建出问题的抽象。在OOP中，抽象是用编程对象表示真实世界中对象的行为。程序对象也不需要真实世界中对象的细节。例如，如果在工资单应用程序中，Employee对象只需要工作和领工资，那么Employee类只需要两个方法，即Work和ReceiveSalary。OOP抽象忽略了一些事实，现实世界中的雇员，还能够做包括吃喝拉撒的很多事情。

类是C#程序的基础构建块。C#初学者在编写一个类的时候要考虑三件事情。

- 类的名称
- 字段
- 方法

类中还会有一些其他内容，我们会在后面介绍。

类声明必须由关键字`class`和紧随其后的类的名称构成。类还拥有用花括号括起来的主体。它的通用语法如下。

```
class className
{
    [class body]
}
```

例如，程序清单4-1展示了一个名为Employee的C#类，其中粗体行是类的主体。

程序清单4-1 Employee类

```
class Employee
{
    int Age;
    double Salary;
}
```

注意

按照惯例，类名中每个单词的首字母要大写。下面这些名称就遵循了这个惯例，例如Employee、Boss、DateUtility、PostOffice和RegularRateCalculator。这种命名惯例叫作Pascal命名惯例。还有一种惯例名为骆驼式命名惯例，则是除首个单词外的每个单词的首字母都大写，例如postOffice、dateUtility和crayon。

在C#中，类定义必须保存成一个带`.cs`后缀的文件。文件名不一定非要和类名相同。

注意

为了使面向对象程序中的模型可见，软件工程师经常使用统一建模语言（Unified Modeling Language, UML）所定义的标准标记。在UML的类图中，类用一个矩形表示，它包含三个部分，顶部是类名，中间部分是字段的列表，底部是方法的列表，如图4-1所示。如果字段和方法的显示不是很重要，我们可以把它们隐藏起来。

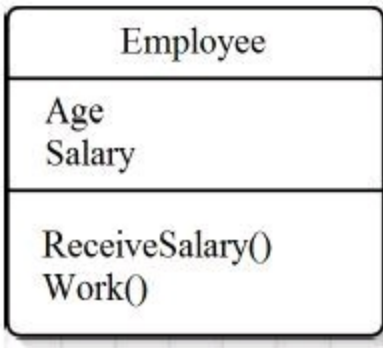


图4-1 UML类图中的Employee类

4.2.1 字段

字段是可见的。它们可能是值类型或对象的引用。例如，程序清单4-1中的Employee类有两个字段Age和Salary，它们很可能是值类型。但是，字段也可能是一个对象的引用。例如，Employee类有一个类型为Address的Address字段，Address是表示街道地址的一个类。

```
Address address;
```

换句话说，一个对象可以包含其他的对象，这就好像是前一个对象的类包含了变量，而变量引用了后一个对象的类。字段的名称应该遵循Pascal命名惯例。字段的每个单词首字母都是大写的。例如，下面就是一些很好的字段的名称：**Age**、**MaxAge**、**Address**、**ValidAddress**和**NumberOfRows**。

4.2.2 方法

方法定义了一个类的对象（或实例）能够做的动作。方法有一个声明部分和一个主体。声明部分由返回值、方法名称和参数列表构成。主体包含所执行的操作的代码。

要声明一个方法，我们用如下语法。

```
returnType methodName (ListOfArguments)
```

一个方法的返回类型可以是一个固定数据类型、一个对象或void。

`void`返回类型意味着方法没有返回任何内容。方法的说明也称为方法的签名。

例如，下面的**GetSalary**方法返回一个`double`。

```
double GetSalary()
```

GetSalary方法不接受参数。

我们再看一个例子，这里返回`Address`对象的一个方法。

```
Address GetAddress()
```

下面是接受一个参数的方法。

```
int Negate(int number)
```

如果一个方法接受多个参数，两个参数间要用逗号隔开。例如，下面的`Add`方法接受两个`int`参数并且返回`int`类型。

```
int Add(int a, int b)
```

4.2.3 **main**方法

有一种特殊的方法叫作**Main**，它提供应用程序的入口点。程序通常有许多类，但只有一个类需要有**Main**方法。这个方法允许包含它的类来调用它。

在C#中，**Main**的返回值可能是`void`或`int`。它也可以接受一个`string[]`参数。下面这些是合法的**Main**方法，在程序中只需要用到其中之一。

Main方法的签名如下。

```
static void Main()  
static void Main(string[] args)  
static int Main()  
static int Main(string[] args)
```


想知道为什么要在Main前要加上“static”吗？我们会在本章末尾揭晓答案。

如果Main方法接受参数，我们可以在运行这个类时传递参数给它。要传递参数，我们在可执行名称后输入参数即可。两个参数之间用空格隔开。

```
MyApp arg1 arg2 arg3
```

所有参数都必须作为字符串来传递。例如，运行Counter类时，要传递两个参数“1”和“safeMode”，我们需要这样输入。

```
Counter 1 safeMode
```

在Visual C# 2010 Express中，我们如果要给Main方法传递行参数，右键点击“Solution Explorer”中的项目名称，点击“Properties”，然后选中“Properties”面板中的“Debug”。我们就会看到如图4-2所示的“Debug”面板，在“Command line arguments”文本框中输入参数，按Ctrl+S键保存参数。

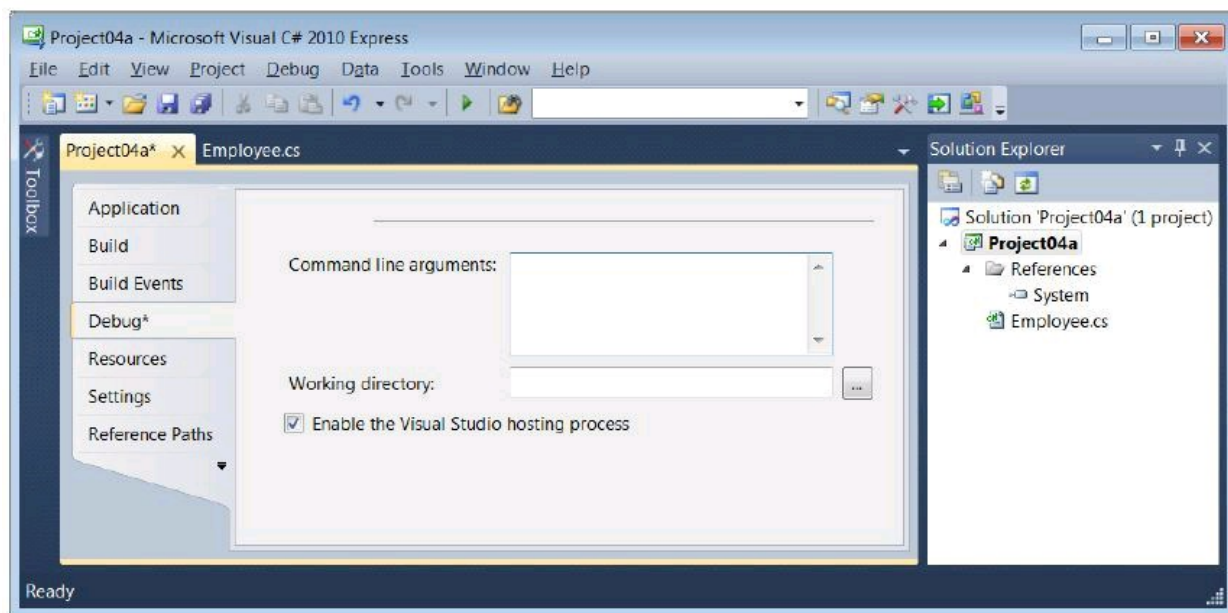


图4-2 在Visual C# 2010 Express中传递行参数

4.2.4 构造函数

每个类必须至少有一个构造函数，否则，这个类就不能创建对象，也没有任何用处。因此，如果类中没有明确定义一个构造函数，编译器会自动添加一个构造函数。

构造函数用于构造对象。构造函数看上去和方法一样，所以有时构造函数也被叫作构造方法。但是和方法不同的是，构造函数没有返回值，甚至没有void返回类型。另外，构造函数的名称必须和类的名称保持一致。

构造函数的语法如下所示。

```
constructorName (ListOfArguments)
{
    [constructor body]
}
```

构造函数可以有零个参数，这种情况下叫作不带参数的构造函数。构造函数的参数可以用于初始化对象中的字段。

如果C#编译器为类添加了一个不带参数的构造函数，那是因为这个类没有构造函数，增加的构造函数是隐式的，也就是说，它在源文件中不显示出来。但是，如果类中有一个构造函数（不管它有几个参数），编译器都不会再为这个类添加构造函数。

类中可以有多多个构造函数，只是每个构造函数要有不同的参数集。例如，程序清单 4-2为程序清单4-1中的Employee类添加了两个构造函数。

程序清单4-2 Employee类中的构造函数

```
public class Employee
{
    public int Age;
    public double Salary;
    public Employee() {
    }
}
```

```
public Employee(int AgeValue, double SalaryValue) {  
    Age = AgeValue;  
    Salary = SalaryValue;  
}  
}
```

第二个构造函数特别有用。如果没有它，要给age和position赋值的话，我们就需要编写额外的代码来初始化字段。

```
Employee employee = new Employee();  
employee.Age = 20;  
employee.Salary = 90000.00;
```

使用第二个构造函数，我们可以在创建对象的同时传递值。

```
new Employee(20, 90000.00);
```

关键字new是一个新鲜事物，我们会在4.3节中学习如何使用它。

4.2.5 UML类图中的类成员

图4-3描绘了UML类图中的一个类。这个图为所有字段和方法提供了一个快速的概括。UML允许包含字段的类型和方法签名。例如，图4-3表示了一个Book类，它有五个字段和一个方法。

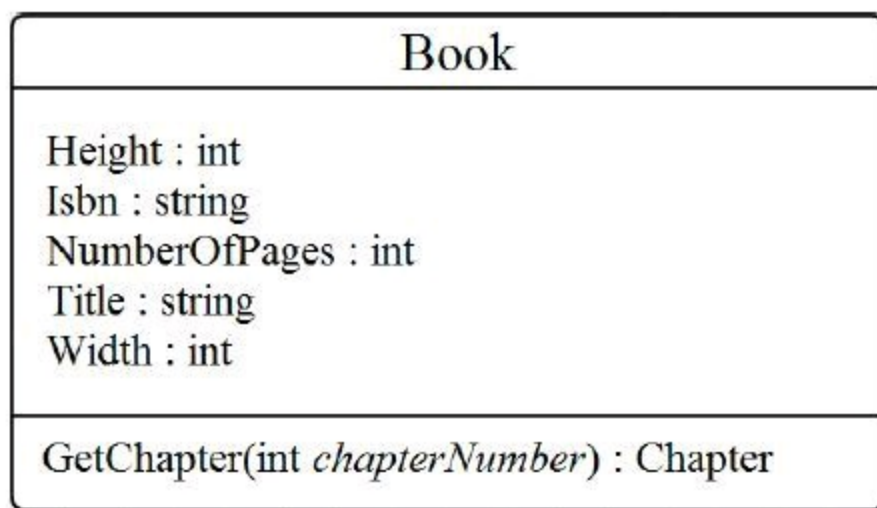


图4-3 类图中包含的类成员的信息

注意，在UML类图中，字段和它的类型是用冒号分开的。方法的参数列表在圆括号中给出，而方法的返回类型写在一个冒号之后。

4.3 创建对象

现在，我们已经知道了如何创建一个类，是时候学习如何通过一个类来创建对象了。对象也叫作实例。我们经常用“构造（construct）”来代替“创建（create）”，例如构造一个Employee对象。

还有一个经常用到的术语是实例化（*instantiate*）。实例化Employee类和创建Employee实例的意思是一样的。

创建对象可以有很多种方法，但是最常用的是使用关键字new。new总是在实例化类时伴随着构造函数而出现。例如，我们要创建一个Employee对象。

```
new Employee();
```

很多时候我们需要把一个创建好的对象赋予一个对象变量（或引用变量），这样以后我们就能够操作这个对象了。为了实现这一目的，我们就需要声明和该对象相同类型的一个对象引用，例子如下。

```
Employee employee = new Employee();
```

这里，employee是Employee类型的一个对象引用。

一旦有了对象，通过赋值给该对象的对象应用，我们就可以调用它的方法并访问它的字段。我们用点（.）来调用方法或字段，例子如下所示。

```
objectReference.MethodName  
objectReference.FieldName
```

例如，以下的代码创建了一个Employee对象并为它的字段Age和Salary赋值。

```
Employee employee = new Employee();
```

```
employee.Age = 24;  
employee.Salary = 50000;
```

当创建了一个对象时，CLR也会执行初始化，将默认值赋予字段。

注意，我们不需要显式地销毁对象来释放内存。CLR中的垃圾回收器会处理它。但是，这并不意味着可以随心所欲地创建对象，因为内存仍然是有限的，而且启动垃圾回收器也要花时间。这就是说，内存还是有可能被耗尽。

4.4 null关键字

一个引用变量引用一个对象。但有时引用变量会没有值（该变量没有引用一个对象）。我们说这样的引用变量有一个null值。例如，下面类层级引用变量是Book类型，但没有赋值。

```
Book book; // book是null
```

如果在方法中声明了一个局部引用变量但没有给它赋一个对象，在随后用到它时，我们需要为它赋一个null值，来避免编译器报错。

```
Book book = null;
```

创建实例时会初始化类层级的引用变量，因此我们不需要为它们赋null值。

试图访问null变量引用的字段或方法时会报错，如下面的代码所示。

```
Book book = null;  
Console.WriteLine(book.title); // 报错，因为book是null
```

我们可以用运算符==来测试引用变量是否为null。例子如下所示。

```
if (book == null)  
{  
    book = new Book();  
}
```

```
Console.WriteLine(book.title);
```

4.5 内存中的对象

当我们在类中声明一个对象时，无论是在类层级还是在方法层级中，都会为已赋值给变量的数据分配内存空间。对于值类型，我们很容易计算所占用的内存数量。例如，声明一个int需要4个字节，声明一个long要用8个字节。但是，引用变量的计算则是不同的。

程序运行时会给数据分配一些内存空间。这些数据空间逻辑上分为两部分，栈和堆，栈中分配值的类型，堆中存储对象。

在我们声明一个值类型时，在栈中分配一些字节。当我们声明一个引用变量时，栈中也会分配一些字节，但是内存没有包含对象数据，它包含的是堆中的对象的地址。也就是说，我们作如下的声明。

```
Book book;
```

这时会为引用变量book预留一些字节。book的初始值是null，因此还没有为它赋一个对象。我们写如下代码。

```
Book book = new Book();
```

我们创建了一个Book实例，它存储在堆中并把实例的地址赋予引用变量book。C#引用变量和C++的指针类似，只是我们不能操纵引用变量。在C#中，引用变量用于访问对象所引用的成员。因此，如果Book类有一个名为Review的public方法，我们就可以用下面的语法调用这个方法。

```
book.Review();
```

一个对象可以由多个引用变量引用。例子如下所示。

```
Book myBook = new Book();  
Book yourBook = myBook;
```

第二行把myBook的值复制给了yourBook。结果，yourBook现在和

myBook一样，引用了同一个Book对象。

图4-4说明了myBook和yourBook所引用的Book对象的内存分配情况。

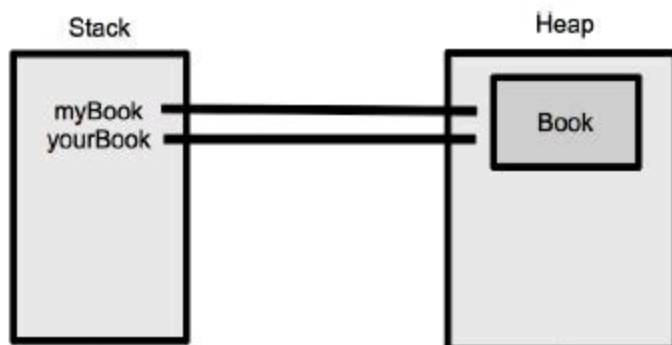


图4-4 两个变量引用一个对象

而另一方面，以下代码则创建了两个不同的Book对象。

```
Book myBook = new Book();  
Book yourBook = new Book();
```

这段代码的内存分配如图4-5所示。

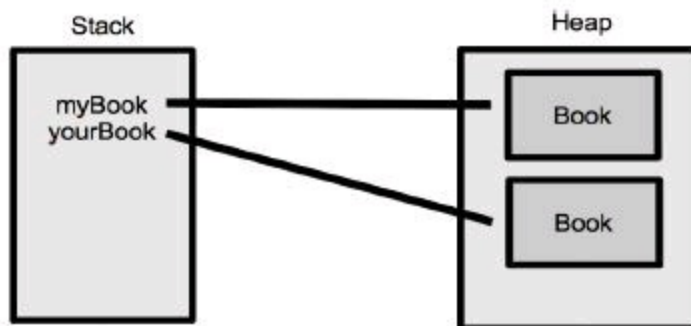


图4-5 两个变量引用两个对象

那么，一个对象包含另一个对象的情况是怎样的呢？例如，我们看一下程序清单4-3，它展示了包含Address类的一个Employee类。

程序清单4-3 包含另一个类的Employee类

```
public class Employee
{
    Address address = new Address();
}
```

当我们用下面代码创建一个Employee对象时，也创建了一个Address对象。

```
Employee employee = new Employee();
```

图4-6 描绘了堆中每个对象的位置。

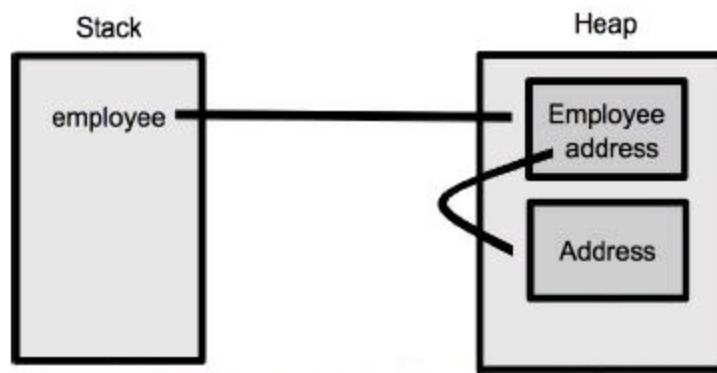


图4-6 一个对象包含另一个对象

实际上，Address对象并不是真的在Employee对象里边。Employee对象中的address引用变量引用了Address对象，这样就允许Employee对象操纵Address对象。因为在C#中，除非通过分配了对象地址的一个引用变量，我们没有其他办法可以访问对象，也没有办法访问Employee对象中的Address对象。

4.6 C#命名空间

命名空间允许我们为自己的类型创建真正独立的名称，以组织我们的类和其他类型并避免名字冲突。命名空间可以包含任意多个类和其他类型，甚至还可以包含其他命名空间。如下代码是在一个命名空间中声明的类。

```
namespace MyNamespace
```



```
{  
    class MyClass  
    {  
    }  
}
```

如果没有把类声明在一个命名空间中，C#编译器会加上一个默认的命名空间，即`global`。

在多个声明中定义一个命名空间也是可能的，如下所示。

```
namespace YourNamespace  
{  
    class Class1  
    {  
    }  
}  
  
namespace YourNamespace  
{  
    class Class2  
    {  
    }  
}
```

.NET Framework也用命名空间来组织它的类库。例如，我们曾用于输出字符串的`Console`类就属于命名空间`System`。

为了在一个命名空间中使用类型，我们要在文件开始的地方使用`using`指令。例如，如果声明以下内容，我们就可以在代码中使用`System`命名空间的成员。

```
using System;
```

因此，我们可以简写为以下格式。

```
Console.WriteLine("Program terminated.");
```

如果没有导入`System`，我们就必须包含要使用的完全限定类名称，

如下所示。

```
System.Console.WriteLine("Program terminated.");
```

4.7 封装和类的访问控制

OOP的一条原则，就是封装机制，即保护一个对象需要安全的部分，只公开其可以确保安全公开的部分。电视机就是封装的一个很好的例子。电视机内部有成千上万个电子元器件，共同构成能够接收信号并把信号转换成图像和声音的部分。但是这些元器件不能展示给用户，因此Sony和其他的生产厂商用硬金属或塑料盖把它们封装起来，很难打开。为了让电视更便于使用，它将一些按钮裸露在外部，用户可以通过它们开机和关机、调节色彩以及调节音量等。

回到OOP的封装，我们来看一个能加密和解密消息的类的例子。这个类公开了Encode和Decode两个方法，类的使用者可以访问这两个方法。它内部有很多用于存储临时数值的变量以及其他执行支持任务的方法。类的所有者隐藏了这些变量和其他方法，因为允许访问它们的话可能会危害加密和解密算法的安全性。而且，公开太多的内容，会使得类很难用。在后面我们将会看到，封装是一个很强大的特性。

C#通过访问控制来实现封装。访问控制通过访问控制修饰符来管理。访问控制修饰符也可以简称为访问修饰符。C#中有四种访问修饰符：**public**、**protected**、**internal**和**private**。

访问控制修饰符可以用于类或类成员。类可能是**public**的，也可能是**internal**的。默认情况下，类是**internal**的，除非明确地声明为**public**。其他任何命名空间中的类型都可以访问**public**类。相反，**internal**类只允许同一个命名空间内的其他类型访问。没有访问修饰符的类是**internal**访问级别的。

例如，下面代码段中的ClassA是一个**public**类。

```
namespace CompanyA
{
    public class ClassA
    {
```

```
}  
}
```

命名空间CompanyA中的任何类型可以访问ClassA，命名空间**CompanyA**之外的其他类型也可以访问ClassA。另一方面，下面例子的类ClassB和ClassC只能被命名空间CompanyA中的其他类型访问。这是因为ClassB和ClassC都是internal访问权限的。

```
namespace CompanyA  
{  
    internal class ClassB  
    {  
    }  
  
    class ClassC  
    {  
    }  
}
```

默认命名空间的类可以被任意其他命名空间中的任何类型访问，即使这个类不是public的。例如，这段代码中的Book类能够使用Chapter类，Chapter没有放在一个命名空间中，因此它在默认命名空间中。

```
class Chapter  
{  
}  
  
namespace MyCompany  
{  
    class Book  
    {  
        Chapter chapter = new Chapter();  
    }  
}
```

另一方面，因为House是命名空间YourCompany的internal类，它不能被另一个命名空间中的类访问，所以这段代码无法编译。

```
namespace YourCompany
{
    class House
    {
    }
}

namespace MyCompany
{
    class Person
    {
        House house = new House(); // compile error
    }
}
```

我们可以把House类改为public，这样其他命名空间就可以访问它，如下所示。

```
namespace YourCompany
{
    public class House
    {
    }
}

namespace MyCompany
{
    using YourCompany;
    class Person
    {
        House house = new House(); // 编译错误
    }
}
```

注意，不带修饰符的命名空间总是public的。

现在我们来看看类成员。类成员可以是以下5个访问级别之一。

- **public**: 访问不受限制。
- **protected**: 访问仅限于包含类或从包含类派生的类型。
- **internal**: 访问仅限于当前程序集。
- **protected internal**: 访问仅限于当前程序集或从包含类派生的类型。

- **private**: 访问仅限于包含类。

public类成员可以由任何其他类访问，只要这些类能够访问包含类成员的类。

例如，.NET Framework类库中System.Object类的ToString方法是**public**的。因此，如果要构造一个Object对象，我们就可以调用它的ToString方法，因为ToString是**public**的。

```
Object obj = new Object();  
obj.ToString();
```

我们可以用下面的语法访问类成员。

```
referenceVariable.memberName
```

在上面的代码中，obj是System.Object的一个实例的引用变量，ToString是System.Object类中定义的一个方法。

protected类成员有更多访问级别限制。只有包含类或包含类的子类可以访问它。类的**private**成员只能在同一类内部可以访问。

我们再来看看构造函数。构造函数的访问级别，和字段与方法的是是一样的。因此，构造函数也有**public**、**protected**、**internal**、**protected internal**和**private**这些访问级别。你可能会认为所有的构造函数都必须是**public**的，因为构造函数的目的就是使类能够实例化。但出人意料的是，不是这样。有些构造函数是**private**的，所以我们不能用关键词**new**来实例化它们的类。**private**的构造函数通常用于单体类。如果对这个话题感兴趣，你可以从互联网上很容易地找到关于这个话题的文章。

注意

在一个UML类图中，你可以包含关于类成员访问级别的信息。**public**成员前边会加上+，**protected**成员前会加上#，**private**成员前会加上-。没有加前缀的成员会被看作是拥有默认的访问级别。如图4-7所示，Manager类的成员有不同的访问级别。

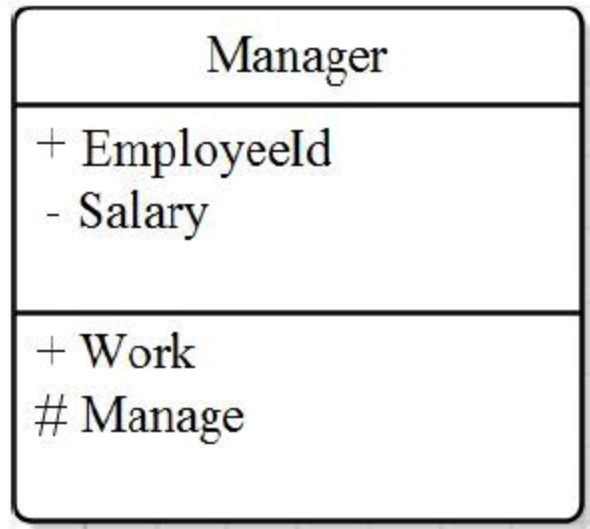


图4-7 UML类图中包含了类成员的访问级别

4.8 关键字**this**

我们可以在任意方法和构造函数中用关键字**this**来表示当前对象。例如，如果有一个类级字段与局部变量具有相同的名称，我们就可以用以下语法来表示类级字段。

```
this.field
```

它通常用来在构造函数中接受初始化字段的值，参见程序清单4-4中的Box类。

程序清单4-4 Box类

```
namespace Project04a
{
    public class Box
    {
        int Length;
        int Width;
        int Height;
        public Box(int length, int width, int height)
        {
            this.Length = length;
            this.Width = width;
        }
    }
}
```

```
        this.Height = height;
    }
}
```

Box类有3个字段，**Length**、**Width**和**Height**。构造函数接受用来初始化这些字段的3个参数。用length、width和height作为参数名非常方便，因为名称可以反映出它们的用途。在构造函数中，length是指length参数，而不是length字段。this.length引用的是类级的length字段。

4.9 使用其他类

在我们编写的类中经常会用到其他的类。默认情况下，使用同一个命名空间中的类作为当前类是允许的。

但是，要使用另一个命名空间的类，我们就必须先要用关键字using来导入命名空间。例如，要在代码中使用System命名空间的成员，例如，System.Console类，我们必须使用如下的using语句。

```
namespace Project04a
{
    using System;
    public class Demo
    {
        public void Test()
        {
            // 因为导入了System, 所以我们可以使用System.Console
            Console.WriteLine("Testing ...");
        }
    }
}
```

注意，using语句必须放在命名空间内，但要放在类声明前。关键字using可以在一个命名空间中出现多次。

```
namespace Project04a
{
    using System;
    using System.IO;
    ...
}
```

使用其他命名空间中的类而不用导入它们的方法只有一种，那就是在代码中使用类的完全限定名。例如，下面语句没有导入**System**，就可以使用**System.Console**。

```
System.Console.Beep();
```

如果要在不同命名空间使用具有相同名称的类，那么我们在声明类时必须要用到完全限定类名。例如，程序清单4-5中，**MyCompany.Person**类用到**Project1.Chair**和**Project2.Chair**这两个类。没有完全限定名称的话，要使用哪个**Chair**类就不明确。

程序清单4-5 使用完全限定类名

```
namespace Project1
{
    public class Chair
    {
    }
}

namespace Project2
{
    public class Chair
    {
    }
}

namespace MyCompany
{
    class Person
    {
        static void Main()
        {
            Project1.Chair p1Chair = new Project1.Chair();
            Project2.Chair p2Chair = new Project2.Chair();
        }
    }
}
```

一个类用到另一个类，就是说这个类“依赖”另一个类。UML图可以表示这种依赖性，如图4-8所示。

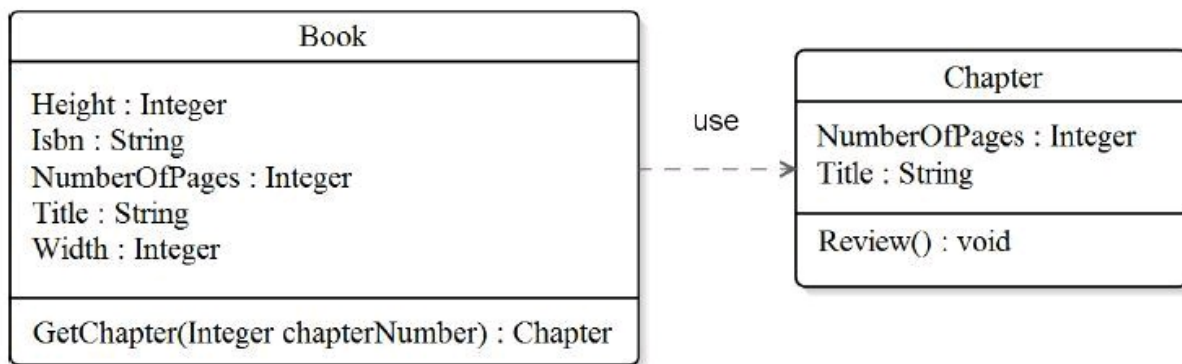


图4-8 UML类图中的依赖性

依赖关系用虚线箭头表示。在图4-8中，Book类依赖Chapter，因为GetChapter方法会返回Chapter对象。

4.10 静态成员

我们已经学习了如何访问对象的public的字段或方法，可以在对象引用后加上一个句点，如下所示。

```
// 创建Book的一个实例
Book book = new Book();
// 访问Review方法
book.Review();
```

这意味着，在访问对象的成员前，我们首先必须要创建一个对象。但是在前边的章节中，示例使用System.Console.WriteLine把值输出到控制台。我们注意到，在调用WriteLine方法时，我们不必先创建Console对象。怎样才能不必做诸如下面这些事呢？

```
Console console = new Console();
console.WriteLine("blah");
```

相反，我们在类名后用一个句点。

```
Console.WriteLine("blah");
```

C#（以及很多OOP语言）支持静态成员的概念，静态成员不需要先

实例化类就可以调用的类成员。在System.Console中，WriteLine方法是静态的，这就是为什么可以不必先实例化System.Console就可以使用该方法。

静态成员没有和类的实例绑定起来，相反，我们可以在没有实例的情况下调用它们。实际上，作为类的入口的Main方法就是静态的，因为必须要在创建任何对象之前就调用它。

要创建一个静态方法，我们要在字段或方法声明前使用关键字static。如果有访问修饰符，关键字static可以放在修饰符前边，也可以放在修饰符后边。这两种写法都是正确的。

```
public static int NumberOfPages;  
static public int NumberOfPages;
```

但是，第一种形式更常用。

如程序清单4-6所示，这是带一个静态方法的MathUtil类。

程序清单4-6 MathUtil类

```
namespace Project04a  
{  
    class MathUtil  
    {  
        public static int Add(int a, int b)  
        {  
            return a + b;  
        }  
    }  
}
```

要使用Add方法，我们可以直接像下面这样调用它。

```
MathUtil.Add(a, b)
```

术语实例化方法或字段（instance methods/fields）用来指那些非静态的方法和字段。

在静态方法内部，我们不能调用实例方法或实例字段，因为它们只

有在创建对象之后才存在。但是，我们可以从一个静态方法访问另一个静态方法或字段。

初学者经常会遇到的一个容易混淆的问题，就是不能编译自己的类，这是因为在Main方法中调用了实例成员。程序清单4-7展示了这样的一个类。

程序清单4-7 静态方法调用非静态成员

```
namespace Project04a
{
    using System;
    public class StaticDemo
    {
        public int B = 8;
        static void Main()
        {
            Console.WriteLine(B);
        }
    }
}
```

粗体行的代码会导致编译错误，因为它试图在静态方法Main中访问非静态字段B。这里有两种解决方案。

- 使B成为静态的。
- 创建这个类的一个实例，然后用该对象引用访问B。

选择哪种解决方案要视情况而定。这往往需要多年的OOP经验才能做出好的决策。

注意

我们只能在类级别声明静态变量。即使方法是静态的，也不能声明局部静态变量。

那么，静态引用变量又是什么情况呢？我们可以声明静态引用变量。这个变量将包含一个地址，但引用的对象存储在堆中，例子如下所示。

```
static Book book = new Book();
```

静态引用变量提供了一种很好的方法来曝光相同的对象，这个对象需要在其他不同对象间共享。

注意

在UML类图中，静态成员有下划线。如图4-9所示，这是带有静态方法Add的MathUtil类。

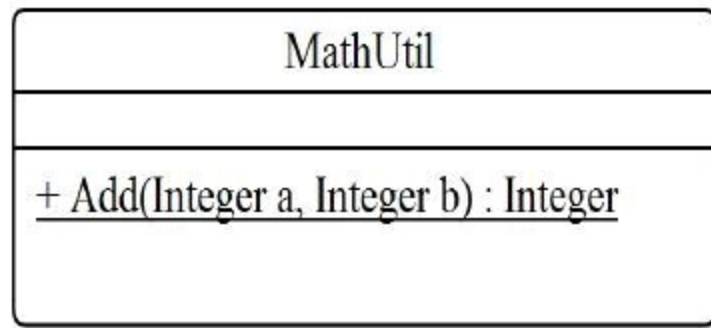


图4-9 UML类图中的静态方法

4.11 变量作用字段

我们已经看到可以在不同的地方声明变量。

- 在类的主体中作为类字段，这里声明的变量指的是类级变量。
- 作为方法或构造函数的参数。
- 在方法的主体中或构造函数的主体中。
- 在语句块中，例如在while或for语句块中。

现在我们来学习变量的作用字段。

变量作用字段指的是变量的可访问性。这条规则是，在一个语句块中定义的变量只能在语句块中访问。变量作用字段就是其定义所在的语句块。例如，我们来看下面的for语句块。

```
for (int x = 0; x < 5; x++)
{
    System.Console.WriteLine(x);
}
```

```
}
```

