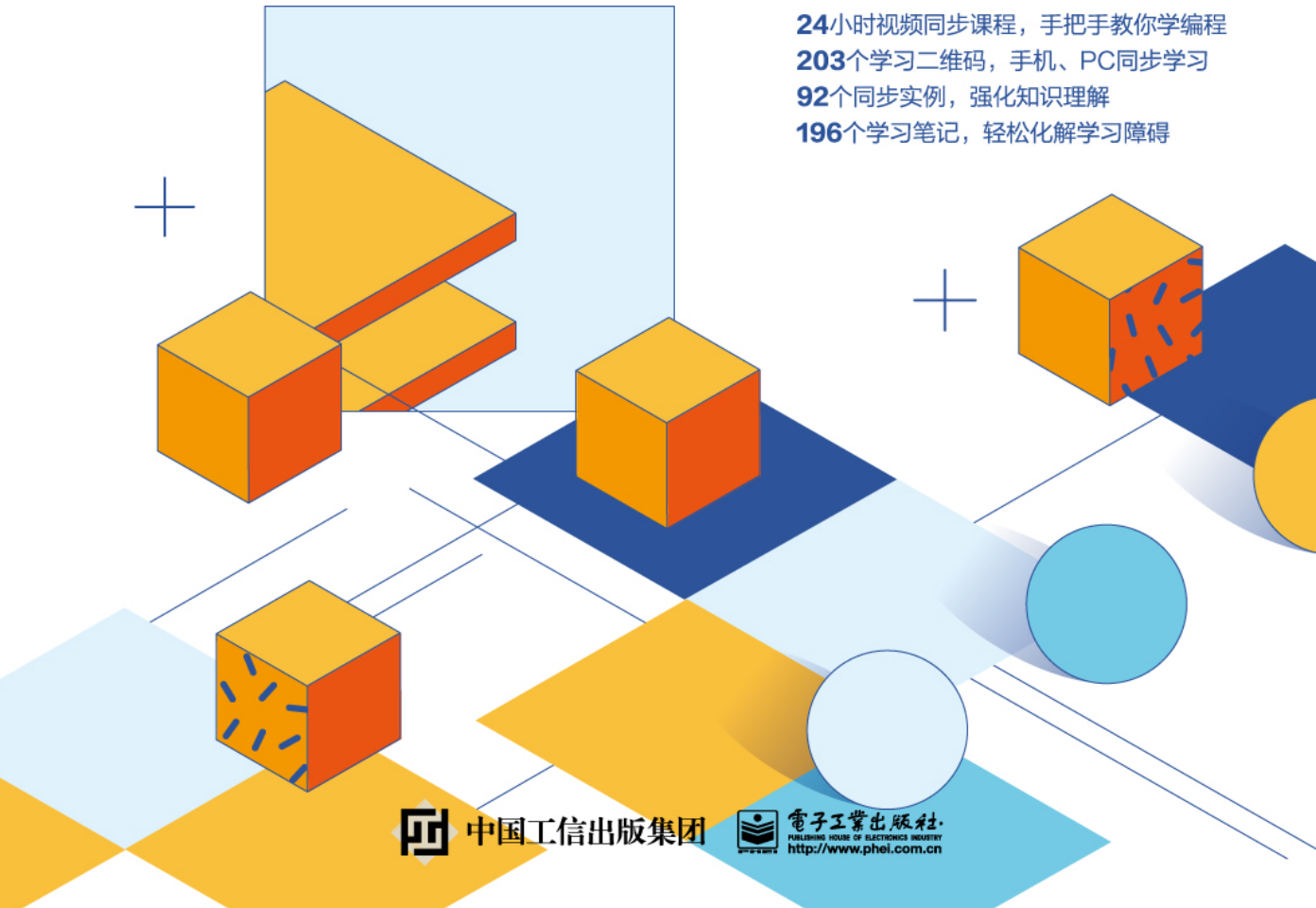
零基础 C# 学习笔记

明日科技 编著

基础篇—进阶篇—高级篇　一步到位

在线 QQ、微信群，400 企业电话支持

24小时视频同步课程，手把手教你学编程
203个学习二维码，手机、PC同步学习
92个同步实例，强化知识理解
196个学习笔记，轻松化解学习障碍



中国工信出版集团
電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

零基础 C# 学习笔记

明日科技 编著

電子工業出版社
Publishing House of Electronics Industry
北京•BEIJING

内 容 简 介

本书从入门读者的角度出发，通过通俗易懂的语言、丰富多彩的示例，循序渐进地让读者在实践中学习 C# 编程知识，并提升自己的实际开发能力。全书分 3 篇共 15 章，内容包括初识 Visual Studio、踏上 C# 开发的征程、必须学会的 C# 语法、流程控制语句、数组的使用、看似简单的字符串、面向对象程序设计、Windows 交互式图形界面、Windows 控件——C/S 程序的基础、数据访问技术、程序调试与异常处理、I/O 数据流技术、GDI+ 绘图应用、Socket 网络编程、多线程编程技术。书中所有知识都结合具体示例进行介绍，涉及的程序代码都给出了详细的注释，可以使读者轻松领会 C# 程序开发的精髓，快速提高开发技能。

本书不仅适合作为软件开发入门者的自学用书，也适合作为高等院校相关专业的教学参考书，还可供开发人员查阅、参考。

图书在版编目（CIP）数据

零基础 C# 学习笔记 / 明日科技编著 . —北京：电子工业出版社，2021.3

ISBN 978-7-121-39951-0

Ⅰ . ①零… Ⅱ . ①明… Ⅲ . ① C 语言 – 程序设计 Ⅳ . ① TP312.8

中国版本图书馆 CIP 数据核字（2020）第 224806 号

责任编辑：张　毅　　特约编辑：田学清
印　　刷：三河市兴达印务有限公司
装　　订：三河市兴达印务有限公司
出版发行：电子工业出版社
　　北京市海淀区万寿路 173 信箱　　邮编：100036
开　　本：787 × 1092　1/16　印张：21　字数：485 千字
版　　次：2021 年 3 月第 1 版
印　　次：2021 年 3 月第 1 次印刷
定　　价：108.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888，88258888。

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

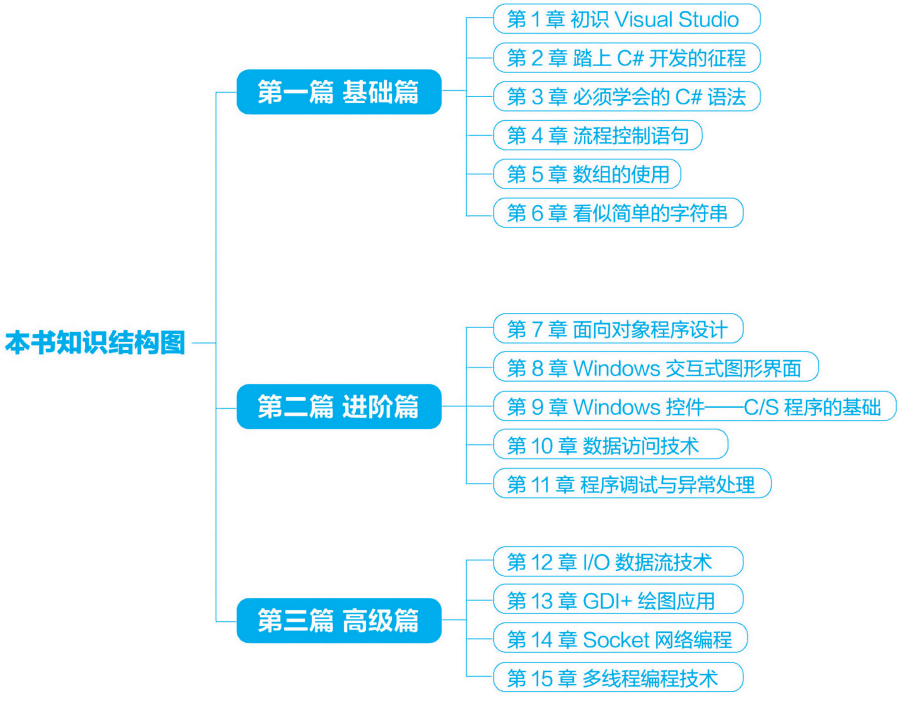
本书咨询联系方式：（010）57565890，meidipub@phei.com.cn。

前　言

C# 是微软公司为 Visual Studio 开发平台推出的一种简洁、类型安全的面向对象的编程语言，开发人员可以通过它编写在 .NET Framework 上运行的各种安全可靠的应用程序。C# 自面世以来以其易学易用、功能强大的优势被广泛应用，而 Visual Studio 开发平台则凭借其强大的可视化用户界面设计，让程序员从复杂的界面设计中解脱出来，使编程成为一种享受。C# 不但可以开发数据库管理系统，也可以开发集声音、动画、视频为一体的多媒体应用程序和网络应用程序，这使得它正在成为程序开发人员使用的主流编程语言。

本书内容

本书包含了学习 C# 从入门到高级应用开发所需的各类必备知识，全书分 3 篇共 15 章，知识结构图如下。



本书特点

- 由浅入深，循序渐进。本书以初、中级程序员为对象，先从 C# 语言基础学起，再学习窗体开发、数据库、数据流、GDI+ 绘图、网络多线程等知识。讲解过程中步骤详尽，使读者在阅读时一目了然，从而快速掌握书中内容。
- 教学视频，讲解详尽。本书基础知识部分提供了配套教学视频，读者可以根据这些视频更快速地学习，感受编程的快乐和成就感，增强进一步学习的信心，从而快速成为编程高手。
- 示例典型，轻松易学。通过示例学习是最好的学习方式，本书在讲解知识时，通过多个示例，透彻详尽地讲述了在实际开发中所需的各类知识。另外，为了便于读者阅读程序代码，快速学习编程技能，书中几乎每行代码都提供了注释。
- 精彩栏目，贴心提醒。本书根据需要在各章安排了很多“学习笔记”小栏目，让读者可以在学习过程中轻松地理解相关知识点及概念，快速掌握个别技术的应用技巧。

读者对象

- 初学编程的自学者
- 大中专院校的老师和学生
- 毕业设计的学生
- 程序测试及维护人员
- 编程爱好者
- 相关培训机构的老师和学员
- 初、中、高级程序开发人员
- 参加实习的“菜鸟”程序员

读者服务

为了方便解决本书疑难问题，我们提供了多种服务方式，并由作者团队提供在线技术指导和社区服务，服务方式如下。

- 服务网站：www.mingrisoft.com
- 服务邮箱：mingrisoft@mingrisoft.com
- 企业 QQ：4006751066
- QQ 群：162973740
- 服务电话：400-67501966、0431-84978981

本书约定

开发环境及工具如下。

- 操作系统：Windows 7、Windows 10 等
- 开发工具：Visual Studio 2017（Visual Studio 2015 及 Visual Studio 2019 兼容）
- 数据库：SQL Server 2014

致读者

本书由明日科技 C# 程序开发团队组织编写，主要编写人员有王小科、申小琦、赵宁、李菁菁、何平、张鑫、周佳星、王国辉、李磊、赛奎春、杨丽、高春艳、冯春龙、张宝华、庞凤、宋万勇、葛忠月等。在编写过程中，我们以科学、严谨的态度，力求精益求精，但错误、疏漏之处在所难免，敬请广大读者批评指正。

感谢您购买本书，希望本书能成为您编程路上的领航者。

祝读书快乐！

目　录

第一篇　基础篇

第 1 章　初识 Visual Studio 1

1.1　了解软件 1
1.2　软件开发相关概念 2
1.3　熟悉 Visual Studio 2017 4
1.3.1　创建项目 4
1.3.2　菜单栏 6
1.3.3　工具栏 6
1.3.4　“工具箱”窗口 7
1.3.5　“属性”窗口 7
1.3.6　“错误列表”窗口 8

第 2 章　踏上 C# 开发的征程 9

2.1　编写第一个 C# 程序 9
2.2　C# 程序结构预览 12
2.2.1　命名空间 12
2.2.2　类 14
2.2.3　关键字与标识符 16
2.2.4　Main 方法 17
2.2.5　C# 语句 19
2.2.6　注释 21
2.2.7　一个完整的 C# 程序 23
2.3　程序编写规范 24
2.3.1　代码编写规则 24
2.3.2　命名规范 25

第 3 章　必须学会的 C# 语法 28

3.1　为什么要使用变量 28

3.2 变量是什么 28
3.3 变量的声明及初始化 29
3.3.1 变量的声明 29
3.3.2 简单数据类型 30
3.3.3 变量的初始化 40
3.3.4 变量的作用域 42
3.4 常量 43
3.4.1 常量是什么 44
3.4.2 常量的分类 44
3.5 运算符 45
3.5.1 算术运算符 46
3.5.2 自增运算符和自减运算符 47
3.5.3 赋值运算符 48
3.5.4 关系运算符 50
3.5.5 逻辑运算符 51
3.5.6 位运算符 53
3.5.7 移位运算符 55
3.5.8 条件运算符 56
3.6 数据类型转换 57
3.6.1 隐式类型转换 58
3.6.2 显式类型转换 58
3.6.3 使用 Convert 类进行转换 59
3.7 运算符优先级与结合性 60

第 4 章 流程控制语句 62

4.1 决策分支 62
4.2 if 语句 63
4.2.1 最简单的 if 语句 63
4.2.2 if…else 语句 65
4.2.3 if…else if…else 语句 68
4.2.4 if 语句的嵌套 71
4.3 switch 多分支语句 74
4.3.1 switch 语句 74
4.3.2 switch 语句与 if…else if…else 语句的区别 77

4.4 while 和 do…while 循环 77
4.4.1 while 循环 78
4.4.2 do…while 循环 80
4.4.3 while 语句和 do…while 语句的区别 82
4.5 for 循环 82
4.5.1 for 循环的一般形式 82
4.5.2 for 循环的变体 84
4.5.3 for 循环中逗号的应用 85
4.6 循环的嵌套 86
4.7 跳转语句 88
4.7.1 break 语句 88
4.7.2 continue 语句 89

第 5 章 数组的使用 91

5.1 数组概述 91
5.2 一维数组 92
5.2.1 一维数组的创建 92
5.2.2 一维数组的初始化 94
5.2.3 一维数组的使用 95
5.3 二维数组 96
5.3.1 二维数组的创建 96
5.3.2 二维数组的初始化 98
5.3.3 二维数组的使用 99
5.3.4 不规则数组的定义 101
5.4 数组与 Array 类 101
5.5 数组的基本操作 103
5.5.1 使用 foreach 语句遍历数组 103
5.5.2 对数组进行排序 105

第 6 章 看似简单的字符串 107

6.1 什么是字符串 107
6.2 字符串的声明与初始化 107
6.2.1 声明字符串 107
6.2.2 字符串的初始化 108
6.3 提取字符串信息 109

6.3.1 获取字符串长度 109
6.3.2 获取指定位置的字符 110
6.3.3 获取子字符串索引位置 110
6.3.4 判断字符串首、尾内容 113
6.4 字符串操作 114
6.4.1 字符串的拼接 114
6.4.2 比较字符串 115
6.4.3 字符串的大小写转换 116
6.4.4 格式化字符串 117
6.4.5 截取字符串 122
6.4.6 分割字符串 123
6.4.7 去除空白内容 124
6.4.8 替换字符串 125
6.5 可变字符串类 126
6.5.1 StringBuilder 类的定义 126
6.5.2 StringBuilder 类的使用 127

第二篇 进阶篇

第 7 章 面向对象程序设计 129

7.1 面向对象概述 129
7.1.1 对象 129
7.1.2 类 131
7.1.3 三大基本特征 132
7.2 类 133
7.2.1 类的声明 133
7.2.2 类的成员 134
7.2.3 构造函数 138
7.2.4 析构函数 141
7.2.5 权限修饰符 141
7.3 方法 143
7.3.1 方法的声明 143
7.3.2 方法的参数 144
7.3.3 方法的重载 146

7.4 类的静态成员 147
7.5 对象的创建及使用 148
7.5.1 对象的创建 149
7.5.2 对象的销毁 150
7.5.3 类与对象的关系 151
7.6 继承 151
7.6.1 继承的实现 151
7.6.2 base 关键字 154
7.6.3 继承中的构造函数与析构函数 156
7.7 多态 157
7.7.1 虚方法的重写 157
7.7.2 抽象类与抽象方法 159
7.7.3 接口的使用 162

第 8 章 Windows 交互式图形界面 166

8.1 开发应用程序的步骤 166
8.2 Form 窗体 168
8.2.1 添加或删除窗体 169
8.2.2 多窗体的使用 170
8.2.3 窗体的属性 171
8.2.4 窗体的显示与隐藏 174
8.2.5 窗体的事件 175
8.3 MDI 窗体 177
8.3.1 MDI 窗体的概念 178
8.3.2 如何设置 MDI 窗体 178
8.3.3 排列 MDI 子窗体 179

第 9 章 Windows 控件——C/S 程序的基础 182

9.1 控件概述 182
9.2 控件的相关操作 183
9.2.1 添加控件 183
9.2.2 对齐控件 184
9.2.3 删除控件 184
9.3 Windows 控件的使用 184
9.3.1 Label 控件 184

9.3.2 Button 控件 185
9.3.3 TextBox 控件 186
9.3.4 RadioButton 控件 187
9.3.5 CheckBox 控件 188
9.3.6 RichTextBox 控件 190
9.3.7 ComboBox 控件 192
9.3.8 ListBox 控件 193
9.3.9 GroupBox 控件 195
9.3.10 ListView 控件 195
9.3.11 TreeView 控件 199
9.3.12 ImageList 组件 202
9.3.13 Timer 组件 203
9.4 菜单控件、工具栏控件与状态栏控件 205
9.4.1 菜单控件 205
9.4.2 工具栏控件 206
9.4.3 状态栏控件 207
9.5 对话框 209
9.5.1 消息对话框 210
9.5.2 打开对话框控件 211
9.5.3 另存为对话框控件 213
9.5.4 浏览文件夹对话框控件 214

第 10 章 数据访问技术 216

10.1 ADO.NET 概述 216
10.1.1 ADO.NET 对象模型 216
10.1.2 数据访问命名空间 217
10.2 Connection 对象 218
10.2.1 熟悉 Connection 对象 218
10.2.2 数据库连接字符串 219
10.2.3 应用 SqlConnection 连接对象连接数据库 220
10.3 Command 对象 221
10.3.1 熟悉 Command 对象 221
10.3.2 使用 Command 对象操作数据 222
10.3.3 使用 Command 对象调用存储过程 223

10.4 DataReader 对象 225
10.4.1 DataReader 对象概述 225
10.4.2 使用 DataReader 对象读取数据 226
10.5 DataSet 对象和 DataAdapter 对象 228
10.5.1 DataSet 对象 228
10.5.2 DataAdapter 对象 229
10.5.3 使用 DataAdapter 对象填充 DataSet 数据集 230
10.6 DataGridView 控件的使用 231
10.7 Entity Framework 编程基础 234
10.7.1 什么是 Entity Framework 234
10.7.2 EF 实体数据模型 235
10.7.3 EF 运行环境 235
10.7.4 创建实体数据模型 236
10.7.5 使用 EF 对数据表进行增、删、改、查操作 239

第 11 章 程序调试与异常处理 243

11.1 程序调试 243
11.1.1 Visual Studio 编辑器调试 243
11.1.2 Visual Studio 调试器调试 244
11.2 异常处理 248
11.2.1 try…catch 语句 249
11.2.2 try…catch…finally 语句 250
11.2.3 throw 语句 251

第三篇 高级篇

第 12 章 I/O 数据流技术 254

12.1 文件的基本操作 254
12.1.1 File 类 254
12.1.2 FileInfo 类 255
12.1.3 判断文件是否存在 256
12.1.4 创建文件 257
12.1.5 复制文件 258
12.1.6 移动文件 259

12.1.7 删除文件 260
12.1.8 获取文件的基本信息 261
12.2 文件夹的基本操作 262
12.2.1 Directory 类 262
12.2.2 DirectoryInfo 类 263
12.2.3 判断文件夹是否存在 263
12.2.4 创建文件夹 264
12.2.5 移动文件夹 265
12.2.6 删除文件夹 266
12.2.7 遍历文件夹 267
12.3 I/O（输入 / 输出） 269
12.3.1 流概述 270
12.3.2 文件 I/O 流的介绍 271
12.3.3 使用 I/O 流操作文本文件 273

第 13 章 GDI+ 绘图应用 276

13.1 GDI+ 绘图基础 276
13.1.1 GDI+ 概述 276
13.1.2 Graphics 类 277
13.2 设置画笔与画刷 278
13.2.1 设置画笔 278
13.2.2 设置画刷 279
13.3 绘制几何图形 280
13.3.1 绘制图形 282
13.3.2 填充图形 284
13.4 绘制图像 285

第 14 章 Socket 网络编程 287

14.1 计算机网络基础 287
14.1.1 局域网与广域网 287
14.1.2 网络协议 288
14.1.3 端口及套接字 289
14.2 IP 地址封装 290
14.3 TCP 程序设计 293
14.3.1 Socket 类 293

14.3.2 TcpClient 类和 TcpListener 类 294
14.3.3 TCP 网络程序示例 295
14.4 UDP 程序设计 298
14.4.1 UdpClient 类 299
14.4.2 UDP 网络程序示例 299

第 15 章 多线程编程技术 303

15.1 线程概述 303
15.1.1 线程的定义与分类 303
15.1.2 多线程的优 / 缺点 304
15.2 线程的实现 305
15.2.1 使用 Thread 类创建线程 305
15.2.2 线程的生命周期 308
15.3 操作线程的方法 309
15.3.1 线程的休眠 309
15.3.2 线程的加入 310
15.3.3 线程的终止 313
15.3.4 线程的优先级 313
15.4 线程的同步 315
15.4.1 线程同步机制 315
15.4.2 使用 lock 关键字实现线程同步 317
15.4.3 使用 Monitor 类实现线程同步 319
15.4.4 使用 Mutex 类实现线程同步 321

第一篇　基础篇

第 1 章　初识 Visual Studio

软件在现代人的日常生活中随处可见，比如，大家使用的 Windows 操作系统、智能手机中的各种应用等都是软件，那么，这些软件是如何生成的呢？我们能不能开发自己的软件呢？答案是可以的。本章将带领大家了解 C#（读作 C Sharp）及其使用的 Visual Studio 2017 开发环境。C# 是微软公司推出的一种语法简洁、类型安全的面向对象的编程语言，使用它可以开发各种软件；而 Visual Studio 2017 开发环境则是使用 C# 开发软件最好的工具。

1.1　了解软件

随着计算机的普及，计算机中安装的软件对人们的日常生活和工作的作用也显得越来越重要。例如，大家在聊天时经常使用 QQ 软件；在工作过程中使用 Office 软件；在处理照片时使用美图秀秀软件；在观看视频时使用优酷等视频播放软件。

以上都是我们经常用到的一些软件，那么，什么是软件呢？

软件其实是一种计算机程序，而计算机程序是指为了得到结果由计算机等具有信息处理能力的硬件装置执行的代码化指令集合。

计算机程序告诉计算机如何完成一个具体的任务，由于现在的计算机还不能理解人类的自然语言，所以不能用自然语言编写计算机程序，而需要借助计算机语言（程序设计语言，它是一种人和计算机交换信息的语言），通过计算机语言指挥计算机进行工作。

综上所述，一个软件的生成过程为：程序员将由计算机语言组成的代码输入计算机，计算机对代码进行解释编译，最后由计算机生成软件，如图 1.1 所示。

图 1.1　软件的生成过程

1.2　软件开发相关概念

计算机程序中涉及的概念都比较抽象、专业，经常有初学编程的人对专业性的名词不明所以。本节将对常见的与软件开发相关的概念进行介绍。

1. 算法

算法是指对计算机工作步骤和方法的描述。算法的每个步骤都有严格规定，这些步骤能够被计算机识别并正确执行，并且每个步骤都能够被计算机理解为一个或一组唯一的动作而不使计算机产生歧义。算法必须有开始和结束，并且必须保证算法规定的每个步骤最终都能够被完成。

下面通过一个例子来说明算法。例如，要交换变量 a 与变量 b 的值，由于计算机本身不能够直接执行这项操作，因此交换两个变量值的通用方法是借用第三方变量将其作为临时变量。具体算法描述如下。

（1）将变量 a 的内容赋值给临时变量 c。

（2）将变量 b 的内容赋值给变量 a。

（3）将临时变量 c 存放的内容赋值给变量 b。

最终算法可以写成：

```
01  （1）c ← a。
02  （2）a ← b。
03  （3）b ← c。
```

综上所述，算法实际上是用自然语言描述的一个计算机程序，编写计算机程序也就是把用某种方式描述的算法通过计算机语言重新进行描述。

2. 数据结构

数据结构是计算机进行存储和组织数据的一种方式。“数据”很好理解，比如我们买东西共花了 50 元，“50”就是一个准确的数据。在计算机中，数据有整数、实数、字符串、图像和声音等多种类型，而数据结构就是指各种类型的数据之间的相互关系。常见的数据结构有数组、栈、队列、链表、树、图等。图 1.2 所示为一个树结构。

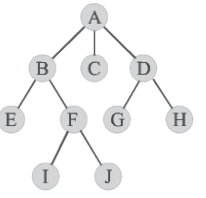


图 1.2　一个树结构

3. IDE

IDE 是“Integrated Development Environment”的缩写，表示集成开发环境。它是一种用于提供程序开发环境的应用程序，一般包括代码编辑器、编译器、调试工具和图形化用户界面工具等。例如，用于开发 C# 程序的 Visual Studio（见图 1.3）、用于开发 Java 程序的 Eclipse（见图 1.4）等都是集成开发环境。

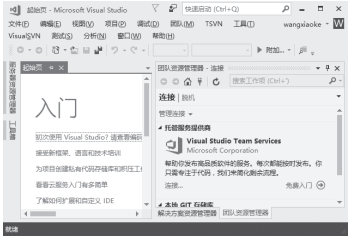


图 1.3　Visual Studio 集成开发环境

图 1.4　Eclipse 集成开发环境

4. SDK

SDK 是“Software Development Kit”的缩写，中文释义为“软件开发工具包”。它是一个覆盖面很广的名词，可以这么说：辅助开发某一类软件的相关文档、实例和工具的集合都可以称为 SDK。例如，在使用 C# 进行开发之前，需要安装由微软公司推出的 .NET SDK（.NET 软件开发工具包）。

5. 编译

编译是指把计算机语言变成计算机可以识别的二进制语言。由于计算机只能识别 0 和 1，所以编译程序就是把使用计算机语言编写的程序编译成计算机可以识别的二进制程序的过程。

1.3 熟悉 Visual Studio 2017

本节对 Visual Studio 2017 中的菜单栏、工具栏、“工具箱”窗口、“属性”窗口和“错误列表”窗口等进行介绍。

1.3.1 创建项目

初期学习 C# 语法和面向对象编程主要在 Windows 控制台应用程序环境下完成，下面将按步骤介绍控制台应用程序的创建过程。

创建控制台应用程序的步骤如下。

步骤 1，依次选择“开始”→“所有程序”→“Visual Studio 2017”命令，进入 Visual Studio 2017 起始页，如图 1.5 所示。

图 1.5 Visual Studio 2017 起始页

步骤 2，启动 Visual Studio 2017 之后，可以通过两种方法创建项目：一种是在菜单栏中依次选择“文件”→“新建”→“项目”命令，如图 1.6 所示；另一种是在 Visual Studio 2017 起始页选择“新建项目”板块中的相应命令，如图 1.7 所示。

图 1.6　在菜单栏中依次选择“文件”→“新建”→“项目”命令

图 1.7　选择“新建项目”板块中的相应命令

使用其中一种方法创建项目，弹出“新建项目”对话框，如图 1.8 所示。

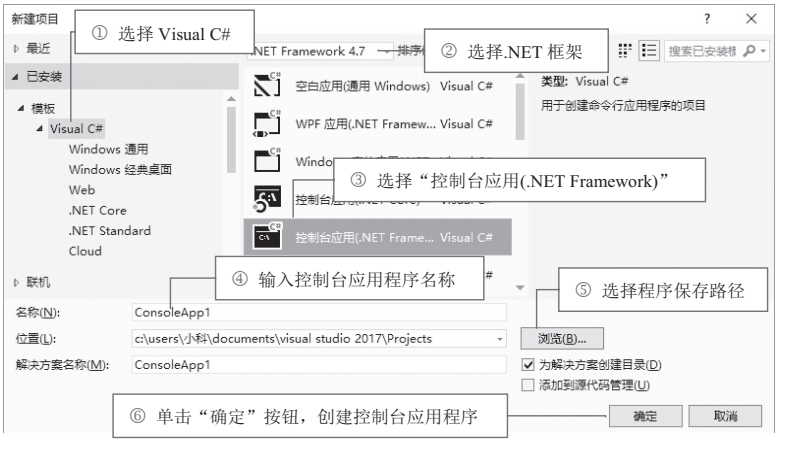


图 1.8　“新建项目”对话框

学习笔记

在图 1.8 中，选择“Windows 窗体应用 (.NET Framework)”即可创建 Windows 的窗体程序。

步骤 3，选择要使用的 .NET 框架和“控制台应用 (.NET Framework)”后，用户可对所要创建的控制台应用程序进行命名、选择存放位置、是否创建解决方案目录等设定（在命名时可以使用用户自定义的名称，也可使用默认名 ConsoleApp1；用户可以单击“浏览”

按钮设置项目存放的位置；需要注意的是，解决方案名称与项目名称一定要一致），然后单击“确定”按钮，完成控制台应用程序的创建。

1.3.2 菜单栏

菜单栏显示了所有可用的 Visual Studio 2017 命令，除“文件”“编辑”“视图”“窗口”“帮助”菜单外，其还提供了编程专用的功能菜单，如“项目”“生成”“调试”“工具”“测试”等菜单，如图 1.9 所示。

每个菜单项中都包含若干个菜单命令，分别执行不同的操作。例如，“调试”菜单包括调试程序的各种命令，如“开始调试”“开始执行”“新建断点”等命令，如图 1.10 所示。

图 1.9 Visual Studio 2017 菜单栏

图 1.10 “调试”菜单

1.3.3 工具栏

为了操作方便和快捷，菜单项中常用的命令按功能分组，分别放在相应的工具栏中。通过工具栏可以快速地访问常用的菜单命令。常用的工具栏有标准工具栏和调试工具栏，下面分别进行介绍。

（1）标准工具栏包括大多数常用的命令按钮，如“新建项目”“打开文件”“保存”“全部保存”等。Visual Studio 2017 标准工具栏如图 1.11 所示。

图 1.11 Visual Studio 2017 标准工具栏

（2）Visual Studio 2017 调试工具栏包括对应用程序进行调试的快捷按钮，如图 1.12 所示。

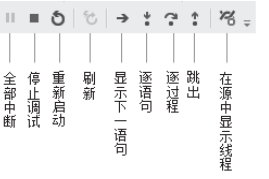


图 1.12　Visual Studio 2017 调试工具栏

1.3.4　“工具箱”窗口

工具箱是 Visual Studio 2017 的重要工具，每一个开发人员都必须对这个工具非常熟悉。工具箱提供了进行 C# 程序开发所必需的控件。通过工具箱，开发人员可以方便地进行可视化的窗体设计，简化了程序设计的工作量，提高了工作效率。根据控件功能的不同，将工具箱划分为 10 个栏目，如图 1.13 所示。单击某个栏目，将显示该栏目下的所有控件。当需要某个控件时，可以通过双击所需要的控件直接将控件加载到 Windows 窗体中，也可以先单击选择需要的控件，再将其拖动到 Windows 窗体上。

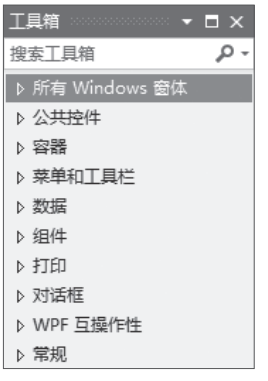


图 1.13　“工具箱”窗口

学习笔记

“工具箱”窗口在 Windows 窗体应用程序或 ASP.NET 网站应用程序中才会显示，在控制台应用程序中没有“工具箱”窗口，图 1.13 中显示的“工具箱”窗口是 Windows 窗体应用程序中的“工具箱”窗口。

1.3.5　“属性”窗口

“属性”窗口是 Visual Studio 2017 中的另一个重要工具，为 C# 程序的开发提供了简单的属性修改方式。Windows 窗体中的各个控件属性都可以由“属性”窗口设置完成。“属

性”窗口不仅提供了属性的设置及修改功能，还提供了事件的管理功能。“属性”窗口可以管理控件的事件，方便编程时对事件进行处理。

另外，“属性”窗口采用了两种方式管理属性和方法，分别为按分类方式和按字母顺序方式，读者可以根据自己的习惯采用不同的方式。在该窗口下方还有简单的功能说明，方便开发人员对控件的属性进行操作和修改。“属性”窗口的左侧是属性名称，右侧是属性值。“属性”窗口如图 1.14 所示。

图 1.14　“属性”窗口

1.3.6　“错误列表”窗口

“错误列表”窗口可以对代码中的错误进行即时提示并提供了可能的解决方法。例如，当某句代码结束时，如果忘记输入分号，“错误列表”窗口中就会显示如图 1.15 所示的错误提示。“错误列表”窗口就好像一个错误提示器，它可以将程序中的错误代码及时地显示给开发人员。

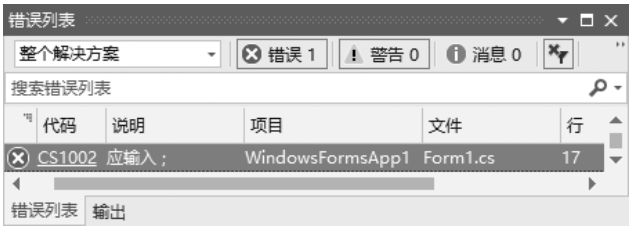


图 1.15　“错误列表”窗口

学习笔记

双击错误列表中的某项，Visual Studio 2017 会自动定位到发生错误的代码处。

第 2 章　踏上 C# 开发的征程

要学习 C# 编程，必然要熟悉 C# 程序的结构，而为了能够养成良好的编码习惯，在学习 C# 之初，熟悉常用的 C# 程序编写规范也是非常重要的。本章将详细介绍如何编写 C# 程序，以及 C# 程序的基本结构，另外，将对 C# 程序的常用编写规范进行介绍。

2.1　编写第一个 C# 程序

在大多数编程学习过程中，编写的第一个程序通常都是输出“Hello World”，这里将使用 Visual Studio 2017 和 C# 来编写这个程序。首先介绍使用 Visual Studio 2017 开发 C# 程序的基本步骤，如图 2.1 所示。

图 2.1　使用 Visual Studio 2017 开发 C# 程序的基本步骤

通过图 2.1 中的 3 个步骤，开发人员可以方便地创建并运行一个 C# 程序。例如，使用 Visual Studio 2017 在控制台中创建输出“Hello World”程序并运行，具体开发步骤如下。

步骤 1，在系统的开始菜单中找到“所有程序”→“Visual Studio 2017”，单击打开 Visual Studio 2017。

学习笔记

对于 Windows 10 操作系统，在开始菜单列表中找到“Visual Studio 2017”，单击即可打开 Visual Studio 2017。

步骤 2，依次选择 Visual Studio 2017 菜单栏中的“文件”→“新建”→“项目”命令，打开“新建项目”对话框，如图 2.2 所示。

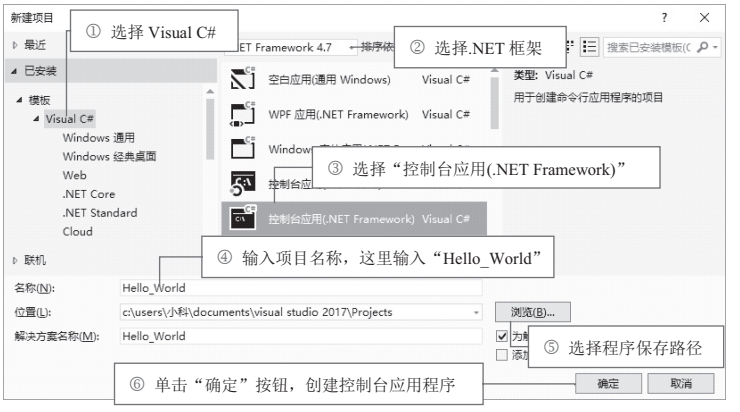


图 2.2 “新建项目”对话框

学习笔记

图 2.2 中的项目保存路径可以设置为计算机中的任意路径。

步骤 3，按照图 2.2 所示步骤创建一个控制台应用程序。

步骤 4，控制台应用程序创建完成后，会自动打开 Program.cs 文件，在该文件的 Main 方法中输入如下代码。

```
01  static void Main(string[] args)              // Main 方法，程序的入口方法
02  {
03    Console.WriteLine("Hello World");           // 输出“Hello World”
04    Console.ReadLine();                         // 定位控制台窗体
05  }
```

学习笔记

第 1 行代码是自动生成的 Main 方法，用来作为程序的入口方法。每个 C# 程序都必须有一个 Main 方法。

第 3 行代码中的 Console.WriteLine 方法主要用来向控制台输出内容。

第 4 行代码中的 Console.ReadLine 方法主要用来获取控制台的输出内容，这里用来将控制台窗体定位到桌面上。

单击 Visual Studio 2017 工具栏中的 ▶ 启动 图标按钮运行该程序，效果如图 2.3 所示。

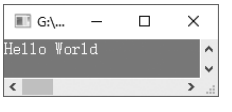


图 2.3 输出“Hello World”

在上面的代码中，使用 C# 输出了开发人员进入“编程世界”后遇到的一个经典语句“Hello World”，下面通过一个示例介绍如何使用 C# 输出中文内容。

示例 1．输出“人因梦想而伟大”

创建一个控制台应用程序，使用 Console.WriteLine 方法输出小米董事长雷军的经典语录“人因梦想而伟大”，完整代码如下。

```
01 using System;
02 using System.Collections.Generic;
03 using System.Linq;
04 using System.Text;
05
06 namespace Test
07 {
08     class Program
09     {
10         static void Main(string[] args)     //Main 方法，程序的入口方法
11         {
12             Console.WriteLine("人因梦想而伟大");            //输出文字
13             Console.WriteLine("                ——雷军");
14             Console.ReadLine();                            //固定控制台界面
15         }
16     }
17 }
```

学习笔记

第 1 ～ 4 行代码是自动生成的代码，用来引用默认的命名空间。

第 6 行代码是自动生成的命名空间。该命名空间的名称默认与创建的项目名称相同，开发人员可以手动修改。

第 8 行代码是自动生成的一个 Program 类。该类是 C# 程序的启动类，类的名称可以手动修改。

输出中文字符串代码的运行结果如图 2.4 所示。

图 2.4　输出中文字符串代码的运行结果

2.2 C# 程序结构预览

前面讲解了如何创建第一个 C# 程序，输出“Hello World”程序完整代码的效果如图 2.5 所示。

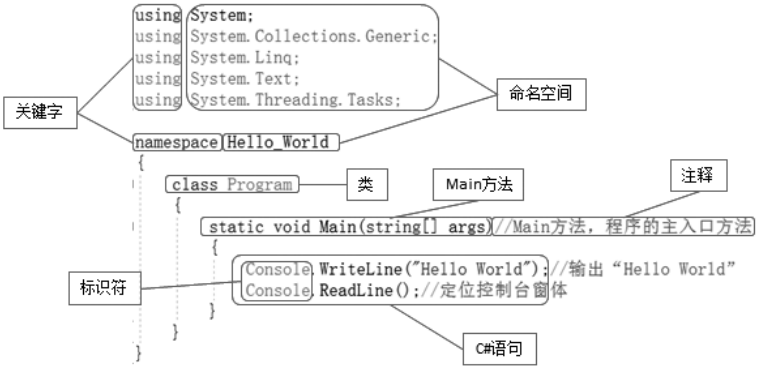


图 2.5 输出“Hello World”程序完整代码的效果

从图 2.5 中可以看出，一个 C# 程序总体可以分为命名空间、类、关键字、标识符、Main 方法、C# 语句和注释。本节将分别对 C# 程序的各个组成部分进行讲解。

2.2.1 命名空间

在 Visual Studio 中创建项目时，会自动生成一个与项目名称相同的命名空间。例如，在创建输出“Hello_World”项目时，会自动生成一个名称为“Hello_World”的命名空间，如图 2.6 所示。

```
namespace Hello_World
```

图 2.6 自动生成的命名空间

命名空间在 C# 程序中起到组成程序的作用，正如图 2.6 所示，在 C# 程序中定义命名空间时，需要使用 namespace 关键字，其语法如下。

```
namespace 命名空间名称
```

学习笔记

开发人员一般不用自定义命名空间，因为在创建项目或创建类文件时，Visual Studio 会自动生成一个命名空间。

命名空间既作为程序的“内部”组织系统，也作为向“外部”公开（一种向其他程序公开自己拥有的程序元素的方法）的组织系统。如果要调用某个命名空间中的类或方法，则首先需要使用 using 指令引入命名空间，这样就可以直接使用该命名空间中所包含的成员（包括类及类中的属性、方法等）了。

using 指令的基本形式如下。

```
using 命名空间名称；
```

学习笔记

C# 程序中的命名空间就像一个包含不同类型物品的仓库，而 using 指令就好比一把钥匙，命名空间的名称就好比仓库的名称，用户可以通过钥匙打开指定名称的仓库，进而从仓库中获取所需要的物品，其示意图如图 2.7 所示。

图 2.7　命名空间与仓库对比示意图

例如，下面的代码定义了一个 Demo 命名空间。

```
namespace Demo                    // 自定义一个名称为 Demo 的命名空间
```

定义命名空间后，如果要使用命名空间中所包含的类，则需要使用 using 指令引用命名空间。例如，下面代码使用 using 指令引用自定义的 Demo 命名空间。

```
using Demo;                       // 引用自定义的 Demo 命名空间
```

学习笔记

如果在使用指定命名空间中的类时，没有使用 using 指令引用命名空间，如下面代码所示，则会出现如图 2.8 所示的错误提示信息。

```
01  namespace Test
02  {
03      class Program
04      {
05          static void Main(string[] args)
06          {
07              Operation oper = new Operation();// 创建 Demo 命名空间中 Operation 类的对象
08          }
09      }
10  }
11  namespace Demo                              // 自定义一个名称为 Demo 的命名空间
12  {
13      class Operation                         // 自定义一个名称为 Operation 的类
14      {
15      }
16  }
```

图 2.8　没有引用命名空间而使用其中的类时出现的错误提示信息

要改正以上代码，可以直接在命名空间区域使用 using 指令引用 Demo 命名空间，代码如下。

```
using Demo;                                 // 引用自定义的 Demo 命名空间
```

学习笔记

在使用命名空间中的类时，如果不想用 using 指令引用命名空间，则可以在代码中使用命名空间调用其中的类。例如，下面的代码直接使用 Demo 命名空间调用其中的 Operation 类。

```
// 创建 Demo 命名空间中 Operation 类的对象
Demo.Operation oper = new Demo.Operation();
```

2.2.2　类

C# 程序的主要功能代码都是在类中实现的。类是一种数据结构，它可以封装数据成

员、方法成员和其他的类。因此，类是 C# 的核心和基本构成模块。C# 支持自定义类，使用 C# 编程就是通过编写自己的类来描述实际需要解决的问题。

学习笔记

如果把命名空间比作一家医院，类就相当于该医院的各个科室，如外科、妇科、内儿科、中医科等，各科室都有自己的工作方法，相当于在类中定义的变量、方法等。命名空间与类的关系示意图如图 2.9 所示。

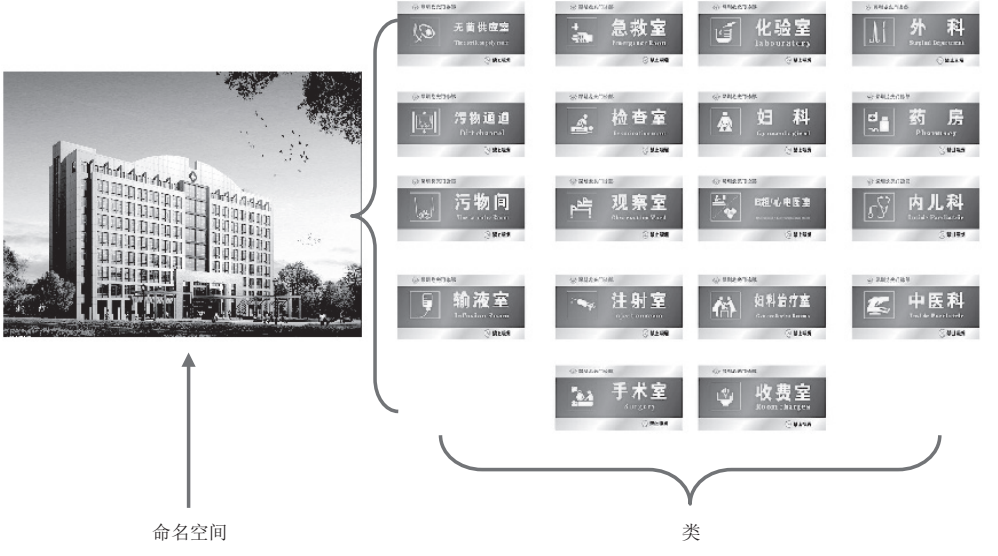


图 2.9　命名空间与类的关系示意图

在使用类之前都必须进行声明，一个类一旦被声明，就可以作为一个新的类使用。在 C# 中通过使用 class 关键字来声明类，声明语法如下。

```
class   [ 类名 ]
{
          [ 类中的代码 ]
}
```

学习笔记

在声明类时，还可以指定类的修饰符和其要继承的基类或者接口等信息，这里只要知道如何声明一个最基本的类即可，关于类的详细内容，会在第 7 章中进行专题讲解。

在上面的语法中，在命名类的名称时，最好能够体现类的含义或用途，而且类名一般采用第一个字母大写的名词，也可以采用多个词构成的组合词。

例如，声明一个汽车类，命名为 Car（该类没有任何意义，只演示如何声明一个类），代码如下。

```
01 class Car
02 {
03 }
```

2.2.3 关键字与标识符

1. 关键字

关键字是 C# 中已经被赋予特定意义的一些单词，在开发程序时，不可以把这些关键字作为命名空间、类、方法或属性等使用。输出“Hello World”程序中的 using、namespace、class、static 和 void 等都是关键字。C# 中的常用关键字如表 2.1 所示。

表 2.1 C# 中的常用关键字

int	public	this	finally	boolean	abstract
continue	float	long	short	throw	return
break	for	foreach	static	new	interface
if	goto	default	byte	do	case
void	try	switch	else	catch	private
double	protected	while	char	class	using

学习笔记

如果在开发程序时，使用 C# 中的关键字作为命名空间、类、方法或属性等的名称，如下面代码使用 C# 中的关键字 void 作为类的名称，则会出现如图 2.10 所示的错误提示信息。

```
01 class void
02 {
03 }
```

图 2.10 使用 C# 中的关键字作为类名时出现的错误提示信息

2. 标识符

标识符可以简单地理解为一个名字，比如每个人都有自己的名字，它主要用来标识类名、变量名、方法名、属性名、数组名等各种成员。

C# 标识符命名规则如下。

- 由任意顺序的字母、下画线（_）和数字组成。
- 第一个字符不能是数字。
- 不能是 C# 中的保留关键字。

下面是合法标识符。

```
_ID
name
user_age
```

下面是非法标识符。

```
4word                                   // 以数字开头
string                                  // C# 中的保留关键字
```

学习笔记

在 C# 中，标识符中不能包含 #、%、$ 等特殊字符。

在 C# 中，标识符中的字母是严格区分大小写的。两个相同的单词，如果大小写格式不一样，那么所代表的意义是完全不同的。例如，下面 3 个变量是完全独立、毫无关系的，就像 3 个长得比较像的人，彼此之间都是独立的个体。

```
01  int number=0;               // 全部小写
02  int Number=1;               // 部分大写
03  int NUMBER=2;               // 全部大写
```

学习笔记

在 C# 中允许使用汉字作为标识符，如“class 运算类”，在程序运行时并不会出现错误，但建议读者尽量不要使用汉字作为标识符。

2.2.4　Main 方法

在 Visual Studio 中创建控制台应用程序后，会自动生成一个 Program.cs 文件，该文件有一个默认的 Main 方法，代码如下。

```
01  class Program
02   {
03       static void Main(string[] args)
04       {
05       }
06   }
```

每个 C# 程序中都必须包含一个 Main 方法，它是类体中的主方法，也称为入口方法，可以说是激活整个程序的开关。Main 方法从“{”开始，至“}”结束。static 和 void 分别是 Main 方法的静态修饰符和返回值修饰符，C# 程序中的 Main 方法必须声明为 static，并且区分大小写。

学习笔记

如果将 Main 方法前面的 static 关键字删除，则程序在运行时会出现如图 2.11 所示的错误提示信息。

图 2.11　删除 static 关键字时 Main 方法出现的错误提示信息

Main 方法一般都是在创建项目时自动生成的，不用开发人员手动编写或修改。如果需要修改，则需要注意以下 3 个方面。

- Main 方法在类或结构内声明，它必须是静态（static）的，而且不应该是公用（public）的。
- Main 方法的返回类型有两种：void 或 int。
- Main 方法可以包含命令行参数 string[] args，也可以不包括。

根据以上 3 个方面的内容可以总结出，Main 方法有以下 4 种声明方式。

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

学习笔记

通常在 Main 方法中不写具体逻辑代码，只进行类实例化和方法调用（这好比手机来电话了，只需要按“接听”键就可以通话，而不需要考虑手机通过怎样的信号转换将电磁信号转换成声音）。这样的代码简洁明了，容易维护。养成良好的编码习惯，可以让程序员的工作事半功倍。

2.2.5　C# 语句

C# 语句是构造所有 C# 程序的基本单位，使用 C# 语句可以声明变量和常量、调用方法、创建对象或执行任何逻辑操作。C# 语句以分号终止。

例如，在输出“Hello World”程序中输出“Hello World”字符串和定位控制台窗体的代码就是 C# 语句。

```
01  Console.WriteLine("Hello World");          // 输出“Hello World”
02  Console.ReadLine();                        // 定位控制台窗体
```

上面的代码是两条最基本的 C# 语句，用来在控制台窗体中输出内容和读取内容，它们都用到了 Console 类。Console 类表示控制台应用程序的标准输入流、标准输出流和错误流，该类中包含很多方法，但与输入、输出相关的主要有 4 个方法，如表 2.2 所示。

表 2.2　Console 类中与输入、输出相关的方法

方　法	说　明
Read	从标准输入流中读取下一个字符
ReadLine	从标准输入流中读取下一行字符
Write	将指定的值写入标准输出流
WriteLine	将当前行终止符写入标准输出流

其中，Console.Read 方法和 Console.ReadLine 方法用来从控制台读取，它们的使用区别如下。

- Console.Read 方法：返回值为 int 类型，只能记录 int 类型的数据。
- Console.ReadLine 方法：返回值为 string 类型，可以将控制台中输入的任何类型的数据存储为字符串类型的数据。

学习笔记

在开发控制台应用程序时，经常使用 Console.Read 方法或 Console.ReadLine 方法定位控制台窗体。

Console.Write 方法和 Console.WriteLine 方法用来向控制台输出，它们的使用区别如下。

- Console.Write 方法：输出后不换行。

例如，使用 Console.Write 方法输出“Hello World”字符串，代码如下。

```
Console.Write("Hello World");
```

代码的运行结果如图 2.12 所示。

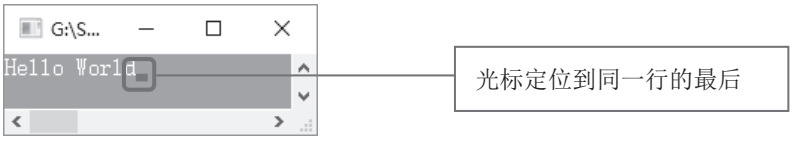


图 2.12　使用 Console.Write 方法输出“Hello World”字符串代码的运行结果

- Console.WriteLine 方法：输出后换行。

例如，使用 Console.WriteLine 方法输出“Hello World”字符串，代码如下。

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

代码的运行结果如图 2.13 所示。

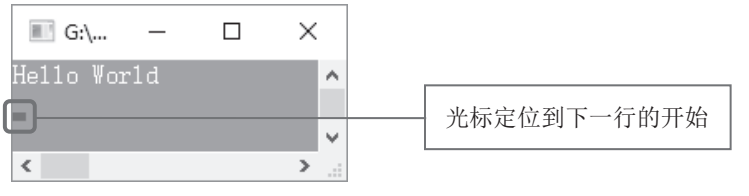


图 2.13　使用 Console.WriteLine 方法输出“Hello World”字符串代码的运行结果

学习笔记

C# 代码中所有的字母、数字、括号及标点符号均为英文输入法状态下的半角符号，而不能是中文输入法或英文输入法状态下的全角符号。例如，图 2.14 所示为中文输入法的分号引起的错误提示信息。

图 2.14　中文输入法的分号引起的错误提示信息

2.2.6　注释

注释是在编译程序时不执行的代码或文字，其主要功能是对某行或某段代码进行说明，以方便代码的理解与维护，或者在调试程序时将某行或某段代码设置为无效代码。常用的注释主要有行注释和块注释两种，下面分别进行简单介绍。

学习笔记

注释就像超市中各商品下面的价格标签，对商品的名称、价格、产地等信息进行说明，如图 2.15 所示。而在程序中，注释的最基本作用就是描述代码的作用，告诉别人你的代码要实现什么功能。

图 2.15　超市中各商品下面的价格标签相当于注释

1. 行注释

行注释以“//”开头，后面跟注释的内容。例如，在输出“Hello World”程序中使用行注释解释每行代码的作用，代码如下。

```
01  static void Main(string[] args)              // Main 方法，程序的入口方法
02  {
03      Console.WriteLine("Hello World");         // 输出“Hello World”
```

```
04      Console.ReadLine();                    // 定位控制台窗体
05  }
```

学习笔记

注释可以出现在代码的任意位置，但是不能分隔关键字和标识符。例如，下面的代码注释是错误的。

```
static void   // 错误的注释 Main(string[] args)
```

2. 块注释

如果注释的行数较少，则一般使用行注释。对于连续多行的大段注释，则使用块注释。块注释通常以“/*”开始，以“*/”结束，注释的内容放在它们之间。

例如，在输出“Hello World”程序中使用块注释将输出“Hello World”字符串和定位控制台窗体的 C# 语句注释为无效代码，代码如下。

```
01  static void Main(string[] args)              // Main 方法，程序的入口方法
02  {
03      /*    块注释开始
04          Console.WriteLine("Hello World");     // 输出“Hello World”字符串
05          Console.ReadLine();
06      */
07  }
```

学习笔记

块注释通常用来为类文件、类或方法等添加版权、功能等信息。例如，下面的代码使用块注释为 Program.cs 类添加版权、功能及修改日志等信息。

```
01  /*
02   * 版权所有：吉林省明日科技有限公司 © 版权所有
03   *
04   * 文件名：Program.cs
05   * 文件功能描述：类的主程序文件，主要作为入口
06   *
07   * 创建日期：2017 年 6 月 1 日
08   * 创建人：王小科
09   *
10   * 修改标识：2017 年 6 月 5 日
11   * 修改描述：增加 Add 方法，用来计算不同类型数据的和
12   * 修改日期：2017 年 6 月 5 日
13   *
14  */
```

```
15
16 using System;
17 using System.Collections.Generic;
18 using System.Linq;
19 using System.Text;
20
21 namespace Test
22 {
23     class Program
24     {
25     }
26 }
```

2.2.7　一个完整的 C# 程序

通过以上内容的讲解，我们熟悉了 C# 程序的基本组成，下面通过一个示例讲解如何编写一个完整的 C# 程序。

示例 2．输出软件启动页

使用 Visual Studio 2017 创建一个控制台应用程序，然后使用 Console.WriteLine 方法在控制台模拟输出“编程词典（珍藏版）”软件的启动页，代码如下。

```
01   static void Main(string[] args)
02   {
03       Console.WriteLine(" ------------------------------------------");
04       Console.WriteLine("|                                          |");
05       Console.WriteLine("|                                          |");
06       Console.WriteLine("|                                          |");
07       Console.WriteLine("|                                          |");
08       Console.WriteLine("|                                          |");
09       Console.WriteLine("|          编程词典（珍藏版）             |");
10       Console.WriteLine("|                                          |");
11       Console.WriteLine("|                                          |");
12       Console.WriteLine("|                                          |");
13       Console.WriteLine("|                    开发团队：明日科技    |");
14       Console.WriteLine("|                                          |");
15       Console.WriteLine("|                                          |");
16       Console.WriteLine("|                                          |");
17       Console.WriteLine("|                                          |");
18       Console.WriteLine("|        copyright   2000——2017   明日科技 |");
19       Console.WriteLine("|                                          |");
20       Console.WriteLine("|                                          |");
21       Console.WriteLine("|                                          |");
22       Console.WriteLine(" ------------------------------------------");
```

```
23          Console.ReadLine();
24      }
```

完成以上操作后，单击 Visual Studio 2017 工具栏中的 ▶ 启动 图标按钮即可运行上面的代码，代码运行结果如图 2.16 所示。

图 2.16　输出软件启动页代码运行结果

2.3　程序编写规范

下面给出两段实现同样功能的 C# 代码，如图 2.17 所示。

```
class Program
{
static void Main(string[] args)//Main方法，程序的入口方法
{
Console.WriteLine("Hello World");//输出“Hello World”
Console.ReadLine();//定位控制台窗体
}
}
```

代码段1

```
class Program
{
    static void Main(string[] args)//Main方法，程序的入口方法
    {
        Console.WriteLine("Hello World");//输出“Hello World”
        Console.ReadLine();//定位控制台窗体
    }
}
```

代码段2

图 2.17　两段实现同样功能的 C# 代码

大家在学习时，愿意阅读图 2.17 中的左侧代码还是右侧代码？相信大家更喜欢阅读图 2.17 中的右侧代码，因为它看上去更加规整。这是一种基本的代码编写规范。本节将对 C# 代码的编写规则及命名规范进行介绍。遵循一定的代码编写规则和命名规范可以使代码更加规范化，对代码的理解与维护起到至关重要的作用。

2.3.1　代码编写规则

代码编写规则通常对应用程序的功能没有影响，但对改善对源代码的理解是有帮助的。养成良好的习惯对于软件的开发和维护都是很有益的，下面列举一些常用的代码编写规则。

- 在编写 C# 程序时，统一代码缩进的样式，比如统一缩进 2 个字符或 4 个字符。
- 每编写完一行 C# 代码，都应该换行编写下一行代码。
- 在编写 C# 代码时，应该合理使用空格，使代码结构更加清晰。
- 尽量使用接口，然后使用类实现接口，以提高程序的灵活性。
- 关键的语句（包括声明关键的变量）必须要写注释。
- 建议局部变量在最接近使用它的地方声明。
- 不要使用 goto 系列语句，除非是用在跳出深层循环时。
- 避免编写超过 5 个参数的方法，如果要传递多个参数，则使用结构。
- 避免书写代码量过大的 try-catch 语句块。
- 避免在同一个文件中编写多个类。
- 生成和构建一个长的字符串时，一定要使用 StringBuilder 类型，而不使用 string 类型。
- 对于 if 语句，应该使用一对“{ }”把语句块包含起来。
- switch 语句一定要配置 default 语句来处理意外情况。

2.3.2　命名规范

命名规范在编写代码中起到很重要的作用，虽然不遵循命名规范程序也可以运行，但是使用命名规范可以更加直观地了解代码所代表的含义。下面介绍 C# 中常用的一些命名规范。

1. 两种命名方法

在 C# 中，常用的命名方法有两种，分别是 Pascal 命名法和 Camel 命名法，下面分别进行介绍。

（1）用 Pascal 命名法来命名方法和类型。以 Pascal 命名法命名时，第一个字母必须为大写，并且后面连接词的第一个字母也要大写。

学习笔记

Pascal 是以纪念法国数学家 Blaise Pascal 而命名的一种编程语言，C# 中的 Pascal 命名法就是根据该语言的特点总结出来的一种命名方法。

例如，定义一个公共类，并在此类中创建一个公共方法，代码如下。

```
01 public  class  User                    // 创建一个公共类
02 {
```

```
03      public  void  GetInfo()          // 在公共类中创建一个公共方法
04      {
05      }
06  }
```

（2）用 Camel 命名法来命名局部变量和方法的参数。使用 Camel 命名法命名时，变量或方法名中第一个单词的首字母需要小写。

学习笔记

Camel 命名法又称为驼峰式命名法，它是由骆驼的体形特征推理出来的一种命名方法。

例如，声明一个字符串变量和创建一个公共方法，代码如下。

```
01  string strUserName;                          // 声明一个字符串变量 strUserName
02  // 创建一个具有两个参数的公共方法
03  public void addUser(string strUserId, byte[] byPassword);
```

2. 程序中的命名规范

在开发项目时，不可避免地要遇到各个程序元素的命名问题，比如项目的命名、类的命名、方法的命名等。例如，在图 2.18 中声明了一个 User 类，在图 2.19 中声明了一个 aaa 类。

```
class User
{
}
```

图 2.18　声明 User 类

```
class aaa
{
}
```

图 2.19　声明 aaa 类

从类的命名上很容易可以看出，图 2.18 中的 User 类应该是与用户相关的一个类，但是图 2.19 中声明的 aaa 类，即使再有想象力的人，恐怕也想象不出这个类到底是做什么用的。由这两个例子可以看出，在对程序元素命名时，如果遵循一定的命名规范，将使代码更加具有可读性。下面介绍常用程序元素的基本命名规范。

- 在命名项目时，可以使用公司域名 + 产品名称，或者直接使用产品名称。

例如，在命名项目时，可以将项目命名为“mingrisoft.ERP”或“ERP”。其中，mingrisoft 是公司域名，ERP 是产品名称。

- 用有意义的名字定义命名空间，如公司名、产品名。

例如，利用公司名和产品名定义命名空间，代码如下。

```
01  namespace Mrsoft                          // 利用公司名定义命名空间
02  {
```

```
}
namespace ERP                                    // 利用产品名定义命名空间
{
}
```

- 接口的名称加前缀“I”。

例如，创建一个公共接口 Iconvertible，代码如下。

```
public  interface  Iconvertible          // 创建一个公共接口 Iconvertible
{
     byte ToByte();                      // 声明一个 byte 类型的方法
}
```

- 类的命名最好能够体现出类的功能或操作。

例如，创建一个名称为 Operation 的类，用来作为运算类，代码如下。

```
public  class  Operation                 // 表示一个运算类
{
}
```

- 方法的命名：一般将其命名为动宾短语，表明该方法的主要作用。

例如，在公共类 File 中创建 CreateFile 方法和 GetPath 方法，代码如下。

```
public  class  File                                      // 创建一个公共类
{
    public  void  CreateFile(string  filePath)           // 创建一个 CreateFile 方法
    {
    }
    public  void  GetPath(string  path)                  // 创建一个 GetPath 方法
    {
    }
}
```

- 在定义成员变量时，最好加前缀“_”。

例如，在公共类 DataBase 中声明一个私有成员变量 _connectionString，代码如下。

```
public  class  DataBase                                  // 创建一个公共类
{
    private  string  _connectionString;                  // 声明一个私有成员变量
}
```

第 3 章　必须学会的 C# 语法

很多人认为在学习 C# 之前必须要学习 C++，其实并非如此。产生这种错误认识的原因是很多人在学习 C# 之前都学习过 C++。事实上，C# 比 C++ 更容易掌握。要掌握并熟练应用 C#，就需要对 C# 的基础语法进行充分了解。本章将对 C# 的基础语法进行详细讲解，对于初学者来说，应该对本章的各个小节进行仔细阅读和深入思考，这样才能达到事半功倍的效果。

3.1　为什么要使用变量

变量关系到数据的存储。计算机是使用内存来存储计算时所使用的数据的，那么内存是如何存储数据的呢？数据有多种类型，如整数、小数、字符串等。在内存中存储这些数据时，首先需要根据数据的需求（类型）为它申请一块合适的空间，然后在这个空间中存储相应的值。实际上，内存就像一家宾馆，如果客人到一家宾馆住宿，则首先需要开房间，然后才能入住；而在开房间时，客人需要选择是开单间、开双人间还是开总统套房等，这其实就对应一个变量的数据类型选择问题。

由于内存中的数据是以二进制格式进行存储的，而这些二进制数据都有相应的内存地址，因此，在内存中为数据分配一定的空间之后，如果要使用定义的这个数据，则还需要通过一种技术使用户能够很方便地访问二进制数据的内存地址，这种技术就是变量！

3.2　变量是什么

变量主要用来存储特定类型的数据，用户可以根据需要随时改变变量中所存储的数据值。变量具有名称、类型和值，其中：名称是变量在程序源代码中的标识；类型用来确定变量所代表的内存的大小和类型；值是指它所代表的内存块中的数据。在程序的执行过程中，变量的值可以发生变化。在使用变量之前必须声明变量，即指定变量的类型和名称。

这里以客人入住宾馆为例，说明一个变量所需要的基本要素。首先，客人需要选择房间类型，也就是确定变量类型的过程；选择房间类型后，客人需要选择房间号，这相当于确定变量的名称；完成以上操作后，这个客人就可以顺利入住，这样这个客人就相当于这个房间中存储的数据。变量的基本要素示意图如图 3.1 所示。

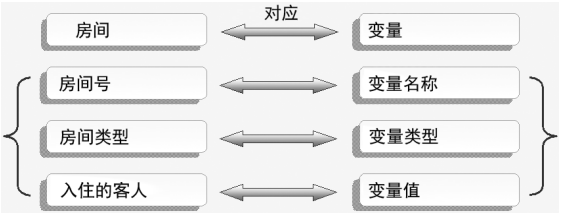


图 3.1　变量的基本要素示意图

3.3　变量的声明及初始化

在使用变量时，首先需要对变量进行命名。对变量命名的过程，其实就是声明一个变量的过程。在使用变量之前，必须对其进行声明并初始化。本节将对变量的声明、简单数据类型、变量的初始化，以及变量的作用域进行详细讲解。

3.3.1　变量的声明

1. 声明变量

声明变量就是指定变量的名称和类型。变量的声明非常重要，未经声明的变量本身并不合法，也无法在程序中使用。在 C# 中，声明一个变量是由一个类型和跟在后面的一个或多个变量名组成的，多个变量之间用逗号分开，声明变量以分号结束，语法如下。

```
变量类型  变量名 ;                                    // 声明一个变量
变量类型  变量名 1, 变量名 2,…变量名 n;                 // 同时声明多个变量
```

例如，声明一个整型变量 mr，再同时声明 3 个字符串变量 mr_1、mr_2 和 mr_3，代码如下。

```
01  int mr;                                          // 声明一个整型变量
02  string mr_1, mr_2, mr_3;                         // 同时声明 3 个字符串变量
```

2. 变量的命名规则

在声明变量时，要注意变量的命名规则。C# 的变量名是一种标识符，应该符合标识符的命名规则。另外，需要注意的一点是，C# 中的变量名是区分大小写的，比如 num 和

Num 是两个不同的变量，在程序中使用时是有区别的。下面列出了变量的命名规则。

- 变量名只能由数字、字母和下画线组成。
- 变量名的第一个符号只能是字母或下画线，不能是数字。
- 不能使用 C# 中的关键字作为变量名。
- 一旦在一个语句块中定义了一个变量名，那么在变量的作用域内不能再定义同名的变量。

例如，下面的变量名是正确的。

```
city
_money
money_1
```

下面的变量名是不正确的。

```
123
2word
int
```

学习笔记

在 C# 中允许使用汉字或其他语言文字作为变量名，如“int 年龄 = 21”，在程序运行时并不会出现错误，但建议读者尽量不要使用这些语言文字作为变量名。

3.3.2 简单数据类型

前面提到，在声明变量时，首先需要确定变量的类型，那么，开发人员可以使用哪些变量类型呢？实际上，可以使用的变量类型是无限的，因为开发人员可以通过自定义类型存储各种数据，但这里要讲解的简单数据类型是 C# 中预定义的一些类型。

C# 中的数据类型根据其定义可以分为两种：一种是值类型；另一种是引用类型。从概念上看，值类型是直接存储值，而引用类型存储的是对值的引用。C# 中的数据类型结构如图 3.2 所示。

由图 3.2 可以看出，值类型主要包括简单类型和复合类型两种，其中：简单类型是程序中使用的基本类型，主要包括整数类型、浮点类型、布尔类型和字符类型 4 种，这 4 种简单类型都是 .NET 中预定义的；复合类型主要包括枚举类型和结构类型，这两种复合类型既可以是 .NET 中预定义的，也可以由用户自定义。本节主要对简单类型进行详细讲解，简单类型在实际中的应用如图 3.3 所示。

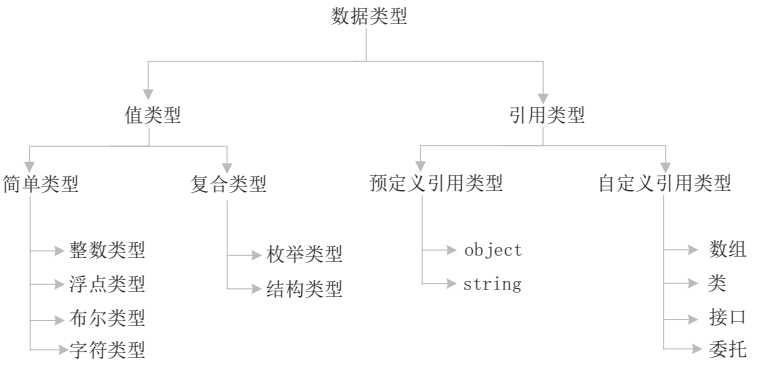


图 3.2　C# 中的数据类型结构

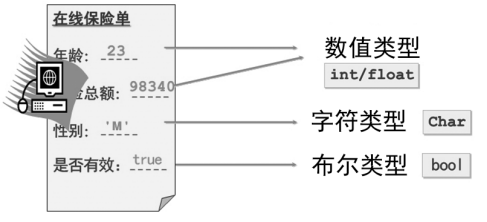


图 3.3　简单类型在实际中的应用

1. 整数类型

整数类型用来存储整数数值，即没有小数部分的数值。可以是正数，也可以是负数。整数类型数据在 C# 程序中有 3 种表现形式，分别为十进制、八进制和十六进制。

（1）十进制：十进制的表现形式比较常见，如 120、0、-127。

学习笔记

不能以 0 作为十进制数的开头。

（2）八进制：以 0 开头的数，如 0123（转换成十进制数为 83）、-0123（转换成十进制数为 -83）。

学习笔记

八进制数必须以 0 开头。

（3）十六进制：以 0x/0X 开头的数，如 0x25（转换成十进制数为 37）、0Xb01e（转换成十进制数为 45086）。

学习笔记

十六进制数必须以 0x 或 0X 开头。

C# 中内置的整数类型如表 3.1 所示。

表 3.1　C# 中内置的整数类型

类　型	说明（8 位等于 1 字节）	范　围
sbyte	8 位有符号整数	-128 ～ 127
short	16 位有符号整数	-32768 ～ 32767
int	32 位有符号整数	-2147483648 ～ 2147483647
long	64 位有符号整数	-9223372036854775808 ～ 9223372036854775807
byte	8 位无符号整数	0 ～ 255
ushort	16 位无符号整数	0 ～ 65535
uint	32 位无符号整数	0 ～ 4294967295
ulong	64 位无符号整数	0 ～ 18446744073709551615

学习笔记

表 3.1 中出现了“有符号 **”和“无符号 **”，其中，“无符号 **”是在“有符号 **”类型的前面加了一个 u，这里的 u 是“unsigned”的缩写。它们的主要区别是：“有符号 **”既可以存储正数，也可以存储负数；“无符号 **”只能存储不带符号的整数，因此，它只能存储正数。例如，下面的代码：

```
int i = 10;                    // 正确
int j = -10;                   // 正确
uint m = 10;                   // 正确
uint n = -10;                  // 错误
```

例如，定义一个 int 类型的变量 i 和一个 byte 类型的变量 j，并分别赋值为 2017 和 255，代码如下。

```
int i = 2017;                  // 声明一个 int 类型的变量 i
byte j = 255;                  // 声明一个 byte 类型的变量 j
```

此时，如果将 byte 类型的变量 j 赋值为 256，即将代码修改如下：

```
int i = 2017;                  // 声明一个 int 类型的变量 i
byte j = 256;                  // 将 byte 类型的变量 j 的值修改为 256
```

在 Visual Studio 中编译程序则会出现如图 3.4 所示的错误提示信息。

图 3.4　取值超出指定类型的范围时出现的错误提示信息

分析图 3.4 中出现的错误提示信息，主要是由于 byte 类型的变量是 8 位无符号整数，它的取值范围为 0 ～ 255，而“256”这个值已经超出了 byte 类型的变量的取值范围，所以编译程序会出现错误提示信息。

学习笔记

整数类型变量的默认值为 0。

2. 浮点类型

浮点类型变量主要用于处理含有小数的数据。浮点类型主要包含 float 和 double 两种类型，表 3.2 列出了这两种浮点类型的描述信息。

表 3.2　两种浮点类型的描述信息

类　型	说　明	取 值 范 围
float	精确到 7 位数	-3.4×10^{38} ～ 3.4×10^{38}
double	精确到 15 或 16 位数	-1.79×10^{308} ～ 1.79×10^{308}

如果不做任何设置，则包含小数点的数值都被认为是 double 类型，如 9.27，在没有特别指定的情况下，这个数值是 double 类型。如果要将数值以 float 类型来处理，就应该通过强制使用 f 或 F 将其指定为 float 类型。

例如，下面的代码就是将数值强制指定为 float 类型。

```
01  float theMySum = 9.27f;                  // 使用 f 将数值强制指定为 float 类型
02  float theMuSums = 1.12F;                 // 使用 F 将数值强制指定为 float 类型
```

如果要将数值强制指定为 double 类型，则应该使用 d 或 D 进行设置，但加不加“d”或“D”没有硬性规定，可以加，也可以不加。

例如，下面的代码就是将数值强制指定为 double 类型。

```
01  double myDou = 927d;                     // 使用 d 将数值强制指定为 double 类型
02  double mudou = 112D;                     // 使用 D 将数值强制指定为 double 类型
```

学习笔记

当需要使用 float 类型变量时，必须在数值的后面跟随 f 或 F，否则编译器会直接将其作为 double 类型处理。另外，可以在 double 类型的值前面加上“(float)”对其进行强制转换。浮点类型变量的默认值是 0，而不是 0.0。

3. decimal 类型

decimal 类型表示 128 位数据类型，它是一种精度更高的浮点类型，其精度可以达到 28 位，取值范围为 -7.9×10^{28} ～ 7.9×10^{28}。

学习笔记

由于 decimal 类型的高精度特性，它更适用于财务和货币计算。

如果希望一个小数被当成 decimal 类型使用，则需要使用后缀 m 或 M，如下所示。

```
decimal myMoney = 1.12m;
```

如果小数没有后缀 m 或 M，则数值将被视为 double 类型，从而导致编译器错误。例如，在开发环境中运行下面的代码，将会出现如图 3.5 所示的错误提示信息。

```
01  static void Main(string[] args)
02  {
03      decimal d = 3.14;
04      Console.WriteLine(d);
05  }
```

图 3.5　不加后缀 m 或 M 时编译器出现的错误提示信息

由图 3.5 可以看出，“3.14”这个数值如果没有后缀，就会直接被当成 double 类型，所以在赋值为 decimal 类型的变量时就会出现错误提示信息。

示例 1．根据身高和体重计算 BMI 指数

创建一个控制台应用程序，声明 double 类型变量 height 来记录身高，单位为米；声明 int 类型变量 weight 来记录体重，单位为千克。根据“BMI = 体重 ÷(身高 × 身高)”计算 BMI 指数（身体质量指数），代码如下。

```
01  static void Main(string[] args)
02  {
03      double height = 1.78;                                  // 身高变量，单位为米
04      int weight = 75;                                       // 体重变量，单位为千克
05      double exponent = weight / (height * height); // BMI 计算公式
06      Console.WriteLine(" 您的身高为：" + height);
07      Console.WriteLine(" 您的体重为：" + weight);
08      Console.WriteLine(" 您的 BMI 指数为：" + exponent);
09      Console.Write(" 您的体重属于：");
10      if (exponent < 18.5)
11      {// 判断 BMI 指数是否小于 18.5
12          Console.WriteLine(" 体重过轻 ");
13      }
14      else if (exponent >= 18.5 && exponent < 24.9)
15      {// 判断 BMI 指数是否大于或等于 18.5，并且小于 24.9
16          Console.WriteLine(" 正常范围 ");
17      }
18      else if (exponent >= 24.9 && exponent < 29.9)
19      {// 判断 BMI 指数是否大于或等于 24.9，并且小于 29.9
20          Console.WriteLine(" 体重过重 ");
21      }
22      else if (exponent >= 29.9)
23      {// 判断 BMI 指数是否大于或等于 29.9
24          Console.WriteLine(" 肥胖 ");
25      }
26      Console.ReadLine();
27  }
```

学习笔记

第 10、14、18 和 22 行代码使用了 if...else if 条件判断语句，该语句主要用来判断是否满足某种条件，我们将在第 4 章对其进行详细讲解，这里只需要了解即可。

根据身高和体重计算 BMI 指数代码的运行结果如图 3.6 所示。

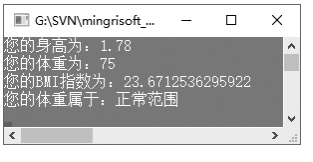


图 3.6　根据身高和体重计算 BMI 指数代码的运行结果

4. 布尔类型

布尔类型主要用来表示 true 值或 false 值，在 C# 中定义布尔类型时，需要使用布尔关键字。例如，下面代码定义一个布尔类型变量。

```
bool x = true;
```

学习笔记

布尔类型通常被用在流程控制语句中作为判断条件。

这里需要注意的是，布尔类型变量的值只能是 true 或 false，不能将其他的值指定给布尔类型变量。例如，将一个整数 10 赋值给布尔类型变量，代码如下。

```
bool x = 10;
```

在 Visual Studio 中运行这行代码，会出现如图 3.7 所示的错误提示信息。

图 3.7　将整数赋值给布尔类型变量时出现的错误提示信息

学习笔记

布尔类型变量的默认值为 false。

5. 字符类型

字符类型在 C# 中使用 Char 类来表示，该类主要用来存储单个字符，它占用 16 位（2 字节）的内存空间。在定义字符类型变量时，要以单引号（' '）表示，如 'a' 表示一个字符，而 "a" 则表示一个字符串。虽然其只有一个字符，但由于使用双引号，所以它仍然表示字符串，而不是字符。字符类型变量的声明非常简单，代码如下。

```
01  Char ch1 = 'L';
02  char ch2 = '1';
```

学习笔记

Char 类只能定义一个 Unicode 字符。Unicode 字符是目前计算机中通用的字符编码，它为针对不同语言中的每个字符设定了统一的二进制编码，用于满足跨语言、跨平台的文本转换和处理要求，这里了解 Unicode 即可。

1）Char 类的使用

Char 类为开发人员提供了许多方法，可以通过使用这些方法灵活地对字符进行各种

操作。Char 类的常用方法及说明如表 3.3 所示。

表 3.3　Char 类的常用方法及说明

方　法	说　明
IsDigit	判断某个 Unicode 字符是否属于十进制数字类别
IsLetter	判断某个 Unicode 字符是否属于字母类别
IsLetterOrDigit	判断某个 Unicode 字符是属于字母类别还是属于十进制数字类别
IsLower	判断某个 Unicode 字符是否属于小写字母类别
IsNumber	判断某个 Unicode 字符是否属于数字类别
IsPunctuation	判断某个 Unicode 字符是否属于标点符号类别
IsSeparator	判断某个 Unicode 字符是否属于分隔符类别
IsUpper	判断某个 Unicode 字符是否属于大写字母类别
IsWhiteSpace	判断某个 Unicode 字符是否属于空白类别
Parse	将指定字符串的值转换为它的等效 Unicode 字符
ToLower	将 Unicode 字符的值转换为它的小写等效项
ToString	将字符的值转换为其等效的字符串
ToUpper	将 Unicode 字符的值转换为它的大写等效项
TryParse	将指定字符串的值转换为它的等效 Unicode 字符

从表 3.3 中可以看到，C# 中的 Char 类提供了很多操作字符的方法，其中，以“Is”和“To”开始的方法比较常用。以“Is”开始的方法大多是判断 Unicode 字符是否为某个类别，如是否为大 / 小写、是否是数字等；而以“To”开始的方法主要用来对字符进行大小写转换及字符串转换的操作。

示例 2．Char 类的常用方法的应用

创建一个控制台应用程序，演示如何使用 Char 类提供的常用方法，代码如下。

```
01 static void Main(string[] args)
02 {
03     char a = 'a';                                    // 声明字符 a
04     char b = '8';                                    // 声明字符 b
05     char c = 'L';                                    // 声明字符 c
06     char d = '.';                                    // 声明字符 d
07     char e = '|';                                    // 声明字符 e
08     char f = ' ';                                    // 声明字符 f
09     // 使用 IsLetter 方法判断 a 是否为字母
10     Console.WriteLine("IsLetter 方法判断 a 是否为字母：{0}", Char.IsLetter(a));
11     // 使用 IsDigit 方法判断 b 是否为数字
12     Console.WriteLine("IsDigit 方法判断 b 是否为数字：{0}", Char.IsDigit(b));
```

```
13        // 使用 IsLetterOrDigit 方法判断 c 是否为字母或数字
14        Console.WriteLine("IsLetterOrDigit 方法判断 c 是否为字母或数字：{0}", Char.
IsLetterOrDigit(c));
15        // 使用 IsLower 方法判断 a 是否为小写字母
16        Console.WriteLine("IsLower 方法判断 a 是否为小写字母：{0}", Char.IsLower(a));
17        // 使用 IsUpper 方法判断 c 是否为大写字母
18        Console.WriteLine("IsUpper 方法判断 c 是否为大写字母：{0}", Char.IsUpper(c));
19        // 使用 IsPunctuation 方法判断 d 是否为标点符号
20        Console.WriteLine("IsPunctuation 方法判断 d 是否为标点符号：{0}", Char.
IsPunctuation(d));
21        // 使用 IsSeparator 方法判断 e 是否为分隔符
22        Console.WriteLine("IsSeparator 方法判断 e 是否为分隔符：{0}", Char. IsSeparator(e));
23        // 使用 IsWhiteSpace 方法判断 f 是否为空白
24        Console.WriteLine("IsWhiteSpace 方法判断 f 是否为空白：{0}", Char. IsWhiteSpace(f));
25        Console.ReadLine();
26    }
```

学习笔记

第 3 ~ 8 行代码声明了 5 个不同类型的字符变量，下面的操作都是围绕这 5 个字符变量进行的。

第 25 行代码主要是为了使控制台界面能够停留在桌面上。

运行上面代码，得到如图 3.8 所示的内容。

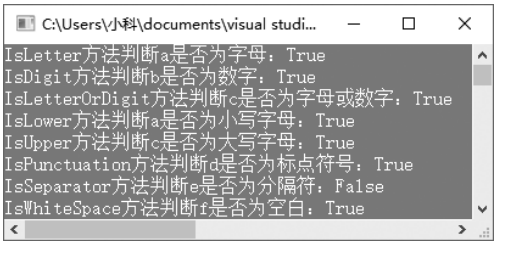


图 3.8　Char 类的常用方法的应用

2）转义字符

前面讲到了字符类型只能存储单个字符，但是，如果在 Visual Studio 中编写如下代码，则会出现如图 3.9 所示的错误提示信息。

```
char ch = '\';
```

从代码表面上看，反斜线“\”是一个字符，在正常情况下，它应该是可以定义为字符的，但为什么会出现错误呢？这里就需要了解转义字符的概念。

图 3.9　定义反斜线时出现的错误提示信息

转义字符是一种特殊的字符变量，以反斜线“\”开头，后跟一个或多个字符。也就是说，在 C# 中，反斜线“\”是一个转义字符，不能单独作为字符使用。因此，如果要在 C# 中使用反斜线，则可以使用下面的代码表示。

```
char ch = '\\';
```

转义字符相当于一个电源变换器。电源变换器通过一定的手段获得所需要的电源形式，如交流变成直流、高电压变为低电压、低频变为高频等。转义字符的功能与其类似，它可以将字符转换成另一种操作形式，或者将无法一起使用的字符进行组合。

学习笔记

转义字符“\”只针对后面紧跟着的单个字符进行操作。

C# 中的常用转义字符及其作用如表 3.4 所示。

表 3.4　C# 中的常用转义字符及其作用

转 义 字 符	作　　用
\n	回车换行
\t	横向跳到下一制表位置
\"	双引号
\b	退格
\r	回车
\f	换页
\\	反斜线符
\'	单引号符
\uxxxx	使用 4 位数的十六进制值所表示的字符，如 \u0052

示例 3．输出 Windows 的系统目录

创建一个控制台应用程序，通过使用转义字符在控制台窗口中输出 Windows 的系统目录，代码如下。

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

```
02  {
03      Console.WriteLine("Windows 的系统目录为：C:\\Windows");// 输出 Windows 的系统目录
04      Console.ReadLine();
05  }
```

输出 Windows 的系统目录代码的运行结果如图 3.10 所示。

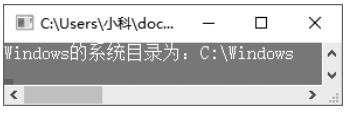


图 3.10　输出 Windows 的系统目录代码的运行结果

学习笔记

在输出系统目录时，如果遇到反斜杠，则使用“\\”表示。但是，如果有多级目录，在遇到反斜杠时，如果都使用“\\”，则会显得非常麻烦。如果遇到下面这种情况：

```
Console.WriteLine("C:\\Windows\\Microsoft.NET\\Framework\\v4.0.30319\\2052");
```

则可以用一个 @ 符号进行多级转义，代码修改如下。

```
Console.WriteLine(@"C:\Windows\Microsoft.NET\Framework\v4.0.30319\2052");
```

3.3.3　变量的初始化

变量的初始化实际上就是给变量赋值，以便在程序中使用。在 Visual Studio 2017 中运行下面一段代码。

```
01  static void Main(string[] args)
02  {
03      string title;
04      Console.WriteLine(title);
05  }
```

在运行上面这段代码时，会出现如图 3.11 所示的错误提示信息。

图 3.11　变量未赋值时出现的错误提示信息

从图 3.11 中可以看出，如果在使用变量时直接定义一个变量，则会提示使用了未赋值的变量，这说明在程序中使用变量时一定要对其进行赋值，也就是初始化，然后才可以使用。那么如何对变量进行初始化呢？

初始化变量有 3 种方法，分别是单独初始化变量、声明时初始化变量、同时初始化多个变量，下面分别进行讲解。

1. 单独初始化变量

在 C# 中，使用赋值运算符“=”（等号）对变量进行初始化，即将等号右边的值赋给左边的变量。

例如，声明一个变量 sum 并初始化，其默认值为 2017，代码如下。

```
01  int sum;                          // 声明一个变量
02  sum = 2017;                       // 使用赋值运算符“=”给变量赋值
```

学习笔记

在对变量进行初始化时，等号右边也可以是一个已经被赋值的变量。例如，首先声明两个变量 sum 和 num，然后将变量 sum 赋值为 2017，最后将变量 sum 赋值给变量 num，代码如下。

```
01  int sum, num;                     // 声明两个变量
02  sum = 2017;                       // 将变量 sum 赋值为 2017
03  num = sum;                        // 将变量 sum 赋值给变量 num
```

2. 在声明时初始化变量

在声明变量时可以对变量进行初始化，即在每个变量名后面加上给变量赋初始值的指令。

例如，先声明一个整型变量 mr 并赋值为 927，然后同时声明 3 个字符串类型的变量并初始化，代码如下。

```
01  int mr = 927;                     // 初始化整型变量 mr
02  // 初始化字符串类型的变量 mr_1、mr_2 和 mr_3
03  string mr_1 = " 零基础学 ", mr_2 = " 项目入门 ", mr_3 = " 实例精粹 ";
```

3. 同时初始化多个变量

在对多个同类型的变量赋同一个值时，为了节省代码的行数，可以同时对多个变量进行初始化。

例如，声明 5 个 int 类型的变量 a、b、c、d、e，然后将这 5 个变量都初始化为 0，代码如下。

```
01  int a, b, c, d, e;
```

```
02  a = b = c = d = e = 0;
```

上面讲解了初始化变量的 3 种方法，我们可以利用这些方法对本节开始的代码段进行修改，使其能够正常运行。修改后的代码如下。

```
01  static void Main(string[] args)
02  {
03      // 第一种方法
04      //string title=" 零基础学 C#";
05      // 第二种方法
06      string title;
07      title = " 零基础学 C#";
08      Console.WriteLine(title);
09  }
```

再次运行代码，即可正常运行。

3.3.4 变量的作用域

由于变量被定义后只是暂时存储在内存中，等程序执行到某个点后该变量会被释放，也就是说变量有生命周期，因此，变量的作用域是指程序代码能够访问该变量的区域。如果超出该区域，则在编译时会出现错误。在程序中，一般会根据变量的有效范围将变量分为成员变量和局部变量。

1. 成员变量

在类体中定义的变量称为成员变量，成员变量在整个类中都有效。类的成员变量又可以分为两种，即静态变量和实例变量。

例如，在 Test 类中声明静态变量和实例变量，代码如下。

```
01  class Test
02  {
03      int x = 45;
04      static int y = 90;
05  }
```

在该段代码中，x 为实例变量，y 为静态变量（也称为类变量）。如果在成员变量的类型前面加上关键字 static，则这样的成员变量称为静态变量。静态变量的有效范围可以跨类，甚至可覆盖整个应用程序。对于静态变量，除了能在定义它的类内存、取，还能直接以“类名 . 静态变量”的方式在其他类内使用。

2. 局部变量

在类的方法体中定义的变量（定义方法的“{”与“}”之间的区域）称为局部变量，

局部变量只在当前代码块中有效。

在类的方法中声明的变量（包括方法的参数）都属于局部变量。局部变量只在当前定义的方法内有效，而不能用在类的其他方法中。局部变量的生命周期取决于方法：当方法被调用时，C# 编译器为方法中的局部变量分配内存空间；当该方法的调用结束后，则会释放方法中局部变量占用的内存空间，局部变量也会被销毁。

变量的有效范围如图 3.12 所示。

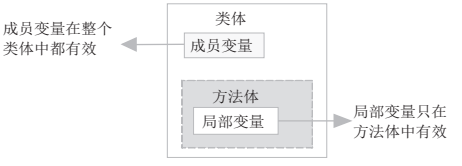


图 3.12　变量的有效范围

示例 4. 使用变量记录用户的登录名

创建一个控制台应用程序，使用一个局部变量记录用户的登录名，代码如下。

```
01  static void Main(string[] args)
02  {
03      Console.WriteLine("     欢迎进入明日科技官网 \n\n     请首先输入用户名：");
04      string Name = Console.ReadLine();                    // 记录用户的输入
05      Console.WriteLine("     登录用户：" + Name);          // 输出当前登录用户
06      Console.ReadLine();
07  }
```

使用变量记录用户的登录名代码的运行结果如图 3.13 所示。

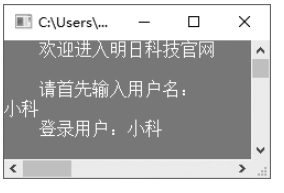


图 3.13　使用变量记录用户的登录名代码的运行结果

3.4　常量

通过对前面的学习，我们知道了变量是随时可以改变值的量，那么，在遇到不允许改变值的情况时，该怎么办呢？这就需要用到本节要讲解的常量。

3.4.1 常量是什么

常量就是在程序运行过程中，值不能改变的量。比如，现实生活中的居民身份证号码、数学运算中的 π 值等，这些都不会发生改变，它们都可以定义为常量。常量可以区分为不同的类型。例如，98、368 是整型常量；3.14、0.25 是实数常量，即浮点类型的常量；'m'、'r' 是字符常量。

3.4.2 常量的分类

常量主要有两种，分别是 const 常量和 readonly 常量，下面分别对这两种常量进行讲解。

1. const 常量

在 C# 中提到常量，通常指的是 const 常量。const 常量也称为静态常量，它在编译时值就已经确定了。const 常量的值必须在声明时就进行初始化，而且之后不可以进行更改。

例如，声明一个正确的 const 常量，同时再声明一个错误的 const 常量，以便读者对比参考，代码如下。

```
01  const double PI = 3.1415926;          // 正确的声明方法
02  const int MyInt;                      // 错误，定义常量时没有初始化
```

2. readonly 常量

readonly 常量是一种特殊的常量，也称为动态常量。从字面上理解，readonly 常量可以进行动态赋值，但需要注意的是，这里的动态赋值是有条件的，它只能在构造函数中进行赋值，例如下面的代码。

```
01  class Program
02  {
03      readonly int Price;               // 定义一个 readonly 常量
04      Program()                         // 构造函数
05      {
06          Price = 368;                  // 在构造函数中修改 readonly 常量的值
07      }
08      static void Main(string[] args)
09      {
10      }
11  }
```

在构造函数以外的位置修改 readonly 常量的值，比如，在 Main 方法中进行修改，代码如下。

```
01 class Program
02  {
03      readonly int Price;              // 定义一个 readonly 常量
04      Program()                        // 构造函数
05      {
06          Price = 368;                 // 在构造函数中修改 readonly 常量的值
07      }
08      static void Main(string[] args)
09      {
10          Program p = new Program();   // 创建类的对象
11          p.Price = 365;               // 试图对 readonly 常量的值进行修改
12      }
13  }
```

这时再运行程序，将会出现如图 3.14 所示的错误提示信息。

图 3.14　在构造函数以外的位置修改 readonly 常量的值时出现的错误提示信息

3. const 常量与 readonly 常量的区别

const 常量与 readonly 常量的主要区别如下。

（1）const 常量必须在声明时初始化，而 readonly 常量则可以在构造函数中初始化。

（2）const 常量在编译时就被解析，即将常量的值替换成初始化的值，而 readonly 常量的值需要在运行时确定。

（3）const 常量可以定义在类中或方法体中，而 readonly 常量只能定义在类中。

3.5　运算符

运算符是具有运算功能的符号。根据使用运算符的个数，可以将运算符分为单目运算符、双目运算符和三目运算符，其中：单目运算符是作用在一个操作数上的运算符，如正号（+）等；双目运算符是作用在两个操作数上的运算符，如加法（+）、乘法（*）等；三目运算符是作用在 3 个操作数上的运算符，C# 中唯一的三目运算符就是条件运算符（?:）。本节将详细讲解 C# 中的运算符。

3.5.1 算术运算符

C# 中的算术运算符是双目运算符，主要包括+、-、*、/ 和 % 5 种，它们分别用于进行加、减、乘、除和模（求余数）运算。C# 中的算术运算符的功能及使用方式如表 3.5 所示。

表 3.5 C# 中的算术运算符的功能及使用方式

运 算 符	说 明	实 例	结 果
+	加	12.45f+15	27.45
-	减	4.56-0.16	4.4
*	乘	5L*12.45f	62.25
/	除	7/2	3
%	模	12%10	2

示例 5．计算学生成绩的分数之差及平均分

某学员的 3 门课的成绩：C 语言为 89 分、C# 为 90 分、SQL 为 60 分，编程实现如下计算。

（1）C# 课和 SQL 课的分数之差。

（2）3 门课的平均分。

代码如下。

```
01 static void Main(string[] args)
02 {
03  int c = 89, csharp = 90, sql = 60;      // 定义 3 个变量，分别存储 C 语言、C# 和 SQL 的分数
04  int sub = csharp - sql;                  // 计算 C# 课和 SQL 课的分数之差
05  double avg = (c + csharp + sql) / 3;     // 计算平均分
06  Console.WriteLine("C# 课和 SQL 课的分数之差：" + sub + " 分 ");
07  Console.WriteLine("3 门课的平均分：" + avg + " 分 ");
08  Console.ReadLine();
09 }
```

计算学生成绩的分数之差及平均分代码的运行结果如图 3.15 所示。

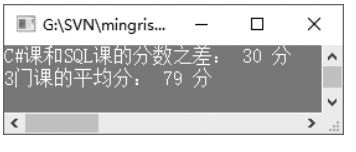


图 3.15 计算学生成绩的分数之差及平均分代码的运行结果

学习笔记

在使用除法运算符和模运算符时，除数不能为 0，否则将会出现异常，如图 3.16 所示。

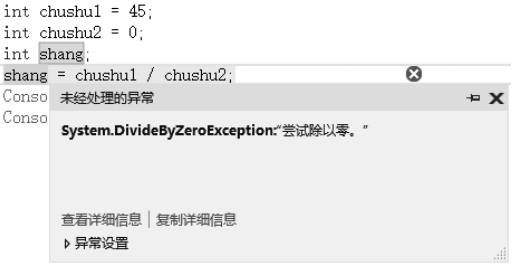


图 3.16　除数为 0 时出现的异常

3.5.2　自增运算符和自减运算符

在使用算术运算符时，如果需要对数值型变量的值进行加 1 或减 1 操作，则可以使用下面的代码。

```
01  int i=5;
02  i=i+1;
03  i=i-1;
```

针对以上功能，C# 中还提供了另外的实现方式：自增运算符、自减运算符，它们分别用 ++ 和 -- 表示，下面分别对它们进行讲解。

自增运算符和自减运算符是单目运算符，在使用时有两种形式，分别是 ++expr、expr++，或者 --expr、expr--。其中，++expr、--expr 是前置形式，它表示 expr 自身先加 1 或减 1，其运算结果是自身修改后的值，再参与其他运算。而 expr++、expr-- 是后置形式，它也表示自身加 1 或减 1，但其运算结果是自身未修改的值。也就是说，expr++、expr-- 是先参加完其他运算，然后进行自身加 1 或减 1 操作。自增运算符和自减运算符放在不同位置时的运算示意图如图 3.17 所示。

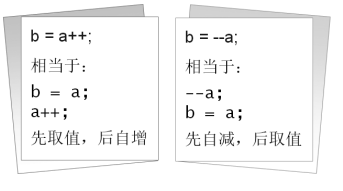


图 3.17　自增运算符和自减运算符放在不同位置时的运算示意图

例如，下面代码演示自增运算符放在变量的不同位置时的运算结果。

```
01  int i = 0, j = 0;          // 定义 int 类型的 i、j
02  int post_i, pre_j;         // post_i 表示后置形式运算的返回结果，pre_j 表示前置形式运算的返回结果
03  post_i = i++;              // 后置形式的自增，post_i 是 0
04  Console.WriteLine(i);      // 输出结果是 1
05  pre_j = ++j;               // 前置形式的自增，pre_j 是 1
06  Console.WriteLine(j);      // 输出结果是 1
```

学习笔记

自增运算符、自减运算符只能作用于变量，因此，下面的形式是不合法的。

```
01      3++;                   // 不合法，因为 3 是一个常量
02      (i+j)++;               // 不合法，因为 i+j 是一个表达式
```

学习笔记

如果程序中不需要使用操作数原来的值，只是需要其自身进行加（减）1，那么建议使用前置自加（减），因为后置自加（减）必须先保存原来的值，而前置自加（减）不需要保存原来的值。

3.5.3 赋值运算符

赋值运算符主要用来为变量等赋值，它是双目运算符。C# 中的赋值运算符分为简单赋值运算符和复合赋值运算符，下面分别对其进行讲解。

1. 简单赋值运算符

简单赋值运算符以符号“=”表示，其功能是将右操作数所含的值赋给左操作数。例如：

```
int a = 100;                                    // 该表达式是将 100 赋值给变量 a
```

2. 复合赋值运算符

在程序中对某个对象进行某种操作后，如果要将操作结果重新赋值给该对象，则可以通过下面的代码实现。

```
01  int a = 3;
02  int temp = 0 ;
03  temp = a + 2 ;
04  a= temp ;
```

上面的代码看起来很烦琐，在 C# 中，上面的代码等价于：

```
01  int a = 3;
```

```
02  a += 2;
```

上面代码中的 += 就是一种复合赋值运算符。复合赋值运算符又称为带运算的赋值运算符，它其实是将赋值运算符与其他运算符合并成一个运算符来使用，从而同时实现两种运算符的效果的。

C# 提供了很多复合赋值运算符，其说明及运算规则如表 3.6 所示。

表 3.6　复合赋值运算符的说明及运算规则

名　称	运　算　符	运 算 规 则	意　义
加赋值	+=	x+=y	x=x+y
减赋值	-=	x-=y	x=x-y
除赋值	/=	x/=y	x=x/y
乘赋值	*=	x*=y	x=x*y
模赋值	%=	x%=y	x=x%y
位与赋值	&=	x&=y	x=x&y
位或赋值	|=	x|=y	x=x|y
右移赋值	>>=	x>>=y	x=x>>y
左移赋值	<<=	x<<=y	x=x<<y
异或赋值	^=	x^=y	x=x^y

3. 复合赋值运算符的优势及劣势

在使用复合赋值运算符时，虽然“a += 1”与“a = a + 1”两者的运算结果是相同的，但是在不同的场景下，两种使用方法都有各自的优势和劣势，下面分别对其进行介绍。

1）低精度类型自增

在 C# 中，因为整数的默认类型为 int 型，所以下面的代码会报错。

```
01  byte a=1;          // 创建 byte 型变量 a
02  a=a+1;             // 让 a 的值 +1，错误提示：无法将 int 型转换成 byte 型
```

在上面的代码中，在没有进行强制类型转换的条件下，a+1 的结果是一个 int 型的值，无法直接赋给一个 byte 型变量。但是如果使用“+=”实现递增计算，就不会出现这种问题，代码如下。

```
01  byte a=1;          // 创建 byte 型变量 a
02  a+=1;              // 让 a 的值 +1
```

2）不规则的多值运算

复合赋值运算符虽然简洁、强大，但是有些时候是不推荐使用的。例如，下面的代码。

```
a = (2 + 3 - 4) * 92 / 6;
```

上面的代码如果改用复合赋值运算符实现，就会显得非常烦琐，代码如下。

```
01  a += 2;
02  a += 3;
03  a -= 4;
04  a *= 92;
05  a /= 6;
```

学习笔记

在 C# 中可以把赋值运算符连在一起使用。例如下面的代码，在这段代码中，变量 x、y、z 都得到同样的值 5，但在程序开发中不建议使用这种赋值语法。

```
x = y = z = 5;
```

3.5.4 关系运算符

关系运算符是双目运算符，它用于程序中的变量之间及其他类型的对象之间的比较，它返回一个代表运算结果的布尔值。当关系运算符对应的关系成立时，运算结果为 true；否则为 false。关系运算符通常用在条件语句中来作为判断的依据。C# 中的关系运算符有 6 个，其作用及说明如表 3.7 所示。

表 3.7　C# 中的关系运算符的作用及说明

运　算　符	作　　用	举　　例	操 作 数 据	结　　果
>	大于	'a'>'b'	整型、浮点型、字符型	false
<	小于	156 < 456	整型、浮点型、字符型	false
==	等于	'c'=='c'	基本数据类型、引用型	true
!=	不等于	'y'!='t'	基本数据类型、引用型	true
>=	大于或等于	479>=426	整型、浮点型、字符型	true
<=	小于或等于	12.45<=45.5	整型、浮点型、字符型	true

学习笔记

不等于运算符（!=）是与等于运算符相反的运算符，它与 !(a==b) 是等效的。

示例 6．使用关系运算符比较大小关系

创建一个控制台应用程序，声明 3 个 int 类型的变量，并分别对它们进行初始化，然后分别使用 C# 中的各种关系运算符对它们的大小关系进行比较，代码如下。

```
01  static void Main(string[] args)
02  {
03      int num1 = 4, num2 = 7, num3 = 7; // 定义 3 个 int 类型的变量，并初始化
04      // 输出 3 个变量的值
05      Console.WriteLine("num1=" + num1 + " , num2=" + num2 + " , num3=" + num3);
06      Console.WriteLine();                                                    // 换行
07      Console.WriteLine("num1<num2 的结果：" + (num1 < num2));       // 小于操作
08      Console.WriteLine("num1>num2 的结果：" + (num1 > num2));       // 大于操作
09      Console.WriteLine("num1==num2 的结果：" + (num1 == num2));// 等于操作
10      Console.WriteLine("num1!=num2 的结果：" + (num1 != num2));// 不等于操作
11      Console.WriteLine("num1<=num2 的结果：" + (num1 <= num2));// 小于或等于操作
12      Console.WriteLine("num2>=num3 的结果：" + (num2 >= num3));// 大于或等于操作
13      Console.ReadLine();
14  }
```

学习笔记

第 5 行代码使用 Console.WriteLine(); 输出了一个空行，起到换行的作用。

第 7 ~ 12 行代码主要演示 6 种关系运算符的使用方法。

使用关系运算符比较大小关系代码的运行结果如图 3.18 所示。

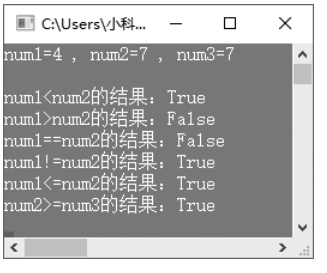


图 3.18　使用关系运算符比较大小关系代码的运行结果

3.5.5　逻辑运算符

假定某面包店在每周二的下午 7 点至 8 点和每周六的下午 5 点至 6 点对生日蛋糕商品进行折扣让利活动，那么想参加折扣活动的顾客就要在时间上满足这样的条件：周二并且 7:00 PM—8:00PM 或周六并且 5:00 PM—6:00PM，这里就用到了逻辑关系，C# 中也提供了这样的逻辑运算符来进行逻辑运算。

逻辑运算符是对真和假这两种布尔值进行运算的，运算后的结果仍是一个布尔值。C# 中的逻辑运算符主要包括 & / &&（逻辑与）、| / ||（逻辑或）、!（逻辑非）。在逻辑运算符中，除“!”是单目运算符外，其他都是双目运算符。表 3.8 列出了逻辑运算符的用法和说明。

表 3.8　逻辑运算符的用法和说明

运　算　符	含　义	用　法	结 合 方 向
&&、&	逻辑与	op1&&op2	左到右
||、|	逻辑或	op1||op2	左到右
!	逻辑非	!op	右到左

在使用逻辑运算符进行逻辑运算时，其运算结果如表 3.9 所示。

表 3.9　使用逻辑运算符进行逻辑运算时的运算结果

表达式 1	表达式 2	表达式 1&& 表达式 2	表达式 1|| 表达式 2	! 表达式 1
true	true	true	true	false
true	false	false	true	false
false	false	false	false	true
false	true	false	true	true

学习笔记

逻辑运算符“&&”与“&”都表示“逻辑与”，那么它们之间的区别是什么呢？从表 3.9 中可以看出，当两个表达式都为 true 时，逻辑与的结果才会为 true。使用“&”会判断两个表达式；而“&&”则是针对 bool 类型的数据进行判断的，当第一个表达式为 false 时，则不去判断第二个表达式，直接输出结果，从而节省计算机判断的次数。通常将这种在逻辑表达式中根据左端的表达式可推断出整个表达式的值称为“短路”，而那些始终执行逻辑运算符两边的表达式称为“非短路”。“&&”属于“短路”运算符，而“&”则属于“非短路”运算符。“||”与“|”的区别跟“&&”与“&”的区别类似。

示例 7．参加面包店的打折活动

创建一个控制台应用程序，使用代码实现本小节开始描述的场景，代码如下。

```
01  static void Main(string[] args)
02  {
03      Console.WriteLine("面包店正在打折，活动进行中……\n");     // 输出提示信息
04      Console.Write("请输入星期：");                          // 输出提示信息
05      string strWeek = Console.ReadLine();                    // 记录用户输入的星期
06      Console.Write("请输入时间：");                          // 输出提示信息
07      int intTime = Convert.ToInt32(Console.ReadLine());      // 记录用户输入的时间
08      // 判断是否满足活动参与条件（使用了 if 条件语句）
09        if((strWeek == "星期二" && (intTime >= 19 && intTime <= 20)) ||
(strWeek == "星期六" && (intTime >= 17 && intTime <= 18)))
```

```
10      {
11    Console.WriteLine("恭喜您，你获得了折扣活动参与资格，请尽情选购吧！ ");// 输出提示信息
12      }
13      else
14      {
15          Console.WriteLine("对不起，您来晚了一步，期待下次活动……"); // 输出提示信息
16      }
17      Console.ReadLine();
18  }
```

学习笔记

第 9 行和第 13 行代码使用了 if…else 条件判断语句，该语句主要用来判断是否满足某种条件。该语句将在第 4 章进行详细讲解，这里只需要了解即可。

第 9 行代码中在对条件进行判断时，使用了逻辑运算符 &&、|| 和关系运算符 ==、>=、<=。

运行参加面包店的打折活动代码，符合条件的运行结果如图 3.19 所示，不符合条件的运行结果如图 3.20 所示。

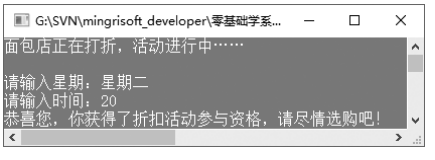


图 3.19　符合条件的运行结果

图 3.20　不符合条件的运行结果

3.5.6　位运算符

位运算符的操作数类型是整型，可以是有符号的，也可以是无符号的。C# 中的位运算符有位与、位或、位异或、取反等运算符。其中，位与、位或、位异或为双目运算符，取反为单目运算符。位运算是完全针对位方面的操作，因此，它在实际使用时，需要先将要执行运算的数据转换为二进制，然后才能执行运算。

学习笔记

整型数据在内存中以二进制的形式表示，如整型变量 7 的 32 位二进制表示是 00000000 00000000 00000000 00000111，其中，左边最高位是符号位，最高位是 0 表示正数；若为 1，则表示负数。负数采用补码表示，如 -8 的 32 位二进制表示为 11111111 11111111 11111111 11111000。

1. 位与运算

位与运算的运算符为“&”。位与运算的运算法则是，如果两个整型数据 a、b 对应位都是 1，则结果位才是 1；否则为 0。如果两个操作数的精度不同，则结果的精度与精度高的操作数相同，如图 3.21 所示。

2. 位或运算

位或运算的运算符为“|”。位或运算的运算法则是，如果两个操作数对应位都是 0，则结果位才是 0；否则为 1。如果两个操作数的精度不同，则结果的精度与精度高的操作数相同，如图 3.22 所示。

```
  0000 0000 0000 1100
& 0000 0000 0000 1000
  0000 0000 0000 1000
```

图 3.21　12&8 的运算过程

```
  0000 0000 0000 0100
| 0000 0000 0000 1000
  0000 0000 0000 1100
```

图 3.22　4|8 的运算过程

3. 位异或运算

位异或运算的运算符是“^”。位异或运算的运算法则是，当两个操作数的二进制表示相同（同时为 0 或同时为 1）时，结果为 0；否则为 1。若两个操作数的精度不同，则结果的精度与精度高的操作数相同，如图 3.23 所示。

4. 取反运算

取反运算也称为按位非运算，运算符为“～”。取反运算就是将操作数对应二进制中的 1 修改为 0，将 0 修改为 1，如图 3.24 所示。

```
  0000 0000 0001 1111
^ 0000 0000 0001 0110
  0000 0000 0000 1001
```

图 3.23　31^22 的运算过程

```
~ 0000 0000 0111 1011
  1111 1111 1000 0100
```

图 3.24　~ 123 的运算过程

在 C# 中使用 Console.WriteLine 输出图 3.21 ～图 3.24 的运算结果，主要代码如下。

```
01  Console.WriteLine("12 与 8 的结果为：" + (12 & 8));      // 位与计算整数的结果
02  Console.WriteLine("4 或 8 的结果为：" + (4 | 8));        // 位或计算整数的结果
03  Console.WriteLine("31 异或 22 的结果为：" + (31 ^ 22)); // 位异或计算整数的结果
04  Console.WriteLine("123 取反的结果为：" + ～123);         // 取反计算整数的结果
```

图 3.21 ～图 3.24 的运算结果如图 3.25 所示。

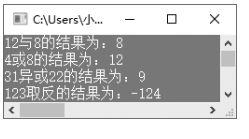


图 3.25　图 3.21 ~ 图 3.24 的运算结果

3.5.7　移位运算符

C# 中的移位运算符有两个，分别是左移位运算符 << 和右移位运算符 >>，这两个运算符都是双目运算符，它们主要用来对整数类型数据进行移位操作。移位运算符的右操作数不可以是负数，并且要小于左操作数的位数。下面分别对左移位运算符 << 和右移位运算符 >> 进行讲解。

1. 左移位运算符 <<

左移位运算符 << 是将一个二进制操作数向左移动指定的位数，左边（高位端）溢出的位被丢弃，右边（低位端）的空位用 0 补充。左移位运算相当于乘以 2 的 n 次幂。

例如，int 类型数据 48 对应的二进制数为 00110000，将其左移 1 位，根据左移位运算符的运算规则可以得出 (00110000<<1)=01100000，所以转换为十进制数就是 96（48×2）；将其左移 2 位，根据左移位运算符的运算规则可以得出 (00110000<<2)=11000000，所以转换为十进制数就是 192（48×2^2），其执行过程如图 3.26 所示。

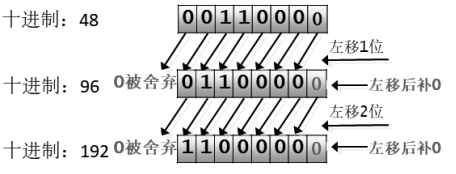


图 3.26　左移位运算执行过程

2. 右移位运算符 >>

右移位运算符 >> 是将一个二进制操作数向右移动指定的位数，右边（低位端）溢出的位被丢弃，而在填充左边（高位端）的空位时，如果最高位是 0（正数），则左侧空位填入 0；如果最高位是 1（负数），则左侧空位填入 1。右移位运算相当于除以 2 的 n 次幂。

正数 48 右移 1 位的运算过程如图 3.27 所示。

负数 −80 右移 2 位的运算过程如图 3.28 所示。

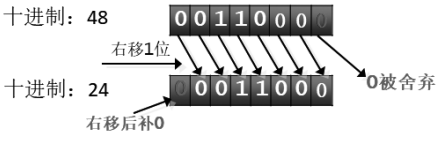


图 3.27　正数 48 右移 1 位的运算过程

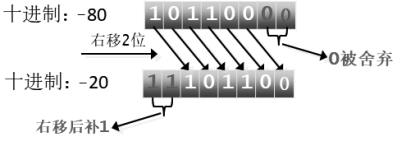


图 3.28　负数 −80 右移 2 位的运算过程

学习笔记

由于移位运算的速度很快，在程序中遇到表达式乘以（或除以）2 的 n 次幂的情况时，一般采用移位运算来代替。

3.5.8 条件运算符

条件运算符用“?:”表示，它是 C# 中唯一的三目运算符，该运算符需要 3 个操作数，形式如下。

< 表达式 1> ? < 表达式 2> : < 表达式 3>

在条件运算符的表达形式中，表达式 1 是一个布尔值，可以为真或假。如果表达式 1 为真，则返回表达式 2 的运算结果；如果表达式 1 为假，则返回表达式 3 的运算结果。例如：

```
01 int  x=5, y=6, max;
02 max=x<y? y : x ;
```

学习笔记

条件运算符相当于一个 if 语句，因此，上面的第 2 行代码可以修改如下。

```
01 if (x<y)
02     max=y;
03 else
04     max=x;
```

关于 if 语句的详细讲解，请参看本书第 4 章。

另外，条件运算符的结合性是从右向左的，即从右向左运算。例如：

```
01 int  x =5 , y = 6 ;
02 int  a = 1 ,b = 2 ;
03 int  z=0;
04 z= x>y ? x : a>b? a : b ;              // z 的值是 2
```

等价于：

```
01 int  x =5 , y = 6 ;
02 int  a = 1 ,b = 2 ;
03 int  z=0;
04 z= x>y ? x : (a>b? a : b) ;            // z 的值是 2
```

示例 8. 使用条件运算符判断人的年龄所处阶段

创建一个控制台应用程序，使用条件运算符判断输入年龄所处的阶段，并输出相应的

提示信息，代码如下。

```
01  static void Main(string[] args)
02  {
03      Console.Write(" 请输入一个年龄：");                    // 输入提示字符串
04      int age = Int32.Parse(Console.ReadLine());      // 将输入的年龄转换成 int 类型
05      // 利用条件运算符判断年龄是否大于 40 岁，并输出相应的内容
06      string info = age > 40 ? " 人到中年了！ " : " 这正是黄金奋斗的年龄 ";
07      Console.WriteLine(info);
08      Console.ReadLine();
09  }
```

学习笔记

在第 4 行代码中，Int32.Parse 方法用来将用户输入的年龄转换为 int 类型，存储到 int 类型变量中。

第 6 行代码定义了一个 string 类型的变量，记录条件表达式的返回结果。

使用条件运算符判断人的年龄所处阶段代码的运行结果如图 3.29 所示。

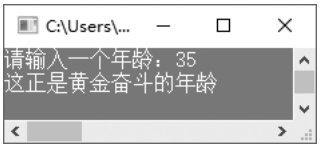


图 3.29　使用条件运算符判断人的年龄所处阶段代码的运行结果

3.6　数据类型转换

类型转换是将一个值从一种数据类型更改为另一种数据类型的过程。例如，可以将 string 类型的数据“457”转换为 int 类型，而且可以将任意类型的数据转换为 string 类型。

数据类型转换有两种方式，即隐式类型转换与显式类型转换。如果从低精度数据类型向高精度数据类型转换，则永远不会溢出，并且总是成功的；而从高精度数据类型向低精度数据类型转换，则必然会有信息丢失，甚至有可能失败。这种转换规则就像图 3.30 所示的两个场景，高精度数据类型相当于一个大水杯，低精度数据类型相当于一个小水杯，大水杯可以轻松装下小水杯中所有的水，但小水杯却无法装下大水杯中所有的水，装不下的部分必然会溢出。

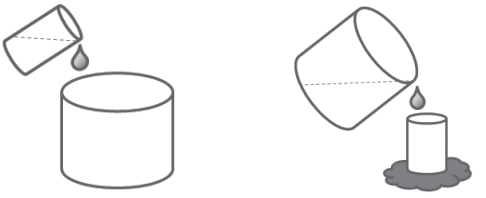


图 3.30　用大、小水杯类比数据类型转换的示意图

3.6.1　隐式类型转换

隐式类型转换是不需要声明就能进行的转换，在进行隐式类型转换时，编译器不需要进行检查就能自动进行转换。下列基本数据类型会涉及数据转换（不包括逻辑类型），这些类型按精度从“低”到“高”排列的顺序为 byte < short < int < long < float < double，可对照图 3.31，其中 char 类型比较特殊，它可以与部分 int 类型数字兼容且不会发生精度变化。

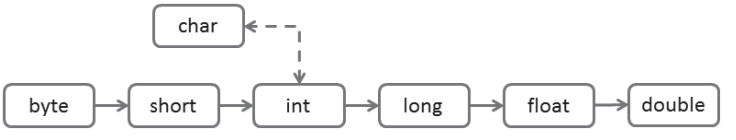


图 3.31　自动转换的兼容顺序图

例如，将 int 类型的值隐式转换成 long 类型，代码如下。

```
01  int i = 927;              // 声明一个整型变量 i 并初始化为 927
02  long j = i;               // 隐式转换成 long 类型
```

3.6.2　显式类型转换

有很多场合不能进行隐式类型转换，否则编译器会出现错误。例如，下面的类型在进行隐式类型转换时会出现错误。

- int 类型转换为 short 类型：会丢失数据。
- int 类型转换为 uint 类型：会丢失数据。
- float 类型转换为 int 类型：会丢失小数点后面的所有数据。
- double 类型转换为 int 类型：会丢失小数点后面的所有数据。
- 数值类型转换为 char 类型：会丢失数据。
- decimal 类型转换为其他数值类型：decimal 类型的内部结构不同于整数和浮点数。

如果遇到上面类型之间的转换，就需要用到 C# 中的显式类型转换。显式类型转换也称为强制类型转换，它需要在代码中明确地声明要转换的类型。如果要把高精度的变量转

换为低精度的变量，就需要使用显式类型转换。

显式类型转换的一般形式为：

（类型说明符）表达式

其功能是把表达式的运算结果强制转换为类型说明符所表示的类型。

例如，下面的代码用来把 x 转换为 float 类型。

```
(float) x;
```

通过显式类型转换，可以解决高精度数据向低精度数据转换的问题。例如，将 double 类型的值 4.5 赋给 int 类型的变量时，可以使用下面的代码。

```
01 int  i ;
02 i = (int)4.5;            // 使用显式类型转换
```

3.6.3　使用 Convert 类进行转换

在 3.6.2 节中讲解了使用“(类型说明符) 表达式”可以进行显式类型转换，下面使用这种方式实现如下类型转换。

```
01 long l=3000000000;
02 int i = (int)l;
```

按照代码的本意，i 的值应该是 3000000000，但在运行上面两行代码时，发现 i 的值是 -1294967296。这主要是由于 int 类型的最大值为 2147483647，而 3000000000 比 2147483647 大，所以在使用上面的代码进行显式类型转换时出现了与预期不符的结果，但是程序并没有报告错误。如果在实际开发中遇到这种情况，则可能会引起大的 Bug。那么，在遇到这种类型的错误时，有没有一种方式能够向开发人员报告错误呢？答案是“有”。C# 中提供了 Convert 类，该类也可以进行显式类型转换，它的主要作用是将一个基本数据类型转换为另一个基本数据类型。Convert 类的常用方法及说明如表 3.10 所示。

表 3.10　Convert 类的常用方法及说明

方　法	说　明
ToBoolean	将指定的值转换为等效的布尔值
ToByte	将指定的值转换为 8 位无符号整数
ToChar	将指定的值转换为 Unicode 字符
ToDateTime	将指定的值转换为 DateTime
ToDecimal	将指定的值转换为 Decimal 数字
ToDouble	将指定的值转换为双精度浮点数字
ToInt32	将指定的值转换为 32 位有符号整数

续表

方　法	说　明
ToInt64	将指定的值转换为 64 位有符号整数
ToSByte	将指定的值转换为 8 位有符号整数
ToSingle	将指定的值转换为单精度浮点数字
ToString	将指定的值转换为等效的 String 表示形式
ToUInt32	将指定的值转换为 32 位无符号整数
ToUInt64	将指定的值转换为 64 位无符号整数

例如，定义一个 double 类型的变量 x，并赋值为 198.99，使用 Convert 类将其显式转换为 int 类型，代码如下。

```
01 double x = 198.99;                    // 定义 double 类型的变量并初始化
02 int y = Convert.ToInt32(x);           // 使用 Convert 类的方法进行显式类型转换
```

下面使用 Convert 类的 ToInt32 方法对本小节开始的两行代码进行修改，修改后的代码如下。

```
01 long l=3000000000;
02 int i = Convert.ToInt32(l);
```

再次运行这两行代码，则会出现如图 3.32 所示的错误提示信息。

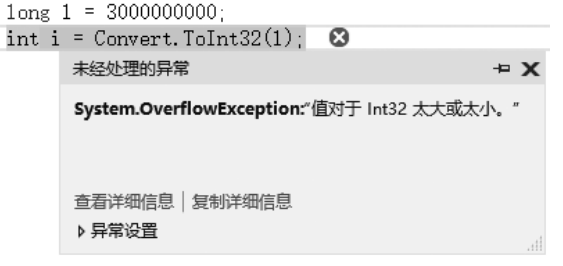


图 3.32　再次运行代码时出现的错误提示信息

这样，开发人员即可根据如图 3.32 所示的错误提示信息对程序代码进行修改，避免程序出现逻辑错误。

3.7　运算符优先级与结合性

C# 中的表达式是使用运算符连接起来的符合 C# 规范的式子，运算符的优先级决定了表达式中运算执行的先后顺序。运算符优先级其实相当于进、销、存的业务流程，如进货→入

库→销售→出库，只能按这个步骤进行操作。运算符的优先级也是这样的，它是按照一定的先后顺序进行计算的。C# 中的运算符优先级按照由高到低的顺序依次是自增运算符和自减运算符、算术运算符、移位运算符、关系运算符、逻辑运算符、条件运算符、赋值运算符。

如果两个运算符具有相同的优先级，则会根据其结合性确定是从左至右运算，还是从右至左运算。表 3.11 列出了运算符从高到低的优先级顺序及结合性。

表 3.11　运算符从高到低的优先级顺序及结合性

运算符类别	运　算　符	数　　目	结　合　性
单目运算符	++, --, !	单目	←
算术运算符	*, /, %	双目	→
	+, -	双目	→
移位运算符	<<，>>	双目	→
关系运算符	>, >=, <, <=	双目	→
	==, !=	双目	→
逻辑运算符	&&	双目	→
	||	双目	→
条件运算符	? :	三目	←
赋值运算符	=,+=,-=,*=,/=,%=	双目	←

学习笔记

表 3.11 中的“←”表示从右至左，“→”表示从左至右。从表 3.11 中可以看出，在 C# 的运算符中，只有单目运算符、条件运算符和赋值运算符的结合性为从右至左，其他运算符的结合性都为从左至右。所以，下面的代码是等效的。

```
01  !a++;              等效于:     !(a++);
02  a ? b : c ? d : e;  等效于:    a ? b : (c ? d : e);
03  a = b = c;          等效于:     a = (b = c);
04  a + b - c;          等效于:     (a + b) - c;
```

第 4 章　流程控制语句

做任何事情都要遵循一定的原则。例如，到图书馆去借书就必须要有借书证，并且借书证不能过期，这两个条件缺一不可。程序设计也是如此，需要利用流程控制实现与用户的交流，并根据用户的需求决定程序“做什么”“怎么做”。

流程控制对于任何一门编程语言来说都是至关重要的，它提供了控制程序如何执行的方法。如果没有流程控制语句，那么整个程序将按照线性顺序来执行，而不能根据用户的需求决定程序执行的顺序。本章将对 C# 中的流程控制语句进行详细讲解。

4.1　决策分支

计算机的主要功能是提供用户计算功能，但在计算的过程中会遇到各种各样的情况，针对不同的情况会有不同的处理方法，这就要求程序开发语言要有处理决策的能力。汇编语言使用判断指令和跳转指令实现决策，高级语言使用选择判断语句实现决策。

一个决策系统就是一个分支结构，这种分支结构就像一个树形结构，每到一个节点都需要做决定，就好比人走到十字路口，是向前走、向左走还是向右走都需要做决定，不同的分支代表不同的决定。例如，十字路口的分支结构如图 4.1 所示。

为描述决策系统的流通，设计人员开发了流程图。流程图使用图形方式描述系统在不同状态下的不同处理方法。开发人员使用流程图表现程序的结构，主要的流程图符号如图 4.2 所示。

使用流程图描述十字路口转向的决策，利用方位做决定，判断是否为南方，如果是南方，则向前行；如果不是南方，则寻找南方。十字路口转向流程图如图 4.3 所示。

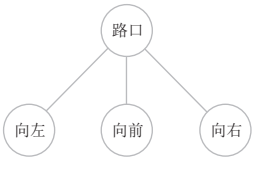


图 4.1　十字路口的分支结构

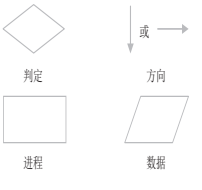


图 4.2　主要的流程图符号

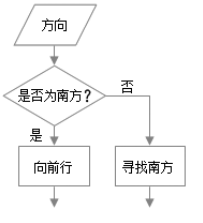


图 4.3　十字路口转向流程图

在程序中使用选择结构语句来做决策，选择结构语句是编程语言的基础语句。在 C# 中有两种选择结构语句，分别是 if 语句和 switch 语句，下面分别对这两种选择结构语句进行讲解。

学习笔记

选择结构语句也称为条件判断语句，或者分支语句。

4.2　if 语句

在生活中，每个人都要进行各种各样的选择。例如，吃什么菜？走哪条路？找什么人？那么当程序遇到选择时，该怎么办呢？这时需要使用选择结构语句。if 语句是最基础的一种选择结构语句，它主要有 3 种形式，分别为 if 语句、if…else 语句和 if…else if…else 多分支语句，本节将分别对它们进行详细讲解。

4.2.1　最简单的 if 语句

C# 中使用 if 关键字来组成选择语句，其最简单的语法形式如下。

```
if(表达式)
{
        语句块
}
```

学习笔记

在使用 if 语句时，如果只有一条语句，那么省略 {} 是没有语法错误的，而且不影响程序的执行，但是为了程序代码的可读性，建议不要省略。

其中，表达式部分必须用 () 括起来，它可以是一个单纯的布尔类型的变量（或常量），也可以是关系表达式（或逻辑表达式）。如果表达式的值为真，则执行“语句块”，之后继续执行“下一条语句”；如果表达式的值为假，则跳过“语句块”，执行“下一条语句”。这种形式的 if 语句相当于汉语中的“如果……那么……”，其流程图如图 4.4 所示。

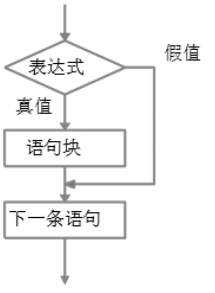


图 4.4　if 语句流程图

示例 1．判断输入的数字是不是奇数

使用 if 语句判断用户输入的数字是不是奇数，代码如下。

```
01  static void Main(string[] args)
02  {
03      Console.WriteLine(" 请输入一个数字：");
04      int iInput = Convert.ToInt32(Console.ReadLine());// 记录用户的输入
05      if (iInput % 2 != 0)                                    // 使用 if 语句进行判断
06      {
07          Console.WriteLine(iInput + " 是一个奇数！ ");
08      }
09      Console.ReadLine();
10  }
```

学习笔记

第 4 行代码使用 Convert.ToInt32 方法将用户输入的数字强制转换成了 int 类型，然后使用 int 类型的变量进行记录。

奇数的条件是不能被 2 整除，因此，第 5 行代码判断用户输入的数字求余 2 是否不等于 0，以此来判断用户输入的数字是不是奇数。

运行上面代码，当输入 5 时，结果如图 4.5 所示；当输入 6 时，结果如图 4.6 所示。

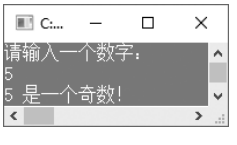


图 4.5　奇数运行结果

图 4.6　不是奇数的运行结果

学习笔记

if 语句后面如果只有一条语句，那么可以不使用大括号（{}），但是，不建议开发人员使用这种形式，不管 if 语句后面有多少要执行的语句，都建议使用大括号括起来，这样方便代码的阅读。例如，下面的代码未使用大括号。

```
01  if (a > b)
02      max = a;
```

学习笔记

- if 语句后面多加了分号。例如，if 语句正确表示如下。

```
01  if (i == 5)
02      Console.WriteLine("i 的值是 5");
```

上面两行代码的本意是，当变量 i 的值为 5 时，执行下面的输出语句。但是，如果在 if 语句后面多加了分号，下面的输出语句将会无条件执行，if 语句就起不到判断的作用，代码如下。

```
01  if (i == 5);
02      Console.WriteLine("i 的值是 5");
```

- 在使用 if 语句时，如果要将多个语句作为复合语句来执行，例如，程序的真正意图是如下语句。

```
01  if(flag)
02  {
03      i++;
04      j++;
05  }
```

那么，如果删去大括号，则代码如下。

```
01  if(flag)
02      i++;
03      j++;
```

在执行程序时，无论 flag 是否为 true，j++ 都会无条件执行，这显然与程序的本意是不符的，但程序并不会报告异常，因此这种错误很难发现。

4.2.2　if…else 语句

如果遇到只能二选一的情况，比如某个公司在发展过程中遇到了“扩张”和“求稳”的抉择，示意图如图 4.7 所示。

图 4.7　公司在发展过程中遇到了“扩张”和“求稳”的抉择示意图

C# 中提供了 if…else 语句解决类似问题，其语法如下。

```
if( 表达式 )
{
        语句块 1;
}
else
{
        语句块 2;
}
```

在使用 if…else 语句时，表达式可以是一个单纯的布尔类型的变量（或常量），也可以是关系表达式（或逻辑表达式）。如果满足条件，则执行 if 后面的语句块；否则，执行 else 后面的语句块。这种形式的选择语句相当于汉语中的“如果……否则……”，其流程图如图 4.8 所示。

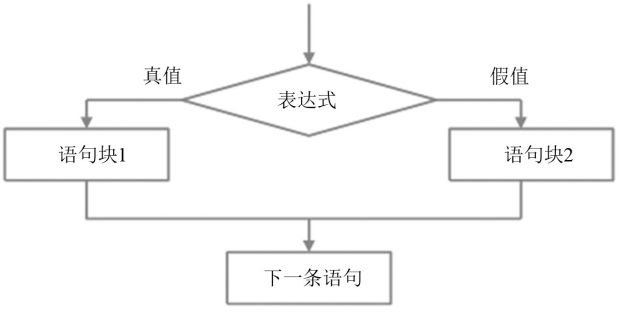


图 4.8　if…else 语句流程图

学习笔记

if…else 语句可以使用条件运算符进行简化，如下面这段代码：

```
01  if(a > 0)
02      b = a;
03  else
04      b = -a;
```

可以简写成：

```
b = a > 0?a:-a;
```

上段代码主要实现求绝对值的功能，如果 a > 0，就把 a 的值赋给变量 b；否则，将 -a 赋给变量 b。使用条件运算符的好处是可以使代码简洁，并且有一个返回值。

示例 2．根据分数划分优秀等级

使用 if…else 语句判断用户输入的分数是不是足够优秀，如果大于 90，则表示优秀；否则，输出“希望你继续努力！”，代码如下。

```
01  static void Main(string[] args)
02  {
03      Console.WriteLine(" 请输入你的分数：");
04      int score = Convert.ToInt32(Console.ReadLine());   // 记录用户的输入
05      if (score > 90)                                    // 判断输入是否大于 90
06      {
07          Console.WriteLine(" 你非常优秀！ ");
08      }
09      else                                               // 不大于 90 的情况
10      {
11          Console.WriteLine(" 希望你继续努力！ ");
12      }
13      Console.ReadLine();
14  }
```

运行上面代码，当输入一个大于 90 的数时，如 93，效果如图 4.9 所示；当输入一个小于 90 的数时，如 87，效果如图 4.10 所示。

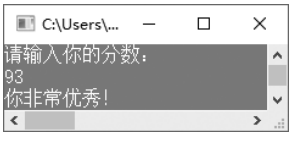


图 4.9　当输入 93 时的运行结果

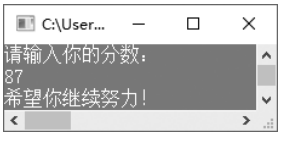


图 4.10　当输入 87 时的运行结果

学习笔记

在使用 if…else 语句时，else 一定不可以单独使用，它必须和关键字 if 一起使用。例如，下面的代码是错误的。

```
01  else
02  {
03      max=a;
04  }
```

在程序中使用 if…else 语句时，如果出现 if 语句多于 else 语句的情况，那么将会出现悬垂 else 问题：究竟 else 语句和哪个 if 语句相匹配呢？例如，下面的代码。

```
01 if(x>1)
02     if(y>x)
03         y++;
04 else
05     x++;
```

如果遇到上面的情况，在没有特殊处理时，那么 else 语句永远都与最后出现的 if 语句相匹配，即上面代码中的 else 语句是与 if(y>x) 语句相匹配的。如果要改变 else 语句的匹配对象，则可以使用大括号。例如，将上面代码修改如下。

```
01 if(x>1)
02 {
03     if(y>x)
04         y++;
05 }
06 else
07     x++;
```

这样，else 语句将与 if(x>1) 语句相匹配。

学习笔记

建议在 if 后面使用大括号将要执行的语句括起来，这样可以避免程序代码混乱。

4.2.3 if…else if…else 语句

大家平时在网上购物需要付款时通常都有多种选择，如图 4.11 所示。

在图 4.11 中，提供了多种付款方式，这时用户就需要从多个选项中选择一个。在开发程序时，如果遇到多选一的情况，则可以使用 if…else if…else 语句。该语句是一个多分支选择语句，通常表现为“如果满足某种条件，则进行某种处理；否则，如果满足另一种条件，则执行另一种处理……”。if…else if…else 语句的语法格式如下。

```
if( 表达式 1)
{
    语句 1;
}
else if( 表达式 2)
{
    语句 2;
}
```

```
else if(表达式3)
{
      语句3
}
   …
else if(表达式m)
{
      语句m
}
else
{
      语句n
}
```

图 4.11　网上购物时的付款页面

使用 if…else if…else 语句时，表达式部分必须用 () 括起来，它可以是一个单纯的布尔类型的变量（或常量），也可以是关系表达式（或逻辑表达式）。如果表达式为真，则执行语句；而如果表达式为假，则跳过该语句，进行下一个 else if 的判断；只有在所有表达式都为假的情况下，才会执行 else 中的语句。if…else if…else 语句的流程图如图 4.12 所示。

学习笔记

if 和 else if 都需要判断表达式的真假，而 else 则不需要判断；另外，else if 和 else 都必须跟 if 一起使用，不能单独使用。

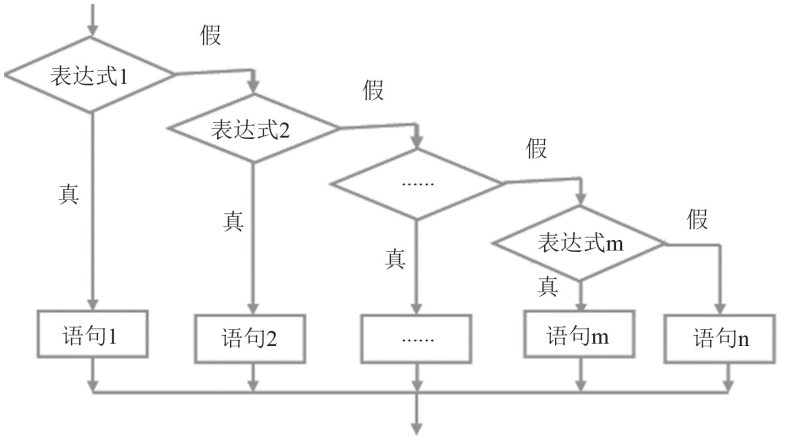


图 4.12　if…else if…else 语句的流程图

示例 3．根据用户输入的年龄输出相应的信息提示

使用 if…else if…else 多分支语句实现根据用户输入的年龄输出相应的信息提示的功能，代码如下。

```
01  static void Main(string[] args)
02  {
03      int YouAge = 0;                    // 声明一个 int 类型的变量 YouAge，值为 0
04      Console.WriteLine(" 请输入您的年龄：");
05      YouAge = int.Parse(Console.ReadLine()); // 获取用户输入的数据
06      if (YouAge <= 18)                  // 调用 if 语句判断输入的数据是否小于或等于 18
07      {
08          // 如果输入的数据小于或等于 18，则输出提示信息
09          Console.WriteLine(" 您的年龄还小，要努力奋斗哦！ ");
10      }
11      else if (YouAge > 18 && YouAge <= 30) // 判断输入的数据是否大于 18 并且小于或等于 30
12      {
13          // 如果输入的数据大于 18 并且小于或等于 30，则输出提示信息
14          Console.WriteLine(" 您现在的阶段正是努力奋斗的黄金阶段！ ");
15      }
16      else if (YouAge > 30 && YouAge <= 50)// 判断输入的数据是否大于 30 并且小于或等于 50
17      {
18          // 如果输入的数据大于 30 并且小于或等于 50，则输出提示信息
19          Console.WriteLine(" 您现在的阶段正是人生的黄金阶段！ ");
20      }
21      else
22      {
23          Console.WriteLine(" 最美不过夕阳红！ ");
24      }
25      Console.ReadLine();
26  }
```

学习笔记

第 5 行代码中的 int.Parse() 方法用来将用户输入的数据强制转换成 int 类型。

运行上面代码，输入一个年龄值，如 35，按回车键即可输出相应的信息提示，结果如图 4.13 所示。

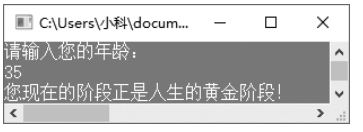


图 4.13　当输入 35 时的运行结果

学习笔记

使用 if 语句时，应尽量遵循以下原则。

- 使用 bool 变量作为判断条件，假设 bool 变量为 falg，较为规范的书写格式如下。

```
if(flag)        // 表示为真
if(!flag)       // 表示为假
```

不符合规范的书写格式如下。

```
if(flag==true)
if(flag==false)
```

- 当使用浮点类型的变量与 0 值进行比较时，规范的书写格式如下。

```
// 这里的 0.00001 是 d_value 的精度，d_value 为 double 类型
if(d_value>=-0.00001&&d_value<=0.00001)
```

不符合规范的书写格式如下。

```
if(d_value==0.0)
```

- 使用 if(1==a) 这样的书写格式可以防止错写成 if(a=1) 这种形式，从而可以避免逻辑上的错误。

4.2.4　if 语句的嵌套

前面讲了 3 种形式的 if 语句，这 3 种形式的语句之间都可以进行互相嵌套。例如，在最简单的 if 语句中嵌套 if…else 语句，形式如下。

```
if( 表达式 1)
{
    if( 表达式 2)
            语句 1;
```

```
    else
            语句 2;
}
```

又如，在 if…else 语句中嵌套 if…else 语句，形式如下。

```
if( 表达式 1)
{
        if( 表达式 2)
            语句 1;
        else
            语句 2;
}
else
{
        if( 表达式 2)
            语句 1;
        else
            语句 2;
}
```

学习笔记

if 语句可以有多种嵌套方式，在开发程序时，可以根据自身需要选择合适的嵌套方式，但一定要注意逻辑关系的正确处理。

示例 4. 判断输入的年份是不是闰年

通过使用嵌套的 if 语句实现判断用户输入的年份是不是闰年的功能，代码如下。

```
01  static void Main(string[] args)
02  {
03      Console.WriteLine(" 请输入一个年份：");
04      int iYear = Convert.ToInt32(Console.ReadLine()); // 记录用户输入的年份
05      if (iYear % 4 == 0)                                    // 四年一闰
06      {
07          if (iYear % 100 == 0)
08          {
09              if (iYear % 400 == 0)                          // 四百年再闰
10              {
11                  Console.WriteLine(" 这是闰年 ");
12              }
13              else                                           // 百年不闰
14              {
15                  Console.WriteLine(" 这不是闰年 ");
16              }
17          }
```

```
18          else
19          {
20              Console.WriteLine(" 这是闰年 ");
21          }
22      }
23      else
24      {
25          Console.WriteLine(" 这不是闰年 ");
26      }
27      Console.ReadLine();
28  }
```

学习笔记

判断闰年的方法是“四年一闰，百年不闰，四百年再闰”。程序使用嵌套的 if 语句对这 3 个条件进行逐一判断，第 5 行代码首先判断年份能否被 4 整除（iYear%4==0），如果不能被 4 整除，则输出字符串“这不是闰年”；如果能被 4 整除，则第 7 行代码继续判断能否被 100 整除（iYear%100==0）。如果不能被 100 整除，则输出字符串“这是闰年”；如果能被 100 整除，则第 9 行代码继续判断能否被 400 整除（iYear%400==0）。如果能被 400 整除，则输出字符串“这是闰年”；如果不能被 400 整除，则输出字符串“这不是闰年”。

运行上面代码，当输入一个闰年年份时，如 2000，结果如图 4.14 所示；当输入一个非闰年年份时，如 2017，结果如图 4.15 所示。

图 4.14　当输入年份为 2000 时的运行结果

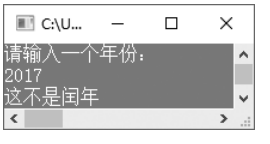


图 4.15　当输入年份为 2017 时的运行结果

学习笔记

在使用 if 语句嵌套时，注意 else 关键字要和 if 关键字成对出现，并且遵守临近原则，即 else 关键字总是和离自己最近的 if 语句相匹配。

在进行条件判断时，应该尽量使用复合语句，以免产生二义性，导致运行结果和预想的不一致。

4.3 switch 多分支语句

在开发中常见的一个问题是检测一个变量是否符合某种条件，如果不符合，则再用另一个值来检测它，依此类推。当然，这种问题可以使用 if 语句来完成。

例如，使用 if 语句检测变量是否符合某种条件。

```
01 char grade = 'B';
02 if (grade == 'A')
03 {
04     Console.WriteLine(" 真棒 ");
05 }
06 if (grade == 'B')
07 {
08     Console.WriteLine(" 做得不错 ");
09 }
10 if (grade == 'C')
11 {
12     Console.WriteLine(" 再接再厉 ");
13 }
```

在执行上述代码时，每一条 if 语句都会进行判断，这样显得非常烦琐。为了简化这种编写代码的方式，C# 提供了 switch 语句，将判断动作组织起来，以一种比较简单的方式实现“多选一”的逻辑。本节将对 switch 语句进行详细讲解。

4.3.1 switch 语句

switch 语句是多分支条件判断语句，它根据参数的值使程序从多个分支中选择一个用于执行的分支，其基本语法如下。

```
switch( 判断参数 )
{
case 常量值 1:
语句块 1
break;
case 常量值 2:
语句块 2
break;
……
case 常量值 n:
```

```
语句块 n
break;
defaul:
语句块 n+1
break;
}
```

switch 关键字后面的括号 () 中的内容是要判断的参数，参数必须是 sbyte、byte、short、ushort、int、uint、long、ulong、char、string、bool 或枚举类型中的一种。大括号 {} 中的代码是由多个 case 子句组成的，每个 case 关键字后面都有相应的语句块，这些语句块都是 switch 语句可能执行的语句块。如果符合常量值，则 case 后面的语句块就会被执行，语句块执行完毕后，执行 break 语句，使程序跳出 switch 语句；如果条件都不满足，则执行 default 中的语句块。

学习笔记

（1）case 后的各常量值不可以相同，否则会出现错误。

（2）case 后面的语句块可以有多条语句，不必使用大括号括起来。

（3）case 语句和 default 语句的顺序可以改变，这不会影响程序执行结果。

（4）一个 switch 语句中只能有一个 default 语句，而且 default 语句可以省略。

switch 语句的执行流程图如图 4.16 所示。

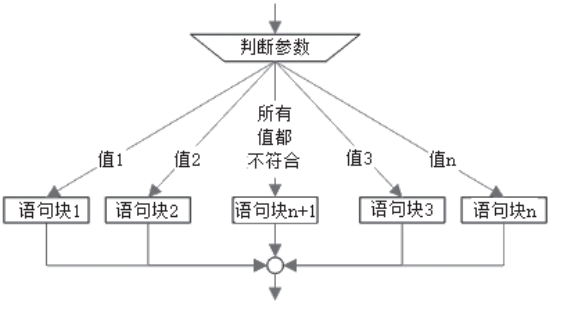


图 4.16　switch 语句的执行流程图

示例 5．查询高考录取分数线

使用 switch 多分支语句实现查询高考录取分数线的功能。其中，民办本科为 350 分、艺术类本科为 290 分、体育类本科为 280 分、二本为 445 分、一本为 555 分。代码如下。

```
01  static void Main(string[] args)
02  {
03        //输出提示问题
04        Console.WriteLine("请输入要查询的录取分数线（比如民办本科、艺术类本科、体育类
```

```
本科、二本、一本）");
05      string strNum = Console.ReadLine();      // 获取用户输入的数据
06      switch (strNum)
07      {
08          case "民办本科":                      // 查询民办本科录取分数线
09              Console.WriteLine("民办本科录取分数线：350");
10              break;
11          case "艺术类本科":                    // 查询艺术类本科录取分数线
12              Console.WriteLine("艺术类本科录取分数线：290");
13              break;
14          case "体育类本科":                    // 查询体育类本科录取分数线
15              Console.WriteLine("体育类本科录取分数线：280");
16              break;
17          case "二本":                          // 查询二本录取分数线
18              Console.WriteLine("二本录取分数线：445");
19              break;
20          case "一本":                          // 查询一本录取分数线
21              Console.WriteLine("一本录取分数线：555");
22              break;
23          default:                              // 如果不是以上输入，则输入错误
24              Console.WriteLine("您输入的查询信息有误！");
25              break;
26      }
27      Console.ReadLine();
28  }
```