变量`x`在`for`语句块中定义。结果，`x`只能在`for`语句块中访问。对于任何其他地方，`x`都是不能访问或不可见。

第2条规则是，内部语句块能够访问在外部语句块中声明的变量，参见下面的代码。

```
for (int x = 0; x < 5; x++)
{
    for (int y = 0; y < 3; y++)
    {
        System.Console.WriteLine(x);
        System.Console.WriteLine(y);
    }
}
```

上述代码是合法的，因为内部的`for`语句块能够访问在外部`for`语句块中声明的`x`。

另一条规则，在方法主体中可以访问作为方法参数声明的变量。此外，类级变量可以在类中的任何地方访问。

如果一个方法声明了一个局部变量，而且还有一个相同名称的类级变量，前者将“屏蔽”后者。要在这个方法主体中访问类级变量，我们要用到关键字`this`。

4.12 方法重载

方法名称很重要，它能反映出方法要做什么。在很多情况下，我们可能想为多个方法起同样的名称，因为它们具有类似的功能。例如，方法`PrintString`可能接受一个`String`参数并且输出字符串。但是，同一个类可能还提供了一个方法，它输出一个`String`的一部分并且接受两个参数，分别是要输出的`String`以及开始打印的字符所在位置。我们想把后一个方法也叫作`PrintString`，因为它确实打印了一条`String`，但是这会和前一个`PrintString`方法重名。

幸好，C#中允许多个方法具有相同的名称，只要每个方法所接受的

参数类型的组合不同就可以了。也就是说，在我们的例子中，同一个类中有两个这样的方法是合法的。

```
public void PrintString(String string)
public void PrintString(String string, int offset)
```

这个特性叫做方法重载。

方法重载不会考虑方法的返回值。因此，同一个类中不允许存在如下的两个方法。

```
public int CountRows(int number);
public string countRows(int number);
```

这是因为调用方法不一定要将其返回值赋予一个变量。在这种情况下，上面的`ountRows`方法会让编译器混淆，因为它不知道如下语句想要调用哪个方法。

```
System.Console.Write(countRows(3));.
```

下面的方法是一种复杂的情况，两个方法的签名非常相似。

```
public int PrintNumber(int i)
{
    return i*2;
}
public long PrintNumber(long l)
{
    return l*3;
}
```

在同一个类中有两个这样的方法是合法的。但是，我们可能想知道**`PrintNumber(3)`**会调用哪个方法。

我们在第2章中讲过，数值型的直接量会转换成`int`，除非有`L`或`l`后缀。因此，`printNumber(3)`会调用以下方法。

```
public int PrintNumber(int i)
```

要调用另一个方法，我们需要传递long。

```
printNumber(3L);
```

注意

静态方法也可以重载。

4.13 小结

在本章中，我们学习了如何在C#中用关键词new创建对象，如何在内存中保存对象，如何把类组织到命名空间中，如何用访问控制实现封装以及C#如何管理未用过的对象。另外，我们还学习了方法重载和静态类成员。

第5章 核心类

在讨论其他面向对象编程特性前，我们先来看看在C#中经常用到的几个重要的类。这些类包含在.NET Framework的类库中。掌握了它们有助于我们了解下面OOP课程的相关案例。

所有类中最著名的，毫无疑问是System.Object。但是不先了解继承的话，我们就很难介绍这个类。继承会在第6章介绍。因此，本章只是对System.Object作一个简单的介绍。现在，我们主要介绍那些在程序中要用到的类。我们从System.String和System.Text.StringBuffer开始，然后我们会讨论数组和System.Console类。这些类以及.NET Framework类库中其他可用类型的完整文档介绍，请参见如下网址。

```
http://msdn.microsoft.com/en-us/library/gg145045
```

5.1 System.Object

System.Object类表示了一个C#对象。实际上，所有的类都是从这个类直接或间接派生的。因为我们还没有学习继承（在第6章学习），派生这个词可能会把你搞晕。所以我们先简单地介绍这个类的一些方法，然后在第6章再重新介绍。

下面是Object类中的一些方法。

```
public boolean Equals(Object obj)
```

该方法比较这个对象和传入的对象。类必须要实现这个方法，以提供一种方法来比较它的实例内容。

```
public Type GetType()
```

该方法返回这个对象的System.Type。

```
public virtual int GetHashCode()
```


该方法返回这个对象的哈希码。

```
public virtual string ToString()
```

该方法返回这个对象的字符串描述。

这里不用考虑一些方法签名中使用的关键字`virtual`。

5.2 System.String

我们从未没见过一个正规的C#程序不使用`System.String`类。它是最常用的类之一，而且也绝对是最重要的类之一。

`String`对象表示一个字符串，也就是一段文本。你也可以把`String`当作一串Unicode字符。一个`String`对象可以包含任意数量的字符。拥有零个字符的`String`叫作空`String`。`String`对象是常量。一旦创建了`String`对象，它的值就不能再改变。因此，我们也说`String`实例是不可变的。

我们可以用关键字`new`和`String`类的一个构造函数来创建一个`String`对象，但这种用法不常见。更多时候，我们将一个字符串直接量赋值给一个`String`变量。

```
System.String s = ".NET is cool";
```

这就生成了一个包含“.NET is cool”的`String`对象并把它的引用赋予`s`。

创建`String`的一种更简单的方法是使用`string`类型，它是`String`类的别名。上述语句可以更简单地写成以下格式。

```
string s = ".NET is cool";
```

字符串可以包含转义字符（我们在第2章学习过），如下所示。

```
string fileName = "C:\\win.txt";
```

`fileName`将包含`C:\win.txt`的值，因为“\\”表示反斜杠。下面的语句会

报错，因为编译器不能识别\w。

```
string wrongFileName = "C:\win.txt";
```

但是，我们可以通过加上前缀“@”，强制编译器解析字符串直接量。例如，下面的语句是正确的，因为“\”不是转义符而是被看作一个普通字符。

```
String myFileName = @"C:\win.txt";
```

5.2.1 字符串连接

我们可以使用运算符“+”把两个字符串连接起来，或者使用String类的Concat方法把两个字符串连接起来。

下面是使用运算符“+”的示例。下面的语句执行后，greeting将包含“Aloha”。

```
string al = "Al";  
string oha = "oha";  
string greeting = al + oha;
```

另一种选择是，我们可以使用String类的Concat方法来连接两个字符串。

```
string al = "Al";  
string oha = "oha";  
string greeting = String.Concat(al, oha);
```

注意，既然Concat是一个静态方法，要调用它，我们可以不必先创建String对象。

5.2.2 比较两个字符串

字符串比较是C#编程中最有用的操作之一。我们来看下面的代码。

```
string a = ".NET is cool";  
string b = a;
```

这里(a == b)的结果是true，因为a和b都引用了相同的实例。

5.2.3 字符串直接量

因为总是要和String对象打交道，所以理解字符串直接量的使用规则很重要。

首先，字符串直接量开始和结束都用双引号（"）。

其次，在结束双引号前换行会导致编译错误。如下的语句就会导致一个编译错误。

```
string s2 = "This is an important  
point to note";
```

我们可以通过加号把两个字符串直接量连接成一个长字符串。

```
string s1 = "Strings " + "are important";  
string s2 = "This is an important " +  
"point to note";
```

我们也可以把字符串和基本类型或另一个对象进行连接。例如，下面这行代码连接了一个字符串和一个整数。

```
string s3 = "String number " + 3;
```

如果一个对象和一个字符串连接，会调用这个对象的ToString方法并把结果进行连接。

5.2.4 转义特定字符

有时我们需要在字符串中使用诸如回车（CR）或换行（LF）这些特殊字符。有时我们可能想在字符串中用到双引号。对于CR和LF，我们不可能输入这些字符，因为按下Enter键就换行了。包含特殊字符的一种方法就是转义它们，也就是说，用字符替代它们。

下面是一些转义字符。

<code>\u</code>	<code>/* 一个 Unicode 字符</code>
<code>\b</code>	<code>/* \u0008: 退格键 BS */</code>
<code>\t</code>	<code>/* \u0009: 水平制表 HT */</code>
<code>\n</code>	<code>/* \u000a: 换行 */</code>
<code>\f</code>	<code>/* \u000c: 换页 */</code>
<code>\r</code>	<code>/* \u000d: 回车 CR */</code>
<code>\"</code>	<code>/* \u0022: 双引号 " */</code>
<code>\'</code>	<code>/* \u0027: 单引号 ' */</code>
<code>\\</code>	<code>/* \u005c: 反斜线 \ */</code>

例如，下面的代码在字符串结尾处加上了Unicode字符0122。

```
string s = "Please type this character \u0122";
```

要获取内容是John "The Great" Monroe的一个字符串，我们转义了双引号。

```
string s = "John \"The Great\" Monroe";
```

5.2.5 String类的属性

String类提供了Length和Chars两个属性。Length属性提供了当前String对象中的字符的数目。例如，下面的代码行把5赋予变量stringLength，因为“Hello”的字符的数目是5。

```
int stringLength = "Hello".Length;
```

Chars属性返回指定索引位置的Char对象。属性Chars的语法如下。

```
public char this[int index] { get; }
```

这里，index表示一个从0开始的位置。因此，要得到字符串中第1个位置，传递的参数是0。例如，在下面代码中，greeting[0]返回“W”。

```
String greeting = "Welcome";  
char firstChar = greeting[0];
```

因此，Chars属性只接受0到字符数减1之间的数。如果我们传递的数超出这个范围，会导致**IndexOutOfRangeException**异常。

例如，下面代码使用Length和Chars属性来输出字符串中的单个字符。

```
string greeting2 = "Hello";
for (int i = 0; i < greeting2.Length; i++)
{
    Console.WriteLine(greeting2[i]);
}
```

5.2.6 String类的方法

String类提供了操作String的值的方法。但是因为String对象是不可改变的，操作的结果总是一个新的String对象。

下面是一些更有用的方法。

```
public static string Concat(string s1, string s2)
```

该方法连接两个字符串。例如，String.Concat("Hello", "World")返回"HelloWorld"。

```
public bool Contains (string value)
```

该方法判断当前字符串是否包含传入的值。例如，"Credit card".Contains("card")返回true。

```
public bool EndsWith(string suffix)
```

该方法判断当前字符串是否以特定的后缀结尾。

```
public int IndexOf(String substring)
```

该方法返回指定子字符串第一次出现的索引，如果没有发现匹配，返回-1。例如，下面的表达式返回6。

```
"C# is cool".IndexOf("cool")  
public int IndexOf(String substring, int fromIndex)
```

该方法返回指定的子字符串第一次出现的索引位置，搜索从指定的索引位置开始，如果没有发现匹配，返回-1。

```
public int LastIndexOf(String substring)
```

该方法返回指定的子字符串最后一次出现的索引位置，如果没有发现匹配，返回-1。

```
public int LastIndexOf(String substring, int fromIndex)
```

该方法返回指定的子字符串最后一次出现的索引位置，搜索从指定的索引位置开始，如果没有发现匹配，返回-1。例如，下面表达式返回7，这是基于0的索引中，最后一次出现“c”的位置。

```
"credit card".LastIndexOf("c")  
public string Substring(int beginIndex)
```

该方法返回一个当前字符串的一个子字符串，从指定的索引位置开始。例如，**"C# is cool".Substring(6)** 返回“cool”。

```
public string Substring(int beginIndex, int length)
```

该方法返回一个当前字符串的一个子字符串，它从beginIndex开始，其长度应该是指定的length。例如，下面的代码返回“is”。

```
"C# is cool".Substring(3, 2)  
public string Replace(char oldChar, char newChar)
```

该方法在当前字符串中用newChar替换掉oldChar的每一次出现，然后返回新的字符串。**"dingdong".Replace('d', 'k')** 返回“kingkong”。

```
public string Replace(string oldValue, string newValue)
```

该方法在当前字符串中用newValue替换掉oldValue的每一次出现，然后返回新的字符串。例如，**"Spring".Replace("Spr", "st")** 返

回“sting”。

```
public static bool IsNullOrEmpty(string value)
```

该方法中如果指定字符串是null或空，返回true。空字符串不包含字符。

```
public string[] Split(char[] separator)
```

该方法把当前字符串按照与指定的字符的匹配而拆分成数组。例如，"big city mayors".Split(" ")返回包含三个字符串的一个数组。第一个数组元素是“big”，第2个是“city”，第3个是“mayors”。

```
public bool StartsWith(String prefix)
```

该方法判断当前字符串是否以指定的前缀开始。

```
public char[] ToCharArray()
```

该方法将当前字符串作为一个字符数组返回。

```
public string ToLower()
```

该方法把当前字符串中所有字符转换成小写。例如"Coffee shop hero".ToLower() 返回“coffee shop hero”。

```
public String ToUpper()
```

该方法把当前字符串中所有字符转换成大写。例如" temporary".ToUpper ()返回“TEMP- ORARY”。

```
public string Trim()
```

该方法截去开头和结尾的空格并返回一个新的字符串。例如“Venus”.Trim() 返回“Venus”。

5.3 System.Text.StringBuilder

String对象是不可改变的，如果我们需要增加或插入字符，String对象不是一个好的选择。这是因为String对象上的字符串操作总是会产生新的String对象，由此会产生代价。对于现代计算机，这种额外处理的时间“花销”几乎可以忽略不见。但是，避免这种操作是一种好的编程做法。要进行增加或插入，我们最好使用System.Text.StringBuilder类。

要使用StringBuilder，首先我们要创建一个实例，指定容量，也就是它所能包含的字符的数目。当我们为StringBuilder增加字符或字符串时，只要有足够的空间容纳所增加的内容，系统就不会创建一个新的实例。如果超出了容量，系统会自动增加容量，但是会有开销。因此，我们应该确保所创建的StringBuilder具有足够大的容量，但是保留的字符空间也不宜太大，即使这些空间还没有用到，也会占用内存空间。一旦完成了对字符的操作，我们就可以把StringBuilder转换成一个字符串。

我们来看看如何创建一个StringBuilder并使用它的方法。

5.3.1 StringBuilder类的构造函数

StringBuilder类有六个构造函数。最简单的一个是不接受参数的构造函数，它可以创建能容纳16个字符的一个StringBuilder。

```
public StringBuilder()
```

第二个构造函数允许指定容量。

```
public StringBuilder(int capacity)
```

第三个构造函数允许指定最大容量，试图增加的字符超过最大容量时，将会抛出异常。

```
public StringBuilder(int capacity, int maximumCapacity)
```

如果愿意的话，我们可以在StringBuilder中预先存放一个字符串，使用第四个构造函数就可以实现。

```
public StringBuilder(string value)
```

我们也可以用第五个构造函数指定初始字符串和容量。

```
public StringBuilder(string value, int capacity)
```

最后一个StringBuilder的构造函数，它允许用子字符串作为初始值。

```
public StringBuilder(string value, int startIndex, int length,  
    int capacity)
```

例如，下面的语句可以创建一个初始值为"World"的StringBuilder。

```
string s = "Hello World";  
StringBuilder builder = new StringBuilder(s, 6, 5, 20);
```

所创建的StringBuilder如果没有指定最大容量，它将会有一个非常大的容量（2GB）。

5.3.2 StringBuilder类的属性

StringBuilder类提供了四个属性：Capacity、Chars、Length和MaximumCapacity。

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

该属性表示StringBuilder的容量。

```
public char this[int index] {get; set; }
```

该属性在指定索引位置获取或设置字符。例如，下面代码创建了一个StringBuilder并且修改了它的第一个和第五个字符。

```
StringBuilder builder4 = new StringBuilder("Kinkong");  
builder4[0] = 'P';  
builder4[4] = 'p';
```

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

该属性获取或设置StringBuilder的长度。

```
public int MaxCapacity { get; }
```

该属性获取StringBuilder的最大容量值。

5.3.3 StringBuilder类的方法

毫无疑问，StringBuilder中最重要的方法就是ToString方法，它会将StringBuilder内容返回为一个字符串。

```
public override string ToString()
```

如果没有这个方法，StringBuilder类几乎就没什么用了，因为有无数的方法接受string参数，但是很少有方法接受StringBuilder。

在StringBuilder的实例中，StringBuilder还定义了用来增加、插入和删除字符的方法。Append方法有多个重载，它们能够增加字符串、字符、数字或其他数据类型。如果我们增加一个整数，在增加前会先把整数转换成字符。

和Append类似，Insert方法也有多个重载，可以允许向一个StringBuilder中插入不同的数据类型。Append和Insert的不同之处在于，Append总是在StringBuilder的末尾增加一个字符，而Insert可以在任何位置增加字符。

如下是Append和Insert的一些重载。

```
public StringBuilder Append(string value)
public StringBuilder Append(char value)
public StringBuilder Append(Object value)
public StringBuilder Append(int value)
public StringBuilder Insert(int index, string value)
public StringBuilder Insert(int index, char value)
public StringBuilder Insert(int index, Object value)
public StringBuilder Insert(int index, int value)
```

除了Append和Insert，StringBuilder还提供了Remove方法来删除字符。

```
public StringBuilder Remove(int startIndex, int Length)
```

有趣的是，Append、Insert和Remove都返回同样的StringBuilder，所以这些方法可以像下面这样串联起来使用。

```
StringBuilder sb = new StringBuilder("Hi Hello");  
sb.Append("World").Insert(8, ' ').Remove(0, 3);  
Console.WriteLine(sb.ToString()); // 输出"Hello World"
```

5.4 数组

在C#中，我们可以用数组来分组基本类型或相同类型的对象。数组中的实体叫作数组的元素或成员。数组是从System.Array派生而来的类的实例。因此，数组继承了System.Array类的所有字段、属性和方法。例如，我们可以调用数组的Length来得到其中的元素的数目。Length字段是在System.Array中定义的一个字段。

一个数组中的所有元素都具有相同的类型，这叫作数组的元素类型。数组不可改变大小，带有0个元素的数组叫作空数组。

数组是一个对象。因此，我们可以把引用数组的变量当作其他引用变量一样对待。例如，我们可以判断它是否为null。

```
String[] names;  
if (names == null)    // 判断结果为true
```

注意

一个数组也可以包含其他的数组，从而创建一个数组的数组。

我们可以用如下的语法来声明数组。

```
type[] arrayName;
```

例如，下面的语句声明了一个名为numbers的long类型的数组。

```
long[] numbers;
```

声明一个数组并不会创建一个数组或为它的元素分配空间，编译器只是创建一个对象引用。创建数组的一种方法是使用关键字new。我们必须指定所要创建的数组的大小。

```
new type[size]
```

例如，下面的代码创建了包含四个int元素的一个数组。

```
new int[4]
```

我们也可以在同一行中声明和创建一个数组。

```
int[] ints = new int[4];
```

要引用一个数组中的元素，就要在变量名称后使用一个索引。数组是从零开始的，也就意味着数组中的第一个元素的索引值是0。要获取数组的第一个元素，我们可以在方括号中用它的索引值。例如，下面的代码创建了包含四个String对象的一个数组并初始化了它的第一个成员。

```
string[] names = new string[4];  
names[0] = "Hello World"; //给name的第一个成员赋值
```

我们也可以不用关键字new来创建和初始化数组。例如，下面的代码创建了包含三个String对象的一个数组。

```
String[] names = { "John", "Mary", "Paul" };
```

下面的代码创建了包含四个int的一个数组并把这个数组赋予了变量matrix。

```
int[] matrix = { 1, 2, 3, 10 };
```

我们在把数组传递给一个方法时要小心，下面的代码就是不合法的，虽然average方法接受了一个int类型的数组。

```
int avg = average( { 1, 2, 3, 10 } ); // 不合法
```

相反，我们必须单独实例化这个数组。

```
int[] numbers = { 1, 2, 3, 10 };  
int avg = average(numbers);
```

或者如下所示。

```
int avg = average(new int[] { 1, 2, 3, 10 });
```

引用超出范围的元素将会导致运行时的错误。例如，下面的代码会导致错误，因为它试图访问数组中的第五个元素，而该数组只包含两个元素。

```
int[] numbers = { 1, 3 };  
int x = numbers[4];
```

注意

当创建一个数组时，它的元素要么是null（如果元素类型是对象类型），要么是该元素类型的默认值（如果数组包含的是基本类型）。例如，int的数组默认包含0。

5.4.1 遍历数组

通常遍历数组可以有两种方法，分别是使用foreach或for。

前者的格式更简短一些。

```
foreach (elementType element in arrayName)  
{  
    // 在这里访问元素  
}
```

例如，下面的代码打印输出了employees中的元素。

```
string[] employees = { "John", "Paul", "George", "Ringo" };
foreach (string employee in employees)
{
    Console.WriteLine("Employee:" + employee); // 打印输出employee
}
```

结果如下。

```
Employee:John
Employee:Paul
Employee:George
Employee:Ringo
```

第二个方法是使用for循环并通过索引来访问每个元素。

```
string[] employees = { "John", "Paul", "George", "Ringo" };
for (int i = 0; i < employees.Length; i++)
{
    Console.WriteLine("Employee(" + (i + 1) + "): " + employees[i]);
}
```

结果如下。

```
Employee(1):John
Employee(2):Paul
Employee(3):George
Employee(4):Ringo
```

我们可以看到，即使foreach更简单，但如果索引对你很重要的话，还是要使用for循环。

5.4.2 改变数组的大小

一旦创建了一个数组，它的大小就不能改变了。如果想要改变其大小，我们必须创建一个新数组并使用旧的数组的值填充它。例如，下面的代码把包含3个int元素的数组numbers的大小增加到4。

```
int[] numbers = { 1, 2, 3 };
int[] temp = new int[4];
for (int j = 0; j < numbers.Length; j++)
```

```
{
    temp[j] = numbers[j];
}
numbers2 = temp;
```

另一种方法是使用System.Array的静态方法Resize。

```
Array.Resize(ref arrayName, newSize)
```

例如，下面的代码把numbers的大小改为5。

```
int[] numbers = { 2, 3, 4 };
Array.Resize(ref numbers, 5);
Console.WriteLine("Length of numbers:" + numbers.Length);
```

5.4.3 为Main传递一个字符串数组

我们可以为Main方法传递一个字符串数组，从而向程序传递值。这个示例中，下面是Main方法的签名。

```
public static void Main(string[] args)
```

程序清单5-1给出了一个类，它遍历了Main方法的String数组参数。

程序清单5-1 访问Main方法的参数

```
public class MainMethodTest
{
    public static void main(String[] args) {
        foreach (string arg in args) {
            Console.WriteLine(arg);
        }
    }
}
```

5.5 System.Console

System.Console类公开了和操作控制台相关的有用的静态字段和静

态方法。下面是System.Console中一些较为重要的方法。

```
public static void Beep()
```

该方法播放蜂鸣声。

```
public static void Clear()
```

该方法清除控制台缓存和控制台窗口。

```
public static ConsoleKeyInfo ReadKey()
```

该方法得到用户的下一个按键信息并在控制台窗口显示它。

```
public static string ReadLine()
```

该方法返回输入流的下一行字符。

```
public static void Write(string s)
```

该方法把字符串参数写入到控制台。有其他重载的Write方法，从而可以向Write方法传递任何类型。

```
public static void WriteLine(string s)
```

该方法把字符串参数以及行结束符写入到控制台。

5.6 小结

在本章中，我们学习了许多重要的类，如System.Object、System.String、System.Text. StringBuilder、System.Array和System.Console。这些是C#中最常用的类。我们将在第6章中学习更多的类。

第6章 继承

继承是一种非常重要的面向对象编程的特性。它使得代码在任何OOP语言中都是可以扩展的。扩展一个类也叫作继承或子类化。在C#中，默认所有的类都是可扩展的，但是我们可以用关键字`sealed`来阻止一个类的子类化。本章介绍C#中的继承。

6.1 继承概述

我们通过创建一个新的类来扩展一个类，后者和前者就会形成父类-子类关系。初始类是父类（parent class）或基类（base class）或超类（superclass）。新的类是子类（child class或subclass），或父类的派生类（derived class）。在OOP中，类的扩展过程叫作继承。在子类中，我们可以增加新的方法、新的字段和新的属性，也可以覆盖已有的方法来改变它们的行为。

UML类图可以表示类和子类之间的父类-子类关系，如图6-1所示。

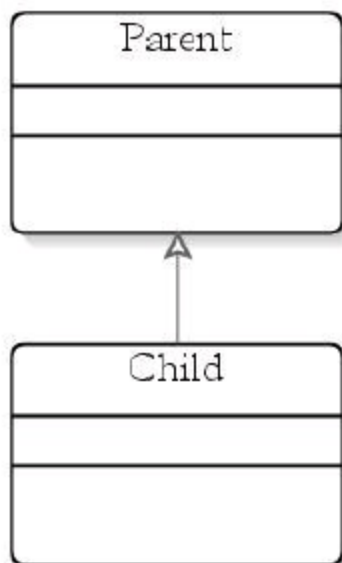


图6-1 UML图中的父类和子类

注意，带箭头的直线用来表示泛化，例如父类-子类关系。

子类也可以被扩展，除非通过用关键字sealed特别声明它以限制它的扩展。本章后边的小节将介绍密封（Sealed）类。

继承的好处显而易见。继承使我们能够增加一些初始类中所没有的功能。它也使我们有机会改变已有的类的行为，以使其更好地满足我们的需求。

6.1.1 扩展一个类

在类的声明中，使用冒号来扩展一个类，冒号放在类名称的后边并且在父类的名称前边。程序清单6-1展示了一个叫做Parent的类，程序清单6-2展示了从Parent扩展而来的一个叫做Child的类。

程序清单6-1 Parent类

```
public class Parent
{
}

```

程序清单6-2 Child类

```
public class Child : Parent
{
}

```

扩展一个类就是这么简单。

注意

C#中所有的类都是从System.Object扩展而来。

Object是.NET中的最高级的超类。程序清单6-1中的Parent默认是Object的子类。

注意

在C#中，一个类只能扩展一个其他的类。这和C++不同，C++允许多继承。但是多

继承的概念可以用接口来实现，我们会在第9章介绍。

6.1.2 is-a关系

当我们通过继承来创建一个新类的时候，就形成一种特殊的关系。子类和父类有一种“is-a”的关系。

例如，Animal是一个表示动物的类。动物类型有许多种，包括鸟、鱼和狗，所以我们可以创建Animal类的子类来表示动物类型。如图6-2所示，Animal类有3个子类，Bird、Fish和Dog。

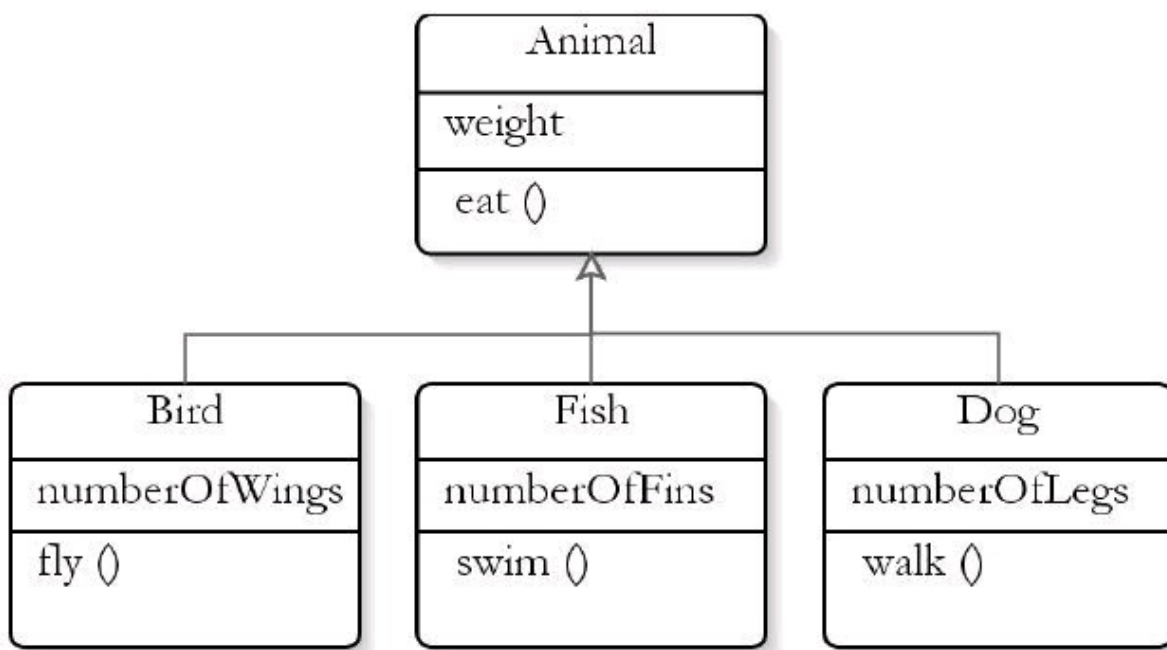


图6-2 继承的一个示例

显然，子类和超类Animal之间是is-a关系。鸟“是一种”动物，狗“是一种”动物，鱼也“是一种”动物。子类是超类的一种特定类型。例如，鸟是动物中的一种特定类型。然而，is-a关系反之则不成立。一种动物不一定就是一只鸟或者一条狗。

程序清单6-3 Animal及其子类

```
class Animal
```

```
{
    public float Weight;
    public void Eat()

    {
    }
}

class Bird : Animal {
    public int NumberOfWings = 2;
    public void Fly()
    {
    }
}

class Fish : Animal {
    public int NumberOfFins = 2;
    public void Swim()
    {
    }
}

class Dog : Animal {
    public int NumberOfLegs = 4;
    public void Walk()
    {
    }
}
```

在这个例子中，Animal类定义了一个Weight字段，它应用于所有的动物中。它还声明了一个Eat方法，因为动物都要吃东西。

Bird类是Animal的一种特殊类型，它继承了Eat方法和Weight字段。Bird还增加了一个NumberOfWings字段和一个Fly方法。这表明更为具体的Bird类从更为通用的Animal类扩展了功能和行为。

子类继承了超类所有public的方法和字段。例如，我们可以创建一个Dog对象并调用它的Eat方法。

```
Dog dog = new Dog();
dog.Eat();
```

Eat方法在Animal类中声明，Dog类直接继承它。

is-a关系导致的结果就是，把子类的一个实例赋予父类的一个引用变量合法。例如，下面的代码就是合法的，因为Bird是Animal的子类，一只鸟也就是一种动物。

```
Animal animal = new Bird();
```

但是，下面的代码是不合法的，因为无法证明一个动物就是一只狗。

```
Dog dog = new Animal();
```

6.2 可访问性

在子类中，我们可以访问它的超类的public和protected的成员（如方法和字段），但是不能访问超类中的private成员。如果子类和超类在同一个程序集中，我们也可以访问超类的internal成员。

我们来看程序清单6-4中的P类和C类。

程序清单6-4 展示可访问性

```
public class P
{
    public void PublicMethod()
    {
    }

    protected void ProtectedMethod()
    {
    }

    internal void InternalMethod()
    {
    }
}

class C : P
{
    public void TestMethods()
    {
        PublicMethod();
    }
}
```

```
        ProtectedMethod();  
        InternalMethod();  
    }  
}
```

P有3个方法，一个访问级别是public，一个访问级别是protected，还有一个访问级别是internal。C是P的子类。我们可以看到，在C类的TestMethods方法中，C可以访问它的父类的public和protected方法。另外，因为C和P都在一个程序集中，C也可以访问P的internal方法。

6.3 方法覆盖

扩展一个类时，我们可以改变父类中方法的行为。这叫作方法覆盖，当我们在子类中编写一个方法，而这个方法与父类中的方法有相同的签名，这时候就产生了方法覆盖。如果只有名称相同，但是参数列表不同，那叫作方法重载。方法重载已经在第4章介绍过。

我们可以覆盖一个方法来改变它的行为。覆盖一个方法，就是在子类中编写一个新的方法。我们可以覆盖超类的public和protected的方法。如果子类和超类在同一个程序集中，我们也可以覆盖internal访问级别的方法。

程序清单6-5通过Shape类和Oval类展示了方法覆盖的一个示例。

程序清单6-5 Shape类和Oval类

```
using System;  
class Shape  
{  
    public void WhatAmI()  
    {  
        Console.WriteLine("I am a shape");  
    }  
}  
  
class Oval : Shape  
{  
    new public void WhatAmI()  
    {
```

```
        Console.WriteLine("I am an oval");  
    }  
}
```

Oval类扩展自Shape类并覆盖了WhatAmI方法。在子类中使用关键字new来表示程序员覆盖了父类中的WhatAmI方法。没有关键字new，程序仍然可以编译，但是会给出警告。

我们可以使用如下语句来测试程序清单6-5中的方法覆盖。

```
Oval oval = new Oval();  
oval.WhatAmI(); // 打印"I am an oval"
```

正如我们所看到的，oval.WhatAmI调用的是Oval类中的方法。

现在猜猜看，如下的代码会输出什么？

```
Shape shape = new Oval();  
shape.WhatAmI();
```

在C#中，把一个子类的实例赋值给对象变量是合法的，因为毕竟椭圆形也是一种形状。但是，输出可能出乎我们的意料。结果如下所示。

```
I am a shape
```

在C#中，使用一个引用变量调用一个方法实际调用的是在该类型中的方法，而不管引用变量所引用的对象是什么类型。

6.4 调用基类的构造函数

子类和普通类一样，我们可以用关键字new来创建它的实例。如果我们没有在子类中显式地编写一个构造函数，编译器会隐式地加上一个不带参数（no-arg）的构造函数。

通过调用一个构造函数来实例化一个子类，构造函数首先要做的，是调用其直接父类的默认构造函数（即不带参数的那个）。在父类中，这个构造函数还会调用它的直属父类的构造函数。这个过程会自动重

复，直到到达System.Object的构造函数。换句话说，创建一个子对象时，其所有的父类也都会被实例化。

如程序清单6-6所示，程序在Employee类和Manager类中展示了这个过程。

程序清单6-6 调用一个基类的默认构造函数

```
using System;
class Employee
{
    public Employee()
    {
        Console.WriteLine("Employee()");
    }
    public Employee(string name)
    {
        Console.WriteLine("Employee() " + name);
    }
}

class Manager : Employee
{
    public Manager(string name)
    {
        Console.WriteLine("Manager() " + name);
    }
}
```