查询高考录取分数线代码的运行结果如图 4.17 所示。

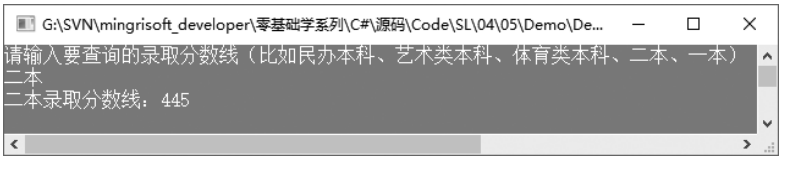


图 4.17　查询高考录取分数线代码的运行结果

学习笔记

在使用 switch 语句时，常量表达式的值绝不可以是浮点类型的。例如，下面的代码就是不合法的。

```
01  double dNum = Convert.ToDouble(Console.ReadLine());
02  switch (dNum)
03  {
04      case 1.0:
05          Console.WriteLine("分支一");
```

```
06          break;
07      case 2.0:
08          Console.WriteLine(" 分支二 ");
09          break;
10  }
```

在 Visual Studio 2017 中运行上述代码时，将会出现如图 4.18 所示的错误提示信息。

图 4.18　判断参数为浮点类型时出现的错误提示信息

（2）在使用 switch 语句时，每个 case 语句或 default 语句后面必须有一个 break 关键字，否则将会出现如图 4.19 所示的错误提示信息。

图 4.19　缺少 break 关键字时出现的错误提示信息

4.3.2　switch 语句与 if…else if…else 语句的区别

4.2.3 节中讲到的 if…else if…else 语句也可以处理多分支选择的情况，但它主要是对布尔表达式、关系表达式或逻辑表达式进行判断的，而 switch 多分支语句主要对常量值进行判断。因此，在程序开发中，如果遇到多分支选择的情况，并且判断的条件不是关系表达式、逻辑表达式或浮点类型，就可以使用 switch 语句代替 if…else if…else 语句，这样执行效率会更高。

4.4　while 和 do…while 循环

学习和使用 C# 的目的是使用它编写出能够解决现实生活问题的程序。生活中存在很多重复性的工作，有时甚至不知道这种工作需要重复的次数，那么如何用简单的 C# 语句解决这种复杂的、带有重复性的问题呢？ C# 提供了循环控制语句来解决这类问题。C# 中的循环控制语句主要有 while、do…while 和 for，本节将首先对 while 和 do…while 循环的使用进行讲解。

4.4.1 while 循环

while 循环用来实现“当型”循环结构，它的语法格式如下。

```
while(表达式)
{
语句
}
```

表达式一般是一个关系表达式或一个逻辑表达式，表达式的值应该是一个逻辑值真或假（true 和 false）。当表达式的值为真时，开始循环执行语句；而当表达式的值为假时，退出循环，执行循环外的下一条语句。循环每次都是执行完语句后回到表达式处重新开始判断，重新计算表达式的值。

while 循环流程图如图 4.20 所示。

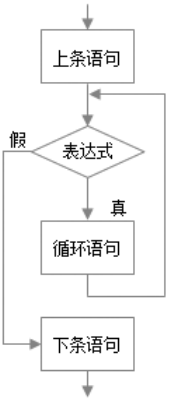


图 4.20　while 循环流程图

示例 6．数学家高斯的故事

许多年前，在德国的一所乡村小学里，有一个很懒的老师，他总是要求学生们不停地做整数加法计算，因为在学生们将一长串整数求和的过程中，他可以在旁边名正言顺地偷懒。有一天，他又用同样的方法布置了一道从 1 加到 100 的求和题目。正当他打算偷懒时，有一个学生说自己算出了答案。老师自然是不信的，不看答案就让学生再去算，可是学生还是站在老师面前不动。老师被激怒了，认为这个学生是在挑衅自己的威严，他不相信一个小学生能在几秒钟内就将从 1 加到 100 的求和题目计算出结果。于是抢过学生的答案，正打算教训学生时，突然发现学生写的答案是 5050。老师愣住了，原来这个学生不是用一个数一个数地加起来的方式计算的，而是将 100 个数分成 1+100=101、2+99=101（一直到 50+51=101）等 50 对，然后使用 101×50=5050 计算得出的。这个聪明的学生就是德国

著名的数学家高斯。本示例将使用 while 循环编写程序，通过程序实现从 1 到 100 的累加，代码如下。

```
01   static void Main(string[] args)
02   {
03       int iNum = 1;              //iNum 从 1 到 100 递增
04       int iSum = 0;              // 记录每次累加后的结果
05       while (iNum <= 100)        //iNum <= 100 是循环条件
06       {
07           iSum += iNum;          // 把每次的 iNum 的值累加到上次累加的结果中
08           iNum++;                // 每次循环 iNum 的值加 1
09       }
10       // 输出结果
11       Console.WriteLine("1 到 100 的累加结果是：" + iSum);
12       Console.ReadLine();
13   }
```

学习笔记

题目要求计算 1 到 100 之间数字的累加结果，那么需要先定义一个变量 iNum 作为循环条件的判定，iNum 的初始值是 1，循环条件是 iNum 必须小于或等于 100。也就是只有在 iNum 小于或等于 100 时才进行累加操作，若 iNum 大于 100，则循环终止。

每次循环只能计算其中一次相加的结果，想要计算 100 个数字的累加值，需要定义一个变量 iSum 来暂存每次累加的结果，并作为下一次累加操作的基数。

iNum 的初始值是 1，要计算 1 到 100 之间数字的累加结果，需要 iNum 每次进入循环，进行累加后，iNum 的值增加 1，为下一次进入循环进行累加做准备，也同时作为循环结束的判断条件。

当 iNum 大于 100 时，循环结束，执行后面的输出语句。

上面代码的运行结果如下。

```
1 到 100 的累加结果是：5050
```

学习笔记

如果将示例 6 代码中 while 语句后面的大括号去掉，即将代码修改如下：

```
01  static void Main(string[] args)
02   {
03       int iNum = 1;                  //iNum 从 1 到 100 递增
04       int iSum = 0;                  // 记录每次累加后的结果
05       while (iNum <= 100)            //iNum <= 100 是循环条件
```

```
            iSum += iNum;                    // 把每次的 iNum 的值累加到上次累加的结果中
            iNum++;                          // 每次循环 iNum 的值加 1
        Console.WriteLine("1 到 100 的累加结果是: " + iSum);    // 输出结果
        Console.ReadLine();
    }
```

那么重新编译并运行代码不会出现任何结果。分析造成这种情况的原因是：当 while 语句循环体中的语句大于一条时，需要把循环体放在大括号中，如果 while 语句后面没有大括号，则 while 循环只会循环 while 语句后的第一条语句。对于上面的代码，则没有对循环变量 iNum 增加的过程，于是每次进入循环时，iNum 的值都是 1，形成死循环，永远不会执行后面的其他语句。

学习笔记

- 循环体如果是多条语句，则需要用大括号括起来，如果不用大括号，则循环体只包含 while 语句后的第一条语句。
- 循环体内或表达式中必须有使循环结束的条件。例如，上述程序中的循环条件是 iNum <= 100，iNum 的初始值为 1，循环体中就用 iNum++ 来使得 iNum 趋向于 100，以使循环结束。

4.4.2 do…while 循环

在有些情况下无论循环条件是否成立，循环体的内容都要被执行一次，这时可以使用 do…while 循环。do…while 循环的特点是先执行循环体，再判断循环条件，其语法格式如下。

```
do
{
语句
}
while(表达式);
```

do 为关键字，必须与 while 配对使用。do 与 while 之间的语句称为循环体，该语句是用大括号括起来的复合语句。do…while 语句中的表达式与 while 语句中的相同，也为关系表达式或逻辑表达式，但特别值得注意的是，do…while 语句后一定要有分号“;”。do…while 循环流程图如图 4.21 所示。

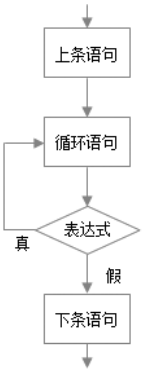


图 4.21　do…while 循环流程图

从图 4.21 中可以看出，当程序运行到 do…while 时，先执行一次循环体的内容，然后判断循环条件。当循环条件为真时，重新返回执行循环体的内容，如此反复，直到循环条件为假，循环结束，程序执行 do…while 循环后面的语句。

示例 7．使用 do…while 循环挑战数学家高斯

使用 do…while 循环编写程序，实现从 1 到 100 的累加，代码如下。

```
01  static void Main(string[] args)
02  {
03      int iNum = 1;                       //iNum 从 1 到 100 递增
04      int iSum = 0;                       // 记录每次累加后的结果
05      do
06      {
07          iSum += iNum;                   // 把每次的 iNum 的值累加到上次累加的结果中
08          iNum++;                         // 每次循环 iNum 的值加 1
09      } while (iNum <= 100);              //iNum <= 100 是循环条件
10      Console.WriteLine("1 到 100 的累加结果是: " + iSum);// 输出结果
11      Console.ReadLine();
12  }
```

学习笔记

在上面代码中将判断条件 iNum <= 100 放到了循环体后面，这样，无论 iNum 是否满足条件，都将至少执行一次循环体。

学习笔记

本示例代码的运行结果与示例 6 代码的运行结果一样。

4.4.3 while 语句和 do…while 语句的区别

while 语句和 do…while 语句都用来控制代码的循环，但 while 语句适用于先进行条件判断，再执行循环结构的场合；而 do…while 语句则适用于先执行循环结构，再进行条件判断的场合。具体来说，在使用 while 语句时，如果条件不成立，则循环结构一次都不会执行；而在使用 do…while 语句时，即使条件不成立，程序也至少会执行一次循环结构。

4.5 for 循环

for 循环是 C# 中最常用、最灵活的一种循环结构，既能够用于循环次数已知的情况，又能够用于循环次数未知的情况，本节将对 for 循环的使用进行详细讲解。for 循环流程图如图 4.22 所示。

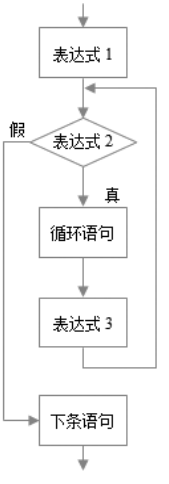


图 4.22 for 循环流程图

4.5.1 for 循环的一般形式

for 循环的常用语法格式如下。

```
for(表达式 1;表达式 2;表达式 3)
{
语句组
}
```

for 循环的执行过程如下。

第 1 步，求解表达式 1。

第 2 步，求解表达式 2。若表达式 2 的值为真，则执行循环体内的语句组，然后执行下面第 3 步；若值为假，则转到下面第 5 步。

第 3 步，求解表达式 3。

第 4 步，转回第 2 步执行。

第 5 步，循环结束，执行 for 循环接下来的语句。

for 循环最常用的格式如下。

```
for(循环变量赋初值；循环条件；循环变量增值)
{
语句组
}
```

示例 8．使用 for 循环挑战数学家高斯

使用 for 循环编写程序，实现从 1 到 100 的累加，代码如下。

```
01  static void Main(string[] args)
02  {
03      int iSum = 0;                       // 记录每次累加后的结果
04      for (int iNum = 1; iNum <= 100; iNum++)
05      {
06          iSum += iNum;                   // 把每次的 iNum 的值累加到上次累加的结果中
07      }
08      Console.WriteLine("1 到 100 的累加结果是: " + iSum);      // 输出结果
09      Console.ReadLine();
10  }
```

学习笔记

在第 4 行代码中，iNum 是循环变量，iNum 的初始值为 1，循环条件是 iNum 小于或等于 100，每次循环结束会对 iNum 进行累加。

学习笔记

本示例代码的运行结果与示例 6 代码的运行结果一样。

学习笔记

可以把 for 循环改成 while 循环，代码如下。

```
表达式 1;
```

```
while（表达式 2）
{
语句组
表达式 3;
}
```

4.5.2 for 循环的变体

for 循环在具体使用时有很多种变体形式，比如，可以省略“表达式 1”、“表达式 2”、“表达式 3”，或者 3 个表达式都省略，下面分别对 for 循环的常用变体形式进行讲解。

1. 省略“表达式 1”的情况

for 循环一般格式中的“表达式 1”可以省略，在 for 循环中“表达式 1”一般用于为循环变量赋初值，若省略了“表达式 1”，则需要在 for 循环的前面为循环条件赋初值。例如：

```
01  for(;iNum <= 100; iNum++)
02  {
03      sum += iNum;
04  }
```

此时，需要在 for 循环之前为 iNum 这个循环变量赋初值。程序在执行时，跳过“表达式 1”这一步，其他过程不变。

学习笔记

把上面的 for 循环语句改成 for(iNum <= 100; iNum ++) 并进行编译，会出现如图 4.23 所示的错误提示信息。出错是因为可以省略“表达式 1”，但是其后面的分号不能省略。

图 4.23　使用 for 循环语句缺少分号时出现的错误提示信息

2. 省略“表达式 2”的情况

在使用 for 循环时，“表达式 2”也可以省略。如果省略了“表达式 2”，则循环没有终止条件，会无限循环下去。针对这种使用方法，一般会配合后面将要讲解的 break 语句等来结束循环。

省略“表达式 2”的情况举例如下。

```
01  for(iNum = 1;;iNum++)
02  {
03      iSum += iNum;
04  }
```

这种情况的 for 循环相当于如下 while 语句。

```
01  while(true)              // 条件永远为真
02  {
03      iSum += iNum;
04      iNum ++;
05  }
```

3. 省略“表达式 3”的情况

在使用 for 循环时，“表达式 3”也可以省略，但此时程序设计者应另外设法保证循环变量的改变。例如，下面的代码在循环体中对循环变量的值进行了改变。

```
01  for(iNum = 1; iNum<=100;)
02  {
03      iSum += iNum;
04      iNum ++;
05  }
```

此时，在 for 循环的循环体内，对 iNum 这个循环变量的值进行了改变，这样才能使程序随着循环的进行逐渐趋近并满足程序终止条件。程序在执行时，跳过“表达式 3”这一步，其他过程不变。

4. 3 个表达式都省略的情况

for 循环语句中的 3 个表达式都可以省略，这种情况既没有对循环变量赋初值的操作，又没有循环条件，也没有改变循环变量的操作。在这种情况下，同省略“表达式 2”的情况类似，都需要配合使用 break 语句来结束循环，否则会形成死循环。

例如，下面的代码就将成为死循环，因为没有能够跳出循环的条件判断语句。

```
01  int i = 100;
02  for(;;)
03  {
04      Console.WriteLine(i);
05  }
```

4.5.3　for 循环中逗号的应用

在 for 循环语句中，“表达式 1”和“表达式 3”处都可以使用逗号表达式，即包含一个以上的表达式，中间用逗号分隔。例如，在“表达式 1”处为变量 iSum 和 iNum 同时赋初值。

```
01  for(iSum = 0, iNum = 1; iNum <= 100; iNum++)
02  {
03      iSum += iNum;
04  }
```

4.6 循环的嵌套

在一个循环里可以包含另一个循环，组成循环的嵌套，而里层循环可以继续进行循环嵌套，构成多层循环结构。

3 种循环（while 循环、do…while 循环和 for 循环）之间都可以相互嵌套。例如，下面的 6 种嵌套形式都是合法的嵌套形式。

（1）在 while 循环中嵌套 while 循环。

```
while（表达式）
{
        语句组
        while（表达式）
        {
             语句组
        }
}
```

（2）在 do…while 循环中嵌套 do…while 循环。

```
do
{
        语句组
        do
        {
            语句组
        }
        while（表达式）;
}while（表达式）;
```

（3）在 for 循环中嵌套 for 循环。

```
for( 表达式 ; 表达式 ; 表达式 )
{
        语句组
        for（表达式 ; 表达式 ; 表达式）
        {
             语句组
        }
}
```

（4）在 while 循环中嵌套 do…while 循环。

```
while(表达式)
{
        语句组
        do
        {
              语句组
        }
        while(表达式) ;
}
```

（5）在 while 循环中嵌套 for 循环。

```
while(表达式)
{
        语句组
        for(表达式;表达式;表达式)
        {
              语句组
        }
}
```

（6）在 for 循环中嵌套 while 循环。

```
for(表达式;表达式;表达式)
{
        语句组
        while(表达式)
        {
              语句组
        }
}
```

示例 9．打印九九乘法表

使用嵌套的 for 循环打印九九乘法表，代码如下。

```
01  static void Main(string[] args)
02  {
03      int iRow, iColumn;                                          // 定义行数和列数
04      for (iRow = 1; iRow < 10; iRow++)                           // 行数循环
05      {
06          for (iColumn = 1; iColumn <= iRow; iColumn++)           // 列数循环
07          {
08              // 输出每一行的数据
09              Console.Write("{0}*{1}={2} ", iColumn, iRow, iRow * iColumn);
10          }
11          Console.WriteLine();                                    // 换行
```

```
12      }
13      Console.ReadLine();
14  }
```

学习笔记

本示例的代码使用了双层 for 循环，第一层循环可以看成对乘法表的行数的控制，同时也是每一个乘法公式的第二个因子；第二层循环控制乘法表的列数，列数的最大值应该等于行数，因为输出的九九乘法表是以等腰直角三角形排列的，所以第二层循环的条件应该是在第一层循环的基础上建立的。

使用嵌套的 for 循环打印九九乘法表代码的运行结果如图 4.24 所示。

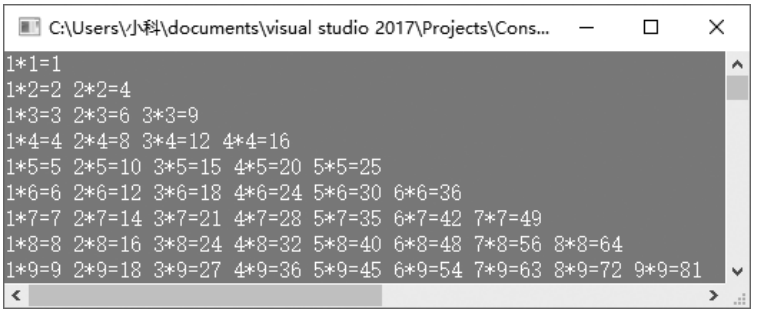


图 4.24　使用嵌套的 for 循环打印九九乘法表代码的运行结果

4.7　跳转语句

C# 中的跳转语句主要包括 break 语句和 continue 语句，跳转语句可以用于提前结束循环，本节将分别对它们进行详细讲解。

4.7.1　break 语句

在 4.3 节中已经介绍过使用 break 语句可以使流程跳出 switch 多分支结构，实际上，break 语句还可以用来跳出循环体，执行循环体之外的语句。break 语句通常应用在 switch 语句、while 语句、do…while 语句或 for 语句中，当多个 switch 语句、while 语句、do…while 语句或 for 语句互相嵌套时，break 语句只应用于最里层的语句。break 语句的语法格式如下。

```
break;
```

学习笔记

break 语句一般与 if 语句搭配使用，表示在某种条件下的循环结束。

示例 10．使用 break 语句跳出循环

修改示例 6，在 iNum 的值为 50 时退出循环，代码如下。

```
01  static void Main(string[] args)
02  {
03      int iNum = 1;                       //iNum 从 1 到 100 递增
04      int iSum = 0;                       // 记录每次累加后的结果
05      while (iNum <= 100)                 //iNum <= 100 是循环条件
06      {
07          iSum += iNum;                   // 把每次的 iNum 的值累加到上次累加的结果中
08          iNum++;                         // 每次循环 iNum 的值加 1
09          if (iNum == 50)                 // 判断 iNum 的值是否为 50
10              break;                      // 退出循环
11      }
12      Console.WriteLine("1 到 49 的累加结果是：" + iSum);       // 输出结果
13      Console.ReadLine();
14  }
```

学习笔记

第 9 行代码中的 iNum == 50 是判断条件，在该条件满足的情况下，执行第 10 行代码中的 break 语句，从而跳出 while 循环，执行后面的输出语句。

上面代码的运行结果如下。

```
1 到 49 的累加结果是：1225
```

4.7.2　continue 语句

continue 语句的作用是结束本次循环，它通常应用于 while 语句、do…while 语句或 for 语句中，用来忽略循环语句内位于它后面的代码而直接开始一次新的循环。当多个 while 语句、do…while 语句或 for 语句互相嵌套时，continue 语句只能使直接包含它的循环开始一次新的循环。continue 语句的语法格式如下。

```
continue;
```

学习笔记

continue 语句一般与 if 语句搭配使用，表示在某种条件下不执行后面的语句，直接开始下一次循环。

示例 11．计算 100 以内所有偶数的和

通过在 for 循环中使用 continue 语句实现计算 100 以内所有偶数的和，代码如下。

```
01  static void Main(string[] args)
02  {
03      int iSum = 0;                                  // 定义变量，用来存储偶数的和
04      int iNum = 1;                                  // 定义变量，用来作为循环变量
05      for (; iNum <= 100; iNum++)                    // 执行 for 循环
06      {
07          if (iNum % 2 == 1)                         // 判断是否为偶数
08              continue;                              // 继续下一次循环
09          iSum += iNum;                              // 记录偶数的和
10      }
11      Console.WriteLine("1 到 100 之间的偶数的和：" + iSum);      // 输出偶数的和
12      Console.ReadLine();
13  }
```

学习笔记

在第 7 行代码中，当所判断的数字是奇数时，会执行第 8 行代码中的 continue 语句，跳过后面的累加操作，直接进入下一次循环。

上面代码的运行结果如下。

```
1 到 100 之间的偶数的和：2550
```

第 5 章　数组的使用

数组是 C# 中一项非常特殊和重要的内容，在开发程序时，如果涉及数据不多，则可以使用变量存取和处理数据，但对于批量数据的处理，就要用到数组。数组是一种相同类型的、用一个标识符封装在一起的基本类型数据，可以使用一个统一的数组名和索引来唯一确定数组中的元素，它的执行效率非常高。本章将对 C# 中的数组进行详细讲解。

5.1　数组概述

假设正在编写一个程序，需要保存一个班级的学生的数学成绩（假设是整数）。如果有 5 个学生，若利用前面所学的知识实现，就需要声明 5 个整型变量来保存每个学生的成绩，代码如下。

```
int score1,score2,score3,score4,score5;
```

但如果有 100 个学生，难道要定义 100 个整型变量？这显然是不现实的，那怎么办呢？这时就可以使用数组来实现。

数组是具有相同数据类型的一组数据的集合，例如，球类的集合：足球、篮球、羽毛球等；电器的集合：电视机、洗衣机、电风扇等。前面学过的变量用来保存单个数据，而数组保存的则是多个相同类型的数据。变量与数组的比较效果如图 5.1 所示。

数组中的每一个变量称为数组的元素，数组能够容纳元素的数量称为数组的长度。数组中的每个元素都具有唯一的索引与其相对应，数组的索引从零开始。

数组是通过指定数组的元素类型、数组的维数（秩）及数组每个维数的上限和下限来进行定义的，即一个数组的定义需要包含以下几个要素。

（1）元素类型。

（2）数组的维数。

（3）每个维数的上限和下限。

数组的组成要素如图 5.2 所示。

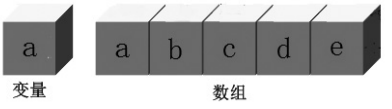


图 5.1　变量与数组的比较效果

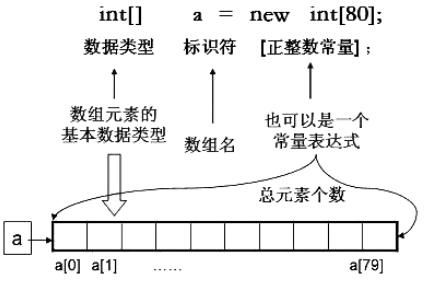


图 5.2　数组的组成要素

在程序设计中引入数组可以更有效地管理和处理数据，根据数组的维数将数组分为一维数组、多维数组和不规则数组等，下面分别进行讲解。

5.2　一维数组

一维数组实质上是一组相同类型的数据的线性集合，比如，把学校中学生们排列的一字长队看成一个一维数组，每个学生都是数组中的一个元素；再如，把一家快捷酒店看作一个一维数组，那么酒店里的每个房间都是这个数组中的元素。本节将介绍一维数组的创建及使用。

5.2.1　一维数组的创建

数组作为对象允许使用 new 关键字进行内存分配。在使用数组之前，必须首先定义数组变量所属的类型。一维数组的创建有如下两种形式。

1. 先声明，再用 new 关键字进行内存分配

声明一维数组使用以下形式。

```
数组元素类型 [ ] 数组名称 ;
```

数组元素的类型决定了数组的数据类型，它可以是 C# 中任意的数据类型。数组名称为一个合法的标识符，“[ ]”表明是一个数组。单个“[ ]”表示要创建的数组是一个一维数组。

例如，声明一维数组，代码如下。

```
01  int[] arr;          // 声明 int 类型的数组，数组中的每个元素都是 int 类型的数值
02  string[] str;       // 声明 string 类型的数组，数组中的每个元素都是 string 类型的数值
```

声明数组后，还不能访问它的任何元素，因为声明数组只是给出了数组名称和元素的

数据类型，要想使用数组，还要为它分配内存。在为数组分配内存时，必须指明数组的长度。为数组分配内存的语法格式如下。

```
数组名称 = new 数组元素类型 [ 数组元素的个数 ];
```

通过上面的语法可知，使用 new 关键字为数组分配内存时，必须指定数组元素的类型和数组元素的个数，即数组的长度。

例如，为数组分配内存，代码如下。

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

学习笔记

在使用 new 关键字为数组分配内存时，整型数组中各元素的初始值都为 0。

上述代码表示要创建一个有 5 个元素的整型数组，其数据存储形式如图 5.3 所示。

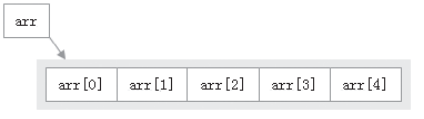


图 5.3　一维数组的数据存储形式

在图 5.3 中，arr 为数组名称，中括号“[ ]”中的值为数组的索引。数组通过索引来区分数组中不同的元素。数组的索引是从 0 开始的。由于创建的数组 arr 中有 5 个元素，因此数组中元素的索引为 0 ～ 4。

学习笔记

上述代码定义了一个长度为 5 的数组，但如果使用 arr[5]，则会引起索引超出范围异常，因为数组的索引是从 0 开始的。索引超出范围的异常提示如图 5.4 所示。

图 5.4　索引超出范围的异常提示

2. 声明的同时为数组分配内存

这种创建数组的方法是将数组的声明和内存的分配合在一起执行。

语法如下。

```
数组元素类型 [] 数组名称 = new 数组元素类型 [ 数组元素的个数 ];
```

例如，声明数组并为其分配内存，代码如下。

```
int[] month = new int[12];
```

上述代码创建名称为 month 的数组，并指定数组长度为 12。

5.2.2 一维数组的初始化

数组的初始化主要分为两种，即为单个数组元素赋值和同时为整个数组赋值，下面分别对其进行介绍。

1. 为单个数组元素赋值

为单个数组元素赋值即首先声明一个数组，并指定长度，然后为数组中的每个元素赋值。例如：

```
int[] arr = new int[5];                    // 定义一个 int 类型的一维数组
arr[0] = 1;                                // 为数组的第 1 个元素赋值
arr[1] = 2;                                // 为数组的第 2 个元素赋值
arr[2] = 3;                                // 为数组的第 3 个元素赋值
arr[3] = 4;                                // 为数组的第 4 个元素赋值
arr[4] = 5;                                // 为数组的第 5 个元素赋值
```

在使用这种方式对数组进行赋值时，通常使用循环实现。例如，上面代码可以修改如下。

```
01 int[] arr = new int[5];                 // 定义一个 int 类型的一维数组
02 for (int i = 0; i < arr.Length; i++)    // 遍历数组
03 {
04     arr[i] = i + 1;                     // 为遍历到的数组元素赋值
05 }
```

学习笔记

第 2 行代码中的 arr.Length 用来获取数组的长度。

学习笔记

数组大小必须与大括号中的元素个数相匹配，否则会产生编译错误。

2. 同时为整个数组赋值

在同时为整个数组赋值时需要使用大括号，将要赋值的数据包含在大括号中，并用逗号（,）隔开。例如：

```
string[] arrStr = new string[7] { "Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat" };
```

或者

```
string[] arrStr = new string[] { "Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat" };
```

或者

```
string[] arrStr = { "Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat" };
```

以上 3 种形式实现的效果是一样的，都定义了一个长度为 7 的 string 类型的数组，并进行了初始化。其中，后两种形式会自动计算数组的长度。

5.2.3　一维数组的使用

示例 1. 输出一年中每个月的天数

创建一个控制台应用程序，其中定义了一个 int 类型的一维数组，实现输出一年中每个月的天数，代码如下。

```
01  static void Main(string[] args)
02  {
03      // 创建并初始化一维数组
04      int[] day = new int[] { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
05      for (int i = 0; i < 12; i++)              // 利用循环输出信息
06      {
07          Console.WriteLine((i + 1) + "月有" + day[i] + "天"); // 输出的信息
08      }
09      Console.ReadLine();
10  }
```

学习笔记

第 7 行代码中的 day[i] 用来获取遍历到的数组元素。

输出一年中每个月的天数代码的运行结果如图 5.5 所示。

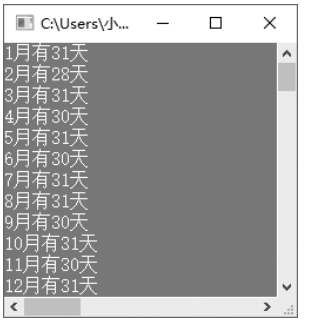


图 5.5　输出一年中每个月的天数代码的运行结果

5.3 二维数组

二维数组是一种特殊的多维数组。多维数组是指可以用多个索引访问的数组，声明时，用多个中括号（[]）或在中括号内加逗号表明是多维数组，有 *n* 个中括号或中括号内有 *n* 个逗号的数组就是 *n*+1 维数组。下面以最常用的二维数组为例讲解多维数组的使用。

5.3.1 二维数组的创建

快捷酒店每个楼层都有很多房间，这些房间都可以构成一维数组。如果这个酒店有 500 个房间，并且所有房间都在同一个楼层里，那么拿到 499 号房间钥匙的旅客可能就不高兴了，因为从 1 号房间走到 499 号房间要花很长时间。因此，每个酒店都不是只有一个楼层，而是有很多个楼层，每个楼层都会有很多房间，这样就形成一个立体的结构，把大量的房间均摊到每个楼层，这种结构就是二维表结构。在计算机中，二维表结构可以使用二维数组来表示。使用二维表结构表示快捷酒店每个楼层的房间号的效果如图 5.6 所示。

楼层	房间号						
一楼	1101	1102	1103	1104	1105	1106	1107
二楼	2101	2102	2103	2104	2105	2106	2107
三楼	3101	3102	3103	3104	3105	3106	3107
四楼	4101	4102	4103	4104	4105	4106	4107
五楼	5101	5102	5103	5104	5105	5106	5107
六楼	6101	6102	6103	6104	6105	6106	6107
七楼	7101	7102	7103	7104	7105	7106	7107

图 5.6 使用二维表结构表示快捷酒店每个楼层的房间号的效果

二维数组常用于表示二维表，表中的信息以行和列的形式表示，第一个下标代表元素所在的行，第二个下标代表元素所在的列。

二维数组的声明语法如下。

```
type[,] arrayName;
type[][] arrayName;
```

type：二维数组的数据类型。

arrayName：二维数组的名称。

例如，声明一个 int 类型的二维数组，可以使用下面两种方式。

```
int[,] myarr;                    // 声明一个 int 类型的二维数组，名称为 myarr
```

或者

```
int[][] myarr;                   // 声明一个 int 类型的二维数组，名称为 myarr
```

同一维数组一样，二维数组在声明时也没有分配内存，同样可以使用 new 关键字来分配内存，然后才可以访问每个元素。

对于二维数组，有如下两种为数组分配内存的方式。

1. 直接为每一维分配内存

例如，定义一个二维数组，并直接为其分配内存，代码如下。

```
int[,] a = new int[2, 4];        // 定义一个 2 行 4 列的 int 类型的二维数组
```

上述代码创建了一个 int 类型的二维数组 a，二维数组 a 中包括两个长度为 4 的一维数组，二维数组内存分配（第一种方式）如图 5.7 所示。

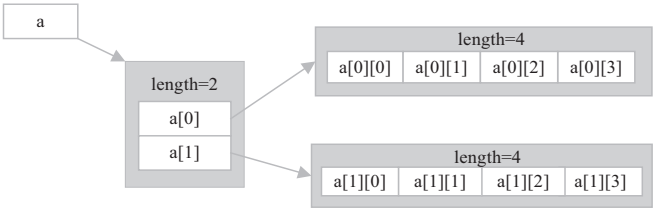


图 5.7　二维数组内存分配（第一种方式）

2. 分别为每一维分配内存

例如，定义一个二维数组，分别为每一维分配内存，代码如下。

```
01  int[][] a = new int[2][];          // 定义一个 2 行的 int 类型的二维数组
02  a[0] = new int[2];                 // 初始化二维数组的第 1 行有 2 个元素
03  a[1] = new int[3];                 // 初始化二维数组的第 2 行有 3 个元素
```

通过第二种方式为二维数组分配内存，如图 5.8 所示。

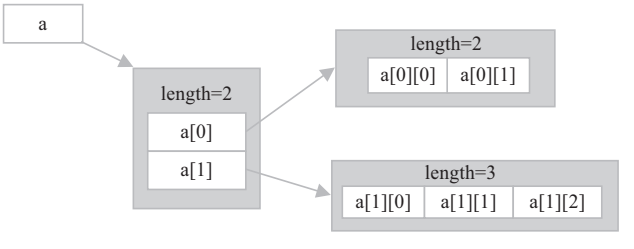


图 5.8　二维数组内存分配（第二种方式）

学习笔记

在上述代码中，由于为每一维分配的内存不同，因此，a 相当于一个不规则二维数组。

5.3.2 二维数组的初始化

二维数组有两个索引（下标），构成一个由行和列组成的矩阵，如图 5.9 所示。

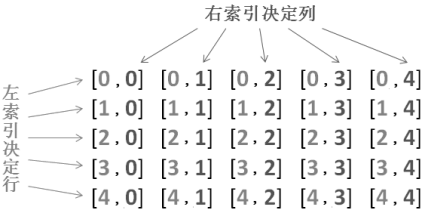


图 5.9 二维数组索引与行和列的关系

二维数组的初始化主要有：为单个二维数组元素赋值、为每一维数组元素赋值、同时为整个二维数组赋值，下面分别对其进行介绍。

1. 为单个二维数组元素赋值

为单个二维数组元素赋值即首先声明一个二维数组，并指定行数和列数，然后为二维数组中的每个元素进行赋值。例如：

```
int[,] myarr = new int[2, 2];                    // 定义一个 int 类型的二维数组
myarr[0, 0] = 0;                                 // 为二维数组中的第 1 行第 1 列赋值
myarr[0, 1] = 1;                                 // 为二维数组中的第 1 行第 2 列赋值
myarr[1, 0] = 1;                                 // 为二维数组中的第 2 行第 1 列赋值
myarr[1, 1] = 2;                                 // 为二维数组中的第 2 行第 2 列赋值
```

在使用这种方式对二维数组进行赋值时，通常使用嵌套的循环实现。例如，上述代码可以修改如下。

```
int[,] myarr = new int[2, 2];                    // 定义一个 int 类型的二维数组
for (int i = 0; i < 2; i++)                      // 遍历二维数组的行
{
    for (int j = 0; j < 2; j++)                  // 遍历二维数组的列
    {
        myarr[i, j] = i + j;                     // 为遍历到的二维数组中的第 i 行第 j 列赋值
    }
}
```

2. 为每一维数组元素赋值

在为二维数组中的每一维数组元素赋值时，首先需要使用“数组类型 [][]”形式声明一个数组，并指定数组的行数，然后分别为每一维数组元素赋值。例如：

```
int[][] myarr = new int[2][];                    // 定义一个 2 行的 int 类型的二维数组
myarr[0] = new int[] { 0, 1 };                   // 初始化二维数组的第 1 行
myarr[1] = new int[] { 1, 2 };                   // 初始化二维数组的第 2 行
```

3. 同时为整个二维数组赋值

在同时为整个二维数组赋值时需要使用嵌套的大括号，并将要赋值的数据包含在里层大括号中，每个大括号中间用逗号（,）隔开。例如：

```
int[,] myarr = new int[2,2] { { 12, 0 }, { 45, 10 } };
```

或者

```
int[,] myarr = new int[,]{ { 12, 0 }, { 45, 10 } };
```

或者

```
int[,] myarr = {{12,0},{45,10}};
```

以上 3 种形式实现的效果是一样的，都定义了一个长度为 2 行 2 列的 int 类型的二维数组，并对其进行初始化。其中，后两种形式会自动计算数组的行数和列数。

5.3.3　二维数组的使用

示例 2．模拟客车售票系统

创建一个控制台应用程序，模拟制作一个简单的客车售票系统。假设客车的座位数是 9 行 4 列，使用一个二维数组记录客车售票系统中的所有座位号，并在每个座位号上都显示“【有票】”。然后用户输入一个坐标位置，按回车键，即可将该座位号显示为“【已售】”。代码如下。

```
static void Main(string[] args)
{
    Console.Title = " 简单客车售票系统 ";                  // 设置控制台标题
    string[,] zuo = new string[9, 4];                    // 定义二维数组
    for (int i = 0; i < 9; i++)                          //for 循环开始
    {
        for (int j = 0; j < 4; j++)                      //for 循环开始
        {
            zuo[i, j] = "【有票】";                       // 初始化二维数组
        }
    }
```

```
12      string s = string.Empty;                                    //定义字符串变量
13      while (true)                                                //开始售票
14      {
15          Console.Clear();                                        //清空控制台信息
16          Console.WriteLine("\n          简单客车售票系统" + "\n");//输出字符串
17          for (int i = 0; i < 9; i++)
18          {
19              for (int j = 0; j < 4; j++)
20              {
21                  System.Console.Write(zuo[i, j]);   //输出售票信息
22              }
23              Console.WriteLine();                               //输出换行符
24          }
25          Console.Write("请输入坐位行号和列号（如：0,2）输入 q 键退出：");
26          s = Console.ReadLine();                                //售票信息输入
27          if (s == "q") break;                                   //输入字符串 "q" 退出系统
28          string[] ss = s.Split(',');                            //拆分字符串
29          int one = int.Parse(ss[0]);                            //得到座位行数
30          int two = int.Parse(ss[1]);                            //得到座位列数
31          zuo[one, two] = "【已售】";                             //标记售出票状态
32      }
33  }
```

学习笔记

第 13 行代码中的 while (true) 用来设置一个无限循环，以便能够循环输入坐标位置。

第 28 行代码用到了字符串的 Split() 方法，该方法用来根据指定的符号对字符串进行分割，这里了解即可，该方法将在第 6 章进行详细讲解。

模拟客车售票系统代码的运行结果如图 5.10 所示。

图 5.10　模拟客车售票系统代码的运行结果

5.3.4　不规则数组的定义

前面讲的二维数组是行和列固定的矩形方阵，如 4×4、3×2 等，另外，C# 中还支持不规则的数组。例如，在二维数组中，不同行的元素个数完全不同。

```
int[][] a = new int[3][];      // 创建二维数组，指定行数，不指定列数
a[0] = new int[5];             // 为第 1 行分配 5 个元素
a[1] = new int[3];             // 为第 2 行分配 3 个元素
a[2] = new int[4];             // 为第 3 行分配 4 个元素
```

对于上述代码中定义的不规则二维数组，其所占的内存空间如图 5.11 所示。

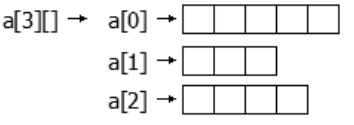


图 5.11　不规则二维数组所占的内存空间

5.4　数组与 Array 类

C# 中的数组是由 Array 类派生而来的引用对象，它们的关系图如图 5.12 所示。

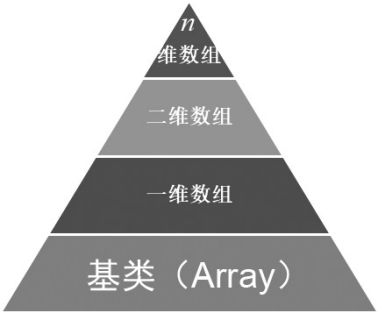


图 5.12　数组与 Array 类的关系图

可以使用 Array 类中的各种属性或方法对数组进行各种操作。例如，可以使用 Array 类的 Length 属性获取数组的长度，也可以使用其 Rank 属性获取数组的维数。

Array 类的常用方法及说明如表 5.1 所示。

表 5.1　Array 类的常用方法及说明

方　法	说　明
Copy	将数组中的指定元素复制到另一个 Array 数组中
CopyTo	从指定的目标数组索引处开始，将当前一维数组中的所有元素复制到另一个一维数组中
Exists	判断数组中是否包含指定的元素
GetLength	获取 Array 数组中的指定维度中的元素数
GetLowerBound	获取 Array 数组中指定维度的下限
GetUpperBound	获取 Array 数组中指定维度的上限
GetValue	获取 Array 数组中指定位置的值
Reverse	反转一维 Array 数组中元素的顺序
SetValue	设置 Array 数组中指定位置的元素
Sort	对一维 Array 数组中的元素进行排序

示例 3．打印杨辉三角

使用数组打印杨辉三角。杨辉三角是一个由数字排列成的三角形数表，其最本质的特征是它的两条边都是由数字 1 组成的，而其余的数则等于它上方的两个数之和。代码如下。

```
01  static void Main(string[] args)
02  {
03      int[][] Array_int = new int[10][];              // 定义一个 10 行的二维数组
04      // 在数组中记录杨辉三角的值
05      for (int i = 0; i < Array_int.Length; i++)      // 遍历行数
06      {
07          Array_int[i] = new int[i + 1];              // 定义二维数组的列数
08          for (int j = 0; j < Array_int[i].Length; j++)   // 遍历二维数组的列数
09          {
10              if (i <= 1)                             // 如果是数组的前两行
11              {
12                  Array_int[i][j] = 1;                // 将其设置为 1
13                  continue;
14              }
15              else
16              {
17                  if (j == 0 || j == Array_int[i].Length - 1) // 如果是行首或行尾
18                      Array_int[i][j] = 1;            // 将其设置为 1
19                  else                                // 根据杨辉算法进行计算
20                      Array_int[i][j] = Array_int[i - 1][j - 1] + Array_int[i - 1][j];
21              }
22          }
23      }
24      for (int i = 0; i <= Array_int.Length-1; i++) // 输出杨辉三角
25      {
26          // 循环控制每行前面打印的空格数
```

```
27          for (int k = 0; k <= Array_int.Length - i; k++)
28          {
29              Console.Write("    ");
30          }
31            // 循环控制每行打印的数据
32          for (int j = 0; j < Array_int[i].Length; j++)
33          {
34              Console.Write("{0}     ", Array_int[i][j]);
35          }
36          Console.WriteLine();                          // 换行
37      }
38      Console.ReadLine();
39  }
```

学习笔记

在第 17 行代码中，j==0 判断是不是行首，j == Array_int[i].Length - 1 判断是不是行尾，因为在杨辉三角中，每一行的行首和行尾都是 1，所以这里进行了特殊处理。

使用数组打印杨辉三角代码的运行结果如图 5.13 所示。

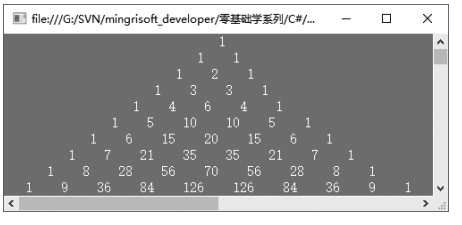


图 5.13　使用数组打印杨辉三角代码的运行结果

5.5　数组的基本操作

在开发程序时，数组常用的操作是遍历及排序，本节将对这两种操作分别进行详细讲解。

5.5.1　使用 foreach 语句遍历数组

除了使用循环输出数组的元素，C# 还提供了一种 foreach 语句，该语句用来遍历集合中的每个元素，而数组也属于集合类型，因此，foreach 语句可以遍历数组。foreach 语句的语法格式如下。

```
foreach(【类型】 【迭代变量名】 in 【集合】)
{
        语句
}
```

其中，【类型】和【迭代变量名】用于声明迭代变量。迭代变量相当于一个范围覆盖整个语句块的局部变量，在 foreach 语句执行期间，迭代变量表示当前正在为其执行迭代的集合元素。【集合】必须有一个从该集合的元素类型到迭代变量的类型的显式转换，如果【集合】的值为 null，则会出现异常。

foreach 语句执行流程如图 5.14 所示。

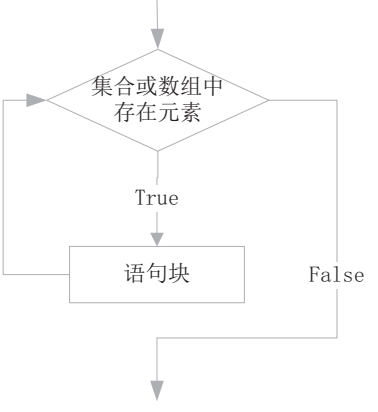


图 5.14　foreach 语句执行流程

示例 4．输出学生的成绩信息及班级平均成绩、总成绩

在控制台输入学生的学号，以及语文成绩、数学成绩、英语成绩，然后输出学生的各科成绩信息、平均成绩和总成绩。代码如下。

```
01  static void Main(string[] args)
02  {
03      Console.Write(" 请输入本班学生总数：");                    // 提示信息
04      int studentcout = Convert.ToInt32(Console.ReadLine());   // 记录学生总数
05      int[,] achivement = new int[studentcout,4];// 创建二维数组，用来记录学生各科成绩
06      for (int i = 0; i < studentcout; i++)          // 遍历二维数组的每一行
07      {
08          Console.Write(" 请输入第 " + (i + 1) + " 个学生的编号："); // 显示第几个学生
09          achivement[i,0] = Convert.ToInt32(Console.ReadLine());    // 记录学生编号
10          Console.Write(" 请输入语文成绩：");
11          achivement[i,1] = Convert.ToInt32(Console.ReadLine());// 记录学生语文成绩
12          Console.Write(" 请输入数学成绩：");
13          achivement[i,2] = Convert.ToInt32(Console.ReadLine());// 记录学生数学成绩
14          Console.Write(" 请输入英语成绩：");
15          achivement[i,3] = Convert.ToInt32(Console.ReadLine());// 记录学生英语成绩
```

```
16      }
17      Console.WriteLine(" 学生成绩结果如下 :");
18      Console.WriteLine("-----------------------------------------------");
19      Console.WriteLine(" 学生编号 \t 语文成绩 \t 数学成绩 \t 英语成绩 \t 平均成绩 \t 总成绩 ");
20      for (int i = 0; i < achivement.GetLength(0); i++)     // 遍历行
21      {
22          double sum = 0;                                   // 总成绩
23           double ave = 0;                                  // 平均成绩
24           for (int j = 0; j < achivement.GetLength(1); j++)   // 遍历列
25           {
26               Console.Write(achivement[i,j] + "\t\t");   // 输出每个学生的成绩
27               if (j > 0)
28               {
29                   sum += achivement[i,j];                  // 计算总成绩
30               }
31           }
32           ave = sum / 3;                                   // 计算平均成绩
33            // 输出平均成绩和总成绩
34           Console.Write(string.Format("{0:F2}", ave) + "\t\t" + (int)sum  + "\n");
35       }
36       Console.ReadLine();
37  }
```

输出学生的成绩信息及班级平均成绩、总成绩代码的运行结果如图 5.15 所示。

学习笔记

foreach 语句通常用来遍历集合，而数组也是一种简单的集合。

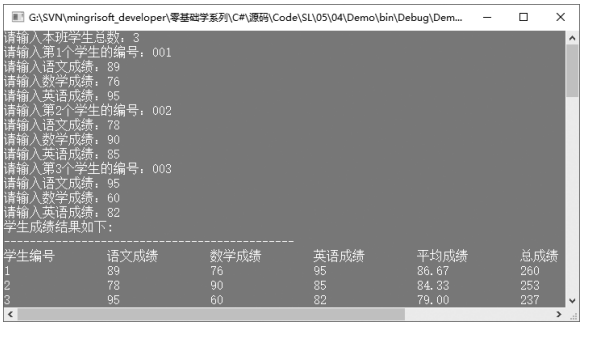


图 5.15　输出学生的成绩信息及班级平均成绩、总成绩代码的运行结果

5.5.2　对数组进行排序

C# 提供了用于对数组进行排序的方法 Array.Sort 和 Array.Reverse，下面分别对其进行讲解。

1. Array.Sort 方法

Array.Sort 方法用于对一维 Array 数组中的元素进行排序， 该方法有多种形式，其常用的两种形式如下。

```
public static void Sort(Array array)
public static void Sort(Array array,int index,int length)
```

array：要排序的一维 Array 数组。

index：排序范围的起始索引。

length：排序范围内的元素数。

例如，使用 Array.Sort 方法对数组中的元素进行从小到大排序，代码如下。

```
01  int[] arr = new int[] { 3, 9, 27, 6, 18, 12, 21, 15 };
02  Array.Sort(arr);                        // 对数组元素进行排序
```

学习笔记

在 Array.Sort 方法中所用到的数组不能为空，也不能是多维数组，它只对一维数组进行排序。

2. Array.Reverse 方法

Array.Reverse 方法用于反转一维 Array 数组中元素的顺序，该方法有两种形式，分别如下。

```
public static void Reverse(Array array)
public static void Reverse(Array array,int index,int length)
```

array：要反转的一维 Array 数组。

index：要反转的部分的起始索引。

length：要反转的部分的元素数。

例如，下面使用 Array. Reverse 方法对数组的元素进行反转，代码如下。

```
01  int[] arr = new int[] { 3, 9, 27, 6, 18, 12, 21, 15 };
02  Array.Reverse(arr);                          // 对数组元素进行反转
```

学习笔记

对数组进行反转并不是反向排序，比如，有一个一维数组，元素为“36 89 76 45 32”，反转之后为“32 45 76 89 36”，而不是“89 76 45 36 32”。

第 6 章　看似简单的字符串

字符串几乎是所有编程语言在项目开发过程中涉及最多的部分。大部分项目的运行结果都需要以文本的形式展示给客户。例如，财务系统的总账报表、电子游戏的比赛结果、火车站的列车时刻表等，这些都是经过程序精密的计算、判断和梳理，将我们想要的内容以文本形式直观地展示出来。“开发一个项目，基本上就是在不断地处理字符串”。本章将对 C# 中的字符串进行详细讲解。

6.1　什么是字符串

前面的章节介绍了 char 类型可以保存字符，但它只能表示单个字符。如果要用 char 类型来展示“版权说明”“功能简介”之类的内容，那程序员就无计可施了，这时可以使用 C# 中常用的字符串。

字符串，顾名思义，就是用字符拼接成的文本值。字符串在存储上与数组类似，不仅字符串的长度可取，而且每一位上的元素也可取。在 C# 中，可以通过 string 类创建字符串。

6.2　字符串的声明与初始化

C# 中的字符串通过 string 类来创建，本节将对字符串的声明及初始化进行讲解。

6.2.1　声明字符串

在 C# 中，字符串必须包含在一对双引号（""）之内，如下所示。

```
"23.23"、"ABCDE"、" 你好 "
```

双引号内的内容都是字符串常量，字符串常量是系统能够显示的任何文字信息，甚至是单个字符。

学习笔记

在 C# 中，由双引号（""）包围的都是字符串，不能作为其他数据类型使用。例如，"1+2" 的输出结果永远不会是 3。

可以通过以下语法格式来声明字符串。

```
string str = [null]
```

string：指定该变量为字符串类型。

str：任意有效的标识符，表示字符串变量的名称。

null：如果省略 null，表示 str 变量是未初始化的状态，否则，表示声明的字符串的值等于 null。

例如，声明一个字符串变量 strName，代码如下。

```
string strName;
```

6.2.2 字符串的初始化

声明字符串之后，如果要使用该字符串，需要对其进行初始化，否则会出现错误。例如，下面的代码。

```
01 string str;
02 Console.WriteLine(str);
```

运行上述代码将会出现如图 6.1 所示的错误提示。

图 6.1 使用未初始化的变量时出现的错误提示

从图 6.1 中可以看出，要使用一个变量，必须首先对其进行初始化（赋值）。对字符串进行初始化的方法主要有以下几种。

（1）引用字符串常量，示例代码如下。

```
01 string a = "时间就是金钱，我的朋友。";
02 string b = "锄禾日当午";
03 string str1, str2;
04 str1 = "We are students";
05 str2 = "We are students";
```

学习笔记

当两个字符串对象引用相同的常量时，就会具有相同的实体。例如，上述代码中的 str1 和 str2 引用了相同的常量，如图 6.2 所示。

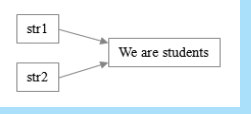


图 6.2　str1 和 str2 对象引用了相同的常量

（2）利用字符数组初始化，示例代码如下。

```
01  char[] charArray = { 't', 'i', 'm', 'e' };
02  string str = new string(charArray);
```

（3）提取字符数组中的一部分初始化字符串，示例代码如下。

```
01  char[] charArray = { '时', '间', '就', '是', '金', '钱' };
02  string str = new string(charArray, 4, 2);
```

6.3　提取字符串信息

当字符串作为对象时，可以通过相应的方法提取其中的有效信息，如获取某字符串的长度、某个索引位置的字符等。本节将对常用的提取字符串信息的方法进行讲解。

6.3.1　获取字符串长度

获取字符串的长度可以使用 string 类的 Length 属性，其语法格式如下。

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

属性值：表示当前字符串中字符的数量。

例如，定义一个字符串变量，并为其赋值，然后使用 Length 属性获取该字符串的长度，代码如下。

```
01  string num = "12345 67890";
02  int size = num.Length;
```

输出变量 size 的值，得到的结果是 11，这表示使用 Length 属性返回的字符串长度是包括字符串中的空格的。

6.3.2 获取指定位置的字符

获取指定位置的字符可以使用 string 类的 Chars 属性，其语法格式如下。

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

index：当前在字符串中的位置。

属性值：位于 index 位置的字符。

例如，定义一个字符串变量，并为其赋值，然后获取该字符串索引位置 5 处的字符并输出，代码如下。

```
01 string str = "努力工作是人生最好的投资";          // 创建字符串对象 str
02 char chr = str[5];                  // 将字符串 str 中索引位置为 5 的字符赋值给 chr
03 Console.WriteLine("字符串中索引位置为 5 的字符是：" + chr); // 输出 chr
```

运行结果如下。

```
字符串中索引位置为 5 的字符是：人
```

学习笔记

字符串中的索引位置是从 0 开始的。

6.3.3 获取子字符串索引位置

string 类提供了两种查找字符串索引的方法，即 IndexOf 方法与 LastIndexOf 方法。其中，IndexOf 方法返回的是搜索的字符或字符串首次出现的索引位置，而 LastIndexOf 方法返回的是搜索的字符或字符串最后一次出现的索引位置，本节将分别对这两种方法进行详细讲解。

1. IndexOf 方法

IndexOf 方法返回的是搜索的字符或字符串首次出现的索引位置，其常用的几种语法格式如下。

```
public int IndexOf(char value)
public int IndexOf(string value)
public int IndexOf(char value,int startIndex)
public int IndexOf(string value,int startIndex)
public int IndexOf(char value,int startIndex,int count)
```

```
public int IndexOf(string value,int startIndex,int count)
```

value：要搜索的字符或字符串。

startIndex：搜索起始位置。

count：要检查的字符位置数。

返回值：如果找到字符或字符串，则为 value 的从零开始的索引位置；如果未找到字符或字符串，则为 -1。

例如，查找字符 e 在字符串 str 中第一次出现的索引位置，代码如下。

```
01  string str = "We are the world";
02  int size = str.IndexOf('e');                    //size 的值为 1
```

若要理解字符串的索引位置，则要对字符串的下标有所了解。在计算机中，string 对象是用数组表示的。字符串的下标为 0 ～ Length-1。上面代码中的字符串 str 的下标排列如图 6.3 所示。

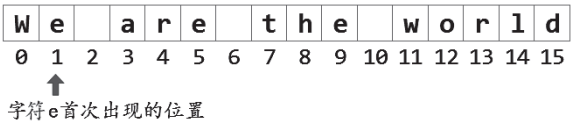


图 6.3　字符串 str 的下标排列

学习笔记

在日常开发工作中，经常会遇到判断一个字符串中是否包含某个字符或某个子字符串的情况，这时就需要使用 IndexOf 方法。

示例 1．查找“r”在“We are the world”中出现的位置

查找字符串“We are the world”中“r”第一次、第二次、第三次出现的索引位置，代码如下。

```
01  static void Main(string[] args)
02  {
03      string str = "We are the world";          // 创建字符串
04      int firstIndex = str.IndexOf("r"); // 获取字符串中“r”第一次出现的索引位置
05      // 获取字符串中“r”第二次出现的索引位置，从第一次出现的索引位置之后开始查找
06      int secondIndex = str.IndexOf("r", firstIndex + 1);
07      // 获取字符串中“r”第三次出现的索引位置，从第二次出现的索引位置之后开始查找
08      int thirdIndex = str.IndexOf("r", secondIndex + 1);
09      // 输出三次获取的索引位置
10      Console.WriteLine("r 第一次出现的索引位置是：" + firstIndex);
```

```
11      Console.WriteLine("r 第二次出现的索引位置是：" + secondIndex);
12      Console.WriteLine("r 第三次出现的索引位置是：" + thirdIndex);
13      Console.ReadLine();
14  }
```

学习笔记

第 6 行代码使用 firstIndex + 1 作为第二次查找的起始查找位置，第 8 行代码使用 secondIndex + 1 作为第三次查找的起始查找位置。

在“We are the world”中查找“r”第一次、第二次、第三次出现的索引位置代码的运行结果如图 6.4 所示。

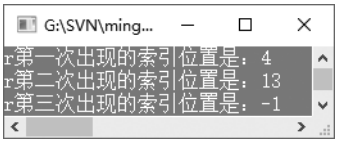


图 6.4　在“We are the world”中查找“r”第一次、第二次、第三次出现的索引位置代码的运行结果

从图 6.4 中可以看出，由于字符串中只有两个“r”，所以程序输出了这两个“r”的索引位置，第三次搜索时已经找不到“r”了，所以返回 -1。

2. LastIndexOf 方法

LastIndexOf 方法返回的是搜索的字符或字符串最后一次出现的索引位置，其常用的几种语法格式如下。

```
public int LastIndexOf(char value)
public int LastIndexOf(string value)
public int LastIndexOf(char value,int startIndex)
public int LastIndexOf(string value,int startIndex)
public int LastIndexOf(char value,int startIndex,int count)
public int LastIndexOf(string value,int startIndex,int count)
```

value：要搜索的字符或字符串。

startIndex：搜索起始位置。

count：要检查的字符位置数。

返回值：如果找到字符或字符串，则为 value 的从零开始的索引位置；如果未找到字符或字符串，则为 -1。

例如，查找字符 e 在字符串 str 中最后一次出现的索引位置，代码如下。

```
01  string str = "We are the world";
```

```
02  int size = str.LastIndexOf('e');              //size 的值为 9
```

字符 e 在字符串 str 中最后一次出现的索引位置如图 6.5 所示。

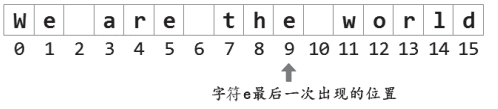


图 6.5　字符 e 在字符串 str 中最后一次出现的索引位置

6.3.4　判断字符串首、尾内容

string 类提供了两种查找字符串首、尾内容的方法，即 StartsWith 方法与 EndsWith 方法。其中，StartsWith 方法用来判断字符串是否以指定的内容开始，而 EndsWith 方法用来判断字符串是否以指定的内容结束，本节将分别对这两种方法进行详细讲解。

1. StartsWith 方法

StartsWith 方法用来判断字符串是否以指定的内容开始，其常用的几种语法格式如下。

```
public bool StartsWith(string value)
public bool StartsWith(string value,bool ignoreCase,CultureInfo culture)
```

value：要比较的字符串。

ignoreCase：如果在比较过程中忽略大小写，则为 true；否则为 false。

culture：CultureInfo 对象，用来确定如何对字符串与 value 进行比较的区域性信息。如果 culture 为 null，则使用当前区域性。

返回值：如果 value 与字符串的开头匹配，则为 true；否则为 false。

例如，使用 StartsWith 方法判断一个字符串是否以“梦想”开始，代码如下。

```
01  string str = "梦想还是要有的，万一实现了呢！ "; // 定义一个字符串，并对其进行初始化
02  bool result = str.StartsWith("梦想 ");       // 判断 str 是否以“梦想”开始
03  Console.WriteLine(result);
```

上面代码的运行结果为 true。

学习笔记

如果在判断某一个英文字符串是否以某字母开始时，需要忽略大小写，则可以使用第二种语法格式，并将第二个参数设置为 true。例如，定义一个字符串“Keep on going never give up”，然后使用 StartsWith 方法判断该字符串是否以“keep”开始，代码如下。

```
01  string str = "Keep on going never give up";
02  bool result = str.StartsWith("keep", true, null); // 判断 str 是否以“keep”开始
03  Console.WriteLine(result);
```

上面代码的返回结果为 true，因为这里使用了 StartsWith 方法的第二种语法格式，并且第二个参数为 true，所以在比较时，“Keep”和“keep”会忽略大小写，返回结果为 true。

2. EndsWith 方法

EndsWith 方法用来判断字符串是否以指定的内容结束，其常用的几种语法格式如下：

```
public bool EndsWith(string value)
public bool EndsWith(string value,bool ignoreCase,CultureInfo culture)
```

value：要比较的字符串。

ignoreCase：如果在比较过程中忽略大小写，则为 true；否则为 false。

culture：CultureInfo 对象，用来确定如何对字符串与 value 进行比较的区域性信息。如果 culture 为 null，则使用当前区域性。

返回值：如果 value 与字符串的末尾匹配，则为 true；否则为 false。

学习笔记

如果在比较时需要忽略大小写，则通常使用第二种形式，并将第二个参数设置为 true。

例如，使用 EndsWith 方法判断一个字符串是否以句号（。）结束，代码如下。

```
01  string str = "梦想还是要有的，万一实现了呢！";  // 定义一个字符串，并初始化
02  bool result = str.EndsWith("。");              // 判断 str 是否以“。”结尾
03  Console.WriteLine(result);
```

上面代码的运行结果为 false。

6.4 字符串操作

6.4.1 字符串的拼接

使用“+”运算符可完成对多个字符串的拼接，“+”运算符可以连接多个字符串并产

生一个 string 对象。

例如，定义两个字符串，使用“+”运算符连接，代码如下。

```
01  string s1 = "hello";           // 声明 string 对象 s1
02  string s2 = "world";           // 声明 string 对象 s2
03  string s = s1 + " " + s2;      // 将对象 s1 和 s2 连接后的结果赋值给 s
```

学习笔记

C# 中一个相连的字符串不能分开在两行中写，如下所示。

```
01  Console.WriteLine("I like
02  C#");
```

这种写法是错误的，如果一个字符串太长，为了便于阅读，则可以将这个字符串分在两行中书写，此时就可以使用“+”将两个字符串拼接起来，之后在加号处换行。因此，上面的语句可以修改如下。

```
01  Console.WriteLine("I like" +
02   "C#");
```

6.4.2　比较字符串

在对字符串值进行比较时，可以使用前面学过的关系运算符“==”实现。

例如，使用关系运算符“==”比较两个字符串的值，代码如下。

```
01  string str1 = "mingrikeji";
02  string str2 = "mingrikeji";
03  Console.WriteLine((str1 == str2));
```

上面代码的输出结果为 true。

除了使用关系运算符“==”，在 C# 中常见的比较字符串的方法还有 Equals 方法，下面对其进行讲解。

Equals 方法主要用于比较两个字符串是否相同，如果相同则返回值是 true，否则为 false，其常用的两种语法格式如下。

```
public bool Equals (string value)
public static bool Equals (string a,string b)
```

value：与此字符串比较的字符串。

a 和 b：要进行比较的两个字符串。

返回值：第一种语法格式，如果 value 参数的值与此字符串相同，则为 true；否则为

false。第二种语法格式，如果 a 的值与 b 的值相同，则为 true；否则为 false。 如果 a 和 b 均为 null，则该方法返回 true。

示例 2．验证用户名和密码是否正确

假设明日学院网站的登录用户名和密码分别是 mr、mrsoft，请编程验证用户输入的用户名和密码是否正确，代码如下。

```
04  static void Main(string[] args)
05  {
06      Console.Write(" 请输入登录用户名： ");
07      string name = Console.ReadLine();                  // 记录输入的用户名
08      Console.Write(" 请输入登录密码： ");
09      string pwd = Console.ReadLine();                   // 记录输入的密码
10      if (name=="mr" && pwd.Equals("mrsoft"))            // 判断用户名和密码是否正确
11      {
12          Console.WriteLine(" 登录成功，欢迎你访问明日学院网站……");
13      }
14      else
15      {
16          Console.WriteLine(" 输入的用户名和密码错误！！！ ");
17      }
18      Console.ReadLine();
19  }
```

运行上面代码，用户名和密码正确的效果如图 6.6 所示，用户名和密码不正确的效果如图 6.7 所示。

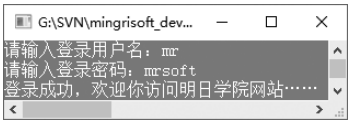


图 6.6　用户名和密码正确的效果

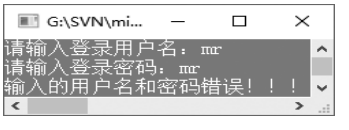


图 6.7　用户名和密码不正确的效果

6.4.3　字符串的大小写转换

在对字符串进行大、小写转换时，需要使用 string 类提供的 ToUpper 方法和 ToLower 方法，其中，ToUpper 方法用来将字符串转换为大写形式，而 ToLower 方法用来将字符串转换为小写形式，它们的语法格式如下。

```
public string ToUpper()
public string ToLower()
```

学习笔记

如果字符串中没有需要被转换的字符（如数字或汉字），则返回原字符串。

例如，定义一个字符串，赋值为“Learn and live”，分别用大写、小写两种格式输出该字符串，代码如下。

```
01  string str = "Learn and live";
02  Console.WriteLine(str.ToUpper());          // 大写输出
03  Console.WriteLine(str.ToLower());          // 小写输出
```

运行结果如下。

```
LEARN AND LIVE
learn and live
```

学习笔记

在各种网站的登录页面，验证码的输入通常都是不区分大小写的，对于这样的情况就可以使用 ToUpper 或 ToLower 方法将网页显示的验证码和用户输入的验证码同时转换为大写或小写，以方便验证。

6.4.4　格式化字符串

在 C# 中，string 类提供了一种静态的 Format 方法用于将字符串数据格式化成指定的格式，其常用的语法格式如下。

```
public static string Format(string format,Object arg0)
public static string Format(string format,params Object[] args)
```

format：用来指定字符串所要格式化的形式，该参数的基本格式如下。

```
{index[,length][:formatString]}
```

- index：要设置格式的对象的参数列表中的位置（从零开始）。
- length：参数的字符串表示形式中包含的最小字符数。 如果该值为正，则参数右对齐；如果该值为负，则参数左对齐。
- formatString：要设置格式的对象支持的标准或自定义格式字符串。

arg0：要设置格式的对象。

arg：一个对象数组，其中包含零个或多个要设置格式的对象。

返回值：格式化后的字符串。

格式化字符串主要有两种情况，分别是数值类型数据的格式化和日期时间类型数据的格式化，下面分别对其进行讲解。

1. 数值类型数据的格式化

在实际开发中，数值类型有多种显示方式，如货币形式、百分比形式等，C# 支持的标准数值格式规范如表 6.1 所示。

表 6.1　C# 支持的标准数值格式规范

格式说明符	名　　称	说　　明	示　　例
C 或 c	货币	结果：货币值 受以下类型支持：所有数值类型 精度说明符：小数位数	¥123 或 -¥123.456
D 或 d	Decimal	结果：整型数字，负号可选 受以下类型支持：仅整型 精度说明符：最小位数	1234 或 -001234
E 或 e	指数（科学型）	结果：指数记数法 受以下类型支持：所有数值类型 精度说明符：小数位数	1.052033E+003 或 -1.05e+003
F 或 f	定点	结果：整数和小数，负号可选 受以下类型支持：所有数值类型 精度说明符：小数位数	1234.57 或 -1234.5600
N 或 n	Number	结果：整数和小数、组分隔符和小数分隔符，负号可选 受以下类型支持：所有数值类型 精度说明符：所需的小数位数	1,234.57 或 -1,234.560
P 或 p	百分比	结果：乘以 100 并显示百分比符号的数字 受以下类型支持：所有数值类型 精度说明符：所需的小数位数	100.00 % 或 100 %
“X”或“x”	十六进制	结果：十六进制字符串 受以下类型支持：仅整型 精度说明符：结果字符串中的位数	FF 或 00ff

学习笔记

在使用 string 类的 Format 方法对数值类型数据进行格式化时，传入的参数必须为数值类型。

示例 3．格式化不同的数值类型数据

使用表 6.1 中的标准数值格式规范对不同的数值类型数据进行格式化，并输出，代码如下。

```
01  static void Main(string[] args)
02  {
03      // 输出金额
04     Console.WriteLine(string.Format("1251+3950 的结果是（以货币形式显示）：{0:C}",
1251 + 3950));
05      // 输出科学计数法
06     Console.WriteLine(string.Format("120000.1 用科学计数法表示：{0:E}", 120000.1));
07      // 输出以分隔符显示的数字
08     Console.WriteLine(string.Format("12800 以分隔符数字显示的结果是：{0:N0}", 12800));
09      // 输出小数点后两位
10     Console.WriteLine(string.Format("π 取两位小数点：{0:F2}", Math.PI));
11      // 输出十六进制
12     Console.WriteLine(string.Format("33 的 16 进制结果是：{0:X4}", 33));
13      // 输出百分号数字
14     Console.WriteLine(string.Format("天才是由 {0:P0} 的灵感,加上 {1:P0} 的汗水 。",
0.01, 0.99));
15      Console.ReadLine();
16  }
```

学习笔记

第 8 行代码中的 {0:N0} 表示显示的数字中不包括小数。

第 10 行代码中的 {0:F2} 表示保留两位小数。

第 12 行代码中的 {0:X4} 表示显示 4 位数形式的十六进制数。

第 14 行代码中的 {0:P0}、{1:P0} 表示显示的百分比中不包括小数。

对数值类型数据进行格式化代码的运行结果如图 6.8 所示。

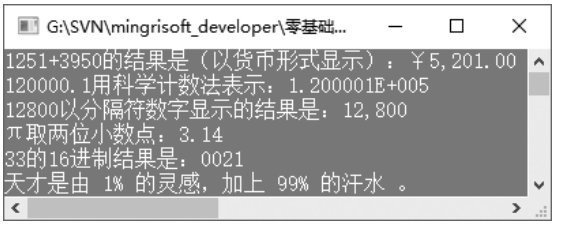


图 6.8　对数值类型数据进行格式化代码的运行结果

2. 日期时间类型数据的格式化

如果希望日期时间按照某种标准格式输出，如短日期时间格式、完整日期时间格式等，那么可以使用 string 类的 Format 方法将日期时间格式化为指定的格式。C# 支持的日期时间类型格式规范如表 6.2 所示。

表 6.2　C# 支持的日期时间类型格式规范

格式说明符	说　明	举　例
d	短日期格式	YYYY-MM-dd
D	长日期格式	YYYY 年 MM 月 dd 日
f	完整日期 / 时间格式（短时间）	YYYY 年 MM 月 dd 日 hh:mm
F	完整日期 / 时间格式（长时间）	YYYY 年 MM 月 dd 日 hh:mm:ss
g	常规日期 / 时间格式（短时间）	YYYY-MM-dd hh:mm
G	常规日期 / 时间格式（长时间）	YYYY-MM-dd hh:mm:ss
M 或 m	月 / 日格式	MM 月 dd 日
t	短时间格式	hh:mm
T	长时间格式	hh:mm:ss
Y 或 y	年 / 月格式	YYYY 年 MM 月

学习笔记

在使用 string 类的 Format 方法对日期时间类型数据进行格式化时，传入的参数必须为 DataTime 类型。

示例 4．输出不同形式的日期时间

使用表 6.2 中的标准日期时间类型格式规范对不同的日期时间数据进行格式化，并输出，代码如下。

```
01  static void Main(string[] args)
02  {
03      DateTime strDate = DateTime.Now;                    // 获取当前日期时间
04      // 输出短日期格式
05      Console.WriteLine(string.Format("当前日期的短日期格式表示：{0:d}", strDate));
06      // 输出长日期格式
07      Console.WriteLine(string.Format("当前日期的长日期格式表示：{0:D}", strDate));
08      Console.WriteLine();                                // 换行
09      // 输出完整日期 / 时间格式（短时间）
10      Console.WriteLine(string.Format("当前日期时间的完整日期 / 时间格式(短时间)表示:
{0:f}", strDate));
11      // 输出完整日期 / 时间格式（长时间）
12      Console.WriteLine(string.Format("当前日期时间的完整日期 / 时间格式（长时间）表示:
{0:F}", strDate));
13      Console.WriteLine();                                // 换行
14      // 输出常规日期 / 时间格式（短时间）
15      Console.WriteLine(string.Format("当前日期时间的常规日期 / 时间格式（短时间）表示:
{0:g}", strDate));
16      // 输出常规日期 / 时间格式（长时间）
```

```
17      Console.WriteLine(string.Format("当前日期时间的常规日期 / 时间格式（长时间）表示：
{0:G}", strDate));
18      Console.WriteLine();                                    // 换行
19      // 输出短时间格式
20      Console.WriteLine(string.Format("当前时间的短时间格式表示：{0:t}", strDate));
21      // 输出长时间格式
22      Console.WriteLine(string.Format("当前时间的长时间格式表示：{0:T}", strDate));
23      Console.WriteLine();                                    // 换行
24      // 输出月 / 日格式
25      Console.WriteLine(string.Format("当前日期的月 / 日格式表示：{0:M}", strDate));
26      // 输出年 / 月格式
27      Console.WriteLine(string.Format("当前日期的年 / 月格式表示：{0:Y}", strDate));
28      Console.ReadLine();
29  }
```

学习笔记

第 3 行代码中获取当前时间时用到了 DateTime 结构，该结构是 .NET Framework 自带的，表示时间上的一刻，通常用日期和当天的时间表示。DateTime.Now 用来获取计算机显示的当前日期和时间。

对日期时间类型数据进行格式化代码的运行结果如图 6.9 所示。

图 6.9　对日期时间类型数据进行格式化代码的运行结果

学习笔记

通过在 ToString 方法中传入指定的“格式说明符”，也可以实现对数值类型数据和日期时间类型数据的格式化。例如，下面的代码分别使用 ToString 方法将数字 1298 格式化为货币形式，当前日期格式化为年 / 月格式，代码如下。

```
01  int money = 1298;
02  Console.WriteLine(money.ToString("C"));// 使用 ToString 方法格式化数值类型数据
03  DateTime dTime = DateTime.Now;
04  Console.WriteLine(dTime.ToString("Y"));// 使用 ToString 方法格式化日期时间类型数据
```

6.4.5 截取字符串

string 类提供了一种 Substring 方法，该方法可以截取字符串中指定位置和指定长度的子字符串。该方法有两种语法格式，分别如下。

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

startIndex：子字符串的起始位置的索引。

length：子字符串中的字符数。

返回值：截取的子字符串。

示例 5．从完整文件名中获取文件名和扩展名

使用 Substring 方法的两种语法格式从一个完整文件名中分别获取文件名称和文件扩展名，代码如下。

```
01 static void Main(string[] args)
02 {
03     string strFile = "Program.cs";                          // 定义字符串
04     Console.WriteLine("文件完整名称：" + strFile);          // 输出文件完整名称
05     string strFileName = strFile.Substring(0, strFile.IndexOf('.'));// 获取文件名
06     string strExtension = strFile.Substring(strFile.IndexOf('.'));// 获取扩展名
07     Console.WriteLine("文件名：" + strFileName);            // 输出文件名
08     Console.WriteLine("扩展名：" + strExtension);           // 输出扩展名
09     Console.ReadLine();
10 }
```

学习笔记

第 5 行代码在设置截取长度时使用了 strFile.IndexOf('.')，表示从 0 截取到“.”的索引位置。

第 6 行代码在获取扩展名时只传入了一个参数，表示从“.”的索引位置开始截取所有的字符。

获取文件名和扩展名代码的运行结果如图 6.10 所示。

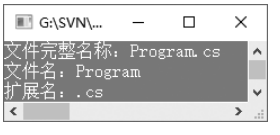


图 6.10　获取文件名和扩展名代码的运行结果

6.4.6　分割字符串

string 类提供了一种 Split 方法用于根据指定的字符数组或字符串数组对字符串进行分割，该方法有 5 种语法格式，分别如下。

```
public string[] Split(params char[] separator)
public string[] Split(char[] separator,int count)
public string[] Split(string[] separator,StringSplitOptions options)
public string[] Split(char[] separator,int count,StringSplitOptions options)
public string[] Split(string[] separator,int count,StringSplitOptions
options)
```

separator：分隔字符串的字符数组或字符串数组。

count：要返回的子字符串的最大数量。

options：如果省略返回的数组中的空数组元素，则为 RemoveEmptyEntries；如果包含返回的数组中的空数组元素，则为 None。

返回值：一个数组，其元素包含分割得到的子字符串，这些子字符串由 separator 中的一个（或多个）字符或字符串分隔。

示例 6．使用 Split 方法分割字符串

有一段体现学习编程最终目标的文字：“让编程学习不再难，让编程创造财富不再难，让编程改变工作和人生不再难”，使用 Split 方法对其进行分割，并输出，代码如下。

```
01 static void Main(string[] args)
02 {
03     // 声明字符串
04     string str = "让编程学习不再难，让编程创造财富不再难，让编程改变工作和人生不再难";
05     char[] separator = { ',' };                    // 声明分割字符的数组
06     // 分割字符串
07       string[] splitStrings = str.Split(separator, StringSplitOptions.
RemoveEmptyEntries);
08     // 使用 for 循环遍历数组，并输出
09     for (int i = 0; i < splitStrings.Length; i++)
10     {
11         Console.WriteLine(splitStrings[i]);
12     }
13     Console.ReadLine();
14 }
```

学习笔记

第 5 行代码用来声明一个字符数组，并初始化一个值。实际上，数组中可以存储多个相同类型的值，这里只存储了一个。

第 9 ~ 12 行代码使用了一个 for 循环遍历字符串数组，并输出数组中的内容，其中，splitStrings.Length 用来获取数组的长度，splitStrings[i] 表示数组中指定索引处的值。

使用 Split 方法分割字符串代码的运行结果如图 6.11 所示。

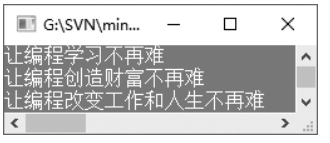


图 6.11　使用 Split 方法分割字符串代码的运行结果

6.4.7　去除空白内容

string 类提供了一种 Trim 方法用来移除字符串中的所有开头空白字符和结尾空白字符，其语法格式如下。

```
public string Trim()
```

Trim 方法的返回值是从当前字符串的开头和结尾删除所有空白字符后剩余的字符串。

例如，定义一个字符串 str，并初始化为“　　abc　　”，然后使用 Trim 方法删除该字符串中开头和结尾处的所有空白字符，代码如下。

```
01  string str = "       abc            ";      // 定义原始字符串
02  string shortStr = str.Trim();                        // 去掉字符串的首、尾空格
03  Console.WriteLine("str 的原值是：[" + str + "]");
04  Console.WriteLine(" 去掉首尾空白的值：[" + shortStr + "]");
```

上面代码的运行结果如下。

```
str 的原值是：[       abc            ]
去掉首尾空白的值：[abc]
```

学习笔记

使用 Trim 方法还可以从字符串的开头和结尾处删除指定的字符，它的使用形式如下。

```
public string Trim(params char[] trimChars)
```

例如，使用 Trim 方法删除字符串开头和结尾处的“*”字符，代码如下。

```
char[] charsToTrim = { '*' };                    // 定义要删除的字符数组
string str = "*****abc*****";                    // 定义原始字符串
string shortStr = str.Trim(charsToTrim);         // 去掉字符串的首、尾空格
```

6.4.8　替换字符串

string 类提供了一种 Replace 方法用于将字符串中的某个字符或字符串替换成其他的字符或字符串，该方法有两种语法格式，分别如下。

```
public string Replace(char OChar,char NChar)
public string Replace(string OValue,string NValue)
```

OChar：待替换的字符。

NChar：替换后的新字符。

OValue：待替换的字符串。

NValue：替换后的新字符串。

返回值：替换字符或字符串之后得到的新字符串。

学习笔记

如果要替换的字符或字符串在原字符串中重复出现，则 Replace 方法会将所有的字符或字符串都进行替换。

示例 7.　“one world,one dream”的变体

创建一个控制台应用程序，声明一个 string 类型的变量 a，并将其初始化为“one world,one dream”。然后使用 Replace 方法的第一种语法格式将字符串中的“,”替换成“*”，最后使用 Replace 方法的第二种语法格式将字符串中的“one”替换成“One”，代码如下。

```
01 static void Main(string[] args)
02 {
03     string strOld = "one world,one dream";      // 声明一个字符串变量并初始化
04     Console.WriteLine(" 原始字符串：" + strOld); // 输出原始字符串
05     // 使用 Replace 方法将字符串中的“,”替换为“*”
06     string strNew1 = strOld.Replace(',', '*');
07     Console.WriteLine("\n 第一种形式的替换：" + strNew1);
08     // 使用 Replace 方法将字符串中的“one”替换为“One”
09     string strNew2 = strOld.Replace("one", "One");
10     Console.WriteLine("\n 第二种形式的替换：" + strNew2);
```

```
11      Console.ReadLine();
12  }
```

学习笔记

第 6 行代码和第 9 行代码分别使用 Replace 方法的两种语法格式替换原字符串中的指定字符和指定子字符串。

字符串的替换代码的运行结果如图 6.12 所示。

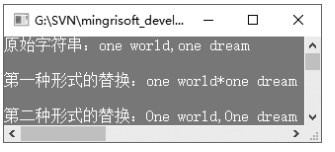


图 6.12　字符串的替换代码的运行结果

学习笔记

要替换的字符或字符串的大小写要与原字符串中字符或字符串的大小写保持一致，否则不能成功替换。例如，将上面的代码修改如下，将不能成功替换。

```
01  string strOld = "one world,one dream";      // 声明一个字符串变量并初始化
02  string strNew2 = strOld.Replace("ONE", "One");  // 对字符串进行替换
```

6.5　可变字符串类

对于创建成功的 string 字符串，它的长度是固定的，其内容不能被改变和编译。虽然使用“+”运算符可以达到附加新字符或新字符串的目的，但“+”运算符会产生一个新的 string 对象，会在内存中创建新的字符串对象。如果重复地对字符串进行修改，将会极大地增加系统开销。而 C# 提供了一个可变的字符序列：StringBuilder 类，大大提高了频繁增加字符串的效率。本节将对其进行讲解。

6.5.1　StringBuilder 类的定义

StringBuilder 类位于 System.Text 命名空间中，如果要创建 StringBuilder 对象，则必须首先引用该命名空间。StringBuilder 类有 6 种不同的构造方法，分别如下。

```
public StringBuilder()
```

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

capacity：StringBuilder 对象的建议起始大小。

value：字符串，包含用于初始化 StringBuilder 对象的子字符串。

maxCapacity：当前字符串可包含的最大字符数。

startIndex：value 中子字符串开始的位置。

length：子字符串中的字符数。

例如，创建一个 StringBuilder 对象，其初始引用的字符串为“Hello World!”，代码如下。

```
StringBuilder MyStringBuilder = new StringBuilder("Hello World!");
```

学习笔记

StringBuilder 类表示值为可变字符序列的类似字符串的对象，之所以说值是可变的，是因为在通过追加、移除、替换或插入字符而创建它后可以对它进行修改。

6.5.2　StringBuilder 类的使用

StringBuilder 类中常用的方法及其说明如表 6.3 所示。

表 6.3　StringBuilder 类中常用的方法及其说明

方　法	说　明
Append	将文本或字符串追加到指定对象的末尾
AppendFormat	自定义变量的格式并将这些值追加到 StringBuilder 对象的末尾
Insert	将字符串或对象添加到当前 StringBuilder 对象中的指定位置
Remove	从当前 StringBuilder 对象中移除指定数量的字符
Replace	用另一个指定的字符来替换 StringBuilder 对象内的字符

学习笔记

StringBuilder 类提供的方法都有多种使用形式，开发者可以根据需要选择合适的使用形式。

示例 8．StringBuilder 类中几种方法的应用

创建一个控制台应用程序，声明一个 int 类型的变量 Num，并初始化为 368，然后创建一个 StringBuilder 对象 SBuilder，其初始值为“明日科技”，之后分别使用 StringBuilder 类的 Append 方法、AppendFormat 方法、Insert 方法、Remove 方法和 Replace 方法对 StringBuilder 对象进行操作，并输出相应的结果。代码如下。

```
01  static void Main(string[] args)
02  {
03      int Num = 368;                  // 声明一个 int 类型的变量 Num 并初始化为 368
04      // 实例化一个 StringBuilder 类，并初始化为“明日科技”
05      StringBuilder SBuilder = new StringBuilder(" 明日科技 ");
06      SBuilder.Append("》C# 编程词典 "); // 使用 Append 方法将字符串追加到 SBuilder 的末尾
07      Console.WriteLine(SBuilder);                  // 输出 SBuilder
08      // 使用 AppendFormat 方法将字符串按照指定的格式追加到 SBuilder 的末尾
09      SBuilder.AppendFormat("{0:C0}", Num);
10      Console.WriteLine(SBuilder);                  // 输出 SBuilder
11      SBuilder.Insert(0, " 软件：");// 使用 Insert 方法将“软件：”追加到 SBuilder 的开头
12      Console.WriteLine(SBuilder);                  // 输出 SBuilder
13      // 使用 Remove 方法从 SBuilder 中删除索引 14 以后的字符串
14      SBuilder.Remove(14, SBuilder.Length - 14);
15      Console.WriteLine(SBuilder);                  // 输出 SBuilder
16      // 使用 Replace 方法将“软件：”替换成“软件工程师必备”
17      SBuilder.Replace(" 软件 ", " 软件工程师必备 ");
18      Console.WriteLine(SBuilder);                  // 输出 SBuilder
19      Console.ReadLine();
20  }
```

学习笔记

第 9 行代码中的 {0:C0} 表示在将数字格式化为货币形式显示时，不显示小数。

使用 StringBuilder 类中几种方法对 StringBuilder 对象进行操作代码的运行结果如图 6.13 所示。

图 6.13　使用 StringBuilder 类中几种方法对 StringBuilder 对象进行操作代码的运行结果

第二篇　进阶篇

第 7 章　面向对象程序设计

面向对象程序设计是在面向过程程序设计的基础上发展而来的，它将数据和对数据的操作看作一个不可分割的整体，力求将现实问题简单化，因为这样不仅符合人们的思维习惯，同时也可以提高软件的开发效率，并方便后期的维护。本章将对面向对象程序设计进行详细讲解。

7.1　面向对象概述

面向对象技术源于面向对象的编程语言（Object Oriented Programming Language，OOPL），从 20 世纪 60 年代提出“面向对象”的概念到现在，它已经发展成为一种比较成熟的程序设计思想，并且逐渐成为目前软件开发领域的主流技术。面向对象（Object Oriented）的英文缩写是 OO，它是一种程序设计思想。

面向对象中的“对象”，通常是指客观世界中存在的对象，这个对象具有唯一性，对象各不相同，各有各的特点，每一个对象都有自己的运动规律和内部状态；对象与对象之间又是可以相互联系、相互作用的。另外，对象也可以是一个抽象的事物。例如，可以从圆形、正方形、三角形等图形中抽象出一个简单图形，一个简单图形就是一个对象，它有自己的属性和行为，图形中边的个数是它的属性，图形的面积也是它的属性，输出图形的面积是它的行为。概括地讲，面向对象技术是一种从组织结构上模拟客观世界的方法。

7.1.1　对象

在现实世界中，随处可见的一种事物就是对象，对象是事物存在的实体，如人类、书桌、计算机、高楼大厦等。人类解决问题的方式总是将复杂的事物简单化，于是就会思考这些对象都是由哪些部分组成的。通常对象分为两个部分，即静态部分与动态部分。静态部分就是不能动的部分，这个部分被称为“属性”，任何对象都具备其自身属性，比如一个人，它包括高 / 矮、胖 / 瘦、性别、年龄等属性；然而具有这些属性的人会执行哪些动作是一

个值得探讨的部分，人可以哭泣、微笑、说话、行走等，这些是人所具备的行为，也就是动态部分。

现实世界中的对象具有以下特征。

（1）每一个对象必须有一个名字以区别其他对象。

（2）用属性来描述对象的某些特征。

（3）有一组操作，每一种操作决定对象的一种行为。

对象的操作可以分为两类：一类是自身所承受的操作，另一类是施加于其他对象的操作。

综上所述，现实世界中的对象可以表示为“属性 + 行为”，其示意图如图 7.1 所示。

在计算机世界中，面向对象程序设计的思想要以对象来思考问题，首先要将现实世界的实体抽象为对象，然后考虑这个对象具备的属性和行为。例如，现在面临这样一个实际问题：一只大雁要从北方飞往南方，尝试以面向对象的思想来解决这一实际问题，步骤如下。

步骤 1，从这一实际问题中抽象出对象，这里抽象出的对象为大雁。

步骤 2，识别这个对象的属性。对象具备的属性都是静态属性，如大雁有一对翅膀、一双脚、一张嘴等，如图 7.2 所示。

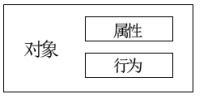


图 7.1　现实世界中的对象表示为“属性 + 行为”示意图

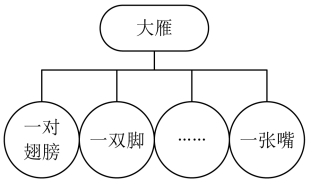


图 7.2　大雁作为对象具备的属性

步骤 3，识别这个对象的动态行为，即这只大雁可以进行的动作，如飞行、觅食等，这些行为都是这个对象基于其属性而具有的动作，如图 7.3 所示。

步骤 4，识别出这个对象的属性和行为后，这个对象就定义完成，然后可以根据这只大雁具有的特性制订这只大雁要从北方飞往南方的具体方案以解决问题。

究其本质，所有的大雁都具有以上的属性和行为，因此可以将这些属性和行为封装起来以描述大雁这类动物。由此可见，类实质上就是封装对象属性和行为的载体，而对象则是类抽象出来的一个实例，描述对象与类之间的关系如图 7.4 所示。

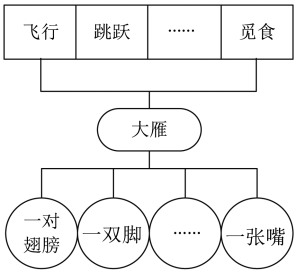


图 7.3　识别对象具有的行为

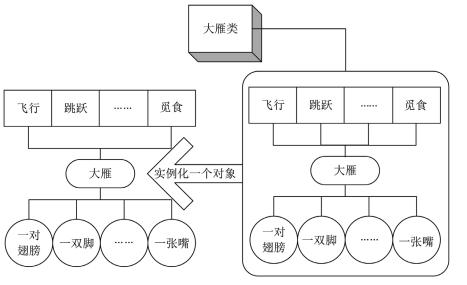


图 7.4　描述对象与类之间的关系

7.1.2　类

不能将一个事物描述成一类事物，一只鸟不能称为鸟类，如果需要对同一类事物进行统称，就需要了解“类”这个概念。

类是同一类事物的统称，如果将现实世界中的一个事物抽象成对象，类就是这类对象的统称，比如鸟类、家禽类、人类等。类是创建对象时所依赖的规范，如一只鸟具有一对翅膀，它可以通过这对翅膀飞行，所有的鸟都具有翅膀这个特性，绝大多数都会飞行，这样的具有相同特性和行为的一类事物就称为类，类的思想就是这样产生的。在图 7.4 中已经描述过类与对象之间的关系，对象就是符合某个类定义所产生出来的实例。而更为恰当的描述是，类是现实世界事物的抽象称谓，而对象则是与事物相对应的实体。如果面临实际问题，通常需要实例化类对象来解决。例如，解决大雁南飞的问题，这里只能以这只大雁为参考来处理这个问题，不能以大雁类为参考来解决。

类是封装对象的属性和行为的载体，反过来说具有相同属性和行为的一类实体被称为类。例如，一种鸟类，鸟类封装了所有鸟的共同属性和应具有的行为，其结构如图 7.5 所示。

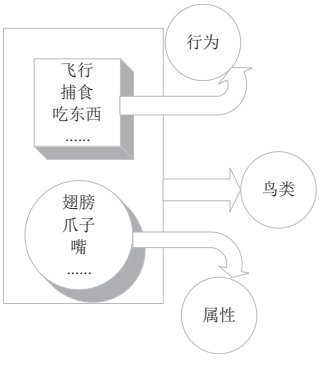


图 7.5　鸟类结构

在类中，对象的行为是以方法的形式定义的，对象的属性是以成员变量的形式定义的，而类包括对象的属性和方法。

学习笔记

一个对象是类的一个实例。

7.1.3 三大基本特征

面向对象程序设计具有三大基本特征：封装、继承和多态，下面分别对其进行描述。

1. 封装

封装是面向对象程序设计的核心思想，将对象的属性和行为封装起来，而将对象的属性和行为封装起来的载体就是类，类通常对客户隐藏其实现细节，这就是封装的思想。例如，用户在使用计算机时，只需要使用手指敲击键盘就可以实现某些功能，而无须知道计算机内部是如何工作的。

采用封装思想保证了类内部数据结构的完整性，使用该类的用户不能直接看到类中的数据结构，而只能执行类允许公开的数据，这样就避免了外部对内部数据的影响，提高了程序的可维护性。

使用类实现封装特性如图 7.6 所示。

2. 继承

矩形、菱形、平行四边形和梯形等都是四边形，因为它们具有共同的特征：拥有 4 条边。只要将四边形进行适当延伸，就会得到上述图形。以平行四边形为例，如果把平行四边形看作四边形的延伸，那么平行四边形就复用了四边形的属性和行为，同时添加了平行四边形特有的属性和行为，如平行四边形的对边平行且相等。在 Java 中，可以把平行四边形类看作继承四边形类后产生的类，其中，将类似于平行四边形的类称为子类，将类似于四边形的类称为父类或超类。值得注意的是，在阐述平行四边形和四边形的关系时，可以说平行四边形是特殊的四边形，但不能说四边形是平行四边形。同理，在 C# 中可以说子类的实例都是父类的实例，但不能说父类的实例是子类的实例。四边形类层次结构示意图如图 7.7 所示。

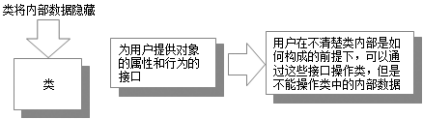


图 7.6 使用类实现封装特性

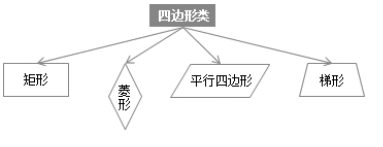


图 7.7 四边形类层次结构示意图

综上所述，继承是实现重复利用的重要手段，子类通过继承在复用了父类的属性和行为的同时又添加了子类特有的属性和行为。

3. 多态

将父类对象应用于子类的特征就是多态。例如，创建一个螺丝类，螺丝类有两个属性，即粗细和螺纹密度；再创建两个类，一个是长螺丝类，另一个是短螺丝类，并且它们都继承了螺丝类。这样长螺丝类和短螺丝类不仅具有相同的特征（粗细相同，且螺纹密度也相同），还具有不同的特征（一个长，一个短，长的可以用来固定大型支架，短的可以用来固定生活中的家具）。综上所述，一个螺丝类衍生出不同的子类，子类在继承父类特征的同时，也具备了自己的特征，并且能够实现不同的效果，这就是多态化的结构。螺丝类层次结构示意图如图 7.8 所示。

图 7.8　螺丝类层次结构示意图

7.2　类

类是一种数据结构，它可以包含数据成员（常量和变量）、函数成员（方法、属性、构造函数和析构函数等）和嵌套类型。

7.2.1　类的声明

在 C# 中，类是使用 class 关键字来声明的，语法如下。

```
class 类名
{
}
```

例如，下面以汽车为例声明一个类，代码如下。

```
01  class Car
```

```
02 {
03 }
```

7.2.2 类的成员

类的定义包括类头和类体两部分，其中，类头是使用 class 关键字定义的类名，而类体是用一对大括号括起来的。在类体中主要定义类的成员，类的成员包括字段、属性、方法、构造函数等。本节将对常用的类成员进行讲解。

1. 字段

字段就是程序开发中常见的常量或变量，它是类的一个构成部分，它使得类可以封装数据。

示例 1. 计算圆面积

创建一个控制台应用程序，在默认的 Program 类中定义一个变量用来存储圆半径；定义一个常量用来存储 π 的值；在 Main 方法中计算圆面积并输出。代码如下。

```
01 class Program
02 {
03     static double r;                          // 定义一个变量，用来存储圆半径
04     const double PI = 3.14;                   // 定义常量，用来存储 π 的值
05     static void Main(string[] args)
06     {
07         Console.Write(" 请输入半径：");
08         Program.r = Convert.ToDouble(Console.ReadLine());     // 输入圆半径
09         Console.WriteLine(" 圆面积为：" + PI * Math.Pow(r, 2));  // 计算圆面积
10         Console.ReadLine();
11     }
12 }
```

学习笔记

第 3 行代码为变量加 static 关键字，说明变量是一个静态变量。

第 9 行代码中的 Math.Pow 方法用来计算某个数的指定次幂。

上面代码中的变量 r 和常量 PI 是 Program 类中的字段。

计算圆面积代码的运行结果如图 7.9 所示。

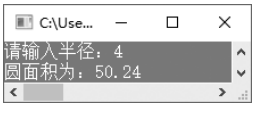


图 7.9 计算圆面积代码的运行结果

学习笔记

字段属于类级别的变量，在未初始化时，C# 将其初始化为默认值，但不会将局部变量初始化为默认值。例如，下面的代码是正确的，输出结果为 0。

```
01  class Program
02  {
03      static int i;
04      static void Main(string[] args)
05      {
06          Console.WriteLine(i);
07      }
08  }
```

但是，如果将变量 i 的定义放在 Main 方法中，在运行时则会出现如图 7.10 所示的错误提示。

图 7.10　局部变量未初始化时出现的错误提示

2. 属性

属性是对现实实体特征的抽象，提供对类或对象的访问。类的属性描述的是状态信息，在类的实例中，属性的值表示对象的状态值。C# 中的属性具有访问器，这些访问器指定在它们的值被读取或写入时需要执行的语句，因此属性提供了一种机制把读取和写入对象的某些特性与一些操作关联起来，开发人员可以像使用公共数据成员一样使用属性。属性的声明语法如下。

```
【权限修饰符】【类型】【属性名】
{
    get   {get 访问器体 }
    set   {set 访问器体 }
}
```

权限修饰符：指定属性的访问级别。

类型：指定属性的类型，可以是任何预定义或自定义类型。

属性名：一种标识符，命名规则与变量相同，但是，属性名的第一个字母通常都需要大写。

get 访问器：相当于一个具有属性类型返回值的无参数方法，它除了作为赋值的目标，当在表达式中引用属性时，还将调用该属性的 get 访问器获取属性的值。get 访问器需要用 return 语句来返回，并且所有的 return 语句都必须返回一个可隐式转换为属性类型的表达式。

set 访问器：相当于一个具有单个属性类型值参数和 void 返回类型的方法。set 访问器的隐式参数始终命名为 value。当一个属性作为赋值的目标被引用时，就会调用 set 访问器，所传递的参数将提供新值。由于 set 访问器存在隐式的参数 value，因此，在 set 访问器中不能自定义名称为 value 的局部变量或常量。

根据是否存在 get 访问器和 set 访问器，属性可以分为以下几种。

（1）可读 / 可写属性：包含 get 访问器和 set 访问器。

（2）只读属性：只包含 get 访问器。

（3）只写属性：只包含 set 访问器。

学习笔记

属性的主要用途是限制外部类对类中成员的访问权限，定义在类级别上。

例如，自定义一个 TradeCode 属性，表示商品编号，要求该属性为可读 / 可写属性，并设置其访问级别为 public，代码如下。

```
01 private string tradecode = "";
02 public string TradeCode
03 {
04     get { return tradecode; }
05     set { tradecode = value; }
06 }
```

由于属性的 set 访问器中可以包含大量的语句，因此可以对赋予的值进行检查，如果值不安全或不符合要求，就可以进行相应处理，这样可以避免为属性设置错误的值而导致异常。

示例 2．通过属性控制年龄的输入范围

创建一个控制台应用程序，在默认的 Program 类中定义一个 Age 属性，设置访问级别为 public。因为 Age 属性提供了 get 访问器和 set 访问器，所以它是可读 / 可写属性；在该属性的 set 访问器中对属性的值进行控制，控制只能输入 1 ～ 70 之间的数据，如果输出其他数据，则会提示相应的信息。代码如下。

```
01 class Program
```

```
02  {
03      private int age;                                    // 定义字段
04      public int Age                                      // 定义属性
05      {
06          get                                             // 设置 get 访问器
07          {
08              return age;
09          }
10          set                                             // 设置 set 访问器
11          {
12              if (value > 0 && value < 70)                // 如果数据合理则将值赋给字段
13              {
14                  age = value;
15              }
16              else
17              {
18                  Console.WriteLine(" 输入数据不合理！ ");
19              }
20          }
21      }
22      static void Main(string[] args)
23      {
24          Program p = new Program();                      // 创建 Program 类的对象
25          while (true)
26          {
27              Console.Write(" 请输入年龄：");
28              p.Age = Convert.ToInt16(Console.ReadLine());// 为年龄属性赋值
29          }
30      }
31  }
```

学习笔记

第 24 行代码用来创建 Program 类的对象，在创建 Program 类的对象时，使用 new 操作符。

通过属性控制年龄的输入范围代码的运行结果如图 7.11 所示。

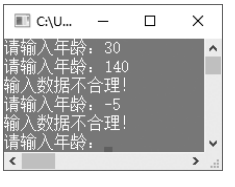


图 7.11　通过属性控制年龄的输入范围代码的运行结果

学习笔记

C# 中支持自动实现的属性，即在属性的 get 访问器和 set 访问器中没有任何逻辑，代码如下。

```
01 public int Age
02 {
03     get;
04     set;
05 }
```

使用自动实现的属性，就不能在属性设置中进行属性的有效验证。例如，在“通过属性控制年龄的输入范围”示例中，不能检查输入的年龄是否在 0 ~ 70 之间；另外，如果要使用自动实现的属性，则必须同时拥有 get 访问器和 set 访问器，只有 get 访问器或只有 set 访问器的代码会出现错误。例如，下面的代码是不合法的。

```
01 public int Age
02 {
03     get;
04 }
```

7.2.3 构造函数

构造函数是一种特殊的函数，它是在创建对象时执行的方法。构造函数具有与类相同的名称，它通常用来初始化对象的数据成员。构造函数的特点如下。

（1）构造函数没有返回值。

（2）构造函数的名称要与本类的名称相同。

1. 构造函数的定义

构造函数的定义语法如下。

```
public class Book
{
    public Book()                   // 无参数构造方法
    {
    }
    public Book(int args)           // 有参数构造方法
    {
        args = 2 + 3;
    }
}
```

public：构造函数修饰符。

Book：构造函数的名称。

args：构造函数的参数。

2. 默认构造函数和有参构造函数

在定义类时，如果没有定义构造函数，则编译器会自动创建一个不带参数的默认构造函数。例如，下面代码定义一个 Book 类。

```
01  class Book
02  {
03  }
```

在创建 Book 类的对象时，可以直接使用如下代码。

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

但是，如果在定义类时，定义了含有参数的构造函数，那么这时如果还想要使用默认构造函数，就需要显式地进行定义了。例如，下面的代码是错误的。

```
01  class Book
02  {
03      public string Name { get; set; }
04      public Book(string name)
05      {
06          Name = name;
07      }
08      void ShowInfo()
09      {
10          Book book = new Book();
11      }
12  }
```

上面代码在运行时，将会出现如图 7.12 所示的错误提示。

图 7.12　使用无参构造函数创建对象时出现的错误提示

上面的错误主要是由于程序中已经定义了一个有参的构造函数，在这种情况下创建对象时，如果想要使用无参构造函数，就必须进行显式定义。

3. 静态构造函数

在 C# 中，可以为类定义静态构造函数，这种构造函数只执行一次。编写静态构造函数的主要原因是类有一些静态字段或属性，需要在第一次使用类之前，从外部源中初始化这些静态字段或属性。

在定义静态构造函数时，不能设置访问修饰符，因为其他 C# 代码从来不会调用它，它只在引用类之前执行一次；另外，静态构造函数不能带任何参数，而且一个类中只能有一个静态构造函数，它只能访问类的静态成员，不能访问实例成员。例如，下面代码用来定义一个静态构造函数。

```
01 static Program()
02 {
03     Console.WriteLine("static");
04 }
```

在类中，静态构造函数和无参数的实例构造函数是可以共存的，因为静态构造函数是在加载类时执行的，而无参数的实例构造函数是在创建类的对象时执行的。

示例 3．静态构造函数和实例构造函数的使用

创建一个控制台应用程序，在 Program 类中定义一个静态构造函数和一个实例构造函数，然后在 Main 方法中创建 3 个 Program 类的对象。代码如下。

```
01 public class Program
02 {
03     static Program()                          // 静态构造函数
04     {
05         Console.WriteLine("static");
06     }
07     private Program()                         // 实例构造函数
08     {
09         Console.WriteLine(" 实例构造函数 ");
10     }
11     static void Main(string[] args)
12     {
13         Program p1 = new Program();           // 创建类的对象 p1
14         Program p2 = new Program();           // 创建类的对象 p2
15         Program p3 = new Program();           // 创建类的对象 p3
16         Console.ReadLine();
17     }
18 }
```

运行上面的代码，得到如图 7.13 所示的内容。

图 7.13　静态构造函数和实例构造函数的使用

从图 7.13 中可以看出，静态构造函数只在引用类之前执行了一次，而实例构造函数每创建一个对象都会执行一次。

7.2.4　析构函数

析构函数主要用来释放对象资源，.NET Framework 类库具有垃圾回收功能，当某个类的实例被认为不再有效，并符合析构条件时，.NET Framework 类库的垃圾回收功能就会调用该类的析构函数实现垃圾回收。析构函数是以类名加～来命名的。例如，为 Program 类定义一个析构函数，代码如下。

```
01  ～Program()                                          //析构函数
02  {
03      Console.WriteLine("析构函数自动调用");
04  }
```

学习笔记

严格来说，析构函数是自动调用的，不需要开发人员显式定义。如果需要定义析构函数，则一个类中只能定义一个析构函数。

构造函数和析构函数是类中比较特殊的两种成员函数，主要用来对对象进行初始化和释放对象资源。一般来说，对象的生命周期从构造函数开始，至析构函数结束。

7.2.5　权限修饰符

C# 中的权限修饰符主要包括 private、protected、internal、protected internal 和 public，这些权限修饰符控制着对类和类的成员变量、成员方法的访问，在表 7.1 中描述了这些权限修饰符的应用范围和访问范围。

表 7.1　C# 中的权限修饰符及其应用范围和访问范围

权限修饰符	应 用 范 围	访 问 范 围
private	所有类或成员	只能在本类中访问
protected	类和内嵌类的所有成员	在本类和其子类中访问
internal	类和内嵌类的所有成员	在同一程序集中访问
protected internal	类和内嵌类的所有成员	在同一程序集和子类中访问
public	所有类或成员	任何代码都可以访问

这里需要注意的是，在定义类时，只能使用 public 或者 internal，这取决于是否希望在包含类的程序集外部访问它。例如，下面的类定义是合法的。

```
01 namespace Demo
02 {
03     public class Program
04     {
05     }
06 }
```

但是，不能把类定义为 private、protected 或 protected internal 类型，因为这些权限修饰符对于包含在命名空间中的类是没有意义的，只能应用于成员；但是，可以使用这些权限修饰符定义内嵌类（包含在其他类中的类），因为在这种情况下，类也具有成员的状态。例如，下面的代码是合法的。

```
01 namespace Demo
02 {
03     public class Program
04     {
05         private class Test
06         {
07         }
08     }
09 }
```

学习笔记

如果有内嵌类型，那么内嵌类型总是可以访问外部类型的所有成员的，因此，上面代码中的 Test 类可以访问 Program 类的所有成员，包括其 private 成员。

7.3　方法

方法用来定义类可执行的操作，它是包含一系列语句的代码块。从本质上来讲，方法就是和类相关联的动作。

7.3.1　方法的声明

方法在类或结构中声明，声明时需要指定访问级别、返回值、方法名称及方法参数。方法参数放在括号中，并用逗号隔开。如果括号中没有内容，则表示声明的方法没有参数。

声明方法的基本格式如下。

```
修饰符  返回值类型  方法名（参数列表）
{
// 方法的具体实现；
}
```

其中，“修饰符”可以是 private、public、protected、internal 中的任何一个；“返回值类型”指定方法返回数据的类型（可以是任何类型），如果方法不需要返回一个值，则使用 void 关键字；“参数列表”是用逗号分隔的类型、标识符，如果方法中没有参数，则“参数列表”为空。

一个方法的名称和参数列表定义了该方法的签名，具体来讲，一个方法的签名由它的名称及参数的个数、修饰符和类型组成。返回值类型不是方法签名的组成部分，参数的名称也不是方法签名的组成部分。

例如，定义一个 ShowGoods 方法，用来输出库存商品信息，代码如下。

```
01  public void ShowGoods()
02  {
03      Console.WriteLine("库存商品名称：");
04      Console.WriteLine(FullName);
05  }
```

学习笔记

方法的定义必须在某个类中，定义方法时如果没有声明访问修饰符，则方法的默认访问权限为 private。

如果定义的方法有返回值，则必须使用 return 关键字返回一个指定类型的数据。例如，

定义一个返回值类型为 int 的方法，就必须使用 return 关键字返回一个 int 类型的值，代码如下。

```
01 public int ShowGoods()
02 {
03     Console.WriteLine("商品信息");
04     return 1;
05 }
```

在上面的代码中，如果将“return 1;”删除，则将会出现如图 7.14 所示的错误提示。

图 7.14　方法无返回值时出现的错误提示

7.3.2　方法的参数

在调用方法时可以给该方法传递一个或多个值，传给方法的值称为实参；在方法内部，接收实参的变量称为形参。形参在紧跟着方法名的括号中声明，形参的声明语法与变量的声明语法一样。形参只在方法内部有效。C# 中的方法的参数主要有 4 种，分别为值参数、ref 参数、out 参数和 params 参数，下面分别对其进行讲解。

1. 值参数

值参数是在声明时不加修饰的参数，它表明实参与形参之间按值传递。当使用值参数的方法被调用时，编译器为形参分配存储单元，然后将对应的实参复制到形参中。由于是值类型的传递方式，所以在方法中对值类型的形参的修改并不会影响实参。

2. ref 参数

ref 参数使形参按引用传递（即使形参为值类型），其效果是，在方法中对形参所做的任何更改都将反映在实参中。如果要使用 ref 参数，则方法声明和方法调用都必须显式使用 ref 关键字。

3. out 参数

out 关键字用来定义输出参数，它会导致参数通过引用来传递，这与 ref 关键字类似，不同之处在于，使用 ref 关键字要求变量必须在传递之前进行赋值，而使用 out 关键字定

义的参数不用进行赋值即可使用。如果要使用 out 参数，则方法声明和方法调用都必须显式使用 out 关键字。

4. params 参数

在声明方法时，如果有多个相同类型的参数，则可以定义为 params 参数。params 参数是一个一维数组，主要用来指定在参数数目可变时所采用的方法参数。

示例 4. 不同类型参数方法的使用

创建一个控制台应用程序，定义 4 个 Add 方法用来计算整型类型数据的和。在这 4 个方法中分别使用值参数、ref 参数、out 参数和 params 参数，然后在 Main 方法中调用这 4 个方法，比较它们的运算结果。代码如下。

```
01 class Program
02 {
03     private int Add(int x, int y)                    // 值参数
04     {
05         x = x + y;                                   // 对 x 进行加 y 操作
06         return x;                                    // 返回 x
07     }
08     private int Add(ref int x, int y)                //ref 参数
09     {
10         x = x + y;                                   // 对 x 进行加 y 操作
11         return x;                                    // 返回 x
12     }
13     private int Add(int x, int y, out int z)         //out 参数
14     {
15         z = x + y;                                   // 记录 x+y 的结果
16         return z;                                    // 返回 out 参数 z
17     }
18     private int Add(params int[] x)                  //params 参数
19     {
20         int result = 0;                              // 记录运算结果
21         for (int i = 0; i < x.Length; i++)           // 遍历参数数组
22         {
23             result += x[i];                          // 执行相加操作
24         }
25         return result;                               // 返回运算结果
26     }
27     static void Main(string[] args)
28     {
29         Program pro = new Program();                 // 创建 Program 对象
30         int x = 30;                                  // 定义实参变量 x
31         int y = 40;                                  // 定义实参变量 y
32         Console.WriteLine("值参数的使用: " + pro.Add(x, y));
```

```
33          Console.WriteLine(" 值参数中实参 x 的值：" + x);// 输出值参数方法中实参 x 的值
34          Console.WriteLine("ref 参数的使用：" + pro.Add(ref x, y));
35          // 输出 ref 参数方法中实参 x 的值
36          Console.WriteLine("ref 参数中实参 x 的值：" + x);
37          Console.WriteLine("out 参数的使用：" + pro.Add(x,y,out int z));
38          Console.WriteLine("params 参数的使用：" + pro.Add(20, 30, 40, 50, 60));
39          Console.ReadLine();
40      }
41  }
```

按 <Ctrl+F5> 键查看代码运行结果，如图 7.15 所示。

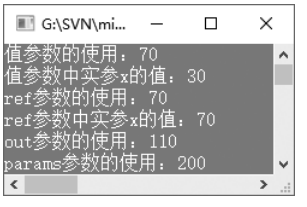


图 7.15　不同类型参数方法的使用代码的运行结果

仔细观察图 7.15，发现第 2 行和第 4 行中的实参 x 的值不一样，第 2 行中的实参 x 的值并没有因为值参数方法的计算结果而改变，但是由于接下来调用了 ref 参数的方法，所以在输出第 4 行内容时，实参 x 的值随之发生了更改。

7.3.3　方法的重载

方法的重载是指方法名相同，但参数的数据类型、个数或顺序不同的方法。只要类中有两个以上的同名方法，但是使用的参数类型、个数或顺序不同，在调用时，编译器就可判断在哪种情况下调用哪种方法。

示例 5．加法的不同运算形式

创建一个控制台应用程序，定义一个 Add 方法，该方法有 3 种重载形式，分别用来计算两个 int 类型的数据的和、计算一个 int 类型的数据和一个 double 类型的数据的和、计算 3 个 int 类型的数据的和；在 Main 方法中分别调用 Add 方法的 3 种重载形式，并输出计算结果。代码如下。

```
01  class Program
02  {
03      // 定义 Add 方法，返回值为 int 类型，有两个 int 类型的参数
04      public static int Add(int x, int y)
05      {
06          return x + y;
07      }
```

```
08      public double Add(int x, double y)// 重载 Add 方法，它与第一个的参数类型不同
09      {
10          return x + y;
11      }
12      public int Add(int x, int y, int z)// 重载 Add 方法，它与第一个的参数个数不同
13      {
14          return x + y + z;
15      }
16      static void Main(string[] args)
17      {
18          Program program = new Program();    // 创建类对象
19          int x = 3;
20          int y = 5;
21          int z = 7;
22          double y2 = 5.5;
23          // 根据传入的参数类型及参数个数的不同调用不同的 Add 方法重载形式
24          Console.WriteLine(" 第 1 种重载形式: " + x + "+" + y + "=" + Program.
Add(x, y));
25          Console.WriteLine(" 第 2 种重载形式: " + x + "+" + y2 + "=" + program.
Add(x, y2));
26          Console.WriteLine(" 第 3 种重载形式: " + x + "+" + y + "+" + z + "="
+ program.Add(x, y, z));
27          Console.ReadLine();
28      }
29  }
```

运行上面的代码，得到如图 7.16 所示的内容。

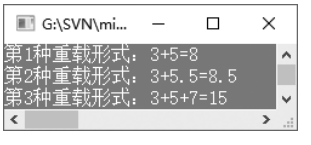


图 7.16　重载方法的应用

在定义重载方法时，需要注意以下两点。

（1）重载方法不能仅在返回值类型上不同，因为返回值类型不是方法签名的一部分。

（2）重载方法不能仅根据参数是否声明为 ref、out 或 params 来区分。

7.4　类的静态成员

很多时候，不同的类之间需要对同一个变量进行操作。比如一个水池，同时打开进水口和出水口，进水和出水这两个动作会同时影响水池中的水量，此时水池中的水量就可以

认为是一个共享的变量。在 C# 程序中，共享的变量或方法用 static 进行修饰，它们被称作静态变量和静态方法，也被称为类的静态成员。静态成员是属于类所有的，在调用静态成员时，不用创建类的对象，可以直接使用类名调用。

示例 6．使用静态方法计算两个数的和

创建一个控制台应用程序，在 Program 类中定义一个静态方法 Add，实现两个整型数相加，然后在 Main 方法中直接使用类名调用静态方法，代码如下。

```
01 class Program
02 {
03     public static int Add(int x, int y)     // 定义静态方法实现整型数相加
04     {
05         return x + y;
06     }
07     static void Main(string[] args)
08     {
09         // 使用类名调用静态方法
10         Console.WriteLine("{0}+{1}={2}", 23, 34, Program.Add(23, 34));
11         Console.ReadLine();
12     }
13 }
```

上面代码的运行结果如下。

```
23+34=57
```

如果在声明类时使用了 static 关键字，则该类就是一个静态类。静态类中定义的成员必须是静态的，在静态类中不能定义实例变量、实例方法或实例构造函数。例如，下面的代码是错误的。

```
01 static class Test
02 {
03     public Test()                    // 错误，因为静态类中不能定义非静态成员
04     {
05     }
06 }
```

7.5 对象的创建及使用

C# 是面向对象的程序设计语言，所有的问题都通过对象来处理，对象可以通过操作类的属性和方法解决相应的问题，所以了解对象的产生、操作和销毁对学习 C# 是十分有必要的。本节将讲解对象在 C# 中的应用。

7.5.1　对象的创建

对象可以认为是在一类事物中抽象出的某一个特例，通过这个特例来处理这类事物出现的问题。在 C# 中通过 new 操作符来创建对象。前文在讲解构造函数时介绍过每实例化一个对象就会自动调用一次构造函数，实质上这个过程就是创建对象的过程。准确地说，可以在 C# 中使用 new 操作符调用构造函数来创建对象。

创建对象语法如下。

```
Test test=new Test();
Test test=new Test("a");
```

创建对象语法中的参数说明如表 7.2 所示。

表 7.2　创建对象语法中的参数说明

参　数	说　明
Test	类名
test	创建 Test 类对象
new	创建对象操作符
"a"	构造函数的参数

当用户使用 new 操作符创建一个对象后，可以使用“对象 . 类成员”来获取对象的属性和行为。对象的属性和行为在类中是通过类成员变量和成员方法的形式来表示的，所以当对象获取类成员时，也就相应地获取了对象的属性和行为。

示例 7．输出库存的商品名称

创建一个控制台应用程序，在程序中创建一个 cStockInfo 类用来表示库存商品类，在该类中定义一个 FullName 属性和一个 ShowGoods 方法；然后在 Program 类中创建 cStockInfo 类的对象，并使用该对象调用 Program 类中的属性和方法。代码如下。

```
01 public class cStockInfo                          // 自定义库存商品类
02 {
03     public string FullName                       // 自动实现属性
04     {
05         get;
06         set;
07     }
08     public void ShowGoods()                      // 定义一个无返回值类型的方法
09     {
10         Console.WriteLine("库存商品名称：");
```

```
11          Console.WriteLine(FullName);          // 输出属性值
12      }
13  }
14  class Program
15  {
16      static void Main(string[] args)
17      {
18          cStockInfo stockInfo = new cStockInfo(); // 创建 cStockInfo 对象
19          stockInfo.FullName = "笔记本电脑";   // 使用对象调用类成员属性
20          stockInfo.ShowGoods();               // 使用对象调用类成员方法
21          Console.ReadLine();
22      }
23  }
```

运行上面代码，得到如图 7.17 所示的内容。

图 7.17　使用对象调用类成员

学习笔记

C# 中提供了一个 this 关键字，它表示本类的一个对象，在局部变量或方法参数覆盖了成员变量时，可以使用 this 关键字明确引用的是类成员还是方法的形参。另外，this 关键字除了可以调用成员变量或成员方法，还可以作为方法的返回值，用来返回本类的对象。

7.5.2　对象的销毁

每个对象都有生命周期，当对象的生命周期结束时，分配给该对象的内存地址将会被回收。在其他语言中需要手动回收废弃的对象，但是 C# 拥有一套完整的垃圾回收机制，用户不必担心废弃的对象占用内存，垃圾回收器将回收无用的但占用内存的资源。

在介绍垃圾回收机制之前，首先需要了解何种对象会被 .NET 垃圾回收器视为垃圾，主要包括以下两种情况。

（1）对象引用超过其作用范围，则这个对象将被视为垃圾，如图 7.18 所示。

（2）将对象赋值为 null，如图 7.19 所示。

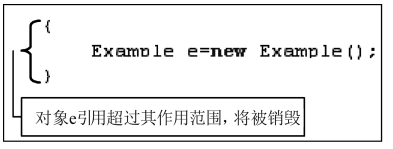


图 7.18　对象引用超过其作用范围将被销毁

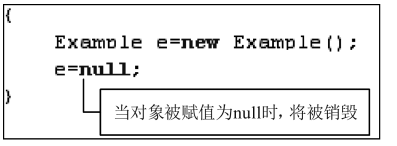


图 7.19　当对象被赋值为 null 时将被销毁

7.5.3　类与对象的关系

类是一种抽象的数据类型，但是其抽象的程度可能不同，而对象是一个类的实例。例如，将农民看作一个类，张三和李四就可以各为一个对象。

从这里可以看出，张三和李四有很多共同点，他们都在某个农村生活，早上都要出门务农，晚上都要回家。对于这样相似的对象，就可以将其抽象为一个数据类型，此处抽象为农民。这样，只要将“农民”这个类型编写好，在程序中就可以方便地创建张三和李四这样的对象。在代码需要进行更改时，只需要对农民类型进行修改即可。

综上所述，可以看出类与对象的区别：类是具有相同（或相似）结构、操作和约束规则的对象组成的集合，而对象是某一类的具体化实例，每个类都是具有某些共同特征的对象的抽象。

7.6　继承

继承是面向对象程序设计重要的特性之一，它源于人们认识客观世界的过程，是自然界普遍存在的一种现象。例如，我们每个人都从父母那里继承了一些体貌特征，但是每个人却又不同于父母，因为每个人都存在自己的一些特性，这些特性是独有的，在父母身上并没有体现。在程序设计中实现继承，表示这个类拥有它继承的类的所有公有成员或受保护成员。在面向对象程序设计中，被继承的类称为父类或基类，实现继承的类称为子类或派生类。

7.6.1　继承的实现

继承的基本思想是基于某个基类扩展出一个新的派生类，派生类可以继承基类原有的属性和方法，也可以增加原来基类所不具备的属性和方法，或者直接重写基类中的某些方法。例如，平行四边形是特殊的四边形，可以说平行四边形类继承了四边形类，这时平行四边形类将所有四边形类具有的属性和方法都保留下来，并基于四边形类扩展出一些平行

四边形类特有的属性和方法。

下面演示一下继承性。创建一个新类 Test，同时创建另一个新类 Test2 继承 Test 类，其中包括重写的基类成员方法及新增成员方法等。在图 7.20 中描述了 Test 类与 Test2 类的结构，以及两者之间的关系。

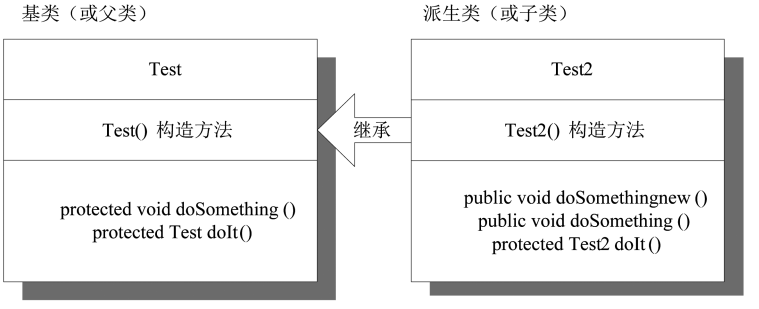


图 7.20　Test 类与 Test2 类之间的继承关系

在 C# 中使用“:”来标识两个类的继承关系。在继承一个类时，类成员的可访问性是一个重要的问题。派生类（或子类）不能访问基类（或父类）的私有成员，但是可以访问其公共成员，这就是说，只要使用 public 声明类成员，就可以让一个类成员被基类（或父类）和派生类（或子类）同时访问，同时也可以被外部的代码访问。

另外，为了解决基类成员的访问问题，C# 还提供了另外一种可访问性：protected，它表示受保护成员，只有基类（或父类）和派生类（或子类）才能访问 protected 成员，外部代码不能访问 protected 成员。

学习笔记

派生类不能继承基类中定义的 private 成员。

示例 8．模拟实现输出进、销、存管理系统的进货信息

创建一个控制台应用程序，模拟实现输出进、销、存管理系统的进货信息。自定义一个 Goods 类，在该类中定义两个公有属性，表示商品编号和名称；自定义 JHInfo 类，继承自 Goods 类，在该类中定义进货编号属性，以及输出进货信息的方法；最后在 Pragram 类的 Main 方法中创建子类 JHInfo 的对象，并使用该对象调用父类 Goods 中定义的公有属性。代码如下。

```
01  class Goods
02  {
03      public string TradeCode { get; set; }                    // 定义商品编号
04      public string FullName { get; set; }                     // 定义商品名称
```

```
05 }
06 class JHInfo : Goods                          // 继承 Goods 类
07 {
08     public string JHID { get; set; }          // 定义进货编号
09     public void showInfo()                    // 输出进货信息
10     {
11         Console.WriteLine(" 进货编号：{0}\n 商品编号：{1}\n 商品名称：{2}", JHID,
TradeCode, FullName);
12     }
13 }
14 class Program
15 {
16     static void Main(string[] args)
17     {
18         JHInfo jh = new JHInfo();             // 创建 JHInfo 对象
19         jh.TradeCode = "T100001";             // 设置父类中的 TradeCode 属性
20         jh.FullName = " 笔记本电脑 ";          // 设置父类中的 FullName 属性
21         jh.JHID = "JH00001";                  // 设置 JHID 属性
22         jh.showInfo();                        // 输出信息
23         Console.ReadLine();
24     }
25 }
```

运行上面的代码，得到如图 7.21 所示的内容。

图 7.21　输出进货信息

学习笔记

C# 只支持单继承，而不支持多重继承，即在 C# 中一次只允许继承一个类，不能同时继承多个类。例如，下面的代码是错误的。

```
01 class Goods
02 {
03 }
04 class JHInfo : Goods                          // 正确：继承单个类
05 {
06 }
07 class Program : Goods, JHInfo                // 错误：继承多个类
08 {
09 }
```

上面的代码在 Visual Studio 2017 中运行时将会出现如图 7.22 所示的错误提示。

图 7.22　继承多个类时出现的错误提示

学习笔记

在实现类的继承时，子类的可访问性一定要低于或等于父类的可访问性。例如，下面的代码是错误的。

```
01  class Goods
02  {
03  }
04  public class JHInfo : Goods
05  {
06  }
```

因为父类 Goods 在声明时没有指定访问修饰符，其默认访问级别为 private，而子类 JHInfo 的可访问性 public 要高于父类 Goods 的可访问性，所以会出现错误，如图 7.23 所示。

图 7.23　子类可访问性高于父类时出现的错误提示

7.6.2　base 关键字

如果子类重写了父类的方法，就无法调用父类的方法了吗？如果想在子类的方法中实现父类原有的方法怎么办？为了解决这些问题，C# 提供了 base 关键字。

base 关键字的使用方法与 this 关键字类似，this 关键字代表本类对象，base 关键字代表父类对象，使用方法如下。

```
base.property;        // 调用父类的属性
base.method();        // 调用父类的方法
```

学习笔记

如果要在子类中使用 base 关键字调用父类的属性或方法，则父类的属性和方法必须定义为 public 类型或 protected 类型。

示例 9．模拟平板电脑和电脑的关系

创建一个 Computer 类用来作为父类，再创建一个 Pad 类，继承自 Computer 类，重写父类方法，并使用 base 关键字调用父类方法原有的逻辑，代码如下。

```
01 class Computer                                          // 父类：电脑
02 {
03     public string sayHello()
04     {
05         return "欢迎使用";
06     }
07 }
08 class Pad : Computer                                    // 子类：平板电脑
09 {
10     public new string sayHello()                        // 子类重写父类方法
11     {
12         return base.sayHello() + "平板电脑";// 子类调用父类方法，在结果后添加字符串
13     }
14 }
15 class Program
16 {
17     static void Main(string[] args)
18     {
19         Computer pc = new Computer();                   // 电脑类
20         Console.WriteLine(pc.sayHello());
21         Pad ipad = new Pad();                           // 平板电脑类
22         Console.WriteLine(ipad.sayHello());
23         Console.ReadLine();
24     }
25 }
```

学习笔记

第 10 行代码在子类中定义 sayHello 方法时，使用了一个 new 关键字，这是因为子类中的 sayHello 方法与父类中的 sayHello 方法同名，而且返回值与参数完全相同。这时，在父类中调用 sayHello 方法时会产生歧义，所以加了 new 关键字来隐藏父类的 sayHello 方法。

运行上面代码，得到如图 7.24 所示的内容。

图 7.24　使用 base 关键字访问父类成员

另外，使用 base 关键字还可以指定创建子类实例时应调用的父类构造函数。例如，修改 7.6.1 节中“模拟实现输出进、销、存管理系统的进货信息”示例中的代码内容，在父类 Goods 中定义一个构造函数，用来为定义的属性赋初始值，代码如下。

```
01 public Goods(string tradecode, string fullname)
02 {
03     TradeCode = tradecode;
04     FullName = fullname;
05 }
```

在子类 JHInfo 中定义构造函数时，即可使用 base 关键字调用父类的构造函数，代码如下。

```
01 public JHInfo(string jhid, string tradecode, string fullname) :
base(tradecode, fullname)
02 {
03     JHID = jhid;
04 }
```

学习笔记

访问父类成员只能在构造函数、实例方法或实例属性中进行，因此，在静态方法中使用 base 关键字是错误的。

7.6.3　继承中的构造函数与析构函数

在进行类的继承时，派生类的构造函数会隐式调用基类的无参构造函数，但是，如果基类也是从其他类派生的，则 C# 会根据层次结构找到顶层的基类，并调用基类的构造函数，然后再依次调用各级派生类的构造函数。析构函数的执行顺序正好与构造函数相反。继承中的构造函数和析构函数执行顺序示意图如图 7.25 所示。

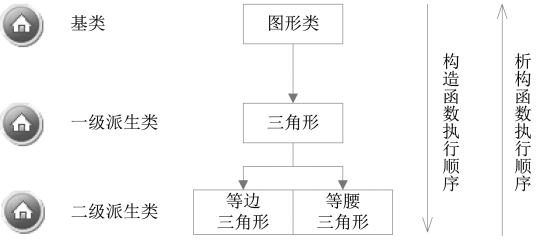


图 7.25　继承中的构造函数和析构函数执行顺序示意图

7.7　多态

多态是面向对象程序设计的基本特征之一，它使得派生类的实例可以直接赋予基类的对象，然后直接通过这个对象调用派生类的方法。在 C# 中，类的多态性是通过在派生类中重写基类的虚方法来实现的。

7.7.1　虚方法的重写

在 C# 中，方法在默认情况下不是虚拟的，但（除构造函数外）可以显式地声明为 virtual，在方法前面加上 virtual 关键字，则称该方法为虚方法。例如，下面代码声明一个虚方法。

```
01  public virtual void Move()
02  {
03      Console.WriteLine(" 交通工具都可以移动 ");
04  }
```

将方法定义为虚方法后，可以在派生类中重写虚方法。重写虚方法需要使用 override 关键字，这样在调用方法时，可以调用对象类型的相应方法。例如，使用 override 关键字重写上面的虚方法。

```
01  public override void Move()
02  {
03      Console.WriteLine(" 火车都可以移动 ");
04  }
```

学习笔记

类中的成员字段和静态方法不能声明为 virtual，因为 virtual 只对类中的实例函数和属性有意义。

示例 10．从交通工具衍生出火车和汽车的不同形态

创建一个控制台应用程序，其中自定义一个 Vehicle 类用来作为基类，在该类中自定义一个虚方法 Move；自定义 Train 类和 Car 类，都继承自 Vehicle 类，在这两个派生类中重写基类中的虚方法 Move，输出不同交通工具的形态；在 Pragram 类的 Main 方法中，分别使用基类和派生类的对象生成一个 Vehicle 类型的数组，使用数组中的每个对象调用 Move 方法，比较它们的输出信息。代码如下。

```
01 class Vehicle
02 {
03     string name;                              //定义字段
04     public string Name                        //定义属性为字段赋值
05     {
06         get { return name; }
07         set { name = value; }
08     }
09     public virtual void Move()                //定义方法输出交通工具的形态
10     {
11         Console.WriteLine("{0}都可以移动", Name);
12     }
13 }
14 class Train : Vehicle
15 {
16     public override void Move()               //重写方法输出交通工具的形态
17     {
18         Console.WriteLine("{0}在铁轨上行驶", Name);
19     }
20 }
21 class Car : Vehicle
22 {
23     public override void Move()               //重写方法输出交通工具的形态
24     {
25         Console.WriteLine("{0}在公路上行驶", Name);
26     }
27 }
28 class Program
29 {
30     static void Main(string[] args)
31     {
32         Vehicle vehicle = new Vehicle();      //创建 Vehicle 类的实例
33         Train train = new Train();            //创建 Train 类的实例
34         Car car = new Car();                  //创建 Car 类的实例
35         //使用基类和派生类对象创建 Vehicle 类型的数组
36         Vehicle[] vehicles = { vehicle, train, car };
37         vehicle.Name = "交通工具";             //设置交通工具的名字
```

```
38          train.Name = "火车";                    //设置交通工具的名字
39          car.Name = "汽车";                      //设置交通工具的名字
40          vehicles[0].Move();                     //输出交通工具的形态
41          vehicles[1].Move();                     //输出交通工具的形态
42          vehicles[2].Move();                     //输出交通工具的形态
43          Console.ReadLine();
44      }
45  }
```

学习笔记

第 36 行代码用来自定义一个 Vehicle 类型的数组，该数组中的元素类型不同，但是都可以向上转型为父类对象。向上转型即可以将子类对象向上转换为父类对象。例如，可以说火车是交通工具，但不能说交通工具是火车。

运行上面代码，得到如图 7.26 所示的内容。

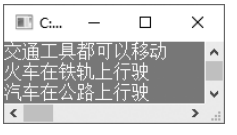


图 7.26　交通工具的形态

7.7.2　抽象类与抽象方法

如果一个类不与具体的事物相联系，而只是表达一种抽象的概念或行为，仅作为其派生类的一个基类，那么这样的类就可以声明为抽象类。例如，去商场买衣服，这句话描述的就是一个抽象的行为。到底去哪个商场买衣服，买什么样的衣服，是短衫、裙子，还是其他类型的衣服？在“去商场买衣服”这句话中，并没有对“买衣服”这个抽象行为指定一个明确的信息。如果要将“去商场买衣服”这个动作封装为一个行为类，那么这个类就应该是一个抽象类。

在 C# 中声明抽象类时需要使用 abstract 关键字，具体语法格式如下。

```
访问修饰符 abstract class 类名 : 基类或接口
{
    // 类成员
}
```

学习笔记

在声明抽象类时，除 abstract 关键字、class 关键字和类名外，其他都是可选项。

抽象类主要用来提供多个派生类可共享的基类的公共定义，它与非抽象类的主要区别如下。

（1）抽象类不能直接实例化。

（2）抽象类中可以包含抽象成员，但非抽象类不可以。

（3）抽象类不能被密封。

学习笔记

由于抽象类本身不能直接实例化，因此很多人认为在抽象类中声明构造函数是没有意义的，其实不然，即使我们不为抽象类声明构造函数，编译器也会自动为其生成一个默认的构造函数。抽象类中的构造函数主要有两个作用，即初始化抽象类的成员和为由它派生出的派生类所使用，因为派生类在实例化时，如果是无参实例化，则首先调用基类（包括抽象类）的无参构造函数，然后再调用派生类自身的无参构造函数；如果是有参实例化，则首先调用基类（包括抽象类）的有参构造函数，然后再调用派生类自身的有参构造函数。

在抽象类中定义的方法，如果加上 abstract 关键字，就是一个抽象方法，抽象方法不提供具体的实现。引入抽象方法的原因在于抽象类本身是一个抽象的概念，有的方法并不需要具体实现，而是留下让派生类来重写实现。在声明抽象方法时需要注意以下两点。

（1）抽象方法必须声明在抽象类中。

（2）在声明抽象方法时，不能使用 virtual、static 和 private 3 个修饰符。

例如，声明一个抽象类，在该抽象类中声明一个抽象方法，代码如下。

```
01 public abstract class TestClass
02 {
03     public abstract void AbsMethod();       // 抽象方法
04 }
```

学习笔记

在 C# 中规定，类中只要有一个方法声明为抽象方法，则这个类也必须被声明为抽象类。

当从抽象类派生一个非抽象类时，需要在非抽象类中重写抽象方法，以提供具体的实现。在重写抽象方法时需要使用 override 关键字。

示例 11．模拟去商场买衣服的场景

使用抽象类模拟去商场买衣服的场景，然后通过派生类确定到底去哪个商场买衣服，

买什么样的衣服，代码如下。

```
01 public abstract class Market
02 {
03     public string Name { get; set; }          //商场名称
04     public string Goods { get; set; }         //商品名称
05     public abstract void Shop();              //抽象方法，用来输出信息
06 }
07 public class WallMarket : Market             //继承抽象类
08 {
09     public override void Shop()               //重写抽象方法
10     {
11         Console.WriteLine(Name + "购买" + Goods);
12     }
13 }
14 public class TaobaoMarket : Market           //继承抽象类
15 {
16     public override void Shop()               //重写抽象方法
17     {
18         Console.WriteLine(Name + "购买" + Goods);
19     }
20 }
21 class Program
22 {
23     static void Main(string[] args)
24     {
25         Market market = new WallMarket();    //使用派生类对象创建抽象类对象
26         market.Name = "沃尔玛";
27         market.Goods = "七匹狼西服";
28         market.Shop();
29         market = new TaobaoMarket();          //使用派生类对象创建抽象类对象
30         market.Name = "淘宝";
31         market.Goods = "韩都衣舍花裙";
32         market.Shop();
33         Console.ReadLine();
34     }
35 }
```

学习笔记

第 25 行代码和第 29 行代码分别使用派生类对象创建了抽象类的一个对象，这样，使用该对象即可调用在抽象类中定义的成员。但是，如果派生类中有单独定义的成员，那么使用该对象是无法进行访问的。

运行上面代码，得到如图 7.27 所示的内容。

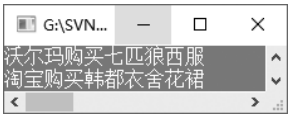


图 7.27　使用抽象类模拟去商场买衣服的场景

7.7.3　接口的使用

由于 C# 中的类不支持多重继承，但是客观世界出现多重继承的情况又比较多，为了避免传统的多重继承给程序带来的复杂性等问题，同时保证多重继承带给程序员的诸多好处，在 C# 中引出了“接口”的概念，通过接口可以实现多重继承的功能。

接口提出了一种契约，或者称为规范，让使用接口的程序设计人员必须严格遵守接口提出的约定。例如，在组装电脑时，主板与机箱之间就存在一种事先约定，不管什么型号或品牌的机箱，什么种类或品牌的主板，都必须遵照一定的标准来进行设计制造。因此，在组装电脑时，电脑的零配件都可以安装在大多数机箱上。接口就可以看作这种标准，它强制性地要求派生类必须实现接口约定的规范，以保证派生类必须拥有某些特性。

在 C# 中声明接口时，需要使用 interface 关键字，其语法格式如下。

```
修饰符 interface 接口名称 ：继承的接口列表
{
    接口内容；
}
```

学习笔记

接口可以继承其他接口，类可以通过其继承的基类（或接口）多次继承同一个基类。接口具有以下特征。

- 接口类似于抽象基类，继承接口的任何类型都必须实现接口的所有成员。
- 接口中不能包括构造函数，因此不能直接实例化接口。
- 接口可以包含属性、方法、索引器和事件。
- 接口中只能定义成员，不能实现成员。
- 接口中定义的成员不允许加访问修饰符，因为接口成员永远是公共的。
- 接口中的成员不能声明为虚拟或静态。

例如，使用 interface 关键字定义一个 Information 接口，在该接口中声明 Code 和 Name 两个属性，分别表示编号和名称；声明一个 ShowInfo 方法，用来输出信息。代码如下。

```
01 interface Information                    // 定义接口
02 {
```

```
    string Code { get; set; }                    //编号属性及实现
    string Name { get; set; }                    //名称属性及实现
    void ShowInfo();                             //用来输出信息
}
```

学习笔记

接口中的成员默认是公共的，因此，不允许加访问修饰符。

接口的实现通过类继承来实现，一个类虽然只能继承一个基类，但可以继承任意多个接口。在声明实现接口的类时，需要在继承列表中包含所实现的接口的名称，多个接口之间用逗号（,）分隔。

示例 12．使用接口模拟老师上课的场景

创建一个 IPerson 接口，定义姓名、年龄两个属性，定义说话、工作两个行为，再创建 Student 类和 Teacher 类，两者继承 IPerson 接口并重写各自的属性和行为。创建两个人 peter 和 mike，让这两个人模拟上课的场景。代码如下。

```
interface IPerson                                //定义 IPerson 接口
{
    string Name { get; set; }                    //姓名属性
    int Age { get; set; }                        //年龄属性
    void Speek();                                //说话行为
    void Work();                                 //工作行为
}
class Student : IPerson                          //定义学生类，继承自 IPerson 接口
{
    public string Name { get; set; }             //实现姓名属性
    private int age;                             //定义 age 字段，用来表示年龄
    public int Age                               //实现年龄属性
    {
        get
        {
            return age;
        }
        set
        {
            if (age > 0 && age < 120)            //控制输入范围
            {
                age = value;
            }
        }
    }
    public void Speek()                          //实现 Speek 方法
    {
        Console.WriteLine(Name + ": 老师好 ");
```

```
29      }
30      public void Work()                          // 实现 Work 方法
31      {
32          Console.WriteLine(Name + " 同学开始记笔记 ");
33      }
34  }
35  class Teacher : IPerson                         // 定义老师类，继承自 IPerson 接口
36  {
37      public string Name { get; set; }            // 实现姓名属性
38      private int age;                            // 定义 age 字段，用来表示年龄
39      public int Age                              // 实现年龄属性
40      {
41          get
42          {
43              return age;
44          }
45          set
46          {
47              if (age > 0 && age < 120)           // 控制输入范围
48              {
49                  age = value;
50              }
51          }
52      }
53      public void Speek()                         // 实现 Speek 方法
54      {
55          Console.WriteLine(Name + ": 同学们好 ");
56      }
57      public void Work()                          // 实现 Work 方法
58      {
59          Console.WriteLine(Name + " 老师开始上课 ");
60      }
61  }
62  class Program
63  {
64      static void Main(string[] args)
65      {
66          // 使用派生类对象创建接口数组
67          IPerson[] person = new IPerson[] { new Student(), new Teacher() };
68          person[0].Name = "peter";               // 为学生姓名赋值
69          person[0].Age = 20;                     // 为学生年龄赋值
70          person[1].Name = "mike";                // 为老师姓名赋值
71          person[1].Age = 40;                     // 为老师年龄赋值
72          person[0].Speek();                      // 学生的说话行为
73          person[1].Speek();                      // 老师的说话行为
74          Console.WriteLine();                    // 换行
75          person[1].Work();                       // 老师的工作行为
76          person[0].Work();                       // 学生的工作行为
```

```
77          Console.ReadLine();
78      }
79  }
```

学习笔记

第 3 行代码和第 4 行代码是定义接口中的属性的，这里并不会自动实现属性，只是提供了 get 访问器和 set 访问器，因此在派生类中可以使用自动实现属性的方式实现这两个属性。例如，在 JHInfo 类中实现 Code 和 Name 两个属性的代码可以修改如下。

```
01  public string Name { get; set; }              // 姓名属性
02  public int Age { get; set; }                  // 年龄属性
```

第 10 行、第 12 行、第 26 行、第 30 行、第 37 行、第 39 行、第 53 行和第 57 行的代码在实现接口成员时都使用了 public 修饰符。这里需要注意的是，在 C# 中实现接口成员（显式接口成员实现除外）时，必须添加 public 修饰符，不能省略或添加其他修饰符。

运行上面代码，得到如图 7.28 所示的内容。

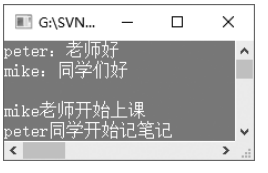


图 7.28　使用接口模拟老师上课的场景

学习笔记

在上面的示例中只继承了一个接口，接口还可以多重继承。在使用多重继承时，要继承的接口之间用逗号（,）分隔。例如，下面代码继承 3 个接口。

```
01  interface ITest1
02  {
03  }
04  interface ITest2
05  {
06  }
07  interface ITest3
08  {
09  }
10  class Test : ITest1, ITest2, ITest3  // 继承 3 个接口，接口之间用逗号分隔
11  {
12  }
```

第 8 章　Windows 交互式图形界面

Windows 环境中主流的应用程序都是窗体应用程序，Windows 窗体应用程序比 Windows 命令行应用程序复杂得多，理解它的结构的基础是理解 Windows 窗体，所以深刻认识 Windows 窗体变得尤为重要。本章将对 Windows 窗体应用程序的基本开发步骤、Form 窗体的使用和 MDI 窗体的使用进行详细讲解。

8.1　开发应用程序的步骤

在使用 C# 开发应用程序时，一般包括创建项目、界面设计、设置属性、编写程序代码、保存项目、运行程序等步骤。

下面以进、销、存管理系统的登录窗体为例说明开发应用程序的具体步骤。

1. 创建项目

在 Visual Studio 2017 开发环境中选择“文件”→“新建”→“项目”菜单命令，弹出“新建项目”对话框，如图 8.1 所示。

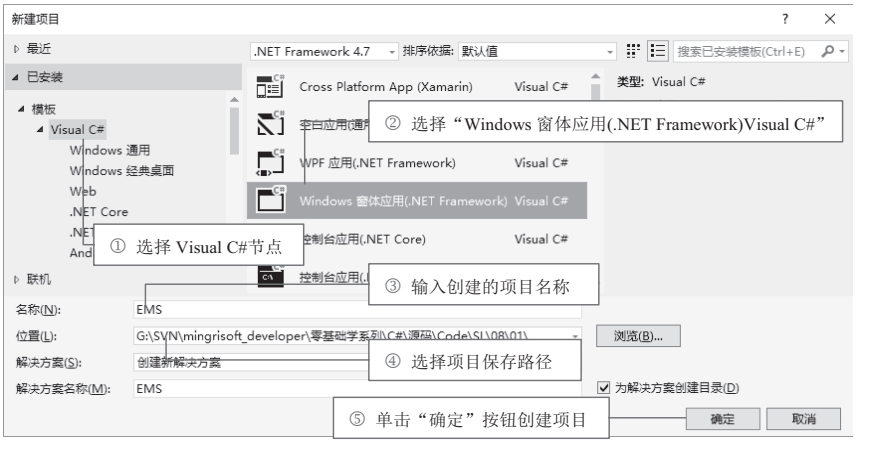


图 8.1　“新建项目”对话框

选择“Windows 窗体应用 (.NET Framework)Visual C#”，输入创建的项目名称，选择项目保存路径，然后单击“确定”按钮即可创建一个 Windows 窗体应用程序。创建完成的 Windows 窗体应用程序如图 8.2 所示。