我们用下面的代码实例化Manager类。

```
Manager manager = new Manager("Jeff");
```

我们就可以在控制台看到以下内容。

```
Employee
Manager() Jeff
```

这证实Employee类的构造函数所做的第一件事，是调用Employee类的默认构造函数，即使这里有接受相同参数集的另一个构造函数。

在创建一个子类实例时，如果要调用基类中的不带参数的构造函数

数，我们可以使用关键字**base**。例如，我们把**Manager**中的构造函数改为如下的形式，来调用**Employee**的第2个构造函数。

```
public Manager(string name) : base(name)
{
    Console.WriteLine("Manager() " + name);
}
```

注意，子类从它自己的构造函数中调用父类的构造函数是有意义的，因为一个子类的实例必须伴随着其每一个父类的实例。通过这种方式，一个在子类中没有被覆盖的方法的调用将会传递到其父类，直到找到继承层级的最高处。

6.5 调用基类的隐藏成员

关键字**base**还有另一个作用。我们可以用它来调用基类中一个隐藏的成员或者一个被覆盖的方法。因为**base**表示直接父类的一个实例，**base.memberName**返回父类中的指定成员。我们可以访问基类中任何对子类可见的成员。例如，程序清单6-7展示了有父类-子类关系的两个类：**Tool**和**Pencil**。

程序清单6-7 用**Super**来访问隐藏成员

```
using System;
public class Tool
{
    public string Identify()
    {
        return "Generic tool";
    }
}

public class Pencil : Tool {
    new public string Identify()
    {
        return "Pencil";
    }

    public void Write()
    {
        Console.WriteLine(base.Identify());
    }
}
```

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

Pencil类覆盖了**Tool**类的**Identify**方法。初始化**Pencil**类并且调用它的**Write**方法，我们就会在控制台看到如下结果。

```
Generic tool  
Pencil
```

6.6 类型转换

我们可以把一个对象转换成另一种类型。规则是，只能把一个子类的实例转换成它的父类。把一个对象转换为其父类叫作向上转换（upcasting）。下面是一个例子，我们假定**Child**是**Parent**的一个子类。

```
Child child = new Child();  
Parent parent = child;
```

要把一个**Child**对象向上转换，我们需要做的就是把这个对象赋予**Parent**类型的一个引用变量。注意，**Parent**引用变量不能访问只在**Child**中可用的那些成员。

因为上面的程序中的**parent**引用了一个**Child**类型的对象，所以我们可以把它转换回**Child**。这叫作向下转换（downcasting），因为这是把类的一个对象转换成继承层级中的下级类。向下转换时我们需要在圆括号中写上子类型，示例如下。

```
Child child = new Child();  
Parent parent = child; // 给parent赋一个Child实例  
Child child2 = (Child) parent; // 向下转换
```

只有父类引用已经指向了子类的一个实例时，才允许向下转换到子类。下面的代码会导致一个编译错误。

```
Object parent = new Object();
```

```
Child child = (Child) parent; // 不合法的向下转换，编译错误
```

6.7 密封类

我们可以在类声明中用关键字`sealed`来密封一个类，以阻止它派生其他的类。`sealed`可以出现在访问修饰符之前或之后，例子如下所示。

```
public final class Pencil  
final public class Pen
```

第一种形式较为常用。

尽管封装一个类会使代码稍微快一点，但这点区别太微不足道了。从设计考虑而不是从速度考虑，才是不允许一个类被扩展的原因。例如，密封`System.String`类，是因为类的设计者不想让我们修改`String`类的行为。

6.8 关键字is

关键字`is`可以用来判断一个对象是否是一种特定类型。关键字`is`通常用在`if`语句中，它的语法如下。

```
if (objectReference is type)
```

语法中`objectReference`引用一个待判断的对象。例如，下面的`if`语句会返回`true`。

```
String s = "Hello";  
if (s is System.String)
```

但是，在`null`的引用变量上使用`is`会返回`false`。如下`if`语句会返回`false`。

```
String s = null;  
if (s is System.String)
```

由于子类 and 基类是“is a”关系，下面的if语句中，Child是Parent的子类，返回true。

```
Child child = new Child();  
if (child is Parent)      // 计算结果为true
```

6.9 小结

继承是面向对象编程中最基本的原理之一。继承使代码得以扩展。在C#中，所有类默认都是从System.Object类扩展而来。要扩展一个类，就要使用冒号。方法覆盖是与继承直接相关的另一个OOP特性。方法覆盖使我们能够改变父类中成员的行为。我们可以通过密封一个类来阻止它的子类化。

第7章 结构

在前言中，我们介绍了.Net类型包括类、结构、枚举、接口和委托。我们已经在第4章中学习过类，到现在你应该对它们很熟悉了。

第二种类型是结构（`structure`或简写为`struct`），它和类一样可以拥有像字段或方法这样的成员。但是，和类不同的是，结构是值类型，它不能继承或被继承。.NET Framework类库提供了许多结构。例如，`System`命名空间定义了诸如`Byte`、`Char`、`Int32`和`DateTime`等重要的结构，它们用来表示重要的数据类型。

本章会介绍什么是结构并且会举一些例子来说明。

7.1 结构概述

有时我们把结构称为轻量级的类，它和类一样拥有诸如字段和方法这样的成员，我们可以用`new`来创建一个结构的实例。但是，结构是值类型，而类是引用类型。值类型在栈中分配或者以内联方式分配，当超出范围时释放分配空间。另一方面，引用类型在堆中分配，它未使用的实例会进行垃圾回收。值类型创建成本更低一些，但如果有许多装箱和拆箱操作，性能却要比引用类型差很多。所有结构都是从`System.Value.Type`隐性派生而来。

和类不同，结构没有“孩子”。换句话讲，结构不支持继承。

7.2 .NET结构

.NET Framework中有许多结构。C#中所有的基本类型，如`int`和`char`，都是在`System`命名空间中所定义的结构体的别名。例如，我们每在C#代码中声明一个`int`，就会创建一个`System.Int32`实例。每次我们用到一个`char`，就会有一个`System.Char`的实例支持它。同样，我们可以像下面这段代码一样，调用基本类型所对应的结构的成员。

```
int a = 123;  
System.Console.WriteLine(a.GetType()); // 打印 System.Int32
```

结构和类之间的另一个不同之处，就是两个变量不能指向同一个结构实例。把一个结构赋值给一个新的变量，会创建这个结构的一个新的实例。在程序清单7-1中，我们使用表示复数的**System.Numerics.Complex**来证实这一论点。

注意，我们可以在**System.Numerics.dll**程序集中找到**System.Numerics**命名空间。要使用其成员，我们需要先在项目中增加对这个程序集的引用。关于如何在项目中增加一个程序集，请参见如下说明。

在项目中增加一个引用

当我们在Visual Studio或类似的IDE中创建一个项目时，默认会包含许多程序集。我们不需要做任何事就可以使用这些程序集中的类型。如果想要使用一种不在默认程序集中的类型，我们就需要在项目中通过如下步骤来引用这个程序集。

- (1) 在Solution Explorer中，选中想要添加引用的项目。
- (2) 在“Project”菜单中，选择“Add Reference”选项。
- (3) 在出现的“Add Reference”对话框中，选择表示了想引用的组件类型的标签页。
- (4) 在最上层面板中，选择引用的控件，然后点击“Select”按钮。

程序清单7-1 复制一个结构

```
using System;  
using System.Numerics;  
  
class Numerics  
{  
  
    public void Test()  
    {
```

```
Complex c1 = new Complex(2d, 3);  
// 创建C1的一个新的副本  
Complex c2 = c1;  
c2 *= c2; // 不影响c1  
Console.WriteLine(c1); // 打印 (2, 3)  
Console.WriteLine(c2); // 打印 (-5, 12)  
}  
}
```

如果运行Test(), 我们可以在控制台看到如下结果。

```
(2, 3)  
(-5, 12)
```

这证实了c1和c2表示两个不同的对象, 因为我们在把c1赋予c2时, 创建了Complex的一个新的副本。因此, 操作c2并不会影响到c1, 反之亦然。

7.3 编写一个结构

本节展示了如何编写一个定制的结构。我们使用关键字struct, 其后跟着结构名称, 来创建一个结构。程序清单7-2展示了一个名为Point的定制的结构代码。这个结构有两个字段(X和Y)和两个方法(Move和Print)。

程序清单7-2 一个定制的结构

```
struct Point  
{  
  
    public int X;  
    public int Y;  
  
    public void Move(int x, int y)  
    {  
        X += x;  
        Y += y;  
    }  
  
    public void Print()  
    {
```

```
        System.Console.WriteLine("(" + X + ", " + Y + ")");  
    }  
}
```

我们可以用下面的代码来测试Point结构。

```
Point point1 = new Point();  
point1.Print();// 打印(0, 0)  
point1.X = 10;  
point1.Y = 20;  
point1.Move(4, 5);  
point1.Print(); // 打印(14, 25)
```

7.4 可为空的类型

引用类型和值类型的另一个重要的不同之处在于，引用类型可以为空（null），而值类型却不可以。因此，在C#中声明一个类变量为null是合法的。

```
string name = null;
```

但是，把null赋值给一个结构变量则会导致编译错误。例如，**System.DateTime**是一个结构，下面的代码不能够编译，因为给一个结构赋null值是不合法的。

```
System.DateTime today = null;
```

不能给一个值类型赋值null，这常常是一件令人头痛的事，特别是要使用一个关系数据库时而数据库中一张表的列可能是空的，这就意味着它没有包含数据。从数据库中获取一条数据并把它赋予一个**System.DateTime**，如果数据库中的这条数据是空值，那么可能会导致问题。

为了规避这个问题，.NET Framework提供了**System.Nullable**结构来使任意结构可以为空的。这种方案的精彩之处在于，我们不必直接使用**System.Nullable**。要让一个结构为nullable的，我们只需要在类型名称后加上一个问号即可。


```
int? x = null; // x 是 nullable
```

7.5 小结

在本章中，我们学习了结构以及它和类之间的区别。我们还学习了值类型和引用类型之间的不同，并且编写了一个定制化的结构。

第8章 错误处理

错误处理在任何编程语言中都是一个重要特性。好的错误处理机制会让编程人员更容易编写出健壮的应用程序，以防止潜在bug。有些语言中，程序员被迫要用多条if语句来检测所有可能会导致错误的条件。这样可能会使得代码过度复杂。而在一些大的程序中，这种做法很容易就导致代码混乱不堪。

C#有很好的错误处理的方法，它是通过try语句来实现的。采用这种策略，我们把可能出错的那部分代码隔离到一个语句块中。当错误出现时，这个错误会在本地被捕获并解决。本章将会介绍C#的错误处理。

8.1 捕获异常

我们可以用try语句把可能导致运行时错误的代码隔离，try语句通常与catch语句和finally语句一起使用。这样的隔离通常会出现在方法体内。如果遇到一个错误，公共语言运行时（CLR）会终止try语句块的处理并且跳转到catch语句块。我们可以很从容地处理这个错误或者抛出一个System.Exception对象来提醒用户。另一种方案是把异常或一个新的Exception对象抛回给调用这个方法的代码，然后由用户端决定如何处理这个错误。如果有一个抛出的异常没有捕获到，那么应用程序将会突然终止。

以下是try语句的语法。

```
try
{
    [可能抛出异常的代码]
}
[catch (ExceptionType-1 [e])
{
    [当抛出ExceptionType-1时要执行的代码]
}]
[catch (ExceptionType-2 [e])
{
    [当抛出ExceptionType-2时要执行的代码]
```

```
}}
...
[catch (ExceptionType-n [e])
{
    [当抛出ExceptionType-n时要执行的代码]
}]
[finally
{
    [无论是否抛出异常都会运行的代码]
}]
```

错误处理的步骤可以总结如下。

(1) 把那些可能会导致错误的代码隔离到try语句块中。

(2) 对于每个catch语句块，编写代码，如果try语句块中出现特定类型的异常，那么执行这些代码。

(3) 在finally语句块中编写代码，无论是否有错误产生，都将执行这些代码。

请注意，catch和finally语句块是可选的，但是二者至少选其一。因此，我们可以让一个try语句块带有一个或多个catch语句块，或者让try语句块带有finally语句块，或者让try语句块带有catch和finally语句块。

前面的语法显示我们可以有多个catch语句块。这是因为有些代码可能会抛出不同类型的异常。当try语句块抛出一个异常时，控制器会把它传递给第一个catch语句块。如果抛出的异常类型和第一个catch语句块的异常类型相匹配，或者是第一个catch语句块的异常的一个子类，就会执行catch语句块中的代码，然后如果存在finally语句块，控制将跳转到finally语句块。

如果抛出的异常类型和第一个catch语句块中的异常类型不匹配，CLR会跳转到下一个catch语句块并做同样的事情，直到找到一个匹配项。如果没有找到匹配项，CLR则会把异常对象抛给方法的调用者。如果调用者没有把调用方法的问题代码放在一个try语句块中，程序将会崩溃。

下面举例说明错误处理的用法，参见程序清单8-1中的NumberDouble类。当该类运行时，它会提醒我们输入。我们可以输入

任何内容，包括非数字。如果输入的内容成功地转换成一个数字，程序会乘以2并打印出结果；如果输入的是不合法的内容，程序将打印出“Invalid input”消息。

程序清单8-1 NumberDouble类

```
using System;

class NumberDoubler
{
    public void Test()
    {
        Console.WriteLine("Please type a number"
            + " between 0 and 255 that you want to double");
        string input = Console.ReadLine();
        try
        {
            Byte number = Byte.Parse(input);
            Console.WriteLine("Result: {0}", 2 * number);
        }
        catch (FormatException e) {
            Console.WriteLine("Invalid input.");
            Console.WriteLine(e.StackTrace);
        }
        catch (OverflowException)
        {
            Console.WriteLine("The number you entered exceeded"
                "capacity");
        }
    }
}
```

NumberDouble类使用System.Console类来获取用户的输入。

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

然后程序会使用System.Byte结构的静态Parse方法，把输入的字符串转换成一个byte。如果我们查看文档就可以了解到，这个方法可能会抛出以下异常之一。

- **ArgumentNullException**: 如果输入的字符串为空。
- **FormatException**: 如果输入的字符串不是一个数字。
- **OverflowException**, 如果输入的字符串小于0或大于255。

NumberDouble类中的Test方法并没有试图捕获ArgumentNullException异常，因为Console.ReadLine()输出的内容永远不会为空。还要注意，第2个catch语句块没有为OverflowException定义一个变量，因为语句块中的代码没有用到它。

测试NumberDouble类，编写并运行如下代码。

```
new NumberDoubler().Test();
```

我们可以在控制台看到如下的内容。

```
Please type a number between 0 and 255 that you want to double
```

如果输入的数字在0到255之间，我们可以看到数字增加了一倍。但是如果输入一个非数字，例如“abcd”，Parse方法会抛出一个FormatException，我们会看到一条错误消息及堆栈跟踪，它会告知错误的原因。

```
Invalid input.  
at System.Number.StringToNumber(String str, NumberStyles options,  
    NumberBuffer& number, NumberFormatInfo info, Boolean  
    parseDecimal)  
at System.Number.ParseInt32(String s, NumberStyles style,  
    NumberFormatInfo info)  
at System.Byte.Parse(String s, NumberStyles style,  
    NumberFormatInfo info)  
at System.Byte.Parse(String s)  
at NumberDoubler.Test() in C:\App08\NumberDoubler.cs:line 12
```

8.2 没有catch的try语句和using语句

try语句可以在没有catch语句块的情况下与finally一起使用。通常，我们用这样的语法来确保无论try语句块中的代码是否能够成功地完成，有些代码总是能够得到执行。例如，在打开一个数据库连接后，我们要确保在执行完连接后会调用它的Close方法。为了说明这种情况，我们来看如下打开数据库连接的虚拟程序代码。

```
Connection connection = null;
```

```
try
{
    // 打开连接
    // 做一些与连接相关的事情并执行其他任务
}
finally {
    if (connection != null)
    {
        // 关闭连接
        connection.Close();
    }
}
```

如果在try语句块中有一些意料之外的事情发生并且抛出了一个异常，这时我们通常会调用Close方法来释放资源。

在C#中，using语句（不要和用于导入命名空间的using指令混淆）可以作为一种方便的语法，来确保访问非托管资源的托管类型在使用完资源后，通过调用托管类型的Dispose方法来释放该资源。访问非托管资源的托管类型的例子有File和Font。例如，当创建一个文件时，System.IO.File类也会创建一个流对象，让我们能够对文件执行读取或写入操作。当我们不再需要这个文件时，也必须把对应的流正确地释放掉。通过using语句，我们可以编写如下非常短的代码。

```
using (FileStream fs = File.Create(fileName))
{
    //在这里对FileStream来做事情
}
```

我们也可以像下面这样用try和finally语句块来替代。

```
FileStream fileStream = null;
try
{
    fileStream = File.Create("C:/temp.txt");
    //在这里用FileStream来做事情
}
finally
{
    //在这里释放FileStream
}
```

我们将在第14章中看到更多使用using语句的示例。

8.3 System.Exception类

错误代码可以抛出任何类型的异常。例如，试图解析一个不合法的参数可能会抛出**System.FormatException**，在一个空引用变量上调用方法则会抛出**System.ArgumentNullException**。所有.NET的异常类都派生自**System.Exception**类。因此，我们有必要花点时间来介绍这个类。

此外，**Exception**类有**Message**和**StackTrace**两个属性。**Message**包含了对异常的描述，而**StackTrace**记录了调用栈上的当前帧。

大多数时候，**try**语句会带有一个**catch**语句块，用来捕获其他**catch**语句块所捕获的异常之外的**System.Exception**。捕获**Exception**的这个**catch**语句块一定要出现在最后。如果其他**catch**语句块都没有能够捕获异常，那么最后这个**catch**将会捕获它，示例如下。

```
try
{
    // 代码
}
catch (FormatException e)
{
    // 处理FormatException
}
catch (Exception e)
{
    // 处理其他异常
}
```

在上面代码中，我们可能会使用多个**catch**语句块，因为**try**语句块中的语句可能会抛出一个**FormatException**或其他类型的异常。如果抛出的是后者，最后一个**catch**语句块将会把它捕获。

但是要注意，**catch**语句块的顺序是很重要的。例如，我们不能把处理**System.Exception**的一个语句块放在任何其他语句块之前。这是因为CLR试图将抛出的异常和**catch**语句块中的参数匹配，而匹配的顺序就是参数出现的顺序。**System.Exception**会捕获所有异常，因此，在它之后的**catch**语句块就不会再执行了。

如果有多个catch语句块，并且这些catch语句块中的一个异常类型是派生自另一个catch语句块的类型，那我们就要确保特殊的异常类型放在最前边。

8.4 从方法中抛出异常

当在方法中捕获到一个异常时，我们有两种选择来处理这个方法中出现的错误，可以在方法内处理这个错误，这样悄悄地捕获这个异常而不需要通知调用者（前边的示例中已经展示过了）；也可以把这个异常抛回给调用者，让调用者来处理它。如果我们选择第二种方法，调用代码必须要捕获方法抛回的异常。

程序清单8-2展示了一个Capitalize方法，它把一个字符串的第一个字母改为大写。

程序清单8-2 Capitalize方法

```
public string Capitalize(string s) throws ArgumentNullException
{
    if (s == null)
    {
        throw new ArgumentNullException(
            "Your passed a null argument");
    }

    Character firstChar = s.charAt(0);
    String theRest = s.substring(1);
    return firstChar.toString().toUpperCase() + theRest;
}
```

如果向Capitalize传递null，它会抛出一个新的ArgumentNullException。我们来关注实例化ArgumentNullException类并抛出该实例的代码。

```
throw new ArgumentNullException(
    "You passed a null argument");
```

关键字throw用于抛出一个异常。注意不要把它和throws语句混淆，throws语句用在方法签名的末尾，用来表示可能会从这个方法中抛出给

定类型的一个异常。

下面的示例展示了调用Capitalize的代码。

```
String input = null;
try
{
    String capitalized = util.Capitalize(input);
    System.Console.WriteLine(capitalized);
}
catch (ArgumentNullException e)
{
    System.Console.Write(e.Message);
}
```

注意

构造函数也可以抛出异常。

8.5 异常处理中的最后注意事项

try语句会牺牲一些性能，因此，不要过度使用它。如果一个条件不难去测试，那么我们应该想办法去做测试而不是依靠try语句。例如，在一个null对象上调用方法会抛出一个ArgumentNullException。因此，我们一般是用一个try语句块将这个方法包围起来。

```
try
{
    ref.MethodA();
    ...
}
```

然而，在调用MethodA之前判断ref是否为空一点都不难。因此，下面的代码会更好一些，因为它没有用try语句块。

```
if (ref != null)
{
    ref.MethodA();
}
```

8.6 小结

本章讨论了结构化错误处理的使用并为每种情况展示了示例。我们还介绍了**System.Exception**类及其属性和方法。

第9章 数字和日期

在C#中，数字可以用byte、short、int、float、double和long等类型表示。这些C#类型分别是.NET中**System.Byte**、**System.Int16**、**System.Int32**、**System.Single**、**System.Double**和**System.Int64**结构的别名。另外，日期是用**System.DateTime**结构表示的。在使用数字和日期时，我们需要注意三个问题：解析、格式化和操作。

解析负责把字符串转换成数字或日期。解析很常见，因为计算机程序经常需要用户来输入并把接收到的用户输入内容当作字符串。如果一个程序期望得到一个数字或日期，但是接收到的是一个字符串，那么它就要先把字符串转换成一个数字或日期。转换不一定很直接。在进行转换前，我们先要读取该字符串，确保它只包含能转换成一个数字或日期的字符。例如，“123abc”就不是一个数字，即便它是以数字开头的。“123.45”是一个浮点数，但不是一个整数。“12/25/2013”看上去像是一个日期，但是只有在程序期望的日期格式是mm/dd/yyyy时才是合法的。把一个字符串转换成一个数字，叫做数字解析。把一个字符串转换成一个日期，叫做日期解析。

如果有一个数字或日期，我们可以用特殊的格式来显示它。例如，1000000如果显示成1,000,000会更容易阅读，而12/25/2013显示成Dec 25, 2013会更容易阅读。这些分别是数字格式化和日期格式化。

数字和日期的解析和格式化是本章的主题。这些任务在.NET中可以很容易地实现，因为前边讲到的结构为目的提供了相应的方法。另外，我们还会介绍**System.Math**类，它提供了数学运算的方法。此外，本章还会有一节介绍用于操作日期的**System.Calendar**类。

9.1 数字解析

C#程序可能要求用户输入一个数字，然后处理它或把它当作一个方法的参数。例如，一个货币转换程序就需要用户输入一个要转换的数值。我们可以用**System.Console**类的**ReadLine**方法来获取用户的输入。但是，输入将是一个字符串，即便它只包含了数字。在使用这个数字

前，例如把它乘以2，我们需要先解析这个字符串。成功的数字解析的结果是一个数字。

因此，数字解析的目的是把一个字符串转换成一个数值类型的数字。如果解析失败，例如，字符串不是数字或是超出了特定范围的数字，那么程序会抛出一个异常。

我们可以使用任何结构（只要该结构表示一个数字类型）的Parse方法来解析一个字符串。Parse的返回类型和包含Parse方法的结构是相同的。例如，**System.Int32**的Parse方法返回**Int32**，如果解析失败，会抛出**System.FormatException**。

举例来说，在程序清单9-1中，**NumberParsingTest**类用**Console.ReadLine()**获得用户的输入并解析输入内容。如果用户输入的是一个不合法的数字，将会显示一条错误消息。

程序清单9-1 解析数字(NumberParsingTest.cs)

```
using System;
namespace app09
{
    class NumberParsingTest
    {
        public static void Main()
        {
            Console.Write("Please type in a number:");
            String input = Console.ReadLine();
            try
            {
                int i = Int32.Parse(input);
                Console.WriteLine("The number entered: " + i);
            }
            catch (FormatException)
            {
                Console.WriteLine("Invalid user input");
            }
            Console.ReadKey();
        }
    }
}
```

9.2 数字格式化

数字格式使得数字的易读性更好。例如，如果把1000000打印成1,000,000或1.000.000，会更容易阅读。选择哪种格式主要看我们住在哪里。在美国和加拿大讲英语的省份，用逗号分隔，而在法国和印尼用的却是点。因此，数字和日期的格式依赖于用户的文化（或区域）。好在，区域信息在.NET中很容易处理。

在.NET中，区域信息用**System.Globalization.CultureInfo**类来表示。**CultureInfo**可能是非特定区域（当只指定了区域的语言元素时）或特定区域（当语言和国家都定义时）。如果对区域性不敏感，它也可以不区分区域性。

区域名称的格式是由ISO 639的两个字母的语言代码、一个连字符和ISO 3166的两个字母的国家代码组成。表9-1列出了ISO 639中的各种语言代码，而表9-2列出了ISO 3166中的一些国家代码（http://userpage.chemie.fu-berlin.de/diverse/doc/ISO_3166.html）。

表9-1 ISO 639语言代码示例

代 码	语 言		代 码	语 言
de	German		it	Italian
el	Greek		ja	Japanese
en	English		nl	Dutch
es	Spanish		pt	Portuguese
fr	French		ru	Russian
hi	Hindi		zh	Chinese

表9-2 ISO 3166国家代码示例

国 家	代 码		国 家	代 码
Australia	AU		France	FR
Brazil	BR		Germany	DE
Canada	CA		India	IN
China	CN		Mexico	MX
Egypt	EG		Switzerland	CH
Taiwan	TW		United States	US
United Kingdom	GB			

例如，**en-US**表示英语（美国），**en-GB**表示英语（英国），**fr-FR**表示法语（法国），**fr-CA**表示法语（加拿大），而**jp-JA**表示日语（日本）。只有**zh-Hans**（中文简体）和**zh-Hant**（中文繁体）是例外，它们两个都是非特定区域。

我们可以从以下网址获得完整的区域信息支持列表。

<http://msdn.microsoft.com/en-us/global/bb896001.aspx>

例如，构建在加拿大使用英语的一个**CultureInfo**对象，如下所示。

```
CultureInfo cInfo = new CultureInfo("en-CA");
```

还有一种方法，我们可以使用**CultureInfo**中的静态**CreateSpecificCulture**方法。

```
CultureInfo cInfo = CultureInfo.CreateSpecificCulture("en-CA");
```

现在，我们已经了解了足够多区域信息的相关背景知识，接下来，我们深入介绍如何在C#中格式化数字。答案很简单，使用数字类型的ToString方法即可。

下面是ToString方法的重载。

```
public string ToString()  
public string ToString(IFormatProvider provider)  
public string ToString(string format)  
public string ToString(string format, IformatProvider provider)
```

如果我们使用不接受参数的ToString方法，会使用默认的系统格式。如果我们在美国生活，系统格式将会是en-US。例如，下面的ToString方法把数字12345格式化成12345，和原始形式是一样的。

```
int value = 12345;  
Console.WriteLine(value.ToString()); // 打印12345
```

下面的代码把2e3f (2×10^3)打印成2000。

```
Float floatValue = 2e3f; //  $2 \times 10^3$   
Console.WriteLine(floatValue.ToString()); //打印2000
```

根据我们的需求，不接受参数的ToString方法可能满足需求，也可能不满足需求。如果不满足需求，我们可以使用这个方法的重载，把一个格式字符串传递给ToString。

```
Public string ToString(string format)
```

表9-3给出了标准数值的格式化字符串，我们可以把其中之一作为参数传递。

表9-3 标准数值的格式字符串

格式指示符	名 称		格式指示符	名 称

“C”或“c”	Currency		“N”或“n”	Number
“D”或“d”	Decimal		“P”或“p”	Percent
“E”或“e”	Exponential		“R”或“r”	Round-trip
“F”或“f”	Fixed-point		“X”或“x”	Hexadecimal
“G”或“g”	General			

关于数值的格式化字符串的更多信息，我们可以浏览如下网址。

<http://msdn.microsoft.com/en-us/library/dwhawy9k>

打印12345的示例代码如下。

```
int intValue = 12345;
Console.WriteLine(intValue.ToString("g"));
```

如果我们住在美国或其他与美国使用同样货币格式的国家，而计算机又设置成默认配置，我们会从如下代码段中得到\$12,345.00。

```
int intValue = 12345;
Console.WriteLine(intValue.ToString("c"));
```

换句话说，没有区域信息时，程序会采用默认的ToString方法。但是，如果我们想打印的内容不是计算机默认的格式，那要怎么办呢？比如，我们可能为另一个国家的客户工作。这种情况下，我们可以通过另一个ToString的重载来设置格式和区域信息。

```
public string ToString(string format, IFormatProvider provider)
```

记得前面的**CultureInfo**实现**IFormatProvider**吧，同样我们可以传递一个**CultureInfo**的实例作为第二个参数。例如，下面的代码以讲法语的加拿大地区（也就是魁北克省）的货币格式打印12345。


```
CultureInfo frenchCanadian = new CultureInfo("fr-CA");
Console.WriteLine(intValue.ToString("c", frenchCanadian));
// 打印12 345,00 $
```

不管计算机怎样设置，运行上面代码都会输出**12 345,00 \$**。

程序清单9-2中的**NumberFormatTest**类展示了如何使用ToString方法的不同重载来格式化一个数字。

程序清单9-2 NumberFormatTest类

```
using System;
using System.Globalization;

namespace app09
{
    class NumberFormatTest
    {
        public static void Main()
        {
            float floatValue = 2e3f; // 2 x 10^3
            Console.WriteLine(floatValue.ToString()); //打印 2000
            int intValue = 12345;
            Console.WriteLine(intValue.ToString());

            // 使用的格式化字符串
            Console.WriteLine(intValue.ToString("g"));
            //打印12345
            Console.WriteLine(intValue.ToString("c"));
            //如果是en-US,打印$12,345.00

            CultureInfo frenchCanadian = new CultureInfo("fr-CA");
            Console.WriteLine(intValue.ToString("c",
                frenchCanadian)); // 打印12 345,00 $
            Console.ReadKey();
        }
    }
}
```

9.3 System.Math类

Math类是一个工具类，它提供了数学运算的静态方法。它还有两

个静态的final double字段：**E**和**PI**。**E**（**e**）表示自然对数的底数，它的值接近2.718。**PI**（**pi**）是圆的周长与直径的比例系数，它的值是22/7或约等于3.1428。

下面是Math类中提供的一些方法。

```
public static double Abs(double a)
```

该方法返回指定的double值的绝对值。

```
public static double Acos(double a)
```

该方法返回一个角的反余弦，返回的角度范围在0.0到pi之间。

```
public static double Asin(double a)
```

该方法返回一个角的反正弦，返回的角度范围在-pi/2到pi/2之间。

```
public static double Atan(double a)
```

该方法返回一个角的反正切，返回的角度范围在-pi/2到pi/2之间。

```
public static double Cos(double a)
```

该方法返回一个角的余弦。

```
public static double Exp(double a)
```

该方法返回欧拉数e的double次幂。

```
public static double Log(double a)
```

该方法返回double值的自然对数（底数是e）。

```
public static double Log10(double a)
```

该方法返回double值的底数为10的对数。

```
public static double Max(double a, double b)
```

该方法返回指定的两个double值中较大的一个。

```
public static double Min(double a, double b)
```

该方法返回指定的两个double值中较小的一个。

9.4 使用Date和Time

在.NET Framework类库中至少有两个类可以用于操作日期和时间，分别是**System.DateTime**和**System.TimeSpan**。它们都提供了解析和格式化的方法，唯一的区别在于，它们所能操作的数据范围不同。本节将介绍这两个类。另外**System.Globalization.Calendar**类及其子类可以操作日期和时间。但是，我们这里只讨论**DateTime**和**TimeSpan**，不再讨论Calendar及其子类。

9.4.1 System.DateTime

DateTime表示一个时间点，例如，1945年8月9日或2020年12月20日下午6点20。

如程序清单9-3所示，**DayCalculator**类展示了如何用DateTime来解析和格式化一个日期。

程序清单9-3 DayCalculator类

```
using System;
using System.Globalization;

namespace App09
{
    class DayCalculator
    {
        public void CalculateDay()
        {
            Console.WriteLine("\nEnter a date in MM/dd/yyyy format: ");
            DateTime selectedDate;
```

```

string dateString = Console.ReadLine();
string format = "MM/dd/yyyy";
CultureInfo provider = CultureInfo.InvariantCulture;

try
{
    selectedDate = DateTime.ParseExact(dateString,
        format, provider);
    Console.WriteLine("{0} is/was a {1}",
        selectedDate.ToString("MMM dd, yyyy"),
        selectedDate.DayOfWeek);

    DateTime now = DateTime.Now;
    //创建一个同样日期的DateTime
    DateTime thisYear = new DateTime(
        now.Year, selectedDate.Month,
        selectedDate.Day);

    Console.WriteLine("This year {0} falls/fell "
        + "on a {1}",
        thisYear.ToString("MMM dd"),
        thisYear.DayOfWeek);
}
catch (FormatException) {
    Console.WriteLine("Invalid date. Note that the "
        + "month and date parts must be two digits. "
        + "For example, instead of 1/1/2011, "
        + "enter 01/01/2011");
}

static void Main(string[] args)
{
    DayCalculator dayCalculator =
        new DayCalculator();
    char tryAgain = 'y';
    while (tryAgain != 'n' && tryAgain != 'N')
    {
        if (tryAgain == 'y' || tryAgain == 'Y')
        {
            dayCalculator.CalculateDay();
        }
        Console.Write("\nTry again (y/n)?");
        tryAgain = Console.ReadKey().KeyChar;
    }
}
}
}
}

```

CalculateDay方法是DayCalculator类的主要方法。它接收用户输入、解析输入内容并打印出所输入日期是星期几。

9.4.2 System.TimeSpan

TimeSpan结构表示时间间隔，能够用于保存时间信息。**DateTime**可以用来包含日期和时间信息。不同的是，**TimeSpan**只能够保存时间信息。因此，如果不需要日期部分的话，**TimeSpan**要比**DateTime**更容易使用。

例如，App09解决方案中的TimeSpanExample项目，展示了在给定出发时间和到达时间的情况下，如何使用TimeSpan来计算一个航班所需的时间。在这个示例中，假设一个航班的出发时间和到达时间都在同一天，我们只需要关注时间信息。