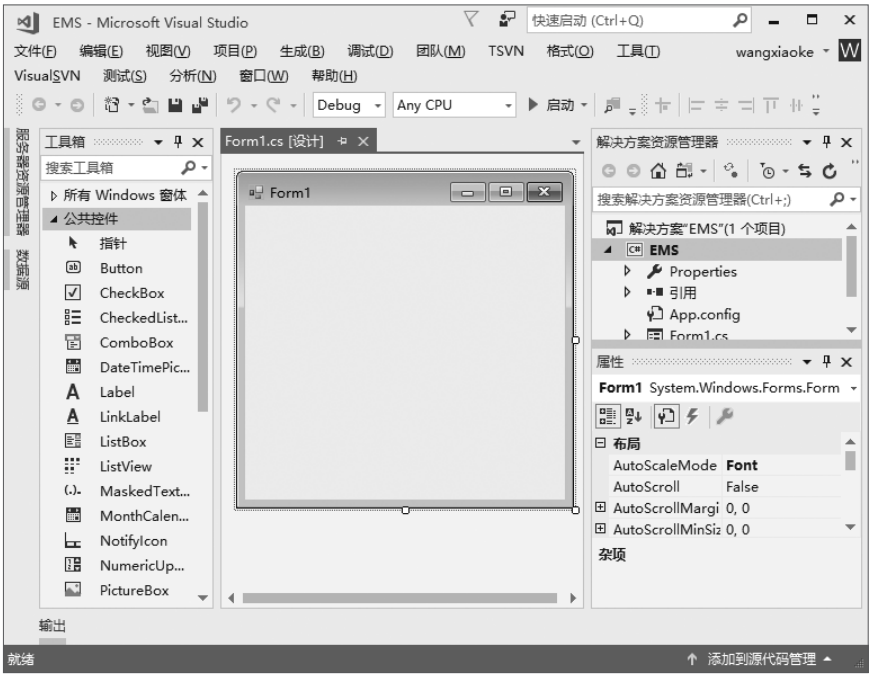


图 8.2　创建完成的 Windows 窗体应用程序

2. 界面设计

创建完项目后，在 Visual Studio 2017 开发环境中会有一个默认的窗体，可以通过工具箱向其中添加各种控件来设计窗体界面。具体步骤是，用鼠标选定工具箱中要添加的控件，然后将其拖放到窗体中的指定位置即可。本示例分别向窗体中添加两个 Label 控件、两个 TextBox 控件和两个 Button 控件，如图 8.3 所示。

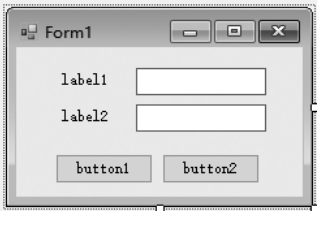


图 8.3　界面设计效果

3. 设置属性

在窗体中选择指定控件，在“属性”窗口中对控件的相应属性进行设置，如表 8.1 所示。

表 8.1　设置控件的相应属性

名　称	属　性	设　置　值
label1	Text	用户名：
label2	Text	密　码：
button1	Text	登录
button2	Text	退出

4. 编写程序代码

双击两个 Button 控件即可进入代码编辑器，并自动触发 Button 控件的 Click 事件，在该事件中即可编写代码，Button 控件的默认代码如下。

```
01 private void button1_Click(object sender, EventArgs e)
02 {
03
04 }
05 private void button2_Click(object sender, EventArgs e)
06 {
07
08 }
```

5. 保存项目

单击 Visual Studio 2017 开发环境工具栏中的按钮，或者选择“文件”→“全部保存”菜单命令，即可保存当前项目。

6. 运行程序

单击 Visual Studio 2017 开发环境工具栏中的 启动 按钮，或者选择“调试”→“开始调试”菜单命令，即可运行当前程序，程序运行效果如图 8.4 所示。

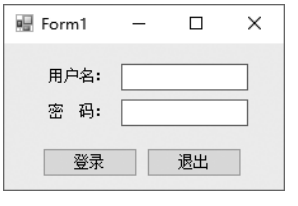


图 8.4　程序运行效果

8.2　Form 窗体

Form 窗体也称为窗口，它是向用户显示信息的可视界面，是 Windows 窗体应用程

序的基本单元。窗体都具有自己的特征，可以通过编程来进行设置。窗体也是对象，窗体类定义了生成窗体的模板，每实例化一个窗体类，就产生一个窗体。.NET 框架类库的 System.Windows.Forms 命名空间中定义的 Form 类是所有窗体类的基类。

如果要编写 Windows 窗体应用程序，推荐使用 Visual Studio 2017。Visual Studio 2017 提供了一个图形化的可视化窗体设计器，可以实现所见即所得的设计效果，可以快速开发 Windows 窗体应用程序。本节将对窗体的基本操作进行详细讲解。

8.2.1　添加或删除窗体

在添加或删除窗体时，首先要创建一个 Windows 窗体应用程序，创建步骤可以参考 8.1 节。

如果要在项目中添加一个新窗体，则可以在选中项目名称后单击鼠标右键，在弹出的快捷菜单中选择“添加”→“Windows 窗体”，或者“添加”→“新建项”，如图 8.5 所示。

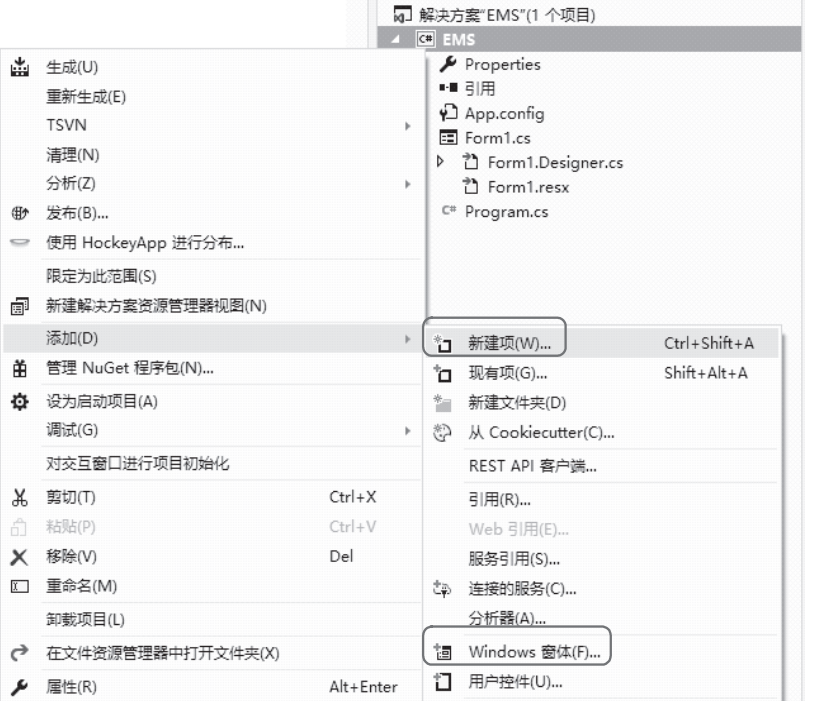


图 8.5　添加新窗体的右键快捷菜单

选择“新建项”或“Windows 窗体”后，都会打开“添加新项 -EMS”对话框，如图 8.6 所示。

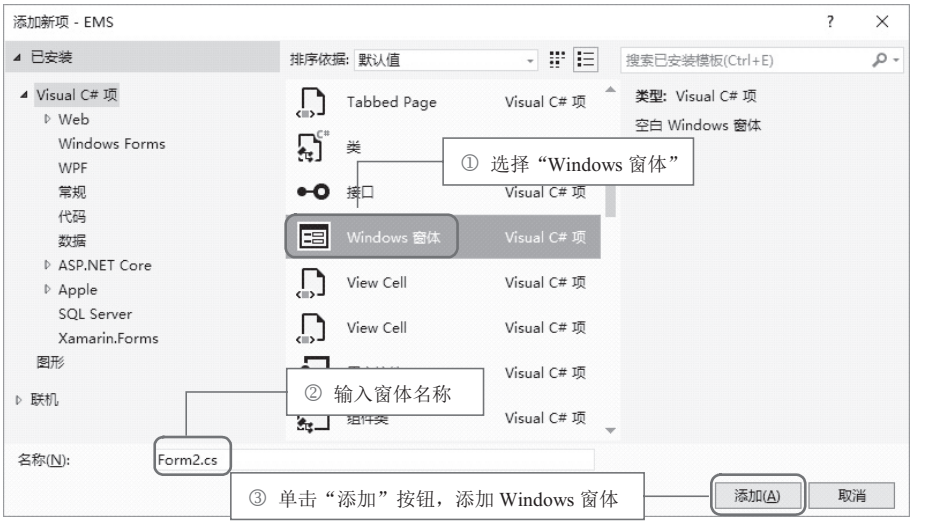


图 8.6 “添加新项 –EMS”对话框

选择“Windows 窗体”，输入窗体名称后，单击“添加”按钮，即可在项目中添加一个新的窗体。

学习笔记

在设置窗体的名称时，不要用关键字进行设置。

删除窗体的方法非常简单，只需要在要删除的窗体名称上单击鼠标右键，在弹出的快捷菜单中选择“删除”选项，即可将窗体删除。

8.2.2 多窗体的使用

一个完整的 Windows 应用程序是由多个窗体组成的，此时，就需要对多窗体设计有所了解。多窗体即向项目中添加多个窗体，在这些窗体中实现不同的功能。下面对多窗体的建立及如何设置启动窗体进行讲解。

1. 多窗体的建立

多窗体的建立是在某个项目中添加多个窗体，还是以 8.1 节中的项目为例，演示如何建立 Windows 多窗体应用程序，添加多窗体后的项目如图 8.7 所示。

具体添加窗体的方法在 8.2.1 节中已经进行了详细介绍，此处不再赘述。该处以在项目中添加 3 个窗体为例演示多窗体的建立，在实际项目中可以添加任意多个窗体。

图 8.7　添加多窗体后的项目

学习笔记

在添加多个窗体时，其名称不能相同。

2. 设置启动窗体

在向项目中添加了多个窗体以后，如果要调试程序，就必须要设置先运行的窗体。这样就需要设置项目的启动窗体。项目的启动窗体是在 Program.cs 文件中设置的，在 Program.cs 文件中改变 Run 方法的参数即可实现设置启动窗体。

Run 方法用于在当前线程上开始运行标准应用程序，并使指定窗体可见。

Run 方法的语法如下。

```
public static void Run (Form mainForm)
```

参数 mainForm 表示要设为启动窗体的对象。

例如，要将 Form1 窗体设置为项目的启动窗体，可以通过下面的代码实现。

```
Application.Run(new Form1());
```

8.2.3　窗体的属性

窗体包含一些基本的组成要素，如图标、标题、位置和背景等，这些要素可以通过窗体的“属性”窗口进行设置，也可以通过代码实现。但是为了快速开发 Windows 窗体应用程序，通常都是通过“属性”窗口进行设置的。下面详细介绍窗体的常见属性设置。

1. 更换窗体的图标

添加一个新的窗体后，窗体的图标是系统默认的图标。如果想更换窗体的图标，则可

以在“属性”窗口中设置窗体的 Icon 属性，窗体的默认图标与更换后的图标如图 8.8 所示。更换窗体图标的过程非常简单，具体操作步骤如下。

步骤 1，选中窗体，然后在窗体的“属性”窗口中选中 Icon 属性，会出现...按钮，如图 8.9 所示。

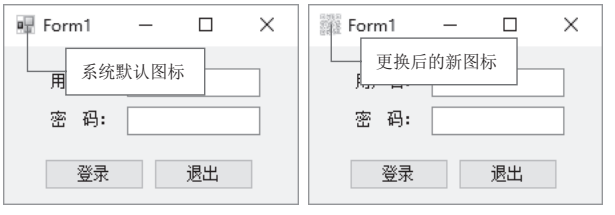


图 8.8　窗体的默认图标与更换后的图标

图 8.9　窗体的 Icon 属性

学习笔记

在设置窗体图标时，其图片格式只能是 .ico。

步骤 2，单击...按钮，打开选择图标文件的窗体，如图 8.10 所示。

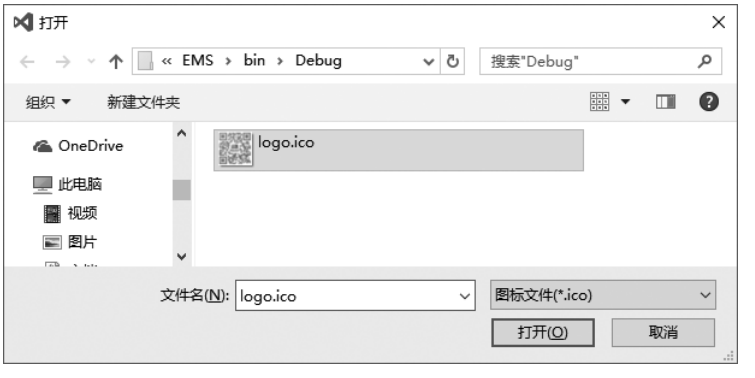


图 8.10　选择图标文件的窗体

步骤 3，选择新的窗体图标文件之后，单击“打开”按钮，完成窗体图标的更换。

2. 隐藏窗体的标题栏

在某些情况下需要隐藏窗体的标题栏。例如，软件的加载窗体大多数都采用无标题栏的窗体。通过设置窗体 FormBorderStyle 属性的属性值即可隐藏窗体的标题栏。FormBorderStyle 属性有 7 个属性值，其属性值及说明如表 8.2 所示。

表 8.2　FormBorderStyle 属性的属性值及说明

属　性　值	说　　明
Fixed3D	固定的三维边框
FixedDialog	固定的对话框样式的粗边框
FixedSingle	固定的单行边框
FixedToolWindow	不可调整大小的工具窗口边框
None	无边框
Sizable	可调整大小的边框
SizableToolWindow	可调整大小的工具窗口边框

如果要隐藏窗体的标题栏，则只需要将 FormBorderStyle 属性设置为 None 即可。

3. 设置窗体的显示位置

可以通过窗体的 StartPosition 属性设置窗体在加载时在显示器中的位置。StartPosition 属性有 5 个属性值，其属性值及说明如表 8.3 所示。

表 8.3　StartPosition 属性的属性值及说明

属　性　值	说　　明
CenterParent	窗体在其父窗体中居中
CenterScreen	窗体在当前显示窗口中居中，其尺寸在窗体大小中指定
Manual	窗体的位置由 Location 属性确定
WindowsDefaultBounds	窗体定位在 Windows 默认位置，其边界也由 Windows 默认指定
WindowsDefaultLocation	窗体定位在 Windows 默认位置，其尺寸在窗体大小中指定

在设置窗体的显示位置时，只需要根据不同的需要选择属性值即可。

4. 设置窗体的大小

在窗体的属性中，通过 Size 属性设置窗体的大小。双击窗体”属性”窗口中的 Size 属性，可以看到其下拉菜单中有 Width 和 Height 两个属性，这两个属性分别用于设置窗体的宽和高。修改窗体的大小，只需要更改 Width 属性和 Height 属性的值即可。

学习笔记

在设置窗体的大小时，其值是 Int32 类型（整数）的，不能使用单精度和双精度（小数）进行设置。

5. 设置窗体的背景图片

为了使窗体设计更加美观，通常会设置窗体的背景，这主要通过设置窗体的

BackgroundImage 属性实现，具体操作如下。

选中窗体“属性”窗口中的 BackgroundImage 属性，会出现...按钮，如图 8.11 所示。

单击...按钮，打开“选择资源”对话框，如图 8.12 所示。

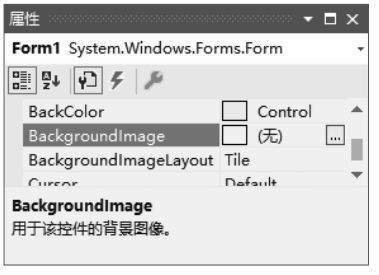


图 8.11　BackgroundImage 属性

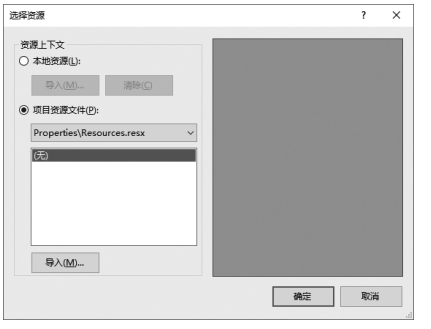


图 8.12　“选择资源”对话框

在如图 8.12 所示的“选择资源”对话框中，有两个单选按钮，一个是“本地资源”单选按钮，另一个是“项目资源文件”单选按钮，两者的差别是选中“本地资源”单选按钮后，直接选择图片，保存的是图片的路径；而选中“项目资源文件”单选按钮后，会将选择的图片保存到项目资源文件 Resources.resx 中。无论选择哪种方式，都需要单击“导入”按钮选择背景图片，单击“确定”按钮完成窗体背景图片的设置。Form1 窗体背景图片设置前、后对比如图 8.13 所示。

图 8.13　Form1 窗体背景图片设置前、后对比

8.2.4　窗体的显示与隐藏

1. 窗体的显示

如果要在一个窗体中通过按钮打开另一个窗体，就必须通过调用 Show 方法显示窗体，语法如下。

```
public void Show ()
```

例如，在 Form1 窗体中添加一个 Button 按钮，在按钮的 Click 事件中调用 Show 方法，

打开 Form2 窗体，关键代码如下。

```
01  Form2 frm2 = new Form2();                    // 创建 Form2 窗体的对象
02  frm2.Show();                                 // 调用 Show 方法显示 Form2 窗体
```

2. 窗体的隐藏

通过调用 Hide 方法可以隐藏窗体，语法如下。

```
public void Hide ()
```

例如，在 Form1 窗体中打开 Form2 窗体后，隐藏当前窗体，关键代码如下。

```
01  Form2 frm2 = new Form2();                    // 创建 Form2 窗体的对象
02  frm2.Show();                                 // 调用 Show 方法显示 Form2 窗体
03  this.Hide();                                 // 调用 Hide 方法隐藏当前窗体
```

8.2.5　窗体的事件

Windows 是事件驱动的操作系统，对 Form 类的任何交互都是基于事件来实现的。Form 类提供了大量的事件用于响应对窗体执行的各种操作。下面详细介绍窗体的 Click（单击）事件、Load（加载）事件和 FormClosing（关闭）事件。

1. Click 事件

当单击窗体时，将会触发窗体的 Click 事件，语法如下。

```
public event EventHandler Click
```

例如，在窗体的 Click 事件中编写代码，实现当单击窗体时，弹出提示框，代码如下。

```
01  private void Form1_Click(object sender, EventArgs e)
02  {
03      MessageBox.Show(" 已经单击了窗体！ ");      // 弹出提示框
04  }
```

学习笔记

上面代码中的第 3 行用到了 MessageBox 类，该类是一个消息提示框类，其 Show 方法用来显示对话框。

运行上面代码，在窗体中进行单击，弹出提示框。单击窗体触发 Click 事件，如图 8.14 所示。

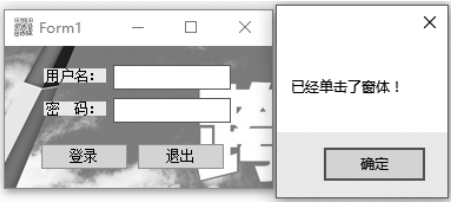


图 8.14　单击窗体触发 Click 事件

学习笔记

在触发窗体或控件的相关事件时，只需要选中指定的窗体或控件，单击鼠标右键，在弹出的快捷菜单中选择“属性”，然后在弹出的“属性”对话框中单击⚡按钮，在列表中找到相应的事件名称，双击即可生成该事件的代码，如图 8.15 所示。

2. Load 事件

窗体在加载时，将触发窗体的 Load 事件，语法如下。

```
public event EventHandler Load
```

例如，当窗体加载时，弹出提示框，询问是否查看窗体，单击“是”按钮，查看窗体，代码如下。

```
01  private void Form1_Load(object sender, EventArgs e)// 窗体的 Load 事件，在加载时执行
02  {
03      // 使用 if 语句判断是否单击了“是”按钮
04      if (MessageBox.Show("是否查看窗体！", "", MessageBoxButtons.YesNo,
MessageBoxIcon.Information) == DialogResult.Yes)
05      {
06      }
07  }
```

运行上面代码，在窗体显示之前，会弹出如图 8.16 所示的对话框。

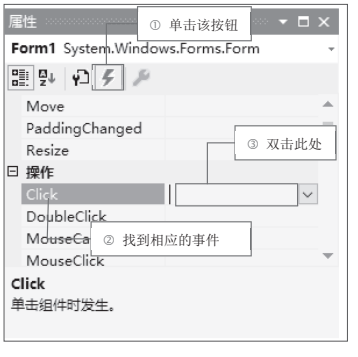


图 8.15　触发窗体或控件的相关事件

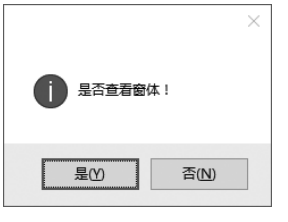


图 8.16　触发窗体的 Load 事件

3. FormClosing 事件

当窗体关闭时，将触发窗体的 FormClosing 事件，语法如下。

```
public event FormClosingEventHandler FormClosing
```

例如，实现在关闭窗体之前，弹出提示框，询问是否关闭当前窗体，单击“是”按钮，关闭窗体；单击“否”按钮，不关闭窗体，代码如下。

```
01 private void Form1_FormClosing(object sender, FormClosingEventArgs e)
02 {
03     DialogResult dr = MessageBox.Show("是否关闭窗体", "提示", MessageBoxButtons.
 YesNo, MessageBoxIcon.Warning);          // 创建对话框对象
04     if (dr == DialogResult.Yes)          // 使用 if 语句判断是否单击“是”按钮
05     {
06         e.Cancel = false;                // 如果单击“是”按钮则关闭窗体
07     }
08     else                                 // 否则
09     {
10         e.Cancel = true;                 // 不执行操作
11     }
12 }
```

运行上面代码，单击窗体中的关闭按钮，如图 8.17 所示，弹出如图 8.18 所示的提示框。在该提示框中，单击“是”按钮将会关闭窗体，单击“否”按钮不执行任何操作。

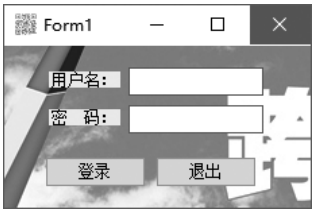


图 8.17　单击窗体中的关闭按钮

图 8.18　单击“是”或“否”按钮

学习笔记

可以使用 FormClosing 事件执行一些任务，如释放窗体使用的资源，还可使用此事件保存窗体中的信息或更新其父窗体。

8.3　MDI 窗体

窗体是所有界面的基础，这就意味着为了打开多个文档，需要具有能够同时处理多个

窗体的应用程序。为了适应这种需求，产生了 MDI 窗体，即多文档界面。本节将对 MDI 窗体进行详细讲解。

8.3.1 MDI 窗体的概念

多文档界面简称 MDI（Multiple-Document Interface）窗体。MDI 窗体用于同时显示多个文档，每个文档显示在各自的窗口中。MDI 窗体中通常有包含子菜单的窗口菜单，用于在窗口或文档之间进行切换。MDI 窗体十分常见。图 8.19 所示为一个 MDI 窗体界面。

MDI 窗体的应用非常广泛。例如，如果某公司的库存系统需要实现自动化，则需要使用窗体来输入客户和货物的数据、发出订单及跟踪订单。这些窗体必须链接或从属于一个界面，并且必须能够同时处理多个文件。这样，就需要建立 MDI 窗体以解决这些需求。

图 8.19　一个 MDI 窗体界面

8.3.2 如何设置 MDI 窗体

在 MDI 窗体中，起到容器作用的窗体被称为父窗体，可以放在父窗体中的其他窗体被称为子窗体，也称为 MDI 子窗体。当 MDI 应用程序启动时，首先会显示父窗体。所有的子窗体都在父窗体中打开，在父窗体中可以在任何时候打开多个子窗体。每个应用程序只能有一个父窗体，子窗体不能移出父窗体的框架区域。下面介绍如何将窗体设置成父窗体或子窗体。

1. 设置父窗体

如果要将某个窗体设置为父窗体，则只要在窗体的“属性”窗口中将 IsMdiContainer 属性设置为 True 即可，如图 8.20 所示。

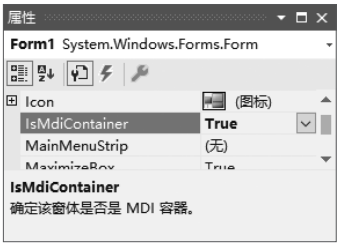


图 8.20　设置父窗体

2. 设置子窗体

设置完父窗体，通过设置某个窗体的 MdiParent 属性来确定子窗体，语法如下。

```
public Form MdiParent { get; set; }
```

属性值表示 MDI 父窗体。

例如，将 Form2、Form3 这两个窗体设置成子窗体，并且在父窗体中打开这两个子窗体，代码如下。

```
01  Form2 frm2 = new Form2();       // 创建 Form2 窗体的对象
02  frm2.MdiParent = this;          // 设置 MdiParent 属性，将当前窗体作为父窗体
03  frm2.Show();                    // 使用 Show 方法打开窗体
04  Form3 frm3 = new Form3();       // 创建 Form3 窗体的对象
05  frm3.MdiParent = this;          // 设置 MdiParent 属性，将当前窗体作为父窗体
06  frm3.Show();                    // 使用 Show 方法打开窗体
```

8.3.3　排列 MDI 子窗体

如果一个 MDI 窗体中有多个子窗体同时打开，假如不对其排列顺序进行调整，那么界面会非常混乱，而且不容易浏览，如何解决这个问题呢？可以通过使用带有 MdiLayout 枚举的 LayoutMdi 方法来排列多文档界面父窗体中的子窗体，语法如下。

```
public void LayoutMdi (MdiLayout value)
```

参数 value 用来定义 MDI 子窗体的布局，它的值是 MdiLayout 枚举值之一。MdiLayout 枚举用于指定 MDI 父窗体中子窗体的布局，其枚举成员及说明如表 8.4 所示。

表 8.4　MdiLayout 枚举的枚举成员及说明

枚 举 成 员	说　　明
Cascade	所有 MDI 子窗体均层叠在 MDI 父窗体的工作区内
TileHorizontal	所有 MDI 子窗体均水平平铺在 MDI 父窗体的工作区内
TileVertical	所有 MDI 子窗体均垂直平铺在 MDI 父窗体的工作区内

示例 1．排列 MDI 父窗体中的多个子窗体

程序开发步骤如下。

步骤 1，新建一个 Windows 窗体应用程序，并将其命名为 Demo，默认窗体为 Form1.cs。

步骤 2，将窗体 Form1 的 IsMdiContainer 属性设置为 True，以用作 MDI 父窗体，然后添加 3 个 Windows 窗体，用作 MDI 子窗体。

步骤 3，在 Form1 窗体中，添加一个 MenuStrip 控件，用作该父窗体的菜单项。

步骤 4，通过 MenuStrip 控件建立 4 个菜单项，分别为“加载子窗体”“水平平铺”“垂直平铺”“层叠排列”。运行程序时，单击“加载子窗体”菜单项后，可以加载所有的子窗体，对应代码如下。

```
01 private void 加载子窗体 ToolStripMenuItem_Click(object sender, EventArgs e)
02 {
03     Form2 frm2 = new Form2();          // 创建 Form2 窗体的对象
04     frm2.MdiParent = this;             // 设置 MdiParent 属性，将当前窗体作为父窗体
05     frm2.Show();                       // 使用 Show 方法打开窗体
06     Form3 frm3 = new Form3();          // 创建 Form3 窗体的对象
07     frm3.MdiParent = this;             // 设置 MdiParent 属性，将当前窗体作为父窗体
08     frm3.Show();                       // 使用 Show 方法打开窗体
09     Form4 frm4 = new Form4();          // 创建 Form4 窗体的对象
10     frm4.MdiParent = this;             // 设置 MdiParent 属性，将当前窗体作为父窗体
11     frm4.Show();                       // 使用 Show 方法打开窗体
12 }
```

步骤 5，加载所有的子窗体之后，单击“水平平铺”菜单项，使窗体中所有的子窗体水平排列，代码如下。

```
01 private void 水平平铺 ToolStripMenuItem_Click(object sender, EventArgs e)
02 {
03     LayoutMdi(MdiLayout.TileHorizontal);// 使用 MdiLayout 枚举实现窗体的水平平铺
04 }
```

步骤 6，单击“垂直平铺”菜单项，使窗体中所有的子窗体垂直排列，代码如下。

```
01 private void 垂直平铺 ToolStripMenuItem_Click(object sender, EventArgs e)
02 {
```

```
03      LayoutMdi(MdiLayout.TileVertical); // 使用 MdiLayout 枚举实现窗体的垂直平铺
04  }
```

步骤 7，单击“层叠排列”菜单项，使窗体中所有的子窗体层叠排列，代码如下。

```
01  private void 层叠排列 ToolStripMenuItem_Click(object sender, EventArgs e)
02  {
03      LayoutMdi(MdiLayout.Cascade);     // 使用 MdiLayout 枚举实现窗体的层叠排列
04  }
```

运行程序，单击“加载子窗体”效果如图 8.21 所示；单击“水平平铺”效果如图 8.22 所示；单击“垂直平铺”效果如图 8.23 所示；单击“层叠排列”效果如图 8.24 所示。

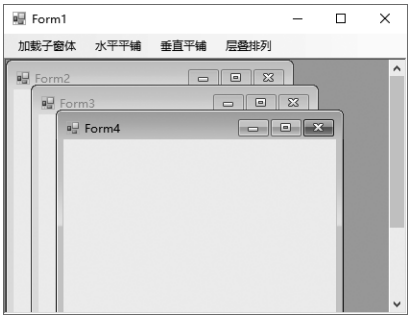


图 8.21　单击“加载子窗体”效果

图 8.22　单击“水平平铺”效果

图 8.23　单击“垂直平铺”效果

图 8.24　单击“层叠排列”效果

第 9 章　Windows 控件——C/S 程序的基础

控件是窗体的基本组成单位，通过使用控件可以高效地开发 Windows 窗体应用程序。所以，熟练掌握控件是合理、有效地进行程序开发的重要前提。本章将对开发 Windows 窗体应用程序中经常用到的控件进行详细讲解。

9.1　控件概述

控件是用户可以用来输入或操作数据的对象，相当于汽车的方向盘、油门、刹车、离合器等，它们都是对汽车进行操作的控件。在 C# 中，控件的基类是位于 System.Windows.Forms 命名空间下的 Control 类。Control 类定义了控件类的共同属性、方法和事件，其他的控件类都直接或间接地派生自这个基类。

可以通过控件默认的名称调用控件。如果自定义控件名称，则应该遵循控件的命名规范。控件的常用命名规范如表 9.1 所示。

表 9.1　控件的常用命名规范

控 件 名 称	命　　名
TextBox	txt
Button	btn
ComboBox	cbox
Label	lab
DataGridView	dgv
ListBox	lbox
Timer	tmr
CheckBox	chbox
RichTextBox	rtbox
RadioButton	rbtn

续表

控 件 名 称	命　　名
Panel	pl
GroupBox	gbox
ImageList	ilist
ListView	lv
TreeView	tv
MenuStrip	menu
ToolStrip	tool
StatusStrip	status
……	……

9.2　控件的相关操作

对控件的相关操作包括添加控件、对齐控件、锁定控件和删除控件等，在以下内容中将会对这几种操作进行讲解。

9.2.1　添加控件

可以通过“在窗体中绘制控件”“将控件拖曳到窗体中”“以编程方式向窗体中添加控件”这 3 种方法添加控件。

1. 在窗体中绘制控件

在工具箱中单击要添加到窗体的控件，将鼠标指针放在该窗体中希望控件左上角所处位置，按住鼠标左键不放，然后拖动，将鼠标指针拖曳至希望该控件右下角所处位置，释放鼠标左键，控件即按指定的位置和大小添加到窗体中。

2. 将控件拖曳到窗体中

在工具箱中单击所需要的控件并将其拖曳到窗体中，控件以其默认大小添加到窗体中的指定位置。

3. 以编程方式向窗体中添加控件

通过 new 关键字实例化要添加控件所在的类，然后将实例化的控件添加到窗体中。

例如，通过 Button 按钮的 Click 事件添加一个 TextBox 控件，代码如下。

```
01 private void button1_Click(object sender, System.EventArgs e)//Button 按钮的 Click 事件
02 {
03     TextBox myText = new TextBox();                    // 实例化 TextBox 类
04     myText.Location = new Point(25, 25);               // 设置 TextBox 类放的位置
05     this.Controls.Add(myText);                         // 将控件添加到当前窗体中
06 }
```

9.2.2 对齐控件

选定一组控件，这些控件需要对齐。在执行对齐之前，首先选定主导控件（第一个被选定的控件就是主导控件），控件组的最终位置取决于主导控件的位置，再选择菜单栏中的“格式”→“对齐”选项，然后选择对齐方式。

左对齐：将选定控件沿它们的左边对齐。

居中对齐：将选定控件沿它们的中心点水平对齐。

右对齐：将选定控件沿它们的右边对齐。

顶端对齐：将选定控件沿它们的顶边对齐。

中间对齐：将选定控件沿它们的中心点垂直对齐。

底部对齐：将选定控件沿它们的底边对齐。

9.2.3 删除控件

删除控件的方法非常简单，可以在控件上单击鼠标右键，在弹出的快捷菜单中选择“删除”选项进行删除；也可以选中控件，然后按下 <Delete> 键，对控件进行删除。

9.3 Windows 控件的使用

在 Windows 应用程序开发中，控件的使用非常重要，本节将对 Windows 常用控件的使用进行详细讲解。

9.3.1 Label 控件

Label 控件又称为标签控件，它主要用于显示用户不能编辑的文本，标识窗体上的对

象（如给文本框、列表框添加描述信息等）；另外，也可以通过编写代码来设置要显示的文本信息。

1. 设置标签文本

可以通过两种方法设置 Label 控件显示的文本：第一种是直接在 Label 控件的属性面板中设置 Text 属性；第二种是通过代码设置 Text 属性。

例如，向窗体中拖曳一个 Label 控件，然后将其显示文本设置为“用户名：”，代码如下。

```
label1.Text = "用户名：";          // 设置 Label 控件的 Text 属性
```

2. 显示 / 隐藏控件

通过设置 Visible 属性来设置显示 / 隐藏 Label 控件，如果 Visible 属性的值为 true，则显示控件；如果 Visible 属性的值为 false，则隐藏控件。

例如，通过代码将 Label 控件设置为可见，将其 Visible 属性设置为 true 即可，代码如下。

```
label1.Visible = true;             // 设置 Label 控件的 Visible 属性
```

9.3.2　Button 控件

Button 控件，又称为按钮控件，它允许用户通过单击来执行操作。Button 控件既可以显示文本，也可以显示图像，当该控件被单击时，它看起来像是被按下，然后被释放。Button 控件最常用的是 Text 属性和 Click 事件，其中，Text 属性用来设置 Button 控件显示的文本，Click 事件用来指定单击 Button 控件时执行的操作。

示例 1. 制作登录和退出按钮

创建一个 Windows 应用程序，在默认窗体中添加两个 Label 控件，分别设置它们的 Text 属性为“用户名：”和“密码：”；添加两个 Button 控件，分别设置它们的 Text 属性为“登录”和“退出”，然后触发它们的 Click 事件，执行相应的操作。代码如下。

```
01 private void button1_Click(object sender, EventArgs e)
02 {
03     MessageBox.Show("系统登录");              // 输出信息提示
04 }
05 private void button2_Click(object sender, EventArgs e)
06 {
07     Application.Exit();                       // 退出当前程序
08 }
```

运行上面代码会显示 Button 控件，如图 9.1 所示，单击“登录”按钮会弹出如图 9.2 所示的信息提示，单击“退出”按钮退出当前的程序。

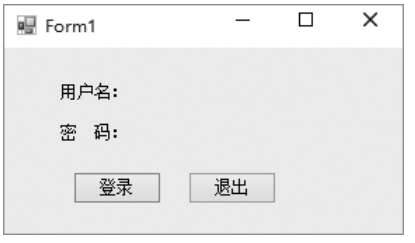


图 9.1　显示 Button 控件

图 9.2　弹出信息提示

9.3.3　TextBox 控件

TextBox 控件又称为文本框控件，它主要用于获取用户输入的数据或显示文本，它通常用于可编辑文本，也可以使其成为只读控件。文本框可以显示多行，开发人员可以使文本换行以便符合控件的大小。

下面对 TextBox 控件的一些常见使用方法进行介绍。

1. 创建只读文本框

通过设置文本框控件（TextBox 控件）的 ReadOnly 属性，可以设置文本框是否为只读。如果 ReadOnly 属性为 true，那么不能编辑文本框，只能通过文本框显示数据。

例如，将文本框设置为只读，代码如下。

```
textBox1.ReadOnly = true;          // 将文本框设置为只读
```

2. 创建密码文本框

通过设置文本框的 PasswordChar 属性或 UseSystemPasswordChar 属性可以将文本框设置成密码文本框。使用 PasswordChar 属性可以设置输入密码时文本框中显示的字符（如将密码显示成“*”或“#”等）。如果将 UseSystemPasswordChar 属性设置为 true，则当输入密码时，在文本框中将密码显示为“*”。

示例 2．制作登录窗体

修改示例 1，在窗体中添加两个 TextBox 控件，分别用来输入用户名和密码，其中，将第二个 TextBox 控件的 PasswordChar 属性设置为 *，以便使密码文本框中的字符显示为“*”，代码如下。

```
01 private void Form1_Load(object sender, EventArgs e)      // 窗体的 Load 事件
02 {
03     textBox2.PasswordChar = '*'; // 设置文本框的 PasswordChar 属性为字符 *
04 }
```

运行上面代码，得到如图 9.3 所示的结果。

3. 创建多行文本框

在默认情况下，TextBox 控件只允许输入单行数据，如果将其 Multiline 属性设置为 true，则在 TextBox 控件中即可输入多行数据。

例如，将文本框的 Multiline 属性设置为 true，使其能够输入多行数据，代码如下。

```
textBox1.Multiline = true;          // 设置文本框的 Multiline 属性
```

多行文本框效果如图 9.4 所示。

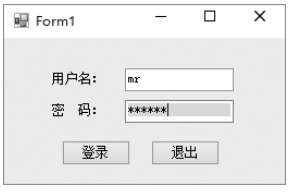


图 9.3　密码文本框

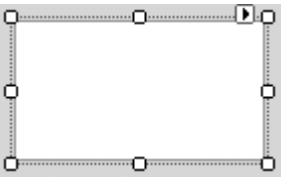


图 9.4　多行文本框效果

4. 响应文本框的 TextChanged 事件

当文本框中的文本发生更改时，将会引发文本框的 TextChanged 事件。

例如，在文本框的 TextChanged 事件中编写代码，实现当文本框中的文本更改时，Label 控件中显示更改后的文本，代码如下。

```
01 private void textBox1_TextChanged(object sender, EventArgs e)
02 {
03     label1.Text = textBox1.Text;//Label 控件显示的文字随文本框中的数据而改变
04 }
```

9.3.4　RadioButton 控件

RadioButton 控件又称为单选按钮控件，它为用户提供由两个或多个互斥选项组成的选项集。当用户选中某单选按钮时，同一组中的其他单选按钮不能同时选定。

学习笔记

单选按钮必须在同一组中才能实现单选效果。

下面详细介绍 RadioButton 控件的一些常见用法。

1. 判断单选按钮是否选中

通过 Checked 属性可以判断 RadioButton 控件的选中状态，如果属性值是 true，则控件被选中；如果属性值为 false，则控件选中状态被取消。

2. 响应单选按钮选中状态更改事件

当 RadioButton 控件的选中状态发生更改时，会引发控件的 CheckedChanged 事件。

示例 3．登录时选择用户角色

修改示例 2，在窗体中添加两个 RadioButton 控件，用来选择管理员登录还是普通用户登录，它们的 Text 属性分别设置为“管理员”和“普通用户”，然后分别触发这两个 RadioButton 控件的 CheckedChanged 事件，在该事件中，通过判断其 Checked 属性确定是否选中。代码如下。

```
01 private void radioButton1_CheckedChanged(object sender, EventArgs e)
02 {
03     if (radioButton1.Checked)                    // 判断“管理员”单选按钮是否选中
04     {
05         MessageBox.Show(" 您选择的是管理员登录 ");
06     }
07 }
08 private void radioButton2_CheckedChanged(object sender, EventArgs e)
09 {
10     if (radioButton2.Checked)                    // 判断“普通用户”单选按钮是否选中
11     {
12         MessageBox.Show(" 您选择的是普通用户登录 ");
13     }
14 }
```

运行程序，选中“管理员”单选按钮，弹出“您选择的是管理员登录”提示框，如图 9.5 所示；选中“普通用户”单选按钮，弹出“您选择的是普通用户登录”提示框，如图 9.6 所示。

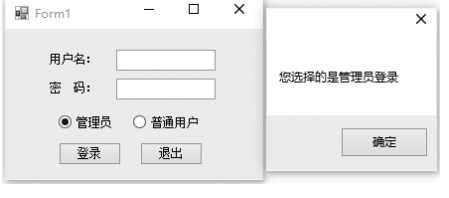


图 9.5　选中“管理员”单选按钮出现的提示信息

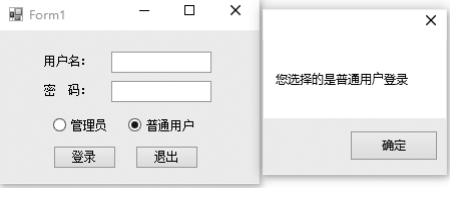


图 9.6　选中“普通用户”单选按钮出现的提示信息

9.3.5　CheckBox 控件

CheckBox 控件又称为复选框控件，它用来表示是否选取了某个选项条件，常用于为用户提供是 / 否或真 / 假选项。

下面详细介绍 CheckBox 控件的一些常见用法。

1. 判断复选框是否被选中

通过 CheckState 属性可以判断复选框是否被选中。CheckState 属性的返回值是 Checked 或 Unchecked，返回值 Checked 表示控件处在选中状态，而返回值 Unchecked 表示控件已经取消选中状态。

学习笔记

CheckBox 控件指示某个特定条件是处于打开状态还是处于关闭状态，它常用于为用户提供是 / 否或真 / 假选项。可以成组使用 CheckBox 控件以显示多重选项，用户可以从中选择一项或多项。

2. 响应复选框的选中状态更改事件

当 CheckBox 控件的选择状态发生改变时，将会引发控件的 CheckStateChanged 事件。

示例 4. 设置并显示用户权限

创建一个 Windows 窗体应用程序，通过复选框的选中状态设置用户的操作权限。在默认窗体中添加 5 个 CheckBox 控件，Text 属性分别设置为“基本信息管理”“进货管理”“销售管理”“库存管理”“系统管理”，它们主要用来表示要设置的权限；添加一个 Button 控件，用来显示选择的权限。代码如下。

```
01 private void button1_Click(object sender, EventArgs e)
02 {
03     string strPop = "您选择的权限如下：";
04     foreach (Control ctrl in this.Controls)          // 遍历窗体中的所有控件
05     {
06         if (ctrl.GetType().Name == "CheckBox")       // 判断是否为 CheckBox
07         {
08             CheckBox cBox = (CheckBox)ctrl;          // 创建 CheckBox 对象
09             if (cBox.Checked == true)          // 判断 CheckBox 控件是否选中
10             {
11                 strPop += "\n" + cBox.Text;          // 获取 CheckBox 控件的文本
12             }
13         }
14     }
15     MessageBox.Show(strPop);
16 }
```

运行上面代码，会得到如图 9.7 所示的内容。

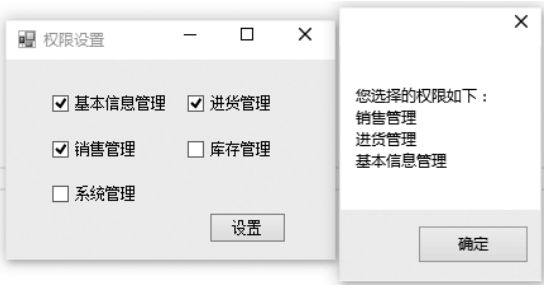


图 9.7　通过复选框的选中状态设置用户权限

9.3.6　RichTextBox 控件

RichTextBox 控件又称为有格式文本框控件，它主要用于显示、输入和操作带有格式的文本。例如，它可以实现显示字体、颜色、链接、从文件加载文本及嵌入的图像、撤销和重复编辑操作及查找指定的字符等功能。

下面详细介绍 RichTextBox 控件的常见用法。

1. 在 RichTextBox 控件中显示滚动条

通过设置 RichTextBox 控件的 Multiline 属性，可以控制在控件中是否显示滚动条。将 Multiline 属性设置为 true，则显示滚动条；否则，不显示滚动条。在默认情况下，此属性被设置为 true。滚动条分为水平滚动条和垂直滚动条，通过 ScrollBars 属性可以设置如何显示滚动条。ScrollBars 属性的属性值及说明如表 9.2 所示。

表 9.2　ScrollBars 属性的属性值及说明

属 性 值	说　明
Both	只有当文本超过控件的宽度或长度时，才显示水平滚动条或垂直滚动条，或两个滚动条都显示
None	从不显示任何类型的滚动条
Horizontal	只有当文本超过控件的宽度时，才显示水平滚动条。必须将 WordWrap 属性设置为 false 才会出现这种情况
Vertical	只有当文本超过控件的高度时，才显示垂直滚动条
ForcedHorizontal	当 WordWrap 属性设置为 false 时，显示水平滚动条。在文本未超过控件的宽度时，该滚动条显示为浅灰色
ForcedVertical	始终显示垂直滚动条。在文本未超过控件的长度时，该滚动条显示为浅灰色
ForcedBoth	始终显示垂直滚动条。当 WordWrap 属性设置为 false 时，显示水平滚动条。在文本未超过控件的宽度或长度时，两个滚动条均显示为灰色

例如，使 RichTextBox 控件只显示垂直滚动条。首先将 Multiline 属性设置为 true，然

后设置 ScrollBars 属性的值为 Vertical，代码如下。

```
01  // 将 Multiline 属性设置为 true，实现多行显示
02  richTextBox1.Multiline = true;
03  // 设置 ScrollBars 属性实现只显示垂直滚动条
04  richTextBox1.ScrollBars = RichTextBoxScrollBars.Vertical;
```

显示垂直滚动条效果如图 9.8 所示。

2. 在 RichTextBox 控件中设置字体属性

在设置 RichTextBox 控件中的字体属性时可以使用 SelectionFont 属性和 SelectionColor 属性，其中，SelectionFont 属性用来设置字体系列、大小和字样，而 SelectionColor 属性用来设置字体的颜色。

例如，将 RichTextBox 控件中文本的字体设置为楷体，字体大小设置为 12，字样设置为粗体，文本的颜色设置为红色，代码如下。

```
01  // 设置 SelectionFont 属性实现控件中的字体为楷体，字体大小为 12，字样是粗体
02  richTextBox1.SelectionFont = new Font(" 楷体 ", 12, FontStyle.Bold);
03  // 设置 SelectionColor 属性实现控件中的文本颜色为红色
04  richTextBox1.SelectionColor = System.Drawing.Color.Red;
```

设置控件中文本的字体属性效果如图 9.9 所示。

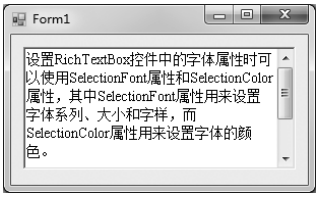


图 9.8　显示垂直滚动条效果

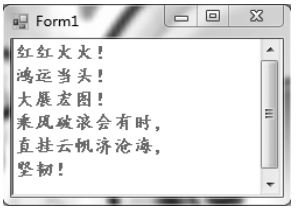


图 9.9　设置控件中文本的字体属性效果

3. 将 RichTextBox 控件显示为超链接样式

利用 RichTextBox 控件可以将 Web 链接显示为彩色或下画线形式，然后通过编写代码，在单击链接时打开浏览器窗口，显示链接文本中指定的网站。其设计思路是，首先通过 Text 属性设置控件中含有超链接的文本，然后在控件的 LinkClicked 事件中编写事件处理程序，将所需要的文本发送到浏览器。

例如，在 RichTextBox 控件的文本内容中含有超链接地址（超链接地址显示为彩色并且带有下画线），单击该超链接地址将打开相应的网站。代码如下。

```
01  private void Form1_Load(object sender, EventArgs e)
02  {
03      richTextBox1.Text = " 欢迎登录 http://www.mingrisoft.com 明日学院网 ";
```

```
04  }
05  private void richTextBox1_LinkClicked(object sender, LinkClickedEventArgs e)
06  {
07      // 在控件的 LinkClicked 事件中编写如下代码实现内容中的网址带下画线
08      System.Diagnostics.Process.Start(e.LinkText);
09  }
```

文本中含有超链接地址效果如图 9.10 所示。

4. 在 RichTextBox 控件中设置段落格式

RichTextBox 控件具有多个用于设置所显示文本的格式的选项，比如可以通过设置 SelectionBullet 属性将选定的段落设置为项目符号列表的格式，也可以使用 SelectionIndent 和 SelectionHangingIndent 属性设置段落相对于控件的左、右边缘的缩进位置。

例如，将 RichTextBox 控件的 SelectionBullet 属性设为 true，使控件中的内容以项目符号列表的格式排列，代码如下。

```
richTextBox1.SelectionBullet = true;
```

向 RichTextBox 控件中输入数据后，效果如图 9.11 所示。

图 9.10　文本中含有超链接地址效果

图 9.11　将控件中的内容设置为项目符号列表效果

9.3.7　ComboBox 控件

ComboBox 控件又称为下拉组合框控件，它主要用于在下拉组合框中显示数据。该控件主要由两部分组成：第一部分是一个允许用户输入列表项的文本框；第二部分是一个列表框，它显示一个选项列表，用户可以从中选择项。

下面详细介绍 ComboBox 控件的一些常见用法。

1. 创建只可以选择的下拉组合框

通过设置 ComboBox 控件的 DropDownStyle 属性，可以将其设置成可以选择的下拉组合框。DropDownStyle 属性有 3 个属性值，这 3 个属性值对应不同的样式，如下所示。

Simple：使得 ComboBox 控件的列表部分总是可见的。

DropDown：DropDownStyle 属性的默认值，使得用户可以编辑 ComboBox 控件的文本框部分，只有单击右侧的箭头才能显示列表部分。

DropDownList：用户不能编辑 ComboBox 控件的文本框部分，呈现下拉列表框的样式。

将 ComboBox 控件的 DropDownStyle 属性设置为 DropDownList，它就只能是可以选择的下拉列表框，而不能编辑文本框部分的内容。

2. 响应下拉组合框的选项值更改事件

当下拉列表的选择项发生改变时，将会引发控件的 SelectedValueChanged 事件。

示例 5．使用 ComboBox 控件选择职位

创建一个 Windows 应用程序，在默认窗体中添加一个 ComboBox 控件和一个 Label 控件，其中，ComboBox 控件用来显示并选择职位，Label 控件用来显示选择的职位。代码如下。

```
01 private void Form1_Load(object sender, EventArgs e)
02 {
03      // 设置 comboBox1 的下拉框样式
04     comboBox1.DropDownStyle = ComboBoxStyle.DropDownList;
05      string[] str = new string[] { "总经理", "副总经理", "人事部经理", "财务部经理", "部门经理", "普通员工" };              // 定义职位数组
06       comboBox1.DataSource = str;                          // 指定 comboBox1 的数据源
07       comboBox1.SelectedIndex = 0;                         // 指定默认选择第一项
08  }
09  // 触发 comboBox1 的选择项更改事件
10  private void comboBox1_SelectedIndexChanged(object sender, EventArgs e)
11  {
12       // 获取 comboBox1 中的选中项
13       label2.Text = "您选择的职位为：" + comboBox1.SelectedItem;
14 }
```

运行上面代码，得到如图 9.12 所示的内容。

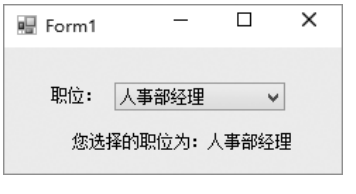


图 9.12　使用 ComboBox 控件选择职位

9.3.8　ListBox 控件

ListBox 控件又称为列表控件，它主要用于显示一个列表，用户可以从中选择一项或

多项，如果选项总数超出可以显示的项数，则控件会自动添加滚动条。

下面详细介绍 ListBox 控件的常见用法。

1. 在 ListBox 控件中添加和移除项

通过使用 ListBox 控件的 Items 属性的 Add 方法，可以向 ListBox 控件中添加项。通过使用 ListBox 控件的 Items 属性的 Remove 方法，可以将 ListBox 控件中选中的项移除。

例如，通过使用 ListBox 控件的 Items 属性的 Add 方法和 Remove 方法，实现向控件中添加及移除项，代码如下。

```
01  listBox1.Items.Add(" 品牌电脑 ");                    // 添加项
02  listBox1.Items.Add("iPhone  6");
03  listBox1.Items.Add(" 引擎耳机 ");
04  listBox1.Items.Add(" 充电宝 ");
05  listBox1.Items.Remove(" 引擎耳机 ");                 // 移除项
```

添加和移除项效果如图 9.13 所示。

2. 创建总显示滚动条的列表控件

通过设置 ListBox 控件的 HorizontalScrollbar 属性和 ScrollAlwaysVisible 属性可以使列表框总显示滚动条。如果将 HorizontalScrollbar 属性设置为 true，则显示水平滚动条；如果将 ScrollAlwaysVisible 属性设置为 true，则始终显示垂直滚动条。

例如，将 ListBox 控件的 HorizontalScrollbar 属性和 ScrollAlwaysVisible 属性都设置为 true，使其显示水平和垂直方向的滚动条，代码如下。

```
01  // 将 HorizontalScrollbar 属性设置为 true，使其能显示水平方向的滚动条
02  listBox1.HorizontalScrollbar = true;
03  // 将 ScrollAlwaysVisible 属性设置为 true，使其能显示垂直方向的滚动条
04  listBox1.ScrollAlwaysVisible = true;
```

控件总显示滚动条效果如图 9.14 所示。

图 9.13　添加和移除项效果

图 9.14　控件总显示滚动条效果

3. 在 ListBox 控件中选择多项

通过设置 SelectionMode 属性的值可以实现在 ListBox 控件中选择多项。SelectionMode

属性的属性值是 SelectionMode 枚举值之一，默认为 SelectionMode.One。SelectionMode 枚举成员及说明如表 9.3 所示。

表 9.3　SelectionMode 枚举成员及说明

枚 举 成 员	说　　明
MultiExtended	可以选择多项，并且用户可使用 Shift 键、Ctrl 键和箭头键来进行选择
MultiSimple	可以选择多项
None	无法选择项
One	只能选择一项

例如，通过设置 ListBox 控件的 SelectionMode 属性值为 SelectionMode 枚举成员 MultiExtended，实现在控件中可以选择多项，用户可使用 Shift 键、Ctrl 键和箭头键来进行选择，代码如下。

```
01  // 设置 SelectionMode 属性值为 SelectionMode 枚举成员 MultiExtended，实现在控件中可以选择多项
02  listBox1.SelectionMode = SelectionMode.MultiExtended;
```

设置列表多选效果如图 9.15 所示。

图 9.15　设置列表多选效果

9.3.9　GroupBox 控件

GroupBox 控件又称为分组框控件，它主要为其他控件提供分组，并且按照控件的分组来细分窗体的功能，其在所包含的控件集周围总是显示边框，而且可以显示标题，但是没有滚动条。

GroupBox 控件最常用的是 Text 属性，它用来设置分组框的标题。例如，下面代码用来为 GroupBox 控件设置标题“系统登录”，代码如下。

```
groupBox1.Text = " 系统登录 ";                    // 设置 groupBox1 控件的标题
```

9.3.10　ListView 控件

ListView 控件又称为列表视图控件，它主要用于显示带图标的项列表，其中可以显示

大图标、小图标和数据。使用 ListView 控件可以创建类似 Windows 资源管理器右边窗口的用户界面。

1. 在 ListView 控件中添加项

在向 ListView 控件中添加项时需要使用其 Items 属性的 Add 方法，该方法主要用于将项添加至项的集合中，其语法格式如下。

```
public virtual ListViewItem Add (string text)
```

text：项的文本。

返回值：已添加到集合中的 ListViewItem。

例如，通过使用 ListView 控件的 Items 属性的 Add 方法向控件中添加项，代码如下。

```
listView1.Items.Add(textBox1.Text.Trim());
```

2. 在 ListView 控件中移除项

在移除 ListView 控件中的项时可以使用其 Items 属性的 RemoveAt 方法或 Clear 方法，其中，RemoveAt 方法用于移除指定的项，而 Clear 方法用于移除列表中的所有项。

RemoveAt 方法用于移除集合中指定索引处的项，其语法格式如下。

```
public virtual void RemoveAt (int index)
```

index：从零开始的索引（属于要移除的项）。

例如，调用 ListView 控件的 Items 属性的 RemoveAt 方法移除选中的项，代码如下。

```
listView1.Items.RemoveAt(listView1.SelectedItems[0].Index);
```

Clear 方法用于从集合中移除所有项，语法格式如下。

```
public virtual void Clear ()
```

例如，调用 Clear 方法清空所有的项，代码如下。

```
listView1.Items.Clear();                    // 使用 Clear 方法移除所有项
```

3. 选择 ListView 控件中的项

在选择 ListView 控件中的项时可以使用其 Selected 属性，该属性主要用于获取或设置一个值，该值指示是否选定此项，其语法格式如下。

```
public bool Selected { get; set; }
```

属性值：如果选定此项，则为 true；否则为 false。

例如，将 ListView 控件中的第 3 项的 Selected 属性设置为 true，即设置为选中第 3 项，

代码如下。

```
listView1.Items[2].Selected = true;      // 使用 Selected 属性选中第 3 项
```

4. 为 ListView 控件中的项添加图标

如果要为 ListView 控件中的项添加图标，则需要使用 ImageList 控件设置 ListView 控件中项的图标。ListView 控件可显示 3 个图像列表中的图标，其中，List 视图、Details 视图和 SmallIcon 视图显示 SmallImageList 属性中指定的图像列表里的图像；LargeIcon 视图显示 LargeImageList 属性中指定的图像列表里的图像；列表视图在大图标或小图标旁显示 StateImageList 属性中设置的一组附加图标。实现的步骤如下。

步骤 1，将相应的属性（SmallImageList 属性、LargeImageList 属性或 StateImageList 属性）设置为想要使用的现有 ImageList 控件。

步骤 2，为每个具有关联图标的列表项设置 ImageIndex 属性或 StateImageIndex 属性，这些属性可以在代码中设置，也可以在“ListViewItem 集合编辑器”中进行设置。若要在“ListViewItem 集合编辑器”中进行设置，则可在“属性”窗口中单击 Items 属性旁的省略号按钮。

例如，设置 ListView 控件的 LargeImageList 属性和 SmallImageList 属性为 imageList1 控件，并设置 ListView 控件中的前两项的 ImageIndex 属性分别为 0 和 1，代码如下。

```
01  listView1.LargeImageList = imageList1;// 设置控件的 LargeImageList 属性
02  listView1.SmallImageList = imageList1;// 设置控件的 SmallImageList 属性
03  listView1.Items[0].ImageIndex = 0;     // 控件中第一项的图标索引为 0
04  listView1.Items[1].ImageIndex = 1;     // 控件中第二项的图标索引为 1
```

5. 在 ListView 控件中启用平铺视图功能

通过启用 ListView 控件的平铺视图功能，可以在图形信息和文本信息之间提供一种视觉平衡。在 ListView 控件中，平铺视图功能与分组功能（或插入标记功能）结合使用。如果要启用平铺视图功能，则需要将 ListView 控件的 View 属性设置为 Tile；另外，还可以通过设置 TileSize 属性来调整平铺的大小。

6. 为 ListView 控件中的项分组

利用 ListView 控件的分组功能可以用分组形式显示相关项目组。在显示时，这些组由包含组标题的水平组标头分隔。可以使用 ListView 控件按字母顺序、日期或任何其他逻辑组合对项进行分组，从而简化大型列表的导航。若要启用分组，则必须首先在设计器中或以编程方式创建一个或多个组，然后才可向组中分配 ListView 项；另外，还可以用编程方式将一个组中的项移至另一个组中。下面介绍为 ListView 控件中的项分组的步骤。

步骤 1，添加组。

使用 Groups 集合的 Add 方法可以向 ListView 控件中添加组，该方法用于将指定的

ListViewGroup 添加到集合中，其语法格式如下。

```
public int Add (ListViewGroup group)
```

group：要添加到集合中的 ListViewGroup。

返回值：该组在集合中的索引；如果集合中已存在该组，则为 -1。

例如，使用 Groups 集合的 Add 方法向控件 listView1 中添加一个分组，标题为“测试”，排列方式为左对齐，代码如下。

```
listView1.Groups.Add(new ListViewGroup("测试", _HorizontalAlignment.Left));
```

步骤 2，移除组。

使用 Groups 集合的 RemoveAt 方法或 Clear 方法可以移除指定的组或移除所有的组。

RemoveAt 方法用来移除集合中指定索引位置的组，其语法格式如下。

```
public void RemoveAt (int index)
```

index：要移除的 ListViewGroup 在集合中的索引。

Clear 方法用于从集合中移除所有组，其语法格式如下。

```
public void Clear ()
```

例如，使用 Groups 集合的 RemoveAt 方法移除索引为 1 的组，使用 Clear 方法移除所有的组，代码如下。

```
01  listView1.Groups.RemoveAt(1);                    // 移除索引为 1 的组
02  listView1.Groups.Clear();                        // 使用 Clear 方法移除所有的组
```

步骤 3，向组分配项或在组之间移动项。

通过设置 ListView 控件中各个项的 System.Windows.Forms.ListViewItem.Group 属性，可以向组分配项或在组之间移动项。

例如，将 ListView 控件的第一项分配到第一个组中，代码如下。

```
listView1.Items[0].Group = listView1.Groups[0];
```

ListView 控件中的项分组示例效果如图 9.16 所示。

学习笔记

ListView 是一种列表控件，在实现诸如显示文件详细信息这样的功能时，推荐使用该控件；另外，由于 ListView 有多种显示样式，因此在实现类似 Windows 系统的“缩略图”“平铺”“图标”“列表”“详细信息”等功能时，经常需要使用 ListView 控件。

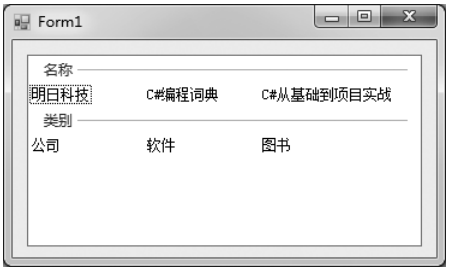


图 9.16　ListView 控件中的项分组示例效果

9.3.11　TreeView 控件

TreeView 控件又称为树控件，它可以为用户显示节点层次结构，而每个节点又可以包含子节点，包含子节点的节点称为父节点，其效果就像在 Windows 操作系统的 Windows 资源管理器功能的左窗口中显示文件和文件夹一样。

学习笔记

TreeView 控件经常用来设计导航菜单。

1. 添加和删除树节点

在向 TreeView 控件中添加树节点时，需要用到其 Nodes 属性的 Add 方法，该方法的语法格式如下。

```
public virtual int Add (TreeNode node)
```

node：要添加到集合中的树节点。

返回值：添加到树节点集合中的树节点从零开始的索引值。

例如，使用 TreeView 控件的 Nodes 属性的 Add 方法向 TreeView 控件中添加两个树节点，代码如下。

```
01  treeView1.Nodes.Add(" 名称 ");
02  treeView1.Nodes.Add(" 类别 ");
```

从 TreeView 控件中移除指定的树节点时，需要使用其 Nodes 属性的 Remove 方法，该方法的语法格式如下。

```
public void Remove (TreeNode node)
```

node：要移除的树节点。

例如，通过 TreeView 控件的 Nodes 属性的 Remove 方法删除选中的子节点，代码如下。

```
treeView1.Nodes.Remove(treeView1.SelectedNode);    // 使用 Remove 方法移除所选项
```

学习笔记

SelectedNode 属性用来获取 TreeView 控件的选中节点。

2. 获取 TreeView 控件中选中的节点

要获取 TreeView 控件中选中的节点，可以在该控件的 AfterSelect 事件中使用 EventArgs 对象返回对已选中节点对象的引用，其中，通过检查 TreeViewEventArgs 类（它包含与事件有关的数据）判断单击了哪个节点。

例如，在 TreeView 控件的 AfterSelect 事件中获取 TreeView 控件中选中节点显示的文本，代码如下。

```
01 private void treeView1_AfterSelect(object sender, TreeViewEventArgs e)
02 {
03     label1.Text = "当前选中的节点：" + e.Node.Text;  // 获取选中节点显示的文本
04 }
```

3. 为 TreeView 控件中的节点设置图标

TreeView 控件可以在每个节点紧挨节点文本的左侧显示图标，但在显示时，必须使 TreeView 控件与 ImageList 控件相关联。为 TreeView 控件中的节点设置图标的步骤如下。

步骤 1，将 TreeView 控件的 ImageList 属性设置为想要使用的现有 ImageList 控件，该属性既可以在设计器中使用“属性”窗口进行设置，也可以在代码中进行设置。

例如，设置 treeView1 控件的 ImageList 属性为 imageList1，代码如下。

```
treeView1.ImageList = imageList1;
```

步骤 2，设置树节点的 ImageIndex 属性和 SelectedImageIndex 属性，其中，ImageIndex 属性用来确定正常状态下和展开状态下的节点显示图像，而 SelectedImageIndex 属性用来确定选定状态下的节点显示图像。

例如，设置 treeView1 控件的 ImageIndex 属性，确定正常状态下或展开状态下的节点显示图像的索引为 0；设置 SelectedImageIndex 属性，确定选定状态下的节点显示图像的索引为 1。代码如下。

```
01 treeView1.ImageIndex = 0;
02 treeView1.SelectedImageIndex = 1;
```

示例 6. 使用 TreeView 控件显示部门结构

创建一个 Windows 应用程序，在默认窗体中添加一个 TreeView 控件、一个 ImageList 控件和一个 ContextMenuStrip 控件，其中，TreeView 控件用来显示部门结构，ImageList

控件用来存储 TreeView 控件中用到的图片，ContextMenuStrip 控件用来作为 TreeView 控件的快捷菜单。代码如下。

```
01 private void Form1_Load(object sender, EventArgs e)
02 {
03     treeView1.ContextMenuStrip = contextMenuStrip1;// 设置 TreeView 控件的快捷菜单
04     TreeNode TopNode = treeView1.Nodes.Add(" 公司 ");       // 建立一个顶级节点
05     // 建立 4 个基础节点，分别表示 4 个大的部门
06     TreeNode ParentNode1 = new TreeNode(" 人事部 ");
07     TreeNode ParentNode2 = new TreeNode(" 财务部 ");
08     TreeNode ParentNode3 = new TreeNode(" 基础部 ");
09     TreeNode ParentNode4 = new TreeNode(" 软件开发部 ");
10     // 将 4 个基础节点添加到顶级节点中
11     TopNode.Nodes.Add(ParentNode1);
12     TopNode.Nodes.Add(ParentNode2);
13     TopNode.Nodes.Add(ParentNode3);
14     TopNode.Nodes.Add(ParentNode4);
15     // 建立 6 个子节点，分别表示 6 个部门
16     TreeNode ChildNode1 = new TreeNode("C# 部门 ");
17     TreeNode ChildNode2 = new TreeNode("ASP.NET 部门 ");
18     TreeNode ChildNode3 = new TreeNode("VB 部门 ");
19     TreeNode ChildNode4 = new TreeNode("VC 部门 ");
20     TreeNode ChildNode5 = new TreeNode("JAVA 部门 ");
21     TreeNode ChildNode6 = new TreeNode("PHP 部门 ");
22     // 将 6 个子节点添加到对应的基础节点中
23     ParentNode4.Nodes.Add(ChildNode1);
24     ParentNode4.Nodes.Add(ChildNode2);
25     ParentNode4.Nodes.Add(ChildNode3);
26     ParentNode4.Nodes.Add(ChildNode4);
27     ParentNode4.Nodes.Add(ChildNode5);
28     ParentNode4.Nodes.Add(ChildNode6);
29     // 设置 imageList1 控件中显示的图像
30     imageList1.Images.Add(Image.FromFile("1.png"));
31     imageList1.Images.Add(Image.FromFile("2.png"));
32     // 设置 treeView1 的 ImageList 属性为 imageList1
33     treeView1.ImageList = imageList1;
34     imageList1.ImageSize = new Size(16, 16);
35     // 设置 treeView1 控件节点的图标在 imageList1 控件中的索引是 0
36     treeView1.ImageIndex = 0;
37     // 选择某个节点后显示的图标在 imageList1 控件中的索引是 1
38     treeView1.SelectedImageIndex = 1;
39 }
40 private void treeView1_AfterSelect(object sender, TreeViewEventArgs e)
41 {
42     // 在 AfterSelect 事件中获取控件中选中节点显示的文本
43     label1.Text = " 选择的部门：" + e.Node.Text;
```

```
44  }
45  private void 全部展开 ToolStripMenuItem_Click(object sender, EventArgs e)
46  {
47      treeView1.ExpandAll();              // 展开所有树节点
48  }
49  private void 全部折叠 ToolStripMenuItem_Click(object sender, EventArgs e)
50  {
51      treeView1.CollapseAll();            // 折叠所有树节点
52  }
```

使用 TreeView 控件显示部门结构效果如图 9.17 所示。

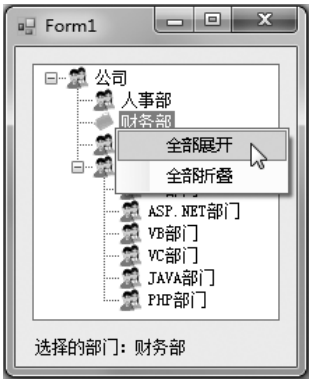


图 9.17　使用 TreeView 控件显示部门结构效果

学习笔记

在实现本示例时，首先需要确保项目的 Debug 文件夹中存在 1.png 和 2.png 这两个文件，这两个文件用来设置 TreeView 控件所显示的图标。

9.3.12　ImageList 组件

ImageList 组件又称为图片存储组件，它主要用于存储图片资源，然后在控件上将图片显示出来，这样就简化了对图片的管理。ImageList 组件的主要属性是 Images，它包含关联控件将要使用的图片。每个单独的图片可以通过其索引值或键值来访问；另外，ImageList 组件中的所有图片都将以同样的尺寸显示，该尺寸由其 ImageSize 属性设置，较大的图片将缩小至适当的尺寸。

ImageList 组件的常用属性及说明如表 9.4 所示。

表 9.4　ImageList 组件的常用属性及说明

属　性	说　明
ColorDepth	获取图像列表的颜色深度
Images	获取此图像列表的 ImageList.ImageCollection
ImageSize	获取或设置图像列表中的图像尺寸
ImageStream	获取与此图像列表关联的 ImageListStreamer

学习笔记

对于一些经常用到图片或图标的控件，经常与 ImageList 组件结合使用。例如，在使用 TreeView 控件和 List Box 控件等时，经常使用 ImageList 组件存储它们需要用到的一些图片或图标，然后在程序中通过 ImageList 组件的索引项来方便地获取需要的图片或图标。

9.3.13　Timer 组件

Timer 组件又称为计时器组件，它可以定期引发事件，时间间隔的长度由其 Interval 属性定义，其属性值以毫秒为单位。若启用了该组件，则每个时间间隔引发一次 Tick 事件，开发人员可以在 Tick 事件中添加要执行操作的代码。

Timer 组件的常用属性及说明如表 9.5 所示。

表 9.5　Timer 组件的常用属性及说明

属　性	说　明
Enabled	获取或设置计时器是否正在运行
Interval	获取或设置相邻两次 Tick 事件的时间间隔（以毫秒为单位）

Timer 组件的常用方法及说明如表 9.6 所示。

表 9.6　Timer 组件的常用方法及说明

方　法	说　明
Start	启动计时器
Stop	停止计时器

Timer 组件的常用事件及说明如表 9.7 所示。

表 9.7　Timer 组件的常用事件及说明

事　件	说　明
Tick	当指定的计时器间隔已过去而且计时器处于启用状态时发生

示例 7．双色球彩票选号器

使用 C# 实现模拟双色球选号的功能，程序开发步骤如下。

步骤 1，创建一个 Windows 应用程序，将其命名为 Double。

步骤 2，在新建项目的默认 Form1 窗体中，首先通过 BackgroundImage 属性设置背景图片；然后添加 7 个 Label 控件，并将它们的 BackColor 属性设置为 Transparent，以便使背景透明，这 7 个 Label 控件分别用来显示红球数字和蓝球数字；添加两个 Button 控件，并设置背景图片；添加一个 Timer 组件，作为计时器。

步骤 3，在两个 Button 控件的 Click 事件中分别使用 Timer 组件的 Start 方法和 Stop 方法启动和停止计时器，代码如下。

```
01 private void button1_Click(object sender, EventArgs e)
02 {
03     timer1.Start();                                          // 启动计时器
04 }
05 private void button2_Click(object sender, EventArgs e)
06 {
07     timer1.Stop();                                           // 停止计时器
08 }
```