如下是程序清单9-4中的**DurationCalculator**类。

程序清单9-4 DurationCalculator类

```
using System;
using System.Globalization;

namespace TimeSpanExample
{
    class DurationCalculator
    {
        public void CalculateDuration()
        {
            TimeSpan departure;
            TimeSpan arrival;
            string format = "h\\:mm";
            bool timeValid = true;
            do
            {
                Console.Write("Departure time (hh:mm):");
                string intervalString = Console.ReadLine();
                timeValid = TimeSpan.TryParseExact(intervalString,
                    format, CultureInfo.CurrentCulture,
                    TimeSpanStyles.None, out departure);
                if (!timeValid)
                {

```

```

        Console.WriteLine("Invalid time. Please "
            + "enter your departure time in hh:mm "
            + "format (Ex: 10:30 or 21:12)\n");
    }
}
while (!timeValid);

do
{
    Console.Write("Arrival time (hh:mm):");
    string intervalString = Console.ReadLine();
    timeValid = TimeSpan.TryParseExact(intervalString,
        format, CultureInfo.CurrentCulture,
        TimeSpanStyles.None, out arrival);
    if (!timeValid)
    {
        Console.WriteLine("Invalid time. Please "
            + "enter your arrival time in hh:mm "
            + "format (Ex: 10:30 or 21:12)\n");

    }
}
while (!timeValid);

if (arrival.CompareTo(departure) > 0)
{
    TimeSpan duration = arrival.Subtract(departure);
    Console.WriteLine("Your flight will take {0} "
        + "hour(s) and {1} minute(s)",
        duration.Hours, duration.Minutes);
}
else
{
    Console.WriteLine("You have entered an arrival "
        + "time that is earlier than the departure "
        + "time.\nPlease try again later. ");
}
}

static void Main(string[] args)
{
    DurationCalculator calculator = new
        DurationCalculator();
    calculator.CalculateDuration();
    Console.ReadKey();
}
}

```

```
}
```

DurationCalculator中的CalculateDuration方法做了所有的工作。它用了两个do-while循环来分别获取出发时间和到达时间。在每个循环中，CalculateDuration调用了Console.ReadLine()来接收用户输入并把输入内容传递给TimeSpan的静态方法TryParseExact。下面是用于出发时间的do-while循环。

```
bool timeValid = true;
do
{
    Console.Write("Departure time (hh:mm):");
    string intervalString = Console.ReadLine();
    timeValid = TimeSpan.TryParseExact(intervalString,
        format, CultureInfo.CurrentCulture,
        TimeSpanStyles.None, out departure);
    if (!timeValid)
    {
        Console.WriteLine("Invalid time. Please "
            + "enter your departure time in hh:mm "
            + "format (Ex: 10:30 or 21:12)\n");
    }
}
while (!timeValid);
```

如果解析成功，TryParseExact会返回true；如果解析失败，则返回false。TryParseExact的最后一个参数是输出参数，它表示成功解析了TimeSpan对象。通常，do-while会一直循环，直到用户输入合法的hh:mm格式的时间。

第二个while循环和第一个循环很类似，只不过它有了错误消息。在接收到合法的时间后，它会把出发时间和到达时间作比较。

```
if (arrival.CompareTo(departure) > 0)
{
    TimeSpan duration = arrival.Subtract(departure);
    Console.WriteLine("Your flight will take {0} "
        + "hour(s) and {1} minute(s)",
        duration.Hours, duration.Minutes);
}
else
{
    Console.WriteLine("You have entered an arrival "
```

```
        + "time that is earlier than the departure "  
        + "time.\nPlease try again later. ");  
    }
```

如果到达时间大于出发时间，`arrival.CompareTo(departure)`会返回一个正值，在这种情况下，到达时间减去出发时间，然后打印出结果（航程所用时间）。另一方面，如果出发时间等于或大于到达时间，**CalculateDuration**会打印出一条错误消息。

最后，Main方法实例化DurationCalculator并且调用CalculateDuration方法。

```
static void Main(string[] args)  
{  
    DurationCalculator calculator = new  
        DurationCalculator();  
    calculator.CalculateDuration();  
    Console.ReadKey();  
}
```

运行这个程序，我们会在控制台看到如下消息。

```
Departure time (HH:mm):
```

输入一个时间，如10: 00，按下“Enter”键。我们可以得到另一条消息，它要求输入到达时间。

```
Arrival time (HH:mm):
```

输入一个时间，比如12:00，按下“Enter”键。会得到第一个时间和第二个时间的间隔值。如果输入错误的时间或到达时间比出发时间小，我们将会得到一个错误消息。

9.5 小结

在C#中，数字用byte、short、int、float、double和long等类型来表示。这些C#类型分别是.NET中System.Byte、System.Int16、System.Int32、System.Single、System.Double和System.Int64结构的别

名。另外，日期用**System.DateTime**结构来表示。在本章中，我们介绍了在使用数字和日期时需要注意的三个问题：解析、格式化和操作。

第10章 接口和抽象类

C#初学者对接口经常会有这样的印象，接口就是没有实现代码的一个类。虽然从技术角度来讲，这种说法没有错，但是它混淆了接口的真正用途。接口的作用远不止于此。接口应该看作是服务提供者和它的用户之间的一个契约。在介绍如何编写一个接口之前，本章会先重点讲述这个概念。

本章的第2个主题是抽象类。从技术角度来讲，抽象类是一个不能被实例化并且必须由子类来实现的类。然而，抽象类很重要，因为许多情况下它能够扮演接口的角色。在本章中，我们还会介绍如何使用抽象类。

10.1 接口的概念

在第一次学习接口时，初学者经常关注如何编写一个接口，而不是理解接口背后的概念。他们认为接口就是使用关键字`interface`声明的一个类，它的方法没有方法体。

然而这样的描述是不准确的，把接口当作没有实现的一个类是很片面的。接口的一个更合适的定义是契约。它是服务提供者（`server`）和该服务的用户（`client`）之间的一个契约。有时这个契约由`server`定义，有时这个契约由`client`定义。

我们来看一个真实世界的案例。微软的Windows系统是当前世界上最流行的操作系统，但是微软并不生产打印机。要想打印，我们还要依靠惠普、佳能和三星等厂家。每个打印机供应商都有自己专有的技术。但是，他们的产品全部都可以用于打印Windows应用程序中的各种文档。这是如何做到的呢？

这是因为微软对打印机供应商表述了实现这些功能的关键：“如果想要你的产品在Windows上使用（我知道你们都这么想），那么你必须要实现这个`IPrintable`接口。”

这个接口就这么简单，如下所示

```
interface IPrintable
{
    void Print(Document document);
}
```

这里document是一个待打印的文档。

为了实现了这个接口，打印机供应商要编写打印机驱动程序。每个打印机都有一个不同的驱动程序，但是它们都会实现IPrintable。打印机驱动程序是IPrintable的一个实现。在这个示例中，这些打印机驱动程序就是服务提供者。

打印服务的用户是所有的Windows应用程序。我们在Windows中可以很容易地打印，是因为应用程序只需要调用Print方法并且传递一个Document对象给它。因为接口可以自由地使用，所以用户应用程序不用等待一个实现就可以进行编译。

关键是，不同的打印机打印不同应用程序的文档成为可能，这得益于IPrintable接口。它是打印服务提供者和打印用户之间的契约。

一个接口可以定义方法和其他成员。但是，接口中的方法并没有实现。为了少用，接口必须要有一个实现类，来真正执行其行为。

如图10-1所示，我们用UML类图表示IPrintable接口和它的实现。

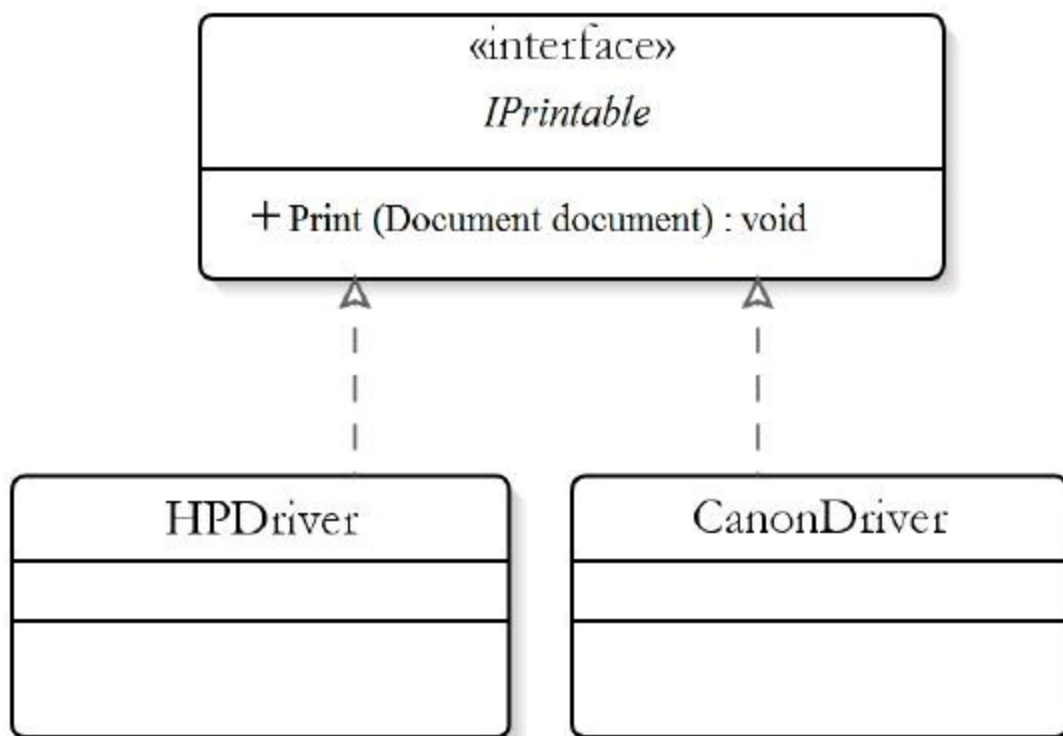


图10-1 类图表示一个接口和两个实现类

在这个类图中，接口和类有相同的形状，但是其名称是斜体的，并且有前缀<<interface>>。HPDriver和CanonDriver类实现了IPrintable接口，实现的内容当然也是不同的。HPDriver类中，Print方法包含了能够驱动惠普打印机进行打印的代码。CanonDriver中包含了佳能驱动程序打印的代码。在UML类图中，类和接口之间用带箭头的虚线连接起来。这种关系类型我们通常称为实现，因为这个类为接口中的抽象提供了真正的实现（实际工作的代码）。

注意

这个示例很牵强，但是问题和解决方案却是真的。我们希望它能让你更好地了解什么是真正的接口。接口就是一个契约。

阐明这一点的另一个真实案例是System.Data.IDbConnection接口，它是为所有数据提供者定义的一个接口，帮助我们在C#中连接到不同的关系数据库。只要数据库厂商提供了IDbConnection接口的实现，我们就能在任何的.NET应用中访问他们的数据库产品。

10.2 从技术角度看接口

现在，我们已经介绍了接口，接下来看看如何创建一个接口。在C#中，接口和类一样，是一种类型。编写一个接口的格式如下。

```
interface interfaceName
{
}
```

接口可以包含方法签名、属性、事件和委托。接口不可以包含那些成员的实现。另外，接口一定不能包含字段。

所有接口成员都是隐式public的。按照惯例，接口的名称用“I”作前缀。我们应该遵循这个规则，因为这是最佳的。

接口是最常用的类型之一，.NET Framework类库中有几百个接口。例如，System.Iclonable、System.IComparable、System.IFormatProvider、System.Collection.IList、System.Runtime.Serialization. ISerializable和System.Data.IDbConnection。

编写一个接口很简单。程序清单10-1展示了一个名为IPrintable的接口。

程序清单10-1 IPrintable接口

```
public interface IPrintable
{
    void Print(Object document);
}
```

这里IPrintable接口定义了一个Print方法。注意，Print是public的，虽然在方法声明的前面没有关键字public。事实上，在一个接口的方法声明中，我们不能使用访问修饰符（如public或protected）。注意，这

里只有Print的签名。实现是写在实现类或结构中的。

和类一样，接口是创建对象的一个模板。然而和常规的类不同的是，接口不能实例化。它直接定义了C#类能够实现的一套方法。

要实现一个接口，我们需要在类声明的后面使用冒号(:)。例如，程序清单10-2展示了实现了IPrintable接口的一个CanonDriver类。

程序清单10-2 IPrintable接口的一个实现

```
public class CanonDriver : IPrintable
{
    public void Print(Object document)
    {
        // 执行打印的代码
    }
}
```

一个实现类必须覆盖接口中的所有方法。接口和实现类的关系就像父类和子类的关系。类的实例也是接口的实例。例如，下面的if语句的结果为true。

```
CanonDriver driver = new CanonDriver();
if (driver is IPrintable)    // 结果为true
```

一个类可以实现多个接口。在类的定义中，多个接口名称之间用逗号隔开。例如，下面是实现了IPrintable和System.IComparable接口的一个类的定义。

```
public class MyPrinter : IPrintable, System.IComparable
```

当然，如果来实现多个接口，我们必须提供接口中所有方法的实现。

我们也可以编写扩展自基类并且实现接口的一个类。例如，下面代码定义了一个类，它扩展自BasePrinter并且实现了IPrintable和System.IComparable接口。

```
public class MyPrinter : BasePrinter, IPrintable, System.IComparable
```

需要注意的一点是，如果我们扩展了一个类并且实现了一个或多个接口，那么所扩展的类的名称要放在接口前边。例如，下面的声明会导致一个编译错误。

```
public class MyPrinter : IPrintable, System.IComparable, BasePrinter
```

接口支持继承。一个接口可以扩展另一个接口。如果接口B扩展了接口A，B就叫作A的子接口，A是B的父接口。因为B直接扩展了A，A就是B的直接父接口。任何扩展了B的接口也是A的间接子接口。如图10-2所示，一个接口扩展了另一个接口。注意，两个接口的连接线和类继承的连接线是一样的。

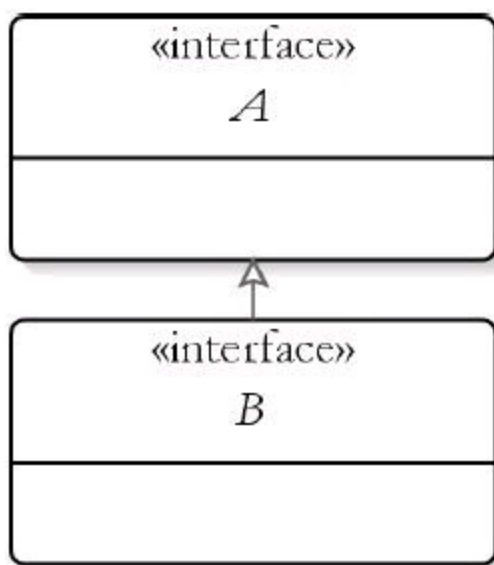


图10-2 扩展接口

大多数时候，我们会在一个接口中定义方法成员。我们可以在一个接口中声明的方法签名与在类中一样。但是，接口中的方法没有方法体，而是直接用分号结束。所有方法都是隐式public的且抽象的，在方法、属性、事件和委托等的签名中，有访问修饰符是不合法的。

接口中的方法的语法如下。

```
ReturnType MethodName(ListOfArgument);
```

注意，我们不能把接口中的方法声明为静态的，因为静态方法不能

是抽象的。

10.3 实现System.IComparable

下面的示例展示了接口作为服务提供者和服务用户之间契约的角色。这个例子的目的是让我们更容易掌握OOP语言中接口的概念。

我们在第5章中介绍过数组，而且简单地介绍了System.Array类有一个叫作Sort的静态方法，它能够对一个数组进行排序。它的签名如下。

```
public static void Sort(Array array)
```

但是，Sort是怎样知道如何为它根本不了解的对象排序的呢？Sort又是怎样知道Elephant对象要用体重来排序而Student对象要用姓名来排序的呢？这就需要接口来发挥作用。因为System.Array的编写者对于对象一无所知，他们只是签署了契约，保证Sort会把数组元素当作System.IComparable实例。所以，如果想要让数组中的对象是可排序的，我们就必须通过确保对象类实现了该接口，从而允许它们强制转型为IComparable的。

IComparable只有CompareTo这一个方法，我们可以覆盖该方法，以确定对象应该如何进行排序。

如程序清单10-3所示，IComparableImplementation项目包含了一个类。

程序清单10-3 实现IComparable

```
using System;

namespace App10
{
    class Student : IComparable
    {
        private string firstName;
        private string lastName;

        public Student(string firstName,
            string lastName)
```



```

    {
        this.lastName = lastName;
        this.firstName = firstName;
    }

    public string FirstName
    {
        get
        {
            return firstName;
        }
        set
        {
            firstName = value;
        }
    }

    public string LastName
    {
        get
        {
            return lastName;
        }
        set
        {
            lastName = value;
        }
    }

    public int CompareTo(Object obj)
    {
        Student anotherStudent = (Student) obj;
        if (this.lastName == anotherStudent.lastName)
        {
            return this.FirstName.CompareTo(
                anotherStudent.FirstName);
        }
        else
        {
            return this.LastName.CompareTo(
                anotherStudent.LastName);
        }
    }
}

class Program
{
    static void Main (String[]args)

```

```

    {
        Student[] students = {
            new Student("John", "Suzuki"),
            new Student("Liam", "Doe"),
            new Student("John", "Smith"),
            new Student("Joe", "Doe"),
            new Student("John", "Tirano"),
            new Student("Louis", "Smith")};

        Console.WriteLine("\nUnsorted:");
        Console.WriteLine("=====");
        foreach (Student student in students)
        {
            Console.WriteLine(student.LastName +
                               ", " + student.FirstName);
        }

        Array.Sort(students);

        Console.WriteLine("\nSorted:");
        Console.WriteLine("=====");
        foreach (Student student in students)
        {
            Console.WriteLine(student.LastName +
                               ", " + student.FirstName);
        }

        Console.ReadKey();
    }
}

```

运行程序清单10-3的代码，控制台会显示如下的结果。

```

Unsorted:
=====
Suzuki, John
Doe, Liam
Smith, John
Doe, Joe
Tirano, John
Smith, Louis

Sorted:
=====
Doe, Joe

```

```
Doe, Liam  
Smith, John  
Smith, Louis  
Suzuki, John  
Tirano, John
```

10.4 抽象类

在接口中，我们必须编写一个实现类来执行实际的操作。如果接口中有很多方法，你也要冒花费时间的风险去覆盖那些不使用的方法。抽象类有一个和接口相似的作用，即提供服务提供者和它的用户之间的契约，但同时抽象类可以提供部分实现。那些必须被显式地覆盖的方法，则可以声明为抽象的。因为我们不能实例化一个抽象类，所以仍然需要创建一个实现类，但是我们不必覆盖那些不需要使用或者改变的方法。

要创建一个抽象类，我们要在类的声明中用到`abstract` 修饰符。要创建一个抽象方法，我们就要在方法声明前使用 `abstract` 修饰符。程序清单 10-4 展示了一个名为 `DefaultPrinter`的抽象类。

程序清单10-4 `DefaultPrinter`类

```
public abstract class DefaultPrinter  
{  
    public string GetDescription()  
    {  
        return "Use this to print documents.";  
    }  
    public abstract void Print(Object document);  
}
```

`DefaultPrinter`中有两个方法，分别是`Print`和`GetDescription`。`GetDescription`方法有一个实现，所以我们不需要在实现类中覆盖这个方法，除非想改变它的返回值。`Print`方法声明为抽象的，而且它没有方法体。程序清单10-5中的`MyPrinterClass`类是`DefaultPrinter`的实现类。

程序清单10-5 `DefaultPrinter`的一个实现

```
public class MyPrinter : DefaultPrinter
```

```
{
    public override void Print(object document)
    {
        Console.WriteLine("Printing document");
        // 这里有一些代码
    }
}
```

例如MyPrinter这样的具体实现类，必须要覆盖所有的抽象方法。否则，它自己必须声明为抽象的。

通过声明一个抽象类的方法，我们可以告知类的使用者想让他们来扩展这个类。即使没有一个抽象方法，我们仍然可以声明一个抽象类。

在UML类图中，抽象类看上去和实体类一样，只不过它的名称是斜体的。图10-3展示了一个抽象类。

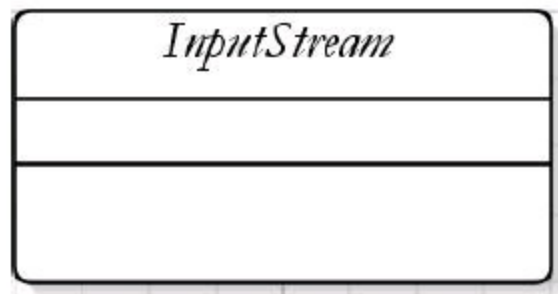


图10-3 一个抽象类

10.5 小结

接口在C#中扮演着重要的角色，因为接口在服务提供者和它的用户之间定义了一个契约。本章介绍了如何使用接口。一个基类通过提供默认的实现代码，为接口提供了一个通用的实现，并加快程序的开发。

抽象类和接口相似，但是它可以提供其方法的部分实现。

第11章 枚举

枚举是.NET Framework中的一种数据类型，它可以用于保存枚举值。我们主要用枚举来限制可以赋给变量或从方法中返回的可能值。

本章重点介绍这种数据类型。

11.1 枚举概览

我们用关键字enum来创建一个字段或一个方法的一系列有效值。例如，customerType字段可能只接受数值Individual和Organization。而对于state字段，合法的值可能是美国所有的州和加拿大所有的省。我们用enum可以很容易地限制程序只接受合法的值之一。

枚举类型可以是独立的，也可以是类的一部分。如果应用程序中的许多地方需要使用它，我们可以把它独立出来。如果只是用于类的内部，那么我们最好是把枚举作为这个类的一部分。

如程序清单11-1所示，它定义了枚举类型EmployeeType。

程序清单11-1 枚举类型EmployeeType

```
enum EmployeeType
{
    FullTime,
    PartTime,
    Permanent,
    Contractor
}
```

枚举类型EmployeeType有4个枚举值：FullTime、PartTime、Permanent和Contractor。枚举值区分大小写，而且按照惯例是首字母大写。两个枚举值之间用逗号隔开，它们可以写在同一行，也可以写成多行。为了提高易读性，我们把程序清单11-1中的枚举写成多行。

使用枚举就像使用类或接口一样。如下所示，程序清单11-2中的代

码使用了程序清单11-1中的EmployeeType作为一个字段类型。

程序清单11-2 使用EmployeeType枚举

```
using System;
namespace EnumExample
{
    enum EmployeeType
    {
        FullTime,
        PartTime,
        Permanent,
        Contractor
    }

    class Employee
    {
        private EmployeeType employeeType;
        public Employee(EmployeeType employeeType)
        {
            this.employeeType = employeeType;
        }
        public String getDescription()
        {
            if (employeeType == EmployeeType.Contractor)
            {
                return "Contractor, pay on hourly basis";
            }
            else if (employeeType == EmployeeType.FullTime)
            {
                return "Permanent, salary-based";
            }
            else if (employeeType == EmployeeType.PartTime)
            {
                return "Part-Time, mostly students";
            }
            else
            {
                return "Full-Time, salary-based";
            }
        }
    }
}
class Program
{
```

```

static void Main(string[] args)
{
    EmployeeType employeeType = EmployeeType.PartTime;
    Employee employee = new Employee(employeeType);
    Console.WriteLine(employeeType); // 打印 "PartTime"
    Console.WriteLine(employee.getDescription());
    Console.ReadKey();
}
}

```

程序清单11-2中，使用枚举中的一个数值就像使用一个类的静态成员一样。例如，下面代码展示了EmployeeType的使用。

```
EmployeeType employeeType = EmployeeType.PartTime;
```

注意，这里是怎样把枚举类型EmployeeType的枚举值PartTime赋值给变量employeeType的呢？因为employeeType变量是EmployeeType类型的，所以只能将EmployeeType中定义的值赋于它。

乍一看，使用枚举和使用常量没有区别。其实枚举和常量之间有一些基本的区别。当只能接受预定义的数值时，常量就不是一个好的解决方案。例如，我们来看一下程序清单11-3中的CustomerTypeStaticFinals类。

程序清单11-3 使用常量

```

class CustomerTypeStaticFinals
{
    public const int INDIVIDUAL = 1;
    public const int ORGANIZATION = 2;
}

```

假设我们有一个叫作OldFashionedCustomer的类，它的customerType字段是整数类型。下面的代码创建了一个OldFashionedCustomer实例并且给它的customerType字段赋值。

```

OldFashionedCustomer ofCustomer = new OldFashionedCustomer();
ofCustomer.customerType = 5;

```

在这里使用常量的话，我们无法避免把一个无效的整数赋予 `customerType`。要确保一个变量只会被赋予正确的数值，枚举要更优于常量。

11.2 类中的枚举

我们可以把枚举当作类的成员。如果枚举只用于类的内部，我们可以使用这种方法。例如，程序清单11-4中的代码是程序清单11-3的修改版。和程序清单11-3不同的是，程序清单11-4中的代码，将枚举类型 `EmployeeType` 声明为 `Employee` 类的一个字段。

程序清单11-4 使用枚举类型作为一个类的成员

```
using System;

namespace EnumExample2
{
    class Employee
    {
        public enum EmployeeType
        {
            FullTime,
            PartTime,
            Permanent,
            Contractor
        }
        private EmployeeType employeeType;
        public Employee(EmployeeType employeeType)
        {
            this.employeeType = employeeType;
        }
        public String getDescription()
        {
            if (employeeType == EmployeeType.Contractor)
            {
                return "Contractor, pay on hourly basis";
            }
            else if (employeeType == EmployeeType.Permanent)
            {
                return "Permanent, salary-based";
            }
            else if (employeeType == EmployeeType.PartTime)
            {

```



```

        return "Part-Time, mostly students";
    }
    else
    {
        return "Full-Time, salary-based";
    }
    }
}
class Program
{
    static void Main(string[] args)
    {
        Employee.EmployeeType employeeType =
            Employee.EmployeeType.FullTime;
        Employee employee = new Employee(employeeType);
        Console.WriteLine(employeeType); // 打印"FullTime"
        Console.WriteLine(employee.getDescription());
        Console.ReadKey();
    }
}
}

```

11.3 switch语句中的枚举

switch语句也可以使用枚举中的枚举值。程序清单11-5中的代码给出了在switch语句中使用枚举类型DayOfWeek的一个示例。

程序清单11-5 switch语句中的枚举

```

using System;

namespace EnumExample3
{
    enum DayOfWeek
    {
        Monday, Tuesday, Wednesday, Thursday, Friday, Saturday,
        Sunday
    }

    class Program
    {
        static void Main(string[] args)
        {
            DayOfWeek day = DayOfWeek.Sunday;

```

```
switch (day)
{
    case DayOfWeek.Monday:
    case DayOfWeek.Tuesday:
    case DayOfWeek.Wednesday:
    case DayOfWeek.Thursday:
    case DayOfWeek.Friday:
        Console.WriteLine("Week day");
        break;
    case DayOfWeek.Saturday:
    case DayOfWeek.Sunday:
        Console.WriteLine("Week end");
        break;
}
Console.ReadKey();
}
```

在程序清单11-5中，switch语句接受DayOfWeek中的一个值。如果值是Monday、Tuesday、Wednesday、Thursday或Friday，它会打印“Week day”；如果值是Saturday或Sunday，它会打印“Week end”。

11.4 小结

C#支持枚举。枚举是一个特殊的类，它是System.Enum的一个子类。枚举要优于整数，因为它们更安全。我们也可以在switch语句中使用枚举并且遍历它的值。

第12章 泛型

泛型是在.NET Framework 2.0中增加的最重要的特性之一。有了泛型，我们就可以编写参数化的类型，通过传递一种（或一些）类型来创建它的实例。那么这些对象就会被限制为这种（或这些）类型。除了参数化的类型，泛型也支持参数化的方法。

泛型的优势包括在编译时进行更严格的类型检查以及性能上的提升。另外，当使用System.Collections命名空间中的成员时，泛型可以消除大部分原来必须要执行的类型强制转换。

12.1 为什么要使用泛型

最简单的方法是通过一个示例来论证。我们来看一下在.NET Framework 1.0中就有的System.Collections.ArrayList类。ArrayList是一个能够包含对象的集合。要增加一个对象到ArrayList，我们就要调用它的Add方法。这个方法的签名如下。

```
public virtual int Add(Object value)
```

例如，下面代码实例化了ArrayList,并且为它增加了一个产品编码（一个字符串）。

```
ArrayList productCodes = new ArrayList();  
codes.Add("ABC");
```

因为所有的引用类型和值类型都是派生自System.Object的，事实上，我们可以给ArrayList.Add传递任意类型的对象。这种设计是有意为之，因为ArrayList要充当能够包含任意类型对象的一个通用容器。

在把对象保存到ArrayList之后，我们就可以像获取一个数组元素一样来获取其元素。

```
arrayList[index]
```

注意，返回值是Object类型，为了充分利用它，我们需要把它向下转换。修改上面的代码，如下所示。

```
ArrayList productCodes = new ArrayList();
codes.Add("ABC");
// 获取第一个元素
string productCode = (string) productCodes[0];
```

我们必须把对象向下转换成实际对象类型，这可能会有一点不方便，但是只能这么做。另一方面，我们来看下面的代码，它会导致更可怕的结果。

```
int x = 123;
ArrayList productCodes = new ArrayList();
productCodes.Add("ABC");
productCodes.Add(x);
```

这里，我们传递了两种不同的类型，字符串（“ABC”）和整数（x）。现在，假设我们试图遍历整个ArrayList，把其中的所有对象都当作字符串。

```
int count = codes.Count;
for (int i = 0; i < count; i++)
{
    // 用大写字母打印
    string code = (string)codes[i];
    Console.WriteLine(code.ToUpper());
}
```

你认为会出现什么情况？当遇到第二个元素时，程序会崩溃，因为这个元素的类型是int，而试图把一个int转换成string，会导致运行时错误。

因为这个致命的缺陷，ArrayList并不好用。

为了弥补这个缺陷，.NET的设计者在.NET 2.0中新增了泛型。所以，现在有了一个叫作System.Collections.Generic的命名空间。

以System.Collections.Generic.List为例，这个类和System.Collections.ArrayList类似，只不过List是参数化的。下面是List这

个类的声明。

```
public class List<T> : IList<T>, ICollection<T>,
    IEnumerable<T>, IList, ICollection, IEnumerable
```

这里，T表示一个类型，它所能够保存的对象的类型。为了实例化List，我们必须指定一个类型，例如int或string。如下例所示，我们声明字符串的一个List。

```
List<string> codes = new List<string>();
```

这意味着，我们只能为codes添加字符串。添加非字符串类型则会导致一个编译错误。

```
List<string> codes = new List<string>();
codes.Add("ABC");
codes.Add(123); // 编译错误
```

换句话说，我们不能犯向一个List传递两种不同的对象类型这样的错误。最重要的是，既然List只能接受一个指定的类型，我们就不需要向下转换获取的元素。如下的代码段展示了这一点。

```
List codes = new List();
codes.Add("ABC");

//这里增加更多的元素

//遍历List
int count = codes.Count;
for (int i = 0; i < count; i++)
{
    //用大写字母打印
    string code = codes[i]; // 不用向下转换
    Console.WriteLine(code.ToUpper());
}
```

在第13章中，我们会看到System.Collections.Generic中的更多成员的应用。现在，让我们了解更多有关泛型的知识。

12.2 泛型介绍

正如12.1节中所介绍的，泛型类型可以接受参数。这就是为什么我们常常把泛型类型叫作参数化的类型。

声明一个泛型类型就像声明一个非泛型类型，只要用尖括号把泛型的类型变量括起来即可。

```
MyType<typeVar1, typeVar2, ...>
```

例如，要声明一个System.Collections.Generic.List，编写方式如下。

```
List<T> myList;
```

T叫作类型变量，也就是将要使用一种类型来替换的变量。类型变量的替代值可以在泛型类型中用作参数类型或方法的返回类型。对于List类，当创建一个实例时，我们可以把T用作Add或其他方法的参数类型，也可以把T用作Find或其他方法的返回类型。List类的Add和Find的签名如下。

```
public void Add<T item>  
public T Find(Predicate<T> match)
```

注意

一个使用类型变量T的泛型类型，在声明或实例化它时，允许传递T。另外，如果T是一个类，我们也可以传递T的一个子类；如果T是一个接口，我们可以传递实现了T的一个类。

我们把string传递给一个List的声明，如下所示。

```
List<string> myList;
```

myList所引用的List实例的Add方法将会得到一个string对象作为其参数，而且其Find方法将会返回一个string。因为Find返回一个指定类型的对象，所以我们不需要向下转换。

为了实例化一个泛型类型，我们可以在声明它的时候传递相同的参

数列表。例如，创建一个使用string的List，可以在尖括号中传递string。

```
List<string> myList = new List<string>();
```

如下例所示，程序清单12-1比较了**ArrayList**和**List**。

程序清单12-1 比较非泛型ArrayList和泛型List

```
using System;
using System.Collections;
using System.Collections.Generic;

namespace ArrayListVsList
{
    class Program
    {
        static void Main(string[] args)
        {
            ArrayList arrayList = new ArrayList();
            arrayList.Add("Life without generics");
            // 转换成字符串
            String s1 = (String) arrayList[0];
            Console.WriteLine(s1.ToUpper());

            List<string> list = new List<string>();
            list.Add("Life with generics");
            // 不需要类型转换
            String s2 = list[0];
            Console.WriteLine(s2.ToUpper());

            Console.ReadKey();
        }
    }
}
```

在程序清单12-1中，ArrayList和List逐行进行比较。声明List<string>告诉编译器，List的这个实例只能保存string。获取这个List的成员元素时不需要向下转换，因为它已经返回了想要的类型，也就是string。

如果运行程序清单12-1中的代码，我们会在控制台看到如下内容。

```
LIFE WITHOUT GENERICS
```

注意

使用泛型类型时，类型检查是在编译时进行的。

这里最让人感兴趣的是，泛型类型本身就是一种类型，它可以当作类型变量来使用。例如，如果想要一个存储string列表的List，我们可以传递List<string>作为其类型变量，从而声明这个List，代码如下。

```
List<List<string>> myListOfListsOfStrings;
```

要获取myList中第1个列表中的第1个字符串，我们可以这样编写。

```
string s = myListOfListsOfStrings[0][0];
```