步骤 4，触发 Timer 计时器的 Tick 事件，在该事件中通过随机生成器随机生成红球数字和蓝球数字，代码如下。

```
01 private void timer1_Tick(object sender, EventArgs e)
02 {
03     Random rnd = new Random();                               // 生成随机数生成器
04     label1.Text = rnd.Next(1, 33).ToString("00");            // 第 1 个红球数字
05     label2.Text = rnd.Next(1, 33).ToString("00");            // 第 2 个红球数字
06     label3.Text = rnd.Next(1, 33).ToString("00");            // 第 3 个红球数字
07     label4.Text = rnd.Next(1, 33).ToString("00");            // 第 4 个红球数字
08     label5.Text = rnd.Next(1, 33).ToString("00");            // 第 5 个红球数字
09     label6.Text = rnd.Next(1, 33).ToString("00");            // 第 6 个红球数字
10     label7.Text = rnd.Next(1, 16).ToString("00");            // 蓝球数字
11 }
```

运行上面代码，单击“开始”按钮，红球和蓝球同时滚动，单击“停止”按钮，则红球和蓝球停止滚动，当前显示的数字就是选中的号码，如图 9.18 所示。

图 9.18　双色球彩票选号器

9.4　菜单控件、工具栏控件与状态栏控件

除了 9.3 节介绍的常用控件，在开发窗体程序时，还少不了菜单控件、工具栏控件和状态栏控件，本节将对这 3 种控件进行详细讲解。

9.4.1　菜单控件

菜单控件使用 MenuStrip 控件来表示，它主要用来设计程序的菜单栏，C# 中的 MenuStrip 控件支持多文档界面、菜单合并、工具提示和溢出等功能，开发人员可以通过添加访问键、快捷键、选中标记、图像和分隔条来增强菜单的可用性和可读性。

下面以“文件”菜单为例演示如何使用 MenuStrip 控件设计菜单栏，具体步骤如下。

步骤 1，从工具箱中将 MenuStrip 控件拖曳到窗体中，如图 9.19 所示。

步骤 2，在输入菜单名称时，系统会自动产生输入下一个菜单名称的提示，如图 9.20 所示。

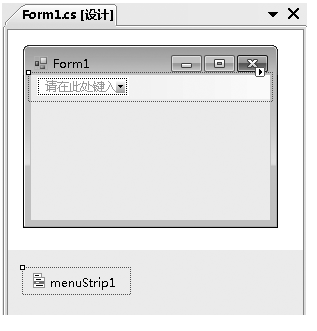


图 9.19　将 MenuStrip 控件拖曳到窗体中

图 9.20　输入菜单名称

步骤 3，在图 9.20 所示的输入框中输入“新建 (&N)”后，菜单中会自动显示“新建 (N)”，

在此处，“&”被识别为确认热键的字符。例如，“新建(N)”菜单就可以通过键盘上的〈Alt+N〉组合键打开。同样，在“新建(N)”菜单下创建“打开(O)”“关闭(C)”“保存(S)”等子菜单，如图 9.21 所示。

步骤 4，菜单设置完成后，运行程序的效果如图 9.22 所示。

图 9.21　添加菜单

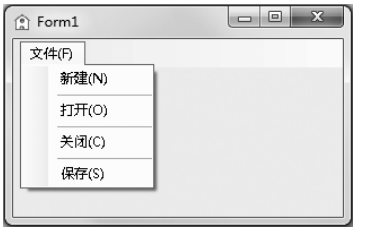


图 9.22　运行程序的效果

9.4.2　工具栏控件

工具栏控件使用 ToolStrip 控件来表示，使用该控件可以创建具有 Windows、Office、IE 或自定义的外观和行为的工具栏及其他用户界面元素，这些元素支持溢出及运行时项的重新排序。

使用 ToolStrip 控件创建工具栏的具体步骤如下。

步骤 1，从工具箱中将 ToolStrip 控件拖曳到窗体中，如图 9.23 所示。

步骤 2，单击工具栏中向下箭头的提示图标，如图 9.24 所示。

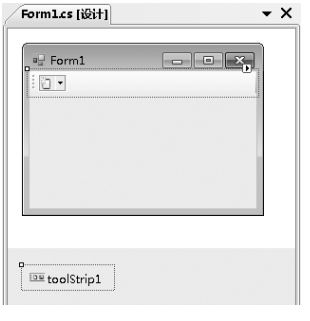


图 9.23　将 ToolStrip 控件拖曳到窗体中

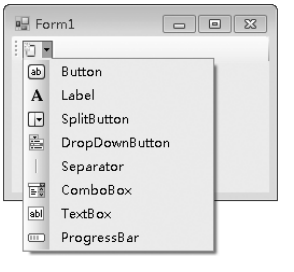


图 9.24　添加工具栏项目

从图 9.24 中可以看到，当单击工具栏中向下的箭头时，在下拉菜单中有 8 种不同的类型，下面分别对其进行介绍。

- Button：包含文本和图像中可让用户选择的项。
- Label：包含文本和图像的项，不可以让用户选择，可以显示超链接。
- SplitButton：在 Button 的基础上增加了一个下拉菜单。
- DropDownButton：用于下拉菜单选择项。
- Separator：分隔符。
- ComboBox：显示一个 ComboBox 的项。
- TextBox：显示一个 TextBox 的项。
- ProgressBar：显示一个 ProgressBar 的项。

步骤 3，添加相应的工具栏按钮后，可以设置其要显示的图像，具体方法是，选中要设置图像的工具栏按钮，单击鼠标右键，在弹出的快捷菜单中选择“设置图像”选项，如图 9.25 所示。

步骤 4，工具栏中的按钮默认只显示图像，如果要以其他方式（如只显示文本、同时显示图像和文本等）显示工具栏按钮，则可以选中工具栏按钮，单击鼠标右键，在弹出的快捷菜单中选择“DisplayStyle”菜单项下面的各个子菜单项。

步骤 5，工具栏设计完成后，运行程序，工具栏效果如图 9.26 所示。

图 9.25　选择“设置图像”选项

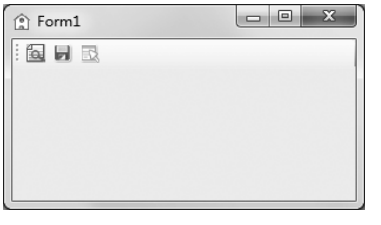


图 9.26　工具栏效果

9.4.3　状态栏控件

状态栏控件使用 StatusStrip 控件来表示，它通常放置在窗体的底部，用于显示窗体上一

些对象的相关信息，或者显示应用程序的信息。StatusStrip 控件由 ToolStripStatusLabel 对象组成，每个这样的对象都可以显示文本、图像或同时显示这二者。另外，StatusStrip 控件还可以包含 ToolStripDropDownButton、ToolStripSplitButton 和 ToolStripProgressBar 等控件。

示例 8．在状态栏中显示登录用户和当前时间

修改示例 2，在示例 2 的基础上再添加一个 Windows 窗体，用来作为登录进去后的主窗体，在该窗体中使用 StatusStrip 控件设计状态栏，并在其中显示登录用户及登录时间，具体步骤如下。

步骤 1，从工具箱中将 StatusStrip 控件拖曳到窗体中，如图 9.27 所示。

步骤 2，单击状态栏中的向下箭头的提示图标，选择“插入”选项，弹出子菜单，如图 9.28 所示。

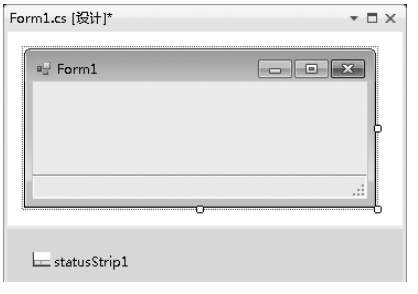


图 9.27　将 StatusStrip 控件拖曳到窗体中

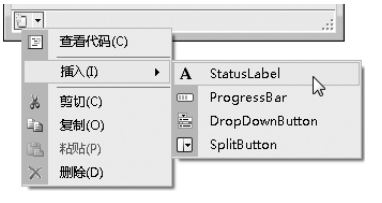


图 9.28　添加状态栏项目

从图 9.28 中可以看到，当单击“插入”选项时，在下拉子菜单中有 4 种不同的类型，下面分别对其进行介绍。

- StatusLabel：包含文本和图像的项，不可以让用户选择，可以显示超链接。
- ProgressBar：进度条显示。
- DropDownButton：用于下拉菜单选择项。
- SplitButton：在 Button 的基础上增加了一个下拉菜单。

步骤 3，在如图 9.28 所示的状态栏中选择需要的项将其添加到状态栏中，这里添加两个 StatusLabel，状态栏设计效果如图 9.29 所示。

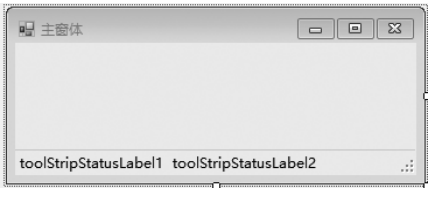


图 9.29　状态栏设计效果

步骤4，打开登录窗体（Form1），在其.cs文件中定义一个成员变量用来记录登录用户名，代码如下。

```
public static string strName;                    // 声明成员变量，用来记录登录用户名
```

步骤 5，触发登录窗体中“登录”按钮的 Click 事件，在该事件中记录登录用户名，并打开主窗体，代码如下。

```
01  private void button1_Click(object sender, EventArgs e)
02  {
03      strName = textBox1.Text;                          // 记录登录用户
04      Form2 frm = new Form2();                          // 创建 Form2 窗体对象
05      this.Hide();                                      // 隐藏当前窗体
06      frm.Show();                                       // 显示 Form2 窗体
07  }
```

步骤 6，触发 Form2 窗体的 Load 事件，在该事件中，在状态栏中显示登录用户及登录时间，代码如下。

```
01  private void Form2_Load(object sender, EventArgs e)
02  {
03      toolStripStatusLabel1.Text = "登录用户：" + Form1.strName;// 显示登录用户
04      // 显示登录时间
05      toolStripStatusLabel2.Text = "  ||  登录时间：" + DateTime.Now.ToLong
TimeString();
06  }
```

在登录窗体中输入用户名和密码，如图 9.30 所示。单击“登录”按钮，进入主窗体，在主窗体的状态栏中会显示登录用户及登录时间，如图 9.31 所示。

图 9.30　输入用户名和密码

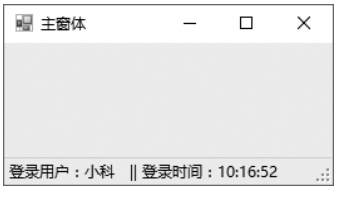


图 9.31　显示登录用户及登录时间

9.5　对话框

如果一个窗体的弹出是为了对诸如打开文件之类的用户请求做出响应，同时停止所有其他“用户与应用程序之间”的交互活动，那么它就是一个对话框。比较常用的对话框操作（如打开文件、保存文件等）都是通过 Windows 提供的标准对话框实现的，C# 也可以

利用这些对话框来实现相应的功能。

对话框控件主要包括打开对话框控件、另存为对话框控件、浏览文件夹对话框控件等。

9.5.1 消息对话框

消息对话框是一个预定义对话框，主要用于向用户显示与应用程序相关的信息，以及来自用户的请求信息。在 .NET 中，使用 MessageBox 类表示消息对话框，通过调用该类的 Show 方法可以显示消息对话框。Show 方法有多种重载形式，其最常用的两种形式如下。

```
public static DialogResult Show(string text)
public static DialogResult Show(string text,string caption,
MessageBoxButtons buttons,MessageBoxIcon icon)
```

text：要在消息框中显示的文本。

caption：要在消息框的标题栏中显示的文本。

buttons：MessageBoxButtons 枚举值之一，可指定在消息框中显示哪些按钮。MessageBoxButtons 枚举值及说明如表 9.8 所示。

表 9.8 MessageBoxButtons 枚举值及说明

枚 举 值	说 明
OK	消息框包含“确定”按钮
OKCancel	消息框包含“确定”和“取消”按钮
AbortRetryIgnore	消息框包含“中止”、“重试”和“忽略”按钮
YesNoCancel	消息框包含“是”、“否”和“取消”按钮
YesNo	消息框包含“是”和“否”按钮
RetryCancel	消息框包含“重试”和“取消”按钮

icon：MessageBoxIcon 枚举值之一，它指定在消息框中显示哪个图标。MessageBoxIcon 枚举值及说明如表 9.9 所示。

表 9.9 MessageBoxIcon 枚举值及说明

枚 举 值	说 明
None	消息框未包含符号
Hand	消息框包含一个符号，该符号是由一个红色背景的圆圈及其中的白色 X 组成的
Question	消息框包含一个符号，该符号是由一个圆圈和其中的一个问号组成的
Exclamation	消息框包含一个符号，该符号是由一个黄色背景的三角形及其中的一个感叹号组成的
Asterisk	消息框包含一个符号，该符号是由一个圆圈及其中的小写字母 i 组成的

续表

枚　举　值	说　　明
Stop	消息框包含一个符号，该符号是由一个红色背景的圆圈及其中的白色 X 组成的
Error	消息框包含一个符号，该符号是由一个红色背景的圆圈及其中的白色 X 组成的
Warning	消息框包含一个符号，该符号是由一个黄色背景的三角形及其中的一个感叹号组成的
Information	消息框包含一个符号，该符号是由一个圆圈及其中的小写字母 i 组成的

返回值：DialogResult 枚举值之一。DialogResult 枚举值及说明如表 9.10 所示。

表 9.10　DialogResult 枚举值及说明

枚　举　值	说　　明
None	从对话框返回了 Nothing，这表明有模式对话框继续运行
OK	对话框的返回值是 OK（通常从标签为“确定”的按钮发送）
Cancel	对话框的返回值是 Cancel（通常从标签为“取消”的按钮发送）
Abort	对话框的返回值是 Abort（通常从标签为“中止”的按钮发送）
Retry	对话框的返回值是 Retry（通常从标签为“重试”的按钮发送）
Ignore	对话框的返回值是 Ignore（通常从标签为“忽略”的按钮发送）
Yes	对话框的返回值是 Yes（通常从标签为“是”的按钮发送）
No	对话框的返回值是 No（通常从标签为“否”的按钮发送）

例如，使用 MessageBox 类的 Show 方法弹出一个“警告”消息框，代码如下。

```
MessageBox.Show("确定要退出当前系统吗？", "警告", MessageBoxButtons.YesNo,
MessageBoxIcon.Warning);
```

显示“警告”消息框效果如图 9.32 所示。

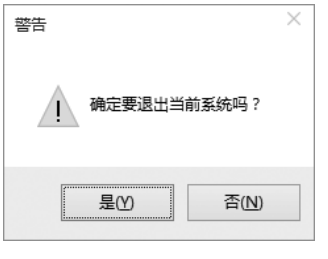


图 9.32　显示“警告”消息框效果

9.5.2　打开对话框控件

打开对话框控件使用 OpenFileDialog 控件表示。OpenFileDialog 控件表示一个通用

对话框，用户可以使用此对话框来指定一个或多个要打开的文件的文件名。打开对话框如图 9.33 所示。

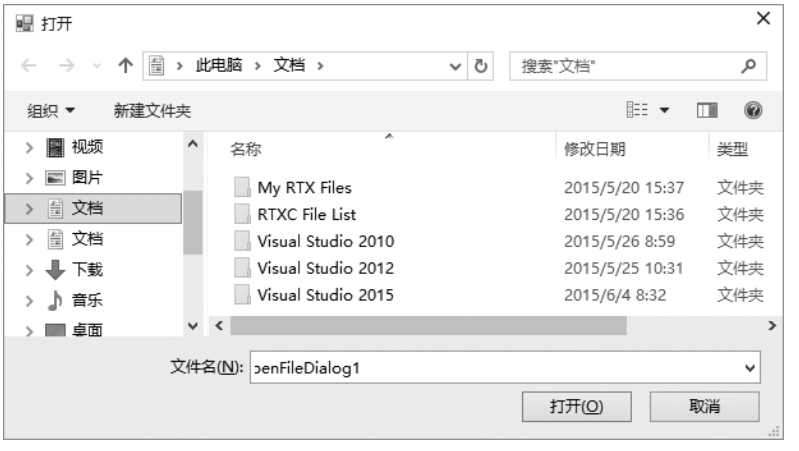


图 9.33　打开对话框

OpenFileDialog 控件常用属性及说明如表 9.11 所示。

表 9.11　OpenFileDialog 控件常用属性及说明

属　性	说　明
AddExtension	指示如果用户省略扩展名，对话框是否自动在文件名中添加扩展名
DefaultExt	获取或设置默认文件扩展名
FileName	获取或设置一个包含在文件对话框中选定的文件名的字符串
FileNames	获取对话框中所有选定文件的文件名
Filter	获取或设置当前文件名筛选器字符串，该字符串决定对话框的“另存为文件类型”或“文件类型”框中出现的选择内容
InitialDirectory	获取或设置文件对话框显示的初始目录
Multiselect	获取或设置一个值，该值指示对话框是否允许选择多个文件
RestoreDirectory	获取或设置一个值，该值指示对话框在关闭前是否还原当前目录

OpenFileDialog 控件常用方法及说明如表 9.12 所示。

表 9.12　OpenFileDialog 控件常用方法及说明

方　法	说　明
OpenFile	此方法以只读模式打开用户选择的文件
ShowDialog	打开“打开文件”对话框

学习笔记

ShowDialog 方法是对话框的通用方法，用来打开相应的对话框。

例如，使用 OpenFileDialog 打开一个“打开文件”对话框，该对话框中只能选择图片文件，代码如下。

```
01   openFileDialog1.InitialDirectory = "C:\\";          // 设置初始目录
02   // 设置只能选择图片文件
03   openFileDialog1.Filter = "bmp 文件 (*.bmp)|*.bmp|gif 文件 (*.gif)|*.gif|jpg
文件 (*.jpg)|*.jpg";
04   openFileDialog1.ShowDialog();
```

9.5.3　另存为对话框控件

另存为对话框控件使用 SaveFileDialog 控件表示。SaveFileDialog 控件表示一个通用对话框，用户可以使用此对话框来指定一个要将文件另存为的文件名。另存为对话框如图 9.34 所示。

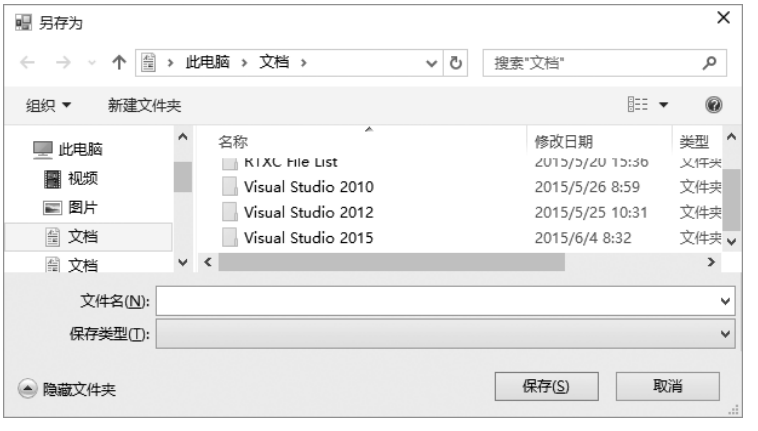


图 9.34　另存为对话框

SaveFileDialog 控件常用属性及说明如表 9.13 所示。

表 9.13　SaveFileDialog 控件常用属性及说明

属　性	说　明
FileName	获取或设置一个包含在文件对话框中选定的文件名的字符串
FileNames	获取对话框中所有选定文件的文件名
Filter	获取或设置当前文件名筛选器字符串，该字符串决定对话框的“另存为文件类型”或“文件类型”框中出现的选择内容

例如，使用 SaveFileDialog 控件来调用一个选择文件路径的对话框窗体，代码如下。

```
saveFileDialog1.ShowDialog();
```

例如，在保存对话框中设置保存文件的类型为 .txt，代码如下。

```
saveFileDialog1.Filter = "文本文件（*.txt）|*.txt";
```

例如，获取在保存对话框中设置文件的路径全名，代码如下。

```
string strName = saveFileDialog1.FileName;
```

9.5.4 浏览文件夹对话框控件

浏览文件夹对话框控件使用 FolderBrowserDialog 控件表示。FolderBrowserDialog 控件主要用来提示用户选择文件夹。浏览文件夹对话框如图 9.35 所示。

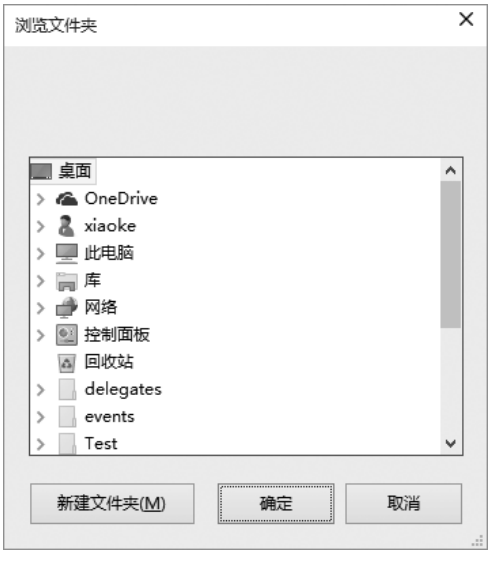


图 9.35 浏览文件夹对话框

FolderBrowserDialog 控件常用属性及描述如表 9.14 所示。

表 9.14 FolderBrowserDialog 控件常用属性及描述

属　性	说　明
Description	获取或设置对话框中在 TreeView 控件上显示的说明文本
RootFolder	获取或设置从其开始浏览的根文件夹
SelectedPath	获取或设置用户选定的路径
ShowNewFolderButton	获取或设置一个值，该值指示“新建文件夹”按钮是否显示在浏览文件夹对话框中

例如，设置在弹出的“浏览文件夹”对话框中不显示“新建文件夹”按钮，然后判断是否选择了文件夹，如果已经选择，则将选择的文件夹显示在 TextBox 文本框中，代码如下。

```
01 folderBrowserDialog1.ShowNewFolderButton = false; // 不显示新建文件夹按钮
02 if (folderBrowserDialog1.ShowDialog() == DialogResult.OK)// 判断是否选择了文件夹
03 {
04     textBox1.Text = folderBrowserDialog1.SelectedPath;// 显示选择的文件夹名称
05 }
```

第 10 章　数据访问技术

学习 C#，必然要学习 ADO.NET，因为使用 ADO.NET 可以非常方便地操作各种主流数据库。大部分应用程序都是使用数据库存储数据的，通过使用 ADO.NET 技术，既可以根据指定条件查询数据库中的数据，又可以对数据库中的数据进行增加、删除、修改等操作。本章将详细讲解 ADO.NET，以及由 ADO.NET 衍生的 Entity Framework 技术。

10.1　ADO.NET 概述

ADO.NET 是微软 .NET 数据库的访问架构，它是数据库应用程序和数据源之间沟通的桥梁，主要提供一个面向对象的数据访问架构，并用该架构来开发数据库应用程序。

10.1.1　ADO.NET 对象模型

为了更好地理解 ADO.NET 架构模型的各个组成部分，这里对 ADO.NET 中的相关对象进行图示理解，图 10.1 所示为 ADO.NET 对象模型。

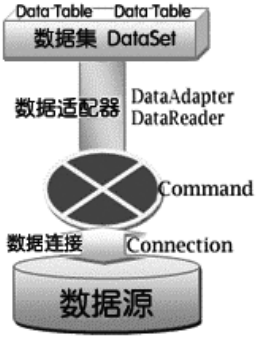


图 10.1　ADO.NET 对象模型

ADO.NET 主要包括 Connection、Command、DataReader、DataAdapter、DataSet 和

DataTable 6 个对象，下面分别对其进行介绍。

（1）Connection 对象主要提供与数据库的连接功能。

（2）Command 对象用于返回数据、修改数据、运行存储过程，以及发送或检索参数信息的数据库命令。

（3）DataReader 对象通过 Command 对象提供从数据库中检索信息的功能，它以一种只读的、向前的、快速的方式访问数据库。

（4）DataAdapter 对象提供连接 DataSet 对象和数据源的功能，它主要使用 Command 对象在数据源中执行 SQL 命令，以便将数据加载到 DataSet 数据集中，并确保 DataSet 数据集中数据的更改与数据源保持一致。

（5）DataSet 对象是 ADO.NET 的核心概念，它是支持 ADO.NET 断开式、分布式数据方案的核心对象。DataSet 对象是一个数据库容器，可以把它当作存在于内存中的数据库，无论数据源是什么，它都会提供一致的关系编程模型。

（6）DataTable 对象表示内存中数据的一个表。

使用 ADO.NET 操作数据库的主要步骤如图 10.2 所示。

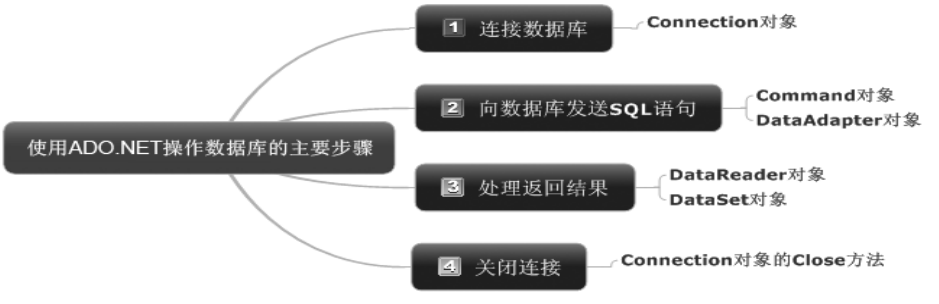


图 10.2　使用 ADO.NET 操作数据库的主要步骤

10.1.2　数据访问命名空间

在 .NET 中，用于数据访问的命名空间如下。

（1）System.Data：提供对 ADO.NET 结构的类的访问。通过 ADO.NET 可以生成一些组件，用于有效管理多个数据源的数据。

（2）System.Data.Common：包含由各种 .NET Framework 数据提供程序共享的类。

（3）System.Data.Odbc：ODBC.NET Framework 数据提供程序，描述用于访问托管空间中的 ODBC 数据源的类集合。

（4）System.Data.OleDb：OLE DB.NET Framework 数据提供程序，描述用于访问托管空间中的 OLE DB 数据源的类集合。

（5）System.Data.SqlClient：SQL Server.NET Framework 数据提供程序，描述用于在托管空间中访问 SQL Server 数据库的类集合。

（6）System.Data.SqlTypes：提供 SQL Server 中本机数据类型的类，SqlTypes 中的每个数据类型在 SQL Server 中具有其等效的数据类型。

（7）System.Data.OracleClient：Oracle.NET Framework 数据提供程序，描述用于在托管空间中访问 Oracle 数据源的类集合。

10.2 Connection 对象

所有对数据库的访问操作都是从建立数据库连接开始的。在打开数据库之前，必须先设置好连接字符串（ConnectionString），再调用 Open 方法打开连接，此时便可对数据库进行访问，最后调用 Close 方法关闭连接。

10.2.1 熟悉 Connection 对象

Connection 对象用于连接到数据库和管理数据库的事务，它的一些属性描述数据源和用户身份验证。Connection 对象还提供一些方法允许程序员与数据源建立连接或断开连接，并且微软公司提供了 4 种数据提供程序的连接对象，分别为：

（1）SQL Server.NET Framework 数据提供程序的 SqlConnection 连接对象，命名空间为 System.Data.SqlClient。

（2）OLE DB.NET Framework 数据提供程序的 OleDbConnection 连接对象，命名空间为 System.Data.OleDb。

（3）ODBC.NET Framework 数据提供程序的 OdbcConnection 连接对象，命名空间为 System.Data.Odbc。

（4）Oracle.NET Framework 数据提供程序的 OracleConnection 连接对象，命名空间为 System.Data.OracleClient。

学习笔记

本章所涉及的关于 ADO.NET 的所有示例都将以 SQL Server 数据库为例，引入的命名空间为 System.Data.SqlClient。

10.2.2　数据库连接字符串

为了让连接对象知道将要访问的数据库文件在哪里，用户必须将这些信息用一个字符串加以描述。数据库连接字符串中需要提供的必要信息包括服务器名称、数据库名称和数据库的身份验证方式（Windows 集成身份验证或 SQL Server 身份验证）；另外，还可以指定其他信息（如连接超时等）。

数据库连接字符串常用的参数及说明如表 10.1 所示。

表 10.1　数据库连接字符串常用的参数及说明

参　数	说　明
Provider	设置或返回连接提供程序的名称，仅用于 OleDbConnection 连接对象
Connection Timeout	在终止尝试并产生异常前，等待连接到服务器的连接时间长度（以秒为单位）。默认值时间长度是 15 秒
Initial Catalog 或 Database	数据库的名称
Data Source 或 Server	连接打开时使用的 SQL Server 服务签名，或者 Access 数据库的文件名
Password 或 pwd	SQL Server 账户的登录密码
User ID 或 uid	SQL Server 登录账户
Integrated Security	此参数决定连接是否为安全连接。可能的值有 True、False 和 SSPI（SSPI 是 True 的同义词）

学习笔记

表 10.1 中列出的数据库连接字符串的参数不区分大小写。例如，uid、UID、Uid、uID、uId 表示的都是登录账户，它们在使用上没有任何分别。

下面分别以连接 SQL Server 数据库和 Access 数据库为例介绍如何定义数据库连接字符串。

1. 连接 SQL Server 数据库

其语法格式如下。

```
string connectionString=“Server=服务器名;User Id=用户;Pwd=密码;DataBase=数据库名称”
```

例如，通过 ADO.NET 技术连接本地 SQL Server 的 db_EMS 数据库，代码如下。

```
string sqlStr = "Server=XIAOKE;User Id=sa;Pwd=;DataBase=db_EMS";
```

2. 连接 Access 数据库

其语法格式如下。

```
string connectionString="provide= 提供者 ; Data Source=Access 文件路径 ";
```

例如，连接 C 盘根目录下的 db_access.mdb 数据库（Access 2003 以下版本），代码如下。

```
string strSQL ="provider = Microsoft.Jet.OLEDB.4.0;Data Source = C:\\db_
access. mdb";
```

例如，连接 C 盘根目录下的 db_access.accdb 数据库（Access 2007 以下版本），代码如下。

```
string strSQL ="provider = Microsoft.ACE.OLEDB.12.0;Data Source = C:\\db_
access. accdb";
```

10.2.3 应用 SqlConnection 连接对象连接数据库

调用 Connection 对象的 Open 方法或 Close 方法可以打开或关闭数据库连接，而且必须在设置好数据库连接字符串后才可以调用 Open 方法，否则 Connection 对象不知道要与哪一个数据库建立连接。

数据库联机资源是有限的，因此应在需要的时候再打开连接，且一旦使用完就应该尽早关闭连接，把资源归还给系统。

示例 1. 使用 SqlConnection 连接对象连接 SQL Server 数据库

创建一个 Windows 应用程序，在默认窗体中添加两个 Label 控件，分别用来显示数据库连接的打开状态和关闭状态，然后在窗体的加载事件中，通过 SqlConnection 连接对象的 State 属性来判断数据库的连接状态，代码如下。

```
01 private void Form1_Load(object sender, EventArgs e)
02 {
03     //创建数据库连接字符串
04     string SqlStr = "Server=XIAOKE;User Id=sa;Pwd=;DataBase=db_EMS";
05     SqlConnection con = new SqlConnection(SqlStr);        //创建数据库连接对象
06     con.Open();                                           //打开数据库连接
07     if (con.State == ConnectionState.Open)                //判断连接是否打开
08     {
09         label1.Text = "SQL Server 数据库连接开启！ ";
10         con.Close();                                      //关闭数据库连接
11     }
12     if (con.State == ConnectionState.Closed)              //判断连接是否关闭
13     {
```

```
14          label2.Text = "SQL Server 数据库连接关闭！";
15      }
16  }
```

学习笔记

在上面的代码中，由于使用了 SqlConnection 类，所以首先需要添加 System.Data.SqlClient 命名空间，下面遇到这种情况时将不再进行说明。

运行上面代码，得到如图 10.3 所示的内容。

图 10.3　使用 SqlConnection 连接对象连接 SQL Server 数据库

10.3　Command 对象

10.3.1　熟悉 Command 对象

使用 Connection 对象与数据源建立连接后，可以使用 Command 对象对数据源执行查询、添加、删除和修改等操作，操作实现的方式可以是使用 SQL 语句，也可以是使用存储过程。根据 .NET Framework 数据提供程序的不同，Command 对象也可以分成 4 种，分别是 SqlCommand、OleDbCommand、OdbcCommand 和 OracleCommand。在实际的编程过程中，应该根据访问的数据源选择相对应的 Command 对象。

Command 对象的常用属性及说明如表 10.2 所示。

表 10.2　Command 对象的常用属性及说明

属　性	说　明
CommandType	获取或设置 Command 对象要执行命令的类型
CommandText	获取或设置要对数据源执行的 SQL 语句、存储过程名或表名
CommandTimeOut	获取或设置在终止对执行命令的尝试并生成错误之前的等待时间
Connection	获取或设置 Command 对象使用的 Connection 对象的名称
Parameters	获取 Command 对象需要使用的参数集合

例如，使用 SqlCommand 对象对 SQL Server 数据库执行查询操作，代码如下。

```
01 // 创建数据库连接对象
02 SqlConnection conn = new SqlConnection("Server=XIAOKE;User Id=sa;Pwd=;
DataBase=db_EMS");
03 SqlCommand comm = new SqlCommand();                    // 创建 SqlCommand 对象
04 comm.Connection = conn;                                // 指定数据库连接对象
05 comm.CommandType = CommandType.Text;                   // 设置要执行的命令类型
06 comm.CommandText = "select * from tb_stock";           // 设置要执行的 SQL 语句
```

Command 对象的常用方法及说明如表 10.3 所示。

表 10.3　Command 对象的常用方法及说明

方　法	说　明
ExecuteNonQuery	用于执行非 SELECT 命令（如 INSERT、DELETE 或 UPDATE 命令）并返回 3 个命令所影响的数据行数；也可以用来执行一些数据定义命令，比如新建、更新、删除数据库对象（如表、索引等）
ExecuteScalar	用于执行 SELECT 查询命令，返回数据中第一行第一列的值，该方法通常用来执行使用了 COUNT 函数（或 SUM 函数）的 SELECT 命令
ExecuteReader	执行 SELECT 命令，并返回一个 DataReader 对象，这个 DataReader 对象是一个只读向前的数据集

学习笔记

表 10.3 中的 3 种方法非常重要，如果要使用 ADO.NET 完成某种数据库操作，那么一定会用到这 3 种方法。这 3 种方法没有优劣之分，只是使用的场合不同，所以一定要清楚它们的返回值类型及使用方法，以便在不同的场合使用合适的方法。

10.3.2　使用 Command 对象操作数据

以操作 SQL Server 数据库为例，在向数据库中添加记录时，首先要创建 SqlConnection 对象连接数据库，然后定义添加数据的 SQL 字符串，最后调用 SqlCommand 对象的 ExecuteNonQuery 方法执行数据的添加操作。

示例 2．向数据表中添加 C# 编程词典价格信息

创建一个 Windows 应用程序，在默认窗体中添加两个 TextBox 控件、一个 Label 控件和一个 Button 控件，其中，TextBox 控件用来输入要添加的信息，Label 控件用来显示添加成功或添加失败的信息，Button 控件用来执行数据添加操作，代码如下。

```
01 private void button1_Click(object sender, EventArgs e)
02 {
03         // 创建数据库连接对象
```

```
04      SqlConnection conn = new SqlConnection("Server=XIAOKE;User Id=sa;Pwd=;
DataBase=db_EMS");
05      string strsql = "insert into tb_PDic(Name,Money) values('" + textBox1.Text +
"'," + Convert.ToDecimal(textBox2.Text) + ")";              // 定义添加数据的 SQL 语句
06      SqlCommand comm = new SqlCommand(strsql, conn); // 创建 SqlCommand 对象
07      if (conn.State == ConnectionState.Closed)           // 判断连接是否关闭
08      {
09          conn.Open();                                    // 打开数据库连接
10      }
11      // 判断 ExecuteNonQuery 方法返回的参数是否大于 0，大于 0 表示添加成功
12      if (Convert.ToInt32(comm.ExecuteNonQuery()) > 0)
13      {
14          label3.Text = "添加成功！";
15      }
16      else
17      {
18          label3.Text = "添加失败！";
19      }
20      conn.Close();                                       // 关闭数据库连接
21  }
```

运行上面代码，得到如图 10.4 所示的内容。

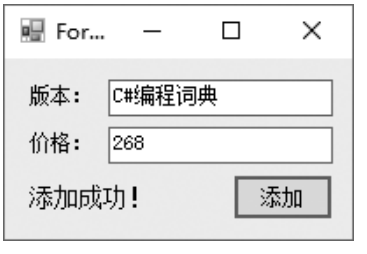


图 10.4　向数据表中添加 C# 编程词典价格信息

10.3.3　使用 Command 对象调用存储过程

存储过程可以使管理数据库和显示数据库信息等操作变得非常容易，它是 SQL 语句和可选控制流语句的预编译集合，它存储在数据库内，在程序中可以通过 Command 对象来调用，其执行速度比 SQL 语句快，同时还保证了数据的安全性和完整性。

示例 3．使用存储过程向数据表中添加 C# 编程词典价格信息

创建一个 Windows 应用程序，在默认窗体中添加两个 TextBox 控件、一个 Label 控件和一个 Button 控件，其中，TextBox 控件用来输入要添加的信息，Label 控件用来显示添加成功或添加失败的信息，Button 控件用来调用存储过程执行数据添加操作，代码如下。

```
01 private void button1_Click(object sender, EventArgs e)
02 {
03     // 创建数据库连接对象
04     SqlConnection sqlcon = new SqlConnection("Server=XIAOKE;User Id=sa;Pwd=;
DataBase=db_EMS");
05     SqlCommand sqlcmd = new SqlCommand();          // 创建 SqlCommand 对象
06     sqlcmd.Connection = sqlcon;                    // 指定数据库连接对象
07     sqlcmd.CommandType = CommandType.StoredProcedure; // 指定执行对象为存储过程
08     sqlcmd.CommandText = "proc_AddData";           // 指定要执行的存储过程名称
09     // 为 @name 参数赋值
10     sqlcmd.Parameters.Add("@name", SqlDbType.VarChar, 20).Value = textBox1.
Text;
11     sqlcmd.Parameters.Add("@money", SqlDbType.Decimal).Value = Convert.ToDecimal
(textBox2.Text);                                   // 为 @money 参数赋值
12     if (sqlcon.State == ConnectionState.Closed)    // 判断连接是否关闭
13     {
14         sqlcon.Open();                             // 打开数据库连接
15     }
16     // 判断 ExecuteNonQuery 方法返回的参数是否大于 0，大于 0 表示添加成功
17     if (Convert.ToInt32(sqlcmd.ExecuteNonQuery()) > 0)
18     {
19         label3.Text = "添加成功！";
20     }
21     else
22     {
23         label3.Text = "添加失败！";
24     }
25     sqlcon.Close();                                // 关闭数据库连接
26 }
```

本示例用到的存储过程代码如下。

```
01 CREATE PROCEDURE [dbo].[proc_AddData]
02 (
03     @name varchar(20),
04     @money decimal
05 )
06 as
07 begin
08     insert into tb_PDic(Name,Money) values(@name,@money)
09 end
10 GO
```

本示例的代码运行结果与示例 2 的代码运行结果相同，如图 10.4 所示。

学习笔记

在 proc_AddData 存储过程中使用了以@开头的两个参数：@name 和 @money。对于存储过程参数名称的定义，通常会参考数据表中列的名称（本示例使用的数据表 tb_PDic 中的列分别为 Name 和 Money），这样可以方便地知道这个参数是套用在哪个列的。当然，参数名称可以自定义，但一般都参考数据表中的列进行定义。

10.4　DataReader 对象

10.4.1　DataReader 对象概述

DataReader 对象是一个简单的数据集，它主要用于从数据源中读取只读的数据集，其常用于检索大量数据。根据 .NET Framework 数据提供程序的不同，DataReader 对象可以分为 SqlDataReader、OleDbDataReader、OdbcDataReader 和 OracleDataReader 四大类。

学习笔记

由于 DataReader 对象每次只能在内存中保留一行，所以使用它的系统开销非常小。

使用 DataReader 对象读取数据时，必须一直保持与数据库的连接，所以读取过程也被称为连线模式，如图 10.5 所示（这里以 SqlDataReader 为例）。

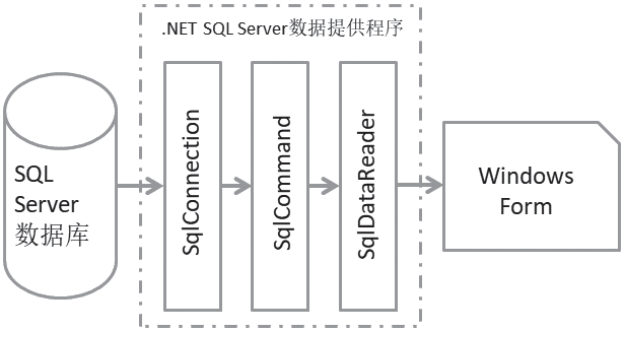


图 10.5　使用 SqlDataReader 对象读取数据

学习笔记

DataReader 对象是一个轻量级的数据对象，如果只需要将数据读出并显示，那么它几乎是最合适的工具，因为它的读取速度比后面要讲解的 DataSet 对象要快，占用的资源也更少；但是，一定要注意，DataReader 对象在读取数据时，要求数据库一直处于连接状态，只有在读取完数据后才能断开连接。

开发人员可以通过 Command 对象的 ExecuteReader 方法从数据源中检索数据来创建 DataReader 对象。DataReader 对象的常用属性及说明如表 10.4 所示。

表 10.4　DataReader 对象的常用属性及说明

属　性	说　明
HasRows	判断数据库中是否有数据
FieldCount	获取当前行的列数
RecordsAffected	获取执行 SQL 语句更改、添加或删除的行数

DataReader 对象的常用方法及说明如表 10.5 所示。

表 10.5　DataReader 对象的常用方法及说明

方　法	说　明
Read	使 DataReader 对象前进到下一条记录
Close	关闭 DataReader 对象
Get	用来读取数据集的当前行的某一列的数据

10.4.2　使用 DataReader 对象读取数据

使用 DataReader 对象读取数据时，首先需要使用其 HasRows 属性判断是否有数据可供读取，如果有数据，则返回 True，否则返回 False；然后使用 DataReader 对象的 Read 方法来循环读取数据表中的数据；最后通过访问 DataReader 对象的列索引来获取读取到的值。例如，sqldr["ID"] 用来获取数据表中 ID 列的值。

示例 4．获取 C# 编程词典信息并分列显示

创建一个 Windows 应用程序，在默认窗体中添加一个 RichTextBox 控件，用来显示使用 SqlDataReader 对象读取到的数据表中的数据，代码如下。

```
01 private void Form1_Load(object sender, EventArgs e)
02 {
03     // 创建数据库连接对象
04     SqlConnection sqlcon = new SqlConnection("Server=XIAOKE;User Id=sa;Pwd=;
DataBase=db_EMS");
05     // 创建 SqlCommand 对象
06     SqlCommand sqlcmd = new SqlCommand("select * from tb_PDic order by ID asc",
sqlcon);
07     if (sqlcon.State == ConnectionState.Closed)          // 判断连接是否关闭
08     {
09         sqlcon.Open();                                   // 打开数据库连接
10     }
11     // 使用 ExecuteReader 方法的返回值创建 SqlDataReader 对象
12     SqlDataReader sqldr = sqlcmd.ExecuteReader();
13     richTextBox1.Text = "编号        版本          价格 \n";  // 为文本框赋初始值
14     try
15     {
16         if (sqldr.HasRows)            // 判断 SqlDataReader 对象中是否有数据
17         {
18             while (sqldr.Read())      // 循环读取 SqlDataReader 对象中的数据
19             {
20                 richTextBox1.Text += "" + sqldr["ID"] + "    " + sqldr["Name"]
+ "    " + sqldr["Money"] + "\n";    // 显示读取的详细信息
21             }
22         }
23     }
24     catch (SqlException ex)                          // 捕获数据库异常
25     {
26         MessageBox.Show(ex.ToString());              // 输出异常信息
27     }
28     finally
29     {
30         sqldr.Close();                               // 关闭 SqlDataReader 对象
31         sqlcon.Close();                              // 关闭数据库连接
32     }
33 }
```

运行上面代码，得到如图 10.6 所示的内容。

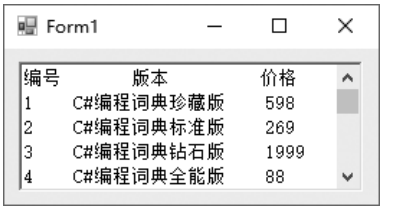


图 10.6　获取 C# 编程词典信息并分列显示

学习笔记

使用 DataReader 对象读取数据之后，务必将其关闭，否则，其所使用的 Connection 对象将无法再执行其他的操作。

10.5 DataSet 对象和 DataAdapter 对象

10.5.1 DataSet 对象

DataSet 对象是 ADO.NET 的核心成员，它是支持 ADO.NET 断开式、分布式数据方案的核心对象，也是实现基于非连接的数据查询的核心组件。DataSet 对象是创建在内存中的集合对象，它可以包含任意数量的数据表及所有表的约束、索引和关系等，它实质上相当于创建在内存中的一个小型关系数据库。一个 DataSet 对象包含一组 DataTable 对象和一组 DataRelation 对象，其中，每组 DataTable 对象都由 DataColumn、DataRow 和 Constraint 集合对象组成，如图 10.7 所示。

对于 DataSet 对象，可以将其看作一个数据库容器，它将数据库中的数据复制了一份放在用户本地的内存中，供用户在不连接数据库的情况下读取数据，以便充分利用客户端资源，降低数据库服务器的压力。

当把 SQL Server 数据库的数据通过起“桥梁”作用的 SqlDataAdapter 对象填充到 DataSet 数据集中后，就可以对数据库进行断开连接、离线状态的操作，如图 10.8 所示。

DataSet 对象的用法主要有以下几种，这些用法可以单独使用，也可以结合使用。

（1）以编程方式在 DataSet 中创建 DataTable、DataRelation 和 Constraint，并使用数据填充表。

（2）通过 DataAdapter 对象用现有关系数据源中的数据表填充 DataSet。

（3）使用 XML 文件加载和保持 DataSet 内容。

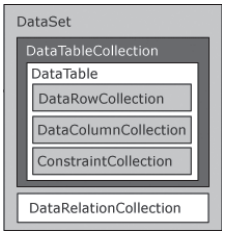


图 10.7　DataSet 对象组成

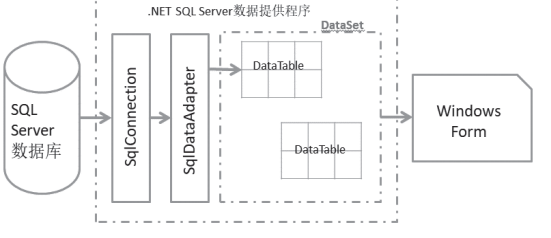


图 10.8　离线模式访问 SQL Server 数据库

10.5.2　DataAdapter 对象

DataAdapter 对象（数据适配器）是一种用来连接 DataSet 对象与实际数据源的对象，可以说有 DataSet 对象的地方就有 DataAdapter 对象，DataAdapter 对象是专门为 DataSet 对象服务的。DataAdapter 对象的工作流程一般有两种：一种是通过 Command 对象执行 SQL 语句，从数据源中检索数据，并将检索到的结果集填充到 DataSet 对象中；另一种是把用户对 DataSet 对象做出的更改写入数据源中。

学习笔记

在 .NET Framework 中使用 4 种 DataAdapter 对象，即 OleDbDataAdapter、SqlDataAdapter、ODBCDataAdapter 和 OracleDataAdapter。其中，OleDbDataAdapter 对象适用于 OLEDB 数据源；SqlDataAdapter 对象适用于 SQL Server 7.0 或更高版本的数据源；ODBCDataAdapter 对象适用于 ODBC 数据源；OracleDataAdapter 对象适用于 Oracle 数据源。

DataAdapter 对象的常用属性及说明如表 10.6 所示。

表 10.6　DataAdapter 对象的常用属性及说明

属　性	说　明
SelectCommand	获取或设置用于在数据源中选择记录的命令
InsertCommand	获取或设置用于将新记录插入到数据源中的命令
UpdateCommand	获取或设置用于更新数据源中的记录的命令
DeleteCommand	获取或设置用于从数据集中删除记录的命令

由于 DataSet 对象是一个非连接的对象，它与数据源无关，也就是说该对象并不能直接与数据源产生联系，而 DataAdapter 对象则正好负责填充它并把它的数据提交给一个特定的数据源。DataAdapter 对象与 DataSet 对象配合使用来执行数据查询、添加、修改和删除等操作。

例如，为 DataAdapter 对象的 SelectCommand 属性赋值，从而实现数据的查询操作，代码如下。

```
01  SqlConnection con = new SqlConnection(strCon);      // 创建数据库连接对象
02  SqlDataAdapter ada = new SqlDataAdapter();          // 创建 SqlDataAdapter 对象
03  // 为 SqlDataAdapter 的 SelectCommand 属性赋值
04  ada.SelectCommand = new SqlCommand("select * from authors", con);
05  ……// 省略后续代码
```

同样，可以使用上述方法为 DataAdapter 对象的 InsertCommand、UpdateCommand 和 DeleteCommand 属性赋值，从而实现数据的添加、修改和删除操作。

DataAdapter 对象的常用方法及说明如表 10.7 所示。

表 10.7　DataAdapter 对象的常用方法及说明

方　法	说　明
Fill	从数据源中提取数据以填充数据集
Update	更新数据源

10.5.3　使用 DataAdapter 对象填充 DataSet 数据集

在使用 DataAdapter 对象填充 DataSet 数据集时，需要使用其 Fill 方法。Fill 方法常用的 3 种重载形式如下。

（1）int Fill(DataSet dataset)：添加或更新参数所指定的 DataSet 数据集，返回值是影响的行数。

（2）int Fill(DataTable datatable)：将数据填充到一个数据表中。

（3）int Fill(DataSet dataset,String tableName)：填充指定的 DataSet 数据集中的指定表。

示例 5．获取所有 C# 编程词典信息并显示在表格中

创建一个 Windows 应用程序，在默认窗体中添加一个 DataGridView 控件，用来显示使用 DataAdapter 对象填充后的 DataSet 数据集中的数据，代码如下。

```
01 private void Form1_Load(object sender, EventArgs e)
02 {
03     // 定义数据库连接字符串
04     string strCon = "Server=XIAOKE;User Id=sa;Pwd=;DataBase=db_EMS";
05     SqlConnection sqlcon = new SqlConnection(strCon);// 创建数据库连接对象
06     // 执行 SQL 查询语句
07     SqlDataAdapter sqlda = new SqlDataAdapter("select * from tb_PDic", sqlcon);
07     DataSet myds = new DataSet();                    // 创建数据集对象
08     sqlda.Fill(myds, "tabName");                     // 填充数据集中的指定表
09     // 为 dataGridView1 指定数据源
10     dataGridView1.DataSource = myds.Tables["tabName"];
11 }
```

运行上面代码，得到如图 10.9 所示的内容。

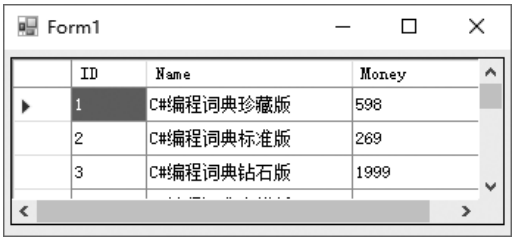


图 10.9　获取所有 C# 编程词典信息并显示在表格中

10.6　DataGridView 控件的使用

DataGridView 控件又称为数据表格控件，它提供一种强大而灵活的、以表格形式显示数据的方式。将数据绑定到 DataGridView 控件非常简单和直观，在大多数情况下，只需设置 DataSource 属性即可。另外，DataGridView 控件具有极高的可配置性和可扩展性，它提供了大量的属性、方法和事件，可以用来对该控件的外观和行为进行自定义。当需要在 Windows 窗体应用程序中显示表格数据时，首先考虑使用 DataGridView 控件。图 10.10 所示为 DataGridView 控件，其拖放到窗体中的效果如图 10.11 所示。

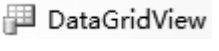


图 10.10　DataGridView 控件

图 10.11　DataGridView 控件拖放到窗体中的效果

DataGridView 控件的常用属性及说明如表 10.8 所示。

表 10.8　DataGridView 控件的常用属性及说明

属　性	说　明
Columns	获取一个包含控件中所有列的集合
CurrentCell	获取或设置当前处于活动状态的单元格
CurrentRow	获取包含当前单元格的行
DataSource	获取或设置 DataGridView 所显示数据的数据源
RowCount	获取或设置 DataGridView 中显示的行数
Rows	获取一个集合，该集合包含 DataGridView 控件中的所有行

DataGridView 控件的常用事件及说明如表 10.9 所示。

表 10.9　DataGridView 控件的常用事件及说明

事　件	说　明
CellClick	在单元格的任意部分被单击时发生
CellDoubleClick	在用户双击单元格中的任意位置时发生

下面通过一个示例讲解如何使用 DataGridView 控件，该示例实现的主要功能有禁止在 DataGridView 控件中添加 / 删除行、禁用 DataGridView 控件的自动排序功能、使 DataGridView 控件隔行显示不同的颜色、使 DataGridView 控件的选中行呈现不同的颜色，以及在选中 DataGridView 控件中的某行时，将其详细信息显示在 TextBox 文本框中。

示例 6．DataGridView 控件的综合应用

创建一个 Windows 应用程序，在默认窗体中添加两个 TextBox 控件和一个 DataGridView 控件，其中，TextBox 控件分别用来显示选中记录的版本和价格信息，DataGridView 控件用来显示数据表中的数据，代码如下。

```
01 // 定义数据库连接字符串
02 string strCon = "Server=XIAOKE;User Id=sa;Pwd=;DataBase=db_EMS";
03 SqlConnection sqlcon;                                    // 声明数据库连接对象
04 SqlDataAdapter sqlda;                                    // 声明数据库桥接器对象
05 DataSet myds;                                            // 声明数据集对象
06 private void Form1_Load(object sender, EventArgs e)
07 {
08     dataGridView1.AllowUserToAddRows = false;      // 禁止添加行
09     dataGridView1.AllowUserToDeleteRows = false;   // 禁止删除行
10     sqlcon = new SqlConnection(strCon);            // 创建数据库连接对象
11     // 获取数据表中的所有数据
12     sqlda = new SqlDataAdapter("select * from tb_PDic", sqlcon);
13     myds = new DataSet();                          // 创建数据集对象
14     sqlda.Fill(myds);                              // 填充数据集
15     dataGridView1.DataSource = myds.Tables[0]; // 为 dataGridView1 指定数据源
16     // 禁用 DataGridView 控件的排序功能
17     for (int i = 0; i < dataGridView1.Columns.Count; i++)
18         dataGridView1.Columns[i].SortMode = DataGridViewColumnSortMode.
NotSortable;
19     // 设置 SelectionMode 属性为 FullRowSelect，以使控件能够整行选择
20     dataGridView1.SelectionMode = DataGridViewSelectionMode.FullRowSelect;
21     // 设置 DataGridView 控件中的数据以各行换色的形式显示
22     foreach (DataGridViewRow dgvRow in dataGridView1.Rows) // 遍历所有行
23     {
24         if (dgvRow.Index % 2 == 0)                 // 判断是否是偶数行
25         {
26             // 设置偶数行颜色
27             dataGridView1.Rows[dgvRow.Index].DefaultCellStyle.BackColor = Color.
```

```
LightSalmon;
28          }
29          else                                    // 奇数行
30          {
31              // 设置奇数行颜色
32              dataGridView1.Rows[dgvRow.Index].DefaultCellStyle.BackColor = Color.
LightPink;
33          }
34      }
35      // 设置 dataGridView1 控件的 ReadOnly 属性，使其为只读
36       dataGridView1.ReadOnly = true;
36      // 设置 dataGridView1 控件的 DefaultCellStyle.SelectionBackColor 属性，使选
中行颜色变色
37      dataGridView1.DefaultCellStyle.SelectionBackColor = Color.LightSkyBlue;
38  }
39  private void dataGridView1_CellClick(object sender, DataGridViewCellEventArgs e)
40  {
41      if (e.RowIndex > 0)                         // 判断选中行的索引是否大于 0
42      {
43          // 记录选中的 ID 号
44          int intID = (int)dataGridView1.Rows[e.RowIndex].Cells[0].Value;
45          sqlcon = new SqlConnection(strCon); // 创建数据库连接对象
46          // 执行 SQL 查询语句
47          sqlda = new SqlDataAdapter("select * from tb_PDic where ID=" + intID + "",
sqlcon);
48          myds = new DataSet();                   // 创建数据集对象
49          sqlda.Fill(myds);                       // 填充数据集
50          if (myds.Tables[0].Rows.Count > 0)  // 判断数据集中是否有记录
51          {
52              textBox1.Text = myds.Tables[0].Rows[0][1].ToString();   // 显示版本
53              textBox2.Text = myds.Tables[0].Rows[0][2].ToString();   // 显示价格
54          }
55      }
56  }
```

运行上面代码，得到如图 10.12 所示的内容。

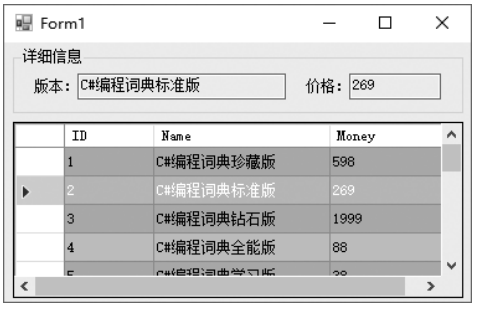


图 10.12　DataGridView 控件的使用

10.7 Entity Framework 编程基础

10.7.1 什么是 Entity Framework

Entity Framework（以下简写为 EF）是微软官方发布的 ORM 框架，其基于 ADO.NET，通过使用 EF 可以很方便地将表映射到实体对象上或将实体对象转换为数据库表。

学习笔记

ORM 是将数据存储从域对象自动映射到关系型数据库的工具。ORM 主要包括 3 部分：域对象、关系数据库对象、映射关系。ORM 使类提供自动化 CRUD，使开发人员从数据库 API 和 SQL 中解放出来。

EF 有 3 种使用场景，分别如下。

（1）从数据库中生成实体类。

（2）由实体类生成数据库表结构。

（3）通过数据库可视化设计器设计数据库，同时生成实体类。

EF 的 3 种使用场景示意图如图 10.13 所示。

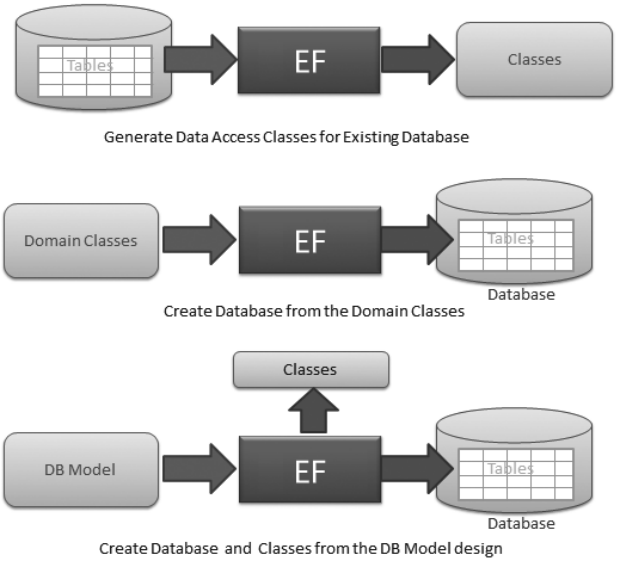


图 10.13　EF 的 3 种使用场景示意图

10.7.2　EF 实体数据模型

EF 实体数据模型（EDM ）包括 3 个模型：概念模型、映射模型和存储模型，下面分别进行说明。

（1）概念模型：概念模型由概念架构定义语言文件（.csdl）来定义，包含模型类和它们之间的关系，独立于数据库表的设计。

（2）映射模型：映射模型由映射规范语言文件（.msl）来定义，它包含有关如何将概念模型映射到存储模型的信息。

（3）存储模型：存储模型由存储架构定义语言文件（.ssdl）来定义，它是数据库设计模型，包括表、视图、存储的过程，以及它们的关系和键。

EDM 旋转 180° 后如图 10.14 所示。

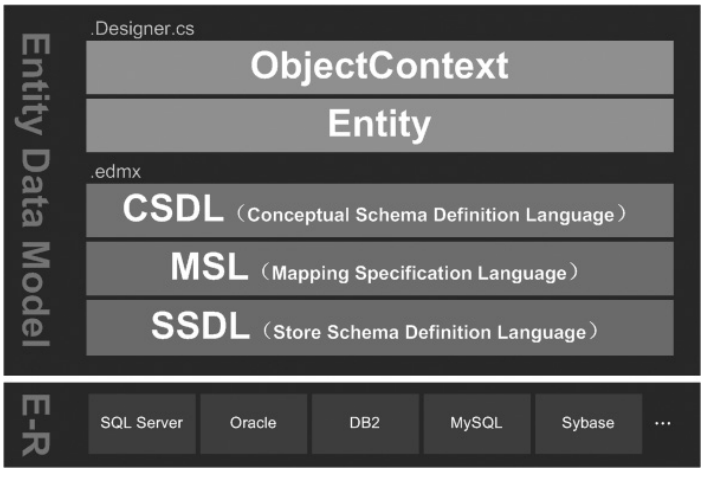


图 10.14　旋转 180° 后的 EDM

EDM 在项目中的表现形式就是扩展名为 .edmx 的文件，这个文件本质上是一个 XML 文件，可以手动编辑此文件来自定义 CSDL、MSL 与 SSDL 这 3 部分。

10.7.3　EF 运行环境

EF 框架曾经是 .NET Framework 的一部分，但在 Version 6 之后，从 .NET Framework 中分离出来。其中，EF 5.0 由两部分组成：EF API 和 .NET Framework 4.0/4.5；而 EF 6.0 是独立的 EntityFramework.dll，不依赖 .NET Framework。使用 NuGet 即可安装 EF，在安装 Visual Studio 2017 时，会自动安装 EF 的 5.0 和 6.0 版本。EF 5.0 运行环

境示意图如图 10.15 所示，EF 6.0 运行环境示意图如图 10.16 所示 .

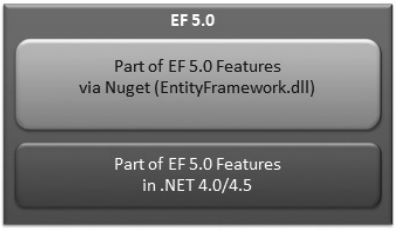


图 10.15　EF 5.0 运行环境示意图

图 10.16　EF 6.0 运行环境示意图

10.7.4　创建实体数据模型

下面以 db_EMS 数据库为例，将已有的数据库表映射为实体数据，步骤如下。

步骤 1，创建一个 Windows 窗体应用程序，选中当前项目，单击鼠标右键，在弹出的快捷菜单中依次选择“添加“→”新建项”命令，在弹出的“添加新项”对话框左侧“已安装”下选择“Visual C# 项”；在右侧列表中找到“ADO.NET 实体数据模型”并选中，在“名称”文本框中输入实体数据模型的名称，可以与数据库名称相同，如图 10.17 所示，然后单击“添加”按钮。

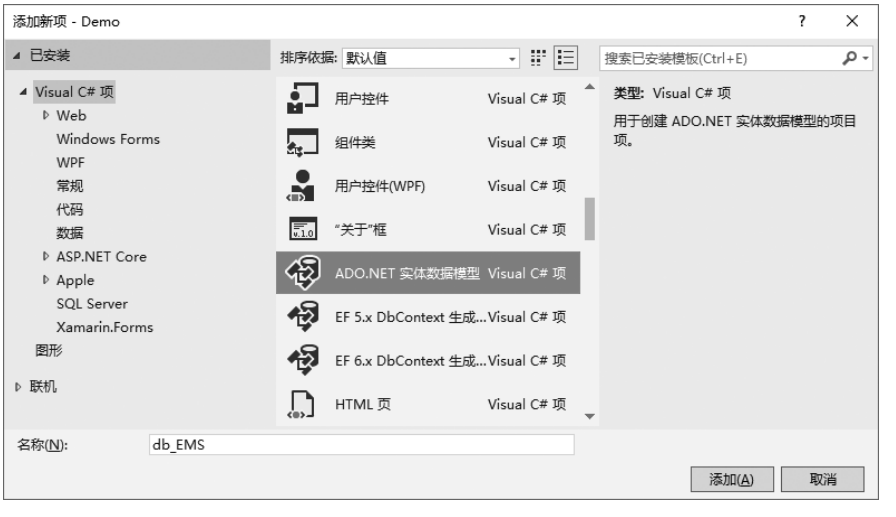


图 10.17　“添加新项”对话框

步骤 2，弹出“实体数据模型向导”对话框，在该对话框中选择“来自数据库的 EF 设计器”，如图 10.18 所示。

图 10.18　“实体数据模型向导”对话框

步骤 3，单击“下一步”按钮，在弹出的窗口中单击“新建连接”按钮，弹出“选择数据源”对话框，如图 10.19 所示，在该对话框中选择“Microsoft SQL Server”。

步骤 4，单击“继续”按钮，弹出“连接属性”对话框，如图 10.20 所示，该对话框中的设置如下。

（1）数据源：单击“更改”按钮，选择“Microsoft SQL Server (SqlClient)”，如果默认为该项，则忽略。

（2）服务器名：单击下拉按钮会自动寻找到本机机器名称，如果数据库在本地，那么选择自己的机器名称即可。

（3）身份验证：选择“SQL Server 身份验证”，填写用户名和密码（数据库登录名和密码）。

（4）在“选择或输入数据库名称”处单击下拉按钮，找到想要映射的数据库名称，本例为 db_EMS。

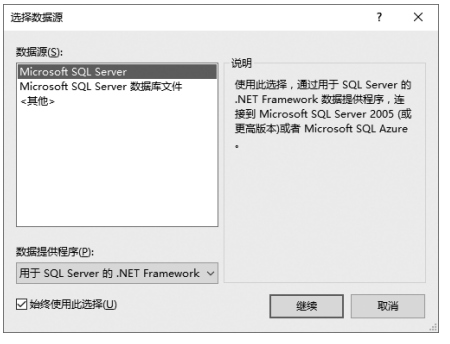


图 10.19 “选择数据源”对话框

图 10.20 “连接属性”对话框

步骤 5，以上信息设置完成后，单击“确定”按钮，返回“实体数据模型向导”对话框。单击“下一步”按钮，跳转到“选择您的版本”页面，如图 10.21 所示，在该页面中可以根据自己的实际需要进行选择，这里选择“实体框架 6.x”。

步骤 6，单击“下一步”按钮，跳转到“选择您的数据库对象和设置”页面，这里暂时用不到视图或存储过程和函数，所以只选择“表”即可，如图 10.22 所示，然后单击“完成”按钮。

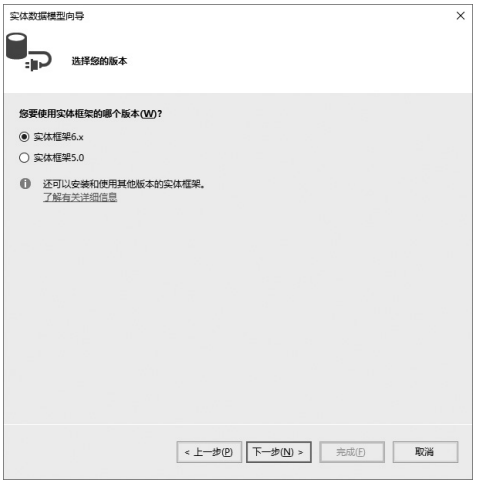


图 10.21 “选择您的版本”页面

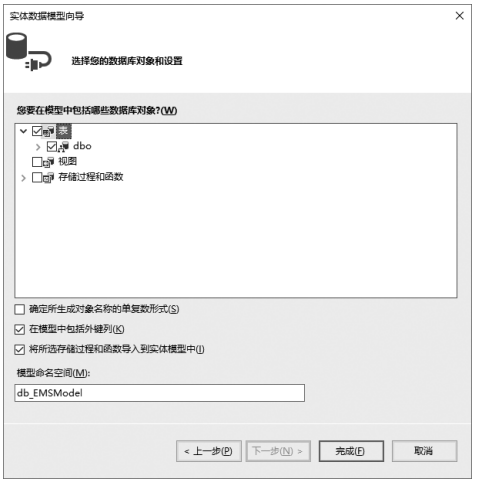


图 10.22 选择要映射的内容

等待生成完成后，编辑器会自动打开模型图页面以展示关联性，这里直接关闭即可。

打开“解决方案资源管理器”，发现当前项目中多了一个“db_EMS.edmx”文件，这就是模型实体和数据库上、下文类。图 10.23 所示为 EF 生成实体架构的整个架构情况。

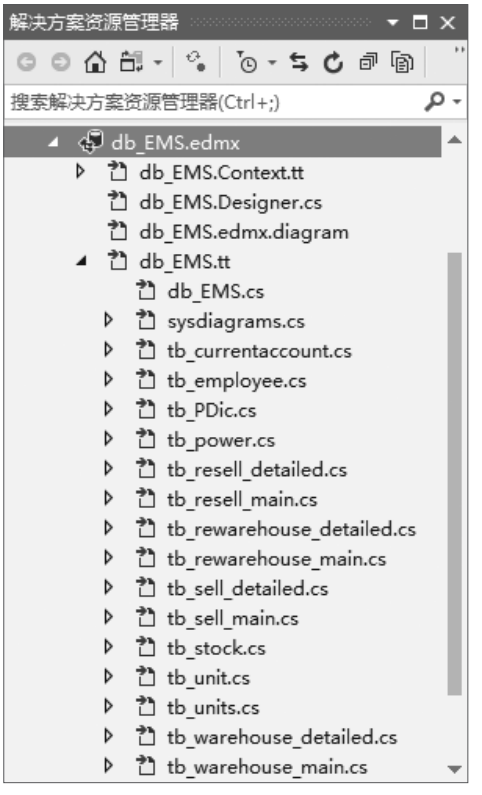


图 10.23　EF 生成实体架构的整个架构情况

10.7.5　使用 EF 对数据表进行增、删、改、查操作

在 10.7.4 节中创建了 EF 实体数据模型，本节将通过一个示例讲解如何使用 EF 对数据表进行增、删、改、查操作。

示例 7．使用 EF 对数据表进行增、删、改、查操作

本示例在 10.7.4 节的基础上实现，在默认窗体中添加 7 个 TextBox 控件，分别用来输入或编辑商品信息；添加一个 ComboBox 控件，用来显示商品的单位；添加两个 Button 控件，分别用来实现添加商品信息和修改商品信息的功能；添加一个 DataGridView 控件，用来实时显示数据表中的所有商品信息，代码如下。

```
01  string strID = "";                                    // 记录选中的商品编号
02  private void Form1_Load(object sender, EventArgs e)
03  {
```

```
04      using (db_EMSEntities db = new db_EMSEntities())
05      {
06          dgvInfo.DataSource = db.tb_stock.ToList(); // 显示数据表中的所有信息
07      }
08  }
09  private void btnAdd_Click(object sender, EventArgs e)
10  {
11      using (db_EMSEntities db = new db_EMSEntities())
12      {
13          tb_stock stock = new tb_stock
14          {
15              // 为 tb_stock 类中的商品实体赋值
16              tradecode = txtID.Text,
17              fullname = txtName.Text,
18              unit = cbox.Text,
19              type = txtType.Text,
20              standard = txtISBN.Text,
21              produce = txtAddress.Text,
22              qty = Convert.ToInt32(txtNum.Text),
23              price = Convert.ToDouble(txtPrice.Text)
24          };
25          db.tb_stock.Add(stock);                              // 构造添加 SQL 语句
26          db.SaveChanges();                                    // 进行数据库添加操作
27          dgvInfo.DataSource = db.tb_stock.ToList();           // 重新绑定数据源
28      }
29  }
30  private void btnEdit_Click(object sender, EventArgs e)
31  {
32      using (db_EMSEntities db = new db_EMSEntities())
33      {
34          tb_stock stock = new tb_stock { tradecode = txtID.Text, fullname =
txtName. Text };
35          db.tb_stock.Attach(stock);                           // 构造修改 SQL 语句
36          // 重新为各个字段赋值
37          stock.unit = cbox.Text;
38          stock.type = txtType.Text;
39          stock.standard = txtISBN.Text;
40          stock.produce = txtAddress.Text;
41          stock.qty = Convert.ToInt32(txtNum.Text);
42          stock.price = Convert.ToDouble(txtPrice.Text);
43          db.SaveChanges();                                    // 进行数据库修改操作
44          dgvInfo.DataSource = db.tb_stock.ToList();           // 重新绑定数据源
45      }
46  }
47  private void 删除 ToolStripMenuItem_Click(object sender, EventArgs e)
48  {
```

```
49      using (db_EMSEntities db = new db_EMSEntities())
50      {
51          // 查找要删除的记录
52          tb_stock stock = db.tb_stock.Where(W => W.tradecode == strID).FirstOr
Default();
53          if (stock != null)                          // 判断要删除的记录是否存在
54          {
55              db.tb_stock.Remove(stock);              // 构造删除 SQL 语句
56              db.SaveChanges();                       // 执行删除操作
57              dgvInfo.DataSource = db.tb_stock.ToList(); // 重新绑定数据源
58              MessageBox.Show(" 商品信息删除成功 ");
59          }
60          else
61              MessageBox.Show(" 请选择要删除的商品！ ");
62      }
63  }
64  private void dgvInfo_CellClick(object sender, DataGridViewCellEventArgs e)
65  {
66      if (e.RowIndex > 0)                             // 判断是否选择了行
67      {
68          // 获取选中的商品编号
69          strID = Convert.ToString(dgvInfo[0, e.RowIndex].Value).Trim();
70          using (db_EMSEntities db = new db_EMSEntities())
71          {
72              // 获取指定编号的商品信息
73              tb_stock stock = db.tb_stock.Where(W => W.tradecode == strID).
FirstOrDefault();
74              if (stock != null)                      // 判断查询结果是否为空
75              {
76                  txtID.Text = stock.tradecode;       // 显示商品编号
77                  txtName.Text = stock.fullname;      // 显示商品全称
78                  cbox.Text = stock.unit;             // 显示商品单位
79                  txtType.Text = stock.type;          // 显示商品类型
80                  txtISBN.Text = stock.standard;      // 显示商品规格
81                  txtAddress.Text = stock.produce;    // 显示商品产地
82                  txtNum.Text = stock.qty.ToString();       // 显示商品数量
83                  txtPrice.Text = stock.price.ToString(); // 显示商品价格
84              }
85          }
86      }
87  }
```