程序清单12-2展示了使用List的一个类，这个List会接受一个string类型的List作为参数。

程序清单12-2 使用List的List

```
using System;
using System.Collections.Generic;

namespace ListOfLists
{
    class Program
    {
        static void Main(string[] args)
        {
            List<string> listOfStrings = new List<string>();
            listOfStrings.Add("Hello again");
            List<List<string>> listOfLists =
                new List<List<string>>();
            listOfLists.Add(listOfStrings);
            string s = listOfLists[0][0];
            Console.WriteLine(s); // 打印 "Hello again"
            Console.ReadKey();
        }
    }
}
```


另外，泛型可以接受多种类型参数。例如可以用于保存键值对的 `System.Collections.Generic. Dictionary` 类，其定义如下。

```
Public class Dictionary<TKey, TValue> : ...
```

`Tkey` 用于表示键的类型，`TValue` 用于表示值的类型。`Dictionary` 类的 `Add` 方法的签名如下。

```
public void Add(TKey key, TValue value)
```

程序清单12-3给出了使用 `Dictionary` 的一个示例。

程序清单12-3 使用泛型 `Dictionary`

```
using System;
using System.Collections.Generic;

namespace DictionaryExample
{
    class Program
    {
        static void Main(string[] args)
        {
            Dictionary<int, string> flowers =
                new Dictionary<int, string>();
            flowers.Add(1001, "Lily");
            flowers.Add(1002, "Rose");
            flowers.Add(1003, "Lotus");

            string favorite = flowers[1002];
            Console.WriteLine(favorite);

            Console.ReadKey();
        }
    }
}
```

在程序清单12-3中，我们创建了一个 `Dictionary`，它用 `int` 作为键类型；用 `string` 作为值类型。即使没有强制类型转换，获取的值也总是 `string` 类型的。

12.3 应用限制

在12.2节中，我们已经介绍过List和Dictionary类允许传递任意类型作为参数类型。但是，我们也可以限制用于参数化类型的类型。例如，如果有一个MathUtility类，它提供了处理数学运算的方法，将MathUtility的参数类型限制为表示数字的值类型是有意义的。

表12-1列出了可以用于类型参数的约束。

表12-1 类型参数的约束

约 束	说 明
where T: struct	类型参数必须是值类型
where T: class	类型参数必须是引用类型，包括任何类、接口、委托和数组类型
where T: new()	类型参数必须有一个不接受参数的public的构造函数，如果还有其他约束，那么本条约束必须放在最后
where T: <base class name>	类型参数必须是指定的基类或是指定基类的子类
where T: <interface name>	类型参数必须是指定的接口或是实现指定接口的一种类型
where T: U	为T提供的类型参数，必须与为U提供的参数相同或者是从它派生的子类型

例如，为了将类型参数限制为值类型，这里使用where T:struct约束。

```
Class MathUtility<T> where T: struct
{
```

```
}
```

如果我们把一个引用类型传递给`MathUtility`，那么会得到一个编译错误，告诉我们这个类型参数必须是值类型。

作为另一个例子，下面的类期望得到了`Comparable`的一个类型参数。

```
class ObjecUtil<T> where T: Comparable
{
}
```

12.4 编写泛型类型

编写泛型类型和编写其他类型并没有太大的区别，只是我们声明一个类型变量的列表，会在类型中的某处有意使用其中的类型变量。这些类型变量放在类型名称之后的尖括号中。例如，程序清单2-4中的`Point`类就是一个泛型类。`Point`对象表示坐标系中的一个点，它有x部分（横轴）和y部分（纵轴）。通过让`Point`泛型化，我们可以指定一个`Point`实例的精确度。例如，如果一个`Point`对象需要非常精确，我们可以传递`Double`作为类型变量，否则，`Integer`就足够了。

程序清单12-4 泛型`Point`类

```
using System;

namespace CustomGenericType
{
    struct Point<T>
    {
        T x;
        T y;
        public Point(T x, T y)
        {
            this.x = x;
            this.y = y;
        }
        public T X
        {
            get { return x; }
        }
    }
}
```

```

        set { this.x = value; }
    }

    public T Y
    {
        get { return y; }
        set { this.y = value; }
    }

    public void Print()
    {
        Console.WriteLine("{0}, {1}", x, y);
    }
}

```

在程序清单12-4中，T是为Point类提供的类型变量。T被用作返回值及X和Y属性的参数类型。另外，这个构造函数也接受两个T类型变量。

使用Point就像使用其他泛型类型一样。例如，程序清单12-5中的代码创建了两个Point对象，即point1和point2。point1将Integer传递为类型变量，point2将Double传递为类型变量。

程序清单12-5 测试Point

```

using System;

namespace CustomGenericType
{
    class Program
    {
        static void Main(string[] args)
        {
            Point<int> a = new Point<int>();
            a.Print();
            Point<double> b = new Point<double>(12.3, 244.4);
            b.Print();

            Console.ReadKey();
        }
    }
}

```

如果运行该程序，我们会在控制台看到如下内容。

```
(0, 0)
(12.3, 244.4)
```

12.5 小结

泛型使得编译时的类型检查更加严格。特别是在使用**System.Collections.Generic**命名空间的成员时，泛型有两大贡献。首先，在编译时，泛型对集合类型增加了类型检查，从而将集合所能保存的对象类型限制为传递给它的类型。例如，我们现在可以创建一个**System.Collections.Generic.List**实例来保存字符串并且它不接受**Integer**或其他类型。其次，从集合获取一个元素时，泛型消除了要进行类型强制转换的必要。

在本章中，我们还介绍了将不同的类型变量传递给泛型类型，会得到不同的类型。这就是说，**List<String>**和**List<Object>**是不同的类型。即便**System.String**是**System.Object**的子类，传递一个**List<String>**给期待接受**List<Object>**泛型的方法，仍然会产生一个编译错误。

最后，我们看到编写泛型和编写普通的C#类型并没有太大的不同。我们只需要在类型名称后面的尖括号内声明一个类型变量列表就可以了。然后，我们可以使用这些类型变量作为方法返回值的类型或作为方法参数的类型。

第13章 集合

在编写一个面向对象的程序时，我们经常会使用一组对象。在第5章我们介绍过，可以用数组来分组对象并且遍历所有的元素。遗憾的是，数组缺少快速开发应用程序所需要的灵活性。比如，数组不能调整大小。好在，.NET Framework提供了一套接口和类，它们使得操作成组的对象更容易一些。这些接口和类是**System.Collections**命名空间和它的子命名空间中的一部分。**System.Collections**包含的非泛型类型，或多或少被**System.Collections.Generic**命名空间中的泛型类型替代了。本章会介绍**System.Collections.Generic**中最为常用的类型。

13.1 概述

集合是用来组织其他对象的一个对象，也可以把它当作一个容器。集合提供了方法来保存、获取和操作其元素。集合帮助C#程序员更容易地管理对象。

C#程序员应该熟悉**System.Collections.Generic**命名空间中最重要的类型。这个命名空间中有很多类、结构和接口，因为篇幅所限，我们这里只介绍最重要的几个：**List**、**HashSet**、**Queue**和**Dictionary**。

List、**HashSet**和**Queue**类似，它们都用于存储相同类型的对象。**Dictionary**很适合保存键值对。这些类型会在后边一一介绍。

13.2 List类

List和数组类似，但是更具有灵活性。在创建一个数组时，我们必须指定一个不能更改的数组大小。而用**List**指定的大小是可选的。在为一个**List**添加元素时，如果对于新元素没有更多的空间，它的大小可以自动增长。

List是一个泛型类，因此，我们需要告诉编译器想要在其中保存何种类型的对象。下面介绍如何创建**string**的一个**List**。

```
List<string> animals = new List<string>();
```

如果有必要，我们也可以为List指定大小。下面例子是指定初始大小为10的List。

```
List<string> animals = new List<String>(10);
```

在这两个示例中，如果添加的元素超过了已有的大小，List会自动增加它的大小。但是，如果我们事先知道List中要保存多少元素，那么，指定一个初始值作为元素的最大数目是一个好办法。通过这种方法，我们可以节省重新调整List大小所需要的时间。

对于List，最常用的操作就是添加元素、获取元素、得到元素的总数目以及遍历所有的元素。

给List添加一个元素需要调用它的Add方法。要统计一个List中包含多少个对象，需要调用它的Count属性。注意，Count和Capacity属性是不同的。Capacity告诉我们List的大小。我们很少需要知道某一个时刻一个List的大小，因为它能够自动增长。

要从一个List中获取一个元素，需要使用它的Item属性。之前我们曾经介绍过，调用这个属性要传递一个索引给这个List变量，就好像List是数组一样，语法如下。

```
myList[0]
```

第1个元素用索引0表示，第2个元素用1表示，以此类推。

最后，要遍历List中的所有元素，需要使用foreach循环。

```
foreach (T element in myList)
{
    // 用元素来做一些事情
}
```

13.2.1 重要的方法

下面是List中的一些较为重要的方法。

```
public void Add(T item)
```

该方法添加一个新的对象到List。新增的项可以为null，而且可以放在List的结尾处。我们比较这个方法和Insert方法。

```
public void Clear()
```

该方法删除List中的所有元素并且把Count属性设置为0。

```
public bool Contains(T item)
```

该方法判断一项是否在List中，如果是则返回true；否则，则返回false。

```
public T Find(Predicate<T> match)
```

该方法查找List并且返回第1个符合指定条件的元素。

```
public void Insert(int index, T item)
```

该方法在指定索引位置添加一个元素。

```
public bool Remove(T item)
```

该方法从List中删除指定的项，如果成功删除元素，返回true；否则，返回false。

```
public void RemoveAt(int index)
```

该方法从List中删除指定索引位置的项，给这个方法传递0，表示要删除List中的第1个元素。

```
public void Sort()
```

该方法用默认的比较方法对List排序。

```
public T[] ToArray()
```


该方法返回元素数组。

13.2.2 List示例

程序清单13-1给出了一个示例，一个List用来保存字符串，我们用foreach语句遍历它。

程序清单13-1 List示例

```
namespace ListExamples
{
    class Program
    {
        static void Main(string[] args)
        {
            List<string> titles = new List<string>();
            titles.Add("No Sun No Moon");
            titles.Add("The Desert Story");
            titles.Add("Belle!");

            string selectedTitle = titles[1];
            Console.WriteLine("Selected: " + selectedTitle);
            Console.WriteLine();

            foreach (string title in titles)
            {
                Console.WriteLine("title: " + title);
            }

            Console.ReadKey();
        }
    }
}
```

这个类的运行结果如下。

```
Selected: The Desert Story

title: No Sun No Moon
title: The Desert Story
title: Belle!
```

13.3 HashSet类

集合（set）是一种数据结构，它能保存没有特定顺序的值，而且不允许重复的值。集合是用哈希表实现的集合（也可以用其他方法实现集合）。哈希这个术语指的是一个函数，它用于计算一个元素的索引来快速获取该元素。

在.NET Framework中，HashSet类表示一个哈希集合。HashSet和List很类似，也有Add和Clear方法，它们的工作方式和List的Add和Clear方法类似，还有用于返回HashSet中元素总数目的Count属性。但是，和List不同的是，HashSet不允许重复的元素。除此之外，它也不能在指定的索引位置添加和删除元素，甚至没有一个Item能够让我们获取指定位置的元素。

13.3.1 有用的方法

下面是HashSet中的一些较为重要的方法。

```
public bool Add(T item)
```

该方法为Hash添加一个新的对象，如果要添加的项在HashSet中已经存在，则不会再新增并且该方法会返回false。

```
public void Clear()
```

该方法删除HashSet中的所有元素并且把Count属性设置为0。

```
public bool Contains(T item)
```

该方法判断一个项是否在HashSet中，如果是则返回true；否则，返回false。

```
public bool IsSubsetOf(IEnumerable other)
```

该方法判断该HashSet是否是指定集合的一个子集。

```
public bool IsSupersetOf(IEnumerable other)
```

该方法判断该HashSet是否是指定集合的一个超集。

13.3.2 HashSet示例

程序清单13-2展示了如何使用HashSet来添加字符串，在添加一个重复值时，我们需要留意Add方法会返回什么。

程序清单13-2 HashSet的一个示例

```
using System;
using System.Collections.Generic;

using System;
using System.Collections.Generic;

namespace HashSetExamples
{
    class Program
    {
        static void Main(string[] args)
        {
            HashSet<string> productCodes = new HashSet<string>();
            bool added = productCodes.Add("1234");
            Console.WriteLine("1234 added? " + added);

            //重复，这个无法添加
            added = productCodes.Add("1234");
            Console.WriteLine("1234 added? " + added);
            added = productCodes.Add("999");
            Console.WriteLine("999 added? " + added);

            Console.WriteLine("\nProduct Codes:");
            foreach (string productCode in productCodes)
            {
                Console.WriteLine("product code: " + productCode);
            }
            Console.ReadKey();
        }
    }
}
```

运行这个程序，我们会在控制台看到如下结果。

```
1234 added? True
1234 added? False
999 added? True

Product Codes:
product code: 1234
product code: 999
```

13.4 Queue类

Queue是类似于List和HashSet的一个集合。Queue的一个显著特点就是可以用Dequeue方法在获取一个元素并同时删除它。要从Queue中获取一个元素但是不删除它，我们需要使用Peek方法。

13.4.1 有用的方法

下面是在Queue中定义的一些较为重要的方法。

```
public void Enqueue(T item)
```

该方法把一个新的元素添加到Queue的末尾。添加的元素可以为null。

```
public void Clear()
```

该方法删除Queue中所有的元素并且把它的Count属性设置为0。

```
public bool Contains(T item)
```

该方法判断一个元素是否在Queue中，如果是则返回true；否则，返回false。

```
public T Dequeue()
```

该方法从Queue的队头返回元素并且把它从Queue中删除。

```
public T Peek()
```

该方法从Queue的队头返回元素，但不删除该元素。

13.4.2 Queue示例

如程序清单13-3所示，它使用一个Queue来添加字符串元素。

程序清单13-3 Queue的一个示例

```
using System;
using System.Collections.Generic;

namespace QueueExamples
{
    class Program
    {
        static void Main(string[] args)
        {
            //测试 Queue
            Queue<string> cities = new Queue<string>();
            cities.Enqueue("Ottawa");
            cities.Enqueue("Ottawa");
            cities.Enqueue("Helsinki");

            foreach (string city in cities)
            {
                Console.WriteLine("city: " + city);
            }

            while (cities.Count > 0)
            {
                Console.WriteLine("selected: " + cities.Dequeue());
            }

            Console.ReadKey();
        }
    }
}
```

运行该程序，我们会在控制台看到如下消息。

```
city: Ottawa  
city: Ottawa  
  
city: Helsinki  
selected: Ottawa  
selected: Ottawa  
selected: Helsinki
```

13.5 Dictionary 类

Dictionary类是用来创建保存键值对的容器的一个模板。**Dictionary**适用于保存由一个键和一个值组成的元素，例如，一个ISBN和一个Book对象，或者一个国家和一个首都。

要构造一个Dictionary，我们需要把键的类型和值的类型传递给构造函数。例如，下面的代码段生成了一个Dictionary，它把一个字符串作为键并且把一个Book对象作为值。

```
Dictionary books = new Dictionary(string, Book);
```

像其他类型集合一样，它也有Add方法来为Dictionary添加键值对，还有Clear方法来删除其所有的元素。

要获取一个值，我们要使用它的Item属性。例如，如果Dictionary包含一个国家及其首都的对，用如下的语法来获取一个值。

```
string selectedCountry = countryDictionary[countryName];
```

Dictionary示例

如程序清单13-4中的代码所示，它展示了用一个Dictionary来保存国家和首都的对。

程序清单13-4 Dictionary示例

```
using System;  
using System.Collections.Generic;
```

```
namespace DictionaryExamples
{
    class Program
    {
        static void Main(string[] args)
        {
            Dictionary<string, string> capitals =
                new Dictionary<string, string>();
            capitals.Add("France", "Paris");
            capitals.Add("Australia", "Canberra");
            capitals.Add("Canada", "Ottawa");

            Console.WriteLine(capitals["Canada"]);
            // 打印 "Ottawa"

            Console.ReadKey();
        }
    }
}
```

运行程序清单13-4中的程序，我们会得到如下输出。

```
Ottawa
```

13.6 小结

集合很适合于操作一组对象。.NET Framework类库提供了许多可以很容易使用的集合类型。.NET Framework的1.0版本主要提供的是System.Collection命名空间中的集合类型。.NET 2.0中添加了泛型，它也带来了参数化的集合类型，可以在System.Collections.Generic命名空间中找到它。我们通常应该使用System.Collections.Generic中的成员，而不是使用System.Collections中的那些成员。在本章中，我们介绍了System.Collections.Generic命名空间中最重要4个成员，它们分别是List、HashSet、Queue和Dictionary。

第14章 输入输出

输入和输出（I/O）是计算机程序中最常执行的操作之一。I/O操作的示例包括以下几种。

- 创建和删除文件
- 从文件或网络套接字中读取或写入
- 序列化（或保存）对象到永久存储并且获取保存的对象

从.NET Framework 1.0开始，就以System.IO命名空间及其子命名空间的形式来支持I/O。本章根据功能来介绍主题，而且选择System.IO命名空间的最重要成员进行介绍。

本章第1个主题就是文件和目录的处理和操作。我们将介绍如何创建和删除文件和目录以及如何操作它们的属性。接下来，我们会在14.2节中介绍什么是流以及如何使用流。流就像水管一样，它可以促进数据传输。要按照顺序对流进行读取和写入，这就意味着在读取数据的第2个数据单元前，必须先要读取第1个数据单元。流有许多种类型。

FileStream、NetworkStream和MemoryStream都是流的一些例子。为了简单起见，一些工具类可以用来操作这些流，从而不必直接使用流。我们也会介绍这些工具类。

14.1 文件和目录的处理与操作

.NET Framework类库分别用File和Directory类来创建和删除文件与目录。此外，我们使用File和Directory就可以操作一个文件或一个目录，例如检查一个文件或一个目录是否存在。另外还有一个FileSystemInfo类，它是FileInfo和DirectoryInfo的父类。FileInfo提供了File中的许多功能，而DirectoryInfo提供的方法，可以做Directory的方法能做的事情。对于初学者来说，选择使用哪一个，有时很容易混淆。

下面我们会详细介绍用File和Directory能做什么。

14.1.1 创建和删除文件

用System.IO.File的Create方法创建一个文件。下面是该方法的签名。

```
public static FileStream Create(string path)
```

例如，下面的代码段创建了一个文件并且把它赋值给一个FileStream：

```
string fileName = @"C:\users\jayden\wintemp.txt";
try
{
    FileStream fs = File.Create(fileName);
}
catch (IOException e)
{
    Console.WriteLine(e.Message);
}
finally
{
    // 这里调用 fs.Close()
}
```

如果要创建的文件已经存在，那么Create方法会用一个新的文件来覆盖这个文件。而且，大多数时候，我们需要把File.Create放在try语句块中，因为如果操作失败的话，它就会抛出一个**IOException**异常。例如，如果我们没有在指定目录下创建文件的权限，这么做可能会失败。

操作File的一种更方便的语法是像下面这样使用using语句。

```
using (FileStream fs = File.Create(fileName))
{
    // 在这里用FileStream做一些事情
}
```

使用这种语法不需要担心如何关闭文件，因为using语句会处理它。但是，如果Create方法创建文件失败，它可能会抛出一个异常。因此，尽管使用using语句，我们也需要将其放入到try语句块中。

```
try
{
    using (FileStream fs = File.Create(fileName))
```

```
    {  
        // 在这里用FileStream做一些事情  
    }  
}  
catch (Exception e)  
{  
    // 处理异常  
}
```

14.1.2 创建和删除一个目录

`System.IO.Directory`类提供了静态方法来创建和删除目录与子目录。为了创建一个目录，我们使用`CreateDirectory`方法。

```
public static DirectoryInfo CreateDirectory(string path)
```

`CreateDirectory`返回一个`DirectoryInfo`，它暴露了一些目录的属性，诸如创建时间、最近访问时间等。另外，`DirectoryInfo`类提供了方法来获取和操作当前目录中的文件。在14.1.3节中，我们会介绍更多关于`DirectoryInfo`类的内容。

如果操作不能完全成功，`CreateDirectory`可能会抛出一个异常。例如，如果没有足够的权限去完成这个任务，该方法可能会抛出一个`UnauthorizedAccessException`异常。同样，试图创建一个与已存在的文件目录路径相同的目录，也会抛出一个`IOException`异常。

要删除一个目录，就要使用`Directory`类的`Delete`方法。下面是这个方法的两个重载方法。

```
public static void Delete(string path)  
public static void Delete(string path, boolean recursive)
```

在第1个重载方法中，目录必须可写并且为空。试图删除不为空的目录将会抛出一个`IOException`异常。但是，在使用第2个重载方法中，如果为它的第2个参数传递`true`，我们就可以删除一个非空的目录。注意，如果目录中包含一个只读文件，第2个重载方法将会失败。

14.1.3 操作File和Directory的属性

FileInfo类和**DirectoryInfo**类分别用于操作文件和目录。我们可以用**FileInfo**创建和删除一个文件，代码甚至比使用**File**类还要简短。下面介绍如何用**FileInfo**来创建一个文件。首先我们需要创建一个**FileInfo**实例。

```
String path = @"C:\temp\note.txt";
FileInfo fileInfo = new FileInfo(path);
using (FileStream fileStream = fileInfo.Create())
{
    // 用fileStream做一些事
}
```

FileInfo有优于**File**的地方。例如，通过调用一个**FileInfo**的**Extension**属性，我们可以很容易地获取一个文件的扩展名。另外，通过调用**Parent**属性，我们也可以得到一个**FileInfo**的父目录。更不用说，其**Length**属性返回文件的字节大小，**CreationTime**返回创建时间，**isReadOnly**表示文件是否是只读的，**Exists**返回一个**boolean**值以表示文件是否存在的。

最后一个属性**Exists**，可能要引起我们的注意。如果已经创建了一个指向文件的**FileInfo**，而该文件不存在，该怎么办呢？事实上，创建一个**FileInfo**，只是简单地在内存中创建一个对象，而并没有真正地创建一个文件。我们仍然需要调用**Create**来创建这个文件。当然，在创建一个**FileInfo**时，如果把这个路径传递给一个已存在的文件，它的**Exists**属性会返回**true**。

DirectoryInfo类和**FileInfo**类似，也提供了一组相似的属性，诸如**CreationTime**、**Exists**、**Extension**和**Parent**。**DirectoryInfo**还提供了创建和删除目录、获取子目录的列表和获取目录中的文件的列表等方法。下面是**GetFiles**的签名，它返回了一个**FileInfo**的数组。

```
public FileInfo[] GetFiles()
```

下面是**GetDirectories**的签名，它返回了**DirectoryInfo**的一个数组，每个**DirectoryInfo**表示当前目录下的一个子目录。

```
public DirectoryInfo[] GetDirectories()
```

14.1.4 列出目录下的文件

获取一个目录下的文件列表的最简单的方法，是使用Directory类的GetFiles方法。下面的代码是打印输出C盘下的所有文件。

```
string[] paths = Directory.GetFiles(@"C:\");
foreach (string path in paths)
{
    Console.WriteLine(path);
}
```

请注意，Directory.GetFiles返回一个字符串数组。数组中的每个元素包含一个文件的完整目录，例如C:\markets.doc。

我们也可以创建一个DirectoryInfo，并且调用它的GetFiles方法，如下所示：

```
DirectoryInfo directoryInfo = new DirectoryInfo("C:\\");
FileInfo[] files = directoryInfo.GetFiles();
foreach (FileInfo file in files)
{
    Console.WriteLine(file.Name);
}
```

DirectoryInfo.GetFiles返回一个FileInfo的数组，而Directory.GetFiles返回的是一个字符串数组。

14.1.5 复制和移动文件

复制一个文件很简单。我们可以通过调用File类的静态方法Copy，来创建一个文件的备份。例如，下面的这行代码，为C:\temp下的market.pdf文件创建了一个副本文件market2.pdf，并将其放在C:\temp下。

```
File.Copy(@"C:\temp\market.pdf", @"C:\temp\market2.pdf");
```

使用File中的静态方法Move，来移动一个文件。下面的代码段把C:\temp\research.pdf移动到C:\research.pdf。

```
File.Move(@"C:\temp\research.pdf", @"C:\research.pdf");
```

14.2 输入/输出流

可以把I/O流比作水管。就像水管连接城市中的家庭和蓄水池，流连接了C#代码和“数据蓄水池”。这个“数据蓄水池”叫做池，可以是文件、网络套接字或内存。使用流的最大优点，就是用统一的方法来向池中或池外传送数据，由此可以简化代码。我们只需要构造正确的流。

所有的流都是从System.IO.Stream抽象类派生而来。我们不需要直接使用Stream，而是使用它的派生类，诸如FileStream、MemoryStream或NetworkStream。我们曾经在前一小节介绍过FileStream的功能。例如，当使用File.Create创建一个文件时，会创建一个FileStream以允许我们写这个文件。

直接使用流很难，因为我们必须要自己管理这个数据流。好在，System.IO命名空间提供了几个工具类来操作流。每个工具类都属于两个组之一，要么是读，要么是写。

下面是最常用的读写方法。

- **StreamReader**: 从流中读取字符的工具类。
- **StreamWriter**: 向流中写入字符的工具类。
- **BinaryReader**: 从流中读取二进制数据的工具类。
- **BinaryWriter**: 向流中写入二进制数据的工具类。

所有reader和writer类，接受一个Stream作为它们的构造函数的参数，所以在创建reader或writer时，我们只需要确保传递正确的流。例如，如果我们想要创建一个文本文件，我们需要获取指向该文件的一个FileStream。最常见的是，在调用File类中的一个方法（如Create）时，会创建一个FileStream。此外，File类中的方法可能已经返回一个reader或writer，而不需要显示地创建一个FileStream。例如，File类的OpenText方法返回一个StreamReader，它与底层文件连接。

14.3节将介绍如何从文件中读取和向文件写入字符和二进制数据。

14.3 读取文本（字符）

我们使用`StreamReader`类从一个`stream`中读取字符。大多数时候，可以使用文件流，因为它是最常用的流类型。但是，还有很多其他类型的流，诸如网络流和内存流。

我们可以创建一个`StreamReader`，只要传递一个`Stream`给它的构造函数就可以了：

```
StreamReader streamReader = new StreamReader(stream);
```

但是，可能不需要显式地创建一个`StreamReader`。例如，`File`类的`OpenText`方法返回了一个`StreamReader`，它和一个`FileStream`关联。因此，我们可以调用`OpenText`并且得到一个`StreamReader`，如下所示。

```
StreamReader reader = File.OpenText(path)
```

创建一个`FileStream`并且把它传递给`StreamReader`类的构造函数，效果也是一样的。

```
FileStream fileStream = [create/obtain a FileStream];  
StreamReader streamReader = new StreamReader(fileStream);
```

不管是怎样创建`StreamReader`的，只要有一个实例，就可以调用`StreamReader`的各种`Read`方法。这个`Read`方法重载，返回了流中的下一个字符。它的签名如下。

```
public override int Read()
```

请注意，字符是作为`int`返回的，所以要打印这个字符，需要把它转化成一个`char`，如下所示。

```
StreamReader streamReader = ...  
char c = (char) streamReader.Read();
```

一次读取一个字符可能不是最有效的方法。我们常常想要一次读取多个字符。在这种情况下，可以使用`Read`方法的如下的重载形式：

```
public override int Read(char[] buffer, int index, int count)
```

这个Read方法的重载，从stream中读取了下一个count字符，并把它们复制到用作第一个参数的字符数组中。index参数（第2个参数）表示写入字符数组的开始位置。如果index为零，那么数组的第1个元素将会得到第1个字符读取。这个方法返回了实际读取的字符数量。

下面是从一个流中读取字符块的一个示例。

```
StreamReader streamReader = ...  
char[] buffer = new char[100];  
streamReader.Read(buffer, 0, 100);  
// buffer 现在包含从流中读取的字符
```

还有一个ReadLine方法，它从流中读取一行文本，并且以一个字符串的形式返回。该方法的签名如下。

```
public override string ReadLine()
```

程序清单14.1展示了从一个文本文件中读取字符的代码。

程序清单14.1 从一个文件中读取字符

```
using System;  
using System.IO;  
namespace StreamReaderExample  
{  
    class Program  
    {  
        static void Main(string[] args)  
        {  
            String path = "C:\\temp\\today.txt";  
            Try  
            {  
                using (StreamReader reader = File.OpenText(path))  
                {  
                    string line;  
                    // 从文件中读取并显示每行文本  
                    while ((line = reader.ReadLine()) != null)  
                    {  
                        Console.WriteLine(line);  
                    }  
                }  
            }  
        }  
    }  
}
```

```
        }  
    }  
    catch (IOException e)  
    {  
        Console.Write(e.Message);  
    }  
  
    Console.ReadKey();  
}  
}
```

程序清单14.1中的代码，打开了C:\temp下的today.txt文件，读取并打印文件中的每行文本。要测试这段代码，请确保在forementioned目录下创建一个today.txt文件。

14.4 写入文本（字符）

使用StreamWriter类，可以把文本或字符写入到流。这个类为各种数据类型提供了很多Write方法，所以不需要先把一个非字符串转换成一个字符串。有针对Single、Unit32、char的方法等。下面是一些Write方法的签名：

```
public virtual void Write(bool value)  
public virtual void Write(char value)  
public virtual void Write(int value)  
public virtual void Write(double value)  
public virtual void Write(string value)
```

还有一个重载方法，它允许在单独的一次操作中就写入一个字符块：

```
public virtual void Write(char[] buffer, int index, int count)
```

在这个例子中，*buffer*包含了要写入的字符，*index*表示了数组的起始元素，*count*表示buffer中要写入的字符的数量。

WriteLine方法也可以接受一个值。在值的末尾，这个方法添加了一个行终止符。下面是WriteLine方法的一些重载。


```
public virtual void WriteLine()  
public virtual void WriteLine(bool value)  
public virtual void WriteLine(char value)  
public virtual void WriteLine(int value)  
public virtual void WriteLine(double value)  
public virtual void WriteLine(string value)
```

第1种重载形式不接受参数，它用来为流增加一个行终止符。

程序清单14.2中的代码接受从控制台的输入，并把它写入到一个文件中。它会一直读取，直到用户输入一个空字符串。

程序清单14.2 把文本写入到一个文件中

```
using System;  
using System.IO;  
  
namespace StreamWriterExample  
{  
    class Program  
    {  
        static void Main(string[] args)  
        {  
            Console.WriteLine(  
                "Please type in some text. " +  
                "Keep typing until you're tired." +  
                "Enter an empty line to exit");  
            using (StreamWriter writer =  
                new StreamWriter(@"C:\temp\yoursay.txt"))  
            {  
                String input = Console.ReadLine();  
                while (input.Trim().Length != 0)  
                {  
                    writer.WriteLine(input);  
                    input = Console.ReadLine();  
                }  
            }  
        }  
    }  
}
```

14.5 读取和写入二进制数据

要从文件中读取二进制数据，需要用BinaryReader类。要创建一个BinaryReader很容易，只需传递一个流给它的构造函数即可：

```
BinaryReader reader = new BinaryReader(stream);
```

然后，我们调用它的众多Read方法之一。例如，要读取整数，就调用ReadInt16或ReadInt32方法。要读取一个双精度浮点数，就调用ReadDouble方法。BinaryReader的其他方法包括ReadBoolean、ReadChar、ReadByte、ReadDecimal、ReadInt64和ReadString。

使用BinaryWriter把一个二进制数据写入到一个文件。就像BinaryReader一样，创建BinaryWriter的一个实例也很容易。只需要把一个流传递给它的构造函数：

```
BinaryWriter writer = new BinaryWriter(stream);
```

BinaryWriter提供了多个Write方法重载。事实上，每种数据类型都有一个Write方法，所以可以写入一个整数、双精度浮点数、小数等。

程序清单14.3展示了如何把10个整数写入到C:\temp目录下的numbers.dat文件中，并且再把它们读取回来。

程序清单14.3 读取并写入二进制数据

```
using System;
using System.IO;

namespace BinaryReaderWriterExample
{
    class Program
    {
        static void Main(string[] args)
        {
            string path = @"C:\temp\numbers.dat";
            Random random = new Random();
            FileStream fileStream = File.OpenWrite(path);
            using (BinaryWriter writer =
                new BinaryWriter(fileStream))
            {
                for (int i = 0; i < 10; i++)
                {
```

```
        writer.Write(random.Next(0, 100));
    }
}

fileStream = File.OpenRead(path);
using (BinaryReader reader =
    new BinaryReader(fileStream))
{
    for (int i = 0; i < 10; i++)
    {
        int number = reader.ReadInt32();
        Console.WriteLine(number);
    }
}

Console.ReadKey();
}
}
```

如果运行这个程序，我们会看到10个0到100之间的数字。每次运行这个程序，都会看到一组不同的数字，因为每个数字都是用Random对象随机生成的。

14.6 小结

输入和输出操作是通过System.IO命名空间中的成员支持实现的。操作文件和目录是通过File、Directory、FileInfo和DirectoryInfo这些类来实现的。读取和写入数据是通过流来实现的。本章介绍了如何用以下的类从文件中读取和写入字符与二进制数据：StreamReader、StreamWriter、BinaryReader和BinaryWriter。

第15章 WPF

到目前为止，前边各章中的所有示例，都是作为控制台应用程序构建的。许多人都想了解另外一种应用程序，也就是桌面应用程序。现在是时候介绍WPF（Windows Presentation Foundation）了，这是可以用于开发桌面应用程序的技术。

本章会介绍WPF，并且会创建一些简单的应用程序。请注意，本章所介绍的内容，仅仅是WPF所能做事情的冰山一角。如果你想对这一技术有更多的了解，需要阅读专门介绍WPF的书籍，或者访问微软WPF和Windows Forms的官方网站<http://windowsclient.net>。

15.1 概述

.NET Framework 1.0版本带有两种开发桌面应用程序的技术，分别是Windows Forms和GDI+。随着.NET 3.0的发布，微软用WPF替代了上述两种技术。为什么要做这种改变呢？很简单。.NET桌面开发人员如果要使用Windows forms和GDI+的话，必须要掌握各种技术，而这些技术都集成在了WPF中。WPF是一种更好的解决方案，如果要开发新的桌面应用程序，应该使用它。