运行上面代码，得到如图 10.24 所示的内容。

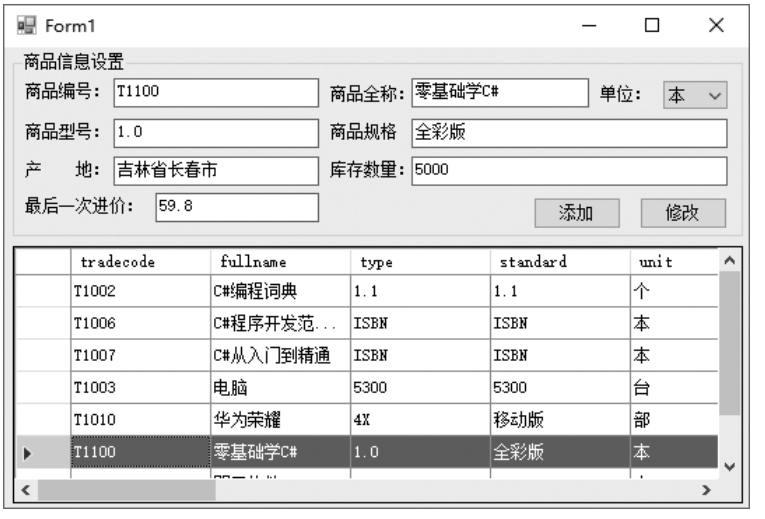


图 10.24　使用 EF 对数据表进行增、删、改、查操作

第 11 章　程序调试与异常处理

开发应用程序的代码必须安全、准确。但是在编写程序的过程中，不可避免地会出现错误，而有的错误不容易被发觉，从而导致程序运行错误。为了排除这些非常隐蔽的错误，需要对编写好的代码进行程序调试，这样才能确保应用程序成功运行。另外，在开发程序时，不仅要注意程序代码的准确性与合理性，还要注意程序中可能出现的异常情况。.NET 框架提供了一套被称为结构化异常处理的标准错误机制，在这种机制中，如果出现错误或任何预期之外的事件，则都会引发异常。本章将对 .NET 中的程序调试与异常处理进行详细讲解。

11.1　程序调试

开发人员在程序的开发过程中会不断体会到程序调试的重要性。为验证 C# 的运行状况，需要经常在某个方法调用的开始和结束位置分别使用 Console.WriteLine() 方法和 MessageBox.Show() 方法输出信息，并根据这些信息判断程序的执行状况，这是非常传统的程序调试方法，经常会导致程序代码混乱。下面介绍几种使用 Visual Studio 开发工具调试 C# 程序的方法。

11.1.1　Visual Studio 编辑器调试

在使用 Visual Studio 2017 开发 C# 程序时，编辑器不但能够为开发者提供代码编写、辅助提示和实时编译等常用功能，而且还能提供对 C# 源代码进行快捷修改、重构和语法纠错等高级功能。通过使用 Visual Studio 2017，可以很方便地找到一些语法错误，并且根据提示进行快速修正。下面对 Visual Studio 2017 提供的常用调试功能进行介绍。

1. 错误提示符

错误提示符位于出现错误的代码行的最右侧，用于指出错误所在的位置。用鼠标右键单击该提示符，可以弹出快捷菜单，在快捷菜单中可以对其进行基本的查看操作，如图 11.1 所示。

2. 代码下方的红色波浪线

在出现错误的代码下方，会显示红色的波浪线，将鼠标指针移动到红色波浪线上，将显示具体的错误内容，如图 11.2 所示的提示框，开发人员可根据错误提示对代码进行修改。

图 11.1　对错误提示符进行查看操作

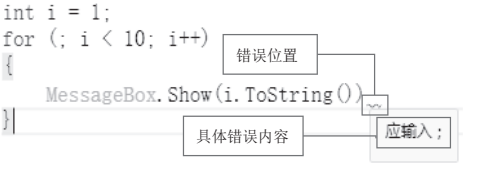


图 11.2　显示具体错误内容

3. 代码下方的绿色波浪线

在出现警告的代码下方，会显示绿色的波浪线，警告不会影响程序的正常运行，将鼠标指针移动到绿色波浪线上，将显示具体的警告内容，如图 11.3 所示的提示框，开发人员可以根据该警告内容对代码进行优化。

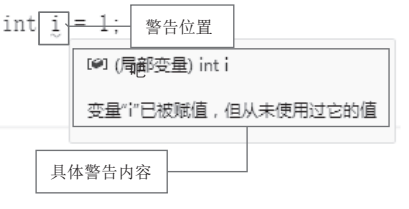


图 11.3　显示具体警告内容

11.1.2　Visual Studio 调试器调试

当代码不能正常运行时，可以通过调试定位错误。常用的程序调试操作包括设置断点、开始执行、中断执行、停止执行、单步执行和逐过程执行等。下面将对这几种常用的程序调试操作进行详细介绍。

1. 设置断点

断点通知调试器，使应用程序在某点上（暂停执行）或某情况发生时中断。在发生中断时，称程序和调试器处于中断模式。进入中断模式并不会终止或结束程序的执行，所有元素（如函数、变量和对象）都保留在内存中。执行可以在任何时候继续。

插入断点有 3 种方式：在要设置断点的代码行旁边的灰色空白处单击；右击要设置断点的代码行，在弹出的快捷菜单中选择“断点”→“插入断点”命令，如图 11.4 所示；

单击要设置断点的代码行，选择“调试”菜单中的“切换断点”命令，如图 11.5 所示。

图 11.4　利用快捷菜单插入断点

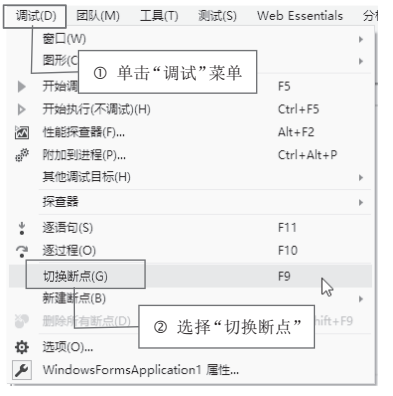


图 11.5　通过菜单栏插入断点

插入断点后，就会在设置断点的代码行旁边的灰色空白处出现一个红色圆点，并且该行代码也呈高亮显示，如图 11.6 所示。

删除断点主要有 3 种方式，分别如下。

（1）单击设置了断点的代码行左侧的红色圆点。

（2）在设置了断点的代码行左侧的红色圆点上单击鼠标右键，在弹出的快捷菜单中选择“删除断点”命令，如图 11.7 所示。

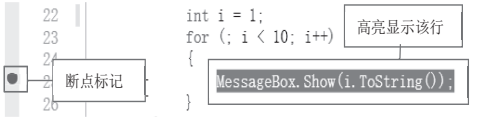


图 11.6　插入断点后效果图

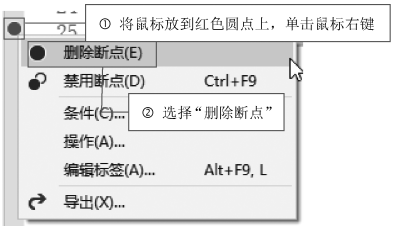


图 11.7　利用右键快捷菜单删除断点

（3）在设置了断点的代码行上单击鼠标右键，在弹出的快捷菜单中选择“断点”→“删除断点”命令。

2. 开始执行

开始执行是基本的程序调试功能之一，在“调试”菜单中选择“开始调试”命令（见图 11.8），或在源代码窗口中右击可执行代码中的某行，在弹出的快捷菜单中选择“运行到光标处”命令，如图 11.9 所示。

图 11.8 在“调试”菜单中选择“开始调试”命令

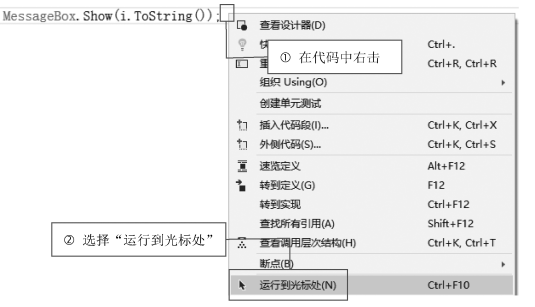


图 11.9 在右键快捷菜单中选择“运行到光标处”命令

除了使用上述方法开始执行，还可以直接单击工具栏中的 ▶ 启动 图标按钮启动调试，如图 11.10 所示。

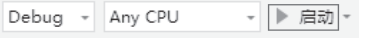


图 11.10 工具栏中的启动调试按钮

如果选择“开始调试”命令，则应用程序启动并一直运行到断点，此时断点处的代码以黄色底色显示，如图 11.11 所示。可以在任何时刻中断执行，以查看值（将鼠标指针移动到相应的变量或对象上即可查看其具体值，如图 11.12 所示）、修改变量或观察程序状态。

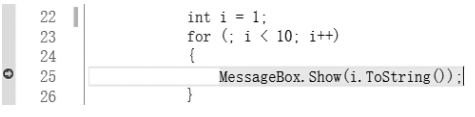


图 11.11 运行到断点效果图

图 11.12 查看变量的值

如果选择“运行到光标处”命令，则应用程序启动并一直运行到断点或光标位置，具体要看是断点在前还是光标在前，可以在源代码窗口中设置光标位置。如果光标在断点的前面，则代码首先运行到光标处，如图 11.13 所示。

3. 中断执行

当程序执行到一个断点或发生异常时，调试器将中断程序的执行。执行“调试”→“全部中断”命令后，调试器将停止所有在调试器下运行的程序的执行。程序并没有退出，可以随时恢复执行，此时应用程序处于中断模式。“调试”菜单中的“全部中断”命令如图 11.14 所示。

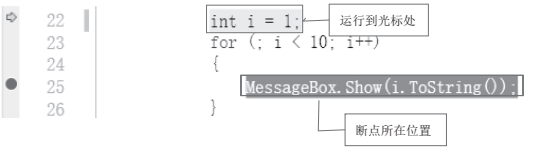


图 11.13　代码运行到光标处

图 11.14　“调试”菜单中的“全部中断”命令

除了通过执行“调试”→“全部中断”命令中断执行，还可以单击工具栏中的⏸图标按钮中断执行，如图 11.15 所示。

4. 停止执行

停止执行意味着终止正在调试的进程并结束调试会话，可以通过执行菜单栏中的“调试”→“停止调试”命令来结束运行和调试，也可以单击工具栏中的■图标按钮停止执行。

5. 单步执行和逐过程执行

通过单步执行，调试器每次只执行一行代码。单步执行主要是通过“逐语句”、“逐过程”和“跳出”这 3 个命令实现的。“逐语句”命令和“逐过程”命令的主要区别是，当某一行包含函数调用时，“逐语句”命令仅执行调用本身，然后在函数内的第一行代码处停止；而“逐过程”命令执行整个函数，之后在函数外的第一行代码处停止。如果位于函数调用的内部并想返回调用函数时，则应使用“跳出”命令。“跳出”命令将一直执行代码，直到函数返回，然后在调用函数中的返回点处中断。

当启动调试后，可以单击工具栏中的⇩图标按钮执行“逐语句”命令，单击⇨图标按钮执行“逐过程”命令，单击⇧图标按钮执行“跳出”命令，如图 11.16 所示。

图 11.15　工具栏中的中断执行按钮

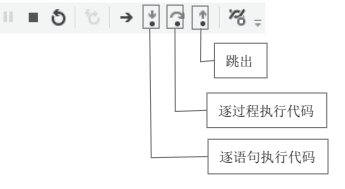


图 11.16　单步执行的 3 个命令

学习笔记

除了在工具栏中单击这 3 个图标按钮，还可以通过快捷键执行这 3 种操作。当启动调试后，按 <F11> 键可以执行“逐语句”操作，按 <F10> 键可以执行“逐过程”操作，按 <Shift+F10> 键可以执行“跳出”操作。

11.2 异常处理

在编写程序时，不仅要关心程序的正常操作，还应该检查代码错误及可能发生的各类不可预期的事件。在现代编程语言中，异常处理是解决这些问题的主要方法。异常处理是一种功能强大的机制，用于处理应用程序可能产生的错误或其他可以中断程序执行的异常情况。异常处理可以捕捉程序执行所发生的错误，通过异常处理可以有效、快速地构建各种用来处理程序异常情况的程序代码。

异常处理实际上相当于大楼失火时自动喷水灭火的过程。在大楼失火时（发生异常），烟雾感应器捕获到高于正常密度的烟雾（捕获异常），将自动喷水进行灭火（处理异常）。

在 .NET 类库中，提供了针对各种异常情形所设计的异常类，这些类包含了异常的相关信息。配合异常处理语句，应用程序能够轻易地避免程序执行时可能中断应用程序的各种错误。.NET 框架中的公共异常类及说明如表 11.1 所示，这些异常类都是 System.Exception 的直接或间接子类。

表 11.1 .NET 框架中的公共异常类及说明

公共异常类	说　明
System.ArithmeticException	在算术运算期间发生的异常
System.ArrayTypeMismatchException	当存储一个数组时，如果被存储的元素的实际类型与数组的实际类型不兼容而导致存储失败，则会引发此异常
System.DivideByZeroException	在试图用零除整数值时引发的异常
System.IndexOutOfRangeException	在试图使用小于零或超出数组界限的下标索引数组时引发的异常
System.InvalidCastException	如果从基类型（或接口）到派生类型的显示转换在运行时失败，则会引发此异常
System.NullReferenceException	在需要使用引用对象的场合，如果使用 null 引用，则会引发此异常
System.OutOfMemoryException	在分配内存的尝试失败时引发的异常
System.OverflowException	在选中的上、下文中所进行的算术运算、类型转换或转换操作导致溢出时引发的异常
System.StackOverflowException	在挂起的方法调用过多而导致执行堆栈溢出时引发的异常
System.TypeInitializationException	在静态构造函数引发异常并且没有可以捕捉到它的 catch 子句时引发的异常

在 C# 程序中，可以使用异常处理语句处理异常。主要的异常处理语句有 try…catch 语句、try…catch…finally 语句、throw 语句，通过这 3 个异常处理语句可以对可能产生异常的程序代码进行监控。下面对这 3 个异常处理语句进行详细讲解。

11.2.1　try…catch 语句

try…catch 语句允许在 try 后面的大括号中放置可能发生异常情况的程序代码，并对这些程序代码进行监控。在 catch 后面的大括号中则放置处理异常的程序代码，以处理程序发生的异常。try…catch 语句的基本格式如下。

```
try
{
    被监控的代码
}
catch(异常类名　异常变量名)
{
    异常处理
}
```

在 catch 子句中，异常类名必须为 System.Exception 或由 System.Exception 派生的类型。当 catch 子句指定了异常类名和异常变量名后，就相当于声明了一个具有给定名称和类型的异常变量，此异常变量表示当前正在处理的异常。

示例 1．未将对象引用设置到对象的实例

创建一个控制台应用程序，声明一个 object 类型的变量 obj，其初始值为 null；然后将 obj 强制转换成 int 类型并赋给 int 类型变量 N，使用 try…catch 语句捕获异常。代码如下。

```
01  static void Main(string[] args)
02  {
03      try                                  // 使用 try...catch 语句
04      {
05          object obj = null;               // 声明一个 object 类型的变量，初始值为 null
06          int i = (int)obj;                // 将 obj 强制转换成 int 类型
07      }
08      catch (Exception ex)                 // 捕获异常
09      {
10          Console.WriteLine("捕获异常：" + ex);        // 输出异常
11      }
12      Console.ReadLine();
13  }
```

运行上面代码，得到如图 11.17 所示的内容。

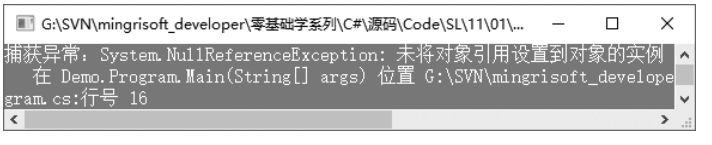


图 11.17　捕获异常

从图 11.17 中可以看出，上面代码的运行结果抛出了异常，这是因为声明的 object 类型的变量 obj 被初始化为 null，然后又将 obj 强制转换成 int 类型，这样就产生了异常。由于使用了 try…catch 语句，所以将这个异常捕获，并将异常输出。

学习笔记

有时为了编程简单会忽略 catch 代码块中的代码，这样 try…catch 语句就成了摆设，一旦程序在运行过程中出现异常，这个异常将很难查找。因此，要养成良好的编程习惯，在 catch 代码块中写入处理异常的代码。

11.2.2 try…catch…finally 语句

完整的异常处理语句应该包含 finally 代码块，在通常情况下，无论程序中有无异常产生，finally 代码块中的代码都会被执行，其基本格式如下。

```
try
{
    被监控的代码
}
catch(异常类名   异常变量名)
{
    异常处理
}
……
finally
{
    程序代码
}
```

对于 try…catch…finally 语句的理解并不复杂，它只是比 try…catch 语句多了一个 finally 语句，如果程序中有一些在任何情形中都必须执行的代码，那么就可以将它们放在 finally 代码块中。

学习笔记

使用 catch 子句是为了允许处理异常。无论是否引发异常，使用 finally 子句都可以执行清理代码。如果分配了昂贵或有限的资源（如数据库连接或流），则应将释放这些资源的代码放置在 finally 代码块中。

示例 2．捕捉将字符串转换为整型数据时的异常

创建一个控制台应用程序，声明一个 string 类型的变量 str，并将其初始化为“用一生下载你”；然后声明一个 object 类型的变量 obj，将 str 赋给变量 obj；最后声明一个 int 类型的变量 i，将 obj 强制转换成 int 类型后赋给变量 i，这样必然会导致转换错误，抛出异常。在 finally 语句中输出“程序执行完毕 …”，这样，无论程序是否抛出异常，都会执行 finally 语句中的代码。代码如下。

```
01  static void Main(string[] args)
02  {
03      string str = "零基础学 C#";                    // 声明一个 string 类型的变量 str
04      object obj = str;                              // 声明一个 object 类型的变量 obj
05      try                                            // 使用 try...catch 语句
06      {
07          int i = (int)obj;                          // 将 obj 强制转换成 int 类型
08      }
09      catch (Exception ex)                           // 获取异常
10      {
11          Console.WriteLine(ex.Message);             // 输出异常信息
12      }
13      finally                                        //finally 语句
14      {
15          Console.WriteLine("程序执行完毕 ...");          // 输出“程序执行完毕 …”
16      }
17      Console.ReadLine();
18  }
```

上面代码的运行结果如下。

```
指定的转换无效。
程序执行完毕 …
```

11.2.3　throw 语句

throw 语句用于主动引发一个异常，使用 throw 语句可以在特定的情形下自行抛出异常。throw 语句的基本格式如下。

```
throw  ExObject
```

ExObject：所要抛出的异常对象，这个异常对象是派生自 System.Exception 类的对象。

学习笔记

通常 throw 语句与 try…catch 语句或 try…catch…finally 语句一起使用。当引发异常时，程序查找处理此异常的 catch 语句，也可以用 throw 语句重新引发已捕获的异常。

示例 3．抛出除数为 0 的异常

创建一个控制台应用程序，创建一个 int 类型的方法 MyInt，此方法有两个 string 类型的参数，即 a 和 b。在这个方法中，使 a 为被除数，b 为除数。如果除数的值是 0，则通过 throw 语句抛出 DivideByZeroException 异常，这个异常被此方法中的 catch 子句捕获并输出。代码如下。

```
01 static int MyInt(string a, string b) // 创建一个 int 类型的方法，参数分别是 a 和 b
02 {
03     int int1;                          // 定义被除数
04     int int2;                          // 定义除数
05     int num;                           // 定义商
06     try                                // 使用 try...catch 语句
07     {
08         int1 = int.Parse(a);  // 将参数 a 强制转换成 int 类型后赋给 int1
09         int2 = int.Parse(b);           // 将参数 b 强制转换成 int 类型后赋给 int2
10         if (int2 == 0)                 // 判断 int2 是否等于 0，如果等于 0，则抛出异常
11         {
12             throw new DivideByZeroException();// 抛出 DivideByZeroException 类的异常
13         }
14         num = int1 / int2;             // 计算 int1 除以 int2 的值
15         return num;                    // 返回计算结果
16     }
17     catch (DivideByZeroException de) // 捕获异常
18     {
19         Console.WriteLine("用零除整数引发异常！ ");
20         Console.WriteLine(de.Message);
21         return 0;
22     }
23 }
24 static void Main(string[] args)
25 {
26     try                                                  // 使用 try...catch 语句
27     {
28         Console.Write("请输入分子：");                    // 提示输入分子
29         string str1 = Console.ReadLine();                // 获取键盘输入的值
30         Console.Write("请输入分母：");                    // 提示输入分母
31         string str2 = Console.ReadLine();                // 获取键盘输入的值
32          // 调用 MyInt 方法，获取键盘输入的分子与分母相除得到的值
33         Console.WriteLine("分子除以分母的值：" + MyInt(str1, str2));
34     }
35     catch (FormatException)                              // 捕获异常
36     {
37         Console.WriteLine("请输入数值格式数据 ");          // 输出提示
38     }
39     Console.ReadLine();
40 }
```

运行上面代码，得到如图 11.18 所示的内容。

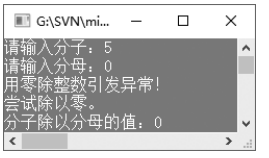


图 11.18　抛出除数为 0 的异常

第三篇　高级篇

第 12 章　I/O 数据流技术

在变量、对象和数组中存储的数据是暂时的，程序结束后就会丢失。为了能够长时间保存程序中的数据，需要将程序中的数据保存到磁盘文件中。C# 的 I/O 数据流技术可以将数据保存到文件（如文本文件等）中，以达到长时间保存数据的目的。掌握 I/O 数据流技术能够提高对数据的处理能力。

12.1　文件的基本操作

对文件的基本操作大体可以分为判断文件是否存在、创建文件、复制或移动文件、删除文件，以及获取文件的基本信息，本节将对文件的基本操作进行详细讲解。

12.1.1　File 类

File 类支持对文件的基本操作，它包含用于创建文件、复制文件、删除文件、移动文件和打开文件的静态方法，并协助创建 FileStream 对象。File 类中包含 40 多种方法，这里只列出其常用的几种方法，如表 12.1 所示。

表 12.1　File 类的常用方法及说明

方　法	说　明
Copy	将现有文件复制到新文件
Create	在指定路径中创建文件
Delete	删除指定的文件。如果指定的文件不存在，则不引发异常
Exists	判断指定的文件是否存在
Move	将指定文件移动到新位置，并提供指定新文件名的选项
Open	打开指定路径上的 FileStream
CreateText	创建或打开一个文件用于写入 UTF-8 编码的文本
GetCreationTime	返回指定文件或目录的创建日期和时间

续表

方　法	说　明
GetLastAccessTime	返回上次访问指定文件或目录的日期和时间
GetLastWriteTime	返回上次写入指定文件或目录的日期和时间
OpenRead	打开现有文件以进行读取
OpenText	打开现有 UTF-8 编码文本文件以进行读取
OpenWrite	打开现有文件以进行写入
ReadAllLines	打开一个文本文件，将文件的所有行都读入一个字符串数组，然后关闭该文件
ReadAllText	打开一个文本文件，将文件的所有内容读入一个字符串，然后关闭该文件
Replace	使用其他文件的内容替换指定文件的内容，这一过程将删除原始文件，并创建被替换文件的备份
SetCreationTime	设置创建该文件的日期和时间
SetLastAccessTime	设置上次访问指定文件的日期和时间
SetLastWriteTime	设置上次写入指定文件的日期和时间
WriteAllLines	创建一个新文件，在其中写入指定的字符串，然后关闭该文件。如果目标文件已存在，则改写该文件
WriteAllText	创建一个新文件，在其中写入内容，然后关闭该文件。如果目标文件已存在，则改写该文件

学习笔记

在使用与文件、文件夹及流相关的类时，首先需要添加 System.IO 命名空间。

12.1.2　FileInfo 类

FileInfo 类和 File 类之间许多方法的调用都是相同的，但是 FileInfo 类没有静态方法，该类中的方法只能用于实例化的对象。File 类是静态类，其调用需要字符串参数为每个方法调用规定文件位置。如果要在对象上进行单一方法的调用，则可以使用静态 File 类，在这种情况下静态调用速度要快一些，因为 .NET 框架不必执行实例化新对象并调用其方法。如果要在文件上执行几种操作，则实例化 FileInfo 对象并调用其方法更好一些，这样会提高效率，因为对象将在文件系统上引用正确的文件，而静态类每次都要寻找文件。

FileInfo 类的常用属性及说明如表 12.2 所示。

表 12.2　FileInfo 类的常用属性及说明

属　性	说　明
CreationTime	获取或设置当前 FileSystemInfo 对象的创建时间
Directory	获取父目录的实例

续表

属　性	说　明
DirectoryName	获取表示目录的完整路径的字符串
Exists	获取指示文件是否存在的值
Extension	获取表示文件扩展名部分的字符串
FullName	获取目录或文件的完整目录
IsReadOnly	获取或设置确定当前文件是否为只读的值
LastAccessTime	获取或设置上次访问当前文件或目录的时间
LastWriteTime	获取或设置上次写入当前文件或目录的时间
Length	获取当前文件的大小
Name	获取文件名

学习笔记

- 由于 File 类中的所有方法都是静态的，所以如果只想执行一种操作，那么使用 File 类中的方法的效率比使用相应的 FileInfo 类中的方法可能更高。
- 在使用 File 类中的方法时，需要对所有方法都执行安全检查，因此如果打算多次重用某个对象，则可以考虑用 FileInfo 类中的相应方法，因为这些方法并不总是需要执行安全检查。

12.1.3　判断文件是否存在

在判断文件是否存在时，可以使用 File 类的 Exists 方法或 FileInfo 类的 Exists 属性来实现，下面分别对其进行介绍。

1. File 类的 Exists 方法

File 类的 Exists 方法主要用于判断指定的文件是否存在，其语法格式如下。

```
public static bool Exists (string path)
```

path：要检查的文件。

返回值：如果调用方具有要求的权限，并且 path 包含现有文件的名称，则为 true；否则为 false。如果 path 为空引用或零长度字符串，则此方法也返回 false。如果调用方不具有读取指定文件所需要的足够权限，则不引发异常并且该方法返回 false，这与 path 是否存在无关。

例如，使用 File 类的 Exists 方法判断 C 盘根目录下是否存在 Test.txt 文件，代码如下。

```
File.Exists("C:\\Test.txt");
```

2. FileInfo 类的 Exists 属性

FileInfo 类的 Exists 属性用于获取指示文件是否存在的值，其语法格式如下。

```
public override bool Exists { get; }
```

属性值：如果该文件存在，则为 true；如果该文件不存在或该文件是目录，则为 false。

例如，首先实例化一个 FileInfo 对象，然后使用该对象调用 FileInfo 类中的 Exists 属性判断 C 盘根目录下是否存在 Test.txt 文件，代码如下。

```
01 FileInfo finfo = new FileInfo("C:\\Test.txt");          // 创建文件对象
02 if (finfo.Exists)                                       // 判断文件是否存在
03 {
04 }
```

12.1.4　创建文件

创建文件可以使用 File 类的 Create 方法或 FileInfo 类的 Create 方法来实现，下面分别对其进行介绍。

1. File 类的 Create 方法

File 类的 Create 方法为可重载方法，具有以下 4 种重载形式。

```
public static FileStream Create (string path)
public static FileStream Create (string path,int bufferSize)
public static FileStream Create (string path,int bufferSize,FileOptions options)
public static FileStream Create (string path,int bufferSize,FileOptions
options,FileSecurity fileSecurity)
```

File 类的 Create 方法的参数及说明如表 12.3 所示。

表 12.3　File 类的 Create 方法的参数及说明

参　数	说　明
Path	文件名
bufferSize	用于读取和写入文件已放入缓冲区的字节数
options	FileOptions 值之一，用于描述如何创建或改写该文件
fileSecurity	FileSecurity 值之一，用于确定文件的访问控制和审核安全性

例如，调用 File 类的 Create 方法在 C 盘根目录下创建一个 Test.txt 文本文件，代码如下。

```
File.Create("C:\\Test.txt");
```

2. FileInfo 类的 Create 方法

FileInfo 类的 Create 方法的语法格式如下。

```
public FileStream Create ()
```

返回值：新文件。在默认情况下，该方法将向所有用户授予对新文件的完全读、写访问权限。

例如，首先实例化一个 FileInfo 对象，然后使用该对象调用 FileInfo 类的 Create 方法在 C 盘根目录下创建一个 Test.txt 文本文件，代码如下。

```
01 FileInfo finfo = new FileInfo("C:\\Test.txt");      // 创建文件对象
02 finfo.Create();                                     // 创建文件
```

学习笔记

在使用 File 类和 FileInfo 类创建文本文件时，其默认的字符编码为 UTF-8；而在 Windows 环境中手动创建文本文件时，其字符编码为 ANSI。

12.1.5 复制文件

在复制文件时，可以使用 File 类的 Copy 方法或 FileInfo 类的 CopyTo 方法来实现，下面分别对其进行介绍。

1. File 类的 Copy 方法

File 类的 Copy 方法为可重载方法，具有以下两种重载形式。

```
public static void Copy (string sourceFileName,string destFileName)
public static void Copy (string sourceFileName,string destFileName,bool
overwrite)
```

sourceFileName：要复制的文件。

destFileName：目标文件的名称，不能是目录；如果是第一种重载形式，则该参数不能是现有文件。

overwrite：如果可以改写目标文件，则为 true；否则为 false。

例如，调用 File 类的 Copy 方法将 C 盘根目录下的 Test.txt 文本文件复制到 D 盘根目

录下，代码如下。

```
File.Copy("C:\\Test.txt", "D:\\Test.txt");
```

2. FileInfo 类的 CopyTo 方法

FileInfo 类的 CopyTo 方法为可重载方法，具有以下两种重载形式。

```
public FileInfo CopyTo (string destFileName)
public FileInfo CopyTo (string destFileName,bool overwrite)
```

destFileName：要复制的新文件的名称。

overwrite：若为 true，则允许改写现有文件；否则为 false。

返回值：第一种重载形式的返回值为带有完全限定路径的新文件。第二种重载形式的返回值为新文件，如果 overwrite 为 true，则为现有文件的改写；如果文件存在，且 overwrite 为 false，则会发生 IOException 异常。

例如，首先实例化一个 FileInfo 对象，然后使用该对象调用 FileInfo 类的 CopyTo 方法将 C 盘根目录下的 Test.txt 文本文件复制到 D 盘根目录下。如果 D 盘根目录下已经存在 Test.txt 文本文件，则将其替换。代码如下。

```
01  FileInfo finfo = new FileInfo("C:\\Test.txt");      // 创建文件对象
02  finfo.CopyTo("D:\\Test.txt", true);                 // 将文件复制到 D 盘
```

12.1.6　移动文件

在移动文件时，可以使用 File 类的 Move 方法或 FileInfo 类的 MoveTo 方法来实现，下面分别对其进行介绍。

1. File 类的 Move 方法

File 类的 Move 方法用于将指定文件移动到新位置，并提供指定新文件名的选项，其语法格式如下。

```
public static void Move (string sourceFileName,string destFileName)
```

sourceFileName：要移动的文件的名称。

destFileName：文件的新路径。

例如，调用 File 类的 Move 方法将 C 盘根目录下的 Test.txt 文本文件移动到 D 盘根目录下，代码如下。

```
File.Move("C:\\Test.txt", "D:\\Test.txt");
```

2. FileInfo 类的 MoveTo 方法

FileInfo 类的 MoveTo 方法用于将指定文件移动到新位置，并提供指定新文件名的选项，其语法格式如下。

```
public void MoveTo (string destFileName)
```

destFileName：要将文件移动到的路径，可以指定另一个文件名。

例如，下面代码首先实例化了一个 FileInfo 对象，然后使用该对象调用 FileInfo 类的 MoveTo 方法将 C 盘根目录下的 Test.txt 文本文件移动到 D 盘根目录下。

```
01 FileInfo finfo = new FileInfo("C:\\Test.txt");      // 创建文件对象
02 finfo.MoveTo("D:\\Test.txt");                       // 将文件移动（剪切）到 D 盘
```

学习笔记

在使用 Move 方法或 MoveTo 方法移动现有文件时，如果原文件和目的文件是同一个文件，将发生 IOException 异常。

12.1.7 删除文件

在删除文件时，可以使用 File 类的 Delete 方法或 FileInfo 类的 Delete 方法来实现，下面分别对其进行介绍。

1. File 类的 Delete 方法

File 类的 Delete 方法用于删除指定的文件，其语法格式如下。

```
public static void Delete (string path)
```

path：要删除的文件名称。

例如，调用 File 类的 Delete 方法删除 C 盘根目录下的 Test.txt 文本文件，代码如下。

```
File.Delete("C:\\Test.txt");
```

2. FileInfo 类的 Delete 方法

FileInfo 类的 Delete 方法用于永久删除文件，其语法格式如下。

```
public override void Delete ()
```

例如，首先实例化一个 FileInfo 对象，然后使用该对象调用 FileInfo 类的 Delete 方法删除 C 盘根目录下的 Test.txt 文本文件，代码如下。

```
01 FileInfo finfo = new FileInfo("C:\\Test.txt");      // 创建文件对象
02 finfo.Delete();                                     // 删除文件
```

12.1.8　获取文件的基本信息

在获取文件的基本信息时，主要用到了 FileInfo 类中的各种属性，下面通过一个示例说明如何获取文件的基本信息。

示例 1．获取选定文件的详细信息

获取选定文件的详细信息程序开发步骤如下。

步骤 1，新建一个 Windows 应用程序，在默认的 Form1 窗体中添加一个 OpenFileDialog 控件、一个 TextBox 控件和一个 Button 控件。其中，OpenFileDialog 控件用来显示“打开”对话框，TextBox 控件用来显示选择的文件名，Button 控件用来打开“打开”对话框并获取所选文件的基本信息。

步骤 2，双击触发 Button 控件的 Click 事件，在该事件中，使用 FileInfo 对象的属性获取文件的详细信息并显示，代码如下。

```
01 private void button1_Click(object sender, EventArgs e)
02 {
03     if (openFileDialog1.ShowDialog() == DialogResult.OK)
04     {
05         textBox1.Text = openFileDialog1.FileName; // 显示打开的文件
06         FileInfo finfo = new FileInfo(textBox1.Text);     // 创建 FileInfo 对象
07         // 获取文件创建时间
08          string strCTime = finfo.CreationTime.ToShortDateString();
09         // 获取上次访问该文件的时间
10         string strLATime = finfo.LastAccessTime.ToShortDateString();
11          // 获取上次写入文件的时间
12         string strLWTime = finfo.LastWriteTime.ToShortDateString();
13         string strName = finfo.Name;                 // 获取文件名称
14         string strFName = finfo.FullName;            // 获取文件的完整目录
15         string strDName = finfo.DirectoryName;       // 获取文件的完整路径
16         string strISRead = finfo.IsReadOnly.ToString();   // 判断文件是否只读
17         long lgLength = finfo.Length;          // 获取文件长度
18          MessageBox.Show("文件信息：\n 创建时间：" + strCTime + " 上次访问时间：" +
strLATime + "\n 上次写入时间：" + strLWTime + " 文件名称：" + strName + "\n 完整目录："
+ strFName + "\n 完整路径：" + strDName + "\n 是否只读：" + strISRead + " 文件长度：" +
lgLength);
19     }
20 }
```

运行上面代码，单击“浏览”按钮，弹出“打开”对话框；选择文件，单击“打开”按钮，在弹出的对话框中将显示所选文件的基本信息。获取选定文件的详细信息程序运行结果如图 12.1 所示。

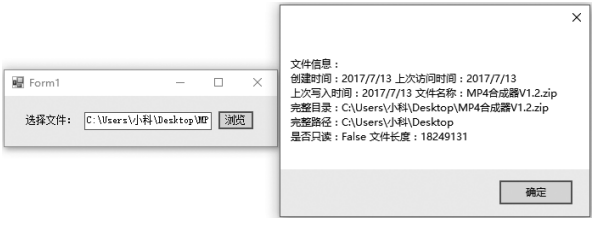


图 12.1　获取选定文件的详细信息程序运行结果

12.2　文件夹的基本操作

对文件夹的基本操作大体可以分为判断文件夹是否存在、创建文件夹、移动文件夹、删除文件夹，以及遍历文件夹中的文件，本节将对文件夹的基本操作进行详细讲解。

12.2.1　Directory 类

Directory 类公开了用于创建、移动、枚举、删除目录和子目录的静态方法，这里介绍一些 Directory 类的常用方法，如表 12.4 所示。

表 12.4　Directory 类的常用方法及说明

方　法	说　明
CreateDirectory	创建指定路径中的所有目录
Delete	删除指定的目录
Exists	判断指定路径是否引用磁盘上的现有目录
GetCreationTime	获取目录的创建日期和时间
GetDirectories	获取指定目录中子目录的名称
GetDirectoryRoot	返回指定路径的卷信息、根信息或同时返回这两种信息
GetFiles	返回指定目录中的文件名称
GetFileSystemEntries	返回指定目录中所有文件和子目录的名称
GetLastAccessTime	返回上次访问指定文件或目录的日期和时间
GetLastWriteTime	返回上次写入指定文件或目录的日期和时间
GetParent	检索指定路径的父目录，包括绝对路径和相对路径
Move	将文件或目录及其内容移动到新位置
SetCreationTime	为指定的文件或目录设置创建日期和时间
SetCurrentDirectory	将应用程序的当前工作目录设置为指定的目录
SetLastAccessTime	设置上次访问指定文件或目录的日期和时间
SetLastWriteTime	设置上次写入目录的日期和时间

12.2.2　DirectoryInfo 类

DirectoryInfo 类和 Directory 类之间许多方法的调用是相同的，但是 DirectoryInfo 类没有静态方法，该类中的方法只能用于实例化的对象。Directory 类是静态类，其调用需要字符串参数为每个方法调用规定文件夹路径。如果要在对象上进行单一方法调用，则可以使用静态 Directory 类，在这种情况下静态调用速度要快一些，因为 .NET 框架不必执行实例化新对象并调用其方法。如果要在文件夹上执行几种操作，则实例化 DirectoryInfo 对象并调用其方法则更好一些，这样会提高效率，因为对象将在文件夹系统上引用正确的文件夹，而静态类每次都要寻找文件夹。

DirectoryInfo 类的常用属性及说明如表 12.5 所示。

表 12.5　DirectoryInfo 类的常用属性及说明

属　　性	说　　明
CreationTime	获取或设置当前 FileSystemInfo 对象的创建时间
Exists	获取指示目录是否存在的值
Extension	获取表示文件扩展名部分的字符串
FullName	获取目录或文件的完整目录
LastAccessTime	获取或设置上次访问当前文件或目录的时间
LastWriteTime	获取或设置上次写入当前文件或目录的时间
Name	获取 DirectoryInfo 实例的名称
Parent	获取指定子目录的父目录
Root	获取路径的根部分

12.2.3　判断文件夹是否存在

在判断文件夹是否存在时，可以使用 Directory 类的 Exists 方法或 DirectoryInfo 类的 Exists 属性来实现，下面分别对其进行介绍。

1. Directory 类的 Exists 方法

Directory 类的 Exists 方法用于判断指定路径是否引用磁盘上的现有目录，其语法格式如下。

```
public static bool Exists (string path)
```

path：要测试的路径。

返回值：如果 path 引用现有目录，则为 true；否则为 false。

例如，使用 Directory 类的 Exists 方法判断 C 盘根目录下是否存在 Test 文件夹，代码如下。

```
Directory.Exists("C:\\Test ");
```

2. DirectoryInfo 类的 Exists 属性

DirectoryInfo 类的 Exists 属性用于获取指示目录是否存在的值，其语法格式如下。

```
public override bool Exists { get; }
```

属性值：如果目录存在，则为 true；否则为 false。

例如，首先实例化一个 DirectoryInfo 对象，然后使用该对象调用 DirectoryInfo 类中的 Exists 属性判断 C 盘根目录下是否存在 Test 文件夹，代码如下。

```
01 DirectoryInfo dinfo = new DirectoryInfo("C:\\Test");// 创建文件夹对象
02 if (dinfo.Exists)                                    // 判断文件夹是否存在
03 {
04 }
```

12.2.4 创建文件夹

在创建文件夹时，可以使用 Directory 类的 CreateDirectory 方法或 DirectoryInfo 类的 Create 方法来实现，下面分别对其进行介绍。

1. Directory 类的 CreateDirectory 方法

Directory 类的 CreateDirectory 方法为可重载方法，其具有以下两种重载形式。

```
public static DirectoryInfo CreateDirectory (string path)
public static DirectoryInfo CreateDirectory (string path,DirectorySecurity
directorySecurity)
```

path：要创建的目录路径。

directorySecurity：要应用于此目录的访问控制。

返回值：第一种重载形式的返回值为由 path 指定的 DirectoryInfo；第二种重载形式的返回值为新创建的目录的 DirectoryInfo 对象。

例如，调用 Directory 类的 CreateDirectory 方法在 C 盘根目录下创建一个 Test 文件夹，代码如下。

```
Directory.CreateDirectory("C:\\Test ");
```

2. DirectoryInfo 类的 Create 方法

DirectoryInfo 类的 Create 方法为可重载方法，其具有以下两种重载形式。

```
public void Create ()
public void Create (DirectorySecurity directorySecurity)
```

directorySecurity：要应用于此目录的访问控制。

例如，首先实例化一个 DirectoryInfo 对象，然后使用该对象调用 DirectoryInfo 类的 Create 方法在 C 盘根目录下创建一个 Test 文件夹，代码如下。

```
01 DirectoryInfo dinfo = new DirectoryInfo("C:\\Test");  // 创建文件夹对象
02 dinfo.Create();                                       // 创建文件夹
```

12.2.5　移动文件夹

在移动文件夹时，可以使用 Directory 类的 Move 方法或 DirectoryInfo 类的 MoveTo 方法来实现，下面分别对其进行介绍。

1. Directory 类的 Move 方法

Directory 类的 Move 方法用于将文件或目录及其内容移动到新位置，其语法格式如下。

```
public static void Move (string sourceDirName,string destDirName)
```

sourceDirName：要移动的文件或目录的路径。

destDirName：指向 sourceDirName 的新位置的路径。

例如，调用 Directory 类的 Move 方法将 C 盘根目录下的 Test 文件夹移动到 C 盘根目录下的“新建文件夹”文件夹中，代码如下。

```
Directory.Move("C:\\Test ", "C:\\ 新建文件夹 \\Test");
```

学习笔记

在使用 Move 方法移动文件夹时需要统一磁盘根目录，如 C 盘中的文件夹只能移动到 C 盘中的某个文件夹下；同样，在使用 MoveTo 方法移动文件夹时也是如此，下面不再强调。

2. DirectoryInfo 类的 MoveTo 方法

DirectoryInfo 类的 MoveTo 方法用于将 DirectoryInfo 对象及其内容移动到新路径，其语法格式如下。

```
public void MoveTo (string destDirName)
```

destDirName：要将此目录移动到的目标位置的名称和路径。目标不能是另一个具有相同名称的磁盘卷或目录，它可以是要将此目录作为子目录添加到其中的一个现有目录。

例如，首先实例化一个 DirectoryInfo 对象，然后使用该对象调用 DirectoryInfo 类的 MoveTo 方法将 C 盘根目录下的 Test 文件夹移动到 C 盘根目录下的“新建文件夹”文件夹中，代码如下。

```
01 DirectoryInfo dinfo = new DirectoryInfo("C:\\Test");      // 创建文件夹对象
02 dinfo.MoveTo("C:\\ 新建文件夹 \\Test");                    // 移动（剪切）文件夹
```

12.2.6 删除文件夹

在删除文件夹时，可以使用 Directory 类的 Delete 方法或 DirectoryInfo 类的 Delete 方法来实现，下面分别对其进行介绍。

1. Directory 类的 Delete 方法

Directory 类的 Delete 方法为可重载方法，其具有以下两种重载形式。

```
public static void Delete (string path)
public static void Delete (string path,bool recursive)
```

path：要移除的空目录或目录的名称。

recursive：若要移除 path 中的目录、子目录和文件，则为 true；否则为 false。

例如，调用 Directory 类的 Delete 方法删除 C 盘根目录下的 Test 文件夹，代码如下。

```
Directory.Delete("C:\\Test");
```

2. DirectoryInfo 类的 Delete 方法

DirectoryInfo 类的 Delete 方法用于永久删除文件夹，其具有以下两种重载形式。

```
public override void Delete ()
public void Delete (bool recursive)
```

recursive：若为 true，则删除此目录、其子目录及所有文件；否则为 false。

学习笔记

对于第一种重载形式，如果 DirectoryInfo 为空，则删除它；对于第二种重载形式，删除 DirectoryInfo 对象并判断是否要删除子目录和文件。

例如，首先实例化一个 DirectoryInfo 对象，然后使用该对象调用 DirectoryInfo 类的 Delete 方法删除 C 盘根目录下的 Test 文件夹，代码如下。

```
01  DirectoryInfo dinfo = new DirectoryInfo("C:\\Test");      // 创建文件夹对象
02  dinfo.Delete();                                            // 删除文件夹
```

12.2.7　遍历文件夹

在遍历文件夹时，可以分别使用 DirectoryInfo 类提供的 GetDirectories 方法、GetFiles 方法和 GetFileSystemInfos 方法来实现，下面分别对这 3 个方法进行详细讲解。

1. GetDirectories 方法

GetDirectories 方法用来返回当前目录的子目录，该方法为可重载方法，其具有以下 3 种重载形式。

```
public DirectoryInfo[] GetDirectories ()
public DirectoryInfo[] GetDirectories (string searchPattern)
public DirectoryInfo[] GetDirectories (string searchPattern,SearchOption
searchOption)
```

searchPattern：搜索字符串，如用于搜索所有以单词 System 开头的目录的“System*”。

searchOption：SearchOption 枚举的一个值，指定搜索操作是应仅包含当前目录还是应包含所有子目录。

返回值：第一种重载形式的返回值为 DirectoryInfo 对象的数组；第二种重载形式和第三种重载形式的返回值为与 searchPattern 匹配的 DirectoryInfo 类型的数组。

2. GetFiles 方法

GetFiles 方法用来返回当前目录的文件列表，该方法为可重载方法，其具有以下 3 种重载形式。

```
public FileInfo[] GetFiles ()
public FileInfo[] GetFiles (string searchPattern)
public FileInfo[] GetFiles (string searchPattern,SearchOption searchOption)
```

searchPattern：搜索字符串（如“*.txt”）。

searchOption：SearchOption 枚举的一个值，指定搜索操作是应仅包含当前目录还是应包含所有子目录。

返回值：FileInfo 类型数组。

3. GetFileSystemInfos 方法

GetFileSystemInfos 方法用来检索表示当前目录的文件和子目录的强类型 FileSystemInfo 对象的数组，该方法为可重载方法，其具有以下两种重载形式。

```
public FileSystemInfo[] GetFileSystemInfos ()
public FileSystemInfo[] GetFileSystemInfos (string searchPattern)
```

searchPattern：搜索字符串。

返回值：第一种重载形式的返回值为强类型 FileSystemInfo 对象的数组；第二种重载形式的返回值为与搜索条件匹配的强类型 FileSystemInfo 对象的数组。

学习笔记

一般在遍历文件夹时都会使用 GetFileSystemInfos 方法，因为 GetDirectories 方法只遍历文件夹中的子文件夹，GetFiles 方法只遍历文件夹中的文件，而 GetFileSystemInfos 方法遍历文件夹中的所有子文件夹及文件。

示例 2．获取文件夹中的所有子文件夹及文件信息

获取文件夹中的所有子文件夹及文件信息程序开发步骤如下。

步骤 1，新建一个 Windows 应用程序，默认窗体为 Form1.cs。

步骤 2，在 Form1 窗体中添加一个 FolderBrowserDialog 控件、一个 TextBox 控件、一个 Button 控件和一个 ListView 控件。其中，FolderBrowserDialog 控件用来显示“浏览文件夹”对话框，TextBox 控件用来显示选择的文件夹路径及名称，Button 控件用来打开“浏览文件夹”对话框并获取所选文件夹中的子文件夹及文件，ListView 控件用来显示选择的文件夹中的子文件夹及文件信息。

步骤 3，双击触发 Button 控件的 Click 事件，在该事件中，使用 DirectoryInfo 对象的 GetFileSystemInfos 方法获取指定文件夹中的所有子文件夹及文件，然后将获取到的信息显示在 ListView 列表中，代码如下。

```
01 private void button1_Click(object sender, EventArgs e)
02 {
03     listView1.Items.Clear();
04     if (folderBrowserDialog1.ShowDialog() == DialogResult.OK)
05     {
06         textBox1.Text = folderBrowserDialog1.SelectedPath;
07         // 创建 DirectoryInfo 对象
08         DirectoryInfo dinfo = new DirectoryInfo(textBox1.Text);
09         // 获取指定目录下的所有子目录及文件类型
10         FileSystemInfo[] fsinfos = dinfo.GetFileSystemInfos();
11         foreach (FileSystemInfo fsinfo in fsinfos)
12         {
13             if (fsinfo is DirectoryInfo)// 判断是否为文件夹
14             {
```

```
15                  // 使用获取的文件夹名称实例化 DirectoryInfo 对象
16                  DirectoryInfo dirinfo = new DirectoryInfo(fsinfo.FullName);
17                  // 为 ListView 控件添加文件夹信息
18                  listView1.Items.Add(dirinfo.Name);
19                  listView1.Items[listView1.Items.Count - 1].SubItems.Add (dirinfo.
FullName);
20                  listView1.Items[listView1.Items.Count - 1].SubItems.Add("");
21                  listView1.Items[listView1.Items.Count - 1].SubItems.Add(dirinfo.
CreationTime.ToShortDateString());
22              }
23              else
24              {
25                  // 使用获取的文件名称实例化 FileInfo 对象
26                  FileInfo finfo = new FileInfo(fsinfo.FullName);
27                  // 为 ListView 控件添加文件信息
28                  listView1.Items.Add(finfo.Name);
29                  listView1.Items[listView1.Items.Count - 1].SubItems.Add (finfo.
FullName);
30                  listView1.Items[listView1.Items.Count - 1].SubItems.Add(finfo.
Length.ToString());
31                  listView1.Items[listView1.Items.Count - 1].SubItems.Add(finfo.
CreationTime.ToShortDateString());
32              }
33          }
34      }
35  }
```

运行上面代码，单击“浏览”按钮，弹出“浏览文件夹”对话框；选择文件夹，单击“确定”按钮，将选择的文件夹中所包含的子文件夹及文件信息显示在 ListView 控件中。获取文件夹中的所有子文件夹及文件信息程序运行结果如图 12.2 所示。

图 12.2　获取文件夹中的所有子文件夹及文件信息程序运行结果

12.3　I/O（输入 / 输出）

作为在 .NET Framework 中执行读、写文件操作的一种非常重要的介质，I/O 数据流提

供了一种向后备存储写入字节和从后备存储读取字节的方式，下面对 I/O 数据流进行详细讲解。

12.3.1 流概述

在程序开发过程中，将输入设备与输出设备之间的数据传递抽象为流。例如，键盘可以输入数据，显示器可以显示键盘输入的数据等。按照不同的分类方式，可以将流分为不同的类型：根据操作流的数据单元的不同，可以将流分为字节流（操作的数据单元是 1 字节）和字符流（操作的数据单元是 2 字节或一个字符，因为一个字符占 2 字节）；根据流的流向的不同，可以将流分为输入流和输出流。

从内存的角度出发，输入流是指数据从数据源（如文件、压缩包或视频等）流入内存的过程，输入流示意图如图 12.3 所示；输出流是指数据从内存流出到数据源的过程，输出流示意图如图 12.4 所示。

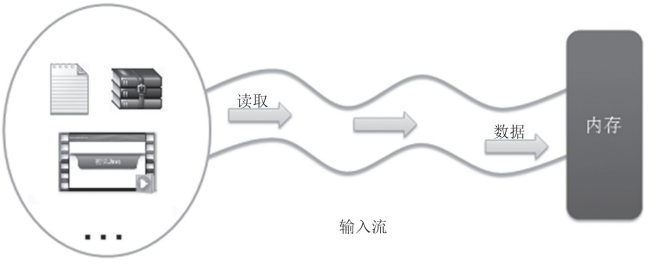


图 12.3　输入流示意图

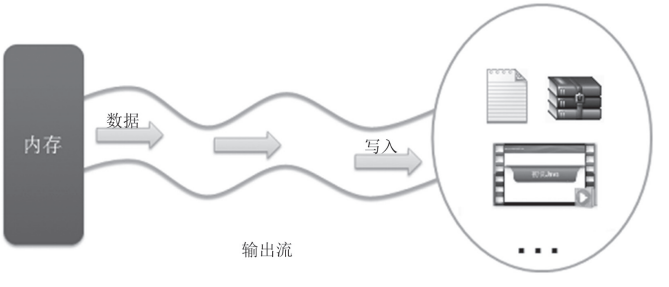


图 12.4　输出流示意图

学习笔记

输入流用来读取数据，输出流用来写入数据。

在 .NET Framework 中，流由 Stream 类来表示，该类构成了所有其他流的抽象类。不能直接创建 Stream 类的实例，但是必须使用它实现某个 I/O 流类。

C# 中有许多类型的流，但在处理文件 I/O 时，最重要的类型的流为 FileStream 类，它提供了读取文件和写入文件的方式。可在处理文件 I/O 时使用的其他流主要包括 BufferedStream、CryptoStream、MemoryStream 和 NetworkStream 等。

12.3.2　文件 I/O 流的介绍

在 C# 中，文件 I/O 流使用 FileStream 类来实现，该类公开以文件为主的 Stream，表示在磁盘或网络路径上指向文件的流。一个 FileStream 类的实例实际上代表一个磁盘文件，它通过 Seek 方法进行对文件的随机访问，同时也包含了流的标准输入、标准输出和标准错误等。FileStream 类默认对文件的打开方式是同步的，但它同样很好地支持异步操作。

1. FileStream 类的常用属性

FileStream 类的常用属性及说明如表 12.6 所示。

表 12.6　FileStream 类的常用属性及说明

属　性	说　明
CanRead	获取一个值，该值指示当前流是否支持读取
CanSeek	获取一个值，该值指示当前流是否支持查找
CanTimeout	获取一个值，该值确定当前流是否可以超时
CanWrite	获取一个值，该值指示当前流是否支持写入
IsAsync	获取一个值，该值指示 FileStream 是异步还是同步打开的
Length	获取用字节表示的流长度
Name	获取传递给构造函数的 FileStream 的名称
Position	获取或设置此流的当前位置
ReadTimeout	获取或设置一个值，该值确定流在超时前尝试读取多长时间
WriteTimeout	获取或设置一个值，该值确定流在超时前尝试写入多长时间

2. FileStream 类的常用方法

FileStream 类的常用方法及说明如表 12.7 所示。

表 12.7　FileStream 类的常用方法及说明

方　法	说　明
BeginRead	开始异步读操作
BeginWrite	开始异步写操作
Close	关闭当前流并释放与之关联的所有资源

续表

方　法	说　明
EndRead	等待挂起的异步读取完成
EndWrite	结束异步写入，在 I/O 操作完成之前一直阻止
Lock	在允许读取访问的同时防止其他进程更改 FileStream
Read	从流中读取字节块并将该数据写入指定缓冲区
ReadByte	从文件中读取 1 字节，并将读取位置提升 1 字节
Seek	将该流的当前位置设置为指定值
SetLength	将该流的长度设置为指定值
Unlock	允许其他进程访问以前锁定的某个文件的全部内容或部分内容
Write	使用从缓冲区读取的数据将字节块写入该流
WriteByte	将 1 字节写入文件流的当前位置

3. 使用 FileStream 类操作文件

要用 FileStream 类操作文件，就要先实例化一个 FileStream 对象。FileStream 类的构造函数具有许多不同的重载形式，其中包括一个最重要的参数，即 FileMode 枚举。

FileMode 枚举规定了如何打开或创建文件。FileMode 枚举的成员及说明如表 12.8 所示。

表 12.8　FileMode 枚举的成员及说明

枚举成员	说　明
Append	打开现有文件并查找到文件尾，或创建新文件。FileMode.Append 只能同 FileAccess.Write 一起使用。任何读尝试都将失败并引发 ArgumentException 异常
Create	指定操作系统应创建新文件。如果文件已存在，则它将被改写。这要求文件具有写入权限。System.IO.FileMode.Create 等效于这样的请求：如果文件不存在，则使用 CreateNew；否则使用 Truncate
CreateNew	指定操作系统应创建新文件。此操作需要 FileIOPermissionAccess.Write。如果文件已存在，则将引发 IOException 异常
Open	指定操作系统应打开现有文件，打开文件的能力取决于 FileAccess 所指定的值。如果该文件不存在，则引发 System.IO.FileNotFoundException 异常
OpenOrCreate	指定操作系统应打开文件。如果要打开的文件不存在，则应创建新文件。如果用 FileAccess.Read 打开文件，则需要 FileIOPermissionAccess.Read。如果文件访问方式为 FileAccess.Write 或 FileAccess.ReadWrite，则需要 FileIOPermissionAccess.Write；如果文件访问方式为 FileAccess.Append，则需要 FileIOPermissionAccess.Append
Truncate	指定操作系统应打开现有文件。文件一旦打开，将被截断为 0 字节大小。此操作需要 FileIOPermissionAccess.Write。试图从使用 Truncate 打开的文件中进行读取操作将引发异常

例如，使用 FileStream 类对象打开 Test.txt 文本文件并对其进行读、写操作，代码如下。

```
FileStream aFile = new FileStream("Test.txt", FileMode.OpenOrCreate,
FileAccess. ReadWrite);
```

12.3.3　使用 I/O 流操作文本文件

在使用 I/O 流操作文本文件时主要会用到 StreamWriter 类和 StreamReader 类，下面对这两个类进行详细讲解。

1. StreamWriter 类

StreamWriter 类是专门用来处理文本文件的类，可以方便地向文本文件中写入字符串，同时它也负责重要的转换和处理向 FileStream 对象写入的工作。

StreamWriter 类的常用属性及说明如表 12.9 所示。

表 12.9　StreamWriter 类的常用属性及说明

属　性	说　明
Encoding	获取将输出写入到其中的 Encoding
Formatprovider	获取控制格式设置的对象
NewLine	获取或设置由当前 TextWriter 使用的行结束符字符串

StreamWriter 类的常用方法及说明如表 12.10 所示。

表 12.10　StreamWriter 类的常用方法及说明

方　法	说　明
Close	关闭当前的 StringWriter 和基础流
Write	写入 StringWriter 的此实例中
WriteLine	写入重载参数指定的某些数据，后跟行结束符

2. StreamReader 类

StreamReader 类是专门用来读取文本文件的类。StreamReader 类可以从底层 Stream 对象创建 StreamReader 对象的实例，而且还能指定编码规范参数。在创建 StreamReader 对象后，StreamReader 类提供了许多用于读取和浏览字符数据的方法。

StreamReader 类的常用方法及说明如表 12.11 所示。

表 12.11　StreamReader 类的常用方法及说明

方　法	说　明
Close	关闭 StringReader
Read	读取输入字符串中的下一个字符或下一组字符

续表

方　法	说　明
ReadBlock	从当前流中读取最大 count 的字符并从 index 开始将该数据写入 Buffer
ReadLine	从基础字符串中读取一行
ReadToEnd	将整个流或从流的当前位置到流的结尾作为字符串读取

示例 3．向文本文件中写入名人名言并进行读取

向文本文件中写入名人名言并进行读取程序开发步骤如下。

步骤 1，新建一个 Windows 应用程序，默认窗体为 Form1.cs。

步骤 2，在 Form1 窗体中添加一个 SaveFileDialog 控件、一个 OpenFileDialog 控件、一个 TextBox 控件和两个 Button 控件。其中，SaveFileDialog 控件用来显示“另存为”对话框，OpenFileDialog 控件用来显示“打开”对话框，TextBox 控件用来输入要写入文本文件的内容和显示选中文本文件的内容，Button 控件分别用来打开“另存为”对话框并执行文本文件写入操作和打开“打开”对话框并执行文本文件读取操作。

步骤 3，分别双击“写入”按钮和“读取”按钮，触发它们的 Click 事件，在这两个事件中，分别使用 StreamWriter 类和 StreamReader 类向文本文件中写入内容和从文本文件中读取内容，代码如下。

```
01 private void button1_Click(object sender, EventArgs e)
02 {
03     if (textBox1.Text == string.Empty)
04     {
05         MessageBox.Show(" 要写入的文件内容不能为空 ");
06     }
07     else
08     {
09         saveFileDialog1.Filter = " 文本文件 (*.txt)|*.txt";// 设置保存文件的格式
10         if (saveFileDialog1.ShowDialog() == DialogResult.OK)
11         {
12             // 使用“另存为”对话框中输入的文件名实例化 StreamWriter 对象
13              StreamWriter sw = new StreamWriter(saveFileDialog1.FileName, true);
14              sw.WriteLine(textBox1.Text);           // 向创建的文件中写入内容
15              sw.Close();                            // 关闭当前文件写入流
16         }
17     }
18 }
19 private void button2_Click(object sender, EventArgs e)
20 {
21     openFileDialog1.Filter = " 文本文件 (*.txt)|*.txt"; // 设置打开文件的格式
22     if (openFileDialog1.ShowDialog() == DialogResult.OK)
```

```
23      {
24          textBox1.Text = string.Empty;
25          // 使用“打开”对话框中选择的文件实例化 StreamReader 对象
26          StreamReader sr = new StreamReader(openFileDialog1.FileName);
27          // 调用 ReadToEnd 方法读取选中文件的全部内容
28          textBox1.Text = sr.ReadToEnd();
29          sr.Close();                                    // 关闭当前文件读取流
30      }
31  }
```

运行上面代码，单击“写入”按钮，弹出“另存为”对话框，输入要保存的文件名，单击“保存”按钮，将文本框中的内容写入文件中；单击“读取”按钮，弹出“打开”对话框，选择要读取的文件，单击“打开”按钮，将选择的文件中的内容显示在文本框中。向文本文件中写入名人名言并进行读取程序运行结果如图 12.5 所示，写入文本文件中的内容如图 12.6 所示。

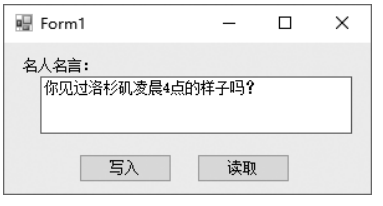


图 12.5　向文本文件中写入名人名言并进行读取程序运行结果

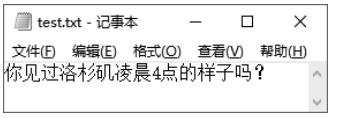


图 12.6　写入文本文件中的内容

第 13 章　GDI+ 绘图应用

使用图形分析数据不仅简单明了，而且清晰可见，它是项目开发中的一项必备功能。那么，在 C# 程序中如何实现图形的绘制呢？答案是使用 GDI+ ！使用 GDI+ 可以轻松绘制用户界面屏幕，GDI+ 还提供画笔、画刷、颜色、图形等。本章将对如何使用 GDI+ 绘制图形进行详细讲解。

13.1　GDI+ 绘图基础

绘图是高级程序设计中非常重要的技术。例如，应用程序需要绘制闪屏图像、背景图像、组件外观，Web 程序可以绘制统计图、数据库存储的图像资源等。正所谓“一图胜千言”，使用图像能够更好地表达程序运行结果，进行细致的数据分析与保存等。本节将对 C# 中的 GDI+ 进行介绍。

13.1.1　GDI+ 概述

GDI+ 指的是 .NET Framework 中提供的二维图形、图像处理等功能，它是构成 Windows 操作系统的一个子系统，提供了图形图像操作的应用程序编程接口（API）。使用 GDI+ 可以用相同的方式在屏幕或打印机上显示信息，而无须考虑特定显示设备的细节。GDI+ 类提供程序员用于绘制的方法，这些方法随后会调用特定设备的驱动程序。GDI+ 将应用程序与图形硬件分隔，使程序员能够创建与设备无关的应用程序。GDI+ 主要用于在窗体上绘制各种图形图像，可以用于绘制各种数据图形、数学仿真等。GDI+ 可以在窗体程序中产生很多自定义的图形，便于开发人员展示各种图形化的数据。

GDI+ 就好比一个绘图仪，它可以将已经制作好的图形绘制在指定的模板中，并可以对图形的颜色、线条粗细、位置等进行设置。

13.1.2　Graphics 类

Graphics 类是 GDI+ 的核心，Graphics 对象表示 GDI+ 绘图表面，提供将对象绘制到显示设备的方法。Graphics 对象与特定的设备上、下向关联，是用于创建图形图像的对象。Graphics 类封装了绘制直线、绘制曲线、绘制图形、绘制图像和文本的方法，是进行一切 GDI+ 操作的基础类。创建 Graphics 对象有以下 3 种方法。

（1）在窗体或控件的 Paint 事件中创建，将其作为 PaintEventArgs 的一部分。在为控件创建绘制代码时，通常会使用此方法来获取对图形对象的引用。

例如，在 Paint 事件中创建 Graphics 对象，代码如下。

```
01 private void Form1_Paint(object sender, PaintEventArgs e) // 窗体的 Paint 事件
02 {
03     Graphics g = e.Graphics;                              // 创建 Graphics 对象
04 }
```

（2）调用控件或窗体的 CreateGraphics 方法以获取对 Graphics 对象的引用，该对象表示控件或窗体的绘图画面。如果在已存在的窗体或控件上绘图，则应该使用此方法。

例如，在窗体的 Load 事件中，通过 CreateGraphics 方法创建 Graphics 对象，代码如下。

```
01 private void Form1_Load(object sender, EventArgs e)       // 窗体的 Load 事件
02 {
03     Graphics g;                                           // 声明一个 Graphics 对象
04     // 使用 CreateGraphics 方法创建 Graphics 对象
05     g = this.CreateGraphics();
06 }
```

（3）由从 Image 继承的任何对象创建 Graphics 对象，此方法在需要更改已存在的图像时十分有用。

例如，在窗体的 Load 事件中，通过 FromImage 方法创建 Graphics 对象，代码如下。

```
01 private void Form1_Load(object sender, EventArgs e)       // 窗体的 Load 事件
02 {
03     Bitmap mbit = new Bitmap(@"C:\mr.bmp");               // 实例化 Bitmap 类
04     // 通过 FromImage 方法创建 Graphics 对象
05     Graphics g = Graphics.FromImage(mbit);
06 }
```

13.2 设置画笔与画刷

13.2.1 设置画笔

Pen 类主要用于设置画笔，其构造函数语法如下。

```
public Pen (Color color,float width)
```

color：设置 Pen 的颜色。

width：设置 Pen 的宽度。

例如，创建一个 Pen 对象，使其颜色为蓝色，宽度为 2，代码如下。

```
Pen mypen1 = new Pen(Color.Blue, 2);     // 实例化一个 Pen 类，并设置其颜色和宽度
```

学习笔记

在上面的语法中，在设置画笔颜色时用到了 Color 结构，该结构主要用来定义颜色，在该结构中定义了表示常用颜色的属性，开发人员可以直接使用。Color 结构中表示颜色的属性及说明如表 13.1 所示。

表 13.1 Color 结构中表示颜色的属性及说明

属　性	说　明
Black	黑色
Blue	蓝色
Cyan	青色
Gray	灰色
Green	绿色
LightGray	浅灰色
Magenta	洋红色
Orange	橘黄色
Pink	粉红色
Red	红色
White	白色
Yellow	黄色

13.2.2　设置画刷

Brush 类主要用于设置画刷，以填充几何图形。例如，将正方形和圆形填充其他颜色。Brush 类是一个抽象基类，不能进行实例化。如果要创建一个画刷对象，则需要使用 Brush 类派生出的类（如 SolidBrush、HatchBrush 等类）。下面对 Brush 类派生出的常用的类进行讲解。

1. SolidBrush 类

SolidBrush 类定义单色画刷，画刷用于填充图形形状，如矩形、椭圆形、扇形、多边形和封闭路径。

SolidBrush 类的语法如下。

```
public SolidBrush(Color color)
```

color：此画刷的颜色。

例如，创建一个画刷对象，设置画刷的颜色为红色，代码如下。

```
Brush mybs = new SolidBrush(Color.Red);          // 创建颜色为红色的画刷
```

2. HatchBrush 类

HatchBrush 类提供了一种特定样式的图形，用来实现填满整个封闭区域的绘图效果。

HatchBrush 类的语法如下。

```
public HatchBrush (HatchStyle hatchstyle,Color foreColor)
```

hatchstyle：HatchStyle 值之一，表示此 HatchBrush 所绘制的图案。

foreColor：Color 结构，表示此 HatchBrush 所绘制线条的颜色。

学习笔记

在使用 HatchBrush 类时，必须添加 System.Drawing.Drawing2D 命名空间。

例如，使用 HatchBrush 类创建一个画刷对象，使用 HatchStyle 指定要绘制的图案为交叉的水平线和垂直线，代码如下。

```
Brush brush = new HatchBrush(HatchStyle.Cross, Color.Red);          // 创建画刷
```

使用上面定义的画刷绘制出的交叉的水平线和垂直线图案如图 13.1 所示。

3. LinerGradientBrush 类

LinerGradientBrush 类提供了一种渐变色彩的特效，用来填满图形的内部区域。

LinerGradientBrush 类的语法如下。

```
public LinerGradientBrush(Point point1, Point point2,Color color1, Color color2)
```

语法中的参数说明如表 13.2 所示。

表 13.2　LinerGradientBrush 类的语法中的参数说明

参　数	说　明
point1	表示线性渐变的开始点
point2	表示线性渐变的结束点
color1	表示线性渐变的开始色彩
color2	表示线性渐变的结束色彩

学习笔记

在使用 LinerGradientBrush 类时，必须添加 System.Drawing.Drawing2D 命名空间。

例如，使用 LinerGradientBrush 类创建一个渐变画刷对象，渐变颜色为从红色到蓝色，代码如下。

```
01  Point p1 = new Point(100, 100);                    // 指定渐变的开始点
02  Point p2 = new Point(150, 150);                    // 指定渐变的结束点
03  // 实例化 LinerGradientBrush 类，设置其使用红色和蓝色进行渐变
04  LinearGradientBrush brush = new LinearGradientBrush(p1, p2, Color.Red,
Color.Blue);
```

使用上面定义的渐变画刷绘制出颜色渐变的图形，如图 13.2 所示。

图 13.1　交叉的水平线和垂直线图案

图 13.2　渐变画刷绘制的图案示意图

13.3　绘制几何图形

在 C# 中使用 Graphics 类来绘制几何图形，Graphics 类使用不同的方法可以实现不同图形的绘制。例如，DrawLine() 方法用于绘制直线，DrawRectangle() 方法用于绘制矩形，DrawEllipse() 方法用于绘制椭圆形等。

Graphics 类中常用的图形绘制方法如表 13.3 所示。

表 13.3　Graphics 类中常用的图形绘制方法

方　法	说　明	举　例	绘 图 效 果
DrawArc()	弧线	g.DrawArc(pen,100, 100, 100, 50, 270, 200);	
DrawLine()	直线	g.DrawLine(pen,10, 10, 50, 10); g.DrawLine(pen, 30, 10, 30, 40);	
DrawEllipse()	椭圆形	g.DrawEllipse(pen, 10, 10, 50, 30);	
DrawPie()	扇形	g.DrawPie(pen,100, 100, 50, 30, 270, 200);	
DrawPolygon()	多边形	Point point1 = new Point(80, 20); Point point2 = new Point(40, 50); Point point3 = new Point(80, 80); Point point4 = new Point(120, 80); Point point5 = new Point(160, 50); Point point6 = new Point(120, 20); Point[] points = { point1, point2, point3, point4, point5, point6 }; g.DrawPolygon(pen, points);	
DrawBezier()	贝塞尔曲线	g.DrawBezier(pen, 10, 50, 30, 80, 10, 10, 50, 80);	
DrawRectangle()	矩形	g.DrawRectangle(pen,10, 10, 100, 50);	
DrawString()	文本	g.DrawString(" 外星人 ", new Font(" 楷体 ", 16), brush, 10, 10);	外星人
FillPie()	实心扇形	g.FillPie(brush, 100, 100, 50, 30, 270, 200);	
FillEllipse()	实心椭圆形	g.FillEllipse(brush, 10, 10, 50, 30);	
FillPolygon ()	实心多边形	Point point1 = new Point(80, 20); Point point2 = new Point(40, 50); Point point3 = new Point(80, 80); Point point4 = new Point(120, 80); Point point5 = new Point(160, 50); Point point6 = new Point(120, 20); Point[] points = { point1, point2, point3, point4, point5, point6 }; g.FillPolygon(brush, points);	
FillRectangle()	实心矩形	g.FillRectangle(brush, 10, 10, 100, 50);	

学习笔记

表 13.3 中“举例”一栏中的 g 表示 Graphics 对象。

13.3.1 绘制图形

Graphics 类中绘制几何图形的方法都是以 Draw 开头的，下面通过一个具体的示例演示如何绘制图形。

示例 1．绘制验证码

绘制验证码程序开发步骤如下。

步骤 1，新建一个 Windows 窗体应用程序，在默认窗体 Form1 中添加一个 PictureBox 控件，用来显示图形验证码；添加一个 Button 控件，用来生成图形验证码。