开发WPF应用程序有两种方法。

- 只使用代码；
- 使用代码和XAML。

在本章中，这两种方法都会介绍到。首先只用代码来构建WPF应用程序，然后用代码和XAML来构建WPF应用程序。

15.2 应用程序和窗口

本节要介绍开发WPF应用程序所用到的两个重要的类：`System.Windows.Application`类和`System.Windows.Window`类。在本节末

尾，还会提供两个示例。

一个WPF应用程序用System.Windows.Application类的一个实例或派生自System.Windows.Application的类的一个实例来表示。在创建了一个Application对象后，调用其Run方法来启动这个应用程序。就这么简单。

Run方法有两种重载形式：

```
public int Run()  
public int Run(Window window)
```

第2种重载形式，接受表示窗口的一个System.Windows.Window对象。没有窗口，就没有可视化。创建扩展自Application的一个类时，通常会用到第1种Run重载形式，在启动应用程序时，会自动调用该方法来创建窗口。在本章所介绍的项目中，这两个Run方法的应用我们都会看到。

需要记住一件重要的事情，就是在创建一个WPF应用程序时，必须要用[STAThread]属性来表示Main方法。这个属性的用途是什么？这个属性把当前线程的单元状态，更改为单线程的。WPF应用程序必须是单线程的。如果忘了使用这个属性，WPF应用程序将会崩溃。

现在，让我们先把注意力转换到Window类上。从名字可以看出，Window表示一个窗口。在它上面，可以增加控件，诸如标签、文本框和按钮等。关于控件的更多细节，会在本章后边的15.3节中介绍。

表15-1 较为常用的Window的属性

属 性	说 明
Content	Window的内容
ContextMenu	Window的右键菜单
FontFamily	在Window中用到的字体

FontSize	Window中用到的字体大小
FontStyle	Window中用到的字体样式
Height	Window的高度（以像素为单位）
Icon	Window使用的图标对象
IsActive	Window是否是激活状态
IsVisible	Window是否可见
Left	Window的左边位置相对于桌面的左边的距离
MaxHeight	Window的最大高度
MaxWidth	Window的最大宽度
MinHeight	Window的最小高度
MinWidth	Window的最小宽度
Opacity	从0.0到1.0的浮点数区间决定了Window的透明度，默认值是1.0
Padding	设置控件在Window中的内边距
Parent	Window的父控件
Title	Window的标题

Top	Window的上边位置相对于桌面的上边的距离
Width	Window的宽度
WindowStartupLocation	在第1次启动时，Window的位置
WindowState	Window的状态。它的值可能是以下三者之一：Minimized、Maximized和Normal

Window的最重要的属性就是Show。Show 打开窗口，并且不等新打开的窗口关闭就返回。如果不调用Show，就永远也看不到漂亮的窗口。Show的签名如下所示。

```
public void Show()
```

另一个需要牢记的Window成员是Content属性，它是Object类型。我们可以给这个属性赋任意的对象，但是通常会赋一个控件或面板。面板是包含其他控件的一个区域，我们会在本章后边的15.4节中讨论。

表15-1列出了Window类中较为重要的属性。现在，我们已经了解了WPF背后的知识，接下来让我们创建两个简单的WPF应用程序，它们会用到Application、Window和控件。

15.2.1 简单的WPF应用程序1

在这个例子中，我们用**Application**类创建了一个WPF应用程序并且通过传递一个Window对象来启动它。

程序清单15-1 简单的WPF 1

```
using System;
using System.Windows;
using System.Windows.Controls;

namespace App15
{
    class SimpleWPF1
```

```

{
    //WPF应用程序应该执行一个单线程单元（STA）
    [STAThread]
    static void Main(string[] args)
    {
        Window window = new Window();
        Label label = new Label();
        label.Content = "The WPF is cool";
        window.Content = label;
        window.Title = "Simple WPF App";
        window.Height = 100;
        window.Width = 300;
        window.WindowStartupLocation =
            WindowStartupLocation.CenterScreen;
        Application app = new Application();
        app.Run(window);
    }
}

```

程序清单15-1中的代码非常直接。首先，创建了一个Window对象和一个Label对象。然后，给这个Label的Content赋了一个字符串，并且将这个Label设置为这个窗口的Content。接下来，设置Window的Title、Height、Width和WindowStartupLocation属性，然后Window已经就绪了。在实例化Application类后，调用其Run方法。

现在，把SimpleWPF1项目设置为Visual Studio Express solution的默认项目，并且按下F5键。将会看到如图15-1所示的一个窗口。

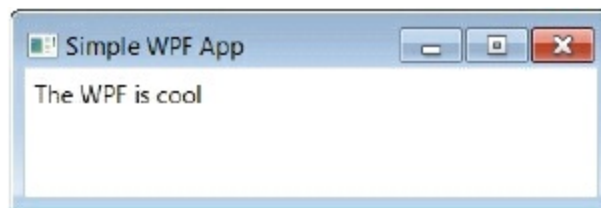


图15-1 第1个简单的WPF应用程序

这看上去就像一个标准的窗口，有最小化、恢复和关闭按钮。如果点击关闭按钮，窗口会关闭，并且结束程序。

15.2.2 简单的WPF应用程序2

我们曾经介绍过，Application类的Run方法有不接受参数的重载形式。你可能会问，如果使用这种Run重载形式，如何能够给这个应用程序传递一个Window呢？答案是，调用Run的任意重载方法，都将导致Application实例的OnStartup方法被调用。这里，编写代码来创建一个Window对象，就可以得到与传递一个Window给其他Run方法相同的结果。在这个例子中，我们通过扩展Application类创建了一个WPF应用程序，并且将这个窗口的创建委托给其OnStartup方法。

代码如程序清单15-2所示。

程序清单15-2 通过扩展Application，创建一个WPF应用程序

```
using System;
using System.Windows;
using System.Windows.Controls;

namespace App15
{
    class SimpleWpf2 : Application
    {
        //WPF应用程序应该执行一个单线程单元（STA）
        [STAThread]
        static void Main(string[] args)
        {
            SimpleWpf2 app = new SimpleWpf2();
            app.Run();
        }

        protected override void OnStartup(StartupEventArgs e)
        {
            base.OnStartup(e);
            Calendar calendar = new Calendar();
            Window window = new Window();
            window.Title = "WPF 2";
            window.Height = 205;
            window.Width = 200;
            window.Content = calendar;

            window.WindowStartupLocation =
                WindowStartupLocation.CenterScreen;
            window.Show();
        }
    }
}
```

SimpleWPF2项目中包含了这些代码。要运行它，把SimpleWPF2设置为默认项目，并且在Visual Studio Express solution中按下F5键。将会看到包含了日历的一个窗口，如图15-2所示。



图15-2 另一个简单的WPF应用程序

15.3 WPF控件

我们已经在前边的示例中见过两个控件，Label和Calendar。.NET Framework提供了几十个像Label和Calendar一样的现成控件。简单一些的控件包括Label、TextBox和Button。复杂一些的控件则包括Calendar、RichTextBox和WebBrowser。还有面板，是可以用来容纳其他控件的控件。像DockPanel、Grid和StackPanel，都是面板的示例。

下面是WPF控件的不完整列表。

- **Border:** 表示用于装饰另一个控件的边框。
- **Button:** 表示一个可以点击的按钮。
- **Calendar:** 表示一个可以滚动的日历。
- **Canvas:** 表示包含子元素的一个空白屏幕。
- **CheckBox:** 表示一个复选框。
- **ComboBox:** 表示带有下拉列表的一个组合框。
- **DataGrid:** 表示在定制表格中显示数据的一个控件。
- **DatePicker:** 表示可从中选择一个日期的一个控件。
- **DockPanel:** 表示一个面板，它的子元素可以相对于彼此水平排列或垂直排列。
- **Grid:** 表示带一个由行和列组成的表格的一个面板。

- **Image**: 表示显示图像的控件。
- **Label**: 表示一个不可编辑的文本框。
- **ListBox**: 表示可选的一个列表。
- **Menu**: 表示元素能够分级组织的一个菜单。
- **Panel**: 表示所有面板元素的基类。
- **PasswordBox**: 表示供输入密码的控件。
- **PrintDialog**: 表示一个打印对话框。
- **ProgressBar**: 表示一个进度条。
- **RadioButton**: 表示一个单选按钮。
- **RichTextBox**: 表示一个富文本编辑控件。
- **Separator**: 表示在项目控件中用于分隔项目的一个控件。
- **Slider**: 表示一个滑块控件。
- **SpellCheck**: 充当诸如**TextBox**或**RichTextBox**这样的文本编辑控件的拼写检查器。
- **StackPanel**: 一个可以容纳子元素的面板，子元素可以排列成水平的一行或垂直的一行。
- **TabControl**: 表示一个控件，它所包含的项目占用屏幕上相同的空间。
- **TextBox**: 表示能够显示文本的一个控件，用户可以编辑它。
- **ToolBar**: 表示包含一组命令按钮或其他控件的一个容器。
- **ToolTip**: 表示一个提示框。
- **TreeView**: 表示在一个树结构中显示数据的一个控件。
- **Validation**: 提供数据验证的支持。
- **WebBrowser**: 表示嵌入在WPF应用程序中的一个Web浏览器。

在接下来的示例中，我们会看到一些控件的使用。

15.4 面板和布局

很少有只用一个控件的WPF应用程序。大多数时候，会在应用程序中用到多个控件。在这种情况下，那些控件必须放置到诸如面板的一个容器中。面板是一个可以向其中添加子元素的长方形。

System.Windows.Controls.Panel类表示一个面板，并且在相同的命名空间中提供了几个实现。一个面板可以包含其他的面板。

在使用面板时，我们必须要考虑子控件应该如何排列。程序清单

15-3中的代码显示了如何使用一个面板以及如何布局它的控件。

程序清单15-3 面板示例

```
using System;
using System.Windows;
using System.Windows.Controls;

namespace App15
{
    class PanelExample : Application
    {
        //WPF应用程序应该执行一个单线程单元 (STA)
        [STAThread]
        static void Main(string[] args)
        {
            PanelExample app = new PanelExample();
            app.Run();
        }

        protected override void OnStartup(StartupEventArgs e)
        {
            base.OnStartup(e);
            PanelWindow window = new PanelWindow();
            window.Title = "Panel Example";
            window.Height = 228;
            window.Width = 200;

            window.WindowStartupLocation =
                WindowStartupLocation.CenterScreen;
            window.Show();
        }
    }

    class PanelWindow : Window
    {
        public PanelWindow()
        {
            StackPanel mainPanel = new StackPanel();
            Calendar calendar = new Calendar();
            Button button1 = new Button();
            button1.Content = "Previous Year";
            Button button2 = new Button();
            button2.Content = "Next Year";

            StackPanel buttonPanel = new StackPanel();
            buttonPanel.Orientation = Orientation.Horizontal;
```

```

        buttonPanel.HorizontalAlignment =
            System.Windows.HorizontalAlignment.Center;
        buttonPanel.Children.Add(button1);
        buttonPanel.Children.Add(button2);

        mainPanel.Children.Add(calendar);
        mainPanel.Children.Add(buttonPanel);

        this.Content = mainPanel;
    }
}

```

如图15-3所示，这里有一些控件布局的很合理。使用所提供的布局的好处是，当调整窗口的大小时，布局是保留不变的，如图15-4所示。



图15-3 使用面板

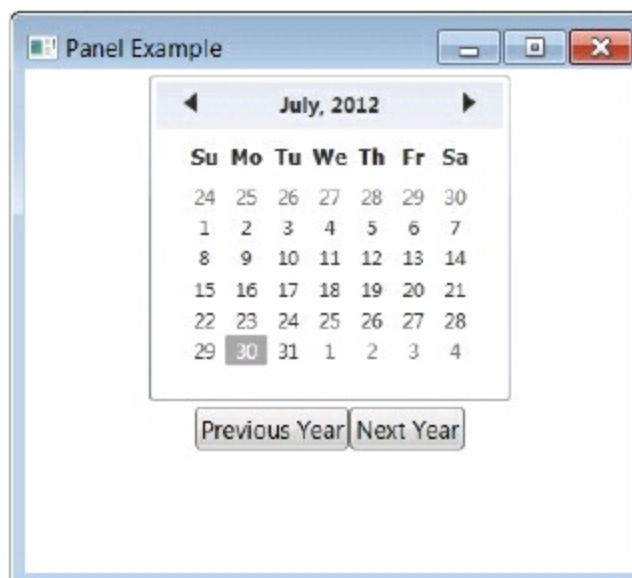


图15-4 面板示例（扩大）

Previous Year和Next Year按钮用来允许用户更改年份。但是，当点击其中之一时，却什么都没有发生。这是因为我们没有告诉引用程序如何做出响应。当事件发生后（例如用户点击一个按钮），要做一些事情，我们需要编写一个事件处理程序。接下来我们将讨论这个话题。

15.5 事件处理

事件处理是一种编程范型，其中，当一个事件发生时，会执行一套指令。这是WPF中最强大的功能之一，因为没有事件处理能力，就什么也实现不了。

程序清单15-4中的类展示了如何编写一个事件处理方法，以及如何把它和一个事件链接起来。

程序清单15-4 事件处理

```
using System;
using System.Windows;
using System.Windows.Controls;

namespace App15
{
    class EventExample : Application
    {
        //WPF应用程序应该执行一个单线程单元（STA）
        [STAThread]
        static void Main(string[] args)
        {
            EventExample app = new EventExample();
            app.Run();
        }

        protected override void OnStartup(StartupEventArgs e)
        {
            base.OnStartup(e);
            EventWindow window = new EventWindow();
            window.Title = "Event Example";
            window.Height = 228;
            window.Width = 200;
        }
    }
}
```

```

        window.WindowStartupLocation =
            WindowStartupLocation.CenterScreen;
        window.Show();
    }
}

class EventWindow : Window
{
    Calendar calendar;
    public EventWindow()
    {
        StackPanel mainPanel = new StackPanel();
        calendar = new Calendar();
        Button button1 = new Button();
        button1.Content = "Highlight";
        Button button2 = new Button();
        button2.Content = "No Highlight";
        StackPanel buttonPanel = new StackPanel();
        buttonPanel.Orientation = Orientation.Horizontal;
        buttonPanel.HorizontalAlignment =
            System.Windows.HorizontalAlignment.Center;
        buttonPanel.Children.Add(button1);
        buttonPanel.Children.Add(button2);

        mainPanel.Children.Add(calendar);
        mainPanel.Children.Add(buttonPanel);

        button1.Click += OnClick1;
        button2.Click += OnClick2;
        this.Content = mainPanel;
    }

    void OnClick1(object sender, RoutedEventArgs e)
    {
        calendar.IsTodayHighlighted = true;
    }
    void OnClick2(object sender, RoutedEventArgs e)
    {
        calendar.IsTodayHighlighted = false;
    }
}
}

```

我们要特别注意粗体的代码行。

```
button1.Click += OnClick1;
```

```
button2.Click += OnClick2;
```

这些代码行主要是通知WPF，button1的Click事件要链接到OnClick1，button2的Click事件要链接到OnClick2。OnClick1和OnClick2都是事件处理程序，而且改变了Calendar控件的IsTodayHighlighted属性的值。

```
void OnClick1(object sender, RoutedEventArgs e)
{
    calendar.IsTodayHighlighted = true;
}
void OnClick2(object sender, RoutedEventArgs e)
{
    calendar.IsTodayHighlighted = false;
}
```

我们运行程序清单15-4中的代码，会显示一个带日历和两个按钮的窗口，如图15-5所示。这两个按钮现在是可用的，因为它们链接到了事件处理程序。



图15-5 带事件处理程序的一个WPF应用程序

15.6 XAML

XAML表示可扩展应用标记语言（eXtensible Application Markup Language）。最初是为了使用WPF而创建的，XAML可以用于任何层级式数据类型。

当用Visual Studio Express创建一个WPF项目时，会创建两个.xaml文件。第1个文件是MainWindow.xaml，它描述WPF应用程序中的main窗口，如下所示。

```
<Window x:Class="XamlExample1.MainWindow"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="MainWindow" Height="350" Width="525">

  <Grid>

  </Grid>
</Window>
```

默认情况下，使用Grid作为所有控件的主面板。但是，我们可以很容易地修改它。

一个WPF项目开始时，会创建第2个XAML文件**App.xaml**，如下所示。

```
<Application x:Class="XamlExample1.App"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  StartupUri="MainWindow.xaml">

  <Application.Resources>

  </Application.Resources>
</Application>
```

这个文件描述了WPF应用程序。要注意，这个Application元素包含了一个StartupUri属性，它可以引用定义Window对象的MainWindow.xaml文件。

如果运行这个项目，我们会看到一个空白窗口。

下面的示例介绍了一个基于XAML的WPF应用程序。程序清单15-5展示了它的MainWindow.xaml文件，程序清单15-6展示了它的App.xaml文件。

程序清单15-5 MainWindow.xaml文件

```

<Window x:Class="XamlExample2.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MainWindow" Height="350" Width="525">
    <DockPanel LastChildFill="True">
        <Label Foreground="Green" Background="Bisque"
            DockPanel.Dock="Top"
            Height="100"
            HorizontalContentAlignment="Center">Top</Label>
        <Label Foreground="Red" Background="Beige"
            DockPanel.Dock="Left"
            Width="50"
            VerticalContentAlignment="Center">Left</Label>
        <TextBlock Foreground="Blue" Background="AliceBlue"
            DockPanel.Dock="Right">Right</TextBlock>
    </DockPanel>
</Window>

```

程序清单15-6 App.xaml文件

```

<Application x:Class="XamlExample2.App"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        StartupUri="MainWindow.xaml">

    <Application.Resources>

    </Application.Resources>
</Application>

```

我们从XAML示例中可以看出，这个应用程序没有包含一行代码。我们所要做的只是配置了这个应用程序。如果不使用XAML来编写，这个应用程序的代码如程序清单15-7所示。

程序清单15-7 相当于XAML示例的代码

```

using System;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Media;

namespace XamlExample2
{

```

```

class CodeOnly : Application
{
    //WPF应用程序应该执行一个单线程单元 (STA)
    [STAThread]
    static void Main(string[] args)
    {
        CodeOnly app = new CodeOnly();
        app.Run();
    }

    protected override void OnStartup(StartupEventArgs e)
    {
        base.OnStartup(e);
        Window window = new Window();
        window.Title = "CodeOnly Equivalent";
        window.Height = 350;
        window.Width = 525;

        DockPanel dockPanel = new DockPanel();
        dockPanel.LastChildFill = true;

        Label topElement = new Label();
        topElement.Content = "Top";
        DockPanel.SetDock(topElement, Dock.Top);
        topElement.Foreground =
            new SolidColorBrush(Colors.Green);
        topElement.Background =
            new SolidColorBrush(Colors.Bisque);
        topElement.Height = 100;
        topElement.HorizontalContentAlignment =
            HorizontalAlignment.Center;

        Label leftElement = new Label();
        leftElement.Content = "Left";
        leftElement.Foreground =
            new SolidColorBrush(Colors.Red);
        leftElement.Background =
            new SolidColorBrush(Colors.Beige);
        leftElement.Width = 50;
        leftElement.VerticalContentAlignment =
            VerticalAlignment.Center;
        DockPanel.SetDock(leftElement, Dock.Left);

        TextBox rightElement = new TextBox();
        rightElement.Text = "Right";
        rightElement.Foreground =
            new SolidColorBrush(Colors.Blue);
    }
}

```

```
        rightElement.Background =  
            new SolidColorBrush(Colors.AliceBlue);  
        DockPanel.SetDock(rightElement, Dock.Right);  
  
        dockPanel.Children.Add(topElement);  
        dockPanel.Children.Add(leftElement);  
        dockPanel.Children.Add(rightElement);  
  
        window.Content = dockPanel;  
        window.Show();  
    }  
}
```

15.7 小结

WPF是用于开发桌面应用程序的、最新的.NET技术。在本章中，我们介绍了WPF中的两个主要的类Application和Window以及用到的一些控件。我们还介绍了通过编写代码以及使用XML和代码来编写WPF应用程序。

第16章 多态

对于面向对象编程（OOP）的初学者，多态是最难解释的概念。事实上，大多数时候，如果没有一两个示例的说明，其定义是很难理解的。好了，我们来试着理解一下。下面是很多编程书籍中给出的定义：“多态是一种OOP特性，当接收到一个方法调用时，它能让一个对象来决定调用哪一个方法实现。”如果你觉得这些内容很难理解，这并不奇怪，因为很多人都和你一样。多态很难用简单的语言来解释，但是如果通过一两个示例，我们就会很容易理解它。

本章从一个简单的示例开始介绍，来使多态的概念清晰起来，然后用另一个示例来展示多态在一个简单的绘图应用程序中的应用。

注意

在其他编程语言中，多态也叫作后绑定、运行时绑定或动态绑定。

16.1 定义多态

在C#或其他OOP语言中，如果满足某些条件，那么把一个对象赋予一个引用变量并且它的类型和变量的类型不同，这种情况是合法的。事实上，如果有一个引用变量的类型是A，把一个类型是B的对象赋值给它，这样是合法的，如下所示。

```
A a = new B();
```

这需要满足以下两个条件之一。

- A是一个类，B是A的子类。
- A是一个接口，B或它的一个父类实现了A。

正如我们第6章介绍过的，这叫作向上转型。

在前面的代码中，当我们把B的一个实例赋值给a的时候，a的类型是A。这意味着，如果一个方法在A中没有定义的话，我们不能调用B中

的这个方法。但是，如果打印GetType().ToString()的值，我们会得到“B”而不是“A”。那么，这是什么意思？在编译时，的类型是A，所以编译器不允许调用B中那些没有在A中定义的方法。另一方面，在运行时a的类型是B，a.GetType().ToString()的返回值可以证实这一点。

现在，我们接触到了多态的本质。如果B覆盖A中的一个方法（假设一个名为Play的方法），调用Play()会导致调用B中的Play的实现。在调用一个方法时，多态允许一个对象（在本例中，是a所引用的对象）决定选择哪个方法实现（A中的方法或者B中的方法）。多态在运行时告诉对象该调用哪个实现。

如果我们调用a中的另一个方法（假设是名为Stop的方法），而该方法没有在B中实现，那该怎么办？CLR足够智能，能够知道这一点并且调查B的继承层级。这种情况下，要么B必须是A的子类，要么A是接口、而B必须是实现了A的另一个类的一个子类。否则，代码不会编译。这些都清楚后，CLR会沿着类的层级向上攀升，找到Stop的实现并且去执行它。

现在，多态有一个更有意义的定义：多态是一种OOP特性，当接收到一个方法调用时，它允许一个对象来确定选择哪个方法来实现调用。

然而，从技术角度来看，C#是如何实现这些的呢？实际上，在遇到调用诸如Play()这样的方法时，C#编译器要查看a所表示的接口和类是否定义了这样的一个方法（Play方法）以及是否给该方法传递了一组正确的参数。但是，编译器所能够做的事情也仅限于此。静态方法和密封方法例外，它们不需要把一个方法调用和方法体连接或绑定在一起。CLR在运行时确定如何将方法调用绑定到方法体。

换句话说讲，除了静态方法和密封方法以外，C#中的方法绑定会在运行时发生，而不是在编译时发生。运行时绑定也叫作延迟绑定或动态绑定。相反的是早期绑定，即在编译时或连接时进行绑定。早期绑定会出现在像C语言这样的语言中。

因此，在.NET Framework中通过延迟绑定机制，多态成为可能。也正由于此，不太准确地讲，多态也叫作延迟绑定、动态绑定或运行时绑定。

我们来看看程序清单16-1中的代码。

```
using System;

namespace PolymorphismExample1
{
    class Employee {
        public virtual void Work()
        {
            Console.WriteLine("I am an employee.");
        }
    }

    class Manager : Employee
    {
        public override void Work()
        {
            Console.WriteLine("I am a manager.");
        }

        public void Manage()
        {
            Console.WriteLine("Managing ...");
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            Employee employee;
            employee = new Manager();
            Console.WriteLine(employee.GetType().ToString());
            employee.Work();
            Manager manager = (Manager) employee;
            manager.Manage();

            Console.ReadKey();
        }
    }
}
```

程序清单16-1定义了两个类：Employee和Manager。Employee有一个叫作Work的虚方法；Manager扩展自Employee并且增加了一个名为Manage的新方法，它还覆盖了Work方法。

Program类中的Main方法定义了一个名为**employee**的对象变量，其类型为Employee:

```
Employee employee;
```

但是，这里给employee赋了一个Manager的实例，如下所示。

```
employee = new Manager();
```

这是合法的，因为Manager是Employee的一个子类，所以Manager“是一个”Employee。给employee赋了一个Manager实例，那么employee.GetType().ToString()会得到什么样的结果呢？没错，是“Manager”，而不是“Employee”。

然后，调用work方法，如下所示。

```
employee.Work();
```

猜猜控制台的结果是什么？

```
I am a manager.
```

这就意味着，调用的是Manager类中的Work方法，这就是多态在起作用。

现在，由于a的运行时类型是Manager，我们可以把a向下转换为Manager，代码如下所示。

```
Manager manager = (Manager) employee;  
manager.Manage();
```

看完这段代码你可能会问，为什么不先把employee声明为Employee？为什么不像下面这样把employee声明为Manager类型呢？

```
Manager employee;  
employee = new Manager();
```

之所以这么做，是为了保证灵活性，因为我们不知道什么时候一个

Manager的实例或别的东西可能会赋值给这个对象引用（employee）。我们继续看下面的示例，多态这种技术就会变得清晰起来。

16.2 多态的应用

假设有一个WPF应用程序并且有一个MakeMoreTransparent的方法，它能够改变一个UIElement的透明度。我们想能够给这个方法传递任何的WPF控件和面板。因此，我们需要让这个方法能够接受一个UIElement，它是WPF中的所有UI元素的基类。该方法如程序清单16-2所示。

程序清单16-2 MakeMoreTransparent方法

```
void MakeMoreTransparent(UIElement uiElement)
{
    double opacity = uiElement.Opacity;
    if (opacity > 0.2)
    {
        uiElement.Opacity = opacity - 0.1;
    }
}
```

好在有多态，MakeMoreTransparent将接受UIElement的一个实例或UIElement的一个子类的实例。程序清单16-3展示了这个完整的WPF应用程序，它是带一个日历和三个按钮的一个窗口。点击第三个按钮，将会遍历窗口的所有子控件并把每一个子控件传递给MakeMoreTransparent。

程序清单16-3 使用了多态的一个WPF应用程序

```
using System;
using System.Windows;
using System.Windows.Controls;

namespace PolymorphismExample2
{
    class PolymorphismExample2 : Application
    {
        //WPF应用程序应该执行一个单线程单元（STA）
        [STAThread]
    }
}
```

```

static void Main(string[] args)
{
    PolymorphismExample2 app = new PolymorphismExample2();
    app.Run();
}

protected override void OnStartup(StartupEventArgs e)
{
    base.OnStartup(e);
    EventWindow window = new EventWindow();
    window.Title = "Polymorphism Example";
    window.Height = 228;
    window.Width = 300;

    window.WindowStartupLocation =
        WindowStartupLocation.CenterScreen;
    window.Show();
}
}

class EventWindow : Window
{
    Calendar calendar;
    public EventWindow()
    {
        StackPanel mainPanel = new StackPanel();
        calendar = new Calendar();
        Button button1 = new Button();
        button1.Content = "Highlight";
        Button button2 = new Button();
        button2.Content = "No Highlight";
        Button button3 = new Button();
        button3.Content = "Make more transparent";

        StackPanel buttonPanel = new StackPanel();
        buttonPanel.Orientation = Orientation.Horizontal;
        buttonPanel.HorizontalAlignment =
            System.Windows.HorizontalAlignment.Center;
        buttonPanel.Children.Add(button1);
        buttonPanel.Children.Add(button2);
        buttonPanel.Children.Add(button3);

        mainPanel.Children.Add(calendar);
        mainPanel.Children.Add(buttonPanel);

        button1.Click += OnClick1;
        button2.Click += OnClick2;
    }
}

```

```

        button3.Click += OnClick3;

        this.Content = mainPanel;
    }

    void OnClick1(object sender, RoutedEventArgs e)
    {
        calendar.IsTodayHighlighted = true;
    }
    void OnClick2(object sender, RoutedEventArgs e)
    {
        calendar.IsTodayHighlighted = false;
    }
    void OnClick3(object sender, RoutedEventArgs e)
    {
        Panel panel = (Panel) this.Content;
        int childrenCount = panel.Children.Count;
        Console.WriteLine("start");
        foreach (UIElement child in panel.Children)
        {
            MakeMoreTransparent(child);
        }
    }

    void MakeMoreTransparent(UIElement uiElement)
    {
        double opacity = uiElement.Opacity;
        if (opacity > 0.2)
        {
            uiElement.Opacity = opacity - 0.1;
        }
    }
}
}

```

如果运行这个WPF应用程序，我们会看到如图16-1所示的窗口。不停点击右边的按钮，我们会看到多态的应用效果。

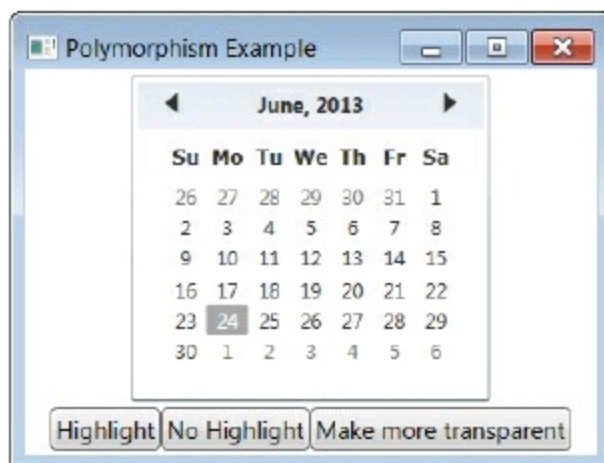


图16-1 使用多态的WPF

16.3 一个绘图程序中的多态

当程序员事先不知道要创建何种类型的对象时，多态的强大之处就显而易见了。

例如，我们来看如图16-2所示的一个简单的绘图程序。在这个程序中，我们可以绘制三种形状：长方形、直线和椭圆形。

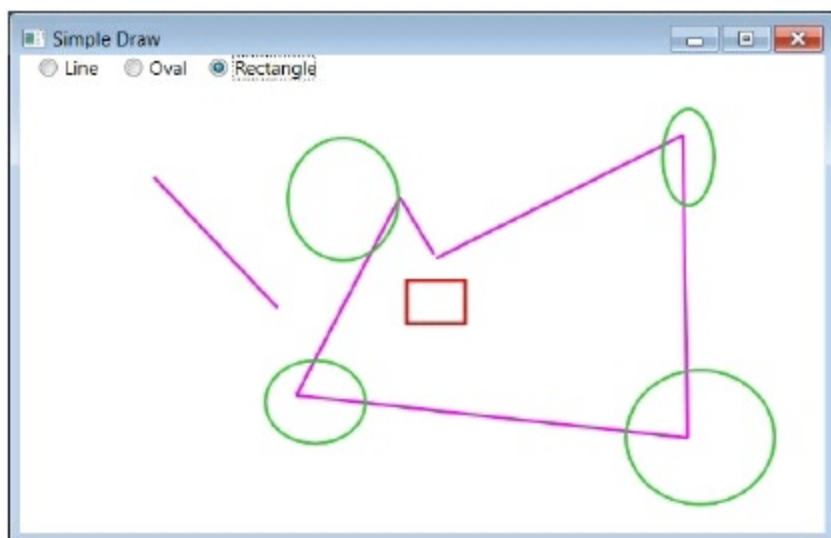


图16-2 一个简单的绘图程序

例如，要绘制一条直线，我们首先要选中Line单选按钮，然后按下

鼠标并在绘图区域拖动，最后释放鼠标按钮。绘图区域中点击的第一个位置就成为起点（x1, y1），释放鼠标按钮的坐标位置会成为终点（x2,y2）。

现在，我们来看看这个应用程序是如何工作的。

我们先来看看程序清单16-4中IShape接口及其实现类（SimpleLine、SimpleOval和SimpleRectangle）。

程序清单16-4 IShape接口及其实现

```
interface IShape
{
    void Draw(Grid myGrid)
}

public class SimpleLine : IShape
{
    double x1, y1, x2, y2;
    public SimpleLine(double x1, double y1, double x2, double y2)
    {
        this.x1 = x1;
        this.y1 = y1;
        this.x2 = x2;
        this.y2 = y2;
    }

    public void Draw(Grid grid)
    {
        //为了节省篇幅省略了方法体
    }
}

public class SimpleOval : IShape
{
    double x1, y1, x2, y2;
    public SimpleOval(double x1, double y1, double x2, double y2)
    {
        this.x1 = x1;
        this.y1 = y1;
        this.x2 = x2;
        this.y2 = y2;
    }
}
```

```

        public void Draw(Grid grid)
        {
            //为了节省篇幅省略了方法体
        }
    }

    public class SimpleRectangle : IShape
    {
        double x1, y1, x2, y2;

        public SimpleRectangle(double x1, double y1, double x2,
                                double y2)
        {
            this.x1 = x1;
            this.y1 = y1;
            this.x2 = x2;
            this.y2 = y2;
        }

        public void Draw(Grid grid)
        {
            //为了节省篇幅省略了方法体
        }
    }
}

```

IShape接口中只有Draw一个方法。这意味着，实现类要覆盖它来绘制适当的形状。例如，SimpleRectangle类中覆盖了Draw方法来绘制一个长方形。

现在，我们来看看完整的程序。程序清单16-5给出了XAML文件，程序清单16-6给出了代码。

程序清单16-5 MainWindow.xaml文件

```

<Window x:Class="SimpleDraw.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Name="idwin" Title="Simple Draw" Height="350" Width="525"
        MouseDown="canvas1_MouseDown" MouseUp="canvas1_MouseUp">

    <Grid Name="buttonGrid">
        <RadioButton Content="Line" Height="16"
            HorizontalAlignment="Left" Margin="12,0,0,0"
            Name="radioButton1" VerticalAlignment="Top"
            IsChecked="True" Checked="radioButton1_Checked" />
    
```

```

        <RadioButton Content="Oval" Height="16"
            HorizontalAlignment="Left" Margin="66,0,0,0"
            Name="radioButton2" VerticalAlignment="Top"
            Checked="radioButton2_Checked" />
        <RadioButton Content="Rectangle" Height="16"
            HorizontalAlignment="Left" Margin="120,0,0,0"
            Name="radioButton3" VerticalAlignment="Top"
            Checked="radioButton3_Checked" />
    </Grid>
</Window>