步骤 2，自定义一个 CheckCode 方法，主要使用 Random 类随机生成 4 位验证码，代码如下。

```
01 private string CheckCode()                              // 使用此方法生成验证码
02 {
03     int number;
04     char code;
05     string checkCode = String.Empty; // 声明变量存储随机生成的 4 位英文字母或数字
06     Random random = new Random();                       // 生成随机数
07     for (int i = 0; i < 4; i++)
08     {
09         number = random.Next();                         // 返回非负随机数
10         if (number % 2 == 0)                            // 判断数字是否为偶数
11             code = (char)('0' + (char)(number % 10));
12         else                                            // 如果不是偶数
13             code = (char)('A' + (char)(number % 26));
14         checkCode += " " + code.ToString();             // 累加字符串
15     }
16     return checkCode;                                   // 返回生成的字符串
17 }
```

步骤 3，自定义一个 CodeImage 方法，用来将生成的验证码绘制成图片，并将该图片显示在 PictureBox 控件中。CodeImage 方法中有一个 string 类型的参数，用来标识要绘制成图片的验证码。代码如下。

```
01 private void CodeImage(string checkCode)
02 {
03     if (checkCode == null || checkCode.Trim() == String.Empty)
```

```
04          return;
05      Bitmap image = new Bitmap((int)Math.Ceiling((checkCode.Length * 9.5)), 22);
06      Graphics g = Graphics.FromImage(image); // 创建 Graphics 对象
07      try
08      {
09          Random random = new Random();          // 生成随机生成器
10          g.Clear(Color.White);                  // 清空图片背景色
11          for (int i = 0; i < 3; i++)            // 画图片的背景噪音线
12          {
13              int x1 = random.Next(image.Width);
14              int x2 = random.Next(image.Width);
15              int y1 = random.Next(image.Height);
16              int y2 = random.Next(image.Height);
17              g.DrawLine(new Pen(Color.Black), x1, y1, x2, y2);
18          }
19          Font font = new Font("Arial", 12, (FontStyle.Bold));
20          g.DrawString(checkCode, font, new SolidBrush(Color.Red), 2, 2);
21          for (int i = 0; i < 150; i++)              // 画图片的前景噪音点
22          {
23              int x = random.Next(image.Width);
24              int y = random.Next(image.Height);
25              image.SetPixel(x, y, Color.FromArgb(random.Next()));
26          }
27          // 画图片的边框线
28          g.DrawRectangle(new Pen(Color.Silver), 0, 0, image.Width - 1, image.
Height - 1);
29          this.pictureBox1.Width = image.Width;      // 设置 pictureBox1 的宽度
30          this.pictureBox1.Height = image.Height;    // 设置 pictureBox1 的高度
31          this.pictureBox1.BackgroundImage = image;       // 设置 pictureBox1 的背景图像
32      }
33      catch
34      { }
35  }
```

步骤 4，在窗体的加载事件和 Button 控件的 Click 事件中分别调用 CodeImage 方法绘制验证码，代码如下。

```
01  private void Form1_Load(object sender, EventArgs e)
02  {
03      CodeImage(CheckCode());
04  }
05  private void button1_Click(object sender, EventArgs e)
06  {
07      CodeImage(CheckCode());
08  }
```

绘制验证码程序运行结果如图 13.3 所示。

图 13.3　绘制验证码程序运行结果

13.3.2　填充图形

Graphics 类中填充几何图形的方法都是以 Fill 开头的，下面通过一个具体的示例演示以 Fill 开头的方法在实际开发中的应用。

示例 2．利用饼形图分析产品市场占有率

利用饼形图分析产品市场占有率程序开发步骤如下。

步骤 1，新建一个 Windows 窗体应用程序，在默认窗体 Form1 中添加两个 Panel 控件，分别用来显示绘制的饼形图和说明信息。

步骤 2，自定义一个 showPic 方法，该方法主要用来绘制饼形图。然后在 Form1 窗体的 Paint 事件中获取数据表中的相应数据，并且调用 showPic 方法绘制饼形图。主要代码如下。

```
01 private void showPic(float f, Brush B)
02 {
03      Graphics g = this.panel1.CreateGraphics();// 通过 panel1 控件创建一个
Graphics 对象
04      if (TimeNum == 0.0f)
05      {
06          // 绘制扇形
07          g.FillPie(B, 0, 0, this.panel1.Width, this.panel1.Height, 0, f *
360);
08      }
09      else
10      {
11          g.FillPie(B, 0, 0, this.panel1.Width, this.panel1.Height, TimeNum,
f * 360);
12      }
13      TimeNum += f * 360;
14 }
15 private void Form1_Paint(object sender, PaintEventArgs e) // 在 Paint 事件中绘制
16 {
17      ht.Clear();
18      Conn();                                                    // 连接数据库
```

```
19      Random rnd = new Random();                          // 生成随机数
20      using (cmd = new SqlCommand("select t_Name,sum(t_Num) as Num  from tb_
product group by t_Name", con))
21      {
22          // 通过 panel2 控件创建一个 Graphics 对象
23          Graphics g2 = this.panel2.CreateGraphics();
24          SqlDataReader dr = cmd.ExecuteReader();     // 创建 SqlDataReader 对象
25          while (dr.Read())                           // 读取数据
26          {
27              ht.Add(dr[0], Convert.ToInt32(dr[1]));  // 将数据添加到 Hashtable 中
28          }
29          float[] flo = new float[ht.Count];
30          int T = 0;
31          foreach (DictionaryEntry de in ht)          // 遍历 Hashtable
32          {
33              flo[T] = Convert.ToSingle((Convert.ToDouble(de.Value) / SumNum).
ToString().Substring(0, 6));
34              Brush Bru = new SolidBrush(Color.FromArgb(rnd.Next(255), rnd.
Next(255), rnd.Next(255)));
35              g2.DrawString(de.Key + "  " + flo[T] * 100 + "%", new Font("Arial", 8,
FontStyle.Regular), Bru, 7, 5 + T * 18);            // 绘制商品及百分比
36              showPic(flo[T], Bru);                   // 调用 showPic 方法绘制饼形图
37              T++;
38          }
39      }
40  }
```

利用饼形图分析产品市场占有率程序运行结果如图 13.4 所示。

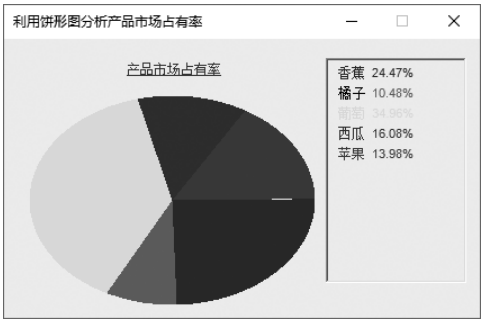


图 13.4　利用饼形图分析产品市场占有率程序运行结果

13.4　绘制图像

Graphics 类不仅可以绘制几何图形和文本，还可以绘制图像。在利用 Graphics 类绘制

图像时需要使用 DrawImage 方法，该方法可以在由一对坐标指定的位置以图像尺寸的原始大小或指定大小绘制图像。DrawImage 方法有多种使用形式，其常用语法格式如下。

```
public void DrawImage(Image image,int x,int y)
public void DrawImage(Image image,int x,int y,int width,int height)
```

DrawImage 方法的参数说明如表 13.4 所示。

表 13.4　DrawImage 方法的参数说明

参　数	说　明
image	要绘制的 Image
x	所绘制图像的左上角的 *x* 轴坐标
y	所绘制图像的左上角的 *y* 轴坐标
width	所绘制图像的宽度
height	所绘制图像的高度

示例 3．绘制公司 Logo

新建一个 Windows 窗体应用程序，触发默认窗体 Form1 的 Paint 事件，在该事件中创建 Graphics 对象，并调用 DrawImage 方法将公司的 Logo 图片绘制到窗体中，代码如下。

```
01 private void Form1_Paint(object sender, PaintEventArgs e)
02 {
03     Image myImage = Image.FromFile("logo.jpg");          // 创建 Image 对象
04     Graphics myGraphics = this.CreateGraphics();         // 创建 Graphics 对象
05     myGraphics.DrawImage(myImage, 50, 20, 90, 92);       // 绘制图像
06 }
```

学习笔记

logo.jpg 文件需要存放到项目的 Debug 文件夹中。

绘制公司 Logo 程序运行结果如图 13.5 所示。

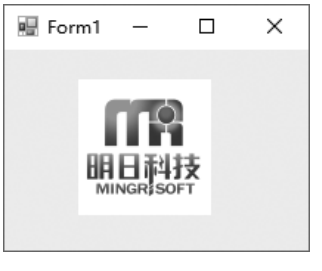


图 13.5　绘制公司 Logo 程序运行结果

第 14 章　Socket 网络编程

Internet 提供了大量、多样的信息，很少有人能在接触过 Internet 后拒绝它的诱惑。计算机网络实现了多个计算机互联系统，相互连接的计算机之间彼此能够进行数据交流。网络应用程序就是在已连接的不同计算机上运行的程序，这些程序之间可以相互交换数据。而编写网络应用程序，首先必须明确网络应用程序所要使用的网络协议。在这些协议中，TCP/IP 协议是网络应用程序的首选。本章将从介绍网络协议开始，向读者介绍 TCP 网络程序和 UDP 网络程序。

14.1　计算机网络基础

14.1.1　局域网与广域网

计算机网络分为局域网与广域网。通常所说的局域网（Local Area Network，LAN），是指在某一区域内由多台计算机通过一定形式连接起来的计算机组。局域网可以由两台计算机组成，也可以由同一区域内的上千台计算机组成，如图 14.1 所示。由局域网延伸到更大的范围，这样的网络称为广域网（Wide Area Network，WAN），大家熟悉的因特网（Internet）就是由无数的局域网和广域网组成的，如图 14.2 所示。

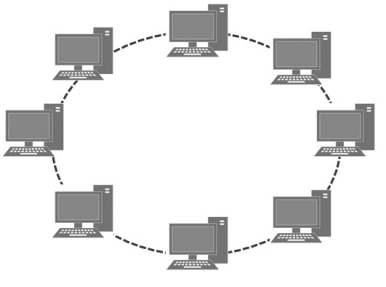


图 14.1　局域网示意图

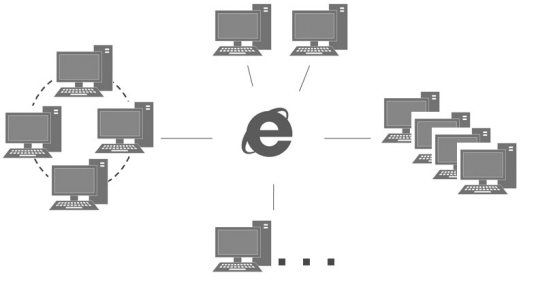


图 14.2　广域网示意图

14.1.2 网络协议

网络协议规定了计算机之间连接的物理、机械（网线与网卡的连接规定）、电气（有效的电平范围）等特征，以及计算机之间的相互寻址规则、数据发送冲突的解决、长数据如何分段传送与接收等。就像不同的国家有不同的法律一样，网络协议也有多种，下面介绍几种常用的网络协议。

1. IP 协议

IP 其实是 Internet Protocol 的简称，由此可知它是一种“网络协议”。因特网网络采用的协议是 TCP/IP 协议，其全称是 Transmission Control Protocol/Internet Protocol 协议。因特网依靠 TCP/IP 协议在全球范围内实现了不同硬件结构、不同操作系统、不同网络系统的互联。在因特网上存在数以亿计的主机，每台主机在网络上通过为其分配的因特网地址表示自己，这个地址就是 IP 地址。IP 地址用 4 个字节，即 32 位的二进制数来表示，称为 IPv4。为了便于使用，通常取用每个字节的十进制数，并且每个字节之间用圆点隔开来表示 IP 地址，如 192.168.1.1。人们正在试验使用 16 个字节来表示 IP 地址，这就是 IPv6，但 IPv6 还没有投入使用。

IP 相当于计算机在网络中的身份证，它必须是唯一的，如图 14.3 所示。

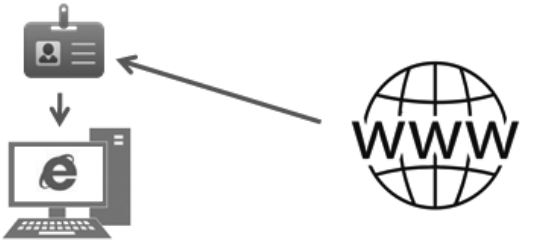


图 14.3　IP 相当于计算机在网络中的身份证

2. TCP 协议

在网络协议栈中，有两个高级协议是网络应用程序编写者应该了解的，分别是 TCP 协议（Transmission Control Protocol，传输控制协议）与 UDP 协议（用户数据报协议）。

TCP 协议是一种以固接连线为基础的协议，可供两台计算机间进行可靠的数据传送。TCP 协议可以保证从一端将数据传送至连接的另一端时，数据能够准确送达，而且送达的数据的排列顺序和送出时的顺序相同，其示意图如图 14.4 所示。TCP 协议适合可靠性要求比较高的场合，它就像接打电话一样，必须先拨号给对方，等两端确定连接后，通话双方能听到对方说话，也知道对方回应的是什么，如图 14.5 所示。

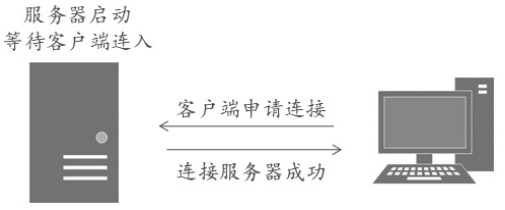


图 14.4　TCP 协议示意图

图 14.5　TCP 协议类似于现实生活中的接打电话

3. UDP 协议

UDP 协议是无连接通信协议，它不能保证可靠的数据传输，但能够向若干个目标发送数据，接收发自若干个源的数据，其示意图如图 14.6 所示。UDP 协议是以独立发送数据包的方式进行的，它就像生活中的广播，村长一发通知，大喇叭一喊，田里耕地的人都能听见，但在家睡觉的人可能就没听见，而哪些人听见了，哪些人没听见，发通知的这个村长是不知道的，如图 14.7 所示。

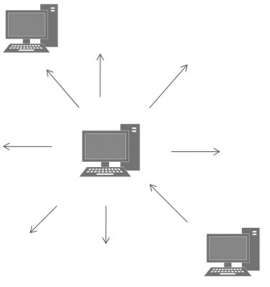


图 14.6　UDP 协议示意图

图 14.7　UDP 协议类似于现实生活中的广播

学习笔记

一些防火墙和路由器会被设置成不允许 UDP 数据包传输，因此若遇到 UDP 连接方面的问题，则应先判断是否允许 UDP 协议。

14.1.3　端口及套接字

一般而言，一台计算机只有单一的连接到网络的“物理连接（Physical Connection）”，所有的数据都通过此连接对内、对外送达特定的计算机，这就是端口。网络程序设计中的端口（Port）并非真实的物理存在，而是一个假想的连接装置。端口被规定为一个 0 ～ 65535 的整数，而 HTTP 服务一般使用 80 端口，FTP 服务使用 21 端口。假如一台计算机提供了 HTTP、FTP 等多种服务，则客户机将通过不同的端口来确定连接到服务器的哪项服务上。

计算机中的端口示意图如图 14.8 所示。

学习笔记

0 ~ 1023 的端口号通常用于一些比较知名的网络服务和应用，普通网络应用程序则应该使用 1024 以上的端口号，以避免该端口号被另一个应用或系统服务所占用。

网络程序中的套接字（Socket）用于将程序与网络连接起来。套接字是一个假想的连接装置，就像用于连接电器与电线的插座，如图 14.9 所示。C# 将套接字抽象化为类，程序设计者只需要创建 Socket 类对象即可使用套接字。

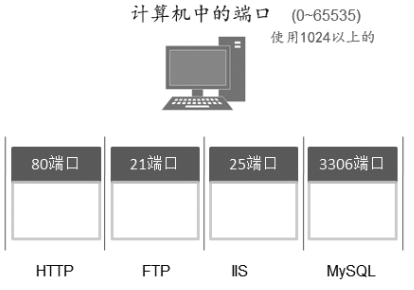


图 14.8 计算机中的端口示意图

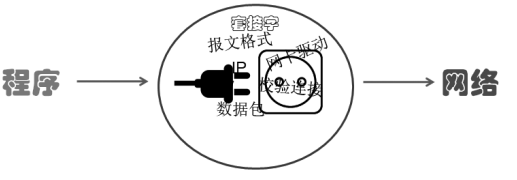


图 14.9 套接字示意图

14.2 IP 地址封装

IP 地址是每个计算机在网络中的唯一标识，它是 32 位或 128 位的无符号数字，使用 4 组数字表示一个固定的编号，如“192.168.128.255”就是局域网中的编号。

IP 地址是一种低级协议，TCP 协议和 UDP 协议都是在它的基础上构建的。

C# 提供了 IP 地址相关的类，包括 Dns 类、IPAddress 类、IPHostEntry 类等，它们都位于 System.Net 命名空间中，下面分别对这 3 个类进行介绍。

1. Dns 类

Dns 类是一个静态类，它通过 DNS（因特网域名系统）检索关于特定主机的信息。在 IPHostEntry 类的实例中返回来自 DNS 查询的主机信息。如果指定的主机在 DNS 中有多个入口，则 IPHostEntry 包含多个 IP 地址和别名。Dns 类的常用方法及说明如表 14.1 所示。

表 14.1　Dns 类的常用方法及说明

方　法	说　明
BeginGetHostAddresses	异步返回指定主机的 IP 地址
BeginGetHostByName	开始异步请求关于指定 DNS 主机名的 IPHostEntry 信息
EndGetHostAddresses	结束对 DNS 信息的异步请求
EndGetHostByName	结束对 DNS 信息的异步请求
EndGetHostEntry	结束对 DNS 信息的异步请求
GetHostAddresses	返回指定主机的 IP 地址
GetHostByAddress	获取 IP 地址的 DNS 主机信息
GetHostByName	获取指定 DNS 主机名的 DNS 信息
GetHostEntry	将主机名或 IP 地址解析为 IPHostEntry 实例
GetHostName	获取本地计算机的主机名

2. IPAddress 类

IPAddress 类包含计算机在 IP 网络上的地址，主要用来提供 IP 地址。IPAddress 类的常用字段、属性、方法及说明如表 14.2 所示。

表 14.2　IPAddress 类的常用字段、属性、方法及说明

字段、属性及方法	说　明
Any 字段	提供一个 IP 地址，指示服务器应侦听所有网络接口上的客户端活动。此字段为只读
Broadcast 字段	提供 IP 广播地址。此字段为只读
Loopback 字段	提供 IP 环回地址。此字段为只读
Address 属性	IP 地址
AddressFamily 属性	获取 IP 地址的地址族
IsIPv6LinkLocal 属性	获取地址是否为 IPv6 链接本地地址
IsIPv6SiteLocal 属性	获取地址是否为 IPv6 站点本地地址
GetAddressBytes 方法	以字节数组形式提供 IPAddress 的副本
Parse 方法	将 IP 地址字符串转换为 IPAddress 实例
TryParse 方法	判断字符串是否为有效的 IP 地址

3. IPHostEntry 类

IPHostEntry 类用来为因特网主机地址信息提供容器类，其常用属性及说明如表 14.3 所示。

表 14.3　IPHostEntry 类的常用属性及说明

属　性	说　明
AddressList	获取或设置与主机关联的 IP 地址列表
Aliases	获取或设置与主机关联的别名列表
HostName	获取或设置主机的 DNS 名称

学习笔记

IPHostEntry 类通常和 Dns 类一起使用。

示例 1．访问同一局域网中的主机的名称

使用 Dns 类的相关方法获得本地主机的本机名和 IP 地址，然后访问同一局域网中的 IP 地址从“192.168.1.50”至“192.168.1.60”范围内的所有可访问的主机的名称（如果对方没有安装防火墙，并且网络连接正常，则都可以访问），代码如下。

```
01 private void Form1_Load(object sender, EventArgs e)
02 {
03     string IP, name, localip = "127.0.0.1";
04     string localname = Dns.GetHostName();                    // 获取本机名
05     IPAddress[] ips = Dns.GetHostAddresses(localname);    // 获取所有 IP 地址
06     foreach(IPAddress ip in ips)
07     {
08         if(!ip.IsIPv6SiteLocal)                          // 如果不是 IPv6 地址
09             localip = ip.ToString();                     // 获取本机 IP 地址
10     }
11     // 将本机名和 IP 地址输出
12     label1.Text += " 本机名：" + localname + "  本机 IP 地址：" + localip;
13     for (int i = 50; i <= 60; i++)
14     {
15         IP = "192.168.1." + i;                           // 生成 IP 字符串
16         try
17         {
18             IPHostEntry host = Dns.GetHostEntry(IP);      // 获取 IP 封装对象
19             name = host.HostName.ToString();       // 获取指定 IP 地址的主机名
20             label1.Text += "\nIP 地址 " + IP + " 的主机名称是：" + name;
21         }
22         catch (Exception ex)
23         {
24             MessageBox.Show(ex.Message);
25         }
26     }
27 }
```

访问同一局域网中的主机的名称程序运行结果如图 14.10 所示。

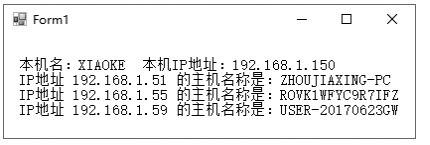


图 14.10　访问同一局域网中的主机的名称程序运行结果

学习笔记

如果想在没有联网的情况下访问本地主机，则可以使用本地回送地址“127.0.0.1”。

14.3　TCP 程序设计

TCP 是一种面向连接的、可靠的、基于字节流的传输层通信协议。在 C# 中，TCP 程序设计是指利用 Socket 类、TcpClient 类和 TcpListener 类编写网络通信程序，这 3 个类都位于 System.Net.Sockets 命名空间中。利用 TCP 协议进行通信的两个应用程序是有主次之分的，一个称为服务器端程序，另一个称为客户端程序。

14.3.1　Socket 类

Socket 类为网络通信提供了一套丰富的方法和属性，它主要用于管理连接，实现 Berkeley 通信端套接字接口；同时它还定义了绑定、连接网络端点及传输数据所需要的各种方法，提供处理端点连接、传输等细节所需要的功能。TcpClient 和 UdpClinet 等类在内部使用 Socket 类。Socket 类的常用属性及说明如表 14.4 所示。

表 14.4　Socket 类的常用属性及说明

属　性	说　明
AddressFamily	获取套接字的地址族
Available	获取已经从网络接收且可供读取的数据量
Connected	获取一个值，该值指示套接字是在上次执行发送操作还是执行接收操作时连接到远程主机
Handle	获取套接字的操作系统句柄
LocalEndPoint	获取本地终结点
ProtocolType	获取套接字的协议类型
RemoteEndPoint	获取远程终结点
SendTimeout	获取或设置一个值，该值指定之后同步 Send 调用将超时的时间长度

Socket 类的常用方法及说明如表 14.5 所示。

表 14.5　Socket 类的常用方法及说明

方　法	说　明
Accept	为新建连接创建新的套接字
BeginAccept	开始一个异步操作来接收一个传入的连接尝试
BeginConnect	开始一个对远程主机连接的异步请求
BeginDisconnect	开始异步请求从远程终结点断开连接
BeginReceive	开始从连接的套接字中异步接收数据
BeginSend	将数据异步发送到连接的套接字
BeginSendFile	将文件异步发送到连接的套接字
BeginSendTo	向特定远程主机异步发送数据
Close	关闭套接字连接并释放所有关联的资源
Connect	建立与远程主机的连接
Disconnect	关闭套接字连接并允许重用套接字
EndAccept	异步接收传入的连接尝试
EndConnect	结束挂起的异步连接请求
EndDisconnect	结束挂起的异步断开连接请求
EndReceive	结束挂起的异步读取
EndSend	结束挂起的异步发送
EndSendFile	结束文件的挂起异步发送
EndSendTo	结束挂起的、向指定位置进行的异步发送
Listen	将套接字置于侦听状态
Receive	接收来自绑定的套接字的数据
Send	将数据发送到连接的套接字
SendFile	将文件和可选数据异步发送到连接的套接字
SendTo	将数据发送到特定终结点
Shutdown	禁用某 Socket 上的发送和接收

14.3.2　TcpClient 类和 TcpListener 类

TcpClient 类用于在同步阻止模式下通过网络来连接、发送和接收流数据。为了使 TcpClient 连接并交换数据，TcpListener 实例或 Socket 实例必须侦听是否有传入的连接请求。可以使用下面两种方法连接到该侦听器。

（1）创建一个 TcpClient，并调用 Connect 方法连接。

（2）使用远程主机的主机名和端口号创建 TcpClient，此构造函数将自动尝试一个连接。

TcpListener 类用于在同步阻止模式下侦听和接收传入的连接请求。可使用 TcpClient 类或 Socket 类来连接 TcpListener，并且可以使用 IPEndPoint、本地 IP 地址及端口号或仅使用端口号来创建 TcpListener 实例对象。

TcpClient 类的常用属性、方法及说明如表 14.6 所示。

表 14.6　TcpClient 类的常用属性、方法及说明

属性及方法	说　明
Available 属性	获取已经从网络接收且可供读取的数据量
Client 属性	获取或设置基础套接字
Connected 属性	获取一个值，该值指示 TcpClient 的基础套接字是否已连接到远程主机
ReceiveBufferSize 属性	获取或设置接收缓冲区的大小
ReceiveTimeout 属性	获取或设置在初始化一个读取操作后 TcpClient 等待接收数据的时间量
SendBufferSize 属性	获取或设置发送缓冲区的大小
SendTimeout 属性	获取或设置 TcpClient 等待发送操作成功完成的时间量
BeginConnect 方法	开始一个对远程主机连接的异步请求
Close 方法	释放此 TcpClient 实例，而不关闭基础连接
Connect 方法	使用指定的主机名和端口号将客户端连接到 TCP 主机
EndConnect 方法	异步接收传入的连接尝试
GetStream 方法	返回用于发送和接收数据的 NetworkStream

TcpListener 类的常用属性、方法及说明如表 14.7 所示。

表 14.7　TcpListener 类的常用属性、方法及说明

属性及方法	说　明
LocalEndpoint 属性	获取当前 TcpListener 的基础 EndPoint
Server 属性	获取基础网络套接字
AcceptSocket/AcceptTcpClient 方法	接收挂起的连接请求
BeginAcceptSocket/BeginAcceptTcpClient 方法	开始一个异步操作来接收一个传入的连接尝试
EndAcceptSocket 方法	异步接收传入的连接尝试，并创建新的套接字来处理远程主机通信
EndAcceptTcpClient 方法	异步接收传入的连接尝试，并创建新的 TcpClient 来处理远程主机通信
Start 方法	开始侦听传入的连接请求
Stop 方法	关闭侦听器

14.3.3　TCP 网络程序示例

示例 2．客户端 / 服务器端的交互

客户端 / 服务器端的交互程序的设计包括服务器端的设计和客户端的设计。

服务器端的设计如下。

创建服务器端项目 Server，在 Main 方法中创建 TCP 连接对象；监听客户端接入，并读取接入的客户端 IP 地址和传入的消息；向接入的客户端发送一条消息。代码如下。

```
01 namespace Server
02 {
03     class Program
04     {
05         static void Main()
06         {
07             int port = 888;                          // 端口
08             TcpClient tcpClient;                     // 创建 TCP 连接对象
09             // 定义 IP 地址
10             IPAddress[] serverIP = Dns.GetHostAddresses("127.0.0.1");
11             IPAddress localAddress = serverIP[0]; //IP 地址
12             // 监听套接字
13               TcpListener tcpListener = new TcpListener(localAddress, port);
14             tcpListener.Start();                     // 开始监听
15             Console.WriteLine(" 服务器启动成功，等待用户接入 ...");  // 输出消息
16             while (true)
17             {
18                 try
19                 {
20                     // 每接收一个客户端则生成一个 TcpClient
21                     tcpClient = tcpListener.AcceptTcpClient();
22                       // 获取网络数据流
23                     NetworkStream networkStream = tcpClient.GetStream();
24                     // 定义流数据读取对象
25                     BinaryReader reader = new BinaryReader(networkStream);
26                     // 定义流数据写入对象
27                     BinaryWriter writer = new BinaryWriter(networkStream);
28                    while (true)
29                     {
30                         try
31                         {
32                          string strReader = reader.ReadString();// 接收消息
33                          // 截取客户端消息
34                          string[] strReaders = strReader.Split(new char[] { ' ' });
35                          // 输出接收的客户端 IP 地址
36                          Console.WriteLine(" 有客户端接入，客户 IP: " + strReaders[0]);
37                             // 输出接收的消息
38                             Console.WriteLine(" 来自客户端的消息: " + strReaders[1]);
39                             // 定义服务器端要写入的消息
40                             string strWriter = " 我是服务器，欢迎光临 ";
41                             writer.Write(strWriter);  // 向对方发送消息
```

```
                        }
                        catch
                        {
                            break;
                        }
                    }
                }
                catch
                {
                    break;
                }
            }
        }
    }
}
```

客户端的设计如下。

创建客户端项目 Client，在 Main 方法中创建 TCP 连接对象，以指定的地址和端口连接服务器；向服务器端发送数据和接收服务器端传输的数据。代码如下。

```
namespace Client
{
    class Program
    {

        static void Main(string[] args)
        {
            // 创建一个 TcpClient 对象，自动分配主机 IP 地址和端口号
            TcpClient tcpClient = new TcpClient();
            // 连接服务器，其 IP 地址和端口号分别为 127.0.0.1 和 888
            tcpClient.Connect("127.0.0.1", 888);
            if (tcpClient != null)                          // 判断是否连接成功
            {
                Console.WriteLine(" 连接服务器成功 ");
                // 获取数据流
                NetworkStream networkStream = tcpClient.GetStream();
                // 定义流数据读取对象
                BinaryReader reader = new BinaryReader(networkStream);
                // 定义流数据写入对象
                BinaryWriter writer = new BinaryWriter(networkStream);
                string localip="127.0.0.1"; // 存储本机 IP 地址，默认值为 127.0.0.1
                // 获取所有 IP 地址
                IPAddress[] ips = Dns.GetHostAddresses(Dns.GetHostName());
                foreach (IPAddress ip in ips)
                {
                    if (!ip.IsIPv6SiteLocal)                // 如果不是 IPv6 地址
```

```
27                          localip = ip.ToString();          // 获取本机 IP 地址
28                  }
29                  writer.Write(localip + " 你好服务器，我是客户端 ");// 向服务器端发送消息
30                  while (true)
31                  {
32                      try
33                      {
34                          // 接收服务器端发送的数据
35                          string strReader = reader.ReadString();
36                          if (strReader != null)
37                          {
38                              // 输出接收的消息
39                              Console.WriteLine(" 来自服务器的消息："+strReader);
40                          }
41                      }
42                      catch
43                      {
44                          break;        // 在接收过程中如果出现异常，则退出循环
45                      }
46                  }
47              }
48              Console.WriteLine(" 连接服务器失败 ");
49          }
50      }
51  }
```

首先运行服务器端，然后运行客户端，客户端运行后的服务器端效果如图 14.11 所示，客户端运行效果如图 14.12 所示。

图 14.11　客户端运行后的服务器端效果

图 14.12　客户端运行效果

14.4　UDP 程序设计

UDP 是 User Datagram Protocol 的简称，中文名是用户数据报协议，它是网络信息传输的另一种形式。UDP 协议通信和 TCP 协议通信不同，基于 UDP 协议通信的信息传递更快，但不提供可靠的保证。在使用 UDP 协议传递数据时，用户无法知道数据能否正确地到达主机，也不能确定到达目的地的顺序是否和发送的顺序相同。虽然 UDP 协议是一种不可

靠的协议，但如果需要较快地传输信息，并能容忍小的错误，那么可以考虑使用 UDP 协议。

基于 UDP 协议通信的基本模式如下。

（1）将数据打包（称为数据包），然后将数据包发往目的地。

（2）接收别人发来的数据包，然后查看数据包。

14.4.1　UdpClient 类

在 C# 中，UdpClient 类用于在同步阻止模式下发送和接收无连接 UDP 数据报。因为 UDP 协议是无连接传输协议，所以不需要在发送和接收数据前建立远程主机连接，但可以选择使用下面两种方法来建立默认远程主机。

（1）使用远程主机名和端口号作为参数创建 UdpClient 类的实例。

（2）创建 UdpClient 类的实例，然后调用 Connect 方法。

UdpClient 类的常用属性、方法及说明如表 14.8 所示。

表 14.8　UdpClient 类的常用属性、方法及说明

属性及方法	说　明
Available 属性	获取从网络接收的可读取的数据量
Client 属性	获取或设置基础网络套接字
BeginReceive 方法	从远程主机异步接收数据报
BeginSend 方法	将数据报异步发送到远程主机
Close 方法	关闭 UDP 连接
Connect 方法	建立默认远程主机
EndReceive 方法	结束挂起的异步接收
EndSend 方法	结束挂起的异步发送
Receive 方法	返回已由远程主机发送的 UDP 数据报
Send 方法	将 UDP 数据报发送到远程主机

14.4.2　UDP 网络程序示例

根据前面所讲的网络编程的基础知识，以及 UDP 网络编程的特点，下面创建一个广播数据报程序。广播数据报是一种较新的技术，类似于电台广播，广播电台需要在指定的波段和频率上广播信息，收听者也要将收音机调到指定的波段、频率才可以收听广播内容。

示例 3．广播数据报程序

本示例要求主机不断地重复播出节目预报，这样可以保证加入同一组的主机随时能接收到广播信息。接收者将正在接收的信息放在一个文本框中，并将接收的全部信息放在另一个文本框中。

创建广播主机项目 Server（控制台应用程序），在 Main 方法中创建 UDP 连接对象；然后通过 UDP 连接对象不断向外发送广播信息。代码如下。

```
01  namespace Server
02  {
03      class Program
04      {
05          static UdpClient udp = new UdpClient();    // 创建 UdpClient 对象
06          static void Main(string[] args)
07          {
08              // 调用 UdpClient 对象的 Connect 方法建立默认远程主机
09              udp.Connect("127.0.0.1", 888);
10              while (true)
11              {
12                  Thread thread = new Thread(() =>
13                  {
14                      while (true)
15                      {
16                          try
17                          {
18
19                              // 定义一个字节数组，用来存放发送到远程主机的信息
20                              Byte[] sendBytes = Encoding.Default.GetBytes("("
+ DateTime.Now.ToLongTimeString() + ") 节目预报：八点有大型晚会，请收听 ");
21                               Console.WriteLine("(" + DateTime.Now.ToLongTimeString()
+ ") 节目预报：八点有大型晚会，请收听 ");
22                              // 调用 UdpClient 对象的 Send 方法将 UDP 数据报发送到远程主机
23                              udp.Send(sendBytes, sendBytes.Length);
24                              Thread.Sleep(2000);           // 线程休眠 2 秒
25                          }
26                          catch (Exception ex)
27                          {
28                              Console.WriteLine(ex.Message);
29                          }
30                      }
31                  });
32                  thread.Start();                           // 启动线程
33              }
34          }
```

```
35      }
36  }
```

学习笔记

上面代码在实现时用到了 Thread 类，该类表示线程类，其详细使用请参看本书第 15 章。

广播主机程序运行结果如图 14.13 所示。

图 14.13　广播主机程序运行结果

创建接收广播项目 Client（Windows 窗体应用程序），在默认窗体中添加两个 Button 控件和两个 TextBox 控件，并且将两个 TextBox 控件设置为多行文本框。单击“开始接收”按钮，系统开始接收广播主机播出的信息；单击“停止接收”按钮，系统会停止接收广播主机播出的信息。代码如下。

```
01  namespace Client
02  {
03      public partial class Form1 : Form
04      {
05          public Form1()
06          {
07              InitializeComponent();
08              CheckForIllegalCrossThreadCalls = false;// 在其他线程中可以调用主窗体控件
09          }
10          bool flag = true;                                   // 标识是否接收数据
11          UdpClient udp;                                      // 创建 UdpClient 对象
12          Thread thread;                                      // 创建线程对象
13          private void button1_Click(object sender, EventArgs e)
14          {
15              udp = new UdpClient(888);                       // 使用端口号创建 UDP 连接对象
16              flag = true;                                    // 标识接收数据
17              // 创建 IPEndPoint 对象，用来显示响应主机的标识
18              IPEndPoint ipendpoint = new IPEndPoint(IPAddress.Any, 888);
19              thread = new Thread(() =>       // 新开线程，执行接收数据操作
20              {
21                  while(flag)                                 // 如果标识为 true
```

```
22                  {
23                      try
24                      {
25                          if (udp.Available <= 0) continue; //判断是否有网络数据
26                          if (udp.Client == null) return;   //判断连接是否为空
27                          //调用 UdpClient 对象的 Receive 方法获得从远程主机返回的
UDP 数据报
28                          byte[] bytes = udp.Receive(ref ipendpoint);
29                          //将获得的 UDP 数据报转换为字符串形式
30                          string str = Encoding.Default.GetString(bytes);
31                          textBox2.Text = "正在接收的信息：\n" + str;//显示正在接
收的数据
32                          textBox1.Text += "\n" + str;  //显示接收的所有数据
33                      }
34                      catch (Exception ex)
35                      {
36                          MessageBox.Show(ex.Message);      //错误提示
37                      }
38                      Thread.Sleep(2000);                   //线程休眠 2 秒
39                  }
40              });
41              thread.Start();                               //启动线程
42          }
43
44          private void button2_Click(object sender, EventArgs e)
45          {
46              flag = false;                             //标识不接收数据
47              if (thread.ThreadState == ThreadState.Running) //判断线程是否运行
48                  thread.Abort();                       //终止线程
49              udp.Close();                              //关闭连接
50          }
51      }
52  }
```

接收广播程序运行结果如图 14.14 所示。

图 14.14　接收广播程序运行结果

第 15 章　多线程编程技术

如果一次只完成一件事情，那是一个不错的想法，但事实上很多事情都是同时进行的，所以在 C# 中为了模拟这种状态，引入了线程机制。简单来说，当程序同时完成多件事情时，就是所谓的多线程程序。多线程编程技术运用广泛，开发人员可以使用多线程编程技术对要执行的操作进行分段执行，这样可以大大提高程序的运行速度和性能。本章将对多线程编程技术进行详细讲解。

15.1　线程概述

世界上的很多事物都会同时完成很多工作。例如，人体同时进行呼吸、血液循环、思考问题等活动；用户既可以使用计算机听歌，又可以使用它打印文件。这些活动完全可以同时进行，这种思想在 C# 中称为并发，而将并发完成的每一件事情称为线程。本节将对线程进行详细讲解。

15.1.1　线程的定义与分类

在讲解线程之前，需要先了解一个概念，即进程。系统中资源分配和资源调度的基本单位称为进程。进程很常见，如 QQ、Word 等，每个独立执行的程序在系统中都是一个进程。

每个进程中都可以同时包含多个线程。例如，QQ 是一款聊天软件，但其功能有很多，如收 / 发信息、播放音乐、查看网页和下载文件等，这些工作可以同时进行并且互不干扰。这就是使用了线程的并发机制，我们把 QQ 软件看作一个进程，而其每一个功能都是一个可以独立运行的线程。进程与线程的关系如图 15.1 所示。

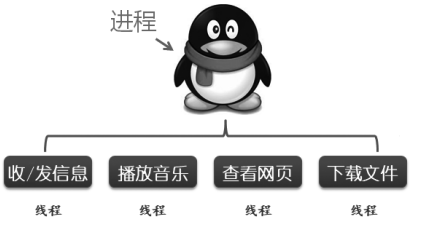


图 15.1　进程与线程的关系

一个进程可以包括多个线程，但计算机的 CPU 只有一个，那么这些线程是怎么做到并发运行的呢？ Windows 操作系统是多任务操作系统，它以进程为单位，每个独立执行的程序称为一个进程。在系统中可以分配给每个进程一段有限的使用 CPU 的时间（也可以称为 CPU 时间片），CPU 在 CPU 时间片内执行某个进程，然后下一个 CPU 时间片又跳至另一个进程中去执行。由于 CPU 转换较快，所以使得每个进程好像同时执行一样。

图 15.2 给出了 Windows 操作系统的执行模式。

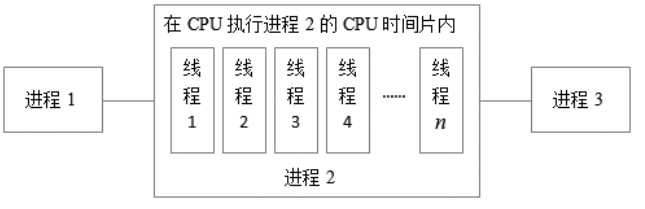


图 15.2　Windows 操作系统的执行模式

一个线程则是进程中的执行流程，一个进程中可以同时包括多个线程，每个线程也可以得到一小段程序的执行时间，这样一个进程就可以具有多个并发执行的线程。

15.1.2　多线程的优 / 缺点

在一般情况下，需要用户交互的软件都必须尽可能快地对用户的操作做出反应，以便提供良好的用户体验，但同时它又必须执行必要的计算，以便尽可能快地将数据呈现给用户，这时可以使用多线程来实现。

1. 多线程的优点

要提高对用户的响应速度，使用多线程是一种十分有效的方式。在具有一个处理器的计算机上，多线程可以通过利用用户事件之间很小的时间段在后台处理数据来达到这种效果。多线程的优点如下。

（1）通过网络与 Web 服务器和数据库进行通信。

（2）方便执行占用大量时间的操作。

（3）区分具有不同优先级的任务。

（4）使用户界面可以在将时间分配给后台任务时仍能快速做出响应。

2. 多线程的缺点

使用多线程有好处，同时也有坏处，建议一般不要在程序中使用太多的线程，这样可以最大限度地减少操作系统资源的使用，并提高性能。使用多线程可能对程序造成负面影响，多线程的缺点如下。

（1）系统将为进程和线程所需要的上、下文信息提供内存。因此，可以创建的进程和线程的数目会受到可用内存的限制。

（2）跟踪大量的线程将占用处理器大量时间。如果线程过多，则其中大多数线程都不会产生明显的进度。如果大多数线程处于一个进程中，则其他进程中的线程的调度频率会很低。

（3）使用多个线程控制代码执行非常复杂，并可能产生许多 Bug。

（4）销毁线程需要了解可能发生的问题并进行处理。

15.2　线程的实现

在 C# 中通过使用 Thread 类来实现线程，本节将对 Thread 类，以及如何创建线程、线程的生命周期进行介绍。

15.2.1　使用 Thread 类创建线程

Thread 类位于 System.Threading 命名空间中，该类主要用于创建并控制线程、设置线程优先级并获取其状态。创建线程需要使用 Thread 类的构造函数，其语法如下。

```
public Thread(ThreadStart start)
public Thread(ParameterizedThreadStart start)
```

start：一个 ThreadStart 委托或 ParameterizedThreadStart 委托，它表示线程开始执行时要调用的方法。

Thread 类的常用属性及说明如表 15.1 所示。

表 15.1　Thread 类的常用属性及说明

属　性	说　明
ApartmentState	获取或设置此线程的单元状态
CurrentContext	获取线程正在其中执行的当前上、下文
CurrentThread	获取当前正在运行的线程
IsAlive	获取一个值，该值指示当前线程的执行状态
ManagedThreadId	获取当前托管线程的唯一标识符
Name	获取或设置线程的名称
Priority	获取或设置一个值，该值指示线程的调度优先级
ThreadState	获取一个值，该值包含当前线程的状态

Thread 类的常用方法及说明如表 15.2 所示。

表 15.2　Thread 类的常用方法及说明

方　法	说　明
Abort	在调用此方法的线程上引发 ThreadAbortException，以开始终止此线程的过程。调用此方法通常会终止线程
Join	阻塞调用线程，直到某个线程终止（或经过）指定时间为止
Sleep	将当前线程挂起（或阻塞）指定的时间
SpinWait	导致线程等待由 iterations 参数定义的时间量
Start	开始执行线程

创建 Thread 类的对象之后，线程对象已存在并已配置，但并未创建实际的线程，只有在调用 Start 方法后，才会创建实际的线程。

Start 方法用来开始执行线程，它有两种重载形式，下面分别对其进行介绍。

（1）导致操作系统将当前实例的状态更改为 ThreadState.Running，语法如下。

```
public void Start ()
```

（2）使操作系统将当前实例的状态更改为 ThreadState.Running，并选择线程执行所需要的方法，语法如下。

```
public void Start (Object parameter)
```

parameter：一个 Object 对象，包含线程执行的方法要使用的数据。

学习笔记

如果线程已经终止，则无法通过再次调用 Start 方法来重新启动线程。

示例 1．向右移动的 C# 图标

创建一个 Windows 窗体应用程序，实现图标移动的功能。在具体实现时，在窗体中添加一个 PictureBox 控件，并设置相应的 C# 图标，然后使用 Thread 线程控制该控件的坐标位置，从而实现移动图标的效果。主要代码如下。

```
01 public partial class Form1 : Form
02 {
03     public Form1()
04     {
05         InitializeComponent();
06         CheckForIllegalCrossThreadCalls = false;  // 使线程可以调用窗体控件
07     }
08     int x = 12;                                   // 定义图标初始横坐标位置
09     void Roll()
10     {
11         while (x <= 260)                          // 设置循环条件
12         {
13               // 将标签的横坐标用变量表示
14             pictureBox1.Location = new Point(x, 12);
15             Thread.Sleep(500);                    // 使线程休眠 500 毫秒
16             x += 4;                               // 使横坐标每次增加 4
17             if (x >= 260)
18             {
19                 x = 12;        // 当图标到达标签的最右边时，使其回到标签最左边
20             }
21         }
22     }
23     private void Form1_Load(object sender, EventArgs e)
24     {
25         Thread th = new Thread(new ThreadStart(Roll));   // 创建线程对象
26         th.Start();                                      // 启动线程
27     }
28 }
```

运行上面代码，C# 图标从初始位置开始移动，如图 15.3 所示；移动中的 C# 图标效果如图 15.4 所示；图标从左向右移动，其移动到窗体最右侧的效果如图 15.5 所示。

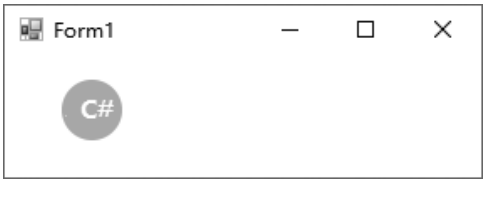


图 15.3　图标初始位置

图 15.4　移动中的 C# 图标效果

图 15.5　图标移动到窗体最右侧的效果

15.2.2　线程的生命周期

任何事物都有始有终。例如，人的一生经历了少年阶段、壮年阶段、老年阶段，最后死亡，这就是一个人的生命周期，如图 15.6 所示。

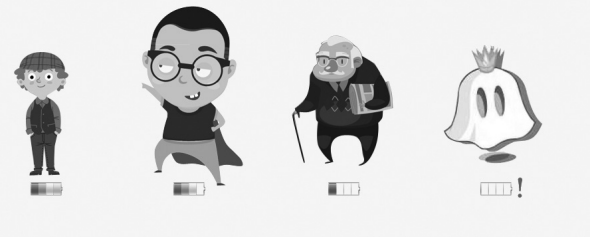


图 15.6　人的生命周期

同样，线程也有自己的生命周期，首先是诞生，即用 new 关键字创建线程对象，这意味着一个线程的诞生，但此时它连声明都没有做；然后线程对象调用 Start 方法，使线程进入就绪状态，也称为可执行状态，此时线程等待 CPU 为其分配 CPU 时间片。当获得系统资源的时候，即 CPU 来执行时，线程就进入了运行状态。

一旦线程进入运行状态，它就会在就绪状态与运行状态下转换，同时也有可能进入暂停状态。如果在运行期间执行了 Sleep、Join 这些方法，或者有外界因素导致线程阻塞（如等待用户输入等），在遇到这些操作场景时，线程就会进入暂停状态。暂停状态与就绪状态不一样，暂停状态下的线程是持有系统资源的，只是没有进行任何操作而已。当休眠时间结束或用户输入完信息后，线程就会从暂停状态回到就绪状态。注意，这里是回到就绪状态，而不是运行状态。因为此时需要检查 CPU 是否有剩余资源来执行线程。当线程中所有的代码都执行完后，调用 Abort 方法终止线程，这个线程就会结束，即进入死亡状态，同时垃圾回收管理器就会回收死亡的线程对象。

图 15.7 描述了线程生命周期的各个状态。

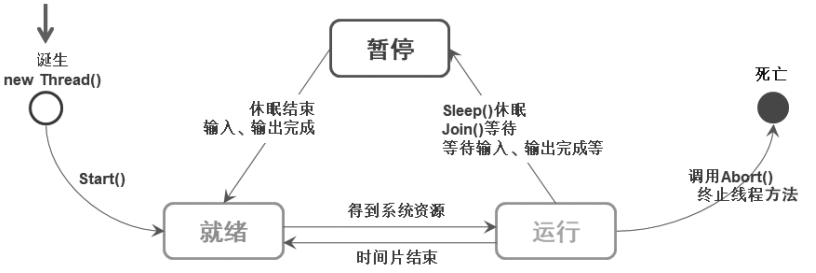


图 15.7　线程生命周期状态图

15.3　操作线程的方法

操作线程有很多方法，这些方法可以使线程从某一种状态过渡到另一种状态。本节将对如何对线程执行休眠、加入和终止等操作进行讲解。

15.3.1　线程的休眠

线程的休眠主要通过 Thread 类的 Sleep 方法实现，该方法用来将当前线程阻塞指定的时间，它有两种重载形式，下面分别对其进行介绍。

（1）将当前线程挂起指定的时间，语法如下。

```
public static void Sleep(int millisecondsTimeout)
```

millisecondsTimeout：线程被阻塞的毫秒数。指定 1 以使其他可能正在等待的线程能够执行；指定 Timeout.Infinite 以无限期阻塞线程。

（2）将当前线程阻塞指定的时间，语法如下。

```
public static void Sleep(TimeSpan timeout)
```

timeout：线程被阻塞的时间量的 TimeSpan。指定持续时间为 1 毫秒，可以使其他可能正在等待的线程能够执行；指定持续时间为 -1 毫秒，可以无限期阻塞线程。

例如，下面代码用来使当前线程休眠 1 秒钟。

```
Thread.Sleep(1000);                                    // 使线程休眠 1 秒钟
```

示例 2．模拟红绿灯变化场景

模拟红绿灯变化场景：红灯亮 8 秒，绿灯亮 5 秒，黄灯亮 2 秒。主要代码如下。

```
01  public partial class Form1 : Form
02  {
```

```
    public Form1()
    {
        InitializeComponent();
        CheckForIllegalCrossThreadCalls = false;  // 使线程可以调用窗体控件
    }
    void ControlLight()
    {
        while (true)
        {                                                  // 线程始终处于被启用状态
            Thread.Sleep(5000);                            // 线程休眠 5 秒
            pictureBox1.Image = Image.FromFile("Yellow.png");   // 黄灯
            Thread.Sleep(2000);                            // 线程休眠 2 秒
            pictureBox1.Image = Image.FromFile("Red.png");      // 红灯
            Thread.Sleep(8000);                            // 线程休眠 8 秒
            pictureBox1.Image = Image.FromFile("Green.png");    // 绿灯
        }
    }
    private void Form1_Load(object sender, EventArgs e)
    {
        Thread th = new Thread(new ThreadStart(ControlLight));// 创建线程对象
        th.Start();                                        // 启动线程
    }
}
```

运行上面代码，绿灯、黄灯和红灯会循环进行显示，绿灯亮如图 15.8 所示，黄灯亮如图 15.9 所示，红灯亮如图 15.10 所示。

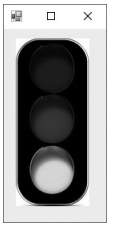


图 15.8　绿灯亮

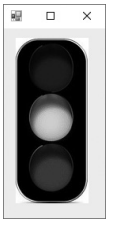


图 15.9　黄灯亮

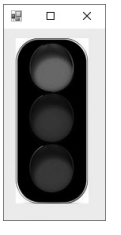


图 15.10　红灯亮

15.3.2　线程的加入

假如当前程序为多线程程序且存在一个线程 A，现在需要插入线程 B，并要求线程 B 执行完毕后再继续执行线程 A，此时可以使用 Thread 类的 Join 方法来实现。这就好比 A 正在看电视，B 突然上门收水费，A 必须付完水费后才能继续看电视。

当某个线程使用 Join 方法加入到另一个线程时，另一个线程会等待该线程执行完毕后再继续执行。

Join 方法用来阻塞调用线程，直到某个线程终止为止，它有 3 种重载形式，下面分别对其进行介绍。

（1）在继续执行标准的 COM 和 SendMessage 消息处理期间，阻塞调用线程，直到某个线程终止为止，语法如下。

```
public void Join ()
```

（2）在继续执行标准的 COM 和 SendMessage 消息处理期间，阻塞调用线程，直到某个线程终止或经过指定时间为止，语法如下。

```
public bool Join (int millisecondsTimeout)
```

- millisecondsTimeout：等待线程终止的毫秒数。
- 返回值：如果线程已终止，则为 true；如果线程在经过 millisecondsTimeout 参数指定的时间后未终止，则为 false。

（3）在继续执行标准的 COM 和 SendMessage 消息处理期间，阻塞调用线程，直到某个线程终止或经过指定时间为止，语法如下。

```
public bool Join (TimeSpan timeout)
```

- timeout：等待线程终止的时间量的 TimeSpan。
- 返回值：如果线程已终止，则为 true；如果线程在经过 timeout 参数指定的时间量后未终止，则为 false。

学习笔记

如果在程序中使用了多线程，辅助线程还没有执行完毕，那么在关闭窗体时，必须要关闭辅助线程，否则会引发异常。

示例 3．控制进度条的滚动

创建一个 Windows 窗体应用程序，默认窗体中包括两个进度条，进度条的进度由线程来控制，通过使用 Join 方法使上面的进度条必须等待下面的进度条运行完成后才可以继续。主要代码如下。

```
01 public partial class Form1 : Form
02 {
03     public Form1()
04     {
05         InitializeComponent();
06         CheckForIllegalCrossThreadCalls = false;  // 使线程可以调用窗体控件
07     }
08     Thread th1, th2;                              // 分别控制进度条 1 和进度条 2
```

```
09      void Pro1()
10      {
11          int count = 0;                                  // 标识何时加入线程 2
12          while (true)
13          {
14              progressBar1.PerformStep();                 // 设置进度条的当前值
15              count += progressBar1.Step;                 // 标识自增
16              Thread.Sleep(100);                          // 使线程 1 休眠 100 毫秒
17              if (count == 20)                            // 当标识为 20 时，执行线程 2
18              {
19                  th2.Join();                             // 使线程 2 调用 Join 方法
20              }
21          }
22      }
23      void Pro2()
24      {
25          int count = 0;                                  // 标识何时执行完
26          while (true)
27          {
28              progressBar2.PerformStep();                 // 设置进度条的当前值
29              count += progressBar2.Step;                 // 标识自增
30              Thread.Sleep(100);                          // 使线程 2 休眠 100 毫秒
31              if (count == 100)                           // 当 count 变量增长为 100 时
32                  break;                                  // 跳出循环
33          }
34      }
35      private void Form1_Load(object sender, EventArgs e)
36      {
37          th1 = new Thread(new ThreadStart(Pro1));    // 创建线程 1 对象
38          th1.Start();                                    // 启动线程 1
39          th2 = new Thread(new ThreadStart(Pro2));    // 创建线程 2 对象
40          th2.Start();                                    // 启动线程 2
41      }
42  }
```

运行程序，两个进度条同时运行，如图 15.11 所示；同时运行至 20 时，上方进度条停止运行，而下方进度条继续运行，如图 15.12 所示；待下方进度条运行完成后，上方进度条继续运行，如图 15.13 所示。

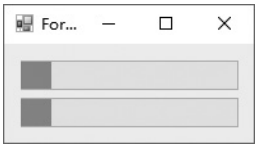


图 15.11　两个进度条同时运行

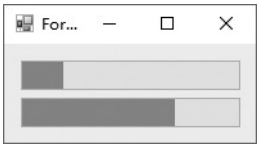


图 15.12　上方进度条等待下方进度条完成

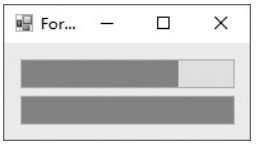


图 15.13　上方进度条继续运行

15.3.3　线程的终止

终止线程使用 Thread 类的 Abort 方法实现，该方法有两种重载形式，下面分别对其进行介绍。

（1）终止线程，在调用此方法的线程上引发 ThreadAbortException 异常，以开始终止此线程的过程，语法如下：

```
public void Abort ()
```

（2）终止线程，在调用此方法的线程上引发 ThreadAbortException 异常，以开始终止此线程并提供有关线程终止的异常信息的过程，语法如下。

```
public void Abort (Object stateInfo)
```

stateInfo：一个 Object 对象，它包含应用程序特定的信息（如状态），该信息可供正被终止的线程使用。

学习笔记

线程的 Abort 方法用于永久地停止托管线程。在调用 Abort 方法时，公共语言运行库在目标线程中引发 ThreadAbortException 异常，目标线程可捕捉此异常。一旦线程终止，它将无法重新启动。

例如，修改示例 3，在关闭窗体时，判断线程 1 和线程 2 是否还在运行；如果是，则调用 Abort 方法将它们关闭。主要代码如下。

```
01 private void Form1_FormClosing(object sender, FormClosingEventArgs e)
02 {
03     if (th1.ThreadState == ThreadState.Running) // 判断线程 1 是否正在运行
04         th1.Abort();                            // 终止线程 1
05     if (th2.ThreadState == ThreadState.Running) // 判断线程 2 是否正在运行
06         th2.Abort();                            // 终止线程 2
07 }
```

15.3.4　线程的优先级

线程的优先级指定一个线程相对于另一个线程的相对优先级，每个线程都有一个分配的优先级。在公共语言运行库内创建的线程最初被分配为 Normal 优先级，而在公共语言运行库外创建的线程，在进入公共语言运行库时将保留其先前的优先级。

线程是根据其优先级而调度执行的，用于确定线程执行顺序的调度算法随操作系统的

不同而不同。在某些操作系统中，具有最高优先级（相对于可执行线程而言）的线程经过调度后总是首先执行。如果具有相同优先级的多个线程都可用，则程序将遍历处于该优先级的线程，并为每个线程提供一个固定的 CPU 时间片来执行。只要具有较高优先级的线程可以执行，具有较低优先级的线程就不会执行。如果在给定的优先级上不再有可执行的线程，则程序将移动到下一个较低的优先级并在该优先级上调度线程以执行。如果具有较高优先级的线程可以执行，则具有较低优先级的线程将被抢先，并允许具有较高优先级的线程再次执行。除此之外，当应用程序的用户界面在前台和后台之间移动时，操作系统还可以动态调整线程优先级。

例如，在多任务操作系统中，每个线程都会得到一小段 CPU 时间片进行执行，在时间结束时，将轮换另一个线程进入执行状态，这时系统会选择与当前线程优先级相同的线程进行执行。系统始终选择就绪状态下优先级较高的线程进入执行状态。图 15.14 给出了处于各个优先级状态下的线程的执行顺序。

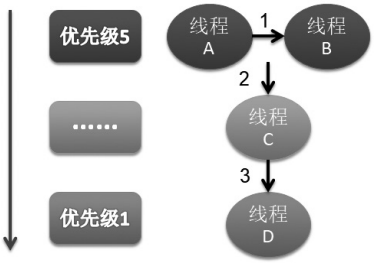


图 15.14　处于各个优先级状态下的线程的执行顺序

在图 15.14 中，优先级为 5 的线程 A 首先得到 CPU 时间片；当该时间结束后，轮换到与线程 A 相同优先级的线程 B；当线程 B 的执行时间结束后，会继续轮换到线程 A；直到线程 A 与线程 B 都执行完毕，才会轮换到线程 C；当线程 C 执行完毕后，才会轮换到线程 D。

学习笔记

多线程的执行本身就是多个线程的交换执行，并非同时执行，通过设置线程优先级的高低，只是说明该线程会优先执行或暂不执行的概率更大一些而已，并不能保证优先级高的线程就一定会比优先级低的线程先执行。

线程的优先级值及说明如表 15.3 所示。

表 15.3　线程的优先级值及说明

优 先 级 值	说　　明
AboveNormal	可以将 Thread 安排在具有 Highest 优先级的线程之后，在具有 Normal 优先级的线程之前
BelowNormal	可以将 Thread 安排在具有 Normal 优先级的线程之后，在具有 Lowest 优先级的线程之前
Highest	可以将 Thread 安排在具有任何其他优先级的线程之前
Lowest	可以将 Thread 安排在具有任何其他优先级的线程之后
Normal	可以将 Thread 安排在具有 AboveNormal 优先级的线程之后，在具有 BelowNormal 优先级的线程之前。在默认情况下，线程具有 Normal 优先级

开发人员可以通过访问线程的 Priority 属性来获取或设置其优先级。Priority 属性用来获取或设置一个值，该值指示线程的调度优先级，其语法如下。

```
public ThreadPriority Priority { get; set; }
```

属性值：ThreadPriority 枚举值之一，默认值为 Normal。

例如，修改示例 3，将线程 1 的优先级设置为最低，将线程 2 的优先级设置为最高，然后再次运行，观察效果是否有变化，主要代码如下。

```
01  th1 = new Thread(new ThreadStart(Pro1));     // 创建线程 1 对象
02  th1.Priority = ThreadPriority.Lowest;        // 设置优先级最低
03  th1.Start();                                 // 启动线程 1
04  th2 = new Thread(new ThreadStart(Pro2));     // 创建线程 2 对象
05  th2.Priority = ThreadPriority.Highest;       // 设置优先级最高
06  th2.Start();                                 // 启动线程 2
```

15.4　线程的同步

在单线程程序中，每次只能做一件事情，后面的事情需要等待前面的事情完成后才可以进行，但是如果使用多线程程序，就会发生两个线程抢占资源的问题。例如，两个人同时说话、两个人同时过同一个独木桥等。所以在多线程编程中，需要防止这些资源访问的冲突。C# 提供线程同步机制来防止资源访问的冲突，其中，主要用到 lock 关键字、Monitor 类和 Mutex 类。本节将对线程的同步机制进行详细讲解。

15.4.1　线程同步机制

在实际开发中，使用多线程程序的情况很多，如银行排号系统、火车站售票系统等。多线程程序通常会发生问题，以火车站售票系统为例，在代码中判断当前票数是否大于 0，如果大于 0，则执行把火车票出售给乘客的操作。但当两个线程同时访问这段代码时（假

如这时只剩下一张票），第一个线程将票售出，与此同时第二个线程也已经执行并完成判断是否有票的操作，并得出票数大于 0 的结论，于是它也执行将票售出的操作，这样票数就会产生负数。所以在编写多线程程序时，应该考虑到线程安全问题。实质上，线程安全问题来源于两个线程同时存取单一对象的数据。

例如，在项目中未考虑到线程安全问题的基础上，模拟火车站售票系统的功能。主要代码如下。

```
01 class Program
02 {
03     int num = 10;                                  // 设置当前总票数
04     void Ticket()
05     {
06         while (true)                               // 设置无限循环
07         {
08             if (num > 0)                           // 判断当前票数是否大于 0
09             {
10                 Thread.Sleep(100);                 // 使当前线程休眠 100 毫秒
11                 // 票数减 1
12                  Console.WriteLine(Thread.CurrentThread.Name + "---- 票数 "
+ num--);
13             }
14         }
15     }
16     static void Main(string[] args)
17     {
18         Program p = new Program();                 // 创建对象，以便调用对象方法
19         // 分别实例化 4 个线程，并设置名称
20         Thread tA = new Thread(new ThreadStart(p.Ticket));
21         tA.Name = " 线程一 ";
22         Thread tB = new Thread(new ThreadStart(p.Ticket));
23         tB.Name = " 线程二 ";
24         Thread tC = new Thread(new ThreadStart(p.Ticket));
25         tC.Name = " 线程三 ";
26         Thread tD = new Thread(new ThreadStart(p.Ticket));
27         tD.Name = " 线程四 ";
28         tA.Start();                                // 分别启动线程
29         tB.Start();
30         tC.Start();
31         tD.Start();
32         Console.ReadLine();
33     }
34 }
```

运行上面代码，得到如图 15.15 所示的内容。

图 15.15　打印后剩下的票数为负值

从图 15.15 中可以看出，打印后剩下的票数为负值，这样就出现了问题。这是由于同时创建了 4 个线程，这 4 个线程同时执行，在 num 变量为 1 时，线程一、线程二、线程三、线程四都对 num 变量有存储功能。当线程一执行时，还没有来得及进行递减操作，就指定它调用 Sleep 方法进入就绪状态，这时线程二、线程三和线程四也都开始执行，发现 num 变量依然大于 0，但此时线程一休眠时间已到，将 num 变量值递减，同时线程二、线程三、线程四也都对 num 变量进行递减操作，从而产生了负值。

那么该如何解决资源共享的问题呢？基本上所有解决多线程资源冲突问题的方法都是采用给定时间只允许一个线程访问共享资源，这时就需要给共享资源上一道锁。这就好比一个人上洗手间时，他进入洗手间后会将门锁上，出来时再将锁打开，然后其他人才可以进入。这就是程序开发中的线程同步。

线程同步是指并发线程高效、有序地访问共享资源所采用的技术。所谓同步，是指某一时刻只有一个线程可以访问资源，只有当资源所有者主动放弃了代码或资源的所有权时，其他线程才可以使用这些资源。

15.4.2　使用 lock 关键字实现线程同步

lock 关键字可以用来确保代码块完成运行，而不会被其他线程中断，它是通过在代码块运行期间为给定对象获取互斥锁来实现的。

lock 语句以 lock 关键字开头，它有一个作为参数的对象，在该参数的后面还有一个一次只能有一个线程执行的代码块。lock 语句的语法格式如下。

```
Object thisLock = new Object();
lock (thisLock)
{
    // 要运行的代码块
}
```

提供给 lock 语句的参数必须为基于引用类型的对象，该对象用来定义互斥锁的范围。

严格来说，提供给 lock 语句的参数只是用来唯一标识由多个线程共享的资源，所以它可以是任意类的实例。实际上，此参数通常表示需要进行线程同步的资源。

示例 4．设置同步块模拟售票系统

修改 15.4.1 节中的代码，使用 lock 关键字锁定售票代码，以便实现线程同步。主要代码如下。

```
01 class Program
02 {
03     int num = 10;                                  // 设置当前总票数
04     void Ticket()
05     {
06         while (true)                               // 设置无限循环
07         {
08             lock (this)                            // 锁定代码块，以便线程同步
09             {
10                 if (num > 0)                       // 判断当前票数是否大于 0
11                 {
12                     Thread.Sleep(100);             // 使当前线程休眠 100 毫秒
13                     // 票数减 1
14                     Console.WriteLine(Thread.CurrentThread.Name + "---- 票数 "
+ num--);
15                 }
16             }
17         }
18     }
19     static void Main(string[] args)
20     {
21         Program p = new Program();                 // 创建对象，以便调用对象方法
22          // 分别实例化 4 个线程，并设置名称
23         Thread tA = new Thread(new ThreadStart(p.Ticket));
24         tA.Name = " 线程一 ";
25         Thread tB = new Thread(new ThreadStart(p.Ticket));
26         tB.Name = " 线程二 ";
27         Thread tC = new Thread(new ThreadStart(p.Ticket));
28         tC.Name = " 线程三 ";
29         Thread tD = new Thread(new ThreadStart(p.Ticket));
30         tD.Name = " 线程四 ";
31         tA.Start();                                // 分别启动线程
32         tB.Start();
33         tC.Start();
34         tD.Start();
35         Console.ReadLine();
36     }
37 }
```

设置同步块模拟售票系统程序运行结果如图 15.16 所示。

图 15.16　设置同步块模拟售票系统程序运行结果

从图 15.16 中可以看出，打印到最后，票数没有出现负数，这是因为将售票代码放置在同步块中。

学习笔记

如果在静态方法中使用 lock 关键字，则不能使用 this 关键字。

15.4.3　使用 Monitor 类实现线程同步

Monitor 类提供了对对象的同步访问机制，它通过向单个线程授予排他锁来控制对对象的访问。排他锁提供限制访问代码块（通常称为临界区）的功能。当一个线程拥有排他锁时，其他任何线程都不能获取该锁。

Monitor 类的主要功能如下。

- 根据需要与某个对象相关联。
- 它是未绑定的，也就是说可以直接从任何上、下文调用它。
- 不能创建 Monitor 类的实例。

Monitor 类的常用方法及说明如表 15.4 所示。

表 15.4　Monitor 类的常用方法及说明

方　法	说　明
Enter	在指定对象上获取排他锁
Exit	释放指定对象上的排他锁
Pulse	通知等待队列中的线程锁定对象状态的更改
PulseAll	通知所有的等待线程对象状态的更改
TryEnter	试图获取指定对象的排他锁
Wait	释放对象上的锁并阻塞当前线程，直到它重新获取该锁

例如，修改示例 4，使用 Monitor 类实现与示例 4 相同的功能，即使用 Monitor 类设置同步块模拟售票系统，主要代码如下（主要改动是下面加粗部分的代码）。

```
01 class Program
02 {
03     int num = 10;                                  // 设置当前总票数
04     void Ticket()
05     {
06         while (true)                               // 设置无限循环
07         {
08             Monitor.Enter(this);                   // 锁定代码块
09             if (num > 0)                           // 判断当前票数是否大于 0
10             {
11                 Thread.Sleep(100);                 // 使当前线程休眠 100 毫秒
12                    // 票数减 1
13                 Console.WriteLine(Thread.CurrentThread.Name + "---- 票数 " + num--);
14             }
15             Monitor.Exit(this);                    // 解锁代码块
16         }
17     }
18     static void Main(string[] args)
19     {
20         Program p = new Program();                 // 创建对象，以便调用对象方法
21          // 分别实例化 4 个线程，并设置名称
22         Thread tA = new Thread(new ThreadStart(p.Ticket));
23         tA.Name = " 线程一 ";
24         Thread tB = new Thread(new ThreadStart(p.Ticket));
25         tB.Name = " 线程二 ";
26         Thread tC = new Thread(new ThreadStart(p.Ticket));
27         tC.Name = " 线程三 ";
28         Thread tD = new Thread(new ThreadStart(p.Ticket));
29         tD.Name = " 线程四 ";
30         tA.Start();                                // 分别启动线程
31         tB.Start();
32         tC.Start();
33         tD.Start();
34         Console.ReadLine();
35     }
36 }
```

学习笔记

使用 Monitor 类可以实现很好的控制功能。例如，它可以使用 Wait 方法指示活动的线程等待一段时间，当线程完成操作后，还可以使用 Pulse 方法或 PulseAll 方法通知等待中的线程。

15.4.4　使用 Mutex 类实现线程同步

实现线程同步还可以使用 Mutex 类，该类是同步基元，它只向一个线程授予对共享资源的独占访问权。如果一个线程获取了互斥体，则要获取该互斥体的第二个线程将被挂起，直到第一个线程释放该互斥体。Mutex 类与 Monitor 类相似，它防止多个线程在某一时间同时执行某个代码块，然而与 Monitor 类不同的是，Mutex 类可以用来使跨进程的线程同步。

Mutex 类的常用方法及说明如表 15.5 所示。

表 15.5　Mutex 类的常用方法及说明

方　法	说　明
Close	当在派生类中被重写时，释放由当前 WaitHandle 持有的所有资源
ReleaseMutex	释放 Mutex 一次
WaitAll	等待指定数组中的所有元素都收到信号
WaitAny	等待指定数组中的任一元素收到信号
WaitOne	当在派生类中被重写时，阻塞当前线程，直到当前的 WaitHandle 收到信号

在使用 Mutex 类实现线程同步时，首先需要实例化一个 Mutex 对象，其构造函数语法如下。

```
public Mutex(bool initallyOwned)
```

initallyOwned：指示创建该对象的线程是否希望立即获得其所有权。当在一个资源得到保护的类中创建 Mutex 对象时，通常将该参数设置为 false。

例如，修改示例 4，使用 Mutex 类实现与示例 4 相同的功能，即使用 Mutex 类设置同步块模拟售票系统，主要代码如下（主要改动是下面加粗部分的代码）。

```
01  class Program
02  {
03      int num = 10;                                    // 设置当前总票数
04      void Ticket()
05      {
06          while (true)                                 // 设置无限循环
07          {
08              Mutex myMutex = new Mutex(this);         // 创建 Mutex 对象
09              myMutex.WaitOne();                       // 阻塞当前线程
10              if (num > 0)                             // 判断当前票数是否大于 0
11              {
12                  Thread.Sleep(100);                   // 使当前线程休眠 100 毫秒
13                  // 票数减 1
14                   Console.WriteLine(Thread.CurrentThread.Name + "---- 票数 "
```

```
+ num--);
15              }
16              myMutex.ReleaseMutex()          // 释放 Mutex 对象
17          }
18      }
19      static void Main(string[] args)
20      {
21          Program p = new Program();          // 创建对象，以便调用对象方法
22          // 分别实例化 4 个线程，并设置名称
23          Thread tA = new Thread(new ThreadStart(p.Ticket));
23          tA.Name = "线程一";
24          Thread tB = new Thread(new ThreadStart(p.Ticket));
25          tB.Name = "线程二";
26          Thread tC = new Thread(new ThreadStart(p.Ticket));
27          tC.Name = "线程三";
28          Thread tD = new Thread(new ThreadStart(p.Ticket));
29          tD.Name = "线程四";
30          tA.Start();                         // 分别启动线程
31          tB.Start();
32          tC.Start();
33          tD.Start();
34          Console.ReadLine();
35      }
36  }
```

学习笔记

尽管 Mutex 类可以用于进程内的线程同步，但是使用 Monitor 类通常更为合适，因为 Monitor 类是专门为 .NET Framework 而设计的，因而它可以更好地利用资源。相比之下，Mutex 类是 Win32 构造的包装。尽管 Mutex 类比 Monitor 类的功能更为强大，但是相对于 Monitor 类，它所需要的互操作转换更消耗计算资源。