```

程序清单16-6 SimpleDraw应用程序

```

using System;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Input;
using System.Windows.Shapes;

namespace SimpleDraw
{
    /// <summary>
    /// MainWindow.xaml的互动逻辑
    /// </summary>
    public partial class MainWindow : Window
    {
        private enum ShapeType
        {
            Line, Oval, Rectangle
        }
        private Point startPoint;
        private Point endPoint;
        private ShapeType shapeType = ShapeType.Line;

        public MainWindow()
        {
            InitializeComponent();
        }

        private void canvas1_MouseDown(object sender,
            MouseButtonEventArgs e)
        {
            startPoint = e.GetPosition(this);
        }

        private void canvas1_MouseUp(object sender,
            MouseButtonEventArgs e)
        {

```

```

{
    endPoint = e.GetPosition(this);
    IShape shape = null;
    if (shapeType == ShapeType.Line)
    {
        shape = new SimpleLine(startPoint.X, startPoint.Y,
                                endPoint.X, endPoint.Y);
    }
    else if (shapeType == ShapeType.Oval)
    {
        shape = new SimpleOval(startPoint.X, startPoint.Y,
                                endPoint.X, endPoint.Y);
    }
    else if (shapeType == ShapeType.Rectangle)
    {
        shape = new SimpleRectangle(startPoint.X,
                                     startPoint.Y, endPoint.X, endPoint.Y);
    }

    if (shape != null)
    {
        Grid grid = (Grid) this.FindName("buttonGrid");
        shape.Draw(grid);
    }
}

private void radioButton1_Checked(object sender,
                                   RoutedEventArgs e)
{
    shapeType = ShapeType.Line;
}

private void radioButton2_Checked(object sender,
                                   RoutedEventArgs e)
{
    shapeType = ShapeType.Oval;
}

private void radioButton3_Checked(object sender,
                                   RoutedEventArgs e)
{
    shapeType = ShapeType.Rectangle;
}
}

interface IShape
{

```



```

        void Draw(Grid myGrid);
    }

    public class SimpleLine : IShape
    {
        double x1, y1, x2, y2;

        public SimpleLine(double x1, double y1, double x2,
                           double y2)
        {
            this.x1 = x1;
            this.y1 = y1;
            this.x2 = x2;
            this.y2 = y2;
        }

        public void Draw(Grid grid)
        {
            Line windowLine = new Line();
            windowLine.X1 = x1;
            windowLine.Y1 = y1;
            windowLine.X2 = x2;
            windowLine.Y2 = y2;
            windowLine.Stroke =
                System.Windows.Media.Brushes.Magenta;
            windowLine.StrokeThickness = 2;
            grid.Children.Add(windowLine);
        }
    }

    public class SimpleOval : IShape
    {
        double x1, y1, x2, y2;

        public SimpleOval(double x1, double y1, double x2,
                           double y2)
        {
            this.x1 = x1;
            this.y1 = y1;
            this.x2 = x2;
            this.y2 = y2;
        }

        public void Draw(Grid grid)
        {
            double x, y;
            Ellipse oval = new Ellipse();

```

```

        Thickness margin;
        if (x1 < x2)
        {
            x = x1;
        }
        else
        {
            x = x2;
        }
        if (y1 < y2)
        {
            y = y1;
        }
        else
        {
            y = y2;
        }
        margin = new Thickness(x, y, 0, 0);
        oval.Margin = margin;
        oval.HorizontalAlignment = HorizontalAlignment.Left;
        oval.VerticalAlignment = VerticalAlignment.Top;
        oval.Stroke = System.Windows.Media.Brushes.LimeGreen;
        oval.Width = Math.Abs(x2 - x1);
        oval.Height = Math.Abs(y2 - y1);
        Canvas.SetLeft(oval, x);
        Canvas.SetTop(oval, y);
        oval.StrokeThickness = 2;
        grid.Children.Add(oval);
    }
}

public class SimpleRectangle : IShape
{
    double x1, y1, x2, y2;

    public SimpleRectangle(double x1, double y1, double x2,
        double y2)
    {
        this.x1 = x1;
        this.y1 = y1;
        this.x2 = x2;
        this.y2 = y2;
    }

    public void Draw(Grid grid)
    {
        double x, y;
    }
}

```

```

        Rectangle rect = new Rectangle();
        if (x1 < x2)
        {
            x = x1;
        }
        else
        {
            x = x2;
        }
        if (y1 < y2)
        {
            y = y1;
        }
        else
        {
            y = y2;
        }
        Thickness margin = new Thickness(x, y, 0, 0);
        rect.Margin = margin;
        rect.HorizontalAlignment = HorizontalAlignment.Left;
        rect.VerticalAlignment = VerticalAlignment.Top;
        rect.Stroke = System.Windows.Media.Brushes.Red;
        rect.Width = Math.Abs(x2 - x1);
        rect.Height = Math.Abs(y2 - y1);
        Canvas.SetLeft(rect, x);
        Canvas.SetTop(rect, y);
        rect.StrokeThickness = 2;
        grid.Children.Add(rect);
    }
}
}

```

MainWindow类是System.Windows.Window的一个子类，它用一个Grid来包含供用户选择绘制图形的三个RadioButton控件。MainWindow也有几个类变量。首先是两个System.Windows.Point，即startPoint和endPoint。startPoint表示在绘图区域按下鼠标的坐标位置，endPoint表示释放鼠标的坐标位置。然后，这里有一个shapeType，这是ShapeType类型的一个引用，还有一个带三个成员（Line、Oval和Rectangle）的枚举类型。shapeType表示用户选中的形状。每一次用户点击一个不同的单选按钮时，它的值都会改变。

我们要特别注意**canvas1_mouseUp**事件处理程序。

```
private void canvas1_MouseUp(object sender,
```

```

        MouseButtonEventArgs e)
    {
        endPoint = e.GetPosition(this);
        IShape shape = null;
        if (shapeType == ShapeType.Line)
        {
            shape = new SimpleLine(startPoint.X, startPoint.Y,
                                   endPoint.X, endPoint.Y);
        }
        else if (shapeType == ShapeType.Oval)
        {
            shape = new SimpleOval(startPoint.X, startPoint.Y,
                                   endPoint.X, endPoint.Y);
        }
        else if (shapeType == ShapeType.Rectangle)
        {
            shape = new SimpleRectangle(startPoint.X,
                                        startPoint.Y, endPoint.X, endPoint.Y);
        }

        if (shape != null)
        {
            Grid grid = (Grid) this.FindName("buttonGrid");
            shape.Draw(grid);
        }
    }
}

```

多态会在这里发生。首先，这个事件处理程序获取了终点，即用户释放鼠标的位置。然后，它声明了一个IShape类型的shape变量。这里根据shapeType的值来给shape赋以对象。在这个示例中我们可以看到，在编写类时，并不知道会生成哪个对象；在编译时，也不知道。

最后，该事件处理程序调用shape的Draw方法，它给对象在Grid上绘制自己的机会。

16.4 小结

多态是面向对象编程的主要支柱之一。在编译时不知道对象类型的情况下，多态很有用。本章通过几个示例介绍了多态。

第17章 ADO.NET

ADO.NET是.NET中用来访问和操作不同格式数据的技术，包括关系数据库中的数据或XML中的数据。ADO.NET的名称来源于ADO（访问数据对象，Access Data Objects），这是微软的一项旧技术，它的功能类似于ADO.NET。尽管名称相似，但是ADO.NET的架构和ADO的相似度却很少。

从.NET 1.0开始，ADO.NET就已经易于供任何.NET语言的开发者（包括C#程序员）使用了。关于访问关系型数据库，ADO.NET的优势在于，它提供了一种统一的方法来访问不同的关系型数据库。不同的数据库服务器使用不同的专有协议，如果不用ADO.NET（或类似的技术，诸如Java Database Connectivity）访问数据库，我们可能需要编写完全不同的代码。对于ADO.NET支持的每一种关系型数据库，有一组类能够与数据库服务器进行通信，这组类叫作数据提供者。

本章介绍了ADO.NET的基本特性，特别是那些访问和操作关系型数据库中的数据的特性。这里假设我们已经具备了SQL的基本知识。

17.1 介绍ADO.NET

ADO.NET是由System.Data和System.Data.Common命名空间中的许多类型组成的，所有这些类型都能够在System.Data.dll程序集中找到。表17-1中的树型抽象类是System.Data.Common的成员，而且是最重要的ADO.NET类型。

表17-1 最重要的ADO.NET类型

类	说 明
DbConnection	表示到数据库的一个连接
DbCommand	表示一条SQL语句或一个存储过程

DbDataReader	表示一个数据读取器，能够读取单向的行/列的流数据
--------------	--------------------------

表17-1中的每一个抽象类分别实现了一个接口，其中System.Data.DbConnection实现了System.Data.IDbConnection，DbCommand实现了System.Data.IDbCommand，而DbDataReader实现了System.Data.IDbDataReader。

因为表17-1中的三种类型都是抽象类，所以要真正使用这些类型，就必须要实现它们。实现是以数据提供者的形式做到的。事实上，每一种DBMS都需要一个不同的数据提供者。也就是说，SQL Server数据库需要一个和访问Sybase数据库不同的数据提供者。但是，所有的数据提供者都实现了同样的核心ADO.NET类型，因此我们可以用统一的方法来访问不同的数据库。换句话说，我们可以用同样的方法访问和操作SQL Server数据库和Oracle数据库。

每一种数据库需要一个不同的数据提供者。.NET类库带有一个用于SQL Server的数据提供者、一个用于Oracle的数据提供者以及通过OLE DB和ODBC这样较旧的技术来间接地访问数据库的数据提供者。

而现在，市场上的每一种数据库服务器至少都有一个数据提供者。因为.NET如此热门，任务数据库厂商都有兴趣提供他们产品的数据提供者。然而，数据提供者也有可能来自于并不生产数据库服务器的第三方。

对于流行的数据库，即使有多个数据提供者也不稀罕。没有相应的数据提供者的数据库经常要通过OLE DB或ODBC来使用。例如，Microsoft Access数据库就没有数据提供者，但是我们可以通过OLE DB来访问它们。

表17-2展示了.NET Framework中的数据提供者。除了表17-2中所介绍的，还有针对实体数据模型（Entity Data Model，EDM）应用程序的数据提供者。但是这超越了本书所讨论的范畴，所以没有包含在表中。

表17-2 .NET Framework中包含的数据提供者

--	--

数据提供者	说 明
支持SQL Server的数据提供者	在System.Data.SqlClient命名空间中，针对Microsoft SQL Server 7.0及其以后的版本
支持OLE DB的数据提供者	在System.Data.OleDb命名空间中，针对OLE DB
支持ODBC的数据提供者	在System.Data.Odbc命名空间中，针对ODBC
支持Oracle的数据提供者	在System.Data.OracleClient命名空间中，针对Oracle，使用来自Oracle的数据提供者替代它

请注意，通过OLE DB或ODBC访问数据库是间接访问，它通常要比使用数据提供者慢。因此，在使用OLE DB或ODBC之前，我们通常应该先尝试为选定的数据库找一个数据提供者。下面的网址给出了一个不是由微软支持的数据提供者的列表。

<http://msdn.microsoft.com/en-us/data/dd363565.aspx>

注意，要访问Oracle数据库，我们应该使用来自Oracle的数据提供者，而不是.NET类库中的数据提供者。

17.2 访问数据的5个步骤

通过ADO.NET访问数据库和操作数据需要五个步骤。

(1) 安装想要访问数据库的数据提供者软件，除非该数据提供者已经包含在.NET类库中。

(2) 获取一个数据库连接。

(3) 创建一个Command对象以表示一条SQL语句。

(4) 可选地创建一个数据读取器，用来从数据库中读取数据。

(5) 关闭ADO.NET对象来释放资源。好在有using语句，所以我们不一定要手动来做这些。

后面的小节会详细介绍这些步骤。

17.2.1 安装数据提供者

数据提供者是第三方部署的一组类。如果我们使用Microsoft SQL Server 7.0或之后的版本，或者通过OLE DB或ODBC访问数据库，那么可以跳过这一步。如果使用的不是SQL Server数据库，也没有用OLE DB或ODBC，那么在通过C#代码访问这个数据库之前，我们必须下载并且安装一个数据提供者。例如，假设我们要连接MySQL数据库，就要从<http://www.mysql.com/products/connector>下载支持MySQL的ADO.NET驱动程序（数据提供者的名称就是ADO.NET driver for MySQL）。数据提供者可以是一个MSI安装包或是一个ZIP文件。

图17-1展示了MySQL数据提供者的安装向导。

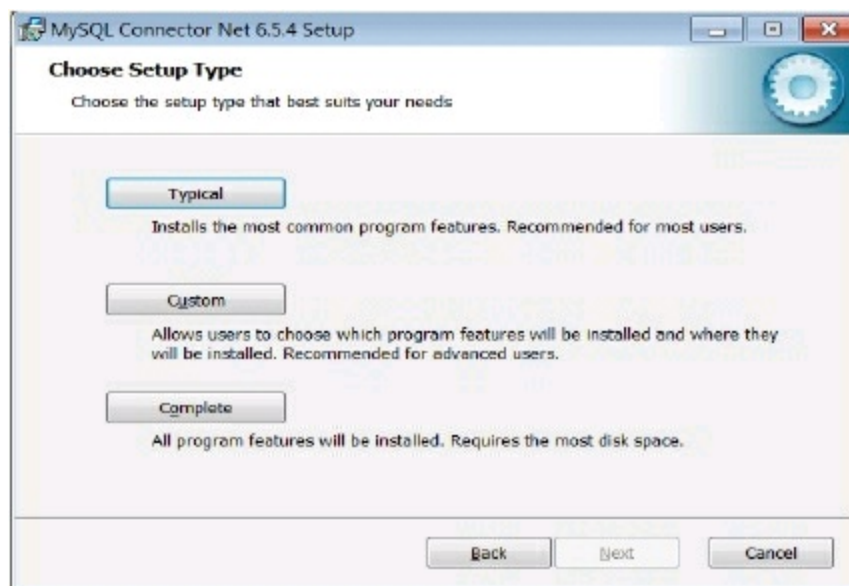


图17-1 安装MySQL的ADO.NET数据提供者

安装好这个数据提供者之后，我们就可以在Visual Studio 2010 Express或其他IDE中添加对它的引用。要添加一个引用，首先在项目图

标上点击鼠标右键，选中“Add Reference”。其次在弹出的窗口中，点击.NET标签页，滚动并从窗口中选择这个程序集。针对MySQL的三个程序集如图17-2所示。

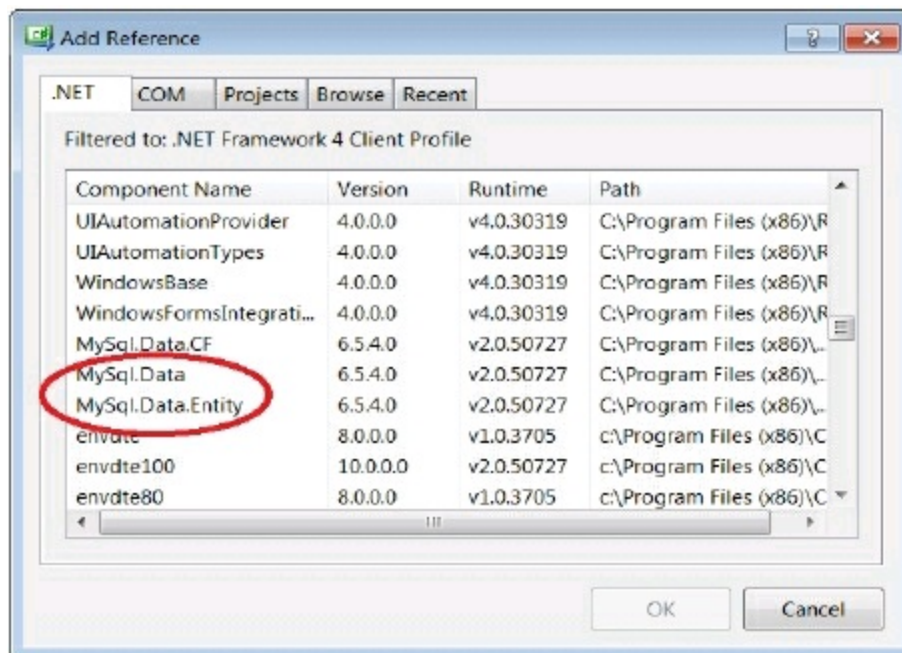


图17-2 MySQL数据提供者的引用

不要选择MySql.Data.CF，因为它包含了和MySql.Data类似的类，而且如果两个都选的话会产生冲突。如果使用.NET Compact Framework（本书不作讨论），那么我们只使用MySql.Data.CF即可。

17.2.2 获取一个数据库连接

数据库连接促进了代码和关系型数据库之间的通信。
`System.Data.Common.DbConnection`抽象类是连接对象的一个模板。我们通常会调用一个实现类的构造函数并且给它传递一个连接字符串。我们需要知道数据提供者中的实现类。类的名称是特定于数据提供者的。例如，在SQL Server数据提供者中，这个扩展自`DbConnection`的类是`System.Data.SqlClient.SqlConnection`。因此，要生成访问SQL Server的一个连接对象，我们可以编写如下的代码。

```
SqlConnection connection = new SqlConnection(connectionString);
```

另一方面，在MySQL数据提供者中，我们用 `MySql.Data.MySqlClient.MySqlConnection` 类表示一个连接。因此，要创建一个连接对象，我们可以使用如下的代码。

```
MySqlConnection connection = new MySqlConnection(connectionString);
```

既然实现类都是从 `DbConnection` 中派生而来，那么给 `DbConnection` 引用变量赋一个连接对象是可能的，如下所示。

```
DbConnection connection = new SqlConnection(connectionString);
```

或者如下所示。

```
DbConnection connection = new MySqlConnection(connectionString);
```

但是，在构造 `DbCommand` 对象时，这种方法可能不好用。因为针对 `DbCommand` 实现的构造函数需要一个特定类型的 `DbConnection`。例如，如果 `connection` 是 `DbConnection` 类型的，程序就无法编译。因为 `SqlCommand` 类的构造函数的第二个参数必须是 `SqlConnection` 类型的。

```
SqlCommand command = new SqlCommand(sql, connection);
```

因此，我们通常会将一个 `connection` 对象赋予一个特定的类型变量。

```
SqlConnection connection = new SqlConnection(connectionString);  
MySqlConnection connection = new MySqlConnection(connectionString);
```

棘手的是建立一个正确的连接字符串。通常，我们需要知道要访问的数据库的类型、服务器的位置（主机或IP地址）以及数据库的用户名和密码（可选的）。然后，我们必须构建由键值对组成的一个字符串，如下所示。

```
key-1=value-1; key-2=value-2; ...; key-n=value-n
```

每对键值对都用分号隔开，分号后边的空格是可选的。

如果数据库支持集成安全性，我们也可以使用Windows身份认证来访问它。

让我们先来介绍SQL Server，因为它最可能成为DBMS的首选。

SQL Server连接字符串的键可以从以下网址找到。

```
http://msdn.microsoft.com/en-us/library/system.data.sqlclient.sqlconnection.connectionstring
```

表17-3介绍了其中一些较为重要的内容。

打开一个数据库连接是数据库操作中最耗资源的操作之一。因此，ADO.NET支持连接池，这意味着，打开的连接不会被关闭而是直接返回到一个池中。连接字符串中的Pooling键应该保留其默认值，除非你有充分的理由不使用连接池。

下面的示例是一个连接字符串，它通过绑定MDF文件而打开到SQL Server数据库一个连接。它还使用了当前的Windows身份认证来验证用户。

```
Data Source=.\SQLEXPRESS;AttachDbFilename=C:\MarketingDB.mdf;
"Integrated Security=True;Connect Timeout=30
```

下面的连接字符串连接到名为PC\SQLEXPRESS的一个SQL Server并且使用MyCustomerDB数据库。

```
Persist Security Info=False;Integrated Security=true;
Initial Catalog=MyCustomerDB;Server=PC\SQLEXPRESS";
```

表17-3 SQL Server连接字符串中有效的键

键	说 明
AttachDBFilename 或 .扩展属性或 初始文件名	只支持扩展名为.mdf的数据库文件，其他文件类型都不支持

Connect Timeout 或 Connection Timeout 或 Timeout	在放弃尝试并产生错误之前，等待与服务器的连接的秒数。有效值大于或等于0，且小于或等于2147483647
Data Source或服务 器或 地址或Addr 或 网络地址	所要连接的SQL Server实例的名称或网络地址，可以在服务器名称之后指定端口号
Initial Catalog 或 Database	要使用的数据库的名称
Integrated Security 或 Trusted_Connection	指定是否使用当前的Windows身份认证进行用户的身份验证，默认值为false，意味着将使用用户ID和密码来验证用户的身份；当为true时，将使用当前的Windows身份认证进行身份验证。除了true和false，其他有效值还包括yes（与true等效）、sspi（与true等效）以及no（与false等效）
Max Pool Size	如果使用一个连接池，池中允许的最大连接数
Min Pool Size	如果使用一个连接池，池中允许的最小连接数
密码或PWD	验证用户身份的密码
Pooling	表示是否要使用连接池。有效值可以是true（默认值）、yes、false和no
用户ID 或 UID	用于验证用户身份的标识符

在生成一个连接字符串以后，我们可以创建一个DbConnection对象并且调用其Open方法来打开一个连接。下面的代码打开了到一个SQL Server实例的连接。

```
SqlConnection connection = new SqlConnection(connectionString);
```

```
try
{
    connection.Open();
    ...
}
catch (Exception e)
{
}
```

请注意，`Open`方法可能会由于各种原因而抛出一个异常，这些原因就包括用户没有得到访问该数据库的许可。另外，连接池无缝工作，我们不需要编写代码就可以享受到连接池的好处。

17.2.3 创建一个**DbCommand**对象

一个**DbCommand**对象表示一条SQL语句。要创建一个**DbCommand**对象，我们需要实例化相应的子类并且给它传递一个**DbConnection**对象。

通常，我们可以在**DbCommand**对象上调用两个方法：`ExecuteNonQuery`和`ExecuteReader`。这两个方法的签名如下。

```
public abstract int ExecuteNonQuery()
public DbDataReader ExecuteReader()
```

`ExecuteNonQuery`用于执行一条没有返回数据的SQL语句，诸如INSERT、UPDATE或DELETE语句。这个方法也适用于目录操作，如创建或删除一张表，或者查询一个数据库的结构。`ExecuteNonQuery`返回该操作所影响到的行数。

`ExecuteReader`用于执行一条Select SQL语句，而且这个方法返回一个**DbDataReader**，它允许我们读取**DbCommand**所返回的数据。

如下所示，这段代码创建了一个**SqlCommand**对象，它从一个SQL Server数据库的customers表中查询所有的行。

```
SqlCommand cmd = new SqlCommand("SELECT * FROM customers",
    connection);
```

17.2.4 创建一个DbDataReader

最后，如果有一个DbCommand对象，我们可以通过调用其ExecuteQuery方法来执行它。返回值是一个DbDataReader对象。

我们调用DbDataReader的Read方法来访问其中的数据。这个方法把这个数据读取器指向下一条记录，如果有下一条记录，就返回true；否则，返回false。

Read方法的签名如下。

```
public abstract bool Read()
```

对Read的每次调用，我们都可以使用Item属性或众多GetXXX方法中的一个来访问当前记录中的每一列。

Item属性提供了一种便利的方法来访问一个值。Item返回了System.Object这样的值，我们可以按照索引或名称来引用一个列。例如，下面的代码返回了当前记录的第一列。

```
object value = dbDataReader[0];
```

请注意，在C#中调用Item属性的方法和调用数组元素的方法相同。

下面的代码返回了当前记录中“last_name”列的值。

```
object lastName = dbDataReader["last_name"];
```

Item很方便，但是它返回的是一个对象，如果我们要进一步操作这个值，可能还需要进行转换。另一种选择是，使用DbDataReader的众多GetXXX方法中的一个方法来获取一个列的值。

```
public abstract byte GetByte(int index)
public abstract char GetChar(int index)
public abstract DateTime GetDateTime(int index)
public abstract double GetDouble(int index)
public abstract float GetFloat(int index)
public abstract short GetInt16(int index)
public abstract int GetInt32(int index)
```

```
public abstract long GetInt64(int index)
public abstract string GetString(int index)
```

例如，下面的代码调用了一个DbCommand对象的ExecuteReader方法，遍历所有的返回行并打印输出第一列的值。

```
SqlDataReader dataReader = cmd.ExecuteReader();
while (dataReader.Read())
{
    Console.WriteLine(dataReader[0]);
}
```

17.3 连接到SQL Server的示例

现在，我们已经知道了用ADO.NET访问关系型数据库的五个步骤，接下来让我们把它们放在一起来看看。程序清单17-1中的代码展示了如何访问一个SQL Server数据库中的users表。为了测试这个示例，我们需要访问一个SQL Server，它有一个名为TestDB的数据库，其中包含了一个users表。我们还需要替换连接字符串。

程序清单17-1 用ADO.NET访问SQL Server

```
using System;
using System.Data;
using System.Data.Common;
using System.Data.SqlClient;

namespace ADONETExample1
{
    class Program
    {
        static void Main(string[] args)
        {
            // 连接到数据库
            string conString = "Persist Security Info=False;" +
                               "Integrated Security=true;" +
                               "Initial Catalog=TestDB;" +
                               "Server=PC\\SQLEXPRESS";
            SqlConnection connection = new SqlConnection(conString);
            SqlDataReader dataReader = null;
            try
            {
```

```

        connection.Open();
        Console.WriteLine("Got connection");
        // 给一个command对象传递一个连接
        SqlCommand cmd = new SqlCommand(
            "SELECT * FROM users", connection);

        // 获取查询记录
        dataReader = cmd.ExecuteReader();

        // 打印每条记录的CustomerID
        while (dataReader.Read())
        {
            Console.WriteLine(dataReader[0]);
        }
    }
    catch (Exception e)
    {
        Console.WriteLine(e.Message);
    }
    finally
    {
        if (dataReader != null)
        {
            dataReader.Close();
        }

        if (connection != null)
        {
            connection.Close();
        }
    }

    Console.ReadKey();
}
}
}

```

运行这个类，将会打印输出这个数据库users表中的第一列。

17.4 小结

ADO.NET 是.NET 中访问和操作不同格式数据的技术，包括关系数据库中的数据或XML中的数据。在本章中，我们介绍了如何用三种

ADO.NET 中最常用的类型（DbConnection、DbCommand和 DbDataReader）来访问关系型数据库。

附录A Visual Studio Express 2012 for Windows Desktop

Visual Studio 2012是微软最新的集成开发环境（Integrated Development Environment, IDE），它用于开发网络、移动和桌面应用程序，版本之一是Visual Studio Express 2012。虽然不及其他Visual Studio 2012的版本功能齐全，但是只要注册成功，Express版本就是永久免费的。Express版本带有许多组件，Visual Studio Express 2012 for Windows Desktop是其中的一个组件，我们使用它来生成本书所介绍的示例。

本附录是使用Visual Studio Express 2012 for Windows Desktop的快速教程。

A.1 硬件和软件的要求

要安装Visual Studio Express 2012 for Windows Desktop，计算机的配置至少要具备1.6GHz的处理器和1GB的内存。计算机还要有5GB的可用硬盘空间。另外，如果是32位或64位的机器，它还必须有能够以1024 x 768或更高显示分辨率运行的，一块兼容DirectX 9的显卡。现在，大多数计算机都能满足这些需求。

在软件方面，计算机必须要运行以下Windows操作系统之一。

- Windows 7 SP1（x86或x64）
- Windows 8（x86或x64）
- Windows Server 2008 R2 SP1（x64）
- Windows Server 2012（x64）

A.2 下载和安装

我们可以从微软的站点免费下载Visual Studio Express 2012 for

Windows Desktop，网址如下。

`http://www.microsoft.com/visualstudio/eng/downloads#d-express-windows-desktop`

我们按照以下步骤来安装Visual Studio Express 2012 for Windows Desktop。

（1）双击我们刚下载的文件。确保要连接到网络，因为我们还需要下载一些其他的文件。然后，我们会看到如图A-1所示的对话框。这个对话框是安装向导中的一系列步骤的第一步。



图A-1 安装Visual Studio Express 2012的第一个对话框

（2）点击“I agree ...”选择对话框，同意许可证的条款和条件。然后在下方可以看到Install按钮。

(3) 点击Install按钮。

(4) 系统提问是否想运行安装程序，点击“OK”按钮。

(5) 现在，等待向导下载必要的程序并准备系统的引导。这需要一段时间，请不要走开，因为在某个步骤，安装向导会要求你重新启动计算机。

(6) 计算机备份后，该向导将恢复正常工作。继续等待。

(7) 最后，当这些都完成后，我们会看到如图A-2所示的对话框。

(8) 点击“LAUNCH”按钮。

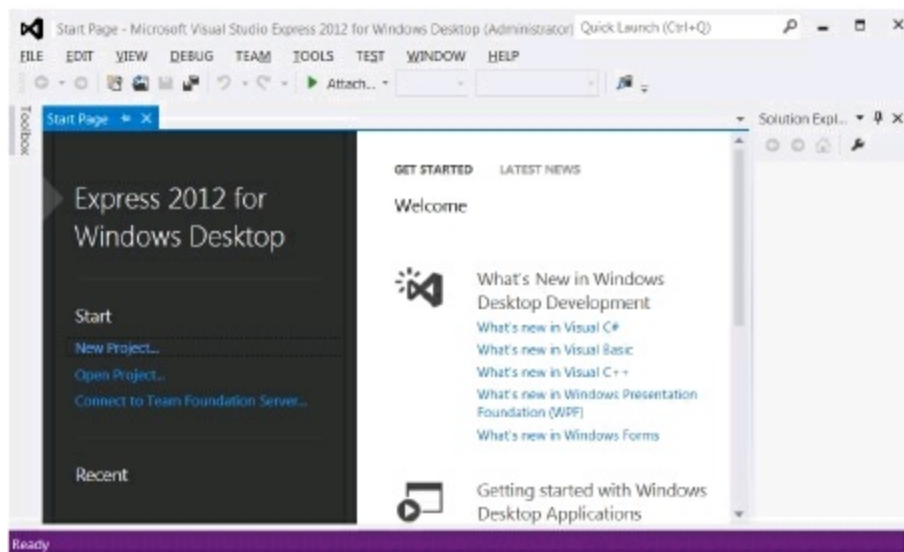


图A-2 安装Visual Studio Express 2012的最后一个对话框

我们可以使用该产品30天，到期后需要注册才能继续使用它。如果我们看到提示注册的对话框，只要点击“Cancel”按钮以启动这个软件即可。

要注册Visual Studio Express 2012，请参见A-3节。

在第一次运行时，Visual Studio Express 2012 for Windows Desktop会弹出如图A-3所示的窗口。



图A-3 开始运行Visual Studio Express 2012

恭喜你，现在系统已经为编程做好准备。

A.3 注册Visual Studio Express 2012

Visual Studio Express 2012是免费的。但是，如果我们计划使用它超过30天，需要在微软进行注册。别担心，注册是免费的。

在打开Visual Studio Express 2012，如果我们看到如图A-4所示的警告对话框，就是需要注册了。我们可以通过一个注册码来注册，如果今天没有时间，只要关闭这个对话框即可。



图A-4 注册提示

要进行注册，我们需要一个产品码，可以通过访问以下链接并点击“Register now”按钮来免费获取。

<http://www.microsoft.com/visualstudio/eng/downloads#d-2012-express>

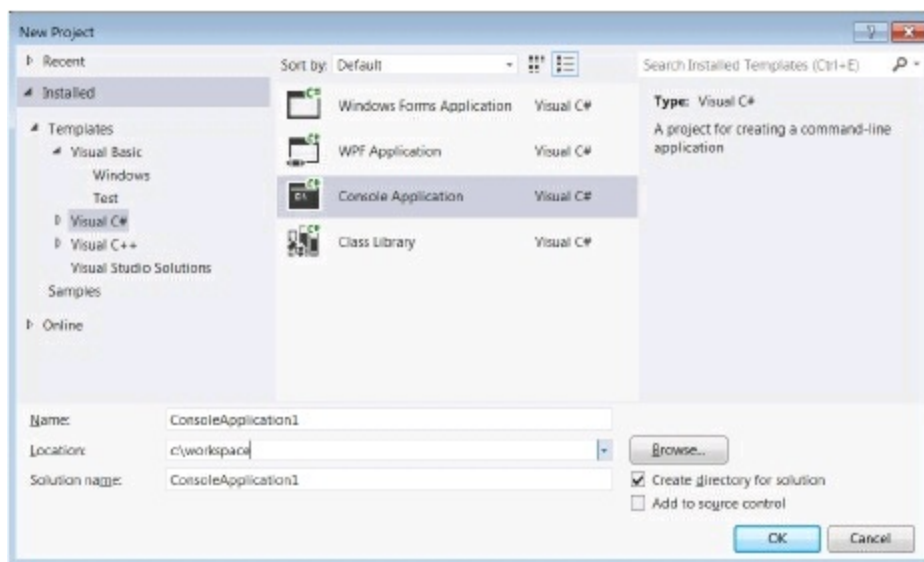
系统要求我们用微软的账户登录。如果还没有微软的账户，那么我们需要创建一个。

当获得一个产品码后，我们把它输入到图A-4中的“Product key”文本框中并且点击“Register online”按钮即可。

A.4 创建一个项目

Visual Studio Express 2012 for Windows Desktop使用项目来组织资源。因此，在生成一个C#类前，我们必须先生成一个项目。

1. 点击“File”菜单下的“New Project”，会显示“**New Project**”对话框（如图A-5所示）。

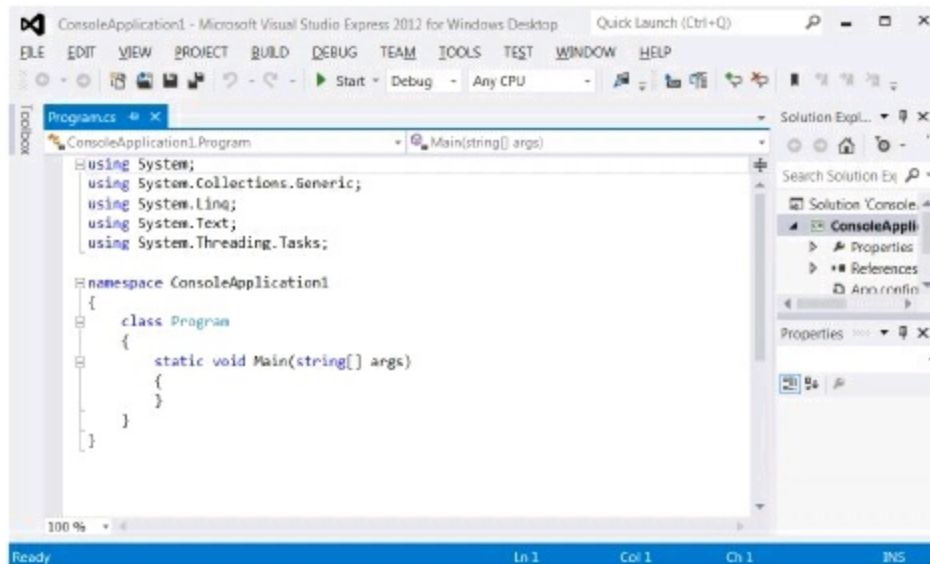


图A-5 “New Project”对话框

2. 点击目录Installed->templates下的“Visual C#”。

3. 选择一个应用程序类型。针对本书，我们需要创建Console或WPF应用程序。

4. 在“Name”文本框中输入项目名称并且找到想要保存项目的资源路径，然后，点击“OK”按钮。Visual Studio Express 2012会创建一个新的项目并在这个项目中加上第一个类，如图A-6所示。

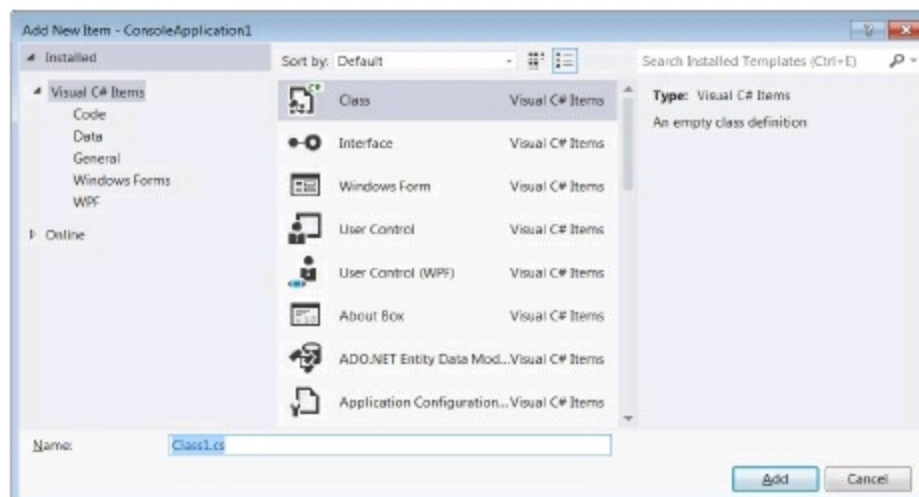


图A-6 一个C#项目

现在，我们已经为编写代码做好了准备。

A.5 创建一个类

创建一个不是Visual Studio Express 2012 for Windows Desktop默认生成的类，在Solution Explorer栏的项目名称上点击鼠标右键。图A-6中，“Solution Explorer”窗口放在了右边。点击“Add”和“Class”按钮，我们将看到如图A-7所示的“Add New Item”对话框。



图A-7 “Add New Item”对话框

在“Name”文本框中输入一个类的名称，然后点击“Class”按钮。我们也可以通过按下Shift+Alt+C快捷键来打开“Add New Item”对话框。

A.6 运行一个项目

按下F5键来运行一个项目，Visual Studio Express 2012 for Windows Desktop将会编译这个项目并且运行Main方法中的类。一个项目中只能有一个Main方法。

在运行一个项目前，Visual Studio Express 2012 for Windows Desktop会捕获所有的编译错误。

附录B Visual C# 2010 Express

Visual C# 2010 Express是C#开发者可用的开发工具之一，虽然不及Visual Studio功能完整，但是它是免费的，而Visual Studio只可以试用90天。

本附录是使用Visual C# 2010 Express的快速教程。如果你使用Windows 7或Windows 8，就要考虑使用微软最新的IDE，也就是Visual Studio Express 2012。

B.1 硬件和软件的要求

要安装Visual C# 2010 Express，计算机的配置至少要具备1.6GHz的处理器及1GB（如果是32位的机器）或2GB（如果是64位的机器）的内存。计算机还要有3GB的可用硬盘空间。

对于软件的要求，计算机必须为以下Windows操作系统之一。

- Windows XP（x86）SP3，除了Starter Edition外的所有版本。
- Windows Vista（x86或x64）SP2，除了Starter Edition外的所有版本。
- Windows 7（x86或x64）。
- Windows Server 2003（x86或x64）SP2。
- Windows Server 2003 R2（x86或x64）。
- Windows Server 2008（x86或x64）SP2。
- Windows Server 2008 R2（x64）。

B.2 下载和安装

我们可以从微软的站点免费下载Visual C# 2010 Express。

http://www.microsoft.com/visualstudio/en-us/products/2010-editions/express

在网页中向下滚动鼠标，直到看到Visual C# 2010 Express链接并且点击，然后会进入另一个页面，里面有英文版本或其他语言版本的安装链接。目前支持的语言有英语、西班牙语、意大利语、法语、德语、俄语、中文、日语和韩语。选择我们需要的语言，点击“Install Now”按钮。

此时，我们会看到一个对话框。这是微软试图吸引用户安装Visual Studio 2010 Professional版本来替代Express版本。不要动摇，坚持使用Visual C# 2010 Express并且保存这个二进制文件。

我们按照以下步骤，来安装Visual C# 2010 Express。

(1) 双击下载的文件。确保要连接到网络，因为我们还需要下载一些其他的文件。然后，我们会看到如图B-1所示的对话框。这个对话框是安装向导中的一系列步骤的第一步。

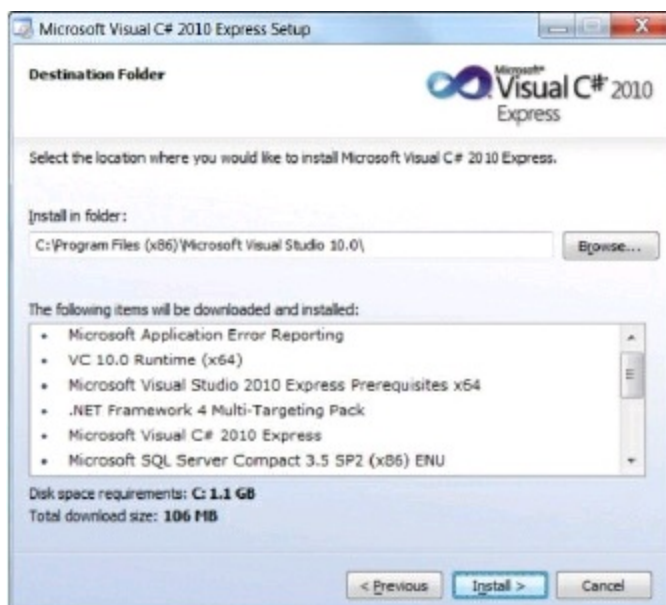


图B-1 安装向导的第一个对话框

(2) 点击“Next”按钮后，显示许可证的条款对话框。阅读条款后，点击“I have read and accept the license terms”单选框。

(3) 点击“Next”按钮。接下来的对话框会要求我们下载Microsoft SQL Server 2008 Express。如果你还没有安装，下载该产品不失为一个好主意。如果我们想要安装它，点击选择按钮；如果不想安装，就保持不选中的状态。

(4) 再点击“下一步”按钮，安装向导将会显示开始安装前的最后一个对话框。这里，我们可以选择安装文件夹并且可以看到将要下载和安装的组件列表。默认情况下，向导会尝试把它们安装到C:\Program Files\Microsoft Visual Studio 10.0目录下。我们可以选择一个不同的文件夹，如图B-2所示。



图B-2 选择一个安装文件夹

(5) 现在准备就绪了，点击“Install”按钮进行安装。接下来的对话框显示安装过程，如图B-3所示。



图B-3 安装过程

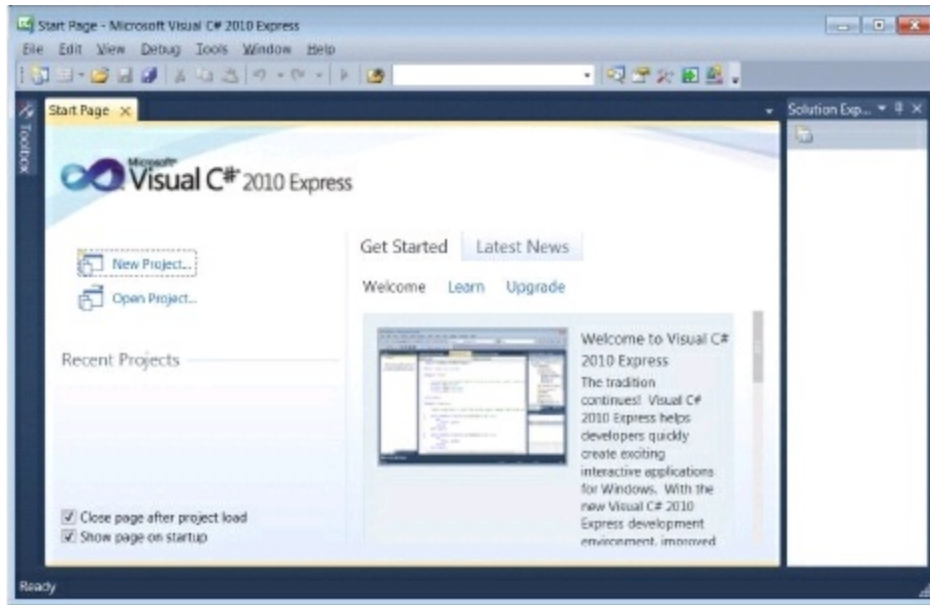
(6) 成功安装Visual C# 2010 Express后，我们会看到如图B-4所示的对话框。



图B-4 完成安装

(7) 点击Exit按钮，退出安装。

恭喜，现在我们已经可以使用Visual C# 2010 Express了。打开这个程序，我们会看到如图B-5所示的窗口。



图B-5 开始运行Visual C# 2010 Express

B.3 注册Visual C# 2010 Express

Visual C# 2010 Express是免费的。但是，如果计划使用超过30天，微软希望我们进行注册。别担心，注册是免费的。

打开Visual C# 2010 Express，如果看到如图B-6所示的警示对话框，别慌，我们可以通过一个注册码来注册；或者如果今天没有时间，我们只要关闭这个对话框即可。



B.4 创建一个项目

Visual C# 2010 Express使用项目来组织资源。因此，在生成一个C#类前，我们必须先生成一个项目，按照下面这些步骤来做。

(1) 点击“File”菜单下的“New Project”，显示“**New Project**”对话框。

(2) 点击Installed->templates目录下的“Visual C#”。

(3) 选择一个应用程序类型。针对本书，我们需要创建一个Console或WPF应用程序。

(4) 在“Name”文本框中输入项目名称并且找到想要保存项目资源的路径，然后点击“OK”按钮。Visual C# 2010 Express会创建一个新的项目并在这个项目中加上第一个类。

现在，我们已经为编写代码做好了准备。

B.5 创建一个类

要创建一个类而不是使用Visual C# 2010默认生成的类，我们需要在“Solution Explorer”窗口中的项目名称上点击鼠标右键，然后点击“Add”和“Class”按钮。我们将看到“Add New Item”对话框。

在“Name”文本框输入一个类的名称，然后点击“**Add**”按钮。我们也可以通过按下Shift+Alt+C快捷键来打开“Add New Item”对话框。

B.6 运行一个项目

按下F5键来运行一个项目，Visual C# 2010 Express将会编译这个项目并且运行带有Main方法的一个类。在运行一个项目前，Visual C# 2010 Express将捕获所有的编译错误。

附录C SQL Server 2012 Express

Microsoft SQL Server是目前最流行的关系数据库服务器，本章将介绍如何下载和安装这个广泛使用的软件的免费版本——SQL Server 2012 Express。

C.1 下载SQL Server 2012 Express

我们可以从以下站点下载SQL Server产品。

http://www.microsoft.com/en-us/sqlserver/get-sql-server/try-it.aspx

找到免费的SQL Server 2012 Express版本的下载链接，点击该产品后面的“Download”按钮并且选择32位或64位版本。

把installation.exe文件保存到硬盘。

C.2 安装SQL Server 2012 Express

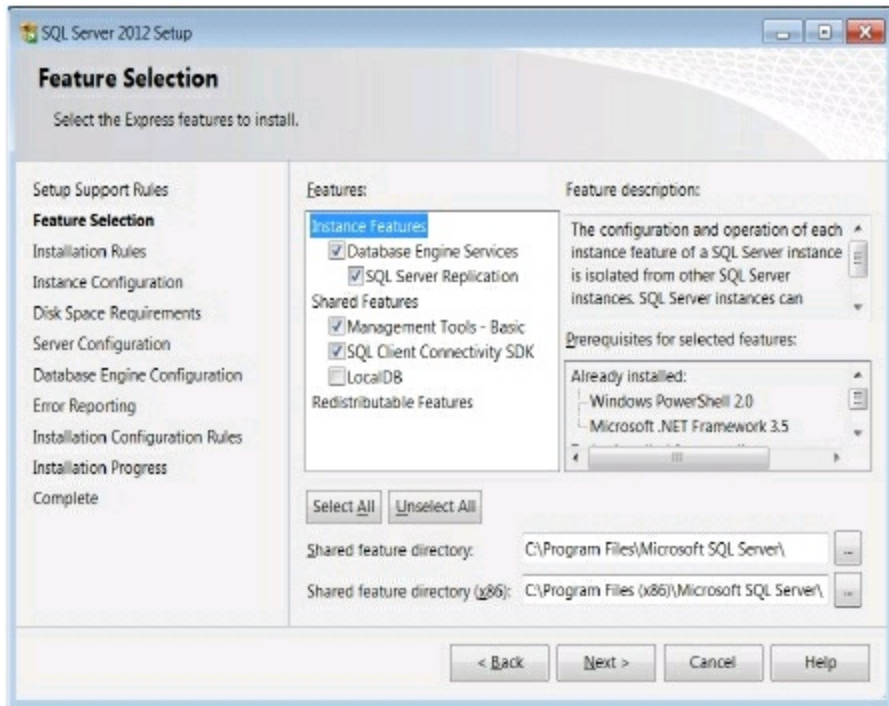
我们按照以下步骤来安装SQL Server 2012 Express。

(1) 双击下载的安装文件，会看到如图C-1所示的对话框。



图C-1 开始安装的过程

- (2) 在右边窗口点击“**New SQL Server stand-alone installation**”按钮。
- (3) 在下一个窗口选中“**I accept the license terms**”复选框。
- (4) 点击“Next”按钮打开“Feature Selection”窗口，如图C-2所示。

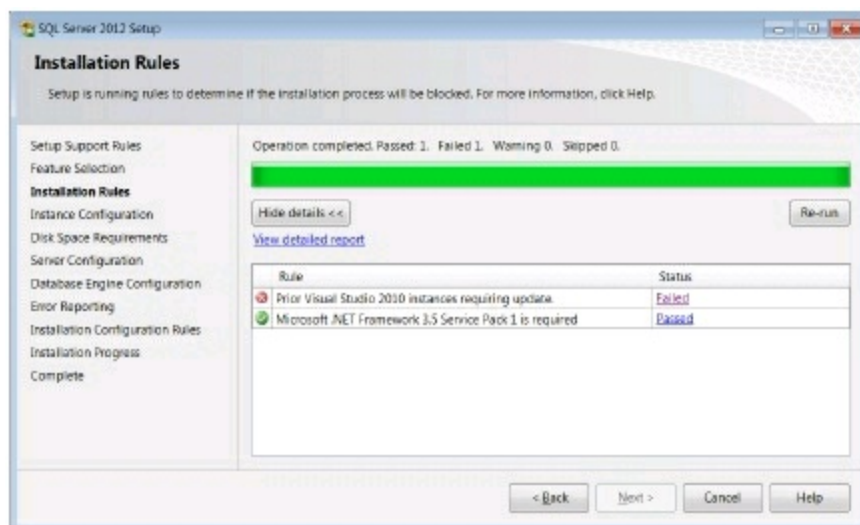


图C-2 Feature Selection窗口

(5) 接受默认选项，然后点击“Next”按钮。

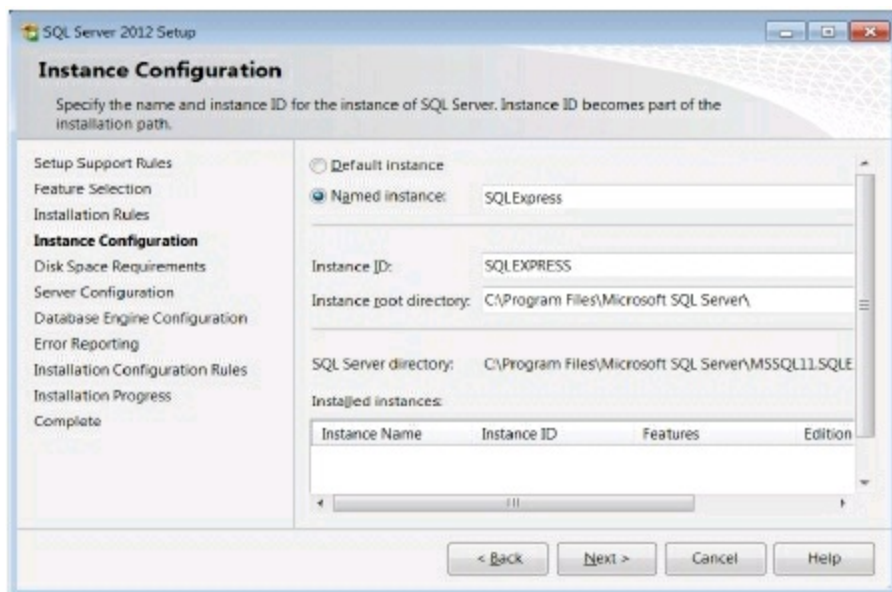
(6) 下一个窗口是“Installation Rules”窗口，它列出了所有需要的组件，如图C-3所示。

如果一个或多个组件安装失败，点击“Failed”链接，我们可以查看失败组件的信息。在继续安装前，我们需要把所有失败组件的问题解决掉。例如，图C-3展示了一个失败的组件。



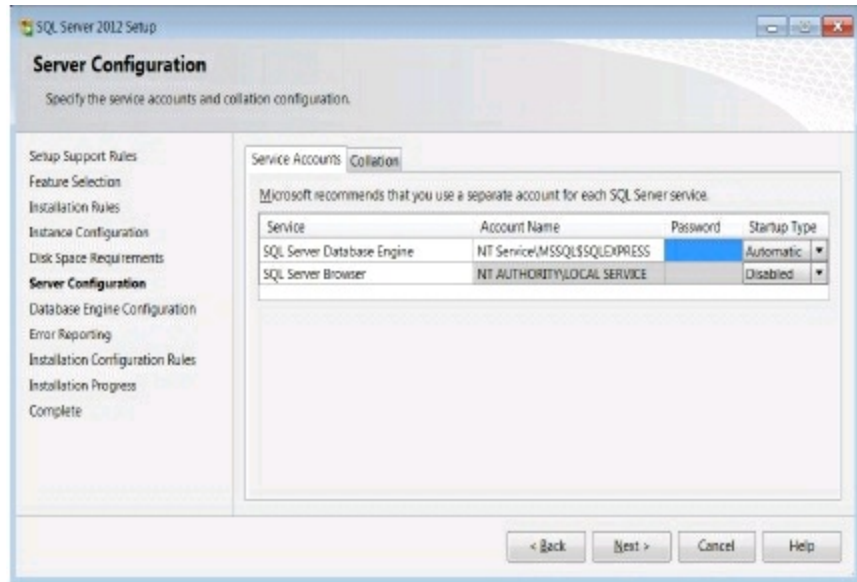
图C-3 “Installation Rules”窗口

(7) 解决了所有的失败组件后，点击“Back”按钮返回到“Feature Selection”窗口，然后在“Feature Selection”窗口点击“Next”按钮再来试一下。如果组件安装正确，将跳过“Installation Rules”窗口，我们会看到“Instance Configuration”窗口，如图C-4所示。



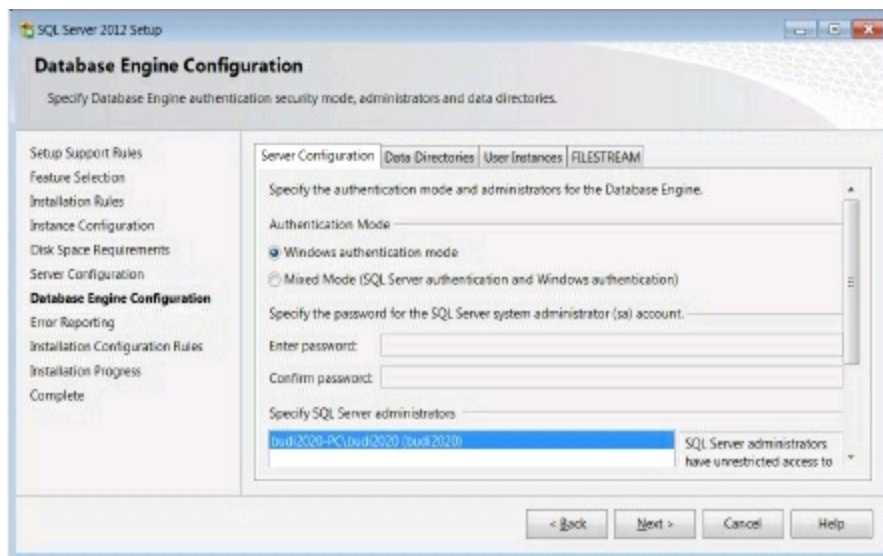
图C-4 “Instance Configuration”窗口

(8) 接受默认设置并且点击“Next”按钮，我们就会看到“Server Configuration”窗口，如图C-5所示。



图C-5 “Server Configuration”窗口

(9) 点击“Next”按钮，出现“Database Engine Configuration”窗口，如图C-6所示。



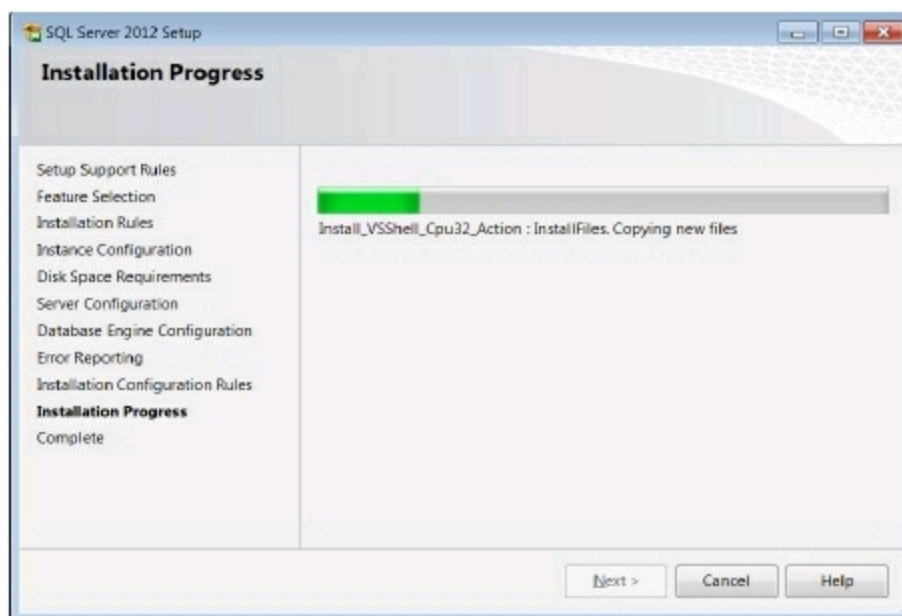
图C-6 “Database Engine Configuration”窗口

(10) 再次点击“Next”按钮，下一个窗口是“Error Reporting”窗口，如图C-7所示。



图C-7 “Error Reporting”窗口

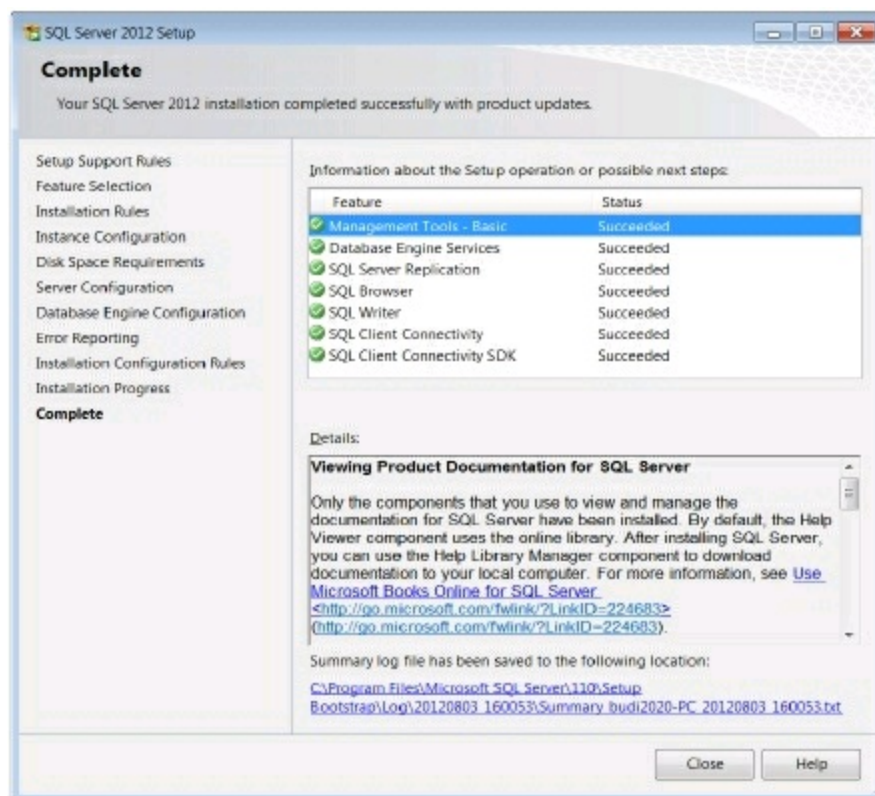
（11）最后点击“Next”按钮，安装开始。“Installation Progress”窗口展示了安装的过程，如图C-8所示。



图C-8 “Installation Progress”窗口

（12）所有步骤完成后，“Installation Progress”窗口将会消失，出现另一个窗口，表示安装已完成，如图C-9所示。检查安装的组件，然后点击“Close”按钮，将会返回到“SQL Server Installation Center”窗口，然

后再次点击“Close”按钮。



图C-9 “Complete”窗口

安装向导将在Microsoft SQL Server 2012的菜单下创建Microsoft SQL Server Management Studio的一个快捷方式。我们用这个管理程序来管理数据库对象，例如创建数据库、表以及向其中存储数据。

C.3 连接到SQL Server并创建一个数据库

我们用Microsoft SQL Server Management Studio来创建一个数据库。运行管理程序来管理一个数据库，在打开SQL Server Management Studio时，它会提示我们连接到一个数据库。登录窗口如图C-10所示。

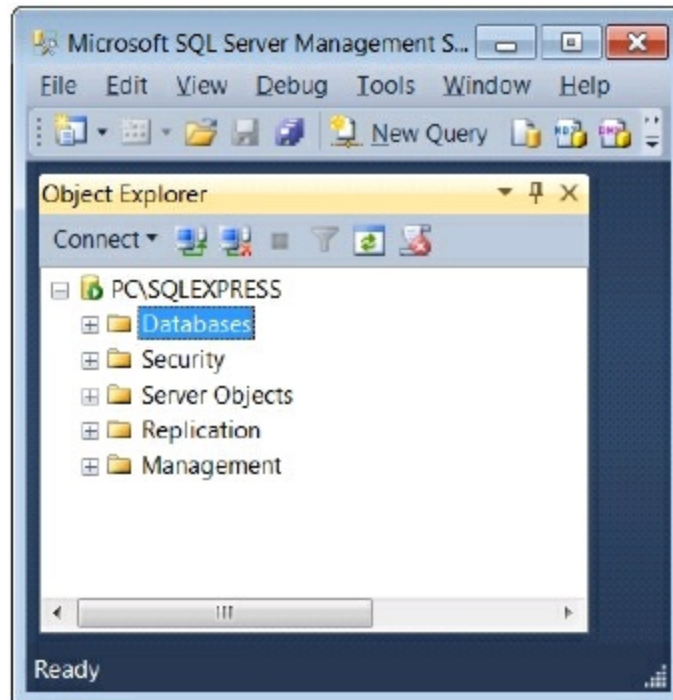


图C-10 SQL Server Management Studio的登录窗口

首先要做的事是连接到SQL Server的一个实例，它应该在安装SQL Server时已经创建了。因此，我们从“Server name”下拉框中选择一个服务器名称。

服务器需要登录才能连接，我们可以使用Windows认证或者SQL Server认证，如果使用前者，点击“Connect”按钮；如果使用后者，输入用户名和密码，然后点击“Connect”按钮。

在成功登录后，SQL Server Management Studio的主窗口如图C-11所示。在“Object Explorer”中，我们可以看到当前登录的SQL Server实例。



图C-11 SQL Server Management Studio的主窗口

要创建一个数据库，右键点击“Object Explorer”中的Databases文件夹，然后在弹出菜单中点击“New Database”，该窗口将会打开。输入一个名称，然后点击“Add”按钮，新的数据库将会出现在“Object Explorer”中的Databases文件夹下。

欢迎来到异步社区！

异步社区的来历

异步社区(www.epubit.com.cn)是人民邮电出版社旗下IT专业图书旗舰社区，于2015年8月上线运营。

异步社区依托于人民邮电出版社20余年的IT专业优质出版资源和编辑策划团队，打造传统出版与电子出版和自出版结合、纸质书与电子书结合、传统印刷与POD按需印刷结合的出版平台，提供最新技术资讯，为作者和读者打造交流互动的平台。

社区内提供随书附赠的资源，如书中的案例或程序源代码。

另外，社区还提供了大量的免费电子书，只要注册成为社区用户就可以免费下载。


与作译者互动

很多图书的作译者已经入驻社区，您可以关注他们，咨询技术问题；可以阅读不断更新的技术文章，听作译者和编辑畅聊好书背后有趣的故事；还可以参与社区的作者访谈栏目，向您关注的作者提出采访题目。

灵活优惠的购书

您可以方便地下单购买纸质图书或电子图书，纸质图书直接从人民邮电出版社书库发货，电子书提供多种阅读格式。

对于重磅新书，社区提供预售和新书首发服务，用户可以第一时间买到心仪的新书。

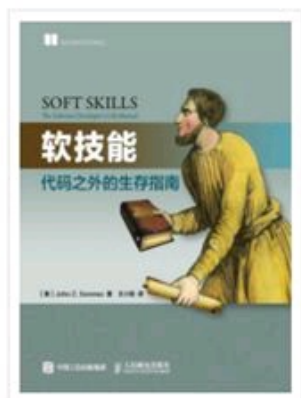
用户帐户中的积分可以用于购书优惠。100积分=1元，购买图书时，在  里填入可使用的积分数值，即可扣减相应金额。

特别优惠

购买本电子书的读者专享异步社区优惠券。使用方法：注册成为社区用户，在下单购书时输入“57AWG”，然后点击“使用优惠码”，即可享受电子书8折优惠（本优惠券只可使用一次）。

纸电图书组合购买

社区独家提供纸质图书和电子书组合购买方式，价格优惠，一次购买，多种阅读选择。



软技能：代码之外的生存指南

[美]约翰·Z·森梅兹 (John Z. Sonmez) (作者)

王小刚 (译者)

杨海玲 (责任编辑)



分享

6

推荐



想读

9.0K

阅读

这是一本真正从“人”（而非技术也非管理）的角度关注软件开发人员自身发展的书。书中论述的内容既涉及生活习惯，又包括思维方式，凸显技术中“人”的因素，全面讲解软件行业从业人员所需知道的所有“软技能”。

本书聚焦于软件开发人员生活的方方面面，从揭秘面试的流程到精耕细作出一份杀手级简历，从创建大受欢迎的博客到打造你的个人品牌，从提高自己工作效率到与如何与“拖延症”做斗争，甚至包括如何投资不动产，如何关注自己的健康。

本书共分为职业篇、自我营销篇、学习篇、生产力篇、理财篇、健身篇、精神篇等七篇，概括了软件行业从业人员所需的“软技能”。

● 纸质版 ¥59.00 **¥46.02 (7.8折)**

● 电子版 **¥35.00**

● 电子版 + 纸质版 **¥59.00**

配套文件下载

立即购买

下载PDF样章

社区里还可以做什么？

提交勘误

您可以在图书页面下方提交勘误，每条勘误被确认后可以获得100积分。热心勘误的读者还有机会参与书稿的审校和翻译工作。

写作

社区提供基于Markdown的写作环境，喜欢写作的您可以在此一试身手，在社区里分享您的技术心得和读书体会，更可以体验自出版的乐趣，轻松实现出版的梦想。

如果成为社区认证作译者，还可以享受异步社区提供的作者专享特色服务。

会议活动早知道

您可以掌握IT圈的技术会议资讯，更有机会免费获赠大会门票。

加入异步

扫描任意二维码都能找到我们：



异步社区



微信订阅号



微信服务号



官方微博



QQ群: 368449889

社区网址: www.epubit.com.cn

官方微信: 异步社区

官方微博: @人邮异步社区, @人民邮电出版社-信息技术分社

投稿&咨询: contact@epubit.com.cn